程序设计 · 算法 · 实践

# 深入浅出
# 程序设计竞赛

## 进阶篇

教材整理排印本

A4 · 算法与数据结构

# 目录


## 第1部分：进阶技巧与思想 6

第1章 常见优化技巧 6

1.1 双指针法 7
1.2 空间换时间 10
1.3 单调栈 15
1.4 单调队列 19
1.5 课后习题与实验 24

第2章 前缀和、差分与离散化 26

2.1 前缀和与二维前缀和 26
2.2 差分 33
2.3 离散化 37
2.4 课后习题与实验 44

第3章 分治与倍增 47

3.1 归并排序与快速幂 47
3.2 其他序列上的分治 53
3.3 倍增与ST表 55
3.4 课后习题与实验 59

第4章 进阶搜索 62

4.1 状态剪枝 62
4.2 迭代加深与双向搜索 67
4.3 启发式搜索 74
4.4 课后习题与实验 78

## 第2部分：进阶数据结构 81

第5章 二叉堆与树状数组 81

5.1 二叉堆的实现 82
5.2 优先队列 88
5.3 树状数组 94
5.4 课后习题与实验 100

第6章 线段树 103

6.1 线段树的建立与操作 103
6.2 线段树的应用 112
6.3 权值线段树（选读） 116
6.4 课后习题与实验 118

第7章 字符串 120

7.1 KMP算法 120
7.2 字典树 125
7.3 课后习题与实验 132

第8章 线段树的进阶用法 135
8.1 可持久化线段树 135
8.2 树状数组套权值线段树 140
8.3 课后习题与实验 146

第3部分：图论 149

第9章 树 149
9.1 树的性质与遍历 150
9.2 树的直径与重心 154
9.3 最近公共祖先 162
9.4 树的重链剖分 169
9.5 课后习题与实验 178

第10章 最短路 181
10.1 单源最短路径 181
10.2 负环与差分约束 190
10.3 次短路 194
10.4 多源最短路径与传递闭包 198
10.5 课后习题与实验 201

第11章 最小生成树 205
11.1 最小生成树及其应用 205
11.2 次小生成树（选读） 216
11.3 课后习题与实验 219

第12章 连通性问题 221
12.1 无向图的边双连通性 221
12.2 无向图的点双连通性 227
12.3 有向图的强连通性 232
12.4 课后习题与实验 240

第4部分：动态规划 243

第13章 动态规划的引入 243
13.1 动态规划 244
13.2 状态、转移与有向无环图 249
13.3 拓扑排序与记忆化搜索 255
13.4 动态规划的解法 258
13.5 课后习题与实验 261

第14章 线性状态动态规划 263
14.1 单序列问题 263
14.2 多序列问题 270
14.3 高维问题 274
14.4 课后习题与实验 277

第15章 区间与环形动态规划 279
15.1 区间动态规划 279
15.2 环形动态规划 285
15.3 课后习题与实验 287

第16章 树与图上的动态规划 289
16.1 树上动态规划 289
16.2 图上动态规划 293
16.3 课后习题与实验 300

第17章 状态压缩动态规划 303
17.1 状态压缩 304
17.2 状态压缩的一般技巧 306
17.3 状态压缩与其他算法的交叉应用 314
17.4 课后习题与实验 317

第18章 动态规划的设计与优化 321
18.1 设计动态规划 321
18.2 优化转移 323
18.3 课后习题与实验 337

第5部分：数学专题选讲 341

第19章 进阶数论 341
19.1 模意义下的数和运算 342
19.2 乘法逆元 347
19.3 同余方程与中国剩余定理 351
19.4 线性筛与积性函数（选读） 353
19.5 欧拉函数（选读） 360
19.6 课后习题与实验 364

第20章 组合数学与计数 367
20.1 集合与容斥原理 367
20.2 常见的组合计数方法 375
20.3 二项式定理 378
20.4 卡特兰数与斯特林数 381
20.5 课后习题与实验 384

第21章 概率与统计 386
21.1 事件 387
21.2 概率模型 390
21.3 数学期望 395
21.4 课后习题与实验 399

第22章 基础线性代数 402
22.1 向量与矩阵变换 402
22.2 高斯消元 409
22.3 课后习题与实验 414

附录 417

附录 A 调试与对拍 417
A.1 IDE调试 417
A.2 输出中间变量调试 421
A.3 GDB调试 422
A.4 对拍 425

A.5 使用Python制造测试数据 428

附录 B 命令行与脚本 429

B.1 命令行的使用 429

B.2 使用命令行完成算法竞赛题目 433

附录 C 常数优化技巧 439

C.1 读写优化 440

C.2 缓存优化 443

C.3 编译优化 447

C.4 算法常数优化 453

C.5 实践出真知 456

# 第1部分：进阶技巧与思想

## 第1章 常见优化技巧

在《深入浅出程序设计竞赛（基础篇）》（以下简称《基础篇》）中，大家已经初窥算法的乐趣；而在本书中，我们会继续发现和感悟更多的算法奥秘。

评价算法的标准，除了正确性以外，最重要的就是程序的运行效率是否足够高，是否可以在限定的时间内处理完指定的数据。在《基础篇》的“暴力枚举”一章中介绍了可以通过剔除无效操作来提升算法效率。除此之外，还可以用“单调性”和“空间换时间”来优化时间复杂度。将这两种思想结合，就产生了**单调栈**和**单调队列**等工具。

这些技巧可以帮助我们排除无效选项，在一定范围内保持单调性，常常可以将一些问题求解的时间复杂度降到 $O(n)$，因此称为线性时间复杂度优化。本章思维导图如图 1-1 所示。




图1-1 本章思维导图

# 1.1 双指针法

### 例 1-1 A-B数对(洛谷P1102)

给出一个数列以及一个数字 $C$，要求计算出所有 $A-B=C$ 的数对的个数（$A$ 和 $B$ 都取自这个数列，不同位置的数字一样的数对算不同的数对）。数字个数不超过 $200000$，数列值域和 $C$ 的值域不超过 $2^{31}-1$。

**分析：**在《基础篇》中介绍了“枚举数字 $B$，使用二分的方式查找 $A=B+C$ 这个数是否存在”的做法。将这些数字排序后，使这个数列具有单调性（有序性），就能以 $O(\log n)$ 的时间复杂度查找到指定的数字。单调性是一种很好用的规律，这里介绍另外一种利用单调性解决问题的方法。

假设 $C=3$，数列是 $2,5,3,7,5,2,4,8$。将数列排序得到 $2,2,3,4,5,5,7,8$。对于序列中的一个数 $s[i]$，想要快速找到 $s[i]-C$。如果 $s[i]-C$ 这个数字存在，则一定在排序后的数列中是连续出现的。

维护两个指针，左指针 $l$ 和右指针 $r$，使得 $s[l]$ 是首个大于或等于 $s[i]-C$ 的数 ，$s[r]$ 是首个大于 $s[i]-C$ 的数。这样，从 $s[l]$ 到 $s[r-1]$ 都是等于 $s[i]-C$ 的数字，则等于 $s[i]-C$ 的数字个数有 $r-l$ 个，累加进答案中。

$s[i]-C$ 是越来越大的，所以 $l$ 和 $r$ 都会越来越往右。具体步骤如图 1-2 所示。

每次增加 l 和 r，使 s[l] 为首个 ≥ s[i] − C 的数，s[r] 为首个 > s[i] − C 的数。

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
|---|---|---|---|---|---|---|---|---|---|
| 排序后的 s[i] | 2 | 2 | 3 | 4 | 5 | 5 | 7 | 8 | |
| 处理 s[0] | **2** | 2 | 3 | 4 | 5 | 5 | 7 | 8 | l = r = 0 |
| 处理 s[1] | 2 | **2** | 3 | 4 | 5 | 5 | 7 | 8 | l = r = 0 |
| 处理 s[2] | 2 | 2 | **3** | 4 | 5 | 5 | 7 | 8 | l = r = 0 |
| 处理 s[3] | 2 | 2 | 3 | **4** | 5 | 5 | 7 | 8 | l = r = 0 |
| 处理 s[4] | 2 | 2 | 3 | 4 | **5** | 5 | 7 | 8 | l = 0，r = 2；答案增加 r − l = 2 |
| 处理 s[5] | 2 | 2 | 3 | 4 | 5 | **5** | 7 | 8 | l = 0，r = 2；答案增加 r − l = 2 |
| 处理 s[6] | 2 | 2 | 3 | 4 | 5 | 5 | **7** | 8 | l = 3，r = 4；答案增加 r − l = 1 |
| 处理 s[7] | 2 | 2 | 3 | 4 | 5 | 5 | 7 | **8** | l = 4，r = 6；答案增加 r − l = 2 |

图1-2 双指针法步骤

这种使用两个指针维护①一段序列区间的方法被称为**双指针法**（又称为尺取法）。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
#define MAXN 200010
int s[MAXN];
int main() {
    int n, c;
    cin >> n >> c;
    for (int i = 0; i < n; i++)
        cin >> s[i];
    sort(s, s + n);
    int l = 0, r = 0;
    long long sum = 0;
    for (int i = 0; i < n; i++) {
        while (l < n && s[l] < s[i] - c)
            l++;
        while (r < n && s[r] <= s[i] - c)
            r++;
        if (l < n && s[i] - s[l] == c) {
            sum += r - l;
        }
    }
    cout << sum << endl;
    return 0;
}
```

> **注：** 1. “维护”的意思是使用某些数据结构记录一些信息，使其符合一定的性质，并且对这些信息进行增加、删除、查询、修改等操作。

虽然for循环里面套着while循环，但是无论是左指针还是右指针，都只是从最左边移动到了最右边，所以双指针法本身的时间复杂度是 $O(n)$。当然不能忽略掉排序本身的算法复杂度 $O(n\log n)$。

### 例 1-2 逛画展(洛谷P1638)

博览馆正在展出 $n(n\leqslant 10^6)$ 幅由 $m$ 位画家所画的画。第 $i$ 幅画的画家编号是 $a_i(1\leqslant a_i\leqslant m)$。

游客在购买门票时必须说明两个数字 $x$ 和 $y$，代表他要看展览中的第 $x$ 幅至第 $y$ 幅画（包含 $x$，$y$）之间的所有画，而门票的价钱就是每看一张画就要花费一元。

小 $S$ 希望入场后可以看到所有画家的画。也就是说，他想从第 $x$ 幅画看到第 $y$ 幅画，这些画的作者覆盖了从 $1$ 号到 $m$ 号的所有画家。但他同时也想花尽可能少的钱。

求出他购买门票时应选择的 $x$ 和 $y$，数据保证一定有解。若存在多组解，输出 $x$ 最小的那组。

**输入样例：**

```
12 5
253132411543
```

**输出样例：**

```
27
```

**分析：**由于需要得到一个区间的两个端点，这个区间出现的数字是 1 到 $m$ 至少各出现一次，那么可以使用**双指针法**维护符合要求的区间。

根据个人习惯，左指针和右指针表示为左闭右开或者左闭右闭区间都是可以的。这里使用左闭右开的方式，也就是区间表示为[$l$，$r$），即参观的画的作者为 $a_l, a_{l+1}, \cdots, a_{r-10}$

考虑贪心策略：当左指针固定时，将右指针向右延伸，直到区间内凑齐 1 到 $m$ 的数字为止。可以使用数组 `sum` 来记录区间内每位画家的画的数量，用变量 `num` 记录区间内不同画家的人数。

如果区间内没有全部的画家，那么需要向右延伸，将新加入画家编号 `a[r-1]` 的数量，也就是 `sum[a[r-1]]++`。如果原来的 `sum[a[r-1]]` 是 0，说明这位画家的画是在区间里第一次看到，那么 `num` 增加 1。此时 $r-l$ 就是区间的长度，打擂台记录最小值和区间端点位置。

如果可以看到所有画家的画，尝试去除区间最左边的画，也就是 `sum[a[l]]--`（如果去掉了这位画家的最后一幅，别忘记 `num` 要减去 1）。然后尝试向右移动左指针，继续判断是否符合要求，还是需要继续将右指针往右移动获得更多的画。代码如下：

```
#include <iostream>
using namespace std;
const int MAXN = 1000010;
int n, m, sum[2005], a[MAXN];
int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; ++i)
        cin >> a[i];
    int l = 1, num = 0, ans = MAXN, ansl = 1, ansr = n;
    for (int r = 1; r <= n; ++r) {
        if (++sum[a[r]] == 1)
            ++num;
        while (num == m) {
            if (r - l + 1 < ans) {
                ans = r - l + 1;
                ansl = l;
                ansr = r;
            }
            if (--sum[a[l]] == 0)
                --num;
            ++l;
        }
    }
    cout << ansl << " " << ansr;
    return 0;
}
```

双指针法本质上是使用队列维护一个符合条件的区间。右指针增加相当于入队，左指针增加相当于出队。在《基础篇》的“算法评价与复杂度”一章中介绍的例题就使用了双指针法。

### 例 1-3 最大子段和(洛谷P1115)

给出一个长度为 $n$ 的序列 $\{a_i\}$，选出其中连续且非空的一段使得这段和最大。

**数据范围：** $1 \leqslant n \leqslant 2 \times 10^5, -10^4 \leqslant a_i \leqslant 10^4$。

**分析：**如果使用暴力算法，枚举区间的两点，然后累加区间的每一个数字，其算法时间复杂度是 $O(n^3)$。如果使用**前缀和**优化（见本书 第2章“前缀和、差分与**离散化**”）使计算一个区间元素之和的时间复杂度降为 $O(1)$，总的时间复杂度降为 $O(n^2)$，但依然难以通过本题。

令 $f_i$ 为序列 $\{a_i\}$ 中，以第 $i$ 个元素为右端点的全部区间中可以取到的最大区间和。在处理第 $i$ 个元素时，可以决定要不要把第 $i-1$ 个元素为右端点对应的区间最大值加上去。如果加上后的和还不如直接只取第 $i$ 个值，就不需要加上，即 $f_i=\max(f_{i-1}+a_i,a_i)$。

由于每一个 $f_i$ 都有可能是答案，所以还需要同时维护 $f_i$ 的最大值。当然，题目要求非空序列，所以至少也得选一个数字。如果求解的最大值是负数，则说明所有的 $a_i$ 都是负数，在统计最大值的变量时应当设置为可能的值域范围的最小值。

由于循环处理 $f_i$ 的时候只会用到 $f_{i-1}$ 的值，所以不需要定义 $f$ 数组，直接使用一个变量来维护即可。代码如下：

```
#include <iostream>
#include <algorithm>
using namespace std;
int main() {
    int n, a, f = 0, ans = -10000;
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> a;
        f = max(f + a, a);
        ans = max(f, ans);
    }
    cout << ans;
}
```

本题还有一种使用**分治**的解决方法，可以参阅本书 第3章“分治与**倍增**”。请读者思考，如果要求输出这个区间的左右端点，需要怎么做？

一些读者刚开始可能没有注意到序列的值域范围，认为“不是直接全部加上就最大了吗?”实际上，序列的值可能是负数。大家一定要认真阅读题面，包括数据范围，也许可以带来一些提示。

## 1.2 空间换时间

计算机提供了相当大的内存，可以巧用内存空间存放中间数据，使用一些技巧提升处理数据的效率。

### 例 1-4 直播获奖(洛谷P7072,CSP-J2020)

某场比赛有 $n$ 名选手参加。主办方打算逐一评出每个选手的成绩（$0$ 到 $600$ 分），并直播即时的获奖分数线。若当前已评出了 $p$ 位选手的成绩，则当前计划获奖人数为 $\max(1,\lfloor p\times w\%\rfloor)$，其中 $w$ 是获奖百分比，$\lfloor x\rfloor$ 表示对 $x$ 向下取整，$\max(x,y)$ 表示 $x$ 和 $y$ 中较大的数。

如有选手成绩相同，则所有成绩并列的选手都能获奖，因此实际获奖人数可能比计划中多。对于每一个输入的成绩，请计算即时的获奖分数线。

**数据范围：**对于 $50\%$ 的测试点，$n\leqslant 2000$；对于 $85\%$ 的测试点，$n\leqslant 10^4$；对于所有的测试点，$n\leqslant 10^5$

**分析：**最直观的做法是，每评测出一个人的成绩就将所有人排序一次，然后输出对应排名。一共有 $n$ 个人，共排序 $n$ 次。如果使用快速排序（或者是 `sort()` 等函数），总的时间复杂度是 $O(n^2 \log n)$，只能获得 50 分；如果使用插入排序，将新得到的分数插入到有序的分数数组中，总的时间复杂度是 $O(n^2)$，可以获得 85 分。

发现需要排序的数值不超过 600，因此考虑其他的排序方法。使用计数排序，每次评出一个人的成绩，就加入对应的“桶”中，然后再从高往低统计累加人数，达到获奖人数时输出分数线。这种做法的时间复杂度是 $O(An)$，其中 $A$ 是指成绩的值域上限，可以通过全部测试点。代码如下：

```
#include <iostream>
using namespace std;
int t[610], n, w;
int main() {
    int tmp;
    cin >> n >> w;
    for (int i = 1; i <= n; i++) {
        cin >> tmp;
        t[tmp]++;    // 加入对应分数的桶
        int sum = 0; // 累加人数
        for (int j = 600; j >= 0; j--) {
            sum += t[j];
            if (sum >= max(1, i * w / 100)) {
                cout << j << " ";
                break;
            }
        }
    }
    return 0;
}
```

解决同一种问题可能有不同的算法，其算法复杂度和某些变量的大小有关，这时应当选择合适的算法使得效率最高。例如，对一个数列进行排序，如果这个数列的值域（假设是 $A$）比较小而数量（假设有 $n$ 个数）较多，可以使用时间复杂度是 $O(n+A)$ 的计数排序；如果值域很大而数量没那么多，则可以使用时间复杂度为 $O(n \log n)$ 的快速排序。

### 例 1-5 求和(洛谷P2671,NOIP2015普及组)

一条狭长的纸带被均匀划分出了 $n$ 个格子，格子编号从 1 到 $n$。每个格子上都染了一种颜色 $b_i$（用 $[1,m]$当中的一个整数表示），并且写了一个数字 $a_i$ 如图 1-3 所示。

| 编号 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 颜色 | 1 | 1 | 2 | 2 | 1 | 2 |
| 数字 | 5 | 5 | 3 | 3 | 3 | 3 |

图1-3 求和例题图

定义一种特殊的三元组：$(x,y,z)$，其中 $x$，$y$，$z$ 都代表纸带上格子的编号，该三元组要求满足以下两个条件：

1. $x,y,z$ 是整数 ,$x<y<z,y-x=z-y$

2. $b_x=b_{z0}$

满足上述条件的三元组的“分数”规定为 $(x+z)\times(a_x+a_z)$。整个纸带的“分数”规定为所有满足条件的三元组的“分数”的和。输出整个纸带的“分数”除以 $10007$ 所得的余数即可。

**数据范围：** $1\leqslant n\leqslant 10^5,1\leqslant m\leqslant 10^5,1\leqslant a_i\leqslant m,1\leqslant b_i\leqslant 10^5\leqslant$

**分析：**题目要求有几个三元组 $(x,y,z)$ 符合要求。最简单的思路就是使用三重循环依次枚举 $x,y,z$，然后验证是否符合要求。这样的时间复杂度是 $O(n^3)$，效率很低，无法在限定时间内完成计算。

根据第 1 个条件，$y-x=z-y$，所以 $x+z=2y$，即 $y=(x+z)/2$。固定 $x,z$ 后，$y$ 被唯一确定；只需满足 $z-x$ 为正偶数，也就是 $x,z$ 奇偶性相同。若两者颜色也相同，就恰好对应一个合法三元组 $(x,y,z)$。因此可以将奇数编号与偶数编号分开处理，枚举 $x,z$ 并检查颜色，得到 $O(n^2)$ 的做法，但仍会超时。

将题目中给出计算“分数”的式子拆开，得到 $(x+z)\times(a_x+a_z)=xa_x+xa_z+za_x+za_z$，这是每对 $(x,z)$ 对答案的贡献①。如何快速计算所有 $(x,z)$ 对的贡献总和？

不妨简化问题，对于一个数列 $a_1,a_2,a_3,a_4,a_5$，每一对满足 $1\leqslant x<z\leqslant 5$ 的 $(a_x,a_z)$，对答案都有 $xa_x+xa_z+za_x+za_z$ 的贡献。那么，贡献的总和是多少呢?对所有情况列出表格，见表 1-1。

| i \ j | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | | $1a_1+1a_2+2a_1+2a_2$ | $1a_1+1a_3+3a_1+3a_3$ | $1a_1+1a_4+4a_1+4a_4$ | $1a_1+1a_5+5a_1+5a_5$ |
| 2 | | | $2a_2+2a_3+3a_2+3a_3$ | $2a_2+2a_4+4a_2+4a_4$ | $2a_2+2a_5+5a_2+5a_5$ |
| 3 | | | | $3a_3+3a_4+4a_3+4a_4$ | $3a_3+3a_5+5a_3+5a_5$ |
| 4 | | | | | $4a_4+4a_5+5a_4+5a_5$ |
| 5 | | | | | |

表1-1 例1-5中所有情况列表

> **注：** 1. 对答案的贡献为 $x$（$x$ 为一个数字），即在最终求解的答案需要加上 $x$。

考查结果中所有系数是“1”的项，发现有 4 个 $1a_1$，此外还有 $1(a_2+a_3+a_4+a_5)$；考察结果中所有系数是“2”的项，发现有 4 个 $2a_2$，此外还有 $2(a_1+a_3+a_4+a_5)$ ……依此类推，可以归纳为：对于所有系数为 $i$ 的项，有 $5-1$ 个 $ia_i$ 和 $i\left(\sum_{j=1}^{S}a_j-a_i\right)$，注意需要扣掉自己本身的一次。

因此，对于同一颜色同一奇偶性的一类格子，都可以用这种方式计算。

使用s1 数组记录某种颜色且编号为奇数或偶数格子的数目，用s2 数组记录某种颜色且编号为奇数或偶数格子的数字之和。那么对于每一个 $i$，对答案的贡献就是 $\mathrm{i}*((\mathrm{s2[y][i\%2]}-\mathrm{a[i]})+\mathrm{a[i]}*(\mathrm{s1[y][i\%2]}-1)$ 。

其中 $y$ 为第 $i$ 个格子的颜色，$i$%2 是指 $i$ 的奇偶性。根据以上思路，可以完成如下代码：

```
#include <iostream>
using namespace std;
const int mod = 10007;
const int maxn = 1e5 + 10;
int n, m;
int a[maxn], b[maxn];
int s1[maxn][2], s2[maxn][2];
int ans;
int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
        cin >> a[i];
    for (int i = 1; i <= n; i++) {
        cin >> b[i];
        s1[b[i]][i % 2]++;
        // 记录颜色为b[i],且编号与编号i奇偶性相同的格子的数目
        s2[b[i]][i % 2] = (s2[b[i]][i % 2] + a[i]) % mod;
        // 记录颜色为b[i],且编号与编号i奇偶性相同的格子上面的数字的和
    }
    for (int i = 1; i <= n; i++) {
        int y = b[i];
        ans += i * (s2[y][i % 2] + a[i] * (s1[y][i % 2] - 2) % mod) % mod;
        // 多次取模的目的是在计算的过程中取余数而不会运算溢出。减法取模要小心,避免产
        //  生负数
        ans %= mod;
    }
    cout << ans;
    return 0;
}
```

对于一个难以直接处理的式子，可以将式子拆开进行处理，再将一些部分进行合并计算就可以获得更低复杂度的算法。这是一种程序设计竞赛中的常用策略。

### 例 1-6 玉蟾宫(洛谷 P4147,By lydrainbowcat)

有一个 $N\times M(1\leqslant N,M\leqslant 1000)$ 的矩阵，每个格子里写着“$R$”或者“$F$”。请找出其中的一个子矩阵，其元素均为“$F$”并且面积最大。输出它的面积乘以 $3$ 的值。

**数据范围：** $1\leqslant N,M\leqslant 1000$

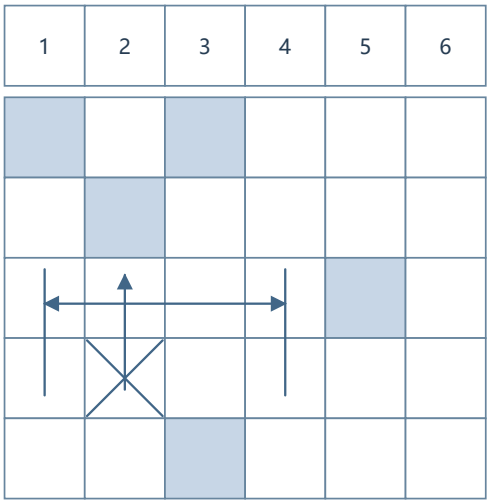
**输入样例：**

```
5 6
RFRFFF
FRFFFF
FFFFRF
FFFFFF
F FRFFF
```

**输出样例：**

**分析：**将矩阵中字母为“$R$”的格子称为“障碍格”，将字母为“$F$”的格子称为“非障碍格”。对于一个非障碍格，可以延伸出一条一直往上延伸、碰到障碍格子或上边界则停止的“线”，这条线往往被称为“悬线”。最大面积矩阵的高一定是一条悬线，否则这个矩阵还可以更高一些。处理每一个非障碍

格时，找到它的悬线长度，并且计算这条悬线往左和往右分别最远可以到达的位置，然后计算面积并比较最大值。计算第 4 行第 2 列的格子时，求面积用到的悬线如图 1-4(a) 所示。




(a)

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| | l=2<br>r=2 | | l=4<br>r=6 | l=4<br>r=6 | l=4<br>r=6 |
| l=1<br>r=1 | | l=3<br>r=6 | l=3<br>r=6 | l=3<br>r=6 | l=3<br>r=6 |
| l=1<br>r=4 | l=1<br>r=4 | l=1<br>r=4 | l=1<br>r=4 | | l=6<br>r=6 |
| l=1<br>r=6 | l=1<br>r=6 | l=1<br>r=6 | l=1<br>r=6 | l=1<br>r=6 | l=1<br>r=6 |
| l=1<br>r=2 | l=1<br>r=2 | | l=4<br>r=6 | l=4<br>r=6 | l=4<br>r=6 |

(b)

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| | l=2<br>r=2 | | l=4<br>r=6 | l=4<br>r=6 | l=4<br>r=6 |
| l=1<br>r=1 | | l=3<br>r=6 | l=4<br>r=6 | l=4<br>r=6 | l=4<br>r=6 |
| l=1<br>r=1 | l=1<br>r=4 | l=3<br>r=4 | l=4<br>r=4 | | l=6<br>r=6 |
| l=1<br>r=1 | l=1<br>r=4 | l=3<br>r=4 | l=4<br>r=4 | l=1<br>r=6 | l=6<br>r=6 |
| l=1<br>r=1 | l=1<br>r=2 | | l=4<br>r=4 | l=4<br>r=6 | l=6<br>r=6 |

(c)

图1-4 悬线法步骤

需要准备以下数据进行预处理①，预处理后得到的 $l$ 数组和 $r$ 数组，如图 1-4(b) 所示。

1. $h$ 数组：悬线高度。初始值设置为 1。如果同一列上一行的格子非障碍，则 $h$ 值是上方格子的 $h$ 值加 1。
2. $l$ 数组：这个格子往左可以找到的最远的非障碍格的列坐标。首先将其设置为本身的列坐标，然后对于每一行，从左往右循环，如果同一行左边的格子非障碍，则 $l$ 值是左边格子的 $l$ 值。

> **注：** 1. 预处理是指在求解统计最终答案之前先对输入的数据进行处理，方便最终求解。

3. $r$ 数组：这个格子往右可以找到的最远的非障碍格的列坐标。和 $l$ 数组的求法类似，只是从右往左处理。

接下来，对每一个非障碍格计算它的悬线可以往左和往右的最远地方。当这个非障碍格的上方也是非障碍格，如果上方格子的 $l$ 值更大，则更新为上方格子的 $l$ 值；如果上方格子的 $r$ 值更小，则更新为上方格子的 $r$ 值。最后这个格子的悬线往左或者往右最远能到的地方就是 $l$ 和 $r$，面积就很容易得到了，如图 1-4(c) 所示。具体代码如下：

```
#include <iostream>
using namespace std;
#define MAXN 1010
int n, m, ans = 0;
char a[MAXN][MAXN];
int h[MAXN][MAXN], l[MAXN][MAXN], r[MAXN][MAXN];
int main() {
    ans = 0;
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            cin >> a[i][j];
            h[i][j] = 1, r[i][j] = l[i][j] = j;
        }
        for (int j = 2; j <= m; j++)
            if (a[i][j] == 'F' && a[i][j - 1] == 'F')
                l[i][j] = l[i][j - 1];
        for (int j = m - 1; j > 0; j--)
            if (a[i][j] == 'F' && a[i][j + 1] == 'F')
                r[i][j] = r[i][j + 1];
    }
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++) {
            if (a[i][j] == 'F') {
                if (i > 1 && a[i - 1][j] == 'F') {
                    h[i][j] = h[i - 1][j] + 1;
                    if (l[i - 1][j] > l[i][j])
                        l[i][j] = l[i - 1][j];
                    if (r[i - 1][j] < r[i][j])
                        r[i][j] = r[i - 1][j];
                }
                if ((r[i][j] - l[i][j] + 1) * h[i][j] > ans)
                    ans = (r[i][j] - l[i][j] + 1) * h[i][j];
            }
        }
    cout << 3 * ans;
    return 0;
}
```

## 1.3 单调栈

**单调栈**是一种内部元素具有单调性的栈，可以解决求“以某个值为最值的最大区间”等问题。

### 例 1-7 糟糕的一天(洛谷P2866,USACO2006月赛)

有 $N(1 \leqslant N \leqslant 80000)$ 头牛，第 $i$ 头牛的身高为 $h_i(1 \leqslant h_i \leqslant 10^9)$。每头牛往右边看，可以看到严格小于它身高的牛的头顶，直到看到了身高不小于它的牛（看不到这头牛的头顶），或者右边没有其他牛为止。第 $i$ 头牛可以看到的头顶数量为 $C_i$，求 $\sum_{i=1}^{n} C_i$

假设有 $6$ 头牛，高度分别为 $10,3,7,4,12,2$，那么它们分别可以看到 $3,0,1,0,1,0$ 头牛的头顶，总和为 $5$。

**分析：**如果使用暴力枚举求解，对于每一头牛都往右边枚举身高小于它的牛，那么时间复杂度为 $O(n^2)$，无法通过本题。

转变思路，与其求一头牛能看见几头牛，不如反过来，计算一头牛能被几头牛看见，因为它们的总和是相等的。一头牛能被哪些其他牛看到呢？左边比它高的牛，然后再左边更高的牛…只需要维护一个数组，满足这个数组里面存的都是对于第 $i$ 头牛来说，身高依次递增的牛的身高（这个数组是单调递减的）。可以使用栈来进行维护。

最开始栈是空的。身高为 10 的牛进栈，原来栈里没有牛，那么答案增加 $0$；身高为 3 的牛来了，从栈顶开始，把所有小于或等于 3 的牛都出栈（这里暂时没有这样的牛），然后自己入栈，自己左边有 1 头牛，答案增加 1：身高为 7 的牛来了，从栈顶开始，把所有小于或等于 7 的牛都出栈（3 就出栈），然后自己入栈，自己左边有 1 头牛，答案增加 1……依此类推，具体步骤如图 1-5 所示。




图1-5 单调栈步骤

像这种维护一个值单调递减（或者递增）的数据结构，称为**单调栈**。根据以上分析，可以完成如下代码：

```
#include <iostream>
using namespace std;
#define MAXN 80010
int n, st[MAXN], top = 0, h;
long long ans;
int main() {
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> h;
        while (top > 0 && h >= st[top])
            top--;
        ans += top;
        top++;
        st[top] = h;
    }
    cout << ans;
    return 0;
}
```

考虑极端情况下，答案是 $n$ 的二次方数量级，超过了int的范围，所以统计答案的变量需要使用 long long 类型。

在本题中，单调栈能以 $O(n)$ 的时间复杂度找到每一个元素右边第一个比它大的数字（就是把这个数字挤出栈的数字），读者可以尝试输出这个数字验证。

### 例 1-8 长方形(洛谷P1950)

小明今天突发奇想，想从一张用过的纸中剪出一个长方形。为了简化问题，小明做出如下规定：

1. 这张纸的长和宽分别为 $n$ 和 $m$（$m$ 和 $n$ 均是不超过 $1000$ 的正整数）。小明将这张纸看成是由 $n \times m$ 个格子组成，在剪的时候，只能沿着格子的边缘剪。
2. 这张纸有些地方小明以前曾画过标记（用“*”表示），剪出来的长方形不能含有有标记的地方（没有标记的地方用“.”表示）。

剪出来的长方形的大小没有限制。

请问有多少种剪裁方法?

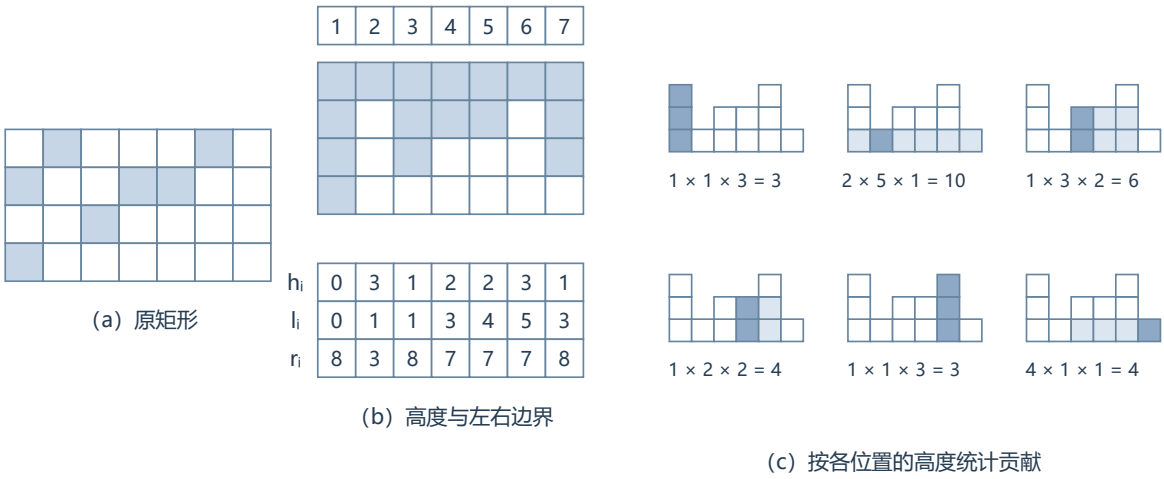
**输入样例:**

```
6 4
....
.*.*
.*..
.***
...*
.***
```

**输出样例:**

复杂度非常高，所以考虑以下做法来优化效率。

以行为单位处理，统计以第 $k$ 行为底边的矩形数量。例如，假设目前已经统计到了第 $4$ 行，如图 1-6(a) 所示。只要上面出现了被标记过的格子，那么这个格子上面的格子都不需要考虑了。因此，只需要统计这样一个“山”形的矩形数量，如图 1-6(b) 所示。




(a) 原矩形

(b) 高度与左右边界

(c) 按各位置的高度统计贡献

图1-6 统计格子数量

令 $h_i$ 为第 $i$ 列从底往上面延续空格子的数量，$l_i$ 为这一列左边第 $1$ 个满足 $h$ 值小于等于 $h_i$ 的列号（如果没有就是 $0$），$r_i$ 为这一列右边第 $1$ 个满足 $h$ 值小于 $h_i$ 的列号（如果没有就是 $m+1$）。如果这一行第 $i$ 列是标记过的格子，则 $h_i=0$，否则 $h_i$ 的值就是同一列上一行的 $h_i$ 的值增加 $1$。

根据乘法原理，包括这一列最底下的小格子，同时又被这一列的高度限制的正方形数量是 $(i-l_i)\times(r_i-i)\times h_i$。图 1-6(c) 表示 $i$ 从 $2$ 到 $7$ 的所有符合要求的计数区域。考虑相同高度的边界只会往右边扩展（因为左边是求小于或等于的边界，右边是求小于的边界），这样计数是不会重复也不会遗漏的。每一行都这么处理，然后将每一行的数据汇总即可。

使用**单调栈**求 $l_i$ 和 $r_i$。 求 $r_i$ 时需要找到严格小于的数字，那么判断是否将栈顶元素挤出时，如果待处理的元素等于栈顶元素时不必出栈，只有遇到更小的元素才会被挤出来，满足栈是单调不减的；而求 $l_i$ 要求小于或等于，则从后往前计算，判断是否将栈顶元素挤出时，如果这个元素小于或等于栈顶元素，那么栈顶元素就会被挤出，满足栈是单调递增的。

```
#include <iostream>
#define MAXN 1010
char ch[MAXN][MAXN];
int n, m, top;
int h[MAXN], l[MAXN], r[MAXN], st[MAXN];
long long ans;
using namespace std;
int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            cin >> ch[i][j];
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++)
            h[j] = ch[i][j] == '*' ? 0 : h[j] + 1;
        top = 0; // 顺着求右边小于这个数的位置,维护不减单调栈
```

```
        for (int j = 1; j <= m; j++) {
            while (top > 0 && h[j] < h[st[top]]) {
                r[st[top]] = j;
                top--;
            }
            top++;
            st[top] = j;
        }
        while (top > 0) {
            r[st[top]] = m + 1;
            top--;
        }
        top = 0; // 倒着求左边小于或等于这个数的位置,维护递增单调栈
        for (int j = m; j >= 1; j--) {
            while (top > 0 && h[j] <= h[st[top]]) {
                l[st[top]] = j;
                top--;
            }
            top++;
            st[top] = j;
        }
        while (top > 0) {
            l[st[top]] = 0;
            top--;
        }
        for (int j = 1; j <= m; j++) {
            ans += 1LL * (j - l[j]) * (r[j] - j) * h[j];
        }
    }
    cout << ans;
    return 0;
}
```

为了方便在出栈时求出每个元素左边和右边的符合要求的位置，单调栈里面存储的实际上是编号而不是数据本身。此外需要考虑极端情况：$n = m = 1000$ 而且这是一张白纸时，长方形的数量是 $n^4$ 的数量级，因此答案应当使用longlong数据类型储存。

这类问题除了需要掌握算法与数据结构工具，还需要借助同学们灵动的思维。刚刚接触这类题目可能会比较不容易思考，但经过反复的思考，想清楚每一步操作的原因，久而久之自然就会培养思维了。

## 1.4 单调队列

**单调队列**是一种内部元素具有单调性的队列，可以解决求“区间内最值”等问题。

### 例 1-9 扫描(洛谷P2032)

有一个 $1 \times n$ 的矩阵，有 $n$ 个整数。现有一个可以盖住连续 $k$ 个数的木板 $(1 \leqslant k \leqslant n \leqslant 2\times\ 10^6)$。一开始木板盖住了矩阵的第 $1$ 到 $k$ 个数，每次将木板向右移动一个单位，直到右端与第 $n$ 个数重合。求每次移动前输出被覆盖住的数字中最大的数是多少。

**输入样例：**

```
8 4
3 19 1 12 5 8 10 6
```

**输出样例：**

```
19
19
12
12
10
```

**分析：**暴力枚举的方式是进行 $n-k$ 次循环，每次查找长度为的 $k$ 区间的最值，这样的算法时间复杂度是 $O(n^2)$，也不能通过本题。

可以建立一个队列来维护这些数据，队首在左，队尾在右。首先队列为空，将元素加入队列中。如果接下来的元素比队尾元素更大，那么将队尾元素出队，直到这个元素不大于队尾元素为止（该操作和**单调栈**是一样的）。当处理到的数字编号大于或等于 $k$，说明已经处理完了 $k$ 个数字，可以查询这个区间的最大值了。这里的最大值就是队首元素。

在每次新加入元素之前，先要检查队首元素是否过期（不再处于需要统计的区间中）：如果队首元素的编号加上 $k$ 没有超过 $i$，则说明队首元素过期了，此时队首元素就需要出队。对本题样例数据，具体步骤如图 1-7 所示。

读入数字 队列记录的编号对应数据

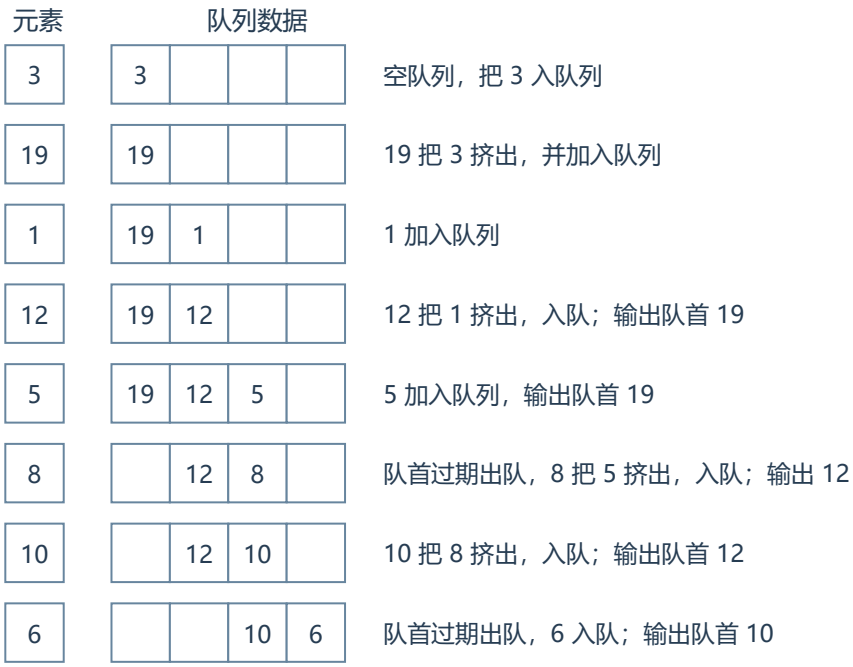



图1-7 单调队列步骤

本题中的队列不仅可以从队尾入队、队首出队，还可以从队尾出队①。使用这种队列，满足队列内的元素具有单调性，因此这种队列被称为**单调队列**。

```
#include <iostream>
using namespace std;
#define MAXN 1000010
int a[MAXN], q[MAXN];
int n, k;
int main() {
    cin >> n >> k;
    for (int i = 1; i <= n; i++)
        cin >> a[i];
    int head = 0, tail = 0;
    for (int i = 1; i <= n; i++) {
        while (head < tail && q[head] + k <= i)
            head++;
        while (head < tail && a[q[tail - 1]] < a[i])
            tail--;
        q[tail] = i;
        tail++;
        if (i >= k)
            cout << a[q[head]] << endl;
    }
    return 0;
}
```

因为需要知道队首元素是否过期，单调队列里面存储的实际上是编号而不是数据本身。本题也可以使用STL中的deque容器实现双端队列，感兴趣的读者可以自行查找相关的资料。

### 例 1-10 理想的正方形(洛谷P2216,河南省队选拔2007)

有一个 $a \times b$ 的整数组成的矩阵，现请从中找出一个 $n \times n$ 的正方形区域，使得该区域所有数中的最大值和最小值的差最小。

保证 $2 \leqslant a, b \leqslant 1000, n \leqslant a, n \leqslant b, n \leqslant 100$，矩阵的元素均为不超过 $10^9$ 的非负整数。

**输入样例：**

```
5 4 2
1 2 5 6
0 17 16 0
16 17 2 1
2 10 2 1
1 2 2 2
```

> **注：** 1. 参考《基础篇》中介绍“队列”中的“超市排队”案例。这种情况相当于在排队买东西的人中，排在最后的人发现后面要加入队伍的人更着急，就主动放弃了排队。

**输出样例：**

**分析：**暴力枚举的方式是枚举所有符合要求的小正方形，每次在小正方形中查找最小或最大值并统计，这样时间复杂度是 $O(n^4)$，不能通过本题。

如何求出一个正方形矩阵的最大值呢?可以将这个正方形的每一行数字求最大值，然后再求每一行最大值中的最大值，剩下的一个值就是这个正方形矩阵的最大值，如图 1-8 所示。

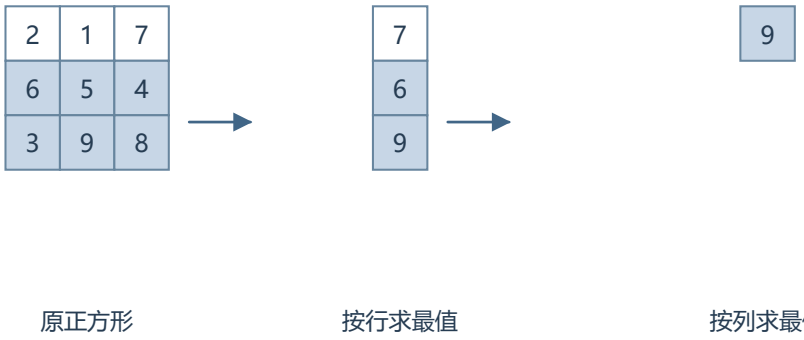



图1-8 求正方形矩阵的最值

以样例为例，首先处理每一行，找出一行中每个长度为 2 的区间的最大值；然后处理每一列，找到一列中每个长度为 2 的区间的最大值。最后得到的矩阵中的一个元素就对应着一个 2×2 的正方形的最大值。使用相同的方式求得每个 2×2 正方形的最小值，然后遍历这两个矩阵，计算这两个数的极差并统计即可得到答案，具体步骤如图 1-9 所示。

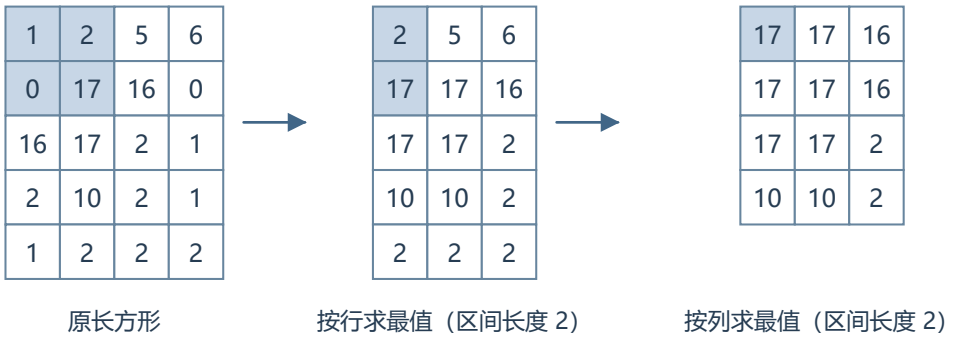



图1-9求矩阵中每一个2×2的正方形矩阵中的最值

使用**单调队列**来求指定区间的最值是已经介绍过的内容，代码如下：

```
#include <iostream>
using namespace std;
#define MAXN 1010
int m[MAXN][MAXN], q[MAXN];
int rowmin[MAXN][MAXN], rowmax[MAXN][MAXN];
int colmin[MAXN][MAXN], colmax[MAXN][MAXN];
int a, b, n, ans = 1000000000;
int main() {
    cin >> a >> b >> n;
    for (int i = 1; i <= a; i++)
        for (int j = 1; j <= b; j++)
            cin >> m[i][j];
    for (int row = 1; row <= a; row++) {
        // 求每行区间最大值
        int h = 0, t = 0;
```

```
        for (int i = 1; i <= b; i++) {
            while (h < t && q[h] + n <= i)
                h++;
            while (h < t && m[row][q[t - 1]] < m[row][i])
                t--;
            q[t] = i;
            t++;
            if (i >= n)
                rowmax[row][i - n + 1] = m[row][q[h]];
        }
        // 求每行区间最小值
        h = 0, t = 0;
        for (int i = 1; i <= b; i++) {
            while (h < t && q[h] + n <= i)
                h++;
            while (h < t && m[row][q[t - 1]] > m[row][i])
                t--;
            q[t] = i;
            t++;
            if (i >= n)
                rowmin[row][i - n + 1] = m[row][q[h]];
        }
    }
    for (int col = 1; col <= b - n + 1; col++) {
        // 在刚得到的新矩阵的基础上,求每列区间最大值
        int h = 0, t = 0;
        for (int i = 1; i <= a; i++) {
            while (h < t && q[h] + n <= i)
                h++;
            while (h < t && rowmax[q[t - 1]][col] < rowmax[i][col])
                t--;
            q[t] = i;
            t++;
            if (i >= n)
                colmax[i - n + 1][col] = rowmax[q[h]][col];
        }
        // 在刚得到的新矩阵的基础上,求每列区间最小值
        h = 0, t = 0;
        for (int i = 1; i <= a; i++) {
            while (h < t && q[h] + n <= i)
                h++;
            while (h < t && rowmin[q[t - 1]][col] > rowmin[i][col])
                t--;
            q[t] = i;
            t++;
            if (i >= n)
                colmin[i - n + 1][col] = rowmin[q[h]][col];
        }
    }
    for (int i = 1; i <= a - n + 1; i++)
        for (int j = 1; j <= b - n + 1; j++)
            if (colmax[i][j] - colmin[i][j] < ans)
                ans = colmax[i][j] - colmin[i][j];
    cout << ans;
    return 0;
}
```

该程序写了 4 遍单调队列：按行求区间最大值，按行求区间最小值，以及分别存为新的矩阵后，按列求区间的最大值和最小值。求区间最小值时，单调队列里的元素值是单调不降的。

实现代码时，可以多次复制结构类似的语句，然后修改特定的部分使其满足要求即可。但是需要特别注意的是，需要修改的变量或者操作符一定要修改完全。

本章提供了一些使用简单数据结构维护信息以求解特定问题的技巧。这些技巧在算法竞赛中往往不是单独使用而是与其他算法或数据结构配合使用的，如使用单调队列优化**动态规划**等。在后面的章节中将详细介绍这些内容。

## 1.5 课后习题与实验

仅仅阅读本书并不能很好地提升能力，必须亲自动手实践方能掌握。请读者尽可能完成每一章的课后练习。

### 习题 1 唯一的雪花(UVa11572)

有 $T(1 \leqslant T \leqslant 100)$ 组测试数据。每组测试数据中有 $N(1 \leqslant N \leqslant 10^6)$ 个整数 $(0 \sim 10^9)$，这些数字可能重复。求一个尽可能长的连续区间，满足区间内所有的数字均不相同。

### 习题 2 稳赚不赔(洛谷P4653,CEOI2017)

有两组物品各 $n(1 \leqslant n \leqslant 10^5)$ 个，每个物品都有一定的价值（$1$ 到 $1000$）。现在需要选择若干个物品，收益为两组物品中价值和较少的那组物品的价值之和，减去所选取的所有物品数，使收益最大化。

### 习题 3 钻石收藏家(洛谷P3143,USACO2016月赛)

小贝有 $N(1 \leqslant N \leqslant 50000)$ 枚钻石，每枚钻石的大小在 $0$ 到 $10^9$ 之间。他想从这些钻石中挑选一些放在两个陈列架上，且满足每个陈列架中最大钻石和最小钻石的大小差值不超过 $K$。请问最多可以放下多少枚钻石？

### 习题 4 插入排序(洛谷P7910,CSP-J2021)

给定 $n(n \leqslant 8000)$ 和 $n$ 个数字，进行 $Q(Q \leqslant 2 \times 10^5)$ 次操作。

操作 $1:1\ x\ y$。将数组的第 $x$ 个元素变成 $y$ 这种操作不会超过 $5000$ 次。

操作 $2:2\ x$。 假设使用对数组进行插入排序，请问原来的 $a[x]$ 排序后的位置是第几个？这种操作不会改变原数组，排序后的数组不会保留。

### 习题 5 奶牛浴场(洛谷P1578)

有一个大小为 $L \times W(1 \leqslant L, W \leqslant 30000)$ 的牛场，有 $N(1 \leqslant N \leqslant 5000)$ 头奶牛，分别在坐标 $(x_i y_i)$ 产奶。现在需要建立一个矩形的浴场，其轮廓和牛场的轮廓平行，且不能覆盖任何产奶点（产奶点可以在浴场边缘），求浴场的最大面积。

### 习题 6 贴海报(洛谷P3467,POI2008)

给出 $n(1 \leqslant n \leqslant 2.5 \times 10^5)$ 个长方形建筑，并知道它们的宽和高分别为 $d_i$ 和 $w_i(1 \leqslant d_i, w_i \leqslant 10^9)$。这些建筑按顺序从左往右紧挨着排列，底边处于同一水平线上。现在需要使用若干矩形海报（大小可以自定义）覆盖住这些建筑，不能重叠也不能超出这些建筑的范围。请问最少需要多少个矩形海报?

### 习题 7 滑动窗口(洛谷P1886)

有一个长为 $n$ 的序列 $a$，以及一个大小为 $k$ 的窗口。现在将这个窗口从左边开始向右滑动，每次滑动一个单位，求出每次滑动后窗口中的最大值和最小值。

**数据范围：** $1 \leqslant k \leqslant n \leqslant 10^6, a_i \in \left[-2^{31}, 2^{31}\right)$。

### 习题 8 平衡的队列(洛谷P2880,USACO2007月赛)

农场里有 $n(1 \leqslant n \leqslant 5 \times 10^4)$ 头牛排队，并已知所有牛的身高 $h_i(1 \leqslant h_i \leqslant 10^6, 1 \leqslant i \leqslant n)$。农夫询问了 $q(1 \leqslant q \leqslant 1.8 \times 10^5)$ 个区间[ $[a_i, b_i]$，他想知道每个区间中，最高和最低的牛的身高差。

### 习题 9 切蛋糕(洛谷P1714)

一块蛋糕是一个长方体，被用不同色彩分成了 $n(1 \leqslant n \leqslant 5 \times 10^5)$ 个相同的小块，每小块都有对应的幸运值，其绝对值不超过 $2^{31}-1$，注意可能出现负数。小 $Z$ 希望吃到的蛋糕的幸运值总和最大，但他最多又只能吃 $m(m \leqslant n)$ 小块的蛋糕。请帮助他从这 $n$ 小块中找出连续的 $k(1 \leqslant k \leqslant m)$ 块蛋糕，使得其幸运值总和最大。

### 习题 10 小琪的格子(洛谷P1725)

有一列格子，依次编号为 $0$ 到 $N$，最开始小琪站在编号为 $0$ 的格子上，她只能从编号小的格子移动到编号大的格子。当她在格子 $i$ 时，她只能移动到区间 $[i+L, i+R]$ 中的任意一格。每一个格子都有一个冰冻指数 $A_i$，编号为 $0$ 的格子的冰冻指数为 $0$。当小琪停留在某格时就可以得到那一格的冰冻指数。请问：她走出这些格子时可以获得的冰冻指数之和的最大值是多少?

**数据范围：** $N \leqslant 2 \times 10^5, -10^3 \leqslant A_i \leqslant 10^3, 1 \leqslant L \leqslant R \leqslant N$。

# 第2章 前缀和、差分与离散化

不积跬步，无以至千里；不积小流，无以成江海。这句话揭示了事物整体和部分之间的关系庞大的整体由若干个体组成；单个个体虽小，但经过一点一滴的累积，也能聚成一个庞大的整体。学习贵在不断积累，不断充实自己，所有的努力都会有回报。

在算法竞赛中，利用整体和部分的性质可以达成很多目的，如利用**前缀和**可以在常数时间复杂度内查询区间和，利用差分在常数时间复杂度内对序列进行区间操作，或者利用**离散化**去除无用数据区间，通过放缩保留有用的数据。这些操作使得人们不必每次都要重复处理个体的数据，而是直接对整体进行处理，从而降低时间复杂度，提升运算效率。本章将会学习前缀和、差分和离散化的思想，并使用这些思想完成一些简单的任务。本章思维导图如图 2-1 所示。




图2-1 本章思维导图

## 2.1 前缀和与二维前缀和

### 例 2-1 求区间和(洛谷P8218)

给定由 $n$ 个正整数组成的数列 $a_1, a_2, \ldots, a_n$ 和 $m$ 个区间 $[l_i, r_i]$，分别求这些区间的区间和。

**数据范围：**对于 $50\%$ 的数据： $n, m \leqslant 1000$；对于 $100\%$ 的数据：$n$， $m \leqslant 10^5, a_i \leqslant 10^4$。

**输入样例：**

```
4
4321
2
1 4
23
```

**输出样例：**

**样例解释：**第 **1** 个数到第 **4** 个数加起来和为 **10**。第 **2** 个数到第 **3** 个数加起来和为 **5**。

**分析：**最直接的思路是对于每次询问，从 $l_i$ 枚举到 $r_i$，并将每个位置上的数加起来。这样时间复杂度为 $O(m \times n)$，可以通过 50%的数据。

设 $s_i = \sum_{j=1}^{i} a_j = a_1 + a_2 + \cdots + a_i$，即前 $i$ 个数之和，并令 $s_0 = 0$。符号 $\sum$ 表示求和。当 $i \geq 1$ 时，$s_i - s_{i-1} = a_i$，因此 $s_i = s_{i-1} + a_i$。数组 $s$ 记录的数值称为**前缀和**，也就是对每个前缀区间求和。

例如，表 2-1 列出了 $a_1$ 到 $a_n$ 的值。

| i | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $a_i$ | 1 | 3 | 4 | 5 | 6 | 2 |
| $s_i$ | 1 | 4 | 8 | 13 | 19 | 21 |

表2-1 \dot { a } _ { \uparrow } 到 \boldsymbol { a } _ { n } 的值

由 表 2-1 可知，$a_3 + a_4 = (a_1 + a_2 + a_3 + a_4) - (a_1 + a_2) = s_4 - s_2$，即 $\sum_{i=l}^{r} a_i = \sum_{i=1}^{r} a_i - \sum_{i=1}^{l-1} a_i = s_r - s_{l-10}$ 求出 $s$ 数组后，即可求出任意区间的区间和。

根据以上思路，完成如下代码：

```
#include <iostream>
#include <cstdio>
using namespace std;
#define MAXN 100010
int n, m;
int a[MAXN], s[MAXN];
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        scanf("%d", &a[i]);
        s[i] = s[i - 1] + a[i];
        // 前缀和数组的第i 项 = a数组的第1 项到第 i-1 项的和 + a数组的第i 项的值
    }
    scanf("%d", &m);
    for (int i = 1, x, y; i <= m; i++) {
        scanf("%d%d", &x, &y);
        printf("%d\n", s[y] - s[x - 1]);
    }
    return 0;
}
```

一维空间**前缀和**可以用来快速求出一维序列的区间总和，而二维前缀和可以快速求出一个二维矩阵中，内部一片长方形区域的所有元素的和。二维前缀和的求解基于**容斥原理**，在本书 第5部分 将详细叙述该原理。

### 例 2-2 最大加权矩形(洛谷P1719)

有一个 $n \times n(n \leqslant 120)$ 矩阵。要求矩阵中最大加权矩形，即矩阵的每个元素都有一个权值，权值定义在整数集上。从中找出一个矩形，矩形大小无限制，使其中包含的所有元素的和最大。矩阵的每个元素在$[-127,127]$区间。图 2-2 给出了一个符合题目要求的矩阵。

| | | | |
|---|---|---|---|
| 0 | -2 | -7 | 0 |
| 9 | 2 | -6 | 2 |
| -4 | 1 | -4 | 1 |
| -1 | 8 | 0 | -2 |

图2-2 最大加权矩形

图 2-2 中左下角的区域为矩阵中的最大加权矩形，和为 $15$。

**分析：**最直接的做法是枚举左上角端点坐标和右下角端点坐标，再将矩形内的元素求和，时间复杂度为$O(n^6)$。如果可以在 $O(1)$ 的时间复杂度内求出一个矩形内的元素和，时间复杂度可以降至 $O(n^4)$

与一维空间**前缀和**类似，设 $s[i,j]$ 表示以（1，1）为矩阵左上角端点， $(i,j)$ 为右下角端点的矩形的和。

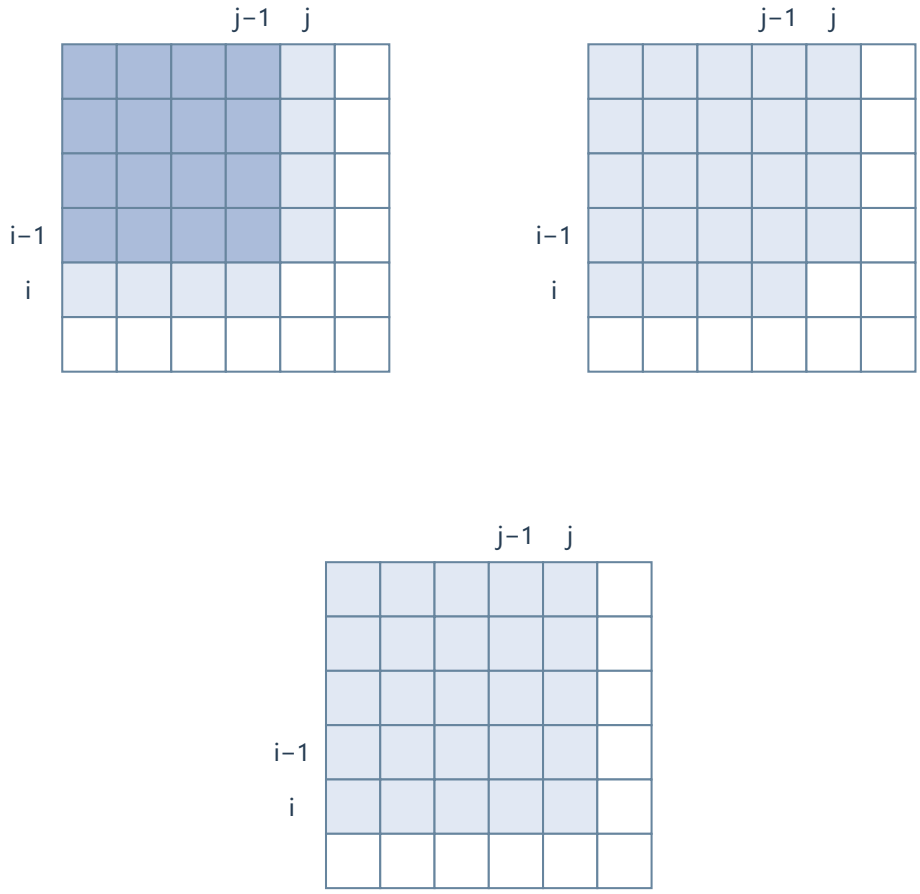



图2-3 预处理二维前缀和矩阵

首先需要快速计算出 $s[i,j]$。先计算 $s[i,j-1]$ 和 $s[i-1,j]$ 之和，如图 2-3(a) 所示，浅色部分表示该元素的值被计算了一次，深色部分表示该元素的值被计算了两次。深色部分si-1，$j-1$]重复计算，于是减去这部分，如图 2-3(b) 所示。最后加上元素 $a[i,j]$ 的值，就可以得到 `s[i,j]` 的结果，如图 2-3(c) 所示。

递推式为 $s[i,j]=s[i,j-1]+s[i-1,j]-s[i-1,j-1]+a[i,j]$。

接下来需要快速求出以 $(x_1,y_1)$ 为左上角端点以 $(x_2,y_2)$ 为右下角端点的矩形内的元素和。首先加上 $s[x_2,y_2]$ 部分与 $s[x_1-1,y_1-1]$ 的部分，如图 2-4(a) 所示；再减去 $s[x_2,y_1-1]$ 的部分，如图 2-4(b) 所示；最后减去 $s[x_1-1,y_2]$ 的部分，剩余部分即为所求，如图 2-4(c) 所示。

递推式

$$\sum_{i=x_1}^{x_2}\sum_{j=y_1}^{y_2}a[i,j]=s[x_2,y_2]-s[x_1-1,y_2]$$
$$-s[x_2,y_1-1]+s[x_1-1,y_1-1].$$

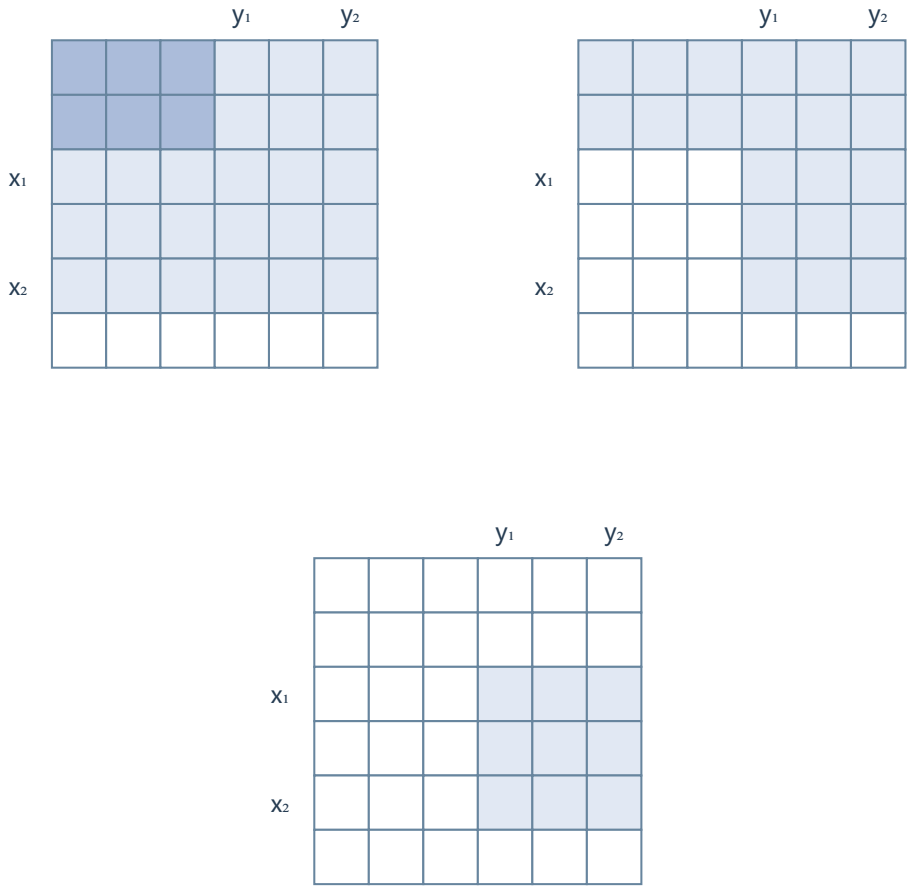



图2-4 子矩阵和的求法

由于常数小，时间复杂度为 $O(n^4)$ 的做法可以通过本题。代码如下：

```
#include <iostream>
#include <cstdio>
using namespace std;
#define MAXN 130
int n;
int a[MAXN][MAXN], s[MAXN][MAXN];
int qsum(int X1, int Y1, int X2, int Y2) {
    // 求以(X1,Y1)为左上角端点,(X2,Y2)为右下角端点的矩形的元素和,由于保留字的
    //  原因X1等采用大写
    return s[X2][Y2] + s[X1 - 1][Y1 - 1] - s[X2][Y1 - 1] - s[X1 - 1][Y2];
}
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++) {
            scanf("%d", &a[i][j]);
```

```
            s[i][j] = s[i][j - 1] + s[i - 1][j] - s[i - 1][j - 1] + a[i][j];
        }
    int ans = -10000000;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            for (int k = i; k <= n; k++)
                for (int l = j; l <= n; l++)
                    ans = max(ans, (qsum(i, j, k, l)));
    printf("%d", ans);
    return 0;
}
```

进一步优化，枚举 $i$ 和 $j$，即求所有上边界为第 $i$ 行、下边界为第 $i$ 行的矩形中，元素和的最大值，进而求解出本题答案。上边界和下边界固定时，可以将子问题转化为给出一个长度为 $n$ 的序列 $a_1,a_2,\cdots,a_n$，其中 $a_k$ 为指定 $i$ 和 $i$ 为上下边界的矩形中，第 $k$ 列的数字之和。选出其中连续且非空的一段的子序列，使得这段和最大。子问题具体求解过程参见例 1-3（最大子段和）。

根据上面的思路，可以完成如下代码：

```
#include <iostream>
#include <cstdio>
using namespace std;
#define MAXN 130
int n;
int a[MAXN][MAXN], s[MAXN][MAXN];
int qsum(int X1, int Y1, int X2, int Y2) {
    return s[X2][Y2] + s[X1 - 1][Y1 - 1] - s[X2][Y1 - 1] - s[X1 - 1][Y2];
}
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++) {
            scanf("%d", &a[i][j]);
            s[i][j] = s[i][j - 1] + s[i - 1][j] - s[i - 1][j - 1] + a[i][j];
        }
    int ans = -10000000, sum;
    for (int i = 1; i <= n; i++)       // 枚举矩形区域的上边界
        for (int j = i; j <= n; j++) { // 枚举矩形区域的下边界
            sum = 0;
            for (int k = 1; k <= n; k++) {
                sum += qsum(i, k, j, k);
                if (sum > ans)
                    ans = sum;
                if (sum < 0)
                    sum = 0;
            }
        }
    printf("%d", ans);
    return 0;
}
```

根据这个思路，也可以使用一维前缀和或不使用前缀和的解决方法，留给读者自行思考。

### 例 2-3 聪明的质监员（洛谷 P1314，NOIP2011 提高组）

小 $T$ 是一名质量监督员，最近负责检验一批矿产的质量。这批矿产共有 $n$ 个矿石，从 $1$ 到 $n$ 逐一编号，每块矿石都有自己的重量 $w_i$ 以及价值 $v_i$。检验矿产的流程如下：

1. 给定 $m$ 个区间 $[l_i, r_i]$。
2. 选出一个参数 $W$。
3. 对区间 $[l_i, r_i]$，检验值为 $y_i = \left(\sum_{j=l_i}^{r_i}[w_j \geq W]\right)\left(\sum_{j=l_i}^{r_i}[w_j \geq W]v_j\right)$。其中 $j$ 为矿石编号；$[w_j \geq W]$ 在条件成立时为 $1$，否则为 $0$。

这批矿产的检验结果 $y$ 为各个区间的检验值之和，即 $\sum_{i=1}^{m} y_i$。小 $T$ 想通过调整参数 $W$ 的值，让检验结果尽可能地靠近标准值 $s$，即使得 $|s-y|$ 最小。求出这个最小值。

**数据范围：**有 $1 \leqslant n, m \leqslant 2\times 10^5, 0 < w_i, v_i \leqslant 10^6, 0 < s \leqslant 10^{12}, 1 \leqslant l_i \leqslant r_i \leqslant n$

**分析：**首先分析题意。作为NOIP的题目，数学符号稍多，对于读者来说理解题意可能是一道门槛。若想得到各个区间的检验值之和，则需要求出各个区间的检验值。令满足 $w_j \geqslant W$ 的第 $j$ 块矿石为“合格品”，因为不合格品不会被计入，对于单个区间的检验值就是“该区间合格品的数量”乘以“该区间合格品的价值总和”。

当检验标准 $W$ 提高时，合格品的数量就会减少，对于每个区间的检验结果 $y_i$ 也会减少，总的检验结果 $y$ 也会降低。当调节合适的 $W$ 值，就可以使检验结果 $y$ 接近标准值 $s$，这具有明显的单调性，因此考虑二分答案的思路。

二分 $W$ 的值，计算对应 $y$ 的值，要求在满足 $x-y\geq 0$ 的情况下 $y$ 尽可能大，$W$ 尽可能小。二分的区间可以从 0 到 $\max(w_i)+1$ （加 1 是因为有可能存在所有矿石都不合格的情况），这个过程时间复杂度为 $O(\log(\max(w_i)))$。

计算 $y$ 值时，如果对每个区间逐个判断元素并累加，时间复杂度为 $O(nm)$，会超时。可以使用**前缀和**：令 $sw_i$ 表示前 $i$ 块矿石中满足 $w_j \geq W$ 的数量，$sv_i$ 表示这些合格矿石的价值和。于是区间 $[l,r]$ 的合格矿石数量为 $sw_r - sw_{l-1}$，价值和为 $sv_r - sv_{l-1}$，该区间的检验值为两者之积。一次计算所有区间的总检验值需要 $O(n+m)$ 时间，结合二分，程序总时间复杂度为 $O((n+m)\log(\max_i w_i + 1))$。

代码如下：

```
#include <bits/stdc++.h>
using namespace std;
#define MAXN 200010
int n, m, x, y, mid;
int l[MAXN], r[MAXN];
int w[MAXN], v[MAXN];         // 重量和价值
long long sv[MAXN], sw[MAXN]; // 价值前缀和、数量前缀和
long long W, S, sum;
int main() {
    scanf("%d%d", &n, &m);
    scanf("%lld", &S);
    W = S; // 答案,初始值设为S
    for (int i = 1; i <= n; i++) {
        scanf("%d%d", &w[i], &v[i]);
        y = max(y, w[i]);
    }
```

```
    x = 1;
    y++;
    for (int i = 1; i <= m; i++)
        scanf("%d%d", &l[i], &r[i]);
    while (x < y) {
        memset(sv, 0, sizeof(sv));
        memset(sw, 0, sizeof(sw)); // 清空前缀和数组
        mid = (x + y) >> 1;
        sum = 0;
        for (int i = 1; i <= n; i++) {
            if (w[i] >= mid) { // 合格品计人统计
                sw[i] = sw[i - 1] + 1;
                sv[i] = sv[i - 1] + v[i];
            } else {
                sw[i] = sw[i - 1];
                sv[i] = sv[i - 1];
            }
        } // sv:记w[i] 符合条件的v[i]的前缀和
        for (int i = 1; i <= m; i++) {
            sum += (long long)(sw[r[i]] - sw[l[i] - 1]) * (sv[r[i]] - sv[l[i] - 1]);
        }
        // 求每一段区间的检验值
        if (sum == S) {
            W = 0;
            break; // 如果刚好是标准值,那答案就是0
        }
        W = min(W, abs(sum - S)); // 更新答案
        if (sum <= S)             // y值太小,则二分左区间以减小w值
            y = mid;
        else if (sum > S)
            x = mid + 1;
    }
    printf("%lld", W);
    return 0;
}
```

这里二分的技巧是将区间 $[x,y]$ 分为两个子区间 $[x,\mathbf{mid}]$ 和 $[\mathbf{mid}+1,y]$，同时要求 $x<y$，这样可以使枚举区间不重不漏。每次枚举mid值都会记录一下最优答案。请读者思考 $s-y<0$ 时，是否也有可能记录到最优答案呢？

## 2.2 差分

### 例 2-4 语文成绩(洛谷P2367)

班上共有 $n$ 个学生，老师在统计语文成绩的时候偶尔会出错。老师需要对学生成绩进行 $p$ 次修改，每次修改需要给第 $x$ 个学生到第 $y$ 个学生每人增加 $z$ 分。老师想知道成绩修改后全 $x_iy_iz_i$

班的最低分。

**数据范围：**对于 $40\%$ 的数据，有 $n\leqslant1000$；对于 $100\%$ 的数据，有 $n\leqslant5\times10^6,p\leqslant n$；学生初始成绩 $\leqslant100,z\leqslant100$，

**分析：**对于 40%的数据，可以直接模拟修改，时间复杂度为 $O(n^2)$，无法通过此题。因此需要使用差分的方法。

对于数组 $a$，定义**差分数组** $b$：$b_1=a_1$，$b_i=a_i-a_{i-1}$（$2\leq i\leq n$），见表 2-2。

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $a_i$ | 1 | 1 | 2 | 2 | 1 | 3 | 3 | 5 |
| $b_i$ | 1 | 0 | 1 | 0 | -1 | 2 | 0 | 2 |

表2-2 差分数组

对数组 $b$ 求**前缀和**：

$$\sum_{i=1}^{n} b_i = a_1 + \sum_{i=2}^{n}(a_i - a_{i-1}) = \sum_{i=1}^{n} a_i - \sum_{i=1}^{n-1} a_i = a_n$$

若将 $b_x$ 增加 1，再对数组 $b$ 求前缀和得到数组 $a$，那么 $a$ 中的 $a_x, a_{x+1}, \cdots, a_n$ 均增加 $1$ 若将 $b_y$ 减少 1，再对数组 $b$ 求前缀和得到数组 $a$，那么 $a$ 中的 $a_y, a_{y+1}, \cdots, a_n$ 均减少 1。例如，对一个全为 0 的差分数组 $b$，将 $b_3$ 增加 1，将 $b_6$ 减少 1，得到的数组 $a$ 见表 2-3。

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $b_i$ | 0 | 0 | +1 | 0 | 0 | -1 | 0 | 0 |
| $a_i$ | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |

表2-3 差分的过程

因此，将数组 `a` 中第 $x$ 到第 $y$ 个元素增加 $z$，可以转化为将 `b[x]` 增加 $z$、`b[y+1]` 减少 $z$，最后再对数组 `b` 求前缀和。每次修改只需改变两个位置，时间复杂度为 $O(1)$。由于只在最后一次修改后才需要数组 `a` 的具体值，所有修改结束后求一次前缀和即可。代码如下：

```
#include <iostream>
#include <cstdio>
using namespace std;
#define MAXN 5000010
int n, m, x, y, z, a[MAXN], b[MAXN];
int main() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++) {
        scanf("%d", &a[i]);
        b[i] = a[i] - a[i - 1]; // 求差分数组 b
    }
    for (int i = 1; i <= m; i++) {
        scanf("%d%d%d", &x, &y, &z);
        b[x] += z; // 修改操作,将 a[x],a[x+1],...,a[y]加上z
        b[y + 1] -= z;
    }
```

```
    int Min = 101;
    for (int i = 1; i <= n; i++) {
        b[i] += b[i - 1]; // 对数组 b 求前缀和
        if (b[i] < Min)
            Min = b[i];
    }
    printf("%d\n", Min);
    return 0;
}
```

### 例 2-5 地毯(洛谷P3397,By阮行止)

在 $n\times$ `n(n≤1000)` 的格子上有 $m(m\leqslant 1000)$ 块地毯，给出这些地毯的信息，问每个点被多少块地毯覆盖。

**输入格式：**第 $1$ 行，两个正整数 $n$ 和 $m$，意义如题所述。

接下来 $m$ 行，每行两个坐标 $(x_1,y_1)$ 和 $(x_2,y_2)$，代表一块地毯，左上角是 $(x_1,y_1)$，右下角是 $(x_2,y_2)$。

**输出格式：**输出共 $n$ 行，每行 $n$ 个非负整数。第 $i$ 行第 $j$ 列的正整数表示 $(i,j)$ 这个格子被多少块地毯覆盖。

**输入样例：**

```
5 3
2233
3 3 5 5
1214
```

**输出样例：**

```
01110
0 11 0 0
0 1 211
0 0 11 1
0 0 111
```

样例解释如图 2-5 所示。

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |

(a) 覆盖第一块地毯后

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 2 | 1 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 1 |

(b)覆盖前两块地毯后

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 2 | 1 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 1 |

(c) 覆盖所有地毯后

图2-5 覆盖地毯

**分析：** 设数组 $a$ 的**差分数组**为 $b$，其中 $a[x,y]=\sum_{i=1}^{x}\sum_{i=1}^{y}b[i,j]$，用**前缀和**的知识可知，$b[x_{\gamma}\neq a[x,y]-a[x-1,y]-a[x,y-1]+a[x-1,y-1]$

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |

将 `b[2,2]` 增加 1，再求二维前缀和，效果如图 2-6(a) 和 图 2-6(b) 所示。单独考虑一次覆盖操作，地毯左上角为 $(x_1,y_1)$，右下角为 $(x_2,y_2)$，可以转化为将 $b[x_1,y_1]$ 和 $b[x_2+1,y_2+1]$ 增加 1，将 $b[x_1,y_2+1]$ 和 $b[x_2+1,y_1]$ 减少 $1$。

当地毯左上角为（2，2），右下角为（3，3），修改操作如图 2-6(c) 所示，求二维前缀和后结果如图 2-6(d) 所示。具体代码如下：

```
#include <iostream>
#include <cstdio>
using namespace std;
#define MAXN 1010
int n, m, X1, Y1, X2, Y2;
int a[MAXN][MAXN];
int main() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= m; i++) {
        scanf("%d%d%d%d", &X1, &Y1, &X2, &Y2);
        a[X1][Y1]++; // 修改操作
        a[X1][Y2 + 1]--;
        a[X2 + 1][Y1]--;
        a[X2 + 1][Y2 + 1]++;
    }
    for (int i = 1; i <= n; i++, printf("\n"))
        for (int j = 1; j <= n; j++) {
            a[i][j] += a[i - 1][j] + a[i][j - 1] - a[i - 1][j - 1]; // 求前缀和
            printf("%d ", a[i][j]);
        }
}
```

## 2.3 离散化

### 例 2-6 火烧赤壁(洛谷P1496)

有一个白色的数轴，现在有 $n$ 个指令，每个指令将数轴上的一个区间 $[a_i,b_i]$ 染成黑色，重复染色效果和一次染色效果相同。求 $n$ 个指令都执行完毕后，被染成黑色的区间的总长度。

**数据范围：** $n\leqslant 20000,-2^{31}\leqslant a_i\leqslant b_i<2^{31}$ 且 $a_i$ 和 $b_i$ 都是整数。

**输入样例：**

```
3
-1 1
5 11
2 9
```

**输出样例：**

```
11
```

样例解释如图 2-7 所示。

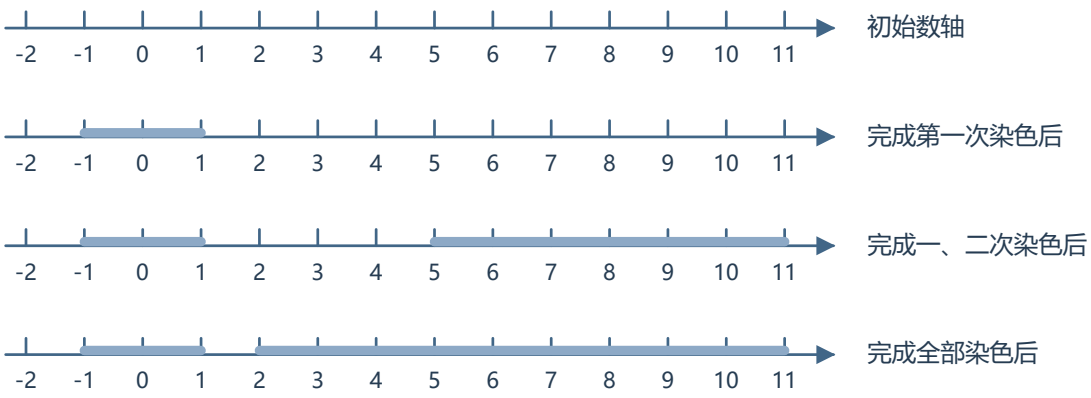



图2-7例2-6样例解释

因此共有长度为 $11$ 的区间被染成黑色。

**分析：**在思考问题时，可以先考虑一些更简单的情况，然后再将其引申到复杂的情况。例如在本题中，不妨处理 $a_i$ 和 $b_i$ 的取值范围都很小的情况。

如果 $a_i$ 和 $b_i$ 的值很小，完全可以用一个数组表示数轴。数组中的元素 $f_i$ 就表示数轴上一个长度为 1 的区间$[i,i+1]$是否被染成黑色 $f_i=0$ 时表示未染色，否则表示已染色。接着只需要模拟执行每个指令，枚举数轴区间对应的数组区间并赋值即可。

若将这个做法引申到 $a_i$ 和 $b_i$ 很大的情况，就可以使用**离散化**的方法。

离散，就是不连续的意思。离散化可以在保持原数值之间相对大小关系不变的情况下，将其映射成正整数，也就是给每个可能用到的数值按照大小关系分配一个编号，用此编号来代替原数值进行各种操作。在数值的范围很大但可能用到的数值较少的情况下，离散化可以将无用的数值剔除，只保留关键数值。例如，将数组 $a$ 中的元素离散化后得到数组 $b$，见表 2-4。

| i | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $a_i$ | 66 | 3 | 16 | 66 | -1 |
| $b_i$ | 4 | 2 | 3 | 4 | 1 |

表2-4 数组离散化值

在本题中，$-2^{31} \leqslant a_i \leqslant b_i < 2^{31}$，无法直接用一个数组 $f$ 来表示这个数轴，且在模拟染色过程中时间复杂度很大，无法在本题限制时间内完成。本题虽然数字的范围很大，但是有用的数字却不多。数轴上相邻两个端点（不一定是同一个区间的端点）之间所有的点都可以被一视同仁，因为它们要么同时被染色，要么都不被染色，如图 2-8 所示。注意，因为题目相关的数值都是整数，所以只考虑数轴上的整数点。

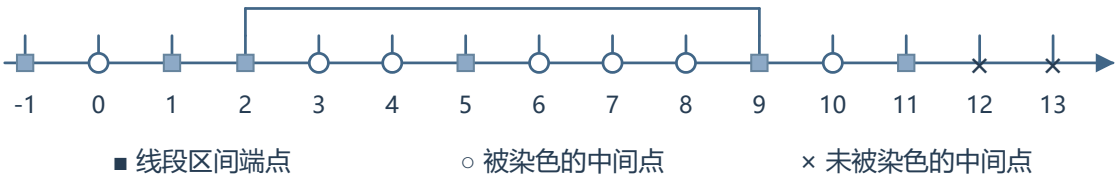



图2-8 数轴上的点

因此将本题样例中的数据进行处理，得到一个新的数轴，如图 2-9 所示。

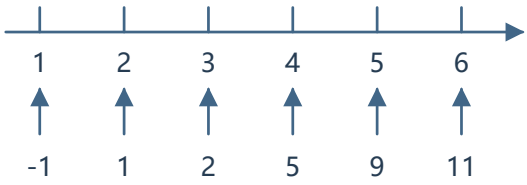



图2-9 离散化后的点

将区间[-1，1]染色可以转化为将区间[1，2]染色，最后统计答案时，将区间[1，2]的长度记为 2 即可。具体代码如下：

```
#include <bits/stdc++.h>
using namespace std;
#define MAXN 20010
int n, a[MAXN], b[MAXN], f[MAXN * 2], dtop, ctop, d[MAXN * 2];
long long c[MAXN * 2];
long long ans; // 最后求解时答案可能会超过 int 的范围
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        scanf("%d%d", &a[i], &b[i]);
        d[++dtop] = a[i]; // 将需要用到数轴上的散点存入到数组 d中
        d[++dtop] = b[i];
    }
    sort(d + 1, d + dtop + 1); // 将数组d 中的元素从小到大排序
    for (int i = 1; i <= dtop; i++)
        if (d[i] != d[i - 1] || i == 1)
            c[++ctop] = d[i]; // 除去重复元素
    for (int i = 1; i <= n; ++i) {
        int x = lower_bound(c + 1, c + ctop + 1, a[i]) - c;
        int y = lower_bound(c + 1, c + ctop + 1, b[i]) - c;
        for (int j = x; j < y; j++)
            f[j] = 1;
    }
    for (int i = 1; i < ctop; i++)
        if (f[i])
            ans += c[i + 1] - c[i];
    printf("%lld\n", ans);
    return 0;
}
```

对数组 $d$ 的排序使得该数组中相同的数值相邻，这样既便于去除重复数值，又便于后续的二分查找操作。排序后的操作为去重的过程，通常的去重方法为将数组 $d$ 中的数值放入另一个数组 $c$，而且只将相邻的重复数值中最左边的一个放入（如果 $d_i \neq d_{i-1}$ 或 $i=1$，它就是相邻重复数值中最靠左的那个）。

通过lower_bound（STL中的二分查找）操作，找到数组 $c$ 中第 1 个大于或等于 $a_i$ 的数所在的位置，赋值给变量 $x$，即 $a_i$ 映射后对应的值是 $x$。 用同样的方式可得 $b_i$ 映射后对应的值 $y$

染色过程循环终止条件为 $j<y$，这是因为 $f_i$ 的值来表示区间 $[c_i,c_{i+1}]$ 的颜色，所以只需要枚举到 $y-1$，就可以枚举到最后一个区间 $[c_{y-1},c_y]$。需要注意的是，因为用到了排序，所以离散化方法的时间复杂度为 $O(n\log n)$，其中 $n$ 为关键数值的数量。

离散化的过程中涉及二分操作，如果不使用lower_bound也可以自己写一个二分查找。以下是离散化中的二分查找的常见代码：

```
int bnrsch(int x) {
    int l = 1, r = n, mid;
    while (l < r) {
        mid = (l + r) / 2;
        if (c[mid] == x)
            return mid;
        if (c[mid] < x)
            l = mid + 1;
        else
            r = mid;
    }
    return r;
}
```

离散化也可以使用STL中的map方法来实现，代码如下：

```
#include <bits/stdc++.h>
using namespace std;
#define MAXN 20010
int n, a[MAXN], b[MAXN], f[MAXN * 2], dtop, ctop, d[MAXN * 2];
long long c[MAXN * 2];
long long ans;
map<int, int> Map;
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        scanf("%d%d", &a[i], &b[i]);
        d[++dtop] = a[i];
        d[++dtop] = b[i];
    }
    sort(d + 1, d + dtop + 1);
    for (int i = 1; i <= dtop; i++)
        if (d[i] != d[i - 1] || i == 1) {
            c[++ctop] = d[i];
            Map[d[i]] = ctop;
        }
    for (int i = 1; i <= n; ++i) {
        int x = Map[a[i]], y = Map[b[i]];
        for (int j = x; j < y; j++)
            f[j] = 1;
```

```
    }
    for (int i = 1; i < ctop; i++)
        if (f[i])
            ans += c[i + 1] - c[i];
    printf("%lld\n", ans);
    return 0;
}
```

### 例 2-7 程序自动分析(洛谷P1955,NOI2015)

有无数个变量 $x_1, x_2, x_3, \cdots$，现有 $n$ 个关系，每个关系有 $3$ 个参数 $i, j, e$ 当 $e=1$ 时关系为 $x_i = x_j$ ，当 $e=0$ 时关系为 $x_i \neq x_j$。问这 $n$ 个关系能否全部满足。

**数据范围：** $1 \leqslant i, j \leqslant 10^9, e \in \{1, 0\}, n \leqslant 10^5$，

**分析：**变量之间关系只有等于和不等于两种，等于关系之间是不会冲突的，不等于关系之间也是不会冲突的，所以需要考虑的只有等于关系和不等于关系之间的冲突。可以先用图染色或并查集的方式预处理相等关系，求出所有的相等连通块，再检查所有不等于关系，判断是否有不等于关系会发生冲突。

现在的问题是元素的数量太多，有 $10^9$ 个，无论是并查集还是图染色都无法承担。观察发现，这 $10^9$ 个变量一定不会全部用上。总共只有 $10^5$ 个关系，每个关系涉及两个变量，最多涉及 $2 \times 10^5$ 个变量，其余的变量和答案没有任何关系，可以直接忽略。

用**离散化**的方法，收集所有变量标号，排序去重，存到数组 $c$ 中，用下标 $i$ 代替标号 $c[i]$。这样使得变量数缩减到 2n个以内，从而可以用并查集或图染色来解决问题。具体代码如下：

```
#include <bits/stdc++.h>
using namespace std;
#define MAXN 100010
int n, btop, ctop, T;
struct node {
    int x, y, z;
} a[MAXN];
int b[MAXN * 2], c[MAXN * 2];
int fa[MAXN * 2];
int find(int x) { // 并查集部分
    if (fa[x] == x)
        return x;
    return fa[x] = find(fa[x]);
}
void merge(int x, int y) { // 并查集部分
    x = find(x);
    y = find(y);
    if (x != y)
        fa[x] = y;
}
int main() {
    scanf("%d", &T);
    while (T--) {
        scanf("%d", &n);
```

```
        btop = 0;
        ctop = 0;
        for (int i = 1; i <= n; i++) {
            scanf("%d%d%d", &a[i].x, &a[i].y, &a[i].z);
            b[++btop] = a[i].x; // 注意数组 b 的大小为 2 * n
            b[++btop] = a[i].y;
        }
        sort(b + 1, b + btop + 1); // 离散化过程
        for (int i = 1; i <= btop; i++)
            if (b[i] != b[i - 1])
                c[++ctop] = b[i];
        for (int i = 1; i <= n; i++) {
            a[i].x = lower_bound(c + 1, c + ctop + 1, a[i].x) - c;
            a[i].y = lower_bound(c + 1, c + ctop + 1, a[i].y) - c;
        }
        for (int i = 1; i <= ctop; i++)
            fa[i] = i;
        for (int i = 1; i <= n; i++) // 将具有相等关系的变量在并查集中合并
            if (a[i].z)
                merge(a[i].x, a[i].y);
        bool mk = true;
        for (int i = 1; i <= n; i++)
            if (!a[i].z)
                if (find(a[i].x) == find(a[i].y)) {
                    mk = false;
                    break;
                }
        printf("%s\n", mk ? "YES" : "NO");
    }
    return 0;
}
```

首先是离散化过程，先收集数据，然后排序去重。需要注意的是，用于收集和排序的两个数组 $c$ 和 $b$ 的大小都应该为关系数量 $n$ 的两倍，因为一个关系能提供两个关键数据，在最坏的情况下这些关键数据没有一个是重复的，因此两个数组的大小都应该设立为 $n$ 的两倍。

然后是应用离散化的过程，本质上是给涉及的所有变量（这里的变量指题目中的变量 $x_i$） 重新编号，这样在接下来的代码中就不再涉及离散化操作，可以直接把离散化的数据当作原始数据处理。

直接将原始数据替换不会影响答案，因为原先相等的标号现在仍然相等，而原先不等的标号仍然不等，因此替换前和替换后的变量编号是等效的。

### 例 2-8 过度种植(洛谷P1884,USACO2012年2月赛)

在平面直角坐标系中有 $n$ 个矩形，现在给出每个矩形的左上角 $(x_1, y_1)$ 和右下角 $(x_2, y_2)$ 问：这些矩形的覆盖面积是多少？重复覆盖的区域只计算一次。

数据范围 ：$n \leq 10^3, -10^8 \leqslant x_1, y_1, x_2, y_2 \leqslant 10^8$。

**分析：** 此题与 例 2-1（区间求和）类似，都是统计覆盖且不计重复，只不过从一维升级到了二维。二维空间的**离散化**可以对两个维度分别离散，也可以统一离散，两者的效率没有太大区别，可以忽略。为了便于代码书写，一般推荐统一离散。

处理覆盖的方法可以类比一维版本，用 `f[i][j]` 表示左上角为 $(c_i, c_j)$、右下角为 $(c_{i+1}, c_{j+1})$ 的矩形。但如果仍然以循环枚举数组元素来染色的方法会导致程序超时。总共有 $10^3$ 个矩形，而离散化后，在最坏情况下，每个矩形要枚举接近 $4000 \times 4000$ 个数组元素（达不到这个数量，但规模相近），此时时间就非常紧张。

这个方法被误认为可以通过本题，是因为只考虑到要枚举 $4000 \times 4000$ 个数组元素，而没有考虑到这一举动要进行 1000 次。因此，使用差分优化时间复杂度即可。具体代码如下：

```
#include <bits/stdc++.h>
using namespace std;
#define MAXN 4010
int n, btop, ctop;
int a[MAXN][4];
int b[MAXN], c[MAXN];
int f[MAXN][MAXN];
map<int, int> Map;
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
        for (int j = 0; j < 4; j++) {
            // a[i][0]和a[i][1]表示左上角,a[i][2]和a[i][3]表示右下角
            scanf("%d", &a[i][j]);
            b[++btop] = a[i][j];
        }
    // 离散化
    sort(b + 1, b + btop + 1);
    b[0] = -1000000007;
    for (int i = 1; i <= btop; i++)
        if (b[i] != b[i - 1]) {
            c[++ctop] = b[i];
            Map[b[i]] = ctop;
        }
    // 使用离散化给关键数值标号
    for (int i = 1; i <= n; i++)
        for (int j = 0; j < 4; j++)
            a[i][j] = Map[a[i][j]];
    // 进行矩形覆盖
    for (int k = 1; k <= n; k++) {
        for (int i = a[k][0]; i < a[k][2]; i++) // 枚举矩形里的行
            f[i][a[k][3]]++, f[i][a[k][1]]--; // 对每一行进行差分意义下的区间加
    }
    // 求前缀和,还原出原序列
    for (int i = 1; i < ctop; i++)
        for (int j = 1; j < ctop; j++)
            f[i][j] += f[i][j - 1];
    // 统计答案
    long long ans = 0;
    for (int i = 1; i < ctop; i++)
        for (int j = 1; j < ctop; j++)
            if (f[i][j])
                ans += (long long)(c[i + 1] - c[i]) * (c[j + 1] - c[j]);
    cout << ans << endl;
    return 0;
}
```

## 2.4 课后习题与实验

### 习题 1 领地选择(洛谷P2004)

小 $Z$ 的领地为包含 $N \times M$ 个地块的矩形，每个地块都有自己的价值 $a_{i,j}$,。首都被认为一个占地 $C \times C$ 的正方形。小 $Z$ 希望寻找到一个合适的位置，使得首都所占领的位置的土地价值和最高。

**数据范围：** $1 \leqslant N, M \leqslant 10^3, 1 \leqslant C \leqslant \min(N, M)$，$\sum_{i=1}^{N} \sum_{j=1}^{M} a_{i,j}$ 在int范围内。

### 习题 2 蛋糕切块(洛谷P3017,USACO2011年12月赛)

小 $B$ 烘焙了一块巧克力蛋糕。这块蛋糕是由 $R \times C (1 \leqslant R, C \leqslant 5 \times 10^2)$ 块小的巧克力蛋糕组成的。第 $i$ 行第 $j$ 列的蛋糕有 $N_{i,j} (1 \leqslant N_{i,j} \leqslant 4 \times 10^3)$ 块巧克力碎屑。

小 $B$ 想把蛋糕分成 $A \times B$ 块 $(1 \leqslant A \leqslant R, 1 \leqslant B \leqslant C)$，分给 $A \times B$ 个人。蛋糕先水平切 $A-1$ 刀（只能切沿整数坐标切），分成 $A$ 块。然后再把剩下来的每一块独立地切 $B-1$ 刀，也只能切沿整数坐标切。其他 $A \times B - 1$ 人每人选一块，留下一块给小 $\mathbf{B}$。其他人只会留给小 $B$ 巧克力碎屑最少的那块。求出小 $B$ 最优情况下会获得多少巧克力碎屑。

例如，考虑一个 $5 \times 4$ 的蛋糕，上面的碎屑分布如图 2-10(a) 所示。小 $B$ 必须把蛋糕切成 $4$ 条，每条分成两块。小 $B$ 最佳切蛋糕方式如图 2-10(b) 所示，此时他可以获得 $3$ 块巧克力碎屑。

### 习题 3 城际高铁(洛谷P3406)

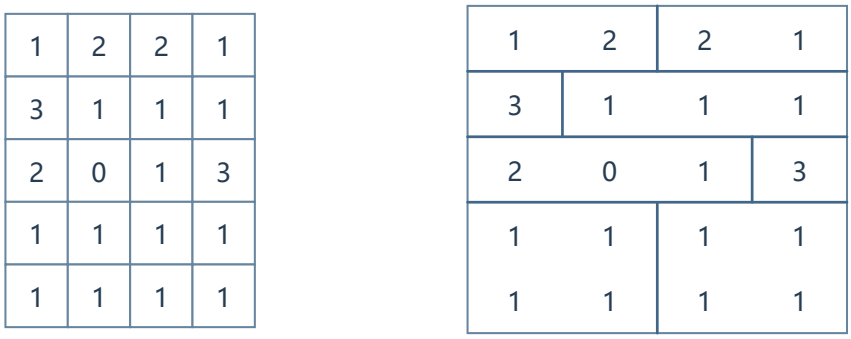



图2-10 第2章习题2样例图

铁路经过 $N$ 个城市，每个城市都有一个站。乘车每经过两个相邻的城市之间（方向不限），必须单独购买这一小段的车票。第 $i$ 段铁路连接了城市 $i$ 和城市 $i+1 (1 \leqslant i < N)$。第 $i$ 段铁路购买纸质单程票需要 $A_i = \overline{\pi}$。铁路公司为了方便乘客，推出了IC卡。

对于第 $i$ 段铁路，需要花 $C_i$ 元的工本费购买一张IC卡，然后乘坐这段铁路一次就只要扣 $B_i (B_i < A_i)$ 元。IC卡可以提前购买，工本费不能退，也不能购买车票。每张卡都可以充值任意数额。对于第 $i$ 段铁路的IC卡，无法乘坐其他段的铁路的车。

小 $U$ 现在需要出差去 $M$ 个城市。他从城市 $P_1$ 出发，分别按照 $P_1, P_2, P_3, \cdots, P_M$ 的顺序访问各个城市，可能会多次访问一个城市，相邻访问的城市位置不一定相邻，而且不会是同一个城市。现在他希望知道，出差结束后至少会花掉多少钱，包括购买纸质车票、买IC卡和充值的总费用。

**数据范围：** $M,N\leqslant10^5,A_i,B_i,C_i\leqslant10^5$。

### 习题 4 借教室(洛谷P1083,NOIP2012提高组)

你需要帮忙处理接下来 $n$ 天的借教室信息，其中第 $i$ 天学校有 $r_i$ 个教室可供租借。共有 $m$ 份订单，每份订单用 $3$ 个正整数描述，分别为 $d_j,s_j,t_j$，表示某租借者需要从第 $s$ 天到第 $s_jt_j$ 天租借教室（包括第 $s_j$ 天和第 $t_j\mathcal{F}$） ，每天需要租借 $d_j$ 个教室。

按照订单的先后顺序依次为每份订单分配教室。如果在分配的过程中遇到一份订单无法完全满足，则需要停止教室的分配，通知当前申请人修改订单。无法满足是指从第 $s_j$ 天到第 $t_j$ 天中有至少一天剩余的教室数量不足 $d_j$ 个。现在需要知道，是否会有订单无法完全满足。如果有，需要通知哪一个申请人修改订单。

**数据范围：** $1\leqslant n,m\leqslant10^6,0\leqslant r_i,d_i\leqslant10^9,1\leqslant s_i\leqslant t_i\leqslant n$。

### 习题 5 面向前方(洛谷P2882,USACO2007年3月赛)

$n$ 头牛排成一列，每头牛的朝向为向前或向后。为了让所有牛都面向前方，农夫每次可以将 $k$ 头连续的牛转向 $(1\leqslant k\leqslant n)$，求操作的最少次数 $m$ 和对应的最小 $k_c$

**数据范围：** $1\leqslant n\leqslant5000$。

### 习题 6 增减序列(洛谷P4552,Poetize6)

给定一个长度为 $n$ 的数列 $a_1,a_2,\cdots,a_n$，每次可以选择一个区间 $[l,r]$，使这个区间内的数都加 $1$ 或者都减 $1$。

请问至少需要多少次操作才能使数列中的所有数都一样，求出在保证最少次数的前提下，最终得到的数列有多少种。

**数据范围：** $n\leqslant10^5,0\leqslant a_i\leqslant2^{31}$。

### 习题 7 奶牛排队(洛谷P3029,USACO2011年11月赛)

有 $n$ 头奶牛站在数轴上，每头牛都有一个坐标和一种花色，二者都是一个整数。现在要给它们拍照片，要求把所有花色的奶牛都拍到，问至少拍多宽的范围才能达到要求。所有的奶牛都不喜欢和其他牛站在一起。

**数据范围：** $n\leqslant5\times10^4$，所有的坐标和花色都不超过 $10^9$。

### 习题 8 天际线(洛谷P1904)

有 $n$ 个矩形被放置在 $X$ 轴上，给出所有的矩形高度和宽度，请找出它们的轮廓线（只需要输出轮廓线的所有转折点）。

**数据范围：** $1\leqslant n\leqslant5\times10^3$，矩阵高度和宽度不大于 $10^4$

### 习题 9 混乱的排序(洛谷P4375,USACO2018年公开赛)

小贝最喜欢的算法是“冒泡排序”。这是她最初的对长度为 $N$ 的数组 $A$ 进行排序的算法实现（伪代码）。

```
sorted =false
while (not sorted) :
sorted = true
moo
for i = 0 toN-2:
if A[i+1] < A[i]:
swap A[i], A[i+1]
sorted = false
```

显然，代码中的moo指令的作用只是输出moo。奇怪的是，小贝看上去执着于在她的代码中的不同位置使用这条语句。在用若干个数组测试了代码之后，小贝发现：大的元素很快就会被拉到数组末尾，然而小的元素需要很长时间“冒泡”到数组的开头。小贝试着修改了她的代码，使代码在每次循环中向前再向后各扫描一次，从而无论是大的元素还是小的元素在每一次循环中都有机会被拉较长的一段距离。其代码如下：

```
sorted =false
while (not sorted):
sorted = true
moo
for i = 0 toN-2:
if A[i+1] < A[i]:
swap A[i], A[i+1]
for i = N-2 downto 0:
if A[i+1] < A[i]:
swap A[i], A[i+1]
for i = 0 to N-2:
if A[i+1] < A[i]:
sorted = false
```

给定一个输入数组，请预测小贝修改后的代码会输出多少次moo。

**数据范围：** $N \leqslant 10^5$，被排序的所有数都不超过 $10^9$。

### 习题 10 博弈游戏(洛谷P5937,CEOI1999)

两个人在玩游戏：其中一个人写一个长度为 $n$ 的 $0-1$ 序列，另一个人选取其中的一段区间，并问写序列的人这段区间的 $1$ 的个数是奇数还是偶数，总共问了 $m$ 次。给出所有的提问和回答，请问前多少个回答是一定没有问题的?一个回答有问题的条件为：存在一个序列，满足此回答之前的所有回答，而不满足此回答。

**数据范围：** $1 \leqslant n \leqslant 10^9, m \leqslant 5 \times 10^3$。

# 第3章 分治与倍增

如果想知道一个国家的人口数量，就需要进行人口普查。让每一个省份都去统计本省有多少人，然后将各省人口累加起来，就可以获得全国的人口数量。而要想知道某一个省的人口数量，可以让省里的每一个城市统计本市有多少人，然后将各市人口累加起来，就可以获得这个省的人口数量……依此类推，层层细分，最后统计一个村子或者一个小区有多少人，这个任务就很简单了。把一个复杂的问题细分成若干结构相同但规模更小的子问题，然后将每个子问题的解合并起来，就得到了复杂问题的解，这就是本章将会介绍的**分治**策略。

在本章的最后还将介绍**倍增**算法。将一个较大的数字分为若干 $2^i$ 的和，同时维护所有长度为 $2^i$ 的区间，可以解决一些特定的问题，如区间求最值的问题。本章思维导图如图 3-1 所示。




图3-1 本章思维导图

## 3.1 归并排序与快速幂

### 例 3-1 快速排序(洛谷P1177)

输入 $n(n\leqslant 100000)$ 个数字 $a_1,a_2,\cdots,a_n(a_i\leqslant 10^9)$，将其从小到大排序后输出。

**分析：**在《基础篇》中介绍了快速排序等算法可以解决这个问题，这里介绍一种新的算法归并排序。要学会归并排序，首先要会归并。考虑一个子问题：给定两个有序的序列 $a_1a_2,\cdots,a_n$ 和 $b_1,b_2,\cdots,b_n$，支持两个元素间的比较操作，要求把两个序列合并成一个序列，使得合并出的这个序列是有序的。

算法的过程很简单，维护两个位置 $j$ 和 $k$，代表当前考虑数组 a 的第 $j$ 个元素与数组 $b$ 的第 $k$ 个元素。如果 $a_j < b_k$，则在答案数组当前位置填充 $a_j$，同时让 $j$ 自增。如果 $a_j \geqslant b_k$，则在答案数组当前位置填充 $b_k$，同时让 $k$ 自增。注意到，如果 $a$ 和 $b$ 两个数组中有一个被用完了，可以直接把另一个数组剩下的部分接到答案数组最后面。

```
void merge(int a[], int b[], int c[]) {
    for (int i = 1, j = 1, k = 1; i <= n + n; i++) {
        if (j == n + 1)
            c[i] = b[k++]; // 如果数组a被用完了,则直接填充数组b的元素
        else if (k == n + 1)
            c[i] = a[j++]; // 如果数组b被用完了,则直接填充数组a的元素
        else
            c[i] = a[j] < b[k] ? a[j++] : b[k++]; // 否则哪个小用哪个
    }
}
```

例如，有两个有序的数组 $a = \{1,3,7,8\}, b = \{2,4,6,9\}$，现在要对这两个数组进行归并，如图 3-2 所示。

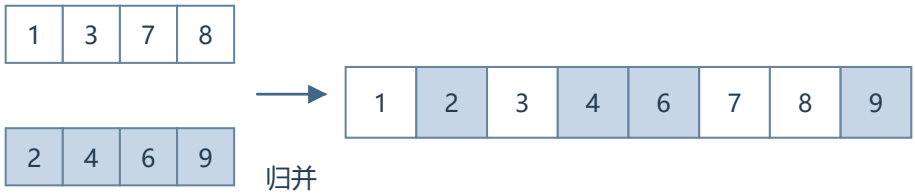



图3-2 归并两个有序数组

在学会归并之后，要对一个长为 $n$ 的序列进行排序，可以考虑采用**分治**的思想来解决：如果 $n = 1$，这个序列是自然有序的，故不用进行排序—这就是可以直接解决的子问题；否则，将序列分为两个长 $\frac{n}{2}$ 的子序列（如果 $n$ 是奇数，则涉及取整的问题，这里默认不考虑细节），对这两个子序列分别递归地进行排序—这是把一个复杂的问题转换为若干个简单一些的问题，然后递归下去解决简单一些的问题。

当两个子序列有序后，对这两个子序列进行归并，使当前这个长 $n$ 的序列有序—这就是当每个子问题都处理完之后，合并子问题的答案得到原问题的答案。

归并排序的时间复杂度为 $T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$。

```
void merge_sort(int a[], int l, int r) {
    if (l == r)
        return; // 如果序列长度为1,则不用排序,可以直接返回
    int mid = (l + r) >> 1;
    merge_sort(a, l, mid), merge_sort(a, mid + 1, r); // 递归到更小的子问题,即
    // 分治
    for (int i = l, j = l, k = mid + 1; i <= r; i++) { // 归并
        if (j == mid + 1)
            t[i] = a[k++];
        else if (k == r + 1)
            t[i] = a[j++];
        else
            t[i] = a[j] < a[k] ? a[j++] : a[k++];
    }
    for (int i = l; i <= r; i++)
        a[i] = t[i];
}
```

一般而言，分治是将原问题转化为规模更小但结构相同的子问题（直到问题规模足够小后可以直接处理），对每个子问题递归处理；当每个子问题都处理完之后，合并子问题的答案得到原问题答案的过程。分治策略往往通过函数递归的方式实现，这在《基础篇》中已经介绍过了。

### 例 3-2 逆序对(洛谷P1908)

对于给定的一段正整数序列 $a$，逆序对是序列中 $a_i > a_j$ 且 $i < j$ 的有序对。求一个长为 $n$ 的序列的逆序对个数，其中 $1 \leqslant n \leqslant 5 \times 10^5$。

**分析：**对于这个问题，可以在归并排序的过程中同时求出序列的逆序对数。

如果 $n = 1$，这个序列的逆序对个数自然是 0—这就是可以直接解决的子问题；否则，将序列分为两个长 $\frac{n}{2}$ 的子序列，对这两个子序列分别递归地求出其内部的逆序对一这是把一个复杂的问题转换为若干个简单一些的问题，然后递归下去解决简单一些的问题。

当递归计算了两个子序列内部的逆序对数后，考虑怎么合并这两个子序列。可以发现还需要维护前一个子序列中选一个元素，后一个子序列中选一个元素，如图 3-3 所示，这样构成的逆序对个数这就是当每个子问题都处理完之后，合并子问题的答案得到原问题的答案。

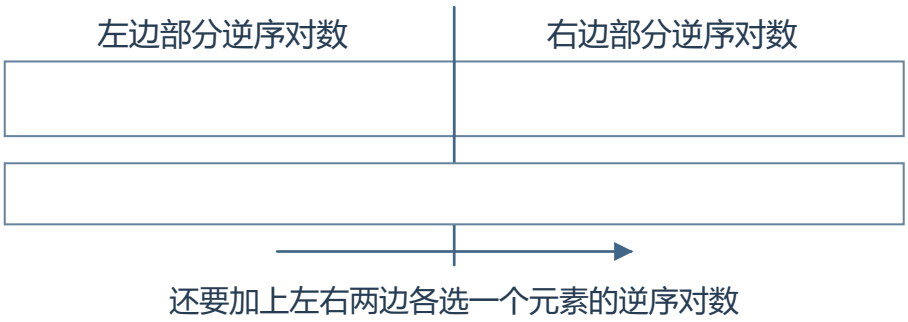



图3-3 逆序对的分治

那么如何算出这两个子序列对答案的贡献呢？由于在序列位置中，前一个子序列中的元素一定在后一个子序列中的元素前面，所以逆序对的第 2 个条件： $i < j$ 已经自然满足了，只需要 $a_i > a_j$ 也满足即可。也就是说，对前一个子序列中的每个位置 $i$，需要统计出在后一个子序列中有多少位置 $j$ 满足

$a_i > a_j$ 即可。

回忆一下归并排序的归并过程，对两个有序数组 $a$ 和 $b$ 归并的时候，如果这次比较之后在答案数组中放入的元素是 $a_i$ 和 $a_i$ 做比较的元素是 $b_j$，那么一定有 $b_1, b_2, \cdots, b_{j-1}$ 均比 $a_i$ 小，而 $b_j, b_{j+1}, \cdots, b_n$ 均比 $a_i$ 大。

所以在归并排序的过程中，每次在答案数组中放入 $a_i$ 时，就有 $j$ 个逆序对产生，因此答案加上 $j$。这样就可以边归并排序边求出整个序列的逆序对数了。

时间复杂度和归并排序一样，为 $O(n \log n)$。代码如下：

```
#include <iostream>
#include <cstdio>
#define MAXN 500010
using namespace std;
int n, a[MAXN], t[MAXN];
long long ans;
void merge_sort(int a[], int l, int r) {
    if (l == r)
        return; // 如果序列长度为1,则不用排序,可以直接返回
    int mid = (l + r) >> 1;
    merge_sort(a, l, mid), merge_sort(a, mid + 1, r); // 递归到更小的子问题,即分治
    for (int i = l, j = l, k = mid + 1; i <= r; i++) // 归并
        if (j == mid + 1)
            t[i] = a[k++];
        // 注意这里不要重复计算贡献
        else if (k == r + 1) {
            t[i] = a[j++];
            ans += k - mid - 1;
            // 排序后a[j]这个元素比[mid+1,k-1]里面所有元素都大,会产生k-mid-1个逆序对
        } else if (a[j] <= a[k]) {
            t[i] = a[j++];
            ans += k - mid - 1;
            // 排序后a[j]这个元素比[mid+1,k-1]里面所有元素都大,会产生k-mid-1个逆序对
        } else
            t[i] = a[k++];
    for (int i = l; i <= r; i++)
        a[i] = t[i];
}
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
        scanf("%d ", &a[i]);
    merge_sort(a, 1, n);
    printf("%lld\n", ans);
    return 0;
}
```

### 例 3-3 火柴排队(洛谷P1966,NOIP2013提高组)

涵涵有两盒火柴，每盒装有 $n$ 根火柴，每根火柴都有一个高度。现在将每盒中的火柴各自排成一列，同一列火柴的高度互不相同，两列火柴之间的距离定义为 $\sum (a_i - b_i)^2$，其中 $a_i$ 表示第 $1$ 列火柴中第 $i$ 个火柴的高度 $\nabla, b_i$ 表示第 $2$ 列火柴中第 $i$ 个火柴的高度。

每列火柴中相邻两根火柴的位置都可以交换，请通过交换使得两列火柴之间的距离最小。请问得到这个最小的距离，最少需要交换多少次？请输出这个最小交换次数对 $99999997$ 取模的结果。

**分析：**首先观察要最小化的式子 $\sum\Omega(a_i\Omega-b_i)\Omega^2$，由于 $(a_i-b_i)^2=a_i^2+b_i^2-2a_ib_i$，所以有 $\sum(a_i-b_i)^2=\sum\left(a_i^2+b_i^2-2a_ib_i\right)=\sum a_i^2+\sum b_i^2-2\sum a_ib_i$。

由于 $\sum a_i^2$ 和 $\sum b_i^2$ 无论如何排列火柴都不会改变，所以问题转化为最大化 $\sum a_ib_i$。可以证明，把两个序列排序后，对每个 $k$，将第 $k$ 小的ak元素与第 $k$ 小的 $a_kb_k$ 元素配对，这样得到的 $\sum a_ib_i$ 是最大的，请读者自行完成证明（可以利用排序不等式证明）。

然后考虑如何计算答案，可以发现在第 1 列中交换了第 $i$ 根火柴和第+1 根火柴，和第 2 列中交换了第 $i$ 根火柴和第+1 根火柴，这两个操作是等价的，所以只考虑对一列进行交换。

首先排序，然后计算出两个数组 $c_i$ 和 $d_i,c_i$ 表示第 1 列中第 $i$ 个元素在第 1 列中是第 $c_i$ 大的， $d_i$ 表示第 2 列中第 $i$ 个元素在第 2 列中是第 $d_i$ 大的。由此，问题便转化为至少需要交换多少个相邻元素，使得 $c_i$ 序列变成 $d_i$ 序列。

这里由于两个数组都是一个排列，所以可以先做一个映射，将 $d_i$ 映射成 $i$，这样数组 $d$ 就变成 1，2，…，$n$ 这样的排列了。对 $c_i$ 同时进行这样的映射，于是问题便转化为每次可以交换相邻两个位置，求至少要交换多少次来排序一个序列。

可以这样考虑：

1. 如果当前序列中没有一个相邻的逆序对，即（$i$，$i+1$）满足 $a_i>a_{i+1}$，则序列已经有序了，不用继续交换了。
2. 如果当前序列中有一个相邻的逆序对，那么交换这个相邻的逆序对，会让序列中的逆序对数恰好减少 1。
3. 每次交换两个相邻的元素，最多能减少序列中一个逆序对。

由上述性质可以发现，每次操作最多能减少一个逆序对，而且一定存在一个方案使得逆序对减少了一个，故最少减少一个逆序对，所以操作次数就是序列中的逆序对数。

最后使用归并排序求序列中的逆序对即可，时间复杂度为 $O(n\log n)$。

### 例 3-4 快速幂、取余运算(洛谷P1226)

给出三个整数 $b,p,k$，求 $b^p \bmod k$。数据范围在 $[0,2147483647]$ 中。

**分析：**考虑如何**分治**来求解。

1. 如果 $p=0$，答案就是 $b^0 \bmod k=1 \bmod k$。
2. 如果 $p=2n$，即 $p$ 是一个偶数，那么可以递归求解 $b^n$ modk，然后将求得的结果平方，即 $b^n\times b^n=b^{2n}=b^p$。
3. 如果 $p=2n+1$，即 $p$ 是一个奇数，那么可以递归求解 $b$” modk，然后将求得的结果平方，再乘上 $b$，即 $b^n\times b^n\times b=b^{2n+1}=b^p$。

例如，$3^{25}=(3^{12})^2\times3=((3^6)^2)^2\times3=(((3^3)^2)^2)^2\times3=((((3^1)^2\times3)^2)^2)^2\times3$。这里省

下了取余的步骤。可以发现时间复杂度和二分搜索类似，每次递归求解时 $p$ 会减半，所以时间复杂度是 $O(\log_2 p)$。完整代码如下：

```
#include <iostream>
#include <cstdio>
using namespace std;
long long b, p, k;
long long power(long long y, long long x) {
    if (x == 0)
        return 1 % k; // 0次方直接返回1%k,这里取模是为了避免k=1的情况
    // 出错
    long long s = power(y, x / 2); // 求 b^x, 分治下去求 b^(x/2)
    if (x % 2 == 0)
        return s * s % k;     // x是偶数
    return s * s % k * y % k; // x是奇数
}
int main() {
    scanf("%lld%lld%lld", &b, &p, &k);
    printf("%lld^%lldmod %lld=%lld", b, p, k, power(b % k, p));
    return 0;
}
```

采用上述写法，递归函数调用 $\log_2 p$ 次，常数较大。通常快速幂用非递归的实现方式，代码如下：

```
#include <iostream>
using namespace std;
int b, p, k;
long long power(long long y, long long x) {
    if (x == 0)
        return 1 % k;
    long long sum = 1 % k;
    while (x) {
        if (x & 1)
            sum = sum * y % k;
        y = y * y % k;
        x >>= 1;
    }
    return sum;
}
int main() {
    scanf("%d%d%d", &b, &p, &k);
    printf("%d^%dmod %d=%lld", b, p, k, power(b % k, p));
    return 0;
}
```

### 例 3-5 麦森数(洛谷P1045,NOIP2003普及组)

输入 $P(1000 < P < 3100000)$，计算 $2^P - 1$ 的位数和最后 $500$ 位数字（用十进制高精度数表示）。

**分析：** 先求 $2^P - 1$ 的十进制位数。对于正整数 $P$，$2^P$ 不是 $10$ 的整数次幂，因此减去 $1$ 不会改变位数。由 $\log_{10}(2^P) = P\log_{10} 2$ 可知，位数为 $\lfloor P\log_{10} 2 \rfloor + 1$。

C++ 的 `<cmath>` 库提供 `log10(x)`，用于计算以 $10$ 为底的对数。例如，`log10(130)` 的值约为 $2.11394$。本题可直接调用该函数，精度足够。

然后考虑如何求 $2^P - 1$ 的后 500 位？可以使用高精度乘法，只保留末尾的 500 位，这样一次乘法的时间不至于太长，然后对 $2^P$ 使用快速幂的方法即可。

## 3.2 其他序列上的分治

### 例 3-6 最大子段和(洛谷P1115)

给出一个长度为 $n$ 的序列 $\{a_i\}$，选出其中连续且非空的一段使得这段和最大。

**数据范围：** $1 \leqslant n \leqslant 2 \times 10^5, -10^4 \leqslant a_i \leqslant 10^4$.

**分析：**还是考虑使用**分治**来解决这个问题。每次将序列从中间分成两半，计算经过中点（即分治中线）的最大子段，然后对两边都递归下去计算答案。

如图 3-4(a) 所示，如果最大子段是跨过中线的，则在这一层统计其答案；如果最大子段没有跨过中线，则对两边递归计算答案的过程会统计其答案，所以这个做法是不会漏掉任何一种情况的。

问题转化为，如何统计经过序列上一个点的所有区间里面最大的一个。这里其实就可以用贪心策略了，即由于一定要经过一个点，因此以这个点为开始向两边走，各取最大的一段拼起来，则一定是最优的，如图 3-4(b) 所示。

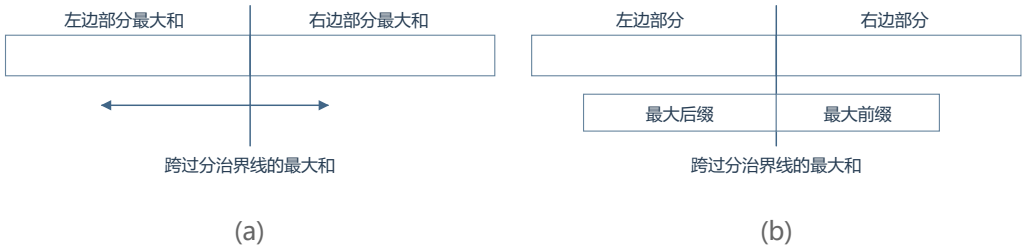



图3-4最大子段和的分治

于是在这一层可以直接从这个点开始向两边遍历每个元素，并取两个最大的前缀或者后缀和相加，再对这一层的答案取最大值即可。

可以列出和运行时间有关的等式 $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$，根据主定理解得时间复杂度为 $O(n \log n)$。

```
#include <iostream>
#include <cstdio>
#define MAXN 200010
#define INF 100000
using namespace std;
int n, a[MAXN], ans = -INF;
void dc(int l, int r) {
    if (l == r)
        ans = max(ans, a[l]);
    // 当区间长度为1的时候,只存在一种子段,用其更新答案
    else {
        int mid = (l + r) >> 1;
        dc(l, mid);                 // 分治计算[1,mid]
        dc(mid + 1, r);             // 分治计算[mid+1, r]
        int pre = -INF, suf = -INF; // 由于不能选空子段,这里设一个-INF
        for (int i = mid, j = 0; i >= l; i--) { // 计算最大的前缀
```

```
            j += a[i];
            pre = max(j, pre);
        }
        for (int i = mid + 1, j = 0; i <= r; i++) { // 计算最大的后缀
            j += a[i];
            suf = max(j, suf);
        }
        ans = max(ans, pre + suf);
        // 最大的前缀和最大的后缀拼起来,得到这一层的最大子段,更新答案
    }
}
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
        scanf("%d", &a[i]);
    dc(1, n);
    cout << ans << endl;
    return 0;
}
```

上面说了这么多，那么什么样的问题可以使用分治呢？

1. 可以把原问题划分为多个无关的子问题，这里划分一般是人为给定的。在简单的题目中一般会考虑将当前问题划分为两个规模基本相当的子问题，也就是这章经常讲的取中间值 $mid = \frac{l+r}{2}$

2. 子问题的答案可以高效合并，也就是知道了子问题的答案后可以高效计算其对全局的答案会产生什么影响。

3. 当前问题的答案可以高效求出，在这里“高效”的复杂度一般为 $O(n)$ 或 $O(n \log n)$，也就是说和扫一遍或者排个序的复杂度差不多。这里可以每次重新计算经过分治中线的答案，然后与子问题的答案合并（如最大子段和），也可以直接由子问题的答案合并上来（如快速幂）。

如果满足上述这 3 条，该问题就可以使用分治来解决了。

在分治算法中，由于子问题和原问题类型相同，时间复杂度经常可以写成形如 $T(n) = aT(n/b) + f(n)$ 的递归式，其中 $a$ 和 $b$ 为正整数，边界条件为当 $n=O(1)$ 时 $T(n) = O(1)$，此时可用主定理得到原问题的渐进时间复杂度。主定理的具体内容如下：

1. 当 $f(n) = O(n^{\log_b a} \log^c n)$ 时， $T(n) = O(n^{\log_b a} \log^{c+1} n)$
2. 当 $f(n) = O(n^{-\varepsilon+\log_b a})$， $\varepsilon > 0$ 时， $T(n) = O\left(n^{\log_b a}\right)$。
3. 当 $f(n)=O\left(n^{c_1} \log^{c_2} n\right), c_1 > \log_b a$ 时， $T(n) = O(n^{c_1} \log^{c_2} n)$

对于更复杂的情况，可以猜测复杂度并用代入法检验，也可以用级数求和的方式计算，由于内容限制，这里不做更多说明。以下给出一些使用主定理分析复杂度的例子：

1. 归并排序的时间复杂度是 $T(n) = 2 \times T(n/2) + O(n)$，此时 $f(n) = O(n), a = 2, b = 2$

$\log_b a = 1, c = 0$，所以满足第 1 种情况的主定理，此时时间复杂度为
$T(n) = O\left(n^{\log_b a} \log^{c+1} n\right) = O(n \log n)$。

2. 对 $T(n) = 2T(n/2) + O(\sqrt{n})$，有 $a = b = 2$，$\log_b a = 1$，且 $f(n) = O(n^{1-1/2})$。由主定理得到 $T(n) = O(n)$。
3. 对 $T(n) = 2T(n/2) + O(n^2 \log n)$，有 $f(n) = O(n^{1+\log_b a} \log n)$。由主定理得到 $T(n) = O(n^2 \log n)$。

## 3.3 倍增与ST表

本节将介绍运用**倍增**思想求区间最值的ST表（Sparse-Table）算法，及倍增思想的运用。

ST表算法可用于解决以下问题：给出一个 $n$ 个元素的数组 $a_1, a_2, \cdots, a_n$，支持查询操作计算区间 $[l, r]$ 的最小值（最大值）。这种问题被称为区间最值查询问题（RangeMinimum/MaximumQuery，简称RMQ问题）。预处理的时间复杂度为 $O(n \log_2 n)$，预处理后数组 $a$ 的值不可以修改，一次查询操作的时间复杂度为 $O(1)$。

### 例 3-7 平衡的队列(洛谷P2880,USACO2007年1月赛)。

有一个包含 $n$ 个数的序列 $h_i$，有 $q$ 次询问，每次询问 $h_{a_i}, h_{a_i+1}, \cdots, h_{b_i-1}, h_{b_i}$ 中最大值与最小值的差。

**数据范围：** $1 \le n \le 5 \times 10^4$，$1 \le q \le 1.8 \times 10^5$，$1 \le h_i \le 10^6$，$a_i \le b_i$。

**分析：** 题目要求最大值和最小值的差难以直接求出，通常需要分别求解最大值和最小值。最直接的做法是每次遍历区间中的每一个数，记录最大值和最小值。这样可以求出正确答案，但是效率低下，时间复杂度高达 $O(nq)$，无法通过本题。

之所以这样做效率如此低下，是因为所有询问区间之间可能有着大量的重叠，这些重叠部分被多次遍历到，因此产生了大量重复。如果可以通过预处理得到一些区间的最小值，再通过这些区间拼凑每一个询问区间，就可以提高效率。

预处理**前缀和**可以拼凑出任意区间的和，借此类比，可惜无法照搬。其问题在于区间和是可以从一个区间中删去一部分的，而区间最值不行，所以只能用小区间去拼出大区间。如何选择预处理的区间成为关键，选择的区间既要能够拼出任意区间，数量上又不能太多，而且预处理和查询都要高效。

预处理以每一个位置为开头，长度为 $2^0, 2^1, \cdots, 2^{[\log_2 n]}$ 的所有区间最值。具体不妨只考虑最大值，用 $f[i,j]$ 表示 $h_i, h_{i+1}, \cdots, h_{i+2^j-2}, h_{i+2^j-1}$ 中的最大值，用递推的方式计算所有的 $f$，转移为 $f[i,j] = \max\left(f[i,j-1], f[i+2^{j-1}, j-1]\right)$。

计算所有的 $f$ 的过程为预处理，预处理时间复杂度 $O(n \log_2 n)$。

例如，`f[2,3]` 的转移如图 3-5(a) 所示，因此可以得到所有的转移，如图 3-5(b) 所示。

```
void init() {
    for (int i = 1; i <= n; i++)
        f[i][0] = h[i];
    for (int j = 1; (1 << j) <= n; j++)
        for (int i = 1; i <= n - (1 << j) + 1; i++)
            f[i][j] = max(f[i][j - 1], f[i + (1 << j - 1)][j - 1]);
}
```

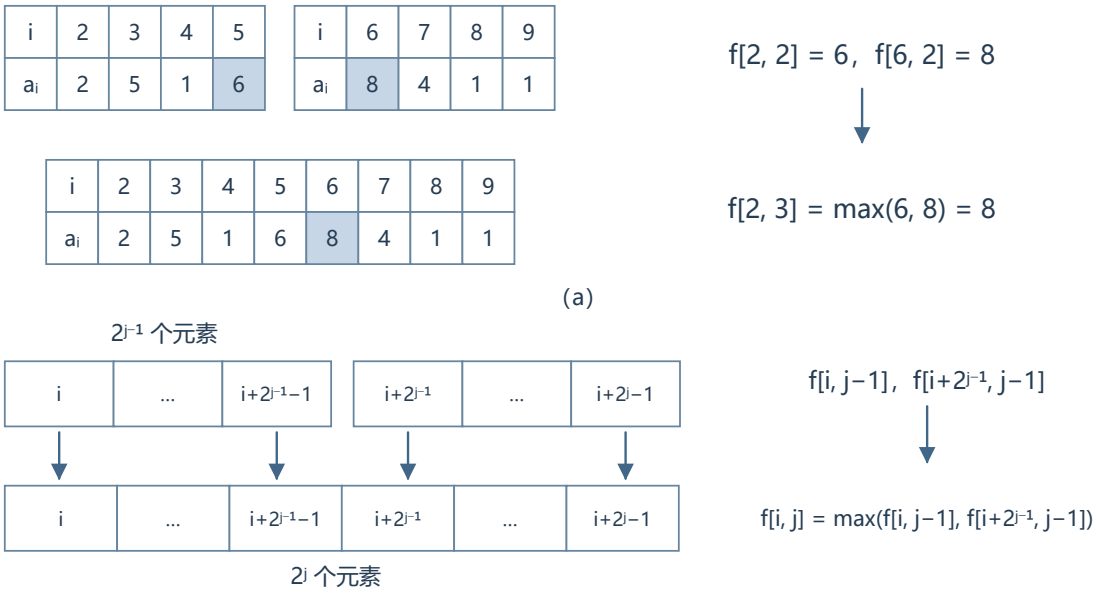



图3-5ST表算法查询预处理过程

接下来解决查询的问题，设需要查询最大值的区间为 $[l,r]$。如前文所述，记区间长度为 $L$，则该区间可以拆分为 $O(\log_2 L)$ 的小区间。将 $L$ 二进制拆分，从 $l$ 开始向后跳，每次跳拆出来的一个 2 的幂个数，就可以拼出整个区间。单次查询时间复杂度为 $O(\log_2 n)$。具体代码如下：

```
int RMQ(int l, int r) {
    int L = r - l + 1, ans = -INF;
    for (int i = 0, j = l; (1 << i) <= L; i++)
        if ((L >> i) & 1) {
            ans = max(ans, f[j][i]);
            j += (1 << i);
        }
    return ans;
}
```

查询区间最值时，区间合并的过程允许重叠，因此只需要找到两个长度为 $2^k(k \in \lfloor \log_2 n \rfloor)$ 的区间并集为 $[l,r]$。令 $k$ 为满足 $2^k \leqslant r-l+1$ 的最大整数，区间 $[l,l+2^k-1]$ 和区间 $[r-2^k+1,\ r]$ 合并起来覆盖了需要查询的区间 $[l,r]$，合并过程如图 3-6 所示。

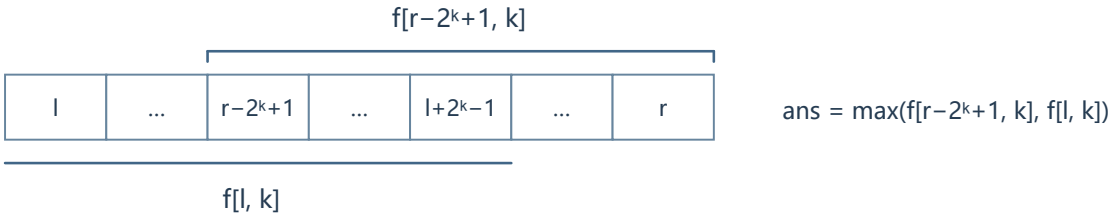



图3-6 ST表算法查询过程

查询部分具体代码如下：

```
int RMQ(int l, int r) {
    int k = log_2[r - l + 1]; // k 如题意,通过预处理将查询复杂度降至o(1)
    return max(f[l][k], f[r - (1 << k) + 1][k]);
}
```

本题完整代码：

```
#include <bits/stdc++.h>
using namespace std;
#define MAXN 50010
int n, q, h[MAXN], log_2[MAXN], rmax[MAXN][18], rmin[MAXN][18];
void init() {
    for (int i = 1; i <= n; i++)
        rmax[i][0] = rmin[i][0] = h[i];
    for (int j = 1; (1 << j) <= n; j++)
        for (int i = 1; i <= n - (1 << j) + 1; i++) {
            rmax[i][j] = max(rmax[i][j - 1], rmax[i + (1 << j - 1)][j - 1]);
            rmin[i][j] = min(rmin[i][j - 1], rmin[i + (1 << j - 1)][j - 1]);
        }
}
int query_max(int A, int B) { // 求区间[A, B]的最大值
    int x = log_2[B - A + 1];
    return max(rmax[A][x], rmax[B - (1 << x) + 1][x]);
}
int query_min(int A, int B) { // 求区间[A, B]的最小值
    int x = log_2[B - A + 1];
    return min(rmin[A][x], rmin[B - (1 << x) + 1][x]);
}
int main() {
    scanf("%d%d ", &n, &q);
    for (int i = 1; i <= n; i++)
        scanf("%d", &h[i]);
    for (int i = 2; i <= n; i++) // 预处理每个区间长度对应的 k
        log_2[i] = log_2[i >> 1] + 1;
    init();
    while (q--) {
        int A, B;
        scanf("%d%d", &A, &B);
        printf("%d\n ", query_max(A, B) - query_min(A, B));
    }
    return 0;
}
```

ST 表算法预处理的时间复杂度为 $O(n\log_2 n)$，单次查询的时间复杂度为 $O(1)$，总时间复杂度为 $O(n\log_2 n)$。

ST表不仅可以求区间最大值和最小值，还可以处理符合结合律和幂等律（与自身做运算，结果仍是自身）的信息查询，如区间最大公因数、区间最小公倍数、区间按位或、区间按位与等。

### 例 3-8 喷泉(洛谷P7167,By一只书虫仔)

一个喷泉由 $n$ 个圆盘组成，从上到下依次编号为 $1$ 到 $n$，第 $i$ 个圆盘的直径为 $d_i$，容量为 $c_i$。 当一个圆盘里的水多于这个圆盘的容量，那么水就会溢出往下流，直到流入半径更大的圆盘里。如果下面没有满足要求的圆盘，水就会流到喷泉下面的水池里。

现在有 $q$ 组询问，每一组询问描述如下：向第 $r_i$ 个圆盘里倒入 $v_i$ 的水，求水最后会流到哪一个圆盘停止。如果最终流入了水池里，那么输出 0。注意，每个询问互不影响，也就是每次向圆盘倒水前，所有的圆盘中都没有水。

**数据范围：** $2 \le n \le 10^5$，$1 \le q \le 2 \times 10^5$，$1 \le c_i \le 1000$，$1 \le d_i, v_i \le 10^9$，$1 \le r_i \le n$。

**分析：** 不妨将水池看作直径和容量都为无穷大的圆盘，编号为 $n+1$ 根据题目意思，第 $i$ 个圆盘中的水溢出后首先会流到编号大于 $i$ 的第一个直径大于 $d_i$ 的圆盘，将该圆盘记为 $next_i$。 由于圆盘的情况是固定的，与询问无关，所以数组next也就固定了，可以在询问前预先处理。

预处理 $next_i$ 的过程即求解第 $i+1$，$i+2$，$\cdots$，$n$，$n+1$ 个圆盘中第一个直径大于 $d_i$ 的圆盘的下标，解决方法有很多，这里采用二分加RMQ的做法。

处理完数组next，就可以根据题目意思模拟：对于第 $i$ 组询问，从 $r_i$ 开始，如果水溢出，则考虑流入的下一个圆盘，直到水流尽为止。这样时间复杂度为 $O(nq)$，无法通过本题。

所有的共 $n+1$ 个圆盘间的关系构成了一个树状结构，如果每次都一步一步向上跳会产生大量重复运算。另外，由于有水量的限制，又不能一次性直接跳到根，所以既不能跳太慢也不能跳太快，要恰到好处。在 例 3-7（平衡的队列）中，长度为 $n$ 的区间可以用不超过 $O(\log_2 n)$ 个长度为 $2^k (k \in [0, \lfloor \log_2 n \rfloor])$ 的小区间不重不漏地拼出来。

而本题需要跳到恰好的结点，这之间是相通的。用 $f[i,t]$ 表示从第 $i$ 个圆盘开始，水溢出 $2^t$ 次后所到达的圆盘编号，同时用 $g[i,t]$ 记录水溢出所到达的共 $2^t$ 个圆盘的容量之和，转移 $f[i,t] = f[f[i,t-1],t-1], g[i,t] = g[i,t-1] + g[f[i,t-1],t-1]$。

要找到那个恰好的圆盘，只需要从大到小枚举 $t$，记当前所在的圆盘编号为 $x$，若水可以溢出到达第 $f[x,t]$ 个圆盘并将继续溢出，则溢出到该圆盘，将 $x$ 变更为 $f[x,t]$，重复 $\log(10^5) \approx 17$ 次过程。最后再判断是否可以再溢出一次，即可得到答案，这样完成一次询问的时间复杂度是 $O(\log_2 n)$。

具体代码如下：

```
#include <bits/stdc++.h>
using namespace std;
#define MAXN 100010
int n, q, d[MAXN], c[MAXN], rmax[MAXN][20], f[MAXN][20], g[MAXN][20], log_2[MAXN];
int query_max(int A, int B) {
    int x = log_2[B - A + 1];
    return max(rmax[A][x], rmax[B - (1 << x) + 1][x]);
}
int main() {
    scanf("%d%d ", &n, &q);
    for (int i = 1; i <= n; i++)
        scanf("%d%d", &d[i], &c[i]);
    for (int i = 2; i <= n; i++)
        log_2[i] = log_2[i >> 1] + 1;
    for (int i = 1; i <= n; i++)
        rmax[i][0] = d[i];
```

```
    for (int j = 1; (1 << j) <= n; j++)
        for (int i = 1; i <= n - (1 << j) + 1; i++)
            rmax[i][j] = max(rmax[i][j - 1], rmax[i + (1 << j - 1)][j - 1]);
    c[n + 1] = 1000000000;
    for (int i = 1; i < n; i++) {
        int l = i + 1, r = n + 1, mid;
        while (l < r) { // 通过二分求出水溢出后会流到哪个盘子
            mid = l + r >> 1;
            if (query_max(i + 1, mid) <= d[i])
                // 如果编号小于或等于mid的盘子直径均小于i
                l = mid + 1;
            else
                r = mid;
        }
        f[i][0] = l; // f[i][0]即为next[i]
        g[i][0] = c[f[i][0]];
    }
    f[n][0] = n + 1;
    g[n][0] = c[f[n][0]];
    for (int t = 1; t <= 16; t++)
        for (int i = 1; i <= n; i++) {
            f[i][t] = f[f[i][t - 1]][t - 1];
            g[i][t] = g[i][t - 1] + g[f[i][t - 1]][t - 1];
        }
    while (q--) {
        int r, v;
        scanf("%d%d", &r, &v);
        if (v > c[r]) {
            v -= c[r];
            for (int t = 16; t >= 0; t--)
                if (v > g[r][t]) { // 如果水可以溢出流到f[r][t]号圆盘且还将继续溢出
                    v -= g[r][t]; // 则溢出流到这里
                    r = f[r][t];
                }
            r = f[r][0]; // 由于水还可以继续溢出一次,所以答案是下一个流入的圆盘
        }
        if (r == n + 1)
            r = 0;
        printf("%d\n", r);
    }
    return 0;
}
```

## 3.4 课后习题与实验

### 习题 1 集合求和(洛谷P2415)

给定一个集合 $S$（集合元素数量 $\leqslant 30$，元素大小 $\bullet \leqslant 1000$） ，求出此集合所有子集元素之和。

### 习题 2 平面上的最接近点对(洛谷P1257)

给定平面上 $n$ 个点，找出其中的一对点的距离，使得在这 $n$ 个点的所有点对中，该距离为最小的。

**数据范围：** $2 \leqslant n \leqslant 10000$。

### 习题 3 地毯填补问题(洛谷P1228)

相传有一座大小为 $2^k \times 2^k$ 的正方形宫殿，公主站在其中一个方格子上。只要谁能用地毯将除公主站立的地方外的所有地方都覆盖上，就可以得到国王的赏识。公主这一个方格不能用地毯盖住，毯子的形状有所规定，只能使用如图 3-7 所示的 $4$ 种地毯，且每一方格只能用一层地毯。请给出一种铺地毯的方案。

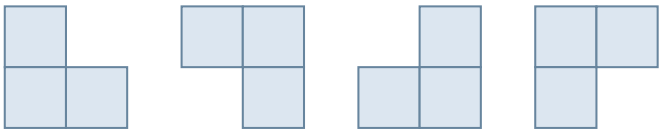


图3-7 毯子的形状

**数据范围：** $0 < k \leqslant 10$。

### 习题 4 奶牛集会(洛谷P2345,USACO04OPEN)

小 $A$ 的 $n$ 头奶牛每年都会参加“呼呼大会”。它们参加活动时会聚在一起，第 $i$ 头奶牛的坐标为 $X_i$，任何两头奶牛的坐标都是不相同的。奶牛们的叫声很大，第 $i$ 头和第 $j$ 头奶牛交流，会发出 $\max(V_i, V_j \times \mid X_i - X_j \mid)$ 的音量，其中 $V_i$ 和 $V_j$ 分别是第 $i$ 头和第 $j$ 头奶牛的听力。

假设每对奶牛之间同时都在说话，请计算所有奶牛产生的音量之和。

**数据范围：** $1 \leqslant N \leqslant 20000, 1 \leqslant V_i \leqslant 20000, 1 \leqslant X_i \leqslant 20000$。

### 习题 5 青蛙(洛谷P3509,POI2010)

数轴上有 $n$ 个点，有一个青蛙在这些点上跳。规则是每次向距离当前点第 $k$ 近的点跳，如果有相同距离则向下标较小的点跳。求从每个点出发跳了 $m$ 次后青蛙的位置。

**数据范围：** $1 \leq k < n \leq 10^6$，$1 \leq m \leq 10^{18}$，$1 \leq p_1 < p_2 < \cdots < p_n \leq 10^{18}$。

### 习题 6 分段(洛谷P3517,POI2011)

给定 $n$ 个点，要求把 $n$ 个点分成不多于 $m$ 段，使得求出每段的最小覆盖圆的半径后，最大的半径最小。

**数据范围：** $1 \leqslant m \leqslant n \leqslant 10^5, -10^6 \leqslant x_i, y_i \leqslant 10^6$，其中 $(x_i, y_i)$ 表示点的坐标。

### 习题 7 覆盖计划(洛谷P4155,四川省队选拔2015)

有一个 $M$ 个点的环，和 $N$ 条线段，第 $i$ 条线段可覆盖环上 $[C_i, D_i]$ 的部分。请问当第 $i$ 条线段被强制选择时，最少选择多少条线段可以覆盖整个环？

**数据范围：** $N \leqslant 2 \times 10^s, M < 10^{9,1} \leqslant C_i, D_i \leqslant M$。

### 习题 8 忠诚数(洛谷 P1816)

有一个 $m$ 个数的序列和 $n$ 个询问，每次询问第 $l_i$ 个数到第 $r_i$ 个数中的最小值。

数据范围 $: n, m \leqslant 10^5$。

### 习题 9 三角形(洛谷P6648,CCC2019)

一个大小为 $m$ 的三角形由 $m$ 行组成，第 $i$ 行包含 $i$ 个元素，并且这些行必须排为等边三角形的形状。每个三角形还包含子三角形（注意，每个三角形都是自身的子三角形）。现在给定一个大小为 $n$ 的三角形，求对于每个大小为 $k$ 的子三角形，子三角形内几个数的最大值的和。

**数据范围：** $1 \leqslant k \leqslant n \leqslant 3000, 0 \leqslant$ 角形内每个数 $\leqslant 10^9$

### 习题 10 活动(洛谷P7562,JOISC 2021)

有 $n$ 场活动，编号为 $1$ 到 $n$。活动 $i$ 从时间 $L_i + 10^{-1}$ 到时间 $R_i - 10^{-1}$ 举行，其中 $L_i$ 和 $R_i$ 均为整数。小 $A$ 要参加其中的 $k$ 场活动，但是他不能在活动中间加入或离开，在两个活动场所间移动的时间忽略不计。

他希望参加编号较小的活动。严格来说，他参加的 $k$ 场活动编号依次为 $a_1, a_2, \cdots, a_k$，其中 $1 \leqslant a_1 < a_2 < \cdots < a_k \leqslant n$ 称序列 $(a_1, a_2, \cdots, a_k)$ 的编号小于 $(b_1, b_2, \cdots, b_k)$，当且仅当存在 $j (1 \leqslant j \leqslant k)$ 满足在区间 $[1, j-1]$ 里所有 $l$ 都有 $a_l = b_l$ 且 $a_j < b_j$

现在给出每个活动的信息和小 $A$ 要参加的活动场数 $k$，判断他是否可以实现参加 $k$ 场活动，如果可以，输出所有的 $k$ 场活动的编号。

**数据范围：** $1 \leqslant n \leqslant 10^5, 1 \leqslant k \leqslant n, 1 \leqslant L_i \leqslant R_i \leqslant 10^9$，且 $1 \leqslant i \leqslant n$。

# 第4章 进阶搜索

搜索算法可以枚举所有可能的结果，找到最优结果或者统计符合要求的结果数量。在《基础篇》中介绍了搜索算法，包括深度优先搜索（Depth First Search，DFS）和广度优先搜索（BreadthFirstSearch，BFS）。在搜索的过程中，每个过程都有若干决策，所以搜索是一类时空开销极大的枚举算法，其时间复杂度往往是指数或者是阶乘级别的。

为了使运行速度快一点，本章将介绍一些更优秀的搜索策略，也许可以帮助我们通过更多的测试点。然而，就算是用上了这些策略，这些搜索算法的时间复杂度往往还是指数级别的。当实在没能想出更好的解法时，可以考虑使用搜索算法来通过一些数据规模较小的测试点。由于搜索程序的运行效率难以估计，现在的算法竞赛中搜索题目的出现频率逐渐降低。此外，随着测试数据强度的增强，本章所介绍的解法可能难以应对部分加强后的测试数据。本章思维导图如图 4-1 所示。

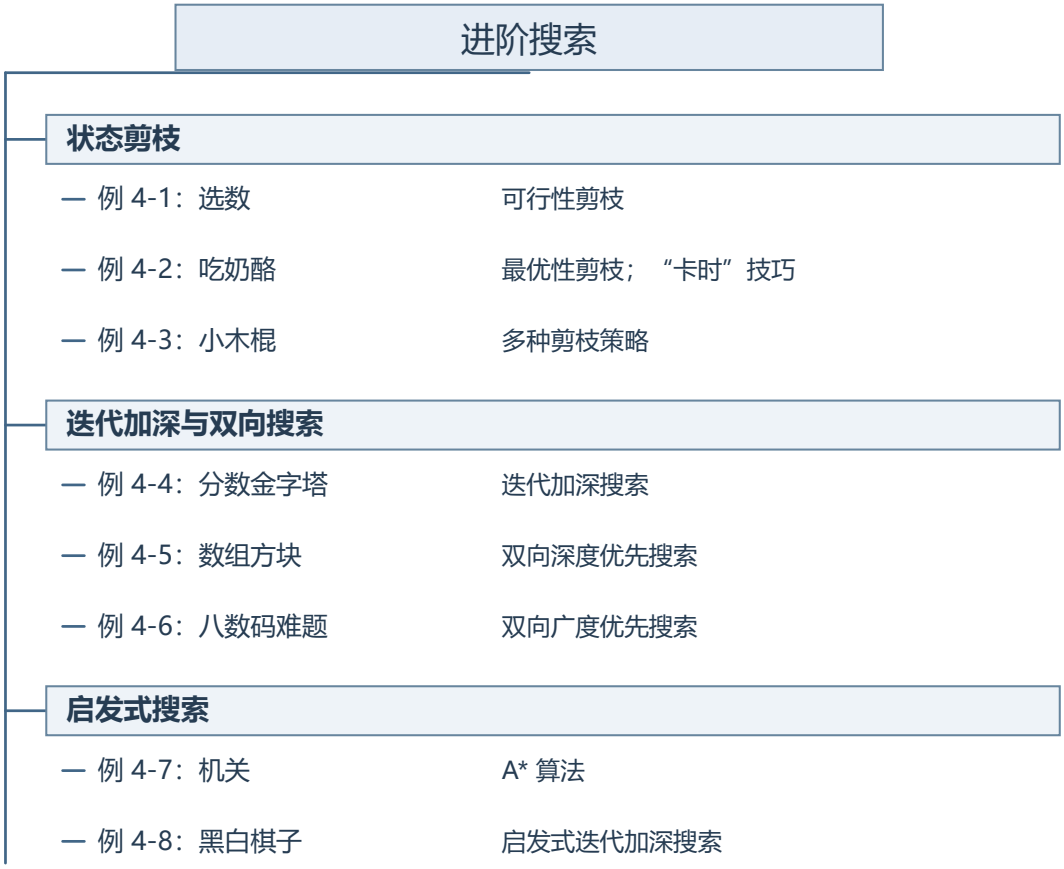



图4-1 本章思维导图

## 4.1 状态剪枝

如果将搜索的状态看作结点，状态之间的转移看作边，搜索的过程就构建出了一棵“搜索树”。所谓“剪枝”，就是在搜索树上去掉一些枝权不进行搜索，从而减小时间消耗。搜索树的剪枝如图 4-2 所示。

系统地说，剪枝的策略分为**可行性剪枝**和**最优性剪枝**两种。需要指出的是，搜索题目的性

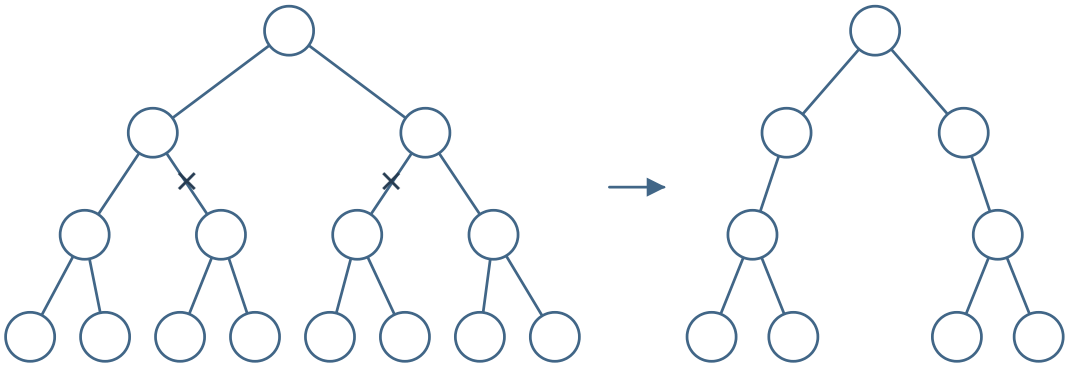


图4-2 状态剪枝

质极其灵活，在应用时一定要具体问题具体分析，设计合适的剪枝策略，不必拘泥于所学。

### 例 4-1 选数(洛谷P1036,NOIP2002普及组)

已知 $n$ 个整数 $a_1,a_2,\cdots,a_n$ 以及 $1$ 个整数 $k(k<n\leqslant 20)$。从这 $n$ 个整数中任选 $k$ 个整数相加，可分别得到一系列的和。现在，要求计算出和为质数共有多少种。

**分析：**设 `dfs(int x, int y, int z)` 表示已经考虑前 $x-1$ 个数，正在决定是否选择第 $x$ 个数，目前选了 $y$ 个数，其和为 $z$。不选时转移到 `dfs(x + 1, y, z)`；选取时转移到 `dfs(x + 1, y + 1, z + a[x])`。

**可行性剪枝**就是“保证当前搜索的状态是合法的”，并且“从当前状态继续搜索时，至少能找到一个最终解”。

注意到只能选择 $k$ 个数，因此当 $\cdot y>k$ 时，当前状态已经不合法了（因为选择了多于 $k$ 个数），此时可以直接结束当前层的递归返回上一状态，因为继续搜索下去绝不可能达到任何一个“选择了 $k$ 个数”的合法状态。

同时，因为必须选择 $k$ 个数，如果将剩下没被考虑的数 $(x\sim n)$ 全部选上，再加上前 $x-1$ 个数中被选择的，还是凑不够 $k$ 个数时，当前状态无论向后怎么选都无法选够 $k$ 个数，那么这种情况也可以直接返回上一层递归，即 $n-x+1+y<k$ 时直接返回。

上述两种剪枝即为可行性剪枝，对应的代码如下：

```
void dfs(int x, int y, int z) {
    if (y > k)
        return; // 可行性剪枝1
    if (n - x + 1 + y < k)
        return; // 可行性剪枝2
    if (x > n) {
        if (is_prime(z)) // 如果总和是质数,则增加答案
            ans++;
        return;
    }
    dfs(x + 1, y, z);
    dfs(x + 1, y + 1, z + a[x]);
}
```

不难发现，在上述代码中，搜索树上最下面一层只有 $C_n^k$ 个结点，而去掉被剪枝的状态，搜索树上每个结点都对应了至少一个最下层的结点，也即每层的合法解点数都不超过 $C_n^k$，又因为搜索树只有 $n$ 层，因此总的合法解点数不超过 $n \times C_n^k$。每一个合法结点最多拓展出两个需要被剪枝的不合法结点，不合法结点不会有后代，因此搜索树的总结点数为 $O(nC_n^k)$。

经过剪枝，将一个原本为 $O(n2^n)$ 的做法优化为了一个与合法结果数相关的复杂度。同时，不是每一层都与最下面一层有相同的结点数，因此实际的结点个数远低于之前分析的 $n \times C_n^k$。

### 例 4-2 吃奶酪(洛谷P1433)

房间里放着 $n(n \leqslant 15)$ 块奶酪，并且知道坐标。一只小老鼠要把它们都吃掉，问至少要跑多少距离？老鼠一开始在（$0$，$0$）点处。

**分析：**考虑一个最优化问题，在搜索过程中已经得出了上述一个解 $x$（但不确定它是否是最优解），在某个状态中，发现从当前状态在最优情况下到达终点所得到的解也不优于已得到的解 $x$，则可以直接剪枝返回。这种优化思路被称为**最优性剪枝**，与 例 4-1（选数）中**“可行性剪枝”**的第 2 种策略类似。

考虑用深度优先搜索解决这个问题：设 `dfs(deep,now,sum)` 表示目前正在决定走第deep步，当前正在第now个点，已经走过的距离是 `sum`。假设已经得到一个吃掉所有奶酪的跑步距离是 `ans`，如果在继续搜索过程中，当到下一块奶酪的距离超过了 `ans`，那么显然之后无论怎么走，都不可能搜索到一个更优的解，因此直接剪枝，不去搜索这个点即可。代码如下：

```
#include <cstdio>
#include <cmath>
#include <cstdlib>
using namespace std;
int n, cnt = 0;
double x[16], y[16], dist[16][16];
double ans;
bool vis[16];
void dfs(int deep, int now, double sum) {
    cnt++;
    if (cnt >= 10000000) { // 卡时,快要超时的情况下输出答案并退出
        printf("%.2f ", ans);
        exit(0); // 退出整个程序,需要用到cstdlib 头文件
    }
    if (deep > n) {
        if (sum < ans)
            ans = sum;
        return;
    }
    for (int i = 1; i <= n; i++)
        if (!vis[i]) {
            double t = sum + dist[now][i];
            if (t >= ans)
                continue; // 最优性剪枝
            vis[i] = 1;
            dfs(deep + 1, i, t);
            vis[i] = 0;
        }
}
int main() {
    scanf("%d", &n);
    ans = 1e6;
```

```
    x[0] = 0;
    y[0] = 0;
    for (int i = 1; i <= n; i++)
        scanf("%lf%lf", &x[i], &y[i]);
    for (int i = 0; i <= n; i++) // 预先处理出各点之间的距离
        for (int j = 0; j <= n; j++)
            dist[i][j] =
                sqrt((x[i] - x[j]) * (x[i] - x[j]) + (y[i] - y[j]) * (y[i] - y[j]));
    dfs(1, 0, 0);
    printf("%.2f", ans);
    return 0;
}
```

本代码中还使用了被称为“卡时”的技巧：记录已经搜索的结点数量 `cnt`，当其值超过一定数量时则停止搜索，直接将目前已经掌握的最佳答案输出。使用“卡时”技巧带有一些“赌”的成分，首先需要赌已经搜索到的最优答案就是最佳答案，其次需要赌搜索次数上限一上限太低可能会漏过最佳答案，而上限过高就会导致程序超时。可以在本地运行程序时构造可能造成超时的极限数据，借助每次IDE运行时间调整确定合适的上限。当然需要考虑到评测机的配置，做一个比较保守的估计。

由于使用“卡时”技巧并不一定搜完全部的状态，运气不好就可能得到错误的答案，但效率高且实现起来较为容易。类似的技巧还有爬山法、模拟退火算法和遗传算法等，读者可以自行查阅相关的资料。本题可以使用**状态压缩动态规划**，其时间复杂度是明确且合理的。

最优性剪枝还有以下几种策略：

1. 使用“贪心”的方式，将可能性大的决策优先枚举。例如在走迷宫时，优先往终点方向前进。就本题而言，对于每次决策，可以先去枚举距离这个点最近的点（当然需要预先处理每个点到其他点的距离排行）。这样可能可以稍微加快到达最优解的速度，但对于本题来说没有太多改善。
2. 目前状态中，如果可以比较方便地判断出后面任何决策都无法使当前状态相比于已经得到的最优解更优，则可以立刻放弃继续搜索下去。例如在走迷宫时，当前已经走的路径距离加上当前点到终点的直线距离已经超过了最优解，则可以不用继续搜索下去。

### 例 4-3 小木棍(洛谷P1120)

给定若干个木棍的长度，它们是由一些一样长的木棍砍断得到的，要求求出原始木棍的最小可能长度。每段的长度 $a_i$ 不超过 $50$，小木棍总数 $n$ 不超过 $65$。

**分析**：简单的想法是，枚举原始木棍的长度 $d$，通过深度优先搜索尝试将所有的木棍拼起来，看能不能拼出若干根长度为 $d$ 的长木棍。具体来说，设 `dfs(u,k)` 表示还剩 $k$ 根长木棍需要拼起来（包括当前的），当前长木棍离拼好还差 $u$ 的长度。边界条件是 $k=0$ 时能拼出长木棍，初始时状态为 $\mathrm{dfs}\left(0,\dfrac{\sum_{i=1}^{n}a_i}{d}\right)$。

转移时，枚举下一根木棍选择哪一根拼在当前长木棍上。可以写出如下代码：

```
void dfs(int u, int k) {
    if (u == 0) {
        dfs(d, k - 1);
        return;
    }
    if (k == 0) {
        printf("%d\n", ans);
        exit(0);
    }
    for (int i = 1; i <= n; ++i)
        if ((used[i] == false) && (a[i] <= u)) {
            used[i] = true;
            dfs(u - a[i], k);
            used[i] = false;
        }
}
```

但是这段代码无法通过本题，考虑进行剪枝：

1. 设 $s = \sum_{i=1}^{n} a_i$，显然 $d$ 应该是 $s$ 的因数，因此对于 $d$ 不能整除 $s$ 的情况，可以直接不搜索。
2. $d$ 最小也需要是所有木棍中最长的长度，因此对 $d$ 的枚举可以从 $a_i$ 的最大值开始。
3. 考虑对于一个被枚举的 $a_i$. 若它不能满足要求，则对于所有的 $j$ 使 $a_j = a_i$ 都不能满足题意，因此当一个长度不符合要求时，无须搜索同长度的其他木棍。
4. 直观上，考虑短木棍比长木棍更加灵活，也就是更能胜任剩余长度较小的情况。因此在枚举新拼接的木棍时，优先考虑较长的木棍，把较短的木棍留给剩余长度更小的情况。
5. 如果当前长棍剩余长度等于当前考虑的木棍长度（即 $a_i = u$） ，但搜索后发现当前木棍拼接并不合适，则说明以前的选择不对，无须考虑其他木棍直接回溯即可（这是因为，用一些短棍拼出当前长度和用一根较长的棍子拼出当前长度完全一致，而短棍更灵活，留到后面再拼更优，因此如果较长的木棍直接拼接上去不符合要求，则短棍无论如何也不可能符合要求）。
6. 如果当前木棍长度就是长木棍的长度，但拼接后不符合要求，也可以直接返回。

根据上述剪枝，重新组织代码，用一个数组 `len` 保存当前长度为 $i$ 的木棍还剩多少根，预处理一个数组pre表示长度小于 $i$ 的最长木棍的长度，在 `dfs` 函数中加人参数 $p$，表示当前长棍用的最短棍长度，则枚举当前小木棍时，只需要从 $p$ 到 $\min\{a_i\}$ 逐次枚举长度即可。代码如下：

```
#include <cstdio>
#include <cstdlib>
#include <algorithm>
const int maxn = 66;
int n, sum, mn = maxn, mx, d;
int len[maxn], a[maxn], pre[maxn];
void dfs(int u, int k, int p) {
    // 当前长棍还有u没有拼,还有k根长棍没有拼,当前长棍的最短短棍长度是p
```

```
    if (u == 0) {
        dfs(d, k - 1, a[n]);
        return;
    }
    if (k == 0) {
        printf("%d\n", d);
        exit(0); // 找到解
    }
    p = (p < u) ? p : u;
    while (p && len[p] == 0)
        --p;
    while (p)
        if (len[p]) { // 枚举当前棍长
            --len[p];
            dfs(u - p, k, p);
            ++len[p]; // 回溯
            if ((u == p) || (u == d))
                return; // 最后一条剪枝
            p = pre[p];
        } else
            p = pre[p];
}
int main() {
    scanf("%d ", &n);
    for (int i = 1, x; i <= n; ++i) {
        scanf("%d", &x);
        sum += x;
        ++len[a[i] = x];
    }
    std::sort(a + 1, a + 1 + n);
    for (int i = 1; i <= n; ++i) // 预处理长度为 i 的上一个长度
        if (a[i] != a[i - 1])
            pre[a[i]] = a[i - 1];
    for (d = a[n]; (d << 1) <= sum; ++d)
        if ((sum % d) == 0)
            dfs(d, sum / d, a[n]);
    printf("%d\n", sum);
    return 0;
}
```

## 4.2 迭代加深与双向搜索

搜索算法的问题在于搜索深度每增加一层，状态数就会呈指数增长。一般来说，由于大部分问题的搜索时间效率会与搜索层数有关，如果将搜索状态分为两部分分别进行搜索，然后合并结果，使用这一方法就可以降低复杂度，提升效率。

### 例 4-4 分数金字塔(洛谷P1763)

给出一个分数，由分子 $a$ 和分母 $b$ 组成，现在需要分解成一系列互不相同的单位分数（形如 $\frac{1}{x}$，即分子为 $1$）。要求：分解成的单位分数数量越少越好，如果数量一样，最小的那个单位分数越大越好。例如：$\frac{19}{45}=\frac{1}{3}+\frac{1}{12}+\frac{1}{180}$ 和 $\frac{19}{45}=\frac{1}{5}+\frac{1}{6}+\frac{1}{18}$ 两种分解方法都需要 $3$ 个单位分数，但第 $2$ 种方法更优，因为 $\frac{1}{18}>\frac{1}{180}$

**分析：**考虑进行DFS或BFS。如果选择DFS，令搜索阶段为现在枚举到第几个数，则不清楚接下来还会选择多少个数，数的个数可能是无穷大，即一个分数可能分解成无穷多个单位分数，搜索树的深度可能是无穷大。如果选择BFS，令搜索状态为现在已经搜索出的数，则可能的状态太多，既难以进行队列储存，也无法进行**记忆化搜索**，因为进行分解时第一个单位分数的取值有无穷多个，分母可能是无穷大，搜索树的宽度可能也是无穷大。

考虑对解进行一定的限制：从小到大枚举分解出的单位分数的个数（也就是枚举深度），在搜索过程中按照分解出的单位分数从大到小（也就是上一层的分母比下一层的小）的顺序进行搜索，步骤如图 4-3 所示。

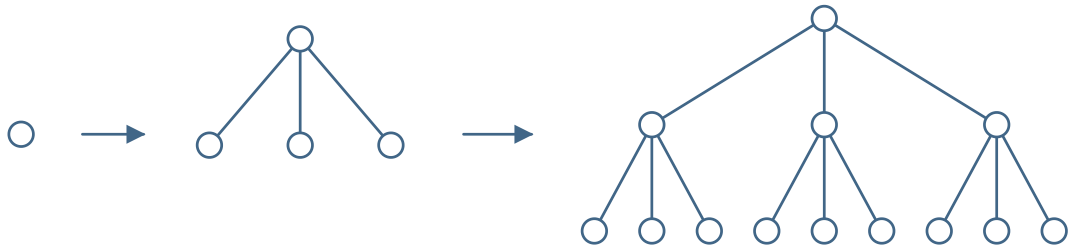


图4-3 迭代加深搜索

不难发现，限制深度后，整个搜索树的大小变成了有限的：深度上，不能超过限制的深度 $h$ 每一层的宽度上，考虑因为是从大到小枚举的，也即若当前结点为 $\frac{1}{i}$，则其后代都小于 $\frac{1}{i}$，于是若当前深度为 $w$，则它加上它后代的和小于 $\frac{\frac{h-w+1}{h}}{i}$， 。

参考**最优性剪枝**的策略，如果当前答案加上所有后代的分数也不能凑出 $\frac{a}{b}$，则可以直接剪枝。同时，每一层的宽度显然至多是 $\frac{a}{b} \times h$ （因为更小的分数无法在 $h$ 层内凑出 $\frac{a}{b}$）

这种通过枚举深度来进行搜索的方式被称为**迭代加深**搜索，或称为迭代加深深度优先搜索（Iterative Deepening Depth-first Search IDS或 IDDFS）。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long
ll dep = 1, st[11], ans[11], flag;
ll gcd(ll x, ll y) {
    if (y == 0)
        return x;
    else
        return gcd(y, x % y);
}
void dfs(ll a, ll b, int x) {
    if (x > dep) // 层数超过了
        return;
    if (a == 1 && b > st[x - 1]) { // 剩下了1/b, 找到了答案
        st[x] = b;
        if (!flag || st[x] < ans[x]) // 复制答案
```

```
            for (int i = 1; i <= dep; i++)
                ans[i] = st[i];
        flag = 1;
        return;
    }
    ll l = max(b / a, st[x - 1] + 1); // 从上一个,或者b/a 开始枚举
    ll r = (dep - x + 1) * b / a;     // 枚举上限最优性剪枝
    if (flag && r >= ans[dep])        // 如果有过解也可以确定上限
        r = ans[dep] - 1;
    for (ll i = l; i < r; i++) {
        st[x] = i;
        ll gc = gcd(a * i - b, b * i);
        dfs((a * i - b) / gc, b * i / gc, x + 1);
    }
}
int main() {
    ll a, b;
    cin >> a >> b;
    ll c = gcd(a, b);
    a /= c;
    b /= c;
    st[0] = 1;
    for (dep = 1; dep <= 10; dep++) { // 迭代加深
        dfs(a, b, 1);
        if (flag) { // 找到一组解
            for (int i = 1; i <= dep; i++)
                printf("%lld", ans[i]);
            break;
        }
    }
    return 0;
}
```

代码中声明了一个全局变量 `dep`，用来规定搜索深度的上限。将其设为全局变量是为了方便跨函数调用，设成函数参数也是可以的。

然后是使用 `dfs` 函数来加深搜索，其中参数 $x$ 表示搜索深度。函数中，如果当前的搜索深度大于规定的搜索深度 `dep`，则直接返回，表示在当前规定的搜索深度下，此状态一定不能到达目标状态。

在主函数中，从小到大枚举深度界限 `dep`，每次让深度界限增加 1，并调用 `dfs` 函数，在新的深度界限下再次搜索。当搜索到目标后不再继续增加深度界限，此时 $d$ 的值就是能搜到答案的最小深度。

考虑时间效率问题：在深度界限 $d$ 不足以找到解的时候，枚举是否会消耗大量的时间？假设搜索树是一个满 $n$ 叉树，则其第 $h$ 层的结点个数为 $n^{h-1}$，而前 $h-1$ 层的结点个数为 $1+n+n^2+\cdots+n^{h-1}=\frac{n^{h-2}-1}{n-1}<n^{h-1}$。

也就是说，前面几层的结点个数加起来都没有最下面一层的结点个数多，因此时间主要消耗于最后一次递归的最下面一层。随着深度的增加，需要搜索的状态数呈指数级增长。前面搜索的状态加起来可能都没有最后一次多，因此不用担心时间浪费。

### 例 4-5 数组方块(洛谷CF525E)

给出一个长度为 $n$ 的数组 $a$，可以从中选出任意个数组成数组 $b$，并在 $b$ 中再选出不超过 $k$ 个数，将这些数变成自己的阶乘。最后求数组 $b$ 的各个元素之和为 $S$ 的方案数。两个方案不同当且仅当从 $a$ 选择时的下标不同或从 $b$ 中选择时的下标不同。

**数据范围：** $1 \leqslant n \leqslant 25, 0 \leqslant k \leqslant n, 1 \leqslant S \leqslant 10^{16}, 1 \leqslant a_i \leqslant 10^9$.

**分析：** 朴素的做法是直接DFS枚举所有可能的情况：对于第 $i$ 个数，有这个数不被选、被选但不变成阶乘、被选且变成阶乘 3 种决策。这样做的时间复杂度为 $O(3^n)$，本题不能通过。

考虑将数列分成两部分，分别是前 $\lfloor \frac{n}{2} \rfloor$ 个数一组和后 $n-\lfloor \frac{n}{2} \rfloor$ 个数一组。分别对于这两部分进行DFS，这种搜索方式被称为双向深度优先搜索。在前半组的搜索结束时，用Hash记录该状态的方案数，然后对后半部分进行DFS，设搜索结束时状态是选了 $i$ 个阶乘，和为 $s$，记为（$i$，$s$），在后半部分搜索结束时，将答案加上 $\sum_{t=0}^{k-i} f[(t, S-s)]$ 即可，其中 $f$ 是前半组搜索对应状态的方案数。

其步骤如图 4-4 所示。

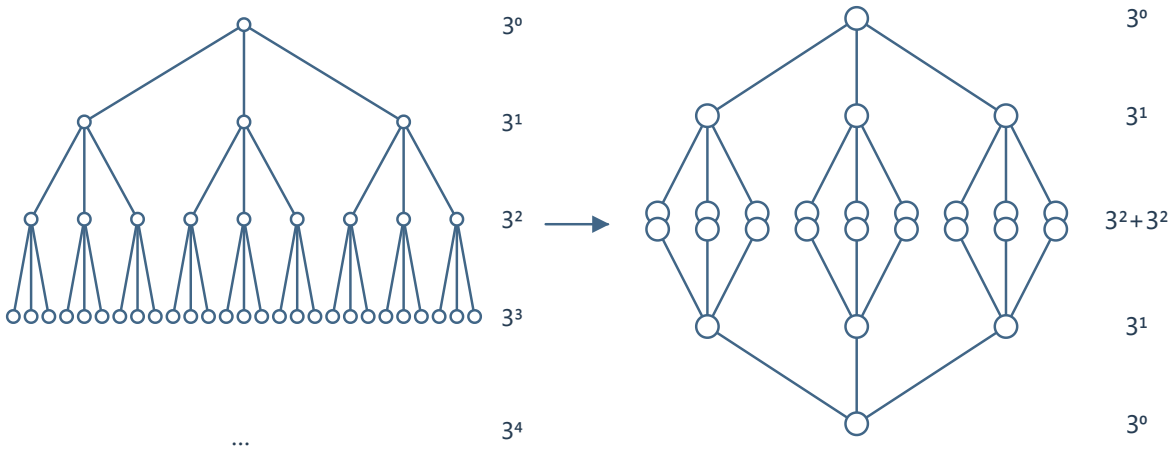



图 4-4 双向深度优先搜索

不难发现，这两部分进行搜索的时间复杂度都是 $O(3^{\frac{n}{2}})$。合并两部分答案的复杂度为 $O(k)$，因此总复杂度为 $O(3^{\frac{n}{2}}k)$，可以通过本题。

```
#include <bits/stdc++.h>
using namespace std;
typedef long long int ll;
const int maxn = 100;
int n, t, m;
ll S, ans, a[maxn];
int b[maxn];
map<pair<int, ll>, int> f;
ll fact(ll x) {
    ll ret;
    for (ret = 1; x; ret *= x--)
        if ((S / ret) < x)
            return -1;
    return ret;
}
```

```
void check1() {
    int cnt = 0;
    ll sum = 0;
    for (int i = 1; i <= m; ++i)
        if (b[i] == 1)
            sum += a[i];
        else if (b[i] == 2) {
            ll s = fact(a[i]);
            if ((s == -1) || (++cnt > t) || ((s + sum) > S))
                return;
            // 这是加上s后当前和大于S和选择的阶乘数大于k的情况
            sum += s;
        }
    ++f[{cnt, sum}]; // 累加方案数
    ++f[{-1, sum}];  // 标记 sum 这一和出现过
}
void dfs1(int u) {
    if (u > m)
        check1(); // 到达搜索终点
    else {
        for (int i = 0; i <= 2; ++i) { // 枚举每个位置的三进制数
            b[u] = i;
            dfs1(u + 1);
        }
    }
}
map<pair<int, ll>, ll> g;
ll qry(int x, ll s) { // 表示统计 i从0到x, f[{i,s}]的和,使用记忆化提高
    if (g.count({x, s}))
        return g[{x, s}];
    return g[{x, s}] = f[{x, s}] + (x ? qry(x - 1, s) : 0);
}
void check2() {
    int cnt = 0;
    ll sum = 0;
    for (int i = m + 1; i <= n; ++i)
        if (b[i] == 1)
            sum += a[i];
        else if (b[i] == 2) {
            ll s = fact(a[i]);
            if ((s == -1) || (++cnt > t) || ((s + sum) > S))
                return;
            sum += s;
        }
    if (f.count({-1, S - sum}))
        ans += qry(t - cnt, S - sum);
    // 如果S-sum这一和出现过,就递归累加答案
}
void dfs2(int u) {
    if (u > n)
        check2();
    else {
        for (int i = 0; i <= 2; ++i) {
            b[u] = i;
            dfs2(u + 1);
        }
    }
```

```
}
int main() {
    scanf("%d %d %lld", &n, &t, &S);
    for (int i = 1; i <= n; ++i)
        scanf("%lld", a + i);
    m = n >> 1;
    dfs1(1);     // 前半部分搜索
    dfs2(m + 1); // 后半部分搜索
    printf("%lld\n ", ans);
    return 0;
}
```

这里使用了pair容器，相当于有两个元素的结构体，但写起来简洁一些；类似于用{`cnt`，`sum`直接表示一个pair的写法是 C++11 以上标准所支持的。读者可以自行查找相关的资料。

### 例 4-6 八数码难题(洛谷P1379)

在 $3\times3$ 的棋盘上，摆有 $8$ 个棋子，每个棋子上标有 $1$ 至 $8$ 中的某一数字。棋盘中留有一个空格，空格用 $0$ 来表示。空格周围的棋子可以移到空格中。求解的问题是：给出一种初始布局（初始状态）和目标布局（为了使题目简单，设目标状态为 $123804765$），找到一种最少步骤的移动方法，实现从初始布局到目标布局的转变。

**分析：**不难发现，本题总的状态数只有 9！个，而状态之间的转移可以使用BFS来轻松完成。但是考虑本题一种状态能够转移出 4 种后继状态，这导致搜索树非常庞大，虽然能够完成搜索，但有没有效率更高的做法呢？

考虑从初始状态和目标状态一起开始BFS。当某一个状态被既从初始状态又从目标状态拓展到时，两个状态到该状态的步数和即为答案。从两个状态为起点进行BFS的方法即为双向BFS。不难发现，两个起点的搜索方向是相对而行的，即一个正着搜一个反着搜，因此被称作双向广度优先搜索。

可以用三叉搜索树近似分析时间效率。设答案为 $k$，单向 BFS 搜索至第 $k$ 层时，结点数约为 $1+3+3^2+\cdots+3^k=(3^{k+1}-1)/2$。双向 BFS 中，两侧各搜索约 $k/2$ 层，每棵树的结点数约为 $(3^{k/2+1}-1)/2$，两棵树合计约为 $3^{k/2+1}-1$。这里忽略深度取整和重复状态，只用于说明搜索规模由 $O(3^k)$ 降到约 $O(3^{k/2})$。

双向BFS的状态数只有单向BFS的平方根级别，可以显著地减小所经过的状态。

本题中每个格子上都有一个数字，这个状态可以轻松地映射为一个 9 位数，即这个数的第 1 位表示第 1 行第 1 列的格子，第 2 位表示第 1 行第 2 列的格子，第 4 位表示第 2 行第 1 列的格子，类推即可。

双向BFS有两种写法，既可以把两个方向的状态放在同一个队列里，同时标记它们是正向搜索到的状态还是反向搜索到的状态；也可以开两个队列分别进行两个方向的搜索，外层使用for循环枚举当前经过的步数。一旦队头的步数超过循环所限制的步数，则切换至另一方向进行枚举。以下代码中使用的是第 1 种方法。

```
#include <bits/stdc++.h>
using namespace std;
int bg, ed = 123804765;
int a[10];
map<int, int> rec[2];     // 1表示从最终状态开始的搜索
queue<pair<int, bool>> Q; // 0表示从初始状态开始的搜索
int change(int x, int op) {
    for (int i = 9; i; --i) // 解开状态
        a[i] = x % 10, x /= 10;
    int p;
    for (int i = 1; i <= 9; ++i) // 找到0 所在的格子
        if (a[i] == 0) {
            p = i;
            break;
        }
    if (op == 1) {
        if (p <= 3)
            return -1; // 不能将 0向上交换
        swap(a[p], a[p - 3]);
    } else if (op == 2) {
        if (p >= 7)
            return -1; // 不能将 0 向下交换
        swap(a[p], a[p + 3]);
    } else if (op == 3) {
        if ((p % 3) == 1)
            return -1; // 不能将0向左交换
        swap(a[p], a[p - 1]);
    } else if (op == 4) {
        if ((p % 3) == 0)
            return -1; // 不能将 0向右交换
        swap(a[p], a[p + 1]);
    }
    x = 0;
    for (int i = 1; i <= 9; ++i)
        x = x * 10 + a[i]; // 计算状态代表数字
    return x;
}
void check(int x, pair<int, bool> h) {
    if (x != -1 && !rec[h.second].count(x)) {
        if (rec[h.second ^ 1].count(x) == true) {
            printf("%d\n", rec[h.second ^ 1][x] + rec[h.second][h.first] + 1);
            exit(0);
        } else {
            rec[h.second][x] = rec[h.second][h.first] + 1;
            Q.push(make_pair(x, h.second));
        }
    }
}
int main() {
    cin >> bg;
    if (bg == ed) {
        puts("0");
        return 0;
    }
    rec[0][bg] = 0;
    rec[1][ed] = 0;
```

```
    Q.push(make_pair(bg, 0));
    Q.push(make_pair(ed, 1));
    while (!Q.empty()) {
        pair<int, bool> h = Q.front();
        Q.pop();
        int x = change(h.first, 1); // 将为0 一格向上变换的转移
        check(x, h);
        x = change(h.first, 2); // 下
        check(x, h);
        x = change(h.first, 3); // 左
        check(x, h);
        x = change(h.first, 4); // 右
        check(x, h);
    }
}
```

用std::set或Hash的方式来分别存储两个方向已经到达过的状态，就可以在较小的时间消耗下完成对某个状态是否已到达的查询。在 C++98 标准中，可以使用make`_pair(x,y)` 来表示一个pair。本程序比较依赖各种STL容器，如果不使用O2 优化，则部分测试点还是有超时的风险。

需要说明的是，在本题中最坏情况仍需要枚举所有的状态，因此并没有降低算法的时间复杂度，只是在一般情况下减少了一些无用枚举，可以更快速地找到搜索终点。

## 4.3 启发式搜索

启发式搜索是指在搜索的过程中，利用问题拥有的信息来引导搜索，达到减小搜索范围、降低问题复杂度的目的。“使用贪心的方式，将可能性大的决策优先枚举”策略就是使用了启发式搜索的思想。

### 例 4-7 机关(洛谷P5507,ByHope2075)

一扇锁着的门上有一个机关，上面一共有 **12** 个旋钮，每个旋钮有 **4** 个状态，用数字 **1** 到 **4** 表示。每个旋钮只能向一个方向旋转（状态： **12341**） ，在旋转时，会引起另外某个旋钮也旋转一次（方向相同，不会引起连锁反应）。当所有旋钮都旋转到状态 **1** 时，机关就打开了。请问最少需要操作几次才可以打开机关呢？

输入 **12** 行，每行 **5** 个整数，描述机关的状态。每一行中先给出按钮的初始状态，然后给出 **4** 个整数，表示这个按钮的状态分别为 **1**、**2**、**3**、**4** 的时候旋转，会引发一起旋转的按钮编号。测试数据保证在 **17** 步内可以得到解。

**分析：**本题有多种解法可以解决，如双向广搜。现使用 $A$ 算法，这种算法属于启发式搜索的一类。例如，我国有很多城市，城市之间由高速公路连接。开车从上海到北京，中途到达了某个城市后，要决定去哪个邻近城市?查看可以选择的城市，哪个离北京直线距离更近，就优先选择这个城市。

对于一个状态 $x$，设 $g(x)$ 是从起点到达 $x$ 的代价，也就是从上海到达这个城市走过的路程；$h^*(x)$ 是从 $x$ 到达终点的代价的估计值，也就是这个城市到北京的直线距离。注意这个估值一定要比实际情况乐观，例如这里的直线距离，也就是估计值，一定不会大于最终的实际距离。又设
$f^*(x)=g(x)+h^*(x)$ 为状态的估值函数。

在搜索时，找到 $f$ 值最小的状态，优先对其进行搜索，配合**最优性剪枝**，可以有效地剪去较差分支，提高算法效率。

对于本题来说 ，$g(x)$ 就是当前状态已经行走的步数， $h^*(x)$ 可以自行设计，例如当前状态在不考虑牵连的情况下，转到目标状态的步数除以 2，因为转一次按钮有可能让另一个按钮也会向目标状态转一次，总之需要得到一个“乐观”但又尽可能接近真实步数的值。假设预先已知道状态 $x$ 到终点的代价准确值 $h(x)$，则 $f(x)=g(x)+h(x)$ 代表的就是答案的总代价。

程序本身就是类似于BFS的做法，不过不是使用队列，而是使用**优先队列**维护各种状态，$f^*(x)$ 小的状态优先。从初始状态开始，扩展到下一个状态时发现目标状态，则可以直接输出结果；计算对应的 $f^*(x)$ 值，将其放入优先队列中。接着从优先队列中找到 $f^*(x)$ 最小的状态来进行扩展。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1 << 24;
int g[N], nxt[12][4], fa[N], ans[20], choice[N];
struct node {
    int state;               // 状态
    double F;                // 状态对应估价函数值
    node(int s) : state(s) { // 构造函数,冒号后面部分相当于 state = s;
        double h = 0;
        F = 0;
        for (int i = 0; i < 12; i++)
            if ((s >> (i * 2)) & 3) // 计算不处在状态1的旋钮的对应的h值
                h += 4 - ((s >> (i * 2)) & 3);
        F = h + g[s]; // 计算估值函数
    }
    bool operator<(const node &y) const {
        return F > y.F; // 估价函数值小的优先
    }
};
priority_queue<node> q;
int main() {
    int button, Start = 0;
    for (int i = 0; i < 12; i++) {
        scanf("%d ", &button);            // 读入第i+1个旋钮状态
        Start |= (button - 1) << (i * 2); // 记录初始状态
        for (int j = 0; j < 4; j++) {
            scanf("%d ", &nxt[i][j]);
            nxt[i][j] -= 1;
        }
    }
    q.push(node(Start)); // 调用构造函数,顺便计算出估价函数值
    g[Start] = 0;
    while (!q.empty()) {
        int state = q.top().state;
        q.pop();
        if (state == 0)
            break;
        int si, sNxt, nx, nextState;
        for (int i = 0; i < 12; i++) {
            si = (state >> (i * 2)) & 3;    // 第i个按钮的状态
            nx = nxt[i][si];                // 受牵连的按钮
            sNxt = (state >> (nx * 2)) & 3; // 受牵连的按钮的状态
            nextState = state ^ (si << (i * 2)) ^ (((si + 1) & 3) << (i * 2));
            nextState = nextState ^ (sNxt << (nx * 2)) ^ (((sNxt + 1) & 3) << (nx * 2));
            // 如果没有访问过就可以转移新状态了
            if (!g[nextState]) {
                g[nextState] = g[state] + 1;
                fa[nextState] = state;     // 记录操作路径
```

```
                choice[nextState] = i + 1; // 记录操作按钮
                q.push(node(nextState));
            }
        }
    }
    int cnt = 0, state = 0;
    while (state != Start) {
        ans[++cnt] = choice[state];
        state = fa[state];
    }
    printf("%d\n", cnt);
    for (int i = cnt; i; i--)
        printf("%d ", ans[i]);
    return 0;
}
```

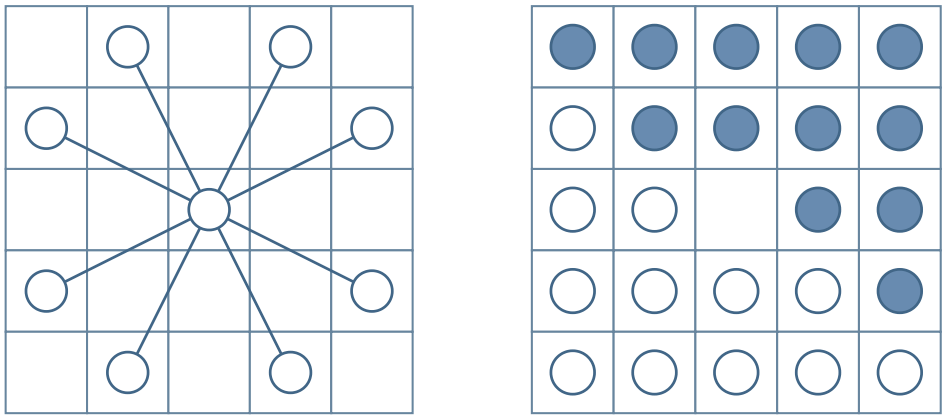
因为状态是由 12 个 1 到 4 的数字组成，可以用 24 位二进制数表示，压缩成一个int类型变量表示一个状态。假设这个按钮当前的状态是si，下一个状态就是（si+1）&3；将整个状态码对应的按钮位数通过异或消去（变为零），在对应位数补上下一个状态码，通过位运算可以高效构造下一个状态的数码。

实际上 $h^*(x)$ 函数可以自行设计。当 $h^*(x)=0$ 时，算法就退化成了普通的BFS，当 $h^*(x)$ 越大，算法效率越高，但如果 $h^*(x)$ 的值超过了实际到达终点的代价 `h(x)`，就可能得到错误的答案。

本题中，如果读者认为这个估计过于保守，可以尝试在计算 $h^*(x)$ 时乘上一个大于 1 的数字，如 1.5，算法也可以通过本题，而且耗时减少，只是需要承担少量的答案错误的风险。当乘以 2.5 的时候，就会得到错误的答案，无法通过测试点了。

### 例 4-8 黑白棋子(洛谷P2324,四川省队选拔2005)

在 $5\times5$ 的棋盘上给定 **12** 个黑色棋子和 **12** 个白色棋子。每个棋子可以走“日”字，如图 4-5(a) 所示，即目标格子的横坐标和原来的差 **2**，纵坐标差 **1**；或者横坐标差 **1**，纵坐标差 **2**。每次可以移动黑棋或者白棋任意一个，要求用最少的步数到达目标状态，如图 4-5(b) 所示。问至少需要多少步呢？如果 **15** 步以内无法完成，则输出 $-1$。



(a) (b)

图4-5 黑白棋子

**分析：**本题的转移很简单。对于一个状态，看看哪个棋子可以跳到空格里面。由于可能的状态数较多，而且不能很好地全部保存下来，所以如果直接使用BFS比较困难（使用双向BFS还是可以的）。例 4-7 中介绍了 $A*$算法，对于本题，还可以将 $\mathbf{A^*}$ 算法的思想和**迭代加深**搜索结合在一起，变为启发式迭代加深搜索，又称为 $\mathbf{IDA^*}$ 算法。

在本题中，使用迭代加深搜索所枚举的搜索上界 `dep` 即为总的代价 `f(x)` 的最大值；而从起点到状态 $x$ 的代价 $g(x)$，即走的步数恰好对应着搜索层数。以上便是通过限制代价的方式限制搜索的深度，而对搜索宽度的限制则是通过“估价函数”来进行的，即只有 $f^*(x)=g(x)+h^*(x)$ $\leqslant$ `dep` 时，该状态才会被本轮搜索拓展。

从本状态到终点代价的估计值 `h(x)` 可以设计为尚未归位的棋子数量，因为每个尚未归位的棋子一定得让开，走至少一步，这是一个相当保守的估计，但足以通过此题。代码如下：

```
#include <bits/stdc++.h>
#define ll long long
using namespace std;
int m[7][7], flag, dep;
const int dx[8] = {1, 1, -1, -1, 2, 2, -2, -2};
const int dy[8] = {2, -2, 2, -2, 1, -1, 1, -1};
const char *goa = "111110111100211000010000"; // 目标状态
int h() {                                     // 到目标状态的估值
    int cnt = 0;
    for (int i = 0; i < 5; i++)
        for (int j = 0; j < 5; j++)
            cnt += m[i][j] != goa[i * 5 + j] - '0';
    return cnt;
}
void dfs(int g, int x, int y) {
    if (g == dep) {
        if (!h())
            flag = 1;
        return;
    }
    for (int i = 0; i < 8; i++) {
        int xx = x + dx[i], yy = y + dy[i];
        if (xx < 0 || xx >= 5 || yy < 0 || yy >= 5)
            continue;
        swap(m[x][y], m[xx][yy]); // 走棋
        if (g + h() <= dep)       // 启发式搜索关键步骤
            dfs(g + 1, xx, yy);
        swap(m[x][y], m[xx][yy]); // 悔棋
    }
}
int main() {
    int T, sx, sy;
    cin >> T;
    while (T--) {
        flag = 0;
        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 5; j++) {
                char ch;
                cin >> ch;
                if (ch == '*') { // 记录起点
```

```
                    m[i][j] = 2;
                    sx = i;
                    sy = j;
                } else
                    m[i][j] = ch - '0';
            }
        }
        for (dep = 0; dep <= 15; dep++) { // 迭代加深
            dfs(0, sx, sy);
            if (flag) {
                cout << dep << endl;
                break;
            }
        }
        if (!flag)
            cout << -1 << endl;
    }
    return 0;
}
```

虽然这些搜索算法一般用于数据范围比较小的题目，但是读者还是应该尝试实现代码。一方面。可以尝试各种方式的剪枝训练挖掘各种性质的能力；另一方面，代码实现起来有些难度，通过实现这些代码可以提升编程能力。

## 4.4 课后习题与实验

### 习题 1 请证明：在 A* 算法中，若启发函数 $h^*(x)$ 不大于到目标的真实代价 $h(x)$，按估价函数扩展状态时，首次取出并扩展目标状态得到的是最优解

### 习题 2 请分别尝试使用A*算法和IDA算法解决例4-6中的八数码难题;使用双向BFS完成例4-7的机关问题

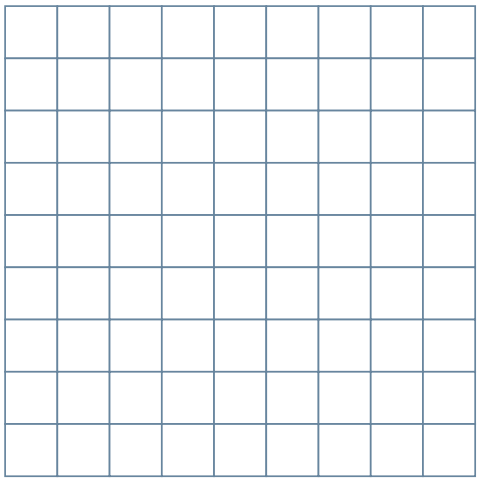
### 习题 3 靶形数独(P1074,NOIP2009提高组)

一个 $9\times9$ 的表格中，每个格子都是 $0$ 到 $9$ 之间的一个数字。初始时，一些格子已经被填入了数字，请继续在空的格子中填写数字，满足表格中每个横行、纵行和如图 4-6(a) 所示的小九宫格刚好 $1$ 到 $9$ 各出现一次。此外，每个格子都有对应的分值如图 4-6(b) 所示。填完所有数字的表格最后得分就是每个格子填的数字乘以这个格子的分值的总和。请问可以得到的最高分数是多少？原表格中非 $0$ 数的个数不少于 $24$。

### 习题 4 Mayan游戏(洛谷P1312,NOIP2011提高组)

某游戏界面是一个 $7\times5$ 的棋盘，上面堆放着一些方块。方块不能悬空堆放，即必须放在最下面一行，或者放在其他方块之上。游戏通关是指在规定的步数内消除所有的方块，规则如下：

1. 每步移动可以且仅可以沿横向（即向左或向右）拖动某一方块一格：当拖动这一方块时，如果拖动后到达的位置（以下称目标位置）也有方块，那么这两个方块将交换位置；如果目标位置上没有方块，那么被拖动的方块将从原来的竖列中抽出，并从目标位置上掉落。



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 6 |
| 6 | 7 | 8 | 8 | 8 | 8 | 8 | 7 | 6 |
| 6 | 7 | 8 | 9 | 9 | 9 | 8 | 7 | 6 |
| 6 | 7 | 8 | 9 | 10 | 9 | 8 | 7 | 6 |
| 6 | 7 | 8 | 9 | 9 | 9 | 8 | 7 | 6 |
| 6 | 7 | 8 | 8 | 8 | 8 | 8 | 7 | 6 |
| 6 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |

(a)

2. 任一时刻，如果在一横行或者竖列上有连续 $3$ 个或者 $3$ 个以上相同颜色的方块，则它们将立即被消除。
3. 如果同时有多组方块满足消除条件，几组方块会同时被消除。
4. 出现行和列都满足消除条件且行列共享某个方块时，行和列上满足消除条件的所有方块会被同时消除。
5. 方块消除之后，消除位置之上的方块将掉落，掉落后可能会引起新的方块消除。注意：掉落的过程中将不会有方块的消除。

如果可以在 $n(n\leqslant 5)$ 步内完成消除，请给出解决方案，否则输出 $-1$。

### 习题 5 宝藏(洛谷P3959,NOIP2017提高组)

给出一个有 $n$ 个宝藏屋的图，有 $m$ 条未开凿道路连接这些宝藏屋且长度已知。小明可以任选一个起点开始挖宝。他可以在已经开凿的道路上行走，自由访问起点和已经开凿道路连接的宝藏屋；也可以选择开凿新的道路，但需要保证已经开凿的道路和连接的宝藏屋是一棵树。小明想让所有的宝藏屋均可自由访问。

开凿某条道路时，要求道路其中一头的藏宝屋是可以自由访问的。开凿新的一条道路需要花费代价 $L\times K$，其中 $L$ 代表这条道路的长度，$K$ 代表待打通的宝藏屋到起点所经过的道路数量。请问工程总代价最小值是多少。

**数据范围：** $1\leqslant n\leqslant 12, 0\leqslant m\leqslant 10^3, v\leqslant 5\times 10^5$。

**提示：**这是一道**状态压缩动态规划**的题目，但是读者可以考虑使用搜索和剪枝的策略通过本题。

### 习题 6 杂草的入侵(洛谷P2960,USACO2009年10月赛)

草地是长 $Y(1\leqslant Y\leqslant 100)$，宽度为 $X(1\leqslant X\leqslant 100)$ 的格子。（$1$，$1$）是左下角的格子坐标。杂草一开始（第 $0$ 周）占领了格子 $(M_x, M_y)$。每星期杂草都会蔓延到上下左右非岩石的空地上。给出地图，请问第几周杂草可以完全占领入侵整片空地呢？

### 习题 7 质数礼物(洛谷CF912E)

给定一个大小为 $n$ 的质数集合 $P(n\leqslant 16$，集合中的数不超过 $100$），求第 $k$ 大的所有质因子都在 $P$ 中的数，保证答案不超过 $1018$。

### 习题 8 四子连棋(洛谷P2346)

在一个 $4\times 4$ 的棋盘上摆放了 $14$ 颗棋子，其中有 $7$ 颗白色棋子、$7$ 颗黑色棋子，有两个空白地带。任何一颗黑白棋子都可以向上、下、左、右 $4$ 个方向移动到相邻的空格，称为行棋一步。黑白双方交替走棋，任意一方可以先走，如果某个时刻使得任意一种颜色的棋子形成四个一线（包括斜线），这样的状态为目标棋局。求用最少的步数移动到目标棋局的步数。

### 习题 9 冰球比赛(洛谷 P4799,CEOI2015)

小 $B$ 想看冰球比赛，买若干张门票。给出他的总预算 $M$ 和 $N$ 场比赛的票价。请问如果总票价不超过预算，他有多少种观赛方案。如果存在以其中一种方案观看某场比赛而另一种方案不观看，则认为这两种方案不同。

**数据范围：** $1\leqslant N\leqslant 40, 1\leqslant M\leqslant 10^{18}$。

### 习题 10 学院迷宫(洛谷P2483,山东省队选拔2010)

给出 $N$ 个结点 $M$ 条边的有向图，边有权值 $e_i$，可能存在重边。从 $1$ 号结点到 $N$ 号结点走过的路线权值总和不超过给定的 $E$。除了 $N$ 号结点，其他结点均可重复经过。请问有几种走法？

数据范围 $:2\leqslant N\leqslant 5000, 1\leqslant M\leqslant 200000, 1\leqslant E\leqslant 10^7, 1\leqslant e_i\leqslant E, E$ 和所有的 $e_i$ 为实数。

**提示：**使用 A * 算法。

### 习题 11 文化之旅①(洛谷P1078,NOIP2012普及组)

有一位使者要游历各国，一共有 $N(2\leqslant N\leqslant 100)$ 个国家， $K(2\leqslant K\leqslant 100)$ 种文化，$M(1\leqslant M\leqslant N^2)$ 条道路，长度不超过 $1000$。他每到一个国家都会学当地文化，但如果他学习了某种文化，则他就不能到达其他有这种文化的国家。不同的国家可能有相同的文化，有些文化会排斥外来文化（即如果他学习了某种文化，则他不能到达排斥这种文化的其他国家）。

现给定各个国家间的地理关系、各个国家的文化、每种文化对其他文化的看法 $(K\times K$ 的 $01$ 矩阵表示是否排斥）、这位使者游历的起点和终点（在起点和终点也会学习当地的文化）以及国家间的道路距离，试求从起点到终点最少需走多少路。

> **注：** 1. 本题尚未找到没有多项式时间复杂度解法，不能保证所有符合数据范围的测试点都有办法在时限内通过，出题人在 $5$ 年后才意识到这一点。读者可以尝试使用 $A$ 算法完成本题。

# 第2部分：进阶数据结构

## 第5章 二叉堆与树状数组

本章将介绍**二叉堆**和**树状数组**两种树形结构。

大学生活是忙碌而丰富多彩的，有大量的任务需要完成，如作业、实验、论文、兼职工作等。这些任务都是必须要做的，而且每个任务都有对应的截止日期。对于大部分学生来说，在每次决定要完成哪些任务时，一定会选择最接近截止日期的任务，因为这个任务最为紧迫，优先级最高。

这种可以查询最大或最小值、可以插入新的元素、可以删除最大或最小值的集合容器，被称为**优先队列**。通过使用二叉堆实现优先队列，可以较高效地完成插入和查询操作。

树状数组可以很方便地维护序列的**前缀和**等信息。因此，可以使用树状数组高效完成单点修改、区间求和的任务。虽然之后介绍的**线段树**也可以完成这些任务，而且时间复杂度都是 $O(\log n)$，但是树状数组运行时间的常数更小，所需空间更小，代码也更加短小精悍。本章思维导图如图 5-1 所示。




图5-1 本章思维导图

## 5.1 二叉堆的实现

### 例 5-1 堆(洛谷P3378,本题和原题相比有所不同)

给定一个数列，初始为空，请支持下面 $3$ 种操作：

1. 给定一个整数 $x$，请将 $x$ 加入数列中。
2. 输出数列中最大的数。（原题是求最小的数）
3. 删除数列中最大的数（如果有多个数最大，只删除 $1$ 个）。

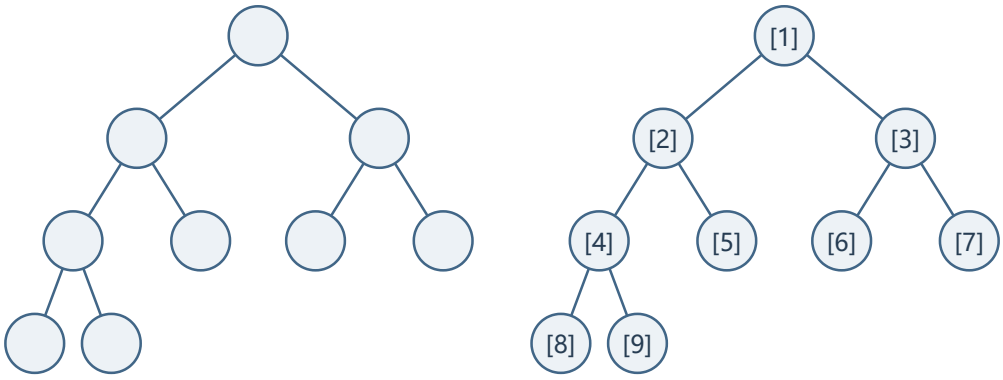
一共要操作 $n(1 \le n \le 10^6)$ 次，插入的数字 $x$ 满足 $1 \leqslant x < 2^{31}$。

**分析：**有两种暴力算法的方式可以解决这个问题。第 1 种方式是准备一个数组，插入时把待插入的数放入到数组的末尾，查找最值时通过打擂台法枚举所有数组中的元素，而删除时直接将其标记为无效。这样插入和删除元素（在已经知道位置的情况下）的时间复杂度都是 $O(1)$，但查找的时间复杂度是 $O(n)$。

第 2 种方式还是准备一个数组，但需要维护这个数组，使其中的元素满足单调性（如从小到大排列）。插入和删除时，利用插入排序的思想，调整各个数组元素的相对位置；而查询最值直接访问数组最开头的元素即可。这样查找的时间复杂度是 $O(1)$，但插入和删除元素的时间复杂度是 $O(n)$。

以上两种方式，在插入与查询次数都很多的情况下效率很低，所以需要利用**二叉堆**提升效率。

堆是建立在完全二叉树基础之上的一种数据结构。首先需要了解使用完全二叉树存储数据的方式。对于一棵完全二叉树①，按顺序编号：根结点是 1 号；第 2 层从左到右是 2，3；第 3 层从左到右是 4，5，6，7，…依此类推，则可以得到如图 5-2 所示的树。




(a) 完全二叉树　　(b) 编号结果

图5-2 二叉树的编号

观察每个结点的编号，可以得到两条结论：

1. 编号为 $x$ 的结点的父结点是 $x/2$]。
2. 编号为 $x$ 的结点的左子结点是 2x，右子结点是 2x+1。

在此基础上，寻找一个点的父结点或孩子，只需要对它的下标进行算术运算。这种存储方式由于经常用于存储二叉堆之类的完全二叉树，因此得名“堆式存储”。由此，只需要用一个数组就可以存储一棵二叉堆。

利用堆式存储，只需要开一个数组 `w[]`，以 `w[x]` 存储编号为 $x$ 的结点的数据的值，代码如下，非常简单。

```
int w[MAXN];
int tot; // 目前堆中有多少个结点
```

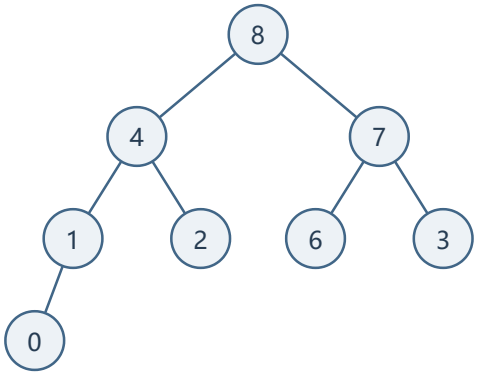
二叉堆是一棵树，每个结点有一个值。它满足以下性质：

> **注：** 1. 如果一个二叉树除了最后一层以外，其他层的结点都是满的，而且最后一层的结点是从左到右的一排连续的，那么这样的二叉树就称为完全二叉树。

1. 二叉堆是一棵完全二叉树。

2. 对于大根堆而言，父结点的值一定大于或等于子结点的值；小根堆反之。

如图 5-3 所示为一个大根堆的案例。下面以这个大根堆为例，介绍如何自己实现一个堆：对于 `top()` 操作，返回集合中的最大值；对于 `push(x)` 操作，将元素 $x$ 加入集合；对于 `pop()` 操作删除掉最大值。




| x | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| w[x] | 8 | 4 | 7 | 1 | 2 | 6 | 3 | 0 |

(a) 大根堆　　(b)二叉堆在数组中的存储方式

图5-3 一个大根堆

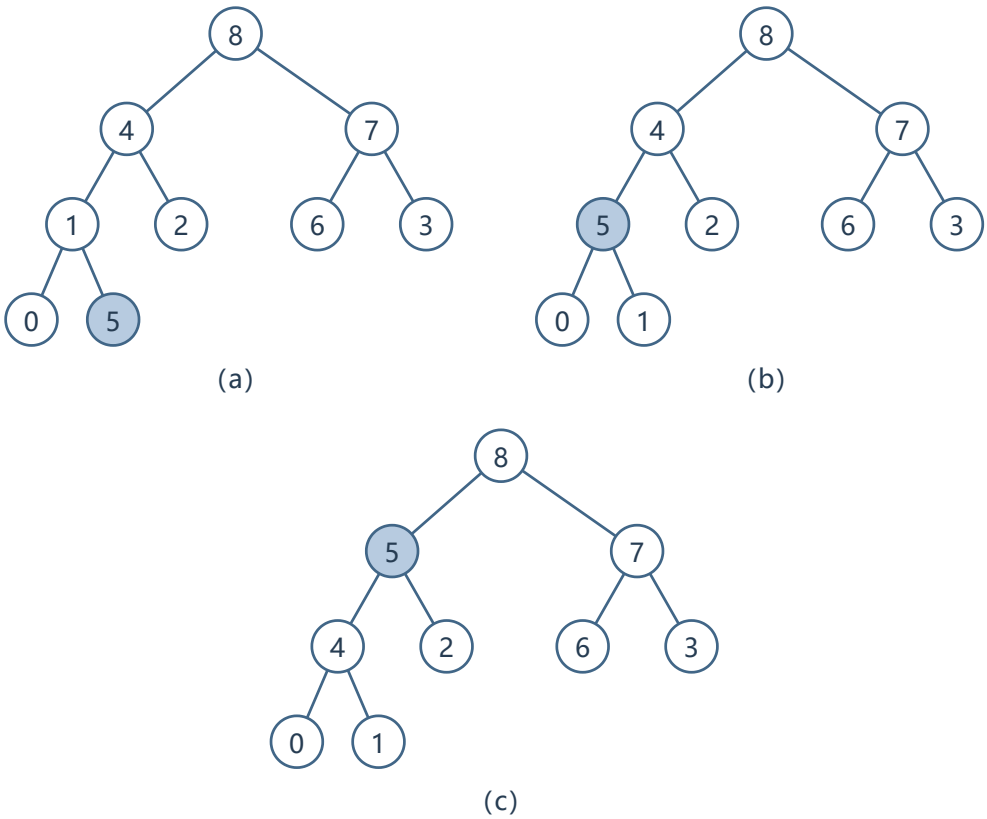
1. `top()` 操作，返回集合中的最值。

当堆符合上面提到的性质时，如果要返回集合的最大值，只需要返回树根的值即可。在 $O(1)$ 的复杂度内可以办到。代码如下：

```
int top() {      // 返回最值
    return w[1]; // 堆顶是w[1]
}
```

2. `push(x)` 操作，将元素 $x$ 加入集合。

无论如何，先把 $x$ 放进树的尾部—这是为了保持“堆是一棵完全二叉树”的性质。假设插入的是 5，这棵树会变为如图 5-4(a) 所示的状态。




(c) 图5-4在堆中插入元素

很遗憾，现在这棵树违反了二叉大根堆的第 2 条性质父结点的值一定大于或等于子结点的值，因此需要修复。“修复”就是二叉堆的主要思想。

刚刚插入的结点一定在最底层，那么对它进行以下操作：

1. 若自己是根，或者自己的值比父结点小，则停止操作。
2. 否则，将自己与父结点的值交换，然后对父结点递归执行同样的修复操作。

根据这一过程，对新加入的结点 5 执行修复，它会将自己与父结点 1 交换，如图 5-4(b) 所示；然而结点 5 仍然是违反堆性质的，所以继续往上递归修复，如图 5-4(c) 所示。至此，堆的性质得到了满足，5 已经成功地插入这个堆。代码如下：

```
void modify(int x) { // 插入时,修复x号结点
    if (x == 1 || w[x] < w[x / 2])
        return; // x为根,或已经符合大根堆的性质
    swap(w[x], w[x / 2]);
    modify(x / 2);
}
void push(int x) { // 加入元素
    w[++tot] = x;  // 将 x直接加入到尾部
    modify(tot);   // 自底向上修复
}
```

3. `pop()` 操作，删除堆顶元素。

由于堆始终是一棵完全二叉树，如直接把堆顶擦掉，它将会分裂成两棵树，此时会遇到大麻烦。一种合适的做法是，先把堆顶和尾部结点的值互换，然后把尾部结点擦掉。在这一过程中，这个堆始终是完全二叉树。如果对上文那棵树执行 `pop()` 操作，堆会变成如图 5-5 所示的状态：

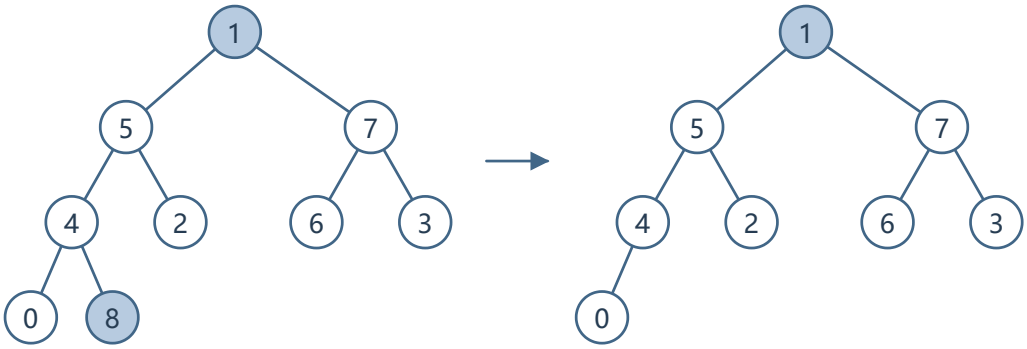



图5-5 删除二叉堆遇到的问题

所谓拆东墙补西墙，“完全二叉树”的性质守住了，但“父结点的值大于或等于子结点的值”的性质便违背了。但是，二叉堆的维护过程正是在于不断地修复。现在的修复任务是：根结点的值较小，但它的两个子结点的值之一却是堆中剩下元素中最大的。面对这个问题，对根结点进行如下修复操作：

1. 如果自己已经是叶子结点，或者自己的值比子结点大，则停止操作。
2. 否则，把自己与子结点中值较大的那个结点交换值，然后递归下去继续修复。

因此，把根结点 1 与它的两个子结点 5、7 中的最大值 7 交换，如图 5-6(a) 所示。但这样仍然违背堆的性质。所以把 1 与其子结点 6、3 之间的最大值 6 交换，如图 5-6(b) 所示。此时，1 已经是叶子结点，整棵树符合堆的性质，删除操作到此结束。代码如下：

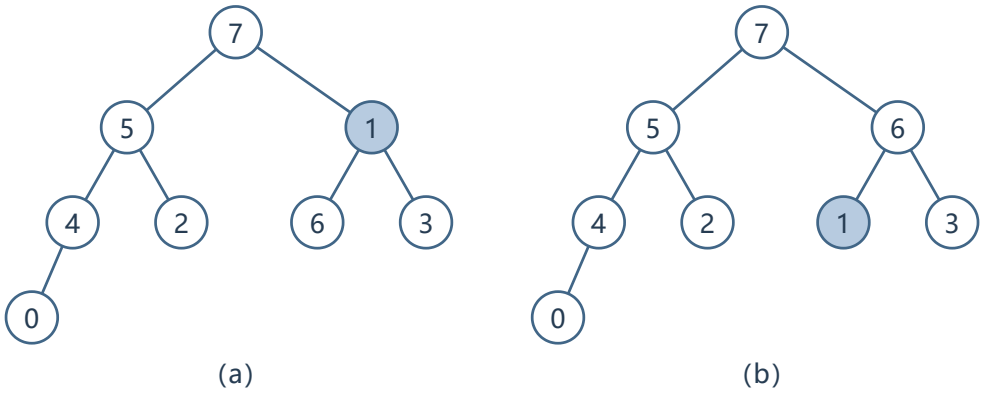



图5-6 删除二叉堆的元素

```
void repair(int x) { // 删除时,修复x号结点
    if (x * 2 > tot)
        return; // x已经是叶子结点
    int tar = x * 2;
    if (x * 2 + 1 <= tot)
        tar = w[x * 2] > w[x * 2 + 1] ? x * 2 : x * 2 + 1;
    // 现在,tar是x子结点中key最大的
    if (w[x] < w[tar]) { // 违反大根堆的性质
        swap(w[x], w[tar]);
        repair(tar);
    }
}
void pop() {
    swap(w[1], w[tot--]); // 与尾部结点交换,然后删掉尾部
    repair(1);            // 自顶向下修复
}
```

有了以上这些工具，本问题就可以得到解决。其他部分程序代码如下：

```
#include <cstdio>
#include <algorithm>
using namespace std;
#define MAXN 1000000 + 5
int w[MAXN];
int tot; // 目前堆中有多少个结点
int top() { return w[1]; }
void modify(int x) {
    while (x > 1 && w[x] < w[x / 2]) {
        swap(w[x], w[x / 2]);
        x /= 2;
    }
}
void push(int x) {
    w[++tot] = x;
    modify(tot);
}
void repair(int x) {
    while (x * 2 <= tot) {
        int y = x * 2;
        if (y + 1 <= tot && w[y + 1] < w[y])
            ++y;
        if (w[x] <= w[y])
            break;
        swap(w[x], w[y]);
        x = y;
    }
}
void pop() {
    w[1] = w[tot--];
    if (tot)
        repair(1);
```

```
}

int main() {
    int n, x;
    scanf("%d ", &n);
    while (n--) {
        int op, x;
        scanf("%d", &op);
        if (op == 1) {
            scanf("%d", &x);
            push(x);
        } else if (op == 2) {
            printf("%d ", top());
        } else {
            pop();
        }
    }
    return 0;
}
```

查询二叉堆顶（也就是根结点）元素的时间复杂度是 $O(1)$ 的，而插入和删除的时间复杂度均为 $O(\log n)$。

### 例 5-2 快速排序(洛谷P1177)

输入 `n(n<100000)` 个数字 $a_1, a_2, \cdots, a_n (a_i < 10^9)$，将其从小到大排序后输出。

**分析**：之前读者应该已经学习过如何利用快速排序或者归并排序完成这个任务了。不难想到，利用堆也是可以做排序的：先把所有的元素都push进去，然后每次取出堆顶（最小值）输出并弹掉堆顶，直到堆空为止。上述排序方法称为堆排序。

为了从小到大排序，实现的是小根堆，在判断堆的性质是否成立时大小关系有所不同。最终代码如下：

```
#include <cstdio>
#include <algorithm>
using namespace std;
int w[100000 + 5];
int tot;         // 目前堆中有多少个结点
int top() {      // 返回最小值
    return w[1]; // 堆顶是w[1]
}
void modify(int x) { // 插入时,修复x号结点
    if (x == 1 || w[x] > w[x / 2])
        return; // x为根,或已经符合小根堆的性质
    swap(w[x], w[x / 2]);
    modify(x / 2);
}
void push(int x) { // 加入元素
    w[++tot] = x;  // 将x直接加入到尾部
    modify(tot);   // 自底向上修复
}
void repair(int x) { // 删除时,修复 x号结点
    if (x * 2 > tot)
        return; // x已经是叶子结点
    int tar = x * 2;
    if (x * 2 + 1 <= tot)
        tar = w[x * 2] < w[x * 2 + 1] ? x * 2 : x * 2 + 1;
```

```
    // 现在,tar是x子结点中key最小的
    if (w[x] > w[tar]) { // 违反小根堆的性质
        swap(w[x], w[tar]);
        repair(tar);
    }
}
void pop() {
    swap(w[1], w[tot--]); // 与尾部结点交换,然后删掉尾部
    repair(1);            // 自顶向下修复
}
int main() {
    int n, x;
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        scanf("%d", &x);
        push(x);
    }
    for (int i = 1; i <= n; i++) {
        printf("%d ", top());
        pop();
    }
    return 0;
}
```

堆排序整体的时间复杂度是 $O(n\log n)$，空间复杂度为 $O(n)$。

## 5.2 优先队列

### 例 5-3 合并果子(洛谷P1090,NOIP2004提高组)

在一个果园里，多多已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。多多决定把所有的果子合成一堆。每一次合并，多多可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。已知果子的种类数 $n(n\leqslant 20000)$ 和每种果子的数目 $a_i(a_i\leqslant 10000)$ 现在的任务是设计出合并的次序方案，使多多耗费的总体力最少，只需要输出这个最小的体力耗费值即可。

例如，合并[$1$，$2$，$3$，$4$，$5$]的过程如下：

1. [$1$，$2$，$3$，$4$，$5$]选取 $1$、$2$，合成 $3$。
2. [$3$，$3$，$4$，$5$]选取 $3$、$3$，合成 $6$。
3. [$4$，$5$，$6$]选取 $4$、$5$，合成 $9$。
4. [$6,9$]选取 $6$、$9$，合成 $15$。
5. [$15$]全部合并完毕。

**分析：**考虑贪心策略，每次取场上最小的两堆合并起来，一直操作到只剩下一堆果子为止，这一过程中耗费的体力值即为答案。在《基础篇》中介绍了使用两个队列维护未合并过的果子的最小值，以及已经合并过的果子的最小值的做法，这样可以高效地寻找哪两堆果子最小。既然如此，为什么不维护一个堆，存放所有果子的重量，每次合并时取出两个最小的，然后合并后再放回去呢?

很幸运，C++ 提供了**优先队列**这种数据结构，也就是STL中的priority_queue，底层由堆实现。要使用优先队列，需要包含queue头文件。优先队列支持的基础操作如下。

1. priority_queue $q$：新建一个保存int 型变量的优先队列 $q$，默认是大根堆。

2. priority_queue <int, vector , greater > $q$：新建一个小根堆。

3. $q$.top()：优先队列查询最大值（或者是最小值）。

4. $q$. pop()：将最大值（最小值）弹出队列。

5. $q$. push(x)：将 $x$ 加入优先队列。

和大多数STL容器一样，仍旧使用 $q$. empty() 判断它是否为空，用 $q$.size() 获取它的大小。

回到本题，只需要先把所有的数都扔进优先队列，然后对于每次合并取出最小值，再弹出；再取出最小值，再弹出；将之前取出的两个值之和加入优先队列，同时统计答案。重复上述操作直到队列中只剩下一个元素。最终代码如下：

```
#include <cstdio>
#include <algorithm>
#include <queue>
using namespace std;
priority_queue<int, vector<int>, greater<int>> q; // 维护最小值
int main() {
    int n, x, y, ans = 0, cost;
    scanf("%d ", &n);
    for (int i = 1; i <= n; i++)
        scanf("%d", &x), q.push(x);
    while (q.size() > 1) {
        x = q.top();
        q.pop();
        y = q.top();
        q.pop();
        cost = x + y;
        q.push(cost);
        ans += cost;
    }
    printf("%d \n ", ans);
    return 0;
}
```

注意，上述代码在定义优先队列时，在最后一个“>”前加入一个空格，刻意规避了…grea-ter>的写法。这是一种良好的代码习惯，因为在比较旧的 C++ 标准中，不加空格会让编译器认为是右移运算。

### 例 5-4 荷马史诗(洛谷P2168,NOI2015)

一部《荷马史诗》中有 $n(n\leqslant 10^6)$ 种不同的单词，从 $1$ 到 $n$ 进行编号。其中第 $i$ 种单词出现的总次数为 $w_i(w_i \leqslant 10^{11})$。现在要用 $k$ 进制串 $s_i$ 来替换第 $i$ 种单词，使得其满足对于任意的 $1\leqslant i,j\in n(i\neq j)$，都有 $s_i$ 不是 $s_j$ 的前缀。

请问如何选择 $s_i$，才能使替换以后得到的新的《荷马史诗》长度最小？在确保总长度最小的情况下，还想知道最长的 $s_i$ 的最短长度是多少？

**分析：**在《基础篇》的“贪心”一章中已经介绍过二进制的哈夫曼编码—像合并果子一样，首先是每个单词都独立成堆，然后每次选择最小的两堆进行合并，建立哈夫曼树。权值越高的结点离根结点距离越小，所以经过编码后的总文章长度最短，具体过程请读者自行证明。一个单词对应哈夫曼树的叶子结点，对应的编码就是从根结点到这个叶子结点的路径，满足编码后的字符串中一个不是另外一个的前缀的性质。

对于本题来说，每次从堆中选出权重最小的 $k$ 个结点，将其合并建边，然后放回堆中，直到建立完哈夫曼树。例如，当各结点权重分别为 1，1，3，3，9，9，需要编码为三进制时，生成的哈夫曼树如图 5-7 所示。

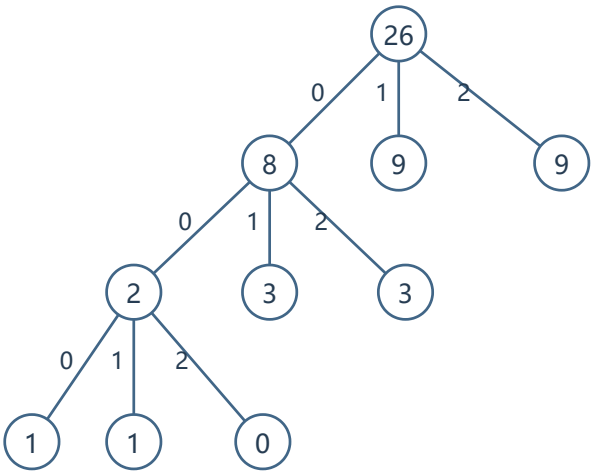



图5-7 哈夫曼树

需要注意的是，每次合并都会减少 $k-1$ 个结点。在合并最后一次的时候，如果可以合并的结点的数量不足 $k$ 个，靠近根结点的位置（短编码）反而没有被利用，所以需要补上 $k-1-(n-1)$ 个权重为 0 的结点，把有权重的结点“推”到离根结点更近的位置。根据数据范围，显然要使用long long类型。代码如下：

```
#include <cstdio>
#include <algorithm>
#include <queue>
using namespace std;
typedef long long LL; // long long 太长,用 LL 代替
struct node {
    LL w, h;
};
bool operator<(node a, node b) { // 关系运算符重载
    if (a.w != b.w)
        return a.w > b.w; // 如果权重相同,则高度小的优先
    return a.h > b.h;
}
priority_queue<node> q;
int main() {
    int n, k;
    LL ans = 0, wi;
    scanf("%d %d ", &n, &k);
    for (int i = 1; i <= n; i++) {
        scanf("%lld", &wi);
        q.push(node{wi, 1}); // 读入结点
    }
    if ((n - 1) % (k - 1) != 0) // 如果最后不够
        for (int i = 1; i <= k - 1 - (n - 1) % (k - 1); i++)
```

```
            q.push(node{0, 1}); // 需要补权重为0的结点的个数
    while (q.size() != 1) {
        node tmp;
        LL sum = 0, maxh = 0;
        for (int i = 1; i <= k; i++) { // 找k次最小的
            tmp = q.top();
            sum += tmp.w;            // 新结点加上子结点权重
            maxh = max(maxh, tmp.h); // 最大深度
            q.pop();
        }
        ans += sum;                  // 统计答案贡献
        q.push(node{sum, maxh + 1}); // 放回去
    }
    printf("%lld\n%lld", ans, q.top().h - 1); // 编码长度是高度减一
    return 0;
}
```

该代码段中将结构体node，也就是哈夫曼树的结点作为**优先队列**中的元素；同时重载了小于号运算符，用于比较两个结点谁的权值更小，如果权值一样，则高度更低的结点优先。

### 例 5-5 蚯蚓(洛谷P2827,NOIP2016提高组)

现在有 $n$ 条蚯蚓（$n$ 为正整数）。第 $i$ 条蚯蚓的长度为 $a_i(i=1,2,\cdots,n)$。每一秒，在所有的蚯蚓中找到最长的那条（如有多条则任选一条）并将其分成两半。分开蚯蚓的位置由常数 $p(0<p<1)$ 决定。设这条析蚓长度为 $x$，将其分成两条长度分别为$\lfloor px \rfloor$和 $x-\lfloor px \rfloor$的蚯蚓（符号 $c$ 表示对 $c$ 向下取整）。

特殊地，如果这两个数的其中一个等于 $0$，则这个长度为 $0$ 的蚯蚓也会被保留。此外，除了刚刚产生的两条新蚯蚓，其余蚯蚓的长度都会增加 $g$（是一个非负整数）。现在已经经过了 $m$ 秒时间，想知道：

1. $m$ 秒内，每秒钟被分的蚯蚓在被分之前的长度（有 $m$ 个数）。
2. $m$ 秒后，从大到小所有蚯蚓的长度（有 $n+m$ 个数）。

对于所有测试数据， $n\leqslant 10^5, m\leqslant 7\times 10^6, a_i\leqslant 10^8$。题目中的 $p$=uv（$u$ 和 v 是测试数据中给出的整数）。其中的子任务的数据范围有梯度，建议参阅洛谷原题。

**分析：**参考合并果子，每次从蚯蚓堆中找到最大的一条，把它分开，再放回去即可。但是每次放回时，其他的蚯蚓全部都要变长 $q$，怎么办呢?除了这两条蚯蚓以外，其他蚯蚓都增加长度 $q$，相当于这两条蚯蚓减少 $q$（当然要记住，计算长度的时候还要全部加上若干个 $q$）。代码如下：

```
#include <cstdio>
#include <queue>
using namespace std;
priority_queue<int> ans; // 储存蚯蚓长度的优先队列
int main() {
    int n, m, q, u, v, t, tmp;
    int alldec = 0; // 所有蚯蚓被减了偏移量
    scanf("%d%d%d%d%d%d ", &n, &m, &q, &u, &v, &t);
```

```
    for (int i = 1; i <= n; ++i) {
        scanf("%d ", &tmp);
        ans.push(tmp);
    }
    for (int i = 1; i <= m; ++i) {
        int top = ans.top() + alldec; // 从蚯蚓堆取最长的,加上偏移量
        ans.pop();
        int q1, q2; // 被分后的两只
        q1 = int((long double)u / v * top);
        q2 = top - q1;
        alldec += q;  // 累加偏移量
        q1 -= alldec; // 减去偏移量
        q2 -= alldec;
        ans.push(q1); // 放回第一只
        ans.push(q2);
        if (i % t == 0)
            printf("%d ", top); // 根据题目要求输出的第1行
    }
    printf("\n ");
    for (int i = 1; ans.size(); ++i) {
        if (i % t == 0)
            printf("%d ", ans.top() + alldec); // 要求输出的第2行
        ans.pop();                              // 弹出
    }
    return 0;
}
```

这种解法进行 $m$ 次操作，单次操作的时间复杂度为 $O(\log n)$，所以总的时间复杂度是 $O(m \log n)$，稍稍大了一点，会有少量测试点无法在规定的时间内运行完毕，必须优化时间复杂度。

可以利用“合并果子”例子中使用两个队列的技巧条。先被分开的两条蚯蚓一定比后面分的对应的两条蚯蚓长。假设两条蚯蚓长度分别为 $a_1$ 和 $a_2(a_1 a_2)$，第 1 条被分成了 $a_{11} = pa_1$ 和 $a_{12} = (1-p)a_1$ 两条。

一段时间后，所有蚯蚓都变长了 $s$，之前被分的两条长度分别变为 $a_{11} = pa_1 + s$ 和 $a_{12} = (1-p)a_1 + s$；而原来的第 2 条被分成了 $a_{21} = p(a_2 + s)$ 和 $a_{22} = (1-p)(a_2 + s)$，可以得到 $a_{11} > a_{21}$ 且 $a_{12} > a_{22}$。

因此，可以准备 3 个队列，第 1 个队列存放原来的蚯蚓，第 2 个队列存每次分蚯蚓后的第 1 段，第 3 个队列存的是第 2 段。这 3 个队列都是单调不增的。比较这 3 个队列的队首，选取最大的一个来进行分割。当然也是需要考虑上一种解法中提到的长度偏移量，代码如下：

```
#include <cstdio>
#include <queue>
#include <algorithm>
#define inf 2100000000
using namespace std;
const int maxn = 1e5 + 20;
int n, m, q, u, v, t;
int a[maxn], x, y;
```

```
queue<int> q1, q2, q3;
bool cmp(int A, int B) { return A > B; }
int findmax() { // 找3 个队列中的最值
    int x, x1, x2, x3;
    if (!q1.empty())
        x1 = q1.front();
    else
        x1 = -inf;
    if (!q2.empty())
        x2 = q2.front();
    else
        x2 = -inf;
    if (!q3.empty())
        x3 = q3.front();
    else
        x3 = -inf;
    if (x1 > x2 && x1 > x3) {
        x = x1;
        q1.pop();
    } else if (x2 > x3) {
        x = x2;
        q2.pop();
    } else {
        x = x3;
        q3.pop();
    }
    return x;
}
int main() {
    scanf("%d%d%d%d%d%d", &n, &m, &q, &u, &v, &t);
    for (int i = 1; i <= n; i++)
        scanf("%d", &a[i]);
    sort(a + 1, a + n + 1, cmp);
    for (int i = 1; i <= n; i++)
        q1.push(a[i]);
    for (int i = 1; i <= m; i++) {
        x = findmax() + q * (i - 1); // 原蚯蚓长度
        if (i % t == 0)
            printf("%d ", x);
        y = (long double)x * u / v; // 第一只
        x = x - y;                  // 第二只
        y -= q * i;
        q2.push(y);
        x -= q * i;
        q3.push(x);
    }
    printf("\n ");
    for (int i = 1; i <= n + m; i++) {
        x = findmax() + q * m;
        if (i % t == 0)
            printf("%d ", x);
    }
    return 0;
}
```

这种方法的时间复杂度为 $O(n\log n+m)$，因为最开始蚯蚓的长度需要排序。

## 5.3 树状数组

**树状数组**是一种用于维护 $n$ 个数**前缀和**信息的数据结构。

所谓前缀和，其概念如下：对于数列 $a$，可以将其储存为前缀和数组 $b$ 的形式，其中 $b_i = \sum_{j=1}^{i} a_j$。那么通过前缀和数组，就可以快速求出原数组中给定区间中数字的和：对于区间$[l,r]$，区间和为 $b_r - b_{l-1}$(假设 $b_0 = 0$）。

显然，对于长度为 $n$ 的数列，前缀和需要用长度为 $n$ 的数组进行储存。而当数列 $a$ 发生变化时，要使数组 $b$ 的内容仍能够正确对应数列 $a$ 的前缀和，就需要对 $b$ 的值进行修改。即使数列中只有一个数发生变化，也可能需要修改数组 $b$ 的多个值，才能保证整个数组仍然储存的是 a 的前缀和。

类似的，对于长度为 $n$ 的数列，树状数组也会使用长度为 $n$ 的数组来进行储存。在这个数组中，每个位置储存的内容则稍微有些复杂。

### 例 5-6 树状数组1(洛谷P3374)

已知一个数列，需要进行以下两种操作：

1. 将某一个数加上 $x$。
2. 求出某区间每一个数的和并输出。

数列长度和操作个数均不超过 $5 \times 10^5$。

**分析：**如果使用朴素的做法，将这个数列保存在一个数组 $a$ 中，那么对于第 2 种操作，需要将查询区间内的每一个数依次加起来。如果这样做，那么最坏情况下每一次操作就要遍历整个数组，导致超时。

另一种想法是，通过将数列储存为**前缀和**数组 $b$ 的形式，那么就可以快速求出给定区间的和；然而，对于第 1 种操作，在最坏情况下则又需要修改整个数组，同样会导致超时。

使用**树状数组**可以结合上面两种做法的优势，使得两个操作均使用较低的复杂度就能完成。

对于任何一种数据结构，可以把其抽象为一个黑匣子：黑匣子里面储存的是数据，可以向其提供操作，包括修改操作和查询操作。当向其提供查询操作时，黑匣子需要通过保存的数据计算出需要的结果然后返回；当向其提供修改操作时，黑匣子需要更新其内部的数据，来保证对于之后的查询操作，黑匣子仍能够返回正确的结果。能否解决问题，取决于这个黑匣子是否能以及能够以何种复杂度实现这些操作，而如何实现这样一个黑匣子，则是本题需完成的任务。

在本问题中，黑匣子需要维护一个数列，需要支持的有单点修改操作和区间和查询操作。表 5-1 列出了使用几种方法完成各项操作的时间复杂度的比较。

| 操作 | 数组 | 前缀和 | 线段树 | 树状数组 |
| --- | --- | --- | --- | --- |
| 单点修改 | O(1) | O(n) | O(log n) | O(log n) |
| 区间和查询 | O(n) | O(1) | O(log n) | O(log n) |

表5-1 各项操作的时间复杂度比较

和前缀和类似，树状数组每个位置保存的也是原数组中某一段区间的和。为了准确说明每个位置分别保存的是哪一段区间，需要使用函数 `lowbit(x)`。

`lowbit(x)` 的值是 $x$ 的二进制表达式中最低位的 1 所对应的值。例如，6 的二进制表示为 $\mathbf{110}_{(2)}$，最低位的 1 为第 2 个 1，其对应的值为 $\mathbf{10}_{(2)}=\mathbf{2}_{(10)}$，故 `lowbit(6)`=2；20 的二进制表示为 $\mathbf{10100}_{(2)}$，最低位的 1 为第 2 个 1，其对应的值为 $\mathbf{100}_{(2)}=\mathbf{4}_{(10)}$，故 `lowbit(20)`= 4。

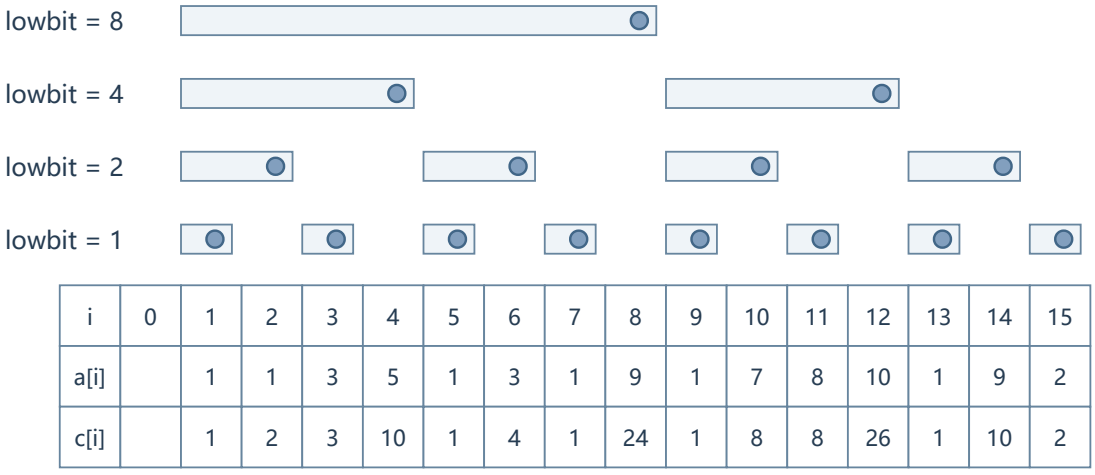
在常用的计算机中，有符号数采用补码表示，而在补码表示下，`lowbit(x)` 也有一种简单的求法：$\mathbf{lowbit}(x)=x\&(-x)$，其中&为按位与。由于$-x$ 的补码为 $x$ 按位取反再加 1，考虑 $x$ 和$-x$ 的二进制表示，$x$ 末尾的若干 0 在取反后变成 1，加上 1 后变成 $_{0;x}$ 最低位的 1 在取反后变成 0，得到进位后变成 1；比该位更高的不会得到进位，维持取反的状态。因此，在按位与过程中，只有那一位得到的结果为 1，其余都为 $\mathbf{0}$。例如：

$x$ 二进制表示：0101…1000…0

$-x$ 的反码表示：1010…0111…1

$-x$ 的补码表示：1010…1000…0

那么，假设树状数组使用数组 $c$ 来进行储存，原来的 $n$ 个数分别为 $a_1,a_2,\cdots,a_n$，则 $c_i=\sum_{j=i-\mathbf{lowbit}(i)+1}^{i}a_i$。换句话说，树状数组中每个位置保存的是其向前 `lowbit` 长度的区间和。如图 5-8 所示是一个树状数组的案例。




| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a[i] | | 1 | 1 | 3 | 5 | 1 | 3 | 1 | 9 | 1 | 7 | 8 | 10 | 1 | 9 | 2 |
| c[i] | | 1 | 2 | 3 | 10 | 1 | 4 | 1 | 24 | 1 | 8 | 8 | 26 | 1 | 10 | 2 |

图5-8 一个树状数组

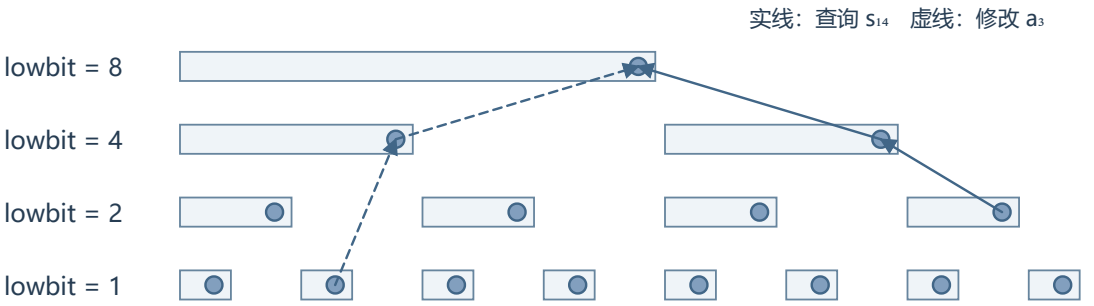
假设已有数组 $c$，要求前缀和 $s_i$，先把 $c_i$ 加入答案，剩余区间为 $a_1$ 到 $a_{i-\mathbf{lowbit}(i)}$。于是令 $i\leftarrow i-\mathbf{lowbit}(i)$，继续加入相应的 $c_i$；重复直到 $i=\mathbf{0}$。代码如下：

```
int sum(int x) {
    int res = 0;
    for (int i = x; i; i -= lowbit(i))
        res += c[i]; // 从大到小将需要的值求和
    return res;
}
```

这个过程的每一步中，把一个数 $x$ 变成 $x-$ `lowbit(x)`，结合之前所说的 `lowbit` 的含义，可以发现实际上是在不断地去掉 $i$ 的二进制表示中最低位的 1。由于一个数 $i$ 的二进制表示位数不超过 $\log_2 i$，故每一次查询的时间复杂度为 $O(\log n)$。

接下来再考虑单点修改操作。假设修改的数是 $a$，由于可能有多个位置对应的区间包含 $a_i a_i$，对于这些位置都要进行修改。

对于刚才给出的例子，如果要查询 $s_{14}$ 的值，可以发现 $s_{14}=c_{14}+c_{12}+c_8=64$；如果要修改 $a_3$ 的值，则需修改所有包含 $a_3$ 的区间值，也就是 $c_3 \setminus c_4$ 和 $c_{80}$ 这两个过程如图 5-9 所示。




| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a[i] | | 1 | 1 | 3 | 5 | 1 | 3 | 1 | 9 | 1 | 7 | 8 | 10 | 1 | 9 | 2 |
| c[i] | | 1 | 2 | 3 | 10 | 1 | 4 | 1 | 24 | 1 | 8 | 8 | 26 | 1 | 10 | 2 |

图5-9 树状数组修改和查询

有哪些位置包含 $a_i$ 呢?先推导几个结论。假设一个位置 $c_j$ 包含 $a_i$，那么有：

1. $j \geq i$。这一点很显然，因为一个位置只会包含它前面的数。
2. $\text{lowbit}(j) \geq \text{lowbit}(i)$，当且仅当 $j=i$ 时取等号。通过观察 $x$ 变为 $x-\text{lowbit}(x)+1$ 的过程，可以发现对于其中任何一个 $x$ 以外的数，考虑其二进制表示，总会至少有一个 1 比 $x$ 的 `low`-bit 位更低，即它的 `lowbit` 更小。
3. `lowbit` 值相等的位置不会包含同一个数。由上一点可知，`lowbit` 值相等的位置对应的区间一定不会互相重叠。

综合以上结论可以得知 $j$ 不小于 $i$，且 `lowbit(j)` 也不小于 `lowbit(i)`，不同的 $j$ 不会有同样的 `lowbit`。于是，可以按 `lowbit` 从小到大的顺序找出满足条件的 $j$。

首先，$i$ 是第 1 个满足条件的 $j$，记为 $j_0=i_s$

下二个 $i$ 需要比 $i$ 大，且 `lowbit` 也要更大，即二进制表示中末尾的 0 更多，因此至少需要把最后一个 1 变成 0，也就是至少加上 $\text{lowbit}(j_0)$；由于 $\text{lowbit}(j_0)<\text{lowbit}(j_0+\text{lowbit}(j_0))$，而 $j_0=i$，所以 $i$ 显然在 $j_0+\text{lowbit}(j_0)$ 对应的区间内，也就是说 $j_0+\text{lowbit}(j_0)$ 就是下一个 $j$，记为 $j_1=j_0+\text{lowbit}(j_0)$ 再下一个 $j$ 又可以通过 $j_1+\text{lowbit}(j_1)$ 得到。

由于 `lowbit` 是翻**倍增**长的，所以 `lowbit` $(j_0)+$ `lowbit(`$j_1$`)` 仍然小于 `lowbit` $(j_1)+\text{lowbit}(j_1)$，意味着 $i$ 也在 $j_1+\text{lowbit}(j_1)$ 对应的区间内，即 $j_2=j_1+$ `lowbit(j)`。

依此类推，即可得到所有需要修改的位置。代码如下：

```
void add(int x, int y) {
    for (int i = x; i <= n; i += lowbit(i))
        c[i] += y; // 从小到大修改需要修改的
    // 位置
}
```

由于 `lowbit` 的值只有不超过 `log(n)` 种，一次修改中一个 `lowbit` 值最多只会对应一个需要修改的位置，所以每一次修改的时间复杂度也为 $O(\log(n))$

至此，我们知道树状数组可以维护一个数列，并以 $O(\log(n))$ 的时间复杂度进行单点修改操作和前缀和查询操作。对于本题，要实现的是区间和查询操作，可以通过前缀和查询操作简单的实现：对于$[l,r]$的查询，只需要用[1，$r$]的和减去[1，$l-1$]的和即可。

此外，还有一个问题：如何得到初始的树状数组？可以认为开始时数列中所有的数都是 0，那么树状数组中每个位置也都是 0，然后依次修改每个数字为正确的值即可。代码如下：

```
#include <bits/stdc++.h>
const int MAXN = 500010;
int n, m;
int a[MAXN], c[MAXN];
int lowbit(int x) { return x & (-x); }
int sum(int x) {
    int res = 0;
    for (int i = x; i; i -= lowbit(i))
        res += c[i];
    return res;
}
void add(int x, int y) {
    for (int i = x; i <= n; i += lowbit(i))
        c[i] += y;
}
int main() {
    scanf("%d %d", &n, &m);
    for (int i = 1; i <= n; i++) {
        scanf("%d", &a[i]);
        add(i, a[i]); // 初始时全部为0,依次修改
    }
    while (m--) {
        int op, x, y;
        scanf("%d %d %d", &op, &x, &y);
        if (op == 1)
            add(x, y);
        else
            printf("%d\n", sum(y) - sum(x - 1));
    }
    return 0;
}
```

当然，有一种更加优雅的树状数组初始化的方法，其时间复杂度是线性的。代码如下

```
int pre[MAXN]; // a 数组的前缀和
void init() {
    for (int i = 1; i <= n; i++) {
        pre[i] = pre[i - 1] + a[i];
        c[i] = pre[i] - pre[i - lowbit(i)];
    }
}
```

虽然推导过程较为烦琐，但树状数组的代码十分简短。

### 例 5-7 树状数组2(洛谷P3368)

已知一个数列，需要进行以下两种操作：将区间 $[x,y]$ 每一个数加上 $x$；或者求出某一个数的值。

数列长度和操作个数均不超过 $5\times10^5$。

**分析：**和 例 5-6 问题相反，这里需要对数列实现区间加法的修改操作和单点的查询操作。乍一看好像没办法使用**树状数组**，但实际上只需要进行一些小处理，就能把这个问题变得和上一个问题相同。

对数组进行差分操作：假设原来的数列为 $a$，令 $b_i=a_i-a_{i-1}$，那么 $a_i=\sum_{j=1}^{i}b_j$，即 $a$ 是 $b$ 的**前缀和**数组。当 $b_i$ 增加 $x$ 时，意味着 $a_i$ 到 $a_n$ 都会增加 $x$。

那么，对于数组 $b$ 而言，第 1 个操作的效果为：假设要将区间$[l,r]$的数增加 $x$，则 $b_l$ 增加 $x,b_{r+1}$ 减少 $x$；第 2 个操作的效果为：求出 $b$ 的某个前缀和。这样一来，数组 $b$ 就可以用树状数组进行维护。代码如下：

```
#include <bits/stdc++.h>
const int MAXN = 500010;
int n, m;
int a[MAXN], b[MAXN], c[MAXN];
int lowbit(int x) { return x & (-x); }
void add(int x, int y) {
    for (int i = x; i <= n; i += lowbit(i))
        c[i] += y;
}
int sum(int x) {
    int res = 0;
    for (int i = x; i; i -= lowbit(i))
        res += c[i];
    return res;
}
int main() {
    scanf("%d %d", &n, &m);
    for (int i = 1; i <= n; i++) {
        scanf("%d", &a[i]);
        b[i] = a[i] - a[i - 1]; // 差分操作
        add(i, b[i]);
    }
    while (m--) {
        int op, x, y;
```

```
        scanf("%d", &op);
        if (op == 1) {
            int x, y, k;
            scanf("%d %d %d", &x, &y, &k);
            add(x, k);
            add(y + 1, -k);
        } else {
            int x;
            scanf("%d", &x);
            printf("%d\n ", sum(x));
        }
    }
    return 0;
}
```

本题是使用树状数组实现区间修改与单点查询的操作。实际上，通过一些技巧，树状数组也可以实现区间修改与区间查询的操作。感兴趣的读者可以自行查阅资料。

### 例 5-8 逆序对(洛谷P1908)

对于给定的一段正整数序列，逆序对就是序列中 $a_i > a_j$ 且ij的有序对。给定长度为 $n$ 的正整数序列，求逆序对数，其中 $n \leqslant 5 \times 10^5$。

**分析：**首先考虑朴素的做法，即枚举 $i$，再枚举比 $i$ 大的位置 $j$，统计 $a_j < a_i$ 的数量。假设把所有 $j > i$ 中 $a_j = k$ 的数量记为 $t_k$，那么也就是统计 $s_{a_i-1} = \sum_{k=1}^{a_i-1} t_k$。

也就是说，查询的是一个数列 $t$ 的**前缀和**。如果按照从大到小的位置枚举 $i$，那么每当 $i$ 前进一步，可用的 $j$ 就增加一个，需要将 $t_{a_j}$ 增加 $1$。可以发现，这是不断地在对数列 $t$ 进行前缀和查询和单点修改操作，因此可以用**树状数组**维护数列 $t$

但是还有一个问题：数列 $t$ 的长度是多少呢？由于 $a$ 的值可以很大，所以 $t$ 的下标也可以很大。为了解决这个问题，可以采用**离散化**的思想。由于数组 $t$ 开始时全为 0，总共会进行 $n$ 次修改，也就是说最多只有 $n$ 个位置不是 0，因此可以只记录这些可能非 0 的位置。具体而言，首先将序列排序并去重，在这个序列上使用std：：lower`_bound()`，可以快速求出原数列中一个数是数列中的第几小。那么 $t_k$ 可以表示序列中第 $k$ 小的数的个数。这样一来，$t$ 的长度最多就是 $n$ 了。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 500010;
int m;
int a[MAXN], b[MAXN], c[MAXN];
int lowbit(int x) { return x & (-x); }
void add(int x, int y) {
    for (int i = x; i <= m; i += lowbit(i))
        c[i] += y;
}
int sum(int x) {
    int res = 0;
    for (int i = x; i; i -= lowbit(i))
        res += c[i];
    return res;
}
```

```
int main() {
    int n;
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        scanf("%d", &a[i]);
        b[i] = a[i];
    }
    sort(b + 1, b + n + 1);
    m = unique(b + 1, b + n + 1) - b - 1; // 离散化
    long long ans = 0;
    for (int i = n; i; i--) {
        int k = lower_bound(b + 1, b + m + 1, a[i]) - b; // 求出当前数的排名
        ans += sum(k - 1);
        add(k, 1);
    }
    printf("%lld", ans);
    return 0;
}
```

其实，除了前缀和以外，类似**线段树**或树状数组也可以维护如前缀最值等广义的“前缀和”。但要注意的是，并不是所有的“区间和”都能通过维护“前缀和”来实现，因为要求区间满足“可减性”：如前缀最值，知道[1，$r$]的最值和[1，$l-1$]的最值并不能得到$[l,r]$的最值。

## 5.4 课后习题与实验

### 习题 1 回顾堆排序的第一步:把数组中所有元素逐一压进堆。这一步的时间复杂度是多少？能改进算法的效率吗？基于这个优化,改写堆排序的代码,完成洛谷P1177,并比较与书中代码的耗时差异

### 习题 2 黑匣子(洛谷P1801)

黑匣子可以储存一个整数数组，还有一个特别的变量 $i$ 最开始的时候黑匣子是空的，而 $i=0$。这个黑匣子需要处理以下两种命令

1. `ADD(x)`：把 $x(|x|\leqslant 2\times 10^9)$ 元素放进黑匣子。
2. GET：$i$ 加 $1$，然后输出黑匣子中第 $i$ 小的数。

ADD命令共有 $m$ 个，GET命令共有 $n$ 个 $(1\leqslant m,n\leqslant 2\times 10^5)$。现在用两个整数数组来表示命令串。

1. $a_1,a_2,\cdots,a_m$：一串将要被放进黑匣子的元素。
2. $u_1,u_2,\cdots,u_n$：表示第 $u$：个元素被放进了黑匣子里后就出现一个GET命令。输入数据 $u_i$ 不用判错。

请在每次遇到GET命令后，输出对应的数字。

### 习题 3 中位数(洛谷P1168)

给出一个长度为 $N(1\leqslant N\leqslant 10^5)$ 的非负整数序列 $A_i(A_i\leqslant 10^9)$，对于所有 $1\leqslant k\leqslant (N+1)/2$ 输出 $A_1,A_1\sim A_3,\cdots,A_1\sim A_{2k-1}$ 的中位数。

即前 $1$，$3$，$5$，…个数的中位数。

### 习题 4 序列合并(洛谷P1631)

有两个长度都是 $N(1 \leqslant N \leqslant 10^5)$ 的非负整数序列 $A$ 和 $B$，其值域不超过 $10^9$.。在 $A$ 和 $B$ 中各取一个数相加可以得到 $N^2$ 个和，求这些和中最小的 $N$ 个。

### 习题 5 建筑抢修(洛谷P4053,江苏省队选拔2007)

有 $N(1 \leqslant N \leqslant 1.5 \times 10^5)$ 个建筑需要修复，同一时间只能修一个建筑，第 $i$ 个建筑修复需要 $A_i$ 时间，且必须在 $B_i$ 时间前修完 $(A_i, B_i \leqslant 2^{31}-1)$，否则此建筑将会倒塌。请计算最多能修好的建筑数量。

### 习题 6 最小函数值(洛谷P2085)

有 $n$ 个函数，分别为 $F_1, F_2, \cdots, F_n$ 定义 $F_i(x) = A_i x^2 + B_i x + C_i (x \in \mathbb{N}^*)$。

给定 $A_i, B_i$ 和 $C_i$ 请求出所有函数的所有函数值中最小的 $m$ 个（如有重复的要输出多个）。

**数据范围：** $1 \leqslant n, m \leqslant 10000, 1 \leqslant A_i \leqslant 10, B_i \leqslant 100, C_i \leqslant 10^4$。

### 习题 7 操作系统(洛谷P2278,湖南省队选拔2003)

假设系统只有一个CPU，每一个进程的到达时间、执行时间和运行优先级都是已知的。其中运行优先级用自然数表示，数字越大，则优先级越高。

如果一个进程到达的时候CPU是空闲的，则它会一直占用CPU直到该进程结束。除非在这个过程中，有一个比它优先级更高的进程要运行。在这种情况下，这个新的（优先级更高的）进程会占用CPU，而老的进程则只有等待。如果一个进程到达时，CPU正在处理一个比它优先级高或优先级相同的进程，则这个（新到达的）进程必须等待。

一日CPU空闲，如果此时有进程在等待，则选择优先级最高的先运行。如果有多个优先级最高的进程，则选择到达时间最早的。

输入文件有若干行，每一行有 $4$ 个自然数（均不超过 $10^8$） ，分别是进程号、到达时间、执行时间和优先级。不同进程有不同的编号，不会有两个相同优先级的进程同时到达。输入数据已经按到达时间从小到大排序。输入数据保证在任何时候，等待队列中的进程不超过 $15000$ 个。

请按照进程结束的时间输出每个进程的进程号和结束时间。

### 习题 8 在常用的计算机中，有符号数采用补码表示。如果有一台采用反码表示的计算机，该如何实现lowbit函数?如果是原码表示呢?

### 习题 9 火柴排队(洛谷P1966,NOIP 2013提高组)

涵涵有两盒火柴，每盒装有 $n$ 根火柴，每根火柴都有一个高度。现在将每盒中的火柴各自排成一列，同一列火柴的高度互不相同，两列火柴之间的距离定义为： $\sum (a_i - b_i)^2$，其中 $a_i$ 表示第 $1$ 列火柴中第 $i$ 个火柴的高度 $b_i$ 表示第 $2$ 列火柴中第 $i$ 个火柴的高度。

每列火柴中相邻两根火柴的位置都可以交换，请通过交换使得两列火柴之间的距离最小。请问得到这个最小的距离，最少需要交换多少次？如果这个数字太大，请输出这个最小交换次数对 $10^8 - 3$ 取模的结果。

### 习题 10 配对统计(洛谷P5677,贵州省队选拔2017)

给定 $n$ 个数 $a_1, a_2, \ldots, a_n$。若配对 $(x, y)$ 满足 $x \neq y$，且对所有 $i \neq x$ 均有 $|a_x - a_y| \leq |a_x - a_i|$，则称其为一组好的配对（$|x|$ 表示绝对值）。给出若干询问，每次求区间 $[l, r]$ 中满足 $l \leq x, y \leq r$ 的好配对数量。

### 习题 11 会场预约(洛谷P2161,上海市省队选拔2009)

需要维护一个在数轴上的线段的集合 $S$，支持以下两种操作。

1. $A$ lr：将 $S$ 中所有与线段 $[l, r]$ 相交的线段删去，并将 $[l, r]$ 加入 $S$ 中，并输出删掉的元素个数。
2. $B$：查询 $S$ 中的元素数量。

**数据范围：** $1 \leqslant n \leqslant 2 \times 10^5, 1 \leqslant l \leqslant r \leqslant 10^5$

### 习题 12 削平(洛谷P3586,POI2015)

维护一个长度为 $n$ 的序列，一开始都是 $0$，支持以下两种操作。

1. $U\ k\ a$：将序列中第 $k$ 个数修改为 $a$。
2. $Z\ c\ s$：在这个序列上，每次选出 $c$ 个正数，并将它们都减去 $1$，询问能否进行 $s$ 次操作。每次询问相互独立，即每次询问不会对序列进行修改。

**数据范围：** $1 \leqslant n, m \leqslant 10^6, 1 \leqslant k, c \leqslant n, 0 \leqslant a \leqslant 10^9, 1 \leqslant s \leqslant 10^9$.。

# 第6章 线段树

如何快速求出一个序列的区间和呢？可以使用**前缀和**。如何快速求出一个序列的最值呢？可以使用ST表。以上两种数据结构在建立的时候颇费工夫，但使用的时候效率很高。如果再增加一个需求：时不时地修改序列的值，那么这两种数据结构就无法高效完成了。不过可以使用**线段树**来解决这类问题。

在《基础篇》中已经学习了二叉树的有关概念，而线段树是一种特殊的二叉树，它可以将一个线性的序列组织成一个树状的结构，从而可以在对数的时间复杂度下访问序列上的任意一个区间并进行维护。在本章中，将学习线段树的构建方法和一些简单的应用，其思维导图如图 6-1 所示。




图6-1 本章思维导图

## 6.1 线段树的建立与操作

### 例 6-1 线段树1(洛谷P3372)

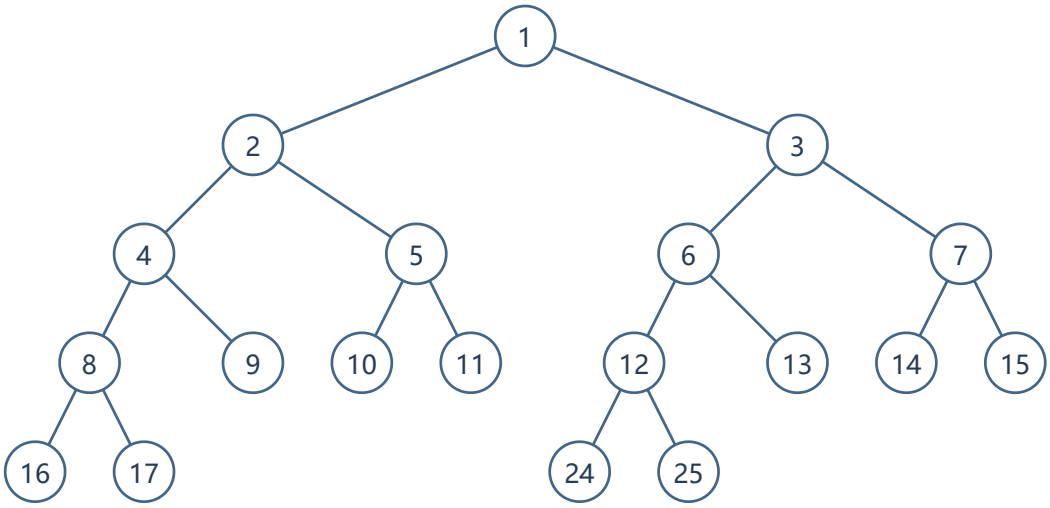
已知一个数列 $a_i$，需要进行以下两种操作：

1. 将区间 $[x,y]$ 内每一个数加上 $k$。
2. 求出某区间 $[x,y]$中每一个数的和。

数列个数和操作次数不超过 $10^5$，$a_i,k$ 以及变化后的数列数字的绝对值不超过 $2^{63}-1$

**分析：**如果使用数组直接存储和维护这个数列，虽然修改的效率是 $O(1)$，但求和的效率是 $O(n)$。以下介绍一种用于维护序列的树形数据结构——**线段树**。

线段树的思想在于将序列中若干个区间在树上用结点表示，其中[1，$n$]区间（$n$ 表示序列长度）是树的根。而对一个表示区间$[l,r]$的结点（$l \neq r$），设 $mid = \left[\frac{l+r}{2}\right]$，将[$l$，mid]和[mid+1，$r$]作为该结点的左子结点和右子结点。例如，对于一个长度为 10 的序列，它所构建的线段树如图 6-2(a) 所示，各个结点所代表的区间如图 6-2(b) 所示。




(a)

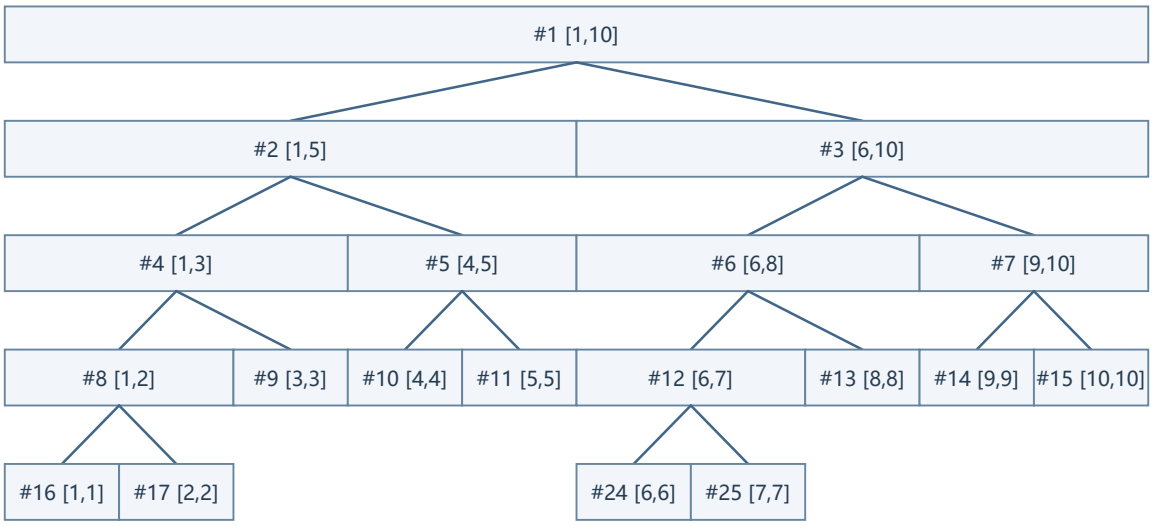



图6-2一个线段树的例子

在 图 6-2 中，对长度为 10 的序列构建线段树，将结点[1，10]作为根结点，设 $mid = \left[\frac{1+10}{2}\right] = 5$，将[1，5]作为根结点的左子结点，[6，10]作为根结点的右子结点。这两个结点的子树构建方法同上。

观察 图 6-2 中的线段树，不难得到如下性质：

1. 对于线段树上的任意一个结点，它要么没有子结点，要么有两个子结点，不存在只有一个子结点的情况。
2. 对于一个长度为 $n$ 的序列，它所建立的线段树只有（2n-1）个结点。
3. 对于一个长度为 $n$ 的序列，它所建立的线段树高为logn。

以下对第 2 条和第 3 条性质加以证明：

对于第 2 条性质，首先线段树有且仅有 $n$ 个叶结点，初始时它们没有父结点。然后，将没有父结点的结点进行两两合并，每次合并会新建一个结点，共给两个结点建立了父结点，新增了一个没有父结点的点。也就是每次合并会新增一个结点，没有父结点的点数减一。最终线段树有且仅有一个没有父结点的结点，因此总共新建了（$n-1$）个结点，加上初始的 $n$ 个叶结点，总共（2n-1）个结点。

对于第 3 条性质，考虑表示区间 $[l,r]$ 的结点，设 $\text{len}=r-l+1$。当 $\text{len}$ 为偶数时，两个子区间长度均为 $\text{len}/2$；为奇数时，两个子区间的长度分别为 $(\text{len}+1)/2$ 和 $(\text{len}-1)/2$。因此，每深入一层，区间长度至多变为原来的一半向上取整。经过 $\lceil \log_2 n \rceil$ 次划分后，区间长度为 $1$，故树高为 $O(\log n)$。

线段树中一个结点上可以维护若干所需要的信息，在访问时，将若干结点的信息合并，就能得到任意所需区间的信息。例如，在 图 6-2 中，如果希望获得区间[1，4]的信息，只需要将结点[1，3]和结点[4，4]的信息合并即可。在这里需要注意的是，使用线段树所维护的信息必须具有可合并性。

例如，如果要求区间和，区间[1，3]的和加上区间[4，4]的和显然就是区间[1，4]的和；但是简单的线段树难以直接用于维护区间众数，因为区间 1,12 的众数不一定是区间[1，6]的众数和区间[7，12]的众数中出现次数较多的。例如考察数列（3，3，1，1，1，2，3，3，4，5，5，5），区间[1，6]的众数是出现了 3 次的 1，区间[7，12]的众数是出现了 3 次的 5，但区间[1，12]的众数却是出现了 4 次的 3。

在《基础篇》中已经介绍了二叉树的存储方法：将根结点定义为 1 号结点；对于编号为 $i$ 的结点，它的左子结点编号为 2i，右子结点编号为 2i+1。不难发现，这样每个结点都有且仅有唯一的编号与之对应，且在线段树上，结点的最大编号不超过 4n-1，其中 $n$ 是序列长度。

线段树支持以下操作：

1. 建立线段树

树是递归定义的，因此可以用递归的方式建立线段树：如果这个区间左端点等于右端点，说明是叶子结点，其数据赋值为对应数列元素的值；否则将这个区间分为左右两部分，分别递归建立线段树，然后将左右两个区间的数据进行汇总（pushup）处理。

回到 例 6-1，假设初始数列是（1，5，4，2，3），那么建立好的线段树如图 6-3 所示，其中圆括号里的数字是这个区间的数字和。显然，非叶子结点的区间数字和是其左右两子结点的区间数字和的和。

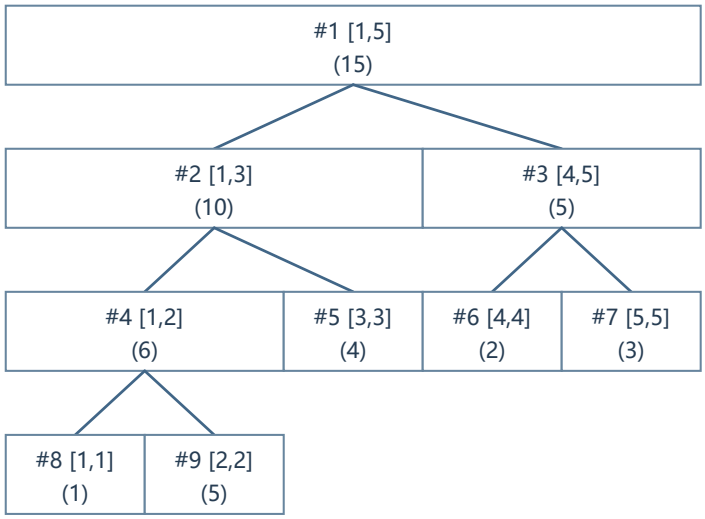



图6-3 建立好的线段树

建立线段树的代码如下：

```
#define ll long long
ll a[maxn], w[maxn * 4];
void pushup(const int u) {
    w[u] = w[u * 2] + w[u * 2 + 1]; // w[u]是区间和,u*2是左子树,u*2+1是右子树
}
void build(const int u, int L, int R) {
    if (L == R) { // 到达叶结点
        w[u] = a[L];
        return;
    }
    int M = (L + R) / 2; // 将区间分成[L,M]和[M+1,R]并递归构造
    build(u * 2, L, M);
    build(u * 2 + 1, M + 1, R);
    pushup(u); // 由子区间的区间和更新当前区间的和
}
```

在以上代码中，$u$ 表示当前线段树结点的编号，$w$ 是线段树所维护的信息，也就是区间和。如果已经到达了叶结点，那么区间和显然就是对应位置的和，直接赋值即可；否则递归构建左右子树，然后通过pushup函数将左右子树所维护的区间和进行合并。不难发现，每调用一次build函数，就新建了一个线段树结点，因此该函数的时间复杂度为 $O(n)$。

整型的乘/除 2 的幂的运算可以通过左右移位运算实现，如代码中（$L+R$）/2 可以改为（$L+R$）>>1，$u*2+1$ 可以改成（$u$<<1）+1。在一些版本较低的编译器上，这样写可能可以带来一定的效率提升。

2. 单点查询与修改

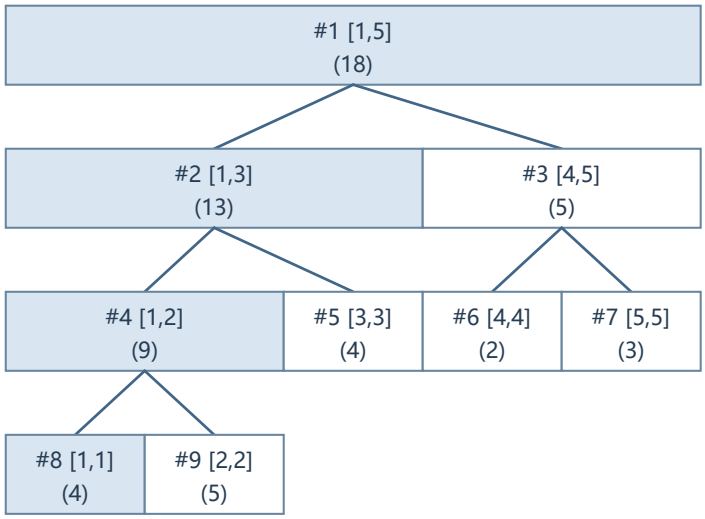
如何精确定位到叶子结点呢？假设需要定位到 $p$ 这个结点，实际上是需要找到[$p$，$p$ 这个区间。初始时，该结点在根结点[1，$n$]的子树中。根结点的左子结点为[1，$M$]，右子结点为[$M+1$，$n$，其中 $M$ =（$1+n$）/2。如果 $p \leqslant M$，那么目标结点显然在左子树中，向左递归即可；否则，目标结点在右子树中，需要向右递归。单点修改也是类似的过程，如果进行的是修改（更新操作），在返回时需要一路pushup，来保证线段树信息的正确性。

例如将数列第 1 个数字加上 3 时，则先找到对应的叶子结点（也就是 8 号）并更新它的数字，然后一直往这个结点的父结点更新区间和，直到根结点为止。这个过程如图 6-4(a) 所示。

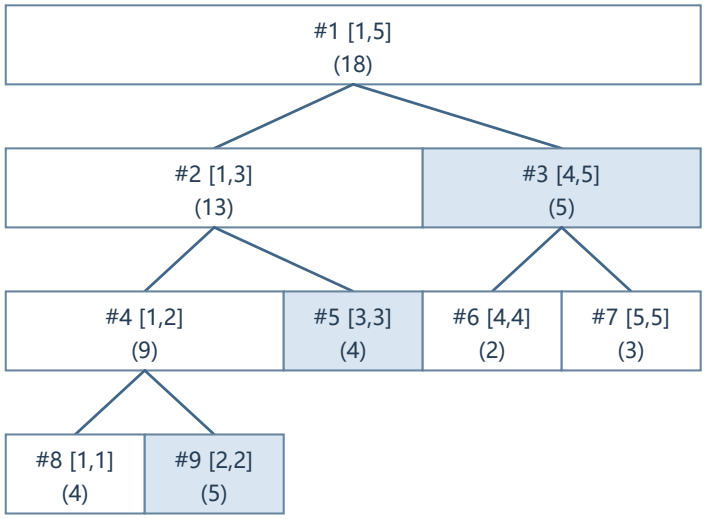
单点查询和单点修改的代码如下：

```
ll query1(int u, int L, int R, int p) {
    if (L == R) { // 到达叶结点则返回即可
        return w[u];
    } else {
        int M = (L + R) / 2;
        if (M >= p) // 如果查询的位置在左子树[L,M]内,就递归查询左子树
            return query1(u * 2, L, M, p);
        else // 反之查询右子树
            return query1(u * 2 + 1, M + 1, R, p);
    } // 因为查询没有对区间和进行修改,因此不需要pushup
}
void update1(int u, int L, int R, int p, ll x) {
    if (L == R) { // 到达叶结点则直接赋值
        w[u] = x;
    } else {
        int M = (L + R) / 2; // 若修改的位置p在左子树[L,M]范围内,递归修改左子树
        if (M >= p)
            update1(u * 2, L, M, p, x);
        else // 反之修改右子树
            update1(u * 2 + 1, M + 1, R, p, x);
        pushup(u); // 别忘记更新以后需要修改当前结点的和
    }
}
```

在以上代码中可以发现，每递归调用一次函数，都会在线段树上向下移一层。因为线段树的树高是 $O(\log n)$，所以递归函数只会调用 $O(\log n)$ 次。也就是说，线段树的单点操作复杂度为 $O(\log n)$。




（a）修改第 1 个数




（b）查询区间 [2,5]

图6-4线段树的单点修改与区间查询

3. 区间查询

只能支持单点操作的线段树是没有意义的，因为可以直接在数组上完成。接下来将介绍如何用线段树快速维护区间信息。对于 例 6-1，就是给定区间$[l,r]$，求这个区间的数字和。

从根开始递归，如果当前结点所代表的区间$[L,R]$被所询问的区间$[l,r]$包含，那么直接返回当前区间的区间和；如果两个区间没有交集，那么直接返回 0；如果没有被包含且两区间有交集，则递归左右子结点处理即可。

例如，查询[2，5]区间时，相当于查询[2，2]、[3，3]和[4，5]这些区间的数据，然后进行汇总即可得到答案，如图 6-4(b) 所示。

区间查询的代码如下：

```
bool InRange(int L, int R, int l, int r) {
    // 判断[L,R]是否被[1,r]包含
    return (l <= L) && (R <= r);
}
bool OutofRange(int L, int R, int l, int r) {
    // 判断[L,R]是否和[1,r]完全无交集
    return (L > r) || (R < l);
}
ll query(int u, int L, int R, int l, int r) {
    if (InRange(L, R, l, r)) { // 如果完全包含则直接返回区间和
        return w[u];
    } else if (!OutofRange(L, R, l, r)) { // 若有相交一部分,则需要递归处理
        int M = (L + R) / 2;
        return query(u * 2, L, M, l, r) + query(u * 2 + 1, M + 1, R, l, r);
    } else
        return 0; // 否则一定是完全无交集
}
```

在query函数中，并不是每层只会向下延伸一个结点，而是对左右子结点分别递归。那么，如何计算其复杂度呢？在线段树每层的递归中，最多只有两个结点会向下继续递归，也就是被查询区间两端点所在的结点，而剩下的结点要么被完全包含，要么与查询区间无交。因此，每一层只会新建两个递归函数。因为树高是 $O(\log n)$，所以总的复杂度还是 $O(\log n)$。

4. 区间修改

在区间修改时，显然不能暴力修改每个叶子，那样效率很低。为此，引入**延迟标记**（又称为懒标记或者lazy-`tag`）记录一些区间修改的信息。当递归至一个被完全包含的区间时，在这个区间上打一个延迟标记，记录这个区间中的每个数都需要被加上某个数，然后直接修改该结点的区间和并返回，不再向下递归。当新访问到一个结点时，先将延迟标记下放到子结点，然后再进行递归。

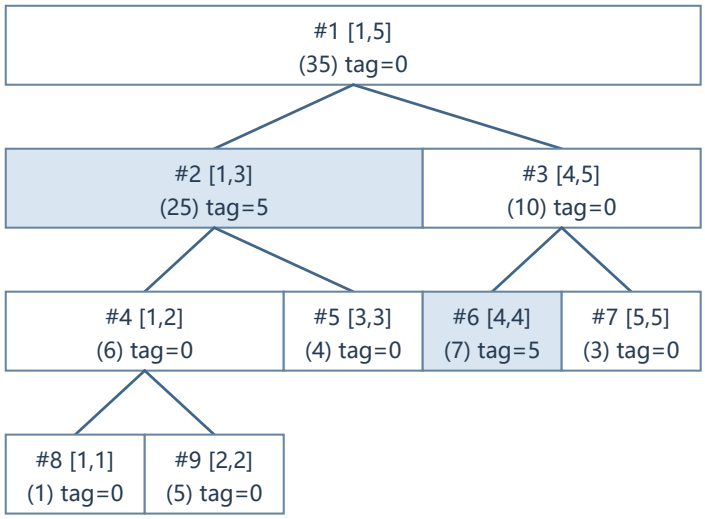
可以发现，这样做可以保证与根相连的某个连通块的信息总是正确的，并且在调用时总能得到正确的信息。同时，因为被完全包含和无交集的情况都直接返回不再递归，所以复杂度同区间查询一样可以证明是 $O(\log n)$。

假设初始数列为 $(1,5,4,2,3)$，对区间 $[1,4]$ 的每个数加上 $5$。修改区间被拆成 $[1,3]$ 和 $[4,4]$：前者的和由 $10$ 增加 $3\times5=15$，变为 $25$；后者由 $2$ 增加 $5$，变为 $7$。两个结点的延迟标记均增加 $5$。区间 $[1,4]$ 的和变为 $25+7=32$；根结点表示的 $[1,5]$ 的和为 $32+3=35$，如图 6-5（a） 所示。

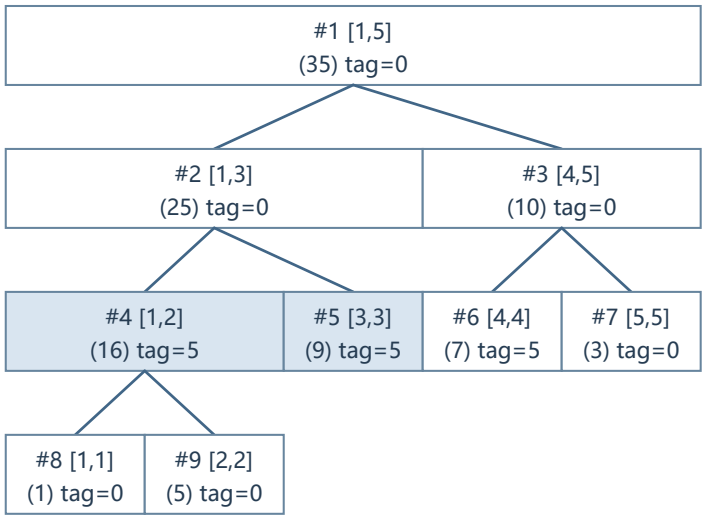
此时查询[3，5]区间和，遍历到[1，3]区间时，发现需要继续遍历下去，这回将标记下放到这个结点的两个子结点。子结点的 `tag` 值要增加父结点的 `tag` 值，同时维护区间和的 $w$ 值也要加上区间长度乘以父结点的 `tag` 值。下放后父结点的 `tag` 值要清空，结果如图 6-5(b) 所示。

从以上可以看出，对于打了延迟标记的结点，其维护的区间和是已经修改完成的值，其子结点的值还没有被修改。也就是说，延迟标记起到的作用是记录子结点的每个数应该加上多少，而不是该结点本身的信息。代码如下（查询的过程别忘了下放标记的过程）：

```
ll lzy[maxn * 4];
void maketag(int u, int len, ll x) {
    lzy[u] += x;     // 修改当前结点的延迟标记
    w[u] += len * x; // 修改当前结点的区间和
}
```




（a）区间加 5 后




（b）下放 #2 的标记

父标记清零，两个子结点累加标记 5

图6-5 延迟标记

```
void pushdown(int u, int L, int R) {
    int M = (L + R) / 2;
    maketag(u * 2, M - L + 1, lzy[u]); // 给左子树加上 lzy[u]
    maketag(u * 2 + 1, R - M, lzy[u]); // 给右子树加上lzy[u]
    lzy[u] = 0;
    // 因为标记信息已经下传到下一层结点了,为了避免重复修改,当前层要清空标记
}
```

```
ll query(int u, int L, int R, int l, int r) {
    if (InRange(L, R, l, r)) { // 如果完全包含,则直接返回区间和
        return w[u];
    } else if (!OutofRange(L, R, l, r)) { // 若有相交一部分,则需要递归处理
        int M = (L + R) / 2;
        pushdown(u, L, R); // 查询的时候也需要将结点标记下放
        return query(u * 2, L, M, l, r) + query(u * 2 + 1, M + 1, R, l, r);
    } else
        return 0; // 否则一定是完全无交集
}
void update(int u, int L, int R, int l, int r, ll x) {
    if (InRange(L, R, l, r)) {
        maketag(u, R - L + 1, x); // 完全包含则直接打标记即可
    } else if (!OutofRange(L, R, l, r)) {
        int M = (L + R) / 2;
        pushdown(u, L, R); // 注意:必须先将当前结点的标记下传,才能递归修改下面
        // 的结点
        update(u * 2, L, M, l, r, x);
        update(u * 2 + 1, M + 1, R, l, r, x);
        pushup(u);
    }
}
```

以上代码中，`pushdown` 函数是将延迟标记下传的过程，`maketag` 函数是更新延迟标记的过程。数组 `lzy` 记录的是当前结点应该加的值的大小，然后该区间的区间和需要增加的值就是长度乘上增加量。需要注意的是，在将标记下传后，应该清空当前结点的延迟标记，并且必须要先判断目标区间在当前区间内，再进行 `pushdown`，否则一旦在叶结点 `pushdown`，可能会造成数组越界。本题主函数的代码如下：

**代码片段（需结合上文的声明和函数）：**

```
int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++)
        scanf("%lld", &a[i]);
    build(1, 1, n);
    for (int t = 1; t <= m; t++) {
        int op, x, y;
        ll k;
        scanf("%d", &op);
        if (op == 1) {
            scanf("%d%d%lld", &x, &y, &k);
            update(1, 1, n, x, y, k);
        } else {
            scanf("%d%d", &x, &y);
            printf("%lld\n ", query(1, 1, n, x, y));
        }
    }
    return 0;
}
```

至此，我们已经掌握了基础的线段树操作。当然，如果不喜欢有太多的数组，也可以将线段树的结点定义为结构体，这样在写各种操作的函数时就不需要传递代表结点的区间结点了。

```
struct node {
    int L, R;       // 记录结点代表的区间的左右端点
    long long w;    // 存储数据
    long long tag; // 标记值
} tree[maxn * 4];
```

另外，建立线段树也可以使用update函数，但这样建立线段树的时间复杂度由原来的 $O(n)$ 变为 $O(n\log n)$，效率稍低。

## 6.2 线段树的应用

**线段树**标记使用的条件是，标记的信息必须具有“可加性”。例如，对区间先加 $a$，再加 $b$，那么最终区间的状态就是加上了 $a+b$；对区间先异或 $a$，再异或 $b$，那么区间的最终状态就是异或了 $a$ xor $b$

### 例 6-2 开关(洛谷P3870,天津市NOI代表队选拔2009)

给定一个初始为 $0$ 的数列，要求支持：

1. 给区间$[a,b]$的所有数字对 $1$ 取异或。
2. 求区间$[a,b]$内 $1$ 的个数。

**数据范围：** $1\leqslant n,m\leqslant 10^5$。

**分析：**不难发现，所求“区间内 1 的个数”这一问题具有可合并性，并且“区间异或”这一操作很容易通过**线段树**的**延迟标记**实现，定义延迟标记的含义为区间内所有数字都异或上该值，修改时，将延迟标记也异或上 1。在每次异或 1 时，原有的 1 会变成 0，原有的 0 会变成 1，也即区间 1 的个数会变成 $r-l+1-w$，其中 $w$ 为原有的 1 的个数。将make_tag函数修改如下即可：

```
void maketag(int u, int l, int r) {
    w[u] = r - l + 1 - w[u];
    tag[u] ^= 1;
}
```

### 例 6-3 无聊的数列(洛谷P1438)

维护一个数列 $a_i$。要求支持：

1. 1lrKD：给出一个长度等于 $r-l+1$ 的等差数列，首项为 $K$，公差为 $D$，并将它对应加到$[l,r]$范围中的每一个数上，即令 $a_l=a_l+K,a_{l+1}=a_{l+1}+K+D,\cdots,a_r=a_r+K+(r-l)D$。
2. 2p：询问序列的第 $p$ 个数的值 $a_p$c

**分析：**这是一个区间加等差数列的问题。考虑等差数列有两个要素：首项 $x$，公差 $y$ 只要这两项确定了，等差数列就唯一确定了。考虑这两个要素也具有“可加性”：将首项分别为 $x_1$ 和 $x_2$、公差分别为 $y_1$ 和 $y_2$ 的两个等差数列的每一项对应相加，得到的数列也是一个等差数列，且它的首项为 $x_1+x_2$，公差为 $y_1+y_2$。

例如，将数列 1，2，3（首项为 1，公差为 1）和 1，3，5（首项为 1，公差为 2）对应相加，得到数列 2，5，8。它的首项为 $1+1=2$，公差为 $1+2=3$

于是使用两个**延迟标记**，分别表示首项和公差即可。

```
long long S(int ax, int ay, int len) {
    long long an = ax + ay * (len - 1LL);
    return 1LL * (ax + an) * len / 2;
}
void maketag(int u, int l, int r, int add_x, int add_y) {
    x[u] += add_x;
    y[u] += add_y;
    w[u] += S(add_x, add_y, r - l + 1);
}
```

### 例 6-4 扶苏的问题(洛谷P1253)

给定一个长度为 $n$ 的序列 $a$，要求支持 $q$ 次操作，共有 $3$ 种：

1. 给定区间$[l,r]$，将区间内每个数都修改为 $x$。
2. 给定区间$[l,r]$，将区间内每个数都加上 $x$。
3. 给定区间$[l,r]$，求区间内的最大值。

**数据范围：** $1 n , q ^ { 6 } , - 10 ^ { 9 } a _ { i } , x ^ { 9 } $。

**分析：**本题中所求的“区间最大值”也可以使用**线段树**维护：父结点的区间最大值就是它两个子结点的区间最大值中较大的那个。

对于修改操作，可以使用两个**延迟标记**：一个表示区间赋值为 $x$（不妨记为 $t_1$） ，另一个表示区间加上 $x$（不妨记为 $t_2$） 。当对一个结点执行操作 1 时，直接将 $t_1$ 赋值为 $x,t_2$ 清空；执行操作 2 时，若 $t_1$ 存在，则将 $t_1$ 加上 $x_i$ 否则将 $t_2$ 加上 $x$ 下传时，如果 $t_1$ 存在则下传 $t_1$，否则下传 $t_2$。

需要注意的是，因为可能赋值为 0 或负数，所以需要用一个不可能出现的数（如 $10^{18}$） 来表示 $t_1$ 不存在。

```
#define maxn 1000010
#define Inf 1000000000000000000LL
#define ll long long
ll a[maxn], w[maxn * 4];
ll lzy_add[maxn * 4], lzy_set[maxn * 4]; // 区间加法标记和区间赋值标记
void pushup(const int u) {
    w[u] = std::max(w[u * 2], w[u * 2 + 1]); // w[u]是区间最大值
}
// 为节省篇幅,本例中build函数和例6-1的build函数基本一致,但需要加上lzy_set
//[u] = Inf;
// InRange和OutofRange函数和例6-1的也是一致的,本处略
void maketag(int u, ll x, int type) { // type 表示是哪种操作
    if (type == 1) {                  // 设置区间为某个数字
        lzy_add[u] = 0;
        lzy_set[u] = x;
        w[u] = x;
```

```
    } else { // 区间的累加和
        if (lzy_set[u] == Inf)
            lzy_add[u] += x;
        else
            lzy_set[u] += x;
        w[u] += x;
    }
    // 修改当前结点的区间和
}
void pushdown(int u) {
    if (lzy_set[u] == Inf) {
        maketag(u * 2, lzy_add[u], 2);
        maketag(u * 2 + 1, lzy_add[u], 2);
        lzy_add[u] = 0;
    } else {
        maketag(u * 2, lzy_set[u], 1);
        maketag(u * 2 + 1, lzy_set[u], 1);
        lzy_set[u] = Inf;
    }
    // 因为标记信息已经下传到下一层结点了,为了避免重复修改,当前层要清空标记
}
// query函数和例6-1的基本一致,只是最后返回的是两个子树的较大值,本处略
// update函数和例6-1的基本一致,但该函数和它调用的maketag函数需要加一项参数type
// 表示是哪种操作,本处略
```

### 例 6-5 线段树2(洛谷P3373)

给定一个长度为 $n(1 \leqslant n \leqslant 10^5)$ 的数列，需要进行以下 $3$ 种操作：

1. 将区间内每个数乘上 $x$
2. 将区间内每个数加上 $x$。
3. 求数列的区间和，答案对一个大数取模。

**分析：**这是一个多标记**线段树**的问题。题目中一共出现了两种修改，分别为加法和乘法，考虑用两个标记分别维护它们。

设add表示某个区间的加法标记，mul表示某个区间的乘法标记。这里mul表示的是原区间和乘上的数，add表示的是原区间和加上的数，设原区间和为 $s$，则最终计算的区间和为 $s\times$ mul+add，否则难以维护。需要注意的是，对区间进行加操作时，对前面的区间乘操作没有影响，而对区间进行乘操作时，相当于将之前区间要加的数也乘上了 $x$ 也即：若设原区间和为 $s$，新加了 $a$，现在要乘上 $b$，则新的区间和是 $(s+a)\times b=s\times b+a\times b$。

因此，标记的下传顺序十分关键：在 `pushdown` 时，必须先下传乘法标记，再下传加法标记。因为乘法标记在下传时，需要让子结点的加法标记也乘上当前结点的乘法标记值，如果先下传加法标记，会让下传的那部分加法标记再乘上乘法标记，而事实上这部分标记已经在原结点乘过了，因此会出现计算错误。

```
#define maxn 1000010
#define ll long long
int p;
ll a[maxn], w[maxn * 4];
ll lzy_add[maxn * 4], lzy_mul[maxn * 4]; //
void pushup(const int u) {
    w[u] = (w[u * 2] + w[u * 2 + 1]) % p; // w[u]是区间和
}
// 为节省篇幅,本例中 build函数和例6-1的build函数基本一致,但需要加上lzy_mul[u]=1;
// InRange和OutofRange函数和例6-1的也是一致的,本处略
void maketag(int u, int L, int R, ll x, int type) {
    if (type == 1) {
        (lzy_add[u] *= x) %= p; // 修改当前结点的加法延迟标记
        (lzy_mul[u] *= x) %= p;
        (w[u] *= x) %= p;
    } else {
        (lzy_add[u] += x) %= p;
        (w[u] += (R - L + 1) * x) %= p;
    }
}
void pushdown(int u, int L, int R) {
    int M = (L + R) / 2;
    maketag(u * 2, L, M, lzy_mul[u], 1);
    maketag(u * 2, L, M, lzy_add[u], 2);
    maketag(u * 2 + 1, M + 1, R, lzy_mul[u], 1);
    maketag(u * 2 + 1, M + 1, R, lzy_add[u], 2);
    lzy_mul[u] = 1;
    lzy_add[u] = 0;
}
// query函数和例6-1的基本一致,只是最后返回的是两个子树信息的和,对p取余数
// update函数和例6-1的基本一致,但该函数和它调用的maketag()函数需要加一项参
// 数 type
// 表示是哪种操作,本处略
```

对于要求支持区间查询的线段树，其结点上所维护的信息必须具有可合并性。也就是说，从某个结点的两个子结点的信息通过汇总操作可以得出该结点的信息。但有时所求的信息如果直接维护可能并不具有可合并性，这时可能需要维护一些额外的信息，从而使得子结点信息可以合并推出父结点信息。

### 例 6-6 小白逛公园(洛谷P4513)

给定一个长度为 $n$ 的数列 $a$，有 $m$ 次操作，每次操作要么对 $a$ 进行单点修改，要么查询数列 $a$ 的最大子段和是多少。区间$[l,r]$的连续和是指 $\sum_{i=l}^{r} a_i = a_l + a_{l+1} + \cdots + a_r$。最大子段和指的是所有的区间连续和中最大的值。

**数据范围：** $1 \leqslant n \leqslant 5 \times 10^5, 1 \leqslant m \leqslant 10^5, -1000 \leqslant a_i \leqslant 1000$

**分析：**考虑对序列 $a$ 建立**线段树**。如果只在结点上维护信息“当前区间内的最大子段和”则无法汇总到父结点上，因为并不能通过子结点的最大子段和推出父结点的最大子段和。例如，对于序列-1，1，1，-1 和序列-1，1，-1，1，两者的区间[1，2]以及区间[3，4]的最大子段和均为 1，但是前者的区间[1，4]的最大子段和为 2，后者的为 1。

进一步地，考虑父结点的最大子段和只可能存在以下 3 种情况：是左子结点的最大子段和；是右子结点的最大子段和；是左子结点和右子结点的两段相邻的和拼起来。对于前两种情况容易转移，现在考虑第 3 种情况。

在这种情况下，左子结点被拼起来的那一段区间必须包含左子结点的右端点，换句话说，它是以左子结点右端点为起点向左找的最大连续和，称为左连续和；同理，右子结点被拼起来的那一段区间是以右子结点左端点为起点向右找的最大连续和，称为右连续和，如图 6-6(a) 所示。这样只需要维护结点的左右连续和以及最大子段和，就可以合并出父结点的最大子段和了。

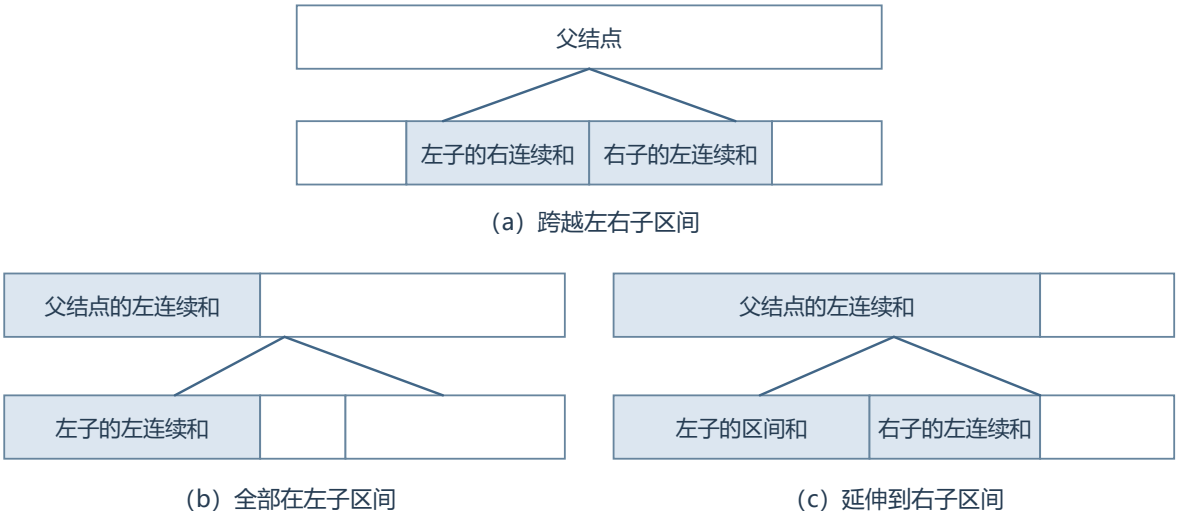



图6-6 维护区间的最大子段和

进一步考虑如何合并出父结点的左连续和以及右连续和：对于父结点的左连续和，要么直接就是左子结点的左连续和，要么是左子结点的全体拼上右子结点的左连续和，如图 6-6(b) 和 图 6-6(c) 所示；右连续和同理。

因此，只需要再维护一个区间和，就可以完成对左右连续和的维护了。区间和的维护在这里不再赘述。

```
struct Node {
    int sum, val, lv, rv; // sum 是区间和, val 是最大子段和
    // lv和rv分别是左右连续和
};
Node t[maxn << 2];
void pushup(int u) {
    int ls = u * 2, rs = u * 2 + 1;
    t[u].sum = t[ls].sum + t[rs].sum;
    t[u].val = std::max({t[ls].val, t[rs].val, t[ls].rv + t[rs].lv});
    t[u].lv = std::max(t[ls].lv, t[rs].lv + t[ls].sum);
    t[u].rv = std::max(t[rs].rv, t[ls].rv + t[rs].sum);
}
```

## 6.3 权值线段树（选读）

对一个数列 $a$ 构造一个数列 $b$，其中 $b_i$ 表示 $a$ 中数值 $i$ 出现的次数，也即满足 $a_j = i$ 的 $j$ 的个数，这样的数列 $b$ 称作数列 $a$ 的权值数列，对 $b$ 构造的**线段树**称为数列 $a$ 的权值线段树。

权值线段树在很多计数问题中有着重要的应用，因为它可以轻松地修改对 $a$ 维护的统计信息。容易发现，当将 $a_i$ 从 $x$ 修改为 $y$ 时，只需要对其权值线段树做两次单点修改，即给 $b_x$ 减去 1，给 $b_y$ 加上 1。

### 例 6-7 逆序对(洛谷P1908)

给定一个数列 $a$，求数列中满足 $i<j$ 且 $a_i>a_j$ 的二元组（$i$，$j$）（又称逆序对）有多少个。

数列长度 $n$ 不超过 $5\times10^5$，数列中的数不超过 $10^9$。

**分析：**首先不难发现只需要 $a$ 中的大小关系而不关心元素具体的值，因此可以通过**离散化**的方式将数列中数值的范围缩小至[1，$n$]。

考虑枚举 $i$，那么对于每个 $j$，只需要知道在它之前有多少个大于 $a_j$ 的位置即可。对 $a$ 的前（$i-1$）位建立权值**线段树**，每次只需要查询线段树上 $[a_j+1,n]$ 的区间和即可。考虑如何从 $a$ 的前（$j-1$）位的权值线段树扩展为前 $j$ 位的线段树：这个操作在权值数组上只是给 $b_{a_j}$ 加上 1，因此只需要在查询完 $a_j$ 对应的答案后，在线段树上令 $a_j$ 处加上 1 即可。代码如下。

```
#include <cstdio>
#include <algorithm>
using namespace std;
const int maxn = 500005;
int n;
int a[maxn], w[maxn * 4];
void Init_hash();
void build(int u, int l, int r);
int query(int u, int l, int r, int v);
void update(int u, int l, int r, int v);
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; ++i)
        scanf("%d", a + i);
    Init_hash();
    long long ans = 0;
    for (int j = 1; j <= n; ++j) {
        ans += query(1, 1, n, a[j] + 1); // 查询a[j]与它前面的位置构成的逆序对
        update(1, 1, n, a[j]);           // 将a[j]加入权值线段树
    }
    printf("%lld\n ", ans);
}
void Init_hash() { // 离散化过程
    static int tmp[maxn];
    for (int i = 1; i <= n; ++i)
        tmp[i] = a[i];
    sort(tmp + 1, tmp + 1 + n);
    int *ed = unique(tmp + 1, tmp + 1 + n);
    for (int i = 1; i <= n; ++i)
        a[i] = lower_bound(tmp + 1, ed, a[i]) - tmp;
}
```

```
int query(int u, int l, int r, int v) {
    if (w[u] == 0)
        return 0;
    if (l >= v)
        return w[u];
    int M = (l + r) / 2;
    if (M < v)
        return query(u * 2 + 1, M + 1, r, v); // 如果左子树右端点M 比v小,那么
    // 所求的一定是右子树的某个子区间(因为左子树区间[1,M]和查询区间[v,n]显然无交集)
    else
        return query(u * 2 + 1, M + 1, r, v) + query(u * 2, l, M, v);
}
void update(int u, int l, int r, int v) {
    ++w[u]; // 经过的区间一定包含v,因此直接自增即可
    if (l == r)
        return; // 到达叶结点
    int M = (l + r) / 2;
    if (M >= v)
        update(u * 2, l, M, v); // 如果左子树右端点M不小于v,则v一定在左
    // 子树中(因为处理过程中保证了1<=v)
    else
        update(u * 2 + 1, M + 1, r, v);
}
```

## 6.4 课后习题与实验

### 习题 1 忠诚数(洛谷P1816)

给定长度为 $n(1 \le n \le 10^5)$ 的序列 $a$ 有 $m(1 \leqslant m \leqslant 10^5)$ 次询问，每次求 $a_l, a_{l+1}, \cdots, a_r$ 的最小值。

### 习题 2 方差(洛谷P1471)

维护长度为 $n(1 \leqslant n \leqslant 10^5)$ 的数列 $a$，有 $m(1 \leqslant m \leqslant 10^5)$ 次操作，每次对 $a$ 进行区间加，或查询区间平均数，或查询区间方差。

### 习题 3 STEP(洛谷 P6492)

维护长度为 $n(1 \leqslant n \leqslant 2 \times 10^5)$ 的 $01$ 串 $a$，有 $m(1 \leqslant m \leqslant 2 \times 10^5)$ 次操作，每次对 $a$ 进行单点修改，并查询当前最长的 $0$ 和 $1$ 交替出现的子串的长度。

### 习题 4 三元上升子序列(洛谷P1637)

给定长度为 $n(1 \leqslant n \leqslant 3 \times 10^4)$ 的序列 $a$，求满足 $a_i < a_j < a_k$ 且 $i < j < k$ 的三元组 $(i, j, k)$ 的个数。

### 习题 5 色板游戏(洛谷P1558)

色板长度 $L$ 是一个正整数，所以可以均匀地将它划分成 $L$ 块 $1$ 厘米长的小方格，并从左到右标记为 $1, 2, \cdots, L$。

现在色板上只有一个颜色，老师告诉阿宝在色板上只能做以下两件事。

1. CABX：在 $A$ 到 $B$ 号方格中涂上颜色 $X$。
2. PAB：询问 $A$ 到 $B$ 号方格中有几种颜色。

学校的颜料盒中一共有 $T$ 种颜料，为简便起见，将其分别标记为 $1,2,\cdots,T$，开始时色板上原有的颜色就为 $1$ 号色。

**数据范围：** $1 \leqslant L \leqslant 10^5, 1 \leqslant T \leqslant 30_c$ 操作数不超过 $10^5$。

### 习题 6 煮酒(洛谷P5522)

给定 $m(1 \leqslant m \leqslant 10^5+7)$ 个长度为 $n(1 \leqslant n \leqslant 30)$ 的可能含有“？”的 $01$ 串，其中“？”既能代表 $0$，也能代表 $1$，进行 $q(1 \leqslant q \leqslant 10^6+7)$ 次操作，每次给定一个区间，求有多少 $01$ 串满足区间内的所有字符串都可以通过将“？”赋值为 $0$ 或 $1$ 后等于该 $01$ 串，或者单点修改某个字符串。

### 习题 7 游历(洛谷P4145)

给定一个长度为 $n$ 的数列，有 $m$ 次操作，分为两种：

1. 将给定区间$[l,r]$内的每个数开平方根后向下取整。
2. 求给定区间 $[l,r]$ 内每个数的和。

**数据范围：** $1 \leqslant n,m \leqslant 10^5$ 数列里的数是不超过 $10^{12}$ 的非负整数。

### 习题 8 序列操作(洛谷P2572)

给定一个 $01$ 序列，要求支持如下几种操作。

1. $0\ l\ r$：把$[l,r]$区间内的所有数全变成 $0$。
2. $11\ r$：把$[l,r]$区间内的所有数全变成 $1$。
3. $2\ l\ r$：把$[l,r]$区间内的所有数全部取反，即把所有的 $0$ 变成 $1$，将所有的 $1$ 变成 $0$。
4. $3\ l\ r$：询问$[l,r]$区间内总共有多少个 $1$。
5. $4\ l\ r$：询问$[l,r]$区间内最多有多少个连续的 $1$。

操作数和序列长度都不超过 $10^5$

### 习题 9 Points(洛谷CF19D)

在平面直角坐标系中定义如下 $3$ 种操作。

1. $\mathbf{add}(x,y)$：将点 $(x,y)$ 标记在坐标系上。
2. $\mathbf{find}(x,y)$：查询在 $(x,y)$ 右上方的点中横坐标最小的点；如果有多个，输出其中纵坐标最小的；如果没有，则输出 $-1$。
3. $\mathbf{remove}(x,y)$：将点 $(x,y)$ 取消标记。

**数据范围：**操作数不超过 $2 \times 10^5, 0 \leqslant x,y \leqslant 10^9$。

# 第7章 字符串

字符串就是由字符连接而成的序列。可以使用不同的算法来解决不同的和字符串有关的问题。《基础篇》中介绍过的哈希表可以将一个字符串映射为一个整数，可以高效地查询这个字符串是否存在于一个集合中。本章将介绍另外两种比较基础的字符串工具—KnuthMorrisPratt算法和**字典树**。

KnuthMorris Pratt算法简称KMP算法，可以高效地从一个字符串中查询另外一个指定的字符串是否存在；如果存在其位置在哪里。它会利用字符串匹配过程中失败的信息，从而减少字符串查找比较的次数。

字典树（Trietree）可以方便地储存、索引和查找字符串。当然除了字符串，还可以将数字拆成二进制并存入字典树中，常常可以高效处理涉及异或的问题。

本章思维导图如图 7-1 所示。




图7-1 本章思维导图

## 7.1 KMP算法

### 例 7-1 字符串的概念与性质

字符串是由字符连接而成的序列。每个字符都是一个特定字符集中的一个元素。一个字符串可以没有任何一个字符，这种字符串被称为空串。

例如，字符集 $\Sigma$ 可以是所有 $26$ 个小写字母组成的集合，字符 `a` 最靠前，字符 `z` 最靠后。字符串 $S=$ `abcabcd` 由 $7$ 个字符组成，长度记为 $|S|=7$。

将 $S$ 的第 $i$ 个字符记为 $S_i$。为与 C++ 数组的表示方式一致，下标一般从 $0$ 开始，因此 $S_0$ 就是字符 `a`。

一个字符串的子串指字符串中连续的一部分。例如，“$c$”“bca”和“abcabcd”都是字符串 $S$ 的子串。可以使用 $S_{i...j}$ 表示 $S$ 从第 $i$ 位到第 $j$ 的字符串，如 $S_{1...3}$ 就是“bca”。相应的，从字符串 $S$ 中按顺序依次挑选出字符组成的字符串，被称为子序列。例如，“$c$”“bca”和“aac”都是 $S$ 的子序列。

包括一个字符串最开头的字符的子串，被称为这个字符串的前缀。例如，“$a$”和“abcd”都是字符串 $S$ 的前缀。相应的，包括一个字符串最后一个字符的子串，被称为这个字符串的后缀。$S$ 的真前缀是指除了字符串 $S$ 本身的其他前缀，真后缀同理。

由于字符集本身是有序的，两个字符串之间也可以比较先后顺序，这种顺序被称为字典序：首先比较第 $1$ 个字符，如果不相同，对应位置字符靠前的字符串的字典序靠前，否则比较第 $2$ 个字符……依此类推。如果一个字符串 $S_1$ 是另外一个字符串 $S_2$ 的真前缀，则 $S_1$ 的字典序在 $S_2$ 之前。

将字符串反着写就是字符串的逆序。例如，字符串 $S$ 的逆序是“dcbacba”。如果一个字符串的逆序等于它本身，则这个字符串是回文字符串。

在 C++ 中，字符串可以使用字符数组存储，也可以使用STL的string存储。这两种存储方式各有特点，尤其需要注意各种操作的时空复杂度。相关内容在《基础篇》中已经介绍过，读者可以根据实际情况选择更加合适的方式。

### 例 7-2 KMP字符串匹配(洛谷P3375)

给出两个字符串 $s_1$ 和 $s_2$，若 $s_1$ 的区间$[l,r]$子串与 $s_2$ 完全相同，则称 $s_2$ 在 $s_1$ 中出现，其出现位置为 $l$。现在请求出 $s_2$ 在 $s_1$ 中所有出现的位置。

测试数据满足 $1 \leqslant |s_1|,|s_2| \leqslant 10^6$,$s_1$ 和 $s_2$ 中均只含大写英文字母。

此外，题目还要求对于s2 的每个前缀 $s$’的最长bordert′ 的长度（定义一个字符串 $s$ 的border $s_2$ 为 $s$ 的一个非 $s$ 本身的子串 $t$，满足 $t$ 既是 $s$ 的真前缀，又是 $s$ 的真后缀），不过读者可以暂时不必考虑这个问题。

**分析：**有些编程语言提供字符串查找函数，例如 C++STL中的string：：`find()` 函数，可以以较高效率地查询子串的位置①。为了避免选手偷懒直接使用各种查找函数，题目中加上了额外的任务。在字符串 $s_1$ 中查找字符串 $s_2$ 的问题，被称为字符串匹配。其中 ,$s_1$ 被称为待匹配串 ,$s_2$ 被称为模式串。一般来说，模式串的长度会远小于待匹配串的长度。

如果是枚举匹配，将 $s_2$ 的第一个字母依次和 $s_1$ 的每一个字母对齐，然后直接比对重合的部分是否一致。这种做法的时间复杂度是 $O(|s_1||s_2|)$。如果字符串比较随机，枚举匹配的效率是不低的，但可以构造测试数据，使枚举匹配的做法超时。枚举匹配效率低是因为比对失败时还需要从头开始匹配。在失配（匹配失败）之后，如果可以将模式串向后移动多位，而不是仅仅移动一位，这样可以提升效率。

例如，待匹配串是“abcacababcab”，模式串是“abcab”，首先将待匹配串和模式串放在一起进行匹配，发现被匹配串第 4 个字符“$c$”（为了和代码中的字符串标号统一，这里定义字符串开头是第 0 个字符）和模式串第 4 个字符“$b$”失配。这时将模式串向右移动 3 位，使移动后的第 0 位和移动前

的第 3 位对齐，然后继续判断匹配，如图 7-2(a) 所示。同理，这里提供了另外一个例子：当模式串第 5 位失配时，可以发现第 0 位到第 2 位的子串和第 2 位到第 4 位的子串是一样的，将模式串往后移动两位，然后继续尝试匹配，如图 7-2(b) 所示。

> **注：** 1. C++ 标准中并没有规定字符串查询子串函数的复杂度，各个编译器厂商可以有自己的实现方式。

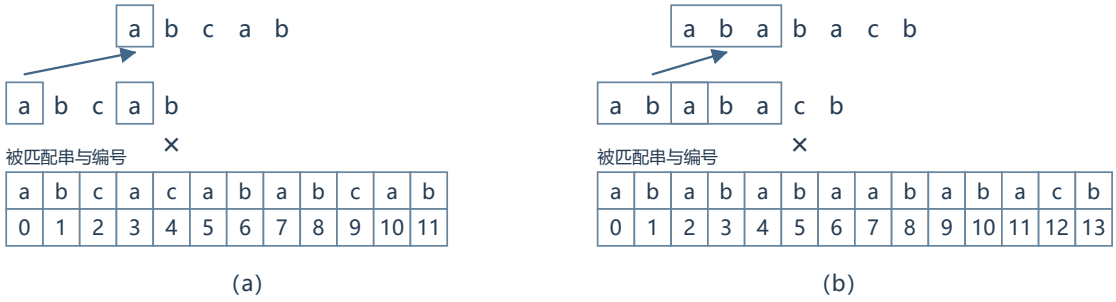



图7-2 匹配字符串

像这样，对于模式串的每一位，都存在唯一的“特定变化位数”：在这一位失配时，可以直接将模式串往右移动这一“特定变化位数”。聪明的读者可能已经发现，对于 图 7-2(a) 来说，模式串第 4 位失配时，考察第 0 位到第 3 位这个子串，它的前缀“$a$”和后缀“$a$”是相同的；对于 图 7-2(b) 来说，模式串第 5 位失配时，考察第 0 位到第 5 位这个子串，它的前缀“aba”和后缀“aba”是相同的。

像这样**前缀和**后缀相同的情况下，在失配前，模式串的后缀已经验证了可以和待匹配串匹配，那么这个前缀也是可以匹配的，不需要重复枚举匹配。这对相同的前缀和后缀，就是题面中提到的border。

如何获得模式串的每个前缀 $s$’的最长bordert’的长度呢？朴素枚举的做法是针对每个模式串的每一个前缀，依次枚举它的前缀和后缀长度，然后比较这两个子串是否相同（注意，判断字符串是否相同的时间复杂度是 $O(n)$）。在下文中为了方便讨论，将模式串记为 $s$ 这种做法的时间复杂度是 $O(|s|^3)$，因此需要进一步优化。

令模式串第 0 位到第 $i$ 位的子串的最长border的长度为 $b_i$，将其称为前缀函数。可以发现，在已经得到 $b_{i-1}$ 的情况下，可以递推出 $b_i$ 的结果。如图 7-3(a) 所示的案例，已知 $b_{i-1}=3$，计算 $b_i$ 的时候，如果发现 $s_3$ 等于 $s_i$: 则可以借助之前已经记录的信息，得到 $b_i=b_{i-1}+1$ 的结论。

读者可以利用反证法证明： $b_i$ 最多比 $b_i-1$ 多 1，不可能多更多。如果 $s_3 \neq s_i$，则继续尝试枚举更短的前后缀并判断是否相同，然后继续尝试匹配。考察枚举更短前后缀的循环，如果需要缩减一次前后缀长度，则必须有对应的增加前后缀的长度的过程，而增加的次数不会超过 $s$，所以减少长度的次数也不会超过 $|s|$ 因此，这种做法的时间复杂度是 $O(s2)$。

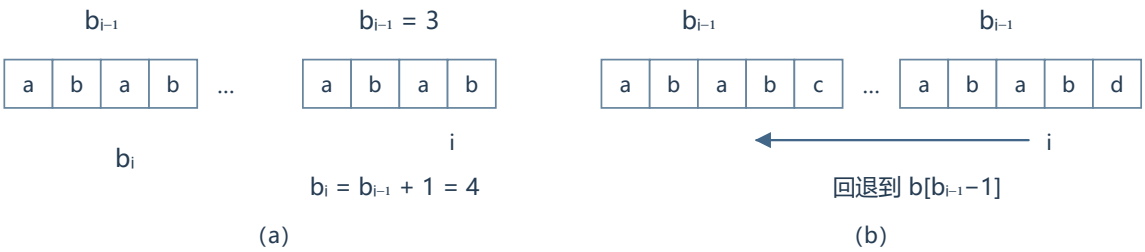



图7-3 计算数组b的值

实际上，找到更短border的过程也可以优化。如图 7-3(b) 所示， $b_{i-1}=4$，发现 $s_4 \neq s_i$，因此需要找 $s_{0\cdots i-1}$ 子串中次短的border。

由于 $s_{0\cdots 3}=s_{i-4\cdots i-1}$，所以计算 $s_{0\cdots i-1}$ 子串中次短的 border长度就是 $s_{0\cdots b_{i-1}-1}$ 的最长 border长度，即 $b_{b_{i-1}}$。

如果发现还是不能匹配，则可以继续迭代，继续缩短长度。这种做法不需要判断子串是否相同，时间复杂度是 $O(|s|)$。

于是，如何借助前缀函数进行字符串匹配的做法就显而易见了：首先将模式串和待匹配串的开头对齐，然后比较待匹配串和模式串的第 0 位是否一致，如果一致则继续匹配下一位；如果不匹配则将模式串往右移动，使它的左border和它原来的右border重合，这种算法就是KMP算法。代码实现中，可以将模式串的指针设为 $j$，将待匹配串的指针设为 $i$，如果失配，则将 $i$ 的值设置为 $b_{j-1}$；当匹配完所有的模式串字符，说明找到了一个匹配，根据 $i$ 的位置推导出匹配到的位置。代码如下：

```
#include <iostream>
#include <string>
#define MAXN 1000010
using namespace std;
int b[MAXN];
int len1, len2;
string s1, s2;
int main() {
    cin >> s1;
    cin >> s2;
    len1 = s1.size();
    len2 = s2.size();
    int j = 0;
    for (int i = 1; i < len2; i++) {    // 计算数组 b
        while (j > 0 && s2[i] != s2[j]) // 失配
            j = b[j - 1];
        if (s2[j] == s2[i])
            j++; // 匹配上了
        b[i] = j;
    }
    j = 0;
    for (int i = 0; i < len1; i++) {    // 进行匹配
        while (j > 0 && s1[i] != s2[j]) // 失配
            j = b[j - 1];
```

```
        if (s2[j] == s1[i])
            j++;
        if (j == len2) { // 模式串都已经匹配完毕
            cout << (i + 1) - len2 + 1 << endl;
            j = b[j - 1];
        }
    }
    for (int i = 0; i < len2; i++)
        cout << b[i] << " ";
    return 0;
}
```

计算前缀函数和进行匹配的两部分可以合并。令 $s_1$ 为被匹配串、$s_2$ 为模式串，构造 `S = s2 + "@" + s1`（确保分隔符 `@` 不在字符集中），然后计算整个字符串 $S$ 的前缀函数 $b$。从分隔符之后开始检查：若 $b_i = |s_2|$，就说明以位置 $i$ 结尾的后缀等于模式串 $s_2$，即找到了一次匹配。代码如下：

```
#include <iostream>
#include <string>
#define MAXN 2000010
// 注意,因为两个字符串连起来了,数组的大小要开够
using namespace std;
int b[MAXN];
int len1, len2, lenS;
string s1, s2, S;
int main() {
    cin >> s1 >> s2;
    S = s2 + "@" + s1;
    len1 = s1.size();
    len2 = s2.size();
    lenS = S.size();
    int j = 0;
    for (int i = 1; i < lenS; i++) {
        while (j > 0 && S[i] != S[j])
            j = b[j - 1];
        if (S[j] == S[i])
            j++;
        b[i] = j;
    }
    for (int i = len2 + 1; i <= len1 + len2; i++)
        if (b[i] == len2)
            cout << i - len2 - len2 + 1 << endl;
    for (int i = 0; i < len2; i++)
        cout << b[i] << " ";
    return 0;
}
```

### 例 7-3 无线传输(洛谷P4391.,BOI2019)

给定一个字符串 $s_1$，它是由某个字符串 $s_2$ 不断自我连接形成的（保证至少重复两次，这时 $s_1$ 被称为周期串）。但是字符串 $s_2$ 是不确定的，现在只想知道它的最短长度是多少。对于全部的测试点，保证 $1 < L \leqslant 10^6$。

**分析：**先考虑一下，如果 $s_1$ 是由 $s_2$ 拼接两次组成的。这种情况下， $s_1$ 的前一半和后一半相同，即 $s_2$ 就是 $s_1$ 的border，如图 7-4(a) 所示。如果发现 $s_1$ 最长 border的长度刚好就是 $\frac{|s_1|}{2}$，就说明 $s_1$ 是由两个相同的字符串拼接，其最长周期就是 $\frac{|s_1|}{2}$

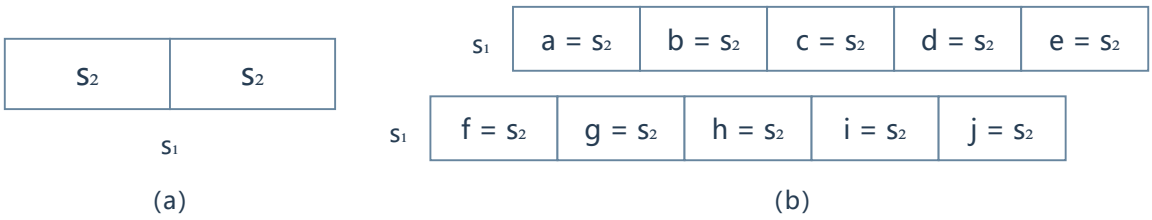



图7-4 周期串的性质

考虑 $s_1$ 是由更多的 $s_2$ 组成的情况（假设 $s_2$ 不是一个周期串）。如图 7-4(b) 所示 $s_2$ 拼接 5 次后变成了 $s_{10}$ 可以看出， $s_1$ 的前 4 段和后 4 段是相同的，因此 $s_1$ 的最长border 长度是 $4|s_2|$ 其周期长度 $|s_2|=|s_1|-4|s_2|=|s_1|-b_{|s_1|-1}$，也就是待匹配串的长度减去其最长border的长度。

反过来是否成立呢?如果有字符串 $s_1$，其前缀 $s_2$ 的长度 $|s_2|=|s_1|-b_{|s_1|-1}$ 且 $|s_1|$是 $|s_2|$的不小于 2 的倍数，是否能证明 $,s_1$ 是由 $\frac{|s_1|}{|s_2|}$ 个 $s_2$ 拼接而成的字符串呢？

这里说明一下证明的思路：如图 7-4(b) 所示，已知 $s_1$ 的 border，前 4 段 $(a+b+c+d)$ 和后 4 段 $(g+h+i+j)$ 是相同的，且每一小段（从 $a$ 到 $j$）的长度一致。此时，可以得到 $a=g$，然后又已知 $g=b$，又可以推得 $b=h\cdots$ 依此类推，最后可以得到 $a=b=c=d=e$ 的结论，即它是一个由长度为 $|s_2|=|s_1|-b_{|s_1|-1}$ 的字符串拼接 5 次的周期串。

读者可以思考当 $s_2$ 本身是周期串时，会出现什么情况。就本题而言，只需要求出 $|s_1|-b_{|s_1|-1}$ 的值就是答案，代码略。

KMP 算法还有很多其他应用。将KMP 算法和下文介绍的**字典树**相结合，就可以变为AC 自动机，可以同时对多个模式串进行匹配。此外，还有扩展KMP算法（$Z$ 函数），可以在线性的时间内计算待匹配串的所有后缀和模式串的最长公共前缀。这些算法可以用于解决更广泛的字符串问题，感兴趣的读者可以自行查阅相关的资料。

## 7.2 字典树

### 例 7-4 密码(洛谷P1481)

给出 $N(1\leqslant N\leqslant 2000)$ 和 $N$ 个单词，每个单词均由小写字母组成，长度不超过 $75$。如果一组单词按照字典序排序后，前一个单词是后一个单词的前缀，那么这一组单词被称为“词链”。请问，如果从这些单词中选一些单词，组成一个单词数量尽可能多的词链，这个词链最多可以包含多少个单词呢?

假设提供了 $7$ 个单词：$i$、ink、int、integer、intern、internet、oh，那么最长词链是 $i$-int-intern-in-ternet，一共是 $4$ 个单词。

**分析：**这里将使用**字典树**，又称为Trie树，来存储和维护这些单词。最开始时，根结点为空，如图 7-5(a) 所示。

接着插入单词 $i$，发现根结点没有字母 $i$ 子结点，于是就建立一个结点代表字母 i 单词的所有字母处理完了，这个结点标记存在这么一个单词，如图 7-5(b) 所示。

接着插入单词ink，发现根结点存在字母 $i$ 子结点，继续查找这个结点是否有字母 $n$ 子结点。若发现没有，就建立一个结点代表字母 n。同理，在刚创立的结点后再创建一个字母 $k$ 子结点。单词的所有字母处理完了，这个结点标记存在单词，如图 7-5(c) 所示。

接着插入单词int，发现根结点存在字母 $i$ 子结点，继续查找这个结点是否有字母 $n$ 子结点，然后在这个结点下继续查找是否有字母 $t$ 子结点。发现没有，就创建一个字母 $t$ 子结点，并标记存在单词，如图 7-5(d) 所示，注意根结点在最左边。

最后建立字典树如图 7-6 所示。

由于只有小写字母，每个结点可能都有 26 个子结点。枚举单词的每个字母，从根结点是依次查询各个字母，如果找不到对应的子结点，则需要建立新结点，并将新结点的索引编号放到上一级结点的子结点位置中。代码如下：

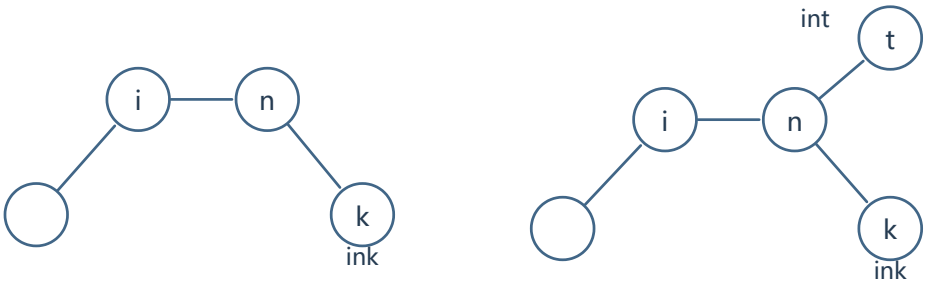



图7-5 字典树的建立步骤

$(d)$

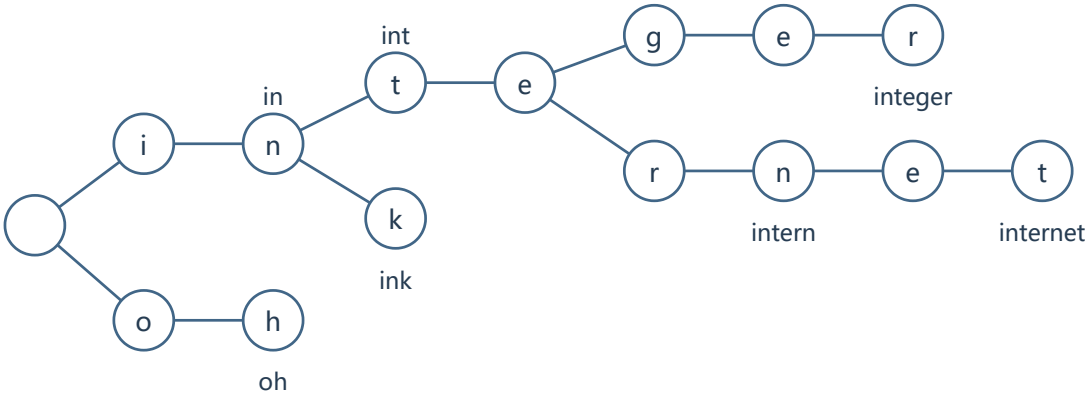



图7-6 建立好的字典树

```
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;
#define MAXN 2000 * 75 + 5
int trie[MAXN][26]; // 每个结点都可能有26个子结点
int word[MAXN];     // 用于记录以这个结点为单词末尾单词个数
char str[80];
int n, tot = 0, ans = 0;
void insert(char *str) {
    int u = 0; // 根结点
    int len = strlen(str);
    int res = 0; // 表示字典树路径上存有几个别的单词
    for (int i = 0; i < len; i++) {
        int a = str[i] - 'a'; // 映射小写字母
        if (trie[u][a] == 0)  // 如果没有这个结点
```

```
            trie[u][a] = ++tot;
        u = trie[u][a]; // 跳到该结点
        res += word[u]; // 如果这个结点存有单词则增加
    }
    word[u]++;          // 以该结点结尾的单词的个数加1
    if (res + 1 > ans) // 比较链长最大值
        ans = res + 1;
}
int main() {
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%s", str);
        insert(str);
    }
    printf("%d\n", ans);
    return 0;
}
```

字典树插入和查询的时间复杂度均为 $O(l)$，其中 $l$ 为当前字符串的长度；采用定长子结点数组时，空间复杂度为 $O(c\sum l)$，其中 $c$ 为字符集大小。

本题还有很多种其他解决方法，读者可以思考一下。

### 例 7-5 点名(洛谷P2580)

提供班级人数 $n(n\leqslant 10^4)$ 和所有人姓名的花名册。接下来进行 $m(m\leqslant 10^5)$ 次点名，并提供这些姓名。对于每次点名，输出一行结果。如果该名字正确且是第一次出现，输出 OK，如果该名字不存在，输出 WRONG，如果该名字正确但不是第一次出现，输出 $\mathbf{REPEAT_。}$ 对于所有输入的姓名字符串，只含小写字母，且长度不超过 $\mathbf{50}$，每行输入一个。

**分析**：使用**字典树**存下这些单词，然后点名时同时记录查询的次数输出即可。代码如下：

```
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;
#define MAXN 10000 * 50 + 5
int trie[MAXN][26]; // 每个结点都可能有 26个子结点
int word[MAXN], cnt[MAXN]; // 用于记录以这个结点为单词末尾单词个数和点名次数
char str[60];
int n, m, tot = 0;
void insert(char *str) {
    int u = 0; // 根结点
    int len = strlen(str);
    for (int i = 0; i < len; i++) {
        int a = str[i] - 'a'; // 映射小写字母
        if (trie[u][a] == 0)  // 如果没有这个结点
            trie[u][a] = ++tot;
        u = trie[u][a]; // 跳到该结点
    }
    word[u]++; // 以该结点结尾的单词的个数加1
}
int find(char *str) { // 查找单词是否存在,或者叫过几次
    int u = 0;        // 根结点
    int len = strlen(str);
```

```
    for (int i = 0; i < len; i++) {
        int a = str[i] - 'a';
        if (trie[u][a] == 0)
            return -1; // 没有这个单词
        u = trie[u][a];
    }
    return word[u] ? cnt[u]++ : -1;
    // 是否存在,存在的话增加次数
}
int main() {
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%s ", str);
        insert(str);
    }
    scanf("%d", &m);
    for (int i = 0; i < m; i++) {
        int res;
        scanf("%s", str);
        res = find(str);
        if (res == -1)
            printf("WRONG\n");
        else if (res == 0)
            printf("OK\n ");
        else
            printf("REPEAT\n ");
    }
    return 0;
}
```

需要注意的是，点名时查找字典树的结点，即使已经枚举完名字的所有字符，还是需要判断数组word中这个单词是否存在，否则遇到名字是花名册上名字的前缀的情况就出错了。

### 例 7-6 最长异或路径(洛谷P4551)

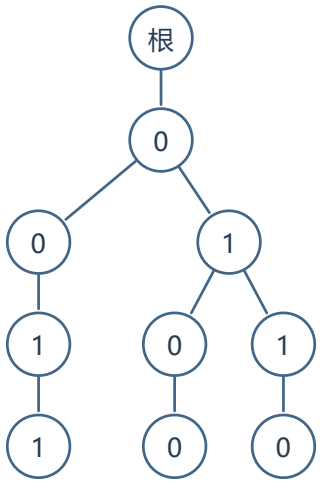
给定一棵有 $n$ 个结点的带权树，结点下标从 $1$ 开始到 $n$。寻找树中的两个结点，使得这两个结点间的异或路径最大。所谓异或路径，是指两个结点之间唯一路径上的所有边权的异或。

**数据范围：** $1 \leqslant n \leqslant 100000, 0 \leqslant w < 2^{31}$。

**分析：**题目中给出的树没有指定根结点，但可以指定结点 1 为根结点，使其变为有根树。建立这个树，求得第 $i$ 个结点到根结点的异或路径 $s_i$ （很容易通过递归求得）。而结点 $i$ 到结点 $j$ 之间的异或路径就是 $s_i \oplus s_j$，其中 $\oplus$ 为异或运算符。这是因为这两个结点分别到根结点的路径，在它们的**最近公共祖先**再往上的路径是重叠的，而将相同的边权异或两次就会归零。

因此对于所有 $s_i$，找到一个 $s_j$，使 $s_i \oplus s_j$ 的值最大并记录之，即可求解。

问题在于，如果两两枚举，效率很低，因此可以用 $0-1$**字典树**来维护所有数字的异或。其实原理和之前介绍的字符串字典树是一样的，将每个数字都变成 31 位的二进制数（和值域相关），将其视为一个字符串来建立字典树。例如，将数字 3、4、6 加入字典树中（假设是 4 位二进制数），可以得到如图 7-7 所示的 $0-1$ 字典树，其中所有叶子结点的深度都是一致的。




3 = $(0011)_2$　4 = $(0100)_2$　6 = $(0110)_2$

图7-7一个0-1字典树的例子

对于一个到根异或路径值 $s_i$，想找到对应的 $s_j$，可以用这个 $\mathbf{0-1}$ 字典树来查找。将 $s_i$ 化为 31 位的二进制数，然后从这个二进制数最左边开始，同时字典树从根结点开始，每一位使用贪心策略。如果字典树当前结点只有一个子结点，则直接往下走，否则选择和这一位不同的数对应结点（这样可以使它们的异或值为 1）。这个贪心策略是对的，因为从左边开

始的位数，尽可能构造答案位 1，可以使答案的值更大。当枚举完所有位数，也就找到了字典树的叶子结点，对应了符合要求的 $s_j$。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
#define MAXN 100010
struct edge {
    int to, w;
};
int s[MAXN], trie[MAXN * 31][2];
vector<edge> p[MAXN];
int n, tot;
void dfs(int x, int fa) {
    for (int i = 0; i < p[x].size(); i++) {
        int nxt = p[x][i].to;
        if (nxt != fa) {
            s[nxt] = s[x] ^ p[x][i].w; // 和父结点的异或路径再异或一次
            dfs(nxt, x);
        }
    }
}
void insert(int val) {
    int x = 0;
    for (int i = (1 << 30); i; i >>= 1) {
        int a = bool(val & i); // 这一位的值是0还是1
        if (!trie[x][a])       // 插入结点
            trie[x][a] = ++tot;
```

```
        x = trie[x][a];
    }
}
int find(int val) {
    int ans = 0, x = 0;
    for (int i = (1 << 30); i; i >>= 1) {
        int a = bool(val & i);
        if (trie[x][!a]) { // 优先取使答案变为1的方向
            ans += i;
            x = trie[x][!a];
        } else
            x = trie[x][a];
    }
    return ans;
}
int main() {
    cin >> n;
    for (int i = 1; i < n; ++i) {
        int u, v, w;
        cin >> u >> v >> w;
        p[u].push_back((edge){v, w});
        p[v].push_back((edge){u, w});
    }
    dfs(1, -1); // 根据读入建立树,计算每个结点到根的异或路径
    for (int i = 1; i <= n; ++i)
        insert(s[i]);
    int ans = 0;
    for (int i = 1; i <= n; ++i) // 对每一个值s[i],都找到对应的另一个数字
        ans = max(ans, find(s[i]));
    cout << ans;
}
```

注意，字典树需要处理 $n$ 个数，但有 31 位，所以保险起见需要开设 31n大小的数组存储。该算法的时间复杂度为 $O(n\log A)$，其中 $A$ 为值域上限。

### 例 7-7 异或综合(洛谷P5283,联合省队选拔2019)

给定长度为 $n$ 的序列 $a_i$。一个区间 $[l,r](1\leqslant l\leqslant r\leqslant n)$ 的价值为从 $a_l$ 到 $a_r$ 之间的每个数字进行的异或值。求这个序列中价值最大的 $k$ 个区间的价值和。

**数据范围：** $1\leqslant n\leqslant 10^5, 1\leqslant k\leqslant 2\times 10^5, 0\leqslant a_i\leqslant 2^{32}-1.$

**分析：**和上一题类似，设 $s_i$ 为序列中 $a_1$ 到 $a_i$ 的每个数字依次异或后的结果，则区间 $[l,r]$ 的价值就是 $s_r\oplus s_{l-1}$，因为区间 $[1,l-1]$的每一个数字都异或了两遍而抵消。

所以对于每一个 $s_i$ 都可以找到一个 $s_j$，使得 $s_i\oplus s_j$ 的值最大。如果能保证 $i<j$，则固定 $i+1$ 为左端点，就可以找到以 $j$ 为右端点的区间 $[i+1,j]$，使其价值最大。不过没有关系，因为 $s_i$ 和 $s_j$ 互为对偶，所以只需要找 $2k$ 个价值最大的 $(i,j)$ 配对，将其价值累加除以 2 就是答案。

如何找到 $2k$ 个最大的价值 $(i,j)$ 配对呢？对于每一个 $i$，找到对应的 $j$，并将对应的价值放入**优先队列**中（价值大的优先）。每次从优先队列中取出价值最大的元素，累加进答案。如果取出的是和 $s_i$ 对应的第 $t$ 大值（$i$ 已经取出了 $t$ 次），则将和 $s_i$ 对应的第 $t+1$ 大值放入优先队列，重复操作 $2k$ 次即可。

最后一个问题：给定 $i$ 和 $t$，如何求 $s_i$ 对应的 $s_j$，使 $s_i \oplus s_j$ 为第 $t$ 大？可以在**字典树**的插入过程中，记录每个结点的数字数量。在查询的过程中，根据优先往“0”方向走还是往“1”方向走，以及左右子结点数量和 $t$ 来决定下一步的方向，具体实现详见代码：

```
#include <bits/stdc++.h>
#define ll long long
using namespace std;
const int maxn = 5e5 + 100;
int n, k;
ll a[maxn];
int trie[maxn * 32][2], sz[maxn * 32], tot = 0;
void insert(ll val) {
    int u = 0;
    for (int i = 31; i >= 0; i--) {
        int z = (val >> i) & 1;
        if (!trie[u][z])
            trie[u][z] = ++tot;
        u = trie[u][z];
        sz[u]++;
    }
}
ll find(ll val, int t) {
    ll s = 0;
    int u = 0;
    for (int i = 31, z; i >= 0; i--) {
        z = (val >> i) & 1;
        if (sz[trie[u][z ^ 1]] >= t) {
            u = trie[u][z ^ 1];
            s |= (1LL << i);
        } else {
            t -= sz[trie[u][z ^ 1]];
            u = trie[u][z];
        }
    }
    return s;
}
struct node {
    ll w;
    int x, y; // x的第y大
    bool operator<(const node &A) const { return w < A.w; }
};
node d[maxn];
priority_queue<node> q;
int main() {
    scanf("%d%d", &n, &k);
    for (int i = 1; i <= n; i++) {
        scanf("%lld", &a[i]);
        a[i] ^= a[i - 1];
    }
    for (int i = 0; i <= n; i++)
        insert(a[i]);
    k = 2 * k;
```

```
    for (int i = 0; i <= n; i++) {
        d[i].w = find(a[i], 1);
        d[i].x = i;
        d[i].y = 1;
        q.push(d[i]);
    }
    ll ans = 0;
    node A;
    for (int i = 1; i <= k; i++) {
        A = q.top();
        ans += A.w;
        q.pop();
        d[A.x].y++;
        d[A.x].w = find(a[A.x], d[A.x].y);
        q.push(d[A.x]);
    }
    ans /= 2;
    printf("%lld", ans);
    return 0;
}
```

因为在字典树的结点中可以记录所管辖的数字数量，而且对于一个非叶子结点来说，往“0”方向走可以找到更小的数字，往“1”方向走可以找到更大的数字，所以 $0-1$ 字典可以完成一些平衡树的操作—插入、删除、根据数据查询排名和根据排名查询数据等。单次操作的复杂度和值域有关。

## 7.3 课后习题与实验

### 习题 1 最长前缀(洛谷P1470,USACO Training)

如果一个集合 $P$ 中的单词可以串起来（单词是一个仅用大写字母组成的字符串，可以重复使用）组成一个序列 $s$，那么可以认为序列 $s$ 可以分解为 $P$ 中的单词。单词不一定要全部出现（如以下案例中“BBC”就没有出现）。例如，序列“ABABACABAAB”可以分解为下面集合中的单词：

$\{\mathrm{A},\mathrm{AB},\mathrm{BA},\mathrm{CA},\mathrm{BBC}\}$

序列 $s$ 的前面 $k$ 个字符称作 $s$ 中长度为 $k$ 的前缀。设计一个程序，输入一个单词集合以及一个大写字母序列 $s$，设 $s'$ 是序列 $s$ 的最长前缀，使其可以分解为给出的集合 $P$ 中的单词，求 $s'$ 的长度 $k$。

**数据范围：** $P$ 集合中的单词数量不超过 $200$，每个单词长度不超过 $10$，$|s| \leqslant 2\times 10^5$ $O$

### 习题 2 测试(洛谷CF25E)

给定 $3$ 个长度不超过 $100000$ 的字符串 $s_1,s_2,s_{30}$ 找出一个字符串，使 $s_1,s_2,s_3$ 都是这个字符串的子串，并使这个字符串最短。输出最短字符串的长度即可。

### 习题 3 单词的周期(洛谷P3435,POI2006)

规定字符串 $Q$（可以是空串）表示为 $a$ 的周期，当且仅当 $Q$ 是 $a$ 的真前缀且 $a$ 是 $Q+Q$ 的前缀。例如，“ab”是“abab”的一个周期，因为“ab”是“abab”的真前缀，且“abab”是“ab”+“ab”的前缀。求给定字符串 $S(\epsilon|S| \leqslant 10^6)$ 所有前缀的最大周期长度之和。

### 习题 4 动物园(洛谷P2375,NOI2014)

给出一个字符串 $S$，求 $\prod_{i=1}^{n}(\mathrm{num}_i+1) \bmod (10^9+7)$。其中对于字符串 $S$ 的前 $i$ 个字符构成的子串，既是它的后缀同时又是它的前缀，并且该后缀与该前缀不重叠，将这种字符串的数量记作 $\mathrm{num}_i$。

每组测试数据提供不超过 $5$ 个字符串，每个字符串长度不超过 $10^6$。对于每个给出的字符串，都要给出答案。

### 习题 5 会议座位(洛谷P5149)

校长在校园网上公布了一份座位表， $n(n\leqslant 10^5)$ 位老师从左到右依次排成一行。然而到了开会时，校长不小心把座位表打乱了。如果有一对老师 $a$ 和 $b$，他们原来的座位是 $a$ 在 $b$ 左边，现在变成了 $a$ 在 $b$ 右边，那么这对老师便会贡献一单位交换值。请问：这些老师的总交换值是多少?

原定座位表和新的实际座位表分别给出 $n$ 位老师的名字（长度不超过 $5$ 的、包含大小写字母的字符串。名字对大小写敏感），名字之间用空格隔开。

### 习题 6 秘密信道(洛谷P2922,USACO2008年12月赛)

奶牛们互相发送秘密信息，信息是二进制的，共有 $M(1\leqslant M\leqslant 50000)$ 条。农夫已经部分拦截了这些信息，知道了第 $i$ 条二进制信息的前 $b_i(l\leqslant b_i\leqslant 10000)$ 位。他同时知道，奶牛使用 $N$（$1\leqslant N\leqslant 50000$） 条暗号。

但是，他仅仅知道第 $j$ 条暗号的前 $c_j(1\leqslant c_j\leqslant 10000)$ 位。

对于每条暗号 $j$，他想知道有多少截得的信息能够和它匹配。也就是说，有多少信息和这条暗号有着相同的前缀。当然，这个前缀长度必须等于暗号和那条信息长度的较小者。

### 习题 7 阅读理解(洛谷P3879,天津市省队选拔2010)

给出 $N$ 篇短文，每篇短文是仅由空格和英文小写字母组成的字符串。然后进行 $M$ 次询问，每次给出一个仅由小写字母组成的单词，求出这个单词在哪些短文中出现过。

**数据范围：** $1\leqslant M\leqslant 10^4, 1\leqslant N\leqslant 10^3$，每篇短文长度（含相邻单词之间的空格） $\leqslant 5\times 10^3$ 字符，每个单词长度≤$20$ 字符。

### 习题 8 最大异或和(洛谷P4735)

给定一个非负整数序列 $a$，初始长度为 $n$ 有 $m$ 个操作，有以下两种操作类型。

1. Ax：添加操作，表示在序列末尾添加一个数 $x$，序列的长度 $n+1$
2. $Q\ l\ r\ x$ 询问操作，需要找到一个位置 $p$，满足 $l\leq p\leq r$，使得 $a_p\oplus a_{p+1}\oplus\cdots\oplus a_N\oplus x$ 最大，并输出最大值。

### 习题 9 异或(洛谷P4592,天津市省队选拔2018)

现在有一棵以 $1$ 为根结点的由 $n$ 个结点组成的树，结点从 $1$ 至 $n$ 编号，树上每个结点上都有一个权值 $v_i$。现在有 $q$ 次操作，操作如下。

1. $1\ x\ z$ 查询结点 $x$ 的子树中的结点权值与 $z$ 异或结果的最大值。
2. `2 x y z`：查询结点 $x$ 到结点 $y$ 的简单路径上的结点的权值与 $z$ 异或结果的最大值。

**数据范围：** $1 < n, q \leqslant 10^5, 1 \leqslant x, y \leqslant n, 1 \leqslant v_i, z < 2^{30}$，

### 习题 10 普通平衡树(洛谷P3369)

需要写一种数据结构来维护一个集合，初始时集合是空的。其中需要提供以下操作：

1. 插入 $x$ 这个数字。
2. 删除 $x$ 这个数字（若有多个相同的数，应只删除一个）。
3. 查询 $x$ 这个数字的排名（排名定义为比当前数小的数的个数加 $1$）。
4. 查询排名为 $x$ 的数。
5. 求 $x$ 的前驱（前驱定义为小于 $x$，且最大的数）。
6. 求 $x$ 的后继（后继定义为大于 $x$，且最小的数）。

**数据范围：** $1 \leqslant n \leqslant 10^s, |x| \leqslant 10^7$。

**提示：**使用 01-**字典树**来维护这个集合。

### 习题 11 可乐(洛谷P6824,洛谷月赛,By pigstd)

现在有 $n$ 箱可乐，第 $i$ 箱可乐上标着一个正整数 $a_i$。 有一个非负整数 $x$，对于第 $i$ 箱可乐，如果 $(a_i \oplus x) \leq k$，那么就能喝到这箱可乐。现在已知 $k$ 与序列 $a_1, a_2, \cdots, a_n$ 可以决定 $x$ 的值，使得能够喝到的可乐的箱数最大。求出这个最大值。

**数据范围：** $1 \leqslant n, k, a_i \leqslant 10^6$。

# 第8章 线段树的进阶用法

本章主要学习**可持久化线段树**与**树状数组**套权值线段树的一些应用。对于一棵线段树而言，如果它的总长度不变，那么它的形态是不会改变的。也就是说，在值域不变的情况下，权值树的形态是不会改变的。这样一来，就可以对权值树进行合并操作。对于权值树 $A$ 和 $B$，若两者形态相同，则可以合并这两棵权值树，合并的方式就是对应结点相加，如图 8-1 所示。

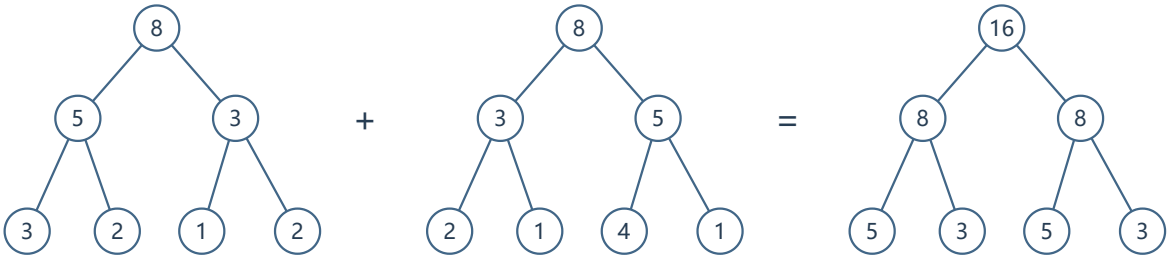



图8-1 线段树的合并

显然，合并后的树依然是一棵权值线段树。可以将权值线段树的合并类比为加法。类似的，也可以类比得到权值线段树的减法，即对应点数相减。这个性质非常重要，接下的内容就要基于这一性质。本章思维导图如图 8-2 所示。本章内容较难，读者可以将本章的阅读优先级调后。




图8-2 本章思维导图

## 8.1 可持久化线段树

### 例 8-1 可持久化线段树(洛谷P3834)

给定 $n$ 个整数构成的序列 $\{a_i\}$，共有 $m$ 次询问，对于指定的闭区间$[l,r]$，查询其区间内第 $k$ 小的值。

**数据范围：** $1 \leqslant n,m \leqslant 2\times 10^5, |a_i| \leqslant 10^9, 1 \leqslant l \leqslant r \leqslant n, 1 \leqslant k \leqslant r-l+1$。

**分析：**本题是一个典型的静态问题。可以直接使用权值树的可合并性解决。在解决问题之前先用**离散化**将值域缩减到 $O(n)$ 级别。

对于每个点，建立一棵权值树，第 $k$ 棵树维护[1，$k$]区间各个数值的出现次数。如果用加号代表合并，那么如图 8-1 所示就是一个权值树合并的例子。

事实上，由于有了权值树的加法，因此可以将第 $k$ 棵权值树看作维护前 $k$ 个点单点的权值树的**前缀和**。

首先，真正需要的树为维护$[l,r]$区间的树（也就是要知道$[l,r]$区间内各数的出现次数）。同时，维护[1r]区间的树可以由维护[1，$l-1$]区间的树和维护$[l,r]$区间的树合并而成。因此只要对这两棵树相减，就可以得到需要的树。事实上，这可以类比为一个差分操作。

所以可以同时从根结点开始在两棵树上进行查找。不妨设两棵树的左子树元素个数之差为 `tmp`，如果 $tmp \geqslant k$，则代表小于或等于mid的元素个数超过 $k$，于是只要去左子树寻找第 $k$ 小即可。若不然，则代表小于或等于mid的数量不足 $k$ 个，此时需要进入右子树寻找。但是由于左子树已经有 `tmp` 棵比目标更小的树，因此进入右子树寻找第 $k-$ `tmp` 小即可。代码如下：

```
int query(int o1, int o2, int l, int r, int q) {
    if (l == r) {
        return l;
    }
    int mid = l + r >> 1, tmp = t[t[o2].ls].v - t[t[o1].ls].v;
    if (tmp >= q)
        return query(t[o1].ls, t[o2].ls, l, mid, q);
    else
        return query(t[o1].rs, t[o2].rs, mid + 1, r, q - tmp);
}
```

但是，这时一个新的问题出现了。每一个权值树的空间都是 $O(n)$ 级别的，那么总复杂度就是 $O(n^2)$，显然这是无法接受的。此时就需要用到一个新技巧——动态开点。所谓动态开点，就是动态地分配内存。由于修改一个权值时，只会对它和它的直属父辈结点产生影响（也就是一条链），因此这个时候只需要对这一条链进行分配内存，而剩下的结点可以借用原结点，如图 8-3 所示。

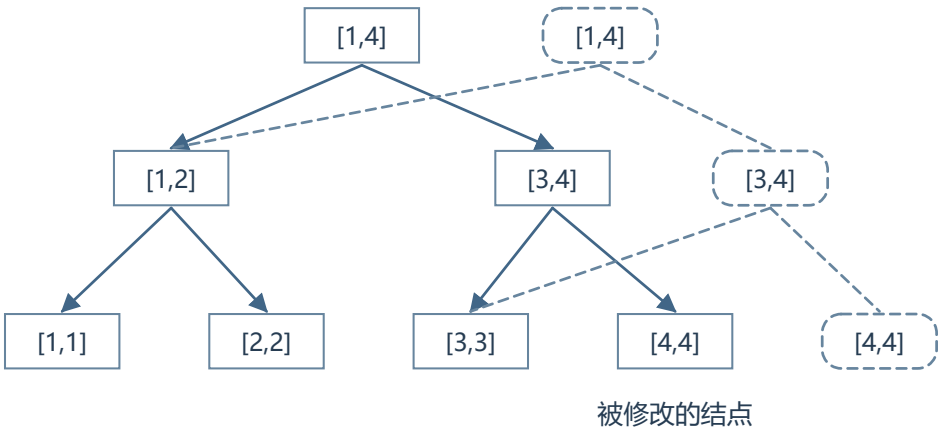



图8-3 可持久化线段树

在上述例子中需要修改 4 这一位置的值。先新建一个根结点，代表修改后的**线段树**的根结点。由于被修改的部分全部位于右子树，因此修改完后左子树和修改之前没有区别。可以将左子树的指针指向修改之前的左子树，从而省下左子树的空间。接着，新建一个右子树结点并重复之前的操作，直到修改到叶子结点为止。采用动态开点之后的修改代码如下：

```
void pushup(int o) { t[o].v = t[t[o].ls].v + t[t[o].rs].v; }
void change(int lsto, int &o, int l, int r, int q, int v) {
    if (!o)
        o = ++cnt;
    if (l == r) {
        t[o].v += v;
        return;
    }
    int mid = l + r >> 1;
    if (q <= mid) {
        t[o].rs = t[lsto].rs;
        t[o].ls = ++cnt;
        t[t[o].ls] = t[t[lsto].ls];
        change(t[lsto].ls, t[o].ls, l, mid, q, v);
    } else {
        t[o].ls = t[lsto].ls;
        t[o].rs = ++cnt;
        t[t[o].rs] = t[t[lsto].rs];
        change(t[lsto].rs, t[o].rs, mid + 1, r, q, v);
    }
    pushup(o);
}
```

这样做之后，空间就会被降到 $O(n \log n)$（$n$ 为值域）级别，可以接受。完整代码如下

**代码片段（需结合上文的声明和函数）：**

```
#include <bits/stdc++.h>
#define MAXN 100005
using namespace std;
struct seg {
    int v, ls, rs;
} t[MAXN << 8];
int rt[MAXN << 2], cnt;
// 省略 query、pushup、change函数的实现
int lsh[MAXN << 2], tot, n, m, node[MAXN << 2];
int main() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++) {
        scanf("%d", &node[i]);
        lsh[i] = node[i];
    }
    sort(lsh + 1, lsh + n + 1);
    tot = unique(lsh + 1, lsh + 1 + n) - lsh - 1;
    for (int i = 1; i <= n; i++) {
        node[i] = lower_bound(lsh + 1, lsh + tot + 1, node[i]) - lsh;
        change(rt[i - 1], rt[i], 1, tot, node[i], 1);
    }
    for (int i = 1; i <= m; i++) {
        int l, r, q;
        scanf("%d%d%d ", &l, &r, &q);
```

```
            printf("%d\n ", lsh[query(rt[l - 1], rt[r], 1, tot, q)]);
        }
        return 0;
    }
```

容易得出，无论修改还是查询操作，单次的时间复杂度都是 $O(\log n)$，总复杂度为 $O(\log n)$。这种数据结构被称为**可持久化**线段树。

### 例 8-2 神秘数(洛谷P4587,福建省队选拔2016)

一个可重复数字集合 $S$ 的神秘数定义为最小的不能被 $S$ 的子集的和表示的正整数。现给定长度为 $n$ 的正整数序列 $a$，进行 $m$ 次询问，每次询问包含两个参数 $l$ 和 $r$，需要求出由 $a_l a_{1+1}, \cdots, a_r$ 所组成的可重集合的神秘数。

**数据范围：** $1 \leqslant n, m \leqslant 10^5, \Sigma a \leqslant 10^9$。

**分析：**考虑对于一个已经确定的可重集合，如何确定其答案。

不妨假设该可重集合的元素从小到大分别为 $a_1, a_2, \cdots, a_s$，并将元素按顺序加入一个集合 $S_\sigma$ 记录一个变量 `ans` 代表目前不能被 $S$ 中若干元素求和表示的最小数，当 $S$ 为空时 $ans = 1$。此时，如果向 $S$ 中加入一个元素 $a_i$，如果 $a_i \in [1, ans]$，那么 `ans` 会变为 $ans + a_i$，否则 `ans` 不会有变化。

对于上述结论可以这样证明：如果 $a_i > ans$，则显然数 `ans` 依然无法被表示；否则，对于 $[1, a_i-1]$ 的元素按照原样表示，对于 $[a_i, ans + a_i - 1]$ （其中 `ans` 为修改之前的 `ans`）的数，则可以拆分为 $a_i + x$，其中 $x \in [1, ans-1]$，可以被之前的元素求和表示。

于是，可以考虑设计如下算法：设上一次加入元素之前的 `ans` 为lst，则查询lst+1 到 `ans` 之间的所有元素。如果存在元素，则代表表示范围可以被扩大，只需要将这些元素加入 $S$ 中，并更新lst和 `ans` 值；如果不存在，则代表表示范围已经最大，此时的 `ans` 即为最大范围。

之所以需要记录一个lst变量，是为了防止同一个数被重复加入集合中。每一次加入集合中的元素是在之前暂时无法使用，但是现在已经可以使用的所有元素，故最终不会重复计算任何一个元素，也不会遗漏任何一个元素，其正确性可以保证。

由于每次加入数据时，加入的数必然大于上一次的答案，因此如果可以加入新数，加入完成之后的答案至少大于原答案的 2 倍。这也就意味着，对于任意一个元素 $a_i$，如果最终可以被加入集合，则至多只需要 $O(\log a_i)$ 次就会被加入，故单次查询操作的次数为 $O(\log(\max\{a_i\}))$。

再回到原题。目前已经知道对于给定可重集合查询的做法，现在来考虑如何对原题进行实现。由于需要 $[l, r]$ 区间的静态信息，因此可以考虑采用**可持久化线段树**。由于每次查询只需要查询 $[l, r]$ 区间所有大小在 $[lst+1, ans]$ 之间的数的和，因此记录每个区间某个值之和（值乘以出现次数）。

查询时就可以方便地找出所需要的数的和，并使得每次查找的复杂度为 $O(\log(\max\{a_i\}))$，最终复杂度为 $O(n\log^2(\max\{a_i\}))$，可以通过本题。

代码如下：

```
#include <bits/stdc++.h>
#define MAXN 100005
using namespace std;
const int inf = 1e9;
struct seg {
    int v, ls, rs;
} t[MAXN << 8];
int rt[MAXN << 2], cnt, n, m;
void pushup(int o) { t[o].v = t[t[o].ls].v + t[t[o].rs].v; }
void change(int lsto, int &o, int l, int r, int q, int v) {
    if (!o)
        o = ++cnt;
    if (l == r) {
        t[o].v += v;
        return;
    }
    int mid = l + r >> 1;
    if (q <= mid) {
        t[o].rs = t[lsto].rs;
        t[o].ls = ++cnt;
        t[t[o].ls] = t[t[lsto].ls];
        change(t[lsto].ls, t[o].ls, l, mid, q, v);
    } else {
        t[o].ls = t[lsto].ls;
        t[o].rs = ++cnt;
        t[t[o].rs] = t[t[lsto].rs];
        change(t[lsto].rs, t[o].rs, mid + 1, r, q, v);
    }
    pushup(o);
}
int query(int o1, int o2, int l, int r, int ql, int qr) {
    if (ql <= l && r <= qr) {
        return t[o2].v - t[o1].v;
    }
    int mid = l + r >> 1, res = 0;
    if (ql <= mid)
        res += query(t[o1].ls, t[o2].ls, l, mid, ql, qr);
    if (qr > mid)
        res += query(t[o1].rs, t[o2].rs, mid + 1, r, ql, qr);
    return res;
}
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        int tmp;
        scanf("%d", &tmp);
        change(rt[i - 1], rt[i], 1, inf, tmp, tmp);
    }
    scanf("%d ", &m);
    while (m--) {
        int ans = 1, l, r, lst = 0;
        scanf("%d%d", &l, &r);
        while (1) {
            int res = query(rt[l - 1], rt[r], 1, inf, lst + 1, ans);
            lst = ans;
            if (res)
```

```
                ans = res + ans;
            else
                break;
        }
        printf("%d\n", ans);
    }
    return 0;
}
```

## 8.2 树状数组套权值线段树

### 例 8-3 平衡树(树套树)(洛谷P3380)

维护一个长度为 $n$ 的有序序列，其中需要提供以下操作：

1. 查询 $k$ 在区间内的排名。
2. 查询区间内排名为 $k$ 的值。
3. 修改某一位值上的数值。
4. 查询 $k$ 在区间内的前驱（前驱定义为严格小于 $x$，且最大的数，若不存在则输出 $-2147483647$）。
5. 查询 $k$ 在区间内的后继（后继定义为严格大于 $x$，且最小的数，若不存在则输出 $2147483647$）。

一共有 $m$ 个操作，对于操作 $1$、$2$、$4$ 和 $5$，各输出一行，表示查询结果。

**数据范围：** $1 \leqslant n, m \leqslant 5 \times 10^4$，序列中的值在任何时刻 $\in [0, 10^8]$。

**分析：**考虑到直接维护动态序列较难，因此可以采用一些数据结构进行维护。

如果本题去掉修改操作，则显然可以使用**可持久化**权值**线段树**进行维护。又由于可持久化权值线段树可以类比为权值树的**前缀和**，因此可以考虑采用类似于维护动态前缀和的方式来维护本题。而对动态前缀和的维护，最简单的方法即为**树状数组**（或线段树）。

在这里，可以考虑采用嵌套的形式，将权值树作为整体，用树状数组或是线段树对其进行维护。由于权值树同时满足可合并性、可减性和交换律，因此这样的维护是可行的。

以树状数组为例。对序列的每一个位置开一个权值树，然后使用树状数组进行维护。每次操作先在树状数组上处理出这次操作涉及哪几棵树，然后直接进行权值树的查询即可。

对于单点修改，先要在树状数组上处理出需要修改哪几棵树。这步操作与树状数组的基本操作相同，只是将树状数组的加减变成了线段树的修改，代码如下：

```
int lb(int x) { return x & (-x); }
// …
void add(int o, int v) {
    for (int i = o; i <= n; i += lb(i))
        change(rt[i], 1, len, a[o], v);
}
```

紧接着就是权值树的修改操作，直接放上去即可，代码如下：

```
void pushup(int o) { t[o].v = t[t[o].ls].v + t[t[o].rs].v; }
void change(int &o, int l, int r, int k, int v) {
    if (!o)
        o = ++tot;
    if (l == r) {
        t[o].v += v;
        return;
    }
    int mid = l + r >> 1;
    if (k <= mid)
        change(t[o].ls, l, mid, k, v);
    else
        change(t[o].rs, mid + 1, r, k, v);
    pushup(o);
}
```

查询第 $k$ 小时，同样先预处理涉及的树。但与上面不同的是，需要将它们记录下来：

```
int find_num(int l, int r, int k) {
    cnt = num = 0;
    for (int i = r; i; i -= lb(i)) {
        tem[++cnt] = rt[i]; // 存储前r棵树的合并在树状数组上的表示
    }
    for (int i = l - 1; i; i -= lb(i)) {
        tmp[++num] = rt[i]; // 存储前1-1 棵树的合并在树状数组上的表示
    }
    return query_num(1, len, k);
}
```

然后进行查询，与权值树查询相同。先统计左子树的元素个数，判断查询的目标在左子树还是右子树，然后递归寻找。在递归寻找时，需要将记录的所有结点全部移向它的左/右子树。代码如下：

```
int query_num(int l, int r, int k) {
    if (l == r) {
        return l;
    }
    int mid = l + r >> 1, sum = 0;
    for (int i = 1; i <= cnt; i++)
        sum += t[t[tem[i]].ls].v;
    for (int i = 1; i <= num; i++)
        sum -= t[t[tmp[i]].ls].v;
    // 统计左子树的个数
    if (k <= sum) {
        for (int i = 1; i <= cnt; i++)
            tem[i] = t[tem[i]].ls;
        for (int i = 1; i <= num; i++)
            tmp[i] = t[tmp[i]].ls;
        // 进入左子树,所有的结点全部进入左子树
```

```
        return query_num(l, mid, k);
    } else {
        for (int i = 1; i <= cnt; i++)
            tem[i] = t[tem[i]].rs;
        for (int i = 1; i <= num; i++)
            tmp[i] = t[tmp[i]].rs;
        // 进入右子树,所有的结点全部进入右子树
        return query_num(mid + 1, r, k - sum);
    }
}
```

对于查询某个数排名的操作，与前者类似。先在树状数组上寻找对应的树，然后在这些树上查询小于该数的数字，最后加 1 即可，代码如下：

```
int query_rnk(int l, int r, int k) {
    if (l == r) {
        return 0;
    }
    int mid = l + r >> 1, sum = 0;
    // 统计左子树的个数
    if (k <= mid) {
        for (int i = 1; i <= cnt; i++)
            tem[i] = t[tem[i]].ls;
        for (int i = 1; i <= num; i++)
            tmp[i] = t[tmp[i]].ls;
        // 进入左子树,所有的结点全部进入左子树
        return query_rnk(l, mid, k);
    } else {
        for (int i = 1; i <= cnt; i++)
            sum += t[t[tem[i]].ls].v, tem[i] = t[tem[i]].rs;
        for (int i = 1; i <= num; i++)
            sum -= t[t[tmp[i]].ls].v, tmp[i] = t[tmp[i]].rs;
        // 进入右子树,所有的结点全部进入右子树
        return sum + query_rnk(mid + 1, r, k);
    }
}
int find_rnk(int l, int r, int k) {
    cnt = num = 0;
    for (int i = r; i; i -= lb(i)) {
        tem[++cnt] = rt[i];
    }
    for (int i = l - 1; i; i -= lb(i)) {
        tmp[++num] = rt[i];
    }
    return query_rnk(1, len, k) + 1;
}
```

对于前驱与后继的寻找，可以通过查询第 $k$ 大和查询排名两个操作一同实现。对于查询前驱操作，只需要查询比其排名小 1 的树即可，后继操作类似，代码如下。

```
int find_pri(int l, int r, int k) {
    int rk = find_rnk(l, r, k) - 1;
    if (rk == 0)
        return 0; // 特判如果其已经是最小数的情况
    return find_num(l, r, rk);
}
int find_nxt(int l, int r, int k) {
    if (k == len)
        return len + 1; // 特判输入已经达到数据范围上限的情况
    int rk = find_rnk(l, r, k + 1); // 查询k+1的排名,查到的即为1到k中的数的个
    // 数+1
    if (rk == r - l + 2)
        return len + 1; // 特判其已经是最大数的情况
    return find_num(l, r, rk);
}
```

最终合并程序代码如下：

**代码片段（需结合上文的声明和函数）：**

```
#include <bits/stdc++.h>
#define MAXN 100005
using namespace std;
int len = 0;
const int inf = 2147483647;
struct seg {
    int v, ls, rs;
} t[MAXN * 128];
int rt[MAXN], n, m, tot, tem[MAXN], tmp[MAXN], cnt, num;
int lsh[MAXN << 1], a[MAXN];
struct cz {
    int a, b, c, d;
} q[MAXN];
// 这里省略函数lb、add、pushup、change、find_num
// 以及 query_num、query_rnk、find_rnk、find_pri、find_nxt的实现
int main() {
    scanf("%d%d", &n, &m);
    tot = cnt = num = 0;
    for (int i = 1; i <= n; i++) {
        scanf("%d", &a[i]);
        lsh[++len] = a[i];
    }
    for (int i = 1; i <= m; i++) {
        scanf("%d%d%d", &q[i].a, &q[i].b, &q[i].c);
        if (q[i].a != 3)
            scanf("%d", &q[i].d);
        else
            lsh[++len] = q[i].c;
        if (q[i].a == 4 || q[i].a == 5)
            lsh[++len] = q[i].d;
    }
    sort(lsh + 1, lsh + len + 1);
```

```
    len = unique(lsh + 1, lsh + len + 1) - lsh - 1;
    for (int i = 1; i <= n; i++) {
        a[i] = lower_bound(lsh + 1, lsh + 1 + len, a[i]) - lsh;
        add(i, 1);
    }
    lsh[0] = -inf;
    lsh[len + 1] = inf;
    for (int i = 1; i <= m; i++) {
        if (q[i].a == 3) {
            add(q[i].b, -1);
            a[q[i].b] = lower_bound(lsh + 1, lsh + 1 + len, q[i].c) - lsh;
            add(q[i].b, 1);
        }
        if (q[i].a == 1) {
            q[i].d = lower_bound(lsh + 1, lsh + 1 + len, q[i].d) - lsh;
            printf("%d\n", find_rnk(q[i].b, q[i].c, q[i].d));
        }
        if (q[i].a == 2) {
            printf("%d\n", lsh[find_num(q[i].b, q[i].c, q[i].d)]);
        }
        if (q[i].a == 4) {
            q[i].d = lower_bound(lsh + 1, lsh + 1 + len, q[i].d) - lsh;
            printf("%d\n", lsh[find_pri(q[i].b, q[i].c, q[i].d)]);
        }
        if (q[i].a == 5) {
            q[i].d = lower_bound(lsh + 1, lsh + 1 + len, q[i].d) - lsh;
            printf("%d\n", lsh[find_nxt(q[i].b, q[i].c, q[i].d)]);
        }
    }
    return 0;
}
```

### 例 8-4 三维偏序(陌上花开)(洛谷P3810)

有 $n$ 个元素，第 $i$ 个元素有 $a_i, b_i, c_i 3$ 个属性，设 `f(i)` 表示同时满足 $a_j \leqslant a_i, b_j \leqslant b_i, c_j \leqslant c_i$ 且 $j \neq i$ 的 $j$ 的数量。对于 $d \in [0, n)$ ，求 $f(i) = d$ 的数量。

**数据范围：** $1 \leqslant n \leqslant 10^5, 1 \leqslant a_i, b_i, c_i \leqslant k \leqslant 2 \times 10^5$。

**分析：**首先介绍离线算法和在线算法相关的概念。所谓离线算法，即在使用核心算法处理数据前，便已经获得所有的输入数据。本题在开始算法之前已经知道所有元素的所有属性，故可以采用离线算法。**离散化**就是一种典型的离线算法，因为需要获得所有的数据后才可以将这些数据进行排序。

与离线算法对应的是在线算法，即在开始前不知道所有的输入数据，需要动态处理。在算法竞赛中，有部分题目的输入数据与上一次询问的结果有关①，此类题目会要求选手必须使用在线算法完成问题，要求会比离线问题更高。

考虑将问题简化为两个属性（设为 $A$ 和 $B$）的时候该如何解决。一种简单的方式是，使用离线算法，先按照属性 $A$ 进行排序并按顺序遍历，再用权值**线段树**对前 $i$ 个元素中的属性 $B$ 进行维护。当插入一个元素时，只要在权值线段树中查找属性 $B$ 小于或等于该元素的元素个数，并统计总数即可。当属性个数达到 3 个时，依然先按照某一元素进行排序，从而将需要考虑的属性减少至两个。

使用**树状数组**套线段树解决这个问题。在对属性 $A$ 进行排序后，依次遍历所有元素。用外层树状数组维护属性 $B$，用内层的线段树维护属性 $C$。例如，对于一个属性 $B$ 为 $b$、属性 $C$ 为 $c$ 的 C。元素，在 $b$ 在树状数组上所对应的所有权值树的 $c$ 位置插入一个元素即可。

当遍历到某一元素时，先查询目前已遍历的元素中 $B$ 和 $C$ 两个属性都比当前元素属性小的元素个数。若当前元素的属性 $B$ 为 $b$，属性 $C$ 为 $c$，则等价于查询第 1 到 $b$ 棵树中值域在[1，$c$]值域内的数并统计答案。在完成查询后，将元素插入到树套树中，并按顺序遍历下一元素。参考代码：

```
#include <bits/stdc++.h>
#define MAXN 100005
using namespace std;
int n, k, ans[MAXN], tot, root[MAXN << 2], tmp[MAXN << 2], cnt, f[MAXN];
struct _node {
    int x, y, z, id;
    friend bool operator<(_node aa, _node bb) {
        if (aa.z == bb.z) {
            if (aa.y == bb.y)
                return aa.x < bb.x;
            return aa.y < bb.y;
        }
        return aa.z < bb.z;
    }
} node[MAXN];
struct _tree {
    int ls, rs, v;
} t[MAXN * 128];
int lb(int x) { return x & -x; }
void pushup(int o) { t[o].v = t[t[o].ls].v + t[t[o].rs].v; }
void change(int &o, int l, int r, int w, int v) {
    if (!o)
        o = ++tot;
    if (l == r) {
        t[o].v += v;
        return;
    }
    int mid = l + r >> 1;
    if (w <= mid)
        change(t[o].ls, l, mid, w, v);
    else
        change(t[o].rs, mid + 1, r, w, v);
    pushup(o);
}
void add(int o, int w, int v) {
    for (int i = o; i <= k; i += lb(i))
        change(root[i], 1, k, w, v);
}
int query(int l, int r, int w) {
    if (l == r) {
        int res = 0;
        for (int i = 1; i <= cnt; i++)
            res += t[tmp[i]].v;
        return res;
    }
    int mid = l + r >> 1, sum = 0;
    if (w <= mid) {
        for (int i = 1; i <= cnt; i++)
```

```
            tmp[i] = t[tmp[i]].ls;
        return query(l, mid, w);
    } else {
        for (int i = 1; i <= cnt; i++)
            sum += t[t[tmp[i]].ls].v, tmp[i] = t[tmp[i]].rs;
        return sum + query(mid + 1, r, w);
    }
}
int find(int r, int w) {
    cnt = 0;
    for (int i = r; i; i -= lb(i))
        tmp[++cnt] = root[i];
    return query(1, k, w);
}
int main() {
    scanf("%d%d", &n, &k);
    for (int i = 1; i <= k; i++)
        root[i] = ++tot;
    for (int i = 1; i <= n; i++) {
        scanf("%d%d%d", &node[i].x, &node[i].y, &node[i].z);
        node[i].id = i;
    }
    sort(node + 1, node + n + 1);
    for (int i = 1; i <= n;) {
        int j = i;
        while (j < n && node[j].z == node[j + 1].z)
            add(node[j].x, node[j].y, 1), j++;
        add(node[j].x, node[j].y, 1);
        j = i;
        while (j < n && node[j].z == node[j + 1].z)
            f[node[j].id] = find(node[j].x, node[j].y), j++;
        f[node[j].id] = find(node[j].x, node[j].y);
        i = j + 1;
    }
    for (int i = 1; i <= n; i++)
        ans[f[i]]++;
    for (int i = 1; i <= n; i++)
        printf("%d\n ", ans[i]);
    return 0;
}
```

> **注：** 1. 例如输入的值需要异或上一询问的答案。

## 8.3 课后习题与实验

### 习题 1 序列(洛谷P4093,天津、河南省队选拔2016)

有一个数列，其中某些项的值可能会变化，但同一个时刻最多只有一个值发生变化。能否选出一个子序列，使得在任意一种变化中，这个子序列都是不降的？求出这个子序列的最长长度。

输入的第 $1$ 行有两个正整数 $n$ 和 $m$，分别表示序列的长度和变化的个数。接下来一行有 $n$ 个整数，表示这个数列原始的状态。接下来 $m$ 行，每行有两个整数 $x$ 和 $y$，表示数列的第 $x$ 项可以变化成 $y$ 这个值。

测试数据满足所有数字均为正整数，且小于或等于 $10^5, 1 \leqslant x \leqslant n$。

### 习题 2 动态逆序对(洛谷P3157,重庆市省队选拔2011)

对于序列 $a$，它的逆序对数定义为集合{（$i$，$j$）1i<$j$ 且 $a_i > a_j$ 中的元素个数。现在给出 $1$ 到 $n$ 的一个排列，按照一定顺序依次删除 $m$ 个元素。要求在每次删除一个元素之前统计整个序列的逆序对数。

**数据范围：** $1 \leqslant n \leqslant 10^5, 1 \leqslant m \leqslant 50000.$

### 习题 3 美味菜品(洛谷P3293,四川省队选拔2016)

一家餐厅有 $n$ 道菜，编号为 $1$，$2$，…，$n$，大众对第 $i$ 道菜的评价值为 $a_i$。 有 $m$ 位顾客，第 $i$ 位顾客的期望值为 $b_i$，而他的偏好值为 $x_i$。 因此，第 $i$ 位顾客认为第 $j$ 道菜的美味度为 $b_i \text{xor} (a_j + x_i)$，其中xor表示异或运算。

第 $i$ 位顾客希望从这些菜中挑出他认为最美味的菜，即美味值最大的菜，但由于价格等因素，他只能从第 $l_i$ 道到第 $r_i$ 道中选择。请帮助他们找出最美味的菜。

**数据范围：**
$1 \leqslant n \leqslant 2 \times 10^5, 0 \leqslant a_i, b_i, x_i < 10^5, 1 \leqslant l_i \leqslant r_i \leqslant n (1 \leqslant i \leqslant m), 1 \leqslant m \leqslant 10^5, 1 \leqslant l_i \leqslant r_i \leqslant 10^5.$

### 习题 4 人行步道(洛谷CF960F)

给定 $n$ 个点、$m$ 条边的有向图，可能不连通，可能有重边，也可能会有自环。求最长的路径（可以经过重复结点），使得这条路径的编号和权值都严格单调递增，其中路径的编号指输入的顺序。路径的长度是指经过边的数量。

**数据范围：** $1 \leqslant n, m, w_i \leqslant 10^5$。

### 习题 5 动态排名(洛谷P2617)

给定一个含有 $n$ 个数的序列 $a_1, a_2, \cdots, a_n$，进行 $m$ 次操作，每次操作为以下两种操作中的一种。

1. $Q\ \ l\ r\ k$：表示查询下标在区间$[l, r]$中的第 $k$ 小的数。
2. $C\ x\ y$：表示将 $a_x$ 改为 $y$

**数据范围：** $, 1 \leqslant n, m \leqslant 10^5, 1 \leqslant l \leqslant r \leqslant n, 1 \leqslant k \leqslant r - l + 1, 1 \leqslant x \leqslant n, 0 \leqslant a_i, y \leqslant 10^9 \leqslant a_i.$

### 习题 6 任务查询系统(洛谷P3168,重庆市省队选拔2011)

超级计算机中的任务用三元组 $(s_i, e_i, p_i)$ 描述：任务从第 $s_i$ 秒开始，在第 $e_i$ 秒结束后停止，两个端点对应的秒都在运行，优先级为 $p_i$。同一时间可以有多个任务执行，优先级可以相同。

调度系统会经常向查询系统询问，第 $x_i$ 秒正在运行的任务中，优先级最小的 $k_i$ 个任务（即将任务按照优先级从小到大排序后取前 $k_i$ 个）的优先级之和是多少?特别地，如果 $k_i$ 大于第 $x_i$ 秒正在运行的任务总数，则直接回答第 $x_i$ 秒正在运行的任务优先级之和。上述所有参数均为整数，时间的范围在 $[1, n]$ 之间。在线询问①。

**数据范围：** $1 \leqslant m, n, s_i, e_i, k_i, a_i, b_i \leqslant 10000, 1 \leqslant p_i \leqslant 10^7, x_i$ 为 $1$ 到 $n$ 的一个排列。

### 习题 7 中值(洛谷P2839,国家集训队作业)

一个长度为 $n$ 的序列 $a$，设其排过序之后为 $b$，其中位数定义为 $b_{\lfloor n/2 \rfloor}$，其中 $a$ 和 $b$ 从 $0$ 开始标号，除法取下整。给一个长度为 $n$ 的序列 $s_0$ 回答 $Q$ 个这样的询问：$s$ 的左端点在 $[a,b]$ 之间，右端点在$[c,d]$之间的子区间中，最大的中位数。其中 $a<b<c<d$，位置也从 $0$ 开始标号。

数据范围 $\cdot n \leqslant 20000, Q \leqslant 25000$。

### 习题 8 混合果汁(洛谷P4602,CTSC2018)

商店里有 $n$ 种果汁，编号为 $0,1,\cdots,n-1$。 $i$ 号果汁的美味度是 $d_i$，每升价格为 $p_i$。 小 $R$ 在制作混合果汁时，还有一些特殊的规定，即在一瓶混合果汁中，$i$ 号果汁最多只能添加 $l_i$ 升。

现在有 $m$ 个小朋友过来找小 $R$ 要混合果汁喝，他们都希望小 $R$ 用商店里的果汁制作成一瓶混合果汁。其中，第 $i$ 个小朋友希望他得到的混合果汁总价格不大于 $\xi_j$，体积不小于 $L_j$， 。在上述这些限制条件下，小朋友们还希望混合果汁的美味度尽可能高，一瓶混合果汁的美味度等于所有参与混合的果汁的美味度的最小值。请计算每个小朋友能喝到的最美味的混合果汁的美味度。如果无法满足他的需求，则输出 $-1$。

数据范围 $:n, m \leqslant 100000, 1 \leqslant d_i, p_i, l_i \leqslant 10^5, 1 \leqslant g_j, L_j \leqslant 10^{18} \leqslant$

### 习题 9 路灯(洛谷P5445,APIO2019)

一辆车在街道上行驶，该街道上有 $n+1$ 个停车站点，将街道划分成 $n$ 条路段。每一路段都拥有一个路灯。当第 $i$ 个路灯亮起，它将照亮连接第 $i$ 与第 $i+1$ 个站点的路段。否则这条路段将是黑暗的。

> **注：** 1. 在线询问，即需要使用例 8-4 中提到的在线算法。

安全起见，出租车只能在被照亮的路段上行驶。在经过一些意外故障或修理之后，街道上的路灯可能是亮起的，也可能是熄灭的。现在给定 $0$ 时刻时，街道上路灯的初始状态。之后 $1, 2, \cdots, q$ 时刻，每时刻会发生下列两种事件之一。

1. togglei：切换第 $i$ 个路灯的状态。具体来说，若路灯原来亮起，则现在将熄灭；若路灯原 来熄灭，则现在将亮起。
2. query $a\,b$：出租车部门的负责人想知道，从 $0$ 时刻起到当前时刻，有多少个时刻满足出租车能够从站点 $a$ 出发到达站点 $b$。

**数据范围：** $1 \leqslant n, q \leqslant 3 \times 10^s, |s| = n, 1 \leqslant i \leqslant n, 1 \leqslant a < b \leqslant n+1$。

# 第3部分：图论

## 第9章 树

在人们的日常生活中，树结构可以刻画一种广泛存在的事物关系。例如，某家公司的一个部门可能有许多员工，但是他们只会有一个直属的中层领导；而每个部门的中层领导，只会有一个直属的上司——总经理。这种组织架构就是树形结构。如果同一部门的两个员工工作发生矛盾，可以找部门的中层领导裁决，因为这两名员工都是这个领导的下属；但两个不同部门的员工工作发生矛盾，就有可能需要提请总经理裁决了。

在《基础篇》的“二叉树”一章中已经对树结构有了初步的认识。树结构是一种常见的数据组织形式，体现的关系是一种“一对多”的关系。本章会进一步对树形数据结构的一些性质进行探究，并且探讨树上的一些特殊的关系。本章思维导图如图 9-1 所示。




图9-1 本章思维导图

## 9.1 树的性质与遍历

### 例 9-1 公司的组织架构

某科技公司的组织架构可以抽象为如图 9-2 所示的关系图。

首先观察每一个人（结点），都可以通过连线联系到公司里的任何一个其他人。树是一种特殊的图，树必须是连通的。同时可以发现，这棵树总共有 $10$ 个结点，连接它们的是 $9$ 条边，边数正好比点数少 $1$。这也是树的一个重要的性质：一个有 $n$ 个结点的树，一定会有 $n-1$ 条边。

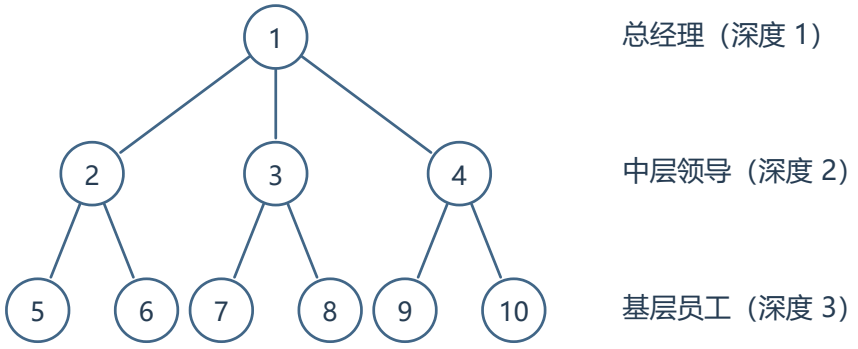



图9-2 公司关系图

除了总经理之外，其余的每一个人都接受自己的一个直属上司的管辖。以结点 $1$（总经理）和结点 $2$（中层领导）为例，结点 $1$ 被称作结点 $2$ 的父结点；相对应的，结点 $2$ 就被称作结点 $1$ 的子结点。结点 $2$ 与结点 $3$ 和结点 $4$ 之间是同级关系，它们互相被称作兄弟结点，因为它们有同一个父结点。这家公司的总经理不存在任何直属上级，因此它就是这棵树的根结点。接下来考虑结点 $5$（基层员工）. 它没有子结点，是这棵树最外层的结点。如同现实中一棵树最外面的部分是叶子一样，这样的结点被称作叶结点。

上文提及的树，为它确定了一个根结点 $A$，因此它被称作有根树，除此之外还有一种树，它没有明确的根结点，这种树被称作无根树。

从结点 $5$ 的视角观察，它的父结点是结点 $2$，而结点 $2$ 的父结点是结点 $1$。那么对应到树结构上，从结点 $5$ 的父结点开始一直到根结点经过的所有结点，被称作结点 $5$ 的祖先。相对应地，结点 $5$ 则被称作这些结点的后代。

一个结点的所有后代，包括它自身，会构成一棵单独的树。例如对于结点 $2$ 而言，结点 $5$ 和结点 $6$ 都是它的后代，接受其管理。故结点 $2$、$5$、$6$ 构成了结点 $2$ 的子树，子树内的所有结点除结点 $2$ 本身外都是其后代，称这棵子树为结点 $2$ 的子树，结点 $2$ 就是这棵子树的根结点。特别地，根结点 $1$ 没有祖先，除根结点以外的所有结点都是根结点的后代，也就是说根结点的子树就是这棵树本身。

### 例 9-2 猫猫和企鹅(洛谷P5908)

动物王国里有 $n(n\leqslant 10^5)$ 个居住区，它们之间有 $n-1$ 条道路相连，并且保证从每个居住区出发都可以到达任何一个居住区，并且每条道路的长度都为 $1$。

除 $1$ 号居住区外，每个居住区住着一只小企鹅，有一天一只猫猫从 $1$ 号居民区出发，想要去拜访一些小企鹅。可是猫猫非常懒，它只愿意去找距离它在 $d$ 以内的小企鹅们。请你告诉它可以拜访多少只小企鹅。

分析。首先要分析出题目给定的图到底是什么样子。将居住区抽象为点，连接居住区的路就是边，这样题目的条件就转化为：有 $n$ 个点，被 $n-1$ 条边相连，同时保证连通。按照上文所讲，可以判断这是一棵树。如果从 1 号结点到达某一结点的路不止一条，应当如何计算呢？这里需要运用到树的另一个非常好的性质：树上任意两点之间，经过不重复结点的路径只有一条。这个性质保证可以唯一地算出每个点到 1 号结点的距离。

对于有根树而言，规定了根结点后，每个结点就自动按照距离根结点的距离划分了层次，层次性是树结构的一个重要性质。每个结点距离根结点的距离则被称作这个结点的深度；同时定义深度最大的那个结点的深度为这棵树的高度。

本题需要计算出每一个点的深度 `dis`，并且统计出 $dis \leqslant d$ 的结点个数。从根结点开始，对整棵树进行遍历，同时计算每个点的深度。可以用深度优先遍历（DFS）和广度优先遍历（BFS）来遍历一张图。树是一种特殊的图，可以使用相同的办法去遍历一棵树。

在遍历之前先考虑存储的问题，《基础篇》中提到，可以使用一个vector容器来存储一张图。在存储树的时候需要注意，由于题目中给的边的方向并未严格指明方向，所以需要作为无向图存储双向边。

解法 1：尝试深度优先遍历。假设有一棵这样的树，如图 9-3 所示。以 1 号结点作为根结点，进行深度优先遍历，过程如下：

1. 从根结点开始遍历。进入 1 号结点，扫描到 2 号结点。

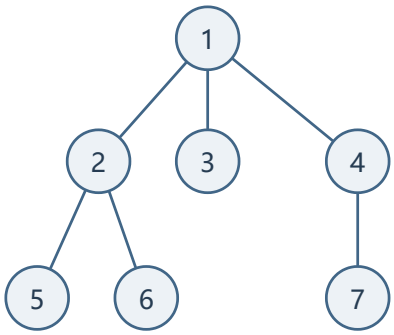



图9-3 一棵树的例子

2. 进入 2 号结点，扫描到 1 号结点，因为已经访问过，跳过它，继续扫描 5 号结点。
3. 进入 5 号结点，扫描到 2 号结点，跳过它，没有扫描到新的结点。
4. 回到 2 号结点，扫描到 6 号结点，进入。
5. 进入 6 号结点，扫描到 2 号结点，跳过，没有扫描到新的结点。
6. 回到 2 号结点，没有扫描到新的结点。
7. 回到 1 号结点，扫描到 3 号结点，进入。
8. 进入 3 号结点，扫描到 1 号结点，跳过，没有扫描到新的结点。
9. 回到 1 号结点，扫描到 4 号结点。

10. 进入 4 号结点，扫描到 1 号结点，跳过，扫描到 7 号结点。
11. 进入 7 号结点，扫描到 4 号结点，跳过，没有扫描到新的结点。
12. 回到 4 号结点，没有扫描到新的结点。
13. 回到 1 号结点，没有扫描到新的结点，递归结束。

在 2 号结点中，扫描和它相连的所有结点，会发现 1 号结点也被扫描到。可以开辟一个数组 `vis`，如果遇到已经被访问过的结点就跳过。但是在树上有一个更好的性质：树上的点对之间路径是唯一的。因此若是想进入 2 号结点，只有可能从它的父结点，即 1 号结点进入。在递归的时候额外维护一个参数 $f$，表示当前结点是从哪个结点连过来的，如果扫描过这个结点就跳过它，这样就可以防止访问重复的结点。

接下来考虑如何在遍历的同时算出每个点的深度。记每个点的深度进入数组 `dis`，考虑到点 $v$ 和它的父结点 $u$ 之间存在这样的关系： $dis[v] = dis[u] + 1$，同时保证 $dis[1] = 0$。在进入新的结点时，用父结点的深度即可求出这个结点的深度。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5 + 10;
vector<int> G[N];
int n, m, ans, d, dis[N];
void addedge(int u, int v) {
    G[u].push_back(v);
    G[v].push_back(u); // 存下双向的边
}
void dfs(int u, int f) {
    if (dis[u] <= d)
        ans++; // 如果当前结点的深度符合条件,就计入答案
    for (int i = 0; i < G[u].size(); i++) {
        int v = G[u][i];
        if (v == f)
            continue;        // 如果扫描到的点是来时的点,就跳过它
        dis[v] = dis[u] + 1; // 用父结点 u的深度计算出v的深度
        dfs(v, u);
    }
}
int main() {
    scanf("%d%d ", &n, &d);
    for (int i = 1; i < n; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        addedge(u, v);
    }
    dfs(1, 0);
    printf("%d\n", ans - 1); // 因为1号结点是没有小企鹅的,所以-1
}
```

由于每个结点最多被新进入一次（注意回溯的时候返回这个结点并不算新进入），深度优先遍历的时间复杂度是 $O(n)$，空间复杂度也是 $O(n)$，因为只使用到了长度为 $n$ 的数组。至此，使用深度优先遍历解决了这个问题。

解法 2：如果使用广度优先遍历，应该怎么解决这个问题呢？以 1 号结点作为根结点，进行广度优先遍历。如果读者还记得《基础篇》中图的广度优先遍历相关内容，一定知道进行BFS的时候需要一个队列来进行维护，从队首中取出结点，找到这个结点的子结点，然后将子结点放入队列，直到队列为空。对本题来说，步骤如下：

1. 将起始的 1 号结点压入队列。
2. 弹出 1 号结点，扫描到相连的 2、3、4 号结点，将其全部加入队列中。
3. 弹出 2 号结点，跳过父结点 1 号结点，将相连的 5、6 号结点加入队列。
4. 弹出 3 号结点，跳过父结点 1 号结点，此时并未拓展出新的结点。
5. 弹出 4 号结点，跳过父结点 1 号结点，将相连的 7 号结点加入队列。
6. 弹出 5 号结点，跳过父结点 2 号结点，此时并未拓展出新的结点。
7. 弹出 6 号结点，跳过父结点 2 号结点，此时并未拓展出新的结点。
8. 弹出 7 号结点，跳过父结点 4 号结点，此时并未拓展出新的结点。
9. 此时队列为空，算法结束。

依旧沿用记录父结点的方法来防止重复访问，但是由于广度优先遍历并不是按照父结点直接跳到子结点这样的方法进行遍历的，因此需要开一个数组 `fa` 来记录每个结点的父结点是谁。记录父结点和计算深度可以在拓展新结点的时候同时完成。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1000010;
vector<int> G[N];
int n, m, ans, d, dis[N], fa[N];
queue<int> Q; // 使用 STL 的队列来维护
void addedge(int u, int v) {
    G[u].push_back(v);
    G[v].push_back(u);
}
void bfs(int u) {
    Q.push(u);
    while (!Q.empty()) {
        int u = Q.front();
        Q.pop(); // 每次遍历时弹出队头结点,并用它拓展新的结点
        if (dis[u] <= d)
            ans++;
        for (int i = 0; i < G[u].size(); i++) {
            int v = G[u][i];
            if (v == fa[u])
                continue;        // 如果扫描到父结点就跳过
            fa[v] = u;           // 记录下结点v的父结点
            dis[v] = dis[u] + 1; // 计算结点v的深度
            Q.push(v);           // 将新拓展出的结点v压人队列
        }
    }
}
int main() {
    scanf("%d%d ", &n, &d);
    for (int i = 1; i < n; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
```

```
            addedge(u, v);
        }
        bfs(1);
        printf("%d\n", ans - 1); // 同上
    }
```

由于广度优先遍历也可以保证每个结点只会被进入一次，故而广度优先遍历的时间复杂度也是 $O(n)$。由于数组长度只为 $n$，而且每个元素最多只会进入队列一次，这就保证了队列的最长大小也是 $n$，故而空间复杂度也是 $O(n)$。

在一些题目中，具有特殊形态的树也有一些特殊的名称，通常也伴随着特殊的性质。许多题目会提供特殊树的部分分解，可以把树上问题简化为序列问题或者是层数为 2 的树的问题，进而得到一些分数。

1. 链：如果一棵树除了最后一个结点，每个结点都只有一个子结点，可以发现此时树退化成了之前研究的线性表，也称为链，如图 9-4(a) 所示。当树退化成链时，从根结点开始采取深度优先遍历会产生最高的递归层数（与结点数相同），通常会用来检验程序是否会递归溢出，即所谓的“爆栈”①。但也可以将树上问题直接转变成序列问题，使问题简化。

> **注：** 1. 多数算法竞赛中栈空间和内存大小是一致的。

2. 菊花图：树的层数为 2，即除了根结点外每个结点都直接连向根结点，如图 9-4(b) 所示。当树的形态是“菊花图”时，某些算法会多次扫描点的连边，产生较高的复杂度。但正是因为这种树的深度为 2.故也可能存在一些可以暴力求解的性质。

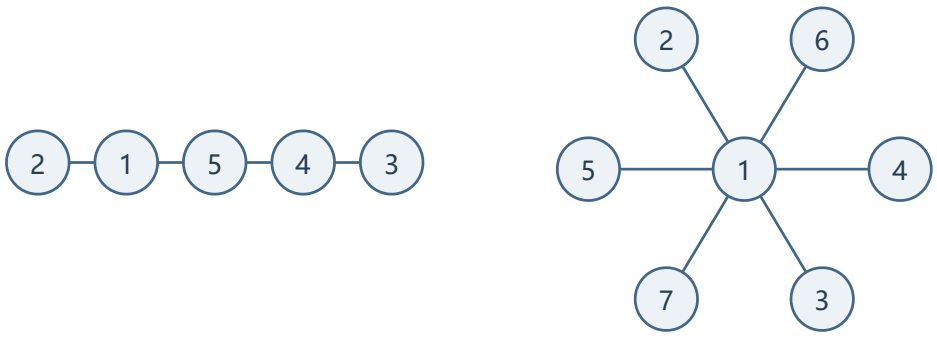



图9-4 两种特殊的树

## 9.2 树的直径与重心

树作为一种特殊的结构，其自身也具有一些特殊的性质来帮助解决一些树中的问题。本节将以树的直径与重心为例，讲解求解方法，以此为例来初步研究如何解决一些树中的问题。

### 例 9-3 树网的核(洛谷P1099,NOIP 2007提高组)

给定一棵有 $n$ 个结点的无根树，边带权。两点之间的唯一路径所包含的所有边的权之和可以称作这两点之间的距离。所有点对中最远的两点之间的距离被称作这棵树的直径。定义偏心距 `ECC(F)`：树中距路径 $F$ 最远的结点到路径 $F$ 的距离。

对于给定的树和非负整数 $s$，找到一个路径 $F$，是某直径上的一段路径（该路径两端均为树中的结点），其长度不超过 $s$，使偏心距 `ECC(F)` 最小，并输出这个值。必要时，$F$ 可以退化为某个结点。这条路径被称为树网的核。在上述定义下，符合要求的路径不一定只有一个，但最小偏心距是唯一的。

原题数据范围： $n \leqslant 300, 1 \leqslant w \leqslant 10^3, 0 \leqslant s \leqslant 10^3$，但这里希望设计出可以接受 $n \leqslant 5 \times 10^5$ 的算法。

**分析**：首先需要求得这个树的直径，其含义也可以从“直径”这个名词本身的含义来考虑——一个圆中的任意两点连接会形成一条弦，这些弦中最长的一条被称作圆的直径。因此，考虑两个关键词：“带边权”和“最远”，它蕴含着以下两条信息。

1. 带边权：一般而言树的直径是在有边权的树上进行讨论的，但是无边权的树也会存在直径这一概念，此时就相当于一个所有边权均为 1 的特殊情况。
2. 最远：如果每一对点之间经过的所有边和点形成的路径叫作树链，那么直径就是其中最长的一条树链。这一性质对很多题目的求解是有启发的。

求解树的直径有两遍DFS法和树形**动态规划**两种常见的方法。此处只介绍简单高效的两遍DFS法，望文生义，该方法的流程如下：

1. 随意选择一个结点 $x$ 作为起点，然后进行第一遍DFS，同时记录下其他结点距离 $x$ 的距离。遍历完之后，可以找到距离 $x$ 最远的结点，将它记作 $y$。
2. 将 $y$ 作为起点，重复上述的操作，再次找到距离 $y$ 距离最远的结点，记作 $z$。
3. 此时 $y$ 与 $z$ 就是直径的两个端点。这样就求出了这棵树的一条直径。

这一方法看起来非常简单，有些读者或许可以发现算法隐含了一个信息，就是在一棵树上距离任何一个结点最远的结点一定是直径的某一端点。其系统的证明较为晦涩，可以简单地去理解这一定理：

1. 如果第一次出发的结点 $x$ 在 $y$ 到 $z$ 的路径上，首先假设这条路径并不是直径，也就是存在比它更长的一条路径。但是如果存在比两个端点距离 $x$ 更远的结点 $k$，深度优先遍历应当会在这个更深的结点 $k$ 停止，而不是 $y$。因此不存在当前路径更长的路径，即它就是直径。
2. 如果 $x$ 不在直径上，那么此时依旧有两种情况：
3. $x$ 到距离其最远的结点 $u$ 与直径有交点。这意味着从交点到 $x$ 或者 $u$ 的距离长于交点到直径某一端点的距离。很显然，这与“直径最长”这一性质产生矛盾。
4. 如果 $x$ 到 $y$ 的路径和直径没有交点，那么如果从直径上距离这一路径的最近点引下来，通向该路径的一个端点，其形成的新的路径会比直径更长，这再次矛盾。

综上所述，第一遍DFS找到的一定是直径的某一端点。从这一端点继续开始DFS，就可以找到直径的第 2 个端点，这样就可以求出树的直径。

首先回顾如何用DFS求出每个点到起始点的距离。根据 例 9-2 的经验，可以大概摸索出一个树中问题的通用思路：对于当前需要求的结点的值，在递归进入该结点的过程中，利用之前求出的父结点信息，或者是在回溯返回这一结点的时候利用之前进入子结点时求出的信息去计算出这一结点的信息。

对本题来说，需要利用父结点的信息去求出当前结点的距离。具体而言，如果当前结点记作 $v$，父结点记作 $u$，连接 $u$ 和 $v$ 的边的边权为 $w$，所有距离存在数组 `dis` 中，那么可以发现，**`v到起始点的距离就是由起始点到u的距离dis[u]`** 和 $u$ 到 $v$ 的距离 $w$ 这两部分拼接而成，故而可以得到：

`dis[v]` = `dis[u]` +$w$。借此可以更新出 $v$ 到起始点的距离。使用数组 `fa` 来记录每一个结点的父结点，这样就可以记录从叶子结点到根结点的路径的结点编号。

最后一个问题，如何求出题目要求的最小偏心距?对于每一个直径上的结点 $x$，都可以求得它不经过直径上的其他结点可以到的最远距离，求法和上面提到的DFS是一样的，只是不能进入特定的结点。接着，在整条直径上使用**双指针法**，维护一段总长度不超过 $s$ 的区间（假设端点分别为 $i$ 和 $j$）中的 `dis` 的最大值，这可以使用**单调队列**来维护。

但实际上，也可以不使用单调队列。需要求得的最小偏心距，要么是由区间的两端点到直径的一端贡献的（取较大值），要么就是直径上的一点不经过直径其他结点可以到达最远距离贡献的，请读者自行证明。可以在使用双指针法的时候求得区间端点可以走的最远距离，并统计直径上的所有结点不经过直径上的其他结点可以到的最远距离，就可以得到答案。代码如下：

```
#include <bits/stdc++.h>
#define MAXN 500000 + 10
using namespace std;
int n, s, ans = MAXN * 1000, far;
int dis[MAXN], fa[MAXN], flag[MAXN];
struct edge {
    int to, w;
};
vector<edge> G[MAXN];
void dfs(int u, int f) {
    fa[u] = f;
    if (dis[u] > dis[far])
        far = u; // 记录最远的距离
    for (int i = 0; i < G[u].size(); i++) {
        int v = G[u][i].to, w = G[u][i].w;
        if (v == f || flag[v])
            continue;
        dis[v] = dis[u] + w;
        dfs(v, u); // 遍历下一个
    }
}
int main() {
    scanf("%d%d", &n, &s);
    for (int i = 1; i < n; i++) {
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        G[u].push_back((edge){v, w});
        G[v].push_back((edge){u, w});
    }
    int A, B; // 直径的两端
    dis[1] = 1;
    dfs(1, 0);
    A = far; // 第1次DFS
    dis[far] = 0;
    dfs(far, 0);
    B = far; // 第2次 DFS
    for (int i = B, j = B; i; i = fa[i]) {
        while (dis[j] - dis[i] > s)
            j = fa[j];                          // 双指针法
        int x = max(dis[B] - dis[j], dis[i]); // 两端最远距离
```

```
            ans = min(ans, x);
        }
        for (int i = B; i != 0; i = fa[i])
            flag[i] = 1; // 直径上的点跳过
        for (int i = B; i != 0; i = fa[i]) {
            dis[i] = 0;
            dfs(i, fa[i]); // 计算直径上每个点可以访问的远距离
        }
        for (int i = 1; i <= n; i++)
            ans = max(ans, dis[i]); // 得到所有点可以访问最远距离的最大值
        printf("%d\n", ans);
        return 0;
    }
```

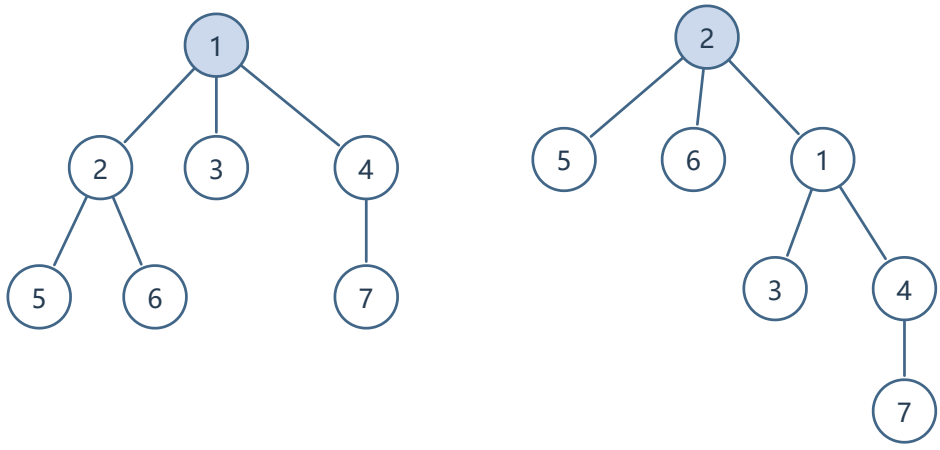
求直径时，每次深度优先遍历的时间复杂度是 $O(n)$；接着双指针法中区间端点符合单调性，统计所有符合要求区间端点可以到达最短距离的时间复杂度也是 $O(n)$；最后枚举直径上所有结点能到的最远距离，每个结点最多访问一次，时间复杂度还是 $O(n)$。所以，最后的时间复杂度就是 $O(n)$。

### 例 9-4 会议(洛谷P1395)

有一个村庄居住着 $n(n \leqslant 5 \times 10^4)$ 个村民，有 $n-1$ 条路径使这 $n$ 个村民的家连通，每条路径的长度都为 $1$。现在村民委员会主任希望在某个村民家中召开一场会议，并希望所有村民到会议地点的距离之和最小。那么村民委员会主任应该要把会议地点设置在哪个村民的家中?距离总和最小是多少?若有多个结点都满足条件，则选择编号最小的那个结点。

**分析：**可能大家都了解物理学中关于重心的知识，简单来说，如果在一个物体的重心处悬挂该物体，那么这个物体将会保持平衡。也就是说，在一个物体的重心处，它的其余各部分的质量分布是均衡的，这也就是它为什么可以保持平衡的原因之一。

那么如果把这个概念沿用到树上，将会是怎么样的呢？如图 9-5(a) 所示，可以发现，假设选择 1 号结点当作根结点，那么这棵树实际上就被它分成了若棵子树。如果这些部分的大小接近的话，这棵树就显得比较平衡。但如果选择 2 号结点作为根结点，如图 9-5(b) 所示，其子树的大小差距就比较大。




(b) (a)

图9-5 树的平衡性

如何衡量这棵树的“平衡程度”呢？如果想让一棵树相对平衡，可以去关注最大的一棵子树。有如下的定义：在一棵树中，如果选择某个结点为根，可以使得它的所有子树中最大的子树最小，那么这个结点就被称作这棵树的重心。从这个定义中，可以推出重心的 4 个性质：

1. 以重心为树根时，所有子树的大小不超过全树大小的一半。

可以用这样的方法去理解：假设当前选择的重心是 $u$，与它相连的子树 $v$ 的大小超过了全树的一半。那么将 $v$ 变为新的重心，让这个最大的子树的大小至少减小 1。依此类推，让所有的子树大小都不超过全树大小的一半，如图 9-6 所示。

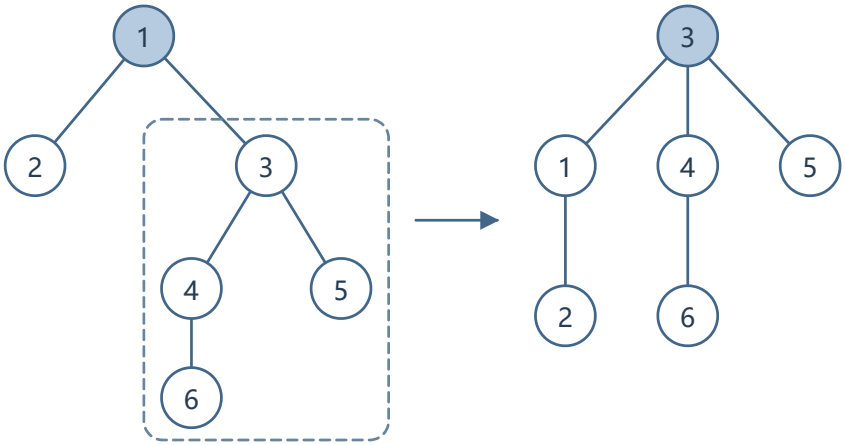



图9-6 性质1的例图

2. 树中所有点到某个点的距离和中，到重心的距离和是最小的；如果有两个重心，那么到它们的距离和一样。

这一点可以由性质 1 来延伸理解。既然最终选择的重心保证了其他所有子树的大小不超过全树的一半，假设重心往别的结点移动了，那么会发生如下的情况：某一个子树内所有结点移动代价减少 1，而该子树外的所有结点移动代价增加 1。因为所有子树的大小都不会超过全树的一半，增加的代价一定会大于或等于减少的代价，所以选择重心最优，如图 9-7 所示。

3. 把两棵树通过一条边相连得到一棵新的树，那么新树的重心在连接原来两棵树的重心的

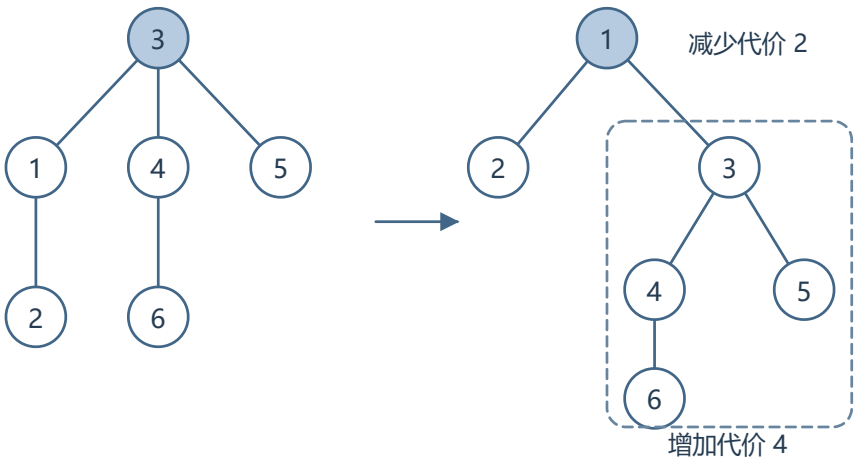



图9-7 性质2的例图

路径上。

依旧可以采取和性质 2 同样的方法去理解。一旦重心偏离了这个路径，那么两棵树剩下的部分移动的代价都会增加，只有进入的结点对应的子树内移动的代价会减少；然而这个子树的大小不会超过原来任何一棵树的一半，所以新的重心不可能出现在连接原来两棵树的重心的路径之外，如图 9-8 所示。

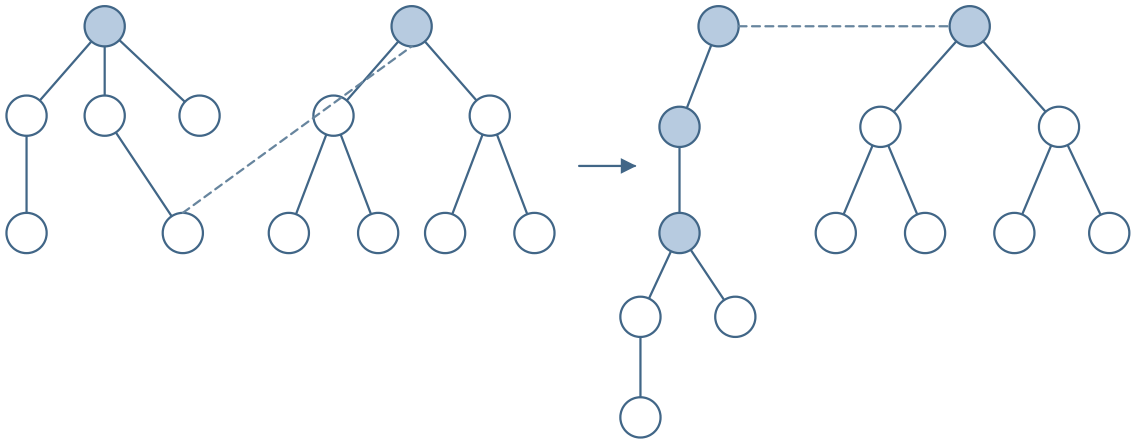


图9-8 性质3的例图

4. 在一棵树上添加或删除一个结点，那么它的重心最多只移动一条边的距离。

沿用性质 1 的方法，如果增加一个结点，最多只可能使某一棵子树的大小比一半多 1，此时只要移动一次重心即可解决问题。

在理解了重心的性质之后，接下来面临的问题就是如何求重心了。

首先，如何用DFS求出每个子树的大小呢？沿用之前总结的经验，求解树中问题时，进入该点的时候通过父结点的信息推出这个结点的信息，或是在回溯进入该结点的时候通过子结点的信息推出这个结点的信息。

从这个问题的角度出发，可以发现，一个结点的子树可以被进一步分解成它的所有子结点形成的子树之和再加上它本身。那么答案也就很明确了，应当在回溯到该结点的时候再推算出这个结点的信息。所以对于求每个结点的子树大小，设当前进入的点为 $u$，，将结点 $u$ 的子树大小记录进数组 $size[u]$ 中，那么可以得到以下的流程：

1. 初始化 $size[u]=1$ （因为它的子树首先包含它自己这一个结点）。
2. 逐一枚举 $u$ 的所有子结点，然后递归DFS，进入这些子结点。
3. 回溯回到 $u$ 的时候，将该子结点的 `size` 累加进 `size[u]` 中。

可以写出以下函数，假设用vector存储了这棵树：

```
void dfs(int u, int fa) {
    size[u] = 1; // 初始化这个点,只包含它自己
    for (int i = 0; i < G[u].size(); i++) {
        int v = G[u][i];
        if (v == fa)
            continue;
        dfs(v, u);          // 先进入子结点
        size[u] += size[v]; // 将其累加进 size[u]
    }
}
```

回到重心问题。严格按照定义的思路，不停地让每个点为根，然后找到它最大的子树进行比较，最大子树最小的那个点就是重心了。这个做法的时间复杂度是 $O(n^2)$，效率较低。

观察如图 9-9 所示的树，将其中一个结点 $u$ 转变为根结点的时候，它的子树的构成。

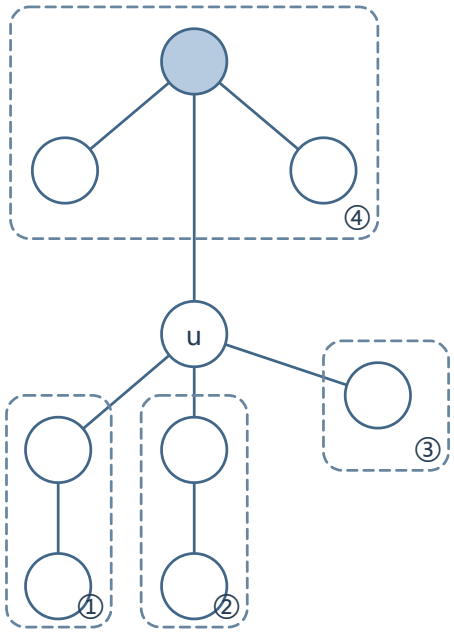



图9-9 计算子树所有子树大小

其中第 1、2、3 部分都是之前就计算过的子树大小，没有必要重新计算。唯一棘手的地方在于 第4部分，也就是原来的父结点连通的其他结点数量。直接求这一部分的大小有一点困难，但是此时可以反过来思考。仔细观察 图 9-9，会发现 第4部分 其实就是整棵树除去 $u$ 的子树外的剩下的部分。因此记这棵树总的结点数为 $n$，在计算出 `size[u]` 之后，可以得到 $n$-`size[u]`，就是想要的答案。令fu]为把 $u$ 作为根结点，其最大的子树的大小。在遍历的过程中同时统计数组 $f$ 的最小值和对应的结点编号，最后留下的结点编号就是树的重心。

知道了树的重心后，求其他结点到这个结点的距离总和就很简单了，有很多种办法可以求得。这里使用BFS求解，代码如下：

```
#include <bits/stdc++.h>
const int MAXN = 1000010;
const int inf = 0x7f7f7f7f;
using namespace std;
int f[MAXN], size[MAXN], head[MAXN], dep[MAXN];
int n, center, sum = 0; // center 为重心
vector<int> G[MAXN];
queue<int> q;
void getcenter(int u, int fa) {
    size[u] = 1;
    f[u] = 0;
    for (int i = 0; i < G[u].size(); i++) {
        int v = G[u][i];
        if (v == fa)
            continue;
        getcenter(v, u);
        size[u] += size[v];
        f[u] = max(f[u], size[v]); // f[u]保存了结点u的最大子树大小
    }
    f[u] = max(f[u], n - size[u]);
    // size[u]计算完之后,计算树中除了该结点的部分,并和之前的f[u]取较大值
    if (f[u] < f[center] || (f[u] == f[center] && u < center))
        center = u; // 如果f[u]较小,更新找到的重心
}
void bfs() {
    q.push(center);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int i = 0; i < G[u].size(); i++) {
            int v = G[u][i];
            if (dep[v] || v == center)
                continue;
            dep[v] = dep[u] + 1; // 深度
            sum += dep[v];       // 总距离要加上这个结点的深度
            q.push(v);
        }
    }
}
int main() {
    cin >> n;
    for (int i = 1; i < n; i++) {
        int u, v;
        cin >> u >> v;
        G[u].push_back(v);
        G[v].push_back(u);
    }
    center = 0;
    f[0] = inf;
    getcenter(1, 0); // 求重心
    bfs();           // 从重心开始求距离
    cout << center << " " << sum;
    return 0;
}
```

当然，也可以使用DFS求解所有点到重心的距离之和。无论哪种解法，整个算法的时间复杂度是 $O(n)$。

## 9.3 最近公共祖先

### 例 9-5 最近公共祖先(洛谷P3379)

给定一棵 $n$ 个结点的有根多叉树（以 $S$ 结点为根），请求出指定两个点最近的公共祖先。共有 $m$ 次询问，每次询问给出两个正整数 $a$ 和 $b$，请输出结点 $a$ 和 $b$ 的**最近公共祖先**的结点编号。

```
// 数据范围：n≤500000,m≤500000。
```

**分析：**每年过年走亲访友时，“认亲戚”是一个常见的话题。两个亲戚在聚会中碰面并开始聊天，不可避免地谈论辈分和称呼的话题。对于他们而言，最简单的开始方法莫过于：“我爸跟

你爸是兄弟”“咱俩太爷爷是同一个人”等。一个族谱的例子如图 9-10 所示。

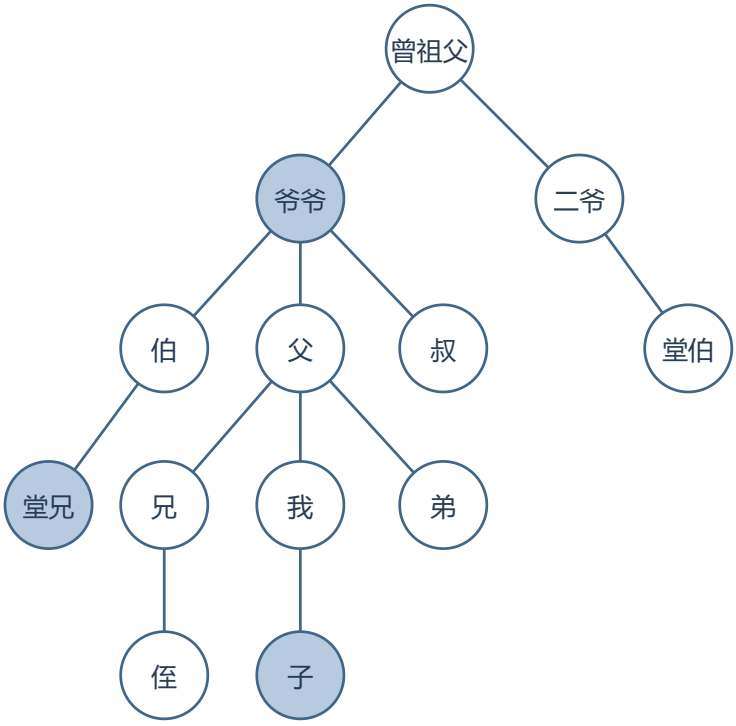



图9-10 族谱

如果把这个话题抽象到上面的族谱树上来看，他们所讨论最接近的直系长辈，就是这两个结点的最近公共祖先（Lowest Common Ancestor，LCA）。最近公共祖先是讨论树上两个结点的联系时最重要的一个概念。从图上来看，最近公共祖先是从树根到这两个结点的路径开始分叉的起点，在讨论树上的一些路径问题时，它是一个非常关键的路径分界点。

现在重新观察最上面的那棵树。寻找两个人最近公共祖先的时候会采取什么样的办法呢?可以利用“辈分”这一概念。例如想要找到“我”的堂兄和“我”的儿子的最近公共祖先是谁，需要将两人中的晚辈（也

就是“我”的儿子）往上追溯，直到和另一个人辈分相同（也就是“我”），进而再去考虑这两个人的公共祖先是谁。

把“辈分”这一概念对应到树上，可以发现之前讨论的人与人之间的“辈分”实质上就是一个结点在树上的深度。深度相同的点就是所谓的平辈，深度的大小关系决定了两个结点之间互为长辈或者晚辈。有了这个对应关系之后，不难想到如下一种做法（假设目前要求点 $u$ 和 $v$ 的最近公共祖先）：

（①首先找到两个结点中深度较深的结点（在树上的深度越深代表其越往下，也就是说它的辈分越小），不妨设深度较深的结点为 $u$，不停地将 $u$ 往上提，直到 $u$ 的深度和 $v$ 一样（也就是化为平辈再讨论接下来的问题）。

2. 同时将 $u$ 和 $v$ 向上提，直到μ和 $v$ 变成了同一个结点，这个结点就是要求的最近公共祖先。

根据以上思路，可以写出如下代码：

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e6 + 10;
int fa[N], d[N], n, m, rt;
vector<int> G[N];
void addedge(int u, int v) {
    G[u].push_back(v);
    G[v].push_back(u);
}
void dfs(int u) { // 先进行一遍dfs,计算出每个结点的深度
    for (int i = 0; i < G[u].size(); i++) {
        int v = G[u][i];
        if (v == fa[u])
            continue;
        fa[v] = u;       // 保存出每个点的父结点
        d[v] = d[u] + 1; // 子结点的深度比父结点多1
        dfs(v);
    }
}
int QueryLCA(int u, int v) { // 求出结点 u 和v 的 LCA
    if (d[u] < d[v])
        swap(u, v); // 找出深度最深的结点,保存在 u中
    while (d[u] != d[v])
        u = fa[u]; // 先不停地让u变成它的父结点,往上提直到同一层
    while (u != v) {
        u = fa[u];
        v = fa[v];
        // 之后再不断地同时往上提u和v,直到它俩相同之后,这个结点就是答案
    }
    return u;
}
int main() {
    scanf("%d%d%d", &n, &m, &rt);
    for (int i = 1; i < n; i++) {
        int u, v;
        scanf("%d%d ", &u, &v);
        addedge(u, v);
    }
    dfs(rt);
```

```
    for (int i = 1; i <= m; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        printf("%d\n", QueryLCA(u, v));
    }
}
```

这个程序正确地计算出了样例数据，可是并不能在规定时间以内通过所有的测试数据。分析这个算法，每次询问最近公共祖先的时候最坏的情况是需要爬完整棵树（想象一条链的情况），所以每次询问的复杂度是 $O(n)$。

有两种比较常见的求最近公共祖先的优化方法—**倍增**算法和Tarjan算法，从两个不同的角度优化这个过程。

解法 1（倍增算法）：观察之前的暴力算法，其瓶颈在于每次只能往上移动一个结点，答案是正确的但是效率太低。在高度差较大的情况下，每一次往上移动多个结点，达到加速的目的；当然，不能超过目标结点。如果需要快速地跳到一个点的某一级祖先，那么需要花费一定的空间去存储它的祖先。如果每个结点存储的信息过多，就容易导致空间超出限制的问题。所以必须要考虑一个时间与空间上都能较为平衡的算法方可解决这一问题。

现在要将一个结点往上移动 $h$ 的高度，可以考虑二进制分解，使得时间和空间上都令人满意。把 $h$ 拆分成二进制数，然后如果第 $i$ 位有 1，就向上提 2’层，这样时间复杂度被大大缩短，只有 $O(\log n)$ 了，同时也只需要记录其 2 的幂次级别的祖先，空间复杂度 $O(\log n)$ 也是可以接受的。如何根据这个思路求解？首先将问题细化：

1. 如何快速求出一个点的所有 $2^k$ 级祖先？

继续按照套路来思考这个问题：每个结点所需要的信息是由父结点计算而来的，还是在回溯的时候由子结点的信息计算的呢?考虑到要计算的是某一个结点的祖先，那么肯定是要用它的祖先结点的信息来计算。如何计算它的 $2^k$ 级祖先呢？使用暴力算法往上爬显然不够现实，但是可以考虑到这样一个事实：$2^{k-1}+2^{k-1}=2^k$。

这就启发我们可以用已有的 $2^{k-1}$ 级祖先的信息去推知 $2^k$ 级祖先的信息。具体来说如下：结点 $u$ 的 $2^k$ 级祖先是结点 $u$ 的 $2^{k-1}$ 级祖先的 $2^{k-1}$ 级祖先。对于根结点，把它的祖先当作 0；如果某个结点没有 $2^k$ 级祖先，则将其也设为 $0$。

这里的 $k$ 的上限就是 $\log(n)$，在本题中就是 18。

2. 如何优化往上提一个结点的过程?

可以发现，这就是要把一个数拆分成若干位二进制数的过程。可以直接从大到小地枚举二进制数完成这一拆分过程。先将比较深的结点提升到和比较浅的结点同一深度：$k$ 从 $\log(n)$ 开始往下枚举，依次尝试较深的结点往上面提升 $2^k$ 级，如果提升后的深度不会比另外一个结点小，则往上提升，直到两个结点深度一样为止。

接着使用同样的方法枚举 $k$，尝试比较这两个结点往上提升 $2^k$ 级后的祖先是否相同，如果不相同则同时往上提升 $2^k$ 级，直到其祖先相同为止。

根据以上的思路，可以完成如下代码：

```
#include <bits/stdc++.h>
const int N = 5e5 + 10;
using namespace std;
int n, m, Q, S;
int d[N], anc[N][20], dis[N];
vector<int> G[N];
void addedge(int u, int v) {
    G[u].push_back(v);
    G[v].push_back(u);
}
void dfs(int u, int fa) {
    for (int i = 0; i < G[u].size(); i++) {
        int v = G[u][i];
        if (v == fa)
            continue;
        d[v] = d[u] + 1;
        anc[v][0] = u; // 一个结点的2^0=1级祖先就是它的父结点
        dfs(v, u);
    }
}
void init() { // 预处理每个结点的 2^j 级祖先
    for (int j = 1; j <= 18; j++)
        for (int i = 1; i <= n; i++)
            anc[i][j] = anc[anc[i][j - 1]][j - 1];
}
int LCA(int u, int v) {
    if (d[u] < d[v])
        swap(u, v);
    for (int i = 18; i >= 0; i--)
        if (d[anc[u][i]] >= d[v])
            u = anc[u][i];
    // 从大到小枚举二进制位,将较深的结点往上提
    if (u == v)
        return u;
    for (int i = 18; i >= 0; i--)
        if (anc[u][i] != anc[v][i])
            u = anc[u][i], v = anc[v][i];
    // 然后用同样的办法同时提两个结点,直到相同为止
    return anc[u][0];
}
int main() {
    scanf("%d%d%d", &n, &m, &S);
    for (int i = 1; i < n; i++) {
        int u, v;
        scanf("%d%d ", &u, &v);
        addedge(u, v);
    }
    d[S] = 1;
    dfs(S, 0);
    init(); // 预处理
    while (m--) {
        int u, v;
        scanf("%d%d", &u, &v);
        printf("%d\n ", LCA(u, v));
    }
    return 0;
}
```

这里有个细节，最后求得的最近公共祖先是 `anc[u][0]`，而不是 $u$。这是因为for循环最终 $u$ 只枚举到了 0，而 $2^0=1$，所以找到的目标并不是两个点本身，而是它们的父结点。

由于 $n$ 的二进制下的位数有 `log(n)` 个，所以使用倍增算法预处理的时间复杂度是 $O(n\log n)$，单次询问的时间复杂度是 $O$（`log(n)`）的。

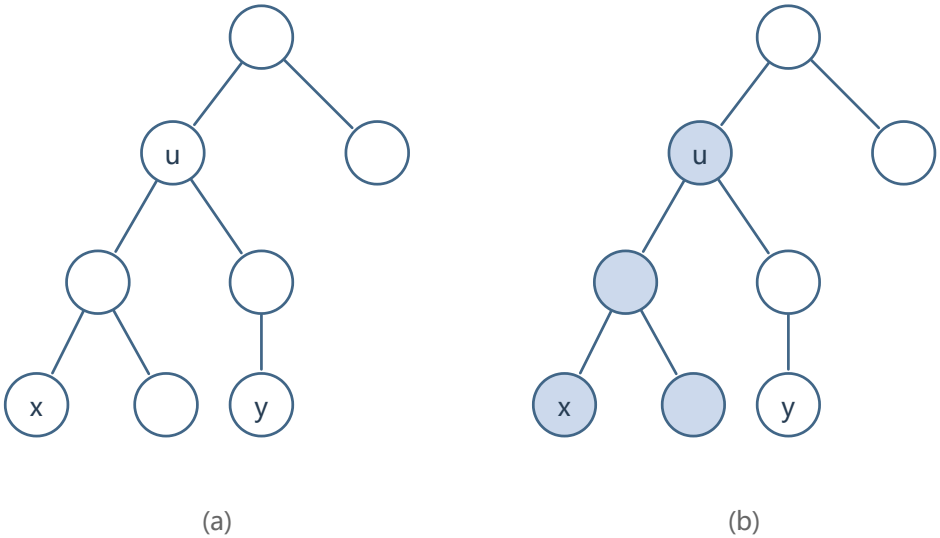
解法 2（Tarjan算法①）：上述倍增算法体现的是加速提升结点的过程，进而就可以做到快速查询。之前使用暴力算法求两个结点的最近公共祖先时，最坏情况下都要对整棵树进行一个完整的遍历，这无疑是非常低效的。但是与之相对的，可以采取另一种思路—能否让遍历整棵树的过程变得更加有价值一点，能够解决不止一次的询问，甚至是所有的询问呢？

这就是介绍**离散化**时提到过的离线思想：将所有的询问记录下来，最后仅通过一次统一的计算，即可求得所有要询问的点对的答案。为了实现这个目的，需要尝试挖掘具有相同的最近公共祖先的结点对所具有的性质。

观察图 9-11(a)：如果 $x$ 和 $y$ 分别位于结点 $u$ 的两个不同子树中，那么它们的最近公共祖先就是 $u$。这里要求它们属于不同分支；仅知道两点都在 $u$ 的子树内，还不足以确定 $u$ 就是最近公共祖先。

同时，这个问题也具有比较突出的递归特性：一个结点的子树一定是它所有祖先结点的子树的一个部分。也就是说，当递归处理完一个结点并且返回的时候，可以将已经访问的该子树的所有部分直接划归为它的父结点的一部分。依此类推，最后所有的结点都会成为根结点的子树。

> **注：** 1. Tarjan算法以它的发明者命名。除此之外，一类求解图的连通性问题的算法也叫作Tarjan算法。




(a) (b)

图9-11 求解LCA问题的Tarjan 算法

对于这一递归特性，需要考虑使用什么数据结构去维护。当回溯时，需要将该结点的所有子树和父结点的已经遍历过的其他子树进行合并。在《基础篇》中介绍过可以将两个集合给合并，以及查找一个集合的代表（在这里也就是并查集树的根结点）的一种数据结构——并查集。由此，整个算法的流程如下：

1. 读入所有的边并建立树，同时将查询也直接读入存储。如果要查询的边为 $u$ 和 $v$，就在 $u$ 的要查询结点中记录 $v$，并且在 $v$ 的要查询结点中记录 $u$，这可以使用STL的vector实现。

2. 从根结点开始DFS。
3. 在进入一个结点的时候，首先在并查集中将它的父亲设置为自己（也就是进去的时候先初始化）。如果它还有没有被访问过的子结点，就先进入子结点。
4. 在回到这个结点时，处理和这个结点相关的所有询问。遍历这个结点的待查询结点vector，如果里面有已经访问过的结点，那么该结点所在的并查集的代表元①就是它们的最近公共祖先。
5. 回溯离开该结点时，在并查集上将它所处的集合与它的父结点所处的集合合并。

如图 9-11(b) 所示，假设需要查询 $x$ 和 $y$ 的最近公共祖先。当结点 $u$ 已经处理好左边的子树，标有阴影的结点属于同一个并查集树，这些结点的并查集中的代表就是 $u$；处理 $y$ 结点时，发现对应的 $x$ 结点已经处理完毕，则它们的最近公共祖先就是 $u$。

根据上述说明，可以写出如下代码：

```
#include <bits/stdc++.h>
using namespace std;
const int N = 4000010;
vector<int> G[N];
vector<pair<int, int>> Q[N];
// Q[i]代表第i个结点的询问,存储另外一个结点的编号和询问编号
int fa[N], vis[N];
// fa是并查集的父亲,vis代表一个元素是否被访问
int ans[N]; // 记录答案
int n, m, p;
inline void addedge(int u, int v) {
    G[u].push_back(v);
    G[v].push_back(u);
}
inline void addquery(int u, int v, int ID) {
    Q[u].push_back(make_pair(v, ID));
    Q[v].push_back(make_pair(u, ID));
}
inline int find(int x) { // 并查集查询
    return x == fa[x] ? x : fa[x] = find(fa[x]);
}
void dfs(int u) {
    fa[u] = u;
    vis[u] = 1; // 在并查集中初始化这个结点,并且将其标记为已访问过
    for (int i = 0; i < G[u].size(); i++) {
        int v = G[u][i];
        if (vis[v])
            continue;
        dfs(v);
        fa[v] = u; // 这里必须将子结点合并到它的父亲,也就是当前u上
        // 因为修改前fa[u]和fa[v]还是指向自己,所以可以直接这么写
    }
    for (int i = 0; i < Q[u].size(); i++) { // 处理询问
        int v = Q[u][i].first;
        if (!vis[v])
            continue; // 如果询问的另一端还没访问,就无法计算答案
        ans[Q[u][i].second] = find(v);
    }
}
```

```
int main() {
    scanf("%d%d%d", &n, &m, &p);
    for (int i = 1; i < n; i++) {
        int x, y;
        scanf("%d%d", &x, &y);
        addedge(x, y);
    }
    for (int i = 1; i <= m; i++) {
        int x, y;
        scanf("%d%d", &x, &y); // 读入查询并且记录
        addquery(x, y, i);
    }
    dfs(p); // 以p为根结点开始 DFS
    for (int i = 1; i <= m; i++)
        printf("%d%c", ans[i], i == m ? '\n' : ' ');
    return 0;
}
```

> **注：** 1. 代表元，即并查集树的根结点。

这是一个离线算法，因为这里的并查集结构较为特殊，因此即使是只使用路径压缩优化，单次find操作的均摊复杂度也是 $O(1)$，因此该算法的时间复杂度为 $O(n+m)$

### 例 9-6 最大流量(洛谷P3128)

给出一棵 $n(n\leqslant 50000)$ 个结点的树，需要进行 $m(m\leqslant 10^5)$ 次修改，每次给定结点编号 $s$ 和 $t$，把 $s$ 到 $t$ 的路径上的点权增加 $1$ 问 $:m$ 次操作后最大点权的值是多少?

**分析：** 如果是在线性表中，多次操作从第 $i$ 个到第 $j$ 个元素都增加 1，最后统计每个元素的值，可以使用差分思想——维护差分序列，在区间开头+1，区间结束后-1，最后求**前缀和**还原序列。在树上也可以使用同样的操作，这种操作被称为树上差分。

如图 9-12 所示，令 $LCA(A,B)$ 表示结点 $A$ 和结点 $B$ 的**最近公共祖先** $fa(u)$ 为结点 $u$ 的父结点，数组 $p$ 就是**差分数组** ,$p[u]-p[fa(u)]$ 表示结点 $u$ 的点权。

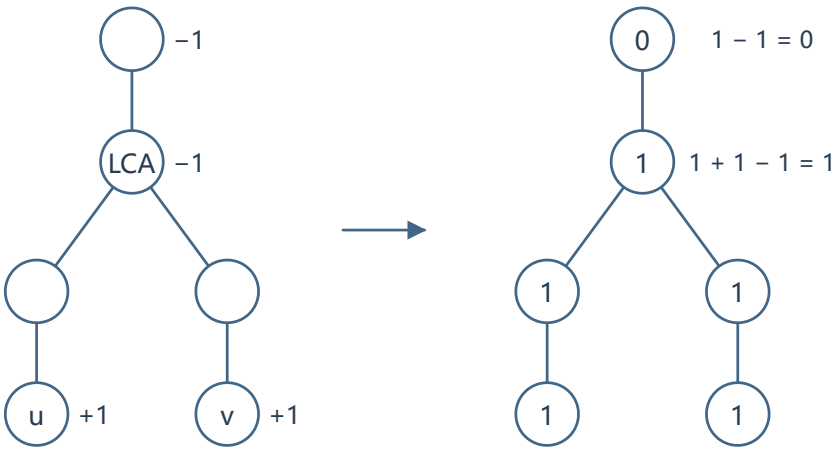



图9-12树上差分

假设从 $u$ 到 $v$ 的路径上每个元素都要增加 1，那么这条路径是从 $u$ 到 $LCA(u,v)$ 再到 $v$，所以可以把从 $u$ 到 $LCA(u,v)$ 与 $LCA(u,v)$ 到 v 两条路径都加 $1$。由于结点 $LCA(u,v)$ 重复计算了一次，所以这个结点的点权还要减 1。

因此，只需要将 $p[u]$ 和 $p[v]$ 增加 1 ，$p[LCA(u,v)]$ 和 $p[fa(LCA(u,v))$ ）］各减小 1 即可。使用**倍增**求解LCA，最后遍历整棵树，根据差分数组计算前缀和，还原出每个结点的点权并统计答案。

## 9.4 树的重链剖分

树的**重链剖分**作为一种进一步划分树的结构的方式，具有良好的性质。它将树上的信息划分到链上，使得大部分处理序列问题的数据结构得以在树上应用。本节将探寻如何对树做重链剖分，以及它的应用。

### 例 9-7 重链剖分(洛谷P3384)

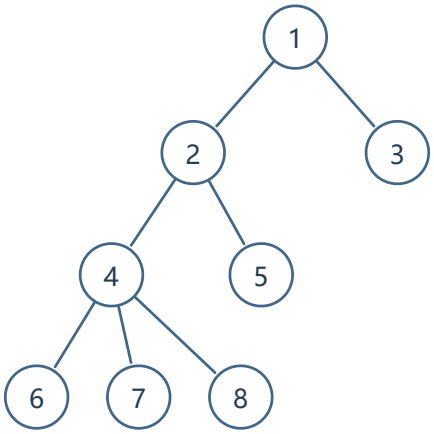
已知一棵包含 $n$ 个结点的树，$r$ 是树的根结点，每个结点上包含一个数值，需要支持以下操作，共 $m$ 次。

1. $1\ x\ y\ z$ 表示将树从结点 $x$ 到 $y$ 的**最短路径**上所有结点的值都加上 $z_c$
2. $2\ x\ y$：表示求树从结点 $x$ 到 $y$ 的最短路径上所有结点的值之和。
3. $3\ x\ z$：表示将以 $x$ 为根结点的子树内所有结点值都加上 $z$。
4. $4\ x$：表示求以 $x$ 为根结点的子树内所有结点值之和。

数据范围满足 $1 \leqslant n,m \leqslant 10^5$，答案对大数 $p$ 取模。

**分析：**先考察前两个操作，即对 $x$ 到 $y$ 的路径上的结点进行的加法和查询操作。这看起来像是**线段树**可以支持的操作，区别在于本题的修改查询应用于树上。如果存在某种方式，把树的结构对应到一个序列上，那么可以使用线段树等维护序列信息的数据结构来处理这个问题。

由此，引入树的DFS序的概念。假设维护一个序列，初始为空。从树的某个结点开始对树进行DFS，每到达一个新的结点，就把这个结点的编号添加到序列的尾部。当DFS结束时，将得到一个长度为 $n$ 的序列，树上的每个结点都在这个序列里出现了一次。例如，如图 9-13 所示的这棵树，如果从 1 开始，按照从左到右的顺序访问孩子结点进行DFS，那么一种可能的DFS序将是 1，2，4，6，7，8，5，3。



根据结点编号的DFS序，可以得到对应点权按照结点DFS序排列的序列 $w$ 尝试使用线段树维护这个点权序列。如果想把 1

图 9-13 树的DFS序

到 6 号结点的路径上的点权都加上 $z$，注意到这条路径上的结点在DFS序上恰好是前 4 个，于是只需要对序列 $w$ 进行一次区间[1，4]加上 $z$ 的操作即可。然而，如果希望对 3 到 8 号结点路径上的点权做修改，则需要修改[8，8]、[1，3]和[6，6]3 个区间。在极端情况下，修改的区间数可以达到 $O(n)$ 个。这样做显然效率极低。考虑到不同的DFS方式可以生成不同的DFS序，是否存在一个DFS策略，使得在最坏情况下被修改的区间数也比较少？答案是肯定的。为了介绍这一策略，首先介绍**重链剖分**。

对于树上任意一个结点 $u$，设 $size_u$ 表示 $u$ 所在的子树大小（包括 $u$ 自身），在 $u$ 的众多孩子中，设 `wcu(weight\_child)` 为子树最大的那个孩子，称为 $u$ 的重孩子。这两个量都可以通过一次DFS（称为 dfs1）求出：

```
int size[maxn], wc[maxn];
void dfs1(int u) {
    size[u] = 1;
    for (int i = 0; i < child[u].size(); ++i) {
        int v = child[u][i];
        dfs1(v);
        size[u] += size[v];
        if (size[v] > size[wc[u]])
            wc[u] = v;
    }
}
```

接下来，假设按照先访问重孩子，再访问轻孩子的策略，从根结点起重新进行一遍DFS，记录此时的DFS序。不难发现，因为先访问了重孩子，所以一个结点 $u$ 和它的重孩子 $wc_u$ 在DFS序上是连续的。

如果一个结点是它父结点的重孩子，那么称这个结点是一个重结点，否则称它是轻结点（规定根为轻结点）。可以想象，从任何一个非重结点起，向下访问它的重孩子，再继续访问重孩子的重孩子……一直到叶子结点，构成了一条树上的链。这条链上的结点在DFS序上是连续的一段，并且除了第 1 个结点以外，其余的结点都是重结点。称这样的链为重链。不属于任何一条重链的边一定是一个轻结点到它父结点的边，这种边被称为轻边。至此，通过使用重链和轻边，完成了对树的划分。

例如，图 9-14 中实线连接的 1—2—6—9 是一条重链，5—8 也是一条重链，虚线为轻边。任意两条不同的重链之间至少由一条轻边相连。因此，从某个结点到根的路径上，经过的重链条数等于轻边数加 **1**。

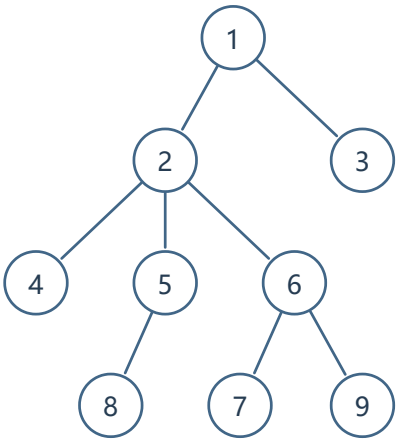



图9-14 树的划分

接下来，证明一个对复杂度分析至关重要的结论。

定理 1：对树上任何一个结点，它到根的路径上的重链条数不会超过logn，轻边条数也不会超过logn。

**证明：**设 $u-v$ 是一条轻边，$u$ 是父结点，$w$ 是 $u$ 的重孩子。由重孩子的定义有 $\text{size}_w \geq \text{size}_v$，所以 $\text{size}_u > \text{size}_v + \text{size}_w \geq 2\text{size}_v$。从结点向根移动时，每经过一条轻边，所在子树的大小至少翻倍。因此一条根路径至多经过 $O(\log n)$ 条轻边，也至多经过 $O(\log n)$ 条重链。

经过至多logn次翻倍后，树的大小就会变成 $n$，也就走到了根。因此证明了路径上至多有logn条轻边，即只有logn条重链。

这个定理有什么用呢?对于一个从结点 $x$ 到 $y$ 的修改，设 $w$ 是 $x$ 和 $y$ 的LCA，把修改的路径拆成 $x-w$ 和 $y-w$ 两条。这两条路径分别是 $x$ 到根和 $y$ 到根的路径的一部分，因此也必然分别至多经过logn条重链和轻边。而重链上的结点在DFS序里是连续的，相当于只需要做 $O(\log n)$ 次区间修改操作，就能完成一次对树上路径的修改。

具体而言，对于一条重链上的每个结点，都记录它的链头 `top`，也就是那个非重结点；对于一个非重结点，设链头是它自身。设 $dfn_u$ 表示结点 $u$ 在DFS序里的下标，那么对于一个 $x-w$ 的修改，从 $x$ 起，修改它到当前链头 `top` 的区间 $.dfn_x, dfn_{top_x}$ ]的点权，然后令 $x$ 为链头的父结点，重复操作直到 $w$ 在当前的链上，此时只要修改 $w-x$ 这段区间 $[d\ fi_x, d\ fi_{n_w}]$ 即可。

例如，假设树上有一条路径如图 9-15 所示，则重链 1—2—3 的链头为 1，4 的链头为 4，5 的链头为 5，6 的链头也为 5，7 的链头为 **7**




图9-15 树上的路径

要修改 2—7 这条路径上的点权，则先修改结点 $dfn_7, dfn_7]$ 的点权，再修改 $dfn_5, dfn_6]$ 的点权，然后再修改结点 $[d\ fi_4, dfn_4]$ 的点权，最后修改[ $.dfn_2, dfn_3]$ 的点权。

求数组 `dfn` 和每个结点对应的链头可以再用一次DFS（称为dfs2）完成。

```
int vistime;
int dfn[maxn], top[maxn];
void dfs2(int u, int Top) {
    top[u] = Top;
    dfn[u] = ++vistime;
    if (wc[u] != 0)
        dfs2(wc[u], Top);
    for (int i = 0; i < child[u].size(); ++i)
        if (child[u][i] != wc[u]) {
            dfs2(child[u][i], child[u][i]);
        }
}
```

现在已经求出了所有结点的数组 `dfn` 和 $top$。下一个问题是如何在没有求出LCA的情况下精确修改到两结点的LCA，不会多做或少做修改。

沿用**倍增**法求LCA的思路。设 $dep_u$ 表示结点 $u$ 的深度，假设在修改 $x-y$ 这条路径的点权。

考察 $top_x$ 和 $top_y$，分两种情况：

1. $top_x=top_y$，说明两点在同一条重链上，则 $x$ 和 $y$ 深度较小的一定是 $\mathbf{LCA}$。直接修改 $[\min(dfh_x,dfh_y),\max(dfh_x,dfh_y)$ ]即可。
2. $top_x\neq top_y$，此时比较 $top_x$ 和 $top_y$ 的深度，若 $dep_{top_x}>dep_{top_y}$，则 $x{-}top_x$ 上不可能有两个结点的 $\mathbf{LCA}$。修改 $[dfn_{top_x},dfn_x]$，然后令 $x{=}fa_{top_x}$ 修改下一条链。如果 $dep_{top_x}\leqslant dep_{top_y}$，则交换 $x$ 和 $y$ 后再进行上述操作。

对于树上两结点间路径的查询问题，可以完全仿照上述思路，把修改操作改为查询操作即可。

至此。已经解决了如何完成树链上的点权加法和查询的问题。现在来考察子树点权加法和查询：不难发现，因为DFS时一定是先访问完一个结点的子树里所有的结点，再离开这个结点，所以任意结点的子树对应的DFS序都是连续的，$u$ 的子树对应的 $dfn$ 值恰好就是 $dfn_u,dfn_u+size_u-1]$。只需要对这个区间做修改和查询即可。

最后一个问题是，在线段树建树时，$[x,x]$ 这一区间对应的权值应该是DFS序上第 $x$ 个结点的权值，没有记录这个信息，但记录了 $dfn_y=x$ 表示结点 $y$ 是DFS序里第 $x$ 个。只要对应记录一个相反的数组 $rdfn_x=y$ 表示DFS序第 $x$ 个为 $y$ 即可。

考虑时间复杂度：在对于每个查询或询问，进行了 $O(\log n)$ 次查询/询问，每次的时间复杂度为 $O(\log n)$，而DFS 的时间复杂度为 $O(n)$，于是总时间复杂度为 $O(n)+O(m\log^2 n)$。

完整代码如下：

```
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;
#define ll long long
#define maxn 100005
int n, m, r, p, vistime;
int a[maxn], fa[maxn], wc[maxn], sz[maxn], top[maxn], dfn[maxn], rdfn[maxn], dep[maxn];
```

```
vector<int> e[maxn];
void dfs1(int u, int f) {
    fa[u] = f;
    sz[u] = 1;
    dep[u] = dep[f] + 1;
    for (int i = 0; i < e[u].size(); ++i)
        if (e[u][i] != f) {
            int v = e[u][i];
            dfs1(v, u);
            sz[u] += sz[v];
            if (sz[v] > sz[wc[u]])
                wc[u] = v; // 确定u的重孩子
        }
}
void dfs2(int u, int Top) {
    dfn[u] = ++vistime;
    rdfn[vistime] = u; // 记录 rdfn,表示 DFS 序第vistime 个结点
    // 为u
    top[u] = Top;
    if (wc[u] != 0) {
        dfs2(wc[u], Top); // 先递归重孩子
        for (int i = 0; i < e[u].size(); ++i)
            if (e[u][i] != fa[u] && e[u][i] != wc[u]) {
                dfs2(e[u][i], e[u][i]); // 再递归轻孩子
            }
    }
}
ll w[maxn * 4];
void pushup(int u) { w[u] = (w[u * 2] + w[u * 2 + 1]) % p; }
void build(int u, int L, int R) {
    if (L == R) {
        w[u] = a[rdfn[L]]; // 到达叶结点,该区间的点权是DFS序上第L 个结点的权值,
        // 即 a[rdfn[L]]
        return;
    }
    int M = (L + R) / 2;
    build(u * 2, L, M);
    build(u * 2 + 1, M + 1, R);
    pushup(u);
}
bool InRange(int L, int R, int l, int r) { return (l <= L) && (R <= r); }
bool OutofRange(int L, int R, int l, int r) { return (L > r) || (R < l); }
ll lzy[maxn * 4];
void maketag(int u, int len, ll x) {
    lzy[u] += x;
    w[u] += len * x % p;
    lzy[u] %= p;
    w[u] %= p;
}
void pushdown(int u, int L, int R) {
    int M = (L + R) / 2;
    maketag(u * 2, M - L + 1, lzy[u]);
    maketag(u * 2 + 1, R - M, lzy[u]);
    lzy[u] = 0;
}
```

```
ll query(int u, int L, int R, int l, int r) {
    if (InRange(L, R, l, r)) {
        return w[u];
    } else if (!OutofRange(L, R, l, r)) {
        int M = (L + R) / 2;
        pushdown(u, L, R);
        return (query(u * 2, L, M, l, r) + query(u * 2 + 1, M + 1, R, l, r)) % p;
    } else
        return 0;
}
void update(int u, int L, int R, int l, int r, ll x) {
    if (InRange(L, R, l, r)) {
        maketag(u, R - L + 1, x);
    } else if (!OutofRange(L, R, l, r)) {
        int M = (L + R) / 2;
        pushdown(u, L, R);
        update(u * 2, L, M, l, r, x);
        update(u * 2 + 1, M + 1, R, l, r, x);
        pushup(u);
    }
}
void upd(int x, int y, int z) {
    while (top[x] != top[y]) {
        if (dep[top[x]] < dep[top[y]])
            swap(x, y);                          // 保证 x 的链头深度较大
        update(1, 1, n, dfn[top[x]], dfn[x], z); // 对[dfn[top[x]], dfn[x]]
        // 这段区间做修改
        x = fa[top[x]];
    }
    update(1, 1, n, min(dfn[x], dfn[y]), max(dfn[x], dfn[y]), z); // 此时x和y 在
    // 同一条链上
}
int qry(int x, int y) {
    ll ans = 0;
    while (top[x] != top[y]) {
        if (dep[top[x]] < dep[top[y]])
            swap(x, y);
        ans += query(1, 1, n, dfn[top[x]], dfn[x]);
        ans %= p;
        x = fa[top[x]];
    }
    return (ans + query(1, 1, n, min(dfn[x], dfn[y]), max(dfn[x], dfn[y]))) % p;
}
int main() {
    scanf("%d %d %d %d", &n, &m, &r, &p);
    for (int i = 1; i <= n; ++i)
        scanf("%d", a + i);
    for (int u, v, i = 1; i < n; ++i) {
        scanf("%d %d", &u, &v);
        e[u].push_back(v);
        e[v].push_back(u);
    }
    dfs1(r, 0);
    dfs2(r, 0);
    build(1, 1, n);
    for (int o, x, y, z; m; --m) {
        scanf("%d", &o);
```

```
        if (o == 1) {
            scanf("%d %d %d", &x, &y, &z);
            upd(x, y, z);
        } else if (o == 2) {
            scanf("%d %d", &x, &y);
            printf("%d\n", qry(x, y));
        } else if (o == 3) {
            scanf("%d %d", &x, &y);
            update(1, 1, n, dfn[x], dfn[x] + sz[x] - 1, y);
        } else {
            scanf("%d", &x);
            printf("%lld\n", query(1, 1, n, dfn[x], dfn[x] + sz[x] - 1) % p);
        }
    }
}
```

### 例 9-8 最近公共祖先(洛谷P3379,题面同例9-5)

**分析**：前面已经学习了如何使用**倍增**法求出两结点的LCA，那么能否使用剖分的算法来求解？事实上，在求解 例 9-7 的过程中，已经介绍了如何在一棵树中使用**重链剖分**的方法求出两结点的LCA：不断地把两结点中链头深度较大的结点跳到链头的父结点，直到两结点的链头相同，此时深度较小的结点就是LCA。

考虑这样做的复杂度：一个结点向上跳 $O(\log n)$ 次就能跳到根，所以总的跳点次数不超过 $O(\log n)$。每次跳到链头后没有额外的操作，因此使用剖分求LCA的时间复杂度也是 $O(q \log n)$。

需要说明的是，显然logn条重链轻边是一个并不紧的上界，在最坏情况下，这一上界也有着大约 $1/2$ 的常数。因此，使用这种方法求LCA的常数远小于倍增法求LCA的常数，同时也可以发现，去掉**线段树**部分的纯剖分代码事实上并不长，甚至比倍增求LCA还要短一些。

这里只给出查询LCA的代码，剖分的部分与 例 9-7 完全相同。

```
int qry(int x, int y) {
    ll ans = 0;
    while (top[x] != top[y]) {
        if (dep[top[x]] < dep[top[y]])
            swap(x, y);
        x = fa[top[x]];
    }
    return (dep[x] < dep[y]) ? x : y;
}
```

### 例 9-9 种草(洛谷P3038,USACO2011年12月赛)

给定一棵 $n$ 个结点的树，边有边权。有 $m$ 次操作，包括如下两种：

1. 给定 $x$ 和 $y$，将 $x$ 到 $y$ 路径上的边的边权增加 $1$。
2. 查询某一条边的权值。

**分析**：之前介绍过树上的点权，但如何维护边权呢？注意到树上有 $n$ 个结点，只有 $n-1$ 条边。一个自然的想法是将边和点绑定，把边权对应到某个结点的点权上，将问题转化为对点权的修改。那么，如何对应呢？

对于一条边，有两种直观的对应方式，分别是把边权对应到两端点中深度较深的结点，和对应到深度较浅的结点。但是后者的对应会出问题，因为当一个结点有多个孩子时，会有多条边对应到该结点。而一个结点只有一个父结点，所以只会被一条边对应。因此，把边权对应到深度较深的结点是可行的。

于是，一次修改操作就变成了对两结点间路径上点权的修改，但是不修改两结点间的LCA（因为LCA对应的边不在 $x$ 到 $y$ 的路径上）。有两种方案：第 1 种是照常修改路径上所有结点的点权，然后把LCA的点权加回去；第 2 种是在修改时直接略过LCA，只需要在 $x$ 和 $y$ 的链头相等时不修改 `min(dfnx,dfn)` 即可。给出第 2 种方案的代码。

```
void upd(int x, int y, int z) {
    while (top[x] != top[y]) {
        if (dep[top[x]] < dep[top[y]])
            swap(x, y);                          // 保证 x 的链头深度较大
        update(1, 1, n, dfn[top[x]], dfn[x], z); // 对[dfn[top[x]],dfn[x]]
        // 这段区间做修改
        x = fa[top[x]];
    }
    if (x != y) // 注意x = y时没有要修改的区间
        update(1, 1, n, min(dfn[x], dfn[y]) + 1, max(dfn[x], dfn[y]), z);
}
```

### 例 9-10 旅游(洛谷P3976,2015年天津市队选拔)

给定一棵有 $n$ 个结点的树，每个结点初始有一个权值。然后有 $q$ 次询问，每次给出两个结点 $u$ 和 $v$，要求在 $u$ 到 $v$ 的单向简单路径上找到两点 $x$ 和 $y$，满足 $x$ 在 $y$ 的前面（$x$ 更接近 $u$）且 $x$ 的权值减去 $y$ 的权值最大。然后再给 $u$ 到 $v$ 的路径上的点权都加上 $w$。

**数据范围：** $1 \leqslant n, q \leqslant 5 \times 10^4$。

**分析：重链剖分**最大的作用就是把一棵树的问题转化成一个序列（链）的问题，所以当感觉对树的问题无从入手时，不妨考虑当树退化成一条链时如何解决问题。

在树是一条链时，问题是这样的：给定一个序列 $a$，每次给出序列中一个区间 $[l,r]$，要求支持区间加法和找到区间内的两个数 $x$ 和 $y$，满足 $x$ 在 $y$ 的前面或后面，且 $a_x - a_y$ 最大。假设要求 $x$ 在 $y$ 前面，尝试用**线段树**去解决这个问题：假设对于线段树上一个结点 $[L,R]$，要维护 $[L,R]$ 内“前面的数”减去“后面的数”的最大值。

如果这两个数均出现在左子区间[$L$，mid]，那么可以从左孩子直接转移；如果这两个数均出现在右子区间[mid+1，$R$]，那么可以通过右孩子直接转移：此外，就只有“左子区间的一个数减去右子区间的一个数“的情况了。因为要求差值最大，显然是左子区间的最大值减去右子区间的最小值。于是维护线段树结点对应区间的最大值 $\max w_u$、最小值 $\min w_u$ 和“前-后”差值的最大值minus，假设某结点 $u$ 的左右子结点分别是 $v$ 和 $w$，则转移为：

$$\text{minus}_u = \max(\text{minus}_v, \text{minus}_w, \text{maxw}_v - \text{minw}_w)$$

区间的最大、最小值按线段树正常维护即可。对于要求“后面-前面”的最大值，也可以类似地维护，只需要把 $\max w_w - \min w_v$ 改为 $\max w_w - \min w_v$，即用右孩子的最大值减去左孩子的最小值即可。

对于区间加操作，被操作覆盖的区间因为最大值和最小值同时加上某个数，所以差值不会改变，只需要修改最大值和最小值，然后一路pushup即可。

至此，解决了这个问题的链版本，考虑把它拓展到树中。对于树中一条从 $u$ 到 $v$ 的路径，DFS序是剖分后维护数列的数组下标。解决数列中“前-后”的问题就相当于解决链中类似“DFS序较小-DFS序较大”的问题。但是结点的DFS序不一定是单调递增的，似乎很难通过DFS序确定一个“前后”顺序。

考虑把 $u$ 到 $v$ 的链拆成 $u$ 到LCA、LCA到 $v$ 两段。发现对于树中有“祖先一后代”关系的两个结点，祖先的DFS序一定比后代的DFS序小。这是因为DFS时一定是先访问祖先，再在祖先的子树里访问到后代，所以 $u$ 到LCA、LCA到 $v$ 的路径上的点DFS序都是单调的。对于 $u$ 到LCA这条路径上每条DFS序连续的链，相当于求“后-前”这样的差值的最大值；对于LCA到这条路径上每条DFS序连续的链，相当于求“前-后”这样的差值的最大值。

首先通过跳链的方式把 $u$ 到LCA和LCA到 $v$ 的所有链都找出来，然后对每条链求出“前-后”或“后-前”的最大值。最后考虑合并这些链的答案：把所有链按照离 $u$ 的距离排好序（事实上提取链的时候天然是有序的，只需要把LCA到 $v$ 提取出的链翻转拼到 $u$ 到LCA的链后面，无须单独排序），再按顺序遍历。维护已遍历的链上所有点权的最大值，这些链相当于序列问题上线段树的“左孩子”。当遍历到一个新的链时，答案可能被新链中维护的“前-后”（或“后-前”）的最大值更新，也可能被已遍历的链中最大值-当前链最小值更新。

在以下代码中展示如何取出所有的链和如何维护答案。

```
vector<pair<int, int>> rec[2]; // rec[1] 是u到LCA的链,rec[0]是LCA到v的链
void split(int u, int v) {     // 这是提取链的函数
    rec[0].clear();
    rec[1].clear();
    int x = 1, y = 0;
    while (top[u] != top[v]) {
        if (depth[top[u]] < depth[top[v]]) {
            swap(u, v);
            swap(x, y); // 保证0/1 与原本的u/v总是对应的
        }
        rec[x].push_back({dfn[top[u]], dfn[u]}); // 取出 u 到链头的那条链
        u = fa[top[u]];
    }
    if (depth[u] > depth[v]) { // 此时 u 和v 在同一条链上
        swap(u, v);
    } else {
        swap(x, y);
    }
    rec[x].push_back({dfn[u], dfn[v]});
    reverse(rec[0].begin(), rec[0].end()); // 把v到LCA 路径上的链翻转
}
void solve(int u, int v) {
    split(u, v); // 先把链提取出来
    int premax = 0, ans = 0;
    for (int i = 1; i >= 0; --i) { // 按这一顺序能按离 u 的距离遍历从 u 到 v 的所有链
        for (int j = 0; j < rec[i].size(); ++j) {
            int l = rec[i][j].first,
                r = rec[i][j].second; // 找到这条链在线段树维护的数列上对
            // 应的区间
            ans = max({ans, premax - querymin(l, r), queryans(l, r, i)});
            premax = max(premax, querymax(l, r));
            // querymin(1,r) 和querymax(1,r)表示用线段树查询对应区间的最小/最大值
```

```
            // queryans(1,r,0)表示查询对应区间"前-后"的最大值,用在LCA到v的链上
            // queryans(1,r,1)表示查询对应区间"后-前"的最大值,用在u到LCA的链上
        }
    }
}
```

## 9.5 课后习题与实验

### 习题 1 道路修建(洛谷P2052,NOI2011)

一共有 $n$ 个国家，他们打算在各个国家之间建设双向道路使得国家之间连通，但只愿意修建恰好 $n-1$ 条双向道路。每条道路的修建都要付出一定的费用，这个费用等于道路长度乘以道路两端的国家个数之差的绝对值。给定这些道路的连接的国家编号 $a$ 和 $b$，以及这条道路的长 $a_ib_i$ 度 $c_i$，计算出所需要的费用。

**数据范围：** $1 \leqslant a_i, b_i \leqslant n, 0 \leqslant c_i \leqslant 10^6, 2 \leqslant n \leqslant 10^6$。

### 习题 2 树的重心(洛谷P5666,CSP-S2019)

给出了一个大小为 $n(n \leqslant 3 \times 10^5)$ 的树 $S$，树中结点从 $1 \sim n$ 编号。求出 $S$ 单独删去每条边后，分裂出的两个子树的重心编号和之和。即：

$$\sum_{(u,v)\in E}\left(\sum_{x\text{是}S'_u\text{的重心}} x+\sum_{y\text{是}S'_v\text{的重心}} y\right)$$

上式中，$E$ 表示树 $S$ 的边集，（$u$，$v$）表示一条连接 $u$ 号结点和 v 号结点的边。 $S'_u$ 与 $S'_v$ 分别表示树 $S$ 删去边 $(u,v)$ 后，$u$ 号结点与 $v$ 号结点所在的被分裂出的子树。

### 习题 3 小猪爬树(洛谷P5588,By_QAQ)

给定 $n(n \leqslant 10^6)$ 个结点的树 $T(V,E)$，第 $i$ 个结点的颜色为 $w_i$，保证有 $1 \leqslant w_i \leqslant n$。

对于 $1 \leqslant i \leqslant n$，分别输出有多少对点对 $(u,v)$，满足 $u<v$，且恰好经过所有颜色为 $i$ 的结点；对于结点颜色不为 $i$ 的其他结点，经过或不经过均可。

### 习题 4 核心城市(洛谷P5536,Byxht37)

有 $n$ 座城市 ,$n-1$ 条长度为 $1$ 的道路，每条道路连接两座城市，且任意两座城市都能通过若干条道路相互到达，显然，城市和道路形成了一棵树。现决定将 $k$ 座城市定为核心城市，这 $k$ 座城市需要满足以下两个条件：

1. 这 $k$ 座城市可以通过道路，在不经过其他城市的情况下两两相互到达。
2. 定义某个非核心城市与这 $k$ 座核心城市的距离为该城市与各核心城市的距离的最小值。

求所有非核心城市中，与核心城市的距离最大的城市，其与核心城市的距离的最小值。

### 习题 5 访问(洛谷P5836,USACO2019年12月赛)

给出一棵大小为 $n\left(n \leqslant 10^5\right)$ 的树，每个结点不是 $H$ 类型就是 $G$ 类型。有 $m(m \leqslant 2 \times 10^3)$ 条路径，对于每条路径，两个端点以 $a$ 和 $b$ 表示，并且给出字母 $H$ 或者 G。请问在这条路径上是否存在 $H$ 或者 $G$ 类型的结点。

### 习题 6 联合权值(洛谷P1351,NOIP2014提高组)

给出一棵大小为 $n(n\leqslant 2\times 10^5)$ 的树，第 $i$ 个结点有点权 $w_i$，所有的边长度均为 $1$。任何一对距离为 $2$ 的结点 $u$ 和 $v$ 会产生 $w_u\times w_v$ 的联合权值。请问：树上所有产生联合权值的有序点对中，联合权值最大的是多少?所有联合权值之和是多少?

### 习题 7 逃离(洛谷P4408,NOI2003)

给出一棵大小为 $n(3\leqslant n\leqslant 2\times 10^5)$ 的树，边带权。令 $d(x,y)$ 表示结点 $x$ 到结点 $y$ 的距离。请找到 $3$ 个点 $A$，$B$，$C$，使得 $d(A,B)+d(B,C)$ （当 $d(A,C)>d(B,C)$ 时），或 $d(A,C)+d(A,B)$ （当 $d(A,C)<d(B,C)$ 时）最大。

### 习题 8 仓鼠找糖(洛谷P3398,Byfjzzq2002)

给出一棵大小为 $n\left(n\leqslant 10^5\right)$ 的树，进行 $q(q\leqslant 10^5)$ 次询问。每次询问给出 $a,b,c,d4$ 个结点，请问从 $a$ 到 $b$ 的路径和从 $c$ 到 $d$ 的路径是否有交叉。

### 习题 9 运输计划(洛谷P2680,NOIP2015提高组)

给出一个大小为 $n(n\leqslant 3\times 10^5)$ 的树，边带权。还有 $m(m\leqslant 3\times 10^5)$ 条路径，每个路径用两个端点 $a$ 和 $b$ 表示。可以将树中一条边的边权改为 $0$，使这些路径中最长的一条路径长度最小化，并输出这个值。

### 习题 10 天天跑步(洛谷P1600,NOIP2016提高组)

给出一棵大小为 $n(n\leqslant 3\times 10^5)$ 的树，第 $i$ 个结点有点权 $w_i$。有 $m(m\leqslant 3\times 10^5)$ 条路径，每条路径的起点和终点用 $a$ 和 $b$ 表示。对于每个结点 $k$，请问有多少条路径从起点开始的第 $w_i$ 个结点就是结点 $k$ 。

### 习题 11 使用重链剖分的思想完成例9-6的题目

### 习题 12 树上操作(洛谷P3178,2015年河南省队选拔)

给定一棵 $n$ 个结点的树，点有点权，进行 $q$ 次操作：

1. 把某个结点的点权增加 $a_c$
2. 把某个结点的子树中所有结点的点权都增加 $a$。
3. 询问某个结点到根的路径点权和。

**数据范围：** $1\leqslant n,q\leqslant 10^5$。

### 习题 13 树(洛谷P4092,2016年天津市河北省队联合选拔)

给定一棵 $n$ 个结点的树，$1$ 是树根。初始时所有结点都是白点。有 $q$ 次操作，每次要么选择一个结点染黑，要么查询某个结点的祖先中离该结点最近（深度最大）的黑点。

**数据范围：** $1\leqslant n,q\leqslant 10^5$

### 习题 14 QTree3(洛谷P4116)

给定一棵 $n$ 个结点的树，$1$ 是树根。初始时所有结点都是白点。有 $q$ 次操作，每次要么选择一个结点染黑，要么查询某个结点的祖先中离该结点最远（深度最小）的黑点。

**数据范围：** $1 \leqslant n, q \leqslant 10^5$。

**提示：**这道题可以用和习题 **14** 类似的方法完成，但有一个更简单的写法。因为题目询问的是深度最小的结点，由此导致习题 **14** 中必需的**线段树**在本题里可以省去，在剖分后用一个码量更小的工具维护。

### 习题 15 旅行(洛谷P3313,2014年山东省队选拔)

给定一棵 $n$ 个结点的树，每个结点有两个参数 $a$ 和 $b$ 有 $q$ 次操作：

1. 调整某个结点的第 **1** 个或第 **2** 个参数。
2. 给定 $u$ 和 $v$，查询结点 $u$ 到 $v$ 的路径上，所有第 **1** 个参数与 $u$ 的第 **1** 个参数相同的结点的第 **2** 个参数之和或最大值。

**数据范围：** $1 \leqslant n, q \leqslant 10^5$，两个参数均为不大于 $10^5$ 的正整数。

# 第10章 最短路

最短路是图论中最经典的模型之一，在生活中也有很多应用。例如，我国幅员辽阔，有很多个城市。这些城市之间由许多高速公路相连接。想从上海出发前往北京，有很多条路径，如何选择一条最短的路径就是最基本的最短路问题。有时候问题会更加复杂，加上别的限制条件，例如某些城市拥有服务区，连续开车达到一定的时间就必须要进服务区休息，或者是某些路段在特定的时间内会堵车，需要绕行等，这就需要更加灵活地使用最短路算法来解决各种问题。

除了解决给定图的最短路问题，最短路模型还可以解决许多看起来不是图的问题。如果解决一个问题的最佳方案的过程中涉及很多状态，这些状态是明确的且数量合适，而且状态之间可以转移，就可以考虑建立图论模型，使用最短路算法求解。本章将介绍多种适用于不同情况的最短路算法，可以根据实际情况选择合适的算法，思维导图如图 10-1 所示。




图10-1 本章思维导图

## 10.1 单源最短路径

### 例 10-1 单源最短路径(洛谷P3371)

给定一有 $n$ 个顶点、$m$ 条边的有向图，以及一个起点 $s$，每条边长度（边权）给定，输出 $s$ 到所有终点的**最短路径**长度（如无法到达则输出 $2^{31}-1$） 。所有边的长度均为非负整数，且保证最短路径长度不会超过 $2^{31}-1$，数据保证随机生成。

**输入格式：** 第 $1$ 行包含 $3$ 个整数 $n$，$m$，$s$，分别表示顶点的个数、有向边的个数、出发点的编号。接下来 $m$ 行每行包含 $3$ 个整数 $u$，$v$，$w$，表示一条 $u \to v$、长度为 $w$ 的边。

**输出格式：** 一行 $n$ 个整数，第 $i$ 个整数表示起点 $s$ 到第 $i$ 个顶点的最短路径，若不能到达则输出 $2^{31}-1$。

**输入样例：**

```
461
122
135
144
232
241
343
//
```

**输出样例：**

```
0243
```

**数据范围：** $1 \leqslant n \leqslant 10^4, 1 \leqslant m \leqslant 5 \times 10^5$，所有边的长度非负，且保证最短路径长度不会超过 $2^{31}-1$，保证数据随机。

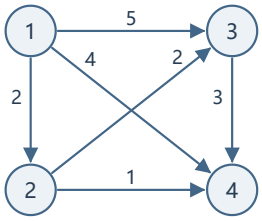
**分析：** 最简单的想法是，使用搜索来寻找所有可能的路径并比较它们的长度，选取最短的路径作为最短路。然而，由于可能的路径较多，时间复杂度是指数级别的，无法在规定时间内运行完毕。因此需要使用最短路算法，过程为从起点开始尝试往外走，不断用已知顶点的最短路长度来更新其他顶点的最短路长度。本章介绍两种算法，其各有优缺点，分别适用于不同情况。

解法 1：Diikstra算法。以上方的样例为例，用一个数组 `dis` 来记录到目前为止起点到各顶点的最短路径长度。执行的步骤如下：

1. 初始时，将整个数组 `dis` 设为 $\mathbf{inf}$，如图 10-2（a） 所示。
2. 对于一条从顶点 $u$ 到 $v$. 长度为 $w$ 的连边，如果 $dis_u + w < dis_v$，那么就可以将 $dis_{,v}$ 的值变成 $dis_u + w$，因为这代表着从起点 $s$ 走到顶点 $u$，经过这条边再走到顶点 $v$，是一条未发现过的且比原先的最优路径还要短的边。这样的操作被称为松弛。由于 $s=1$，所以显然 $dis_1 = 0$ 且不会再变化。于是就可以用 1 号点的所有出边来进行松弛并将 1 标记，代表这个点的 `dis` 值将不再变化。松弛的结果如图 10-2(b) 所示。
3. 可以发现 2 号点的 `dis` 值是当前未标记的点中最小的。由于从 1 走来的边已经全部完成松弛，且 $dis_3$ 和 $dis_4$ 都大于 $dis_2($ （这意味着从 3、4 号点走到 2 号点的边都无法成功松弛），所以此时的 $dis_2$ 就是从 1 到 2 的最短路。接着就可以用与 2 号点相连的所有边进行松弛并将 2 标记，松弛结果如图 10-2(c) 所示。
4. 同样地，发现 4 号点的 `dis` 是当前未标记的点中最小的，它的 `dis` 值不会再改变。接着用 4 号点进行松弛并标记，如图 10-2(d) 所示。

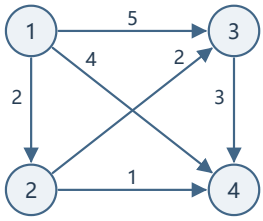
> **注：** 1. `inf` 表示无穷大，一般用足够大的数代替，如 $231-1$。

5. 最后，用 3 号点进行松弛并标记得到最终结果，如图 10-2(e) 所示。



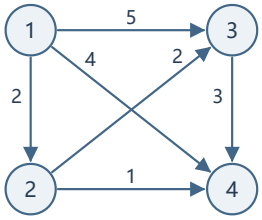

| 编号 | dis |
|---|---|
| 1 | ∞ |
| 2 | ∞ |
| 3 | ∞ |
| 4 | ∞ |

(a)



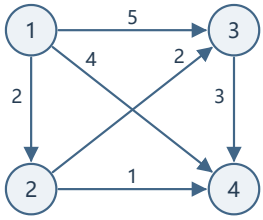

| 编号 | dis |
|---|---|
| 1* | 0 |
| 2 | 2 |
| 3 | 5 |
| 4 | 4 |

(b)



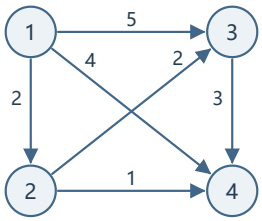

| 编号 | dis |
|---|---|
| 1 | 0 |
| 2* | 2 |
| 3 | 4 |
| 4 | 3 |

(c)




| 编号 | dis |
|---|---|
| 1 | 0 |
| 2 | 2 |
| 3 | 4 |
| 4* | 3 |

(d)




| 编号 | dis |
| --- | --- |
| 1 | 0 |
| 2 | 2 |
| 3* | 4 |
| 4 | 3 |

(e)

这种求最短路的方法是Dijkstra算法，其过程概括如下：

1. 将起始点的 `dis` 置为 0。
2. 选择当前未标记的顶点中 `dis` 值最小的一个。
3. 对该顶点的所有连边依次进行松弛操作。
4. 对该顶点进行标记。
5. 重复第（2）步至第（4）步，直到不存在一条从已标记顶点通往未标记点的连边。代码如下：

```
#include <bits/stdc++.h>
#define MAXN 500010
using namespace std;
const int inf = 2147483647;
int n, m, s, cnt; // 标记
int dis[MAXN], h[MAXN], to[MAXN], val[MAXN], nxt[MAXN];
bool vis[MAXN];
void add(int a, int b, int c) {
    to[++cnt] = b;
    val[cnt] = c;
    nxt[cnt] = h[a];
    h[a] = cnt;
} // 链式前向星加边函数
void Dijkstra() {
    for (int i = 1; i <= n; i++)
        dis[i] = inf;
    dis[s] = 0;
    dis[0] = inf;
    while (true) {
        int u = 0;
        for (int i = 1; i <= n; i++) {
            if (!vis[i] && dis[u] > dis[i]) {
                u = i;
            }
```

```
        }
        // 寻找下一个处理的点
        if (u == 0)
            break; // 如果没有可处理的点就结束程序
        vis[u] = 1;
        for (int i = h[u]; i; i = nxt[i]) {
            if (dis[to[i]] > (long long)dis[u] + val[i]) { // 松弛操作,注意溢出
                dis[to[i]] = dis[u] + val[i];
            }
        }
    }
}
int main() {
    scanf("%d%d%d ", &n, &m, &s);
    for (int i = 1, u, v, w; i <= m; i++) {
        scanf("%d%d%d", &u, &v, &w);
        add(u, v, w);
    }
    Dijkstra();
    for (int i = 1; i <= n; i++)
        printf("%d ", dis[i]);
    return 0;
}
```

本题的代码在存储图的数据时采用了链式前向星，也是一种邻接表，其原理是将从每个顶点出发的所有边以一个链表的形式保存，每一条边都是链表的一个结点。数组nxt代表该列表下一项的数组下标：数组 $h$ 是每个结点对应链表第一项的数组下标。加入一条边可以认为是在该边起点的链表上插入一个结点。这种存图方法是利用在链表的头部存储新的结点，因此遍历所有边时其遍历顺序恰好与输入顺序相反。

如果是无向图，每加入一条边，都可以认为是加入两条方向相反的有向边。这个时候需要特别注意，链式前向星需要定义的数组元素数量上限，要超过边数的两倍。

经过简单的分析可以发现，该算法的时间复杂度是 $O(n^2+m)$。尽管这一复杂度可以通过本题，但依然不是很能够被接受。

有一种比较简单的优化方法：使用一个小根堆来维护 `dis` 最小的结点。用结构体记录结点编号 $i$ 和 $dis_i$ . 并通过重载运算符以 $dis_i$ 为关键字进行排序。每次松弛成功时，将被松弛的边的终点和对应的 `dis` 值打包放入堆中，在需要寻找 `dis` 最小的结点时将堆顶端的点取出，验证是否已经被标记即可。由于每一条边最多被经过入堆、出堆各一次，且堆内元素最多为 $m$ 个，其时间复杂度为 $O(m\log m)$。代码如下：

```
struct node {
    int v, w;
    friend bool operator<(node a, node b) { return a.w > b.w; }
} tmp;                  // 用于存储点与其dis值组合的结构体
priority_queue<node> q; // 用于维护未标记的dis 最小的结点的优先队列
void Dijkstra() {
    for (int i = 1; i <= n; i++)
        dis[i] = inf; // 初始化
```

```
    dis[s] = 0;
    tmp.v = s, tmp.w = 0;
    q.push(tmp);
    while (!q.empty()) {
        int u = q.top().v;
        q.pop();
        if (vis[u])
            continue; // 判断是否被标记过
        vis[u] = 1;   // 标记并处理
        for (int i = h[u]; i; i = nxt[i]) {
            if (dis[to[i]] > (long long)dis[u] + val[i]) { // 松弛操作(注意防止越界)
                dis[to[i]] = dis[u] + val[i];
                tmp.w = dis[to[i]], tmp.v = to[i];
                q.push(tmp);
            }
        }
    }
}
```

容易发现，当 $m$ 和 $n$ 同级时，采用堆优化会使得程序运行效率获得极大的提升；但如果 $m$ 与 $n^2$ 同级①，那么使用堆优化之后复杂度变为 $O(n^2 \log n^2)$，反而劣于不加优化的Dijkstra。在实际应用时，应当根据实际数据的范围来选择使用哪种方法。

Dijkstra算法的优势是时间复杂度较低，缺点是一旦边长出现负数其正确性就无法保证。在本题中保证了边长为非负数，如果碰到解决边长可能为负的单源最短路问题，则需要使用一些其他算法。

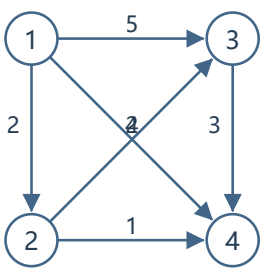
解法 2：SPFA算法。还是以上面的样例为例，数组 `dis` 意义不变，进行如下操作：

1. 同样先对 1 号点出发的边进行松弛；同时，将松弛成功的边的终点放入一个队列之中。此时的结果如图 10-3(a) 所示。
2. 接着，依次以队列中的元素为起点，对该点出发的边进行松弛。在对 4 号点与 3 号点出发的边进行松弛的过程中，没有点的 `dis` 值被改变，此时队列中仅剩 2 号点。
3. 然后对 2 号点出发的所有边进行松弛，发现两条边均松弛成功，即 3 号点和 4 号点的 `dis` 值均被改变。将这两个点重新加入队列中，此时队列中还有 3 号和 4 号两个点，结果如图 10-3(b) 所示。
4. 最后，对 3 号点与 4 号点出发的边进行松弛，没有边松弛成功。此时队列为空，程序结束，得到最终结果，如图 10-3(c) 所示。

这种使用队列维护，求解最短路的方法被称为SPFA算法，其过程概括如下：

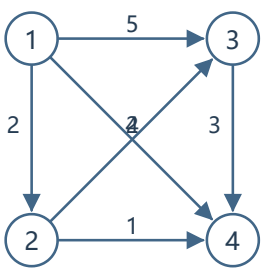
1. 将起始点的 `dis` 置为 0 并放入队列。

> **注：** 1. 例如完全图等稠密图。



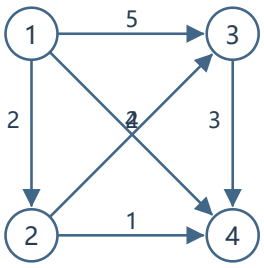

| 编号 | dis |
|---|---|
| 1 * | 0 |
| 2 | 2 |
| 3 | 5 |
| 4 | 4 |

(a) 队列：4 3 2




| 编号 | dis |
|---|---|
| 1 | 0 |
| 2 * | 2 |
| 3 | 4 |
| 4 | 3 |

(b) 队列：3 4




| 编号 | dis |
|---|---|
| 1 | 0 |
| 2 | 2 |
| 3 | 4 |
| 4 | 3 |

(c) 队列：空

(a) (b) (c)

| 编号 | dis |
|---|---|
| 1 | 0 |
| 2 | 2 |
| 3 | 4 |
| 4 | 3 |

图10-3 SPFA算法的步骤

2. 选择当前队列中的第 1 个顶点并将其从队列中删除。

3. 对该顶点的所有连边依次进行松弛操作。

4. 将第（3）步中松弛成功且不在队列内的顶点放入队列。

5. 重复第（2）步至第（4）步，直到队列为空。

该算法的代码如下：

```
queue<int> q;
void spfa() {
    for (int i = 1; i <= n; i++)
        dis[i] = inf;
    dis[s] = 0;
    vis[s] = 1;
    q.push(s);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        vis[u] = 0; // 标记出队
        for (int i = h[u]; i; i = nxt[i])
            if (dis[to[i]] > (long long)dis[u] + val[i]) { // 松弛操作
                dis[to[i]] = dis[u] + val[i];
                if (!vis[to[i]]) { // 松弛成功且不在队列中则入队
                    q.push(to[i]);
                    vis[to[i]] = 1;
                }
            }
    }
}
```

在SPFA的过程中会有一个点可能多次入队，这使寻找最短路的效率大大降低。实际上，SPFA的时间复杂度在最劣情况下可以达到 $O(nm)$，因此只要数据稍大且构造特定图就可能超时，但它在随机数据下速度较快，且当边长出现负数时依然正确。由于本题保证数据随机，因此SPFA可以通过本题，但若去掉此限制，则SPFA会超时。因此在算法竞赛中，如果确定图中边长非负且没有特殊的限制时，不应使用SPFA算法。

SPFA算法除了最短路之外还有一些其他的应用（如判断负环），将在 10.2 中介绍。

### 例 10-2 飞行路线(洛谷P4568,吉林省队选拔2011)

某航空公司一共在 $n$ 个城市设有业务，这些城市分别标记为 $0$ 到 $n-1$，一共有 $m$ 条航线，每条航线连接两个城市，乘坐一次飞机需要支付对应的费用 $c_i$。 小 $A$ 现在要从编号为 $s$ 的城市沿着航线到达编号为 $t$ 的城市，途中可以进行转机。航空公司为她提供了优惠，可以免费在最多 $k$ 条航线上搭乘飞机。那么小 $A$ 本次出行最少花费多少?数据范围：
$2 \leqslant n \leqslant 10^4, 1 \leqslant m \leqslant 5 \times 10^4, 0 \leqslant k \leqslant 10, 0 \leqslant c_i \leqslant 10^3$，

**分析：**由于购买机票需要一定花费，因此肯定不会重复乘坐多次同样的航线或者多次访问同一个城市。如果 $k=0$，本题就是最基础的最短路问题，但题目中提供了一些特殊情况（对有限条边设置为免费），可以使用分层图的方式。将图多复制 $k$ 次，原编号为 $i$ 的顶点复制为编号 $i+jn(1 \leqslant j \leqslant k)$ 的顶点，然后对于原图存在的边，第 $j$ 层和第 $j+1$ 层的对应顶点也需要连上，看起来就是相同的图上下堆叠起来，因此被称为分层图，如图 10-4 所示。

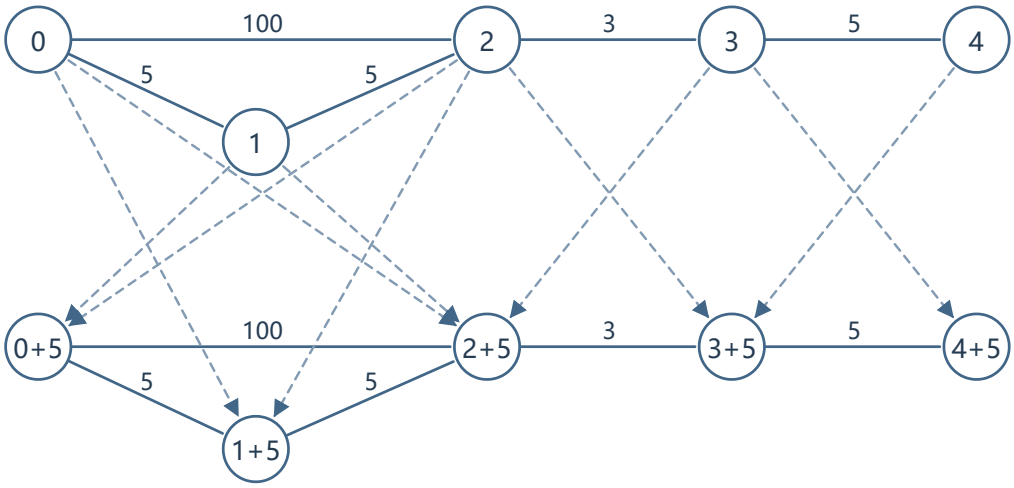



图10-4 分层图

聪明的读者一定可以发现，从上面一层跳到下面一层就是乘坐免票的飞机，花费的代价是 $0$。这个过程是不可逆的，每乘坐一次飞机就跳到下一层图中，因此上一层到下一层的边是单向边。从编号为 $s$ 的顶点开始，计算到其他结点的单源**最短路径**。因为并不一定要坐满 $k$ 次免费航班，所以查找所有编号 $t+jn(0 \leqslant j \leqslant n)$ 的顶点，也就是终点的最短路的值，找到最小的值就是答案。其主函数代码如下（其余部分见例 10-1 中堆优化的Dijkstra算法）：

**代码片段（需结合上文的声明和函数）：**

```
int main() {
    scanf("%d%d%d", &n, &m, &k);
    scanf("%d%d", &s, &t);
    for (int i = 0, u, v, w; i < m; i++) {
        scanf("%d%d%d", &u, &v, &w);
        add(u, v, w);
        add(v, u, w);
        for (int j = 1; j <= k; j++) {
            add(u + j * n, v + j * n, w);
            add(v + j * n, u + j * n, w);
            add(u + j * n - n, v + j * n, 0);
            add(v + j * n - n, u + j * n, 0);
        }
    }
    Dijkstra();
    int ans = inf;
    for (int j = 0; j <= k; j++)
        if (dis[t + j * n] < ans)
            ans = dis[t + j * n];
    printf("%d\n", ans);
    return 0;
}
```

本题中一共有 $k+1$ 层图，每条边实际上可能存储了 4 次，所以这里表示数组上限的 `MAXN` 一定要设置为大于 $5 \times 10^4 \times 4 \times 11 = 2.2 \times 10^6$ 的数字，否则有可能存不下所有的边。

### 例 10-3 邮递员送信(洛谷P1629)

给定一张有 $n$ 个顶点、$m$ 条边的有向图，邮递员从顶点 $1$ 出发，有 $n-1$ 个快递，分别要送到顶点 $2$ 到 $n$，每次邮递员只能携带一个快递出发并在送完后返回邮局。求邮递员走过的最少的路程。数据范围： $1 \leqslant n \leqslant 10^3, 1 \leqslant m \leqslant 10^5$，保证任意两点可相互到达。

**分析：** 由于邮递员必须每次都要从顶点 1 出发，到达顶点 $i$（后面简称 $i$），然后返回顶点 1，因此就是求从顶点 1 到 $i$ 的最短路，加上顶点 $i$ 到 1 的最短路。考虑到是有向图，从顶点 1 到的距离不一定等于从顶点 $i$ 到 1 的距离。计算从顶点 1 到 $i$ 的距离方式很简单，但如何求从其他顶点到顶点 1 的距离呢？想象从某个顶点到顶点 1 的**最短路径**，不就是从顶点 1 开始，倒着沿路走吗？

于是可以建立另外一个图，将读入的边起点和终点对调后存入这个新图中，从顶点 1 开始计算到其他顶点的单源最短路径，得到的距离就是原图从其他顶点到起点的最短路径。一来一回加起来就是送货的最短路径。代码略。

```
一些最短路相关问题的小技巧:
1．某些边只能走有限的次数,可以将图复制多遍,成为分层图。
2．给出若干个关键点,求其他顶点离这些关键点的距离,可以建立一个虚拟点,并使用长
// 度为0的边将虚拟点和这些关键点连接。
3．查找有向图中,所有顶点到某个顶点的最短路,可以将图反向建边。
4．注意图中是否存在负环、自环、重边等情况。
```

## 10.2 负环与差分约束

### 例 10-4 负环(洛谷P3385)

给定一有 $n$ 个顶点的有向图，判断图内是否存在从顶点 $1$ 出发能到达的负环。

**输入格式：** 本题单测试点有多组测试数据。输入的第 $1$ 行是一个整数 $T$，表示测试数据的组数。对于每组数据的格式如下：

第 $1$ 行有两个整数，分别表示图的点数 $n$ 和接下来给出边信息的条数 $m$。

接下来 $m$ 行，每行 $3$ 个整数 $u, v, w$，

1. 若 $w \geqslant 0$，则表示存在一条从顶点 $u$ 至 $v$、边权为 $w$ 的边，还存在一条从顶点 $v$ 至 $u$、边权为 $w$ 的边。
2. 若 $w < 0$，则只表示存在一条从顶点 $u$ 至 $v$、边权为 $w$ 的边。

**输出格式：** 每组数据输出一行。若存在所求负环，输出 `YES`；否则输出 `NO`。

**数据范围：** $1 \leqslant n \leqslant 2 \times 10^3, 1 \leqslant m \leqslant 3 \times 10^3, 1 \leqslant u, v \leqslant n, -10^4 \leqslant w \leqslant 10^4, 1 \leqslant T \leqslant 10$。

**分析：** 在图论中边权之和为负数的环被称为负环。显然，如果一个连通图中存在一个从顶点 1 出发能到的负环，则顶点 1 到该负环上的任意一点不存在最短路（因为不停地沿着负环绕路径长度会越来越短）。因此如果对其使用SPFA算法求最短路，那么负环上的任何顶点都会进入队列无数次，最终程序进入死循环。

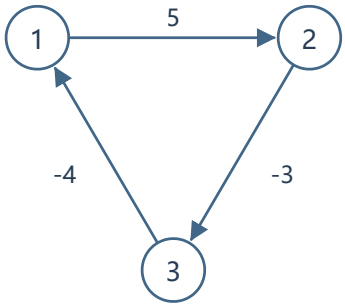



图10-5 负环

如图 10-5 所示是一个负环。从开始SPFA，每绕负环走一圈，任何顶点的 `dis` 值都会减小 2，不断地绕，`dis` 就会趋向负无穷。因此对于 图 10-5 中的 3 个顶点而言，都没有从顶点 1 到该点的最短路。

又由于对于一个不存在顶点 1 能到达的负环的图，从顶点 1 出发到所有能到达的顶点至少有一条从顶点 1 出发到该点的**最短路径**点数（下面简称最短路径点数）不会超过 $n$（否则由抽屉原理，就会有一个顶点经过了两次，那么这个顶点两次经过的中间那一段就是一个环；又由

于不存在负环，所以环的边权之和大于等于去掉该环后的路径，将中间那段去掉不会更劣）。

因此可以最短路径上点数入手。由于队列的性质，队列中元素在当前状态下的最短路径点数是单调不减的，所以如果一个元素出队又进队了，那么它再进队时的最短路径点数一定比上一次出队的时候至少增加 1。因此如果一个顶点进队超过 $n$ 次，它的当前最短路径上的点数就超过了 $n$ 个点，这就意味着出现了负环。

于是只需要将SPFA的代码略做修改，统计入队次数即可。

```
#include <bits/stdc++.h>
#define MAXN 2005
#define MAXM 20005
const long long inf = 2147483647;
using namespace std;
int n, m, cnt0;
int dis[MAXN], vis[MAXN], head[MAXN], cnt[MAXN]; // 数组 cnt 用于统计入队次数
struct Edge {
    int nxt, to, dis;
} edge[MAXM];
void add(int u, int v, int w) {
    edge[++cnt0].nxt = head[u];
    edge[cnt0].to = v;
    edge[cnt0].dis = w;
    head[u] = cnt0;
}
bool spfa() { // spfa 算法,返回值为 1 代表有负环
    queue<int> q;
    for (int i = 1; i <= n; i++) {
        dis[i] = inf;
        vis[i] = 0;
    }
    q.push(1);
    dis[1] = 0;
```

```
    vis[1] = 1;
    cnt[1]++;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        vis[u] = 0;
        for (int i = head[u]; i; i = edge[i].nxt) {
            int v = edge[i].to;
            if (dis[v] > (long long)dis[u] + edge[i].dis) {
                dis[v] = dis[u] + edge[i].dis;
                if (vis[v] == 0) {
                    vis[v] = 1;
                    q.push(v);
                    cnt[v]++;
                    if (cnt[v] > n)
                        return 1;
                    // 找到入队次数大于n的顶点,存在可到达的负环
                }
            }
        }
    }
    return 0; // spfa过程结束,未找到可到达的负环
}
void init() { // 测试点预处理
    memset(dis, 0, sizeof(dis));
    memset(vis, 0, sizeof(vis));
    memset(cnt, 0, sizeof(cnt));
    memset(edge, 0, sizeof(edge));
    memset(head, 0, sizeof(head));
    cnt0 = 0;
}
int main() {
    int T;
    cin >> T;
    for (int a = 1; a <= T; a++) {
        init();
        cin >> n >> m;
        for (int i = 1; i <= m; i++) {
            int u, v, w;
            cin >> u >> v >> w;
            add(u, v, w);
            if (w >= 0)
                add(v, u, w);
        }
        if (spfa())
            cout << "YES " << endl;
        else
            cout << "NO" << endl;
    }
    return 0;
}
```

学习完负环的判断，下面接着介绍负环的一个重要应用：差分约束。

### 例 10-5 差分约束算法(洛谷P5960)

给出一组包含 $m$ 个不等式、有 $n$ 个未知数的形如下面提供的公式的不等式组，求任意一组满足这个不等式组的解。

$$\begin{cases} x_{c_1} - x_{c'_1} \leqslant y_1 \\ x_{c_2} - x_{c'_2} \leqslant y_2 \\ \vdots \\ x_{c_m} - x_{c'_m} \leqslant y_m \end{cases}$$

**输入格式：** 第 $1$ 行为两个正整数 $n$ 和 $m$，代表未知数的数量和不等式的数量。

接下来 $m$ 行，每行包含 $3$ 个整数 $c, c', y$，代表一个不等式 $x_c - x_{c'} \leqslant y_c$

**输出格式：** 一行，$n$ 个数，表示 $x_1, x_2, \cdots, x_n$ 的一组可行解，如果有多组解，请输出任意一组，无解请输出 $\mathbf{NO}$。

**数据范围：** $1 \leqslant n, m \leqslant 5 \times 10^3, -10^4 \leqslant y \leqslant 10^4, 1 \leqslant c, c' \leqslant n, c \neq c'$

**分析：** 对于约束 $x_u - x_v \leq y$，可以改写为 $x_u \leq x_v + y$。这与最短路中的 $\mathbf{dis}_u \leq \mathbf{dis}_v + w$ 形式相同。因此，对每个约束建立一条从顶点 $v$ 指向顶点 $u$、边权为 $y$ 的边，使图中的边与不等式一一对应，从而转化为最短路问题。

建一个源点 0 作为最短路的起点，向其他每个顶点连一条边权为 0 的边。此时从源点 0 出发SPFA，就可以判断图中是否存在负环。显然，如果图中不存在负环，那么此时可以直接令 $x_i = dis_i$，就是一组满足条件的解（因为数组 `dis` 一定满足 $dis_{c_i} \leqslant dis_{c'_i} + y_i$，和不等式一一对应）。

而如果图中出现负环，不妨设组成负环的边对应的所有不等式是 $x_{a_i} - x_{a_{i+1}} \leqslant y_{b_i} (1 \leqslant i \leqslant k\ 2 a_{k+1} = a_1$ ），则将所有不等式左边右边分别相加，得到 $0 \leqslant \sum_{i=1}^{k} y_{b_i}$；又由于该环为负环，故 $\sum_{i=1}^{k} \gamma_{b_i} < 0$，矛盾，因此出现负环就意味着原不等式组无解。

在本题中所有的约束条件都是形如 $x_i - x_j \leqslant y$ 的。在其他题目中，如果出现了形如 $x_i - x_j \geqslant y$ 的条件，处理方法是在不等式两旁同时乘以-1，得到 $x_j - x_i \leqslant -y$。如果出现了形如 $x_i - x_j = y$，则只须将它拆分成 $x_i - x_j \leqslant y$ 与 $x_i - x_j \geqslant y$ 两个约束条件进行处理。

本题的代码如下：

```
#include <bits/stdc++.h>
#define MAXN 50005
const int inf = 2147483647;
using namespace std;
int n, m, cnt0;
int dis[MAXN], vis[MAXN], head[MAXN], cnt[MAXN];
struct Edge {
    int nxt, to, dis;
} edge[MAXN * 2];
void add(int u, int v, int w) {
    edge[++cnt0].nxt = head[u];
    edge[cnt0].to = v;
    edge[cnt0].dis = w;
    head[u] = cnt0;
}
bool spfa() { // SPFA 找负环
```

```
    queue<int> q;
    for (int i = 1; i <= n; i++) {
        dis[i] = inf;
        vis[i] = 0;
    }
    q.push(0);
    dis[0] = 0;
    vis[0] = 1;
    cnt[0] = 1;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        vis[u] = 0;
        for (int i = head[u]; i; i = edge[i].nxt) {
            int v = edge[i].to;
            if (dis[v] > (long long)dis[u] + edge[i].dis) {
                dis[v] = dis[u] + edge[i].dis;
                if (vis[v] == 0) {
                    vis[v] = 1;
                    q.push(v);
                    cnt[v]++;
                    if (cnt[v] > n + 1)
                        return 1;
                }
            }
        }
    }
    return 0;
}
int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        add(v, u, w); // 依据不等式建图
    }
    for (int i = 1; i <= n; i++) {
        add(0, i, 0); // 将源点0放入图中
    }
    if (!spfa()) {
        for (int i = 1; i <= n; i++) {
            printf("%d ", dis[i]);
        }
    } else
        cout << "NO " << endl;
    return 0;
}
```

## 10.3 次短路

### 例 10-6 路障(洛谷P2865,USACO2006年11月赛)

给定一有 $n$ 个顶点、$r$ 条边的无向图，求从顶点 $1$ 到顶点 $n$ 的严格次短路（即长度大于**最短路径**的路径中最短的一条，一条边可以经过多次）。

**数据范围：** $1 \leqslant n \leqslant 5 \times 10^3, 1 \leqslant r \leqslant 10^5, 1 \leqslant w \leqslant 5 \times 10^3$。

**分析：**在任何时刻，如果发现了一条新的从顶点 1 到 $i$ 的路径，则该路径可能存在以下几种情况：

1. 该路径长度小于目前的顶点 1 到 $i$ 的最短路。此时的处理方式是将目前的次短路用目前的最短路替换，而最短路径替换为新路径。
2. 该路径长度等于目前的最短路，直接舍去。
3. 该路径长度大于目前的最短路且小于目前的次短路。此时的处理方式是直接用这条路径替换当前的次短路径。
4. 该路径长度大于或等于目前的次短路，直接舍去。

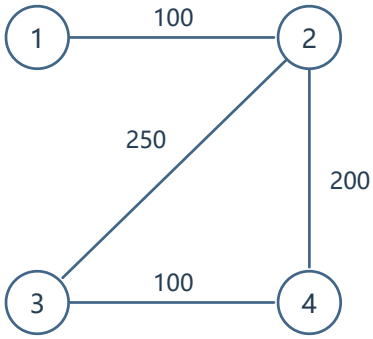
对于本题而言，长度大于目前已知路径中的次短路的路径都是没有意义的，因此只需要考虑当前状态下的最短路和次短路即可。这里用一个数组 `dis`[**0/1**，$i$ 记录当前状态下从 1 号点出发到 $i$ 号点的最短路和次短路（第一维为 0 表示最短路，为 1 表示次短路）。

回顾Dijkstra算法，由于边长非负，可以将所有顶点的最短路装进一个**优先队列**，每一次入队都代表找到了一条更优的路径。而每当一个顶点第一次被从优先队列中取出时，代表其已知路径中最短者从优先队列中被取出；而又由于边权非负，这个顶点的最短路就不会再更改。

同样地，在本题中，由于只有最短路和次短路有效，因此这个顶点的所有可能的有效路径（更新成功的最短路和次短路）全部都在这个优先队列中。当这个顶点第 1 次从优先队列中出来时，其最短路值就不会再被更改；第 2 次从优先队列中出来时，其次短路值就不会再更改（因为其他的路径显然都长于这两条路径）。

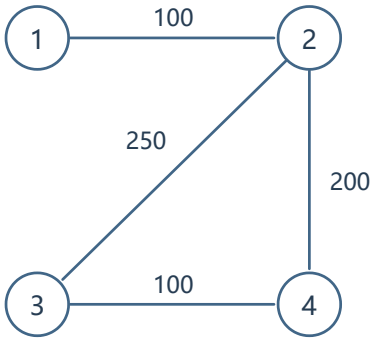
然后再用从优先队列中取出的路径来更新相邻的点即可，更新规则按照上方叙述的几步进行。这里使用了一个例子来直观感受次短路的过程。

1. 最开始时的图，以及最短路、次短路的数据如图 10-6(a) 所示。
2. 先用 1 号点到 2 号点的路径来更新 2 号点的最短路，如图 10-6(b) 所示。
3. 此时优先队列中仅有一条到达 2 号点，长度为 100 的一条路径。将取出并用于更新，如图 10-6(c) 所示。
4. 此时优先队列中存在 3 条路径，分别是到达 1 号点长度为 200、到达 4 号点长度为 300、到达 3 号点长度为 350 的路径。取出第 1 条进行更新，如图 10-6(d) 所示。
5. 此时优先队列中存在 3 条路径，分别是到达 4 号点长度为 300、到达 2 号点长度为 300 到达 3 号点长度为 350 的路径。取出第 1 条进行更新。此时 4 号点共有 2 号和 3 号两个相邻点。可以发现此时无法更新 2 号点的路径长度，但可以更新 3 号点的次短路长度，如图 10-6(e) 所示。
6. 此时优先队列中存在 3 条路径，分别是到达 2 号点长度为 300、到达 3 号点长度为 350、到达 3 号点长度为 400 的路径。注意尽管 3 号点已经有一条路径在优先队列中，但由于其次短路被更新，需要再次进入优先队列。按照上述方式继续进行更新，最后的结果如图 10-6(f) 所示。




| 编号 | 最短路 | 次短路 |
|---|---|---|
| 1 | 0 | ∞ |
| 2 | ∞ | ∞ |
| 3 | ∞ | ∞ |
| 4 | ∞ | ∞ |

(a)




| 编号 | 最短路 | 次短路 |
|---|---|---|
| 1 | 0 | ∞ |
| 2 | 100 | ∞ |
| 3 | ∞ | ∞ |
| 4 | ∞ | ∞ |

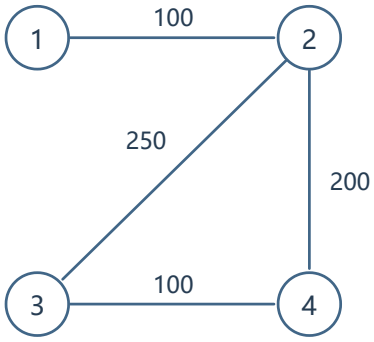
(b)

| 编号 | 最短路 | 次短路 |
|---|---|---|
| 1 | 0 | 200 |
| 2 | 100 | ∞ |
| 3 | 350 | ∞ |
| 4 | 300 | ∞ |

(c)

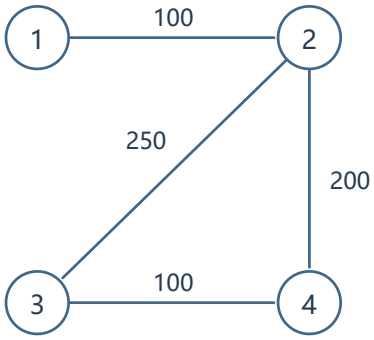
| 编号 | 最短路 | 次短路 |
|---|---|---|
| 1 | 0 | 200 |
| 2 | 100 | 300 |
| 3 | 350 | ∞ |
| 4 | 300 | ∞ |

(d)




| 编号 | 最短路 | 次短路 |
|---|---|---|
| 1 | 0 | 200 |
| 2 | 100 | 300 |
| 3 | 350 | 400 |
| 4 | 300 | ∞ |

(e)




| 编号 | 最短路 | 次短路 |
|---|---|---|
| 1 | 0 | 200 |
| 2 | 100 | 300 |
| 3 | 350 | 400 |
| 4 | 300 | 450 |

(f)

图10-6 次短路

实现代码如下：

```
#include <bits/stdc++.h>
#define MAXN 200010
using namespace std;
struct Edge {
    int to, nxt, val;
} e[MAXN];
int h[MAXN], cnt, dis[3][MAXN], n, m;
void add(int from, int to, int value) {
    cnt++;
    e[cnt].nxt = h[from];
    e[cnt].to = to;
    e[cnt].val = value;
    h[from] = cnt;
}
struct node {
    int pos, dis;
```

```
    friend bool operator<(node a, node b) { return a.dis > b.dis; }
} tmp;
priority_queue<node> q;
void Dijkstra2() {
    for (int i = 1; i <= n; i++) {
        dis[0][i] = dis[1][i] = 2147483647;
    }
    dis[0][1] = 0;
    tmp.dis = 0;
    tmp.pos = 1;
    q.push(tmp);
    while (!q.empty()) {
        tmp = q.top();
        q.pop();
        int u = tmp.pos, d = tmp.dis;
        if (d > dis[1][u])
            continue; // 说明这条路径已经无效,可以舍去
        for (int i = h[u]; i; i = e[i].nxt) {
            int v = e[i].to;
            int w = e[i].val;
            if (dis[0][v] > d + w) { // 情况 1
                dis[1][v] = dis[0][v];
                tmp.dis = dis[0][v] = d + w;
                tmp.pos = v;
                q.push(tmp);
            }
            if (dis[1][v] > d + w && dis[0][v] < d + w) { // 情况 3
                tmp.dis = dis[1][v] = d + w;
                tmp.pos = v;
                q.push(tmp);
            }
        }
    }
}
int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int a, b, c;
        cin >> a >> b >> c;
        add(a, b, c);
        add(b, a, c); // 无向图建双向边
    }
    Dijkstra2();
    cout << dis[1][n];
    return 0;
}
```

## 10.4 多源最短路径与传递闭包

### 例 10-7 危险降临(洛谷 P2910,USACO08 OPEN)

给定一张有 $n$ 个顶点的图和图中每两个顶点之间的边权，以及一个 $m$ 个顶点的序列，求该序列中相邻的顶点的**最短路径**之和。

**输入格式：**

第 $1$ 行：两个用空格隔开的正整数 $n$ 和 $m$

第 $2$ 行到第 $m+1$ 行：第 $i+1$ 行用一个整数 $A_i$ 表示需要经过的第 $i$ 个顶点。

第 $m+2$ 行到第 $n+m+1$ 行：第 $i+m+1$ 行包含 $n$ 个用空格隔开的非负整数分别表示 $i$ 号点到第 $1$，$2$，…，$n$ 号点的直接路径长度。保证第 $i$ 个数是 $0$。

**输出格式：**一行一个整数代表最终答案。

**输入样例：**

```
3 4
1
2
1
3
0 5 1
502
12 0
```

**输出样例：**

**数据范围：** $0 \leqslant n \leqslant 100, 0 \leqslant m \leqslant 10^4$，保证边权非负且不超过 $10^4$。

**分析：**本题同样是一道最短路题，但是它与之前题目的不同之处在于，它需要处理从不同顶点出发的最短路径，也就是多源最短路径。顶点的数量较少，但任意两点之间都有连边（即边数约为 $O(n^2)$ 级别）。最简单的方法是采用 $n$ 遍Dijkstra算法求最短路，其复杂度为 $O(n^3)$，已经足够通过本题。

但事实上，对于多源最短路，可以使用Floyd-Warshall算法（简称Floyd算法）更便捷地解决此类问题。

采用**动态规划**的思路：设 $dis[i,j,k]$ 表示当只途径编号为 1 到 $k$ 的顶点时（起始点和终点不算），$i$ 到 $j$ 的最短路的长度。特别地， $dis[i,j,0]$ 表示初始时顶点 $i$ 和 $j$ 间的边的边权。

考虑如何进行状态更新。显然从顶点 $i$ 到 $i$ 且只经过编号为 1 到 $k$ 的顶点的路径分为两类：

1. 从 $i$ 到 $j$，且只经过编号为 1 到 $k-1$ 的顶点的路径。

2. 从 $i$ 到 $k$，再从 $k$ 到 $j$，且只经过编号为 1 到 $k$ 的顶点的路径。

显然对于第 1 类路径，最短者的长度就是 $dis[i,j,k-1]$；对于第 2 类路径，最短者的长度就是 $dis[i,k,k-1]+dis[k,j,k-1]$。两者当中取最小值，就得到转移方程：

$$dis[i,j,k]=\min(dis[i,j,k-1],dis[i,k,k-1]+dis[k,j,k-1])$$

然后只需要依次枚举 $k,i,j$ 并进行转移即可，最终结果即为第 3 维为 $n$ 的数组。注意由于在第 3 维为 $k$ 的时候需要用到第 3 维为 $k-1$ 的数据，所以在枚举时应当先枚举第 3 维 $k$，再枚举 $i$ 和 $j$。

代码非常简洁，内容如下：

```
for (int k = 1; k <= n; k++)
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            dis[i][j][k] = min(dis[i][j][k - 1], dis[i][k][k - 1] + dis[k][j][k - 1]);
```

时间复杂度为 $O(n^3)$，空间复杂度为 $O(n^3)$。在时间上已经难以对该算法进行优化，但在空间上还可以进一步优化。

显然 $dis[i,k,k-1]$ 与 $dis[i,k,k]$ $:,dis[k,j,k-1]$ 与 $dis[k,j,k]$ 分别相等，于是发现数组第三维 $k$ 是多余的，可以直接用二维数组来解决此题。转移方程变为：

$$dis[i,j]=\min(dis[i,j],dis[i,k]+dis[k,j])$$

空间复杂度变为 $O(n^2)$，时间复杂度不变。最终代码如下：

```
#include <bits/stdc++.h>
#define MAXN 105
#define MAXM 10005
using namespace std;
int n, m, a[MAXM], dis[MAXN][MAXN];
int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++)
        cin >> a[i];
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            cin >> dis[i][j];
    for (int k = 1; k <= n; k++) // Floyd 主体部分
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                dis[i][j] = min((long long)dis[i][j], (long long)dis[i][k] + dis[k][j]);
    int ans = 0;
    for (int i = 1; i <= m - 1; i++) {
        ans += dis[a[i]][a[i + 1]];
    }
    cout << ans;
    return 0;
}
```

当图中存在负权边但不存在可达负环时，Floyd 算法仍可计算最短路径。它既可以处理无向图，也可以处理有向图，注意设置 `dis` 的初始值即可。

### 例 10-8 传递闭包(洛谷B3611)

给定一点数为 $n$ 的有向图的邻接矩阵，图中不包含自环，求该有向图的传递闭包。

一图的邻接矩阵定义为一个 $n\times n$ 的矩阵 $A=(a_{ij})_{n\times n}$，其中：

$$a_{ij}=\begin{cases}1, & \text{顶点 } i \text{ 到 } j \text{ 存在直接连边},\\ 0, & \text{顶点 } i \text{ 到 } j \text{ 没有直接连边}.\end{cases}$$

图的传递闭包定义为一个 $n\times n$ 的矩阵 $B=(b_{ij})_{n\times n}$，其中：

$$b_{ij}=\begin{cases}1, & \text{顶点 } i \text{ 可以直接或间接到达顶点 } j,\\ 0, & \text{顶点 } i \text{ 无法直接或间接到达顶点 } j.\end{cases}$$

**输入样例：**

```
4
0 0 0 1
1 0 0 0
0 0 0 1
0 1 0 0
```

**输出样例：**

```
1 1 0 1
1 1 0 1
1 1 0 1
1 1 0 1
```

**数据范围：** $1 \le n \le 100$，保证 $a_{ij} \in \{0,1\}$ 且 $a_{ii} = 0$。

**分析：** 处理传递闭包同样可以采用 Floyd－Warshall 算法。设 $f[i,j,k]$ 表示当只允许编号为 $1$ 到 $k$ 的顶点作为中间顶点时，顶点 $i$ 能否直接或间接到达顶点 $j$。特别地，$f[i,j,0]$ 表示初始状态下是否存在顶点 $i$ 到 $j$ 的直接连边。

可以写出本题的**动态规划**转移方程：

$$f[i,j,k] = f[i,j,k-1] \vee \big(f[i,k,k-1] \wedge f[k,j,k-1]\big).$$

其中，$\vee$ 和 $\wedge$ 分别表示“或”和“与”的逻辑关系。依次枚举 $k,i,j$ 进行转移即可得到最终结果。

类似的，容易发现最后一维没有意义，可以省去。省去之后的时间复杂度为 $O(n^3)$，空间复杂度为 $O(n^2)$。代码的主体部分如下：

```
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            f[i][j] = f[i][j] | (f[i][k] & f[k][j]);
        }
    }
}
```

许多算法竞赛题目看起来和图没什么关系，但是如果可以找到状态顶点和连接不同顶点的边，也可以将其建立图模型，使用图论算法解决。

## 10.5 课后习题与实验

### 习题 1 最短路计数(洛谷P1144)

给出一有 $N$ 个顶点、$M$ 条边的无向无权图，顶点编号为 $1 \sim N$。问从顶点 $1$ 开始，到其他每个顶点的最短路有几条？输出答案对 $100003$ 取余后的结果。

**数据范围：** $1 \leqslant n \leqslant 10^6, 1 \leqslant m \leqslant 2 \times 10^6$，保证任意两点可相互到达。

### 习题 2 买药水(洛谷P1875)

给定 $n$ 种药水（从 $0$ 到 $n-1$ 编号），每种药水有相应的价格。同时给定若干条形如 $a\,b\,c$ 的配方，代表可以用一瓶 $a$ 药水加一瓶 $b$ 药水配置一瓶 $c$ 药水。求得到一瓶 $0$ 号药水的最低价格以及以最低价格得到 $0$ 号药水的不同方案数。

**数据范围：** $1 \leqslant n \leqslant 10^3$。

### 习题 3 通行费(洛谷P1462)

给定一张 $n$ 个顶点 $\nabla.m$ 条边的无向图，顶点代表城市，边代表道路。小 $A$ 需要从城市 $1$ 走到城市 $n$ 走过 $i$ 号城市需要付 $f_i$ 元；第 $j$ 条边连接 $a_j$ 和 $b_j$，走过时要消耗 $c_j$ 点体力。共有 $b$ 点体力，问在体力不消耗光的情况下，小 $A$ 经过的所有城市中最多的一次收取的费用的最小值是多少。若无法到达输出AFK。

**数据范围：** $1 \leqslant n \leqslant 10^4, 1 \leqslant m \leqslant 5 \times 10^4, 1 \leqslant f_i, c_i, b \leqslant 10^9$。

### 习题 4 最优贸易(洛谷P1073,NOIP2009提高组)

有 $n$ 座城市，之间有 $m$ 条道路连接，有些道路是单向的，有些是双向的。每个城市的水晶球价格为 $w_{ic}$，阿龙从 $1$ 号城市出发，沿着道路旅行，并经过某个城市购买一个水晶球，然后再经过另外一个城市卖掉它，最后到达第 $n$ 个城市。请问他最多可以获利多少?当无利可图时，他也可以不购买水晶球。

数据范围 $;2 \leqslant n \leqslant 10^5, 1 \leqslant m \leqslant 5 \times 10^5, 1 \leqslant w_i \leqslant 100$，

### 习题 5 小K的农场(洛谷P1993)

有 $n$ 个农场，给定 $m$ 个信息，以下列 $3$ 种形式描述：

1. 农场 $\alpha$ 比农场 $b$ 至少多种植了 $c$ 个单位的作物。
2. 农场 $a$ 比农场 $b$ 至多多种植了 $c$ 个单位的作物。
3. 农场 $\alpha$ 与农场 $b$ 种植的作物数一样多。

问：是否存在一种情况，使得农场的种植作物数量与上述 $3$ 种形式描述的所有信息吻合？数据范围：$1 \leqslant n, m \leqslant 5 \times 10^3$。

### 习题 6 糖果(洛谷P3275,四川省队选拔2011)

有 $n$ 位小朋友分糖果，提出 $k$ 个要求，要求以XAB形式描述，分为以下 $5$ 类：

1. 如果 $X=1$，表示第 $A$ 个小朋友分到的糖果必须和第 $B$ 个小朋友分到的糖果一样多。
2. 如果 $X=2$，表示第 $A$ 个小朋友分到的糖果必须少于第 $B$ 个小朋友分到的糖果。
3. 如果 $X=3$，表示第 $A$ 个小朋友分到的糖果必须不少于第 $B$ 个小朋友分到的糖果。
4. 如果 $X=4$. 表示第 $A$ 个小朋友分到的糖果必须多于第 $B$ 个小朋友分到的糖果。
5. 如果 $X=5$. 表示第 $A$ 个小朋友分到的糖果必须不多于第 $B$ 个小朋友分到的糖果。

问至少需要几颗糖果才能满足条件？若无法满足输出 $-1$。数据范围： $1 \leqslant n, k \leqslant 10^5$。

**提示：** 本题需要使用 第12章 涉及的“缩点”方法。

### 习题 7 传送门(洛谷P6464)

有 $n$ 幢教学楼、$m$ 条双向通道，保证图连通。可以在两幢教学楼间建一个传送门，通过传送门走距离为 $0$。求在建一个传送门的情况下任意两幢教学楼间最短路线之和的最小值。

**数据范围：** $1 \leqslant n \leqslant 100, 1 \leqslant m \leqslant \frac{n(n-1)}{2}$

### 习题 8 电梯(洛谷P3403)

有一幢 $h$ 层高的摩天大楼，它的电梯可以向上移动 $x$ 层，也可以向上移动 $y$ 层，还可以向上移动 $z$ 层。现在访客在第一层，请问他可以乘坐电梯前往的楼层数量。

**数据范围：** $1 \leqslant h \leqslant 2^{63}-1, 1 \leqslant x, y, z \leqslant 10^5$

### 习题 9 灾后重建(洛谷P1119)

有 $n$ 个村庄（编号从 $0$ 到 $n-1$）、$M$ 条双向公路，并给出第 $i$ 个村庄在地震后重建完成的时间 $t_i$ 若 $t_i$ 为 $0$ 则说明一开始就可以通车。之后有 $Q$ 个询问xyt，求在第 $t$ 天，从村庄 $x$ 到村庄 $y$ 的**最短路径**长度。若 $x$ 或 $y$ 尚未重建好或无法通过重建好的村庄到达，则输出 $-1$。

**数据范围：** $1 \leqslant n \leqslant 200, 1 \leqslant M \leqslant \frac{n(n-1)}{2}, 1 \leqslant Q \leqslant 5 \times 10^4$。

### 习题 10 产生数(洛谷P1037,NOIP2002普及组)

给出一个整数 $n(n < 10^{30})$ 和 $k(k \leqslant 15)$ 个变换规则。变换规则是指一位数可变换成另一个一位数，而且不能变为 $0$。例如，将数字中的一位 $2$ 变为数字 $5$ 就是一个合法的规则。请求出从 $n$ 开始，经过任意次的变换（$0$ 次或多次），能产生出多少个不同的整数。

### 习题 11 奶牛比赛(洛谷P2419,USACO2008年8月赛)

有 $N(1 \leqslant N \leqslant 100)$ 头奶牛按 $1, 2, \cdots, N$ 依次编号。每头奶牛的都有能力值 $x_i$ 且互不相同。如果编号为 $A$ 的奶牛的能力强于编号为 $B$ 的奶牛，那么编号为 $A$ 的奶牛获胜。现在想知道奶牛能力的具体排名，于是找来了奶牛们所有 $M(1 \leqslant M \leqslant 4500)$ 轮比赛的结果，保证不会自相矛盾。根据这些信息，求出可以确定能力值准确排名（$1$ 到 $N$ 名）的奶牛数目。

### 习题 12 社交网络(洛谷P2047,NOI2007)

在一个社交圈子里有 $n$ 个人，人与人之间有不同程度的关系。将这个关系网络对应到一有 $n$ 个顶点的无向图上，两个不同的人若互相认识，则在他们对应的顶点之间连接一条无向边，并附上一个正数权值 $c$。 $c$ 越小，表示两个人之间的关系越密切。可以用对应顶点之间的最短路长度来衡量两个人 $s$ 和 $t$ 之间的关系密切程度，注意到最短路径上的其他顶点为 $s$ 和 $t$ 的联系提供了某种便利，即这些顶点对于 $s$ 和 $t$ 之间的联系有一定的重要程度。

可以通过统计经过一个顶点 $v$ 的最短路径的数目来衡量该顶点在社交网络中的重要程度。考虑到两个顶点 $A$ 和 $B$ 之间可能会有多条最短路径，修改重要程度的定义如下：令 $C_{s,t}$ 表示从顶点 $s$ 到 $t$ 的不同的最短路的数目， $C_{s,t}(v)$ 表示经过顶点 $v$ 从顶点 $s$ 到 $t$ 的最短路的数目；则定义：

$$I(v)=\sum_{s \neq v, t \neq v} \frac{C_{s,t}(v)}{C_{s,t}}$$

为顶点 $v$ 在社交网络中的重要程度。社交网络都是连通的无向图，即任意两个顶点之间都有一条有限长度的最短路径。现在给出这样一幅描述社交网络的加权无向图，求出每个顶点的重要程度。

**数据范围：** $1 \leqslant n \leqslant 100, 1 \leqslant m \leqslant 4.5 \times 10^3, 1 \leqslant c \leqslant 10^3$，保证给出的无向图连通，且任意两个顶点之间的最短路径数目不超过 $10^{10}$

# 第11章 最小生成树

试想要在若干城市之间建立若干条高速公路，这些高速公路可以建成直线，连接在两个城市之间。如何规划这些高速公路使建造距离总和最短，就是一种典型的**最小生成树**问题。

树是一种特殊的图，具有很多特殊的性质。生成树问题研究的是将图中的所有顶点保留，但只选择图中的部分边，得到一棵树（也就是图的生成树）的问题。最小生成树则是在这些生成树中，边权之和最小的生成树。可以使用Prim算法或者Kruskal算法求解最小生成树。最小生成树思维导图如图 11-1 所示。

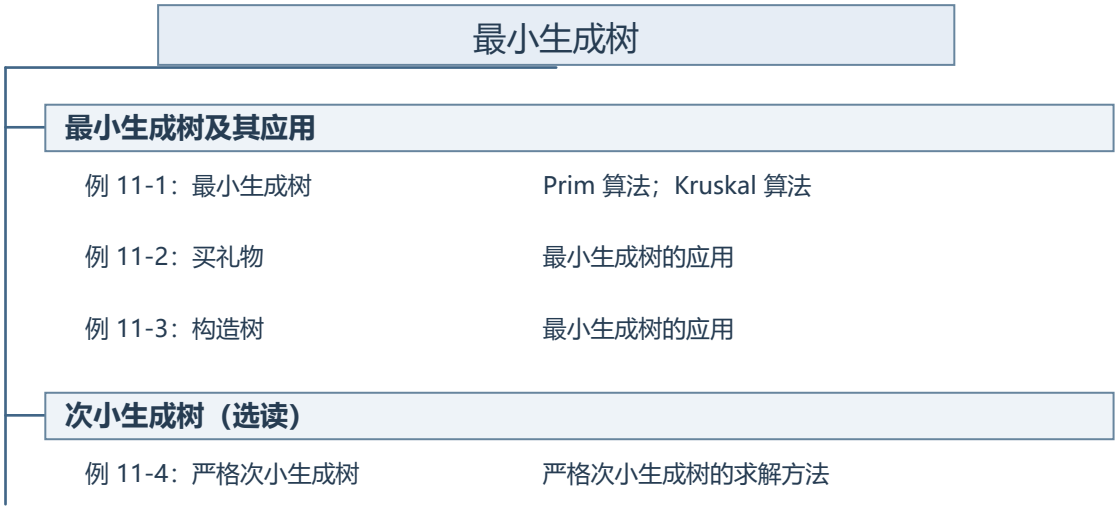



图11-1 本章思维导图

## 11.1 最小生成树及其应用

### 例 11-1 最小生成树(洛谷P3366)

生成树的定义为从一有 $n$ 个顶点的连通图中选择 $n-1$ 条边与这 $n$ 个顶点组成的树。**最小生成树**定义为所有生成树中边权之和最小者（可能存在多棵）。

现有一个含 $n$ 个顶点、$m$ 条边的无向图，求其最小生成树的边权之和；若图不连通，则输出 `orz`。

**输入格式：**第 $1$ 行包含两个整数 $n,m$。

接下来 $m$ 行每行 $3$ 个整数 $u_i,v_i,w_i$，表示存在一条从顶点 $u_i$ 到 $v_i$ 的无向边，边权为 $w_i$ $O$

**输出格式：**一行一个整数，表示最小生成树的边权之和；如果图不连通，则输出 `orz`。

**输入样例：**

```
4 6
1 4 4
1 3 5
1 2 2
2 3 2
2 4 1
3 4 3
```

**输出样例：**

```
5
```

**数据范围：** $1 \leqslant n \leqslant 5 \times 10^3, 1 \leqslant m \leqslant 2 \times 10^5, 1 \leqslant z \leqslant 10^4$.

**分析：**最简单的想法是，通过搜索去寻找所有的生成树，选取其中最小者。然而由于生成树数量较多，枚举顺序无序，容易导致程序运行过慢或超时。

因此，需要使用最小生成树算法。本章将介绍两种算法，供读者学习掌握。

解法 1（Prim算法）：不妨先假设图连通。由于所有顶点最后都会在最小生成树中，所以不妨任意设置一个起点（这里设为 1 号点）。该算法的大体思路是从 1 号点开始通过贪心算法不断向外扩展当前的树，直到所有顶点都进入生成树中。以上方的样例为例，用一个数组 `dis` 来记录到目前为止当前树中的点到其他点的最短边长。

1. 在初始状态下，先将整个数组 `dis` 设为 `inf`，表示未更新，如图 11-2(a) 所示。
2. 先将 1 号点放入生成树中并打上标记。此时当前的树中只有一个点，没有边。接着通过 1 号点来更新数组 `dis`，结果如图 11-2(b) 所示。这里有一个结论：任何时刻，未标记的点中 `dis` 最小者，其 `dis` 值所对应的边一定在一棵最小生成树中。关于这一结论的正确性将在后面证明。
3. 容易发现，此时 2 号点的 $dis$ 值是没有标记的点中最小的。根据结论，2 号点的 `dis` 值所对应的边（即 1 到 2 的边）一定在最小生成树中。将 2 号点打上标记并用与 2 号点相连的边来更新数组 `dis`，结果如图 11-2(c) 所示。
4. 此时 4 号点的 `dis` 值是没有标记的点中最小的。根据结论，4 号点的 `dis` 值所对应的边（即 2 到 4 的边）一定在最小生成树中。此时，将 4 号点放进最小生成树（打上标记），并用与其相连的边来更新数组 `dis`，结果如图 11-2(d) 所示。
5. 最后，剩余一个 3 号点。根据结论，3 号点 `dis` 值所对应的边（即 2 到 3 的边一定在最小生成树中。将 3 号点放入最小生成树中并打上标记，程序结束。最终得到的最小生成树如图 11-2(e) 所示。

在求最小生成树的同时累加生成树各边的边权，可以得到答案为 **5**。

证明刚才所提到的结论：任何时刻，未标记的点中 `dis` 最小者，其 `dis` 值所对应的边（设为 $e_1$）一定在一棵最小生成树中。

可以采用反证法。假设最小生成树已经建完，此时再将 $e_1$ 边加入最小生成树中，则树中出现了一个环。由于这条边连接的是一个已标记的点与一个未标记的点，则环中显然还有一条边 $e_2$ 也是连接一个已标记的点与一个未标记的点。此时用 $e_1$ 对 $e_2$ 进行替换，得到的结果一定更优。

| 编号 | dis |
|---|---|
| 1 ✓ | 0 |
| 2 ✓ | 2 |
| 3 | 2 |
| 4 | 1 |

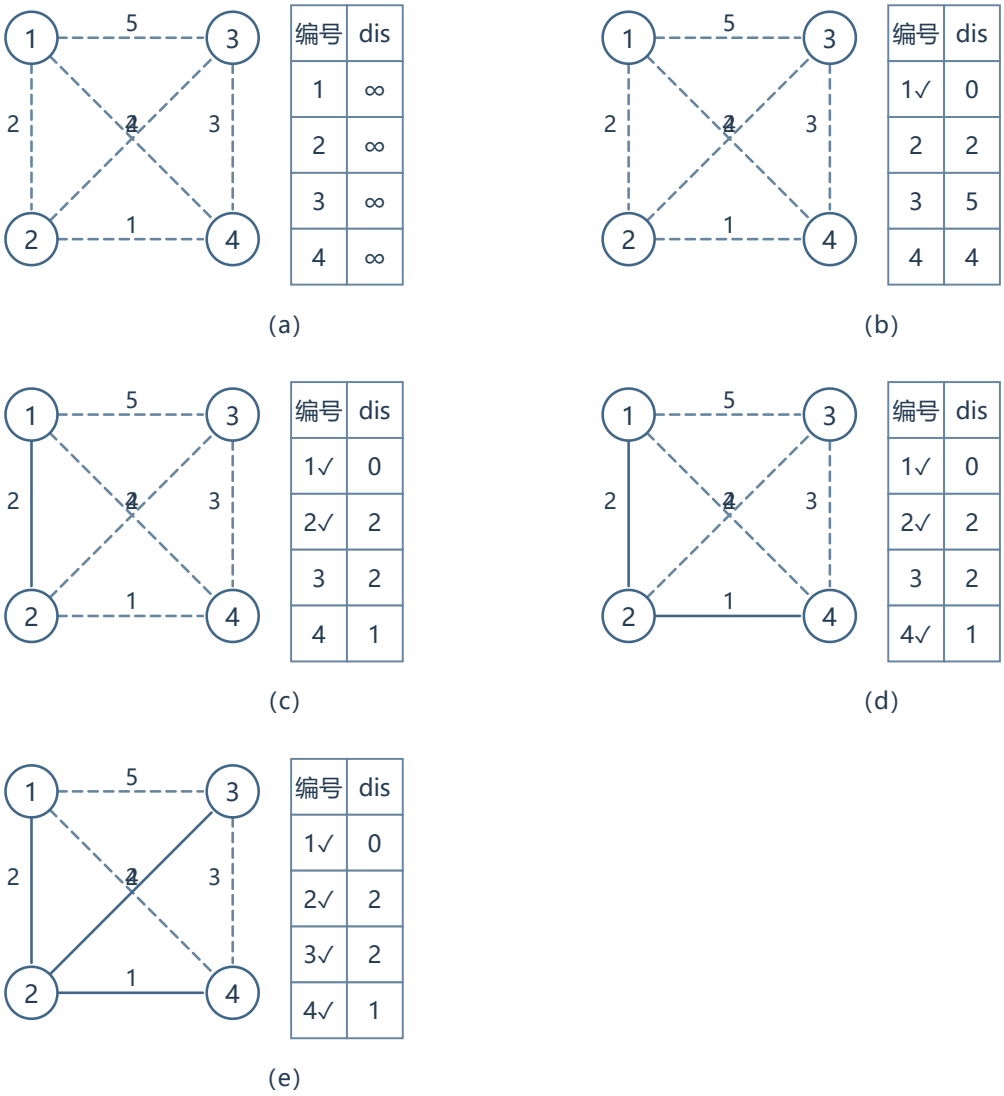



图11-2 Prim算法的步骤

如果此时的 $e$，边权和e2 相等，那将 $e_1e_2e_1$ 用 $e_2$ 替换同样是一棵最小生成树。对于这种情况两条边可以任选一条放入最小生成树中，对答案没有影响。这也印证了题目中最小生成树可能不止一棵的说法。

这种求最小生成树的方法就是Prim算法，其过程概括如下：

1. 将 1 号点加入当前生成树中并打上标记，同时更新数组 `dis`。
2. 选择所有未标记的点中 `dis` 最小的点。
3. 将该点加入当前生成树中并打上标记。
4. 使用与该点相连的所有连边更新数组 `dis`。
5. 重复第 2 步到第 4 步，直到所有点都已经被标记或所有未标记的点都不存在连边与已标记的点直接相连。

需要注意的是，如果出现所有未标记的点都不存在连边与已标记的点直接相连的情况，就意味着图不连通，此时也应当结束程序。可以再记录一个变量 `tot`，代表打上标记的点的数量。如果最后 `tot` $\neq n$，则可以直接判定无解。代码如下：

```
#include <bits/stdc++.h>
#define MAXN 400500
using namespace std;
struct edge {
    int to, nxt, val;
} e[MAXN];
int h[MAXN], cnt, dis[MAXN], tot, ans;
bool vis[MAXN];
void add(int from, int to, int value) {
    cnt++;
    e[cnt].nxt = h[from];
    e[cnt].to = to;
    e[cnt].val = value;
    h[from] = cnt;
}
int n, m;
void prim() { // 算法主体部分
    for (int i = 1; i <= n; i++)
        dis[i] = 2147483647; // 初始化数组 dis
    dis[1] = 0;
    dis[0] = 2147483647;
    while (true) {
        int u = 0;
        for (int i = 1; i <= n; i++) // 寻找 dis 最小的点
            if (!vis[i] && dis[i] < dis[u])
                u = i;
        if (!u)
            break;
        tot++;
        vis[u] = 1;
        ans += dis[u]; // 将边加入生成树并将点标记
        for (int i = h[u]; i; i = e[i].nxt) {
            int v = e[i].to;
            int w = e[i].val;
            if (dis[v] > w) {
                dis[v] = w;
                // 更新数组 dis
            }
        }
    }
```

```
}
int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int a, b, c;
        cin >> a >> b >> c;
        add(a, b, c);
        add(b, a, c);
    }
    prim();
    if (tot == n)
        cout << ans;
    else
        puts("orz ");
    return 0;
}
```

容易分析得出这段程序的时间复杂度为 $O(n^2+m)$，在本题中可以通过。但如果 $n$ 较大，则可能出现超时的情况。

容易发现，这一算法与 第10章 中的Dijkstra算法十分相似，于是可以采用一个类似于Dijk-stra算法的堆来进行优化，每次更新数组 `dis` 后将 `dis` 值与对应点存入一个**优先队列**，依据 `dis` 值排序，这样就可以将时间复杂度优化到 $O(m\log n)$。代码如下：

```
struct node {
    int pos, dis;
    friend bool operator<(node a, node b) { return a.dis > b.dis; }
} tmp;                   // 用于存储的结构体
priority_queue<node> q; // 用于寻找dis 最小的点的优先队列
void prim() {
    for (int i = 1; i <= n; i++) {
        dis[i] = 2147483647;
    }
    dis[1] = 0;
    tmp.dis = 0;
    tmp.pos = 1;
    q.push(tmp);
    while (!q.empty()) {
        tmp = q.top();
        q.pop(); // 取出 dis 最小的点
        int u = tmp.pos, d = tmp.dis;
        if (vis[u])
            continue; // 如果已经标记过就跳过
        tot++;
        vis[u] = 1;
        ans += dis[u];
        for (int i = h[u]; i; i = e[i].nxt) {
            int v = e[i].to;
```

```
            int w = e[i].val;
            if (dis[v] > w) {
                tmp.dis = dis[v] = w;
                tmp.pos = v;
                q.push(tmp); // 更新成功后更新优先队列
            }
        }
    }
}
```

解法 2（Kruskal算法）：假设图连通，该算法的主体思路是直接对边进行排序，然后从小到大进行合并。可以想象，在程序运行的过程中将会出现许多的连通块，并且连通块的数量会越来越少。在最后它们会合并为同一个连通块，即最终的最小生成树。

这里选用一个新的例子进行阐释，如图 11-3(b) 所示。

1. 最开始时，所有的点和边都没有被选中。这里同样有一个结论，任何时刻，所有未使用的边中边权最小的一条，如果它所连接的两个点不在同一个连通块中，则这条边一定在最小生成树中。该结论的证明将在后面给出。此时，将所有的边按照长度从小到大排序，如图 11-3(a) 所示。
2. 选择目前所有边中边权最小的一条（即 3 到 5 的边）。由于 3 号点和 5 号点目前不在同一个连通块中，根据结论，这条边一定在一棵最小生成树中，如图 11-3(c) 所示。
3. 类似的，接下来枚举到 1 号点和 2 号点的连边，以及 3 号点和 4 号点的连边。根据结论，这两条边也在最小生成树中，如图 11-3(d) 所示。
4. 枚举到的是 4 号点和 5 号点的连边。然而此时发现，4 号点和 5 号点已经连通，所以这条边不在最小生成树中，可以直接将它舍弃，如图 11-3(e) 所示。
5. 枚举到连接 1 号点和 3 号点的连边。将这条边加入最小生成树，发现整张图已经连通。最终结果如图 11-3(f) 所示。

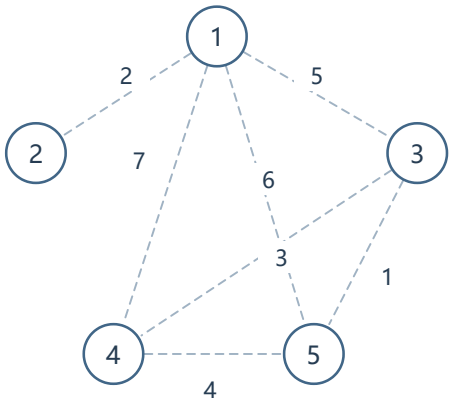
下面再证明刚才所提到的结论：所有未使用的边中边权最小的一条（设为 $e_1$） ，如果它所连接的两个点不在同一个连通块中，则这条边一定在一棵最小生成树中。

类似于Prim算法的证明，假设最小生成树已经建完，此时再将e1 边加入最小生成树中，则 $e_1$ 树中出现了一个环。由于这条边连接的是两个未连通的点，则环中显然还有一条边e2 也未被标 $e_2$ 记。由于e1 的边权为未标记的边中边权最小者矛盾，所以以e1 替换e2 显然更优。同理，若 $e_1e_1e_2e_1$ 与 $e_2$ 边权相同，则互相替换不影响答案。

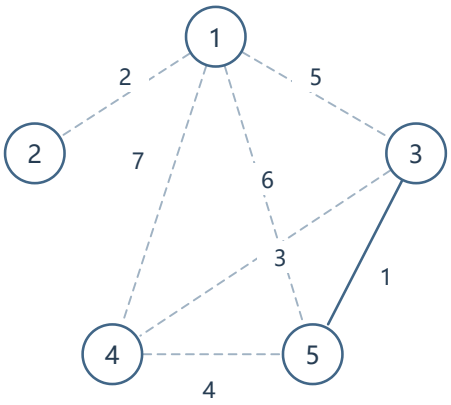
对于连通块的记录，可以使用一个并查集进行处理。最初每个点自成一个集合，每当一条边加入生成树时就合并其所连接的两个点所属的集合即可。对于无解的判断，只需要判断合并的次数是否为 $n-1$ 次即可。

| 边 | 长度 |
| --- | --- |
| 3—5 | 1 |
| 1—2 | 2 |
| 3—4 | 3 |
| 4—5 | 4 |
| 1—3 | 5 |
| 1—5 | 6 |
| 1—4 | 7 |

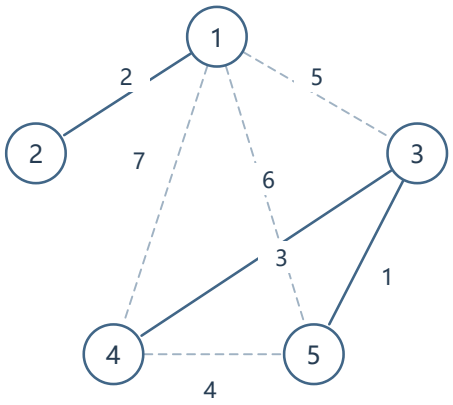
(a)



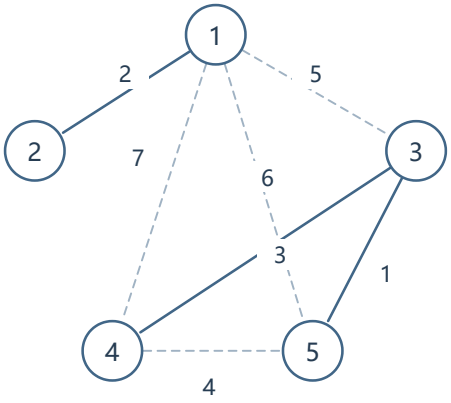

(b)



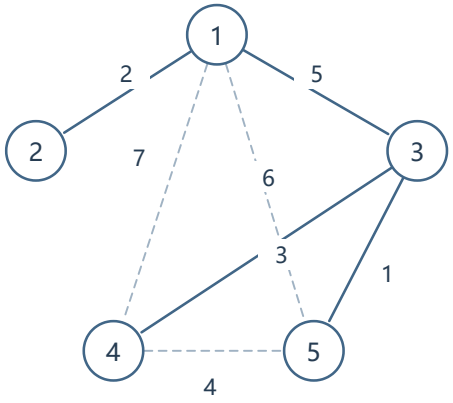

(c)




(d)




(e)




(f)

图11-3 Kruskal算法的步骤

这种求最小生成树的算法是Kruskal算法，其过程概括如下：

1. 将所有的边从小到大排序。
2. 将所有点分别放入各自的并查集中。
3. 选择所有未使用的边中边权最小的。
4. 若该边所连接的两个点已经连通，则舍去，否则合并这两个并查集。
5. 重复第 3 步到第 4 步，直到所有点都已经被标记或所有未标记的点都不存在连边与已标记的点直接相连。

因为Kruskal算法的瓶颈在于将所有的边排序，其复杂度为 $O(m\log m)$。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
struct edge {
    int u, v, w;
} e[200005];
int fa[5005], n, m, ans, cnt;
bool cmp(edge a, edge b) { return a.w < b.w; }
int find(int x) {
    if (x != fa[x])
        return fa[x] = find(fa[x]);
    return x;
} // 并查集寻找根结点函数(含路径压缩)
void kruskal() {
    sort(e + 1, e + m + 1, cmp); // 将边排序
    for (int i = 1; i <= m; i++) {
        int u = find(e[i].u);
        int v = find(e[i].v);
        if (u == v) { // 判断是否属于连通块
            continue;
        }
        ans += e[i].w; // 将边加入最小生成树并统计答案
        fa[v] = u;     // 合并两个并查集,表示已经连通
        cnt++;
    }
}
int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
        fa[i] = i;
    for (int i = 1; i <= m; i++)
        cin >> e[i].u >> e[i].v >> e[i].w;
    kruskal();
    if (cnt == n - 1)
        cout << ans;
    else
        puts("orz ");
    return 0;
}
```

在稠密图中，原版的 Prim 算法复杂度较优，但在稀疏图中，堆优化的 Prim 以及 Kruskal算法表现会更好。在应用时，读者可以根据实际情况选择最优的算法。

介绍完基础算法，下面来看几个最小生成树的例题。

### 例 11-2 买礼物(洛谷P1194)

需要购买 $B$ 件物品，每件原价 $A$ 元。有若干促销活动：若已购买第 $i$ 件，再购买第 $j$ 件只需花费 $K_{i,j}$ 元。$K_{i,j}=0$ 表示这两件物品之间没有促销活动。求购买所有物品的最少开销。

**数据范围：** $1\leqslant B\leqslant 500, 0\leqslant A,K_{i,j}\leqslant 10^3$。数据保证 $K_{i,j}=K_{j,i}$ 且 $K_{i,i}=0$

**分析：**由于涉及两点之间的关系，考虑建图。根据各点间的优惠关系来建图，边权就代表优惠后的价格。例如假设 $K_{i,j}=a$ 就连一条 $i$ 到、长度为 $a$ 的无向边。而对于原价购买，可以建立一个新的源点（设为 0 号点）。从 0 号点向每条边连一个边权为 $A$ 的边，代表原价购买这件物品。

在建图之后可以发现。一条从 0 号点开始的路径就代表着以路径的边权和购买了路径上的所有物品。例如一条 0→2→1 的路径就表示以原价购买 2 号，再以 $K_{2,1}$ 的价格购买 1 号。于是购买所有物品，就等价于寻找若干条从 0 号点开始的路径，且路径包含所有的点。实际上这些路径合起来也就是一棵生成树。由于要使价格最低，也就是路径边权之和最小，于是只要直接求**最小生成树**，得到的答案就是最终的结果，代码如下。

**代码片段（需结合上文的声明和函数）：**

```
int fa[5005], B, A, cnt, ans;
int main() {
    cin >> A >> B;
    for (int i = 1; i <= B; i++) {
        fa[i] = i;
        e[++cnt].v = i;
        e[cnt].w = A;
    } // 插入源点"0"(由于数组定义在全局变量,e[cnt].u默认值为0,无须更改)
    for (int i = 1; i <= B; i++) {
        for (int j = 1; j <= B; j++) {
            int w;
            cin >> w;
            if (w) {
                e[++cnt].u = i;
                e[cnt].v = j;
                e[cnt].w = w;
            } // 根据优惠规则建图
        }
    }
    kruskal();
    cout << ans;
    return 0;
}
```

### 例 11-3 构造树(洛谷CF472D)

一棵有 $n$ 个结点的树，其中每条边有一个正整数边权，但并不知道这棵树的样子，只知道每两点间最短路长度，但是这些最短路长度可能是错误的。请判断是否存在一棵树满足条件。

**数据范围：** $1\leqslant n\leqslant 2\times 10^3$。

**分析：**按照两两点对之间的距离建图。显然，如果这 $n$ 个点可以构成一棵树，那么这棵树一定是该图的生成树。进一步讨论这棵树的性质。对于相隔距离较大的点，它们的路径上很有可能会经过其他点，但对于相隔距离较小的点，路径上经过其他点就会比较困难。不难理解，这棵树应该是由若干相隔距离较小的点直接相连构成的，所以其边权和应该很小。这时可以猜测，这棵树可能就是原图的**最小生成树**。

事实上，回顾Kruskal算法的过程。一条点 $u$ 到 $v$、边权为 $w$ 的边属于最小生成树，当且仅当所有边权小于 $w$ 的边组成的图没能使 $u$、$v$ 两个点连通。这也就意味着在最后的生成树上这两点的路径一定会经过至少一条边权大于或等于 $w$ 的边，这样两点之间的距离一定大于或等于 $w$，当且仅当点 $u$ 与 $v$ 直接相连才能取等。因此只有在这棵树是最小生成树的时候，它才可能是满足条件的树。

由于是稠密图，在求生成树时，使用原版Prim算法复杂度更优。在维护 `dis` 的时候，再维护一个数组，代表 `dis` 值表示的边从哪里来，与 `dis` 值一同更新。在每个点打上标记之前进行连边，这样在算法结束的时候最小生成树也就建好了。接着只需要从每个点开始DFS遍历验证，查看各点间距离是否与输入数据一致即可。

最终时间复杂度为 $O(n^2)$，代码如下：

```
#include <bits/stdc++.h>
#define MAXN 4005
using namespace std;
int n, cnt, lst[MAXN], h[MAXN]; // lst[i] 表示 dis[i] 对应边的起点,数组 h 与下方
// 数组 e 存储图
int dis[MAXN], d[MAXN][MAXN]; // 数组 d 为原图
bool vis[MAXN];
struct edge {
    int nxt, to, val;
} e[MAXN * 2]; // 用于存储最小生成树的数组
void add(int u, int v, int w) {
    e[++cnt].nxt = h[u];
    e[cnt].to = v;
    e[cnt].val = w;
    h[u] = cnt;
}
void prim() { // Prim 算法求最小生成树
    for (int i = 1; i <= n; i++) {
        dis[i] = 2147483647;
    }
    dis[1] = 0;
    dis[0] = 2147483647;
    int tot = 0;
    while (true) {
        int u = 0;
        for (int i = 1; i <= n; i++)
            if (!vis[i] && dis[i] < dis[u])
                u = i;
        if (!u)
            break;
        vis[u] = 1;
```

```
        if (u != 1) {
            add(u, lst[u], dis[u]);
            add(lst[u], u, dis[u]);
            // 由于u到lst[u]的连边在最小生成树中,因此在这两点之间连一条边。需要特判1号点
        }
        for (int v = 1; v <= n; v++) {
            if (vis[v])
                continue;
            int w = d[u][v];
            if (dis[v] > w) {
                dis[v] = w;
                lst[v] = u;
            }
        }
    }
}
void dfs(int u, int fa) { // DFS 判断求两点间距离,用于验证
    for (int i = h[u]; i; i = e[i].nxt) {
        int v = e[i].to, w = e[i].val;
        if (v == fa)
            continue;
        dis[v] = dis[u] + w;
        dfs(v, u);
    }
}
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            scanf("%d", &d[i][j]);
            if (i != j && d[i][j] == 0) {
                puts("No");
                return 0;
            }
            if (i == j && d[i][j] != 0) {
                puts("No ");
                return 0;
            } // 特判
        }
    }
    prim();
    for (int i = 1; i <= n; i++) {
        dis[i] = 0;
        dfs(i, 0); // DFS 求两点间距离后验证
        for (int j = 1; j <= n; j++)
            if (dis[j] != d[i][j]) {
                puts("No");
                return 0;
            }
    }
    puts("YES ");
    return 0;
}
```

## 11.2 次小生成树（选读）

### 例 11-4 严格次小生成树(洛谷P4180,BJWC 2010)

设一张图的**最小生成树**边权之和为 $S$，则该图的严格次小生成树定义为该图所有边权之和大于 $S$ 的生成树中边权之和最小者（可能不存在，也可能存在多棵）。

现给出一张有 $n$ 个点、$m$ 条边的无向图，边权为 $w_i$，求出该图的严格次小生成树边权之和。数据保证原图存在严格次小生成树。

**数据范围：** $n \leqslant 10^5, m \leqslant 3 \times 10^5, 0 \leqslant w_i \leqslant 10^9$

**分析：**一种简单的思路是尝试找到原图的所有生成树，然后通过比较得出答案。但由于生成树数量过多，这样的算法显然不可行。

由于严格次小生成树的边权和仅大于最小生成树边权和，因此可以猜测，严格次小生成树很可能就是在最小生成树上替换一条或几条边得到的。事实上，可以证明，一定存在一棵严格次小生成树，使得它与某棵最小生成树仅有一条边的差距。

设最小生成树为 $T$，严格次小生成树为 $T'$，总权值分别为 $S$ 和 $S_1$，其中 $S_1 > S$。根据生成树的交换性质，可以将 $T' \setminus T$ 中的边与 $T \setminus T'$ 中的边配对，使每对边交换后仍得到一棵生成树。

因为 $T$ 是最小生成树，每次这样的交换，其权值增量都非负；所有增量的总和为 $S_1 - S > 0$。因此至少有一次交换的增量 $\delta$ 满足 $0 < \delta \leq S_1 - S$。

这次交换得到的生成树权值为 $S + \delta$，严格大于 $S$，且不超过 $S_1$。由严格次小生成树的定义，它只能等于 $S_1$。因此，存在一棵严格次小生成树，可以由最小生成树替换一条边得到。

有了这条性质，就可以考虑在建完最小生成树之后，寻找那条不属于最小生成树、但属于严格次小生成树的边。枚举每一条非树边，在加入这条边之后，生成树上出现了一个环。再断掉环中其他边里面边权最大的边（若该边的边权与环内其他边中边权最大者相等，则断掉边权次大者，注意可能不存在边权次大的边），那么就得到了包含这条边的生成树中权值最小者。将所有上述生成树权值和取min后，就可以得到最终的答案。

采取树上**倍增**的方法，定义 1 号点为根，并存储每个点向上 $2$ 条边的最大值与次大值。在 $2^i$ 寻找时，通过倍增取出这些最大值与次大值并依次进行更新，就可以得到需要断的边的权值。注意在存储和寻找次大值时的分类讨论。

由于是稀疏图，最小生成树采用复杂度为 $O(m \log m)$ 的Kruskal算法或是堆优化Prim算法。最终时间复杂度为 $O(n \log n + m \log n)$。代码如下：

```
#include <bits/stdc++.h>
#define MAXN 100007
#define MAXM 300007
using namespace std;
struct edge {
    int nxt, to, val;
} e[MAXN << 1];
int h[MAXN], cnt; // 存储最小生成树的数组
```

```
void add(int u, int v, int w) {
    e[++cnt].nxt = h[u];
    e[cnt].to = v;
    e[cnt].val = w;
    h[u] = cnt;
}
struct edge1 {
    int u, v, w;
    bool used;
} e1[MAXM]; // 存储原图,其中 used表示是否在最小生成树中
int n, m, fa[MAXN];
long long ans0; // fa 用于Kruskal 算法的并查集,ans0 为最小生成树
// 边权和
bool cmp(edge1 a, edge1 b) { return a.w < b.w; }
int find(int x) {
    if (x != fa[x])
        return fa[x] = find(fa[x]);
    return x;
}
void kruskal() {
    sort(e1 + 1, e1 + m + 1, cmp);
    for (int i = 1; i <= m; i++) {
        int u = find(e1[i].u), v = find(e1[i].v);
        if (u == v)
            continue;
        ans0 += e1[i].w;
        add(e1[i].u, e1[i].v, e1[i].w);
        add(e1[i].v, e1[i].u, e1[i].w);
        e1[i].used = 1;
        fa[v] = u;
    } // 以上为 kruskal算法
}
int f[MAXN][20], mx[MAXN][20], mx2[MAXN][20], dep[MAXN];
// f/mx/mx2[u][i]代表u向上22层祖先边权最大值/边权次大值。dep存储深度
void dfs(int u) {
    dep[u] = dep[f[u][0]] + 1;
    for (int i = 1; i <= 18; i++) {
        f[u][i] = f[f[u][i - 1]][i - 1]; // 处理树上倍增。注意讨论及更新顺序
        if (mx[u][i - 1] == mx[f[u][i - 1]][i - 1]) {
            mx[u][i] = mx[u][i - 1];
            mx2[u][i] = max(mx2[f[u][i - 1]][i - 1], mx2[u][i - 1]);
        }
        if (mx[u][i - 1] > mx[f[u][i - 1]][i - 1]) {
            mx[u][i] = mx[u][i - 1];
            mx2[u][i] = max(mx[f[u][i - 1]][i - 1], mx2[u][i - 1]);
        }
        if (mx[f[u][i - 1]][i - 1] > mx[u][i - 1]) {
            mx[u][i] = mx[f[u][i - 1]][i - 1];
            mx2[u][i] = max(mx[u][i - 1], mx2[f[u][i - 1]][i - 1]);
        }
    }
    for (int i = h[u]; i; i = e[i].nxt) {
        int v = e[i].to, w = e[i].val;
        if (v == f[u][0])
            continue;
        f[v][0] = u;
```

```
        mx[v][0] ^= w;
        dfs(v);
    }
}
int lca(int u, int v) {
    if (dep[u] < dep[v])
        swap(u, v);
    for (int i = 18; i >= 0; i--)
        if (dep[u] - dep[v] >= (1 << i))
            u = f[u][i];
    if (u == v)
        return u;
    for (int i = 18; i >= 0; i--)
        if (f[u][i] != f[v][i]) {
            u = f[u][i];
            v = f[v][i];
        }
    return f[u][0];
    // 倍增求lca。
}
long long cal(int u, int v, int w) {
    int l = lca(u, v);
    int nmx = 0, nmx2 = 0; // nmx 为环中其他边中的最大者,nmx2 为次大者
    for (int i = 18; i >= 0; i--) {
        if (dep[f[u][i]] >= dep[l]) {
            if (nmx == mx[u][i])
                nmx2 = max(mx2[u][i], nmx2);
            if (nmx > mx[u][i])
                nmx2 = max(mx[u][i], nmx2);
            if (nmx < mx[u][i]) {
                nmx2 = max(mx2[u][i], nmx);
                nmx = mx[u][i];
            }
            u = f[u][i];
        }
        if (dep[f[v][i]] >= dep[l]) {
            if (nmx == mx[v][i])
                nmx2 = max(mx2[v][i], nmx2);
            if (nmx > mx[v][i])
                nmx2 = max(mx[v][i], nmx2);
            if (nmx < mx[v][i]) {
                nmx2 = max(mx2[v][i], nmx);
                nmx = mx[v][i];
            }
            v = f[v][i];
            // 树上倍增的同时更新 nmx与nmx2
        }
    }
    if (w != nmx)
        return ans0 - nmx + w;
    if (nmx2)
        return ans0 - nmx2 + w;
    return 0x7f7f7f7f7f7f7f7f;
    // 注意可能该边与环内最长边相等且环内不存在次长边,同时注意数据范围可能超过int
}
```

```
int main() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= m; i++) {
        cnt++;
        cin >> e1[cnt].u >> e1[cnt].v >> e1[cnt].w;
        if (e1[cnt].u == e1[cnt].v)
            cnt--; // 忽略自环
    }
    m = cnt;
    cnt = 0;
    for (int i = 1; i <= n; i++)
        fa[i] = i;
    kruskal();
    dfs(1);
    long long ans = 0x7f7f7f7f7f7f7f7f;
    for (int i = 1; i <= m; i++)
        if (!e1[i].used)
            ans = min(cal(e1[i].u, e1[i].v, e1[i].w), ans);
    cout << ans;
    return 0;
}
```

## 11.3 课后习题与实验

### 习题 1 营救(洛谷P1396)

一张有 $n$ 条边、$m$ 个顶点的无向图，每条边有一个拥挤程度。求一条从顶点 $s$ 到 $t$ 的路径，使得途径最拥挤的边的拥挤程度最小。

**数据范围：** $1 \leqslant n \leqslant 10^4, 1 \leqslant m \leqslant 2 \times 10^4, 1 \leqslant s, t \leqslant n$，保证从顶点 $s$ 出发一定能到达顶点 $t$

### 习题 2 天空(洛谷P1195)

给定一张有 $n$ 个顶点 ,$m$ 条边的无向图，每条边有一个边权。现要选择一部分边将其连成 $k$ 个连通块（每个连通块至少一个点），求使用的边的边权之和最小值。若不可能实现，则输出NoAnswer。

**数据范围：** $1 \leqslant n \leqslant 10^3, 1 \leqslant m \leqslant 10^4, 1 \leqslant k \leqslant 10$。

### 习题 3 灌溉(洛谷P1550,USACO2008年10月赛)

有 $n$ 块田。需要挖若干口井，并在各块田中修筑水道来连通以供水。在第 $i$ 号田中挖一口井需要花费 $W_i(1 \leqslant W_i \leqslant 10^5)$ 元。连接第 $i$ 号田与第 $j$ 号田需要 $P_{i,j}(P_{j,i} = P_{i,j})\overrightarrow{T}$。现需要保证所有田地有水，求最小的费用。

**数据范围：** $1 \leqslant n \leqslant 300, 1 \leqslant W_i \leqslant 10^s, 1 \leqslant P_{i,i} \leqslant 10^s$。

### 习题 4 货车运输(洛谷P1967,NOIP2013)

有 $n$ 座城市，编号从 $1$ 到 $n$，城市之间有 $m$ 条双向道路。每一条道路对车辆都有重量限制，简称限重。现在有 $q$ 辆货车在运输货物，司机们想知道每辆车在不超过车辆限重的情况下，最多能运多重的货物。

**数据范围：** $1 \leqslant n < 10^4, 1 \leqslant m < 5 \times 10^4, 1 \leqslant q < 3 \times 10^4, 0 \leqslant z \leqslant 10^5$ o

### 习题 5 逐个击破(洛谷P2700)

有 $n$ 座城市，其中 $k$ 座被敌军所占领。$n$ 座城市间有 $n-1$ 条公路相连，破坏其中某条公路的代价是已知的。现在，已知 $k$ 个敌军所在的城市，以及所有公路破坏的代价，请计算出花费最少的代价将这 $k$ 个城市互相隔离开，以便逐个击破。

**数据范围：** $2 \leqslant n \leqslant 10^5, 2 \leqslant k \leqslant n.$。

### 习题 6 电网(洛谷CF1245D)

已知有 $n$ 座城市，需要都通上电。一座城市有电，必须在这座城市有发电站或者和一座有电的城市用电缆相连。在城市 $i$ 建造发电站的代价是 $c_i, i$ 和 $j$ 两座城市相连的代价是 $(k_i + k_j) \times (|x_i - x_j| + |y_i - y_j|)$。求最小的价格使得所有城市都有电。

**数据范围：** $1 \leqslant n \leqslant 2 \times 10^3, 1 \leqslant x_i, y_i \leqslant 10^6, 1 \leqslant c_i, k_i \leqslant 10^9$

### 习题 7 免费道路(洛谷P3623,APIO2008)

给定一张有 $n$ 个点、$m$ 条边、边权为 $0$ 或 $1$ 的图，求一棵边权和为 $k$ 的**最小生成树**。若不存在，则输出 no solution。

**数据范围：** $1 \leqslant k < n \leqslant 2 \times 10^4, 1 \leqslant m \leqslant 10^5$。

### 习题 8 能量树(洛谷CF1120D)

给定一棵有根树，定义叶子结点的度为 $1$ 的结点。以 $w_u$ 的代价可以控制 $u$ 点。选择控制某个结点之后可以给它子树里的叶子都加上同一个整数 $t$。现在需要以最小的总代价使得：在叶子上任意放数，都可以把它们变成 $0$。

请输出最小的总代价，有多少结点可能被控制，以及每个可能被控制的结点的编号。

**数据范围：** $1 \leqslant n \leqslant 2 \times 10^5$。

# 第12章 连通性问题

在之前的章节中已经讨论过最短路问题和生成树问题。在无向图中，连通性问题可以使用并查集解决。但是当删去某个点或某条边时，如何判断图是否连通？对于无向图来说，有一些点或者边是“关键”的，没有这些点或者边，这个图就会分裂成各个部分。

无向图中连通的点可以相互到达，那么有向图中可以相互到达的点对又有什么性质呢？有序性是一个很好的性质，但对于多数错综复杂的有向图来说并不能直接进行**拓扑排序**。可以将一些可以互相到达的点合并成一个点，使这个有向图变为有向无环图，即可进行拓扑排序。

本章将分别介绍无向图的双连通性和有向图的强连通性。本章思维导图如图 12-1 所示。




图12-1 本章思维导图

## 12.1 无向图的边双连通性

### 例 12-1 切断线路(洛谷P1656)

有 $n$ 座城市，总共 $m$ 条铁路相连。任意两个城市都可以通过铁路直接或者间接到达。如果某条铁路被切断之后，某两座城市无法互通，这样的铁路就被称为关键路线。现在要使某两个城市无法互通，请问切断哪些铁路可以达成这个效果?

**分析：**在无向图中，如果删去一条边后，图不连通了，这说明存在两个点 $u$ 和 $v$，$u$ 到 $v$ 的路径必定经过这条边，那么把这条边叫作割边（或者桥）。如果一张图不存在割边，那么称这张图为边双连通图。一张图的极大边双连通子图被称作**边双连通分量**。一张图的边双连通分量之间是不相交的：如果两

个双连通分量相交了，那么删去其任意一条边，两个子图之间仍然连通。这说明点的“属于同一个双连通分量”的关系是带有传递性的。

对于边双连通还存在另一个等价定义：对于任意两点之间，存在至少两条边不相交的路径。如图 12-2(a) 所示，这个图中的割边有 3 条（加粗的边），双连通分量有 4 个，分别是{1，2，3}、{4}、{5}和{6，7，8}。

对于无向图，求出其所有双连通分量后，如果把每个双连通分量替换成一个点，那么剩下的是一棵树。这一点在取出其生成树上也有体现：每一个双连通分量在生成树上都是一个连通块，如图 12-2(b) 所示。

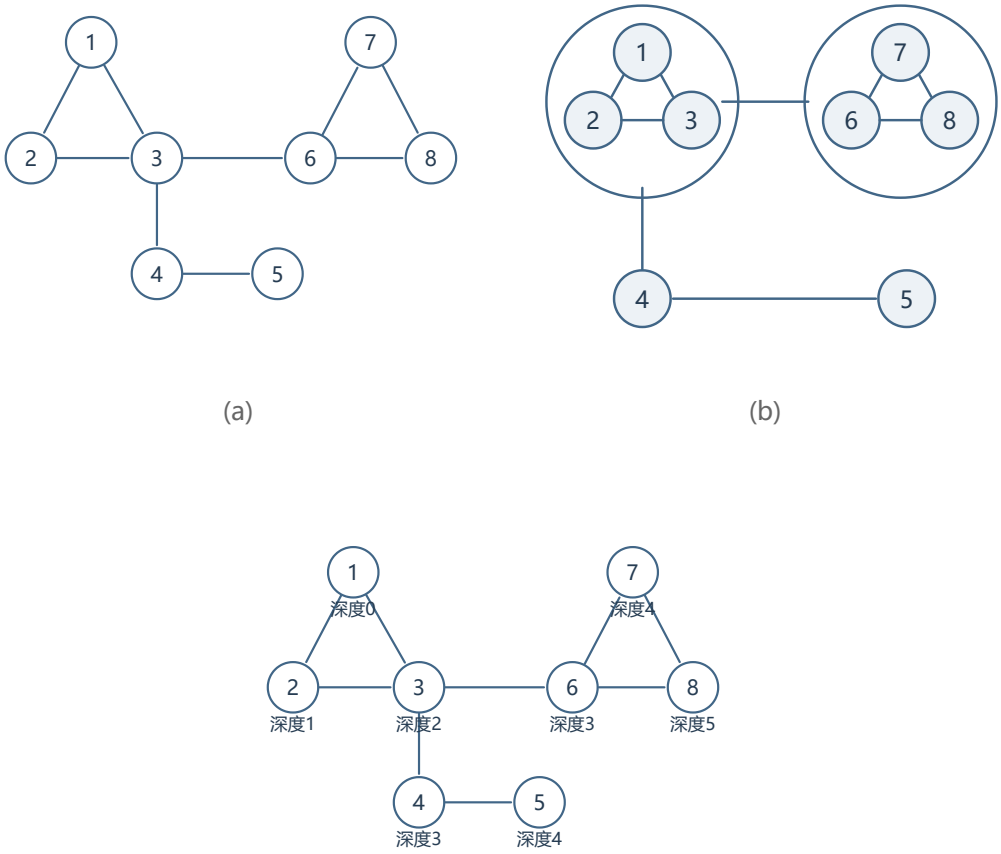



图12-2 无向图的边双连通性

对上图从 1 号结点开始，进行深度优先遍历，下次访问优先进入编号较小的结点，每个结点只会访问一次。第 $i$ 号结点访问到的顺序（也叫作DFS序或者时间戳）为 `dfn`，这里的结点的 $\mathbf{dim}_i$ DFS序刚好和结点编号一致。遍历时经过的边可以组成一棵树，即这个图的搜索树（也叫作DFS生成树），如图 12-2(c) 所示。每个结点通过搜索树到 1 号结点的距离就是这个结点的深度。

在生成树的基础上，额外的边的意义就是将树中的一条链归成同一个集合。有了这个转化，在不断加边的情况下，可以通过并查集维护树中的边双连通块。具体来说，记录每个点所属的集合，并且记录每个集合中深度最小的点，成为集合的顶点。在加入一条新的边（$u$，$v$）的时候，找到这条边对应的树中的

链，如果链的两个端点不属于同一个集合，那么取顶点深度较大的点 $u$，让 $u$ 和其父结点所在的集合合并。

这样做刚好能将链合并，并且不会合并多余的集合。因为多余的集合顶点深度只能小于两端点的LCA（**最近公共祖先**）所属集合顶点的深度，所以一旦被合并，说明两端点所在的集合顶点深度都小于LCA的深度，与两端点不属于同一个集合矛盾。

但是一般的题目图是给定后不再变化的，也就是可以通过树上差分，来得到每个点的“是否和父结点属于同一个集合”这一信息。因为将一条链上所有点合并实际上就是沿着链的每一条边合并，所以只需要标记每条边是否合并过。

这样实现的缺点是需要LCA，导致代码不好写。经过分析，额外的边有两种：

1. $u$ 和 $v$ 是祖先—子孙关系。
2. $u$ 和 v 所在的子树没有相交。

其中第 1 种类型不需要LCA，第 2 种被称作横叉边，这正是代码难写的主要原因。所以任务是找到一种方法，使得得到生成树后，额外的边中没有横叉边。

常见的方法是选定一个根，并且沿着这个根进行DFS，每次不断访问没到过的点，并把经过的边加入集合，这样得到的生成树就是DFS生成树，代码如下：

```
void dfs(int u) {
    vis[u] = true;
    for (int i = head[u]; i; i = nxt[i])
        if (!vis[to[i]]) {
            AddEdgeToTree(u, to[i]);
            dfs(to[i]);
        }
}
```

对于DFS生成树，是不存在横叉边的。因为如果存在横叉边 $(u,v)$，其中 $u$ 比 $v$ 更早被访问到，那么在DFS到 $u$ 的时候，这条边一定会被走过，与这条边没访问过矛盾了。

于是在得到DFS生成树后可以直接计算每条边是否要和祖先合并。当然在计算完之后直接使用并查集复杂度要乘上并查集的 $O(\alpha(n))$，所以需要更快且好写的方法。

考虑树上连通块的性质：记其点集DFS序最小为 $l$，最大为 $r$，那么其代表一个区间 $[l,r]$ 将树分成若干个连通块后，所有的连通块之间代表的区间只有相离和包含两种关系。也就是可以像DFS一样，用一个栈维护DFS到的点，当一个点是连通块深度最浅的点时，弹出栈顶的一段区间就是弹出这个连通块。

有了这些性质，可以在一遍DFS中直接求出所有的边双连通分量：维护每个点是否是连通块深度最低的点，即通过树形DP求出第 $i$ 个结点的子树，通过至多一条非树边能到的深度最低的点的DFS序，也就是 $low_i$。这个图的各个结点的 `low` 值如图 12-3 所示（实线是搜索树的边，虚线是其他边，粗线是割边）。需要注意的是，对于同一个图，遍历边的顺序不同，各点的 `dfn` 和 `low` 值也可能不同，但不影响求解答案。

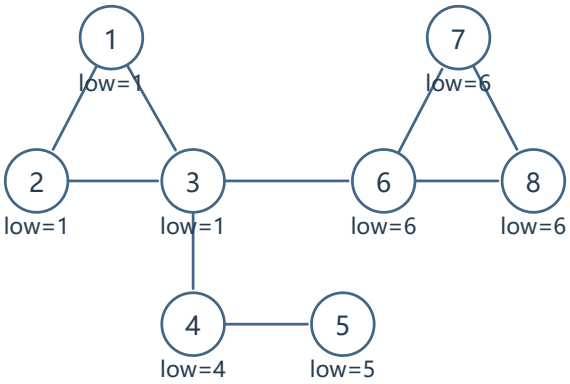



图 12-3 各点的 DFS 序和low 值

遍历完一个子树，当发现 $low_v > dfn_u$，说明结点 $v$ 后续不会接回 $u$ 的祖先上，（$u$，$v$）就是一条割边。遍历完毕后，发现 $low_u = dfn_u$ 即子树没有非树边能到达自己祖先的时候，就一直弹出栈的点，直到在栈中遇到自己为止，表示这些点属于同一个边双连通分量。本题代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 1e5 + 10;
int head[MAXN], nxt[MAXN << 1], to[MAXN << 1], tot = 1;
void addedge(int b, int e) {
    nxt[++tot] = head[b];
    to[head[b] = tot] = e;
    nxt[++tot] = head[e];
    to[head[e] = tot] = b;
}
pair<int, int> edge[MAXN];
// dfn 表示 DFS 序
// low表示子树通过至多一条非树边能到的最低深度点的DFS序
// st表示用来提取连通块的栈
// bel表示每个点属于的连通块
int dfn[MAXN], low[MAXN], st[MAXN], top;
int cnt, idx, bel[MAXN], anscnt = 0;
void dfs(int u, int lst) {
    low[u] = dfn[u] = ++cnt;
    st[++top] = u;
    for (int i = head[u]; i; i = nxt[i])
        if (i != (lst ^ 1)) {
            int v = to[i];
            // DFS时要注意防止走刚刚走过的边
            if (!dfn[v]) {
                dfs(v, i);
                // 当边是DFS树上的时候,直接取子树min
                low[u] = min(low[u], low[v]);
                // 发现割边
                if (low[v] > dfn[u])
                    edge[anscnt++] = make_pair(min(u, v), max(u, v));
            } else
```

```
                // 当边不是DFS树上的时候,更新到达的深度最低的点
                low[u] = min(low[u], dfn[v]);
        }
    // 当子树没有非树边能到达自己祖先的时候,自己就是连通块深度最低的点
    if (low[u] == dfn[u]) {
        int v;
        ++idx;
        do {
            v = st[top--];
            bel[v] = idx; // 给栈顶一段点标记为同一个边双连通分量
        } while (v != u);
    }
}
bool cmp(pair<int, int> x, pair<int, int> y) {
    if (x.first == y.first)
        return x.second < y.second;
    else
        return x.first < y.first;
}
int main() {
    int n, m;
    scanf("%d%d ", &n, &m);
    for (int i = 0; i < m; ++i) {
        int a, b;
        scanf("%d%d", &a, &b);
        addedge(a, b);
    }
    // 对于原图每个连通块都求一遍边双连通分量
    for (int i = 1; i <= n; ++i)
        if (!dfn[i])
            dfs(i, -1);
    sort(edge, edge + anscnt); // 对边的答案进行排序
    for (int i = 0; i < anscnt; ++i) {
        int a = edge[i].first, b = edge[i].second;
        printf("%d %d\n ", a, b);
    }
    return 0;
}
```

在求出边双连通分量以后，就得到了所有割边：一条边是割边当且仅当两端点在不同的边双连通分量。这个算法是Tarjan**强连通分量**算法的变种，关于Tarjan算法在后面会介绍。由于这个变种更加易于学习，所以先介绍无向图的双连通分量。

### 例 12-2 多余的路径(洛谷P2860,USACO2006年1月赛)

给一张有 $n$ 个点、$m$ 条边的连通无向图，求最少加多少条无向边，才能变成边双连通图。

**数据范围：** $n \leqslant 5000, m \leqslant 10000$，

**分析：**将所有**边双连通分量**看作一个点，那么得到的新图里的边全是原图的割边。也就是要加若干条边，使这些边覆盖的链的并集是所有割边。一个容易猜到的结论是，将新图的叶子两两配对，那么当新图点数不为 1 的时候，假设有 $k$ 个叶子，那么答案是 $\left\lceil \frac{k}{2} \right\rceil$。

对于答案的构造，就是取一个点作为根，最小化叶子数最多的子树的叶子数。可以证明对于点数有限的树，这个数量不大于 $\lceil \frac{k}{2} \rceil$，因为如果所有点的这个值大于 $\lceil \frac{k}{2} \rceil$，那么任选一个点，沿着叶子数最多的子树走，因为这个子树的叶子数 $> \lceil \frac{k}{2} \rceil$ 所以除去这个子树，本身的叶子数 $< \lceil \frac{k}{2} \rceil$ 所以走向新的点一定不会再走回来。

这样不断地走会经过无限多的不同点，与点数有限矛盾。

此时可以证明。一定有方案使得任意一对配对的叶子源自根的不同子树，因为每次可以取出剩余叶子最多的子树，与剩下的配对，此时记剩余叶子为 $k'$，那么剩余叶子最多的子树数量一定不会超过 $\lceil \frac{k'}{2} \rceil$

所以统计叶子个数就得到了答案，算法的时间复杂度为 $O(n+m)$。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 1e5 + 10;
int head[MAXN], nxt[MAXN << 1], to[MAXN << 1], tot = 1;
void adde(int b, int e) {
    nxt[++tot] = head[b];
    to[head[b] = tot] = e;
    nxt[++tot] = head[e];
    to[head[e] = tot] = b;
}
int dfn[MAXN], low[MAXN], st[MAXN], top;
int cnt, idx, bel[MAXN];
void dfs(int u, int lst) {
    low[u] = dfn[u] = ++cnt;
    st[++top] = u;
    for (int i = head[u]; i; i = nxt[i])
        if (i != (lst ^ 1)) {
            if (!dfn[to[i]]) {
                dfs(to[i], i);
                low[u] = min(low[u], low[to[i]]);
            } else
                low[u] = min(low[u], dfn[to[i]]);
        }
    if (low[u] == dfn[u]) {
        int v;
        ++idx;
        do
            bel[v = st[top--]] = idx;
        while (v != u);
    }
}
int deg[MAXN];
int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    for (int i = 1, b, e; i <= m; ++i)
        scanf("%d%d", &b, &e), adde(b, e);
    dfs(1, -1); // 由于图连通,只需要一次 DFS
    for (int i = 2; i <= tot; i += 2) {
        int u = to[i], v = to[i ^ 1];
```

```
        // 找出所有割边
        if (bel[u] != bel[v])
            ++deg[bel[u]], ++deg[bel[v]];
    }
    int ans = 0;
    // 统计叶子的个数
    // 当idx =1时实际上得到的个数是0,所以边界情况正确
    for (int i = 1; i <= idx; ++i)
        ans += deg[i] == 1;
    ans = (ans + 1) / 2;
    printf("%d \n ", ans);
    return 0;
}
```

## 12.2 无向图的点双连通性

### 例 12-3 割点(割顶)(洛谷P3388)

给出一张有 $n$ 个点、$m$ 条边的无向图，对每个连通分量求出割点。输出割点个数，并从小到 大输出所有割点。

**数据范围：** $1 \leqslant n \leqslant 2 \times 10^4, 1 \leqslant m \leqslant 1 \times 10^5$

**分析：**对应着割边，定义割点为删去一个点以及其相邻的边后，使得图不连通的点。如果一张图不存在割点，那么称这张图为点双连通图。类似的，极大点双连通子图被称作**点双连通分量**。

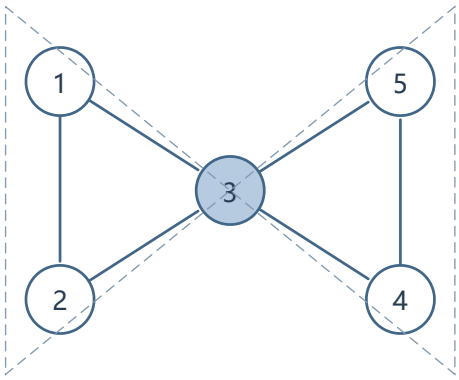



图12-4 无向图的点双连通性

如图 12-4 所示，{1，2，3}和{3，4，5}都是点双连通分量，两个点双连通分量的点集可能会相交，其中，结点 3 为割点。

对于点双连通还存在另一个等价定义：对于任意两点 $u$ 和 $v$ 之间，存在至少两条点不相交（不包含 $u$ 和 $v$）的路径。

如果两个点双连通分量的交大于 1，则删去任意一点其交仍大于或等于 1，且图连通，所以两个点双连通分量的交最大为 1。这说明一条边不可能同时属于两个点双连通分量，所以点双连通

可以看作是边之间带有传递性的关系。同样可以通过生成树来得到一张图所有的点双连通分量，对于非树边相当于将一条链上所有的边合并到一个集合。这说明点双连通分量仍然是树中的一个连通块。

下面首先介绍如何求解点双连通分量。类似**边双连通分量**的变种Tarjan算法，求点双连通分量可以直接计算边的DFS序。但是这些连通分量都是基于点的，所以修改Tarjan算法：用一个点代表其通向父结点的边，每次处理完点双连通分量深度最低的边 $(low_v = dfn_u)$ 或者处理完割边 $(low_v > dfn_u)$ 时，代表找到一个点双连通分量。

对于本题来说，统计每个点属于的点双连通分量的数量，如果一个点属于多个点双连通分量，那么它就是一个割点。本题代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 1e5 + 10;
int head[MAXN], nxt[MAXN << 1], to[MAXN << 1], tot = 1;
void addedge(int b, int e) {
    nxt[++tot] = head[b];
    to[head[b] = tot] = e;
    nxt[++tot] = head[e];
    to[head[e] = tot] = b;
}
int dfn[MAXN], low[MAXN], st[MAXN], top;
int cnt, deg[MAXN], idx;
vector<int> vdcc[MAXN];
void dfs(int u, int lst) {
    low[u] = dfn[u] = ++cnt;
    st[++top] = u;
    for (int i = head[u]; i; i = nxt[i])
        if (i != (lst ^ 1)) {
            int v = to[i], vv;
            if (!dfn[v]) {
                dfs(v, i);
                low[u] = min(low[u], low[v]);
                if (low[v] >= dfn[u]) {
                    // 找到新的点双连通分量
                    ++idx;
                    vdcc[idx].push_back(u);
                    ++deg[u]; // 增加这个点的属于点双连通分量数量
                    // 因为记的是点,所以要加入深度最浅的点
                    do {
                        vv = st[top--];
                        vdcc[idx].push_back(vv);
                        // 将连通块中边代表的点加入,并在栈中删去这条达
                        ++deg[vv];
                    } while (vv != v);
                }
            } else
                low[u] = min(low[u], dfn[v]);
        }
}
int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    for (int i = 1, b, e; i <= m; ++i)
        scanf("%d%d ", &b, &e), addedge(b, e);
    for (int i = 1; i <= n; ++i)
        if (!dfn[i]) {
            dfs(i, -1);
            top = 0;
```

```
                // 深度最浅的点不会代表任何边,不会被删去,因此要清空栈
            }
        int ans = 0;
        // 找出每个度大于1的结点
        for (int i = 1; i <= n; ++i)
            ans += deg[i] > 1;
        printf("%d\n ", ans);
        for (int i = 1, lst = 0; i <= n; ++i)
            if (deg[i] > 1) {
                if (lst)
                    putchar(' ');
                lst = true;
                printf("%d", i);
            }
        putchar('\n');
        /*这段语句是输出每个点连通分量所对应的点
        for (int i = 1; i <= idx; ++i) {
        printf ("%d : ", i);
        for (int j = 0; j < vdcc[i].size () ; j++)
        printf ("%d", vdcc[i][j]);
        printf ("\n ") ;
        }
        */
        return 0;
    }
```

每个点连接着若干不同的点双连通分量，同时每个点双连通分量包含着若于点，可以依此性质将图变为树结构，其中将原图顶点称为圆点，代表着点双连通分量的点被称作方点，则这样的树结构就被称作圆方树。由于其描述了点双连通的结构，因此圆方树是解决点相关路径问题的利器。一个圆方树的例子如图 12-5 所示。

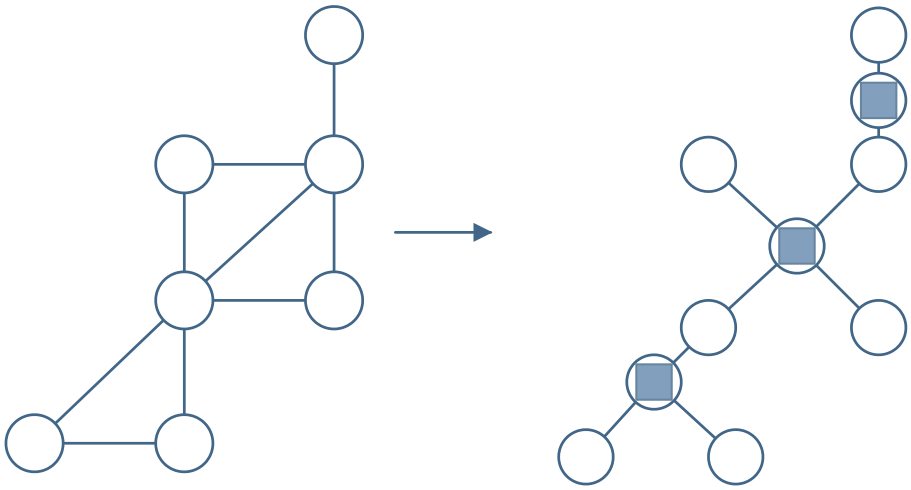


图12-5 圆方树

要建立圆方树，由于已经有了深度关系，只需要更改找到点双连通分量时的代码：

```
if (low[to[i]] >= dfn[u]) {
    ++idx;
    ans[idx].push_back(u);
    addedge(u, idx + n); // u 深度最浅
    int v;
    do {
        v = st[top--];
        ans[idx].push_back(v);
        addedge(idx + n, v); // v深度比u大
    } while (v != to[i]);
}
```

利用圆方树，可以很方便地得到割点的充要条件：该结点在圆方树上的度大于 1。

### 例 12-4 铁人两项(P4630,APIO2018)

某城市路网由 $m$ 条双向道路连接的 $n$ 个交叉路口组成，保证 $n\leqslant 100000, m\leqslant 2000000$，

最近，该市获得了一场铁人两项锦标赛的主办权。这场比赛共有两段赛程：选手先完成一段长跑赛程，然后骑自行车完成第二段赛程。比赛的路线要按照如下方法规划：

1. 先选择 $3$ 个两两互不相同的路口 $s$，$c$，$f$，分别作为比赛的起点、切换点（运动员在长跑到达这个点后，骑自行车前往终点）和终点。
2. 选择一条从 $s$ 出发，经过 $c$ 最终到达 $f$ 的路径。考虑到安全因素，选择的路径经过同一个点至多一次。

在规划路径之前，市长想请你帮忙计算，总共有多少种不同的选取 $s$，$c$，$f$ 的方案，使得在第 $2$ 步中至少能设计出一条满足要求的路径。

**分析：**问题可以转化为枚举 $s$ 和 $f$，计算 $s$ 到 $f$ 的不经过重复点的路径上有多少可能经过的点（除去 $s$ 和 $f$）。

关于这个问题，可以转化到圆方树上。路径 $s$ 到 $f$ 对应到链上的权值和，其中方点的贡献为**点双连通分量**的点数，圆点的贡献为-1.这样计算权值和刚好是可能经过的点数：因为题目要求去掉两个端点，那么用路径上的方点算贡献时，路径上所有的圆点都被多算了一次，圆点-1 的权值刚好去除掉多余的贡献。

于是求出每个点的权值后，直接计算以这个点为 $c$，$s$，$f$ 的方案数，在计算贡献的意义下，$s$ 或 $f$ 可以与 $c$ 相同，但是 $s\neq f$。所以 $s$ 和 $f$ 只能在圆点上，且必须与 $c$ 相同或者在以 $c$ 为根的不同子树中。关于这样的限制，直接在圆方树上求出各子树大小并计算两两之积的和即可。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 500010;
const int MAXM = 500010;
typedef long long LL;
inline LL pows(int x) { return 1ll * x * x; }
struct G {
    int head[MAXN], nxt[MAXM << 1], to[MAXM << 1], tot;
```

```
    void addedge(int b, int e) {
        nxt[++tot] = head[b];
        head[b] = tot;
        to[tot] = e;
        nxt[++tot] = head[e];
        head[e] = tot;
        to[tot] = b;
    }
} G, T;
int dfn[MAXN], low[MAXN], st[MAXN], top, idx, t0t, n, m, t1, t2, sz[MAXN], sznow;
void tarjan(int u) { // 建立圆方树
    sz[u] = -1;
    ++sznow;
    dfn[u] = low[u] = ++t0t;
    st[++top] = u;
    for (int i = G.head[u]; i; i = G.nxt[i])
        if (!dfn[G.to[i]]) {
            tarjan(G.to[i]);
            low[u] = min(low[u], low[G.to[i]]);
            if (dfn[u] <= low[G.to[i]]) {
                ++idx;
                int v;
                do {
                    v = st[top--], T.addedge(v, idx), ++sz[idx];
                } while (v != G.to[i]);
                ++sz[idx];
                T.addedge(u, idx);
            }
        } else
            low[u] = min(low[u], dfn[G.to[i]]);
}
LL ans;
int sz2[MAXN];
void dfs(int u, int fa) {
    sz2[u] = u <= n;
    // 只有圆点才能作为 s或f
    // 而c在圆点方点都行
    LL cnt = 0;
    for (int i = T.head[u]; i; i = T.nxt[i])
        if (T.to[i] != fa) {
            dfs(T.to[i], u);
            // 计算两两子树的积
            cnt += 1LL * sz2[T.to[i]] * sz2[u];
            sz2[u] += sz2[T.to[i]];
        }
    // 在DFS的时候,还有父结点的子树需要计算
    // 在这个计算同时还考虑了c和s相同的情况
    cnt += 1LL * sz2[u] * (sznow - sz2[u]);
    // 上面的计算值算了一半情况
    cnt <<= 1;
    // 计算对答案的贡献
    ans += 1LL * sz[u] * cnt;
}
int main() {
    scanf("%d%d", &n, &m);
    idx = n;
```

```
    for (int i = 1; i <= m; ++i) {
        scanf("%d%d", &t1, &t2);
        G.addedge(t1, t2);
    }
    for (int i = 1; i <= n; ++i)
        if (!dfn[i]) {
            // 对于图的每个连通分量计算,所以不要忘了清空
            sznow = 0;
            top = 0;
            tarjan(i);
            dfs(i, 0);
        }
    printf("%lld\n", ans);
    return 0;
}
```

## 12.3 有向图的强连通性

### 例 12-5 奶牛舞会(洛谷P2863,USACO2006年1月赛)

有一张 $n$ 个点、$m$ 条边的有向图，求出这个图点数大于 1 的**强连通分量**个数。

数据范围 $:2 \leq n \leq 10^4, 2 \leq m \leq 5 \times 10^4$。

**分析：** 先介绍有向图的强连通分量。在有向图中，定义强连通为两点之间可以互相到达，这与无向图连通的性质十分类似。当一张图任意两个点都强连通，那么称这张图为强连通图。因为如果 $u$ 能到 $v$，$v$ 能到 $w$，就有 $u$ 能到 $v$，所以强连通具有传递性。如果把强连通的点归为一类的话，原图的点集就会被分成若干不相交的子集，其中每个子集都是原图中的一个极大强连通子图，这样的子图被称作强连通分量。

很多有向图问题中需要求出强连通分量。把强连通分量替换为点的操作称为缩点，即建立一个点数为强连通分量个数的新图，当一条边的两端属于不同的强连通分量，那么在新图上在强连通分量对应的点之间连边。缩点之后的图是一张有向无环图，因为如果不是则与原图取到的是“极大强连通子图”矛盾。

图 12-6 是缩点的一个例子，其中{1，3，4}是强连通分量，当然{1}、{5}、{6}也是。

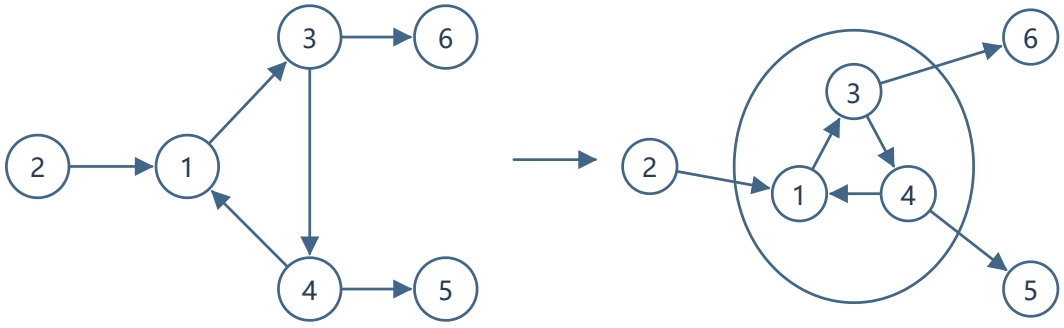



图12-6 缩点

在得到每个点属于的强连通分量编号后，缩点十分容易解决，所以问题就是怎么求出图的所有强连通分量。

解法 1（Tarjan算法）：有了先前的经验，尝试在DFS生成树中计算。不一样的是，得到有向

图的DFS生成树后，非树边是存在横叉边的，但是这些横叉边起点的DFS序必定比终点大。

类似在解决**边双连通分量**时的情况，除了横叉边，对于不在DFS生成树中的边，终点是起点祖先的边，被称作返祖边（也可称后向边）；起点是终点祖先的边被称作前向边。

容易发现，前向边 $(u,v)$ 对于可到达性的作用相当于树中 $u$ 到 $v$ 的链，所以在求强连通分量的时候，前向边是没有用的。

对于返祖边，其意义恰好和在求边双连通分量的时候一样：将树中一条链的点全部合并到一个集合。

剩下的就是要考虑横叉边的意义。因为所有横叉边都是DFS序大的点 $u$ 连向DFS序小的点 $v$，对于其 $\mathrm{LCA}(u,v)=l$，如果 $v$ 和 $l$ 在同一强连通分量里，那么横叉边会将 $u$ 到 $l$ 的链合并。容易发现，这些DFS序大的点的强连通性依赖于DFS序小的点，而在DFS过程中，横叉边的起点的DFS序也是递增的，按这样的顺序合并，正确性得以保证。

如图 12-7 所示，展现的就是树边、返祖边、前向边与横叉边的区别。

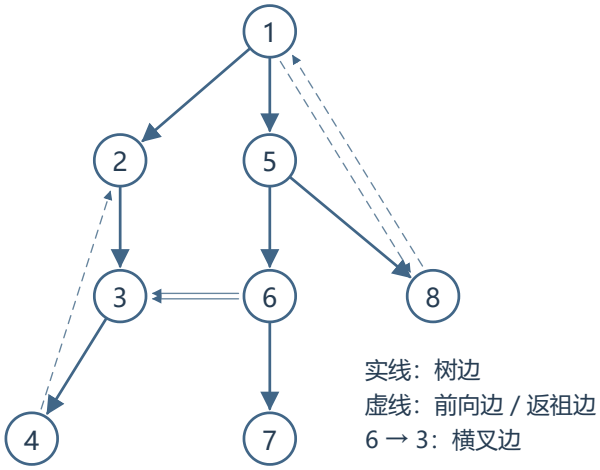



图12-7树边、返祖边、前向边与横叉边

修改求边双连通分量的算法，为了横叉边方便合并，考虑当横叉边的终点 $v$ 在栈中的时候，说明 $u$ 和 $v$ 的**最近公共祖先**$l$ 并不是 $v$ 所在的强连通分量深度最小的点。也就是说，淡化“仅通过至多一条返祖边能到达的深度最小结点的DFS序”的意义，将数组 `low` 当作辅助合并的标记。因为 $u$ 到 $l$ 的路径上除了 $l$，所有点的DFS序都比 $v$ 大，所以可以直接用 $low_u$ 与 $df_{n_v}$ 取最小指表示合并到 $l$。

对于如图 12-7 所示的有向图运行Tarjan算法，步骤如下（这张图刚好结点编号等于这个结点的遍历顺序，也就是 `dfn` 值）：

1. 从结点 1 开始运行。将结点 1 放入栈内， $d/n$ 值和 `low` 值都设置为 1，如图 12-8(a) 所示。
2. 枚举结点 1 出来的边，进入到结点 2，放入栈内；进入结点 3，放入栈内；进入结点 4，放入栈内。发现有一个连到 2 的返祖边，将其 `low` 值设为 2，如图 12-8(b) 所示。
3. 返回结点 3， `low` 值设为其子结点的 2；返回结点 2， `low` 值也设置为 **2**。此时发现结点 2 的 `low` 值等于 $d\ f_n$ 值，说明发现了一个强连通分量。一直退出栈元素直到遇到自己为止。找到的第 1 个强连通分量就是{4，3，2}，如图 12-8(c) 所示；

4. 返回结点 1；进入结点 5；进入结点 6，发现连向结点 3 的边是横向边（此时栈中无 3），什么也不做：进入结点 7，无路可走，发现其 `low` 值等于 $dfn$ 值，找到第 2 个强连通分量{7}，如图 12-8(d) 所示。
5. 返回结点 6.发现其 `low` 值等于 $dfn$ 值，找到第 3 个强连通分量 **{6}**；回到结点 5；进入到结点 8，发现有一个连到结点 1 的返祖边，将其 `low` 值设为 1，如图 12-8(e) 所示。
6. 回到结点 5，`low` 值设置为 1；返回结点 1，`low` 值也设置为 1，然后发现连向结点 8 的前向边（在栈中），于是什么都不做；此时发现结点 1 的 `low` 值等于 $dfn$ 值，找到的第 4 个强连通分量就是{8，5，1}，如图 12-8(f) 所示。此时已经找到所有的强连通分量。

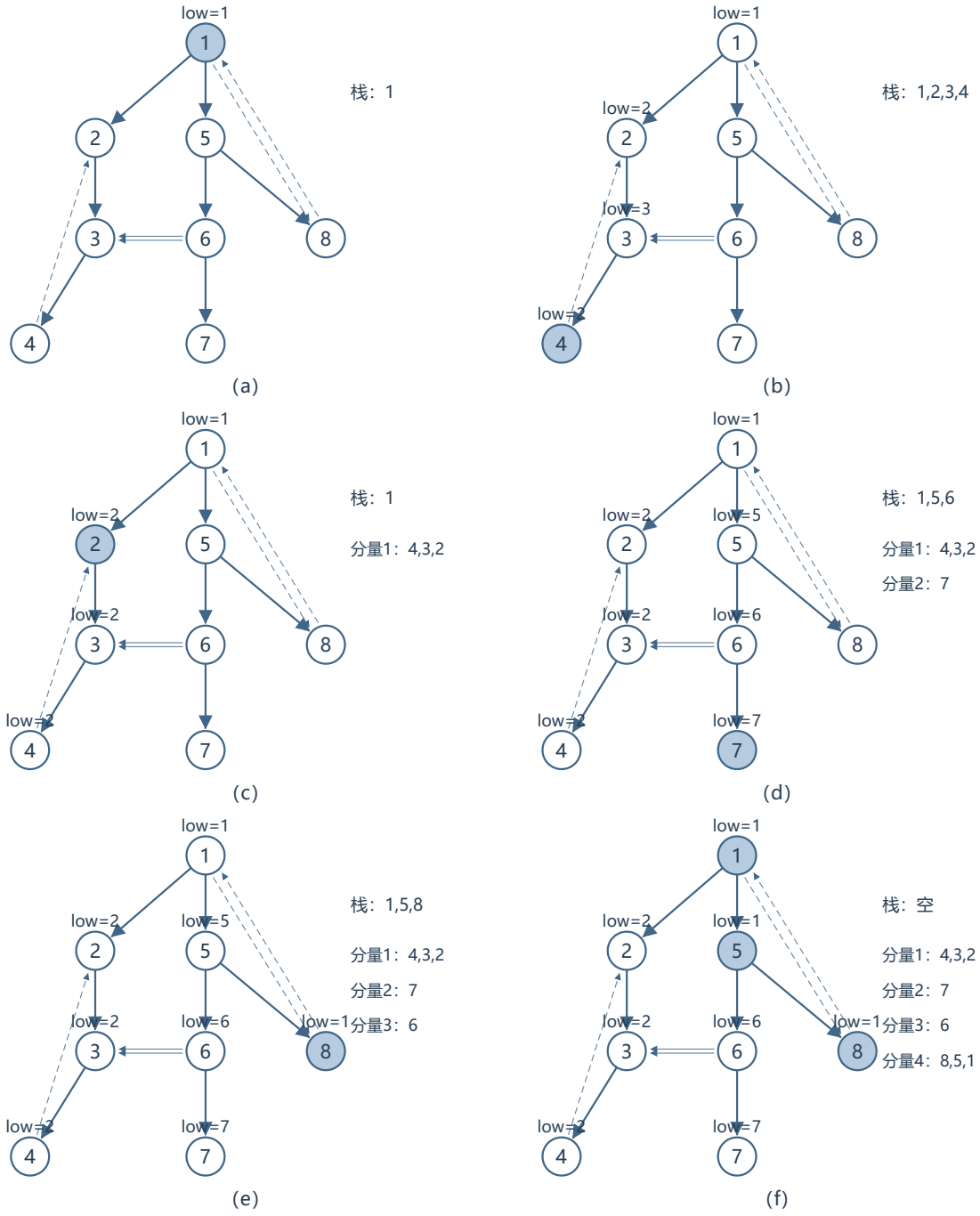



图12-8 Tarjan算法过程

注意到在一有向图中，`dfs` 函数可能会被执行多次。这是因为选择作为DFS生成树根的结点不一定能够到达图上所有的点。这样做强连通分量不会被漏找或多找，因为对于任意一个结点，其总能到达其所在的强连通分量中所有的点。

按上述代码的方法对强连通分量进行标号，则在缩点以后的图中，如果 $u$ 能到达 $v$，那么 $u$ 所代表的强连通分量编号一定大于 $v$。因为只有在 $u$ 代表的点访问完所有能到达的点后，$u$ 代表的强连通分量才会被标号。

例题是Tarjan算法的模板，只需要统计每个强连通分量的大小，然后输出符合要求的数量即可。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 1e5 + 10;
int head[MAXN], nxt[MAXN], to[MAXN], tot;
void adde(int b, int e) {
    nxt[++tot] = head[b];
    to[head[b] = tot] = e;
}
int dfn[MAXN], low[MAXN], st[MAXN], top;
bool ins[MAXN]; // 表示一个结点是否在栈中
int cnt, idx, siz[MAXN], bel[MAXN];
void dfs(int u) {
    low[u] = dfn[u] = ++cnt;
    ins[st[++top] = u] = true; // 标记在栈中
    for (int i = head[u]; i; i = nxt[i]) {
        if (!dfn[to[i]]) { // 进入没有遍历过的结点
            dfs(to[i]);
            low[u] = min(low[u], low[to[i]]);
        } else if (ins[to[i]]) {
            // 这样的写法包含了前向边和返祖边两种情况
            // 干是low的DP意义被淡化,成了辅助合并的标记
            // 前向边终点在栈中,则dfn[to[i]]一定比 dfn[u]大,不会影响
            low[u] = min(low[u], dfn[to[i]]);
        }
    }
    if (low[u] == dfn[u]) {
        int v;
        ++idx;
        do {
            v = st[top--];
            bel[v] = idx;
            ins[v] = false; // 弹出栈的时候维护在栈中的信息
            ++siz[idx];     // 统计大小
        } while (v != u);
    }
}
int main() {
    int n, m;
    scanf("%d%d ", &n, &m);
    for (int i = 1, b, e; i <= m; ++i)
```

```
        scanf("%d%d ", &b, &e), adde(b, e);
    // 因为一次DFS不一定遍历到所有结点,所以跑多次
    for (int i = 1; i <= n; ++i)
        if (!dfn[i])
            dfs(i);
    int ans = 0;
    for (int i = 1; i <= idx; ++i)
        ans += siz[i] > 1;
    printf("%d\n", ans);
    // 输出各点强连通分量
    // for (int i = 1; i <= n; ++i)
    //  printf ("%d%c", bel[i], "\n "[i == n]);
    return 0;
}
```

解法 2（Kosaraju算法）：考虑缩点后得到的有向无环图（Directed Acyclic Graph，DAG），将点进行**拓扑排序**。如果不像Tarjan算法一样，先找拓扑序较大的点代表的强连通分量，而是先找拓扑序较小点的会怎么样呢？也就是在找连通块的时候，不能DFS到拓扑序较大的点代表的强连通分量。这个性质与原图得到的性质相反，所以可以在找强连通分量的时候，在原图的反图（即将所有的有向边（$u$，$v$）反向，变成（$v$，$u$）上遍历，并且在找到一个强连通分量后，将其在反图上删除。

这样按拓扑序从小到大做，每次DFS能得到恰好一个强连通分量。所以问题就转化为：找到一个恰当的顺序来遍历求强连通分量。

将目光放在DFS生成树中，一个强连通分量标志性的结点上，这个标志性的结点就是在生成树中深度最小的结点，将其称为一个强连通分量的特殊点。考虑Tarjan算法中，一个强连通分量越早被找到，其特殊点在遍历的过程中越早遍历完子树。而一个特殊点一定比其强连通分量内其他点更晚被DFS完子树。这说明按“DFS完子树的时间”从晚到早，就刚好能得到刚才所说的“恰当的顺序”。

算法过程如下：在求DFS生成树的时候，维护一个栈，在每个结点遍历完出边后，将这个点加入栈。然后，在遍历找强连通分量的时候，每次取出栈顶对应的结点，并从这个结点开始遍历找强连通分量。这么做成功规避了Tarjan算法中更为烦琐的DP过程，所以Kosaraju算法的代码非常简洁：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 1e5 + 10;
std::vector<int> G[MAXN], Gi[MAXN];
// 原图与反图
bool vis[MAXN];
int st[MAXN], top, ans;
void dfs1(int u) { // 获取 DFS 生成树
    vis[u] = true;
    for (int i : G[u])
        if (!vis[i])
            dfs1(i);
    st[++top] = u;
}
int siz[MAXN], bel[MAXN];
void dfs2(int u, int id) { // 在反图上 DFS 获取强连通分量
```

```
    vis[u] = true;
    bel[u] = id;
    siz[id]++;
    for (int i : Gi[u])
        if (!vis[i])
            dfs2(i, id);
}
int main() {
    int n, m;
    scanf("%d%d ", &n, &m);
    for (int i = 1, b, e; i <= m; ++i) {
        scanf("%d%d", &b, &e);
        G[b].push_back(e);
        Gi[e].push_back(b);
    }
    for (int i = 1; i <= n; ++i)
        if (!vis[i])
            dfs1(i);
    memset(vis, 0, sizeof vis);
    int idx = 0;
    while (top) {
        int u = st[top--];
        if (!vis[u])
            dfs2(u, ++idx);
    }
    for (int i = 1; i <= idx; ++i)
        ans += siz[i] > 1;
    printf("%d\n", ans);
    // 输出个点强连通分量
    // for (int i = 1; i <= n; ++i)
    // printf ("%d%c", bel[i], "\n "[i == n]) ;
    return 0;
}
```

上述代码是朴素的 $O(n+m)$ 复杂度。下面介绍该算法在稠密图上的优化。

深度优先搜索中，对顶点 $u$ 只需找到满足 $(u,v)\in E$ 且尚未访问的顶点 $v$。令 $g_u=\{v\mid(u,v)\in E\}$，并用集合 $t$ 保存未访问的顶点，则可从 $g_u\cap t$ 中取出一个顶点。使用 `bitset`，这些操作可按机器字批量进行，每次处理的复杂度为 $O(n/w)$，其中 $w$ 为机器字长。对稠密图可据此加速，复杂度为 $O(n^2/w)$。

> **注：** 1. `bitset` 是 C++ 中的一种STL容器，用于高效批量处理二进制相关的数据，读者可以自行查阅相关资料。

### 例 12-6 缩点（洛谷 P3387）

给定一张有 $n$ 个点、$m$ 条边有向图，每个点有一个权值。求一条路径，使路径经过的点权值之和最大，需要求出这个权值和。允许多次经过一条边或者一个点，但是，重复经过的点，权值只计算一次。

**数据范围：** $1\leqslant n\leqslant 10^4, 1\leqslant m\leqslant 10^5, 0\leqslant a_i\leqslant 10^3$.。

**分析：**因为**强连通分量**中的点可以互相到达，所以到达强连通分量里任意一个点，其所有点的权值都能累加。

将强连通分量看作一个权值为其点集权值和的点，这样原图就变成了一个DAG（有向无环图），可以在DAG中**动态规划**以获得答案。

注意到如果按强连通分量得到的顺序早到晚进行从小到大编号，就不用**拓扑排序**了：如果拓扑序小的强连通分量 $A$ 的DP结果依赖于拓扑序大的强连通分量 $B$，那么 $A$ 一定比 $B$ 晚得到。所以在Tarjan算法中，找到强连通分量时就可以进行动态规划了。

代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 1e5 + 10;
int head[MAXN], nxt[MAXN], to[MAXN], tot;
void adde(int b, int e) {
    nxt[++tot] = head[b];
    to[head[b] = tot] = e;
}
int dfn[MAXN], low[MAXN], st[MAXN], top;
bool ins[MAXN];
int cnt, idx, bel[MAXN];
int dp[MAXN], val[MAXN], ans;
void dfs(int u) {
    low[u] = dfn[u] = ++cnt;
    ins[st[++top] = u] = true;
    for (int i = head[u]; i; i = nxt[i])
        if (!dfn[to[i]]) {
            dfs(to[i]);
            low[u] = min(low[u], low[to[i]]);
        } else if (ins[to[i]])
            low[u] = min(low[u], dfn[to[i]]);
    if (low[u] == dfn[u]) {
        int v;
        ++idx;
        int sum = 0;
        do {
            ins[v = st[top--]] = false;
            bel[v] = idx;
            sum += val[v];
            // 找到强连通分量的所有出边进行转移
            for (int i = head[v]; i; i = nxt[i])
                dp[idx] = max(dp[idx], dp[bel[to[i]]]);
        } while (v != u);
        // 路径的值要加上本身的值
        dp[idx] += sum;
        ans = max(ans, dp[idx]);
    }
}
int main() {
    int n, m;
```

```
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; ++i)
        scanf("%d", &val[i]);
    for (int i = 1, b, e; i <= m; ++i)
        scanf("%d%d ", &b, &e), adde(b, e);
    for (int i = 1; i <= n; ++i)
        if (!dfn[i])
            dfs(i);
    printf("%d\n ", ans);
    return 0;
}
```

### 例 12-7 2-SAT问题(洛谷P4782)

有 $n$ 个布尔变量 $x_1 \sim x_n$，另有 $m$ 个需要满足的条件，每个条件的形式都是“$x$; 为true/false或 $^{66}x_i x_j$ 为 true/false”，如 $^{66}x_1$ 为真或 $x_3$ 为假 $^{1346}x_7$ 为假或 $x_2$ 为假”。

2-SAT问题的目标是给每个变量赋值使得所有条件得到满足。输出是否有解，如果有解则输出任意一组满足条件的方案。

**数据范围：** $1 \leqslant n, m \leqslant 10^6$

**分析：**2-SAT问题的标志就是每个变量只有取值 0 或 1，并且条件是 $A \Rightarrow B$($A$ 推出 $B$）这种形式。例如题目中的“或”形式，如果一个为假，那么能推出另一个为真。

于是可以将每个变量拆成两个点Pi。和 $p$」表示两种取值，用有向边 $A \rightarrow B$ 表示 $A \Rightarrow B$ 这种 $p_{i,0} P_{i,1}$ 形式，同时加入每个命题的逆否命题，就能通过计算一个点能够到达的所有点来表示一个取值能推导出的其他所有变量。

如果 $A$ 和$\neg A$ 属于同一个**强连通分量**，则必定是无解的。如果不存在这样的情况，则给出用Tarjan算法得到的一种构造方法：对于每个变量取对应点所在强连通分量编号较小的值。例如，记 $b_x$ 为一个点所在的强连通分量，如果 $b_{p_{x,0}} < b_{p_{x,1}}$，就选择 0 为最后的值。

现在证明当 $b_A < b_{\neg A}$ 时，若 $A \Rightarrow B$，则有 $b_B < b_{\neg B}$：由 $A \Rightarrow B$ 推出 $b_B \leqslant b_A$，由逆否命题的连边，图具有对称性，则有 $b_{\nu_{\nu_B}} \geqslant b_{\nu_{\nu_{\nu_A}}} > b_A \geqslant b_B$　也就是这样取，不会发生 $A \Rightarrow \neg A$，也不会发生 $A \Longrightarrow B$，却同时选择了 $A$ 和$\neg B$，这说明了是一个合法的解。

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 2e6 + 10;
int head[MAXN], nxt[MAXN], to[MAXN], tot;
void adde(int b, int e) {
    nxt[++tot] = head[b];
    to[head[b] = tot] = e;
}
int dfn[MAXN], low[MAXN], st[MAXN], top;
bool ins[MAXN];
int cnt, idx, bel[MAXN];
void dfs(int u) {
    low[u] = dfn[u] = ++cnt;
    ins[st[++top] = u] = true;
    for (int i = head[u]; i; i = nxt[i])
```

```
        if (!dfn[to[i]]) {
            dfs(to[i]);
            low[u] = min(low[u], low[to[i]]);
        } else if (ins[to[i]])
            low[u] = min(low[u], dfn[to[i]]);
    if (low[u] == dfn[u]) {
        int v;
        ++idx;
        do {
            ins[v = st[top--]] = false;
            bel[v] = idx;
        } while (v != u);
    }
}
int pt[MAXN][2];
int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    for (int i = 1, t = 0; i <= n; ++i)
        pt[i][0] = ++t, pt[i][1] = ++t;
    for (int i = 1; i <= m; ++i) {
        int x, a, y, b;
        scanf("%d%d%d%d", &x, &a, &y, &b);
        // 加入!a推出b,!b推出a两个命题
        // 且它们互为逆否命题
        adde(pt[x][!a], pt[y][b]);
        adde(pt[y][!b], pt[x][a]);
    }
    for (int i = 1; i <= n * 2; ++i)
        if (!dfn[i])
            dfs(i);
    for (int i = 1; i <= n; ++i)
        if (bel[pt[i][0]] == bel[pt[i][1]]) {
            // 如果a当且仅当!a,则无解
            puts("IMPOSSIBLE");
            return 0;
        }
    puts("POSSIBLE");
    for (int i = 1; i <= n; ++i) {
        // 取强连通分量编号小的是一组合法的解
        int d = bel[pt[i][0]] < bel[pt[i][1]] ? 0 : 1;
        printf("%d%c", d, " \n"[i == n]);
    }
    return 0;
}
```

## 12.4 课后习题与实验

### 习题 1 最远的路(洛谷CF1000E)

给定一张有 $n$ 个点、$m$ 条边的无向图，找到两个点 $s$ 和 $t$，使得 $s$ 到 $t$ 必须经过的边最多（一条边无论走哪条路线都经过它，这条边就是必须经过的边）。

**数据范围：** $\cdot 2 \leqslant n \leqslant 3 \times 10^5, 1 \leqslant m \leqslant 3 \times 10^5$。

### 习题 2 封锁(洛谷P3469,POI2008)

给定一无向图，求每个点被封锁之后有多少个有序点对 $(x,y)\,(x\neq y,1\leqslant x,y\leqslant n)$ 满足 $x$ 无 法到达 $y$

**数据范围：** $1\leqslant n\leqslant 10^5,1\leqslant m\leqslant 5\times 10^5$

### 习题 3 战略游戏(洛谷P4606,山东省队选拔2018)

小 $Q$ 和小 $C$ 玩一款战略游戏，游戏地图由 $n$ 座城市以及 $m$ 条连接这些城市的双向道路构成，并目从任意一座城市出发总能沿着道路走到任意其他城市。

现在小 $C$ 已经占领了其中至少两座城市，小 $Q$ 可以摧毁一座小 $C$ 没占领的城市，同时摧毁所有连接这座城市的道路。只要在摧毁这座城市之后能够找到某两座小 $C$ 占领的城市 $u$ 和 $v$，使得从 $u$ 出发沿着道路无论如何都不能走到 $v$，那么小 $Q$ 就能赢下这一局游戏。

小 $0$ 和小 $C$ 一共进行了 $q$ 局游戏，每一局游戏会给出小 $C$ 占领的城市集合 $S$。请帮助小 $Q$ 数出有多少座城市在他摧毁之后能够让他赢下这一局游戏。

保证单组测试点中数据组数 $T\leqslant 10,2\leqslant n\leqslant 10^5,n-1\leqslant m\leqslant 2\times 10^5,1\leqslant q\leqslant 10^5$，对于每组数据 $\Sigma\,|S|\leqslant 2\times 10^5$。

### 习题 4 旅行者(洛谷CF487E)

给定一张有 $n$ 个结点、$m$ 条边的无向图，第 $i$ 个顶点有一个权值 $w_i$，进行 $q$ 次询问，需要支持动态修改一个点的权值，以及询问两个点之间简单路径上权值最小值最小可能是多少。

数据范围 $:n,m,q\leqslant 10^5$。

### 习题 5 间谍网络(洛谷P1262)

有 $n$ 个间谍，如果 $A$ 间谍手中掌握着关于 $B$ 间谍的犯罪证据，则称 $A$ 可以揭发 $\mathrm{B}_{\circ}$ 有些间谍收受贿赂，只要给他们一定数量的金钱，他们就愿意交出手中掌握的全部情报。所以，如果能够收买一些间谍，就可能控制间谍网中的每个人，因为一旦逮捕了一个间谍，就可以掌握他手中的所有情报，这样就有可能逮捕新的间谍，再掌握新的情报。

判断是否有可能控制全部的间谍，如果可以，求出所需要支付的最少资金。否则，输出不能被控制的一个间谍。

**数据范围：** $n\leqslant 3000$，收买每个间谍的价格 $\leqslant 20000$，证据个数 $\leqslant 8000$。

### 习题 6 受欢迎的牛(洛谷P2341,USACO2003年月赛)

每头奶牛都梦想成为牛棚里的明星。奶牛之间的“喜欢”是可以传递的——如果 $A$ 喜欢 $B$，$B$ 喜欢 $C$，那么 $A$ 也喜欢 $C$。被所有奶牛喜欢的奶牛就是一头明星奶牛。牛栏里共有 $N$ 头奶牛，给定 $M$ 个奶牛之间的“喜欢”关系，请计算出有多少头奶牛可以当明星。

**数据范围：** $1\leqslant N\leqslant 10^4,1\leqslant M\leqslant 5\times 10^4$。

### 习题 7 炸弹(洛谷P5025,陕西省队选拔2017)

在一条直线上有 $n$ 个炸弹，每个炸弹的坐标是 $x_i$，爆炸半径是 $r_i$。 当一个炸弹爆炸时，如果另一个炸弹所在位置 $x_j$ 满足 $|x_j - x_i| \leq r_i$，那么，该炸弹也会被引爆。现在，请计算先把第 $i$ 个炸弹引爆，将引爆多少个炸弹。

**数据范围：** $1 \leqslant n \leqslant 500000, -10^{18} \leqslant x_i \leqslant 10^{18}, 0 \leqslant r_i \leqslant 2 \times 10^{18}$。

### 习题 8 游戏(洛谷P3825,NOI2017)

给定字符串 $S$，仅由 `x`、`a`、`b`、`c` 四个字母组成。构造一个与 $S$ 等长且仅包含 `a`、`b`、`c` 的字符串 $T$，要求对所有 $1 \leq i \leq |S|$ 都有 $T_i \neq S_i$。同时还要满足若干条件：若 $T_i = \alpha$，则 $T_j = \beta$。

**数据范围：** $|S| \leqslant 5 \times 10^4$，条件数 $\leqslant 10^5$，$x$ 的个数 $\leqslant 8$

### 习题 9 追查逃犯(洛谷P4819)

一位逃犯假装成平 $\boxed{\frac{1}{2}}$ 警察希望能在 $N$ 个人里面查出谁是逃犯。警察能够对每个人进行查证，假如查证的对象是平民，他会告诉警察，他认识的人里谁是逃犯，谁是平民。假如查证的对象是逃犯，任务就会失败。现在警察掌握了每个人认识谁，共有 $M$ 条他们的人际关系的线索。每个人都有可能是逃犯，可看作他们是逃犯的概率是相同的。请问，根据最优的情况，知道谁是逃犯的概率最大是多少？

**数据范围：** $1 \leqslant N \leqslant 100000, 0 \leqslant M \leqslant 300000$

### 习题 10 庆典(洛谷P7737,NOI2021)

某国由 $n$ 座城市和 $m$ 条有向道路组成，城市从 $1$ 到 $n$ 编号。如果从 $x$ 号城市出发，经过若干条道路后能到达 $y$ 号城市，那么称 $x$ 号城市可到达 $y$ 号城市，记作 $x \Longrightarrow y($ 道路都有一个特点：对于 $3$ 座城市 $x, y, z$，若 $x \Longrightarrow z$ 且 $y \Longrightarrow z$，那么有 $x \Longrightarrow y$ 或 $y \Longrightarrow x_i$。

现在计划进行 $q$ 次游行庆典活动，第 $i$ 次游行庆典希望从城市 $s_i$ 出发，经过若干个城市后，在城市 $t_i$ 结束，且在游行庆典过程中，一个城市可以被经过多次。为了增加庆典活动的乐趣，每次游行还会临时修建出 $k(0 \leqslant k \leqslant 2)$ 条有向道路专门供本次游行庆典使用，即其他游行庆典计划不能通过本次游行修建的道路。

现在想知道，每次游行庆典计划可能会经过多少座城市。注意：临时修建出的道路可以不满足道路原有的特点。

**数据范围：** $1 \leqslant n, q \leqslant 3 \times 10^5, n - 1 \leqslant m \leqslant 6 \times 10^5, 0 \leqslant k \leqslant 2$。

# 第4部分：动态规划

## 第13章 动态规划的引入

相信读者们接触过很多“求最佳方案”的题目了，遇到这种问题，可以考虑使用贪心算法在每步决策时使用单步最佳方案，或者使用枚举（搜索）算法枚举所有可能的决策。使用贪心算法的问题是“目光短浅”，不一定可以站在全局的角度来统筹决策，可能会出现结果并不是全局最佳的情况；而使用枚举搜索算法，虽然可以遍历所有的可能，但如果数据量稍大，就可能超时。这个时候可以考虑使用**动态规划**解决这些问题。

动态规划的核心是“状态缓存”，将一个状态的统计结果传递到另外一个状态，这有一点类似递推算法。动态规划是算法竞赛的常客，非常考验选手问题分析和代码实现能力。本章思维导图如图 13-1 所示。




图13-1 本章思维导图

# 13.1 动态规划

### 例 13-1 纸币问题1(洛谷P2842)

某国有 `n(n≤1000)` 种纸币，每种纸币面额为正整数 $a_i$ 并有无限张，试问使用至少多少张纸币可以凑出 `w(w≤10000)` 的金额。

例如，提供 **1**、**5**、**10**、**20**、**50**、**100** 元面额的纸币，要凑足 **15** 元，至少需要 **2** 张；提供 **1**、**5**、**11** 元面额的纸币，要凑足 **15** 元，则至少需要 **3** 张。

**分析：**第 1 个样例代表了生活中纸币的面额。按照经验，付钱时尽量先用大面额，超过了再用较小面额的，以此类推。对于第 1 个样例应用上述贪心思路，得到 $15=10+5$，两张纸币，恰好是最优解。然而这个贪心策略对于第 2 个例子就行不通了。按照策略将会得到 $15=11+1+1+1+1$，用了 5 张纸币。但是最优策略是 $15=5+5+5$，贪心算法陷入僵局。

贪心策略的最大问题在于鼠目寸光：它过分关注了最大面值，全程没有给 5 任何机会。那么在正式思考这个问题之前，不妨先来看看如何使用搜索解决第 2 个样例，代码如下：

```
#include <bits/stdc++.h>
const int INF = 1e9;
using namespace std;
int dfs(int w) {
    if (w == 0)
        return 0;
    if (w < 0)
        return INF;
    return min(dfs(w - 1), min(dfs(w - 5), dfs(w - 11))) + 1;
}
int main() {
    cout << dfs(15);
    return 0;
}
```

令 `dfs(w)` 的返回值为凑够 $w$ 元最少需要的纸币数。在每个阶段，分别尝试使用 1 张 1 元纸币、1 张 5 元纸币、1 张 11 元纸币，分别对剩下的金额继续搜索，即 `dfs(w-1)`、`dfs(w-5)` 和 `dfs(w-11)`。在这 3 种情况下，找到需要纸币总张数最小的情况，加 1 返回即可。

这段程序又短又好写，但是它效率较低，因为产生了重复计算。考虑这样一个问题：`dfs(w)` 的值对于 $w$ 来说，是不是固定的呢？显然，凑够 10 元钱的最少纸币数只与纸币面值有关，不会因为 `dfs(w)` 的调用先后而发生改变。换句话说，`dfs(w)` 的返回值在程序运行过程中是恒定的，每次计算同一个 $w$，得到的答案必然相同。比起函数，`dfs(w)` 的作用更像是一个数组。

因为一个 `dfs(w)` 只需要被计算一次，那么在 `dfs(w)` 被第 1 次计算时把它的答案固定下来，存储到数组 $f$ 的元素 $f_w$ 中①。具体代码如下：

```
#include <bits/stdc++.h>
const int INF = 1e9;
using namespace std;
int f[10010];
int dfs(int w) {
    if (w < 0)
        return INF;
    if (f[w] != INF)
        return f[w]; // 如果已计算过了就直接返回
    if (w == 0)
        return 0;
    int x = min(dfs(w - 1), min(dfs(w - 5), dfs(w - 11))) + 1;
    f[w] = x;
    return f[w];
}
int main() {
    int k = 15;
    for (int i = 1; i <= k; i++)
        f[i] = INF;
    cout << dfs(k);
    return 0;
}
```

> **注：** 1. 仅在本小节中用下标记号 $f$. 表示数组元素，括号记号 `dfs(w)` 表示函数，接下来将不再用该记号区分。

因为每个 `dfs(w)` 对于不同的 $w$ 值只被计算了一遍，所以这段代码的复杂度是 $O(k)$ 的。如果计入纸币种类数 $n$，那么复杂度应当是 $O(kn)$。这类算法因为结合了搜索与固定答案的思想，称之为**记忆化搜索**。

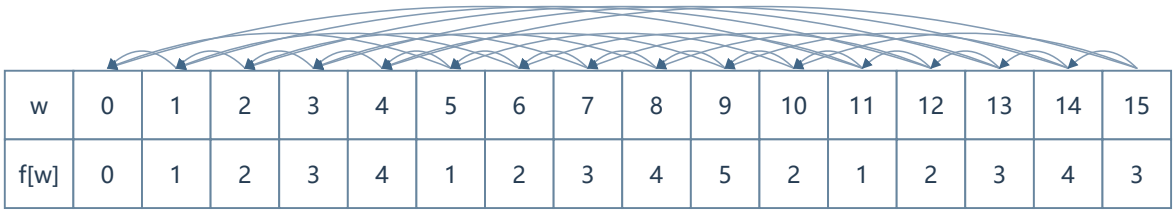
换个角度思考，每个 $f_w$ 和 `dfs(w)` 在运行的时候是绑定的，调用 `dfs(w)` 函数的本质其实就是计算 $f_w$。如果程序运行的栈空间有限，而 $w$ 又很大的时候，应该怎么处理？

首先想到的思路或许是控制栈深度。栈的深度与递归调用的次数有关，`dfs(w)` 函数反复调用自己，容易导致超过栈空间的限制。然而，在计算 `dfs(w)` 值的时候，`dfs(w-1)`、`dfs(w-5)` 和 `dfs(w-11)` 的值全部已知，那么栈的深度就得到了控制，因为它在第 2 层就停下来了。可以依次调用 `dfs(0)` 到 `dfs(w)`，这样就可以把栈的深度控制到最小。

上述做法，可以不需要通过递归来计算 $f_w$，只需要使用递推按照从小到大的顺序对于每个 $w$ 求出 $f_w$ 即可。抛开 `dfs(w)`，直接让 $f_w$ 表示凑够 $w$ 的金额最少需要多少张纸币，也完全能计算。具体代码如下：

```
#include <bits/stdc++.h>
#define INF 2000000000
using namespace std;
int f[10010], a[1010]; // 用f[i]表示凑够i的金额最少需要多少张纸币
int main() {
    int n, w;
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
        scanf("%d", &a[i]); // 读入面值
    scanf("%d", &w);
    f[0] = 0;                       // 对应dfs()函数中的特殊边界
    for (int i = 1; i <= w; i++) { // 考虑计算f[i]
        f[i] = INF;
        for (int j = 1; j <= n; j++) // 找到最小的f[i - a[j]] + 1
            if (a[j] <= i)
                f[i] = min(f[i], f[i - a[j]] + 1);
    }
    printf("%d", f[w]);
    return 0;
}
```

本题的数组 $f$ 的值和转移路径如图 13-2 所示（深色线条代表生效的转移路径）



| w | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| f[w] | 0 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2 | 1 | 2 | 3 | 4 | 3 |

转移依赖：f[w−1]、f[w−5]、f[w−11]（下标非负时）

图13-2 纸币问题的状态转移

这里用 $f_w$ （数组元素）表示凑够 $w$ 的金额最少需要多少张纸币，取代了意义几乎相同的搜索函数 `dfs(w)`。计算 `dfs(w)` 的时候，枚举这个阶段用了哪张纸币，继续搜索，然后选择答案最优的那个。同理，在计算 $f_w$ 的时候，也是枚举了这个阶段用了哪张纸币，只不过使用之前的答案来计算 $f$，最终得到目标答案。

这样通过把原问题分解成若干相关子问题（所有这些问题在一定意义上答案固定），再利用已知子问题的答案依次计算未知问题答案，最终得到原问题答案的方式，称之为**动态规划**（Dynamic Programming， DP）。

### 例 13-2 数字三角形(洛谷P1216)

给定一个数字金字塔。求从最高点到底部任意处结束的一条路径，使路径经过数字的和最大。层数 $n \leqslant 1000$。

下面给出一个数字金字塔。

```
7
3 9
8 1 0
2 7 4 4
4 5 2 6 5
```

在这个样例中，$7→3→8→7→5$ 的路径产生了最大数字和 $30$。

**分析**：刚拿到这道题，或许也会陷入贪心的陷阱，最后的答案会不会刚好经过最大值呢？样例就刚好给出了一个反例。如果认为最大值行不通转而去分析次大值、次次大值的性质，就可能越陷越深。

暴力写法并不难，代码如下：

```
int dfs(int x, int y) { // 首先运行 dfs (1, 1)
    if (x == n)
        return a[x][y];
    return max(dfs(x + 1, y), dfs(x + 1, y + 1)) + a[x][y];
}
```

`dfs(x,y)` 返回了从第 $x$ 行第 $y$ 个数字走到最下层的最大数字和。依样画葫芦，思考 `dfs(x,y)` 的性质：`dfs(x,y)` 的值相对于 $x$，$y$ 来说是也是固定的，从第 $x$ 行第 $y$ 个数字走到最下层的最大数字和只与下面的数字相关，和前面的走法无关。固定 $\mathbf{dfs(x,y)}$ 到 $g_{x,y}$，得到代码如下：

```
int g[1010][1010];      // 初始化为无穷
int dfs(int x, int y) { // 首先运行 dfs (1, 1)
    if (g[x][y] != INF)
        return g[x][y];
    if (x == n)
        return g[x][y] = a[x][y];
    return g[x][y] = max(dfs(x + 1, y), dfs(x + 1, y + 1)) + a[x][y];
}
```

本题的数组 $g$ 的值和转移路径如图 13-3 所示。

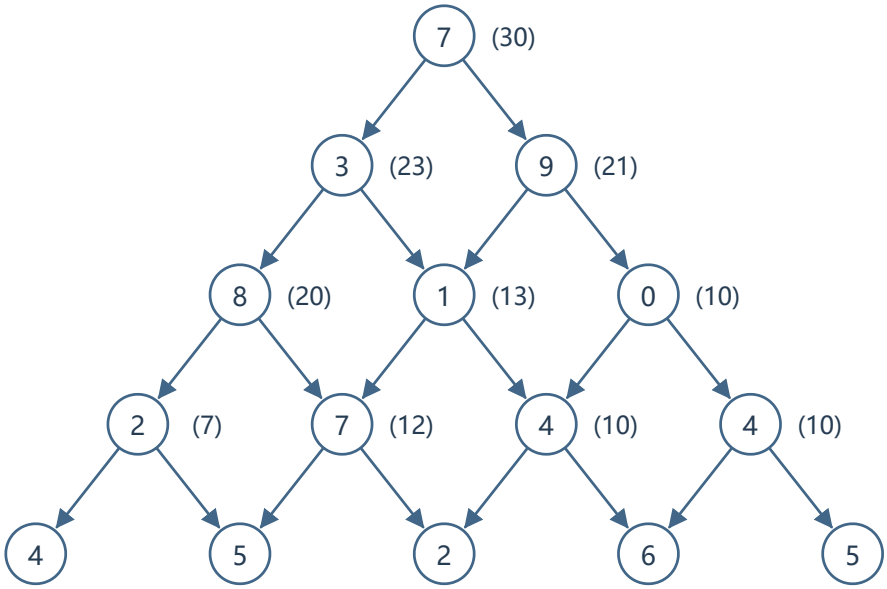



图13-3 数字三角形的状态转移

感觉一切都是惊人地相似。为了控制栈深度，`dfs(x,y)` 最好沿着 $x$ 从下往上一层一层计算，那么再把 `dfs(x,y)` 这个壳脱掉：

```
// 输入n和a[][]
for (int j = 1; j <= n; j++)
    g[n][j] = a[n][j]; // 对应 dfs() 函数中的边界
for (int i = n - 1; i >= 1; i--)
    for (int j = 1; j <= i; j++)
        g[i][j] = min(g[i + 1][j], g[i + 1][j + 1]) + a[i][j];
// 输出 g[1][1]
```

如果注意到数组 $a$ 和 $g$ 完全可以合并（原来数组 $a$ 的值不会再被用到），那么代码就变成：

```
#include <bits/stdc++.h>
using namespace std;
int a[1010][1010], n;
// 用a[x][y]表示从第x行第y列出发走到底层的最大数字和
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= i; j++)
            scanf("%d", &a[i][j]);
    for (int i = n; i > 1; i--)
        for (int j = 1; j <= i - 1; j++)
            a[i - 1][j] += max(a[i][j], a[i][j + 1]);
    printf("%d", a[1][1]);
    return 0;
}
```

这与前面如出一辙，本题中将原问题 $g_{x,y}$ 分解成了 $\theta_{x+1,y}$ 和 $g_{x+1,y+1}$，并按照金字塔从下到上的方式更新答案。处理好边界问题，就可以得到目标答案 $g_{1,1}$。

## 13.2 状态、转移与有向无环图

在对**动态规划**有了一定的认识后，为了建立良好的理论基础，需要掌握以下术语。

### 1. 状态

动态规划的状态可以笼统地解释成“问题所在的局面”。例如，“凑够 $w$ 的金额最少需要多少张纸币”“从第 $x$ 行第 $y$ 个数字走到最下层的最大数字和”都是状态。某状态、某局面、某状态表示的答案（的值）、某状态代表的方案（集合）、某状态代表的有向无环图上的点，在本质上相通，有时会用同一个字母（如 $f$）表示。本书中不做区分，请读者仔细感受甄别。

可以看到.状态具有明确良好的定义。一般可以写成：（用变量表示的）所在局面的（最优）答案，或者满足某种性质的方案数。这里“凑够 $w$ 的金额”是一种关于 $w$ 的局面，“从第 $x$ 行第 $y$ 个数字走到最下层”是一种关于 $x$， $y$ 的局面。然后“最少需要多少张纸币”和“最大数字和”分别 $x,y$ 是问题所对应的最优答案。根据 13.1 节的经验，如果初学者实在不知道怎么确定状态，可以考虑套用最朴素的搜索算法中的状态。

状态的答案应该只依赖状态定义的局面和一些状态以外的常量（如纸币的面额相对于状态来说是常量）。也就是说，在一般情况下，状态变化并不会影响状态以外的量，该状态之后的演变不受这个状态之前决策的影响。因此，如果要用动态规划来解决一道题，应把所有可能的变动加入状态的定义中去。

### 2. 转移

因为要求解最终状态的答案，所以状态之间需要存在某些关系。一般把这样的计算关系称作转移，对应于 图 13-2 和 图 13-3 中的连边，计算式称为（状态）转移方程。例如，例 13-1 中的转移方程为：

$$f(w)=\min_{1\le i\le n,\,a_i\le w}\{f(w-a_i)+1\}$$

例 13-2 中的状态转移方程为：

$$g(x,y)=\max\left(g(x+1,y),g(x+1,y+1)\right)+a_{x,y}$$

可以看到，只有依据正确的状态定义和状态转移方程，才可以有理有据地做对一道题。

状态转移方程一般具有它自己的意义，体现的是从一个状态到另一个状态的变化。例如，例 13-1 中，从 $f(w-a_i)$ 转移到 `f(w)` 的变化是“多取用了一张面值为 $a_i$ 的纸币”，对应于答案的变化是“多用了一张纸币”，故需要将答案加 1；例 13-2 中，对于 $g(x,y)$ 这个局面来说，考虑的是它要往左下走还是右下走，但无论向哪儿走，它都多获得了 $a_{x,y}$ 上的值，所以要找到左下和右下中较优的那个局面再加上转移时获得的 $a_{x,y}$

对于最优化问题来说，转移方程的目的就是找到最优的前继情况的答案。在无特殊性质的情况下，转移方程需要找出所有前继情况并取出最优答案。

注意，若前继状态的最优答案不能转移成为当前状态的最优答案，则该问题（至少目前这个定义的状态）不满足用动态规划解决的要求。前继状态最优解与当前状态之间最优解的关联性被称为动态规划的最优子结构性质。

对于计数问题来说，转移方程的目的就是不遗漏不重复地统计局面的所有方案数。一般来说，转移方程需要找出所有前继情况并进行恰当地组合计数。

值得一提的是，转移存在两种方式，一种是发送型转移，另一种是接收型转移。前者枚举状态的后继，并计算对后继的贡献；后者枚举状态的前继，当即计算出状态的答案。两种转移方式的选取主要取决于枚举前后继状态的难度和转移的复杂程度。不同的题选用恰当的转移方式，代码编写难度会显著下降。

### 3. 有向无环图

接下来，借用有向无环图（DAG）的概念来理解动态规划为何如此高效。DAG是一种可以表示“依赖关系”的图，在动态规划中，依赖关系体现为“一个局面可以从哪些局面转移过来”。这里不容许“互相依赖”的情形发生：一旦两个局面可以互相到达，那么这个问题（至少目前这个定义的状态）不具备用动态规划解决的性质。

这个比较好理解。对于 例 13-1，所有纸币的面值都是正整数，不可能多拿了一张纸币，面值还变小了（转移到了已经计算好的状态），即状态间不可能相互到达。对于 例 13-2，只能向左下或者右下走，也不可能多走一步后反而走到了更高的位置（转移到了已经计算好的状态），状态也不可能互相到达。这个状态间不能互相到达的性质以及未来与过去无关的性质统称为动态规划的无后效性。

把动态规划的状态和转移与DAG的点和边建立对应关系。因为转移顺序的需要，有时题意中的DAG与DP建模时DAG的边恰好是相反的，即所谓“自底向上”。在此不做区分，希望读者根据上下文自行理解。由于有向无环图具有严格的依赖性质，这就要求处理每个状态时，必须先处理完它的前继状态。那些没有前继（或者前继相对不完全）的状态称为边界状态，如例 13-1 的 `f(0)` 和 例 13-2 的 $g(n,y)$。

在《基础篇》已经学过了**拓扑排序**——按照拓扑排序遍历图就能保证当前的状态的前继状态全部被遍历过，恰好解决了这个问题。当然对于绝大多数动态规划问题来说，拓扑排序是明显的，有时还是分层的。例如，例 13-1 中的 `f(ww)` 就可以从小到大处理（明显），例 13-2 中的 $g(x,y)$ 可以从下到上处理（分层）。

有时，把这种明显的层次称为阶段。

例 13-1 中的 `f(w)` 从小到大被称为阶段，即纸币金额；例 13-2 中的 $g(x,y)$ 从下到上也同样被称为阶段，即数字金字塔的层。阶段的含义呼之欲出，它体现了一种步骤，只要按照步骤执行，整个问题可以有条不紊地（按照拓扑排序）被解决。同时阶段也和状态的定义紧密相关。只要找到了正确的阶段，就不需要刻意建图寻找拓扑排序，只要按照某种相对简单的顺序遍历整个状态数组即可

熟悉了几个术语以后，以下用例题来巩固动态规划的相关概念以及解决思路。

### 例 13-3 纸币问题2(洛谷P2840)

有 $n(n\leqslant 1000)$ 种面额不相同的纸币，每种纸币面额为 $a_i$ 并有无限张，现在需要支付 $w(w\leqslant 10000)$ 的金额，请问有多少种方式。这里同样的纸币组合，支付顺序不一样视作不同方案。输出答案是对于 $1000000007$ 取模的结果。

分析。这是一道计数题，先要定义状态。这道题比较明显的可能作为阶段的量有两个：纸币种类数 $n$ 和总金额 $w$。这道题与 例 13-1 很像，可以猜测这道题的阶段可以是纸币的总金额。 $w$ 仿照状态定义的一般格式，给出该题状态的定义 $f$，表示支付 $w$ 的金额的合法方案数。这里的 $f_w$ 合法其实很模糊，因为每道题都有不同的合法方案，这道题里的合法方案顺序可以不同，但后面的例题就无视了顺序这个概念。

现在考虑怎么转移。需要计算 $f$，它是凑够 $w$ 的金额的总合法方案数。考虑每种不同的支 $f_w$ 付方案，付钱的最后一张纸币是唯一确定的。如果确定了最后一张纸币面额为 $a$，那么之前 $a_i w \mathcal{U} - a_i$ 的金额的支付方案数完全划归到了子局面 $f$–。有些地方把**动态规划**的这种性质成为重 $f_{w-a_i}$。叠子结构，即状态的子局面一定是动态规划的子问题，局面之间具有重叠的相关性。

根据加法原理，把所有这种情况加起来就是 $f$ 的答案，得出状态转移方程： $f_w$

$$f_v = \sum_{i=1}^{n} f_{v-a_i}$$

边界条件很明显是 $f_0 = 1$。

推荐练习动态规划时把状态写进注释以确保思路清晰，本题代码如下。

```
#include <cstdio>
using namespace std;
const int mod = 1000000007;
// f[i]表示凑够i的金额的总方案数
int f[10010], a[1010];
int main() {
    int n, w;
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
        scanf("%d", &a[i]); // 读入面值
    scanf("%d", &w);
    f[0] = 1;
    for (int i = 1; i <= w; i++) // 考虑计算f[i]
        for (int j = 1; j <= n; j++)
            if (i >= a[j])
                f[i] = (f[i] + f[i - a[j]]) % mod;
    printf("%d", f[w]);
    return 0;
}
```

### 例 13-4 纸币问题3(洛谷P2834)

有 $n(n \leqslant 1000)$ 种纸币，每种纸币面额为 $a_i$ 并有无限张，试问有多少种纸币组合可以凑到 $w$ $(w \leqslant 10000)$ 的金额。输出答案是对于 $1000000007$ 取模的结果。

**分析：**如果仍然采用 例 13-3 中的状态，用 $f$ 表示凑出 $w$ 的金额的组合方案数，会发现无法 $f_w$ 转移 $f_w$ 与 $f_{w-a_i}$ 完全没有关系。无法知道 $f-a$ 中再加一张ai的纸币会产生多少重复的，即无法 $f_{w-a_i}\, a_i$ 进行组合计数，自然无法转移。

再考虑以纸币种类数为阶段。仅用 $f_i$ 表示“只考虑前 $i$ 类纸币，凑齐金额 $w$ 的方案数”，还不足以描述转移。可以增加金额这一维，令 $f_{i,j}$ 表示只使用前 $i$ 类纸币凑齐金额 $j$ 的方案数。这样只需考虑加入第 $i$ 类纸币 $a_i$ 后，各个金额对应的方案数如何变化。

考虑 $f_{i,j}$ 的成分，只考虑前 $i$ 类纸币凑齐金额 $j$ 的方案数，可以分为“用了至少一张 $a_i^{\flat}$ 和“没有用 $a_i''$ 两种情况之和。后者很明显恰好是 $f_{i-1,j}$。

前者看起来好像情况复杂，看起来像是对若干 $f_{i-1,j-k} \cdot a_i$ 求和，但实际上恰好等于 $f_{i,j-a_i}0$ 根据定义 $f_{i,j-a_i}$ 的局面再加一张 $a_i$ 就成为“用了至少一张 $a_i{}^{p}$ 的局面。

把当前局面分成“至少使用一张 $a_i$”和“没有使用 $a_i$”两种情况，就能完成统计；再增加一维来表示已考虑的纸币种类，代码如下：

```
for (int i = 1; i <= n; i++) {
    for (int j = 0; j < a[i]; j++)
        f[i][j] = f[i - 1][j];
    for (int j = a[i]; j <= w; j++) // 保证j≥ a[i],j小的先算
        f[i][j] = (f[i][j - a[i]] + f[i - 1][j]) % mod;
}
```

但是这样空间复杂度会变成 $O(wn)$。观察发现完全可以覆盖上一个答案，一步到位（想一想，为什么）。

```
#include <cstdio>
#define maxn 1010
#define maxw 10010
#define INF 2000000000
using namespace std;
const int mod = 1000000007;
// f[i][j]表示使用前i种纸币,凑够j的金额的纸币组合数
int f[maxw], a[maxn];
int main() {
    int n, w;
    scanf("%d ", &n);
    for (int i = 1; i <= n; i++)
        scanf("%d", &a[i]);
    scanf("%d", &w);
    f[0] = 1;
    for (int i = 1; i <= n; i++)
        for (int j = a[i]; j <= w; j++)
            f[j] = (f[j] + f[j - a[i]]) % mod;
    printf("%d", f[w]);
    return 0;
}
```

状态对于解题十分重要，即使类似的模型也可能采用不同的状态定义，甚至有可能状态定义相同但含义完全不同，请读者多多留心。

### 例 13-5 采药(洛谷P1048,NOIP 2005普及组)

某个山洞里有 $M(\Delta M\leqslant 100)$ 株草药。采第 $i$ 株需要 $a_i$ 的时间，能获得 $a_iv_i$ 的价值。可以在一段时间 $T(T\leqslant 1000)$ 限制内采集这些草药，最大化采到药的总价值。

假设采集 $3$ 种草药，第 $1$ 株耗时 $2$，价值 $3$；第 $2$ 株耗时 $3$，价值 $5$；第 $3$ 株耗时 $5$，价值 $7$。 共给予 $7$ 个单位的时间，那么采第 $1$ 株、第 $3$ 株草药耗时 $7$，得到价值 $10$。

**分析：**这题和前几题很像，只不过计数问题变成了最优化问题。用 $f$ 表示只用前 $i$ 种物品 $f_{i,j}$ 在 $j$ 时间内能获得的最大价值。因这里只能使用一个物品，可以把fi.分成两类：“用上了物品 $f_{i,j}a_i^\flat$ 和“没用 $a_i^\flat$ 两种情况。

前者的前继很明显是 $f_{i-1,j-a_i}$ 的局面，此时可以获得 $f_{i-1,j-a_i}+v_i$ 的价值；后者的前继明显是 $f_{i-1,j}$ 的局面，获得 $f_{i-1,j}$ 的价值。根据最优子结构的性质，对两个局面的价值取最大值就行了。

得到代码如下：

```
for (int i = 1; i <= M; i++) {
    for (int j = 0; j < a[i]; j++)
        f[i][j] = f[i - 1][j];
    for (int j = a[i]; j <= T; j++)
        f[i][j] = max(f[i - 1][j], f[i][j - a[i]] + v[i]);
}
```

本题的数组 $f$ 的值和转移路径如图 13-4 所示。

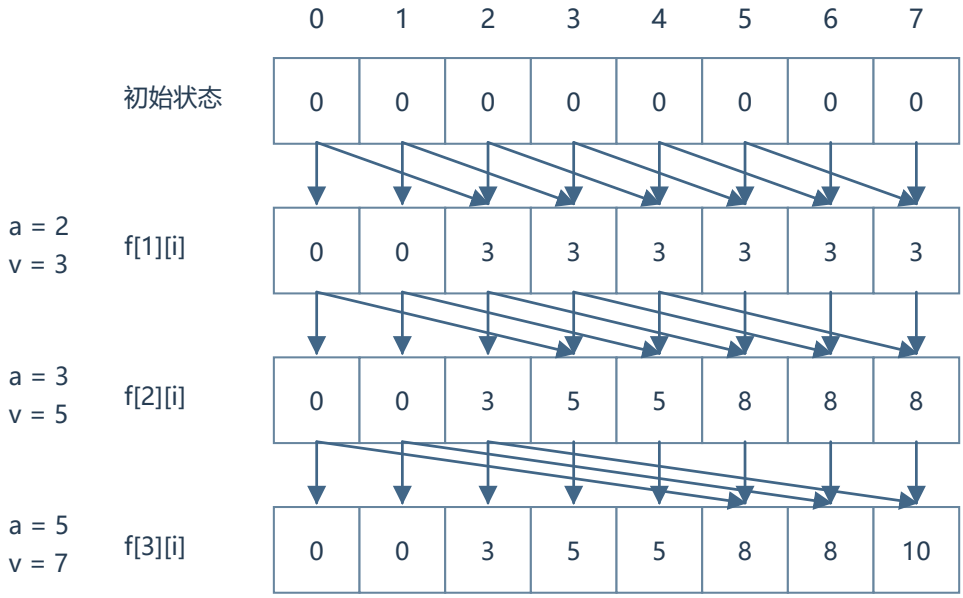



图13-4 背包问题的状态转移

由于有了前车之鉴，可能读者想问，这里能不能也把前一维优化掉呢。先来看直接优化掉会怎样。

```
for (int i = 1; i <= M; i++)
    for (int j = a[i]; j <= T; j++)
        f[j] = max(f[j], f[j - a[i]] + v[i]);
```

看上去没有任何毛病，但是这样吗？如果只有一株草药，采集时间为 1，价值为 1，运行这个程序会发生什么？这个结果得到了每一个体积都能被凑到的结论，但这并不符合题意这段代码把一株草药采了多次，就因为它的 $j$ 是从前向后枚举的，这株草药被“堆起来”连续使用了！

为了修正这个问题，只需改变 $j$ 的枚举方向，问题迎刃而解。

```
#include <bits/stdc++.h>
using namespace std;
int T, m, t, v, f[1010];
// f[j]表示j的时间内能采到的最大价值
int main() {
    for (cin >> T >> m; m--;) {
        cin >> t >> v;
        for (int j = T; j >= t; j--)
            f[j] = max(f[j], f[j - t] + v);
    }
    cout << f[T];
}
```

这类具有多个物品可选，又有一定（体积）限制的问题模型称作背包问题。背包问题是**动态规划**最基础的模型之一。本题中所有物品只有一件的模型被称为 $\mathbf{0-1}$ 背包，所有物品可以选择任意多次的模型被称为完全背包。上面对于 $\mathbf{0-1}$ 背包错误的代码，恰恰就是完全背包问题的正确解法。此外还有很多其他背包问题的变式。

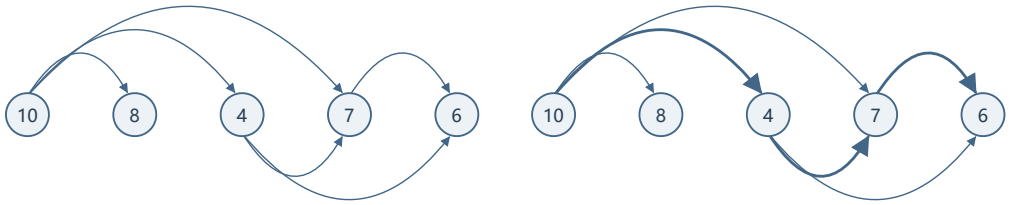
### 例 13-6 挖雷(洛谷P2196,NOIP1996提高组)

在一幅地图上有 $N$ 个地窖（$N\leqslant 20$），给定每个地窖的地雷数量。同时，给出地窖之间的连接路径（上三角邻接矩阵，只能从前往后挖）。当地窖及其连接的数据给出之后，某人可以从任意处开始挖雷，然后可以沿着指出的连接往下挖（仅能选择一条路径），当无连接时挖地雷工作结束。设计一个挖雷的方案，使某人能挖到最多的地雷。

输入样例（加上了一些空格便于理解）

```
5
10 84 7 6
1110
0 0 0
11
1
```

描述的地图如图 13-5(a) 所示。




(a) (b)

图13-5 挖雷的状态转移

**分析：**本例中的图是以上三角邻接矩阵的形式给出的，这意味着第 $i$ 个地窖只能走到 $i$ 以后的地窖——地窖编号从小到大构成**拓扑排序**。先画出DAG，显然合理的拓扑排序应该是从靠后的地窖开始向前计算。用 $f$ 表示从地窖 $i$ 走到底的最大答案，计算每个地窖时只需要取后继 $f_i$ 可达点的最大答案就行了。记得输出答案的时候要选取全局最大答案。

本例需要输出最优的方案，但**动态规划**中没有记录方案路径，观察 图 13-5(b) 中黑色的边，这些是在做转移时找到最大答案时对应的转移边。对于每个子问题，凡是成功转移过来的边必然是一个合法的（最大）路径。这意味着只要从（全局最大）答案开始出发按照转移过来的边往回找到源头，就一定是答案了，转移条件是 $a_{i,j}$ 且 $f_i = f_j + w_{ic}$ 也可以直接在计算 $f_i$ 的时候记录转移过来的状态，用空间换时间。

代码如下。

```
#include <bits/stdc++.h>
using namespace std;
int n, a[30][30], f[30], w[30], ans;
// f[i]表示从i点出发往后走最多能挖多少地雷
int main() {
    scanf("%d ", &n);
    ans = n;
    for (int i = 1; i <= n; i++)
        scanf("%d", &w[i]);
    for (int i = 1; i <= n; i++)
        for (int j = i + 1; j <= n; j++)
            scanf("%d", &a[i][j]);
    for (int i = n; i >= 1; i--) {
        f[i] = w[i];
        for (int j = i + 1; j <= n; j++)
            if (a[i][j])
                f[i] = max(f[i], f[j] + w[i]);
        if (f[ans] < f[i])
            ans = i; // 选取全局最大答案
    }
    printf("%d ", ans);
    for (int i = ans, j = ans + 1; j <= n; j++)
        if (a[i][j] && f[i] == f[j] + w[i])
            printf("%d ", i = j); // 按照 dp 路径往回找
    printf("\n%d", f[ans]);
}
```

## 13.3 拓扑排序与记忆化搜索

### 例 13-7 滑雪(洛谷P1434,上海市省队选拔2002)

小 $M$ 想知道在一个区域中最长的滑坡。区域由一个二维数组给出。数组的每个数字代表点的高度。例如：

```
1 2 3 4 5
16 17 18 19 6
```

| | | | | |
|---|---|---|---|---|
| 15 | 24 | 25 | 20 | 7 |
| 14 | 23 | 22 | 21 | 8 |
| 13 | 12 | 11 | 10 | 9 |

一个人可以从某个点滑向上下左右相邻 $4$ 个点之一，当且仅当高度减小。长和宽 $n,m\leqslant 100$，

在本例中，一条可行的滑坡为 $24-17-16-1$（从 $24$ 开始，在 $1$ 结束）。当然，$25-24-23-\cdots-3-2-1$ 更长，事实上，这是最长的一条。

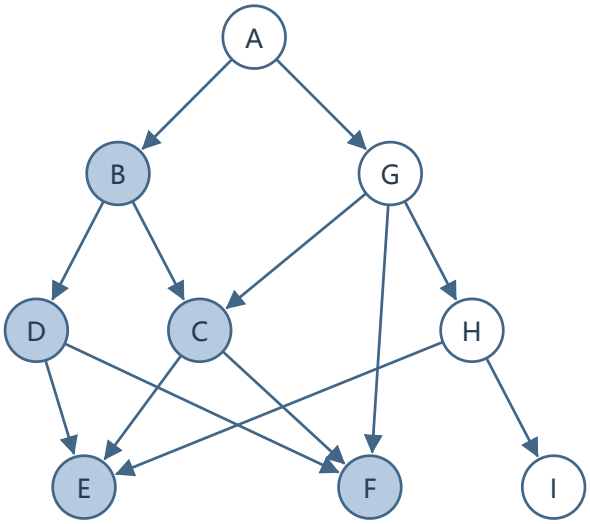
**分析：**网格中，把数值高的格点连向数值较低的相邻格点，形成DAG，并寻找最长路。

在熟悉DAG上寻找最长路的状态和转移方程的基础上，参考例 13-6 的挖雷案例，只要把每个地窖的地雷数固定为 1，就相当于是求解最长路。也就是说，用 $f_{x,y}$ 表示从点（$x$， $y$ ）出发的最长路，那么 $f(x,y)$ 的值就等于四周（能滑过去的点的）最大的 $f$ 值加 1。定义完了状态和转移方程，剩下的只需要按照正确的**拓扑排序**更新答案便可以解决问题。

然而本例并没有明显的阶段，不能很方便地归纳性质并找出拓扑排序。但没关系，再复杂的DAG，也总能通过真正意义上的建图、寻找拓扑排序、按照拓扑排序DP来解决问题。

追求简洁，当然不可能满足于需要多写一个拓扑排序的解法。本例的拓扑排序藏得不深：既然滑雪只能向低处滑，那所有点按照高度升序（因为要先计算最低点）就必然是一个正确的拓扑排序。那么有了第 1 个相对容易实现的解法：将所有点按照高度升序的顺序应用状态转移方程依次计算DP值，然后寻找全局最大答案就可以了。代码留给读者实现。

本章 例 13-1 和 例 13-2 提示了一个可以无视拓扑排序的方法：**记忆化搜索**。首先，它的搜索形式决定了它“先计算后到的结点，后计算先到的结点”的性质。这恰好就是DAG中的依赖关系：只要后继结点都已经算过了，就可以处理当前结点完全符合**动态规划**的转移要求。




记忆化搜索是可以“断点续传”的。例如，先计算完成了 $B$ 的所有后继结点（灰色结点），虽然尚没有处理完整张图，但是在计算 $A$ 的所有后继结点就只需要更新 $A$、$G$、$H$ 和 $I$，并且能够巧妙地用上

灰色结点的答案——虽然没有一口气计算完整张图，但是每个点一 图 13-6 DAG与记忆化搜索旦算过就确定下来了，再计算其他点时不需要重新处理已知的点。

因此，只要对DAG上的每个点按照任意顺序调用一遍记忆化搜索，就能得到整张图的答案，如图 13-6 所示。

那么这题的代码就很简洁了：在搜索中固定下每个状态的答案，然后对每个点都执行一遍记忆化搜索。因为每个点只被计算了一次，所以复杂度是 $O(nm)$。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
int n, m, a[110][110], f[110][110], ans;
// f[i][j]表示从第i行第j列出发的最长路长度
int dfs(int x, int y) {
    if (x > n || x < 1 || y > m || y < 1)
        return 0;
    if (f[x][y] != -1)
        return f[x][y];
    int ans = 0;
    if (a[x][y] > a[x][y + 1])
        ans = max(ans, dfs(x, y + 1));
    if (a[x][y] > a[x][y - 1])
        ans = max(ans, dfs(x, y - 1));
    if (a[x][y] > a[x + 1][y])
        ans = max(ans, dfs(x + 1, y));
    if (a[x][y] > a[x - 1][y])
        ans = max(ans, dfs(x - 1, y));
    return f[x][y] = ans + 1;
}
int main() {
    scanf("%d%d ", &n, &m);
    memset(f, -1, sizeof(f));
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            scanf("%d", &a[i][j]);
    int ans = 0;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            ans = max(ans, dfs(i, j));
    printf("%d", ans);
}
```

### 例 13-8 最大食物链总数(洛谷P4017)

有一个食物网，要求出这个食物网中最大食物链的数量，对 $80112002$ 取模（这里的“最大食物链”，指的是生物学意义上的食物链，即最左端是不会捕食其他生物的生产者，最右端是不会被其他生物捕食的消费者）。保证不会出现环状捕食的情况。点数 $n \leqslant 5000$。

**分析：**本题在《基础篇》中作为**拓扑排序**的例题介绍过，但可以更换一种思路，就是求从起点（入度为 0）出发到终点（出度为 0）的所有不同路径数。根据加法原理，只要计算从所有起点到每一个终点的不同路径数，然后对其求和就是所求的答案。

根据这个思路定义状态，$f$ 表示从所有起点到点 $u$ 的不同路径数。至于转移，再次根据加法原理，到达点 $u$ 的路径数就是到达点 $u$ 的所有前继的路径数之和。那么用**记忆化搜索**，可以无视拓扑排序便捷地求解此题。代码如下：

```
#include <bits/stdc++.h>
#define maxn 5010
using namespace std;
const int mod = 80112002;
int n, m, ind[maxn], oud[maxn], f[maxn], ans;
// f[i]表示从每个起点到点i的不同路径数
vector<int> a[maxn]; // 存图
```

```
int dfs(int u) {
    if (f[u] != 0)
        return f[u];
    int tmp = 0;
    for (int i = 0; i < a[u].size(); i++) // 枚举原图点u的所有前继,统计路径和
        tmp = (tmp + dfs(a[u][i])) % mod;
    return f[u] = tmp;
}
int main() {
    scanf("%d%d ", &n, &m);
    for (int i = 1; i <= m; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        a[v].push_back(u); // 因为在转移时需要枚举前继,故加了反向边
        ind[v]++;          // 统计原图入度
        oud[u]++;          // 统计原图出度
    }
    for (int i = 1; i <= n; i++)
        if (ind[i] == 0)
            f[i] = 1; // 边界问题:所有起点的答案等于1
    for (int i = 1; i <= n; i++)
        dfs(i); // 计算每个点
    for (int i = 1; i <= n; i++)
        if (oud[i] == 0)
            ans = (ans + f[i]) % mod; // 计算所有终点的路径和
    printf("%d", ans);
}
```

相对《基础篇》的拓扑排序做法，记忆化搜索法思路更加清晰。但要注意此方法会占用较多的栈空间，在栈空间受限的情况下慎用该方法。

## 13.4 动态规划的解法

### 例 13-9 求最大子段和(洛谷P1115)

给出一段序列，选出其中连续且非空的一段使得这段和最大。序列长度 $n \leqslant 200000$，序列值 $|a_i| \leqslant 10^4$。

**输入样例：**

```
7
2 -4 3 -1 2 -4 3
```

**输出样例：**

**分析：**题面很短，明显的阶段只有序列。不妨把序列作为阶段，以 $f$ 表示前 $i$ 个数中的最大子段和，却发现不知道如何转移：加入下一个数时根本不知道和前面的答案有什么关系。希望能够明确知道，加入的某个数是跟在之前的子段里还是作为一个新的子段的起点。

至于为什么能想到刚好是分成这两种情况，这是由题目的性质决定的。如果要从一个序列中选出一个子段，若是以序列为阶段，那么加入下一个数时要么是跟随、要么是无视前面结果开始新的一段。类似的序列问题被称作线性状态**动态规划**，将会在下一章详细解释。

考虑转移需要，需要定义的状态里保证当前正在处理的数在选中的子段里，所以需要更改状态的定义：以 $f_i$ 表示以第 $i$ 个数结尾的最大子段和。

这样改有什么好处？可以明确地知道加入某一个数时情况的变化：如果让 $a_i$ 跟在之前的序列里，那么新子段的和是 $f_{i-1}+a_i$；如果是让 $a_i$ 新开始一段，那么子段和就是 $a_i$。 在动态规划的过程中。一直在维护着这样一个虚拟的子段，不能轻易得知它从哪儿开始，只知道它的子段和，以及它一定以第 $i$ 个数结尾。

读者应该试着自己证明 $f_i$ 代表的子段要么是 $f_{i-1}$ 后面接上 $a_i$，要么是 $a_i$ 单个数（即其他情况不可能比这两种情况更优）。这个性质是构造转移的先决条件，如果没有发现这个性质，那么就无法用这个状态进行转移。

到这里已经确定了状态转移方程（再稍稍改写一下）：

$$f_i=\max(f_{i-1}+a_i,a_i)=\max(f_{i-1},0)+a_i$$

右式表示：如果当前子段和已经小于 0，就不应继续接在它后面，而应重新开始一段。虽然这个性质看起来很像贪心，但此处方法的本质是**动态规划**。

动态规划算法的代码就不难写出了，这里要注意最终答案不一定存在最后的数里，需要对整个数组 $f$ 取最大值。可应用滚动思想和覆盖思想来优化到 $O(1)$ 空间，具体实现参考如下代码：

```
#include <bits/stdc++.h>
#define maxn 200010
using namespace std;
int n, f[maxn], a[maxn], ans = -2000000000;
// f[i]表示以第i个数结尾的最大子段和
int main() {
    cin >> n;
    for (int i = 1; i <= n; i++)
        cin >> a[i];
    for (int i = 1; i <= n; i++)
        f[i] = max(f[i - 1], 0) + a[i], ans = max(ans, f[i]);
    cout << ans;
}
```

可以看到，对于状态构造和转移设计的关系是辩证的：具体的状态决定转移的方式；理想的转移方式引导人们构造状态。状态设计必然是为顺利转移服务的，请牢记。

### 例 13-10 整数划分

将 $n(n\leqslant 50000)$ 分为若干不同整数的和，有多少种不同的划分方式?例如： $n=6$，可行的划分有{6}、{1，5}、{2，4}、{1，2，3}，共 4 种。由于数据较大，输出对 $10^9+7$ 取模的结果。

> **注：** 1. 贪心算法的严格形式是先排序，再按照某种要求进行直接决策。

**分析：**如果把该题当作纸币问题 3 做，将每个整数视作一种纸币，那么会得到一个时间复杂度为 $O(n^2)$ 的划分，显然会超时。

考虑一个十分隐蔽的阶段：使用的数字个数。由于 $1+2+\cdots+316=50086$，这必然是使用数字最多的情况。也就是说，使用数字的个数是 $O(\sqrt{n})$ 级别的（请读者自证）。那么，这个阶段有什么用，怎么利用它来统计所有情况呢？

先定义一个状态 $f_{i,j}$，表示用了 $i$ 个不同的正整数，其和为 $j$ 的方案数（若把 $f_{i,j}$ 视为集合，那么它是一个以集合为元素的集合 $f_{i,j}$ 中每一个集合的元素个数为 $i$，和为 $j$）。首先 $f_{i,j}$ 这个集合里面会有两类集合：不含有 1 的和含有 1 的。

对于其中不含有 1 的集合，把其中每个数减去 1，会和 $f_{i,j-i}$ 里的所有元素一一对应（请读者自证）。同理，对于其中含有 1 的集合，把 1 去掉，再对其中每个数减去 1，会和 $f_{i-1,j-i}$ 里的所有元素一一对应。即状态转移方程为：

$$f_{i,j}=f_{i-1,j-i}+f_{i,j-i}$$

例如，考虑 $f_{2,6}$ 里面的元素应该只有 1，5 和 2，4。其中 2，4 是第一类集合，它与 $f_{2,4}$ 里的 1，3 相对应；1，5 是第二类集合，它去掉 1 以后与 $f_{1,4}$ 里的 4 相对应。

这个转移其实暗示了一个全新的正整数集合的生成方式：对已有集合的所有元素加 1；对已有集合的所有元素加 1，再补一个 $1$ 这里要注意，计数问题的转移与方案的生成方式紧密相关，有利的生成方式会带来意想不到的优化。代码如下：

```
#include <cstdio>
int dp[350][50010], ans, n, mod = 1e9 + 7;
// dp[i][j]表示用了i个不同的正整数,其和为j的方案数
int main() {
    scanf("%d", &n);
    dp[0][0] = 1;
    for (int i = 1; i <= 317; (ans += dp[i][n]) %= mod, i++) // ans统计所有
        // 答案
        for (int j = i; j <= n; j++)
            dp[i][j] = (dp[i - 1][j - i] + dp[i][j - i]) % mod; // 对应于分析中的状
    // 态转移
    printf("%d", ans);
}
```

一旦定义出了这个状态，并且想到了这个巧夺天工的转移，所有问题就迎刃而解。读者要学会尝试拿各种各样的阶段来定义状态。

一道**动态规划**题目会不可避免地涉及阶段、状态、转移、性质、优化等多个概念。将它们之间的关系归纳，如图 13-7 所示，并总结成解决大多数动态规划题目的步骤。

1. 理解题意：阅读题目信息并理解题意。
2. 寻找阶段/性质：对简单题来说要在题目中寻找可能可以成为阶段的量。对较复杂的题则需要寻找一些性质来支撑转移或者状态设计。
3. 构造状态：用之前找到的阶段或者性质来构造状态。定义状态的通常形式：（用变量表示的）所在局面]的[（最优）答案/满足某种性质的方案数]。注意这里的状态需要满足重叠子结构性质。
4. 设计转移：依靠定义的状态以及题意来写出状态之间的转移关系。对最优化题来说是最优解的转移关系，对计数题来说是组合计数关系。

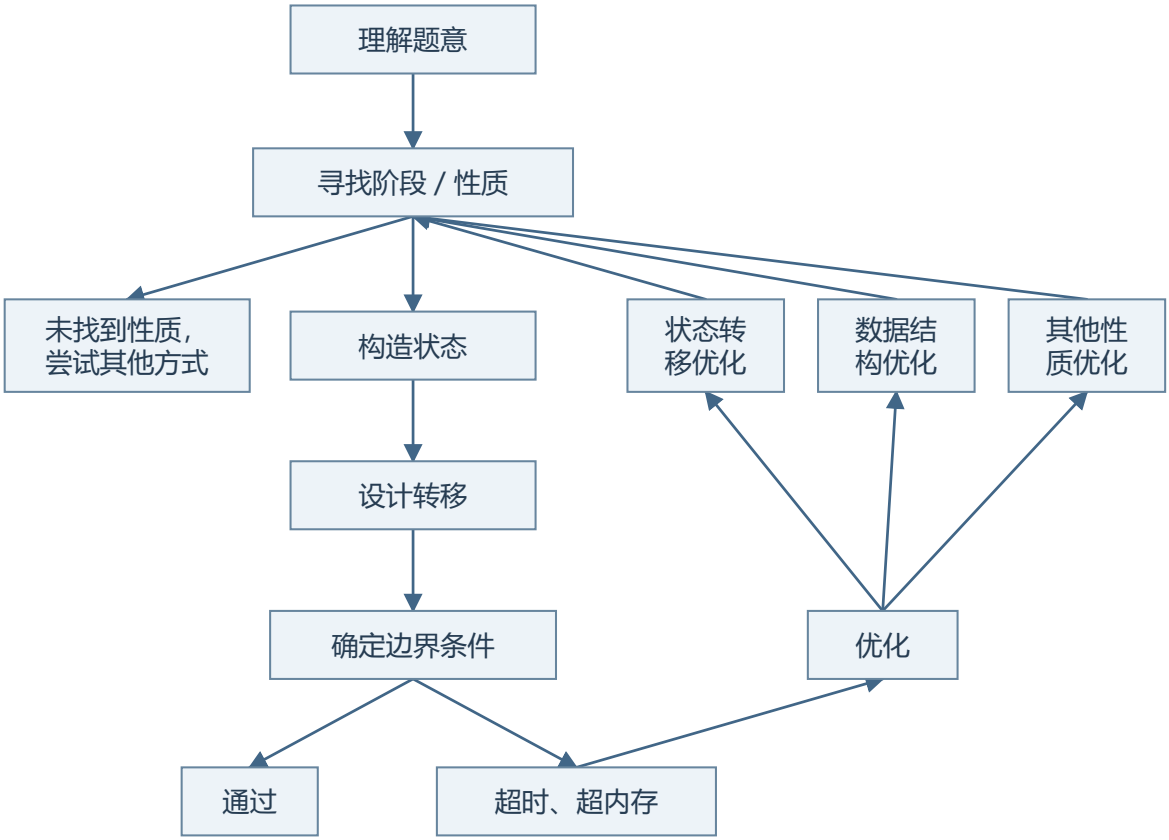



图13-7 动态规划的解题步骤

5. 确定边界条件：边界点通常有两类。一是先验答案点，即答案由题意确定的点，一般是作为整张DAG的起点；二是不完全转移点，即对于一般的转移方程可能会出现越界情况的点。这两类点的处理需要注意规避错误，特别是后一种比较难查错。
6. 优化：动态规划的优化是一门深奥的学问。在信息学竞赛中可能碰到的优化方式包括（但不限于）状态优化、转移优化、数据结构加速、单调性优化（斜率优化、四边形不等式）、FFT加速、反演计数、根号**分治**、CDQ分治等。对于计数问题和最优化问题通常有着不同的优化思路。

关于状态优化的例子在 例 13-4 和 例 13-5 有所讨论。接下来会尝试使用上面动态规划解题步骤来讲解一些比较巧妙的题目，分析解决的思路，并举例涉猎一些其他的简单优化方式。

## 13.5 课后习题与实验

### 习题 1 打怪、吃药、获得经验(洛谷P1802,Byabsi2011)

主角打过第 $i$ 个怪需要吃 $u_i$ 瓶药并获得 $w_i$ 点经验，输了则会获得 $l_i$ 点经验。现在主角拥有 $x$ 瓶药，求能获得的最多经验。

### 习题 2 过河卒(洛谷P1002,NOIP 2002普及组)

棋盘上 $A$ 点（0,0）有一个卒需要走到 $B$ 点（$n$，$m$），卒在棋盘上只能向下或者向右走，同时在棋盘上有一个对方的马，该马所在的点和所有跳跃一步可达的点称为马的控制点，现在要求计算出卒从 $A$ 点能够到达 $B$ 点的路径的条数，假设马固定不动。

### 习题 3 装箱问题(洛谷P1049,NOIP2001普及组)

有一个箱子容量为 $V(0 \leqslant V \leqslant 20000)$，同时有 $n$ 个物品 $(0 < n \leqslant 30)$，每个物品有一个整数体积。

要求从 $n$ 个物品中，任取若干装入箱内，使箱子的剩余空间为最小。输出最小的剩余空间。

### 习题 4 疯狂的采药(洛谷P1616)

某个山洞里有一些不同的草药，采每一株都需要一些时间，每一株也有它自身的价值。在一段时间里，可以采到一些草药。求在一段时间内最大化采到药的总价值。每种草药有无限株。

### 习题 5 小A点菜(洛谷P1164)

小 $A$ 请客，但他口袋里只剩 $M(M \leqslant 10000)$ 元。餐馆虽低端，但是菜品种类不少，有 $N$ 种 $(N \leqslant 100)$，第 $i$ 种卖 $a_i$ 元 $(a_i \leqslant 1000)$。每种菜只有一份。他想知道有多少种点菜方法能把身上的钱花光。

### 习题 6 摆花(洛谷P1077,NOIP2012普及组)

小明的花店新开张，为了吸引顾客，他想在花店的门口摆上一排花，共 $m$ 盆。通过调查顾客的喜好，小明列出了顾客最喜欢的 $n$ 种花，从 $1$ 到 $n$ 标号。为了在门口展出更多种花，规定第种花不能超过 $a_i$ 盆，摆花时同一种花放在一起，且不同种类的花需按标号从小到大的顺序依次摆列。求一共有多少种不同的摆花方案。

**数据范围：** $0 < n \leqslant 100, 0 < m \leqslant 100, 0 \leqslant a_i \leqslant 100$。

### 习题 7 线段(洛谷P3842,天津市队选拔2007)

在一个 $n \times n$ 的平面上，在每一行中有一条线段，第 $i$ 行的线段的左端点是 $(i, L_i)$，右端点是 $(i, R_i)$ 从（$1$，$1$）点出发，要求沿途走过所有的线段，最终到达（$n$，$n$）点，且所走的路程长度要尽量短。更具体一些说，在任何时候只能选择向下走一步（行数增加 $1$）、向左走一步（列数减少 $1$）或是向右走一步（列数增加 $1$）。当然，由于不能向上行走，因此在从任何一行向下走到另一行的时候，必须保证已经走完本行的那条线段。

**数据范围：** $n \leqslant 2 \times 10^4, 1 \leqslant L_i \leqslant R_i \leqslant n$。

### 习题 8 金明的预算方案(洛谷P1064,NOIP2006提高组)

金明有 $n(n \leqslant 32000)$ 元，他想买一些物品。物品有 $m\,(m \leqslant 60)$ 件，分为主件与附件两类，附件是从属于某个主件的。如果要买归类为附件的物品，必须先买该附件所属的主件。每个主件可以有 $0 \sim 2$ 个附件。每个附件对应一个主件，附件不再有从属于自己的附件。

他把每件物品规定了一个重要度，用整数 $1 \sim 5$ 表示。他还查到了每件物品的价格 $v_i(0 \leqslant v_i \leqslant 60)$。他希望在不超过 $n$ 元的前提下，使每件物品的价格与重要度的乘积的总和最大。

### 习题 9 小k考前临时抱佛脚(洛谷P2392)

有 $4$ 个科目的作业，每个科目有不超过 $20$ 题，解决每道题都需要一定的时间。小 $k$ 可以同时处理同一科目的两道不同的题，求他完成所有题目所需要的时间。

**提示：**虽然本题数据范围小到即使枚举子集也不会超时，但可以考虑使用背包模型解决。

# 第14章 线性状态动态规划

本章将介绍**动态规划**中最常规的一类题型—线性状态动态规划，以及其变种问题。

线性状态动态规划，顾名思义，指的是一类状态定义与题设内容线性相关的动态规划。题设中有序列（数组），那动态规划的状态就是一维的；如果题设中有棋盘（网格），那么动态规划的状态就是二维的……线性状态动态规划定义状态时经常会考虑“某类有序事件中前若干子事件的答案”。

一般来说，线性的状态最为直观，容易用有向无环图（DAG）表示，也比较容易理解。所以遇到动态规划问题构造状态时，通常会优先考虑能否采用线性状态。线性状态动态规划思维导图如图 14-1 所示。




图14-1 本章思维导图

## 14.1 单序列问题

### 例 14-1 导弹拦截(洛谷P1020,NOIP1999普及组)

为了防御导弹袭击，现开发出一种导弹拦截系统。该系统的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都不能高于前一发的高度。一套系统可能无法拦截所有的导弹。

输入 $n(n\leqslant 10000)$ 枚导弹依次飞来的高度（不超过 $50000$ 的非负整数），计算这套系统最多能拦截多少导弹，以及如果要拦截所有导弹最少要配备多少套这种导弹拦截系统。

**分析：**求最多可以拦截多少导弹，就是找到导弹高度序列中的一个子序列，使其长度最大且单调不增。这种序列被称为最长不增序列。更形式化的定义：已知一个长度为 $n$ 的整数序列 $a_i$，试找到一个下标递增的整数序列 $p_k(1 \leqslant p_1 < \cdots < p_k \leqslant n)$，使得对任意 $1 \leqslant i < k$，有 $a_{p_i} \geqslant a_{p_{i+1}}$，最大化 $k$。

不加证明地给出一个结论：最少要配备导弹拦截系统数量就是该序列的最长上升子序列长度①。

这是一道经典的“子序列提取”问题。用 $f_i$ 表示前 $i$ 个数中，以第 $i$ 个数结尾的最长不增子序列的长度。明确结尾元素，可以为状态转移提供必要的信息。接下来考虑如何计算 $f_i$，图 14-2 展示了状态和转移的例子：




图14-2 状态和转移

只需要考虑把第 $i$ 个数接在某个已知序列后面，这个序列必然是以 $a_1$ 到 $a_{i-1}$ 中大于或等于 $a_i$ 的某一个数结尾的，而且越长越好。因此，只需要枚举 1 到 $i-1$ 中符合条件的那些，找到最长的序列 ：即可。注意边界条件 $f_i \geq 1$ 以及答案应当是全局最大值。转移方程如下：

$$f_i = 1 + \max(\{0\} \cup \{f_j : j < i,\ a_j \geq a_i\})$$

这里基于一个很显然的性质：最长子序列一定是由前面的某个最长子序列转移而来的，即最长子序列长度满足此题的最优子结构，所以即使在对序列其他信息一无所知、只知道结尾和子序列长度的情况下也能放心转移。像这类只需要考虑相邻事件（“新事件”和“上一个事件”）关系的问题，是“子序列提取”题的一类常见模型。

类似的还有最长上升子序列和最长不下降子序列，区别在于相邻两元素能否相等。在不严格区分的情况下，两者都可以被称为 `LIS(LongestIncreasingSubsequence)` 问题。

```
#include <bits/stdc++.h>
#define maxn 10010
using namespace std;
int ans, a[maxn], f[maxn], n;
int main() {
    while (cin >> a[++n])
        ;
    n--;
    for (int i = 1; i <= n; i++) {  // 求最长不增子序列
        f[i] = 1;                   // 边界条件
        for (int j = 1; j < i; j++) // 枚举1 到 i-1 中小于或等于 a[i]的数
            if (a[j] >= a[i])
                f[i] = max(f[i], f[j] + 1); // 找到最长的那个序列+1
        ans = max(ans, f[i]);               // 寻找全局最大值
    }
    cout << ans << endl;
```

```
    for (int i = 1; i <= n; i++) {
        f[i] = 0;
    }
    ans = 0;
    for (int i = 1; i <= n; i++) {  // 求最长上升子序列
        f[i] = 1;                   // 边界条件
        for (int j = 1; j < i; j++) // 枚举1到i-1 中小于或等于 a[i]的数
            if (a[j] < a[i])
                f[i] = max(f[i], f[j] + 1); // 找到最长的那个序列+1
        ans = max(ans, f[i]);               // 寻找全局最大值
    }
    cout << ans << endl;
}
```

> **注：** 1. 可以用Dilworth定理证明。

当然这道题也可以用DAG上最长路的模型理解，只需要把靠前的较小的数连向靠后的较大的数即可。注意到此题中，数字的前后关系与大小关系这两个维度同时影响DAG的形态，构成一种确定的偏序，故把这种模型称为“二维偏序”。以此类推还有三维偏序等。

这种做法的时间复杂度是 $O(n^2)$，可以通过维护一个存储不上升序列的栈来查找对应的状态，将时间复杂度优化到 $O(n\log n)$.，请读者自行思考与实现。

### 例 14-2 打鼹鼠(洛谷P2285,湖南省队选拔2004)

在一个 $n\times n$ 的网格中，在某些时刻鼹鼠会在某一个网格出现。可以控制一个机器人来打鼹鼠，如果 $i$ 时刻鼹鼠在某个网格中出现，而机器人也处于同一网格的话，那么这个鼹鼠就会被机器人击中。而机器人每一时刻只能够移动一格或停留在原地不动。机器人的移动是指从当前所处的网格移向相邻的网格，即从坐标为 $(i,j)$ 的网格移向 $(i-1,j),(i+1,j),(i,j-1),(i,j+1)4$ 个网格，机器人不能走出整个 $n\times n$ 的网格。

游戏开始时，可以自由选定机器人的初始位置。

按照时间递增的顺序，给出 $m(m\leqslant 10000)$ 个鼹鼠出现的时间 $t_i$ 和地点 $(x_i,y_i)$。请编写一个程序使机器人在这一段时间内打尽可能多的鼹鼠。

**分析：**不难看出，此题也是一道标准的“子序列提取”问题。把所有的鼹鼠按照时间排序，那么只需要找一个子序列，使得相邻两个鼹鼠之间的“直角距离”（曼哈顿距离）不超过出现的时间差。注意到，这个转移成功的条件也仅存在于相邻两个事件中，所以此题与前一题如出一辙。

可以如法炮制：用 $f$ 表示只考虑前 $i$ 只鼹鼠，并且确定打了第 $i$ 只鼹鼠的情况下，最多能打 $f_i$ 多少只鼹鼠。这里确定了第 $i$ 只鼹鼠被击打，是为了确定信息，以便转移。转移方程如下：

$$f_i=1+\max\left(\{0\}\cup\{f_j:j<i,\ |x_i-x_j|+|y_i-y_j|\leq t_i-t_j\}\right)$$

代码如下：

```
#include <bits/stdc++.h>
#define maxm 10010
using namespace std;
int n, m, ans;
int f[maxm], t[maxm], x[maxm], y[maxm];
// f[i]表示打了第i只鼹鼠的情况下,之前一共能打多少鼹鼠
int main() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= m; i++) {
        scanf("%d%d%d", &t[i], &x[i], &y[i]); // t[i]有序
        f[i] = 1;                             // 边界条件
        for (int j = 1; j < i; j++)           // 常数较小,所以m方也能过
            if (f[j] + 1 > f[i] && abs(x[i] - x[j]) + abs(y[i] - y[j]) <= t[i] - t[j])
                f[i] = f[j] + 1;
        ans = max(ans, f[i]);
    }
    printf("%d", ans);
    return 0;
}
```

比较两题解法，可以总结出“子序列提取”问题的常规思路：考虑新加入的数与前面某每个状态之间的关系和转移可行性，如果恰好转移只和相邻两数相关，那么就可以获得一个复杂度至多为 $O(n^2)$ 的**动态规划**。当然情况不可能总是这么顺利，接下来看“子序列提取”问题的几个变体。

### 例 14-3 小琪移动(洛谷P1725)

小琪从 $0$ 号格子出发向后跳。当她在格子 $i$ 时，只移动到区间$[i+l,i+r]$中的任意一格，当跳到大于 $n$ 的格子时视为结束。第 $0$ 个格子到第 $n(n\leqslant 200000)$ 个格子上均有个数字，第 $0$ 格上的数字保证为 $0$。给出 $n$、$l$、$r$ 和这些数字，试最大化经过数字和。

**输入样例：**

```
523
0 12 3 11 7 -2
```

**输出样例：**

```
11
```

**分析：**到现在为止这题已经很套路了，是一个“子序列提取”题，即最大化一个子序列的数字和，使得子序列相邻两数在原序列中相距 $l$ 到 $r$ 个位置。那么已经可以很轻易地想出**动态规划**：用 $f$ 表示从起点出发到格子 $i$ 的最大数字和。转移方程如下：

$$f_i=a_i+\max_{l\leqslant i-j\leqslant r}f_j$$

其复杂度为 $O$（`n(r-l)`），无法通过本题，需要进行优化。仔细观察上面如此简洁的方程，发现是一个简单的（带修）RMQ问题：找到数组 $f$ 在某个区间的最小值。

可能最先想到的是强大的**线段树**，但是这样时间复杂度就到了 $O(n\log n)$，有继续优化的余地。重新思考查询区间，它其实相当于一个滑动窗口，那么应用一个**单调队列**就能解决了。代码如下：

```
#include <bits/stdc++.h>
#define maxn 200010
#define inf 210000000
using namespace std;
int n, l, r;
int a[maxn], f[maxn], q[maxn];
// f[i]表示从起点出发到格子i的最大数字和
// q是单调队列,记录了f[i-r]到f[i-1]之间的单调性信息,保存下标
int main() {
    cin >> n >> l >> r;
    for (int i = 0; i <= n; i++)
        cin >> a[i];
    int head = 0, tail = 0;
    int ans = -inf;
    f[0] = 0; // 初始化
    for (int i = 1; i <= n; i++)
        f[i] = -inf;
    for (int i = l; i <= n; i++) {
        // 1到1-1的格子完全跳不到,注意区分数字1和字母1
        while (head < tail && f[i - l] >= f[q[tail - 1]])
            tail--;        // 剔除不比f[i-1]大、还比f[i-1]靠前的数
        q[tail++] = i - l; // 加入f[i-1]
        while (head < tail && q[head] < i - r)
            head++; // 删除已经过期的数据
        f[i] = f[q[head]] + a[i];
        if (i + r > n)
            ans = max(ans, f[i]);
    }
    cout << ans;
}
```

考虑到答案可能是负数，所以需要将数组 $f$ 和表示答案的 `ans` 变量先初始化为负无穷，当下一步可以跳到 $n$ 后面时，即可比较统计最优答案。

可以看到，数据结构优化动态规划通常是有迹可循的，即十分清楚需要一个支持什么操作的数据结构。这类题的动态规划部分相对思考难度较低，考查点一般是数据结构的应用。

### 例 14-4 大师赛(洛谷P4933)

给出一个非负数列，求它有多少个等差子序列。数列长度 $n \leqslant 1000$，最大数 $v \leqslant 20000$。

**输入样例：**

```
8
13 14 6 20 27 34 34 41
```

**输出样例：**

**分析：**虽然形式上变成了计数，但这还是一个“子序列提取”问题，只不过这次转移的条件不再只取决于相邻两数。想要保证加入的下一个数能是等差数列，得知道再上一个数或者公差，也就是需要记录的信息还得更多。给**动态规划**加上公差这一维度：用 $f_{i,d}$ 表示以第 $i$ 个数结尾，公差为 $d$（可以是负数）的子序列（长度不少于 2）①的个数，那么就可以只考虑相邻事件，很快得到合理的一个转移方程：

$$f_{i,d} = \sum_{\substack{j<i \\ a_i=a_j+d}} (f_{j,d}+1)$$

然而时间复杂度是 $O(n^2v)$ 的。这里有个比较明显的冗余是（除了单个数构成的等差数列以外）以第 $i$ 个数结尾的等差数列的公差个数十分有限，很明显不超过 $i-1$ 个。也就是说，完全没有必要把 $f$ 填满（它大多数的答案是 0），只需要枚举再之前的数就可以枚举出所有有效公差。给转移方程换个写法：

$$f_{i,a_i-a_j} = \sum_{j<i} (f_{j,a_i-a_j}+1)$$

很明显，这样需要枚举的两个维度都不超过 $n$，所以时间复杂度就是 $O(n^2)$ 了。而空间复杂度是 $O(nv)$，也能够胜任。

```
#include <bits/stdc++.h>
#define maxn 1010
#define maxv 20010
#define mod 998244353
using namespace std;
int f[maxn][maxv * 2], n, a[maxn], ans;
// f[i][d + maxv]表示以a[i]结尾公差为d的等差数列的方案数
int main() {
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        for (int j = 1; j < i; j++) {
            (f[i][a[i] - a[j] + maxv] += f[j][a[i] - a[j] + maxv] + 1) %= mod;
            // 下标加个偏移可以实现负数数组
            (ans += f[j][a[i] - a[j] + maxv] + 1) %= mod;
            // 注意在这一维上每次存取都必须加上偏移
        }
    }
    cout << (ans + n) % mod; // 把单个数等差数列算上
}
```

本题考虑先枚举 $d$，然后对每个 $d$ 做一次动态规划（用 $f_{i,d}$ 表示以 $a_i$ 结尾、公差为 $d$ 的等差数列个数），再用一个辅助数组 $g_k$ 记录当前以数 $k$（不是第 $k$ 个数 $a_k$） 结尾的等差数列的方案数，可以做到 $O(nv)$ 时间复杂度， $O(n+v)$ 空间复杂度，具体实现留给读者。

> **注：** 1. 这里要求长度不少于 2 是为了让公差具有良好的定义，在动态规划中定义不规范的东西会给思维和调试带来很多麻烦。

### 例 14-5 快速求和(洛谷P1874)

给定一个数字字符串 $S$（长度不超过 $40$），在其中插入最少的加号来让算式的答案等于一个给定的目标数字 $t$（小于 $10^6$） 。若无论如何都不能让答案成为目标数字，输出 $-1$。

**输入样例：**

```
2222 8
```

**输出样例：**

**分析：**这是一道同时具有“子段划分”和“背包”两个模型的题，既要考虑把原串划分成若干段，又要让它们的和等于目标数字。因为不需要区分段的个数，只要最小化它，所以状态记录的是某个局面的最少加号。记 $f_{i,s}$ 表示只使用字符串前 $i$ 位，达到数字和恰为 $s$ 的最少加号数。那么考虑 $f_{i,s}$ 的前继，就只需要知道具体的“最后一段”，那么只要枚举这个“最后一段”的起点就行了。

记 $v_{i,j}$ 表示字符串从第 $i$ 位到第 $j$ 位所代表的数值，那么就有状态转移方程：

$$f_{i,s}=1+\min_{j<i} f_{j,s-v_{j+1,i}}$$

这个方程的意义是，把字符串的第+1 到 $i$ 这段数字提出来，接在某个以 $j$ 结尾的数字和为 $s-v_{j+1,i}$ 的加法串上，就能得到以 $i$ 结尾，数字和为 $s$ 的加法串了。而本题任务只是找到那个加号数量最小的串，作为新状态的答案。目前答案里还会出现一些不可能凑到的和，只需要简单地先把整个数组 $f$ 置为正无穷大，再设置好唯一起点 $f_{0,0}$ 就行了。

可是现在的时间复杂度是 $O(n^2t)$，有点危险，所以仍然可以剪枝：从后往前枚举 $j$，当 $v_{j+1,i}$ 超过 $t$ 时便不再继续，这可以将时间复杂度控制在 $O(nt\log t)$，可以很稳健地通过所有数据点。代码如下：

```
#include <bits/stdc++.h>
#define maxn 50
#define maxs 100010
#define INF 0x7F7F7F7F
using namespace std;
int f[maxn][maxs], t, n, num[maxn][maxn];
string S;
// f[i][s]表示使用s的前i位获得s的和的最少加号数
int main() {
    cin >> S >> t;
    n = S.length();
    memset(f, 127, sizeof(f));
    f[0][0] = -1;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= i; j++) {
            if (num[j][i - 1] > t)
                num[j][i] = INF;
            else
                num[j][i] = num[j][i - 1] * 10 + S[i - 1] - 48; // 预处理出 s
            // [j..i]对应的数字
        }
    for (int i = 1; i <= n; i++)
        for (int s = 0; s <= t; s++)
            for (int j = i - 1; j >= 0 && num[j + 1][i] <= t; j--) // s减去j+1 到
                // i 构成的数字
                if (s >= num[j + 1][i])
                    f[i][s] = min(f[i][s], f[j][s - num[j + 1][i]] + 1);
    cout << (f[n][t] < maxn ? f[n][t] : -1);
}
```

## 14.2 多序列问题

### 例 14-6 最长公共子序列(经典问题)

给出两个字符串 $a$，$b$，求它们的最长公共子序列，字符串长度不超过 $2000$。例如，当两个字符串分别是acdfg和afhgf，其最长公共子序列是 $afg$，长度是 $3$.。

**分析**：这里有两个序列，把**动态规划**的维度从一维拓展到二维。仿照线性状态动态规划的方程一般表示方法：用 $f_{i,j}$ 表示字符串 $a$ 前 $i$ 个字符与字符串 $b$ 前 $j$ 个字符的最长公共子序列长度（Longest Common Subsequence，LCS）。

考虑 $f_{i,j}$ 的来源：可以对LCS不产生影响，如在 $f_{i-1,j}$ 上加 $a$ 的一个字符，或者是在 $f_{i,j-1}$ 上加 $b$ 的一个字符，这些单个的新字符无法加入LCS，所以没有影响；也可以对LCS产生影响，如当字符 $a_i$ 与 $b_j$ 相等时，可以在 $f_{i-1,j-1}$ 上同时加 $a_i$ 和 $b_j$ 并把它们加入LCS，此时LCS长度加 $1$ 那么就可以得到以下状态转移方程：

$$f_{i,j}=\max(f_{i-1,j},f_{i,j-1},f_{i-1,j-1}+[a_i=b_j])$$

其中 $[P]=1$，若 $P$ 为真；否则为 $0$。事实上，在 $a_i\neq b_j$ 的时候可以不用特殊考虑 $f_{i-1,j-1}$，读者可以自行证明。

```
#include <bits/stdc++.h>
#define maxn 2010
using namespace std;
int f[maxn][maxn];
// f[i][j]表示字符串a的前i个字符与字符串b的前j 个字符的LCS 长度
string a, b;
int main() {
    cin >> a >> b;
    int n = a.length(), m = b.length();
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            f[i][j] =
                max(max(f[i - 1][j], f[i][j - 1]),
                    f[i - 1][j - 1] + (a[i - 1] == b[j - 1])); // string从0开始,偏移-1位
    cout << f[n][m];
}
```

### 例 14-7 编辑距离(洛谷P2758)

设 $A$ 和 $B$ 是两个只包括小写字母的字符串。要用最少的字符操作次数，将字符串 $A$ 转换为字符串 $B$。这里所说的字符操作共有以下 $3$ 种：

操作 $1$：删除一个字符。

操作 $2$：插入一个字符。

操作 $3$：将一个字符改为另一个字符。

例如，当两个字符串分别是sfdqxbw和 gfdgw，需要将 $s$ 改为 $g$，$q$ 改为 $g$，删除 $x$ 和 $b$，一共操作 $4$ 次。

**分析：**因为此题的字符串长度似乎在操作中发生了变化，不好入手。仍然可以尝试大胆定义状态：用 $f_{i,j}$ 表示将字符串 $a$ 前 $i$ 个字符完全转换成字符串 $b$ 前 $j$ 个字符的最少操作次数。这样只需要考虑新的字符会对前继产生什么影响就行了。

还是考虑 $f_{i,j}$ 的来源：可以是 $f_{i-1,j}$ 多加个 $a$，需要多用一步操作 1 删掉这个新字符；也可以 $a_i$ 是 $f_{i,j-1}$ 多加一个 $b_j$，需要多用一步操作 2 来加上这个新字符；也可以是 $f_{i-1,j-1}$，如果 $a_i \neq b_j$，还需要多用一步操作 3 来改变这个不相同的字符，否则不用额外操作。那么很容易给出状态转移方程：

$$f_{i,j}=\min(f_{i-1,j}+1, f_{i,j-1}+1, f_{i-1,j-1}+[a_i \neq b_j])$$

注意边界条件，非空字符串和空字符串互相转换需要的操作次数等于非空字符串的长度。

```
#include <bits/stdc++.h>
#define maxn 2010
using namespace std;
int n, f[maxn][maxn];
// fi][j表示将a的前i个字符完全转换成b的前j个字符的最少操作次数
string a, b;
int main() {
    cin >> a >> b;
    int n = a.length(), m = b.length();
    for (int i = 1; i <= n; i++)
        f[i][0] = i;
    for (int j = 1; j <= m; j++)
        f[0][j] = j;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            f[i][j] = min(min(f[i - 1][j] + 1, f[i][j - 1] + 1),
                          f[i - 1][j - 1] + (a[i - 1] != b[j - 1]));
    cout << f[n][m];
}
```

### 例 14-8 最长公共子序列(洛谷P1439)

给出 $1$ 到 $n(n\leqslant 10^5)$ 的两个排列 $P_1$ 和 $P_2$ 排列是指 $1$ 到 $n$ 出现一次。求它们的最长公共子序列。

**输入样例：**

```
5
31 42 5
41325
```

**输出样例：**

**分析：**对于两个排列的LCS，可以使用一个技巧把它转换为LIS，即将第 2 行的数字替换成{1，2，…，$n$}，同时也对第 1 行的相同数字做这一替换，可以将样例转为：

```
32145
12345
```

可以看到第 2 行已经有序了，但由于只做了位置不变的字符替换，两序列的本质没有变，这样第 1 行任意一个子序列都可以是上下两行的公共子序列，也对应了原问题的一个公共子序列（位置不变，做逆替换可以得到）。

这个操作有多种实现方式，这里通过记录置换{4，1，3，2，5}→{1，2，3，4，5}（4 变 1，1 变 2…），再把这个置换应用于{3，1，4，2，5}来获得新的第 1 行。

接下来，只要尝试优化LIS的 $O(n^2)$ 算法就能解决这个问题了。回顾LIS的状态转移方程：

$$f_i = 1 + \max(\{0\} \cup \{f_j : j < i,\ a_j < a_i\})$$

可以看到max有两个限制 $\omega_{j<i}$ 和 $a_j < a_i$。如果考虑时序，那么在处理 $f_i$ 的时候，所有已知的 $a_j$ 必然在 $a_i$ 前面，也就是说 $j < i$ 自然满足。

把 $a_j$ 想象成某个数组的下标 $\mathcal{I}_j$ 想象成记录在那个数组里的数。现在需要一个这样的数据结构：可以查询那个数组下标不超过 $a_i$ 的所有元素最大值，并且可以支持对某个位置的值修改。简单来说，此时又回到了带修改的查询区间最值问题，只不过这次它是一个求前缀最小值问题。

本书在前面章节中介绍过**树状数组**，树状数组能处理区间询问的前提是记录的信息可以执行区间减法。最小值看起来不能进行区间减法，但这里需要的仅仅是维护前缀最小值，而记录前缀信息也恰好是树状数组最原始的功能。在查询时，寻找树状数组中下标（位置）不超过 $a_i$ 的最大元素。更新时，把 $f_i$ 的值写到 $a_i$ 所在的位置，并维护这个树状数组即可。代码如下

```
#include <bits/stdc++.h>
#define maxn 100010
using namespace std;
int n, a[maxn], b[maxn], f[maxn], c[maxn], r[maxn], ans;
void ins(int k, int x) { // 修改
    for (; k < maxn; k += k & -k)
        c[k] = max(c[k], x);
}
int ask(int k) { // 询问
    int ans = 0;
    for (; k; k -= k & -k)
        ans = max(c[k], ans);
    return ans;
}
int main() {
    cin >> n;
    for (int i = 1; i <= n; i++)
        cin >> a[i];
    for (int i = 1, tmp; i <= n; i++)
        scanf("%d ", &tmp), r[tmp] = i; // 做映射, tmp->i
    for (int i = 1; i <= n; i++)
        f[i] = ask(r[a[i]]) + 1,  // 对查询结果加1
            ins(r[a[i]], f[i]),   // 更新树状数组
            ans = max(ans, f[i]); // 顺便更新全局答案
    cout << ans;
}
```

### 例 14-9 子串(洛谷P2679,NOIP2015提高组)

有两个仅包含小写英文字母的字符串 $A$ 和 $B$。现在要从字符串 $A$ 中取出 $k$ 个互不重叠的非空子串（子串要求元素是连续的，子序列不需要），然后把这 $k$ 个子串按照其在字符串 $A$ 中出现的顺序依次连接起来，得到一个新的字符串。请问：有多少种方案可以使得这个新串与字符串 $B$ 相等？答案对 $1000000007$ 取模。

注意：子串取出的位置不同也认为是不同的方案。例如，从aabaab中取出 2 个不重叠的子串，有 7 种方案。

**分析：**从题目出现的元素来说，这是一道双序列子段划分题。既然要在字符串 $A$ 取出 $k$ 段去匹配字符串 $B$，最好把子段数和两个序列的位置都放在状态的定义里：记 $f_{k,i,j}$ 表示字符串 $A$ 以 $i$ 个字符结尾取出 $k$ 段后，在字符串 $B$ 恰好匹配到 $j$ 的方案数。

先考虑最朴素的转移：$f_{k,i,j}$ 在大部分情况下为 $0$，除非 $A_i = B_j$。当 $A_i = B_j$ 时，$A_i$ 要么跟在上一段后面，要么新起一段。

情况 1 比较简单：段数没变，前继的段数仍为 $k$，两个字符串的指针分别为 $i-1$ 和 $j-1$。

情况 2 稍微复杂：前继的段数为 $k-1$，字符串 $B$ 中的指针为 $j-1$，而字符串 $A$ 中的指针可以追溯到 $i$ 之前的任意位置。

状态数已经是 $O(mnk)$ 级别（甚至已超出空间限制），因此每次转移必须控制在 $O(1)$。使用辅助数组记录 $f$ 在第 2 维上的**前缀和**：$g_{k,i,j}$ 表示取出 $k$ 段后，在字符串 $A$ 的前 $i$ 个位置内结尾、匹配完字符串 $B$ 前 $j$ 个字符的方案数。即：

$$g_{k,i,j} = \sum_{x=1}^{i} f_{k,x,j}$$

这样，新起一段时直接使用 $g_{k-1,i-1,j-1}$，就能获得所有合法前继，完成 $O(1)$ 转移。

因为三维数组超出了空间，但是 $k$ 这一维只和上一阶段相关，所以可以用滚动数组降低空间复杂度。代码如下：

```
#include <bits/stdc++.h>
#define maxn 1005
#define maxm 205
#define mod 1000000007
using namespace std;
int f[2][maxn][maxm], g[2][maxn][maxm], n, m, k;
char a[maxn], b[maxm], ans;
int main() {
    scanf("%d%d%d%s%s ", &n, &m, &k, a + 1, b + 1);
    for (int i = 0; i <= n; i++)
        g[0][i][0] = 1;
    int cr = 1; // 当前行(current line),cr^1 表示上一行
    for (int K = 1; K <= k; K++, cr ^= 1) {
        memset(f[cr], 0, sizeof(f[cr]));
        memset(g[cr], 0, sizeof(g[cr]));
        for (int i = 1; i <= n; i++)
```

```
            for (int j = 1; j <= m; j++) {
                if (a[i] == b[j])
                    f[cr][i][j] = (f[cr][i - 1][j - 1] + g[cr ^ 1][i - 1][j - 1]) % mod;
                g[cr][i][j] = (g[cr][i - 1][j] + f[cr][i][j]) % mod;
            }
    }
    printf("%d\n", g[cr ^ 1][n][m]); // 该项恰好代表了所求答案
    return 0;
}
```

## 14.3 高维问题

### 例 14-10 三倍经验(洛谷P1544)

有一个数字金字塔，同时有 `k(k≤n)` 次机会将上面的任意一个数乘以 $3$（不能对同一个数重复乘），最大化从最高点到底部任意一点的最大数字和。每一步可以走到左下方的点也可以到达右下方的点。层数 $n \leqslant 100$。

```
7
3 9
8 1 0
2 7 4 4
4 5 2 6 5
```

在样例中，若 $k=1$，则 $7 \rightarrow 9 \times 3 \rightarrow 1 \rightarrow 7 \rightarrow 5$ 的路径产生了最大数字和。

**分析：**和 例 13-2“数字三角形”题类似，本题也有一个显性的塔形DAG，但是关键在于发现这题隐藏在塔形DAG下的其实是个分层图。题目中的操作包括左下走、右下走和乘以 3，决定把所有的操作都视作转移。即把乘以 3 视作一种技能，把技能的使用次数也加入阶段。

先定义出状态：用 $f_{i,j,k}$ 表示从第 $i$ 行第 $i$ 列出发到最底层，恰使用了 $k$ 次乘以 3 的最大数字和。那么在每个点，它的前继状态（对应原题的下一步）有 4 个选项：走左下不乘以 3；走右下不乘以 3，走左下乘以 3，走右下乘以 3。分别对应了 4 个局面：$f_{i+1,j,k}+a_{i,j}, f_{i+1,j+1,k}+a_{i,j}, f_{i+1,j,k-1}+3a_{i,j}$ 和 $f_{i+1,j+1,k-1}+3a_{i,j}$

虽然显性的DAG只有这个单独的金字塔，但如果把金字塔复制 $k+1$ 份，依次层叠，分别表示使用了若干（从 0 到 $k$）次乘以 3 技能，如图 14-3 所示。这时就又回到了熟悉的DAG最大权值和问题：从若干个起点（最底层，没有用过的情况）出发，到达某些终点。

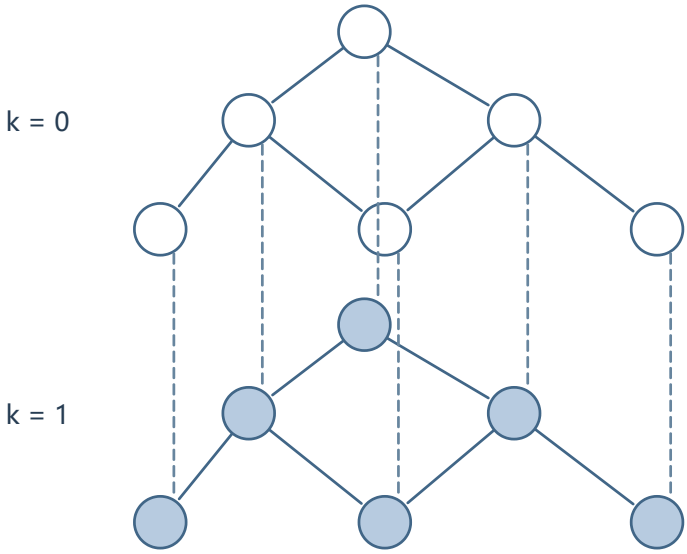



图14-3 分层图

```
#include <bits/stdc++.h>
using namespace std;
int a[110][110], f[110][110][110], n, K, ans = -2000000000;
// 用f[x][y]k]表示从第x行第y列出发用了k次乘以3道具走到底层的最大数字和
int main() {
    scanf("%d%d", &n, &K);
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= i; j++)
            scanf("%d", &a[i][j]);
    memset(f, 0x80, sizeof(f));
    for (int j = 1; j <= n; j++)
        f[n][j][0] = a[n][j], f[n][j][1] = a[n][j] * 3; // 边界
    for (int i = n - 1; i >= 1; i--)
        for (int j = 1; j <= i; j++)
            for (int k = 0; k <= K; k++) {
                f[i][j][k] = max(f[i + 1][j][k], f[i + 1][j + 1][k]) + a[i][j];
                // 不使用道具,正常转移
                if (k >= 1)
                    f[i][j][k] = max(f[i][j][k],
                                     max(f[i + 1][j][k - 1], f[i + 1][j + 1][k - 1]) +
                                         a[i][j] * 3);
                // 使用道具,则道具数+1,数字乘以3
            }
    for (int i = 0; i <= K; i++)
        ans = max(ans, f[1][1][i]);
    printf("%d", ans);
    return 0;
}
```

### 例 14-11 方格取数(P1004加强版)

设有 $n \times m (2 \leqslant n, m \leqslant 200)$ 的矩阵，每个格子里有不同的正整数。某人从图的左上角出发，可以向下走，也可以向右走，直到到达右下角。在走过的路上，他可以取走方格中的数（取走后的方格中数字将变为 $0$）。

此人从左上角格子到右下角格子共走两次，试找出 $2$ 条这样的路径，使得取得的数之和为最大。

**输入样例：**

```
3 3
1 4 4
5 1 4
5 5 1
```

**输出样例：**

```
29
```

**分析：**首先，用最基本的**动态规划**模拟走两遍不一定是最优的，读者可以自行构造反例，本题必须从双管齐下的角度人手。那么首先要想到一种方案来存储两条路线的状态，先想到的一般是 $f_{i,j,k,l}$ 来表示路径 1 走到点 $(i,j)$，路径 2 走到（$k$，$I$）的状态。那想想怎么解决一个格子路过两遍只能收益一次的问题。按照这个状态，没法知道在某个状态，一个点有没有被路径 1 或者路径 2 遍历过。

两条路径每次同时走一步，所以始终满足 $i+j=k+l$。若两条路径经过同一个点，它们必然在同一步到达。因此可以去掉一个冗余的坐标状态，并在两条路径当前位置重合时只计算一次该点的权值。

顺便化简状态，记 $s=i+j=k+l$，相当于是走过的步数（加 2），用状态 $f_{s,i,k}$ 表示走了 $s-2$ 步以后路径 1 在第 $i$ 行（第 $s-i$ 列），路径 2 在第 $k$ 行（第 $s-k$ 列），就能不带冗余地表示出所有需要的状态了。

接下去的转移相信对读者来说已经很轻松了 $\cdot f_{s,i,k}$ 这个状态的前继，是由 4 个前继转移过来的，每条路径都可以选择向下或者向右，即 $f_{s-1,i,k}, f_{s-1,i-1,k}, f_{s-1,i,k-1}$ 和 $f_{s-1,i-1,k-10}$ 多出来的贡献是这两个当前点数值和，如果是同一个点就只加一次。

```
#include <bits/stdc++.h>
#define maxn 202
using namespace std;
int dp[2 * maxn][maxn][maxn], a[maxn][maxn], m, n;
int main() {
    cin >> m >> n;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            cin >> a[i][j];
    for (int s = 2; s <= n + m; s++)
        for (int i = max(s - m, 1); i <= min(n, s - 1); i++)
            for (int k = max(s - m, 1); k <= min(n, s - 1); k++) {
                dp[s][i][k] = dp[s - 1][i][k];
                dp[s][i][k] = max(dp[s][i][k], dp[s - 1][i - 1][k - 1]);
                dp[s][i][k] = max(dp[s][i][k], dp[s - 1][i][k - 1]);
                dp[s][i][k] = max(dp[s][i][k], dp[s - 1][i - 1][k]);
                if (i != k)
                    dp[s][i][k] += a[i][s - i] + a[k][s - k];
                else
                    dp[s][i][k] += a[i][s - i];
            }
    cout << dp[n + m][n][n];
    return 0;
}
```

## 14.4 课后习题与实验

### 习题 1 合唱队形(洛谷P1091,NOIP2004)

$N(2 \leqslant N \leqslant 100)$ 位同学站成一排，音乐老师要请其中的 $N-K$ 位同学出列，使得剩下的 $K$ 位同学排成合唱队形。合唱队形是指这样的一种队形：设 $K$ 位同学从左到右依次编号为 $1$，$2$，…，$K$，他们的身高分别为 $T_1, T_2, \cdots, T_K$，则他们的身高满足 $T_1 < \cdots < T_i > T_{i+1} > \cdots > T_K (1 \leqslant i \leqslant K)$

已知所有 $N$ 位同学的身高，计算最少需要几位同学出列，可以使得剩下的同学排成合唱队形。

### 习题 2 回文词(洛谷P1435,IOI2000)

回文词是一种对称的字符串。任意给定一个字符串，长度不超过 $1000$，通过插入若于字符，都可以变成回文词。此题的任务是，求出将给定字符串变成回文词所需要插入的最少字符数。

### 习题 3 花店橱窗布置(洛谷P1854)

花店有花束数 $F$ 和花瓶数 $V(1 \leqslant F \leqslant 100, F \leqslant V \leqslant 100)$。花束可以移动，并且每束花用 $1$ 到 $F$ 的整数标识。如果 $I<J$ 则花束 $I$ 必须放在花束 $J$ 左边的花瓶中。当各个花瓶中放入不同的花束时，会产生不同的美学效果，并以美学值（一个绝对值不超过 $100$ 的整数）来表示，空置花瓶的美学值为 $0$。求使花的摆放取得最大的美学值并输出方案。如果具有最大美学值的摆放方式不止一种，则输出任何一种方案即可。

### 习题 4 樱花(洛谷P1833)

小爱在后院里种了 $n(n \leqslant 1000)$ 棵樱花树，每棵都有美学值 $C_i(0 \leqslant C_i \leqslant 200)$。小爱每天上学前都会来赏花。有一种樱花树看过一遍，一种樱花树最多看 $A_i(0 \leqslant A_i \leqslant 100)$ 遍，一种樱花树可以看无数遍。

但是看每棵樱花树都有一定的时间 $T_i(0 \leqslant T_i \leqslant 100)$。小爱离去上学的时间只剩下一小会儿了（输入数据中将给出，不超过 $1000$）。求解看哪几棵樱花树能使美学值最高且小爱能准时（或提早）去上学。

### 习题 5 奶牛展览(洛谷P2340,USACO2003年秋季月赛)

小贝对 $N$ 头奶牛进行了面试，确定了每头奶牛的智商和情商 $S_i$ 和 $F_i$( 小贝有权选择让哪些奶牛参加展览。小贝不希望出展奶牛的智商之和小于零，或情商之和小于零。在此条件下她希望出展奶牛的智商与情商之和越大越好，请帮助小贝求出这个最大值。

**数据范围：** $1 \leqslant N \leqslant 400, -1000 \leqslant S_i; F_i \leqslant 1000$。

### 习题 6 乌龟棋(洛谷P1541,NOIP2010提高组)

乌龟棋的棋盘是一行 $N$ 个格子，每个格子上一个分数（非负整数）。棋盘第 $1$ 格是唯一的起点，第 $N$ 格是终点，游戏要求玩家控制一个乌龟棋子从起点出发走到终点。棋子自动获得起点格子的分数。

乌龟棋中有 $M$ 张爬行卡片. 分成 $4$ 种不同的类型，每种类型的卡片上分别标有 $1$，$2$，$3$，$4$ 这 $4$ 个数字之一，表示使用这种卡片后，乌龟棋子将向前爬行相应的格子数。游戏中，玩家每次选择一张之前没有使用过的爬行卡片，控制乌龟棋子前进相应的格子数，得到该格子相应的分数。每张卡片只能使用一次。已知棋盘上每个格子的分数和所有的爬行卡片，求最多能得到多少分。

**数据范围：** $1 \leqslant N \leqslant 350, 1 \leqslant M \leqslant 120, 0 \leqslant a_i \leqslant 100$，且 $4$ 种爬行卡片，每种卡片的张数不会超过 $40$。

### 习题 7 绝世好题(洛谷P4310)

给定一个长度为 $n(n \leqslant 10^5)$ 的数列 $a_i$，求 $a_i$ 的子序列 $b_i$ 的最长长度 $k$，满足 $b_i \& b_{i-1} \neq 0$，其中 $2 \leqslant i \leqslant k$，&表示位运算取与。

### 习题 8 262144(洛谷P3147,USACO2016年公开赛)

有一个长为 $n(2 \leqslant n \leqslant 262144)$ 的正整数序列，范围为$[1, 40]$。在一步中，可以选相邻的两个相同的数，然后合并成一个比原来的大 $1$ 的数（如两个 $7$ 合并成一个 $8$），求最大化出现过的最大的数。

# 第15章 区间与环形动态规划

在学习了线性**动态规划**后，本章将会讨论两个稍显复杂模型：区间动态规划，及其变种环形动态规划。这两类动态规划的特点是，囿于问题本身限制，“某类有序事件中前若干个事件的子答案”不再能支撑状态转移，需要去寻找“从某个元素起到另一个元素结束所包含所有的（连续）元素的子答案”作为状态。

可以想象，在这个描述下，每个状态会对应于原题序列上的一个区间。区间具有良好的性质：短的区间可以拓宽成长的区间，相同长度的区间互相不包含。这样，可以把所有状态理解成DAG，即不会有可以互相到达的局面。基于这个原则，可以思考如何构造转移，并在例题中深入理解区间与环形动态规划。本章思维导图如图 15-1 所示。




图15-1 本章思维导图

## 15.1 区间动态规划

### 例 15-1 回文串(洛谷P1435,IOI2000)

回文串是一种对称的字符串。任意给定一个字符串，长度不超过 **1000**，通过插入若干字符，都可以变成回文串。本题任务是，求出将给定字符串变成回文串所需要插入的最少字符数。例如，当输入样例是Ab3bd，插入 **2** 个字符变为dAb3bAd就是回文串。

**分析：**这是一道很经典的回文模型问题，会发现很难使用 第14章 介绍的线性**动态规划**的思路来解决。为了确保读者能准确理解后续的讨论，先来看看一个小剧场：

Q0：怎么把“***a***”“aa”和“qaq”变成回文串？

A0：它们已经是回文串了。

**问题 1：** 怎么把 `qa` 变成回文串？

**解答：** 在左边添加 `a`，得到 `aqa`；或者在右边添加 `q`，得到 `qaq`。

**问题 2：** 怎么把 `qaa` 变成回文串？

**解答：** `aa` 已经是回文串。让它两边的字符相同，即在 `qaa` 右边添加 `q`，得到 `qaaq`。

**问题 3：** 怎么把 `aqaqas` 变成回文串？

**解答：** `aqaqa` 已经是回文串。在左边添加 `s`，得到 `saqaqas`。

从以上的小剧场可以看出，如果要一个串和回文串“很接近”时，可以很轻松地将其变成回文串。再详细点：对于如何把一个长度为 $n$ 的字符串 $S$ 变成回文串这个问题，可以：

1. 先把 $S_{1,n-1}$ 变成回文串，再把 $S_n$ 接在整个串的左边。
2. 先把 $S_{2,n}$ 变成回文串，再把 $S_1$ 接在整个串的右边。
3. 如果 $S_1$ 和 $S_n$ 相同，那只要把 $S_{2,n-1}$ 变成回文串就行了。

这个递归结构给出了一个通过添加字符把字符串变成回文串的有效算法，而且这个方案插入的字符数是最少的，也就是说这个问题具有最优子结构的性质，能够使用动态规划解决，其证明留给读者。

下一步把这个子结构翻译成动态规划的语言：考虑区间上的状态 $f_{i,j}$ 表示把从字符串第个位置到第 $j$ 个位置所表示的子串（记为 $S_{l,r}$） 变成回文串所需要的最少插入字符数。按照该状态定义，最终答案会放在 $f_{1,n}$ 里。

在计算 $f_{i,j}$ 的时候，只需要考察 3 种情况 $\cdot f_{i+1,j}, f_{i,j-}$ 1 和 $f_{\ell}-1,j-1$，分别对应以上子结构的 3 类：

1. $f_{i,j}$ 可能是 $f_{i,j-1}+1$，即把 $S_{i,j-1}$ 变成回文串，再把 $S_j$ 接在这个串的左边。
2. $f_{i,j}$ 可能是 $f_{i+1,j}+1$，即把 $S_{i+1,j}$ 变成回文串，再把 $S_i$ 接在这个串的左边。
3. 如果 $S_i=S_j$，那么 $f_{i,j}$ 可能是 $f_{i+1,j}$，即只需要把 $S_{i+1,j-1}$ 变成回文串。

区间动态规划常见的转移策略如图 15-2 所示。

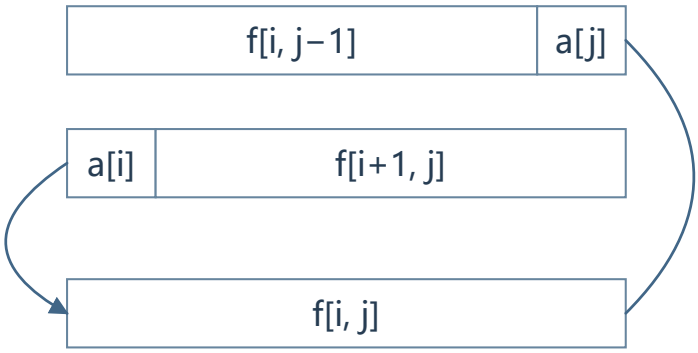



图15-2 区间动态规划常见的转移策略

现在，已经差不多为这道题构建完了动态规划理论基础，接下来要考虑实现的问题：如何选择正确的**拓扑排序**以及如何处理边界情况。

在这个规则中，因为区间 $[i,j]$ 的答案涉及 $[i,j-1],[i+1,j],[i+1,j-1]3$ 个区间的情况，所以较短区间的答案需要比较长区间的答案先算出来，否则会出现依赖问题。那么不如把区间长度 $L$ 作为阶段来入手：让区间长度 $L$ 递增，然后考虑所有长度为 $L$ 的区间（即 $\cdot j=L$ 的区间的情况），这样可以保证此时长度为 $L-1$ 和 $L-2$ 的区间的答案已经是计算正确的了。

至于边界情况，可以看到在 $L=0$ 和 $L=1$ 的答案需要提前算出，好在这两类情况极其平凡，处理方法详见代码：

```
#include <cstdio>
#include <iostream>
#include <string>
#define maxn 1010
using namespace std;
string S;
int f[maxn][maxn], n;
// f[i]i]表示把从字符串第i个位置到第j个位置所表示的子串变成回文串的最小插入字符数
int main() {
    cin >> S;
    n = S.length();
    for (int i = 0; i < n; i++) // 边界条件L=0
        f[i][i] = 0;
    for (int i = 1; i < n; i++) // 边界条件L=1
        if (S[i] == S[i - 1])
            f[i - 1][i] = 0; // 如果是 aa形式则恰为回文串
        else
            f[i - 1][i] = 1;    // 如果是ab 形式则需要一次调整
    for (int L = 2; L < n; L++) // 动态规划主过程,从L=2开始
        for (int i = 0; i + L < n; i++) {
            int j = i + L;
            f[i][j] = min(f[i + 1][j] + 1, f[i][j - 1] + 1); // 对应前两类情况
            if (S[i] == S[j])
                f[i][j] = min(f[i + 1][j - 1], f[i][j]); // 对应特殊情况
        }
    cout << f[0][n - 1] << endl; // 最终答案放在 f[0][n-1]
    return 0;
}
```

事实上，可以发现动态规划主过程对 $L=0$ 和 1 也适用，可以把前两个for循环删掉，把 $L$ 的起点改成 0。但是不建议在竞赛现场过早这么进行优化，除非很明确知道是在优化什么，并知道这么优化一定是对的。

当然本题的拓扑排序不唯一，读者也可以通过研究题目的结构来实现其他拓扑排序作为练习。

### 例 15-2 石子合并弱化版(洛谷P1775)

设有 $N(N\leqslant 300)$ 堆石子排成一排，其编号为 $1$，$2$，$3$，…， $N$。每堆石子有一定的质量 $m_i(m_i\leqslant 1000)$。现在要将这 $N$ 堆石子合并成为一堆。每次只能合并相邻的两堆，合并的代价为这两堆石子的质量之和，合并后与这两堆石子相邻的石子将和新堆相邻。合并时由于选择的顺序不同，合并的总代价也不相同。试找出一种合理的方法，使总的代价最小，并输出最小代价。

例如，$4$ 堆石子分别是 $2$，$5$，$3$，$1$，将 $3$ 和 $1$ 合并变为 $4$，将 $2$ 和 $5$ 合并变为 $7$，将 $4$ 和 $7$ 合并变为 $11$，共计代价 $22$。

**分析：**本题有一个很明显的特征，即一堆石子只能和相邻的石子进行合并操作，这代表某堆石子只可能是一段区间内连续合并的结果。那么可以用一个区间状态 $f_{i,j}$ 来表示合并区间 $[i,j]$ 上所有石子所能产生的最小总代价。获得 $f_{i,j}$ 这堆石子的答案一定需要合并它的两个子区间所代表的石子 $f_{i,k}, f_{k+1,j}(i \leqslant k < j)$，而且这两堆石子的总质量一定是 $[i,j]$ 上所有石子的总质量（这甚至与 $k$ 的选取无关，记为 $w_{i,j}$），因此只需要最小化 $f_{i,k}+f_{k+1,j}$ 即可，从而也能推导出本题具备的最优子结构性质（请读者自证）。

于是可以写出状态转移方程：

$$f_{i,j} = \min_{i \leqslant k < j}(f_{i,k} + f_{k+1,j}) + w_{i,j}$$

如法炮制，也使用递增步长 $L$ 来确定**拓扑排序**，这几乎在所有区间**动态规划**实现中适用。至于边界条件，应该注意到 $f_{i,i}$ 应该为 0，这意味着直接从 $L=1$ 开始枚举即可，不用考虑边界特殊情况了。代码如下：

```
#include <iostream>
#define maxn 510
using namespace std;
int n, a[maxn], f[maxn][maxn], w[maxn][maxn];
// f[i][j]表示合并区间[i,j]上所有石子产生的最小总代价
int main() {
    cin >> n;
    for (int i = 1; i <= n; i++)
        cin >> a[i];
    for (int i = 1; i <= n; i++)
        for (int j = i; j <= n; j++)
            w[i][j] = w[i][j - 1] + a[j];
    for (int L = 1; L < n; L++)
        for (int i = 1; i + L <= n; i++) {
            int j = i + L, &fij = f[i][j];
            fij = 2000000000;
            for (int k = i; k < j; k++)
                fij = min(fij, f[i][k] + f[k + 1][j]);
            fij += w[i][j];
        }
    cout << f[1][n] << endl;
    return 0;
}
```

### 例 15-3 祖玛游戏(洛谷CF607B)

给定一个字符串 $S$（字符以数字的形式给出），字符串长度不超过 $500$。每一步可以选择一个回文子串并删除，此时两边留下的子串（如果有）会按原顺序拼起来。试计算至少要多少步才能把该串清空（指拿走最后一个回文子串后不剩下任何其他串）。

例如，当原序列 $1442321$，先删除 $232$，再删除 $1441$，此时字符串清空。

**分析：**参照前例，直接定义 $f_{i,j}$ 表示将子串 $S_{i,j}$ 清空所需要的最少步数。接下来思考怎么利用区间内的信息计算 $f_{i,j}$ 的答案。初见本题的时候也许会想到以下 3 种情况：

1. 整个串自身是个回文串，可以通过比较 $S_i$ 和 $S_j$ 和 $f_{i+1,j-1}$ 判断。

2. 和上题类似，可以把区间$[i,j]$分成区间$[i,k]$和 $[k+1,j]$，分别利用这两个区间的答案。

3. 一个可消除串夹在另一个可消除串里（见样例，内部串是 232，外部串是 1441）。

对于第 3 种情况，如果考虑枚举内部串的两个端点，不仅会超时，也不容易计算外部串的答案。那么重新考察回文串的性质。

直接研究样例，可以注意到：消除 44232 需要两次操作，消除 1442321 也只需要两次操作！这两个区间之间存在关联吗？

答案是肯定的：只要某个串的两端字符相同，那么这两个字符不会给消除这个串造成额外的负担（除非这个串长度为 2，即只有这两个字符，如aa）。证明也很简单，例如串是aSa，其中 $S$ 可以是一个比较复杂的串，只要一直对 $S$ 内部进行消除，直到只剩下最后一个回文串，那么多出来的两个 $a$ 自动和这个回文串组成了另一个回文串！

观察样例，对 44232 删除到只剩下 44，那么多出来的一对 1 和 44 自动组成了 1441，这对 1 不会给消除整个串造成负担（相应的，对 44232 删除到只剩下 232，就能组成 12321，也符合这个情况）。可以发现，这个思路和前面提到的第 2 个情况是一致的，没必要分开考虑（12321 的这对 1 不会给消除整个串带来额外负担）。

本题的边界情况与 例 15-1 极为类似，需要特殊处理长度为 1 的串和长度为 2 的回文串（消除它们的代价都是 1）。具体的实现细节可以参考如下代码：

```
#include <bits/stdc++.h>
#define maxn 510
using namespace std;
int n, a[maxn], f[maxn][maxn];
// f[i][j]表示i到j区间消除次数
int main() {
    scanf("%d", &n);
    for (int i = 0; i < n;)
        scanf("%d", a + ++i);
    for (int i = 1; i <= n; i++)
        for (int j = i; j <= n; j++)
            f[i][j] = maxn;
    for (int L = 0; L < n; L++)
        for (int i = 1, j = i + L; i <= n - L; i++, j++) {
            if (a[i] == a[j])
                f[i][j] = (L < 2) ? 1 : min(f[i][j], f[i + 1][j - 1]);
            // 如果长度超过2,则没有额外负担,否则需要1次操作来消除
            for (int k = i; k < j; k++)
                f[i][j] = min(f[i][j], f[i][k] + f[k + 1][j]);
            // 通过划分区间来取最小的答案
        }
    printf("%d", f[1][n]);
}
```

接下会来考查略有难度的区间**动态规划**变形问题，作为区间动态规划的补充。读者也需要在了解经典模型的基础上具备一定应变能力，才能机敏地解决很多困难问题。

### 例 15-4 合唱队形(洛谷P3205,湖南省队选拔2010)

为了在即将到来的晚会上有更好的演出效果，作为AAA合唱队负责人的小 $A$ 需要将合唱队的人根据他们的身高排出一个队形。假定合唱队一共 $n$ 个人，第 $i$ 个人的身高为 $h_i$ 毫米 $(1000 \leqslant h_i \leqslant 2000)$，并已知任何两个人的身高都不同。假定最终排出的队形是 $A$ 个人站成一排，为了简化问题，小 $A$ 想出了如下排队的方式：他让所有的人先按任意顺序站成一个初始队形，然后从左到右按以下原则依次将每个人插入最终排出的队形中：

1. 第 $1$ 个人直接插入空的当前队形中。
2. 对从第 $2$ 个人开始的每个人，如果他比前面那个人高（$h$ 较大），那么将他插入当前队形的最右边。如果他比前面那个人矮（$h$ 较小），那么将他插入当前队形的最左边。

当 $n$ 个人全部插入当前队形后便获得最终排出的队形。小 $A$ 心中有一个理想队形，他想知道多少种初始队形可以获得理想的队形。请求出答案对 $19650827$ 取模的值。

**分析：**可以发现在理想队列的生成过程中，每进来一个人，则队列就会向一边扩展。因此，对于理想队形中的一个区间，可能是先在这个区间内生成的队列，然后在区间外继续生成。这问题带有“大区间包含小区间”的性质，所以可以使用区间**动态规划**。

考虑用 $f_{i,j}$ 表示在理想队列中，区间 $[i,j]$ 的方案数。但考虑到每一种状态和前一种状态有关（和前一个人的高度比较），发现还不够。可以增加一维状态，用 $f_{i,j,0}$ 表示这个区间中刚刚加入的人是从左边进来的，用 $f_{i,j,1}$ 表示从右边进来。

对于前者，新加入的人比前一个人矮，前一个人可能在 $i+1$ 号位置，也有可能在 $j$ 号位置；对于后者，新加人的人比前一个人高，前一个人可能在 $j-1$ 号位置，也有可能在 $i$ 号位置。

考虑边界条件，显然当 $i=j$ 时有一种情况，所以 $f_{i,i,0}=f_{i,i,1}=1$。但是，对于区间 $[i,j]$ 的方案和应当为 $f_{i,j,0}+f_{i,j,1}$，这里发生了重复计数。因此当区间长度是 1 的时候，默认从左边（右边也可以）加入队列，只须选择一边进行初始化。代码如下：

```
#include <bits/stdc++.h>
#define maxn 1010
#define mod 19650827
using namespace std;
int f[maxn][maxn][2], a[maxn];
int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++)
        cin >> a[i];
    for (int i = 1; i <= n; i++)
        f[i][i][0] = 1;
    for (int len = 1; len <= n; len++)
        for (int i = 1, j = i + len; j <= n; i++, j++) {
            if (a[i] < a[i + 1])
                f[i][j][0] += f[i + 1][j][0];
```

```
            if (a[i] < a[j])
                f[i][j][0] += f[i + 1][j][1];
            if (a[j] > a[i])
                f[i][j][1] += f[i][j - 1][0];
            if (a[j] > a[j - 1])
                f[i][j][1] += f[i][j - 1][1];
            f[i][j][0] %= mod;
            f[i][j][1] %= mod;
        }
    cout << (f[1][n][0] + f[1][n][1]) % mod;
    return 0;
}
```

像这样以区间左右端点来描述状态，一个状态由比它更小的其他状态转移而来的动态规划方法被称为区间动态规划。转移时，考虑区间边界上的变化。边界条件一般是最小单位的区间（即 $i=i$ 的情况下）。当不便很好地构建转移顺序时，可以考虑使用**记忆化搜索**实现。

## 15.2 环形动态规划

### 例 15-5 石子合并(洛谷P1880,NOI1995)

在一个圆形操场的四周摆放 $N$ 堆石子，现要将石子有次序地合并成一堆。规定每次只能选相邻的 $2$ 堆合并成新的一堆，并将新的一堆的石子数，记为该次合并的得分。试设计出一个算法，计算出将 $N$ 堆石子合并成一堆的最小得分和最大得分。

**数据范围：** $1 \leqslant N \leqslant 100, 0 \leqslant a_i \leqslant 20$

例如，$4$ 堆石子分别是 $4$、$5$、$9$、$4$，最小和最大得分分别是 $43$ 和 $54$。

**分析：**这是一道环形**动态规划**的经典题。在 例 15-2 中已经介绍了在不是“环形”的情况下要怎么完成本题。在加上环形的限制条件下，其实只需要做小小的修改就行。可以证明一定存在一个“断点”在某两个相邻的石子中间，对于任意环上的合并方式，都不会跨越这个断点进行合并。因此，可以将这个环破解成一个链来处理。

如果每次从环中生成一个新的链，分别进行动态规划，则时间复杂度无法接受。可以将这个序列复制一次，变为长度为 2N的序列。其中每一个长度为 $N$ 的连续子序列都是剪开的一条链。剩下的事情就可以和 例 15-4 一样进行动态规划了。这么做的好处是同时处理多个不同的链的过程，可以共同引用较小区间的数据以节约时间。

代码实现时，长度还是从 1 枚举到 $N$，但原始链的长度扩大一倍。这里还可以同时处理最大值和最小值。别忘了最后还需要再次统计，断点在什么地方时可以得到最大或者最小的答案。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
int n, x, y, ans1 = 2000000000, ans2 = -ans1;
#define maxn 210
int a[maxn][maxn];
int f1[maxn][maxn], f2[maxn][maxn];
int main() {
    cin >> n;
```

```
    for (int i = 1; i <= n; i++)
        cin >> a[i][i];
    for (int i = 1; i <= n; i++) // 单个区间
        a[n + i][n + i] = a[i][i];
    for (int i = 2; i <= n; i++) {
        for (int j = 1; j <= n + n - i + 1; j++) {
            x = j, y = i + j - 1;
            for (int k = 1; k < i; k++) {
                a[x][y] = a[x][x + k - 1] + a[x + k][y];
                if (f1[x][y] != 0)
                    f1[x][y] = min(f1[x][y], a[x][y] + f1[x][x + k - 1] + f1[x + k][y]);
                else
                    f1[x][y] = a[x][y] + f1[x][x + k - 1] + f1[x + k][y];
                if (f2[x][y] != 0)
                    f2[x][y] = max(f2[x][y], a[x][y] + f2[x][x + k - 1] + f2[x + k][y]);
                else
                    f2[x][y] = a[x][y] + f2[x][x + k - 1] + f2[x + k][y];
            }
        }
    }
    for (int i = 1; i <= n; i++)
        ans1 = min(ans1, f1[i][n + i - 1]);
    for (int i = 1; i <= n; i++)
        ans2 = max(ans2, f2[i][n + i - 1]);
    cout << ans1 << endl << ans2 << endl;
    return 0;
}
```

### 例 15-6 环上最大子段和

有个环状数组，每个位置上有个数字（可正可负）。试最大化环上连续一段数字的和（可以是空段，输出 $0$）。序列长度 $n \leqslant 200000$。

**分析：**对环上问题熟悉的读者可能知道环的常见处理方式是复制一倍，再当成区间问题解决。但复制一倍后无法利用 例 15-5 的思路转移，处理的虚拟子段经常会超过 $n$，这明显是非法情况。

现在考虑答案的情况：作为答案的子段要么经过 $1-n$ 的分界线，要么不经过 $1-n$ 的分界线，刚好是非环上序列的最大子段和。再考虑前者答案的性质：最大子段和经过 $1-n$ 的分界线，那么它在环上的补集恰是非环上序列的最小子段和（请读者自证）。

获得了这个巧妙的性质，便可以得知最终答案就是非环情况的最大子段和或者最小子段的补段和。代码如下：

```
#include <bits/stdc++.h>
#define maxn 100010
using namespace std;
long long n, a, w1, w2, sum, mi, ma;
// w1、w2表示正在处理的子段和,mi、ma表示全局最小、最大子段和
int main() {
    scanf("%lld ", &n);
    for (; n--;) {
        scanf("%lld", &a);
        sum += a;
        ma = max(ma, w1 = max(0LL, w1 + a)); // 最大子段和
        mi = min(mi, w2 = min(0LL, w2 + a)); // 最小子段和
    }
    printf("%lld", max(ma, sum - mi)); // 最大子段和与最小子段和的补
}
```

## 15.3 课后习题与实验

### 习题 1 相似基因(洛谷P1140)

给出两个仅有 $A$C$G$ 种字母组成的碱基序列，长度均不超过 $100$。碱基和碱基之间可以配对，产生一定的类似度；碱基也可以与空碱基配对。碱基和碱基（或者空碱基）之间的相似度见表 15-1。

| | A | C | G | T | 空 |
|---|---|---|---|---|---|
| A | 5 | -1 | -2 | -1 | -3 |
| C | -1 | 5 | -3 | -2 | -4 |
| G | -2 | -3 | 5 | -2 | -2 |
| T | -1 | -2 | -2 | 5 | -1 |
| 空 | -3 | -4 | -2 | -1 | 不适用 |

表15-1碱基和碱基之间的相似度

需要给这两个序列的碱基配对（可以在两个碱基中分别插入若干空碱基，然后分别匹配），求这两个碱基序列的相似度之和的最大值。例如给出两个碱基序列AGTGATG和GTTAG，进行这样的匹配：

```
AGTGATG
-GTTA-G
```

可以获得相似度最大值 $14$。

### 习题 2 涂色(洛谷P4170,重庆市队选拔赛2007)

有一条长度为 $5$ 的木板，初始时没有涂过任何颜色。给定一个目标（长度为 $5$，仅由大写字母组成），每次可以把一段连续的木板涂成一种颜色，后涂的颜色覆盖先涂的颜色。请用尽量少的涂色次数达到目标。

例如，希望把它别涂上红、绿、蓝、绿、红色，用一个长度为 5 的字符串表示这个目标：RGB-GR。第 1 次把木板涂成RRRRR，第 2 次涂成RGGGR，第 3 次涂成RGBGR，达到目标。

### 习题 3 玩具取名(洛谷P4290,河南省队选拔2008)

某人有一套玩具，并想给玩具命名。首先他选择 $W$I$N$ 个字母中的任意一个字母作为玩具的基本名字。然后会根据自己的喜好，将名字中任意一个字母用 $W$I$N$G$ 中任意两个字母代替，使得自己的名字能够扩充得很长。输入一个长度不超过 200 的字符串，表示这个玩具的名字。输出若干大写字母，标明该名字可能由哪些字母变形而得到。

### 习题 4 248(洛谷P3146,USACO2016年公开赛)

给定一个 $1 \times n$ 的地图，在里面玩游戏，每次可以合并相邻且相同两个数字（数值范围 1 到 40），合并后这两个数字就会合并为一个数字，其值增加 1。问序列中出现的最大数字的值最大是多少。

### 习题 5 项链(洛谷P1063,NOIP2006提高组)

在项链上有 $N(4 \leqslant N \leqslant 100)$ 颗能量珠。能量珠有头和尾两个标记，对应着两个不超过 1000 的正整数。对于相邻的两颗珠子，前一颗珠子的尾标记一定等于后一颗珠子的头标记。两颗珠子可以聚合成一颗珠子，同时释放出可以被吸盘吸收的能量。如果前一颗能量珠的头标记为 $m$，尾标记为 $r$，后一颗能量珠的头标记为 $r$，尾标记为 $n$，则聚合后释放的能量为 $m \times r \times n$，新产生的珠子的头标记为 $m$ 尾标记为 $n$ 不同的聚合顺序得到的总能量是不同的，请设计一个聚合顺序，使一串项链聚合为一颗能量珠，释放出的总能量最大。

输出其最大值。

### 习题 6 道路游戏(洛谷P1070,NOIP 2009普及组)

有一个环，环上有 $n$ 个工厂，每个工厂可以花费一定金币购买金币收割机器人，每个机器人在购买后可以沿着环最多走 $p$ 条边，一秒走一条。不能同时出现超过一个机器人。每条边不同时间上出现的金币是不同的。问如何安排购买机器人，可以在 $m$ 的时间内，净赚最多的金币（中途可以透支）。

**数据范围：** $2 \leqslant n \leqslant 1000, 1 \leqslant m \leqslant 1000, 1 \leqslant p \leqslant m$。

### 习题 7 多边形(洛谷P4342,IOI1998)

给定一个环，有 $n(3 \leqslant n \leqslant 50)$ 个结点，以边连接。点上有一个数字，边上有一个符号（加号或者乘号）。这个游戏中，首先可以删掉一条边，然后选择一条边，将这条边上的两个结点上的数使用这个边的符号计算，结果变为一个结点。输出最高分数，以及写出所有可以选择清除后仍能得到最高得分的边。保证游戏过程中，所有结点在 $-32768$ 到 32767 之间。

### 习题 8 关路灯(洛谷P1220)

某一村庄在一条路线上安装了 $n(n \leqslant 50)$ 盏路灯，每盏灯的功率（单位时间耗电量不超过 100）有大有小。老张就住在这条路中间某一盏路灯旁，他有一项工作就是每天早上天亮时一盏一盏地关掉这些路灯。老张关灯所用的时间很短而可以忽略不计。

老张走的速度为 1，在第 $c$ 个路灯位置。已知每个路灯的位置（整数，指距路线起点的距离）、功率。请安排关灯的顺序，使从老张开始关灯时刻算起所有灯消耗电最少（灯关掉后便不再消耗电了）。

# 第16章 树与图上的动态规划

树与图上的**动态规划**，顾名思义，就是以树或图为模型的动态规划。树上动态规划是最常见的动态规划形式之一，因为树形本身就带来了子结构（树上的父子关系），大部分情况下，都是通过子结点的DP值推导出父结点的DP值。而图上的动态规划，由于没有很明显的子结构，所以比较受限。有向无环图（DAG）由于点之间有着明显的分层，性质相较于图更加良好。许多图上的问题可以转化后成为树上的或图上的问题。剩下的许多问题模型，又可以转化为最短路问题。本章思维导图如图 16-1 所示。




图16-1 本章思维导图

## 16.1 树上动态规划

### 例 16-1 舞会(洛谷P1352)

给定一棵有 $n(n\leqslant 6\times 10^3)$ 个结点的树，每个点有权值（$-128$ 到 $127$），需要选出一些点，使得它们的权值和最大，并且不存在两个点使得它们是父子关系。

**分析：**思考如何用子结点的状态推导出父结点的状态。设 $f_u$ 表示 $u$ 号点所在的子树内最大的权值和，（$u$，$v$）为父子关系，可以写出以下式子：

$$f_u=\sum_{v\in\pi} f_v$$

但这样无法保证满足题目限制，所以需要设立更多的状态。题目中限制了父子关系，而上述转移是子结点向父结点转移。两者相一致，只需要记录自身的相关信息即可，这样转移的时候，既拥有父结点的信息，也拥有子结点的信息。

设 $f_{u,0}$ 表示 $u$ 号点没有被选时，$u$ 所在子树内最大的权值和 $\cdot f_{u,1}$ 表示 $u$ 号点被选时，$u$ 所在子树内最大的权值和。这样，再来看转移：

$$f_{u,0}=\sum_{v\text{是}u\text{的子结点}}\max(f_{v,0},f_{v,1}),$$
$$f_{u,1}=a_u+\sum_{v\text{是}u\text{的子结点}}f_{v,0}.$$

也就是说，4 种转移中删去了一种转移 $:f_{v,1}\longrightarrow f_{u,1}$，这表示两个为父子关系的点同时被选了，不符合题目的要求。理解了本题，就能够更加深刻地体会到所谓“状态代表着信息”的含义了。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int Maxn = 6005;
vector<int> G[Maxn];
int n, a[Maxn], f[Maxn][2];
void dfs(int u, int fa) {
    f[u][0] = 0, f[u][1] = a[u];
    for (auto v : G[u])
        if (v != fa) {
            dfs(v, u);
            f[u][0] += max(f[v][0], f[v][1]);
            f[u][1] += f[v][0];
        }
}
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
        scanf("%d", &a[i]);
    for (int i = 1; i < n; i++) {
        int x, y;
        scanf("%d%d", &x, &y);
        G[x].push_back(y), G[y].push_back(x);
    }
    dfs(1, 0);
    printf("%d", max(f[1][0], f[1][1]));
    return 0;
}
```

代码中 `for(autov: G[u])` 是指变量 $v$ 遍历 `G[u]` 的所有元素，是 C++11 标准开始支持的写法。在只支持 C++98 标准的情况下，也可以直接使用迭代器，或者当作普通数组遍历。

树上**动态规划**一般在树的 DFS 中来完成。因为 DFS 能够展现出父结点走向子结点，子结点回溯到父结点的过程。

### 例 16-2 二叉苹果树(洛谷 P2015)

给定一个 $n$ 个结点的二叉树，满足每个结点要么是叶子，要么有两个子结点。每条边上有若干苹果，要求砍掉一些子树（即取出一个包含根结点的连通块），求刚好剩下 $Q$ 条边时，这些边苹果数之和最大是多少。

**数据范围：** $1\leqslant Q\leqslant n\leqslant 100$，每条边的苹果数 $\leqslant 3\times 10^4$。

**分析：**剩下的边连接的点一定是连通的。从根开始考虑，枚举其左子树和右子树分别选了多少条边，因为左、右子树互相不干扰，所以是单独的子问题。

所以以子树中选了多少条边为状态 $\cdot f_{u,v}$ 表示以 $u$ 为根的子树，恰好选了 $v$ 条边，苹果数最大是多少。

转移要考虑从子树合并上来。对于 $u$ 为根的子树，分别记左、右两个子结点为 $l$，$r$，如果 $u$ 到 $l,r$，左子树的边不选，那么左子树的边数和苹果数贡献只能是 0，否则对于所有 $f$ 都能选。右子结 $f_{l,k}$ 点同理。因此有 3 种转移：

$$
\begin{aligned}
f_{u,x+1} &\longleftarrow \max\{f_{u,x+1}, f_{l,x}\} \\
f_{u,y+1} &\longleftarrow \max\{f_{u,y+1}, f_{r,y}\} \\
f_{u,x+y+2} &\longleftarrow \max\{f_{u,x+y+2}, f_{l,x} + f_{r,y}\}
\end{aligned}
$$

这样做其实就是在二叉树上完成了一个背包。这印证了树形DP也可以承载很多种类的信息。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 110;
void gma(int &x, int y) { x < y ? x = y : 0; }
int dp[MAXN][MAXN];
int son[MAXN][2], siz[MAXN];
int val[MAXN][2];
int n, Q;
void dfs(int u) {
    int x = son[u][0], y = son[u][1];
    if (!x && !y)
        return;
    dfs(x);
    dfs(y);
    siz[u] = siz[x] + siz[y] + 2;
    for (int i = -1; i <= siz[x]; ++i)
        for (int j = -1; j <= siz[y]; ++j) {
            // i:-1 代表着不选边,否则代表子树中选择i条边
            // j: 和i类似
            // 因为不选刚好贡献0条边,选i条贡献i +1条边
            // 所以用-1代表不选十分合适
            int vl = i == -1 ? 0 : dp[x][i] + val[u][0];
            int vr = j == -1 ? 0 : dp[y][j] + val[u][1];
            gma(dp[u][i + j + 2], vl + vr);
        }
}
int main() {
    scanf("%d%d", &n, &Q);
    for (int i = 1; i < n; ++i) {
        int x;
        scanf("%d ", &x);
        int b = son[x][0] > 0;
        scanf("%d%d", &son[x][b], &val[x][b]);
    }
    dfs(1);
    printf("%d\n", dp[1][Q]);
    return 0;
}
```

### 例 16-3 选课(洛谷P2014,CTSC 1997)

给定 $n$ 门课程来选，每门课程 $i$ 有依赖的课程 $k_i$，即选了 $k_i$ 才能选 $i$，如果 $k_i = 0$ 则表示没有依赖。每门课程有一个权值 $s_i$，选出 $m$ 门课，要求最大化被选的课程权值和。

数据范围 $:n,m\leqslant 300,s_i\leqslant 20$

**分析：**将 $k$：向 $i$ 连一条有向边，得到的图由一棵根为 0 的外向树和若干环构成。环是没有 $k_i$ 意义的，于是问题被转化成在树上选择若干个结点，最大化权值。因此可以考虑树形DP。考虑 $\mathbf{DP}_c$ 子树作为子结构的时候，无论怎么选课，只有“子树中一共选了多少课程”这一条件能够影响其到父结点的转移。

根据先前的经验，很容易写出DP状态 $\cdot f_{u,w}$ 表示根为 $u$ 的子树中，选择 $w$ 门课程，权值和最大是多少。

在转移到父结点的时候，对于每个子树枚举其选择了多少课程。直接枚举的话复杂度是错误的，简单分析发现这是一个分组背包模型（即物品有若干类，同一类里面恰好只能选一个物品的背包模型）。

在 $u$ 为根的子树中，先假设 $u$ 不选。用 $g_{v,w}$ 来辅助背包的转移，表示当前考虑到子树 $v,-$ 共选了 $w$ 门课程，权值和最大是多少。在枚举到下一个子树 $v'$ 的时候有转移：

$$g_{v',w+w'}=\max_{0\leqslant w+w'\leqslant m}\left\{g_{v,w}+f_{v',w'}\right\}$$

然后再考虑 $u$ 怎么选的情况。当子树中选了 0 门课程，$u$ 可以选，也可以不选。一旦子树中选择了大于 0 个，$u$ 就不得不选了。记 $v$ 为最后一个枚举到的子树，有转移：

$$f_{u,0}=g_{v,0}$$
$$f_{u,w}=g_{v,w-1}+s_u(w>0)$$

至此完成了一个较为复杂的树上 $\mathbf{DP}$。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 310;
int f[MAXN][MAXN], n, m;
int req[MAXN], scr[MAXN];
int head[MAXN], nxt[MAXN];
void adde(int b, int e) {
    nxt[e] = head[b];
    head[b] = e;
}
void dfs(int u) {
    memset(f[u] + 1, 0xc0, m << 2);
    for (int v = head[u]; v; v = nxt[v]) {
        dfs(v);
        static int g[MAXN]; // 使用滚动数组
        memset(g, 0xc0, (m + 1) << 2);
        for (int j = 0; j <= m; ++j)
            for (int k = 0; k + j <= m; ++k) {
                // 对子树进行分组背包
                g[j + k] = max(g[j + k], f[u][j] + f[v][k]);
            }
        memcpy(f[u], g, (m + 1) << 2);
    }
    if (u == 0)
```

```
        return;
    // 考虑根的选与不选
    for (int i = m; i; --i)
        f[u][i] = f[u][i - 1] + scr[u];
}
int main() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; ++i) {
        scanf("%d%d", &req[i], &scr[i]);
        adde(req[i], i);
    }
    dfs(0);
    printf("%d\n", f[0][m]);
    return 0;
}
```

## 16.2 图上动态规划

本节先介绍路径模型的**动态规划**。在此类模型中，状态和路径有关，例如说两点之间可达性/最短路，从 $u$ 出发走 $k$ 步到 $v$ 的方案数等。由于路径的分层不是特别明显，一般以路径长度或者点的顺序，或者题目的特殊条件来构造DP状态。

### 例 16-4 跑步(洛谷P1613)

有一幅 $n$ 个点 $m$ 条边的有向图，每条边长度均为 $1$。小 $A$ 在一个空间跑路器的帮助下，每秒钟可以跑 $2$ 千米（$k$ 是任意自然数）。求小 $A$ 从点 $1$ 到点 $n$ 最少需要几秒。 $2^k$

**数据范围：** $\cdot n \leqslant 50, m \leqslant 10000$，数据保证 $1$ 到 $n$ 至少有一条路径且最优解路径长度不超过 $2^{31}-1$

**分析：**由于空间跑路器的存在，直接求出的**最短路径**不一定是最优解。如果点 $u$ 到点 $v$ 有长度为 2（$i$ 是整数）的路径，则只需要花费 1 秒就可以从点 $u$ 移动到点 $v$。因此需要对原图做出 $2^i(iv$ 一定的改变，再求解最短路。

设最优解路径的长度为 $s$，可以得到最优解消耗的时间是在二进制下 $s$ 中 1 的个数。例如当 $s=(101100)_2$ 时，消耗的时间为 3 秒。也等价于将这条路径拆分成若干长度为 $2^k(k \in lN)$ 的路径，每条路径消耗的时间都是一秒。因此只需要关注两点之间是否存在长度为 2 的路径。 $2^k$

从而可以构建一张图，如果在原图中 $u$ 到 $v$ 存在一条长度为 2 的路径，则在新图中连一条 $2^k$ $u$ 到 $v$ 边权为 1 的边，代表可以 1 秒从 $u$ 跑到 $v$。对于原图中的任意一条路径，经过拆分之后都 $v$。会等价于新图上的一条路径。新图上该路径的长度即为在原图上跑完该路径所花的最短时间。

设 $f[i,u,v]$ 表示是否存在从 $u$ 到 $v$、长度为 $2^i$ 的路径。特别地，$f[0,u,v]$ 表示是否存在长度为 $1$ 的路径，也就是原图中的边。长度为 $2^i$ 的路径可以拆成两条长度为 $2^{i-1}$ 的路径，因此枚举中转点 $t$：若存在 $t$ 使 $f[i-1,u,t]=1$ 且 $f[i-1,t,v]=1$，则 $f[i,u,v]=1$；否则为 $0$。

两条长度为 $2^i$ 的路径可以连接拼成一条长度为 $2^{i+1}$ 的路径。反过来，任意一条长度为 $2^{i+1}$ 的路径，都可以由两条连在一起的长度为 2 的路径组成。所以只要用长度为 2 的路径就可以 $2^i2^i$ 转移得到长度为 $2+1$ 的路径，而不需要借助其他长度的路径。`f[i,u,v]` 表示点 $u$ 到点 $v$ 是否存 $2^{i+1}$ 在长度为 $2^i$ 的路径。可以得到转移（$V$ 表示逻辑或，$\wedge$表示逻辑与）：

$$f_{i,u,v}=\bigvee_{w}(f_{i-1,u,w} \wedge f_{i-1,w,v})$$

得到数组 $f$ 后，只须依照 $f$ 重新建图，并在新图上求最短路即可得到最后的答案。完整代码如下：

```
#include <bits/stdc++.h>
using namespace std;
int n, m, dis[55][55];
bool G[55][55], f[35][55][55];
void dp() {
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            if (G[i][j])
                f[0][i][j] = true;
    for (int i = 1; i <= 30; i++)
        for (int k = 1; k <= n; k++)
            for (int u = 1; u <= n; u++)
                for (int v = 1; v <= n; v++)
                    f[i][u][v] |= f[i - 1][u][k] & f[i - 1][k][v];
}
int main() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= m; i++) {
        int x, y;
        scanf("%d%d", &x, &y);
        G[x][y] = true; // G为邻接矩阵,G[i][j]表示是否有从点u到点v的边
    }
    dp();                          // 求出数组f
    for (int i = 1; i <= n; i++) // 初始化
        for (int j = 1; j <= n; j++)
            if (i != j)
                dis[i][j] = 1000000000;
    for (int i = 0; i <= 30; i++) // 建新图
        for (int u = 1; u <= n; u++)
            for (int v = 1; v <= n; v++)
                if (f[i][u][v])
                    dis[u][v] = min(dis[u][v], 1);
    for (int k = 1; k <= n; k++) // Floyd求最终答案
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                if (k != i && k != j && i != j)
                    dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
    printf("%d\n", dis[1][n]);
    return 0;
}
```

### 例 16-5 美食家(洛谷P6772,NOI2020)

某国家共有 $n$ 座城市. 城市从 $1$ 到 $n$ 编号，其中城市 $i$ 的美食能为小 $W$ 提供 $c_i$ 的愉悦值。各城市通过 $m$ 条单向道路连接，道路从 $1$ 到 $m$ 编号，其中道路 $i$ 的起点为城市 $u$，终点为城市 $u_iv_i$，沿它通行需要花费 $w$；天。也就是说，若小 $W$ 在第 $d$ 天从城市 $u$；沿道路 $i$ 通行，那么他会在 $w_iu_i$ 第 $d+w_i$ 天到达城市 $v_i$。

小 $W$ 计划在该国进行一场为期 $T$ 天的旅行。他会在第 $0$ 天从城市 $1$ 出发，经过 $T$ 天的旅行，最终在恰好第 $T$ 天回到城市 $1$ 结束旅行。小 $W$ 每到达一座城市时（包括第 $0$ 天和第 $T$ 天的城市 $1$ ），都会品尝该城市的美食并获得其所提供的愉悦值，若小 $W$ 多次到达同一座城市，他将获得多次愉悦值。旅行途中小 $W$ 不能在任何城市停留，即当他到达一座城市且还未结束旅行时，他当天必须立即从该城市出发前往其他城市。

此外，该国会举办 $k$ 次美食节。第 $i$ 次美食节在第 $t_i$ 天、城市 $x_i$ 举办。如果小 $W$ 当天恰好在城市 $x_i$，品尝美食时就会额外获得 $y_i$ 的愉悦值。求整个旅行中能获得的愉悦值之和的最大值。

**数据范围**：$1 \le n \le 50$，$n \le m \le 501$，$0 \le k \le 200$；$1 \le t_i \le T \le 10^9$，$1 \le w_i \le 5$，$1 \le c_i \le 52501$；$1 \le u_i, v_i, x_i \le n$，$1 \le y_i \le 10^9$。

**分析**：注意到边权很小，如果将边拆成若干个点，那么花费 $w$；天相当于经过wi个点。如果 $w_i w_i$ 不考虑美食节，那么可以以经过的点数为状态：定义 $f$，表示经过 $t$ 个点，当前在点 $v$ 的最大愉 $f_{t,v}$ 悦值。

取出图上所有的边，很容易写出转移：

$$f_{t+1,v} = \max_{(u,v)\in E} \{f_{t,u} + c_v\}$$

在美食节的时候，给对应的 $f_{t_i,x_i}$ 加上 $y_i$ 即可。

有一个问题就是 $T$ 特别的大，直接暴力转移是会超时的。此时可以使用矩阵来加速转移。（min，+）的矩阵满足结合律，对于边 $(u,v)$，其对应矩阵 $A$ 上 $a_{v,u} = c_{v0}$ 对于初始值是向量 $a$，其中 $a_1 = c_1$，其他的 $a_i$ 则赋上一个足够大的值。这样对 DP 的转移就可以使用 $A^k a$ 来表示。

通过预处理 $A^{2^{\frac{k}{2}}}$，可以在 $O(n^3 \log V)$ 的时间复杂度里获得任意的 $A^V$，其中 $n$ 是矩阵大小，在这题中为 $O(n \times \max w_i)$。这样的技巧叫矩阵快速幂优化**动态规划**。

但是注意到预处理就需要 $O(n^3 \log T)$ 的复杂度，而美食节最多又有 200 次，这样也会超时。原因是矩阵乘法是 $O(n^3)$ 的，所以需要采用更快的矩阵乘向量，时间复杂度为 $O(n^2)$。

这样就可以用 $O\left(n^2 \log V\right)$ 时间复杂度，让动态规划转移 $V$ 次。最终的时间复杂度为 $O\left(n^2(n+k)\log T\right)$。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
typedef long long LL;
const int MAXN = 260;
LL A[31][MAXN][MAXN];
int n, m, T, K, val[MAXN];
struct Celebration {
    int t, u, v;
} ts[MAXN];
bool cmp(Celebration a, Celebration b) { return a.t < b.t; }
inline int get(int t, int x) { return t * n + x - 1; }
void gma(LL &x, LL y) { x < y ? x = y : 0; }
int main() {
    scanf("%d%d%d%d", &n, &m, &T, &K);
    for (int i = 1; i <= n; ++i)
        scanf("%d", &val[i]);
    memset(A, 0xcf, sizeof A);
    for (int i = 1; i <= m; ++i) {
        int b, e, w;
        scanf("%d%d%d", &b, &e, &w);
        A[0][get(4, e)][get(5 - w, b)] = val[e];
    }
    for (int i = 0; i < 4; ++i)
        for (int j = 1; j <= n; ++j)
```

```
            A[0][get(i, j)][get(i + 1, j)] = 0;
    const int lim = n * 5;
    // 预处理矩阵幂次
    for (int i = 0; i != 30; ++i) {
        for (int j = 0; j < lim; ++j)
            for (int k = 0; k < lim; ++k)
                for (int l = 0; l < lim; ++l)
                    gma(A[i + 1][j][l], A[i][j][k] + A[i][k][l]);
    }
    static LL dp[2][MAXN];
    memset(dp, 0xcf, sizeof dp);
    int now = 0, lst = 1;
    dp[now][get(4, 1)] = val[1];
    auto trans = [&](int dt) {
        // 对 dt 进行二进制拆分来转移
        for (int j = 0; j != 31; ++j)
            if (dt >> j & 1) {
                swap(now, lst);
                memset(dp[now], 0xcf, sizeof dp[now]);
                for (int k = 0; k < lim; ++k)
                    for (int l = 0; l < lim; ++l)
                        gma(dp[now][k], dp[lst][l] + A[j][k][l]);
            }
    };
    for (int i = 1; i <= K; ++i)
        scanf("%d%d%d", &ts[i].t, &ts[i].u, &ts[i].v);
    sort(ts + 1, ts + 1 + K, cmp);
    for (int i = 1; i <= K; ++i) {
        int dt = ts[i].t - ts[i - 1].t;
        trans(dt);
        dp[now][get(4, ts[i].u)] += ts[i].v;
    }
    trans(T - ts[K].t);
    LL ans = dp[now][get(4, 1)];
    if (ans < 0)
        ans = -1;
    printf("%lld\n", ans);
    return 0;
}
```

再来讲述有向无环图的动态规划，在这一类问题中，由于**拓扑排序**导致拓扑排序小的点只依赖拓扑排序大的点（反过来类似），将拓扑排序作为构造动态规划状态的依据十分合适。有些题目经过转化，也能够将一般图上的问题转化为有向无环图上的问题。

### 例 16-6 绿豆蛙的归宿(洛谷P4316)

给出一幅 $n$ 个点 $m$ 条边的有向无环图，起点为 $1$，终点为 $n$，每条边都有一个长度，并且从起点出发能够到达所有的点，所有的点也都能够到达终点。

绿豆蛙从起点出发，走向终点。到达每一个顶点时，如果该结点有 $k$ 条出边，绿豆蛙可以选择任意一条边离开该点，并且走向每条边的概率为 $\frac{1}{k}$。现在绿豆蛙想知道，从起点走到终点的所经过的路径总长度期望是多少？

保证 $1 \leqslant n \leqslant 10^5, 1 \leqslant m \leqslant 2 \times n, 1 \leqslant u, v \leqslant n, 1 \leqslant w \leqslant 10^9$，给出的图无重边和自环。

**分析**：由于是有向无环图，**拓扑排序**使得**动态规划**的顺序十分明显。记 $d_u$ 为 $u$ 的出边数量，$w$ 为边的长度 $f_u$ 为从 $u$ 走到终点时路径的期望长度，那么有转移：

$$f_u = \sum_{(u,v)\in E} \frac{f_v + w_{(u,v)}}{d_u}$$

拓扑排序小的依赖拓扑排序大的，因此按拓扑排序从大到小进行转移即可。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 100010;
int head[MAXN], nxt[MAXN << 1], to[MAXN << 1], val[MAXN << 1], tot;
int out[MAXN], deg[MAXN], n, m, t1, t2, t3;
double f[MAXN];
void addedge(int b, int e, int v) {
    nxt[++tot] = head[b];
    head[b] = tot;
    to[tot] = e;
    val[tot] = v;
}
int main() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= m; ++i) {
        scanf("%d%d%d", &t1, &t2, &t3);
        // 反向建图以获得从大到小的拓扑序
        addedge(t2, t1, t3);
        ++out[t1];
        ++deg[t1];
    }
    queue<int> q;
    f[n] = 0;
    q.push(n);
    while (!q.empty()) {
        int t = q.front();
        q.pop();
        for (int i = head[t]; i; i = nxt[i]) {
            // 计算v到u的贡献
            f[to[i]] += (f[t] + val[i]) / out[to[i]];
            --deg[to[i]];
            if (!deg[to[i]])
                q.push(to[i]);
        }
    }
    printf("%.2lf\n", f[1]);
    return 0;
}
```

### 例 16-7 采蘑菇(洛谷P2656)

森林间有 $N$ 个小树从. $M$ 条小径，每条小径都是单向的，连接两个小树丛，上面都有一定数量的蘑菇。经过某条小径一次，可以采走这条路上所有的蘑菇。一条路上的蘑菇被采完后，又会长出一些新的蘑菇，数量为原来蘑菇的数量乘上这条路的“恢复系数”，再向下取整。例如，一条路上有 $4$ 个蘑菇，这条路的“恢复系数”为 $0.7$，则第 $1$ 次到第 $4$ 次经过这条路径所能采到的蘑菇数量分别为 $4$，$2$，$1$，$0$。从 $S$ 号小树从出发，请问最多能采到多少蘑菇。

```
// 保证1≤N≤8×104,1≤M≤2×105,0≤恢复系数≤0.8且仅有一位小数，$1 \leqslant S \leqslant N _ { \mathrm { { c } } }$
```

**分析：**称一条不断走过，最终蘑菇数为 0 的边为被走完。很容易发现如果两条边在同一个**强连通分量**，那么它们会同时被走完或者同时不被走。所以可以通过强连通分量算法，将图变成一个边有边权 $w$，点有点权 $p$ 的有向无环图。其中路径的权值定义为边权和点权之和。找到 $S$ 所在的强连通分量后，问题被转化为在有向无环图上，从一个点出发的最长路径。

对于每个点 $u$ 记 $f_u$ 为从 $u$ 出发的最长路径，可以写出转移：

$$f_u = \max_{(u,v)\in E} \left(w_{(u,v)} + f_v\right) + p_u$$

对于每个点，其 $f$ 值只依赖**拓扑排序**比它大的点，所以要按拓扑排序从大到小进行**动态规划**。使用拓扑排序即可完成。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 8e4 + 10;
const int MAXM = 4e5 + 10;
int head[MAXN], nxt[MAXM], to[MAXM], val[MAXM], coe[MAXM], tot;
void adde(int b, int e, int v, int c) {
    nxt[++tot] = head[b];
    to[head[b] = tot] = e;
    val[tot] = v;
    coe[tot] = c;
}
int dfn[MAXN], low[MAXN], st[MAXN], ins[MAXN], bel[MAXN], top, idx;
void tarjan(int u) {
    static int t0t;
    dfn[u] = low[u] = ++t0t;
    ins[st[++top] = u] = true;
    for (int i = head[u]; i; i = nxt[i])
        if (!dfn[to[i]]) {
            tarjan(to[i]);
            low[u] = std::min(low[u], low[to[i]]);
        } else if (ins[to[i]])
            low[u] = std::min(low[u], dfn[to[i]]);
    if (low[u] == dfn[u]) {
        int v;
        ++idx;
        do {
            ins[v = st[top--]] = false;
            bel[v] = idx;
        } while (v != u);
    }
}
int vt[MAXN], ind[MAXN], dp[MAXN];
```

```
int n, m;
int main() {
    scanf("%d%d ", &n, &m);
    for (int i = 1, u, v, w; i <= m; ++i) {
        double _c;
        scanf("%d%d%d%lf", &u, &v, &w, &_c);
        int c = std::round(_c * 10);
        adde(u, v, w, c);
    }
    for (int i = 1; i <= n; ++i)
        if (!dfn[i])
            tarjan(i);
    static int th[MAXN];
    memcpy(th, head, (n + 1) << 2);
    memset(head, 0, (n + 1) << 2);
    for (int u = 1; u <= n; ++u)
        for (int i = th[u]; i; i = nxt[i]) {
            int v = to[i];
            if (bel[u] == bel[v]) {
                int t = val[i];
                while (t) { // 处理一条边被不断走
                    vt[bel[u]] += t;
                    t = t * coe[i] / 10;
                }
            } else { // 将边反向以获得从大到小的拓扑排序
                adde(bel[v], bel[u], val[i], 0);
                ++ind[bel[u]];
            }
        }
    std::queue<int> que;
    for (int i = 1; i <= idx; ++i)
        if (!ind[i])
            que.push(i);
    while (!que.empty()) {
        int t = que.front();
        que.pop();
        dp[t] += vt[t];
        for (int i = head[t]; i; i = nxt[i]) {
            dp[to[i]] = std::max(dp[to[i]], dp[t] + val[i]);
            if (!--ind[to[i]])
                que.push(to[i]);
        }
    }
    int S;
    scanf("%d", &S);
    printf("%d\n ", dp[bel[S]]);
    return 0;
}
```

注意本题的 $a \times b$，在计算机中容易引起精度误差，所以通过计算 $\left\lfloor \frac{a \times (10 \times b)}{10} \right\rfloor$，使得所有计算在整数下进行，能够得到精确的答案。

## 16.3 课后习题与实验

### 习题 1 加分二叉树(洛谷P1040,NOIP2003)

设一个 $n$ 个结点的二叉树的中序遍历为 $(1,2,3,\cdots,n)$，其中数字 $1,2,3,\cdots,n$ 为结点编号。每个结点都有一个分数（均为正整数），记第 $i$ 个结点的分数为 $d$，二叉树及它的每个子树都有 $d_i$ 一个加分，任一棵子树的子树（也包含它本身）的加分计算方法如下：这棵树的左子树的加分×这棵树的右子树的加分+这棵树的根的分数。

若某个子树为空，规定其加分为 $1$，叶子的加分就是叶结点本身的分数。不考虑它的空子树。求一棵符合中序遍历为 $(1,2,3,\cdots,n)$ 且加分最高的二叉树。要求输出最高加分以及其前序遍历。

保证 $1\leqslant n<30$，结点的分数是小于 $100$ 的正整数，答案不超过 $4\times 10^9$。

### 习题 2 最大子树和(洛谷P1122)

给一棵 $n(n\leqslant 16000)$ 个点的树，每个点有点权 $w_i$，点权可能为负。要求求出树上点权和最大的连通块，输出点权和。保证答案绝对值 $<2^{31}$

### 习题 3 战略游戏(洛谷P2016)

玩家要建立一个古城堡，城堡中的路形成一棵无根树，有 $n(n\leqslant 1500)$ 个结点。他要在这棵树的结点上放置最少数目的士兵，使得这些士兵能瞭望到所有的路。当某个士兵在一个结点上时，与该结点相连的所有边将都可以被瞭望到。请问至少需要放置几个士兵。

### 习题 4 三色二叉树(洛谷P2585,浙江省队选拔2006)

一棵二叉树可以按以下规则表示为由 $0$、$1$、$2$ 组成的字符串，称为“二叉树序列” $S$：

1. 若没有子结点，则 $S=0$。
2. 若有一个子结点，则 $S=1S_1$，其中 $S_1$ 是该子树的序列。
3. 若有两个子结点，则 $S=2S_1S_2$，其中 $S_1$、$S_2$ 分别是左右子树的序列。

任务是要对一棵二叉树的结点进行染色。每个结点可以被染成红色、绿色或蓝色。并且，一个结点与其子结点的颜色必须不同，如果该结点有两个子结点，那么这两个子结点的颜色也必须不同。给定一棵二叉树的二叉树序列，请求出这棵树中最多和最少有多少个点能够被染成绿色。

保证 $1\leqslant |S|\leqslant 5\times 10^5$。

### 习题 5 有线电视网(洛谷P1273)

某有线电视网计划转播一场比赛。其转播网和用户终端构成树结构，这棵树的根结点位于足球比赛的现场，树叶为各个用户终端，其他中转站为该树的内部结点。从一个转播站到另一个转播站以及从一个转播站到某个用户终端的信号传输费用都是已知的，一场转播的总费用等于传输信号的费用总和。

现在每个用户都准备了一笔费用想观看这场比赛，有线电视网有权决定给哪些用户提供信号而不给哪些用户提供信号。找出一个方案使得有线电视网在不亏本的情况下使观看转播的用户尽可能多，输出最大用户数。

数据范围 $\cdot 2 \leqslant N \leqslant 3000, 1 \leqslant M \leqslant N-1$，$N$ 为整个有线电视网的结点总数，$M$ 为用户终端的数量。单次传输成本和用户愿意交的费用均不超过 $10$。

### 习题 6 软件安装(洛谷P2515,河南省队选拔2010)

有 $N$ 个软件，对于一个软件 $i$，它要占用 $\mathbb{W}_i$ 的磁盘空间，它的价值为 $V_i$。 希望从中选择一些软件安装到一台磁盘容量为 $M$ 计算机上，使得这些软件的价值尽可能大（即 $V_i$ 的和最大）。但软件之间存在依赖关系，即软件 $i$ 只有在安装了软件 $j$（包括软件 $j$ 的直接或间接依赖）的情况下才能正确工作，一个软件最多依赖另外一个软件。如果一个软件不能正常工作，那么它能够发挥的作用为 $0$。

现在知道了软件之间的依赖关系：软件 $i$ 依赖软件 $D_i$。现在请设计出一种方案，安装价值尽量大的软件。一个软件只能被安装一次，如果一个软件没有依赖则 $D_i=0$，这时只要这个软件安装了，它就能正常工作。

保证 $0 \leqslant N \leqslant 100, 0 \leqslant M \leqslant 500, 0 \leqslant W_i \leqslant M, 0 \leqslant V_i \leqslant 1000, 0 \leqslant D_i \leqslant N, D_i \leqslant i_0.$

### 习题 7 奶牛大集会(洛谷P2986,USACO2010年3月赛)

来自整个农场的奶牛将举办一次集会。农场是 $N$ 个牛棚组成的树形结构，每个牛棚中居住着 $C_i$ 只奶牛。道路 $i$ 连接牛棚 $A_i$ 和 $B_i$，长度为 $L_i$。 集会可以在 $N$ 个牛棚中的任意一个举行。例如选择第 $X$ 个农场作为集会地点，总代价是其他牛棚中每只奶牛去参加集会所走的路程之和。即农场 $i$ 到达农场 $X$ 的距离是 $20$，那么其代价就是 $C_i \times 20$ 请找出最方便的地点来举行大集会。

**数据范围：** $1 \leqslant N \leqslant 10^5, 0 \leqslant C_i, L_i \leqslant 10^3$。

### 习题 8 逛公园(洛谷P3953,NOIP2017)

公园是一张 $N$ 个点 $M$ 条边构成的有向图，没有自环和重边。其中 $1$ 号点是公园的入口，$N$ 号点是公园的出口，每条边有一个非负权值，代表经过这条边所要花的时间。策策每天从 $1$ 号点进去，从 $N$ 号点出来。他不希望有两天逛公园的路线完全一样，也不希望每天在逛公园这件事上花费太多的时间。如果 $1$ 号点到 $N$ 号点的最短路长为 $d_i$ 那么策策只会喜欢长度不超过 $d+K$ 的路线。策策同学想知道总共有多少条满足条件的路线。为避免输出过大，答案对 $P$ 取模。如果有无穷多条合法的路线，请输出 $-1$。

保证 $1 \leqslant P \leqslant 10^9, 1 \leqslant a_i, b_i \leqslant N, 0 \leqslant c_i \leqslant 1000$，至少存在一条合法的路线。

### 习题 9 函数调用(洛谷P7077,CSP-S2020)

某数据库应用程序提供了若干函数用以维护数据。已知这些函数的功能可分为以下 $3$ 类：

1. 将数据中的指定元素加上一个值。
2. 将数据中的每一个元素乘以一个相同值。
3. 依次执行若干次函数调用，保证不会出现递归（即不会直接或间接地调用本身）。

在使用该数据库应用时，用户可一次性输入要调用的函数序列（一个函数可能被调用多次），在依次执行完序列中的函数后，系统中的数据被加以更新，需要计算出更新后的数据应该是多少。答案对 $998244353$ 取模。

保证元素个数、函数个数、函数序列长度都 $\leqslant 10^5$，被调用的函数个数之和 $\leqslant 10^6$

### 习题 10 图函数(洛谷P7516,联合省队选拔2021)

对于一张 $n$ 个点 $m$ 条边的有向图 $G$（顶点从 $1\sim n$ 编号），定义函数 $f(u,G)$

1. 初始化返回值 $cnt=0$，图 $G'=G$。
2. 从 $1$ 至 $n$ 按顺序枚举顶点 $v$，如果当前的图 $G'$ 中，从 $u$ 到 v 与从 $v$ 到 $u$ 的路径都存在，则将 $cnt+1$，并在图 $G'$ 中删去顶点 $v$ 以及与它相关的边。
3. 第 $2$ 步结束后，返回值 `cnt` 即为函数值。

现在给定一张有向图 $G$，求出 $h(G)=f(1,G)+f(2,G)+\cdots+f(n,G)$ 的值。

更进一步地，记删除（按输入顺序给出的）第 $1$ 到 $i$ 条边后的图为 $G_i(1\leqslant i\leqslant m)$，求出所有 $h(G_i)$ 的值。

保证 $2\leqslant n\leqslant 10^3,1\leqslant m\leqslant 2\times 10^5,1\leqslant x_i,y_i\leqslant n$。

# 第17章 状态压缩动态规划

在《基础篇》的“暴力枚举”章中介绍了子集枚举，使用二进制数来表示一种状态。对于“**状态压缩动态规划**”中的状态，通常与集合相关联。集合本身具有确定性、互异性和无序性 3 个性质。这也就决定了集合只关心每个元素的存在状态，而这通常可以使用 0 或者 1 表示存在或者不存在。例如，一顿饭有 8 道菜，小明同学最终吃了某几道菜，必然是这 8 道菜的子集。令 1 表示吃了.0 表示没吃，那么以下几个状态：10000001 表示只吃了 1 号菜和 8 号菜；01010101 表示吃了 2，4，6，8 四道菜。

由此可见，可以通过一串 0、1 码来清晰地表示一个集合的状态。同时，在确定了最低位的前提下，一串 0、1 码与一个二进制数一一对应。这种表示状态的方式被称为状态压缩，简称状压。所谓“压缩”，即是用一个二进制数来保存一组状态，将一个集合的状态压缩进了一个二进制数中，而通常这个二进制数在十进制下的大小可以作为其编号。状态压缩本质上是进行了两步操作：

1. 给这个集合的每个状态一个编号。
2. 通过该编号，轻易地访问该状态。

本章将会从状态压缩谈起，描述一些简单的应用，并最终落脚到状态压缩应用最广泛的动态规划题目。本章思维导图如图 17-1 所示。




图17-1 本章思维导图

## 17.1 状态压缩

### 例 17-1 国际象棋(洛谷P1896,四川省队选拔2005)

在 $n \times n(1 \leqslant n \leqslant 9)$ 的棋盘里面放 $k(1 \leqslant k \leqslant n^2)$ 个国王，使其互不攻击，共有多少种摆放方案？国王能攻击到其上、下、左、右，以及左上、左下、右上、右下 8 个方向上附近的各一个格子，共 8 个格子。

**分析：**简单模拟后不难发现，答案可能会很大。而对于较复杂的计数，**动态规划**的思想往往是解决此类问题的有力武器。通过观察性质，不难发现按一定顺序，例如行号递增的方式摆放棋子时，当前棋子能不能放到某个位置上完全取决于上一行的棋子是怎么摆放的。因此，设计状态时可以将当前行、上一行的状态都记录下来，使用 0 或者 1 来表示某一行的状态，考虑用二进制数来表示：

$$100010_{(2)} = 34_{(10)}$$

那么，对于这一行的状态，就可以用 34 这个十进制数值来记录。此时需要注意，一般情况下状压之后，状态中的每一位都是倒着存储的，因为这符合二进制的定义规则。例如一行的状态为 110101，那么二进制下应该是 101011。可以类比，从左至右列标逐渐增大，但是二进制位从左到右权值逐渐减小。

这种访问方式可以实现快速而准确的状态访问。例如，`(1 << (k - 1)) & s` 用于判断状态 `s` 从最低位起的第 $k$ 位是否为 1；`(j >> k) << k` 可以将非负整数 `j` 的最低 $k$ 位清零，而逐项存储的数组不能在 $O(1)$ 时间内完成这样的批量清零。

对于本题，结合状压原理，考虑用 $f_{i,j,k}$ 表示前 $i$ 行，第 $i$ 行状态的二进制表示（十进制下的值）为 $j$，放置了 $k$ 个国王的方案数，那么状态转移方程就是：

$$f_{i,j,k} = \sum_{\text{new 与 } j \text{ 相容}} f_{i-1,\text{new},k-\text{getlen}(j)}$$

其中new表示所枚举的上一行的所有合法状态，`getlen(j)` 表示 $j$ 状态有多少位为 1，因为所定义的 1 即为放了国王。最终复杂度是 $O(4^n \times nk)$。其中有许多需要统计的数据是可以通过预处理来进行效率优化的，具体留作思考。代码如下：

```
#include <bits/stdc++.h>
#define ll long long
using namespace std;
ll dp[20][210][1026], ans;
int N, K, NN, i, j, k, l, L;
inline int _get(ll x) {
    int ret = 0;
    while (x) {
        if (x & 1)
            ret++;
        x >>= 1;
    }
    return ret;
}
inline bool check(ll x, ll y) { // 判断是否合法
    if (x & y)
```

```
        return 1;
    if ((x << 1) & y)
        return 1;
    if ((x >> 1) & y)
        return 1;
    if ((y >> 1) & y)
        return 1;
    return 0;
}
int main() {
    cin >> N >> K;
    dp[0][0][0] = 1, NN = (1 << N) - 1;
    for (i = 1; i <= N; i++)            // 前 i行
        for (j = 0; j <= K; j++)        // 放了 j 个
            for (k = 0; k <= NN; k++) { // 当前行的状态
                L = _get(k);
                if (L > j)
                    continue;
                if (k & (k >> 1))
                    continue;
                for (l = 0; l <= NN; l++) {
                    if (check(k, l))
                        continue;
                    dp[i][j][k] += dp[i - 1][j - L][l];
                }
            }
    for (i = 0; i <= NN; i++) {
        ans += dp[N][K][i];
    }
    cout << ans;
}
```

### 例 17-2 售货员的难题(洛谷P1171)

有 $n$ 个村庄，一名售货员要从起点（$1$ 号村庄）开始，遍历每个村庄售货，并最后返回起点。各村庄之间的道路是单向的，第 $i$ 个村庄到第 $j$ 个村庄的路程是 $u_{i,j}$， 。求最短遍历路线的长度。

数据范围 $:2\leqslant n\leqslant 20,1\leqslant u_{i,j}\leqslant 1000$。

分析 • 本题是著名的旅行商问题（Traveling SalesmanProblem，TSP），是一个非常重要的NP完全（NP-Complete）问题。NP完全问题是一类特殊的问题，这类问题目前还没找到多项式时间复杂度解法，其常见复杂度表示中往往包含 $n$!或 2”（其中的 $n$ 为输入数据的规模）等非多项式 $2^n($ 因子，所以这类问题的解法通常是暴力搜索或**状态压缩动态规划**。

本题需要记录“现在走到了哪个村庄”。但“走过了 1、2、3，来到 4，还需要走到 5”与“走过了 1，来到 4，还需要走到 2、3、5”虽然都在 4 号村庄，却是不同状态。因此，进一步增加状态 $s$：$f_{i,s}$ 表示当前走到 $i$ 号村庄，且已访问的村庄集合为 $s$ 时的最短路程。$s$ 可以表示为长度为 $n$ 的二进制串，其中 $s_i=0$ 表示第 $i$ 个村庄未被访问，$s_i=1$ 表示已经访问。初始状态为：

$$f_{1,\{1\}}=0,\qquad f_{x,S}=+\infty\quad((x,S)\neq(1,\{1\}))$$

其中，$\{x,y,z,\cdots\}$ 表示一个包含元素 $x,y,z,\cdots$ 的集合，因此{1}表示一个仅包含元素 1 的集合。最终结果即为走到某个结点且走完了全部状态的 $f$ 的最小值（因为需要从 $i$ 再走回到 1，所以需要加上 `u(i,1)`），也就是：

$$\min_{1\le i\le n}\{w(i,1)+f_{i,\{1,2,\ldots,n\}}\}$$

基于以上状态定义，可以写出转移方程：

$$f_{i,S}=\min_{j\in S\setminus\{i\}}\{f_{j,S\setminus\{i\}}+w(j,i)\}$$

转移的意义是，当需要在集合中新加入一个点 $i$，且“已经过点的集合”为 $s$ 时，点 $i$ 是从某个点 $i$ 走过来的，枚举 $j$，取路程的最小值即可，总的时间复杂度是 $O(n^2 2^n)$。代码如下：

```
#include <bits/stdc++.h>
#define Inf 1000000000
#define MAXN 1100010
// 状态的大小不能小于2^20
using namespace std;
int W[21][21], Ans, S[21], t;
int M, N, H[MAXN], Len[MAXN], F[MAXN][21];
int main() {
    cin >> N;
    for (int i = 1; i <= N; i++)
        for (int j = 1; j <= N; j++)
            cin >> W[i][j];
    Ans = Inf;
    M = (1 << N) - 1; // 所有状态的总数
    memset(F, 0x3f, sizeof(F));
    F[1][1] = 0;
    for (int s = 1; s <= M; s++) { // 枚举所有的状态
        t = 0;
        memset(S, -1, sizeof(S));
        for (int i = 0; i < N; i++)
            if (1 << i & s)
                S[++t] = i + 1; // 把状态对应的集合对象放入数组 S 中
        for (int i = 1; i <= t; i++)
            for (int j = 1; j <= t; j++) {
                if (i == j)
                    continue;
                F[s][S[i]] = min(F[s][S[i]], // 枚举所有集合内除了 S[i]的元素
                                 F[s ^ (1 << (S[i] - 1))][S[j]] + W[S[j]][S[i]]);
            }
    }
    for (int i = 1; i <= N; i++) // 最后回到起点
        Ans = min(Ans, F[M][i] + W[i][1]);
    cout << Ans << endl;
}
```

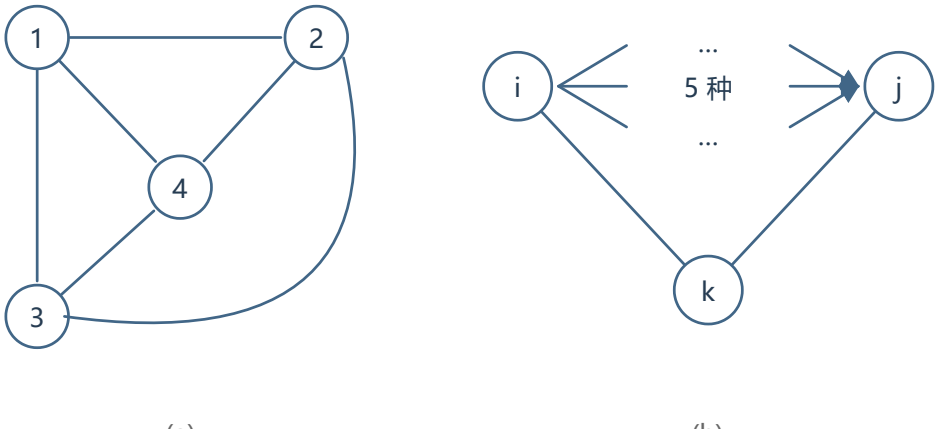
## 17.2 状态压缩的一般技巧

以上就是两种常见的**状态压缩**模型。接下来讨论状态压缩问题中常见的技巧。

### 例 17-3 简单的任务(洛谷CF11D)

给定一个没有重边自环的无向图，点数不超过 **19**，求无向图中环的数量。

**分析：**观察样例对应的无向图，如图 17-2(a) 所示：




(a) (b)

图17-2 无向图的环

通过观察数据范围可知，或许对点数进行状压是一个思路。先考虑如何统计环的数量。假设点 $i$ 到点 $j$ 有 5 条简单路径，点 $i$ 和点 $k$ 相连，点 $k$ 也和点 $j$ 相连，如图 17-2(b) 所示。可以知道，必然存在 5 个包含 $i$，$j$，$k$ 这 3 个点的环。因此一个想法是，考虑把统计环的个数转化为统计简单路径的条数。

设 $f_{i,j,s}$ 表示集合 $s$ 中从 $i$ 到 $j$ 的简单路径条数。值得注意的是，在无向图里，$i \to j \to k$、$j \to k \to i$ 和 $k \to i \to j$ 这 3 种递推关系本质上是在描述同一个环。观察并计算运算量，$19^2 \times 2^{19} = 189267968$，无法使用数组存下所有的状态，因此考虑一个在**动态规划**中比较常见的技巧：按秩转移。

此处可以将“秩”理解为“顺序”。为了使同一个集合内的点是按顺序转移的，考虑使用 `lowbit(s)` 来记录答案。整个算法流程可以理解为对于每个状态，枚举其中的元素 $i$，同时枚举另一个元素 $j$。若 $j$ 在 $s$ 里，按照上文分析的只有 `lowbit(s)`$=j$ 时要计入答案；如果 $j$ 不在 $s$ 里，就可以将方案数从 $s$ 转移到 $s \cup \{j\}$ 这个集合。

于是最后时间复杂度为 $O(n^2 \cdot 2^n)$，空间复杂度 `O(n·2")`。其中由于并不会跑满，所以可以较为轻松地通过。代码如下：

```
#include <bits/stdc++.h>
#define MAXN 20
#define MAXS 600010
#define LL long long
#define low(x) (x & (-x))
using namespace std;
LL ans;
int N, M, Mx, A[MAXN][MAXN];
LL f[MAXS][MAXN];
int main() {
    int i, j, s, u, v, now;
    cin >> N >> M, Mx = (1 << N) - 1;
    for (i = 1, j = 0; j < N; ++j, i <<= 1)
        f[i][j] = 1; // 初始化
    for (i = 1; i <= M; ++i)
        cin >> u >> v, u--, v--, A[u][v] = A[v][u] = 1;
```

```
    // 此处要注意二进制下最右侧是第0位,所以编号要从0开始
    for (s = 1; s <= Mx; ++s) {
        for (i = 0; i < N; ++i) {
            if (!((1 << i) & s) || !f[s][i])
                continue;
            for (j = 0; j < N; ++j) {
                if (!A[i][j] || low(s) > (1 << j))
                    continue;
                if ((1 << j) == low(s))
                    ans += f[s][i];
                else if (!(1 << j & s))
                    f[s | (1 << j)][j] += f[s][i];
            }
        }
    }
    cout << ((ans - M) >> 1) << endl;
}
```

### 例 17-4 宝藏(洛谷P3959,NOIP2017提高组)

给出一个有 $n$ 个宝藏屋的图，有 $m$ 条未开凿道路连接这些宝藏屋且长度已知。小明可以任选一个起点开始挖宝。他可以在已经开凿的道路上行走，自由访问起点和已经开凿道路连接的宝藏屋；他也可以选择开凿新的道路，需要保证已经开凿的道路和连接的宝藏屋是一棵树。小明想让所有的宝藏屋均可自由访问。

开凿某条道路时，要求道路的其中一头的宝藏屋是可以自由访问的。开凿新的一条道路需要花费代价 $L\times K$。其中 $L$ 代表这条道路的长度，$K$ 代表待打通的宝藏屋到起点所经过的道路数量。请问工程总代价最小值是多少。

**数据范围：** $1\leqslant n\leqslant 12, 0\leqslant m\leqslant 10^3, v\leqslant 5\times 10^5$，

**分析：**观察数据范围，$n=12$，且 $n$ 为集合大小。本题在 第4章“进阶搜索”中被当做习题，但这里考虑状压。一个比较简单的思路是把树高，即道路的长度加到状态里。那么就可以有 $f_{s,h}$ 表示现在选的点集为 $s$，树高为 $h$ 的最小花费。不难得到转移方程：

$$f_{s,i}=\min_{t\subsetneq s}\{f_{t,i-1}+\mathrm{cost}_{t\to s}\}$$

$cost_{ts}$ 所表示的花费应当在可接受的复杂度以内预处理出来，而使用Floyd算法求解**最短路径**时的时间复杂度是 $O(n^3)$，在这道题目中也是完全可以接受的。因此最终的复杂度就是 $O(3^n\cdot n^2)$。这里分析复杂度时需要用到一个求解“子集的子集数目”的技巧。

对于一个集合，一共有 $\Sigma\binom{n}{k}2^k$ 个子集的子集。考虑二项式定理进行变形：

$$\sum_{k=0}^{n}\binom{n}{k}2^k=\sum_{k=0}^{n}\binom{n}{k}1^{n-k}2^k=(1+2)^n=3^n$$

所以一共有 $3^n$ 个。实现也较为简洁：

**结构示意（含省略部分）：**

```
for (t=s; t; t= (t-1) &s)
```

其中 $t$ 就会遍历 $s$ 的全部子集。此时运算量上限约为 $531441 \times 12^2 = 76527504$，对于时限来说有点勉强。考虑优化，不难发现同一树高的所有结点可以同时转移。不妨设 $f_{s,t}$ 表示已选点集为 $s$，下一层要加入的点集为 $t$ 时，新加入的所有点与原有点之间最小的边权之和。转移如下：

$$f_{s,t} = f_{s,t\setminus\{k\}} + \text{cost}_{k,s}, \qquad k = \log_2(\text{lowbit}(t))$$

其中 $\text{cost}_{k,s}$ 表示点 $k$ 到已选点集 $s$ 的最小直接连边费用。利用 $\text{lowbit}(t)$ 逐点预处理，可得将新一层点集 $t$ 接到 $s$ 上的费用之和 $f_{s,t}$。令 $g_{s,h}$ 表示已选点集为 $s$、当前树高为 $h$ 的最小花费，则转移为：

$$g_{s,h} = \min_{\varnothing \neq t \subseteq s} \{g_{s\setminus t,h-1} + h \cdot f_{s\setminus t,t}\}$$

因此最后的时间复杂度就是 $O(3^n \times n)$。而最终答案应当是对所有 $g_{2^n-1,h}, h \in [1,n]$ 中取最小值。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 14, MAX = (1 << 12) + 5, Inf = 10000000;
int cnt, T, D[MAX][MAX], a, b, c;
int F[MAXN][MAX], Next[MAX], Ans, Sup, Now;
int N, M, i, j, k, Log[MAX], Max, A[MAXN][MAXN];
int main() {
    memset(A, 63, sizeof(A));
    memset(F, 63, sizeof(F)), Ans = Inf;
    cin >> N >> M;
    Max = (1 << N) - 1;
    for (i = 1; i <= M; ++i)
        scanf("%d%d%d", &a, &b, &c), A[a][b] = A[b][a] = min(c, A[a][b]); // 注意处理重边
    for (i = 0; i <= N; ++i)
        Log[1 << i] = i;
    for (i = 0; i < N; ++i)
        F[0][1 << i] = 0;
    for (i = 1; i <= Max; ++i) {
        cnt = 0;
        for (j = Sup = Max ^ i; j; j = (j - 1) & Sup)
            Next[j] = cnt, cnt = j;
        for (j = cnt; j; j = Next[j]) {
            Now = Log[j & (-j)] + 1, T = Inf;
            for (k = 1; k <= N; ++k)
                if (1 << (k - 1) & i)
                    T = min(T, A[Now][k]);
            D[i][j] = D[i][j ^ (j & -j)] + T;
        }
    } // 上面的循环实质上是预处理了距离,跟普通的最短路有着相似的精神
    for (i = 1; i < N; ++i)
        for (j = 1; j <= Max; ++j)
            for (k = j; k; k = (k - 1) & j)
                F[i][j] = min(F[i][j], F[i - 1][j ^ k] + i * D[j ^ k][k]);
    for (i = 0; i <= N; ++i)
        Ans = min(Ans, F[i][Max]);
    cout << Ans << endl;
    return 0;
}
```

### 例 17-5 最长上升子序列（洛谷P4484）

现在有一个长度为 $n$ 的随机排列，求它的最长上升子序列长度的期望。为了避免精度误差，只需要输出答案模 $998244353$ 的余数。

**数据范围：**本题数据范围为 $1 \leqslant n \leqslant 24$ （原题中数据范围为 $1 \leqslant n \leqslant 28$）

例如，当 $n=2$ 时有两种排列，$12$ 和 $21$。其中第 $1$ 种情况的LIS长度为 $2$，第 $2$ 种情况的LIS长度为 $1$。所以最终期望为 $\frac{1}{2} \times 2 + \frac{1}{2} \times 1 = \frac{3}{2}$。在模 $998244353$ 意义下可以写作 $499122178$，这将在 第19章“进阶数论”中有详细介绍。

**分析：**首先尝试得到部分分数，使用STL里的std：：next_permutation枚举全排列，然后对于每个排列用nlogn求一遍长度，最终复杂度会维持在 $O(n! \times n\log n)$ 的水平。对于 $n \leqslant 9$ 的数据，在常数较小的情况下是完全可以通过的。

对于更大的数据范围，考虑尝试使用**动态规划**来解决。比较简单的思路是考虑从“空”开始构造一个序列。但事实上从左向右构建和从右向左构建，均存在一定的问题，如在如何记录更新元素时，整个序列LIS长度的更新。

不妨考虑一种更加方便的状态设计。对于一个排列，把数从小到大插入到一个空的数列里面，即增量转移。首先令 $f$ 表示在当前已经确定的一个序列里面，以从左至右第 $i$ 个数为结尾的 $f_i$ 最长上升子序列长度。基于此，令 $\max(L_i)$ 表示前缀最大值，即：

$$\max(L_i)=\max(f_1,f_2,\cdots,f_i)$$

而对于 $\{\max(L_i)$ 这个数列，有一个显然的性质：

$$\max(L_i) \leq \max(L_{i+1}) \leq \max(L_i)+1$$

基于此可以发现 $\{\max(L_i)$ 有差分性。不难发现，差分之后得到的序列必然只会由 0、1 组成。不妨设该差分序列为 $\{dif_n\}$。

按数值从小到大插入新元素时，新插入的数是当前最大值。若插入位置在第 $i$ 位之后，它对应的前缀 LIS 最大长度为 $\max(L_i)+1$，所以新增的差分位为 1；之后遇到的第一个原本为 1 的差分位改为 0。其余差分位不变。

对差分序列进行**状态压缩**。令 $g_{i,j}$ 表示在 $1 \sim i$ 的排列中，差分状态为 $j$ 的方案数；省略首位恒为 1 的差分位后，LIS 长度为 $\text{popcount}(j)+1$。因此期望为：

$$\text{ans} = \frac{1}{n!} \sum_{i=0}^{2^{n-1}-1} g_{n,i}\big(\text{popcount}(i)+1\big)$$

也就是：

也就是将各个状态的方案数与其对应的 LIS 长度相乘后求和，再除以排列总数 $n!$。

值得一提的是，由于状压了数组dif，所以每个方案中LIS的长度，就是该状态里 1 的个数。 $dif$ 而 $g$ 的转移较为清晰：

$$g_{i,j} = \sum g_{i-1,k}$$

其中 $k$ 和 $i$ 为两个状态，之间的关系如上文所述。代码如下：

```
#include <bits/stdc++.h>
#define ll long long
using namespace std;
const ll mod = 998244353LL;
vector<ll> dp[2];
vector<unsigned char> getlen;
ll Mx, N, i, j, k, ans, fac = 1, now, t, pos;
inline ll lowbit(ll x) { return x & (-x); }
inline ll my_pow(ll a, ll b) {
    ll res = 1;
    while (b) {
        if (b & 1)
            res = (res * a) % mod;
        a = (a * a) % mod;
        b >>= 1;
    }
    return res;
}
int main() {
    cin >> N;
    --N;
    Mx = 1LL << N;
    dp[0].resize(Mx);
    dp[1].resize(Mx);
    getlen.resize(Mx);
    dp[0][0] = 1;
    for (now = i = 1; i <= N; now ^= 1, i++) {
        fill(dp[now].begin(), dp[now].begin() + (1 << i), 0);
        for (j = 0; j < (1 << (i - 1)); j++) {
            dp[now][j << 1] = (dp[now][j << 1] + dp[now ^ 1][j]) % mod;
            pos = -1;
            for (k = i - 1; ~k; k--) {
                t = ((j >> k) << (k + 1)) | (1 << k) | (j & ((1 << k) - 1));
                //(j>>k) << (k+1)是为了先清掉后面的几位所以不能简写成j<<1
                if (j & (1 << k))
                    pos = k;
                if (~pos)
                    t ^= (1 << (pos + 1)); // 取反的意义上文有提到
                dp[now][t] = (dp[now][t] + dp[now ^ 1][j]) % mod;
            }
        }
    }
    for (i = 1; i < Mx; i++)
        getlen[i] = getlen[i - lowbit(i)] + 1;
    for (i = 0; i < Mx; i++)
        ans = (ans + 1ll * dp[N & 1][i] * (getlen[i] + 1)) % mod;
    for (i = 1; i <= N + 1; i++)
        fac = (fac * i) % mod;
    ans = ans * my_pow(fac, mod - 2) % mod;
    cout << ans;
    return 0;
}
```

代码实现方面，有几个细节值得留意：

1. 实际上，在最大数据范围下的状态数达到了 $2^{27}$。不难发现，数组DP的第一维 $i$ 可以通过滚动数组的方式来节省空间。
2. 因为一定会有 $\max(L_1)=\max(L_0)+1=1$，所以初始化时可以少操作一次。
3. 为了便于递推，上述枚举的状态以及系列定义与数组的排列规则相同，跟二进制数的排列规则相反。
4. 注意是取反号而不是取非号。因为在状压时，全 0 也是状态的一部分，所以在从后往前枚举（为了便于计算后面将要被置成的 0）时应该到-1 停止而不是到 0 停止。

原题中较大的数据范围可以基于此，通过打表的方式解决。当然，此题目另有一种使用“杨表”解决、效率更高的拓展解法，大家可以自行研究。

### 例 17-6 下棋(洛谷P4363,联合省选2019)

菲菲和牛牛在一块 $n$ 行 $m$ 列的棋盘上下棋，菲菲执黑棋先手，牛牛执白棋后手。棋局开始时，棋盘上没有任何棋子，两人轮流在格子上落子，直到填满棋盘时结束。

落子的规则是：一个格子可以落子当且仅当这个格子内没有棋子且这个格子的左侧及上方的所有格子内都有棋子。棋盘的每个格子上，都写有两个非负整数，从上到下第 $i$ 行中从左到右第 $j$ 列的格子上的两个整数记作 $a_{i,j}$ 和 $b_{i,j}$。在游戏结束后，菲菲和牛牛会分别计算自己的得分：菲菲的得分是所有有黑棋的格子上的 $a$ 之和，牛牛的得分是所有有白棋的格子上的 $a_{i,j}b_{i,j}$ 的和。

菲菲和牛牛都希望，自己的得分减去对方的得分得到的结果最大。现在他们想知道，在给定的棋盘上，如果双方都采用最优策略且知道对方会采用最优策略，那么，最终的结果如何？

**数据范围：** $1\leqslant n,m\leqslant 10, 0\leqslant a_{i,j},b_{i,j}\leqslant 10^5$。

**分析：**首先一个比较直接的做法是状压行和列，复杂度为 $O(2^{n\times m})$，可以通过规模较小的数据。

不难发现如此状压下，状态数过多。但是真的需要记录完整的 2”个局面吗？简单模拟可知，可能合法的状态数并不会很多。由于这个状态数比较难算，可以考虑暴力搜索出状态数，发现极限情况下只有不到 $2\times 10^5$ 种合法的局面。

于是考虑一个技巧：控制状态数。具体而言，由于并不知道具体有多少个合法的状态，可以考虑把合法的状态都搜出来，然后按照顺序把所有状态都存储到vector里。具体判断状态是否被访问过，可以通过一个map来记录。

值得注意的是，虽然初始状态和终止状态，也就是第一步和最后一步的状态（包括位置和先后手）都是固定的，但是不能顺推。原因也比较简单，题目要求选手都“绝顶聪明”。真正的“绝顶聪明”应该是通过观察将来局面的可能结果再去判断如何行动，即当前决策应该取决于未来而不是过去，所以考虑逆推。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N = 11;
const int Base = 233;
const int M = 1000010;
const int P = 998244353;
int tot, n, m;
```

```
int cnt[N], A[N][N], B[N][N], base[N];
map<ll, int> h;
vector<int> s[M];
inline ll get_s() {
    ll ret = 0;
    for (int i = 1; i <= n; ++i)
        ret = (ret * Base + cnt[i]) % P;
    return ret;
}
int mn, dep, Id[M], rev[M], val[M], f[2][M];
void dfs() { // 搜出全部的状态
    ++dep;
    if (!h[get_s()]) { // 记录是否记录过
        ++tot;
        mn = m + 1;
        h[get_s()] = tot;
        for (int i = 1; i <= n; ++i) {
            s[tot].push_back(cnt[i]);
            mn = min(mn, cnt[i]);
            val[tot] += cnt[i];
        }
        if (mn == m)
            return;
    } else
        return;
    for (int i = 1; i <= n; ++i) {
        if (cnt[i] >= m || cnt[i] >= cnt[i - 1])
            continue;
        cnt[i]++;
        dfs();
        cnt[i]--;
    }
}
int main() {
    cin >> n >> m;
    cnt[0] = m + 1;
    dfs();
    for (int i = 1; i <= tot; ++i)
        Id[i] = i;
    sort(Id + 1, Id + tot + 1, [](int x, int y) { return val[x] > val[y]; });
    for (int i = 1; i <= tot; ++i)
        rev[Id[i]] = i;
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= m; ++j)
            scanf("%d", &A[i][j]);
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= m; ++j)
            scanf("%d", &B[i][j]);
    memset(f[1], 63, sizeof(f[1]));
    memset(f[0], -63, sizeof(f[0]));
    if ((n * m) & 1)
        f[1][1] = 0;
    else
        f[0][1] = 0;                // 0是先手,1是后手
    for (int i = 1; i < tot; ++i) { // 按照编号从后到前进行 DP
        int x = Id[i], y = 0, z, t = 0, o;
```

```
        for (auto p = s[x].begin(); p != s[x].end(); ++p) {
            ++t;
            y = 0;
            int mv = m;
            for (auto q = s[x].begin(); q != s[x].end(); ++q) {
                int x = *q;
                cnt[++y] = x;
                if (p == q)
                    --cnt[y], o = cnt[y];
                if (mv < cnt[y])
                    goto cccc;
                mv = min(cnt[y], mv);
            }
            if (!(z = h[get_s()]))
                continue;
            if (f[0][i] < P)
                f[1][rev[z]] = min(f[1][rev[z]], f[0][i] - B[t][o + 1]);
            if (f[1][i] < P)
                f[0][rev[z]] = max(f[0][rev[z]], f[1][i] + A[t][o + 1]);
        cccc:
            continue;
        }
    }
    printf("%d\n", f[0][tot]);
    return 0;
}
```

这个代码同样使用了 C++14 标准的语法。当然这道题也可以使用轮廓线DP解决，读者可以自行查阅相关资料。

## 17.3 状态压缩与其他算法的交叉应用

### 例 17-7 花园(洛谷P1357)

小 $L$ 有一座环形花园，沿花园的顺时针方向，他把各个花圃编号为 $1\sim n$。花园 $1$ 和 $n$ 是相邻的。他的环形花园每天都会换一个新花样，但都不外乎一个规则：任意相邻 $m$ 个花圃中都只有不超过 $k$ 个 $C$ 形的花圃，其余花圃均为 $P$ 形的花圃。请帮小 $L$ 求出符合规则的花园种数对 $10^9+7$ 取模的结果。

例如，若 $n=10,m=5,k=3$，则CCPCPPPPCC 是一种不符合规则的花圃，CCPPPPCPCP是一种符合规则的花圃。

**数据范围：**对于 $100\%$ 的数据，保证 $2\leqslant n\leqslant 10^{15},2\leqslant m\leqslant \min(n,5),1\leqslant k<m$。

**分析：**发现一共只有两种方格，并且转移只跟 $m$ 有关，而 $2\leqslant m\leqslant 5$，不难想到可以状压。考虑 $g(s,t)$ 表示从状态 $s$ 转移到 $t$ 的方案数。其中转移指的是向右扩展一格。那么该转移可以如此预处理：

```
void calc(int used, int st) {
    ok[st] = 1;
    int stt = st >> 1;
    t.M[stt][st] = 1;
    if (used == K && !(st & 1))
        return;
    t.M[stt | o[M]][st] = 1;
}
void dfs(int now, int used, int st) {
    // 参数分别为用了几个花圃,几个C形花圃,结尾M个花圃的状态如何
    if (now == M + 1) {
        calc(used, st);
        return;
    }
    dfs(now + 1, used, st);
    if (used < K)
        dfs(now + 1, used + 1, st | o[now]);
}
```

同时考虑 $2^5$非常小，且从状态 $A$ 转移到 $C$，和从 $A$ 先转移到 $B$，再从 $B$ 转移到 $C$ 的过程是完 $2^5$ 全等价的。因此这是一个满足结合律的转移。不难联想到同样具有结合律的矩阵乘法，所以可以通过矩阵快速幂①加速转移。

> **注：** 1. 关于矩阵快速幂，可以阅读 第22章“基础线性代数”。

**代码片段（需结合上文的声明和函数）：**

```
#include <bits/stdc++.h>
#define MAXS 72
#define MAXN 200010
#define Mod 1000000007
using namespace std;
long long N, res;
bool ok[MAXS];
int M, L, K, o[MAXS], dp[MAXN][MAXS];
struct Matrix { // 矩阵
    long long M[MAXS][MAXS];
    void clear() { memset(M, 0, sizeof(M)); }
    void reset() {
        clear();
        for (int i = 0; i <= L; ++i)
            M[i][i] = 1;
    }
    Matrix friend operator*(const Matrix &A, const Matrix &B) {
        Matrix Ans;
        Ans.clear();
        for (int i = 0; i <= L; ++i)
            for (int j = 0; j <= L; ++j)
                for (int k = 0; k <= L; ++k)
                    Ans.M[i][j] = (Ans.M[i][j] + A.M[i][k] * B.M[k][j]) % Mod;
        return Ans;
```

```
    }
} t, ans;
Matrix expow(Matrix T, long long P) { // 矩阵快速幂
    Matrix Ans;
    Ans.reset();
    while (P) {
        if (P & 1)
            Ans = Ans * T;
        T = T * T, P >>= 1;
    }
    return Ans;
}
// 省略 calc 和 dfs 函数
int main() {
    cin >> N >> M >> K;
    int i;
    o[1] = 1;
    L = (1 << M) - 1;
    for (i = 2; i <= M; ++i)
        o[i] = o[i - 1] << 1;
    dfs(1, 0, 0);
    ans = expow(t, N);
    for (i = 0; i <= L; ++i)
        if (ok[i])
            (res += ans.M[i][i]) %= Mod;
    cout << (res % Mod) << endl;
    return 0;
}
```

### 例 17-8 硬币购物(洛谷P1450,河南省队选拔2008)

共有 $4$ 种硬币。面值分别为 $c_1, c_2, c_3, c_{40}$ 某人去商店买东西，去了 $n$ 次，对于每次购买，他带了 $d$ 枚 $i$ 种硬币，想购买 $s$ 的价值的东西。请问每次有多少种付款方法。 $d_i$

**数据范围：** $1 \leqslant c_i, d_i, s \leqslant 10^5, 1 \leqslant n \leqslant 1000$。

**分析：**注意到一共有 4 种面值的硬币，因此共有 24 种使用方式。那不妨直接对这些情况分 $2^4$ 别讨论。考虑由于硬币个数的限制，最终是要做一个多重背包计数。直接计数的复杂度是 $O(4ns)$，运算量大概为 $4 \times 10^8$，正常情况下无法通过本题。

考虑一个更加优秀的思路。考虑虽然硬币个数很多，但是种类很少，同时发现不限制使用次数的方案数是很好计算的。于是考虑容斥。$f$ 表示不考虑硬币个数，用 4 种面值凑出 $v$ 的方 $f_v$ 案数。而对于不合法的方案数，考虑对于一种硬币 $(c_i, d_i)$，一个直观的想法下最后方案数应当是：

$$\sum_{j=d_i+1}^{+\infty} f_{s-c_i \cdot j}$$

但是发现背包模型在计算方案时，状态本身具有简并性。也就是对于任何一个状态 $f_{i,k}$ 都是由更小的 $f_{i,k-t} \cdot c_i^*$ 拼插”得到的。所以方案数应该为：

$$f_{s-c_i(d_i+1)}$$

因此，最终对所有不合法的情况容斥即可。最终复杂度为 $O(4 \cdot s + 16 \cdot n)$。代码如下：

```
#include <bits/stdc++.h>
#define MAXN 5
#define MAX 100010
#define ll long long
using namespace std;
ll T, Ans;
ll N, C[5], F[MAX], D[5], S, i, j, k;
inline ll WORK(ll X) {
    ll Y = S, Mark = 1;
    for (k = 0; k < 4; ++k)
        if ((X >> k) & 1)
            Y -= (D[k + 1] + 1) * C[k + 1], Mark = -Mark;
    return Y < 0 ? 0 : F[Y] * Mark;
}
int main() {
    F[0] = 1;
    for (i = 1; i <= 4; ++i)
        cin >> C[i];
    for (i = 1; i <= 4; ++i)
        for (j = C[i]; j < MAX; ++j)
            F[j] += F[j - C[i]];
    cin >> T;
    for (i = 1; i <= T; ++i) {
        Ans = 0, cin >> D[1] >> D[2] >> D[3] >> D[4] >> S;
        for (j = 0; j < 16; ++j)
            Ans += WORK(j);
        printf("%lld\n", Ans);
    }
    return 0;
}
```

## 17.4 课后习题与实验

### 习题 1 邦邦的大合唱站队(洛谷P3694)

$N$ 个偶像排成一列，他们来自 $M$ 个不同的乐队。每个乐队至少有一个偶像。现在要求重新安排队列，使来自同一乐队的偶像连续地站在一起。重新安排的办法是，，让若干偶像出列（剩下的偶像不动），然后让出列的偶像一个个归队到原来的空位，归队的位置任意。请问最少要让多少偶像出列?

**数据范围：** $1 \leqslant N \leqslant 10^5, 1 \leqslant M \leqslant 20$。

### 习题 2 砝码称重(洛谷P1441)

现有 $n$ 个砝码，重量分别为 $a_i$ 在去掉 $m$ 个砝码后，问最多能称量出多少不同的重量（不包括 $0$）。请注意，砝码只能放在天平一边。

**数据范围：** $n \leqslant 20, m \leqslant 4, m < n, a_i \leqslant 100.$

### 习题 3 玉米田(洛谷P1879,USACO2006年11月赛)

农夫买了一块 $M$ 行 $N$ 列长矩形的新牧场，每一格都是一块正矩形的土地。农夫打算在牧场上的某几格里种上草供奶牛享用。有些土地相当贫瘠不能用来种草。并且，农夫不会选择两块相邻的正矩形土地种草。农夫想知道，如果不考虑草地的总块数，那么，一共有多少种种植方案可供他选择?当然，把新牧场完全荒废也是一种方案。

**数据范围：** $1 \leqslant M \leqslant 12, 1 \leqslant N \leqslant 12$。

### 习题 4 炮兵阵地(洛谷P2704,NOI2011)

在 $N \times M$ 的网格地图上部署炮兵部队。一个 $N \times M$ 的地图由 $N$ 行 $M$ 列组成，地图的每一格可能是山地（用 $H$ 表示），也可能是平原（用 $P$ 表示）。在每一格平原地形上最多可以布置一支炮兵部队，山地上不能部署炮兵部队；一支炮兵部队在地图上的攻击范围是沿横向左右各两格，沿纵向上下各两格。图上其他白色网格均攻击不到。炮兵的攻击范围不受地形的影响。

在保证任何两支炮兵部队之间不能互相攻击的情况下（即任何一支炮兵部队都不在其他炮兵部队的攻击范围内），在整个地图区域内最多能够摆放多少炮兵部队？

**数据范围：** $N \leqslant 100, M \leqslant 10$，保证字符仅包含 $p$ 与 $\mathrm{h}$

### 习题 5 愤怒的小鸟(洛谷P2831,NOIP提高组2016)

“愤怒的小鸟”这款游戏是在一个平面上进行的。有一架弹弓位于（$0$，$0$）处，每次可以用弹弓向第一象限发射一只鸟，鸟的飞行轨迹均为形如 $y = ax^2 + bx$ 的曲线，其中 $a$，$b$ 是指定的参数，且必须满足 $a < 0, a, b$ 都是实数。当小鸟落回地面（即 $X$ 轴）时，它就会瞬间消失。

在游戏的某个关卡里，平面的第一象限中有 $n$ 只猪，其中第 $i$ 只猪所在的坐标为 $(x_i, y_i)$ 如果某只小鸟的飞行轨迹经过了 $(x_i, y_i)$，那么第 $i$ 只猪就会被消灭，同时小鸟将会沿着原先的轨迹继续飞行。例如，若两只小猪分别位于（$1$，$3$）和（$3$，$3$），可以选择发射一只飞行轨迹为 $y = -x^2 + 4x$ 的小鸟，这样两只小猪就会被这只小鸟一起消灭。

这款游戏一共有 $T$ 个关卡，对于每一个关卡，至少需要发射多少只小鸟才能消灭所有的小猪呢?

**数据范围：** $1 \leqslant n \leqslant 18, 1 \leqslant T \leqslant 30$。当 $n \geqslant 16$ 时保证 $T \leqslant 5$。

### 习题 6 密码(洛谷P4045,江苏省队选拔2009)

为了获取对方的登录口令，在暴力破解密码之前，必须先做大量的准备工作。经过情报的搜集，现在得到了若干有用信息，如“我观察到密码中含有字符串 $S$”。例如，对于一个 $10$ 位的密 $S''$ 码以及观察到的字符串 hello与world，可能的密码组合为helloworld 与worldhello；而对于 $6$ 位的密码以及观察到的字符串good与day，可能的密码组合为gooday。

有了这些信息，就能够大大地减少尝试的次数了。请编写一个程序，计算所有密码组合的可能。密码中仅可能包含 $a-z$ 之 $a-z$ 间的小写字母。

**数据范围：** $1 \leqslant L \leqslant 25, 1 \leqslant N \leqslant 10$，每个观察到的字符串长度不超过 $10$，并且保证输出结果小于 $2^{63}$。

### 习题 7 绿洲(AGC 12E)

给定 $n$ 个绿洲，第 $i$ 个绿洲的坐标为 $x$，保证 $x_i - 10^9 \leqslant x_1 < x_2 < \cdots < x_n \leqslant 10^9$，现在有一个人在沙漠中旅行，他初始背包的水容积为 $V$ 升，同时他初始拥有 $V$ 升水，每次到达一个绿洲时，他拥有的水的量将自动重置为容积上限（可以使用多次）。他现在可以选择两个操作来进行旅行：

1. 走路，行走距离为 $d$ 时，需要消耗 $d$ 升水。请注意，任意时刻拥有的水的数量不能为负数。
2. 跳跃，令 $v$ 为当前拥有的水量，若 $v > 0$，则可以跳跃至任意一个绿洲，然后重置容积上界 $v$>0 和所拥有的水量为 $v/21$ 向下取整）。

对于每一个 $i$ 满足 $1 \leqslant i \leqslant n$，需要求解，当把第 $i$ 个绿洲作为起点时，能否依次遍历其他所有绿洲。如果可以，输出Possible，否则输出Impossible。

数据范围 $:2 \leqslant N, V \leqslant 2 \times 10^5, -10^9 \leqslant x_1 < x_2 < \cdots < x_N \leqslant 10^9, V, x_i$ 均为整数。

### 习题 8 循环移位(洛谷CF1209E)

给定一个 $n \times m$ 的矩阵，可以对每一列进行若干次循环移位，求操作完成后每一行的最大值之和。

**数据范围:**

1. 简单版： $1 \leqslant t \leqslant 40, 1 \leqslant n \leqslant 4, 1 \leqslant m \leqslant 100, 1 \leqslant a_{i,j} \leqslant 10^5 \leqslant t \leqslant 10^6$
2. 困难版： $1 \leqslant t \leqslant 40, 1 \leqslant n \leqslant 12, 1 \leqslant m \leqslant 2000, 1 \leqslant a_{i,j} \leqslant 10^5$,

### 习题 9 最大前缀和(洛谷P5369,2018年北京大学信息学夏令营)

小 $C$ 遇到了一个非常难的问题：求一个序列的最大子段和。但是小 $C$ 并不会做这个题，于是小 $C$ 决定把序列随机打乱，然后取序列的最大前缀和作为答案。

小 $C$ 是一个非常有自知之明的人，他知道自己的算法完全不对，所以并不关心正确率，他只关心求出的解的期望值，现在请你帮他解决这个问题。由于答案可能非常复杂，所以只需要输出答案乘上 $n!$ 后对 $998244353$ 取模的值，显然这是个整数。

最大前缀和的定义： $\forall i \in [1, n], \sum_{j=1}^{i} a_j$ 的最大值。

**数据范围：** $1 \leqslant n \leqslant 20, :\sum_{i=1}^{n} : :|a[i]| \leqslant 10^9$,

### 习题 10 软件补丁问题(洛谷P2761)

$T$ 公司发现其研制的一个软件中有 $n$ 个错误，随即为该软件发放了一批共 $m$ 个补丁程序。每一个补丁程序都有其特定的适用环境，某个补丁只有在软件中包含某些错误而同时又不包含另一些错误时才可以使用。一个补丁在排除某些错误的同时，往往会加入另一些错误。

换句话说，对于每一个补丁 $i$，都有 $2$ 个与之相应的错误集合 `B1[i]` 和 `B2[i]`，使得仅当软件包含 `B1[i]` 中的所有错误，而不包含 B2[$i$ 中的任何错误时，才可以使用补丁 $i$。补丁 $i$ 将修复软 $i$ 件中的某些错误 `F1[i]`，而同时加入另一些错误 `F2[i]`。另外，每个补丁都耗费一定的时间。

试设计一个算法，利用 $T$ 公司提供的 $m$ 个补丁程序将原软件修复成一个没有错误的软件，并使修复后的软件耗时最少。对于给定的 $n$ 个错误和 $m$ 个补丁程序，找到总耗时最少的软件修复方案。

**数据范围：** $1 \leqslant n \leqslant 20, 1 \leqslant m \leqslant 100$

### 习题 11 奖励关(洛谷P2473,四川省队选拔2008)

在一个游戏关卡中，系统将依次随机抛出 $k$ 次宝物，每次都可以选择吃或者不吃。必须在抛出下一个宝物之前做出选择，且现在决定不吃的宝物以后也不能再吃。宝物一共有 $n$ 种，系统每次抛出这 $n$ 种宝物的概率都相同且相互独立。也就是说，即使前（$k-1$）次系统都抛出宝物

$1$（这种情况是有可能出现的，尽管概率非常小），第 $k$ 次抛出各个宝物的概率依然均为 $\frac{1}{n}$。

获取第 $i$ 种宝物将得到 $p_i$ 分，但并不是每种宝物都是可以随意获取的。第 $i$ 种宝物有一个前提宝物集合 $s$。只有当 $s$; 中所有宝物都至少吃过一次，才能吃第 $i$ 种宝物（如果系统抛出了一 $s_i$。 $s_i$ 个目前不能吃的宝物，相当于白白地损失了一次机会）。注意 $,p_i$ 可以是负数，但如果它是很多高分宝物的

前提，损失短期利益而吃掉这个负分宝物将获得更大的长期利益。假设采取最优策略，平均情况一共能在奖励关得到多少分值？

**数据范围：** $|\leqslant k\leqslant 100, 1\leqslant n\leqslant 15, -10^6\leqslant p_i\leqslant 10^6$

### 习题 12 小B的挑战(洛谷P2167,山东省队选拔2009)

给出 $N$ 个长度相同的字符串（由小写英文字母和?组成）， $S_1, S_2, \cdots, S_N$，求与这 $N$ 个串中的刚好 $K$ 个串匹配的字符串 $T$ 的个数，答案对 $1000003$ 取模。若字符串 $S_x(1\leqslant x\leqslant N)$ 和 $T$ 匹配，满足以下条件：

1. $|S_x|=|T|$
2. 对于任意的 $1\leqslant i\leqslant |S_x|$，满足 $S_x[i]=?$ 或者 $S_x[i]=T[i]$。

**数据范围：** $1\leqslant T\leqslant 5, 1\leqslant N\leqslant 15, 1\leqslant |S_i|\leqslant 50$，其中 $T$ 只包含小写英文字母。

# 第18章 动态规划的设计与优化

**动态规划**作为算法竞赛中出现频率高，且难度上限很高的话题，在状态设计以及转移的优化上有大量的技巧。如果读者尚不能熟练构造状态，请相信熟能生巧，在大量的练习下，自然而然就能理解如何构造状态以及如何在不同的状态之间转移。除此之外，使动态规划算法变得更加高效也是一个重要的目标。本章将介绍如何设计动态规划，以及一些动态规划的优化案例。本章思维导图如图 18-1 所示。




图18-1 本章思维导图

## 18.1 设计动态规划

状态的设计关系到转移的复杂度。设计出一个优秀的状态定义是很多问题的难点所在。在前面已经介绍过如何做**动态规划**题，这里提供理解动态规划的另一种角度。当然读者也可以总结出一套属于自己的方法。如果不喜欢这里的解释，也可以跳过本节。

首先，设计动态规划算法的过程有两个特别重要的地方。

### 1. 明确目标

设计状态前，需要明确要计算什么东西。这是动态规划的第一步，如果选择不当会导致无法解决问题或者复杂度偏高，所以要谨慎。如果选择计算的目标不太优秀，请尝试转换目标计算。

动态规划只是个工具：对于石子合并问题，直接对着题目的目标设计状态；对于错位排列问题，有了**容斥原理**转换后才容易进行动态规划。很多题目需要在一系列推导下，用性质转换题目，将其变为若干容易用动态规划解决的问题。

### 2. 设计状态和转移

在确定好计算的目标后，必然会得到与这个目标局面有关的若干变量（如每一步的决策、每个数组的值）。提取出若干变量作为状态的定义，那么每个局面可以通过这些变量来分类，形成若干“等价类”。一个状态可以对应着多个局面，一个局面也可以属于多个状态。

转移的意义就在于将多个局面一视同仁：同一个操作作用于一个状态的多个局面得到若干新局面，如果这些新局面属于同一个状态，那么设计这个转移的目标就达到了。

当对于所有局面能够执行的所有的操作都成功地对应到转移上，这个动态规划就设计成功了。这样也能够保证无后效性，因为一旦一次转移关系到后面的转移，那么“新局面属于同一状态”就不满足。也就是说，优秀的动态规划实际上就是设计出一个优秀的分类方案。

于是就有一个通用的步骤：

1. 设计分类方案，并将局面分类。
2. 对于每一个状态，将题目中每一个操作，将其套用到本身对应的局面上。
3. 验证操作后的局面是否属于同一个状态。
4. 验证每个状态所有局面是否能被转移到（这是不漏，有些题目还要求不重）。
5. 验证转移是否有环。

一般来说，都需要通过分析操作对局面的改变，来构造分类方案，然后再优化，这是比较自然的方法。上述的步骤不适合在构造时套用，而是适合用来验证。在构造时，时不时验证，效率会得到提升。

根据上面的步骤，容易发现在构造时要注意哪些局面可以归为一类；哪些局面不相关；哪些局面有先后顺序，哪些没有。

在这样的模型下，就存在一种从状态定义上优化动态规划的通用方法：不断地合并等价状态，即转移本质相同的状态，本质相同指执行相同的操作能到达相同状态。例如以计算序列 $A$ 最长上升子序列来解释，可以令 $f_i$ 为末尾元素下标是 $i$ 时的最长上升子序列，当存在一个 $j$ 使得 $A_j>A_i$ 时 $f_i$ 会转移到 $f_j$。

而从等价状态的角度来看，拿出所有末尾下标是 $i$ 的上升子序列，它们在末尾加上 $A_j$ 后都是末尾下标为 $j$ 的上升子序列。能够证明末尾下标相等的子序列转移后末尾仍相等，说明这是本质相同的转移。所以可将末尾下标相同的上升子序列作为相同的状态，在转移的时候一视同仁。

至于考虑到像之前定义的 $f_i$ 的意义：一个状态代表的局面是所有满足条件的局面，而最大值则是取自这些局面中的，那么 $f_i$ 存在的意义就是辅助计算这个最大值的过程。因为如果直接在最终状态取出所有局面计算答案，复杂度会变高，而随着状态的转移同时计算 $f_i$，就能快速计算答案。因为转移本质相同，所以对 $f$ 的转移也本质相同，这在计数型动态规划中体现得更加明显。

如果一开始直接以选择子序列的方案来作为状态，这么做就是成功优化了状态。再看看通用步骤中，转移后属于同一状态这个条件被满足了，每个局面的所有状态都被转移到了，并且转移没环。所以成功设计了一个动态规划算法。

这一套步骤有一定程度上的抽象，但是在“为什么动态规划要按子结构和无后效性的方式设计”这个问题上提供了一种解释。

## 18.2 优化转移

**动态规划**优化是指在不改变状态定义的情况下，尽可能加快转移速度，减少时间复杂度。本节将会通过一些例题介绍几种常见的动态规划的优化方法。

### 例 18-1 （前缀和优化）逆序对数列(洛谷P2513,河南省队选拔2009)

对于一个数列 $\{a_i\}$，如果有 $i<j$ 且 $a_i>a_j$，那么称 $a_i$ 与 $a_j$ 为一对逆序对数。求满足逆序对数为 $k$ 的 $1\sim n$ 排列有多少个。答案对 $10000$ 取模。$1\sim n$ 排列是指从 $1$ 到 $n$ 的每个自然数各出现一次的数列。

数据范围 $\cdot n\leqslant 1000, k\leqslant 1000$。

**分析：** 令 $f_{i,j}$ 表示 $1\tilde{}i$ 的排列中，逆序对数为 $j$ 的方案数。在 $1\tilde{}i-1$ 的排列中不同位置插入数字 $i$，从逆序对数为 $r$ 状态变化而来（逆序对数最少增加为 0，其实前面的数列已经是正序；最多增加 $i-1$，此时前面的数列是倒序），可得：

$$f_{i,j}=\sum_{r=\max\{0,j-(i-1)\}}^{j} f_{i-1,r}$$

直接这么做，复杂度是 $O(nk^2)$，会超时，需要进一步优化。

假设 $r$ 一直从 0 开始循环（不考虑 $r$ 的初始值），可以令 $S_j=\sum_{r=0}^{r=j} f_{i-1,r}$，这里的 $S$ 就是记录了 $f_{i,j}$ 的关于 $j$ 累加和。可得 $S_j=S_{j-1}+f_{i-1,j}, f_{i,j}=S_j$。

但是这里的 $r$ 可能不是从 0 开始循环的。当 $i$ 稍微大一点，要求 $f_{i,j}=\sum_{r=j-i+1}^{j} f_{i-1,i}$，则可以很容易得到 $f_{i,j}=S_j-S_{j-i}$。这里利用了**前缀和**缓存状态结果，把时间复杂度降到了 $O(nk)$。

实际上，并不需要建立一个数组 $S$，因为只需要维护一个长度固定的滑动窗口的和，因此可以使用一个变量 `sum` 记录这个滑动窗口的和每次计算前，把右侧新增值 $f_{i-1,j}$ 加入变量 `sum` 中；计算后，把最左侧的值 $f_{i-1,j-i+1}$ 从变量 `sum` 中减去。初始状态是 $f_{1,0}=1$，代码如下：

```
#include <cstdio>
#include <iostream>
using namespace std;
#define MAXN 1000 + 10
#define MOD 10000
int n, k, f[MAXN][MAXN];
int main() {
    cin >> n >> k;
    f[1][0] = 1;
    for (int i = 2; i <= n; i++) {
        int sum = 0;
        for (int j = 0; j <= k; j++) {
            sum = (sum + f[i - 1][j]) % MOD;
            f[i][j] = sum;
            if (j >= i - 1)
                sum = (sum - f[i - 1][j - i + 1] + MOD) % MOD;
        }
    }
    cout << f[n][k];
    return 0;
}
```

### 例 18-2 (前缀和优化，选读)夏令营(洛谷CF708E)

有一个（$n+2$）×$m$ 的网格。除了第 $1$ 行和最后一行，其他每一行每一天最左边和最右边的格子都有 $p$ 的概率消失。求 $k$ 天后，网格始终保持连通的概率。答案对 $10^9+7$ 取模。

数据范围 $n,m\leqslant 1.5\times 10^3, k\leqslant 10^5$

**分析：**为了方便分析，称两个不会变动的行分别为 0 和 $n+1$ 行。很容易想到状态 $f_{i,l,r}$ 表示第 $i$ 行，区间为$[l,r]$，第 0 行到第 $i$ 行都连通的概率。一行区间为$[l,r]$的概率，为左边消失恰好 $\iota_{l-1}$ 个和右边恰好消失 $m-r$ 个的概率。通过枚举哪几天有格子消失，可以得到 $k$ 天恰好消失 $i$ 个的概率 $g_i=\binom{k}{i}p^i(1-p)^{k-i}$。所以很容易得到转移：

$$f_{i,l,r}=g_{l-1}g_{m-r}\sum_{\max(l,l')\leqslant\min(r,r')}f_{i-1,l',r'}$$

然而对于这样的状态定义无论怎么优化，状态数都是 $O(nm^2)$ 的，显然过大了。考虑减小状态，只保留区间长度是无法继续转移的，所以要保留一个端点。令 $f_{i,r}$ 为第 $i$ 行的右端点为 $r$，第 0 行到第 $i$ 行都连通的概率。在转移的时候，枚举左端点 $l$，并计算与之相交的区间和 $|_0$ 由于这样定义状态丢失了左端点信息，所以考虑求不相交的方案，并将其从总方案中减去，因为求区间不相交仅仅需要 $r'<l$ 或 $l'>r$

对于 $r'<l$，在枚举 $l$ 的时候，直接计算 $f_{i-1,j}(j\leqslant l-1)$ 的和即可。

对于 $l'>r$，由于网格是对称的，所以 $f_{i,r}$ 同样可以表示左端点为 $m-r+1$ 且连通的概率，直接计算 $f_{i-1,j}(j\leqslant m-r)$ 的和即可。

所以得到转移：

$$f_{i,r}=\sum_{l=1}^{r}p_{l-1}p_{m-r}\left(\sum_{j=1}^{m}f_{i-1,j}-\sum_{j=1}^{l-1}f_{i-1,j}-\sum_{j=1}^{m-r}f_{i-1,j}\right)$$

如果不做任何优化，这样转移时间复杂度是 $O(nm^3)$ 的 。为了加速求和，将 $f_{i-1,j}$ 的**前缀和**记为 $h_j$ ，那么有：

$$f_{i,r} = \sum_{l=1}^{r} p_{l-1}p_{m-r}(h_m - h_{l-1} - h_{m-r})$$
$$= p_{m-r}\left((h_m - h_{m-r})\sum_{l=1}^{r} p_{l-1} - \sum_{l=1}^{r} p_{l-1}h_{l-1}\right)$$

若 $p_i$ 的前缀和为 $a_i, p_ih_i$ 的前缀和为 $b_i$，则转移显而易见：

$$f_{i,r} = p_{m-r}((h_m - h_{m-r})a_{r-1} - b_{r-1})$$

这样就通过前缀和优化将时间复杂度降到了 $O(nm)$。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 1.5e3 + 10;
const int MAXS = 1e5 + 10;
const int mod = 1e9 + 7;
typedef long long LL;
void reduce(int &x) { x += x >> 31 & mod; }
int mul(int a, int b) { return (LL)a * b % mod; }
int pow(int a, int b, int res = 1) {
    for (; b; b >>= 1, a = mul(a, a))
        if (b & 1)
            res = mul(res, a);
    return res;
}
void fma(int &x, int y, int z) { x = ((LL)y * z + x) % mod; }
int n, m, K;
int ps[MAXN], pre[MAXN];
int fac[MAXS], inv[MAXS];
int C(int a, int b) { return (LL)fac[a] * inv[b] % mod * inv[a - b] % mod; }
int f[MAXN][MAXN];
int main() {
    fac[0] = fac[1] = inv[0] = inv[1] = 1;
    for (int i = 2; i < MAXS; ++i) {
        fac[i] = mul(fac[i - 1], i);
        inv[i] = mul(inv[mod % i], mod - mod / i);
    }
    for (int i = 2; i < MAXS; ++i)
        inv[i] = mul(inv[i - 1], inv[i]);
    scanf("%d%d", &n, &m);
    int a, b;
    scanf("%d%d", &a, &b);
    int P = pow(b, mod - 2, a);
    scanf("%d", &K);
    for (int i = 0; i <= m && i <= K; ++i)
        ps[i] = pow(P, i, pow(1 + mod - P, K - i, C(K, i)));
    for (int i = 0; i <= m; ++i)
        reduce(pre[i] = ps[i] + (i ? pre[i - 1] - mod : 0));
    f[0][m] = 1;
    for (int i = 1; i <= n; ++i) {
        static int tp[MAXN], ts[MAXN];
```

```
        for (int j = 1; j <= m; ++j) { // 处理前缀和
            reduce(tp[j] = tp[j - 1] + f[i - 1][j] - mod);
            fma(ts[j] = ts[j - 1], ps[j], tp[j]);
        }
        for (int r = 1; r <= m; ++r) { // 处理转移
            int sm = mul(pre[r - 1], tp[m] - tp[m - r] + mod);
            reduce(sm -= ts[r - 1]);
            f[i][r] = mul(ps[m - r], sm);
        }
    }
    int ans = 0;
    for (int i = 1; i <= m; ++i)
        reduce(ans += f[n][i] - mod);
    printf("%d\n", ans);
    return 0;
}
```

### 例 18-3 (单调队列优化)能量收集(洛谷P3800,Byorangebird)

给定一个 $N$ 行 $M$ 列的棋盘，有 $K$ 个格子上的值为非零。要求在每一行选择一个格子，使得相邻行的格子列标号差不超过 $T$。最大化选取的格子权值和。

**数据范围：** $1 \leqslant N, M, T, K \leqslant 4000$

**分析：**记 $A_{i,j}$ 为格子的权值，很容易写出 $DP$ 状态：$f_{i,j}$ 为走到第 $i$ 行，第 $j$ 列最大的权值和。转移如下：

$$f_{i,j} = \max_{1 \leqslant k \leqslant m, |k-j| \leqslant T} f_{i-1,k} + A_{i,j}$$

从小到大枚举 $i$，发现转移实际上就是滑动窗口问题，即区间查询的左端点和右端点都是单调的。对于滑动窗口问题，一般使用**单调队列**解决。单调队列优化依赖的是元素的偏序关系，在本题中，如果查询的左、右端点是单调不降的，那么对于 $i<j$，如果有 $A_i < A_j$，则在右端点 $r \geq j$ 时，如果 $i$ 能贡献到答案，那么换成 $i$ 一定不会劣，也就是说 $i$ 被全面超越，不再有用。

这样对于队列中的元素，值 $A_i$ 是单调降的，只需要保证在加入元素的同时，删掉队列尾无用元素，就能维护队列的单调性。在查询最大值的时候，队列首会因为超出询问范围而被弃用。

至此通过使用队列维护区间内的单调降序列，完成了单调队列优化DP，代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 4010;
int f[MAXN][MAXN], A[MAXN][MAXN], n, m, K, T;
int main() {
    scanf("%d%d%d%d ", &n, &m, &K, &T);
    for (int i = 1; i <= K; ++i) {
        int x, y, v;
        scanf("%d%d%d", &x, &y, &v);
        A[x][y] = v;
    }
    for (int i = 1; i <= n; ++i) {
        static int que[MAXN];
        int head = 1, tail = 0, r = 0;
        const int *g = f[i - 1];
```

```
        for (int j = 1; j <= m; ++j) {
            while (r < m && r < j + T) {
                ++r;
                // 去除无用元素
                while (head <= tail && g[que[tail]] <= g[r])
                    --tail;
                que[++tail] = r;
            }
            // 查看队列首是否"过期"
            while (head <= tail && que[head] < j - T)
                ++head;
            // 由单调性,此时队列首一定是区间最大值
            f[i][j] = (head <= tail ? g[que[head]] : 0) + A[i][j];
        }
    }
    int ans = 0;
    for (int i = 1; i <= m; ++i)
        ans = max(ans, f[n][i]);
    printf("%d\n", ans);
    return 0;
}
```

### 例 18-4 (单调栈优化)最长上升子序列

给定一个长度为 $n(n\leqslant 10^6)$ 的序列 $a$，求出 $a$ 最长的子序列使得这个子序列是单调增的。

**分析**：之前介绍过这个问题的状态定义方法，令 $f_i$ 为末尾为 $a_i$ 的最长上升子序列长度。那么有转移：

$$f_i=\max_{j<i,a_j<a_i} f_j+1$$

直接转移时间复杂度是 $O(n^2)$ 的，考虑到单调性：对于末尾为 $a_k$ 的所有上升子序列中，如果长度为 $i$ 的上升子序列存在，那么长度为 $i-1$ 的也存在。在满足 $j<i,k<i$，且 $f_j=f_k$ 时，如果 $a_j<a_k$ 显然取 $j$ 比取 $k$ 优。因此如果记 $g$ 为考虑前 $i$ 个数，长度为 $j$ 的上升子序列，末尾最小是多少， $g_{i,j}$ 那么这个序列是单调不降的。

接下来只要维护栈 $g$，那么在 $g$ 中找到最大的 $g_j^*<a_i$，就能得到 $\max_{j<i,a_j<a} f_j$。

在得到 $f_i=j+1$ 之后，由于 $g_j<a_i\leqslant g_{j+1}<g_{j+2}$，那么直接修改 $g_{j+1}=a_i$，不影响单调性，并且维护了 $g$。

如果栈 $g$ 的元素不够 $j+1$ 个，就在栈顶添加新元素。这种技巧称为**单调栈优化**。本题时间复杂度为 $O(n\log n)$，代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 1e6 + 10;
int n, A[MAXN], f[MAXN], top;
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; ++i)
        scanf("%d", &A[i]);
    memset(f, 0x3f, (n + 1) << 2);
    for (int i = 1; i <= n; ++i) {
        int at = lower_bound(f + 1, f + 1 + top, A[i]) - f;
        f[at] = A[i];
        top = max(top, at);
    }
    printf("%d\n", top);
    return 0;
}
```

### 例 18-5 (线段树优化) 面包师(洛谷CF833B)

将一个长度为 $n$ 的序列分为 $k$ 段，使得总价值最大。一段区间的价值表示为区间内不同数字的个数。

数据范围 $\cdot n\leqslant 35000, k\leqslant 50$。

**分析：**很容易想到用 $f_{i,k}$ 表示将前 $i$ 个数分成 $k$ 段的最大价值。但是这样子转移必须是一段一段转移的，也就是记 `w(l,r)` 表示区间$[l,r]$中有多少个不同的数字，得到转移：

$$f_{i,k+1}=\max_{0\leqslant j<i}\{f_{j,k}+w(j+1,i)\}$$

这样时间复杂度是 $O(n^2k)$，明显会超时。

问题出在“区间有多少不同数字”上面，考虑这样一个问题：给定一个序列，多次询问区间$[l,r]$有多少不同数字。

传统的离线做法，是使用扫描线加上**线段树**的方法，对于询问的右端点从小到大处理。在右端点 $r$ 增大的时候，不断维护每个 $l$ 对应的 $w(l,r)$ 考虑 $r$ 到 $r+1$ 带来的变化，记 $pre_i=\max_{a_j=a_i,j<i} j$，即 $a_i$ 上次出现的位置，那么当右端点到达 $r+1$ 时，对于 $l\in(pre_{i+1},i+1]$，其 $w\,(l,r+1)=w(l,r)+1$，其余的 $w(l,r+1)=w(l,r)$。

这样在线段树上做区间修改单点查询即可得到 $w$（$l$，$r$）。

类似的，在本题中，考虑到 $k$ 很小，为每个 $k$ 维护一棵线段树，线段树里维护 $f_{l-1}+w\,(l\,,r)$ 的区间最大值，这样就可以通过区间查询来完成转移，并通过单点修改来更新 $f_{i,k0}$ 本题时间复杂度为 $O(nk\log n)$，代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 35010;
struct T {
    int tree[MAXN << 2], tag[MAXN << 2];
    void pushdown(int u) {
        if (tag[u]) {
            tree[u << 1] += tag[u];
```

```
            tree[u << 1 | 1] += tag[u];
            tag[u << 1] += tag[u];
            tag[u << 1 | 1] += tag[u];
            tag[u] = 0;
        }
    }
    void modify(int u, int l, int r, int L, int R, int v) {
        if (L > R)
            return;
        if (L <= l && r <= R) {
            tree[u] += v;
            tag[u] += v;
            return;
        }
        pushdown(u);
        int mid = l + r >> 1;
        if (L <= mid)
            modify(u << 1, l, mid, L, R, v);
        if (mid < R)
            modify(u << 1 | 1, mid + 1, r, L, R, v);
        tree[u] = max(tree[u << 1], tree[u << 1 | 1]);
    }
    int query(int u, int l, int r, int L, int R) {
        if (L > R)
            return 0;
        if (L <= l && r <= R)
            return tree[u];
        pushdown(u);
        int mid = l + r >> 1;
        int res = 0;
        if (L <= mid)
            res = max(res, query(u << 1, l, mid, L, R));
        if (mid < R)
            res = max(res, query(u << 1 | 1, mid + 1, r, L, R));
        return res;
    }
} tree[51];
int n, K, a[MAXN], pre[MAXN], lst[MAXN];
int main() {
    scanf("%d%d", &n, &K);
    for (int i = 1; i <= n; ++i) {
        scanf("%d", &a[i]);
        pre[i] = lst[a[i]];
        lst[a[i]] = i;
        for (int j = 0; j < K; ++j) {
            // 动态维护f+w
            tree[j].modify(1, 0, n, pre[i], i - 1, 1);
            // 利用区间查询加速转移中的max
            int tv = tree[j].query(1, 0, n, 0, i - 1);
            // 更新转移后的结果
            tree[j + 1].modify(1, 0, n, i, i, tv);
        }
    }
    printf("%d\n", tree[K].query(1, 0, n, n, n));
    return 0;
}
```

### 例 18-6 (斜率优化)任务安排(洛谷P5785,山东省队选拔2008)

机器上有 $n$ 个需要处理的任务，编号为 $1$ 到 $n$。这 $n$ 个任务需要被依次完成，且需要将它们分成若干批，每批包含相邻的若干任务。从时刻 $0$ 开始，这些任务被分批加工，第 $i$ 个任务单独完成所需的时间是 $T_i$。在每批任务开始前，机器需要启动时间 $s$，而完成这批任务所需的时间是各个任务需要时间的总和。注意，同一批任务将在同一时刻完成。每个任务的费用是它的完成时刻乘以一个费用系数 $C_i$。请确定一个分组方案，使得总费用最小。

**数据范围：** $1 \leqslant n \leqslant 3 \times 10^5, 1 \leqslant s \leqslant 2^8, |T_i| \leqslant 2^8, 0 \leqslant C_i \leqslant 2^8$。

**分析：** 抛开数据范围，先考虑朴素的做法。用 $f_{i,j}$ 表示将前 $i$ 个任务分成 $j$ 批完成的最小费用，那么对于每一个任务，都可以选择将它和前一个任务放在同一批，或者是单独设置成新的一批。假设最后一批任务之前的最后一个任务是 $k$，则完成前 $k$ 个任务所需花费是 $f_{k,j-1}$，最后一批任务完成时间是 $j \times s + T_1 + \cdots + T_i$，最后一批任务的费用系数和为 $C_{k+1} + \cdots + C_{i0}$ 令 $St_n = \sum_{i=1}^{n} TiSc_n = \sum_{i=1}^{n} Ci$（即 $T$ 数列和 $C$ 数列的**前缀和**），则可以得到转移方程：

$$f_{i,j} = \min_{0 \leq k < i} f_{k,j-1} + (j \times s + St_i)(Sc_i - Sc_k)$$

这种做法的时间复杂度是 $O(n^3)$，需要进一步优化，尝试优化 $i$ 这一维。可以发现，这里的只用于计算最后一批任务的总体时间偏移量。如果不知道 $j$，就不能确定最后一批任务的开始时间。但如果增加了一个任务分组，则从这个任务开始之后的每一个任务都会延迟 $s$ 的时间，这将导致总费用增加了 $s \times (C_k + \cdots + C_n) = s \times (Sc_n - Sc_{k-1})$。

这时的 $f_i$ 就可以表示为前 $i$ 个任务分成若干批后的最小费用，加上后续任务因为增加任务批次而增加的额外费用，最后 $f_n$ 就是总费用的最小值（毕竟第 $n$ 个任务后面没有其他任务了，自然就没有额外的费用了）。转移方程如下：

$$f_i = \min_{0 \leqslant k < i} \{f_k + St_i(Sc_i - Sc_k) + s(Sc_n - Sc_k)\}$$

这种做法的时间复杂度是 $O(n^2)$，还需要继续优化。继续整理这个转移方程，把与 $k$ 无关的项放在 `min()` 函数外面，可得：

$$f_i = \min_{0 \leqslant k < i} \{f_k - (St_i + s)Sc_k\} + St_i \cdot Sc_i + s \cdot Sc_n$$

把 `min()` 函数去掉，用其他项表示 $f_k$ 可得：

$$f_k = (St_i + s)Sc_k + f_i - St_i \cdot Sc_i - s \cdot Sc_n$$

这可以视为关于自变量 $Sc_k$ 的一次函数，斜率为 $St_i + s$，截距为 $f_i - St_i \cdot Sc_i - s \cdot Sc_n$。每个决策对应平面中的点 $(Sc_k, f_k)$。在固定 $i$ 时，需要选出使 $f_i$ 最小的决策 $k$。图 18-2（a） 中，每个点代表一个决策；选择对应直线截距最小的点，就使 $f_i$ 最小。

限定 $T_i$ 非负时，当 $k$ 增大时， $Sc_k$ 也是增大的。随着 $i$ 的增大，斜率 $St_i + s$ 也会越来越大。考查图 18-2(b) 的情况， $P_1, P_2, P_3$ 呈现 $\overline{\int^{\infty}}$ 上凸”的状态。

假设 $P_1$ 是状态为 $i$ 时的最佳决策，那么 $P_2$ 不可能成为状态为 $i+1$ 时的最佳决策，以后也不可能是。这是因为如果选择 $P_3$ 作为决策，结果会更优。

因此，这些最佳决策点组成了一个下凸的凸包，如图 18-2(c) 所示。将这些点依次相连，每条线段的斜率是单调递增且非负的。使用**单调队列**维护这一组点，满足斜率符合单调递增的性质。具体而言，处理状态 $i$ 时，先连续判断单调队列的前两个决策点的斜率是否不大于当前状态对应的斜率 $St_i+s$，如果不大于则弹出队首，直到前两个决策点的斜率大于当前状态对应的斜率，或者只剩下唯一点。

此时，队首元素就是最佳决策点，用来转移状态。然后尝试将新的决策点插入单调队列中，如果队列中最后两个点和新的决策点不满足下凸性质（连线的斜率不是依次递增的），将队尾元素退队，直到满足这个性质，或者只剩下最后一个点为止。这种做法的时间复杂度是 $O(n)$ $C$

然而，由于 $T_i$ 可能是负数，斜率也有可能是负数；同时，每个状态对应的斜率 $St_i+s$ 也不具有单调性。这时，这一系列最佳决策点也组成了一个下凸的凸包，只是这个凸包的所有点都需要被保留，队首不需要被弹出。这时需要找到一个点，这个点和前一个点连线的斜率是 $k_1$，和后一个点的连线斜率是 $k_2$，满足 $k_1 < St_i + s < k_2$，这个点就是最佳决策点，如图 18-2(d) 所示。

可以在单调队列使用二分查找，找到符合要求的决策点。而新的决策点入队的方式和上一段介绍的相同。代码如下：

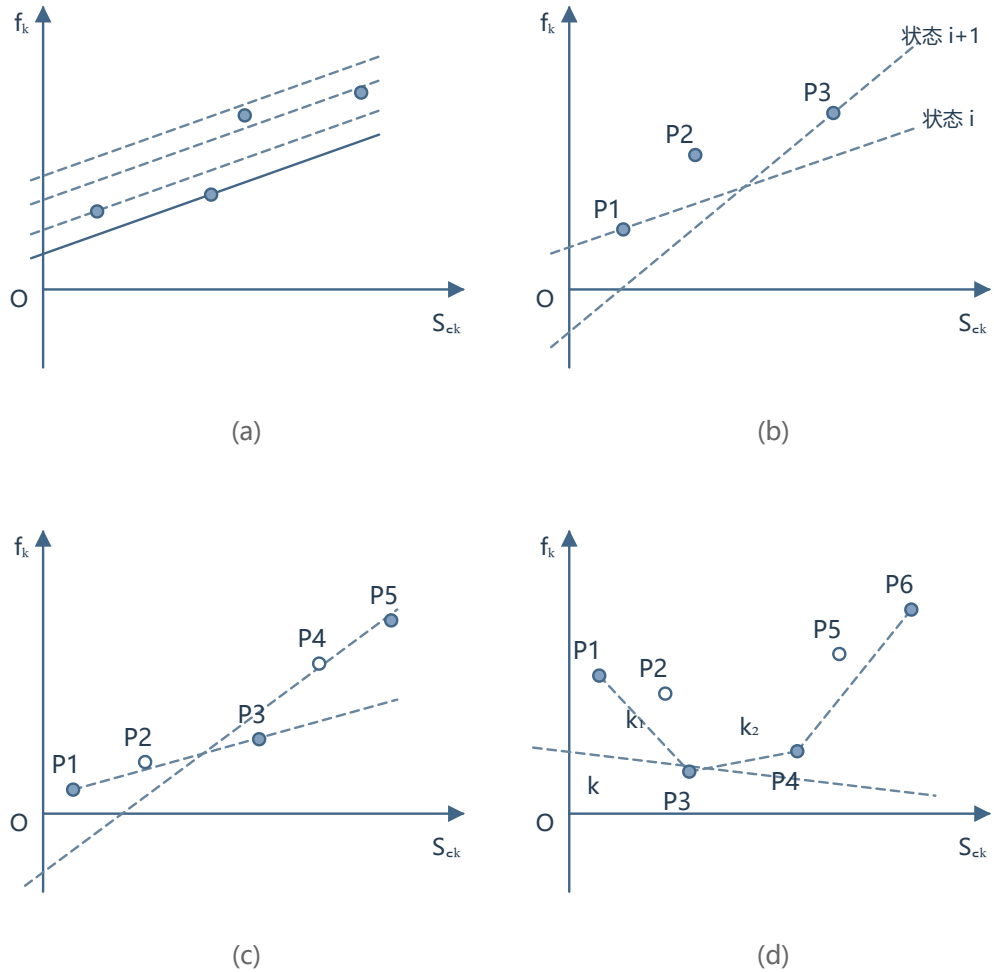



图18-2 斜率优化

```
#include <iostream>
using namespace std;
#define MAXN 300000 + 10
long long q[MAXN];
long long head = 1, tail = 0, n, s;
long long Sc[MAXN], St[MAXN], f[MAXN];
int find(int L, int R, long long S) {
    int M = 0, ans = tail;
    while (L <= R) {
        M = (L + R) / 2;
        if (M == tail) {
            ans = M;
            break;
        }
        if (f[q[M + 1]] - f[q[M]] > S * (Sc[q[M + 1]] - Sc[q[M]]))
            R = M - 1, ans = M;
        else
            L = M + 1;
    }
    return q[ans];
}
int main() {
    cin >> n >> s;
    for (int i = 1; i <= n; i++) {
        cin >> St[i] >> Sc[i];
        St[i] += St[i - 1];
        Sc[i] += Sc[i - 1];
    }
    q[++tail] = 0;
    for (int i = 1; i <= n; i++) {
        int p = find(head, tail, s + St[i]);
        f[i] = f[p] + s * (Sc[n] - Sc[p]) + St[i] * (Sc[i] - Sc[p]);
        while (head < tail && (f[q[tail]] - f[q[tail - 1]]) * (Sc[i] - Sc[q[tail]]) >=
                                  (f[i] - f[q[tail]]) * (Sc[q[tail]] - Sc[q[tail - 1]]))
            tail--;
        q[++tail] = i;
    }
    cout << f[n];
    return 0;
}
```

这种做法的时间复杂度是 $O(n\log n)$。代码实现时遇到比较斜率大小，可以将其除法转换为乘法（同乘分母）以避免浮点数运算。

对于不同的斜率优化题目，由于单调性的不同，维护凸包的数据结构可能是队列、栈，甚至是平衡树。在凸包上查询的方法可能是单调的指针，也可能是二分。所以在建立点积最大化的模型后（最小化的问题可以通过取负号转化为最大化的），需要对所有点的单调性进行分析。

### 例 18-7 (斜率优化，选读) 序列分割(洛谷P3648,APIO2014)

有一个关于长度为 $n$ 的非负整数序列的游戏，这个游戏中需要把序列分成 $k+1$ 个非空的块。为了得到 $k+1$ 块，需要重复下面的操作 $k$ 次：

1. 选择一个有超过一个元素的块（初始时只有一个块，即整个序列）。
2. 选择两个相邻元素把这个块从中间分开，得到两个非空的块。

每次操作后将获得那两个新产生的块元素和的乘积作为分数。最大化最后的总得分。

**数据范围：** $2 \leqslant n \leqslant 100000, 1 \leqslant k \leqslant \min\{n-1, 200\}$。

**分析：**根据乘法分配律，最后的得分只和分割位置有关，和分割顺序无关，也就是每一块权值和两两相乘。为了快速地计算每一块的权值和，对原序列进行**前缀和**得到数组 $s_i$。那么计算每一块的贡献，令 $f_{k,i}$ 为前 $i$ 个元素中分成 $k$ 段的最大得分。很容易得到转移：

$$f_{k+1,i} = \max_{j<i}\{f_{k,j} + (s_i - s_j)s_j\}$$

但是，这样的转移复杂度是 $O(n^2k)$，明显会超时。

观察转移式，可以发现 $(f_{k,j} - s_j s_j) + s_i s_j$ 其实就是向量 $(s_j, f_{k,j} - s_j^2)$ 和向量 $(s_i, 1)$ 的点积。

而转移实际上就是最大化点积的形式。向量可以对应到二维平面上的点，当给定点 $B$ 时，在一系列点 $S$ 中寻找一个点 $A \in S$，最大化 $\overrightarrow{OA} \cdot \overrightarrow{OB}$，就相当于最大化 $\overrightarrow{OA}$ 到 $\overrightarrow{OB}$ 的投影长度。

由凸包的性质，答案一定在点集的凸包中选定：要最大化投影长度，就相当于用垂直于向量的直线找凸包的切点。如图 18-3 所示， $B_1, B_2$ 分别对应着凸包的切点 $A_1, A_2$ .

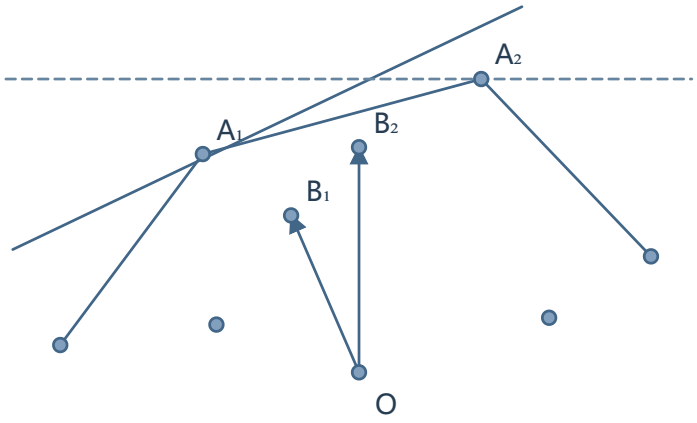



图18-3二维平面上的斜率优化

特殊地，在这道题中，由于查询用的向量 $(s_i, 1)$ 在一、二象限，只需保存上凸壳即可。

容易发现也容易证明，随着查询向量的 $s_i$ 单调不降，点积最大的点的横坐标也就是 $s_j$ 单调不降。对应到 $s$ 单调不降也就是转移点 $i$ 单调不降。这样的单调性对应到凸包上，则表现为斜率单调不增，因此这样的技巧被称作斜率优化。

于是在固定 $k$，转移到 $k+1$ 时，问题被转化成：

操作 1：动态插入点（$x$， $y$ ），维护点集的凸包，保证 $x$ 单调不降。

操作 2：给定点（$x$， $y$ ），查询使得点积最大化的点的编号，保证 $x$ 单调不降。

对于操作 1，只需要不断地判断插入点后是否还是凸包，如果不是则删去凸包中横坐标最大的点再继续判断。判断凸包的过程可以使用叉积，假设插入点为 $P$，最后两个点为 $A$， $B(x_A < x_B)$ 则如果 $(B/-P) \times (A/-B/ > 0$ 则插入后就是凸的。这道题中因为相邻两次插入的横坐标可能会相等，所以还需要注意一些细节。

叉积的比较方法经过移项也可以看作是斜率比较，如果精度要求不严格，也可以使用计算斜率的方法来维护凸包，这样的方法更加容易理解。笔者更加推荐使用叉积的方法，因为这样在运行效率和精度上都占有优势。

对于操作 2.可以在凸包上维护单调不降的指针表示目前最优解。如果在新的一次查询中，指针的后继能带来更大的答案，就让指针后移，因为在接下来的询问中，前面的点一定不会比后面的更好。

一个细节少的实现方法就是使用队列，因为抛弃点的过程，只需要比较队列最前面两个元素占积的大小，在取答案时只需要取队列首即可。同时这样做也不会因为操作 1 删去了点，导致维护的指针指向一个被删去的点。

由于每个点被插入一次，至多删除一次，查询点单调，所以复杂度是 $O(nk)$。代码如下

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 100010;
typedef long long LL;
LL f[2][MAXN], pre[210][MAXN];
int head, tail, n, k, a[MAXN];
struct vec {
    int id, x;
    LL y;
    LL operator()(int v) { return (LL)x * v + y; }
} q[MAXN];
LL cross(vec a, vec b, vec c) {
    // 计算 (B-P) 和(A-B) 的叉积
    a.x -= b.x, a.y -= b.y;
    b.x -= c.x, b.y -= c.y;
    return b.x * a.y - a.x * b.y;
}
int main() {
    scanf("%d%d", &n, &k);
    for (int i = 1; i <= n; ++i) {
        scanf("%d", &a[i]);
        a[i] += a[i - 1];
    }
    for (int T = 1; T <= k; ++T) {
        head = tail = 0;
        q[0] = (vec){0, 0, 0};
        for (int i = 1; i <= n; ++i) {
            // 计算的是队列首第2个点是否比第1个点优
            // 如果更优就永久抛弃第1个点
            while (head < tail && q[head](a[i]) <= q[head + 1](a[i]))
                ++head;
            // 这时候队列首一定是点积最大化的
            f[T & 1][i] = q[head](a[i]);
            pre[T][i] = q[head].id;
            vec x = (vec){i, a[i], f[T & 1 ^ 1][i] - (LL)a[i] * a[i]};
            // 在插入点的时候通过不断弹出点保持凸壳的凸性
            while (head < tail && cross(q[tail - 1], q[tail], x) <= 0)
                --tail;
            q[++tail] = x;
        }
```

```
    }
    printf("%lld\n", f[k & 1][n]);
    int now = n;
    for (int i = k; i >= 1; --i) {
        printf("%lld ", pre[i][now]);
        now = pre[i][now];
    }
    return 0;
}
```

该技巧的本质，就是在凸包上分析插入和查询的单调性，利用这些单调性来加速转移。

对于不同的斜率优化题目，由于单调性的不同，维护凸包的数据结构可能是队列、栈，甚至是平衡树。在凸包上查询的方法可能是单调的指针，也可能是二分。所以在建立点积最大化的模型后，需要对所有点的单调性进行分析。

### 例 18-8 (分治套斜率优化)货币兑换(洛谷P4027,NOI2007)

某家证券交易所只发行交易 $A$ 证券和 $B$ 证券两种证券。持有证券的顾客都有一个自己的账户。证券的数目可以是实数。每一单位证券当天可以兑换的人民币数目会随着市场的波动而变化。记录第 $K$ 天中 $A$ 证券和 $B$ 证券的价值分别为 $A_K$ 和 $B_K$ （元/单位证券）。同一天内可以进行多次操作。

为了方便顾客，证券交易所提供了一种比例交易法，分为以下两方面：

1. 卖出证券。顾客提供一个$[0,100]$内的实数OP作为卖出比例，其意义为：将 OP%的 $A$ 证券和OP%的 $B$ 证券以当时的价值兑换为人民币。
2. 买入证券。顾客支付IP元人民币，交易所将会兑换给用户总价值为IP的证券，并且满足提供给顾客的 $A$ 证券和 $B$ 证券的比例在第 $K$ 天恰好为 $Rate_K$。

小 $Y$ 预判了未来 $N$ 天内的 $A$ 证券和 $B$ 证券的价值以及Rate。他还希望能够计算出来，如果开始时拥有 $S$ 元钱，那么 $N$ 天后最多能够获得多少元钱。必然存在一种最优的买卖方案满足：每次买进操作使用完所有的人民币，每次卖出操作卖出所有的证券。

保证 $N\leqslant 10^5, 0<A_K\leqslant 10, 0<B_K\leqslant 10, 0<Rate_K\leqslant 100$，答案 $\leqslant 10^9$。

**分析：**限制中最后一句话其实是提示。假设每时每刻手上只有钱，即 $f_i$ 表示前 $i$ 天手上最多有多少钱。那么 $f$ 作为手上只有钱，只可能是第 $i-1$ 天手上只有钱，也可能是第 $j<i$ 天买入证 $f_ij<i$ 券后第 $i$ 天刚好卖出。所以很容易写出转移：

$$f_i=\max\left\{f_{i-1},\max_{j<i}\left\{f_j\times\frac{A_iRate_j+B_i}{A_jRate_j+B_j}\right\}\right\}$$

注意到右边一项，仍然可以表示成 $\left(\frac{f_jRate_j}{A_jRate_j+B_j},\frac{f_j}{A_jRate_j+B_j}\right)$ 与 $(A_i,B_i)$ 的点积。这是一个点积最大化问题。

很容易发现，由于插入凸包的点和查询的点，其横坐标都不是单调的。直接使用平衡树并且使用平衡树上二分即可，时间复杂度为 $O(n\log n)$，但是这样太难写，常数还不小。

考虑使用**分治**，分成区间[$l$，mid]和[mid+1，$r$]的时候，先转移完[$l$，mid]，再考虑[$l$，mid]对[mid+1，$r$]的贡献，再考虑[mid+1，$r$]区间内的贡献。这样分治只需要着重考虑[$l$，mid]对[mid+1.$r$]的贡献，对于[$l$，mid]是计算凸包，对于[mid+1，$r$]是在凸包上查询，由于区间不相交所以不用考虑转移时 $i,j$ 的

大小关系。

在分治的同时，使用归并排序按横坐标来归并询问，因为当询问的横坐标单调，就能以 $O(\partial|V|+Q)$ 也就是凸包点数加上查询个数的时间复杂度完成查询。

在分治的同时，使用归并排序按横坐标来归并查询用的点集，因为当点集的横坐标单调，就能用 $O(|v|)$ 也就是点集大小求出其上凸壳。

本题时间复杂度为 $O(n\log n)$，代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 100010;
typedef long long LL;
const double eps = 1e-8;
double f[MAXN];
int n, q[MAXN];
struct P {
    double a, b, r, x, y, k;
    int w, id;
    inline bool operator<(const P &b) { return k > b.k; }
} p[MAXN], t[MAXN];
inline double slope(int j, int k) {
    if (fabs(p[j].x - p[k].x) < eps)
        return 1e15;
    return (p[j].y - p[k].y) / (p[j].x - p[k].x);
}
void solve(int l, int r) {
    if (l == r) {
        f[l] = max(f[l], f[l - 1]);
        p[l].y = f[l] / (p[l].a * p[l].r + p[l].b);
        p[l].x = p[l].y * p[l].r;
        return;
    }
    int mid = l + r >> 1, cur1 = l, cur2 = mid + 1, cur;
    // 由于事先对查询点已按横坐标排序,所以只须将查询点分成左右两个区间
    for (int i = l; i <= r; ++i)
        if (p[i].id > mid)
            t[cur2++] = p[i];
        else
            t[cur1++] = p[i];
    for (int i = l; i <= r; ++i)
        p[i] = t[i];
    solve(l, mid); // 处理左半边区间
    // 处理左半边区间对右半边区间的贡献
    cur1 = 1;
    cur2 = 0;
    // 建立凸壳
    for (int i = l; i <= mid; ++i) {
        while (cur1 < cur2 && slope(q[cur2 - 1], q[cur2]) < slope(q[cur2], i) + eps)
            --cur2;
        q[++cur2] = i;
    }
    // 使用单调的指针查询答案
    for (int i = mid + 1; i <= r; ++i) {
        while (cur1 < cur2 && slope(q[cur1], q[cur1 + 1]) + eps > p[i].k)
            ++cur1;
        int t = q[cur1];
```

```
            f[p[i].id] = max(f[p[i].id], p[i].a * p[t].x + p[i].b * p[t].y);
        }
        solve(mid + 1, r); // 处理右半边区间
        cur = cur1 = l;
        cur2 = mid + 1;
        // 对查询用的点集进行归并排序
        while (cur1 <= mid && cur2 <= r)
            if (p[cur1].x < p[cur2].x + eps)
                t[cur++] = p[cur1++];
            else
                t[cur++] = p[cur2++];
        while (cur1 <= mid)
            t[cur++] = p[cur1++];
        while (cur2 <= r)
            t[cur++] = p[cur2++];
        for (int i = l; i <= r; ++i)
            p[i] = t[i];
    }
    int main() {
        scanf("%d%lf", &n, &f[0]);
        for (int i = 1; i <= n; ++i) {
            scanf("%lf%lf%lf", &p[i].a, &p[i].b, &p[i].r);
            p[i].k = -p[i].a / p[i].b;
            p[i].id = i;
        }
        sort(p + 1, p + 1 + n);
        solve(1, n);
        printf("%.3lf\n", f[n]);
        return 0;
    }
```

## 18.3 课后习题与实验

### 习题 1 翻转矩形(ARC 081F)

给定一个 $n \times m (1 \leqslant n, m \leqslant 2000)$ 的 $0-1$ 矩阵 $A$，可以任意取反一些行或列，求可能出现的面积最大的全 $0$ 子矩形。

### 习题 2 缩小社交圈(洛谷P5464)

有 $n$ 个人，第 $i$ 人有属性区间 $[l_i, r_i]$。当两个人的属性区间交集不为空时，这两个人有联系。现在希望挑选出一些人组成一个集合 $S$，如果将所有集合内的人中有联系的两人连上边，则 $S$ 刚好成为一棵树。请问，有多少种可以选择的方案？由于答案可能很大，请对 $10^9+7$ 取模。

数据范围 $\cdot n \leqslant 2000, 1 \leqslant l_i < r_i \leqslant 4000$。

### 习题 3 游戏闯关(洛谷P4099,河北省队选拔2013)

某款游戏对 $n$ 个关卡给出 $n-1$ 条先后顺序限制，例如第 $i$ 个关卡必须先于第 $j$ 个关卡。忽略限制的方向后，这些关系构成一棵树，即任意两个关卡之间都有关联。求满足所有限制的关卡挑战顺序数，答案对 $10^9+7$ 取模。

**数据范围：** 一个测试点最多 $5$ 组数据， $1 \leqslant n \leqslant 1000$。

### 习题 4 找宝石(Codeforces Gym101002H)

有 $n$ 个物品，每个物品有一个体积 $w_i$ 和价值 $v_i$，现在要求对 $V \in [1,m]$，求出体积为 $V$ 的背包能够装下的最大价值。

**数据范围：** $,n \leqslant 10^6, m \leqslant 10^5, 1 \leqslant w_i \leqslant 300, 1 \leqslant v_i \leqslant 10$。

### 习题 5 小鸟(洛谷P3572,POI2014)

有 $n$ 棵树，第 $i$ 棵树的高度是 $d_i$。小鸟要去第 $n$ 棵树。如果小鸟在第 $i$ 棵树，那么它可以跳到第 $i+1, i+2, \cdots, i+k$ 棵树。如果它跳到一棵不矮于当前树的树，那么它的劳累值会 $+1$；否则不会。求小鸟到达第 $n$ 棵树的最小劳累值。$q$ 次询问，每次给出不同的 $k$，求答案。

数据范围 $\cdot 2 \leqslant n \leqslant 10^6, 1 \leqslant q \leqslant 25, 1 \leqslant k_i \leqslant n-1$。

### 习题 6 舞会(洛谷P2254,NOI2005)

船上的舞厅是一个 $N$ 行 $M$ 列的矩阵，矩阵中某些方格上堆放了家具，其余方格是空地。钢琴可以在空地上滑动，但不能撞上家具或滑出舞厅。每个时刻，钢琴都会随着船体倾斜的方向向相邻的方格（东、西、南或北）滑动一格。可以选择施魔法或不施魔法：如果不施魔法，则钢琴会滑动；如果施魔法，则钢琴会原地不动。已知 $T$ 个单位的时间，还有 $K$ 段时间区间，以及每段时间的船体的倾斜情况。希望钢琴在舞厅里滑行的路程尽量长，请求最长滑行长度。

**数据范围：** $1 \leqslant N, M \leqslant 200, K \leqslant 200, T \leqslant 40000$

### 习题 7 股票交易(洛谷P2569,四川省队选拔2010)

某同学预测到了未来 $T$ 天内某只股票的走势，第 $i$ 天的股票买入价为每股 $AP_i$，第 $i$ 天的股票卖出价为每股 $BP_i$（数据保证对于每个 $i$，都有 $AP_i \geqslant BP_i$）。但股票买卖需要遵循以下的规则：

1. 第 $i$ 天的一次买入至多只能购买 $AS_i$ 股，一次卖出至多只能卖出 $BS_i$ 股。
2. 两次交易（某一天的买入或者卖出均算是一次交易）之间，至少要间隔 $W$ 天，也就是说如果在第 $i$ 天发生了交易，那么从第 $i+1$ 天到第 $i+W$ 天，均不能发生交易。
3. 在任何时间，一个人的手里的股票数不能超过 $MaxP_c$

在第 $1$ 天之前，假设这位同学有无限资金，但是没有任何股票。请问：$T$ 天以后他最多可以赚到多少钱?

**数据范围：** $0 \leq W < T \leq 2000$，$1 \leq \mathrm{MaxP} \leq 2000$，$1 \leq BP_i \leq AP_i \leq 1000$，$1 \leq AS_i, BS_i \leq \mathrm{MaxP}$。

### 习题 8 购票(洛谷P2305,NOI2014)

有一棵 $n$ 个结点的树，$S$ 市编号为 $1$。对除 $S$ 市之外的城市 $v$，给出其父亲城市 $f_v$ 及连接道路的长度 $s_v$。从城市 $v$ 前往 $S$ 市时，可以选择一个祖先城市 $a$，支付购票费用并乘车到达 $a$；再选择 $a$ 的祖先 $b$，支付费用并到达 $b$，依此类推，直至 $S$ 市。

对于任意一个城市 $v$，给出一个交通工具的距离限制 $l_v$。对于城市 $v$ 的祖先 $A$，只有当它们之间所有道路的总长度不超过 $l_v$ 时，从城市 $v$ 才可以通过一次购票到达城市 $A$，否则不能通过一次购票到达。对于每个城市 $v$，还会给出两个非负整数 $p_v, q_v$ 作为票价参数。

若城市 $v$ 到城市 $A$ 的道路总长度为 $d$，则从城市 $v$ 到城市 $A$ 的票价为 $dp_v+q_v$。求每个城市的同学所需的最少资金。

保证 $n\leqslant 2\times 10^5, 0\leqslant p_v\leqslant 10^6, 0\leqslant q_v\leqslant 10^{12}, 1\leqslant f_v<v, 0<s_v\leqslant l_v\leqslant 2\times 10^{11}$，且任意城市到 $S$ 市的总路程长度不超过 $2\times 10^{11}$。

### 习题 9 NOI嘉年华(洛谷P1973,NOI2011)

NOI嘉年华活动分在两个不同的地点举办。每个嘉年华可能包含很多个活动，而每个活动只能在一个嘉年华中举办。一共收到了 $n$ 个活动的举办申请，其中第 $i$ 个活动的起始时间为 $S_i$，活动的持续时间为 $T_i$，这些活动都可以安排到任意一个嘉年华的会场，也可以不安排。不能有两个活动在两个会场同时进行，同一会场内的活动可以任意进行。

另外希望通过合理的安排，使得活动相对较少的嘉年华的活动数量最大。如果第 $i$ 个活动必须举办（可以安排在两场嘉年华中的任何一个），求活动相对较少的嘉年华的活动数量的最大值。请依次输出 $i$ 从 $1$ 到 $n$ 的结果。

**数据范围：** $1\leqslant n\leqslant 200, 0\leqslant S_i\leqslant 10^9, 1\leqslant T_i\leqslant 10^9$

### 习题 10 柠檬(洛谷P5504江苏省队选拔2011)

将一个数列分成若干段，从每一段中选定一个数 $s$，假设这一段中这个数有 $t$ 个，则这一段价值为 $st^2$。数列的总价值为每一段的价值和。最大化数列的总价值。

保证 $1\leqslant n\leqslant 10^6$，数列中的数均为不超过 $10000$ 的正整数。

### 习题 11 诗人小G(洛谷P1912,NOI2009)

一首诗包含了 $N$ 个句子，对于一些连续的短句，可以将它们用空格隔开并放在一行中，一行中可以放的句子数目是没有限制的。小 $G$ 给每首诗定义了一个行标准长度（行的长度为一行中符号的总个数），他希望排版后每行的长度都和行标准长度相差不远。显然排版时，不应改变原有的句子顺序，并且小 $G$ 不允许把一个句子分在两行或者更多的行内。

在满足上面两个条件的情况下，小 $G$ 对于排版中的每行定义了一个不协调度，为这行的实际长度与行标准长度差值绝对值的 $P$ 次方，而一个排版的不协调度为所有行不协调度的总和。

小 $G$ 最近又作了几首诗，现在请你对这些首诗进行排版，使得排版后的诗尽量协调（即不协调度尽量小），并把排版的结果告诉他。

**数据范围：** $1\leqslant n\leqslant 10^5, 1\leqslant 3\times 10^6\leqslant 10^9, 0\leqslant P\leqslant 10$。

### 习题 12 物品装箱(洛谷P3195,湖南省队选拔2008)

教授打包行李，使用压缩器进行压缩，其可以将任意物品变成一维，再放到一种特殊的一维容器中。有编号为 $1,\cdots,n$ 的 $n$ 件物品，第 $i$ 件物品经过压缩后的一维长度为 $C_i$。为了方便整理，教授要求：在一个一维容器中的物品编号是连续的；同时，如果一个一维容器中有多个物品，那么两件物品之间要加入一个单位长度的填充物。

举例来说，如果将第 $i$ 件物品到第 $j$ 个物品放到一个容器中，那么容器的长度将为 $x=j-i+\sum_{k=i}^{j}C_k$。如果容器长度为 $x$，其制作费用为 $(x-L)^2($ 其中 $L$ 是一个常量。教授不关心容器的数目，他可以制作出任意长度的容器，甚至超过 $L$。但他希望所有容器的制作总费用最小。

**数据范围：** $1 \leqslant n \leqslant 5 \times 10^4, 1 \leqslant L \leqslant 10^7, 1 \leqslant C_i \leqslant 10^7$.

### 习题 13 回家路线(洛谷P5468,NOI2019)

某铁路系统中有 $n$ 个站点，从 $1$ 到 $n$ 编号。小毛准备从 $1$ 号站点出发，乘坐列车回到他家所在的 $n$ 号站点。他查询了能够乘坐的列车，这些列车共 $m$ 班，从 $1$ 到 $m$ 编号。小毛将在 $0$ 时刻到达 $1$ 号站点。对于 $i$ 号列车，它将在时刻 $p_i$ 从站点 $x_i$ 出发，在时刻 $q_i$ 直达站点 $y_i$，小毛只能在时刻 $p$：上 $i$ 号列车，也只能在时刻 $p_iq_i$ 下 $i$ 号列车。

小毛可以通过多次换乘到达 $n$ 号站点。一次换乘是指对于两班列车，假设分别为 $u$ 号与 $v$ 号列车，若 $y_u = x_v$ 并且 $q_n \leqslant p_v$，那么小毛可以乘坐完 $u$ 号列车后在 $y_u$ 号站点等待 $p_v{}^{-}q_u$ 个时刻，并在时刻 $p_v$ 乘坐 v 号列车。

小毛在站点等待时将增加疲劳值，对于一次 $t(t \geqslant 0)$ 个时刻的等待，疲劳值将增加 $At^2 + Bt + C$，其中 $A$，$B$，$C$ 是给定的常数。注意：小毛登上第一班列车前，即从 $0$ 时刻起停留在 $1$ 号站点的那些时刻也算作一次等待。若小毛最终在时刻 $z$ 到达 $n$ 号站点，则疲劳值将再增加 $z$。

举例来说，若小毛共乘坐了 $k$ 班列车，依次乘坐的列车编号可用序列 $s_1, s_2, \cdots, s_k$ 表示。该方案被称作一条可行的回家路线，当且仅当它满足下列两个条件：

1. $x_{s1} = 1, y_{sk} = n$
2. 对于所有 $j(1 \leqslant j < k)$，满足 $y_{sj} = x_{s_{j+1}}$ 且 $q_{sj} \leqslant p_{s_{j+1}}$。

对于该回家路线，小毛得到的疲劳值将为：

$$q_{s_k} + (A \times p_{s_1}^2 + B \times p_{s_1} + C) + \sum_{j=1}^{k-1} (A(p_{s_{j+1}} - q_{s_j})^2 + B(p_{s_{j+1}} - q_{s_j}) + C)$$

小毛想让自己的疲劳值尽量小，请帮他求出所有可行的回家路线中，能得到的最小疲劳值。题目保证至少存在一条可行的回家路线。

保证 $2 \leqslant n \leqslant 10^5, 1 \leqslant m \leqslant 2 \times 10^5, 0 \leqslant A \leqslant 10, 0 \leqslant B, C \leqslant 10^6, 0 \leqslant p_i < q_i \leqslant 10^3 \leqslant C$，

### 习题 14 编程较量(ARC 066F)

对一个序列 $A$ 进行 $Q$ 次询问，每次询问修改一个元素，询问后复原。每次询问需要选择若干元素，记 $B_i$ 为是否选了第 $i$ 个，价值为连续区间数减去选择的元素权值和，即

$$\sum_{l=1}^{n} \sum_{r=l}^{n} \prod_{k=l}^{r} B_k - \sum_{i=1}^{n} A_i B_i$$

使价值最大化。保证 $|A|, Q \leq 3 \times 10^5$。

# 第5部分：数学专题选讲

## 第19章 进阶数论

自文明产生以来，人类就一直和整数打交道了。对于整数，可以进行加、减、乘、除这样的四则运算。整数之间的加、减、乘运算，其结果还是整数；但是遇到了除法，就可能产生余数。人们在研究整数的过程中，发现了越来越多数字的性质，如整除性，并在此基础上出现了质数与合数的概念。进一步，数学家探索出了更多整数的性质，并将其发展成为数学的一个重要分支数论。数论有许多有趣和复杂的数学规律，被称为“数学皇冠”。

在《基础篇》的“整除理论”章中已经研究过质数与合数、公约数与公倍数等概念。本章将更加深入地研究整数的性质，包括同余、逆元、同余方程、线性筛和**欧拉函数**等。本章内容会有些抽象，读者需要花费一些精力理解这些概念。本章思维导图如图 19-1 所示。

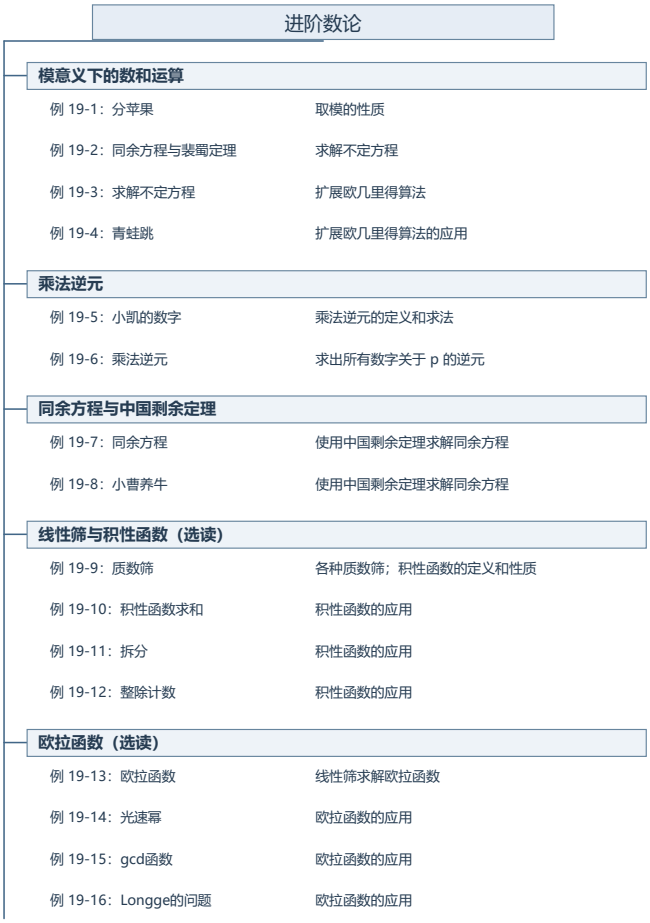



图19-1 本章思维导图

## 19.1 模意义下的数和运算

在《基础篇》中出现过这样一道题：12 个苹果，恰好平分给 $x$ 个小朋友（每个小朋友分到的苹果完整且数量相同），问 $x$ 能取到哪些值。答案是 1，2，3，4，6，12。接下来就可能有这样一个问题：如果真实的人数不是这些数中的某一个，并且每人分到的苹果依然需要完整且数量相同，会出现什么情况?

### 例 19-1 分苹果

一共有 $12$ 个苹果，平均分给 $5$ 个小朋友，最后会剩下几个苹果?

通过简单的计算就可以得出，每个人最多能分到 $2$ 个苹果，那么所有人分到了一共 $10$ 个苹果，所以会剩下 $2$ 个苹果。使用小学学过的带余除法来表示，就是： $12 \div 5 = 2 \cdots 2.$

这里，前面一个 $2$ 被称作“商”，代表每个小朋友能分到 $2$ 个苹果，在 C++ 中可以用 $12/5$ 来计算（本章在没有特殊说明的情况下，除法指的都是整除）；后面一个 $2$ 被称作“余数”，代表最后还剩下 $2$ 个苹果，在 C++ 中可以用 $12\%5$ 来计算，在其他语言中，也有用 12mod5 来表示的。在这个算式中的“%”叫作“模”，也就是“mod”的音译。

基于以上内容，可以给出模的具体定义。

定义 $1$：对于整数 $a$ 和 $b$，满足 $b > 0$，则存在唯一的整数 $q$ 和 $r$，满足 $a = bq + r$，其中 $0 \leqslant r < b$ 其中称 $q$ 为商、$r$ 为余数。余数用 $a$ mod $b$ 或者（$a\%b$）表示。

介绍了模的定义，接着就要掌握带模算式的运算。事实上，这些运算和普通的运算规则基本上是一样的。

$$(1)\ (a+b) \bmod M = (a \bmod M + b \bmod M) \bmod M$$

假如先把 $a$ 个苹果分给一些小朋友，剩下 $2$ 个苹果；再把 $b$ 个苹果分给这些小朋友，剩下 $3$ 个苹果。那么显然把 $a+b$ 个苹果分给这些小朋友后剩下的苹果数，一定和把 $5$ 个苹果分给小朋友后剩下的数量是一样的。

$$(2)\ (a-b) \bmod M = (a \bmod M - b \bmod M) \bmod M$$

加法和减法本质上是一样的，只不过即使 $a \geqslant b$，也不能保证 $a$ mod $M \geqslant$bmod $M$，假如 $12$ 个苹果分给 $5$ 个小朋友剩下 $2$ 个，而分 $15$ 个苹果就不剩了。在 C++ 中，当 $a\%M - b\%M$ 小于 $0$ 时，（$a\%M - b\%M$）$\%M$ 的结果是负数，那么就可以写成 $(a\%M - b\%M + M)\ \%M$ 以保证结果是非负数。

$$(3)\ (a \times b) \bmod M = (a \bmod M \times b \bmod M) \bmod M$$

这个运算可以从抽象一点的角度来证明。首先有 $x/M \times M + (x \bmod M) = x_0$ 设 $a = x_a M + r_a, b = x_b M + r_b$，则 $a \times b = (x_a x_b M + r_a x_b + r_b x_a + r_a r_b / M)M + r_a r_b \bmod M$，那么显然有 $a \times b \bmod M = r_a \times r_b \bmod M$。

加法、减法和乘法都没有什么特殊的，但是除法就有些不一样了。读者可能会根据上面的运算规则写出“ $(a/b) \bmod M = (a \bmod M / b \bmod M) \bmod M$” 这样的式子，但是这个等式大多数情况下是不成立的。

例如， $a=12,b=3,M=6$，那么 $(a/b) \bmod M=4$，但是（$a \bmod M/b \bmod M$） $\bmod M=0$，显然是不相等的。模意义下的除法究竟该怎么做，将会在“**乘法逆元**”节中继续学习。现在只需要记住加法、减法和乘法即可。

从模的定义中可知，所有数modb的结果只有 $b$ 种可能，分别是 $0,1,2,\cdots,b-1$ （因为 $0 \leqslant r<b$）。根据这个结果，可以把所有整数分成 $b$ 种类型。如果两个数modb的结果相同，就把它们分在同一类中。可以将这种“结果相同”称作同余，也就是“同一个余数”的简称。

定义 2：若两数 $x$ 和 $y$ 除以 $b$ 的余数相等，则称 $x$ 和 $y$ 模 $b$ 同余，记作 $x \equiv y(\bmod b)$

依据这种分类方法，可以得到同余的以下几个最基础的性质。

1. 反身性： $x \equiv x(\bmod M)$。
2. 对称性：若 $x \equiv y(\bmod M)$，则 $y \equiv x(\bmod M)$
3. 传递性：若 $x \equiv y(\bmod M), y \equiv z(\bmod M)$，则 $x \equiv z(\bmod M)$。

根据上述取模运算的计算方法，同余还有以下性质：

4. 同加性：若 $x \equiv y(\bmod M)$，则 $x+z \equiv y+z(\bmod M)$。
5. 同乘性：若 $x \equiv y(\bmod M)$，则 $x \times z \equiv y \times z(\bmod M)$。
6. 同幂性：若 $x \equiv y(\bmod M)$，则 $x^n \equiv y^n(\bmod M)$。

同余还有以下这个非常重要的性质：

7. $x \equiv y(\bmod b) \Leftrightarrow b \mid (x-y)$

还是以分苹果的例子解释这个性质。12 个苹果分给 5 人，最后剩下 2 个；22 个苹果分给 5 人，依然剩下 2 个。也就是 $22 \equiv 12(\bmod 5)$。可以发现，22 个苹果比 12 个苹果多出来的那些苹果，一定平分到所有 5 个小朋友手上，每个小朋友多分到 2 个苹果，一共多了 $5 \times 2=10$ 个苹果；如果每个小朋友多分到 $x$ 个苹果，那么总共就多了 5x个苹果，并且 5 一定是 5x的一个约数，也可表示为 51（$22-12$）。类似地，也可证明右式可以推导出左式。

### 例 19-2 同余方程与裴蜀定理

大家从小学就开始接触各种各样的方程，如一元一次方程 $2x-5=0$，解为 $x=2.5$；一元二次方程 $x^2-4x+3=0$，解为 $x_1=1,x_2=3$；二元一次方程 $3x+4y=7,x-y=0$，解为 $x=y=1\cdots$。

但是在初等数论中，通常只需在整数范围内考虑所有问题。例如，现有 5 块金牌，要把它们平均分给两人，那么每人能分到多少块金牌?从正常的角度来讲，他们都不希望分到“半块”金牌，所以实际上“平均分”这个操作不能实现。 $x=2.5$ 并不是这个问题的一个解。

基于这种需求，定义一种新的方程形式：不定方程。

定义 3：不定方程又称丢番图方程，是一种通常涉及两个或更多未知数的多项式方程，求解仅仅在整数范围内进行①。

在本节中，只需考虑解决最简单的不定方程，即形如 $ax+by=c$ 的二元一次不定方程。并且在信息竞赛（OI）中，只需要解决 $a,b,c$ 都是整数的情况。虽然这是一个之前没有遇到过的数学问题，但也不用担心，法国数学家艾蒂安·裴蜀已经把路铺好了。

定理 1（裴蜀定理）：对于整数 $a$ 和 $b$，设它们的最大公约数 $\gcd(a,b)=d$，一定存在一组整数 $(x,y)$ 使得 $ax+by=d$.

裴蜀定理的证明可以用构造法来解决，稍后就将介绍。但是这里需要考虑，如何通过裴蜀定理来解决一般的 $ax+by=c$ 这类不定方程问题。

定理 2：对于整数 $a,b,c,ax+by=c$ 有整数解当且仅当 $\gcd(a,b)\mid_{c_0}$

- 充分性：如果 $c$ 是 $\gcd(a,b)$ 的倍数，那么可以先通过裴蜀定理找到 $ax+by=\gcd(a,b)$ 的一组解 $(x,y)$。然后将 $(x,y)$ 同时乘以 $\frac{c}{\gcd(a,b)}$，就得到了 $ax+by=c$ 的一组解。
- 必要性：如果不定方程有解，因为α是 $\gcd(a,b)$ 的倍数，$b$ 是 $\gcd(a,b)$ 的倍数，并且 $x$ 和 $y$ 都是整数，所以 $ax+by$，也就是 $c$，也一定是 $\gcd(a,b)$ 的倍数。

> **注：** 1. 实际上，中文语境下“不定方程”有两种含义，一种是“Indeterminateequation”，即字面意义上的“不定方程”，表示至少有 2 个解。但本节中取“丢番图方程”，即“Diophantineequation”的含义，“丢番图方程”可以没有整数解。

通过定理 2，就只需要考虑求出 $ax+by=\gcd(a,b)=d$ 的一组解即可。而扩展欧几里得算法，通常被称为exgcd，就是用来解决这个问题的。扩展欧几里得算法也就是在欧几里得算法上进行一些扩展。用欧几里得算法求出 $a$ 和 $b$ 最大公约数的步骤如下：

1. 当 $b=0$ 时，$a$，$b$ 的最大公约数就是 $a$
2. 求出 $b,a \bmod b$ 的最大公约数，设为 $g_0$，
3. 令 $g=g_0,g$ 就是 $a,b$ 的最大公约数。

那么可以仿照这个过程，来考虑怎样求解这个不定方程步骤如下：

1. 当 $b=0$ 时， $ax+by=d$ 的解是 $x=\frac{d}{a},y$ 任意。
2. 求出 $bx+(a \bmod b)y=d$ 的一组解，设为 $(x_0,y_0)$。
3. 令 $(x,y)=f((x_0,y_0)),(x,y)$ 就是 $ax+by=d$ 的一组解。其中 $f((a,b))$ 指的是将一对数 $\cdot^{(3}_{(a,b)}$ 做一种变换。

现在的问题就在于，如何从 $(x_0,y_0)$ 生成一个新的 $(x,y)$

已知 $bx_0+(a \bmod b)y_0=d$，那么可以把 $a \bmod b$ 用 $a-\lfloor\frac{a}{b}\rfloor\times b$ 来表示，得到：

$$bx_0+\left(a-\lfloor\frac{a}{b}\rfloor\times b\right)y_0=d$$

把α和 $b$ 分别提出来，就得到：

$$ay_0+b\left(x_0-\left\lfloor\frac{a}{b}\right\rfloor y_0\right)=d$$

这时候可以发现，只要令 $(x,y)=(y_0,x_0-\left\lfloor\frac{a}{b}\right\rfloor y_0)$，就可以解决这个问题了。通过这种递归构造，也就证明了裴蜀定理。

具体实现中，虽然从数学上讲，最后的迭代中 $y$ 可以取任意值，但如果 $y$ 取值很大，那么运算中途可能超出int或者longlong的范围。但可以通过归纳法证明，如果在最后的迭代中取 $y=0$，那么对于其他的迭代步骤，求解 $ax+by=\gcd(a,b)$ 得到的一组解 $(x,y)$ 一定满足 $|x|\leqslant\left|\frac{b}{2\gcd(a,b)}\right|,|y|\leqslant\left|\frac{a}{2\gcd(a,b)}\right|$（这里只考虑 $a>b$ 的情况）。

$a\leqslant b$ 时迭代一次就会变成 $a>b$ 的情况，且在迭代中途是不会出现 $a\leqslant b$ 的）。

首先，当 $a \bmod b=0$ 时，由于 $a>b$ 且 $a$ 是 $b$ 的倍数，有 $a\geq 2b$，并且 $\gcd(a,b)=b$。迭代得到的解满足：

$$x=y_0=0<\left|\frac{b}{2\gcd(a,b)}\right|,$$

$$y=x_0-\left\lfloor\frac{a}{b}\right\rfloor y_0=1\leq\left|\frac{a}{2\gcd(a,b)}\right|.$$

设递归求得 $bx+(a \bmod b)y=\gcd(a,b)$ 的解为 $(x_0,y_0)$，并满足相应的绝对值上界。则回代得到 $x=y_0$，$y=x_0-\lfloor a/b\rfloor y_0$，有：

$$①\ |x|=|y_0|\leqslant\left|\frac{b}{2\gcd(b,a \bmod b)}\right|=\left|\frac{b}{2\gcd(a,b)}\right|$$

2. 由三角不等式以及 $x_0$、$y_0$ 的绝对值上界可得：

$$|x_0|+\left|\left\lfloor\frac{a}{b}\right\rfloor y_0\right|\leq\frac{\lfloor a/b\rfloor b+(a \bmod b)}{2\gcd(a,b)}=\frac{a}{2\gcd(a,b)}.$$

又可以发现，设 $ax+by=d=k\gcd(a,b)$，那么每次迭代得到的解都是在求解 $ax+by=\gcd(a,b)$ 时得到的解的 $k$ 倍，所以不管 $d$ 取什么值，在最后的迭代中取 $x=k,y=0$ 同样是没有问题的。

### 例 19-3 求解不定方程

求 $16x+6y=\gcd(16,6)=2$ 的一组整数解 $(x,y)$，并求出这个方程的所有整数解。

**分析**：根据 例 19-2 的分析，可以写出扩展欧几里得算法的程序，代码如下：

```
void exgcd(int a, int b, int &x, int &y) { // 求解 ax+by=gcd (a, b)
    if (b == 0) {
        x = 1, y = 0;
        // 此时gcd (a, b) =gcd (a, 0) = a, a*1+0 * 0=a
        return;
    }
    exgcd(b, a % b, y, x);
    // 求解bx0+ (a%b) y0=gcd (b, a%b)
    // 执行完这个语句后暂时x=y0,y=x0
    y -= a / b * x;
    // y=x0-a/b*y0
}
```

代入题目中的数据，模拟扩展欧几里得算法的递归过程，如图 19-2 所示。

扩展欧几里得算法的递归回代过程：

1. $2x_0+0y_0=2$，取 $(x_0,y_0)=(1,0)$。
2. $4x_1+2y_1=2$，得到 $(x_1,y_1)=(0,1)$。

3. $6x_2+4y_2=2$，得到 $(x_2,y_2)=(1,-1)$。
4. $16x_3+6y_3=2$，得到 $(x_3,y_3)=(-1,3)$。

由此求出 $16x+6y=2$ 的一组解 $x=-1,y=3$。

现在已经求出 $ax+by=c$ 的一组解了，求出所有解也就不难了。假设这个不定方程还有一组解 $(x',y')$，则又有 $ax'+by'=c$，把这个方程与原来的不定方程相减，得到 $a(x-x')+b(y-y')=0$。也就是说，设 $\Delta x=x'-x,\Delta y=y'-y$，这个不定方程的任何一个解与求出的特解的差 $(\Delta x,\Delta y)$ 一定满足 $a\Delta x+b\Delta y=0$。

同时，对于不定方程 $a\Delta x+b\Delta y=0$ 的任何一组解 $(\Delta x,\Delta y),a(x+\Delta x)+b(y+\Delta y)=(ax+by)+(a\Delta x+b\Delta y)=c+0=c,(x+\Delta x,y+\Delta y)$ 也一定是原不定方程的一组解。所以说，只需要求出 $a\Delta x+b\Delta y=0$ 的所有解，把这些解加上求出的原不定方程的特解，就可以得到原不定方程的所有解。

首先求出 $ax+by=0$ 的一组绝对值最小的非零整数解 $(x_{\min},y_{\min})$。因为 $|ax|=|by|$，这个共同的正值必须同时是 $a$ 和 $b$ 的倍数；取它为 $\operatorname{lcm}(a,b)$，即可得到这样的解。

要让 $|x|$ 尽量小，就要让 $|ax|=|by|$ 尽量小，即取 $a$、$b$ 的最小公倍数。因此可以取 $(x_{\min},y_{\min})=(b/\gcd(a,b),-a/\gcd(a,b))$。这组解的整数倍仍然是 $ax+by=0$ 的解。是否还存在不属于这组解整数倍的其他整数解呢？

如果存在，$(x'-x_{\min},y'-y_{\min}),(x'-2x_{\min},y'-2y_{\min})$，…也是 $ax+by=0$ 的解。

设 $k=\left\lfloor\frac{x'}{x_{\min}}\right\rfloor$，那么 $(x'-kx_{\min},y'-ky_{\min})=(x' \bmod x_{\min},y'-ky_{\min})$ 也是 $ax+by=0$ 的解，但是 $0<x' \bmod x_{\min}<x_{\min}$，与 $(x_{\min},y_{\min})$ 是最小非零解矛盾，所以不是 $(x_{\min},y_{\min})$ 的整数倍数的解是不可能存在的。

于是，就可以求出形如 $ax+by=c$ 的方程的所有解了。对于本题的不定方程来说，$ax+by=0$ 最小的一组非零解是 $x'=3,y'=-8$。所以 $16x+6y=2$ 的所有解就是 $x=-1+3k,y=3-8k$，其中 $k$ 是整数。

若想要求出 $16x+6y=6$ 的解呢？6 是 2 的 3 倍，通过扩展欧几里得算法求出的特解就是求出 $16x+6y=2$ 的特解的 3 倍，也就是 $x=-3,y=9$，同时 $16x+6y=0$ 的解是不变的，因此最后的解为 $x=-3+3k,y=9-8k$，其中 $k$ 是整数。

作为一本算法竞赛参考书，接下来将会介绍扩展欧几里得算法在竞赛中的一个应用。

### 例 19-4 青蛙跳(洛谷P1516)

有两只青蛙，分别叫作青蛙 $A$ 和青蛙 $B$。纬度线上东经 $0$。处为原点，由东往西为正方向，单位长度 $1$ 米，纬度线总长 $L$ 米，这样就得到了一条首尾相接的数轴。设青蛙 $A$ 的出发点坐标是 $x$. 青蛙 $B$ 的出发点坐标是 $y$。青蛙 $A$ 一次能向正方向跳 $m$ 米，青蛙 $B$ 一次能向正方向跳 $n$ 米，两只青蛙跳一次所花费的时间相同。请问它们跳了几次以后才会碰面。

**数据范围：** $1\leqslant x\neq y\leqslant 2\times 10^9,1\leqslant m,n\leqslant 2\times 10^9,1\leqslant L\leqslant 2.1\times 10^9.a$

**分析：** 青蛙 $A$ 跳 $k$ 步后的坐标为 $(x+km) \bmod L$，青蛙 $B$ 的坐标为 $(y+kn) \bmod L$。因此要求同余方程 $x+km\equiv y+kn\pmod L$ 的最小正整数解 $k$。改写为 $k(m-n)+zL=y-x$，即可使用扩展欧几里得算法求解。代码如下：

```
#include <cstdio>
#include <algorithm>
using namespace std;
long long x, y, m, n, L, d, k, z;
int f = 1;
long long exgcd(long long a, long long b, long long &x, long long &y) {
    if (b == 0) {
        x = 1, y = 0;
        return a;
    }
    long long d = exgcd(b, a % b, y, x);
    y -= a / b * x;
    return d;
}
int main() {
    scanf("%lld%lld%lld%lld%lld", &x, &y, &m, &n, &L);
    if (m < n) { // 处理m-n 是负数的情况
        swap(n, m);
        f = -1;
    }
    d = exgcd(m - n, L, k, z); // 求解 k* (m-n) +zL=gcd (m-n,L)
    if ((y - x) % d != 0) {    // 判断是否无解
        puts("Impossible ");
        return 0;
    }
    k = k * f * (y - x) / d;               // 求解k*(m-n)+zL=y-x
    k = (k % (L / d) + (L / d)) % (L / d); // 求出最小的正整数 k
    printf("%lld\n", k);
    return 0;
}
```

## 19.2 乘法逆元

### 例 19-5 小凯的数字(洛谷 P4942,ByVoldermod)

小凯将从 $l$ 到 $r$ 的整数顺次连接为一串数字。例如，$l=2,r=5$ 时得到 `2345`；$l=8,r=12$ 时得到 `89101112`。求这个大整数除以 $9$ 的余数。例如 $2345 \bmod 9=5$。数据范围：$0<l\leq r\leq 10^{12}$。

**分析：**先观察题目中给出的例子，2345 可以分解为 $2\times10^3+3\times10^2+4\times10^1+5\times10^0$，又可以发现，对于任意的 $x\geqslant0$，都有 $10^x \bmod 9=1$，那么根据模运算的性质，2345mod9 的结果其实和（$2+3+4+5$）mod9 的结果是一样的。

也就是说，对于这道题，只需要求出 $[l+(l+1)+(l+2)+\cdots+$（$r-1$）$+r]$mod9 的结果就可以了。根据等差数列求和公式，要求的就是 $\frac{(l+r)(r-l+1)}{2} \bmod 9$ 的结果。

分子上的（$l+r$）（$r-l+1$）可以直接计算，但是，分母该怎么表示呢？现在已经知道了整数在模意义下该怎么表示，那么问题就自然而然来到：分数（这里指有理数）在模意义下该怎么表示。也就是说，在modp的意义下，能不能表示出 $\frac{1}{a}$ 呢（其中 $a$ 是整数）？

在实数意义下，对于任意 $a\neq0$，都有 $a\times\frac{1}{a}=1$。所以在模意义下，也希望 $\frac{1}{a}$ 满足同样的性质：$a\times\frac{1}{a}\equiv1(\bmod p)$，。也就是说，尝试找到一个整数 $x$，满足 $a\times x\equiv1(\bmod p)$。

定义 4：若 $a\times x\equiv1(\bmod p)$，则称 $x$ 为 $a$ 在模 $p$ 意义下的**乘法逆元**，记为 $a^{-1}(\bmod p)$

在本章中“模意义下的乘法逆元”简称为“逆元”。例如，因为 $2\times5 \bmod 9=10 \bmod 9=1$，所以 5 是 2 关于模 9 的逆元。也就是说， $\frac{1}{2}$ 在模 9 的意义下可以用 5 来表示。例如， $8\times\frac{1}{2}=4$，并且 $8\times5 \bmod 9=40 \bmod 9=4$，这两个算式的结果是一样的。

对于本题，只需要求出 $5(l+r)(r-l+1) \bmod 9$ 的值就可以了。代码如下：

```
#include <cstdio>
using namespace std;
int T;
long long l, r, ans;
int main() {
    scanf("%d", &T);
    while (T--) {
        scanf("%lld%lld", &l, &r);
        ans = ((1 + r) % 9 * (r - 1 + 1) % 9 * 5) % 9;
        printf("%lldn ", ans);
    }
    return 0;
}
```

接下来，将用两种不同的方法来说明给定一组 $(a,p)$，如何求出 $a$ 在模 $p$ 意义下的逆元。

方法 1：扩展欧几里得算法。

可以从 $a\times x\equiv1(\bmod p)$ 这个式子入手。很容易发现，它和 $\gamma,ax+py=1$ 等价（日表示存在）。而这个式子实际上就是一个关于 $(a,p)$ 的不定方程，就可以用扩展欧几里得算法来解决。并且由裴蜀定理可以发现存在 $a$ 在模 $p$ 意义下的逆元当且仅当 $\gcd(a,p)=1,a$ 和 $p$ 互质，即 $a\perp p$。代码如下。

```
int inv(int a, int p) { // 用 exgcd 求逆元
    int x, y;
    exgcd(a, p, x, y);
    // 求不定方程ax+py=1的一组解
    // exgcd的详细代码见上一节
    return x;
    // 求出的x就是a在模p意义下的乘法逆元
}
```

方法 2：费马小定理。

费马小定理：若 $p$ 是质数，且 $a$ 不是 $p$ 的倍数，则 $a^{p-1}\equiv1(\bmod p)$。

费马小定理实际上是欧拉定理的一个特殊情况，但是因为现在还没有介绍**欧拉函数**，所以这个定理的证明依然放到后面的章节，大家现在只需要记住这一定理即可。

那么，假如 $p$ 是质数就可以知道，如果 $a$ 不是 $p$ 的倍数，就有 $a\times a^{p-2}=a^{p-1}\equiv1(\bmod p)$。根据乘法逆元的定义.就可以知道 $a$ 在模 $p$ 意义下的逆元就是 $a^{p-2}$，这个就可以用快速幂来解决了。如果 $a$ 是 $p$ 的倍数，那么对于任意 $x$，ax也一定是 $p$ 的倍数，不可能在模 $p$ 的意义下等于 1，也就不存在逆元了。

```
int inv(int a, int p) { // 用费马小定理求逆元
    return pow(a, p - 2, p);
    // 求出a^(p-2) modp的值
    // 快速幂的详细代码见分治一章
}
```

不管使用什么方法，都能求出一个 $a$ 模 $p$ 意义下的逆元。但是不管想要解决什么问题都要有严谨的作风。有没有可能，$a$ 在模 $p$ 意义下有多个逆元，但是只能找出一个呢？这样的算法不就在某种意义上存在问题了吗？事实上，这个问题是不需要担心的。

定理 3：在大于或等于 0，小于 $p$ 的范围内，模 $p$ 的逆元（若存在）是唯一的。

可以用反证法来证明这个定理。

假设 3 $0 \leqslant x_1, x_2 < p, x_1 < x_2, ax_1 \equiv ax_2 \equiv 1 (\mathrm{mod} p)$。那么就有 $a(x_2 - x_1) \equiv 0 (\bmod p)$ 也就是说，$a(x_2 - x_1)$ 是 $p$ 的倍数，因为 $a$ 和 $p$ 是互质的，所以 $x_2 - x_1$ 是 $p$ 的倍数。

但是可以发现 $0 < x_2 - x_1 < p$，所以 $x_2 - x_1$ 不是 $p$ 的倍数，于是就产生了矛盾。这样，定理 3 就得到了证明。

当然，如果不限定 $0 < x_2 < p$，那么按照上面所说的，当且仅当 $x_2 - x_1$ 是 $p$ 的倍数时 $x_2$ 也就是 $a$ 在模 $p$ 意义下的逆元了。也就是说，假设 $x_1$ 是 $a$ 在模 $p$ 意义下的逆元，那么 $a$ 在模 $p$ 意义下的所有逆元就是 $x_1 + kp$，其中 $k$ 是整数。

现实中，人们肯定不满足于只求出一个数的逆元，所以来看 例 19-6：

### 例 19-6 乘法逆元(洛谷P3811)

给定 $n$ 和 $p$ 求 $1 \sim n$ 中所有整数在模 $p$ 意义下的**乘法逆元**。

$1 \leqslant n \leqslant 3 \times 10^6, n < p < 20000528, p$ 是质数。

**分析：**数据范围暗示了这道题需要一个线性时间的做法。而一般来说，线性时间的算法要么是递推，要么是倒推，所以接下来还是用两种方法来解决这道题。

方法 1：递推。假设已经知道了 $1 \tilde{} i-1$ 的逆元，现在想要求 $i$ 的逆元。

递推的初始情况：当 $i = 1$ 时 $,i^{-1} \equiv 1 (\mathrm{mod} p)$。接下来考虑 $2 \leqslant i < p$ 的情况。

首先有 $p = i \times \lfloor \frac{p}{i} \rfloor + p \bmod i$，写成同余式，就是 $i \times \lfloor \frac{p}{i} \rfloor + p \bmod i \equiv 0 (\bmod p)$。

因为 $p$ 是质数，且 $0 < p \bmod i < i$ 所以 $i$ 和 $p$ modi的逆元一定存在，在同余式两边同乘 $i^{-1}(p \bmod i)^{-1}$。得到 $(p \bmod i)^{-1} \times \left\lfloor \frac{p}{i} \right\rfloor + i^{-1} \equiv 0 (\bmod p)$

移项，得 $i^{-1} \equiv -(p \bmod i)^{-1} \times \left\lfloor \frac{p}{i} \right\rfloor (\bmod p)$。

$\left\lfloor \frac{p}{i} \right\rfloor$ 可以直接计算，并且根据假设，因为pmodi是一个小于 $i$ 的数，所以它的逆元之前已经求出了，所以 $i$ 的逆元现在也就可以计算了。并且知道 1 的逆元一定是 1，这就是递推的初始条件。代码如下：

```
#include <cstdio>
using namespace std;
int n, p;
long long inv[3000010];
int main() {
    scanf("%d%d", &n, &p);
    inv[1] = 1;
    // 递推初始化
    for (int i = 2; i <= n; i++)
        inv[i] = (p - inv[p % i] * (p / i) % p);
    // i^ (-1) = p-(p%i)^ (-1) * (p/i) %p
    for (int i = 1; i <= n; i++)
        printf("%lld\n", inv[i]);
    return 0;
}
```

方法 2：倒推。

直接倒推可能比较困难，但是因为知道 $\frac{1}{k}=\frac{(k-1)!}{k!}$，所以假如说已经知道了 1!，2!，3!，$\cdots n!$ modp的结果，并且知道了 1!，2!，3!，$\cdots$，$n!$ 的逆元，那么就可以知道 1~$n$ 这些数的逆元了，并且 $i!$modp 的结果是容易递推的。

可以用下面 4 步来解决这个问题：

1. 用循环求出 1!，2!，3!，$\cdots$，$n!$mod $p$ 的结果。
2. 用扩展欧几里得算法/快速幂求出 $n!$的逆元。
3. 根据 $\frac{1}{(k-1)!}=\frac{k}{k!}$，倒推出 $(n-1)!,(n-2)!,\cdots,1!$ 的逆元。
4. 根据 $\frac{1}{k}=\frac{(k-1)!}{k!}$，求出 1~$n$ 的逆元。

在整个过程中，只用扩展欧几里得算法/快速幂直接求得一个数，也就是 $n!$ 的逆元，所以时间复杂度为 $O(n+\log p)$。

代码如下：

```
#include <cstdio>
using namespace std;
int n, p;
long long inv[3000010], fac[3000010];
long long pow(long long x, int y, int p) {
    long long ans = 1;
    for (; y; y >>= 1) {
        if (y & 1)
            ans = ans * x % p;
        x = x * x % p;
    }
    return ans;
}
int main() {
    scanf("%d%d ", &n, &p);
    fac[0] = 1;
```

```
    for (int i = 1; i <= n; i++)
        fac[i] = fac[i - 1] * i % p;
    // 求出1!,2!,…,n!
    inv[n] = pow(fac[n], p - 2, p);
    // 求出(n!)^(-1)
    for (int i = n - 1; ~i; i--)
        inv[i] = inv[i + 1] * (i + 1) % p;
    // 求出((n-1) !^ (-1), (n-2) !^ (-1), ...,1!^ (-1))
    for (int i = 1; i <= n; i++)
        printf("%lld\n", fac[i - 1] * inv[i] % p);
    // 求出1^(-1),2^(-1),…,n^(-1)
    return 0;
}
```

用同样的思路，也可以在 $O(n+\log p)$ 的时间内求出任意 $n$ 个数 $a_1,a_2,\cdots,a_n$ 模 $p$ 意义下的逆元：

1. 用循环求出 $\prod_{i=1}^{1}a_i,\prod_{i=1}^{2}a_i,\prod_{i=1}^{3}a_i,\cdots,\prod_{i=1}^{n}a_i \bmod p$ 的结果。
2. 用扩展欧几里得算法/快速幂求出 $\prod_{i=1}^{n}a_i$ 的逆元。
3. 根据 $\frac{1}{k-1}=\frac{a_k}{k}$，倒推出 $\prod_{i=1}^{n-1}a_i,\prod_{i=1}^{n-2}a_i,\cdots,\prod_{i=1}^{1}a_i$ 的逆元。$\prod_{i=1}a_i$　　$\prod_{i=1}a_i$
4. 根据 $\dfrac{1}{a_k}=\dfrac{\prod_{i=1}^{k-1}a_i}{\prod_{i=1}^{k}a_i}$，求出 $a_1,\ldots,a_n$ 的逆元（此处除法按模意义解释）。

## 19.3 同余方程与中国剩余定理

### 例 19-7 同余方程

“有物不知其数，三三数之剩二，五五数之剩三，七七数之剩二。问物几何？”公元 5 世纪，《孙子算经》中就提出了这样一个问题。而用现代的数学语言来描述，即设这个物品有 $x$ 个，那么就可以转写成下面这一个方程组：

$$\begin{cases}x\equiv 2\pmod 3\\x\equiv 3\pmod 5\\x\equiv 2\pmod 7\end{cases}$$

“三三数之剩二”，就是说这个物品的数量一定是 3 的若干倍加上 2，也就是说这个数量除以 3 得到的余数是 2，这就得到了这个方程组的第 1 个式子。在 19.1 节中已经提到，这种包含同余号的式子，将它称之为同余方程；而由若干同余方程组成的方程组，也就是同余方程组了。而最早出自《孙子算经》的**中国剩余定理**，也就是用来求解一类一元线性同余方程组（未知数只包含一个，且未知数的次数为 1）的方法。

一般来讲，中国剩余定理可以用来求解下面这样一类同余方程组：

$$\begin{cases}x\equiv a_1\pmod{m_1}\\x\equiv a_2\pmod{m_2}\\\quad\vdots\\x\equiv a_n\pmod{m_n}\end{cases}$$

其中，$m_1,m_2,\cdots,m_n$ 两两互质，则对于任意的整数 $a_1,a_2,\cdots,a_n$，上面这个方程组有解。在本例中，$a=\{2,3,2\},m=\{3,5,7\}$。中国剩余定理同时给出了解的构造方式：

1. 设 $M=m_1\times m_2\times\cdots\times m_n=\prod_{i=1}^{n}m_i$。

$$M = m_1 \times m_2 \times m_3 = 3 \times 5 \times 7 = 105$$

2. 设 $M_i = \frac{M}{m_i}$

$$\begin{cases} M_1 = \dfrac{M}{m_1} = \dfrac{105}{3} = 35 \\ M_2 = \dfrac{M}{m_2} = \dfrac{105}{5} = 21 \\ M_3 = \dfrac{M}{m_3} = \dfrac{105}{7} = 15 \end{cases}$$

3. 设 $t_i = M_i^{-1}$ 为 $M_i$ 模 $m_i$ 的逆元（易知 $M_i$ 和 $m_i$ 是互质的，所以逆元一定存在）。

$$\begin{aligned} 2M_1 = 70 \equiv 1 \pmod 3, \quad & t_1 = 2, \\ M_2 = 21 \equiv 1 \pmod 5, \quad & t_2 = 1, \\ M_3 = 15 \equiv 1 \pmod 7, \quad & t_3 = 1. \end{aligned}$$

4. $x$ 的通解为：

$$x = a_1t_1M_1 + a_2t_2M_2 + \cdots + a_nt_nM_n + kM = kM + \sum_{i=1}^{n} a_it_iM_i, \quad k \in \mathbb{Z}$$

可以先来验证一开始的那个具体的问题。假设这个物品有 $23$ 个，那么三三数会数到 $3 \times 7 = 21$，剩下 $2$ 个；五五数会数到 $5 \times 4 = 20$，剩下 $3$ 个；七七数会数到 $7 \times 3 = 21$，剩下 $2$ 个，完全符合题设。

首先，证明通过这个方法得出的 $x$ 的通解是正确的。

1. 由于 $t_i$ 是 $M_i$ 模 $m_i$ 的逆元，所以 $a_it_iM_i \equiv a_i \pmod{m_i}$。

2. 由于 $M_j = \prod_{i \neq j} m_i$，所以对于 $i \neq j, M_j$ 一定是 $m_i$ 的倍数，那么 $a_jt_jM_j \equiv 0 \pmod{m_i}$

$$(3)\ x \equiv kM + \sum_{j=1}^{n} a_jt_jM_j \equiv 0 + a_i + \sum_{j=1, j \neq i}^{n} 0 \equiv a_i \pmod{m_i}\$。$$

接下来，证明不存在其他解。

这里可以使用反证法，假设存在另一个解 $x', x' \equiv x \pmod M$ 不成立。由于 $x \equiv a_i \pmod{m_i}$ 且 $x' \equiv a_i \pmod{m_i}$，所以 $x - x' \equiv 0 \pmod{m_i}$。

也就是说， $x - x' -$ 定是每一个 $m_i$ 的倍数，那么 $x - x'$ 也就一定是所有 $m_i$ 的LCM的倍数。但是由于 $m_i$ 是两两互质的，所以所有 $m_i$ 的 LCM就是 $M$，$x - x'$ 也就一定是 $M$ 的倍数，那么有 $x' \equiv x \pmod M$，这就和前提假设矛盾了。

所以在modM的意义下 $x$ 是这个方程组的唯一解。

最后，来看中国剩余定理的具体实现。

### 例 19-8 小曹养牛(洛谷P1495)

小曹在乡下养了一群牛。假如有 $16$ 头母牛，如果建了 $3$ 个牛圈，剩下 $1$ 头牛就没有地方安家了。如果建造了 $5$ 个牛圈，但是仍然有 $1$ 头牛没有地方去，然后如果建造了 $7$ 个牛圈，还有 $2$ 头没有地方去。请问他有几头牛。

输入 $n$ 表示条件的组数；接下来 $n$ 行，每行给出两个整数 $a_i, b_i$，表示按 $a_i$ 个牛圈分配时，剩下 $b_i$ 头牛没有去处。输出满足所有条件的最小正整数。数据满足 $1 \le n \le 10$，$0 \le b_i < a_i \le 100000$，$1 \le \prod_i a_i \le 10^{18}$，且 $a_i$ 两两互质。

**分析：**这道题中的 $a_i$ 和 $b_i$ 分别对应上述的 $m_i$ 和 $a_i$，直接用**中国剩余定理**解决即可。

```
#include <cstdio>
using namespace std;
int n;
long long x, y, MM, m[11], a[11], t[11], M[11];
long long exgcd(long long a, long long b, long long &x, long long &y) {
    if (b == 0) {
        x = 1, y = 0;
        return a;
    }
    long long d = exgcd(b, a % b, y, x);
    y -= a / b * x;
    return d;
}
int main() {
    scanf("%d", &n);
    MM = 1;
    x = 0;
    for (int i = 1; i <= n; i++) {
        scanf("%lld%lld", &m[i], &a[i]);
        MM = MM * m[i]; // M=m1 *m2 *...*mn
    }
    for (int i = 1; i <= n; i++) {
        M[i] = MM / m[i];                   // Mi=M/mi
        exgcd(M[i], m[i], t[i], y);         // ti=Mi^-1 (mod mi)
        t[i] = (m[i] + t[i] % m[i]) % m[i]; // 保证0<=ti<mi
        for (int j = 1; j <= a[i]; j++) // 代码逻辑为x= (x+a[i] *M[i] *t[i]) % MM
            x = (x + M[i] * t[i]) % MM; // 但由于直接乘可能溢出,用多次加法代替乘法
    } // 当然也可以用快速幂类似的方法实现快速乘
    printf("%lld\n", x);
    return 0;
}
```

当 $m_i$ 不两两互质时，中国剩余定理就显得有些无力了，但也有其他的方法来处理这种情况，参见 洛谷P4777 扩展中国剩余定理。有兴趣的读者可以自行查阅相关资料。

## 19.4 线性筛与积性函数（选读）

### 例 19-9 质数筛

在《基础篇》中，读者已经了解了怎么在 $O(\sqrt{n})$ 的时间内求出 $n$ 的所有因数，所以能在 $O(\sqrt{n})$ 的时间内判断一个数 $n$ 是否是一个质数，也就是判断它的因子是否只有 $1$ 和 $n$ 即可。但是，如果期望通过这样的方法来找出 $1 \sim n$ 这 $n$ 个数中的所有质数，那么时间复杂度将会是 $O(n\sqrt{n})$，很多情况下在时间上是不可承受的。

同样，在《基础篇》中介绍过时间复杂度为 $O(n \log \log n)$ 的埃氏筛，但在这里，希望回到筛法的本源，通过对筛法更深入的了解来找出一种更快的方法。

筛法，顾名思义，就是把所有不是质数的数筛去，剩下的数就是质数了。那么第 $1$ 个同时也是最重要的问题就是：什么样的数不是质数呢？

首先，可以从质数的定义入手，因子只有 $1$ 和自己的数是质数。也就是说，对于一个整数 $n$，如果存在一个整数 $i(i \neq 1, i \neq n)$，并且 $i$ 也是 $n$ 的因子，那么 $n$ 就是一个合数。同时，在这种情况下，显然 $\frac{n}{i}$ 也是一个不等于 $1$、不等于 $n$ 的整数。换句话说，对于两个大于 $1$ 的整数 $i$ 和 $j$，ixj一定不是质数，并且对于任意一个不是质数的整数 $n$，也可以找到两个大于 $1$ 的整数 $i$ 和 $j$，使得 $i \times j = n$。

由此，就可以得到一个最简单的筛法，时间复杂度为 $O(n \log n)$。

1. 普通筛：（大于 $1$ 的）整数的（大于 $1$ 的）整数倍数是合数

```
for (int i = 2; i <= n; i++) {
    if (!vis[i])
        prime[++tot] = i;
    for (int j = 2 * i; j <= n; j += i)
        vis[j] = 1;
}
```

//如果 $i$ 没有被筛去//那么 $i$ 就是一个质数//枚举 $i$ 的所有倍数 $j$//$i$ 的倍数是合数

但是，这样的方法显然会出现冗余。例如，$10$ 既是 $2$ 的 $5$ 倍，也是 $5$ 的 $2$ 倍，那么当 $i = 2$ 和 $i = 5$ 时，$10$ 这个数都会被标记为合数。于是希望找出一种更优秀的判断一个数是否是质数的方法来减少这种冗余。可以发现，一个合数不仅仅一定有一个不等于 $1$ 和本身的因子，也一定有一个不等于 $1$ 和本身的质因子。通过这种思想，就可以将原来的普通筛法转化为更优秀的，时间复杂度为 $O(n \log \log n)$ 的埃式筛。

2. 埃式筛（形式 $1$）：质数的（大于 $1$ 的）整数倍数是合数

```
for (int i = 2; i <= n; i++)
    if (!vis[i]) {                          // 如果i没有被筛去
        prime[++tot] = i;                   // 那么i就是一个质数
        for (int j = 2 * i; j <= n; j += i) // 枚举 i的所有倍数 j
            vis[j] = 1;                     // 质数i的倍数是合数
    }
```

既然质数的（大于 $1$ 的）整数倍数是合数，那么（大于 $1$ 的）整数的质数倍数也一定是合数。就像 $3$ 的 $4$ 倍 $12$ 是合数，那么 $4$ 的 $3$ 倍也一定是合数。于是可以得到埃式筛的另外一种形式。

3. 埃式筛（形式 $2$）：（大于 $1$ 的）整数的质数倍数是合数

```
for (int i = 2; i <= n; i++) {
    if (!vis[i])                                        // 如果i没有被筛去
        prime[++tot] = i;                               // 那么i就是一个质数
    for (int j = 1; j <= tot && i * prime[j] <= n; j++) // 枚举当前的所有质数 prime[j]
        vis[i * prime[j]] = 1;                          // i的 prime[j]倍是合数
}
```

形式 2 实际上是形式 1 的一种优化，因为在形式 2 中，只有当 `prime[j]` $\leqslant i$ 时，$i*$ `prime[j]` 才会被筛去。为什么这样优化是正确的呢?考虑 $prime[j] > i$ 的情况，设 $i$ 的最小质因子是 $p_i$，则 $i$ * `prime[j]` 会在 $i' = prime[j] * \frac{i}{p_i}, prime[j'] = p_i$ 的时候被筛去 $(i' \geqslant prime[j] > i \gneqq p_i = prime[j'])$ 0 最后，添加了一行代码，就可以得到最终想要的算法—时间复杂度为 $O(n)$ 的线性筛。

4. 线性筛：（大于 1 的）整数的质数倍数是合数

```
for (int i = 2; i <= n; i++) {
    if (!vis[i])                                          // 如果i没有被筛去
        prime[++tot] = i;                                 // 那么i就是一个质数
    for (int j = 1; j <= tot && i * prime[j] <= n; j++) { // 枚举当前的所有质数
        // prime[j]
        vis[i * prime[j]] = 1; // i的prime[j]倍是合数
        if (i % prime[j] == 0)
            break; // 魔法!
    }
}
```

为什么增加这一句判断，就可以结束内层循环呢？设 $n$ 的最小质因子为 $p_n$，例如 $p_9 = 3 p_{12} = 2.$。那么在线性筛中，一个合数 $n$ 会且仅会在 $i = \frac{n}{p_n}$ 的时候被筛去。

先证明充分性：合数 $n$ 会在 $i = n/p_n$ 时被筛去，其中 $p_n$ 是 $n$ 的最小质因子。因为 $n/p_n \geq p_n$，所以此时 $p_n$ 已保存在质数数组 `p` 中。如果循环在访问 $p_n$ 之前中断，就说明某个小于 $p_n$ 的质数整除了 $i$，它也会整除 $n$，这与 $p_n$ 为最小质因子矛盾。因此一定能在这次循环中筛去 $n$。

然后证明必要性：合数 $n$ 仅会在 $i=\frac{n}{p_n}$ 的时候被筛去。假设 $p'$ 是一个大于 $p_n$ 的 $n$ 的质因子，那么需要证明的是，当 $i=\frac{n}{p'}$ 的时候，在 $prime[j]$ 枚举到 $p'$ 之前，第 2 个循环就已经中断了。

这其实也很简单：因为 $p'$ 和 $p_n$ 互质，且同时是 $n$ 的质因子，所以 $p_n$ 也一定是 $\frac{n}{p'}$ 的一个质因子（否则与整数的唯一分解定理矛盾）。那么当 $i = \frac{n}{p'}, prime[j]$ 枚举到 $p_n$ 时，`prime[j]` 就整除 $i$ 了，此时第 2 个循环就会中断，而不会让 `prime[j]` 再往后枚举到 $p'$。

也就是说，对于每一个合数，它只会被 $\mathbf{vis[i * prime[j]] = 1}$；这条语句标记为合数一次。也就是说，这条语句实际上只会被执行不超过 $n$ 次，那么整个算法的时间复杂度显然就是 $O(n)$ 了。

以上，就是线性筛的基本流程。线性筛除了可以用来求解 $1 \sim n$ 的所有质数这个最基本的用途之外，也可以用来计算积性函数。首先，来了解积性函数的定义。

定义 5：定义域为正整数、值域为复数的子集的函数称为数论函数。

定义 6：对于一个数论函数 $f$，若 $\forall x, y \in \mathbb{N}_+, x, y$ 互质（记为 $x \perp y$） ，都有 $f(xy) = f(x)f(y)$，则称 $f$ 是一个积性函数。

定义 7：对于一个数论函数 $f$，若 $\forall x, y \in \mathbb{N}_+$，都有 $f(xy) = f(x)f(y)$，则称 $f$ 是一个完全积性函数。

可以看出，完全积性函数一定是积性函数，而积性函数又一定是数论函数。

在这里先给出一些常见的积性函数（自变量均为 $n$）：

1. 1 函数（完全积性）： $\mathbb{1}(n) = 1$。
2. 幂函数（完全积性） $\mathbf{id}_k(n) = n^k$。

3. 狄利克雷卷积单位元（完全积性）：$\varepsilon(n)=\begin{cases}1,n=1\\0,n\neq 1\end{cases}$

4. 因子幂和函数：$\sigma_k(n)=\sum_{i|n} i^k; k=0$ 时又称为 `d(n)` 除数函数，表示 $n$ 的因子个数。

5. **欧拉函数**：$\varphi(n)=1\sim n$ 中与 $n$ 互质的数的个数。

6. 莫比乌斯函数 $\mathbf{z}\cdot\mu(n)=\begin{cases}0,\\(-1)^k\end{cases}$ $n$ 有平方因子，$n$ 无平方因子，$k$ 为 $n$ 的质因子个数

7. 最大公因数：$\gcd_k(n)=\gcd(n,k)$。

8. `r(n)`=以原点为圆心，半径为 $n$ 的 $\frac{1}{4}$ 圆上的整点个数（不包含一侧端点）。

其中，有一些积性函数的积性性质很好证明，如 $(xy)=1=1\times 1=\mathbb{1}(x)\mathbb{1}(y)$，以及 $\mathrm{id}_k(xy)=(xy)^k=x^ky^k=\mathrm{id}_k(x)\mathrm{id}_k(y)$。

但另一些函数的积性性质可能就需要一些数学技巧来证明，如 $\frac{1}{4}$ 圆上的整点个数就与高斯质数有关，有兴趣的读者可以去查阅相关资料，但遇到以上函数的时候最好要记住这些是积性函数。

而积性函数的计算又如何与线性筛结合起来呢？接下来通过 例 19-10 来探索。

### 例 19-10 积性函数求和

给定 $n$，输出 $\sum_{i=1}^{n}\sum_{j=1}^{n}(-1)^{d(ij)}$。$1\le n\le 10^7$。

**分析：**$d(n)$ 为奇数当且仅当 $n$ 是完全平方数，因为其余因子 $i$ 和 $n/i$ 成对出现。因此，只需统计所有乘积 $ij$ 中有多少个完全平方数。将 $n$ 写成 $ab^2$，其中 $a$ 不含大于 1 的平方因子，定义 $f(n)=a$。例如 $12=3\times 2^2$，故 $f(12)=3$。于是 $ij$ 为完全平方数等价于 $f(i)=f(j)$，只需求出 $f(1),\dots,f(n)$ 并统计每种取值的数量。

接下来证明 `f(n)` 是一个积性函数，设 $x=a_xb_x^2, f(x)=a_x, y=a_yb_y^2, f(y)=a_y, x\perp y$，则 $xy=(a_xa_y)(b_xb_y)^2$，因为 $x\perp y$，所以 $a_x\perp a_y$ 那么 $a_xa_y$ 也一定不会被大于 1 的平方数整除（若能被 $z^2$ 整除，要么 $a_x$ 能被 $z^2$ 整除，要么 $a_y$ 能被 $z^2$ 整除，要么 $a_x,a_y$ 有公因子 $z$，都不满足条件），所以 $f(xy)=a_xa_y=f(x)f(y)$。

然后考虑当 ρ 是质数时 $f(p^k),k\in\mathbb{Z}$ 的取值情况。当 $k$ 为奇数时，$p^k=p\times(p^{\frac{k-1}{2}})^2$，所以 $f(p^k)=p$；当 $k$ 为偶数时 $\rho,p^k=1\times(p^{\frac{k}{2}})^2$，所以 $f(p^k)=1$。

最后考虑线性筛。当 $i=n/p_n$、`p[j]` 等于 $p_n$ 时筛到 $n$。设 $k_n$ 是 $n$ 中最小质因子 $p_n$ 的次数，记录 $K[n]=p_n^{k_n}$。若 $p_n\mid n/p_n$，则 $K[n]=K[n/p_n]\times p_n$；否则 $K[n]=p_n$。

由于 $K[n]$ 与 $n/K[n]$ 互质，若 \$K[n]\e\$ $n$ ，便可用积性函数性质计算 `f(n)`=`f(K[n])` `f(n/K[n])`；否则 $n$ 为质数幂，直接使用前面得到的 `f(p^k)` $公式。

代码如下$

```
f[1] = 1;
for (int i = 2; i <= n; i++) {
    if (!vis[i]) {
        prime[++tot] = i;
        K[i] = i; // 此时 K[p] = p
        f[i] = i; //* f (p) = p
    }
```

```
    for (int j = 1; j <= tot && i * prime[j] <= n; j++) {
        vis[i * prime[j]] = 1;
        if (i % prime[j] == 0) {                 // 如果 pn 可以整除 n/pn
            K[i * prime[j]] = K[i] * prime[j];   // K[n] = K[n/pn] * pn
            if (K[i * prime[j]] != i * prime[j]) // 如果 n 不是 pn^kn
                f[i * prime[j]] =
                    f[K[i * prime[j]]] * f[i * prime[j] / K[i * prime[j]]];
            // 积性性质f(n) = f(K[n]) *f
            //  (n/K[n])
            else
                f[i * prime[j]] = prime[j] / f[i]; // * * f (p^k) 为 1, p, 1,
            // p,...循环
            break; // 线性筛的中断
        } else {
            K[i * prime[j]] = prime[j];           // K[n] = pn
            f[i * prime[j]] = f[prime[j]] * f[i]; // 积性性质f (n) = f (K[n]) *
            // f (n/K[n])
        }
    }
}
```

而对于其他的积性函数，也只需要将上述代码注释中标记“*”的一行改成相对应的 `f(p)` 的值，标记“**”的一行改成相对应的 $f(p^k)$ 的值就可以了。上述代码可以用来计算所有的积性函数，而考虑具体的某一函数时，代码也许可以进一步简化，如本题中求解 $f$ 的代码就可以进一步简化为：

```
f[1] = 1;
for (int i = 2; i <= n; i++) {
    if (!f[i]) // f(i) = 0,说明i还没被筛去,是
        // 质数
        prime[++tot] = f[i] = i; // f(p) = p
    for (int j = 1; j <= tot && i * prime[j] <= n; j++) {
        if (f[i] % prime[j] == 0) // 如果p能整除f[i],则除去p,否则乘上p
            f[i * prime[j]] = f[i] / prime[j];
        else
            f[i * prime[j]] = f[i] * prime[j];
        if (i % prime[j] == 0)
            break;
    }
}
```

因此。假如希望求解 $f$ 在 1，2，…，$n$ 处的取值。若 $f$ 是一个积性函数，先考虑 $f(p^k),k\in\mathbb{N}_+$ 的取值，然后就可以套用上面的代码利用线性筛快速求解 $f$ 了。那么，在具体实践过程中该如何识别一个函数是不是积性函数呢？最保险的方法当然是和上面一样进行严格的数学证明，但如果时间不足或者数学技巧不足以完成这项任务，也可以先暴力地求出一些比较小的 $f$ 的值，如 `f(1)`，`f(2)`，…，`f(1000)`，然后在这个小范围内判断函数 $f$ 是否满足积性性质，如果全部满足，那么它很可能就是一个积性函数。

而关于积性函数，另一个需要注意的就是 $d(n)$ 除数函数，也就是 $n$ 的正因子个数。考虑 1~$n$ 中约数个数最多的那个数的约数个数是多少。一个朴素的想法是约数一定小于或等于 $2\sqrt{n}$，因为每一对约数一定有一个是小于或等于 $\sqrt{n}$ 的。但如果只能考虑 $2\sqrt{n}$ 这个简单的上界，遇到有些题就会有些困难，如例 19-11。

### 例 19-11 拆分(洛谷P3861)

给定整数 $n$，求将 $n$ 分解为互不相同且不小于 $2$ 的数的乘积的方案数。答案对 $998244353$ 取模。数据范围：$n \le 10^{12}$。

例如 $n=688$ 时 $688=2\times4\times86=2\times8\times43=2\times344=4\times172=8\times86=16\times43$，答案为 $6$。

**分析：**先求出 $n$ 的全部约数 $x_1,x_2,\dots,x_{d(n)}$。设 $f_{i,j}$ 为将 $x_i$ 分解为若干互不相同且不大于 $x_j$ 的因子的方案数（包括仅有因子 $x_i$ 的方案）。用 $\mathrm{pos}(v)$ 表示约数 $v$ 在数组中的位置，即 $x_{\mathrm{pos}(v)}=v$。

直接以 $v$ 为下标会使用 $O(n)$ 空间；可以按 $v\le\sqrt{n}$ 与 $v>\sqrt{n}$ 分别使用 $v$ 和 $n/v$ 作下标，将这部分空间降为 $O(\sqrt{n})$。

将空间复杂度缩小到 $O(\sqrt{n})$。

接下来考虑 $f_{i,j}$ 的求法。初始化时，$j=1$：只有 $i=1$（即 $x_i=1$）时令 $f_{i,1}=1$，其余均为 $0$。转移分为两种情况：不包含因子 $x_j$ 时，方案数为 $f_{i,j-1}$；包含 $x_j$ 时，必须有 $x_j\mid x_i$，余下部分为 $x_i/x_j$，方案数为 $f_{\mathrm{pos}(x_i/x_j),j-1}$。

将两者相加即可。最终答案为 $f_{d(n),d(n)}-1$，其中 $x_{d(n)}=n$；减去 $1$ 是为了排除只包含因子 $n$ 的情况。代码如下：

```
#include <cmath>
#include <cstdio>
#include <algorithm>
using namespace std;
const int mod = 998244353;
const int N = 1000010;
int f[6721][6721], pos1[N], pos2[N], T, tot, pos;
long long x[N], n, sqrtn, tmp;
int main() {
    scanf("%d", &T);
    while (T--) {
        scanf("%lld", &n);
        tot = 0, sqrtn = (long long)sqrt(n);
        for (long long i = 1; i * i <= n; i++) // 求出n的所有约数
            if (n % i == 0) {
                x[++tot] = i;
                if (i * i != n)
                    x[++tot] = n / i;
            }
        sort(x + 1, x + tot + 1);
        for (int i = 1; 2 * i <= tot + 1; i++) { // 记录 pos
            pos1[x[i]] = i;
            pos2[x[i]] = tot - i + 1;
        }
        for (int i = 1; i <= tot; i++) {
            f[i][1] = i == 1 ? 1 : 0; // 数组 dp 初始化
            for (int j = 2; j <= tot; j++) {
                f[i][j] = f[i][j - 1]; // f[i][j] = f[i][j-1]
                if (j > i)
                    continue;
                if (x[i] % x[j] == 0) { // f[i][j]+=f[pos[xi/xj]]
                    // [j-1]
```

```
                    tmp = x[i] / x[j];
                    pos = tmp <= sqrtn ? pos1[tmp] : pos2[n / tmp];
                    f[i][j] = (f[i][j] + f[pos][j - 1]) % mod;
                }
            }
        }
        printf("%d\n", (f[tot][tot] + mod - 1) % mod); // 注意最后去掉1*n的情况
    }
    return 0;
}
```

考虑这个算法的时间复杂度：首先要花 $O(\sqrt{n})$ 的时间求出 $n$ 的所有约数，然后要花 $O(d^2(n))$ 时间进行**动态规划**，那么最后的时间复杂度就是 $O\left(\sqrt{n}+d^2(n)\right)$。

如果认为 $d(n)$ 可能达到 $2\sqrt{n}$，那么这个时间复杂度将会是 $O(n)$，显然不能通过本题。但实际上是完全可以通过的，这就说明 $d(n)$ 实际上比一开始想象的 $2\sqrt{n}$ 要小很多。

表 19-1 给出了 $n$ 和 $d(n)$ 的对应情况。

| n ≤ | $10^1$ | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ | $10^8$ | $10^9$ |
|---|---|---|---|---|---|---|---|---|---|
| d(n) ≤ | 4 | 12 | 32 | 64 | 128 | 240 | 448 | 768 | 1344 |
| n ≤ | $10^{10}$ | $10^{11}$ | $10^{12}$ | $10^{13}$ | $10^{14}$ | $10^{15}$ | $10^{16}$ | $10^{17}$ | $10^{18}$ |
| d(n) ≤ | 2304 | 4032 | 6720 | 10752 | 17280 | 26880 | 41472 | 64512 | 103680 |

表19-1 n和d(n)的对应情况

可以看到当 $n\leqslant 10^{12}$ 时 $d(n)\leq 6720$，所以 $O(\sqrt{n}+d^2(n))$ 就能通过这道题。

事实上，`d(n)` 的级别也的确比 $\dot{O}(n^{0.5})$ 小得多，甚至对于任意大于 0 的 $\varepsilon_i$ 都有 $O(d(n))<O(n^{\varepsilon})$。具体来说，SeverinWigert证明了：对于 $\delta>0$，设 $N_\delta=\prod_{p\in Prime}p^{\frac{1}{p^\delta-1}}$.（有高等数学基础的读者可以看出，显然对于任意 $\delta>0, N_\delta$ 是收敛的），当 $n>N_\delta$ 时，$d(n)<n^{(1+\delta)\frac{\log 2}{\log\log n}}$。

当然，对于大多数读者来说，只需要通过上面的那张表记住 $d(n)$ 的大概大小就可以了（或者不确定且时间充裕的时候也可以通过搜索来找出约数个数最多的数的约数个数）。

此外，一些积性函数也经常出现在数学推导中。

### 例 19-12 整除计数

给出 $a$，求满足 $(a+b)\mid ab$ 的正整数 $b$ 的个数。 $a\leqslant 10^{12}$。

**分析：**设 $\frac{ab}{a+b}=c$，得到 $ab-ac-bc=0$，两边同时加上 $a^2$ 得到 $ab-ac-bc+a^2=a^2$，左边因式分解得 $(a-c)(a+b)=a^2\sin(a+b)$。

也就是说，$a+b$ 一定是a2 的一个大于 $a$ 的约数，所以满足条件的 $b$ 的个数 $a^2$ 就是比 $a$ 大的 $a^2$ 的约数个数，也就是 $\frac{d(a^2)-1}{2}$。用 $O(\sqrt{a})$ 的时间求出 $a$ 的质因数分解，将每个质因数的幂数乘 2 就得到了 $a^2$ 的质因数分解。

最后，根据《基础篇》中已经学习过的乘法原理，设 $a^2=p_1^{k_1}p_2^{k_2}\cdots p_m^{k_m}$，那么 $d(a^2)$ 就等于 $(k_1+1)(k_2+1)\cdots(k_m+1)$ 了（每个质因数的幂数 $k_i'$ 在某个约数中取 $0,1,\cdots,k_i$，共 $k_i+1$ 种方案），最后就可以求出答案了。

代码如下。

```
#include <cstdio>
using namespace std;
long long a, ans, tmp;
int main() {
    scanf("%lld", &a);
    ans = 1;
    for (long long i = 2; i * i <= a; i++)
        if (a % i == 0) {
            tmp = 0;
            while (a % i == 0)
                tmp++, a /= i;
            ans *= tmp * 2 + 1;
        }
    if (a != 1)
        ans *= 3LL;
    printf("%lld\n", (ans - 1) / 2);
    return 0;
}
```

## 19.5 欧拉函数（选读）

### 例 19-13 欧拉函数

在算法竞赛中，最经常出现也是最重要的积性函数就是**欧拉函数**。欧拉函数的定义 $\varphi(n)$ 表示 $1\sim n$ 中与 $n$ 互质的数的个数。例如 $1\sim12$ 中，$1$，$5$，$7$，$11$ 这 $4$ 个数与 $12$ 互质，所以 $\varphi(12)=4$。

欧拉函数是一个积性函数，但是还没有严格证明。先证明欧拉函数的积性性质。

设 $n,m$ 为互质的正整数。记 $1\sim n$ 中与 $n$ 互质的数为 $x_1,\ldots,x_{\varphi(n)}$，$1\sim m$ 中与 $m$ 互质的数为 $y_1,\ldots,y_{\varphi(m)}$，$1\sim nm$ 中与 $nm$ 互质的数为 $z_1,\ldots,z_{\varphi(nm)}$。下面证明每对 $(x_i,y_j)$ 与一个 $z_k$ 一一对应。

1. 由**中国剩余定理**，由于 $n\perp m$，存在唯一的 $1\leqslant z\leqslant nm$ 满足 $\begin{cases}z\equiv x_i\pmod n\\z\equiv y_j\pmod m\end{cases}$；并且由于 $x_i\perp n,y_j\perp m$，所以 $z\perp n,z\perp m$，也就是说 $z$ 一定和nm互质，是某一个 $z_k$。 所以，一对 $(x_i,y_j)$ 一定对应一个 $z_k,\varphi(n)\varphi(m)\geqslant\varphi(nm)$。
2. 由于 $z_k\perp nm$，所以 $z_k\perp n,z_k\perp m$，所以 $z_k\bmod n\perp n,z_k\bmod m\perp m$ 同时 $z_k\bmod n\leqslant n$ $z_k\bmod m\leqslant m$。于是 $z_k-$ 定对应着一对 $(x_i,y_j)$，即 $(z_k\bmod n,z_k\bmod m),\varphi(nm)\geqslant\varphi(n)\varphi(m)$。

综上，若 $n\perp m$，则 $\varphi(nm)=\varphi(n)\varphi(m)$，所以欧拉函数是一个积性函数。

证明了欧拉函数是一个积性函数之后，只需要考虑 $\varphi(p^k)$ 的取值就能计算所有数的欧拉函数了。可以发现，若一个数和 $p^k$ 不互质，则这个数一定是 $p$ 的倍数，那么 $1\sim p^k$ 中有 $\frac{p^k}{p}=p^{k-1}$ 个数和 $p^k$ 是不互质的，反过来和 $p^k$ 互质的数的个数就是 $\varphi(p^k)=p^k-p^{k-1}=p^k\left(1-\frac1p\right)$。

所以设 $n=p_1^{k_1}p_2^{k_2}\cdots p_m^{k_m}$，则 $\varphi(n)=n\left(1-\frac{1}{p_1}\right)\left(1-\frac{1}{p_2}\right)\cdots\left(1-\frac{1}{p_m}\right)$。

```
// o(sqrt (n)) 计算phi (n)
phi = n;
for (int i = 2; i * i <= n; i++)
    if (n % i == 0) {            // 如果i 是n的一个质因子
        phi = phi / i * (i - 1); // 乘上(1-1/p)
        while (n % i == 0)
            n /= i; // 把n 中的 i 全部筛去
    }
if (n != 1)
    phi = phi / n * (n - 1); // 不要忘了 n 有一个大于 sqrt (n) 的质因子的情况
```

同样，可以根据上面的分析写出线性筛求解欧拉函数的代码：

```
phi[1] = 1;
for (int i = 2; i <= n; i++) {
    if (!phi[i]) // phi (i) = 0,说明 i 还没被筛去
        // 是质数
        p[++tot] = i, phi[i] = i - 1; // phi (p) = p-1
    for (int j = 1; j <= tot && i * p[j] <= n; j++)
        if (i % p[j] == 0) {
            phi[i * p[j]] = phi[i] * p[j]; // phi (p^k*x) = phi (p^(k-1) *
            // X) * p
            break;
        } else
            phi[i * p[j]] = phi[i] * (p[j] - 1); // phi (p*x) = phi (x) * (p-1)
}
```

在了解了欧拉函数的基础知识之后，再回到 19.2 节“乘法逆元”中提到的“欧拉定理”。

定理 4（欧拉定理）：设 $a,n\in\mathbb{N}_+$，若 $a\perp n$，则 $a^{\varphi(n)}\equiv 1(\mathrm{mod} n)$。

当 $n$ 是一个质数的时候，可知 $\varphi(n)=n-1$，于是就可以得费马小定理。若 $p$ 为质数 $\mathcal{P}\notin a$，则 $a^{p-1}\equiv 1(\mathrm{mod}\, p)$。

例如，$\gcd(5,12)=1$，$\varphi(12)=4$，故 $5^4=625\equiv 1\ (\mathrm{mod}\ 12)$。又如 7 是质数且 $\gcd(2,7)=1$，所以 $2^{7-1}=64\equiv 1\ (\mathrm{mod}\ 7)$。接下来证明欧拉定理。

1. 考虑 1~$n$ 中与 $n$ 互质的数组成的集合 $X_n=\{x_1,x_2,\cdots,x_{\varphi(n)}\}$。

$$n=12, X_{12}=\{1,5,7,11\}$$

2. 将 $X_n$ 集合中的每一个数乘上 $a$ 再 mod $n$，得到 $X'_n=\{ax_1 \bmod n, ax_2 \bmod n,\cdots,ax_{\varphi(n)} \bmod n\}$

$$\begin{aligned} a&=5,\\ X'_{12}&=\{5 \bmod 12, 25 \bmod 12, 35 \bmod 12, 55 \bmod 12\}\\ &=\{5,1,11,7\}=X_{12}. \end{aligned}$$

3. 可以证明 $\mathcal{X}_n=\mathcal{X}'_n$ 两个集合中的元素都是 1~$n$ 中与 $n$ 互质的数，并且原来不相同的两个数乘 $a$ 后依然不可能相同（否则再乘上 $a$ 的逆元后就一样了），所以可以得出：

$$\begin{aligned}\prod_{x\in X_n} x &\equiv \prod_{x\in X_n'} x \\ &\equiv \prod_{x\in X_n} (ax) \\ &\equiv a^{\varphi(n)} \prod_{x\in X_n} x \pmod{n}.\end{aligned}$$

两边同时乘 $\prod_{x_i\in X_n} x_i$ 的逆元，于有 $1\equiv a^{\varphi(n)}(\bmod n)$

而当 $a$ 和 $n$ 不互质时，也有“扩展欧拉定理”。

定理 5（扩展欧拉定理）：设 $a,n\in\mathbb{N}_+$，有 $a^{2\varphi(n)}\equiv a^{\varphi(n)}\left(\bmod n\right)$。

例如，虽然 2 和 12 不互质，但是 $2^{2\varphi(12)}=2^8=256\equiv 4\ (\bmod\ 12)\ , 2^{\varphi(12)}=2^4=16\equiv 4\ (\bmod\ 12)$，两者依然是相同的。

现在来考虑扩展欧拉定理的证明。由唯一分解定理，设 $n=\prod_{i=1}^m p_i^{k_i}$。如果对于每个 $i$，都可以证明 $a^{2\varphi(n)}\equiv a^{\varphi(n)}(\mathrm{mod} p_i^*)$，那么 $a^{2\varphi(n)}$ 和 $a^{\varphi(n)}$ 就都是下面这个同余方程组的解：

$$\begin{cases} x\equiv a^{\varphi(n)}(\bmod M_1^{k_1}) \\ x\equiv a^{\varphi(n)}(\bmod M_2^{k_2}) \\ \quad\vdots \\ x\equiv a^{\varphi(n)}(\bmod M_m^{k_m}) \end{cases}$$

由中国剩余定理解的唯一性，只需分别证明：对每个 $i$，都有 $a^{2\varphi(n)}\equiv a^{\varphi(n)}\ (\bmod\ p_i^{k_i})$。

1. 若 $a$ 与 $p_i^{k_i}$ 互质，设 $s=\frac{n}{p_i^{k_i}}$，那么 $s$ 和 $p_i^{k_i}$ 也显然是互质的。由于 $\varphi$ 是积性函数，$\varphi(n)=\varphi(s)\varphi(p_i^{k_i})$。由欧拉定理 $a^{\varphi(n)}=a^{\varphi(s)\varphi(p_i^{k_i})}=\left(a^{\varphi(p_i^{k_i})}\right)^{\varphi(s)}=1^{\varphi(s)}=1\ (\bmod\ p_i^{k_i})$。两边同乘 $a^{\varphi(n)}$，即得到 $a^{2\varphi(n)}\equiv a^{\varphi(n)}(\mathrm{mod} p_i^{k_i})$。
2. 若 $a$ 与 $p_i^{k_i}$ 不互质，那么 $a$ 一定是 $p_i$ 的倍数， $a^{\varphi(n)}$ 一定是 $p_i^{\varphi(n)}$ 的倍数。同样设 $s=\frac{n}{p_i^{k_i}}\mathrm{c}$

首先，可以用数学归纳法证明 $p_i^{k_i-1}\geqslant k_i$。当 $k_i=1$ 时 $p_i^{k_i-1}=p_i^0=1=k_i$ 当 $k_i>1$ 时，因为 $p_i\geqslant 2, p_i^{k_i-2}\geqslant k_i-1$，所以 $p_i^{k_i-1}=p_i\times p_i^{k_i-2}\geq 2\left(k_i-1\right)\geq k_i$ 于是有：

$$\varphi(n)=\varphi(s)\varphi(p_i^{k_i})\geqslant\varphi(p_i^{k_i})=p_i^{k_i-1}(p_i-1)\geqslant p_i^{k_i-1}\geqslant k_i$$

所以 $p_i^{\varphi(n)}$ 是 $p_i^{k_i}$ 的倍数，那么 $a^{\varphi(n)}$ 也就是 $p_i^{k_i}$ 的倍数，所以可以得到 $a^{2\varphi(n)}\equiv a^{\varphi(n)}\equiv 0(\mathrm{mod}\, p_i^{k_i})$。

综上， $\forall i, a^{2\varphi(n)}\equiv a^{\varphi(n)}(\bmod p_i^{k_i})$，扩展欧拉定理成立。

将扩展欧拉定理的两边同乘 $a^{\varphi(n)}$，可以得到 $a^{3\varphi(n)}\equiv a^{2\varphi(n)}(\mathrm{mod} n)_\Omega$ 不断重复这个过程，就可以得到 $a^{\varphi(n)}\equiv a^{2\varphi(n)}\equiv\cdots\equiv a^{k\varphi(n)}\equiv\cdots(\bmod n)$。

所以对于任意的 $a$，$k$，$n$，就有 $a^k\equiv a^{k\bmod\varphi(n)+\varphi(n)}\left(\bmod\ n\right)$ ）。

### 例 19-14 光速幂(洛谷P5091)

给定 $q,a,c$，以及 $q$ 组询问，每组询问给出 $b$，求出 $a^b\bmod c$ 的值。要求时间复杂度：预处理 $O(\sqrt{c})$，每次询问在 $O(1)$ 的时间内给出答案。

**分析：**同样是要求 $a^b\bmod c$，快速幂的复杂度是每次询问 $O(\log b)$，在这里是不能接受的，但是，这里的 $a$ 和 $c$ 是一定的，就考虑从这一点下手。

首先利用扩展欧拉定理缩小指数。若 $b\geq\varphi(c)$，可将 $b$ 替换为 $b\bmod\varphi(c)+\varphi(c)$；若 $b<\varphi(c)$，则保留原指数。由于 $\varphi(c)\leq c$，处理后的指数小于 $2c$。

令 $B=\lceil\sqrt{c}\rceil$，把处理后的指数写为 $b=B\lfloor b/B\rfloor+(b \bmod B)$，于是 $a^b=(a^B)^{\lfloor b/B\rfloor}a^{b \bmod B}$。因为 $b<2c$，所以 $\lfloor b/B\rfloor<2B$，且 $b \bmod B<B$。预处理 $(a^B)^i \bmod c$（$0\le i<2B$）与 $a^j \bmod c$（$0\le j<B$），便可在 $O(1)$ 时间内回答一次询问。

最后，再来了解一些**欧拉函数**自身在算法竞赛中的应用。

### 例 19-15 gcd函数(洛谷P2568)

给定正整数 $n$，求 $1\leqslant x,y\leqslant n$ 且 $\gcd(x,y)$ 为质数的数对 $(x,y)$ 有多少对。 $n\leqslant 10^7$。

**分析：**线性筛可求不大于 $n$ 的所有质数 $p$，问题转化为对每一个 $p$ 求 $(x,y)=p$ 的个数。

设 $x=x'p$、$y=y'p$，则 $\gcd(x',y')=1$，且 $1\le x',y'\le\lfloor n/p\rfloor$。因此，对每个质数 $p$，需要统计这个范围内互质的有序数对。先考虑 $x'\le y'$：固定 $y'=k$ 时有 $\varphi(k)$ 种选择。交换两个坐标可得到另一半，而 $(1,1)$ 被重复计算了一次。因此方案数为 $2\sum_{k=1}^{\lfloor n/p\rfloor}\varphi(k)-1$。

使用线性筛同时求出**欧拉函数**，预处理**前缀和**即可。代码如下：

```
#include <cstdio>
using namespace std;
const int N = 1e7 + 10;
int n, tot, phi[N], p[N];
long long ans, sumphi[N];
int main() {
    scanf("%d", &n);
    phi[1] = sumphi[1] = 1;
    for (int i = 2; i <= n; i++) { // 线性筛
        if (!phi[i])
            p[++tot] = i, phi[i] = i - 1;
        sumphi[i] = sumphi[i - 1] + phi[i]; // 计算 phi的前缀和
        for (int j = 1; j <= tot && i * p[j] <= n; j++)
            if (i % p[j] == 0) {
                phi[i * p[j]] = phi[i] * p[j];
                break;
            } else
                phi[i * p[j]] = phi[i] * (p[j] - 1);
    }
    for (int i = 1; i <= tot; i++)
        ans += sumphi[n / p[i]] * 2 - 1; // ans += 2 * (phi (1) +phi (2) +...+
    // phi (n/p)) -1
    printf("%lld\n", ans);
    return 0;
}
```

### 例 19-16 Longge的问题(洛谷P2303,山东省队选拔2012)

求 $\sum_{i=1}^{n}\gcd(i,n)$，其中 $n\le 2^{32}$。

**分析：**注意到 `gcd(i,n)` 一定是 $n$ 的约数，而 $n$ 的约数个数很少（`d(n)` 个）。考虑枚举 $n$ 的约数（设为 $d$），再求出满足 $\gcd(i,n)=d$ 的 $i$ 有多少个。考虑 $i=i'd$，那么和 例 19-15 一样，必然有 $i'\perp\frac{n}{d}$ 且 $1\le i\le\frac{n}{d}$，这样的 $i$’ 也就是这样的 $i$ 有 $\varphi\left(\frac{n}{d}\right)$ 个。

而对于 $n$ 的每一个约数，也直接暴力求出它的**欧拉函数**即可。

注意到 $n$ 的约数 $d$ 的约数也一定是 $n$ 的约数，所以在枚举 $d$ 的约数求解 $\varphi(d)$ 时只要枚举之前已经求出的比 $d$ 小的 $n$ 的约数即可。时间复杂度为 $O(\sqrt{n}+d^2(n))$。代码如下：

```
#include <cstdio>
#include <algorithm>
using namespace std;
long long n, ans, tmp, phi, x[10000];
int tot;
int main() {
    scanf("%lld", &n);
    for (long long i = 1; i * i <= n; i++) // 找出n的所有因数,注意i*i也
        // 会爆int
        if (n % i == 0) {
            x[++tot] = i;
            if (i * i != n)
                x[++tot] = n / i;
        }
    sort(x + 1, x + tot + 1);
    for (int i = 1; i <= tot; i++) {
        phi = tmp = x[i];
        for (int j = 2; j <= i && x[j] * x[j] <= tmp; j++) // 求出 phi (d),注意x[1] 是1
            if (tmp % x[j] == 0) {
                phi = phi / x[j] * (x[j] - 1);
                while (tmp % x[j] == 0)
                    tmp /= x[j];
            }
        if (tmp != 1)
            phi = phi / tmp * (tmp - 1);
        ans += n / x[i] * phi; // gcd 为 n/d的i有 phi (d) 个
    }
    printf("%lld\n", ans);
    return 0;
}
```

## 19.6 课后习题与实验

### 习题 1 同余方程(洛谷P1082,NOIP2012提高组)

求关于 $x$ 的同余方程 $ax \equiv 1 (\bmod b)$ 的最小正整数解。

数据范围 $\cdot 2 \leqslant a, b \leqslant 2 \times 10^9$.。

### 习题 2 裴蜀定理(洛谷P4549)

给定一个包含 $n$ 个元素的整数序列 $A$，记作 $A_1, A_2, A_3, \cdots, A_n$

求另一个包含 $n$ 个元素的待定整数序列 $X$，记 $S = \sum_{i=1}^{n} A_i \times X_i$，使得 $S > 0$ 且 $S$ 尽可能小。请输出在 $S > 0$ 的前提下 $S$ 的最小值。

**数据范围：** $1 \leqslant n \leqslant 20, |A_i| \leqslant 10^5$，且 $A$ 序列不全为 $0$。

### 习题 3 荒岛求生(洛谷P2421,NOI2002)

某岛上有排列成环形的 $m$ 个山洞。这些山洞顺时针编号为 $1, 2, \cdots, m$。岛上住着 $n$ 个人，一开始依次住在山洞 $C_1, C_2, \cdots, C_n$ 中，以后每年，第 $i$ 个人会沿顺时针向前走 $P_i$ 个洞住下来。每个人 $i$ 有一个寿命值 $L_i$，即生存的年数。奇怪的是，虽然人有很多，但没有任何两个人在有生之年处在同一个山洞中。请问：至少有多少个山洞才能维持岛上的人不会抢山洞？

**数据范围：** $1 \leqslant N \leqslant 15, 1 \leqslant C_i, P_i \leqslant 100, 0 \leqslant L_i \leqslant 10^6, M \leqslant 10^6$

### 习题 4 有理数取余(洛谷P2613)

给出一个有理数 $c=\frac{a}{b}$，求 $c$ mod $P$ 的值，$P$ 是某个质数。

**数据范围：** $0 \leqslant a \leqslant 10^{10001}, 1 \leqslant b \leqslant 10^{10001}$，且 $a$，$b$ 不同时是 $P$ 的倍数。

### 习题 5 好日子(洛谷CF757B)

给定 $n$ 个正整数 $a_1, a_2, \cdots, a_n$，求一个子集 $S$，满足 $\gcd(S_1, S_2, \cdots, S_k) > 1$，同时$|S|$尽可能大。数据范围 $:1 \leqslant n, a_i \leqslant 10^5$。

### 习题 6 朋友(洛谷CF776B)

$n$ 个点，标号 $2, 3, \cdots, n+1$ 给这些点染色，要求若 $a$ 是 $b$ 的质因子，则 $a$ 和 $b$ 的颜色不同。求一种颜色数最少的方案。

**数据范围：** $n\leqslant 1000$。

### 习题 7 求质数

求 $[1, 10^7]$ 内的所有质数，内存限制 1MB。

### 习题 8 困局(洛谷P2155,山东省队选拔2008)

给定一个质数 $p, T$ 组询问。每组询问[$1$，$n!$]中与 $m!$互质的数的个数，结果对 $p$ 取模。

**数据范围：** $1 \leqslant m \leqslant n \leqslant 10^7, n < p < 10^9, T \leqslant 10$，

### 习题 9 超级最大公约数(洛谷P2152,山东省队选拔2009)

求 `gcd(a,b)`。数据范围： $0 < a, b \leqslant 10^{10000}$。

### 习题 10 最长子序列(洛谷CF632D)

给出 $n$ 个数，要求选出尽可能多的数，满足它们的最小公倍数不大于 $m_c$

**数据范围：** $1 \leqslant n, m \leqslant 10^6, 1 \leqslant a_i \leqslant 10^9$，

### 习题 11 最大公约数表(洛谷CF582A)

对一个长度为 $n$ 的数列 $a$，定义它的最大公约数表 $G$ 是一张 $n \times n$ 的二维表，其中 $G_{i,j} = \gcd(a_i, a_j)$。现在乱序给出 $G$ 中所有 $n^2$ 个数，求原数列 $a$。 数据保证有解；输出任意一种方案即可。

**数据范围：** $1 \leqslant n \leqslant 500, G_{i,i} \leqslant 10^9$ ,

### 习题 12 余数游戏(洛谷CF687B)

小 $A$ 和小 $B$ 协定一个数 $k$，以及 $n$ 个数 $c_1, c_2, \cdots, c_n$ 然后小 $A$ 想一个数 $x$，并把 $x$ mod $c_i$ $(i = 1, 2, \cdots, n)$ 告诉小 B 。请问是否对任意的 $x$，小 $B$ 都能猜出xmod $k$。

**数据范围：** $1 n , k , c _ { i } ^ { 6 } $。

### 习题 13 余数求和(洛谷P2261,重庆市队选拔2009)

给出正整数 $n$ 和 $k$，请计算：

$$G(n,k)=\sum_{i=1}^{n} k \bmod i$$

其中 $k \bmod i$ 表示 $k$ 除以 $i$ 的余数。

**数据范围：** $1 \leqslant n,k \leqslant 10^9$。

### 习题 14 仪仗队(洛谷P2158,山东省队选拔2008)

作为体育委员，小 $C$ 负责这次运动会仪仗队的训练。仪仗队是由学生组成的 $N \times N$ 的方阵，为了保证队伍在行进中整齐划一，小 $C$ 会跟在仪仗队的左后方，根据其视线所及的学生人数来判断队伍是否整齐，如图 19-3 所示。现在，小 $C$ 希望你告诉他队伍整齐时能看到的学生人数。

**数据范围：** $1 \leqslant N \leqslant 40000$。

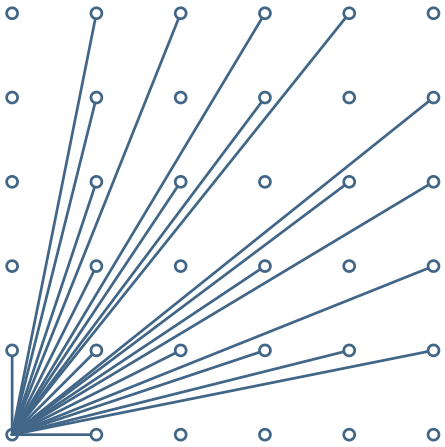


图19-3 仪仗队例图

### 习题 15 回首过去(洛谷P6583,By paulzrm)

给定正整数 $n$，求出有序整数对 $(x,y)$ 的个数，满足 $1 \leqslant x,y \leqslant n$ 且 $\frac{x}{y}$ 可以表示为十进制有限 小数。

**数据范围：** $1 \leqslant n \leqslant 10^{12}$

### 习题 16 集合的正确用法(洛谷P4139)

设 $S=2^{2^{2^{\cdots}}}$，$T$ 组询问，每次询问Smod $m_i$ 的值。

**数据范围：** $T \leqslant 10^3, m_i \leqslant 10^7$

# 第20章 组合数学与计数

在《基础篇》中已经介绍了计数原理与排列组合相关的问题，可以帮助求解在各种场合下的方案数。有一些概念需要复习一下。

1. 加法原理：做一件事，有 $n$ 类方法，第 1 类方法中有 $m_1$ 种方法，第 2 类有 $m_2$ 种……第 $n$ 类有 $m_n$ 种，则完成这件事有 $\sum_{i=1}^{n} m_i$ 种不同的方法。
2. 乘法原理：做一件事，有 $n$ 个步骤需要依次完成，第 1 步中有 $m_1$ 种方法，第 2 步有 $m_2$ 种……第 $n$ 步有 $m_n$ 种，则完成这件事有 $\prod_{i=1}^{n} m_i$ 种不同的方法。
3. 排列数：从 $n$ 个人中选出 $m$ 个人站成一排，考虑这些人的相对顺序，方案数是 $\mathrm{A}_n^m = n \cdot (n-1) \cdots (n-m+1) = \frac{n!}{(n-m)!}$
4. 组合数：从 $n$ 个人中选出 $m$ 个人，不考虑这些人的相对顺序，方案数是 $\mathrm{C}_n^m = \frac{\mathrm{A}_n^m}{m!} = \frac{n!}{m!(n-m)!}$ c

本章将会介绍更多和计数、排列组合相关知识，可以解决更多方案数量统计的问题。本章思维导图如图 20-1 所示。

## 20.1 集合与容斥原理

集合，简称集，定义为由若干（可以为零）个同类事物构成的整体。集合内的事物称为元素。例如，一个由 3 个数字 3，5，8 构成的集合可以记作{3，5，8}，3，5，8 都是该集合的元素。

集合拥有以下性质：

1. 确定性，即对于任何事物是否属于该集合都应该有一个明确的定义。例如，“大于 100 的正整数”可以是一个集合的定义，但“较大的整数”不行。
2. 无序性，即集合内的元素没有顺序区分，如{3，5}和{5，3}这两个集合是相等的。
3. 互异性，即集合内不会出现两个相等的元素。

元素 $a$ 属于集合 $A$ 记作 $a \in A$，反之则记作 $a \notin A$。

如果对于任意集合 $A$ 内的元素 $a$，都有 $a \in B$，则称 $A$ 为 $B$ 的子集，记作 $A \subseteq B$，

若集合 $A$ 和 $B$ 满足集合 $A$ 中的元素都能够在 $B$ 中找到，且集合 $B$ 中的元素都能在 $A$ 中找到，则称集合 $A$ 与 $B$ 相等，记作 $A = B$。




图20-1 本章思维导图

若一个集合中不存在任何元素，则称该集合为空集，记作 $\varnothing$。空集是所有集合的子集。

一个集合内的元素个数称为该集合的大小。$A$ 集合的大小可记作 $|A|$。

集合 $C$ 是集合 $A$ 与 $B$ 的交集，当且仅当对于任何元素 $a \in C$，有 $a \in A$ 且 $a \in B$，并且对于任何元素 $b \notin C$ 有 $b \notin A$ 或 $b \notin B$。记作 $C=A \cap B_c$

集合 $C$ 是集合 $A$ 与 $B$ 的并集，当且仅当对于任何元素 $a \in C$，有 $a \in A$ 或 $a \in B$，并且对于任何元素 $b \notin C$，有 $b \notin A$ 且 $b \notin B$。记作 $C = A \cup B$。

简单地说，交集是由所有同时属于两个集合的元素所构成的，就像两个集合相交；而并集是由所有属于其中任意一个集合的元素所构成的，就像两个集合合并。

设集合 $A \subseteq U$，称集合 $B$ 为集合 $A$ 关于集合 $U$ 的补集，当且仅当 $B$ 集合内的所有元素恰好为所有属于 $U$ 集合但不属于 $A$ 集合的元素。记作 $B = \complement_U A$。显然可以得到这样的一个式子：$|U| = |A| + |\complement_U A|$，对于任何 $A \subseteq U$ 成立。

简单地说，补集就是与一个集合互补的集合，每个元素恰好是其中两者之一。

特别地，对于形如 $\{x \mid a \leqslant x \leqslant b\}$ 的集合，可以记作 $[a, b]$。其中，如果左侧符号改为小于号，则左侧中括号改为小括号。右侧同理。

### 例 20-1 密码(洛谷P3197,湖南省队选拔2008)

有 $n$ 个房间，每个房间关着一个人。有 $m$ 种密码，每个人掌握其中一种。如果相邻房间的人的密码相同，就可以开门。求有多少种状态可以开门。答案对 $100003$ 取模。

**数据范围：** $1 \leqslant m \leqslant 10^8, 1 \leqslant n \leqslant 10^{12}$。

**分析：**由于本题的 $n$ 和 $m$ 都很大，因此可以猜测答案是一个与 $n$，$m$ 有关的表达式。然而，如果想要直接求出所有可以开门的情况数量会很困难。此时不妨从补集的角度来入手。

设全集 $U$ 为所有可能的房间状态所构成的集合，$A$ 为所有可以开门的房间状态所构成的集合，那么 $\complement_U A$ 就是所有无法开门的房间状态所构成的集合。由于 $|U| = |A| + |\complement_U A|$，因此要求 $|A|$，只需要求出 $|U|$ 与 $|\complement_U A|$ 即可。

由乘法原理，长度为 $n$、每个元素均在 $[1, m]$ 中的整数序列共有 $\underbrace{m \times \cdots \times m}_{n\text{个}} = m^n$ 个。将每个因子理解为相应房间的 $m$ 种密码选择，就得到 $|U| = m^n$。

接着考虑如来求 $|\complement_U A|$，也就是不可能开门的情况有多少种。可以按位置考虑。显然，第 1 个人的密码有 $m$ 个选择。对于接下来每一个人，只需要保证其与上一个人的密码不同即可。因此，无论上一个人掌握的密码如何，这个人都有 $m-1$ 种可以选择的密码。由乘法原理，得出 $|\complement_U A| = m \times (m-1)^{n-1}$。

也许这时有人会质疑：在求 $|\complement_U A|$ 的过程中，只保证了每个人和前一个人的密码不同，没有保证他与后一个人的密码不同。其实，在求第+1 个人的密码时，保证了他与第 $i$ 个人的密码是不同的，因此对于第 $i$ 个人，其密码一定与第 $i+1$ 个，也就是后一个不同，这也就保证了做法的正确性。在做组合题的时候，采用合理的枚举顺序，往往会给解题带来很大的便利。

最后，由于 $|A| = |U| - |\complement_U A|$，得到最终的答案就是 $m^n - m(m-1)^{n-1}$，使用快速幂计算即可，时间复杂度为 $O(\log n)$。

代码如下：

```
#include <bits/stdc++.h>
using namespace std;
int qpow(int b, long long e) {
    int ans = 1;
    while (e) {
        if (e & 1)
            ans = 1LL * ans * b % 100003;
        b = 1LL * b * b % 100003;
        e >>= 1;
    }
    return ans;
}
int main() {
    long long n, m;
    cin >> m >> n;
    int all = qpow(m, n);
```

```
    int no = 1LL * m * qpow(m - 1, n - 1) % 100003;
    int ans = (all - no) % 100003;
    if (ans < 0)
        ans += 100003;
    // 由于取模的缘故,此处相减可能导致ans小于0,故需判断
    cout << ans << endl;
}
```

### 例 20-2 Emiya家今天的饭(洛谷P5664,CSP-S2019)

有 $m$ 个食材以及 $n$ 种烹饪方法。每道菜都将使用恰好一种烹饪方法与恰好一种食材。使用烹饪方法 $i$ 和食材 $i$ 可以做出 $a_{i,j}$ 道不同的菜。问可以摆出多少桌不同的菜肴，满足：

限制 $1$：至少一道菜。

限制 $2$：每道菜的烹饪方法互不相同。

限制 $3$：设共有 $k$ 道菜，则每一种食材在该集合内被应用至多 $\lfloor \frac{k}{2} \rfloor$ 次。

答案对 $998244353$ 取模。

**数据范围：** $1 \leqslant n \leqslant 100, 1 \leqslant m \leqslant 2000, 0 \leqslant a_{i,j} < 998244353$。

**分析：**本题较难，如果暂时无法理解可以先跳过，继续学习后面的内容。

如果只有第 1 个和第 2 个限制，怎么做？在每一种烹饪方法中，选择出至多一个菜，也可以不选。因为这些选择是独立的，所以可以用乘法原理叠加。

第 $i$ 种烹饪方法总共可以得到 $\sum_{j=1}^{m} a_{i,j}$ 个物品，由于可以不选，对乘积贡献为 $1+\sum_{j=1}^{m} a_{i,j}$ 对所有 $i$ 相乘得到 $\prod_{i=1}^{n}\left(1+\sum_{j=1}^{m} a_{i,j}\right)$。

但如果所有烹饪方法都不选，就违反了限制 1，因此答案需要在此基础上减 1，即 $\prod_{i=1}^{n}\left(1+\sum_{j=1}^{m} a_{i,j}\right)-1$。将这些方案设为全集，那么违反限制 3 的方案与满足限制 3 的方案互为补集。要求解后者，只需求出前者即可。

接下来，去除违反限制 3 的方案。发现最多有一种食材出现次数大于 $\lfloor \frac{k}{2} \rfloor$，否则菜肴的总数至少为 $\left(\lfloor \frac{k}{2} \rfloor+1\right) \times 2 > k$ 矛盾。

于是，可以直接枚举哪种食材被用了大于 $\left\lfloor \frac{k}{2} \right\rfloor$ 次。考虑**动态规划**，令该食材为 $x, dp_{i,k,c}$ 表示前 $i$ 种烹饪方法，总共选出来 $k$ 个菜，其中 $c$ 个菜用食材 $x$ 枚举所有烹饪方法、菜肴数量和使用食材 $x$ 的菜的数量，尝试转移。

由于每一种烹饪方法至多选一种食材，所以要么选一种食材，要么什么都不选：

1. 如果选一种食材，转移为：如果 $j=x$，则共 $a_{i,j} \times dp_{i-1,k-1,c-1}$ 种；如果 $j \neq x$，则共 $a_{i,j} \times dp_{i-1,k-1,c}$ 种。
2. 如果什么都不选，共 $dp_{i-1,k,c}$ 种。

最后，对所有 $c$，$k$ 的 $dp_{n,k,c}$ 求和，其中 $c > \lfloor \frac{k}{2} \rfloor$ 且 $0 < k$ 这样，时间复杂度为 $O(m \times n^3 m)$（因为 $k$ 和 $c$ 的范围都在 $n$ 以内），不可接受。

考虑 $c > \lfloor \frac{k}{2}$ 实际上意味着什么。注意，这条件等价于 $c > \frac{k}{2}$

1. 如果 $k$ 是偶数，则 $\left\lfloor \frac{k}{2} \right\rfloor = \frac{k}{2}$。

2. 如果 $k$ 是奇数 ，$c > \lfloor \frac{k}{2} \rfloor$ 等于 $c > \frac{k}{2} - \frac{1}{2}$，对于整数 $c$ 这等价于 $c > \frac{k}{2}$。

整理表达式，得到 $2c-k>0$，或者 $c-(k-c)>0$。其实际意义是 $j = x$ 的物品数量，减掉 $j \neq x$ 的物品数量，大于 $0$。所以只需要关心这两个值的差，而不关心菜肴的数量。动态规划就可以进行以下修改。

$dp_{i,\Delta c}$ 为只考虑前 $i$ 个操作，食材 $x$ 被用的次数减去食材 $x$ 未被用到的次数为 $\Delta c$ 的方案数。注意此处的 $\Delta c$ 可能为负数。

当烹饪方法为 $i$，食材为 $j$，转移为：

1. 如果 $j = x$，则执行 $dp_{i,\Delta c+1} \mathrel{+}= a_{i,j}\, dp_{i-1,\Delta c}$。

2. 如果 $j \neq x$，则执行 $dp_{i,\Delta c-1} \mathrel{+}= a_{i,j}\, dp_{i-1,\Delta c}$。

若不选，则执行 $\mathrm{dp}_{i,\Delta c} \mathrel{+}= \mathrm{dp}_{i-1,\Delta c}$。最后将所有 $\Delta c > 0$ 对应的 $\mathrm{dp}_{n,\Delta c}$ 相加。

将同类转移合并后，不必枚举所有 $j$，可以分为 $j = x$、$j \ e \ x$ 和不选三种情况。以下=$ 表示将右侧贡献累加到左侧：

1. $j = x$：$\mathrm{dp}_{i,\Delta+1} \mathrel{+}= a_{i,x}\, \mathrm{dp}_{i-1,\Delta}$。

② $j\ne x$: $\mathrm{dp}_{i,\Delta-1}\mathrel{+}=\left(\sum_{j=1}^{m}a_{i,j}-a_{i,x}\right)\mathrm{dp}_{i-1,\Delta}$。

3. 不选：$\mathrm{dp}_{i,\Delta} \mathrel{+}= \mathrm{dp}_{i-1,\Delta}$。

虽然最终只考虑 $\Delta c$ 为正的部分，但中间过程中的负值也可能对答案有贡献，因此需要同时维护正、负的 $\Delta c$。实现时可以把坐标整体平移；以下代码将所有 $\Delta c$ 加上 101，以便使用非负数组下标。最终时间复杂度为 $O(n^2 m)$。参考代码如下：

```
#include <bits/stdc++.h>
#define MAXN 105
#define MAXM 2005
using namespace std;
const int MOD = 998244353;
int S[MAXN];
int a[MAXN][MAXM];
int dp[MAXN][MAXN << 1];
int main() {
    int ops, mats;
    cin >> ops >> mats;
    int prod = 1;
    for (int i = 1; i <= ops; i++) {
        for (int j = 0; j < mats; j++) {
            cin >> a[i][j];
            S[i] = (S[i] + a[i][j]) % MOD; // 前缀和
```

```
        }
        prod = 1LL * prod * (S[i] + 1) % MOD; // 统计满足限制 1、2 的方案总数
    }
    int ans = (prod + MOD - 1) % MOD;
    for (int x = 0; x < mats; x++) { // 枚举烹饪方法。
        memset(dp, 0, sizeof dp);
        dp[0][0 + 101] = 1;
        int no = 0;
        for (int i = 1; i <= ops; i++) {
            int re = (S[i] + MOD - a[i][x]) % MOD; // 选择其他烹饪方法的方案总数
            for (int dt = 1; dt <= 201; dt++) { // 分类讨论转移
                dp[i][dt + 1] = (dp[i][dt + 1] + 1LL * a[i][x] * dp[i - 1][dt]) % MOD;
                dp[i][dt - 1] = (dp[i][dt - 1] + 1LL * re * dp[i - 1][dt]) % MOD;
                dp[i][dt] = (dp[i][dt] + dp[i - 1][dt]) % MOD;
            }
        }
        for (int dt = 1; dt <= ops; dt++)
            no = (no + dp[ops][dt + 101]) % MOD; // 统计总数违反限制 3 的方案总数
        ans = (ans + MOD - no) % MOD;
    }
    cout << ans << endl;
}
```

### 例 20-3 拿糖果

假设张三有 $a$ 颗不同的糖果，李四有 $b$ 颗不同的糖果，其中张三和李四都有的糖果有 $c$ 颗，请问不同糖果的总数是多少

相信读者都明白本题的答案是 $a+b-c$ 但这个简单问题的背后就是**容斥原理**的思想。

设张三的糖果的集合是 $A$，李四糖果的集合是 $B$，那么张三和李四共有的糖果就是 $A \cap B$，不同糖果的集合就是 $A \cup B$。根据上述结论，得到了关系式 $|A \cup B| = |A| + |B| - |A \cap B|$。这就是容斥原理的二元形式。

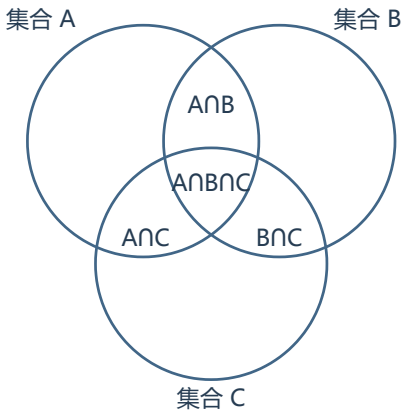



图20-2 容斥原理的集合

如果人数增加到 3 个人呢?直接进行讨论会比较麻烦。这时可以用一个韦恩图直观感受，如图 20-2 所示。

首先加上 $|A| + |B| + |C|$，然后发现，对于 $|A \cap C||B \cap C|, |A \cap B$ 部分重复计算了。

接着减去 $|A\cap C|+|B\cap C|+|A\cap B|$。可以发现，两个集合相交部分解决了，但是 $3$ 个集合相交部分却成为新的问题。因为在第 $1$ 次加上 $|A|+|B|+|C|$ 时，$|A\cap B\cap C$ 部分加了 $3$ 次，而第 $2$ 次减去 $|A\cap C|+|B\cap C|+|A\cap B$ 时，中间部分又减去了 $3$ 次。

因此，需要再加上一个 $|A\cap B\cap C|$，才是最后的正确答案。

三元容斥的式子如下：

$$|A\cup B\cup C|=|A|+|B|+|C|-|A\cap B|-|A\cap C|-|B\cap C|+|A\cap B\cap C|$$

对于多元的情况，可以得到最后的式子，即

$$\begin{aligned}&|A_1\cup A_2\cup\cdots\cup A_n|\\&=\sum_{i=1}^{n}|A_i|-\sum_{1\leqslant i<j\leqslant n}|A_i\cap A_j|+\sum_{1\leqslant i<j<k\leqslant n}|A_i\cap A_j\cap A_k|-\cdots\\&+(-1)^{n-1}|A_1\cap A_2\cap\cdots\cap A_n|\end{aligned}$$

这一结论可以由归纳法证明，本书不再赘述，感兴趣的读者可以自行研究。

### 例 20-4 盒子与球1(洛谷P1287)

有多少种方案将 $n$ 个互不相同的球放入 $r$ 个互不相同的盒子，使得没有一个盒子为空？数据范围：$0\leqslant r\leqslant n\leqslant 10$，且答案小于 $2^{31}$。

**分析**：容易想到，如果去掉没有空的条件，由乘法原理，最终的答案为 $r^n$。但如果有出现空的盒子呢?

指定第 $a_1,a_2,\cdots,a_k$ 盒子为空，则和 $n$ 个球分别有 $r-k$ 个盒子可以放入。由乘法原理，在这种情况下有 $(r-k)^n$ 种方案。又由于对于任意一个 $k$，有 $C_r^k$ 个满足条件的 $a_1,a_2,\cdots,a_k$，于是有 $k$ 个盒子为空的情况总数为 $C_r^k(r-k)^n$。

依照**容斥原理**，可以得到答案为

$$\sum_{k=0}^{r-1}(-1)^kC_r^k(r-k)^n$$

注意若 $r=0$，则答案为 $1$ 当且仅当 $n=0$，否则为 $0$。由于本题中 $n$ 足够小，组合数可以通过组合数计算公式直接求解。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
long long pw(int a, int b) {
    long long res = 1;
    for (int i = 1; i <= b; i++) {
        res *= a;
    }
    return res;
} // 求幂的函数。
long long C(int a, int b) {
    if (a < b)
        return 0;
    long long res = 1;
    for (int i = a; i > a - b; i--) {
        res *= i;
        res /= (a - i + 1);
```

```
    }
    return res;
} // 求组合数的函数。
int main() {
    int n, r;
    cin >> n >> r;
    long long ans = 0;
    for (int k = 0; k <= r - 1; k++) {
        ans += ((k & 1) ? -1 : 1) * C(r, k) * pw(r - k, n); // 统计答案
    }
    cout << ans;
    return 0;
}
```

### 例 20-5 硬币购物(洛谷P1450,河南省队选拔2008)

有 $4$ 类硬币，面值分别为 $c_1, c_2, c_3, c_{40}$ 现在求 $n$ 次：第 $i$ 类硬币共有 $d_i$ 枚，有多少种方案购 买价值为 $s$ 的东西刚好花完。

**数据范围：** $1 \leqslant c_i, d_i, s \leqslant 10^5, 1 \leqslant n \leqslant 1000$

**分析：** “第 $i$ 类硬币有 $d_i$ 枚”可被看作限制：“第 $i$ 类硬币不能用多于 $d_i$ 枚”。

考虑如果一个限制都没有，即每一个硬币量都无限多，如何回答询问。使用背包**动态规划**递推预处理所有 $s$ 的答案，设 $dp_{i,j}$ 表示前 $i$ 类硬币支付 $j$ 元的方案数。转移方程为 $dp_{s,z} = dp_{s-c_z,z} + dp_{s,z-1}$，其中初始状态是 $dp_{0,0} = 1$，答案为 $dp_{s,4}$ 这里采用滚动数组对空间进行优化。

接下来考虑，如果第 $a_1, a_2 \cdots a_1 (_{t \leqslant 4})$ 类硬币超过了限制，其他硬币不考虑，如何回答询问。事实上只需要强制用了至少 $d_{a_i} + 1$ 个硬币。

于是可以对这些硬币预先从 $s$ 里扣走 $\sum_{i=1}^{t} (d_{a_i} + 1) \times c_{a_i}$ 个硬币，而将这些直接安排给超过限制的硬币即可。而对于剩下的 $s - \sum_{i=1}^{t} (d_{a_i} + 1) \times c_{a_i}$ 个硬币如何分配，又是一个新的完全背包问题，其处理方法与之前的动态规划一致。

接着，由**容斥原理**，可以得到最终的答案为“至少偶数种硬币超限”的情况减去“至少奇数种硬币超限”的情况。在这里，可以预处理动态规划，然后直接统计得出答案。代码如下：

```
#include <bits/stdc++.h>
#define MAXN 100000
using namespace std;
long long dp[MAXN + 5];
long long get(long long x) {
    if (x < 0)
        return 0;
    return dp[x]; // 查找 dp 值
}
int main() {
    int c1, c2, c3, c4, queries;
    cin >> c1 >> c2 >> c3 >> c4 >> queries;
    dp[0] = 1;
    for (int i = c1; i <= MAXN; i++)
        dp[i] += dp[i - c1]; // 完全背包预处理数组 dp。注意此处采用了滚动数组优化
    for (int i = c2; i <= MAXN; i++)
```

```
        dp[i] += dp[i - c2];
    for (int i = c3; i <= MAXN; i++)
        dp[i] += dp[i - c3];
    for (int i = c4; i <= MAXN; i++)
        dp[i] += dp[i - c4];
    while (queries--) {
        long long d1, d2, d3, d4, s;
        cin >> d1 >> d2 >> d3 >> d4 >> s;
        d1 = (d1 + 1) * c1;
        d2 = (d2 + 1) * c2;
        d3 = (d3 + 1) * c3;
        d4 = (d4 + 1) * c4;
        cout << get(s) - get(s - d1) - get(s - d2) - get(s - d3) - get(s - d4) +
                    get(s - d1 - d2) + get(s - d1 - d3) + get(s - d1 - d4) +
                    get(s - d2 - d3) + get(s - d2 - d4) + get(s - d3 - d4) -
                    get(s - d1 - d2 - d3) - get(s - d1 - d2 - d4) -
                    get(s - d1 - d3 - d4) - get(s - d2 - d3 - d4) +
                    get(s - d1 - d2 - d3 - d4) // 容斥求结果
             << endl;
    }
}
```

## 20.2 常见的组合计数方法

接下来学习几种组合数学中常见的计数方法。

### 例 20-6 种树(洛谷P5520)

有多少种方案在 $n$ 个位置种 $m$ 株不同的幼苗，使得幼苗两两不相邻？答案对 $p$ 取模。

**数据范围：** $1 \leqslant n \leqslant 2 \times 10^6, 1 \leqslant m \leqslant 10^6$。

**分析：**对于本题，直接对着构造取计数看起来会很困难。因此可以考虑使用一些其他的思路进行解题。

采用插空的思路。由于需要选取 $m$ 个位置放置幼苗，因此共有 $n-m$ 个位置不放置幼苗。这 $n-m$ 个位置中间共有 $n-m-1$ 个“空位”，加上第 1 个位置的前面和最后一个位置的后面，共有 $n-m+1$ 个“空位”。需要将幼苗放在这些“空位”中。

显然，两株幼苗不可以放在同一个“空位”中，否则这两株幼苗将会相邻。因此需要从 $n-m+1$ 个“空位”中选取 $m$ 个不同的“空位”来种植幼苗。由于幼苗是各不相同的，因此“空位”选取需要考虑顺序，故最终的答案为 $A_{n-m+1}^{m}$ 只需要 $O(n-m)$ 计算组合数即可，参考代码如下：

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int t, n, m, p;
    cin >> t >> n >> m >> p;
    int a = 1;
    n = n + 1 - m;
    while (m--) {
        a = 1ll * a * n % p;
        n--;
    }
    cout << a << endl;
}
```

本题所采用的做法被称为插空法（或隔板法），是组合类题中一种常见的计算方法。对于有“不能相邻”条件的问题，直接计算较为复杂时，可以将其他元素先完成放置，然后将不能相邻的元素分别放到空位之中，从而计算出答案。

与隔板法对应的是捆绑法。用于有“必须相邻”条件的问题，可以将必须相邻的若干元素算作一个“捆绑体”参与排序。在定下顺序之后，再将“捆绑体”内的元素进行排序，就可以计算得到答案。

### 例 20-7 排列计数(洛谷P4071,山东省队选拔2016)

有多少个 $1$ 到 $n$ 的排列 $a$，使得有恰好 $m$ 个位置满足 $a_i=i$? 回答 $T$ 组询问。答案对 $10^9+7$ 取模。

**数据范围：** $1\leqslant n\leqslant 10^6, 0\leqslant m\leqslant n, 1\leqslant T\leqslant 5\times 10^5$.

**分析：**不妨先来考虑 $m=0$ 的情况，也就是所有数字全部“错位”时的答案。

发现很难用组合数直接表示出本题的答案，因此考虑使用其他方式进行求解。不妨假设有 $n$ 个数时的答案为 $d_n$，则尝试寻找 $d$ 数列的递推关系。

考虑第 $a_n$ 的取值。显然 $\Delta_n\Delta_{a_n}$ 共有 $n-1$ 种取值，且每种取值等价。不妨假设 $a_n=1$，那么接着来考虑 $a_1$ 的取值。根据 $a_1$ 的取值，将所有情况分为两类：

1. $a_1=n$。在这种情况下，可以直接去掉 $a_1$ 和 $a_n$ 两个数字，剩下的就是将 2 到 $n-1$ 这 $n-2$ 个数字进行全部“错位”的排序，这样的答案就是 $d_{n-2}$ $O$
2. $a_1\neq n$。在这种情况下，对数列做一个变换：将 $a_1$ 和 $a_n$ 互换，也就是 $a_n$ 变成了原来的 $a_1$，而 $a_1$ 变成了 $1$。这样的变换显然是一一对应的，因此变换前后的情况数相同。对于变换后的序列，去除第 1 个数后就是一个将 2 到 $n$ 这 $n-1$ 个数进行全部“错位”的排序，这样的答案就是 $d_{n-1}$ 。

两者相加，得到 $d_{n-2}+d_{n-1}$ 由于 $a_n$ 有 $n-1$ 种取值，因此可以得出，当有 $n$ 个数时，全部“错位”的排序的递推式为 $d_n=(n-1)(d_{n-2}+d_{n-1})$。

接着考虑 $m\neq 0$ 的情况。可以先任意选出 $m$ 个位置，指定其为满足 $a_i=i$ 的位置，显然这样的选择方法共有 $C_n^m$ 种。对于剩下的 $n-m$ 个元素和位置，就要求全部满足 $a_i\neq i$，这与完全“错位”的排序是等价的。因此，就可以得到最终的答案即为 $C_n^m\times d_{n-m}$。

由于询问的次数较多，因此需要预处理出 $d_n$ 的值。对于组合数，可以预处理出 $n!$ 的值及其关于 $10^9+7$ 的逆元，然后通过 $C_n^i=\frac{n!}{i!\,(n-i)!}$ 快速计算出组合数。参考代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int MOD = 1000000007;
const int MAXN = 1000006;
int qpow(int b, int e = MOD - 2) { // 逆元
    int a = 1;
    while (e) {
        if (e & 1)
```

```
            a = 1LL * a * b % MOD;
        b = 1ll * b * b % MOD;
        e >>= 1;
    }
    return a;
}
int fac[MAXN], D[MAXN];
int main() {
    fac[0] = 1;
    D[0] = 1;
    for (int i = 1; i <= 1000000; i++) {
        fac[i] = 1ll * fac[i - 1] * i % MOD;
        D[i] = (1LL * D[i - 1] * i + (i % 2 ? MOD - 1 : 1)) % MOD;
    }
    int T;
    cin >> T;
    while (T--) {
        int n, m;
        cin >> n >> m;
        // 组合数
        int C = 1LL * fac[n] * qpow(fac[m]) % MOD * qpow(fac[n - m]) % MOD;
        cout << 1LL * C * D[n - m] % MOD << endl;
    }
}
```

这里不加证明地给出另外一个快速求出组合数的方式。

定理 1（Lucas定理）： $C_n^m \equiv C_{n \bmod p}^{m \bmod p} \times C_{\lfloor n/p \rfloor}^{\lfloor m/p \rfloor} (\mathrm{mod} p)$。

参阅 洛谷P3807，可以使用递归求解，核心代码如下：

```
#define ll long long
ll pow(ll y, int z, int p) {
    y %= p;
    ll ans = 1;
    for (int i = z; i; i >>= 1, y = y * y % p)
        if (i & 1)
            ans = ans * y % p;
    return ans;
}
ll C(ll n, ll m) {
    if (m > n)
        return 0;
    return (a[n] * pow(a[m], p - 2, p)) % p * pow(a[n - m], p - 2, p) % p;
    // 这里的 a[i] 的值是 i!mod p
}
ll Lucas(ll n, ll m) {
    if (!m)
        return 1;
    return C(n % p, m % p) * Lucas(n / p, m / p) % p;
}
```

## 20.3 二项式定理

### 例 20-8 计算系数(洛谷P1313,NOIP2011提高组)

给定一个多项式 $(by+ax)^k$，请求出多项式展开后 $x^n \times y^m$ 项的系数，对 $10007$ 取模后的结果。数据范围： $0 \leqslant k \leqslant 1000, 0 \leqslant a,b \leqslant 10^6$。

**分析：**这里介绍一个重要的组合数定理。

定理 2（二项式定理）： $(x+y)^n=\sum_{i=0}^{n} C_n^i x^i y^{n-i}$，

为了便于理解，先写出几个 $n$ 较小时该定理的表达式（这些式子可以由乘法分配律得到）：

$$(x+y)^2=x^2+2xy+y^2$$

容易发现，其展开式的各项系数与杨辉三角的对应层（也就是组合数）恰好对应相等。关于杨辉三角的性质，例如 $C_n^m=C_{n-1}^m+C_{n-1}^{m-1}$，在《基础篇》已有阐述，本书不再赘述。

那么，如何证明这一规律对于所有 $n$ 都成立呢？下面给出这一结论的归纳证明：

1. 当 $n=1$ 时，原式为 $(x+y)=x+y$，结论显然成立。
2. 不妨假设 $n=k$ 时结论成立，即

$$(x+y)^k=\sum_{i=0}^{k}\binom{k}{i}x^i y^{k-i}$$

成立。而 $(x+y)^{k+1}=(x+y)\left(\sum_{i=0}^{k} C_k^i x^i y^{k-i}\right)$，由乘法分配律展开后得到原式等于 $x^{k+1}+y^{k+1}+\sum_{i=1}^{k}\left(C_k^{i-1}+C_k^i\right)x^i y^{k+1-i}$，由帕斯卡公式可得原式等于 $x^{k+1}+y^{k+1}+\sum_{i=1}^{k}\left(C_{k+1}^i\right)x^i y^{k+1-i}$，即 $\sum_{i=0}^{k+1}\left(C_{k+1}^i\right)x^i y^{k+1-i}$ 即结论对 $n=k+1$ 成立。

3. 综上所述，结论对任意 $n \in N_+$ 成立，其中 $N_+$ 代表正整数集合。

这里的证明方法被称为数学归纳法，只需通过证明最简单形式（$n$ 为较小常数时）的结论以及使用 $n=k$ 的结论推出 $n=k+1$ 的结论即可证明命题对任意满足条件的 $n$ 成立。这是一种重要的数学证明方法，但不要求提高组选手必须掌握，感兴趣的同学可以自行了解，本书不再赘述。

也可以从组合计数的角度理解二项式定理：把 $(x+y)^n$ 看作 $n$ 个 $(x+y)$ 相乘。要得到 $x^i y^{n-i}$，必须从其中 $i$ 个因式中选取 $x$，从其余 $n-i$ 个因式中选取 $y$。这样的选择共有 $\binom{n}{i}$ 种，因此 $x^i y^{n-i}$ 的系数是 $\binom{n}{i}$。

由乘法分配律，只需将所有的项累加，就得到 $C_n^i(x+y)^n$ 的展开式，即 $\sum_{i=0}^{n} C_n^i x^i y^{n-i}$，

回到题目，计算 $(by+ax)^k$ 中 $x^n y^m$ 的系数。将原式展开，$x$ 取了 $n$ 次，$y$ 取了 $m$ 次，所以这一项的系数就是 $C_k^m a^n b^m$.，数据范围不大，可以借助杨辉三角的思路，计算出 $C_k^m$，并通过暴力枚举计算出 $a$”和 $b$，即可完成本题。

如果数据范围再大一些，也可以借助逆元求解组合数，使用快 $a^n b^m$ 速幂求解 $a^n$ 和 $b^m$。代码如下：

```
#include <iostream>
#define MAXN 1000 + 5
#define MOD 10007
using namespace std;
int a, b, k, n, m;
long long f[MAXN][MAXN];
long long an = 1, bm = 1;
int main() {
    cin >> a >> b >> k >> n >> m;
    for (int i = 0; i <= k; i++)
        for (int j = 0; j <= i; j++)
            if (j == 0 || j == i)
                f[i][j] = 1;
            else
                f[i][j] = (f[i - 1][j - 1] + f[i - 1][j]) % MOD;
    for (int i = 1; i <= n; i++)
        an = (an * a) % MOD;
    for (int i = 1; i <= m; i++)
        bm = (bm * b) % MOD;
    cout << (f[k][n] * an * bm) % MOD;
    return 0;
}
```

二项式定理是代数和组合中一条重要的定理。关于这条定理，也可引申出许多有趣的推论，如以下这些结论：

```
(1) Σ = Σc
i为偶数 i为奇数
(2) $\sum _ { i \mathop { = } 0 } ^ { n } C _ { n } ^ { i } = 2 ^ { n }$
```

这些结论证明方法很多，但使用二项式定理证明是很简单的。对于第 1 条结论，只需要在原式中取 $x=1,y=-1$ 并进行移项整理，即可得到本式。第 2 条结论，只需要在原式中取 $x=1,y=1$ 即可证明。

### 例 20-9 高度相同(洛谷CF1332E)

给定一个 $n\times m$ 的网格，初始第 $i$ 行第 $j$ 列的格子有一个数字 $a$，可以进行两种操作： $a_{i,j}$

操作 $1$：让两个相邻格子中的数字各加 $1$

操作 $2$：让一个格子中的数字加 $2$。

给定 $n,m,L,R$，问有多少种初始局面满足以下条件：

对于所有的 $1\leqslant i\leqslant n,1\leqslant j\leqslant m,L\leqslant a_{i,j}\leqslant R$。

玩家可以通过执行若干次上述操作，使得所有格子上的数相同。

答案对 $998244353$ 取模。

**数据范围：** $1\leqslant n,m,L,R\leqslant 10^9,L\leqslant R,n\times m\geqslant 2$

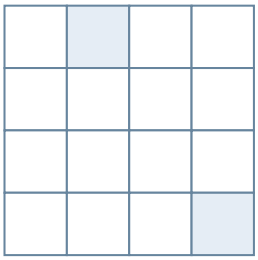
**分析：**容易发现，使用若干次加 2 操作，可以将一个奇/偶数变成比它大的任意一个奇/偶数。因此，只要能够使所有格子中的数字奇偶性相同，就可以达成最终的目的。因此对于本题，可以考虑从奇偶性入手。

接下来考虑操作 1 的作用。每使用一次操作 1 会使两个相邻的数奇偶性发生改变。若原来有 $a$ 个奇数，则操作后可能会出现 $a-2$ 个奇数、$a$ 个奇数、$a+2$ 个奇数共 3 种情况；而无论是哪 $\cdot a+2$ 种情况，$a$ 的奇偶性都是不变的。偶数也是如此。

由于最后需要将所有数全部变成奇数或偶数，则最终要将奇数或偶数中某一个的个数变成 $0$。不妨假设原来奇数、偶数的个数分别为 $a$ 和 $b$。由于 0 是偶数，因此 $a$ 与 $b$ 中至少有一个为 $b$ 偶，否则就不可能满足题意。

那么，是不是只需要 $a$ 或 $b$ 中至少有一个为偶，就可以满足题意呢?

事实上，对于任意两个奇数（或偶数），只需要选择一条由若干组相邻格子组成的“路径”，如图 20-3 所示，将所有路径上的每组格子分别进行一次操作 1，就可以将这两个数全部变成偶数（奇数），并且不对其他任何一个数字的奇偶性产生影响。



因此，只需要 $a$ 和 $b$ 中有任意一者为偶数（不妨设 a 为偶数），则只需要将所有的奇数两两配对，并通过上述方式使其全部变为偶数即可。综上所述，符合题意的初始情况等价于存在偶数个格子是奇数或偶数个格子是偶数。

那么如何来统计可行的方案数呢?

图 20-3 格子组成的路径

首先，若 $n\times m$ 为奇数，则显然 $a$ 和 $b$ 一奇一偶，满足条件。由于每个格子有 $R-L+1$ 种选择，故答案为 $(R-L+1)^{n\times m}$。

接下来考虑 $n\times m$ 为偶数的情况。由于 $a+b=n\times m$，因此需要满足 $a$ 与 $b$ 分别为偶数。从所有格子中选出 $a$（$a$ 为偶数）个并指定这些位置为奇数，剩余为偶数。假设$[L,R]$中的奇数有 $k$ 个，偶数有 $l$ 个，则最终的答案为 $\sum_{21a} C_{n\times m}^{a}\times k^{a}\times l^{b}$ （其中 $b=n\times m-a$）。

接着，考虑如何求解这个公式。由二项式定理，$(k+l)^{n\times m}=\sum_{a=0}^{n\times m} C_{n\times m}^{a}\times k^{a}\times l^{b}$。可以发现，需要求的是这里面次数为偶数的部分，因此只需要消去其中次数为奇数的部分即可。

同理，由二项式定理，$(k-l)^{n\times m}=\sum_{a=0}^{n\times m}(-1)^{a}\times C_{n\times m}^{a}\times k^{a}\times l^{b}$，需要的同样是其次数为偶数的部分。

而这两个式子的奇数次幂抵消，偶数次幂相同。因此只需要将两者相加并除以 2，即 $\frac{(k+l)^{n\times m}+(k-l)^{n\times m}}{2}$，就是最终的结果。参考代码如下：

```
#include <bits/stdc++.h>
using namespace std;
const int p = 998244353;
long long fp(long long a, long long b) {
    long long res = 1;
    while (b) {
        if (b & 1)
            res = res * a % p;
        a = a * a % p, b >>= 1;
    }
    return res;
}
int main() {
    long long n, m, l, r;
    cin >> n >> m >> l >> r;
    if ((n & 1) && (m & 1)) {
        cout << fp(r - l + 1, n * m);
    } else {
        long long a = r / 2 - (l - 1) / 2;
        long long b = r - l + 1 - a;
        cout << (fp((a + b) % p, n * m) + fp((a - b + p) % p, n * m)) * 499122177 % p;
        // 499122177为2在模998244353意义下的逆元
    }
    return 0;
}
```

## 20.4 卡特兰数与斯特林数

### 例 20-10 栈(洛谷P1044,NOIP2003普及组)

有一个空栈，有 $n$ 个元素（编号从 $1$ 到 $n$）依次按照顺序入栈，但在入栈的过程中可能随时会有元素出栈的情况。求最终合法的出栈顺序一共有多少种。

**数据范围：** $n\leqslant 18$，

**分析：**不妨设 $C$ 是元素个数为 $n$ 时的答案。考虑采用递推的方法进行求解。 $C_n$

考虑编号为 1 的元素出栈的位置。容易发现，如果 1 号元素是第 $k$ 个出栈的，那么编号从 2 到 $k$ 的元素在此时必然已经出栈，而后 $n-k$ 个元素在此时还没有进栈。

由于前后两部分的出栈顺序同样满足先进后出的条件，因此编号为 2 到 $k$ 的元素的出栈顺序种类数恰为 $C_{k-1}$ 种，而后 $n-k$ 个元素的出栈顺序恰为 $C_{n-k}$ 种。所以只需要将 $k$ 取遍 1 到 $n$ 即可得到最终的答案，即

$$C_n=\sum_{k=1}^{n}C_{k-1}\times C_{n-k}$$

也可以写成

$$C_n=\sum_{i=0}^{n-1}C_i\times C_{n-i-1}$$

其中，定义 $C_0=1$。而这道题的答案即为 $C_n$，直接求解即可得到答案。代码如下：

```
#include <bits/stdc++.h>
int n, C[30];
int main() {
    scanf("%d", &n);
    C[0] = 1, C[1] = 1;
    for (int i = 2; i <= n; i++) {
        for (int j = 0; j < i; j++) {
            C[i] += C[j] * C[i - j - 1];
        }
    }
    printf("%d", C[n]);
    return 0;
}
```

在本题中出现的 $C$ 数列（从 0 开始），称为卡特兰数。虽然本题很简单，但卡特兰数却是一个重要的数列，它有许多优秀的性质。这里列举其中两条：

1. $C_n = C_{n-1} \times \frac{4n-2}{n+1}$
2. $C_n = \frac{C_{2n}^n}{n+1}$

其中 $C_{2n}^n$ 为组合数。

事实上，可以换一种角度来看本题。本题等价于进行 $n$ 次进栈操作和 $n$ 次出栈操作，一旦操作的顺序固定，则出栈结果同样固定。而一个合法的操作序列要求在任何时候栈中的元素个数要大于或等于 0。因此可以考虑通过求补集来求出最后的结果。 $0$。

假设先不考虑要求栈中元素个数大于或等于 0 这一条件，可以认为最后的情况总数就是在 $2n$ 个操作中选择 $n$ 个作为进栈操作，剩余的作为出栈操作。这样的情况总数恰好为 $C_{2n}^n$。

对于所有不合法的操作序列，将其进行一个变换不妨设进栈操作为+1，出栈操作为-1，则对于一个不合法的操作序列，必然有一个位置使得其前面的数（包含它本身）相加为 $-1$。取第 1 个这样的数字，显然这个数必然是第奇数个，否则不可能相加为-1。将这个位置后面的数（不包含它本身）符号全部改变，就得到了一个由 $n+1$ 个-1 与 $n-1$ 个+1 组成的数列。

接着来考虑，对于一个由 $n+1$ 个-1 与 $n-1$ 个+1 组成的数列，由于其所有数相加为负，必然有一个位置使得其前面的数（包含它本身）相加为-1。取第 1 个这样的数字。同样这个数必然是第奇数个，否则不可能相加为-1。将这个位置后面的数（不包含它本身）符号全部改变，就得到了一个由 $n$ 个-1 与 $n$ 个+1 组成的数列，且这个数列是不合法的操作序列。

经过上述变换，不合法的操作序列与由 $n+1$ 个 $-1$、$n-1$ 个 $+1$ 组成的数列一一对应。因此不合法序列共有 $\binom{2n}{n-1}$ 个。用全部序列数减去不合法数，即得 $C_n = \binom{2n}{n} - \binom{2n}{n-1}$。

利用 $C_n^m = \frac{n!}{m!(n-m)!}$ 易得第 2 个式子成立，接着将 $C_n$ 与 $C_{n-1}$ 相除可得第 1 个式子成立。

事实上，卡特兰数还是很多经典问题的答案，例如：

1. $n$ 对括号的合法匹配数目。
2. $n \times n$ 的网格，从左下角沿网格线走到右上角，且不经过对角线下方的情况数。
3. $n$ 个结点的二叉搜索树个数。
4. $n+2$ 边形通过对角线划分成 $n$ 个三角形的方法数…

对于以上例子感兴趣的同学可以自己推算其公式。在碰到题目时，如果感觉其前几项与卡特兰数相同，也可以采用猜测+逆向证明的方式来尝试证明，或许答案就是卡特兰数。

接下来，先回顾前面的 例 20-4，再用另一种方式来思考。

### 例 20-11 盒子与球2(洛谷P1287)

现有 $r$ 个互不相同的盒子和 $n$ 个互不相同的球，要将这 $n$ 个球放入 $r$ 个盒子中，且不允许有空盒子。请求出有多少种不同的放法。

**数据范围：** $1 \leqslant r \leqslant n \leqslant 10$。

**分析：**在“**容斥原理**”节中，求出了本题的答案为 $\sum_{k=0}^{r-1}(-1)^k C_r^k (r-k)^n$.。在这里用另一种方式再来思考本题。不妨先来考虑，如果所有的盒子是相同的，本题该如何处理。

考虑递推。设 $S(n,r)$ 表示有 $n$ 个球 $r$ 个盒子，且盒子相同时的答案。不妨将球按从 1 到 $n$ 编号。考虑编号为 $n$ 的球的放置情况：

1. 该球独占一个盒子。此时剩下的 $n-1$ 个盒子应当共享 $r-1$ 个盒子，故方案数应为 $S(n-1,r-1)$ ）。
2. 该球与其他球共享一个盒子。此时不妨先摆放好其他球（共 $S(n-1,r)$ 种情况）。接着，由于编号为 $n$ 的球可以放置在 $r$ 个盒子内，故由乘法原理，方案数为 $r \times S(n-1,r)$。

由加法原理，得出 $S(n,r)$ 的递推式为 $S(n,r)=S(n-1,r-1)+r\times S(n-1,r)$，

接着来考虑盒子不同时的情况。可以将其分成两步：

1. 决定盒子的顺序，显然摆放盒子的方式共有 $r!$ 种。
2. 将球摆放入盒子。由于此时的盒子顺序已经决定好，故此时的答案即为 $S(n,r)$

由乘法原理，两者相乘，得到最终答案为 $r! \times S(n,r)$。参考代码如下：

```
#include <bits/stdc++.h>
using namespace std;
long long S[20][20];
int main() {
    int n, r;
    cin >> n >> r;
    long long ans = 1;
    for (int i = 1; i <= r; i++) {
        ans *= i;
    }
    S[1][1] = 1;
    for (int i = 2; i <= n; i++) {
        for (int j = 1; j <= i; j++) {
            S[i][j] = S[i - 1][j - 1] + j * S[i - 1][j];
        }
    }
    ans *= S[n][r];
    cout << ans;
    return 0;
}
```

本题中的 $S(n,r)$ 被称为第二类斯特林数，同样拥有许多优秀的性质。由本题的两种计算方法可得

$$S(n,r)=\frac{1}{r!}\sum_{k=0}^{r-1}(-1)^k C_r^k(r-k)^n$$

那么也许有人会问，既然有第二类斯特林数，那么第一类斯特林数是什么呢？

第一类斯特林数表示将 $n$ 个不同元素构成 $m$ 个圆排列的数目（计作 $s(n,m)$，此处 $s$ 小写）。同时，第一类斯特林数又分有、无符号两种，分别计作 $s_s(n,m)$ 与 $s_u(n,m)$，其关系为 $s_s(n,m)=s_u(n,m)\times(-1)^{n+m}$。在组合数学中，一般默认为研究无符号第一类斯特林数。

依然可以使用递推的方式来求第一类斯特林数的递推式。将元素按 1 到 $n$ 编号，考虑编号为 $n$ 的元素的排列情况：

1. 第 $n$ 个元素单独构成一个圆排列。此类情况共有 $s(n-1,m-1)$ 种。
2. 第 $n$ 个元素与其他元素同处一个圆排列。对于此类情况，先将其他元素完成排序，此时共有 $s(n-1,m)$ 种；接着将这一元素插入到任意一个元素的顺时针下一顺位，共有 $n-1$ 种选择。由乘法原理，共有 $s(n-1,m)\times(n-1)$ 种。

综上，由加法原理得 $s(n,m)=s(n-1,m-1)+(n-1)\times s(n-1,m)$。

与第二类斯特林数不同的是，第一类斯特林数的通项公式较难被求出；但随着学习的深入，会有越来越多的方式来求解斯特林数。

## 20.5 课后习题与实验

### 习题 1 幸运数字(洛谷P2567,四川省队选拔2010)

十进制数表示中只包含数字 6 和 8 的那些号码称作“幸运号码”，幸运号码的倍数称为“近似幸运号码”，求$[a,b]$中的幸运号码个数。

**数据范围：** $1\leqslant a\leqslant b\leqslant 10^{10}$。

### 习题 2 分特产(洛谷P5505,江苏省队选拔2011)

有 $n$ 名同学和 $m$ 种特产，第 $i$ 种特产有 $a_i$ 个。现需要将特产分给所有同学，要求每个同学 至少分到 1 个。答案对 $10^9+7$ 取模。

**数据范围：** $1\leqslant n,m,a_i\leqslant 10^3$。

### 习题 3 排队(洛谷P3223,湖南省队选拔2012)

某中学有 $n$ 名男同学，$m$ 名女同学和两名老师要排队参加体检。他们排成一条直线，并且任意两名女同学不能相邻，两名老师也不能相邻，那么一共有多少种排法?

**数据范围：** $1\leqslant n\leqslant 2\times 10^3, 1\leqslant m\leqslant 2\times 10^3$

### 习题 4 组合数问题(洛谷P2822,NOIP2016提高组)

给定 $n,m$ 和 $k$，对于所有的 $0\leqslant i\leqslant n, 0\leqslant j\leqslant \min(i,m)$ 有多少对 $(i,j)$ 满足 $k|\mathrm{C}_i^j$.

**数据范围：** $0\leqslant n,m\leqslant 2\times 10^3, 1\leqslant t\leqslant 10^4$

### 习题 5 音阶(洛谷P3214,湖南省队选拔2011)

将声音分成 $n$ 个音阶，并将音乐分成若干片段。音乐的每个片段都是由 $1$ 到 $n$ 个音阶构成的和声，即从 $n$ 个音阶中挑选若干音阶同时演奏出来。要求任意两个片段所包含的音阶集合都不同，且一段音乐中每个音阶被奏响的次数为偶数。求包含 $m$ 个片段的音乐一共有多少种。答案对 $10^8+7$ 取模。

**数据范围：** $1 \leqslant n,m \leqslant 10^6$.

### 习题 6 树屋阶梯(洛谷P2532,安徽省队选拔2012)

有若干空心、横截面积不同的钢材，任意一种尺寸（尺寸边长均为正数）都有无数个，求搭建高度为 $n$ 的阶梯的方案数。

**数据范围：** $1 \leqslant n \leqslant 500$。

### 习题 7 有趣的数列(洛谷P3200,湖南省队选拔2009)

称一个长度为 2n的数列是有趣的，当且仅当该数列满足以下 $3$ 个条件：

1. 它是从 $1 \sim 2n$ 共 2n个整数的一个排列 $\{a_n\}_{n=10}^{2n}$
2. 所有的奇数项满足 $a_1 < a_3 < \cdots < a_{2n-1}$，所有的偶数项满足 $a_2 < a_4 < \cdots < a_{2n}$。
3. 任意相邻的两项 $a_{2i-1}$ 与 $a_{2i}$ 满足 $a_{2i-1} < a_{2i}$。

对给定的 $n$，求出有多少个不同的长度为 2n的有趣的数列。答案对 $p$ 取模。

**数据范围：** $1 \leqslant n \leqslant 10^6, 1 \leqslant p \leqslant 10^9$。

### 习题 8 纸牌(洛谷CF1278F)

有 $m$ 张牌，其中一张是王牌。现在执行 $n$ 次如下操作：洗牌并查看第 $1$ 张牌是什么。令 $x$ 为洗牌后第 $1$ 张牌为王牌的次数，现在假设洗牌时 $m!$ 种牌的排列出现的概率均相等，求 $x^k$ 的期望，答案对 $998244353$ 取模。

**数据范围：** $1 \leqslant n,m < 998244353, 1 \leqslant k \leqslant 5 \times 10^3$

### 习题 9 数列(洛谷P3228,湖南省队选拔2013)

已知股票每天的价格（正整数且不超过 $n$）。在 $k$ 天中除第一天外每天的股价都比前一天高，高出的价格不会超过 $m$，且 $m(k-1) < n$、假如忘记了具体每天的股价，求这 $k$ 天的股价有多少种可能。答案对 $p$ 取模。

**数据范围：** $1 \leqslant m,k,p \leqslant 10^9, n \leqslant 10^{18}$

### 习题 10 划分(洛谷 CF9616)

给出 $n$ 个物品。每个物品有一个权值 $w_{i\,c}$ 定义一个集合 $S$ 的权值 $W(S) = \mid S \mid \sum_{x \in S} w_x$. 定义一个划分的权值为其所有分出的子集的权值和。求将 $n$ 个物品划分成 $k$ 个集合的所有方案的权值和，答案对 $10^9+7$ 取模。

**数据范围：** $1 \leqslant n,k \leqslant 2 \times 10^5, w_i \leqslant 10^9$

# 第21章 概率与统计

在日常生活中，有些事情是必然的——例如太阳从东方升起；有些事情是偶然的——例如明天下雨。随着对偶然事件的研究不断深入，发现其中很大一部分是有章可循的。抛 100 次硬币，大概有 50 次是正面朝上的；掷 600 次骰子，其中大概有 100 次得到点数 1。

本章讨论的概率论，就是研究这些随机现象中蕴含的数学规律。概率论是实用的科学，尽管人们无法预知明天是否真的会下雨，但掌握了明天下雨的概率后，至少可以做出“要不要带伞出门”的决策；当计算出一场游戏的期望收益是负数时，还是不要参与游戏为宜。

本章先讨论一些非常基础的概率模型（如古典概型、几何概型），然后给出其形式化的定义。接下来，讨论全概率公式、贝叶斯公式，并引人**数学期望**的概念。本章思维导图如图 21-1 所示。




图21-1 本章思维导图

## 21.1 事件

### 例 21-1 生活中的事件

在讨论概率之前，有必要了解事件。随机试验指的是在相同的条件下，对某个随机现象进行的大量重复观测。随机试验可以在相同条件下重复进行；随机试验出现的所有可能的结果不止一个，但都事先已知；每次试验总是出现可能的结果之一，但试验前无法预知得到哪一种结果。抛硬币是随机试验，因为可以重复地抛这枚硬币很多次；可能的试验结果是“正面朝上”或“反面朝上”，不存在其他结果；每次抛硬币前所有人都不知道这次是正面朝上，还是反面朝上。

抛硬币有正面、反面朝上两种结果。生孩子有男性、女性两种结果。掷骰子有 $6$ 种结果。把这些随机试验的每一种可能的结果，称为基本事件，或称样本点，一般用 $e$ 表示。而由基本事件的全体构成的集合，称为样本空间，一般用 $S$ 来表示。于是，每一次随机试验，都是从样本空间中抽取出了一个样本点。

将一枚硬币连续抛两次，一共只有 $4$ 种可能：正正、正反、反正、反反。写出基本事件：

$$e_1=(\text{正},\text{正}),\quad e_2=(\text{正},\text{反}),\\ e_3=(\text{反},\text{正}),\quad e_4=(\text{反},\text{反}).$$

样本空间就是上述所有样本点的集合 $S=\{e_1,e_2,e_3,e_4\}$，

随机试验出现的结果并不保证是有限的。例如，在区间$[0,1]$中随机取一个点，每一个实数都对应一个基本事件；样本空间是 $\{x\in R\ |0\leqslant x\leqslant 1\}$，是无限的，其中的样本个数比所有自然数的个数还要多！

基本事件是概率论中的“原子”。例如，抛两次硬币，“恰有一次正面朝上”包含 $e_2,e_3$ 两个基本事件；从 $[0,1]$ 中取数，“取出的数小于 $0.5$”则包含无穷多个基本事件。**随机事件** $A$ 是样本空间 $S$ 的一个子集；当且仅当 $A$ 中某个基本事件发生，称事件 $A$ 发生。

事件是接下来讨论的基础。在样本空间 $S$ 中，如何刻画“必然事件”和“不可能事件”？以连续抛两次硬币为例，“正面朝上的次数不超过 $2$”必然发生，而“正面朝上的次数超过 $2$”不可能发生。下面给出严格的定义。

1. 若 $A=S$，则称 $A$ 是必然事件。由于随机试验总是出现可能的结果之一，故当 $A$ 包含了所有的样本点时，它是一定会发生的。
2. 若 $A=0$，则称 $A$ 是不可能事件。由于它没有包含任意基本事件，所以它永远不会发生。

### 例 21-2 事件的运算

从刚刚的讨论中，可以看出随机事件的本质是样本点的集合。因此，对事件的运算，可以自然而然地转化到集合的运算。

1. **包含关系。** 例如，掷骰子两次，“点数之和小于 $8$”成立时，“点数之和小于 $10$”也一定成立，因此前者包含于后者。一般地，若事件 $A$ 包含的每个样本点都属于事件 $B$，则记为 $A\subseteq B$，表示 $A$ 发生时 $B$ 一定发生。
2. 事件也有可能相等。抛两次硬币，“两次都是正面朝上”和“没有一次是反面朝上”这两个事件，表达的是同一个意思。如果两个事件所包含的基本事件是相同的，那么这两个事件也是相等的，记为 $A=B_s$

3. 几个不同的事件可能会同时发生。例如，连续掷两次骰子，“点数之和等于 8，且第一次点数小于 4”，这个事件既要求了点数之和，又要求了第一次掷骰子的点数，所以整个事件是{（2，6），（3，5）}，只包含两个样本点。这种情况被称为事件的交，以 $A \cap B$ 表示，简记为AB，含义是“事件 $A$ 与事件 $B$ 都发生”。如果有许多个事件一起发生，可以简记为 $\bigcap_{i=1}^{n} A_i = A_1 \bigcap A_2 \bigcap \cdots \bigcap A_n$。
4. 两个事件的交也有可能是空集。所以 $AB = \emptyset$ 的时候，就可以说“$A$ 与 $B$ 不能同时发生”。

这里的 $A$ 与 $B$ 就是互斥事件，或称“互不相容事件”。另外，$n$ 个事件互斥（不能同时发生）的充分必要条件，是其中任意两个事件互斥。

5. 事件既然有交，那就也可以有并。使用 $A \cup B$ 表示“$A$ 与 $B$ 中至少一个发生”。特殊地，如果 $A \cap B = \emptyset$，也就是说 $A$，$B$ 包含的基本事件是互不相同的，则可以选择进一步地用加号来描述它们的并： $A \cup B = A + B$。如果有许多个事件，其中至少一个发生，可以记为 ∪Ai。 $\bigcup_{i=1}^{n} A_i$
6. 接下来讨论事件的差。把“$A$ 发生、$B$ 不发生”记为 $A - B$，类似于集合的差运算。例如抛两次硬币，事件 $A$ 为“至少有一次正面朝上”，事件 $B$ 为“第一次抛硬币是反面朝上”，那么 $A - B$ 就是{（正，正），（正，反）}；$B - A$ 是{（反，反）}。

如果读者阅读过《基础篇》的“逻辑命题与位运算”节，应该就不会对事件的运算感到陌生。以上事件的运算结果如图 21-2 所示。

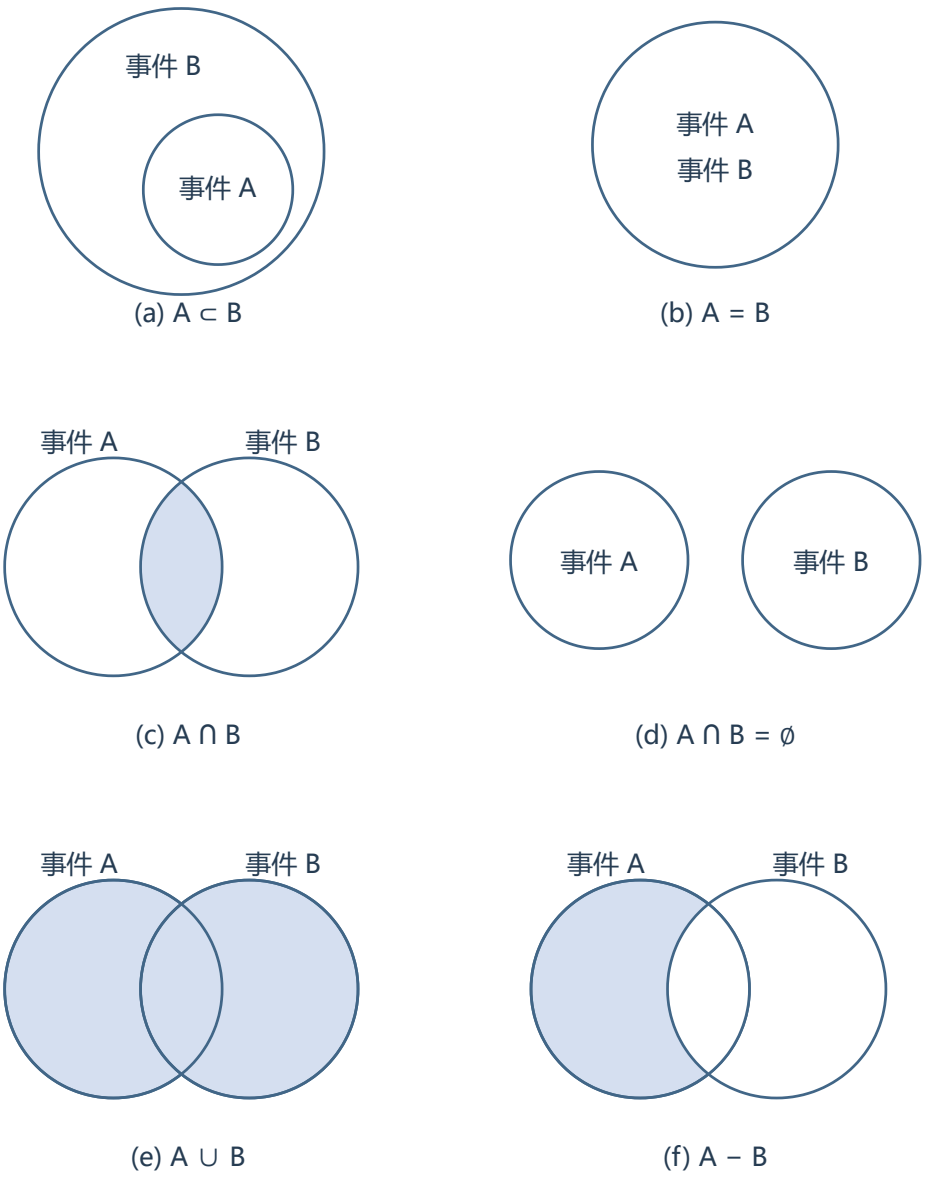



图21-2 事件的运算

有了这些基础，接下来讨论对立事件：由 $A$ 不发生所构成的事件，称为 $A$ 的对立事件。如果 $A$ 与 $B$ 对立，那么随机试验的结果一定是 $A$ 和 $B$ 其中的一个。抛两次硬币，“至少一次正面朝上”和“全都反面朝上”是对立事件，试验结果要么是前者要么是后者，不能都发生，也不能都不发生。对立事件对应了集合中的补运算，记 $A$ 的对立事件为 $\overline{A}$ 它是由 $S$ 中所有不在 $A$ 中的基本事件构成的。

以下运算性质不难证明 $\cdot A\overline{A}=\infty, A+\overline{A}=S, \overline{A}=A.$

若 $A=S$，则称 $A$ 是必然事件。由于随机试验总是出现可能的结果之一，故当 $A$ 包含了所有的样本点时，它是一定会发生的。

若 $A=\emptyset$，则称 $A$ 是不可能事件。由于它没有包含任意基本事件，所以它永远不会发生。

集合的交和并运算，是可以通过补运算来互相转化的。这就是著名的德·摩根定律：

$$\overline{A\cup B}=\overline{A}\cap\overline{B}, \qquad \overline{A\cap B}=\overline{A}\cup\overline{B}$$

第 1 条可以感性理解为，如果想要“（$A$ 或 $B$）不发生”，那就得 $A$ 不发生，同时 $B$ 不发生；第 2 条可以理解为，如果想要“（$A$ 且 $B$）不发生”，那么要么 $A$ 不发生，要么 $B$ 不发生，要么两者都不发生。德·摩根可以拓展到多个集合的情形，请读者自行推导。

这里花了很大的篇幅来讲述事件以及事件之间的运算。读者马上会发现这是值得的，因为可以帮助读者在接下来的概率学习中找准方向，不走弯路。当读者无法准确理解各种概率模型的时候，不妨牢牢抓住“这个事件包含了哪些基本事件”这个视角，以“样本点所组成的集合”的运算来理解事件的运算，问题可能会迎刃而解。

## 21.2 概率模型

### 例 21-3 古典概型

一枚质地均匀的硬币，其抛出正面、反面的概率是一致的；掷一枚骰子，其 6 个面朝上的概率是均等的。像这样，各个基本事件出现的概率相等的情形，称为古典概型。概型是“概率模型”的简称。

在古典概型中，人们关注的事件是基本事件之和，例如“掷一枚骰子，点数大于 4”这个事件即是点数恰为 5、点数恰为 6 这两个基本事件的和。因为基本事件的发生概率是相等的，可以得到

$$P(A)=\frac{A\text{ 包含的基本事件个数}}{\text{基本事件的总数}}$$

因此，掷一枚骰子，点数大于 4 的概率是 $2/6$。

需要指出的是，抛一枚硬币朝上的概率为 50%，不代表抛 10 枚硬币里面就一定会有 5 枚硬币朝上。究竟抛 10 次硬币，其中 $x$ 次向上的概率是多少将在二项分布中讨论。

### 例 21-4 生日问题

假设一年有 365 天，每个人的出生日期是均匀随机的。班里面有 30 位同学，问“存在两个同学生日相同”的概率。

首先，不妨认为 30 位同学是有各自的学号的。那么在这个“试验”中，可能会得到“1 号同学的生日是四月一日、2 号同学的生日是九月五日…30 号同学的生日是六月三日”这样的结果，这就是试验中的基本事件。它可以写成一个拥有 30 个元素的元组，所以总共有 36530 种可能的基本事件，它们出现的概率是相等的，符合古典概型的定义。

那么，存在两个同学生日一样的概率是多少呢？直接计算是不简单的，可以采用另一种思路：算出“所有同学生日都不同”的概率，这与“存在两个同学生日相同”是互斥事件。拿 1 减去前者，就得到需要的结果。

想知道“所有同学生日都不同”这个事件发生的概率，就需要知道其包含的基本事件个数。第 1 个同学可以任选一天作为生日，第 2 个同学为了不与之碰撞，只有 364 种选项。依此类推，第 30 个同学只有 336 种选项。乘起来即为所有同学生日不同的情况总数。这采用基础的排列组合知识就能得到

$$A_{365}^{30}=365\times364\times\cdots\times336$$

于是，$P$（所有同学生日不同 $)=\frac{365\times364\times\cdots\times336}{365\times365\times\cdots\times365}\approx0.29368$ 可以发现，在这个班里存在两个学生生日相同的概率达到了 70%。从这道趣题可以看出，利用古典概型进行概率计算，常常需要掌握排列组合的相关知识。

### 例 21-5 世界杯门票(洛谷P2719)

售票窗口出售 $A$ 类门票和 $B$ 类门票各 $n(n\leqslant 1250)$ 张，有 2n个人排在窗口前买票。购买时由工作人员通过抛硬币决定，抛得正面的买 $A$ 类票，抛得反面的买 $B$ 类票。抛硬币出现正、反面的可能性是一样的。如果一类门票提前卖完，则接下来排队买票的人直接买到另外一类门票而无须抛硬币 。请问：排在队尾的两个人同时拿到同一种票的概率是多少？

解法 1：和 例 21-4 一样，需要确定基本事件的数量，以及符合题目要求的事件“最末尾两人拿到相同门票”的事件数量。所有基本事件数量好说，在 2n个人中，选择 $n$ 个人持有 $A$ 类票， $2n$ 那么一共有 $C_{2n}^{n}$ 种组合。但满足“最末尾两人拿到相同门票”的各种事件概率并不一定相等，不是很好计算。其实也可以反过来思考，计算“最末尾两人拿到不同门票”的概率即可。

假设两类门票各有 5 张，一共 10 人。如果最后两个人拿到不同类别的门票，那么前 8 个人买票的时候，两类门票都没有被售空，因此这 8 个人买票前都是靠抛硬币决定门票的类别，可以计算“排到最后两人之前，两类门票都没有被卖空”的概率。符合要求的事件数量是C4 种，每种 $C_8^4$ 事件出现的概率是一（因为要连续抛 8 次硬币才能达成一种符合要求的事件），8 人排完后门票 $\frac{1}{2^8}$ 都没有卖光的概率就是 $C_8^4\times\frac{1}{2^8}$。

因此，2（$n-1$）人排完后门票都没有卖光的概率是 $C_{2n-2}^{n-1}\times\frac{1}{2^{2n-2}}$，符合题意要求的“最末尾两人拿到相同门票”是“排到最后两人之前，两类门票都没有被卖空”的对立事件，最后的答案是 $1-C_{2n-2}^{n-1}\times\frac{1}{2^{2n-2}}$

编写程序计算这个值，但不可以用循环分别计算分子和分母，因为阶乘的结果太大，无法使用double类型保存。可以将原式变为 $\frac{(2n-2)!}{(n-1)!(n-1)!}\cdot\frac{1}{4^{n-1}}=\frac{2n-2}{(n-1)\times 4}\cdot\frac{2n-3}{(n-2)\times 4}\cdots\frac{n}{1\times 4}$，分别计算这些值然后乘起来即可，可以理解为边乘边除以保证精度。

代码如下：

```
#include <iostream>
#include <cstdio>
using namespace std;
int main() {
    int n;
    double ans = 1.0;
    scanf("%d", &n);
    n = n / 2; // 题目中输入给的是 2n 的值
    for (int i = 1; i <= n - 1; i++) {
        ans = ans * (i + n - 1) / (i * 4);
    }
    printf("%.4lf", 1 - ans);
    return 0;
}
```

解法 2：可以使用类似于递推的方式完成本题。令 $f_{n,m}$ 是还剩下 $n$ 张 $A$ 类票 $\backslash m$ 张 $B$ 类票时，最后两张票相同的概率。如果 $n\leqslant 1$ 且 $m\leqslant 1$，说明不可能存在两张相同的票，则 $f_{n,m}=0($ 初始化的时候直接就是 0 了）。

否则，如果 $n=0$ 或 $m=0$，说明只剩下一种票了，则 $f_{n,m}=1$ 除此之外，这个状态是前面的一个人要么拿走 $A$ 类票，要么拿走 $B$ 类票，概率各半，所以 $f_{n,m}=\frac{1}{2}(f_{n-1,m}+f_{n,m-1})$。

代码如下：

```
#include <cstdio>
#include <iostream>
#define MAXN 1250 + 5
using namespace std;
double f[MAXN][MAXN];
int main() {
    int n;
    scanf("%d", &n);
    n /= 2;
    for (int i = 2; i <= n; i++)
        f[i][0] = f[0][i] = 1;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            f[i][j] = 0.5 * (f[i - 1][j] + f[i][j - 1]);
    printf("%.4lf", f[n][n]);
    return 0;
}
```

这种解题方法需要记录所有状态的概率信息，也需要处理状态之间的转移，被称为概率**动态规划**。

### 例 21-6 几何概型

假设某人在等车，他知道车在 $8$ 时到 $9$ 时之间的每一时刻，来的概率是均等的。那么很容易想到，车在 $08:30$ 之前到达的概率是 $50\%$。

这样的概率模型和古典概型有一些相似，因为每一个基本事件（车在某一特定时刻来）的概率都相等；但在本题中，基本事件的个数有无限个，以至于任何单个基本事件的发生概率都是 $0$。这种概率模型被称为几何概型。求解几何概型的概率问题，常常采用数形结合的方法。

考虑另一个问题：现在（$0$ 时刻）有甲乙两个人开始等车，甲的车在前 $60$ 分钟内的随机时刻会到；乙的车在前 $120$ 分钟内的随机时刻会到。问甲比乙早上车的概率。

以原点为左下角画一个矩形，横轴代表甲何时上车，纵轴代表乙何时上车，如图 21-3(a) 所示。

那么，很容易想到，（甲上车时间，乙上车时间）这个随机变量的分布，是满足几何概型的。它随机地分布在上图所画出的矩形中。接下来考虑“甲比乙早上车”这个事件。这个时间是满足 $x < y$ 的区域，也就是函数 `f(x)`：$y = x$ 的上方区域，如图 21-3(b) 所示。于是可以计算出甲比乙早上车的概率是

$$\frac{\text{事件对应面积}}{\text{总面积}} = \frac{60 \times 60 + 60 \times 60/2}{60 \times 120} = \frac{3}{4}$$

### 例 21-7 估计π

某人往一个宽度为 1m的正矩形里均匀随机撒点，问点与左下角的距离不超过 1m的概率。

画出该正矩形，则满足条件的点须落在以左下角为圆心、半径为 1m的扇形上，如图 21-4 所示。

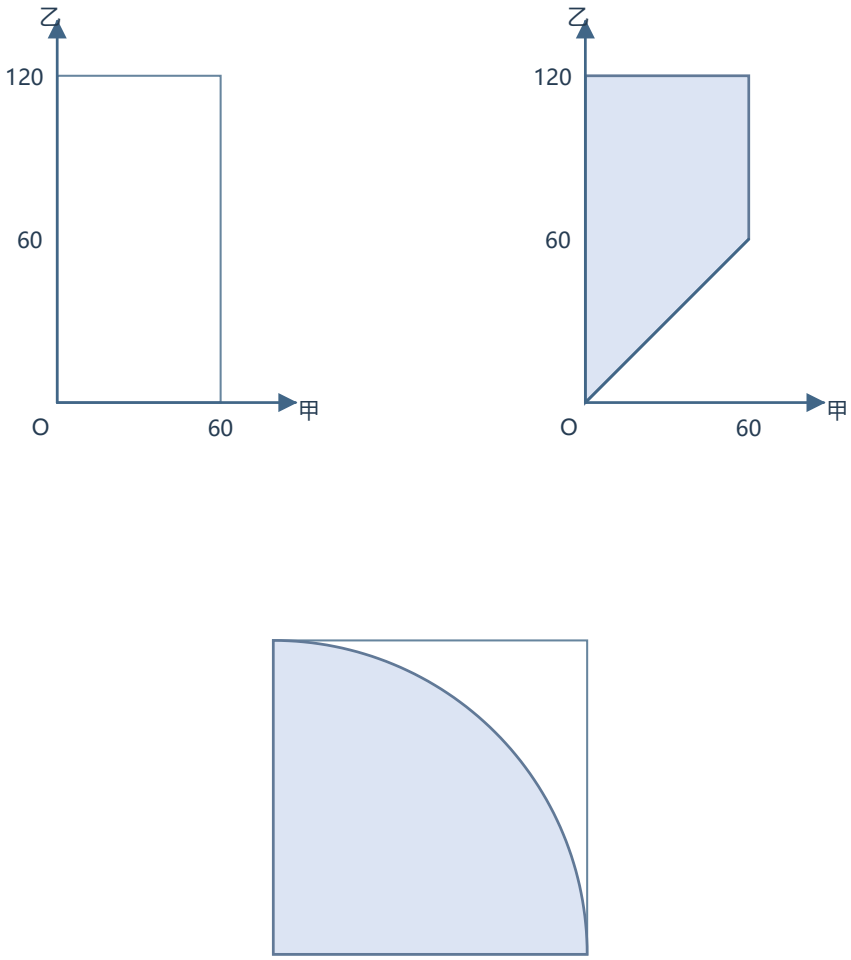



图21-4 估计圆周率

那么，落在这个圆内的概率是：$\frac{\frac{2\pi}{4}\cdot|4-x+5|\pi/1|}{|\frac{2\pi}{4}\cdot|4|}=\frac{\pi/4}{1}=\frac{\pi}{4}$。

通过这个试验，可以估计圆周率。试验次数越多，事件发生的频率会越来越接近概率。如果抛了 $1\times10^8$ 个点，其中有 $c$ 个点与左下角距离不超过 $1$，那么可以用以下式子估算圆周率：$\frac{\pi}{4}\approx\frac{c}{10^8}$ 编写程序，代码如下：

```
#include <cstdio>
#include <random>
using namespace std;
int main() {
    int c = 0;
    random_device rd;
    mt19937 gen(rd());
    uniform_real_distribution<> dis(0, 1);
    int maxn = 100000000;
    for (int i = 0; i < maxn; i++) {
        double x = dis(gen);
        double y = dis(gen);
        if (x * x + y * y <= 1)
            c++;
    }
    printf("%.7lf\n", 4.0 * c / maxn);
    return 0;
}
```

使用Linux系统和 C++14 标准编译并运行这个程序，就可以输出一个 π 的近似值。这里使用了random头文件和相关的对象生成 0 到 1 之间的浮点数，相比于函数 `rand()` 生成随机数效率高且精度高，感兴趣的读者可以自行查阅相关的资料。

### 例 21-8 条件概率与贝叶斯公式

按往年经验，本机场飞机延误的概率是 20%；刚刚有一个台风在附近登陆，显然不能还按照 20% 的概率来估计今天的航班延误的可能性。这就是说，$P$（航班延误|台风登陆）>$P$（航班延误）。所谓条件概率，指的是“$B$ 事件发生的情况下，$A$ 事件发生的概率”，记为 `P(AIB)`。按照条件概率的定义，不难得到条件概率的公式：

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}$$

首先来看两个很有意思的问题。假设生男孩和女孩的概率都是 50%。

问题 1：一对夫妻生了两个孩子，其中第 1 个孩子是女孩。问第 2 个孩子是女孩的概率。

感性地看，第一胎生啥与第二胎是没关联的。所以答案应该是 50%。

严格来说，一共有 4 个基本事件：（男男），（男女），（女男），（女女）。现在求 $P$（第 2 个孩子 $P$（第 1 个孩子是女孩、第 2 个孩子是女孩）是女孩|第 1 个孩子是女孩），它的值就等于 $P$（第 1 个孩子是女孩）$\frac{1/4}{2/4} = 50\%$

问题 2：一对夫妻生了两个孩子，其中有一个孩子是女孩。问另一个孩子是女孩的概率。

这个问题乍一看，也会认为“有一个孩子是女孩”与“另一个孩子是男是女”无关。但是仔细一 $\frac{4}{7}, \frac{\frac{1}{7}(7+\frac{1}{12}-1)\times 5+5+3\times 3}{9(7+1)} - \frac{1}{12} \times \frac{1}{7} \times \frac{1}{12} - \frac{1}{3} \times \frac{1}{12} = \frac{1}{3}$， 。

发现并不是一半。这说明这两个事件并不是相互独立的。因为“有一个孩子是女孩”排除掉了（男男）这个基本事件，于是只剩下 3 个可能的基本事件了。

如果已知 $B$ 发生的概率，求 $A$ 发生的概率，可以使用贝叶斯（Bayes）公式。使用条件概率公式直接推导：

$$P(A \mid B)=\frac{P(A \cap B)}{P(B)}=\frac{P(B \mid A)P(A)}{P(B)}$$

这在实际生活中非常有用。假设某种疾病在一般人群中的发病率是 $0.01\%$，而某种检测方式可以检测这种疾病。如果某人患病，则检测阳性的概率是 $99\%$；如果没患病，则检测出阳性的概率是 $1\%$。现在某人检测阳性，问他真的患上该病的概率。

常理上来讲，患者有 $99\%$ 的概率检测呈阳性，且健康人被检测出阳性的概率只有 $1\%$，我们可能会不假思索就认为这人真患病的可能性大。但是计算一下：

$$\begin{aligned} P(\text{患病} \mid \text{检测阳性}) &= \frac{P(\text{检测阳性} \mid \text{患病})P(\text{患病})}{P(\text{检测阳性})} \\ &= \frac{99\% \times 0.01\%}{99\% \times 0.01\% + 1\% \times 99.99\%} \\ &\approx 0.0098. \end{aligned}$$

原来真的患病的概率还不到 $1\%$，这是该检测方法过高的假阳性率导致的。上面的计算中对分母的处理方式，即认为 $P$（检测阳性）=$P$（检测阳性且患病）+$P$（检测阳性且不患病）是没有问题的，因为患病和不患病是互补事件。事实上，人们经常使用这种方式计算贝叶斯公式中的分母。

## 21.3 数学期望

### 例 21-9 数学期望和分布

抛 $100$ 次硬币，预期大约有 $50$ 次正面朝上；掷 $600$ 次骰子，预期大约有 $100$ 次出现点数 $4$。所谓**数学期望**，简称期望，就是试验中每次可能的结果乘以其结果概率的总和。换句话讲，期望值是随机变量的输出值的加权平均数，权重即为各个事件对应的概率。

掷一枚骰子，用随机变量 $X$ 表示朝上的点数，则以计算 $X$ 的期望：

$$E(X) = 1 \times \frac{1}{6} + 2 \times \frac{1}{6} + \cdots + 6 \times \frac{1}{6} = 3.5$$

那么，掷非常多次骰子，向上的点数之均值就应该落在 $3.5$ 附近。

数学期望具有线性性质。形式化地讲 $E(aX + bY) = aE(X) + bE(Y)$，这可以很大程度地方便运算。例如，一场竞赛由两个游戏组成，分数是这两个游戏得分相加。那么某选手在整场竞赛中得分的期望，就等于这两场游戏的得分期望之和。但是数学期望不满足乘法的性质，也就是 `E(XY)` 不一定等于 $E(x)E(y)$。

概率分布，是指用于表述随机变量取值的概率规律。如果试验结果用变量 $X$ 的取值来表示，则随机试验的概率分布就是随机变量的概率分布，即随机变量的可能取值及取得对应值的概率。以下是几种常见的分布。

#### 1. 两点分布

两点分布，又称伯努利分布、$0-1$ 分布。它描述了一个非常简单的模型：一场试验只有成功（结果是 $1$）和失败（结果是 $0$）两种可能，其成功的概率为 $p$，失败的概率为 $1-p$。“抛一枚硬币，正面向上则试验成功”就是典型的伯努利试验。

两点分布的期望值是 $E(X) = p \times 1 + (1-p) \times 0 = p$。

### 2. 几何分布

在伯努利实验中，得到一次成功所需的试验次数是 $X$。那么在 $k$ 次试验中，第 $k$ 次才能得到成功的概率是

$$P(X=k)=(1-p)^{k-1}p$$

几何分布的期望是 $E(X)=\frac{1}{p}$，证明如下：

令 $q=1-p, E(x)=p+2qp+3q^2p+\cdots=\sum_{k=1}^{\infty}kq^{k-1}p=\frac{p}{(1-q)^2}=\frac{1}{p}$ 得证。化简 $\sum_{k=1}^{\infty}kq^{k-1}$ 可以使用错位相减和等比数列求和公式。

这说明，如果某彩票中奖率为 $10\%$，如果没有中奖就再抽一次，直到中奖为止，这被称为一次试验。这种情况下期望抽奖次数是 $10$。它不意味着抽 $10$ 次一定会中奖，而是指进行很多次抽奖试验后，平均抽奖次数是 $10$ 次。

### 3. 二项分布

二项分布是用于描述“进行 $n$ 次伯努利试验，成功的次数”的模型。“抛 $100$ 次硬币，正面朝上的次数”就是典型的二项分布。可以推导出

$$P(X=k)=C_n^kp^k(1-p)^{n-k}$$

其期望值的形式却很简单： $E(X)=np$。虽然直接证明也比较复杂，但这个很好理解：掷 $600$ 次骰子，出现每面的概率都是 $1/6$，可以预期大约有 $100$ 次出现点数 $4$。

### 4. 超几何分布

超几何分布，用于描述“抽样不放回”的模型。例如，今有一批 $N$ 个产品，其中有 $K$ 个不合格。抽出 $n$ 个样品送检，输出送检样品中不合格样本的数量（即随机变量 $X$）。通过组合数学的知识，可以得到

$$P(X=k)=\frac{C_K^kC_{N-K}^{n-k}}{C_N^n}$$

一共有 $C_N^n$ 种抽取的方式（基本事件），每个基本事件发生的概率相同。分子上就是从 $K$ 个不合格样本里面抽出 $k$ 个不合格样本的方案数，乘以从剩下 $N-K$ 个合格样本中抽出 $n-k$ 个合格样本的方案数。

超几何分布的期望是抽样数量乘以总的不合格概率，即 $E(x)=n\frac{K}{N}$ or 证明过程稍复杂，请读者自行查阅资料。需要注意的是，前一次抽取的结果会影响后续试验的样本空间，但概率权重会抵消这个影响。

### 例 21-10 成功与失败(洛谷P1654)

一共有 $n$ 次操作，每次操作只有成功与失败之分，成功对应 $1$，失败对应 $0$，$n$ 次操作对应为 $1$ 个长度为 $n$ 的 $0-1$ 串。在这个串中连续的 $x$ 个 $1$ 可以获得 $x^3$ 的分数，这 $x$ 个 $1$ 不能被其他连续的 $1$ 所包含。最后得分就是每次获得的分数的总和。现在给出 $n$，以及每个操作的成功率 $p_i(0\leqslant p_i\leqslant 1)$，请输出期望得分，输出四舍五入后保留 $1$ 位小数。

**分析：**得分与连续 $1$ 的长度有关。设 $E(f)_i$ 为考虑到位置 $i$ 时总得分的期望，$E(x)_i$ 为以 $i$ 结尾的连续 $1$ 的长度期望，$E(x^2)_i$ 为该长度的平方的期望。若位置 $i$ 成功的概率为 $p_i$，结合前一位置的连续长度，可得：

$$E(x)_i = p_i(E(x)_{i-1} + 1) + (1 - p_i) \times 0 = p_i(E(x)_{i-1} + 1)$$

因为期望只有线性性质，$E(x^2) = E^2(x)$ 是不成立的，所以可以推导出

$$E(x^2)_i = p_i E[(x+1)^2]_{i-1} = p_i\big(E(x^2)_{i-1} + 2E(x)_{i-1} + 1\big)$$

假设第 $i$ 次操作成功，产生连续长度为 $x$ 的概率为 $P_x$，那么对于每一个 $x$，对分数产生 $(x+1)^3 - x^3 = 3x^2 + 3x + 1$ 的贡献。第 $i$ 次操作成功有 $p_i$ 的概率，可以让得分增加如上的贡献，有

$$E(f)_i = E(f)_{i-1} + p_i\Big(\sum_{x=0}^{i-1} P_x(3x^2 + 3x + 1)\Big)$$

由于 $\sum_{x:=0}^{i-1} x^2 P_x = E\big(x^2\big)_{i-1}$ 且 $\sum_{x:=0}^{i-1} : xP_x = E(x_{i-1}$，有：

$$E(f)_i = E(f)_{i-1} + p_i(3E(x^2)_{i-1} + 3E(x)_{i-1} + 1)$$

所以可以使用递推的方式求出 $E(x)_{i\backslash E}(x^2)$ $i$ 和 $E(f)_i$，初始条件均为 0，代码如下：

```
#include <cstdio>
#define maxn 100000 + 5
using namespace std;
int n;
double p[maxn], Ex[maxn], Ex2[maxn], Ef[maxn];
signed main() {
    scanf("%lld", &n);
    for (int i = 1; i <= n; i++)
        scanf("%lf", &p[i]);
    for (int i = 1; i <= n; i++) {
        Ex[i] = p[i] * (Ex[i - 1] + 1);
        Ex2[i] = p[i] * (Ex2[i - 1] + 2 * Ex[i - 1] + 1);
        Ef[i] = Ef[i - 1] + p[i] * (3 * (Ex[i - 1] + Ex2[i - 1]) + 1);
    }
    printf("%.11f", Ef[n]);
    return 0;
}
```

### 例 21-11 树上的游戏(洛谷CF280C)

给定一棵有根树，结点编号从 $1$ 到 $n(1 \leqslant n \leqslant 10^5)$。根结点为 $1$ 号结点。对于每一次操作，等概率地选择一个尚未被删去的结点并将它及其子树全部删去。当所有结点被删除之后，游戏结束；也就是说，删除 $1$ 号结点后游戏即结束。需要求出删除所有结点的期望操作次数，保留不少于 $6$ 位小数。

例如，对于一棵有 $3$ 个顶点的树，结点 $1$ 和 $2$ 相连，$1$ 和 $3$ 相连。可能的操作有：

1. 删除 $1$，直接没了，概率 $\frac{1}{3}$，操作 $1$ 次。
2. 删除 $2$，然后删除 $1$，概率 $\frac{1}{3} \cdot \frac{1}{2}$，操作 $2$ 次。
3. 删除 $2$，然后删除 $3$，最后删除 $1$，概率 $\frac{1}{3} \cdot \frac{1}{2}$，操作 $3$ 次。

4. 删除 $3$，然后删除 $1$，概率 $\frac{1}{3}\cdot\frac{1}{2}$，操作 $2$ 次。

5. 删除 $3$，然后删除 $2$，最后删除 $1$，概率 $\frac{1}{3}\cdot\frac{1}{2}$，操作 $3$ 次。

期望操作次数为 $\frac{1}{3}+2\times(\frac{1}{6}\times 2+\frac{1}{6}\times 3)=2$

**分析：** 令 $f_i$ 是在一次试验（完整地删除一棵树）中，结点 $i$ 被直接选中导致的删除（因为它的祖先结点被删除而受牵连的不算）的次数，其取值只有 1 或者 0，因此 $E(f_i)=p_i$，即结点 $i$ 在一次试验中被直接选中导致删除的概率。题目需要的是 $E(\sum f_i)$，由期望的线性的性质可得

$$E(\sum f_i)=\sum E(f_i)=\sum p_i$$

对于结点 $i$ 来说，设沿着它的父结点一直往上直到根结点组成的链由 $d_i$ 个结点组成（换句话说，这个结点的深度是 $d_i$） 。在一次试验中，要么这个结点被选择删除，要么它的某个祖先被选择删除而受牵连。第一次删除到这条链上时显然各点被选中的概率均等，那么这个结点被选择删除的概率为 $\frac{1}{d_i}$，所以 $p_i=\frac{1}{d_i}$，最后的答案就是 $E(\sum f_i)=\sum\frac{1}{d_i}$。

只需要统计每个点的深度，计算求和就可以得到答案。代码如下：

```
#include <cstdio>
#include <vector>
#define MAXN 100000 + 5
using namespace std;
int n, d[MAXN];
double ans = 0;
vector<int> son[MAXN];
void dfs(int x, int depth) {
    d[x] = depth;
    for (int i = 0; i < son[x].size(); i++)
        if (d[son[x][i]] == 0)
            dfs(son[x][i], depth + 1);
}
int main() {
    scanf("%d", &n);
    for (int i = 1; i < n; i++) {
        int x, y;
        scanf("%d%d", &x, &y);
        son[x].push_back(y);
        son[y].push_back(x);
    }
    dfs(1, 1);
    for (int i = 1; i <= n; i++)
        ans += 1.0 / d[i];
    printf("%.7lf", ans);
}
```

题目并不保证输入的边中，前面那个结点深度较小，或者编号较小的结点深度较小，因此需要双向连边建无向图，然后遍历的时候同时记录深度；在寻找子结点时，如果遇到的结点的深度已经被记录，说明遇到了它的父结点，跳过即可。

## 21.4 课后习题与实验

### 习题 1 请解决以下和事件相关的问题

1. 连续掷两次骰子，每个骰子出现的点数 **1** 到 **6** 的可能性相等。请写出“点数之和为 **10**”这个事件包含的所有基本事件。
2. 从 **1** 到 **10** 中等概率选取一个数，请写出“取出的数是质数”这个事件所包含的所有样本点。
3. 今有甲乙丙 **3** 人，心里随便想一个自然数，想出一个奇数或者一个偶数的可能性相同，然后报出来。关注他们 **3** 个人想的数的奇偶性，请写出“甲想的数的奇偶性与乙相同”这个事件所包含的全部基本事件。

### 习题 2 根据事件的运算原理，请思考以下问题，可以举一些例子来验证你的判断

1. 互斥事件一定是对立事件吗?
2. 对立事件一定互斥吗?
3. 如果 $A$ 和 $B$ 互斥，那么 $\bar{A}$ 与 $B$ 是否一定不互斥?
4. 交运算是否符合交换律、结合律?
5. 并运算是否符合交换律、结合律?
6. $A\cup(B\cap C)$ 是否等于 $(A\cup B)\cap(A\cup C)$?
7. $A\cap(B\cup C)$ 是否等于 $(A\cap B)\cup(A\cap C)$?
8. $A\cup B$ 是否等于 $A+\overline{A}B$?
9. $A$ 是否等于 $AB+A\overline{B}$?

### 习题 3 计算以下概率与期望相关的题目

1. （NOIP2017 普及组初赛）一家 **4** 口人，至少两个人生日属于同一月份的概率是（）（假定每个人生日属于每个月份的概率相同且不同人之间相互独立）。
2. （NOIP2017 提高组初赛）小明要去旅游，一共乘坐三趟航班才能到达目的地，其中第 **1** 个航班准点的概率是 **0.9**，第 **2** 个航班准点的概率为 **0.8**，第 **3** 个航班准点的概率为 **0.9**。如果存在第 `i(i=1,2)` 个航班晚点，第 $i+1$ 个航班准点，则小明将赶不上第 $i+1$ 个航班，旅行失败；除了这种情况，其他情况下旅行都能成功。请问小明此次旅行成功的概率是（）。
3. （NOIP2017 提高组初赛）儿童游乐场有个游戏叫“欢乐喷球”，正矩形场地中心能不断喷出彩色乒乓球，以场地中心为圆心还有一个圆形轨道，轨道上有一列小火车在匀速运动，火车有 **6** 节车厢，如图 21-5 所示。

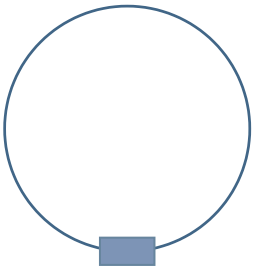


图21-5 欢乐喷球

假设乒乓球等概率落到正矩形场地的每个地点，包括火车车厢。小朋友玩这个游戏时，只能坐在同一个火车车厢里，可以在自己的车厢里捡落在该车厢内的所有乒乓球，每个人每次游戏有 $3$ 分钟时间，则一个小朋友独自玩一次游戏期望可以得到（）个乒乓球。假设

乒乓球喷出的速度为 $2$ 个/秒，每节车厢的面积是整个场地面积的 $1/20$。

4. （NOIP2018 提高组初赛）假设一台抽奖机中有红、蓝两色的球，任意时刻按下抽奖按钮，都会等概率获得红球或蓝球之一。有足够多的人每人都用这台抽奖机抽奖，假如他们的策略均为：抽中蓝球则继续抽球，抽中红球则停止。最后每个人都把自己获得的所有球放到一个大箱子里，最终大箱子里的红球与蓝球的比例接近于（）。
5. （NOIP2018 提高组初赛）在一条长度为 $1$ 的线段上随机取两个点，则以这两个点为端点的线段的期望长度是（）。

### 习题 4 条件概率(洛谷UVa11181)

有 $n(n\leqslant 20)$ 个人要去买东西，他们去买东西的概率为 $p_i$。现在已知恰有 $r(0\leqslant r\leqslant n)$ 个人买了东西，在这种条件下，求每个人有去买东西的概率。

### 习题 5 连击游戏(洛谷P1365)

有 $n$ 次点击要做，成功了就是 $o$，失败了就是 $x$，分数是按连击次数计算的，连击就是极大的连续 $o$，连击次数为 $a$ 就能得到 $a^2$ 分。有些点击跟运气无关，要么是 $o$，要么是 $x$；有些地方 $o$ 或者 $x$ 各有 $50\%$ 的可能性，用？号来表示。那么，玩家这场连击游戏的期望得分是多少呢？

### 习题 6 绿豆蛙(洛谷P4316)

给出一有 $n$ 个点、$m$ 条边的有向无环图，起点为 $1$，终点为 $n$，第 $i$ 条边的长度为 $l_{ic}$ 从起点出发能够到达所有的点，所有的点也都能够到达终点。绿豆蛙从起点出发，走向终点。到达每一个顶点时，如果该结点有 $k$ 条出边，绿豆蛙可以选择任意一条边离开该点，并且走向每条边的概率为 $\frac{1}{k}$。现在绿豆蛙想知道，从起点走到终点的所经过的路径总长度期望是多少?

### 习题 7 单选错位(洛谷P1297)

试卷上共有 $n(n\leqslant 10^7)$ 道单选题，第 $i$ 道单选题有 $a_i(a_i\leqslant 10^8)$ 个选项，这 $a_i$ 个选项编号是 $1,2,3,\cdots,a_i$，每个选项成为正确答案的概率都是相等的。

某同学认真地做完了这 $n$ 道题目，且答案正确，但他匆忙之下抄错位了：第 $i$ 道题目的答案抄到了答题纸上的第 $i+1$ 道题目的位置上，特别地，第 $n$ 道题目的答案抄到了第 $1$ 道题目的位置上。他想知道自己期望能做对几道题目。由于数据范围很大，请参考原题面提供的数据生成器。

### 习题 8 考试(洛谷P2111)

一场考试共有 $N(N\leqslant 10000)$ 道判断题，已知每道题目小明回答正确的概率都是 $A$%且题目之间互相独立。小红和小明对自己写的答案，结果是 $S$（一个长为 $N$ 的 $0-1$ 串，其中 $1$ 表示两人答案一样，$0$ 表示不一样），需要计算出小红答对 $Q$ 道题及以上的概率。

### 习题 9 换教室(洛谷P1850,NOIP2016提高组)

学校有 $v$ 个教室 $.^e$ 条边连接这些教室的图，每条边有边权 $w_i$。小明一共有 $n$ 节课需要依次前往 $c_i$ 号教室上课。小明有最多 $m$ 次机会申请换教室，从班级 $c_i$ 转换到班级 $d_i$，但是只有 $k_i$ 的概率被批准。每节课结束后去另外一个教室，需要走的距离就是这两个教室的最短路。期望一下小明走动距离总和的最小值。

**数据范围：** $1\leqslant m\leqslant n\leqslant 2000, 1\leqslant v\leqslant 300, 0\leqslant e\leqslant 90000$，

# 第22章 基础线性代数

线性代数是关于向量和线性映射的一个数学分支，主要处理线性关系问题。虽然读者可能没有系统地学习过线性代数，但肯定已经和线性代数打过交道了：在数学课中学习过如何求解二元一次方程，以及如何对一个函数进行伸缩变换；在物理课中，学习过力的平行四边形法则，也知道力作用在物体上，并使物体在力的方向上通过了一段距离的过程就是做功。这些问题都可以抽象成线性代数，相信读者有能力理解基础的线性代数。

许多读者学习线性代数的时候过于重视概念和运算，而忽视了其几何意义，这会让读者觉得线性代数是晦涩难懂的东西。同时，线性代数是一个很大的话题，在有限的篇幅全部介绍清楚是不现实的。本节将结合线性代数的几何意义介绍线性代数基本概念及其在算法竞赛中的应用，帮助读者初步理解线性代数。本章思维导图如图 22-1 所示。



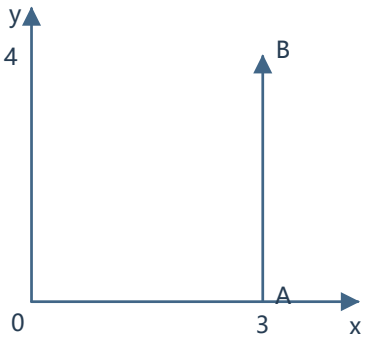

图22-1 本章思维导图

## 22.1 向量与矩阵变换

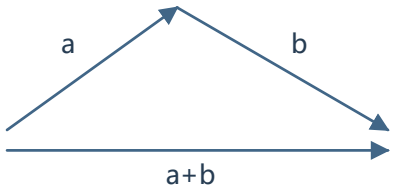
### 例 22-1 中学物理题

初始时小 $A$ 在直角坐标系的零点。小 $A$ 往 $x$ 轴的正方向走 $3$ 个单位，然后往 $y$ 轴的正方向走 $4$ 个单位，如图 22-2(a) 所示。问移动后小 $A$ 相比于直角坐标系零点的位置。如果重复上面的过程 $3$ 次，小 $A$ 又在什么地方呢？

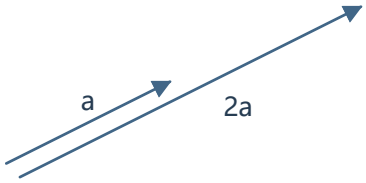
**分析：**线性代数中最根本、最基础的部分就是向量。




(a)




(b)




(c)

图22-2 向量的基本运算

在二维平面直角坐标系中，一个**向量**（也称为矢量）由两个数字组成，表示从起点到终点的位移。例如，$\begin{bmatrix}3\\0\end{bmatrix}$ 表示沿 $x$ 轴正方向移动 3 个单位，沿 $y$ 轴方向移动 0 个单位。也可以用 $(3,0)$ 表示，但容易与点的坐标混淆。一个向量可以记为 $\boldsymbol{a}$。

有明确起点和终点的线段称为**有向线段**。向量不指定起点，只规定终点相对于起点的位置。从点 $A$ 到点 $B$ 的有向线段记为 $\overrightarrow{AB}$。它的长度称为向量的**模**，记为 $|\overrightarrow{AB}|$ 或 $|\boldsymbol{a}|$。图中 $|\overrightarrow{AB}|=4$。模为 0 的向量称为**零向量**。

向量可以做加减运算。将第二个向量 $\boldsymbol{b}=\begin{bmatrix}3\\0\end{bmatrix}$ 的起点移动到第一个向量 $\boldsymbol{a}=\begin{bmatrix}0\\4\end{bmatrix}$ 的终点，连接 $\boldsymbol{a}$ 的起点与最终的终点，便得到它们的和，如图 22-2(b) 所示。向量相加，就是将对应分量分别相加：

$$\begin{bmatrix}x_1\\y_1\end{bmatrix}+\begin{bmatrix}x_2\\y_2\end{bmatrix}=\begin{bmatrix}x_1+x_2\\y_1+y_2\end{bmatrix}.$$

向量也可以做**数乘运算**，即按相应倍数拉长或缩短，如图 22-2(c) 所示。本题中 $3(\boldsymbol{a}+\boldsymbol{b})=\begin{bmatrix}9\\12\end{bmatrix}$。数乘是将各分量乘以同一个数；乘数为负数时，新向量的方向与原向量相反。

$$a\begin{bmatrix}x\\y\end{bmatrix}=\begin{bmatrix}ax\\ay\end{bmatrix}.$$

如果一组向量中，任意一个向量都无法由其余向量通过加法和数乘得到，则这些向量是**线性无关**的。在二维平面中，两个非零向量 $\boldsymbol{a},\boldsymbol{b}$ 不共线（不存在实数 $\boldsymbol{x}$ 使 $\boldsymbol{a}=\boldsymbol{x}\boldsymbol{b}$），就线性无关。

平面上的任意向量都可以用两个不共线向量表示：对于 $\boldsymbol{e}_1,\boldsymbol{e}_2$，任意向量 $\boldsymbol{a}$ 都存在唯一的实数对 $(\boldsymbol{x},\boldsymbol{y})$，使 $\boldsymbol{a}=\boldsymbol{x}\boldsymbol{e}_1+\boldsymbol{y}\boldsymbol{e}_2$。这两个向量称为**基向量**。常用的基向量是 $\boldsymbol{i}=\begin{bmatrix}1\\0\end{bmatrix}$ 与 $\boldsymbol{j}=\begin{bmatrix}0\\1\end{bmatrix}$。

模为 1 的向量称为**单位向量**，这两个基向量都是单位向量。

向量的加法和数乘贯穿线性代数的始终，二者起到很重要的作用。

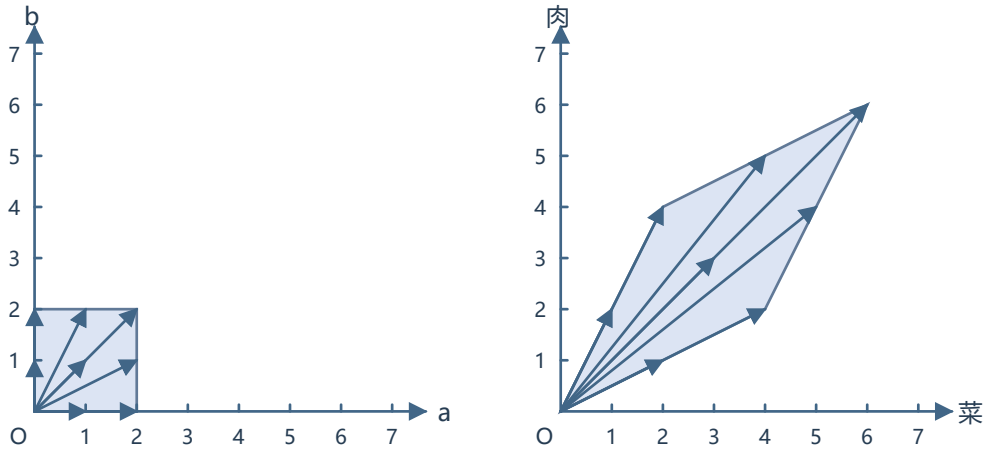
### 例 22-2 线性变换

一张甲类饭票可以换 $\boldsymbol{x}_1$ 千克蔬菜和 $\boldsymbol{y}_1$ 千克肉，一张乙类饭票可以换 $\boldsymbol{x}_2$ 千克蔬菜和 $\boldsymbol{y}_2$ 千克肉。现在有 a 张甲类饭票和 $\boldsymbol{b}$ 张乙类饭票。请问：可以换得多少千克的蔬菜和肉？

**分析：**所谓“变换”，其实就是一种函数：接收一个输入，并给出一个输出。在线性代数中，线性变换可被认为一种运动，可以输入向量，通过线性计算的方式转为另外一个向量并输出。将平面上的每一个点的坐标对应的向量都进行线性变换，就可以看出一定的规律。

假设 $(\boldsymbol{x}_1,\boldsymbol{y}_1)=(1,2),(\boldsymbol{x}_2,\boldsymbol{y}_2)=(2,1)$，显然向量 $\boldsymbol{p}=\begin{bmatrix}\boldsymbol{a}\\\boldsymbol{b}\end{bmatrix}$ 可以被变换为 $\begin{bmatrix}\boldsymbol{a}+2\boldsymbol{b}\\2\boldsymbol{a}+\boldsymbol{b}\end{bmatrix}$，这个输出向量代表可以换得蔬菜和肉的数量。

当向量 $\begin{bmatrix}\boldsymbol{a}\\\boldsymbol{b}\end{bmatrix}$ 的各个分量都在 0 到 2 之间，可以组成一系列变量，构成一个正矩形。这一系列变量经过变换，就构成了一个平行四边形，如图 22-3 所示。




例如，原向量 $\begin{bmatrix}2\\1\end{bmatrix}$ 经过变换后就变成了 $\begin{bmatrix}4\\5\end{bmatrix}$

线性变换后，原来是直线的元素仍然是直线，且原点还是原点。当然这边的 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 不局限于非负整数，即使是负数或实数也是可以的，相信读者可以从本题中理解线性变换趋势。

此时，使用 $\boldsymbol{a}$ 张甲类饭票和 $\boldsymbol{b}$ 张乙类饭票（$\boldsymbol{a}$ 和 $\boldsymbol{b}$ 都在实数范围内的），可以知道对应的蔬菜数量和肉的数量。形式化的描述就是

$$\begin{pmatrix}\boldsymbol{a}\\\boldsymbol{b}\end{pmatrix}\longmapsto\boldsymbol{a}\begin{pmatrix}\boldsymbol{x}_1\\\boldsymbol{y}_1\end{pmatrix}+\boldsymbol{b}\begin{pmatrix}\boldsymbol{x}_2\\\boldsymbol{y}_2\end{pmatrix}=\begin{pmatrix}\boldsymbol{a}\boldsymbol{x}_1+\boldsymbol{b}\boldsymbol{x}_2\\\boldsymbol{a}\boldsymbol{y}_1+\boldsymbol{b}\boldsymbol{y}_2\end{pmatrix}$$

可见，一个二维的线性变换由这 4 个数字就可以确定。 $\begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$ 和 $\begin{bmatrix} x_2 \\ y_2 \end{bmatrix}$ 就是基向量，它们是线性无关的。将这两个向量包装在一个 2×2 的方阵中，就变成了一个 2×2 的矩阵

$$\begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \end{bmatrix}$$

矩阵可以用一个字母表示，如 $\boldsymbol{A}$。该矩阵描述了一个变换的信息。对于本题，对一个向量做线性变换可以认为是矩阵向量乘法，记为 $\boldsymbol{Ap} = \begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} ax_1 + bx_2 \\ ay_1 + by_2 \end{bmatrix}$。

有一些有趣的矩阵变换，例如：

1. 将向量逆时针旋转 90°：$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$。
2. 将向量延长至 2 倍：$\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$。
3. 维持向量不变：$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ 这被称为单位矩阵。
4. 变成零向量：$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$，这被称为零矩阵。

第 4 点说明如果矩阵存在一列和其他的列线性相关（也就是一列对应的向量可以由其他列向量使用数乘和加法得到）时，就会出现“降维打击”，损失一些信息。读者可以尝试使用矩阵 $\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$ 来变换一个向量，看看会出现什么后果。

### 例 22-3 矩阵乘法

描述这样一种作用：一种变化之后再进行另外一种变换，例如，将向量的 $\boldsymbol{X}$ 轴分量拉长为原来的 $\mathbf{2}$ 倍，$\boldsymbol{Y}$ 轴分量拉长为原来的 $\mathbf{3}$ 倍，然后将其逆时针旋转 $\mathbf{90}$。 ，其变换过程如图 22-4 所示。

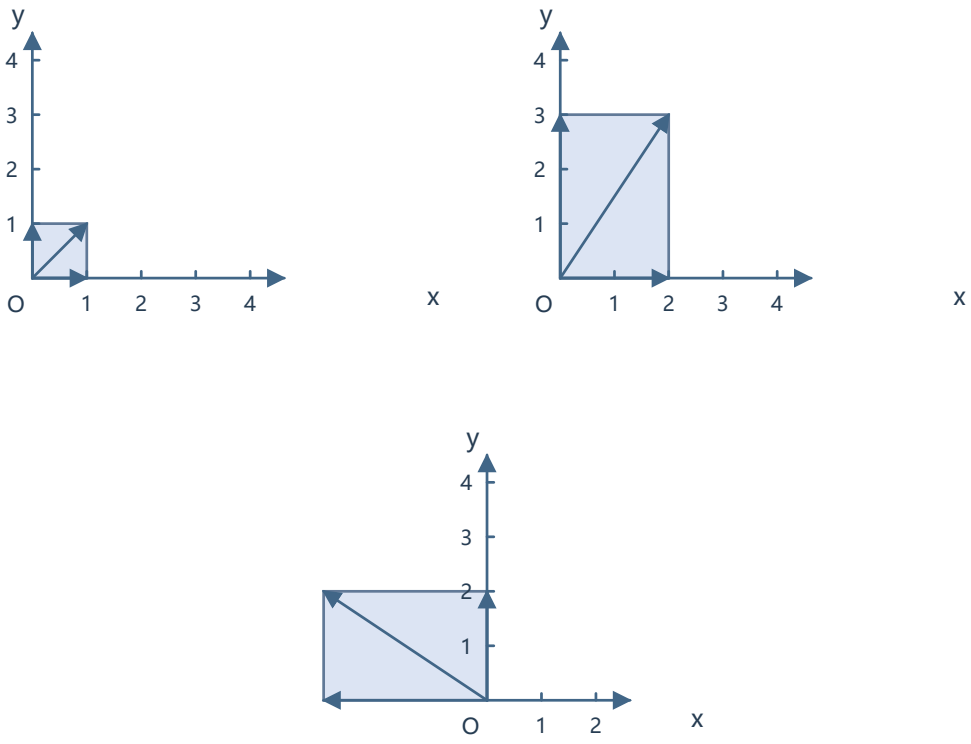



图22-4矩阵变换

**分析**：选择 $\begin{bmatrix}1\\0\end{bmatrix}$、$\begin{bmatrix}0\\1\end{bmatrix}$ 为基向量。依次施加伸缩和旋转后，它们分别变为 $\begin{bmatrix}0\\2\end{bmatrix}$、$\begin{bmatrix}-3\\0\end{bmatrix}$，因此复合变换的矩阵为 $\begin{bmatrix}0&-3\\2&0\end{bmatrix}$。这个结果也可以通过两个变换矩阵相乘得到。

令伸缩矩阵 $\boldsymbol{A}=\begin{bmatrix}2&0\\0&3\end{bmatrix}$，旋转矩阵 $\boldsymbol{B}=\begin{bmatrix}0&-1\\1&0\end{bmatrix}$。对向量 $\boldsymbol{p}=\begin{bmatrix}a\\b\end{bmatrix}$ 先施加 $\boldsymbol{A}$ 再施加 $\boldsymbol{B}$，可写成：

$$\boldsymbol{B}(\boldsymbol{A}\boldsymbol{p})=\begin{bmatrix}0&-1\\1&0\end{bmatrix}\left(\begin{bmatrix}2&0\\0&3\end{bmatrix}\begin{bmatrix}a\\b\end{bmatrix}\right)=\begin{bmatrix}0&-3\\2&0\end{bmatrix}\begin{bmatrix}a\\b\end{bmatrix}$$

根据这样的说法，将连续多次矩阵变化称为矩阵乘法，即

$$\boldsymbol{B}\boldsymbol{A}=\begin{bmatrix}0&-1\\1&0\end{bmatrix}\begin{bmatrix}2&0\\0&3\end{bmatrix}=\begin{bmatrix}0&-3\\2&0\end{bmatrix}$$

注意：矩阵乘法是从右往左读的，因为先应用右侧的矩阵，然后再应用左侧的矩阵。

矩阵乘法的结果是怎么计算的呢？在分析的最开始已经暗示了做法：考虑基向量的变换结果，然后组成一组新的基向量。

对于任意两个 2×2 的矩阵 $\boldsymbol{B}=\begin{bmatrix}a&b\\c&d\end{bmatrix},\boldsymbol{A}=\begin{bmatrix}e&f\\g&h\end{bmatrix}$。

根据定义，基向量 $\begin{bmatrix}1\\0\end{bmatrix}$ 和 $\begin{bmatrix}0\\1\end{bmatrix}$ 经过矩阵 $\boldsymbol{A}$ 的变换后分别会变为 $[\boldsymbol{L}_g^e]$ 和 $\begin{bmatrix}f\\h\end{bmatrix}$。

将前者继续进行矩阵 $\boldsymbol{B}$ 的变换，产生了新的向量 $\begin{bmatrix}ae+bg\\ce+dg\end{bmatrix}$。同理，另一个基向量变换后的矩阵是 $\begin{bmatrix}af+bh\\cf+dh\end{bmatrix}$。

将这两个变换后的向量写在一起，就是矩阵乘法的结果：

$$BA = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}$$

矩阵乘法不满足交换律，也就是BA不一定等于 $AB$。读者可以想象：先将一个向量往 $Y$ 轴方向拉伸到 2 倍，再逆时针旋转 $90$。 ，和先将原向量逆时针旋转 $90$。 ，再往 $Y$ 轴方向拉伸到 2 倍的结果是不一样的。但矩阵乘法满足结合律，即 $(AB)C = A(BC)$，这个性质非常重要。

对于更高维的矩阵。虽然会复杂一些，但本质上是一样的。即使是两个行列数不相同的矩阵，在一定情况下也可以做矩阵乘法，但本章不讨论。

计算 $L$ 维矩阵 $C = AB$ 的乘法，其结果矩阵 $C$ 的第 $i$ 行第 $i$ 列的元素就是

$$C_{i,j} = \sum_{k=1}^{L} A_{i,k} B_{k,j}$$

其实就是前面的矩阵对应的是行，后面的矩阵对应的是列，依次两两相乘然后累加后的结果，如图 22-5 所示。

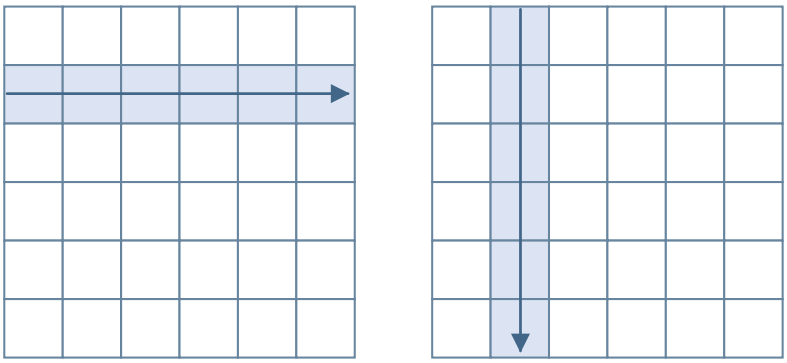


图22-5 矩阵乘法的过程

编写程序记录矩阵时，可以使用二维数组模拟。为了方便使用，可以将其封装成一个结构体。可以重载*运算符，并使用三重循环进行矩阵乘法的操作，这样进行矩阵乘法就像计算整数乘法一样方便。代码如下：

```
struct Matrix { // 矩阵
    long long M[MAXN][MAXN];
    void clear() { // 矩阵清零
        memset(M, 0, sizeof(M));
    }
    void reset() { // 设置为单位矩阵
        clear();
        for (int i = 0; i < L; ++i)
            M[i][i] = 1;
    }
    Matrix friend operator*(const Matrix &A, const Matrix &B) {
        Matrix Ans;
        Ans.clear(); // L是矩阵的大小(行数与列数)
        for (int i = 0; i < L; ++i)
            for (int j = 0; j < L; ++j)
                for (int k = 0; k < L; ++k)
                    Ans.M[i][j] = (Ans.M[i][j] + A.M[i][k] * B.M[k][j]) % Mod;
        return Ans;
    }
};
// 可以这么计算矩阵乘法,其中A、B、C均为Matrix类型
C = A * B;
```

### 例 22-4 斐波那契数列(洛谷P1962)

斐波那契数列是满足如下性质的一个数列：

$$F_n = \begin{cases} 1 & (n \leqslant 2) \\ F_{n-1} + F_{n-2} & (n \geqslant 3) \end{cases}$$

请求出 $F_n \bmod 10^9 + 7$ 的值。保证 $1 \leqslant n < 2^{63}$

**分析：**这是一个简单的递推问题，但由于 $n$ 的数据范围极大，直接一项项递推会超时。这个时候可以考虑使用矩阵加速递推过程。在前面的内容中，已经介绍过矩阵可以将一个向量进行线性变换，成为另外一个向量。这种递推式具有很明显的线性关系，可以使用矩阵来帮助递推过程。

考虑当 $n \geqslant 3$ 时，每次变换都和前两次的状态有关，所以如果有矩阵可以把向量 $\binom{F_{n-1}}{F_{n-2}}$ 变换为 $\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix}$ 就好了。

可以使用待定系数法，假设 $\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix}$，则容易解出这个矩阵是 $A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$。

将初始向量进行一次变换，可以得到 $\begin{bmatrix} F_3 \\ F_2 \end{bmatrix} = A \begin{bmatrix} 1 \\ 1 \end{bmatrix}$。

使用数学归纳法，就可以得知 $\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = A^{n-2} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

矩阵乘法满足结合律，所以可将矩阵的幂乘使用快速幂的方式来处理。最终代码如下：

```
#include <bits/stdc++.h>
#define Mod 1000000007
#define MAXN 3
using namespace std;
long long n, L = 2;
struct Matrix { // 矩阵
    long long M[MAXN][MAXN];
    void clear() { memset(M, 0, sizeof(M)); }
    void reset() {
        clear();
        for (int i = 0; i < L; ++i)
            M[i][i] = 1;
    }
    Matrix friend operator*(const Matrix &A, const Matrix &B) {
        Matrix Ans;
        Ans.clear();
        for (int i = 0; i < L; ++i)
            for (int j = 0; j < L; ++j)
                for (int k = 0; k < L; ++k)
                    Ans.M[i][j] = (Ans.M[i][j] + A.M[i][k] * B.M[k][j]) % Mod;
        return Ans;
    }
} A;
Matrix expow(Matrix T, long long P) { // 矩阵快速幂
    Matrix Ans;
    Ans.reset();
    while (P) {
        if (P & 1)
            Ans = Ans * T;
        T = T * T, P >>= 1;
    }
    return Ans;
}
int main() {
    cin >> n;
    if (n <= 2) {
        cout << 1;
        return 0;
    }
    A.M[0][0] = 1;
    A.M[0][1] = 1; // 构造递推矩阵
    A.M[1][0] = 1;
    A.M[1][1] = 0;
    A = expow(A, n - 2);                    // 矩阵快速幂
    cout << (A.M[0][0] + A.M[0][1]) % Mod; // 计算结果
    return 0;
}
```

## 22.2 高斯消元

### 例 22-5 线性方程组(洛谷P2455,山东省队选拔2006)

已知 $n$ 元线性一次方程组

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n = b_1, \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n = b_2, \\ \quad \vdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n = b_n. \end{cases}$$

请根据输入的 $a_{i,j}$ 和 $b_i$，输出方程组解的情况：给出唯一解，或告知是无穷多组解或者无解。数据范围： $1 \leqslant n \leqslant 50$ 对于 $\forall 1 \leqslant i,j \leqslant n$，有 $a_{i,j}| \leqslant 100, |b_i| \leqslant 300$。

**分析：**方程组就是和未知量相关的一组方程，需要同时满足约束条件。线性方程组的每一个方程中，每一个未知量都是一次的。也就是说，每一个方程都是各个未知量乘以一个常数项的总和。习惯上把关于每个未知量相关的项放在每个等于号的左边，而把常数项放在等号的右边。

这个形式看起来有一点复杂，可以拿一个稍微简单一点的三元一次方程来作为一个例子。求解以下方程组：

$$\begin{cases} x + y + 4z = 19 \\ 5x + 3y = 25 \\ 2y + 3z = 19 \end{cases}$$

将未知变量竖直对齐，如果未知变量不存在，也把它写上（系数为 0），可以变成这样：

$$\begin{cases} x + y + 4z = 19, \\ 5x + 3y = 25, \\ 2y + 3z = 19. \end{cases}$$

这看起来漂亮多了。也许读者注意到了，看起来就是一个 3×3 矩阵乘上一个 3 项的向量，也就是

$$\begin{bmatrix} 1 & 1 & 4 \\ 5 & 3 & 0 \\ 0 & 2 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 19 \\ 25 \\ 19 \end{bmatrix}$$

将左边的系数矩阵称为系数矩阵 $A$，中间的未知量向量 $x$ 和右边的常数向量 v，这个线性方程组就可以缩为 $Ax = \nu$。这种形式看起来非常简洁，但也有一个很棒的几何意义：寻找一个未知向量 $x$，将它经过矩阵 $A$ 的线性变换后，可以变为常数向量 $\nu_c$。

求解二元一次方程使用的主要方法就是加减消元法和代入消元法：将方程组的其中一个方程的所有系数都乘上一个常数，加到另外一个方程中，方程仍然成立；得到一个未知量的解后，将其代入到其他方程中，可以解出其他未知量的解。现在，尝试求解这个方程。

首先，将系数矩阵提取出来，然后将常数向量写在系数矩阵的右边，成为增广矩阵。将第 1 行乘上若干倍，使第 1 行第 1 个数字变为 1（这里本来就是 1 了），像这样：

$$\begin{bmatrix} 1 & 1 & 4 & 19 \\ 5 & 3 & 0 & 25 \\ 0 & 2 & 3 & 19 \end{bmatrix}$$

然后模拟加减消元法。将第 1 行分别乘上若干倍，加到后面的每一行中，使得后面的每一行的第 1 个数字都为零。这里每一行矩阵第 1 个非零元素对应的未知量被称为主元。在本题中，将第 1 行的每一项乘上-5 后加到第 2 行，乘上 0 后加到第 3 行（第 3 行第 1 个数字本来就是 0 了），变成如下：

$$\begin{bmatrix} 1 & 1 & 4 & 19 \\ 0 & -2 & -20 & -70 \\ 0 & 2 & 3 & 19 \end{bmatrix}$$

将第 2 行乘上若于倍，使第 2 行第 2 个数字（也是主元）变为 $1$。然后将第 2 行乘上若干倍，加到其余的每一行中，使得其他每一行第 2 个数字都变为零。将第 2 行乘上-1 加到第 1 行，乘上-2 后加到第 3 行，得到

$$\begin{bmatrix} 1 & 1 & 4 & 19 \\ 0 & 1 & 10 & 35 \\ 0 & 2 & 3 & 19 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & -6 & -16 \\ 0 & 1 & 10 & 35 \\ 0 & 0 & -17 & -51 \end{bmatrix}$$

将第 3 行乘上若干倍，使第 3 行第 3 个数字（还是主元）变为 $1$。然后将第 3 行乘上若干倍，加到其余的每一行中，使得其他每一行第 3 个数字都变为零。将第 3 行乘上 6 加到第 1 行，乘上-10 后加到第 2 行。这个过程得到的矩阵为

$$\begin{bmatrix} 1 & 0 & -6 & -16 \\ 0 & 1 & 10 & 35 \\ 0 & 0 & 1 & 3 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 1 & 3 \end{bmatrix}$$

第 3 行就是最后一行，所以结束加减消元法的流程。最后得到的矩阵，去掉最右边的对应的常数向量，可以发现其对角线都是 1，是个单位矩阵。直接使用这个单位矩阵乘上未知量向量，其结果就是常量向量，就可以得到这个方程组的解：

$$\begin{cases} x = 2 \\ y = 5 \\ z = 3 \end{cases}$$

有时候在处理第 $i$ 行时，可能出现本应当是主元位置（假设是第 $k$ 列）的矩阵元素刚好是 0 的情况。这时可以往下搜索其他行的第 $k$ 列的数字，找到非 0 元素，将其整行和第 $i$ 行交换。如果第 $i$ 行第 $k$ 列和下方的元素均为 0，则尝试处理下一列的消元。例如，对于如下的增广矩阵：

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 1 \\ 0 & 0 & 0 & 2 & 3 & 1 \\ 0 & 0 & 0 & 2 & 3 & 1 \\ 0 & 0 & 2 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

这个增广矩阵已经是处理完第 1 行的情况，准备处理第 2 行。但是发现第 2 行第 2 列和它下面的元素都是零，只好处理第 2 行第 3 列。发现第 4 行第 3 列的元素非零，所以将第 4 行和第 2 行进行交换，然后继续消元。

如果消到最后，发现了类似于 $0x_i = 0$ 的情况（某一行全是 0），说明 $x_i$ 取任意实数都可以，有无穷多组解。如果出现了 $0x_i \neq 0$ 的情况，那就输出无解。

如果大循环是处理每一行，可能在同一行中需要处理多个列，有点麻烦。所以可以按照列来处理，同时记录当前处理到了第几行。代码如下：

```
#include <bits/stdc++.h>
using namespace std;
#define MAXN 110
#define eps 1e-6
int n, curi = 0; // curi 表示枚举哪一行
double a[MAXN][MAXN];
int main() {
    cin >> n;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n + 1; j++)
            cin >> a[i][j];
    for (int j = 0; j < n; j++) {
        int t;
        for (t = curi; t < n; t++)
            if (fabs(a[t][j]) > eps)
                break; // 找到一列非零元素
        if (t == n)
            continue; // 这一列没有找到非零元素
        for (int i = j; i <= n; i++)
            swap(a[t][i], a[curi][i]); // 把非0元素所在行交换到当前行
        for (int i = n; i >= j; i--)
            a[curi][i] /= a[curi][j]; // 当前行同除,令主元系数化一
        for (int i = 0; i < n; i++)   // 对其他行消元
            if (i != curi)
                for (int k = n; k >= j; k--)
                    a[i][k] -= a[curi][k] * a[i][j];
        curi++;
    }
    if (curi < n) { // 判断0!=0
        for (int i = curi; i < n; i++)
            if (fabs(a[i][n]) > eps) { // 发现0!=0,无解
                puts("-1");
                return 0;
            }
        puts("0"); // 发现0=0,无穷多解
    } else
        for (int i = 0; i < n; i++)
            printf("x%d=%.2lf\n", i + 1, a[i][n]);
    return 0;
}
```

需要注意的是，因为精度误差，比较两个浮点数是否相等不应当直接使用==，而是判断其差的绝对值是否小于一个很小的常数（这里取 $10^{-6}$）。

### 例 22-6 行列式与秩

在直角坐标系中，将和这两个基向量作为一个平行四边形（其实是一个正矩形）的 $\begin{bmatrix}1\\0\end{bmatrix}$ $\begin{bmatrix}0\\1\end{bmatrix}$ 两条边。将它经过以下 $3$ 个矩阵的变换。变换后的一对新向量作为一个新的平行四边形的两条边，这些平行四边形的面积是原来的平行四边形的面积几倍？

1. $\begin{bmatrix}0 & -1\\1 & 0\end{bmatrix}$

（2）$\begin{bmatrix}2 & 1\\0 & 3\end{bmatrix}$

（3）$\begin{bmatrix}1 & 2\\ 1 & 2\end{bmatrix}$

**分析：**将这两个单位基向量进行矩阵变换，其结果就是这个矩阵各列组成的向量，其组成的平行四边形如图 22-6 所示。经过第 1 个矩阵变换后，原来的正矩形逆时针旋转了 90°，面积 **90**。

不变。经过第 2 个矩阵变换后，面积变成了原来的 6 倍。而第 3 个矩阵，变换后的两个向量共线（实际上，将任何向量经过这个矩阵的变换，其终点都会落在直线 $y=x$ 上），组成的平行四边形的面积自然就是 0 了。

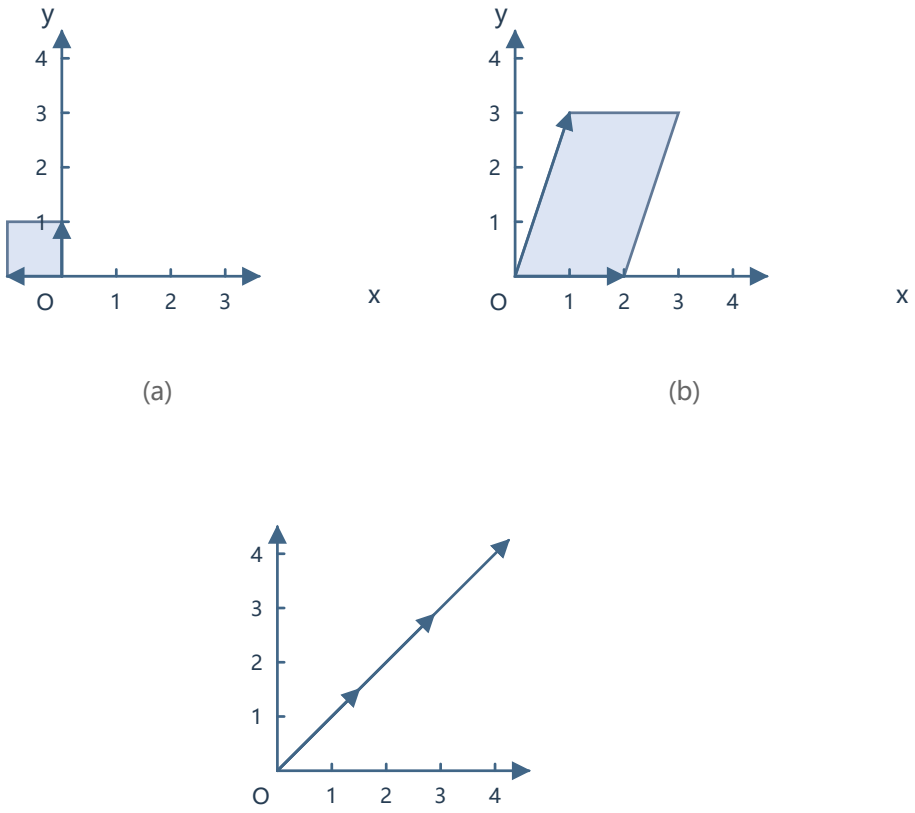



图22-6一些矩阵变换的例子

像这样，在二维坐标系上将一个图形经过一个矩阵的变换后得到的新图形面积和原图形面积的比值（三维情况下，就是体积的比值，更高维的……可以想象一下），就是这个矩阵的行列式的值。对于矩阵 $A=\begin{bmatrix}0 & -1\\ 1 & 0\end{bmatrix}$，它的行列式记为

$$\det(A)=\begin{vmatrix}0 & -1\\ 1 & 0\end{vmatrix}=1$$

对于一个二维矩阵 $\begin{bmatrix}a & b\\ c & d\end{bmatrix}$，其行列式 $\begin{vmatrix}a & b\\ c & d\end{vmatrix}=ad-bc$。

行列式的值是可能存在负数的，例如 $\begin{vmatrix}0 & 1\\ 1 & 0\end{vmatrix}=-1$，这和变换后的向量方向有关。对于这个矩阵来说，相当于将 $X$ 轴和 $Y$ 轴相互对调原来的正矩形翻了一个面。而更高维的空间中，也有类似的情况①。

至于更高维的行列式如何计算，请读者自行查阅相关资料。

当二维矩阵的行列式等于 0 时，就会出现“降维打击”的情况，如同本题的第 3 个矩阵。此时观察向量的变换式：

$$\begin{bmatrix}1 & 2\\1 & 2\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}=\begin{bmatrix}x+2y\\x+2y\end{bmatrix}$$

可以发现，无论 $x$ 和 $y$ 如何取值，其变换后的向量都落在一条直线上。这说明这个变换将二维空间映射到了一维空间上，产生了“降维打击”。

考虑如下线性方程组：

$$\begin{bmatrix}1 & 2 & 2\\2 & 1 & 1\\1 & 1 & 1\end{bmatrix}\begin{bmatrix}x\\y\\z\end{bmatrix}=\begin{bmatrix}a\\b\\c\end{bmatrix}\iff\begin{cases}x+2y+2z=a,\\2x+y+z=b,\\x+y+z=c.\end{cases}$$

可以发现，第 3 个方程等号左边的部分是与第 1 行和第 2 行的等号左边部分线性相关的，第 3 个方程就是一个凑数的“废话”，并不能起到作用。使用消元操作，可以将线性方程组变为：

> **注：** 1. 例如在三维空间中，初始的直角坐标系的 3 个轴，是符合“右手定则”的：将右手食指指向左边，中指指向自己，拇指指向上方，分别对应 $X$ 轴、$Y$ 轴和 $Z$ 轴。如果变换后的行列式值变为负数，则对应的坐标轴的顺序需要用左手表示。

$$\begin{bmatrix}1 & 0 & 0\\0 & 1 & 1\\0 & 0 & 0\end{bmatrix}\begin{bmatrix}x\\y\\z\end{bmatrix}=\begin{bmatrix}-\frac{1}{3}a+\frac{2}{3}b\\\frac{2}{3}a-\frac{1}{3}b\\-\frac{1}{3}a-\frac{1}{3}b+c\end{bmatrix}$$

像这样，尽可能把各行使主元的系数变成 1.发现只有两行是真正起到作用的，因此原矩阵的秩就是 2。如果 $n\times n$ 的矩阵的秩等于 $n$。则说明这个矩阵是满秩的。使用满秩矩阵进行变换不会出现“降维打击”的情况，其行列式的值非零。读者从本题可以看出，对线性方程组进行**高斯消元**后，如果原系数矩阵不满秩，会出现无解或者是无穷多解的情况。

至此，《深入浅出程序设计竞赛（进阶篇）》的正文部分就结束了。如果读者认真学习完本书的大多数内容，并完成大量的练习，相信实力也会变强很多。当然，算法的攀登是永无止境的。希望读者继续保持对新知识的求知欲，并通过自己的努力，朝着更高的山峰进发！

## 22.3 课后习题与实验

### 习题 1 向量(洛谷P2520,河南省队选拔2011)

给定一对数 $a,b$，可以使用 $(a,b),(a,-b),(-a,b),(-a,-b),(b,a),(b,-a),(-b,a),(-b,-a)$ 这些向量，数量不限。问能否选出若干向量，使它们的和为 $(x,y)$。

**数据范围：** 不超过 $50000$ 组数据，且 $-2\times10^9\leqslant a,b,x,y\leqslant2\times10^9$。

### 习题 2 广义斐波那契数列(洛谷P1349)

广义的斐波那契数列是指形如 $a_n=p\times a_{n-1}+q\times a_{n-2}$ 的数列。给定数列的两系数 $p$ 和 $q$，以及数列的最前两项 $a_1$ 和 $a_2$，另给出两个整数 $n$ 和 $m$，试求数列的第 $n$ 项 $a_n \bmod m$ 所有输入均为整数。

数据范围 $:p,q,a_1,a_2\in\left[0,2^{31}-1\right],1\leqslant n,m\leqslant2^{31}-1$

### 习题 3 矩阵加速(洛谷P1939)

已知一个数列 $a$ 满足：

$$a_x=\begin{cases}1, & x\in\{1,2,3\},\\ a_{x-1}+a_{x-3}, & x\geq 4.\end{cases}$$

求 $a$ 数列的第 $n$ 项对 $10^9+7$ 取余的值。

**数据范围：** $1\leqslant T\leqslant 100, 1\leqslant n\leqslant 2\times 10^9$。

### 习题 4 千足虫(洛谷P2447)

求解关于 $x_i$ 的方程组：

$$\begin{cases}\sum_{i=1}^{n}a_{1,i}x_i\equiv b_1 \pmod 2,\\ \sum_{i=1}^{n}a_{2,i}x_i\equiv b_2 \pmod 2,\\ \quad\vdots\\ \sum_{i=1}^{n}a_{m,i}x_i\equiv b_m \pmod 2.\end{cases}$$

其中，$\mathbf{\Delta}_{a_{i,j}}$ 和 $b_i$ 均为 $0$ 或者 $1$。输出获得前面几个这样的方程就可以得到唯一解，并输出各解。如果解不唯一，则输出Cannot Determine。

**数据范围：** $1\leqslant N\leqslant 10^3, 1\leqslant M\leqslant 2\times 10^3$。

### 习题 5 行列式求值(洛谷P7112)

给定一个 $n$ 阶行列式 $A$，求 $|A|$。结果对 $p$ 取模。

**数据范围：** $1\leqslant n\leqslant 600, 1\leqslant a_{i,i}<10^9+7, 1\leqslant p\leqslant 10^9+7$。

### 习题 6 数学作业(洛谷P3216,湖南省队选拔2011)

给定正整数 $n$ 和 $m$，要求计算 `C(n)` mod $m$ 的值，其中 `C(n)` 是将 $1\sim n$ 所有正整数按顺序连接起来得到的数。例如 $n=13, \mathrm{C}(n)=12345678910111213$。

**数据范围：** $1\leqslant n\leqslant 10^{18}, 1\leqslant m\leqslant 10^9$。

### 习题 7 小莉的手环(洛谷P4910)

给定一个由 $n(n\leqslant 10^{18})$ 颗绿色和金色珠子串成的手环，不能有两个绿色珠子是相邻的，求方案数。

### 习题 8 去散步(洛谷P2151,山东省队选拔2009)

给定一个有 $n$ 个点、$m$ 条边的无向图，边权均为 $1$。同时给定起点和终点，以及需要走的路程 $t$。走过一条路后，不能立刻从这条路返回。请问有几种走法。

**数据范围：** $N\leqslant 50, M\leqslant 60, t\leqslant 2^{30}$

### 习题 9 球形空间产生器(洛谷P4035,江苏省队选拔2008)

有一个球形空间产生器能够在 $n$ 维空间中产生一个坚硬的球体。现在，小明被困在了这个 $n$ 维球体中，小明只知道球面上 $n+1$ 个点的坐标，需要以最快的速度确定这个 $n$ 维球体的球心坐标，以便于摧毁这个球形空间产生器。请注意以下的定义：

1. 球心：到球面上任意一点的距离都相等的点。
2. 距离：设两个 $n$ 维空间上的点 $A$ 和 $B$ 的坐标为 $(a_1,a_2,\cdots,a_n),(b_1,b_2,\cdots,b_n)$，则 AB 的距离定义为 $d=\sqrt{(a_1-b_1)^2+(a_2-b_2)^2+\cdots+(a_n-b_n)^2}$。

**数据范围：** $1\leqslant n\leqslant 10$，各点坐标值的绝对值不超过 $20000$。

### 习题 10 炸弹问题(洛谷P2973,USACO2010年公开赛)

有一由 $n$ 个点、$m$ 条边连接的无向图，结点 $1$ 有一个炸弹，在每个单位时间内，这个炸弹有 $\frac{P}{Q}$ 的概率在这个结点炸掉，剩下 $1-\frac{P}{Q}$ 的概率选择一条与当前结点相连的边到其他的结点上。求炸弹在每个结点上爆炸的概率。

**数据范围：** $2\leqslant n\leqslant 300,1\leqslant m\leqslant 44850,1\leqslant P,Q\leqslant 10^6$。

# 附录

# 附录 A 调试与对拍

将代码实现之后，提交评测之前，必须确保程序的正确性。但智者千虑必有一失，难免在代码中出现一些错误，以至于无法返回期望的结果。程序调试就是写好的程序投入实际应用前对程序进行测试并修正语法错误和逻辑错误的过程。如果程序不能返回预期的结果，就必须通过调试找到出错的位置和错误的原因。

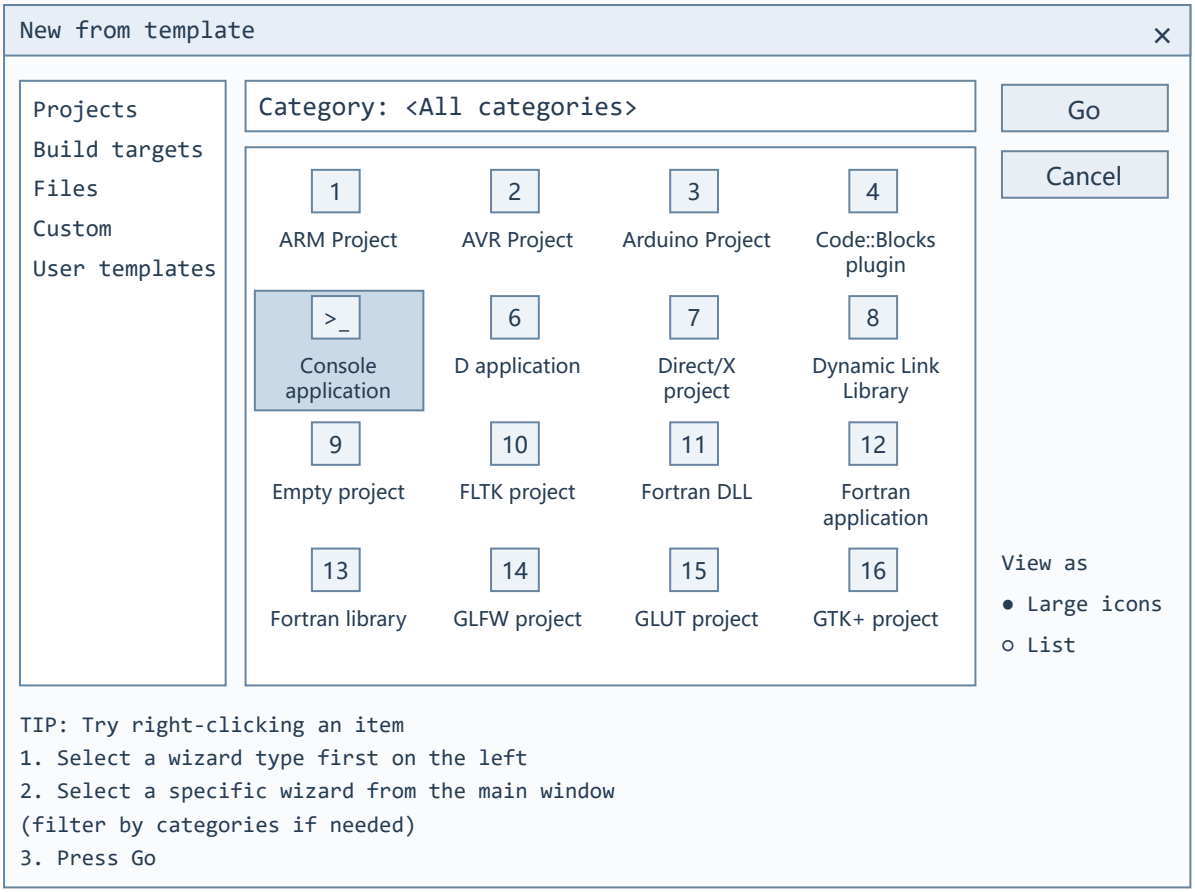
写完程序，通过了样例，这固然是好事，但还没有结束。建议从头开始，逐行审阅代码。你可以假装有一个听众（甚至是一只橡皮小黄鸭），对其讲解每一行代码的意思。这个过程中，着重检查数组范围是否不足、数据类型是否正确、是否有初始化、变量和运算符是否有错误等问题。这种方式也许就可以发现写得不对的地方，该方式叫作静态查错，又称为小黄鸭调试法。

当然，静态查错不一定可以发现所有问题，借助以下手段，可以辅助定位出错的地方。

## A.1 IDE调试

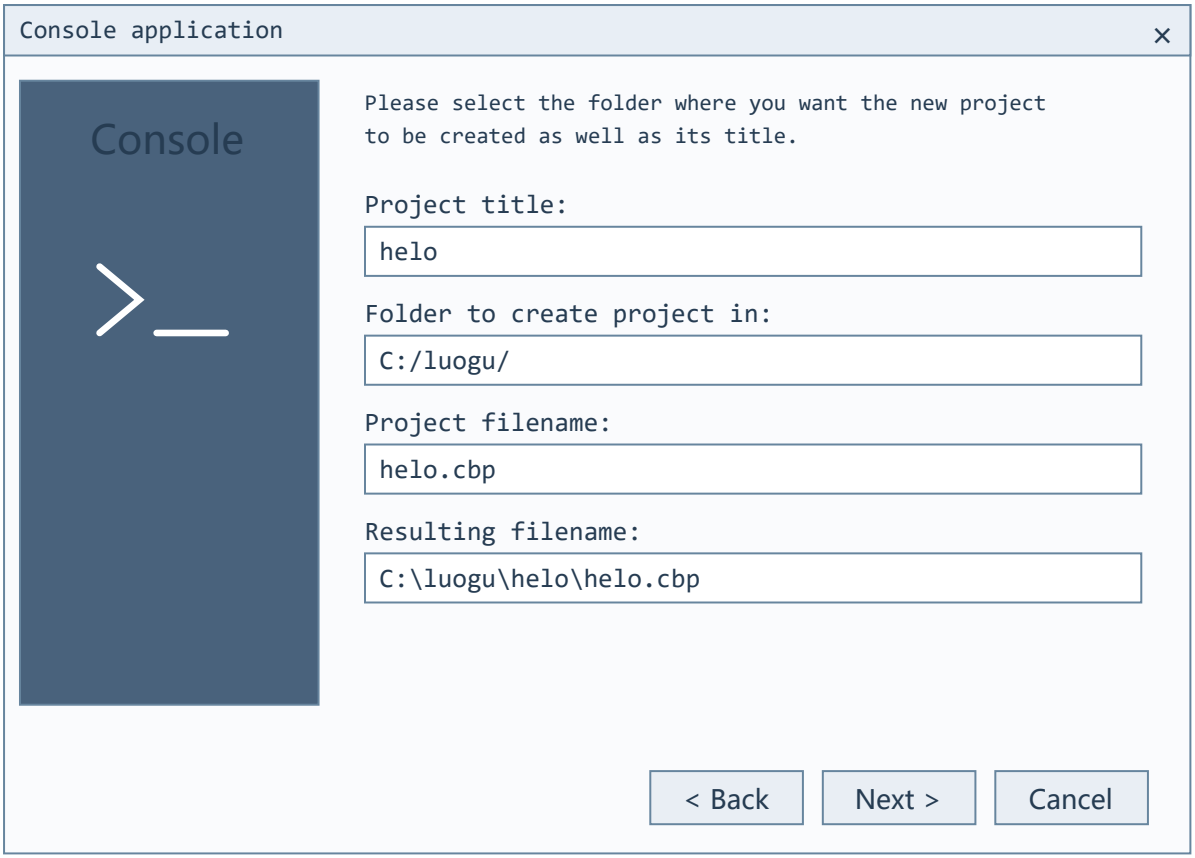
在《基础篇》介绍了如何使用CodeBlocks编译运行程序。这里还是以它为例，介绍如何使用IDE调试程序。其他IDE的调试方法大同小异。

使用CodeBlocks进行调试之前，需要先建立项目文件（而不只是单独的cpp文件）。打开CodeBlocks以后，在菜单栏中依次选择“File”→“New”→“Project”命令，在弹出的界面中间若干图标中选择“Console application”（控制台应用），如附图A-1 所示。




附图A-1 新建工程

单击“Go”按钮，然后在语言选择界面选择 C++，单击“Next”按钮，就能看到如附图A-2 所示的界面。在这个界面中填入项目标题、文件路径等，再单击两次“Next”按钮，就完成创建了。文件名和路径名称都不建议使用中文字符。




附图A-2 创建文件

创建完成后，在界面的左侧窗口可以看见当前项目的文件结构。双击文件名前的小图标可以打开这份源代码，进入编辑页面。

以下介绍如何使用CodeBlocks对程序进行调试。

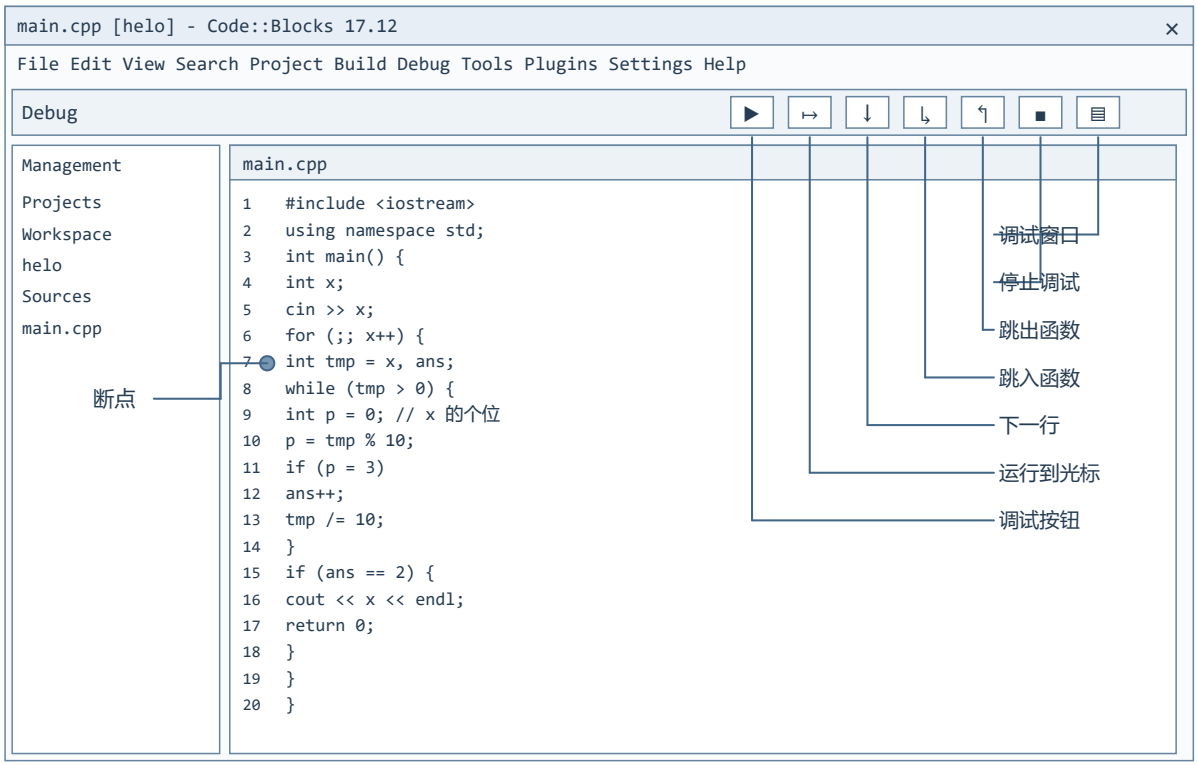
例题：输入一个不超过 $10^5$ 的正整数 $x$，输出一个不小于 $x$ 且各个数位中刚好有两个数字 3 的数字。例如，输入 200 时，应该输出 233。

本题看起来很简单，很快就可以完成如下代码。这个代码有两处错误。首先请读者借助火眼金睛，尝试通过静态查错的方式，找到这两处错误。

> **注意：** 本段是调试练习，故意保留 `ans` 未初始化和 `if (p = 3)` 的逻辑错误，供下文演示排错。

```
#include <iostream>
using namespace std;
int main() {
    int x;
    cin >> x;
    for (;; x++) {
        int tmp = x, ans;
        while (tmp > 0) {
            int p = tmp % 10;
            if (p = 3)
                ans++;
            tmp /= 10;
        }
        if (ans == 2) {
            cout << x << endl;
            return 0;
        }
    }
}
```

在菜单栏中“编译运行”按钮右侧的下拉菜单应当设置为Debug，如附图A-3 所示。编写完代码后，编译并运行这个程序，会发现它无法得到结果，因此尝试调试这个程序。




附图A-3 调试工具

单击代码编辑区的行号数字右边的位置上，就会产生一个断点。这里单击第 7 行的右侧。然后单击工具栏中的“调试”按钮（红色的三角形，或者按F8 键），就可以运行程序了，如附图A-3 所示。在弹出的控制台窗口中输入数字（如 20），按Enter键。此时可以发现程序停在了第 7 行，有箭头提示目前程序运行的位置。

这时单击工具栏的“调试窗口”按钮，单击Watches，会弹出一个小的窗口，显示全局变量和有效的局部变量的值，如附图A-4(a) 所示。

接着单击工具栏的“下一行”按钮（或者按F7 键），可以发现程序运行到了第 8 行。在Watches窗口中，可以发现 `tmp` 的值是 20，但是 `ans` 的值却是一个奇怪的数字，如附图A-4(b) 所示。这里就找到了程序的第 1 处错误—`ans` 没有初始化。

这个时候可以单击“停止调试”按钮（或者按组合键 Ctrl+F8）停止调试去修改代码，然后重新调试。不过也可以硬着头皮继续调试。多次单击“下一行”按钮，当运行到第 12 行时发现，表示个位数的变量 $p$ 的值变成了 3，和期望的不一样，如附图A-4(c) 所示。观察这一行，发现把 $p$==3 写成了 $p=3$。修正这些错误，程序也就得到了正确的运行结果。

IDE的调试还提供了其他的工具，可以协助人们进行调试。例如，“运行到光标”可以让程序运行到编辑器里的光标时停住；“跳入函数”和“跳出函数”可以快速地运行到下一个函数的入口和函数的出口；调试窗口也提供了堆栈的显示。读者可以尝试看看菜单栏中的Debug菜单里面的功能。




## A.2 输出中间变量调试

除了使用IDE调试以外，还有另一种更加方便的调试方法，即输出中间变量调试，相当于手动设置断点和Watches窗口，在需要查看变量值的时候输出对应变量，观察变量的变化是否正确即可。

输出中间变量调试同样可以定位使程序异常退出（或死循环）的语句。只需要“二分”出错位置即可。例如下述代码：

```
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= n; ++j) {
        // 做一些事情
    }
}
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= n; ++i) {
        // 做一些事情
    }
}
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= n; ++j)
    // 做一些事情
}
}
```

首先在第 6 行处输出一个字符串（如LuoGu），发现程序可以正常输出；再在第 12 行处输出一个字符串，此时程序没有输出。那么导致死循环的代码就定位在了第 7~11 行之间。此时在第 9 行处输出一个字符串，发现这里的输出不会停止，进而确定是这一循环导致了死循环，从而很容易发现第 8 行的 `++j` 写成了 `++i`。

需要注意的是，在竞赛结束前，一定要检查代码是否已经删除了全部的调试信息，否则在评测时会因为多余的输出而被判定为错误的答案。

此外，还可以使用std::cerr或 `fprintf(stderr)` 来输出到标准错误流，以输出变量 $a$[3 为例，其使用方法为 std::cerr<<`a[3]` 或fprintf（stderr，“%$d$”，`a[3]`）。此时即使忘记删除调试信息，输出文件中也不会有多余的内容。但是如果输出的内容过多，可能会造成程序超时。因此，无论输出到哪单，都应在正式提交程序前保证将调试信息全部删除。

一个更加方便的办法是，手写一个输出函数，然后使用 `printf` 输出中间变量。提交前，把输出函数的函数体注释掉或直接return。

## A.3 GDB调试

GDB是在命令行中调试程序的工具。各种IDE调试代码的能力往往来自GDB调试器。GDB功能强大，而且因为是命令行工具，熟练使用后会比使用IDE调试效率更高。

下面给出在NOILinux（本质上是Ubuntu）中使用GDB调试的例子。假设完成了如下代码，保存为 hello.cpp。

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int a;
    cin >> a;
    cout << a << endl;
    cout << "Hello World!";
    return 0;
}
```

在调试之前，必须对代码进行编译生成可执行文件，然后方可继续进行调试。进入终端（命令行环境），在终端中输入 `g++` -ghello.cpp-ohello-std=`c++`14 来进行编译。

-***g*** 命令是在编译时向其中加入GDB调试选项；-***o*** hello命令是编译后生成名称为hello的可执行文件，-std=`c++`14 命令是表示按照 C++14 标准编译。如果读者的 `g++` 版本过低，可能不支持 C++14 标准编译，可以将 C++14 替换为 C++11，但是这里更建议升级 `g++` 到最新版本。

```
noilinux@ubuntu: \~\$ cd Desktop
noilinux@ubuntu: \~Desktop\$ g++ -g hello.cpp -o hello -std=c++14
noilinux@ubuntu: \~Desktop\$
```

当目录下生成可执行文件后，表明编译已经正常结束。此时输入./hello即可运行编译后的文件。

在终端中输入gdb hello即可进入GDB调试界面。

```
noilinux@ubuntu: \~\$ cd Desktop
noilinux@ubuntu: \~/Desktop\$ g++ -g hello.cpp -o hello
noilinux@ubuntu: \~Desktop\$.hello
3
```

```
3
Hello World!noilinux@ubuntu: \~/Desktop \$ gdb hello
GNU gdb (Ubuntu 9.2-0ubuntu1\~20.04) 9.2
#此处是版权和说明信息,略去
Reading symbols from hello...
(gdb)
```

输入run（或 ***r***）即可使程序运行，但是会一直运行至程序结束。

此时，可以为程序设置断点，程序到达断点就会自动暂停运行。设置断点遵循附 表A-1 的格式进行：

| 格式 | 说明 |
| --- | --- |
| break 函数名 | 对当前正在执行的文件中的指定函数设置断点 |
| break 行号 | 对当前正在执行的文件中的指定行设置断点 |
| break 文件名:行号 | 对指定文件的指定行设置断点 |
| break 文件名:函数名 | 对指定文件的指定函数设置断点 |
| break +/-偏移量 | 对当前指令行处 +/- 偏移量处设置断点 |
| break *地址 | 对指定地址处设置断点 |

附表 A-1 break 函数功能

应当注意的是，break可以被简写为 ***b***。

设置好的断点可以通过infobreak查看（该命令也可被简写为ib）。例如，b10 指令可以在程序的第 10 行（也即 `cout` <<***a***<< `endl` ; ）设置断点，然后输入 ***r*** 以运行程序。

```
(gdb) b 10
Breakpoint 1 at 0x1237: file hello.cpp, line 10.
(gdb) r
Starting program: /home/noilinux/Desktop/hello
```

此时程序正等待输入。

```
(gdb) b10
Breakpoint 1 at 0x1237: file hello.cpp, line 10.
(gdb) r
Starting program: /home/noilinux/Desktop/hello
10
Breakpoint 1, main () at hello.cpp: 10
10 cout<<a<<endl;
(gdb)
```

输入数据后，程序在断点处停下，可以使用print $a$（或 $p$ $a$）命令来查看变量 $a$ 的值。

```
(gdb) b 10
```

```
Breakpoint 1 at 0x1237: file hello.cpp, line 10.
(gdb)r
Starting program:/home/noilinux/Desktop/hello
10
Breakpoint 1, main () at hello.cpp: 10
10 cout<<a<<endl;
(gdb) print a
\$1 = 10
(gdb)
```

类似的，可以使用displaya（或disp $a$）、infolocals（或 $i$ lo）命令来查看变量 $a$ 的值。值得注意的是，infolocals命令是查看所有局部变量的情况。

如果想查看断点附近的代码，可以使用list（或 1）命令。相似的，也可以使用 1 行号或 1 函数名以查看特定行或特定函数附近的代码。

如果想要程序继续运行，可以输入continue（或 $c$）继续。

如果想要逐条语句地执行程序，则频繁地使用 $b$ 和 $c$ 命令会过于烦琐，gdb提供了两个更加方便的指令——next（或 $n$）和step（或 $s$）。二者的区别在于如果当前行存在函数调用，next会将函数作为一个整体执行完毕，而step只会进入函数内部。

即便如此，不断输入 $n$ 或 $s$ 依然会让人感到厌烦。在gdb命令中，如果在提示符下直接按下Enter键，等价于执行上一条命令。

此外，另一个常用命令为until（或 $u$），通过 $u$ 行号或 $u$ 函数名使得程序在运行到特定行或特定函数处暂停运行。

通过clear（或cl）命令可以清除之前设置的断点，使用方法与 $b$ 命令一致。各命令及其功能见附 表 A-2。

| 简写 | 全称 | 备注 |
| --- | --- | --- |
| l | list | 显示特定行或特定函数附近的源代码 |
| b | break | 在特定行或特定函数开头处设置断点 |
| r | run | 运行程序，直到程序结束或遇到断点而停下 |
| c | continue | 在程序中断后继续执行程序，直到程序结束或遇到断点而停下；在程序执行前只能使用 r 而不能使用 c |
| n | next | 执行一条语句，如果有函数调用，则将其当作一个整体运行 |
| s | step | 执行一条语句，如果有函数调用，则将停在函数开头处 |
| u | until | 执行到特定行或者特定函数的开头 |
| p | print | 显示变量或表达式的值 |
| disp | display | 把一个变量或表达式设为 display，每次程序停下时都会输出其值 |
| cl | clear | 取消断点，与 b 的格式相同；如果该位置有多个断点则同时取消 |

附表A-2 命令的全称及其作用

| 简写 | 全称 | 备注 |
| --- | --- | --- |
| i | info | 显示各种信息：i b 显示所有断点；i display 显示所有 display；i locals 显示所有局部变量 |
| wa 变量名 | watch 变量名 | 变量发生变化时停止程序，并输出变化前后的值 |
| aw 变量名 | awatch 变量名 | 变量被读或写时停止程序运行 |
| rw 变量名 | rwatch 变量名 | 变量被访问时停止程序运行 |

GDB 常用命令（续）

如果对某个命令有疑问，可以输入help以获得详细的帮助信息。

如果选手使用的是Windows系统，只需要在源代码目录下打开命令行（打开对应目录下的命令行可以参考附录的“命令行”章），使用 $\mathbf{\Pi_g + + hello.\ cpp - o\ hello.\ exe\ -g\ -std = c + +14}$ 进行编译，然后输入gdbhello. exe即可进入gdb命令。调试的方式和 Linux是相同的。

## A.4 对拍

对拍是竞赛中用来检验程序正确性的方法，一般来说需要 3 个程序：根据题意所写的保证正确的暴力程序、数据生成程序、需要检验的程序。这里需要检验的程序将会被提交，但目前还不知道是否存在错误。

首先使用数据生成程序生成一组数据，然后用暴力和被检验程序分别运行该数据得到结果，比较两个程序的输出，如果不同，则说明有一个程序错误了。这个方法还可以用来找到使解法错误的数据，或者对某些难以证明的贪心算法的验证（对拍很多次没有出现问题的话，贪心策略大概率是对的）。

首先介绍两个用于数据生成的函数：

1. 函数 `time()`：用法为 `time(0)`，返回的是当前时刻距某个标准时间经过的（在模意义下）的秒数。这个数字一秒一变，是方便的随机数种子，需要头文件ctime。
2. 函数 `rand()`：返回一个随机数。这个随机数的范围在Windows下是 $[0, 2^{15}-1]$，而在Linux下是 $[0, 2^{31}-1]$。需要头文件cstdlib。在程序的开头（不要在别的地方重复调用后面的代码），需要调用一次srand（`time(0)`）来设定随机数种子，不然每次运行生成的随机数将相同。

以高精度加法问题为例，写出如下的数据生成代码，兼容较旧版本的Windows下的 `g++` 编译器，使用 C++98 标准：

```
#include <ctime>
#include <cstdio>
#include <cstdlib>
#define MAXN 10000000
int main() {
    srand(time(0));
    freopen("data.in", "w", stdout);
    long long a = (1ll * rand() * RAND_MAX + rand()) % MAXN;
    long long b = (1LL * rand() * RAND_MAX + rand()) % MAXN;
    printf("%lld %lld\n ", a, b);
}
```

需要注意的是，`rand()`*RAND_MAX+`rand()` 可以生成一个很大随机数，但是在Windows中和Linux中的范围是不同的。这里可以产生两个 0 到 9999999 的整数，并保存为data.in文件。

如果编译器支持 C++11 以上的标准，则使用mt19937（在random头文件中）更好。

```
#include <bits/stdc++.h>
using namespace std;
#define MAXN 10000000
int main() {
    mt19937 Rand(time(0)); // 初始化随机数种子
    // Rand 是自己命名的对象
    freopen("data.in ", "w ", stdout);
    // Rand() 可以生成一个32位无符号的随机整数
    cout << Rand() % MAXN << "" << Rand() % MAXN << endl; // 输出两个范围内的整数
    return 0;
}
```

除了这种写法，mt19937 还支持各种分布的随机数生成，而且效率高，感兴趣的读者可以自行查阅相关资料。

使用 `time(0)` 会导致同一秒产生的随机数列是相同的。为了克服这个问题，在运行一次对拍程序时，可以把一个生成的随机数保存在硬盘，下一次运行时从硬盘中读取这个数作为随机数种子。

接着写一个保证正确的程序（一般是暴力枚举等时间复杂度较高的算法），从data.in读入文件，将结果输出到 data.`ans` 中：

```
#include <cstdio>
int main() {
    freopen("data.in", "r", stdin);
    freopen("data.ans", "w", stdout);
    long long a, b;
    scanf("%lld%lld", &a, &b);
    printf("%lld\n", a + b);
    return 0;
}
```

然后使用高精度代码（打算提交的正确解法的代码）输出到data.out中。

将 3 份文件放在同一目录下，在对应命令行中分别编译上述 3 份代码，并运行下面的脚本文件（Linux系统）：

```
while true; do
    gen
    bruteforce
    my
    if diff data.ans data.out; then
        echo AC
    else
        echo WA
        exit 0
    fi
done
```

其中bruteforce和my分别是暴力程序和待检验程序。当脚本退出时，就得到了错误数据；如果脚本一直不停，则说明对拍一直通过。

如果选手在竞赛时使用Windows系统，那么需要先按上述要求写出数据生成代码、暴力正确代码、待检验代码，并将它们放在同一文件夹下。使用任意文本编辑软件，输入如下命令，保存扩展名为bat的文件并退出。

```
:loop
gen.exe
bruteforce.exe
my.exe
fc data.ans data.out
if not errorlevel 1 goto loop.
```

然后运行bat文件，即可开始对拍。打开的黑色窗口一直不消失则说明对拍一直通过，而当窗口自动退出时，data.in中就是引发错误的数据。然后使用上文介绍的调试方法，修正程序的错误。

在算法竞赛的过程中，一旦一道题目取得了阶段性成果（如完成了暴力算法或者部分），最好将其复制一份。在必要时还有机会恢复这个半成品而不是推倒重来。

## A.5 使用Python制造测试数据

如果打算出题，则需要编造对应的测试数据。对于简单的测试数据，可以使用上文介绍的方式随机生成。但对于稍微复杂的情况，编写数据生成器的难度不亚于编写解决算法问题程序的难度。这种情况下可以使用Python来生成测试数据。

Python是一种脚本语言—但是读者不需要因为要学习一门新的编程语言而感到恐惧。Python学习成本很低，只要熟练掌握 C++，只需花费很短的时间学习Python的基础用法并完成实验，即可初步掌握Python。

Python代码简洁，易于编写和理解，可以使用大量的第三方库实现丰富的功能。许多Linux发行版自带Python，这意味着用户可以很方便地使用Python。当然Python运行效率较低，即使在ICPC系列比赛中允许使用Python，也不适合实现时间限制比较紧张的题目。

CYaRon是一个用于生成测试数据的Python库，内置多种数据结构，如随机图、树、向量、字符串、数列、多边形等，可以帮助生成有一定强度的测试数据。

以 洛谷P1339 为例生成一组测试数据。这个测试数据有 3 个测试点，每个测试点包括一个 $n$ 个结点、$m$ 条边的无向图，以及起始结点 $s$ 和终止结点 $t$。各个测试点的数据规模有梯度。使用Python 和CYaRon 完成的数据生成器代码如下：

```
#!/usr/bin/env python3
from random import randint
from cyaron import *  # 引入 CYaRon 库

# ati 将各元素转换为整数，便于使用 1e4 这样的数据规模。
_n = ati([0, 7, 50, 1e4])
_m = ati([0, 11, 100, 1e4])

for i in range(1, 4):  # 依次生成三组测试数据
    test_data = IO(file_prefix="heat", data_id=i)
    n = _n[i]  # 点数
    m = _m[i]  # 边数
    s = randint(1, n)  # 源点
    t = randint(1, n)  # 汇点

    # 输入数据以空格分隔，行末自动换行。
    test_data.input_writeln(n, m, s, t)
    graph = Graph.graph(n, m, weight_limit=5)
    test_data.input_writeln(graph)  # 每行输出一组 u、v、w

    # 运行已编译的标准程序；该程序应使用标准输入输出。
    test_data.output_gen(r"D:\std_binary.exe")
```

限于篇幅，本书不会介绍Python和CYaRon的使用方法，感兴趣的读者可以自己查阅相关的资料。

# 附录 B 命令行与脚本

命令行界面（Command-LineInterface，CLI）是通过输入指令使用户能够和计算机交互的方式。Windows里较为常用的是 cmd 和 powershell等，类UNIX系统（如 Linux、macOS等）则有bash和zsh等。虽然图形化界面（GUI）功能也很强大且非常直观，但仅使用命令行输入指令可以减少操作时间，而且提供了更多的专业功能。

本附录介绍cmd和bash的简单使用，并给出了一个在算法竞赛中使用命令行完成整个项目的例子。

## B.1 命令行的使用

首先需要学会打开命令行。在不同系统下有不同的打开方式。

在Windows系统中，单击“开始”菜单，搜索命令提示符并打开，对应的命令行是cmd。可以看到类似如下的界面：

```
Microsoft Windows [版本 xxxx]
(c) Microsoft Corporation。 保留所有权利
C: \Users \oier>
```

在NOILinux（本质上是Ubuntu）中按下 Ctrl+Alt+T 组合键打开终端，对应的命令行是bash。可以看到类似如下的界面：

```
oier@noilinux: ~$
```

在未输入任何命令时，所见的 C:>和oier@noilinux:~$两条语句叫作提示符。

C:>的格式是 工作目录>。其中工作目录是 C:。

oier@noilinux:~$的格式是用户名@主机名：工作目录$。用户名是oier，主机名是noil-inux，工作目录是~。

这里出现了工作目录的概念，在命令行中执行命令都是以工作目录为基础的，这会在使用中不断体现。提示符是可以自定义的，但由于和本节主要内容关系不大，这里不再详细叙述。想要让命令行做事，就要和命令行交互。一般情况下和命令行交互靠键盘输入。每次操作需要输入一行指令，按Enter键后指令会被执行。

指令的格式如下：

```
// 命令[参数1参数2参数3...]
```

参数个数是任意的。例如，在NOILinux下bash中用 `g++` 编译程序并运行：

```
oier@noilinux: \~ \$ g++ edge.cpp -o edge
oier@noilinux: \~\$./edge
```

其中，`g++` 是命令，edge.cpp，***-o*** 和edge都是参数。/edge也是命令，但是并未提供任何参数。

在cmd中输入cd，可以获得当前工作目录。

```
C: \Users\oier\edge>cd
C: \Users\oier\edge
```

在bash中输入pwd，可以获取当前工作目录。

```
oier@noilinux: \~/edge\$ pwd
/home/oier/edge
```

其中~表示当前用户的主目录。例如当前用户名叫luogu，且这个用户的主目录是/home/luogu，那么~代表的就是/home/luogu。注意，Windows中的路径中使用的是，而类UNIX系统中使用的/。

如果想要知道工作目录下有什么文件，在cmd中可以使用dir。

```
C: \Users\oier\edge>dir
// 驱动器 X 中的卷是 XXXXXXX
// 卷的序列号是XXXX-XXXX
C: \Users \oier\edge的目录
2021/07/26 11:34 <DIR>
2021/07/26 11:34 <DIR> ..
2021/07/2611:34 <DIR> clap
2021/07/26 11: 12 4, 797 edge.cpp
1 个文件 4,797字节
3个目录123,456,789,012可用字节
```

同样的，在bash中可以使用ls。

```
oier@noilinux: \~/edge\$ 1s
clap edge.cpp
oier@noilinux: \~/edge\$ 1s -al
// 总用量20
drwxrwxr-x 3oier oier 40967月 2611: 34
```

```
drwxr-xr-x17 oieroier 4096 7月 26 11: 34 ..
drwxrwxr-x 2 oier oier 4096 7月 26 11: 34 clap
-rw-rw-r-- 1 oier oier 4267 7月 26 11: 12 edge.cpp
```

其中.表示的是工作目录。.指的是工作目录的父目录，下面的例子可以解释这两个目录的作用。此时如果装有vim①，就可以通过运行vim edge. cpp来编辑 edge. cpp这个文件了。

工作目录让人们能够方便地使用相对路径，程序能够很容易地获得文件的实际位置。因此，以下几个操作都可以使用vim编辑 edge.cpp。

```
oier@noilinux: \~/\$ vim ./edge/edge.cpp
oier@noilinux: \~/edge\$ vim edge.cpp
oier@noilinux: \~/edge\$ vim ./edge.cpp
oier@noilinux: \~/edge\$ vim ../edge/edge.cpp
```

读者在机器上实践这 4 种写法的时候请注意工作目录的变化。第 2 种情况中省略了工作目录./，这是可行的。第 4 种情况中使用了父目录，../代表的是~/，即~/edge的父目录。

如果想要基于绝对路径操作，只需要写下文件实际的路径。

```
C: \Users \oier>vim C: \Users \oier \edge \edge.cpp
oier@noilinux: \~/\$ vim /home/oier/edge/edge.cpp
```

比较后发现一般情况下使用相对路径能得到更多的便利。dir和ls都可以在参数中指定要列出文件和目录的相对路径或绝对路径，而不是限于工作目录，即：

```
C: \Users \oier>dir .\edge
oier@noilinux: \~\$ ls ./edge
oier@noilinux: \~\$ ls /home/oier/edge -al
```

使用cd命令可以切换工作目录，同样地，参数可以选择相对路径或绝对路径。

```
C: \Users \oier>cd edge
C: \Users \oier \edge> cd C:\Users \oier \edge \clap
C: \Users \oier \edge \clap> cd
C: \Users \oier \edge \clap
oier@noilinux: \~ \$ cd edge
oier@noilinux: \~/edge\$ cd home/oier/clap
oier@noilinux: \~/edge/clap\$ cd
oier@noilinux: \~\$
```

这里说明了bash中如果cd如果没有参数是会将工作目录变成主目录的，而不是像cmd输

> **注：** 1. vim是一款在命令行下的文本编辑工具，当然也适合拿来编写代码。vim提供丰富的文本编辑功能，通过键盘输入指令来操作。在熟练掌握vim的使用方法后，可以不用鼠标，提升编写代码的效率。

出工作目录。

当能够熟练搭配cd使用ls或dir时，其他的命令便很容易学习了。用命令行来做简单的文件管理是上手命令行的好方法。附 表B-1 和附 表B-2 中列出了一些常用命令，推荐读者在自己的机器上亲自尝试。

cmd 中的命令：

| 功能 | 使用方法 |
| --- | --- |
| 输出工作目录 | cd |
| 列出文件夹里的文件 | dir 或 dir 文件夹名 |
| 切换工作目录 | cd 目录名 |
| 切换工作目录（盘符改变） | cd /d 目录名 |
| 复制文件 | copy 旧的文件 新的文件 |
| 移动文件 | move 旧的文件 新的文件 |
| 重命名文件 | rename 旧的文件名 新的文件名 |
| 创建文件夹 | mkdir 文件夹名 |
| 删除文件 | del 文件名 |
| 删除空文件夹 | rmdir 文件夹 |
| 删除文件夹 | rmdir /s 文件夹 |
| 比较两个文件 | fc 文件1 文件2 |

附表 B-1 cmd 中的命令

bash 中的命令：

| 功能 | 使用方法 |
| --- | --- |
| 输出工作目录 | pwd |
| 列出文件夹里的文件 | ls 或 ls 文件夹名 |
| 切换工作目录 | cd 目录名 |
| 复制文件 | cp 旧的文件 新的文件 |
| 移动文件 | mv 旧的文件 新的文件 |
| 创建文件夹 | mkdir 文件夹名 |
| 删除文件 | rm 文件名 |
| 删除空文件夹 | rmdir 文件夹 |
| 删除文件夹 | rm -r 文件夹 |
| 比较两个文件 | diff 文件1 文件2 |

附表 B-2 bash 中的命令

对于创建文件这个功能，写入文件时若文件不存在则会自动创建。因此搭配上合适的编辑器就能使用命令行完成大部分日常工作。

指令非常多，一般来说命令都会提供帮助。在cmd或bash中，直接输入help，就可以得到指令简洁的列表。使用help命令可以获得命令的详细帮助，如helpcd。

特别地，在cmd中，可以使用命令/?的方法获取帮助，如cd/?。在类UNIX系统中，可以使用命令-help，命令 `--help` 或man命令来获取帮助，如 fpc -help、ls -help、man cd。

以下通过帮助来学习编译器 `g++` 的使用方式（这里省略了非关键的信息）。

```
oier@noilinux: \~\$ g++--help
Usage: g++[options] file...
Options:
--help Display this information.
-0 <file> Place the output into <file>.
```

`g++` 的使用中经常出现选项-***o***，从帮助中可以了解到-***o*** 的作用是指定输出文件名，即编译好的可执行文件叫什么。根据用法所说的，可以得出 `g++` -***o*** edge edge.cpp的写法，效果和 `g++` edge. cpp -***o*** edge 是相同的。

## B.2 使用命令行完成算法竞赛题目

这里提供了使用命令完成一道算法竞赛题目的各个步骤：

1. 打开edge这道题，阅读题面，得知评测方式为从edge. in读入数据，答案输出到edge. out。
2. 打开命令行，如果是 cmd则输入cd/D:,如果是bash 则输入 cd /home/oier。
3. 创建目录 edge： mkdir edge。
4. 进入目录 edge： cd edge。
5. 使用编辑器编写本题程序。
6. 编译程序 `g++` edge. cpp -0 edge。
7. 使用编辑器编辑样例输入 edge. in，以及样例输出 edge. `ans`。
8. 使用相对路径运行程序。在cmd中，可以写作..exe，其中.，同时edge.exe会被当作相对路径识别。在bash中，可以写作./edge，其中./不能省略。

检查输出是否符合样例输出要求，如果是 cmd则输入fc edge.out edge. `ans`，如果是bash则输入 diff edge. out edge. `ans`。

1. 0 最后一步，确认程序万无一失，并自信地提交，获得Accepted。

下面列出了一些使用命令行提升编程效率的技巧。

### 1. 输入输出重定向

在使用标准输入输出写程序的时候，通常会使用类似./edge < edge. in > edge. out 的命令重定向标准输入输出来测试程序。在重定向与管道的简单应用上，Windows和类UNIX系统写法差别不大。

使用<某个文件表示将标准输入重定向到这个文件上。

使用>某个文件表示将标准输出重定向到这个文件上。

例如将程序放到edge.cpp中，如果创建一个 edge2.cpp，其内容为edge.cpp开头加上

```
freopen("edge.in", "r", stdin);
freopen("edge.out", "w", stdout);
```

然后再编译运行./edge2，会得到和命令./edge < edge. in > edge.out一样的效果。对于语句：

```
freopen("edge.out", "a", stdout);
```

表示在文件末尾追加内容的，可以使用>>，即./edge>>edge.out。

在本书附录“调试与对拍”节中介绍了使用标准错误流来输出调试信息。重定向标准错误可以使用2>，即./edge2>edge.err，作用类似以下代码：

```
freopen ("edge.err", "w", stderr)
```

如果标准错误重定向到的文件是追加的，使用 2>>即可，即./edge2>>edge.err，作用类似以下代码：

```
freopen ("edge.err", "a", stderr)
```

还有一种用法就是将标准错误重定向到标准输出 2>&1，下面用一个代码来测试：

```
#include <iostream>
int main() {
    std::cerr << "stderr 1" << std::endl;
    std::cout << "stdout 1" << std::endl;
    std::cerr << "stderr 2" << std::endl;
    std::cout << "stdout 2" << std::endl;
    std::cerr << "stderr 3" << std::endl;
    std::cout << "stdout 3" << std::endl;
    return 0;
}
```

编译成可执行文件test如果直接运行./test > test. out，则会发现屏幕上输出：

```
stderr 1
stderr 2
stderr 3
```

打开 test. out，内容是：

```
stdout 1
stdout 2
stdout 3
```

如果运行./test> test. out 2>&1，则屏幕上没有任何输出，而test.out中的内容是：

```
stderr 1
stdout 1
```

```
stderr 2
stdout 2
stderr 3
stdout 3
```

容易发现两个输出被合并了。

命令中的 1 和 2 的意义已经被规定了：标准输入对应 0，标准输出对应 1，标准错误对应 2。

**管道 `|`：** 即竖线。例如，`./A | ./B` 表示将程序 `A` 的标准输出连接到程序 `B` 的标准输入。

标准输入中的内容先传给 `./A`；`./A` 的输出作为 `./B` 的输入，最后 `./B` 的输出传到标准输出。

在 cmd 中，可以使用 `type edge.cpp | sort`。`type` 将文件内容输出到标准输出，`sort` 读取标准输入并按行排序；两者组合后即可将 `edge.cpp` 的各行按字典序排序并输出。

在 bash 中，对应的命令是 `cat edge.cpp | sort`。

### 2. 在命令行中输入EOF

某些程序会一直读入数据直到读入EOF，而如果要在命令行中运行这个程序，如何输入EOF呢?

在 cmd中需要输入`Ctrl+Z`并按 Enter键，在bash中需要输入`Ctrl+D`。

读者可以尝试在命令行中运行sort并尝试输入EOF来终止sort的运行。

### 3. 在程序中读取参数

在命令行中运行会给程序提供参数，那如果自己写的程序想得到这个参数呢?

以 C++ 为例，一般地，有如下写法：

```
#include <cstdio>
/*
 * argc:参数个数
 *argv:参数的字符串数组
 * env:环境变量的字符串数组
 */
int main(int argc, char *argv[], char *env[]) {
    printf("argc=%d\n", argc);
    for (int i = 0; i < argc; ++i)
        printf("argv[%d]=%s\n", i, argv[i]);
    for (int i = 0; env[i] != NULL; ++i)
        printf("env[%d]=%s\n ", i, env[i]);
    return 0;
}
```

标准 C++ 中，`main` 可以不接收参数，也可以接收 `argc` 与 `argv` 两个参数，如下所示。上面的三参数形式属于部分编译器支持的扩展。

**代码片段（需结合上文的声明和函数）：**

```
int main();
int main(int argc, char *argv[]);
```

将其编译，并提供一些参数运行得到如下结果（环境变量过多，已省略）：

```
oier@ noilinux: \~ \$ g++ arg.cpp -o arg
oier@ noilinux: \~ \$ ./arg hello luogu
argc = 3
argv[0]=./arg
argv[1]=hel1o
argv[2]=luogu
env[0]=SHELL=/bin/bash
env[1]=...
...
```

这里可以看到，`argv` 实际上就是把命令分割成若干段的结果。其中 `argv[0]` 一定是程序本身所在路径。

尝试在cmd中输入set或者在bash中输入 `env` 就能获得环境变量，依此可以验证输出的数组 `env` 是否正确。

有了这些知识，就可以利用命令行参数写SpecialJudge了。假设要实现这样一个chk：

```
./chk <input-file><output-file><answer-file>
```

可以在代码中通过 `argv[1]` 获取输入文件名，`argv[2]` 获取输出文件名，`argv[3]` 获取答案文件名。

### 4. 将自己的程序加入命令

发现要运行自己写的程序，必须加上绝对路径或者相对路径，如./chk，./edge，而某些显然不是内置命令的程序如java，emacs等，可以像内置命令一样随处运行?

Windows的环境变量中，有一项为Path，而类UNIX系统中为PATH。path是一个字符串，如果按规定的特殊字符分割它则可以得到一个路径的数组。如果无法通过绝对路径或相对路径找到命令，那么执行命令时会在这个数组中取出所有的路径，分别作为前缀查找。

Windows 里的 Path 以分号分隔，形如 C:;C:;.. • 。如果要在Path中临时添加新的目录，需要在命令行中运行setPath=%Path%；新的目录。如果要永久改变，可以去高级系统设置里修改环境变量。

类UNIX系统里的PATH以冒号分隔，形如/usr/local/sbin:/usr/local/bin:…。如果要在PATH中临时添加新的目录，需要在命令行中运行export PATH=$PATH：新的目录。如果要永久改变，可以学习如何编写.bashrc。最简单的方法就是把exportPATH=$PATH：新的目录这句话作为新的一行写入~/.bashrc，此后每次启动bash这条命令都会生效。

而所有临时添加的内容，在命令行关闭以后都会失效。

例如，对于目录树：

```
home
\---oier
\---edge
clap.sh
```

目前工作目录是/home/oier/edge，需要运行/home/oier/edge/clap. sh，环境变量PATH=/home/oier:/home/。

如果此时输入./clap.sh，则通过工作目录找到程序成功运行。如果此时输入 clap.sh，则会按顺序在/home/oier/，/home 查找。结果是找不到，因为/home/oier/clap. sh和/home/clap. sh不存在。如果改变PATH，变为PATH=/home/oier:/home/:/home/oier/edge,则不管工作目录是什么，输入clap. sh都能成功运行。

### 5. 运行脚本

如果有些命令需要经常执行，可以将其封装为脚本。编辑完一个脚本后，可以使用对应的解释器运行，如 python3edge.py。如果想要像执行命令一样，直接运行一个脚本，像./edge.py这样的形式执行，需要满足特殊的条件。

在 Windows里可以用cmd解释批处理脚本，即用cmd的指令一行行写进扩展名为 bat的文件。bat文件可以直接执行。其他类型的脚本如果指定了打开方式也可以直接打开。

在类UNIX系统里脚本可以指定解释器，在文件开头一行写上#! 解释器目录即可指定解释器。例如下面是一段Python3 脚本：

```
#!/usr/bin/python3
print ('Hello world!')
print ('Hello luogu!')
```

将其保存为 hello.py，但是直接运行可能会提示权限不足，这是因为文件没有执行权限，运行chmod+***x*** 脚本文件名之后就可以直接运行了：

```
vim hello.py
chmod +x hello.py
./hello.py
```

在算法竞赛中，如果经常在命令行中输入形如 `g++` edge.cpp-oedge的命令来编译，这里希望改掉这个习惯，使用脚本能够让用户避免产生类似 `g++` edge-***o*** edge.cpp的致命错误①。例如，在NOILinux中，创建compile. sh，并在每次编译中使用它：

```
#!/usr/bin/bash
g++ edge.cpp -o edge
```

或是使用工程中广泛使用的Makefile。创建文件Makefile：

```
edge: edge.cpp
g++ edge.cpp -o edge
```

这样在每次编译时，在命令行中运行make即可。Makefile还有更加深入的应用，读者可以参阅其他资料学习。

> **注：** 1. 这样的话，cpp代码就会被生成的可执行文件覆盖掉。

# 附录 C 常数优化技巧

相同渐进复杂度的算法，实际运行速度可能不尽相同，这是因为在讨论渐进复杂度时会使用大 0 记号忽略掉常数因子带来的影响。

例如，扫描 1 遍数组和扫描 10 遍数组的渐进复杂度是一样的，但是后者显然会花费更多的时间：0.5 秒与 5 秒这两种运行结果对于同样是常数的时间限制来说是不同的。类似的例子还有**树状数组**、**线段树**和平衡树，尽管它们结构、功能、复杂度类似，但是速度上存在显著的差异。

与此同时，计算机的体系结构特性也会导致不同代码性能上存在差距。例如，对于矩阵乘法来说，一种实现是这样的：

```
const int N = 512, n = 512;
void multiply_ijk(int c[N][N], int a[N][N], int b[N][N]) { // c = a * b;
    memset(c, 0, sizeof(int) * N * N);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];
}
```

还有一种实现是这样的：

```
void multiply_ikj(int c[N][N], int a[N][N], int b[N][N]) {
    memset(c, 0, sizeof(int) * N * N);
    for (int i = 0; i < n; i++)
        for (int k = 0; k < n; k++)
            for (int j = 0; j < n; j++)
                c[i][j] += a[i][k] * b[k][j];
}
```

以上两段代码仅仅交换了两层循环，看起来几乎一致。然而，将这两段代码在“测测你的矩阵乘法”（洛谷B3615）题中测试可以发现：它们的性能存在显著差异，见附 表C-1。

| 方法 | 时间（C++17，O2） |
| --- | --- |
| multiply_ijk | 251 ms |
| multiply_ikj | 78 ms |

附表 C-1 两段代码测试结果

将在后面了解到，后者相比前者能够更好地利用计算机的缓存。

算法竞赛考查形式是编写代码实现算法，即使渐进复杂度正确也不一定能够通过；与此同时，渐进复杂度大也不一定超时。因此，掌握算法设计与理论分析固然是学习算法竞赛的首要目标，追求程序实现高效亦是有价值的。

以下一些是对于优化的忠告。

1. “过早的优化是万恶之源”①

提前考虑优化技巧可能会破坏程序的可读性，并会因为额外的代码引入潜在的缺陷（Bug）。因此，在比赛时先写出一份最直白、好写的程序，保证正确性后再对其进行修改来优化常数。

2. 只进行有效的优化

只有对真正耗时高的部分优化是有效的。对程序进行常数优化时，重点应当放在最占时间的部分上，如循环内部、被调用次数最多的函数以及递归的尾巴等。如果花费精力优化运行时间占比小的部分，那么不仅浪费代码时间、将代码变得更晦涩难懂，而且收效甚微。

可以在程序内加入计时器来计算每部分的耗时，以判断哪部分最需要优化。通常可以使用以下代码来简单测试一个函数使用的时间，在Windows上输出的是以毫秒为单位的时间，在Linux上则是以微秒为单位的时间。

```
time_t start = clock();
// 这里写测试代码
time_t duration = clock() - start;
cerr << "time = " << duration << endl;
```

如果需要更精细的测量，可以使用 C++11 以后有的头文件里提供的纳秒级别时钟。以下代码输出的是以纳秒为单位的时间。

```
auto start = chrono::system_clock::now().time_since_epoch().count();
// 这里写测试代码
auto duration = chrono::system_clock::now().time_since_epoch().count() - start;
cerr << "time = " << duration / 1e9 << endl;
```

在测速比较两个方法时，需要做好控制变量。例如，应当用同样的输入数据和同样的编译选项；尽可能规避生成数据、输入输出、其他程序占用CPU等可能引起测速不稳定的因素；重复测试几次以避免偶然因素等。

如果用户习惯使用外部工具来检查，gcc中提供的gprof或许是一个不错的选择，这个工具可以显示每个函数花费的时间—尽管在调试的优化等级下每个函数时间占比和开启优化开关后略有不同。

3. 不要局限于常数优化

有时，可能因为提交代码恰好超时一点点而抱怨出题人卡常数。事实上，看起来卡常数的题目或许存在更优秀的做法。对较差复杂度算法进行常数优化来提高得分只是比赛策略，学习更好的算法仍是最重要的。

> **注：** 1. 这是《计算机程序设计的艺术》的作者DonaldKnuth的名言。

以下介绍一些常数优化的技巧，并分析这么做的原因。

## C.1 读写优化

计算复杂度时，输入输出复杂度很容易被忽略。

例如，某题目考查 $O(n)$ 复杂度的算法，要求输入超过 $10^6$ 个整数。读入这些数的字符处理量为 $O(n \log W)$，其中 $W$ 为值域。取常见的 $W = 10^9$ 时，输入文件可能超过 $10\,\mathrm{MB}$。这意味着即使主程序很快，读入仍可能需要处理超过 $10^7$ 个字符。

标准库提供了强大的输入输出方法。但是，出于设计与安全上的原因，它们的性能不一定能满足要求，因此在面对巨大的输入输出文件时可能需要考虑优化。

**结论：**

1. 通常来说，不需要特别在意读写优化。
2. 如果习惯 `cin` / `cout`，在文件较大时需要记得关闭流同步、使用$n$ 而不是 `endl`。
3. 输入输出文件大小十分极端时，才需要考虑手写读写优化。这通常意味着输入输出文件超过了 106 个int类型数据。即使在这个量级下，手写输入与标准库的差距仍通常不到 0.1s。

**分析：**

通常使用的 `scanf` / `printf` 和 `cin` / `cout` 在算法竞赛时性能并不一定能是最优的：

1. 通用库功能强大，但代价是会做一些额外的安全性保障，如异常处理、线程安全等。
2. `scanf` / `printf` 需要在程序运行时解析格式串，这会花费不少时间。
3. `cin` / `cout` 为了和 `scanf` / `printf` 等stdio库函数同步，内部缓冲区被禁用，写入或读取每个字符时都需要与stdio交互。因此，即使 `cin` / `cout` 在编译期确定读入格式，性能也通常比 `scanf` / `printf` 差。

当然，可以通过以下代码关闭流同步提升 `cin` / `cout` 性能。关闭流同步后，`cin` / `cout` 会拥有自己的缓冲区，与 `scanf` / `printf` 混用会带来异常。

```
ios::sync_with_stdio(0);
cin.tie(0);
```

4. `cout` << `endl` 会使用系统调用刷新输出缓冲区，输出大量换行时非常缓慢。

如果需要手写输入函数，使用getchar读取整数是一个有效减少输入时间的手段。

```
int read() {
    int ret = 0, sgn = 0, ch = getchar();
    while (!isdigit(ch))
        sgn |= ch == '-', ch = getchar(); // 假设 getchar 必
    // 然成功
    while (isdigit(ch))
        ret = ret * 10 + ch - '0', ch = getchar();
    return sgn ? -ret : ret;
}
```

以上是一种可能的实现。有以下几点需要注意：

1. 在输入到达文件结束时继续读取，`getchar()` 会返回-1，导致死循环。如果遇到输入可能不完整的情况，需要在第 1 个while里判断EOF。
2. 在读取 $-2^{32}$ 时，尽管返回值通常是正确的，但是ret会整型溢出，触发未定义行为。

3. 输人均为正数时，可以删掉sgn相关语句。如果习惯不写sgn，读负数时需千万注意。

上面的程序中 `getchar()` 仍可以优化。stdio库里维护了一个缓冲区，以减少读取硬盘的次数：可以自己调用fread，来进一步减少读取硬盘的次数和标准库边界检查的时间。标准库里通常缓冲区的大小为8KB，事实上这是一个合理的大小：更大的缓冲区性能提升非常小。可以选择稍大一些的 2 的次幂作为缓冲区的大小，以同时兼顾硬盘性能、缓存友好和内存开销。

```
const int SIZE = 1 << 14; // 16 KB
char getc() {
    static char buf[SIZE], *begin = buf, *end = buf;
    if (begin == end) {
        begin = buf;
        end = buf + fread(buf, 1, SIZE, stdin); // 假设 fread 必然成功
    }
    return *begin++;
}
```

使用以上代码中的 `getc()` 函数替代 `getchar()`，可以获得部分性能提升。

如果评测系统为Linux，可以考虑直接使用getchar`_unlocked()`替代`getchar()`，效率接近fread。这是一种放弃了线程安全的函数，但是对算法竞赛来说没有弊端。

手写输出和手写读入没有本质区别，只需要注意一点：如果自己维护了输出缓冲区，那么在程序结束时一定要记得清空缓冲区。

在洛谷上测试读入 $\mathbf{2\times10^6}$ 个 $\mathbf{10^9}$ 以内的随机数（19.4 MB），结果见附表 $\mathbf{C-2}$。

| 方法 | 时间（C++17，O2） |
|---|---|
| cin | 440 ms |
| cin 关闭同步 | 120 ms |
| scanf | 120 ms |
| getchar | 53 ms |
| getchar_unlocked（Linux） | 39 ms |
| fread | 37 ms |

附表C-2 读入数据耗时

在关闭流同步时，计算 $10^6$组 $\boldsymbol{A+B}$ 问题并输出，值域 $10^9$，测试结果见附 表C-3。 $\mathbf{10^6}$ $\mathbf{10^9}$

| 方法 | 时间（C++17，O2） |
|---|---|
| endl | 652 ms |
| \n | 180 ms |

附表C-3 输出数据耗时

## C.2 缓存优化

很多时候人们认为有些常数优化是“玄学”：一点微小的实现差别可能带来截然不同的性能表现，甚至相同的代码在不同评测环境下都可能有不同的表现。但是，常数理应是可以分析的，这种分析很多时候依赖于计算机体系结构与操作系统相关的知识。

**结论：**

要利用好缓存的特性，简而言之，理想的状态首先是访问缓存存得下的一段连续内存；其次是连续访问的内存；再次是随机访问的内存与完全不连续的访问；最坏是接连访问地址低位完全相同的不同地址，如同时访问多个大小为 2 的次幂的数组的相同下标。通常来说，可以在数组最低维是 2 的高次幂时多开 1，以很小的代价规避这样的问题。

**分析：**

从一个最简单的 $a+b$ 程序出发。

```
#include <cstdio>
int a, b, c;
int main() {
    scanf("%d %d", &a, &b);
    c = a + b;
    printf("%d\n", c);
}
```

`c = a + b` 这条语句在 GCC 9.3.0（未启用优化）下，被翻译为：

```
mov edx, DWORD PTR a[ rip];从 a 的地址读取值存放到 edx
mov eax, DWORD PTR b[rip] ;从b的地址读取值存放到 eax
add eax, edx i eax += edx
mov DWORD PTR c[rip], eax; 将 eax 存到 c 的地址
```

代码里出现的eax edx被称作寄存器（Register）。寄存器内置在CPU里，通常只有几十个。CPU只能直接对寄存器进行计算，所有经过CPU处理的数据都需要被加载到寄存器里；同时寄存器也是数据处理的最小单位。

CPU会花一个固定的时间来进行一系列基本操作，这个固定的周期被称为CPU的周期，通常说的CPU频率就是每秒执行的周期数。理想情况下，整数的加减法、位运算等简单的指令需要花费一个周期，对于整数乘除法、浮点数操作等复杂的指令则需要更多的时间。

寄存器十分珍贵，应该让最需要被处理的数据使用，而其他变量和常量储存在内存里，例如上面程序里的 3 个全局变量：mov指令就的作用就是读取和写入内存。内存容量很大，但是由于在CPU外，访问起来相对较慢：内存典型延迟接近 100ns，而如今CPU典型频率在 4GHz左右，这意味着一次读写内存会花费几百个周期。

为了加速内存访问，CPU中引入了缓存（Cache），存取最近用到的一部分数据。缓存利用了程序运行时的时空局部性来加速运算：程序运行时关心的很多数据都是连续访问的。缓存会依据访问地址按块加载访问目标附近的数据（称为缓存行，常见为 64B）以减少内存访问，并将剩余地址位划分为高位标签

（`tag`）和低位索引（index）两部分。对于最简单的缓存模型来说，访问时会查缓存里下标为index的位置是否存在 `tag`，如果是则直接由缓存处理，称为缓存命中；否则称为缓存未命中，继续交给内存来处理，并在内存访问完成后替换index的内容。

因此，如果index冲突非常多，也就是地址低位相同的访问很集中，那么缓存效率将会大大降低，而这种降低通常体现为访问步长为 2 的次幂间隔的内存引起的，这也就是通常不建议将二维数组低维开 2 的高次幂的原因：连续访问低维下标相同的内容会引起严重的缓存冲突。相比之下，随机访问引起的问题仅仅是缓存行的利用率低，并不会达到最坏情况。

从更高层次的观点地来说，可以把整个存储结构看作一个缓存的结构，每层是下一层的缓存。缓存之于寄存器，与内存之于缓存、硬盘之于内存本质相同：越靠近CPU则越快，越小，同时越昂贵；离CPU越远，容量越大，同时相对更便宜。硬盘的读写也有类似缓存行的概念，称为扇，这也是读入优化时缓冲区的大小取 4KB的倍数的原因：现代机械硬盘的扇区大小是 4KB。

附 表C-4 是在 2018 年之后NOI系列比赛中评测机的（CPU是Intel酷睿i78700k，主频 3.70GHz；内存32GB）的相关参数①。

| 位置 | 大小 | 访问延迟 |
| --- | --- | --- |
| 寄存器 | < 1 KB | 一个周期（约 0.27 ns） |
| L1 缓存 | 每核 32 KB 指令、32 KB 数据 | 约 1 ns |
| L2 缓存 | 1.5 MB | 约 4 ns |
| L3 缓存 | 12 MB | 比 L2 慢 10 倍 |
| 主存 | 32 GB | 比 L3 慢 2 倍 |
| 硬盘 | — | 比主存慢 100 倍起 |

附表C-4 评测机的相关参数

以下列举了一些算法中的常数优化策略，借助缓存提升算法的效率。

1. 图的存储

对于图的存储，以下代码：

```
struct edge {
    int u, v, w;
};
edge e[M];
```

的性能通常是优于下面代码：

```
int u[M], v[M], w[M];
```

原因是很可能同时关心一条边的所有信息，如求最短路时会同时关注出边和边权。类似地，存储关联的一组数据时，用结构体存储一般都会让数据空间局部性变得更好，这样可以更好地利用缓存特性。

2. 矩阵乘法

以两个矩阵乘法为例来分析为什么mul_ijk会比mul_ikj慢。

考虑最内层循环。mul`_ijk的一次内层循环会访问ret[i][j], a[i][0:n]` 和 $\mathbf{b}[0:n][j]$，对于 $b$ 来说是不连续的，会导致加载的缓存行中只有一个int有用，访问效率很低。而对于mul_ikj

> **注：** 1. 评测机处理器型号在NOI官网上公布；CPU具体参数可以在英特尔的官网上查到。

来说，一次内层for循环会访问 `ret[i][0:n]`，`a[i][k]` 和 `b[k][0:n]`，内存访问是完全连续的，效率会有明显提升。

同时，矩阵大小开 $N = 512$ 也对mul_jk的访问效率产生了很大的影响：每次内层循环对于 $b$ 的访问地址下标低 11 位完全相同，导致缓存的冲突率非常高。如果将 $N$ 开大 1，两个函数的性能差距就没那么明显了。在不同情况下的实验结果见附 表C-5：

| 方法（n = 512） | N = n | N = n + 1 | N = n + 16 | N = n + 32 |
|---|---|---|---|---|
| multiply_ijk | 251 ms | 128 ms | 107 ms | 125 ms |
| multiply_ikj | 78 ms | 78 ms | 79 ms | 79 ms |

附表C-5 不同情况下的实验结果

可以发现，对于缓存友好的写法将数组开大一点性能损失很小，但是能让缓存不友好的写法性能提升一倍以上，因此在数组最低维为 2 的高次幂时，开大一点通常会是没有损失的。

从附 表C-5 可以发现不同增加量会有不同的性能，留给有兴趣的读者来思考与探索（提示：可以通过缓存行大小为 64B，以及缓存的组关联特性来解释）。不过由于具体取值对性能影响并不算大，一般只需要选一个喜欢的数字加上去就好了。

注意，这种开大 1 的技巧通常只针对 2 的大次幂。对于小数组来说，更加紧凑的布局会提高缓存行的利用率，例如 `inta[4][4]` 可以被恰好一次缓存读取完成，而如果此时将数组下标开大 1 就是画蛇添足了。

3. ST表

用于查询RMQ的ST表一般实现类似如下：

```
int ST[LOG][N + 1];
void init() {
    // ST[0]存储原数组
    for (int j = 1; j < LOG; j++)
        for (int i = 1; i + (1 << j) - 1 <= N; i++)
            ST[j][i] = min(ST[j - 1][i], ST[j - 1][i + (1 << (j - 1))]);
}
int query(int l, int r) {
    int k = __lg(r - l + 1);
    return min(ST[k][l], ST[k][r - (1 << k) + 1]);
}
```

通常会用一个二维数组存储ST表：一维为log $n$，一维为 $n$。通常来说，将log $n$ 放在数组高维，$n$ 放在数组低维会更好。如果将logn放在数组低维，每次缓存读取的缓存行总是只有一个int是有用的；而如果将 $n$ 放在低维，预处理时内存访问连续，查询时对于区间长度略超过 2 的次幂的查询只需要一次缓存行读取。

4. 分块离线

对于分块查询的问题，可以将每次询问头尾的块内查询离线下来，对于属于同一块的查询一并处理。这么做不仅将空间复杂度降低了，更提升了数据访问的时空局部性，能显著提升程序运行效率。

5. 提高复杂度来提速标准库里的std::sort在递归到区间长度小于 16 时，会切换到 $O(n^2)$ 的插入排序。这是因为插入排序内存访问非常连续，缓存非常友好，而且复杂度中本身带有 $1/2$ 的常数；相比之下，对于快速排序等递归排序来说，对于长度为 16 的区间都还需要进行至少 4 次递归，同时还会有栈上的开销，因此表现并不好。换一个角度来看，这里 $n$ 和logn差距并不大，因此常数就显得很重要了。在实现**分治**时也可以采取类似的技巧，降低分治时间复杂度余项的开销。

其实，类似分治尾部的暴力，有时更坏的复杂度能换取更好的表现。例如，从渐进复杂度的角度来看，$O(n\log^k n)$ 是一定比 $O(n\sqrt{n})$ 好的，但是在算法竞赛语境下 $k \geqslant 2$ 时前者通常都不比后者快，因为前者的log通常是分治和数据结构的互相嵌套，不仅缓存不友好，而且常数还会连乘在一起。因此，如果因为低复杂度、高常数的算法无法通过而烦恼时，不妨退一步，考虑高复杂度低常数的算法。

一个最典型的例子是 `bitset`。`bitset` 通过利用存储的特性，将布尔压入整数里，在批量逻辑操作上甚至有 $1/w$ 的常数，其中 $w$ 为机器位宽（现在通常为 64）。在实现传递闭包时， $O(nm)$ 的算法甚至可以通过 $n$，$m$ 在 $10^5$ 量级的数据范围。

6. 基数排序

读者可以参考“挑战”（洛谷P4604，WC2017）这道题目，其第 1 问是整数排序，要求在 6 秒内将 $2 \times 10^8$ 个整数排序。一个针对基数排序的优化是：采用更小的桶，多跑几趟。多跑几趟会进行更多操作，但由于小桶可以完整放进L1 缓存，相比起大桶效率反而可能更高。

```
const int SZ = 1 << 16;
void SORT(unsigned a[], unsigned c[], int n, int w) {
    static int b[SZ];
    for (int i = 0; i < SZ; i++)
        b[i] = 0;
    for (int i = 1; i <= n; i++)
        b[(a[i] >> w) & (SZ - 1)]++;
    for (int i = 1; i < SZ; i++)
        b[i] += b[i - 1];
    for (int i = n; i; i--)
        c[b[(a[i] >> w) & (SZ - 1)]--] = a[i];
}
unsigned tmp[N + 1];
void Sort(unsigned a[], int n) { // sort (a + 1, a + n + 1)
    SORT(a, tmp, n, 0);
    SORT(tmp, a, n, 16);
}
```

以上代码是采用大小为 $2^{16}=65536$ 的桶跑两趟的代码，可以类似地选取 $2^{11}$ 和 $2^{8}$ 分别跑 3 趟和 4 趟做比较（注意奇数趟时需要额外从 `tmp` 复制到 $a$ 中一次）。对于 $N=2\times10^{7}$ 的输入数组，测试结果见附表 C−6，

| 方法 | 时间（C++17，O2） |
| --- | --- |
| std::sort | 1390 ms |
| $2^{16}$，跑两趟 | 417 ms |
| $2^{11}$，跑 3 趟 | 321 ms |
| $2^{8}$，跑 4 趟 | 397 ms |

附表 C-6 测试结果

对于CCF评测机的i7-8700k来说，L1 缓存能存下 $2^{13}$ 个int类型数据。由于存储数据的数组也要进入缓存，因此选择 $2^{11}$ 作为桶大小是相对合理的。缓存大小对于不同机器可能不一样，在比赛中可以按照测试结果来选择桶的大小。

## C.3 编译优化

编译器会对程序进行一系列优化，了解编译器相关知识可以更好地配合编译器对代码进行优化。

然而，除了“编译器能做什么优化”这个问题，笔者还会介绍“什么样的优化代码是无效的”。随着编译器的更新与主流赛事O2 优化的开启，编译器的优化程度已经极大提高了，很多手写优化并无必要。因此，可以在不损失性能的情况下编写更加简单直白的代码，花费更少的精力在常数优化上，提升编码效率。

**结论：**

在开启O2 优化的前提下：

1. 请相信编译器。编译器会进行常量运算优化与位运算优化，无须手动展开。
2. 请不要使用inline，因为它们通常会被编译器忽略掉，而且某些函数inline以后可能反而导致性能下降。如果想强制内联，试试inlineattribute（（always_inline））或者改用#define。
3. 请不要使用register，因为它们会被编译器忽略掉，而且register在 C++11 已经被废弃、在 C++17 被禁止了。
4. 多使用局部变量，尽可能使用局部的、无副作用的代码，以方便编译器进行寄存器分析。

**分析：**

编译器是如何分析并优化代码的呢?一种常见的手段是通过将源代码翻译为静态单赋值（StaticSingle-Assignment，SSA）形式的中间表示形式，并在每个函数的局部域层面上进行优化。在静态单赋值形式里，每个局部变量被视为虚拟寄存器，并且只会被赋值一次；如果存在分支合并则通过插入特殊赋值语句Φ来完成。例如，如下代码段：

```
int test() {
    int a = 1;
    int b = 2;
    int c = a + 1;
    a = 2;
    int d;
    if (a + b > 3)
        d = 1;
    else
        d = 2;
    return d;
}
```

可能被翻译为类似下面的句子：

```
test:
a1 = 1
b1 = 2
```

```
c1 = a1 + 1
a2 = 2
tmp1 = a2 + b1
if tmp1 > 3:
d1 = 1
else:
d2 = 2
d3 = phi (if : d1, else: d2)
return d3
```

SSA的好处是：可以很清晰地记录计算结果的生存周期和之间的依赖关系。例如，如果将SSA所有变量赋值时使用的变量打上标记，那么大部分无用变量都存在一个显著特征：没有标记。因此，通过维护标记数量，可以很方便删除无用的计算代码。与此同时，一个值是否是常量也可以通过查看这个变量被定义时，所有使用的变量是否恒为常量来判断。这两种优化方法分别被称为死代码删除和常量传播，如果简单应用在上面的代码块里，就可以逐步分析出：

1. a1 是常量：1
2. b1 是常量：2
3. c1 没有被使用过
4. 删除c1 后，a1 没有被使用过
5. a2 是常量：2
6. tmp1 是常量：$\mathbf{2+2=4}$
7. 替换tmp1 后，a2、b1 没有被使用过
8. tmp1>1 是常量：True

因此，可以得到如下优化过的代码：

```
test:
if True:
d1 = 1
else:
d2 = 2
d3 = phi (if : d1, else : d2)
return d3
```

读者可能发现这里的ifTrue完全可以优化，这种减少无效分支的优化叫控制流图简化，可以删除掉无用的分支，在删除之后，test函数变成了一个代码块：

```
test:
d1 = 1
d3 = d1
return d3
```

再应用之前的优化，我们可以发现test函数变成了直接返回 1。

02 中开启的常见优化还有：

1. 公共子表达式消除

如果接连算了相同的式子，那么完全可以将前一个式子的结果存起来给下一个继续使用。对于SSA来说，判断两个结果是不是一样是非常容易的，只需要看表达式右边是不是完全等价即可。

这种优化的典型例子是高维数组寻址。对于数组int `a[N][N][N]`，如果接连访问 `a[i][j][x]` 和 `a[i][j][y]`，由于左边的地址是 $a$ + （$i * N * N + j * N + x$） * `sizeof(int)`，而右边是 $a$ +（$i * N * N + j * N + y$） * `sizeof(int)`，可以发现这两个地址关于 $a$， $i$， $j$ 的运算都是完全相同的，因此可以将除了最后的加法以外所有式子合并而只计算一次。

2. 尾递归

某些函数会在结束时返回调用自己的值。一种典型例子是**线段树**，例如，某个询问完全落入当前结点的左子结点时，常见的写法是直接递归调用自己，并将结点和区间改为左子结点而询问区间不变。

**结构示意（含省略部分）：**

```
int query (int x, int l, int r, int ql, int qr) {
if (l == ql && r == qr) { ... }
int mid = (l + r) /2;
if (qr <= mid)
return query (t[x].l, l, mid, ql, qr);
else ...
}
```

采用尾递归优化时，这里的returnquery并不会调用函数，而是直接将参数替换并跳转到函数开头。这样，就避免了一次额外的函数调用和函数返回，调用栈也只会有一层。因此，将线段树写为上面这样的适合尾递归的形式，比依次if累加起来性能更好。

3. 代数简化

将一系列代数运算应用交换律、结合律等进行简化。例如：

```
int plus3(int x) {
    x++;
    x++;
    return x + 1;
}
```

可以通过加法交换律和结合律，将常数合并，最终简化为returnx+3。

在02 优化下，甚至还有更加强大的检测。例如，计算阶乘的如下递归函数会被直接简化为循环（注意这不是尾递归）。

```
int fac(int n) { return n == 1 ? 1 : n * fac(n - 1); }
```

以下for循环会直接将 `sum` 标记为 5050。

```
int sum = 0;
for (int i = 1; i <= 100; i++)
    sum += i;
```

需要注意的是，部分类型的数值表示在应用代数简化时会发生错误，如浮点数操作。编译器为了避免计算结果不一致，在Ofast以下级别不对非整数进行改变顺序的代数简化。

4. 循环不变量外提

将在循环里计算的、整个循环过程中不变的值提到循环外计算。例如下列代码：

**结构示意（含省略部分）：**

```
for (int i = 1; i <= x; i++)
for (int j = 1; j <= x; j++)
for (int k = 1; k <= i + j; k++)
...
```

这段代码中，$i+j$ 就是可以提到循环外计算来提升效率。

5. 内联

也就是inline，方法是将函数调用内嵌到调用者的代码块里。这种方法除了减少调用函数的代码开销，还可以方便编译器对代码进行分析。例如，将之前代数优化中的 `x++` 用

```
int inc(int x) { return x + 1; }
```

包装起来，那么编译器由于难以分析函数外的内容，就没法合并了。如果将其内联，那么再结合代数优化就可以直接分析得出和之前一样的结论了。

需要注意的是，内联并不一定对效率有帮助。内联后，被内联的代码长度会膨胀；如果嵌套内联，函数大小甚至可能指数级增长。这不仅会使可执行文件变大，更会让程序运行时指令缓存变差。因此，02 设置了一定的大小和层数限制，来避免过度内联。尽管这样的限制并不一定对所有程序是最好的，但绝

大多数情况下用户并不需要手动强制内联。

6. 强度降低

整数乘除法会相对于加减法和位运算耗费更多时间。编译器会对例如乘除 2 的次幂、模常数运算等替换为等价的简单运算组合，来减少运行时间。由于这种优化简单有效，现代编译器作者编写了很多替换规则。例如，编译器会自动将 $x*17$ 替换为（$x<<4$）+$x$，将 $x*31$ 优化为（$x<<5$）-$x$。因此，通常不需要花费心思去实现这种小优化。

由于有符号整数右移语义并不完全等价于除法，如-$1/2$ 为 0 但-1>>1 为-1，因此如果测试有符号整数除法时，会发现其略慢于右移，这是因为编译器会通过几条简单运算组合来计算这个除法。编写程序时也需要注意这个问题，避免对负数进行用位运算来优化除法。

7. 寄存器染色

SSA中所有局部变量都视为虚拟寄存器，但物理寄存器是有限的，而且一部分变量必须被安排在特定寄存器上。如果遇到两个需要同时处理的变量被安排在同一个物理寄存器上，那么它们就会冲突，其中一个变量将会在使用完毕后写入内存。然而，寻求一个最优的寄存器安排方案是NP困难的，因此现代编译器采用了启发式的方法完成这个过程。在寄存器压力很大时，寄存器分配结果是不稳定的：可能因为一点微小的代码变化而改变结果，导致性能差距。为了避免这种情况发生，编写代码时可以考虑缩短每个变量的生存周期，减少同时使用很多变量进行计算。

编写有02 优化的代码时，主要应该注意的是：

1. 减少与编译器会帮你做的操作，如：

- 没有必要写 register和 inline。
- 没有必要将常数等式手动计算结果。

。没有必要将 $x*10$ 写为 $(x << 3) + (x << 1)$。

。对于整数来说，没有必要将 `i++` 写为 `++i`。

- 没有必要刻意规避高维数组。

事实上，在开启了 02 下，无论前面提到的这些技巧是否采用，在编译后是完全相同的。

2. 将计算密集的变量存为局部。

全局变量和将地址暴露给其他函数的变量的赋值在编译器看来是有副作用的，因此编译器不会选择优化它们的读取与写入。一种典型的场景是多线程：两个线程同时写入和读取一个全局变量时，两个线程就会互相影响。即使是单线程，考虑以下这段（魔鬼）代码，

```
int n;
void devil() { n = 2; }
void test(const int &n) {
    // 现在 n = 1;
    devil();
    // 现在n = 2
}
int main() {
    n = 1;
    test(n);
}
```

可以发现：即使是constint&这样看起来很像是永远不会发生改变的变量，由于其值可能发生变化，编译器为了正确性也应当让其读取内存。

然而，私有的局部变量就不会有这样的问题，编译器就可以放心地对其进行分析优化。由于无用的局部变量会被优化器删掉，因此多使用局部变量是没有任何坏处的（除非没有开栈—可能会爆栈），因此可以放心大胆使用局部变量。

应用这个思路，可以继续优化矩阵乘法。通过交换数组 $b$ 的两维，可以在缓存友好的情况下直接计算 `c[i][j]`。而计算 `c[i][j]` 时可以用一个局部变量将其保存起来，最后再赋值。这样的话，内层循环原本总共需要 $n^3$ 次读取 $.n^3$ 次写入，现在只需要 $n^2$ 次写入。

```
void multiply_reg(int c[N][N], int a[N][N], int b[N][N]) {
    // 注意:这个函数会改变b的内容
    for (int i = 0; i < n; i++)
        for (int j = i + 1; j < n; j++)
            std::swap(b[i][j], b[j][i]);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            int sum = 0;
            for (int k = 0; k < n; k++)
                sum += a[i][k] * b[j][k];
            c[i][j] = sum;
        }
    }
}
```

在不同情况下的实验结果见附 表C-7：

| 方法（N = n = 512） | 时间（C++17，O2） |
| --- | --- |
| multiply_ijk | 251 ms |
| multiply_ikj | 78 ms |
| multiply_reg | 62 ms |

附表C-7 不同情况下的实验结果

3. 也可以利用编译器优化，写一些更容易读懂的、看起来跑得慢的代码，把减小常数的工作丢给编译器，来减少代码失误。

## C.4 算法常数优化

本节介绍将常数因子从大 O 记号里找出来的案例。核心思想都是一致的：在瓶颈处采取了较慢的方法时，可以考虑将其规避来提升性能。

1. 模运算相关

很多计数题会要求输出结果对一个整数取模。然而，取模是较为缓慢的操作，因此可以针对其特性进行优化。

**结论：**

1. 对于整数来说，加减法、位运算是最快的，乘法次之，除法（取模）最慢。在 64 位机器上，32 位与 64 位加减法、位运算速度一样。
2. 为了避免乘除法，编译器会对乘除常数进行强度减少优化，因此如果遇到对一个特定的数乘、除、取模时，尽可能将其加上const修饰符以方便编译器优化。
3. 对于一些需要对结果求和取模的题，可以考虑用存储一部分结果的和，来减少取模次数；需要注意一定不要整数溢出。

**分析：**

在考虑算法复杂度时，通常会将加、减、乘、除、比较、位运算、函数调用等看作一次基本操作。尽管在渐进意义上是相同的，它们的实际运行速度存在差距。事实上，加减法和位运算都只会占用一个周期：乘、除法耗时并不稳定，但通常乘法慢于加法，而除法（取模）远慢于乘法。

考虑下面两个函数：分别通过加法和乘法取模来遍历[1，19260823）。

```
unsigned add(unsigned n) {
    unsigned sum = 0;
    for (unsigned i = 1; i < n; i++)
        sum += i;
    return sum;
}
unsigned mod(unsigned n) {
    unsigned sum = 1;
    for (unsigned i = 3; i != 1; i = i * 3 % n)
        sum += i;
    return sum;
}
```

在洛谷上测试，运行时间见附 表C-8。

| 方法 | 时间（C++17，O2） |
| --- | --- |
| add(19260823) | 4 ms |
| mod(19260823) | 140 ms |

附表C-8 测试运行时间

由于编译器会将这里的乘法优化为位移和减法，因此主要的时间差距来自取模。可以看出，除法（取模）是非常花时间的操作。尝试运行以下两个函数，分别采用固定模数和传入的模数来比较：

```
int mod_constant(int n) {
    const int P = 1000000007;
    int sum = 1;
    for (int i = 1; i < n; i++)
        sum = (sum * i) % P;
    return sum;
}
int mod_variable(int n, int P) {
    int sum = 1;
    for (int i = 1; i < n; i++)
        sum = (sum * i) % P;
    return sum;
}
```

测试结果见附 表C-9。

| 方法 | 时间（C++17，O2） |
| --- | --- |
| mod_constant(1e7) | 35 ms |
| mod_variable(1e7, 1000000007) | 79 ms |

附表 C-9 测 试结果

因此，在需要对答案取模时，应当尽可能让模数是被标记为常数。

也可以通过减少取模次数来规避取模。例如，对一系列数值求和取模时，可以考虑用longlong将取模的结果先加起来，最后再取模，例如以下代码中的solve2 是对于solve1 的优化。64 位机上，longlong 的加减位运算操作和int是一样快的，而 longlong 足以存下 $n \times n$ 个不超过 998244353 的数之和，因此用了一次longlong取模的代价优化了 $n \times n$ 次int 取模。

```
const int N = 5005, P = 998244353;
int a[N][N];
int solve1(int n) {
    int res = 0;
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n; ++j)
            res = (res + a[i][j]) % P;
    return res;
}
int solve2(int n) {
    long long res = 0;
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n; ++j)
            res += a[i][j];
    return int(res % P);
}
```

类似地，采用常见的小于 $2^{31}$ 的模数时，可以将不超过 4 个乘法的和放到一起累加，最后再取模。

需要注意的是，这种优化会导致程序语义发生改变。在应用时，应当确认题目限制，一定不能让结果超过求和变量表示的范围，如模数不常见，或者求和次数非常多时。

2. 预处理

类似模运算，有时也会把一些复杂操作看作 $O(1)$，如三角函数的计算、求逆元等。当这种操作出现在瓶颈上时，重复计算相同的值会带来极大的性能损耗，而对它们进行预处理可以有效减少运行时间。

一个典型的例子是FFT（快速傅里叶变换）中的预处理三角函数。对于长度为 $n$ 的FFT，仅需要计算 $n$ 个 $n$ 次单位根，因此预处理三角函数值可以将调用三角函数的次数从 $O(n\log n)$ 减少到 $O(n)$。

还有一个例子是求模质数意义下的逆元。如果需要对一个固定的模数求逆元，并且范围是类似于 $10^7$ 的可以接受的值域，那么递推预处理是一种有效地避免使用快速幂，降低复杂度的方法。

```
// P 为质数, 1 < N <= P
inv[1] = 1;
for (int i = 2; i < N; i++)
    inv[i] = 1LL * (P - P / i) * inv[P % i] % P;
```

当值域很大时采用的一个小技巧是，设置一个阈值 $N$（如 $10^7$），预处理前 $N$ 项，对于超过 $N$ 的数调用递归处理。尽管这么做精确复杂度并不好算，但是这么做通常是比直接快速幂快不少的。

```
int Inv(int x) {
    if (x < N)
        return inv[x];
    else
        return 1LL * (P - P / x) * Inv(P % x) % P;
}
```

3. STL 相关

STL提供了很多强大的方法与模板类，但是有时不一定能满足人们需要的性能要求。

例如，“蚯蚓”（洛谷P2827，NOIP提高组 2016）一题的正解是维护 3 个队列来模拟**优先队列**，进行 $7 \times 10^6$ 次入队出队操作。有选手在考场上由于想出满分解法过度兴奋，忽略了queue的性能不好—特别是在没开 O2 的情况下。最终，由于queue与速龙评测机的出色发挥，喜提暴力分。

由于STL库设计比较复杂，这里仅简要介绍粗略的结论。

**结论：**

1. STL中通常只有数据结构需要担心性能，sort等函数其实非常优秀；而数据结构在开O2 后会有显著性能提升。
2. 如果习惯使用STL，不妨直接使用STL先实现，然后再考虑替代性能较差的部分。
3. vector，priority_queue通常不必担心性能。替代方案：当扩容成为性能瓶颈时，应考虑用数组替代。
4. list性能很差，而且不能完全满足竞赛中各种链表的实现，建议避免使用。替代方案：手写链表非常方便。
5. deque和基于其的stack，queue性能较差，数据量大时避免使用。替代方案：用固定大小的数组实现。
6. set，map功能强大，需要时可以使用，但是性能一般。替代方案：有时可以考虑**离散化**等离线技巧规避。
7. unorderedmap比map好，但不算快。替代方案：由于很多情况下哈希表不必动态扩容，手写哈希表可以成倍提升性能。实践中，二次探测性能比拉链要好。
8. pb_ds库通常比较慢，并且可能存在bug（如rope. `erase(x)`），建议谨慎使用。

## C.5 实践出真知

常数优化的思路并非一成不变。在评测系统条件不同的情况下，程序可能会有完全不同的表现。某些技巧并不一定对于所有情况都正确；针对评测系统精心调优的代码，也很有可能比常数高手写的代码跑得还快。因此，如果读者有任何想法，不妨动手测一测。

如果想更加深入地了解体系结构相关的常数优化技巧，事实上在 2009 年的国家集训队论文里骆可强就在《论程序底层优化的一些方法与技巧》里讨论了相关内容，涵盖了本章没有提到的更多技巧：不过，这篇论文里提到的部分技巧随着编译器的升级和 O2 开启而失效了，读者可以测试一下里面的代码来确认有效性。如果想更加深入地了解计算机体系结构相关的知识，也可以阅读相关的书籍，如《深入理解计算机系统》等。

还可以利用工具来指引自己优化，例如：

1. 使用 `g++ -S` 的命令输出汇编代码，查看代码的具体翻译，来确定一些技巧是否有效。
2. 在练习时，也可以访问“CompilerExplorer”（可以使用搜索引擎搜索该工具）来阅读并可视化定位更加简洁的汇编代码。
3. 评测系统通常存在波动，可以使用“评测鸭”（可以使用搜索引擎搜索该工具）来更加精确测量程序运行时间。评测鸭上也有很多有趣的常数优化例子，包括矩阵乘法、整数排序等。读者也可以看到其他人的提交的优秀代码，了解更多优化程序的方法。