CSP-J 第二轮 核心知识讲义

一、要考试的范围有哪些

C++ 基础知识，基本的算法：枚举、模拟、排序、贪心、前缀和、差分、二分、搜索和动态规划、高精度运算，数据结构：栈、队列、树与常用数学知识。

二、C++ 基础知识

1. 数据类型与溢出

常见竞赛环境中的 32 位 `int` 最大值约为 21 亿，64 位 `long long` 最大值约为 9.22×10^{18}。输入能装下，不代表计算结果也能装下。例如两个数都不超过十亿，各自能放进 `int`，相乘却可能达到 10^{18}。

```
int a = 1000000000, b = 1000000000;
long long product = 1LL * a * b; // 或者(long long)a * b
```

这里的 `1LL` 是 `long long` 类型的整数 1，让右侧乘法从一开始就按 `long long` 计算。如果只写 `long long product = a * b;`，右侧可能先按 `int` 运算而溢出，再把错误结果赋给左侧变量。左侧类型再大，也无法补回已经丢失的结果。

累加时也要这样估计。十万个数，每个最多十亿，总和可达 10^{14}，因此答案和保存中间结果的前缀和数组，都需要足够大的类型。

整数类型要防溢出，`double` 则要留意近似误差。计数、循环次数，以及能换成整数单位的金额，尽量用整数处理。比较两个近似值时，可以用 `if(abs(a - b) < eps)` 判断它们是否足够接近。`eps` 是允许误差，例如 `1e-5`，具体取多大要看题目要求和数值范围。

2. 整数除法

两个整数相除，结果会向 0 截断，例如 `5 / 2` 得到 2。需要小数时，让其中一个数先变成浮点数：

```
double x = 1 / 2;       // 结果为 0
double y = 1.0 / 2;     // 结果为 0.5
```

对非负整数 a 和正整数 b，向上取整常用下面两种方式：

```
long long x = (a + b - 1) / b; // 整数方式，保证加法不溢出
long long y = ceil(1.0 * a / b); // 浮点方式，适用于数值和精度允许时
```

`ceil` 会把结果向上取整，a、b 都是整数时，要先乘 `1.0` 让除法产生小数；直接写 `ceil(a / b)`，里面已经完成了整数除法。如果 a 或 b 本来就是浮点数，才可以直接写 `ceil(a / b)`。

3. 条件与运算符

比较两个值时，用 `==` 判断相等，`!=` 判断不相等，大小比较使用 `<`、`<=`、`>`、`>=`。其中最容易写错的是等号：`=` 用来赋值，判断相等要写两个等号。

多个条件可以用 `&&` 表示“都成立”，用 `||` 表示“至少一个成立”，用 `!` 表示“取反”。例如判断 x 是否在闭区间内，要写 `l <= x && x <= r`，不能写成连续比较 `l <= x <= r`。

访问数组时，把边界判断放在前面：

```
if (pos >= 0 && pos < n && a[pos] == target) {
    // pos 合法，并且找到了目标
}
```

这样写，`&&` 前面的条件一旦不成立，后面就不再计算。下标越界时，程序不会继续访问这个数组位置。

4. 数组与 vector

数组通过下标访问元素。声明 `int a[100] = {};`，就建立了 100 个初值为 0 的位置，合法下标为 0～99。采用 1～n 编号时，至少开 n+1 个位置。

需要随时添加或删除末尾元素时，可以使用长度能够调整的数组 `vector`。常见的建立方式如下：

```
vector<int> a;              // 空容器，还没有元素
vector<int> b(5);           // 5 个元素，初值都是 0
vector<int> c(5, -1);       // 5 个元素，初值都是 -1
```

这几种写法建立出的长度不同：`b` 有 5 个元素，可以访问下标 0～4；`a` 还是空的，要先添加元素，才能访问 `a[0]`。添加、删除和查询长度时，常用下面这些操作：

操作	写法	含义
在末尾添加	`v.push_back(x)`	新增一个值为 x 的元素，长度增加 1
按下标访问	`v[i]`	读取或修改已有的第 i 项，下标从 0 开始
访问最后一个元素	`v.back()`	读取或修改最后一个元素，容器必须非空
删除最后一个元素	`v.pop_back()`	长度减少 1，不返回被删除的元素，容器必须非空
查询长度	`v.size()`	得到当前元素个数
判断是否为空	`v.empty()`	没有元素时返回真
删除全部元素	`v.clear()`	长度变成 0，原来的下标都不能再使用

例如从空容器开始，依次 `push_back(2)`、`push_back(4)`，内容就变成 `{2,4}`；再执行 `pop_back()`，就只剩 `{2}`。

读入 n 个数，可以先建立 n 个元素，再按下标读入：

```
vector<int> a(n);
for (int i = 0; i < n; i++) cin >> a[i];
```

读入后，就可以按下标逐个访问。循环中用 `size()` 取得当前长度：

```
for (int i = 0; i < a.size(); i++) {
    cout << a[i] << ' ';
}
```

只需要依次查看元素时，也可以写 `for (int x : a)`；要排序整个容器，就用 `sort(a.begin(), a.end())`。注意读入时要分清"放进已有位置"和"在末尾追加"：如果初始化为 `vector<int> a(n)` 代表着已经有 n 个元素，再用 `push_back` 添加 n 个数，长度就会变成 2n。

5. 字符和字符串

字符串同样可以按下标逐个处理，不过取出来的是字符。数字字符 `'7'` 与整数 `7` 不同，已知字符 `c` 是数字时，用 `c - '0'` 才能得到对应的数值。

```
string s;
cin >> s;
int n = s.size();
for (int i = 0; i < n; i++) {
    // 处理 s[i]
}
```

字符串有效字符的下标为 0 ~ n-1。`cin >> s` 读取一个不含空白的字符串；要读取含空格的一整行，就用 `getline(cin, s)`。

例：读入一个整数和下一行文字

第一行给出整数 n，第二行是可能含空格的文字。在这种数字后面紧接换行的情况，就需要在 `cin >> n` 后加一次 `cin.ignore()`，再读下一行：

```
int n;
string s;
cin >> n;
cin.ignore();
getline(cin, s);
```

之所以多这一步，是因为 `cin >> n` 读走数字后，会留下行末换行。`getline` 直接遇到它就结束读取，于是得到空串；`cin.ignore()` 默认丢弃一个字符，在这里正好取走换行。数字后若还有多个空格，可改用 `cin.ignore(INT_MAX, '\n');` 丢弃本行剩余内容。连续用 `getline` 读多行时，上一次已经取走了换行，不用再额外调用 `ignore()`。

除了读入方式，还要分清楚字符串的比较规则：字符串的比较是按字典序比较，在比较时，`"10"` 会排在 `"2"` 前面，和整数的大小顺序不同。

6. 循环与函数

`break` 退出最近的一层循环；`continue` 跳过本轮剩下的操作。对于 `for`，执行 `continue` 后仍会执行第三部分的更新；对于 `while`，则直接回到条件判断，写在后面的更新语句可能被跳过。

`return` 结束当前函数。在 `main` 中执行 `return 0;` 会结束整个程序，在搜索函数里执行它则只是返回上一层调用。

如果函数里改过一个值，回到调用处却没变，可以先查传参方式。普通的值参数保存的是副本，通过引用参数才能直接修改对应的原变量。递归时也一样，每一层的局部变量各自独立，全局数组则由各层共用，修改后是否恢复要看它的用途。

自增自减运算放进一个表达式时，`i++` 代表的先用后加，`++i` 代表的先加后用。

7. 输入输出

使用 `cin`、`cout` 时，可以在正式读写前设置：

```
ios::sync_with_stdio(false);
cin.tie(nullptr);
```

关闭同步后，读取数据的速率会大大加快，但是就要统一使用 `cin/cout` 来输入输出，避免与 `scanf`、`printf` 交错使用。一般换行用 `'\n'`，不必每次都用会刷新缓冲区的 `endl`。

对于复赛的题目，明确要求了要使用文件输入输出，就要按题面给出的文件名设置，例如：

```
freopen("task.in", "r", stdin);
freopen("task.out", "w", stdout);
```

这里的 `task` 是示意名称，必须替换成题面要求的名称，并放在正式读写之前。

三、通过数据范围推测算法

语法能写对之后，还要判断算法能否在规定时间内跑完。看到 n 的上限，可以先按下面的范围筛选。这里按 C++、约 1 ~ 2 秒、循环体只做简单操作来粗估；接近上限时，仍要计算实际工作量。大量输出、长字符串比较和多组数据，都会增加耗时。

方法	大致工作量	通常可考虑的 n 范围
枚举所有子集	2^n	通常先考虑 $n \le 20$；每个子集还要遍历 n 项时，应按 $n2^n$ 再估算
枚举所有排列	$n!$	通常先考虑 $n \le 10$；若每个方案都要逐项处理或输出，范围还需缩小
三层各枚举 n 次	n^3	通常先考虑 $n \le 200$，此时约八百万次简单操作
枚举所有数对	n^2	通常先考虑 $n \le 3000$；循环体很简单且时间允许时，可再评估到约 5000
排序	$n \log n$	普通整数排序通常可考虑 $n \le 10^6$
一次遍历、前缀和	n	简单线性处理通常可考虑 $n \le 10^7$，还要确认输入量与内存是否允许

例如 n 的上限是十万，枚举所有数对会达到百亿量级，就要想办法用排序、前缀和等方法减少重复计算。若 n 只有 20，直接枚举各种选择也可能跑得完，指数级搜索就值得考虑。

估算时别漏掉外层调用：一次判断扫描 n 个数，二分又调用多次，两部分的工作量就要相乘。内存也要一起算，二维数组的总元素个数是两维相乘，两边看着都不大，乘起来也可能占用很多空间。

四、枚举与模拟

1. 枚举

数据范围允许时，可以把可能的情况逐个检查，这就是枚举。写循环前，先确定枚举的对象和范围，再把题目的合法条件放进去。

例：统计和为目标值的数对

给定 n 个整数和目标值 target，统计有多少对不同位置的元素之和等于 target。同一对位置只算一次，交换两个位置的顺序不产生新方案。

这道题枚举的是两个位置。让第二个位置始终在第一个位置之后，就能排除同一个位置，也避免交换顺序后重复计算：

```
long long ans = 0;
for (int i = 1; i <= n; i++) {
    for (int j = i + 1; j <= n; j++) {
        if (a[i] + a[j] == target) ans++;
    }
}
```

代码中让 `j` 从 `i+1` 开始，因此 `(i,j)` 和 `(j,i)` 只会统计其中一对。

题目说“两个数”时，要分清是在选两个位置，还是选两个不同数值。相同数值出现在不同位置，是否可以同时选，要由题意决定。

对于暂时想不到优化方法得题目，可以先写出小范围暴力。之后优化时，用它核对结果；比赛中也可能拿到对应数据范围的分数，具体得分要看测试点或子任务如何组织。

2. 模拟

模拟题直接按照题目规则执行即可，但要注意题目得操作要求/顺序，看看应该先判断还是先修改，改过的值会不会影响本轮后面的判断。例如装物品不能超重，就应先检查加入后的总重量，再决定是否加入。

如果题目规定“同时发生变化”，就要先保留旧状态，再统一更新。否则边扫描边修改时，后面的位置可能读到本轮刚改过的新值，执行出来就和题意不同了。

举个简单例子：在一排格子中，每一轮让中间各格的数变成左右相邻两格的数之和，两端保持不变，所有格子同时变化。例如原来的数是：`1 2 3 4`。

如果直接从左往右修改原数组：

```
for (int i = 1; i < n - 1; i++) {
    a[i] = a[i - 1] + a[i + 1];
}
```

第二格先变成了 4。接下来计算第三格时，读到的第二格已经是新值，于是算成 `4 + 4 = 8`，最终得到：`1 4 8 4`

这就不符合“同时变化”的要求了。正确做法是用另一个数组保存本轮结果，**始终从旧数组读取，向新数组写入**，这样，计算第三格时读取的仍是旧值 2，不会受到第二格新值的影响。

五、排序与贪心

1. 排序

当原有顺序不影响答案时，可以考虑先排序，让大小关系变得清楚。常用的升序写法如下：

```
sort(a + 1, a + n + 1);   // 排序 a[1] 到 a[n]
sort(v.begin(), v.end()); // 排序整个 vector
```

`sort` 的范围是左闭右开：包含第一个参数指向的位置，不包含第二个参数指向的位置。要排序下标 1 ~ n，就把右端写成 `a+n+1`；少写这个 `+1`，最后一个元素就没有参与排序。

例：按成绩和编号排序

给定若干学生的成绩与编号，要求按成绩从高到低排列；成绩相同的，编号较小者在前。那么就要自己书写比较函数：

```
struct Student { int score, id; };
bool cmp(const Student& a, const Student& b) {
    if (a.score != b.score) return a.score > b.score;
    return a.id < b.id;
}
sort(v.begin(), v.end(), cmp);
```

这个比较函数先看成绩，成绩相同才看编号。比较时使用严格的先后关系，不能写成 `a.score >= b.score`；两个元素完全相同时，比较结果应为假。

整数的取值范围很小时，也可以先用计数数组记录出现次数，再按数值顺序输出。但是开数组前要看值域有多大，值太大就不适合直接作为数组下标。

2. 贪心算法

有些题排序之后，还需要决定按什么规则选择，这时可以考虑贪心。它在每一步作出选择，因此需要说明：这一步选完，为什么仍能得到整体最优的结果？

例：安排尽可能多的活动

给定若干活动的开始、结束时间，选择尽可能多的互不冲突活动，求最多能选几个。本例允许一个活动结束时，下一个活动立刻开始，且活动时长为正。

先按结束时间从小到大排序，每次选择当前能接上的活动。

结束得越早，留给后面的时间越多。把最优方案中第一个活动换成结束最早的可选活动，后面的活动仍然接得上；之后对剩余活动继续这样选择，就有了这个策略的依据。

```
int count = 0, last = 0; // 此处约定区间起点均不小于 0
for (int i = 1; i <= n; i++) {
    if (seg[i].l >= last) {
        count++;
        last = seg[i].r;
    }
}
```

六、前缀和与差分

1. 前缀和

同一个数组上反复求区间和，每次从头相加会做很多重复工作。可以先记下前面各项的总和：令 `s[i]` 表示前 i 个数的和，`s[0]=0`，逐项累加得到 $s[i] = s[i-1] + a[i]$。

查询闭区间时，去掉左端点之前的部分：$\text{sum}(l, r) = s[r] - s[l-1]$。

```
vector<long long> s(n + 1, 0);
for (int i = 1; i <= n; i++) s[i] = s[i - 1] + a[i];
// 查询闭区间 [l,r]：s[r] - s[l - 1]
```

例如数组为 2、−1、4、3，要查询第 2 ~ 4 项的和。前缀和依次为 0、2、1、5、8，因此区间 `[2,4]` 的和是 `8-2=6`。去掉的是第 l 项之前的部分，所以减 `l-1` 对应的前缀和。

预处理扫描一遍，之后每次查询只做一次减法。它适合数组不变、查询很多的情况；原数组修改后，旧前缀和也可能失效。

2. 差分

如果题目要反复给一段数反复执行区间修改得操作，那么可以记录相邻元素的差，把整段修改变成两个位置的修改。这里从下标 0 开始：定义 `d[0]=a[0]`，对于 `i>=1`，令 `d[i]=a[i]-a[i-1]`。把这些差从头累加，就能还原原数组。

给闭区间 `[l,r]` 全部加 x，只改两处：

```
d[l] += x;
d[r + 1] -= x;
```

还原时，增量从 `l` 处开始生效，到 `r+1` 处被抵消，所以中间整段都会增加 x。等全部修改完成后，再统一求前缀和，得到最终数组：

```
long long value = 0;
for (int i = 0; i < n; i++) {
    value += d[i];
    a[i] = value;
}
```

七、二分

1. 二分查找

给定升序数组和目标值 x，找到第一个大于或等于 x 的元素下标；如果不存在，就返回 `n+1`。

由于数组 `a[1..n]` 已经升序排列，小于 x 的元素都在前面，不小于 x 的都在后面，两部分之间只有一道分界。每次检查中间位置，就能确定接下来该往哪一半找。

```
int l = 1, r = n, ans = n + 1;
while (l <= r) {
    int mid = l + (r - l) / 2;
    if (a[mid] >= x) {
        ans = mid;   // 先记下这个满足条件的位置
        r = mid - 1; // 再去左边找更靠前的位置
    } else {
        l = mid + 1;
    }
}
// ans==n+1 表示不存在，否则 ans 就是所求位置
```

当 `mid` 满足条件，先把它记进 ans，再去左半边找更靠前的位置；不满足时，就去右半边。ans 始终留着已经找到的结果，后面只有找到更早的位置时才替换它。

例如数组为 `2 4 4 7`，查找第一个不小于 4 的位置，最终得到 ans=2。要判断 x 是否确实存在，可以检查 `ans <= n && a[ans] == x`，先确认下标合法，再读取元素。

这里使用闭区间 `[l,r]`，循环条件是 `l <= r`。检查过 mid 后，用 `mid-1` 或 `mid+1` 缩小范围，不再重复检查这个位置。

2. 二分答案

二分也可以用来找最优答案。先尝试一个值，判断它能否满足要求；如果可行与不可行之间只有一道分界，就可以不断缩小答案范围。

例：分段的最大段和

给定一个包含 n 个正整数的序列，以及一个正整数 k，其中 $1 \le k \le n$。请在不改变元素顺序的前提下，将整个序列分成不超过 k 个非空连续段。每个元素必须恰好属于其中一段，不能遗漏或重复使用。

对于一种分段方案，先分别计算每一段的元素之和，再取这些段和中的最大值，作为该方案的结果。请找到一种分段方案，使这个最大值尽可能小，并输出它的最小值。例如，序列为 `7 2 5 10 8`，k=2。若分成 `7 2 5` 和 `10 8` 两段，段和分别为 14、18，最大段和为 18。其他合法方案都无法让最大段和小于 18，因此答案为 18。

那么就可以假设允许的最大段和是 `limit`，可以从左到右尽量往当前段加入元素，装不下就新开一段。如果这样需要的段数不超过 k，就说明这个上限可行。

分析题目发现，如果上限越大，限制越宽松：某个值已经可行，更大的值也可行。存在着单调性，因此可以二分“第一个可行值”。

确定了单调性，再找搜索范围。一个元素不能拆开，所以答案至少是最大元素；把整串作为一段，则得到总和这个上界。代码中用 `a[1..n]` 保存正整数，约定 `1<=k<=n`，相关变量已准备好。

先写判断函数，看看上限 limit 是否能让段数不超过 k：

```
bool check(long long limit) {
    int count = 1;
    long long sum = 0;
    for (int i = 1; i <= n; i++) {
        if (a[i] > limit) return false;
        if (sum + a[i] > limit) {
            count++;
            sum = a[i];
```

```
        } else {
            sum += a[i];
        }
    }
    return count <= k;
}
```

再记录已经找到的可行答案，继续尝试更小的上限：

```
long long l = 0, r = 0;
for (int i = 1; i <= n; i++) {
    l = max(l, (long long)a[i]);
    r += a[i];
}
long long ans = r;
while (l <= r) {
    long long mid = l + (r - l) / 2;
    if (check(mid)) {
        ans = mid;
        r = mid - 1;
    } else {
        l = mid + 1;
    }
}
// ans 是最大段和能达到的最小值
```

`check(mid)` 为真，说明 mid 能用，先存下来，再向左找；为假，说明上限太小，就向右找。

对于题目出现需要求“最大的最小”的时候，就可以确认单调性，再决定是否通过二分去解决。

八、栈、队列与搜索

1. 栈

栈只在一端放入、取出元素，这一端叫栈顶。像一摞盘子，通常先取走最上面、也就是最后放上去的那个。

依次压入 2、5、7，此时栈顶是 7；弹出一次后，栈顶变成 5，再弹出才轮到 2。这就是“后进先出”。

操作	`stack<int> st` 的写法	要记住的事
压入元素	`st.push(x)`	x 成为新栈顶
读取栈顶	`st.top()`	只读取，不删除
弹出栈顶	`st.pop()`	只删除，不返回元素
判断是否为空	`st.empty()`	空栈不能读取或弹出
元素个数	`st.size()`	表示当前还留在栈中的数量

需要取出并保存栈顶时，先读再删：

```cpp
if (!st.empty()) {
    int x = st.top();
    st.pop();
    // 使用 x
}
```

例：判断括号是否匹配

给定一个只包含 `(`、`)`、`[`、`]` 的字符串，判断括号能否按正确的类型和嵌套顺序全部配对。

右括号要先和最近一个尚未配对的左括号匹配，这正好符合栈的顺序。遇到左括号就入栈，遇到右括号就检查栈顶。

例如 `([])`：先压入 `(`，再压入 `[`；遇到 `]` 时，配对并弹出 `[`；遇到 `)` 时，再弹出 `(`。处理完后栈为空，匹配成功。

`([)]` 则在读到 `)` 时出错：栈顶是 `[`，类型对不上，也不能越过它去找更下面的 `(`。

```cpp
bool matched(const string& s) {
    stack<char> st;
    for (char c : s) {
        if (c == '(' || c == '[') {
            st.push(c);
        } else {
            if (st.empty()) return false;
            char t = st.top();
            if ((c == ')' && t != '(') ||
                (c == ']' && t != '[')) return false;
            st.pop();
        }
    }
    return st.empty();
}
```

除了中途检查空栈和括号类型，结束时还要确认栈为空。比如 `(()` 中途没有冲突，却留下了一个左括号，所以函数最后还要判断一次是否配完。

撤销最近一次操作、处理嵌套结构时，也常会用到栈。递归调用有类似的返回顺序：后调用的那一层先执行完，再回到调用它的位置。

2. 队列

队列在队尾加入元素，在队首取出元素。依次加入 2、5、7，处理顺序就是 2、5、7，这叫“先进先出”。

操作	`queue<int> q` 的写法	要记住的事
入队	`q.push(x)`	加在队尾
读取队首	`q.front()`	取得下一个待处理元素
读取队尾	`q.back()`	取得最后加入的元素
出队	`q.pop()`	删除队首，不返回元素
判断是否为空	`q.empty()`	读队首、队尾或出队前确认非空

操作	`queue<int> q` 的写法	要记住的事
元素个数	`q.size()`	当前队列长度

例：循环报数

编号 1 ~ n 的人围成一圈，从 1 开始报数，报到 k 的人出队，再从下一个人重新报 1。要求输出所有人的出队顺序，其中 n、k 均为正整数。

可以把围成一圈的报数过程放进队列：没数到 k 的人先移到队尾，报到 k 的人输出后删除。比如 1、2、3、4、5 每次数到第 3 人出队，就先移动前两个人，再删除当前队首。

```
queue<int> q;
for (int i = 1; i <= n; i++) q.push(i);
while (!q.empty()) {
    for (int i = 1; i < k; i++) {
        int x = q.front();
        q.pop();
        q.push(x);
    }
    cout << q.front() << ' ';
    q.pop();
}
```

`n=5`、`k=3` 时，出队顺序为 `3 1 5 2 4`。每轮只移动 k−1 次，紧接着的队首才是第 k 人，多移动一次就会错位。

这份写法的操作次数约为 nk，适合规模允许时使用。k 很大时，要留意反复移到队尾的开销。

按到达顺序办理任务，也可以用队列。BFS 就借助队列，让早发现、距离近的状态先被处理。

3. DFS 与回溯

如果每一步都有几种选择，可以沿着一种选择继续尝试，走到完成方案或无法继续时，再返回尝试其他选择，这就是 DFS 的过程。分支层数随输入变化时，用递归比写很多层固定循环更方便。

回溯中最容易漏掉的是恢复现场：如果这一步修改了后续分支共用的路径或标记，返回后要记得撤销修改。

例一：选数凑目标和

给定 n 个整数和目标和 target，每个位置最多选一次，统计选出的数之和等于 target 的方案数。按位置区分选法，允许空集。

例如三个数为 2、3、5，目标和为 5，有两种选法：选 2 和 3，或者只选 5。

处理一个位置时，只有“选”和“不选”两个分支。可以进行搜索，令 `dfs(pos, sum)` 表示前面的位置已经决定完，现在处理第 pos 个数，当前和为 sum。

```
int n;
long long target, ans = 0;
vector<long long> a; // 使用 a[1..n]
void dfs(int pos, long long sum) {
    if (pos == n + 1) {
        if (sum == target) ans++;
        return;
```

```
    }
    sum += a[pos];    // 选当前位置
    dfs(pos + 1, sum);
    sum -= a[pos];    // 恢复本层的和，再尝试不选
    dfs(pos + 1, sum);
}
// 读入数据、令 ans=0 后，调用 dfs(1, 0)
```

等每个位置都决定完，再检查和是否等于目标，这样一组选择只会统计一次。若题目要求至少选一个数，还要记录是否已经选择过元素。

代码先把当前数加进本层的 sum，试完"选"的分支后再减回来，接着尝试"不选"。下一层按值接收参数，对它自己局部变量的修改不会传回本层。另一种写法是直接传入 `sum + a[pos]`，这样本层的和没有改变，就省去了加减操作。

如果还要记录具体选了哪些数，可以用数组或 `vector` 保存路径：选时加入，递归回来后删掉。n 个位置各有两种选择，总方案数为 2^n，只能用于较小规模。所有数非负时，和超过目标后可以停止该分支；有负数时，后面可能把和减回来，不能这样剪枝。

例二：选数组合

从整数 1～n 中选出 k 个不同的数，输出所有组合，每个组合只输出一次。这里约定 `1≤k≤n`，组合内部按递增顺序输出。

例如从 1、2、3、4 中选两个数，`1 3` 和 `3 1` 是同一种组合。让选出的数字始终递增，就只会生成一次。

```
int n, k;
vector<int> path;
void choose(int start) {
    if (path.size() == k) {
        for (int x : path) cout << x << ' ';
        cout << '\n';
        return;
    }
    for (int x = start; x <= n; x++) {
        path.push_back(x);
        choose(x + 1);
        path.pop_back();
    }
}
// 约定 1<=k<=n，初始路径为空，调用 choose(1)
```

代码用 `start` 限定下一次可选的最小值。选了 2，下一层就从 3 开始，既不会重复选 2，也就不会回头产生顺序颠倒的组合，因此不用额外记录哪些数已经用过。

与前一个例子的局部和不同，`path` 是各层递归时共用的，所以它是一个全局变量，那么 加入后尝试深入 和 回溯后撤销 的操作就要配套：`push_back` 记录选择，返回后用 `pop_back` 撤销。比如试完 `1 2`，退回到路径只含 1 的状态，才能接着试 `1 3`。

例三：迷宫路径计数

给定一个网格、起点和终点，只能上下左右移动，不能经过障碍，也不能在同一条路径中重复经过格子。求满足这些条件的路径总数。

因为每一步都可以尝试向上、下、左、右移动，因此可以用递归搜索：先决定当前这一层处理所在的格子，然后递归到下一层处理准备走到的相邻格子。如果到达终点，就说明找到了一条完整路径，将答案加 1，并结束这条分支，不再从终点继续走。不过尝试一个方向之前，要先检查下一格是否在网格范围内、是否为障碍，以及是否已经在当前路径中经过。只有这三个检查都通过，才能继续递归。

且为了避免同一条路径重复经过格子，进入一个格子时，要把它标记为“已经经过”。例如从起点向右走了一格，这一格就加入了当前路径；接下来继续搜索时，不能再走回它。

当这个格子后面的所有走法都尝试完，就要返回上一层，并取消它的访问标记。因为题目限制的是“同一条路径不能重复经过格子”，其他路径仍然可以经过这里。如果返回时不取消标记，后续路径就可能被错误地拦住，导致少算。

因此，整个搜索过程可以理解为：**进入格子并标记，尝试四个方向，搜索结束后取消标记，再返回上一层。**每次返回，都恢复到进入这一格之前的状态，方便继续尝试其他走法，这个过程就叫作回溯。

下面约定行列从 0 开始，`grid` 是全局字符串数组，`'#'` 表示障碍；`vis` 大小为 n×m，初始全为 0，起终点均合法且可通行：

```
int n, m, tx, ty;
long long ways = 0;
vector<string> grid;
vector<vector<int>> vis;
int dx[4] = {1, -1, 0, 0}, dy[4] = {0, 0, 1, -1};
void walk(int x, int y) {
    if (x == tx && y == ty) {
        ways++;
        return;
    }
    vis[x][y] = 1;
    for (int d = 0; d < 4; d++) {
        int nx = x + dx[d], ny = y + dy[d];
        if (nx < 0 || nx >= n || ny < 0 || ny >= m) continue;
        if (grid[nx][ny] == '#' || vis[nx][ny]) continue;
        walk(nx, ny);
    }
    vis[x][y] = 0;
}
// ways=0，vis 清零后，调用 walk(sx, sy)
```

如果把最后的恢复语句删掉，某条路径走过的格子会挡住其他路径，统计就可能漏掉。相反，在同一条路径还没结束时就取消标记，又会允许绕圈重复进入。

4. BFS

BFS 改用队列安排访问顺序，先处理距离起点近的状态，再处理更远的状态。它适合无权图或每一步代价相同的最少步数问题；当边权不同时，就不能直接沿用这种最短路判断。

一层里的点处理完，才轮到下一层，所以第一次发现一个点时，就已经找到了到达它的最少边数。

例：求无权图中的最少步数

给定一个无权图和起点 s，求从 s 到各点最少经过多少条边；不可达的点记为 −1。

通过邻接表 `g[u]` 保存 u 的邻居，`dist` 长度为 `n+1` 且全部初始化为 `-1`：

```
queue<int> q;
dist[s] = 0;
q.push(s);
while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int v : g[u]) {
        if (dist[v] != -1) continue;
        dist[v] = dist[u] + 1;
        q.push(v);
    }
}
```

代码在入队时设置距离，同时完成访问标记，因此同一个点不会被反复加入队列。起点距离为 0，不可达的点则一直保持 −1，两者含义不同。

对于二维网格中的 BFS 也是同样的过程，只是一个状态改成行、列。扩展邻格时，需要依次检查：坐标合法、不是障碍、尚未访问。

6. 图的存储与连通块

前面的 BFS 通过邻居列表查找下一步能到哪里。存图时，邻接表只记录实际存在的边，通常适合边较少的图；邻接矩阵则直接记录两点是否相连，查边方便，但空间约为点数的平方。

无向边 `u-v` 用邻接表存储时，通常要加入两个方向。遍历无向图时必须防止沿边反复返回，使用访问标记，或在确实为树的条件下跳过父节点。

统计连通块时，从每个尚未访问的点开始一次 DFS 或 BFS，每次新启动的遍历对应一个新连通块。网格中的连通区域也可以这样做，这就是常见的泛洪填充。

九、动态规划

1. 一维 DP

例：爬台阶

有 n 级台阶，每次可以上 1 级或 2 级，求恰好走到第 n 级的不同走法数。步子的先后顺序不同，算不同走法。

令 `dp[i]` 表示恰好到达第 i 级的方法数。

考虑最后一步：要么从第 i−1 级走 1 级，要么从第 i−2 级走 2 级。两种走法不会重复，加起来就是 $dp[i] = dp[i-1] + dp[i-2]$。

起点设为 `dp[0]=1`，表示“还没走”这一种起始方案，另外有 `dp[1]=1`。接着从小到大计算，每次用到的前两项都已经算好。

走法数增长很快，要估计结果范围。可以先考虑 `long long`，但它也有上限；题目要求取模时，应在递推过程中及时取模，而不能只对结果取模。

2. 最长上升子序列

给定一个数组，可以删去一些元素，但保留剩余元素的相对顺序，求严格上升子序列的最大长度。

这次把终点固定在第 i 个数：令 `dp[i]` 表示“以第 i 个数结尾的最长严格上升子序列长度”。它前面可以接上一个位置更早、数值更小的元素，因此逐个尝试前面的 j。初值设为 1，表示只选当前这个数。

```
int ans = 0;
for (int i = 1; i <= n; i++) {
    dp[i] = 1;
    for (int j = 1; j < i; j++) {
        if (a[j] < a[i]) dp[i] = max(dp[i], dp[j] + 1);
    }
    ans = max(ans, dp[i]);
}
```

最长的那一段子序列可能在任何位置结束，所以答案要取所有 `dp[i]` 的最大值，不能只看 `dp[n]`。严格上升用 `<`，允许相等的不下降版本才使用 `<=`。

子序列可以跳过元素，连续子数组则不能。读题时先分清这两个要求。上面写法的工作量为平方级，使用前要看 n 的范围。

3. 0/1 背包

给定容量 V 和 n 件物品，第 i 件的体积为 `w[i]`、价值为 `v[i]`。每件最多选一次，求总体积不超过 V 时的最大总价值，允许一件也不选。

令 `dp[j]` 表示已考虑的物品中，总体积不超过 j 时的最大价值。

初始全为 0，表示还没选物品。处理当前物品时，不拿就保留原来的最大价值；拿一次，就从扣除当前体积后的状态转移过来，再加上它的价值：

```
vector<long long> dp(V + 1, 0);
for (int i = 1; i <= n; i++) { // 考虑前 i 件物品
    for (int j = V; j >= w[i]; j--) { // 考虑前 i 件物品的情况下，背包容量为 j 时能获得的最大价值
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]); // 对于第 i 件物品，要么拿，要么不拿取最大值
    }
}
```

这里的容量从大到小更新。计算 `dp[j]` 时，较小容量的 `dp[j-w[i]]` 还没被本件物品更新，用的仍是上一轮结果(考虑前 i - 1 件物品的情况下，背包容量为 j 时能获得的最大价值)，因此每件物品只会选一次。

如果从小到大更新，刚刚拿过本件物品的状态就可能再次参与转移，相当于又拿了一次。这正是下一种背包允许的情况。

答案为 `dp[V]`。

4. 完全背包

给定容量 V 和 n 种物品，每种体积为正、数量不限，求总体积不超过 V 时的最大总价值。

每种物品既然可以重复选择，就允许继续使用本轮已经更新过的状态(考虑前 i 件物品的情况下，背包容量为 j 时能获得的最大价值)，而这个已经被更新过的状态就有可能已经拿了第 i 件物品了，符合可以重复拿取的定义，那么沿用上一节的状态定义，把容量改为正序枚举即可：

```
for (int i = 1; i <= n; i++) {
    for (int j = w[i]; j <= V; j++) {
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    }
}
```

记不清循环方向时，可以思考依赖的来源状态：它必须来自上一轮，还是允许本轮重复使用？

十、数学与位运算

1. 最大公约数与最小公倍数

求两个非负整数的最大公约数，可以反复用"除数、余数"替换当前两个数，直到余数为 0。这样做不会改变最大公约数。

```
long long gcd(long long a, long long b) { // a、b 非负
    while (b != 0) {
        long long r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

求出最大公约数后，还可以用它计算正整数 a、b 的最小公倍数 lcm：

$$\mathrm{lcm}(a,b) = \frac{a}{\gcd(a,b)} \times b$$

2. 质数判断与因数分解

判断一个整数 n 是否为质数，先排除小于 2 的数，再尝试 2 到平方根之间的除数。只要找到一个能整除它的数，就可以确定它是合数：

```
bool isPrime(int n) {
    if (n < 2) return false;
    for (int d = 2; 1LL * d * d <= n; d++) {
        if (n % d == 0) return false;
    }
    return true;
}
```

循环条件保留等号，才能检查到 9、25 这样的完全平方数的平方根。需要判断很多个数时，可以考虑筛法，减少反复试除的工作。

同样的试除思路也能用来分解质因数：找到一个因数后，反复除尽，并记录除过几次。剩余数变小，试除范围也随之缩小；结束时若还剩一个大于 1 的数，它本身就是一个质因数。

3. 位运算

位运算处理的是整数的二进制位。先把数位对齐，再对同一位置上的 0、1 分别计算。例如 5 写成 `101`，3 写成 `011`，就可以按下面的规则逐位运算。

运算	每一位的规则	例子
按位与 `&`	两位都是 1，结果才是 1	`5 & 3` 得到 `001`，即 1
按位或 `|`	两位至少有一个 1，结果就是 1	`5 | 3` 得到 `111`，即 7
按位异或 `^`	两位不同得 1，相同得 0	`5 ^ 3` 得到 `110`，即 6
按位取反 `~`	每一位的 0、1 互换	对类型的全部位取反，不只处理写出来的几位
左移 `<<`、右移 `>>`	把二进制位向左或向右移动	对非负整数，`5 << 1` 为 10，`5 >> 1` 为 2

左移时低位补 0；非负整数右移时高位补 0，移出的低位丢弃。移位次数必须在类型允许范围内，左移后的值也要能表示。

这些规则可以用来检查某一位。`x & 1` 只看最低位，结果为 1 就是奇数；`(x >> k) & 1` 先把第 k 位移到最低位，再取出它，最低位编号为 0。异或还满足两个常用性质：相同数异或为 0，一个数与 0 异或仍是它自己。

`^` 表示异或，不是乘方。若只有一个数出现奇数次，其余数都出现偶数次，把所有数异或起来，相同的数会两两抵消，最后就剩下那个数。

十一、高精度运算

整数超过 `long long` 的范围时，可以用字符串保存，按照手算竖式的规则逐位计算。下面的输入均为非负整数，减法结果可以为负数。

1. 数位存储

按平常写数的顺序保存，例如 5079 存成字符串“5079”。计算前去掉多余前导零，全为零时保留一个 0。

加、减、乘法可以先反转字符串，让下标 0 对应个位、下标 1 对应十位，算完后再反转结果。除法从最高位开始，不需要反转。

字符串中存的是字符，参与计算时要转成对应数字，算出的数位再转回字符保存。

2. 高精度加法

要求计算两个非负大整数的和。

1. 反转两个字符串，在较短串末尾补零，使对应数位对齐。
2. 从个位开始，将两个对应数位与上一位的进位相加。
3. 总和除以 10 的余数作为本位，整数商作为下一位的进位。

4. 所有数位算完后，若仍有进位，就追加到结果末尾，最后反转结果。

例如 $999+1$，每一位都会产生进位，原有数位算完后还要补上最高位的 1，得到 1000。

3. 高精度减法

要求计算两个非负大整数的差，结果可以为负数。

1. 先比较大小：去掉前导零后，位数多的更大；位数相同，从最高位开始比较第一个不同的数位。
2. 若被减数较小，交换两数并记录负号，之后用较大的数减较小的数。
3. 反转两个字符串，给较短串末尾补零。从个位开始，计算对应数位的差，再扣除上一位的借位。
4. 本位不够减时加 10，并向更高位借 1；下一轮计算要扣掉这次借位。
5. 去掉倒序结果末尾的多余零，至少保留一位，再反转结果，按需添加负号。

例如 $1000-1=999$，要连续借位；$1-1000=-999$，先算差的绝对值，再加负号。两数相等时结果为 0。

4. 高精度乘法

要求计算两个非负大整数的积，采用"先累加乘积，再统一进位"的方法。

1. 任意一个数为 0，结果直接为 0。否则反转两个字符串，让下标 0 对应个位。
2. n 位数乘 m 位数，积最多 n+m 位，准备这么多个位置的整数数组，初值全为 0。
3. 每一对数位分别相乘，第 i 位与第 j 位的乘积累加到第 i+j 位，暂不进位。同一位置会收到多个乘积，累加值可以大于 9。
4. 从低位到高位统一进位：当前累加值加上上一位的进位，除以 10 的余数作为本位，整数商继续向高位传递。
5. 处理完剩余进位，去掉倒序结果末尾的多余零，再反转结果。

例如 23×47，个位先累加 $3\times7=21$，十位累加 $3\times4+2\times7=26$，百位累加 $2\times4=8$。统一进位后，个位留 1、进 2，十位得到 28、留 8、进 2，百位得到 10、留 0、进 1，最终得到 1081。

两组数位分别遍历，时间复杂度为 $O(nm)$。

5. 高精度除以小整数

要求用非负大整数除以正整数 b，求整数商和余数。大整数用字符串保存，b 能放进 `int`。

手算除法时，先看前面的数够除几次，再留下余数、落下下一位。因此字符串也从最高位开始，向右逐位计算。

1. 余数初始为 0，从左到右依次读取数位。
2. 把上一轮余数乘 10，再加上当前数位，组成这一轮待除的数。
3. 用它除以 b，整数商作为当前一位商，余数留给下一轮。
4. 全部数位处理完后，只去掉商开头多余的零，中间的零必须保留。最后留下的余数就是最终余数。

例如计算 1234 除以 12：

读入数位	上轮余数乘 10，再加当前位	本位商	新余数
1	$0\times10+1=1$	0	1

读入数位	上轮余数乘 10，再加当前位	本位商	新余数
2	$1\times10+2=12$	1	0
3	$0\times10+3=3$	0	3
4	$3\times10+4=34$	2	10

依次得到 0102，去掉开头多余的零，商为 102，余数为 10。商十位上的 0 不能省略，结果满足 $1234=12\times102+10$。

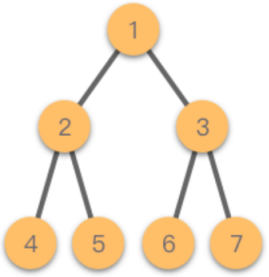
被除数小于 b 时，商为 0，余数为被除数；被除数为 0 时，商和余数均为 0。余数乘 10 的中间值可能超出 `int`，要用 `long long` 保存。整个过程扫描一次字符串，时间复杂度为 $O(n)$。

十二、树

1. 树的基本概念

树是连通且没有环的无向图。n 个节点的树有 n−1 条边，任意两个不同节点之间只有一条不重复经过节点的路径。

给树选定一个根后，就可以从根往下确定父子关系。下图以 1 为根：



名称	怎样理解	图中的例子
根节点	选定的起点，没有父节点	1
父节点、子节点	相邻两点中，靠近根的是父，远离根的是子	2 是 4、5 的父节点
兄弟节点	具有同一个父节点	4 和 5
叶节点	有根树中没有子节点的节点	4、5、6、7
子树	一个节点和它的全部后代构成的树	以 2 为根的子树含 2、4、5
深度	从根到该节点经过的边数，根深度为 0	2 的深度为 1，4 的深度为 2

例如，节点 5 的深度为 2，以 2 为根的子树包含 2、4、5。这里根的深度为 0；有些题目把根所在层记为第 1 层，要注意区分。

2. 树的几种存储方式

树可以用编号和数组、邻接表或动态节点保存。本节重点掌握动态创建节点的方法，前两种方式了解基本形式即可。

用编号和数组保存二叉树

二叉树的每个节点最多有两个孩子，按编号保存时，可以用结构体数组记录左右孩子：

```
struct TreeNode {
    int left;
    int right;
};

TreeNode tree[8] = {}; // 下标 1~7 对应节点，初值全为 0
tree[1].left = 2;   tree[1].right = 3;
tree[2].left = 4;   tree[2].right = 5;
tree[3].left = 6;   tree[3].right = 7;
```

数组下标表示节点编号，`left`、`right` 保存孩子编号，0 表示没有孩子。例如 `tree[2].right` 的值为 5，表示节点 2 的右孩子是 5。

用邻接表保存相邻关系

普通树可以用邻接表保存相邻节点。`neighbors[node]` 保存与 node 直接相连的节点。无向边要在两端各保存一次：

```
vector<vector<int>> neighbors(n + 1);
for (int i = 1; i < n; i++) {
    int u, v;
    cin >> u >> v;
    neighbors[u].push_back(v);
    neighbors[v].push_back(u);
}
```

3. 动态创建节点

动态存储时，用结构体保存节点的值和两个孩子的地址，需要新节点时用 `new` 创建，再连接到树上。

以二叉树为例，一个节点可以这样定义：

```
struct Node {
    int value;
    Node* left;
    Node* right;
};
```

`value` 保存节点的值，`left`、`right` 指向左右孩子，`nullptr` 表示没有孩子。创建根节点：

```
Node* root = new Node{1, nullptr, nullptr};
```

`new` 返回新节点的地址，`root` 保存这个地址。创建并连接左孩子：

```
Node* child = new Node{2, nullptr, nullptr};
root->left = child;
```

`->` 用来通过指针访问成员。也可以直接写成 `root->left = new Node{2, nullptr, nullptr};`。

例一：动态建立一棵二叉树

按上面的二叉树图创建 7 个节点，并输出节点 5 的值：

```
Node* root = new Node{1, nullptr, nullptr};
root->left = new Node{2, nullptr, nullptr};     root->right = new Node{3, nullptr,
nullptr};
root->left->left = new Node{4, nullptr, nullptr};
root->left->right = new Node{5, nullptr, nullptr};
root->right->left = new Node{6, nullptr, nullptr};
root->right->right = new Node{7, nullptr, nullptr};
cout << root->left->right->value; // 输出 5
```

`root->left->right->value` 表示根的左孩子的右孩子的值，也就是节点 5 的值。但是访问成员前，要保证前面的指针不是 `nullptr`。

4. 二叉树的遍历

常见的二叉树遍历如下：

遍历方式	处理顺序
前序	根、左子树、右子树
中序	左子树、根、右子树
后序	左子树、右子树、根

例二：遍历动态建立的二叉树

对例一建立的二叉树，前序序列为 `1 2 4 5 3 6 7`。前序遍历如下：

```
void preOrder(Node* node) {
    if (node == nullptr) return;
    cout << node->value << ' ';
    preOrder(node->left);
    preOrder(node->right);
}
// 建好树后调用 preOrder(root)，输出 1 2 4 5 3 6 7
```

中序序列为 `4 2 5 1 6 3 7`，把输出语句放在两次递归之间；后序序列为 `4 5 2 6 7 3 1`，把输出语句放在两次递归之后。空指针要先判断。

层序遍历使用队列，输出顺序为 `1 2 3 4 5 6 7`。