《初级算法与数据结构》

集合



并查集

并查集

并查集

问题引入：

想象一下，我们班里有 n 个同学，一开始大家谁也不认识，每个人都是一个独立的“小团体”。

随着时间的发展可能“小明和小红会变成好朋友”，“小红和小刚也变成了好朋友”。

而朋友的朋友，也算是朋友。那么老师可能会做出如下的操作：

1. **合并：** 当两个人成为了新朋友时，老师想快速的把他们的 朋友圈 合并成一个更大的圈子。
2. **查询：** 可能随时提问：小明和小华现在是同一个朋友圈里的人吗？ 怎么能快速回答？

并查集

解决这个问题最简单的想法是：使用数组，将每个同学的圈子编号储存起来。

我们用一个数组 `group_id[]`，`group_id[i]` 的值表示 `i` 号同学所属圈子编号。

对于查询(x，y) 操作：

- 直接判断 `group_id[x]` 是否等于 `group_id[y]` 。 时间复杂度：O(1)

并查集

解决这个问题最简单的想法是：使用数组，将每个同学的圈子编号储存起来。

我们用一个数组 group_id[]，group_id[i] 的值表示 i 号同学所属圈子编号。

对于合并(u，v) 操作：

- 假设 u 所在的圈子ID是 A，v 所在的圈子ID是 B。为了合并，我们需要把所有圈子ID为 B 的同学，都改成 A（或者反过来）。

这就意味着必须遍历整个 group_id 数组，检查每个同学 i，如果 group_id[i] == A，就把它改成 B。 时间复杂度：O(N)，其中 N 是总人数。

使用本方法查询很快，但合并太慢了！如果提问 M 次合并操作，那时间复杂度就变为了 O(M*N)，数据量大会导致超时。

方法二：用图的邻接表 + 遍历（DFS/BFS）

既然学过图论，我们知道 **朋友关系** 就是图的**边**。于是我们用一个邻接表 `adj[]` 来存图。

对于合并（u，v）操作：

- u 和 v 是朋友，在图中加双向边就可以了。`adj[u].push_back(v)` 和 `adj[v].push_back(u)`
- 时间复杂度：O(1)。

对于查询(x，y) 操作：

- 要判断 x 和 y 是否在同一个朋友圈，就是判断他们在图中是否连通。 我们需要从 x 开始，进行一次 DFS 或 BFS，看是否能访问到 y。

- 时间复杂度：O(N + M)，其中 N 是人数，M 是关系数（边数）。在最坏情况下，需要遍历整个连通块。

如果提问 X 次合并操作，时间复杂度显然也不能接受。

并查集

我们发现，上面两种办法都有一个操作很快，另一个操作很慢。有没有一种方法，能让两个操作都变得非常快呢？

有！通过并查集去操作！

并查集是怎么做到的呢？

它的核心思想像我们现实生活中的组织架构：**给每个圈子（集合）指派一个唯一的老大（代表元），所有事情都由老大出面。**

我们只需要一个非常简单的数据结构：一个数组，我们叫它 `parent[]`。

- `parent[i]` 存储的是 `i` 号同学的“上级”是谁。

并查集

现在通过**每个圈子（集合），选出一个“老大”（代表）**的并查集的思想。

想判断**两个人是不是一伙**的?

- 直接查查他们的**老大是不是同一个人**!

想让两个圈子**合并**?

- 让一个圈子的老大去**认另一个圈子的老大当新老大**!

算法思路：

1．初始化

- 一开始，所有同学都互不相识，每个人都是自己圈子的老大。
- 我们遍历所有人，让 `parent[i] = i`。这表示 `i` 的上级就是他自己。他自己就是老大。

并查集

2. find(x) 函数：假设现在已经形成了一些圈子，需要寻找圈子里的老大！

find(x) 的目标是：**找出 x 所在圈子的最终老大是谁？**



基本思路（递归查找）：

1. 先问 x 的直接上级 parent[x] 是谁。
2. 如果 x 上级就是我自己（parent[x] == x），那 x 就是最终老大，返回即可！
3. 如果 x 的上级不是自己，那就得去问x的上级：你的老大是谁？ 把这个问题抛给他。
4. 这个过程会像一条链一样不断向上传递，直到找到某个上级的上级是 k，但是 parent[k] == k 的情况，这说明 k 已经没有上级了，那么整个圈子的老大就是 k。

2. find(x) 函数：寻找圈子里的老大！

find(x) 的目标是：**找出 x 所在圈子的最终老大是谁?**

假设我们有这样一个关系链：1 -> 2 -> 3 -> 4（4是老大）。 当我们调用 find(1) 时：

1. find(1) 发现上级是2，于是去调用 find(2)。
2. find(2) 发现上级是3，于是去调用 find(3)。
3. find(3) 发现上级是4，于是去调用 find(4)。
4. find(4) 发现自己的上级是自己，返回给 find(3) 结果 4。
5. find(3) 得到结果 4，返回 find(2) 结果 4。find(2) 得到结果 4，返回 find(1) 结果 4。
6. find(1) 得到结果 4，最终返回 4。

那如果下次在执行 find(1) 会怎么样?

并查集

有没有发现上面这个过程有些啰嗦了。因为小兵 1 每次要找自己圈子里的老大的时候，都要通过班长 2、排长 3 去传话去问才能找到自己圈子里的老大司令 4。

而聪明的做法是：**小兵 1 在第一次见到司令 4 之后，就记住司令的位置，如果下次还需要见到司令的话就可以直接向司令汇报！不需要找别人帮忙了。**

路径压缩：**在我向上查找的返回过程中，把我经过的所有节点，都直接挂到最终老大的名下。**

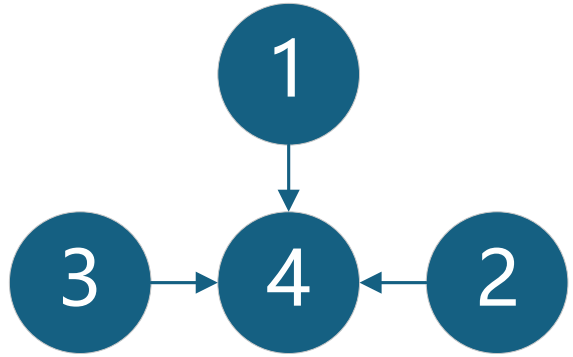
代码实现：

大家先自己去尝试理解

```
int find(int x) {
    if (parent[x] == x) {
        return x;
    }
    // 在递归返回时，用最终老大的结果来更新我自己的parent
    return parent[x] = find(parent[x]);
}
```

并查集

```
int find(int x) {
    if (parent[x] == x) {
        return x;
    }
    // 在递归返回时，用最终老大的结果来更新我自己的parent
    return parent[x] = find(parent[x]);
}
```

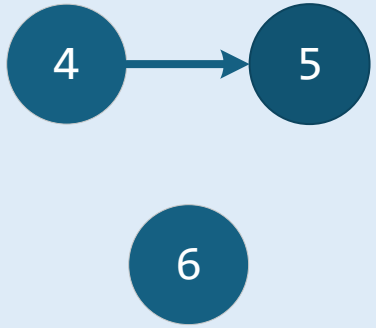


1. `parent[x] = find(parent[x])` 这一行代码是精髓。
2. `find(parent[x])` 会递归找到最终老大。
3. 然后 `=` 赋值操作会把这个最终老大的编号，直接赋给 `parent[x]`。
4. 这就像小兵 `1` 的上级直接变成了司令 `4`。
5. 经过路径压缩后，原来的长链结构就消失了，而下次再执行find，就仅需O(1)的时间复杂度

3. union(u, v) 操作：合并两个圈子

现在我们有了find 工具，合并的操作就变得非常简单了。

u 和 v 成为朋友，要求合并他们的朋友圈。



操作步骤：

- 找到 u 的最终老大 rootU = find(u)。
- 找到 v 的最终老大 rootV = find(v)。
- 判断：
  - 如果 rootU 和 rootV 是同一个人，什么都不用做。
  - 如果他们老大不一样，那就合并：让一个其中老大去认另一个当老大。比如，我们让 u 的老大去认 v 的老大当新老大：parent[rootU] = rootV

并查集

```
void unite(int u, int v) {
    int rootU = find(u);
    int rootV = find(v);
    if (rootU != rootV) {
        parent[rootU] = rootV; // u的老大认v的老大当上级
    }
}
```

假设我们通过 `unite(1, 6)` 这个操作，我们找到了 1 的老大 4 和 6 的老大 6，然后执行了 `parent[4] = 6`。

只改动了一个 `parent` 数组的值是如何就把两个圈子合并的呢?

并查集

原本以 4 为首的圈子（比如有成员 1，2，3），当他们再去执行 find 操作时，会发生什么？

成员 1 去 find 老大，他顺着路径找到了 4。但他会发现，4 不再是最终老大了，因为 parent[4] 不等于 4 了，4 上面有了新的老大 6。

于是，1 的查找过程会继续向上，会查到 4 的老大是 6。最终，1 找到了 6 这个新的老大。

而且因为有路径压缩，在 1 这次查找结束后，parent[1] 会被直接更新为 6！

同样，这个圈子里的其他成员在下次查找时，也会一个接一个地把自己的上级直接更新为 6。

unite 操作就像是一次权力交接。我们只需要让一个老大向另一个老大臣服（修改 parent 值），之后所有小弟们在下一次找老大(find)时，就会自动顺藤摸瓜找到新的最高领导，并且还会通过路径压缩来直接更新自己的归属。

例17-1 亲戚(B0247，P238页)

现在给出某个亲戚关系图，求任意给出的两个人是否具有亲戚关系。

规定：x 和 y 是亲戚，y 和 z 是亲戚，那么 x 和 z 也是亲戚。如果 x，y 是亲戚，那么 x 的亲戚都是 y 的亲戚，y 的亲戚也都是 x 的亲戚。

输入格式

第一行：三个整数 n,m,p,分别表示有 n 个人，m 个亲戚关系，询问 p 对亲戚关系。

以下 m 行：每行两个数 M_i，M_j，$1 \leq M_i,\sim M_j \leq n$，表示 M_i 和 M_j 具有亲戚关系。

接下来 p 行：每行两个数 P_i,P_j，询问 P_i 和 P_j 是否具有亲戚关系。

输出格式

p 行，每行一个 `Yes` 或 `No`。表示第 i 个询问的答案为“具有”或“不具有”亲戚关系。

并查集

对于书本上这道例题，通过刚刚的讲解，大家应该就能够完成了。

查找 x 的老大

1. 先问 x 的直接上级 `parent[x]` 是谁。
2. 如果 x 上级就是我自己（`parent[x] == x`），那 x 就是最终老大，返回即可！
3. 如果 x 的上级不是自己，那就得去问x的上级：你的老大是谁？ 把这个问题抛给他。
4. 这个过程会像一条链一样不断向上传递，直到找到某个上级的上级是 k，但是 `parent[k] == k` 的情况，这说明 k 已经没有上级了，那么整个圈子的老大就是 k。
5. 通过路径压缩，将整个圈子的老大直接赋值给自身，即 `parent[x] == find(x);`

那么这里给出思路，尝试着自行完成本题目！

u 和 v 成为朋友，要求合并他们的朋友圈。

- 操作步骤：
  - 找到 u 的最终老大 `rootU = find(u)`。
  - 找到 v 的最终老大 `rootV = find(v)`。
    - 如果 rootU 和 rootV 是同一个人，什么都不用做。
    - 如果他们老大不一样，那就合并：让一个其中老大去认另一个当老大。比如，我们让 u 的老大去认 v 的老大当新老大：`parent[rootU] = rootV;`

例17-2 村村通(B0248)

某市调查城镇交通状况，得到现有城镇道路统计表。表中列出了每条道路直接连通的城镇。市政府目标是使全市任何两个城镇间都可以实现交通（但不一定有直接的道路相连，只要相互之间可达即可）。请你计算出最少还需要建设多少条道路?

输入格式

每组测试数据的第一行给出两个用空格隔开的正整数，分别是城镇数目 n 和道路数目 m；

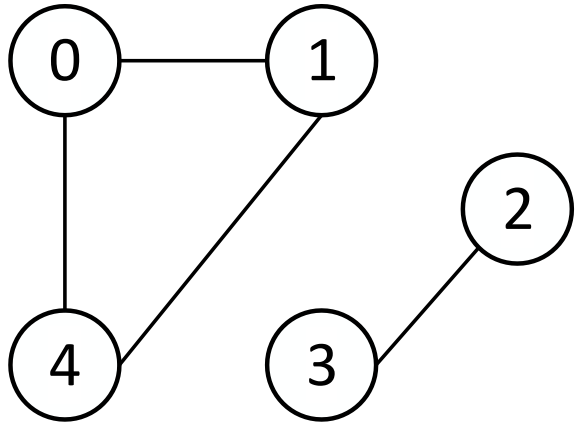
随后的 m 行对应 m 条道路，每行给出一对用空格隔开的正整数，分别是该条道路直接相连的两个城镇的编号。简单起见，城镇从 1 到 n 编号。两个城市间可以有多条道路相通。

在输入数据的最后，为一行一个整数 0，代表测试数据的结尾。

输出格式

对于每组数据，对应一行一个整数。表示最少还需要建设的道路数目。

并查集



- 城镇就是图的**节点**。道路就是图的**边**。我们可以根据输入的数据抽象为一个图。
- 任何两个城镇间都可以实现交通意味着**整个图必须是连通**的，不能互为岛屿。
- 而最少还需要建设多少条道路就是问，最少需要加几条边，才能让整个图**从不连通变为连通**。
- 现在，问题就是：我们有一个由 `n` 个节点和 `m` 条边构成的图，它可能由**好几个互不连通的图**组成。我们的目标是把所有这些图连接起来，需要加几条边？
- 显然可知，需要边的数量为互不连通的图的数量 -1。

并查集

如何计算一个图中连通图的数量?

我们可以把每一个**连通图都**看作一个**集合**。

1. **初始状态：** 一开始，没有任何道路。`n` 个城镇各自互不连通。所以我们有 `n` 个独立的连通图。这就是并查集的初始化操作，每个节点的老大都是它自己。
2. **处理已有道路：** 每输入一条道路 `(u, v)`，就意味着 `u` 和 `v` 这两个城镇连通了。这就是并查集中的 `union`（合并）操作。
    1. 我们检查 `u` 和 `v` 的老大是否是同一个人（`find(u) == find(v)`）。
    2. 如果不是，说明这条路连接了两个之前不连通的图(城镇集合)。我们就执行 `unite(u, v)`，将它们合并。
    3. 如果是，说明 `u` 和 `v` 本来就在同一个连通图里，这条路是多余的（对于连通性而言），我们什么都不用做。

并查集

在所有 m 条道路处理完毕后，我们需要数一下在整个图中还剩下多少个独立的城镇集合。

目标是把所有这些城镇集合连接起来，也就是集合数量-1。

那如何统计独立的集合数量呢?

在每个独立的集合中，都有且仅有唯一的一个老大。

所以想知道有多少个独立的集合，问题就转化成了：**我们只需要统计并查集中老大的数量有多少即可。**

那么，老大的特点是什么呢?

老大一定有着 `parent[i] == i` 这个特征。因为只有老大，他的上级才是他自己。

因此，算法就非常简单了：我们只需要写一个循环，从 `1` 遍历到 `n`，统计所有满足 `parent[i] == i` 条件的城镇 `i` 的数量。统计这个数量 `k`，就是我们最终的独立集合数!

最后只需要输出 `k - 1` 即可。

Hash

哈希表

例17-3 字符串哈希(B0249，P242页)

如题，给定 N 个字符串（第 i 个字符串长度为 M_i，字符串内包含数字、大小写字母，大小写敏感），请求出 N 个字符串中共有多少个不同的字符串。

输入格式

第一行包含一个整数 N，为字符串的个数。

接下来 N 行每行包含一个字符串，为所提供的字符串。

输出格式

输出包含一行，包含一个整数，为不同的字符串个数。

Hash

题目要求给出一大堆字符串，需要**快速数出有多少个不一样的字符串**。最笨的办法是什么？建一个列表，每来一个新字符串，就从头到尾扫一遍列表，看看它在不在。如果字符串有成千上万个，这个办法就太慢了。

那我们换个思路。想象一下，你是一个图书管理员，面前堆着 N 本书，你要把它们分门别类地放到书架上，并且统计有多少种不同的书。

你会怎么做？你肯定不希望每次来一本新书，都跑遍整个图书馆去比对吧？

希望有一个**规则**，让你一看到书名（字符串），就能**立刻知道**它应该被放在哪个书架（数组编号）上。

而这个规则，就是哈希函数！

Hash

我们希望能够：

字符串 -> （通过哈希函数） -> 唯一的书架编号 （哈希值）

理想情况下， 不同的书对应不同的书架。当你拿到一本新书时：

1. 通过哈希函数计算出它的书架号。
2. 通过编号查询书架的位置，直接走到那个书架前。
3. 看看书架上是不是已经有这本书了。

这样，每次查找的时间复杂度就从O(N) 变成了 O(1) 的定点查找

第二步：如何给每个字符串分配一个对应的书架编号？（字符串哈希函数）

怎么把一个字符串（比如 "abc"）变成一个数字（书架编号）呢？

- **想法1： 直接用字符的ASCII码相加。**
  - "abc" -> 97 + 98 + 99 = 294
  - "acb" -> 97 + 99 + 98 = 294
  - 顺序不同的字符串，书架号却一样！这会导致大量不必要的冲突。
  - 所以字符串的位置和顺序很重要！不同的位置代表的数值千差万别。
  - 在数学中有什么最常用的东西的顺序也很重要呢？

Hash

我们平时用的十进制数 123，它的值其实是 $1 * 10^2 + 2 * 10^1 + 3 * 10^0$。每个数字的位置都赋予了它不同的权重。

所以可以对字符串做同样的事，把这个字符串看成一个 P 进制数！

我们选一个进制 P，然后把字符串 s = “abc” 看成一个 P 进制数：

$$'a' * P^2 + 'b' * P^1 + 'c' * P^0$$

这个值可能会非常巨大，所以我们最后再对一个巨大的质数 M（比如 10^9 + 7）取模，把它压缩到一个可控的范围内，hash(x)代表着求取 x 的哈希值。

$$hash("abc") = ('a' * P^2 + 'b' * P^1 + 'c' * P^0) \% M$$

Hash

如何用代码高效计算 ('a' * P² + 'b' * P¹ + 'c' * P⁰) % M ?

我们不需要真的算P的次方，因为这个公式可以通过 **hash = (hash * P + s[i]) % M** 迭代计算：

1. 初始化：**hash = 0，设 P = 10。**
2. 读入 'a'：hash = (hash * 10 + 'a') % M 结果是 'a'
3. 读入 'b'：hash = (hash * 10 + 'b') % M 结果是 ('a' * 10 + 'b')
4. 读入 'c'：hash = (hash * 10 + 'c') % M 结果是 (('a' * 10 + 'b') * 10 + 'c')，展开就是 ('a' * 10² + 'b'* 10 + 'c')

可以通过这个迭代公式 hash = (hash * P + s[i]) % M 实现了我们想要的效果！

也可以这样思考：**因为我们是从字符串末尾，也就是个位一个个读入的，每读取到新的一位，就代表原本读入的结果应该向小数点往右挪一位，在十进制中也就是乘以10，然后加上当前的个位。**

Hash

现在我们就可以将某一个字符串通过刚刚的哈希函数转变为一个数值编号了。

但是我们的图书馆书架数量是有限的（模数 `M` 的大小，也就是编号只能在 `0 ~ M-1` 的范围以内），但世界上书的数量是无限的（字符串的数量）。

根据鸽巢原理，**必然**会有不同的书（字符串）被分配到同一个书架编号（哈希值）。这种情况，就叫**哈希冲突**。如果我们的哈希函数足够好，可以让冲突变少，但无法完全避免。

那如果发生了哈希冲突，也就是当图书管理员发现两本不同的书要放到同一个书架上时，我们会怎么办？直接把其中一本书扔掉吗？

肯定不会扔掉，大部分的想法是会拿一个**箱子**，把所有要放在这个书架上的书都放进这个箱子里。故这个书架不能直接储存单个书本，而是通过数组储存这个书架上的所有书。

Hash

所以一个编号所对应的书架就应该是一个**数组**，里面存放了所有通过哈希函数计算后编号相同的书本。而这就是解决哈希冲突的方法之一：**链地址法**。

所以数据结构要定义为二维数组：

```
vector<vector<string> > hashTable(SIZE);
```

- 本质上该结构是一个由 `vector` 构成的**桶数组**。`hashTable` 是一个拥有 `SIZE` 个元素的数组，代表有 `SIZE` 个桶(书架)。
- `hashTable[i]` 是第 `i` 个桶，它本身是一个 `vector`，而所有哈希后地址为 `i` 的数据（即 `key % SIZE == i`），都会被存储在 `hashTable[i]` 这个 `vector` 中。
- 通过将冲突的数据组织在同一个桶里，既保证了能找到所有数据，又将查找范围从整个数据集缩小到了一个桶内，从而实现了高效。**而这个整体结构就我们称为哈希表**。

可以得出 **求 *N* 个字符串中共有多少个不同的字符串** 的解题思路：

1. 我们的哈希表，要定义为二维数组。 `vector<vector<string> > hashTable(SIZE);`
2. 来了一个新字符串 `s`，通过哈希函数算出它的书架编号 `h = hash(s)`。这个 `h` 可能很大，需要通过 `h = h % SIZE;` 来确保索引的范围是合法的。
3. 我们在 `hashTable[h]` 这个 箱子 里去查找。
4. 遍历 `hashTable[h]` 这个小箱子，挨个比较里面的字符串和新来的 `s` 是否完全一致。
   - 如果找到了一个一模一样的，说明这本书已经有了，我们什么都不用做。
   - 如果把这个小数组遍历完毕了都没找到，说明这是一个全新的字符串！
5. 那就把新字符串 `s` 扔到这个箱子里：`hashTable[h].push_back(s)`，并且把我们的“不同字符串总数”加一。

Hash

接下来我们就来实际编写本题目的哈希函数 int hash(string str)

```
const int P = 131;
const int TABLE_SIZE = 1e6;
const long long MOD = 1e9 + 7;
int hash_string(const string& str) {
    long long h = 0;
    for (char c : str) {
        h = (h * P + c) % MOD;
    }
    return h % TABLE_SIZE;
}
```

需要实现实现：

字符串 -> （通过哈希函数） -> 唯一的书架编号（哈希值）

通过哈希函数就能够对任意字符串都找到一个大概率唯一的编号，就可以通过这个编号快速的在数组中存放、读取、查询该字符串。

Hash

在主函数中，计算每个字符串 s 的哈希值，根据哈希值检查哈希表中是否已经存在了字符串 s，如果不存在，将其放入，且因为这是一个不同的字符串，故将计数器加 1 。

```
int N;   cin >> N;
vector<vector<string> >
words(TABLE_SIZE);
int cnt = 0;
```

```
for (int i = 0; i < N; i++) {
    string word;  cin >> word;
    int hash = hash_string(word);
    bool found = false;
    for (string s : words[hash]) {
        if (s == word) {
            found = true; break;
        }
    }
    if (!found) {
        cnt++;  words[hash].push_back(word);}
    }
}
cout << cnt;
```

哈希表总结

1. **地址映射**：我们使用哈希函数（如gethash）将某个类型的数据映射为一个整数键值 key。然后通过取模运算 key % SIZE，将这个键值转换为一个在桶数组范围内的索引 bucket_index。这个桶数组就是哈希表的主体。

2. **冲突处理**：不同的键值可能会被映射到同一个 bucket_index，这称为哈希冲突。我们采用**链地址法**来解决。具体实现上，我们将桶数组的每个元素定义为一个数组。

3. **查询**：以后任意的某类型数据都可以通过哈希函数映射为一个整数键值 key，而通过这个键值就可以快速的在桶数组中查询到 **是否有对应的元素** 或 **与该元素相关的数据**。

例17-4 Cities and States S(B0250)

Farmer John 有若干头奶牛。为了训练奶牛们的智力，Farmer John 在谷仓的墙上放了一张美国地图。地图上表明了每个城市及其所在州的代码（前两位大写字母）。

由于奶牛在谷仓里花了很多时间看这张地图，他们开始注意到一些奇怪的关系。例如，FLINT 的前两个字母就是 MIAMI 所在的 FL 州，MIAMI 的前两个字母则是 FLINT 所在的 MI 州。

确切地说，对于两个城市，它们的前两个字母互为对方所在州的名称。

我们称两个城市是一个一对「特殊」的城市，如果他们具有上面的特性，并且来自不同的州。

对于总共 N 座城市，奶牛想知道有多少对「特殊」的城市存在。请帮助他们解决这个有趣的地理难题！

题目的问题本质

题目要求寻找“特殊的一对”城市，满足条件：

- 城市 1 的前两个字母 == 城市 2 的所在州代码
- 城市 2 的前两个字母 == 城市 1 的所在州代码
- 两个城市来自不同的州

简化思路： 对于每一个城市，我们只需要关心两个信息：

1. **A** ：城市名的前两个字母。**B** ：城市所在的州代码。

那么每个城市都可以简化为一个组 `(A, B)`。 我们要寻找两个城市 `(A1, B1)` 和 `(A2, B2)`，使得：

- $A1 = B2$ 且 $A2 = B1$ 且 $B1 \neq B2$

这等价于：我们处理一个城市 `(A, B)`，我们想知道之前有没有出现过 `(B, A)`，且 $A \neq B$。如果有，这一对就可以凑成一个“特殊的一对”。

Hash

算法思路

- 每个城市都可以看作一个二元组 (A, B)，其中 A 是城市名前缀，B 是州代码。
- 目标：对于我们正在处理的每一个 (A, B)，都需要找到在这之前已经出现过的，能够与之配对的 (B, A) 的数量。
- 约束条件：A 和 B 不能相同（即 A ≠ B），因为要求来自不同的州。

那么当我们处理第 i 个城市时，如何**快速地**知道在前面 i-1 个城市中，与它能配对的城市出现了多少次？例如对于(MI, FL)而言，如何快速找到与之配对的(FL, MI)出现了多少次。

如果每来一个新城市，都回头去遍历一遍之前的所有城市，时间复杂度将是 $O(N^2)$，对于 $N = 200{,}000$ 的数据量来说太慢了，一定会超时。

需要一个能够快速查找的数据结构，而之前学习**哈希表**就可以处理需要快速查找的情况。

也就是说我们要自己动手实现一个哈希表来管理这些（城市前缀，州代码）的组合，我们需要解决三个关键问题：

1．将（A，B）的字符串组合转换成一个数字索引

计算机擅长处理的是数字，下标也需要是数字。如果我们能把（"MI"，"FL"）这样的组合转换成一个独一无二的整数，事情就好办多了。这个转换过程就是**哈希函数**。

一个简单的方法是采用**进制转换**的思想。因为城市前缀和州代码都由大写字母构成（共26种可能），我们可以把一个四字母组合 P1 P2 S1 S2 看作一个26进制的数。

Hash

gethash 函数的实现：

```
// 将 (P, S) 的字符串组合转换为唯一的整数的哈希值的哈希函数
int gethash(string P, string S) {
    int p1 = P[0] - 'A'; // 'A'->0, 'B'->1 ...
    int p2 = P[1] - 'A';
    int s1 = S[0] - 'A';
    int s2 = S[1] - 'A';

    // 按照26进制计算，每一位权重为 26^0, 26^1, 26^2, 26^3
    return p1 + p2 * 26 + s1 * 26 * 26 + s2 * 26 * 26 * 26;
}
```

通过这个函数，类似于 (“MI”, “FL”) 的二元组会被转换成固定的整数。一般把这个整数称为**键值**。

思考：上述哈希函数有可能产生冲突吗？例如 “MI, FL” 和 “TX, DA”，经过 gethash 计算后，是否有可能会得到**完全相同**的整数结果？

不会导致冲突，使用26进制可以保证一一对应

想象一下我们最熟悉的**十进制**。为什么我们能肯定 12 这个数和 22 不一样？因为十进制要求：**每个位置上的数字只能是 0 到 9**。

当你数数数到 9，下一个数不是发明一个新符号，而是**向左边的“十位”进一**，自己变回 0，组合成了 10。这个“逢十进一”的规则，保证了任何一个整数都有独一无二的表示方法。

在十进制中，表示“123”的这个数字有且仅有这一种办法。而使用 26 进制，就可以保证表示某个字母串时有且仅有一种办法。

Hash

我们可以用一个生活中的例子来举例：

假设你只有两种面值的钱：**1元** 和 **10元**。你想凑出 **12元，且如果使用一元满十元要更换为十元纸币。**

- 凑法：一张 10元 + 两张 1元。 且这种组合是唯一的。 对应十进制就是：`1 * 10 + 2 * 1 = 12`

现在，如果允许**不遵守进位规则**，还可以怎么凑？

- 零张 10元 + 十二张 1元。 强行对应到十进制上就是：`0 * 10 + 12 * 1 = 12`

我们用两种**不同**的纸币组合方式，凑出了**完全相同**的总金额！

- 组合一：(一张10元，两张1元)
- 组合二：(零张10元，十二张1元)

如果我们把凑钱的总金额当作哈希值，那么两个完全不同的原始输入，产生了相同的哈希值(哈希冲突)。

Hash

前面讲解过，为什么十进制可以保证任何一个整数都有独一无二的表示方法。因为十进制要求：**每个位置上的数字只能是 0 到 9**，逢十进一。

- 组合二：（零张10元，十二张1元）

那么对于组合二而言，十二张 1 元**本来应该进位却没有进位**，就导致了用两种不同的数字组合方式，可以凑出了完全相同的数值！

现在，我们把这个思想应用到我们的字母问题上：

- 我们有多少个“符号”？'A' 到 'Z'，共 26 个。
- 我们将 'A' 映射为 0，'B' 为 1，... 'Z' 为 25。
- 因此，我们这里有 26 个数，为了保证一一对应，我们就应该采用 26 进制。而任何一个不同的四字母组合，通过 26 进制都一定会产生一个独一无二的整数值。这个性质在数学上称为**单射**。

步骤二：存放键值数据

现在我们通过哈希函数对字符串求哈希后，可以获得一个对应的唯一整数键值，但这些键值可能非常大。我们不可能创建一个能容纳所有可能键值的巨大数组。

所以，我们只能创立一个固定大小的存储空间，通常是一个数组。我们也称为桶数组。

那么如何把一个巨大的键值 key 映射到这个小小的桶数组里呢？

通过取模运算。

```
#define SIZE 233333 // 选择一个数字作为桶的数量，最好选个质数

// ...
int key = gethash(A, B);
int bucket_index = key % SIZE; // 计算它应该放在哪个桶里
```

bucket_index 就是字符串数据在桶数组中的下标。

3．处理哈希冲突

如果两个不同的组合，比如 ("AA", "AA") 和 ("AB", "BA")，它们的键值经过取模运算后，得到的 bucket_index 恰好相同怎么办？这种情况称为**哈希冲突**。

解决方法是**链地址法**。我们不让每个桶只存放一个数据，而是让每个桶都成为一个**链表**（或者 vector）。所有被映射到同一个桶的数据，都依次挂在这个链表上。

而且我们需要找到与之配对的城市**出现了多少次**，所以，不光要储存某个城市出现过了，**还要储存次数**。之后需要使用某个能与之配对的伴侣城市时，就可以在表中直接查询了。

```
// 哈希表数据结构
vector<pair<string, int>> hash_table[SIZE];
// pair.first 是原始的四字母字符串，如 "MIFL"
// pair.second 是出现次数
```

总结：算法流程

现在把这三步整合起来，形成最终的算法流程：

1. 准备工作：

- 定义一个大的质数 `SIZE`。
- 创建一个哈希表桶数组 `vector<pair<int, int>> hash_table[SIZE]`。
- 初始化答案 `long long ans = 0`。

2. 遍历每个城市 `(A, B)`：

- 因为题目要求一对特殊的城市要来自不同的州，所以如果 `A == B`，跳过。

遍历每个城市 (A, B) 查找其对应的伴侣城市：

- 计算伴侣城市 (B, A) 的键值：`partner_key = gethash(B, A)`。
- 计算伴侣在哈希表中的桶的位置：`bucket_index = partner_key % MOD`。
- 遍历 `hash_table[bucket_index]` 这个小箱子，在里面查找 `pair` 中 `first` 值等于 `partner_key` 的元素。
- 如果找到了，因为当前城市可以和每个伴侣城市凑成一对，所以就把它的 `second`（出现次数）加到 `ans` 上。

登记自己：

- 当前城市也需要加入到哈希表中，等待将来可能被遍历到的其他城市所配对。
- 计算自己 `(A, B)` 的键值：`self_key = gethash(A, B)`。
- 计算自己所在的桶：`bucket_index = self_key % MOD`。
- 遍历 `table_size[bucket_index]` 这个链表，查找 `pair` 中 `first` 值等于 `self_key` 的元素。
- 如果找到了，就将其 `second`（出现次数）加一。
- 如果没找到，就在这个链表的末尾添加一个新的 `pair`：`{self_key, 1}`。

Hash

查询有多少个伴侣城市关键代码(**查询**操作)：

在哈希表中查找键值 key 的出现次数。

```
int find(int key) {
    int bucket_index = key % SIZE;
    // 遍历桶内的数组
    for (int i = 0; i < hash_table[bucket_index].size(); i++) {
        // 通过原始键值进行匹配
        if (hash_table[bucket_index][i].first == key) {
            return hash_table[bucket_index][i].second; // 找到，返回出现次数
        }
    }
    return 0; // 没找到，返回0
}
```

Hash

关键代码(**插入**操作):

对于新遍历到的一个城市，将其**插入到哈希表**中 或 已经存在**需要更新计数值**。

```
// 将键值 key 插入哈希表，或更新其计数值
void insert(int key) {
    int bucket_index = key % SIZE;
    // 遍历桶内的链表，检查是否已存在
    for (int i = 0; i < hash_table[bucket_index].size(); i++) {
        if (hash_table[bucket_index][i].first == key) {
            hash_table[bucket_index][i].second++; // 已存在，次数加一
            return; // 完成操作直接退出
        }
    }
    // 若不存在，则在链表末尾添加新元素
    hash_table[bucket_index].push_back({key, 1});
}
```

接下来就只需要在主函数中：

遍历每个城市 (A, B)，通过 find() 函数在哈希表中查找对应的伴侣城市，再通过 insert 函数插入到哈希表中，最终输出 ans 即可。

在 OJ 上完成 B0250 这道题目

哈希表总结

核心思想

哈希表，又称**散列表**，是一种能够实现**极快查找、插入和删除**操作的数据结构。

它的核心思想是：想象一个字典，如果你知道一个**单词（键 Key），**你不想从第一页翻到最后一页，而是希望通过某种方式能**立即翻到它所在的那一页（地址）**，直接获取单词的**释义（值 Value）**。

哈希表就是能偶实现这种立即翻到的数据结构。

哈希表总结

而使用哈希表依赖于二个部分：

1．哈希函数

作用：它负责将任意类型的键（如字符串 ）转换成一个整数，这个整数被称为**哈希编码或 键值。**

目标：

- 快速计算：转换过程需要快。
- 均匀分布：应尽可能地将不同的键映射到不同的哈希码上，避免扎堆。
- 一致性：对于同一个键，每次计算出的哈希码必须完全相同。

哈希表总结

而使用哈希表依赖于二个部分：

2．桶数组

- 作用：一个固定大小的数组，是数据实际存储的地方。数组的每一个位置被称为一个桶。
- 下标计算：通过取模运算 哈希码 % 数组大小，将一个可能非常大的哈希码映射到桶数组的一个有效下标上，而桶内的元素就是键值所对应的数据。

但是由于桶数组的大小是有限的，不同的键经过哈希和取模运算后，可能会被分配到同一个桶中，这种情况称为**哈希冲突**。最常用的**解决方案**是**链地址法**。

链地址法原理

为了解决不同元素却产生了相同哈希值的哈希冲突问题，可以不让每个桶只存放一个元素，而是让每个桶都成为一个链表（或 `vector` 等动态容器）。

- **流程：**所有被映射到同一个桶的元素，都依次添加到这个桶的链表中。
- **精确查找：**当通过键值查找元素时，先定位到桶，然后遍历这个（通常很短的）链表或动态数组，通过比较原始键（或其原始的数据）来找到我们真正要找的那个元素。

Hash

哈希表平均情况时间复杂度：O(1)

- 哈希函数够好，数据能被均匀地分布在各个桶中，每个桶的链表长度就会非常短，查找、插入、删除操作就像直接访问数组一样。故我们一般认为哈希表的时间复杂度是近似 O(1) 的。

哈希表最坏情况时间复杂度：O(N)

- 这种情况会发生在一个糟糕的哈希函数将所有 N 个元素都映射到同一个桶里时。
- 此时，哈希表退化成一个长长的数组，所有操作都变成了对这个数组的线性遍历。

哈希表本质上是一种空间换时间的策略。我们预先分配了一个可能比实际数据量大得多的桶数组（空间），目的就是为了减少哈希冲突，将查找操作的时间复杂度从 $O(N)$ 降低到近乎 $O(1)$。

哈希表通过**哈希函数**将无规律的、复杂的键映射到了有规律的数组下标上，并通过**链地址法**处理了不可避免的冲突，是计算机科学中应用最广泛、最重要的基础数据结构之一。

Hash

我们刚刚亲自设计并实现了一个哈希表，虽然能正常工作，但同样也有很多缺点：代码量大、需要自己处理很多底层细节、容易出错。

但是 C++ 的工程师已经想到了这些繁琐的工作每次都做会非常麻烦，于是在 C++ 内部，工程师帮助我们封装了一个强大、易用且高效的工具 —— `unordered_map`。

而 `unordered_map` 的底层原理和我们手写哈希表的原理是完全一致的。

手动实现	`unordered_map` 的内部实现
`gethash`函数 + `% SIZE`	**内置了多种类型的哈希函数**和**取模**过程。
`vector<pair<int, int>> hash_map[SIZE]`	内置了动态管理的**桶数组**和**防哈希冲突链**。
`find()` 和 `insert()` 函数	内置高效的成员函数，如 `find()`，`insert()`，以及更方便的 `[]` 运算符。

而 unordered_map 的名字其实也表明了它的特性：

- map：意味着它是存在一个“映射关系”（Key -> Value）。
- unordered：意味着它内部元素储存的顺序是无序的。

unordered_map 的定义语法为： unordered_map<T1, T2>

代表着定义了一个**键类型**为 T1，**值类型**为 T2 的哈希表。

unordered_map 的定义语法为： unordered_map<T, T>

例如定义一个键值类型为 int，值类型为 string 的哈希表其语法为：

```
unordered_map<int, string> // 可以通过一个 int 类型键找到其一一对应的 string 类型值
```

Hash

并且 unordered_map 提供了一个极其方便的操作符 []

对于哈希表 unordered_map<int, int> umap;

代码中书写 umap[key] 这个表达式，会执行以下判断：

如果 key 已经存在于 umap 中(假设为 5)：

- 它会返回这个 key 所对应的那个值的引用(原件)。我们可以直接读取或修改它。
- 例如 int num = umap[5];，如果键值为 5 的元素存在且值为3，umap[5] 就会返回3。

所以对于一个哈希表 umap，使用下标运算符 umap[key] 如果 key **已经存在于**哈希表中的情况下，意义为查询哈希表中 key 这个键所对应的值。

如果调用 umap[key] 但是 key 不存在于 umap 中：

- unordered_map 会**自动地、立即地**在内部创建一个新的键值对。
- 这个新条目的键就是 key。值则是该值类型的默认值，对于 int 类型，默认值就是 0。string 类型，默认值是空串。
- 如果 key 不存在于哈希表中，值类型为 int 的 umap[key] 会返回为0。

所以对于一个哈希表 counts，使用下标运算符 counts[key] 如果 key **不存在**于哈希表中的情况下，意义新建一个键值对，**键为key，值为类型默认值。**

insert 成员函数

以下的实例演示为定义了 unordered_map<string, int> umap;

虽然 [] 运算符很方便，但有时我们需要更精细的插入控制。此时就可以使用 insert 函数。

- 函数原型：pair<iterator, bool> insert(const Tvalue& val);
- 用法：umap.insert({"apple", 10});
  - 它尝试插入一个键值对 (在大括号内部中左侧参数为键，右侧参数为对应的值)。

insert 函数有返回值，它会返回一个 pair，这个 pair 包含了两个信息：

1. first (一个迭代器 iterator)：指向刚刚插入的元素或者已存在的那个元素。
2. second (一个布尔值 bool)：
   - 如果返回 true：表示插入成功，因为键之前不存在。
   - 如果返回 false：表示插入失败，因为键已经存在了。

find 成员函数

以下的实例演示为定义了 unordered_map<string, int> umap;

iterator find(const TKey& key)

- 作用：在哈希表中查找键为 key 的元素。
- 返回值是一个迭代器：
  - 如果找到了，返回一个指向该键值对的迭代器，是 pair<Tkey, Tvalue> 类型。
  - 如果没找到，返回一个特殊的迭代器 umap.end()。

```
auto it = umap.find("apple");
if (it != umap.end()) {
    // 找到了, it->first 就是键 "apple", it->second 就是值 10
    cout << it->first << " " << it->second << endl;
} else {
    cout << "not found." << endl; // 没找到
}
```

count 成员函数

以下的实例演示为定义了 unordered_map<string, int> umap;

int count(const TKey& key)

- 作用：计算键为 key 的元素有多少个。
- 返回值：因为 unordered_map 的键是唯一的，所以返回值只可能是 0（不存在）或 1（存在）
- 用法：可以用来判断某个键存在与否。

```
if (umap.count("apple")) { // 如果返回 1，等价于 true
    cout << "exists." << endl;
} else { // 返回 0，等价于 false
    cout << "not exist." << endl;
}
```

erase 成员函数

以下的实例演示为定义了 unordered_map<string, int> umap;

通过键删除：int erase(const TKey& key);

- 作用：删除键为 key 的键值对。
- 返回值：返回被删除的元素数量（对于 unordered_map 总是 0 或 1）。
- 用法：umap.erase("apple");

通过迭代器删除：iterator erase(iterator position);

- 作用：删除迭代器 position 指向的那个键值对。
- 返回值：返回被删除元素的下一个元素的迭代器。

```
auto it = umap.find("apple");
if (it != umap.end()) {
    umap.erase(it);
}
```

容量、状态 成员函数 以下的实例演示为定义了 `unordered_map<string, int> umap;`

这些函数用来获取哈希表自身的状态信息。

- `empty()`
  - 作用：检查哈希表是否为空。
  - 返回值：`true` (空) 或 `false` (非空)。
- `size()`
  - 作用：返回哈希表中键值对的数量。
- `clear()`
  - 作用：清空哈希表，删除所有元素。

遍历哈希表

如何访问哈希表中的所有元素？通常使用**基于范围的 for 循环**。但是一般来说较少需要遍历整个哈希表，因为哈希表是一个**高频用于查询、插入、更新**的数据结构。

```
unordered_map<string, int> fruit = {{“apple”, 5}, {“banana”, 3}};

// 使用 auto 自动推断类型为 pair<const string, int> 比较方便
for (const auto& pair : fruit) {
    // pair.first 是键 (key)
    // pair.second 是值 (value)
    cout << pair.first << " " << pair.second << endl;
}
```

重要提示：因为是 unordered_map，所以遍历输出的顺序是不确定的，可能每次运行都不一样。

我们可以用 unordered_map 重写城市与州的这道问题

```
unordered_map<string, int> counts;
long long ans = 0;

for (int i = 0; i < n; i++) {
    // ...省略输入的部分代码，city_prefix 为城市的前缀、state 为州的名称
    string partner_key = state + city_prefix;
    string self_key = city_prefix + state;

    // 需要查询伴侣城市的数量 partner_key 看能凑成多少特殊城市对
    // 1. 查找伴侣：如果 partner_key 不存在，counts[partner_key] 会返回0
    ans += counts[partner_key];
    // 2. 登记自己：如果 self_key 不存在，会先创建 {self_key, 0}，再++变为1
    counts[self_key]++;
}
```

Hash

刚刚已经掌握了 unordered_map 这个强大的工具。我们知道通过键，它就能**在**平均 O(1)的时间内把值查找出来。它的优点是速度，但它表里的东西是**无序**的（unordered）。

但是如果我不仅想快速找到东西，还希望这个表里的所有东西，都能自动按照名字的字母顺序排列整齐，该怎么办?

比如，在“学生管理系统”中，如果我们想打印一份按学生姓名字母顺序排列的全班成绩单，unordered_map 能做到吗?

不能，通过基于范围的 for 循环它只会给我们一个乱序的列表。

map：有序的映射表

unordered_map 是追求最快速度，而 map 是追求顺序不那么追求速度的映射表。

- **特性：**放任何键值对放到 map 中，它会自动地根据键的字母顺序（或数字大小），把它插到正确的位置上。而表内永远保持整洁有序。
- **底层原理**：map 的来自它的内部结构 —— 是由一棵叫做红黑树的“自平衡二叉搜索树”来保证它的有序性的。而我们目前不需要知道红黑树的细节，只需要知道：这种树状结构，保证了它的有序性。

Hash

map 对比 unordered_map

特性	unordered_map	map
底层原理	**哈希表**	**红黑树** (自平衡二叉搜索树)
元素顺序	**无序** (混乱的)	**有序** (按 键 自动排序)
操作性能	平均 **O(1)** (快)	**O(log N)** (比 O(1) 慢)
核心优势	**速度**	**有序性** 和 **基于顺序的查找**

对于 map 的使用方法，有一个好消息，因为对于插入、删除、查询操作等常见操作而言，map 和 unordered_map 的语法**几乎完全一样**！

map 的使用方法

```cpp
int main() {
    map<string, int> students; // 只需要改变类型为 map

    // [] 操作符用法完全相同!
    students["Tom"] = 90;
    students["Alice"] = 95;
    students["Jerry"] = 85;

    // count() 和 find() 用法完全相同!
    if (students.count("Tom")) {
        cout << "Tom's score: " << students["Tom"] << endl;
    }

    // erase() 和 size() 用法也完全相同!
    students.erase("Jerry");
    cout << mp.size();
}
```

map 的使用方法

```cpp
int main() {
    map<string, int> students; // 只需要改变类型为 map

    // [] 操作符用法完全相同!
    students["Tom"] = 90;
    students["Alice"] = 95;
    students["Jerry"] = 85;

    // count() 和 find() 用法完全相同!
    if (students.count("Tom")) {
        cout << "Tom's score: " << students["Tom"] << endl;
    }

    // erase() 和 size() 用法也完全相同!
    students.erase("Jerry");
    cout << mp.size();
}
```

map 的遍历，同样使用基于范围的 for 循环

```cpp
int main() {
    map<string, int> students;

    students["Tom"] = 90;
    students["Alice"] = 95;
    students["Jerry"] = 85;

    for (const auto& pair : students) {
        cout << pair.first << ": " << pair.second << endl;
    }
}
```

对于 map 的遍历，因为map是有序的，所以输出顺序会是

Alice: 95; Tom: 90; Jerry: 85

因为 map 按照键进行排序，而 A 在 T 前面(也就是树的中序遍历的顺序)

因为 map 是有序的，所以还可以解决一些 unordered_map 很难解决得问题，例如查找一个元素的 前驱/后继。

在以前我们学习了二分查找，可以在有序的区间内快速找到目标值，而 map 也是有序的，所以支持使用 lower_bound 和 upper_bound 函数。

```
lower_bound(mp.begin(), mp.end(), key);
upper_bound(mp.begin(), mp.end(), key);
```

它的意义分别是：

- lower_bound(key)：查找第一个大于或等于 key 的元素。
- upper_bound(key)：查找第一个严格大于 key 的元素。

对于选择使用 unordered_map 还是 map，主要取决于**核心需求**：

如果只关心“键 -> 值”的快速映射，完全不在乎顺序就选择 unordered_map

如果需要让数据保持有序，或者需要查找某个元素的前驱/后继就选择 map

简单来说：默认用 unordered_map 追求速度，需要顺序时才换成 map

集合

例17-5 木材仓库(B0251)

博艾市有一个木材仓库，里面可以存储各种长度的木材，但是保证没有两个木材的长度是相同的。作为仓库负责人，你有时候会进货，有时候会出货，因此需要维护这个库存。有不超过 10^5 条的操作：

- 进货，格式 `1 Length`：在仓库中放入一根长度为 Length（不超过 10^9）的木材。如果已经有相同长度的木材那么输出 `Already Exist`。
- 出货，格式 `2 Length`：从仓库中取出长度为 Length 的木材。如果没有刚好长度的木材，取出仓库中存在的和要求长度最接近的木材。如果有多根木材符合要求，取出比较短的一根。输出取出的木材长度。如果仓库是空的，输出 `Empty`。

集合

首先，分析题目需求，它对这个木材仓库提出了哪些要求：

1. **没有两根木材长度相同：**这意味着我们存放木材长度的容器，必须能保证元素的唯一性。
2. **进货/出货：**我们需要能方便地插入和删除元素。
3. **最接近的木材：**这是核心要求。要找到“最接近”，我们必须能快速地在一个有序的集合中进行查找，找到目标值的左邻右舍。

于是可以将这个问题抽象为：

维护一个集合，可以插入一个元素，且同时判断 x 是否已经存在；

查询 x 的前驱后继，x 的前驱定义为小于 x 的最大的数，x 的后继为大于 x 的最小的数；且可以删除指定元素。

而 C++ STL 提供了一个工具，正好能满足这个问题的需求。它就是 set(集合)！

集合

set 的定义：`set<T> 变量名;`

意义：创建一个用来存放数据类型为 T 的集合。

对于本题目，需要定义：`set<int> wood_set;`

- 作用：创建一个名为 `wood_set` 的集合，专门用来记录 `int` 类型的木材长度。
- 而 `set` 有如下的特性：
  - 自动去重：如果尝试存放两个相同的数据进去，只会记录下一个，不会重复记录。
  - 自动排序：不管按什么顺序插入，它内部总是会按照升序进行排列记录。
  - 因为 set 内部其实是一颗平衡二叉搜索树，所以在内部能保证其有序性。

处理进货操作

进货操作（输入格式：`1 Length`）要求我们：

1. 放入一根长度为 `Length` 的木材。
2. 如果已经有了，就输出"Already Exist"。

这正好对应了 `set` 的两个基础操作。

1.插入：`wood_set.insert(length);`

- 作用：把 `length` 插入到集合 `wood_set` 中。由于 `set` 的特性，它会自动排序并处理重复。

如何判断数据是否已经存在?

也就是需要在插入前检查 `set` 中是否已经存在了这个记录。存在需要输出"Already Exist"

查找某元素是否在集合中： `wood_set.find(length);`

- 作用：在集合中查找 `length`。
- 返回值：它会返回一个为迭代器。
  - 如果找到了，它返回一个指向元素 `length` 的迭代器。
  - 如果没找到，它会返回末尾迭代器 `wood_set.end()`。它本身不是一个有效的位置，只是作为一个未找到的标志而已。

集合

根据题目要求，我们可以写出代码：

1. 放入一根长度为 Length 的木材。
2. 如果已经有了，就输出"Already Exist"。

```
if (wood_set.find(length) != wood_set.end()) {
    cout << "Already Exist\n";
} else {
    wood_set.insert(length);
}
```

处理出货操作

出货操作（输入格式：`2 Length`）它要求我们取出“最接近”的木材。

首先，最简单的情况：仓库是空的。

判断集合是否为空： `wood_set.empty();`

- 作用：检查集合是否为空。如果是，返回 `true`。
- 用法：`if (wood_set.empty()) { cout << "Empty\n"; }`

接下来是比较复杂的部分：**如何定义和寻找 最接近的木材 呢?**

如何定义和寻找最接近的木材呢?

在一个排好序的队伍里，离你最近的人，无非就是你**左边**紧挨着你的那个人（前驱），和你**右边**紧挨着你的那个人（后继）。

所以，对于要求的 `Length`，我们要找的“最接近”的木材，只可能是：

- 大于或等于 `Length` 的那群木材里，最矮的那一根。
- 严格小于 `Length` 的那群木材里，最高的那一根。

对于第一个条件应该可以想到，这不就是我们以前在二分算法中讲解到的下界吗?

而且在前面提到过，`set` 拥有自动排序的特性，所以 `set` 支持使用二分查找!

```
auto it = lower_bound(wood_set.begin(), wood_set.end(), length);
```

lower_bound 用来寻找下界，也就是大于或等于 Length 的那群木材里，最矮的那一根。

而对于另一个候选者：严格小于 Length 的那群木材里，最高的那一根。

不就是通过 **lower_bound 找到的下界** it 的左边的那个吗？故只需要让迭代器 -1 即可。

找到这两根候选的木头后，就是简单的比大小了： 看看 length 离“候选者A”和“候选者B”哪个距离目标值更近。如果距离相同，选长度短的那个。

执行出货

当确定了要取出的木材 `choose` 之后，就要把它从集合中删除掉。

移除元素：`wood_set.erase(value);`

- 作用：删除值为 `value` 的记录。
- 用法：`wood_set.erase(best_choice);`

并且 `set` 允许用迭代器来删除，比如 `wood_set.erase(it_A)`，效果是一样的

之后对于每一个出货/进货需求都执行上述的流程，输出对应的数据即可解决本问题。

集合

通过木材仓库问题，我们了解了 set 的各项强大功能：

- 用 set<T> 创建一个自动排序、去重的集合。
- 用 insert 添加元素。
- 用 find 和 end 判断元素是否存在。
- 用 empty 检查是否为空。
- 使用 lower_bound 在有序集合中进行定位，从而解决复杂的查找最近邻居的问题。
- 最后用 erase 删除元素。

掌握了 set 的基本操作后就可以尝试用去编写代码来解决本问题了！