第 3 章 数据结构

1. 单调栈

1. 什么是单调栈?

单调栈，顾名思义，就是具有单调性的栈。它依旧是一个栈结构，只不过里面存储的数据是递增或者递减的。这种结构是很容易实现的（如下面的代码），但重点是维护一个单调栈的意义是什么?

```cpp
#include <iostream>
#include <stack>

using namespace std;
const int N = 3e6 + 10;
int a[N], n;

void test1()
{
    stack<int> st; // 维护一个单调递增的栈
    for(int i = 1; i <= n; i++)
    {
        // 栈里面大于等于 a[i] 的元素全部出栈
        while(st.size() && st.top() >= a[i]) st.pop();
        st.push(a[i]);
    }
}

void test2()
{
    stack<int> st; // 维护一个单调递减的栈
    for(int i = 1; i <= n; i++)
    {
        // 栈里面小于等于 a[i] 的元素全部出栈
        while(st.size() && st.top() <= a[i]) st.pop();
        st.push(a[i]);
    }
}
```

2. 单调栈解决的问题

单调栈能帮助我们解决以下四个问题：

- 寻找当前元素左侧，离它最近，并且比它大的元素在哪；
- 寻找当前元素左侧，离它最近，并且比它小的元素在哪；
- 寻找当前元素右侧，离它最近，并且比它大的元素在哪；
- 寻找当前元素右侧，离它最近，并且比它小的元素在哪。

虽然是四个问题，但是原理是一致的。因此，只要解决一个，举一反三就可以解决剩下的几个。

3. 寻找当前元素左侧，离它最近，并且比它大的元素在哪

> 从左往右遍历元素，构造一个单调递减的栈。插入当前位置的元素的时：
>
> - 如果栈为空，则左侧不存在比当前元素大的元素；
> - 如果栈非空，插入当前位置元素时的栈顶元素就是所找的元素。
>
> 注意，因为我们要找的是最终结果的位置。因此，栈里面存的是每个元素的下标。

【测试用例】

```
输入：
9
1 4 10 6 3 3 15 21 8

输出：
0 0 0 3 4 4 0 0 8
```

【代码实现】

```cpp
#include <iostream>
#include <stack>

using namespace std;

const int N = 3e6 + 10;

int a[N], n;
int ret[N];

void test()
{
    stack<int> st; // 维护一个单调递减的栈
```

```
14
15      for(int i = 1; i <= n; i++)
16      {
17          // 栈里面小于等于 a[i] 的元素全部出栈
18          while(st.size() && a[st.top()] <= a[i]) st.pop();
19
20          // 此时栈顶元素存在，栈顶元素就是所求结果
21          if(st.size()) ret[i] = st.top();
22
23          st.push(i); // 存的是下标
24      }
25
26      for(int i = 1; i <= n; i++)
27      {
28          cout << ret[i] << " ";
29      }
30      cout << endl;
31  }
32
33  int main()
34  {
35      cin >> n;
36      for(int i = 1; i <= n; i++) cin >> a[i];
37
38      test(); cout << endl;
39
40      return 0;
41  }
```

4. 寻找当前元素左侧，离它最近，并且比它小的元素在哪

从左往右遍历元素，构造一个单调递增的栈。插入当前位置的元素的时：

- 如果栈为空，则左侧不存在比当前元素小的元素；
- 如果栈非空，插入当前位置元素时的栈顶元素就是所找的元素。

注意，因为我们要找的是最终结果的位置。因此，栈里面存的是每个元素的下标。

【测试用例】

```
1  输入：
2  9
3  1 4 10 6 3 3 15 21 8
```

```
4
5  输出:
6  0 1 2 2 1 1 6 7 6
```

【代码实现】

```cpp
#include <iostream>
#include <stack>

using namespace std;

const int N = 3e6 + 10;

int a[N], n;
int ret[N];

void test()
{
    stack<int> st; // 维护一个单调递增的栈

    for(int i = 1; i <= n; i++)
    {
        // 栈里面大于等于 a[i] 的元素全部出栈
        while(st.size() && a[st.top()] >= a[i]) st.pop();

        // 此时栈顶元素存在，栈顶元素就是所求结果
        if(st.size()) ret[i] = st.top();

        st.push(i); // 存的是下标
    }

    for(int i = 1; i <= n; i++)
    {
        cout << ret[i] << " ";
    }
    cout << endl;
}

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];

    test(); cout << endl;
```

```
39
40      return 0;
41  }
```

针对其余两种情况，我们仅需逆序遍历数组即可。

5. 寻找当前元素右侧，离它最近，并且比它大的元素在哪

> **从右往左**遍历元素，构造一个单调递减的栈。插入当前位置的元素的时：
>
> - 如果栈为空，则左侧不存在比当前元素大的元素；
> - 如果栈非空，插入当前位置元素时的栈顶元素就是所找的元素。
>
> 注意，因为我们要找的是最终结果的位置。因此，栈里面存的是每个元素的下标。

【测试用例】

```
1  输入：
2  9
3  1 4 10 6 3 3 15 21 8
4
5  输出：
6  2 3 7 7 7 7 8 0 0
```

【代码实现】

```cpp
#include <iostream>
#include <stack>

using namespace std;

const int N = 3e6 + 10;

int a[N], n;
int ret[N];

void test()
{
    stack<int> st; // 维护一个单调递减的栈

```

```
    for(int i = n; i >= 1; i--)
    {
        // 栈里面小于等于 a[i] 的元素全部出栈
        while(st.size() && a[st.top()] <= a[i]) st.pop();

        // 此时栈顶元素存在，栈顶元素就是所求结果
        if(st.size()) ret[i] = st.top();

        st.push(i); // 存的是下标
    }

    for(int i = 1; i <= n; i++)
    {
        cout << ret[i] << " ";
    }
    cout << endl;
}

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];

    test(); cout << endl;

    return 0;
}
```

6. 寻找当前元素右侧，离它最近，并且比它小的元素在哪

> **从右往左**遍历元素，构造一个单调递增的栈。插入当前位置的元素的时：
>
> - 如果栈为空，则左侧不存在比当前元素小的元素；
> - 如果栈非空，插入当前位置元素时的栈顶元素就是所找的元素。
>
> 注意，因为我们要找的是最终结果的位置。因此，栈里面存的是每个元素的下标。

【测试用例】

```
输入：
9
1 4 10 6 3 3 15 21 8

```

```
输出:
0 5 4 5 0 0 9 9 0
```

【代码实现】

```cpp
#include <iostream>
#include <stack>

using namespace std;

const int N = 3e6 + 10;

int a[N], n;
int ret[N];

void test()
{
    stack<int> st; // 维护一个单调递增的栈

    for(int i = n; i >= 1; i--)
    {
        // 栈里面大于等于 a[i] 的元素全部出栈
        while(st.size() && a[st.top()] >= a[i]) st.pop();

        // 此时栈顶元素存在，栈顶元素就是所求结果
        if(st.size()) ret[i] = st.top();

        st.push(i); // 存的是下标
    }

    for(int i = 1; i <= n; i++)
    {
        cout << ret[i] << " ";
    }
    cout << endl;
}

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];

    test(); cout << endl;

```

```
40        return 0;
41    }
```

💡 **总结：**

- 找左侧，正遍历；找右侧，逆遍历；
- 比它大，单调减；比它小，单调增。

1.1 【模板】单调栈

题目来源： 洛谷

题目链接： P5788 【模板】单调栈

难度系数： ★★

【题目描述】

给出项数为 n 的整数数列 $a_{1...n}$。

定义函数 $f(i)$ 代表数列中第 i 个元素之后第一个大于 a_i 的元素的下标，即 $f(i)=min_{i<j\leq n,aj>ai}j$。若不存在，则 $f(i)=0$。

试求出 $f(1...n)$。

【输入描述】

第一行一个正整数 n。

第二行 n 个正整数 $a_{1...n}$。

对于 100% 的数据，$1\leq n\leq 3\times 10^6, 1\leq a_i\leq 10^9$。

【输出描述】

一行 n 个整数表示 $f(1),f(2),...,f(n)$ 的值。

【示例一】

输入：

```
5
1 4 2 3 5
```

输出：

> 25450

【解法】

右侧离它最近并且比它大的元素：

- 逆序遍历数组；
- 构造一个单调递减的栈；
- 进栈时，栈顶元素就是最终结果。

【参考代码】

```cpp
#include <iostream>
#include <stack>

using namespace std;

const int N = 3e6 + 10;

int n;
int a[N];
int ret[N];

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];

    stack<int> st;
    for(int i = n; i >= 1; i--)
    {
        // 单调递减的栈 - 元素的下标
        while(st.size() && a[st.top()] <= a[i]) st.pop();

        if(st.size()) ret[i] = st.top();
        st.push(i);
    }

    for(int i = 1; i <= n; i++) cout << ret[i] << " ";
    cout << endl;

    return 0;
}
```

1.2 发射站

> **题目来源：** 洛谷
>
> **题目链接：** P1901 发射站
>
> **难度系数：** ★★

【题目描述】

> 某地有 N 个能量发射站排成一行，每个发射站 i 都有不相同的高度 H_i，并能向两边（两端的发射站只能向一边）同时发射能量值为 V_i 的能量，发出的能量只被两边最近的且比它高的发射站接收。显然，每个发射站发来的能量有可能被 0 或 1 或 2 个其他发射站所接受。
>
> 请计算出接收最多能量的发射站接收的能量是多少。

【输入描述】

> 第 1 行一个整数 N。
>
> 第 2 到 $N+1$ 行，第 $i+1$ 行有两个整数 H_i 和 V_i，表示第 i 个发射站的高度和发射的能量值。

【输出描述】

> 输出仅一行，表示接收最多能量的发射站接收到的能量值。答案不超过 32 位带符号整数的表示范围。

【示例一】

> 输入：
>
> 3
>
> 4 2
>
> 3 5
>
> 6 10
>
> 输出：
>
> 7

【解法】

有了单调栈之后，这道题就变成模拟题了.....

【参考代码】

```
1  #include <iostream>
2  #include <stack>
```

```cpp

using namespace std;

typedef long long LL;

const int N = 1e6 + 10;

int n;
LL h[N], v[N];
LL sum[N];

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> h[i] >> v[i];

    // 找左边
    stack<int> st;
    for(int i = 1; i <= n; i++)
    {
        // 单调递减的栈 - 存下标
        while(st.size() && h[st.top()] <= h[i]) st.pop();
        if(st.size())
        {
            sum[st.top()] += v[i];
        }
        st.push(i);
    }

    // 找右边
    while(st.size()) st.pop(); // 清空
    for(int i = n; i >= 1; i--)
    {
        // 单调递减的栈 - 存下标
        while(st.size() && h[st.top()] <= h[i]) st.pop();
        if(st.size())
        {
            sum[st.top()] += v[i];
        }
        st.push(i);
    }

    LL ret = 0;
    for(int i = 1; i <= n; i++) ret = max(ret, sum[i]);

    cout << ret << endl;

```

```
50        return 0;
51    }
```

1.3 Largest Rectangle in a Histogram

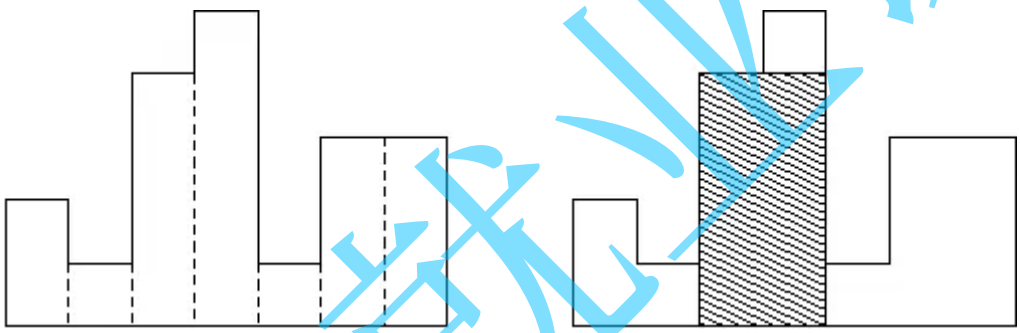
题目来源： 洛谷

题目链接： HISTOGRA - Largest Rectangle in a Histogram

难度系数： ★★★

【题目描述】

如图所示，在一条水平线上有 n 个宽为 1 的矩形，求包含于这些矩形的最大子矩形面积（图中的阴影部分的面积即所求答案）。



【输入描述】

有多组测试数据，每组数据占一行。输入零时读入结束。

每行开头为一个数字 $n(1 \leq n \leq 10^5)$，接下来在同一行给出 n 个数字 $h_1, h_2, \cdots, h_n (0 \leq h_i \leq 10^9)$，表示每个矩形的高度。

【输出描述】

对于每组数据，输出最大子矩阵面积，一组数据输出一行。

【示例一】

输入：

```
7 2 1 4 5 1 3 3
4 1000 1000 1000 1000
0
```

输出：

```
8
4000
```

【解法】

对于 x 位置子矩阵，找到左侧离它最近并且比它小的位置 y ，那么 $[x+1, y]$ 之间就是该矩阵能到达的左端。

同理再找到右侧离它最近并且比它小的位置 z ，那么 $[y, z-1]$ 之间就是该矩阵能到达的右端。

对于每一个子矩阵，求出它向左以及向右能延伸的最大长度即可。

【参考代码】

```cpp
#include <iostream>
#include <stack>

using namespace std;

typedef long long LL;

const int N = 1e5 + 10;

int n;
LL h[N];
LL x[N], y[N];

int main()
{
    while(cin >> n, n)
    {
        for(int i = 1; i <= n; i++) cin >> h[i];

        // 找左边，小
        stack<int> st;
        for(int i = 1; i <= n; i++)
        {
            // 单调递增的栈 - 存下标
            while(st.size() && h[st.top()] >= h[i]) st.pop();

            if(st.size()) x[i] = st.top();
            else x[i] = 0;

            st.push(i);
        }

        // 找右边，小
        while(st.size()) st.pop();

        for(int i = n; i >= 1; i--)
        {
```

```
38              // 单调递增的栈 - 存下标
39              while(st.size() && h[st.top()] >= h[i]) st.pop();
40
41              if(st.size()) y[i] = st.top();
42              else y[i] = n + 1;
43
44              st.push(i);
45          }
46
47          LL ret = 0;
48          for(int i = 1; i <= n; i++)
49          {
50              ret = max(ret, h[i] * (y[i] - x[i] - 1));
51          }
52
53          cout << ret << endl;
54      }
55
56      return 0;
57  }
```

2. 单调队列

1. 什么是单调队列?

单调队列，顾名思义，就是存储的元素要么单调递增要么单调递减的队列。注意，这里的队列和普通的队列不一样，是一个双端队列。

2. 单调队列解决的问题

一般用于解决滑动窗口内最大值最小值问题，以及优化动态规划。

2.1 【模板】单调队列

> **题目来源：** 洛谷
>
> **题目链接：** P1886 滑动窗口 /【模板】单调队列
>
> **难度系数：** ★★

【题目描述】

> 有一个长为 n 的序列 a ，以及一个大小为 k 的窗口。现在这个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。

例如，对于序列 $[1,3,-1,-3,5,3,6,7]$ 以及 $k=3$，有如下过程：

窗口位置	最小值	最大值
[1 3 -1] -3 5 3 6 7	-1	3
1 [3 -1 -3] 5 3 6 7	-3	3
1 3 [-1 -3 5] 3 6 7	-3	5
1 3 -1 [-3 5 3] 6 7	-3	5
1 3 -1 -3 [5 3 6] 7	3	6
1 3 -1 -3 5 [3 6 7]	3	7

【输入描述】

输入一共有两行，第一行有两个正整数 n,k。 第二行 n 个整数，表示序列 a

【输出描述】

输出共两行，第一行为每次窗口滑动的最小值

第二行为每次窗口滑动的最大值

【示例一】

输入：

```
8 3
1 3 -1 -3 5 3 6 7
```

输出：

```
-1 -3 -3 -3 3 3
3 3 5 5 6 7
```

【解法】

窗口内最大值：

从左往右遍历元素，维护一个单调递减的队列：

- 当前元素进队之后，注意维护队列内的元素在大小为 k 的窗口内；
- 此时队头元素就是最大值。

窗口内最小值：

从左往右遍历元素，维护一个单调递增的队列：

- 当前元素进队之后，注意维护队列内的元素在大小为 k 的窗口内；

- 此时队头元素就是最小值。

【参考代码】

```cpp
#include <iostream>
#include <deque>

using namespace std;

const int N = 1e6 + 10;

int n, k;
int a[N];

int main()
{
    cin >> n >> k;
    for(int i = 1; i <= n; i++) cin >> a[i];

    deque<int> q; // 存下标
    // 窗口内最小值 - 单调递增的队列 - 存下标
    for(int i = 1; i <= n; i++)
    {
        while(q.size() && a[q.back()] >= a[i]) q.pop_back();

        q.push_back(i);

        // 判断队列里面元素是否在合法窗口内
        if(q.back() - q.front() + 1 > k) q.pop_front();

        if(i >= k) cout << a[q.front()] << " ";
    }
    cout << endl;

    // 窗口内最大值 - 单调递减的队列 - 存下标
    q.clear();
    for(int i = 1; i <= n; i++)
    {
        while(q.size() && a[q.back()] <= a[i]) q.pop_back();

        q.push_back(i);

        if(q.back() - q.front() + 1 > k) q.pop_front();
        if(i >= k) cout << a[q.front()] << " ";
    }
```

```
42        cout << endl;
43
44        return 0;
45    }
```

2.2 质量检测

题目来源： 洛谷

题目链接： P2251 质量检测

难度系数： ★★

【题目描述】

为了检测生产流水线上总共 N 件产品的质量，我们首先给每一件产品打一个分数 A 表示其品质，然后统计前 M 件产品中质量最差的产品的分值 $Q[m]=min\{A_1,A_2,...A_m\}$，以及第 2 至第 $M+1$ 件的 $Q[m+1],Q[m+2]...$ 最后统计第 $N-M+1$ 至第 N 件的 $Q[n]$。根据 Q 再做进一步评估。

请你尽快求出 Q 序列。

【输入描述】

输入共两行。

第一行共两个数 N、M，由空格隔开。含义如前述。

第二行共 N 个数，表示 N 件产品的质量。

【输出描述】

输出共 $N-M+1$ 行。

第 1 至 $N-M+1$ 行每行一个数，第 i 行的数 $Q[i+M-1]$。含义如前述。

【示例一】

输入：

```
10 4
16 5 6 9 5 13 14 20 8 12
```

输出：

```
5
5
5
5
```

```
5

8

8
```

【解法】

滑动窗口内的最小值~

【参考代码】

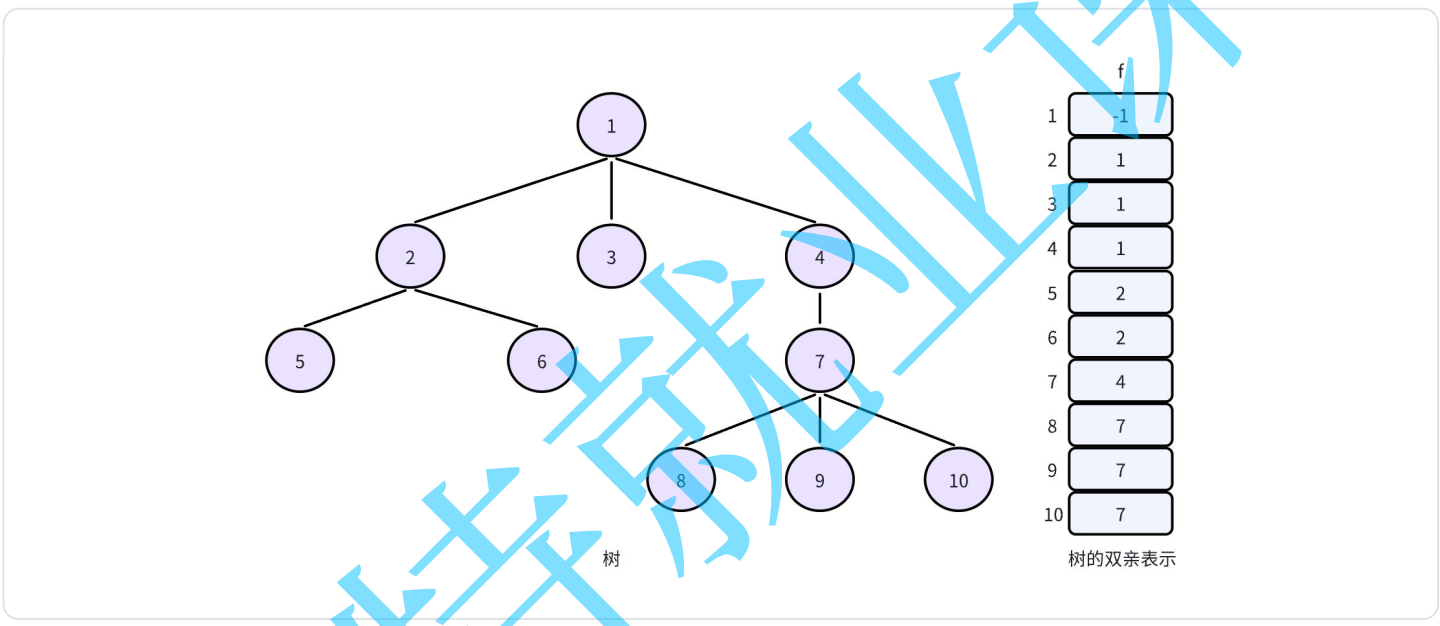
```cpp
#include <iostream>
#include <deque>

using namespace std;

const int N = 1e6 + 10;

int n, k;
int a[N];

int main()
{
    cin >> n >> k;

    // 窗口内最小值
    deque<int> q; // 单调递增
    for(int i = 1; i <= n; i++)
    {
        cin >> a[i];

        // 维护递增
        while(q.size() && a[q.back()] >= a[i]) q.pop_back();

        q.push_back(i);

        // 维护队列内元素合法
        if(i - q.front() + 1 > k) q.pop_front();

        // 输出结果
        if(i >= k) cout << a[q.front()] << endl;
    }

    return 0;
}
```

3. 并查集

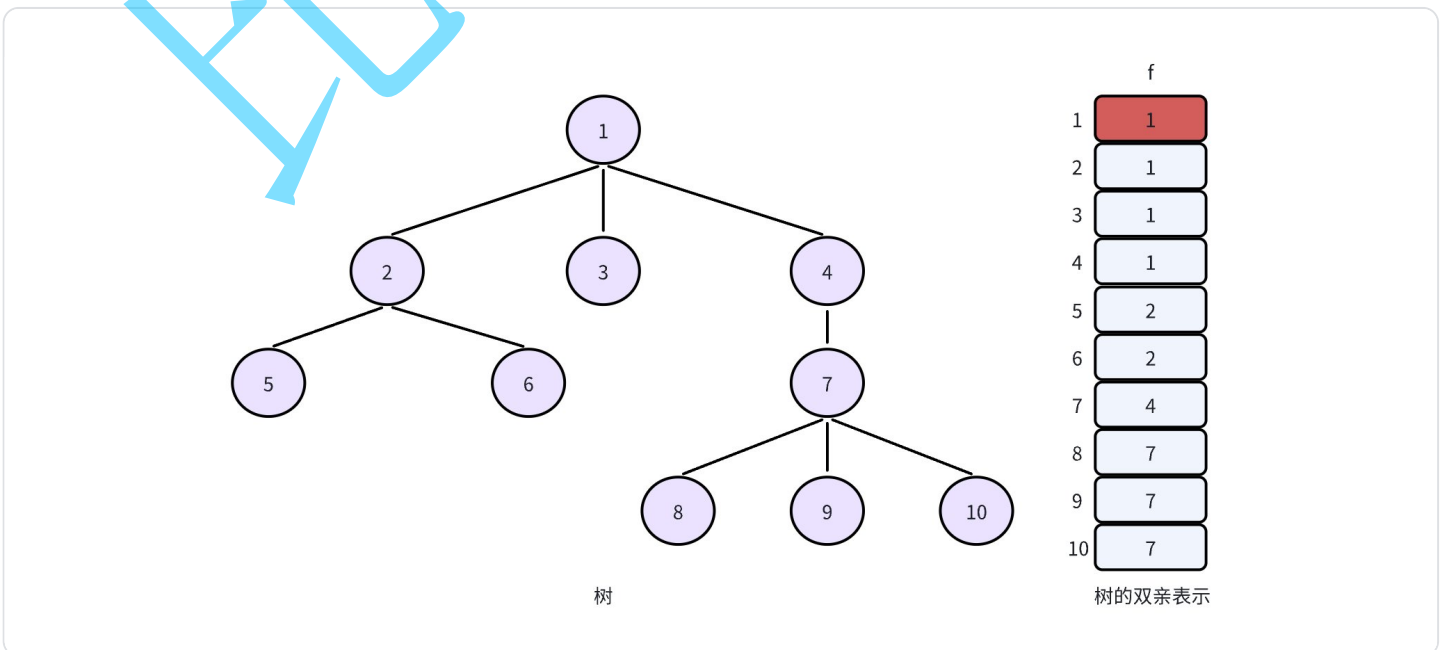
3.1 双亲表示法

接下来要学习到的并查集，本质上就是用双亲表示法实现的森林。因此，我们先认识一下双亲表示法。

在学习树这个数据结构的时，讲到树的存储方式有很多种：孩子表示法，双亲表示法、孩子双亲表示法以及孩子兄弟表示法等。对一棵树而言，除了根节点外，其余每个结点一定有且仅有一个双亲，双亲表示法就是根据这个特点存储树的，也就是把每个结点的双亲存下来。因此，我们可以采用数组来存储每个结点的父亲结点的编号，这就实现了双亲表示法(so easy)。



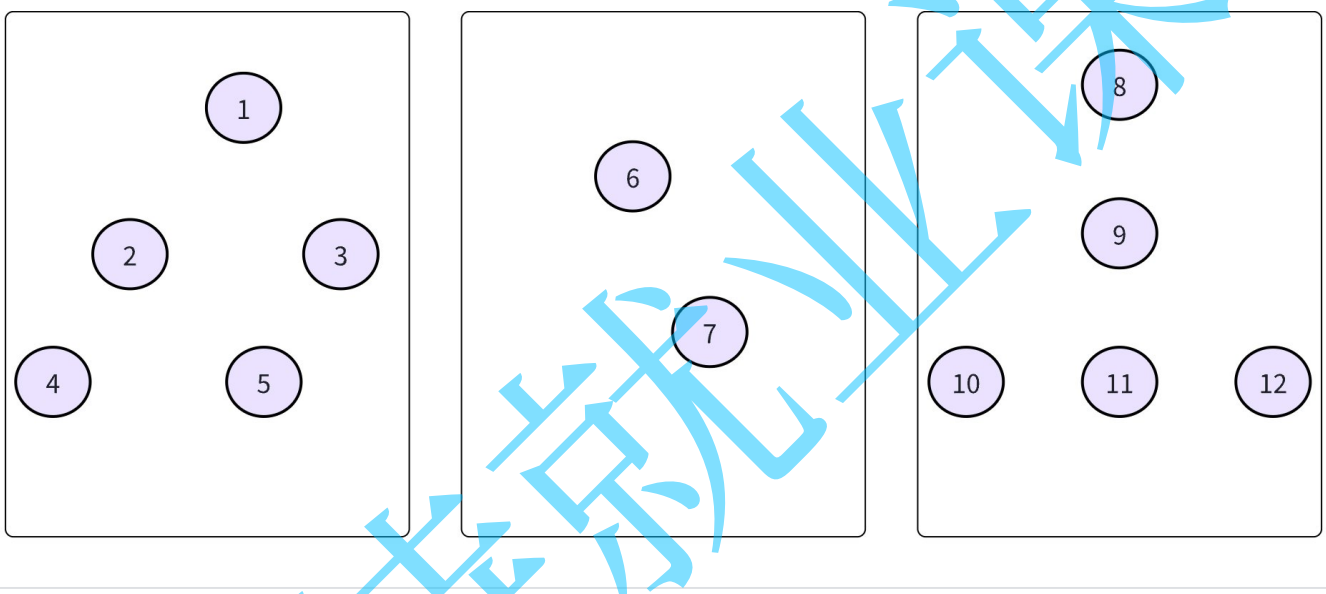
但是，在实现并查集的时，我们一般让**根节点自己指向自己**。因此，上述存储就变成：



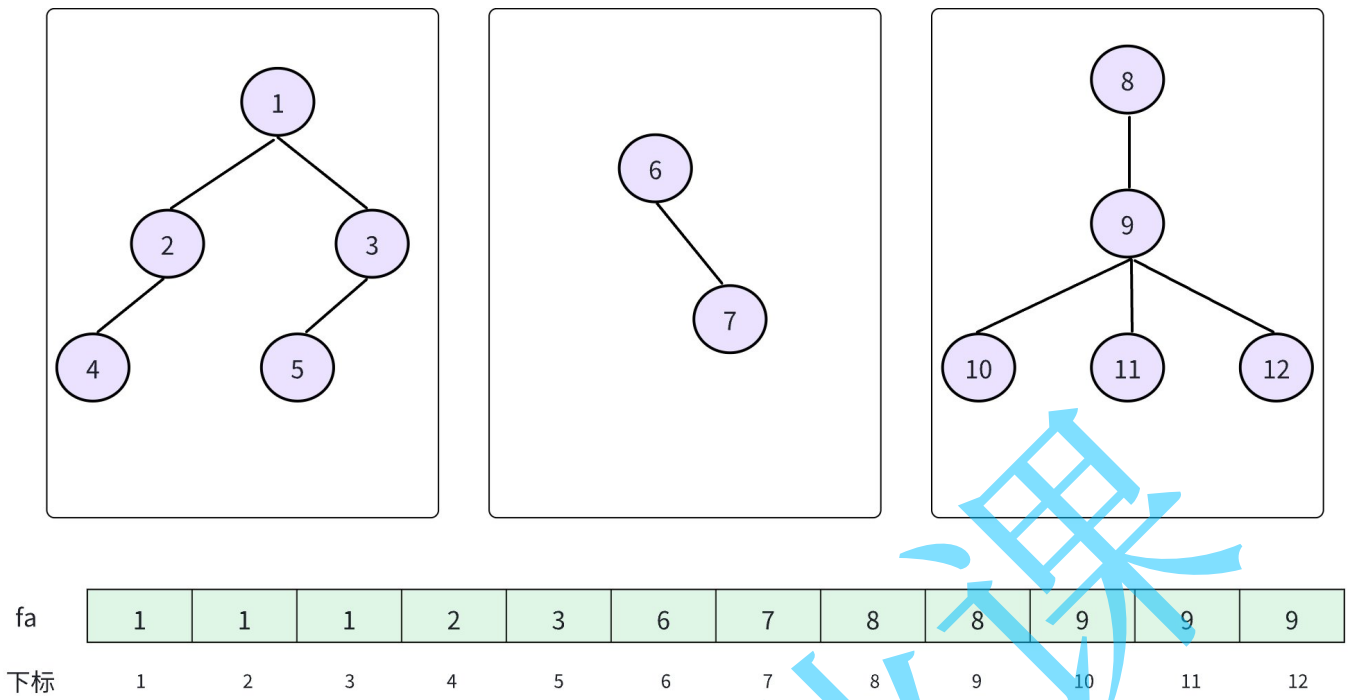
3.2 并查集的概念

在有些问题中，我们需要维护若干个集合，并且基于这些集合要频繁执行下面的操作：

- **查询操作**：查找元素 x 属于哪一个集合。一般会在每个集合中选取一个元素作为代表，查询的是这个集合中的代表元素；
- **合并操作**：将元素 x 所在的集合与元素 y 所在的集合合并成一个集合；（注意，合并的是元素所在的集合，不是这两个元素！）
- **判断操作**：判断元素 x 和 y 是否在同一个集合。



并查集（Union Find）：是一种用于维护元素所属集合的数据结构，实现为一个森林，其中每棵树表示一个集合，树中的节点表示对应集合中的元素，根节点来代表整个集合。



3.3 并查集的实现

3.3.1 初始化

> 初始状态下，所有的元素单独成为一个集合：
>
> - 让元素自己指向自己即可。

代码实现：

```
const int N = 1e6 + 10;
int n;
int fa[N]; // 双亲表示法所需的数组

// 初始化并查集
void init()
{
    for(int i = 1; i <= n; i++) fa[i] = i;
}
```

3.3.2 查询操作

> 查询操作是并查集的核心操作，其余所有的操作都是基于查询操作实现的！
>
> 找到元素 x 所属的集合：
>
> - 一直向上找爸爸~

代码实现：

```
// 查询操作
int find(int x)
{
    if(fa[x] == x) return x;
    return find(fa[x]);

    // 一行实现
    return fa[x] == x ? x : find(fa[x]);
}
```

3.3.3 合并操作

> 将元素 x 所在的集合与元素 y 所在的集合合并成一个集合：
>
> - 让元素 x 所在树的根节点指向元素 y 所在树的根节点。
>
> （反过来也是可以的）

代码实现：

```
// 合并操作
void un(int x, int y) // 注意，函数名字不能用 union，因为它是 C++ 的关键字
{
    int fx = find(x);
    int fy = find(y);
    fa[fx] = fy;
}
```

3.3.4 判断操作

> 判断元素 x 和元素 y 是否在同一集合：

- 看看两者所在树的根节点是否相同。

代码实现：

```
// 判断是否在同一集合
bool issame(int x, int y)
{
    return find(x) == find(y);
}
```

3.4 并查集的优化

极端情况：在合并的过程中，整棵树变成一个链表。

路径压缩：在查询时，把被查询的节点到根节点的路径上的所有节点的父节点设置为根节点，从而减小树的深度。也就是说，在向上查询的同时，把在路径上的每个节点都直接连接到根上，以后查询时就能直接查询到根节点。

代码实现：

```
// 找根节点 - 路径压缩
int find(int x)
{
    if(fa[x] == x) return x;
    return fa[x] = find(fa[x]);

    // 一行实现
    return fa[x] == x ? x : fa[x] = find(fa[x]);
}
```

还有一种优化方式是按秩合并，但是基本上不用按秩合并，并查集的时间复杂度就很优秀了。感兴趣的同学可以搜一下按秩合并，按照大家现在的水平，应该很容易就能看懂~

在《算法导论》中有严格的证明，并查集查询根节点的最坏时间复杂度为 $O(\alpha(n))$，是一个很小的常数。因此，并查集查询以及合并的效率近似可以看成 $O(1)$。

3.5 普通并查集

3.5.1 【模板】并查集

> **题目来源：** 洛谷
>
> **题目链接：** P3367 【模板】并查集
>
> **难度系数：** ★★

【题目描述】

> 如题，现在有一个并查集，你需要完成合并和查询操作。

【输入描述】

> 第一行包含两个整数 N, M，表示共有 N 个元素和 M 个操作。
>
> 接下来 M 行，每行包含三个整数 Z_i, X_i, Y_i。
>
> 当 $Z_i = 1$ 时，将 X_i 与 Y_i 所在的集合合并。
>
> 当 $Z_i = 2$ 时，输出 X_i 与 Y_i 是否在同一集合内，是的输出 `Y`；否则输出 `N`。

【输出描述】

> 对于每一个 $Z_i = 2$ 的操作，都有一行输出，每行包含一个大写字母，为 `Y` 或者 `N`。

【示例一】

> 输入：
>
> ```
> 4 7
> 2 1 2
> 1 1 2
> 2 1 2
> 1 3 4
> 2 1 4
> 1 2 3
> 2 1 4
> ```
>
> 输出：
>
> ```
> N
> Y
> N
> Y
> ```

【解法】

模板题~

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 2e5 + 10;

int n;
int fa[N];

int find(int x)
{
    if(fa[x] == x) return x;
    return fa[x] = find(fa[x]);
}

int main()
{
    int T;
    cin >> n >> T;

    // 初始化
    for(int i = 1; i <= n; i++) fa[i] = i;

    while(T--)
    {
        int z, x, y; cin >> z >> x >> y;
        if(z == 1) // 合并
        {
            int fx = find(x);
            int fy = find(y);
            fa[fx] = fy;
        }
        else // 判断
        {
            if(find(x) == find(y)) cout << "Y" << endl;
            else cout << "N" << endl;
        }
    }

    return 0;
}
```

3.5.2 亲戚

> **题目来源：** 洛谷
>
> **题目链接：** P1551 亲戚
>
> **难度系数：** ★★

【题目描述】

> 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。

【输入描述】

> 第一行：三个整数 n,m,p，$(n,m,p \leq 5000)$，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 $M_i,M_j,1 \leq M_i,M_j \leq n$，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。

【输出描述】

> 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。

【示例一】

> 输入：
>
> ```
> 6 5 3
> 1 2
> 1 5
> 3 4
> 5 2
> 1 3
> 1 4
> 2 3
> 5 6
> ```
>
> 输出：
>
> ```
> Yes
> Yes
> No
> ```

【解法】

具有亲戚关系的两个集合就合并在一个集合中。因此，可以用并查集解决。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 5010;

int n, m, p;
int fa[N];

int find(int x)
{
    return fa[x] == x ? x : fa[x] = find(fa[x]);
}

void un(int x, int y)
{
    int fx = find(x), fy = find(y);
    fa[fy] = fx;
}

bool issame(int x, int y)
{
    return find(x) == find(y);
}

int main()
{
    cin >> n >> m >> p;

    // 初始化
    for(int i = 1; i <= n; i++) fa[i] = i;

    while(m--)
    {
        int x, y; cin >> x >> y;
        un(x, y);
    }

    while(p--)
```

```
    {
        int x, y; cin >> x >> y;
        if(issame(x, y)) cout << "Yes\n";
        else cout << "No\n";
    }

    return 0;
}
```

3.5.3 Lake Counting

> **题目来源：** 洛谷
>
> **题目链接：** P1596 [USACO10OCT] Lake Counting S
>
> **难度系数：** ★★

【题目描述】

> 由于近期的降雨，雨水汇集在农民约翰的田地不同的地方。我们用一个 $N \times M (1 \leq N, M \leq 100)$ 的网格图表示。每个网格中有水（`W`） 或是旱地（`.`）。一个网格与其周围的八个网格相连，而一组相连的网格视为一个水坑。约翰想弄清楚他的田地已经形成了多少水坑。给出约翰田地的示意图，确定当中有多少水坑。

【输入描述】

> 输入第 1 行：两个空格隔开的整数：N 和 M。
>
> 第 2 行到第 $N+1$ 行：每行 M 个字符，每个字符是 `W` 或 `.`，它们表示网格图中的一排。字符之间没有空格。

【输出描述】

> 输出一行，表示水坑的数量。

【示例一】

> 输入：
>
> 10 12
>
> W........WW.
>
> .WWW.....WWW
>
> ....WW...WW.
>
> ....WW.
>
> ....W..
>
> ..W......W..

> .W.W.....WW.
>
> W.W.W.....W.
>
> .W.W.....W.
>
> ..W.....W.
>
> 输出:
>
> 3

【解法】

遍历整个矩阵，每次遇到一个水坑时，就把这个水坑右、下，左下以及右下的水坑合并在一起。

最终判断一下一共有多少个集合。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 110;

int n, m;
char a[N][N];
int fa[N * N];

int dx[] = {0, 1, 1, 1};
int dy[] = {1, 1, 0, -1};

int find(int x)
{
    return fa[x] == x ? x : fa[x] = find(fa[x]);
}

void un(int x, int y)
{
    fa[find(x)] = find(y);
}

int main()
{
    cin >> n >> m;

    for(int i = 0; i < n; i++)
        for(int j = 0; j < m; j++)
```

```
            cin >> a[i][j];

    // 初始化
    for(int i = 0; i < n * m; i++) fa[i] = i;

    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < m; j++)
        {
            if(a[i][j] == '.') continue;

            for(int k = 0; k < 4; k++)
            {
                int x = i + dx[k], y = j + dy[k];
                if(y >= 0 && a[x][y] == 'W')
                {
                    un(i * m + j, x * m + y);
                }
            }
        }
    }

    int ret = 0;
    for(int i = 0; i < n * m; i++)
    {
        // 一维转二维
        int x = i / m, y = i % m;
        if(a[x][y] == 'W' && fa[i] == i) ret++;
    }

    cout << ret << endl;

    return 0;
}
```


3.5.4 程序自动分析

> **题目来源：** 洛谷
>
> **题目链接：** P1955 [NOI2015] 程序自动分析
>
> **难度系数：** ★★★

【题目描述】

> 在实现程序自动分析的过程中，常常需要判定一些约束条件是否能被同时满足。

考虑一个约束满足问题的简化版本：假设 $x_1, x_2, x_3, \cdots$ 代表程序中出现的变量，给定 n 个形如 $x_i = x_j$ 或 $x_i \neq x_j$ 的变量相等/不等的约束条件，请判定是否可以分别为每一个变量赋予恰当的值，使得上述所有约束条件同时被满足。例如，一个问题中的约束条件为：$x_1 = x_2, x_2 = x_3, x_3 = x_4, x_4 \neq x_1$，这些约束条件显然是不可能同时被满足的，因此这个问题应判定为不可被满足。

现在给出一些约束满足问题，请分别对它们进行判定。

【输入描述】

输入的第一行包含一个正整数 t ，表示需要判定的问题个数。注意这些问题之间是相互独立的。

对于每个问题，包含若干行：

第一行包含一个正整数 n ，表示该问题中需要被满足的约束条件个数。接下来 n 行，每行包括三个整数 i, j, e，描述一个相等/不等的约束条件，相邻整数之间用单个空格隔开。若 $e = 1$，则该约束条件为 $x_i = x_j$。若 $e = 0$，则该约束条件为 $x_i \neq x_j$。

【输出描述】

输出包括 t 行。

输出文件的第 k 行输出一个字符串 `YES` 或者 `NO`（字母全部大写），`YES` 表示输入中的第 k 个问题判定为可以被满足，`NO` 表示不可被满足。

【示例一】

输入：

```
2
2
1 2 1
1 2 0
2
1 2 1
2 1 1
```

输出：

```
NO
YES
```

【解法】

先利用并查集维护所有相等的信息，然后遍历所有的不相等信息，判断一下是否合法。

因为数据范围的问题，需要先对所有的数离散化处理。

【参考代码】

```cpp
#include <iostream>
#include <unordered_map>
#include <algorithm>

using namespace std;

const int N = 1e5 + 10;

int n;
struct node
{
    int x, y, e;
}a[N];

// 离散化
int pos;
int disc[N * 2];
unordered_map<int, int> mp;

// 并查集
int fa[N * 2];

int find(int x)
{
    return fa[x] == x ? x : fa[x] = find(fa[x]);
}

void un(int x, int y)
{
    fa[find(x)] = find(y);
}

bool issame(int x, int y)
{
    return find(x) == find(y);
}

bool solve()
{
    cin >> n;
    // 清空数据
    pos = 0;
    mp.clear();

```

```
45      for(int i = 1; i <= n; i++)
46      {
47          cin >> a[i].x >> a[i].y >> a[i].e;
48          disc[++pos] = a[i].x; disc[++pos] = a[i].y;
49      }
50
51      // 离散化
52      sort(disc + 1, disc + 1 + pos);
53      int cnt = 0;
54      for(int i = 1; i <= pos; i++)
55      {
56          int x = disc[i];
57          if(mp.count(x)) continue;
58
59          cnt++;
60          mp[x] = cnt;
61      }
62
63      // 初始化
64      for(int i = 1; i <= cnt; i++) fa[i] = i;
65
66      // 1. 把所有相等的信息，用并查集维护起来
67      for(int i = 1; i <= n; i++)
68      {
69          int x = a[i].x, y = a[i].y, e = a[i].e;
70          if(e == 1) un(mp[x], mp[y]);
71      }
72
73      // 2. 拿出不等的信息，判断是否合法
74      for(int i = 1; i <= n; i++)
75      {
76          int x = a[i].x, y = a[i].y, e = a[i].e;
77          if(e == 0)
78          {
79              if(issame(mp[x], mp[y])) return false;
80          }
81      }
82
83      return true;
84  }
85
86  int main()
87  {
88      int T; cin >> T;
89
90      while(T--)
91      {
```

```
92            if(solve()) cout << "YES" << endl;
93            else cout << "NO" << endl;
94        }
95
96        return 0;
97    }
```

3.6 扩展域并查集

普通的并查集只能解决各元素之间仅存在一种相互关系，比如《亲戚》题目中：

- a 和 b 是亲戚关系，b 和 c 是亲戚关系，这时就可以查找出 a 和 c 也存在亲戚关系。

但如果存在各元素之间存在多种相互关系，普通并查集就无法解决。比如下面的案例：

- a 和 b 是敌人关系，b 和 c 是敌人关系，但是 a 和 c 其实不是敌人关系，而是另一种朋友关系。

此时，就不仅仅是简单的敌人关系，还是出现一种朋友关系。

解决这类问题就需要对并查集进行扩展：将每个元素拆分成多个域，每个域代表一种状态或者关系。通过维护这些域之间的关系，来处理复杂的约束条件。

敌人朋友问题中，我们会将 x 分成两个域，朋友域 x 以及敌人域 y：

- x 和 y 是朋友，正常处理，把 x 和 y 合并成一个集合；
- x 和 y 是敌人：那么 x 和 y 的敌人 $y+n$ 就是朋友，合并 x 与 $y+n$；y 和 x 的敌人 $x+n$ 就是朋友，合并 y 与 $x+n$。

这样就可以利用两个域，将所有的关系维护起来。

3.6.1 团伙

> **题目来源：** 洛谷
>
> **题目链接：** P1892 [BOI2003] 团伙
>
> **难度系数：** ★★★

【题目描述】

> 现在有 n 个人，他们之间有两种关系：朋友和敌人。我们知道：
>
> - 一个人的朋友的朋友是朋友

- 一个人的敌人的敌人是朋友

现在要对这些人进行组团。两个人在一个团体内当且仅当这两个人是朋友。请求出这些人中最多可能有的团体数。

【输入描述】

第一行输入一个整数 n 代表人数。

第二行输入一个整数 m 表示接下来要列出 m 个关系。

接下来 m 行，每行一个字符 opt 和两个整数 p, q，分别代表关系（朋友或敌人），有关系的两个人之中的第一个人和第二个人。其中 opt 有两种可能：

- 如果 opt 为 `F`，则表明 p 和 q 是朋友。
- 如果 opt 为 `E`，则表明 p 和 q 是敌人。

【输出描述】

一行一个整数代表最多的团体数。

【示例一】

输入：

```
6
4
E 1 4
F 3 5
F 4 6
E 1 2
```

输出：

```
3
```

【解法】

扩展域并查集模板题：

- a 和 b 如果是朋友，那就直接合并在一起；
- a 和 b 如果是敌人，那就把 a 和 $b+n$ 以及 $a+n$ 和 b 合并在一起。

【参考代码】

```cpp
#include <iostream>

using namespace std;
```

```cpp

const int N = 1010;

int n, m;
int fa[N * 2]; // 扩展域并查集

int find(int x)
{
    return fa[x] == x ? x : fa[x] = find(fa[x]);
}

// 一定要让朋友域作为父结点
void un(int x, int y)
{
    fa[find(y)] = find(x);
}

int main()
{
    cin >> n >> m;

    // 初始化
    for(int i = 1; i <= n * 2; i++) fa[i] = i;

    while(m--)
    {
        char op; int x, y;
        cin >> op >> x >> y;
        if(op == 'F')
        {
            un(x, y);
        }
        else // 敌人
        {
            un(x, y + n);
            un(y, x + n);
        }
    }

    int ret = 0;
    for(int i = 1; i <= n; i++)
    {
        if(fa[i] == i) ret++;
    }

    cout << ret << endl;

```

```
51        return 0;
52    }
```

3.6.2 食物链

> **题目来源：** 洛谷
>
> **题目链接：** P2024 [NOI2001] 食物链
>
> **难度系数：** ★★★

【题目描述】

> 动物王国中有三类动物 A,B,C，这三类动物的食物链构成了有趣的环形。A 吃 B，B 吃 C，C 吃 A。
>
> 现有 N 个动物，以 $1\sim N$ 编号。每个动物都是 A,B,C 中的一种，但是我们并不知道它到底是哪一种。
>
> 有人用两种说法对这 N 个动物所构成的食物链关系进行描述：
>
> - 第一种说法是 `1 X Y`，表示 X 和 Y 是同类。
> - 第二种说法是 `2 X Y`，表示 X 吃 Y。
>
> 此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。
>
> - 当前的话与前面的某些真的话冲突，就是假话；
> - 当前的话中 X 或 Y 比 N 大，就是假话；
> - 当前的话表示 X 吃 X，就是假话。
>
> 你的任务是根据给定的 N 和 K 句话，输出假话的总数。

【输入描述】

> 第一行两个整数，N,K，表示有 N 个动物，K 句话。
>
> 第二行开始每行一句话（按照题目要求，见样例）

【输出描述】

> 一行，一个整数，表示假话的总数。

【示例一】

> 输入：
>
> 100 7
>
> 1 101 1

```
2 1 2
2 2 3
2 3 3
1 1 3
2 3 1
1 5 5
```

输出：

```
3
```

【解法】

针对 x ，扩展三个域：同类域 x，捕食域 x + n，被捕食域 x + n + n。

如果 x 和 y 是同类：

- x 和 y 是同类
- x + n 与 y + n 是同类
- x + n + n 与 y + n + n 是同类

如果 x 捕食 y：

- x + n 与 y 同类
- x 与 y + n + n 同类
- x + n + n 与 y + n 同类

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 5e4 + 10;

int n, k;
int fa[N * 3]; // 扩展域并查集

int find(int x)
{
    return x == fa[x] ? x : fa[x] = find(fa[x]);
```

```cpp
}

void un(int x, int y)
{
    fa[find(x)] = find(y);
}

int main()
{
    cin >> n >> k;

    // 初始化
    for(int i = 1; i <= n * 3; i++) fa[i] = i;

    int ret = 0;
    while(k--)
    {
        int op, x, y; cin >> op >> x >> y;
        if(x > n || y > n) ret++;
        else if(op == 1) // 同类
        {
            // y -> x 以及 x -> y
            if(find(x) == find(y + n) || find(x) == find(y + n + n)) ret++;
            else
            {
                // 维护这个关系
                un(x, y);
                un(x + n, y + n);
                un(x + n + n, y + n + n);
            }
        }
        else // x -> y
        {
            // x 和 y 是同类，y -> x
            if(find(x) == find(y) || find(x) == find(y + n)) ret++;
            else
            {
                // 维护这个关系
                un(x, y + n + n);
                un(y, x + n);
                un(x + n + n, y + n);
            }
        }
    }

    cout << ret << endl;

```

```
60        return 0;
61    }
```

3.7 带权并查集

1. 带权并查集的概念

带权并查集在普通并查集的基础上，为每个结点增加了一个权值。这个权值可以表示当前结点与父结点之间的关系、距离或其他信息（注意，由于我们有路径压缩操作，所以最终这个权值表示的是当前结点相对于根结点的信息）。有了这样一个权值，就可以推断出集合中各个元素之间的相互关系。

2. 带权并查集的实现

我们以最简单的距离问题为例，实现一个能够查询任意两点之间距离的并查集。

实现带权并查集的核心是在进行 `Find` 和 `Union` 操作时，不仅要维护集合的结构，还要维护结点的权值。

注意：带权并查集的实现是多种多样的，基本上换一道题，实现的代码就要更改。因此一定要重点关注实现过程的思考方式，这才是通用的。

初始化 `init`：

```
1   const int N = 1e5 + 10, INF = 0x3f3f3f3f;
2   int n;
3   int fa[N], d[N];
4
5   void init()
6   {
7       for(int i = 1; i <= n; i++)
8       {
9           fa[i] = i;
10          d[i] = 0; // 根据题目要求来初始化
11      }
12  }
```

查询根节点操作 `find`：

```cpp
int find(int x)
{
    if(fa[x] == x) return x;

    int t = find(fa[x]); // 这句代码一定要先执行，先让父结点挂在根节点的后面
    d[x] += d[fa[x]]; // 注意，可能会根据权值的意义有所改变

    return fa[x] = t;
}
```

合并操作 `union`：

```cpp
// x 所在集合与 y 所在集合合并，x 与 y 之间的权值是 w
void un(int x, int y, int w)
{
    int fx = find(x), fy = find(y);
    if(fx != fy) // 不在同一个集合中
    {
        fa[fx] = fy;
        d[fx] = d[y] + w - d[x]; // 注意，可能会根据权值的意义有所改变
    }
}
```

查询距离操作 `query`：

```cpp
// 查询 x 到 y 的距离
int query(int x, int y)
{
    int fx = find(x), fy = find(y);

    if(fx != fy) return INF; // 如果不在同一个集合中，说明距离未知
    return d[y] - d[x];
}
```


3.7.1 食物链

> **题目来源： 洛谷**

题目链接： P2024 [NOI2001] 食物链

难度系数： ★★★

【题目描述】

动物王国中有三类动物 A,B,C，这三类动物的食物链构成了有趣的环形。A 吃 B，B 吃 C，C 吃 A。

现有 N 个动物，以 $1\sim N$ 编号。每个动物都是 A,B,C 中的一种，但是我们并不知道它到底是哪一种。

有人用两种说法对这 N 个动物所构成的食物链关系进行描述：

- 第一种说法是 `1 X Y`，表示 X 和 Y 是同类。
- 第二种说法是 `2 X Y`，表示 X 吃 Y。

此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。

- 当前的话与前面的某些真的话冲突，就是假话；
- 当前的话中 X 或 Y 比 N 大，就是假话；
- 当前的话表示 X 吃 X，就是假话。

你的任务是根据给定的 N 和 K 句话，输出假话的总数。

【输入描述】

第一行两个整数，N,K，表示有 N 个动物，K 句话。

第二行开始每行一句话（按照题目要求，见样例）

【输出描述】

一行，一个整数，表示假话的总数。

【示例一】

输入：

```
100 7
1 101 1
2 1 2
2 2 3
2 3 3
1 1 3
2 3 1
1 5 5
```

> 输出：
>
> 3

【解法】

把真话里面的相互关系，用"带权并查集"维护起来，权值表示当前节点相对于根节点的距离。那么对于集合中的任意两点 x 和 y ：

- 如果 $(d[y] - d[x]) \ \% \ 3 == 0$ ，表示两者是同类关系；
- 如果 $(d[y] - d[x]) \ \% \ 3 == 1$ ，表示两者是捕食关系；
- 如果 $(d[y] - d[x]) \ \% \ 3 == 2$ ，表示两者是天敌关系。

`find` 操作：

- 更新 d 数组：按照最基础的距离更新的方式， `d[x] = d[x] + d[fa[x]]` ；

`union` 操作：

- 如果 x 和 y 是同类，那么边权就是 0 ；
- 如果 x 吃 y ，那么边权就是 1 ；

【参考代码】

```
#include <iostream>

using namespace std;

const int N = 5e4 + 10;

int n, k;
int fa[N], d[N]; // 带权并查集

int find(int x)
{
    if(fa[x] == x) return x;

    int t = find(fa[x]);
    d[x] += d[fa[x]];
    return fa[x] = t;
}

void un(int x, int y, int w)
{
```

```
21      int fx = find(x), fy = find(y);
22      if(fx != fy)
23      {
24          fa[fx] = fy;
25          d[fx] = d[y] + w - d[x];
26      }
27  }
28
29  int main()
30  {
31      cin >> n >> k;
32
33      for(int i = 1; i <= n; i++) fa[i] = i;
34
35      int ret = 0;
36      while(k--)
37      {
38          int op, x, y; cin >> op >> x >> y;
39          int fx = find(x), fy = find(y);
40
41          if(x > n || y > n) ret++;
42          else if(op == 1) // 同类
43          {
44              if(fx == fy && ((d[y] - d[x]) % 3 + 3) % 3 != 0) ret++;
45              else un(x, y, 0);
46          }
47          else // x -> y
48          {
49              if(fx == fy && ((d[y] - d[x]) % 3 + 3) % 3 != 1) ret++;
50              else un(x, y, 2);
51          }
52      }
53
54      cout << ret << endl;
55
56      return 0;
57  }
```


3.7.2 银河英雄传说

> **题目来源：** 洛谷
>
> **题目链接：** P1196 [NOI2002] 银河英雄传说
>
> **难度系数：** ★★★

【题目描述】

杨威利擅长排兵布阵，巧妙运用各种战术屡次以少胜多，难免恣生骄气。在这次决战中，他将巴米利恩星域战场划分成 30000 列，每列依次编号为 1,2,…,30000。之后，他把自己的战舰也依次编号为 1,2,…,30000，让第 i 号战舰处于第 i 列，形成“一字长蛇阵”，诱敌深入。这是初始阵形。当进犯之敌到达时，杨威利会多次发布合并指令，将大部分战舰集中在某几列上，实施密集攻击。合并指令为 `M i j`，含义为第 i号战舰所在的整个战舰队列，作为一个整体（头在前尾在后）接至第 j 号战舰所在的战舰队列的尾部。显然战舰队列是由处于同一列的一个或多个战舰组成的。合并指令的执行结果会使队列增大。

然而，老谋深算的莱因哈特早已在战略上取得了主动。在交战中，他可以通过庞大的情报网络随时监听杨威利的舰队调动指令。

在杨威利发布指令调动舰队的同时，莱因哈特为了及时了解当前杨威利的战舰分布情况，也会发出一些询问指令：`C i j`。该指令意思是，询问电脑，杨威利的第 i 号战舰与第 j 号战舰当前是否在同一列中，如果在同一列中，那么它们之间布置有多少战舰。

作为一个资深的高级程序设计员，你被要求编写程序分析杨威利的指令，以及回答莱因哈特的询问。

【输入描述】

第一行有一个整数 T（1≤T≤5×10^5），表示总共有 T 条指令。

以下有 T 行，每行有一条指令。指令有两种格式：

1. `M i j`：i 和 j 是两个整数（1≤i,j≤30000），表示指令涉及的战舰编号。该指令是莱因哈特窃听到的杨威利发布的舰队调动指令，并且保证第 i 号战舰与第 j 号战舰不在同一列。
2. `C i j`：i 和 j 是两个整数（1≤i,j≤30000），表示指令涉及的战舰编号。该指令是莱因哈特发布的询问指令。

【输出描述】

依次对输入的每一条指令进行分析和处理：

- 如果是杨威利发布的舰队调动指令，则表示舰队排列发生了变化，你的程序要注意到这一点，但是不要输出任何信息。
- 如果是莱因哈特发布的询问指令，你的程序要输出一行，仅包含一个整数，表示在同一列上，第 i 号战舰与第 j 号战舰之间布置的战舰数目。如果第 i 号战舰与第 j 号战舰当前不在同一列上，则输出 −1。

【示例一】

输入：

```
4
M 2 3
C 1 2
M 2 4
```

> C 4 2
>
> 输出：
>
> -1
>
> 1

【解法】

这道题中有明显的边权关系，因此可以用"带权并查集"解决。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 3e4 + 10;

int n = 3e4;
int fa[N], d[N], cnt[N]; // 维护集合的合并、维护权值、维护集合的大小

int find(int x)
{
    if(fa[x] == x) return x;

    int t = find(fa[x]);
    d[x] += d[fa[x]];
    return fa[x] = t;
}

void un(int x, int y)
{
    int fx = find(x), fy = find(y);

    if(fx != fy)
    {
        fa[fx] = fy;
        d[fx] = cnt[fy];
        cnt[fy] += cnt[fx];
    }
}

int query(int x, int y)
{
    int fx = find(x), fy = find(y);
```

```
34
35      if(fx != fy) return -1;
36      else return abs(d[y] - d[x]) - 1;
37  }
38
39  int main()
40  {
41      // 初始化
42      for(int i = 1; i <= n; i++)
43      {
44          fa[i] = i;
45          cnt[i] = 1;
46      }
47
48      int T; cin >> T;
49
50      while(T--)
51      {
52          char op; int x, y;
53          cin >> op >> x >> y;
54          if(op == 'M') // 合并
55          {
56              un(x, y);
57          }
58          else // 查询
59          {
60              cout << query(x, y) << endl;
61          }
62      }
63
64      return 0;
65  }
```


4. 字符串哈希

1. 回忆：哈希函数与哈希冲突

- **哈希函数：**将关键字映射成对应的地址的函数，记为 `Hash(key) = Addr`。
- **哈希冲突：**哈希函数可能会把两个或两个以上的不同关键字映射到同一地址，这种情况称为哈希冲突。

2. 字符串哈希

定义一个把字符串映射到整数的函数 $hash$，这就是字符串哈希。说白了，就是将一个字符串用一个整数表示。

3. 字符串哈希中的哈希函数

在字符串哈希中，有一种冲突概率较小的哈希函数，将字符串映射成 p 进制数字：

$$hash(s) = \sum_{i=0}^{n-1} s[i] \times p^{n-i-1} \ (\mod M)$$

其中，p 通常取质数 131 或者 13331。如果把哈希值定义为 `unsigned long long` 类型，在 C++ 中，溢出就会自动取模。

（你没有看错，字符串哈希就是背一个公式即可……）

但是，实际求哈希值时，我们用的是前缀哈希的思想来求，这样会和下面的多次询问子串哈希一致。

4. 前缀哈希数组

单次计算一个字符串的哈希值复杂度是 $O(N)$。如果需要多次询问一个字符串的子串的哈希值，每次重新计算效率非常低下。

一般利用前缀和思想先预处理字符串中每个前缀的哈希值，这样的话每次就能快速求出子串的哈希了。

```cpp
typedef unsigned long long ULL;
const int N = 1e6 + 10, P = 13331;
char s[N];
int len;
ULL f[N]; // 前缀哈希数组
ULL p[N]; // 记录 p 的 i 次方

// 处理前缀哈希数组以及 p 的 i 次方数组
void init_hash()
{
    f[0] = 0; p[0] = 1;
    for(int i = 1; i <= len; i++)
    {
        f[i] = f[i - 1] * P + s[i];
        p[i] = p[i - 1] * P;
    }
}

// 快速求得任意区间的哈希值
ULL get_hash(int l, int r)
{
```

```
22        return f[r] - f[l - 1] * p[r - l + 1];
23    }
```

如果题目只是简单的求单个字符串的哈希值：

```
1   typedef unsigned long long ULL;
2   const int N = 1e6 + 10;
3   int len;
4   char s[N];
5
6   ULL gethash()
7   {
8       ULL ret = 0;
9       for(int i = 1; i <= len; i++)
10      {
11          ret = ret * p + s[i];
12      }
13      return ret;
14  }
```

4.1 【模板】字符串哈希

> **题目来源：** 洛谷
>
> **题目链接：** P3370 【模板】字符串哈希
>
> **难度系数：** ★★

【题目描述】

> 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。
>
> **友情提醒：如果真的想好好练习哈希的话，请自觉。**

【输入描述】

> 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。

【输出描述】

> 输出包含一行，包含一个整数，为不同的字符串个数。

【示例一】

> 输入：
>
> 5
>
> abc
>
> aaaa
>
> abc
>
> abcc
>
> 12345
>
> 输出：
>
> 4

【解法】

字符串哈希模板题，就是背一个哈希函数~

【参考代码】

```cpp
#include <iostream>
#include <algorithm>

using namespace std;

typedef unsigned long long ULL;

const int N = 1e4 + 10, P = 131;

int n;
ULL a[N];

// 字符串哈希
ULL get_hash(string& s)
{
    ULL ret = 0;
    for(int i = 1; i <= s.size(); i++)
    {
        ret = ret * P + s[i - 1];
    }

    return ret;
}

int main()
{
```

```
27      cin >> n;
28      for(int i = 1; i <= n; i++)
29      {
30          string s; cin >> s;
31          a[i] = get_hash(s);
32      }
33
34      int ret = 1;
35      sort(a + 1, a + 1 + n);
36      for(int i = 2; i <= n; i++)
37      {
38          if(a[i] != a[i - 1]) ret++;
39      }
40
41      cout << ret << endl;
42
43      return 0;
44  }
```

4.2 兔子与兔子

> **题目来源：** 洛谷
>
> **题目链接：** P10468 兔子与兔子
>
> **难度系数：** ★★

【题目描述】

> 很久很久以前，森林里住着一群兔子。
>
> 有一天，兔子们想要研究自己的 DNA 序列。
>
> 我们首先选取一个好长好长的 DNA 序列（小兔子是外星生物，DNA 序列可能包含 26 个小写英文字母）。
>
> 然后我们每次选择两个区间，询问如果用两个区间里的 DNA 序列分别生产出来两只兔子，这两个兔子是否一模一样。
>
> 注意两个兔子一模一样只可能是他们的 DNA 序列一模一样。

【输入描述】

> 第一行输入一个 DNA 字符串 S。
>
> 第二行一个数字 m，表示 m 次询问。
>
> 接下来 m 行，每行四个数字 l1,r1,l2,r2，分别表示此次询问的两个区间，注意字符串的位置从 1 开始编号。

【输出描述】

> 对于每次询问，输出一行表示结果。
>
> 如果两只兔子完全相同输出 `Yes`，否则输出 `No`（注意大小写）。

【示例一】

> 输入：
>
> aabbaabb
>
> 3
>
> 1 3 5 7
>
> 1 3 6 8
>
> 1 2 1 2
>
> 输出：
>
> Yes
>
> No
>
> Yes

【解法】

构造字符串的前缀哈希数组，在前缀哈希数组中快速拿到区间的哈希值。

【参考代码】

```cpp
#include <iostream>

using namespace std;

typedef unsigned long long ULL;

const int N = 1e6 + 10, P = 13331;

int n;
string s;
ULL f[N]; // 前缀哈希数组
ULL p[N]; // p 的 i 次方

void init_hash()
{
    p[0] = 1;

    for(int i = 1; i <= n; i++)
```

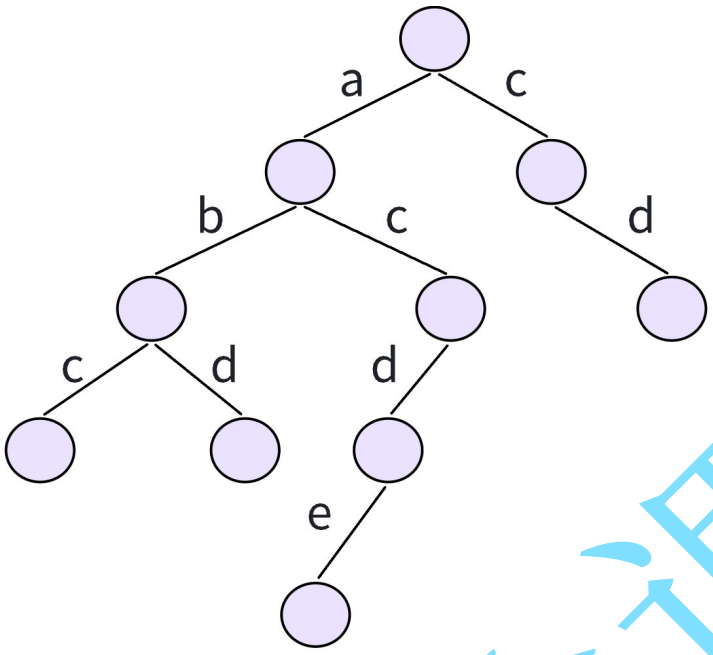
```
    {
        f[i] = f[i - 1] * P + s[i];
        p[i] = p[i - 1] * P;
    }
}

ULL get_hash(int l, int r)
{
    return f[r] - f[l - 1] * p[r - l + 1];
}

int main()
{
    cin >> s;
    n = s.size();
    s = " " + s;

    init_hash();

    int m; cin >> m;
    while(m--)
    {
        int l1, r1, l2, r2; cin >> l1 >> r1 >> l2 >> r2;

        ULL x = get_hash(l1, r1), y = get_hash(l2, r2);

        if(x == y) cout << "Yes" << endl;
        else cout << "No" << endl;
    }

    return 0;
}
```


5. Trie 树

1. 字典树的概念

Trie 树又叫字典树或前缀树，是一种能够快速插入和查询字符串的数据结构。它利用字符串的公共前缀，将字符串组织成一棵树形结构，从而大大提高了存储以及查找效率。

我们可以把字典树想象成一棵多叉树，每一条边代表一个字符，从根节点到某个节点的路径就代表了一个字符串。例如，要存储 `"abc"`、`"abd"`、`"acde"` 以及 `"cd"` 时，构建的字典树如下：



2. 字典树的作用

当我们在字典树的每一个结点位置，额外维护一些信息时，就可以做到很多事情：

- 查询某个单词是否出现过，并且出现几次；
- 查询有多少个单词是以某个字符串为前缀；
- 查询所有以某个前缀开头的单词；（这个作用可以用到输入法中，输入拼音的时候，可以提示可能的单词）

当然，除了上述作用以外，字典树还可以解决别的问题，后续可以在做题中体会。

3. 字典树的实现

实现一个能够**查询单词出现次数**以及**查询有多少个单词是以某个字符串为前缀**的字典树，默认全是小写字母。

准备工作：

```cpp
#include <iostream>
#include <cstring>

using namespace std;
const int N = 1e6 + 10;

int tree[N][26], p[N], e[N];
```

```
8   int idx;
```

插入字符串：

```
1   void insert(string& s)
2   {
3       int cur = 0; // 从根结点开始
4       p[cur]++; // 这个格子经过一次
5
6       for(auto ch : s)
7       {
8           int path = ch - 'a';
9           // 如果没有路
10          if(tree[cur][path] == 0) tree[cur][path] = ++idx;
11          cur = tree[cur][path];
12          p[cur]++;
13      }
14      e[cur]++;
15  }
```

查询字符串出现的次数：

```
1   int find(string& s)
2   {
3       int cur = 0;
4       for(auto ch : s)
5       {
6           int path = ch - 'a';
7           if(tree[cur][path] == 0) return 0;
8           cur = tree[cur][path];
9       }
10      return e[cur];
11  }
```

查询有多少个单词以字符串 `s` 为前缀：

```
1   int find_pre(string& s)
```

```
    {
        int cur = 0;
        for(auto ch : s)
        {
            int path = get_num(ch);
            if(tree[cur][path] == 0) return 0;
            cur = tree[cur][path];
        }
        return p[cur];
    }
```

5.1 【模板】字典树

题目来源： 洛谷

题目链接： P8306 【模板】字典树

难度系数： ★★

【题目描述】

给定 n 个模式串 $s_1, s_2, \ldots, s_n$ 和 q 次询问，每次询问给定一个文本串 t_i，请回答 $s_1 \sim s_n$ 中有多少个字符串 s_j 满足 t_i 是 s_j 的前缀。

一个字符串 t 是 s 的前缀当且仅当从 s 的末尾删去若干个（可以为 0 个）连续的字符后与 t 相同。

输入的字符串大小敏感。例如，字符串 `Fusu` 和字符串 `fusu` 不同。

【输入描述】

本题单测试点内有多组测试数据。

输入的第一行是一个整数，表示数据组数 T。

对于每组数据，格式如下：
第一行是两个整数，分别表示模式串的个数 n 和询问的个数 q。
接下来 n 行，每行一个字符串，表示一个模式串。
接下来 q 行，每行一个字符串，表示一次询问。

【输出描述】

按照输入的顺序依次输出各测试数据的答案。

对于每次询问，输出一行一个整数表示答案。

【示例一】

输入：

```
3
```

3 3
fusufusu
fusu
anguei
fusu
anguei
kkksc
5 2
fusu
Fusu
AFakeFusu
afakefusu
fusuisnotfake
Fusu
fusu
1 1
998244353
9

输出：

2
1
0
1
2
1

【解法】

模板题，仅需维护一个 p 数组，查询多少前缀即可。

【参考代码】

```
1   #include <iostream>
```

```cpp
#include <cstring>

using namespace std;

const int N = 3e6 + 10;

int n, q;
int tr[N][62], p[N];
int idx;

int get_num(char ch)
{
    if(ch >= 'a' && ch <= 'z') return ch - 'a';
    else if(ch >= 'A' && ch <= 'Z') return ch - 'A' + 26;
    else return ch - '0' + 52;
}

void insert(string& s)
{
    int cur = 0;
    p[cur]++;

    for(auto ch : s)
    {
        int path = get_num(ch);
        if(tr[cur][path] == 0) tr[cur][path] = ++idx;

        cur = tr[cur][path];
        p[cur]++;
    }
}

int find_pre(string& s)
{
    int cur = 0;

    for(auto ch : s)
    {
        int path = get_num(ch);
        if(tr[cur][path] == 0) return 0;

        cur = tr[cur][path];
    }
    return p[cur];
}

int main()
```

```
{
    int T; cin >> T;

    while(T--)
    {
        // 清空之前的数据
        // memset(tr, 0, sizeof tr);
        // memset(p, 0, sizeof p);
        for(int i = 0; i <= idx; i++)
        {
            for(int j = 0; j < 62; j++)
            {
                tr[i][j] = 0;
            }
        }

        for(int i = 0; i <= idx; i++) p[i] = 0;

        idx = 0;

        cin >> n >> q;
        while(n--)
        {
            string s; cin >> s;
            insert(s);
        }

        while(q--)
        {
            string s; cin >> s;
            cout << find_pre(s) << endl;
        }
    }


    return 0;
}
```


5.2 于是他错误的点名开始了

> **题目来源：** 洛谷
>
> **题目链接：** P2580 于是他错误的点名开始了
>
> **难度系数：** ★★

【题目描述】

> 这之后校长任命你为特派探员，每天记录他的点名。校长会提供化学竞赛学生的人数和名单，而你需要告诉校长他有没有点错名。（为什么不直接不让他玩炉石。）

【输入描述】

> 第一行一个整数 n ，表示班上人数。
>
> 接下来 n 行，每行一个字符串表示其名字（互不相同，且只含小写字母，长度不超过 50 ）。
>
> 第 $n+2$ 行一个整数 m ，表示教练报的名字个数。
>
> 接下来 m 行，每行一个字符串表示教练报的名字（只含小写字母，且长度不超过 50 ）。

【输出描述】

> 对于每个教练报的名字，输出一行。
>
> 如果该名字正确且是第一次出现，输出 `OK` ，如果该名字错误，输出 `WRONG` ，如果该名字正确但不是第一次出现，输出 `REPEAT` 。

【示例一】

> 输入：
>
> ```
> 5
> a
> b
> c
> ad
> acd
> 3
> a
> a
> e
> ```
>
> 输出：
>
> ```
> OK
> REPEAT
> WRONG
> ```

【解法】

用字典树维护字符串，当查询之后，将 e 数组修改成 -1 即可，

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 5e5 + 10;

int n, m;
int tr[N][26], e[N], idx;

void insert(string& s)
{
    int cur = 0;

    for(auto ch : s)
    {
        int path = ch - 'a';
        if(tr[cur][path] == 0) tr[cur][path] = ++idx;
        cur = tr[cur][path];
    }
    e[cur]++;
}

int find(string& s)
{
    int cur = 0;

    for(auto ch : s)
    {
        int path = ch - 'a';
        if(tr[cur][path] == 0) return 0;
        cur = tr[cur][path];
    }
    // e[cur]

    if(e[cur] > 0)
    {
        int t = e[cur];
        e[cur] = -1;
        return t;
    }

    return e[cur];
}

```

```cpp
int main()
{
    cin >> n;
    while(n--)
    {
        string s; cin >> s;
        insert(s);
    }

    cin >> m;
    while(m--)
    {
        string s; cin >> s;
        int ret = find(s);
        if(ret == 0) cout << "WRONG" << endl;
        else if(ret > 0) cout << "OK" << endl;
        else cout << "REPEAT" << endl;
    }

    return 0;
}
```

5.3 最大异或对

> **题目来源：** 洛谷
>
> **题目链接：** P10471 最大异或对 The XOR Largest Pair
>
> **难度系数：** ★★★

【题目描述】

> 给定 N 个整数 $A_1, A_2, \cdots, A_N$ 中选出两个进行异或计算，得到的结果最大是多少？

【输入描述】

> 第一行一个整数 N，第二行 N 个整数 $A_1, A_2, \cdots, A_N$。

【输出描述】

> 一个整数表示答案。

【示例一】

> 输入：
>
> 3
>
> 1 2 3

> 输出：
>
> 3

【解法】

将所有数按照二进制位存在字典树中，针对每一个数 x ，按照二进制匹配最优的方案。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 1e5 + 10;

int n;
int a[N];

int tr[N * 32][2], idx;

void insert(int x)
{
    int cur = 0;
    for(int i = 31; i >= 0; i--)
    {
        int path = ((x >> i) & 1);
        if(tr[cur][path] == 0) tr[cur][path] = ++idx;
        cur = tr[cur][path];
    }
}

int find(int x)
{
    int cur = 0;
    int ret = 0;
    for(int i = 31; i >= 0; i--)
    {
        int path = ((x >> i) & 1);

        // 贪心：要去相反的路
        if(tr[cur][path ^ 1])
        {
            ret |= (1 << i);
            cur = tr[cur][path ^ 1];
        }
```

```
37          else
38          {
39              cur = tr[cur][path];
40          }
41      }
42
43      return ret;
44  }
45
46  int main()
47  {
48      cin >> n;
49      for(int i = 1; i <= n; i++)
50      {
51          cin >> a[i];
52          insert(a[i]);
53      }
54
55      int ret = 0;
56      for(int i = 1; i <= n; i++)
57      {
58          ret = max(ret, find(a[i]));
59      }
60
61      cout << ret << endl;
62
63      return 0;
64  }
```