板子

STL

vector

```
//将向量b中从下标0 1 2（共三个）的元素赋值给a，a的类型为int型
//它的初始化不和数组一样
vector<int>a(b.begin(),b.begin+3);

//从数组中获得初值
int b[7]={1,2,3,4,5,6,7};
vector<int> a(b,b+7）;

for(auto x : a) {//遍历输出
    cout << x << " ";
}

a.front();  //返回a的第1个元素,当且仅当a存在
a.back();   //返回vector的最后一个数
a.clear();  //清空a中的元素
a.erase(p)//从a中删除迭代器p指定的元素，p必须指向c中的一个真实元素，不能是最后一个元素end()
a.erase(b,e)//从a中删除迭代器对b和e所表示的范围中的元素，返回e
```

支持比较运算 `比较操作==，！=，<，<，<=，>，>=`

```
int main () {
    //支持比较运算
    vector<int> a(4, 3), b(3, 4);
    //a: 3 3 3 3   b:4 4 4
    //比较原理字典序 (根据最前面那个判断，如果一样就往后比较)
    if (a < b) {
        puts("a < b");
    }
    return 0;
}
```

pair

```
//俩种方法初始化
pair<string,int> p("hello",1);
p = make_pair("hello",1);

p("hello",1);
p.first; //第一个元素 =hello
p.second; //第二个元素 = 1
```

string

```
string s = "Hello World!";
s.c_str()           // 返回这个string对应的字符数组的头指针
printf("%s", s.c_str()); //输出 "Hello World!"
```

queue、priority_queue

```
//queue <类型> 变量名
//priority_queue <类型> 变量名;
queue <int> q; //定义一个名为q队列
priority_queue <int> q; //默认是大根堆
//定义小根堆
// 小根堆：priority_queue <类型,vecotr <类型>,greater <类型>> 变量名

// 优先队列获取元素
q.top();// 返回堆顶元素

// 队列和堆没有clear函数
// 所以清空的方法就是重新初始化
```

deque（双向队列）

```
dq.size(); //返回这个双端队列的长度
dq.empty(); //返回这个队列是否为空，空则返回true，非空则返回false
dq.clear(); //清空这个双端队列
dq.front(); //返回第一个元素
dq.back(); //返回最后一个元素
dq.push_back(); //向最后插入一个元素
dq.pop_back(); //弹出最后一个元素
dq.push_front(); //向队首插入一个元素
dq.pop_front();//弹出第一个元素
dq.begin(); //双端队列的第0个数
dq.end(); //双端队列的最后一个的数的后面一个数
```

set、multiset

`set`不允许元素重复，如果有重复就会被忽略，但`multiset`允许.

```
size();// 返回元素个数
empty(); //返回set是否是空的
clear(); //清空
begin(); //第0个数，支持++或--，返回前驱和后继
end(); //最后一个的数的后面一个数，支持++或--，返回前驱和后继
insert(); //插入一个数
find(); //查找一个数
count(); //返回某一个数的个数
erase(x); //删除所以x  时间复杂度 O(k + logn)
erase(s.begin(),s.end());//删除一个迭代器

// 核心函数
lower_bound(x); //返回大于等于x的最小的数的迭代器  核心操作
upper_bound(x); //返回大于x的最小的数的迭代器  不存在返回end()
```

map、multimap

```
insert(); //插入一个数，插入的数是一个pair
erase();
//（1）输入是pair
//（2）输入一个迭代器，删除这个迭代器
find(); //查找一个数
lower_bound(x); //返回大于等于x的最小的数的迭代器
upper_bound(x); //返回大于x的最小的数的迭代器
```

哈希表

```
unordered_set，unordered_map，unordered_muliset,unordered_multimap
//头文件就是加上对应名称
```

不支持`lower_bound()`和`upper_bound()`

bitset

> 它是一种类似数组的结构，它的每一个元素只能是0或1，每个元素仅用1 bit空间,用于节省空间，
> 并且可以直接用01串赋值，可以理解为一个二进制的数组
> ①头文件
> include
> ②初始化

```
bitset<4> bs;    //无参构造，长度为 4，默认每一位为 0

bitset<8> b(12);    //长度为 8，二进制保存，前面用 0 补充

string s = "100101"; //01串赋值
bitset<10> bs(s);    //长度为10，前面用 0 补充
```

```
count(); //返回1的个数
any(); //判断是否至少有一个1
none(); //判断是否全为0
set(); //把所有位置赋值为1
set(k,v); //将第k位变成v
reset(); //把所有位变成0
flip(); //把所有位取反，等价于~
flip(k); //把第k位取反
```

算法Algorithm

- __gcd 最大公约数

```cpp
#include<cstdio>
#include<algorithm>
using namespace std;
int n,m;
int main()
{
    scanf("%d %d",&n,&m);
    int k=__gcd(n,m);//最大公约数
    printf("%d ",k);
    printf("%d", n * m / k); //最小公倍数
    return 0;
}
```

- ```cpp
  swap(a,b);*//交换a和b*
  ```

- `lower_bound()`与`upper_bound()`[二分查找]
  - 时间复杂度O(log n)
  - 使用之前一定要先排序
  - lower_bound()返回数组中第一个大于等于x的数的地址
  - upper_bound()返回数组中第一个大于x的数的地址
  - 将得到的地址减去数组的起始地址可得在数组中的下标

语法技巧

- 加快cin和cout

```cpp
ios::sync_with_stdio(false);
cin.tie(0);
cout.tie(0);
```

审题

一般ACM或者笔试题的时间限制是1秒或2秒。
在这种情况下，C++代码中的操作次数控制在 $10^7 \sim 10^8$ 为最佳。

下面给出在不同数据范围下，代码的时间复杂度和算法该如何选择：

1. $n \leq 30$, 指数级别, dfs+剪枝，状态压缩dp
2. $n \leq 100$ => $O(n^3)$，floyd，dp，高斯消元
3. $n \leq 1000$ => $O(n^2)$，$O(n^2logn)$，dp，二分，朴素版Dijkstra、朴素版Prim、Bellman-Ford
4. $n \leq 10000$ => $O(n * \sqrt{n})$，块状链表、分块、莫队
5. $n \leq 100000$ => $O(nlogn)$ => 各种sort，线段树、树状数组、set/map、heap、拓扑排序、dijkstra+heap、prim+heap、Kruskal、spfa、求凸包、求半平面交、二分、CDQ分治、整体二分、后缀数组、树链剖分、动态树
6. $n \leq 1000000$ => $O(n)$, 以及常数较小的 $O(nlogn)$ 算法 => 单调队列、 hash、双指针扫描、并查集，kmp、AC自动机，常数比较小的 $O(nlogn)$ 的做法：sort、树状数组、heap、dijkstra、spfa
7. $n \leq 10000000$ => $O(n)$，双指针扫描、kmp、AC自动机、线性筛素数
8. $n \leq 10^9$ => $O(\sqrt{n})$，判断质数
9. $n \leq 10^{18}$ => $O(logn)$，最大公约数，快速幂，数位DP
10. $n \leq 10^{1000}$ => $O((logn)^2)$，高精度加减乘除
11. $n \leq 10^{100000}$ => $O(logk \times loglogk)$，k表示位数，高精度加减、FFT/NTT

二叉堆模板
```

> 它可以实现$O(\log n)$地插入或删除某个值，并且$O(1)$地查询最大（或最小)值。

```cpp
// 序列最小和
#include <cstdio>
#include <queue>
#include <algorithm>
using namespace std;

const int N = 1000010;
int n, x, y, a[N], b[N];

struct node{
    int x, y, num;
    node(){}
    node(int X, int Y): x(X),y(Y),num(a[X] + b[Y]){}
} heap[N];

inline void sink(int p){
    int q = p << 1;
    node x = heap[p];
    while (q <= n){
        if (q < n && heap[q+1].num < heap[q].num) ++q;
        if (heap[q].num >= x.num) break;
        heap[p] = heap[q];
        p = q;
        q = p << 1;
    }
    heap[p] = x;
}

inline node getmin(){
    node res = heap[1];
    heap[1] = heap[n--];
    sink(1);
    return res;
}

int main(){
    scanf("%d", &n);
    for (int i = 1; i <= n; ++i)
        scanf("%d", &a[i]);
    for (int i = 1; i <= n; ++i)
        scanf("%d", &b[i]);
    sort(a + 1, a + n + 1);
    sort(b + 1, b + n + 1);
    for (int i = 1; i <= n; ++i)
        heap[i] = node(1, i);
    for (int i = n / 2; i > 0; --i) sink(i);
    for (int i = 1; i <= n; ++i){
        node tmp = heap[1];
        printf("%d ", tmp.num);
        if (tmp.x < n){
            heap[1] = node(tmp.x + 1, tmp.y);
            sink(1);
        }else getmin();
    }
    return 0;
```

```
}
```

```
// 轮廓线
#include <cstdio>
#include <queue>
#include <algorithm>
using namespace std;

const int N = 1000010;
int n = 1, k = 0, nums[N][3], a[N] = {0};

struct Node{
    int x;
    int pos;
    int op;
    Node(){}
    Node(int X, int P, int O): x(X), pos(P), op(O){}
} node[N << 1];

struct Q{
    int h, r;
    Q(){}
    Q(int H, int R): h(H), r(R){}
    bool operator <(const Q &x) const{ return h < x.h; }
};

priority_queue<Q> que;

int main(){
    // freopen("1.txt", "r", stdin);
    while (scanf("%d%d%d", &nums[n][0], &nums[n][2], &nums[n][1]) != EOF){
        node[++k] = Node(nums[n][0], n, 0);
        node[++k] = Node(nums[n][1], n, 1);
        ++n;
    }
    --n;
    sort(node + 1, node + k + 1, [](Node &a, Node &b) { return a.x < b.x;
});
    int pre = 0;
    for (int i = 1; i <= k; ++i) {
        while (!que.empty() && que.top().r <= node[i].x) que.pop();
        if (node[i].op == 0){
            que.emplace(Q(nums[node[i].pos][2], nums[node[i].pos][1]));
        }
        if (i < k && node[i].x == node[i + 1].x) continue;
        int tmp = que.empty() ? 0 : que.top().h;
        if (tmp != pre){
            printf("%d %d ", node[i].x, tmp);
            pre = tmp;
        }
    }
    return 0;
}
```

树状数组模板

```cpp
// 前缀和
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int n;
long long c1[100010] = {0};
long long c2[100010] = {0};

inline int lowbit(int x){
    return x & (x ^ (x - 1));
}

inline void update(int index, long long x){
    for (int i = index; i <= n; i += lowbit(i)){
        c1[i] += x;
        c2[i] += index * x;
    }
    return;
}

inline long long query(int index){
    long long res = 0;
    for (int i = index; i; i -= lowbit(i)){
        res += c1[i] * (index + 1) - c2[i];
    }
    return res;
}

signed main(){
    int m;
    scanf("%d%d", &n, &m);
    vector<long long> nums(n + 1);
    for (int i = 1; i <= n; ++i){
        scanf("%lld", &nums[i]);
        update(i, nums[i]);
    }
    while (m--){
        char s[10];
        scanf("%s", s);
        if (s[0] == 'Q'){
            int index;
            scanf("%d", &index);
            printf("%lld\n", query(index));
        }else{
            int index;
            long long x;
            scanf("%d%lld", &index, &x);
            update(index, x - nums[index]);
            nums[index] = x;
        }
    }
    return 0;
```

```
}
```

线段树模板

线段树是算法中常用的用来维护**区间信息**的数据结构

构建线段树
s和t是当前线段树的左右结点范围，p为父结点下标，arr为构建树的输入数组

```
void build(int s, int t, int p, const vector<int>& arr) {
    if (s == t) {
        tree[p] = SegmentItem(arr[s], 1);
        return;
    }
    int m = s + ((t - s) >> 1);
    build(s, m, p * 2, arr), build(m + 1, t, p * 2 + 1, arr);
    // push_up
    tree[p] = tree[p * 2] + tree[(p * 2) + 1];
}
```

查询

```
SegmentItem find(int l, int r, int s, int t, int p) {
    // [l, r] 为查询区间, [s, t] 为当前节点包含的区间, p 为当前节点的编号
    if (l <= s && t <= r)
        return tree[p];  // 当前区间为询问区间的子集时直接返回当前区间的和
    int m = s + ((t - s) >> 1);
    SegmentItem sum;
    if (r <= m) return find(l, r, s, m, p * 2);
    // 如果左儿子代表的区间 [l, m] 与询问区间有交集, 则递归查询左儿子
    if (l > m) return find(l, r, m + 1, t, p * 2 + 1);
    // 如果右儿子代表的区间 [m + 1, r] 与询问区间有交集, 则递归查询右儿子
    return find(l, r, s, m, p * 2) + find(l, r, m + 1, t, p * 2 + 1);
}
```

例题

给你一个整数数组 `nums` 以及两个整数 `lower` 和 `upper` 。求数组中值位于范围 `[lower, upper]` （包含 `lower` 和 `upper` ）之内的 区间和的个数 。

区间和 `S(i, j)` 表示在 `nums` 中，位置从 `i` 到 `j` 的元素之和，包含 `i` 和 `j` `(i ≤ j)` 。

示例 1：

```
输入：nums = [-2,5,-1], lower = -2, upper = 2
输出：3
解释：存在三个区间：[0,0]、[2,2] 和 [0,2] ，对应的区间和分别是：-2 、-1 、2 。
```

示例 2：

```
输入：nums = [0], lower = 0, upper = 0
输出：1
```

```cpp
class Solution {
    using LL = long long;
public:
    int countRangeSum(vector<int>& nums, int lower, int upper) {
        int ans = 0;
        // 离散
        vector<LL> preSum{0LL};
        for(auto& i : nums) preSum.push_back(preSum.back()+i);
        set<LL> sums;
        for(auto& i : preSum){
            sums.insert(i);
            sums.insert(i-lower);
            sums.insert(i-upper);
        }
        unordered_map<LL, int> idx;
        int i = 0;
        for(auto& v : sums) idx[v] = i++;
        int n = idx.size() - 1;
        // 建树
        for(auto& x : preSum){
            auto l = idx[x-upper], r = idx[x-lower];
            ans += query(l, r, 0, n, 1); //从根节点id=1开始查询
            update(idx[x], idx[x], 0, n, 1, 1);//从根结点id=1开始更新
        }
        return ans;
    }

    inline int ls(int p){return p<<1;}//左儿子
    inline int rs(int p){return p<<1|1;}//右儿子
    inline void f(int l, int r, int p, int k){
        tag[p] += k;
        arr[p] += k * (r - l + 1);
        //由于是这个区间统一改变，所以ans数组要加元素个数次
    }
    void push_up(int p){arr[p] = arr[ls(p)] + arr[rs(p)];}
    void push_down(int s, int t, int p){
        // 如果当前节点的懒标记非空,则更新当前节点两个子节点的值和懒标记值
        auto m = (s + t) >> 1;
        f(s, m, ls(p), tag[p]);
        f(m + 1, t, rs(p), tag[p]);
        // 清空父节点懒标记
        tag[p] = 0;
    }
    void build(int s, int t, int p) {
        tag[p] = 0;
        // 对 [s,t] 区间建立线段树,当前根的编号为 p
        if (s == t) {
            arr[p] = arr[s];
            return;
        }
        int m = (t + s) >> 1;
        build(s, m, ls(p));
        build(m + 1, t, rs(p));
        // push_up
        push_up(p);
```

```
    }
    void update(int l, int r, int s, int t, int p, int k) {
        /// [l, r] 为修改区间, [s, t] 为当前节点包含的区间, p 为当前节点的编号, k 为被修改的元素的变化量
        if (l <= s && t <= r) {
            f(s, t, p, k);
            //arr[p] += k;
            return;
        }
        // push down 懒标记
        push_down(s, t, p);

        int m = (t + s) >> 1;
        if (l <= m) update(l, r, s, m, ls(p), k);
        if (r > m) update(l, r, m + 1, t, rs(p), k);
        push_up(p);
    }

    int query(int l, int r, int s, int t, int p) {
        // [l, r] 为查询区间, [s, t] 为当前节点包含的区间, p 为当前节点的编号
        if (l <= s && t <= r)
            return arr[p];  // 当前区间为询问区间的子集时直接返回当前区间的和
        int m = (t + s) >> 1;
        if (r <= m) return query(l, r, s, m, ls(p));
        // 如果左儿子代表的区间 [l, m] 与询问区间有交集, 则递归查询左儿子
        if (l > m) return query(l, r, m + 1, t, rs(p));
        // 如果右儿子代表的区间 [m + 1, r] 与询问区间有交集, 则递归查询右儿子
        return query(l, r, s, m, ls(p)) + query(l, r, m + 1, t, rs(p));
    }

private:
    int arr[2000000]{};
    int tag[2000000]{};
    int n{0};
};
```

并查集模板

```
// 亲戚
#include <stdio.h>
#include <algorithm>
using namespace std;

const int N = 5010;
int n, m, p, x, y;
int fa[N];
int r[N];

inline void set(int n){
    for(int i = 1; i <= n; ++i){
        fa[i] = i;
        r[i] = 1;
    }
}
```

```

inline int find(int x){
    int y = x;
    while (y != fa[y])
        y = fa[y];
    return y;
}

inline void join(int x, int y){
    int xRoot = find(x);
    int yRoot = find(y);
    if (xRoot == yRoot) return;
    else if (r[xRoot] < r[yRoot]) fa[xRoot] = yRoot;
    else if (r[xRoot] > r[yRoot]) fa[yRoot] = xRoot;
    else{
        fa[yRoot] = xRoot;
        ++r[xRoot];
    }
}

int main(){
    scanf("%d%d%d", &n, &m, &p);
    set(n);
    for (int i = 0; i < m; ++i){
        scanf("%d%d", &x, &y);
        join(x, y);
    }
    while (p--){
        scanf("%d%d", &x, &y);
        if (find(x) == find(y)) printf("Yes\n");
        else printf("No\n");
    }
    return 0;
}
```

最小生成树模板

```
// 道路修建
#include <stdio.h>
#include <cmath>
#include <algorithm>
using namespace std;

const int N = 1010;
const int M = 1000010;
int n, m, x, y, k = 0;
int nums[N][2], fa[N], r[N];
double len[N];

struct Edge{
    int x, y;
    double len;
} edge[M];

inline double length(int i, int j){
```

```
    return sqrt(pow(nums[i][0] - nums[j][0], 2) + pow(nums[i][1] - nums[j][1], 2));
}

inline void set(int n){
    for(int i = 1; i <= n; ++i){
        fa[i] = i;
        r[i] = 1;
        len[i] = 0;
    }
}

inline int find(int x){
    while (x != fa[x])
        x = fa[x];
    return x;
}

inline void join(int x, int y, double d, int mode){
    int xRoot = find(x);
    int yRoot = find(y);
    if (xRoot == yRoot) return;
    else if (r[xRoot] < r[yRoot]){
        fa[xRoot] = yRoot;
        if (mode) len[yRoot] += len[xRoot] + d;
    }else if (r[xRoot] > r[yRoot]){
        fa[yRoot] = xRoot;
        if (mode) len[xRoot] += len[yRoot] + d;
    }else{
        fa[yRoot] = xRoot;
        if (mode) len[xRoot] += len[yRoot] + d;
        ++r[xRoot];
    }
}

int main(){
    scanf("%d%d", &n, &m);
    set(n);
    for (int i = 1; i <= n; ++i){
        scanf("%d%d", &nums[i][0], &nums[i][1]);
    }
    for (int i = 0; i < m; ++i){
        scanf("%d%d", &x, &y);
        join(x, y, 0, 0);
    }
    for (int i = 1; i < n; ++i){
        for (int j = i + 1; j <= n; ++j){
            edge[k].x = i;
            edge[k].y = j;
            edge[k++].len = length(i, j);
        }
    }
    sort(edge, edge + k, [](Edge &a, Edge &b) { return a.len < b.len; });
    for (int i = 0; i < k; ++i){
        join(edge[i].x, edge[i].y, edge[i].len, 1);
    }
    printf("%.2f\n", len[find(1)]);
    return 0;
```

```
}
```

KMP模板

```
// 给定一个 haystack 字符串和一个 needle 字符串，在 haystack 字符串中找出 needle 字符串出现的第一个位置 (从0开始)。如果不存在，则返回  -1。
// 示例 1: 输入: haystack = "hello", needle = "ll" 输出: 2
// 示例 2: 输入: haystack = "aaaaa", needle = "bba" 输出: -1
class Solution {
public:
    int strStr(string haystack, string needle) {
        int n = haystack.size(), m = needle.size();
        //特例
        if (!m){
            return 0;
        }
        //next数组
        int j = 0, k = -1;
        vector<int> next(m);
        next[0] = -1;
        while (j < m - 1){
            if (k == -1 || needle[j] == needle[k]){
                j++;
                k++;
                if (needle[j] != needle[k]){
                    next[j] = k;
                }else{
                    next[j] = next[k];
                }
            }else{
                k = next[k];
            }
        }
        //KMP算法
        int i = 0;
        j = 0;
        while (i < n && j < m){
            if (j == -1 || haystack[i] == needle[j]){
                i++;
                j++;
            }else{
                j = next[j];
            }
        }
        if (j == m){
            return i - j;
        }else{
            return -1;
        }
    }
};
```

其他

万能头： `#include <bits/stdc++.h>`

dev:添加-static-libgcc -std=c++14

因子数集合

```
vector<int>fac[M+10]; // 每个数的所有因子（包括1，不包括它自身）
void get_fac(){
    for(int i=1;i<=M/2;i++){
        for(int j=2*i;j<=M;j+=i){ // 2*i,...,k*i(<=M)
            fac[j].push_back(i);
        }
    }
}
```