

## 基础算法

类型	范围
int	\$[-2^{31},2^{31}]\$
long long	\$[-2^{63},2^{63} - 1]\$

### 基本单位

基本单位	详解
Bit (比特)	计算机中最小数据单位，表示二进制的一位
Byte (字节)	计算机存储的基本单位， 1 Byte = 8 Bits

## 快速排序

1. 确定分界点
2. 重点：调整区间，根据分界点重新划分数据分布，保证左边数小于分界点，右边数大于分界点
  - 暴力做法：
    - 创建两个数组a, b
    - 扫描原数组，小于分界点的数放到a，大于分界点的数放到b
    - 然后再把a和b里面的数放回原数组
  - 优雅做法（双指针）：
    - 创建两个指针l、r，分别直指向数组的头部和尾部
    - l从左边开始向右扫描，碰到大于等于分界点的数就停下来
    - r从右边开始向左扫描，碰到小于等于分界点的数就停下来
    - 交换两个指针指向的数，然后各自移动1位
    - 直到l和r相遇
3. 递归处理左右两端

```
##include<iostream>
using namespace std;
const int N = 1e6 + 10;

int n;
```

```

int q[N];

void quick_sort(int q[], int l, int r){
    if(l >= r) return;
    int x = q[(int)((l + r) / 2)];
    int i = l - 1, j = r + 1;
    while(i < j){
        do i++; while(q[i] < x);
        do j--; while(q[j] > x);
        if(i < j) swap(q[i], q[j]);
    }
    quick_sort(q, l, j);
    quick_sort(q, j + 1, r);
}

int main(){
    scanf("%d", &n);
    for(int i = 0; i < n; i++) scanf("%d", &q[i]);

    quick_sort(q, 0, n - 1);

    for(int i = 0; i < n; i++) printf("%d ", q[i]);

    return 0;
}

```

## 归并排序

- 时间复杂度:  $O(n \log n)$
- 确定分离点:  $\text{mid} = \frac{l + r}{2}$
  - 递归排序数组左部分和右部分
  - 归并排序完的数组，合二为一
    - 用双指针，分别指向两个已排序好的数组的第一位
    - 比较指针指向的数，更小的数写入答案数组
    - 当访问完任意一个数组后退出循环

```

##include<iostream>
using namespace std;

const int N = 1e6 + 10;
int n;
int q[N], temp[N];

```

```

void merge_sort(int q[],int l,int r){
    if(l >= r) return;
    int mid = (l + r) / 2;
    merge_sort(q, l, mid);
    merge_sort(q, mid + 1, r);

    int k = 0,i = l,j = mid + 1;
    while(i <= mid && j <= r){
        if(q[i] <= q[j]) temp[k++] = q[i++];
        if(q[i] > q[j]) temp[k++] = q[j++];
    }
    while(i <= mid) temp[k++] = q[i++];
    while(j <= r) temp[k++] = q[j++];

    for(int i = l, j = 0;i <= r;i++,j++) q[i] = temp[j];
}

int main(){
    scanf("%d",&n);

    for(int i = 0;i < n;i++) scanf("%d",&q[i]);

    merge_sort(q,0,n - 1);

    for(int i = 0;i < n;i++) printf("%d ",q[i]);

    return 0;
}

```

## 整数二分

```

// 检查查找的数是否满足性质
bool check(int x){...}

// 查找最小的满足check条件的值
int binary_search1(int l,int r){
    while(l < r){
        int mid = (l + r) / 2;
        if(check(mid)) r = mid;
        else l = mid + 1;
    }
    return l;
}

```

```

// 查找最大的满足check条件的值
int binary_search2(int l, int r){
    while(l < r){
        int mid = (l + r + 1) / 2; // 加1是防止死循环
        if(check(mid)) l = mid;
        else r = mid - 1;
    }
    return l;
}

```

## 浮点数二分

```

bool check(double x){}

// 不需要处理边界
double binary_search(double l, double r){
    while(r - l > eps){ // eps表示精度，取决题目对精度要求
        double mid = (l + r) / 2;
        if(check(mid)) r = mid;
        else l = mid;
    }
    return l;
}

```

## 高精度

也就是把大整数的每一位存到数组里，个位存到数组的第一个位置，后面以此类推

- 加法：

```

##include<iostream>
##include<vector>
const int N = 1e6 + 10;
int n;
using namespace std;

vector<int> add(vector<int> &a, vector<int> &b) { //加引用可以提高效率
    vector<int> c;
    int t = 0; //进位
    for(int i = 0; i < a.size() || i < b.size(); i++) {
        if(i < a.size()) t += a[i];
        if(i < b.size()) t += b[i];
        c.push_back(t % 10);
        t /= 10;
    }
    return c;
}

```

```

        c.push_back(t % 10);
        t /= 10;
    }
    if(t) c.push_back(1);
    return c
}
int main(){
    string a, b;
    vector<int> A, B;
    cin >> a >> b;
    for(int i = a.size() - 1 ;i >= 0;i--) A.push_back(a[i] - '0');
    for(int i = b.size() - 1 ;i >= 0;i--) A.push_back(b[i] - '0');
    vector<int> c = add(A,B);
    for(int i = c.size() - 1;i >= 0 ;i--) printf("%d",c[i]);
}

```

- 減法：

- \$A > B\$: 算\$A-B\$
- \$A < B\$: 算\$-(B-A)\$

```

##include<iostream>
##include<vector>
const int N = 1e6 + 10;
int n;
using namespace std;

// 判断A是不是大于等于B
bool cmp(vector<int> &a,vector<int> &b){
    if(a.size() != b.size()) return a.size() > b.size();
    for(int i = a.size() - 1;i >= 0;i++){
        if(a[i] != b[i]) return a[i] > b[i];
    }
    return true;
}

vector<int> sub(vector<int> &a,vector<int> &b){
    vector<int> c;
    int t = 0;
    for(int i = 0; i < a.size();i++){
        t = a[i] - t;
        if(i < b.size()) t -= b[i];
        c.push_back((t + 10) % 10);
        if(t < 0) t = 1;
        else t = 0;
    }
}

```

```

    }
    while(c.size() > 1 && c.back() == 0) c.pop_back();
    return c;
}

int main(){
    String a, b;
    vector<int> A, B;
    cin >> a >> b;
    for(int i = a.size() - 1 ;i >= 0;i--) A.push_back(a[i] - '0');
    for(int i = b.size() - 1 ;i >= 0;i--) B.push_back(b[i] - '0');
    if(cmp(A,B)){
        vector<int> c = sub(A,B);
        for(int i = c.size() - 1;i >= 0 ;i--) printf("%d",c[i]);
    }
    else{
        vector<int> c = sub(B,A);
        printf("-");
        for(int i = c.size() - 1;i >= 0 ;i--) printf("%d",c[i]);
    }
}

```

- 乘法：

```

vector<int> multi(vector<int> &a, int b){
    vector<int> c;
    int t = 0;
    for(int i = 0;i < a.size();i++){
        t += a[i] * b;
        c.push_back(t % 10);
        t = t / 10;
    }
    while(t > 0){
        c.push_back(t % 10);
        t /= 10;
    }
    // if(t > 0) c.push_back(t);
    while(c.size() > 1 && c.back() == 0) c.pop_back();
    return c;
}

```

- 除法

每次push\_back()的值都不会超过两位数，证明如下  

$$r_k \&= r_{k-1} \% b \>\& \leq b - 1 \>\& r_k * 10 \& \leq (b-1) * 10 \>\& r_k * 10 + a \& \leq 10b - 1 < 10b$$
也就是说\$

/ b\$最大是\$(10 b -1)/b\$, 最大到9

```
vector<int> div(vector<int> &a, int b, int &r){
    vector<int> c;
    r = 0;
    for(int i = a.size() - 1; i >= 0; i--) {
        r = r * 10 + a[i];
        c.push_back(r / b);
        r %= b;
    }
    while(t > 0) {
        c.push_back(t)
    }
    while(c.size() > 1 && c.back() == 0) c.pop_back();
    reverse(c.begin(), c.end());
}
```

## 前缀和

- 一维前缀和

给定数组\$a\_1,a\_2,\dots,a\_n\$

前缀和数组为\$S\_i = a\_1 + a\_2 + a\_3 + \dots + a\_i\$

\$S\_k - S\_l\$表示\$(l,k]\$的元素总和

```
int t;
int b[n];
// 令s0 = 0 前缀和数组长度变为 n + 1
void sum(int a[]){
    int t = 0;
    for(int i = 0 ; i <= n; i++) {
        b[i] = t;
        t += a[i];
    }
}
```

- 二维前缀和

- 给定二维数组，二维前缀和\$S\_{ij} = \sum\_{m \leq i, n \leq j} \{a\_{mn}\}\$

- 二维数组从  $a[1][1]$  开始存储
- $S_{ij} - S_{in} - S_{mj} + S_{mn}$  表示在二维数组里， $(i,j)$ 、 $(m,n)$ 、 $(i,n)$ 、 $(m,j)$  围起来的矩形的所有元素的和
- $S_{ij} = a_{ij} + S_{i-1,j} + S_{i,j-1} - S_{i-1,j-1}$
- $S_{0,1}, S_{0,0}, S_{1,0}$  等都是 0

```

int s[M][N];
int main(){
    s[0][0] = 0;
    for(int i = 1; i < n; i++){
        for(int j = 1; j < m; j++){
            scanf("%d", &a[i][j]);
            s[i][j] = s[i][j - 1] + a[i][j] - s[i - 1][j - 1] + s[i - 1][j]
        }
    }
}

```

## 差分

- 一维差分

给定  $a_1, a_2, \dots, a_n$ ，构造  $b_1, b_2, \dots, b_n$ ，使得  $a_i = b_1 + b_2 + \dots + b_i$ ，则  $b$  是  $a$  的差分， $a$  是  $b$  的前缀和。  
 $b_1 = a_1$ ， $b_2 = a_2 - a_1$ ， $b_3 = a_3 - a_2$ ， $\dots$ ， $b_n = a_n - a_{n-1}$ 。  
假设我们需要对  $[l, r]$  区间内的  $a_i$  全部加上  $C$ ，这时候如果直接扫描时间复杂度是  $O(N)$ 。但我们可以求  $a$  的差分  $b$ ，让  $b_l + C$  那么  $a_l$  后面的数字都会加上  $C$ ，我们再让  $b_{r+1} - C$ ，就可以让  $[l, r]$  区间内的  $a$  都加上  $C$ ，而其他地方不变，也适用于给  $a[i]$  赋初值。

- 二维差分

原矩阵：  $a_{ij}$

差分矩阵：  $b_{ij}$

```

##include<iostream>
using namespace std;

int n, m, q;

const int N = 1010;

```

```

int a[N][N], b[N][N];

void insert(int x1,int y1,int x2,int y2,int c){
    b[x1][y1] += c;
    b[x2 + 1][y2 + 1] += c;
    b[x1][y2 + 1] -= c;
    b[x2 + 1][y1] -= c;
}

int main(){
    cin >> n >> m >> q;
    for(int i = 1;i <= n;i++){
        for(int j = 1;j <= m;j++){
            int t;
            cin >> t;
            insert(i,j,i,j,t);
        }
    }
    while(q--){
        int x1,y1,x2,y2,c;
        cin >> x1 >> y1 >> x2 >> y2 >> c;
        insert(x1,y1,x2,y2,c);
    }

    for(int i = 1;i <= n;i++){
        for(int j = 1;j <= m;j++){
            b[i][j] = b[i][j] + b[i - 1][j] + b[i][j - 1] - b[i - 1][j - 1]
        }
    }

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

```

## 双指针算法

可以将\$O(n^2)\$优化到\$O(n)\$

没什么好说的，做题

## 位运算

n的二进制表示中第k位数字是多少

从0开始算的

1. 先把第k位移到最后一位:  $n >> k$
  2. 看个位是几:  $n \& 1$
  3.  $n >> k \& 1;$
- $\$lowbit(x)$ : 返回x的**最右边（最后）**一位1和后面的数字
    - $x = 1010$ , 返回10
    - $x = 101000$ , 返回1000
    - $x \& -x$

## 离散化

将不连续的数字映射成连续的数字

值域跨度很大，但是用到的值很少，用到的值十分稀疏

- 要对原始数据去重
- 用二分找要映射的位置
- 本质上是把数组里面的数映射成数组的下标

## 区间求和

- alls数组是为了存储需要离散化的点，对其进行去重和排序是为了防止映射错误
- add和query分别存储要插入的数据和搜索的数据
- 最后进行区间搜索的时候要先离散化

```
##include<iostream>
##include<vector>
##include<algorithm>
using namespace std;

const int N = 3e5 + 10; // 不仅有插入的点，还有查询的点，离散化后的数组的长度要能容
int a[N], s[N]; // 离散化后用前缀和求解

vector<int> alls; // 所有要离散化的下标

typedef pair<int, int> PII;
vector<PII> add, query;
int n, m;
```

```

// x为数轴上的点，返回的是离散化后的结果
int find(int x){
    int l = 0, r = alls.size() - 1;
    while(l < r){
        int mid = l + r >> 1;
        if(alls[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return r + 1;
}

int main(){
    cin >> n >> m;
    // 插入操作
    for(int i = 0;i < n;i++){
        int x, c;
        cin >> x >> c;
        alls.push_back(x);
        add.push_back({x,c});
    }

    // 查找操作
    while(m--){
        int l,r;
        cin >> l >> r;
        query.push_back({l,r});
        alls.push_back(l);
        alls.push_back(r);
    }

    //排序、去重，不然映射出错
    sort(alls.begin(),alls.end());
    alls.erase(unique(alls.begin(),alls.end()),alls.end());

    // 处理插入
    for(auto x: add){
        int t = find(x.first);
        a[t] += x.second;
    }

    // 前缀和
    for(int i = 1;i <= alls.size();i++){
        s[i] = a[i] + s[i - 1];
    }

    // 输出答案
    for(auto x: query){
        int l = find(x.first);
        int r = find(x.second);

```

```
        cout << s[r] - s[l - 1] << endl;
    }
}
```

## 区间合并

把多个有交集的区间合并成一个区间

1. 按区间左端点排序
2. 扫描区间，并把区间进行合并

```
##include<iostream>
##include<algorithm>
##include<vector>
using namespace std;
const int N = 100010;

vector<pair<int,int>> t;

int n;

void merge(vector<pair<int,int>> &s){
    vector<pair<int,int>> temp;
    sort(s.begin(),s.end());
    // 用第一对数据作为初始维护区间
    int l = s[0].first,r = s[0].second;
    for(int i = 1;i < s.size();i++){
        if(s[i].first > r) {
            temp.push_back({l,r});
            l = s[i].first;
            r = s[i].second;
        }
        else r = max(r,s[i].second); // 将两种情况合二为一
    }
    // 循环结束后一定会有一个区间没有被处理，在这里进行处理
    temp.push_back({l,r});
    t = temp;
}

int main(){
    cin >> n;
    while(n--){
        int a,b;
```

```
    cin >> a >> b;
    t.push_back({a, b});
}
merge(t);
cout << t.size();
}
```

## 数据结构

以数组模拟为主

### 链表

1. 单链表：邻接表（存储树和图）
  - e[N]: 存的是节点的值
  - ne[N]: 存的是节点的next指针
2. 双链表：每个节点有两个指针
  - l[N]: 记录节点的左指针
  - r[N]: 记录节点的右指针
3. 邻接表：n个单链表

#### 单链表

```
##include<iostream>

using namespace std;

const int N = 1000010;
/*
e[N]存的是当前节点的值
ne[N]存的是当前节点的指针
head存的是头节点
idx存的是链表已有节点的个数
*/
int e[N], ne[N];
int head, idx;

void init(){
    head = -1;
    idx = 0;
}
```

```
void add_to_head(int c){
    e[idx] = c;
    ne[idx] = head;
    head = idx++;
}

void erase(int k){
    // 处理k为0的情况
    if(k == -1) {
        head = ne[head];
    }
    ne[k] = ne[ne[k]];
}

void add(int x,int c){
    e[idx] = c;
    ne[idx] = ne[x];
    ne[x] = idx++;
}

int main(){
    init();
    int m;
    cin >> m;
    while(m--){
        int k,x;
        char op;
        cin >> op;
        if(op == 'H'){
            cin >> x;
            add_to_head(x);
        }
        else if(op == 'D'){
            cin >> k ;
            erase(k - 1);
        }
        else if(op == 'I'){
            cin >> k >> x;
            add(k - 1,x);
        }
    }
    for(int i = head;i != -1;i = ne[i]){
        cout << e[i] << " ";
    }
}
```

## 双链表

```
##include<iostream>
using namespace std;

const int N = 100010;
int l[N], r[N], e[N];
int idx;
// 令双链表的头节点下标为0, 尾节点下标为1
void init(){
    r[0] = 1, l[1] = 0;
    idx = 2;
}
// 模拟的是在k的右侧插入
void add(int k, int x){
    e[idx] = x;
    r[idx] = r[k];
    l[idx] = k;
    l[r[k]] = idx;
    r[k] = idx;
    idx++;
}

void remove(int k){
    l[r[k]] = l[k];
    r[l[k]] = r[k];
}

int main(){
    int m;
    init();
    cin >> m;
    while(m--){
        string op;
        cin >> op;
        int k, x;
        if(op == "L"){
            cin >> x;
            add(0, x);
        }
        else if(op == "R"){
            cin >> x;
        }
    }
}
```

```

        add(l[1],x);
    }
    else if(op == "D"){
        cin >> k;
        remove(k + 1); // 第1个插入的数实际上idx是2
    }
    else if(op == "IL"){
        cin >> k >> x;
        add(l[k + 1],x);
    }else if(op == "IR"){
        cin >> k >> x;
        add(k + 1,x);
    }
}
for(int i = r[0];i != 1;i = r[i]){
    cout << e[i] << " ";
}
}

```

## 栈

### 先进后出

```

##include<iostream>
using namespace std;

const int N = 100010;
int s[N];
int top;

void init(){
    top = -1;
}

bool isempty(){
    return top == -1;
}

void push(int x){
    s[++top] = x;
}

void pop(){
    top--;
}

```

```

int query(){
    return s[top];
}

int main(){
    init();
    int m;
    cin >> m;
    while(m--){
        int x;
        string op;
        cin >> op;
        if(op == "push"){
            cin >> x;
            push(x);
        }
        else if(op == "pop") pop();
        else if(op == "empty"){
            if(isempty()) cout << "YES" << endl;
            else cout << "NO" << endl;
        }
        else if(op == "query") cout << query()<< endl;
    }
}

```

## 队列

### 先进先出

```

##include<iostream>
using namespace std;

const int N = 100010;
int s[N], top, tail;

void init(){
    top = 0;
    tail = -1;
}

void push(int x){
    s[++tail] = x;
}

bool isempty(){

```

```

        return tail < top;
    }

    void pop(){
        top++;
    }

    int query(){
        return s[top];
    }

int main(){
    int m;
    init();
    cin >> m;
    while(m--){
        int x;
        string op;
        cin >> op;
        if(op == "push"){
            cin >> x;
            push(x);
        }
        else if(op == "pop"){
            pop();
        }
        else if(op == "empty"){
            if(isempty()) cout << "YES" << endl;
            else cout << "NO" << endl;
        }
        else if( op == "query"){
            cout << query() << endl;
        }
    }
}

```

## 单调栈和单调队列

**单调栈：**给定一个序列，求每一个数左边离他最近的数且比它小在什么地方

题解出处：<https://www.acwing.com/video/5400/>

```

##include<iostream>

using namespace std;

```

```

const int N = 1e5 + 10;
int st[N];
int top = -1;
int n;
// 先用朴素算法模拟，然后找出规律，可以用单调栈求解
int main(){
    cin >> n;
    for(int i = 0;i < n;i++){
        int x;
        cin >> x;
        while(top >= 0 && st[top] >= x) top--;
        if(top != -1) cout << st[top] << " ";
        else cout << -1 << " ";
        st[++top] = x;
    }
}

```

### 单调队列：求滑动窗口最大值或者最小值

- 用一个单调（双端）队列维护，每一时刻这个单调队列都是有序的，每移动一次输出一次队头元素，本质上和单调栈思想一致，都是把不可能的答案给去掉

```

##include<iostream>
using namespace std;

const int N = 1e6 + 10;
int a[N],st[N];
int head = 0,tail = -1;
int main(){

    int n,k;
    cin >> n >> k;
    for(int i = 0;i < n;i++){
        cin >> a[i];
    }
    // 求最小值
    for(int i = 0;i < n;i++){
        if( i - k + 1 > st[head]) head++;
        while(tail >= head && a[st[tail]] >= a[i]) tail--;
        st[++tail] = i;
        if(i >= k - 1) cout << a[st[head]] << " ";
    }
    cout << endl;
    // 求最大值
    head = 0,tail = -1;
    for(int i = 0;i < n;i++){

```

```

    if(i - k + 1 > st[head]) head++;
    while(tail >= head && a[st[tail]] <= a[i]) tail--;
    st[++tail] = i;
    if(i >= k - 1) cout << a[st[head]] << " ";
}
cout << endl;
}

```

## 队列存值的写法

```

##include<iostream>

using namespace std;
const int N = 1e6 + 10;
int n,k;
int tail = -1,head = 0;
int q[N],a[N];

int main(){
    cin >> n >> k;
    for(int i = 1;i <= n;i++){
        cin >> a[i];
    }
    for(int i = 1;i <= n;i++){
        if(head <= tail && i > k && q[head] == a[i - k]) head++;
        while(tail >= head && q[tail] > a[i]) tail--;
        q[++tail] = a[i];
        if(i > k - 1){
            cout << q[head] << " ";
        }
    }
    cout << endl;
    head = 0, tail = -1;
    for(int i = 1;i <= n;i++){
        if(head <= tail && i > k && q[head] == a[i - k]) head++;
        while(tail >= head && q[tail] < a[i]) tail--;
        q[++tail] = a[i];
        if(i > k - 1){
            cout << q[head] << " ";
        }
    }
}

```

## KMP

- 字符串匹配的一种算法，本质上是根据已有信息减少重复匹配。
- 算法维护一个next数组，存储的是最长的相等真前缀和真后缀的长度。
- 当前面n个字符匹配成功，但是第  $n + 1$  个字符匹配失败，算法将模式串向右滑动  $n - next[n]$  位，再从第  $n+1$  位比较

```
##include<iostream>
using namespace std;

const int N = 1e6 + 10;
char s[N], p[N];
// 字符串是从下标为1开始，当然也可以从0开始，所以第1个字符能移动的距离ne[1]为0
int ne[N];
int main(){
    int n, m;
    cin >> n >> p + 1 >> m >> s + 1;
    /*
    j是存储下标i可以往前移动从而避免重复匹配的下标
        a b a b a
        1 2 3 4 5
    ne  0 0 1 2 3
        a b a b a
        a b a b a
    */
    for(int i = 2, j = 0; i <= n; i++){
        while(j && p[i] != p[j + 1]) j = ne[j];
        if(p[i] == p[j + 1]) j++;
        ne[i] = j;
    }

    for(int i = 1, j = 0; i <= m; i++){
        while(j && s[i] != p[j + 1]) j = ne[j];
        if(s[i] == p[j + 1]) j++;
        if(j == n) {
            // 我们代码的下标是从1开始计数，题目要求从0开始，因此不是i-n+1
            cout << i - n << " ";
            j = ne[j];
        }
    }
}
```

## Trie

高效地存储和查找字符串集合的数据结构

如图所示：



如果某个字符是某个单词结尾，需要在这个字符后打上标记

根节点是空节点，下标（编号）为0

```
##include<iostream>
##include<string.h>
using namespace std;

const int N = 1e5 + 10;
int son[N][26], idx, cnt[N];
int n;

void insert(char str[]){
    int p = 0;
    for(int i = 0; str[i]; i++){
        if(!son[p][str[i] - 'a']) son[p][str[i] - 'a'] = ++idx;
        p = son[p][str[i] - 'a'];
    }
    cnt[p]++;
}

int query(char str[]){
    int p = 0;
    for(int i = 0; str[i]; i++){
        if(!son[p][str[i] - 'a']) return 0;
        p = son[p][str[i] - 'a'];
    }
    return cnt[p];
}

int main(){
    ios::sync_with_stdio(false);
    cout.tie(0), cin.tie(0);
    cin >> n;
    while(n--){
        string s;
        char x[N];
        cin >> s;
        insert(s);
    }
}
```

```

    if(s == "I"){
        cin >> x;
        insert(x);
    }
    else if(s == "Q"){
        cin >> x;
        cout << query(x) << endl;
    }
}

}

```

## 最大异或对

```

##include<iostream>
using namespace std;

const int N = 1e5 + 10;
const int M = 31 * N; //用二进制表示最多有31*N个节点
int n;
int a[N], son[M][2];
int idx = 0;
void insert(int x){
    int p = 0;
    for(int i = 30;i >= 0;i--){
        int u = x >> i & 1;
        if(!son[p][u]) son[p][u] = ++idx;
        p = son[p][u];
    }
}

int search(int x){
    int p = 0,res = 0;
    for(int i = 30;i >= 0;i--){
        int u = x >> i & 1;
        if(son[p][!u]){
            p = son[p][!u]; // 要找最大的肯定要找不同的，比如1和0, 0和1
            res = res * 2 + 1; //比如100 -> 1001 , 起始就是 (100)_2 * 2 + 1
        }
        else {
            p = son[p][u];// 这是因为找不到不同的，但是为了左移只能退而求其次
            res = res * 2;
        }
    }
    return res;
}

```

```

}

int main(){
    scanf("%d",&n);
    int ans = 0;
    for(int i = 0;i < n;i++) {
        scanf("%d",&a[i]);
        insert(a[i]);
    }
    for(int i = 0;i < n;i++) {
        ans = max(ans,search(a[i]));
    }
    cout << ans;
}

```

## 并查集

1. 合并两个集合
2. 询问两个元素是否在一个集合中

### 用树维护集合

- 每一个集合的编号是根节点的编号
- 对于每个节点都存下父节点是谁，对于根节点，它的爸爸是他自己
- 合并集合：p[x]是x的编号，p[y]是y的编号，只要让p[x] = y即可
- 路径压缩：某一结点第一次向上搜索根节点，找到根节点后把走过的路径的节点全部指向根节点

### 维护额外信息

- 记录集合元素数量：维护一个size数组，这个数组只对根节点有效

### 合并集合

```

##include<iostream>
using namespace std;

const int N = 1e5 + 10;
int n,m;

int h[N];

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

```

```

}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0), cout.tie(0);
    cin >> n >> m;
    for(int i = 0;i < n;i++) h[i] = i; // 每个节点就是一个集合
    int a,b;
    while(m--){
        char c;
        cin >> c;
        if(c == 'M'){
            cin >> a >> b;
            if(find(a) == find(b)) continue;
            h[find(a)] = find(b);
        }
        else{
            cin >> a >> b;
            if(find(a) == find(b)) cout << "Yes" << endl;
            else cout << "No" << endl;
        }
    }
}

```

## 堆

是一个完全二叉树，最后一层节点从左往右排列，其他层节点非空，下标从1开始

用数组模拟，支持修改或者删除任意一个元素（STL里面的不支持）

基本操作

1. 插入一个数
2. 求集合最小值
3. 删除最小值
4. 删除、修改任意一个元素

存储

- x的左儿子：2x
- x的右儿子：2x+1

向上和向下调整

- 从 $n/2$ 开始调整
- 向上调整 ( `up` ) : 用于插入或节点值变小时。
- 向下调整 ( `down` ) : 用于删除或节点值变大时。

## 哈希表

### 存储结构

#### 1. 开放寻址法

- 只开一个一维数组存储所有哈希值
- 如果位置已经被占，那么看下一个位置，循环找到没有人占领的位置
- 核心是`find`函数
  - 如果 $x$ 已经存在，返回 $x$ 存储的位置
  - 如果 $x$ 不存在，返回 $x$ 要插入的位置

```
// 开放寻址法
##include<iostream>
##include<cstring>
using namespace std;

const int N = 2e5 + 3; // 用两倍是因为要让每个数都能找到位置, +3是经验设定, 可以减/
const int NULL = 0x3f3f3f3f; // 设定初值
int h[N];
int n;

// 这个函数能返回x要插入的位置, 也能返回x在哈希表的位置
int find(int x){
    int k = (x % N + N) % N;
    while(h[k] != NULL && h[k] != x){
        k++;
        if(k == N) k = 0;
    }
    return k;
}

int main(){
    memset(h, 0x3f, sizeof(h)); // 初始化
```

```

string s;
cin >> n;
while(n--){
    cin >> s;
    int x;
    if(s == "I"){
        cin >> x;
        h[find(x)] = x;
    }
    else if(s == "Q"){
        cin >> x;
        int k = find(x);
        if(h[k] == x) cout << "Yes" << endl;
        else cout << "No" << endl;
    }
}
}

```

## 2. 拉链法

- 开一个一维数组存所有的哈希值
- 这个一维数组存的是链表的头节点
- 有相同哈希值的数字插到对应的链表里

```

// 拉链法
##include<iostream>
##include<cstring>
using namespace std;

const int N = 1e5 + 3; // 找比100000大的最小质数，可以减少冲突出现的概率
int h[N];
int e[N], ne[N];
int idx = 0;
int n ;

void insert(int x){
    int k = (x % N + N) % N; // 设定哈希函数，这里这么写是因为有负数 先把x压缩到[-1
    e[idx] = x;
    ne[idx] = h[k];
    h[k] = idx++;
}

bool find(int x){

```

```

int k = (x % N + N) % N;
for(int i = h[k]; i != -1; i = ne[i]){
    if(e[i] == x) return true;
}
return false;
}

int main(){
    memset(h, -1, sizeof(h)); // 初始化哈希表全为-1
    cin >> n;
    string s;
    int x;
    while(n--){
        cin >> s;
        if(s == "I"){
            cin >> x;
            insert(x);
        }
        else if(s == "Q"){
            cin >> x;
            if(find(x)) cout << "Yes" << endl;
            else cout << "No" << endl;
        }
    }
}

```

## 字符串哈希

用于快速判断两个字符串是不是相等

- 把字符串的每个字符都看成一个整数，组合成一个p进制，然后算出这个字符串的十进制数再模一个q，得到这个字符串的哈希值
  - p一般取131或者13331
  - q取 $2^{64}$ ，可以直接用unsigned long long进行存储，溢出也不管，这样就不用写取模
- 会把字符串的每个前缀的哈希值都算出来，用h数组存储
  - $h[0] = 0$
  - $h[i] = h[i - 1] * 131 + str[i];$
- 对于任意子串，可以用这个算法以O(1)复杂度算出来其哈希值
  - $h[L-R] = h[R] - h[L-1] * 131^{R-L+1}$
  - $131^{R-L+1}$ 可以用数组求

- 这是因为在  $h[R]$  中， $h[L-1]$  其实贡献了高位，但是实际  $h[L-1]$  是低位，因此要减去右移后的

```

##include<iostream>
using namespace std;

const int N = 1e5 + 10;
char s[N];
unsigned long long P[N], h[N];
int n, m;
int p = 131;

int search(int l, int r){
    return h[r] - h[l - 1] * P[r - l + 1];
}

int main(){
    cin >> n >> m;
    cin >> s;
    P[0] = 1;
    for(int i = 0; i < n; i++){
        P[i + 1] = P[i] * p;
        h[i + 1] = h[i] * p + s[i];
    }

    while(m--){
        int l1, r1, l2, r2;
        cin >> l1 >> r1 >> l2 >> r2;
        if(search(l1, r1) == search(l2, r2)) cout << "Yes" << endl;
        else cout << "No" << endl;
    }
}

```

## STL

### vector

- 变长数组，支持比较运算
- 初始化：`vector<int> a = (10, 3)`：初始化长度为10，初值为3的数组
- `size()`：查看元素个数
- `empty()`：判空
- `clear()`：清空

- `front()/back()`: 返回第一个/最后一个数
- `push_back() /pop_back()`: 插入/删除最后一个数
- `begin()/end()`: 第0个数/最后一个数的后面一个数的迭代器

### **pair<int,int>**

- 可以定义一个二元组
- `first()/second()`: 第1/2个元素
- 初始化: `p = (20, "zxb")` 或者 `p = make_pair(20, "zxb")`

### **string**

- `substr()`: 返回子串
  - `substr(1,2)`: 从下标为1的字符开始, 输出长度为2的子串
- `size()`: 查看元素个数
- `empty()`: 判空
- `clear()`: 清空

### **queue**

- `size()`: 查看元素个数
- `empty()`: 判空
- `push()`: 往队尾插入
- `front()`: 返回队头元素
- `back()`: 返回队尾元素
- `pop()`: 弹出队头元素

### **priority\_queue (优先队列, 堆)**

#### **默认是大根堆**

- `push()`: 插入元素
- `front()`: 返回堆顶元素
- `pop()`: 弹出堆顶元素

### **deque (双端队列)**

- `size()`: 查看元素个数

- empty(): 判空
- clear(): 清空
- front(): 返回队头元素
- back(): 返回队尾元素
- push\_back() /pop\_back(): 插入/弹出最后一个数
- push\_front() /pop\_front(): 插入/弹出第一个数
- begin()/end(): 第0个数/最后一个数的后面一个数的迭代器

## stack

- size(): 查看元素个数
- empty(): 判空
- push(): 向栈顶插入元素
- top(): 返回栈顶元素
- pop(): 弹出栈顶元素

## set

### 不能有重复元素

- size(): 查看元素个数
- empty(): 判空
- clear(): 清空
- insert(): 插入一个元素
- find(): 查找一个元素
- count(): 返回某个元素个数
- erase()
  - 如果输入是一个数x, 删除所有x
  - 输入一个迭代器, 删除这个迭代器
- lower\_bound(): 返回大于等于x最小的数的迭代器, 不存在返回end()
- upper\_bound(): 返回大于x的最小的数的迭代器, 不存在返回end()
- begin()/end(): 第0个数/最后一个数的后面一个数的迭代器

## map

- size(): 查看元素个数

- empty(): 判空
- clear(): 清空
- insert(): 插入一个元素，插入的是一个pair
- erase(): 输入的参数是pair或者迭代器
- `m['key']`: 可以用key去索引value
- lower\_bound(): 返回大于等于x最小的数的迭代器，不存在返回end()
- upper\_bound(): 返回大于x的最小的数的迭代器，不存在返回end()

## multiset

### 可以有重复元素

- size(): 查看元素个数
- empty(): 判空
- clear(): 清空
- insert(): 插入一个元素
- find(): 查找一个元素
- count(): 返回某个元素个数
- erase()
  - 如果输入是一个数x，删除所有x
  - 输入一个迭代器，删除这个迭代器
- lower\_bound(): 返回大于等于x最小的数的迭代器，不存在返回end()
- upper\_bound(): 返回大于x的最小的数的迭代器，不存在返回end()

## multimap

- size(): 查看元素个数
- empty(): 判空
- clear(): 清空
- insert(): 插入一个元素，插入的是一个pair
- erase(): 输入的参数是pair或者迭代器
- `m['key']`: 可以用key去索引value
- lower\_bound(): 返回大于等于x最小的数的迭代器，不存在返回end()
- upper\_bound(): 返回大于x的最小的数的迭代器，不存在返回end()

## unordered\_map、unordered\_set、unordered\_multiset、unordered\_multimap

基于哈希表实现，增删改查时间复杂度是O(1)

不支持 lower\_bound() 和 upper\_bound()

## 搜索和图论

### DFS

深度优先搜索，使用栈

重点：递归结束条件的选择+状态标记+递归后的恢复

### 排列数字

```
##include<iostream>
using namespace std;
const int N = 10;
int n,t[N];
bool state[N];

void dfs(int p){
    if(p == n){
        for(int i = 0;i < n;i++) cout << t[i] << " ";
        cout << endl;
    }
    for(int i = 1;i <= n;i++){
        if(!state[i]){
            t[p] = i;
            state[i] = true;
            dfs(p + 1);
            state[i] = false;
        }
    }
}

int main(){
    cin >> n;
    dfs(0);
}
```

n皇后

```

##include<iostream>
using namespace std;

const int N = 12,M = 2 * N;
char d[N][N];
int n;
bool col[N],e[M],ne[M];
// 第u行
void dfs(int u){
    if(u == n + 1){
        for(int i = 1;i <= n;i++){
            for(int j = 1;j <= n;j++)
                cout << d[i][j];
            cout << endl;
        }
        cout << endl;
        return ;
    }

    for(int i = 1;i <= n;i++){
        if(!col[i] && !e[i + u] && !ne[n - i + u]){
            col[i] = true,e[i + u] = true,ne[n - i + u] = true;
            d[u][i] = 'Q';
            dfs(u + 1);
            col[i] = false,e[i + u] = false,ne[n - i + u] = false;
            d[u][i] = '.';
        }
    }
}

int main(){
    cin >> n;
    for(int i = 1;i <= n;i++){
        for(int j = 1;j <= n;j++)
            d[i][j] = '.';
    }
    dfs(1);
    return 0;
}

```

## BFS

广度优先搜索，使用队列，一般求什么最短的，最少操作次数的，用BFS

当所有边的权重都为1的时候，才能用BFS

## 走迷宫

```
##include<iostream>
using namespace std;
typedef pair<int,int> PII;
const int N = 110;
int d[N][N],p[N][N]; //d存迷宫 p存步数
PII q[N * N];
bool st[N][N];
int head = 0,tail = -1;
int dx[4] = {0,1,0,-1},dy[4] = {-1, 0, 1, 0};
int n,m;
int bfs(){
    st[0][0] = true;
    p[0][0] = 0;
    q[++tail] = {0,0};
    while(head <= tail){
        auto x = q[head++];
        for(int i = 0;i < 4;i++){
            int a = x.first + dx[i],b = x.second + dy[i]; // a横坐标、b纵坐标
            if(a >= 0 && a < n && b >= 0 && b < m && !st[a][b] && d[a][b]
                st[a][b] = true;
                p[a][b] = p[x.first][x.second] + 1;
                q[++tail] = {a,b};
            }
        }
    }
    return p[n - 1][m - 1];
}
int main(){
    cin >> n >> m;
    for(int i = 0;i < n;i++){
        for(int j = 0;j < m;j++)
            cin >> d[i][j];
    }
    cout << bfs();
}
```

## 图中点的层次

```
##include<iostream>
##include<queue>
```

```

##include<cstring>
using namespace std;
const int N = 1e5 + 10;
int dist[N], e[N], ne[N], h[N], idx;
bool st[N];
int n, m;
queue<int> q;

void add(int a, int b){
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

int main(){
    cin >> n >> m;
    memset(h, -1, sizeof h);
    memset(dist, 0x3f, sizeof(dist));
    while(m--){
        int a, b;
        cin >> a >> b;
        add(a, b);
    }
    q.push(1);
    dist[1] = 0;
    st[1] = true;
    while(q.size()){
        int t = q.front();
        q.pop();
        for(int i = h[t]; i != -1; i = ne[i]){
            int j = e[i];
            if(!st[j]){
                dist[j] = dist[t] + 1;
                q.push(j);
                st[j] = true;
            }
        }
    }
    if(dist[n] == 0x3f3f3f3f) cout << -1;
    else cout << dist[n];
}

```

## 拓扑排序

核心思想是将图中的所有顶点排成一个序列，使得对于图中的每一条有向边  $(u \rightarrow v)$ ，在排序中顶点  $u$  都位于顶点  $v$  的前面

适用于有向无环图

```

##include<iostream>
##include<cstring>
using namespace std;
const int N = 1e5 + 10;
int n,m;
int h[N],e[N],ne[N],idx;
int q[N],d[N];

void add(int a,int b){
    e[idx] = b,ne[idx] = h[a],h[a] = idx++;
}

bool topsort(){
    int head = 0,tail = -1;
    for(int i = 1;i <= n;i++){
        if(!d[i]) q[++tail] = i;
    }
    while(head <= tail){
        int x = q[head++];
        for(int i = h[x];i != -1;i = ne[i]){
            int j = e[i];
            d[j]--;
            if(!d[j]) q[++tail] = j;
        }
    }
    return tail == n - 1;
}

int main(){
    memset(h,-1,sizeof h);
    cin >> n >> m;
    int a,b;
    for(int i = 1;i <= m;i++){
        cin >> a >> b;
        add(a,b);
        d[b]++;
    }
    if(topsort()){
        for(int i = 0;i < n;i++) cout << q[i] << " ";
    }
    else {
        cout << -1;
    }
}

```

# 最短路



- 朴素Dijkstra算法适用于稠密图
- 堆优化版Dijkstra算法适用于稀疏图：用优先队列
- 稀疏图用邻接表存
- 稠密图用邻接矩阵存
- 没有负环才能用SPFA

## 朴素Dijkstra算法

- 找到路径最短的点，标记一下
- 用这个点更新其他的路径

```
##include<iostream>
##include<cstring>
##include<algorithm>
using namespace std;

const int N = 510,M = 1e5 + 10;
int h[N],e[M],w[M],ne[M],idx;
int n,m;
int dist[N];
bool st[N];

void add(int a,int b,int c){
    e[idx] = b,ne[idx] = h[a],w[idx] = c,h[a] = idx++;
}

int dijkstra(){
    memset(dist,0x3f,sizeof(dist));
    dist[1] = 0;
    for(int i = 1;i < n;i++){
        int t = -1;
        for(int j = 1;j <= n;j++){
            if(!st[j] && (t == -1 || dist[t] > dist[j]) )
                t = j;
        }
        if (dist[t] == 0x3f3f3f3f) break;
        st[t] = true;
        for(int j = h[t]; j != -1;j = ne[j]){
            int k = e[j]; // k为节点编号
            dist[k] = min(dist[k],dist[t] + w[j]);
        }
    }
}
```

```

    }
    return dist[n];
}

int main(){
    cin >> n >> m;
    memset(h, -1, sizeof h);
    int a,b,c;
    while(m--){
        cin >> a >> b >> c;
        add(a,b,c);
    }
    int res = dijkstra();
    if(res == 0x3f3f3f3f) cout << -1;
    else cout << res;
}

```

## 堆优化版

- 找到路径最短的点，标记一下（用优先队列直接找到）
- 用这个点更新其他的路径

```

##include<iostream>
##include<cstring>
##include<algorithm>
##include<queue>
using namespace std;

const int N = 2e5 + 10, M = 2e5 + 10;
int h[N], e[M], w[M], ne[M], idx;
int n, m;
int dist[N];
bool st[N];

typedef pair<int, int> PII;

void add(int a, int b, int c){
    e[idx] = b, ne[idx] = h[a], w[idx] = c, h[a] = idx++;
}

int dijkstra(){
    memset(dist, 0x3f, sizeof(dist));
    dist[1] = 0;
    priority_queue <PII, vector<PII>, greater<PII>> heap;
    heap.push({0, 1});

```

```

        while(heap.size()){
            auto x = heap.top();
            heap.pop();
            if(st[x.second]) continue;
            st[x.second] = true;
            for(int i = h[x.second]; i != -1; i = ne[i]){
                int j = e[i];
                if(dist[j] > x.first + w[i]){
                    dist[j] = x.first + w[i];
                    heap.push({dist[j],j});
                }
            }
        }
        return dist[n];
    }

int main(){
    cin >> n >> m;
    memset(h, -1, sizeof h);
    int a,b,c;
    while(m--){
        cin >> a >> b >> c;
        add(a,b,c);
    }
    int res = dijkstra();
    if(res == 0x3f3f3f3f) cout << -1;
    else cout << res;
}

```

## Bellman-ford算法

- 先循环k次（循环k次就是找到不少于k条边的最短路径）
  - 需要拷贝上一次遍历的结果
- 内循环遍历所有的边，更新每个点的最短路径

## SPFA算法

- 初始化所有数组
- 当队列不空的时候取出队头元素并更新其他点的路径
- 更新过的点放到队列里面，不能重复放，用st数组判断点是否已经放进队列

```

##include<iostream>
##include<queue>
##include<cstring>

```

```

using namespace std;
const int N = 1e5 + 10;
typedef pair<int,int> PII;
int n,m;

int e[N],ne[N],idx,w[N],h[N],dist[N];
bool st[N];
void add(int x,int y,int z){
    e[idx] = y,ne[idx] = h[x],w[idx] = z,h[x] = idx++;
}

int spfa(){
    memset(dist,0x3f,sizeof dist);
    queue<PII> q;
    dist[1] = 0;
    q.push({0,1}); // 最短距离是0, 编号是1
    st[1] = true;
    while(q.size()){
        auto t = q.front();
        q.pop();
        int dis = t.first,vec = t.second;
        st[vec] = false;
        for(int i = h[vec];i != -1;i = ne[i]){
            int j = e[i];
            if(dist[j] > dist[vec] + w[i]){ // 这里不能用dis, 因为dist[vec]在
                dist[j] = dist[vec] + w[i];
                if(!st[j]){
                    q.push({dist[j],j});
                    st[j] = true;
                }
            }
        }
    }
    return dist[n] >= 0x3f3f3f3f / 2; // 可能有负权边
}

int main(){
    memset(h,-1,sizeof h);
    cin >> n >> m;
    while(m--){
        int x,y,z;
        cin >> x >> y >> z;
        add(x,y,z);
    }
    if(spfa()) cout << "impossible" << endl;
    else cout << dist[n] << endl;
}

```

- 判断负环

- 可以直接维护一个cnt数组，记录最短路径的边数，当边数大于节点数，说明有负环
- 需要将所有点都加到队列里面
- dist数组并不是用来记录最短路径的，是一个工具数组，因为只要有负权环，dist存的数就会越来越小，因此其初始值也是任意的

```
##include<iostream>
##include<queue>
##include<cstring>
using namespace std;
const int N = 2010, M = 10010;

int n,m;

int e[M],ne[M],idx,w[M],h[N],dist[N],cnt[N];

bool st[N];
void add(int x,int y,int z){
    e[idx] = y,ne[idx] = h[x],w[idx] = z,h[x] = idx++;
}

bool spfa(){
    queue<int> q;
    for(int i = 1;i <= n;i++){
        q.push(i);
        st[i] = true;
    } // 最短距离是0，编号是1
    while(q.size()){
        auto t = q.front();
        q.pop();
        st[t] = false;
        for(int i = h[t];i != -1;i = ne[i]){
            int j = e[i];
            if(dist[j] > dist[t] + w[i]){ // 这里不能用dis，因为dist[vec]在循
                dist[j] = dist[t] + w[i];
                cnt[j] = cnt[t] + 1;
                if(cnt[j] >= n) return true;
                if(!st[j]){
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }
}
```

```

    return false;
}

int main(){
    memset(h, -1, sizeof h);
    cin >> n >> m;
    while(m--){
        int x,y,z;
        cin >> x >> y >> z;
        add(x,y,z);
    }
    if(spfa()) cout << "Yes" << endl;
    else cout << "No" << endl;
}

```

## Floyd算法

- 用邻接矩阵存
- 三层循环，最后邻接矩阵存的就是最短路径长度

```

##include<iostream>
##include<cstring>
##include<algorithm>
using namespace std;

const int N = 210;
int n,m,k;
int d[N][N];

void floyd(){
    for(int k = 1;k <= n;k++){
        for(int i = 1;i <= n;i++){
            for(int j = 1;j <= n;j++)
                d[i][j] = min(d[i][j],d[i][k] + d[k][j]);
        }
    }
}

int main(){
    memset(d, 0x3f, sizeof d);
    cin >> n >> m >> k;
    for(int i = 1;i <= n;i++) d[i][i] = 0;
    while(m--){

```

```
int x,y,z;
cin >> x >> y >> z;
d[x][y] = min(d[x][y],z);
}
floyd();
while(k--){
    int x,y;
    cin >> x >> y;
    if(d[x][y] >= 0x3f3f3f3f / 2) cout << "impossible" << endl;
    else cout << d[x][y] << endl;
}
}
```