一、基础算法

快速排序算法模板

```
void quick_sort(int q[], int l, int r)
{
    //递归的终止情况
    if (l >= r) return;

     //选取分界线。这里选数组中间那个数
    int i = l - 1, j = r + 1, x = q[l + r >> 1];
    //划分成左右两个部分
    while (i < j)
    {
        do i ++ ; while (q[i] < x);
        do j -- ; while (q[j] > x);
        if (i < j) swap(q[i], q[j]);
    }
    //对左右部分排序
    quick_sort(q, l, j), quick_sort(q, j + 1, r);
}
```

边界问题

因为边界问题只有这两种组合，不能随意搭配

```
x不能取q[l]和q[l+r>>1];
quick_sort(q,l,i-1),quick_sort(q,i,r);
```

```
x不能取q[r]和q[(l+r+1)>>1];
quick_sort(q,l,j),quick_sort(q,j+1,r);
```

mid 取 l+r >> 1
则 (l, j), (j+1, r)
mid 取 ((l+r+1)>>1
则 (l, i-1), (i, r)

归并排序算法模板

```
void merge_sort(int q[], int l, int r)
{
    //递归的终止情况
    if (l >= r) return;
    //第一步：分成子问题
    int mid = l + r >> 1;
    //第二步：递归处理子问题
    merge_sort(q, l, mid);
    merge_sort(q, mid + 1, r);

    //第三步：合并子问题
    int k = 0, i = l, j = mid + 1;
    while (i <= mid && j <= r)
        if (q[i] <= q[j]) tmp[k ++ ] = q[i ++ ];
        else tmp[k ++ ] = q[j ++ ];
```

```
    while (i <= mid) tmp[k ++ ] = q[i ++ ];
    while (j <= r) tmp[k ++ ] = q[j ++ ];
    //第四步：复制回原数组
    for (i = l, j = 0; i <= r; i ++, j ++ ) q[i] = tmp[j];
}
```

整数二分算法模板

> 对lower_bound来说，它寻找的就是第一个满足条件“值大于等于x”的元素的位置；对upper_bound函数来说，它寻找的是第一个满足“值大于 x”的元素的位置。

```
bool check(int x) {/* ... */} // 检查x是否满足某种性质

// 区间[l, r]被划分成[l, mid]和[mid + 1, r]时使用：
int bsearch_1(int l, int r)
{
    while (l < r)
    {
        int mid = l + r >> 1;
        if (check(mid)) r = mid;    // check()判断mid是否满足性质
        else l = mid + 1;//左加右减
    }
    return l;
}
// 区间[l, r]被划分成[l, mid - 1]和[mid, r]时使用：
int bsearch_2(int l, int r)
{
    while (l < r)
    {
        int mid = l + r + 1 >> 1;//如果下方else后面是l则这里加1
        if (check(mid)) l = mid;
        else r = mid - 1;//左加右减
    }
    return l;
}
```

浮点数二分算法模板

```
bool check(double x) {/* ... */} // 检查x是否满足某种性质

double bsearch_3(double l, double r)
{
    const double eps = 1e-6;   // eps 表示精度，取决于题目对精度的要求
    while (r - l > eps)
    {
        double mid = (l + r) / 2;
        if (check(mid)) r = mid;
        else l = mid;
    }
    return l;
}
```

高精度加法

```
// C = A + B, A >= 0, B >= 0
vector<int> add(vector<int> &a,vector<int> &b){
    //c为答案
    vector<int> c;
    //t为进位
    int t=0;
    for(int i=0;i<a.size()||i<b.size();i++){
        //不超过a的范围添加a[i]
        if(i<a.size())t+=a[i];
        //不超过b的范围添加b[i]
        if(i<b.size())t+=b[i];
        //取当前位的答案
        c.push_back(t%10);
        //是否进位
        t/=10;
    }
    //如果t!=0的话向后添加1
    if(t)c.push_back(1);
    return c;
}
```

高精度减法

```
// C = A - B, 满足A >= B, A >= 0, B >= 0
vector<int> sub(vector<int> &A, vector<int> &B)
{
    //答案
    vector<int> C;
    //遍历最大的数
    for (int i = 0, t = 0; i < A.size(); i ++ )
    {
        //t为进位
        t = A[i] - t;
        //不超过B的范围t=A[i]-B[i]-t;
        if (i < B.size()) t -= B[i];
        //合二为一，取当前位的答案
        C.push_back((t + 10) % 10);
        //t<0则t=1
        if (t < 0) t = 1;
        //t>=0则t=0
        else t = 0;
    }
    //去除前导零
    while (C.size() > 1 && C.back() == 0) C.pop_back();
    return C;
}
```

高精度乘低精度

```
// C = A * b, A >= 0, b >= 0
vector<int> mul(vector<int> &A, int b)
{
    //类似于高精度加法
    vector<int> C;
```

```
    //t为进位
    int t = 0;
    for (int i = 0; i < A.size() || t; i ++ )
    {
        //不超过A的范围t=t+A[i]*b
        if (i < A.size()) t += A[i] * b;
        //取当前位的答案
        C.push_back(t % 10);
        //进位
        t /= 10;
    }
    //去除前导零
    while (C.size() > 1 && C.back() == 0) C.pop_back();

    return C;
}
```

高精度乘高精度

```
char a1[10001], b1[10001];
int a[10001], b[10001], i, x, len, j, c[10001];

int main () {
    cin >> a1 >> b1; //不解释，不懂看前面
    int lena = strlen(a1); //每个部分都很清楚
    int lenb = strlen(b1); //这只是方便你们复制
    for (i = 1; i <= lena; i++)
        a[i] = a1[lena - i] - '0';//倒序存储
    for (i = 1; i <= lenb; i++)
        b[i] = b1[lenb - i] - '0';//倒序存储
    for (i = 1; i <= lenb; i++)
        for (j = 1; j <= lena; j++)
            c[i + j - 1] += a[j] * b[i];//存每位答案
    for (i = 1; i < lena + lenb; i++)
        if (c[i] > 9) {
            c[i + 1] += c[i] / 10;//进位
            c[i] %= 10;//取当前位答案
        }
    len = lena + lenb;
    while (c[len] == 0 && len > 1)//去除前导零
        len--;
    for (i = len; i >= 1; i--)//输出答案
        cout << c[i];
    return 0;
}
```

高精度除低精度

```
// A / b = C ... r, A >= 0, b > 0
vector<int> div(vector<int> &A, int b, int &r)//高精度A，低精度b，余数r
{
    vector<int> C;//答案
    r = 0;
    for (int i = A.size() - 1; i >= 0; i -- )
    {
```

```
        r = r * 10 + A[i];//补全r>=b
        C.push_back(r / b);//取当前位的答案
        r %= b;//r%b为下一次计算
    }
    reverse(C.begin(), C.end());//倒序为答案
    while (C.size() > 1 && C.back() == 0) C.pop_back();//去除前导零
    return C;
}
```

一维前缀和

前缀和可以用于快速计算一个序列的区间和，也有很多问题里不是直接用前缀和，但是借用了前缀和的思想。

```
预处理:s[i]=a[i]+a[i-1]
求区间[l,r]:sum=s[r]-s[l-1]
"前缀和数组"和"原数组"可以合二为一
```

应用

```
const int N=100010;
int a[N];
int main(){
    int n,m;
    scanf("%d",&n);
    for(int i=1;i<=n;i++)scanf("%d",&a[i]);
    for(int i=1;i<=n;i++)a[i]=a[i-1]+a[i];
    scanf("%d",&m);
    while(m--){
        int l,r;
        scanf("%d%d",&l,&r);
        printf("%d\n",a[r]-a[l-1]);
    }
    return 0;
}
```

二维前缀和

```
计算矩阵的前缀和：s[x][y] = s[x - 1][y] + s[x][y -1] - s[x-1][y-1] + a[x][y]
以(x1, y1)为左上角，(x2, y2)为右下角的子矩阵的和为：
计算子矩阵的和：s = s[x2][y2] - s[x1 - 1][y2] - s[x2][y1 - 1] + s[x1 - 1][y1 -1]
```

应用

```
int s[1010][1010];

int n,m,q;

int main(){
    scanf("%d%d%d",&n,&m,&q);
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++)
            scanf("%d",&s[i][j]);
```

```
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++)
            s[i][j]+=s[i-1][j]+s[i][j-1]-s[i-1][j-1];
    while(q--){
        int x1,y1,x2,y2;
        scanf("%d%d%d%d",&x1,&y1,&x2,&y2);
        printf("%d\n",s[x2][y2]-s[x2][y1-1]-s[x1-1][y2]+s[x1-1][y1-1]);
    }
    return 0;
}
```

一维差分

差分是前缀和的逆运算，对于一个数组a，其差分数组b的每一项都是a [i]和前一项a [i - 1]的差。

注意：差分数组和原数组必须分开存放！！！！

```
给区间[l, r]中的每个数加上c：B[l] += c, B[r + 1] -= c
```

应用

```
using namespace std;
int a[100010],s[100010];

int main(){
    int n,m;
    cin>>n>>m;
    for(int i=1;i<=n;i++)cin>>a[i];
    for(int i=1;i<=n;i++)s[i]=a[i]-a[i-1];// 读入并计算差分数组
    while(m--){
        int l,r,c;
        cin>>l>>r>>c;
        s[l]+=c;
        s[r+1]-=c;// 在原数组中将区间[l, r]加上c
    }
    for(int i=1;i<=n;i++){
        s[i]+=s[i-1];
        cout<<s[i]<<' ';
    }// 给差分数组计算前缀和，就求出了原数组
    return 0;
}
```

二维差分

```
给以(x1, y1)为左上角，(x2, y2)为右下角的子矩阵中的所有元素加上c：
S[x1, y1] += c, S[x2 + 1, y1] -= c, S[x1, y2 + 1] -= c, S[x2 + 1, y2 + 1] += c
```

应用

```
const int N = 1e3 + 10;
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][y1] -= c;
    b[x1][y2 + 1] -= c;
    b[x2 + 1][y2 + 1] += c;
}
int main()
{
    int n, m, q;
    cin >> n >> m >> q;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            cin >> a[i][j];
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            insert(i, j, i, j, a[i][j]);      //构建差分数组
        }
    }
    while (q--)
    {
        int x1, y1, x2, y2, c;
        cin >> x1 >> y1 >> x2 >> y2 >> c;
        insert(x1, y1, x2, y2, c);//加c
    }
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            b[i][j] += b[i - 1][j] + b[i][j - 1] - b[i - 1][j - 1];  //二维前缀和
        }
    }
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            printf("%d ", b[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

关于前缀和 与 差分的相关博客链接：https://blog.csdn.net/qq_39757593/article/details/129219491

位运算

```
求n的第k位数字: n >> k & 1
返回n的最后一位1：lowbit(n) = n & -n
```

双指针算法

```
for (int i = 0, j = 0; i < n; i ++ )
{
    while (j < i && check(i, j)) j ++ ;
    // 具体问题的逻辑
}
常见问题分类：
    (1) 对于一个序列，用两个指针维护一段区间
    (2) 对于两个序列，维护某种次序，比如归并排序中合并两个有序序列的操作
```

离散化

离散化的本质是建立了一段数列到自然数之间的映射关系（value -> index)，通过建立新索引，来缩小目标区间，使得可以进行一系列连续数组可以进行的操作比如二分，前缀和等...

离散化首先需要排序去重：

1. 排序：sort(alls.begin(),alls.end())
2. 去重：alls.earse(unique(alls.begin(),alls.end()),alls.end());

```
vector<int> alls; // 存储所有待离散化的值
sort(alls.begin(), alls.end()); // 将所有值排序
alls.erase(unique(alls.begin(), alls.end()), alls.end());   // 去掉重复元素

// 二分求出x对应的离散化的值
int find(int x) // 找到第一个大于等于x的位置
{
    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; // 映射到1, 2, ...n
}
```

应用

```
typedef pair<int, int> PII;

const int N = 300010;

int n, m;
int a[N], s[N];

vector<int> alls;//存入下标容器
vector<PII> add, query;//add增加容器，存入对应下标和增加的值的大小
//query存入需要计算下标区间和的容器
int find(int x)
{
    int l = 0, r = alls.size() - 1;
    while (l < r)//查找大于等于x的最小的值的下标
    {
        int mid = l + r >> 1;
        if (alls[mid] >= x) r = mid;
        else l = mid + 1;
```

```
    }
    return r + 1;//因为使用前缀和，其下标要+1可以不考虑边界问题
}

int main()
{
    cin >> n >> m;
    for (int i = 0; i < n; i ++ )
    {
        int x, c;
        cin >> x >> c;
        add.push_back({x, c});//存入下标即对应的数值c

        alls.push_back(x);//存入数组下标x=add.first
    }

    for (int i = 0; i < m; i ++ )
    {
        int l, r;
        cin >> l >> r;
        query.push_back({l, r});//存入要求的区间

        alls.push_back(l);//存入区间左右下标
        alls.push_back(r);
    }

    // 区间去重
    sort(alls.begin(), alls.end());
    alls.erase(unique(alls.begin(), alls.end()), alls.end());

    // 处理插入
    for (auto item : add)
    {
        int x = find(item.first);//将add容器的add.secend值存入数组a[]当中，
        a[x] += item.second;//在去重之后的下标集合alls内寻找对应的下标并添加数值
    }

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

    // 处理询问
    for (auto item : query)
    {
        int l = find(item.first), r = find(item.second);//在下标容器中查找对应的左右两端[l~r]下标，然后通过下标得到前缀和相减再得到区间a[l~r]的和
        cout << s[r] - s[l - 1] << endl;
    }

    return 0;
}
```

区间合并

```
// 将所有存在交集的区间合并
void merge(vector<PII> &segs)
{
```

```
    vector<PII> res;

    sort(segs.begin(), segs.end());

    int st = -2e9, ed = -2e9;
    for (auto seg : segs)
        if (ed < seg.first)
        {
            if (st != -2e9) res.push_back({st, ed});
            st = seg.first, ed = seg.second;
        }
        else ed = max(ed, seg.second);

    if (st != -2e9) res.push_back({st, ed});

    segs = res;
}
```

二、数据结构

单链表

```
const int N=100010;

int head,e[N],ne[N],idx;
//初始化
void init(){
    head=-1;
    idx=0;
}
//在链表头部添加节点
void add_to_head(int x){
    e[idx]=x,ne[idx]=head,head=idx++;
}
//在位置k添加节点x
void add(int k,int x){
    e[idx]=x,ne[idx]=ne[k],ne[k]=idx++;
}
//删除位置k的节点
void remove(int k){
    ne[k]=ne[ne[k]];
}
```

应用

```
int main(){
    int m;
    init();
    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;
            if(!k)head=ne[head];
            remove(k-1);
        }else {
            cin>>k>>x;
            add(k-1,x);
        }
    }
    for(int i=head;i!=-1;i=ne[i])cout<<e[i]<<' ';
    cout<<endl;
    return 0;
}
```

双链表

```
const int N=100010;

int e[N],l[N],r[N],idx;

//初始化
void init(){
    l[1]=0;
    r[0]=1;
    idx=2;
}

//在节点a的右边插入一个数x
void insert(int a,int x){
    e[idx]=x;
    l[idx]=a,r[idx]=r[a];
    l[r[a]]=idx,r[a]=idx++;
}

//删除节点a
void remove(int a){
    l[r[a]]=l[a];
    r[l[a]]=r[a];
}
```

应用

```
int main(){
    int m;
    cin>>m;
    init();
    while(m--){
        string op;
        cin>>op;
        int k,x;
        if(op=="L"){//在最左端插入数x
            cin>>x;
            insert(0,x);
```

```
        }else if(op=="R"){//在最右端插入数x
            cin>>x;
            insert(l[1],x);
        }else if(op=="D"){//删除第k个插入的数
            cin>>k;
            remove(k+1);
        }else if(op=="IL"){//在第k个位置的左侧插入一个数
            cin>>k>>x;
            insert(l[k+1],x);
        }else if(op=="LR"){//在第k个位置的右侧插入一个数
            cin>>k>>x;
            insert(k+1,x);
        }
    }
    for(int i=r[0];i!=1;i=r[i])printf("%d ",e[i]);
    cout<<endl;
    return 0;
}
```

栈

```
// tt表示栈顶
int stk[N], tt = 0;
// 向栈顶插入一个数
stk[ ++ tt] = x;
// 从栈顶弹出一个数
tt -- ;
// 栈顶的值
stk[tt];
// 判断栈是否为空，如果 tt > 0，则表示不为空
if (tt > 0)
{

}
```

应用

```
const int N=100010;
int stk[N],tt;

int main(){
    int m;
    cin>>m;
    while(m--){
        string op;
        int x;
        cin>>op;
        if(op=="push"){
            cin>>x;
            stk[tt++]=x;
        }else if(op=="pop"){
            tt--;
        }else if(op=="query"){
            cout<<stk[tt-1]<<endl;
        }else{
```

```
            if(!tt)cout<<"YES"<<endl;
            else cout<<"NO"<<endl;
        }
    }
    return 0;
}
```

队列

普通队列

```
// hh 表示队头，tt表示队尾
int q[N], hh = 0, tt = -1;

// 向队尾插入一个数
q[ ++ tt] = x;

// 从队头弹出一个数
hh ++ ;

// 队头的值
q[hh];

// 判断队列是否为空，如果 hh <= tt，则表示不为空
if (hh <= tt)
{

}
```

应用

```
int const N=100010;

int que[N],hh,tt=-1;

int main(){
    int m;
    cin>>m;
    while(m--){
        string op;
        int x;
        cin>>op;
        if(op=="push"){
            cin>>x;
            que[++tt]=x;
        }else if(op=="query"){
            cout<<que[hh]<<endl;
        }else if(op=="pop"){
            hh++;
        }else{
            if(hh>tt)cout<<"YES"<<endl;
            else cout<<"NO"<<endl;
        }
    }
    return 0;
```

```
}
```

循环队列

```
// hh 表示队头，tt表示队尾的后一个位置
int q[N], hh = 0, tt = 0;

// 向队尾插入一个数
q[tt ++ ] = x;
if (tt == N) tt = 0;

// 从队头弹出一个数
hh ++ ;
if (hh == N) hh = 0;

// 队头的值
q[hh];

// 判断队列是否为空，如果hh != tt，则表示不为空
if (hh != tt)
{

}
```

单调栈

```
常见模型：找出每个数左边离它最近的比它大/小的数
int tt = 0;
for (int i = 1; i <= n; i ++ )
{
    while (tt && check(stk[tt], i)) tt -- ;
    stk[ ++ tt] = i;
}
```

应用

```
找出每个数左边离它最近的比它大/小的数
stack<int> stk;
int main(){
    int n;
    cin >> n;
    stk.push(-1);
    for (int i = 0; i < n; i ++){
        int x;
        cin >> x;
        while (stk.size() && stk.top() >= x) stk.pop();
        cout << stk.top() << " ";
        stk.push(x);
    }
    return 0;
}
```

单调队列

```
常见模型：找出滑动窗口中的最大值/最小值
int hh = 0, tt = -1;
for (int i = 0; i < n; i ++ )
{
    while (hh <= tt && check_out(q[hh])) hh ++ ;  // 判断队头是否滑出窗口
    while (hh <= tt && check(q[tt], i)) tt -- ;
    q[ ++ tt] = i;
}
```

```
const int N = 1000010;
int a[N];
int main()
{
    int n, k;
    cin >> n >> k;
    for (int i = 1; i <= n; i ++ ) cin >> a[i];//读入数据
    deque<int> q;
    for(int i = 1; i <= n; i++)
    {
        while(q.size() && q.back() > a[i]) //新进入窗口的值小于队尾元素，则队尾出队列
            q.pop_back();
        q.push_back(a[i]);//将新进入的元素入队
        if(i - k >= 1 && q.front() == a[i - k])//若队头是否滑出了窗口，队头出队
            q.pop_front();
        if(i >= k)//当窗口形成，输出队头对应的值
            cout << q.front() <<" ";
    }
    q.clear();
    cout << endl;

    //最大值亦然
    for(int i = 1; i <= n; i++)
    {
        while(q.size() && q.back() < a[i]) q.pop_back();
        q.push_back(a[i]);
        if(i - k >= 1 && a[i - k] == q.front()) q.pop_front();
        if(i >= k) cout << q.front() << " ";

    }
}
```

KMP字符串匹配

下标从1开始的kmp算法

```
const int N = 100010, M = 1000010;
int n, m;
int ne[N];
char s[M], p[N];
int main()
{
    cin >> n >> p + 1 >> m >> s + 1;
    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;
    }//处理ne数组
    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)
        {
            printf("%d ", i - n);
            j = ne[j];
        }
    }//匹配算法
    return 0;
}
```

```
// s[]是长文本，p[]是模式串，n是s的长度，m是p的长度
求模式串的Next数组：
for (int i = 2, j = 0; i <= m; i ++ )
{
    while (j && p[i] != p[j + 1]) j = ne[j];
    if (p[i] == p[j + 1]) j ++ ;
    ne[i] = j;
}

// 匹配
for (int i = 1, j = 0; i <= n; i ++ )
{
    while (j && s[i] != p[j + 1]) j = ne[j];
    if (s[i] == p[j + 1]) j ++ ;
    if (j == m)
    {
        j = ne[j];
        // 匹配成功后的逻辑
    }
}
```

下标从0开始的kmp算法

```
const int N = 1000010;

int n, m;
char s[N], p[N];
int ne[N];

int main()
{
    cin >> m >> p >> n >> s;
    ne[0] = -1;
    for (int i = 1, j = -1; i < m; i ++ )
    {
        while (j >= 0 && p[j + 1] != p[i]) j = ne[j];
        if (p[j + 1] == p[i]) j ++ ;
        ne[i] = j;
    }
    for (int i = 0, j = -1; i < n; i ++ )
```

```
    {
        while (j != -1 && s[i] != p[j + 1]) j = ne[j];
        if (s[i] == p[j + 1]) j ++ ;
        if (j == m - 1)
        {
            cout << i - j << ' ';
            j = ne[j];
        }
    }
    return 0;
}
```

Trie树

Trie 树是一种多叉树的结构，每个节点保存一个字符，一条路径表示一个字符串。

相关链接：https://www.acwing.com/solution/content/27771/

```
int son[N][26], cnt[N], idx;
// 0号点既是根节点，又是空节点
// son[][]存储树中每个节点的子节点
// cnt[]存储以每个节点结尾的单词数量

// 插入一个字符串
void insert(char *str)
{
    int p = 0;
    for (int i = 0; str[i]; i ++ )
    {
        int u = str[i] - 'a';
        if (!son[p][u]) son[p][u] = ++ idx;
        p = son[p][u];
    }
    cnt[p] ++ ;
}

// 查询字符串出现的次数
int query(char *str)
{
    int p = 0;
    for (int i = 0; str[i]; i ++ )
    {
        int u = str[i] - 'a';
        if (!son[p][u]) return 0;
        p = son[p][u];
    }
    return cnt[p];
}
```

```
const int N = 100010;

int son[N][26], cnt[N], idx;
char str[N];

void insert(char *str)
{
```

```
    int p = 0;
    for (int i = 0; str[i]; i ++ )
    {
        int u = str[i] - 'a';
        if (!son[p][u]) son[p][u] = ++ idx;
        p = son[p][u];
    }
    cnt[p] ++ ;
}//插入

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

int main()
{
    int n;
    scanf("%d", &n);
    while (n -- )
    {
        char op[2];
        scanf("%s%s", op, str);
        if (*op == 'I') insert(str);
        else printf("%d\n", query(str));
    }

    return 0;
}
```

并查集

```
(1)朴素并查集：

    int p[N]; //存储每个点的祖宗节点

    // 返回x的祖宗节点
    int find(int x)
    {
        if (p[x] != x) p[x] = find(p[x]);
        return p[x];
    }

    // 初始化，假定节点编号是1~n
    for (int i = 1; i <= n; i ++ ) p[i] = i;

    // 合并a和b所在的两个集合：
    p[find(a)] = find(b);
```

```
(2)维护size的并查集：

    int p[N], size[N];
    //p[]存储每个点的祖宗节点, size[]只有祖宗节点的有意义，表示祖宗节点所在集合中的点的数量

    // 返回x的祖宗节点
    int find(int x)
    {
        if (p[x] != x) p[x] = find(p[x]);
        return p[x];
    }

    // 初始化，假定节点编号是1~n
    for (int i = 1; i <= n; i ++ )
    {
        p[i] = i;
        size[i] = 1;
    }

    // 合并a和b所在的两个集合：
    size[find(b)] += size[find(a)];
    p[find(a)] = find(b);


(3)维护到祖宗节点距离的并查集：

    int p[N], d[N];
    //p[]存储每个点的祖宗节点, d[x]存储x到p[x]的距离

    // 返回x的祖宗节点
    int find(int x)
    {
        if (p[x] != x)
        {
            int u = find(p[x]);
            d[x] += d[p[x]];
            p[x] = u;
        }
        return p[x];
    }

    // 初始化，假定节点编号是1~n
    for (int i = 1; i <= n; i ++ )
    {
        p[i] = i;
        d[i] = 0;
    }

    // 合并a和b所在的两个集合：
    p[find(a)] = find(b);
    d[find(a)] = distance; // 根据具体问题，初始化find(a)的偏移量
```

应用

```
const int N=100010;
int p[N],n,m;
```

```
int find(int x){//找到祖宗节点+路径压缩
    if(p[x]!=x)p[x]=find(p[x]);
    return p[x];
}

int main(){
    scanf("%d%d",&n,&m);
    for(int i=1;i<=n;i++)p[i]=i;
    while(m--){
        char op[2];
        int a,b;
        scanf("%s%d%d",op,&a,&b);
        if(op[0]=='M')p[find(a)]=find(b);
        else {
            if(find(a)==find(b))puts("Yes");
            else puts("No");
        }
    }
    return 0;
}
```

堆

```
// h[N]存储堆中的值, h[1]是堆顶，x的左儿子是2x, 右儿子是2x + 1
// ph[k]存储第k个插入的点在堆中的位置
// hp[k]存储堆中下标是k的点是第几个插入的
int h[N], ph[N], hp[N], size;

// 交换两个点，及其映射关系
void heap_swap(int a, int b)
{
    swap(ph[hp[a]],ph[hp[b]]);
    swap(hp[a], hp[b]);
    swap(h[a], h[b]);
}

void down(int u)
{
    int t = u;
    if (u * 2 <= size && h[u * 2] < h[t]) t = u * 2;
    if (u * 2 + 1 <= size && h[u * 2 + 1] < h[t]) t = u * 2 + 1;
    if (u != t)
    {
        heap_swap(u, t);
        down(t);
    }
}

void up(int u)
{
    while (u / 2 && h[u] < h[u / 2])
    {
        heap_swap(u, u / 2);
        u >>= 1;
    }
```

```
}

// O(n)建堆
for (int i = n / 2; i; i -- ) down(i);
```

应用：堆排序

```
const int N=100010;
int heap[N],cnt;

void down(int u){
    int t=u;
    if(u*2<=cnt&&heap[u*2]<=heap[t])t=u*2;
    if(u*2+1<=cnt&&heap[u*2+1]<=heap[t])t=u*2+1;
    if(t!=u){
        swap(heap[t],heap[u]);
        down(t);
    }
}//down操作

void up(int u){
    while(u/2&&heap[u/2]>heap[u]){
        swap(heap[u/2],heap[u]);
        u>>=1;
    }
}//up操作

int main(){
    int n,m;
    scanf("%d%d",&n,&m);
    for(int i=1;i<=n;i++)scanf("%d",&heap[i]);
    cnt=n;
    for(int i=n/2;i;i--)down(i);
    while(m--){
        printf("%d ",heap[1]);
        heap[1]=heap[cnt--];
        down(1);
    }
    return 0;
}
```

一般hash

```
(1) 拉链法
    int h[N], e[N], ne[N], idx;

    // 向哈希表中插入一个数
    void insert(int x)
    {
        int k = (x % N + N) % N;
        e[idx] = x;
        ne[idx] = h[k];
        h[k] = idx ++ ;
    }
```

```
    // 在哈希表中查询某个数是否存在
    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;
    }

(2) 开放寻址法
    int h[N];

    // 如果x在哈希表中，返回x的下标；如果x不在哈希表中，返回x应该插入的位置
    int find(int x)
    {
        int t = (x % N + N) % N;
        while (h[t] != null && h[t] != x)
        {
            t ++ ;
            if (t == N) t = 0;
        }
        return t;
    }
```

字符串哈希

```
核心思想：将字符串看成P进制数，P的经验值是131或13331，取这两个值的冲突概率低
小技巧：取模的数用2^64，这样直接用unsigned long long存储，溢出的结果就是取模的结果

typedef unsigned long long ULL;
ULL h[N], p[N]; // h[k]存储字符串前k个字母的哈希值, p[k]存储 P^k mod 2^64

// 初始化
p[0] = 1;
for (int i = 1; i <= n; i ++ )
{
    h[i] = h[i - 1] * P + str[i];
    p[i] = p[i - 1] * P;
}

// 计算子串 str[l ~ r] 的哈希值
ULL get(int l, int r)
{
    return h[r] - h[l - 1] * p[r - l + 1];
}
```

STL

```
vector, 变长数组，倍增的思想
    size()  返回元素个数
    empty()  返回是否为空
    clear()  清空
    front()/back()
```

```
    push_back()/pop_back()
    begin()/end()
    []
    支持比较运算，按字典序

pair<int, int>
    first, 第一个元素
    second, 第二个元素
    支持比较运算，以first为第一关键字，以second为第二关键字（字典序）

string，字符串
    size()/length()  返回字符串长度
    empty()
    clear()
    substr(起始下标，(子串长度))  返回子串
    c_str()  返回字符串所在字符数组的起始地址

queue, 队列
    size()
    empty()
    push()  向队尾插入一个元素
    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，默认是大根堆
    size()
    empty()
    push()  插入一个元素
    top()  返回堆顶元素
    pop()  弹出堆顶元素
    定义成小根堆的方式：priority_queue<int, vector<int>, greater<int>> q;

stack, 栈
    size()
    empty()
    push()  向栈顶插入一个元素
    top()  返回栈顶元素
    pop()  弹出栈顶元素

deque, 双端队列
    size()
    empty()
    clear()
    front()/back()
    push_back()/pop_back()
    push_front()/pop_front()
    begin()/end()
    []

set, map, multiset, multimap, 基于平衡二叉树（红黑树），动态维护有序序列
    size()
    empty()
    clear()
    begin()/end()
    ++, -- 返回前驱和后继，时间复杂度 O(logn)

    set/multiset
```

```
        insert()  插入一个数
        find()  查找一个数
        count()  返回某一个数的个数
        erase()
            (1) 输入是一个数x，删除所有x   O(k + logn)
            (2) 输入一个迭代器，删除这个迭代器
        lower_bound()/upper_bound()
            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
    map/multimap
        insert()  插入的数是一个pair
        erase()  输入的参数是pair或者迭代器
        find()
        []  注意multimap不支持此操作。 时间复杂度是 O(logn)
        lower_bound()/upper_bound()

unordered_set, unordered_map, unordered_multiset, unordered_multimap, 哈希表
    和上面类似，增删改查的时间复杂度是 O(1)
    不支持 lower_bound()/upper_bound()， 迭代器的++，--

bitset, 圧位
    bitset<10000> s;
    ~, &, |, ^
    >>, <<
    ==, !=
    []

    count()  返回有多少个1

    any()  判断是否至少有一个1
    none()  判断是否全为0

    set()  把所有位置成1
    set(k, v)  将第k位变成v
    reset()  把所有位变成0
    flip()  等价于~
    flip(k) 把第k位取反
```

三、搜索与图论

树与图的存储

树是一种特殊的图，与图的存储方式相同。
对于无向图中的边ab，存储两条有向边a->b, b->a。
因此我们可以只考虑有向图的存储。

邻接矩阵

邻接矩阵：g[a][b] 存储边a->b的距离

邻接表

```
// 对于每个点k，开一个单链表，存储k所有可以走到的点。h[k]存储这个单链表的头结点
int h[N], e[N], ne[N], idx;
// 添加一条边a->b
void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
}
// 初始化
idx = 0;
memset(h, -1, sizeof h);
```

树与图的遍历

时间复杂度O(n+m)，n表示点数，m表示边数

深度优先遍历

```
int dfs(int u)
{
    st[u] = true; // st[u] 表示点u已经被遍历过

    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j]) dfs(j);
    }
}
```

应用：数字全排列

```
#include <iostream>

using namespace std;

int res[10],b[10],n;

void dfs(int k){
    if(k==n){//k==n则输出n个数字
        for(int i=0;i<n;i++)printf("%d ",res[i]);
        cout<<endl;
    }
    for(int i=1;i<=n;i++){
        if(!b[i]){//判断是否被用过
            res[k]=i;//当前k位存入位置
            b[i]=1;//表示被占用
            dfs(k+1);
            b[i]=0;//恢复现场
        }
    }
}

int main(){
    cin>>n;
    dfs(0);//从0开始枚举
    return 0;
```

```
}
```

宽度优先遍历

```
queue<int> q;
st[1] = true; // 表示1号点已经被遍历过
q.push(1);

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])
        {
            st[j] = true; // 表示点j已经被遍历过
            q.push(j);
        }
    }
}
```

应用：走迷宫

```
typedef pair<int,int> PII;//声明pair时候必须要在代码前面写上using namespace std;

const int N=110;
int g[N][N],f[N][N],n,m;

int bfs(int x,int y){
    queue<PII> que;
    que.push({x,y});
    int dx[4]={0,1,0,-1},dy[4]={1,0,-1,0};
    while(!que.empty()){
        PII t=que.front();
        que.pop();
        g[t.first][t.second]=1;
        for(int i=0;i<4;i++){
            int a=t.first+dx[i],b=t.second+dy[i];
            if(a>=0&&b>=0&&a<n&&b<m&&!g[a][b]){
                g[a][b]=1;
                f[a][b]=f[t.first][t.second]+1;
                que.push({a,b});
            }
        }
    }
    return f[n-1][m-1];
}

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

```
    cout<<bfs(0,0)<<endl;
    return 0;
}
```

应用：八数码

```
using namespace std;

int bfs(string state) {
    queue<string> q;
    unordered_map<string, int> d;

    int dx[4] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};
    string ed = "12345678x";
    q.push(state);
    d[state] = 0;

    while (q.size()) {
        auto t = q.front();
        q.pop();
        if (t == ed)//等于结果就输出步数
            return d[t];
        int distance = d[t];
        int k = t.find('x');//寻找x
        int x = k / 3, y = k % 3;//计算下标
        for (int i = 0; i < 4; i ++ ) {
            int a = x + dx[i], b = y + dy[i];
            if (a >= 0 && a < 3 && b >= 0 && b < 3) {
                swap(t[a * 3 + b], t[k]);//交换
                if (!d.count(t)) {//不存在就入队
                    d[t] = distance + 1;
                    q.push(t);
                }
                swap(t[a * 3 + b], t[k]);//还原
            }
        }
    }
    return -1;
}

int main() {
    char s[2];
    string state;
    for (int i = 0; i < 9; i ++ ) {
        cin >> s;
        state += *s;
    }
    cout<<bfs(state)<<endl;
    return 0;
}
```

拓扑排序

啥事拓扑排序?

一个有向图，如果图中有入度为 0 的点，就把这个点删掉，同时也删掉这个点所连的边。

一直进行上面出处理，如果所有点都能被删掉，则这个图可以进行拓扑排序。

纯净版

```
bool topsort()
{
    int hh = 0, tt = -1;

    // d[i] 存储点i的入度
    for (int i = 1; i <= n; i ++ )
        if (!d[i])
            q[ ++ tt] = i;

    while (hh <= tt)
    {
        int t = q[hh ++ ];

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (-- d[j] == 0)
                q[ ++ tt] = j;
        }
    }

    // 如果所有点都入队了，说明存在拓扑序列；否则不存在拓扑序列。
    return tt == n - 1;
}
```

解说版

```
using namespace std;
const int N = 100010;
int e[N], ne[N], idx; //邻接表存储图
int h[N];//邻接表的每个头链表
int q[N], hh = 0, tt = -1; //队列保存入度为0的点，也就是能够输出的点
int n, m; //保存图的点数和边数
int d[N];//保存各个点的入度

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

void topsort() {
    for (int i = 1; i <= n; i++) {//遍历一遍顶点的入度。
        if (!d[i])//如果入度为0，则可以入队列
            q[++tt] = i;
    }
    while (tt >= hh) { //循环处理队列中点的
        int a = q[hh++];
        for (int i = h[a]; i != -1; i = ne[i]) {
            int b = e[i]; //a 有一条边指向b
            d[b]--;//删除边后，b的入度减1
            if (!d[b])//如果b的入度减为 0,则 b 可以输出，入队列
                q[++tt] = b;
        }
```

```
    }
    if (tt == n - 1) {//如果队列中的点的个数与图中点的个数相同，则可以进行拓扑排序
        for (int i = 0; i < n; i++)//队列中保存了所有入度为0的点，依次输出
            printf("%d ", q[i]);
    } else//如果队列中的点的个数与图中点的个数不相同，则可以进行拓扑排序
        cout << -1;
}

int main() {
    cin >> n >> m; //保存点的个数和边的个数
    memset(h, -1, sizeof h); //初始化领接矩阵
    while (m--) { //依次读入边
        int a, b;
        cin >> a >> b;
        d[b]++;//顶点b的入度+1
        add(a, b); //添加到邻接矩阵
    }
    topsort();//进行拓扑排序
    return 0;
}
```

Dijkstra算法

朴素版

时间复杂是$O(n^2+m)$，n表示点数，m表示边数

```
int g[N][N];  // 存储每条边
int dist[N];  // 存储1号点到每个点的最短距离
bool st[N];   // 存储每个点的最短路是否已经确定

// 求1号点到n号点的最短路，如果不存在则返回-1
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    for (int i = 0; i < n - 1; i ++ )
    {
        int t = -1;     // 在还未确定最短路的点中，寻找距离最小的点
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        // 用t更新其他点的距离
        for (int j = 1; j <= n; j ++ )
            dist[j] = min(dist[j], dist[t] + g[t][j]);

        st[t] = true;
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

应用

```
const int N = 510, M = 100010;

int h[N], e[M], ne[M], w[M], idx;//邻接表存储图
int state[N];//state 记录是否找到了源点到该节点的最短距离
int dist[N];//dist 数组保存源点到其余各个节点的距离
int n, m;//图的节点个数和边数

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

void Dijkstra()
{
    memset(dist, 0x3f, sizeof(dist));//dist 数组的各个元素为无穷大
    dist[1] = 0;//源点到源点的距离为置为 0
    for (int i = 0; i < n; i++)
    {
        int t = -1;
        for (int j = 1; j <= n; j++)//遍历 dist 数组，找到没有确定最短路径的节点中距离
源点最近的点t
        {
            if (!state[j] && (t == -1 || dist[j] < dist[t]))
                t = j;
        }

        state[t] = 1;//state[i] 置为 1。

        for (int j = h[t]; j != -1; j = ne[j])//遍历 t 所有可以到达的节点 i
        {
            int i = e[j];
            dist[i] = min(dist[i], dist[t] + w[j]);//更新 dist[j]
        }


    }
}

int main()
{
    memset(h, -1, sizeof(h));//邻接表初始化

    cin >> n >> m;
    while (m--)//读入 m 条边
    {
        int a, b, w;
        cin >> a >> b >> w;
        add(a, b, w);
    }

    Dijkstra();
    if (dist[n] != 0x3f3f3f3f)//如果dist[n]被更新了，则存在路径
        cout << dist[n];
    else
        cout << "-1";
}
```

堆优化版

时间复杂度$O(mlogn)$，n表示点数，m表示边数

```
typedef pair<int, int> PII;

int n;      // 点的数量
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N];        // 存储所有点到1号点的距离
bool st[N];     // 存储每个点的最短距离是否已确定

// 求1号点到n号点的最短距离，如果不存在，则返回-1
int dijkstra(){
    memset(dist,0x3f,sizeof dist);//距离初始化为无穷大
    dist[1]=0;//1->1的节点距离为0
    priority_queue<PII,vector<PII>,greater<PII>> heap;//小根堆
    heap.push({0,1});//插入距离和节点编号

    while(heap.size()){
        auto t=heap.top();//取距离源点最近的点
        heap.pop();

        int ver=t.second,distance=t.first;//ver：节点编号，distance源点距离ver
        if(st[ver])continue;//如果距离已经确定，则跳过该点
        st[ver]=true;
        for(int i=h[ver];i!=-1;i=ne[i])//更新ver所指向的节点距离
        {
            int j=e[i];
            if(dist[j]>dist[ver]+w[i]){
                dist[j]=dist[ver]+w[i];
                heap.push({dist[j],j});//距离变小，则入堆
            }
        }
    }
    if(dist[n]==0x3f3f3f3f)return -1;
    return dist[n];
}
```

Bellman-Ford算法

时间复杂度$O(nm)$，n表示点数，m表示边数

注意在模板题中需要对下面的模板稍作修改，加上备份数组，详情见模板题。

```
int n, m;       // n表示点数，m表示边数
int dist[N];        // dist[x]存储1到x的最短路距离

struct Edge     // 边，a表示出点，b表示入点，w表示边的权重
{
    int a, b, w;
}edges[M];

// 求1到n的最短路距离，如果无法从1走到n，则返回-1。
int bellman_ford()
{
    memset(dist, 0x3f, sizeof dist);
```

```
    dist[1] = 0;

    // 如果第n次迭代仍然会松弛三角不等式，就说明存在一条长度是n+1的最短路径，由抽屉原理，路径中至少存在两个相同的点，说明图中存在负权回路。
    for (int i = 0; i < n; i ++ )
    {
        for (int j = 0; j < m; j ++ )
        {
            int a = edges[j].a, b = edges[j].b, w = edges[j].w;
            if (dist[b] > dist[a] + w)
                dist[b] = dist[a] + w;
        }
    }

    if (dist[n] > 0x3f3f3f3f / 2) return -1;
    return dist[n];
}
```

应用

```
int n,m,k;
const int N=512,M=10012;
struct Edge{
    int a,b,w;
}e[M];
int dist[N];
int back[N];
void bellman_ford(){
    memset(dist,0x3f,sizeof dist);
    dist[1]=0;
    for(int i=0;i<k;i++){
        memcpy(back,dist,sizeof dist);
        for(int j=0;j<m;j++){
            int a=e[j].a,b=e[j].b,c=e[j].w;
            dist[b]=min(dist[b],back[a]+c);
        }
    }
}

int main(){
    scanf("%d%d%d",&n,&m,&k);
    for(int i=0;i<m;i++){
        int a,b,w;
        scanf("%d%d%d",&a,&b,&w);
        e[i]={a,b,w};
    }
    bellman_ford();
    if(dist[n]>0x3f3f3f3f/2)cout<<"impossible"<<endl;
    else cout<<dist[n]<<endl;
    return 0;
}
```

SPFA算法（队列优化的Bellman-Ford算法）

时间复杂度平均情况下$O(m)$，最坏情况下$O(nm)$，n表示点数，m表示边数

模板

```
int n;      // 总点数
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N];        // 存储每个点到1号点的最短距离
bool st[N];     // 存储每个点是否在队列中

// 求1号点到n号点的最短路距离，如果从1号点无法走到n号点则返回-1
int spfa()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    queue<int> q;
    q.push(1);
    st[1] = true;

    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])
            {
                dist[j] = dist[t] + w[i];
                if (!st[j])     // 如果队列中已存在j，则不需要将j重复插入
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

应用

```
const int N = 1e6 + 10;

int n, m;//节点数量和边数
int h[N], w[N], e[N], ne[N], idx;//邻接矩阵存储图
int dist[N];//存储距离
bool st[N];//存储状态

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

```
int spfa()
{
    memset(dist, 0x3f, sizeof dist);//距离初始化为无穷大
    dist[1] = 0;//初始化1到1的距离为0
    queue<int> que;//队列
    que.push(1);//1入队

    while (que.size())//判断是否存在
    {
        int t=que.front();
        que.pop();//获取第一个并出队
        st[t]=false;//第一个取消占用
        for(int i=h[t];i!=-1;i=ne[i]){//遍历第一个可以到达的结点
            int j=e[i];
            if(dist[j]>dist[t]+w[i]){//1号点可到达的节点距离是否大于上次的距离距离加上当前的距离
                dist[j]=dist[t]+w[i];//赋值给可到达的节点
                if(!st[j]){//如果可到达的节点未被占用
                    que.push(j);//则入队
                    st[j]=true;//占用
                }
            }
        }
    }

    return dist[n];
}

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

    memset(h, -1, sizeof h);
    while (m -- )
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c);
    }

    int t=spfa();
    if(t==0x3f3f3f3f)cout<<"impossible"<<endl;
    else printf("%d\n",t);

    return 0;
}
```

spfa判断图中是否存在负权

```
int n;      // 总点数
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N], cnt[N];        // dist[x]存储1号点到x的最短距离，cnt[x]存储1到x的最短路中经过的点数
bool st[N];     // 存储每个点是否在队列中

// 如果存在负环，则返回true，否则返回false。
bool spfa()
```

```
{
    // 不需要初始化dist数组
    // 原理：如果某条最短路径上有n个点（除了自己），那么加上自己之后一共有n+1个点，由抽屉原理一定有两个点相同，所以存在环。

    queue<int> q;
    for (int i = 1; i <= n; i ++ )
    {
        q.push(i);
        st[i] = true;
    }

    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])
            {
                dist[j] = dist[t] + w[i];
                cnt[j] = cnt[t] + 1;
                if (cnt[j] >= n) return true;       // 如果从1号点到x的最短路中包含至少n个点（不包括自己），则说明存在环
                if (!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    return false;
}
```

floyd算法

时间复杂度$O(n^3)$，n表示点数

```
初始化：
    for (int i = 1; i <= n; i ++ )
        for (int j = 1; j <= n; j ++ )
            if (i == j) d[i][j] = 0;
            else d[i][j] = INF;

// 算法结束后，d[a][b]表示a到b的最短距离
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]);
}
```

prim算法

时间复杂度是$O(n^2+m)$，n表示点数，m表示边数

```
int n;      // n表示点数
int g[N][N];        // 邻接矩阵，存储所有边
int dist[N];        // 存储其他点到当前最小生成树的距离
bool st[N];     // 存储每个点是否已经在生成树中


// 如果图不连通，则返回INF(值是0x3f3f3f3f), 否则返回最小生成树的树边权重之和
int prim()
{
    memset(dist, 0x3f, sizeof dist);

    int res = 0;
    for (int i = 0; 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 (i && dist[t] == INF) return INF;

        if (i) res += dist[t];
        st[t] = true;

        for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);
    }

    return res;
}
```

应用

```
const int N = 510, INF = 0x3f3f3f3f;

int n, m;
int g[N][N];
int dist[N];
bool st[N];

int prim()
{
    memset(dist, 0x3f, sizeof dist);

    int res = 0;
    for (int i = 0; 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 (i && dist[t] == INF) return INF;

        if (i) res += dist[t];
        st[t] = true;

        for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);
    }

    return res;
}


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

    memset(g, 0x3f, sizeof g);

    while (m -- )
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        g[a][b] = g[b][a] = min(g[a][b], c);
    }

    int t = prim();

    if (t == INF) puts("impossible");
    else printf("%d\n", t);

    return 0;
}
```

Kruskal算法

时间复杂度$O(mlogm)$，n表示点数，m表示边数

```
int n, m;       // n是点数，m是边数
int p[N];       // 并查集的父节点数组

struct Edge     // 存储边
{
    int a, b, w;
    bool operator< (const Edge &W)const
    {
        return w < W.w;
    }
}edges[M];

int find(int x)     // 并查集核心操作
{
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}
```

```
int kruskal()
{
    sort(edges, edges + m);

    for (int i = 1; i <= n; i ++ ) p[i] = i;    // 初始化并查集

    int res = 0, cnt = 0;
    for (int i = 0; i < m; i ++ )
    {
        int a = edges[i].a, b = edges[i].b, w = edges[i].w;

        a = find(a), b = find(b);
        if (a != b)     // 如果两个连通块不连通，则将这两个连通块合并
        {
            p[a] = b;
            res += w;
            cnt ++ ;
        }
    }
    if (cnt < n - 1) return INF;
    return res;
}
```

应用

```
#include <cstring>
#include <iostream>
#include <algorithm>

using namespace std;

const int N = 100010, M = 200010, INF = 0x3f3f3f3f;

int n, m;
int p[N];

struct Edge
{
    int a, b, w;

    bool operator< (const Edge &W)const
    {
        return w < W.w;
    }
}edges[M];

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

int kruskal()
{
    sort(edges, edges + m);
```

```
    for (int i = 1; i <= n; i ++ ) p[i] = i;    // 初始化并查集

    int res = 0, cnt = 0;
    for (int i = 0; i < m; i ++ )
    {
        int a = edges[i].a, b = edges[i].b, w = edges[i].w;

        a = find(a), b = find(b);
        if (a != b)
        {
            p[a] = b;
            res += w;
            cnt ++ ;
        }
    }

    if (cnt < n - 1) return INF;
    return res;
}

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

    for (int i = 0; i < m; i ++ )
    {
        int a, b, w;
        scanf("%d%d%d", &a, &b, &w);
        edges[i] = {a, b, w};
    }

    int t = kruskal();

    if (t == INF) puts("impossible");
    else printf("%d\n", t);

    return 0;
}
```

染色法判别二分图

时间复杂度是$O(n+m)$，n表示点数，m表示边数

```
int n;      // n表示点数
int h[N], e[M], ne[M], idx;     // 邻接表存储图
int color[N];       // 表示每个点的颜色，-1表示未染色，0表示白色，1表示黑色

// 参数：u表示当前节点，c表示当前点的颜色
bool dfs(int u, int c)
{
    color[u] = c;
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (color[j] == -1)
        {
```

```
            if (!dfs(j, !c)) return false;
        }
        else if (color[j] == c) return false;
    }

    return true;
}

bool check()
{
    memset(color, -1, sizeof color);
    bool flag = true;
    for (int i = 1; i <= n; i ++ )
        if (color[i] == -1)
            if (!dfs(i, 0))
            {
                flag = false;
                break;
            }
    return flag;
}
```

匈牙利算法

时间复杂度$O(nm)$，n表示点数，m表示边数

```
int n1, n2;     // n1表示第一个集合中的点数，n2表示第二个集合中的点数
int h[N], e[M], ne[M], idx;     // 邻接表存储所有边，匈牙利算法中只会用到从第一个集合指向第二个集合的边，所以这里只用存一个方向的边
int match[N];       // 存储第二个集合中的每个点当前匹配的第一个集合中的点是哪个
bool st[N];     // 表示第二个集合中的每个点是否已经被遍历过

bool find(int x)
{
    for (int i = h[x]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j])
        {
            st[j] = true;
            if (match[j] == 0 || find(match[j]))
            {
                match[j] = x;
                return true;
            }
        }
    }

    return false;
}

// 求最大匹配数，依次枚举第一个集合中的每个点能否匹配第二个集合中的点
int res = 0;
for (int i = 1; i <= n1; i ++ )
{
    memset(st, false, sizeof st);
```

```
    if (find(i)) res ++ ;
}
```

四、数学知识

试除法判定质数

```
bool is_prime(int x)
{
    if (x < 2) return false;
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0)
            return false;
    return true;
}
```

试除法分解质因数

```
void divide(int x)
{
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0)
        {
            int s = 0;
            while (x % i == 0) x /= i, s ++ ;
            cout << i << ' ' << s << endl;
        }
    if (x > 1) cout << x << ' ' << 1 << endl;
    cout << endl;
}
```

埃氏筛法求质数

```
int primes[N], cnt;     // primes[]存储所有素数
bool st[N];         // st[x]存储x是否被筛掉

void get_primes(int n)
{
    for (int i = 2; i <= n; i ++ )
    {
        if (st[i]) continue;
        primes[cnt ++ ] = i;
        for (int j = i + i; j <= n; j += i)
            st[j] = true;
    }
}
```

线性筛法求质数

```
int primes[N], cnt;     // primes[]存储所有素数
```

```
bool st[N];         // st[x]存储x是否被筛掉

void get_primes(int n)
{
    for (int i = 2; i <= n; i ++ )
    {
        if (!st[i]) primes[cnt ++ ] = i;
        for (int j = 0; primes[j] <= n / i; j ++ )
        {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0) break;
        }
    }
}
```

试除法求所有约数

```
vector<int> get_divisors(int x)
{
    vector<int> res;
    for (int i = 1; i <= x / i; i ++ )
        if (x % i == 0)
        {
            res.push_back(i);
            if (i != x / i) res.push_back(x / i);
        }
    sort(res.begin(), res.end());
    return res;
}
```

欧几里得算法

```
int gcd(int a, int b)
{
    return b ? gcd(b, a % b) : a;
}
```

求欧拉函数

```
int phi(int x)
{
    int res = x;
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0)
        {
            res = res / i * (i - 1);
            while (x % i == 0) x /= i;
        }
    if (x > 1) res = res / x * (x - 1);

    return res;
}
```

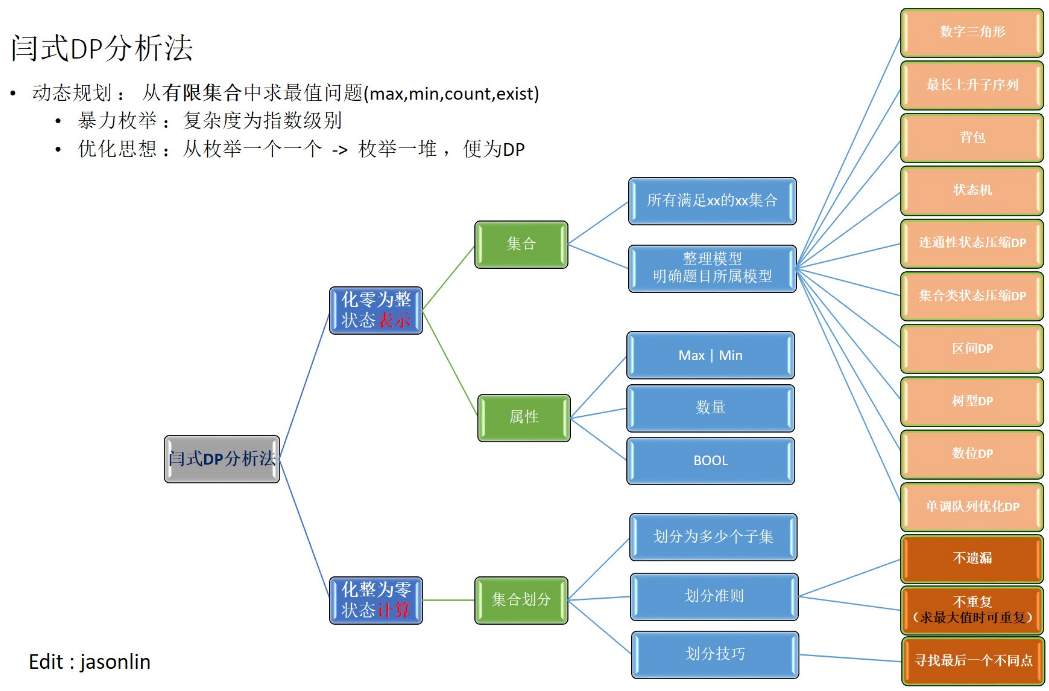
快速幂

```
求 m^k mod p，时间复杂度 O(logk)。
m为底数，k为幂
int qmi(int m, int k, int p)
{
    int res = 1 % p, t = m;
    while (k)
    {
        if (k&1) res = res * t % p;
        t = t * t % p;
        k >>= 1;
    }
    return res;
}
```

递推法求组合数

```
// c[a][b] 表示从a个苹果中选b个的方案数
for (int i = 0; i < N; i ++ )
    for (int j = 0; j <= i; j ++ )
        if (!j) c[i][j] = 1;
        else c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % mod;
```

五、动态规划



背包问题

01背包每件物品只能装一次
完全背包每件物品可以装无限次
多重背包每件物品只能装有限次（多次）
分组背包每组只能选择一件物品装入（01背包升级）

相关链接：https://zhuanlan.zhihu.com/p/166439661

01背包问题

	A	B	C	D	E	F	G
1		0	1	2	3	4	5
2		0	0	0	0	0	0
3	1，2	0	2	2	2	2	2
4	2，4	0	2	4	6	6	6
5	3，4	0	2	4	6	6	8
6	4，5	0	2	4	6	6	8
7							

```
const int N=1010;
int n,m;
int v[N],w[N];//v代表体积，w代表价值
int f[N][N];

int main(){
    cin>>n>>m;
    for(int i=1;i<=n;i++)cin>>v[i]>>w[i];
    for(int i=1;i<=n;i++)//i代表这n件物品
    {
        for(int j=1;j<=m;j++){//j代表背包容量
            if(v[i]>j)//如果v[i]的容量大于当前的背包容量则不装进行下一个
                f[i][j]=f[i-1][j];
            else f[i][j]=max(f[i-1][j],f[i-1][j-v[i]]+w[i]);//如果v[i]的容量小于当前背包容量则可以选择装与不装得到最大值
        }
    }

    cout<<f[n][m]<<endl;//输出最后的一个一定是最大的
    return 0;
}
```

01背包，使用滚动数组，倒序遍历

```
const int N=1010;
int n,m;
int v[N],w[N];//v代表体积，w代表价值
int dp[N];

int main(){
    cin>>n>>m;
    for(int i=1;i<=n;i++)//i代表这n件物品
    {
        cin>>v[i]>>w[i];//在线算法
        for(int j=m;j>=v[i];j--){//j代表背包容量，滚动数组必须倒序遍历
            dp[j]=max(dp[j],dp[j-v[i]]+w[i]);//滚动数组
        }
    }
    cout<<dp[m]<<endl;//输出最后的一个一定是最大的
    return 0;
```

```
}
```

状态转移方程：`dp[j]=max(dp[j],dp[j-v[i]]+w[i]);`

完全背包问题

```
int v[N],w[N];
int dp[N];
int main(){
    int n,m;
    cin>>n>>m;
    for(int i=1;i<=n;i++){//遍历物品
        cin>>v[i]>>w[i];//在线算法
        for(int j=v[i];j<=m;j++){//正序遍历背包容量
            dp[j]=max(dp[j],dp[j-v[i]]+w[i]);//滚动数组
        }
    }
    cout<<dp[m]<<endl;//输出答案
    return 0;
}
```

完全背包问题和01背包优化版的区别在于第二重循环的v[i]和m做交换

多重背包问题

```
int n,m;
int v[N],w[N],s[N];
int dp[N][N];

int main(){
    cin>>n>>m;
    for(int i=1;i<=n;i++)cin>>v[i]>>w[i]>>s[i];
    for(int i=1;i<=n;i++)//物品
        for(int j=0;j<=m;j++)//背包容量
            for(int k=0;k<=s[i]&&k*v[i]<=j;k++)
                dp[i][j]=max(dp[i][j],dp[i-1][j-v[i]*k]+w[i]*k);
    cout<<dp[n][m]<<endl;
    return 0;
}
```

状态转移方程：`dp[i][j]=max(dp[i][j],dp[i-1][j-v[i]*k]+w[i]*k);` k为第i个物品的个数

分组背包问题

分组背包每组只能选择一件物品装入

```
const int N=110;
int f[N];
int v[N][N],w[N][N],s[N];
int n,m,k;

int main(){
    cin>>n>>m;
    for(int i=0;i<n;i++){
        cin>>s[i];
```

```
        for(int j=0;j<s[i];j++){
            cin>>v[i][j]>>w[i][j];
        }
    }

    for(int i=0;i<n;i++){
        for(int j=m;j>=0;j--){
            for(int k=0;k<s[i];k++){    //for(int k=s[i];k>=1;k--)也可以
                if(j>=v[i][k])
                    f[j]=max(f[j],f[j-v[i][k]]+w[i][k]);
            }
        }
    }
    cout<<f[m]<<endl;
}
```

线性DP

数字三角形

```
const int N=510,INF=1e9;
int n;
int a[N][N];
int f[N][N];

int main(){
    scanf("%d",&n);
    for(int i=1;i<=n;i++){
        for(int j=1;j<=i;j++){
            scanf("%d",&a[i][j]);
        }
    }
    for(int i=0;i<=n;i++){
        for(int j=0;j<=i+1;j++){
            f[i][j]=-INF;
        }
    }
    f[1][1]=a[1][1];
    for(int i=2;i<=n;i++)
        for(int j=1;j<=i;j++)
            f[i][j]=max(f[i-1][j-1]+a[i][j],f[i-1][j]+a[i][j]);//状态转移方程
    int res=-INF;
    for(int i=1;i<=n;i++)res=max(res,f[n][i]);
    printf("%d",res);
    return 0;
}
```

最长上升子序列

```
const int N = 1010;

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

int main()
```

```
{
    scanf("%d", &n);
    for (int i = 1; i <= n; i ++ )scanf("%d",&a[i]);
    for (int i = 1; i <= n; i ++ ){
        f[i]=1;//只有a[i]一个数
        for (int j = 1; j <= n; j ++ )
            if(a[j]<a[i])
                f[i]=max(f[i],f[j]+1);
    }
    int res=0;
    for (int i = 1; i <= n; i ++ )res=max(res,f[i]);
    printf("%d\n",res);
    return 0;
}
```

状态转移方程：`if(a[j]<a[i])f[i]=max(f[i],f[j]+1);`

最长公共子序列

```
const int N=1010;
int n,m;
char a[N],b[N];
int f[N][N];

int main()
{
    cin>>n>>m>>a+1>>b+1;
    for (int i = 1; i <= n; i ++ ){
        for (int j = 1; j <= m; j ++ ){
            f[i][j]=max(f[i-1][j],f[i][j-1]);
            if(a[i]==b[j])f[i][j]=max(f[i][j],f[i-1][j-1]+1);
        }
    }
    cout<<f[n][m]<<endl;
    return 0;
}
```

状态转移方程：

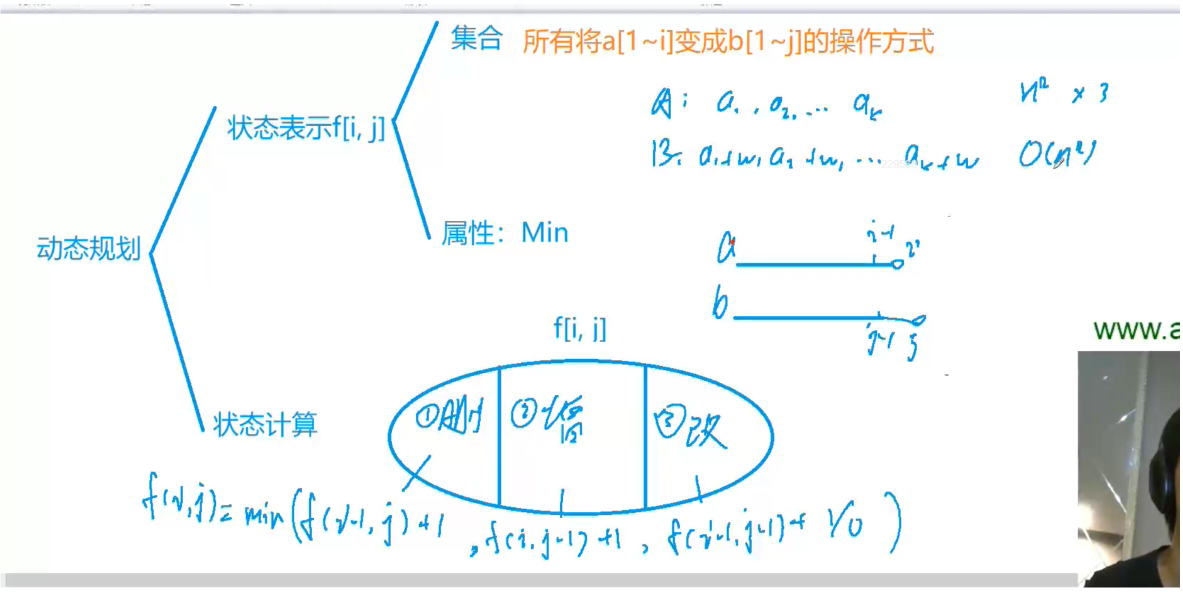
```
f[i][j]=max(f[i-1][j],f[i][j-1]);
if(a[i]==b[j])f[i][j]=max(f[i][j],f[i-1][j-1]+1);
```

最短编辑距离

给定两个字符串 A和 B，现在要将 A 经过若干操作变为 B，可进行的操作有：

1. 删除–将字符串 A中的某个字符删除。
2. 插入–在字符串 A 的某个位置插入某个字符。
3. 替换–将字符串 A中的某个字符替换为另一个字符。

现在请你求出，将 A变为 B 至少需要进行多少次操作。




```
const int N = 1010;
int n,m;
char a[N],b[N];
int f[N][N];

int main()
{
    scanf("%d%s", &n, a+1);
    scanf("%d%s", &m, b+1);

    for (int i = 0; i <= m; i ++ )f[0][i]=i;
    for (int i = 0; i <= n; i ++ )f[i][0]=i;
    for (int i = 1; i <= n; i ++ ){
        for (int j = 1; j <= m; j ++ ){
            f[i][j]=min(f[i-1][j]+1,f[i][j-1]+1);
            if(a[i]==b[j])f[i][j]=min(f[i][j],f[i-1][j-1]);
            else f[i][j]=min(f[i][j],f[i-1][j-1]+1);//状态转移方程
        }
    }
    printf("%d\n",f[n][m]);
    return 0;
}
```

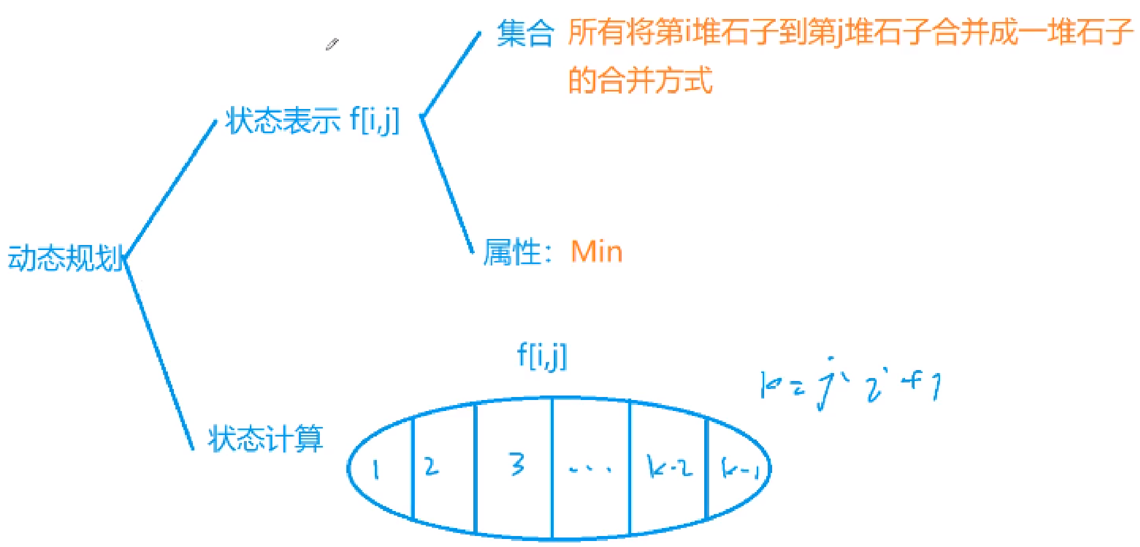
状态转移方程:

```
f[i][j]=min(f[i-1][j]+1,f[i][j-1]+1);
if(a[i]==b[j])f[i][j]=min(f[i][j],f[i-1][j-1]);
else f[i][j]=min(f[i][j],f[i-1][j-1]+1);//状态转移方程
```

区间DP

每堆石子有一定的质量，可以用一个整数来描述，现在要将这 N堆石子合并成为一堆。

每次只能合并相邻的两堆，合并的代价为这两堆石子的质量之和，合并后与这两堆石子相邻的石子将和新堆相邻，合并时由于选择的顺序不同，合并的总代价也不相同。




```
const int N = 310;

int n;
int s[N];
int f[N][N];//状态表示：集合f[l][r]为[l,r]区间；属性：所堆成的最小值
int main()
{
    scanf("%d", &n);
    for (int i = 1; i <= n; i ++ )scanf("%d",&s[i]);
    for (int i = 1; i <= n; i ++ )s[i]+=s[i-1];//前缀和用来求一段区间的和

    for (int len = 2; len <= n; len ++ )//区间长度为len//枚举长度
        for (int i = 1; i+len-1 <= n; i ++ ){//意思就是i在区间[1,n-len+1]中去//枚举
区间
            int l=i,r=i+len-1;//区间在[i,i+len-1]中间长度为len//设置l和r的区间
            f[l][r]=1e9;//初始化最大值
            for (int k = l; k < r; k ++ )//枚举分界点//不取r
                f[l][r]=min(f[l][r],f[l][k]+f[k+1][r]+s[r]-s[l-1]);//找到最小值状
态转移方程为f[l][k]+f[k+1][r]+s[r]-s[l-1];
        }
    printf("%d\n",f[1][n]);//输出区间[1,n]的最小值
    return 0;
}
```

状态转移方程 `找到最小值状态转移方程为f[l][k]+f[k+1][r]+s[r]-s[l-1];`

计数类DP

一个正整数 n 可以表示成若干个正整数之和，我们将这样的一种表示称为正整数 n 的一种划分。

现在给定一个正整数 n，请你求出 n共有多少种不同的划分方法。

完全背包写法

```
//完全背包的写法
#include <iostream>

using namespace std;
```

```
const int M=1e9+7;
int f[1010],n;

int main()
{
    cin>>n;

    f[0]=1;
    for (int i = 1; i <= n; i ++ )
        for (int j = i; j <= n; j ++ ){
            f[j]=(f[j-i]+f[j])%M;
        }
    cout<<f[n]<<endl;
    return 0;
}
```

状态转移方程：`f[j]=(f[j-i]+f[j])`

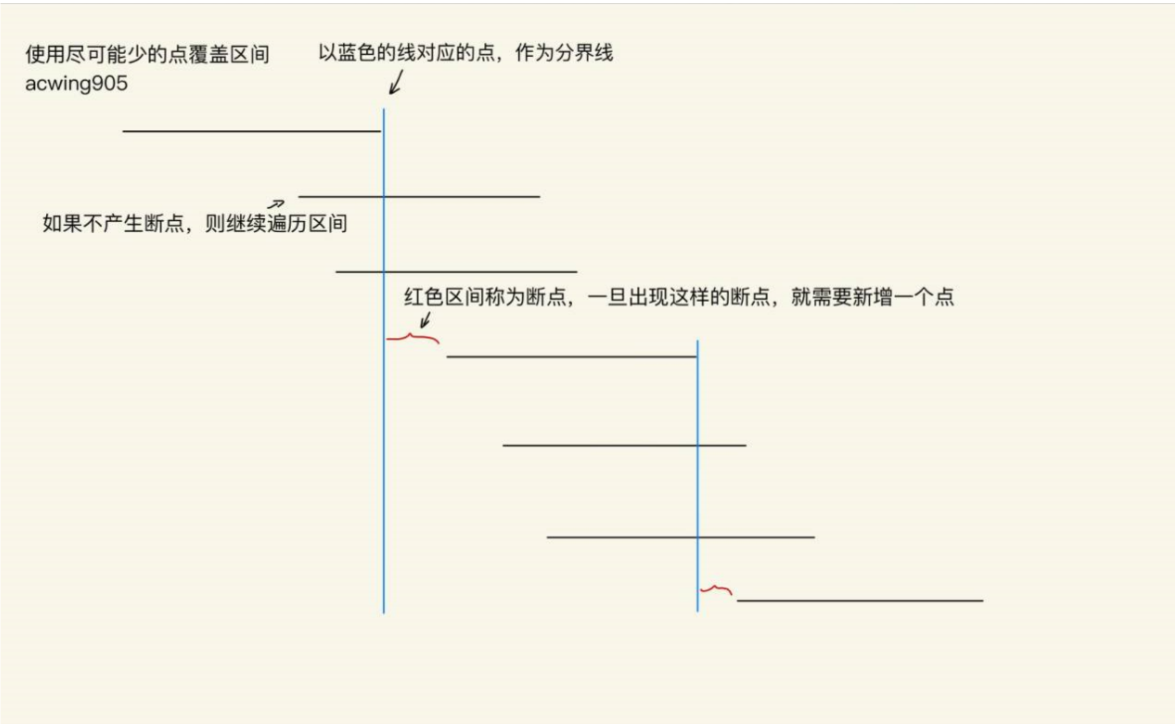
六、贪心

一个贪心算法总是做出当前最好的选择，也就是说，它期望通过局部最优选择从而得到全局最优的解决方案。--- **《算法导论》**

区间选点

给定 N个闭区间$[a_i, b_i]$，请你在数轴上选择尽量少的点，使得每个区间内至少包含一个选出的点。

输出选择的点的最小数量。




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

using namespace std;
```

```
const int N = 100010;

int n;
struct Range{
    int l,r;
    bool operator <(const Range& W)const{
        return r<W.r;
    }
}range[N];

int main()
{
    scanf("%d", &n);
    for (int i = 0; i < n; i ++ ){
        int l,r;
        scanf("%d%d", &l, &r);
        range[i]={l,r};
    }
    sort(range,range+n);
    int res=0,ed=-2e9;
    for (int i = 0; i < n; i ++ ){
        if(range[i].l>ed){
            res++;
            ed=range[i].r;
        }
    }
    printf("%d\n",res);
    return 0;
}
```

笔记作者QQ：`2468197060`

欢迎一起交流技术