

# 一、基础算法

## 快速排序算法模板

```
1 void quick_sort(int q[], int l, int r)
2 {
3     //递归的终止情况
4     if (l >= r) return;
5
6     //选取分界线。这里选数组中间那个数
7     int i = l - 1, j = r + 1, x = q[l + r >> 1];
8     //划分成左右两个部分
9     while (i < j)
10    {
11        do i ++ ; while (q[i] < x);
12        do j -- ; while (q[j] > x);
13        if (i < j) swap(q[i], q[j]);
14    }
15    //对左右部分排序
16    quick_sort(q, l, j), quick_sort(q, j + 1, r);
17 }
```

### 边界问题

因为边界问题只有这两种组合，不能随意搭配

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

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

## 归并排序算法模板

```
1 void merge_sort(int q[], int l, int r)
2 {
3     //递归的终止情况
4     if (l >= r) return;
5     //第一步：分成子问题
6     int mid = l + r >> 1;
7     //第二步：递归处理子问题
8     merge_sort(q, l, mid);
9     merge_sort(q, mid + 1, r);
10
11    //第三步：合并子问题
12    int k = 0, i = l, j = mid + 1;
```

```

13     while (i <= mid && j <= r)
14         if (q[i] <= q[j]) tmp[k ++ ] = q[i ++ ];
15         else tmp[k ++ ] = q[j ++ ];
16
17     while (i <= mid) tmp[k ++ ] = q[i ++ ];
18     while (j <= r) tmp[k ++ ] = q[j ++ ];
19     //第四步：复制回原数组
20     for (i = l, j = 0; i <= r; i ++, j ++ ) q[i] = tmp[j];
21 }

```

## 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。

```

1 bool check(int x) /* ... */ // 检查x是否满足某种性质
2
3 // 区间[1, r]被划分成[l, mid]和[mid + 1, r]时使用:
4 int bsearch_1(int l, int r)
5 {
6     while (l < r)
7     {
8         int mid = l + r >> 1;
9         if (check(mid)) r = mid;    // check()判断mid是否满足性质
10        else l = mid + 1; //左加右减
11    }
12    return l;
13 }
14 // 区间[1, r]被划分成[l, mid - 1]和[mid, r]时使用:
15 int bsearch_2(int l, int r)
16 {
17     while (l < r)
18     {
19         int mid = l + r + 1 >> 1; //如果下方else后面是1则这里加1
20         if (check(mid)) l = mid;
21         else r = mid - 1; //左加右减
22     }
23     return l;
24 }

```

## 浮点数二分算法模板

```

1 | bool check(double x) {/* ... */} // 检查x是否满足某种性质
2 |
3 | double bsearch_3(double l, double r)
4 |
5 | {
6 |     const double eps = 1e-6;    // eps 表示精度, 取决于题目对精度的要求
7 |     while (r - l > eps)
8 |     {
9 |         double mid = (l + r) / 2;
10 |        if (check(mid)) r = mid;
11 |        else l = mid;
12 |    }
13 |    return l;
14 |

```

## 高精度加法

```

1 // C = A + B, A >= 0, B >= 0
2 vector<int> add(vector<int> &a, vector<int> &b){
3     //c为答案
4     vector<int> c;
5     //t为进位
6     int t=0;
7     for(int i=0;i<a.size()||i<b.size();i++){
8         //不超过a的范围添加a[i]
9         if(i<a.size())t+=a[i];
10        //不超过b的范围添加b[i]
11        if(i<b.size())t+=b[i];
12        //取当前位的答案
13        c.push_back(t%10);
14        //是否进位
15        t/=10;
16    }
17    //如果t!=0的话向后添加1
18    if(t)c.push_back(1);
19    return c;
20 }

```

## 高精度减法

```

1 // C = A - B, 满足A >= B, A >= 0, B >= 0
2 vector<int> sub(vector<int> &A, vector<int> &B)
3 {
4     //答案
5     vector<int> C;
6     //遍历最大的数
7     for (int i = 0, t = 0; i < A.size(); i++)
8     {
9         //t为进位
10        t = A[i] - t;
11        //不超过B的范围t=A[i]-B[i]-t;
12        if (i < B.size()) t -= B[i];
13        //合二为一, 取当前位的答案
14        C.push_back((t + 10) % 10);

```

```

15     //t<0则t=1
16     if (t < 0) t = 1;
17     //t>=0则t=0
18     else t = 0;
19 }
20 //去除前导零
21 while (C.size() > 1 && C.back() == 0) C.pop_back();
22 return C;
23 }
```

## 高精度比大小 (cmp函数)

```

1 //高精度比大小
2 bool cmp(vector<int> &A, vector<int> &B) {
3     if (A.size() != B.size())
4         return A.size() > B.size();
5     for (int i = A.size() - 1; i >= 0; i -- )
6         if (A[i] != B[i])
7             return A[i] > B[i];
8     return true;
9 }
```

## 高精度乘低精度

```

1 // C = A * b, A >= 0, b >= 0
2 vector<int> mul(vector<int> &A, int b)
3 {
4     //类似于高精度加法
5     vector<int> C;
6     //t为进位
7     int t = 0;
8     for (int i = 0; i < A.size() || t; i ++ )
9     {
10         //不超过A的范围t=t+A[i]*b
11         if (i < A.size()) t += A[i] * b;
12         //取当前位的答案
13         C.push_back(t % 10);
14         //进位
15         t /= 10;
16     }
17     //去除前导零
18     while (C.size() > 1 && C.back() == 0) C.pop_back();
19
20     return C;
21 }
```

## 高精度乘高精度

高精度加减乘除：<https://www.bilibili.com/video/BV1LA411v7mt/>

```

1  vector<int> mul(vector<int> &A, vector<int> &B) {
2      vector<int> C(A.size() + B.size()); // 初始化为 0, C的size可以大一点
3
4      for (int i = 0; i < A.size(); i++)
5          for (int j = 0; j < B.size(); j++)
6              C[i + j] += A[i] * B[j];
7      for (int i = 0, t = 0; i < C.size(); i++) { // i = C.size() - 1时 t 一定小于 10
8          t += C[i];
9          C[i] = t % 10;
10         t /= 10;
11     }
12
13     while (C.size() > 1 && C.back() == 0) C.pop_back(); // 必须要去前导 0, 因为最高
位很可能是 0
14     return C;
15 }
```

## 高精度除低精度

```

1 // A / b = C ... r, A >= 0, b > 0
2 vector<int> div(vector<int> &A, int b, int &r)//高精度A, 低精度b, 余数r
{
3     vector<int> C;//答案
4     r = 0;
5     for (int i = A.size() - 1; i >= 0; i -- )
6     {
7         r = r * 10 + A[i];//补全r>=b
8         C.push_back(r / b);//取当前位的答案
9         r %= b;//r%b为下一次计算
10    }
11    reverse(C.begin(), C.end());//倒序为答案
12    while (C.size() > 1 && C.back() == 0) C.pop_back();//去除前导零
13    return C;
14 }
15 }
```

## 高精度除高精度

高精度加减乘除：<https://www.bilibili.com/video/BV1LA411v7mt/>

```

1 vector<int> div(vector<int> &A, vector<int> &B, vector<int> &r) {
2     vector<int> C;
3     if (!cmp(A, B)) {
4         C.push_back(0);
5         r.assign(A.begin(), A.end());
6         return C;
7     }
8     int j = B.size();
9     r.assign(A.end() - j, A.end());
10    while (j <= A.size()) {
11        int k = 0;
12        while (cmp(r, B)) {
13            r = sub(r, B);
14            k++;
15        }
16    }
17 }
```

```

16     C.push_back(k);
17     if (j < A.size())
18         r.insert(r.begin(), A[A.size() - j - 1]);
19     if (r.size() > 1 && r.back() == 0)
20         r.pop_back();
21     j++;
22 }
23 reverse(C.begin(), C.end());
24 while (C.size() > 1 && C.back() == 0)
25     C.pop_back();
26 return C;
27 }
```

## 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。

```

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

### 应用

```

1 const int N=100010;
2 int a[N];
3 int main(){
4     int n,m;
5     scanf("%d",&n);
6     for(int i=1;i<=n;i++)scanf("%d",&a[i]);
7     for(int i=1;i<=n;i++)a[i]=a[i-1]+a[i];
8     scanf("%d",&m);
9     while(m--){
10         int l,r;
11         scanf("%d%d",&l,&r);
12         printf("%d\n",a[r]-a[l-1]);
13     }
14     return 0;
15 }
```

## 二维前缀和

```

1 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]
2 以(x1, y1)为左上角, (x2, y2)为右下角的子矩阵的和为:
3 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

```

1 int s[1010][1010];
2
3 int n,m,q;
```

```

5 int main(){
6     scanf("%d%d%d", &n, &m, &q);
7     for(int i=1; i<=n; i++)
8         for(int j=1; j<=m; j++)
9             scanf("%d", &s[i][j]);
10    for(int i=1; i<=n; i++)
11        for(int j=1; j<=m; j++)
12            s[i][j] += s[i-1][j] + s[i][j-1] - s[i-1][j-1];
13    while(q--){
14        int x1, y1, x2, y2;
15        scanf("%d%d%d%d", &x1, &y1, &x2, &y2);
16        printf("%d\n", s[x2][y2] - s[x2][y1-1] - s[x1-1][y2] + s[x1-1][y1-1]);
17    }
18    return 0;
19 }

```

## 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！！！**

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

### 应用

```

1 using namespace std;
2 int a[100010], s[100010];
3
4 int main(){
5     int n, m;
6     cin >> n >> m;
7     for(int i=1; i<=n; i++) cin >> a[i];
8     for(int i=1; i<=n; i++) s[i] = a[i] - a[i-1]; // 读入并计算差分数组
9     while(m--){
10         int l, r, c;
11         cin >> l >> r >> c;
12         s[l] += c;
13         s[r+1] -= c; // 在原数组中将区间[l, r]加上c
14     }
15     for(int i=1; i<=n; i++){
16         s[i] += s[i-1];
17         cout << s[i] << ' ';
18     } // 给差分数组计算前缀和，就求出了原数组
19     return 0;
20 }

```

## 二维差分

1 | 给以 $(x_1, y_1)$ 为左上角,  $(x_2, y_2)$ 为右下角的子矩阵中的所有元素加上c:  
2 |  $S[x_1, y_1] += c$ ,  $S[x_2 + 1, y_1] -= c$ ,  $S[x_1, y_2 + 1] -= c$ ,  $S[x_2 + 1, y_2 + 1] += c$

### 应用

```

1 | const int N = 1e3 + 10;
2 | int a[N][N], b[N][N];
3 | void insert(int x1, int y1, int x2, int y2, int c)
4 |
5 |     b[x1][y1] += c;
6 |     b[x2 + 1][y1] -= c;
7 |     b[x1][y2 + 1] -= c;
8 |     b[x2 + 1][y2 + 1] += c;
9 |
10| int main()
11|
12| {
13|     int n, m, q;
14|     cin >> n >> m >> q;
15|     for (int i = 1; i <= n; i++)
16|         for (int j = 1; j <= m; j++)
17|             cin >> a[i][j];
18|     for (int i = 1; i <= n; i++)
19|     {
20|         for (int j = 1; j <= m; j++)
21|         {
22|             insert(i, j, i, j, a[i][j]);           //构建差分数组
23|         }
24|     }
25|     while (q--)
26|     {
27|         int x1, y1, x2, y2, c;
28|         cin >> x1 >> y1 >> x2 >> y2 >> c;
29|         insert(x1, y1, x2, y2, c); //加c
30|     }
31|     for (int i = 1; i <= n; i++)
32|     {
33|         for (int j = 1; j <= m; j++)
34|         {
35|             b[i][j] += b[i - 1][j] + b[i][j - 1] - b[i - 1][j - 1]; //二维前缀和
36|         }
37|     }
38|     for (int i = 1; i <= n; i++)
39|     {
40|         for (int j = 1; j <= m; j++)
41|         {
42|             printf("%d ", b[i][j]);
43|         }
44|         printf("\n");
45|     }
46|     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](https://blog.csdn.net/qq_39757593/article/details/129219491)

## 位运算

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

## 双指针算法

```

1 | for (int i = 0, j = 0; i < n; i++)
2 |
3 | {
4 |     while (j < i && check(i, j)) j++;
5 |     // 具体问题的逻辑
6 | }
7 | 常见问题分类:
8 | (1) 对于一个序列，用两个指针维护一段区间
9 | (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：all.erase(unique(alls.begin(), alls.end()), alls.end());

```

1 | vector<int> alls; // 存储所有待离散化的值
2 | sort(alls.begin(), alls.end()); // 将所有值排序
3 | alls.erase(unique(alls.begin(), alls.end()), alls.end()); // 去掉重复元素
4 |
5 | // 二分求出x对应的离散化的值
6 | int find(int x) // 找到第一个大于等于x的位置
7 |
8 | {
9 |     int l = 0, r = alls.size() - 1;
10 |    while (l < r)
11 |    {
12 |        int mid = l + r >> 1;
13 |        if (alls[mid] >= x) r = mid;
14 |        else l = mid + 1;
15 |    }
16 |    return r + 1; // 映射到1, 2, ...n
}

```

## 应用

```

1 | #include <iostream>
2 |
3 | #define N 300010
4 |
5 | using namespace std;
6 |
7 | int n, m;
8 | int a[N], s[N];
9 |
10 | vector<int> alls; // 存入下标容器
11 | vector<PII> add, query; // add增加容器，存入对应下标和增加的值的大小
12 | // query存入需要计算下标区间和的容器
13 | int find(int x)
14 |
15 | {
16 |     int l = 0, r = alls.size() - 1;
17 |     while (l < r) // 查找大于等于x的最小的值的下标
18 |     {
19 |         int mid = l + r >> 1;
20 |         if (alls[mid] >= x) r = mid;
21 |         else l = mid + 1;
22 |     }
23 |     return l;
24 | }
25 |
26 | int sum(int l, int r)
27 | {
28 |     int res = 0;
29 |     for (int i = l; i <= r; i++) res += s[i];
30 |     return res;
31 | }
32 |
33 | int main()
34 | {
35 |     cin >> n >> m;
36 |     for (int i = 0; i < n; i++) cin >> a[i];
37 |     for (int i = 0; i < m; i++) cin >> query[i].first >> query[i].second;
38 |     for (int i = 0; i < n; i++) s[i] = a[i];
39 |     for (int i = 1; i < n; i++) s[i] += s[i - 1];
40 |     for (int i = 0; i < m; i++)
41 |     {
42 |         PII p = query[i];
43 |         add.push_back({p.first, p.second});
44 |         if (p.first == 0) s[p.first] += p.second;
45 |         else s[p.first] -= p.second;
46 |     }
47 |     for (int i = 0; i < m; i++)
48 |     {
49 |         cout << sum(find(query[i].first), query[i].second) << endl;
50 |     }
51 | }

```

```

16     int mid = l + r >> 1;
17     if (alls[mid] >= x) r = mid;
18     else l = mid + 1;
19 }
20 return r + 1;//因为使用前缀和，其下标要+1可以不考虑边界问题
21 }
22
23 int main()
24 {
25     cin >> n >> m;
26     for (int i = 0; i < n; i ++ )
27     {
28         int x, c;
29         cin >> x >> c;
30         add.push_back({x, c});//存入下标即对应的数值c
31
32         alls.push_back(x); //存入数组下标x=add.first
33     }
34
35     for (int i = 0; i < m; i ++ )
36     {
37         int l, r;
38         cin >> l >> r;
39         query.push_back({l, r}); //存入要求的区间
40
41         alls.push_back(l); //存入区间左右下标
42         alls.push_back(r);
43     }
44
45     // 区间去重
46     sort(alls.begin(), alls.end());
47     alls.erase(unique(alls.begin(), alls.end()), alls.end());
48
49     // 处理插入
50     for (auto item : add)
51     {
52         int x = find(item.first); //将add容器的add.secend值存入数组a[]当中,
53         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
54     }
55
56     // 预处理前缀和
57     for (int i = 1; i <= alls.size(); i ++ ) s[i] = s[i - 1] + a[i];
58
59     // 处理询问
60     for (auto item : query)
61     {
62         int l = find(item.first), r = find(item.second); //在下标容器中查找对应的左右
63         //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
64         cout << s[r] - s[l - 1] << endl;
65     }
66
67     return 0;
68 }

```

## 区间合并

1 // 将所有存在交集的区间合并

```

2 void merge(vector<PII> &segs)
3 {
4     vector<PII> res;
5
6     sort(segs.begin(), segs.end());
7
8     int st = -2e9, ed = -2e9;
9     for (auto seg : segs)
10        if (ed < seg.first)
11        {
12            if (st != -2e9) res.push_back({st, ed});
13            st = seg.first, ed = seg.second;
14        }
15        else ed = max(ed, seg.second);
16
17    if (st != -2e9) res.push_back({st, ed});
18
19    segs = res;
20 }
```

## 二、数据结构

### 单链表

```

1 const int N=100010;
2
3 int head,e[N],ne[N],idx;
4 //初始化
5 void init(){
6     head=-1;
7     idx=0;
8 }
9 //在链表头部添加节点
10 void add_to_head(int x){
11     e[idx]=x,ne[idx]=head,head=idx++;
12 }
13 //在位置k添加节点x
14 void add(int k,int x){
15     e[idx]=x,ne[idx]=ne[k],ne[k]=idx++;
16 }
17 //删除位置k的节点
18 void remove(int k){
19     ne[k]=ne[ne[k]];
20 }
```

### 应用

```

1 int main(){
2     int m;
3     init();
4     cin>>m;
5     while(m--){
6         int k,x;
```

```

7     char op;
8     cin>>op;
9     if(op=='H'){
10         cin>>x;
11         add_to_head(x);
12     }else if(op=='D'){
13         cin>>k;
14         if(!k)head=ne[head];
15         remove(k-1);
16     }else {
17         cin>>k>>x;
18         add(k-1,x);
19     }
20 }
21 for(int i=head;i!=-1;i=ne[i])cout<<e[i]<<' ';
22 cout<<endl;
23 return 0;
24 }
```

## 双链表

```

1 const int N=100010;
2
3 int e[N],l[N],r[N],idx;
4
5 //初始化
6 void init(){
7     l[1]=0;
8     r[0]=1;
9     idx=2;
10 }
11
12 //在节点a的右边插入一个数x
13 void insert(int a,int x){
14     e[idx]=x;
15     l[idx]=a,r[idx]=r[a];
16     l[r[a]]=idx,r[a]=idx++;
17 }
18
19 //删除节点a
20 void remove(int a){
21     l[r[a]]=l[a];
22     r[l[a]]=r[a];
23 }
```

## 应用

```

1 int main(){
2     int m;
3     cin>>m;
4     init();
5     while(m--){
6         string op;
7         cin>>op;
8         int k,x;
9         if(op=="L"){//在最左端插入数x
10             cin>>x;
```

```

11     insert(0,x);
12 }else if(op=="R"){//在最右端插入数x
13     cin>>x;
14     insert(l[1],x);
15 }else if(op=="D"){//删除第k个插入的数
16     cin>>k;
17     remove(k+1);
18 }else if(op=="IL"){//在第k个位置的左侧插入一个数
19     cin>>k>>x;
20     insert(l[k+1],x);
21 }else if(op=="LR"){//在第k个位置的右侧插入一个数
22     cin>>k>>x;
23     insert(k+1,x);
24 }
25 }
26 for(int i=r[0];i!=1;i=r[i])printf("%d ",e[i]);
27 cout<<endl;
28 return 0;
29 }
```

## 栈

```

1 // tt表示栈顶
2 int stk[N], tt = 0;
3 // 向栈顶插入一个数
4 stk[ ++ tt] = x;
5 // 从栈顶弹出一个数
6 tt -- ;
7 // 栈顶的值
8 stk[tt];
9 // 判断栈是否为空, 如果 tt > 0, 则表示不为空
10 if (tt > 0)
11 {
12 }
```

## 应用

```

1 const int N=100010;
2 int stk[N],tt;
3
4 int main(){
5     int m;
6     cin>>m;
7     while(m--){
8         string op;
9         int x;
10        cin>>op;
11        if(op=="push"){
12            cin>>x;
13            stk[tt++]=x;
14        }else if(op=="pop"){
15            tt--;
16        }else if(op=="query"){
17            cout<<stk[tt-1]<<endl;
18        }else{
19            if(!tt)cout<<"YES"<<endl;
```

```
20         else cout<<"NO"<<endl;
21     }
22 }
23 return 0;
24 }
```

## 队列

### 普通队列

```
1 // hh 表示队头, tt表示队尾
2 int q[N], hh = 0, tt = -1;
3
4 // 向队尾插入一个数
5 q[ ++ tt] = x;
6
7 // 从队头弹出一个数
8 hh ++ ;
9
10 // 队头的值
11 q[hh];
12
13 // 判断队列是否为空, 如果 hh <= tt, 则表示不为空
14 if (hh <= tt)
15 {
16
17 }
```

### 应用

```
1 int const N=100010;
2
3 int que[N],hh,tt=-1;
4
5 int main(){
6     int m;
7     cin>>m;
8     while(m--){
9         string op;
10        int x;
11        cin>>op;
12        if(op=="push"){
13            cin>>x;
14            que[++tt]=x;
15        }else if(op=="query"){
16            cout<<que[hh]<<endl;
17        }else if(op=="pop"){
18            hh++;
19        }else{
20            if(hh>tt)cout<<"YES"<<endl;
21            else cout<<"NO"<<endl;
22        }
23    }
24 }
25 }
```

## 循环队列

```
1 // hh 表示队头, tt表示队尾的后一个位置
2 int q[N], hh = 0, tt = 0;
3
4 // 向队尾插入一个数
5 q[tt ++ ] = x;
6 if (tt == N) tt = 0;
7
8 // 从队头弹出一个数
9 hh ++ ;
10 if (hh == N) hh = 0;
11
12 // 队头的值
13 q[hh];
14
15 // 判断队列是否为空, 如果hh != tt, 则表示不为空
16 if (hh != tt)
17 {
18
19 }
```

## 单调栈

```
1 常见模型: 找出每个数左边离它最近的比它大/小的数
2 int tt = 0;
3 for (int i = 1; i <= n; i ++ )
4 {
5     while (tt && check(stk[tt], i)) tt -- ;
6     stk[ ++ tt] = i;
7 }
```

### 应用

```
1 找出每个数左边离它最近的比它大/小的数
2 stack<int> stk;
3 int main(){
4     int n;
5     cin >> n;
6     stk.push(-1);
7     for (int i = 0; i < n; i ++){
8         int x;
9         cin >> x;
10        while (stk.size() && stk.top() >= x) stk.pop();
11        cout << stk.top() << " ";
12        stk.push(x);
13    }
14    return 0;
15 }
```

## 单调队列

```

1 常见模型：找出滑动窗口中的最大值/最小值
2 int hh = 0, tt = -1;
3 for (int i = 0; i < n; i++)
4 {
5     while (hh <= tt && check_out(q[hh])) hh ++ ; // 判断队头是否滑出窗口
6     while (hh <= tt && check(q[tt], i)) tt -- ;
7     q[ ++ tt] = i;
8 }

```

```

1 const int N = 1000010;
2 int a[N];
3 int main()
4 {
5     int n, k;
6     cin >> n >> k;
7     for (int i = 1; i <= n; i++) cin >> a[i]; //读入数据
8     deque<int> q;
9     for (int i = 1; i <= n; i++)
10    {
11        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
12            q.pop_back();
13        q.push_back(a[i]); //将新进入的元素入队
14        if (i - k >= 1 && q.front() == a[i - k]) //若队头是否滑出了窗口，队头出队
15            q.pop_front();
16        if (i >= k) //当窗口形成，输出队头对应的值
17            cout << q.front() << " ";
18    }
19    q.clear();
20    cout << endl;
21
22    //最大值亦然
23    for (int i = 1; i <= n; i++)
24    {
25        while (q.size() && q.back() < a[i]) q.pop_back();
26        q.push_back(a[i]);
27        if (i - k >= 1 && a[i - k] == q.front()) q.pop_front();
28        if (i >= k) cout << q.front() << " ";
29    }
30 }
31 }

```

## KMP字符串匹配

下标从1开始的kmp算法

```

1 const int N = 100010, M = 1000010;
2 int n, m;
3 int ne[N];
4 char s[M], p[N];
5 int main()
6 {
7     cin >> n >> p + 1 >> m >> s + 1;
8     for (int i = 2, j = 0; i <= n; i++)
9     {
10         while (j && p[i] != p[j + 1]) j = ne[j];
11         if (p[i] == p[j + 1]) j++;

```

```

12     ne[i] = j;
13 } // 处理ne数组
14 for (int i = 1, j = 0; i <= m; i++)
15 {
16     while (j && s[i] != p[j + 1]) j = ne[j];
17     if (s[i] == p[j + 1]) j++;
18     if (j == n)
19     {
20         printf("%d ", i - n);
21         j = ne[j];
22     }
23 } // 匹配算法
24 return 0;
25 }

```

```

1 // s[]是长文本，p[]是模式串，n是s的长度，m是p的长度
2 求模式串的Next数组：
3 for (int i = 2, j = 0; i <= m; i++)
4 {
5     while (j && p[i] != p[j + 1]) j = ne[j];
6     if (p[i] == p[j + 1]) j++;
7     ne[i] = j;
8 }
9
10 // 匹配
11 for (int i = 1, j = 0; i <= n; i++)
12 {
13     while (j && s[i] != p[j + 1]) j = ne[j];
14     if (s[i] == p[j + 1]) j++;
15     if (j == m)
16     {
17         j = ne[j];
18         // 匹配成功后的逻辑
19     }
20 }

```

下标从0开始的kmp算法

```

1 const int N = 1000010;
2
3 int n, m;
4 char s[N], p[N];
5 int ne[N];
6
7 int main()
8 {
9     cin >> m >> p >> n >> s;
10    ne[0] = -1;
11    for (int i = 1, j = -1; i < m; i++)
12    {
13        while (j >= 0 && p[j + 1] != p[i]) j = ne[j];
14        if (p[j + 1] == p[i]) j++;
15        ne[i] = j;
16    }
17    for (int i = 0, j = -1; i < n; i++)
18    {
19        while (j != -1 && s[i] != p[j + 1]) j = ne[j];
20        if (s[i] == p[j + 1]) j++;
21        if (j == m - 1)

```

```

22     {
23         cout << i - j << ' ';
24         j = ne[j];
25     }
26 }
27 return 0;
28 }
```

## 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/>

```

1 int son[N][26], cnt[N], idx;
2 // 0号点既是根节点，又是空节点
3 // son[][]存储树中每个节点的子节点
4 // cnt[]存储以每个节点结尾的单词数量
5
6 // 插入一个字符串
7 void insert(char *str)
8 {
9     int p = 0;
10    for (int i = 0; str[i]; i++)
11    {
12        int u = str[i] - 'a';
13        if (!son[p][u]) son[p][u] = ++idx;
14        p = son[p][u];
15    }
16    cnt[p]++;
17 }
18
19 // 查询字符串出现的次数
20 int query(char *str)
21 {
22     int p = 0;
23     for (int i = 0; str[i]; i++)
24     {
25         int u = str[i] - 'a';
26         if (!son[p][u]) return 0;
27         p = son[p][u];
28     }
29     return cnt[p];
30 }
```

```

1 const int N = 100010;
2
3 int son[N][26], cnt[N], idx;
4 char str[N];
5
6 void insert(char *str)
7 {
8     int p = 0;
9     for (int i = 0; str[i]; i++)
10    {
11        int u = str[i] - 'a';
12        if (!son[p][u]) son[p][u] = ++idx;
```

```

13     p = son[p][u];
14 }
15 cnt[p]++;
16 }//插入
17
18 int query(char *str)
19 {
20     int p = 0;
21     for (int i = 0; str[i]; i++)
22     {
23         int u = str[i] - 'a';
24         if (!son[p][u]) return 0;
25         p = son[p][u];
26     }
27     return cnt[p];
28 }//查询
29
30 int main()
31 {
32     int n;
33     scanf("%d", &n);
34     while (n--)
35     {
36         char op[2];
37         scanf("%s%s", op, str);
38         if (*op == 'I') insert(str);
39         else printf("%d\n", query(str));
40     }
41
42     return 0;
43 }

```

## 并查集

(1)朴素并查集:

```

1 int p[N]; //存储每个点的祖宗节点
2
3 // 返回x的祖宗节点
4 int find(int x)
5 {
6     if (p[x] != x) p[x] = find(p[x]);
7     return p[x];
8 }
9
10
11 // 初始化, 假定节点编号是1~n
12 for (int i = 1; i <= n; i++) p[i] = i;
13
14 // 合并a和b所在的两个集合:
15 p[find(a)] = find(b);
16
17
18 (2)维护size的并查集:
19
20 int p[N], size[N];
21 //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
22 //数量

```

```

23 // 返回x的祖宗节点
24 int find(int x)
25 {
26     if (p[x] != x) p[x] = find(p[x]);
27     return p[x];
28 }
29
30
31 // 初始化，假定节点编号是1~n
32 for (int i = 1; i <= n; i++)
33 {
34     p[i] = i;
35     size[i] = 1;
36 }
37
38 // 合并a和b所在的两个集合:
39 size[find(b)] += size[find(a)];
40 p[find(a)] = find(b);
41
42
43 (3)维护到祖宗节点距离的并查集:
44
45 int p[N], d[N];
46 //p[]存储每个点的祖宗节点, d[x]存储x到p[x]的距离
47
48 // 返回x的祖宗节点
49 int find(int x)
50 {
51     if (p[x] != x)
52     {
53         int u = find(p[x]);
54         d[x] += d[p[x]];
55         p[x] = u;
56     }
57     return p[x];
58 }
59
60 // 初始化，假定节点编号是1~n
61 for (int i = 1; i <= n; i++)
62 {
63     p[i] = i;
64     d[i] = 0;
65 }
66
67 // 合并a和b所在的两个集合:
68 p[find(a)] = find(b);
69 d[find(a)] = distance; // 根据具体问题, 初始化find(a)的偏移量

```

## 应用

```

1 const int N=100010;
2 int p[N],n,m;
3
4 int find(int x){//找到祖宗节点+路径压缩
5     if(p[x]!=x)p[x]=find(p[x]);
6     return p[x];
7 }
8
9 int main(){
10     scanf("%d%d",&n,&m);

```

```

11  for(int i=1;i<=n;i++)p[i]=i;
12  while(m--){
13      char op[2];
14      int a,b;
15      scanf("%s%d%d",op,&a,&b);
16      if(op[0]=='M')p[find(a)]=find(b);
17      else {
18          if(find(a)==find(b))puts("Yes");
19          else puts("No");
20      }
21  }
22  return 0;
23 }
```

## 堆

```

1 // 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
2 // ph[k]存储第k个插入的点在堆中的位置
3 // hp[k]存储堆中下标是k的点是第几个插入的
4 int h[N], ph[N], hp[N], size;
5
6 // 交换两个点, 及其映射关系
7 void heap_swap(int a, int b)
8 {
9     swap(ph[hp[a]],ph[hp[b]]);
10    swap(hp[a], hp[b]);
11    swap(h[a], h[b]);
12 }
13
14 void down(int u)
15 {
16     int t = u;
17     if (u * 2 <= size && h[u * 2] < h[t]) t = u * 2;
18     if (u * 2 + 1 <= size && h[u * 2 + 1] < h[t]) t = u * 2 + 1;
19     if (u != t)
20     {
21         heap_swap(u, t);
22         down(t);
23     }
24 }
25
26 void up(int u)
27 {
28     while (u / 2 && h[u] < h[u / 2])
29     {
30         heap_swap(u, u / 2);
31         u >>= 1;
32     }
33 }
34
35 // O(n)建堆
36 for (int i = n / 2; i; i --) down(i);
```

应用：堆排序

```

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

```

3
4 void down(int u){
5     int t=u;
6     if(u*2<=cnt&&heap[u*2]<=heap[t])t=u*2;
7     if(u*2+1<=cnt&&heap[u*2+1]<=heap[t])t=u*2+1;
8     if(t!=u){
9         swap(heap[t],heap[u]);
10        down(t);
11    }
12 } //down操作
13
14 void up(int u){
15     while(u/2&&heap[u/2]>heap[u]){
16         swap(heap[u/2],heap[u]);
17         u>>=1;
18     }
19 } //up操作
20
21 int main(){
22     int n,m;
23     scanf("%d%d",&n,&m);
24     for(int i=1;i<=n;i++)scanf("%d",&heap[i]);
25     cnt=n;
26     for(int i=n/2;i;i--)down(i);
27     while(m--){
28         printf("%d ",heap[1]);
29         heap[1]=heap[cnt--];
30         down(1);
31     }
32     return 0;
33 }
```

## 一般hash

```

1 (1) 拉链法
2     int h[N], e[N], ne[N], idx;
3
4     // 向哈希表中插入一个数
5     void insert(int x)
6     {
7         int k = (x % N + N) % N;
8         e[idx] = x;
9         ne[idx] = h[k];
10        h[k] = idx++;
11    }
12
13    // 在哈希表中查询某个数是否存在
14    bool find(int x)
15    {
16        int k = (x % N + N) % N;
17        for (int i = h[k]; i != -1; i = ne[i])
18            if (e[i] == x)
19                return true;
20
21        return false;
22    }
23
```

```

24 (2) 开放寻址法
25     int h[N];
26
27     // 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
28     int find(int x)
29     {
30         int t = (x % N + N) % N;
31         while (h[t] != null && h[t] != x)
32         {
33             t++;
34             if (t == N) t = 0;
35         }
36         return t;
37     }

```

## 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

```

1
2
3
4     typedef unsigned long long ULL;
5     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  $P^k \bmod 2^{64}$ 
6
7     // 初始化
8     p[0] = 1;
9     for (int i = 1; i <= n; i++)
10    {
11        h[i] = h[i - 1] * P + str[i];
12        p[i] = p[i - 1] * P;
13    }
14
15    // 计算子串 str[l ~ r] 的哈希值
16    ULL get(int l, int r)
17    {
18        return h[r] - h[l - 1] * p[r - l + 1];
19    }

```

## STL

名称	容器	头文件	创建	添加	删除	访问
变长数组	vector	<vector>	vector<int> v	v.push_back(x) O(1)	v.pop_back() O(1)	v[i] O(1)
双端队列	deque	<deque>	deque<int> d	d.push_front(x) O(1) d.push_back(x) O(1)	d.pop_front() O(1) d.pop_back() O(1)	d.front() O(1) d.back() O(1)
栈	stack	<stack>	stack<int> s	s.push(x) O(1)	s.pop() O(1)	s.top() O(1)
队列	queue	<queue>	queue<int> q	q.push(x) O(1)	q.pop() O(1)	q.front() O(1)
优先队列 (大根堆)	priority_queue	<queue>	priority_queue<int> q	q.push(x) O(logn)	q.pop() O(logn)	q.top() O(1)
有序集合	set	<set>	set<int> s	s.insert(x) O(logn)	s.erase(it) O(logn)	s.count(x) O(k+logn)
无序集合	unordered_set	<unordered_set>	unordered_set<int> s	s.insert(x) O(1)/O(n)	s.erase(it) O(1)/O(n)	s.count(x) O(1)/O(n)
有序键值对映射	map	<map>	map<string, int> h	h[str]=x O(logn)	h.erase(it) O(logn)	h.count(str); h[str] O(logn)
无序键值对映射	unordered_map	<unordered_map>	unordered_map<string, int> h	h[str]=x O(1)/O(n)	h.erase(it) O(1)/O(n)	h.count(str); h[str] O(1)/O(n)

### 视频讲解: 100 STL 容器哔哩哔哩bilibili

```

1  vector, 变长数组, 倍增的思想
2      size()    返回元素个数
3      empty()   返回是否为空
4      clear()   清空
5      front()/back()
6      push_back()/pop_back()
7      begin()/end()
8      []
9      支持比较运算, 按字典序
10
11 pair<int, int>
12     first, 第一个元素
13     second, 第二个元素
14     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)
15
16 string, 字符串
17     size()/length()  返回字符串长度
18     empty()
19     clear()
20     substr(起始下标, (子串长度))  返回子串
21     c_str()  返回字符串所在字符数组的起始地址
22
23 queue, 队列
24     size()
25     empty()
26     push()  向队尾插入一个元素
27     front()  返回队头元素
28     back()  返回队尾元素
29     pop()  弹出队头元素
30
31 priority_queue, 优先队列, 默认是大根堆
32     size()
33     empty()
34     push()  插入一个元素
35     top()  返回堆顶元素
36     pop()  弹出堆顶元素
37     定义成小根堆的方式: priority_queue<int, vector<int>, greater<int>> q;
38
39 stack, 栈

```

```

40     size()
41     empty()
42     push() 向栈顶插入一个元素
43     top() 返回栈顶元素
44     pop() 弹出栈顶元素
45
46 deque, 双端队列
47     size()
48     empty()
49     clear()
50     front()/back()
51     push_back()/pop_back()
52     push_front()/pop_front()
53     begin()/end()
54     []
55
56 set, map, multiset, multimap, 基于平衡二叉树（红黑树），动态维护有序序列
57     size()
58     empty()
59     clear()
60     begin()/end()
61     ++, -- 返回前驱和后继，时间复杂度 O(logn)
62
63 set/multiset
64     insert() 插入一个数
65     find() 查找一个数
66     count() 返回某一个数的个数
67     erase()
68         (1) 输入是一个数x, 删除所有x O(k + logn)
69         (2) 输入一个迭代器, 删除这个迭代器
70     lower_bound()/upper_bound()
71         lower_bound(x) 返回大于等于x的最小的数的迭代器
72         upper_bound(x) 返回大于x的最小的数的迭代器
73 map/multimap
74     insert() 插入的数是一个pair
75     erase() 输入的参数是pair或者迭代器
76     find()
77     [] 注意multimap不支持此操作。 时间复杂度是 O(logn)
78     lower_bound()/upper_bound()
79
80 unordered_set, unordered_map, unordered_multiset, unordered_multimap, 哈希表
81 和上面类似，增删改查的时间复杂度是 O(1)
82 不支持 lower_bound()/upper_bound(), 迭代器的++, --
83
84 bitset, 压位
85     bitset<10000> s;
86     ~, &, |, ^
87     >>, <<
88     ==, !=
89     []
90
91     count() 返回有多少个1
92
93     any() 判断是否至少有一个1
94     none() 判断是否全为0
95
96     set() 把所有位置成1
97     set(k, v) 将第k位变成v
98     reset() 把所有位变成0
99     flip() 等价于~
100    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

```
1 // 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
2 int h[N], e[N], ne[N], idx;
3 // 添加一条边a->b
4 void add(int a, int b)
5 {
6     //存下b的值，b下一个指向a的下一个节点，a的下一个节点指向b
7     e[idx] = b, ne[idx] = h[a], h[a] = idx++;
8 }
9 // 初始化
10 idx = 0;
11 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

```
1 int dfs(int u)
2 {
3     st[u] = true; // st[u] 表示点u已经被遍历过
4
5     for (int i = h[u]; i != -1; i = ne[i])
6     {
7         int j = e[i];
8         if (!st[j]) dfs(j);
9     }
10 }
```

## 应用：数字全排列

```
1 #include <iostream>
2
3 using namespace std;
4
5 int res[10], b[10], n;
6
7 void dfs(int k){
8     if(k==n){//k==n则输出n个数字
9         for(int i=0; i<n; i++) printf("%d ", res[i]);
10        cout<<endl;
11    }
12    for(int i=1; i<=n; i++){
13        if(!b[i]){//判断是否被用过
14            res[k]=i;//当前k位存入位置
15            b[i]=1;//表示被占用
16            dfs(k+1);
17            b[i]=0;//恢复现场
18        }
19    }
20}
21
22 int main(){
23     cin>>n;
24     dfs(0); //从0开始枚举
25     return 0;
26 }
```

## 应用：树的重心

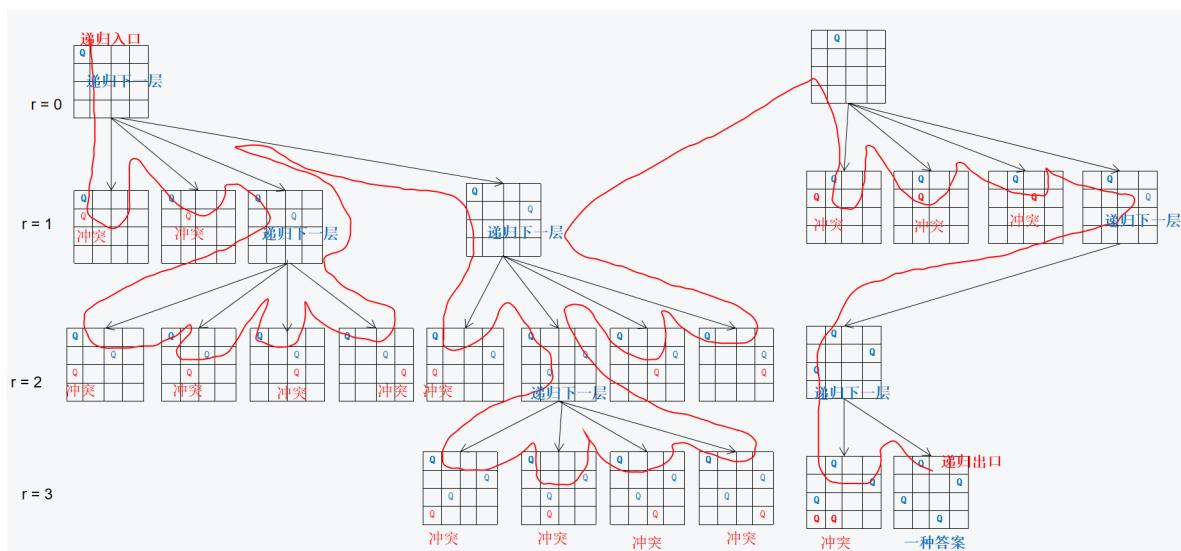
```
1 #include <cstdio>
2 #include <cstring>
3 #include <iostream>
4 #include <algorithm>
5
6 using namespace std;
7
8 const int N = 100010, M = N * 2; //无向图n条边时，最多2n个idx，因为每条边在邻接表中会出现两次
9
10 int n; //n个结点, n-1条边
11 int h[N], e[M], ne[M], idx; //n个链表头， e每一个结点的值， ne每一个结点的next指针
12 int ans = N; //最小的最大值
13 bool st[N]; //状态数组，防止子节点搜索父节点
14
15 void add(int a, int b) //a->b
16 { //e记录当前点的值(地址->值), ne下一点的地址(地址->地址), h记录指向的第一个点的地址(值->地址)
17     e[idx] = b, ne[idx] = h[a], h[a] = idx++;
18 }
19 //头插法
20
21 int dfs(int u) //通过h数组找到子结点的向
22 {
23     st[u] = true; //st标记当前点被搜过
24
25     int size = 0, sum = 0;
26     //size删掉元素后各个子连通块的最大值
```

```

27 //sum当前子树大小，遍历叶节点时，返回1
28
29     for (int i = h[u]; i != -1; i = ne[i])//遍历单链表，链表末端初始化为-1
30     {
31         int j=e[i];
32         if(st[j])continue;//此处防逆向dfs
33         int s = dfs(j);//s各个子连通块的大小
34         size = max(size, s);//size删掉元素后各个连通块的最大值
35         sum += s;//各个连通块大小之和
36     }
37
38     size = max(size, n - sum - 1);//判断最大子连通块与父连通块的最大值
39     ans = min(ans, size);//全局变量ans存最小的最大值
40 //注意：本题若求最大的最大值，则只需去除任意叶节点即可，即n-1
41     return sum + 1;//各个子连通块，当前结点之和
42 }
43
44 int main()
45 {
46     scanf("%d", &n);
47
48     memset(h, -1, sizeof h);//n个头节点全部指向-1
49
50     for (int i = 0; i < n - 1; i++)//n个结点，n-1条边
51     {
52         int a, b;
53         scanf("%d%d", &a, &b);
54         add(a, b), add(b, a);//不知道子节点还是父节点，所以需要建两条边可以双向查找
55     }
56
57     dfs(1); //结点编号为1~n且可能只有一个结点，则参数只能为1
58     printf("%d\n", ans);
59     return 0;
60 }

```

## 应用：n-皇后问题



```

1 using namespace std;
2
3 const int N = 11;
4
5 char q[N][N];//存储棋盘
6 bool dg[N * 2], udg[N * 2], cor[N];//点对应的两个斜线以及列上是否有皇后

```

```

7   int n;
8
9
10 void dfs(int r)
11 {
12     if(r == n)//放满了棋盘，输出棋盘
13     {
14         for(int i = 0; i < n; i++)
15         {
16             for(int j = 0; j < n; j++)
17                 cout << q[i][j];
18             cout << endl;
19         }
20         cout << endl;
21         return;
22     }
23
24     for(int i = 0; i < n; i++)//第 r 行，第 i 列 是否放皇后
25     {
26         if(!cor[i] && !dg[i + r] && !udg[n - i + r])//不冲突，放皇后
27         {
28             q[r][i] = 'Q';
29             cor[i] = dg[i + r] = udg[n - i + r] = 1;//对应的列，斜线状态改变
30             dfs(r + 1);//处理下一行
31             cor[i] = dg[i + r] = udg[n - i + r] = 0;//恢复现场
32             q[r][i] = '.';
33         }
34     }
35 }
36
37 int main()
38 {
39     cin >> n;
40     for (int i = 0; i < n; i++)
41         for (int j = 0; j < n; j++)
42             q[i][j] = '.';
43     dfs(0);
44     return 0;
45 }
```

## 宽度优先遍历

```

1 queue<int> q;
2 st[1] = true; // 表示1号点已经被遍历过
3 q.push(1);
4
5 while (q.size())
6 {
7     int t = q.front();
8     q.pop();
9
10    for (int i = h[t]; i != -1; i = ne[i])
11    {
12        int j = e[i];
13        if (!st[j])
14        {
15            st[j] = true; // 表示点j已经被遍历过
16            q.push(j);
17        }
18    }
19}
```

```

16         q.push(j);
17     }
18 }
19 }
```

## 应用：走迷宫

```

1 typedef pair<int,int> PII;//声明pair时候必须要在代码前面写上using namespace std;
2
3 const int N=110;
4 int g[N][N],f[N][N],n,m;
5
6 int bfs(int x,int y){
7     queue<PII> que;
8     que.push({x,y});
9     int dx[4]={0,1,0,-1},dy[4]={1,0,-1,0};
10    while(!que.empty()){
11        PII t=que.front();
12        que.pop();
13        g[t.first][t.second]=1;
14        for(int i=0;i<4;i++){
15            int a=t.first+dx[i],b=t.second+dy[i];
16            if(a>=0&&b>=0&&a<n&&b<m&&!g[a][b]){
17                g[a][b]=1;
18                f[a][b]=f[t.first][t.second]+1;
19                que.push({a,b});
20            }
21        }
22    }
23    return f[n-1][m-1];
24 }
25
26 int main(){
27     scanf("d%d",&n,&m);
28     for(int i=0;i<n;i++)
29         for(int j=0;j<m;j++)
30             scanf("d",&g[i][j]);
31     cout<<bfs(0,0)<<endl;
32     return 0;
33 }
```

## 应用：八数码

```

1 using namespace std;
2
3 int bfs(string state) {
4     queue<string> q;
5     unordered_map<string, intint dx[4] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};
8     string ed = "12345678x";
9     q.push(state);
10    d[state] = 0;
11
12    while (q.size()) {
13        auto t = q.front();
14        q.pop();
15        if (t == ed)//等于结果就输出步数
```

```

16         return d[t];
17     int distance = d[t];
18     int k = t.find('x');//寻找x
19     int x = k / 3, y = k % 3;//计算下标
20     for (int i = 0; i < 4; i ++ ) {
21         int a = x + dx[i], b = y + dy[i];
22         if (a >= 0 && a < 3 && b >= 0 && b < 3) {
23             swap(t[a * 3 + b], t[k]);//交换
24             if (!d.count(t)) {//不存在就入队
25                 d[t] = distance + 1;
26                 q.push(t);
27             }
28             swap(t[a * 3 + b], t[k]);//还原
29         }
30     }
31 }
32     return -1;
33 }
34
35 int main() {
36     char s[2];
37     string state;
38     for (int i = 0; i < 9; i ++ ) {
39         cin >> s;
40         state += *s;
41     }
42     cout<<bfs(state)<<endl;
43     return 0;
44 }
```

## 拓扑排序

啥是拓扑排序？

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

一直进行上面处理，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

```

1 bool topsort()
2 {
3     int hh = 0, tt = -1;
4
5     // d[i] 存储点i的入度
6     for (int i = 1; i <= n; i ++ )
7         if (!d[i])
8             q[ ++ tt] = i;
9
10    while (hh <= tt)
11    {
12        int t = q[hh ++ ];
13
14        for (int i = h[t]; i != -1; i = ne[i])
```

```

15     {
16         int j = e[i];
17         if (--d[j] == 0)
18             q[ ++tt ] = j;
19     }
20 }
21
22 // 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。
23 return tt == n - 1;
24 }

```

## 解说版

```

1 using namespace std;
2 const int N = 100010;
3 int e[N], ne[N], idx; //邻接表存储图
4 int h[N]; //邻接表的每个头链表
5 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
6 int n, m; //保存图的点数和边数
7 int d[N]; //保存各个点的入度
8
9 void add(int a, int b) {
10     e[idx] = b, ne[idx] = h[a], h[a] = idx++;
11 }
12
13 void topsort() {
14     for (int i = 1; i <= n; i++) { //遍历一遍顶点的入度。
15         if (!d[i]) //如果入度为0，则可以入队列
16             q[ ++tt ] = i;
17     }
18     while (tt >= hh) { //循环处理队列中点的
19         int a = q[hh++];
20         for (int i = h[a]; i != -1; i = ne[i]) {
21             int b = e[i]; //a有一条边指向b
22             d[b]--; //删除边后，b的入度减1
23             if (!d[b]) //如果b的入度减为0，则b可以输出，入队列
24                 q[ ++tt ] = b;
25         }
26     }
27     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
28         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
29             printf("%d ", q[i]);
30     } 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
31         cout << -1;
32 }
33
34 int main() {
35     cin >> n >> m; //保存点的个数和边的个数
36     memset(h, -1, sizeof h); //初始化邻接矩阵
37     while (m--) { //依次读入边
38         int a, b;
39         cin >> a >> b;
40         d[b]++; //顶点b的入度+1
41         add(a, b); //添加到邻接矩阵
42     }
43     topsort(); //进行拓扑排序
44     return 0;

```

45 }

# Dijkstra算法

## 朴素版

时间复杂度是 $O(n^2+m)$ , n表示点数, m表示边数

```
1 int g[N][N]; // 存储每条边
2 int dist[N]; // 存储1号点到每个点的最短距离
3 bool st[N]; // 存储每个点的最短路是否已经确定
4
5 // 求1号点到n号点的最短路, 如果不存在则返回-1
6 int dijkstra()
7 {
8     memset(dist, 0x3f, sizeof dist);
9     dist[1] = 0;
10
11    for (int i = 0; i < n - 1; i++)
12    {
13        int t = -1; // 在还未确定最短路的点中, 寻找距离最小的点
14        for (int j = 1; j <= n; j++)
15            if (!st[j] && (t == -1 || dist[t] > dist[j]))
16                t = j;
17
18        // 用t更新其他点的距离
19        for (int j = 1; j <= n; j++)
20            dist[j] = min(dist[j], dist[t] + g[t][j]);
21
22        st[t] = true;
23    }
24
25    if (dist[n] == 0x3f3f3f3f) return -1;
26    return dist[n];
27 }
```

## 应用

```
1 const int N = 510, M = 100010;
2
3 int h[N], e[M], ne[M], w[M], idx; //邻接表存储图
4 int state[N]; //state 记录是否找到了源点到该节点的最短距离
5 int dist[N]; //dist 数组保存源点到其余各个节点的距离
6 int n, m; //图的节点个数和边数
7
8 void add(int a, int b, int c) //插入边
{
9     e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
10}
11
12
13 void Dijkstra()
14 {
15     memset(dist, 0x3f, sizeof(dist)); //dist 数组的各个元素为无穷大
16     dist[1] = 0; //源点到源点的距离置为 0
17     for (int i = 0; i < n; i++)
18     {
```

```

19         int t = -1;
20         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
21     {
22         if (!state[j] && (t == -1 || dist[j] < dist[t]))
23             t = j;
24     }
25
26     state[t] = 1;//state[i] 置为 1。
27
28     for (int j = h[t]; j != -1; j = ne[j])//遍历 t 所有可以到达的节点 i
29     {
30         int i = e[j];
31         dist[i] = min(dist[i], dist[t] + w[j]);//更新 dist[j]
32     }
33
34     }
35 }
36
37
38 int main()
39 {
40     memset(h, -1, sizeof(h));//邻接表初始化
41
42     cin >> n >> m;
43     while (m--)//读入 m 条边
44     {
45         int a, b, w;
46         cin >> a >> b >> w;
47         add(a, b, w);
48     }
49
50     Dijkstra();
51     if (dist[n] != 0x3f3f3f3f)//如果dist[n]被更新了，则存在路径
52         cout << dist[n];
53     else
54         cout << "-1";
55 }
```

## 堆优化版

时间复杂度\$O(m\log n)\$, n表示点数, m表示边数

```

1  typedef pair<int, int> PII;
2
3  int n;      // 点的数量
4  int h[N], w[N], e[N], ne[N], idx;      // 邻接表存储所有边
5  int dist[N];        // 存储所有点到1号点的距离
6  bool st[N];        // 存储每个点的最短距离是否已确定
7
8  // 求1号点到n号点的最短距离，如果不存在，则返回-1
9  int dijkstra(){
10     memset(dist, 0x3f, sizeof dist); // 距离初始化为无穷大
11     dist[1]=0;//1->1的节点距离为0
12     priority_queue<PII, vector<PII>, greater<PII>> heap; // 小根堆
13     heap.push({0, 1}); // 插入距离和节点编号
14 }
```

```

15     while(heap.size()){
16         auto t=heap.top(); //取距离源点最近的点
17         heap.pop();
18
19         int ver=t.second,distance=t.first; //ver: 节点编号, distance源点距离ver
20         if(st[ver]) continue; //如果距离已经确定, 则跳过该点
21         st[ver]=true;
22         for(int i=h[ver];i!= -1;i=ne[i]) //更新ver所指向的节点距离
23         {
24             int j=e[i];
25             if(dist[j]>dist[ver]+w[i]){
26                 dist[j]=dist[ver]+w[i];
27                 heap.push({dist[j],j}); //距离变小, 则入堆
28             }
29         }
30     }
31     if(dist[n]==0x3f3f3f3f) return -1;
32     return dist[n];
33 }
```

关于Dijkstra的相关博客链接:

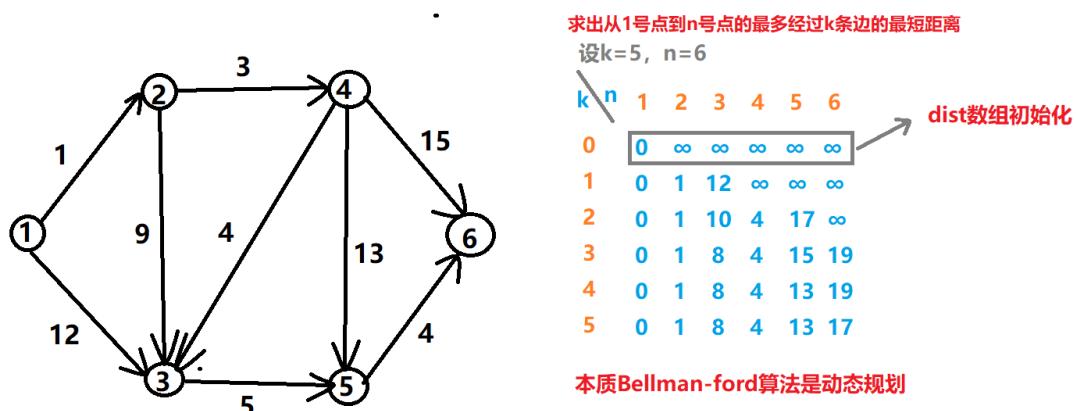
[AcWing 849. Dijkstra求最短路 I: 图解 详细代码 \(图解\) - AcWing](#)

[AcWing 850. Dijkstra求最短路 II: 详解+代码注释 - AcWing](#)

## 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。



QQ: 946808247

上图为Bellman-ford草稿图

```

1 int n, m; // n表示点数, m表示边数
2 int dist[N], backup[N]; // dist[x]存储1到x的最短路距离
3
4 struct Edge // 边, a表示出点, b表示入点, w表示边的权重
5 {
6     int a, b, w;
7 }edges[M];
8
```

```

9 // 求1到n的最短路距离, 如果无法从1走到n, 则返回-1。
10 int bellman_ford()
11 {
12     memset(dist, 0x3f, sizeof dist);
13     dist[1] = 0;
14
15     // 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,
16     // 路径中至少存在两个相同的点, 说明图中存在负权回路。
17     for (int i = 0; i < n; i++)
18     {
19         memcpy(back, dist, sizeof dist);
20         for (int j = 0; j < m; j++)
21         {
22             int a = edges[j].a, b = edges[j].b, w = edges[j].w;
23             if (dist[b] > backup[a] + w)
24                 dist[b] = backup[a] + w;
25         }
26
27         if (dist[n] > 0x3f3f3f3f / 2) return -1;
28     }
29 }

```

## 应用

```

1 int n,m,k;
2 const int N=512,M=10012;
3 struct Edge{
4     int a,b,w;
5 }e[M];
6 int dist[N];
7 int back[N];
8 void bellman_ford(){
9     memset(dist,0x3f,sizeof dist);
10    dist[1]=0;
11    for(int i=0;i<k;i++){
12        memcpy(back,dist,sizeof dist);
13        for(int j=0;j<m;j++){
14            int a=e[j].a,b=e[j].b,c=e[j].w;
15            dist[b]=min(dist[b],back[a]+c);
16        }
17    }
18 }
19
20 int main(){
21     scanf("%d%d%d",&n,&m,&k);
22     for(int i=0;i<m;i++){
23         int a,b,w;
24         scanf("%d%d%d",&a,&b,&w);
25         e[i]={a,b,w};
26     }
27     bellman_ford();
28     if(dist[n]>0x3f3f3f3f/2)cout<<"impossible"<<endl;
29     else cout<<dist[n]<<endl;
30     return 0;
31 }

```

问题：为什么把每一条边用不等式刷k次就是k条件下的值？

你可以想象这个图是 $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \dots \rightarrow n$ 这样一条直线。比如说第一次迭代，为什么只有与原点相连的点才能被更新dist呢？因为原点的dist是0，其他点的dist是 $+\infty$ ，满足 $\text{dist}[2] > \text{dist}[1] + c$ ，而 $+\infty$ 并不 $> +\infty + c$ ，所以第一次迭代结束就是不超过一条边走到i节点最短路的距离，依次类推，第二次迭代，只有3会被更新，因为只有1、2的dist不是 $+\infty$ ，第二次迭代就是不超过2条边走到i节点的最短距离。这就是为什么k次迭代最多是走了k条边，同时也是为什么一共只用迭代n-1次，因为n个点的有向图，如果能走到，原点到n号点的最短距离最多是n-1次，也就是 $1 \rightarrow 2 \rightarrow \dots \rightarrow n$ 直线这种。

## 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

```
1 int n;      // 总点数
2 int h[N], w[N], e[N], ne[N], idx;      // 邻接表存储所有边
3 int dist[N];      // 存储每个点到1号点的最短距离
4 bool st[N];      // 存储每个点是否在队列中
5
6 // 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
7 int spfa()
8 {
9     memset(dist, 0x3f, sizeof dist);
10    dist[1] = 0;
11
12    queue<int> q;
13    q.push(1);
14    st[1] = true;
15
16    while (q.size())
17    {
18        auto t = q.front();
19        q.pop();
20
21        st[t] = false;
22
23        for (int i = h[t]; i != -1; i = ne[i])
24        {
25            int j = e[i];
26            if (dist[j] > dist[t] + w[i])
27            {
28                dist[j] = dist[t] + w[i];
29                if (!st[j])      // 如果队列中已存在j，则不需要将j重复插入
30                {
31                    q.push(j);
32                    st[j] = true;
33                }
34            }
35        }
36    }
37
38    if (dist[n] == 0x3f3f3f3f) return -1;
39    return dist[n];
40 }
```

应用

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

```

2
3 int n, m; //节点数量和边数
4 int h[N], w[N], e[N], ne[N], idx; //邻接矩阵存储图
5 int dist[N]; //存储距离
6 bool st[N]; //存储状态
7
8 void add(int a, int b, int c)
{
9     e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
10}
11
12
13 int spfa()
14 {
15     memset(dist, 0x3f, sizeof dist); //距离初始化为无穷大
16     dist[1] = 0; //初始化1到1的距离为0
17     queue<int> que; //队列
18     que.push(1); //1入队
19
20     while (que.size()) //判断是否存在
21     {
22         int t = que.front();
23         que.pop(); //获取第一个并出队
24         st[t] = false; //第一个取消占用
25         for (int i = h[t]; i != -1; i = ne[i]) //遍历第一个可以到达的结点
26         {
27             int j = e[i];
28             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
29             {
30                 dist[j] = dist[t] + w[i]; //赋值给可到达的节点
31                 if (!st[j]) //如果可到达的节点未被占用
32                 {
33                     que.push(j); //则入队
34                     st[j] = true; //占用
35                 }
36             }
37         }
38     }
39
40     return dist[n];
41 }
42
43
44 int main()
45 {
46     scanf("%d%d", &n, &m);
47
48     memset(h, -1, sizeof h);
49     while (m--)
50     {
51         int a, b, c;
52         scanf("%d%d%d", &a, &b, &c);
53         add(a, b, c);
54     }
55
56     int t = spfa();
57     if (t == 0x3f3f3f3f) cout << "impossible" << endl;
58     else printf("%d\n", t);
59
60     return 0;
61 }

```

## 应用: spfa判断图中是否存在负权

```
1 int n;      // 总点数
2 int h[N], w[N], e[N], ne[N], idx;      // 邻接表存储所有边
3 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
4 bool st[N];      // 存储每个点是否在队列中
5
6 // 如果存在负环, 则返回true, 否则返回false。
7 bool spfa()
8 {
9     // 不需要初始化dist数组
10    // 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
11    // 原理一定有两个点相同, 所以存在环。
12
13    queue<int> q;
14    for (int i = 1; i <= n; i++)
15    {
16        q.push(i);
17        st[i] = true;
18    }
19
20    while (q.size())
21    {
22        auto t = q.front();
23        q.pop();
24
25        st[t] = false;
26
27        for (int i = h[t]; i != -1; i = ne[i])
28        {
29            int j = e[i];
30            if (dist[j] > dist[t] + w[i])
31            {
32                dist[j] = dist[t] + w[i];
33                cnt[j] = cnt[t] + 1;
34                if (cnt[j] >= n) return true;      // 如果从1号点到x的最短路中包含
35                if (!st[j])
36                {
37                    q.push(j);
38                    st[j] = true;
39                }
40            }
41        }
42
43    return false;
44 }
```

## floyd算法

时间复杂度\$O(n^3)\$, n表示点数

视频讲解: <https://www.bilibili.com/video/BV14R4y1x7GB/>

模板

```

1 | 初始化:
2 |     for (int i = 1; i <= n; i++)
3 |         for (int j = 1; j <= n; j++)
4 |             if (i == j) d[i][j] = 0;
5 |             else d[i][j] = INF;
6 |
7 | // 算法结束后, d[a][b]表示a到b的最短距离
8 | void floyd()
9 |
10| {
11|     for (int k = 1; k <= n; k++) //k为中转节点
12|         for (int i = 1; i <= n; i++)
13|             for (int j = 1; j <= n; j++)
14|                 d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}

```

## 应用

```

1 | using namespace std;
2 |
3 | const int N = 210, INF = 1e9;
4 |
5 | int n, m, Q;
6 | int d[N][N];
7 |
8 | void floyd()
9 |
10| {
11|     for (int k = 1; k <= n; k++) //k为中转节点
12|         for (int i = 1; i <= n; i++)
13|             for (int j = 1; j <= n; j++)
14|                 d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}
15 |
16 | int main()
17 |
18 | {
19 |     scanf("%d%d%d", &n, &m, &Q);
20 |
21 |     for (int i = 1; i <= n; i++)
22 |         for (int j = 1; j <= n; j++)
23 |             if (i == j) d[i][j] = 0;
24 |             else d[i][j] = INF;
25 |
26 |     while (m --)
27 |     {
28 |         int a, b, c;
29 |         scanf("%d%d%d", &a, &b, &c);
30 |         d[a][b] = min(d[a][b], c);
31 |     }
32 |
33 |     floyd();
34 |
35 |     while (Q --)
36 |     {
37 |         int a, b;
38 |         scanf("%d%d", &a, &b);
39 |
40 |         int t = d[a][b];
41 |         if (t > INF / 2) puts("impossible");
42 |         else printf("%d\n", t);
43 |     }
}

```

```
44     return 0;
45 }
```

## 最短路算法总结

### 最短路

单源最短路：给定V中的一个顶点，称为源。要计算从源到其他所有各顶点的最短路径长度。这里的长度就是指路上各边权之和。这个问题通常称为单源最短路径问题。

所有边权都是正数：

朴素Dijkstra算法  $O(n^2)$  适合稠密图，贪心思想

堆优化版的Dijkstra算法  $O(m \log n)$  适合稀疏图，贪心思想

存在负权边：

Bellman-ford  $O(nm)$ ，动态规划思想

SPFA 一般： $O(m)$ ，最坏  $O(nm)$

多源汇最短路：任意两点最短路径被称为多源最短路径，即给定任意两个点，一个出发点，一个到达点，求这两个点的之间的最短路径，就是任意两点最短路径问题

Floyd算法  $O(n^3)$

## prim算法

时间复杂度是 $O(n^2+m)$ ，n表示点数，m表示边数

```
1 int n;      // n表示点数
2 int g[N][N];    // 邻接矩阵，存储所有边
3 int dist[N];    // 存储其他点到当前最小生成树的距离
4 bool st[N];    // 存储每个点是否已经在生成树中
5
6
7 // 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
8 int prim()
9 {
10     memset(dist, 0x3f, sizeof dist);
11
12     int res = 0;
13     for (int i = 0; i < n; i++)
14     {
15         int t = -1;
16         for (int j = 1; j <= n; j++)
17             if (!st[j] && (t == -1 || dist[t] > dist[j]))
18                 t = j;
19
20         if (i && dist[t] == INF) return INF;
21
22         if (i) res += dist[t];
23         st[t] = true;
24
25         for (int j = 1; j <= n; j++) dist[j] = min(dist[j], g[t][j]);
26     }
27
28     return res;
29 }
```

## 应用

```
1 const int N = 510, INF = 0x3f3f3f3f;
2
3 int n, m;
4 int g[N][N];
5 int dist[N];
6 bool st[N];
7
8 int prim()
9 {
10     memset(dist, 0x3f, sizeof dist);
11
12     int res = 0;
13     for (int i = 0; i < n; i++)
14     {
15         int t = -1;
16         for (int j = 1; j <= n; j++)
17             if (!st[j] && (t == -1 || dist[t] > dist[j]))
18                 t = j;
19
20         if (i && dist[t] == INF) return INF;
21
22         if (i) res += dist[t];
23         st[t] = true;
24
25         for (int j = 1; j <= n; j++) dist[j] = min(dist[j], g[t][j]);
26     }
27
28     return res;
29 }
30
31
32 int main()
33 {
34     scanf("%d%d", &n, &m);
35
36     memset(g, 0x3f, sizeof g);
37
38     while (m--)
39     {
40         int a, b, c;
41         scanf("%d%d%d", &a, &b, &c);
42         g[a][b] = g[b][a] = min(g[a][b], c);
43     }
44
45     int t = prim();
46
47     if (t == INF) puts("impossible");
48     else printf("%d\n", t);
49
50     return 0;
51 }
```

## Kruskal算法

时间复杂度\$O(m\log m)\$, n表示点数, m表示边数

```

1 int n, m;           // n是点数, m是边数
2 int p[N];          // 并查集的父节点数组
3
4 struct Edge        // 存储边
5 {
6     int a, b, w;
7     bool operator< (const Edge &W) const
8     {
9         return w < W.w;
10    }
11 }edges[M];
12
13 int find(int x)    // 并查集核心操作
14 {
15     if (p[x] != x) p[x] = find(p[x]);
16     return p[x];
17 }
18
19 int kruskal()
20 {
21     sort(edges, edges + m);
22
23     for (int i = 1; i <= n; i++) p[i] = i;      // 初始化并查集
24
25     int res = 0, cnt = 0;
26     for (int i = 0; i < m; i++)
27     {
28         int a = edges[i].a, b = edges[i].b, w = edges[i].w;
29
30         a = find(a), b = find(b);
31         if (a != b)      // 如果两个连通块不连通, 则将这两个连通块合并
32         {
33             p[a] = b;
34             res += w;
35             cnt++;
36         }
37     }
38     if (cnt < n - 1) return INF;
39     return res;
40 }

```

## 应用

```

1 #include <cstring>
2 #include <iostream>
3 #include <algorithm>
4
5 using namespace std;
6
7 const int N = 100010, M = 200010, INF = 0x3f3f3f3f;
8
9 int n, m;
10 int p[N];
11
12 struct Edge
13 {
14     int a, b, w;
15
16     bool operator< (const Edge &W) const
17     {

```

```

18     return w < W.w;
19 }
20 }edges[M];
21
22 int find(int x)
23 {
24     if (p[x] != x) p[x] = find(p[x]);
25     return p[x];
26 }
27
28 int kruskal()
29 {
30     sort(edges, edges + m);
31
32     for (int i = 1; i <= n; i++) p[i] = i; // 初始化并查集
33
34     int res = 0, cnt = 0;
35     for (int i = 0; i < m; i++)
36     {
37         int a = edges[i].a, b = edges[i].b, w = edges[i].w;
38
39         a = find(a), b = find(b);
40         if (a != b)
41         {
42             p[a] = b;
43             res += w;
44             cnt++;
45         }
46     }
47
48     if (cnt < n - 1) return INF;
49     return res;
50 }
51
52 int main()
53 {
54     scanf("%d%d", &n, &m);
55
56     for (int i = 0; i < m; i++)
57     {
58         int a, b, w;
59         scanf("%d%d%d", &a, &b, &w);
60         edges[i] = {a, b, w};
61     }
62
63     int t = kruskal();
64
65     if (t == INF) puts("impossible");
66     else printf("%d\n", t);
67
68     return 0;
69 }

```

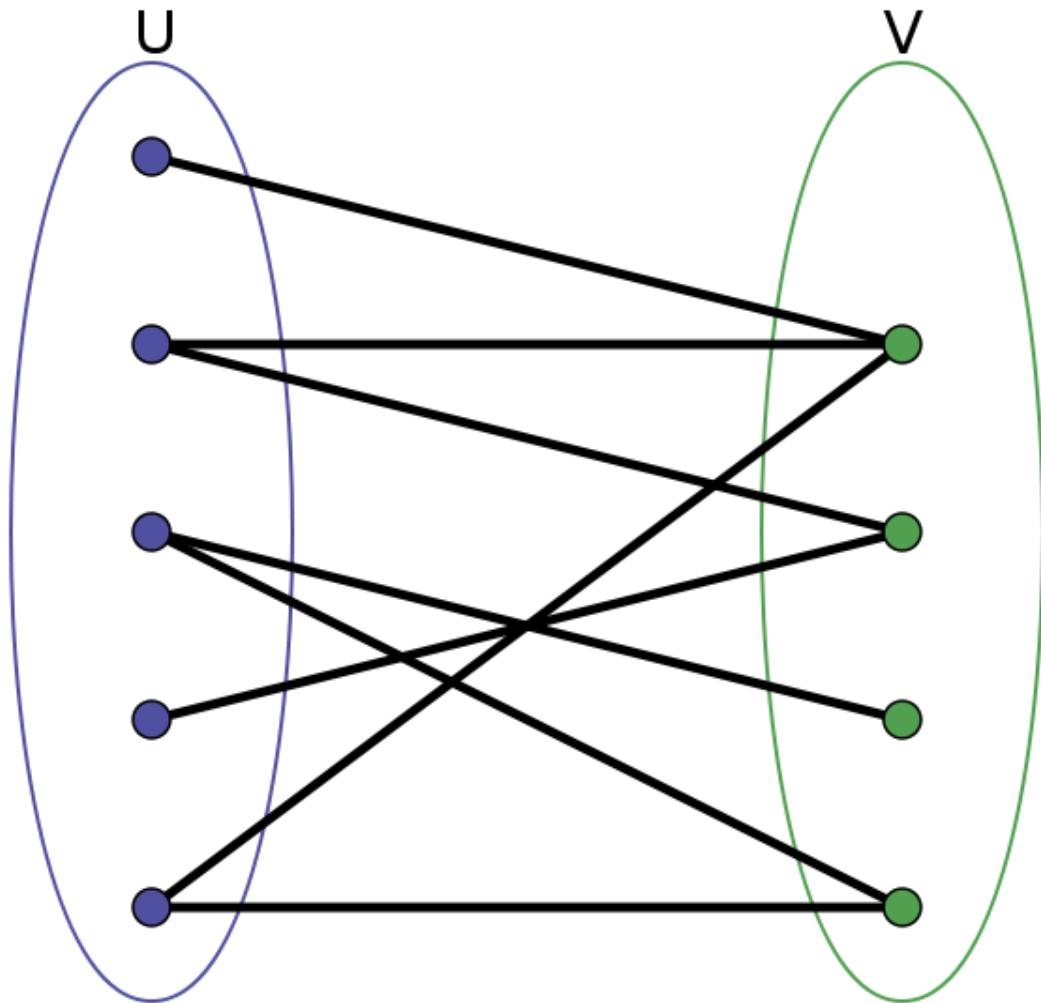
## 染色法判别二分图

什么叫二分图

| 有两顶点集且图中每条边的两个顶点分别位于两个顶点集中，每个顶点集中没有边直接相连接！

说人话的定义：图中点通过移动能分成左右两部分，左侧的点只和右侧的点相连，右侧的点只和左侧的点相连。

下图就是个二分图：



时间复杂度是\$O(n+m)\$，n表示点数，m表示边数

```
1 int n;      // n表示点数
2 int h[N], e[M], ne[M], idx;      // 邻接表存储图
3 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
4
5 // 参数: u表示当前节点，c表示当前点的颜色
6 bool dfs(int u, int c)
7 {
8     color[u] = c;
9     for (int i = h[u]; i != -1; i = ne[i])
10    {
11        int j = e[i];
12        if (color[j] == -1)
13        {
14            if (!dfs(j, !c)) return false;
15        }
16        else if (color[j] == c) return false;
17    }
18
19    return true;
20 }
```

```

21
22     bool check()
23 {
24     memset(color, -1, sizeof color);
25     bool flag = true;
26     for (int i = 1; i <= n; i++)
27     {
28         if (color[i] == -1)
29         {
30             if (!dfs(i, 0))
31             {
32                 flag = false;
33                 break;
34             }
35         }
36     }
37     return flag;
38 }
```

## 应用

```

1 using namespace std;
2
3 const int N = 100010, M = 200010;// 由于是无向图，顶点数最大是N，那么边数M最大是顶点数的2倍
4
5 int n, m;
6 int h[N], e[M], ne[M], idx;
7 int color[N];
8
9 void add(int a, int b)
10{
11    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
12}
13
14 bool dfs(int u, int c)
15{
16    color[u] = c;
17
18    for (int i = h[u]; i != -1; i = ne[i])
19    {
20        int j = e[i];
21        if (!color[j])
22        {
23            if (!dfs(j, 3 - c)) return false;
24        }
25        else if (color[j] == c) return false;
26    }
27
28    return true;
29}
30
31 int main()
32{
33    scanf("%d%d", &n, &m);
34
35    memset(h, -1, sizeof h);
36
37    while (m--)
38    {
39        int a, b;
40        scanf("%d%d", &a, &b);
41        add(a, b), add(b, a); // 无向图, a->b, b->a
42    }
43}
```

```

43     bool flag = true;
44     for (int i = 1; i <= n; i++)
45         if (!color[i])
46         {
47             if (!dfs(i, 1))
48             {
49                 flag = false;
50                 break;
51             }
52         }
53     }
54
55     if (flag) puts("Yes");
56     else puts("No");
57
58     return 0;
59 }
```

## 匈牙利算法

要了解匈牙利算法必须先理解下面的概念：

**匹配：**在图论中，一个「匹配」是一个边的集合，其中任意两条边都没有公共顶点。

**最大匹配：**一个图所有匹配中，所含匹配边数最多的匹配，称为这个图的最大匹配。

下面是一些补充概念：

**完美匹配：**如果一个图的某个匹配中，所有的顶点都是匹配点，那么它就是一个完美匹配。

**交替路：**从一个未匹配点出发，依次经过非匹配边、匹配边、非匹配边...形成的路径叫交替路。

**增广路：**从一个未匹配点出发，走交替路，如果途径另一个未匹配点（出发的点不算），则这条交替路称为增广路（augmenting path）。

时间复杂度\$O(nm)\$，n表示点数，m表示边数

```

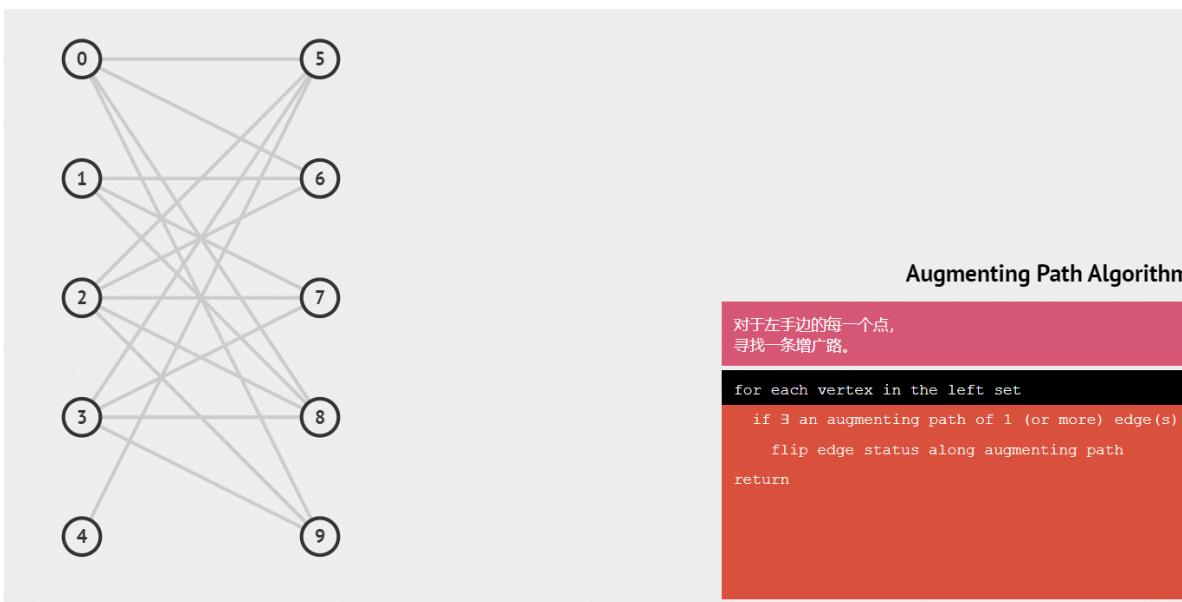
1 //遍历自己喜欢的女孩 int n1, n2;      // n1表示第一个集合中的点数, n2表示第二个集合中的点数
2 int h[N], e[M], ne[M], idx;      // 邻接表存储所有边，匈牙利算法中只会用到从第一个集合指向第二个集合的边，所以这里只用存一个方向的边
3 int match[N];      // 存储第二个集合中的每个点当前匹配的第一个集合中的点是哪个
4 bool st[N];      // 表示第二个集合中的每个点是否已经被遍历过
5
6 void add(int a, int b)
7 {
8     e[idx] = b, ne[idx] = h[a], h[a] = idx++;
9 }
10
11
12 bool find(int x)
13 {
14     //遍历自己喜欢的女孩
15     for (int i = h[x]; i != -1; i = ne[i])
16     {
17         int j = e[i];
18         if (!st[j])//如果在这一轮模拟匹配中，这个女孩尚未被预定
19         {
```

```

20     st[j] = true; // 那 x 就预定这个女孩了
21     // 如果女孩 j 没有男朋友，或者她原来的男朋友能够预定其它喜欢的女孩。配对成功
22     if (match[j] == 0 || find(match[j]))
23     {
24         match[j] = x;
25         return true;
26     }
27 }
28 }
29 // 自己中意的全部都被预定了。配对失败。
30 return false;
31 }
32
33 // 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
34 int res = 0;
35 for (int i = 1; i <= n1; i++)
36 {
37     memset(st, false, sizeof st);
38     if (find(i)) res++;
39 }

```

## 应用：二分图的最大匹配



相关题解：[AcWing 861. 二分图的最大匹配----图解--\\$color{red}{海绵宝宝来喽}\\$\(转\) - AcWing](#)

```

1 using namespace std;
2
3 const int N = 510, M = 100010;
4
5 int n1, n2, m;
6 int h[N], e[M], ne[M], idx;
7 int match[N];
8 bool st[N];
9
10 void add(int a, int b)
11 {
12     e[idx] = b, ne[idx] = h[a], h[a] = idx++;
13 }
14

```

```

15  bool find(int x)
16  {
17      // 和各个点尝试能否匹配
18      for (int i = h[x]; i != -1; i = ne[i])
19      {
20          int j = e[i];
21          if (!st[j])//打标记
22          {
23              st[j] = true;
24              // 当前尝试点没有被匹配或者和当前尝试点匹配的那个点可以换另一个匹配
25              if (match[j] == 0 || find(match[j]))
26              {
27                  // 和当前尝试点匹配在一起
28                  match[j] = x;
29                  return true;
30              }
31          }
32      }
33      return false;
34  }
35
36 int main()
37 {
38     scanf("%d%d%d", &n1, &n2, &m);
39
40     memset(h, -1, sizeof h);
41     // 保存图, 因为只从一遍找另一边, 所以该无向图只需要存储一个方向
42     while (m -- )
43     {
44         int a, b;
45         scanf("%d%d", &a, &b);
46         add(a, b);
47     }
48
49     int res = 0;
50     //为各个点找匹配
51     for (int i = 1; i <= n1; i ++ )
52     {
53         memset(st, false, sizeof st);
54         //找到匹配
55         if (find(i)) res ++ ;
56     }
57
58     printf("%d\n", res);
59
60     return 0;
61 }
```

## 四、数学知识

算法的数学知识定理证明可以在这里查阅：[数学部分简介 - OI Wiki \(oi-wiki.org\)](https://oi-wiki.org/math/)

### 试除法判定质数

```
1 bool is_prime(int x)
2 {
3     if (x < 2) return false;
4     for (int i = 2; i <= x / i; i++)
5         if (x % i == 0)
6             return false;
7     return true;
8 }
```

## 试除法分解质因数

```
1 void divide(int x)
2 {
3     for (int i = 2; i <= x / i; i++)
4         if (x % i == 0)//i 一定是质数
5             {
6                 int s = 0;
7                 while (x % i == 0) x /= i, s++;
8                 cout << i << ' ' << s << endl;
9             }
10            if (x > 1) cout << x << ' ' << 1 << endl;
11            cout << endl;
12 }
```

## 埃氏筛法求质数

```
1 int primes[N], cnt;           // primes[]存储所有素数
2 bool st[N];                  // st[x]存储x是否被筛掉
3
4 void get_primes(int n)
5 {
6     for (int i = 2; i <= n; i++)
7     {
8         if (st[i]) continue;
9         primes[cnt++] = i;
10        for (int j = i + i; j <= n; j += i)
11            st[j] = true;
12    }
13 }
```

## 线性筛法求质数

算法动画讲解：<https://www.bilibili.com/video/BV1LR4y1Z7pm>

```
1 int primes[N], cnt;      // primes[]存储所有素数
2 bool st[N];             // st[x]存储x是否被筛掉
3
4 void get_primes(int n)
{
    for (int i = 2; i <= n; i++)

```

```

7     {
8         if (!st[i]) primes[cnt ++ ] = i;
9         for (int j = 0; primes[j] <= n / i; j ++ )
10        {
11            st[primes[j] * i] = true;
12            if (i % primes[j] == 0) break;
13        }
14    }
15 }
```

## 试除法求所有约数

```

1 vector<int> get_divisors(int x)
2 {
3     vector<int> res;
4     for (int i = 1; i <= x / i; i ++ )
5         if (x % i == 0)
6         {
7             res.push_back(i);
8             if (i != x / i) res.push_back(x / i);
9         }
10    sort(res.begin(), res.end());
11    return res;
12 }
```

## 约数个数

约数个数定理和约数和定理公式推导: <https://www.bilibili.com/video/BV13R4ylo777>

约数个数定理推导: <https://www.bilibili.com/video/BV1NY41187GM>

$$360 = 2^3 \times 3^2 \times 5^1$$

$$\text{约数个数} = (3+1) \times (2+1) \times (1+1)$$

```

1 using namespace std;
2 typedef long long LL;
3 const int N = 110, mod = 1e9 + 7;
4 int main()
5 {
6     int n;
7     cin >> n;
8     unordered_map<int, int> primes;
9     while (n -- )
10    {
11        int x;
12        cin >> x;
13        for (int i = 2; i <= x / i; i ++ )
14            while (x % i == 0)
15            {
16                x /= i;
```

```

17         primes[i] ++ ;
18     }
19     if (x > 1) primes[x] ++ ;
20 }
21 LL res = 1;
22 for (auto p : primes) res = res * (p.second + 1) % mod;
23 cout << res << endl;
24 return 0;
25 }

```

## 约数之和

约数个数定理和约数和定理公式推导: <https://www.bilibili.com/video/BV13R4ylo777>

$$\because 360 = 2^3 \times 3^2 \times 5^1$$

$$\therefore \text{约数个数} = (3+1) \times (2+1) \times (1+1)$$

$$\text{约数之和} = (2^0 + 2^1 + 2^2 + 2^3) \times (3^0 + 3^1 + 3^2) \times (5^0 + 5^1)$$

```

1 using namespace std;
2 typedef long long LL;
3 const int N = 110, mod = 1e9 + 7;
4 int main()
5 {
6     int n;
7     cin >> n;
8     unordered_map<int, int> primes;
9     while (n -- )
10    {
11        int x;
12        cin >> x;
13        for (int i = 2; i <= x / i; i++)
14            while (x % i == 0)
15            {
16                x /= i;
17                primes[i]++;
18            }
19        if (x > 1) primes[x]++;
20    }
21    LL res = 1;
22    for (auto p : primes)
23    {
24        LL a = p.first, b = p.second;
25        LL t = 1;
26        while (b -- ) t = (t * a + 1) % mod; // 遍历b次后得到t=p^b+p^(b-1)+...+p+1
27        res = res * t % mod;
28    }
29    cout << res << endl;
30    return 0;
31 }

```

代码第26行解释:

**while(b--) t=a\*t+1;**

**当b=3时， 经过3次迭代如下**

**t=1**

**t=a+1**

**t=a\*(a+1)+1=a^2+a+1**

**t=a\*(a^2+a+1)+1=a^3+a^2+a+1**

## 欧几里得算法(求最大公约数)

```
1 int gcd(int a, int b)
2 {
3     return b ? gcd(b, a % b) : a;
4 }
```

## 求欧拉函数

### 前置知识

互质：互质是公约数只有1的两个整数，叫做互质整数。

### 欧拉函数定义

$\phi(N)$ 中与N互质的数的个数被称为欧拉函数，记为 $\phi(N)$ 。

若在算数基本定理中， $N=p_1^{a_1}p_2^{a_2}\dots p_m^{a_m}$ ，则：

$$\phi(N)=N\cdot\frac{p_1-1}{p_1}\cdot\frac{p_2-1}{p_2}\cdot\dots\cdot\frac{p_m-1}{p_m}$$

### 欧拉函数推导

首先我们要知道 $1,2,3,\dots,N-1$ 与 $N$ 互质的个数是 $N$ 数列去除 $N$ 的质因子的倍数。

例如  $N=10$ ，即  $1,2,3,4,5,6,7,8,9,10$  去除  $N$  的 质 因 子 的 倍 数  $2,4,5,6,8,10$ 。

显然， $1,3,7,9$ 与 $10$ 互质。

由上方结论使用容斥原理进行数学推导如下：

$$N=p_1^{a_1}p_2^{a_2}\dots p_m^{a_m}$$

①.从 $1-n$ 中去掉 $p_1, p_2, \dots, p_k$ 的所有倍数的个数，即

$$n - \frac{n}{p_1} - \frac{n}{p_2} - \dots - \frac{n}{p_k}$$

②.由容斥原理， $p_i \cdot p_j$ 的倍数被①减了两次，所以加上所有 $p_i \cdot p_j$ 的倍数的个数（其中 $p_i, p_j$ 是 $p_1 \sim p_k$ 的组合），即

$$n \leftarrow n + \frac{n}{p_1 \cdot p_2} + \frac{n}{p_1 \cdot p_3} + \dots + \frac{n}{p_1 \cdot p_{k-1} \cdot p_k}$$

③.减去所有 $p_i \cdot p_j \cdot p_k$ 的倍数个数，即

$$n \leftarrow n - \frac{n}{p_1 \cdot p_2 \cdot p_3} - \frac{n}{p_1 \cdot p_2 \cdot p_4} - \dots - \frac{n}{p_{k-2} \cdot p_{k-1} \cdot p_k}$$

④.同理，加上所有 $p_i \cdot p_j \cdot p_k \cdot p_l$ 的倍数个数，即

$$n \leftarrow n + \frac{n}{p_1 \cdot p_2 \cdot p_3 \cdot p_4} + \frac{n}{p_1 \cdot p_2 \cdot p_3 \cdot p_5} + \dots + \frac{n}{p_{k-3} \cdot p_{k-2} \cdot p_{k-1} \cdot p_k}$$

⋮

因此，

$$\begin{aligned} \phi(n) &= n - \frac{n}{p_1} - \frac{n}{p_2} - \dots - \frac{n}{p_k} \\ &\quad + \frac{n}{p_1 \cdot p_2} + \frac{n}{p_1 \cdot p_3} + \dots + \frac{n}{p_{k-1} \cdot p_k} \\ &\quad - \frac{n}{p_1 \cdot p_2 \cdot p_3} - \frac{n}{p_1 \cdot p_2 \cdot p_4} - \dots - \frac{n}{p_{k-2} \cdot p_{k-1} \cdot p_k} \\ &\quad + \frac{n}{p_1 \cdot p_2 \cdot p_3 \cdot p_4} + \frac{n}{p_1 \cdot p_2 \cdot p_3 \cdot p_5} + \dots + \frac{n}{p_{k-3} \cdot p_{k-2} \cdot p_{k-1} \cdot p_k} \\ &\quad - \dots \end{aligned}$$

也就是 $n$ 减去奇数个质因子的倍数个数，加上偶数个质因子的倍数个数，循环往复。

将上式等价变形，得到

$$\phi(n) = n \cdot (1 - \frac{1}{p_1}) \cdot (1 - \frac{1}{p_2}) \cdots \cdot (1 - \frac{1}{p_k})$$

证必。

## 代码模板

```

1 int phi(int x)
2 {
3     int res = x;
4     for (int i = 2; i <= x / i; i++)
5         if (x % i == 0)
6             {
7                 res = res / i * (i - 1);
8                 while (x % i == 0) x /= i;
9             }
10    if (x > 1) res = res / x * (x - 1);
11
12    return res;
13}

```

## 线性筛法求欧拉函数

```

1 int primes[N], cnt;      // primes[]存储所有素数
2 int euler[N];           // 存储每个数的欧拉函数
3 bool st[N];              // st[x]存储x是否被筛掉
4
5 void get_eulers(int n)   // 线性筛法求1~n的欧拉函数

```

```

6  {
7      euler[1] = 1;
8      for (int i = 2; i <= n; i++)
9      {
10         if (!st[i])
11         {
12             primes[cnt++] = i;
13             euler[i] = i - 1;
14         }
15         for (int j = 0; primes[j] <= n / i; j++)
16         {
17             int t = primes[j] * i;
18             st[t] = true;
19             if (i % primes[j] == 0)
20             {
21                 euler[t] = euler[i] * primes[j];
22                 break;
23             }
24             euler[t] = euler[i] * (primes[j] - 1);
25         }
26     }
27 }

```

## 快速幂

快速幂公式证明：[快速幂 - OI Wiki \(oi-wiki.org\)](#)

```

1 // 求 m^k mod p, 时间复杂度 O(logk)。
2 // m为底数, k为幂
3 int qmi(int m, int k, int p)
4 {
5     int res = 1 % p, t = m;
6     while (k)
7     {
8         if (k&1) res = res * t % p;
9         t = t * t % p;
10        k >= 1;
11    }
12    return res;
13 }

```

## 扩展欧几里得算法

[扩展欧几里得算法讲解](#)：<https://www.bilibili.com/video/BV1KU4y1a7E2/>

[优秀题解](#)：<https://www.acwing.com/solution/content/1393>

[优秀博客](#)：<https://blog.csdn.net/mango114514/article/details/121048335>

x的第一个正解就是 $(x \% k + k) \% k$

其中， $k = b / \text{gcd}(a, b)$

```

1 // 求x, y, 使得ax + by = gcd(a, b)
2 int exgcd(int a, int b, int &x, int &y)
3 {
4     if (!b)
5     {
6         x = 1, y = 0;
7         return a;
8     }
9     int d = exgcd(b, a % b, y, x);
10    y -= (a/b) * x;
11    return d;
12 }

```

## 中国剩余定理

中国剩余定理讲解: <https://www.bilibili.com/video/BV1AN4y1N7Su/>

### 《孙子算经》

有物不知其数，三三数之剩二，五五数之剩三，七七数之剩二。问物几何？

求解线性同余方程组

$$\begin{cases} x \equiv r_1 \pmod{m_1} \\ x \equiv r_2 \pmod{m_2} \\ \vdots \\ x \equiv r_n \pmod{m_n} \end{cases}$$

其中模数  $m_1, m_2, \dots, m_n$  为两两互质的整数，求  $x$  的最小非负整数解。

中国剩余定理 (Chinese Remainder Theorem, CRT)

1. 计算所有模数的积  $M$

2. 计算第  $i$  个方程的  $c_i = \frac{M}{m_i}$

3. 计算  $c_i$  在模  $m_i$  意义下的逆元  $c_i^{-1}$

4.  $x = \sum_{i=1}^n r_i c_i c_i^{-1} \pmod{M}$

例  $\begin{cases} x \equiv 2 \pmod{3} \\ x \equiv 3 \pmod{4} \\ x \equiv 1 \pmod{5} \end{cases}$

1.  $M = 3 \times 4 \times 5 = 60$

2.  $c_1 = 20, c_1^{-1} = 2$ , 因  $20x \equiv 1 \pmod{3}$

$c_2 = 15, c_2^{-1} = 3$ , 因  $15x \equiv 1 \pmod{4}$

$c_3 = 12, c_3^{-1} = 3$ , 因  $12x \equiv 1 \pmod{5}$

3.  $x = (2 \times 20 \times 2 + 3 \times 15 \times 3 + 1 \times 12 \times 3) \% 60$   
 $= 251 \% 60 = 11$

证明:

先证明  $\sum_{i=1}^n r_i c_i c_i^{-1}$  满足每个  $x \equiv r_i \pmod{m_i}$ 。

当  $i \neq j$  时,  $c_j \equiv 0 \pmod{m_i}$ , 则  $c_j c_j^{-1} \equiv 0 \pmod{m_i}$

当  $i = j$  时,  $c_i \not\equiv 0 \pmod{m_i}$ , 则  $c_i c_i^{-1} \equiv 1 \pmod{m_i}$

故  $x \equiv \sum_{j=1}^n r_j c_j c_j^{-1} \pmod{m_i}$   
 $\equiv r_i c_i c_i^{-1} \pmod{m_i}$   
 $\equiv r_i \pmod{m_i}$

而  $\sum_{i=1}^n r_i c_i c_i^{-1} \pmod{M}$  对  $m_i$  来说, 只是减去了  $m_i$  的若干倍, 不影响余数  $r_i$ 。证毕

```

1 LL exgcd(LL a,LL b,LL &x,LL &y){
2     if(b==0){
3         x=1,y=0;
4         return a;
5     }
6     LL d=exgcd(b,a%b,y,x);
7     y -= (a/b) * x;
8     return d;
9 }
10 LL CRT(LL m[],LL r[]){
11     LL m=1,ans=0;
12     for(int i=1;i<=n;i++) M*=m[i];
13     for(int i=1;i<=n;i++){
14         LL c=M/m[i],x,y;
15         exgcd(c,m[i],x,y);
16         ans=(ans+r[i]*c*x%M)%M;
17     }
18     return (ans%M+M)%M;
19 }

```

# 扩展中国剩余定理

扩展中国剩余定理讲解: <https://www.bilibili.com/video/BV1Ut4y1F7HG/>

## 问题

求解线性同余方程组

$$\begin{cases} x \equiv r_1 \pmod{m_1} \\ x \equiv r_2 \pmod{m_2} \\ \dots \\ x \equiv r_n \pmod{m_n} \end{cases}$$

其中  $m_1, m_2, \dots, m_n$  为不一定两两互质的整数，求  $x$  的最小非负整数解。

## 中国剩余定理 (CRT) 已不可行

其构造解  $x = \sum_{i=1}^n r_i c_i c_i^{-1} \pmod{M}$

其中  $c_i x \equiv 1 \pmod{m_i}$ , 即  $c_i x + m_i y = 1 = \gcd(c_i, m_i)$

根据裴蜀定理,  $c_i, m_i$  应该互质,  $c_i = \frac{m_1 \times \dots \times m_n}{m_i}$

如果  $c_i, m_i$  不互质, 则  $c_i^{-1}$  不存在, 算法失效

## 扩展中国剩余定理 (EXCRT)

前两个方程:  $x \equiv r_1 \pmod{m_1}, x \equiv r_2 \pmod{m_2}$

转化为不定方程:  $x = m_1 p + r_1 = m_2 q + r_2$

则  $m_1 p - m_2 q = r_2 - r_1$

由裴蜀定理,

当  $\gcd(m_1, m_2) \nmid (r_2 - r_1)$  时, 无解

当  $\gcd(m_1, m_2) \mid (r_2 - r_1)$  时, 有解

由扩欧算法,

得特解  $p = p * \frac{r_2 - r_1}{\gcd}, q = q * \frac{r_2 - r_1}{\gcd}$

其通解  $P = p + \frac{m_2}{\gcd} * k, Q = q - \frac{m_1}{\gcd} * k$

所以  $x = m_1 P + r_1 = \frac{m_1 m_2}{\gcd} * k + m_1 p + r_1$

前两个方程等价合并为一个方程  $x \equiv r \pmod{m}$

其中  $r = m_1 p + r_1, m = \text{lcm}(m_1, m_2)$

所以  $n$  个同余方程只要合并  $n - 1$  次, 即可求解

CSDN @极光

```
1 LL exgcd(LL a,LL b,LL &x,LL &y){  
2     if(b==0){  
3         x=1,y=0;  
4         return a;  
5     }  
6     LL d=exgcd(b,a%b,y,x);  
7     y -= (a/b) * x;  
8     return d;  
9 }  
10 LL EXCRT(LL m[],LL r[]){  
11     LL m1,m2,r1,r2,p,q;  
12     m1=m[1],r1=r[1];  
13     for(int i=2;i<=n;i++){  
14         m2=m[i],r2=r[i];  
15         LL d = exgcd(m1,m2,p,q);  
16         if((r2-r1)%d){  
17             return -1;  
18         }  
19         p=p*(r2-r1)/d;//特解  
20         p=(p%(m2/d)+m2/d)%(m2/d);  
21         r1=m1*p+r1;  
22         m1=m1*m2/d;  
23     }  
24     return (r1%m1+m1)%m1;  
25 }
```

# 高斯消元法

高斯消元  $O(n^3)$

求解例如下面方程组

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{cases}$$

高斯消元讲解: <https://www.bilibili.com/video/BV1Kd4y127vZ/>

## 模板

```

1 // a[N][N]是增广矩阵
2 int gauss()
3 {
4     int c, r;
5     for (c = 0, r = 0; c < n; c++)
6     {
7         int t = r;
8         for (int i = r; i < n; i++) // 找到绝对值最大的行
9             if (fabs(a[i][c]) > fabs(a[t][c]))
10                t = i;
11
12         if (fabs(a[t][c]) < eps) continue;
13
14         for (int i = c; i <= n; i++) swap(a[t][i], a[r][i]); // 将绝对值最大的行换到最顶端
15         for (int i = n; i >= c; i--) a[r][i] /= a[r][c]; // 将当前行的首位变成1
16         for (int i = r + 1; i < n; i++) // 用当前行将下面所有的列消成0
17             if (fabs(a[i][c]) > eps)
18                 for (int j = n; j >= c; j--)
19                     a[i][j] -= a[r][j] * a[i][c];
20
21         r++;
22     }
23
24     if (r < n)
25     {
26         for (int i = r; i < n; i++)
27             if (fabs(a[i][n]) > eps)
28                 return 2; // 无解
29         return 1; // 有无穷多组解
30     }
31
32     for (int i = n - 1; i >= 0; i--)
33         for (int j = i + 1; j < n; j++)
34             a[i][n] -= a[i][j] * a[j][n];
35
36     return 0; // 有唯一解
37 }
```

## 应用

```

1 using namespace std;
2
3 const int N = 110;
4 const double eps = 1e-6;
5
6 int n;
7 double a[N][N];
8
9 int gauss()
10 {
```

```

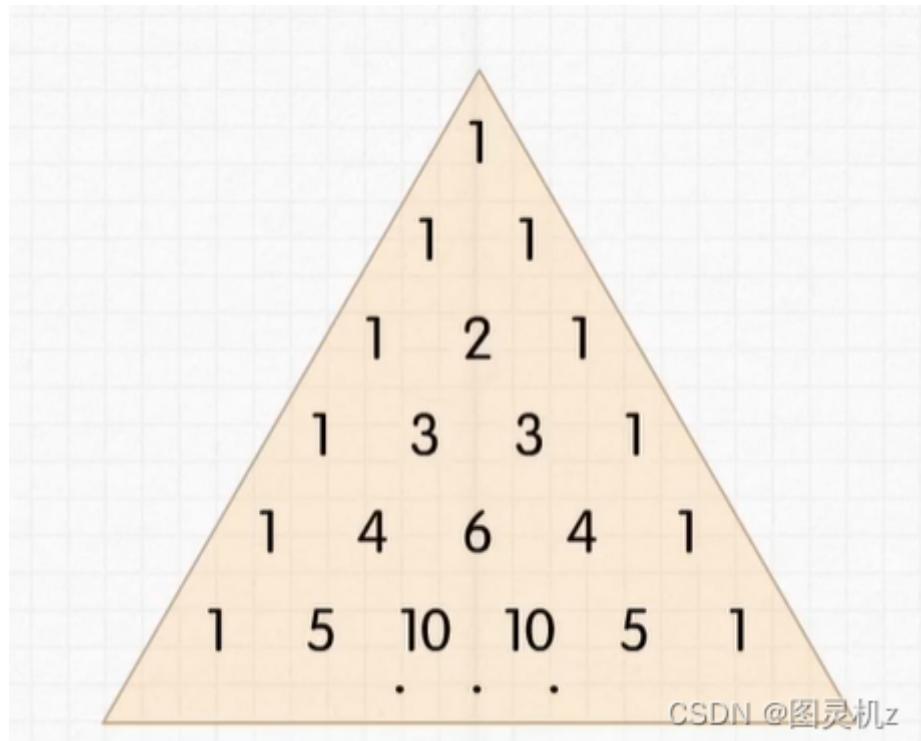
11  int c, r; // c 代表列 col , r 代表行 row
12  for (c = 0, r = 0; c < n; c++)
13  {
14      int t = r; // 先找到当前这一列，绝对值最大的一个数字所在的行号
15      for (int i = r; i < n; i++)
16          if (fabs(a[i][c]) > fabs(a[t][c]))
17              t = i;
18
19      if (fabs(a[t][c]) < eps) continue; // 如果当前这一列的最大数都是 0，那么所有
20      // 数都是 0，就没必要去算了，因为它的约束方程，可能在上面几行
21
22      for (int i = c; i < n + 1; i++) swap(a[t][i], a[r][i]); // 把当前这一
23      // 行，换到最上面（不是第一行，是第 r 行）去
24      for (int i = n; i >= c; i--) a[r][i] /= a[r][c]; // 把当前这一行的第一个
25      // 数，变成 1，方程两边同时除以第一个数，必须要到着算，不然第一个数直接变1，系数就被篡改，
26      // 后面的数字没法算
27      for (int i = r + 1; i < n; i++) // 把当前列下面的所有数，全部消成 0
28          if (fabs(a[i][c]) > eps) // 如果非0 再操作，已经是 0 就没必要操作了
29              for (int j = n; j >= c; j--) // 从后往前，当前行的每个数字，都减去对
30              // 应列 * 行首非0的数字，这样就能保证第一个数字是 a[i][0] -= 1*a[i][0];
31              a[i][j] -= a[r][j] * a[i][c];
32
33      r++; // 这一行的工作做完，换下一行
34
35  if (r < n) // 说明剩下方程的个数是小于 n 的，说明不是唯一解，判断是无解还是无穷多解
36  { // 因为已经是阶梯型，所以 r ~ n-1 的值应该都为 0
37      for (int i = r; i < n; i++)
38          if (fabs(a[i][n]) > eps) // a[i][n] 代表 b_i ,即 左边=0, 右边=b_i, 0 != b_i, 所以无解。
39              return 2;
40      return 1; // 否则，0 = 0，就是r ~ n-1的方程都是多余方程
41
42  // 唯一解 ↓，从下往上回代，得到方程的解
43  for (int i = n - 1; i >= 0; i--)
44      for (int j = i + 1; j < n; j++)
45          a[i][n] -= a[j][n] * a[i][j]; // 因为只要得到解，所以只用对 b_i 进行操作，
46      // 中间的值，可以不用操作，因为不用输出
47
48  return 0;
49}
50
51 int main()
52 {
53     cin >> n;
54     for (int i = 0; i < n; i++)
55         for (int j = 0; j < n + 1; j++)
56             cin >> a[i][j];
57
58     int t = gauss();
59
60     if (t == 0)
61     {
62         for (int i = 0; i < n; i++) printf("%.2lf\n", a[i][n]);
63     }
64     else if (t == 1) puts("Infinite group solutions");
65     else puts("No solution");
66     return 0;
67 }

```

# 求组合数

## 递推法求组合数

排列组合详细讲解: <https://www.bilibili.com/video/BV1e7411J7SC/>



$$\begin{array}{c} C_0^0 \\ C_1^0 \quad C_1^1 \\ C_2^0 \quad C_2^1 \quad C_2^2 \\ C_3^0 \quad C_3^1 \quad C_3^2 \quad C_3^3 \\ C_4^0 \quad C_4^1 \quad C_4^2 \quad C_4^3 \quad C_4^4 \end{array}$$

1. 左右两侧斜线都是1,  $C^0_n = C^n_{n-1}$
2. 其他数等于其左上角和右上角两数之和  $C^m_{n-1} + C^m_{n-2} = C^{m+1}_n$

```
1 // c[a][b] 表示从a个苹果中选b个的方案数
2 int c[N][N];
3 for (int i = 0; i < N; i++)
4     for (int j = 0; j <= i; j++)
5         if (!j) c[i][j] = 1;
6         else c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % mod;
7 //本质上杨辉三角
```

# 通过预处理逆元的方式求组合数

## 模板

```
1 // 首先预处理出所有阶乘取模的余数fact[N]，以及所有阶乘取模的逆元infact[N]
2 // 如果取模的数是质数，可以用费马小定理求逆元
3 int qmi(int a, int k, int p)    // 快速幂模板
4 {
5     int res = 1;
6     while (k)
7     {
8         if (k & 1) res = (LL)res * a % p;
9         a = (LL)a * a % p;
10        k >>= 1;
11    }
12    return res;
13 }
14
15 // 预处理阶乘的余数和阶乘逆元的余数
16 fact[0] = infact[0] = 1;
17 for (int i = 1; i < N; i++)
18 {
19     fact[i] = (LL)fact[i - 1] * i % mod;
20     infact[i] = (LL)infact[i - 1] * qmi(i, mod - 2, mod) % mod;
21 }
```

## 应用

```
1 using namespace std;
2
3 typedef long long LL;
4
5 const int N = 100010,mod=1e9+7;//1e9+7是质数所以与[1,1e9+7)中的数互质
6
7 int fact[N],in fact[N];
8
9 int qmi(int a,int k,int p){
10     int res=1;
11     while(k){
12         if(k&1)res=(LL)res*a%p;
13         a=(LL)a*a%p;
14         k>>=1;
15     }
16     return res;
17 }
18
19 int main()
20 {
21     fact[0]=in fact[0]=1;
22     for (int i = 1; i <= N; i++){
23         fact[i]=(LL)fact[i-1]*i%mod;
24         in fact[i]=(LL)in fact[i-1]*qmi(i,mod-2,mod)%mod;
25     }
26
27     int n;
28     scanf("%d",&n);
29     while (n -- ){
30         int a,b;
31         scanf("%d%d", &a, &b);
```

```

32     printf("%d\n", (LL)fact[a]*infact[b]%mod*infact[a-b]%mod);
33 }
34 return 0;
35 }
```

## Lucas定理求组合数

Lucas定理证明: [https://blog.csdn.net/Qiuker\\_jl/article/details/109528164](https://blog.csdn.net/Qiuker_jl/article/details/109528164)

### 模板

```

1 // 若p是质数, 则对于任意整数 1 <= m <= n, 有:
2 // C(n, m) = C(n % p, m % p) * C(n / p, m / p) (mod p)
3
4 int qmi(int a, int k, int p) // 快速幂模板
{
5     int res = 1 % p;
6     while (k)
7     {
8         if (k & 1) res = (LL)res * a % p;
9         a = (LL)a * a % p;
10        k >>= 1;
11    }
12    return res;
13}
14
15
16 int C(int a, int b, int p) // 通过定理求组合数C(a, b)
17 {
18     if (a < b) return 0;
19
20     LL x = 1, y = 1; // x是分子, y是分母
21     for (int i = a, j = 1; j <= b; i --, j ++ )
22     {
23         x = (LL)x * i % p;
24         y = (LL) y * j % p;
25     }
26
27     return x * (LL)qmi(y, p - 2, p) % p;
28 }
29
30 int lucas(LL a, LL b, int p)
31 {
32     if (a < p && b < p) return C(a, b, p);
33     return (LL)C(a % p, b % p, p) * lucas(a / p, b / p, p) % p;
34 }
```

### 应用

```

1 using namespace std;
2
3 typedef long long LL;
4
5 int qmi(int a,int k,int p)
6 {
7     int res = 1;
8     while(k)
```

```

9     {
10        if(k&1)res = (LL)res*a%p;
11        a = (LL)a*a%p;
12        k>>=1;
13    }
14    return res;
15 }
16
17 int C(int a,int b,int p)//自变量类型int
18 {
19    if(b>a)return 0;//漏了边界条件
20    int res = 1;
21    // a!/(b!(a-b)!) = (a-b+1)*...*a / b! 分子有b项
22    for(int i=1,j=a;i<=b;i++,j--)//i<=b而不是<
23    {
24        res = (LL)res*j%p;
25        res = (LL)res*qmi(i,p-2,p)%p;
26    }
27    return res;
28 }
29 //对公式敲
30 int lucas(LL a,LL b,int p)
31 {
32    if(a<p && b<p)return C(a,b,p);//lucas递归终点是C_{bk}^{ak}
33    return (LL)C(a%p,b%p,p)*lucas(a/p,b/p,p)%p;//a%p后肯定是<p的,所以可以用C(),但a/p
后不一定<p 所以用lucas继续递归
34 }
35
36 int main()
37 {
38    int n;
39    cin >> n;
40    while(n--)
41    {
42        LL a,b;
43        int p;
44        cin >> a >> b >> p;
45        cout << lucas(a,b,p) << endl;
46    }
47    return 0;
48 }

```

## 分解质因数法求组合数

### 模板

1 | 当我们需求求出组合数的真实值，而非对某个数的余数时，分解质因数的方式比较好用：

2 | 1. 筛法求出范围内的所有质数

3 | 2. 通过  $C(a, b) = a! / b! / (a - b)!$  这个公式求出每个质因子的次数。  $n!$  中  $p$  的次数是  $n / p + n / p^2 + n / p^3 + \dots$

4 | 3. 用高精度乘法将所有质因子相乘

5 |

6 | int primes[N], cnt; // 存储所有质数

7 | int sum[N]; // 存储每个质数的次数

8 | bool st[N]; // 存储每个数是否已被筛掉

9 |

10 |

```

11 void get_primes(int n)      // 线性筛法求素数
12 {
13     for (int i = 2; i <= n; i++)
14     {
15         if (!st[i]) primes[cnt++] = i;
16         for (int j = 0; primes[j] <= n / i; j++)
17         {
18             st[primes[j] * i] = true;
19             if (i % primes[j] == 0) break;
20         }
21     }
22 }
23
24
25 int get(int n, int p)      // 求n! 中的次数
26 {
27     int res = 0;
28     while (n)
29     {
30         res += n / p;
31         n /= p;
32     }
33     return res;
34 }
35
36
37 vector<int> mul(vector<int> a, int b)      // 高精度乘低精度模板
38 {
39     vector<int> c;
40     int t = 0;
41     for (int i = 0; i < a.size(); i++)
42     {
43         t += a[i] * b;
44         c.push_back(t % 10);
45         t /= 10;
46     }
47
48     while (t)
49     {
50         c.push_back(t % 10);
51         t /= 10;
52     }
53
54     return c;
55 }
56
57 get_primes(a); // 预处理范围内的所有质数
58
59 for (int i = 0; i < cnt; i++)      // 求每个质因数的次数
60 {
61     int p = primes[i];
62     sum[i] = get(a, p) - get(b, p) - get(a - b, p);
63 }
64
65 vector<int> res;
66 res.push_back(1);
67
68 for (int i = 0; i < cnt; i++)      // 用高精度乘法将所有质因子相乘
69     for (int j = 0; j < sum[i]; j++)
70         res = mul(res, primes[i]);

```

## 应用

```
1 using namespace std;
2
3 const int N = 5010;
4 int primes[N],cnt=0;
5 // v[i] 记录数字 i 为素数还是合数, v[i]=true时 i 为合数, 否则 i 为素数
6 bool v[N];
7 // sum[i]=c 表示质数 i 的个数为 c
8 int sum[N];
9
10 // 线性筛法
11 void get_primes(int n)
12 {
13     for(int i=2;i<=n;++i)
14     {
15         // i为质数, 则存在primes中
16         if(!v[i])primes[cnt++]=i;
17         // 给当前数i乘上一个质因子pj
18         for(int j=0;primes[j]<=n/i;++j)
19         {
20             v[primes[j]*i]=true;
21             if(i%primes[j]==0)break;
22         }
23     }
24 }
25
26 // 计算 n 里面含有质数 p 的个数, 这里的计算是不重不漏的。
27 // p^k的倍数会被计算k次: 第一次算p的倍数时, 被加一次; 第二次算p^2的倍数时, 被加一次; 第三次算p^3的倍数时, 被加一次...第k次算p^k的倍数时, 被加一次。总共被加了k次, 是不重不漏的。
28 int get(int n,int p)
29 {
30     int res=0;
31     while(n)
32     {
33         res+=n/p;
34         n/=p;
35     }
36     return res;
37 }
38
39 // A * b: 把 b 看成一个整体, 然后与 A 中每一位相乘, A中的数字采用小端存储, 即低位数字存储在数组的前面, 高位数字存储在数组的后面
40 vector<int> mul(const vector<int>& A,const int b)
41 {
42     if(b==0)return {0};
43     vector<int> res;
44     // t 表示乘法进位, 这里的进位不限于0 1, 可以为任意数字
45     for(int i=0,t=0,n=A.size();i<n||t>0;++i)
46     {
47         // 获得当前位的乘积和
48         if(i<n)t+=A[i]*b;
49         // 添加个位数字
50         res.push_back(t%10);
51         // 保留进位
52         t/=10;
53     }
54
55     // 如 1234 * 0 = 0000, 需要删除前导0
56     while(res.size()>1&&res.back()==0)res.pop_back();
57     return res;
```

```

58 }
59
60 int main()
61 {
62     int a,b;cin>>a>>b;
63
64     // 将 a 分解质因数
65     get_primes(a);
66
67     for(int i=0;i<cnt;++i)
68     {
69         // 当前的质数为 p
70         int p=primes[i];
71         // 用分子里面 p 的个数减去分母里面 p 的个数。这里的计算组合数的公式为 $a!/(b!*(a-b)!)$ ，因此用 a 里面 p 的个数减去 b 里面 p 的个数和 (a-b) 里面 p 的个数。
72         sum[i]=get(a,p)-get(b,p)-get(a-b,p);
73     }
74
75     // 使用高精度乘法把所有质因子乘到一块去就好了
76     vector<int> res={1};
77     for(int i=0;i<cnt;++i)
78         // res*p^k, 这里是k个p相乘，不是k*p，所以需要使用一个循环
79         for(int j=0;j<sum[i];++j)
80             res=mul(res,primes[i]);
81
82     // 倒序打印 res 即可，由于采用小端存储，所以高位在后，从后往前打印即可
83     for(int i=res.size()-1;i>=0;i--)printf("%d",res[i]);
84
85 }

```

## 容斥原理应用

经典例题：890. 能被整除的数 - AcWing题库

AC代码：

```

1  using namespace std;
2  typedef long long LL;
3
4  const int N = 20;
5  int p[N], n, m;
6
7  int main() {
8      cin >> n >> m;
9      for(int i = 0; i < m; i++) cin >> p[i];
10
11     int res = 0;
12     //枚举从1 到 1111... (m个1)的每一个集合状态，(至少选中一个集合)
13     for(int i = 1; i < 1 << m; i++) {
14         int t = 1;           //选中集合对应质数的乘积
15         int s = 0;           //选中的集合数量
16
17         //枚举当前状态的每一位
18         for(int j = 0; j < m; j++){
19             //选中一个集合
20             if(i >> j & 1){
21                 //乘积大于n，则n/t = 0，跳出这轮循环
22                 if((LL)t * p[j] > n){

```

```

23         t = -1;
24         break;
25     }
26     s++;           //有一个1, 集合数量+1
27     t *= p[j];
28 }
29 }
30
31 if(t == -1) continue;
32
33 if(s & 1) res += n / t;           //选中奇数个集合, 则系数应该是1, n/t为当
前这种状态的集合数量
34 else res -= n / t;               //反之则为 -1
35 }
36
37 cout << res << endl;
38 return 0;
39 }
```

**详细题解:** [AcWing 890. 能被整除的数 - AcWing](#)

## 博弈论

### NIM游戏

**定理1:** 必胜态的后继状态至少存在一个必败态

**定理2:** 必败态的后继状态均为必胜态

**NIM游戏科普:** 尼姆游戏 (学霸就是这样欺负人的) [哔哩哔哩bilibili](#)

再看nim游戏 [哔哩哔哩bilibili](#)

**经典例题:** [P2197 【模板】nim 游戏 - 洛谷 | 计算机科学教育新生态 \(luogu.com.cn\)](#)

**AC代码:**

```

1 using namespace std;
2 int T;
3
4 int main() {
5     cin >> T;
6     while (T--) {
7         int n;
8         scanf("%d", &n);
9         int ans = 0;
10        for (int i = 0; i < n; i++) {
11            int k;
12            scanf("%d", &k);
13            ans ^= k;
14        }
15        if (ans)
16            puts("Yes");
17        else
18            puts("No");
19    }
20    return 0;
}
```

**结论：**

若初态为**必胜态**( $a_1 \oplus a_2 \oplus \dots \oplus a_n \neq 0$ ).则先手必胜

若初态为**必败态**( $a_1 \oplus a_2 \oplus \dots \oplus a_n = 0$ ).则先手必败

**视频讲解：**[581 尼姆（Nim）游戏【博弈论】](#)哔哩哔哩bilibili

**台阶型NIM游戏**

**经典例题：**[892. 台阶-Nim游戏 - AcWing题库](#)

**AC代码：**

```

1  using namespace std;
2
3  const int N = 100010;
4
5  int main()
6  {
7      int n;
8      scanf("%d", &n);
9      int res = 0;
10     for (int i = 1; i <= n; i ++ )
11     {
12         int x;
13         scanf("%d", &x);
14         if (i & 1) res ^= x;
15     }
16     if (res) puts("Yes");
17     else puts("No");
18
19     return 0;
20 }
```

**结论：**若奇数台阶上的 $a_1 \oplus a_3 \oplus a_5 \oplus \dots \neq 0$ , 则先手必胜, 反之先手必败。

**视频讲解：**[582 台阶型 Nim游戏【博弈论】](#)哔哩哔哩bilibili

**集合型NIM游戏**

**经典例题：**[893. 集合-Nim游戏 - AcWing题库](#)

**AC代码：**

```

1  using namespace std;
2
3  const int N=110,M=10010;
4  int n,m;
5  int f[M],s[N];//s存储的是可供选择的集合,f存储的是所有可能出现过的情况的sg值
6
```

```

7 int sg(int x)
8 {
9     if(f[x]!=-1) return f[x];
10    //因为取石子数目的集合是已经确定了的,所以每个数的sg值也都是确定的,如果存储过了,
11    //直接返回即可
12    unordered_set<int> S;
13    //set代表的是有序集合(注:因为在函数内部定义,所以下一次递归中的S不与本次相同)
14    for(int i=0;i<m;i++)
15    {
16        int sum=s[i];
17        if(x>=sum) S.insert(sg(x-sum));
18        //先延伸到终点的sg值后,再从后往前排查出所有数的sg值
19    }
20    for(int i=0;;i++)
21    //循环完之后可以进行选出最小的没有出现的自然数的操作
22    if(!S.count(i))
23        return f[x]=i;
24    }
25 int main()
26 {
27     cin>>m;
28     for(int i=0;i<m;i++)
29         cin>>s[i];
30
31     cin>>n;
32     memset(f, -1, sizeof(f)); //初始化f均为-1,方便在sg函数中查看x是否被记录过
33
34     int res=0;
35     for(int i=0;i<n;i++)
36     {
37         int x;
38         cin>>x;
39         res^=sg(x);
40         //观察异或值的变化,基本原理与Nim游戏相同
41     }
42
43     if(res) printf("Yes");
44     else printf("No");
45
46     return 0;
47 }

```

思路：转换成有向图游戏

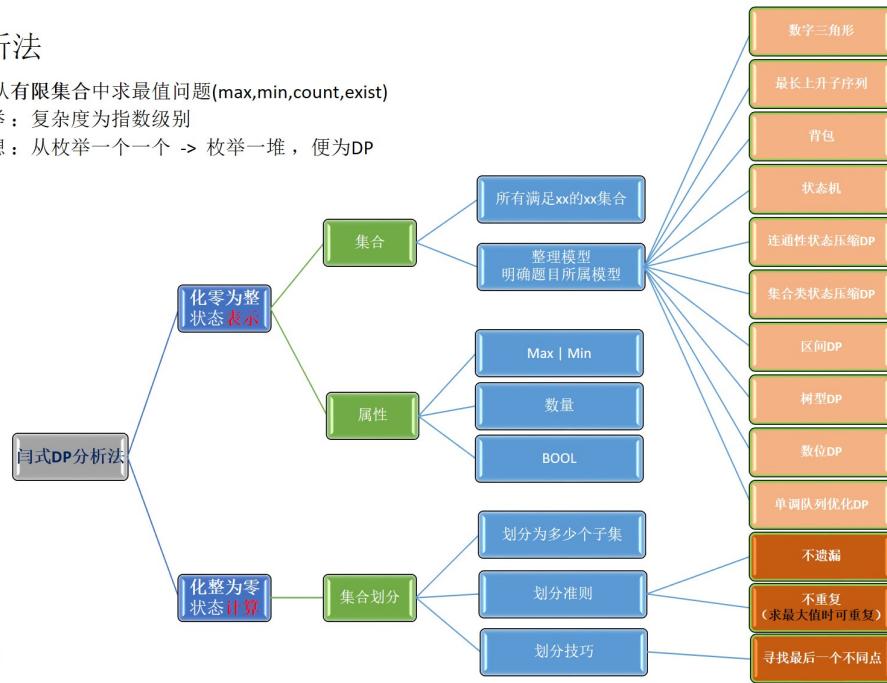
视频讲解：[583 有向图游戏 SG函数【博弈论】哔哩哔哩bilibili](#)

## 五、动态规划

## 闫式DP分析法

- 动态规划：从有限集合中求最值问题(max,min,count,exist)
  - 暴力枚举：复杂度为指数级别
  - 优化思想：从枚举一个一个 -> 枚举一堆，便为DP

Edit : jasonlin



## 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)

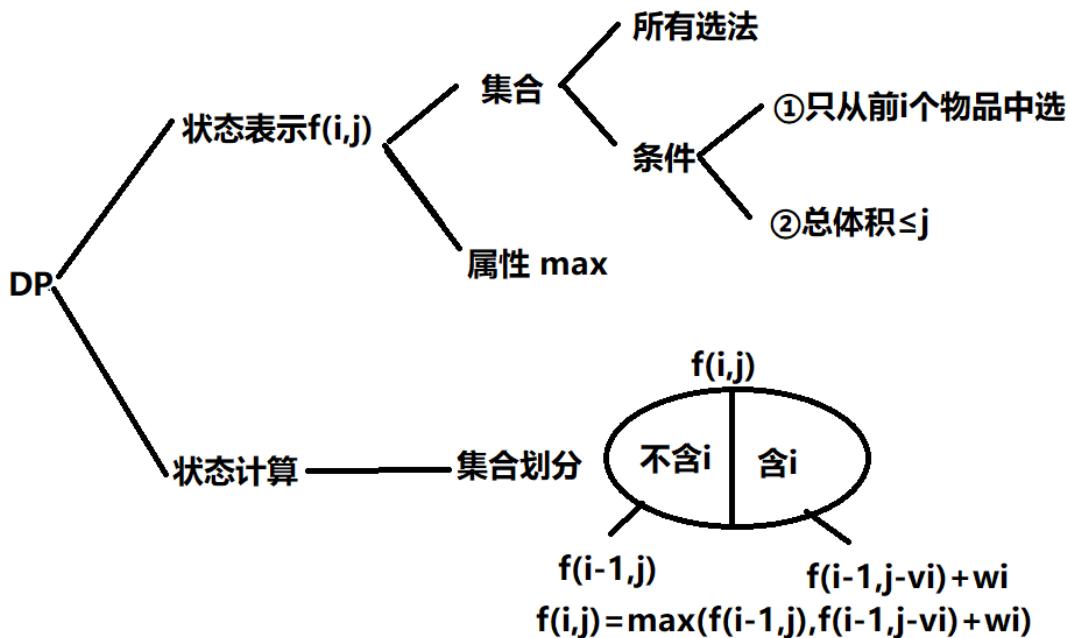
相关链接：<https://zhuanlan.zhihu.com/p/166439661>

## 01背包问题

01背包每件物品只能装一次

视频讲解：[408 背包DP【模板】01背包](#)哔哩哔哩bilibili

## 01背包DP分析



	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							

```

1 using namespace std;
2 const int N=1010;
3 int n,m;
4 int v[N],w[N];//v代表体积， w代表价值
5 int f[N][N];
6
7 int main(){
8     cin>>n>>m;
9     for(int i=1;i<=n;i++)cin>>v[i]>>w[i];
10    for(int i=1;i<=n;//i代表这n件物品
11    {
12        for(int j=1;j<=m;j++){//j代表背包容量
13            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
14                f[i][j]=f[i-1][j];
15            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
16        }
17    }
18
19    cout<<f[n][m]<<endl;//输出最后的一个一定是最大的
20    return 0;
21 }
```

01背包，使用滚动数组，倒序遍历

```

1  using namespace std;
2  const int N=1010;
3  int n,m;
4  int v[N],w[N];//v代表体积， w代表价值
5  int dp[N];
6
7  int main(){
8      cin>>n>>m;
9      for(int i=1;i<=n;i++)//i代表这n件物品
10     {
11         cin>>v[i]>>w[i];//在线算法
12         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
13             dp[j]=max(dp[j],dp[j-v[i]]+w[i]);//滚动数组
14         }
15     }
16     cout<<dp[m]<<endl;//输出最后的一个一定是最大的
17     return 0;
18 }
```

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

## 完全背包问题

完全背包每件物品可以装无限次

[视频讲解: 409 背包DP 完全背包【动态规划】哔哩哔哩bilibili](#)

```

1  using namespace std;
2  int v[N],w[N];
3  int dp[N];
4  int main(){
5      int n,m;
6      cin>>n>>m;
7      for(int i=1;i<=n;i++){//遍历物品
8          cin>>v[i]>>w[i];//在线算法
9          for(int j=v[i];j<=m;j++){//正序遍历背包容量
10              dp[j]=max(dp[j],dp[j-v[i]]+w[i]);//滚动数组
11          }
12      }
13      cout<<dp[m]<<endl;//输出答案
14      return 0;
15 }
```

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

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

## 多重背包问题1

多重背包每件物品只能装有限次 (多次)

```

1  using namespace std;
2  int n,m;
```

```

3 int v[N],w[N],s[N];
4 int dp[N][N];
5
6 int main(){
7     cin>>n>>m;
8     for(int i=1;i<=n;i++)cin>>v[i]>>w[i]>>s[i];
9     for(int i=1;i<=n;i++)//物品
10        for(int j=0;j<=m;j++)//背包容量
11            for(int k=0;k<=s[i]&&k*v[i]<=j;k++)
12                dp[i][j]=max(dp[i][j],dp[i-1][j-v[i]*k]+w[i]*k);
13     cout<<dp[n][m]<<endl;
14     return 0;
15 }

```

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

## 多重背包问题2(二进制优化)

**思路:** 转换成2进制, 再用01背包求解

**视频讲解:** 410 背包DP 多重背包 二进制优化【动态规划】哔哩哔哩bilibili

```

1 using namespace std;
2
3 const int N = 12010, M = 2010;
4
5 int n, m;
6 int v[N], w[N];
7 int f[M];
8
9 int main()
10 {
11     cin >> n >> m;
12
13     int cnt = 0;
14     for (int i = 1; i <= n; i++)
15     {
16         int a, b, s;
17         cin >> a >> b >> s;
18         int k = 1;
19         while (k <= s)
20         {
21             cnt++;
22             v[cnt] = a * k;
23             w[cnt] = b * k;
24             s -= k;
25             k *= 2;
26         }
27         if (s > 0)
28         {
29             cnt++;
30             v[cnt] = a * s;
31             w[cnt] = b * s;
32         }
33     }//二进制优化操作
34
35     n = cnt;

```

```

36
37     for (int i = 1; i <= n; i ++ )
38         for (int j = m; j >= v[i]; j -- )
39             f[j] = max(f[j], f[j - v[i]] + w[i]);
40
41     cout << f[m] << endl;
42
43     return 0;
44 }
```

## 分组背包问题

分组背包每组只能选择一件物品装入

视频讲解: 416 背包DP 分组背包【动态规划】哔哩哔哩bilibili

```

1  using namespace std;
2  const int N=110;
3  int f[N];
4  int v[N][N],w[N][N],s[N];
5  int n,m,k;
6
7  int main(){
8      cin>>n>>m;
9      for(int i=0;i<n;i++){
10          cin>>s[i];
11          for(int j=0;j<s[i];j++){
12              cin>>v[i][j]>>w[i][j];
13          }
14      }
15
16      for(int i=0;i<n;i++){
17          for(int j=m;j>=0;j--){
18              for(int k=0;k<s[i];k++){ //for(int k=s[i];k>=1;k--)也可以
19                  if(j>=v[i][k])
20                      f[j]=max(f[j],f[j-v[i][k]]+w[i][k]);
21              }
22          }
23      }
24      cout<<f[m]<<endl;
25 }
```

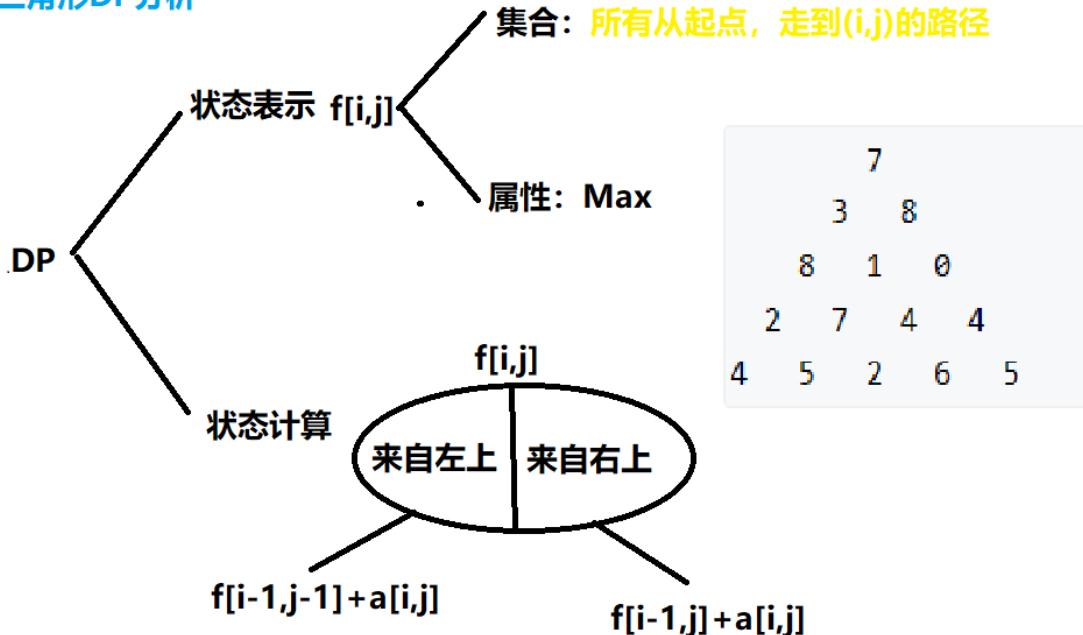
状态转移方程:  $f[j] = \max(f[j], f[j-v[i][k]] + w[i][k]);$

## 线性DP

### 数字三角形

视频讲解: 402 线性DP 数字三角形【动态规划】哔哩哔哩bilibili

## 数字三角形DP分析



```

1  using namespace std;
2  const int N=510,INF=1e9;
3  int n;
4  int a[N][N];
5  int f[N][N];
6
7  int main(){
8      scanf("%d",&n);
9      for(int i=1;i<=n;i++){
10          for(int j=1;j<=i;j++){
11              scanf("%d",&a[i][j]);
12          }
13      }
14      for(int i=0;i<=n;i++){
15          for(int j=0;j<=i+1;j++){
16              f[i][j]=-INF;
17          }
18      }
19      f[1][1]=a[1][1];
20      for(int i=2;i<=n;i++){
21          for(int j=1;j<=i;j++)
22              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
23      int res=-INF;
24      for(int i=1;i<=n;i++)res=max(res,f[n][i]);
25      printf("%d",res);
26  }
27 }
```

状态转移方程:  $f[i][j]=\max(f[i-1][j-1]+a[i][j], f[i-1][j]+a[i][j])$ ;

## 最长上升子序列I

视频讲解: 403 线性DP 最长上升子序列【动态规划】哔哩哔哩bilibili

```

1  using namespace std;
2  const int N = 1010;
3
4  int n;
5  int a[N], f[N];
6
7  int main()
8  {
9      scanf("%d", &n);
10     for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
11     for (int i = 1; i <= n; i++){
12         f[i] = 1; // 只有 a[i] 一个数
13         for (int j = 1; j <= i; j++)
14             if (a[j] < a[i])
15                 f[i] = max(f[i], f[j] + 1);
16     }
17     int res = 0;
18     for (int i = 1; i <= n; i++) res = max(res, f[i]);
19     printf("%d\n", res);
20     return 0;
21 }
```

状态转移方程:  $\text{if}(a[j] < a[i]) f[i] = \max(f[i], f[j] + 1);$

## 最长上升子序列2(二分优化)

视频讲解: 404 线性DP 最长上升子序列 二分优化 [哔哩哔哩bilibili](#)

```

1  using namespace std;
2
3  const int N = 100010;
4
5  int n;
6  int a[N];
7  int q[N];
8
9  int main()
10 {
11     scanf("%d", &n);
12     for (int i = 0; i < n; i++) scanf("%d", &a[i]);
13
14     int len = 0;
15     for (int i = 0; i < n; i++)
16     {
17         int l = 0, r = len;
18         while (l < r)
19         {
20             int mid = l + r + 1 >> 1;
21             if (q[mid] < a[i]) l = mid;
22             else r = mid - 1;
23         }
24         len = max(len, r + 1);
25         q[r + 1] = a[i]; // 替换或添加
26     }
27
28     printf("%d\n", len);
29 }
```

```
30     return 0;
31 }
```

## 最长公共子序列

视频讲解：[405 线性DP 最长公共子序列【动态规划】哔哩哔哩bilibili](#)

```
1 using namespace std;
2 const int N=1010;
3 int n,m;
4 char a[N],b[N];
5 int f[N][N];
6
7 int main()
8 {
9     cin>>n>>m>>a+1>>b+1;
10    for (int i = 1; i <= n; i ++ ){
11        for (int j = 1; j <= m; j ++ ){
12            f[i][j]=max(f[i-1][j],f[i][j-1]);
13            if(a[i]==b[j])f[i][j]=max(f[i][j],f[i-1][j-1]+1);
14        }
15    }
16    cout<<f[n][m]<<endl;
17    return 0;
18 }
```

状态转移方程：

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

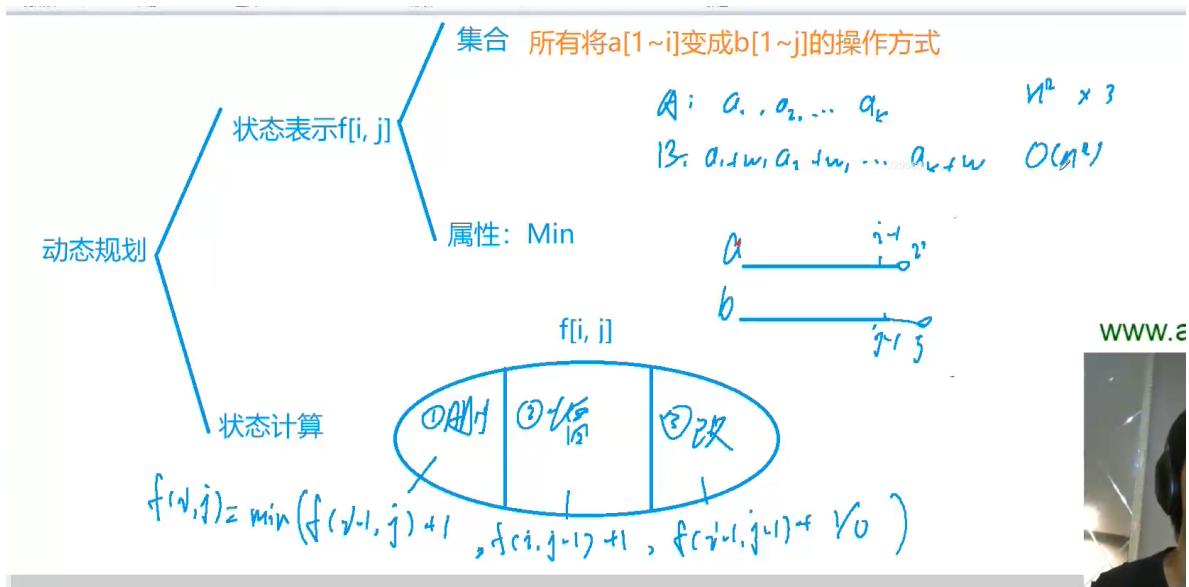
## 最短编辑距离

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。

视频讲解：[407 线性DP 编辑距离【动态规划】哔哩哔哩bilibili](#)



```

1 using namespace std;
2 const int N = 1010;
3 int n, m;
4 char a[N], b[N];
5 int f[N][N];
6
7 int main()
8 {
9     scanf("%d%s", &n, a+1);
10    scanf("%d%s", &m, b+1);
11
12    for (int i = 0; i <= m; i++) f[0][i] = i;
13    for (int i = 0; i <= n; i++) f[i][0] = i; // 初始化字符串的编辑操作
14    for (int i = 1; i <= n; i++) {
15        for (int j = 1; j <= m; j++) {
16            f[i][j] = min(f[i-1][j] + 1, f[i][j-1] + 1);
17            if (a[i] == b[j]) f[i][j] = min(f[i][j], f[i-1][j-1]);
18            else f[i][j] = min(f[i][j], f[i-1][j-1] + 1); // 状态转移方程
19        }
20    }
21    printf("%d\n", f[n][m]);
22    return 0;
23 }
```

状态转移方程:

```

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

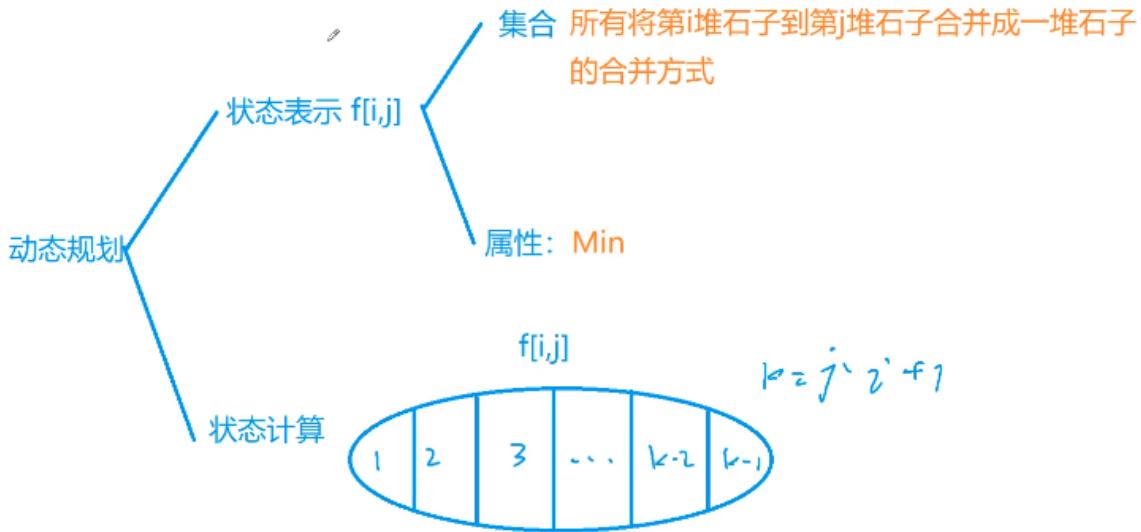
```

## 区间DP

### 石子合并

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。



```

1 using namespace std;
2 const int N = 310;
3
4 int n;
5 int s[N];
6 int f[N][N]; //状态表示: 集合f[1][r]为[1,r]区间; 属性: 所堆成的最小值
7 int main()
8 {
9     scanf("%d", &n);
10    for (int i = 1; i <= n; i++) scanf("%d", &s[i]);
11    for (int i = 1; i <= n; i++) s[i] += s[i-1]; //前缀和用来求一段区间的和
12
13    for (int len = 2; len <= n; len++) //区间长度为len//枚举长度
14        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
15            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
16            f[l][r]=1e9; //初始化最大值
17            for (int k = l; k < r; k++) //枚举分界点//不取r
18                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];
19        }
20    printf("%d\n", f[1][n]); //输出区间[1,n]的最小值
21    return 0;
22 }
```

状态转移方程找到最小值状态转移方程为 $f[l][r]=\min(f[l][r], f[l][k]+f[k+1][r]+s[r]-s[l-1])$

## 计数类DP

### 整数划分

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

```
1 //完全背包的写法
2 using namespace std;
3 const int M=1e9+7;
4 int f[1010],n;
5
6 int main()
7 {
8     cin>>n;
9     f[0]=1;
10    for (int i = 1; i <= n; i++)
11        for (int j = i; j <= n; j++)
12        {
13            f[j]=(f[j-i]+f[j])%M;
14        }
15    cout<<f[n]<<endl;
16 }
```

状态转移方程:  $f[j] = (f[j-i] + f[j]) \bmod M$

## 数位统计DP

### 计数问题

题目链接: [338. 计数问题 - AcWing题库](#)

1~n, x = 1

n = abcdefg

分别求出1在每一位上出现的次数

求1在第4位上出现的次数

1 <= xxx1yyy <= abcdefg

(1) xxx = 000~abc - 1, yyy = 000~999, abc \* 1000

(2) xxx = abc

(2.1) d < 1, abc1yyy > abc0efg , 0

(2.2) d = 1, yyy = 000~efg, efg + 1

(2.3) d > 1, yyy = 000~999, 1000

求1~n中x的个数  
设x=1, n=abcdefg  
分别求出1在每一位上出现的次数

例如求1在第4位上出现的次数

1<=xxx1yyy<=abcdefg

(1)当xxx $\in$ [0,abc-1]时, 那么yyy就能取[000,999], 即在4位出现的次数为abc\*1000  
(如果x=0的话, 因为不能出现前导零所以这里要特判一下, xxx $\in$ [1,abc-1]要从最小为1开始计数)

(2)当xxx=abc时, 有下三种情况

(2.1) d<1, 即d=0, 因为abc1yyy>abcdefg, 所以在第4位出现的次数为0

(2.2) d=1, 那么yyy $\in$ [000,efg], 所以在第4位出现的次数为efg+1

(2.3) d>1, 那么yyy $\in$ [000,999], 所以在第4位出现的次数位1000

综上所述: 把情况(1)和情况(2)累加起来就是求1~abcdefg中的第4位1出现的次数

```
1 using namespace std;
2
3 //因为我们举的分类中, 有需要求一串数字中某个区间的数字, 例如abcdefg有一个分类需要求出
4 //efg+1
5 int get(vector<int> num,int l,int r){
6     int res=0;
7     for(int i=l;i>r;i--)res=res*10+num[i];//这里从小到大枚举的是因为下面count的时候
8     //读入数据是从最低为读到最高位, 那么此时在num里, 最高位存的就是数字的最低位, 那么假如我们要
9     //求efg, 那就是从2算到0
10    return res;
11 }
12
13 int power10(int i)//这里有power10是因为有一个分类需要求得十次方得值
14 {
15     int res=1;
16     while(i--)res*=10;
17     return res;
18 }
19
20 int count(int n,int x){
21     if(!n)return 0;//n=0则返回0
22     vector<int> num;//num用来存储数中的每一位数字
23     while(n){
24         num.push_back(n%10);
25         n/=10;
26     }
27     n=num.size();//得出它的长度
28     int res=0;
29     for (int i = n-1-!x; i >=0; i -- )
30         //这里需要注意, 我们的长度需要减一, 是因为num是从0开始存储, 而长度是元素的个数, 因此
31         //需要减1才能读到正确的数值, 而!x出现的原因是因为我们不能让前导零出现, 如果此时需要我们列举
32         //的是0得出现的次数, 那么我们自然不能让他们出现第一位, 而是从第二位开始枚举
33     {
34         if(i<n-1)//其实这里可以不用if判断, 因为for循环里面实际上就已经达成了if得判断,
35         //但为了方便理解还是加上if来理解, 这里i要小于n-1的原因是因为我们不能越界只有7位数就最高从七
36         //位数开始读起
37     {
38         res+=get(num,n-1,i+1)*power10(i);//这里就是第一个分类, 000~abc-1, 那么此
39         //时情况个数就会是abc*103, 这里的3取决于后面的efg的长度, 假如他是efgh, 那么就是4
40         ////这里的n-1, i+1, 自己将数组列出然后根据分类标准就可以得出为什么1是n-1, r=i+1
41     }
42 }
```

```

33         if(!x)res-=power10(i); //假如此时我们要列举的是0出现的次数，因为不能出现前
导零，这样是不合法也不符合我们的分类情况，例如abcdefg我们列举d，那么他就得从001~abc-1,
这样就不会直接到efg，而是会到0efg，因为前面不是前导零，自然就可以列举这个时候0出现的次
数，所以要减掉1个power10
34     }
35     if(num[i]==x)res+=get(num,i-1,0)+1;
36     else if(num[i]>x)res+=power10(i);
37   }
38   return res;//返回res，即出现次数
39 }
40
41 int main()
42 {
43   int a,b;
44   while(cin>>a>>b,a||b){
45     if(a>b)swap(a,b);//a大于b则交换a, b使得变成合法参数
46     for(int i=0;i<10;i++)
47       cout<<count(b,i)-count(a-1,i)<<' ';//使用前缀和思想解决[a,b]的i出现的次
数
48     cout<<endl;
49   }
50   return 0;
51 }
```

## 状态压缩DP

### 蒙德里安的梦想

**题目链接：**[U204941 蒙德里安的梦想 - 洛谷 | 计算机科学教育新生态 \(luogu.com.cn\)](https://www.luogu.com.cn/problem/U204941)

**视频讲解：**[431 状态压缩DP 蒙德里安的梦想【动态规划】哔哩哔哩bilibili](#)

```

1 using namespace std;
2
3 const int N = 12,M=1<<N;
4
5 int n,m;
6 long long f[N][M];
7 bool st[M];
8
9 int main()
10{
11    int n,m;
12    while(cin>>n>>m,n||m){
13        memset(f, 0, sizeof f);
14        //预处理：判断合并列的状态i是否合法
15        //如果合并列的某行是1表示横放，是0表示竖放
16        //如果合并列不存在连续的奇数个0，即为合法状态
17        for (int i = 0; i < 1<<n; i ++ ){
18            st[i]=true;
19            int cnt=0;//记录合并列中连续0的个数
20            for (int j = 0; j < n; j ++ ){
21                if(i>>j&1){//如果是1
22                    if(cnt&1){//如果连续0的个数是奇数
23                        st[i]=false;//记录i不合法
24                        break;
25                    }
26                }
27            }
28        }
29    }
30}
```

```

26         }else cnt++;//如果是0，记录0的个数
27     }
28     if(cnt&1)st[i]=false;//处理高位0的个数
29 }
30 //状态计算
31 f[0][0]=1;//第0列不横放是一种合法的方案
32 for (int i = 1; i <= m; i ++ )//阶段：枚举列
33     for (int j = 0; j < 1<<n; j ++ )//状态：枚举i列的状态
34         for (int k = 0; k < 1<<n; k ++ )//状态：枚举i-1列的状态
35             //两列状态兼容：不出现重叠的1，不出现连续奇数个0
36             if((j&k)==0&&st[j|k])
37                 f[i][j]+=f[i-1][k];
38 cout<<f[m][0]<<endl;//第m列不横放，既答案
39 }
40 return 0;
41 }

```

状态转移方程：

```

1 | if((j&k)==0&&st[j|k])
2 | f[i][j]+=f[i-1][k];

```

## 最短Hamilton路径

题目链接：[U122241 最短Hamilton路径 - 洛谷 | 计算机科学教育新生态 \(luogu.com.cn\)](https://www.luogu.com.cn/problem/U122241)

```

1 using namespace std;
2
3 const int N = 20,M = 1 << N;
4
5 int n;
6 int w[N][N];
7 int f[M][N];//第一维表示是否访问到该点的压缩状态，第二维是走到点j
8 //f[i][j]表示状态为i并且到j的最短路径
9
10 int main(){
11     cin>>n;
12     for (int i = 0; i < n; i ++ )
13         for (int j = 0; j < n; j ++ )//读入i到j的距离
14             cin>>w[i][j];
15     memset(f, 0x3f, sizeof f);
16     f[1][0]=0;
17     for (int i = 0; i < 1 << n; i ++ )//枚举压缩的状态
18         for (int j = 0; j < n; j ++ )//枚举到0~j的点
19             if(i >> j & 1)//该状态存在j点
20                 for (int k = 0; k < n; k ++ )//枚举从j倒数第二个点k
21                     if(i >> k & 1)//倒数点k存在
22                         f[i][j]=min(f[i][j],f[i-(1<<j)][k]+w[k][j]);//状态转移方
程，在f[i][j]和状态去掉j的点f[i-(i<<j)][k]+w[k][j]取最小值
23             cout<<f[(1<<n)-1][n-1]<<endl;//输出状态全满也就是所有点都经过且到最后一个点的最短
距离
24     return 0;
25 }

```

状态转移方程：

```
1 | f[i][j]=min(f[i][j],f[i-(1<<j)][k]+w[k][j]);
```

## 树形DP

### 没有上司的舞会

题目：P1352 没有上司的舞会 - 洛谷 | 计算机科学教育新生态 (luogu.com.cn)

视频讲解：417 树形DP 没有上司的舞会【动态规划】哔哩哔哩bilibili

```
1 using namespace std;
2
3 const int N = 6010;
4
5 int n;
6 int w[N];//每个节点的高兴度
7 int h[N], e[N], ne[N], idx;//邻接表存储树
8
9 bool st[N];//判断是否有父节点
10 int f[N][2];
11
12 void add(int a, int b) // 添加一条边a->b
13 {
14     e[idx] = b, ne[idx] = h[a], h[a] = idx++;
15 }
16 void dfs(int u){
17     f[u][0]=0;
18     f[u][1]=w[u];//初始化f[u][1], 当第二维是0则不选该点即高兴度为0, 同理f[u][1]=w[u];
19     for (int i = h[u]; i != -1; i = ne[i]){//遍历u的子节点进行深度优先遍历
20         int j=e[i];
21         dfs(j);
22         //状态转移方程
23         f[u][0]+=max(f[j][0],f[j][1]);//f[u][0]表示不选择父节点u, 所以在f[j][0]和
24         //f[j][1]取最大值
25         f[u][1]+=f[j][0];//f[u][1]表示选择根节点u, 所以累加不选择子节点的f[j][0]
26     }
27 }
28
29 int main()
30 {
31     cin>>n;
32     for (int i = 1; i <= n; i++) cin>>w[i];
33     memset(h, -1, sizeof h);
34     for (int i = 0; i < n-1; i++){
35         int a,b;
36         cin>>a>>b;
37         add(b,a);
38         st[a]=true;//存储是否存在父节点
39     }
40     int root=1;
41     while(st[root])root++;//判断是否是根节点
42     dfs(root);//dfs对f[i][j]进行状态转移计算
43     cout<<max(f[root][0],f[root][1])<<endl;//取选与不选根节点的最大值
44 }
```

状态转移方程：

```
1 | f[u][0] += max(f[j][0], f[j][1]);  
2 | f[u][1] += f[j][0];
```

## 记忆化搜索

### 滑雪

题目链接：[P1434 SHOI2002] 滑雪 - 洛谷 | 计算机科学教育新生态 (luogu.com.cn)

```
1 | using namespace std;  
2 | const int N = 310;  
3 |  
4 | int n,m;  
5 | int h[N][N];  
6 | int f[N][N];  
7 |  
8 | int dx[4]={-1,0,1,0},dy[4]={0,1,0,-1};  
9 |  
10 | int dp(int x,int y){  
11 |     int &v=f[x][y];  
12 |     if(v!=-1) return v;//记忆化搜索核心  
13 |     v=1;  
14 |     for (int i = 0; i < 4; i ++ ){  
15 |         int a=x+dx[i],b=y+dy[i];  
16 |         if(a>=1&&a<=n&&b>=1&&b<=m&&h[a][b]<h[x][y])//判断是否越界且上一个经过的点的高度是否大于当前高度  
17 |             v=max(v,dp(a,b)+1);  
18 |     }  
19 |     return v;  
20 | }  
21 |  
22 | int main()  
23 | {  
24 |     scanf("%d%d", &n, &m);  
25 |     for (int i = 1; i <= n; i ++ )  
26 |         for (int j = 1; j <= m; j ++ )  
27 |             scanf("%d", &h[i][j]);  
28 |     memset(f, -1, sizeof f);  
29 |     int res=0;  
30 |     for (int i = 1; i <= n; i ++ )  
31 |         for (int j = 1; j <= m; j ++ )  
32 |             res=max(res,dp(i,j));  
33 |     printf("%d\n",res);  
34 |     return 0;  
35 | }
```

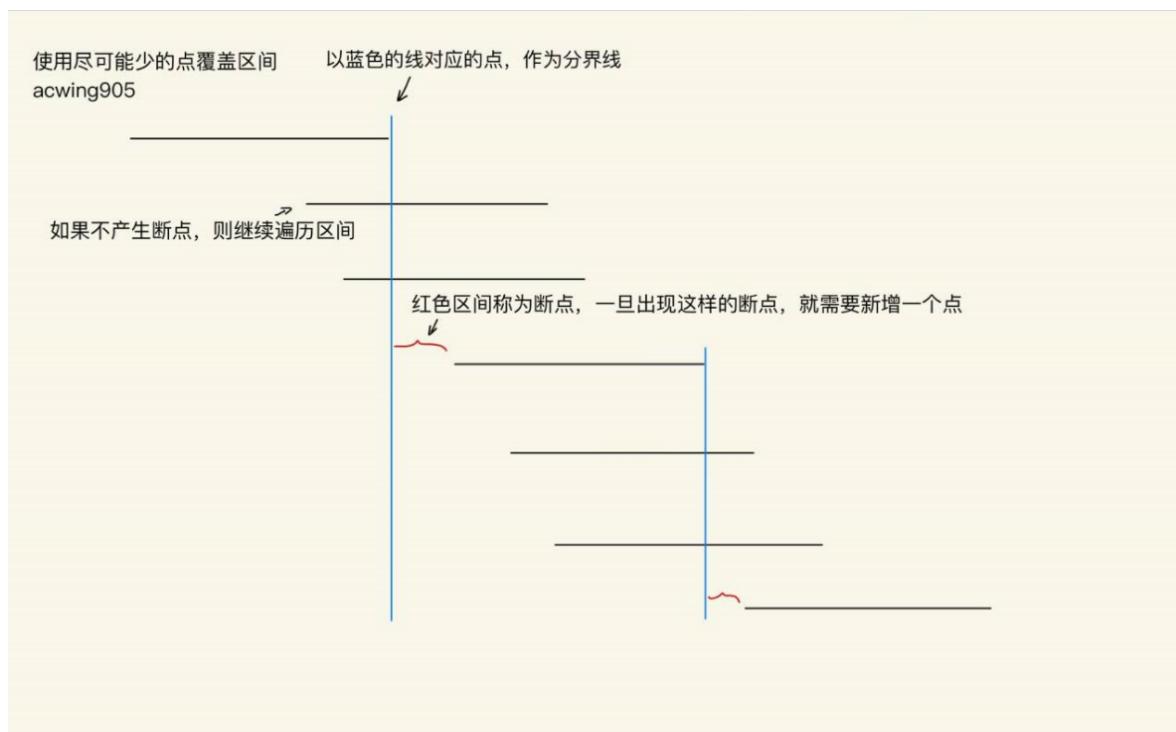
状态转移方程：  $v=\max(v, dp(a,b)+1);$

## 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》

## 区间问题

### 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。

```
1 using namespace std;
2
3 const int N = 100010;
4
5 int n;
6 struct Range{
7     int l,r;
8     bool operator <(const Range& W) const{
9         return r<W.r;
10    } //重载小于号
11 }range[N];
12
13 int main()
14 {
15     scanf("%d", &n);
16     for (int i = 0; i < n; i ++ ){
17         int l,r;
18         scanf("%d%d", &l, &r);
19         range[i]={l,r}; //读入l,r
20     }
21     sort(range,range+n); //按右端点进行排序
22     int res=0,ed=-2e9; //ed代表上一个点的右端点
23     for (int i = 0; i < n; i ++ ){
```

```

24     if(range[i].l>ed){
25         res++; //点的数量加一
26         ed=range[i].r;
27     }
28 }
29 printf("%d\n",res);
30 return 0;
31 }
```

## 最大不相交区间数量

给定  $N$  个闭区间  $[a_i, b_i]$ , 请你在数轴上选择若干区间, 使得选中的区间之间互不相交 (包括端点)。

输出可选取区间的最大数量。

**结论:** 最大不相交区间数量=最少覆盖区间点数

为什么最大不相交区间数=最少覆盖区间点数呢?

因为如果几个区间能被同一个点覆盖

说明他们相交了, 所以有几个点就是有几个不相交区间

```

1 using namespace std;
2
3 const int N = 100010;
4
5 int n;
6 struct Range{
7     int l,r;
8     bool operator <(const Range& W) const{
9         return r<W.r;
10    }
11 }range[N];
12
13 int main()
14 {
15     scanf("%d", &n);
16     for (int i = 0; i < n; i ++ ){
17         int l,r;
18         scanf("%d%d", &l, &r);
19         range[i]={l,r};
20     }
21     sort(range,range+n);
22     int res=0,ed=-2e9;
23     for (int i = 0; i < n; i ++ ){
24         if(range[i].l>ed){
25             res++;
26             ed=range[i].r;
27         }
28     }
29     printf("%d\n",res);
30     return 0;
31 }
```

## 区间分组

1. 将所有区间按左端点从小到大排序

2. 从前往后处理每个区间

判断能否将其放到某个现有的组中  $L[i] > Max\_r$

1. 如果不存在这样的组，则开新组，然后再将其放进去；

2. 如果存在这样的组，将其放进去，并更新当前组的Max\_r

```
1 using namespace std;
2
3 const int N = 1e5+10;
4
5 int n;
6 struct Range{
7     int l,r;
8     bool operator<(const Range &W) const{
9         return l<W.l;
10    } //按左端点排序
11 }range[N];
12
13 int main()
14 {
15     scanf("%d", &n);
16     for (int i = 0; i < n; i ++ ){
17         int l,r;
18         scanf("%d%d", &l, &r);
19         range[i]={l,r};
20     }
21     sort(range,range+n); //sort排序
22     priority_queue<int,vector<int>,greater<int>> heap; //小根堆维护所有组的右端点最小值
23     for (int i = 0; i < n; i ++ ){ //从左往右枚举
24         auto r=range[i]; //选择当前区间
25         if(heap.empty()||heap.top()>=r.l)heap.push(r.r);
26         else{
27             heap.pop();
28             heap.push(r.r);
29         }
30     }
31     printf("%d\n",heap.size());
32     return 0;
33 }
```

## 排序不等式

### 排队打水

有  $n$  个人排队到  $s$  个水龙头处打水，第  $i$  个人装满水桶所需的时间是  $t_i$ ，请问如何安排他们的打水顺序才能使所有人的等待时间之和最小？

$t[i]$  从小到大排序