算法模板总结

- **排序**
  - **快速排序** $O(nlog_2(n))$

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

  - **快速选择** $O(n)$

```
int quick_choose(int q[], int l, int r, int k){//k为寻找第k小的数
                if(l >= r) return q[l];
                int x = q[l + r >> 1], i = l - 1, j = r + 1;
                while(i < j){
                        while(q[++ i] < x);
                        while(q[-- j] > x);
                        if(i < j) swap(q[i], q[j]);
                }
                int s = j - l + 1;
                //选择一边来递归
                if(s >= k) quick_choose(q, l, j, k);
                else quick_choose(q, j + 1, r, k - s);
        }
        ```
```

  - **归并排序** $O(nlog_2(n))$

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

```
        int k = 1, i = l, j = mid + 1;
        while(i <= mid && j <= r)
                if(q[i] <= q[j]) tmp[k ++] = q[i ++];
                else tmp[k ++] = q[j ++];
        while(i <= mid) tmp[k ++] = q[i ++];
        while(j <= r) tmp[k ++] = q[j ++];
        for(int i = l, j = 1; i <= r; ++ i, ++ j)
                 q[i] = tmp[j];
}
```

- **二分**
  - **整数二分** $O(log_2(n))$

```
//找最小
int binary(int l, int r){
        while(l < r){
                int mid = r + l >> 1;
                if(check(mid)) r = mid;
                else l = mid + 1;
        }
        return l;
}
//找最大
int binary(int l , int r){
        while(l < r){
                int mid = r + l + 1 >> 1;
                if(check(mid)) l = mid;
                else r = mid - 1;
        }
        return l;
}
```

  - **浮点二分** $O(log_2(n))$

```
//浮点数二分模板一：
void binary(int x){
        double l = 0, r = 1000;
        while(r - l > 1e-9){
                double mid = (r + l) / 2;
                if(check(mid)) l = mid;
                else r = mid;
        }
        printf("%.6f",l);
        return ;
```

```
}

//浮点数二分模板二：
void binary(int x){
        double l = 0, r = 1000;
        for(int i = 1; i < 100; i ++){
                double mid = (r + l) / 2;
                if(check) l = mid;
                else r = mid;
        }
        printf("%.6f",l);
        return ;
}
```

- **高精度**
  - **高精度加法** $O(n)$

```
//存储方式
string a, b;//a = "123456"
vector<int> A, B;//A = [6,5,4,3,2,1]
for(int i = a.size() - 1; i >= 0; ++ i) A.push_back(a[i] - '0');
for(int i = b.size() - 1; i >= 0; ++ i) B.push_back(b[i] - '0');

//加法
vector<int> add(vector<int> &A, vector<int> &B){
        vector<int> C;
        int t = 0;
        for(int i = 0; i < A.size() || i < B.size(); ++ i){
                if(i < A.size()) t += A[i]; //t = A[i] + B[i];
                if(i < B.size()) t += B[i];
                C.push_back(t % 10);
                t /= 10;
        }
        if(t) C.push_back(1);
        return C;
}
//逆序输出
for(int i = C.size() - 1; i >= 0; -- i) cout << C[i];
```

  - **高精度减法** $O(n)$

```
//判断是否有A > B
//注意不能直接用字符串比较a,b的大小
bool cmp(vector<int> &A, vector<int> &B){
```

```
        if(A.size() != B.size()) return A.size() > B.size();
        for(int i = A.size() - 1; i >= 0; ++ i)
                if(A[i] != B[i]) return A[i] > B[i];
}

//减法，要保证A > B
vector<int> sub(vector<int> &A, vector<int> &B){
        vector<int> C;
        int t = 0;
        for(int i = 0; i < A.size(); ++ i){
                t = A[i] - t;
                if(i < B.size()) t -= B[i]; //t = A[i] - B[i] - t
                C.push_back((t + 10) % 10);
                if(t < 0) t = 1;
                else t = 0;
        }
        //删除前导零
        while(C.size() > 1 && C.back() == 0) C.pop_back();
        return C;
}
```

- **高精度乘法** $O(n)$

```
//乘法
vector<int> mul(vector<int> &A, int b){
        vector<int> C;
        int t = 0;
        for(int i = 0; i < A.size() || t; ++ i){
                if(i < A.size()) t += A[i] * b;
                C.push_back(t % 10);
                t /= 10;
        }
        while(C.size() > 1 && C.back() == 0) C.pop_back();//去前导零（b==0
时）
        return C;
}
```

- **高精度除法** $O(n)$

```
vector<int> div(vector<int> &A, int b, int &r){//r是余数，C存储商
        vector<int> C;
        r = 0;
        for(int i = A.size() - 1; i >= 0; -- i){//从高位开始算
                r = r * 10 + A[i];
                C.push_back(r / b);
```

```
                t %= b;
        }
        reverse(C.begin(), C.end());
        while(C.size() > 1 && C.back() == 0) C.pop_back();
        return C;
}
```

- **前缀和**
  - **一维前缀和** 预处理$O(n)$,区间求和$O(1)$

```
//预处理
for(int i = 1; i <= n; ++ i) s[i] = s[i - 1] + a[i];
//求和
cout << s[r] - s[l - 1];
```

  - **二维前缀和** 预处理$O(n^2)$,区间求和$O(1)$

```
//预处理
for(int i = 1; i <= n; ++ i)
        for(int j = 1; j <= m; ++ j)
                s[i][j] = s[i][j - 1] + s[i - 1][j] - s[i - 1][j - 1] +
a[i][j];
//区间求和 左上角（a, b） 右下角（c, d）
cout << s[c][d] - s[a - 1][d] - s[c][b - 1] + s[a - 1][b - 1];
```

- **差分**
  - **一维差分** 预处理$O(n)$,区间修改$O(1)$

```
void add(int l, int r, int c){ //在区间[l, r] 上加上c
        b[l] += c;
        b[r + 1] -= c;
}
//差分数组初始化，初始化就相当于在[i, i] 区间上加上a[i]
//所以初始化就与区间修改操作统一了
for(int i = 1; i <= n; ++ i) add(i, i, a[i]);

//差分数组初始化
for(int i = 1; i <= n; ++ i) b[i] = a[i] - a[i - 1];

//求原数组,直接利用b数组
for(int i = 1; i <= n; ++ i) b[i] += b[i - 1];
```

- **二维差分** 预处理$O(n^2)$,区间修改$O(1)$

```
//在以(x1, y1)为右上角(x2, y2)为左上角的矩形区域加上c
void add(int x1, int y1, int x2, int y2, int c){
        d[x1][y1] += c;
        d[x2 + 1][y1] -= c;
        d[x1][y2 + 1] -= c;
        d[x2 + 1][y2 + 1] += c;
}
//求原数组
for(int i = 1; i <= n; ++ i)
        for(int j = 1; j <= m; ++ j)
                d[i][j] += d[i - 1][j] + d[i][j - 1] + d[i - 1][j - 1];
```

- **链表**
  - **单链表**

```
int head;//头指针，指向头结点
int e[N];//e[i]表示节点i的值
int ne[N],//en[i]表示节点i的下一个节点
int idx;//存储新节点的下标

//初始化
void init(){
        head = 0; //0代表空节点
        idx = 1;//第一个插入的节点的下标为1
}

//头插法
void add_to_head(int x){
        e[idx] = x; //存值
        ne[idx] = head; //连接
        head = idx; //转移连接
        idx ++ ; //
}

//在下标为k的节点后面插入一个数
void add(int k, int x){
        e[idx] = x;
        ne[idx] = ne[k];
        ne[k] = idx;
        idx ++ ;
}
```

```
//删除下标为k的节点的后面的数
void del(int k){
        ne[k] = ne[ne[k]];
}

//头删法，删除头结点
void del_the_head(){
        head = ne[head];
}

//遍历输出所有节点
for(int i = head; i; i = ne[i]) cout << e[i];
```

- **双链表**

```
const int N = 1e5 + 10;
int l[N], r[N], e[N], idx;

//初始化
void init(){
        //0表示空头节点，1表示空尾节点
        r[0] = 1;
        l[1] = 0;
        idx = 2; //1，0已经被使用，初始化为2，表示第一个插入的节点的下标为2
}

//在下标为k的节点后面插入节点
void add(int k, int x){
        e[idx] = x; //赋值
        l[idx] = k; //连接左
        r[idx] = r[k]; //连接右
        l[r[idx]] = idx; //转移连接
        r[l[idx]] = idx; //转移连接
        idx ++ ; //更新下标
}

//删除下标为k的节点
void del(int k){
        r[l[k]] = r[k];
        l[r[k]] = l[k];
}

//遍历输出节点，注意起点与终点
for(int i = r[0]; i != 1; i = r[i]) cout << e[i];
```

```
//以下是统一插入个数与下标的代码
//初始化
void init(){
        //0表示空头结点，N - 1表示空尾节点
        r[0] = N - 1;
        l[N - 1] = 0;
        idx = 1; //0，N - 1被使用，下标从1开始，表示第一个插入的节点下标就是1
}
//add函数不变
//遍历输出，注意起点和终点
for(int i = r[0]; i != N - 1; i = r[i]) cout << e[i];
```

- **邻接表**

```
//h[i]存储以节点i为起点的单链表，单链表中的节点存的是节点i能够到达的所有节点
//以节点的编号代指结点，但是节点有两类，一类是图中的节点，一类是链表中的节点
//idx分配单链表中的节点的编号，而不是图中节点的编号，注意区分
//h[i]中的i是图中节点的编号，存的是单链表节点编号
//e[i]中的i是单链表中节点的编号，存的是图中节点的编号，e[i]表示i节点存储的图中的节点
//ne[i]中的i是单链表中节点的编号，存的是单链表点的编号，ne[i]表示i节点的下一个链表节点
//链表节点下标以1开始，0表示空节点
int h[N], e[N], ne[N], idx = 1;

void add(int a, int b){//插入节点a指向b的一条边
        e[idx] = b;//单链表节点的值就是图中节点
        ne[idx] = h[a];
        h[a] = idx;
        idx ++ ;
}
```

- **栈** $O(n)$

```
    int stk[N], tt;//声明栈和栈顶下标
    stk[ ++ tt] = x; //x入栈
    tt --; //出栈
    stk[tt]; //取栈顶元素
    if(tt > 0) not empty //判断是否为空
    else empty
```

- **单调栈** $O(n)$

```
//求最近的最小值
int stk[N], tt;
//栈存储元素的下标
for(int i = 1; i <= n; ++ i){ //遍历所有数
        while(tt && a[i] <= a[stk[tt]]) tt --; //栈非空且栈顶元素大于当前元素就出栈
        if(tt) cout << a[stk[tt]] << " "; //栈顶元素即为最小值
        else cout << -1 << " ";
        stk[ ++ tt] = i; //当前元素入栈
}

//求最近的最大值
for(int i = 1; i <= n; ++ i){ //遍历所有数
        while(tt && a[i] >= a[stk[tt]]) tt --;
        if(tt) cout << a[stk[tt]] << " ";
        else cout << -1 << " ";
        stk[ ++ tt] = i; //当前元素入栈
}
```

- **队列**
  - **普通队列** $O(n)$

```
int q[N], hh, tt = -1; //声明队列，队首下标，队尾下标
q[ ++ tt] = x; //入队只能从队尾
hh --; //出队只能从队首
q[hh];//取队首
q[tt];//取队尾
if(hh <= tt)not empty //判断是否为空
else empty
```

  - **循环队列**

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

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

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

// 队头的值
```

```
q[hh];

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

}
```

- **单调队列（单调双端队列）$O(n)$**

```
int n, k;
int q[N], hh, tt = -1;//队列存储元素的下标，便于检查其是否超出窗口范围
int a[N];

//求滑动窗口最小值
for(int i = 1; i <= n; ++ i){
        //入队前只做两件事：1.去除不在窗口中的元素2.去除不可能成为答案的元素
        //队首元素下标最小，每入队一个元素，会超出窗口的一定是它，将它出队
        if(hh <= tt && q[hh] < i - k + 1) hh ++ ;
        //比要入队的元素大的元素一定不可能成为窗口内的最小值，因此出队
        while(hh <= tt && a[q[tt]] >= a[i]) tt --;
        q[++ tt] = i; //将要入队的元素入队
        if(i >= k) //窗口从右段从k开始的时候才输出最小值
                cout << a[q[hh]] << " ";
}

//求滑动窗口最大值
hh = 0, tt = -1;
for(int i = 1; i <= n; ++ i){
        if(hh <= tt && q[hh] < i - k + 1) hh ++ ;
        while(hh <= tt && a[q[tt]] <= a[i]) tt --;
        q[++ tt] = i;
        if(i >= k)
                cout << a[q[hh]] << " ";
        }
```

- **KMP $O(n+m)$**

```
int s[N], p[M];//主串和模式串，下标从1开始

//求next数组
void get_next(){
        for(int i = 2, j = 0; i <= m; ++ i){
                while(j && p[i] != p[j + 1]) j = ne[j];//j可以回跳并且匹配不成功时， j回跳
```

```
                if(p[i] == p[j + 1]) j ++ ; //匹配成功
                ne[i] = j; //存i可以回跳的位置
        }
}

//KMP
for(int i = 1, j = 0; i <= n; ++ i){
        while(j && s[i] != p[j + 1]) j = ne[j]; //同上
        if(s[i] == p[j + 1]) j ++ ;
        if(j == m){ //当j等于模式串的长度时匹配成功
                j = ne[j]; //继续进行下一次匹配
                cout << i - m << endl; //输出匹配的起点下标
        }
}
//j = ne[j] ，j往前跳，相当于字串往后移
```

- **字典树（Tire树）**

建树$O(n)$,查询$O(k)n$为字符串总长度，k为查询字符串长度

```
 int son[N][26], cnt[N], idx;//son[p][u] = x p表示父节点的编号，u表示p节点的儿子的名字，x表示儿子的编号
 //cnt[p]表示以p节点为结尾的单词的个数
 //编号0既代表根节点又代表空节点，所有没有儿子的节点都指向空节点
 void insert(char str[]){
        int p = 0;//编号指针
        for(int i = 0; str[i]; ++ i){ //遍历字符串的每一个字符
                int u = str[i] - 'a'; //求出每个字符的名字（a~z名字对应0~25）
                if(!son[p][u]) son[p][u] = ++ idx; //p节点没有u儿子，就添加u儿子并且给儿子分配编号为++idx
                p = son[p][u]; //指向儿子节点
        }
        cnt[p] ++ ; //循环完毕以p节点为结尾的单词个数加一
 }

 //查询某个字符串的次数
 int query(char str[]){
        int p = 0;
        for(int i = 1; str[i]; ++ i){
                int u = str[i] - 'a';
                if(!son[p][u]) return 0;
                p = son[p][u];
        }
        return cnt[p]; //虽然有些节点的名字一样但是编号各不相同，所以可以保证cnt[p]就是所要查询的字符串的个数
 }
```

- **最大异或对**

```
const int N = 1e5 + 10;
const int M = 3e6 + 10;
int a[N], son[M][2], idx;
int ans;

void insert(int x){
        int p = 0;
        for(int i = 30; ~i; -- i){
                int u = x >> i & 1;
                if(!son[p][u]) son[p][u] = ++ idx;
                p = son[p][u];
        }
}

int query(int x){
        int res = 0;//异或值
        int p = 0;
        for(int i = 30; ~i; -- i){
                int u = x >> i & 1;
                int v = 1 - u;
                if(son[p][v]){
                        res += 1 << i;
                        p = son[p][v];
                }else p = son[p][u];
        }
        return res;
}

for(int i = 1; i <= n; ++ i){
        cin >> a[i];
        insert(a[i]);
}
for(int i = 1; i <= n; ++ i) ans = max(ans, query(a[i]));
```

- **并查集**
  - **普通并查集** 初始化$O(n)$,查$O(lon_2(n))$,合并$O(1)$

```
int size[N], f[N]; //size统计根节点所在集合的元素个数，f存储每个节点的祖先节点
//初始化，每个节点的父节点为自己
void init(){
        for(int i = 1; i <= n; ++ i) f[i] = i;
}
```

```
//查找祖宗节点 + 路径压缩
//路径压缩：在找祖宗节点的过程中将路径上节点的父节点都修改为祖宗节点
int find(int x){
        return x == f[x] ? x : (f[x] = find(f[x]));
}

//合并x，y所在的集合
void merge(int x, int y){
        f[find(x)] = f[find(y)];
}

//判断x，y是否在同一个集合中
if(f[find(x)] == f[find(y)]) 在
else not 在
```

- **维护大小的并查集**

```
//需要统计集合元素个数时
//初始化
void init(){
        for(int i = 1; i <= n; ++ i){
                f[i] = i;
                size[i] = 1;
}
//合并集合
void merge(int x, int y){
        f[find(x)] = f[find(y)];
        size[find(x)] += size[find(y)];
}
```

- **维护距离的并查集**

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

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

```
        return p[x];
}

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

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

- **堆** $down/up$操作$O(log_2(n))$

```
//一般操作版
int h[N], idx;//h数组存储每个节点的值，idx表示最后一个节点的下标，所有节点下标从1开始

//down函数
void down(int u){
        int t = u, l = 2 * u, r = 2 * u + 1;//t用来指向三个节点中的最小值节点的下标，l，r表示u的左右儿子的下标
        if(l <= idx && h[l] < h[u]) t = l;//左儿子存在且值小于u，t更新为左儿子
        if(r <= idx && h[r] < h[u]) t = r;//右儿子存在且值小于u，t更新为右儿子
        if(t != u) swap(h[u], h[t]), down(t);//发生了更新说明需要u下移，递归处理
}

//up函数
void up(int u){
//不论u是左右儿子，其父节点下标均为u/2
        while(u / 2 && h[u / 2] > h[u]){//父节点存在且值比u小，就交换值
                swap(h[u], h[u / 2])
                u /= 2; //更新下标
        }
}

//插入值x
void add(int x){
        h[ ++ idx] = x;//赋值
        up(idx); //上升
}

//删除下标为k的节点
void del(int u){
```

```
        h[u] = h[idx];
        idx -- ;
        up(u);down(u);//要么往上要么往下
}

//将下标为u的节点的值改为x
void change(int k, int x){
        h[u] = x;
        up(u);down(u);
}

//初始化
void init(){
        //输入
        int x;
        for(int i = 1; i <= n; ++ i){
                cin >> x;
                h[ ++ idx] = x;
        }
        //建堆
        //建堆O(n),i不必从n开始因为有n/2的元素本来就在最底层
        for(int i = idx / 2; i; -- i) down(i);
}


//高级操作版
//支持修改第k个插入的节点

//h数组存储每个节点的值
//hp[i] = k;表示下标为i的点是第k个插入的节点
//ph[k] = i;表示第k个插入的节点的下标为i
//idx表示最后一个节点的下标，所有节点下标从1开始
//k记录第几次插入
int h[N], hp[N], ph[N], idx, k;

void swap_head(int u, int v){
        swap(h[u], h[v]);//交换值
        swap(ph[u], ph[v]);//交换下标
        swap(hp[ph[u]], hp[ph[v]]);//交换次序
}

//上面代码中的swap函数均需要替换为swap_head()函数
//down函数
void down(int u){
        int t = u, l = 2 * u, r = 2 * u + 1;//t用来指向三个节点中的最小值节点的下
标，l，r表示u的左右儿子的下标
```

```
        if(l <= idx && h[l] < h[u]) t = l;//左儿子存在且值小于u，t更新为左儿子
        if(r <= idx && h[r] < h[u]) t = r;//右儿子存在且值小于u，t更新为右儿子
        if(t != u) swap_head(u, t), down(t);//发生了更新说明需要u下移，递归处理
}

//up函数
void up(int u){
//不论u是左右儿子，其父节点下标均为u/2
        while(u / 2 && h[u / 2] > h[u]){//父节点存在且值比u小，就交换值
                swap_head(u, u / 2);
                u /= 2; //更新下标
        }
}

//插入x
void add(int x){
        ++ idx; ++ k;
        h[idx] = x;//赋值
        ph[k] = idx;
        hp[idx] = k;
        up(idx);
}

//删除下标为k的节点
void del(int u){
        h[u] = h[idx];
        idx -- ;
        up(u);down(u);
}

//将下标为u的节点的值改为x
void change(int k, int x){
        h[u] = x;
        up(u);down(u);
}

//初始化
void init(){
        //赋值
        int x;
        for(int i = 1; i <= n; ++ i){
                cin >> x;
                ++ idx; ++ k;
                h[idx] = x;
                hp[idx] = k;
                ph[k] = idx;
```

```
        }
        //建堆
        //建堆O(n),i不必从n开始因为有n/2的元素本来就在最底层
        for(int i = idx / 2; i; -- i) down(i);
}
```

- **大根堆**
  - 与小根堆互逆不多赘述
- **堆排序** $O(nlog_2(n))$

```
int h[N], idx;

void down(int u){
        int t = u, l = 2 * u, r = l + 1;
        if(l <= idx && h[l] < h[t]) t = l;
        if(r <= idx && h[r] < h[t]) t = r;
        if(t != u){
                swap(h[u], h[t]);
                down(t);
        }
}

for(int i = 1; i <= n; ++ i){
        int x;
        cin >> x;
        h[ ++ idx] = x;
}
for(int i = 1; i <= n; ++ i) down(i);
for(int i = 1; i <= n; ++ i){
        cout << h[1]; //输出最小值
        h[1] = h[idx];
        idx -- ;
        down(1);
}
```

- **哈希表**
  - **开放寻址法** 插/查$O(n)$

```
const int N = 2e5 + 3;
const int null = 0x3f3f3f3f;//表示无穷大
int h[N];
//查找 可以插入的/查找到的位置
void find(int x){
        int k = (x % N + N) % N;//第一个哈希的位置
```

```
        while(h[k] != null && h[k] != x){//当前位置有元素且不为x
                k ++ ;//后移
                if(k == N) k = 0;//循环找位置
        }
        return k;//返回可以插入的位置，或者查找到x的位置
}
int k = find(x);//返回一个可以插入的或者查找到的位置
//插入
if(h[k] == null) h[k] = x;//当前位置为空可以插入
//查询
if(h[k] != null) return found//当前位置不空必为x
else return not found //当前位置为空，查找不到x
memset(h, 0x3f, sizeof h);
```

- **拉链法** 插$O(1)$, 查$O(n)$

```
const int N = 1e5 + 3; //质数
int h[N], e[N], ne[N], idx = 1;//链表节点从1开始，0代表空节点
//添加
void insert(int x){
        int k = (x % N + N) % N;//x可能为负数，结果相当于|x| % N
        e[idx] = x;
        ne[idx] = h[k];
        h[k] = idx;
        idx ++ ;
}
//查询
void find(int x){
        int k = (x % N + N) % N;
        for(int i = h[k]; i; i = ne[i])
                if(e[i] == x) return true;
                else return false;
}
```

- **字符串哈希方式** 预处理$O(n)$, 查$O(1)$

```
typedef unsigned long long ULL;
const int N = 1e5 + 10;
const int P = 131; //P进制
char str[N];
ULL h[N], p[N];//h存储字符串前缀哈希值，p[i]存储P进制下的P^i的值
//取字串的哈希值
ULL gets(int l, int r){
        return h[r] - h[l - 1] * p[r - l + 1];
}
```

```
//递推预处理p^i, 字符串前缀哈希值
p[0] = 1;
for(int i = 1; i <= n; ++ i){
        p[i] = p[i - 1] * P;
        h[i] = h[i - 1] * P + str[i]; //求前i个字符组成的字符串的哈希值（字符串前缀哈希值）
}
```

- **DFS**
  - **求全排列** $O(n \cdot n!)$

```
const int N = 10;
int a[N];
int st[N];
int n;

void dfs(int x){
        if(x == n){ //结束
                for(int i = 0; i < n; ++ i) cout << a[i] << " ";
                cout << endl;
        }
        for(int i = 1; i <= n; ++ i){
                if(!st[i]){
                        st[i] = 1;
                        a[x] = i;
                        dfs(x + 1);
                        st[i] = 0; //回溯
                }
        }
        return ;
}

dfs(0);
```

  - **n-皇后** $O(n \cdot n!)$

```
const int N = 20; //N > 2 * n + 1
char g[N][N];
int row[N], rol[N], dg[N], udg[N];
int n;

void dfs(int x, int y, int s){
        if(y > n) y = 1, x ++ ;
```

```
        if(x > n){
                if(s > n){
                        for(int i = 1; i <= n; ++ i){
                                for(int j = 1; j <= n; ++ j)
                                        cout << g[i][j] ;
                        cout << endl;
                        }
                        cout << endl;
                }
                return ;
        }
        //选
        dfs(x, y + 1, s);
        //不选
        if(!rol[y] && !row[x] && !dg[x + y] && !udg[x - y + n]){
                g[x][y] = 'Q';
                rol[y] = row[x] = dg[x + y] = udg[x - y + n] = 1;
                dfs(x, y + 1, s + 1);
                rol[y] = row[x] = dg[x + y] = udg[x - y + n] = 0; //回溯
                g[x][y] = '.';
        }

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

        dfs(1, 1, 1);
        return 0;
}
```

- **BFS**
  - **走迷宫** $O(n)$

```
typedef pair<int, int> PII;
const int N = 110;
int n, m;
int g[N][N], d[N][N];

int bfs(){
        queue<PII> q; //队列
        memset(d, -1, sizeof d);
        d[0][0] = 0;
```

```
        q.push({0, 0});

        int dx[4] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1}; //增量数组

        while (q.size()){ //队列非空
                auto t = q.front();
                q.pop();

                for (int i = 0; i < 4; i ++ ){
                        int x = t.first + dx[i], y = t.second + dy[i];

                        if (x >= 0 && x < n && y >= 0 && y < m && g[x][y]
== 0 && d[x][y] == -1){
                                d[x][y] = d[t.first][t.second] + 1;
                                q.push({x, y});
                        }
                }
        }

        return d[n - 1][m - 1];
}
```

- **八数码**

```
queue<string> q;
unordered_map<string, int> d;
int dx[] = {0, 0, -1, 1};
int dy[] = {-1, 1, 0, 0};

int bfs(string s){
        q.push(s);
        d[s] = 0;

        string end = "12345678x";

        while(q.size()){
                string str = q.front();
                int dis = d[str];
                q.pop();
                if(str == end) return dis;

                int k = str.find('x');
                int x = k / 3, y = k % 3;

                for(int i = 0; i < 4; ++ i){
```

```
                        int xn = x + dx[i];
                        int yn = y + dy[i];
                        if(xn >= 0 && xn < 3 && yn >= 0 && yn < 3){
                                swap(str[k], str[xn * 3 + yn]);
                                if(!d.count(str)){
                                        d[str] = dis + 1;
                                        q.push(str);
                                }
                                swap(str[k], str[xn * 3 + yn]);
                        }
                }
        }
        return -1;
}
```

- **树与图的遍历**
  - **树与图的深度优先遍历**
    - **树的重心** $O(n+m)$

```
const int N = 1e5 + 10;
int h[N], e[N * 2], ne[N * 2], idx = 1;//无向图
bool st[N];
int ans = N, n;

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

int dfs(int u){
        st[u] = true;
        int res = 0, sum = 1;
        for(int i = h[u]; i; i = ne[i]){
                int j = e[i];
                if(!st[j]){
                        int s = dfs(j);
                        sum += s;
                        res = max(res, s);
                }
        }
        res = max(res, n - sum);
        ans = min(ans, res);
        return sum;
}
```

```
dfs(1);
```

- **树与图的深度优先遍历**
  - **图的层次** $O(n+m)$

```
const int N = 1e5 + 10;
int h[N], e[N], ne[N], idx = 1;
int d[N];
int n, m;
queue<int> q;

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

void bfs(int u){
        q.push(u);
        d[u] = 0;
        while(!q.empty()){
                int v = q.front();
                q.pop();
                for(int i = h[v]; i; i = ne[i]){
                        int j = e[i];
                        if(d[j] == -1){
                                q.push(j);
                                d[j] = d[v] + 1;
                        }
                }
        }
}

bfs(1);
```

- **拓扑排序** $O(n+m)$

```
//将所有入度为0的节点i优先入队
//删除以节点i为起点j为终点的所有边，同时更新j的入度
//再次将所有入度为0的节点j入队
//建图步骤省略
int q[N], hh, tt = -1;//队列
int d[N];//存储每个节点的入度
```

```
//初始化入度与图
cin >> a >> b;
add(a, b);
d[b] ++ ;

void topsort(){
        for(int i = 1; i <= n; ++ i)//遍历所有节点
                if(!d[i]) q[ ++ tt] = i;//将入度为0的节点优先入队
        //BFS
        while(hh <= tt){ //队列非空
                int x = q[hh ++ ]; //队首出队
                for(int i = h[x]; i; i = ne[i]){
                        int j = e[i];//遍历以x为起点的所有边的终点
                        d[j] -- ; //删边，更新入度
                        if(!d[j]) q[ ++ tt] = j; //将所有入度为0的点入队
                }
        }
        if(tt == n - 1) //所有点都在q数组时，存在拓扑序
                for(int i = 0; i < n; ++ i) cout << q[i];
        else cout << -1;//不存在拓扑序
}
```

- **最短路**
  - **朴素Dijkstra算法** $O(n^2)$

```
const int N = 510;
int g[N][N];//稠密图用邻接矩阵
int dist[N];//到原点的距离
int st[N];//标记数组

int dijkstra(){
        //初始化距离
        memset(d, 0x3f, sizeof d);
        d[1] = 0;

        //遍历n - 1次每次找距离原点最近的节点
        //起点已经被选中所以只需要迭代n - 1次
        for(int i = 1; i <= n - 1; ++ i){
                //在n个接节点中找
                int t = 0;
                for(int j = 1; j <= n; ++ j)
                        if(!st[j] && (!t || d[t] > d[j])
                                t = j; //找到了
        }
```

```
        st[t] = 1; //标记已经找到最短距离
        //更新所有节点的距离
        for(int j = 1; j <= n; ++ j)
                d[j] = min(d[j], d[t] + g[t][j]);

        if(d[n] == 0x3f3f3f3f) return -1;//1-n不连通
        else return d[n]; //返回1-n的最短距离
}

//注意初始化边权为inf
memset(g, 0x3f, sizeof g);
//输入边权
while(m -- ){
        int a, b, c;
        cin >> a >> b >> c;
        //有重边就取最小边权
        g[a][b] = min(g[a][b], c);
```

- **堆优化Dijkstra算法** $O(mlog_2(n))$

```
const int N = 1.5e5 + 10;
typedef pair<int, int> PII;
int st[N];
int h[N], e[N], ne[N], idx = 1;
int d[N], w[N];//w[i]边i的权值
int n, m;

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

int dijkstra(){
        //初始化
        memset(d, 0x3f, sizeof d);
        d[1] = 0;

        priority_queue<PII, vector<PII>, greater<PII> > q;//小根堆

        q.push({0, 1});//起点入堆

        while(q.size()){
```

```
                auto t = q.top();//取堆顶元素，离起点最近的节点
                q.pop();

                int dist = t.first, node = t.second;//获取信息

                if(st[node]) continue;//已经求出最短距离就跳过
                st[node] = 1;

                for(int i = h[node]; i; i = ne[i]){
                        int j = e[i];//遍历相连的所有节点
                        if(d[j] > dist + w[i]){//更新最短距离
                                d[j] = dist + w[i];
                                q.push({d[j], j});//入堆
                        }

                }
        }

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

- **Bellman-Ford算法** $O(nm)$

```
const int N = 510;
const int M = 1e4 + 10;

struct egde{ //边
        int u, v, w;
}edges[M];
int d[N], backup[N];//备份数组
int n, m, k;

int ballmen_ford(){
        //初始化
        memset(d, 0x3f, sizeof d);
        d[1] = 0;

        for(int i = 1; i <= k; ++ i){ //k条边限制
                memcpy(backup, d, sizeof d); //存备份
                for(int j = 1; j <= m; ++ j){ //遍历每一条边
                        int u = edges[j].u; //获取信息
                        int v = edges[j].v;
                        int w = edges[j].w;
                        d[v] = min(d[v], backup[u] + w); //利用备份更新
```

```
                }
        }

        if(d[n] > 0x3f3f3f3f / 2) return -N;
        else return d[n];
}

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

        for(int i = 1; i <= m; ++ i){
                int x, y, z;
                cin >> x >> y >> z;
                edges[i] = {x, y, z};
        }

        int t = ballmen_ford();

        if(t == -N) cout << "impossible";
        else cout << t << endl;
        return 0;
}
```

- **SPFA算法** 一般$O(m)$最坏$O(nm)$

```
const int N = 1e5 + 10;
int h[N], e[N], ne[N], idx = 1;
int d[N], w[N], cnt[N];//cnt[i]数组表示从起点到i的最短路中的边的数量
int st[N]; //标记元素是否已经进入队列
int n, m;

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

bool spfa(){
        queue<int> q;
        for(int i = 1; i <= n; ++ i){//所有节点入队，负环有可能不是从1-n路径上的，所以以每个节点为起点的路径都要检查
                q.push(i);
                st[i] = 1;//标记已入队
```

```
        }

        while(q.size()){
                int t = q.front();
                q.pop();
                st[t] = 0;//出队
                for(int i = h[t]; i; i = ne[i]){
                        int j = e[i];
                        if(d[j] > d[t] + w[i]){
                                d[j] = d[t] + w[i]; //更新
                                cnt[j] = cnt[t] + 1;//边数加一
                                if(cnt[j] >= n) return true;//边数>=n时，说明有环

                                if(!st[j]){
                                        q.push(j);//只将更新过的元素入队
                                        st[j] = 1;
                                }
                        }
                }
        }
        return false;
}

int main(){
        cin >> n >> m;
        while(m -- ){
                int x, y, z;
                cin >> x >> y >> z;
                add(x, y, z);
        }

        if(spfa()) cout << "Yes";
        else cout << "No";
        return 0;
}
```

- **SPFA算法** 一般$O(n)$最坏$O(nm)$

```
const int N = 1e5 + 10;
int h[N], e[N], ne[N], idx = 1;
int d[N], w[N];
int st[N]; //标记元素是否已经进入队列
int n, m;

void add(int a, int b, int z){
```

```
        e[idx] = b;
        ne[idx] = h[a];
        h[a] = idx;
        w[idx] = z;
        idx ++ ;
}

int spfa(){
        //初始化
        memset(d, 0x3f, sizeof d);
        d[1] = 0;
        //队列存储更新过距离的节点
        queue<int> q;
        q.push(1);
        st[1] = 1;//标记已入队

        while(q.size()){//循环
                int t = q.front();
                q.pop();
                st[t] = 0;
                for(int i = h[t]; i; i = ne[i]){
                        int j = e[i];
                        if(d[j] > d[t] + w[i]){
                                d[j] = d[t] + w[i]; //更新
                                if(!st[j]){
                                        q.push(j);//只将更新过的元素入队
                                        st[j] = 1;
                                }
                        }
                }
        }
        if(d[n] > 0x3f3f3f3f / 2) return -N;
        else return d[n];
}

int main(){
        cin >> n >> m;
        while(m -- ){
                int x, y, z;
                cin >> x >> y >> z;
                add(x, y, z);
        }

        int t = spfa();
        if(t == -N) cout << "impossible";
        else cout << t;
```

```
        return 0;
}
```

- **Floyd算法 $O(n^3)$**

```
const int N = 210;
const int M = 2e4 + 10;
const int INF = 0x3f3f3f3f;
int d[N][N];//存储两点间的最短路
int n, m, k;

void floyd(){
        //基于动态规划
        for(int k = 1; k <= n; ++ k)
                for(int i = 1; i <= n; ++ i)
                        for(int j = 1; j <= n; ++ j)
                                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}

int main(){
        cin >> n >> m >> k;
        //初始化
        for(int i = 1; i <= n; ++ i)
                for(int j = 1;j <= n; ++ j)
                        if(i == j) d[i][j] = 0; //去自环
                        else d[i][j] = INF;


        while(m -- ){
                int x, y, z;
                cin >> x >> y >> z;
                d[x][y] = min(d[x][y], z);//去重边
        }

        floyd();

        while(k -- ){
                int u, v;
                cin >> u >> v;
                int t = d[u][v];
                if(t > INF / 2) cout << "impossible" << endl;
                else cout << t << endl;
        }
```

```
        return 0;
}
```

- **最小生成树**
  - **朴素Prim算法** $O(n^2)$

```
const int N = 510;
const int INF = 0x3f3f3f3f;
int g[N][N];
int d[N];//非生成树节点与生成树节点的最短距离
int st[N];
int n, m;

int prim(){
        memset(d, 0x3f, sizeof d);//初始化

        int res = 0;//记录生成树权值和

        for(int i = 0; i < n; ++ i){//遍历n次选取n个点加入生成树

                int t = 0;
                for(int j = 1; j <= n; ++ j){
                        if(!st[j] && (!t || d[j] < d[t]))//在非生成树集合中选取最小距离的点
                                t = j;
                }
                st[t] = 1;//加入生成树集合

                if(i && d[t] == INF) return INF;//某个点与生成树集合不联通，则不存在生成树
                if(i) res += d[t]; //累加最小权值

                for(int j = 1; j <= n; ++ j) //更新其他点到生成树的距离
                        d[j] = min(d[j], g[t][j]);
        }

        return res;

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

        memset(g, 0x3f, sizeof g);

        while(m -- ){
```

```
                int u, v, w;
                cin >> u >> v >> w;
                g[u][v] = g[v][u] = min(g[u][v], w);//无向图去重边
        }

        int t = prim();
        if(t == INF) cout << "impossible";
        else cout << t;
        return 0;
}
```

- **堆优化版Prim** $O(mlog_2(n))$
  - 用堆优化找最短距离的过程将O(n)-->O(1)
- **Kruskal算法** $O(mlog_2(m))$

```
const int N = 2e5 + 10;
const int INF = 0x3f3f3f3f;
int f[N];//并查集表示两节点是否在生成树中
struct edge{
        int u, v, w;
        //运算符重载
        bool operator< (const edge &e)const{
                return w < e.w;
        }
}edges[N];

int n, m;

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

int kruskal(){
        sort(edges + 1, edges + 1 + m);//权值排序

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

        int res = 0, cnt = 0;//res记录总权值，cnt记录边的条数
        for(int i = 1; i <= m; ++ i){//遍历每条边
                int u = edges[i].u;//获取信息
                int v = edges[i].v;
                int w = edges[i].w;
                u = find(u), v = find(v);//祖宗节点
                if(u != v){//不在同一集合中
                        res += w;
```

```
                        cnt ++ ;
                        f[u] = v;//加入同一集合
                }
        }

        if(cnt < n - 1) return INF;//边数小于节点数-1，则不连通，不存在最小生成树
        else return res;
}
int main(){
        cin >> n >> m;

        for(int i = 1; i <= m; ++ i){
                int a, b, c;
                cin >> a >> b >> c;
                edges[i] = {a, b, c};
        }

        int t = kruskal();
        if(t == INF) cout << "impossible";
        else cout << t;
        return 0;
}
```

- **二分图**
  - **判别二分图-染色法** $O(m+n)$

```
const int N = 1e5 + 10;
const int M = 2 * N;
int h[N], e[M], ne[M], idx = 1;//无向图开两倍
int color[N];
int n, m;

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

//dfs染色，并且判断染色过程是否有矛盾
bool dfs(int u, int c){
        color[u] = c;//染色

        for(int i = h[u]; i; i = ne[i]){//遍历邻接点
```

```
                int j = e[i];
                if(!color[j]){//没有染色就染色
                        if(!dfs(j, 3 - c)) return false;//染色出现矛盾
                }else if(color[j] == c) return false;//已经染色，判断相邻点是否同色
        }
        return true;
}

bool check(){
        for(int i = 1; i <= n; ++ i)//以所有点为起点染色，因为可能不连通
                if(!color[i])
                        if(!dfs(i, 1)) return false;
        return true;
}

int main(){
        cin >> n >> m;
        while(m -- ){
                int u, v;
                cin >> u >> v;
                add(u, v);//无向图建边
                add(v, u);
        }

        if(check()) cout << "Yes";
        else cout << "No";
        return 0;
}
```

- **匈牙利算法** 最坏$O(mn)$
  - **求二分图的最大匹配数**

```
const int N = 510;
const int M = 1e5 + 10;
int h[N], e[M], ne[M], idx = 1;
int match[N];
int st[N];
int n1, n2, m;

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

```
bool find(int x){
    for(int i = h[x]; i; i = ne[i]){ //遍历该节点邻接的所有节点
        int j = e[i];
        if(!st[j]){
            st[j] = 1;//不管j有没有匹配过，强制标记为已经匹配，为了防止匹配j的节点再次将j匹配
            if(!match[j] || find(match[j])){ //j没有匹配或者匹配j的节点可以换一个匹配，在换匹配的过程中不会再匹配j，因为j已经被标记
                match[j] = x; //将x与j匹配
                return true; //成功匹配
            }
        }
    }
    return false;
}

int main(){
    cin >> n1 >> n2 >> m;

    while(m -- ){
        int a, b;
        cin >> a >> b;
        add(a, b); //只用一次
    }

    int res = 0;//记录最大匹配数
    for(int i = 1; i <= n1; ++ i){//匹配每一个节点
        memset(st, 0, sizeof st);
        if(find(i)) res ++ ; //匹配成功
    }

    cout << res;
    return 0;
}
```

- **质数**
  - **质数的判定**
    - **试除法** $O(\sqrt{n})$

```
//不用for(int i = 1; i * i <= n; ++ i) i * i 会溢出
//不用for(int i = 1; i <= sqrt(n); ++ i) sqrt() 函数缓慢
//用for(int i = 1; i <= n / i; ++ i)  不会溢出，快
bool is_prime(int x){
        if(x < 2) return false;
```

```
        for(int i = 2; i <= n / i; ++ i)
                if(n % i == 0) return false;
        return true;
}
```

- **分解质因数**
  - **试除法** $O(\sqrt{n})$

```
//暴力做法
void divide(int n){
        for(int i = 2; i <= n; ++ i){
                if(n % i == 0){
                        int s = 0; //i的个数
                        while(n % i == 0){
                                n /= i; //除干净
                                s ++ ;
                        }
                        cout << i << s << endl;
                }
        }
}
//优化
void divide(int n){
        for(int i = 2; i <= n / i; ++ i){
                if(n % i == 0){
                        int s = 0;
                        while(n % i == 0){
                                n /= i;
                                s ++ ;
                        }
                        cout << i << s << endl;
                }
        }
        if(n > 1) cout << n << 1;//n就是最后大于sqrt(n)的因子
}
```

- **筛选质数**
  - **埃氏筛法** $O(n)$到$O(nln(ln(n)))$

```
int primes[N];
int st[N];//标记是否被筛过
void get_primes(int n){
        int cnt = 0;
        for(int i = 2; i <= n; ++ i){
```

```
                if(!st[i]){ //没有被筛过，说明是质数
                        primes[ ++ cnt] = i;
                        //筛该质数的所有倍数
                        for(int j = i + i; j <= n; j += i) st[j] = 1;
                }
        }
}
```

- **线性筛法** $O(n)$

```
int primes[N];
int st[N];
void get_primes(int n){
        int cnt = 0;
        for(int i = 2; i <= n; ++ i){
                if(!st[i]) primes[ ++ cnt] = i;//没有被筛掉就是质数
                for(int j = 1; primes[j] <= n / i; ++ j){//从小到大遍历质数
                        st[primes[j] * i] = 1; //筛掉合数，若x是合数，当i遍历到x时，一定会先遍历到x的最小质因子prime，所以此时prime进入primes数组，当i遍历到x/prime时，x = prime*i会被筛掉，所以每个合数都能被筛掉，并且是在遍历到该合数之前被筛掉
                        if(i % pimes[j] == 0) break; //i是合数筛掉i，i在之前就被标记过，所以不用再标记
                }
        }
}
```

- **约数**
  - **求所有约数**
    - **试除法** $O(\sqrt{n})$

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

- **约数的个数**

```
const int mod = 1e7 + 10;
unordered_map<int, int> primes;//hash表存储每个质数有多少
void get_primes(int n){
        for(int i = 2; i <= n / i; ++ i){
                while(n % i == 0){
                        n /= i;
                        primes[i] ++ ;
                }
        }
        if(n > 1) primes[n] ++ ;
}
for(int i = 1; i <= n; ++ i) get_primes(a[i]);
long long res = 0;
for(auto p : primes) res = res * (p.second + 1) % mod;
```

- **约数之和**

```
const int mod = 1e7 + 10;
unordered_map<int, int> primes;
void get_primes(int n){
                for(int i = 2; i <= n / i; ++ i){
                        while(n % i == 0){
                                n /= i;
                                primes[i] ++ ;
                        }
                }
                if(n > 1) primes[n] ++ ;
        }
        for(int i = 1; i <= n; ++ i) get_primes(a[i]);
        long long res = 1;
        for(auto p : primes){
                long long t = 1;
                int a = p.first, b = p.second;
                while(b -- ) t = (t * a + 1) % mod;
                res = res * t % mod;
        }
```

- **最大公约数**
  - **辗转相除法（欧几里得算法）** $O(log_2(n))$

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

```
}
```

- **欧拉函数**
  - 欧拉函数$\varphi(N)$：1-N中与N互质的数的个数
  - 若$N=p_1^{a_1}\cdot p_2^{a_2}\cdot p_3^{a_3}\cdot\cdots\cdot p_n^{a_n}$ 其中p为N的所有质因子
  - 则$\varphi(N)=N(1-\frac{1}{p_1})(1-\frac{1}{p_2})\cdots(1-\frac{1}{p_n})$
  - **公式法** $O(\sqrt{n}+n)$

```
int res = a;
    for(int i = 2; i <= a / i; ++ i){
        if(a % i == 0){//i为质因子
            res = res / i * (i - 1);//套公式
            while(a % i == 0) a /= i;//把因子除干净
        }
    }
    if(a > 1) res = res / a * (a - 1);//最后一个因子可能大于sqrt(a)
```

  - **筛选法** $O(n)$
    - 利用线性筛选质数的过程求出每个数的欧拉函数

```
const int N = 1e6 + 10;
typedef long long LL;
int primes[N], cnt;//质数数组，下标
int st[N];//标记为合数
int phi[N];//欧拉函数
int n;

LL get_eulers(int n){
    phi[1] = 1;
    for(int i = 2; i <= n; ++ i){
        if(!st[i]){
            primes[ ++ cnt] = i;
            phi[i] = i - 1;//质数i的欧拉函数为i - 1
        }
        for(int j = 1; primes[j] <= n / i; ++ j){
            int pj = primes[j];
            st[pj * i] = 1;
            if(i % pj == 0){//i是合数
                phi[pj * i] = pj * phi[i];//pj为i的质因子
                break;
            }
            else phi[pj * i] = (pj - 1) * phi[i];//pj不是i的因子
        }
```

```
    }

    LL ans = 0;
    for(int i = 1; i <= n; ++ i) ans += phi[i];
    return ans;
}
```

- **欧拉函数的应用**
  - **欧拉定理**
    - 若a与n互质，则

$$a^{\varphi(n)} \pmod{n} \equiv 1$$

- **快速幂** $O(log(k))$

```
typedef long long LL;
//求a^k mod b
int qmi(int a, int k, int b){
        int res = 1;
        while(k){
                if(k & 1) res = (LL)res * a % b;//k的当前位非0，则将a累乘到答案
                k >>= 1;//k右移一位
                a = (LL)a * a % b;//a的幂倍增
        }
        return res;
}
```

  - **快速幂求逆元**
    - $b \cdot x = 1 \pmod{p} \rightarrow x = b^{p-2} \mod p / x = b^{\varphi(p)-1} \mod p$

```
int qmi(int a, int b, int p){
        略...
}
if(b与q互质){ //互质是有解的前提
        if(p为质数) cout << qmi(b, p - 2, p);
        else cout << qmi(b, phi[p] - 1, p);
}else 无解
```

- **扩展欧几里得定理** $O(log_2(n))$

```
int exgcd(int a, int b, int &x, int &y){
        if(!b){ //b = 0时，ax + by = gcd(a, 0) = a,所以x = 1, y = 0;
                x = 1, y = 0;
```

```
                return a;
        }else {
                int d = exgcd(b, a % b, x, y);//交换a与b，x与y
                int t = x;
                x = y;//x1 = y2
                y = t - a / b * y;//y1 = x2 - a / b * y2
                return d;
        }
}

//简化版
void exgcd(int a, int b, int &x, int &y){
        if(!b){
                x = 1, y = 0;
                return a;
        }else{
                int d = exgcd(b, a % b, y, x);
                y -= a / b * x;
                return d;
        }
}
```

- **扩展欧几里得求解线性同余方程**
  - 线性同余方程：$ax \equiv b \pmod{m}$

```
const int N = 1e5 + 10;
typedef long long LL;
int n;
int a, b, m;
int x, y;
//扩展欧几里得算法
int exgcd(int a, int b , int &x, int &y){
    if(!b){
        x = 1, y = 0;
        return a;
    }else{
        int t = exgcd(b, a % b, y, x);
        y -= a / b * x;
        return t;
    }
}

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

```
        cin >> a >> b >> m;
        int t = exgcd(a, m, x, y);
        //b是gcd(a,m)的倍数才有解
        if(b % t != 0) cout << "impossible" << endl;
        else cout << (LL)x * (b / t) % m << endl;//特解乘上倍数
    }
    return 0;
}
```

- **中国剩余定理**
  - 中国剩余定理 (Chinese Remainder Theorem, CRT) 可求解如下形式的一元线性同余方程组（其中 $n_1, n_2, n_3, \cdots n_k$ 两两互质）：
  - $$\begin{cases} x & \equiv a_1 \pmod{n_1} \\ x & \equiv a_2 \pmod{n_2} \\ & \vdots \\ x & \equiv a_k \pmod{n_k} \end{cases}$$
  - 过程：
    - 计算所有模数的积
    - 对于第 i 个方程：
      - 计算 $m_i = \frac{n}{n_i}$
      - 计算 m_i 在模 n_i 意义下的逆元 m_i^{-1}
      - 计算 $c_i = m_i m_i^{-1}$ （不要对 n_i 取模）
    - 方程组在模 n 意义下的唯一解为：$x = \sum_{i=1}^{k} a_i c_i \pmod{n}$
  - **扩展版中国剩余定理** $O(nlog_2(n))$
    - 当不满足 $n_1, n_2, n_3, \cdots n_k$ 两两互质时，求解线性同余方程组
    - $x \equiv a_1 \pmod{n_1} \Leftrightarrow x = k_1 n_1 + a_1$

```
const int N = 30;
typedef long long LL;
int n;

LL exgcd(LL a, LL b, LL &x, LL &y){
    if(!b){
        x = 1, y = 0;
        return a;
    }
    LL d = exgcd(b, a % b, y, x);
    y -= a / b * x;
    return d;
}
int main(){
```

```
    cin >> n;
    LL a1, m1;
    bool flag = true;
    cin >> a1 >> m1;//用于存储更新的a与m

    for(int i = 1; i < n; ++ i){ //合并n-1次
        LL an, mn;//接受新的a与m
        LL k1, k2; //存储用欧几里得求的系数
        cin >> an >> mn;
        LL d = exgcd(a1, an, k1, k2); //求出gcd(a1, an)以及系数k1, k2
        if((mn - m1) % d){ //无解判断
            flag = false;
            break;
        }
        LL t = an / d; //通解k1 = k1 + k * (an / d), k2 = k2 + k * (a1 / d),题目为了求最小的x，故要让k1为最小解故要k1不断膜上t
        k1 = k1 * (mn - m1) / d; //更新k1
        k1 = (k1 % t + t) % t; //取k1的最小解
        m1 = k1 * a1 + m1; //m更新为k1 * a1 * m1
        a1 = abs(a1 / d * an);  //a1更新为gcd(a1, an)
    }

    if(flag){
        cout << (m1 % a1 + a1) % a1; //最后一个式子的就是余数m
    }else cout << -1;
    return 0;
}
```

- **高斯消元 $O(n^3)$**
  - **高斯消元解普通方程组**

```
const int N = 110;
const double esp = 1e-6; //x < esp,则x = 0,否则 x != 0,由于浮点数精度问题,0可能是0.000001
double a[N][N];
int n;

int gauss(){
        int r, c;
        for(r = 1, c = 1; c <= n; ++ c){//遍历每一列
                int t = r;
                for(int i = r; i <= n; ++ i) //找当前列的系数最大的行
                        if(fabs(a[t][c]) < fabs(a[i][c]))
                                t = i; //记录最大行
```

```
                if(fabs(a[t][c]) < esp) continue; //最大行系数为0说明该列系数均为0

                for(int i = c; i <= n + 1; ++ i) swap(a[r][i], a[t][i]);//否则交换第一行与系数最大行
                for(int i = n + 1; i >= c; -- i) a[r][i] /= a[r][c]; //将第一行当前列系数变为1

                for(int i = r + 1; i <= n; ++ i) //将下面所有行的当前列的系数变为0
                        if(fabs(a[i][c]) > esp) //当前列系数非0
                                for(int j = n + 1; j >= c; -- j)
                                        a[i][j] -= a[r][j] * a[i][c];

                r ++ ;
        }

        if(r <= n){
                for(int i = r; i <= n; ++ i)
                        if(fabs(a[i][n + 1]) > esp) return 0; //无解
                return 1; //无数解
        }

        for(int i = n; i >= 1; -- i) //从后往前求解
                for(int j = i + 1; j <= n; ++ j)
                        a[i][n + 1] -= a[i][j] * a[j][n + 1];

        return 2; //唯一解
}
```

- **高斯消元解异或方程组**
  - 异或方程组：系数只有0和1，一个方程由若干解异或起来，如：
    $x_1 \wedge x_2 \wedge x_3 \cdots \wedge x_n = 1$

```
const int N = 110;
int a[N][N];
int n;

int gauss(){
    int r, c;
    for(r = c = 1; c <= n; ++ c){
        int t = r;
        for(int i = r; i <= n; ++ i)
            if(a[i][c]){
                t = i;
                break;
            }
```

```
        if(!a[t][c]) continue;
        for(int i = c; i <= n + 1; ++ i) swap(a[t][i], a[r][i]);

        for(int i = r + 1; i <= n; ++ i)
            if(a[i][c])
                for(int j = n + 1; j >= c; -- j)
                    a[i][j] ^= a[r][j];

        r ++ ;
    }

    if(r <= n){
        for(int i = r; i <= n; ++ i)
            if(a[i][n + 1]) return 3;
        return 2;
    }

    for(int i = n; i >= 1; -- i)
        for(int j = i + 1; j <= n; ++ j)
            a[i][n + 1] ^= a[i][j] & a[j][n + 1];
    return 1;
}
```

- **求组合数**
  - $C_a^b = \frac{a(a-1)(a-2)\cdots(a-b+1)}{b!} = \frac{a!}{b!(a-b)!}$
  - 递推式：$C_a^b = C_{a-1}^{b-1} + C_{a-1}^b$ $O(N^2)$
  - 当a，b <= 2e3，询问n = 1e4次时 $O(n^2)$

```
//询问10000次
//a,b <= 2000
const int N = 2e3 + 10;
const int mod = 1e9 + 7;
int c[N][N];
int n;

void init(){
    for(int i = 0; i < N; ++ i)
        for(int j = 0; j <= i; ++ j)
            if(!j) c[i][j] = 1;
            else c[i][j] = (c[i - 1][j - 1] + c[i - 1][j]) % mod;
}
```

  - 当a，b <= 1e5，询问n = 1e4次，模数p = 1e9 + 7时
    - 开二维数组会爆内存，于是利用公式，预处理出阶乘以及阶乘的逆 $O(Nlog(N))$

- 代码：

```
typedef long long LL;
const int N = 1e5 + 10;
const int mod = 1e9 + 7;
int fact[N], infact[N];
int n;

//快速幂
int qmi(int a, int b, int p){
    int res = 1;
    while(b){
        if(b & 1) res = (LL)res * a % p;
        a = (LL)a * a % p;
        b >>= 1;
    }
    return res;
}

//预处理各阶乘以及阶乘的逆
void init(){
    fact[0] = infact[0] = 1;
    for(int i = 1; i < N; ++ i){
        fact[i] = (LL)fact[i - 1] * i % mod; //递推式n! = (n - 1)! * n;
        infact[i] = (LL)infact[i - 1] * qmi(i, mod - 2, mod) % mod; //
递推式(n!)^-1 = ((n - 1)!)^-1 * n^-1，取模意义下的逆就是逆元
    }
}

int main(){
    cin >> n;
    init();
    while(n -- ){
        int a, b;
        cin >> a >> b;
        cout << (LL)fact[a] * infact[b] % mod * infact[a - b] % mod <<
endl;//两个int最大值相乘不会爆longlong，但是三个相乘会爆
    }
    return 0;
}
```

- 当a，b <= 1e18，询问n = 2e5次，模数p不定时
  - 两数相乘会爆longlong，利用卢卡斯定理预处理组合数 $O(plog(N)log(p))$
  - 定理：$C_a^b \equiv C_{a \bmod p}^{b \bmod p} \cdot C_{a/p}^{b/p} \pmod p$
  - 代码：

```
typedef long long LL;
int p;
int n;

//快速幂求逆元
int qmi(int a, int b){
    int res = 1;
    while(b){
        if(b & 1) res = (LL)res * a % p;
        a = (LL)a * a % p;
        b >>= 1;
    }
    return res;
}

int C(int a, int b){ //求组合数C_a^b
    if(a < b) return 0; //可以不要
    int x = 1, y = 1; //分子分母
    for(int i = 0; i < b; ++ i){
        x = (LL)x * (a - i) % p; //求分子
        y = (LL)y * (i + 1) % p; //求分母
    }
    return (LL)x * qmi(y, p - 2) % p;
}

//卢卡斯定理
int lucas(LL a, LL b){
    if(a < p && b < p) return C(a, b); //不必用卢卡斯，可以直接利用公式求出
    return (LL)C(a % p, b % p) * lucas(a / p, b / p) % p; //用卢卡斯定理
}

int main(){
    cin >> n;
    while(n -- ){
        LL a, b;
        cin >> a >> b >> p;
        cout << lucas(a, b) << endl;
    }
    return 0;
}
```

- 当a，b <= 5e5，询问一次，不取模时
  - 此时组合数较大，需要用高精度存储结果
  - 阶乘中素数p的次数：

- $v_p(n!) = \sum_{i=1}^{\infty} \left\lfloor \frac{n}{p^i} \right\rfloor$
- 证明：将 $n!$ 记为 $1 \times 2 \times \cdots \times p \times \cdots \times 2p \times \cdots \times \lfloor n/p \rfloor p \times \cdots \times n$ 那么其中 p 的倍数有 $p \times 2p \times \cdots \times \lfloor n/p \rfloor p = p^{\lfloor n/p \rfloor} \lfloor n/p \rfloor!$ 然后在 $\lfloor n/p \rfloor!$ 中继续寻找 p 的倍数即可，这是一个递归的过程

- 代码：

```
const int N = 5e3 + 10;
int primes[N], cnt;
int st[N];
int sum[N];

//线性筛，1-n中质数就是n!中的质因子
void get_primes(int a){
    for(int i = 2; i <= a; ++ i){
        if(!st[i]) primes[ ++ cnt] = i;
        for(int j = 1; primes[j] <= a / i; ++ j){
            st[primes[j] * i] = 1;
            if(i % primes[j] == 0) break;
        }
    }
}

//求a!中质数p的个数
int get(int a, int p){
    int res = 0;
    while(a){
        res += a / p;
        a /= p;
    }
    return res;
}

//高精度乘法
vector<int> mul(vector<int> &A, int b){
    vector<int> c;
    int t = 0;
    for(int i = 0; i < A.size() || t; ++ i){
        if(i < A.size()) t += A[i] * b;
        c.push_back(t % 10);
        t /= 10;
    }
    while(c.size() > 1 && c.back() == 0) c.pop_back();
    return c;
}
```

```
int main(){
        int a, b;
    cin >> a >> b;
    get_primes(a);
    for(int i = 1; i <= cnt; ++ i){
        int p = primes[i];
        sum[i] = get(a, p) - get(b, p) - get(a - b, p); //a!中质因子个数-b!中质因子个数-(a-b)!中质因子个数
    }

    vector<int> res;
    res.push_back(1);

    for(int i = 1; i <= cnt; ++ i) //累乘
        for(int j = 1; j <= sum[i]; ++ j)
            res = mul(res, primes[i]);

    for(int i = res.size() - 1; i >= 0; -- i)//输出
        cout << res[i];

    return 0;
}
```

- **卡特兰数**
  - $C_{2n}^{n} - C_{2n}^{n-1} = \frac{1}{n+1} \cdot C_{2n}^{n}$
  - 用于解决序列排列等问题 $O(n^2 + log_2(n))$

```
typedef long long LL;
const int N = 1e5 + 10;
const int p = 1e9 + 7;
int n;

int qmi(int a, int k){
    int res = 1;
    while(k){
        if(k & 1) res = (LL)res * a % p;
        a = (LL)a * a % p;
        k >>= 1;
    }
    return res;
}

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

```
    int res = 1;

    int a = 2 * n, b = n;
    int x = 1, y = 1;
    for(int i = 0; i < b; ++ i){
        x = (LL)x * (a - i) % p;
        y = (LL)y * (i + 1) % p;
    }

    res = (LL)x * qmi(y, p - 2) % p;
    res = (LL)res * qmi(n + 1, p - 2) % p;
    cout << res;

    return 0;
}
```

- **容斥原理** $O(n \cdot 2^n)$

```
typedef long long LL;
const int N = 20;
int p[N];
int n, m;

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

        int res = 0; //记录答案
        for(int i = 1; i < 1 << m; ++ i){ //遍历2^m-1次，即遍历所有的方案
                int t = 1, cnt = 0; //记录当前p的倍数，以及p的个数
                for(int j = 1; j <= m; ++ j){ //遍历m位，求出当前方案选取的质数及个数
                        if(i >> (j - 1) & 1){
                                cnt ++ ; //选取了当前位的质数，质数加一
                                if((LL)t * p[j] > n){ //判断质数的倍数是否超出范围
                                        t = -1;
                                        break;
                                }
                                t = (LL)t * p[j]; //累乘当前位的质数
                        }
                }
                if(t != -1){
                        if(cnt % 2) res += n / t; //个数质数位奇数，则取加号
                        else res -= n / t; //否则取减号
                }
```

```
        }

        cout << res;
        return 0;
}
```

- **博弈论**
  - **集合-NIM游戏** $O(n+m)$
    - n堆石子，玩家可以从任意一堆石子中取走石子，但是每次取的石子个数必须在一个已知的集合S中，最后无法操作的人判负
    - 结论：当每堆石子的sg异或值 $sg(a_1) \wedge sg(a_2) \wedge sg(a_3) \cdots \wedge sg(a_n) = 0$ 时 先手必败，否则先手必胜

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


using namespace std;

const int N = 110;
const int M = 1e4 + 10;
int s[N], f[M]; //s[i]存储可以取的石子数，f[i]存储每个状态的sg值，每个状态用当前石子数表示
int n, k;

//dfs求sg值
int sg(int x){
    if(f[x] != -1) return f[x]; //记忆化搜索

    unordered_set<int> S; //储存可以走到的局面的sg值
    for(int i = 1; i <= k; ++ i){ //遍历所有可以取的石子数量
        int sum = s[i];
        if(x >= sum) S.insert(sg(x - sum)); //将走到的局面的sg值存储
    }

    for(int i = 0; ; ++ i) //从小到大遍历可能的sg值
        if(!S.count(i)) //若该sg值是不能达到的sg中的最小值
            return f[x] = i; //取该sg值
}

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

```
    cin >> n;

    memset(f, -1, sizeof f);//别忘了初始化

    int res = 0;
    while(n -- ){
        int h;
        cin >> h;
        res = res ^ sg(h); //起点的sg异或值
    }

    if(res) cout << "Yes";
    else cout << "No";
    return 0;
}
```

- **拆分-NIM游戏** $O(n+m)$
  - 给定 n堆石子，两位玩家轮流操作，每次操作可以取走其中的一堆石子，然后放入两堆**规模更小**的石子（新堆规模可以为 0，且两个新堆的石子总数可以大于取走的那堆石子数），最后无法进行操作的人视为失败。
  - 结论：==当每堆石子的sg异或值 $sg(a_1) \wedge sg(a_2) \wedge sg(a_3) \cdots \wedge sg(a_n) = 0$ 时 先手必败，否则先手必胜

```
#include <iostream>
#include <cstring>
#include <unordered_set>

using namespace std;

const int N = 110;
int f[N];
int n;

int sg(int x){
    if(f[x] != -1) return f[x];//记忆化

    unordered_set<int> s;
    for(int i = 0; i < x; ++ i)
        for(int j = 0; j <= i; ++ j) //避免重复，约定第二堆的数量小于等于第一堆
            s.insert(sg(i) ^ sg(j)); //两个石子为一个局面，由sg定理：两个石子的sg异或值为当前局面的sg值

    for(int i = 0; ; ++ i) //选取不存在的最小的自然数
        if(!s.count(i))
```

```
            return f[x] = i;
}

int main(){
    cin >> n;
    memset(f, -1, sizeof f);
    int res = 0;
    while(n -- ){
        int x;
        cin >> x;
        res ^= sg(x);
    }

    if(res) cout << "Yes";
    else cout << "No";
    return 0;
}
```

- **背包问题**
  - **01背包** $O(nm)$
    - 每件物品最多取一次

```
const int N = 1e3 + 10;
int f[N][N], v[N], w[N];
int n, m;

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

    //f[1~n][0] = f[0][1~m] = 0;
    for(int i = 1; i <= n; ++ i) //遍历物品
        for(int j = 1; j <= m; ++ j){ //遍历容量
            f[i][j] = f[i - 1][j]; //不选第i个物品
            if(j >= v[i])
                f[i][j] = max(f[i][j], f[i - 1][j - v[i]] + w[i]); //选第i个物品
        }

    cout << f[n][m];
    return 0;
}
```

    - 滚动数组优化：

```cpp
const int N = 1e3 + 10;
int f[N], v[N], w[N];
int n, m;

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

        //f[0] = 0;
        for(int i = 1; i <= n; ++ i) //遍历物品
                for(int j = m; j >= v[i]; -- j) //从小往大遍历容量
                                f[j] = max(f[j], f[j - v[i]] + w[i]);
//选第i个物品

        cout << f[m];
        return 0;
}
```

- **完全背包**
  - 每件物品可取无限次 $O(nmk)$

```cpp
const int N = 1e3 + 10;
int f[N][N], w[N], v[N];
int n, m;

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

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

    for(int i = 1; i <= n; ++ i) //遍历物品
        for(int j = 1; j <= m; ++ j) //遍历容量
            for(int k = 0; k * v[i] <= j; ++ k) //遍历个数
                f[i][j] = max(f[i][j], f[i - 1][j - k * v[i]] + k *
w[i]);

    cout << f[n][m];
    return 0;
}
```

  - 时间优化 $O(nm)$
    - 发现: $f[i][j] = max(f[i-1][j], f[i][j-v] + w)$

```
const int N = 1e3 + 10;
int f[N][N], w[N], v[N];
int n, m;

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

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

    for(int i = 1; i <= n; ++ i) //遍历物品
        for(int j = 1; j <= m; ++ j){ //遍历容量
            f[i][j] = f[i - 1][j]; //第i个物品不选
            if(j >= v[i])
                f[i][j] = max(f[i][j], f[i][j - v[i]] + w[i]);//第i
个物品选
        }

    cout << f[n][m];
    return 0;
}
```

- 滚动数组优化

```
const int N = 1e3 + 10;
int f[N], w[N], v[N];
int n, m;

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

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

        for(int i = 1; i <= n; ++ i) //遍历物品
                for(int j = v[i]; j <= m; ++ j) //遍历容量
                        f[j] = max(f[j], f[j - v[i]] + w[i]);

        cout << f[m];
        return 0;
}
```

- **多重背包**
  - 每件物品可取有限次 $O(nmk)$

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

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

    for(int i = 1; i <= n; ++ i)
        for(int j = 1; j <= m; ++ j)
            for(int k = 0; k <= s[i] && k * v[i] <= j; ++ k)
                f[i][j] = max(f[i][j], f[i - 1][j - k * v[i]] + k *
w[i]);

    cout << f[n][m];
    return 0;
}
```

- 滚动数组优化

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

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

        for(int i = 1; i <= n; ++ i)
                for(int j = m; j >= v[i]; -- j)
                        for(int k = 0; k <= s[i] && k * v[i] <= j; ++
k)
                                f[j] = max(f[j], f[j - k * v[i]] + k *
w[i]);

        cout << f[m];
        return 0;
}
```

- 二进制优化： $O(nm)$

```
const int N = 2e4 + 500;
int v[N], w[N], s[N];
int f[N];
int n, m;
```

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

        int cnt = 0; //记录物品个数
        while(n -- ){
                int V, W, S;
                cin >> V >> W >> S;
                int k = 1; //记录分解后每个物品的次数
                while(k <= S){ //将数量S分解
                        cnt ++ ; //每次分解个数加一
                        w[cnt] = W * k;
                        v[cnt] = V * k;
                        S -= k;
                        k *= 2;
                }
                if(S > 0){ //剩余次数
                        cnt ++ ;
                        w[cnt] = W * S;
                        v[cnt] = V * S;
                }
        }

        n = cnt; //01背包问题
        for(int i = 1; i <= n; ++ i)
                for(int j = m; j >= v[i]; -- j)
                        f[j] = max(f[j], f[j - v[i]] + w[i]);

        cout << f[m];
        return 0;
}
```

- **分组背包** $O(nmk)$
  - **每个组里面最多选一件物品**

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

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

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

```
        cin >> s[i];
        for(int j = 1; j <= s[i]; ++ j)
            cin >> v[i][j] >> w[i][j];
    }

    for(int i = 1; i <= n; ++ i)
        for(int j = 1; j <= m; ++ j){
            f[i][j] = f[i - 1][j]; //该组不选物品
            for(int k = 1; k <= s[i]; ++ k){ //该组选物品
                if(j >= v[i][k])
                    f[i][j] = max(f[i][j], f[i - 1][j - v[i][k]] + w[i]
[k]);
            }
        }


    cout << f[n][m];
    return 0;
}

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

- **滚动数组优化：**

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

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

    for(int i = 1; i <= n; ++ i){
        cin >> s[i];
        for(int j = 1; j <= s[i]; ++ j)
            cin >> v[i][j] >> w[i][j];
```

```
    }

    for(int i = 1; i <= n; ++ i)
        for(int j = m; j >= 1; -- j)
            for(int k = 1; k <= s[i]; ++ k)
                if(j >= v[i][k])
                    f[j] = max(f[j], f[j - v[i][k]] + w[i][k]);


    cout << f[m];
    return 0;
}
```

- **线性DP**
  - **数字三角形**
  - **状态表示**：$f[i][j]$ 表示从(1,1)到(i, j)的路径上数字之和集合
  - **状态计算**：$f[i][j] = max(f[i-1][j-1] + a[i][j], f[i-1][j] + a[i][j]);$
  - 代码：

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

using namespace std;

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

int main(){
    cin >> n;

    for(int i = 1; i <= n; ++ i)
        for(int j = 1; j <= i; ++ j)
            cin >> a[i][j];
    //初始化为-INF
    for(int i = 1; i <= n; ++ i)
        for(int j = 0; j <= i + 1; ++ j) //每行多初始化左右两个
            f[i][j] = -INF;

    //递推
    for(int i = 1; i <= n; ++ i)
        for(int j = 1; j <= i; ++ j)
            f[i][j] = max(f[i - 1][j - 1] + a[i][j], f[i - 1][j] + a[i]
```

```
[j]);

    //从最后一行中找最大值
    int res = -INF;
    for(int i = 1; i <= n; ++ i)
        res = max(res, f[n][i]);
    cout << res;

    return 0;
}
```

- **最长上升子序列 I**
- 状态表示：$f[i]$ 表示以 $a[i]$ 结尾的上升子序列的长度的集合
- 状态计算：$if(a[j] < a[i])\quad f[i] = max(f[i], f[j] + 1)$

```
#include <iostream>

using namespace std;

const int N = 1e3 + 10;
int f[N], a[N];
int n;


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

        for(int i = 1; i <= n; ++ i){ //遍历每个状态
                f[i] = 1; //只含a[i]一个数，最小是1
                for(int j = 1; j < i; ++ j) //遍历前面所有小于a[i]的数
                        if(a[j] < a[i])
                                f[i] = max(f[i], f[j] + 1); //状态转移
        }

        int res = 0;
        for(int i = 1; i <= n; ++ i)
                res = max(res, f[i]);
        cout << res;
        return 0;
}
```

- **最长上升子序列 II**
    - 优化I中的算法到 O(nlog(n))

```
const int N = 1e5 + 10;
int q[N];
int a[N];
int n;

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

    int len = 0;
    for(int i = 1; i <= n; ++ i){
        int l = 0, r = len + 1; //二分查找
        while(l + 1 < r){
            int mid = l + r >> 1;
            if(q[mid] < a[i]) l = mid;
            else r = mid;
        }

        len = max(len, l + 1); //更新长度
        q[l + 1] = a[i]; //替换/插入
    }
    cout << len;
    return 0;
}
```

- **最长公共子序列**
- 状态表示：$f[i][j]$ 表示A串中前i个字符，B串中前j个字符的公共子序列的集合

```
const int N = 1e3 + 10;
int f[N][N];
char a[N], b[N];
int n, m;

int main(){
    cin >> n >> m;
    cin >> a + 1 >> b + 1;

    for(int i = 1; i <= n; ++ i)
        for(int j = 1; j <= m; ++ j){
            f[i][j] = max(f[i - 1][j], f[i][j - 1]);
            if(a[i] == b[j])
                f[i][j] = max(f[i][j], f[i - 1][j - 1] + 1);
        }

    cout << f[n][m];
```

```
    return 0;
}
```

- **编辑距离**
  - 状态表示：$f[i][j]$ 表示将原序列中 $a[1 \to i]$ 个字母与原序列中 $b[1 \to j]$ 个字母匹配的方案的集合

```
const int N = 1e3 + 10;
int f[N][N];
char a[N], b[N];
int n, m;

int main(){
    cin >> n >> a + 1 >> m >> b + 1;

    for(int i = 1; i <= n; ++ i) f[i][0] = i;
    for(int i = 1; i <= m; ++ i) f[0][i] = i;

    for(int i = 1; i <= n; ++ i)
        for(int j = 1; j <= m; ++ j){
            f[i][j] = min(f[i - 1][j] + 1, f[i][j - 1] + 1);
            if(a[i] == b[j]) f[i][j] = min(f[i][j], f[i - 1][j - 1]);
            else f[i][j] = min(f[i][j], f[i - 1][j - 1] + 1);
            //f[i][j] = min(f[i][j], f[i - 1][j - 1] + !(a[i] ==
b[j]));
        }
    cout << f[n][m];
    return 0;
}
```

- **区间DP**
  - **石子合并**
    - 状态表示：$f[i][j]$ 所有表示从第i堆石子和并到第j堆石子的合并所产生的代价

```
const int N = 310;
int f[N][N];
int s[N];
int n;

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

```
    }

    for(int len = 2; len <= n; ++ len) //遍历区间长度
        for(int i = 1; i <= n - len + 1; ++ i){ //遍历起点
            int l = i, r = i + len - 1; //确定区间边界
            f[l][r] = 1e8;
            for(int k = l; k < r; ++ k) //分割区间
                f[l][r] = min(f[l][r], f[l][k] + f[k + 1][r] + s[r] -
s[l - 1]);//状态转移
        }

    cout << f[1][n];

    return 0;
}
```

- **计数类DP**
  - **整数划分**
    - 法一：
    - 利用完全背包，第i个物品的体积为i
    - 状态表示：$f[i][j]$ 表示前i个数中选，总体积等于j的方案数
    - 代码：

```
const int N = 1e3 + 10, mod = 1e9 + 7;
int f[N];
int n;

int main(){
    cin >> n;

    f[0] = 1;

    for(int i = 1; i <= n; ++ i)
        for(int j = i; j <= n; ++ j)
            f[j] = (f[j] + f[j - i]) % mod; //属性：sum

    cout << f[n];
    return 0;
}
```

- 法二：
- 状态表示：$f[i][j]$ 表示总和为i，并且总和恰好由j个数相加而成的方案数
- 所以：$f[i][j] = f[i-1][j-1] + f[i-j][j]$

```
const int N = 1e3 + 10, mod = 1e9 + 7;
int f[N][N];
int n;

int main(){
    cin >> n;

    f[0][0] = 1;

    for(int i = 1; i <= n; ++ i)
        for(int j = 1; j <= i; ++ j)
            f[i][j] = (f[i - 1][j - 1] + f[i - j][j]) % mod;


    int ans = 0;
    for(int i = 1; i <= n; ++ i)
        ans = (ans + f[n][i]) % mod;
    cout << ans;
    return 0;
}
```

- **数位统计DP**
  - **计数问题**

```
const int N = 1e8 + 10;
int a, b;

int power10(int x){ //求10^i
        int res = 1;
        while(x -- ) res = res * 10;
        return res;
}
int get(vector<int> num, int l, int r){ //求l~r位对应的数
        int res = 0;
        for(int i = l; i >= r; -- i)
                res = res * 10 + num[i];
        return res;
}

int count(int n, int x){ //统计x在1~n中出现的次数
```

```
        if(!n) return 0;
        vector<int> num; //存储n的每一位数
        while(n){
                num.push_back(n % 10);
                n /= 10;
        }
        n = num.size();
        int res = 0; //答案
        for(int i = n - 1 - !x; i >= 0; -- i){ //逆序遍历，当x为0是只用遍历1~9位，最高位不可能为0
                //前缀不相等
                if(i < n - 1){ //i不在第一位，表示i有前缀
                        res += get(num, n - 1, i + 1) * power10(i); //+前缀数字 * 10 ^ 后缀位数
                        if(!x) res -= power10(i); //当x为0时前缀从1开始
                }
                //前缀相等
                if(num[i] == x) res += get(num, i - 1, 0) + 1; //+后缀数字
                else if(num[i] > x) res += power10(i); //+10 ^ 后缀位数
        }
        return res;
}
int main(){

        while(cin >> a >> b, a || b){
                if(a > b) swap(a, b); //保持a < b
                for(int i = 0; i <= 9; ++ i)
                        cout << count(b, i) - count(a - 1, i) << " ";//1~b内i的个数-1~a-1内i的个数，就是a~b内i的个数
                cout << endl;
        }

        return 0;
}
```

- **状态压缩DP**
  - **蒙德里安的梦想**
    - **状态表示**：$f[i][j]$ 表示横条的最右端恰好在第i列并且每行的摆放为j的方案集合。其中j为n位二进制数，每一位表示每一行的摆放状态，该位为1表示该行已经摆放，否则该行没有摆放
    - **状态计算**：$f[i][j] = f[i][j] + f[i-1][k]$ 总方案数等于摆放每一列的方案数累加

```
const int N = 12, M = 1 << N;
long long f[N][M]; //f[i][j] 表示横条在第i列摆放为j的状态， j为n位二进制
```

```
数，每一位表示每一行的摆放情况
int st[M]; //对所有的二进制数预处理合法情况
int n, m;

//所有摆放情况就是横条的摆放情况
int main(){

    while(cin >> n >> m, n || m){
        //预处理所有合法状态
        for(int i = 0; i < 1 << n; ++ i){ //遍历每个二进制数
            int cnt = 0; st[i] = 1; //统计连续0的个数
            for(int j = 0; j < n; ++ j) //遍历每一位
                if(i >> j & 1){ //当前位为1说明上一段0统计完成
                    if(cnt & 1) st[i] = 0; //连续0的个数为奇数个说明不合
法
                    cnt = 0;
                }else cnt ++ ;
            if(cnt & 1) st[i] = 0; //最后一段连续0
        }

        memset(f, 0, sizeof f); //每次循环要清空
        f[0][0] = 1; //边界初始化
        for(int i = 1; i <= m; ++ i) //遍历每一列
            for(int j = 0; j < 1 << n; ++ j) //遍历i列的每一种摆放状态
                for(int k = 0; k < 1 << n; ++ k) //遍历i-1列的每一种摆放
状态
                    if(!(k & j) && st[k | j]) //相邻两列的摆放状态不矛盾
                        f[i][j] += f[i - 1][k]; //累加方案数

        cout << f[m][0] << endl;
    }


    return 0;
}
```

- **最短Hamilton路径**
  - **状态表示**：$f[i][j]$ 表示经过 0~j 中的点，并且经过所有点的压缩状态为 i 的所有方案
  - **状态计算**：$f[i][j] = min(f[i][j], f[i - (1 << j)][k] + w[k][j])$

```
const int N = 20, M = 1 << N;

int n;
int w[N][N];
```

```
int f[M][N];

int main(){
    cin >> n;

    for (int i = 0; i < n; i ++ )
        for (int j = 0; j < n; j ++ )
            cin >> w[i][j];

    memset(f, 0x3f, sizeof f);
    f[1][0] = 0;

    for (int i = 1; i < 1 << n; i += 2 ) //起点0总是在路径中，所以压缩状态始终为奇数
        for (int j = 0; j < n; j ++ ) //遍历所有节点
            if (i >> j & 1) //节点在路径中
                for (int k = 0; k < n; k ++ ) //遍历倒数第二个节点
                    if (i - (1 << j) >> k & 1) //节点在路径中
                        f[i][j] = min(f[i][j], f[i - (1 << j)][k] + w[k][j]); //状态转移

    cout << f[(1 << n) - 1][n - 1]; //最终状态

    return 0;
}
```

- **树形DP**
  - **状态表示：$f[u][1]$ 表示以u为根节点且选择节点u的树的最大快乐指数，$f[u][0]$ 表示以u为根节点且不选择节点u的树的最大快乐指数**
  - **状态计算：$f[u][0]+=f[j][1]$ and $f[u][1]+=max(f[j][0],f[j][1])$**

```
const int N = 6010;
int n;
int h[N], e[N], ne[N], idx = 1;
int f[N][2]; //f[u][1]表示以u为根节点且包括节点u的最大快乐指数，f[u][0]表示以u为根节点且不包括节点u的最大快乐指数
int happy[N];
int has_father[N]; //标记是否有父节点，以便后续找到根节点


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

```
void dfs(int u){ //dfs
        f[u][1] = happy[u]; //选择节点u

        for(int i = h[u]; i; i = ne[i]){ //遍历所有儿子节点
                int j = e[i];

                dfs(j); //递归处理儿子节点的最大快乐指数
                f[u][0] += max(f[j][0], f[j][1]); //该节点不选，则选择儿子快乐指数最大的一种选法
                f[u][1] += f[j][0]; //该节点选，儿子节点只能都不选
        }
}

int main(){

        cin >> n;

        for(int i = 1; i <= n; ++ i) cin >> happy[i];

        n -- ;
        while(n -- ){
                int a, b;
                cin >> a >> b;
                has_father[a] = 1;
                add(b, a);
        }

        int root = 1;
        while(has_father[root]) root ++ ; //找父节点

        dfs(root);

        cout << max(f[root][0], f[root][1]); //输出选或不选根节点的最大值
        return 0;
}
```

- **记忆化搜索：**
  - 每一道动态规划的题都可以用递归实现，将循环变为递归
  - **滑雪**
    - 状态表示：$f[i][j]$ 表示从(i, j)开始的所有路径
    - 状态计算：$f[i][j] = f[i \pm 1][j \pm 1] + 1$

```
const int N = 310;
int f[N][N];
int h[N][N];
int n, m;
int dx[] = {0, -1, 1, 0};
int dy[] = {1, 0, 0, -1};

int dp(int x, int y){
    int &v = f[x][y]; //引用v相当于f[x][y]
    if(v) return v; //已经求过直接返回
    v = 1; //初始化
    for(int i = 0; i < 4; ++ i){
        int xn = x + dx[i];
        int yn = y + dy[i];
        if(xn >= 1 && xn <= n && yn >= 1 && yn <= m && h[xn][yn] < h[x]
[y])
            v = max(v, dp(xn, yn) + 1); //状态转移
    }
    return v; //返回
}
int main(){

    cin >> n >> m;
    for(int i = 1; i <= n; ++ i)
        for(int j = 1; j <= m; ++ j)
            cin >> h[i][j];

    int res = 0;
    for(int i = 1; i <= n; ++ i)
        for(int j = 1; j <= m; ++ j)
            res = max(res, dp(i, j)); //多起点，高度最大的起点并不一定路径
最长
    cout << res;
    return 0;
}
```

- **区间问题**
  - **区间选点**
    - **贪心策略：将所有区间按右端点排序，维护一个区间，遍历所有区间，当当前区间左端点大于维护区间的右段点时，答案加一，否则更新维护区间的右端点为当前区间的右端点**

```
const int N = 1e5 + 10;
```

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

int n;

int main(){
    cin >> n;

    for(int i = 1; i <= n; ++ i){
        int l, r;
        cin >> l >> r;
        range[i] = {l, r};
    }

    sort(range + 1, range + 1 + n);

    int res = 0, ed = -2e9;
    for(int i = 1; i <= n; ++ i){
        if(ed < range[i].l){
            res ++ ;
            ed = range[i].r;
        }
    }

    cout << res;
    return 0;
}
```

- **最大不相交区间数量**
  - 贪心策略：将所有区间按右端点排序，维护一个区间，遍历所有区间，当当前区间左端点大于维护区间的右段点时，答案加一，否则更新维护区间的右端点为当前区间的右端点

```
const int N = 1e5 + 10;

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

```
int n;

int main(){
    cin >> n;

    for(int i = 1; i <= n; ++ i){
        int l, r;
        cin >> l >> r;
        range[i] = {l, r};
    }

    sort(range + 1, range + 1 + n);

    int res = 0, ed = -2e9;
    for(int i = 1; i <= n; ++ i){
        if(ed < range[i].l){
            res ++ ;
            ed = range[i].r;
        }
    }

    cout << res;
    return 0;
}
```

- **区间分组**
  - 贪心策略：将所有区间按左端点排序，从小到大处理每个区间，判断其能否放到某个组内，依次遍历每个组，将该区间左端点与组内右端点的最大值比较，若相交就比较下一个组，若不相交就加入组内并更新组内右端点最大值，若都相交就开新组。
  - 代码：

```
using namespace std;
const int N = 1e5 + 10;
struct Range{
    int l, r;
    bool operator< (const Range &w)const{
        return l < w.l;
    }
}range[N];
priority_queue<int, vector<int>, greater<int> > q;
int n;

int main(){
```

```
    cin >> n;
    for(int i = 1; i <= n; ++ i){
        int l, r;
        cin >> l >> r;
        range[i] = {l, r};
    }

    sort(range + 1, range + 1 + n);
    for(int i = 1; i <= n; ++ i){
        Range t = range[i];
        if(q.empty() || t.l <= q.top()) q.push(t.r);
        else{
            q.pop();
            q.push(t.r);
        }
    }
    cout << q.size();
    return 0;
}
```

- **区间覆盖**
  - 贪心策略：将所有区间按左端点排序，从小到大依次枚举每个区间，在所有能覆盖到start 的区间中选择右端点最大的区间，将start更新为右端点，然后循环，知道区间覆盖end

```
using namespace std;
const int N = 1e5 + 10;
struct Range{
        int l, r;
        bool operator< (const Range &w)const {
                return l < w.l;
        }
}range[N];
int n;

int main(){
        int st, ed;
        cin >> st >> ed;

        cin >> n;

        for(int i = 1; i <= n; ++ i){
                int l, r;
                cin >> l >> r;
                range[i] = {l, r};
```

```
        }

        sort(range + 1, range + 1 + n);

        int res = 0;
        for(int i = 1, j = 1; i <= n; ++ i){
                int r = -2e9;
                while(j <= n && range[j].l <= st){
                        r = max(r, range[j].r);
                        j ++ ;
                }

                if(r < st){
                        res = - 1;
                        break;
                }

                res ++ ;
                st = r;
                if(st >= ed) break;
                i = j - 1;
        }
        if(st >= ed) cout << res;
        else cout << -1;
        return 0;
}
```

- **Huffman树**
  - **合并果子**
    - **贪心策略：每次合共重量最小的两堆果子**

```
using namespace std;
const int N = 1e4 + 10;
priority_queue<int, vector<int>, greater<int> > q;
int n;

int main(){
    cin >> n;
    for(int i = 1; i <= n; ++ i){
        int x;
        cin >> x;
        q.push(x);
    }

    int sum = 0;
```

```
    while(q.size() != 1){
        int a = q.top(); q.pop();
        int b = q.top(); q.pop();
        sum += a + b;
        q.push(a + b);
    }
    cout << sum;
    return 0;
}
```

- **排序不等式**
  - **排队打水**
    - 贪心策略：将所有人的打水时间从小到大排序，打水时间少的先打水

```
const int N = 1e5 + 10;
typedef long long LL;
int a[N];
int n;

int main(){
    cin >> n;
    for(int i = 1; i <= n; ++ i) cin >> a[i];
    sort(a + 1, a + 1 + n);
    LL sum = 0;
    for(int i = 1; i <= n; ++ i)
        sum +=(LL)a[i] * (n - i);
    cout << sum;
    return 0;
}
```

- **绝对值不等式**
  - **货仓选址**
    - 贪心策略：仓库坐标选取到所有数的中位数

```
const int N = 1e5 + 10;
int a[N];
int n;

int main(){
    cin >> n;

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

```
    sort(a + 1, a + 1 + n);
    int ans = 0;
    for(int i = 1; i <= n; ++ i) ans += abs(a[i] - a[n / 2 + 1]);
    cout << ans;
    return 0;
}
```

- **推公式**
  - **耍杂技的牛**
    - 贪心策略：将所有牛按 $w_i + s_i$ 从小往大的顺序向下排列，最小的在最上面，最大的在最下面

```
const int N = 5e4 + 10;
typedef pair<int, int> PII;
PII cow[N];
int n;

int main(){
    cin >> n;
    for(int i = 1; i <= n; ++ i){
        int w, s;
        cin >> w >> s;
        cow[i] = {{w + s}, w};
    }

    sort(cow + 1, cow + 1 + n);

    int res = -2e9, sum = 0;
    for(int i = 1; i <= n; ++ i){
        int w = cow[i].second, s = cow[i].first - w;
        res = max(res, sum - s);
        sum += w;
    }

    cout << res;
    return 0;
}
```