STL

String

```
    substr 生成子串  substr(pos,npos)
    insert 插入子串   insert(pos,"") // insert(pos,"word",2)
    find 查找         find(str,pos)  要找的元素，查找开始的位置
    erase 删除
    clear 删除全部
    replace 替换    repalce(pos,len,"#")
    string.back()
    string.front()
    string.pop_back() 删除最后一个元素

    getline(cin,str);//读取空格，不读取回车

    isupper  //是一个函数，可以用来判断字符c是否为大写英文字母。
    isupper   //功能：如果参数是大写字母字符，函数返回非零值，否则返回零值
    islower() //如果参数是小写字母字符，函数返回非零值，否则返回零值

    sscanf(b,"%d",&c);  //如果是数字就把b转换为int存储到c里面

    sprintf(s,"%04d%02d%02d",c,d,b); s是字符串


char:
    strcat(char dest,char src) 拼接
    strstr(str1,str2) 查找
    strcpy(dest,src) 复制  strcpy(str1,str2)
    strlen 获取长度
    strcmp 比较    strcmp(str1,str2)
    strlwr 转换为小写
    strupr 转换为大写
    atoi(str)  字符串转整形
    atof(str)  字符串转浮点型
    "\0"

a * b mod p = a*b - [a*b/p]*p
__builtin_popcount()
unique//排序会把重复元素往后排，返回不重复的最后一个元素下标
erase//可以去重
to_string //把整数转换为字符串
lower_bound( begin,end,num)
C++中有函数strrev，功能是对字符串实现反转
```

vector

> `end()`返回的是最后一个元素的后一个位置的地址，不是最后一个元素的地址，**所有STL容器均是如此**

> vector为可变长数组（动态数组），定义的vector数组可以随时添加数值和删除元素。

> 注意：**在局部区域中（比如局部函数里面）开vector数组，是在堆空间里面开的。**
>
> 在局部区域开数组是在栈空间开的，而栈空间比较小，如果开了非常长的数组就会发生爆栈。
>
> 故局部区域**不可以**开大长度数组，但是可以开大长度vector。

```
#include <vector>

vector<int> a; //定义了一个名为a的一维数组,数组存储int类型数据
vector<double> b;//定义了一个名为b的一维数组，数组存储double类型数据
vector<node> c;//定义了一个名为c的一维数组，数组存储结构体类型数据，node是结构体类型
vector<int> v(n);//定义一个长度为n的数组，初始值默认为0，下标范围[0, n - 1]
vector<int> v(n, 1);//v[0]到v[n-1]所有的元素初始值均为1 //注意：指定数组长度之后（指定长度后的数组就相当于正常的数组了）

//拷贝初始化
vector<int> a(n + 1, 0);
vector<int> b(a);//两个数组中的类型必须相同,a和b都是长度为n+1，初始值都为0的数组

//二维
vector<int> v[5];//定义可变长二维数组
//注意：行不可变（只有5行），而列可变,可以在指定行添加元素
//第一维固定长度为5，第二维长度可以改变


//初始化二维均可变长数组
vector<vectot<int>> v;//定义一个行和列均可变的二维数组
vector<int> t1{1, 2, 3, 4};
vector<int> t2{2, 3, 4, 5};
v.push_back(t1);
v.push_back(t2);
v.push_back({3, 4, 5, 6}) // {3, 4, 5, 6}可以作为vector的初始化,相当于一个无名vector


//行列长度均固定 n + 1行 m + 1列初始值为0
vector<vector<int> > a(n + 1, vector<int>(m + 1, 0));
```

> vector<int> v[5]可以这样理解：长度为5的v数组，数组中存储的是vector<int>数据类型，而该类型就是数组形式，故v为二维数组。其中每个数组元素均为空，因为没有指定长度，所以第二维可变长。可以进行下述操作：
>
> ```
> v[1].push_back(2);
> v[2].push_back(3);
> ```

```
vector<int> s:
    s.front() // 返回首元素
    s.back() // 返回尾元素
    s.push_back(x) // 向表尾插?元素x
    s.size() // 返回表?
    s.empty() // 表为空时，返回真，否则返回假
    s.clear() //清空
    s.pop_back() // 删除表尾元素
    s.begin() // 返回指向?元素的随机存取迭代器
    s.end() // 返回指向尾元素的下?个位置的随机存取迭代器
    s.insert(it, val) // 向迭代器it指向的元素前插?新元素val
    s.insert(it, n, val) // 向迭代器it指向的元素前插?n个新元素val
    s.insert(it, first, last) // 将由迭代器first和last所指定的序列[first, last)插?到
迭代器it指向的元素前?
    s.erase(it) // 删除由迭代器it所指向的元素
    s.erase(first, last) // 删除由迭代器first和last所指定的序列[first, last]
    s.resize(n, v)  //改变数组大小为n,n个空间数值赋为v，如果没有默认赋值为0
    s.insert(c.begin()+2,-1) //将-1插入c[2]的位置

    sort(c.begin(), c.end());  //排序
    //如果要对指定区间进行排序，可以对sort()里面的参数进行加减改动。
    sort(a.begin() + 1, a.end());

//迭代器访问法
//迭代器访问 vector<int>::iterator it;
//相当于声明了一个迭代器类型的变量it
//通俗来说就是声明了一个指针变量

//方式一:
vector<int>::iterator it = vi.begin();
for(int i = 0; i < 5; i++)
    cout << *(it + i) << " ";
cout << "\n";

//方式二:
vector<int>::iterator it;
for(it = vi.begin(); it != vi.end();it ++)
    cout << *it << " ";
//vi.end()指向尾元素地址的下一个地址

//智能指针
vector<int> v;
v.push_back(12);
v.push_back(241);
for(auto val : v)
    cout << val << " "; // 12 241
```

stack

> 栈为数据结构的一种，是STL中实现的一个先进后出，后进先出的容器。

```
//头文件需要添加
#include<stack>

//声明
stack<int> s;
stack<string> s;
stack<node> s;//node是结构体类型
```

```
stack<int> s
    s.push(val) //元素val入栈，增加元素 O(1)
    s.pop()   //移除栈顶元素 O(1)
    s.top()   //取得栈顶元素（但不删除）O(1)
    s.empty()   //检测栈内是否为空，空为真 O(1)
    s.size()    //返回栈内元素的个数 O(1)

    //栈只能对栈顶元素进行操作，如果想要进行遍历，只能将栈中元素一个个取出来存在数组中
```

queue

> 队列是一种先进先出的数据结构， 头出尾进

```
//头文件
#include<queue>
//定义初始化
queue<int> q;


q.front()   //返回队首元素 O(1)
q.back()    //返回队尾元素 O(1)
q.push(element) //尾部添加一个元素element 进队O(1)
q.pop()  //删除第一个元素 出队 O(1)
q.size()    //返回队列中元素个数，返回值类型unsigned int O(1)
q.empty()   //判断是否为空，队列为空，返回true O(1)
```

deque

> 首尾都可插入和删除的队列为双端队列

```
//添加头文件
#include<deque>
//初始化定义
deque<int> dq;
```

```
push_back(x)/push_front(x)  //把x插入队尾后 / 队首 O(1)
back()/front()  //返回队尾 / 队首元素 O(1)
pop_back() / pop_front()  //删除队尾 / 队首元素 O(1)
erase(iterator it)    //删除双端队列中的某一个元素 erase(iterator first,iterator
last) 删除双端队列中[first,last)中的元素 empty() 判断deque是否空 O(1)
size() //返回deque的元素数量 O(1)
clear() //清空deque

//deque可以进行排序
    //从小到大
    sort(q.begin(), q.end())
    //从大到小排序
    sort(q.begin(), q.end(), greater<int>()); //deque里面的类型需要是int型
    sort(q.begin(), q.end(), greater());  //高版本C++才可以用
```

priority_queue

> 优先队列是在正常队列的基础上加了优先级，保证每次的队首元素都是优先级最大的。
>
> 可以实现每次从优先队列中取出的元素都是队列中**优先级最大**的一个。
>
> 它的底层是通过**堆**来实现的。

```
//头文件
#include<queue>
//初始化定义
priority_queue<int> q;


q.top()  //访问队首元素
q.push()    //入队
q.pop()    //堆顶（队首）元素出队
q.size()    //队列元素个数
q.empty()   //是否为空
//注意没有clear()!

//设置优先级
priority_queue<int> pq; // 默认大根堆，即每次取出的元素是队列中的最大值
priority_queue<int, vector<int>, less<int> > q2; // 大根堆，每次取出的元素是队列中的
最大值

priority_queue<int, vector<int>, greater<int> > q; // 小根堆，每次取出的元素是队列中
的最小值


//自定义排序
struct cmp1
{
    bool operator()(int x,int y)
```

```
    {
        return x > y;
    }
};
struct cmp2 {
    bool operator()(const int x,const int y)
    {
        return x < y;
    }
};
priority_queue<int, vector<int>, cmp1> q1; // 小根堆
priority_queue<int, vector<int>, cmp2> q2; //大根堆
```

> **第二个参数：** `vector< int >`是用来承载底层数据结构堆的容器，若优先队列中存放的是`double`型数据，就要填`vector< double >`**总之存的是什么类型的数据，就相应的填写对应类型。同时也要改动第三个参数里面的对应类型。**
>
> **第三个参数：** `less< int >`表示数字大的优先级大，堆顶为最大的数字 `greater< int >`表示数字小的优先级大，堆顶为最小的数字 **int代表的是数据类型，也要填优先队列中存储的数据类型**

```
//结构体内部重载运算符规则

//定义的比较结构体
//注意：cmp是个结构体
struct cmp
{ //自定义堆的排序规则
    bool operator()(const Point& a,const Point& b)
    {
        return a.x < b.x;
    }
}; //初始化定义，
priority_queue<Point, vector<Point>, cmp> q; // x大的在堆顶

//方式一
struct node {
    int x, y;
    friend bool operator < (Point a, Point b)
    {   //为两个结构体参数，结构体调用一定要写上friend
        return a.x < b.x;  //按x从小到大排，x大的在堆顶
    }
};

//方式二
struct node
{
    int x, y;
    bool operator < (const Point &a) const
    {   //直接传入一个参数，不必要写friend
        return x < a.x;  //按x升序排列，x大的在堆顶
    }
};
```

```
//优先队列定义
priority_queue<Point> q;
```

> **注意：** 优先队列自定义排序规则和`sort()`函数定义`cmp`函数很相似，但是最后返回的情况是**相反**的。即相同的符号，最后定义的排列顺序是完全相反的。 所以只需要记住`sort`的排序规则和优先队列的排序规则是相反的就可以了。

```
//存储pair类型
/*
默认先对pair的first进行降序排序，然后再对second降序排序
对first先排序，大的排在前面，如果first元素相同，再对second元素排序，保持大的在前面。

*/
#include<bits/stdc++.h>
using namespace std;
int main()
{
    priority_queue<pair<int, int> >q;
    q.push({7, 8});
    q.push({7, 9});
    q.push(make_pair(8, 7));
    while(!q.empty())
    {
        cout << q.top().first << " " << q.top().second << "\n"; q.pop();
    }
    return 0;
}

//结果
8 7
7 9
7 8
```

map

> - 映射类似于函数的对应关系，每个`x`对应一个`y`，而`map`是每个键对应一个值
> - map会按照键的顺序从小到大自动排序，键的类型必须可以比较大小

```
//头文件
#include<map>
//初始化定义
map<string,string> mp;
map<string,int> mp;
map<int,node> mp;//node是结构体类型

    mp.find(key)  //返回键为key的映射的迭代器 O(logN) 注意：用find函数来定位数据出现位置，它返回一个迭代器。当数据存在时，返回数据所在位置的迭代器，数据不存在时，返回mp.end()
    mp.erase(it) //删除迭代器对应的键和值O(1)
```

```
    mp.erase(key) //根据映射的键删除键和值 O(logN)
    mp.erase(first,last) //删除左闭右开区间迭代器对应的键和值 O(last-first)
    mp.size() //返回映射的对数O(1)
    mp.clear() //清空map中的所有元素O(N)
    mp.insert() //插入元素，插入时要构造键值对 例如 mp.insert({ 2, 30 });
    mp.empty()  //如果map为空，返回true，否则返回false mp.begin() 返回指向map第一个元素的迭代器（地址）
    mp.end()   //返回指向map尾部的迭代器（最后一个元素的下一个地址）
    mp.rbegin() //返回指向map最后一个元素的迭代器（地址）
    mp.rend()   //返回指向map第一个元素前面(上一个) 的逆向迭代器（地址）
    mp.count(key)  //查看元素是否存在，因为map中键是唯一的，所以存在返回1，不存在返回0
    mp.lower_bound()    //返回一个迭代器，指向键值>= key的第一个元素
    mp.upper_bound()   //返回一个迭代器，指向键值> key的第一个元素
```

> 注意： 查找元素是否存在时，可以使用 ①`mp.find()` ② `mp.count()` ③ `mp[key]` 但是第三种情况，如果不存在对应的`key`时，会自动创建一个键值对（产生一个额外的键值对空间） 所以为了不增加额外的空间负担，最好使用前两种方法

```
//map遍历方法
map<int, int> mp
//第一种
auto it = mp.begin();
while(it != mp.end())
{
    cout << it -> first << ' ' << it -> second << endl;
    it++;
}
for (it = mp.begin(); it != mp.end(); it++)
{
    cout << it->first << ' ' << it->second << endl;
}
//第二种
for(auto [u, v] : mp) //这种遍历方式在C17中才可以使用
{
    cout << u << ' ' << v << endl;
}
//第三种
for(auto &v : mp)
{
    cout << v.first << ' ' << v.second << endl;
}


//逆向遍历
map<int,int> mp;
mp[1] = 2;
mp[2] = 3;
mp[3] = 4;
auto it = mp.rbegin();
while(it != mp.rend())
```

```
{
    cout << it->first << " " << it->second << "\n";
    it ++;
}


//二分查找
//map的二分查找以第一个元素（即键为准），对键进行二分查找,返回值为map迭代器类型
#include<bits/stdc++.h>
using namespace std;
int main()
{
    map<int, int> m{{1, 2}, {2, 2}, {1, 2}, {8, 2}, {6, 2}};//有序
    map<int, int>::iterator it1 = m.lower_bound(2);
    cout << it1->first << "\n";//it1->first=2
    map<int, int>::iterator it2 = m.upper_bound(2);
    cout << it2->first << "\n";//it2->first=6
    return 0;
}
```

```
//添加元素
map<string,string> mp;

//方式一:
mp["学习"] = "看书";
mp["玩耍"] = "打游戏";

//方式二:
mp.insert(make_pair("vegetable","蔬菜"));

//方式三
mp.insert(pair<string,string>("fruit","水果"));

//方式四
mp.insert({"hahaha","wawawa"});
```

```
与unordered_map的比较

map: 内部用红黑树实现，具有自动排序（按键从小到大）功能。

unordered_map: 内部用哈希表实现，内部元素无序杂乱。

map:
优点：内部用红黑树实现，内部元素具有有序性，查询删除等操作复杂度为O(logN)
缺点：占用空间，红黑树里每个节点需要保存父子节点和红黑性质等信息，空间占用较大。

unordered_map:
优点：内部用哈希表实现，查找速度非常快（适用于大量的查询操作）。
```

```
缺点：建立哈希表比较耗时。

使用[]查找元素时，如果元素不存在，两种容器都是创建一个空的元素；如果存在，会正常索引对应的值。所以如果查询过多的不存在的元素值，容器内部会创建大量的空的键值对，后续查询创建删除效率会大大降低。
// 以 map 为例
map<int, int> mp;
int x = 999999999;
if(mp.count(x)) // 此处判断是否存在x这个键
    cout << mp[x] << "\n"; // 只有存在才会索引对应的值，避免不存在x时多余空元素的创建
```

set

> set容器中的元素不会重复，当插入集合中已有的元素时，并不会插入进去，而且set容器里的元素自动从小到大排序。
>
> 即：set里面的元素**不重复 且有序**
>
> 当set里在同一个键里插入不同值时，set会变成一个二维数组
>
> ```
> //遍历键相同的所有值
> for(auto v : e[1])
> {
>     cout << v << ' ';
> }
> cout << endl;
> ```

```
//头文件
#include<set>
//初始化定义
set<int> s;


 s.begin()  //返回set容器的第一个元素的地址（迭代器）O(1)
 s.end()   //返回set容器的最后一个元素的下一个地址（迭代器）O(1)
 s.rbegin()  //返回逆序迭代器，指向容器元素最后一个位置O(1)
 s.rend()   //返回逆序迭代器，指向容器第一个元素前面的位置O(1)
 s.clear()  //删除set容器中的所有的元素,返回unsigned int类型O(N)
 s.empty()  //判断set容器是否为空O(1)
 s.insert()  //插入一个元素 s.size() 返回当前set容器中的元素个数O(1)
 erase(iterator)  //删除定位器iterator指向的值 erase(first,second)  删除定位器first和second之间的值
 erase(key_value)   //删除键值key_value的值

查找
s.find(element)  //查找set中的某一元素，有则返回该元素对应的迭代器，无则返回结束迭代器
s.count(element)  //查找set中的元素出现的个数，由于set中元素唯一，此函数相当于查询element是否出现
s.lower_bound(k)    //返回大于等于k的第一个元素的迭代器O(logN)
```

```
s.upper_bound(k)    //返回大于k的第一个元素的迭代器O(logN)

```

```

//访问

//第一种
for(set<int>::iterator it = s.begin(); it != s.end(); it++)
    cout << *it << " ";
//第二种
for(auto i : s)
    cout << i << endl;


//访问最后一个元素
//第一种
cout << *s.rbegin() << endl;

//第二种
set<int>::iterator iter = s.end();
iter--;
cout << (*iter) << endl; //打印2;

//第三种
cout << *(--s.end()) << endl;

//排序
set<int> s1; // 默认从小到大排序
set<int, greater<int> > s2; // 从大到小排序


//方式三：初始化时使用匿名函数定义比较规则
set<int, function<bool(int, int)>> s([&](int i, int j)
{
    return i > j; // 从大到小
 });
for(int i = 1; i <= 10; i++)
    s.insert(i);
for(auto x : s)
    cout << x << " ";
```

Function

```

#include<bits/stdc++.h>
using namespace std;

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int, int> PII;
typedef long long ll;
```

```
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e9+7;
const int N = 20, M = 10010;
int dxm[] = {2, 1, -1, -2, -2, -1, 2, 1};
int dym[] = {-1, -2, -2, -1, 1, 2, 1, 2};
int dx[] = {1, 0, -1, 0};
int dy[] = {0, 1, 0, -1};
ll n, m;
int p[N];
bool f1, f2, f3, f4;
map<int, int> mp;
ll res = 0;
int h, w;
//int a[N][N];


int main()
{
    cin >> h >> w;
    vector<vector<int> > a(h + 1, vector<int> (w + 1, 0));
    for(int i = 1; i <= h; i++)
    {
        for(int j = 1; j <= w; j++)
        {
            cin >> a[i][j];
        }
    }
    function<void(int , int, map<int, int>) > dfs = [&](int x, int y, map<int,
int> mp){
        if(x < 1 || x > h || y < 1 || y > w || mp.count(a[x][y])) return;
        if(x == h && y == w)
        {
            res++;
            return;
        }
        dfs(x + 1, y, mp);
        dfs(x, y + 1, mp);
    };
    map<int, int> mp;
    dfs(1, 1, mp);
    cout << res << endl;
    return 0;
}
```

基础算法

排序

快速排序

> - 确定划分界限 （注意加小心）
> - 根据划分界限将数据划分为两边 小于等于分界点的为一边，大于等于分界点的为一边
> - 递归求解子问题
> - O(nlogn)

```
#include<bits/stdc++.h>
using namespace std;

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 100010;
int n;
int a[N];

void quick_sort(int a[],int l,int r)
{
    if(l >= r) return;  //递归终止条件
    int i = l - 1, j = r + 1;
    int x = a[l + r >> 1];
    while(i<j)
    {
        do i++; while(a[i]<x);
        do j--; while(a[j]>x);
        if(i<j) swap(a[i],a[j]); //当a[i]的值大于等于x,a[j]的值小于等于x时，交换
a[i],a[j]
    }
    quick_sort(a,l,j);
    quick_sort(a,j+1,r);
}

int main()
{
    cin>>n;
    for(int i=0;i<n;i++)
    {
        cin>>a[i];
    }
    quick_sort(a,0,n-1);
    for(int i=0;i<n;i++)
    {
        cout<<a[i]<<' ';
    }
    cout<<endl;
    return 0;
}
```

归并排序

> - 确定划分界限，分成子问题
> - 递归处理子问题
> - 合并子问题，记得收尾
> - **O(nlogn)**

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

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 100010;
int a[N];
int q[N];
int n;

void merge_sort(int a[],int l,int r)
{
    if(l>=r) return;
    int x = l+r>>1;
    merge_sort(a,l,x);
    merge_sort(a,x+1,r);
    int k =0;
    int i = l, j = x+1;
    while(i<=x && j<=r)
    {
        if(a[i]<=a[j]) q[k++] = a[i++];
        else q[k++] = a[j++];
    }
    while(i<=x) q[k++] = a[i++];  //收尾
    while(j<=r) q[k++] = a[j++];
    for(int i=l,j=0;i<=r;i++,j++) a[i] = q[j];
}

int main()
{
    cin>>n;
    for(int i = 0;i < n;i++)
    {
        cin>>a[i];
    }
    merge_sort(a,0,n-1);
```

```
    for(int i = 0;i < n;i++)
    {
        cout<<a[i]<<' ';
    }
    cout<<endl;
    return 0;
}
```

二分

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

// 区间[l, r]被划分成[l, mid]和[mid + 1, r]时使用:
//一般用来查询符合条件的左端点
int bsearch_1(int l, int r)
{
    while(l < r)
    {
        int mid = l + r >> 1;
        if(check(mid)) r = mid;    // check()判断mid是否满足性质
        else l = mid + 1;
    }
    return l;
}
// 区间[l, r]被划分成[l, mid - 1]和[mid, r]时使用:
//一般用来查询符合条件的右端点
int bsearch_2(int l, int r)
{
    while(l < r)
    {
        int mid = l + r + 1 >> 1;
        if(check(mid)) l = mid;
        else r = mid - 1;
    }
    return l;
}
```

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

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 100010;
```

```
int a[N];
int n,q,k;

int main()
{
    cin>>n>>q;
    for(int i=1;i<=n;i++)
    {
        cin>>a[i];
    }
    while(q--)
    {
        cin>>k;
        int  l = 1,r = n;
        while(l<r)   //找不小于x的第一个数
        {
            int mid = l+r>>1;  // 区间[l, r]被划分成[l, mid]和[mid + 1, r]时使用:
            if(a[mid] >= k) r = mid;
            else l = mid+1;
        }
        if(a[l] != k) cout<<"-1 -1"<<endl;
        else
        {
            cout<<l-1<<' ';
            int l = 1,r = n;
            while(l<r)   //找不大于x的最后一个数
            {
                int mid = l+r+1>>1;    // 区间[l, r]被划分成[l, mid - 1]和[mid, r]
时使用:
                if(a[mid] <= k) l=mid;
                else r = mid-1;
            }
            cout<<r-1<<endl;
        }
    }
    return 0;
}
```

高精度

高精度加法

> - 字符串输入，然后拿变长数组从最后一位开始存储字符串的每一位
> - 用 t 表示借位，注意最后 t 的数字是否为 0 ，如果不是 ，把它存储到C的最后
> - 倒着输出，因为个位存在数组的首位

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

```
#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 100010;
vector<int> A;
vector<int> B;
vector<int> C;
void add(vector<int> A, vector<int> B)
{
    int t = 0;   //存进位，初始进位为0
    for(int i=0;i<A.size() || i<B.size();i++)
    {
        if(i<A.size()) t += A[i];
        if(i<B.size()) t += B[i];
        C.push_back(t%10);  //个位数放进C容器中，类似于加法的个位相加
        t = t/10;           //其余的皆为进位数进行下一次运算
    }
    if(t) C.push_back(t); //若t最后不为0,就把它加到数组的末尾

}
int main()
{
    string a,b;
    cin>>a>>b;
    for(int i=a.size()-1;i>=0;i--) A.push_back(a[i]-'0');
    for(int i = b.size()-1;i>=0;i--) B.push_back(b[i]-'0');
    add(A,B);
    for(int i=C.size()-1;i>=0;i--) cout<<C[i];
    cout<<endl;
    return 0;
}
```

```
#include <iostream>        //高精度加法压九位操作
#include <vector>

using namespace std;

const int base = 1000000000;

vector<int> add(vector<int> &A, vector<int> &B)
{
    if (A.size() < B.size()) return add(B, A);

    vector<int> C;
    int t = 0;
    for (int i = 0; i < A.size(); i ++ )
    {
```

```
        t += A[i];
        if (i < B.size()) t += B[i];
        C.push_back(t % base);
        t /= base;
    }

    if (t) C.push_back(t);
    return C;
}

int main()
{
    string a, b;
    vector<int> A, B;
    cin >> a >> b;

    for (int i = a.size() - 1, s = 0, j = 0, t = 1; i >= 0; i -- )
    {
        s += (a[i] - '0') * t;
        j ++, t *= 10;
        if (j == 9 || i == 0)
        {
            A.push_back(s);
            s = j = 0;
            t = 1;
        }
    }
    for (int i = b.size() - 1, s = 0, j = 0, t = 1; i >= 0; i -- )
    {
        s += (b[i] - '0') * t;
        j ++, t *= 10;
        if (j == 9 || i == 0)
        {
            B.push_back(s);
            s = j = 0;
            t = 1;
        }
    }

    auto C = add(A, B);

    cout << C.back();
    for (int i = C.size() - 2; i >= 0; i -- ) printf("%09d", C[i]);
    cout << endl;

    return 0;
}
```

高精度减法

> - 同高精度加法操作

> - 需要注意判断 A 和 B 大小 若 A < B , 需要加"-"号,具体看代码注释
> - 需要去除前导 0 具体看代码注释
> - 注意这里的 t 是本位的借位操作，所以是被减数减去t(具体看代码注释)

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

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 100010;
vector<int> A;
vector<int> B;
vector<int> C;

bool cmp(vector<int> A, vector<int> B)   //判断 A,B的大小
{
    if(A.size() != B.size()) return A.size() > B.size();
    for(int i = A.size()-1;i >= 0;i--)
    {
        if(A[i] != B[i]) return A[i] > B[i];
    }
    return true;
}

void mul(vector<int> A, vector<int> B)
{
    int t = 0; //存储上一位的借位
    for(int i=0;i<A.size() || i<B.size();i++)
    {
        if(i<A.size()) t = A[i]-t;
        if(i<B.size()) t -= B[i];
        C.push_back((t+10)%10);
        if(t<0) t = 1; // 若 t<0 说明本位向上一位借去了一个 1 ,所以再进行本位计算的时
候，被减数需要减去1
        else t = 0;
    }
    while(C.size()>1 && C.back()==0) C.pop_back();   //去除前导 0
}

int main()
{
    string a, b;
    cin>>a>>b;
    for(int i=a.size()-1;i>=0;i--) A.push_back(a[i] - '0');
    for(int i=b.size()-1;i>=0;i--) B.push_back(b[i] - '0');
    if(!cmp(A,B))
```

```
    {
        cout<<"-";   // 若 A < B ,输出的时候需要加个 - 号
        mul(B,A);
    }
    else mul(A,B);
    for(int i=C.size()-1;i>=0;i--) cout<<C[i];
    cout<<endl;
    return 0;
}
```

高精度乘法

> - 字符串读入，数组从后往前存
> - 存储最后结果的数组可以开大一点
> - 模拟乘法运算
> - 处理进位
> - 去除前导0

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

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 100010;
vector<int> A;
vector<int> B;


vector<int> mul(vector<int> &A, vector<int> &B)
{
    vector<int> C(A.size()+B.size()+7,0);
    for(int i=0;i<A.size();i++)
    {
        for(int j=0;j<B.size();j++)
        {
            C[i+j] += A[i]*B[j];
        }
    }
    int t = 0;
    for(int i=0;i<C.size();i++)
    {
        t += C[i];
        C[i] = t%10;
        t /= 10;
    }
```

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

int main()
{
    string a, b;
    cin>>a>>b;
    for(int i=a.size()-1;i>=0;i--) A.push_back(a[i] - '0');
    for(int i=b.size()-1;i>=0;i--) B.push_back(b[i] - '0');
    vector<int> C = mul(A,B);
    for(int i=C.size()-1;i>=0;i--) cout<<C[i];
    cout<<endl;
    return 0;
}
```

```
//高精度x高精度

#include <iostream>
#include <vector>

using namespace std;

vector<int> mul(vector<int> &A, vector<int> &B) {
    vector<int> C(A.size() + B.size() + 7, 0); // 初始化为 0，C的size可以大一点

    for (int i = 0; i < A.size(); i++)
        for (int j = 0; j < B.size(); j++)
            C[i + j] += A[i] * B[j];

    int t = 0;
    for (int i = 0; i < C.size(); i++) { // i = C.size() - 1时 t 一定小于 10
        t += C[i];
        C[i] = t % 10;
        t /= 10;
    }

    while (C.size() > 1 && C.back() == 0) C.pop_back(); // 去前导 0，因为最高位很可
能是 0
    return C;
}

int main() {
    string a, b;
    cin >> a >> b; // a = "1222323", b = "2323423423"

    vector<int> A, B;
    for (int i = a.size() - 1; i >= 0; i--)
        A.push_back(a[i] - '0');
    for (int i = b.size() - 1; i >= 0; i--)
        B.push_back(b[i] - '0');
```

```
    auto C = mul(A, B);

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

return 0;
}
```

高精度除法

> - 字符串读入，倒序存入数组
> - 由于除法运算是从高位开始，所以函数中也从高位开始
> - t 表示前一位的余数，进行本位运算时需要 * 10；
> - 将 t/b 的商存入数组，余数进行下一位的运算
> - 由于新数组(存结果)里是从高位开始存储的，为了方便去除前导0，可将新数组翻转
> - 倒序输出新数组，最后的 t 便是余数

```
#include<bits/stdc++.h>
using namespace std;

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);

typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 2e5+10;
const int N = 10010;
vector<int> A;
vector<int> C;

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

int main()
{
    string a;
    int b;
    int t = 0;
```

```
    cin>>a>>b;
    for(int i=a.size()-1;i>=0;i--) A.push_back(a[i]-'0');
    div(A,b,t);
    for(int i=C.size()-1;i>=0;i--) cout<<C[i];
    cout<<endl<<t<<endl;
    return 0;
}
```

前缀和与差分

一维前缀和

> - 前缀和的预处理 s[i] = s[i-1] + a[i]; (前缀和运算数组下标最好从1开始)
> - 求某一个区间的和 s[r] - s[l-1]

```
#include<bits/stdc++.h>
using namespace std;

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);

typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 2e5+10;
const int N = 100010;
int n,m;
int a[N];
int s[N];
int main()
{
    cin>>n>>m;
    int l, r;
    for(int i=1;i<=n;i++)
    {
        cin>>a[i];
    }
    for(int i=1;i<=n;i++) s[i] = s[i-1] + a[i];
    while(m--)
    {
        cin >> l >> r;
        cout<<s[r]-s[l-1]<<endl;
    }
    return 0;
}
```

二维前缀和

> - 二维前缀和预处理：$S[i, j] = S[i, j-1] + S[i-1, j] - S[i-1, j-1] + a[i, j]$
> - 求某个二维区间的和：$S[x2, y2] - S[x1-1, y2] - S[x2, y1-1] + S[x1-1, y1-1]$

```
#include<bits/stdc++.h>
using namespace std;

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);

typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 2e5+10;
const int N = 2000;
int a[N][N];
int s[N][N];
int n, m, q;

int main()
{
    cin>>n>>m>>q;
    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=m;j++)
        {
            cin>>a[i][j];
        }
    }
    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=m;j++)
        {
            s[i][j] = s[i-1][j] + s[i][j-1] - s[i-1][j-1] + a[i][j];
        }
    }
    while(q--)
    {
        int x1, y1, x2, y2;
        cin>>x1>>y1>>x2>>y2;
        cout<<s[x2][y2]-s[x1-1][y2]-s[x2][y1-1]+s[x1-1][y1-1]<<endl;
    }
    return 0;
}
```

一维差分

> - 构造差分数组b：使得 b[1] = a[1], b[2] = a[2]-a[1],如此构造可让 a 数组成为 b 数组的前缀和数组

```
//a是b的前缀和数组，则b是a的差分数组。对b数组的b[i]的修改，会影响到a数组中从a[i]及往后的每一个数。
#include<iostream>
#include<vector>
using namespace std;
```

```
#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 100010;

int n,m;
int a[N];
int b[N];
int s[N];
void insert(int l,int r,int c)
{
    b[l] += c;
    b[r+1] -= c;
}
int main()
{
    cin >> n >> m;
    for(int i = 1; i <= n; i++) cin >> a[i];
    for(int i = 1; i <= n; i++) // 经过这次循环插入操作，可以让的 a 数组成为是 b 数组
的前缀和数组的，将下面代码的注释取消即可查验
    {
        insert(i,i,a[i]);
    }
//  cout<<endl;
//  cout<<"b"<<' ';
//  for(int i=1;i<=n;i++) cout<<b[i]<<' ';
//  cout<<endl;
//  cout<<endl;
    while(m--)
    {
        int l, r, c;
        cin>>l>>r>>c;
        insert(l,r,c);
    }
//  cout<<endl;
//  cout<<"a"<<' ';
//  for(int i = 1;i<=n;i++) cout<<a[i]<<' ';
//  cout<<endl;
//  cout<<"b"<<' ';
//  for(int i=1;i<=n;i++) cout<<b[i]<<' ';
//  cout<<endl;
    for(int i=1;i<=n;i++) s[i] = s[i-1] + b[i]; //为了方便理解，这里再开一个数组来存
储 b 数组的前缀和，当然也可以在 b数组上直接操作
//  cout<<"b"<<' ';
    for(int i=1;i<=n;i++) cout<<s[i]<<' ';
    cout<<endl;
    return 0;
}
```

二维差分

> - 二维差分插入操作：`b[x1][y1] += c`;
> - `b[x2+1][y1] -= c`;
> - `b[x1][y2+1] -= c`;
> - `b[x2+1][y2+1] += c`;
> - 二维差分数组求其前缀和：`a[i][j] = a[i-1][j] + a[i][j-1] - a[i-1][j-1] + b[i][j]`;
> - 其他同一维差分

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

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 2000;

int n, m, q;

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

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

int main()
{
    cin>>n>>m>>q;
    for(int i = 1;i <= n;i++)
    {
        for(int j = 1;j <= m;j++)
        {
            cin>>a[i][j];
        }
    }
    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=m;j++)
        {
            insert(i,j,i,j,a[i][j]);
        }
```

```
    }
    while(q--)
    {
        int x1, y1, x2, y2, c;
        cin>>x1>>y1>>x2>>y2>>c;
        insert(x1,y1,x2,y2,c);
    }
    for(int i=1; i <= n; i++)
    {
        for(int j=1; j <= m; j++)
        {
            a[i][j] = a[i-1][j] + a[i][j-1] - a[i-1][j-1] + b[i][j];
        }
    }
    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=m;j++)
        {
            cout<<a[i][j]<<' ';
        }
        cout<<endl;
    }
    return 0;

}
```

双指针

AcWing 799. 最长连续不重复子序列

> - 创建一个新数组用来记录每个数值的个数 即 s[a[i]]
> - 遍历数组 a 中的每一个元素a[i],对于 i ,找到 j 使得[j,i]维护的是一段以a[i]结尾的最长连续不重复子区间
> - 对于每个a[i] 结尾的最长连续不重复子区间，找到其中的最大值，记录下长度，即 i-j+1

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

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 1e5+10;

int n;
int a[N];
```

```
int s[N];
int main()
{
    cin>>n;
    for(int i=1;i<=n;i++)
    {
        cin>>a[i];
    }
    int res = -1;
    for(int i = 1, j = 1;i <= n;i++)
    {
        s[a[i]]++;
        while(s[a[i]]>1 && j<i)
        {
            s[a[j]]--;
            j++;
        }
        res = max(res,i-j+1);
    }
    cout<<res<<endl;
    return 0;
}
```

AcWing 800. 数组元素的目标和

> - 循环遍历a[i]，j 从 m 的最后一个元素开始，当a[i] + a[j] < x 时，此时 i 已经是该数组当前情况下的最小值所以只需要让 j 所代表的数组元素值从后往前遍历(数组已保证升序)，直到找到符合条件元素
> - 若没有符合条件的元素，i 就向后走一位，再进行 "1" 操作
> - 注意这里的 j 所指的数组元素再进行第二次循环时已无需从最后一位开始，因为 a[i] 增大。假设 a[j]是m数组里的最后一为元素 则 a[i-1]+a[j] 一定小于 a[i]+a[j]所以j只需要从上一次循环操作后的位置开始就可

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

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 1e5+10;

int a[N],b[N];
int main()
{
    int n,m,x;
```

```
    cin>>n>>m>>x;
    for(int i = 0; i < n; i++) cin >> a[i];
    for(int j = 0; j < m ;j++) cin >> b[j];
    for(int i = 0,j=m-1;i<n;i++)
    {
        while(a[i]+b[j] > x && j >= 0) j--;
        if(a[i] + b[j] == x)
        {
            cout<<i<<' '<<j<<endl;

        }
    }
    return 0;
}
```

位运算

> - 返回一个数的二进制表示中最后一位 1 的操作 x & -x 相当于 lowbit(x)

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

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 1e5+10;

int n;

int main()
{
    cin >> n;
    int x;
    while(n--)
    {
        int res = 0;
        cin >> x;
        for(int i = x; i != 0; i -= i & -i)
        {
            res++;
        }
        cout << res << ' ';
    }
    cout << endl;
    return 0;
}
```

离散化

AcWing 802. 区间和

> - 离散化的本质，就是映射。把无限空间内的间隔很大的有限个点映射到有限空间中去。
> - 明确题意 ， 读输入，把所有和下标有关的点用一个数组存储，这里用了 vector alls;
> - 把和插入有关的数据存储到一个 pair 数组里，这里是 add，把和询问有关的数据也用一个 pair 类型的数组存储起来，这里是 query
> - 对 alls 数组排序，去重，这时候 alls 数组里存储的是一行有序的下标，之后的 插入 询问 操作都基于这个 alls 数组，
> - 创建 a[N]数组用来存对应位置插入的值,s[N]数组用来存数组 a 前缀和
> - 遍历存储 插入 数据的数组,对每一个需要插入的位置下标，通过二分查找在 alls 数组里找到对应的下标所在的位置 然后在 a 数组里相同位置插入数据
> - 前缀和预处理
> - 处理询问

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

using namespace std;

typedef pair<int, int> PII;
const int N = 300010;

int n, m;
int a[N], s[N];
vector<int> alls;    //储存所有待离散化的值(这里是下标)
vector<PII> add, query;

int find(int x)    //二分求出x对应离散化的值
{
    int l = 0, r = alls.size() - 1;
    while (l < r)
    {
        int mid = l + r >> 1;
        if (alls[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return r + 1;    //映射到1,2, ...n
}

int main()
{
    cin >> n >> m;
    for (int i = 0; i < n; i++)
    {
        int x, c;
        cin >> x >> c;
        add.push_back({ x,c });
        alls.push_back(x);     //下标放进去
```

```
    }
    for (int i = 0; i < m; i++)
    {
        int l, r;
        cin >> l >> r;
        query.push_back({ l,r });
        alls.push_back(l);
        alls.push_back(r);
    }
    //去重
    sort(alls.begin(), alls.end());
    alls.erase(unique(alls.begin(), alls.end()), alls.end());

    //for(int i=0;i<=alls.size()-1;i++)
    //{
        //cout<<alls[i]<<' ';
    //}
    //puts("");

    //处理插入
    for (auto item : add)
    {
        int x = find(item.first);
        a[x] += item.second;
    }
    //预处理前缀和
    for (int i = 1; i <= alls.size(); i++) s[i] = s[i - 1] + a[i];

    //处理询问
    for (auto item : query)
    {
        int l = find(item.first), r = find(item.second);
        cout << s[r] - s[l - 1] << endl;
    }

    return 0;
}
```

AcWing 803. 区间合并

> - 将给定的区间存到 pair 类型的数组里，再按区间左端点对数组进行排序（一定不要忘了排序）
> - 定义 st, ed 为当前维护的一段区间，赋初始值 -2e9
> - 遍历数组，判断 当前维护的区间的右端点 与 当前遍历的区间左端点 是否有交集, 如果没有，该段维护的区间即是一段符合要求的区间，存入新数组,如果有，比较 维护的区间的右端点 与 当前遍历的区间的右端点的大小，将较大的赋给ed(即 维护区间的右端点)
> - 将最后一段区间存入数组

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

```
using namespace std;

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 1e5+10;

int n;
vector<PII> a;

void merge(vector<PII> &a)
{
    vector<PII> res;
    int st = -2e9;
    int ed = -2e9;
    sort(a.begin(), a.end());
    for(auto it : a)
    {
        if(ed < it.first)
        {
            if(st != -2e9) res.push_back({st, ed});
            st = it.first;
            ed = it.second;
        }
        else ed = max(ed,it.second);
    }
    if(st != -2e9) res.push_back({st, ed});

    a = res;

//  for(auto it : res)
//  {
//      int l = it.first;
//      int r = it.second;
//      cout << l << ' ' << r << endl;
//  }
}

int main()
{
    IOS;
    cin >> n;
    for(int i = 1;i <= n; i++)
    {
        int l, r;
        cin >> l >> r;
        a.push_back({l, r});
    }
    merge(a);
    cout << a.size() << endl;
```

```
    return 0;

}
```

数据结构

链表

单链表

> - 注意初始化
> - 头结点 head 里存放的指向下一个数据的下标
> - e[N] 存放数据，ne[N] 存放下一个数据的位置下标

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

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 1e5+10;

int head, idx, e[N], ne[N];

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

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

void add(int k, int x)
{
    e[idx] = x;
    ne[idx] = ne[k];
    ne[k] = idx;
```

```
    idx++;
}

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

int main()
{
    int n;
    cin >> n;
    init();
    while(n--)
    {
        char op;
        int k, x;
        cin >> op;
        if(op == 'H')
        {
            cin >> x;
            add_to_head(x);
        }
        else if(op == 'I')
        {
            cin >> k >> x;
            add(k-1, x);
        }
        else{
            cin >> k;
            if(!k) head = ne[head];
            else remove(k-1);
        }

    }
    for(int i = head; i != -1; i = ne[i])
    {
        cout << e[i] << ' ';
    }
    cout << endl;
    return 0;
}
```

模拟栈

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

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
```

```
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 1e5+10;

int stk[N],tt;

int main()
{
    IOS;
    string op;
    int n;
    cin >> n;
    while(n--)
    {
        cin >> op;
        if(op == "push")
        {
            int x;
            cin >> x;
            stk[++tt] = x;
        }
        else if(op == "pop") tt--;
        else if(op == "empty") cout << (tt ? "NO" : "YES") <<endl;
        else cout << stk[tt] << endl;
    }
    return 0;
}
```

单调栈

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

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 1e5+10;

int stk[N],tt;

int main()
```

```
{
    int n;
    cin >> n;
    int x;
    for(int i = 0; i < n; i++)
    {
        cin >> x;
        while(tt && x <= stk[tt]) tt--; //当栈不空 并且栈顶元素大于等于 x时；弹出栈顶
元素
        if(tt) cout << stk[tt] << ' '; //如果栈不空，栈顶元素即为所求
        else cout << "-1" <<' ';  //栈空输出 -1
        stk[++tt] = x; //最后将 x 放进栈
    }
    cout << endl;
    return 0;
}
```

队列

```
#include<iostream>

using namespace std;

const int N = 100010;

int m;
int q[N],hh,tt=-1;  //hh表示队头，tt表示队尾 ，队头弹出元素，队尾插入元素

int main()
{
    cin>>m;
    while(m--)
    {
        string op;
        int x;

        cin>>op;
        if(op=="push")
        {
            cin>>x;
            q[++tt] = x;
        }
        else if(op=="pop") hh++;      //弹出队头元素
        else if(op=="empty") cout<<(hh<=tt ? "NO" : "YES")<<endl;   //判断队列是否
为空
        else cout<<q[hh]<<endl;       //取出队头元素
    }
    return 0;
}
```

单调队列

> - 处理队首已经滑出窗口的问题
> - 处理队尾元素和下一个元素的关系(即是否满足单调性)
> - 将当前元素加入队尾
> - 满足条件则输出结果
> - 找最大值和最小值分开来

```
#include<iostream>

using namespace std;

const int N = 1000010;

int a[N],q[N];

int main()
{
    int n,k;
    scanf("%d%d",&n,&k);
    for(int i=0;i<n;i++) scanf("%d",&a[i]);
    int hh=0,tt=-1;
    for(int i=0;i<n;i++)
    {
        //判断队头是否已经滑出窗口
        if(hh<=tt&&i-k+1>q[hh]) hh++; //队列起点是i-k+1，终点是i

        //若新插入的数a[i]小于队尾元素，则队尾出队，保证单调性
        while(hh<=tt&&a[q[tt]]>=a[i]) tt--;

        q[++tt]=i;
        if(i>=k-1) printf("%d ",a[q[hh]]);  //不足k个数就不用输出了
    }
    puts("");
    hh=0,tt=-1;
    for(int i=0;i<n;i++)
    {
        if(hh<=tt&&i-k+1>q[hh]) hh++;
        while(hh<=tt&&a[q[tt]]<=a[i]) tt--;
        q[++tt]=i;
        if(i>=k-1) printf("%d ",a[q[hh]]);
    }
    puts("");
    return 0;
 }
```

字典树

> - Trie数是高效存储和查找字符串集合的一种数据结构
> - 其思路是用一颗多叉树、字典序、最长前缀 的形式来存储每个字符串，我们假设起始节点为树根，用p = 0代替,把字符串中的每个字符转换成数字，由于这里全是小写字母，可用 0~25 代替，即每个节点最多有 26 个分支, 所以可以创建一个二维数组 `c[p][x]` 来存储该节点，p 相当于该节点的父亲节点，x即为字符转换为数字的值,这样就可以用来精确表示该字符的位置，每当来一个新的节点时，就 ++idx 来开辟空间存储该节点

```
#include<bits/stdc++.h>
using namespace std;

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 2e5 + 10;   //尽量开大一点，因为这里其实是用空间换时间，空间会占用很多

int cnt[N];  //记录以该字母结尾个的字符串的个数
int c[N][26]; // N 可以相当于该节点的父亲节点，因为都是小写字母，所以每个节点最多有26个分支
int idx;  //相当于开辟一个新空间来记录没有出现过的节点，即每闯创建一个节点值就 + 1
int p;

void insert(string s)
{
    p = 0;  //类似于一个指向作用的指针
    for(int i = 0; i < s.size(); i++)
    {
        int x = s[i] - 'a';  // 将字母转换为数字
        if(!c[p][x]) c[p][x] = ++idx;   //
//      cout << idx << ' ';
        p = c[p][x];  //类似于让指针指向该节点位置
    }
    //cout << endl;
    cnt[p]++;  //相当于对该位置做标记，代表以该位置结束的字符串的个数
}

int query(string s)
{
    p = 0;
    for(int i = 0; i < s.size(); i++)
    {
        int x = s[i] - 'a';
        if(!c[p][x]) return 0; //如果查找过程中有没找到的节点，就说明没有找到这个单词，直接返回
        p = c[p][x];
    }
    return cnt[p];
}
```

```
int main()
{
    IOS;
    int n;
    cin >> n;
    while(n--)
    {
        char op;
        string s;
        cin >> op >> s;
        if(op == 'I') insert(s);
        else cout << query(s) << endl;
    }
    return 0;
}
```

并查集

> - 并查集可以将两个集合合并，也可以查找两个元素是否在同一个集合中 时间复杂度均为 o(1)
> - 原理：每个集合都可以用一棵树来表示，树根就是整个集合的父亲节点，由于每次查询都要从该节点依次向上找到父亲节点，直到找到根节点，我们可以对这个操作进行路径压缩，即让每个节点在查询父亲节点时都和根节点(因为根节点是最终父亲节点)直接相连。
> - 在判断两个节点是否在一个集合中，我们只需查找他们的父亲节点是否相同即可

```
#include<bits/stdc++.h>
using namespace std;

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 1e5 + 10;

int p[N];

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

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

```
    while(m--)
    {
        char op;
        int a, b;
        cin >> op >> a >> b;
        if(op == 'M')
        {
            p[find(a)] = find(b); //这里让 b 的父亲节点设为 a 的父亲节点
        }
        else
        {
            if(find(a) == find(b)) cout << "Yes" << endl;
            else cout << "No" << endl;
        }
    }
    return 0;
}
```

哈希表

开放寻址法

> - 哈希表是把比较庞大的数据映射到 0 ~ N 范围内
> - 创建一个哈希数组，用来存储映射后的值
> - 开放寻址法：先将数组中的每一个位置都赋予初始值，这个初始值要比所有可能需要哈希的数据都要大
>
>   然后find函数有两个功能，一个是将需要哈希的数据映射到哈希表中，一个是查找哈希表中是否有改数据，具体看代码注释
> - 切记给数组赋初始值

```
#include<bits/stdc++.h>
using namespace std;

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int  INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 200003;  // 开放寻址法 一般需要开一个两倍以上素数大小的数组，这样可以大大减小哈希冲突

int h[N];

int find(int x)
```

```
{
    int k = (x % N + N) % N;   // 把 x 映射到 0 ~ N 范围内，后面的 + N ) % N 是为了
防止前面是负数的情形
    while(h[k] != INF && h[k] != x) // 当 该位置的数是初始化的数没有改变 或 该位置的数
是 x 时
    {
        k++;
        if(k == N) k = 0;
    }
    return k;
    // 这里 k 有两成含义 如果 x 在哈希表中，k 就是下标（对应查找
    //                   如果 x 不在哈希表中，k 就是 x 应该存储的位置 （对应插入
}

int main()
{
    IOS;
//  for(int i=200000; ;i++)    寻找合适的 N 的大小
//  {
//      bool flag = true;
//      for(int j=2;j*j<=i;j++)
//      {
//          if(i%j==0)
//          {
//              flag=false;
//              break;
//          }
//
//      }
//      if(flag)
//      {
//          cout<<i<<endl;
//          break;
//      }
//  }
    int n;
    cin >> n;
    memset(h, 0x3f, sizeof h);
    while(n--)
    {
        char op;
        int x;
        cin >> op >> x;
        int k = find(x);
        if(op == 'I') h[k] = x;
        else
        {
            if(h[k] != INF) cout << "Yes" << endl;
            else cout << "No" << endl;
        }
    }
    return 0;
}
```

拉链法

> - 哈希数组里的每个位置都可以单链表头，初始时，链表只有表头，且赋初始值为 -1，这里采用头插法建立链表
> - 2：注意链表里存储的是 x
> - 3：一定要注意初始化数组

```
#include<bits/stdc++.h>
using namespace std;

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 100003;

int e[N],ne[N],idx;
int h[N];

void insert(int x)
{
    int k = (x % N + N) % N;
    e[idx] = x;         // 头插法
    ne[idx] = h[k];
    h[k] = idx++;
}

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

int main()
{
    IOS;
    int n;
    cin >> n;
    memset(h, -1, sizeof h);  //每一个槽相当于一个单链表头，赋初始值-1
    while(n--)
    {
        char op;
        int x;
```

```
        cin >> op >> x;
        if(op == 'I') insert(x);
        else{
            if(find(x)) cout << "Yes" << endl;
            else cout << "No" << endl;
        }
    }
    return 0;
}
```

图论

DFS

排列数字

> - 设置一个 bool 类型的数组来判断当前的数有没有被选择，设置一个数组来存储被选择的数（即答案数组）
> - 对 n 以内的数进行循环判断，如果它没有被选择过，就把它存入答案数组，接着把它的 bool 类型设为true
>
>   代表这个数被选择了，接下来进行下一个位置的判断(注意这里就有深度优先搜索的思想了，它是先看
>
>   下一个位置该选什么数，而不是看这个位置还可以选哪些数，可以把它画成一棵树，就像从数根先一条路走到
>
>   最下面一个树叶节点 对应代码中的操作是 dfs(u + 1))
> - 回溯：第 i 个位置填写某个数字的所有情况都遍历后， 第 i 个位置填写下一个数字.
> - 这里提供两种代码，思路基本一样。只是存储和判断换了种方式

```
#include<bits/stdc++.h>
using namespace std;

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 1e5+10;

int n;
bool st[N];
int path[N];
```

```
void dfs(int u)
{
    if(u == n)
    {
        for(int i = 0; i < n; i++)
        {
            cout << path[i] << ' ';
        }
        cout << endl;
        return;
    }

    for(int i = 1; i <= n; i++)
    {
        if(!st[i])
        {
            st[i] = true;
            path[u] = i;
            dfs(u + 1);
            st[i] = false;
        }
    }
}


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

================================================================================
#include<bits/stdc++.h>
using namespace std;

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 1e5+10;

int n;

int state;
vector<int> path;
void dfs(int u, int state)
{
    if(u == n)
    {
```

```
        for(auto it : path)
        {
            cout << it << ' ';
        }
        cout << endl;
        return;
    }
    for(int i = 1; i <= n; i++)
    {
        if(!(state >> i & 1))
        {
            path.push_back(i);
            dfs(u + 1, state | 1 << i); //state | 1 << i 是将 state 的二进制表示中
第 i 位设置成1
            path.pop_back();
        }
    }
}


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

AcWing 843. n-皇后问题

> - 对于每一行，我们需要判断它的每一列，每一个点的对角线和反对角线是否已经存在皇后了，若都没有，该点符合条件
> - 对角线和反对角线我们可以写出它的一元函数，该函数的截距可以用来表示对角线，因为不同截距可以用来表示不同对角线
> - 至于正对角线可能为负值，所以在后面 + n

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

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int  INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 20;
typedef pair<int, int> PII;
```

```
char g[N][N]; //存储数据
bool col[N];  //列
bool dg[N];   // 对角线
bool udg[N];  // 反对角线

int n;

// 按行搜索 x 是行，y 是列
void dfs(int x)
{
    if(x == n)  // 当开始搜第 n 行时，表示 0 ~ n-1 行已经搜过了，即已经搜了 n 行了，可以输出答案
    {
        for(int i = 0; i < n; i++)
        {
            for(int j = 0; j < n; j++)
            {
                cout << g[i][j];
            }
            puts("");  //换行
        }
        cout << endl;
        return;
    }

    for(int y = 0; y < n; y++)
    {
        if(!col[y] && !dg[y - x + n] && !udg[y + x])  // 判断该行所在的列，对角线，反对角线是否已经有皇后了，若没有，该点就是符合要求的点
        {
            g[x][y] = 'Q';
            col[y] = dg[y - x + n] = udg[y + x] = true;
            dfs(x + 1);
            col[y] = dg[y - x + n] = udg[y + x] = false;
            g[x][y] = '.';
        }
    }
}

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

BFS

迷宫板子

> - 只有当边权是 1 时，才可以用 BFS 求最短路
> - BFS 求最短路的思路是从起始点开始，每次向四个方向扩展距当前点距离为 1 的点是否可走，如果可以走，把该点到起点的距离记录下来，也是对该点做一个标记，表示接下来不会在走过该点了，以上步骤用一个二维数组即可记录下数据，这里是 `d[N][N]`,然后将该点放入队列，接着再以该点向外扩展，寻找符合条件的点

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

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int  INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 110;
typedef pair<int, int> PII;
int n, m;
int g[N][N];  // 存图的信息
int d[N][N];  //存储每个点到起点的距离，走过点就不会再走边了
PII q[N * N];   //数组模拟队列

int bfs()
{
    int hh = 0, tt = 0;
    q[0] = {0,0}; //将起点放入队列
    memset(d , -1, sizeof d); //赋初始值，代表该点还没有走过
    d[0][0] = 0; //起点到自身的距离为0
    int dx[4] = {-1, 0 ,1, 0};
    int dy[4] = {0, 1, 0, -1};
    while(hh <= tt)
    {
        auto t = q[hh++];
        for(int i = 0; i < 4; i++)
        {
            int x = t.first + dx[i];
            int y = t.second + dy[i];
            if(x >= 0 && x < n && y >= 0 && y < m && d[x][y] == -1 && g[x][y] == 0)
            {
                d[x][y] = d[t.first][t.second] + 1;  //记录下该点到起点的距离，由于边权是 1 ，所以距离就是上一个点到起点的距离 + 1 的距离
```

```
                q[++tt] = {x, y};  //存入队尾
            }
        }
    }
    return d[n-1][m-1];
    //bfs函数里面的if语句确保了最短距离，图里面只有没走过的点才会向下计算距离，
    //当最短距离出来之后，出口的点就相当于走过了，所以不会有更长的出现了
    // 所以只需输出最后一个点到起点的距离即可

}

int main()
{
    cin >> n >> m;
    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < m; j++)
        {
            cin >> g[i][j];
        }
    }
    cout << bfs() << endl;
    return 0;
}
```

AcWing 847. 图中点的层次

> - 求 1~n 最短距离，这里说了两存在边关系的点之间的边权为 1，所以可以用 BFS 求最短路径
> - 广度优先搜索的核心就是队列，这里数组模拟队列来做，也可以用 STL 中的 queue
> - 数据采用邻接表的方式存储，对 h 赋初始值 -1， 用 d 数组来记录某个点到起点的距离，赋初始值 -1

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

using namespace std;

const int N =1e5+10;

int h[N],e[N],ne[N],idx;
int n,m;
int d[N]; //存储每个节点离起点的距离  d[1]=0
int q[N]; //存储层次遍历序列

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

```
}
int bfs()
{
    int hh=0,tt=0; //初始化队列
    q[0]=1;     //编号为1的节点放在q中
    memset(d,-1,sizeof d);
    d[1]=0;
    while(hh<=tt)  //队列不空
    {
        int t=q[hh++]; //取出队头
        for(int i=h[t];i!=-1;i=ne[i]) //遍历t节点的每一个邻边
        {
            int j=e[i];
            if(d[j]==-1)  //如果j没有被扩展过
            {
                d[j]=d[t]+1;//d[j]存储j节点离起点的距离，并标记为访问过
                q[++tt]=j;   //把j节点压入队列
            }
        }
    }
    return d[n];
}

int main()
{
    cin>>n>>m;
    memset(h,-1,sizeof h);
    for(int i=0;i<m;i++)
    {
        int a,b;
        cin>>a>>b;
        add(a,b);
    }
    printf("%d",bfs());
}
```

拓扑序

> - 拓扑序是指在一张图中，有一条只能从前往后，并且经过所有的点的路径，由此我们可以知道，拓扑序列只存在于有向无环图中。
> - 如何判断它存在拓扑序列，再开始读入数据的时候，我们可以先记录每个点的入度（入度：指指向该路径的边数），然后把所有入度为 0 的点放入队列(注意：这里的队列 tt 要设置成-1)
> - 遍历队列中每个点的子节点，如果它们的 入度数 - 它的一个父结点后为 0 ，那么该节点可以放入队列
> - 最后我们只要判断队列中的点数是否达到 n ，如果达到，说明存在一条从 1 号点到 n 号点的拓扑序列， 否则不存在
> - 拓扑序列可能不止存在一条

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

using namespace std;

const int N = 1e5+10;

int n,m;
int h[N],e[N],ne[N],idx;
int q[N];
int d[N];  //存储每个节点的入度数

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

bool topsort()
{
    int hh=0,tt=-1;
    for(int i=1;i<=n;i++) //遍历每个节点，入度为0则入队
    {
        if(!d[i]) q[++tt]=i;
    }
    while(hh<=tt)
    {
        int t=q[hh++];
        for(int i=h[t];i!=-1;i=ne[i]) //遍历该节点的每一个出边
        {
            int j=e[i];
            if(--d[j]==0) q[++tt]=j;  //节点j入度为0则入队
        }
    }
    return tt==n-1;
}

int main()
{
    cin>>n>>m;
    memset(h,-1,sizeof h);
    for(int i=0;i<m;i++)
    {
        int a,b;
        cin>>a>>b;
        add(a,b);
        d[b]++;   //b节点入度加1
    }
    if(topsort())
    {
        for(int i=0;i<n;i++) printf("%d ",q[i]);
```

```
        puts("");
    }
    else puts("-1");

    return 0;
}
```

最短路径

Dijkstra

AcWing 849. Dijkstra求最短路 I

> - 初始化化每个点到起点的距离为 0x3f3f3f3f,初始化图的边权，每条边初始化为0x3f3f3f3f
> - 边数很多，稠密图用领接矩阵存储
> - 起点到起点的距离可以确定，即dist[1] = 0;
> - 循环遍历 n 次，确定每个点到起点的距离，对于每次遍历，找到一个没有确定最短路的距离:源点最近的点 t,t点的最短路也随之确定(st[t] = true)，用 t 来更新每个点到起点最短距离，由于一开始初始化了所有边的边权为无穷大，所 以不用担心与该点不存在 边 关系的点距离会被更新到(即不存在边关系的点到起点的距离在本次迭代中没有变化)
> - 最后判断最后一个点的距离是否为无穷大，若是，则该点与起点不存在最短路，否则返回dist[n]
> - Dijkstra 只适用于边权为正值的情况

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

const int N = 510;

int g[N][N];  //稠密图用邻接矩阵来存储
int dist[N];  //用来记录每一个点距离起点的距离
bool st[N];   //用于记录该点的最短距离是否确定
int n,m;

int Dijkstra()
{
    memset(dist, 0x3f, sizeof dist);  //初始化距离
    dist[1] = 0;    //第一个点到起点（即自身)的距离为0
    for(int i = 0;i < n;i++)  //迭代n次，确定每个点到起点的最短路
    {
        int t = -1;    //t存储当前访问的点 ， 设置为-1因为Dijkstra适用于不存在负权边的
图

        for(int j = 1;j <= n;j++)   //
```

```
        {
            if(!st[j] && (t == -1 || dist[t] > dist[j]))  //当前点还没有确定最短路
            {
                t = j;
            }
        }
        st[t] = true;

        for(int j = 1;j <= n;j++)  //依次更新每个点到相邻的点路径值
            dist[j] = min(dist[j], dist[t]+g[t][j]);
    }
    if(dist[n] == 0x3f3f3f3f) return -1;  //若第n个点路径为无穷大即不存在最低路径
    return dist[n];
}

int main()
{
    cin>>n>>m;
    memset(g, 0x3f, sizeof g); //初始化图，求最短路径，每个点初始为无限大
    while(m--)
    {
        int x,y,z;
        cin >> x >> y >> z;
        g[x][y] = min(g[x][y], z);  //若发生重边，则保留最短的一条
    }
    cout << Dijkstra() << endl;
    return 0;
}
```

bellman_ford

AcWing 853. 有边数限制的最短路

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

using namespace std;

const int N = 510,M=10010;

struct Edge{
    int a;
    int b;
    int w;
}e[M];
int dist[N];
int back[N];  //备份数组防止串联
int n,m,k;    //k代表最短路径最多只能选取k条边

int bellman_ford()
{
```

```
    memset(dist,0x3f,sizeof dist);
    dist[1]=0;
    for(int i=0;i<k;i++)
    {
        memcpy(back,dist,sizeof dist); //备份上一次迭代后的结果
        for(int j=0;j<m;j++)
        {
            int a=e[j].a,b=e[j].b,w=e[j].w;
            dist[b]=min(dist[b],back[a]+w);
        }
    }
    if(dist[n]>0x3f3f3f3f/2) return -1;
    else return dist[n];
}

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

SPFA

> - 用对列来更新每个点到起点的距离，数据可以存在负权边，但不可以存在负权回路
> - 用邻接表的方式存储，将 1 号点放入队列，同时对该点做标记，记录已经存在于队列中的点，防止队队列出现重复点
> - 对于每一个在队列中的点，循环搜索他们的子节点，如果 子节点到起点的距离 > 父节点到起点的距离 + 他们的边权，则更新该点到起点的距离，同时判断它是否在队列中，若不在，就放入队列，并做标记

```
#include<bits/stdc++.h>
using namespace std;

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int  INF = 0x3f3f3f3f;
```

```
const int mod = 1e5+10;
const int N = 100010;

int n, m;
int e[N];
int ne[N];
int w[N];
int idx;
int h[N];
int dist[N];

bool st[N];

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

int spfa()
{
    queue<int> q;
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    q.push(1);
    st[1] = true;  //标记该点，代表该点已经存在于队列中

    while(q.size())
    {
        auto t = q.front();
        q.pop();
        st[t] = false;  //出队列
        for(int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if(dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                if(!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }

        }
    }
    return dist[n];

}

int main()
```

```
{
    cin >> n >> m;
    memset(h, -1, sizeof h);
    for(int i = 0; i < m; i++)
    {
        int a, b ,c;
        cin >> a >> b >> c;
        add(a, b, c);
    }
    int res = spfa();

    if(res == 0x3f3f3f3f) cout << "impossible" << endl;
    else cout << res << endl;
    return 0;
}
```

AcWing 852. spfa判断负环

> - 总体思路上没有变化，只是多加了存边数的数组用来判断负环，初始时要将所有的点都放入队列

```
#include<bits/stdc++.h>
using namespace std;

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int  INF = 0x3f3f3f3f;
const int mod = 1e5+10;
const int N = 100010;

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

bool st[N];

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

```
bool spfa()
{
    queue<int> q;
    for(int i = 1; i <= n; i++)  //因为有可能1号点走不到的点，所以初始便把所有点放入队列
    {
        st[i] = true;
        q.push(i);
    }

    while(q.size())
    {
        auto t = q.front();
        q.pop();
        st[t] = false;
        for(int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if(dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                cnt[j] = cnt[t] + 1;   //记录该点到起点的边数
                if(cnt[j] >= n) return true;  //当边数超过 n 了，说明肯定有负环存在
                if(!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }

        }
    }
    return false;

}

int main()
{
    cin >> n >> m;
    memset(h, -1, sizeof h);
    for(int i = 0; i < m; i++)
    {
        int a, b ,c;
        cin >> a >> b >> c;
        add(a, b, c);
    }
    if(spfa()) cout << "Yes" << endl;
    else cout << "No" <<endl;
    return 0;
}
```

最小生成树

Prime

> 1. 初始化 g 为无穷大，dist为无穷大，用st来记录该点是否在集合中，res记录生成树的边权和，pre 记录最小生成树边 权关系
> 2. Prime算法：循环迭代 n 次，每次第一步找到集合外到集合中的点的距离最近的点（我们假定最小生成树是一个集合）
> 3. 我们假设这个“最近的点”为 t ，如果它的距离任是无穷大，则说明它是一个孤立点，可以直接输出“impossible”,然后 终止函数，否则我们把这个点加入集合，并将距离加到res中去。
> 4. 用 t 来更新集合外的点到集合的最短距离，并记录边权关系（即pre）

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

using namespace std;

const int N = 510;
int g[N][N];  //存储图
int dist[N];   //存储各个节点到生成树的距离
bool st[N];  //用来判断节点是否被加到生成树中
int pre[N];  //节点的前驱节点
int n,m;   // n个节点，m条边

void prim()
{
    memset(dist,0x3f,sizeof dist); //初始化距离数组为一个很大的数
    int res=0;
    dist[1]=0;  //从 1 号节点开始生成
    for(int i=0;i<n;i++)  //每次循环选出一个点加入到生成树中
    {
        int t=-1;
        //找通往最小生成树的集合的距离最短的点
        for(int j=1;j<=n;j++)   //每个节点一次判断
        {
            if(!st[j] && (t==-1 || dist[j]<dist[t]))  //如果没有在树中，且到树的距离
最短，则选择该点
            {
                t=j;
            }
        }

        if(dist[t] == 0x3f3f3f3f)  //如果是孤立点，直接输出不能 然后退出
        {
            cout<<"impossible";
            return;
        }

        st[t]=1;  //选择该点
        res += dist[t];
```

```
        for(int i=1;i<=n;i++)  //更新生成树外的点到生成树的距离
        {
            if(dist[i] > g[t][i] && !st[i]) //从t到节点i的距离小于原来的距离，则更新
            {
                dist[i]=g[t][i];  //更新距离
                pre[i]=t;   //从t到i的距离更短，i的前驱变为t //相当于建立树枝，长度即
边权
            }
        }
    }
    cout<<res<<endl;
}



int main()
{
    memset(g,0x3f,sizeof g);  //各个点之间的距离初始化为很大的数
    cin>>n>>m;
    while(m--)
    {
        int a,b,w;
        cin>>a>>b>>w;   //输出边的两个顶点和权重
        g[a][b]=g[b][a]=min(g[a][b],w);  //可能存在重边
    }

    prim();  //求最小生成树

    return 0;
}
```

二分图

AcWing 860. 染色法判定二分图

> - 一个图是二分图，当且仅当图中不含有奇数环（组成这个环的边数是奇数）
> - 由此，我们可以通过染色法来解决这道题，当我么们给一个点染上了一种颜色，则给它的子节点染上不同的颜色(这里用 1 和 2 两个数字来当作两种颜色) 当图中不含奇数环时，染色的过程中是不会存在矛盾的
> - 此题用 dfs 和 bfs 都可以做

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

using namespace std;
const int N = 1e5+10,M=2e5+10;
int e[M],ne[M],h[N],idx;
int st[N];
```

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

bool dfs(int u,int color)
{
    st[u]=color;  // u 点染成 color 颜色
    for(int i=h[u];i!=-1;i=ne[i])
    {
        int j = e[i];
        if(!st[j])    //相邻的点没有颜色，则递归处理这个相邻点
        {
            if(!dfs(j,3-color)) return false;  // 3-1=2 , 如果 u 的颜色是 2，则把和
u 相邻的染成 1
        }                                      // 3-2=1 , 如果 u 的颜色是 1，则把和
u 相邻的染成 2
        else if(st[j] == color)  // u ,j 颜色相同，则染色失败
        {
            return false;
        }
    }
    return true;
}

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

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

    bool flag = true;
    for(int i=1;i<=n;i++)
    {
        if(!st[i]){  //未染色
            if(!dfs(i,1))
            {
                flag = false;
                break;
            }
        }
    }

    if(flag) puts("Yes");
```

```
    else puts("No");

    return 0;
}

==============================================================================
//BFS
#include<bits/stdc++.h>
using namespace std;

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int,int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 2e5+10;
const int N = 2e5+10;
int n ,m;
int st[N];
int h[N], e[N], ne[N], idx;


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

bool bfs(int u)
{
    queue<PII> q;
    q.push({u, 1});
    st[u] = 1;
    while(q.size())
    {
        auto x = q.front();
        q.pop();
        int t = x.first;
        int c = x.second;
        for(int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if(!st[j])
            {
                st[j] = 3 - c;
                q.push({j, 3 - c});
            }
            else
            {
                if(st[j] == c) return false;
            }
        }
```

```
    }
    return true;
}

int main()
{
    memset(h, -1, sizeof h);
    cin >> n >> m;
    for(int i = 0; i < m; i++)
    {
        int a, b;
        cin >> a >> b;
        add(a, b);
        add(b, a);
    }
    bool flag = true;
    for(int i = 1; i <= n; i++)
    {
        if(!st[i])
        {
            if(!bfs(i))
            {
                flag = false;
                break;
            }
        }
    }
    if(flag) cout << "Yes" << endl;
    else cout << "No" << endl;
    return 0;
}
```

数论

试除法判定质数

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

试除法分解质因数

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

朴素筛法求素数

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

void get_primes(int n)
{
    for(int i = 2; i <= n; i++)
    {
        if(st[i]) continue;
        primes[cnt ++ ] = i;
        for (int j = i + 1; j <= n; j += i)
        {
            st[j] = true;
        }
    }
}
```

线性筛法求素数

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

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

```
    }
}
```

试除法求所有约数

```
vector<int> get_divisors(int x)
{
    vector<int> res;
    for(int i = 2; i <= x / i; i++)
    {
        if(x % i == 0)
        {
            res.push_back(i);
            if(i != x / i) res.push_back(x / i);
        }
    }
    sort(res.begin(), res.end());
    return res;
}
```

约数个数和约数之和

> 如果 N = p1^c1 * p2^c2 * ... *pk^ck 约数个数： (c1 + 1) * (c2 + 1) * ... * (ck + 1) 约数之和： (p1^0 + p1^1 + ... + p1^c1) * ... * (pk^0 + pk^1 + ... + pk^ck)

欧几里得算法

```
int gcd(int a, int b)
{
    if(b == 0) return a;
    return gcd(b, a % b);
}
```

求欧拉函数

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

```
    }
    if(x > 1) res = res / x * (x - 1);

    return res;
}
```

筛法求欧拉函数

```
int primes[N], cnt;     // primes[]存储所有素数
int euler[N];           // 存储每个数的欧拉函数
bool st[N];         // st[x]存储x是否被筛掉


void get_eulers(int n)
{
    euler[1] = 1;
    for (int i = 2; i <= n; i ++ )
    {
        if (!st[i])
        {
            primes[cnt ++ ] = i;
            euler[i] = i - 1;
        }
        for (int j = 0; primes[j] <= n / i; j ++ )
        {
            int t = primes[j] * i;
            st[t] = true;
            if (i % primes[j] == 0)
            {
                euler[t] = euler[i] * primes[j];
                break;
            }
            euler[t] = euler[i] * (primes[j] - 1);
        }
    }
}
```

快速幂

```
//求 m^k mod p，时间复杂度 O(logk)

int qmi(int m, int k, int p)
{
    int res = 1 % p, t = m;
    while(k)
    {
        if(k & 1) res = res * t % p;
        t = t * t % p;
```

```
        k >>= 1;
    }
    return res;
}
```

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;
    }
    int d = exgcd(b, a % b, y, x);
    y -= (a/ b) * x;
    return d;
}
```

高斯消元

```
// a[N][N]是增广矩阵
int gauss()
{
    int c, r;
    for (c = 0, r = 0; c < n; c ++ )
    {
        int t = r;
        for (int i = r; i < n; i ++ )   // 找到绝对值最大的行
            if (fabs(a[i][c]) > fabs(a[t][c]))
                t = i;

        if (fabs(a[t][c]) < eps) continue;

        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
        for (int i = n; i >= c; i -- ) a[r][i] /= a[r][c];      // 将当前行的首位变
成1
        for (int i = r + 1; i < n; i ++ )       // 用当前行将下面所有的列消成0
            if (fabs(a[i][c]) > eps)
                for (int j = n; 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]) > eps)
                return 2; // 无解
        return 1; // 有无穷多组解
    }

    for (int i = n - 1; i >= 0; i -- )
        for (int j = i + 1; j < n; j ++ )
            a[i][n] -= a[i][j] * a[j][n];

    return 0; // 有唯一解
}
```

递推法求组合数

```
/*
    C[a][b]=C[a - 1][b] + C[a - 1][b - 1]
    可以用动态规划的思想来证明:
    即从 a 个数选出 b 个数， 那么我们可以先计算某一个数选或不选的情况;当选择该数，则可以表示为C[a - 1][b - 1](从a - 1个数中选b个数，因为有一个数已经被确定了);不选该数时，方案数可以表示为C[a - 1][b];
*/
// c[a][b] 表示从a个苹果中选b个的方案数
for (int i = 0; i < N; i ++ )
    for (int j = 0; j <= i; j ++ )
        if (!j) c[i][j] = 1;
        else c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % mod;
```

```
#include<bits/stdc++.h>

using namespace std;

typedef pair<int, int> PII;
typedef long long ll;
const int mod = 1e9 + 7;
const int N = 2010;

int c[N][N];
int n;

void init()
{
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j <= i; j++)
        {
            if(j == 0) c[i][j] = 1;
```

```
            else c[i][j] = (c[i - 1][j - 1] + c[i - 1][j]) % mod;
        }
    }
}

int main()
{
    init();
    cin >> n;
    while(n--)
    {
        int a, b;
        cin >> a >> b;
        cout << c[a][b] << endl;
    }
    return 0;
}
```

通过预处理逆元的方式求组合数

```
//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]
//如果取模的数是质数，可以用费马小定理求逆元
int qmi(int a, int k, int p)    // 快速幂模板
{
    int res = 1;
    while (k)
    {
        if (k & 1) res = (LL)res * a % p;
        a = (LL)a * a % p;
        k >>= 1;
    }
    return res;
}

// 预处理阶乘的余数和阶乘逆元的余数
fact[0] = infact[0] = 1;
for (int i = 1; i < N; i ++ )
{
    fact[i] = (LL)fact[i - 1] * i % mod;
    infact[i] = (LL)infact[i - 1] * qmi(i, mod - 2, mod) % mod;
}
```

Lucas定理

```
///若p是质数，则对于任意整数 1 <= m <= n，有:
    C(n, m) = C(n % p, m % p) * C(n / p, m / p) (mod p)
```

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

int C(int a, int b, int p)  // 通过定理求组合数C(a, b)
{
    if (a < b) return 0;

    LL x = 1, y = 1;  // x是分子，y是分母
    for (int i = a, j = 1; j <= b; i --, j ++ )
    {
        x = (LL)x * i % p;
        y = (LL) y * j % p;
    }

    return x * (LL)qmi(y, p - 2, p) % p;
}

int lucas(LL a, LL b, int p)
{
    if (a < p && b < p) return C(a, b, p);
    return (LL)C(a % p, b % p, p) * lucas(a / p, b / p, p) % p;
}
```

分解质因数求组合数

```
当我们需要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：
    1. 筛法求出范围内的所有质数
    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 + ...
    3. 用高精度乘法将所有质因子相乘

int primes[N], cnt;     // 存储所有质数
int sum[N];     // 存储每个质数的次数
bool st[N];     // 存储每个数是否已被筛掉


void get_primes(int n)      // 线性筛法求素数
{
    for (int i = 2; i <= n; i ++ )
    {
        if (!st[i]) primes[cnt ++ ] = i;
        for (int j = 0; primes[j] <= n / i; j ++ )
```

```
        {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0) break;
        }
    }
}


int get(int n, int p)       // 求n! 中的次数
{
    int res = 0;
    while (n)
    {
        res += n / p;
        n /= p;
    }
    return res;
}


vector<int> mul(vector<int> a, int b)       // 高精度乘低精度模板
{
    vector<int> c;
    int t = 0;
    for (int i = 0; i < a.size(); i ++ )
    {
        t += a[i] * b;
        c.push_back(t % 10);
        t /= 10;
    }

    while (t)
    {
        c.push_back(t % 10);
        t /= 10;
    }

    return c;
}

get_primes(a);  // 预处理范围内的所有质数

for (int i = 0; i < cnt; i ++ )     // 求每个质因数的次数
{
    int p = primes[i];
    sum[i] = get(a, p) - get(b, p) - get(a - b, p);
}

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

for (int i = 0; i < cnt; i ++ )     // 用高精度乘法将所有质因子相乘
    for (int j = 0; j < sum[i]; j ++ )
```

```
        res = mul(res, primes[i]);
```

卡特兰数

> 给定n个0和n个1，它们按照某种顺序排成长度为2n的序列，满足任意前缀中0的个数都不少于1的个数的序列的数量为： `Cat(n) = C(2n, n) / (n + 1)`

NIM游戏

> 给定N堆物品，第i堆物品有Ai个。两名玩家轮流行动，每次可以任选一堆，取走任意多个物品，可把一堆取光，但不能不取。取走最后一件物品者获胜。两人都采取最优策略，问先手是否必胜。
>
> 我们把这种游戏称为NIM博弈。把游戏过程中面临的状态称为局面。整局游戏第一个行动的称为先手，第二个行动的称为后手。若在某一局面下无论采取何种行动，都会输掉游戏，则称该局面必败。 所谓采取最优策略是指，若在某一局面下存在某种行动，使得行动后对面面临必败局面，则优先采取该行动。同时，这样的局面被称为必胜。我们讨论的博弈问题一般都只考虑理想情况，即两人均无失误，都采取最优策略行动时游戏的结果。 NIM博弈不存在平局，只有先手必胜和先手必败两种情况。
>
> 定理： NIM博弈先手必胜，当且仅当 `A1 ^ A2 ^ … ^ An != 0`

公平组合游戏ICG

> 若一个游戏满足：
>
> - 由两名玩家交替行动；
> - 在游戏进程的任意时刻，可以执行的合法行动与轮到哪名玩家无关；
> - 不能行动的玩家判负；
>
>   则称该游戏为一个公平组合游戏。
>
>   NIM博弈属于公平组合游戏，但城建的棋类游戏，比如围棋，就不是公平组合游戏。因为围棋交战双方分别只能落黑子和白子，胜负判定也比较复杂，不满足条件2和条件3。

有向图游戏

> 给定一个有向无环图，图中有一个唯一的起点，在起点上放有一枚棋子。两名玩家交替地把这枚棋子沿有向边进行移动，每次可以移动一步，无法移动者判负。该游戏被称为有向图游戏。 任何一个公平组合游戏都可以转化为有向图游戏。具体方法是，把每个局面看成图中的一个节点，并且从每个局面向沿着合法行动能够到达的下一个局面连有向边。

Mex运算

> 设S表示一个非负整数集合。定义mex(S)为求出不属于集合S的最小非负整数的运算，即： `mex(S) = min{x}`, x属于自然数，且x不属于S

SG函数

> 在有向图游戏中，对于每个节点x，设从x出发共有k条有向边，分别到达节点y1, y2, ..., yk，定义SG(x)为x的后继节点y1, y2, ..., yk 的SG函数值构成的集合再执行mex(S)运算的结果，即：`SG(x) = mex({SG(y1), SG(y2), …, SG(yk)})` 特别地，整个有向图游戏G的SG函数值被定义为有向图游戏起点s的SG函数值，即`SG(G) = SG(s)`。

有向图游戏的和

> 设G1, G2, ..., Gm 是m个有向图游戏。定义有向图游戏G，它的行动规则是任选某个有向图游戏Gi，并在Gi上行动一步。G被称为有向图游戏G1, G2, ..., Gm的和。 有向图游戏的和的SG函数值等于它包含的各个子游戏SG函数值的异或和，即：`SG(G) = SG(G1) ^ SG(G2) ^ … ^ SG(Gm)`

定理

> - 有向图游戏的某个局面必胜，当且仅当该局面对应节点的SG函数值大于0。
> - 有向图游戏的某个局面必败，当且仅当该局面对应节点的SG函数值等于0。

容斥原理

> n = 10, p1 = 2, p2 = 3,求1 ~ 10 中能满足被p1 或 p2 整除的数的个数
>
> 即 2， 3， 4， 6，8， 9， 10
>
> C21 + C22 = 3种集合，即 S2, S3, S2 ∩ S3

```
/*

这里的Pi都为质数
记Si为1 ~ n中能被Pi整除的数的集合;
那么S2 = {2, 4, 6, 8, 10}, S3 = {3, 6, 9},故S2 ∪ S3 = {2, 3, 4, 6, 8, 9, 10};
则S2 ∩ S3 = {6};
∪∑Pi~Pm Si = s2 + S3 - (S2 ∩ S3) (即加减加减加减);
定义|Si|为该集合内元素的个数
定义Sp为1~n中p的倍数的个数，则 Sp = [n / p](下取整)
则有在1 ~ n中 可以被p整除的数的个数为 ([n / p](此时有 p|n (即 p 可以整除 n) ) || [n / p](下取整， 此时 p !| n ))
那么| S2 ∩ S3 | = [n / (2 * 3)](下取整)
组合恒等式：Cn0 + Cn1 + Cn2 + ... + Cnn = 2^n
所以 Cn1 + Cn2 + Cn3 + ... + Cnn = 2^n - 1

*/
```

```
//AcWing 890. 能被整除的数


/*
用二进制表示所有可能的选法，根据容斥原理,求这些集合的并集，需要把选到偶数个集合的结果用答案减去，把选到奇数个的集合的结果加到答案中(1代表该集合被选中，多个1即代表求其交集)
 1: 0000...001
```

```
  2: 0000...010
  3: 0000...011
 */

#include<bits/stdc++.h>
using namespace std;

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef long long ll;
const int INF = 0x3f3f3f3f;
const int N =20;
int p[N];

int n;

int main()
{
    IOS;
    int n,m;
    cin>>n>>m;
    for(int i=0;i<m;i++) cin >> p[i];
    int res = 0; //答案

    for(int i=1;i < 1 << m;i++)  //从 1 枚举到 2^m - 1次方(由组合恒等式得到)
    {
        int t = 1; //用来表示当前被选到的所有数的乘积
        int s = 0;   //用来表示 当前 i 里面包含几个 1，即当前这个选法里选到的集合的个数
        for(int j=0;j<m;j++)
        {
            if(i >> j & 1)
            {
                if((ll)t * p[j] > n)
                {
                    t=-1;
                    break;
                }
                t *= p[j];
                s++;
            }
        }
        if(t != -1)
        {
            if(s % 2) res += n / t;
            else res -= n / t;
        }
    }
    cout << res << endl;
    return 0;
}
```

动态规划

背包问题

01背包

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

using namespace std;

const int N = 1005;

int v[N];   //体积
int w[N];   //价值
int f[N][N];  //f[i][j] ,j体积下前i个物品的最大价值

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

    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=m;j++)
        {
            if(j<v[i])    //当背包容量装不进第i个物品时，则价值等于前i-1个物品
            {
                f[i][j]=f[i-1][j];
            }
            else                 //背包容量够，决策是否选择第i个物品
                {
                    f[i][j]=max(f[i-1][j],f[i-1][j-v[i]]+w[i]);
                }
        }
    }
    cout<<f[n][m]<<endl;

    return 0;
}

==================================================================================
=======
//一维优化
#include <iostream>
#include <algorithm>
```

```
using namespace std;

const int N = 1010;

int n, m;
int v[N], w[N];
int f[N];

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

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

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

    cout << f[m] << endl;

    return 0;
}
```

完全背包

```
// N 种物品和一个容量是 V 的背包，每种物品都有无限件可用

#include<iostream>
using namespace std;
const int N = 1010;
int f[N][N];
int v[N],w[N];
int main()
{
    int n,m;
    cin>>n>>m;
    for(int i = 1 ; i <= n ;i ++)
    {
        cin>>v[i]>>w[i];
    }

    for(int i = 1 ; i<=n ;i++)
    {   for(int j = 0 ; j<=m ;j++)
        {
            for(int k = 0 ; 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]<<endl;
```

```
}
----------------------------------------------------------------------------------
//优化版本

#include<iostream>
#include<algorithm>

using namespace std;

const int N = 1010;

int n,m;
int v[N],w[N];
int f[N];

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

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

    cout<<f[m]<<endl;

    return 0;
}
```

多重背包

```
//每个物品的数量有限

#include<iostream>
#include<algorithm>

using namespace std;

const int N =110;

int n,m;
int v[N],w[N],s[N];
int f[N][N];

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

```
    for(int i=1;i<=n;i++) cin>>v[i]>>w[i]>>s[i];
    for(int i=1;i<=n;i++)
    {
        for(int j=0;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-v[i]*k]+w[i]*k);
            }
        }
    }
    cout<<f[n][m]<<endl;
    return 0;
}
-----------------------------------------------------------------------------------
------
//多重背包二进制优化
#include<iostream>
#include<algorithm>
#include<vector>

using namespace std;
typedef long long LL;
typedef pair<int,int> PII;
const int N = 12010,mod = 1e9+7,M = 2010;


int n,m;
int v[N],w[N];
int f[N];

int main()
{
    cin>>n>>m;
    int cnt = 0;
    for(int i=1;i<=n;i++)
    {
        int a,b,s;   // 第 i 件物品的体积 价值 数量
        cin>>a>>b>>s;
        int k = 1;   //从 1 开始分
        while(k <= s)  //每次把 k 个第 i 个物品打包在一起
        {
            cnt++;    //当前物品的编号++
            v[cnt] = a*k;  //存 k 个物品打包在一起的体积
            w[cnt] = b*k;  //存 k 个物品打包在一起的价值
            s -= k;  // 算过的数减去
            k *= 2;  //打包物品的个数*2(即按 1 2 4 8 来打包)
        }
        if(s>0)  //最后遗留的物品个数
        {
            cnt++;
            v[cnt] = a*s;
            w[cnt] = b*s;
        }
```

```
    }
    for(int i=1;i<=cnt;i++)
    {
        for(int j=m;j>=v[i];j--)
        {
            f[j] = max(f[j],f[j-v[i]]+w[i]);
        }
    }
    cout <<f[m] << endl;
    return 0;
}
```

分组背包

```
//每组物品有若干个，同一组内的物品最多只能选一个

#include<iostream>
#include<algorithm>

using namespace std;

const int N = 110;

int n,m;
int v[N][N],w[N][N],s[N];
int f[N];

int main()
{
    cin>>n>>m;
    for(int i=1;i<=n;i++)
    {
        cin>>s[i];
        for(int j=0;j<s[i];j++)
        {
            cin>>v[i][j]>>w[i][j];
        }
    }

    for(int i=1;i<=n;i++)    //物品组数
    {
        for(int j=m;j>=0;j--)   //背包容量
        {
            for(int k=0;k<s[i];k++)
            {
                if(v[i][k] <= j)
                {
                    f[j]=max(f[j],f[j-v[i][k]]+w[i][k]);
                }
            }
        }
     }
```

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

    return 0;
}
```

线性DP

```
//AcWing 898.数字三角形
#include<iostream>
#include<algorithm>

using namespace std;
const int N = 510,INF=1e9;

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

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

    f[1][1]=a[1][1];
    for(int i=2;i<=n;i++)
        for(int j=1;j<=i;j++)
            f[i][j]=max(f[i-1][j-1]+a[i][j],f[i-1][j]+a[i][j]);

    int res=-INF;
    for(int i=1;i<=n;i++) res=max(res,f[n][i]);

    printf("%d\n",res);
    return 0;
}
```

```
//AcWing 895.最长上升子序列
#include<iostream>
```

```
#include<algorithm>

using namespace std;

const int N = 1010;

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

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

```
//AcWing 896.最长上升子序列II
#include <iostream>
#include <algorithm>

using namespace std;

const int N = 100010;

int n;
int a[N];
int q[N];

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

    int len = 0;
    for (int i = 0; i < n; i ++ )
    {
        int l = 0, r = len;
```

```
        while (l < r)
        {
            int mid = l + r + 1 >> 1;
            if (q[mid] < a[i]) l = mid;  //注意去看题解
            else r = mid - 1;
        }
        len = max(len, r + 1);
        q[r + 1] = a[i];
    }

    printf("%d\n", len);

    return 0;
}
```

```
//AcWing 897.最长公共子序列

#include<iostream>
#include<algorithm>

using namespace std;

const int N = 2010;

char a[N],b[N];
int f[N][N];
int n,m;

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

            //printf("%d %d %d\n",i,j,f[i][j]);
        }
        //puts("");

    }
    printf("%d\n",f[n][m]);
    return 0;
}
```

区间DP

AcWing 282. 石子合并

```
#include<bits/stdc++.h>
using namespace std;

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int, int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 2e5 + 10;
const int N = 2000, M = 2e5 + 10;
int n, m;
ll s1, s2;
int s[N];
int f[N][N];

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

    //f[i][j]:合并区间 [i ~ j] 的所有方案数中代价最小的方案
    //先枚举区间长度，再枚举区间左端点，让k为[i ~ j] 中间的一个下标，把[i ~ j] 分为两段
    for(int len = 2; len <= n; len++)
    {
        for(int i = 1; i + len - 1 <= n; i++)
        {
            int j = i + len - 1;
            f[i][j] = INF;
            for(int k = i; k < j; k++)
            {
                f[i][j] = min(f[i][j], f[i][k] + f[k + 1][j] + s[j] - s[i - 1]);
            }
        }
    }
    cout << f[1][n] << endl;

    return 0;
}
```

计数DP

AcWing 900. 整数划分

```
#include<bits/stdc++.h>
using namespace std;

#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
typedef pair<int, int> PII;
typedef long long ll;
typedef unsigned long long ull;
const int INF = 0x3f3f3f3f;
const int mod = 1e9 + 7;
const int N = 2000, M = 2e5 + 10;
int n;
int dp[N][N];

/**
 * 状态表示：dp[i][j]表示所有总和为i,并且恰好分成j个数的和的方案数的数量
 *
 * 集合划分：1. 最小值为 1 的集合，那么去掉这个1，dp[i - 1][j - 1];
 *          2. 最小值大于1，让每个数都减去一个1，dp[i - j][j];
 *
 * 状态计算： dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j];
 *
 * */

int main()
{
    cin >> n;
    dp[0][0] = 1;

    for(int i = 1; i <= n; i++)
    {
        for(int j = 1; j <= i; j++) // i 最多可以表示成 i 个数的和
        {
            dp[i][j] = (dp[i - 1][j - 1] + dp[i - j][j]) % mod;
        }
    }

    int res = 0;

    // 总和为 n 的方案数为dp[n][1] + dp[n][2] + ... + dp[n][n];
    for(int i = 0; i <= n; i++) res = (res + dp[n][i]) % mod;

    cout << res << endl;

    return 0;
}

============================================================================
//完全背包解法
//状态表示:
//f[i][j]表示只从1~i中选，且总和等于j的方案数
```

```
//状态转移方程:
//f[i][j] = f[i - 1][j] + f[i][j - i];


#include <iostream>
#include <algorithm>

using namespace std;

const int N = 1010, mod = 1e9 + 7;

int n;
int f[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;

    cout << f[n] << endl;

    return 0;
}
```

数位统计DP

```
//AcWing 338.计数问题

#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

const int N = 10;

/*

001~abc-1, 999

abc
    1. num[i] < x, 0
    2. num[i] == x, 0~efg
    3. num[i] > x, 0~999
```

```
*/

int get(vector<int> num, int l, int r)
{
    int res = 0;
    for (int i = l; i >= r; i -- ) res = res * 10 + num[i];
    return res;
}

int power10(int x)
{
    int res = 1;
    while (x -- ) res *= 10;
    return res;
}

int count(int n, int x)
{
    if (!n) return 0;

    vector<int> num;
    while (n)
    {
        num.push_back(n % 10);
        n /= 10;
    }
    n = num.size();

    int res = 0;
    for (int i = n - 1 - !x; i >= 0; i -- )
    {
        if (i < n - 1)
        {
            res += get(num, n - 1, i + 1) * power10(i);
            if (!x) res -= power10(i);
        }

        if (num[i] == x) res += get(num, i - 1, 0) + 1;
        else if (num[i] > x) res += power10(i);
    }

    return res;
}

int main()
{
    int a, b;
    while (cin >> a >> b , a)
    {
        if (a > b) swap(a, b);

        for (int i = 0; i <= 9; i ++ )
            cout << count(b, i) - count(a - 1, i) << ' ';
```

```
        cout << endl;
    }

    return 0;
}
```

状态压缩DP

```
//AcWing 291.蒙德里安的梦想
//朴素写法

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

using namespace std;

const int N = 12, M = 1 << N;

int n, m;
long long f[N][M];
bool st[M];

int main()
{
    while (cin >> n >> m, n || m)
    {
        for (int i = 0; i < 1 << n; i ++ )
        {
            int cnt = 0;
            st[i] = true;
            for (int j = 0; j < n; j ++ )
                if (i >> j & 1)
                {
                    if (cnt & 1) st[i] = false;
                    cnt = 0;
                }
                else cnt ++ ;
            if (cnt & 1) st[i] = false;
        }

        memset(f, 0, sizeof f);
        f[0][0] = 1;
        for (int i = 1; i <= m; i ++ )
            for (int j = 0; j < 1 << n; j ++ )
                for (int k = 0; k < 1 << n; k ++ )
                    if ((j & k) == 0 && st[j | k])
                        f[i][j] += f[i - 1][k];

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

```
    }
    return 0;
}

----------------------------------------------------------------------------
//优化写法

#include <cstring>
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

typedef long long LL;

const int N = 12, M = 1 << N;

int n, m;
LL f[N][M];
vector<int> state[M];
bool st[M];

int main()
{
    while (cin >> n >> m, n || m)
    {
        for (int i = 0; i < 1 << n; i ++ )
        {
            int cnt = 0;
            bool is_valid = true;
            for (int j = 0; j < n; j ++ )
                if (i >> j & 1)
                {
                    if (cnt & 1)
                    {
                        is_valid = false;
                        break;
                    }
                    cnt = 0;
                }
                else cnt ++ ;
            if (cnt & 1) is_valid = false;
            st[i] = is_valid;
        }

        for (int i = 0; i < 1 << n; i ++ )
        {
            state[i].clear();
            for (int j = 0; j < 1 << n; j ++ )
                if ((i & j) == 0 && st[i | j])
                    state[i].push_back(j);
        }
```

```
        memset(f, 0, sizeof f);
        f[0][0] = 1;
        for (int i = 1; i <= m; i ++ )
            for (int j = 0; j < 1 << n; j ++ )
                for (auto k : state[j])
                    f[i][j] += f[i - 1][k];

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

    return 0;
}
```

```
//AcWing 91.最短Hamilton路径
#include <cstring>
#include <iostream>
#include <algorithm>

using namespace std;

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 = 0; i < 1 << n; i ++ )
        for (int j = 0; j < n; j ++ )
            if (i >> j & 1)
                for (int k = 0; k < n; k ++ )
                    if (i >> k & 1)
                        f[i][j] = min(f[i][j], f[i - (1 << j)][k] + w[k][j]);

    cout << f[(1 << n) - 1][n - 1];

    return 0;
}
```

树形DP

```
//AcWing 285.没有上司的舞会
#include<cstring>
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 6010;

int n;
int h[N],e[N],ne[N],idx;
int happy[N];  //每个职工的高兴度
int f[N][2];
//f[u][1]:以u为根节点的子树并且包括u的总快乐指数,
//f[u][0]:以u为根节点并且不包括u的总快乐指数;
bool has_fa[N];  //用来判断当前节点是否有父节点

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

void dfs(int u)
{
    f[u][1]=happy[u];  //若选当前节点，快乐值至少为该节点的快乐值

    for(int i=h[u];i!=-1;i=ne[i])  //遍历以u为头结点的所有子节点
    {
        int j=e[i];  //取出相邻点存的值
        dfs(j);
        f[u][1] += f[j][0];  //上司来，我就不来
        f[u][0] += max(f[j][0],f[j][1]);  //上司不来，我看心情来
    }
}

int main()
{
    scanf("%d",&n);
    for(int i=1;i<=n;i++) scanf("%d",&happy[i]); //输入每个人的高兴度
    memset(h,-1,sizeof h);
    for(int i=0;i<n-1;i++)
    {
        int a,b;
        scanf("%d%d",&a,&b);
        add(b,a);      //表示b是a的上司
        has_fa[a]=true;   //说明a有爸爸（即上司）
    }
    int root=1;  //用来找根节点
```

```
    while(has_fa[root]) root++;  //找到没有上司的节点，即为根节点
    dfs(root);  //从根节点开始深度搜索
    printf("%d\n",max(f[root][0],f[root][1]));  //比较头节点的两种方案 <大oss到底来不来好>
    return 0;
}
```

记忆化搜索

```
//AcWing 901.滑雪
#include<cstring>
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 310;

int n,m;
int g[N][N];  //网格滑雪场
int f[N][N];  //状态转移方程

int dx[4]={-1,0,1,0},dy[4]={0,1,0,-1};

int dp(int x,int y)
{
    int &v = f[x][y];  //&是引用符号，相当于给f[x][y]起了个新名字v，v发生变化，f[x][y]也会发生变化
    if(v!=-1) return v;  //若已经计算过了，就可以直接返回答案
    v=1;  //先赋值1,因为最少也有一个起点可滑
    for(int i=0;i<4;i++)
    {
        int a= x+dx[i],b=y+dy[i];
        if(a >= 1 && a<=n && b>=1 && b<=m && g[x][y]>g[a][b])  //判断该点是否能走
            v=max(v,dp(a,b)+1); //更新
    }
    return v;
}

int main()
{
    scanf("%d%d",&n,&m);
    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=m;j++)
        {
            scanf("%d",&g[i][j]);
        }
    }
    memset(f,-1,sizeof f);
```

```
    int res=0;  //最后答案
    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=m;j++)
        {
            res=max(res,dp(i,j));
        }
    }
    printf("%d\n",res);
    return 0;
}
```

贪心