一、输入输出总结：

1.知识点：

```
// 1.cin：
    (1) 键盘的回车符(\r)在缓冲区中被转换为换行符(\n)
    (2) cin会忽略并清除缓冲区开头处的tab/space/enter，直接从第一个有效字符读取
    (3) 【cin遇到空格/换行/tab，结束读取，不会读取空格/换行，继续留在缓冲区】
    (4) cin可以作为if、while的条件，成功读取返回true，遇到结束符（windows中为：ctrl+Z ， Unix中为：ctrl+D）或无效输入（比如用一个整型变量来接收一个字符），返回false
    (5) 一旦接收失败（while if中条件为false），cin失效，想要重新使用需要调用cin.clear()，但接收失败的数据仍然存在缓冲区，cin.clear()之后，使用cin获取的时之前缓冲区中的数据

// 2.getline
    (1) 头文件：#include<string>
    (2) getline(istream& is,string& str,char 结束符)
    (3) 默认结束符为'\n'，可以指定，比如指定为','
    (4) 可以读入空格、tab，遇到换行结束读取，会读取结束符（换行），但结束符会被丢弃掉
    (5) 由于cin不会读取最后的换行符，而getline遇到换行符又会停止读取，所有使用cin之后要先getchar()，吞掉最后的换行符才能使用getline

// 3.stringstream
    (1) 头文件：#include<sstream>
    (2) stringstream ss(str)
    (3) 将字符串转变成流，后序ss当作cin用，也可以作为getline的参数

// 4.C++输入结束
    (1) 机考结束输出时，会输入EOF，表示输入结束了
    (2) EOF是个宏，End Of File，文件尾标志。，从数值上来看，就是整数-1。
    (3) 当读文件操作时，遇到文件结束位置或读数据出错均会返回EOF
    (4) 如何在键盘输入时，产生EOF呢？ 不同的系统方法不同：linux系统下，在输入回车换行后的空行位置，按ctrl+d，windows系统下，在输入回车换行后的空行位置，按ctrl+z，再回车确认

// 5.四舍五入输出
#include<iostream>
#include <iomanip>// 需要包含同文件
using namespace std;

int main()
{
    double res = 3.567;

    cout << res << endl;
    cout << fixed << setprecision(2) << res << endl;

    return 0;
}
```

2.举例：

```
// 1.直接输入一个数/字符串/字符
int main() {

    int data;
    // string data;
    // char data;

    while (cin >> data) {
        cout << data << endl;
    }
    // 当cin检查输入被放在循环条件中来终止循环，循环终止后不再允许用户输入，即使调用cin>>也没用；
    // 如需要用户输入，则需要设置标识符为有效才可，调用clear()
    cin.clear();
    cin >> data;
    cout << data << endl;

    return 0;
```

```
}


// 2.在终端输入固定数目的整型数字/字符串/字符，并存到数组中，中间以空格/换行分隔
// 例如：
// 输入：1 5 3 2 8 9
int main() {

    int n;              // 输入数据的个数
    cin >> n;
    vector<int>nums(n); // 也可以是 string、char等

    for (int i = 0; i < n; i++) {
        cin >> nums[i];
    }

    return 0;
}
// 2.1在终端输入固定数目的整型数字/字符串/字符，并存到数组中，中间以字符sep分隔
int main() {

    int n;              // 输入数据的个数
    cin >> n;
    char sep;           // 用于分割的字符
    cin >> sep;
    vector<int>nums(n); // 也可以是 string、char等

    for (int i = 0; i < n-1; i++) {
        cin >> nums[i] >> sep;
    }
    cin >> nums[n - 1];

    return 0;
}

// 3.在终端一行中输入非固定数目的整型数字/字符串/字符，并存到数组中，中间以空格分隔
// 例如：输入的个数不定，需要用换行符\n来作为每一行结束的标志
// 输入1：1 5 3
// 输入2：1 5 3 6 9
int main() {

    vector<int>nums;
    int num;

    while (cin >> num) {
        nums.push_back(num);
        // if (getchar() == '\n') {
        if (cin.get() == '\n') {        // 判断是否到达行尾
            break;
        }
    }

    return 0;
}

// 3.1多行数据，每行数据个数固定、多种类型
// 例如，每行数据分别为：name age，以空格分割
// 输入1：  alice 18
//          bob 20
//          ...
class Data{
private:
    Data():name(""),age(0){}
    Data(string na,int ag):name(na),age(ag){}
public:
    string name;
    int age;
};
int main() {
```

```
    vector<Data>arr;
    string input;

    while (getline(cin,input)) {
        // 需要头文件#include<sstream>，
        // 将字符串转变成流，后序ss当作cin用，也可以作为getline的参数
        stringstream ss(input);
        string name;
        int age;
        ss>>name>>age;

        arr.push_back(Data(name,age));
    }

    return 0;
}
// 如果行数固定
int main() {
    int n;// 行数
    cin>>n;
    getchar();

    vector<Data>arr(n);

    for(int i=0;i<n;i++){
        cin>>arr[i].name>>arr[i].age;
    }

    return 0;
}

// 3.2多行数据，每行数据类型一样，个数不定
// 例如：拓扑排序的依赖数据，共有n各编号，0..n-1
// 第一行：n
// 后面的行，编号i，编号i依赖的编号
// 1 2 6 3
int main() {
    int n;
    cin>>n;
    getchar();// 重点关注

    vector<vector<int>>arr(n);
    string input;
    while(getline(cin,input)){
        stringstream ss(input);
        int i;
        ss>>i;
        int num;
        while(ss>>num){
            arr[i].push_back(num);
        }

    }

    return 0;
}
```

二、模板总结

1.前缀和数组

(1) 模板说明

1. 前缀和技巧用于快速、频繁地计算一个区间内元素的和
2. 有两种前缀和计算方式：

```
1.presum[i]包含nums[i]
presum[i]=nums[0]+..+nums[i]

2.presum[i]不包含nums[i]，此时需要presum的size比nums的size大1
presum[i]=nums[0]+..+nums[i-1]
presum[0]=0

一维数组一般用第一种方式，二维数组一般用第二种方式
```

3. 核心代码：

```
// 计算区间[l,r]的和
int f(vector<int>& nums,int l,int r){
    vector<int>presum(nums.size()+1,0);// 采用第二种方式

    // 计算前缀和数组
    for(int i=0;i<nums.size();i++){
        presum[i+1]=presum[i]+nums[i];
    }

    return presum[r+1]-presum[l];
}
```

(2) 模板举例

303.区域检索和-数组不可变

304.二维区域检索和-矩阵不可变

560.和为K的子数组

862.和至少为 K 的最短子数组

2.差分数组

(1) 模板说明

1. 差分数组适用于频繁对原始数组的一些区间的元素进行增减

```
比如，给定数组nums，然后要求在区间[2,6]全部加1、在区间[3,9]全部减3、在区间[0,4]全部加2、...
求最终的nums？
```

2. 差分数组等于原数组相邻元素做差；原数组等于差分数组求前缀和（方式一求前缀和）；前缀和数组求差分数组也等于原数组
3. 由于差分数组求前缀和就等于原数组，那么对差分数组diff[i]+val，相当于给原数组[i..end)全部加val。因此想要将原数组在区间[l,r]内全部+val，只需diff[l]+=val并且diff[r+1]-=val。相当于在l位置施加影响，在r+1位置消除影响
4. 核心代码：

```
vector<int>nums;// 原数组

// 求差分数组
vector<int>diff(nums.size());// 差分数据
diff[0]=nums[0];
for(int i=1;i<diff.size();i++){
    diff[i]=nums[i]-nums[i-1];
}

// 将区间[l,r]范围内的原数组元素加上val
int l;
int r;
int val;
diff[l]+=val;
if(r+1<diff.size()){
    diff[r+1]-+val;
}

// 求经过一些区间操作之后的数组nums
```

```
nums[0]=diff[0];
for(int i=1;i<nums.size();i++){
    nums[i]=nums[i-1]+diff[i];
}
```

(2) 模板举例

370.区间加法

1109.航班预定统计

1094.拼车

3.滑动窗口

(1) 模板说明

1. 滑动窗口模板主要解决：符合模式串p某种条件下，求字符串s的子串问题
2. 核心代码：

```
// 在s中寻找符合与p有关条件的子串
void slidingWindow(string s, string p) {
    // needs存储p的词频
    // window存储滑动窗口中的词频，不一定是将窗口的字符全部加入
    unordered_map<char, int>needs, window;

    // 匹配上的字符的数量，需要needs[c]==window[c]
    int valid = 0;

    for (char c : p) {
        needs[c]++;
    }

    // 窗口左闭右开[l,r)
    int l = 0;
    int r = 0;

    while (r < s.size()) {
        // c是将要移入窗口的字符
        char c = s[r];
        // 窗口增加
        r++;
        // 将c移入窗口之后，窗口内的数据要更新
        ...

        while (滑动窗口收缩的条件) {
            // d是将移出窗口的字符
            char d = s[l];
            // 窗口收缩
            l++;
            // 将d移出窗口之后，窗口内的数据要更新
            ...
        }
        // 这两个 ... 处的操作分别是右移和左移窗口更新操作，它们的操作是完全对称的
    }
}
```

(2) 模板举例

76.最小覆盖子串-hot100

567.字符串的排列

438.找到字符串中所有字母异位词-hot100

4.二分查找（二分答案法）

(1) 模板说明

1. 二分法的本质是二段性，以数组元素为自变量x，带入某个函数f(x)，能够判断答案是在数组的左区间还是右区间
2. 最简单的二段性就是有序数组查找target
3. 二分答案法也是利用二段性，首先确定最终答案的取值范围`[minv,maxv]`（这是前提，如果确定不了，则无法使用二分答案法）；然后以这个范围作为搜索的左右边界`[l,r]=[minv,maxv]`；然后在这个区间内二分查找最终的答案；如果二分的中点m符合条件，则可能是最终的答案，更新res，更新左边界或右边界；如果二分的中点m不符合条件，更新右边界或左边界
4. 二分答案法核心代码：

```
void mianfunc(){
    int l;// 根据题目确定的答案的左边界
    int r;// 根据题目确定的答案的右边界
    int res;
    while(l<=r){
        int m=l+((r-l)>>1);

        int cur=f(m);

        if(根据cur判断，m是否能作为答案){
            res=m;// 更新答案
            l=m+1;// 怎么更新，依据题意，是找更大的答案，还是找更小的答案
            // r=m-1;
        }else{
            r=m-1;
            // l=m+1;
        }
    }

    return res;
}

int f(m){
    // 假设以m为答案，...
}
```

(2) 模板举例

34.在排序的数组中查找元素的第一个和最后一个位置-hot100

33.搜索旋转数组-hot100

```
// 一种新的思路：
// 二分法先找到旋转点，然后在再左边/右边二分查找
```

面试题11：旋转数组的最小数字

面试题53-II：0~n-1中缺失的数字

240.搜索二维矩阵II&面试题4

875.爱吃香蕉的珂珂

1011.在D天内送达包裹的能力

410.分割数组的最大值

793.阶乘函数后K个

本题需要用到：172.阶乘后的零

2141.同时运行N台电脑的最长时长

5.单调栈

(1) 模板说明

1. 单调栈的意义：用O(n)的时间复杂度遍历一遍数组找到每个元素前后最近的更小/大元素位置
2. 维护栈内元素递增或递减（以递增为例），当前遍历到的元素满足递增，则直接入栈；当前遍历到的原理不满足递增，则将栈内元素出栈，直到当前遍历到的元素可以入栈为止；需要出栈的栈内元素就找到了比它小的元素最近的位置，右边比它小的就是当前遍历到的元素，左边比它小的就是它出栈后的栈顶元素
3. 寻找更大元素，则维持栈内元素递减；寻找更小元素，则维持栈内元素递增
4. 核心代码：

```
// 给定一个数组，找到每个位置对应元素的下一个更大元素所在位置，如果右边没有更大元素了，则为-1
vector<int> f(vector<int>& nums){
    vector<int>res(nums.size(),-1);
    stack<int>sk;//
    sk.push(0);

    for(int i=1;i<nums.size();i++){
        if(nums[i]<=nums[sk.top()]){// 为了方便理解，没有进行优化
            // 满足栈内元素递减，直接入栈
            sk.push(i);
        }else{
            // 不满足栈内元素递减，出栈，计算栈顶元素对应结果
            while(!sk.empty()&&nums[i]>nums[sk.top()]){
                int idx=sk.top();
                sk.pop();
                res[idx]=i;
            }
            sk.push(i);
        }
    }

    return res;
}
```

(2) 模板举例

84.柱状图中最大的矩形-hot100

85.最大矩形-hot100

496.下一个更大元素I

739.每日温度-hot100

503.下一个更大元素II

316.除去重复字母

6.单调队列

(1) 模板说明

1. **单调队列用于解决滑动窗口中求最值的问题**
2. 用双端队列deque
3. 给定一个数组nums和一个窗口大小k，每次窗口向右移动一位，求滑动窗口内的最大值/最小值，以最大值为例
4. 用一个双端队列deque维护窗口内可能成为为最大值的元素的下标，并且队头元素就是当前窗口的最大值；deque中的元素是递减的；遍历nums，遍历到的元素如果不满足deque的单调性要求，则将队列尾部元素出队，因为当前元素的下标比队尾元素下标大（更晚出滑动窗口），值还比队尾元素大（可能成为窗口中的最大值），那么队尾元素再不可能成为窗口中的最大值，所以可以放心出队；如果队头元素已经出窗口了，则要将其删除，因此在队头队尾都有可能删除元素，这也是为什么用双端队列的原因。
5. 核心代码：

```
vector<int> maxSlidingWindow(vector<int>& nums,int k){
    deque<int>dq;// 单调队列用deque

    vector<int>res(nums.size()-k+1);

    for(int i=0;i<nums.size();i++){
        while(!dq.empty()&&nums[dq.back()]<=nums[i]){
            // 不满足滑动进入队列的要求，则将队尾元素弹出
            dq.pop_back();
        }
        dq.push_back(i);
        if(dq.front()<=i-k){// 队首元素已经出窗口了，需要删除
            dq.pop_front();
        }
        if(i+1>=k){// 窗口形成，更新res
            res[i-k+1]=nums[dq.front()];
        }
    }

    return res;
}
```

(2) 模板举例

239.滑动窗口最大值-面试题59-I-hot100

面试题59-II：队列的最大值

7.图的遍历

(1) 模板说明

1. 深度优先遍历dfs：

```
void mainfunc(vector<vector<int>>& nexts){
    // nexts为采用邻接表法表示的图
    // dfs遍历该图：
    int n=nexts.size();// 图中节点个数
    vector<bool>visited(n,false);// 图中可能存在环，防止走回头路

    for(int i=0;i<nexts.size();i++){// 图可能不是相互连同的
        if(visited[i]){
            continue;
        }
        dfs(nexts,visited,i);
    }
}

void dfs(vector<vector<int>>& nexts,vector<bool>visited,int i){
    if(visited[i]){
        return;
    }
    visited[i]=true;

    for(int next:nexts[i]){
        dfs(nexts,visited,next);
    }
}

// 对节点是否遍历过的判断也可以放到for循环里
void dfs(vector<vector<int>>& nexts,vector<bool>visited,int i){
    visited[i]=true;

    for(int next:nexts[i]){
        if(!visited[i]){
            dfs(nexts,visited,next);
        }
    }
```

```
}
```

2. 广度优先遍历bfs

```
// 1.二叉树的bfs
void bfsTree(Node* root){
    if(root==nullptr){
        return;
    }

    queue<Node*>q;
    q.push(root);
    int depth=0;// 当前所在的深度

    // while循环控制一层一层往下走，for循环利用sz变量控制从左到右遍历每一层二叉树节点
    while(!q.empty()){
        depth++;
        int len=q.size();
        // 从左到右遍历每一的节点
        for(int i=0;i<len;i++){
            Node* cur=q.front();
            q.pop();

            // 处理逻辑...

            // 将下一层节点放入队列
            if(cur->left!=nullptr){
                q.push(cur->left);
            }
            if(cur->right!=nullptr){
                q.push(cur->right);
            }
        }
    }
}

// 2.多叉树的bfs
void bfsMultiTree(Node* root){
    if(root==nullptr){
        return;
    }

    queue<Node*>q;
    q.push(root);
    int depth=0;
    while(!q.empty()){
        depth++;

        int len=q.size();

        for(int i=0;i<len;i++){
            Node* cur=q.front();
            q.pop();

            // 处理逻辑...

            // 将下一层节点放入队列
            for(cur的孩子节点child){
                q.push(child);
            }
        }
    }
}

// 3.图的bfs
// 邻接表法表示图
void bfsGraph(vector<vector<int>>nexts){
    // 从0号节点开始bfs
    int len=nexts.size();// 节点个数
```

```
    queue<int>q;
    q.push(0);

    vector<bool>visited(len,false);// 避免走回头路
    visited[0]=true;

    int step=0;

    while(!q.empty()){
        int len=q.size();
        // 将当前队列中的所有节点向四周扩散一步
        for(int i=0;i<len;i++){
            int cur=q.front();
            q.pop();

            // 处理逻辑...

            // 将cur的相邻节点加入队列
            for(int next:nexts[cur]){
                if(!visited[next]){
                    q.push(next);
                    visited[next]=true;
                }
            }
        }
    }
}
```

(2) 模板举例

797.所有可能的路径

下面是bfs的题目：

752.打开转盘锁

773.滑动谜题

126.单词接龙 II

417.太平洋大西洋水流问题

332.重新安排行程

给你一份航线列表 tickets ， 其中 tickets[i] = [fromi, toi] 表示飞机出发和降落的机场地点。请你对该行程进行重新规划排序。

所有这些机票都属于一个从 JFK（肯尼迪国际机场）出发的先生，所以该行程必须从 JFK 开始。如果存在多种有效的行程，请你按字典排序返回最小的行程组合。

例如，行程 ["JFK", "LGA"] 与 ["JFK", "LGB"] 相比就更小，排序更靠前。
假定所有机票至少存在一种合理的行程。且所有的机票 必须都用一次 且 只能用一次。

```
vector<string> findItinerary(vector<vector<string>>& tickets) {
    if(tickets.size()==0){
        return vector<string>();
    }
    // 当res中的元素个数等于n+1时，说明已经将票用完，得到答案
    int n=tickets.size();

    // 邻接表法表示图
    // key：起点，value：<终点，票数>
    unordered_map<string,map<string,int>>mp;
    for(auto& t:tickets){
        string from=t[0];
        string to=t[1];
        mp[from][to]++;
    }

    vector<string>res;
    res.push_back("JFK");
    dfs(mp,res,n);
```

```
    return res;
}
bool dfs(unordered_map<string,map<string,int>>& mp,vector<string>& res,int n){
    if(n+1==res.size()){
        return true;
    }
    string place=res[res.size()-1];// 当前所在地方

    // 遍历从当前所在地，可以飞往的地方
    for(map<string,int>::iterator it=mp[place].begin();it!=mp[place].end();it++){

        if((*it).second==0){// 票已经用完
            continue;
        }
        (*it).second--;
        res.push_back((*it).first);
        if(dfs(mp,res,n)){
            return true;
        }
        res.pop_back();
        (*it).second++;
    }
    return false;
}
```

8.拓扑排序

(1) 模板说明

1. 要求图是有向无环图（存在入度为0的节点），如果最后不存在入度为0的点，则不能进行拓扑排序
2. 在具体题目中，需要将题目给出的数据转换成有向图，一般使用邻接表法表示图`vector<vector<int>>nexts`，`nexts[i]`表示节点i的指向的节点数组
3. 在建图的过程中需要计算每个节点的入度，用`vector<int>indegree`表示，`indegree[i]`表示i节点的入度；
4. 在得到`indegree`之后，遍历，将入度为0的节点加入队列`queue<int>inzero`中；
5. 每次从队列`inzero`中取出一个元素`cur`，加到结果数组`res`中，并将其影响消除，也即将`nexts[cur]`中的元素的入度全部减1，如果减1之后，有入度变为0的节点，则将其加入`inzero`
6. 核心代码：

```
vector<int> f(vector<vector>& nums){
    // 根据nums建图，假设有n个节点
    vector<vector<int>>nexts(n);     // 邻接表表示图
    vector<int>indegree(n,0);        // 入度矩阵
    queue<int>inzero;                // 入度为0的节点

    for(遍历nums){
        初始化nexts;
        初始化indegree;
    }

    // 将入度为0的节点加入inzero
    for(int i=0;i<indegree.size();i++){
        if(indegree[i]==0){
            inzree.push(i);
        }
    }

    vector<int>res;// 拓扑排序结果
    while(!inzero.empty()){
        int len=inzero.size();// 每次消除一层
        for(int i=0;i<len;i++){
            int idx=q.front();
            q.pop();
            res.push_back(idx);
            // 消除idx的影响
            for(auto next:nexts[idx]){
                indegree[next]--;
```

```
                if(indegree[next]==0){
                    q.push(next);
                }
            }
        }
    }
    if(res.size()!=n){
        // 说明图中存在环，没有完成拓扑排序
        return vector<int>();
    }
    return res;
}
```

(2) 模板举例

207.课程表-hot100

210.课程表II

9.二分图

(1) 模板说明

1. 什么是二分图？二分图的顶点集可以分割成两个互不相交的子集，图中每条边依附的两个顶点都分属于这两个子集，子集内的顶点不存在边
2. 判断二分图其实就是用两种颜色尝试给图的每一个节点染色，如果能够做到相邻节点的颜色不同即为二分图
3. 用dfs判断二分图/给图染色，核心代码：

```
// 给定一个邻接表法表示的图，判断是否为二分图
bool isBipartite(vector<vector<int>>& nexts){
    int n=nexts.size();
    vector<bool>visited(n,false);
    vector<int>color(n,0);// 尝试给每个节点染色，0表示尚未染色，1表示染成红色，-1表示染成蓝色

    bool res=true;
    for(int i=0;i<nexts.size();i++){// 图可能不是相互连同的
        if(visited[i]){
            continue;
        }
        color[i]=1;
        dfs(res,nexts,color,visited,i);
        if(!res){
            return res;
        }
    }

    return res;
}

void dfs(bool& res,vector<vector<int>>& nexts, vector<int>& color,vector<bool>& visited,int i){
    visited[i]=true;
    for(int next:nexts[i]){
        if(color[next]==0){
            // 说明尚未染色，也就是没有访问过，因此visited也可以省略
            color[next]=-color[i];
            dfs(res,nexts,color,visited,next);
        }else{
            // 说明之前访问过，染过色
            if(color[next]!=-color[i]){
                // 说明相邻节点的同色了
                res=false;
                return;
            }
        }
    }
}
```

(2) 模板举例

785.判断二分图

886.可能的二分法

10.并查集

(1) 模板说明

1. 并查集（指针向上的树）的重要思想是用集合中的一个元素（父节点）代表集合，主要用于解决一些元素分组的问题，管理一系列不相交的集合
2. 并查集支持两种操作：

```
1. 合并（Union）：把两个不相交的集合合并为一个集合
2. 查询（Find）：查询两个元素是否在同一个集合中，或者说是查询节点的父节点，通过比较两个父节点是否相同来判断是否在同一个集合

时间复杂度都是O(1)
```

3. 核心代码：

```
class UnionFindSet{
private:
    vector<int>fathers;// fathers[i]：i节点的父节点为fathers[i]
    int cnt;// 并查集中的集合数

public:
    // 初始时，每个节点为一个集合
    UnionFindSet(int n){// n为并查集的节点数
        this->cnt=n;
        for(int i=0;i<n;i++){
            fathers[i]=i;
        }
    }

    // 查找
    int findFather(int i){
        if(fathers[i]==i){
            return i;
        }
        fathers[i]=findFather(fathers[i]);// 路径压缩
        return fathers[i];
    }

    // 合并
    void unionSet(int i,int j){
        int fa_i=findFather(i);
        int fa_j=findFather(j);
        if(fa_i==fa_j){
            return;
        }
        fathers[fa_i]=fa_j;// 将i所在的集合接在j所在的集合上
        cnt--;
    }

    // 判断是否连同
    bool connected(int i,int j){
        int fa_i=findFather(i);
        int fa_j=findFather(j);

        return fa_i==fa_j;
    }

    // 返回当前的集合数
    int getCnt(){
        return cnt;
    }
```

```
};
```

(2) 模板举例

990.等式方程的可满足性

399.除法值

684.冗余连接

1584.连接所有点的最小费用

本题是kruskal算法，其中用到了并查集

547.省份数量

11.Kruskal 最小生成树算法

(1) 模板说明

1. 最小生成树：包含图的所有节点，形成树结构（不存在环），且整体边的权值最小。一般求无向加权图的最小生成树
2. Kruskal算法是站在边的角度贪心，每次将权重最小的边尝试加入生成树边的集合，加入之后不能形成环
3. 怎么判断加入某条边是否会形成环呢？形成环，说明之间已经将该边的两个顶点加入最小生成树集合了，因此可以通过判断边的两个顶点是否在同一集合来判断是否会形成环，而判断是否在同一集合可以用并查集
4. 怎么获得最小权重的边呢？按照边的权重对边进行排序，也可以用优先队列
5. 核心代码：

```
// 假设图是用边的数组vector<vector<int>>edges给出的，其中edges[i][0]、edges[i][1]表示这条边链接的两个顶点，edges[i][2]表示该跳变的权重
// 一共有n个顶点，编号分别非0 1..n-1
// 返回最小生成树的边的集合
vector<vector<int>> minimumSpanningTree(int n,vector<vector<int>>& edges){
    UnionFindSet ufs(n);// 并查集

    sort(edges.begin(),edges.end(),
        [](vector<int>& a,vector<int>& b){
            return a[2]<b[2];
        }
    );

    vector<vector<int>>res;

    for(int i=0;i<edges.size();i++){
        int p1=edges[i][0];
        int p2=edges[i][0];
        if(ufs.connected(p1,p2)){
            // 该边连接的两个顶点都已经在生成树集合中了
            continue;
        }
        res.push_back(edges[i]);
        ufs.unionSet(p1,p2);
        if(ufs.getCnt()==1){
            // 说明所有节点都加入集合了，最小生成树构建完成
            break;
        }
    }

    return res;
}
```

(2) 模板举例

1584.连接所有点的最小费用

12.Prim 最小生成树算法

(1) 模板说明

1. Kruskal算法是站在边的角度做贪心，而Prim算法是站在顶点的角度做贪心
2. 从任意一个顶点A开始，A的所有边被解锁，选择其中权值最小的边加入最小生成树集合，已经选择的边后序不再选择；现在来到该边的另一个顶点B，B的所有边被解锁；从所有已解锁的边中选择权值最小的边加入最小生成树集合...
3. 上述过程什么时候停止呢？加入最小生成树集合的边包含所有顶点时就可以停止了，因此需要一个能够快速检查当前节点是否在最小生成树集合的数据结构—`unordered_set`
4. Prim算法不像Kruskal算法可用边是静态的，一次性给出的，可以用排序来选择权值最小的边；而Prim算法可用的边是动态解锁的，因此需要用优先队列`priority_queue`来存储可用的边，从而快速获得权值最小的边
5. 核心代码：

```
// 注意Prim算法中的vector<vector<vector<int>>>& edges，不同于Kruskal算法
// edges[i]表示节点i所有相连的边，其中edges[i][j][0]==i、edges[i][j][1]表示j边链接的两个顶点，edges[i][j][2]表示j边的权重
// 一共有n个顶点，编号分别非0 1..n-1
// 返回最小生成树的边的集合
class my_com{
public:
    bool operator()(vector<int>& a,vector<int>& b){
        return a[2]>b[2];
    }
};
vector<vector<int>> minimumSpanningTree(int n,vector<vector<vector<int>>>& edges){

    priority_queue<vector<int>,vector<vector<int>>,my_com>q;// 存放已经解锁的边
    unordered_set<int>node_s;               // 已经加入到最小生成树集合中的节点
    vector<vector<int>>res;                 // 存放已经加入到最小生成树的边

    for(int i=0;i<n;i++){                   // 防止有的节点不相互联通

        if(node_s.find(i)==node_s.end()){   // 该顶点没有加入最小生成树
            node_s.insert(i);

            // 解锁与i相连的所有边
            for(vector<int>& e:edges[i]){
                q.push(e);
            }

            while(!q.empty()){
                vector<int>e=q.top();
                q.pop();
                if(node_s.find(e[1])==node_s.end()){
                    res.push_back(e);
                    node_s.insert(e[1]);
                    // 解锁与e[1]相连的所有边
                    for(vector<int>& e2:edges[e[1]]){
                        q.push(e2);
                    }
                }

                if(node_s.size()==n){
                    break;
                }

            }

        }

        if(node_s.size()==n){
            break;
        }
    }

    return res;
```

```
}
```

(2) 模板举例

1584.连接所有点的最小费用

Kruskal和Prim都是最小生成树算法，优先使用Kruskal算法

13.Dijkstra 单源最短路径算法

(1) 模板说明

1. Dijkstra算法是给定一个图和一个节点，求该节点到图中其他节点的最小距离
2. 初始时，给定节点到自己的距离为0，到其他节点的距离为无穷大，用一个`vector`表示，也是最后的返回结果
3. 用一个优先队列`priority_queue`存储可能发生距离更新的节点状态`State`，每次从优先队列中选择一个距离最小的节点，尝试经过该节点能否使与其相邻的节点距离变小，如果可也，更新之；用过的节点距离确定，不再更新，不再使用，需要从优先队列中弹出
4. 核心代码：

```
// 用邻接表法表示图，但又和之前的不太一样
// edges[i]是一个list，表示与节点i相连的所有边
// edges[i][j]是一个vector，表示与节点i相连的边j的信息：to和weight
// k：源节点
// 节点编号从0开始，0 1 ...
class State{// 存放每个节点的状态
public:
    int id;// 节点id
    int curDis;// 目前 节点到源节点的最小距离
    State(int i,int d):id(i),curDis(d){}
};
class my_com{
public:
    bool operator()(const State& s1,const State& s2){
        return s1.curDis>s2.curDis;
    }
};
vector<int> dijkstra(vector<list<vector<int>>>edges,int k){
    int n=edges.size();                             // 节点个数

    vector<int>res(n,INT_MAX);                      // 目前 源节点到个节点的最小值
    res[k]=0;

    priority_queue<State,vector<State>,my_com>q;    // 存储可能发生距离更新的节点状态State
    q.push(State(k,0));

    while(!q.empty()){
        State cur=q.top();
        q.pop();

        // 尝试经过该节点能否使与其相邻的节点距离变小
        for(vector<int>edge:edges[cur.id]){
            int to=edge[0];
            int weight=edge[1];

            if(res[to]>cur.curDis+weight){
                res[to]=cur.curDis+weight;
                q.push(State(to,res[to]));
            }
        }
    }

    return res;
}
```

(2) 模板举例

743.网络延迟时间

1514.概率最大的路径

1631.最小体力消耗路径

14.DFS/回溯算法

(1) 模板说明

1. 回溯算法就是个多叉树（图）的遍历问题，关键就是在前序遍历和后序遍历的位置做一些操作

```
vector<vector<int>> mainfunc(参数){
    vector<vector<int>>res;
    vector<int>cur;
    // vector<bool>visited;// 有时需要
    dfs(res,cur,其他参数);

    return res;
}

void dfs(vector<vector<int>>& res,vector<int>& cur,其他参数){
    if(终止条件){
        res.push_back(cur);
        return;
    }

    for(next:nexts[i]){
        处理next;              // 先序遍历的位置
        dfs(res,cur,其他参数);
        回溯，撤销处理结果;       // 后序遍历的位置
    }
}
```

(2) 模板举例

子集、组合、排列：

1. 子集、组合、排列问题，一般都有两种思维模式：a.`nums`来到`i`位置，当前位置要不要；b.画出问题的决策树，通过`dfs`来遍历决策树，关键是确定当前节点的子节点有哪些
2. 预排序可以用来去重，或者剪枝
3. 用`vector<bool>used`或者`unordered_set<int>`来记录已经使用过的数

78.子集-hot100

90.子集II

77.组合

39.组合总和-hot100

40.组合总和II

216.组合总和III

46.全排列-hot100

47.全排列II

面试题38：字符串的排列

698.划分为k个不相等的子集

岛屿类问题：

1. 岛屿类问题一般会给出一个二维vector，其中的元素为0/1表示海水，1/0表示陆地，问有多少岛屿？岛屿的面积？等..
2. 一般会把遍历过的岛屿编程海洋—淹掉，可用不用维护visited数组

3. 核心代码：

```
int mainfunc(vector<vector<int>>& nums){
    if(nums.size()==0||nums[0].size()==0){
        return 0;
    }
    int m=nums.size();
    int n=nums[0].size();
    int res;
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            if(nums[i][j]==1){// 假设1为岛屿
                dfs(nums,i,j);
            }
        }
    }

    return res;
}

void dfs(vector<vector<int>>& nums),int i,int j){
    if(i<0||i>=nums.size()||j<0||j>=nums[0].size()){
        return;
    }

    dfs(nums,i+1,j);
    dfs(nums,i-1,j);
    dfs(nums,i,j+1);
    dfs(nums,i,j-1);
}
```

200.岛屿数量-hot100

1254.统计封闭岛屿的数目

1020.飞地的数量

695.岛屿的最大面积

1905.统计子岛屿

130.被包围的区域

934.最短的桥

本题用到了dfs和bfs，特别是bfs，可以作为如何对矩阵这种图进行bfs的一种参考

51.N皇后

面试题17：打印从1到最大的n位数

面试题13：机器人的运动范围

79.单词搜索&面试题12-hot100

17.电话号码的字母组合-hot100

332.重新安排行程

15.区间类问题

(1) 模板说明

1. 区间类题目一般要先对区间进行排序，按照起点进行排序、按照终点进行排序
2. 画图，通过画图分析两个区间相对位置的可能情况
3. 差分数组是解决区间类问题的常用技巧

(2) 模板举例

435.无重叠区间

452.用最少数量的箭引爆气球

1024.视频拼接

1228.删除被合并的区间

56.合并区间-hot100

986.区间列表的交集

252.会议室

253.会议室II-hot100

406.根据身高重建队列-hot100

16.数组

(1) 模板说明

1. 左右双指针：一般相向而行、也有背道而行的（回文相关，中心拓展法）
2. 快慢双指针：一前一后，快指针相当于是个探针，先去探测情况
3. 只要数组有序，就应该想到双指针技巧
4. 滑动窗口也属于双指针技巧

(2) 模板举例

面试题57-I：和为s的两个数字

面试题57-II：和为s的连续正整数序列

581.最短无序连续子数组

11.盛最多水的容器-hot100

26.删除有序数组中的重复项

83.删除排序链表中的重复元素（与上题相似）

27.移除元素

167.两数之和II-输入有序数组

1.两数之和-hot100

15.三数之和-hot100

18.四数之和

344.反转字符串

(3) 其他数组相关题目

面试题61：扑克牌中的顺子

面试题42：连续子数组的最大和

面试题29：顺时针打印矩阵

448.找到所有数组中消失的数字

面试题3：数组中的重复数字

287.寻找重复数-hot100

238.除自身以外数组的乘积&面试题66-hot100

169.多数元素&面试题39-hot100

152.乘积最大子数组-hot100

128.最长连续序列-hot100

49.字母异位词分组-hot100

48.旋转图像-hot100

42.接雨水-hot100

31.下一个排列-hot100

134.加油站

645.错误的集合

659.分割数组为连续子序列

391.完美矩形

870.优势洗牌

54.螺旋矩阵

135.分发糖果

81.搜索旋转排序数组 II

88.合并两个有序数组

455.分发饼干

17.字符串

面试题67：把字符串转换成整数

面试题58-I：翻转单词顺序

面试题58-II：左旋转字符串

面试题50：第一个只出现一次的字符

面试题20：表示数组的字符串

面试题5：替换空格

面试题45：把数组排成最小的数

241.为运算表达式设计优先级

242.有效的字母异位词

87.扰乱字符串

1417.格式化字符串

205.同构字符串

318.最大单词长度乘积

415.字符串相加

18.链表

(1) 模板说明

1. 伪头节点的使用
2. 双指针：双指针在两个链表中；快指针先走k步；快指针每次走两步，慢指针每次走一步

(2) 模板举例

面试题35：复杂链表的复制

面试题18：删除链表的节点

面试题6：从尾到头打印链表

234.回文链表-hot100

206.反转链表&面试题24-hot100

148.排序链表-hot100

141.环形链表 & 142.环形链表II & 面试题23

一系列链表相交的问题：

(1) 如何判断链表有没有环?

面试题23

(2) 两个都没环的链表是否相交？相交求第一个交点

leetcode 160-hot100：给你两个单链表的头节点 headA 和 headB ，请你找出并返回两个单链表相交的起始节点。如果两个链表不存在相交节点，返回 null 。

```
// 方法一：将一个链表添加如set中，遍历第二个链表，判读当前节点是否存在set中，如果存在即为所求
// 方法二：双指针，求两个链表的长度，让长的那个链表先走差值步，在一块走，相等时即为所求，如果不相交，最后也会都是空
// 求链表的长度
int getListLength(ListNode* h){
    int len=0;
    while(h!=nullptr){
        h=h->next;
        len++;
    }
    return len;
}
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    if(headA==nullptr||headB==nullptr){return nullptr;}
    int lenA=getListLength(headA);
    int lenB=getListLength(headB);
    if(lenA<lenB){
        ListNode* temp=headA;
        headA=headB;
        headB=temp;
    }
    int d=abs(lenA-lenB);
    while(d>0){
        headA=headA->next;
        d--;
    }
    while(headA!=headB){
        headA=headA->next;
        headB=headB->next;
    }
    return headA;
}
// 方法三：
// 你变成我，走过我走过的路，
// 我变成你，走过你走过的路，
// 然后我们便相遇了..
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    ListNode* node1=headA;
    ListNode* node2=headB;
    while(node1!=node2){
        node1=node1?node1->next:headB;
        node2=node2?node2->next:headA;
    }
    return node1;
}
```

(3) 两个都有环的链表是否相交？相交求第一个交点

若相交，存在两种相交的情况：在环之前相交和在环中相交。第二种情况：在环前相交（根据入环节点是否相同判断）等效于两个无环链表相交的问题，只不过终止节点不再是空指针，而是入环节点（通过上面的分析，入环节点已经求出）；第一种情况和第二种情况判断：让其中一个链表经过入环节点后继续走直到回到入环节点，如果在这个过程中和另一个链表的入环节点相遇，则是情况二（返回哪个入环节点都对）；否则是情况一。

21.合并两个有序链表&面试题25-hot100

86.分隔链表

23.合并K个升序链表-hot100

面试题22：链表中倒数第k个节点

19.删除链表的倒数第N个节点-hot100

876.链表的中间节点

2.两数相加-hot100

24.两两交换链表中的节点

19.二叉树

(1) 模板说明

解二叉树题目的思维模式分两类：

1. 二叉树的递归套路：基于后序遍历，问他的左子树要信息，问他的右子树要信息，然后生成自己的信息向上返回，所需信息可能有多种，用一个类来传递
2. 遍历的思维：能否通过遍历二叉树得到答案

二叉树的遍历：一般使用递归形式的dfs和bfs，非递归形式的dfs很少用

1. 深度优先遍历：递归、非递归（用栈）

```
// 递归
void dfs(Node* root){
    if(root==nullptr){
        return;
    }
    // 先序
    dfs(root->left);
    // 中序
    dfs(root->right);
    // 后序
}
```

```
// 非递归
// 1.先序
void pre(Node* root){
    if(root==nullptr){
        return;
    }
    stack<Node*>sk;
    sk.push(root);
    while(!sk.empty()){
        Node* cur=sk.top();
        sk.pop();
        // 处理逻辑..
        if(cur->right!=nullptr){
            sk.push(cur->right);
        }
        if(cur->left!=nullptr){
            sk.push(cur->left);
        }
    }
}

// 2.中序
void in(Node* root){
```

```
    if(root==nullptr){
        return;
    }
    stack<Node*>sk;
    Node* cur=root;
    while(!sk.empty()||cur!=nullptr){
        while(cur!=nullptr){// 整课左子树进栈
            sk.push(cur);
            cur=cur->left;
        }
        cur=sk.top();
        sk.pop();
        // 处理逻辑..
        cur=cur->right;
    }
}

// 3.后序
// 先序：头左右，先序撇：头右左；该处理逻辑时不处理，而是进入到辅助栈中，待所有节点入辅助栈后，再处理逻辑；由于栈先进后出的特性，处理顺序为左右头，即实现了后序
void post(Node* root){
    if(root==nullptr){
        return;
    }
    stack<Node*>sk1;
    stack<Node*>sk2;
    sk1.push(root);
    while(!sk1.empty()){
        Node* cur=sk1.top();
        sk1.pop();
        sk2.push(cur);
        if(cur->left!=nullptr){
            sk1.push(cur->left);
        }
        if(cur->right!=nullptr){
            sk1.push(cur->right);
        }
    }

    while(!sk2.empty()){
        Node* cur=sk2.top();
        sk2.pop();
        // 处理逻辑..
    }
}
```

2. 广度优先遍历：用队列

```
void bfsTree(Node* root){
    if(root==nullptr){
        return;
    }

    queue<Node*>q;
    q.push(root);
    int depth=0;// 当前所在的深度

    // while循环控制一层一层往下走，for循环利用sz变量控制从左到右遍历每一层二叉树节点
    while(!q.empty()){
        depth++;
        int len=q.size();
        // 从左到右遍历每一的节点
        for(int i=0;i<len;i++){
            Node* cur=q.front();
            q.pop();

            // 处理逻辑...

            // 将下一层节点放入队列
```

```
            if(cur->left!=nullptr){
                q.push(cur->left);
            }
            if(cur->right!=nullptr){
                q.push(cur->right);
            }
        }
    }
}
```

(2) 模板举例

面试题54：二叉搜索树的第k大节点

面试题36：二叉搜索树与双向链表

面试题33：二叉搜索树的后序遍历序列

面试题34：二叉树中和为某一值的路径

面试题26：树的子结构

面试题8：二叉树的下一节点

617.合并二叉树

543.二叉树直径

437.路径总和III-hot100

297.二叉树的序列化与反序列化&面试题37-hot100

236.二叉树的最近公共祖先 & 面试题68-I

124.二叉树的最大路径和

面试题55-I：二叉树的深度

104.二叉树的最大深度&面试题55-II-hot100

面试题32-I：从上到下打印二叉树

102.二叉树的层序遍历-面试题32II-hot100

面试题32-III：从上到下打印二叉树III

101.对称二叉树&面试题28

98.验证二叉搜索树-hot100

94.二叉树的中序遍历-hot100

226.翻转二叉树&面试题27-hot100

116.填充每个节点的下一个右侧节点指针

114.将二叉树展开为链表-hot100

654.最大二叉树

105.从前序与中序遍历序列构造二叉树&面试题7-hot100

106.从中序与后序遍历序列构造二叉树

889.根据前序和后序遍历构造二叉树

230.二叉搜索树中第K小的元素

538.把二叉搜索树转换为累加树-hot100

96.不同的二叉搜索树-hot100

95.不同的二叉搜索树II

1373.二叉搜索树的最大键值和

111.二叉树的最小深度

652.寻找重复的子树

99.恢复二叉搜索树

669.修剪二叉搜索树

144.二叉树的前序遍历

637.二叉树的层平均值

1110.删点成林

110.平衡二叉树

20.括号类题目

(1) 模板说明

1. 一般括号类问题使用栈，遇到左括号进行，遇到右括号，与栈顶元素匹配
2. 对应只含有一种括号的问题，只要在任意位置左括号的数量不小于右括号的数量，并且最后左右括号的数量相同，即为有效括号

(2) 模板举例

301.删除无效的括号-hot100

20.有效的括号-hot100

22.括号生成

32.最长有效括号-hot100

921.使括号有效的最少添加

1541.平衡括号字符串的最少插入次数

以下为动态规划类题目

21.动态规划

面试题60：n个骰子的点数

面试题46：把数字翻译成字符串

面试题47：礼物的最大价值

面试题49：丑数

面试题14-I：剪绳子

面试题14-II：剪绳子II

面试题10-I：斐波那契数列

面试题10-II：青蛙跳台阶问题

70.爬楼梯-hot100

647.回文子串

494.目标和-hot100

312.戳气球-hot100

221.最大正方形-hot100

1277.统计全为1的正方形子矩阵（和上题相似）

139.单词拆分-hot100

62.不同路径-hot100

55.跳跃游戏-hot100

3.无重复字符的最长子串&面试题48.-hot100

931.下降路径最小和

53.最大子数组和&面试题45-hot100

10.正则表达式匹配&面试题19-hot100

877.石头游戏

64.最小路径和-hot

174.地下城游戏

514.自由之路

650.只有两个键的键盘

91.解码方法

542.01 矩阵

413.等差数列划分

22.背包问题

(1) 模板说明

1. 背包问题就是给定N种物品，每种物品有重量和价值两个属性，分别用`weights[i]`，价值为`values[i]`给出。在不同的条件限制下，使得背包种装载的价值最大化
2. 限制条件可能是：背包装载的重量不超过w，背包装载的重量刚好等于w，等
3. 01背包每种物品只有一个，完全背包每种物品有无限个，多重背包每种物品有限个
4. `dp[i][j]`：在考虑`0..i`种物品时，在条件`j`的限制下获得的最大价值；`i`物品可以取`0..k`个

(2) 01背包

1. 给定一个可以装载`w`重量的背包和`N`个物品，每个物品有重量和价值两个属性，其中`i`个物品的重量为`weights[i]`，价值为`values[i]`。问用这个背包装载的最大价值?
2. `dp[i][j]`：`0..i`件物品，在容量不超过`j`的情况下，能装的最大价值
3. 状态转移：对于第`i`件物品可以选择 要/不要，二者求最大

$$dp[i][j] = max \begin{cases} dp[i-1][j], & \text{不要}i\text{物品} \\ dp[i-1][j-weights[i]] + values[i], & \text{要}i\text{物品，前提是}j - w[i] >= 0 \end{cases}$$

4. `dp[i][j]`依赖于上一行的结果，所有base case为`dp[0][j]`
5. 核心代码：

```
int knapsack01(vector<int>& weights,vector<int>& values,int w){
    if(w<=0||weights.size()==0){
        return 0;
    }

    int n=weights.size();
    vector<vector<int>>dp(n,vector<int>(w+1,0));

    // base case
    for(int j=1;j<=w;j++){
        dp[0][j]=j>=weights[0]?values[0]:0;
    }

    for(int i=1;i<n;i++){
        for(int j=0;j<=w;j++){
```

```
            dp[i][j]=dp[i-1][j];
            if(j-weights[i]>=0){
                dp[i][j]=max(dp[i][j],dp[i-1][j-weights[i]]+values[i]);
            }
        }
    }

    return dp[n-1][w];
}
// 空间优化略
```

(3) 完全背包

1. 给定一个可以装载 `w` 重量的背包和 `N` 种物品，每种物品有重量和价值两个属性，其中 `i` 种物品的重量为 `weights[i]`，价值为 `values[i]`，每种物品数量无限。问用这个背包装载的最大价值?
2. `dp[i][j]`：`0..i` 种物品，在容量不超过 `j` 的情况下，能装的最大价值
3. 状态转移：对于第 `i` 件物品可以选择 要0件、1件、...、k件，求最大

$$dp[i][j]=max\begin{cases}dp[i-1][j], & \text{不要}i\text{物品}\\ dp[i-1][j-k*weights[i]]+k*values[i], & \text{要}k\text{件}i\text{物品，前提是}j-k*w[i]>=0\end{cases}$$

4. 优化：由于

$$dp[i][j-w[i]]=max\begin{cases}dp[i-1][j-w[i]], & \text{不要}i\text{物品}\\ dp[i-1][j-(k-1)*weights[i]]+(k-1)*values[i], & \text{要}k-1\text{件}i\text{物品，前提是}j-(k-1)*w[i]>=0\end{cases}$$

所以：

$$dp[i][j]=max\begin{cases}dp[i-1][j], & \text{不要}i\text{物品}\\ dp[i][j-weights[i]]+values[i], & \text{前提是}j-k*w[i]>=0\end{cases}$$

4. `dp[i][j]` 依赖于上一行的结果，所有base case为 `dp[0][j]`
5. 核心代码：

```
int knapsackComplete(vector<int>& weights,vector<int>& values,int w){
    if(weights.size()==0||w<=0){
        return 0;
    }
    int n=weights.size();
    vector<vector<int>>dp(n,vector<int>(w+1,0));
    for(int j=1;j<=w;j++){
        dp[0][j]=(j/weights[0])*values[0];
    }

    for(int i=1;i<n;i++){
        for(int j=0;j<=w;j++){
            dp[i][j]=dp[i-1][j];
            if(j-weights[i]>=0){
                dp[i][j]=max(dp[i][j],dp[j-weights[i]]+values[i]);
            }
        }
    }

    return dp[n-1][w];
}
// 空间优化略
```

(4) 多重背包

1. 给定一个可以装载 `w` 重量的背包和 `N` 种物品，每种物品有重量和价值两个属性，其中 `i` 种物品的重量为 `weights[i]`，价值为 `values[i]`，`i` 物品数量为 `nums[i]`。问用这个背包装载的最大价值?
2. `dp[i][j]`：`0..i` 种物品，在容量不超过 `j` 的情况下，能装的最大价值
3. 状态转移：对于第 `i` 件物品可以选择 要0件、1件、...、k件，求最大

$$dp[i][j]=max\begin{cases}dp[i-1][j], & \text{不要}i\text{物品}\\ dp[i-1][j-k*weights[i]]+k*values[i], & \text{要}k\text{件}i\text{物品，前提是}j-k*w[i]>=0\text{并且}k<=nums[i]\end{cases}$$

4. 核心代码：

```
int knapsackK(vector<int>& weights,vector<int>& values,vector<int>& nums,int w){
    if(weights.size()==0||w<=0){
        return 0;
    }
    int n=weights.size();

    vector<vector<int>>dp(n,vector<int>(w+1,0));

    for(int j=0;j<=w;j++){
        if(j<=weights[0]*nums[0]){
            dp[0][j]=(j/weights[0])*values[0];
        }else{
            dp[0][j]=values[0]*nums[0];
        }
    }

    for(int i=1;i<n;i++){
        for(int j=0;j<=w;j++){
            dp[i][j]=dp[i-1][j];
            // 这里没法像多重背包一样优化时间复杂度
            for(int k=1;k<=nums[i]&&j-k*weights[i]>=0;k++){
                dp[i][j]=max(dp[i][j],dp[i-1][j-k*weights[i]]+k*values[i]);
            }
        }
    }

    return dp[n-1][w];
}
// 空间优化略
```

(5) 模板举例

279.完全平方数-hot100

322.零钱兑换-hot100

518.零钱兑换II

416.分割等和子集-hot100

474.一和零

23.股票买卖问题

(1) 模板说明

1. 给定一个整数数组 `prices`，其中 `prices[i]` 表示某只股票第 `i` 天的价格，每天可以决定是否买入/卖出股票，最多持有一股股票（买入之后必须卖出，才能再次买入），在一些条件限制下，问能获得的最大利润?
2. 限制条件：a.只能买卖股票一次；b.买卖股票次数不限；c.买卖股票次数为2次；d.买卖股票次数为k次；e.买卖股票次数不限，但是卖出股票之后，第二天无法买入股票（含冷冻期）；f.买卖股票次数不限，但每次买入股票都要付手续费，分别对应以下6题
3. 采用优先状态机进行求解：假设股票的买卖次数限制为k次，那么

   `vector<vector<int>>dp(n,vector<int>(2*k,0))`，

   `dp[i][j]` 表示第i天股票交易的状态为 `j` 时的最大收益，正在进行 `x=(j/2+1)` 次交易，`j%2==0` 第 `i` 天手里持有股票，`j%2==1` 第 `i` 天手里不持有股票。

   持有股票可能是今天买入，也可能是之前买入；不持有股票可能是今天卖出了，也可能是之前卖出了，且一直没有再买入

(2) 模板举例

121.买卖股票的最佳时机&面试题63-hot100

122.买卖股票的最佳时机II

123.买卖股票的最佳时机III

188.买卖股票的最佳时机IV

309.最佳买卖股票时机含冷冻期-hot100

714.买卖股票的最佳时机含手续费

24.打家劫舍问题

(1) 模板说明

1. 前两题打家劫舍，`f(i)`：从`nums[i..len-1]`获取的最高金额；需要保证无后效性，i位置是否盗取和之前的决定无关，保证i位置可以自由选择，所以`f(i)=max(f(i+1),f(i+2)+nums[i])`，分别是盗取`i`房间和不盗取`i`房间
2. 第三题打家劫舍，二叉树的递归套路
3. 第四题打家劫舍，二分答案法

(2) 模板举例

198.打家劫舍-hot100

213.打家劫舍II

337.打家劫舍III

2560.打家劫舍IV

25.最长递增子序列

(1) 模板说明

1. 给定一个整数数组nums，返回其中最长的严格递增子序列的长度。子序列可以删除数组中的元素，但是不能改变其余元素的顺序
2. 有的题目会是一个二维数组，需要先按照每行的第一个元素排序，然后再求最长递增子序列
3. 核心代码：

```
// 方法一：动态规划，O(N^2)
// dp[i]：以i位置为结尾的最长递增子序列的长度
// dp[i]等于，nums[0..i-1]<nums[i]下，dp[0..i-1]+1：相当于尝试以nums[0..i-1]为最长递增子序列的倒数第二个元素
int lengthOfLIS(vector<int>& nums){
    if(nums.size()<2){
        return nums.size();
    }

    int res=1;
    vector<int>dp(nums.size(),1);
    for(int i=1;i<nums.size();i++){
        for(int j=i-1;j>=0;j--){
            if(nums[i]>nums[j]){
                dp[i]=max(dp[i],dp[j]+1);
            }
        }
        res=max(res,dp[i]);
    }

    return res;
}
```

```
// 方法二：二分查找
// dp[i]：长度为i+1的递增子序列末尾元素的最小值，随时更新
// dp是一个严格递增的数组
// 如果nums[i]>dp[end]，直接将nums[i]插入dp末尾；否则替换掉dp中大于等于nums[i]的最小元素
int lengthOfLIS(vector<int>& nums){
    if(nums.size()<2){
        return nums.size();
```

```
    }

    int res=1;
    vector<int>dp;
    dp.push_back(nums[0]);

    for(int i=1;i<nums.size();i++){
        if(nums[i]>dp[dp.size()-1]){// 直接插在dp的末尾
            dp.push_back(nums[i]);
        }else{// 替换掉dp中大于等于nums[i]的最小元素
            int idx=f(dp,nums[i]);
            dp[idx]=nums[i];
        }
    }
    return dp.size();
}
// 在增序数组dp中，查找大于等于num的最小元素的下标
int f(vector<int>& dp,int num){
    int l=0;
    int r=dp.size()-1;
    int idx=-1;
    while(l<=r){
        int m=l+((r-l)>>1);
        if(dp[m]==num){
            return m;
        }
        if(dp[m]>num){
            idx=m;
            r=m-1;
        }else{
            l=m+1;
        }
    }
    return idx;
}
```

(2) 模板举例

300.最长递增子序列-hot100

354.俄罗斯套娃信封问题

26.子序列/子串问题

(1) 模板说明

1. 子序列不同于子串/子数组，子序列不要连续，只要前后顺序不变即可
2. 动态规划常见的尝试方法有：a.从左往右尝试，`i`位置的元素要还是不要，`dp[i]`一般依赖于`dp[i-1]`；b.范围上尝试，`[i,j]`范围内的答案是多少，`dp[i][j]`一般依赖于`dp[i+1][j-1]`、`dp[i+1][j]`、`dp[i][j-1]`；c.以`i`位置为结尾的结果是啥，如果是两个字符串就是`dp[i][j]`，`s1`以`i`为结尾、`s2`以`j`为结尾
3. 子序列的`dp`一般是在范围内（`[0,i]`和`[i,j]`）的结果
4. 最长递增子序列（lc：300）：上面总结的最长递增子序列中的动态规划方法，就是必须以`i`为结尾时的最长递增子序列
5. 最长公共子数组（lc：718）：`dp[i][j]`公共子数组必须以`nums1[i]`、`nums[j]`为结尾的最长长度
6. 最长公共子序列（lc：1143）：`dp[i][j]`表示`s1`在`[0..i-1]`范围、`s2`在`[0..j-1]`范围的最长公共子序列（`s1`以`i`为结尾、`s2`以`j`为结尾）
7. 两个字符串的删除操作（lc：583）：`dp[i][j]`表示`s1`在`[0..i-1]`范围、`s2`在`[0..j-1]`范围变得相同删除的最少操作。本题可以转化成最长公共子序列，因为两个字符串删完之后就是最长公共子序列
8. 两个字符串的最小ASCII删除和（lc：712）：与上题类似，但不能转化成最长公共子序列
9. 最长回文子串（lc：5）：`dp[i][j]`表示子串`s[i..j]`是否为回文，本题使用中心扩展法效率更高，详细见manacher算法
10. 最长回文子序列（lc：516）：`dp[i][j]`表示在`[i,j]`范围内回文子序列的最长长度
11. 编辑距离（lc：72）：`dp[i][j]`表示`s1`来到`i`位置、`s2`来到`j`位置，将`s1[0..i]`变为`s2[0..j]`的最小编辑距离

(2) 模板举例

300.最长递增子序列-hot100

718.最长重复子数组

1143.最长公共子序列

583.两个字符串的删除操作

712.两个字符串的最小ASCII删除和

5.最长回文子串-hot100

516.最长回文子序列

72.编辑距离-hot100

27.manacher算法

(1) 模板说明

1. **manacher算法（`O(N)`）是用于求最长回文子串的，是对中心扩展法（暴力`O(N^2)`）的加速，也可以使用动态规划（`O(N^2)`）**
2. 中心扩展法：以`i`位置为中心向两边扩，求出以`i`为中心的最长回文子串，为了使偶数长度的回文也可以通过此法来解决，将原字符串各字符之间增加特殊字符`#`（包括头尾）
3. 中心扩展法核心代码：

```
// 给原字符串增加#
string getManacherStr(const string& s){
    string res="#";
    for(int i=0;i<s.size();i++){
        res+=s[i];
        res+="#";
    }
    return res;
}
// 将带有#的回文字符串还原成不带#的字符串
string f(string& str,int l,int r){
    string res="";
    for(int i=l+1;i<r;i+=2){
        res+=str[i];
    }
    return res;
}
// 中心扩展法
string longestPalindrome(string s){
    if(s.size()<2){
        return s;
    }
    string str=getManacherStr(s);
    int len=str.size();
    int x=-2;// 回文字符串的起始位置
    int y=-1;// 回文字符串的终止位置
    for(int i=1;i<len-1;i++){
        int j=i-1;
        for(;j>=0&&2*i-j<len;j--){
            if(str[j]!=str[2*i-j]){
                break;
            }
        }
        if(y-x+1<2*(i-j-1)+1){
            x=j+1;
            y=2*i-j-1;
        }
    }
    return f(str,x,y);
}
```

4. Manacher算法核心代码

```
// 两个辅助函数同上
// manacher算法：对暴力方法的加速
// 明确几个概念：
//  1.回文直径：整个回文的长度；
//  2.回文半径：从中心（包括）到一端的回文长度
//  3.字符串的回文半径数组：以每个位置为中心，求回文半径
//  4.之前所有扩的位置中到达的最右回文右边界R（回文不包括该位置），初始值为-1
//  5.取得右边界时的中心点C：初始值为-1

// 算法步骤：依据从中心向两边扩的思想，当前来到i位置
//  1.如果i=R（不可能大于R） --> 暴力扩
//  2.如果i<R --> 根据i关于C的对称点ip=2C-i的回文情况分成三类：
//      2.1 ip的回文整个在L..R里（L是以C为中心的回文左边界，不包括）-->i的回文半径等于ip的回文半径
//      2.2 ip的回文区域有部分跑到了L的外面 --> i的回文半径等于i到R这段
//      2.3 ip的回文区域左边界刚好在L-->i的回文半径至少为i到R这段，是否会更长，需要进一步验证
string longestPalindrome(string s){
    if(s.size()<2){
        return s;
    }
    string str=getManacherStr(s);
    int len=str.size();
    int x=-2;// 回文字符串的起始位置
    int y=-1;// 回文字符串的终止位置

    vector<int>pArr(len);// 回文半径数组
    int R=-1;// 之前所有扩的位置中到达的最右回文右边界R，不包括R
    int C=-1;// 达到R时的回文中心

    for(int i=0;i<len;i++){
        int ip=2*C-i;// i关于C的对称点
        pArr[i]=i<R?min(pArr[ip],R-i):1;// 至少不用验证的部分
        // 尝试往外扩
        int r=i+pArr[i];// 开始验证的地方
        int l=i-pArr[i];
        while(r<len&&l>=0){
            if(str[l]==str[r]){
                pArr[i]++;
                r++;
                l--;
            }else{
                break;
            }
        }
        // 更新R和C
        if(i+pArr[i]>R){
            R=i+pArr[i];
            C=i;
        }
        // 更新答案
        if(y-x+1<r-l-1){
            x=l+1;
            y=r-1;
        }
    }
    return f(str,x,y);
}
```

(2) 模板举例

5.最长回文子串-hot100

28.KMP算法

(1) 模板说明

1. **KMP算法用来判断一个字符串 `p` 是否为另一个字符串 `s` 的子字符串的算法，他是对暴力算法的加速**
2. 字符串 `p` 中字符 `p[i]` 的前缀： `p[0..j]` ， `j<i-1` ， `abcdef` 中 `f` 的前缀为 `a、ab、abc、abcd`
3. 字符串 `p` 中字符 `p[i]` 的后缀： `p[k..i-1]` ， `k>0` ， `abcdef` 中 `f` 的后缀为 `e、de、cde、bcde`
4. 前缀和后缀的最大匹配长度：使得 `p[0..j]==p[k..i-1]` 的最大长度，求出 `p` 每个位置的最长匹配长度放到 `nexts` 数组中， `nexts[0]=-1、nexts[1]=0`
5. 暴力解法 `O(N*M)` ：尝试从 `s` （长度M）的每个位置 `i` 作为开始位置判断是否和 `p` （长度N）相等；当 `s` 匹配到 `x` 位置， `p` 匹配到 `y` 位置时发现不匹配， `s` 跳回到 `i+1` 位置， `p` 跳回到 `0` 位置，重新开始匹配
6. KMP的过程是，当发现不匹配时， `s` 在 `x` 位置不动， `p` 跳回到 `y` 的最长匹配前缀的下一个字符串处（ `y=nexts[y]` ），继续开始匹配。根据最长匹配长度的定义知， `p[0..nexts[y]-1]` 与 `s[..x-1]` 是匹配的，无需验证
7. 核心代码：

```
// 判断p是否为s的子串，是，返回起始位置；不是，返回-1
int kmp(const string& s,const string& p){
    if(s.size()==0||p.size()==0||s.size()<p.size()){
        return -1;
    }
    vector<int>nexts=getNexts(p);// 获取最长前缀匹配数组
    int i=0;// s
    int j=0;// p
    while(i<s.size()&&j<p.size()){
        // 匹配上了，向后推进
        if(s[i]==p[j]){
            i++;
            j++;
        }
        // j已经来到0位置了，无法再往前移动了，i往后移动（同暴力）
        else if(j==0){
            i++;
        }
        // j往回移动，继续与s[i]
        else{
            j=nexts[j];
        }
    }
    return j==p.size()?i-j:-1;
}
```

8. 给出一个字符串p，如何求 `nexts` ？假设求 `nexts[i]` ：令 `j=nexts[i-1]` ，如果 `p[i-1]==p[j]` ， `nexts[i]=j+1` ；如果 `p[i-1]!=p[j]` ， `j=nexts[j]` ，继续上述过程，直到 `p[i-1]==p[j]` ，则， `nexts[i]=j+1` 或者 `j==0` （无法再往前跳了）， `nexts[i]=0` ；核心代码：

```
// 求每个位置的最长匹配长度
// nexts[i]也表示最长匹配长度对应的前缀的下一位置
vector<int> getNexts(const string& p){
    if(p.size()<2){
        return vector<int>({-1});
    }

    vector<int>nexts(p.size());
    nexts[0]=-1;
    nexts[1]=0;

    // 如果i-1位置的最长前缀匹配长度对应的下一个位置的字符等于p[i-1]，那么nexts[i]=nexts[i-1]+1
    int i=2;// 求nexts[i]
    int j=0;// 与i-1位置字符比较的位置
    while(i<p.size()){
        if(p[i-1]==p[j]){
            nexts[i]=j+1;
            i++;
            j=nexts[i-1];// j++;
        }else if(j==0){
```

```
            nexts[i]=0;
            i++;
            // j=0;
        }else{
            j=nexts[j];
        }
    }
    return nexts;
}
```

(2) 模板举例

28.找出字符串中第一个匹配项的下标

就是上述算法

29.求等长有序数组的上中位数

(1) 模板说明

1. 给定两个等长，并且递增的数组，求两个数组的上中位数。上中位数为排完序之后中间两个的前一个
2. 也算是二分法
3. 核心代码：`O(logN)`

```
// 求等长有序数组的上中位数
// l、r为nums1所求上中位数的左右边界
// x、y为nums2所求上中位数的左右边界
int getMedian(vector<int>& nums1,int l,int r,vector<int>& nums2,int x,int y){
    if(l==r){// 防止m-1越界
        return nums1[l]<nums2[x]?nums1[l]:nums2[x];
    }
    int m1=l+((r-l)>>1);
    int m2=x+((y-x)>>1);
    if(nums1[m1]==nums2[m2]){// 中间值相等，直接返回，奇偶情况相同
        return nums1[m1];
    }

    // 长度为偶数：
    //  中间值小的上半区间中的元素可能是上中位数（不包含中间值）
    //  中间值大的下半区间中的元素可能是上中位数（包含中间值）
    if((r-l+1)%2==0){
        if(nums1[m1]>nums2[m2]){
            return getMedian(nums1,l,m1,nums2,m2+1,y);
        }else{
            return getMedian(nums1,m1+1,r,nums2,x,m2);
        }
    }

    // 长度为奇数：
    //  中间值小的上半区间中的元素可能是上中位数（包含中间值）
    //  中间值大的下半区间中的元素可能是上中位数（不包含中间值）
    //  所确定的区间长度不等，因此需要单独判断所包含的中间值是否为上中位数
    //  如果是，直接返回；否则舍弃掉，区间长度相等，调用递归
    else{
        if(nums1[m1]>nums2[m2]){
            if(nums2[m2]>=nums1[m1-1]){
                return nums2[m2];
            }else{
                return getMedian(nums1,l,m1-1,nums2,m2+1,y);
            }
        }else{
            if(nums1[m1]>=nums2[m2-1]){
                return nums1[m1];
            }else{
                return getMedian(nums1,m1+1,r,nums2,x,m2-1);
            }
        }
```

```
    }

    return 0;
}
```

4. 根据求上中位数的方法可以求：两个有序不等长数组的第k（从1开始）小的数，核心代码：`O(log(min(M,N))`

```
// 求第k小的数
int kthMin(vector<int>& nums1,vector<int>& nums2,int k){
    // 始终让nums1为长数组
    if(nums1.size()<nums2.size()){
        nums1.swap(nums2);
    }
    int m=nums1.size();
    int n=nums2.size();
    if(k<1||k>m+n){
        return INT_MIN;
    }
    if(n==0){
        return nums1[k-1];
    }

    // k<=n：
    //  直接求两数组前k元素的上中位数
    if(k<=n){
        return getMedian(nums1,0,m-1,nums2,0,n-1);
    }

    // k>m：
    //  nums1：[k-n-1,m-1]可能为第k小
    //  nums2：[k-m-1,n-1]可能为第k小
    //  需要先判断nums1[k-n-1]和nums2[k-m-1]是否为第k小，否则最后求得的是第k-1小，具体举例分析
    if(k>m){
        int i=k-n-1;
        int j=k-m-1;
        if(nums1[i]>=nums2[n-1]){
            return nums1[i];
        }else if(nums2[j]>=nums1[m-1]){
            return nums2[j];
        }else{
            return getMedian(nums1,i+1,m-1,nums2,j+1,n-1);
        }
    }

    // n<k<=m：
    //  nums1：[k-n-1,k-1]可能为第k小
    //  nums2：[0,n-1]可能为第k小
    //  区间[k-n-1,k-1]的长度比[0,n-1]大1，可以先判断nums1[k-n-1]是否为第k小，不是，再掉递归
    else{
        int i=k-n-1;
        int j=k-1;
        if(nums1[i]>=nums2[n-1]){
            return nums1[i];
        }else{
            return getMedian(nums1,i+1,j,nums2,0,n-1);
        }
    }

    return INT_MIN;
}
```

(2) 模板举例

4.寻找两个正序数组的中位数-hot100

```
// 本题只需要调用上述函数即可得出答案
double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2){
    int m=nums1.size();
    int n=nums2.size();
```

```
    int res=kthMin(nums1,nums2,(m+n)/2+1);
    if((m+n)%2==1){return res;}
    res+=kthMin(nums1,nums2,(m+n)/2);
    return res/2.0;
}
```

30.排序

(1) 选择排序

```
// 前面是已排序部分，每次在未排序部分选择最小的放到已排序的后面
void sort(vector<int>& nums){
    if(nums.size()<2){
        return;
    }
    for(int i=0;i<nums.size();i++){
        int idx=i;
        for(int j=i+1;j<nums.size();j++){
            if(nums[j]<nums[idx]){
                idx=j;
            }
        }
        swap(nums[i],nums[idx]);
    }
}
```

(2) 冒泡排序

```
// 后面是已排序部分，每次比较未排序部分相邻的两个数，将大的放右边
void sort(vector<int>& nums){
    if(nums.size()<2){
        return;
    }
    for(int i=nums.size()-1;i>0;i--){
        for(int j=0;j<i;j++){
            if(nums[j]>nums[j+1]){
                swap(nums[j],nums[j+1]);
            }
        }
    }
}
```

(3) 插入排序

```
// 前面是已排序部分，后面是未排序部分，每次将未排序部分的第一个数与前面已排序部分的数比较，较小则交换，直至前面的数都比其小或者已经交换到了最左边
void sort(vector<int>& nums){
    if(nums.size()<2){
        return;
    }
    for(int i=1;i<nums.size();i++){
        for(int j=i-1;j>=0&&nums[j]>nums[j+1];j--){
            swap(nums[j],nums[j+1]);
        }
    }
}
```

(4) 归并排序

```
// 左边排好序，右边排好序，然后再合并
void sort(vector<int>& nums){
    if(nums.size()<2){
        return;
    }
    help(nums,0,nums.size()-1);
```

```
}

void help(vector<int>& nums,int l,int r){
    if(l==r){
        return;
    }
    int m=l+((r-l)>>1);
    help(nums,l,m);
    help(nums,m+1,r);
    merge(nums,l,m,r);
}
void merge(vector<int>& nums,int l,int m,int r){
    vector<int>tmp(r-l+1);
    int p1=l;
    int p2=m+1;
    int i=0;
    while(p1<=m&&p2<=r){
        tmp[i++]=nums[p1]<nums[p2]?nums[p1++]:nums[p2++];
    }
    while(p1<=m){
        tmp[i++]=nums[p1++];
    }
    while(p2<=r){
        tmp[i++]=nums[p2++];
    }
    for(i=0;i<tmp.size();i++){
        nums[i+l]=tmp[i];
    }
}
```

归并排序相关题目：

21.合并两个有序链表&面试题25-hot100

23.合并K个升序链表-hot100

面试题51：数组中的逆序对

315.计算右侧小于当前元素的个数

上题只要求逆序对的数量，本题需要求出各个各个位置的逆序对的数量，排序后元素位置会发生改变，所以用一个`vector<pair<int,int>>`来处理原数组

(5) 快速排序

```
// 荷兰国旗问题
void sort(vector<int>& nums){
    if(nums.size()<2){
        return;
    }
    help(nums,0,nums.size()-1);
}
void help(vector<int>& nums,int l,int r){
    if(l==r){
        return;
    }
    srand((unsigned)time(NULL));
    swap(nums[r],nums[rand()%(r-l+1)+l]);
    vector<int>p=partion(nums,l,r);
    help(nums,l,p[0]-1);
    help(nums,p[1]+1,r);
}

vector<int> partion(vector<int>& nums,int l,int r){
    int t=nums[r];
    int less=l-1;
    int more=r;
    int i=l;
    while(i<more){
        if(nums[i]<t){
            swap(nums[i],nums[less+1]);
```

```
            less++;
            i++;
        }else if(nums[i]==t){
            i++;
        }else{
            swap(nums[i],nums[more-1]);
            more--;
        }
    }
    swap(nums[r],nums[more]);
    return vector<int>({l+1,more});
}
```

快排的partition可以用O(N)的时间复杂度，找到数组中第k大/小的元素

partition相关题目：

面试题40：最小的k个数

215.数组中第K个最大元素-hot100

75.颜色分类-hot100

347.前K个高频元素-hot100

86.分隔链表

面试题21：调整数组顺序使奇数位于偶数前面

283.移动零-hot100

(6) 堆排序

1. 堆在逻辑上是一个完全二叉树（二叉树从左往右依次变满），可以用数组表示
2. 将完全二叉树的节点与数组下标 `0..len-1` 依次对应：`0` 为根节点，根节点的父节点为自己；`i` 节点的父节点为 `(i-1)/2`；`i` 的左右孩子分别为 `2*i+1` 和 `2*i+2`
3. 手写一个大根堆：

```
class BigHeap {
private:
    vector<int>nums;
    int capacity;   // 容量
    int size;       // 已存元素数量
public:
    // 构造函数
    BigHeap(int cpc=10) :capacity(cpc), size(0) {
        nums.resize(cpc);
    }
    BigHeap(vector<int>& arr){
        nums.resize(arr.size());
        capacity=arr.size();
        size=0;
        for(int i=0;i<arr.size();i++){
            push(arr[i]);
        }
    }

    // 向堆中插入一个元素
    void push(int num) {
        if (size == capacity) {// 扩容
            nums.resize(2 * capacity);
            capacity = 2 * capacity;
        }
        int i = size;
        size++;
        nums[i] = num;
        while (nums[i] > nums[(i - 1) / 2]) {
            swap(nums[i], nums[(i - 1) / 2]);
            i = (i - 1) / 2;
```

```
        }
    }

    // 返回堆顶元素
    int top() {
        if (size == 0) {
            return INT_MIN;
        }
        return nums[0];
    }

    // 弹出堆顶元素
    void pop() {
        if (size == 0) {
            return;
        }
        swap(nums[0], nums[size - 1]);
        size--;
        heapify(0);
    }

private:
    // 从i位置开始堆化
    void heapify(int i) {
        while (2 * i + 1 < size) {
            if (2 * i + 2 < size) {
                int idx = nums[2 * i + 1] > nums[2 * i + 2] ? 2 * i + 1 : 2 * i + 2;
                if (nums[idx] > nums[i]) {
                    swap(nums[idx], nums[i]);
                    i = idx;
                }
                else {
                    break;
                }
            }
            else {
                if (nums[2 * i + 1] > nums[i]) {
                    swap(nums[2 * i + 1], nums[i]);
                    i = 2 * i + 1;
                }
                else {
                    break;
                }
            }
        }
    }
};
```

31.设计数据结构

面试题41：数据流中的中位数

295.数据率的中位数

面试题9：用两个栈实现队列

相关题目：leetcode 225

208.实现前缀树Trie-hot100

155.最小栈&面试题30-hot100

146.LRU缓存-hot100

460.LFU(Least Frequently Used，最不常使用)

380.O(1)时间插入、删除和获取随即元素

710.黑名单中的随机数

224.基本计算器

227.基本计算器II

770基本计算器IV

295.数据率的中位数

528.按权重随机选择

32.位运算

(1) 模板说明

```
1. 异或运算/不进位加法
    a.0^N=N; N^N=0
    b.满足交换律和结合率：a^b=b^a; (a^b)^c=a^(b^c)
    c.一堆数进行异或可以随意排列异或的顺序

2. n=(n&(n-1));// 消掉最后一位的1

3. n=x&(~x+1);// 提取x最右侧的1，结果n的对应位为1，其余位全为0
```

(2) 模板举例

面试题56-I：数组中数字出现的次数

136.只出现一次的数字-hot100

面试题56-II：数组中数字出现的次数II

面试题15：二进制中1的个数

461.汉明距离

338.比特位计数-hot100

面试题65：不用加减乘除做加法

232.用栈实现队列

190.颠倒二进制位

33.其他

面试题64：求1+2+..+n

面试题62：圆圈中最后剩下的数字

面试题44：数字序列中的某一位的数字

面试题43：1~n整数中1出现的次数

面试题31：栈的压入、弹出序列

面试题16：数值的整数次方

621.任务调度器

394.字符串解码-hot100

384.打乱数组

382.链表随机节点

398.随机数索引

754.到达终点数字

738.单调递增的数字

906超级回文数

696.计数二进制子串

149.直线上最多的点数

769.最多能完成排序的块

342.4的幂

326.3的幂

504.七进制数

204.计数质数

69.x 的平方根

13.罗马数字转整数

9.回文数

7.整数反转

三、知识点总结

1.priority_queue

(1) 基础知识

```
// 1.头文件
#include <queue>     //优先队列

// 2.用法
priority_queue< type, container, function >
    type：数据类型
    container：实现优先队列的底层容器，必须是数组形式的实现容器，如vector、deque，不能是list
    function：元素之间的比较方式
    默认情况下（省略后面两个参数）是以vector为容器，以 operator< 为比较方式，所以在只使用第一个参数时，优先队列默认是一个最大堆，每次输出的堆顶元素是此时堆中的最大元素

// 3.成员函数
    empty();
    size();
    pop();
    top();
    push();

// 4.大根堆
    priority_queue<int> big_heap;
    priority_queue<int,vector<int>,less<int> > big_heap2;

// 5.小根堆
    priority_queue<int, vector<int>, greater<int> > small_heap;
    注意使用less<int>和greater<int>，需要头文件functional（也可以没有）
```

(2) 自定义排序规则

1. 在类中重载 `<`和`>`，注意需要两个const

```
class T{
public:
    bool operator>(const T& b) const{
        return ...
    }
    bool operator<(const T& b) const{
        return ...
    }
};
priority_queue<T,vector<T>,greater<T>>q;     // 小根堆
priority_queue<T,vector<T>,less<T>>q;        // 大根堆
```

2. 使用函数对象—最推荐使用

```
class mycom{
public:
    bool operator()(const T& a,const T& b){// 此处的const可以不要，最好加上
        return ...
    }
};
priority_queue<T,vector<T>,mycom>q;
```

3. lambda表达式

```
auto mycom=[](const T& a,const T& b){
    return ...
};
priority_queue<T,vector<T>,decltype(mycom)>q(mycom);
```

4. 函数指针

```
bool mycom(const T& a,const T& b){
    return ...
}
priority_queue<T,vector<int>,decltype(&mycom)>q(&mycmp);// 第一个&必须要，第二个可以不要
```

2.sort()

(1) 基础知识

```
// 1.头文件
#include <algorithm>

// 2.用法
sort(iterator beg, iterator end, _Pred)
    适用对象：支持随机访问的容器，即只支持序列式容器（vector，deque，array）
    排序范围：左闭右开，即 [ )
    默认从小到大排序less<T>，从大到小排序greater<T>
    不稳定的排序，基于快排实现
```

(2) 自定义排序规则

1. 函数指针-普通函数

   比较函数compare要声明为静态成员函数或全局函数，不能作为普通成员函数，即不可以写在类中；如果写在类中，也需要加static关键字，否则会报错，原因如下：

   > 1）非静态成员函数只能由具体的类对象来调用；2）std::sort这类函数是全局的，因此无法在sort中调用非静态成员函数。
   > 静态成员函数或者全局函数是不依赖于具体对象的，可以独立访问，无须创建任何对象实例就可以访问。同时，静态成员函数不可以调用类的非静态成员。

   也可以使用lambda表达式

2. 利用仿函数（函数对象）

   仿函数即为重载了()运算符的类，注意，用的时候要创建临时对象，而不能直接使用类名

3. 重载关系运算符

当关联式容器中存储的数据类型为自定义的结构体变量或者类对象时，通过对现有排序规则中所用的关系运算符进行重载，也能实现自定义排序规则的目的。

3.max/min_element

```
1. 头文件：include<algorithm>
2. max/min_element(beg,ed,pred)
    beg和ed可以是vector的迭代器，也可以是普通数组的指针
    pred：仿函数，可以省略
    返回max/min第一次出现的迭代器/指针
```

4.lower/upper_bound

```
#include <algorithm>
//在[first, last)区域内查找第一个大于等于val的元素
ForwardIterator lower_bound (ForwardIterator first, ForwardIterator last,
                             const T& val);
//在[first, last)区域内查找第一个不符合comp规则的元素
ForwardIterator lower_bound (ForwardIterator first, ForwardIterator last,
                             const T& val, Compare comp);
1.输入参数可以指针/迭代器（正向），输出参数对应
2.底层通过二分查找实现，因此要求符合条件的都在数组的左侧，不符合条件的都在数组的右侧（升序只是符合条件的一种情况）
3.comp可以是仿函数/函数指针

upper_bound与lower_bound类似，只不过返回的是第一个大于val的元素
```

四、算法二刷

14.DFS/回溯算法

(1) 模板说明

1. 回溯算法就是个多叉树（图）的遍历问题，关键就是在前序遍历和后序遍历的位置做一些操作

```
vector<vector<int>> mainfunc(参数){
    vector<vector<int>>res;
    vector<int>cur;
    // vector<bool>visited;// 有时需要
    dfs(res,cur,其他参数);

    return res;
}

void dfs(vector<vector<int>>& res,vector<int>& cur,其他参数){
    if(终止条件){
        res.push_back(cur);
        return;
    }

    for(next:nexts[i]){
        处理next;              // 先序遍历的位置
        dfs(res,cur,其他参数);
        回溯，撤销处理结果;       // 后序遍历的位置
    }
}
```

(2) 未通过

90.子集II

给你一个整数数组 nums ，其中可能包含重复元素，请你返回该数组所有可能的子集（幂集）。

解集 不能 包含重复的子集。返回的解集中，子集可以按 任意顺序 排列。

```
输入：nums = [1,2,2]
输出：[[],[1],[1,2],[1,2,2],[2],[2,2]]
```

```cpp
// 与78题的区别在于，本题给出的数组可能包含重复元素，因此要想办法去重
// 方法一是用一个unordered_set去重，当不要一个元素之后，后面相同的元素都不能要了，都需要跳过
vector<vector<int>> subsetsWithDup(vector<int>& nums) {
    vector<vector<int>>res;
    vector<int>cur;
    unordered_set<int>st;// 记录不要的元素，如果不要，后面相同的元素都不能要
    dfs(nums,res,cur,st,0);
    return res;
}

void dfs(vector<int>& nums,vector<vector<int>>&res,vector<int>&cur,unordered_set<int>&st,int i){
    if(i==nums.size()){
        res.push_back(cur);
        return;
    }

    // 可要可不要
    if(st.find(nums[i])==st.end()){
        // 要
        cur.push_back(nums[i]);
        dfs(nums,res,cur,st,i+1);
        cur.pop_back();

        // 不要
        st.insert(nums[i]);
        dfs(nums,res,cur,st,i+1);
        st.erase(nums[i]);
    }
    // 只能不要
    else{
        dfs(nums,res,cur,st,i+1);
    }
}

// 方法二是通过预排序来去重，每层相同元素只能取一个，取第一个
vector<vector<int>> subsetsWithDup(vector<int>& nums) {
    sort(nums.begin(),nums.end());

    vector<vector<int>>res;
    vector<int>cur;
    dfs(nums,res,cur,0);

    return res;
}

void dfs(vector<int>& nums,vector<vector<int>>&res,vector<int>&cur,int i){
    res.push_back(cur);
    if(i==nums.size()){
        return;
    }

    for(int j=i;j<nums.size();j++){
        if(j>i&&nums[j]==nums[j-1]){
            continue;
        }
        cur.push_back(nums[j]);
        dfs(nums,res,cur,j+1);
        cur.pop_back();
    }

}
```

47.全排列II

给定一个可包含重复数字的序列 `nums` ，***按任意顺序***返回所有不重复的全排列。

```
输入：nums = [1,1,2]
输出：
[[1,1,2],
 [1,2,1],
 [2,1,1]]
```

```cpp
// 方法一：当前位置要不要，不要从后面交换过来一个，需要保证交换过来的数之前没有在给位置出现过
vector<vector<int>> permuteUnique(vector<int>& nums) {
    vector<vector<int>>res;
    vector<int>cur;
    dfs(nums,res,cur,0);

    return res;
}

void dfs(vector<int>& nums,vector<vector<int>>& res,vector<int>& cur,int i){
    if(i==nums.size()){
        res.push_back(cur);
        return;
    }
    unordered_set<int>st;
    for(int j=i;j<nums.size();j++){
        if(st.find(nums[j])!=st.end()){
            continue;
        }
        st.insert(nums[j]);
        swap(nums[i],nums[j]);
        cur.push_back(nums[i]);
        dfs(nums,res,cur,i+1);
        cur.pop_back();
        swap(nums[i],nums[j]);
    }
}

// 方法二：决策树遍历，当前层不能有重复的数，cur中用过的数（以位置为标识）也不能再用
vector<vector<int>> permuteUnique(vector<int>& nums) {
    sort(nums.begin(),nums.end());
    vector<vector<int>>res;
    vector<int>cur;
    unordered_set<int>st;
    dfs(nums,res,cur,st,0);

    return res;
}

void dfs(vector<int>& nums,vector<vector<int>>& res,vector<int>& cur,unordered_set<int>& st,int i){
    if(i==nums.size()){
        res.push_back(cur);
        return;
    }

    for(int j=0;j<nums.size();j++){
        if(st.find(j)!=st.end()){// 用过的数不能再用
            continue;
        }
        if((j>0&&nums[j]==nums[j-1]&&st.find(j-1)!=st.end())){// 同一层不能有重复的，需要保证之前用过
            continue;
        }

        st.insert(j);
        cur.push_back(nums[j]);
        dfs(nums,res,cur,st,i+1);
        cur.pop_back();
        st.erase(j);
    }
}
```

面试题38：字符串的排列

本题与上题相似，知识将数组换成了字符串

698.划分为k个不相等的子集

给定一个整数数组 `nums` 和一个正整数 `k`，找出是否有可能把这个数组分成 `k` 个非空子集，其总和都相等。

```
输入： nums = [4, 3, 2, 3, 5, 2, 1], k = 4
输出： True
说明： 有可能将其分成 4 个子集（5），（1,4），（2,3），（2,3）等于总和。
```

```
// 方法一：有k个子集（桶），现在是其中某个桶做选择，选择i位置的数要不要
bool canPartitionKSubsets(vector<int>& nums, int k) {
    if(k<1||k>nums.size()){                         // k不符合要求
        return false;
    }
    int len=nums.size();
    int maxv=INT_MIN;
    int sumv=0;
    for(int i=0;i<len;i++){
        sumv+=nums[i];
        maxv=max(maxv,nums[i]);
    }
    if(sumv%k!=0){                                  // 不能整除，则一定为false
        return false;
    }
    int avg=sumv/k;
    if(avg<maxv){                                   // 最大值无处安放
        return false;
    }
    sort(nums.begin(),nums.end(),greater<int>());   // 必须排序
    vector<bool>used(len);                          // 存放之前用过的数

    return dfs(nums,used,k,0,0,avg);
}


// 现在考虑第k个子集
// 现在来到nums的i位置，第k个子集要不要nums[i]?
// cursum：第k个子集目前的和
// target：目标和
// used：之前已经用过的数
bool dfs(vector<int>& nums,vector<bool>& used,int k,int i,int cursum,int target){
    if(k==0){
        return true;
    }
    if(i==nums.size()){
        if(cursum==target){
            return dfs(nums,used,k-1,0,0,target);
        }else{
            return false;
        }
    }

    // 只能不要
    if(
        (i>0&&nums[i]==nums[i-1]&&!used[i-1])   // 前一个相等的数没要，自己也不能要，否则就和前面要的情况重复了
        ||(used[i])                             // 当前元素以及被用了
        ||(nums[i]>target-cursum)               // 剪枝
    )
    {
        return dfs(nums,used,k,i+1,cursum,target);
    }

    // 要
    used[i]=true;
    if(dfs(nums,used,k,i+1,cursum+nums[i],target)){
        return true;
```

```
    }
    used[i]=false;
    // 不要
    return dfs(nums,used,k,i+1,cursum,target);
}
```

```
// 有k个子集（桶），现在来到nums[i]位置，选择进入k个桶的哪一个
bool canPartitionKSubsets(vector<int>& nums, int k) {
    if(k<1||k>nums.size()){                         // k不符合要求
        return false;
    }
    int len=nums.size();
    int maxv=INT_MIN;
    int sumv=0;
    for(int i=0;i<len;i++){
        sumv+=nums[i];
        maxv=max(maxv,nums[i]);
    }
    if(sumv%k!=0){                                  // 不能整除，则一定为false
        return false;
    }
    int avg=sumv/k;
    if(avg<maxv){                                   // 最大值无处安放
        return false;
    }
    sort(nums.begin(),nums.end(),greater<int>());   // 必须排序，便于剪枝，否则会超时
    vector<int>buckets(k);

    return dfs(nums,buckets,0,avg);
}

// k个桶，其中存的是该自己的和
// 现在来到nums的i位置，选择进入哪个桶
// target：目标和
bool dfs(vector<int>& nums,vector<int>& buckets,int i,int target){
    if(i==nums.size()){
        // 检验所有桶的和是否为target
        // 下面代码可以省略，因为进桶之前会检查，i能来到最后，说明所有同都没超
        // for(int j=0;j<buckets.size();j++){
        //     if(buckets[j]!=target){
        //         return false;
        //     }
        // }
        return true;
    }

    for(int j=0;j<buckets.size();j++){
        if(j>0&&buckets[j]==buckets[j-1]){
            continue;
        }
        if(buckets[j]+nums[i]<=target){
            buckets[j]+=nums[i];
            if(dfs(nums,buckets,i+1,target)){
                return true;
            }
            buckets[j]-=nums[i];
        }
    }
    return false;
}
```

51.N皇后

按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。

n 皇后问题 研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此之间不能相互攻击。

给你一个整数 n ，返回所有不同的 n 皇后问题 的解决方案。

每一种解法包含一个不同的 n 皇后问题 的棋子放置方案，该方案中 'Q' 和 '.' 分别代表了皇后和空位。

```cpp
// 方法一：回溯
bool isvaild(vector<string>& cur, int n, int i, int j) {
    for (int k = 0; k < cur.size(); k++) {
        if (cur[k][j] == 'Q'||(k - i + j>=0&& k - i + j <n&&cur[k][k - i + j]=='Q')||(i-k  + j>=0&& i - k
+ j<n&&cur[k][i-k+j]=='Q')) {
            return false;
        }
    }
    return true;
}
void dfs(vector<vector<string>>& res, vector<string>& cur, int n, int i) {
    if (i == n) {
        res.push_back(cur);
        return;
    }
    for (int j = 0; j < n; j++) {
        if (isvaild(cur, n, i, j)) {
            string t(n, '.');
            t[j] = 'Q';
            cur.push_back(t);
            dfs(res, cur, n, i + 1);
            cur.pop_back();
        }
    }
}
vector<vector<string>> solveNQueens(int n) {
    if (n < 1) {
        return vector<vector<string>>();
    }
    if (n == 1) {
        vector<vector<string>>(1, vector<string>(1, "Q"));
    }

    vector<vector<string>>res;
    vector<string>cur;
    dfs(res, cur, n, 0);
    return res;
}

// 方法二：用位运算加速回溯
void process(vector<vector<string>>& res, vector<string>& broad,int limit,int i,int col,int left,int right)
{
    if(col==limit){
        res.push_back(broad);
        return;
    }
    int pos=limit&(~(col|right|left));
    while(pos!=0){
        int mostright1=pos&((~pos)+1);
        pos-=mostright1;
        broad[i][log2(mostright1)]='Q';
        process(res,broad,limit,i+1,col|mostright1,(left|mostright1)<<1,(right|mostright1)>>1);
        broad[i][log2(mostright1)]='.';
    }

}
vector<vector<string>> solveNQueens(int n) {
    vector<vector<string>>res;
    if(n<1){
        return res;
    }
    vector<string> broad(n,string(n,'.'));
    int limit=n==32?-1:(1<<n)-1;
    process(res,broad,limit,0,0,0,0);
    return res;
}
```

面试题17：打印从1到最大的n位数

输入数字 `n`，按顺序打印出从 1 到最大的 n 位十进制数。比如输入 3，则打印出 1、2、3 一直到最大的 3 位数 999。

```
// 不考虑大数的情况
vector<int> printNumbers(int n) {
    n=pow(10,n);
    vector<int>res(n-1);
    for(int i=1;i<n;i++){
        res[i-1]=i;
    }
    return res;
}

// 考虑大数的情况
// 用回溯法
vector<int> printNumbers(int n) {
    vector<string>res_s=printNumbers_s(n);

    vector<int>res(res_s.size());
    for(int i=0;i<res.size();i++){
        res[i]=stoi(res_s[i]);
    }
    return res;
}

// 考虑n非常大，可能会超出int表示的范围，用字符串作为输出
vector<string> printNumbers_s(int n){
    vector<string>res;

    // 分别求位数为i的数
    for(int i=1;i<=n;i++){
        string cur="";
        dfs(res,cur,0,i);
    }
    return res;
}
// len：当前长度限制
// x：目前的长度
void dfs(vector<string>& res,string& cur,int x,int len){
    if(x==len){
        res.push_back(cur);
        return;
    }
    int start=x==0?1:0;
    for(int i=start;i<=9;i++){
        cur+=('0'+i);
        dfs(res,cur,x+1,len);
        cur.pop_back();
    }
}
```

(3) 通过

78.子集-hot100

给你一个整数数组 `nums` ，数组中的元素 **互不相同** 。返回该数组所有可能的子集（幂集）。

解集 **不能** 包含重复的子集。你可以按 **任意顺序** 返回解集。

```
输入：nums = [1,2,3]
输出：[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
```

```
// 两种思维方式：
// 方法一：当前位置要还是不要，两种选择
vector<vector<int>> subsets(vector<int>& nums) {
    vector<vector<int>>res;
```

```
    vector<int>cur;
    dfs(nums,res,cur,0);
    return res;
}

void dfs(vector<int>& nums,vector<vector<int>>& res,vector<int>& cur,int i){
    if(i==nums.size()){
        res.push_back(cur);
        return;
    }

    // 不要
    dfs(nums,res,cur,i+1);

    // 要
    cur.push_back(nums[i]);
    dfs(nums,res,cur,i+1);
    cur.pop_back();
}

// 方法二：画出决策树，树上的每个节点都是一个可行解
vector<vector<int>> subsets(vector<int>& nums) {
    vector<vector<int>>res;
    vector<int>cur;
    dfs(nums,res,cur,0);
    return res;
}

void dfs(vector<int>& nums,vector<vector<int>>& res,vector<int>& cur,int i){
    res.push_back(cur);
    if(i==nums.size()){
        return;
    }
    for(int j=i;j<nums.size();j++){
        cur.push_back(nums[j]);
        dfs(nums,res,cur,j+1);
        cur.pop_back();
    }
}
```

77.组合

给定两个整数 `n` 和 `k`，返回范围 `[1, n]` 中所有可能的 `k` 个数的组合。你可以按 **任何顺序** 返回答案。

```
输入：n = 4, k = 2
输出：
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

```
// 还是两种思维方式：
// 方法一：当前位置要不要
vector<vector<int>> combine(int n, int k) {
    if(k>n||k<1||n<1){
        return vector<vector<int>>();
    }
    vector<vector<int>>res;
    vector<int>cur;
    dfs(n,k,res,cur,1);
    return res;
}

void dfs(int n,int k,vector<vector<int>>& res,vector<int>& cur,int i){
    if(cur.size()==k){
```

```
            res.push_back(cur);
            return;
        }
        if(k-cur.size()>n-i+1){
            return;
        }
        // 要
        cur.push_back(i);
        dfs(n,k,res,cur,i+1);
        cur.pop_back();

        // 不要
        dfs(n,k,res,cur,i+1);
    }

    // 方法二：画出决策树，多叉树遍历
    vector<vector<int>> combine(int n, int k) {
        if(k>n||k<1||n<1){
            return vector<vector<int>>();
        }
        vector<vector<int>>res;
        vector<int>cur;
        dfs(n,k,res,cur,1);
        return res;
    }

    void dfs(int n,int k,vector<vector<int>>& res,vector<int>& cur,int i){
        if(cur.size()==k){
            res.push_back(cur);
            return;
        }
        if(k-cur.size()>n-i+1){
            return;
        }

        for(int j=i;j<=n;j++){
            cur.push_back(j);
            dfs(n,k,res,cur,j+1);
            cur.pop_back();
        }
    }
```

39.组合总和-hot100

给你一个 无重复元素 的整数数组 candidates 和一个目标整数 target ， 找出 candidates 中可以使数字和为目标数 target 的 所有 不同组合 ， 并以列表形式返回。你可以按 任意顺序 返回这些组合。

candidates 中的 同一个 数字可以 无限制重复被选取 。如果至少一个数字的被选数量不同，则两种组合是不同的。

对于给定的输入，保证和为 target 的不同组合数少于 150 个。

```
输入：candidates = [2,3,6,7], target = 7
输出：[[2,2,3],[7]]
解释：
2 和 3 可以形成一组候选，2 + 2 + 3 = 7 。注意 2 可以使用多次。
7 也是一个候选， 7 = 7 。
仅有这两种组合。

提示：
1 <= candidates.length <= 30
2 <= candidates[i] <= 40
candidates 的所有元素 互不相同
1 <= target <= 40
```

```
    // 还是两种思维方式：本题也可以通过预排序，加速剪枝
    // 方法一：当前位置要不要
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        sort(candidates.begin(),candidates.end());
        vector<vector<int>>res;
```

```
    vector<int>cur;
    dfs(candidates,res,cur,0,target);
    return res;
}

void dfs(vector<int>& candidates,vector<vector<int>>& res,vector<int>& cur,int i,int target){
    if(target==0){
        res.push_back(cur);
        return;
    }
    if(i==candidates.size()||target<candidates[i]){
        return;
    }
    // 不要
    dfs(candidates,res,cur,i+1,target);

    // 要
    cur.push_back(candidates[i]);
    dfs(candidates,res,cur,i,target-candidates[i]);// 因为可以重复选取一个数，所以这里的i不能加1
    cur.pop_back();
}

// 方法二：决策树遍历
vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
    sort(candidates.begin(),candidates.end());
    vector<vector<int>>res;
    vector<int>cur;
    dfs(candidates,res,cur,0,target);
    return res;
}

void dfs(vector<int>& candidates,vector<vector<int>>& res,vector<int>& cur,int i,int target){
    if(target==0){
        res.push_back(cur);
        return;
    }
    if(i==candidates.size()||target<candidates[i]){
        return;
    }

    for(int j=i;j<candidates.size();j++){
        cur.push_back(candidates[j]);
        dfs(candidates,res,cur,j,target-candidates[j]);
        cur.pop_back();
    }
}
```

40.组合总和II

给定一个候选人编号的集合 candidates 和一个目标数 target ， 找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的每个数字在每个组合中只能使用 一次 。

注意：解集不能包含重复的组合。

```
输入: candidates = [10,1,2,7,6,1,5], target = 8,
输出:
[
[1,1,6],
[1,2,5],
[1,7],
[2,6]
]

提示:
1 <= candidates.length <= 100
1 <= candidates[i] <= 50
1 <= target <= 30
```

```cpp
// 还是两种思维方式：
// 方法一：要还是不要，可以用unordered_set来去重，也可以通过预排序来去重（效率更好）
// 用unordered_set
vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
    vector<vector<int>>res;
    vector<int>cur;
    unordered_set<int>st;// 之前不要的数

    dfs(candidates,res,cur,st,0,target);
    return res;
}

void dfs(vector<int>& candidates,vector<vector<int>>& res,vector<int>& cur,unordered_set<int>& st,int i,int
target){
    if(target==0){
        res.push_back(cur);
        return;
    }
    if(i==candidates.size()||target<0){
        return;
    }
    // 只能不要
    if(st.find(candidates[i])!=st.end()){
        dfs(candidates,res,cur,st,i+1,target);
    }
    // 可要可不要
    else{
        // 要
        cur.push_back(candidates[i]);
        dfs(candidates,res,cur,st,i+1,target-candidates[i]);
        cur.pop_back();

        // 不要
        st.insert(candidates[i]);
        dfs(candidates,res,cur,st,i+1,target);
        st.erase(candidates[i]);

    }
}
// 用sort
vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
    sort(candidates.begin(),candidates.end());
    vector<vector<int>>res;
    vector<int>cur;

    dfs(candidates,res,cur,0,target);
    return res;
}

void dfs(vector<int>& candidates,vector<vector<int>>& res,vector<int>& cur,int i,int target){
    if(target==0){
        res.push_back(cur);
        return;
    }
    if(i==candidates.size()||target<candidates[i]){
        return;
    }
    // 要
    cur.push_back(candidates[i]);
    dfs(candidates,res,cur,i+1,target-candidates[i]);
    cur.pop_back();

    // 不要，跳过所有相同的值
    int j=i+1;
    while(j<candidates.size()&&candidates[j]==candidates[i]){
        j++;
    }
    dfs(candidates,res,cur,j,target);

}
```

```
// 方法二：决策树遍历，只能通过排序来去重
vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
    sort(candidates.begin(),candidates.end());
    vector<vector<int>>res;
    vector<int>cur;

    dfs(candidates,res,cur,0,target);
    return res;
}

void dfs(vector<int>& candidates,vector<vector<int>>& res,vector<int>& cur,int i,int target){
    if(target==0){
        res.push_back(cur);
        return;
    }
    if(i==candidates.size()||target<candidates[i]){
        return;
    }

    for(int j=i;j<candidates.size();j++){
        if(j>i&&candidates[j]==candidates[j-1]){
            continue;
        }
        cur.push_back(candidates[j]);
        dfs(candidates,res,cur,j+1,target-candidates[j]);
        cur.pop_back();
    }
}
```

216.组合总和III

找出所有相加之和为 n 的 k 个数的组合，且满足下列条件：

1. 只使用数字1到9
2. 每个数字 最多使用一次

返回 所有可能的有效组合的列表 。该列表不能包含相同的组合两次，组合可以以任何顺序返回。

```
输入: k = 3, n = 7
输出: [[1,2,4]]
解释:
1 + 2 + 4 = 7
没有其他符合的组合了。
```

```
// 两种思维方式：
// 方法一：当前位置要还是不要
vector<vector<int>> combinationSum3(int k, int n) {
    vector<vector<int>>res;
    vector<int>cur;
    dfs(k,n,res,cur,1);
    return res;
}

void dfs(int k,int n,vector<vector<int>>& res,vector<int>& cur,int i){
    if(n==0&&cur.size()==k){
        res.push_back(cur);
        return;
    }
    if(i>9||n<i||k-cur.size()>9-i+1){
        return;
    }
    // 要
    cur.push_back(i);
    dfs(k,n-i,res,cur,i+1);
    cur.pop_back();

    // 不要
    dfs(k,n,res,cur,i+1);
```

```
}
// 方法二：决策树遍历
vector<vector<int>> combinationSum3(int k, int n) {
    vector<vector<int>>res;
    vector<int>cur;
    dfs(k,n,res,cur,1);
    return res;
}

void dfs(int k,int n,vector<vector<int>>& res,vector<int>& cur,int i){
    if(n==0&&cur.size()==k){
        res.push_back(cur);
        return;
    }
    if(i>9||n<i||k-cur.size()>9-i+1){
        return;
    }

    for(int j=i;j<=9;j++){
        cur.push_back(j);
        dfs(k,n-j,res,cur,j+1);
        cur.pop_back();
    }
}
```

46.全排列-hot100

给定一个不含重复数字的数组 `nums` ，返回其 *所有可能的全排列* 。你可以 **按任意顺序** 返回答案。

```
输入：nums = [1,2,3]
输出：[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

```
// 方法一：当前位置要不要，不要从后面交换过来一个
vector<vector<int>> permute(vector<int>& nums) {
    vector<vector<int>>res;
    vector<int>cur;// 可以省略这个数组，用nums来代替，排列问题都可以这么做
    dfs(nums,res,cur,0);
    return res;
}

void dfs(vector<int>& nums,vector<vector<int>>& res,vector<int>& cur,int i){
    if(i==nums.size()){
        res.push_back(cur);
        return;
    }

    for(int j=i;j<nums.size();j++){
        swap(nums[i],nums[j]);
        cur.push_back(nums[i]);
        dfs(nums,res,cur,i+1);
        cur.pop_back();
        swap(nums[i],nums[j]);
    }

}

// 方法二：决策树遍历
vector<vector<int>> permute(vector<int>& nums) {
    vector<vector<int>>res;
    vector<int>cur;
    unordered_set<int>st;// cur中已经存在的数，也可以用一个数组
    dfs(nums,res,cur,st);
    return res;
}

void dfs(vector<int>& nums,vector<vector<int>>& res,vector<int>& cur,unordered_set<int>& st){
    if(st.size()==nums.size()){
        res.push_back(cur);
        return;
```

```
    }

    for(int i=0;i<nums.size();i++){
        if(st.find(nums[i])!=st.end()){
            continue;
        }
        cur.push_back(nums[i]);
        st.insert(nums[i]);
        dfs(nums,res,cur,st);
        st.erase(nums[i]);
        cur.pop_back();
    }
}
```

200.岛屿数量-hot100

给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。

```
int numIslands(vector<vector<char>>& grid) {
    if(grid.size()==0||grid[0].size()==0){
        return 0;
    }
    int res=0;
    int m=grid.size();
    int n=grid[0].size();

    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            if(grid[i][j]=='1'){
                res++;
                dfs(grid,i,j);
            }
        }
    }
    return res;
}
// 淹掉当前岛屿
void dfs(vector<vector<char>>& grid,int i,int j){
    if(i<0||i>=grid.size()||j<0||j>=grid[0].size()){
        return;
    }

    if(grid[i][j]=='0'){
        return;
    }
    grid[i][j]='0';

    dfs(grid,i+1,j);
    dfs(grid,i-1,j);
    dfs(grid,i,j+1);
    dfs(grid,i,j-1);
}
```

1020.飞地的数量

给你一个大小为 m x n 的二进制矩阵 grid ， 其中 0 表示一个海洋单元格、1 表示一个陆地单元格。

一次 移动 是指从一个陆地单元格走到另一个相邻（上、下、左、右）的陆地单元格或跨过 grid 的边界。

返回网格中 无法 在任意次数的移动中离开网格边界的陆地单元格的数量。

```
int numEnclaves(vector<vector<int>>& grid) {
    // 把能连接到边界的陆地全部淹掉，剩下的就是连接不到的了
    if(grid.size()==0||grid[0].size()==0){
        return 0;
    }
```

```
    int m=grid.size();
    int n=grid[0].size();
    // 淹掉与上边界、下边界相连的陆地
    for(int j=0;j<n;j++){
        if(grid[0][j]==1){
            dfs(grid,0,j);
        }
        if(grid[m-1][j]==1){
            dfs(grid,m-1,j);
        }
    }

    // 淹掉与左边界、右边界相连的陆地
    for(int i=0;i<m;i++){
        if(grid[i][0]==1){
            dfs(grid,i,0);
        }
        if(grid[i][n-1]==1){
            dfs(grid,i,n-1);
        }
    }

    // 还剩几个1，就是右几块飞地
    int res=0;
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            if(grid[i][j]==1){
                res++;
            }
        }
    }

    return res;
}

void dfs(vector<vector<int>>& grid,int i,int j){
    if(i<0||i>=grid.size()||j<0||j>=grid[0].size()){
        return;
    }
    if(grid[i][j]!=1){
        return;
    }
    grid[i][j]=0;
    dfs(grid,i+1,j);
    dfs(grid,i-1,j);
    dfs(grid,i,j+1);
    dfs(grid,i,j-1);
}
```

1254.统计封闭岛屿的数目

二维矩阵 grid 由 0 （土地）和 1 （水）组成。岛是由最大的4个方向连通的 0 组成的群，封闭岛是一个 完全 由1包围（左、上、右、下）的岛。

请返回 封闭岛屿 的数目。

```
int closedIsland(vector<vector<int>>& grid) {
    // 和上次一样：将边界的陆地淹掉，之后再求岛屿的数量

    if(grid.size()==0||grid[0].size()==0){
        return 0;
    }
    int m=grid.size();
    int n=grid[0].size();
    // 淹掉与上边界、下边界相连的陆地
    for(int j=0;j<n;j++){
        if(grid[0][j]==0){
            dfs(grid,0,j);
        }
        if(grid[m-1][j]==0){
```

```
            dfs(grid,m-1,j);
        }
    }

    // 淹掉与左边界、右边界相连的陆地
    for(int i=0;i<m;i++){
        if(grid[i][0]==0){
            dfs(grid,i,0);
        }
        if(grid[i][n-1]==0){
            dfs(grid,i,n-1);
        }
    }

    // 求剩下的岛屿数量
    int res=0;
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            if(grid[i][j]==0){
                dfs(grid,i,j);
                res++;
            }
        }
    }

    return res;
}

void dfs(vector<vector<int>>& grid,int i,int j){
    if(i<0||i>=grid.size()||j<0||j>=grid[0].size()){
        return;
    }
    if(grid[i][j]!=0){
        return;
    }
    grid[i][j]=1;
    dfs(grid,i+1,j);
    dfs(grid,i-1,j);
    dfs(grid,i,j+1);
    dfs(grid,i,j-1);
}
```

695.岛屿的最大面积

给你一个大小为 m x n 的二进制矩阵 grid 。

岛屿 是由一些相邻的 1 (代表土地) 构成的组合，这里的「相邻」要求两个 1 必须在 水平或者竖直的四个方向上 相邻。你可以假设 grid 的四个边缘都被 0（代表水）包围着。

岛屿的面积是岛上值为 1 的单元格的数目。

计算并返回 grid 中最大的岛屿面积。如果没有岛屿，则返回面积为 0 。

```
int maxAreaOfIsland(vector<vector<int>>& grid) {
    if(grid.size()==0||grid[0].size()==0){
        return 0;
    }

    int m=grid.size();
    int n=grid[0].size();

    int res=0;
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            if(grid[i][j]==1){
                res=max(res,dfs(grid,i,j));
            }
        }
```

```
    }

    return res;
}

int dfs(vector<vector<int>>& grid,int i,int j){
    if(i<0||i>=grid.size()||j<0||j>=grid[0].size()){
        return 0;
    }
    if(grid[i][j]==0){
        return 0;
    }
    grid[i][j]=0;

    return 1+dfs(grid,i+1,j)+dfs(grid,i-1,j)+dfs(grid,i,j+1)+dfs(grid,i,j-1);
}
```

1905.统计子岛屿

给你两个 m x n 的二进制矩阵 grid1 和 grid2 ，它们只包含 0 （表示水域）和 1 （表示陆地）。一个 岛屿 是由 四个方向 （水平或者竖直）上相邻的 1 组成的区域。任何矩阵以外的区域都视为水域。

如果 grid2 的一个岛屿，被 grid1 的一个岛屿 完全 包含，也就是说 grid2 中该岛屿的每一个格子都被 grid1 中同一个岛屿完全包含，那么我们称 grid2 中的这个岛屿为 子岛屿 。

请你返回 grid2 中 子岛屿 的 数目 。

```
int countSubIslands(vector<vector<int>>& grid1, vector<vector<int>>& grid2) {
    // 假设输入有效
    int m=grid2.size();
    int n=grid2[0].size();

    int res=0;
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            if(grid2[i][j]==1){
                if(dfs(grid1,grid2,i,j)){
                    res++;
                }
            }
        }
    }
    return res;
}

bool dfs(vector<vector<int>>& grid1, vector<vector<int>>& grid2,int i,int j){
    if(i<0||i>=grid2.size()||j<0||j>=grid2[0].size()){
        return true;
    }
    if(grid2[i][j]==0){
        return true;
    }
    bool res=true;
    if(grid1[i][j]==0){// 不能直接放回，要将该岛屿全部淹掉
        res=false;
    }
    grid2[i][j]=0;
    res=dfs(grid1,grid2,i+1,j)&&res;
    res=dfs(grid1,grid2,i-1,j)&&res;
    res=dfs(grid1,grid2,i,j+1)&&res;
    res=dfs(grid1,grid2,i,j-1)&&res;
    return res;
}
```

130.被包围的区域

给你一个 `m x n` 的矩阵 `board` ，由若干字符 `'X'` 和 `'O'` ，找到所有被 `'X'` 围绕的区域，并将这些区域里所有的 `'O'` 用 `'X'` 填充。

```
void solve(vector<vector<char>>& board) {
    // 1.将与边界相连的'O'用'#'来填充
    // 2.剩下的'O'用'X'填充
    // 3.再将'#'恢复成'O'

    // 假设输入有效
    int m=board.size();
    int n=board[0].size();

    // 淹掉上边界和下边界所连岛屿
    for(int j=0;j<n;j++){
        if(board[0][j]=='O'){
            dfs(board,0,j);
        }
        if(board[m-1][j]=='O'){
            dfs(board,m-1,j);
        }
    }

    // 淹掉左边界和右边界所连岛屿
    for(int i=1;i<m-1;i++){
        if(board[i][0]=='O'){
            dfs(board,i,0);
        }
        if(board[i][n-1]=='O'){
            dfs(board,i,n-1);
        }
    }

    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            if(board[i][j]=='#'){
                board[i][j]='O';
            }else if(board[i][j]=='O'){
                board[i][j]='X';
            }
        }
    }
}

void dfs(vector<vector<char>>& board,int i,int j){
    if(i<0||i>=board.size()||j<0||j>=board[0].size()){
        return;
    }
    if(board[i][j]!='O'){
        return;
    }
    board[i][j]='#';
    dfs(board,i+1,j);
    dfs(board,i-1,j);
    dfs(board,i,j+1);
    dfs(board,i,j-1);
}
```

934.最短的桥

给你一个大小为 n x n 的二元矩阵 grid ，其中 1 表示陆地，0 表示水域。

岛 是由四面相连的 1 形成的一个最大组，即不会与非组内的任何其他 1 相连。grid 中 恰好存在两座岛 。

你可以将任意数量的 0 变为 1 ，以使两座岛连接起来，变成 一座岛 。

返回必须翻转的 0 的最小数目。

```cpp
// 本题先使用dfs，将其中一个岛的1改成2，用来区分另一个岛，并且在这过程中，将该岛的边界0也变成2，并将下标加入到queue中（用下面的bfs）
// 有了queue之后，下面就是常规的bfs了
int shortestBridge(vector<vector<int>>& grid) {
    // 假设输入有效
    int m=grid.size();
    int n=grid[0].size();


    // 将其中一个岛的1变成2，并将边界添加入queue中
    bool flag=false;
    queue<vector<int>>q;
    for(int i=0;i<m;i++){
        if(flag){break;}
        for(int j=0;j<n;j++){
            if(grid[i][j]==1){
                dfs(grid,i,j,q);
                flag=true;
                break;
            }
        }
    }
    vector<int> drct({-1,0,1,0,-1});// 方向数组
    int res=0;
    while(!q.empty()){
        res++;
        int sz=q.size();

        for(int i=0;i<sz;i++){
            int x=q.front()[0];
            int y=q.front()[1];
            q.pop();
            for(int j=0;j<4;j++){
                if(x+drct[j]>=0&&x+drct[j]<m&&y+drct[j+1]>=0&&y+drct[j+1]<n){
                    if(grid[x+drct[j]][y+drct[j+1]]==0){
                        grid[x+drct[j]][y+drct[j+1]]=2;
                        q.push({x+drct[j],y+drct[j+1]});
                    }else if(grid[x+drct[j]][y+drct[j+1]]==1){
                        return res;
                    }
                }
            }
        }
    }
    return res;
}

void dfs(vector<vector<int>>& grid,int i,int j,queue<vector<int>>& q){
    if(i<0||i>=grid.size()||j<0||j>=grid[0].size()){
        return;
    }

    if(grid[i][j]==0){
        grid[i][j]=2;
        q.push({i,j});
        return;
    }
    if(grid[i][j]==2){
        return;
    }

    grid[i][j]=2;
    dfs(grid,i+1,j,q);
    dfs(grid,i-1,j,q);
    dfs(grid,i,j+1,q);
    dfs(grid,i,j-1,q);
}
```

面试题13：机器人的运动范围

地上有一个m行n列的方格，从坐标 [0,0] 到坐标 [m-1,n-1] 。一个机器人从坐标 [0, 0] 的格子开始移动，它每次可以向左、右、上、下移动一格（不能移动到方格外），也不能进入行坐标和列坐标的数位之和大于k的格子。例如，当k为18时，机器人能够进入方格 [35, 37] ，因为3+5+3+7=18。但它不能进入方格 [35, 38]，因为3+5+3+8=19。请问该机器人能够到达多少个格子？

```
int movingCount(int m, int n, int k) {
    int res=0;
    vector<vector<bool>>visited(m,vector<bool>(n,false));
    dfs(m,n,k,0,0,res,visited);
    return res;
}

void dfs(int m,int n,int k,int i,int j,int& res,vector<vector<bool>>& visited){
    if(i<0||i>=m||j<0||j>=n||visited[i][j]||!vaild(i,j,k)){
        return;
    }
    visited[i][j]=true;
    res++;
    dfs(m,n,k,i+1,j,res,visited);
    dfs(m,n,k,i-1,j,res,visited);
    dfs(m,n,k,i,j-1,res,visited);
    dfs(m,n,k,i,j+1,res,visited);
}

bool vaild(int i,int j,int k){
    int sum=0;
    while(i>0||j>0){
        sum+=i%10;
        sum+=j%10;
        i/=10;
        j/=10;
    }
    if(sum>k){
        return false;
    }
    return true;
}
```

79.单词搜索&面试题12-hot100

给定一个 m x n 二维字符网格 board 和一个字符串单词 word 。如果 word 存在于网格中，返回 true ； 否则，返回 false 。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

```
bool exist(vector<vector<char>>& board, string word) {
    if(board.size()==0||board[0].size()==0||board.size()*board[0].size()<word.size()){
        return false;
    }
    for(int i=0;i<board.size();i++){
        for(int j=0;j<board[0].size();j++){
            if(dfs(board,word,i,j,0)){
                return true;
            }
        }
    }

    return false;
}

bool dfs(vector<vector<char>>& board, string& word,int i,int j,int k){
    if(k==word.size()){
        return true;
    }
    if(i<0||i>=board.size()||j<0||j>=board[0].size()||board[i][j]!=word[k]){
```

```
        return false;
    }
    board[i][j]='0';
    bool res=dfs(board,word,i+1,j,k+1);
    res=res||dfs(board,word,i-1,j,k+1);
    res=res||dfs(board,word,i,j+1,k+1);
    res=res||dfs(board,word,i,j-1,k+1);
    board[i][j]=word[k];

    return res;
}
```

17.电话号码的字母组合-hot100

给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。答案可以按 任意顺序 返回。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。


```
vector<string> letterCombinations(string digits) {
    if(digits.size()==0){
        return vector<string>();
    }
    vector<string>strs({"abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"});
    vector<string>res;
    string cur;
    dfs(strs,res,cur,digits,0);
    return res;
}

void dfs(vector<string>& strs,vector<string>& res,string& cur,string& digits,int i){
    if(i==digits.size()){
        res.push_back(cur);
        return;
    }
    for(int j=0;j<strs[digits[i]-'2'].size();j++){
        cur+=strs[digits[i]-'2'][j];
        dfs(strs,res,cur,digits,i+1);
        cur.pop_back();
    }
}
```

21.动态规划

(1) 未通过

面试题60：n个骰子的点数

把n个骰子扔在地上，所有骰子朝上一面的点数之和为s。输入n，打印出s的所有可能的值出现的概率。

你需要用一个浮点数数组返回答案，其中第 i 个元素代表这 n 个骰子所能掷出的点数集合中第 i 小的那个的概率。

```
// 方法一：暴力递归
// f(i,j)：i个骰子掷出j点数有多少种可能
// f(i,j)=f(i-1,j-1)+f(i-1,j-2)+f(i-1,j-3)+f(i-1,j-4)+f(i-1,j-5)+f(i-1,j-6)
int f(int i,int j){
```

```
    if(i<1){return 0;}
    if(j<i||j>6*i){return 0;}
    if(i==1){
        return 1;
    }
    return f(i-1,j-1)+f(i-1,j-2)+f(i-1,j-3)+f(i-1,j-4)+f(i-1,j-5)+f(i-1,j-6);
}
vector<double> dicesProbability(int n) {
    if(n<1){return vector<double>();}
    vector<double>res(5*n+1);
    int sum=pow(6,n);
    for(int i=n;i<=6*n;i++){
        res[i-n]=f(n,i)*1.0/sum;
    }
    return res;
}
```

```
// 方法二：严格表结构
void f(vector<vector<int>>&dp,int n){
    for(int j=1;j<=6;j++){
        dp[1][j]=1;
    }
    for(int i=2;i<=n;i++){
        for(int j=i;j<=6*i;j++){
            dp[i][j]=dp[i-1][j-1];
            if(j-2>0){dp[i][j]+=dp[i-1][j-2];}
            if(j-3>0){dp[i][j]+=dp[i-1][j-3];}
            if(j-4>0){dp[i][j]+=dp[i-1][j-4];}
            if(j-5>0){dp[i][j]+=dp[i-1][j-5];}
            if(j-6>0){dp[i][j]+=dp[i-1][j-6];}
        }
    }
}
vector<double> dicesProbability(int n) {
    if(n<1){return vector<double>();}
    vector<double>res(5*n+1);
    int sum=pow(6,n);
    vector<vector<int>>dp(n+1,vector<int>(6*n+1));
    f(dp,n);
    for(int i=n;i<=6*n;i++){
        res[i-n]=dp[n][i]*1.0/sum;
    }
    return res;
}
```

```
// 方法三：空间压缩，二维
void f(vector<vector<int>>&dp,int n){
    for(int j=1;j<=6;j++){
        dp[1][j]=1;
    }
    for(int i=2;i<=n;i++){
        for(int j=i;j<=6*i;j++){
            dp[i%2][j]=dp[(i-1)%2][j-1];
            // 注意这里将>0改成了>=i-1
            // 在没有空间压缩时，每行最开始的部分为0
            // 进行空间压缩（复用）之后，每行最开始的部分不在为0
            if(j-2>=i-1){dp[i%2][j]+=dp[(i-1)%2][j-2];}
            if(j-3>=i-1){dp[i%2][j]+=dp[(i-1)%2][j-3];}
            if(j-4>=i-1){dp[i%2][j]+=dp[(i-1)%2][j-4];}
            if(j-5>=i-1){dp[i%2][j]+=dp[(i-1)%2][j-5];}
            if(j-6>=i-1){dp[i%2][j]+=dp[(i-1)%2][j-6];}
        }
    }
}
vector<double> dicesProbability(int n) {
    if(n<1){return vector<double>();}
    vector<double>res(5*n+1);
    int sum=pow(6,n);
```

```
    vector<vector<int>>dp(2,vector<int>(6*n+1));
    f(dp,n);
    for(int i=n;i<=6*n;i++){
        res[i-n]=dp[n%2][i]*1.0/sum;
    }
    return res;
}
```

```
// 方法三：空间压缩，一维
void f(vector<int>&dp,int n){
    for(int j=1;j<=6;j++){
        dp[j]=1;
    }
    for(int i=2;i<=n;i++){
        for(int j=6*i;j>=i;j--){
            dp[j]=dp[j-1];
            if(j-2>=i-1){dp[j]+=dp[j-2];}
            if(j-3>=i-1){dp[j]+=dp[j-3];}
            if(j-4>=i-1){dp[j]+=dp[j-4];}
            if(j-5>=i-1){dp[j]+=dp[j-5];}
            if(j-6>=i-1){dp[j]+=dp[j-6];}
        }
    }
}
vector<double> dicesProbability(int n) {
    if(n<1){return vector<double>();}
    vector<double>res(5*n+1);
    int sum=pow(6,n);
    vector<int>dp(6*n+1);
    f(dp,n);
    for(int i=n;i<=6*n;i++){
        res[i-n]=dp[i]*1.0/sum;
    }
    return res;
}
```

647.回文子串

给你一个字符串 s ，请你统计并返回这个字符串中 回文子串 的数目。

回文字符串 是正着读和倒过来读一样的字符串。

子字符串 是字符串中的由连续字符组成的一个序列。

具有不同开始位置或结束位置的子串，即使是由相同的字符组成，也会被视作不同的子串。

```
// 双指针，中心拓展法
int countSubstrings(string s) {
    if(s.size()==1){
        return 1;
    }
    string str=getString(s);

    int res=0;
    for(int i=1;i<str.size()-1;i++){
        res+=f(str,i);
    }
    return res;
}

int f(string& str,int i){
    int len=1;
    int l=i-1;
    int r=i+1;
    while(l>=0&&r<str.size()&&str[l]==str[r]){
        len+=2;
        l--;
        r++;
    }
    len/=2;// 真正回文子串的长度
```

```
    return (len+1)/2;
}

string getString(string& s){
    string str="#";
    for(int i=0;i<s.size();i++){
        str+=s[i];
        str+='#';
    }
    return str;
}
```

```
// 方法二：动态规划
int countSubstrings(string s) {
    int len=s.size();
    vector<vector<bool>>dp(len,vector<bool>(len));

    int res=0;
    for(int i=0;i<len;i++){
        dp[i][i]=true;
        res++;
    }

    // 从下到上，从左往右遍历
    for(int i=len-2;i>=0;i--){
        for(int j=i+1;j<len;j++){
            if(s[i]==s[j]){
                dp[i][j]=(j-i==1||dp[i+1][j-1]);
            }
            if(dp[i][j]){res++;}
        }
    }
    return res;

}
// 空间压缩
int countSubstrings(string s) {
    int len=s.size();
    vector<bool>dp(len);

    int res=0;

    res+=len;
    dp[len-1]=true;
    for(int i=len-2;i>=0;i--){
        dp[i+1]=true;// 关键，对角线上的元素为true
        for(int j=len-1;j>=i+1;j--){
            if(j-i==1&&s[i]==s[j]){dp[j]=true;}
            else{dp[j]=dp[j-1]&&s[i]==s[j];}
            if(dp[j]){res++;}
        }
    }

    return res;
}
```

221.最大正方形-hot100

在一个由 `'0'` 和 `'1'` 组成的二维矩阵内，找到只包含 `'1'` 的最大正方形，并返回其面积。

```
int maximalSquare(vector<vector<char>>& matrix) {
    if(matrix.size()==0||matrix[0].size()==0){return 0;}
    int m=matrix.size();
    int n=matrix[0].size();
    int res=0;
    // dp[i][j]：以(i-1,j-1)为正方形的右下角的最大边长
    // 多一行一列，为basecase，全为0
    vector<vector<int>>dp(m+1,vector<int>(n+1));
```

```
    for(int i=1;i<m+1;i++){
        for(int j=1;j<n+1;j++){
            if(matrix[i-1][j-1]=='1'){
                dp[i][j]=min({dp[i-1][j-1],dp[i-1][j],dp[i][j-1]})+1;
                res=max(res,dp[i][j]);
            }
        }
    }
    return res*res;
}
// 空间压缩
int maximalSquare(vector<vector<char>>& matrix) {
    if(matrix.size()==0||matrix[0].size()==0){return 0;}
    int m=matrix.size();
    int n=matrix[0].size();
    int res=0;
    vector<vector<int>>dp(2,vector<int>(n+1));
    for(int i=1;i<m+1;i++){
        for(int j=1;j<n+1;j++){
            if(matrix[i-1][j-1]=='1'){
                dp[i%2][j]=min({dp[(i-1)%2][j-1],dp[(i-1)%2][j],dp[i%2][j-1]})+1;
                res=max(res,dp[i%2][j]);
            }else{
                dp[i%2][j]=0;// 关键，要置0
            }
        }
    }
    return res*res;
}
// 一维
int maximalSquare(vector<vector<char>>& matrix) {
    if(matrix.size()==0||matrix[0].size()==0){return 0;}
    int m=matrix.size();
    int n=matrix[0].size();
    int res=0;
    vector<int>dp(n+1);
    for(int i=1;i<m+1;i++){
        int pre=dp[0];
        for(int j=1;j<n+1;j++){
            if(matrix[i-1][j-1]=='1'){
                int t=dp[j];
                dp[j]=min({pre,dp[j],dp[j-1]})+1;
                res=max(res,dp[j]);
                pre=t;
            }else{
                dp[j]=0;// 可以直接置0，而不用设置pre，因为，下一位置依赖项的最小值肯定是0
            }
        }
    }
    return res*res;
}
```

1277.统计全为1的正方形子矩阵（和上题相似）

给你一个 `m * n` 的矩阵，矩阵中的元素不是 `0` 就是 `1`，请你统计并返回其中完全由 `1` 组成的 **正方形** 子矩阵的个数。

```
// 几乎和上题一模一样
// dp[i][j]：以(i-1,j-1)为右下角的矩形的个数（就是最大矩形的边长）
int countSquares(vector<vector<int>>& matrix) {
    if(matrix.size()==0||matrix[0].size()==0){
        return 0;
    }
    int m=matrix.size();
    int n=matrix[0].size();
    vector<vector<int>>dp(m+1,vector<int>(n+1));
    int res=0;
    for(int i=1;i<m+1;i++){
        for(int j=1;j<n+1;j++){
            if(matrix[i-1][j-1]==1){
```

```
                dp[i][j]=min({dp[i-1][j],dp[i][j-1],dp[i-1][j-1]})+1;
                res+=dp[i][j];
            }
        }
    }
    return res;
}
// 空间优化
int countSquares(vector<vector<int>>& matrix) {
    if(matrix.size()==0||matrix[0].size()==0){
        return 0;
    }
    int m=matrix.size();
    int n=matrix[0].size();
    vector<int>dp(n+1);
    int res=0;
    for(int i=1;i<m+1;i++){
        int pre=0;
        for(int j=1;j<n+1;j++){
            if(matrix[i-1][j-1]==1){// 与上一题的区别，这里是int，不是char
                int t=dp[j];
                dp[j]=min({dp[j],dp[j-1],pre})+1;
                res+=dp[j];
                pre=t;
            }else{
                dp[j]=0;
            }
        }
    }
    return res;
}
```

139.单词拆分-hot100

给你一个字符串 s 和一个字符串列表 wordDict 作为字典。请你判断是否可以利用字典中出现的单词拼接出 s 。

注意：不要求字典中出现的单词全部都使用，并且字典中的单词可以重复使用。

```
// 动态规划：dp[i]：s[0..i]能否用wordDict表示
// 1.s[0..i]本身在wordDict中
// 2.s[0..j-1]能够用wordDict表示，并且s[j..i]在wordDict中
// 为了方便查找，用unordered_set转存wordDict
bool wordBreak(string s, vector<string>& wordDict) {
    int len = s.size();
    unordered_set<string>st;
    for (string s : wordDict) {
        st.insert(s);
    }
    vector<bool>dp(len, false);
    dp[0] = st.find(s.substr(0, 1)) != st.end();
    for (int i = 1; i < len; i++) {
        dp[i] = st.find(s.substr(0, i + 1)) != st.end();
        for (int j = 1; j <= i; j++) {
            if (dp[i]) { break; }
            dp[i] = (dp[j - 1] && st.find(s.substr(j, i - j + 1)) != st.end());
        }
    }
    return dp[len - 1];
}
```

```
// 方法二：不用哈希set
// 更快
bool wordBreak(string s, vector<string>& wordDict) {
    int len = s.size();
    vector<bool>dp(len+1, false);// dp[i]：s[0..i-1]能否用wordDict表示/从0开始，长度为i的字符串能否用wordDict表示
    dp[0]=true;
    for (int i = 1; i < len+1; i++) {
        for (string word:wordDict) {
            int wlen=word.size();
```

```
            if(i>=wlen&&s.substr(i-wlen,wlen)==word){
                dp[i]=dp[i-wlen];
                if(dp[i]){break;}
            }
        }
    }
    return dp[len];
}
```

3.无重复字符的最长子串&面试题48.-hot100

请从字符串中找出一个最长的不包含重复字符的子字符串，计算该最长子字符串的长度。

```
// 方法一：暴力递归
// 以i为结尾的最长无重复子字符串的长度f(i)=
// 1.当以i-1为结尾的最长无重复子字符串的长度范围内无s[i]，f(i)=f(i-1)+1
// 2.当以i-1为结尾的最长无重复子字符串的长度范围内有s[i]，记上一次出现的位置为j，f(i)=i-j
int f(unordered_map<char,int>& mp,string& s,int i,int& res){
    if(i==0){
        mp[s[0]]=0;// 注意，这里需要设置完之后在返回
        return 1;
    }
    int curLen=f(mp,s,i-1,res)+1;
    if(mp.find(s[i])!=mp.end()){
        int j=mp[s[i]];
        if(i-j<curLen){
            curLen=i-j;
        }
    }
    mp[s[i]]=i;
    res=max(res,curLen);
    return curLen;
}
int lengthOfLongestSubstring(string s) {
    if(s.length()<2){return s.length();}
    int res=1;
    unordered_map<char,int>mp;
    f(mp,s,s.length()-1,res);
    return res;
}
```

```
// 方法二：严格表结构的动态规划
int f(const string& s){
    int res=1;
    unordered_map<char,int>mp;
    vector<int>dp(s.length());
    dp[0]=1;
    mp[s[0]]=0;
    for(int i=1;i<s.length();i++){
        dp[i]=dp[i-1]+1;
        if(mp.find(s[i])!=mp.end()){
            int j=mp[s[i]];
            if(i-j<dp[i]){
                dp[i]=i-j;
            }
        }
        mp[s[i]]=i;
        res=max(res,dp[i]);
    }
    return res;
}
int lengthOfLongestSubstring(string s) {
    if(s.length()<2){return s.length();}

    return f(s);
}
```

```
// 方法三：空间压缩
```

```
int f(const string& s){
    int res=1;
    unordered_map<char,int>mp;
    vector<int>dp(2);
    dp[0]=1;
    mp[s[0]]=0;
    for(int i=1;i<s.length();i++){
        dp[i%2]=dp[(i-1)%2]+1;
        if(mp.find(s[i])!=mp.end()){
            int j=mp[s[i]];
            if(i-j<dp[i%2]){
                dp[i%2]=i-j;
            }
        }
        mp[s[i]]=i;
        res=max(res,dp[i%2]);
    }
    return res;
}
int lengthOfLongestSubstring(string s) {
    if(s.length()<2){return s.length();}

    return f(s);
}
```

```
//方法四：双指针
int lengthOfLongestSubstring(string s) {
    if(s.length()<2){return s.length();}

    return f(s);
}
int f(const string& s){
    int res=1;
    unordered_map<char,int>mp;
    int head=0;
    int tail=0;
    while(head<s.length()){
        if(mp.find(s[head])!=mp.end()&&mp[s[head]]>=tail){
            res=max(res,head-tail);
            tail=mp[s[head]]+1;
        }
        mp[s[head]]=head;
        head++;
    }
    return max(res,head-tail);
}
```

```
// 滑动窗口框架
int lengthOfLongestSubstring(string s) {
    if(s.length()<2){return s.length();}

    unordered_map<char,int>window;
    int l=0;
    int r=0;
    int res=0;

    while(r<s.length()){
        char c=s[r];
        r++;
        window[c]++;
        while(window[c]>1){
            char d=s[l];
            l++;
            window[d]--;
        }
        res=max(res,r-l);
    }
    return res;
```

```
}
```

10.正则表达式匹配&面试题19-hot100

给你一个字符串 s 和一个字符规律 p，请你来实现一个支持 '.' 和 '*' 的正则表达式匹配。

'.' 匹配任意单个字符
'*' 匹配零个或多个前面的那一个元素
所谓匹配，是要涵盖 整个 字符串 s的，而不是部分字符串。

```
bool isMatch(string s, string p) {
    return f(s,p);
}
bool killP(string p, int j){
    if(j==p.length()){return true;}
    int len=p.length();
    if((len-j)%2!=0){return false;}
    for(int i=j+1;i<len;i+=2){
        if(p[i]!='*'||p[i-1]=='*'){return false;}
    }
    return true;
}
bool f(string s, string p){
    int slen=s.length();
    int plen=p.length();
    vector<vector<int>>dp(slen+1,vector<int>(plen+1,false));
    for(int j=0;j<=plen;j++){
        dp[slen][j]=killP(p,j);
    }
    for(int i=0;i<slen;i++){
        if(s[i]==p[plen-1]||p[plen-1]=='.'){dp[i][plen-1]=dp[i+1][plen];}
    }
    for(int i=slen-1;i>=0;i--){
        for(int j=plen-2;j>=0;j--){
            if(p[j+1]=='*'){
                int ii=i;
                while((ii<slen)&&(s[ii]==p[j]||p[j]=='.')){
                    if(dp[ii++][j+2]){
                        dp[i][j]=true;
                        break;
                    }
                }
                dp[i][j]=dp[i][j]||dp[ii][j+2];
            }else{
                if(s[i]==p[j]||p[j]=='.'){dp[i][j]=dp[i+1][j+1];}
            }
        }

    }
    return dp[0][0];
}
```

877.石头游戏

Alice 和 Bob 用几堆石子在做游戏。一共有偶数堆石子，排成一行；每堆都有 正 整数颗石子，数目为 piles[i] 。

游戏以谁手中的石子最多来决出胜负。石子的 总数 是 奇数 ，所以没有平局。

Alice 和 Bob 轮流进行，Alice 先开始 。 每回合，玩家从行的 开始 或 结束 处取走整堆石头。 这种情况一直持续到没有更多的石子堆为止，此时手中 石子最多 的玩家 获胜 。

假设 Alice 和 Bob 都发挥出最佳水平，当 Alice 赢得比赛时返回 true ，当 Bob 赢得比赛时返回 false 。

```
// 方法一：暴力递归
int f(vector<int>& arr, int l, int r) {
    if (l == r) {
        return arr[l];
    }
    return max(arr[l] + s(arr, l + 1, r), arr[r] + s(arr, l, r - 1));
}
```

```
int s(vector<int>& arr, int l, int r) {
    if (l == r) {
        return 0;
    }
    return min(f(arr, l + 1, r), f(arr, l, r - 1));
}

bool stoneGame(vector<int>& arr) {
    return f(arr, 0, arr.size() - 1)>s(arr, 0, arr.size() - 1);
}

// 方法二：动态规划
bool stoneGame(vector<int>& piles) {

    int len=piles.size();

    vector<vector<int>>f(len,vector<int>(len));
    vector<vector<int>>s(len,vector<int>(len));

    for(int i=0;i<len;i++){
        f[i][i]=piles[i];
    }

    for(int k=1;k<len;k++){// 第k条对角线
        for(int i=0;k+i<len;i++){
            int j=i+k;
            f[i][j]=max(s[i+1][j]+piles[i],s[i][j-1]+piles[j]);
            s[i][j]=min(f[i+1][j],f[i][j-1]);
        }
    }
    return f[0][len-1]>s[0][len-1];
}
```

174.地下城游戏

恶魔们抓住了公主并将她关在了地下城 dungeon 的 右下角 。地下城是由 m x n 个房间组成的二维网格。我们英勇的骑士最初被安置在左上角 的房间里，他必须穿过地下城并通过对抗恶魔来拯救公主。

骑士的初始健康点数为一个正整数。如果他的健康点数在某一时刻降至 0 或以下，他会立即死亡。

有些房间由恶魔守卫，因此骑士在进入这些房间时会失去健康点数（若房间里的值为负整数，则表示骑士将损失健康点数）；其他房间要么是空的（房间里的值为 0），要么包含增加骑士健康点数的魔法球（若房间里的值为正整数，则表示骑士将增加健康点数）。

为了尽快解救公主，骑士决定每次只 向右 或 向下 移动一步。

返回确保骑士能够拯救到公主所需的最低初始健康点数。

注意：任何房间都可能对骑士的健康点数造成威胁，也可能增加骑士的健康点数，包括骑士进入的左上角房间以及公主被监禁的右下角房间。

```
int calculateMinimumHP(vector<vector<int>>& dungeon) {
    // 本题是根据右下角往左上角推
    // dp(i,j)：从(i,j)位置到达终点的最小生命值
    // dp(i,j)=min(dp(i+1,j),dp(i,j+1))-dungeon[i][j]
    // 如果dp(i,j)<1,dp(i,j)=1
    int m=dungeon.size();
    int n=dungeon[0].size();
    vector<vector<int>>dp(m,vector<int>(n));
    dp[m-1][n-1]=dungeon[m-1][n-1]<0?-dungeon[m-1][n-1]+1:1;
    for(int i=m-2;i>=0;i--){
        dp[i][n-1]=dp[i+1][n-1]-dungeon[i][n-1];
        if(dp[i][n-1]<=0){
            dp[i][n-1]=1;
        }
    }

    for(int j=n-2;j>=0;j--){
        dp[m-1][j]=dp[m-1][j+1]-dungeon[m-1][j];
        if(dp[m-1][j]<=0){
            dp[m-1][j]=1;
        }
```

```
    }

    for(int i=m-2;i>=0;i--){
        for(int j=n-2;j>=0;j--){
            dp[i][j]=min(dp[i+1][j],dp[i][j+1])-dungeon[i][j];
            if(dp[i][j]<=0){
                dp[i][j]=1;
            }
        }
    }

    return dp[0][0];
}
```

(2) 通过

面试题46：把数字翻译成字符串

给定一个数字，我们按照如下规则把它翻译为字符串：0 翻译成 “a”，1 翻译成 “b”，……，11 翻译成 “l”，……，25 翻译成 “z”。一个数字可能有多个翻译。请编程实现一个函数，用来计算一个数字有多少种不同的翻译方法。

```
// 方法一：暴力递归
// s[i..]有多少翻译方法
int f(string& s,int i){
    if(i==s.length()||i==s.length()-1){return 1;}
    if(s[i]=='1'||(s[i]=='2'&&s[i+1]<'6')){
        return f(s,i+1)+f(s,i+2);
    }
    return f(s,i+1);
}
int translateNum(int num) {
    if(num<0){return 0;}
    string s=to_string(num);
    return f(s,0);
}
```

```
// 方法二：严格表结构
int f(string& s){
    int len=s.length();
    vector<int>dp(len+1);
    dp[len]=1;
    dp[len-1]=1;
    for(int i=len-2;i>=0;i--){
        dp[i]=dp[i+1];
        if(s[i]=='1'||(s[i]=='2'&&s[i+1]<'6')){
            dp[i]+=dp[i+2];
        }
    }
    return dp[0];
}
int translateNum(int num) {
    if(num<0){return 0;}
    string s=to_string(num);
    return f(s);
}
```

```
// 方法三：空间压缩
int f(string& s){
    int len=s.length();
    vector<int>dp(3);
    dp[len%3]=1;
    dp[(len-1)%3]=1;
    for(int i=len-2;i>=0;i--){
        dp[i%3]=dp[(i+1)%3];
        if(s[i]=='1'||(s[i]=='2'&&s[i+1]<'6')){
            dp[i%3]+=dp[(i+2)%3];
        }
    }
```

```
    return dp[0];
}
int translateNum(int num) {
    if(num<0){return 0;}
    string s=to_string(num);
    return f(s);
}
```

面试题47：礼物的最大价值

在一个 m*n 的棋盘的每一格都放有一个礼物，每个礼物都有一定的价值（价值大于 0）。你可以从棋盘的左上角开始拿格子里的礼物，并每次向右或者向下移动一格、直到到达棋盘的右下角。给定一个棋盘及其上面的礼物的价值，请计算你最多能拿到多少价值的礼物？

```
// 动态规划
int maxValue(vector<vector<int>>& grid) {
    if(grid.size()==0||grid[0].size()==0){
        return 0;
    }
    int m=grid.size();
    int n=grid[0].size();
    vector<vector<int>>dp(m+1,vector<int>(n+1));
    for(int i=1;i<m+1;i++){
        for(int j=1;j<n+1;j++){
            dp[i][j]=max(dp[i-1][j],dp[i][j-1])+grid[i-1][j-1];
        }
    }
    return dp[m][n];
}

// 空间压缩
int maxValue(vector<vector<int>>& grid) {
    if(grid.size()==0||grid[0].size()==0){
        return 0;
    }
    int m=grid.size();
    int n=grid[0].size();
    vector<int>dp(n+1);
    for(int i=1;i<m+1;i++){
        for(int j=1;j<n+1;j++){
            dp[j]=max(dp[j],dp[j-1])+grid[i-1][j-1];
        }
    }
    return dp[n];
}
```

面试题49：丑数

我们把只包含质因子 2、3 和 5 的数称作丑数（Ugly Number），1也是丑数。求按从小到大的顺序的第 n 个丑数。

```
// 丑数除1外，全部可以由1乘2/3/5得到
int nthUglyNumber(int n) {
    if(n<1){return -1;}
    vector<int>dp(n);
    dp[0]=1;
    // 分别指向丑数序列*2 3 5的序列，归并排序
    int p2=0;
    int p3=0;
    int p5=0;
    for(int i=1;i<n;++i){
        dp[i]=min({dp[p2]*2,dp[p3]*3,dp[p5]*5});
        if(dp[i]==dp[p2]*2){p2++;}
        if(dp[i]==dp[p3]*3){p3++;}
        if(dp[i]==dp[p5]*5){p5++;}
    }
    return dp[n-1];
}
```

面试题14-I：剪绳子

给你一根长度为 n 的绳子，请把绳子剪成整数长度的 m 段（m、n都是整数，n>1并且m>1），每段绳子的长度记为 k[0],k[1]...k[m-1] 。请问$k[0]*k[1]...k[m-1]$可能的最大乘积是多少？例如，当绳子的长度是8时，我们把它剪成长度分别为2、3、3的三段，此时得到的最大乘积是18。

```
// 方法一：暴力递归，对于长度为n的绳子，假设剪的第一段长度为i，那么后续就有两种情况：
// 1.后续不在剪了，那么得到的乘积为i*(n-i)
// 2.后续的绳子继续剪，就变成了i*规模为(n-i)的子问题的乘积
int cuttingRope(int n) {
    if(n<2){return 0;}
    return f(n);
}
int f(int n){
    if(n<1){return 0;}
    if(n==1){return 1;}
    int res=0;
    for(int i=1;i<n;i++){
        res=max(res,max(i*(n-i),i*f(n-i)));
    }
    return res;
}
```

```
// 方法二：记忆化搜索，暴力递归效率低下的原因是，会有大量重复的计算
// 如果将每次计算的结果都存入一个数组，那么就可以避免重复计算
int cuttingRope(int n) {
    if(n<2){return 0;}
    vector<int>dp(n+1);
    dp[2]=1;
    return f(n,dp);
}
int f(int i,vector<int>& dp){
    if(dp[i]!=0){return dp[i];}
    for(int j=1;j<i;j++){
        dp[i]=max(dp[i],max(j*(i-j),j*f(i-j,dp)));
    }
    return dp[i];
}
```

```
// 方法三：严格表结构的动态规划
// 记忆化搜索是自上而下，严格表结构是根据递推关系，自下而上严格填写表格
int cuttingRope(int n) {
    if(n<2){return 0;}
    if(n==2){return 1;}
    return f(n);
}
int f(int n){
    vector<int>dp(n+1);
    dp[2]=1;
    for(int i=3;i<n+1;++i){
        for(int j=1;j<i;j++){
            dp[i]=max(dp[i],max(j*(i-j),j*dp[i-j]));
        }
    }
    return dp[n];
}
```

方法四：贪心算法

假设将长度为n的绳子切成a段：

$$n = n_1 + n_2 + \ldots + n_a$$

本题等价于求解：

$$max(n_1 * n_2 * \ldots * n_a)$$

根据几何均值不等式，我们知道，等号当且仅当$n_1=n_2=...=n_a$时成立：

$$\frac{n_1+n_2+\ldots+n_a}{a} \geq \sqrt[a]{n_1 n_2 \ldots n_a}$$

因此，将绳子以相等的长度等分为多段 ，得到的乘积最大。设将绳子按照x长度等分为 a段，即n=ax，则乘积为x^a：

$$y = x^a = (x^{\frac{1}{x}})^n$$

总长度n不变，求：$y=x^{\frac{1}{x}}$的最大值，求导得：

$$y' = \frac{1-lnx}{x^2} \cdot x^{\frac{1}{x}}$$

即在x=e时，取最大值。x为整数，可知x=3时，取最大值。

因此：**尽可能把绳子分成长度为3的小段，这样乘积最大。**步骤如下：

1. 如果 n == 2，返回1
2. 如果 n == 3，返回2
3. 如果 n == 4，返回4
4. 如果 n > 4，分成尽可能多的长度为3的小段，每次循环长度n减去3，乘积res乘以3；最后返回时乘以小于等于4的最后一小段

```
// 方法四：动态规划
int cuttingRope(int n) {
    if(n<2){return 0;}
    if(n<4){return n-1;}
    if(n==4){return 4;}
    int res=1;
    while(n>4){
        res*=3;
        n-=3;
    }
    return res*n;
}
```

面试题14-II：剪绳子II

给你一根长度为 n 的绳子，请把绳子剪成整数长度的 m 段（m、n都是整数，n>1并且m>1），每段绳子的长度记为 k[0],k[1]...k[m - 1] 。请问 k[0]*k[1]*...*k[m - 1] 可能的最大乘积是多少？例如，当绳子的长度是8时，我们把它剪成长度分别为2、3、3的三段，此时得到的最大乘积是18。

答案需要取模 1e9+7（1000000007），如计算初始结果为：1000000008，请返回 1。

```
// 只需在上题得基础上取模即可
int cuttingRope(int n) {
    if(n<2){return 0;}
    if(n<4){return n-1;}
    if(n==4){return 4;}
    int res=1;
    int p=1000000007;
    while(n>4){
        res=((long)res*3)%p;// long的目的是防止取模前溢出
        n-=3;
    }
    return ((long)res*n)%p;
}
```

面试题10-I：斐波那契数列

写一个函数，输入 `n` ，求斐波那契（Fibonacci）数列的第 `n` 项（即 `F(N)`）。斐波那契数列的定义如下：

$$F(0) = 0, F(1) = 1$$
$$F(N) = F(N-1) + F(N-2), \text{其中} N > 1.$$

斐波那契数列由 0 和 1 开始，之后的斐波那契数就是由之前的两数相加而得出。

答案需要取模 1e9+7（1000000007），如计算初始结果为：1000000008，请返回 1。

```cpp
// 方法一：根据公式，递归，超时
int fib(int n) {
    if(n<0){return -1;}
    if(n<2){return n;}
    return (fib(n-1)+fib(n-2))%(int)(1e9+7);
}
```

```cpp
// 方法二：严格表结构的动态规划
int fib(int n){
    if(n<0){return -1;}
    if(n<2){return n;}
    vector<int>dp(n+1);
    dp[0]=0;
    dp[1]=1;
    for(int i=2;i<n+1;i++){
        dp[i]=(dp[i-1]+dp[i-2])%1000000007;
    }
    return dp[n];
}
```

```cpp
// 方法三：空间压缩
int fib(int n){
    if(n<0){return -1;}
    if(n<2){return n;}
    vector<int>dp(3);
    dp[0]=0;
    dp[1]=1;
    for(int i=2;i<n+1;i++){
        dp[i%3]=(dp[(i-1)%3]+dp[(i-2)%3])%1000000007;
    }
    return dp[n%3];
}
```

```cpp
// 方法四：斐波那契套路
// 辅助函数可以根据具体的题目，具体对待，比如求二维的乘积/幂，不一定要具有通用性
// 辅助函数1：求两矩阵的乘积
vector<vector<int>> matrixMult(const vector<vector<int>>&a,const vector<vector<int>>&b){
    if(// 需保证两矩阵均不为空，且a的列数等于b的行数
        a.size()==0||a[0].size()==0
        ||b.size()==0||b[0].size()==0
        ||a[0].size()!=b.size()
    ){return vector<vector<int>>();}

    int m=a.size();
    int n=a[0].size();
    int r=b[0].size();
    vector<vector<int>>res(m,vector<int>(r));

    for(int i=0;i<m;++i){
        for(int j=0;j<r;++j){
            for(int k=0;k<n;++k){
                res[i][j]+=((long)a[i][k])*((long)b[k][j])%1000000007;//防止相乘后溢出
                res[i][j]%=1000000007;
            }
        }
    }
    return res;
}
// 辅助函数2：矩阵的快速幂
vector<vector<int>> matrixPower(vector<vector<int>>& base,int n){
    if(n<0||base.size()==0||base.size()!=base[0].size()){
        //求幂需保证为方阵
        return vector<vector<int>>();
    }
    vector<vector<int>>res(base.size(),vector<int>(base.size));
    for(int i=0;i<base.size();++i){res[i][i]=1;}
    while(n!=0){
```

```
        if((n&1)==1){res=matrixMult(res,base);}
        base=matrixMult(base,base);
        n=n>>1;
    }
    return res;
}
// 主函数
int fib(int n){
    if(n<0){return -1;}
    if(n<2){return n;}
    vector<vector<int>>base({{1,1},{1,0}});// 根据递推关系得出
    vector<vector<int>>pow_base=matrixPower(base,n-1);//求base的n-1次方
    return pow_base[0][0];
}
```

总结：斐波那契套路

除了初始项之外，后序每一项都有严格递归式的问题，时间复杂度O(logN)。注意是严格递归式，不能有条件选择。

斐波那契数列问题（二阶）：$f(n)=f(n-1)+f(n-2)$

$$[f(2),f(1)]=[f(1),f(0)]\cdot\begin{bmatrix}a & b\\ c & d\end{bmatrix}$$

$$[f(3),f(2)]=[f(2),f(1)]\cdot\begin{bmatrix}a & b\\ c & d\end{bmatrix}$$

$$[f(n),f(n-1)]=[f(n-1),f(n-2]\cdot\begin{bmatrix}a & b\\ c & d\end{bmatrix}$$

$$[f(n),f(n-1)]=[f(1),f(0)]\cdot\begin{bmatrix}a & b\\ c & d\end{bmatrix}^{n-1}$$

矩阵中的a b c d可以根据初始项算出，依次问题转化为矩阵求幂，而矩阵求幂有时间复杂度O(logN)的方法，矩阵的快速幂。

斐波那契套路：首先判断是几阶的递推关系，决定是几阶的系数矩阵；根据初始项求出系数矩阵；递推出F(N)与初始项的关系；利用矩阵的快速幂法求系数矩阵的幂。

例：有一只母牛，每年生一只母牛，三年后新生的母牛成熟了，一块生母牛，并且假设，牛不会死，问n年后母牛的数量?

$$f(n)=f(n-1)+f(n-3)$$

例：一只母兔，每年生两只母兔，新生的兔子两年之后成熟，会一块生母兔，兔子会在五年之后死掉，求n年后的兔子数量?

$$f(n)=f(n-1)+f(n-2)-f(n-5)$$

面试题10-II：青蛙跳台阶问题

一只青蛙一次可以跳上1级台阶，也可以跳上2级台阶。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

答案需要取模 1e9+7（1000000007），如计算初始结果为：1000000008，请返回 1。

n级的台阶最后一步可以由跳2级到达，也可以由跳1级到达，因此：

$$f(n)=f(n-1)+f(n-2)$$

和斐波那契数列是同一个问题，只是初始条件不同而已。

70.爬楼梯-hot100

设你正在爬楼梯。需要 `n` 阶你才能到达楼顶。

每次你可以爬 `1` 或 `2` 个台阶。你有多少种不同的方法可以爬到楼顶呢?

与青蛙跳台阶问题相同

494.目标和-hot100

给你一个整数数组 nums 和一个整数 target 。

向数组中的每个整数前添加 '+' 或 '-' ，然后串联起所有整数，可以构造一个 表达式 ：

例如，nums = [2, 1] ，可以在 2 之前添加 '+' ，在 1 之前添加 '-' ，然后串联起来得到表达式 "+2-1" 。
返回可以通过上述方法构造的、运算结果等于 target 的不同 表达式 的数目。

```
// 方法一：暴力递归
```

```
int findTargetSumWays(vector<int>& nums, int target) {
    if(nums.size()==0){
        return 0;
    }
    return f(nums,0,target);
}
// 现在来到i位置,nums[i]可以取正/负，nums[i..len-1]组成taget的方法数
int f(vector<int>& nums, int i, int target){
    if(i==nums.size()){
        if(target==0){
            return 1;
        }else{
            return 0;
        }
    }
    return f(nums,i+1,target+nums[i])+f(nums,i+1,target-nums[i]);
}

// 改动态规划
int findTargetSumWays(vector<int>& nums, int target) {
    if(nums.size()==0){
        return 0;
    }
    return f(nums,target);
}
// 现在来到i位置,nums[i]可以取正/负，nums[i..len-1]组成taget的方法数
int f(vector<int>& nums, int target){
    int sum=0;
    for(int i=0;i<nums.size();i++){
        sum+=nums[i];
    }
    target=abs(target);
    if(sum<target){
        return 0;
    }
    int len=nums.size();
    vector<vector<int>>dp(len+1,vector<int>(2*sum+1));
    dp[len][sum-target]=1;
    for(int i=len-1;i>=0;i--){
        for(int j=0;j<2*sum+1;j++){
            if(j+nums[i]<2*sum+1){
                dp[i][j]+=dp[i+1][j+nums[i]];
            }
            if(j-nums[i]>=0){
                dp[i][j]+=dp[i+1][j-nums[i]];
            }
        }
    }
    return dp[0][sum];
}

// 空间压缩
int findTargetSumWays(vector<int>& nums, int target) {
    if(nums.size()==0){
        return 0;
    }
    return f(nums,target);
}
// 现在来到i位置,nums[i]可以取正/负，nums[i..len-1]组成taget的方法数
int f(vector<int>& nums, int target){
    int sum=0;
    for(int i=0;i<nums.size();i++){
        sum+=nums[i];
    }
    target=abs(target);
    if(sum<target){
        return 0;
    }
    int len=nums.size();
    vector<vector<int>>dp(2,vector<int>(2*sum+1));
```

```
    dp[len%2][sum-target]=1;
    for(int i=len-1;i>=0;i--){
        for(int j=0;j<2*sum+1;j++){
            dp[i%2][j]=0;
            if(j+nums[i]<2*sum+1){
                dp[i%2][j]+=dp[(i+1)%2][j+nums[i]];
            }
            if(j-nums[i]>=0){
                dp[i%2][j]+=dp[(i+1)%2][j-nums[i]];
            }
        }
    }

    return dp[0][sum];
}
```

312.戳气球-hot100

有 n 个气球，编号为0 到 n - 1，每个气球上都标有一个数字，这些数字存在数组 nums 中。

现在要求你戳破所有的气球。戳破第 i 个气球，你可以获得 nums[i - 1] * nums[i] * nums[i + 1] 枚硬币。 这里的 i - 1 和 i + 1 代表和 i 相邻的两个气球的序号。如果 i - 1或 i + 1 超出了数组的边界，那么就当它是一个数字为 1 的气球。

求所能获得硬币的最大数量。

```
// 方法一：暴力递归
int maxCoins(vector<int>& arr) {
    if(arr.size()==0){return 0;}
    int len=arr.size();
    // 将原数组的前后都增加一个1
    arr.resize(len+2,1);
    for(int i=len;i>0;i--){
        arr[i]=arr[i-1];
    }
    arr[0]=1;
    return f(arr,1,len);
}
// f(l,r)：在arr[l..r]范围内所能获得的硬币最大数量
// arr[l..r]范围内的每个气球都尝试最后被打爆，取max
// f(l,r)=max(arr[l-1]*arr[r+1]*arr[i]+f(l,i-1)+f(i+1,r))
// 可以发现调取f(l,r)的位置一定没爆，因为他是在尝试最后被打爆时调用的该函数，那么它肯定没爆
int f(vector<int>& arr,int l,int r){
    if(l==r){return arr[l-1]*arr[l]*arr[r+1];}
    int res=0;
    res=max(arr[l-1]*arr[l]*arr[r+1]+f(arr,l+1,r),arr[l-1]*arr[r]*arr[r+1]+f(arr,l,r-1));

    for(int i=l+1;i<r;i++){
        res=max(res,arr[l-1]*arr[r+1]*arr[i]+f(arr,l,i-1)+f(arr,i+1,r));
    }
    return res;
}

// 改动态规划，注意观察依赖关系：从左下角开始
int maxCoins(vector<int>& arr) {
    if(arr.size()==0){return 0;}
    int len=arr.size();
    arr.resize(len+2,1);
    for(int i=len;i>0;i--){
        arr[i]=arr[i-1];
    }
    arr[0]=1;
    return f(arr);
}
int f(vector<int>& arr){
    int len=arr.size();
    vector<vector<int>>dp(len-1,vector<int>(len-1));
    for(int i=1;i<=len-2;i++){
        dp[i][i]=arr[i-1]*arr[i]*arr[i+1];
    }
    for(int i=len-3;i>=1;i--){
```

```
        for(int j=i+1;j<=len-2;j++){
            dp[i][j]=max(arr[i-1]*arr[i]*arr[j+1]+dp[i+1][j],arr[i-1]*arr[j]*arr[j+1]+dp[i][j-1]);
            for(int k=i+1;k<j;k++){
                dp[i][j]=max(dp[i][j],arr[i-1]*arr[j+1]*arr[k]+dp[i][k-1]+dp[k+1][j]);
            }
        }
    }
    return dp[1][len-2];
}
```

62.不同路径-hot100

一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为 “Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为 “Finish” ）。

问总共有多少条不同的路径?

```
int uniquePaths(int m, int n) {
    if(m==0||n==0){return 0;}
    return f(m,n);
}
// 方法一：暴力递归
int f(int m,int n,int i,int j){
    if(i==m||j==n){return 0;}
    if(i==m-1,j==n-1){return 1;}
    return f(m,n,i+1,j)+f(m,n,i,j+1);
}
// 方法二：严格表结构
int f(int m,int n){
    vector<vector<int>>dp(m,vector<int>(n,1));

    for(int i=m-2;i>=0;i--){
        for(int j=n-2;j>=0;j--){
            dp[i][j]=dp[i+1][j]+dp[i][j+1];
        }
    }
    return dp[0][0];
}
// 方法三：压缩成一维
int f(int m,int n){
    vector<int>dp(n,1);
    for(int i=m-2;i>=0;i--){
        dp[n-1]=1;
        for(int j=n-2;j>=0;j--){
            dp[j]=dp[j]+dp[j+1];
        }
    }
    return dp[0];
}
```

55.跳跃游戏-hot100

给定一个非负整数数组 `nums` ，你最初位于数组的 **第一个下标** 。数组中的每个元素代表你在该位置可以跳跃的最大长度。判断你是否能够到达最后一个下标。

```
// 方法一：深搜，自己想的
// 对于位置i，他能跳到的位置是i..i+nums[i]
bool canJump(vector<int>& nums) {
    if(nums.size()<2){return true;}
    return dfs(nums,0);
}
bool dfs(vector<int>& nums,int i){
    if(i==nums.size()-1){return true;}
    if(i>=nums.size()){return false;}
    if(nums[i]==0){return false;}
    bool res=false;
    for(int j=1;j<=nums[i];j++){
        if(res){break;}
        res=dfs(nums,i+j);
```

```
    }
    return res;
}
// 方法二：改动态规划
bool canJump(vector<int>& nums) {
    if(nums.size()<2){return true;}
    return dfs(nums);
}
bool dfs(vector<int>& nums){
    int len=nums.size();
    vector<bool>dp(len);
    dp[len-1]=true;

    for(int i=len-2;i>=0;i--){
        for(int j=1;j<=nums[i]&&i+j<len;j++){
            if(dp[i]){break;}
            dp[i]=dp[i+j];
        }
    }
    return dp[0];
}
// 方法三：贪心
// 如果从某一位置可以跳到位置j，那么小于j的任意位置均能跳到
// 假设是由位置位置i跳到的位置j，那么nums[i]>=j-i,那么大于i小于j的位置均能由i跳到；
//至于小于i的位置能否跳到j，由于当前已经在i位置了，那么必存在某一位置能够跳到i，因此小于i的位置也能跳到j
bool canJump(vector<int>& nums) {
    if(nums.size()<2){return true;}
    int k=0;
    for(int i=0;i<=k;i++){
        k=max(k,i+nums[i]);
        if(k>=nums.size()-1){return true;}
    }
    return false;
}
```

931.下降路径最小和

给你一个 n x n 的 方形 整数数组 matrix ， 请你找出并返回通过 matrix 的下降路径 的 最小和 。

下降路径 可以从第一行中的任何元素开始，并从每一行中选择一个元素。在下一行选择的元素和当前行所选元素最多相隔一列（即位于正下方或者沿对角线向左或者向右的第一个元素）。具体来说，位置 (row, col) 的下一个元素应当是 (row + 1, col - 1)、(row + 1, col) 或者 (row + 1, col + 1) 。

```
int minFallingPathSum(vector<vector<int>>& matrix) {
    if(matrix.size()==0||matrix[0].size()==0){
        return 0;
    }
    int m=matrix.size();
    int n=matrix[0].size();
    vector<vector<int>>dp(2,vector<int>(n));
    dp[0]=matrix[0];
    for(int i=1;i<m;i++){
        for(int j=0;j<n;j++){
            dp[i%2][j]=dp[(i-1)%2][j];
            if(j>0){
                dp[i%2][j]=min(dp[i%2][j],dp[(i-1)%2][j-1]);
            }
            if(j<n-1){
                dp[i%2][j]=min(dp[i%2][j],dp[(i-1)%2][j+1]);
            }
            dp[i%2][j]+=matrix[i][j];
        }
    }
    return *min_element(dp[(m-1)%2].begin(),dp[(m-1)%2].end());

}
```

53.最大子数组和&面2试题45-hot100

给你一个整数数组 `nums` ，请你找出一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。**子数组** 是数组中的一个连续部分。

本和面试题45相同：面试题45的思路是用一个变量curSum表示以当前位置为结尾的最大子数组和，更新之前判断，上一位置的curSum是否大于0，大于0就加上，否则当前位置为结尾的子数组的最大和就是其本身。代码如下：

```
    int len=nums.size();
    if(len==0){return INT_MIN;}
    if(len==1){return nums[0];}
    int res=nums[0];
    int curSum=nums[0];
    for(int i=1;i<len;i++){
        if(curSum>0){
            curSum+=nums[i];
        }else{
            curSum=nums[i];
        }
        res=max(res,curSum);
    }
    return res;
```

一种更为朴素的解法是先求数组的前缀和数组，找到当前位置之前的最小前缀和，用当前的前缀和减去之前最小的前缀和即为以当前位置为结尾的子数组的最大前缀和，代码如下：

```
int maxSubArray(vector<int>& nums) {
    int len=nums.size();
    if(len==0){return INT_MIN;}
    if(len==1){return nums[0];}
    vector<int>presum(len);
    presum[0]=nums[0];
    for(int i=1;i<len;i++){
        presum[i]=presum[i-1]+nums[i];
    }
    int minSum=0;
    int res=INT_MIN;
    for(int i=0;i<len;i++){
        res=max(res,presum[i]-minSum);
        minSum=min(minSum,presum[i]);
    }
    return res;
}
```

上面的前缀和数组可以压缩称一个变量：

```
int maxSubArray(vector<int>& nums) {
    int len=nums.size();
    if(len==0){return INT_MIN;}
    if(len==1){return nums[0];}
    int presum=nums[0];
    int minSum=min(nums[0],0);
    int res=nums[0];
    for(int i=1;i<len;i++){
        presum+=nums[i];
        res=max(res,presum-minSum);
        minSum=min(minSum,presum);
    }
    return res;
}
```

64.最小路径和-hot

给定一个包含非负整数的 `*m* x *n*` 网格 `grid` ，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。**说明：**每次只能向下或者向右移动一步。

```
int minPathSum(vector<vector<int>>& grid) {
    if(grid.size()==0||grid[0].size()==0){
```

```
        return 0;
    }
    int m=grid.size();
    int n=grid[0].size();
    vector<vector<int>>dp(m,vector<int>(n));
    dp[0][0]=grid[0][0];
    for(int i=1;i<m;i++){
        dp[i][0]=dp[i-1][0]+grid[i][0];
    }
    for(int j=1;j<n;j++){
        dp[0][j]=dp[0][j-1]+grid[0][j];
    }
    for(int i=1;i<m;i++){
        for(int j=1;j<n;j++){
            dp[i][j]=min(dp[i-1][j],dp[i][j-1])+grid[i][j];
        }
    }
    return dp[m-1][n-1];
}
```

514.自由之路

650.只有两个键的键盘

91.解码方法

542.01 矩阵

413.等差数列划分

31.设计数据结构

(1) 未通过

(2) 通过

146.LRU缓存-hot100

请你设计并实现一个满足 LRU (最近最少使用) 缓存 约束的数据结构。
实现 LRUCache 类：
LRUCache(int capacity) 以 正整数 作为容量 capacity 初始化 LRU 缓存
int get(int key) 如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1 。
void put(int key, int value) 如果关键字 key 已经存在，则变更其数据值 value ；如果不存在，则向缓存中插入该组 key-value 。如果插入操作导致关键字数量超过 capacity ，则应该 逐出 最久未使用的关键字。
函数 get 和 put 必须以 O(1) 的平均时间复杂度运行。

```
// 1.根据LRU的含义，需要频繁改变元素的位置或删除元素；链表插入、删除元素方便，但查找麻烦；哈希表查找方便，因此可以让链表搭配哈希表使用
// 2.哈希表的key为节点的key，哈希表的value为节点的地址，这样就可以很方便的查找到key对应的节点
// 3.链表采用双向链表，里面存储题目给的key和val
// 4.LRU中维护head和tail，分别是伪头节点和伪尾结点
class Node{
public:
    int key;
    int val;
    Node* next;
    Node* pre;
    Node(int k=0,int v=0):key(k),val(v),next(nullptr),pre(nullptr){}

};
class LRUCache {
private:
    int capacity;
    Node* head;
    Node* tail;
```

```
    unordered_map<int,Node*>mp;// key：key；value：Node*
private:
    // 将节点n脱离链表
    void separate(Node* n){
        Node* pre=n->pre;
        Node* next=n->next;
        pre->next=next;
        next->pre=pre;
    }
    // 将孤立的节点n插入到链表尾
    void insert2tail(Node* n){
        Node* pre=tail->pre;
        Node* next=tail;
        n->pre=pre;
        pre->next=n;
        n->next=next;
        next->pre=n;
    }
public:
    LRUCache(int capacity) {
        // 假设输入的capacity合法
        this->capacity=capacity;
        head=new Node();
        tail=new Node();
        head->next=tail;
        tail->pre=head;
    }

    ~LRUCache(){
        Node* p=head;
        while(p!=nullptr){
            Node* t=p;
            p=p->next;
            delete t;
        }
    }

    int get(int key) {
        if(mp.find(key)==mp.end()){
            return -1;
        }
        Node* n=mp[key];
        separate(n);
        insert2tail(n);
        return n->val;
    }

    void put(int key, int value) {
        // key已经存在
        if(mp.find(key)!=mp.end()){
            Node* n=mp[key];
            n->val=value;
            separate(n);
            insert2tail(n);
        }
        // key不存在
        else{
            // 容量已满，需要先删除一个节点再插入新的节点
            // 删除链表中的节点，和删除mp中的节点
            if(mp.size()==capacity){
                Node* del=head->next;
                separate(del);
                mp.erase(del->key);
                delete del;
                del=nullptr;
            }
            Node* n=new Node(key,value);
            mp[key]=n;
            insert2tail(n);
        }
```

```
    }
};
```

五、校招真题

华为

4.19暑期实习

1.服务器能耗统计

服务器有三种运行状态:空载、单任务、多任务，每个时间片的能耗的分别为1、3、4;

每个任务由起始时间片和结束时间片定义运行时间:

如果一个时间片只有一个任务需要执行，则服务器处于单任务状态;

如果一个时间片有多个任务需要执行，则服务器处于多任务状态;

给定一个任务列表，请计算出从第一个任务开始，到所有任务结束，服务器的总能耗。

解答要求

时间限制: C/C++ 100ms,其他语言: 200ms

内存限制: C/C++ 128MB,其他语言: 256MB

输入

一个只包含整数的二维数组:

```
1. num
2. start0 end0
3. start1 end1
4. ...
```

第一行的数字表示一共有多少个任务

后续每行包含由空格分割的两个整数，用于确定每一个任务的起始时间片和结束时间片;

任务执行时间包含起始和结束时间片，即任务执行时间是左闭右闭的;

结束时间片一定大于等于起始时间片;

时间片范围: [0，1000000]: 任务数范围: [1,10000];

输出

一个整数，代表服务器的总能耗。

样例1

```
输入: 2
          25
          89
输出: 20
解释: [01] 没有任务需要执行，能耗为0
[2.5]处于单任务状态，能耗为3*4 = 12
[6,7] 处于空载状态，能耗为1*2 = 2
[8,9]处于单任务状态，能耗为3*2 =6
共计能耗为12 + 2 + 6 = 20
```

样例2

```
输入: 3
          4 8
          1 6
          2 9
输出: 3 4
解释:
[1,1] 处于单任务状态,能耗为3*1= 3
[2,8] 处于多任务状态，能耗为4*7=28
[9,9]处于单任务状态，能耗为3*1 = 3
共计能耗为3 + 28 + 3 = 34
```

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



int f(vector<vector<int>>& tasks) {
    int minv = INT_MAX;
    int maxv = INT_MIN;

    for (int i = 0; i < tasks.size(); i++) {
        minv = min(minv, tasks[i][0]);
        maxv = max(maxv, tasks[i][1]);
    }

    vector<int>vec(maxv - minv + 2);

    for (int i = 0; i < tasks.size(); i++) {
        int begin = tasks[i][0];
        int end = tasks[i][1];

        vec[begin - minv] += 1;
        vec[end - minv+1] -= 1;
    }

    int cnt0 = 0;
    int cnt1 = 0;
    int cnt2 = 0;

    int pre = vec[0];
    if (pre == 1) {
        cnt1 = 1;
    }
    else {
        cnt2 = 1;
    }

    for (int i = 1; i < maxv - minv + 1; i++) {
        pre += vec[i];
        if (pre == 0) {
            cnt0++;
        }
        else if (pre == 1) {
            cnt1++;
        }
        else {
            cnt2++;
        }
    }

    return cnt0 * 1 + cnt1 * 3 + cnt2 * 4;
}



int main() {
```

```
    // --------------接收数据-----------------------------------
    // n个任务
    int n = 0;
    cin >> n;

    // 每个任务的起始时间和终止时间
    vector<vector<int>>tasks(n, vector<int>(2));

    int i = 0;

    while (i < n) {
        cin >> tasks[i][0] >> tasks[i][1];
        i++;
    }

    // --------------处理逻辑-----------------------------------
    int res = f(tasks);


    // ---------------打印输出-----------------------------------
    cout << res;

    return 0;
}
```

2.树上逃离

给定一棵树，这个树有n个节点，节点编号从0开始依次递增，0固定为根节点。在这棵树上有一个小猴子，初始时该猴子位于根节点(0号) 上，小猴子一次可以沿着树上的边从一个节点挪到另一个节点，但这棵树上有一些节点设置有障碍物，如果某个节点上设置了障碍物，小猴子就不能通过连接该节点的边挪动到该节点上。问小猴子是否能跑到树的叶子节点(叶子节点定义为只有一条边连接的节点)，如果可以，请输出小猴子跑到叶子节点的最短路径(通过的边最少)，否则输出字符串NULL。

解答要求

时间限制: C/C++ 1000ms,其他语言: 2000ms

内存限制: C/C++ 256MB,其他语言: 512MB

输入

第一行给出数字n，表示这个树有n个节点，节点编号从0开始依次递增，0固定为根节点，1<=n<10000

第二行给出数字edge，表示接下来有edge行，每行是一条边

接下来的edge行是边: x y，表示x和y节点有一条边连接

边信息结束后接下来的一行给出数字block，表示接下来有block行，每行是个障碍物

接下来的block行是障碍物: X，表示节点x上存在障碍物

输出

如果小猴子能跑到树的叶子节点，请输出小猴子跑到叶子节点的最短路径(通过的边最少)，比如小猴子从0经过1到达2 (叶子节点) ，那么输出“0->1->2”，否则输出“NULL”。注意如果存在多条最短路径，请按照节点序号排序输出，比如0->1和0->3两条路径，第一个节点0一样，则比较第二个节点1和3，1比3小，因此输出0->1这条路径。再如 0->5->2->3 和0->5->1->4，则输出 0->5-31->4

样例1

```
输入: 4
          3
          0 1
          0 2
          0 3
          2
          2
          3
输出: 0->1

解释: n=4,edge=[[0,1],[0,2],[0,3]],block=[2,3]表示一个有4个节点、3条边的树，其中节点2和节点3上有障碍物，小猴子能从0到达叶子节点1 (节点1只有一条边[1,0]和它连接，因此是叶子节点) ，即可以跑出这个树，所以输出为0->1。
```

样例2

```
输入: 7
          6
          0 1
          0 3
          1 2
          3 4
          1 5
          5 6
          1
输出: 0->1->2
解释: 节点4上有障碍物，因此0-3-4这条路不通，节点2和节点6都是叶子节点，但0->1->2比0->1->5->6路径短(通过的边最少) ，因此输出为0->1->2
```

样例3

```
输入: 2
          1
          0 1
          1
          1
输出: NULL
解释:节点1是叶子节点，但存在障碍物，因此小猴子无法到达叶子节点，输出NULL
```

样例4

```
输入:4
         3
         0 1
         0 2
         0 3
         1
         2
输出: 0->1
解释: n=4,edge=[[0,1],[0,2],[0,3]],block=[2] 表示一个有4个节点、3条边的树，其中节点2上有障碍物，小猴子能从0到达叶子节点1 (节点1只有一条边[0,1]和它连接，因此是叶子节点) ，路径是0->1，也能从0到达叶子节点3(节点3只有一条边[0,3]和它连接，因此是叶子节点) ,路径是0->3，因此按通过节点的顺序及序号比较选择 0->1。
```

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


void help(int i,vector<vector<int>>& nexts, vector<bool>& haveBloc, vector<int>& res, vector<int>& cur) {
    if (haveBloc[i]) {
        return;
    }

    cur.push_back(i);

    if (nexts[i].size() == 0) {
        if (res.size() > cur.size()) {
            res = cur;
        }
        return;
    }

    for (int j = 0; j < nexts[i].size(); j++) {
        help(nexts[i][j], nexts, haveBloc, res, cur);
    }

    cur.pop_back();
}

vector<int> f(int n,vector<vector<int>>& edges, vector<int>& blocks) {
```

```
    // 保证相同长度时，节点小的在前
    sort(edges.begin(), edges.end(),
            [](vector<int>& a, vector<int>& b) {
                return a[0] == b[0] ? a[1] < b[1] : a[0] < b[0];
            }
        );

    vector<bool>haveBloc(n);

    for (int i = 0; i < blocks.size(); i++) {
        haveBloc[blocks[i]] = true;
    }

    if (haveBloc[0]) {
        return vector<int>();
    }

    vector<vector<int>>nexts(n);

    for (int i = 0; i < edges.size(); i++) {
        int begin = edges[i][0];
        int end = edges[i][1];
        nexts[begin].push_back(end);
    }

    vector<int>res(n+1);
    vector<int>cur;

    help(0, nexts, haveBloc, res, cur);

    return res;

}


int main() {



    // --------------接收数据-----------------------------------
    // n个节点
    int n = 0;
    cin >> n;

    // edge条边
    int edge = 0;
    cin >> edge;

    // 边
    vector<vector<int>>edges(edge, vector<int>(2));

    int i = 0;

    while (i < edge) {
        cin >> edges[i][0] >> edges[i][1];
        i++;
    }

    int block = 0;
    cin >> block;


    // 障碍物
    vector<int>blocks(block);
    i = 0;
    while (i < block) {
        cin >> blocks[i];
        i++;
    }
```

```
    // --------------处理逻辑----------------------------------
    vector<int>res = f(n,edges, blocks);


    // ---------------打印输出----------------------------------
    if (res.size() > n) {
        cout << "NULL";
    }
    else {
        i = 0;
        for (; i < res.size()-1; i++) {
            cout << res[i] << "->";
        }

        cout << res[i];
    }

    return 0;
}
```

5.10暑期实习

1.栈数据合并

向一个空栈压入正整数，每当压入一个整数时，执行以下规则(设: 栈顶至栈底整数依次编号为n1、n2...nx， n1为最新压入的整数)

1.如果n1=n2，则n1、n2全部出栈，压入新数据m(m=2*n1)

2.如果n1=n2+...+ny(y的范围为[3,x])，则n1、n2...ny全部出栈，压入新数据m(m=2*n1)。

3.如果上述规则都不满足，则不做操作

如: 依次向栈压入6、1、2、3，当压入2时，栈顶至栈底依次为[2、1.6];当压入3时，3=2+1，3、2、1全部出栈，重新入栈整数6，此时栈顶至栈底依次为[6、6]; 6=6，两个6全部出栈，压入12，最终栈中只剩个元素12。

向栈中输入一串数字，请输出应用此规则后栈中最终存留的数字。

```
输入: 10 20 50 80 1 1
输出: 2 160
解释: 向栈压入80时，10+20+50=80，数据合并后入栈160，压入两个1时，合并为2，最终栈顶至栈底的数字为2和160。
```

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


void f(vector<int>& nums, stack<int>& sk) {
    sk.push(nums[0]);

    for (int i = 1; i < nums.size(); i++) {
        stack<int>sk2;
        int sum = 0;
        bool flag = false;
        while (!sk.empty() && sum < nums[i]) {
            int tp = sk.top();
            sk.pop();
            sum += tp;
            if (sum == nums[i]) {
                flag = true;
                sk.push(2 * sum);
                break;
            }
            sk2.push(tp);
        }
        if (!flag) {
            while (!sk2.empty()) {
                sk.push(sk2.top());
```

```
                sk2.pop();
            }
            sk.push(nums[i]);
        }
    }
}


int main() {

    vector<int>nums;
    int cur;
    while (cin >> cur) {
        nums.push_back(cur);
    }

    stack<int>sk;
    f(nums, sk);

    while (!sk.empty()) {
        cout << sk.top() << " ";
        sk.pop();
    }
    cout << endl;

    return 0;
}
```

2.寻找密码串

敌占区地下工作者冒死提供了加密后的字符串，需要你根据预先定好的方式进行解密，得到其中真正的密码。加密后字符串M是由0~9这10个数字组成的字符串，你手上是一个给定秘钥数字N和一个运算符k (加减乘中的一个)，需要按如下规则找出直正的密码。

1.截取M中的某一段数字x，和数字N进行k运算 (x k N)，如果结果是一个所有位数相同的数，则这段数字有可能就是所找密码，例如x为222，N为3，k为"，则计算结果是222*3=666，满足要求，x是所寻目标彩码串之一。

2.如果存在满足第1点的多种情况，则以计算结果最大的为准;

3.如果没有找到符合条件的密码串，则输出-1，表示密码串不存在

3.M的长度<100，N为正整数，且N<=9999999999，3<=所找密码长度<=12。k为+或-或中的一种，不考虑除法。为避免数据过于庞大，约定在乘法场景下，乘数最大为3位数。

输入：

```
提供加密后字符串M，秘钥数字N和运算符k，例如：
(加密字符串M) 748443217098
(秘钥数字N) 123
(运算符k) +
```

输出：

```
满足计算结果所有位数相同，且计算结果最大的值。

例如:上述例子截取44321，和123相加，则为44321+123=44444，结果所有位的数字字符相同，包括个位数、十位数、百位数、千位数和万位数都是同一个数字字符4，且其值最大。

(目标字符串) 44321
```

样例1：
输入：
68330238877930769988104187l0
2211
-

输出：
9988

解释：
通过计算，8877-2211=6666，而9988-2211=7777，因为7777>6666，则目标密码串为9988。

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

bool issame(int x) {
    if (x < 0) {
        return false;
    }

    int bit = x % 10;
    x = x / 10;
    while (x != 0) {
        if (x % 10 != bit) {
            return false;
        }
        x = x / 10;
    }

    return true;
}

int process(string& subs, int num, char op) {
    long long x = stoll(subs);
    int res = -1;
    if (op == '+') {
        res = x + num;
    }
    else if (op == '-') {
        res = x - num;
    }
    else {
        res = x * num;
    }
    if (issame(res)) {
        return res;
    }

    return -1;
}

int f(string& s, int num, char op) {
    int res = -1;// 最佳码串
    int n = -1;// 最佳码串对应的计算结果

    int len = s.length();
    for (int i = 3; i <= min(len, 12); i++) {
        for (int j = 0; j <= len - i; j++) {
            string subs = s.substr(j, i);
            int cur = process(subs, num, op);// 当前码串对应的计算结果
            if (cur > n) {
                n = cur;
                res = stoi(subs);
            }
        }
    }

    return res;
```

```
}


int main() {
    string s;
    int num;
    char op;

    cin >> s >> num >> op;
    cout << f(s, num, op) << endl;
}
```

3.微服务调用链路染色最短时间

在微服务架构中，一个请求可能会经过若干个服务节点，其所经过的路径，我们称之为请求调用链路。通常，我们会把一些服务治理相关的字段(例traceId)，通过请求头的方式在整个链路中透传。当我们把有特定含义的请求头透传到整个链路，然后链路中每个服务会针对这个请求头做一些特殊的处理，我们把这种行为称之为链路染色。现给出在某一请求下其经过所有微服务节点的响应时长列表rsTimes，其中rsTimes[i]=(srcSerivce,dstService,rsTime)，其中srcSerivce为调用方服务,dstService为被调用方服务，所有服务名称由一个1到n范围内的整数表示，rsTime为接口调用响应耗时。如果srcSerivce与dstService相同，则表示系统自身计算耗时。所以如果服务srcService到dstService的染色总时长为srcService到dstService响应时长+dstService计算耗时，现给出一个调用源头服务名称service，请给出以这个服务作为源头服务发起调用，最终可实现染色的服务个数，并给出这些服务全部完成染色的最短时间。

输入

第一行表示服务节点总数m

第二行表示服务间调用响应耗时或者服务自身计算耗时rsTmes的长度n

接下来n行表示具体的服务间调用响应耗时或者服务自身计算耗时

rsTmes[i]，每个数字间用空格隔开，比如 10 20 3，则表示10号服务调用20号服务的耗时为3秒

最后一行表示调用方源服务名称

提示:

1<=rsTimes.lenath<=5000 1<=srcSerivce<=100 1<=dstService<=100 1<=rsTime<=100

输出

输出分为两行，第一行输出最终可实现染色的微服务个数，第二行输出这些服务全部完成染色的最短时间

```
样例:
输入:
5
9
1 1 3
1 2 6
1 3 10
3 3 8
2 2 7
2 4 12
2 5 20
5 5 5
4 4 9
1
输出:
5
38
```

```
#include<iostream>
#include<string>
#include<vector>
#include<queue>
#include<list>
using namespace std;

class State{
public:
    int id;
    int curDis;
```

```
    State(int i, int d) :id(i), curDis(d){}
};

class my_com {
public:
    bool operator()(const State& s1, const State& s2) {
        return s1.curDis > s2.curDis;
    }
};

vector<int> f(vector<list<vector<int>>>& edges,vector<int>& selfs,int k) {
    int n=edges.size();
    vector<int>res(n, INT_MAX);
    res[k] = 0;
    priority_queue<State, vector<State>, my_com>q;
    q.push(State(k, 0));
    while (!q.empty()) {
        State cur = q.top();
        q.pop();

        for (vector<int>edge : edges[cur.id]) {
            int to = edge[0];
            int weight = edge[1];
            if (res[to] > cur.curDis + weight+selfs[to]) {
                res[to] = cur.curDis + weight + selfs[to];
                q.push(State(to, res[to]));
            }
        }
    }
    int cnt = 0;
    int maxv = 0;
    res[k] += selfs[k];
    for (int i = 0; i < n; i++) {
        if (res[i] != INT_MAX) {
            cnt++;
            maxv = max(maxv, res[i]);
        }
    }
    return vector<int>({ cnt,maxv });
}


int main() {

    int n;// 节点数
    cin >> n;
    vector<list<vector<int>>>edges(n);
    vector<int>selfs(n, 0);
    int i;
    cin >> i;
    while (i != 0) {
        int from;
        int to;
        int weight;
        cin >> from >> to >> weight;
        if (from == to) {
            selfs[from - 1] = weight;
        }
        else {
            edges[from - 1].push_back({ to - 1,weight });
        }
        i--;
    }
    int k;
    cin >> k;

    vector<int>res = f(edges, selfs, k-1);
    cout << res[0] << " " << res[1] << endl;
```

```
    return 0;
}
```