2023.7.8

下午：

题目一：

给你两个偶整数n和m。你的任务是找到任何具有n行和m列的二进制矩阵a，其中每个单元格（i，j）恰好有两个与ai.j值不同的邻居。

基质中的两个细胞被认为是相邻的，当且仅当它们共用一个侧。更正式地说，细胞（2，4）的邻居是：（x - 1，y），（x，y+ 1），（x+ 1,y）和（x，y-1）

可以证明，在给定的约束下，答案总是存在的。

输入描述

每个测试都包含多个测试用例。第一行输入包含单个整数t（1 < t < 100）-测试用例的数量。以下几行包含测试用例的描述。

每个测试用例的唯一行包含两个偶整数n和m（2 <=n，m < =50）-分别是二进制矩阵的高度和宽度。

输出描述

对于每个测试用例，打印n行，每行都包含m个数字，等于0或1-任何满足以下约束的二进制矩阵

声明。

思路：每一个元素附近都有两个元素跟自己不一样，呈十字 左右上下会有两个和自己不同的元素

然后就是找规律

1001

0110

0110

1001

代码：自敲，未完待续.....

题目二：

我想知道，下大雨吗

永远渴望它的蔑视?

心灵的 Effluvium

你得到一个正整数n。

找到长度为n的任何置换p，使得总和lem（1，p1）+

lcm（2，p.）+ . .. + lcm（n，pm.）尽可能大。

这里cm(2,4)表示整数2和y的最小公倍数（LCM）。

置换是由从1到n的任意顺序的n个不同整数组成的数组。例如，[2, 3, 1, 5, 4]是置换，但（1，2，2不是置换（2在数组中出现两次），（1，3，4）也不是置换（n = 3，但数组中有4）。

输入描述

每个测试都包含多个测试用例。第一行包含测试用例t的数量（1 < t < 1 000）。测试用例的描述如下。

每个测试用例的onlv行包含一个sinale整数n（1 < n < 105）

保证所有测试用例的n的总和不超过105

思路： a，b最小公倍数等于=(a*b)/最大公约数

这道题就是a和b相差1的时候最优

假设共有n个数

若n为偶数，则两两一对刚刚好

所以就是1，2，3，4，5，6

两两互换，即：2，1，4，3，6，5

2·1+4·3+6·5 就是最大值

若n为奇数，则从大到小开始排，优先满足大的两两相乘（大的两两相乘能更大）

例：1，2，3，4，5

则：1，3，2，5，4

代码如下：

没写，未完待续.......

题目三：

A permutation of length n is a sequence of integers from 1 to n such that each
integer appears in it exactly once.
Let the fixedness of a permutation o be the number of fixed points in it - the
number of positions j such that p; = j, where p; is the j-th element of the
permutation p.
You are asked to build a sequence of permutations a1, a2, . . ., starting from the
identity permutation (permutation a1 = (1, 2,.
.., n). Let's call it a
permutation chain. Thus, a; is the i-th permutation of length n.
For every i from 2 onwards, the permutation a should be obtained from the
permutation a;-1 by swapping any two elements in it (not necessarily

neighboring). The fixedness of the permutation a;, should be strictly lower than the fixedness of the permutation di_1.
Consider some chains for n. = 3•
• a1 = 1,2,31, a2 = [1, 3, 21 - that is a valid chain of length 2. From a1 to an, the elements on positions 2 and 3 get swapped, the fixedness decrease from 3 to 1.
a1 = 12,1, 31, a2
= 13,1,21.

- that is not a valid chain. The first
  permutation should always be (1, 2, 3] for n = 3
  at
  = 1, 2, 31, a2 = [1, 3,21, a3 = [1, 2, 31
- that is not a valid chain.
- From ay to a3, the elements on positions 2 and 3 get swapped but the fixedness increase from 1 to 3.
  a1 = 1,2,31, a2 = [3, 2, 11, az
  = 3,1,2 - that is a valid chain of length
    3. From a1 to a2, the elements on positions 1 and 3 get swapped, the fixedness decrease from 3 to 1. From a2 to a3, the elements on positions 2 and 3 get swapped, the fixedness decrease from 1 to 0.
       Find the longest permutation chain. If there are multiple longest answers, print anv of them.
       输入描述
       The first line contains a single integer t (1 < t < 99) - the number of testcases.
       The only line of each testcase contains a single integer n (2 < n < 100) - the required length of permutations in the chain.

思路：

长度为n，两两互换

如：1 2 3　n=3

换一次：3 2 1 n=1（只有一个对上）

再换一次，如 2 3 1 n=1-1=0

所以是2

题目四：

2023.7.10

String a;

cin>>a;(读到空格停止)

get line(a,cin);

getline(a,100);

栈：后进先出

队列：先进先出

stl

pair:和结构体差不多

pair(a,b) 第一个用first访问，第二个用second访问

vector：两倍扩展

set：返回迭代器

去重

第一题

思路：查找、去重

去重用set

简单版：//

// 4.cpp

```
// c++
//
// Created by 语何 on 2023/7/10.
//

//第一题简单版

#include <stdio.h>
#include
#include
using namespace std;
set<int>s1;
int main(){
 int n,m;
 while(cin>>n>>m){
  s1.clear();   //用完清空
  int x;
  for(int i=0;i<n+m;i++){
   cin>>x;
   s1.insert(x);
  }
  set<int>::iterator it;
  int flag=1;
  for(it=s1.begin();it!=s1.end();it++)
  {
   if(flag){
    cout<<*it;
    flag=0;
   }

  }
```

```
    cout<<"\n";
  }

  return 0;
}
```

复杂版：//

```
// 3.cpp
// c++
//
// Created by 语何 on 2023/7/10.
//

#include <stdio.h>



//stl 第一题复杂版

#include
#include
using namespace std;
vector<int>v1,v2,v3;
int a[1000]={0};
void in(int a,int b){
  for (int i = 0; i < a; ++i)
  {
    int c;
    cin >> c;
    v1.push_back(i);
  }
```

```
  for (int i = 0; i < b; ++i)
  {
    v2.push_back(i);
  }
      v3.insert(v3.end(),v1.begin(),v1.end());
      v3.insert(v3.end(),v2.begin(),v2.end());



}
int t;
void choose(vector<int>v){
  int x;
  for(int j=0;j<v.size();j++){
    t=v[j];
    a[v[j]]++;
    if(a[v[j]]>1)v.erase(v.begin()+j-1);
  }
  for(int i=0;i<v.size()-1;i++){
        for(int j=0;j<v.size()-i-1;j++){
          if(v[j]>v[j+1]){
            x=v[j];
            v[j]=v[j+1];
            v[j+1]=x;
          }
        }




  }
```

```
}
void print(vector<int>v){
  for(int j=0;j<v.size();j++){
    cout<<v[j] <<endl;
  }
}

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

    in(n,m);
    choose(v3);
    print(v3);

  }

}
```

第三题：

解题思路：出现在a，b里面，但不在c里面

是一个查找问题

第一组和第三组用set、map存都行，第二组数据是要有序的，按序输出，就用queue，先进先出。

代码：//

```
// 5.cpp
// c++
//
// Created by 语何 on 2023/7/10.
//
```

//第三题 间谍问题：查重

```
#include
#include
#include
using namespace std;
sets1,s3;
queueq;
int main(){
  int a,b,c,flag;
  cin>>a>>b>>c;
  for(int i=0;i<a;i++){
    string s;
    cin>>s;
    s1.insert(s);
  }
  for(int i=0;i<b;i++){
    string s;
    cin>>s;
    q.push(s);
  }
  for(int i=0;i<c;i++){
    string s;
    cin>>s;
    s3.insert(s);
  }
  while(!q.empty()){
    string str=q.front();
    q.pop();
```

```
    if(s1.find(str)!=s1.end()&&s3.find(str)==s3.end()) { //在a里面找到了，在c里面没找到
      cout<<str<<" ";
      flag=1;

    }
  }
  if(flag==0)
}
```

第四题：

水果

思路：map的嵌套使用，因为一个map只能对应一个数据。

第六题：

圆桌问题：

把看成一个环形的，取出一个推进一个

第五题

思路：要看优先级，结构体排序

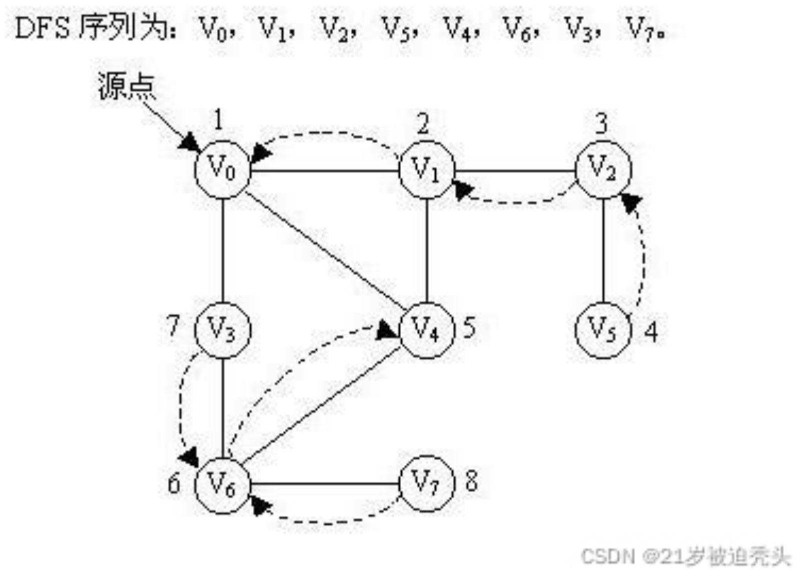
2023.7.11

dfs

一、基本思想
为了求得问题的解，先选择某一种可能情况向前探索；
在探索过程中，一旦发现原来的选择是错误的，就退回一步重新选择，继续向前探索；
如此反复进行，直至得到解或证明无解。
二、操作步骤：
初始原点为v0，使用深度优先搜索，首先访问 v0 -> v1 -> v2 -> v5，到 v5 后面没有结点，则回溯到 v1 ，即最近的且连接有没访问结点的结点v1；
此次从 v1 出发，访问 v1 -> v4 -> v6 -> v3，此时与v3相连的两个结点 v0 与 v6 都已经访问过，回溯到 v6 (v6 具有没访问过的结点)；
此次从 v6 出发，访问 v6 -> v7，到 v7 后面没有结点，回溯；
一直回溯到源点 v0 ，没有没访问过的结点，程序结束。
注：下面图中箭头为回溯方向

深搜：跑一遍打个标记

dfs



dfs详细过程：

```
#include<iostream>
using namespace std;
int n;
int ans[20];
bool vis[20];

```

```
void dfs(int step){  //step = 3

    if(step == n+1){
        for(int i=1;i<=n;i++){
            printf("%d",ans[i]);
        }
        printf("\n");
        return;
    }


    for(int i=1;i<=n;i++){
        if(vis[i])continue;
        vis[i] = true;
        ans[step] = i;
        dfs(step+1);
        //执行完成
        vis[i] = false;
    }

    return;
}

int main(){
    cin>>n;  //  n = 4

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

1 2 3 4

dfs1{ 1.i=1,vis=true,ans[1]=1;

2.开出新的分支dfs2{1.i=1 被标记了，跳过

i=2,vis=true,ans[2]=2,

2.开出新的分支dfs3{ 1.i=1、2都被标记了，跳过，vis=true(即打上标记)

2.开出新的分支dfs4{ 1.i=1、2、3都被标记了，跳过，vis=true(即打上标记)

开出新的分支dfs5{ 1.step=5=n+1,输出a[1-4]即1、2、3、4，return(回溯) 回溯一级

回到dfs4:这个时候i=4，vis[4]=false，释放4

回到dfs3:这个时候i=3，vis[3]=false，释放3

然后继续未走完的循环i=4,ans[step]=ans[3]=i=4, 标记4

dfs(step+1)--dfs(step)---dfs(4)---step=4

dfs4:i=1,2都被标记，i=3,ans[step]=ans[4]=i=3;

然后dfs5,输出1243 return

dfs4:i=3还有最后一步vis[3]=false释放3

dfs模版-c

int a[510]; //存储每次选出来的数据
int book[510]; //标记是否被访问
int ans = 0; //记录符合条件的次数

void DFS(int cur){
  if(cur == k){ //k个数已经选完，可以进行输出等相关操作
    for(int i = 0; i < cur; i++){
      printf("%d ", a[i]);
    }
    ans++;
    return ;
  }

```
for(int i = 0; i < n; i++){ //遍历 n个数，并从中选择k个数
  if(!book[i]){ //若没有被访问
    book[i] = 1; //标记已被访问
    a[cur] = i;  //选定本数，并加入数组
    DFS(cur + 1);  //递归，cur+1
    book[i] = 0;  //释放，标记为没被访问，方便下次引用
  }
}
```

}

dfs模版--c++

```
vector a; // 记录每次排列
vector book; //标记是否被访问

void DFS(int cur, int k, vector& nums){
    if(cur == k){ //k个数已经选完，可以进行输出等相关操作
        for(int i = 0; i < cur; i++){
            printf("%d ", a[i]);
        }
        return ;
    }
    for(int i = 0; i < k; i++){ //遍历 n个数，并从中选择k个数
        if(book[nums[i]] == 0){ //若没有被访问
            a.push_back(nums[i]); //选定本输，并加入数组
            book[nums[i]] = 1; //标记已被访问
            DFS(cur + 1, n, nums); //递归，cur+1
            book[nums[i]] = 0; //释放，标记为没被访问，方便下次引用
            a.pop_back(); //弹出刚刚标记为未访问的数
        }
    }
}
```

2023.7.12

二分查找

模版

```
int search(int nums[], int size, int target) //nums是数组，size是数组的大小，target是需要查找的值
{
    int left = 0;
    int right = size - 1;    // 定义了target在左闭右闭的区间内，[left, right]
    while (left <= right) {    //当left == right时，区间[left, right]仍然有效
        int middle = left + ((right - left) / 2);//等同于 (left + right) / 2，防止溢出
        if (nums[middle] > target) {
            right = middle - 1; //target在左区间，所以[left, middle - 1]
        } else if (nums[middle] < target) {
            left = middle + 1;  //target在右区间，所以[middle + 1, right]
        } else {    //既不在左边，也不在右边，那就是找到答案了
            return middle;
        }
    }
    //没有找到目标值
    return -1;
}
```

最小化最大值

注意注意：有可能left或者right刚刚好等于mid 所以有一个不能选择=mid+1 不然会刚好错过mid

2023.7.13

博弈论

共有n个数，一次能取m个，谁先取最后一个谁赢。那我们将n/(m+1);就是一次我们能确保后手始终能达到的数量

能被整除：后手必胜

不能被整除：、当1=k*(1+1+x0<x<m+1时，先手可以先取x个，之后的局势就回到
了第了种情况，无论后手取多少个，先手都能取走 m＋1 个中剩下的，因此先手必
胜

a=(b-a)·(1+sqr(根号5))/2

用（i,j）来表示两堆石头数量，可以发现当（0,0）时，先手不能取石头，所以先手输。我们称(0，0）为奇异局势，如果数量为(0,1),(1,0)，（1,1)，先手都可以一步操作
到达奇异局势，后手便会失败，所以可以得出结论，当(i，j 为奇异局势时，先手输，否则先手赢。我们可以发现(1,2)(3，5)（4，7) 等也是奇异局势，并且它们有一个规律，
这个规律就是通过 Beatty定理发现公式：

a=(b-a)·(1+sqr(根号5))/2

两对石头头分别为a，b。当满足前面这个条件时，(a，b）就是奇异点。
而且 个式子正是等于黄金分割比0.618+1是1.618.

如果是10和5 那先手必胜，因为先手可以先取7个 形成(3,5)来构造奇异局势

模版--八什博弈

•两个人轮流报数，一次最少报一最多报十，先报到100的人获胜，谁能赢?

思路：假设最多能报m，讲每次凑成m+1

#include

```
using namespace std;
int main(){
  int t;
  cin>>t;
  while (t--) {
    int n,m;
    cin>>n>>m;
    if(!(n%(m+1)))cout<<"second"<<endl;
    else cout<<"first"<<endl;
  }
  return 0;
}
```

2023.7.14

单调栈

模版：

```
#include
#include
using namespace std;
const int N=1e5+10;
int a[N],b[N];
signed main(){
  stack<int>st;
  int n;cin>>n;
  for(int i=0;i<n;i++){
    while(st.size()!=0&&st.top()>=a[i]) st.pop(); //如果栈顶比a[i]大 就把栈顶推出去，把a[i]推进去
    st.push(a[i]);
```

```
  }
  while(st.size()!=0){
    cout<<st.top()<<" ";
    st.pop();
  }
  return 0;
}
```

例题：

```
#include
#include
using namespace std;
const int N=1e6+7;
int a[N],b[N];
signed main(){
  stack<int>st;
  int n;cin>>n;
  for(int i=0;i<n;i++) cin>>a[i];
    //5
  //3 4 2 7 5
  //-1 3 -1 2 2
  //2 5
  //3 4 2 5
  for(int i=0;i<n;i++){
    while(st.size()!=0&&st.top()>=a[i])st.pop(); //如果出现一个比栈内元素小的数字，则该数字没有前面比它小的数字，清空栈，输出-1，将其作为top
    if(st.size()==0)cout<<"-1"<<" ";  //第一个数字前面没有，所以栈没东西，输出-1
    else cout<<st.top()<<" ";
    st.push(a[i]);
  }
  return 0;
```

}

例题：观山

“我看见，一座座山，一座座山川，一座座山川相连…”。现在有n座山连成一排，请问当你站在两座山之间时，向前和向后共能看到多少座山（当前面的山的高度大于等于后面的山的高度时，后面的山将被挡住）。

输入描述

输入第一行为一个整数n，表示山的个数。1<= n <=10^5

第二行包含n个空格隔开的整数，表示n座山的高度。1<= a[i] <=10^5

输出描述

输出空格隔开的n-1个数，表示在第i座山和第i + 1座山之间，能看到的山的个数。

1 <= i < n

解题思路：分为从左边看和从右边看

从左边看 若山不挡住 例如1 2 3 4

我们需要维护一个单调递增的栈；

从左右看 若山不挡住 例如4 3 2 1

我们需要维护一个单调递减的栈

代码：

观牛

代码：//

// 观牛（也是单调栈）.cpp

// c++

//

// Created by 语何 on 2023/7/14.

//

```
#include
#include
using namespace std;
#define int long long
const int N=1e6+7;
int a[N];
signed main(){
  int n;cin>>n;
  for(int i=0;i<n;i++)cin>>a[i];

  stack<int>st;
  int sum=0;
  for(int i=0;i<n;i++){
    while(st.size()!=0&&st.top()<=a[i])st.pop();
    sum+=st.size();
    st.push(a[i]);
  }
  cout<<sum<<endl;
  return 0;

}
```

2023.7.19

树状数组

求一个大的数组

2023.7.20

并查集

并查集：判断两个元素是不是在同一个集合里面

快读：

```
std::ios::sync_with_stdio(false);
    std::cin.tie(0);
```

2023.7.25

最小生成树

2023.7.26

倍增LCA

2023.7.27

tarjan

2023.7.28

网络流

网络流：所有的链式前向星存图都得是双向存边

2023.7.31

欧拉图 二分图

G=(V,E)

V:点集   E:边集

欧拉路径

对于无向图：

存在欧拉路径的充要条件：度数为奇数的点只能有0个或两个（且以该点为起点）

存在欧拉回路的充要条件：度数为奇数的点不能有

对于有向图：

存在欧拉路径的充要条件：要么所有点除了出度均等于入度；要么除了两个点之外，其余所有点的出度等于入度，剩余的两个点：一个满足出度比入度多一（起点），另一个满足入度比出度多一（终点）。

存在欧拉回路的充要条件：所有点的出度均等于入度。

匈牙利算法：如果是带权值的就用匈牙利去做

不带权值的就用最大流（网络流）

看看有多少个不同的并查集

在每个集子里面的奇度点

2023.8.1

动态规划-背包问题

01背包问题

```
对于01背包问题选择方法的集合可以分成2种：
```

①不选第i个物品，并且总体积不大于j的集合所达到的最大值:f[i-1][j]
②选择1~i个物品，并且总体积不大于j的集合所达到的最大值f[i][j]

对于第二种情况我们很难计算，因此需要思考从另一个角度解决问题。当选择1i个物品，总体积不大于j的集合的最大值可以转化成选择1i-1个物品，总体积不大于j-V[i]的集合+最后一个物品的价值：f[i-1][j-V[i]]+w[i]
因此总结：f[i][j]= Max{f[i-1][j],f[i-1][j-v[i]]+w[i]}!!!

背包问题的解决过程

在解决问题之前，为描述方便，首先定义一些变量：Vi表示第 i 个物品的价值，Wi表示第 i 个物品的体积，定义V(i,j)：当前背包容量 j，前 i 个物品最佳组合对应的价值，同时背包问题抽象化（X1，X2，…，Xn，其中 Xi 取0或1，表示第 i 个物品选或不选）。
1、建立模型，即求max(V1X1+V2X2+…+VnXn)；
2、寻找约束条件，W1X1+W2X2+…+WnXn<capacity；
3、寻找递推关系式，面对当前商品有两种可能性：
包的容量比该商品体积小，装不下，此时的价值与前i-1个的价值是一样的，即V(i,j)=V(i-1,j)；
还有足够的容量可以装该商品，但装了也不一定达到当前最优价值，所以在装与不装之间选择最优的一个，即V(i,j)=max｛V(i-1,j)，V(i-1,j-w(i))+v(i)｝。
其中V(i-1,j)表示不装，V(i-1,j-w(i))+v(i) 表示装了第i个商品，背包容量减少w(i)，但价值增加了v(i)；
由此可以得出递推关系式：

j<w(i)　　V(i,j)=V(i-1,j)

j>=w(i)　　V(i,j)=max｛V(i-1,j)，V(i-1,j-w(i))+v(i)｝

这里需要解释一下，为什么能装的情况下，需要这样求解（这才是本问题的关键所在！）：
可以这么理解，如果要到达V(i,j)这一个状态有几种方式?
肯定是两种，第一种是第i件商品没有装进去，第二种是第i件商品装进去了。没有装进去很好理解，就是V(i-1,j)；装进去了怎么理解呢？如果装进去第i件商品，那么装入之前是什么状态，肯定是V(i-1,j-w(i))。由于最优性原理（上文讲到），V(i-1,j-w(i))就是前面决策造成的一种状态，后面的决策就要构成最优策略。两种情况进行比较，得出最优。
4、填表，首先初始化边界条件，V(0,j)=V(i,0)=0；

如，i=1，j=1，w(1)=2，v(1)=3，有j<w(1)，故V(1,1)=V(1-1,1)=0；
又如i=1，j=2，w(1)=2，v(1)=3，有j=w(1),故V(1,2)=max｛V(1-1,2)，V(1-1,2-w(1))+v(1)｝=max｛0，0+3｝=3；
如此下去，填到最后一个，i=4，j=8，w(4)=5，v(4)=6，有j>w(4)，故V(4,8)=max｛V(4-1,8)，V(4-1,8-w(4))+v(4)｝=max｛9，4+6｝=10……

背包问题最优解回溯

通过上面的方法可以求出背包问题的最优解，但还不知道这个最优解由哪些商品组成，故要根据最优解回溯找出解的组成，根据填表的原理可以有如下的寻解方式：
V(i,j)=V(i-1,j)时，说明没有选择第i 个商品，则回到V(i-1,j)；
V(i,j)=V(i-1,j-w(i))+v(i)时，说明装了第i个商品，该商品是最优解组成的一部分，随后我们得回到装该商品之前，即回到V(i-1,j-w(i))；
一直遍历到i＝0结束为止，所有解的组成都会找到。
就拿上面的例子来说吧：
最优解为V(4,8)=10，而V(4,8)!=V(3,8)却有V(4,8)=V(3,8-w(4))+v(4)=V(3,3)+6=4+6=10，所以第4件商品被选中，并且回到V(3,8-w(4))=V(3,3)；
有V(3,3)=V(2,3)=4，所以第3件商品没被选择，回到V(2,3)；

而V(2,3)!=V(1,3)却有V(2,3)=V(1,3-w(2))+v(2)=V(1,0)+4=0+4=4，所以第2件商品被选中，并且回到V(1,3-w(2))=V(1,0)；
有V(1,0)=V(0,0)=0，所以第1件商品没被选择。

例题：

有 N 件物品和一个容量是 V 的背包。每件物品只能使用一次。

第 i 件物品的体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。
输出最大价值。

输入格式

第一行两个整数，N，V，用空格隔开，分别表示物品数量和背包容积。

接下来有 N 行，每行两个整数 v_i, w_i，用空格隔开，分别表示第 i 件物品的体积和价值。

输出格式

输出一个整数，表示最大价值。

数据范围

$0 < N, V \leq 1000$
$0 < v_i, w_i \leq 1000$

输入样例

```
4 5
1 2
2 4
3 4
4 5
```

输出样例：

```
8
```

代码板子：

```
#include
using namespace std;

const int N=1010;
int v[N],w[N],f[N][N];
int n,m;

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

  for(int i=1;i<=n;i++) scanf("%d%d",&v[i],&w[i]);
```

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

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

#注意： 为什么i和j从1开始遍历，因为如果i或j不管哪个为0,f[i][j]其实都等于0！！

优化板子：

```
#include
using namespace std;

const int N=1010;
int v[N],w[N],f[N];
int n,m;

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

  for(int i=1;i<=n;i++) scanf("%d%d",&v[i],&w[i]);

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


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

如何优化:

　　从二维做法中可以看出f[i] [j]最大值的更新只用到了 f[i-1] [j]，即 f[i-2][j] 到 f[0][j] 是没有用的。
所以第二层循环可以直接从v[i] 开始。

二维优化到一维后：

　　如果删掉f[i]这一维，结果如下：如果j层循环时递增的，则是错误的

为什么一维情况下枚举背包容量需要逆序?

　　在二维情况下，状态f[i][j]是由上一轮i - 1的状态得来的，f[i][j]与f[i - 1][j]是独立的。而优化到一维后，如果我们还是正序，则有f[较小体积]更新到f[较大体积]，则有可能本应该用第i-1轮的状态却用的是第i轮的状态。

　　例如，一维状态第i轮对体积为 3 的物品进行决策，则f[7]由f[4]更新而来，这里的f[4]正确应该是f[i - 1][4]，但从小到大枚举j这里的f[4]在第i轮计算却变成了f[i][4]。当逆序枚举背包容量j时，我们求f[7]同样由f[4]更新，但由于

是逆序，这里的f[4]还没有在第i轮计算，所以此时实际计算的f[4]仍然是f[i - 1][4]。

完全背包问题

描述

设有n种物品，每种物品有一个重量及一个价值。但每种物品的数量是无限的，同时有一个背包，最大载重量为M，今从n种物品中选取若干件(同一种物品可以多次选取)，使其重量的和小于等于M，而价值的和为最大。

输入描述

第一行：两个整数，M(背包容量，M≤200)和N(物品数量，N≤30)；

第2..N+1行：每行二个整数Wi,Ci，表示每个物品的重量和价值。

输出描述

仅一行，一个数，表示最大总价值。

完全背包问题和01背包问题很相似。
01背包问题每个物品只能选一个，而完全背包问题每个物品可以选无限次。
DP问题的关键是找到状态转移方程：
①定义f[i][j]表示从前 i 个物品中选择,体积为 j 的时候的最大价值。
②那么转移方程f[i][j] = max(f[i - 1][j],f[i - 1][j - v[i]],f[i - 1][j - 2 * v[i]],.....,f[i - 1][j - k * v[i]],....)
因此代码就是：

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

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

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

}

由于数据量级的原因，此代码肯定会发生TLE，因此需要进行优化。
f[i , j] = max(f[i-1,j] , f[i- 1,j - v[i]+ w[i] , f[i - 1,j-2 * v[i]]+2 * w[i] , f[i -1,j - 3 * v[i]]+3 * w[i] , ...)
f[i , j - v[i]]= max(f[i - 1,j - v[i]] ,   f[i - 1,j - 2 * v[i]] + w[i] , f[i - 1,j- 3 * v[i]]+2 * w[i] , ...)
由上两式，可得出如下递推关系：
f[i][j]=max(f[i, j - v[i]] + w[i] , f[i - 1][j])

因此优化后的代码变为：

```
#include
using namespace std;
const int N=1010;
int v[N],w[N],f[N][N];
int n,m;

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

 for(int i = 1;i <= n;i++){
   scanf("%d%d",&v[i],&w[i]);
 }
 for(int i = 1;i <= n;i++)
  for(int j = 1;j <= m;j++){
   f[i][j] = f[i - 1][j];
   if(j >= v[i])
    f[i][j] = max(f[i][j],f[i][j - v[i]] + w[i]);
  }

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

```
#include
using namespace std;
```

```
const int N=1010;
int v[N],w[N],f[N];
int n,m;

int main()
{
  scanf("%d%d",&n,&m);
  for(int i = 1;i <= n;i++){
      scanf("%d%d",&v[i],&w[i]);
  }
  for(int i=1;i<=n;i++)
    for(int j=v[i];j<=m;j++){
        f[j]=max(f[j],f[j-v[i]]+w[i]);
    }
  printf("%d",f[m]);
  return 0;
}
```

分组背包问题

描述

一个旅行者有一个最多能装V公斤的背包，现在有n件物品，它们的重量分别是W_1，W_2，...,W_n，它们的价值分别为$C_1, C_2, ..., C_n$。这些物品被划分为若干组，每组中的物品互相冲突，最多选一件。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

输入描述

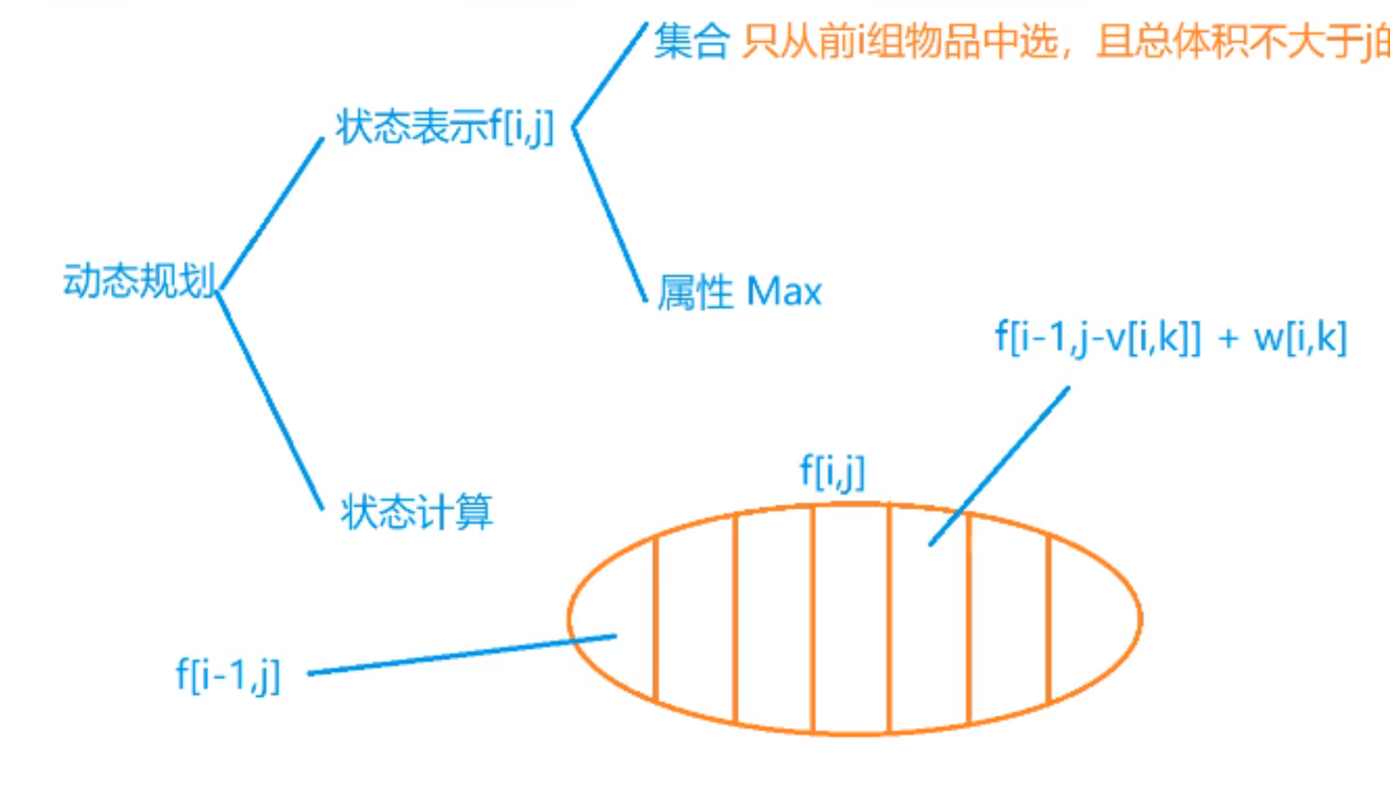
第一行：三个整数，V(背包容量，V≤200)，N(物品数量，N≤30)和T(最大组号，T≤10)；
第 2 到 N+1行：每行三个整数W_i, C_i, P，表示每个物品的重量，价值，所属组号。

输出描述

仅一行，一个数，表示最大总价值。

和01背包相似

题解


动态规划
状态表示f[i,j]
集合 只从前i组物品中选，且总体积不大于j的
属性 Max
状态计算
f[i-1,j-v[i,k]] + w[i,k]
f[i,j]
f[i-1,j]

优化

Fate

描述

最近xhd正在玩一款叫做FATE的游戏，为了得到极品装备，xhd在不停的杀怪做任务。久而久之xhd开始对杀怪产生的厌恶感，但又不得不通过杀怪来升完这最后一级。现在的问题是，xhd升掉最后一级还需n的经验值，xhd还留有m的忍耐度，每杀一个怪xhd会得到相应的经验，并减掉相应的忍耐度。当忍耐度降到0或者0以下时，xhd就不会玩这游戏。xhd还说了他最多只杀s只怪。请问他能升掉这最后一级吗?

输入描述

输入数据有多组，对于每组数据第一行输入n，m，k，s(0 < n,m,k,s < 100)四个正整数。分别表示还需的经验值，保留的忍耐度，怪的种数和最多的杀怪数。接下来输入k行数据。每行数据输入两个正整数a，b(0 < a,b < 20)；分别表示杀掉一只这种怪xhd会得到的经验值和会减掉的忍耐度。(每种怪都有无数个)

输出描述

输出升完这级还能保留的最大忍耐度，如果无法升完这级输出-1。

代码:

```
//
// lis.cpp
// c++
//
// Created by 语何 on 2023/8/2.
//


#include <stdio.h>
#include
#include <math.h>
using namespace std;
```

```
int v[10005],w[100005],u[100005];
int bi[100005],dp[200][200];
int main(){
 int i,j,k,l,n,m,s;
 while(~scanf("%d %d %d %d",&n,&m,&k,&s)){
  for(int i=1;i<=k;i++){
   scanf("%d %d",&v[i],&w[i]);
  }
  for(int i=1;i<=k;i++){
   for(j=w[i];j<=m;j++){
    for(l=1;l<=s;l++){
     dp[j][l]=max(dp[j][l],dp[j-w[i]][l-1]+v[i]);
    }
   }
  }
      int ww,kk=-1;
  for(i=1;i<=m;i++){
   for(int j=1;j<=s;j++){
    if(dp[i][j]>=n){
     ww=m-i;
     kk=max(ww,kk);
    }
    dp[i][j]=0;
   }
  }
  printf("%d\n",kk);
 }
 return 0;
}
```

```
#include<stdio.h>
#include<algorithm>
#include<math.h>
```

```
using namespace std;
int v[100005],w[100005],u[100005];
int bi[100005],dp[200][200];
int main()
{
    int i,j,k,l,n,m,s;
    while(~scanf("%d %d %d %d",&n,&m,&k,&s))
    {
        for(i=1;i<=k;i++)
        {
            scanf("%d %d",&v[i],&w[i]);
        }
        for(i=1;i<=k;i++)
        {
            for(j=w[i];j<=m;j++)
            {
                for(l=1;l<=s;l++)
                {
                    dp[j][l]=max(dp[j][l],dp[j-w[i]][l-1]+v[i]);
                }
            }
        }
        int ww,kk=-1;
        for(i=1;i<=m;i++)
        {
            for(j=1;j<=s;j++)
            {
                if(dp[i][j]>=n)
                {
                    ww=m-i;
                    kk=max(ww,kk);
                }
                dp[i][j]=0;
            }
        }
        printf("%d\n",kk);
    }
}
```

你已经解决了该问题

在线自测

提交评测

饭卡

电子科大本部食堂的饭卡有一种很诡异的设计，即在购买之前判断余额。如果购买一个商品之前，卡上的剩余金额大于或等于5元，就一定可以购买成功（即使购买后卡上余额为负），否则无法购买（即使金额足够）。所以大家都希望尽量使卡上的余额最少。
某天，食堂中有n种菜出售，每种菜可购买一次。已知每种菜的价格以及卡上的余额，问最少可使卡上的余额为多少。

输入描述

多组数据。对于每组数据：
第一行为正整数n，表示菜的数量。n<=1000。
第二行包括n个正整数，表示每种菜的价格。价格不超过50。
第三行包括一个正整数m，表示卡上的余额。m<=1000。

n=0表示数据结束。

输出描述

对于每组输入,输出一行,包含一个整数，表示卡上可能的最小余额。

题意：电子科大的食堂的饭卡有比较特殊，当余额小于5元时，不能购饭。大于等于五元的时候就算余额不足，也可以买饭。求饭卡里的最少余额。
思路：我们可以找到最贵的饭，当余额最小且大于等于5元的时候，我们就用大于等于5元购买这个饭了。因此，我们现在要做的就是**求 （余额-5）能买能花最多的钱，因此就问题就转换成 01背包问题了。最后剩余的余额再减去最大值即可。**

```
#include
#include
#include
using namespace std;
int num[1010],dp[1010];
int main()
{
    int n,m,i,j,k;
    while(~scanf("%d",&n)&&n)
    {
        int maxx=-1;
        for(i=0;i<n;i++)
        {
            scanf("%d",&num[i]);
```

```
        if(num[i]>maxx)       //最贵的饭
            k=i,maxx=num[i];
    }
    scanf("%d",&m);
    if(m<5)            //小于5的时候不能买饭
    {
        printf("%d\n",m);
        continue;
    }
    num[k]=0;              //最贵的饭定为0元 不影响总金额
    memset(dp,0,sizeof(dp));  //01背包
    for(i=0;i<n;i++)
    {
        if(i==k)           //最贵的饭先不算
            continue;
        for(j=m-5;j>=num[i];j--)
        {
            dp[j]=max(dp[j],dp[j-num[i]]+num[i]);
        }
    }
    printf("%d\n",m-dp[m-5]-maxx);
  }
}
```

```
#include<stdio.h>
#include<iostream>
#include<algorithm>
using namespace std;
long long  dp[10005];
long long m[10005];
long long val[10005];
long long last[10005];

int main()
{
    long long i,j,k,l,n,mm,top=0,t;
    while(1)
    {
    scanf("%lld",&n);
    if(n==0)
    break;
    //scanf("%lld",&mm);
    for(i=1;i<=n;i++)
    {
        scanf("%lld",&m[i]);
```

```
	}
	sort(m+1,m+n+1);
//	sort(val+1,val+n+1);
	scanf("%lld",&mm);
	if(mm<5)
	{
	printf("%lld\n",mm);
	continue;
	}
	for(i=1;i<n;i++)
	{
		for(j=mm-5;j>0;j--)
		{
			if(j>=m[i])
			dp[j]=max(dp[j],dp[j-m[i]]+m[i]);
		}
	}
//	printf("%lld %lld %lld\n",mm,dp[mm-5],m[n]);
	printf("%lld\n",mm-dp[mm-5]-m[n]);
	for(i=1;i<=mm;i++)
	{
		m[i]=0;
		dp[i]=0;

	}
//	printf("%lld",dp[mm]);
	}
	return 0;
}
```

你已经解决了该问题

在线自测

提交评测

```
#include
#include
#include<string.h>
using namespace std;
int dp[1005];
```

int main()
{

```
int n;
int p[1005];
while (cin >> n && n)//先减去5(保证最后剩的钱大于5)，然后按01背包问题一样选择背包能装的最多的物件（避开了最重值），最后再加上减少的5再减去最贵的商品
{
    memset(dp, 0, sizeof(dp));
    memset(p, 0, sizeof(p));
    for (int i = 1; i <= n; i++)
    {
        cin >> p[i];//01背包问题
    }
    sort(p + 1, p + 1 + n);  //升序排序
    int max1 = p[n];//寻找数组中的最大值，这是最后减成负数值

    int all;//总金额
    cin >> all;

    if (all < 5)
    {
        cout << all<<endl;
        continue;
    }

    all = all - 5;
    for (int i = 1; i <= n-1 ; i++)//n-1就避开了最大值，成了01背包问题
    {
        for (int j = all; j >= p[i]; j--)
        {
                dp[j] = max(dp[j], dp[j - p[i]] + p[i]);
        }
    }
    cout << all + 5 - dp[all] - max1<<endl;//dp[n-1][all]这里的n也要减一
}
```

}

装箱问题

描述

有一个箱子容量为V（正整数，0≤v≤20000），同时有n个物品（0< n ≤30），每个物品有一个体积（正整数）。

要求n个物品中，任取若干个装入箱内，使箱子的剩余空间为最小。

输入描述

第一行是一个整数V，表示箱子容量。

第二行是一个整数n，表示物品数。

接下来n行，每行一个正整数（不超过10000），分别表示这n个物品的各自体积。

输出描述

一个整数，表示箱子剩余空间。

```
#include<stdio.h>
#include<algorithm>
using namespace std;
int dp[200005];
int main(){
    int i,j,k,l,n,m,w;
    scanf("%d %d",&m,&n);
    dp[0]=1;
    for(i=1;i<=n;i++){
        scanf("%d",&w);
        for(j=m;j>=w;j--){
            dp[j]|=dp[j-w];
        }
    }
    for(i=m;i>=0;i--){
        if(dp[i]){
            printf("%d",m-i);
            break;
        }
    }
}
```


关于这一题，有两种解法，一种是利用一维数组dp[]来解决，另一种是利用二维数组dp[][]来解决，我们先来讲第二种，比较好理解

dp[i][j]表示在前i个物品在容量为j的体积中所存放的最大价值，只不过这里的最大价值等价与最大体积，我们在这里应该知道这应该是一个背包问题，对于一个物品，存在着两种情况，放与不放，如果不放则dp[i][j] == dp[i-1][j]，因为你们没有放物体，则在逻辑上等价于前一物体，如果放则dp[i][j] == dp[i-1][j-a[i]]+a[i]，在前一状态的基础上，加入物体，剩余空间必然减少，价值增大，但是两者又是等价，则可以如是，这一点很难想，也应该是我

们所说的背包问题，具体问题请看代码：

```
#include
using namespace std;
int v, n, a[31], dp[31][10000];
/*
    dp[i][j]表示前i个物品放入容量为j的背包中所得到的最大值
    采用二维数组更好理解一些，但是其空间与时间的开销更大
*/
int main()
{
    int i, j;
    cin>>v>>n;
    for(i = 1; i <= n; i++)
        cin>>a[i];
    for(i = 1; i <= n; i++)
        dp[i][0] = 0;
    for(j = 1; j <= v; j++)
        dp[0][j] = 0;
    for(i = 1; i <= n; i++)
    {
        for(j = 1; j <= v; j++)
        {
            if(j < a[i])//装不下
                dp[i][j] = dp[i-1][j];
            else
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-a[i]]+a[i]);
        }
    }
    printf("%d", v-dp[n][v]);
    return 0;
}
```

但是这会超时，只有80%的正确率。

采用一种更简单的方式，只需要用一维数组即可，dp[j]表示在容量剩余为j的体积中所能装入的最大价值，同样有放与不放两种状态，不放则不变即dp[j] == dp[j]，放则dp[j] = dp[j-a[i]]+a[i]，也是属于背包问题，剩余容量减少，但是价值增大，两者也是等价，具体问题请看代码:

```
#include
#include
using namespace std;
int v, n, a[31], dp[20005];
int main()
{
    int i, j;
    cin>>v>>n;
    memset(dp, 0, sizeof(dp));
    for(i = 0; i < n; i++)
```

```
    {
        cin>>a[i];
        /*
            针对一个物品只有两种状态，装与不装，这里的j表示剩余容量
            如果装箱，则背包价值为dp[j-a[i]]+a[i]，如果不装箱，则为dp[j]
            所以dp[j] = max(dp[j], dp[j-a[i]]+a[i])(伪背包问题)
            //不太好理解可以采用二维数组解题
        */
        for(j = v; j >= a[i]; j--)
        {
            dp[j] = max(dp[j], dp[j-a[i]]+a[i]);
        }
    }
    printf("%d", v-dp[v]);
    return 0;
}
```

2023.8.3

kmp与manacher

最长回文

描述

给出一个只由小写英文字符a,b,c...y,z组成的字符串S,求S中最长回文串的长度.
回文就是正反读都是一样的字符串,如aba, abba等

输入描述

输入有多组case,不超过120组,每组输入为一行小写英文字符a,b,c...y,z组成的字符串S
两组case之间由空行隔开(该空行不用处理)
字符串长度len <= 110000

输出描述

每一行一个整数x,对应一组case,表示该组case的字符串中所包含的最长回文长度.

本题有三种解法

解法一：暴力 时间复杂度：o(n^3)

解法二：dp

令dp[i][j]表示S[i]至S[j]所表示的子串是否是回文子串，是则为1，不是为0。这样根据S[i]是否等于S[j]，可以把转移情况分为两类：
①若S[i]=S[j],那么只要S[i+1]和S[j-1]是回文子串，S[i+1]至S[j-1]就是回文子串；如果S[i+1]至S[j-1]不是回文子串，则S[i]至S[j]一定不是回文子串。
②若S[i]!=S[j]，那S[i]至S[j]一定不是回文子串。
由此可以写出状态转移方程

$$dp[i][j]=\begin{cases} dp[i+1][j-1], S[i]==S[j] \\ 0, S[i]!=S[j] \end{cases}$$

到这里还有一个问题没有解决,那就是如果按照i和j从小到大的顺序来枚举子串的两个端点,然后更新dp[i]lj],会无法保证dp[i + 1][- 1]已经被计算过,从而无法得到正确的dp[i][i]。
如图11-4所示，先固定i=0,然后枚举j从2开始。当求解dp[0][2]时，将会转换为dp[1][],而dp[1][1]是在初始化中得到的;当求解dp[0][3]时,将会转换为dp[1][2], 而dp[1][2]也是在初始化中得到的;当求解dp[0][4]时，将会转换为dp[1][3], 但是dp[1][3]并不是已经计算过的值，因此无法状态转移。事实上，无论对ij和j的枚举顺序做何调整，都无法调和这个矛盾，因此必须想办法寻找新的枚举方式。
根据递推写法从边界出发的原理，注意到边界表示的是长度为1和2的子串，且每次转移时都对子串的长度减了1,因此不妨考虑按子串的长度和子串的初始位置进行枚举，即第一遍将长度为3的子串的dp值全部求出，第二遍通过第一遍结果计算出长度为4的子串的dp值...这样就可以避免状态无法转移的问题。如图11-5所示，可以先枚举子串长度L (注意: L是可以取到整个字符串的长度S.len()的)，再枚举左端点i,这样右端点i+ L- 1也可以直接得到。

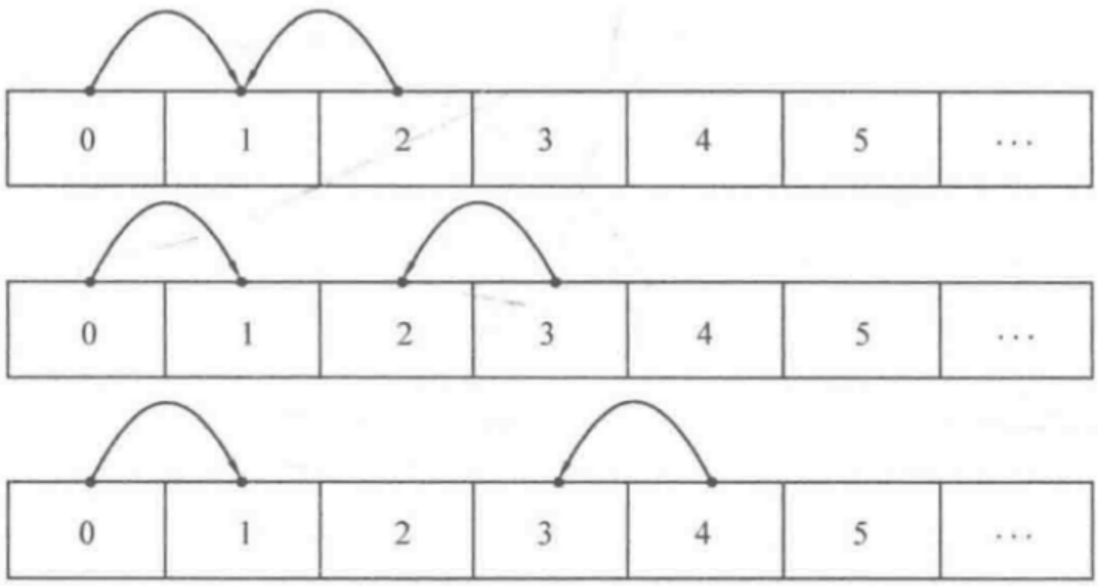


图 11-4　最长回文子串示意图

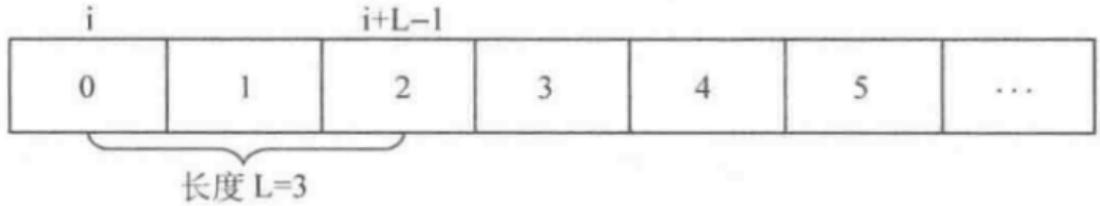


图 11-5　枚举 L 的最长回文子串做法示意图


```
#include
#include
#include
#include
#include
#include<string.h>
#include
#include<math.h>
#include
#define llu unsigned long long
using namespace std;

int dp[1010][1010];
int main()
{
    string s1;
    int ans=1;
    getline(cin,s1);
    //cout << s1 << endl ;
    int n=s1.size();
    for(int i=0;i<n;i++)
```

```
{
    dp[i][i]=1;
    if(i<n-1){
        if(s1[i]s1[i+1]){
            dp[i][i+1]=1;
            ans=2;
        }
    }
}
for(int L=3;L<=n;L++){
    for(int i=0;i+L-1<n;i++)
    {
        int j=i+L-1;
        if(s1[i]s1[j]&&dp[i+1][j-1]==1)
        {
            dp[i][j]=1;
            ans=L;
        }
    }
}
cout << ans << endl ;
```

```
return 0;
```

}

这个马拉车算法 Manacher‘s Algorithm 是用来查找一个字符串的最长回文子串的线性方法，由一个叫 Manacher 的人在 1975 年发明的，这个方法的最大贡献是在于将时间复杂度提升到了线性，这是非常了不起的。对于回文串想必大家都不陌生，就是正读反读都一样的字符串，比如 "bob", "level", "noon" 等等，那么如何在一个字符串中找出最长回文子串呢，可以以每一个字符为中心，向两边寻找回文子串，在遍历完整个数组后，就可以找到最长的回文子串。但是这个方法的时间复杂度为 O(n*n)，并不是很高效，下面我们来看时间复杂度为 O(n)的马拉车算法。

由于回文串的长度可奇可偶，比如 "bob" 是奇数形式的回文，"noon" 就是偶数形式的回文，马拉车算法的第一步是预处理，做法是在每一个字符的左右都加上一个特殊字符，比如加上 '#'，那么

bob --> #b#o#b#

noon --> #n#o#o#n#

这样做的好处是不论原字符串是奇数还是偶数个，处理之后得到的字符串的个数都是奇数个，这样就不用分情况讨论了，而可以一起搞定。接下来我们还需要和处理后的字符串t等长的数组p，其中 p[i] 表示以 t[i] 字符为中心的回文子串的半径，若 p[i] = 1，则该回文子串就是 t[i] 本身，那么我们来看一个简单的例子：

1 # 2 # 2 # 1 # 2 # 2
1 2 1 2 5 2 1 6 1 2 3 2 1

为啥我们关心回文子串的半径呢？看上面那个例子，以中间的 '1' 为中心的回文子串 "#2#2#1#2#2#" 的半径是6，而未添加#号的回文子串为 "22122"，长度是5，为半径减1。这是个普遍的规律么？我们再看看之前的那个 "#b#o#b#"，我们很容易看出来以中间的 'o' 为中心的回文串的半径是4，而 "bob"的长度是3，符合规律。再来看偶数个的情况 "noon"，添加#号后的回文串为 "#n#o#o#n#"，以最中间的 '#' 为中心的回文串的半径是5，而 "noon" 的长度是4，完美符合规律。所以我们只要找到了最大的半径，就知道最长的回文子串的字符个数了。只知道长度无法定位子串，我们还需要知道子串的起始位置。

我们还是先来看中间的 '1' 在字符串 "#1#2#2#1#2#2#" 中的位置是7，而半径是6，貌似 7-6=1，刚好就是回文子串 "22122" 在原串 "122122" 中的起始位置1。那么我们再来验证下 "bob"，"o" 在 "#b#o#b#" 中的位置是3，但是半径是4，这一减成负的了，肯定不对。所以我们应该至少把中心位置向后移动一位，才能为0啊，那么我们就需要在前面增加一个字符，这个字符不能是#号，也不能是s中可能出现的字符，所以我们暂且就用美元号吧，毕竟是博主最爱的东西嘛。这样都不相同的话就不会改变p值了，那么末尾要不要对应的也添加呢，其实不用的，不用加的原因是字符串的结尾标识为 '\0'，等于默认加过了。那此时 "o" 在 "
Error: You can't use 'macro parameter character #' in math mode#1#2#2#1#2#2#" 中的位置是8，而半径是6，这一减就是2了，而我们需要的是1，所以我们要除以2。之前的 "bob" 因为相减已经是0了，除以2还是0，没有问题。再来验证一下 "noon"，中间的 '#' 在字符串 "$#n#o#o#n#" 中的位置是5，半径也是5，相减并除以2还是0，完美。可以任意试试其他的例子，都是符合这个规律的，最长子串的长度是半径减1，起始位置是中间位置减去半径再除以2。

那么下面我们就来看如何求p数组，需要新增两个辅助变量 mx 和 id，其中 id 为能延伸到最右端的位置的那个回文子串的中心点位置，mx 是回文串能延伸到的最右端的位置，需要注意的是，这个 mx 位置的字符不属于回文串，所以才能用 mx-i 来更新 p[i] 的长度而不用加1，由 mx 的更新方式 mx = i + p[i] 也能看出来 mx 是不在回文串范围内的，这个算法的最核心的一行如下：

```
p[i] = mx > i ? min(p[2 * id - i], mx - i) : 1;
```

可以这么说，这行要是理解了，那么马拉车算法基本上就没啥问题了，那么这一行代码拆开来看就是

如果 mx > i, 则 p[i] = min(p[2 * id - i] , mx - i)

否则，p[i] = 1

当 mx - i > P[j] 的时候，以 S[j] 为中心的回文子串包含在以 S[id] 为中心的回文子串中，由于 i 和 j 对称，以 S[i] 为中心的回文子串必然包含在以 S[id] 为中心的回文子串中，所以必有 P[i] = P[j]，其中 j = 2*id - i，因为 j 到 id 之间到距离等于 id 到 i 之间到距离，为 i - id，所以 j = id - (i - id) = 2*id - i，参见下图。

当 P[j] >= mx - i 的时候，以 S[j] 为中心的回文子串不一定完全包含于以 S[id] 为中心的回文子串中，但是基于对称性可知，下图中两个绿框所包围的部分是相同的，也就是说以 S[i] 为中心的回文子串，其向右至少会扩张到 mx 的位置，也就是说 P[i] = mx - i。至于 mx 之后的部分是否对称，就只能老老实实去匹配了，这就是后面紧跟到 while 循环的作用。

对于 mx <= i 的情况，无法对 P[i] 做更多的假设，只能 P[i] = 1，然后再去匹配了。

参见如下实现代码：

```
#include <vector>
#include <iostream>
#include <string>

using namespace std;

string Manacher(string s) {
    // Insert '#'
    string t = "$#";
    for (int i = 0; i < s.size(); ++i) {
        t += s[i];
        t += "#";
    }
    // Process t
    vector<int> p(t.size(), 0);
    int mx = 0, id = 0, resLen = 0, resCenter = 0;
    for (int i = 1; i < t.size(); ++i) {
        p[i] = mx > i ? min(p[2 * id - i], mx - i) : 1;
        while (t[i + p[i]] == t[i - p[i]]) ++p[i];
        if (mx < i + p[i]) {
            mx = i + p[i];
            id = i;
        }
        if (resLen < p[i]) {
            resLen = p[i];
            resCenter = i;
        }
    }
    return s.substr((resCenter - resLen) / 2, resLen - 1);
}

int main() {
    string s1 = "12212";
    cout << Manacher(s1) << endl;
    string s2 = "122122";
    cout << Manacher(s2) << endl;
    string s = "waabwswfd";
    cout << Manacher(s) << endl;
}
```

2023.8.4

高精度（上午）和ac自动机（下午）

ac自动机：建立在树状数组上面的kmp

2023.8.7

树状dp

2023.8.8

矩阵快速幂

矩阵快速幂：快速幂的思想就是优先给底数平方，再把幂除2，例如：2的12次方就是4的6次方，就把时间复杂度优化了一半