2000	计算机器的改良	字符数组
	税收与补贴问题	数组处理
	乘积最大	动态规划
	单词接龙	动态规划
2001	数的记数	递推
	最大公约数和最小公	数理逻辑
	求先序排列	递归
	装箱问题	0/1背包问题
2002	级数求和	累加器
	选数	回溯法、全排列
	产生数	递推
	过河卒	递推
2003	乒乓球	字符数组
	数字游戏	动态规划
	栈	递推
	麦森数	高精度乘法
2004	不高兴的规律	一般数据处理
	花生采摘	桶排序
	FBI 树	递归
	火星人	全排列
2005	陶陶摘苹果	一堆数组处理
	校门外的树	桶排序或数组处理
	采药	0/1背包问题
	循环	递推加递归
2006	明明的随机数	桶排序
	开心的金明	0/1背包问题
	jam的计数	全排列
	数列	递推加递归
2007	奖学金	数组处理
	纪念品分组	桶排序、贪心
	守望者的逃离	数组处理
	hanoi双塔问题	递推
2008	ISBN号码	字符处理
	排座椅	数组处理（较复杂）
	传球游戏	递推
	立体图	递归
2009	多形式输出	数据处理
	分数线划定	快排
	细胞分裂	分析、数论算法
	道路游戏	动态规划
2010	数字统计	数组
	接水问题	数组处理
	导弹拦截	贪心+枚举
	三国游戏	分析关系，找次大值，贪心
2011	数字反转	字符数组
	统计单词数	字符数组
	瑞士轮	快排
	表达式的值	字符处理、递归

2012	质因数分解	质数
	寻宝	数组、模拟
	摆花	深搜加优化
	文化之旅	DFS+优化+剪枝
2013	计数问题	模拟、枚举
	表达式求值	模拟、字符串、栈
	小朋友的数字	动态规划
	车站分级	图论、贪心、拓扑排序
2014	珠心算测验	模拟、枚举
	比例简化	枚举优化、数论
	螺旋矩阵	模拟、数论
	子矩阵	动态规划、搜索、剪枝
2015	金币	模拟、数论
	扫雷游戏	模拟、矩阵、字符串
	求和	排序、前缀和
	推销员	贪心、线段树、树状数组
2016	买铅笔	模拟
	回文日期	枚举
	海港	模拟
	魔法阵	枚举优化、排序、数论
2017	成绩	模拟
	图书管理员	模拟、字符串
	棋盘	深搜、最短路
	跳房子	二分答案+dp、单调队列优化
2018	标题统计	模拟、字符串
	龙虎斗	模拟、枚举
	摆渡车	动态规划
	对称二叉树	树形结构、哈希
2019CSP	**数字游戏**	**字符串入门题**
	公交换乘	**模拟**
	纪念品	**完全背包**
	数字游戏	**广搜/最短路**
2020CSP	**优秀的拆分**	**二进制转化**
	直播获奖	**桶排序**
	表达式	**二叉树表达式、位运算**
	方格取数	**动态规划**

Day3：贪心&动态规划

引例：【排队打水问题】

有n个人排队到r个水龙头打水，他们装满水桶的时间为t1，t2，...，tn为整数且各不相等，应该如何安排他们打水顺序才能使他们总共花费的时间最少？

输入格式：

第1行，n和r（n<=500,r<=75）

第2行，n个人打水所用的时间Ti（Ti<=100）

输入样例：

3 2

1 2 3

输出格式：

最少花费时间。

输出样例：

7

分析：【排队打水问题】

因为总的等待时间和大家打水顺序是息息相关的。由于排队时，对于每一个水龙头，后面打水的人都需要等待前面的打水人，因此越靠前面的打水的人被重复计算的次数就越多，显然越小的排在越前面得出的结果越小。

解题步骤如下：

1、将输入的时间按照从小到大顺序排列。

2、将排序后的时间按照顺序依次放入每个水龙头的队列中。（当轮到你的时候，你当然更希望找个整体等待时间最少的水龙头排队咯）

3、统计，输出答案。

注意：总花费时间=每个人花费时间的总和。

参考程序：【排队打水问题】

```cpp
#include<bits/stdc++.h>
using namespace std;
int a[501]={0},v[501]={0};
int main(){
    int n,r,sum=0;
    scanf("%d%d",&n,&r);
    for(int i=0;i<n;i++) scanf("%d",&a[i]);
    sort(a,a+n);          //从小到大排序
    for(int i=0;i<n;i++)
    {
        sort(v,v+r);      //每次计算总花费时间后，水龙头前置时间都要重新排序
        sum+=v[0]+a[i]; //当前时间+前置等待时间
        v[0]+=a[i];       //更新前置等待时间
    }
    printf("%d\n",sum);
    return 0;
}
```

贪心算法

顾名思义，贪心算法总是作出在当前看来最好的选择。也就是说贪心算法并不从整体最优考虑，它所作出的选择只是在某种意义上的**局部最优**选择。当然，希望贪心算法得到的最终结果也是整体最优的。虽然贪心算法不能对所有问题都得到整体最优解，但对许多问题它能产生整体最优解。如单源最短路经问题，最小生成树问题等。在一些情况下，即使贪心算法不能得到整体最优解，其最终结果却是最优解的很好近似。

贪心算法的基本要素

对于一个具体的问题，怎么知道是否可用贪心算法解此问题，以及能否得到问题的最优解呢?这个问题很难给予肯定的回答。

但是，从许多可以用贪心算法求解的问题中看到这类问题一般具有2个重要的性质：贪心选择性质和最优子结构性质。

贪心算法的基本要素

1. 贪心选择性质

所谓**贪心选择性质**是指所求问题的**整体最优解**可以通过一系列**局部最优**的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。

动态规划算法通常以**自底向上**的方式解各子问题，而贪心算法则通常以**自顶向下**的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题。

对于一个具体问题，要确定它是否具有贪心选择性质，必须证明每一步所作的贪心选择最终导致问题的整体最优解。

贪心算法的基本要素

2. 最优子结构性质

当一个问题的最优解包含其子问题的最优解时，称此问题具有**最优子结构性质**。问题的最优子结构性质是该问题可用动态规划算法或贪心算法求解的关键特征。

样例：活动安排问题

活动安排问题就是要在所给的活动集合中选出最大的相容活动子集合，是可以用贪心算法有效求解的很好例子。该问题要求高效地安排一系列争用某一公共资源的活动。贪心算法提供了一个简单、漂亮的方法使得尽可能多的活动能兼容地使用公共资源。

活动安排问题

设有n个活动的集合$E=\{1,2,\ldots,n\}$，其中每个活动都要求使用同一资源，如演讲会场等，而在同一时间内只有一个活动能使用这一资源。

每个活动i都有一个要求使用该资源的起始时间si和一个结束时间fi,且si <fi 。如果选择了活动i，则它在半开时间区间[si, fi)内占用资源。

若区间[si, fi)与区间[sj, fj)不相交，则称活动i与活动j是相容的。也就是说，当si≥fj或sj≥fi时，活动i与活动j相容。

活动安排问题就是要在所给的活动集合中选出最大的相容活动子集合。

活动安排问题

- 算法分析:
  - 本题是要选出最多的活动个数，使得这些活动在使用时间上不会冲突。容易想到如下做法:
    - 将所有活动按结束时间从小到大排序。（贪心）
    - 选择第1个活动，做为要选择的活动
    - 在与活动1不冲突活动中，再选择出结束最早的活动，做为第2个要选择的活动。
    - ……
    - 直到所有活动被扫描结束。

活动安排问题

样例代码

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

struct activity{
    int s,f;  //结构体数组存放每个活动的开始与结束时间
}a[100001];
int k=0; //活动编号

int greedyselect()//贪心算法，返回答案count
{
    int j=0,count=1;//选择第一个活动，0号活动，活动个数为1
    for(int i=1;i<k;i++)
    {
        if(a[i].s>=a[j].f)
        {
            j=i;count++;//活动不冲突，答案加1，更新当前活动编号
        }
    }
    return count;
}
bool cmp(activity m,activity n){return m.f<n.f;}
int main(){
    int x,y,ans;
    scanf("%d%d",&x,&y);
    while(x!=0||y!=0){ //读入所有活动的开始与结束时间，存入数组a
        a[k].s=x;a[k++].f=y;
        scanf("%d%d",&x,&y);
    }
    sort(a,a+k,cmp);//将a数组按照结束时间从小到大排序
    printf("%d",greedyselect());
    return 0;
}
```

活动安排问题

由于输入的活动以其完成时间的**非减序**排列，所以算法greedySelector每次总是选择**具有最早完成时间**的相容活动加入集合A中。直观上，按这种方法选择相容活动为未安排活动留下尽可能多的时间。也就是说，该算法的贪心选择的意义是**使剩余的可安排时间段极大化**，以便安排尽可能多的相容活动。

算法greedySelector的效率极高。当输入的活动已按结束时间的非减序排列，算法只需O(n)的时间安排n个活动，使最多的活动能相容地使用公共资源。如果所给出的活动未按非减序排列，可以用O(nlogn)的时间重排。

活动安排问题

若被检查的活动i的开始时间Si小于最近选择的活动j的结束时间fi，则不选择活动i，否则选择活动i加入集合A中。

贪心算法并不总能求得问题的**整体最优解**。但对于活动安排问题，贪心算法greedySelector却总能求得的整体最优解，即它最终所确定的相容活动集合A的规模最大。这个结论可以用数学归纳法证明。

部分背包问题

（引）最优装载问题

给出n个物体，第i个物体重量为wi。选择尽量多的物体，使得总重量不超过C。

【分析】

由于只关心物体的数量，所以装重的没有装轻的划算。只需要把所有物体按照重量从小到大排序，依次选择每个物体，直到装不下为止。这是一种典型的贪心算法，它只顾眼前，但却能得到最优解。

部分背包问题

部分背包问题

有n个物体，第i个物体的重量为wi，价值为vi。在总重量不超过 C 的情况下让总价值尽量高。每一个物体都可以只取走一部分，价值和重量按比例计算。

【分析】

此问题比上一问题增加了价值，所以不能简单地像上一问题那样先拿轻的（轻的可能价值也小），也不能先拿价值大的（可能它特别重），而应该综合考虑两个因素。一种直观的贪心策略是：

优先拿“价值除以重量的值”最大的，直到重量和正好为C。

注意：由于每一个物体可以只拿一部分，因此一定可以让总重量恰好为C（或者全部拿走重量也不足C），而且除了最后一个外，所有物体要么不拿，要么拿走全部。

部分背包问题

- 用贪心算法解背包问题的基本步骤：
  - 首先计算每种物品单位重量的价值Vi/Wi，然后，依贪心选择策略，将尽可能多的**单位重量价值最高**的物品装入背包。若将这种物品全部装入背包后，背包内的物品总重量未超过C，则选择单位重量价值次高的物品并尽可能多地装入背包。依此策略一直地进行下去，直到背包装满为止。

  - 下面我们用贪心策略实现这个背包问题：

```cpp
#include<bits/stdc++.h>
using namespace std;
struct Good{
    //物品有三个属性，编号id，重量w，价值v
    int id;
    float w,v;
};
//初始化6件物品及它们的重量和价值
Good goods[]={{1,10,12},{2,9,9},{3,11,7},{4,12,9},{5,6,8},{6,3,4}};
bool cmp(Good a,Good b){ return (a.v/a.w>b.v/b.w); }
float totalw=30;//背包总重量

int main(){
    sort(goods,goods+6,cmp);//以物品的价值/重量 的比值降序排列
    float leftw=totalw,totalv=0;//定义剩余背包可承受重量和背包此刻总价值
    for(int i=0;i<6;i++)
    {//如果当前背包所能承受的重量大于i物品的总量，那么把i物品全部放进去
        if(leftw>=goods[i].w){
            leftw-=goods[i].w;
            totalv+=goods[i].v;
            printf("choose good[id=%d],%.1f weight,make %.1f value\n",goods[i].id,goods[i].w,goods[i].v);
        }else{
            //如果不能，那么取当前背包所能承受重量的相应数量物品，价值按照比例来。
            totalv+=leftw/goods[i].w*goods[i].v;//单位重量价值*剩余重量
            printf("choose good[id=%d],%.1f weight,make %.1f value\n",goods[i].id,leftw,leftw/goods[i].w*goods[i].v);
            leftw=0;break;
        }
    }
    printf("max total value:%.1f\n",totalv);
    return 0;
}
```

部分背包问题

乘船问题

有n个人，第i个人重量是wi。每艘船的最大载重量均为C，且最多只能乘两个人。用最少的船装载所有人。

【分析】

考虑最轻的人i，他应该和谁一起坐呢？如果每个人都无法和他一起坐船，则唯一的方法就是每人坐一艘船。否则，他应该选择能和他一起坐船的人中最重的一个j。这样的方法是贪心的，因此它只是让“眼前”的浪费最少。幸运的是，这个贪心策略也是对的，可以用反证法说明。

部分背包问题

反证：

假设这样做不是最好的，那么最好方案中i是什么样的呢？

情况1：i不和任何一个人坐同一艘船。那么可以把j拉过来和他一起坐，总船数不会增加（而且可能减少）。

情况2：i和另外一人k同船。由贪心策略，j是“可以和i一起坐船的人”中最重的。因此k比j轻。把j和k交换后，k所在的船仍然不会超重（因为k比j轻），而i和j所在的船也不会超重。因此所得到的新解也不会更差。

部分背包问题

总结：

由此可见，贪心法不会丢失最优解。

程序实现过程：

1）两个下标i和j分别表示当前考虑的最轻和最重的人。

2）将j往左移动，直到i和j可以工坐一艘船，然后将i加1，j减1。

3）重复上述操作。

整个程序时间复杂度为0（n）。

（真题）：

导弹拦截

活动安排

整数去位

合并果子

样例：【整数去位】

键盘输入一个高精度的正整数N，去掉其中任意M个数字后剩下的数字按照原来左右次序将组成一个新的正整数。编程对给定的N和M寻找一种方案使得剩下的数字组成的新数最小。输出组成的新的正整数。

输入数据均不需判错。如果去掉某几位后得到的新整数开头为0，保留0.

输入格式：

第1行，高精度正整数N（N长度不超过10^6）

第2行，M（0<=M<=N的长度）

输入样例：

82386782

3

输出格式：

去掉M位后的最小新数。

输出样例：

23672

样例：【整数去位】

- 算法分析:
  - 我们先考虑一个数字串删除一位，使得剩余的数字组成的数最小。
    - 为了保证删除1个数字后的数最小，我们按高位→低位的方向搜索递减区间。若不存在递减区间，则删尾数符;否则删递减区间的首字符
  - 这样形成一个新数串。然后回到串首，重复以上操作，直到删除m个数字为止。
- 以上算法的最坏时间复杂度为O（n^2），对于10^6长度的最坏情况，可能会超时。

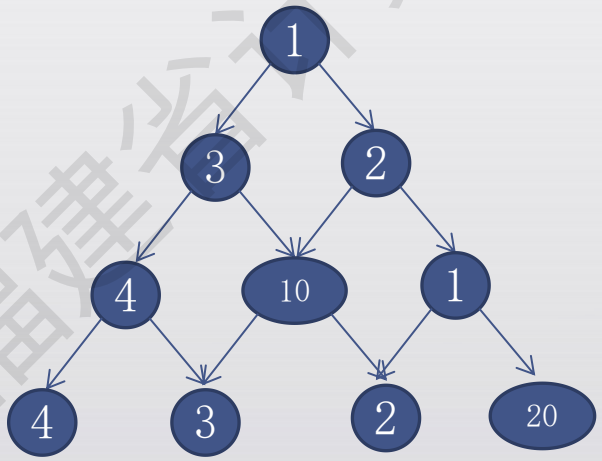
样例：【整数去位】

- 算法分析（优化）：
  - 其实以上做法慢在两个地方：
    - 每删除一个数字，从头重新开始
    - 大字符串中删除一个字符，代价为O（n）
  - 经过分析发现，当找到递减序列头位置时，下一次递减序列头位置，要么就在当前位置，要么在当前位置后面，绝对不会出现在前面。所以，每删除一位后，不必回到首位置。
  - 另外，为了解决大字符串删除一个字符代价大问题。可以采用另设一个字符串栈b，在搜索原字符串a时，当栈b为空或a[i]>=b[top]，将a[i]压入b栈，直到原串搜索至末尾或a[i]<b[top]，删除b[top]字符（这个删除操作只要O（1））。
  - 这样复重以上操作，直到m个数字被删除。输出b栈内容及a串当前位置及之后所有剩余字符。
  - 以上算法时间复杂度为O（m）。

```cpp
#include<bits/stdc++.h>
using namespace std;
char a[1000001],b[1000001];
int main(){
    int len,m,i=0,top=-1;
    scanf("%s%d",a,&m);
    len=strlen(a);
    if(m==0){                    //特殊情况
        puts(a);
        return 0;
    }
    if(m==len) return 0;//特殊情况
    while(m>0){//当m个数字没有完全删除
    //当i+1没有超界并且top>=0同时b[top]<=a[i] 或者top<0时，将a[i]入栈
    while(top<0||i<len&&(b[top]<=a[i]))b[++top]=a[i++];
    top--;m--;//找到一个递减区间的开始位置并出栈
    }
    //输出b栈中的前top个字符，a字符串中的第i到最后字符，就是答案
    for(int k=0;k<=top;k++) printf("%c",b[k]);
    for(int k=i;k<len;k++) printf("%c",a[k]);
    return 0;
}
```

引例：【数字三角形问题】

有一个由非负整数组成的三角形，第一行只有一个数，除了最下行之外每个数的左下方和右下方有一个数。从第一行的数开始，每次可以往左下或者右下走一格，直到走到最下行，把沿途经过的数字全部加起来，如何走才能使得这个和尽量大？

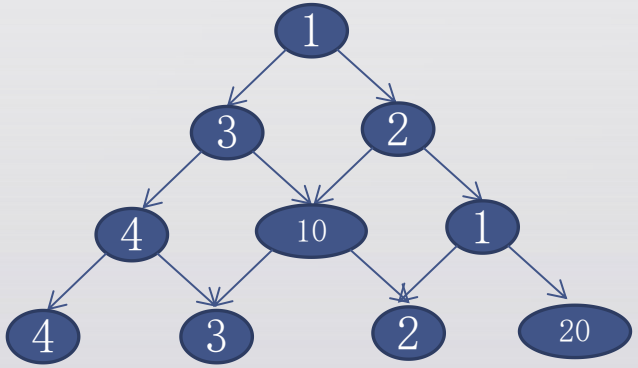


引例：【数字三角形问题】

【分析】

【10分做法】利用贪心从上往下，每一步选择较大数字进行相加。

【30分做法】利用回溯求出所有可能的路线，就可以从中选出最优路线。但是效率太低：一个n层的数字三角形的完整路线有2^n-1条，当n很大时回溯法的速度将让人崩溃。



引例：【数字三角形问题】

【分析】

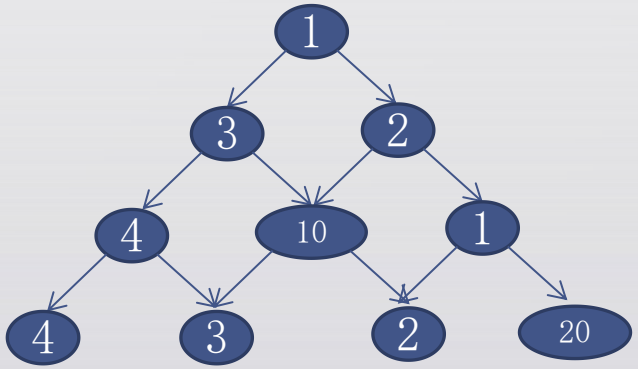
定义d(i,j)为从位置（i，j）出发时能得到的最大和（包含位置（i，j）本身的值a(i,j)）。在这个状态定义下，原问题的解时d(1,1)。

下面我们看一下状态之间是如何转移的。

1）如果下一步向左走需要求出d(i+1,j).

2）如果下一步向右走需要求出d(i+1,j+1).

3）因此选择两者中较大的一个 + a(i,j).



引例：【数字三角形问题】

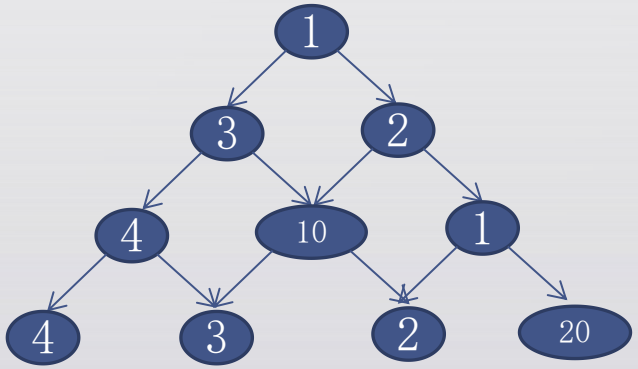
【分析】

因此我们得到了状态转移方程：

$$d(i,j)=a(i,j)+\max\{\ d(i+1,j),\ d(i+1,j+1)\ \};$$

从上式子可以得到结论：全局最优解包含局部最优解。

有了状态转移方程，该怎么计算呢？



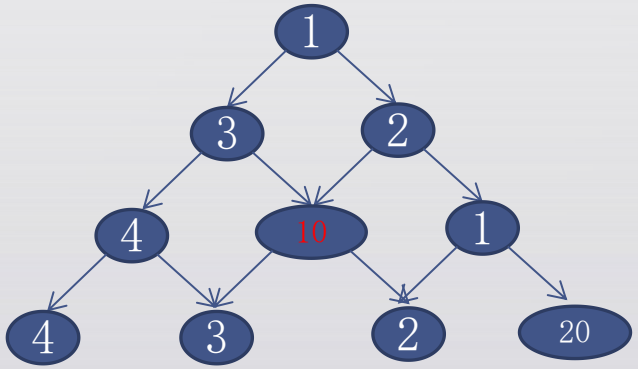
引例：【数字三角形问题】

【计算一】

递归计算：

```
int solve(int i,int j)
{
        return a[i][j]+(i==n? 0:max(solve(i+1,j),solve(i+1,j+1)));
}
```

这样计算会导致相同子问题被多次计算。效率较低！

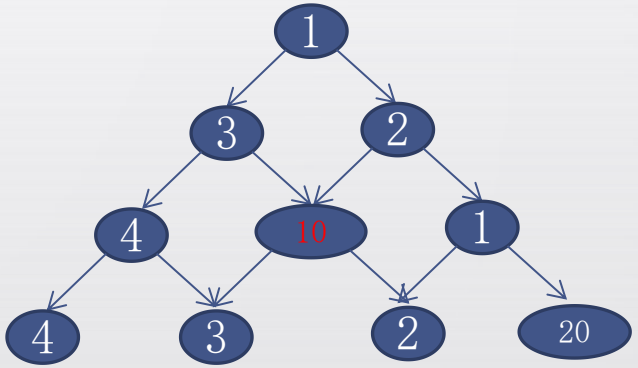


引例：【数字三角形问题】

【计算二】

递推计算：

```
int i, j;
for(j=1; j<=n; j++) d[n][j] = a[n][j];
for(i=n-1; i>=1; i--)
    for(j=1; j<=i; j++)
        d[i][j]=a[i][j] + max(d[i+1][j], d[i+1][j+1]);
```



时间复杂度O(n^2)，可以递推的原因是：i是逆序枚举的，因此在计算d[i][j]前，它所需要的d[i+1][j]和d[i+1][j+1]一定已经计算出来了。

引例：【数字三角形问题】

【计算三】

记忆化搜索：

1）首先“**memset(d,-1,sizeof(d))**;

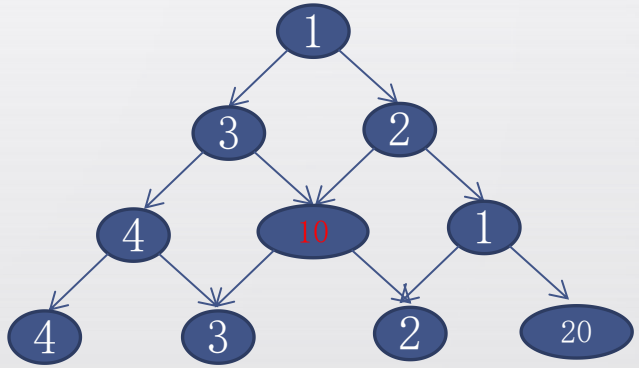
2）int solve (int i, int j)

{

if(d[i][j]>=0) return d[i][j];

return d[i][j]=a[i][j]+(i==n? 0 : max(solve(i+1,j),solve(i+1,j+1)));

}



上述方法称为记忆化搜索，可以保证每个结点只访问一次。

动态规划的基本概念

动态规划的实质是分治思想和解决冗余，因此，动态规划是一种将问题分解为更小的、相似的子问题，并存储子问题的解而避免计算重复的子问题，以解决最后化问题的算法策略。

这类问题会有多种可能的解，每一个解都有一个值，而动态规划找出其中最优（最大或最小）解。若存在多个最优解的话，它只取其中的一个。

动态规划的基本概念

动态规划的实质是分治思想和解决冗余，因此，动态规划是一种将问题分解为更小的、相似的子问题，并存储子问题的解而避免计算重复的子问题，以解决最后化问题的算法策略。

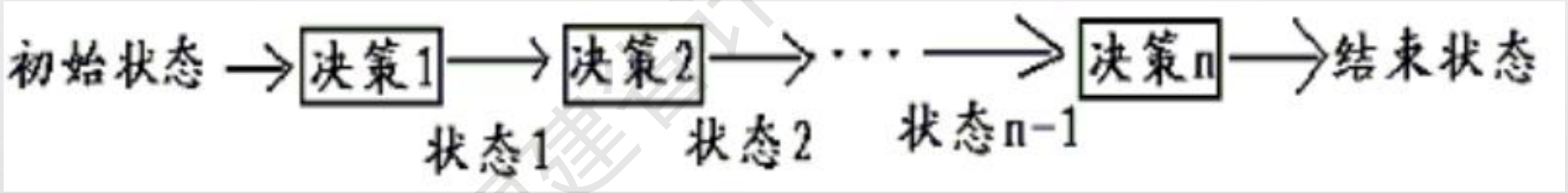
这类问题会有多种可能的解，每一个解都有一个值，而动态规划找出其中最优（最大或最小）解。若存在多个最优解的话，它只取其中的一个。

动态规划的基本概念

动态规划所处理的问题是一个“多阶段决策问题”

目的是得到一个最优解（方案）

大概思想如下图所示：



动态规划的基本概念

1、阶段

用动态规划求解一个问题时，需要将问题的全过程恰当地划分成若干个相互联系的阶段，以便按一定的次序去求解。阶段的划分一般是根据时间和空间的自然特征来定的，一般要便于把问题转化成多阶段决策的过程。

2、状态

状态表示的是事物某一阶段的性质，状态通过一个变量来描述，这个变量称为状态变量。各个状态之间是可以相互转换的。

动态规划的基本概念

3、决策

对问题的处理中做出某种选择性的行动就是决策。一个实际问题可能要有多次决策，在每一个阶段中都需要有一次决策。一次决策就会从一个阶段进入另一个阶段，状态也就发生了转移。

4、策略和最优策略

所有阶段按照约定的决策方法，依次排列构成问题的求解全过程。这些决策的总体称为策略。在实际问题中，从决策允许集合中找出最优效果的策略称为最优策略。

5、状态转移方程

前一阶段的终点就是后一阶段的起点，这种关系描述了由K阶段到K+1阶段状态的演变规律，是关于两个相邻阶段状态的方程，称为状态转移方程，是动态规划的核心。

样例：【最长不下降序列】（线性DP）

给定一个长度为n的序列a，求出这个序列中最长不下降子序列，所谓最长不下降子序列列，就是对于j<i，有a[j]<=a[i].

输入格式：

第1行，一个整数n（n<=5000）

下面n行，每行一个整数，其中第i+1行的数为序列a[i]的值。（a[i]<=10^9).

输出格式：

第1行，最长不下降子序列长度。

第2行，最长不下降子序列（若有多个，输出一个即可）。

样例：【最长不下降序列】

【分析】本题给出的是一个序列，若将该序列的每一个数看成一个点，则数据模型就是一个线性队列。

设f[i]表示到当前第i个元素为止最长不下降子序列长度，对于当前的第i个元素，考虑跟之前哪一个子序列能继续构成不下降序列，从而选取决策。状态转移方程：

$$f[i]=\max\{f[i],f[j]+1\}\text{满足}j<i\&a[j]<=a[i]$$

因为要输出方案，所以在状态转移的时候，需要记录下这个状态是由哪个状态转移过来的。即记录最优决策。pre[i]=j表示当前子序列i的前继为j。

i	1	2	3	4	5	6	7	8
a[i]	38	27	55	30	29	70	58	65
f[i]	1	1	2	2	2	3	3	4

样例：【最长不下降序列】

```cpp
#include<bits/stdc++.h>
using namespace std;
const int N=5005;
const int inf=0x3f3f3f3f;
int n,len,a[N],f[N],pre[N],ans[N]; //f为状态描述，pre记录决策树
int main(){
    scanf("%d",&n);
    for(int i=1;i<=n;i++) scanf("%d",&a[i]);
    a[0]=-inf;//尽管a[0]不存在，因为f[1]要与a[0]比较，因此设为最小
    for(int i=1;i<=n;i++)
    {
        pre[i]=0;//初始化决策
        for(int j=1;j<i;j++)
            if(a[j]<=a[i]&&f[j]>f[pre[i]])pre[i]=j;//记录决策
        f[i]=f[pre[i]]+1;//更新最优值
    }
    int tail=0;
    for(int i=1;i<=n;i++)//找到最长子序列的位置
        if(f[tail]<f[i])tail=i;
    while(tail){//将最优子序列逆序放入ans中
        ans[++len]=tail;
        tail=pre[tail];
    }
    printf("%d\n",len);//输出长度
    while(len)printf("%d ",a[ans[len--]]);//输出子序列
    printf("\n");
    return 0;
}
```

```
14
13 7 9 16 38 24 37 18 44 19 21 22 63 15
8
7 9 16 18 19 21 22 63
```

真题1

某国为了防御敌国的导弹袭击，发展出一种导弹拦截系统。但是这种导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都不能高于前一发的高度。某天，雷达捕捉到敌国的导弹来袭。由于该系统还在试用阶段，所以只有一套系统，因此有可能不能拦截所有的导弹。

输入导弹依次飞来的高度（雷达给出的高度数据是≤50000的正整数），计算这套系统最多能拦截多少导弹，如果要拦截所有导弹最少要配备多少套这种导弹拦截系统。

输入格式

1行，若干个整数（个数≤100000）

输出格式

2行，每行一个整数，第一个数字表示这套系统最多能拦截多少导弹，第二个数字表示如果要拦截所有导弹最少要配备多少套这种导弹拦截系统。

输入

389 207 155 300 299 170 158 65

输出

6

2

题解1（DP+贪心）

分析:

第一问：使用动态规划，求最长不降序列。

第二问：根据不同的拦截系统，每颗导弹选择最优情况（拦截高度最低）可以使用贪心算法实现。

```cpp
#include<iostream>
#include<cstring>
using namespace std;
int main(){
    int a[1001],b[1001],h[1001],maxx,x,k;
    //a记录导弹高度，b记录当前最长不下降序列，c记录后继节点
    memset(a,0,sizeof(a));
    memset(b,0,sizeof(b));
    memset(h,0,sizeof(h));
    int i=1,m=0,n=0;
    while(cin>>a[i]){
        maxx=0;//巧妙点在于用一个maxx变量来记录长度。
        for(int j=1;j<=i-1;j++){  //计算前i-1个导弹最佳拦截方案
            if(a[j]>a[i])
                if(b[j]>maxx) maxx=b[j];
        }
        b[i]=maxx+1;//在长度上延续
        if(b[i]>m)m=b[i];//m表示最长不降序
        x=0; //下面贪心求得需要多少拦截系统
        for(k=1;k<=n;k++)
            if(h[k]>=a[i])
                if(x==0)x=k;
                else if(h[k]<h[x])x=k;
                //如果有多套系统可以拦截，则选择上一次拦截高度最低的
        if(x==0){
            n++;x=n;//如果没有一套可以拦截，也就是说a[i]大于任意一套的最小值
        }           //就新增一套系统拦截导弹
        h[x]=a[i];
        i++;
    }
    cout<<m<<endl<<n<<endl;
    return 0;
}
```

题解2（DP）：

为什么最长上升子序列长度表示系统个数?

当我们发现后一发导弹的高度高于前一发时我们就需要多用一套拦截系统，所以，求需要多少套拦截系统，就是求最长上升子序列长度。

（证明略）

如果先求最长下降序列，再求最长上升，超时。

所以一起计算减少时间复杂度。

```cpp
#include<cstdio>
#include<iostream>
using namespace std;
int v[10005],f[10005],g[10005]={0,1},maxn,minn;
//maxn表示最长上升子序列长度，minn表示最长下降子序列的长度
int main()
{
    int n;
    scanf("%d",&n);
    for(int i=1;i<=n;i++)
    {
        scanf("%d",&v[i]); //输入每枚导弹的高度
        f[i]=g[i]=1;
        for(int j=1;j<i;j++)
        {
            if(v[i]<=v[j]) //求最长下降子序列
            {
                f[i]=max(f[i],f[j]+1);//两个状态转移方程
            }
            if(v[i]>v[j]) //求最长上升子序列
            {
                g[i]=max(g[i],g[j]+1);
            }
        }
        if(f[i]>maxn) //求出最长下降子序列的长度
        {
            maxn=f[i];
        }
        if(g[i]>minn) //求出最长上升子序列的长度
        {
            minn=g[i];
        }
    }
    printf("%d %d\n",maxn,minn);
    /*输出最长下降子序列长度与最长上升子序列的长度，也就是
    一套系统最多拦截多少枚导弹与需要多少套拦截系统*/
    return 0;
}
```

谢谢大家！