最少钞票问题

假设您是个土豪，身上带了足够的1、5、10、20、50、100元面值的钞票。现在您的目标是凑出某个金额w，需要用到尽量少的钞票，请在某个w下最少的钞票是多少？

依据生活经验，我们显然可以采取这样的策略：能用100的就尽量用100的，否则尽量用50的……
依次类推，某些数据情况下贪心策略是正确的。
但是，如果我们换一组钞票的面值，贪心策略就也许不成立了。
如果一个奇葩国家的钞票面额分别是1、5、11，那么我们在凑出15的时候，贪心策略会出错：
15=1×11+4×1
15=3×5

最少钞票问题

w=15时，我们如果取11，接下来就面对w=4的情况；
如果取5，则接下来面对w=10的情况。
我们发现这些问题都有相同的形式：“给定w，凑出w所用的最少钞票是多少张？”
我们用f(n)来表示“凑出n所需的最少钞票数量”。

取11　　cost=f(4)+1

取5　　cost=f(10)+1

取1　　cost=f(14)+1

取三种方案的最小值，**我们通过上面三个式子，做出了正确的决策**

$$f(n) = \min\{f(n-1), f(n-5), f(n-11)\} + 1$$

信息学暑期

动态规划第一讲

(概念与基本DP模型)

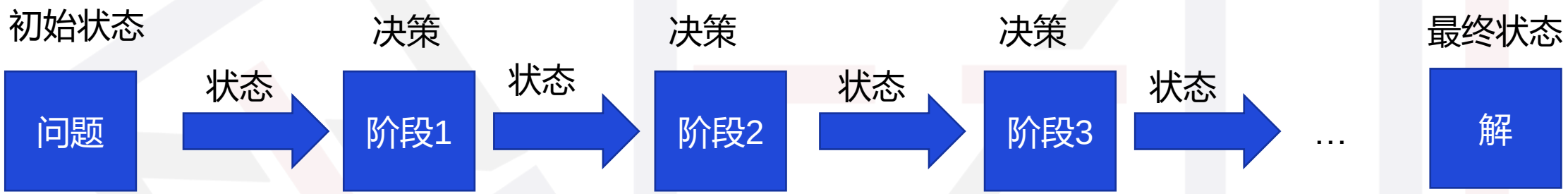
动态规划

dynamic programming

和贪心、分治一样，DP并不是一种特定算法，而是一种算法**思想**

问题解决的过程

经过前人的研究，一个问题的解决大致是这样的：



DP基于这样的一个问题解决模式，设计算法

动态规划主要分为4个核心部分:

- 确定「动态规划状态」
- 划分「阶段」
- 确定「动态规划转移方程(决策)」
- 确定「边界条件」

最少钞票问题

w=15时，我们如果取11，接下来就面对w=4的情况；
如果取5，则接下来面对w=10的情况。
我们发现这些问题都有相同的形式：“给定w，凑出w所用的最少钞票是多少张？”
我们用f(n)来表示“凑出n所需的最少钞票数量”。→ **定义的问题状态**

取11　　cost=f(11)+1

取5　　cost=f(10)+1 → **面值金额作为解决问题的阶段**

取1　　cost=f(14)+1

取三种方案的最小值，**我们通过上面三个式子，做出了正确的决策**

$$f(n) = \min\{f(n-1), f(n-5), f(n-11)\} + 1$$ → **转移方程**

边界条件　f(0)=0;

隐藏条件：f(i)=inf

确定DP状态的两个原则

最优子结构

无后效性

动态规划状态：最优子结构

前置芝士：子问题，比大问题规模更小，但问题描述、解法相似的问题

将原有问题化分为一个个子问题，即为子结构。
对于每一个子问题，其最优值均由「更小规模的子问题的最优值」推导而来，即为最优子结构。
因此「DP 状态」设置之前，需要将原有问题划分为一个个子问题，且需要确保子问题的最优值

例如在「钞票问题」中，

原有问题是「和为s的最少的钞票组合」

子问题是「和为i (1~s) 的最少的钞票组合」

子问题是由「$i-v_j$」推出

子问题由更小规模的子问题组成，因此满足「最优子结构」原则。

动态规划状态：无后效性

对于「无后效性」，某个状态一旦确定，此后的过程演变不再受此前各状态的影响，未来与过去无关

仍以「钞票问题」例题为例，我们令 DP 状态　　表示「和为s的最少的钞票组合」
我们只关心f(i)这个子问题的最优值，并不关心这个子问题的最优值是从哪个其它子问题转移而来。

为什么需要无后效性?

随着DP题目训练量增多，你会发现，DP其实利用的是前面子问题的信息来进行求解，如果后面的计算影响到了前面的结果，那么DP的正确性就无法保证了

动态规划高效率的根源

动态规划比递归搜索效率高的原因在于：问题需要具有重复子问题，这是DP时间复杂度低的前提

因为递归搜索并不会记录子问题的答案，所以会进行大量的重复计算　优化：记忆化搜索

如果拿DP去解决一个无重复子问题的大问题，时间复杂度会很高。

参考代码


```
for(int i=0;i<=s;i++)
    f[i]=inf;
for(int i=0;i<=s;i++)  //金额为阶段
    for(int j=1;j<=m;j++)  //枚举钞票种类
        if(i>v[j])
         f[i]=min(f[i],f[i-v[j]]+1)
```

Tips:外层循环一般是阶段

疑问：我怎么知道题目的状态和阶段怎么确定呢?

答：状态一般是求什么设什么，阶段以划分的子问题规模出发
但DP题目千变万化，但是常见的题目考查的无非就是几个DP模型的变形，多做题多积累多总结

小结：动态规划一般步骤

1、确定状态和状态变量

将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来。当然，状态的选择要满足**最优子结构、无后效性**。**重复子问题**是DP高效率的保证。

2、划分阶段

按照问题的时间或空间特征，把问题划分为若干个阶段。在划分阶段时，注意划分后的阶段一定是有序的或者是可排序的，否则问题就无法求解。

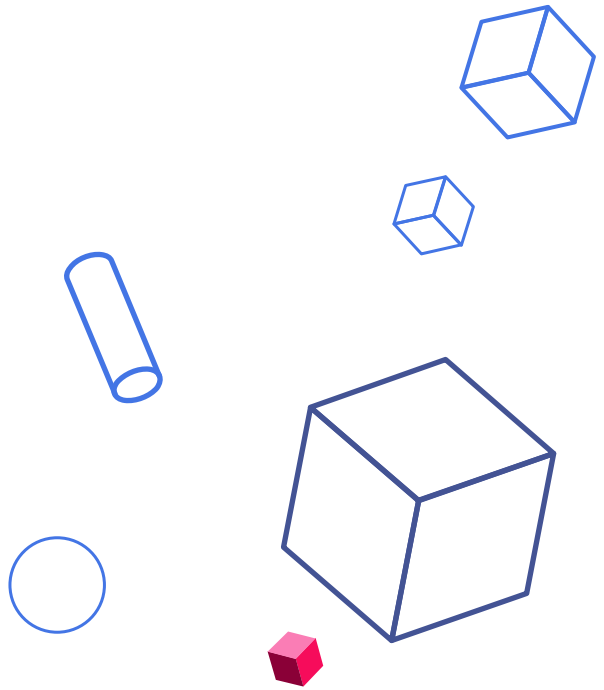
3、确定决策并写出状态转移方程

因为决策和状态转移有着天然的联系，状态转移就是根据上一阶段的状态和决策来导出本阶段的状态。所以如果确定了决策，状态转移方程也就可以写出。但事实上常常是反过来做，根据相邻两段的各个状态之间的关系来确定决策。

4、寻找边界条件

给出的状态转移方程是一个递推式，需要一个递推的终止条件或边界条件。

线性DP模型-LIS



LIS分为两类问题：最长不下降子序列、最长上升子序列

最长不下降子序列 LIS

求最长不下降子序列的长度(长度小于5000)

例如：1，3，3，2，7，6，8，5
最长不下降子序列是： 1 3 3 7 8

1、描述状态

以a[x] 存储原始数据

f[x] 表示以 a[x] 结尾的序列中，最长的序列。

a[x] 1，3，3，2，7，6，8

f[6]=4 1，3，3，6

最长不下降子序列 LIS

求最长不下降子序列的长度(长度小于5000)

例如：1，3，3，2，7，6，8，5
最长不下降子序列是：1 3 3 7 8

2、划分阶段

每一段数字
-每个a[i]的位置

1
1，3
1，3，3
1，3，3，2
1，3，3，2，7
1，3，3，2，7，6
1，3，3，2，7，6，8

最长不下降子序列 LIS

求最长不下降子序列的长度(长度小于5000)

例如：1，3，3，2，7，6，8，5
最长不下降子序列是： 1 3 3 7 8

3、确定决策并写出状态转移方程

f[x] 表示**以 a[x] 结尾**的序列中，最长的序列。

a[x] 1，3，3，2，7，6，8

f[6]=4 1，3，3，6

f[6]可以由哪些的状态转移过来

f[1]=1	1	f[6]=2
f[2]=2	1，3	f[6]=3
f[3]=3	1，3，3	f[6]=4
f[4]=2	1，2	f[6]=3
f[5]=2	1，3，3，7	**无法转移**

最长不下降子序列 LIS

求最长不下降子序列的长度(长度小于5000)

例如：1，3，3，2，7，6，8，5
最长不下降子序列是： 1 3 3 7 8

3、确定决策并写出状态转移方程

$$f[x] = max(f[i]) + 1, (i < x且a[i] <= a[x])$$

4、寻找边界条件

```
f[i] = 1
```

最长不下降子序列 LIS 代码


```cpp
#include <bits/stdc++.h>
using namespace std;
int a[5010], n;
int dp[5010]; //设计状态,以i为结束点的子序列的最长不下降子序列的长度
int main()
{
    cin >> n;
    for (int i = 1; i <= n; i++)
        cin >> a[i];
    for (int i = 1; i <= n; i++) { //以每个a[i]作为结束点,计算dp
        dp[i] = 1; //开始它自己为一个子序列时长度为1
        for (int j = 1; j < i; j++) { //遍历它前面的序列子序列,不一定要相邻
            if (a[j] <= a[i]) { //如果满足“ 不下降”
                dp[i] = max(dp[i], dp[j] + 1); //就用dp[j]+1来更新答案
            }
        }
    }
    int ans = -1;
    for (int i = 1; i <= n; i++)
        ans = max(ans, dp[i]);
    cout << ans;
    return 0;
}
```

【应用1】1570 导弹拦截

某国为了预防敌国的导弹袭击，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达到任意的高度，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不大于30000的正整数，导弹数不超过1000），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】
389 207 155 300 299 170 158 65
【输出】
6
2

第一问：LIS
第二问：贪心

【应用2】 1082 合唱队形

N位同学排成一排，要求同学们从左到右先递增再递减（不可以相等）问至少需要几位同学出列，剩下的同学才能满足要求?

N <= 100

核心:

从左往右求一个最长上升dp1[],
从右往左求一个最长上升dp2[],
然后枚举每个点i，求n-(max(dp1[i]+dp2[i])-1)

小结： DP中常见模型与应用

有些时候感到智商突然离线，想不出来状态如何定义？

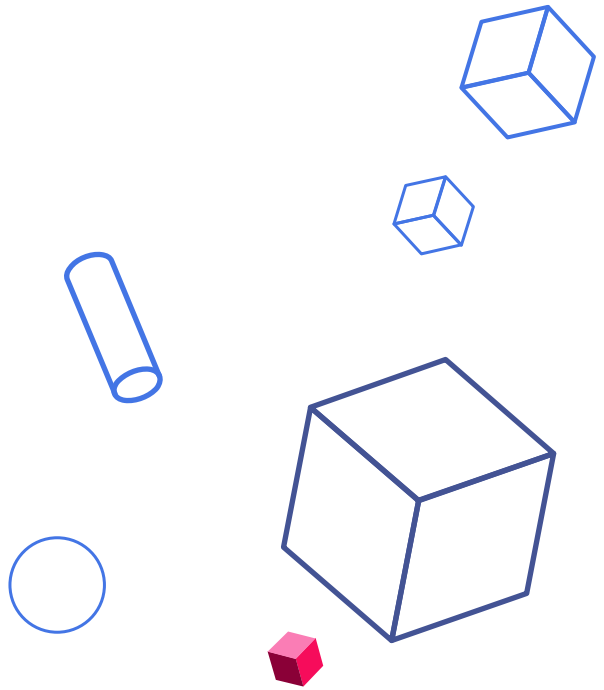
其实常见的状态定义**通常是一些模型的演化**。。。。

你只需要熟悉常见的模型，然后就能有机会做出某些题。

当然我命由我不由天，多做DP题，量变引起质变

例如 应用1 和 应用2 均为**LIS问题**的应用题

线性DP模型-LCS



最长公共子序列 LCS

给定 S T 两字母串，现分别从两串中选取部分字符组成新字母串。
要求两字母串相同，问新字母串最长多长。

【输入】abccd aecd
【输出】3
【范围】len(A),len(B) <2000

和最长不下降LIS类似，可以用两个**子序列**的结尾做为状态

1.定义状态： C(i,j)是Si和Tj的LCS长度:
字符串S取从1到i的子串，字符串T取1到j的子串的最长公共子序列长度;

最长公共子序列 LCS

给定 S T 两字母串，现分别从两串中选取部分字符组成新字母串。
要求两字母串相同，问新字母串最长多长。

【输入】abccd aecd
【输出】3
【范围】len(A),len(B) <2000

设：**序列S=<s1,s2,…，sm>和T=<t1,t2,…，tn>的一个最长公共子序列Z=<z1,z2,…,zk>。**

最后一个元素相同时 $t_n==s_m$ 说明一定在两个字母串的LCS中，
所以我们只需要找到C(Sm-1,Tn-1)的最优值+1即可

$t_n! =s_m$ 说明不在两个字母串的LCS中 就分解出两种情况： MAX{ 1. C(Sm-1,Tn) ; 2. C(Sm,Tn-1) }

最长公共子序列 LCS

给定 S T 两字母串，现分别从两串中选取部分字符组成新字母串。
要求两字母串相同，问新字母串最长多长。

【输入】abccd aecd
【输出】3
【范围】len(A),len(B) <2000

2.确定决策并写出状态转移方程

$$C[i,j]=\begin{cases} C[i-1,j-1]+1 & \text{若 } i,j>0, x_i=y_j \\ \max\{C[i,j-1],C[i-1,j]\} & \text{若 } i,j>0, x_i\neq y_j \end{cases}$$

最长公共子序列 LCS

给定 S T 两字母串，现分别从两串中选取部分字符组成新字母串。
要求两字母串相同，问新字母串最长多长。

【输入】abccd aecd
【输出】3
【范围】len(A),len(B) <2000

2.确定决策并写出状态转移方程

$$C[i,j]=\begin{cases}0 & \text{若 } i=0 \text{或} j=0\\ C[i-1,j-1]+1 & \text{若 } i,j>0, x_i=y_j\\ \max\{C[i,j-1],C[i-1,j]\} & \text{若 } i,j>0, x_i\neq y_j\end{cases}$$

最长公共子序列 LCS

给定 S T 两字母串，现分别从两串中选取部分字符组成新字母串。
要求两字母串相同，问新字母串最长多长。

【输入】abccd aecd
【输出】3
【范围】len(A),len(B) <2000

3.边界条件

f[0][]=0;
f[][0]=0;

核心代码

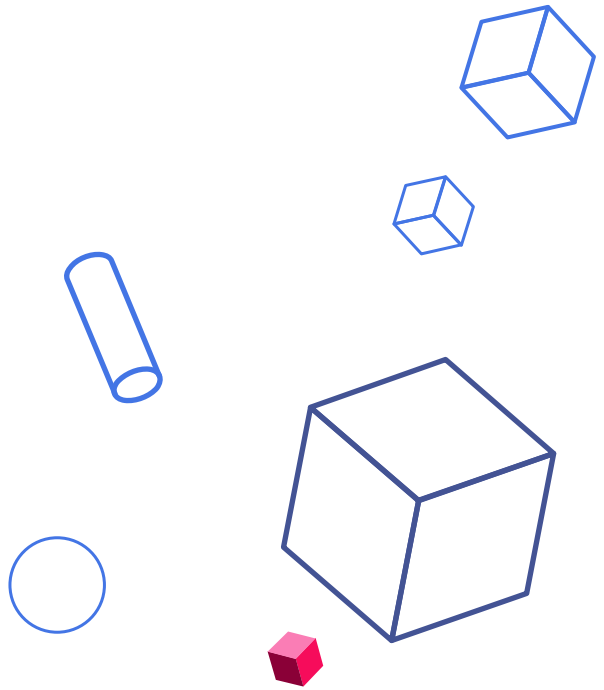
```
for(int i=1;i<=ls;i++){
    for(int j=1;j<=lt;j++){
        f[i][j] = max(f[i-1][j],f[i][j-1]);
        if(S[i-1] == T[j-1]){ //从0开始存的
            f[i][j] = max(f[i][j],f[i-1][j-1]+1);
        }
    }
```

最长不下降子序列 （LIS）
最长公共子序列（LCS）
都是动态规划中的常用模型

很多题目都是通过这些模型演变而来，多加练题，这类题就容易解决

FZOI
西大附中信息学竞赛
High School Affiliated to Southwest University

西南大学附属中学
High School Affiliated to Southwest University



区间DP

合并石子

在一个操场上一排地摆放着N堆石子。现要将石子有次序地合并成一堆。规定每次只能选相邻的2堆石子合并成新的一堆，并将新的一堆石子数记为该次合并的得分。
试设计一个程序，计算出将N堆石子合并成一堆的最小得分。

输入一个n，接下来n行石头的的得分

【输入样例】

7
13
7
8
16
21
4
18

【输出样例】

239

合并石子

西南大学附属中学 High School Affiliated to Southwest University

【输入样例】

7
13
7
8
16
21
4
18

【输出样例】

239

题意：每次选择相邻两堆合并，代价为两堆之和
经过n-1次合并后变成一堆，求总代价最小

贪心是否可行？ 每次选择代价最小的相邻两堆合并

28	16	21	22	
28	37	22	37	
28	59		37+59	
87			37+59+87	

目光短浅

28	16	21	22	
44	21	22	44	
44	43		44+43	
87			44+43+87	

更好的合并方案

合并石子

假设只有2堆石子

显然只有一种合并方案

如果有3堆

则有两种:　（1（2 3））　　（（1 2）3）

如果有n堆呢?

不管怎么合并，最终都将成为2堆，如果我们将最后两堆分开，左边和右边无论怎么合并，都必须满足最优合并方案，整个问题才是最优解

合并石子

1、划分阶段

区间长度;

2、确定状态和状态变量

用t[i,j]表示第i堆到第j堆石头的总数
f[i,j]表示第i堆到第j堆合并的最小得分

3、确定决策并写出状态转移方程

因为[i, j] 一定是由[i, k] 和[k, j]合并而来。所以我们只需要遍历k，求min即可。
f[i][j]=min{f[i][k]+f[k+1][j]+t[i,j]} i<=k<=j-1

4、寻找边界条件

f[i][i]=0;

(前缀和优化)
s[i]表示前i堆石头的价值总和;
t[i,j]=s[j]-s[i-1]

合并石子


```
for(        阶段        )
   for (    每个阶段中的状态    )
   {
        计算每个状态

   }
```

```
for(i=1;i<=n;i++) f[i][i]=0;
for(len=2;len<=n;len++) //阶段为区间长度
     for(i=1;i<=n-len+1;i++){ //枚举区间起点
       j=i+len-1; //区间终点
       //计算f[i][j]
       for(k=i;k<=j-1;k++)
            f[i][j]=min(f[i][j],f[i][k]+f[k+1][j]+s[j]-s[i-1]);
      }
```

区间DP特点：

此类题有个共同之处：

有一个k，称为断点，将区间分为两段，决策则是找到哪一个k能分出最优的分段方案

通过断点，可以将问题不断的缩小到更小的子问题

石子合并

将（1<=n<=200）堆石子**绕圆形**操场摆放，现在要将石子有次序的合并成一堆。规定每次只能选相邻的两堆合并成新的一堆，并将新的一堆石子数，记作该次合并的得分。

【变化】数据变为了环

解法1：破环为链，枚举破坏点。展开为链后再做dp
只需要做n次dp，然后打擂台即可。

解法2： 2*n数组再多存一遍开始 DP

假设输入的数是a[1],a[2],.....,a[n] 那么使a[n+1]=a[1],a[n+2]=a[2],.....,a[2*n]=a[n]
发现n个数的环上的每一段都在2*n个数的链上了 于是就在链上dp等价于在环上dp

剩下的就和合并石子解法一样

环上dp通常都是先变为链再操作

例：能量项链

括号匹配

给你一个字符串，里面只包含"(",")","[","]"四种符号，请问你需要至少添加多少个括号才能使这些括号匹配起来。

如：

[]是匹配的

([])[]是匹配的

((]是不匹配的

([)]是不匹配的

1积分：确定这道题的状态和转移方程

输入

第一行输入一个正整数N，表示测试数据组数(N<=10)

每组测试数据都只有一行，是一个字符串S，S中只包含以上所说的四种字符，S的长度不超过100

输出

对于每组测试数据都输出一个正整数，表示最少需要添加的括号的数量。每组测试输出占一行

样例输入

```
4
[]
([])[]
((]
([)]
```

样例输出

```
0
0
3
2
```

状态

DP[i][j]表示字符串s的第i个字符到第j个字符需要添加的最少括号数

①、当i==j时，只有一个符号，需再添加一个字符，所以DP[i][i]=1;
②、当i<j时，若是s[i]与S[j]配对，那么DP[i][j]=DP[i+1][j-1];
若是不配对，那么从中任选一个字符作为分界点，i<=k<j

$$DP[i][j]=\min\{DP[i][k]+DP[k+1][j]\},\ i<=k<j$$

区间DP模板

1、至少有两维，表示左右端点

2、通常的代码格式

```
for(          区间长度          )
    for（    左端点   ）{
        计算右端点
        for（    枚举断点k，求最优决策）{

        }
    }
```

3、遇到环形先转化成链式，再用区间DP

断环为链的技巧：2倍存环

拓展学习：O(nlogn)的LIS求法

这里介绍LIS-O（nlogn）算法。

设dp[i]表示以i为结尾的最长递增子序列的长度

则状态转移方程为：dp[i] = max{dp[j]+1}，1≤j＜i,a[j]＜a[i].

考虑两个数a[x]和a[y]，x＜y且a[x]＜a[y],且dp[x]=dp[y]，当a[t]要选择时，到底取哪一个构成最优的呢?

显然选取a[x]更有潜力，因为可能存在a[x]＜a[z]＜a[y]，这样a[t]可以获得更优的值。

在这里给我们一个启示，当dp[t]一样时，尽量选择更小的a[x].按dp[t]=k来分类，只需保留dp[t]=k的所有a[t]中的最小值，设g[k]记录这个值，g[k]=min{a[t],dp[t]=k}。

这时注意到g的两个性质（重点）：

1．g[k]在计算过程中单调不升;

2．g数组是有序的，g[1]＜g[2]＜..g[n]。

利用这两个性质，可以很方便的求解：(1).设当前已求出的最长上升子序列的长度为len（初始时为1），每次读入一个新元素x：(2).若x＞g[len]，则直接加入到d的末尾，且len++; （利用性质2）

否则，在g中二分查找，找到第一个比x小的数g[k]，并g[k+1]=x，在这里x≤g[k+1]一定成立（性质1,2）。

Thanks

For Your Watching