码上回顾

动态规划

动态规划概念

通过把原问题分解为简单的子问题来解决复杂问题，把每个子问题的最优解保存，用于求解最终最优解，已经算过的不会重复计算，从而提高解决问题的效率。

动态规划基本要素

1.最优子结构
2.无后效性
3.重叠子问题

动态规划求解步骤

1. 确定状态转移方程
2. 初始化状态
3. 状态转移

码上得知

最大子段和概念

码上得知

最大子段和概念

如下序列：

1 -2 5 3 -4 5

子段是从序列中截取某段连续的序列

这些都属于原序列的子段

码上得知

最大子段和概念

如下序列：

1 -2 5 3 -4 5

子段是从序列中截取某段连续的序列

1 -2 5　　-2 5 3 -4　　1 -2 5 3 -4 5

这些都属于原序列的子段

而这些：

1 -2 5 -4　　5 3 5　　-2 5 -4 5

非连续序列，都不算子段

码上得知

最大子段和概念

如下序列：

1 -2 5 3 -4 5

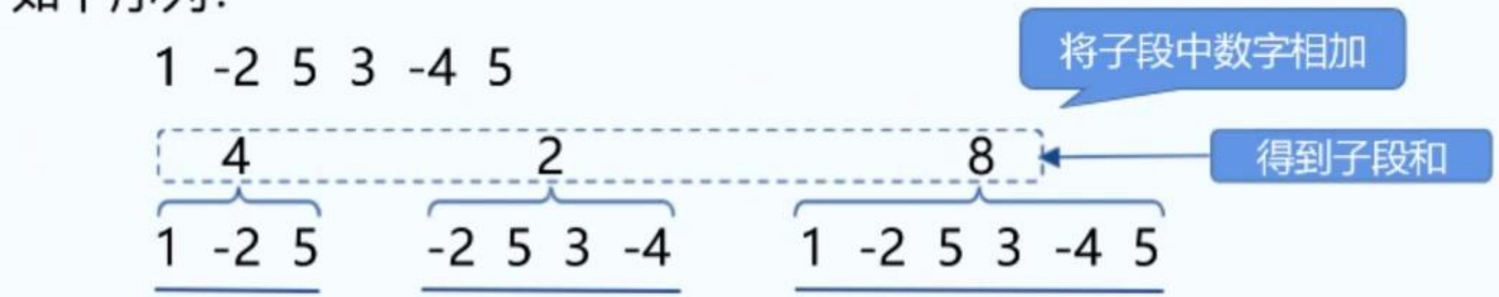
将子段中数字相加

1 -2 5　　-2 5 3 -4　　1 -2 5 3 -4 5

码上得知

最大子段和概念

如下序列：

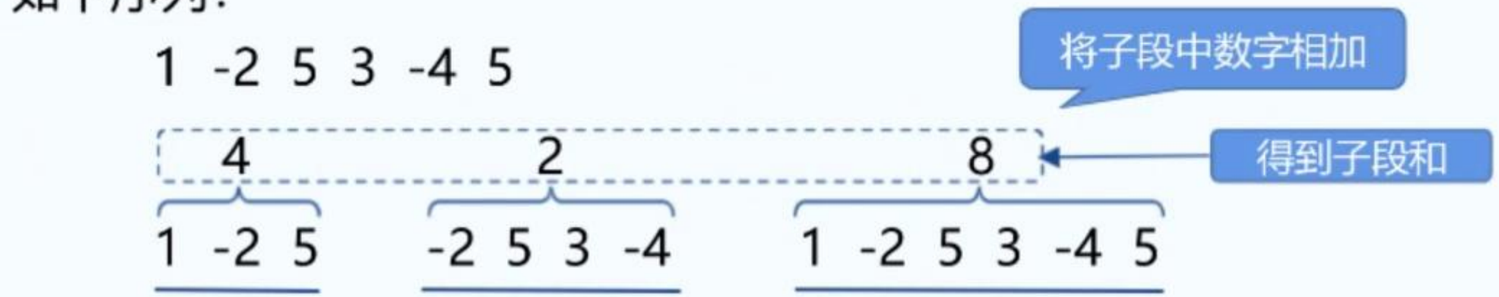


可以有多少个子段，就会有多少个子段和，所有的子段和中，最大的，称为最大子段和

码上得知

最大子段和概念

如下序列：

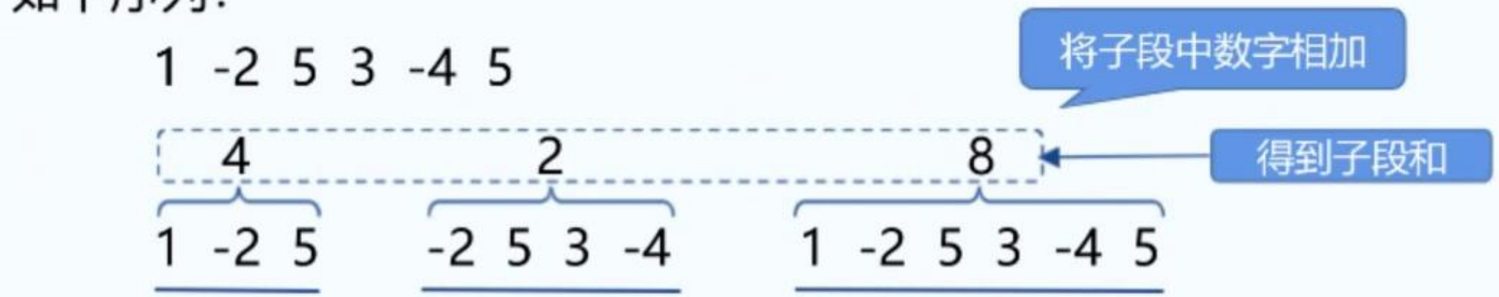


可以有多少个子段，就会有多少个子段和，所有的子段和中，最大的，称为最大子段和

码上得知

最大子段和概念

如下序列：



可以有多少个子段，就会有多少个子段和，所有的子段和中，最大的，称为最大子段和

解码时刻

样例分析

【输入样例】
6
1 -2 5 3 -4 5
【输出样例】
9

第一行数字6，表示序列中有6个数字

1 -2 5 3 -4 5

子段必须连续

5 + 3 + (-4) + 5 = 9

数字9为最大子段和，最终输出9

解码时刻

问题分析

最大子段和问题依然可以用动态规划处理。

数字序列：	1	-2	5	3	-4	5
最大子段和：						

某个数字为结尾时形成的最大子段和

从最小规模问题出发，一步步求解最大子段和。

问题分析

最大子段和问题依然可以用动态规划处理。

数字序列：	1	-2	5	3	-4	5
最大子段和：	1					

某个数字为结尾时形成的最大子段和

从最小规模问题出发，一步步求解最大子段和。

问题分析

最大子段和问题依然可以用动态规划处理。

数字序列：	1	-2	5	3	-4	5
最大子段和：	1					

某个数字为结尾时形成的最大子段和

从最小规模问题出发，一步步求解最大子段和。

从第2个位置开始每个位置的最优解面临两种选择

和左侧相邻子段结合　　加左侧值

自己单独形成一个子段　　不加左侧值

子段必须连续，所以只需考虑是否加左侧挨着的值。

解码时刻

问题分析

最大子段和问题依然可以用动态规划处理。

数字序列：	1	-2	5	3	-4	5
最大子段和：	1	-1				

某个数字为结尾时形成的最大子段和

从最小规模问题出发，一步步求解最大子段和。

从第2个位置开始每个位置的最优解面临两种选择

和左侧相邻子段结合　　加左侧值

自己单独形成一个子段　　不加左侧值

子段必须连续，所以只需考虑是否加左侧挨着的值。

问题分析

最大子段和问题依然可以用动态规划处理。

数字序列：	1	-2	5	3	-4	5
最大子段和：	1	-1	5	8		

某个数字为结尾时形成的最大子段和

从最小规模问题出发，一步步求解最大子段和。

从第2个位置开始每个位置的最优解面临两种选择

和左侧相邻子段结合　　加左侧值

自己单独形成一个子段　　不加左侧值

子段必须连续，所以只需考虑是否加左侧挨着的值。

解码时刻

问题分析

最大子段和问题依然可以用动态规划处理。

数字序列：	1	-2	5	3	-4	5
最大子段和：	1	-1	5	8	4	

某个数字为结尾时形成的最大子段和

从最小规模问题出发，一步步求解最大子段和。

从第2个位置开始每个位置的最优解面临两种选择

和左侧相邻子段结合　　加左侧值

自己单独形成一个子段　　不加左侧值

子段必须连续，所以只需考虑是否加左侧挨着的值。

解码时刻

问题分析

最大子段和问题依然可以用动态规划处理。

数字序列：	1	-2	5	3	-4	5
最大子段和：	1	-1	5	8	4	9

某个数字为结尾时形成的最大子段和

状态转移方程

当前子段和 = max(当前数字， 左侧子段和 + 当前数字)

数组a存储数字序列，dp存储以不同数字为结尾的最大子段和。i位置数字结尾的最大子段和为：

dp[i] = max(a[i] , dp[i-1] + a[i])

解码时刻

问题分析

最大子段和问题依然可以用动态规划处理。

数字序列：	1	-2	5	3	-4	5
最大子段和：	1	-1	5	8	4	9

某个数字为结尾时形成的最大子段和

状态转移方程

当前子段和 = max(当前数字， 左侧子段和 + 当前数字)

数组a存储数字序列，dp存储以不同数字为结尾的最大子段和。i位置数字结尾的最大子段和为：

dp[i] = max(a[i] , dp[i-1] + a[i])

解码时刻

问题分析

最大子段和问题依然可以用动态规划处理。

数字序列：	1	-2	5	3	-4	5
最大子段和：	1	-1	5	8	4	9

某个数字为结尾时形成的最大子段和

状态转移方程

当前子段和 = max(当前数字， 左侧子段和 + 当前数字)

数组a存储数字序列，dp存储以不同数字为结尾的最大子段和。i位置数字结尾的最大子段和为：

dp[i] = max(a[i] , dp[i-1] + a[i])

解题步骤

1. 数据输入

```
//n表示数字序列长度，a存储数字序列
int n, a[101];
cin>>n;
for(int i=1;i<=n;i++){
    cin>>a[i];
    ... ...
}
```

解题步骤

1. 数据输入

```
//n表示数字序列长度，a存储数字序列
int n, a[101];
cin>>n;
for(int i=1;i<=n;i++){
    cin>>a[i];
    ... ...
}
```

解码时刻

解题步骤

1. 数据输入

```
//n表示数字序列长度，a存储数字序列，dp存子段和
int n, a[101], dp[101];
cin>>n;
for(int i=1;i<=n;i++){
    cin>>a[i];
```

2. 初始化dp数组

```
    if(i==1) dp[i]=a[i];

}
```

数字序列：	1	-2	5	3	-4	5
最大子段和：	1					

用序列中第1个数进行初始化

解码时刻

解题步骤

1. 数据输入

```
//n表示数字序列长度，a存储数字序列，dp存子段和
int n, a[101], dp[101];
cin>>n;
for(int i=1;i<=n;i++){
    cin>>a[i];
```

2. 初始化dp数组

```
    if(i==1) dp[i]=a[i];

}
```

数字序列：	1	-2	5	3	-4	5
最大子段和：	1					

用序列中第1个数进行初始化

解码时刻

解题步骤

1. 数据输入

```
//n表示数字序列长度，a存储数字序列，dp存子段和
int n, a[101], dp[101];
cin>>n;
for(int i=1;i<=n;i++){
    cin>>a[i];
```

2. 初始化dp数组

```
    if(i==1) dp[i]=a[i];
```

3. 状态转移

```
    else dp[i]=max(a[i],dp[i-1]+a[i]);
}
```

数字序列:	1	-2	5	3	-4	5
最大子段和:	1	-1				

max(-2 , 1 + (-2))

解码时刻

解题步骤

1. 数据输入

```
//n表示数字序列长度，a存储数字序列，dp存子段和
int n, a[101], dp[101];
cin>>n;
for(int i=1;i<=n;i++){
    cin>>a[i];
```

2. 初始化dp数组

```
    if(i==1) dp[i]=a[i];
```

3. 状态转移

```
    else dp[i]=max(a[i],dp[i-1]+a[i]);
}
```

数字序列：	1	-2	5	3	-4	5
最大子段和：	1	-1	5			

max(5 , -1 + 5)

解码时刻

解题步骤

1. 数据输入

```
//n表示数字序列长度，a存储数字序列，dp存子段和
int n, a[101], dp[101];
cin>>n;
for(int i=1;i<=n;i++){
    cin>>a[i];
```

2. 初始化dp数组

```
    if(i==1) dp[i]=a[i];
```

3. 状态转移

```
    else dp[i]=max(a[i],dp[i-1]+a[i]);
}
```

数字序列：	1	-2	5	3	-4	5
最大子段和：	1	-1	5	8		

max(3 , 5 + 3)

解码时刻

解题步骤

1. 数据输入

```
//n表示数字序列长度，a存储数字序列，dp存子段和
int n, a[101], dp[101];
cin>>n;
for(int i=1;i<=n;i++){
    cin>>a[i];
```

2. 初始化dp数组

```
    if(i==1) dp[i]=a[i];
```

3. 状态转移

```
    else dp[i]=max(a[i],dp[i-1]+a[i]);
}
```

数字序列：	1	-2	5	3	-4	5
最大子段和：	1	-1	5	8	4	9

max(5 , 4 + 5)

解码时刻

解题步骤

1. 数据输入

```
//n表示数字序列长度，a存储数字序列，dp存子段和，ans初始化成极小值
int n, a[101], dp[101] ,ans=-1000;
cin>>n;
for(int i=1;i<=n;i++){
    cin>>a[i];
```

2. 初始化dp数组

```
    if(i==1) dp[i]=a[i];
```

3. 状态转移

```
    else dp[i]=max(a[i],dp[i-1]+a[i]);
    //保存最大子段和
    ans=max(ans,dp[i]);
}
```

数字序列：	1	-2	5	3	-4	5
最大子段和：	1	-1	5	8	4	9

解码时刻

解题步骤

1. 数据输入

```
//n表示数字序列长度，a存储数字序列，dp存子段和，ans初始化成极小值
int n, a[101], dp[101] ,ans=-1000;
cin>>n;
for(int i=1;i<=n;i++){
    cin>>a[i];
```

2. 初始化dp数组

```
    if(i==1) dp[i]=a[i];
```

3. 状态转移

```
    else dp[i]=max(a[i],dp[i-1]+a[i]);
    //保存最大子段和
    ans=max(ans,dp[i]);
}
```

数字序列：	1	-2	5	3	-4	5
最大子段和：	1	-1	5	8	4	9

解码时刻

完整代码

```
#include<bits/stdc++.h>
using namespace std;
int n,a[101],dp[101],ans=-1000;
int main(){
    //输入数据
    cin>>n;
    for(int i=1;i<=n;i++){
        cin>>a[i];
        //初始化dp数组
        if(i==1) dp[i]=a[i];
        //状态转移
        else dp[i]=max(a[i],dp[i-1]+a[i]);
        //保存最大子段和
        ans=max(ans,dp[i]);
    }
    //输出结果
    cout<<ans;
    return 0;
}
```

解码时刻

完整代码

```
#include<bits/stdc++.h>
using namespace std;
int n,a[101],dp[101],ans=-1000;
int main(){
    //输入数据
    cin>>n;
    for(int i=1;i<=n;i++){
        cin>>a[i];
        //初始化dp数组
        if(i==1) dp[i]=a[i];
        //状态转移
        else dp[i]=max(a[i],dp[i-1]+a[i]);
        //保存最大子段和
        ans=max(ans,dp[i]);
    }
    //输出结果
    cout<<ans;
    return 0;
}
```

码上得知

最长上升子序列

有数字序列：

5 3 9 11 4 2 15

码上得知

最长上升子序列

有数字序列：

5 3 9 11 4 2 15

什么是子序列?

5 3 9 15

9 4

2

5 3 9 11 4 2 15

...

这些都属于原序列的子序列

原数字序列去掉一些数字形成的新序列
原数字序列本身也算子序列

码上得知

最长上升子序列

有数字序列：

5 3 9 11 4 2 15

码上得知

最长上升子序列

有数字序列：

5 3 9 11 4 2 15

什么是上升子序列？

码上得知

最长上升子序列

有数字序列：

5 3 9 11 4 2 15

什么是上升子序列？

5 11 15
3 9 11 15
4 15
5 9 11 15
...

码上得知

最长上升子序列

有数字序列：

5 3 9 11 4 2 15

什么是上升子序列？

5 11 15
3 9 11 15
4 15
5 9 11 15
...

码上得知

最长上升子序列

有数字序列：

5 3 9 11 4 2 15

什么是上升子序列？

5 11 15

3 9 11 15

4 15

5 9 11 15

...

在所有上升子序列中
最长的为最长上升子序列

码上得知

最长上升子序列

有数字序列：

5 3 9 11 4 2 15

什么是上升子序列？

5 11 15

3 9 11 15

4 15

5 9 11 15

...

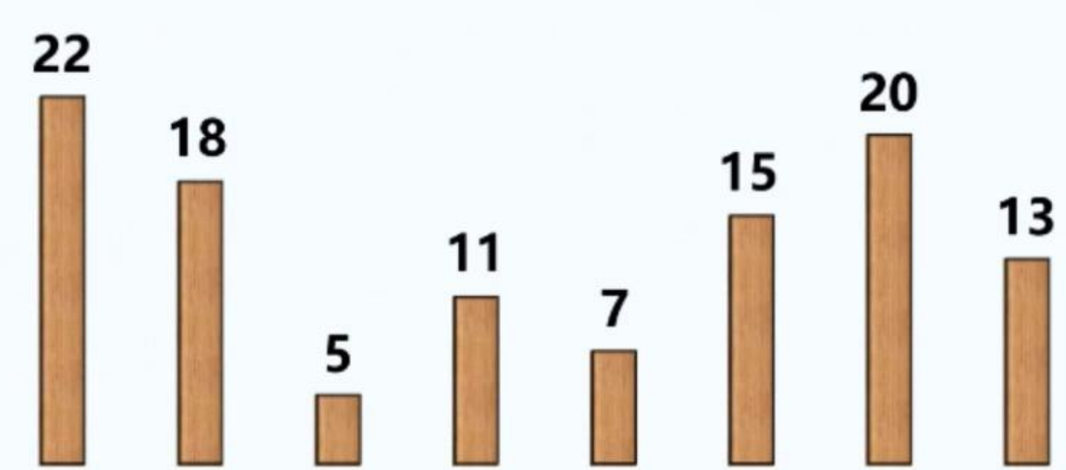
在所有上升子序列中
最长的为最长上升子序列

解码时刻

制作信箱

【问题描述】

小童想亲手制作一个木质的信箱，于是他到林中收集木棍，在收集木棍的过程中有一个原则，新收集的木棍要比上一个收集的木棍要长，现有如下木棍，从左到右顺序收集的话，小童最多可以收集多少根木棍。

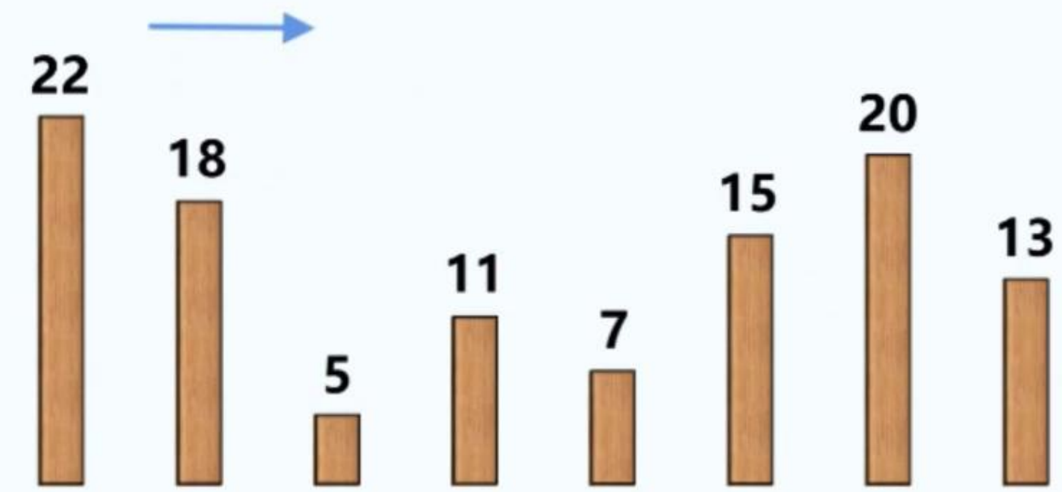


样例分析

【输入样例】
8
22 18 5 11 7 15 20 13
【输出样例】
4

从左到右顺序选取，每次选取的木棍要比前一次选取的木棍要长。

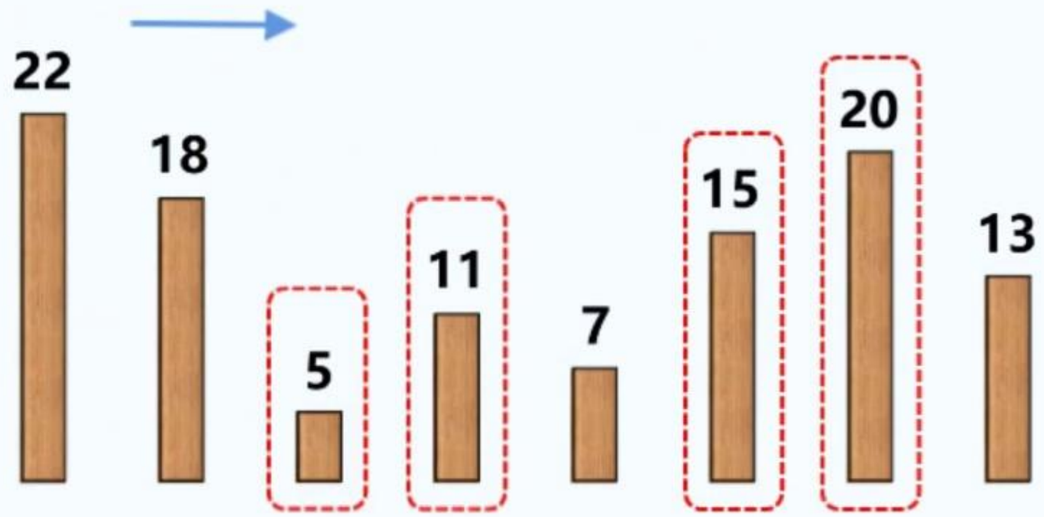
第1次拿22，最多拿1根。
第1次拿18，最多拿2根。
第1次拿5，最多拿4根。
5、11、15、20
或
5、7、15、20
这是最多的拿法，结果为4。



解码时刻

问题分析

此题的本质是在序列中找到最长的上升序列。



最长上升序列长度为4

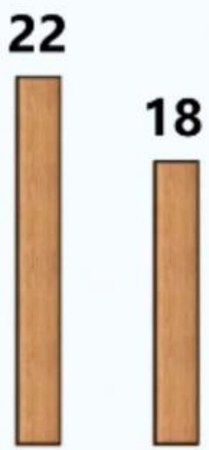
最长上升序列问题属于经典的动态规划问题。

解码时刻

问题分析

先把问题简化，从1根木头开始，把不同木头数量的最大上升序列长度都求出来：

2根木头



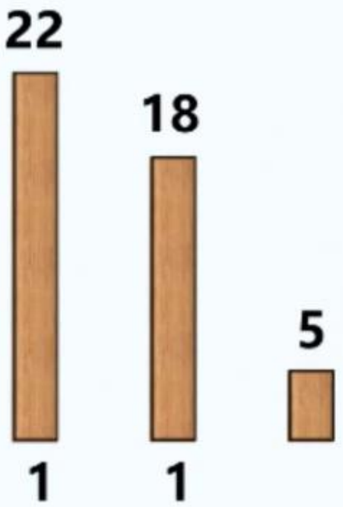
最大上升序列长度：1

解码时刻

问题分析

先把问题简化，从1根木头开始，把不同木头数量的最大上升序列长度都求出来：

3根木头

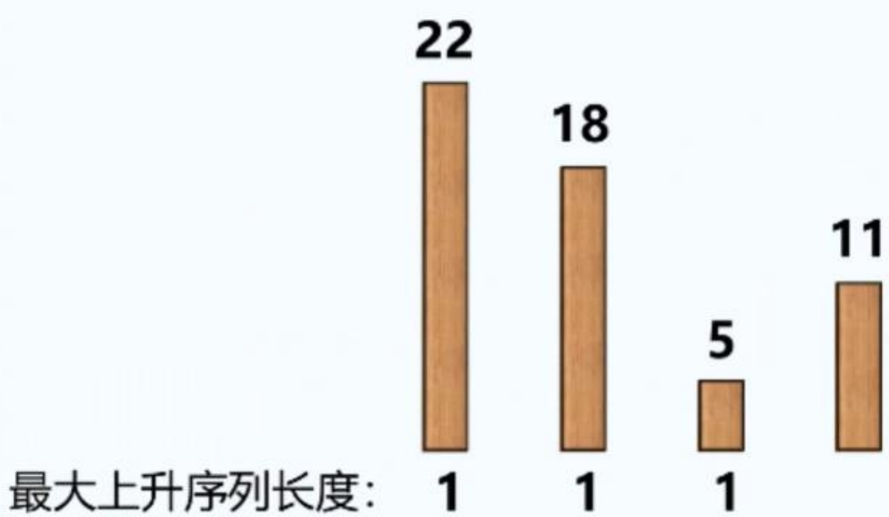


解码时刻

问题分析

先把问题简化，从1根木头开始，把不同木头数量的最大上升序列长度都求出来：

4根木头

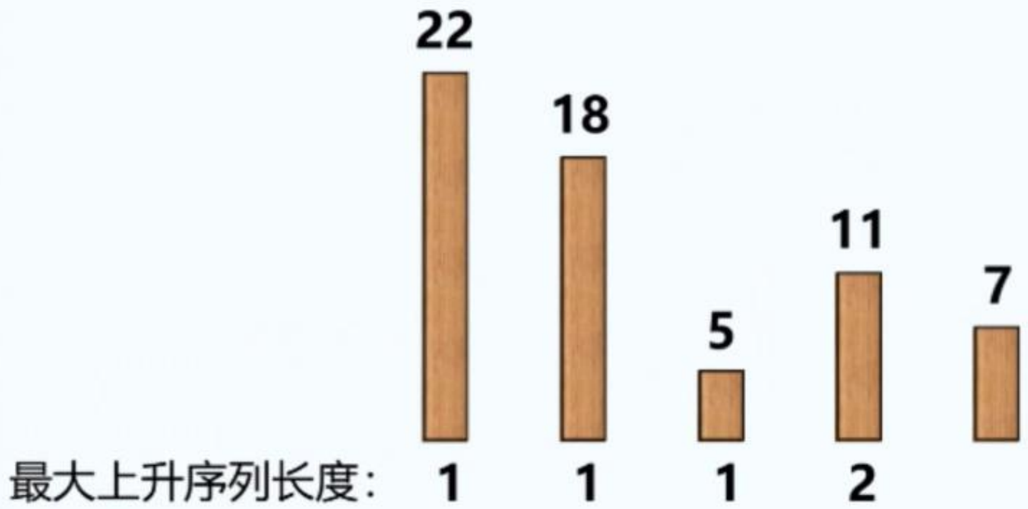


解码时刻

问题分析

先把问题简化，从1根木头开始，把不同木头数量的最大上升序列长度都求出来：

5根木头

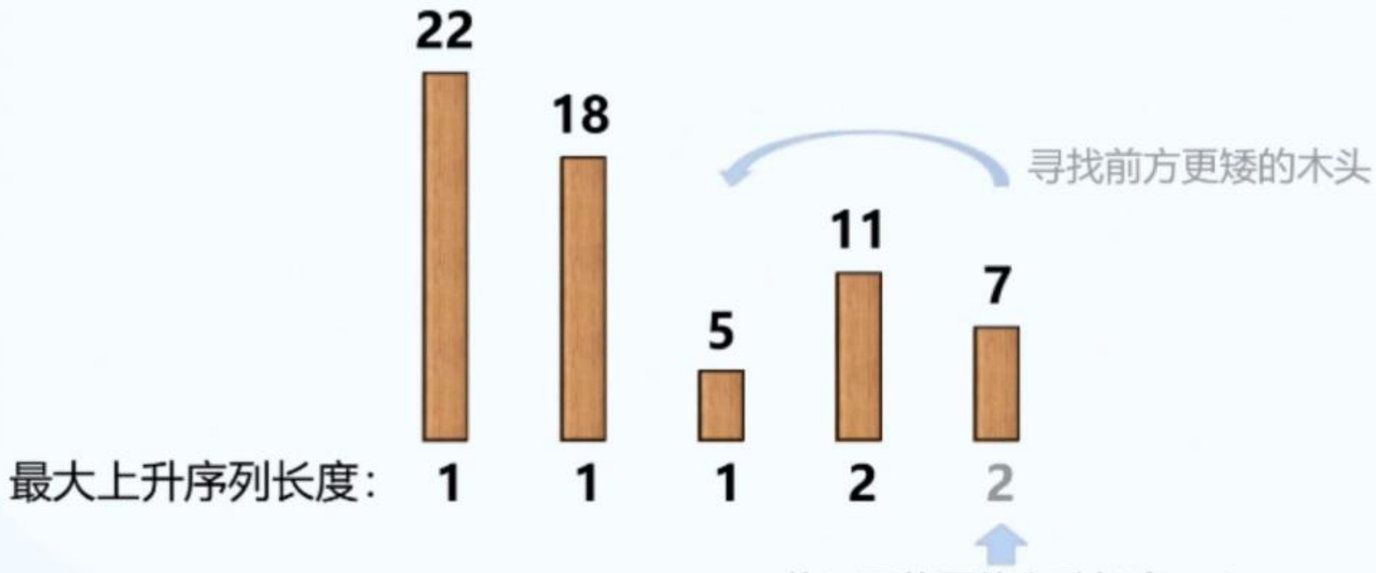


解码时刻

问题分析

先把问题简化，从1根木头开始，把不同木头数量的最大上升序列长度都求出来：

5根木头

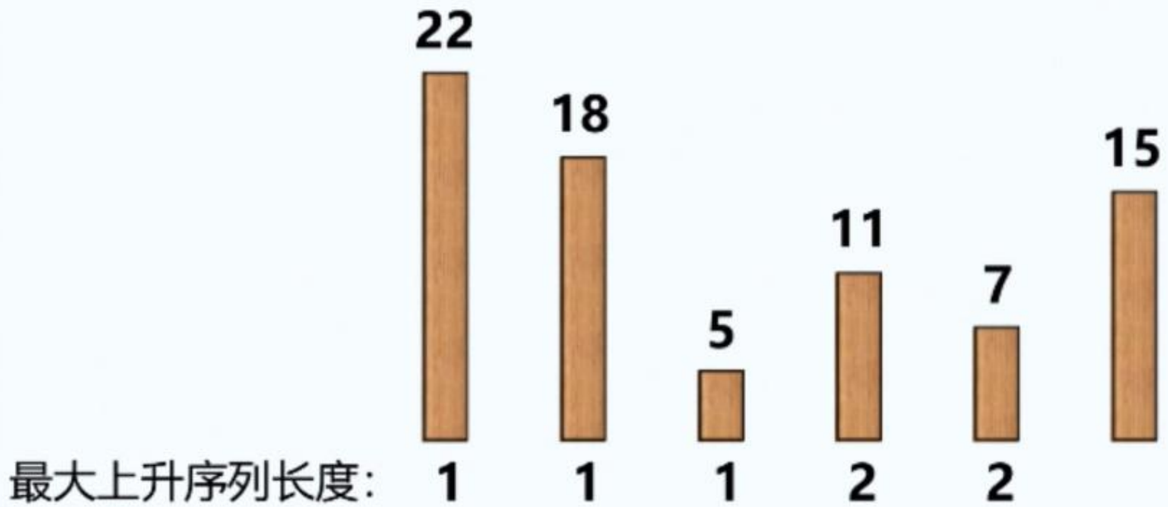


解码时刻

问题分析

先把问题简化，从1根木头开始，把不同木头数量的最大上升序列长度都求出来：

6根木头

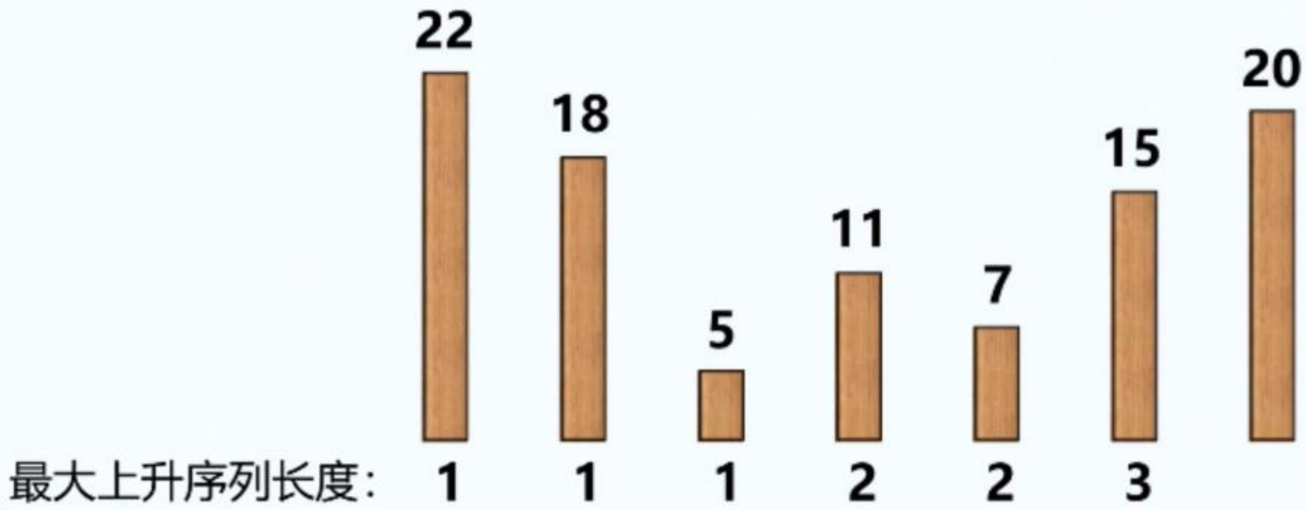


解码时刻

问题分析

先把问题简化，从1根木头开始，把不同木头数量的最大上升序列长度都求出来：

7根木头

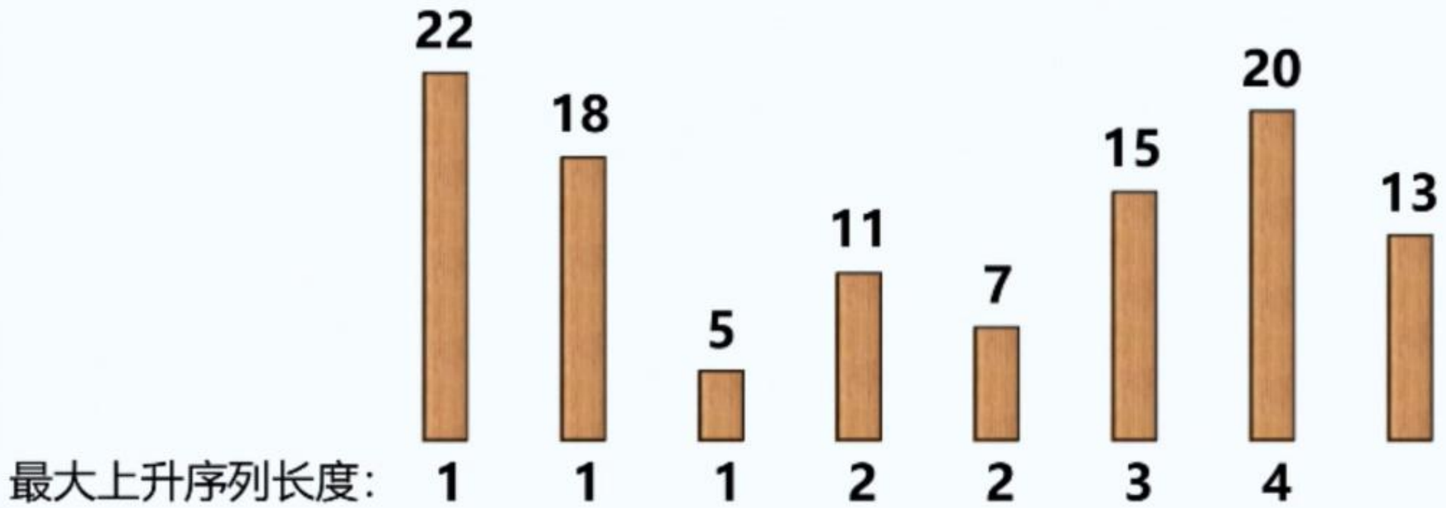


解码时刻

问题分析

先把问题简化，从1根木头开始，把不同木头数量的最大上升序列长度都求出来：

8根木头

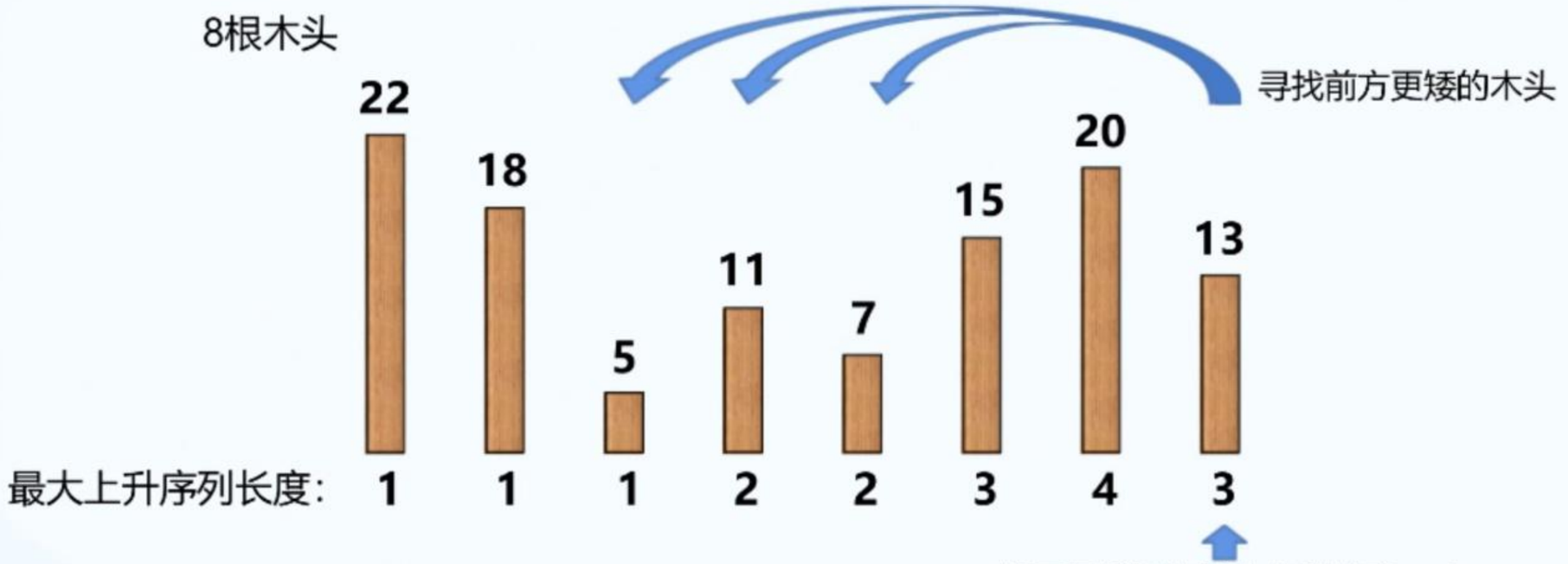


解码时刻

问题分析

先把问题简化，从1根木头开始，把不同木头数量的最大上升序列长度都求出来：

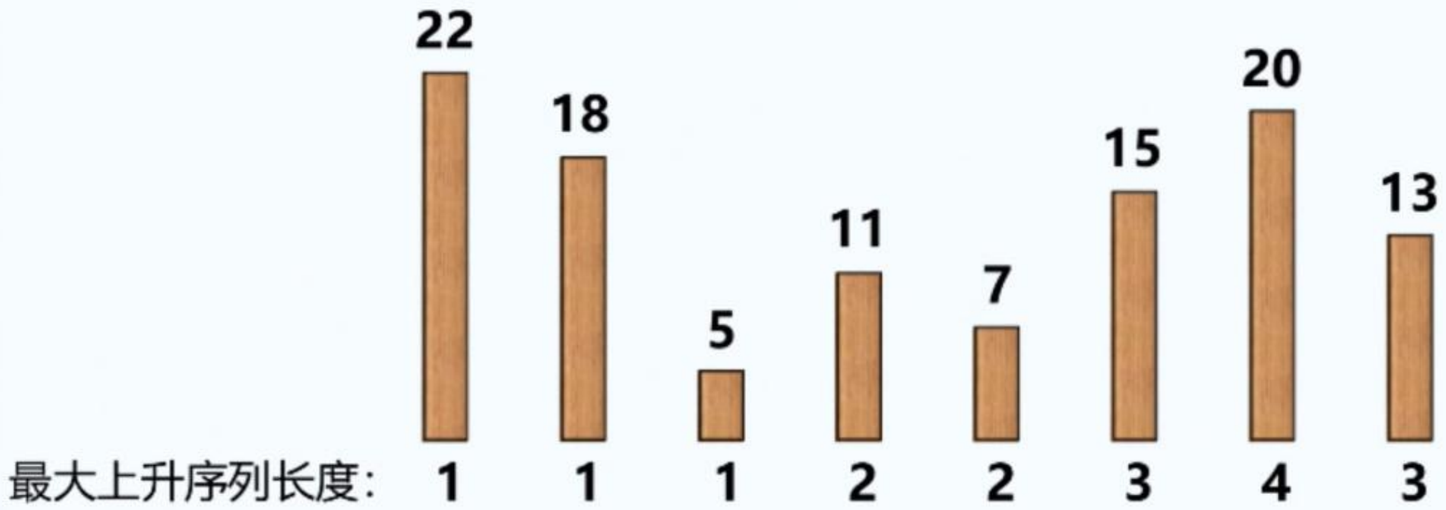
8根木头



解码时刻

问题分析

当前问题具备哪些动规要素？

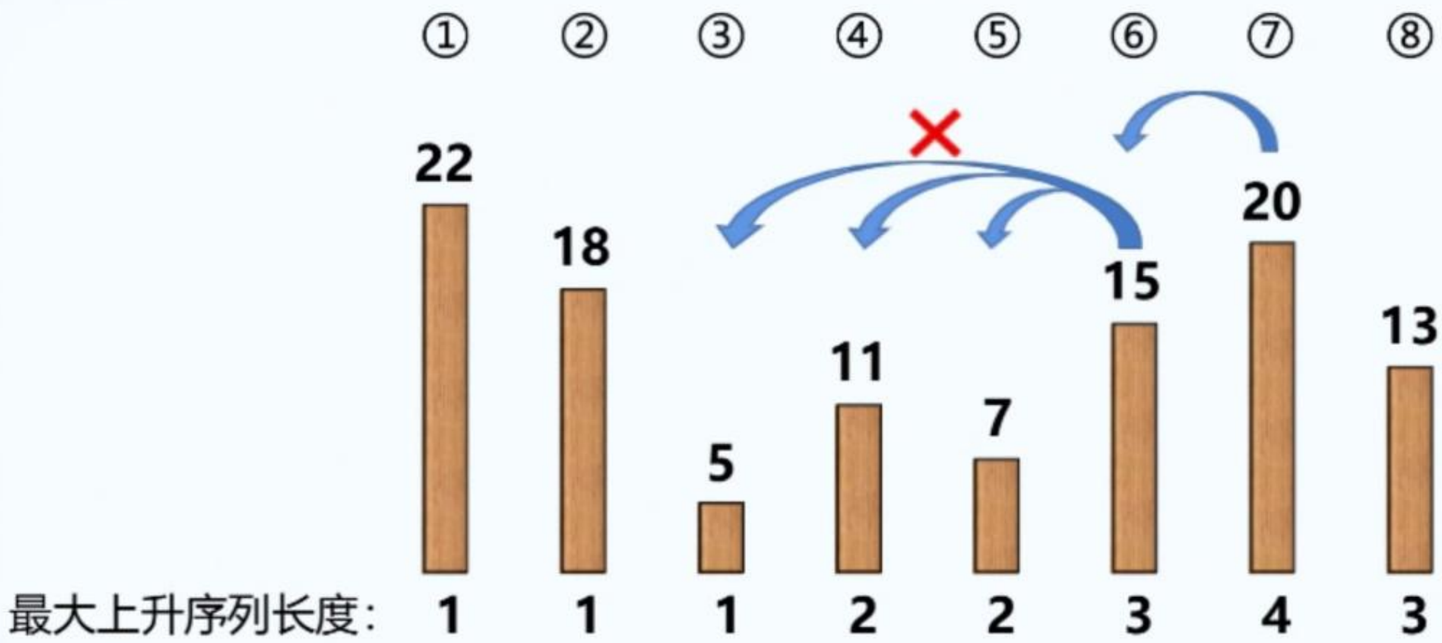


1. 每一步的解都是当前木头作为终点时的最优解，由小问题最优解向更大问题最优解转移，最优子结构。

解码时刻

问题分析

当前问题具备哪些动规要素？

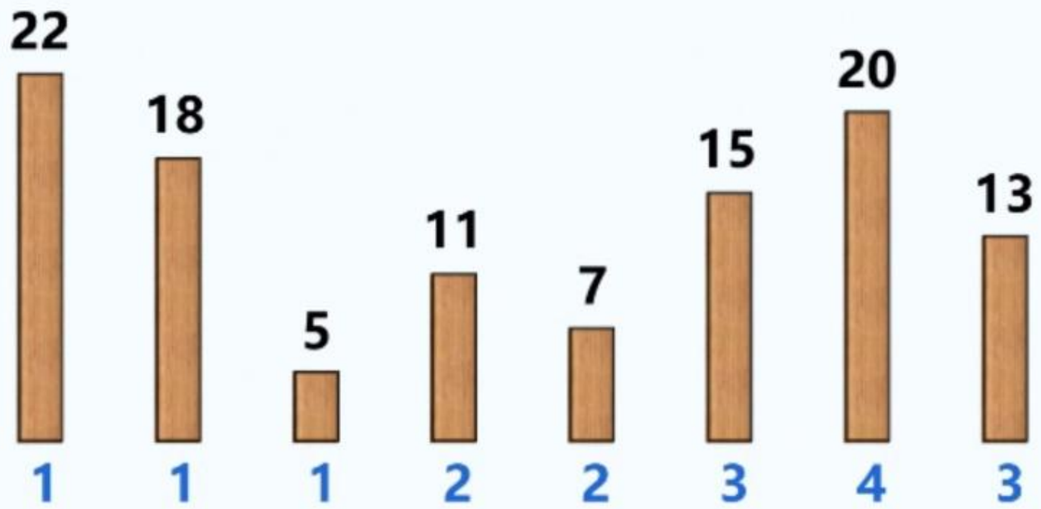


2. 例如求第7根木头的解，是用第6根木头的解计算而来，用不到第6根木头前一步的解，无后效性。

解码时刻

问题分析

状态转移方程



状态转移方程：左侧可用的最大值 + 1 = 当前位置可拿数量

解题步骤

1. 输入数据

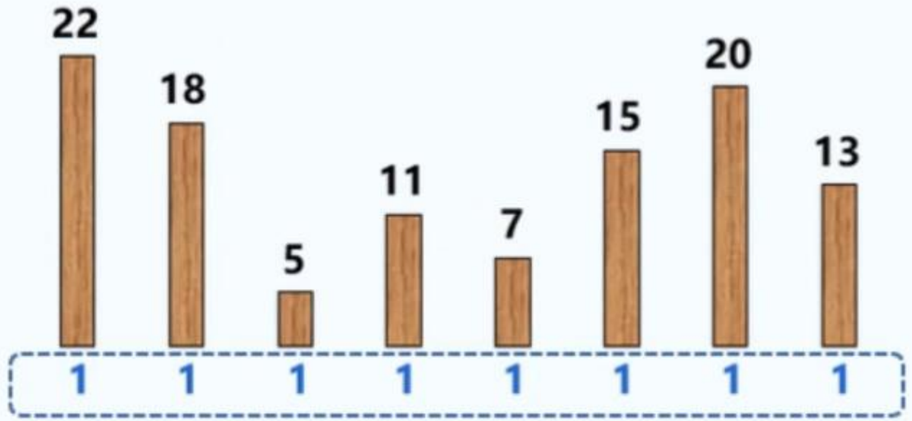
```
int a[1001], n; //定义在全局

cin>>n; //木棍数量
for (int i=1; i<=n; i++) {
    cin>>a[i]; //木棍长度
}
```

2. 初始化状态

```
int dp[1001]; //定义在全局

for (int i=1; i<=n; i++) {
    dp[i]=1;
}
```



dp 数组存储不同位置最大上升子序列长度。
最小长度为1，所以全部初始化为1。

解题步骤

1. 输入数据

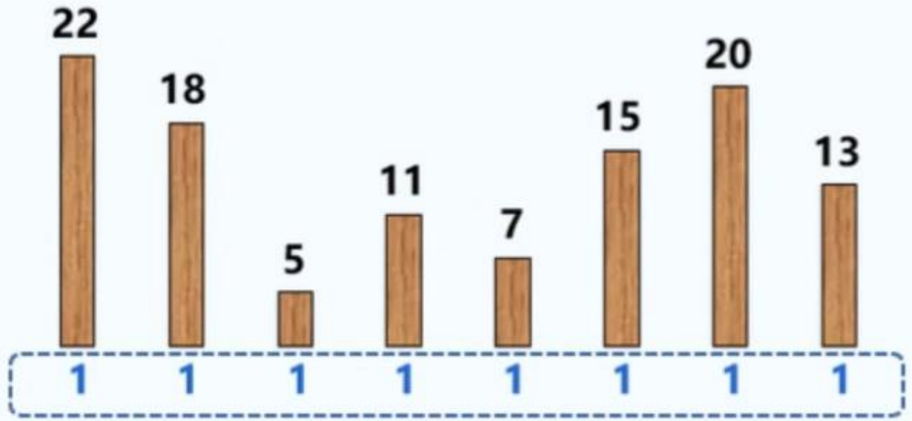
```
int a[1001], n; //定义在全局

cin>>n; //木棍数量
for (int i=1; i<=n; i++) {
    cin>>a[i]; //木棍长度
}
```

2. 初始化状态

```
int dp[1001]; //定义在全局

for (int i=1; i<=n; i++) {
    dp[i]=1;
}
```



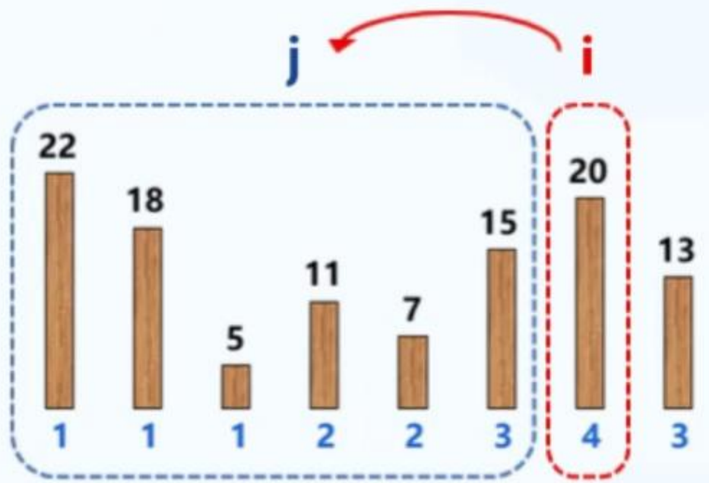
dp 数组存储不同位置最大上升子序列长度。
最小长度为1，所以全部初始化为1。

解码时刻

解题步骤

3. 状态转移

```
//遍历所有木棍
for(int i=1; i<=n; i++) {
    //遍历第i根木棍左侧的所有木棍
    for(int j=1; j<i; j++) {
        //判断木棍是否可用        状态转移方程
        if(a[j]<a[i]) dp[i]=max(dp[i],dp[j]+1);
    }
}
```



每次遍历1根木棍（i），检查左侧所有木棍（j）是否有可用的，进行状态转移，留存最大值。

4. 输出结果

```
int ans=0; //存储结果
for(int i=1;i<=n;i++) {
    ans=max(ans,dp[i]); //保存最大值
}
cout<<ans;
```

完整代码

```
#include<bits/stdc++.h>
using namespace std;
int a[1001], dp[1001], n;
int main() {
    //输入
    cin>>n;
    for (int i=1; i<=n; i++) {
        cin>>a[i];
    }
    //初始化
    for (int i=1; i<=n; i++) {
        dp[i]=1;
    }
    for(int i=1; i<=n; i++) {
        for(int j=1; j<i; j++) {
                                //状态转移
            if(a[j]<a[i]) dp[i]=max(dp[i],dp[j]+1);
        }
    }
    int ans=0;
    for(int i=1;i<=n;i++) {
        ans=max(ans,dp[i]);
    }
    //输出结果
    cout<<ans;
    return 0;
}
```

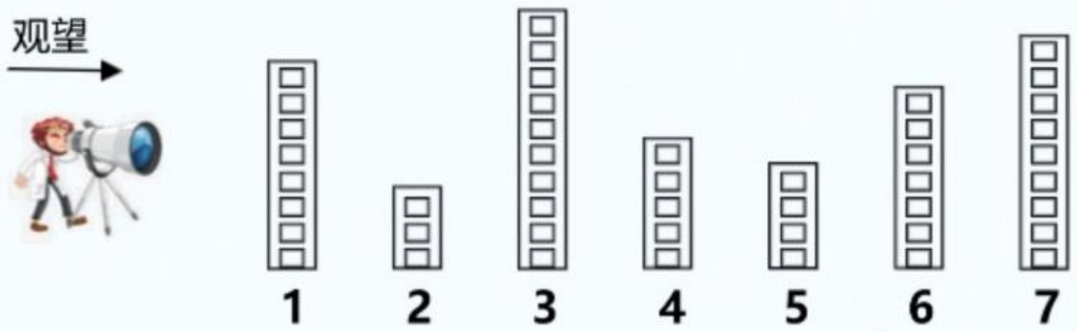
解码时刻

高低错落的大楼

【问题描述】

有n座大楼排在一条直线上，编号分别为1~n，每栋大楼的高度不同，你刚好站在这条直线上，从小编号的楼往大编号楼方向观望，如下图，如果某栋楼在高度上没有被挡住，那就可以被观望到，这里只考虑楼高不考虑其他因素。

可任意选择从某一栋楼前开始观望，观望方向只能从小编号往大编号方向观望，求最大可观望数量。



解码时刻

高低错落的大楼

【输入格式】

第一行一个整数n，表示楼的数量。(1≤n≤1000)

第二行n个正整数，用空格隔开，表示楼的高度。(独栋高度≤1e9)

【输出格式】

一个整数，表示最大的观望数量。

【输入样例】

7

65 28 87 45 30 63 70

【输出样例】

3

样例分析

【输入样例】
7
65 28 87 45 30 63 70
【输出样例】
3

当只有一栋楼时

1号楼

观望数量： 1

只可以看到1栋楼。

样例分析

【输入样例】
7
65 28 87 45 30 63 70
【输出样例】　　当只有一栋楼时
3
1号楼

观望数量：　1

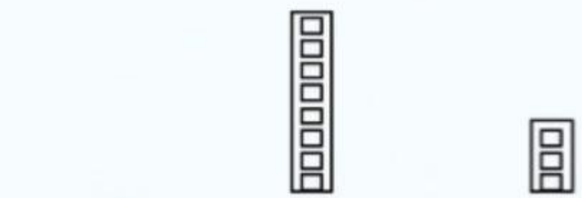
只可以看到1栋楼。

解码时刻

样例分析

【输入样例】
7
65 28 87 45 30 63 70
【输出样例】
3

当有两栋楼时，2号楼作为观看终点



观望数量：1

解码时刻

样例分析

【输入样例】
7
65 28 87 45 30 63 70
【输出样例】
3

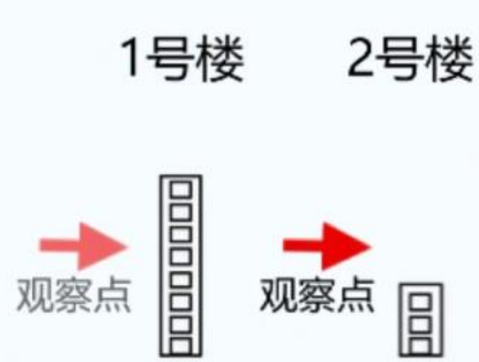
当有两栋楼时，2号楼作为观看终点

观望数量：1

样例分析

【输入样例】
7
65 28 87 45 30 63 70
【输出样例】
3

当有两栋楼时，2号楼作为观看终点



观望数量： 1

可以看到1栋
无法观望到2号楼

解码时刻

样例分析

【输入样例】
7
65 28 87 45 30 63 70
【输出样例】
3

当有两栋楼时，2号楼作为观看终点

可以看到1栋
无法观望到2号楼

解码时刻

样例分析

【输入样例】

7

65 28 87 45 30 63 70

【输出样例】　　当有三栋楼时，3号楼作为观看终点

3

样例分析

【输入样例】
7
65 28 87 45 30 63 70
【输出样例】
3

当有三栋楼时，3号楼作为观看终点

观望数量：　1　　1

可以看到1栋
可以看到2栋
可以看到2栋

解码时刻

样例分析

【输入样例】

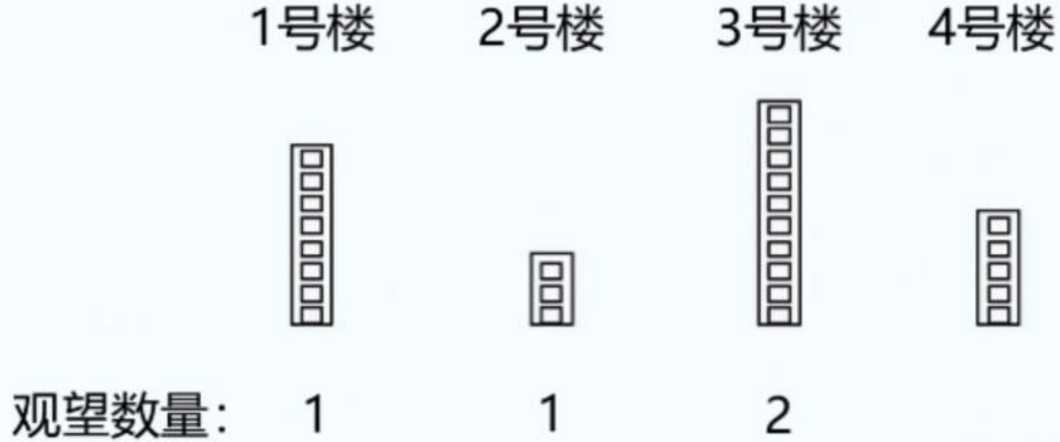
7

65 28 87 45 30 63 70

【输出样例】

3

当有四栋楼时，4号楼作为观看终点

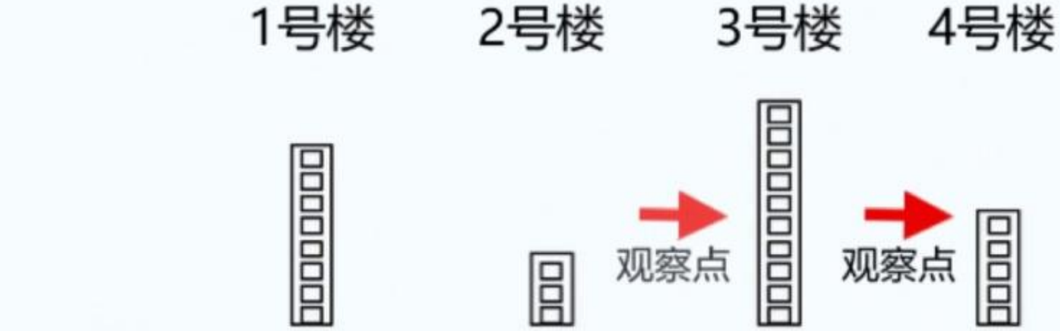


解码时刻

样例分析

【输入样例】
7
65 28 87 45 30 63 70
【输出样例】
3

当有四栋楼时，4号楼作为观看终点



可以看到1栋
从3号楼开始看不到4号楼，不用再往左尝试

解码时刻

样例分析

【输入样例】

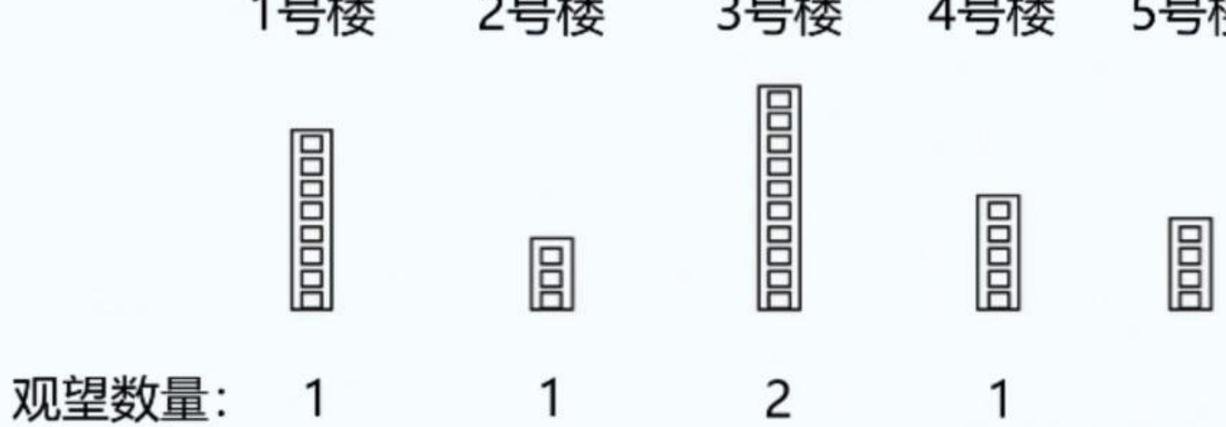
7

65 28 87 45 30 63 70

【输出样例】

3

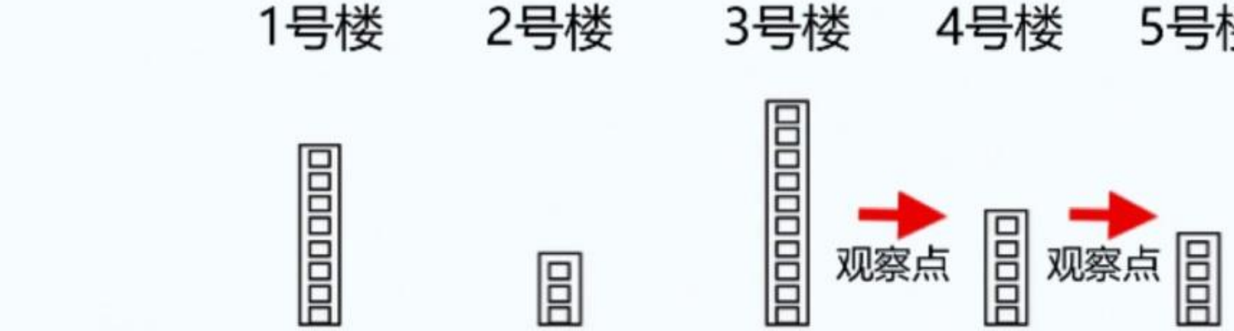
当有五栋楼时，5号楼作为观看终点



样例分析

【输入样例】
7
65 28 87 45 30 63 70
【输出样例】
3

当有五栋楼时，5号楼作为观看终点

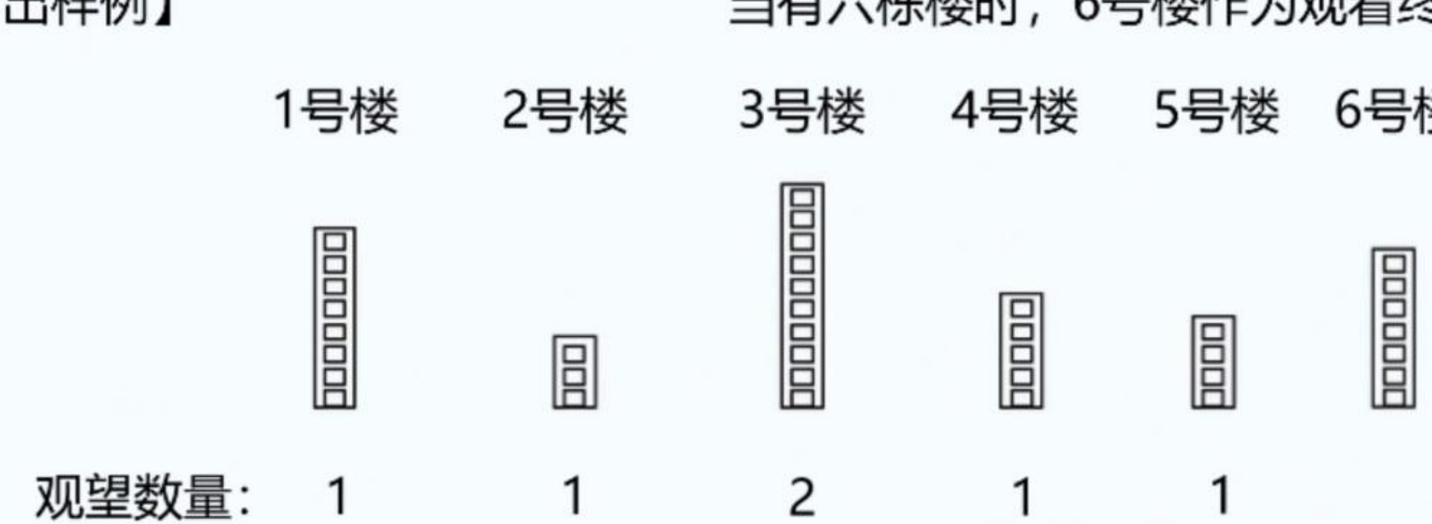


可以看到1栋
从4号楼开始看不到5号楼，不用再往左尝试

样例分析

【输入样例】
7
65 28 87 45 30 63 70
【输出样例】
3

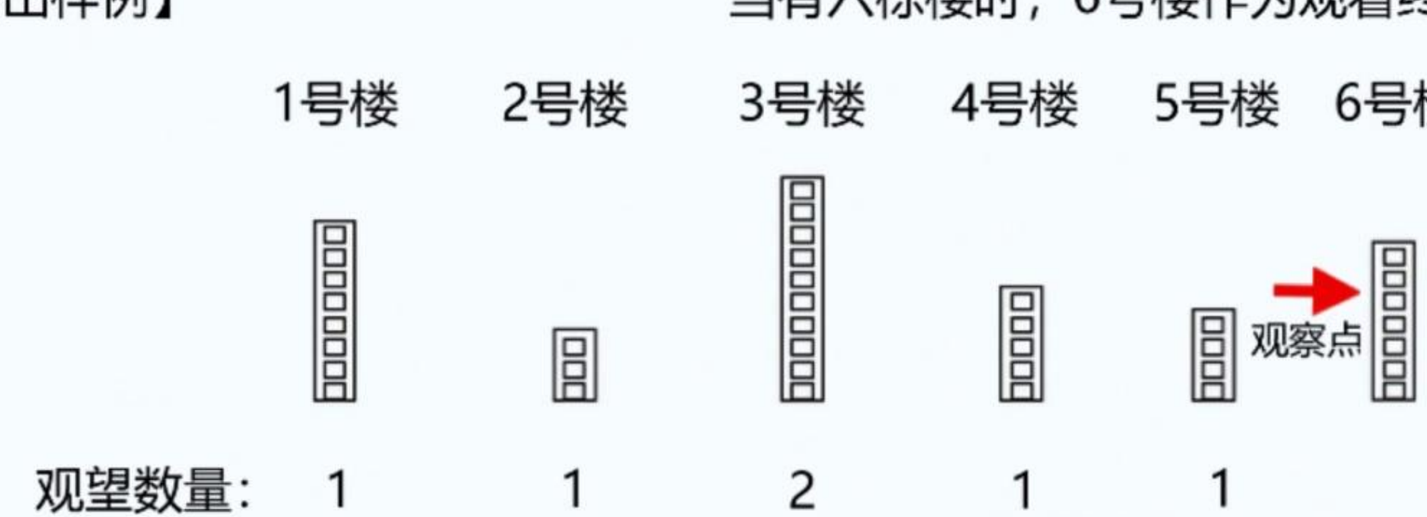
当有六栋楼时，6号楼作为观看终点



样例分析

【输入样例】
7
65 28 87 45 30 63 70
【输出样例】
3

当有六栋楼时，6号楼作为观看终点



可以看到1栋

样例分析

【输入样例】
7
65 28 87 45 30 63 70
【输出样例】
3

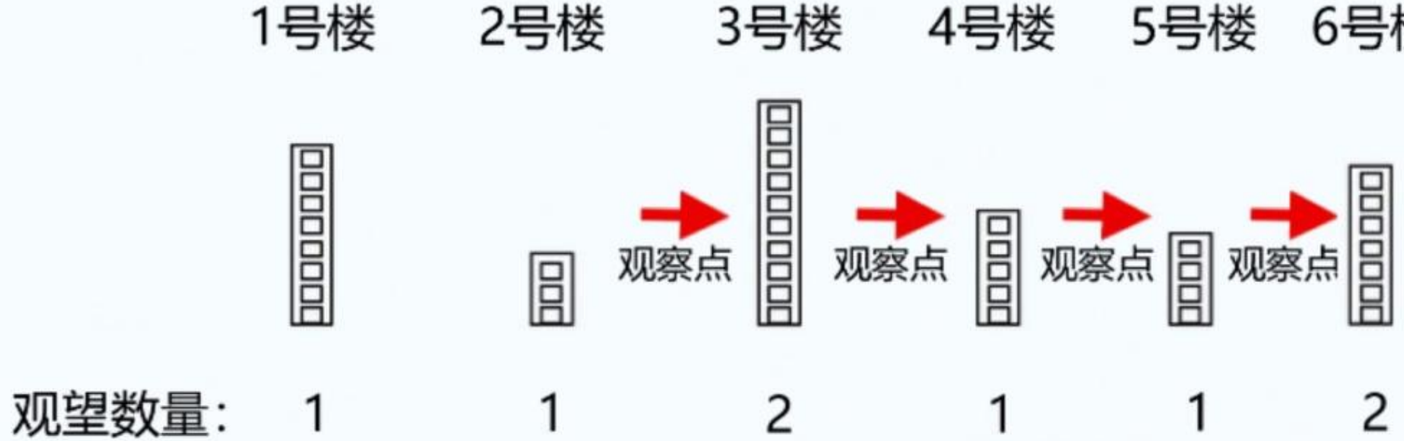
当有六栋楼时，6号楼作为观看终点

可以看到1栋
可以看到2栋
可以看到2栋

样例分析

【输入样例】
7
65 28 87 45 30 63 70
【输出样例】
3

当有六栋楼时，6号楼作为观看终点



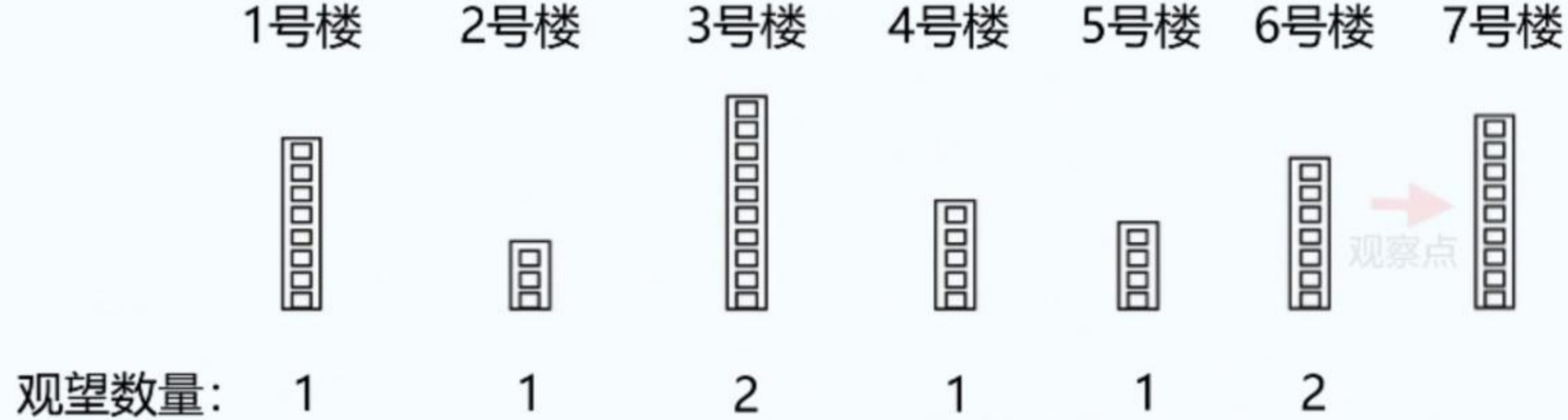
可以看到1栋
可以看到2栋
可以看到2栋
从3号楼开始看不到6号楼，不用再往左尝试

解码时刻

样例分析

【输入样例】
7
65 28 87 45 30 63 70
【输出样例】
3

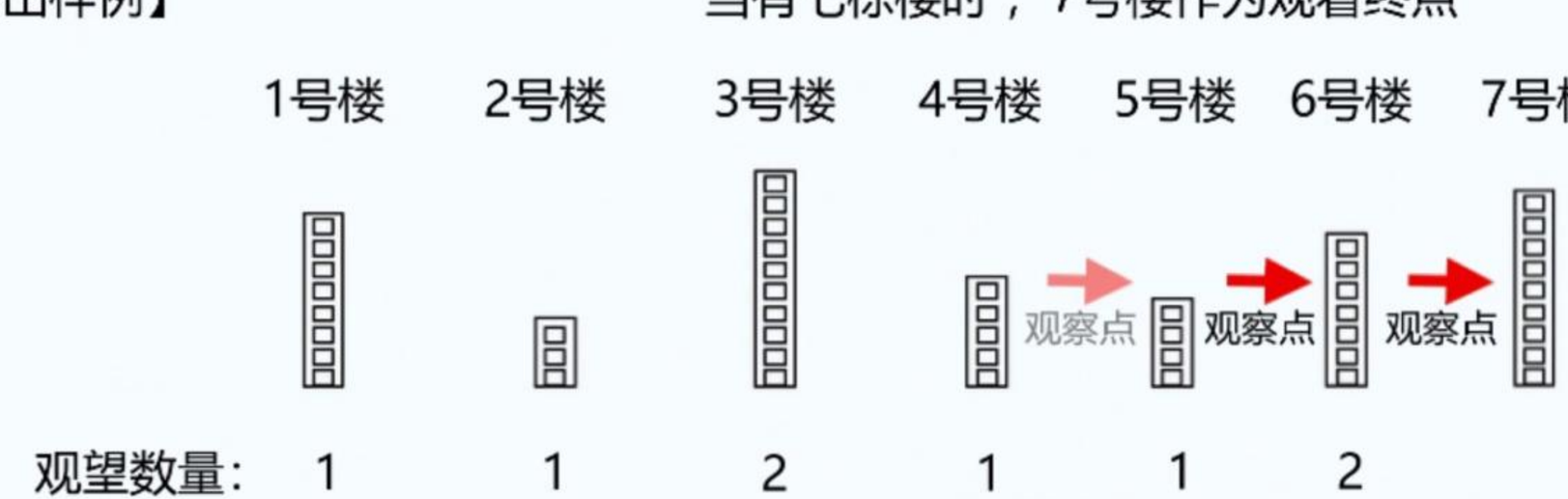
当有七栋楼时，7号楼作为观看终点



样例分析

【输入样例】
7
65 28 87 45 30 63 70
【输出样例】
3

当有七栋楼时，7号楼作为观看终点



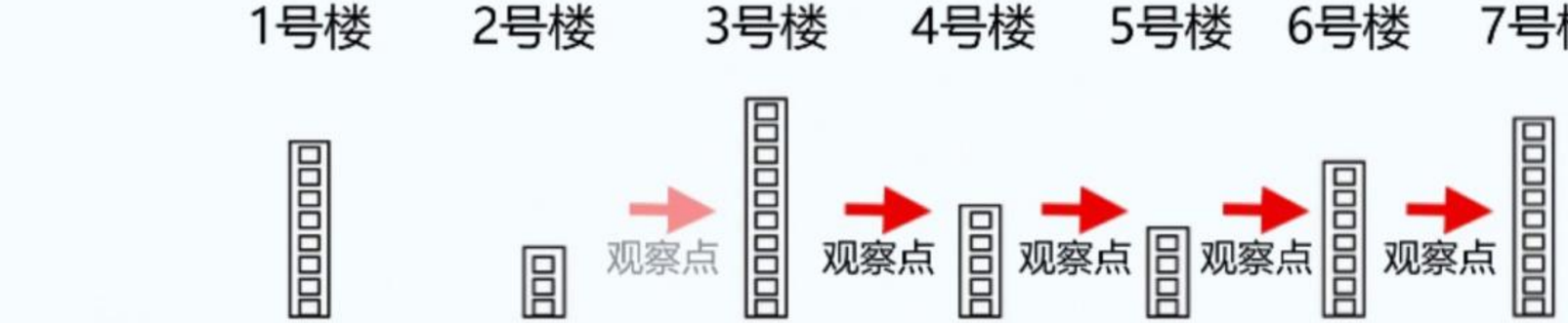
可以看到1栋
可以看到2栋
可以看到3栋

解码时刻

样例分析

【输入样例】
7
65 28 87 45 30 63 70
【输出样例】
3

当有七栋楼时，7号楼作为观看终点



可以看到1栋
可以看到2栋
可以看到3栋
可以看到3栋

从3号楼开始看不到7号楼，
不用再往左尝试

解码时刻

样例分析

【输入样例】

7

65 28 87 45 30 63 70

【输出样例】

3

最大观望数量为3

当有七栋楼时，7号楼作为观看终点

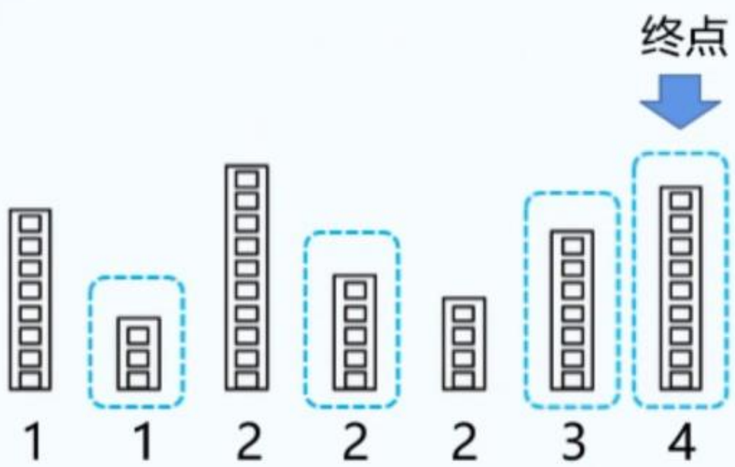
可以看到1栋
可以看到2栋
可以看到3栋
可以看到3栋

从3号楼开始看不到7号楼，
不用再往左尝试

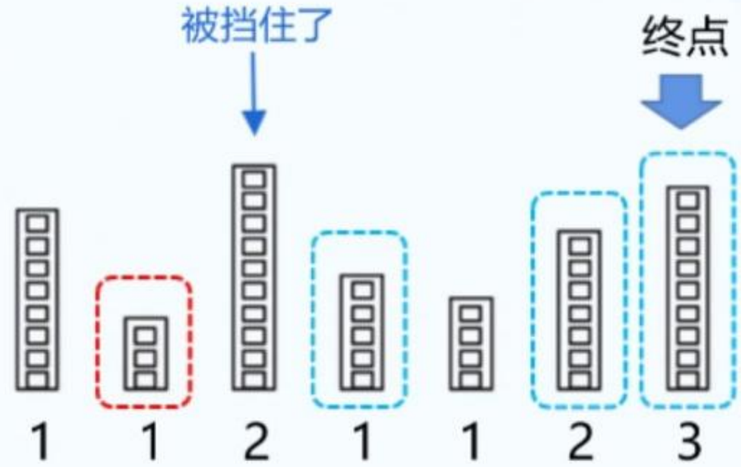
解码时刻

问题分析

与"制作信箱"问题的区别



制作信箱思路，这里最大上升子序列长度4。



此题情况，无法形成长度为4的上升序列。

```
if(a[j]<a[i]) dp[i]=max(dp[i],dp[j]+1)
```

终点左侧可用高度　　终点

两者状态转移方程类似，不同之处是**可用高度的范围**不一样。

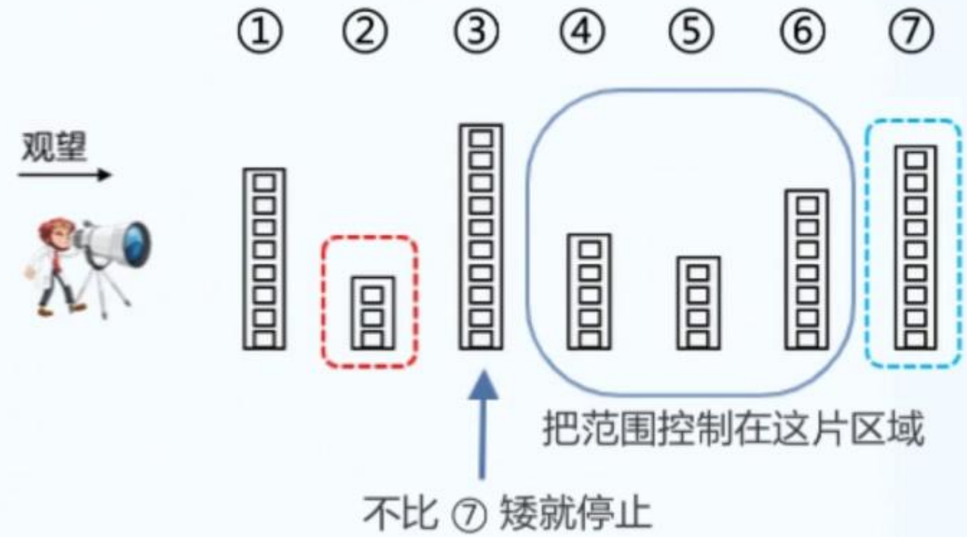
解码时刻

问题分析

调整“制作信箱”的状态转移代码

dp数组：存储以第i栋楼为终点的观望数量
a数组：存储楼高

```
//遍历楼高
for(int i=1; i<=n; i++) {
    //遍历第i栋楼左侧所有楼高
    for(int j=1; j<i; j++) {
        //状态转移
        if(a[j]<a[i]) dp[i]=max(dp[i],dp[j]+1);
    }
}
```



假设求 ⑦ 的观望数量，以上代码会把 ② 也算进来。

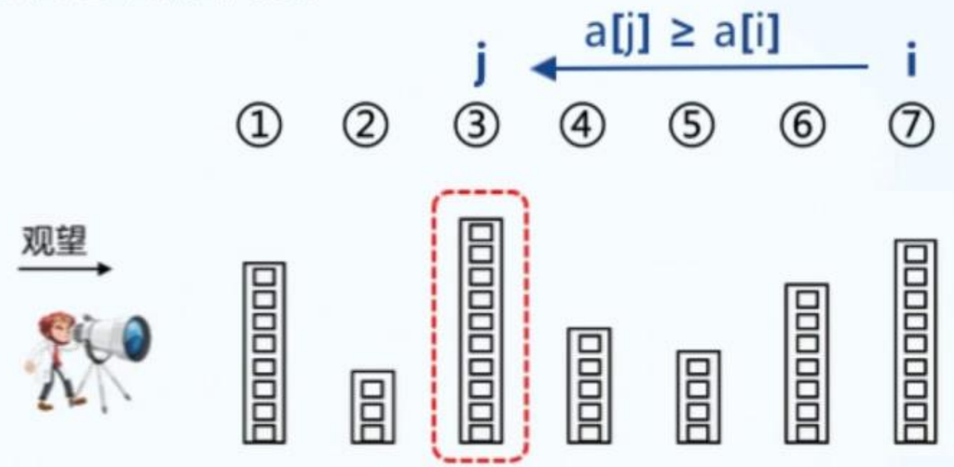
解码时刻

问题分析

调整“制作信箱”的状态转移代码

dp数组：存储以第i栋楼为终点的观望数量
a数组：存储楼高

```
//遍历楼高
for(int i=1; i<=n; i++) {
    //遍历第i栋楼左侧楼高
    for(int j=i-1; j>=1; j--) {
        //状态转移
        if(a[j]<a[i]) dp[i]=max(dp[i],dp[j]+1);
        else break;
    }
}
```



内层循环**从右至左遍历**，当大于等于第 i 栋高度，结束当前循环(内层循环)。

解码时刻

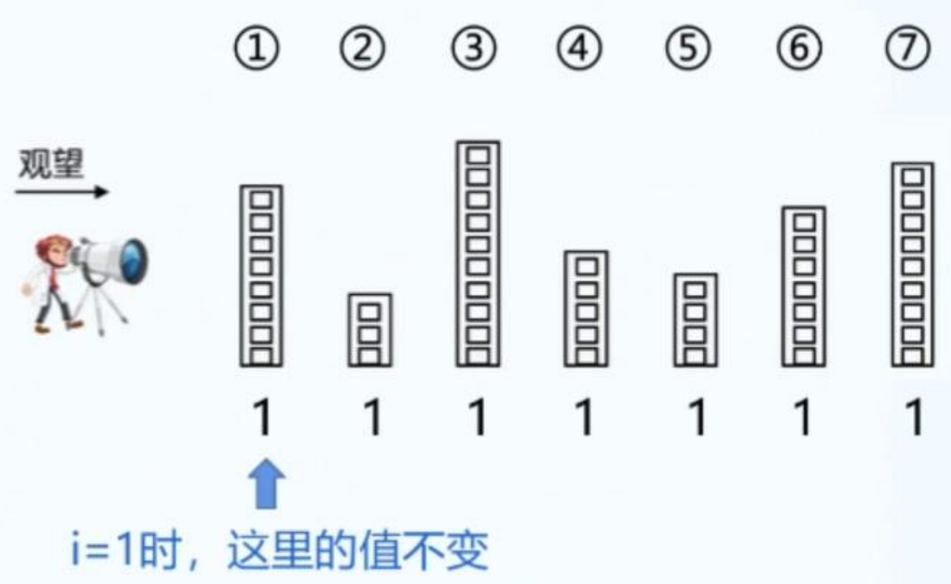
问题分析

调整“制作信箱”的状态转移代码

dp数组：存储以第i栋楼为终点的观望数量
a数组：存储楼高

```
//遍历楼高
for(int i=1; i<=n; i++) {
    //遍历第i栋楼左侧楼高
    for(int j=i-1; j>=1; j--) {
        //状态转移
        if(a[j]<a[i]) dp[i]=max(dp[i],dp[j]+1);
        else break;
    }
}
```

执行流程演示



内层循环**从右至左遍历**，当大于等于第 **i** 栋高度，结束当前循环(内层循环)。

解码时刻

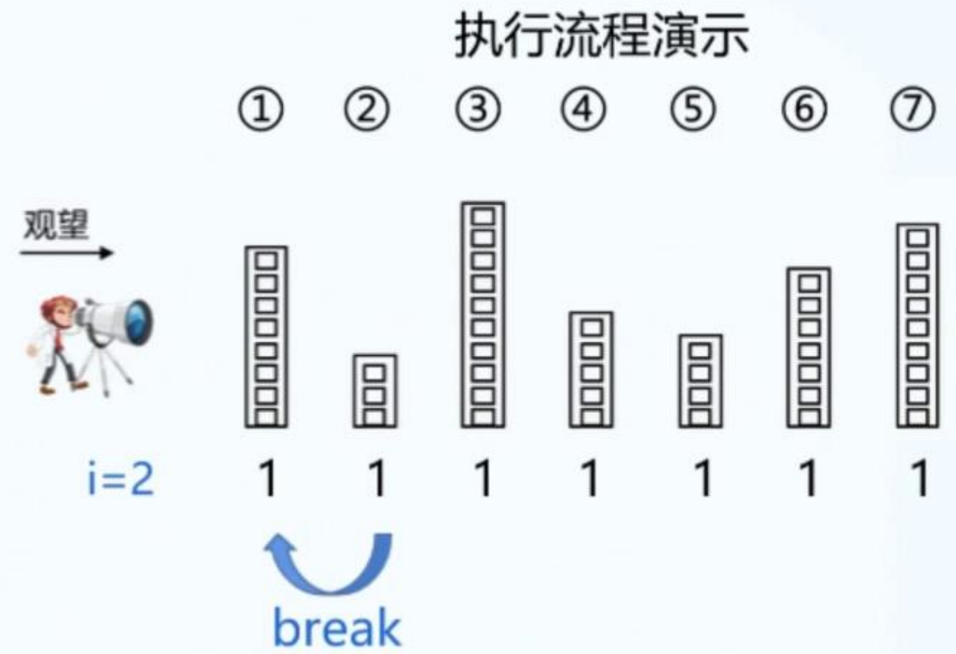
问题分析

调整“制作信箱”的状态转移代码

dp数组：存储以第i栋楼为终点的观望数量
a数组：存储楼高

```
//遍历楼高
for(int i=1; i<=n; i++) {
    //遍历第i栋楼左侧楼高
    for(int j=i-1; j>=1; j--) {
        //状态转移
        if(a[j]<a[i]) dp[i]=max(dp[i],dp[j]+1);
        else break;
    }
}
```

执行流程演示



内层循环**从右至左遍历**，当大于等于第 i 栋高度，结束当前循环(内层循环)。

解码时刻

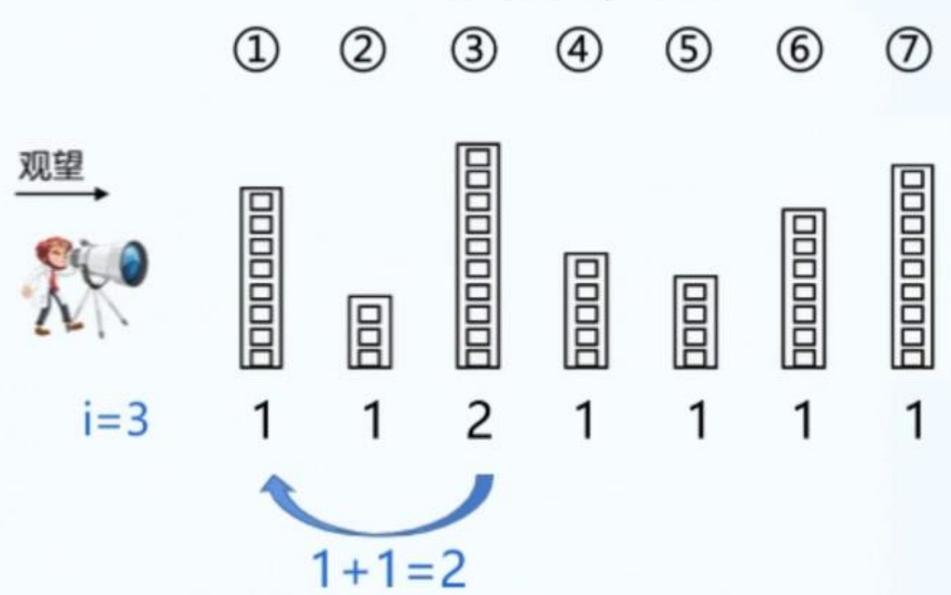
问题分析

调整“制作信箱”的状态转移代码

dp数组：存储以第i栋楼为终点的观望数量
a数组：存储楼高

```
//遍历楼高
for(int i=1; i<=n; i++) {
    //遍历第i栋楼左侧楼高
    for(int j=i-1; j>=1; j--) {
        //状态转移
        if(a[j]<a[i]) dp[i]=max(dp[i],dp[j]+1);
        else break;
    }
}
```

执行流程演示



内层循环**从右至左遍历**，当大于等于第 **i** 栋高度，结束当前循环(内层循环)。

解码时刻

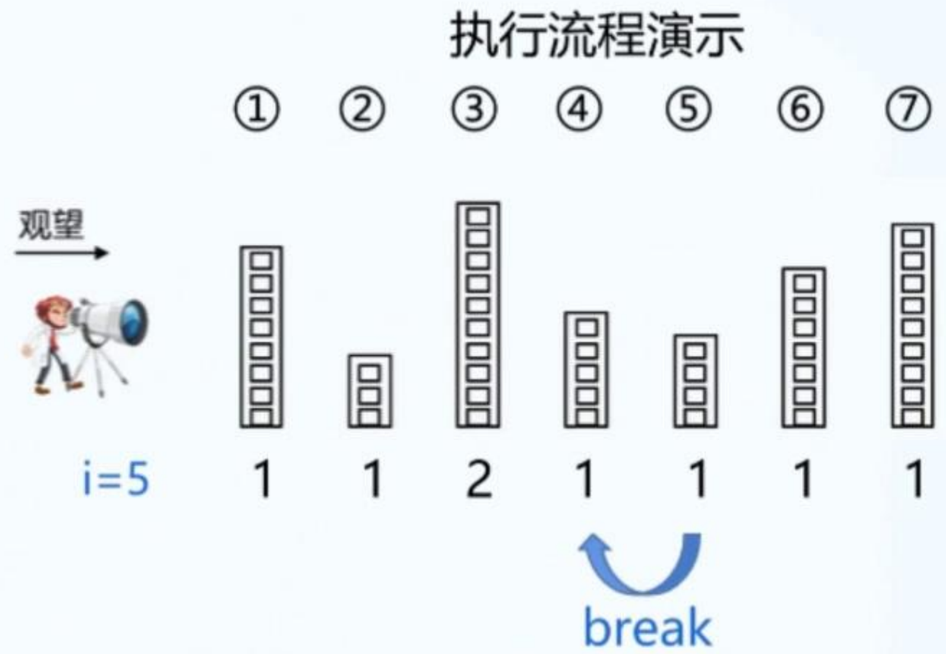
问题分析

调整“制作信箱”的状态转移代码

dp数组：存储以第i栋楼为终点的观望数量
a数组：存储楼高

```
//遍历楼高
for(int i=1; i<=n; i++) {
    //遍历第i栋楼左侧楼高
    for(int j=i-1; j>=1; j--) {
        //状态转移
        if(a[j]<a[i]) dp[i]=max(dp[i],dp[j]+1);
        else break;
    }
}
```

执行流程演示



内层循环**从右至左遍历**，当大于等于第 **i** 栋高度，结束当前循环(内层循环)。

解码时刻

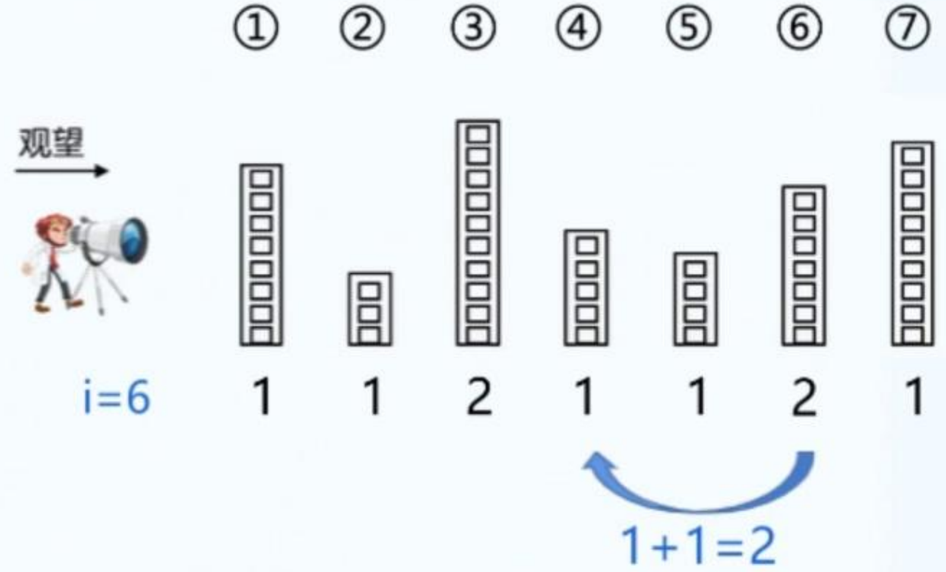
问题分析

调整“制作信箱”的状态转移代码

dp数组：存储以第i栋楼为终点的观望数量
a数组：存储楼高

```
//遍历楼高
for(int i=1; i<=n; i++) {
    //遍历第i栋楼左侧楼高
    for(int j=i-1; j>=1; j--) {
        //状态转移
        if(a[j]<a[i]) dp[i]=max(dp[i],dp[j]+1);
        else break;
    }
}
```

执行流程演示



内层循环**从右至左遍历**，当大于等于第 i 栋高度，结束当前循环(内层循环)。

解码时刻

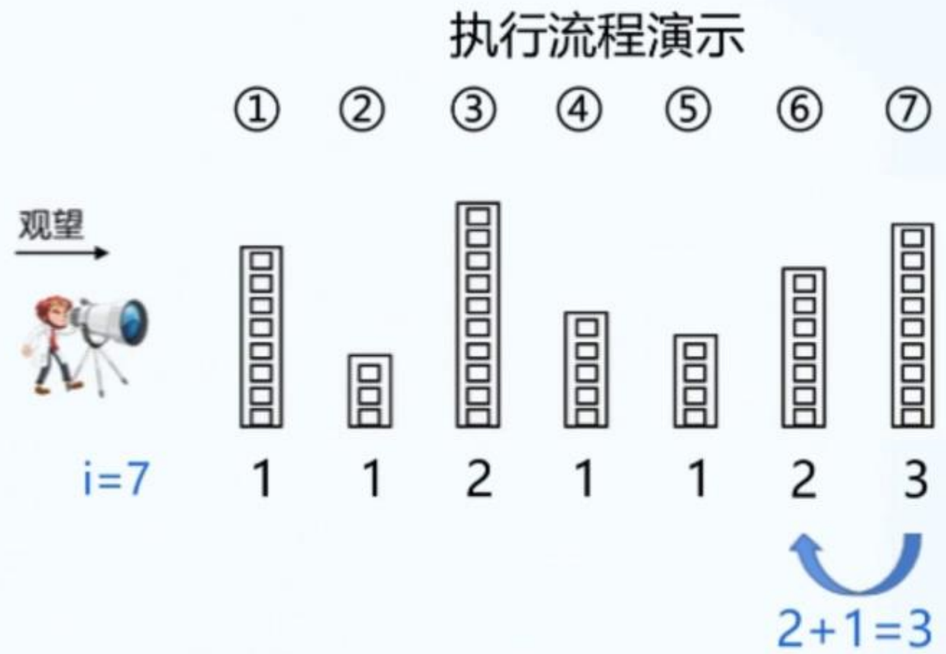
问题分析

调整“制作信箱”的状态转移代码

dp数组：存储以第i栋楼为终点的观望数量
a数组：存储楼高

```
//遍历楼高
for(int i=1; i<=n; i++) {
    //遍历第i栋楼左侧楼高
    for(int j=i-1; j>=1; j--) {
        //状态转移
        if(a[j]<a[i]) dp[i]=max(dp[i],dp[j]+1);
        else break;
    }
}
```

执行流程演示



内层循环**从右至左遍历**，当大于等于第 **i** 栋高度，结束当前循环(内层循环)。

解码时刻

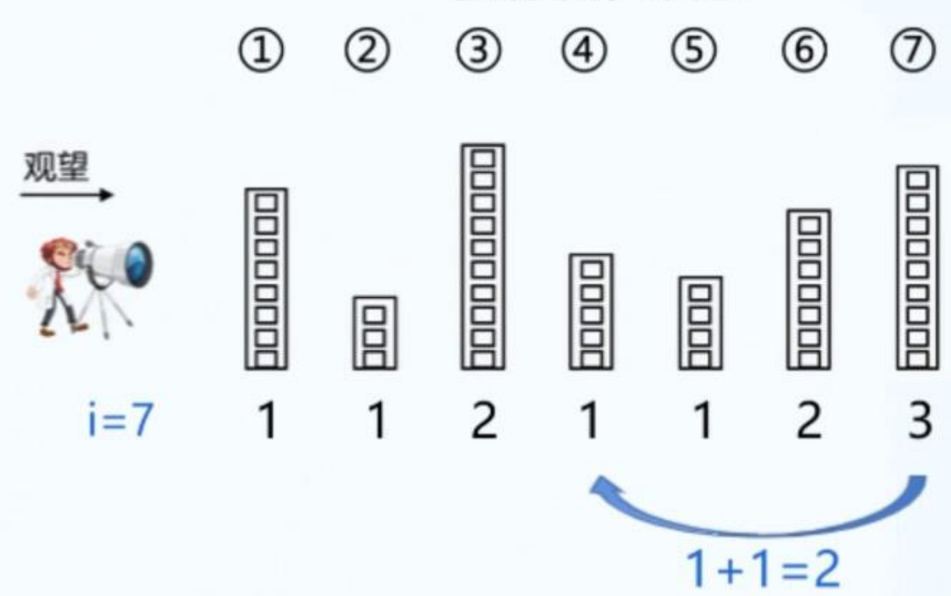
问题分析

调整“制作信箱”的状态转移代码

dp数组：存储以第i栋楼为终点的观望数量
a数组：存储楼高

```
//遍历楼高
for(int i=1; i<=n; i++) {
    //遍历第i栋楼左侧楼高
    for(int j=i-1; j>=1; j--) {
        //状态转移
        if(a[j]<a[i]) dp[i]=max(dp[i],dp[j]+1);
        else break;
    }
}
```

执行流程演示



内层循环**从右至左遍历**，当大于等于第 **i** 栋高度，结束当前循环(内层循环)。

解题步骤

1. 完成数据输入

```
int a[1001], n; //定义在全局，a存储楼高，n存储楼数量
//输入数量
cin>>n;
for (int i=1; i<=n; i++) {
    cin>>a[i]; //输入楼高
}
```

2. 初始化状态

```
int dp[1001]; //定义在全局
for (int i=1; i<=n; i++) {
    dp[i]=1;
}
```

解码时刻

解题步骤

3. 状态转移

```
//遍历楼高
for(int i=1; i<=n; i++){
    //遍历第i栋楼左侧楼高
    for(int j=i-1; j>=1; j--){
                    //状态转移
        if(a[j]<a[i]) dp[i]=max(dp[i],dp[j]+1);
        else break; //出现高度大于等于第i栋楼跳出内层循环
    }
}
```

4. 输出结果

```
int ans=0; //存储结果
for(int i=1;i<=n;i++) {
    ans=max(ans,dp[i]); //保存最大值
}
cout<<ans;
```

完整代码

```
#include<bits/stdc++.h>
using namespace std;
int a[1001], dp[1001], n;
int main() {
    //输入
    cin>>n;
    for (int i=1; i<=n; i++) {
        cin>>a[i];
    }
    //初始化
    for (int i=1; i<=n; i++) {
        dp[i]=1;
    }
    for(int i=1; i<=n; i++){
        for(int j=i-1; j>=1; j--){
            //状态转移
            if(a[j]<a[i]) dp[i]=max(dp[i],dp[j]+1);
            else break;
        }
    }
    int ans=0;
    for(int i=1;i<=n;i++) ans=max(ans,dp[i]);
    //输出结果
    cout<<ans;
    return 0;
}
```

完整代码

```
#include<bits/stdc++.h>
using namespace std;
int a[1001], dp[1001], n;
int main() {
    //输入
    cin>>n;
    for (int i=1; i<=n; i++) {
        cin>>a[i];
    }
    //初始化
    for (int i=1; i<=n; i++) {
        dp[i]=1;
    }
    for(int i=1; i<=n; i++){
        for(int j=i-1; j>=1; j--){
            //状态转移
            if(a[j]<a[i]) dp[i]=max(dp[i],dp[j]+1);
            else break;
        }
    }
    int ans=0;
    for(int i=1;i<=n;i++) ans=max(ans,dp[i]);
    //输出结果
    cout<<ans;
    return 0;
}
```

最长公共子序列

码上得知

最长公共子序列

码上得知

最长公共子序列

码上得知

最长公共子序列

有s1、s2两个数字序列：

s1：5 3 9 11 4 2 15

码上得知

最长公共子序列

有s1、s2两个数字序列：

s1: 5 3 9 11 4 2 15

s2: 3 4 5 7 2 17

最长公共子序列

有s1、s2两个数字序列：

s1：5 3 9 11 4 2 15

s2：3 4 5 7 2 17

什么是公共子序列?

5 2

4 2

3 4 2

...

码上得知

最长公共子序列

有s1、s2两个数字序列：

s1：5 3 9 11 4 2 15

s2：3 4 5 7 2 17

什么是公共子序列？

5 2

4 2

3 4 2

...

这些子序列在s1、s2中共有

码上得知

最长公共子序列

有s1、s2两个数字序列：

s1：5 3 9 11 4 2 15

s2：3 4 5 7 2 17

什么是公共子序列？

5 2

4 2

3 4 2 ⇢ 最长的称为最长公共子序列

...

码上得知

最长公共子序列

有s1、s2两个数字序列：

s1：5 3 9 11 4 2 15

s2：3 4 5 7 2 17

什么是公共子序列？

5 2

4 2

3 4 2 ⇢ 最长的称为最长公共子序列

...

解码时刻

挑选糖果

【问题描述】

小童和小美来到了一个奇怪的糖果店，在挑选糖果时，他们每人需要提前把想要的糖果的名字首字母写下来分别交给店长，店长会仔细对比两个人写的内容，只有两个人挑的是一样的糖果并且挑选的先后顺序也要一样才算数，最终小童和小美每人最多可以获得多少糖果呢？

【输入格式】

两行，每行一个字符串，分别表示小童和小美糖果名字的首字母，字母小写，字母可以重复。（字母范围：a~z，单个字符串长度≤100）

【输出格式】

一个整数，表示两个字符串的最长公共子序列的长度。

【输入样例】

abcbdab

bdcaba

【输出样例】

4

样例分析

【输入样例】
abcbdab
bdcaba
【输出样例】
4

字符串1：a b c b d a b
字符串2：b d c a b a

b c a b

在所有的公共子序列中，最长的公共子序列长度可以达到4。

最长公共子序列问题，同样是经典的动态规划问题。

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b								
2	d								
3	c								
4	a								
5	b								
6	a								
j	s2								

问题分析

		1	2	3	4	5	6	7	i
		a	**b**	**c**	**b**	**d**	**a**	**b**	s1
1	**b**								
2	**d**								
3	**c**								
4	**a**								
5	**b**								
6	**a**								
j	s2								

问题分析

		1	2	3	4	5	6	7	i
		a	**b**	**c**	**b**	**d**	**a**	**b**	s1
1	**b**								
2	**d**								
3	**c**								
4	**a**								
5	**b**								
6	**a**								
j	s2								

s1: a
s2: b

公共子序列
长度为0

问题分析

		1	2	3	4	5	6	7	i
		a	**b**	**c**	**b**	**d**	**a**	**b**	s1
1	**b**	**0**							
2	**d**								
3	**c**								
4	**a**								
5	**b**								
6	**a**								
j	s2								

s1: ab
s2: b

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1						
2	d								
3	c								
4	a								
5	b								
6	a								
j	s2								

s1: ab
s2: b

公共子序列
长度为1

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1						
2	d								
3	c								
4	a								
5	b								
6	a								
j	s2								

s1: abc
s2: b

公共子序列
长度为1

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1					
2	d								
3	c								
4	a								
5	b								
6	a								
j	s2								

s1: abcb
s2: b

公共子序列
长度为1

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1				
2	d								
3	c								
4	a								
5	b								
6	a								
j	s2								

s1: abcbd
s2: b

公共子序列
长度为1

解码时刻

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1			
2	d								
3	c								
4	a								
5	b								
6	a								
j	s2								

s1: abcbda
s2: b

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1		
2	d								
3	c								
4	a								
5	b								
6	a								
j	s2								

s1: abcbda
s2: b

公共子序列
长度为1

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1		
2	d								
3	c								
4	a								
5	b								
6	a								
j	s2								

s1: abcbdab
s2: b

公共子序列
长度为1

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d								
3	c								
4	a								
5	b								
6	a								
j	s2								

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d								
3	c								
4	a								
5	b								
6	a								
j	s2								

s1: a
s2: bd

公共子序列
长度为0

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d	0							
3	c								
4	a								
5	b								
6	a								
j	s2								

s1: a
s2: bd

公共子序列
长度为0

解码时刻

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d	0	1						
3	c								
4	a								
5	b								
6	a								
j	s2								

s1: ab
s2: bd

公共子序列
长度为1

解码时刻

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d	0	1	1					
3	c								
4	a								
5	b								
6	a								
j		s2							

s1: abc
s2: bd

公共子序列
长度为1

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d	0	1	1	1				
3	c								
4	a								
5	b								
6	a								
j		s2							

s1: abcb
s2: bd

公共子序列
长度为1

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d	0	1	1	1	2			
3	c								
4	a								
5	b								
6	a								
j	s2								

s1: abcbd
s2: bd

公共子序列
长度为2

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d	0	1	1	1	2	2		
3	c								
4	a								
5	b								
6	a								
j	s2								

s1: abcbda
s2: bd

公共子序列
长度为2

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d	0	1	1	1	2	2	2	
3	c	0	1	2	2	2	2	2	
4	a	1	1	2	2	2	3	3	
5	b	1	2	2	3	3	3	4	
6	a	1	2	2	3	3	4	4	
j		s2							

s1: abcbdab
s2: bdcaba

公共子序列
长度为4

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d	0	1	1	1	2	2	2	
3	c	0	1	2	2	2	2	2	
4	a	1	1	2	2	2	3	3	
5	b	1	2	2	3	3	3	4	
6	a	1	2	2	3	3	4	4	
j		s2							

表格中长度的上升由两种情况决定。

1. 结尾字母相同
2. 结尾字母不同

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d	0	1	1	1	2	2	2	
3	c	0	1	2	2	2	2	2	
4	a	1	1	2	2	2	3	3	
5	b	1	2	2	3	3	3	4	
6	a	1	2	2	3	3	4	4	
j		s2							

公共子序列长度上升情况划分。

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d	0	1	1	1	2	2	2	
3	c	0	1	2	2	2	2	2	
4	a	1	1	2	2	2	3	3	
5	b	1	2	2	3	3	3	4	
6	a	1	2	2	3	3	4	4	
j	s2								

公共子序列长度上升情况划分。

s1: a b c b d
s2: b d

结尾出现相同字母

d与d公共长度为1。

解码时刻

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d	0	1	1	1	2	2	2	
3	c	0	1	2	2	2	2	2	
4	a	1	1	2	2	2	3	3	
5	b	1	2	2	3	3	3	4	
6	a	1	2	2	3	3	4	4	
j		s2							

公共子序列长度上升情况划分。

s1: a b c b d
s2: b d

结尾出现相同字母

d与d公共长度为1。

把状态还原成没有出现字母d的状态

s1: a b c b
s2: b

公共子序列长度为1

所以下列公共长度为：1+1=2

s1: a b c b | d
s2: b | d

1 + 1

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d	0	1	1	1	2	2	2	
3	c	0	1	2	2	2	2	2	
4	a	1	1	2	2	2	3	3	
5	b	1	2	2	3	3	3	4	
6	a	1	2	2	3	3	4	4	
j		s2							

公共子序列长度上升情况划分。

s1: a b c b d
s2: b d

结尾出现相同字母

d与d公共长度为1。

把状态还原成没有出现字母d的状态

s1: a b c b
s2: b

公共子序列长度为1

所以下列公共长度为：1+1=2

s1: a b c b | d
s2: b | d

1 + 1

解码时刻

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d	0	1	1	1	2	2	2	
3	c	0	1	2	2	2	2	2	
4	a	1	1	2	2	2	3	3	
5	b	1	2	2	3	3	3	4	
6	a	1	2	2	3	3	4	4	
j		s2							

公共子序列长度上升情况划分。

s1: a b c

s2: b d c a

结尾字母不相同

从左上方转移

s1: a b | c

s2: b d c | a

1 + 0

1+0=1

结果不对

解码时刻

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d	0	1	1	1	2	2	2	
3	c	0	1	2	2	2	2	2	
4	a	1	1	2	2	2	3	3	
5	b	1	2	2	3	3	3	4	
6	a	1	2	2	3	3	4	4	
j		s2							

公共子序列长度上升情况划分。

s1: a b c

s2: b d c a

结尾字母不相同

从左上方转移

s1: a b | c

s2: b d c | a

1 + 0

1+0=1

结果不对

c

a

整体看没有公共字母

s1: a b c

s2: b d c a

但单独看可能产生更长的公共序列

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d	0	1	1	1	2	2	2	
3	c	0	1	2	2	2	2	2	
4	a	1	1	2	2	2	3	3	
5	b	1	2	2	3	3	3	4	
6	a	1	2	2	3	3	4	4	
j		s2							

公共子序列长度上升情况划分。

s1: a b c

s2: b d c a

结尾字母不相同

分别对比以下两种情况：

s1: a b c

s2: b d c

← 最长公共子序列长度为2

s1: a b

s2: b d c a

← 最长公共子序列长度为1

问题分析

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d	0	1	1	1	2	2	2	
3	c	0	1	2	2	2	2	2	
4	a	1	1	2	2	2	3	3	
5	b	1	2	2	3	3	3	4	
6	a	1	2	2	3	3	4	4	
j		s2							

公共子序列长度上升情况划分。

s1: a b c

s2: b d c a

所以当前状态最长公共子序列长度为2

当结尾字母不相同

如（4,3）位置

它的最优解为：

max((3,3),(4,2))

从这两种里选择最大

解码时刻

问题分析

状态转移方程

s1、**s2**分别存储两个字符串，**dp**存储不同阶段的字符串能形成的最长公共子序列。

① s1[i]==s2[j]
dp[i][j]=dp[i-1][j-1]++;
② s1[i]!=s2[j]
dp[i][j]=max(dp[i-1][j]，dp[i][j-1]);

两种情况：结尾字母相等时、结尾字母不相等时。

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d	0	1	1	1	2	2	2	
3	c	0	1	2	2	2	2	2	
4	a	1	1	2	2	2	3	3	
5	b	1	2	2	3	3	3	4	
6	a	1	2	2	3	3	4	4	
j	s2								

解码时刻

问题分析

状态转移方程计算

如右侧情况

① s1[i]==s2[j]
dp[i][j]=dp[i-1][j-1]++;

s1[4]='b' s2[1]='b' ;
dp[4][1] = dp[4-1][1-1]++ ;

s1下标　s2下标

dp[3][0]

表格中没有dp[3][0]，需要添加

		1	2	3	4	5	6	7	i
		a	b	c	b	d	a	b	s1
1	b	0	1	1	1	1	1	1	
2	d	0	1	1	1	2	2	2	
3	c	0	1	2	2	2	2	2	
4	a	1	1	2	2	2	3	3	
5	b	1	2	2	3	3	3	4	
6	a	1	2	2	3	3	4	4	
j	s2								

解码时刻

问题分析

状态转移方程计算

如右侧情况

① s1[i]==s2[j]
dp[i][j]=dp[i-1][j-1]++;

s1[4]='b' s2[1]='b' ;
dp[4][1] = dp[4-1][1-1]++ ;

s1下标　s2下标　dp[3][0]

表格中没有dp[3][0]，需要添加

如下位置同理：

dp[2][1] = dp[2-1][1-1]++ ;
dp[1][4] = dp[1-1][4-1]++ ;

… …

		0	1	2	3	4	5	6	7	i
			a	b	c	b	d	a	b	s1
0			0	0	0	0	0	0	0	
1	b	0	0	1	1	1	1	1	1	
2	d	0	0	1	1	1	2	2	2	
3	c	0	0	1	2	2	2	2	2	
4	a	0	1	1	2	2	2	3	3	
5	b	0	1	2	2	3	3	3	4	
6	a	0	1	2	2	3	3	4	4	
j	s2									

添加第0行、第0列，初始化为0

问题分析

状态转移方程计算

如右侧情况

① s1[i]==s2[j]
dp[i][j]=dp[i-1][j-1]++;

② s1[i]!=s2[j]
dp[i][j]=max(dp[i-1][j], dp[i][j-1]);

s1[3]='c' s2[1]='b' ;
dp[3][1] =
max (dp[3-1][1] , dp[3][1-1]);

dp[2][1] dp[3][0]

		0	1	2	3	4	5	6	7	i
			a	b	c	b	d	a	b	s1
0			0	0	0	0	0	0	0	
1	b	0	0	1	1	1	1	1	1	
2	d	0	0	1	1	1	2	2	2	
3	c	0	0	1	2	2	2	2	2	
4	a	0	1	1	2	2	2	3	3	
5	b	0	1	2	2	3	3	3	4	
6	a	0	1	2	2	3	3	4	4	
j	s2									

如下位置同理：

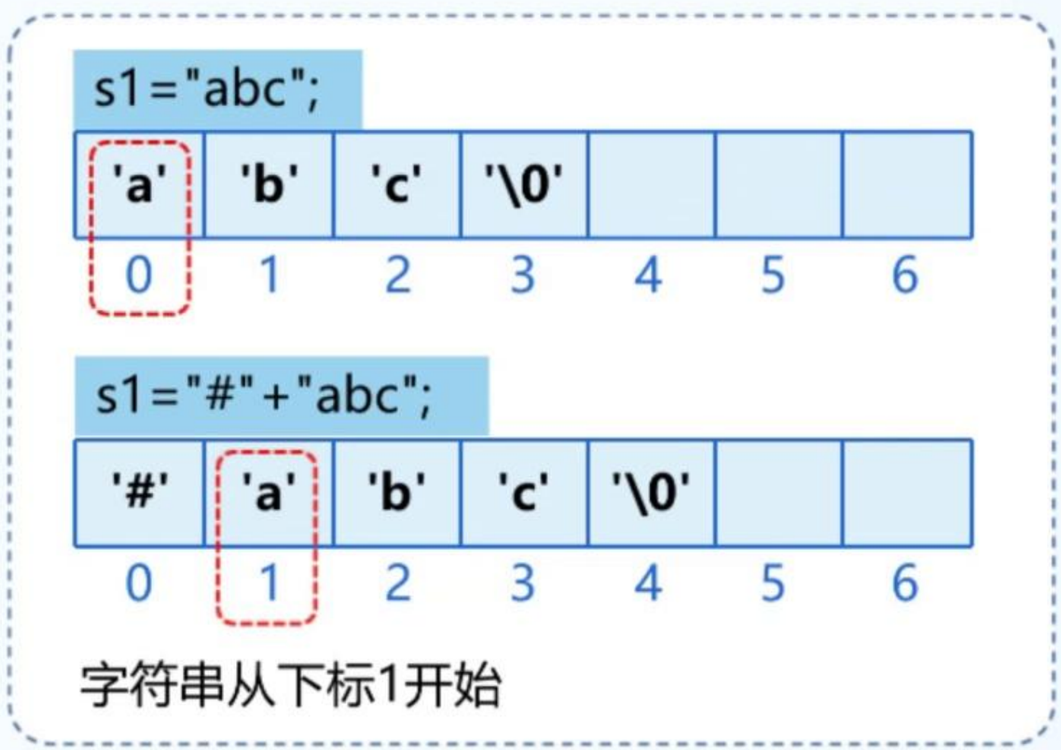
dp[1][2] = max (dp[1-1][2] , dp[1][2-1]);

… …

解题步骤

1. 数据输入

```
string s1, s2;
cin>>s1>>s2;
//获取字符串长度
int m=s1.size();
int n=s2.size();
//为了在循环中使用方便
//使字符串的首字符从下标1开始
s1="#"+s1;
s2="#"+s2;
```



解题步骤

2. 初始化及状态转移

```
int dp[1001][1001]; //在全局声明，所有元素默认初始化为0

//m表示第1个字符串长度
for(int i=1;i<=m;i++){
    //n表示第2个字符串长度
    for(int j=1;j<=n;j++){
        //判断两字符串结尾字母
        if(s1[i]==s2[j]) dp[i][j]=dp[i-1][j-1]+1;
        else dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
    }
}
```

前面已经把字符串首下标改到1，
这样i-1、j-1不会发生越界。

3. 输出结果

```
cout<<dp[m][n];
```

完整代码

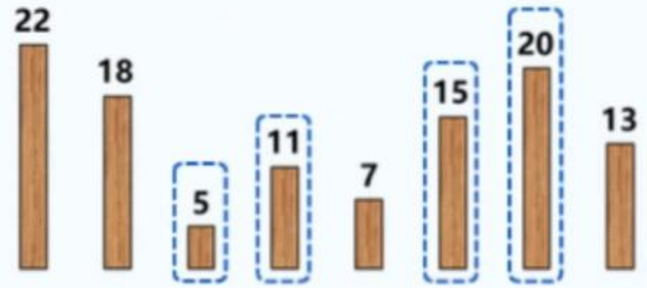
```
#include<bits/stdc++.h>
using namespace std;
int dp[1001][1001];
int main() {
    // 输入数据
    string s1, s2;
    cin>>s1>>s2;
    //获取字符串长度
    int m=s1.size();
    int n=s2.size();
    //使字符串从下标1开始
    s1="#"+s1;
    s2="#"+s2;
    //状态转移
    for(int i=1;i<=m;i++){
        for(int j=1;j<=n;j++){
            if(s1[i]==s2[j]) dp[i][j]=dp[i-1][j-1]+1;
            else dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
        }
    }
    //输出结果
    cout<<dp[m][n];
    return 0;
}
```

完整代码

```
#include<bits/stdc++.h>
using namespace std;
int dp[1001][1001];
int main() {
    // 输入数据
    string s1, s2;
    cin>>s1>>s2;
    //获取字符串长度
    int m=s1.size();
    int n=s2.size();
    //使字符串从下标1开始
    s1="#"+s1;
    s2="#"+s2;
    //状态转移
    for(int i=1;i<=m;i++){
        for(int j=1;j<=n;j++){
            if(s1[i]==s2[j]) dp[i][j]=dp[i-1][j-1]+1;
            else dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
        }
    }
    //输出结果
    cout<<dp[m][n];
    return 0;
}
```

码上总结

最长上升子序列



状态转移方程

dp数组：存储以第 i 根木棍为最后1根，所形成的最长上升子序列长度
a数组：存储木棍长度

if(a[j]<a[i]) dp[i]=max(dp[i],dp[j]+1)

a[i]：第 i 根木根长度
a[j]：第 i 根左侧木棍长度

码上总结

最长公共子序列

例如：
s1：1 2 3 4 5
s2：2 3 4 5 6

序列：2 3
序列：3 4
序列：2 3 4 5

在这些公共子序列中，序列：2 3 4 5，最长。

所以，s1、s2的最长公共子序列为 2 3 4 5。

码上总结

最长公共子序列

求解最长公共子序列的状态转移方程

1.从两个序列长度为1的阶段开始处理
2.判断序列末尾元素，做状态转移

末尾元素相等：

s1[i]==s2[j]，ans[i][j]=ans[i-1][j-1]++

末尾元素不相等：

s1[i]!=s2[j]，ans[i][j]=max(ans[i-1][j]，ans[i][j-1])

上升点列

【问题描述】

在一个二维平面内，给定 n 个整数点 (x_i, y_i)，此外你还可以自由添加 k 个整数点。

你在自由添加 k 个点后，还需要从 $n+k$ 个点中选出若干个整数点并组成一个序列，使得序列中任意相邻两点间的欧几里得距离恰好为 1 而且横坐标、纵坐标值均单调不减，即 $x_{i+1}-x_i=1, y_{i+1}=y_i$ 或 $y_{i+1}-y_i=1, x_{i+1}=x_i$。请给出满足条件的序列的最大长度。

【输入格式】

第一行两个正整数 n,k 分别表示给定的整点个数、可自由添加的整点个数。$1\le n\le 500$，$0\le k\le 100$。

接下来 n 行，第 i 行两个正整数 x_i，y_i 表示给定的第 i 个点的横纵坐标。$1\le x_i,y_i\le 10^9$。

【输出格式】

输出一个整数表示满足要求的序列的最大长度。

上升点列

【输入样例1】
8 2
3 1
3 2
3 3
3 6
1 2
2 2
5 5
5 3
【输出样例1】
8

【输入样例2】
4 100
10 10
15 25
20 20
30 30
【输出样例2】
103

样例分析

【输入样例1】

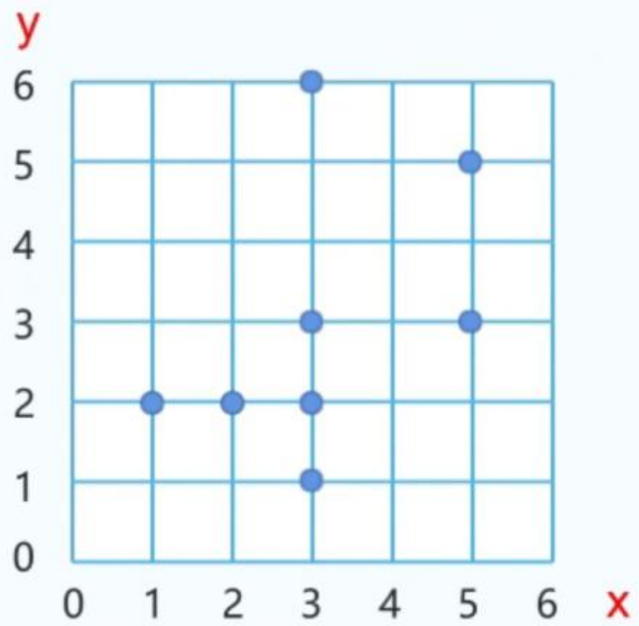
```
8 2
3 1
3 2
3 3
3 6
1 2
2 2
5 5
5 3
```

8 2 → 已知点有8个，可以再添加两个点

3 1 … 5 3 → 已知点的坐标

【输出样例1】

```
8
```



①把点连成路径
②路径点与点间距为1
③以上图画法路径方向只能向右、向上
④路径点的数量要尽量多
⑤样例1数据表示可以再额外添加两个点到图中

样例分析

【输入样例1】

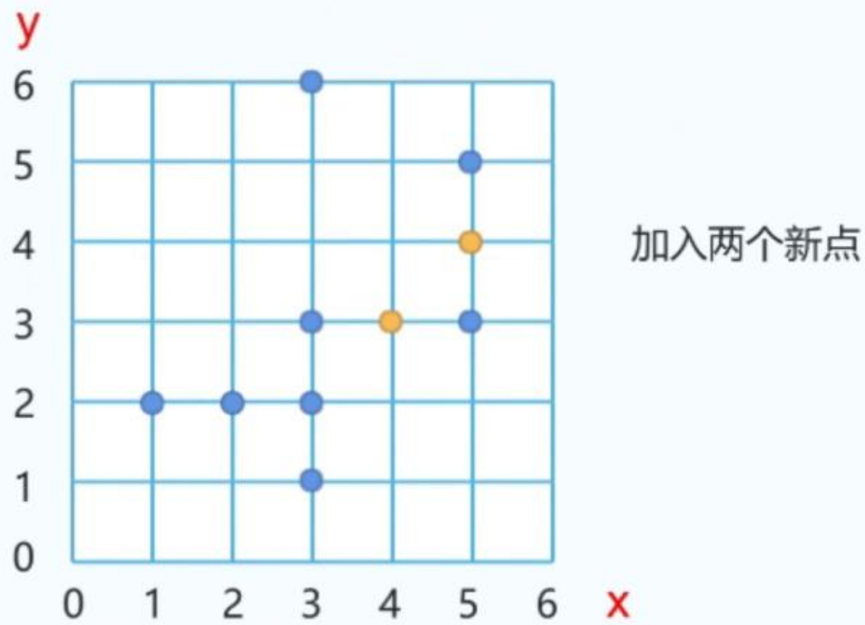
```
8 2
3 1
3 2
3 3
3 6
1 2
2 2
5 5
5 3
```

8 2 → 已知点有8个，可以再添加两个点

3 1 … 5 3 → 已知点的坐标

【输出样例1】

```
8
```



①把点连成路径
②路径点与点间距为1
③以上图画法路径方向只能向右、向上
④路径点的数量要尽量多
⑤样例1数据表示可以再额外添加两个点到图中

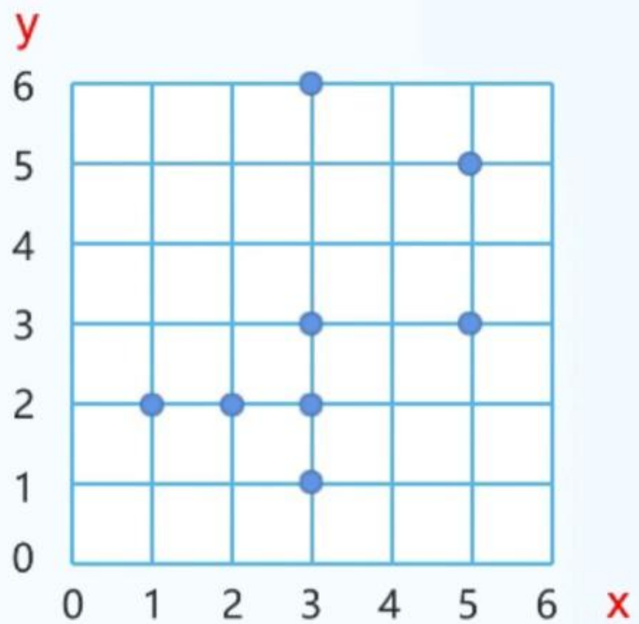
快码加鞭

问题分析

可以把每一个点看做是一个小问题，这样问题就简单很多。

把每一个点都看做终点，分别求他们的最优解。

需要遍历所有的点，而点在使用时x、y的值需要单调递增，可以先对点进行排序。



优先x值小的排在前，x值相等y值小的排在前

(1,2) (2,2) (3,1) (3,2) (3,3) (3,6) (5,3) (5,5)

这样排序的好处是，可以保证x、y值都形成单调减的点只出现在当前点的前面，方便查找。

解释：

$x1 \le x2$

$y1 \le y2$

(x1,y1) 在 (x2,y2) 之前

x1=x2 y1=y2 不同时取等，同时相等代表的是相同点，输入数据中没有相同点。

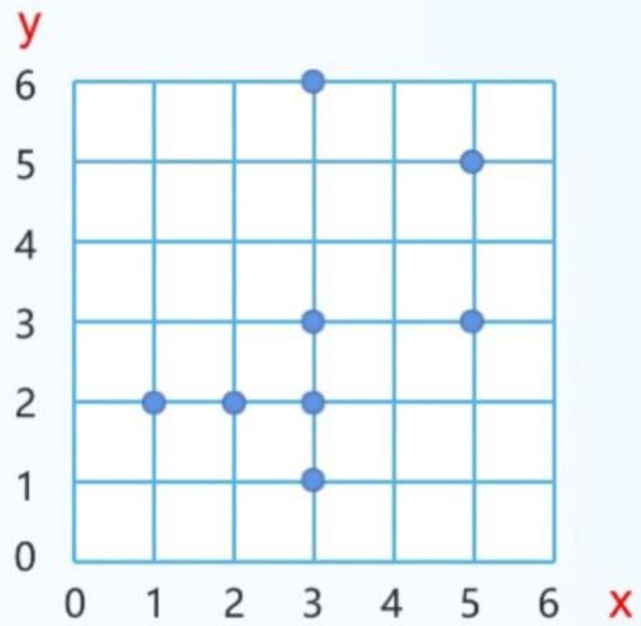
问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0									
1									
2									

↑可添加的点数

以不同点作为终点，尝试不同的添加点数，每种情况的最优解体现在表格中。

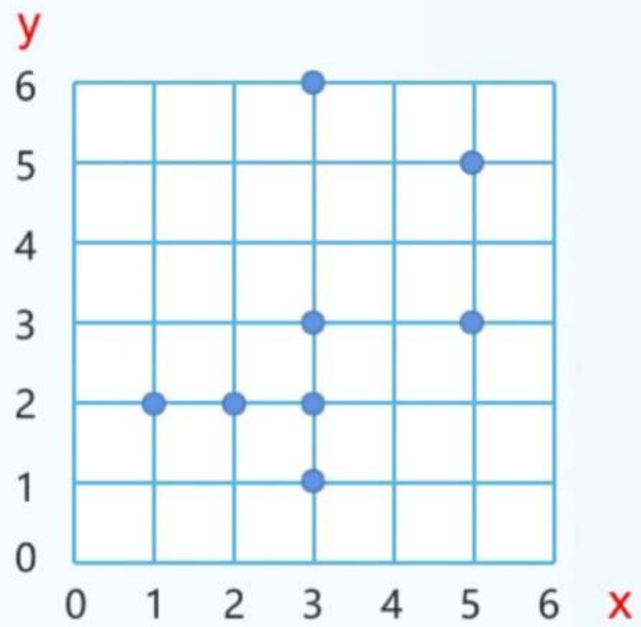


问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0									
1									
2									

可添加的点数**为0时**



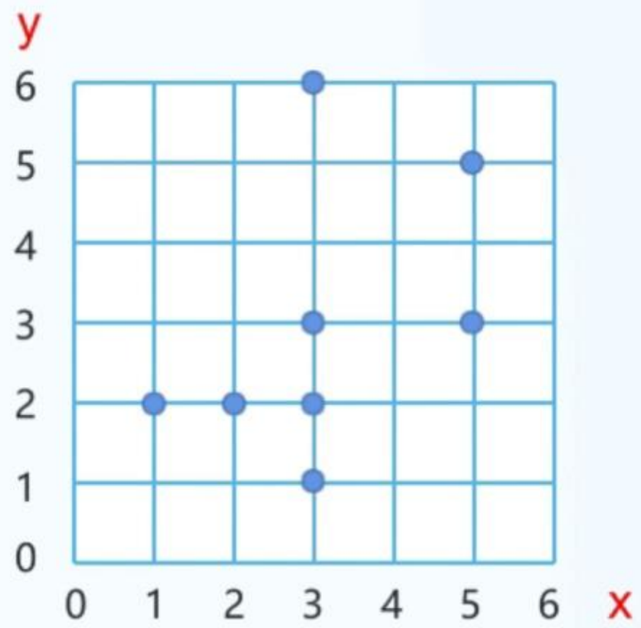
问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**								
1									
2									

可添加的点数**为0时**

(1,2)为终点时，长度为1　　前方无点



快码加鞭

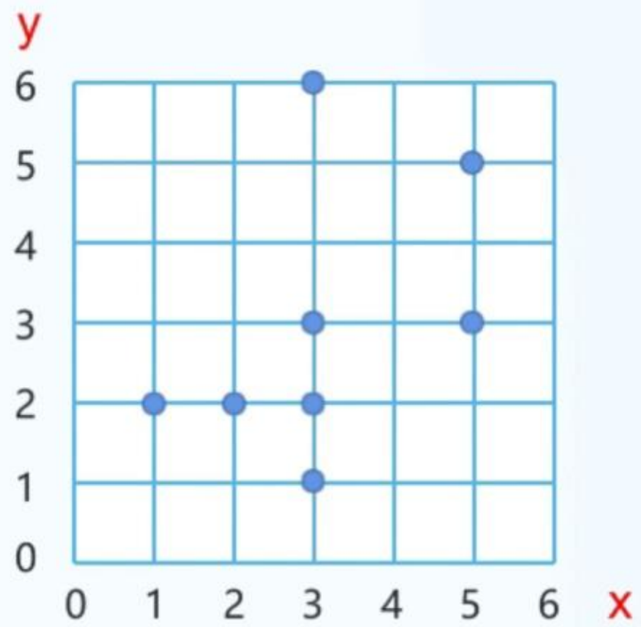
问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**						
1									
2									

可添加的点数**为0时**

(1,2)为终点时，长度为1　　前方无点
(2,2)为终点时，长度为2　　(1,2)-->(2,2)
(3,1)为终点时，长度为1　　前方无点

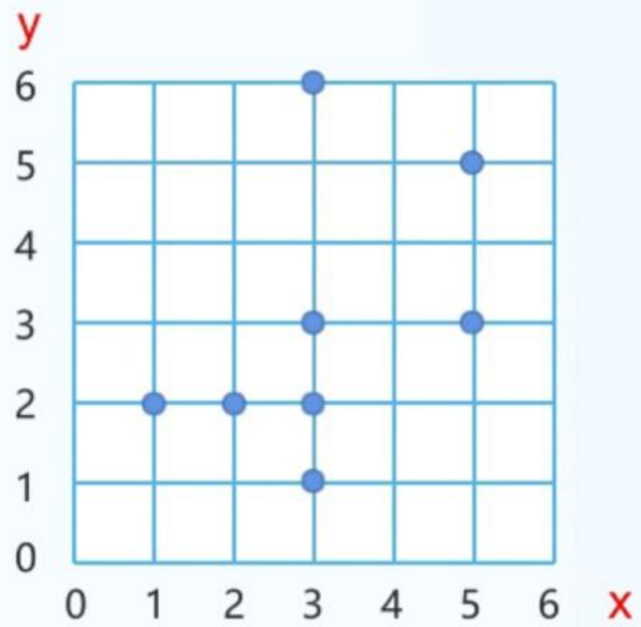


问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**				
1									
2									

可添加的点数**为0时**



(1,2)为终点时，长度为1　前方无点

(2,2)为终点时，长度为2　(1,2)-->(2,2)

(3,1)为终点时，长度为1　前方无点

(3,2)为终点时，长度为3　(1,2)-->(2,2)-->(3,2)　选长的
(3,1)-->(3,2)

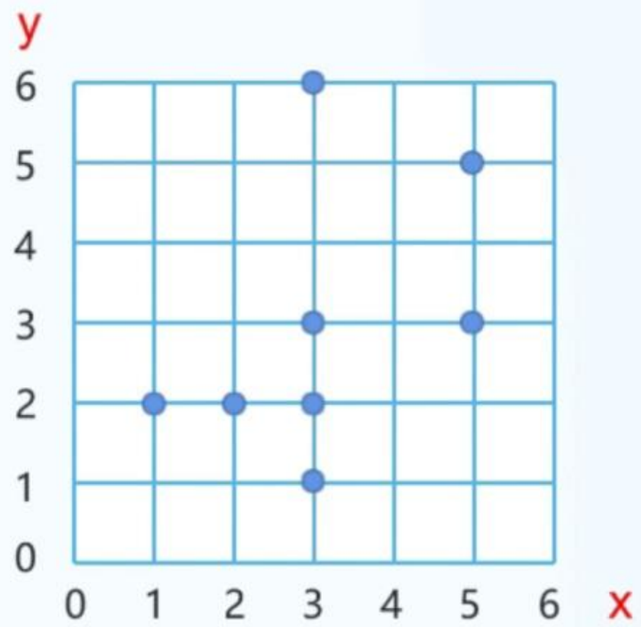
(3,3)为终点时，长度为4　(1,2)-->(2,2)-->(3,2)-->(3,3)　选长的
(3,1)-->(3,2)-->(3,3)

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1									
2									

可添加的点数**为1时**



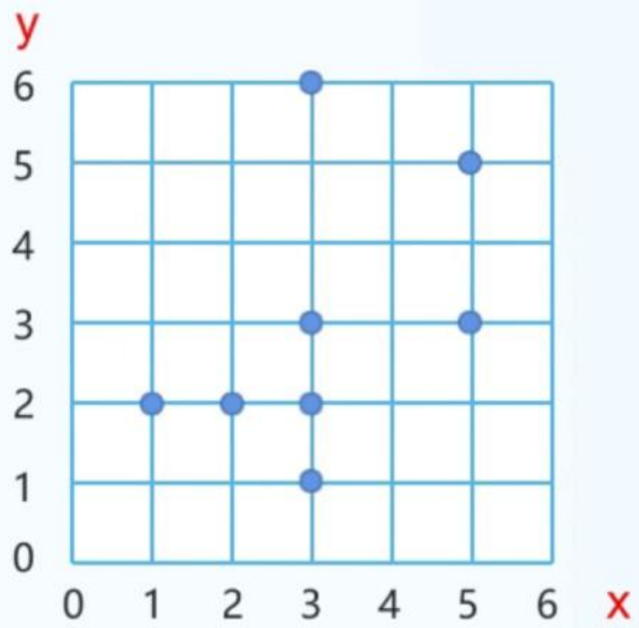
问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**							
2									

可添加的点数**为1时**

(1,2)为终点时，长度为2　　+1个点
(2,2)为终点时，长度为3　　+1个点　(1,2)-->(2,2)



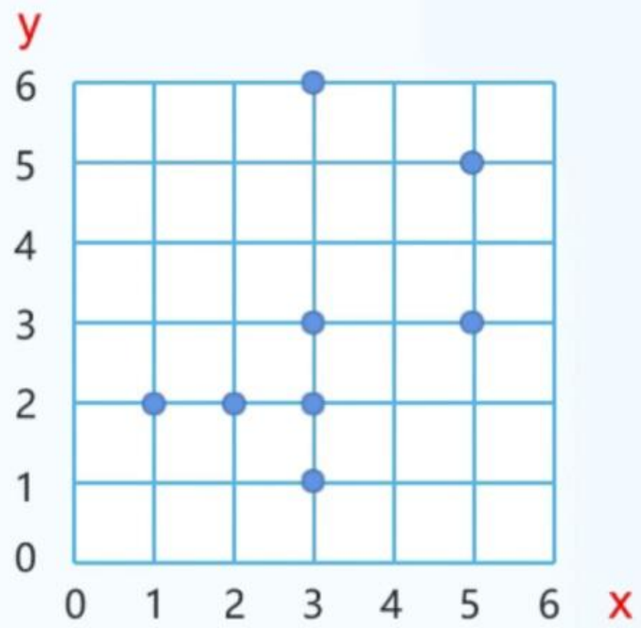
问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**						
2									

可添加的点数**为1时**

(1,2)为终点时，长度为2　+1个点
(2,2)为终点时，长度为3　+1个点　(1,2)-->(2,2)
(3,1)为终点时，长度为2　+1个点



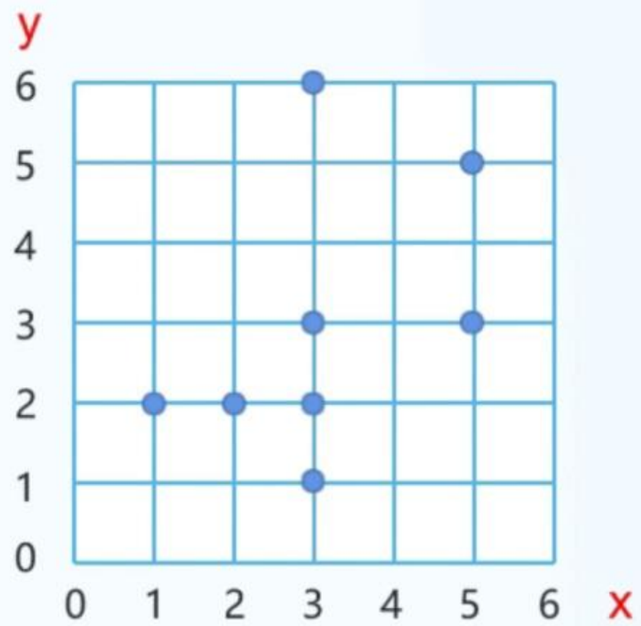
快码加鞭

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**				
2									

可添加的点数**为1时**



(1,2)为终点时，长度为2 +1个点

(2,2)为终点时，长度为3 +1个点 (1,2)-->(2,2)

(3,1)为终点时，长度为2 +1个点

(3,2)为终点时，长度为4 +1个点 (1,2)-->(2,2)-->(3,2) 选长的

+1个点 (3,1)-->(3,2)

(3,3)为终点时，长度为5 +1个点 (1,2)-->(2,2)-->(3,2)-->(3,3) 选长的

+1个点 (3,1)-->(3,2)-->(3,3)

快码加鞭

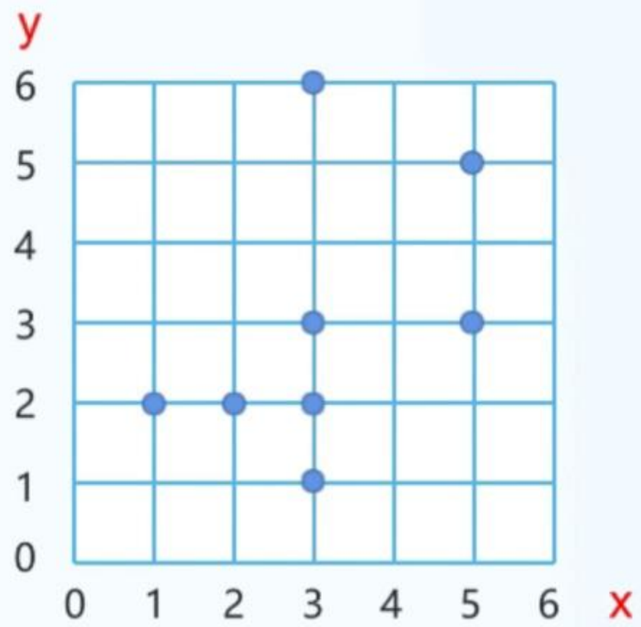
问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**			
2									

可添加的点数**为1时**

(3,6)为终点时，长度为2　　+1个点

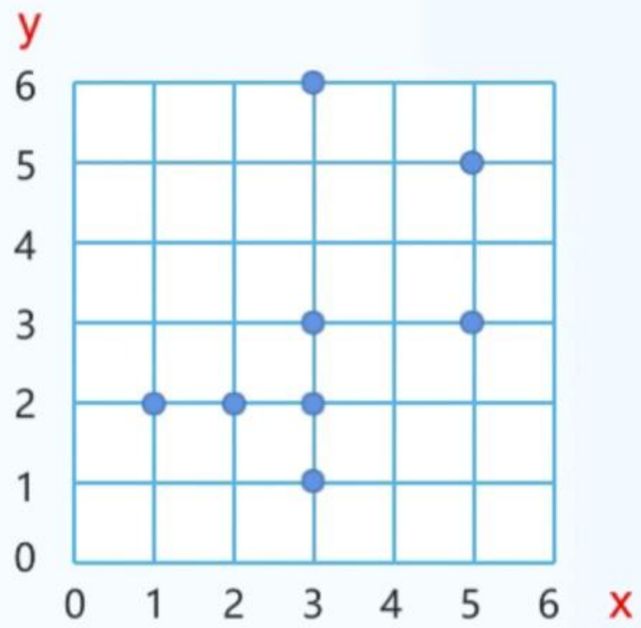


问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**		
2									

可添加的点数**为1时**



(3,6)为终点时，长度为2　+1个点
(5,3)为终点时，长度为6　这里比较特殊

(1,2)-->(2,2)-->(3,2)-->(3,3) +1个点 -->(5,3)

这个点作为桥梁

在求(5,3)最优解时，需要搜索所有可以添加“桥梁”的点，如（3,3），比较它们添加完“桥梁”后的长度，选取最长最为最优解。

等同于搜索（5,3）点前方，所有距离为2的点。

快码加鞭

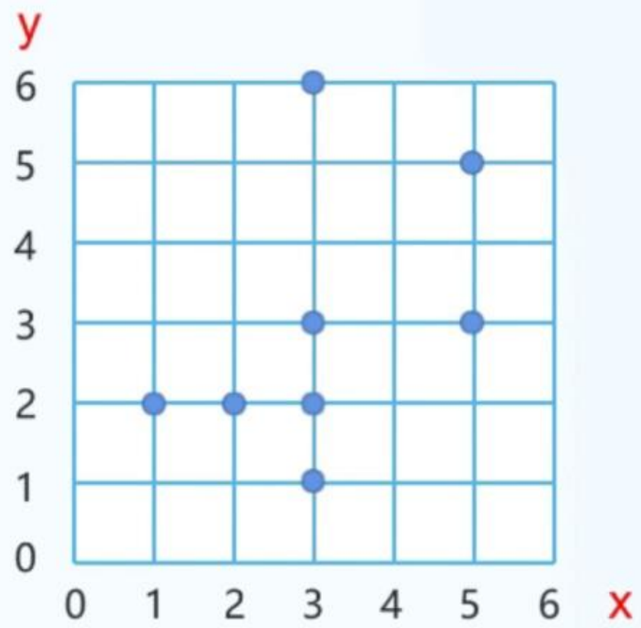
问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2									

可添加的点数**为1时**

(5,5)为终点时，长度为3　这里也比较特殊



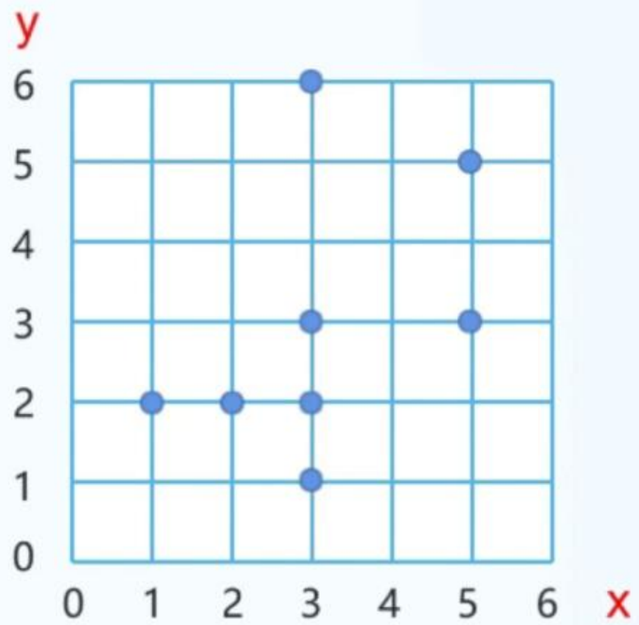
快码加鞭

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2									

可添加的点数**为1时**



(5,5)为终点时，长度为3　这里也比较特殊

(5,5)前方距离为2的点只有1个，就是（5,3）

(5,3) +1个点 -->(5,5)

1　　6

(5,3)两个解

6这个解已经把添加点数用完了，所以不能再用这个解去连接（5,5）点，因为可添加点数目前只有1个。

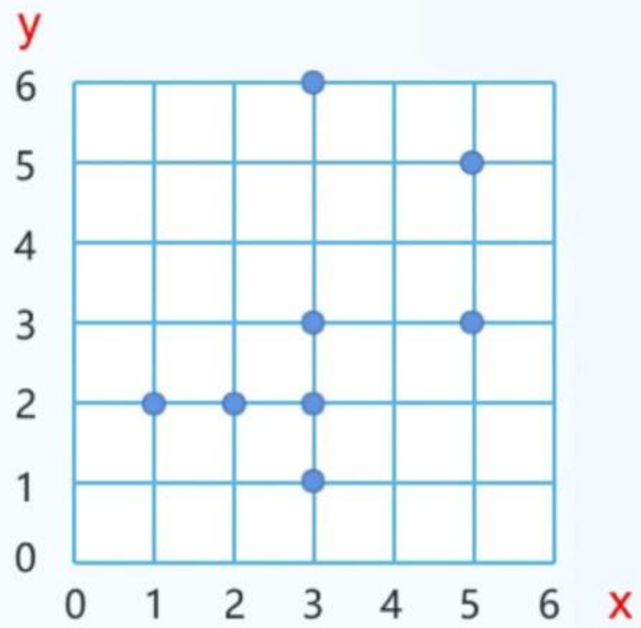
快码加鞭

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2									

可添加的点数**为1时**



(5,5)为终点时，长度为3　这里也比较特殊

(5,5)前方距离为2的点只有1个，就是（5,3）

(5,3) +1个点 -->(5,5)

1　　6

(5,3)两个解

6这个解已经把添加点数用完了，所以不能再用这个解去连接（5,5）点，因为可添加点数目前只有1个。

1 + 1 + 1 = 3

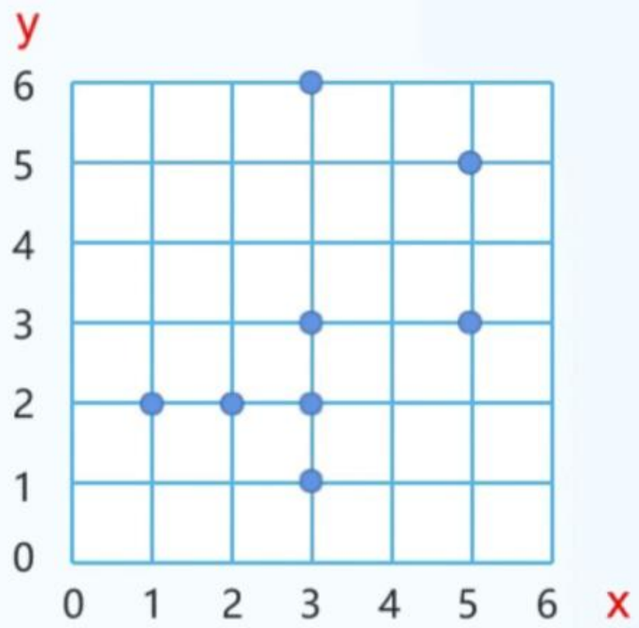
(5,3)解　加点　(5,5)点

快码加鞭

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2									



可添加的点数**为1时**　　总结可添加1个点时的规律

以（3,2）点为例：

可添加点数为1，搜索（3,2）点前方所有距离为2的点

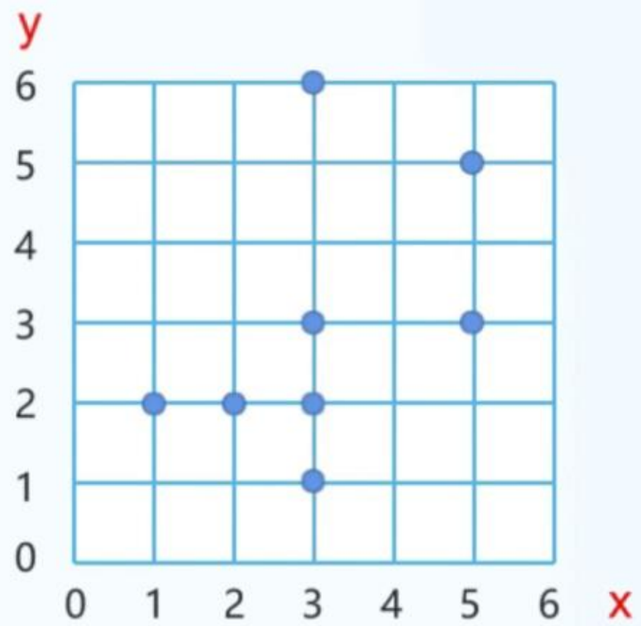
快码加鞭

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2									

可添加的点数**为1时**　　总结可添加1个点时的规律



以（3,2）点为例：

可添加点数为1，搜索（3,2）点前方所有距离为2的点

(1,2) +1个点 -->(3,2)　　**1 + 1 + 1 = 3**　不是最优解

距离为2　　(1,2)解　加点　(3,2)点

不考虑地图中(1,2)到(3,2)之间已经存在的点，只用可添加点

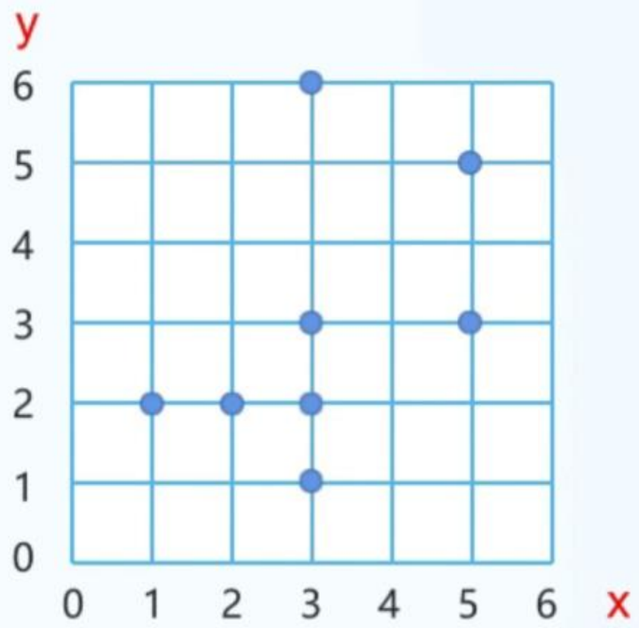
这里需要添加1个点，所以（1,2）点的解不能用已经添加完一个点的解

快码加鞭

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2									



可添加的点数**为1时**　　总结可添加1个点时的规律

以（3,2）点为例：

继续搜索（3,2）点前方所有距离为1的点

(2,2) +0个点 -->(3,2)

距离为1

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2									

可添加的点数**为1时**　　总结可添加1个点时的规律

以（3,2）点为例：

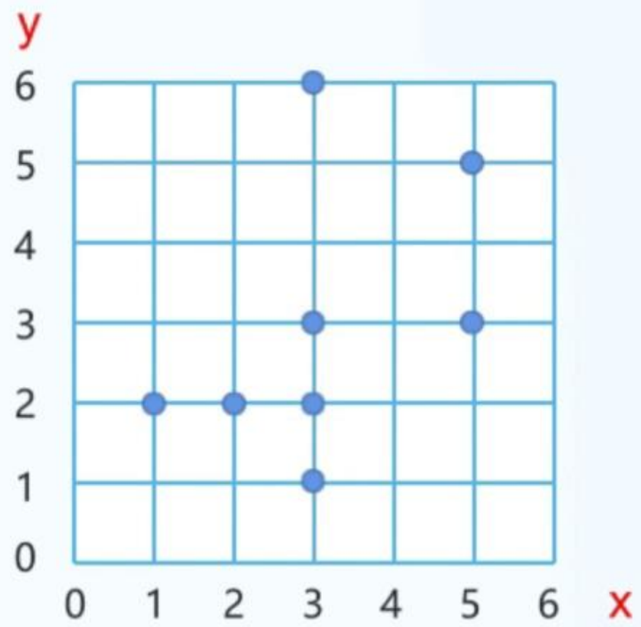
继续搜索（3,2）点前方所有距离为1的点

(2,2) +0个点 -->(3,2)　　**3** + **0** + **1** = **4**

距离为1　　使用1个添加点　　加点　　(3,2)点

(3,1) +0个点 -->(3,2)

距离为1

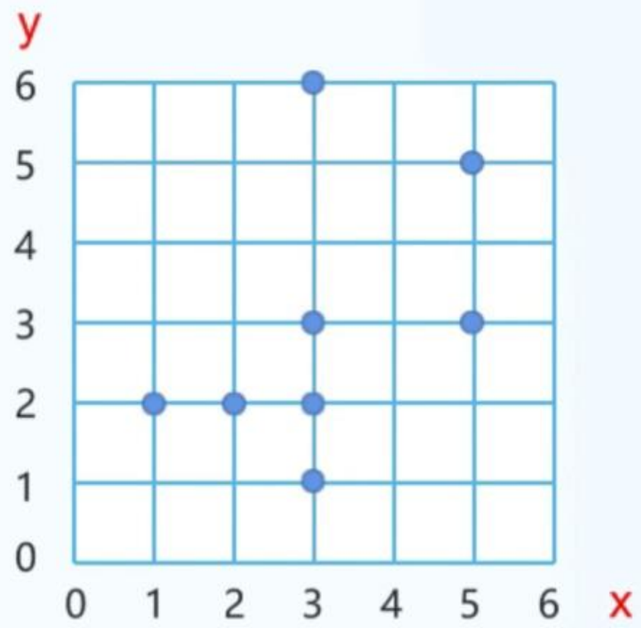


快码加鞭

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2									



可添加的点数**为1时**　　总结可添加1个点时的规律

以（3,2）点为例：

可用点-->自身　间隔的点数

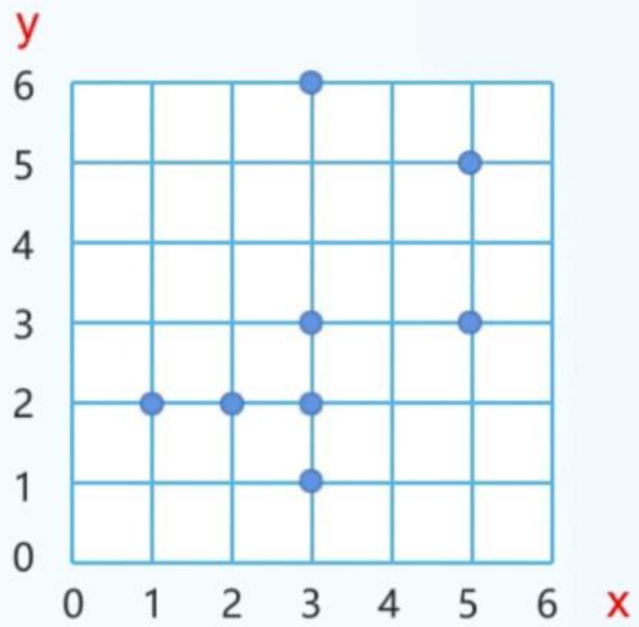
（3,2）最优解的转换方式　**前方可用解 + 间隔点数 + 自身**

快码加鞭

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2									



可添加的点数**为1时**　　总结可添加1个点时的规律

以（3,2）点为例：

可用点-->自身　间隔的点数

（3,2）最优解的转换方式　**前方可用解 + 间隔点数 + 自身**

距离为2的点 → **1 + 1 + 1 = 3**

距离为1的点 → **3 + 0 + 1 = 4**　选最大

距离为1的点 → **2 + 0 + 1 = 3**

当可添加的点数为1，遍历与当前点距离小于等于2的所有点，从这些点的最优解进行状态转移，挑选最大值。

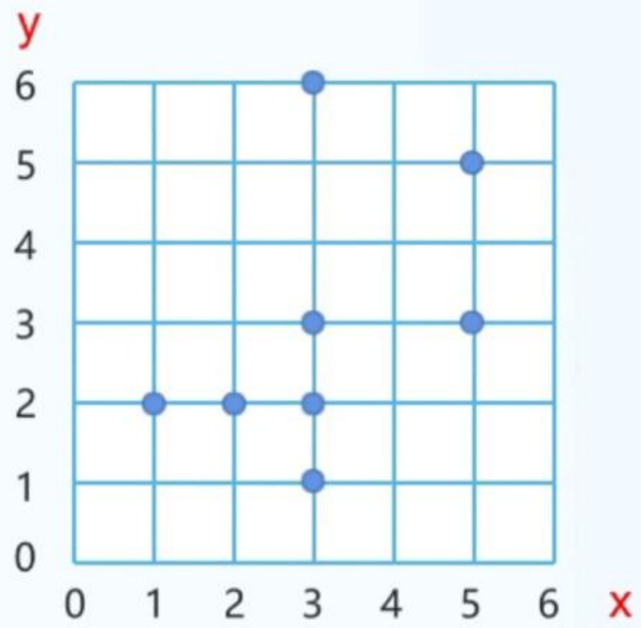
问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2									

可添加的点数**为2时**

验证结论，遍历长度小于等于3的所有点。



状态转移方程

max **前方可用解 + 间隔点数 + 自身**

可用解可能会有多个，需要多次比较，选最大

快码加鞭

问题分析

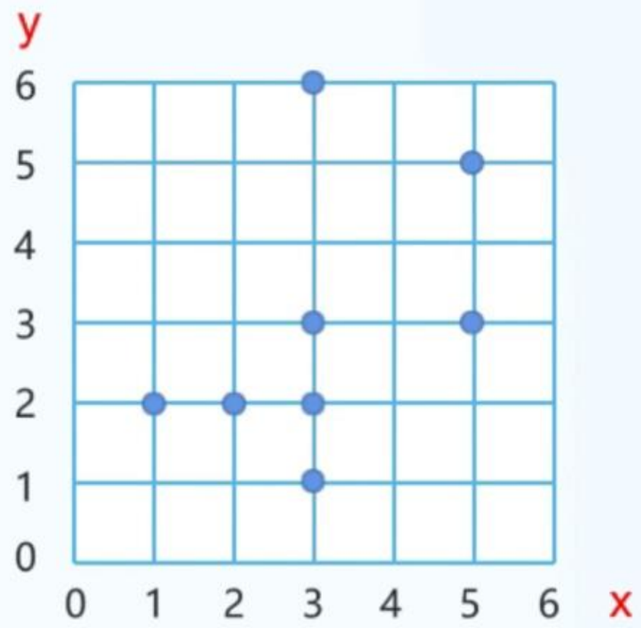
用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2	**3**								

可添加的点数**为2时**

验证结论，遍历长度小于等于3的所有点。

(1,2)为终点时，长度为3　　0 + 2 + 1 = 3



状态转移方程

max　**前方可用解 + 间隔点数 + 自身**

可用解可能会有多个，需要多次比较，选最大

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2	**3**	**4**							

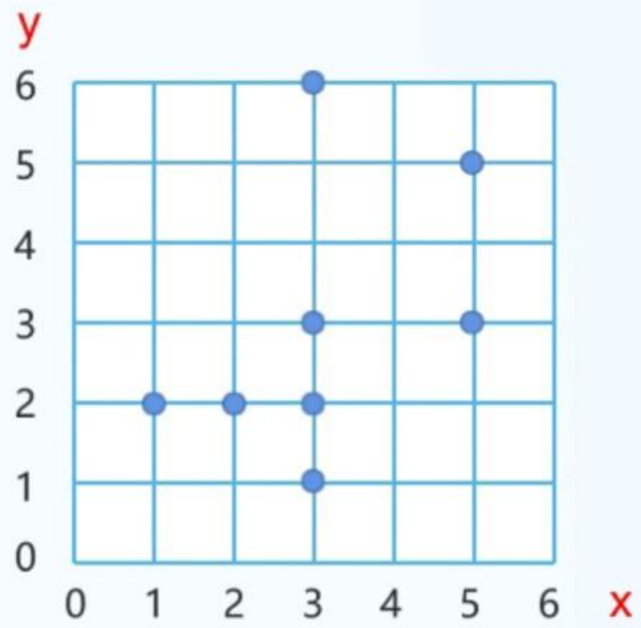
可添加的点数**为2时**

验证结论，遍历长度小于等于3的所有点。

(1,2)为终点时，长度为3　　0 + 2 + 1 = 3

(2,2)为终点时，长度为4　　3 + 0 + 1 = 4

(1,2) 与 (2,2) 中间无间隔点



状态转移方程

max **前方可用解 + 间隔点数 + 自身**

可用解可能会有多个，需要多次比较，选最大

快码加鞭

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2	**3**	**4**	**3**	5					

↑可添加的点数**为2时**

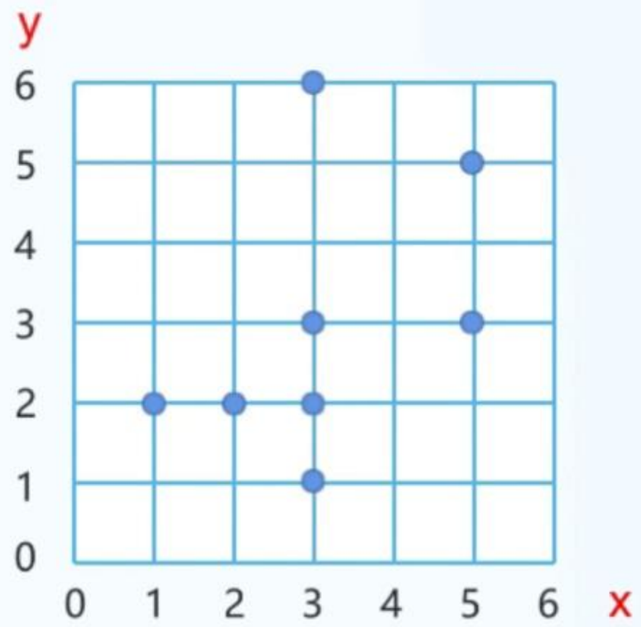
验证结论，遍历长度小于等于3的所有点。

(1,2)为终点时，长度为3　0 + 2 + 1 = 3

(2,2)为终点时，长度为4　3 + 0 + 1 = 4

(3,1)为终点时，长度为3　0 + 2 + 1 = 3

(3,2)为终点时，长度为5←　2 + 1 + 1 = 4
4 + 0 + 1 = 5
3 + 0 + 1 = 4



状态转移方程

max **前方可用解 + 间隔点数 + 自身**

可用解可能会有多个，需要多次比较，选最大

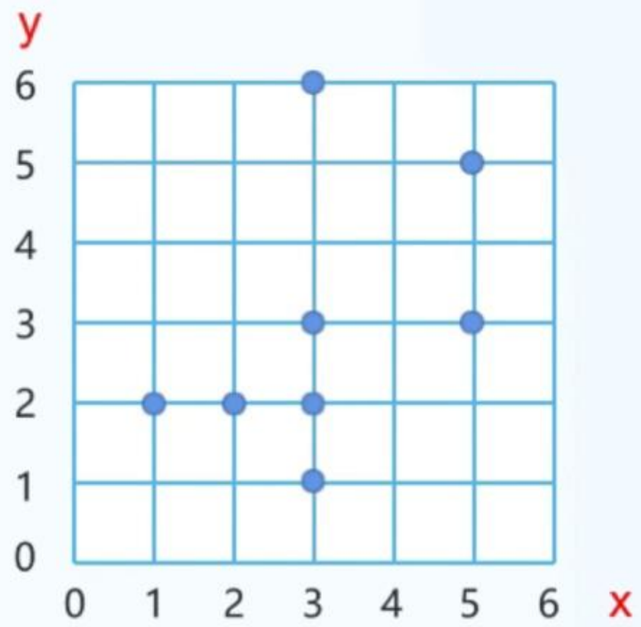
问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2	**3**	**4**	**3**	**5**	**6**				

↑可添加的点数**为2时**

验证结论，遍历长度小于等于3的所有点。



状态转移方程

max **前方可用解 + 间隔点数 + 自身**

↑可用解可能会有多个，需要多次比较，选最大

快码加鞭

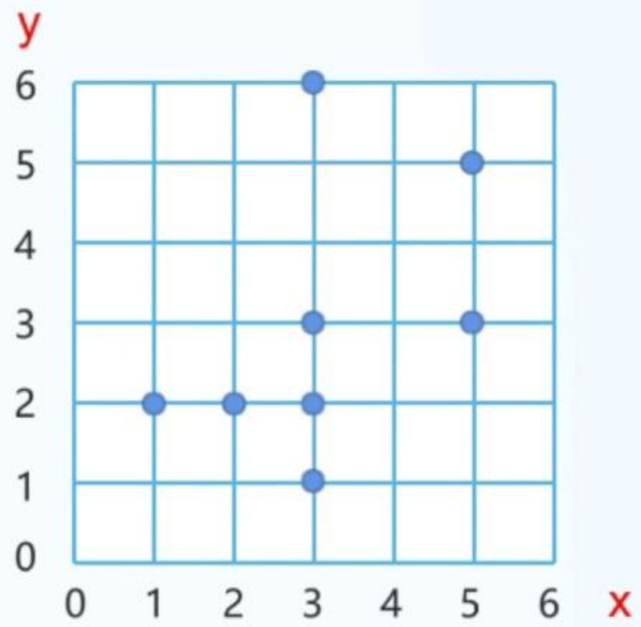
问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2	**3**	**4**	**3**	**5**	**6**				

可添加的点数**为2时**

验证结论，遍历长度小于等于3的所有点。



状态转移方程

max **前方可用解 + 间隔点数 + 自身**

可用解可能会有多个，需要多次比较，选最大

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8
序号	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)
0	1	2	1	3	4	1	1	1
1	2	3	2	4	5	2	6	3
2	3	4	3	5	6	7	7	

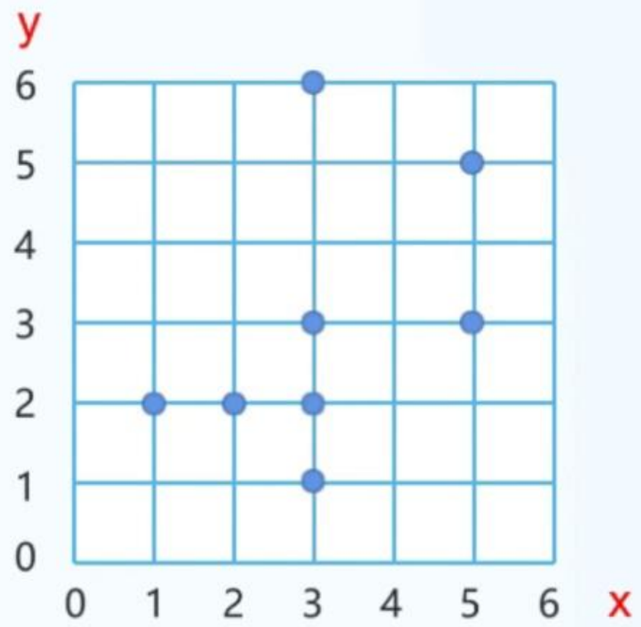
可添加的点数**为2时**

验证结论，遍历长度小于等于3的所有点。

(3,6)为终点时，长度为7　　4 + 2 + 1 = 7

(5,3)为终点时，长度为7　　5 + 1 + 1 = 7

3 + 2 + 1 = 6



状态转移方程

max　**前方可用解 + 间隔点数 + 自身**

可用解可能会有多个，需要多次比较，选最大

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2	**3**	**4**	**3**	**5**	**6**	**7**	**7**	**8**	

可添加的点数**为2时**

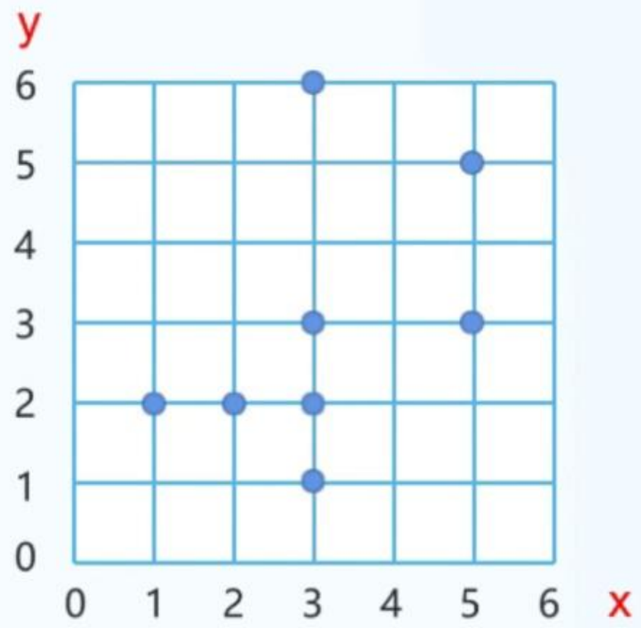
验证结论，遍历长度小于等于3的所有点。

(3,6)为终点时，长度为7　　4 + 2 + 1 = 7

(5,3)为终点时，长度为7　　5 + 1 + 1 = 7

3 + 2 + 1 = 6

(5,5)为终点时，长度为8　　6 + 1 + 1 = 8



状态转移方程

max　**前方可用解 + 间隔点数 + 自身**

可用解可能会有多个，需要多次比较，选最大

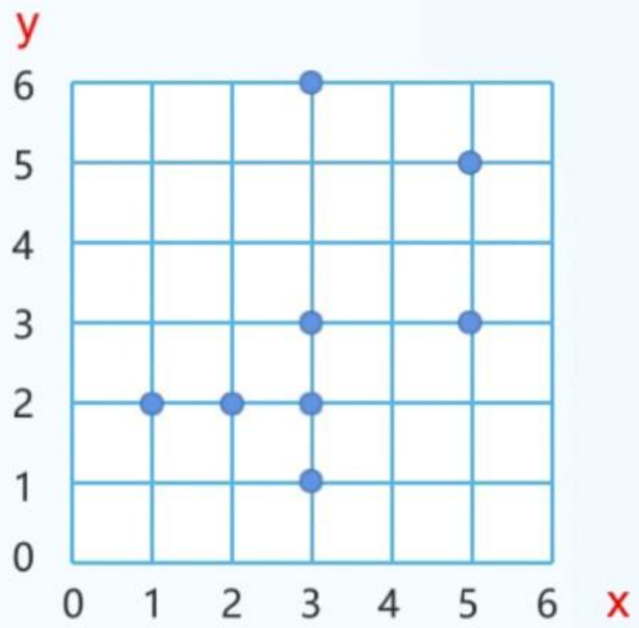
快码加鞭

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	1	2	1	3	4	1	1	1	
1	2	3	2	4	5	2	6	3	
2	3	4	3	5	6	7	7	8	

可添加的点数



寻找可用解的关键点：**判断　定位**

例如：（3,3）可添加两个点的最优解

设dp数组存储表格中的状态

可用解　间隔点数　自身

dp[2][5] = dp[0][1] + 2 + 1
dp[1][2] + 1 + 1
dp[1][3] + 1 + 1
dp[2][4] + 0 + 1

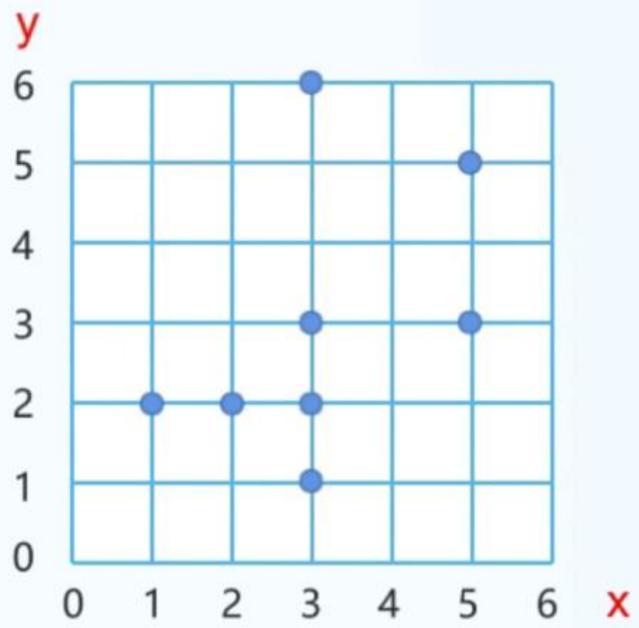
快码加鞭

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2	**3**	**4**	**3**	**5**	**6**	**7**	**7**	**8**	

↑可添加的点数



寻找可用解的关键点：**判断　定位**

例如：（3,3）可添加两个点的最优解

设dp数组存储表格中的状态

可用解　间隔点数　自身

dp[2][5] = dp[0][1] + 2 + 1
dp[1][2] + 1 + 1
dp[1][3] + 1 + 1
dp[2][4] + 0 + 1

快码加鞭

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2	**3**	**4**	**3**	**5**	**6**	**7**	**7**	**8**	

↑可添加的点数

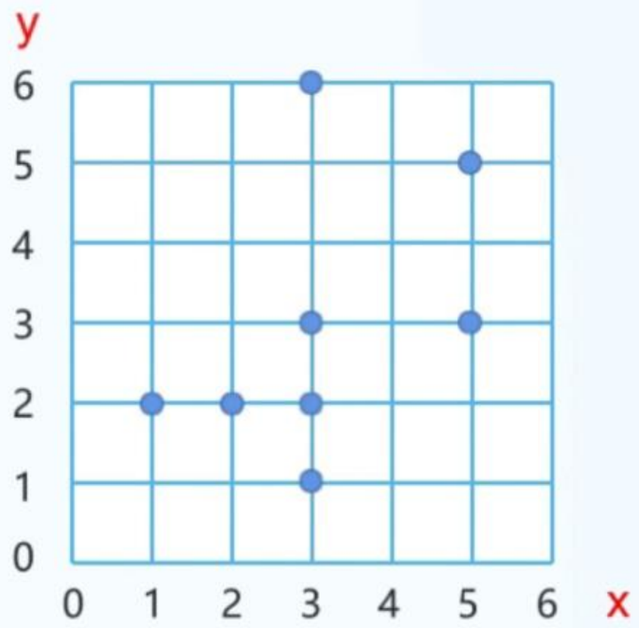
寻找可用解的关键点：**判断　定位**

例如：（3,3）可添加两个点的最优解

设dp数组存储表格中的状态

可用解　间隔点数　自身

dp[2][5] = dp[0][1] + 2 + 1
dp[1][2] + 1 + 1
dp[1][3] + 1 + 1
dp[2][4] + 0 + 1



①**判断**　间隔点数 ≤ 可添加点数　证明点可连接

(1,2) (2,2) (3,1) (3,2) 与 **(3,3)** 间隔点不超过2

②**定位**　找到可用解所在行

在代码中详细讲

解题步骤

1. 输入数据完成结点排序

```
struct node {
    //存储点的坐标
    int x, y;
};
//n已有点数量 k可添加点数量
int n, k;
node a[510]; //存点
```

在全局创建

```
//设置排序规则
bool cmp(node a, node b) {
    if (a.x==b.x) {
        return a.y<b.y;
    }
    return a.x<b.x;
}
```

排序规则：
按照x值由小到大，x值相等按照y值由小到大。

```
cin>>n>>k;
for (int i=1; i<=n; i++) {
    cin>>a[i].x>>a[i].y;
}
//a数组中所有点进行排序
sort(a+1, a+n+1, cmp);
```

解题步骤

1. 输入数据完成结点排序

```
struct node {
    //存储点的坐标
    int x, y;
};
//n已有点数量 k可添加点数量
int n, k;
node a[510]; //存点
```

在全局创建

```
//设置排序规则
bool cmp(node a, node b) {
    if (a.x==b.x) {
        return a.y<b.y;
    }
    return a.x<b.x;
}
```

排序规则：
按照x值由小到大，x值相等按照y值由小到大。

```
cin>>n>>k;
for (int i=1; i<=n; i++) {
    cin>>a[i].x>>a[i].y;
}
//a数组中所有点进行排序
sort(a+1, a+n+1, cmp);
```

解题步骤

2. 遍历结点完成状态转移

```
int dp[510][510]; //全局数组存状态
//可增加的结点数为k
for (int i=0; i<=k; i++) {
    //已有结点有n个
    for (int j=1; j<=n; j++) {
        //初始化，自身结点加可添加结点
        dp[i][j]=1+i;
    }
}
```

	1	2	3	4	5	6	7	8	序号j
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0									
1									
2									

可添加的点数i

解题步骤

2. 遍历结点完成状态转移

```
int dp[510][510]; //全局数组存状态
//可增加的结点数为k
for (int i=0; i<=k; i++) {
    //已有结点有n个
    for (int j=1; j<=n; j++) {
        //初始化，自身结点加可添加结点
        dp[i][j]=1+i;
    }
}
```

	1	2	3	4	5	6	7	8	序号j
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0									
1									
2									

↑可添加的点数i

解题步骤

2. 遍历结点完成状态转移

```
int dp[510][510]; //全局数组存状态
//可增加的结点数为k
for (int i=0; i<=k; i++) {
    //已有结点有n个
    for (int j=1; j<=n; j++) {
        //初始化，自身结点加可添加结点
        dp[i][j]=1+i;
        //从l到j里选择合适的最优解进行状态转移
        for (int l=1; l<j; l++) {



        }
    }
}
```

	1	2	3	4	5	6	7	8	序号j
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	1	2	1	3	?				
1									
2									

可添加的点数i

求j点最优解时，遍历j点前方可用的最优解，进行状态转移。

解题步骤

2. 遍历结点完成状态转移

```
int dp[510][510]; //全局数组存状态
//可增加的结点数为k
for (int i=0; i<=k; i++) {
    //已有结点有n个
    for (int j=1; j<=n; j++) {
        //初始化，自身结点加可添加结点
        dp[i][j]=1+i;
        //从l到j里选择合适的最优解进行状态转移
        for (int l=1; l<j; l++) {

            判断 l 点是否与 j 点形成递增关系


        }
    }
}
```

	1	2	3	4	5	6	7	8	序号j
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	1	2	1	3	?				
1									
2									

可添加的点数i

求j点最优解时，遍历j点前方可用的最优解，进行状态转移。

解题步骤

2. 遍历结点完成状态转移

```
int dp[510][510]; //全局数组存状态
//可增加的结点数为k
for (int i=0; i<=k; i++) {
    //已有结点有n个
    for (int j=1; j<=n; j++) {
        //初始化，自身结点加可添加结点
        dp[i][j]=1+i;
        //从l到j里选择合适的最优解进行状态转移
        for (int l=1; l<j; l++) {

            判断 l 点是否与 j 点形成递增关系


        }
    }
}
```

	1	2	3	4	5	6	7	8	序号j
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	1	2	1	3	?				
1									
2									

可添加的点数i

求j点最优解时，遍历j点前方可用的最优解，进行状态转移。

解题步骤

2. 遍历结点完成状态转移

```
int dp[510][510]; //全局数组存状态
//可增加的结点数为k
for (int i=0; i<=k; i++) {
    //已有结点有n个
    for (int j=1; j<=n; j++) {
        //初始化，自身结点加可添加结点
        dp[i][j]=1+i;
        //从l到j里选择合适的最优解进行状态转移
        for (int l=1; l<j; l++) {
            //判断两点是否是递增关系
            if (a[l].x <= a[j].x && a[l].y <= a[j].y) {

            }
        }
    }
}
```

	1	2	3	4	5	6	7	8	序号j
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	1	2	1	3	?				
1									
2									

↑可添加的点数i

解题步骤

2. 遍历结点完成状态转移

```
int dp[510][510]; //全局数组存状态
//可增加的结点数为k
for (int i=0; i<=k; i++) {
    //已有结点有n个
    for (int j=1; j<=n; j++) {
        //初始化，自身结点加可添加结点
        dp[i][j]=1+i;
        //从l到j里选择合适的最优解进行状态转移
        for (int l=1; l<j; l++) {
            //判断两点是否是递增关系
            if (a[l].x <= a[j].x && a[l].y <= a[j].y) {
                计算 l 点到 j 点的间隔点数
            }
        }
    }
}
```

	1	2	3	4	5	6	7	8	序号j
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	1	2	1	3	?				
1									
2									

↑ 可添加的点数i

解题步骤

计算点与点之间的间隔点数

如：(1,2)-->(5,3)

这两个点 x轴 与 y轴 的直线距离加在一起为5

5-1=4　　3-2=1

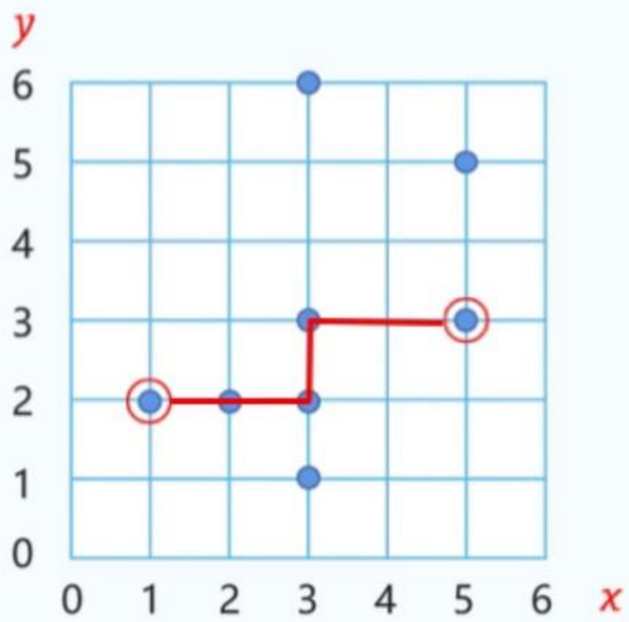
4+1=5

结点数与距离刚好相差1

距离为5，间隔结点数为 5-1=4

(1,2)-->(5,3) 间隔结点数为：

$$\underbrace{5-1}_{x\text{值}}+\underbrace{3-2}_{y\text{值}}-1$$



解题步骤

2. 遍历结点完成状态转移

```
int dp[510][510]; //全局数组存状态
//可增加的结点数为k
for (int i=0; i<=k; i++) {
    //已有结点有n个
    for (int j=1; j<=n; j++) {
        //初始化，自身结点加可添加结点
        dp[i][j]=1+i;
        //从l到j里选择合适的最优解进行状态转移
        for (int l=1; l<j; l++) {
            //判断两点是否是递增关系
            if (a[l].x <= a[j].x && a[l].y <= a[j].y) {
                //连接l~j实际需要添加的点数
                int t = a[j].x - a[l].x + a[j].y - a[l].y - 1;
            }
        }
    }
}
```

	1	2	3	4	5	6	7	8	序号j
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	1	2	1	3	?				
1									
2									

↑可添加的点数i

解题步骤

2. 遍历结点完成状态转移

```
int dp[510][510]; //全局数组存状态
//可增加的结点数为k
for (int i=0; i<=k; i++) {
    //已有结点有n个
    for (int j=1; j<=n; j++) {
        //初始化，自身结点加可添加结点
        dp[i][j]=1+i;
        //从l到j里选择合适的最优解进行状态转移
        for (int l=1; l<j; l++) {
            //判断两点是否是递增关系
            if (a[l].x <= a[j].x && a[l].y <= a[j].y) {
                //连接l~j实际需要添加的点数
                int t = a[j].x - a[l].x + a[j].y - a[l].y - 1;
                if (t <= i) {
                    dp[i][j] = max(dp[i][j], dp[    ][l] + t + 1);
                }
            } ...
```

	1	2	3	4	5	6	7	8
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)
0	1	2	1	3	?			
1								
2								

序号j

可添加的点数i

间隔点数≤可添加点数
可做状态转移

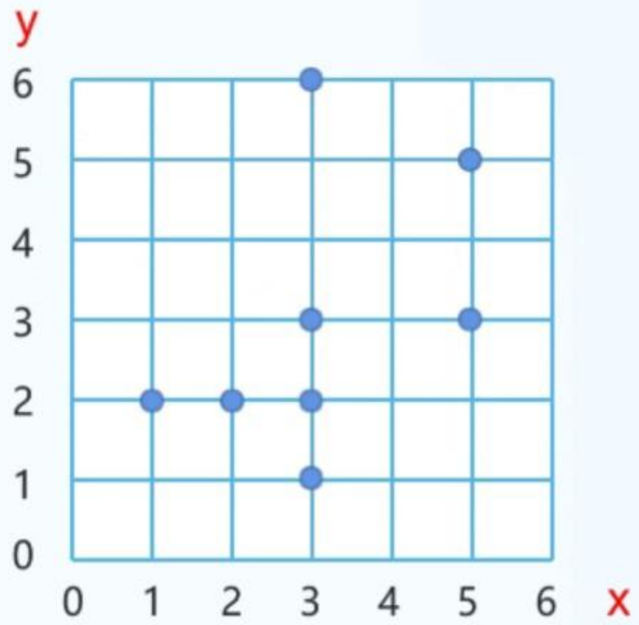
可用解

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2	**3**	**4**	**3**	**5**	**6**	**7**	**7**	**8**	
	可添加的点数								

用（3,3）最优解举例



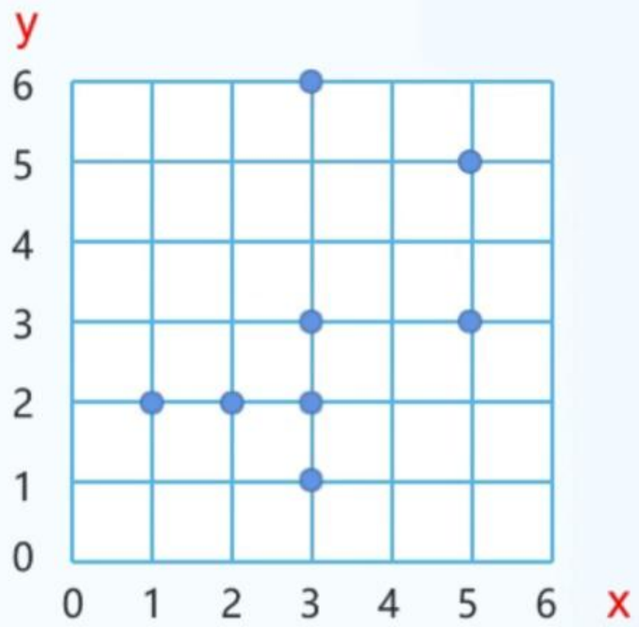
快码加鞭

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	**1**	**2**	**1**	**3**	**4**	**1**	**1**	**1**	
1	**2**	**3**	**2**	**4**	**5**	**2**	**6**	**3**	
2	**3**	**4**	**3**	**5**	**6**	**7**	**7**	**8**	

↑可添加的点数



用（3,3）最优解举例

间隔点数

dp[2][5] = dp[0][1] + 2 + 1
dp[1][2] + 1 + 1
dp[1][3] + 1 + 1
dp[2][4] + 0 + 1

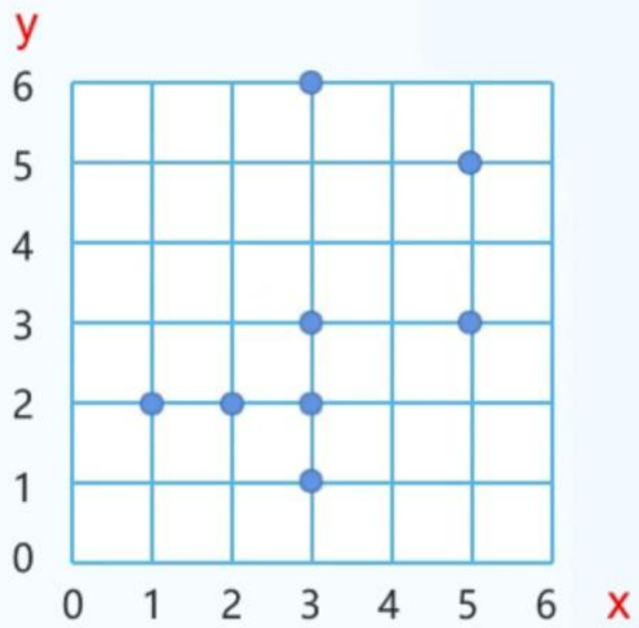
快码加鞭

问题分析

用表格统计所有状态

	1	2	3	4	5	6	7	8	序号
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	1	2	1	3	4	1	1	1	
1	2	3	2	4	5	2	6	3	
2	3	4	3	5	6	7	7	8	

可添加的点数



用（3,3）最优解举例

间隔点数

dp[2][5] =
- dp[0][1] + 2 + 1
- dp[1][2] + 1 + 1
- dp[1][3] + 1 + 1
- dp[2][4] + 0 + 1

可添加点数为2　间隔点数

- dp[2 - 2][1] + 2 + 1
- dp[2 - 1][2] + 1 + 1
- dp[2 - 1][3] + 1 + 1
- dp[2 - 0][4] + 0 + 1

可添加点数减去间隔点数得到可用解的行号

解题步骤

2. 遍历结点完成状态转移

```
int dp[510][510]; //全局数组存状态
//可增加的结点数为k
for (int i=0; i<=k; i++) {
    //已有结点有n个
    for (int j=1; j<=n; j++) {
        //初始化，自身结点加可添加结点
        dp[i][j]=1+i;
        //从l到j里选择合适的最优解进行状态转移
        for (int l=1; l<j; l++) {
            //判断两点是否是递增关系
            if (a[l].x <= a[j].x && a[l].y <= a[j].y) {
                //连接l~j实际需要添加的点数
                int t = a[j].x - a[l].x + a[j].y - a[l].y - 1;
                if (t <= i) {
                    dp[i][j] = max(dp[i][j], dp[i - t][l] + t + 1);
                }
            } ...
```

	1	2	3	4	5	6	7	8
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)
0	1	2	1	3	?			
1								
2								

序号j

可添加的点数i

可添加点数 - 间隔点数

解题步骤

3. 存储最大值

```
int dp[510][510]; //全局数组存状态
int ans; //全局变量存最优解
//可增加的结点数为k
for (int i=0; i<=k; i++) {
    //已有结点有n个
    for (int j=1; j<=n; j++) {
        ...
    }
    //在dp数组中选取最大值
    ans = max(ans, dp[i][j]);
}
```

	1	2	3	4	5	6	7	8	序号j
	(1,2)	(2,2)	(3,1)	(3,2)	(3,3)	(3,6)	(5,3)	(5,5)	
0	1	2	1	3	?				
1									
2									

↑可添加的点数i

4. 输出最大值

```
cout<<ans;
```

完整代码

```
#include<bits/stdc++.h>
using namespace std;
struct node {
    int x, y;
};
bool cmp(node a, node b) {
    if (a.x == b.x) {
        return a.y < b.y;
    }
    return a.x < b.x;
}
//n已有点数量 k可添加点数量
int n, k;
node a[510];
int dp[510][510];
int ans;
int main() {
    cin >> n >> k;
    for (int i = 1; i <= n; i++) {
        cin >> a[i].x >> a[i].y;
    }
    sort(a + 1, a + n + 1, cmp);
```