动态规划入门

在现实生活中，有些事物的发展过程可以分为若干个互相联系的阶段，在每一个阶段都要做出决策，一个阶段的决策确定以后，常常会影响到下一个阶段的决策，这样一个任务则可以称为**多阶段决策问题**. 其中，各个阶段的决策构成一个决策序列，称为一个**策略**. 动态规划就是求解决策过程优化问题的一种思想，用一句话说：**每个阶段的最优状态可以从之前某个阶段的某个或某些状态直接得到，而不管之前这个状态是如何得到**.

动态规划的本质是**分治思想和解决冗余**. 所谓分治是将实际问题分解为更小的、相似的子问题；而解决冗余是动态规划算法的根本目的. 动态规划实质是一种**以空间换时间**的技术，它在实现的过程，不得不存储产生过程中的各个状态，目的是在求解重叠子问题不必做重复的计算，故它的空间复杂度一般要大于其他的算法，但这是可以接受的.

动态规划算法解决的问题有 3 个特点：

1. **最优子结构性质**：问题的最优解所包含的子问题也是最优的，从局部解逐渐得到最优解；
2. **无后效性**：即子问题的解一旦确定，就不再改变，不受之后、包含更大的问题的求解决策影响；
3. **子问题重叠性质**：每次产生的子问题并不总是新问题，有些子问题会被重复计算多次，所以对于每个子问题只需要计算一次，将结果保存在一个表格中（程序中就是数组），当再次需要计算已经计算过的子问题，只是在表格中简单查看结果即可，从而获得较高的效率.

动态规划(Dynamic Programming)：利用问题的最优子结构性质，将待求解问题分解成若干子问题求解，从这些子问题的解得到原问题的解. 在此过程中，需记录已经解决的子问题的解，以避免重复计算.

动态规划的**大体步骤分为两步**：

第一步(**化 0 为整**的过程)

解释：每次在处理这个问题的时候，不是一个方案一个方案地去枚举，而是每次枚举一堆东西，去枚举一个子集一类东西，这个可以看成是一个化 0 为整的过程，把一些有相似点的元素划分成一个子集，然后用某种状态表示.

状态表示 $f[i]$　　可再细分为**集合**和**属性**.

集合：　$f[i]$ 一般描述的方式是所有**满足 xxx 条件的方案**的集合

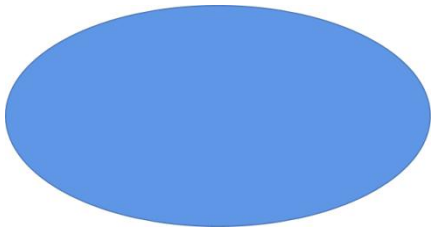
正是 $f[i]$ 每次可以表示一类东西，而不是一个东西，它可以优化时间复杂度(**优化的核心**).

属性：　$f[i]$ 存的数与集合的关系，如 max（最大值）、min（最小值）、*count*（数量）等，题目问什么，属性一般是什么.

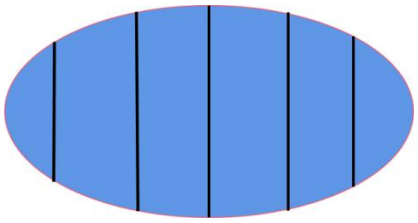
第二步(**化整为 0** 的过程——看每一个状态怎么计算（**状态转移方程**）)：

状态计算

1. 看 $f[i]$ 表示的大集合(用韦恩图的椭圆表示一个集合)：



2. 将它划分成若干个子集(部分)，然后分别去求解每一部分(划分依据：找最后一个不同点)



划分的子集一般需满足以下两个原则：
①**不重复**：每个元素只属于其中的一个集合；

②**不遗漏**：所有子集里的元素一定是把 $f[i]$ 中所有元素都包含了，不能漏掉某个元素.

注：并不一定所有的情况下都需要满足这两个原则. 不遗漏原则是所有情况下都需要满足，但不重复原则不一定，当我们求的是数量时必须保证不重复，若当求最值时，其实是可以重复的，但一般也不重复.

3. 在求 $f[i]$ 的时候，把整体问题划分成了某些子集的问题，然后把每一个子集计算出来，整理出来计算出 $f[i]$.

① $f[i]$ 的属性是最大值，$整个集合的最大值=\max(每个子集的最大值)$，最小值同理；

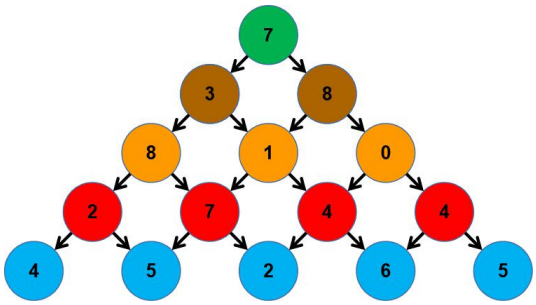
② $f[i]$ 的属性是数量，$总数量=每个子集的数量和$

后面以几个**经典例题**认识动态规划.

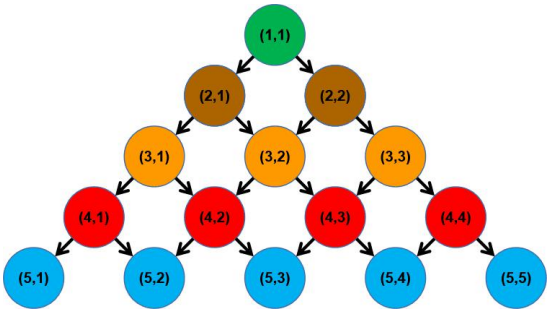
1. 数字三角形	2. 滑雪	3. 最大子段和	4. 最长公共子序列
5. 最长上升子序列	6. 石子合并(不讲)	7. 最长公共上升子序列(不讲)	8. 矩阵连乘(不讲)

例一 数字三角形

给定一个数字三角形，从顶部出发，在每一结点可以选择移动至其左下方的结点或移动至其右下方的结点，一直走到底层，要求找出一条路径，使路径上的数字的和最大。



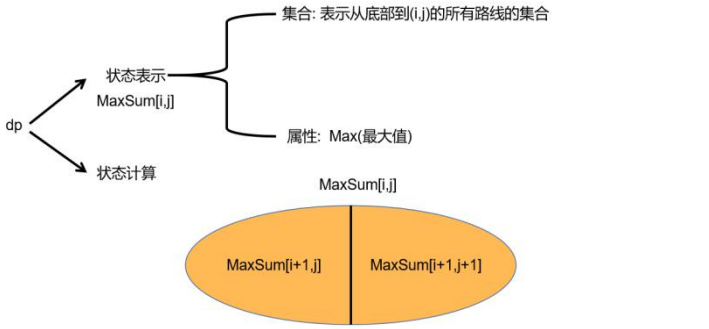
数字三角形



格子编号

用二维数组 d 存放数字三角形的每个位置的值，即 $d[i,j]$ 表示第 i 行第 j 列的数字（i,j 均从 1 开始算）；

用 $MaxSum[i][j]$ 表示从底部到 (i,j) 的各条路径中，数字之和的最大值.



$$MaxSum[i,j]=\max(MaxSum[i+1][j]+d[i][j],MaxSum[i+1][j+1]+d[i][j])$$

①暴力递归（回溯）

从 $d[i][j]$ 出发，下一步只能走 $d[i+1][j]$ 或 $d[i+1][j+1]$，故可以得到

$$\begin{cases} MaxSum[i][j]=d[i][j], 当 i=n; \\ MaxSum[i][j]=\max(MaxSum[i+1][j],MaxSum[i+1][j+1])+d[i][j], 当 1<i<n. \end{cases}$$

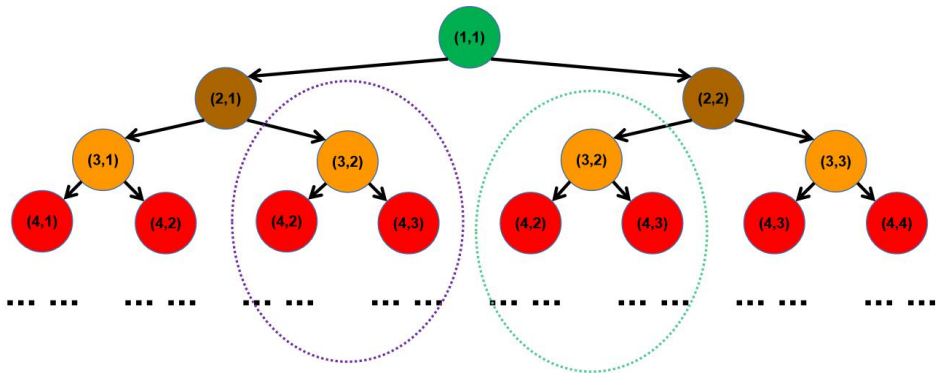
一个 n 层的数字三角形的完整路线有 2^{n-1} 且中间存在**重复计算**，时间复杂度为 $O(2^n)$（指数级别），当 n 很大时，递归回溯法会超时.

[代码链接]

```
int d[MAXN][MAXN],MaxSum[MAXN][MAXN];

int dpMaxSum(int i,int j){
    if(i==n)
        MaxSum[i][j]=d[i][j];
    else{
        int x=dpMaxSum(i+1,j);
        int y=dpMaxSum(i+1,j+1);
        MaxSum[i][j]=max(x,y)+d[i][j];
    }
    return MaxSum[i][j];
}
```

②记忆化搜索

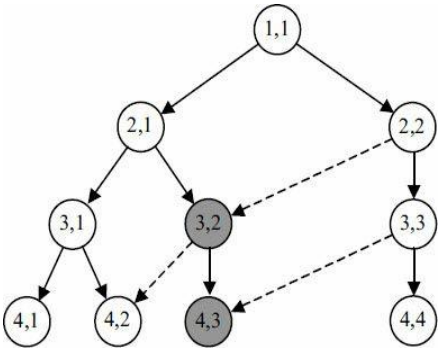
为了方便（第五层就不画了），函数 $dpMaxSum(3,2)$ 被计算了两次（一次是 $dpMaxSum(2,1)$ 需要的，一次是 $dpMaxSum(2,2)$ 需要的）. 这样的重复不是单个结点，而是一棵子树。调用的层数为 n 层，就需要访问 2^n-1 个结点（二叉树）。



记忆化搜索程序可分成两部分，首先将 $MaxSum$ 数组**全部初始化**为 -1，然后编写递归函数。

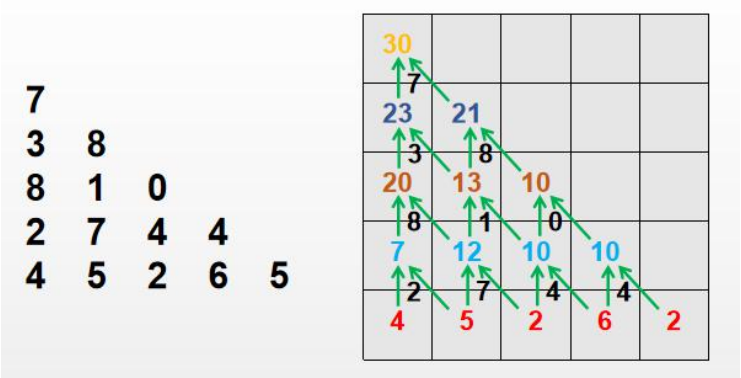
```
int d[MAXN][MAXN],MaxSum[MAXN][MAXN];
int dpMaxSum(int i,int j){
    /*****新增内容*****/
    if(MaxSum[i][j]!=-1)//在主函数中将MaxSum 数组初始化为-1 或者负无穷
        return MaxSum[i][j];
    //为了防止重复计算带来的时间超限，
    //对于已经求得最大和的该点，直接使用结果，不再进行重复计算
    /*****新增内容*****/
    if(i==n)
        MaxSum[i][j]=d[i][j];
    else{
        int x=dpMaxSum(i+1,j);
        int y=dpMaxSum(i+1,j+1);
```

```
        MaxSum[i][j]=max(x,y)+d[i][j];
    }
    return MaxSum[i][j];
}
```



上述程序的方法称为记忆化（memorization），由于 i,j 都在 $1\sim n$ 之间，所有不相同的结点一共只用 n^2 个. 无论以怎样的顺序访问，时间复杂度均为 $O(n^2)$. 时间从 2^n 到 n^2 是一个巨大的优化，正是利用了数字三角形具有大量重叠子问题的特点.

③二维迭代（从递归到递推）



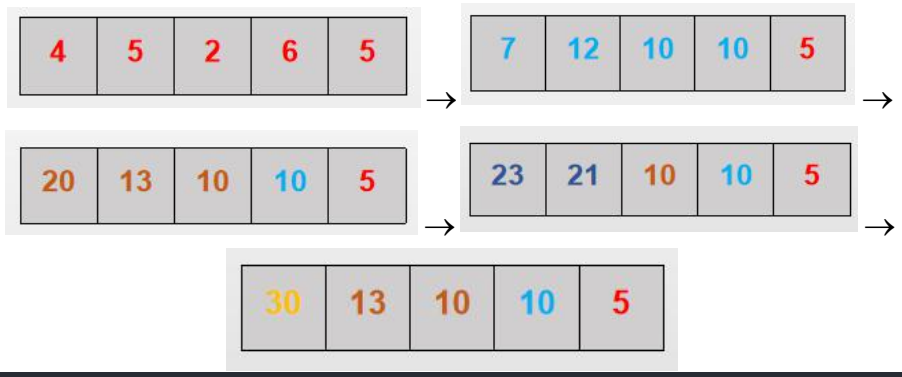
递归函数的调用也会有一些时间和空间上的开销，实际上不用递归也能解决这个问题，而且能够解决更快，更节省空间.

结果为 $MaxSum[1][1]$.

```
int d[MAXN][MAXN],MaxSum[MAXN][MAXN];

int dpMaxSum(){
    for(int i=1;i<=n;i++)
        MaxSum[n][i]=d[n][i];
    for(int i=n-1;i>=1;i--)
        for(int j=1;j<=i;j++)
            MaxSum[i][j]=max(MaxSum[i+1][j],MaxSum[i+1][j+1])+d[i][j];
    return MaxSum[1][1];
}
```

④一维迭代(优化空间)

二维迭代所使用的二维数组,还可以使用一维进行优化,即利用**滚动数组(一维数组)**进行更新, $MaxSum[1]$即为答案.




```
int d[MAXN][MAXN],MaxSum[MAXN];

int dpMaxSum(){
    for(int i=1;i<=n;i++)
        MaxSum[i]=d[n][i];
    for(int i=n-1;i>=1;i--)
        for(int j=1;j<=i;j++)
            MaxSum[j]=max(MaxSum[j],MaxSum[j+1])+d[i][j];
    return MaxSum[1];
}
```

例二 滑雪

给定一个 R 行 C 列的矩阵，表示一个矩形网格滑雪场。

矩阵中第 i 行第 j 列的点表示滑雪场的第 i 行第 j 列区域的高度。

一个人从滑雪场中的某个区域内出发，每次可以向上下左右任意一个方向滑动一个单位距离。

当然，一个人能够滑动到某相邻区域的前提是该区域的高度低于自己目前所在区域的高度。

下面给出一个矩阵作为例子：

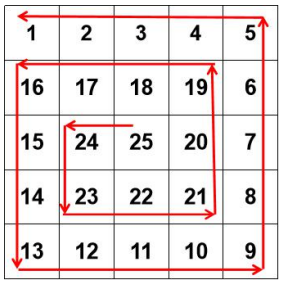
```
 1  2  3  4 5
16 17 18 19 6
15 24 25 20 7
14 23 22 21 8
13 12 11 10 9
```

在给定矩阵中，一条可行的滑行轨迹为 $24-17-2-1$。

在给定矩阵中，最长的滑行轨迹为 $25-24-23-\ldots-3-2-1$，沿途共经过 25 个区域。

现在给定你一个二维矩阵表示滑雪场各区域的高度，请你找出在该滑雪场中能够完成的最长滑雪轨迹，并输出其长度(可经过最大区域数)。

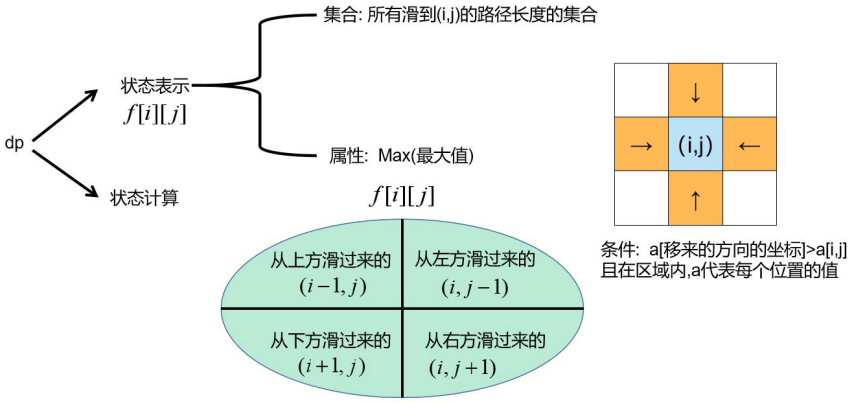
①搜索　　代码链接



对区域内每一个点进行 dfs，对每以个点进行四个方向的扩展，满足不超过边界且扩展的方向的元素比当前元素小，就对扩展方向的点继续进行 dfs，并每次更新最大长度 $\max len$．可发现存在某些点有重复使用的情况，故会超时

```
int dir[4][2]={{-1,0},{0,1},{1,0},{0,-1}};
int a[MAXN][MAXN];
void dfs(int x,int y,int len){
    if(len>maxlen)
        maxlen=len;
    for(int i=0;i<=3;i++){
        int dx=x+dir[i][0],dy=y+dir[i][1];
        if(dx>=1&&dx<=n&&dy>=1&&dy<=m&&a[dx][dy]<a[x][y])
            dfs(dx,dy,len+1);
    }
}
```

②记忆化搜索+动态规划



$$f[i][j]=\max(\{f[i-1][j],f[i+1][j],f[i][j-1],f[i][j+1]\})+1$$

```
int dir[4][2]={{-1,0},{0,1},{1,0},{0,-1}};
int a[MAXN][MAXN],f[MAXN][MAXN];
int dfs(int x,int y){
    if(f[x][y]!=-1)
        return f[x][y];
    f[x][y]=1;//记得赋值为1
    for(int i=0;i<=3;i++){
        int dx=x+dir[i][0],dy=y+dir[i][1];
        if(dx>=1&&dx<=n&&dy>=1&&dy<=m&&a[dx][dy]<a[x][y])
            f[x][y]=max(f[x][y],dfs(dx,dy)+1);
    }
    return f[x][y];
}
```

例三 最大子段和

给出一个长度为 $n(1 \le n \le 2 \times 10^5)$ 的序列 a，选出其中连续且非空的一段使得这段和最大.

例：该序列选取区间 $[3,5]$ 的子段 $\{3,-1,2\}$，其和为 4.

2	-4	3	-1	2	-4	3

代码链接

①暴力枚举（超时）

枚举区间 $[i,j](1 \le i \le j \le n)$，将区间 $[i,j]$ 的数求和并更新最大值，时间复杂度为 $O(n^3)$.

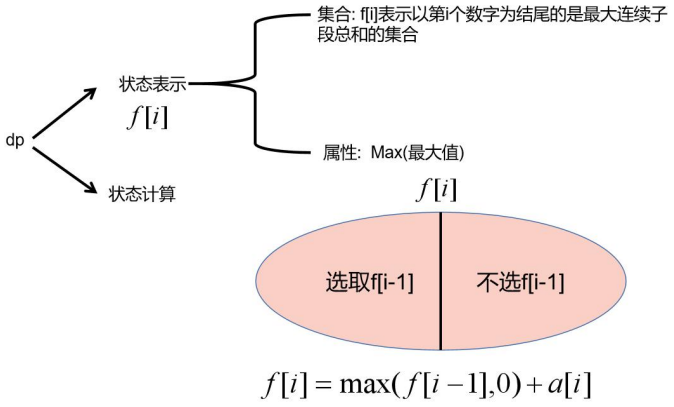
```
for(int i=1;i<=n;i++)
    for(int j=i;j<=n;j++){
        int sum=0;
        for(int k=i;k<=j;k++)
            sum+=a[k];
        maxsum=max(maxsum,sum);
    }
```

②前缀和（超时）

在暴力枚举区间 $[i,j]$ 的基础上，借助前缀和公式 $sum[j]-sum[i-1]$ 快速计算区间和，时间复杂度为 $O(n^2)$.

```
for(int i=1;i<=n;i++)
    for(int j=i;j<=n;j++)
        maxsum=max(maxsum,sum[j]-sum[i-1]);
```

③动态规划



$$f[i]=\max(f[i-1],0)+a[i]$$

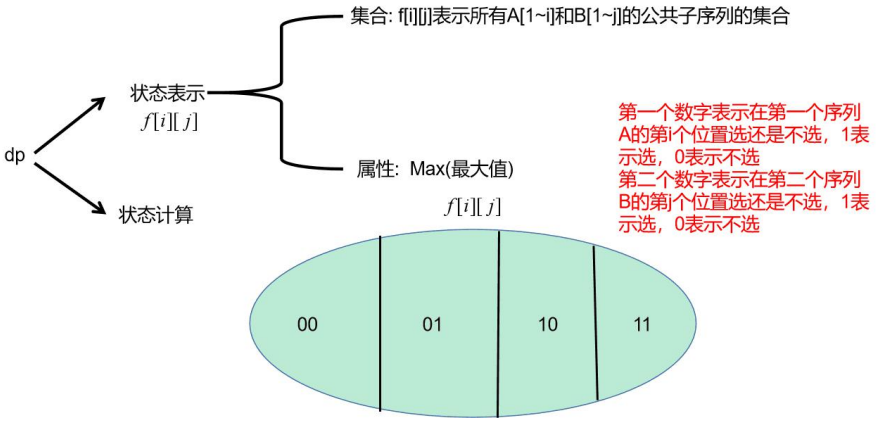
结果为 $res=\max(\{f[1],f[2],...,f[n]\})$，时间复杂度为 $O(n)$.

```
for(int i=1;i<=n;i++){
    f[i]=max(f[i-1],0)+a[i];
    maxsum=max(maxsum,f[i]);
}
```

例四 最长公共子序列（LCS(Longest Common Subsequence)问题）

给定两个长度分别为 n 和 m 的字符串 A 和 B，求既是 A 的子序列又是 B 的子序列的字符串长度最长是多少.

举例：字符串 A 为 acbd，字符串 B 为 abedc，它们的公共子序列为 abd（长度为 3）.



依据最后一个位置，可分为四类：

① $a[i],b[j]$ 均选进最长公共子序列中（11）；

② $a[i],b[j]$ 均不选进最长公共子序列中（00）；

③ $a[i]$ 不选进，$b[j]$ 选进（01）

④ $a[i]$ 选进，$b[j]$ 不选进（10）

第①类，须满足 $a[i]=b[j]$，此时 $f[i][j]=f[i-1][j-1]+1$（其中 $f[i-1][j-1]$ 表示的含义为所有 $A[1\sim i-1]$ 和 $B[1\sim j-1]$ 的公共子序列集合的最大值）；

第②类，此时 $f[i][j]=f[i-1][j-1]$（都不选，可直接由 $f[i-1,j-1]$ 转移过去）；

第③类，尝试用 $f[i-1,j]$ 对它进行计算，但是 $f[i-1,j]$ 并**不完全等价**于这一类.

这一类的含义是：“$a[i]$ 不选进，$b[j]$ 选进”，而 $f[i-1,j]$ 表示“在 $A[1\sim i-1]$ 中和 $B[1\sim j]$ 中公共子序列集合的最大值，**一定不包含** $a[i]$，**但可能包含** $b[j]$ **也可能不包含** $b[j]$ 两种情况”，即 $f[i-1,j]$ 表示的集合可分为两类：“含 $b[j]$”和“不含 $b[j]$”.

第③类实际上是与 $f[i-1,j]$ 分的第一类（含 $b[j]$）是恰好完全等价的，但是可发现没有一个式子能够**恰好覆盖（直接计算）**这一类.

对于求一个集合的 **数量属性** 时，我们要保证 “**不重不漏**”，但是求 **最值属性** 的时候我们只要保证 “**不漏**” 即可，“不重” 其实是无所谓的，不影响最终的答案.

例如：求三个数 a,b,c 的最大值，可以写成 $\max(\max(a,b),\max(b,c))$，虽然 b 被重复计算了两次，但不影响最大值的结果，只要保证不漏即可.

因此求第③类最大值的时候，可以**直接用 $f[i-1][j]$ 来覆盖**（注：并不是直接计算出这一类），$f[i-1][j]$ 表示集合**完全包含**第③类，且一定在 $f[i][j]$ 表示的总集合内部.

第④类，同理，可以**直接用 $f[i][j-1]$ 来覆盖**.

对于第②类，它既包含于第③类，又包含于第④类，故求最大值是只需关注 $f[i-1][j]$，$f[i][j-1]$，$f[i-1,j-1]+1$（需满足 $a[i]=b[j]$）

故状态转移方程为：

$$f[i][j]=\begin{cases}\max(f[i-1][j],f[i][j-1]),a[i]\neq b[j];\\ \max(\{f[i-1][j],f[i][j-1],f[i-1][j-1]+1\}),a[i]=b[j]\end{cases}$$

结果为 $f[n][m]$，时间复杂度为 $O(nm)$.　　代码链接

i \ j	0	1(a)	2(b)	3(c)	4(d)
0	0	0	0	0	0
1(b)	0	0	1	1	1
2(e)	0	0	1	1	1
3(c)	0	0	1	2	2
4(d)	0	0	1	2	3

$A:becd$

$B:abcd$

$$f[i][j]=\begin{cases}\max(f[i-1][j],f[i][j-1]),a[i]\neq b[j];\\ \max(\{f[i-1][j],f[i][j-1],f[i-1][j-1]+1\}),a[i]=b[j]\end{cases}$$

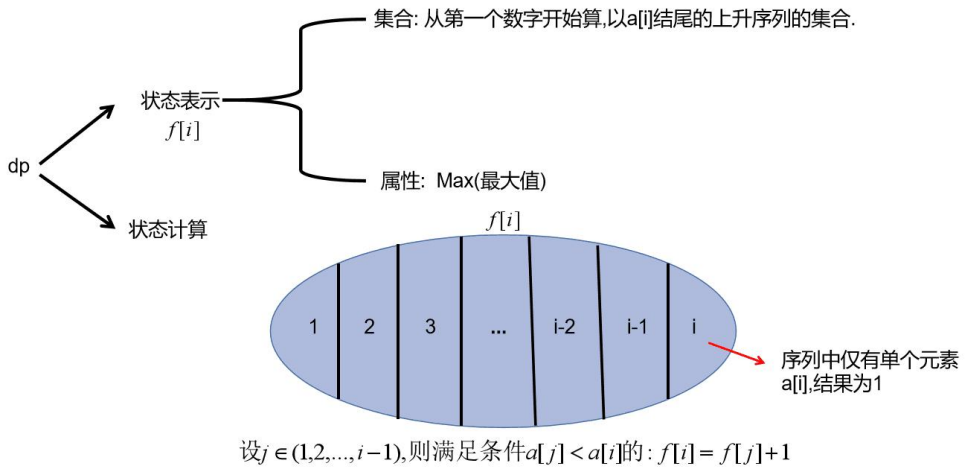
```
for(int i=1;i<=n;i++)
    for(int j=1;j<=m;j++){
        f[i][j]=max(f[i-1][j],f[i][j-1]);
        if(a[i]==b[j])
            f[i][j]=max(f[i][j],f[i-1][j-1]+1);
    }
```

例五 最长上升子序列（LIS(Longest Increasing Subsequence)问题）

给定一个长度为 N 的数列，求数值严格单调递增的子序列 A 的长度最长是多少？

例如：数列 $\{3,1,2,1,8,5,6\}$ 的最长上升子序列为 $\{1,2,5,6\}$（长度为 4）.

①朴素动态规划



边界：若前面无比 $a[i]$ 小的元素，$f[i]=1$；

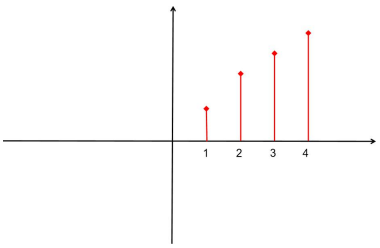
状态转移方程为：$f[i]=\max(f[i],f[j]+1),a[j]<a[i]且j\in\{1,2,...,i-1\}$.

时间复杂度为 $O(n^2)$.　　代码链接

```
#include<iostream>

#define MAXN 1010
using namespace std;

int n,a[MAXN],maxlen,pos;
int f[MAXN],pre[MAXN];
//拓展:输出最长上升子序列的元素(注最长上升子序列可能不唯一)
void printpath(int pos){
    if(pre[pos]==0){
        printf("%d",a[pos]);
        return ;
    }
    printpath(pre[pos]);
    printf(" %d",a[pos]);
}

int main(){
    scanf("%d",&n);
    for(int i=1;i<=n;i++)
```

```
        scanf("%d",&a[i]);
    for(int i=1;i<=n;i++){
        f[i]=1;
        for(int j=1;j<=i-1;j++){
            /*if(a[j]<a[i])
                f[i]=max(f[i],f[j]+1);*/
            if(a[j]<a[i]&&f[i]<f[j]+1){
                f[i]=f[j]+1;
                pre[i]=j;
            }
        }
    }
    for(int i=1;i<=n;i++)
        if(f[i]>maxlen){
            maxlen=f[i];
            pos=i;
        }
    printf("%d\n",maxlen);
    printpath(pos);
    return 0;
}
```

②贪心+二分（不讲，可参考了解）



坐标轴 4 表示的是序列长度为 4，序列结尾的最小值.

例如:1 2 4 7 和 1 2 4 9

应选择哪一个更好呢?贪心选择较小的.

假设在最后插入 8，1 2 4 7 8 更优于使用 1 2 4 9 8

状态表示：$f[i]$ 表示长度为 i 的最长上升子序列，末尾最小的数字的集合.

状态计算：对于每一个 $a[i]$，如果大于 $f[cnt-1]$，那就 $cnt+1$，使得最长上升子序列长度+1，当前末尾的最小元素为 $a[i]$；如果 $a[i]$ 小于等于 $f[cnt-1]$，说明不会更新当前的长度，但之前末尾的最小元素要发生改变，找到第一个大于等于 $a[i]$，更新该末尾的最小元素.

$f[i]$ 一定是一个单调递增的数组，所以可用**二分法**找第一个大于或等于 $a[i]$ 的数字.

结果为 $cnt-1$. 时间复杂度为 $O(n\log n)$.

```cpp
#include<iostream>

#define MAXN 1010
using namespace std;

int n,a[MAXN],cnt=1;
int f[MAXN];

int main(){
    scanf("%d",&n);
    for(int i=1;i<=n;i++)
        scanf("%d",&a[i]);
    f[cnt++]=a[1];
    for(int i=2;i<=n;i++){
        if(a[i]>f[cnt-1])
            f[cnt++]=a[i];
        else{
            int l=1,r=cnt;
            while(l<r){
                int mid=(l+r)/2;
                if(f[mid]>=a[i])
                    r=mid;
                else l=mid+1;
            }
            f[r]=a[i];
        }
    }
    printf("%d\n",cnt-1);
    return 0;
}
```

- - - 后三种不讲,可自行学习

例六 石子合并（区间动态规划）

设有 N 堆石子排成一排，其编号为 1，2，3，$\ldots$，N。

每堆石子有一定的质量，可以用一个整数来描述，现在要将这 N 堆石子合并成为一堆。

每次只能合并相邻的两堆，合并的代价为这两堆石子的质量之和，合并后与这两堆石子相邻的石子将和新堆相邻，合并时由于选择的顺序不同，合并的总代价也不相同。

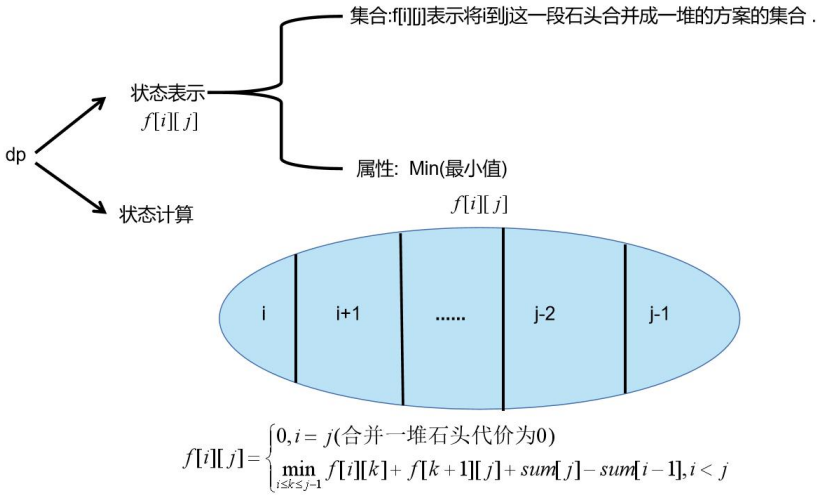
例如有 4 堆石子分别为 `1 3 5 2`，我们可以先合并 1、2 堆，代价为 4，得到 `4 5 2`，又合并 1，2 堆，代价为 9，得到 `9 2`，再合并得到 11，总代价为 $4+9+11=24$；

如果第二步是先合并 2，3 堆，则代价为 7，得到 `4 7`，最后一次合并代价为 11，总代价为 $4+7+11=22$。

问题是：找出一种合理的方法，使总的代价最小，输出最小代价。

关键点：最后一次合并一定是**左边连续的一部分**和**右边连续的一部分**进行合并.

设 $sum[]$ 为 n 堆石头的前缀和数组.



状态转移方程为：

$$f[i][j]=\begin{cases}0, i=j\\ \min\limits_{i\le k\le j-1} f[i][k]+f[k+1][j]+sum[r]-sum[i-1], i<j\end{cases}$$

初始化 f 数组为无穷大，结果为 $f[1][n]$，时间复杂度为 $O(n^3)$　　代码链接

```
memset(f,0x3f,sizeof(f));
for(int len=1;len<=n;len++){//枚举长度
    for(int l=1;l+len-1<=n;l++){//枚举左端点
        int r=l+len-1;//确定右端点
        if(len==1)
            f[l][r]=0;
        else
            for(int k=l;k<=r-1;k++)//枚举 k 的位置
                f[l][r]=min(f[l][r],f[l][k]+f[k+1][r]+sum[r]-sum[l-1]);
    }
}
```

假如问最大值，与最小值求法类似，注意**初始化为无穷小**.

拓展：环形石子合并（将环转换成链式，自行尝试做一做） 代码链接

将 n 堆石子绕圆形操场排放，现要将石子有序地合并成一堆。

规定每次只能选相邻的两堆合并成新的一堆，并将新的一堆的石子数记做该次合并的得分。

请编写一个程序，读入堆数 n 及每堆的石子数，并进行如下计算：

- 选择一种合并石子的方案，使得做 $n-1$ 次合并得分总和最大。
- 选择一种合并石子的方案，使得做 $n-1$ 次合并得分总和最小。

例七 最长公共上升子序列

小沐沐先让奶牛研究了最长上升子序列，再让他们研究了最长公共子序列，现在又让他们研究最长公共上升子序列了。

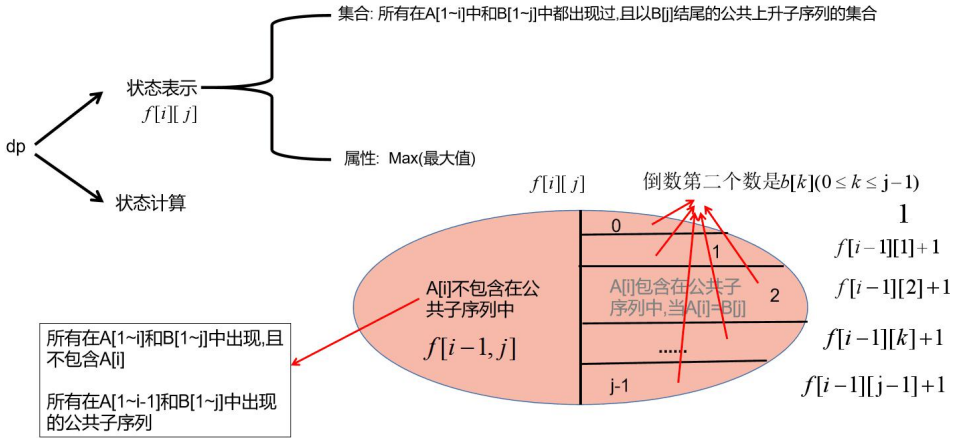
小沐沐说，对于两个数列 A 和 B，如果它们都包含一段位置不一定连续的数，且数值是严格递增的，那么称这一段数是两个数列的公共上升子序列，而所有的公共上升子序列中最长的就是最长公共上升子序列了。

奶牛半懂不懂，小沐沐要你来告诉奶牛什么是最长公共上升子序列。

不过，只要告诉奶牛它的长度就可以了。

数列 A 和 B 的长度均不超过 3000。

①朴素动态规划——时间复杂度为 $O(n^3)$ 代码链接



状态计算(文字版)：

依据公共子序列中是否包含 $a[i]$，将 $f[i][j]$ 所代表的集合划分成两个不重不漏的子集：

- 不包含 $a[i]$ 的子集，最大值是 $f[i-1][j]$；
- 包含 $a[i]$ 的子集(此时意味着 $a[i]=b[j]$)，将这个子集继续划分，依据是子序列的倒数第二个元素在 b 中是哪个数：
  - 子序列只包含 $b[j]$ 一个数，长度是 1，即图中为 0 的情况；
  - 子序列的倒数第二个数是 $b[1]$ 的集合，最大长度是 $f[i-1][1]+1$；

✧ ...

✧ 子序列的倒数第二个数是 b[j-1] 的集合，最大长度是 $f[i-1][j-1]+1$；

按上述思路实现，需要三重循环来解决：

```
for(int i=1;i<=n;i++)
    for(int j=1;j<=n;j++){
        f[i][j]=f[i-1][j];
        if(a[i]==b[j]){
            int tempmax=1;
            for(int k=1;k<=j-1;k++)
                if(b[k]<b[j])
                    tempmax=max(tempmax,f[i-1][k]+1);
            f[i][j]=max(f[i][j],tempmax);
        }
    }
```

②优化动态规划——$O(n^2)$

可发现每次循环求的 maxv 是满足 $b[k]<a[i]$ 的 $f[i-1][k]+1$ 的前缀最大值. 可直接将 maxv 提到第一层循环外面，减少重复计算，此时只剩下两重循环.

最终答案枚举子序列结尾取最大值即可，时间复杂度为 $O(n^2)$.

```
for(int i=1;i<=n;i++){
    int tempmax=1;
    for(int j=1;j<=n;j++){
        f[i][j]=f[i-1][j];
        if(a[i]==b[j])
            f[i][j]=max(f[i][j],tempmax);
        if(b[j]<a[i])
            tempmax=max(tempmax,f[i-1][j]+1);
    }
}
```

例八 矩阵连乘——和例六类似(此题不再详细讲解)　　代码链接

有n个矩阵，大小分别为
$a[0]*a[1], a[1]*a[2], a[2]*a[3], \ldots, a[n-1]*a[n]$，现要将它们依次相乘，只能使用结合率，求最少需要多少次运算。 (2<=n<=1000 , a[i]<=100)

两个大小分别为$p*q$和$q*r$的矩阵相乘时的运算次数计为$p*q*r$。

注意不同的运算顺序会导致运算次数不一样，以样例为例:

如果我们先算前两个矩阵的乘积，将运算$1*10*5=50$次，并得到一个$1*5$的矩阵，之后再算这个$1*5$的矩阵和最后一个$5*20$的矩阵的乘积，将运算$1*5*20=100$次，共运算150次;

如果我们先算后两个矩阵的乘积，将运算$10*5*20=1000$次，并得到一个$10*20$的矩阵，之后再算第一个$1*10$的矩阵和这个$10*20$的矩阵的乘积，运算$1*10*20=200$次，共运算1200次。

样例：$\{1,10,5,20\}$——结果为 150.

状态表示：$f[i][j]$表示区间$[i,j]$内的矩阵连乘的次数的集合.

状态计算：

$$f[i][j]=\begin{cases}0, i=j\\ \min\limits_{i\le k\le j-1} f[i][k]+f[k+1][j]+a[i-1]\times a[k]\times a[j], i<j\end{cases}$$

```
memset(f,0x3f,sizeof(f));
for(int len=1;len<=n;len++)
    for(int l=1;l+len-1<=n;l++){
        int r=l+len-1;
        if(l==r)
            f[l][r]=0;
        else
            for(int k=l;k<=r-1;k++)
                f[l][r]=min(f[l][r],f[l][k]+f[k+1][r]+a[l-1]*a[k]*a[r]);
    }
```