

动态规划初步

Wang Houfeng

EECS, PKU

wanghf@pku.edu.cn

内容

➤ 动态规划基本概念

- 几个典型的例子

什么是动态规划

- **动态规划**：将复杂问题分解为相对简单子问题再由子问题求解使复杂问题得以求解的方法：
 - 一种多阶段决策过程的**最优化**方法，把 n 阶段的决策变成**分阶段**的优化问题（**划分阶段非常重要**）
 - 对于求解的问题，按一定的策略分解为子问题求解，再根据子问题推导原始问题的整体解
 - 分解后的**每个子问题求解方法通常相同**，而且存在重复求解各子问题的情况。这样，通过求解每个子问题一次，之后再需要求解时直接利用已有的结果，避免重复求解（称为**记忆式使用**）

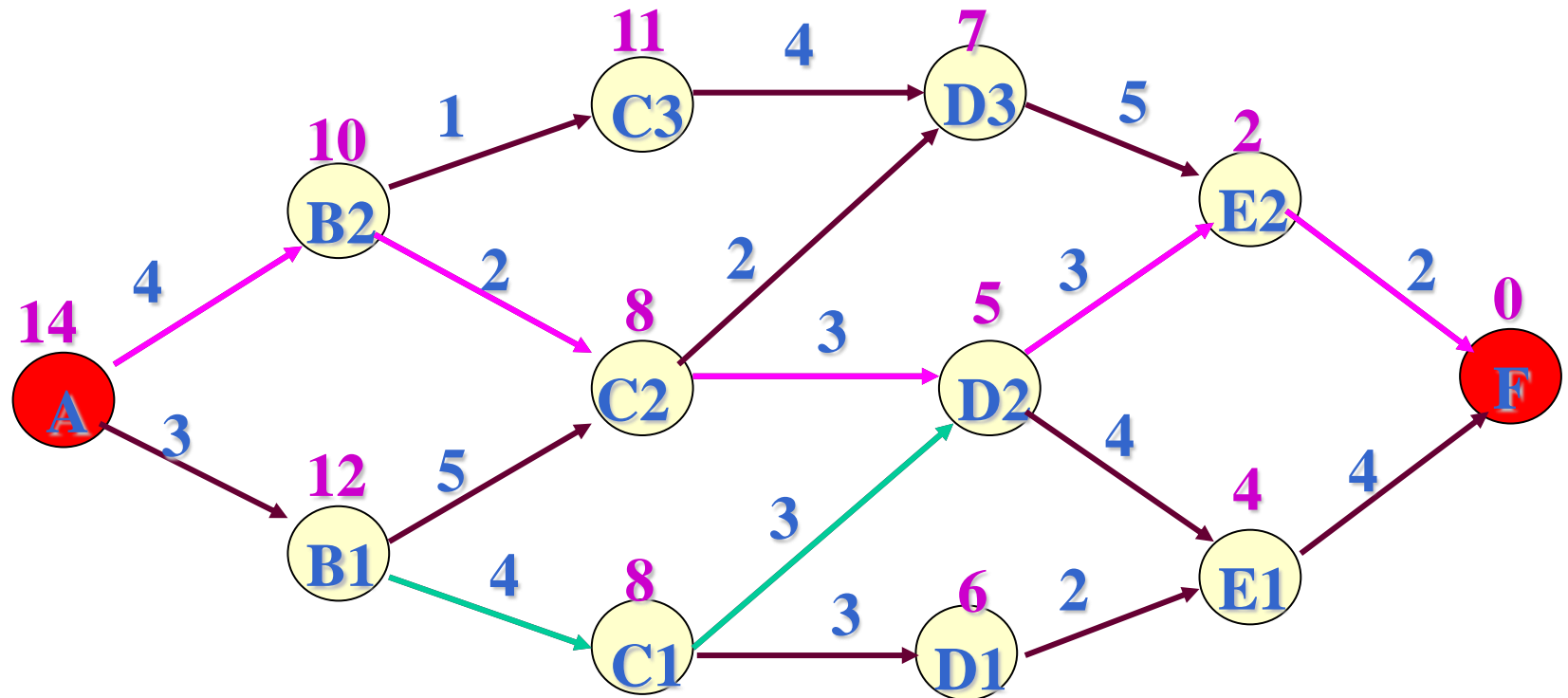
最优化，极值

基本原理

- 多阶段逐步决策，充分利用已经计算的结果
- 基本框架：
 - 按时间或空间将问题分解成若干个相互联系的阶段，以便按次序去求每阶段的解，常用 k 表示阶段变量（重要）。
 - 各阶段开始时的客观条件叫做状态。描述各阶段状态的变量称为状态变量，常用 s_k 表示第 k 阶段的状态变量，变量的取值集合称为状态集合，用 S_k 表示。
 - 当某阶段状态给定以后，之后过程的发展不受以前阶段状态的影响。只能通过当前状态影响未来，这称为无后效性。如果所选定的变量不具备无后效性，就不能作为状态变量来构造动态规划模型。

分阶段：形成统一的递推表示

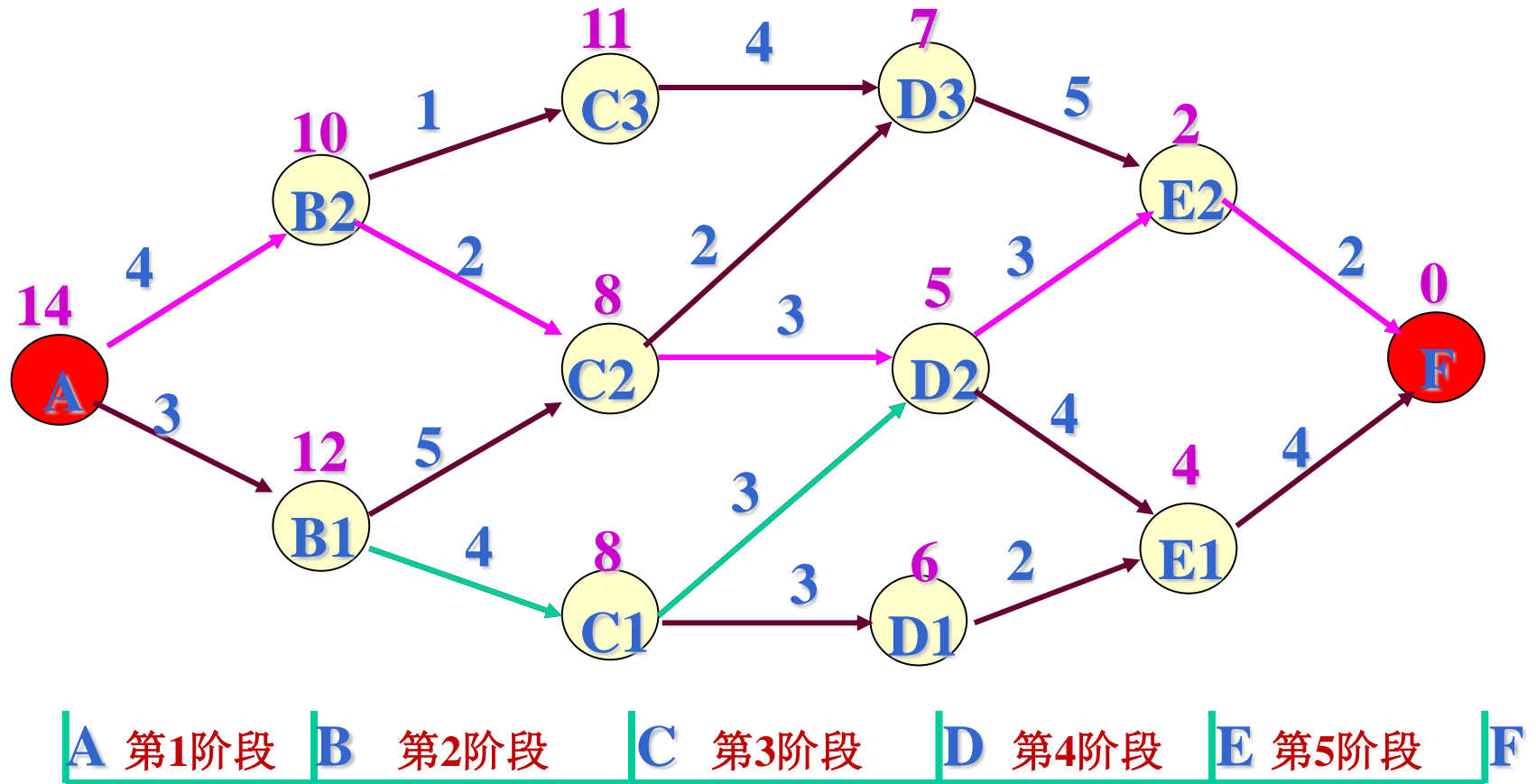
以最短路径问题为例——阶段决策



5阶段决策

A 第1阶段 | B 第2阶段 | C 第3阶段 | D 第4阶段 | E 第5阶段 | F

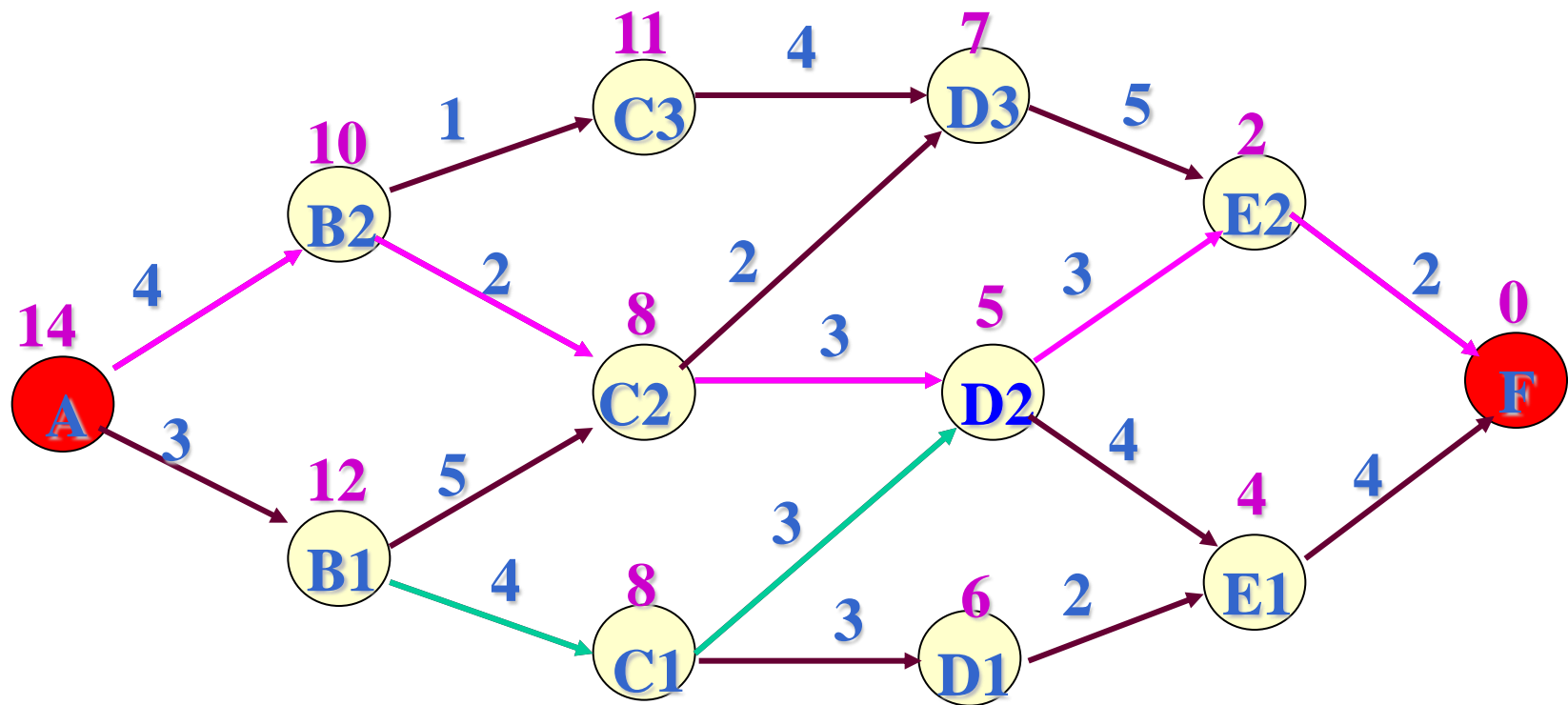
通用的最短路径问题—阶段



终点为F。从最后阶段开始决策 ($E \rightarrow F$) : $f_5(E1)=4$, $f_5(E2)=2$

也可以从起点 A 开始决策

通用的最短路径问题—阶段

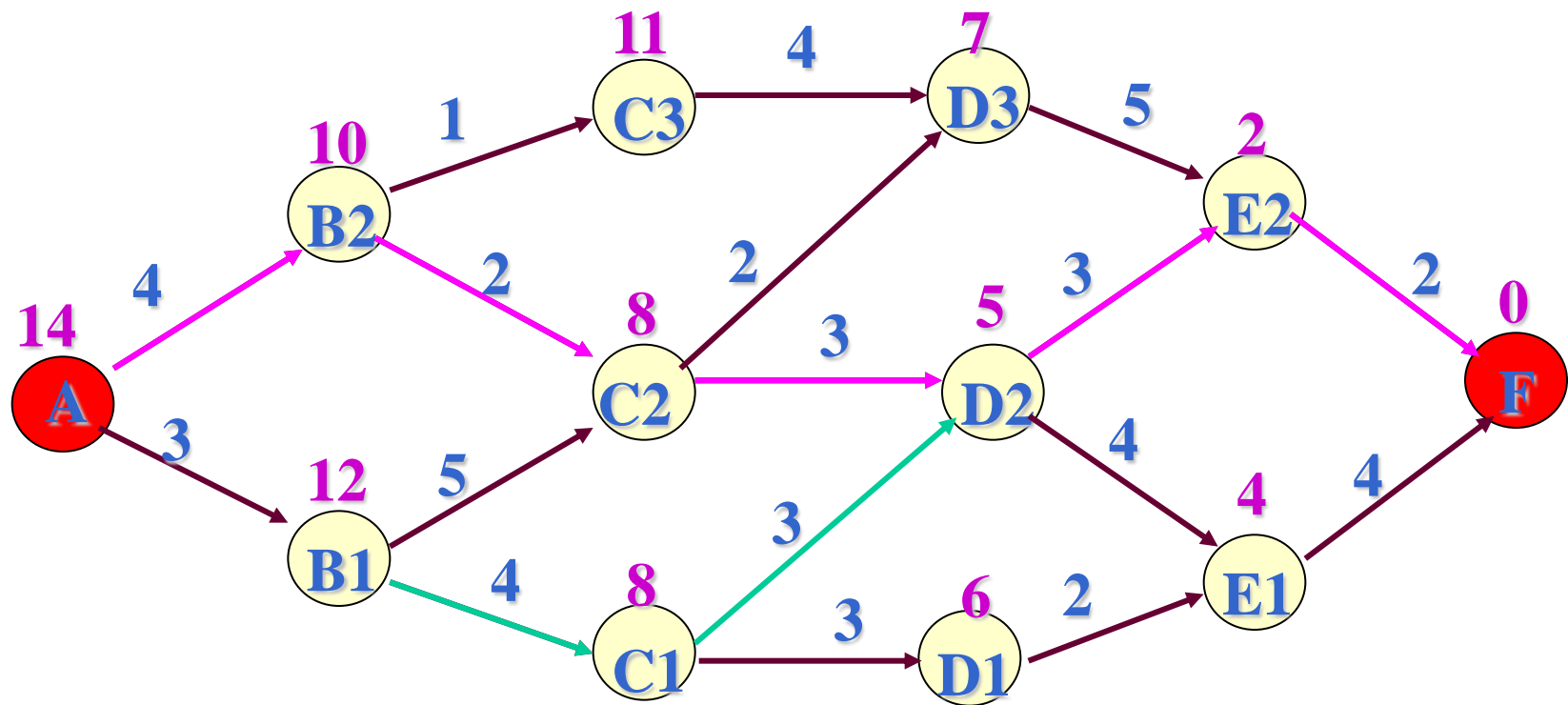


再考虑第 4 阶段的决策 ($D \rightarrow E$) : 以 $D2 \rightarrow$ 经 E 到 F 为例

$$f_4(D_2) = \min \left\{ \begin{array}{l} d(D_2, E_2) + f_5(E_2) \\ d(D_2, E_1) + f_5(E_1) \end{array} \right\} = \min \left\{ \begin{array}{l} 3 + 2 \\ 4 + 4 \end{array} \right\} = 5$$

路径为: $D2 \rightarrow E2 \rightarrow F$, D 的决策只与相连的 E 相关, 不考虑 F

通用的最短路径问题—阶段



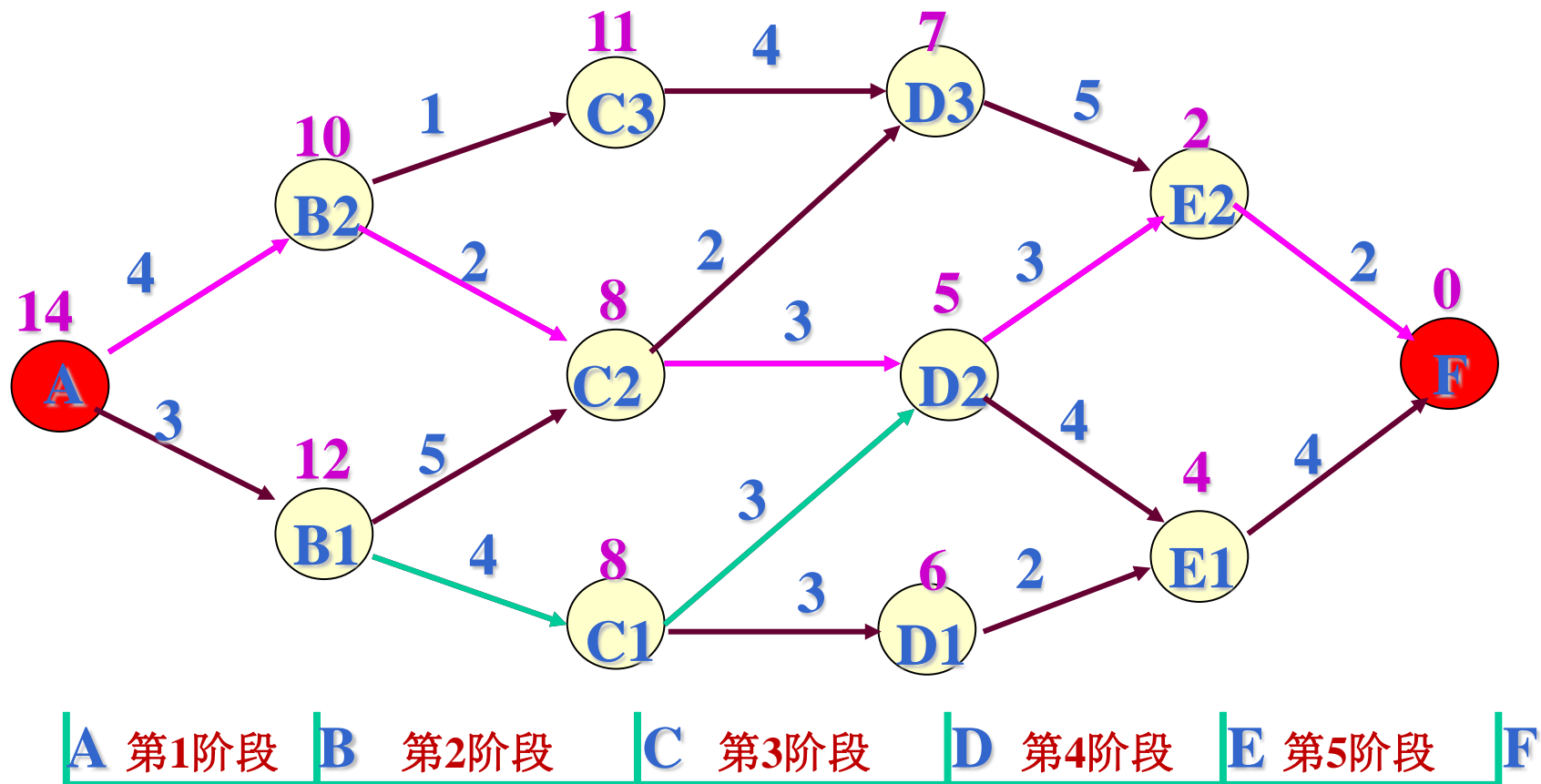
考虑第3 阶段 ($C \rightarrow F$) , 以C1 到 F为例:

$$f_3(C_1) = \min \left\{ \begin{array}{l} d(C_1, D_2) + f_4(D_2) \\ d(C_1, D_1) + f_4(D_1) \end{array} \right\} = \min \left\{ \begin{array}{l} 3 + 5 \\ 3 + 6 \end{array} \right\} = 8$$

同理, C2到F:

$$f_3(C_2) = \min \left\{ \begin{array}{l} d(C_2, D_2) + f_4(D_2) \\ d(C_2, D_3) + f_4(D_3) \end{array} \right\} = \min \left\{ \begin{array}{l} 2 + 7 \\ 3 + 5 \end{array} \right\} = 8$$

通用的最短路径问题—阶段



如前类似，之后再考虑第2阶段（B）、第1阶段（A）的决策。
通过逐次分阶段决策，最终得到整体决策（从A \rightarrow ... \rightarrow ...F），
每一次决策只与之前（右）相邻的结果相关，与更前的无关。

再看一个具体例子

- 最短路径问题

1

2 3

6 5 4

9 7 8 10

当前层的节点只能经过下面层的
左侧或右侧节点往下

找出从第一层到最后一层的一条路,使得
所经过的数值之和最小

输入格式

1

2 3

6 5 4

9 7 8 10

(行,列)



令： $f(i, j)$ 为三角形上从点 (i, j) 出发向下走到底层的最短路经(数值和),于是:

$$f(i, j) = a[i, j] + \min\{f(i+1, j), f(i+1, j+1)\}$$

枚举求解

- 递归过程：

$x = f(i+1, j);$

$y = f(i+1, j+1);$

if ($x > y$) $x = y;$

result = $x + a[i, j];$

- 问题：

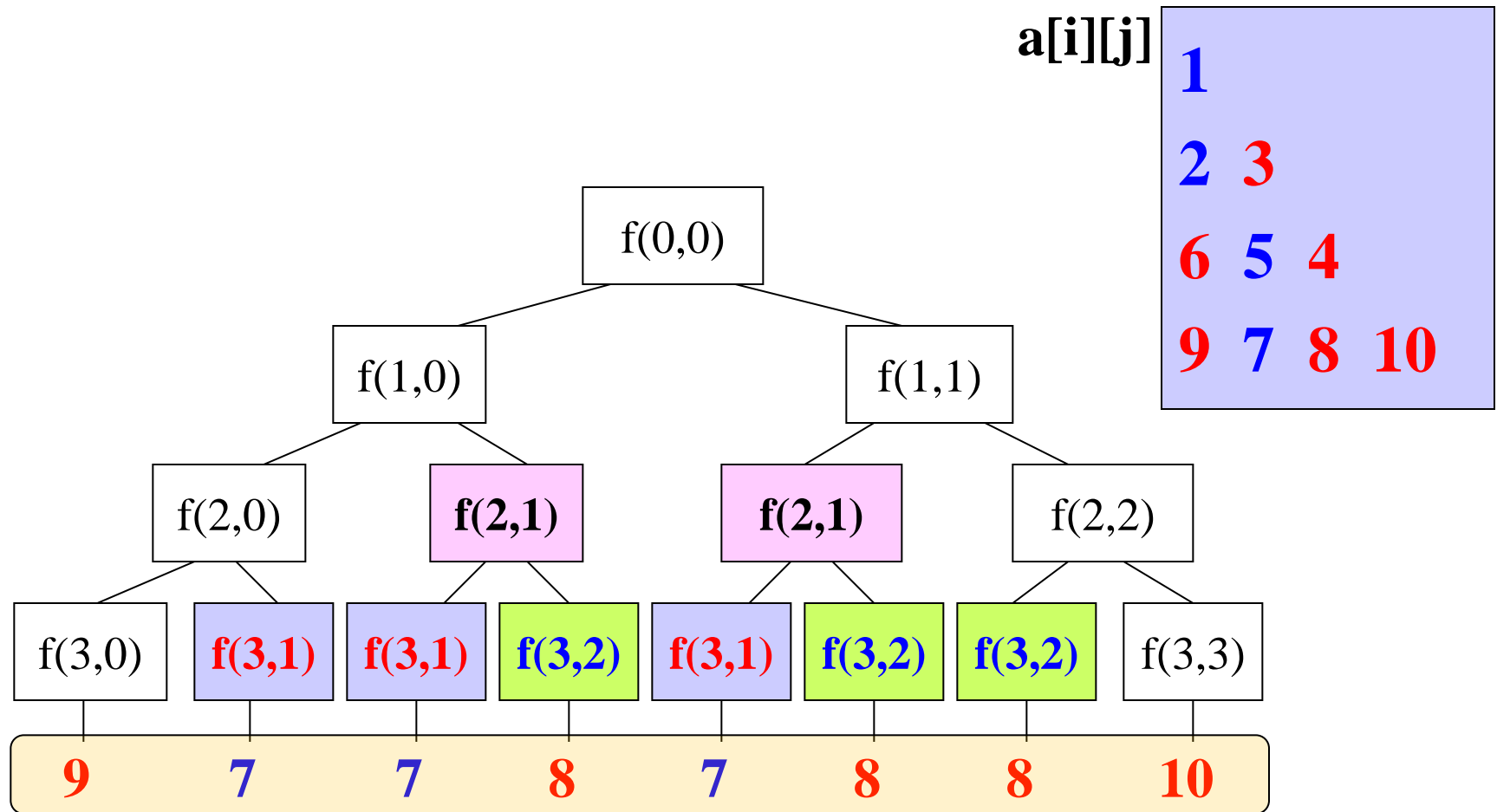
- 容易超时。

- 超时原因：大量重复计算

```
int f(int i, int j) //递归
{
    if(i==n) return 0;
    int x = f(i+1, j);
    int y = f(i+1, j+1);
    if(x > y) x = y;
    return x+a[i][j];
}
```

```
main()
{... printf("result=%d\n", f(0,0)); }
```

再看哪些重复计算



如果继续向下延伸?

重复计算

- 看重重复计算:

- $f(3,1)$: 3 次计算;

- $f(3,2)$: 3 次计算;

- $f(2,1)$: 2 次计算;

深度增加, 重复计算量大幅增加

- 如何避免重复计算, 使得:

- $f(3,1)$: 仅1 次计算;

- $f(3,2)$: 仅1 次计算;

- $f(2,1)$: 仅1 次计算;

记忆!

动态规划

记忆化搜索，
避免重复计算
从下往上逐层推

```
int f(int n)
{  int i, j, temp;
    for(i=n-2;i>=0;i--)
        for(j=0;j<=i;j++)
        {
            if (a[i+1][j] < a[i+1][j+1]) temp=a[i+1][j];
            else temp=a[i+1][j+1];
            a[i][j] += temp;
        }
    return a[0][0];
}
```

i表示层，j表示层中的元素

从下往上计算
每个 **f(i,j)** 仅计算1次
直接保留在 **a** 中！

类似问题：收益最大路径

- $n*n$ 的方阵，从左上角进，右下角出。每次只能**向右**或者**向下**走一步。每一块小方格上有一个收益值。按什么路径走，才能使收益之和最大？

2	8	5	1	10
5	9	7	2	5
6	6	2	1	2
9	6	3	8	7
9	9	9	4	3

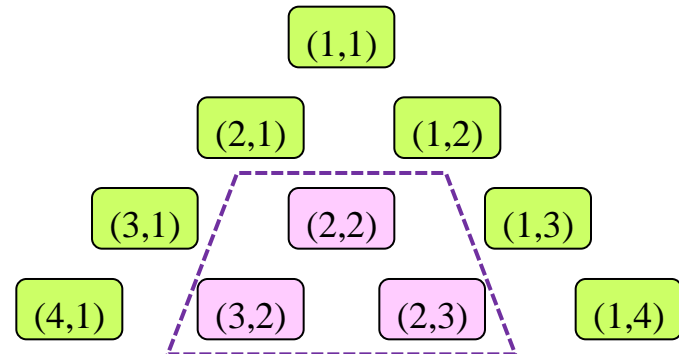
最大收益值：56

收益最大路径分析

- 用 $v(i, j)$ 表示第 i 行第 j 列的收益值，其中 (i, j) 表示行列位置
- 引入与 v 同样大小的 f ，记录从左上角进入，到达每一块方格的最大收益和

v	1	2	3	4	5
1	2	8	5	1	10
2	5	9	7	2	5
3	6	6	2	1	2
4	9	6	3	8	7
5	9	9	9	4	3

本质上可以转化为前面例子的结构：
二叉树表示，向下用左分支表示，
向右用右分支表示



到达每一块小方格的收益最大值

- $f(0, 0) = 0$
- $f(i, j) = v(i, j) + \max\{f(i-1, j), f(i, j-1)\}$

上面

左边

v	1	2	3	4	5
1	2	8	5	1	10
2	5	9	7	2	5
3	6	6	2	1	2
4	9	6	3	8	7
5	9	9	9	4	3

f	1	2	3	4	5
1	2	10	15	16	26
2	7	19	26	28	33
3	13	25	28	29	35
4	22	31	34	42	49
5	31	40	49	53	56

用递归算法计算 $f(i,j)$

```
int f(int i, int j)
```

```
{ if(i==0||j==0) return 0; 额外增加第0行第0列，其f值设为0
```

```
else return (v[i][j]+  $\max(f(i,j-1), f(i-1,j))$ );
```

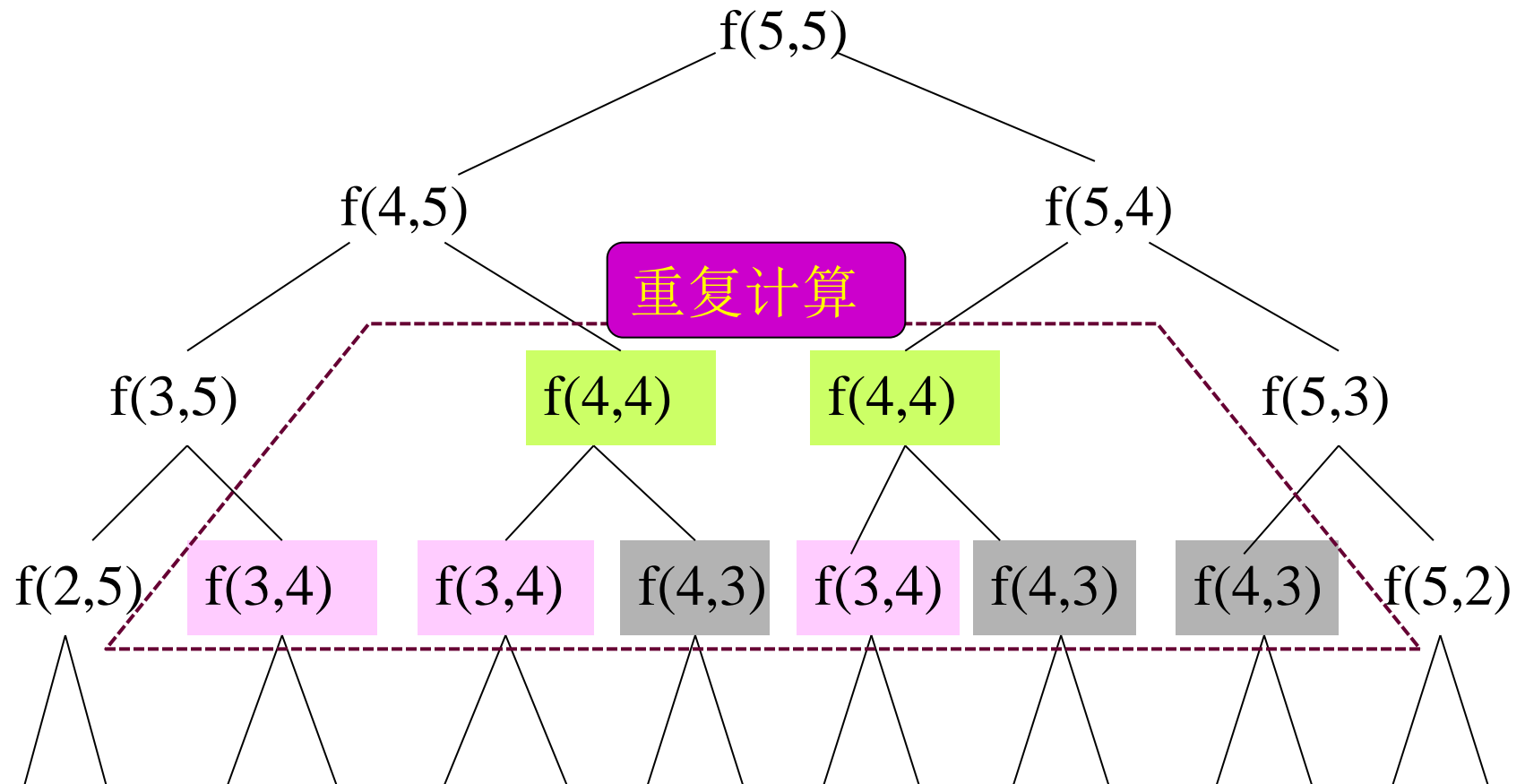
```
}
```

多次重复计算，复杂度太高：Timeout!

v	1	2	3	4	5
1	2	8	5	1	10
2	5	9	7	2	5
3	6	6	2	1	2
4	9	6	3	8	7
5	9	9	9	4	3

f	1	2	3	4	5
1	2	10	15	16	26
2	7	19	26	28	33
3	13	25	28	29	35
4	22	31	34	42	49
5	31	40	49	53	56

递归式子展开




递归导致了大量的重复计算

如何优化？记忆化搜索

动态规划：记忆性搜索

```
#define max(x,y) (x>y? x:y)
int main()
{  int i,j,v[6][6]={0},f[6][6]={0};
    for(i=1; i<6; ++i)
        for(j=1; i<6; ++j)
        {
            cin>>v[i][j];
            f[i][j]=max(f[i-1][j], f[i][j-1])+v[i][j];
        }
    cout<<f[5][5];
    return 0;
}
```

递归改迭代



内容

➤ 动态规划基本概念

➤ 几个典型的例子

例1：拦截导弹问题

最长降序子序列

某国为了防御敌国导弹袭击，开发了一种导弹拦截系统。但导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够打到任意高度，但以后的每一发都不高于前一发的高度。

某天，雷达发现了敌国导弹来袭，并观察到导弹依次飞来的高度，请计算这套拦截系统最多能拦截多少导弹。

拦截来袭导弹时，必须按照来袭导弹袭击的时间顺序，不允许先拦截后到的导弹，再拦截先到的导弹。

输入：2行

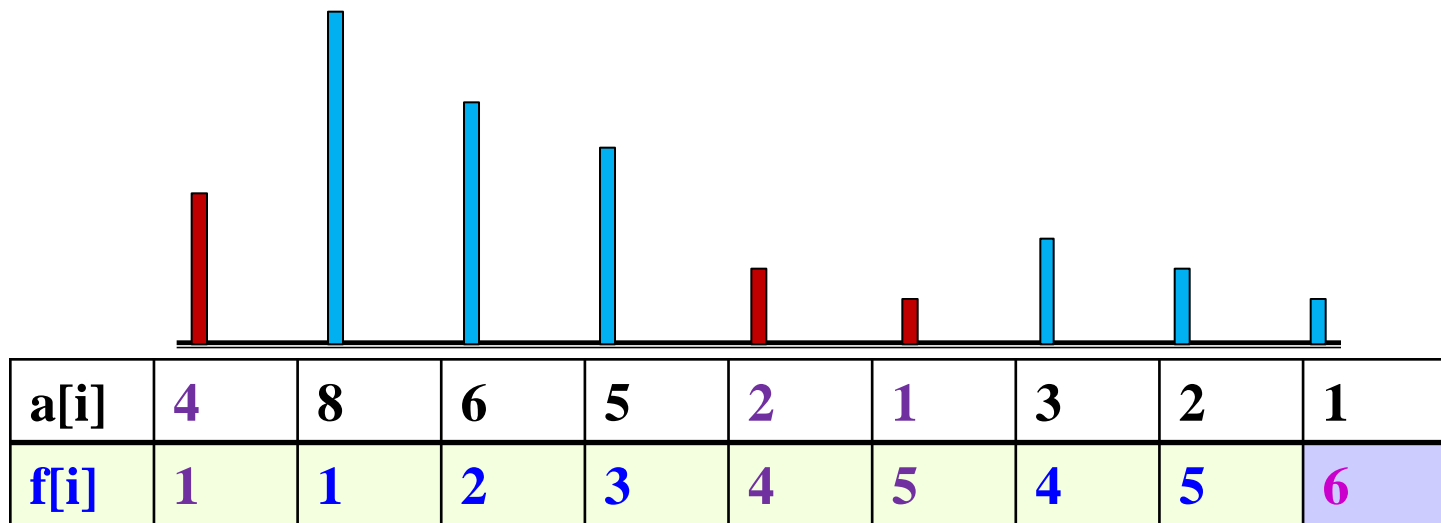
第1行：来袭导弹的数量 k

第2行： k 枚导弹依次的高度（按到达的时间顺序）

输出：1行

拦截导弹问题的一个例子

- **题目抽象：** 一个整数序列 X_1, X_2, \dots, X_n ，用从该序列中删除若干数的方法，使剩下的长度为 m 的序列为单调递减序列。注意：不可改变原序列中的数字先后顺序。要求构造出的序列为最长序列——**最长降序子序列**
- **假如序列为：** 4, 8, 6, 5, 2, 1, 3, 2, 1 ($n=9$)
- **删除3个数：** 4, 8, 6, 5, 2, 1, 3, 2, 1 / 4, 8, 6, 5, 2, 1, 3, 2, 1



导弹拦截



- 设原始序列为: $a[i]$;
- 引入序列 $f[i]$, 表示 $a[0], a[1], \dots, a[i]$ 子序列中最长的降序子序列长度。于是, 有:
 - $f[i] = \max\{1, f[j] + 1\}$, 其中, $0 \leq j < i$ 且 $a[i] \leq a[j]$
 - 如果 $a[i]$ 前面 (左边) 的值均小于 $a[i]$ 则取值1;
 - 如果 $a[i]$ 前面存在比 $a[i]$ 大的值, 则找满足条件的最大 $f[j]$ 再加1
- 序列 $a[i]$ 的最长降序子序列必为 $\{f[0], f[1], \dots, f[n]\}$ 最大值

导弹拦截

a[i]	4	8	6	5	2	1	3	2	1
f[i]	1	1	2	3	4	5	4	5	6

```
int solve(int f[], int a[], int n);
```

```
{  int i, j, ans=0;
```

```
    f[0]=1;
```

```
    for( i=1; i<n; ++i )
```

```
    {    f[i]=1;
```

```
        for(j=0; j<i; j++)
```

```
            if(a[i]<=a[j])&&(f[i]<f[j]+1)
```

```
                f[i]=f[j]+1;
```

```
    }
```

```
    for(i=0; i<n; ++i)
```

```
        if(f[i]>ans) ans=f[i];
```

```
    return ans;
```

```
}
```

严格单调下降应改为“<”

找 i 时刻的最大值

最后选择最大的

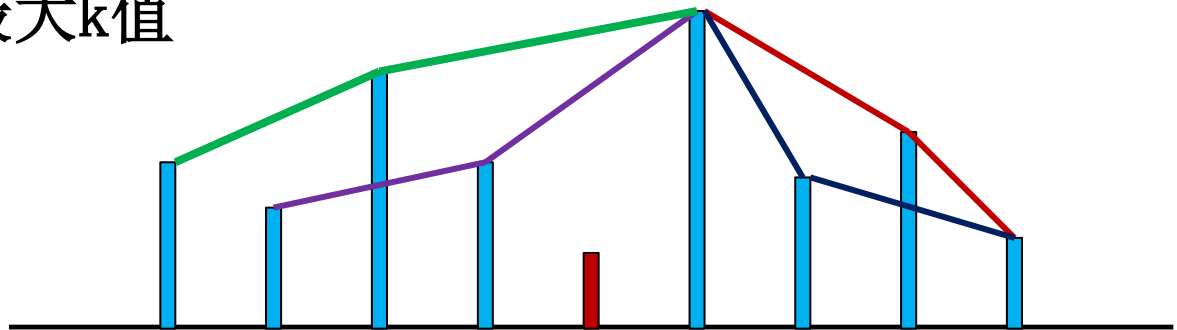
例2：合唱队队形

- **题目：** n 位同学站成 1 行队列，音乐老师要请 $(n-k)$ 名同学出队列，剩下的 k 位同学不交换顺序便能站成合唱队队形。假定剩余的 k 位同学的身高依次为 X_1, X_2, \dots, X_k ，则，中间一定存在某个 m 满足：

X_1, X_2, \dots, X_m 是严格单调增序， X_m, X_{m+1}, \dots, X_k 为严格单调降序。

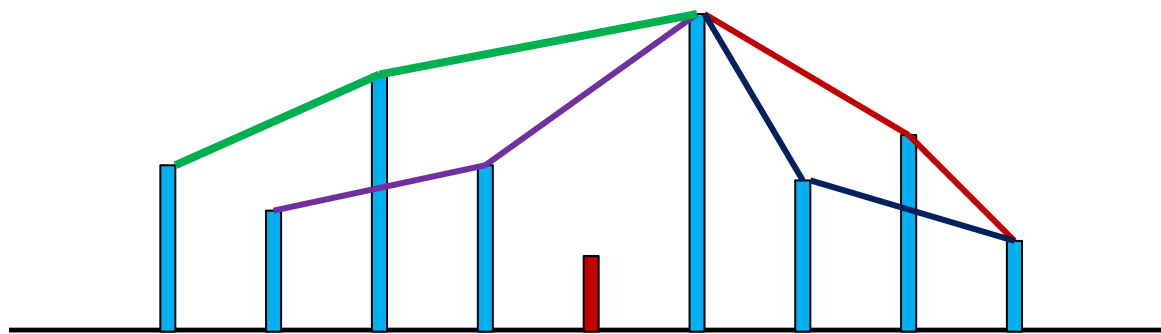
若 $m=1$ ；则整个序列为严格单调降序；若 $m=k$ ，则为严格单调增序。

- **问题：** 求最大 k 值



合唱队队形问题分析

- 与导弹拦截比较？
 - 导弹拦截只有一个最长单调降序序列
 - 合唱队队形两个单调子序列：左边为单增，右边为单降
- 引入如下表示：
 - $a[m]$ 表示从左边开始，到 m 为最高点的 longest 单增子序列长度；
 - $b[m]$ 表示从 m 开始到最末位置为止，以 m 为最高点的 longest 降序子序列的长度；
 - $c[m] = a[m] + b[m] - 1$ ，那么最大 $c[m]$ 值变为最长合唱队长度



合唱队队形分析

- 如何计算 $a[m]$?
 - 假设 n 位学生的身高依次为 $h[0], h[1], \dots, h[n-1]$
 - 显然: $a[0]=1$;
 - $a[m]=\max\{1, a[j]+1\}$, 其中, $0 \leq j < m$ 且 $h[m]>h[j]$;
 - 与导弹拦截的方法有何区别?
 - 没有区别!
- 如何计算 $b[m]$?
 - 与上面的增序计算方法几乎一样, 不同之处在哪里?
 - 从尾部向首部反向计算
 - $b[n-1]=1$;
 - $b[m]=\max\{1, b[j]+1\}$, 其中, $m < j \leq n-1$ 且 $h[m]>h[j]$;
- 计算 $c[m]$: $c[m]=a[m]+b[m]-1$

合唱队队形算法


- 自己实现

例3：采药


描述：辰辰是个天资聪颖的孩子，他的梦想是成为世界上最伟大的医师。为此，他想拜附近最有威望的医师为师。医师为了判断他的资质，给他出了一个难题。医师把他带到一个到处都是草药的山洞里对他说：“孩子，这个山洞里有一些不同的草药，采每一株都需要一些时间，每一株也有它自身的价值。我会给你一段时间，在这段时间里，你可以采到一些草药。如果你是一个聪明的孩子，你应该可以让采到的草药的总价值最大。” 如果你是辰辰，你能完成这个任务吗？

输入：第一行有两个整数 T ($1 \leq T \leq 1000$) 和 M ($1 \leq M \leq 100$)，用一个空格隔开， T 代表采药的全部时间， M 代表山洞里的草药的数目。接下来的 M 行每行包括两个在1到100之间（包括1和100）的整数，分别表示采摘某株草药的**时间**和这株草药的**价值**。

输出：包括一行，只有一个整数，表示在规定的时间内，可以采到的草药的最大总价值。

输入示例： 

70	3
71	100
69	1
1	2

 输出结果： 3

采药问题分析

重要数据的引入:

1. M (≤ 100) 种草药的时间与价值2维数组 100×2 , 用 TimeValue 表示。
2. 先引入一个矩阵 (2维数组): 100×1000 的矩阵, 用 MaxValue 表示, 其中 $\text{MaxValue}[i][j]$ 表示前 i 种草药在 **时间 j 以内** 获得的最大价值。
 - 以3种草药, 最大时间值取10为例:

不选任何草药, 价值为0

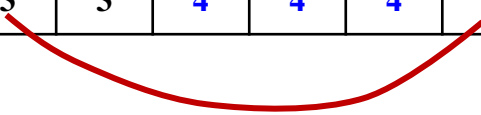
0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0
0										
0										
0										

0时刻价值为0

采药问题分析

假定有3种草药：(10,1)，(5,4)，(3,3)

		0	1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0	0	0
(10,1)	1	0	0	0	0	0	0	0	0	0	0	1
(5,4)	2	0	0	0	0	0	4	4	4	4	4	4
(3,3)	3	0	0	0	3	3	4	4	4	7	7	7



若只有草药1可选，则花10分钟获得价值1；

若只有前2种可选，则只能在1、2之间先选一种，选第2种花5分钟变得到价值4，其后不够10分钟，所以总的价值保持为4；

若有3种可选，第3分钟开始到第5分钟前只能选择第3种，且获得价值为3，从第5分钟开始，到第8分钟之前，选择第2种保持最优，价值为4，如有8分钟或更长，可以同时选2和3，价值为7（即：3+4）

采药问题的规律

		0	1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0	0	0
(10,1)	1	0	0	0	0	0	0	0	0	0	0	1
(5,4)	2	0	0	0	0	0	4	4	4	4	4	4
(3,3)	3	0	0	0	3	3	4	4	4	7	7	7

前 i 种草药在时间 j 以内获得的最大价值

计算 $\text{MaxValue}[i][j]$, 已知第 i 种草药需要的时间 $\text{TimeValue}[i][0]$ 及具有的价值 $\text{TimeValue}[i][1]$

- 不采第 i 种草药, 只在前 $(i-1)$ 种草药中选, 结果为:
 - $\text{MaxValue}[i-1][j]$
- 采第 i 种草药, 剩余时间 $j - \text{TimeValue}[i][0]$ 在前 $(i-1)$ 种草药中选 ($j - \text{TimeValue}[i][0]$ 必须大于0), 结果为:
 - $\text{TimeValue}[i][1] + \text{MaxValue}[i-1][j - \text{TimeValue}[i][0]]$
- 在上述两种情况中取最大值。

采药问题的动态规划实现

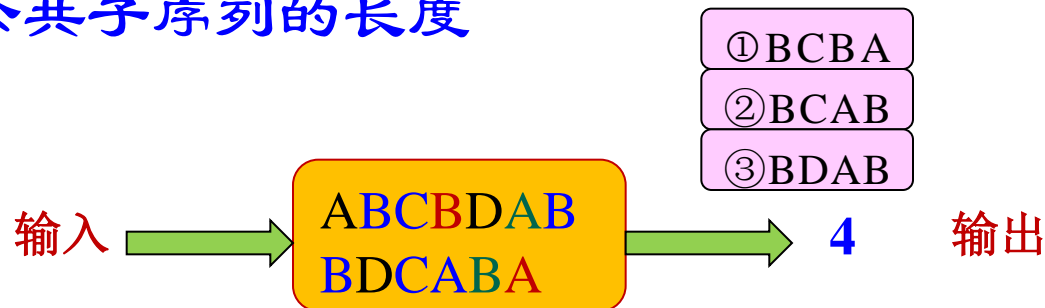
```
int main()
{
    int t,m,i,j;
    int MaxValue[101][1001],TimeValue[101][2];
    cin>>t>>m;
    for(i=1; i<=m; ++i)
        cin>>TimeValue[i][0]>>TimeValue[i][1]; //输入每种药的时间与价值
    for(i=0; i<=t; ++i)
        MaxValue[0][i]=0;
    for(i=1; i<=m; ++i)
        for(j=0; j<=t; ++j)
        {
            MaxValue[i][j]= MaxValue[i-1][j]; //不选当前药材 i
            if(TimeValue[i][0]<=j && MaxValue[i-1][j-TimeValue[i][0]]
+ TimeValue[i][1]> MaxValue[i-1][j])
MaxValue[i][j]=MaxValue[i-1][j-TimeValue[i][0]] + TimeValue[i][1]
        }
    cout<<MaxValue[m][t]<<endl;
}
```

例4：最长公共子序列

- 问题：

- 一个字符数组S为一个序列。对于另外一个字符数组Z,如果满足以下条件，则称Z是S的一个子序列：
(1) Z中的每个元素都是S中的元素 (2) Z中元素的顺序与在S中的顺序一致。例如：当S = (E,R,C,D,F,A,K)时，(E, C, F) 和 (E, R) 都是它的子序列。而 (R, E) 则不是
- 现在我们给定两个序列（两行输入），求它们最长的公共子序列的长度

- 例：



最长公共子序列分析

- 主要数据
 - 两个字符串分别是
 - `char str1[MAXL]`; 实际长度是 `len1`
 - `char str2[MAXL]`; 实际长度是 `len2`
- 公共子串长度关系
 - 设 $f(\text{str1}, \text{len1}, \text{str2}, \text{len2})$ 为 `str1` 和 `str2` 的最大公共子串的长度;
 - **情况1:** $\text{str1}[\text{len1}-1] == \text{str2}[\text{len2}-1]$,
 - $f(\text{str1}, \text{len1}, \text{str2}, \text{len2}) = 1 + f(\text{str1}, \text{len1}-1, \text{str2}, \text{len2}-1)$
 - **情况2:** $\text{str1}[\text{len1}-1] != \text{str2}[\text{len2}-1]$
 - $f(\text{str1}, \text{len1}, \text{str2}, \text{len2}) = \max\{ f(\text{str1}, \text{len1}-1, \text{str2}, \text{len2}), f(\text{str1}, \text{len1}, \text{str2}, \text{len2}-1) \}$

最长公共子序列的递归实现

- 递归函数为: $f(\text{str1}, \text{len1}, \text{str2}, \text{len2})$
- 初始条件:
 - $f(\text{str1}, 0, \text{str2}, \text{len2}) = 0$, $f(\text{str1}, \text{len1}, \text{str2}, 0) = 0$
- 归纳式子:
 - **情况1:** $\text{str1}[\text{len1}-1] == \text{str2}[\text{len2}-1]$,
 - $f(\text{str1}, \text{len1}, \text{str2}, \text{len2}) = 1 + f(\text{str1}, \text{len1}-1, \text{str2}, \text{len2}-1)$
 - **情况2:** $\text{str1}[\text{len1}-1] != \text{str2}[\text{len2}-1]$
 - $f(\text{str1}, \text{len1}, \text{str2}, \text{len2}) = \max\{ f(\text{str1}, \text{len1}-1, \text{str2}, \text{len2}), f(\text{str1}, \text{len1}, \text{str2}, \text{len2}-1) \}$

递归函数

```
int f(char str1[],int len1,char str2[], int len2)
{
    if(len1==0||len2==0) return 0;

    else if (str1[len1-1] == str2[len2-1])
        return 1+ f(str1,len1-1,str2,len2-1);

    else return max(f(str1,len1-1,str2,len2), f(str1,len1,str2,len2-1));
}
```

会有大量重复计算

最长公共子序列动态规划

- 基本思想

- 从str1和str2的第一个字符开始考虑;

- ```
int comLen[MAX][MAX];
```

- 用一个二维数组记录str1的前*i*个字符与str2中的前*j*个字符的最大公共子串长度。

- 设立初值:

- ```
comLen [0][0~MAX]=0; comLen [0~MAX][0]=0; //表示两个字符串中有一个没有参与比较时，最大公共子串长度为 0 .
```

- 递推: **comLen是从1开始有效表示，但2个串均从0开始**

- ```
if (str1[i-1]==str2[j-1]) comLen[i][j] =1+comLen[i-1][j-1];
else comLen[i][j] = max{ comLen[i-1][j], comLen[i][j-1] };
```



# 最长公共子序列动态规划实现

```
#define MAXLEN 1000
#define MAX(x,y) (x>y)?x:y
int commonlen(char str1[], char str2[])
{
 int comLen[MAXLEN][MAXLEN], len1, len2, i, j;
 len1=strlen(str1); len2=strlen(str2);
 for(i=0; i<MAXLEN; ++i)
 comLen[0][i]=comLen[i][0]=0;
 for(i=1; i<=len1; ++i)
 for(j=1; j<=len2; ++j)
 if(str1[i-1]==str2[j-1]) comLen[i][j]=comLen[i-1][j-1]+1;
 else comLen[i][j]=MAX(comLen[i-1][j], comLen[i][j-1]);
 return comLen[len1][len2];
}
```

# 例5：矩阵乘法

- 矩阵乘法：  $A \times B \times C \times D$  最少乘法运算：

$$A = 50 \times 10 \quad B = 10 \times 40 \quad C = 40 \times 30 \quad D = 30 \times 5$$

$(A((BC)D))$     $(A(B(CD)))$     $((AB)(CD))$     $((((AB)C)D)$     $((A(BC))D)$

|                            |                           |                            |                            |                            |
|----------------------------|---------------------------|----------------------------|----------------------------|----------------------------|
| $10 \times 40 \times 30 +$ | $40 \times 30 \times 5 +$ | $50 \times 10 \times 40 +$ | $50 \times 10 \times 40 +$ | $10 \times 40 \times 30 +$ |
| $10 \times 30 \times 5 +$  | $10 \times 40 \times 5 +$ | $40 \times 30 \times 5 +$  | $50 \times 40 \times 30 +$ | $50 \times 10 \times 30 +$ |
| $50 \times 10 \times 5$    | $50 \times 10 \times 5$   | $50 \times 40 \times 5$    | $50 \times 30 \times 5$    | $50 \times 30 \times 5$    |

16000,   **10500**,   36000,   87500,   34500

# 递归计算

- 计算 $A[i:j]$ ,  $1 \leq i \leq j \leq n$  ( $n$ 个矩阵), 所需要的最少数乘次数 $m[i, j]$ , 则原问题的最优值为 $m[1, n]$
- 当 $i=j$ 时,  $A[i:j]=A_i$ , 因此,  $m[i,i]=0$ ,  $i=1,2,\dots,n$
- 当 $i < j$ 时,  
$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$$
  
这里  $A_i$  的维数为  $p_{i-1} \times p_i$

于是,

$$m[i, j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{ m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j \} & i < j \end{cases}$$

$k$  的位置只有  $j - i$  种可能

# 避免重复-动态规划

如何计算？

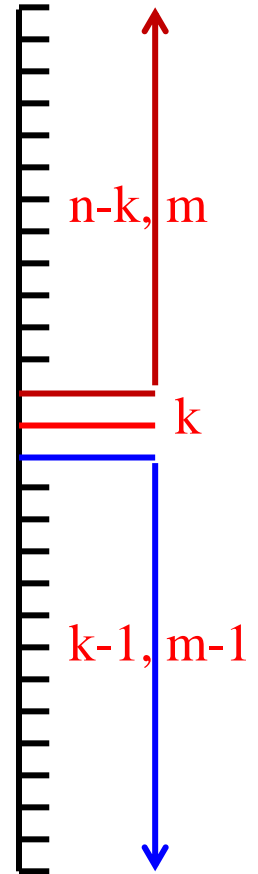
- 递推：
  - 初始：  $m[i,i]=0$
  - 计算：  $m[i,i+1]$
  - 计算：  $m[i,i+2]$
  - ...
  - 计算：  $m[0,n-1]$

# 例6：鸡蛋硬度

- 问题：
  - 从高楼扔鸡蛋来测试鸡蛋的硬度，如果从高楼的第 $a$ 层摔下来没摔破，但是从 $a+1$ 层摔下来时摔破了，那么鸡蛋的硬度是 $a$
  - 假如足够多同样硬度的鸡蛋，那么可以用二分的方法用最少的次数测出鸡蛋的硬度
  - 假如鸡蛋不够多，比如只有1个鸡蛋，就不得不从第1楼开始一层一层地扔，直至到第100层
  - 如果有 $n$ 层楼 $m$ 个鸡蛋（鸡蛋硬度相同），计算使用最优策略在**最坏情况下**所需要扔鸡蛋的次数
  - **设：  $n$ 不大于100，  $m$ 不大于10**

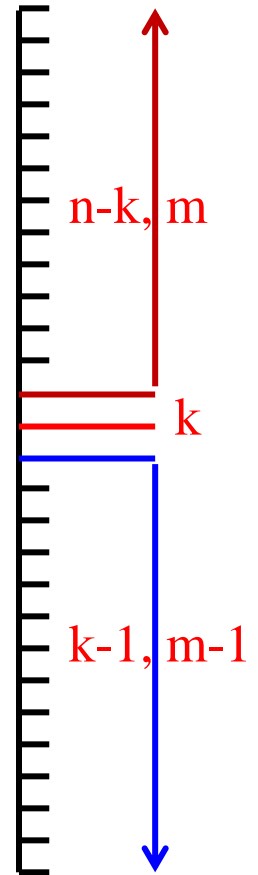
# 鸡蛋硬度分析

- 分析：
  - 用  $f(n, m)$  表示用  $m$  个鸡蛋测试  $n$  层楼 **最坏情况下的最少次数**；
  - 先枚举第1次扔鸡蛋的楼层，假设在第  $k$  层，有2种情况：
    - ① 鸡蛋碎了，说明硬度小于  $k$ ，之后还剩下  $m-1$  枚鸡蛋，在下面的  $k-1$  层测试，最坏情况下的最少次数为：  $f(k-1, m-1)$ ；
    - ② 鸡蛋完好，说明硬度大于或等于  $k$ ，在上面的  $n-k$  层测试。此时还剩下  $m$  枚鸡蛋，最坏情况下最少次数为：  $f(n-k, m)$ ；



# 鸡蛋硬度: 动态规划

- 分析:
  - 第k层扔鸡蛋, 最坏情况下的最小次数为:  
 $1 + \max\{f(k-1, m-1), f(n-k, m)\}$
  - 从1到n枚举k, 从中选择最坏情况下的最好方案:  $f(n, m) = \min\{1 + \max\{f(k-1, m-1), f(n-k, m)\} \mid k=1 \dots n\}$
  - 有2个边界条件:
    - ①  $n=0$ (第0层),  $f(0, m)=0$
    - ②  $m=1$ (只有1个鸡蛋),  $f(n, 1)=n$  //最坏情况n次 (从第1楼依次扔到第n楼)



# 鸡蛋硬度: 动态规划

- 实例分析:

- 1层楼, **2个鸡蛋**:  $f(1,2)=1$ 
  - 只有1次尝试 (在第1层扔1次)

- 2层楼, **2个鸡蛋**:  $f(2,2)=2$

- 第1次尝试: 从第1层扔  $1+\text{Max}\{f(0,1), \mathbf{f(1,2)}\}=2$

- 第2次尝试: 从第2层扔  $1+\text{Max}\{f(1,1), \mathbf{f(0,2)}\}=2$

- 3层楼, **2个鸡蛋**:  $f(3,2)=2$

- 第1次尝试: 从第1层开始扔  $1+\text{Max}\{f(0,1), \mathbf{f(2,2)}\}=3$

- **第2次尝试: 从第2层开始扔**  $1+\text{Max}\{f(1,1), \mathbf{f(1,2)}\}=2$

- 第3次尝试: 从第3层开始扔  $1+\text{Max}\{\mathbf{f(2,1)}, f(0,2)\}=3$

鸡蛋

|   | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 2 | 0 | 2 | 2 | 2 |
| 3 | 0 | 3 | 2 | 2 |
| 4 | 0 | 4 | 3 | 3 |
| 5 | 0 | 5 | 3 | 3 |

楼层

取最小值

2楼之上没楼层了

取最小值



# 鸡蛋硬度: 动态规划 (核心)

对于给定的鸡蛋数 eggs

```
for(i=1; i<=level; ++i)
```

```
{
```

```
 num=1+Max(f[i-1][eggs-1], f[level-i][eggs]);
```

```
 if(num<f[level][eggs]
```

```
 f[level][eggs]=num;
```

```
}
```

} 取最小值

```
#define MAX 100000000
```

```
void calc()
{ int current;
 for (int eggs = 1; eggs <= 10; eggs++) //鸡蛋数
 for (int level = 0; level <= 100; level++) //楼层数
 if (eggs == 1) // 只有一个鸡蛋的情况, 边界条件
 f[level][eggs] = level; //最坏的情况是有多层试多少次
 else if (level == 0) // 0层楼的情况, 边界条件
 f[level][eggs] = 0; //没有楼层的情况, 一次都不用试
 else //鸡蛋数大于1, 楼层大于0的情况
 { f[level][eggs] = MAX; //先假设一个最大值
 for (int try1 = 1; try1 <= level; ++try1)
 //枚举第一次尝试的层次
 {
 current = 1 + max(f[try1 - 1][eggs - 1],
 //鸡蛋碎了的情况
 f[level - try1][eggs]); //鸡蛋没有碎的情况
 if (current < f[level][eggs])
 f[level][eggs] = current;
 }
 }
}
```

# 动态规划的关键点

- 正确写出基本**递推关系**和恰当的**初始条件**（状态转移方程）
- 将问题的过程分解成**相互联系的几个阶段**，选取状态变量和决策变量并定义最优值函数，将一个大问题转化成同类型的子问题再求解
- 从初始条件出发，逐段递进寻优，在求解每一子问题的过程中均利用**前面子问题的最优结果**，依次进行，最后有一个子问题的最优解便为整体最优解
- 在多阶段决策过程中，既将**当前段与未来段分开**，又将**当前利益与未来利益结合**考虑，因此，每段决策的选取是整体考虑的

# 动态规划的关键点

- 在求解问题最优策略时，初始状态是一致的，每阶段的决策都是该阶段状态的函数，于是，**最优策略所经过的各阶段便可逐步变换**得到，从而确定最优路线
- 每个阶段的最优决策过程只与本阶段的初始状态相关，与以前各阶段的决策无关
- 本阶段之前的状态与决策仅影响本阶段的初始状态
- 上述性质也称为**无后效性**（**马尔可夫性**）
- 动态规划法只适合求解**具有无后效性状态**的多阶段决策问题

# 动态规划的特点与实现

- 特点
  - 递归向递推（迭代）的转换
    - 递归关系式一致
  - 避免重复计算提高计算效率
  - 以空间换时间
- 动态规划 & 递归
  - 动态规划：最优解（多种解中最优解），通常用递推
  - 何时用递归：不涉及最优要求时