



Part5：动态规划算法

- 动态规划算法思想
- 三个基本例子
- 背包问题
- 最优二叉树

回顾：分治法

分治法的基本思想

- ① 将规模较大的问题分解为规模较小的问题
- ② 解决这些小问题
- ③ 然后将小问题的解合并成为原来大问题的解

分治法的基础

- ① 小问题比大问题更容易解决
- ② 将小问题的解组装成大问题的解比直接求解大问题成本更低
- ③ 小问题又可以按照同样的方法分解为更小的问题，便于将问题的规模进一步缩小

分治法面临的问题

问题：大量重复的小问题

- ① 在分解原问题为小问题的过程中，有时会产生大量重复的小问题
- ② 对这些重复问题的计算，产生了巨大的时间开销。

例子：Fibonacci序列问题

$$f(n) = \begin{cases} 1 & n = 1, 2 \\ f(n - 1) + f(n - 2) & n \geq 3 \end{cases}$$

可用分治法解决该问题

分治法面临的问题

可用分治法解决该问题

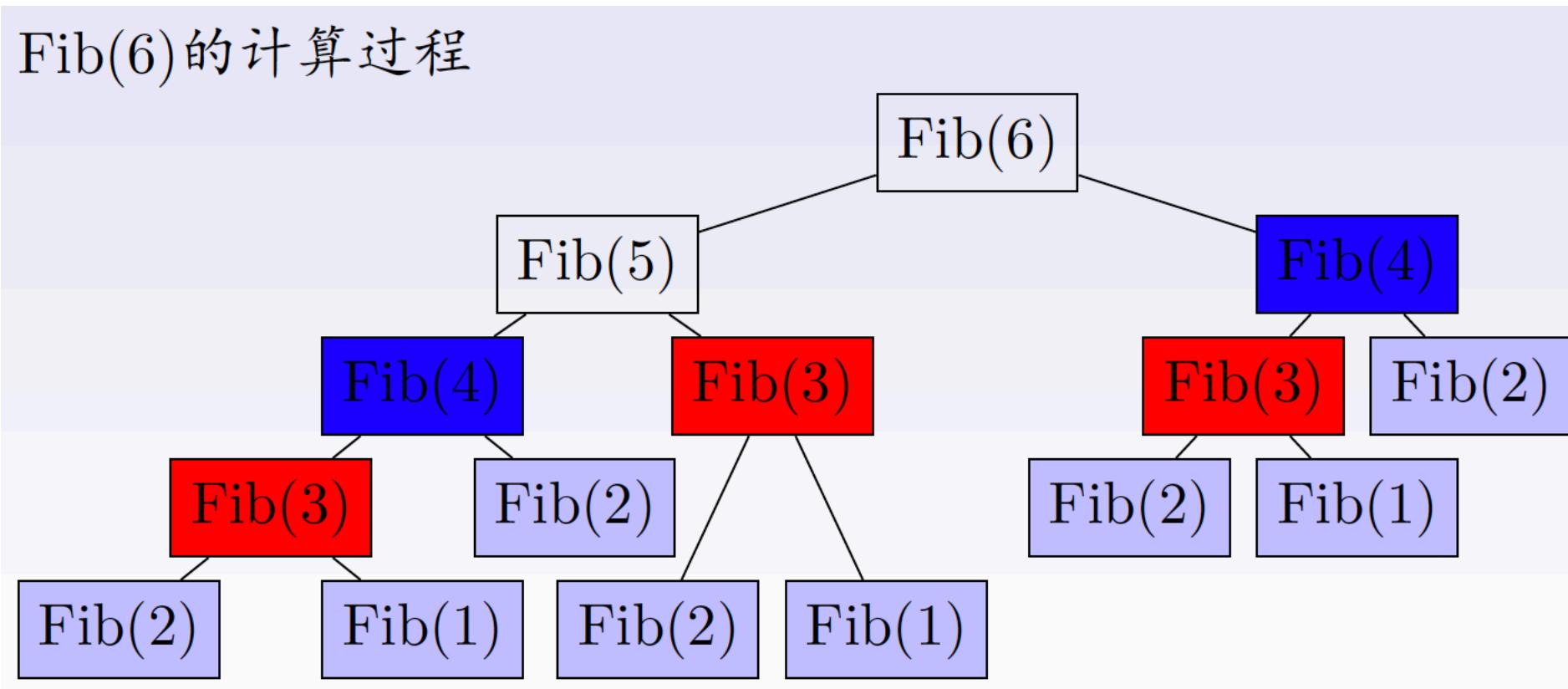
- ① 规模为 n 的问题 $f(n)$ 被划分为规模更小的两个子问题 $f(n - 1)$ 和 $f(n - 2)$
- ② 求解这两个子问题的解之后，通过简单组合(加法操作)，即可求原问题解
- ③ 子问题可以采用同样地方式进一步缩减规模(直至为1或2)

Fibonacci(n)

```
if n == 1 or n == 2 then
    return n
else
    return Fibonacci(n-1) + Fibonacci(n-2)
end if
```

分治法面临的问题

Fib(6)的计算过程



Fib(6)的问题

- ① Fib(4)计算了两次，Fib(3)计算了三次
- ② 采用分治法计算Fib(n)，n越大，分解出的相同子问题就越多

分治法面临的问题

定义 (交叠子问题)

在划分的一组子问题中，存在若干或大量的雷同子问题，它们称为交叠子问题

解决办法：避免相同子问题的重复计算

- ① 对子问题只求解一次，并把它的解记录在表中。
- ② 当再次需要求解该子问题时，直接从记录表中查询获取它的解

基于动态规划的Fibonacci序列算法伪代码

```
Fibonacci(n)
```

```
    Fibonacci(1) = 1;
```

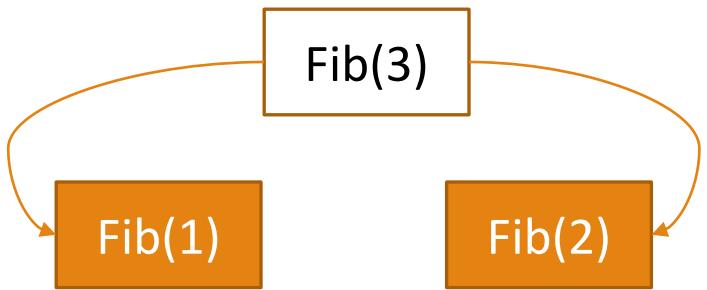
```
    Fibonacci(2) = 2;
```

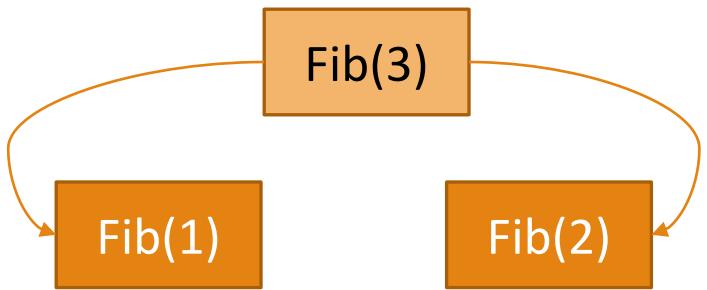
```
    for i = 3 to n do
```

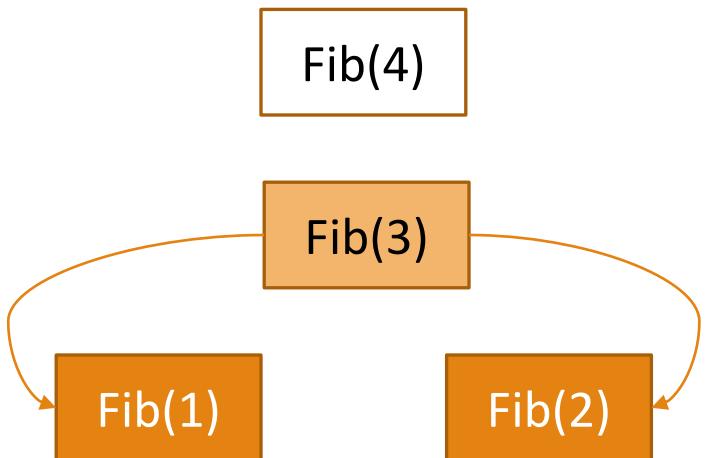
```
        Fibonacci(i) = Fibonacci(i-1) + Fibonacci(i-2)
```

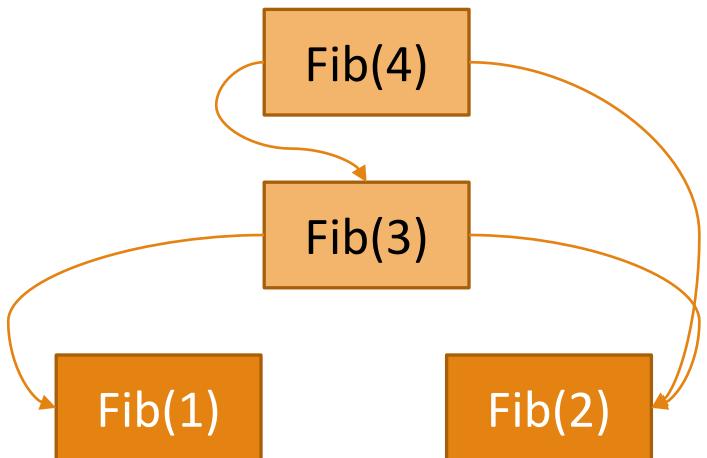
```
    end for
```

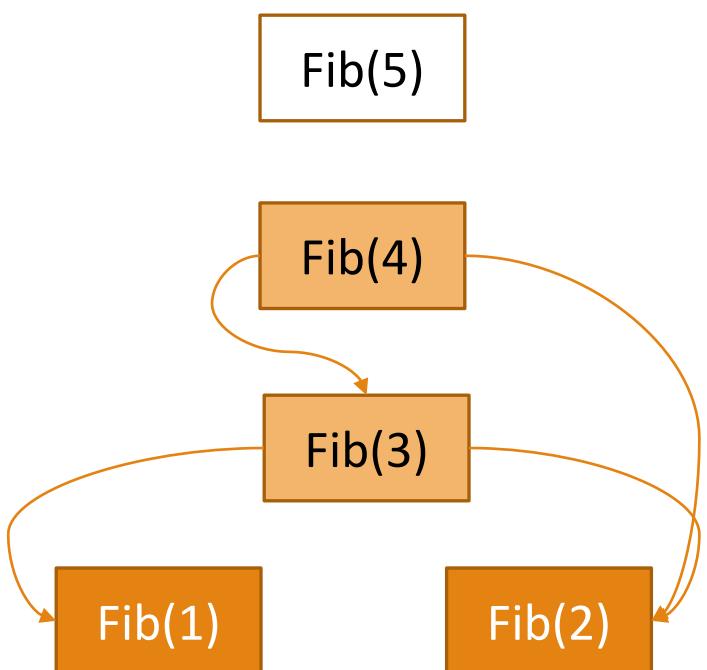
```
    return Fibonacci(n)
```

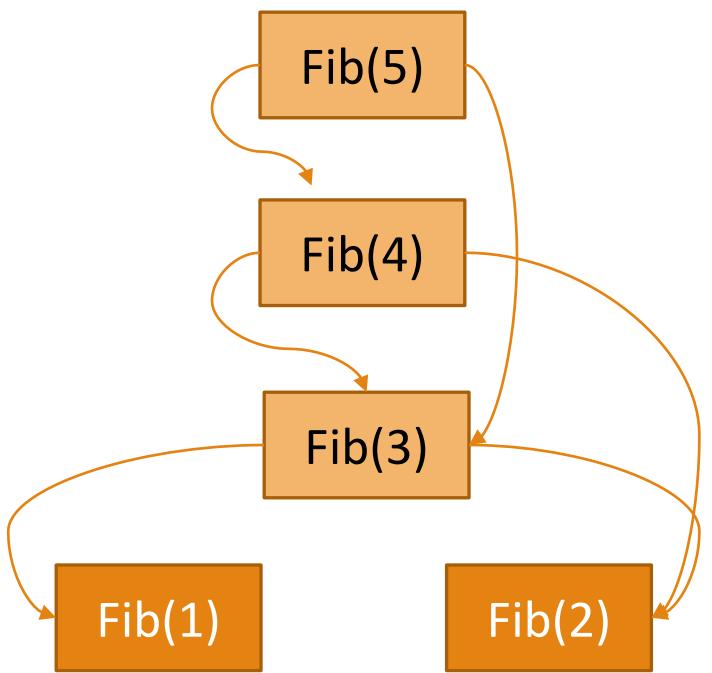


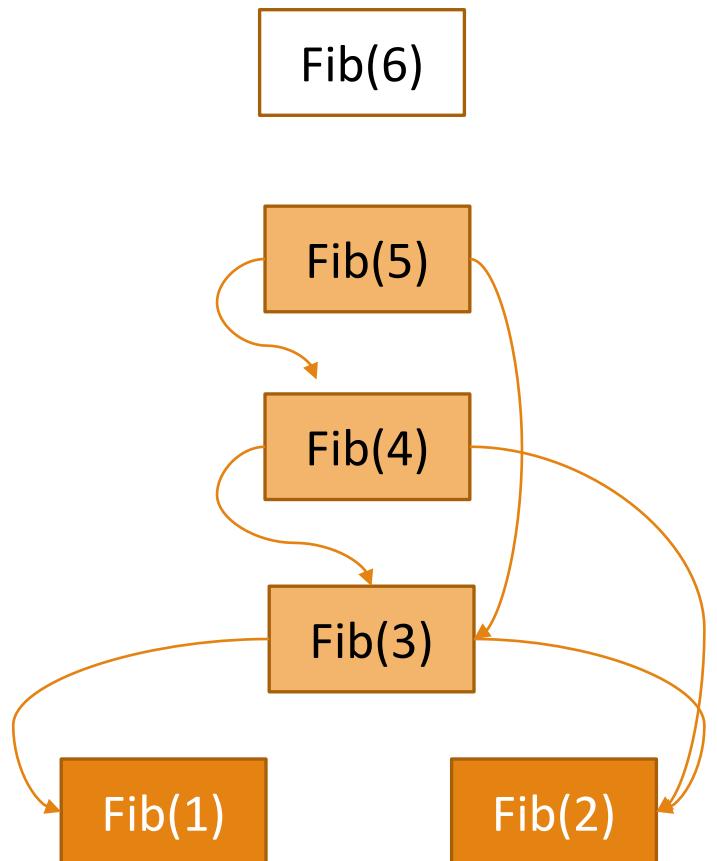


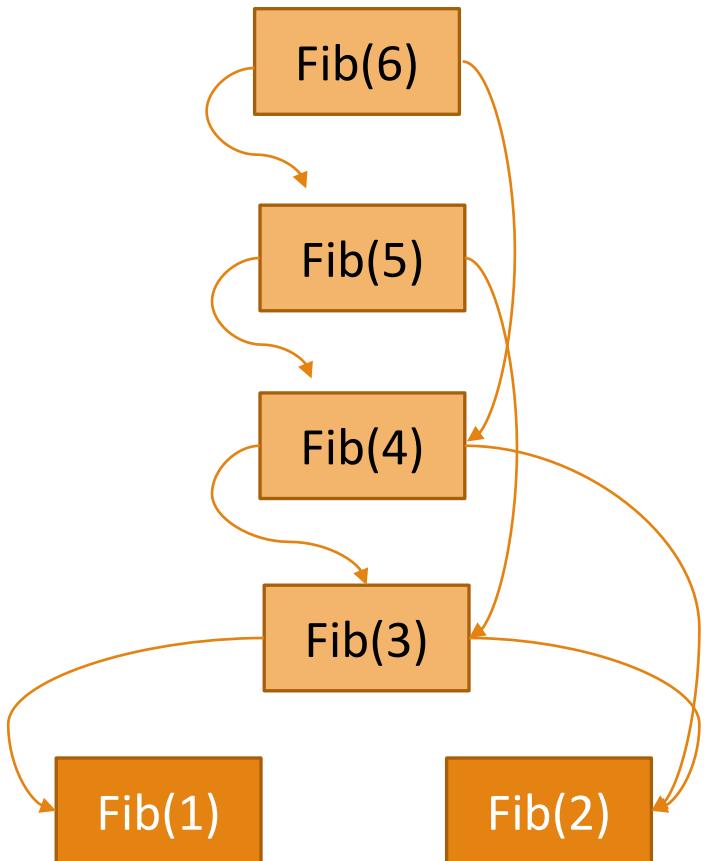












动态规划与分治思想比较

动态规划与分治思想的比较

- ① 二者思想类似：将大问题划分为若干小问题
- ② 分治算法是采用自上而下的方式求值，导致了不止一次的递归调用
- ③ 动态规划法是采取自下向上的方式递推求值，并把中间结果存储起来，避免重复运算，从而降低时间复杂度。

动态规划的主要目的

动态规划提出来的主要目的是优化，即不只是解决一个问题，而是以最优的方式解决这个问题，或者说，针对特定问题寻求最优解。

发明者

动态规划由理查德·贝尔曼(Richard E. Bellman)于1957年在其著作《动态规划 (Dynamic Programming)》一书中提出。



Part5：动态规划算法

- 动态规划算法思想
- **三个基本例子**
- 背包问题
- 最优二叉树

例子1：币值最大化问题

问题

给定一排硬币（ n 个），其面值均为正整数 c_1, c_2, \dots, c_n 。如何选择硬币，使得在其原位置互不相邻的条件下，所选硬币的总金额最大。

实例

coin's index(i)	0	1	2	3	4	5	6
coin's value(c_i)	-	5	1	2	10	6	2
F	0	5					



基于递归分治的币值最大化问题算法

选择方法的表示

假设 $F(n)$ 表示在 n 个可选硬币下，所选硬币的最大金额，则

$$F(n) = \begin{cases} \max\{c_n + F(n - 2), F(n - 1)\}, & n > 1 \\ c_1, & n = 1 \\ 0, & n = 0 \end{cases}$$

基于递归分治的币值最大化问题算法

F(n)

数组C[1..n]保存n个硬币的面值

if n=1 **then**

return C₁

else if n=0 **then**

return 0

end if

if n ≥ 2 **then**

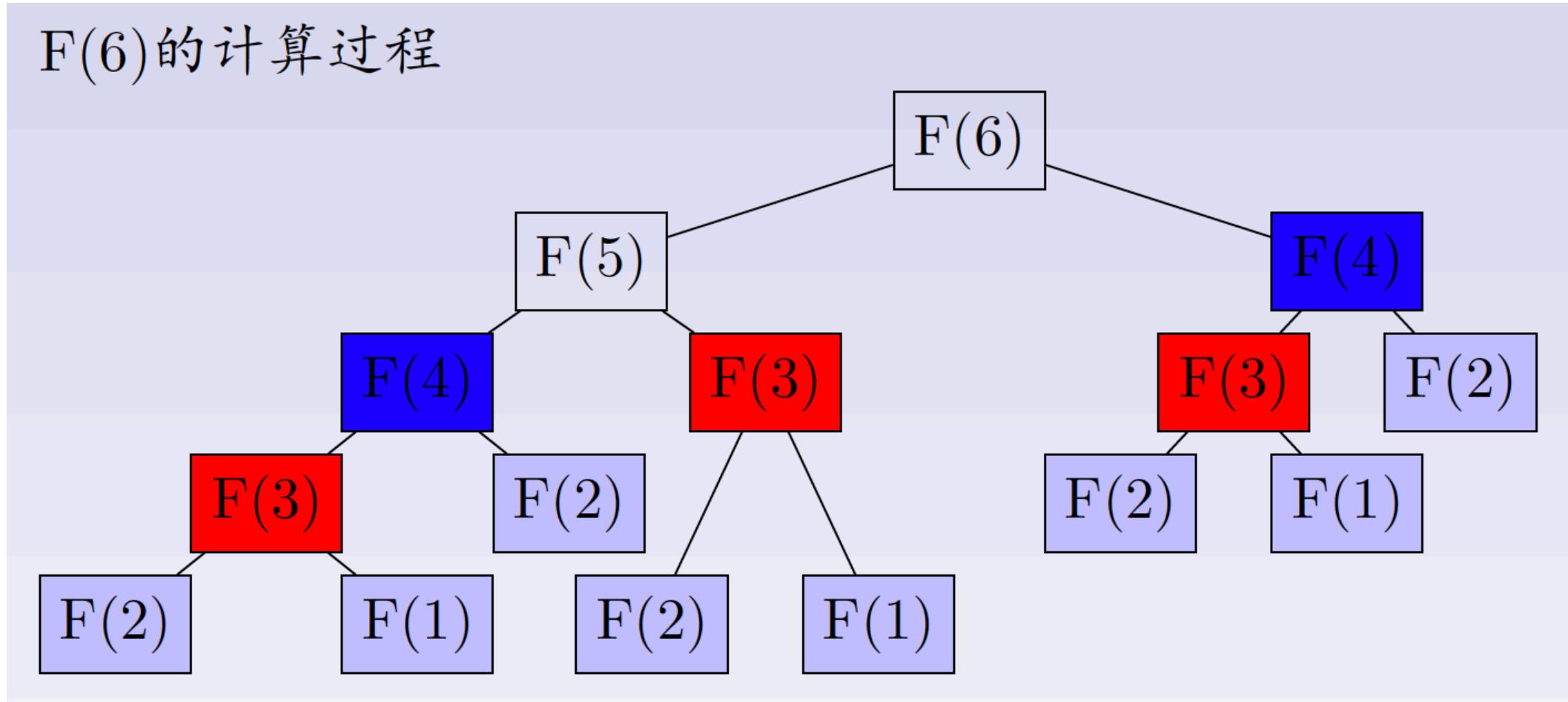
$F(n) = \max\{C_n + F(n - 2), F(n - 1)\}$

end if

return F(n)

基于递归分治的币值最大化问题算法

F(6)的计算过程



问题

存在大量的相同子问题，导致大量的重复计算

币值最大问题

coin's index(i)	0	1	2	3	4	5	6
coin's value(c_i)	-	5	1	2	10	6	2

$$F(n) = \max\{C_n + F(n - 2), F(n - 1)\}, n > 1$$

$$F(0) = 0, F(1) = 5$$

自下而上求解:动态规划的币值最大问题求解方式

$$F(2) = \max\{C_2 + F(2 - 2), F(2 - 1)\}$$

币值最大问题

coin's index(i)	0	1	2	3	4	5	6
coin's value(c_i)	-	5	1	2	10	6	2

$$F(n) = \max\{C_n + F(n - 2), F(n - 1)\}, n > 1$$

$$F(0) = 0, F(1) = 5$$

自下而上求解:动态规划的币值最大问题求解方式

$$\begin{aligned}F(2) &= \max\{C_2 + F(2 - 2), F(2 - 1)\} \\&= \max\{1 + F(0), F(1)\}\end{aligned}$$

币值最大问题

coin's index(i)	0	1	2	3	4	5	6
coin's value(c_i)	-	5	1	2	10	6	2

$$F(n) = \max\{C_n + F(n - 2), F(n - 1)\}, n > 1$$

$$F(0) = 0, F(1) = 5$$

自下而上求解:动态规划的币值最大问题求解方式

$$\begin{aligned}F(2) &= \max\{C_2 + F(2 - 2), F(2 - 1)\} \\&= \max\{1 + F(0), F(1)\} \\&= \max\{1, 5\} = 5\end{aligned}$$

币值最大问题

coin's index(i)	0	1	2	3	4	5	6
coin's value(c_i)	-	5	1	2	10	6	2

$$F(n) = \max\{C_n + F(n - 2), F(n - 1)\}, n > 1$$

$$F(0) = 0, F(1) = 5$$

自下而上求解:动态规划的币值最大问题求解方式

$$F(2) = \max\{C_2 + F(2 - 2), F(2 - 1)\}$$

$$= \max\{1 + F(0), F(1)\}$$

$$= \max\{1, 5\} = 5$$

$$F(3) = \max\{C_3 + F(3 - 2), F(3 - 1)\}$$

币值最大问题

coin's index(i)	0	1	2	3	4	5	6
coin's value(c_i)	-	5	1	2	10	6	2

$$F(n) = \max\{C_n + F(n - 2), F(n - 1)\}, n > 1$$

$$F(0) = 0, F(1) = 5$$

自下而上求解:动态规划的币值最大问题求解方式

$$\begin{aligned} F(2) &= \max\{C_2 + F(2 - 2), F(2 - 1)\} \\ &= \max\{1 + F(0), F(1)\} \\ &= \max\{1, 5\} = 5 \end{aligned}$$

$$\begin{aligned} F(3) &= \max\{C_3 + F(3 - 2), F(3 - 1)\} \\ &= \max\{2 + F(1), F(2)\} \end{aligned}$$

币值最大问题

coin's index(i)	0	1	2	3	4	5	6
coin's value(c_i)	-	5	1	2	10	6	2

$$F(n) = \max\{C_n + F(n - 2), F(n - 1)\}, n > 1$$

$$F(0) = 0, F(1) = 5$$

自下而上求解:动态规划的币值最大问题求解方式

$$F(2) = \max\{C_2 + F(2 - 2), F(2 - 1)\}$$

$$= \max\{1 + F(0), F(1)\}$$

$$= \max\{1, 5\} = 5$$

$$F(3) = \max\{C_3 + F(3 - 2), F(3 - 1)\}$$

$$= \max\{2 + F(1), F(2)\}$$

$$F(4) = \max\{C_4 + F(4 - 2), F(4 - 1)\}$$

$$= \max\{10 + F(2), F(3)\} = \max\{15, 7\} = 15$$

币值最大问题

coin's index(i)	0	1	2	3	4	5	6
coin's value(c_i)	-	5	1	2	10	6	2

$$F(n) = \max\{C_n + F(n - 2), F(n - 1)\}, n > 1$$

$$F(0) = 0, F(1) = 5$$

自下而上求解:动态规划的币值最大问题求解方式(续)

$$F(5) = \max\{C_5 + F(5 - 2), F(5 - 1)\}$$

$$= \max\{6 + F(3), F(4)\}$$

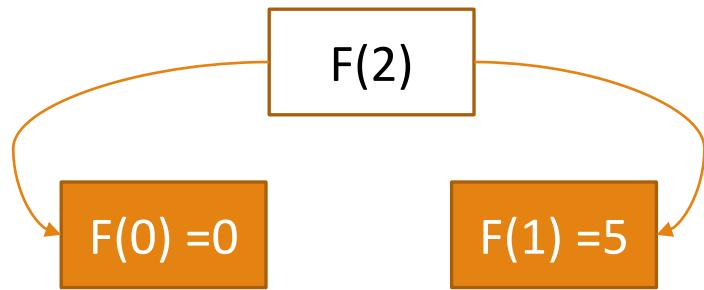
$$= \max\{13, 15\} = 15$$

$$F(6) = \max\{C_6 + F(6 - 2), F(6 - 1)\}$$

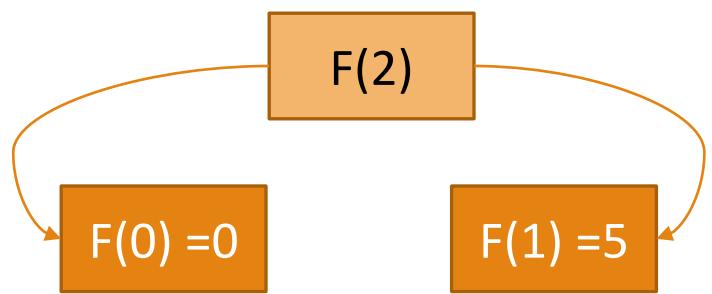
$$= \max\{2 + F(4), F(5)\}$$

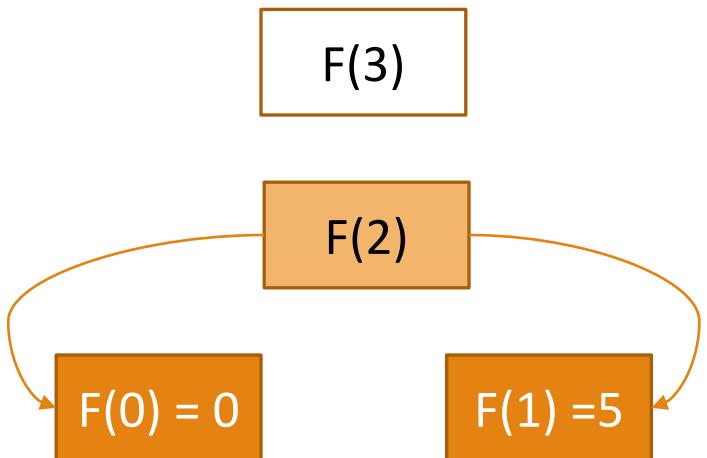
$$= \max\{17, 15\} = 17$$

自下而上求解过程

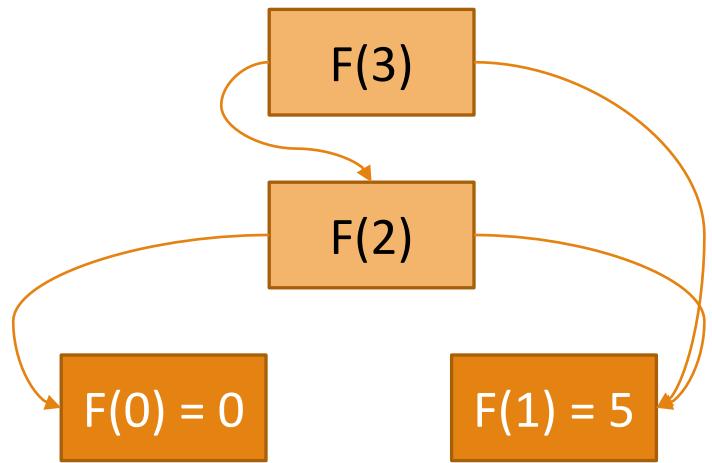


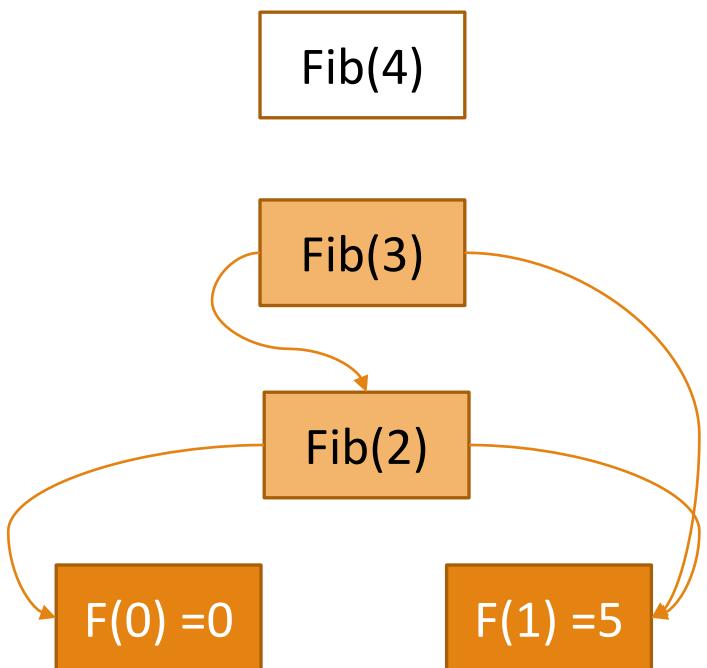
自下而上求解过程

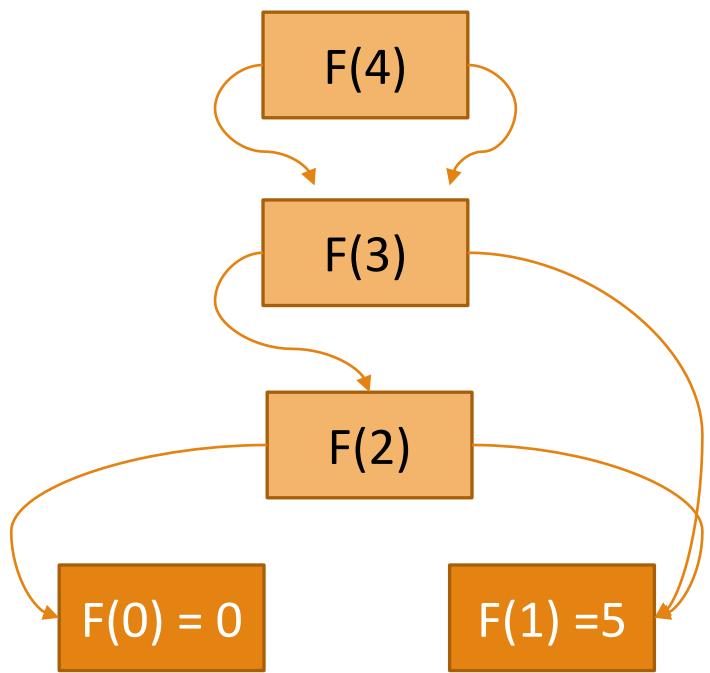


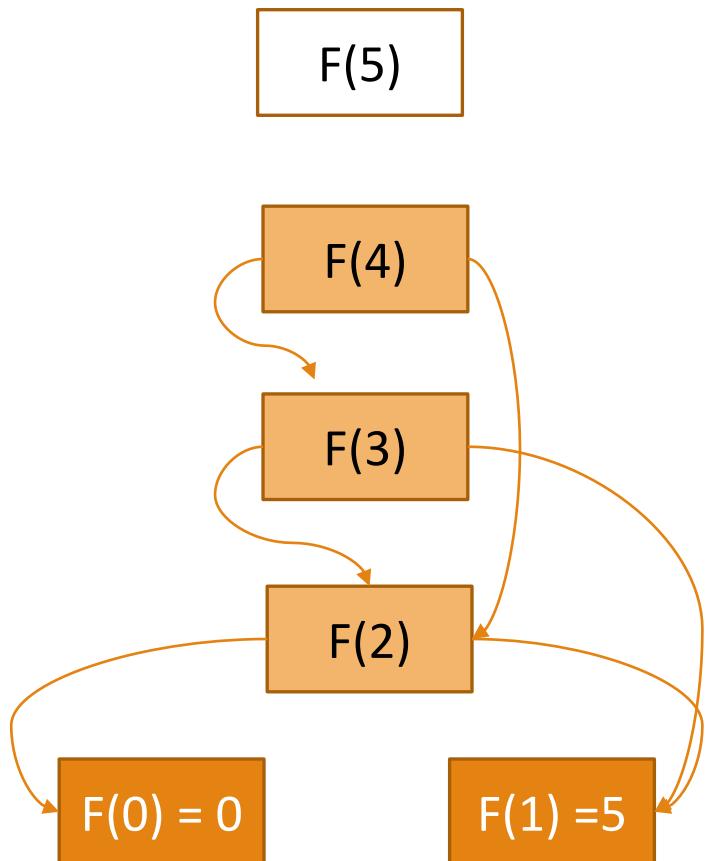


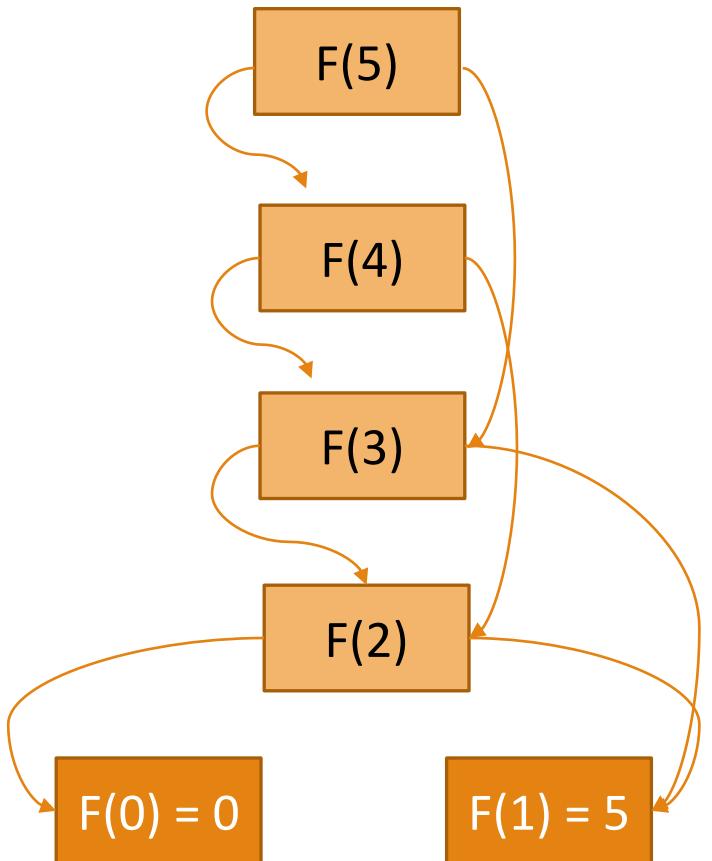
自下而上求解过程











基于动态规划的币值最大问题算法

CoinRow(C[1…n])

//算法基础: $F(n) = \max\{C_n + F(n - 2), F(n - 1)\}$ $n > 1$

//输入: C[1…n]表示n个硬币及其对应的值

//输出: 能选取的最大币值

$F[0] = 0; F[1] = c_1$

for $i = 2$ to n **do**

$F[i] = \max\{C[i] + F[i - 2], F[i - 1]\}$

end for

return $F[n]$

例子2：找零问题

找零问题

有m种规格的硬币，其面值依次为： $d_1 < d_2 < \dots < d_m$ 。现需要找零给客户，其找零金额为n，请问最少需要多少个硬币？

实例

假设有3个硬币，它们的面值依次为1, 3, 6

找零金额n	0	1	2	3	4	5	6
所需硬币数F	-						

找零问题

假设 $F(n)$ 表示找零金额为 n 所需的最小硬币数

- 如果第一个硬币被用于找零，则 $F(n) = F(n-d_1) + 1$
- 如果第二个硬币被用于找零，则 $F(n) = F(n-d_2) + 1$
- 如果第三个硬币被用于找零，则 $F(n) = F(n-d_3) + 1$

- 比较这三种选择，最优选择应该如下：

$$F(n) = \min(F(n - d1) + 1, F(n - d2) + 1, F(n - d3) + 1)$$



找零问题

等式扩展

上述等式的前提是只有3个硬币可用于找零。若有m个硬币可用于找零，则上述等式可表示如下，

$$F(n) = \begin{cases} \min_{n \geq d_i} \{F(n - d_i) + 1\}, & n > 1, 1 \leq i \leq m \\ 0, & n = 0 \end{cases}$$



找零问题

实例

假设有3个硬币，它们的面值依次为1, 3, 4，现需要找零6，最少需要多少硬币？

$$F(n) = \begin{cases} \min_{n \geq d_i} \{F(n - d_i) + 1\}, & n > 1, 1 \leq i \leq m \\ 0, & n = 0 \end{cases}$$



找零问题

自下向上：基于动态规划的计算过程

- $F(1) = \min\{F(1 - d_1) + 1\} = \min\{F(0) + 1\} = 1$
- $F(2) = \min\{F(2-d_1)+1\} = \min\{F(1)+1\} = 2$
- $F(3) = \min\{F(3-d_1)+1, F(3-d_2)+1\} = \min\{F(2)+1, F(0)+1\} = 1$
- $F(4) = \min\{F(4-d_1)+1, F(4-d_2)+1, F(4-d_3)+1\} = \min\{F(3)+1, F(1)+1, F(0)+1\} = 1$

找零问题

实例

假设有3个硬币，它们的面值依次为1, 3, 4, 现需要找零6，最少需要多少硬币

$$F(n) = \begin{cases} \min_{n \geq d_i} \{F(n - d_i) + 1\}, & n > 1, 1 \leq i \leq m \\ 0, & n = 0 \end{cases}$$

找零金额n	0	1	2	3	4	5	6
所需硬币数F	-	1	2	1	1		

找零问题

实例

假设有3个硬币，它们的面值依次为1, 3, 4, 现需要找零6，最少需要多少硬币

$$F(n) = \begin{cases} \min_{n \geq d_i} \{F(n - d_i) + 1\}, & n > 1, 1 \leq i \leq m \\ 0, & n = 0 \end{cases}$$

找零金额n	0	1	2	3	4	5	6
所需硬币数F	-	1	2	1	1		



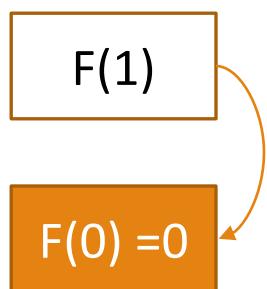
找零问题

自下向上：基于动态规划的计算过程(续)

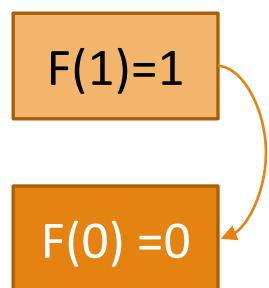
- $F(5) = \min\{F(5-d_1)+1, F(5-d_2)+1, F(5-d_3)+1\} = \min\{F(4)+1, F(2)+1, F(1)+1\} = 2$
- $F(6) = \min\{F(6-d_1)+1, F(6-d_2)+1, F(6-d_3)+1\} = \min\{F(5)+1, F(3)+1, F(2)+1\} = 2$



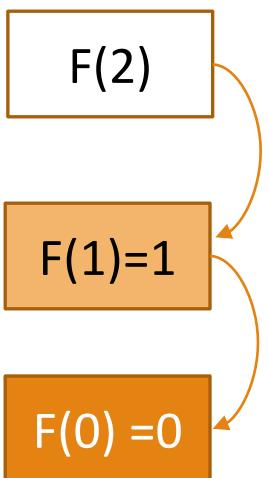
自下而上求解过程



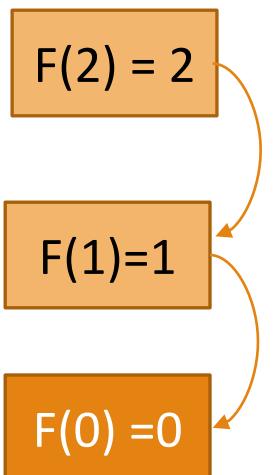
自下而上求解过程



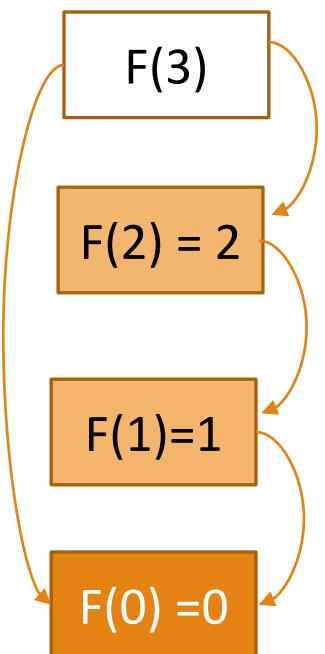
自下而上求解过程



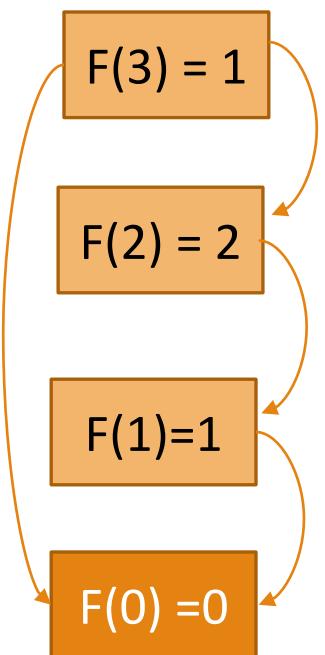
自下而上求解过程



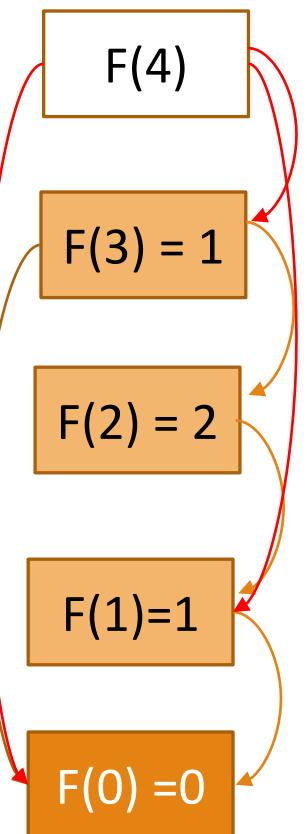
自下而上求解过程



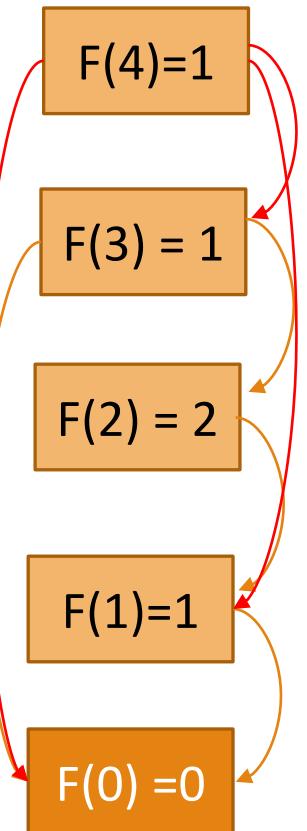
自下而上求解过程



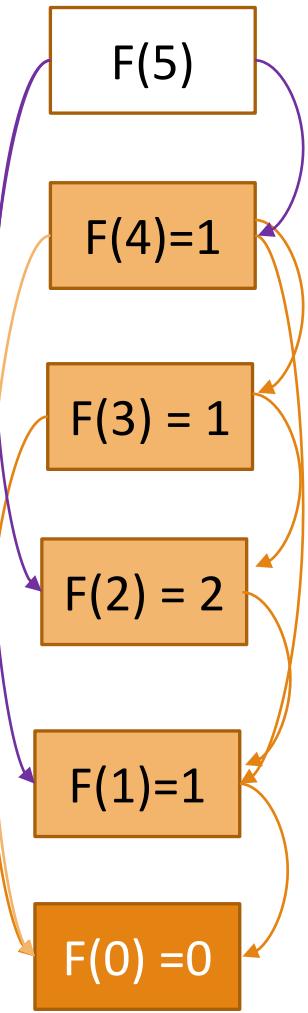
自下而上求解过程



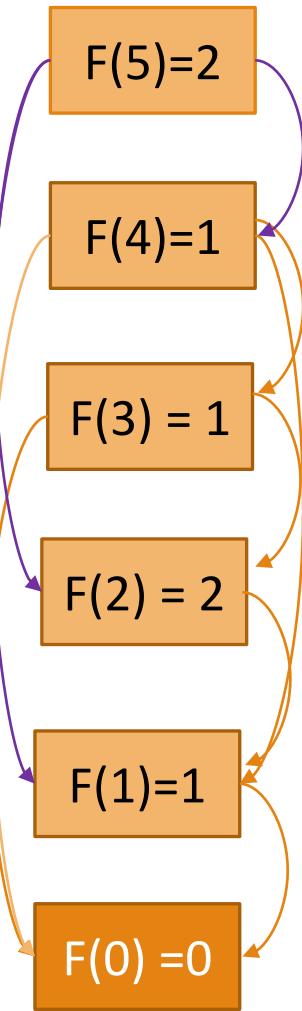
自下而上求解过程



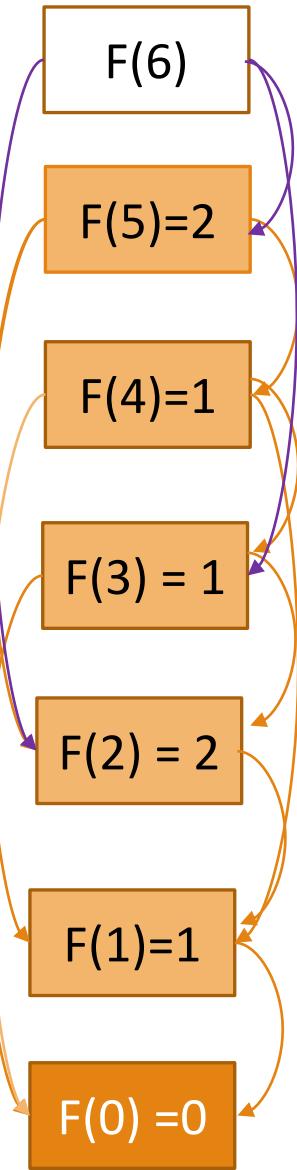
自下而上求解过程



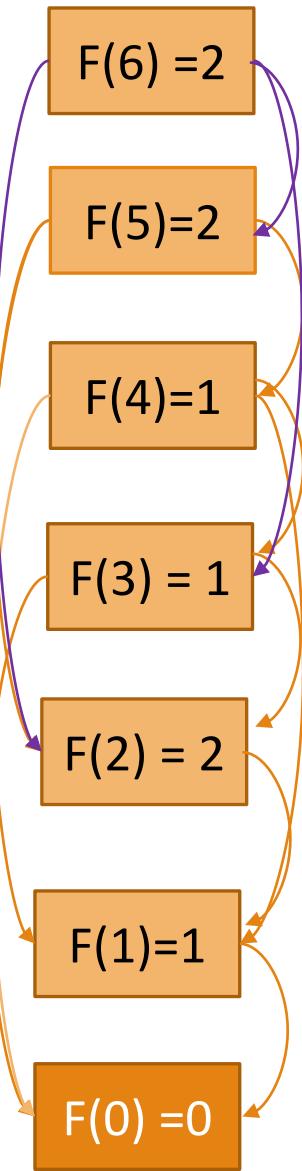
自下而上求解过程



自下而上求解过程



自下而上求解过程



基于动态规划的找零算法

```
ChangeMaking(C[1· · · m], n)
```

```
//算法基础:  $F(n) = \min_{n \geq d_i} \{F(n - d_i) + 1\} = \min_{n \geq d_i} \{F(n - d_i)\} + 1$ 
```

```
//输入: C[1· · · m]表示用于找零的m个硬币及其对应的值
```

```
//输入: n表示找零金额
```

```
//输出: 最少的用币数
```

```
F[0] = 0
```

```
for i = 1 to n do
```

```
    temp =  $\infty$ ; j = 1
```

```
    while j ≤ m and i ≥ C[j] do
```

```
        temp = min(F(i - C[j]), temp)
```

```
        j = j + 1
```

```
    end while
```

```
    F[i] = temp + 1
```

```
end for
```

```
return F[n]
```

例子3：硬币收集问题

问题描述

$n * m$ 的格木板中放有一些硬币，每格的硬币数目最多一个。机器人从木板的左上方移动到右下方（每次只能移动一个格子，并且不能后退），遇到有硬币的单元格就会收集硬币。设计算法找出机器人能够收集最大硬币数的路径。

例子3：硬币收集问题

递归关系

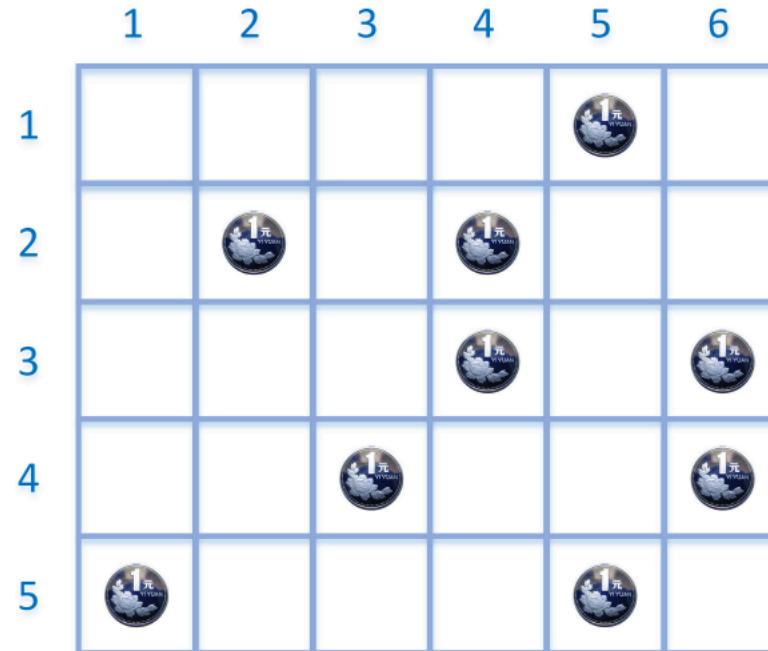
- $F(i, j)$ 为机器人截止到单元格 (i, j) 能够收集到的最大硬币数；
- 第一行上方和第一列左边的单元格，假定其硬币数为0；
- 递推关系：

$$F(i, j) = \max\{F(i - 1, j), F(i, j - 1)\} + c_{ij}, 1 \leq i \leq n, 1 \leq j \leq m$$

$$F(0, j) = 0, 1 \leq j \leq m; \quad F(i, 0) = 0, 1 \leq i \leq n$$

例子3：硬币收集问题——实例

问题



动态规划计算

1	2	3	4	5	6	
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	5

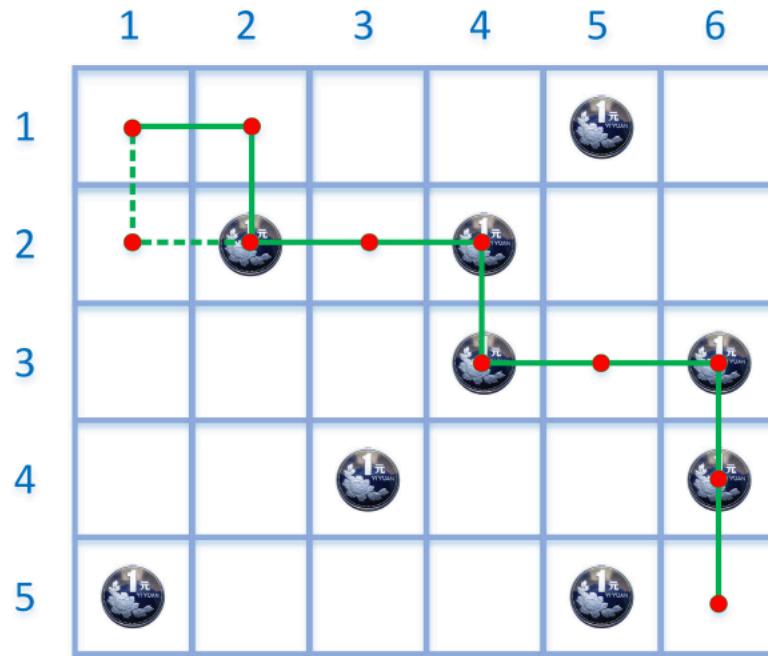
例子3：硬币收集问题——实例

回溯过程

1	2	3	4	5	6
1	0	0	0	1	1
2	0	1	1	2	2
3	0	1	1	3	3
4	0	1	2	3	3
5	1	1	2	3	4

Diagram illustrating the backtracking process for the coin collection problem. The grid shows the minimum number of coins required to reach each cell. Red numbers indicate the current state, while green arrows show the path taken to reach the final solution.

最优路径





Part5：动态规划算法

- 动态规划算法思想
- 三个基本例子
- **背包问题**
- 最优二叉树

背包问题

问题回顾

- 求能够
- 穷举查
- 贪心算

背包问题

一个背包容量为C，n个物品的重量和价值分别如下：

重量	w ₁	w ₂	…	w _n
价值	v ₁	v ₂	…	v _n

要求使背包所装物品价值最大化，即

$$\text{Maximize} \sum_{i=1}^n v_i x_i$$

自然约束条件：

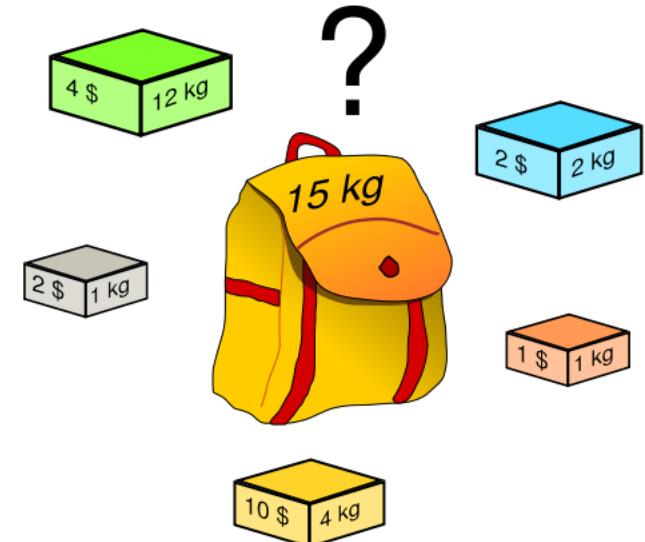
$$\sum_{i=1}^n w_i x_i \leq C$$

$$x_i = 0, 1, \quad 1 \leq i \leq n$$

大价值的物品集合；

计算复杂度较高 2^n ；

近似解；



背包问题——实例

背包实例

假设有4个物品： $U=\{u_1, u_2, u_3, u_4\}$

它们的重量为： $W=\{w_1, w_2, w_3, w_4\} = \{2, 3, 4, 5\}$

它们的价值为： $V=\{v_1, v_2, v_3, v_4\} = \{3, 4, 5, 7\}$

背包的总容量： $C=9$

背包所装物品的最大价值是多少？

背包问题——实例

穷举法

- 对于某一物品而言，或者装入或者不装入，只有两种选择，可以0或1表示。
- n 个物品，相当于构成一个具有 n 位的二进制数，则共有 2^n 个装入方案。
- 需要从 2^n 个方案中找出最佳值，故穷举法的时间复杂性为 $\Theta(2^n)$ 。

背包问题——动态规划算法

动态规划法之背包问题的定义

- 设 U_i 是物品集 U 的子集，由 U 的前*i*项 $\{u_1, u_2, \dots, u_i\}$ 构成
- 背包的容积为 j ， $0 \leq j \leq C$
- $V[i, j]$ 表示从 U_i 中取出若干个物品装入背包的最大总价值， $0 \leq i \leq n$
- 显然原问题可表示为 $V[n, C]$

如何求取 $V[n, C]$ 或 $V[i, j]$ ？

背包问题——动态规划算法

V[i,j]的子问题

- 有物品子集 U_i , 对第 i 个物品, 可以有几种放置方法?
- 将第 i 个物品放入到背包中, 那么还可选物品为前 $i-1$ 个物品, 背包容量变为 $j-w_i$
- 不将第 i 个物品放入到背包中, 那么还可选物品为前 $i-1$ 个物品, 但背包容量不变, 仍为 j
- 上述两种情况可表示为: $V[i-1, j-w_i]$ 和 $V[i-1, j] - V[i, j]$ 的子问题

背包问题——动态规划算法

V[i,j]的子问题

- 有物品子集 U_i , 对第 i 个物品, 可以有几种放置方法?
- 将第 i 个物品放入到背包中, 那么还可选物品为前 $i-1$ 个物品, 背包容量变为 $j-w_i$
- 不将第 i 个物品放入到背包中, 那么还可选物品为前 $i-1$ 个物品, 但背包容量不变, 仍为 j
- 上述两种情况可表示为: $V[i-1, j-w_i]$ 和 $V[i-1, j] - V[i, j]$ 的子问题

背包问题——动态规划算法

V[i,j]的子问题

- 有物品子集 U_i , 对第 i 个物品, 可以有几种放置方法?
- 将第 i 个物品放入到背包中, 那么还可选物品为前 $i-1$ 个物品, 背包容量变为 $j-w_i$
- 不将第 i 个物品放入到背包中, 那么还可选物品为前 $i-1$ 个物品, 但背包容量不变, 仍为 j
- 上述两种情况可表示为: $V[i-1,j-w_i]$ 和 $V[i-1,j]-V[i,j]$ 的子问题

背包问题——动态规划算法

V[i,j]的子问题

- 有物品子集 U_i , 对第 i 个物品, 可以有几种放置方法?
- 将第 i 个物品放入到背包中, 那么还可选物品为前 $i-1$ 个物品, 背包容量变为 $j-w_i$
- 不将第 i 个物品放入到背包中, 那么还可选物品为前 $i-1$ 个物品, 但背包容量不变, 仍为 j
- 上述两种情况可表示为: $V[i-1,j-w_i]$ 和 $V[i-1,j]-V[i,j]$ 的子问题

背包问题——动态规划算法

V[i,j]的子问题

- 有物品子集 U_i , 对第 i 个物品, 可以有几种放置方法?
- 将第 i 个物品放入到背包中, 那么还可选物品为前 $i-1$ 个物品, 背包容量变为 $j-w_i$
- 不将第 i 个物品放入到背包中, 那么还可选物品为前 $i-1$ 个物品, 但背包容量不变, 仍为 j
- 上述两种情况可表示为: $V[i-1, j-w_i]$ 和 $V[i-1, j] - V[i, j]$ 的子问题

背包问题——动态规划算法

V[i,j]和它的子问题之间的关系

- $V[i,j] = V[i-1,j]$, 还是 $V[i,j] = V[i-1,j-w_i] + v_i$
- $V[i,j] = \max\{V[i-1,j], V[i-1,j-w_i] + v_i\}$

背包问题——动态规划算法

- 递推关系：

$$F(i, j) = \begin{cases} \max\{F(i - 1, j), v_i + F(i - 1, j - w_i)\}, & j - w_i \geq 0 \\ F(i - 1, j), & j - w_i < 0 \end{cases}$$

$$F(0, j) = 0, j \geq 0; F(i, 0) = 0, i \geq 0$$

关系总结：

- 揭示原问题与子问题之间的关系，递归地定义出原问题的最优解
- 可以自底向上的方式计算出最优解，即，
- 从求取最简单的子问题开始，然后依次求取更复杂的子问题，直至原问题被求解

背包问题——动态规划算法实例

物品	重量 (w)	价值 (v)
1	2	12
2	1	10
3	3	20
4	2	15
背包承重量 $W = 5$		

$F(i, j)$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

背包问题——动态规划算法实例

回溯求装入背包的物品

- $F(4, 5) > F(3, 5)$: 物品4装入背包;
- $F(3, 5 - 2) = F(2, 5 - 2)$: 物品3不是最优子集一部分;
- $F(2, 3) > F(1, 3)$: 物品2装入背包;
- $F(1, 3 - 1) > F(0, 3 - 1)$: 物品1装入背包;

背包问题——动态规划算法实例

背包实例

假设有4个物品： $U=\{u_1, u_2, u_3, u_4\}$

它们的重量为： $W=\{w_1, w_2, w_3, w_4\} = \{2, 3, 4, 5\}$

它们的价值为： $V=\{v_1, v_2, v_3, v_4\} = \{3, 4, 5, 7\}$

背包的总容量： $C=9$

背包所装物品的最大价值是多少？

问题的最优解定义： $V[i,j] = \max\{V[i-1,j], V[i-1,j-w_i]+v_i\}$

背包问题——动态规划算法实例

它们的重量为： $W = \{w_1, w_2, w_3, w_4\} = \{2, 3, 4, 5\}$

它们的价值为： $V = \{v_1, v_2, v_3, v_4\} = \{3, 4, 5, 7\}$

算法数学基础： $V[i,j] = \max\{V[i-1,j], V[i-1,j-w_i] + v_i\}$

C:0	1	2	3	4	5	6	7	8	9	
U:0	V[0,0]	V[0,1]	V[0,2]	V[0,3]	V[0,4]	V[0,5]	V[0,6]	V[0,7]	V[0,8]	V[0,9]
1	V[1,0]	V[1,1]	V[1,2]	V[1,3]	V[1,4]	V[1,5]	V[1,6]	V[1,7]	V[1,8]	V[1,9]
2	V[2,0]	V[2,1]	V[2,2]	V[2,3]	V[2,4]	V[2,5]	V[2,6]	V[2,7]	V[2,8]	V[2,9]
3	V[3,0]	V[3,1]	V[3,2]	V[3,3]	V[3,4]	V[3,5]	V[3,6]	V[3,7]	V[3,8]	V[3,9]
4	V[4,0]	V[4,1]	V[4,2]	V[4,3]	V[4,4]	V[4,5]	V[4,6]	V[4,7]	V[4,8]	V[4,9]

$$V[1,1] = \max\{V[0,1], V[0,1-w_1] + v_1\} = \max\{0, 0\} = 0$$

背包问题——动态规划算法实例

它们的重量为： $W = \{w_1, w_2, w_3, w_4\} = \{2, 3, 4, 5\}$

它们的价值为： $V = \{v_1, v_2, v_3, v_4\} = \{3, 4, 5, 7\}$

算法数学基础： $V[i,j] = \max\{V[i-1,j], V[i-1,j-w_i] + v_i\}$

		C:0	1	2	3	4	5	6	7	8	9
U:0		V[0,0]	V[0,1]	V[0,2]	V[0,3]	V[0,4]	V[0,5]	V[0,6]	V[0,7]	V[0,8]	V[0,9]
1		V[1,0]	0	V[1,2]	V[1,3]	V[1,4]	V[1,5]	V[1,6]	V[1,7]	V[1,8]	V[1,9]
2		V[2,0]	V[2,1]	V[2,2]	V[2,3]	V[2,4]	V[2,5]	V[2,6]	V[2,7]	V[2,8]	V[2,9]
3		V[3,0]	V[3,1]	V[3,2]	V[3,3]	V[3,4]	V[3,5]	V[3,6]	V[3,7]	V[3,8]	V[3,9]
4		V[4,0]	V[4,1]	V[4,2]	V[4,3]	V[4,4]	V[4,5]	V[4,6]	V[4,7]	V[4,8]	V[4,9]

$$V[1,1] = \max\{V[0,1], V[0,1-w_1] + v_1\} = \max\{0, 0\} = 0$$

背包问题——动态规划算法实例

它们的重量为： $W = \{w_1, w_2, w_3, w_4\} = \{2, 3, 4, 5\}$

它们的价值为： $V = \{v_1, v_2, v_3, v_4\} = \{3, 4, 5, 7\}$

算法数学基础： $V[i,j] = \max\{V[i-1,j], V[i-1,j-w_i] + v_i\}$

C:0		1	2	3	4	5	6	7	8	9
U:0	V[0,0]	V[0,1]	V[0,2]	V[0,3]	V[0,4]	V[0,5]	V[0,6]	V[0,7]	V[0,8]	V[0,9]
1	V[1,0]	0	3	3	3	3	3	3	3	3
2	V[2,0]	0	3	4	4	7	7	7	7	7
3	V[3,0]	0	3	4	5	7	8	9	9	12
4	V[4,0]	0	3	4	5	7	8	10	11	V[4,9]

背包问题——动态规划算法实例

它们的重量为： $W = \{w_1, w_2, w_3, w_4\} = \{2, 3, 4, 5\}$

它们的价值为： $V = \{v_1, v_2, v_3, v_4\} = \{3, 4, 5, 7\}$

算法数学基础： $V[i,j] = \max\{V[i-1,j], V[i-1,j-w_i] + v_i\}$

C:0	1	2	3	4	5	6	7	8	9	
U:0	V[0,0]	V[0,1]	V[0,2]	V[0,3]	V[0,4]	V[0,5]	V[0,6]	V[0,7]	V[0,8]	V[0,9]
1	V[1,0]	0	3	3	3	3	3	3	3	3
2	V[2,0]	0	3	4	4	7	7	7	7	7
3	V[3,0]	0	3	4	5	7	8	9	9	12
4	V[4,0]	0	3	4	5	7	8	10	11	12

背包问题——动态规划算法实例

背包问题的计算过程总结

4个物品、背包容量为9的计算过程，总共计算 4×9 次！

计算次序

		C:0 1 2 3 4 5 6 7 8 9									
		V[0,0]	V[0,1]	V[0,2]	V[0,3]	V[0,4]	V[0,5]	V[0,6]	V[0,7]	V[0,8]	V[0,9]
↓	U:0	V[1,0]	0	3	3	3	3	3	3	3	3
	1	V[2,0]	0	3	4	4	7	7	7	7	7
	2	V[3,0]	0	3	4	5	7	8	9	9	12
	3	V[4,0]	0	3	4	5	7	8	10	11	12
	4										

背包问题——动态规划算法实例

计算的依据：

if $w_i > j$ **then**

$$V[i,j] = V[i-1,j]$$

else

$$V[i,j] = \max\{V[i-1,j], V[i-1,j-w_i] + v_i\}$$

end if

背包问题——动态规划算法伪代码

Knapsack

//输入： 背包容量C、 物品集 $U=\{u_1, u_2, \dots, u_n\}$ 、 物品的重量集合 $W=\{w_1, w_2, \dots, w_n\}$, 物品价值集合 $V=\{v_1, v_2, \dots, v_n\}$
//输出： 背包中物品的最大价值

定义二维数组 $V[0 \dots n, 0 \dots C]$, 用于计算和存放 $V[i,j]$ 的值

初始化数组 $V[0 \dots n, 0 \dots C]$ [Jump to initialization](#)

```
for i = 0 to n do
    for j = 0 to C do
        if  $w_i > j$  then
             $V[i,j] = V[i-1,j]$ 
        else
             $V[i,j] = \max\{V[i-1,j], V[i-1,j-w_i]+v_i\}$ 
        end if
    end for
end for
return V
```

背包问题——动态规划算法伪代码

```
initialization()
```

```
//输入：二维数组V[0…n,0…C]
```

```
//输出：数组初始化
```

```
for i = 0 to n do
```

```
    V[i,0] = 0
```

```
end for
```

```
for j = 0 to C do
```

```
    V[0,j] = 0
```

```
end for
```

背包问题——动态规划算法时间效率分析

时间效率分析

- 由于计算表的每一项需要 $\Theta(1)$ 时间，所以算法的时间复杂性恰好是表的大小 $\Theta(nC)$ ；

动态规划优化

记忆化

- 自顶向下的递归调用求解会多次求解子问题；
- 自下向上的动态规划过程会求解一些不必要的子问题；
- 结合两种方法，只求解一次必要的子问题（记忆化）；

物品	重量 (w)	价值 (v)
1	2	12
2	1	10
3	3	20
4	2	15
背包承重量 $W = 5$		

$F(i, j)$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	-	12	22	-	22
3	0	-	-	22	-	32
4	0	-	-	-	-	37

背包问题——动态规划算法背包问题总结

动态规划法应用小结

动态规划法主要是采用**自下而上、自左至右**的方法逐步求解

背包问题的求解

- 自下而上表现为：可选择的物品(i)逐个增加
- 自左至右表现为：在可选择物品确定的情况下(j)，背包容量按单位容积增加
- 逐个求出对应子问题的最优解