



动态规划法

思想，就像幽灵一样……
在它自己解释自己之前，必须
先告诉它些什么。

——狄更斯





思考

- 假设你正在管理一条公路的广告牌建设，这条路从西到东 M 英里。广告牌可能的地点假设为 $x_1, x_2, x_3 \dots x_n$ ，处于 $[0, M]$ 中。若在 x_i 放一块广告牌，可以得到 $r_i > 0$ 的收益。
- 国家公路局规定，两块广告牌相对不能小于或等于5英里之内。
- 如何找一组地点使你的总收益达到最大？





分治法回顾

- 分治法思想的核心是：将复杂的问题分解为小的简单的问题，解决这些小问题，再合并成为大问题的解。
- 施行分治策略时基于以下几点认识：
 - 1) 小问题比大问题更容易解决
 - 2) 将小问题解答组装成大问题解答所需成本低于直接解决大问题的成本
 - 3) 小问题又可按照同样的方法分解为更小的问题





什么是动态规划法 (Dynamic Programming)

动态规划是运筹学的一个分支。

它是解决多阶段决策过程最优化问题的一种方法。

1951年，美国数学家**R.Bellman**提出了解决这类问题的“最优化原则”，**1957**年发表了他的名著《动态规划》。





Dynamic Programming Applications

- Areas.
 - Bioinformatics (生物信息学) .
 - Control theory.
 - Information theory.
 - Computer science: theory, graphics, AI, systems,
- Some famous dynamic programming algorithms.
 - Viterbi (维特比) for hidden Markov models.
 - Unix diff for comparing two files.
 - Smith-Waterman for sequence alignment.
 - Bellman-Ford for shortest path routing in networks.
 - Cocke-Kasami-Younger for parsing context free grammars.





认识动态规划





斐波那契数列





我们先来做一个游戏！





十秒钟加数

$$\begin{array}{r} 1 \\ 2 \\ 3 \\ 5 \\ 8 \\ 13 \\ 21 \\ 34 \\ 55 \\ + 89 \\ \hline ?? \end{array}$$

- 请用十秒，计算出左边一列数的和。

时间到！

- 答案是 231。





十秒钟加数

34

55

89

144

233

377

610

987

1597

$$\begin{array}{r} + \quad 2584 \\ \hline \quad \quad \quad \end{array}$$

????

• 再来一次！

时间到！

■ 答案是 6710。



这与“斐波那契数列”有关

- 若一个数列，前两项等于1，而从第三项起，每一项是其前两项之和，则称该数列为斐波那契数列。即：

1, 1, 2, 3, 5, 8, 13,





一、兔子问题和斐波那契数列

1. 兔子问题

1) 问题

——取自意大利数学家
斐波那契的《算盘书》
(1202年)

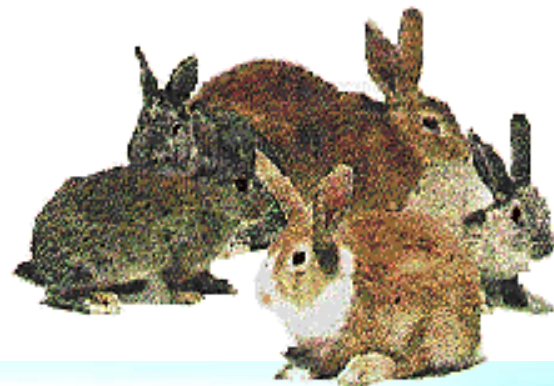
(L. Fibonacci, 1170–1250)





兔子问题

假设一对初生兔子要一个月才到成熟期，而一对成熟兔子每月会生一对兔子，那么，由一对初生兔子开始，12 个月后会 有多少对兔子呢？





解答

- 可以将结果以列表形式给出：

| 1月 | 2月 | 3月 | 4月 | 5月 | 6月 |
|----|----|----|----|----|----|
| 1 | 1 | 2 | 3 | 5 | 8 |

| 7月 | 8月 | 9月 | 10月 | 11月 | 12月 |
|----|----|----|-----|-----|-----|
| 13 | 21 | 34 | 55 | 89 | 144 |

- 因此，斐波那契问题的答案是 144对。
- 以上数列， 即“斐波那契数列”



2. 斐波那契数列

1) 公式

用 F_n 表示第 n 个月大兔子的对数，则有
二阶递推公式

$$\begin{cases} F_1 = F_2 = 1 \\ F_n = F_{n-1} + F_{n-2}, n = 3, 4, 5 \dots \end{cases}$$



2) 斐波那契数列

令 $n = 1, 2, 3, \dots$ 依次写出数列，就是

1, 1, 2, 3, 5, 8, 13, 21, 34,

55, 89, 144, 233, 377, ...

这就是斐波那契数列。其中的任一个数，都叫斐波那契数。



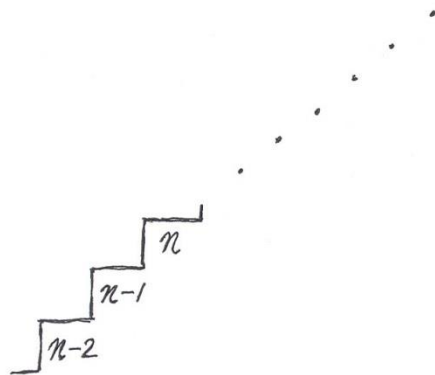
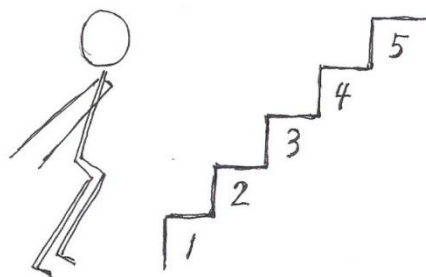
二、 相关的问题

斐波那契数列是从兔子问题中抽象出来的，在许多问题中出现。



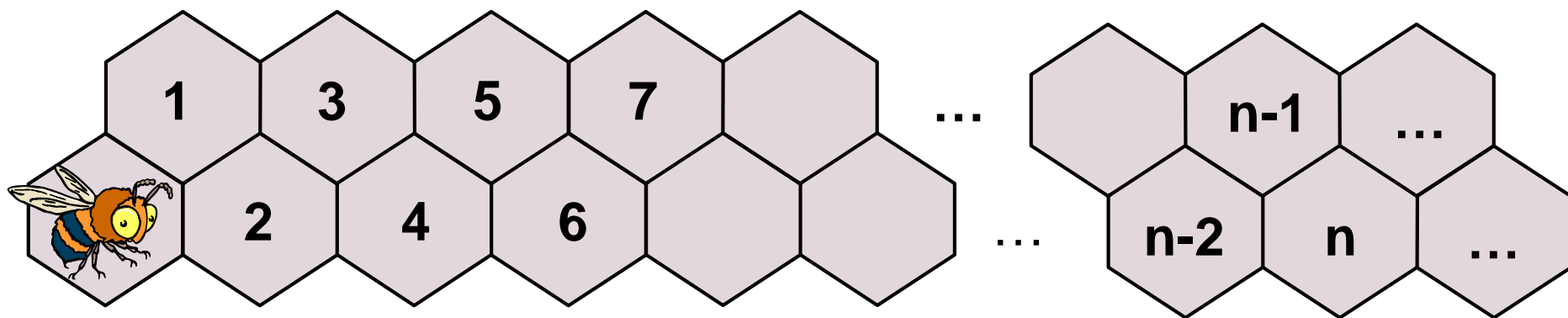


1. 跳格游戏



2. 蜜蜂进蜂房问题:

一次蜜蜂从蜂房A出发，想爬到、 \dots 、 n 号蜂房，只允许它自左向右（不许反方向倒走）。则它爬到各号蜂房的路线多少？





3. 自然界中的斐波那契数

斐波那契数列中的任一个数，都叫斐波那契数。斐波那契数是大自然的一个基本模式，它出现在许多场合。

下面举几个例子。



花瓣中的斐波那契数
花瓣的数目



海棠 (2)



铁兰 (3)

花瓣中的斐波那契数

花瓣的数目



洋紫荊 (5)



黃蟬 (5)



蝴蝶蘭 (5)



花瓣中的斐波那契数

花瓣的数目



雏菊 (13)

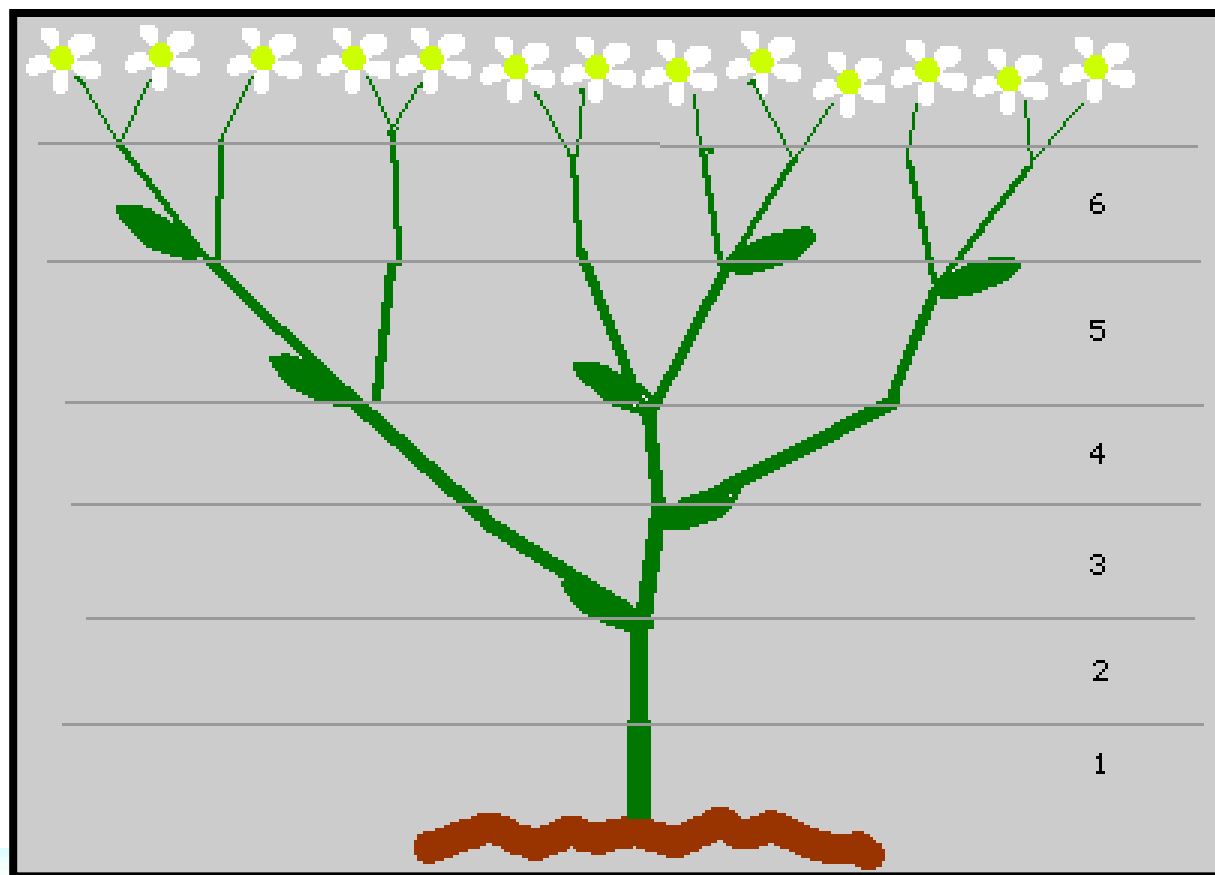


雏菊 (13)





2) 树杈的数目

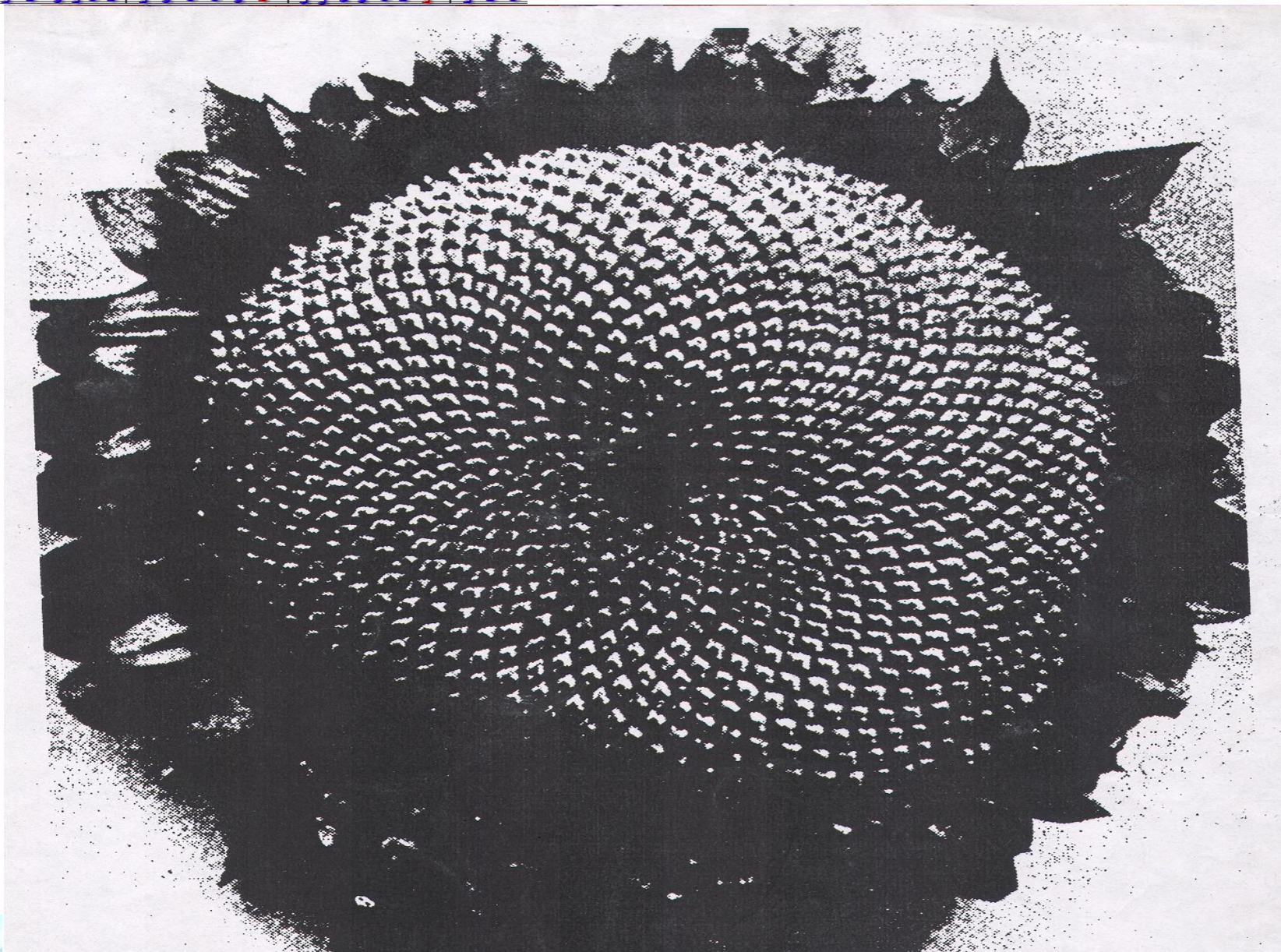


13
8
5
3
2
1
1



3) 向日葵花盘内葵花子排列的螺线数







向日葵花盘内，种子是按对数螺线排列的，有顺时针转和逆时针转的两组螺线。

一般是34和55

大向日葵是89和144

更大的向日葵有144和233条螺线



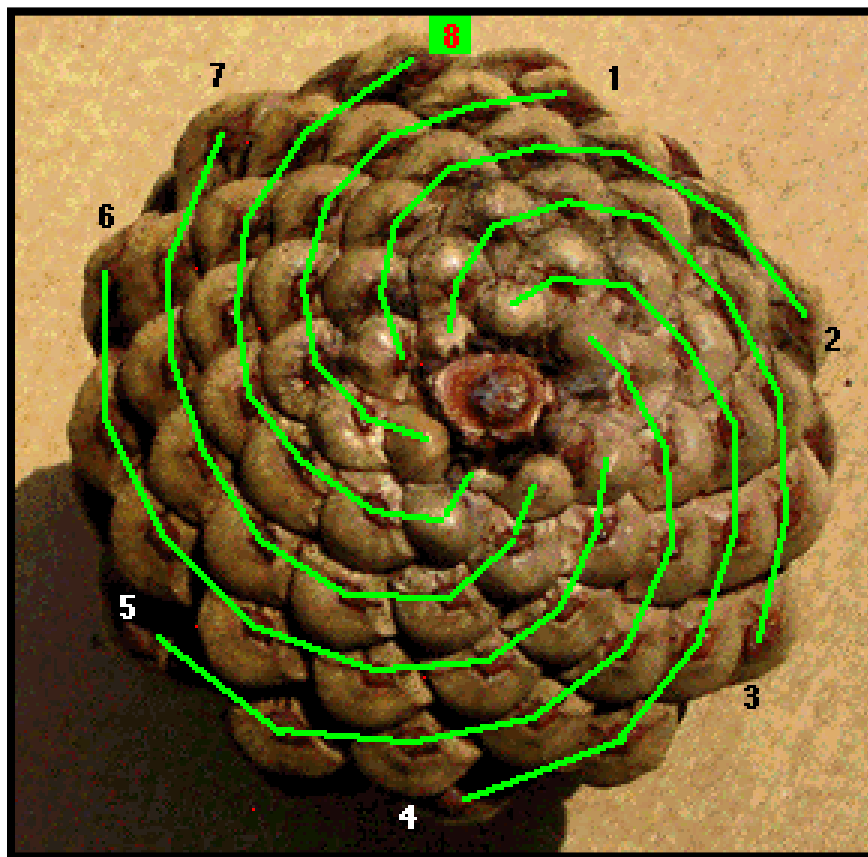


松果种子的排列



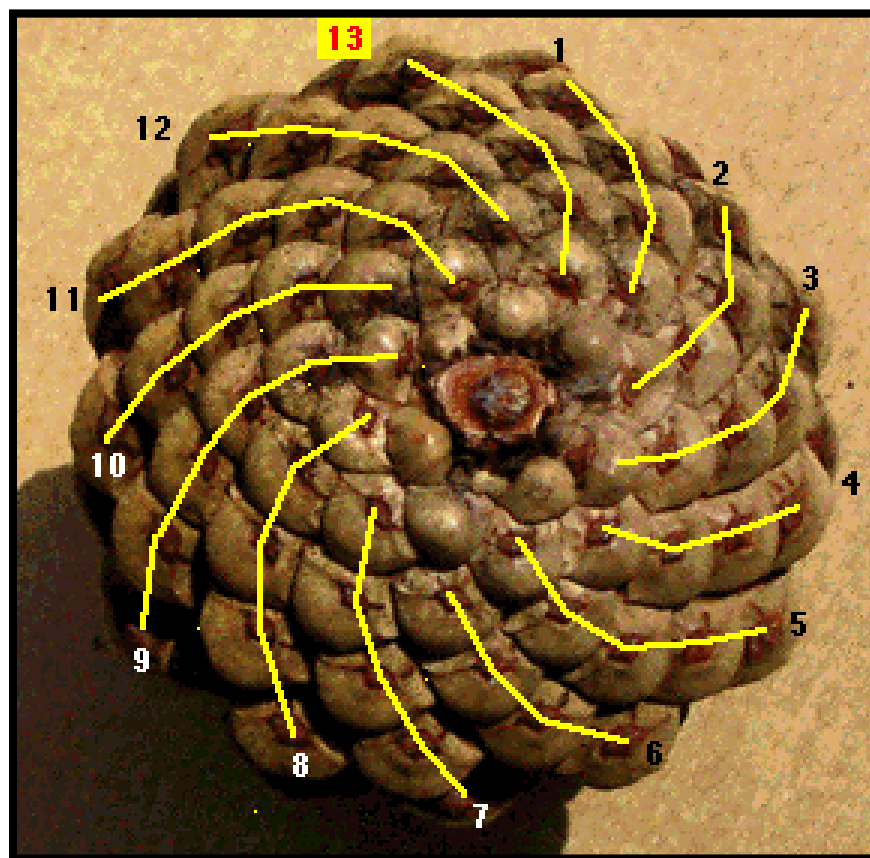


松果种子的排列



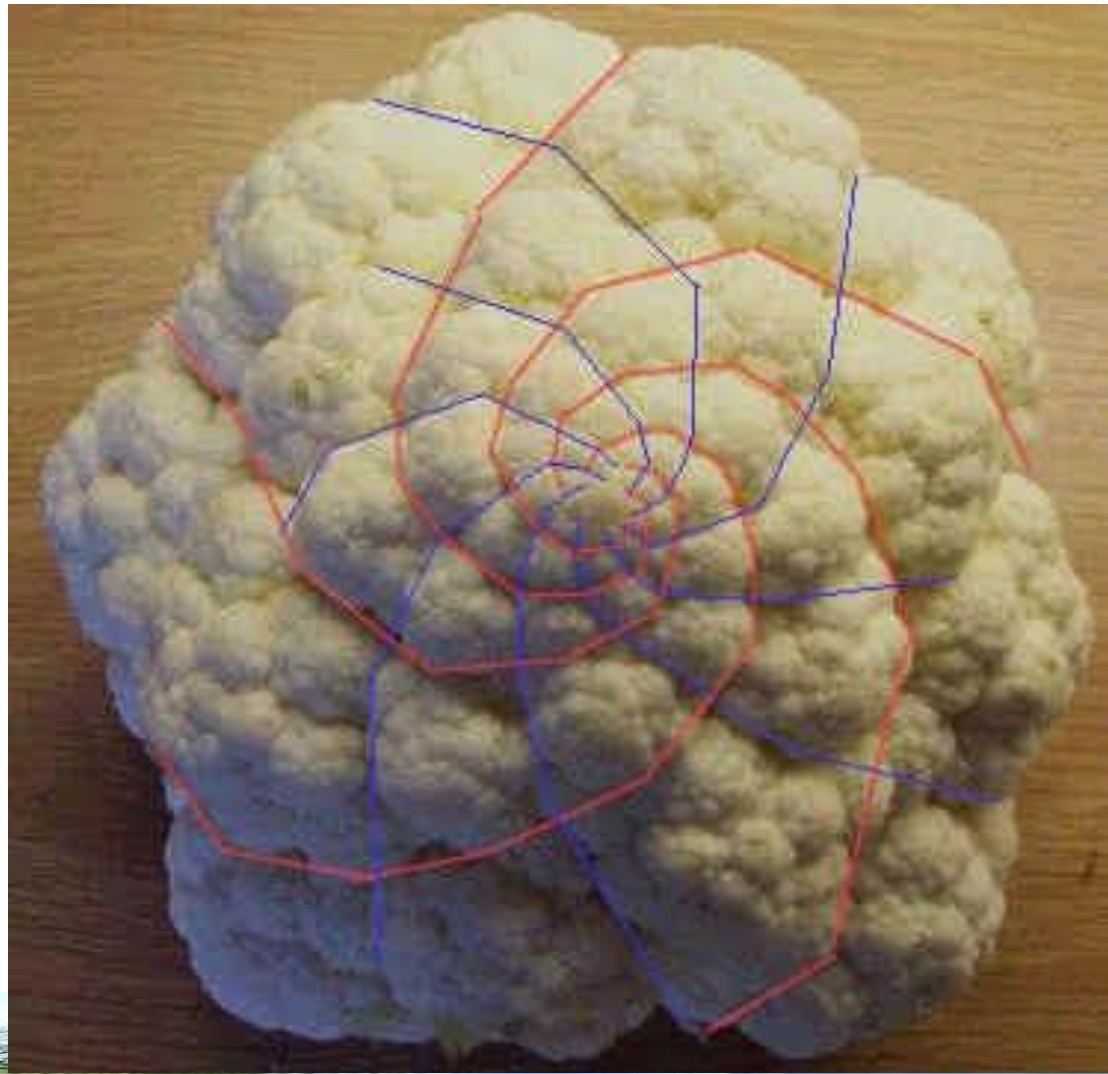


松果种子的排列





菜花表面排列的螺线数 (5-8)



“十秒钟加数”的秘密

- 数学家发现：连续 10个斐波那契数之和，必定等于第 7个数的 11 倍！

- 所以右式的答案是：

$$21 \times 11 = 231$$

$$\begin{array}{r} 1 \\ 2 \\ 3 \\ 5 \\ 8 \\ 13 \\ 21 \\ 34 \\ 55 \\ + 89 \\ \hline ?? \end{array}$$

“十秒钟加数”的秘密

- 又例如：

- 右式的答案是：

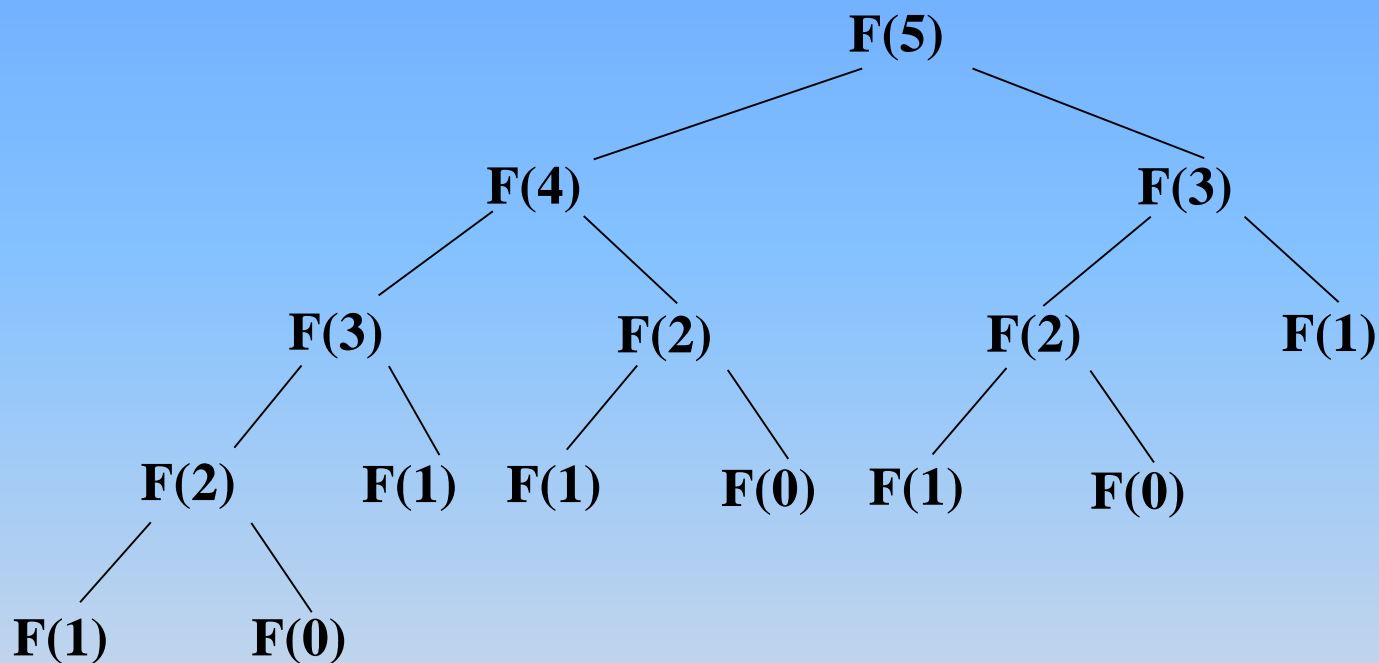
$$610 \times 11 = 6710$$

$$\begin{array}{r} 34 \\ 55 \\ 89 \\ 144 \\ 233 \\ 377 \\ 610 \\ 987 \\ 1597 \\ + 2584 \\ \hline \end{array}$$

????

例：计算斐波那契数：

$n=5$ 时分治法计算斐波那契数的过程。



$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n \geq 2 \end{cases}$$



向日葵花盘内，种子是按对数螺线排列的，有顺时针转和逆时针转的两组螺线。

一般是34和55

大向日葵是89和144

更大的向日葵有144和233条螺线





注意到，计算 $F(n)$ 是以计算它的两个重叠子问题 $F(n-1)$ 和 $F(n-2)$ 的形式来表达的，所以，可以设计一张表填入 $n+1$ 个 $F(n)$ 的值。

动态规划法求解斐波那契数 $F(9)$ 的填表过程：

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|----------|----------|----------|----------|----------|----------|----------|-----------|-----------|-----------|
| 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 |





我们通过一个简单的例子来说明动态规划的多阶段决策与贪婪算法有什么区别。

数塔问题

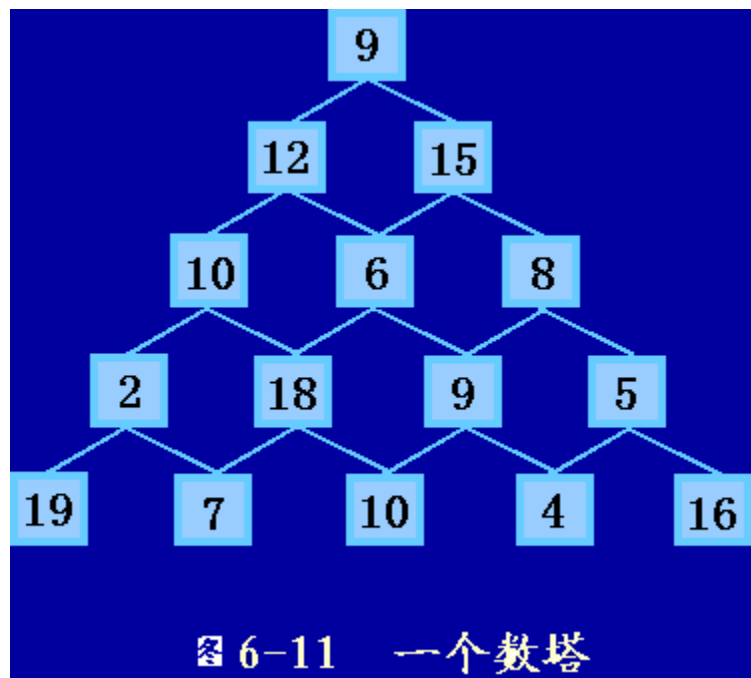




【例】数塔问题

有形如图所示的一个数塔，从顶部出发，在每一结点可以选择向左走或是向右走，一直走到底层，要求找出一条路径，使路径上的数值和最大。

- 问题分析
- 算法设计
- 算法
- 小结





问题分析

这个问题用贪婪算法有可能会找不到真正的最大和。
用贪婪的策略，路径和分别为：

$$9+15+8+9+10=51 \quad (\text{自上而下}),$$

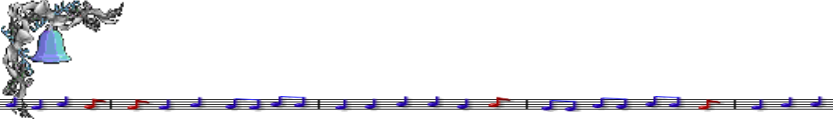
$$19+2+10+12+9=52 \quad (\text{自下而上}).$$

都得不到最优解，真正的最大和是：

$$9+12+10+18+10=59.$$

在知道数塔的全貌的前提下，可以用蛮力法来完成。





算法设计

动态规划设计过程如下：

1. 阶段划分：

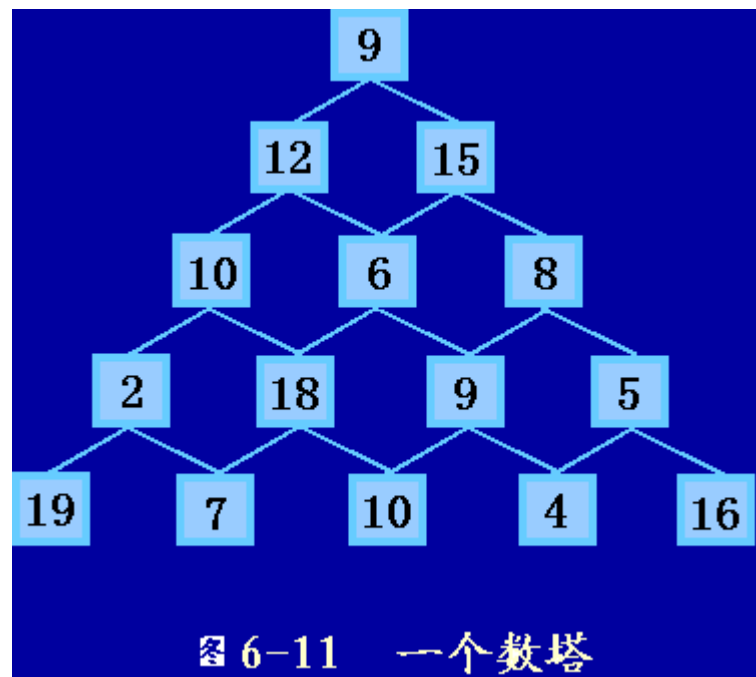
第一步对于第五层的数据，我们做如下五次决策：

对经过第四层2的路径选择第五层的19，

对经过第四层18的路径选择第五层的10，

对经过第四层9的路径也选择第五层的10，

对经过第四层5的路径选择第五层的16。



算法设计

动态规划设计过程如下：

1. 阶段划分：

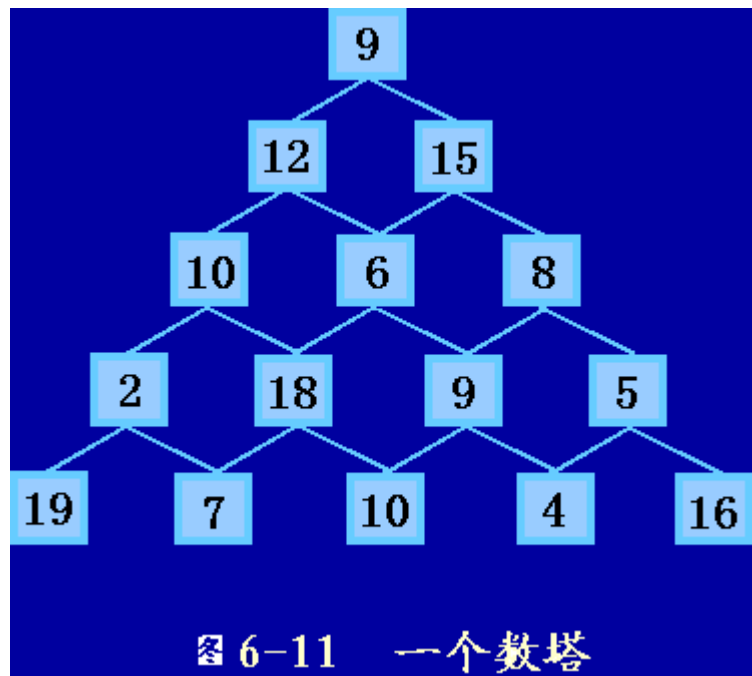
第一步对于第五层的数据，我们做如下五次决策：

对经过第四层2的路径选择第五层的19，

对经过第四层18的路径选择第五层的10，

对经过第四层9的路径也选择第五层的10，

对经过第四层5的路径选择第五层的16。



以上的决策结果将五阶数塔问题变为4阶子问题，递推出第四层与第五层的和为：

21 (2+19), 28 (18+10), 19 (9+10), 21 (5+16)。

用同样的方法还可以将4阶数塔问题, 变为3阶数塔问题。
..... 最后得到的1阶数塔问题，就是整个问题的最优解。

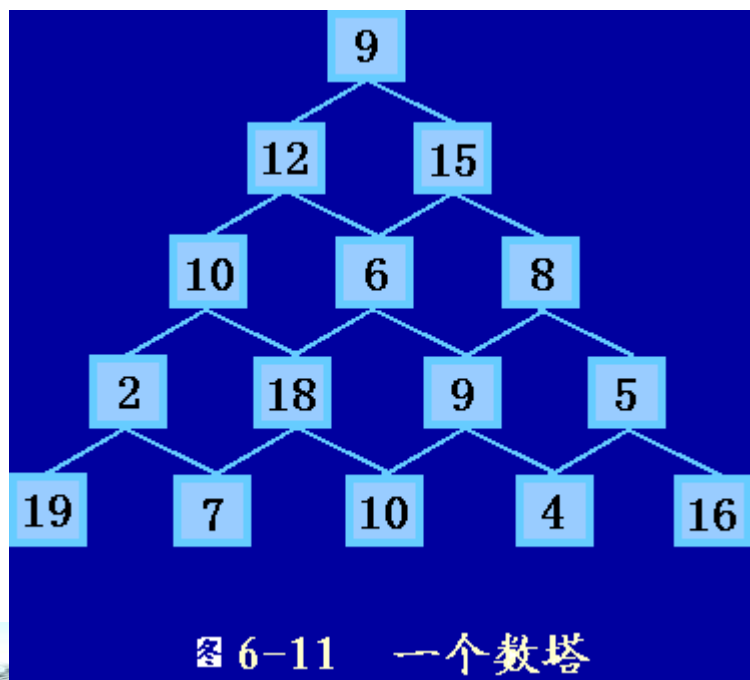


图 6-11 一个数塔

2. 存储、求解:

1) 原始信息存储

原始信息有层数和数塔中的数据，层数用一个整型变量n存储，数塔中的数据用二维数组data，存储成如下的下三角阵：

| | | | | | |
|----|----|----|---|----|--|
| 9 | | | | | |
| 12 | 15 | | | | |
| 10 | 6 | 8 | | | |
| 2 | 18 | 9 | 5 | | |
| 19 | 7 | 10 | 4 | 16 | |



2) 动态规划过程存储

用二维数组d存储各阶段的决策结果。二维数组d的存储内容如下：

$$d[n][j] = \text{data}[n][j] \quad j=1, 2, \dots, n;$$

$$d[i][j] = \max(d[i+1][j], d[i+1][j+1]) + \text{data}[i][j] \\ i=n-1, n-2, \dots, 1, \quad j=1, 2, \dots, i$$

最后 $d[1][1]$ 存储的就是问题的结果。

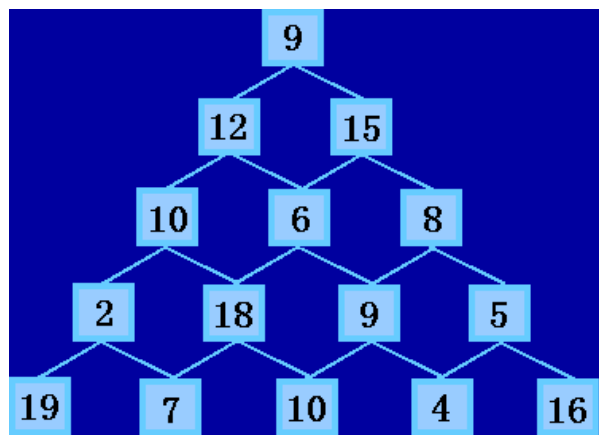


图 6-11 一个数塔



从例子中可以看到：

动态规划=贪婪策略+递推(降阶)+存储递推结果

贪婪策略、递推算法都是在“线性”地解决问题，而动态规划则是全面分阶段地解决问题。可以通俗地说动态规划是“带决策的多阶段、多方位的递推算法”。



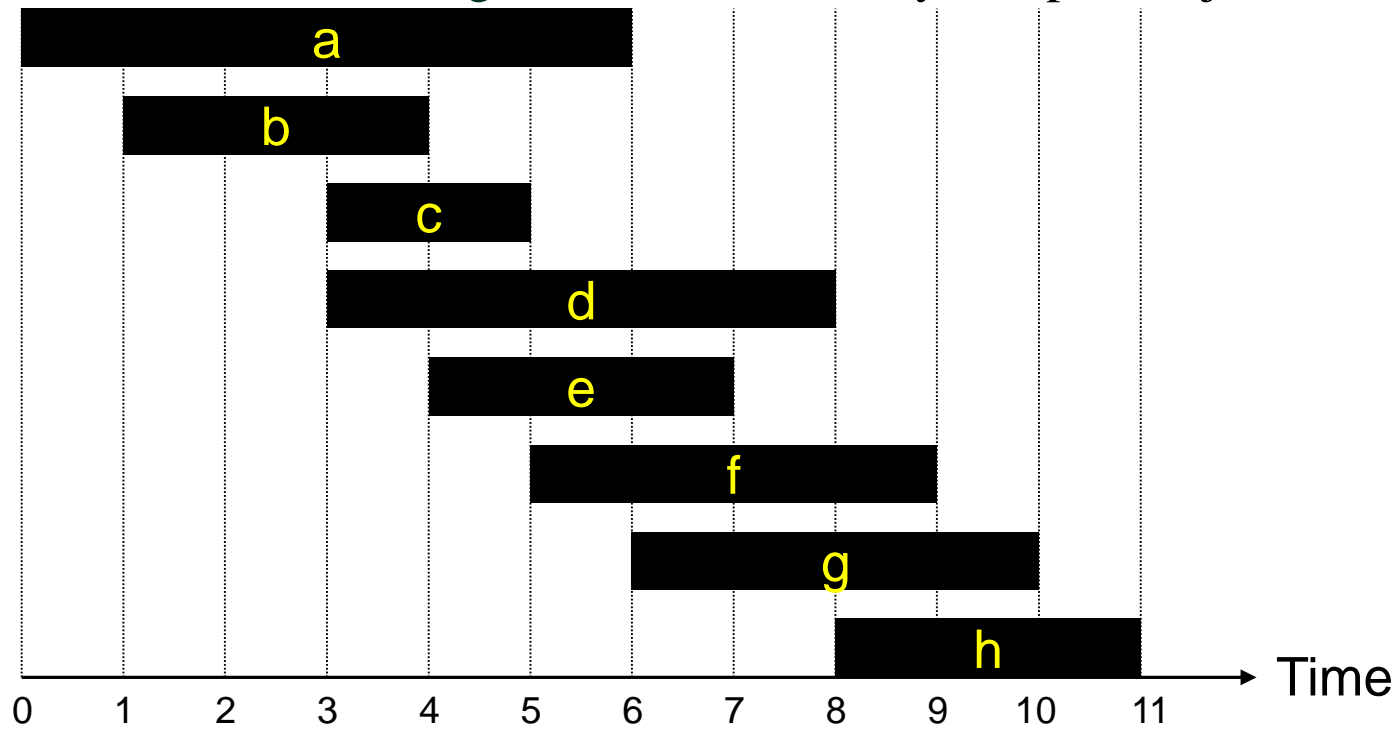


Weighted Interval Scheduling



Weighted Interval Scheduling

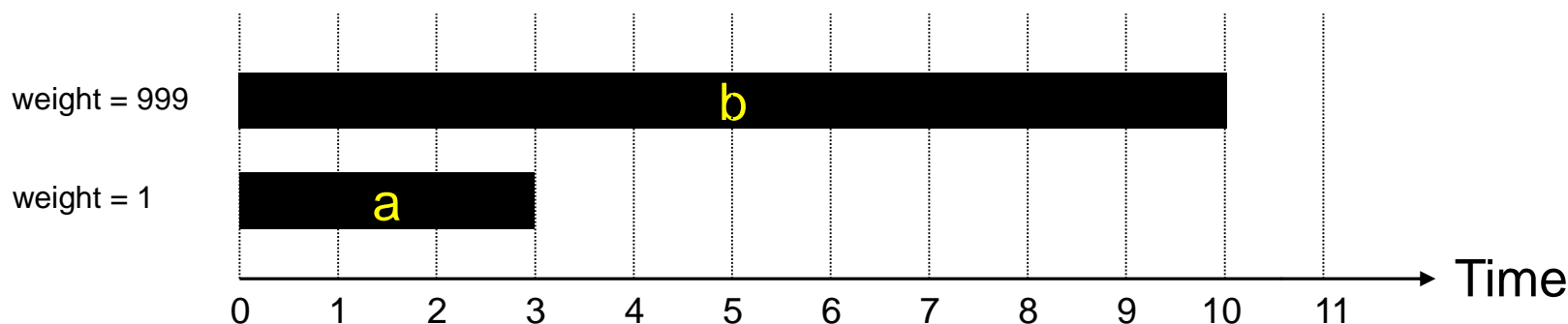
- Weighted interval scheduling problem.
 - Job j starts at s_j , finishes at f_j , and has weight or value v_j .
 - Two jobs **compatible** if they don't overlap.
 - Goal: find maximum **weight** subset of mutually compatible jobs.





Unweighted Interval Scheduling Review

- Recall. Greedy algorithm works if all weights are 1.
 - Consider jobs in ascending order of finish time.
 - Add job to subset if it is compatible with previously chosen jobs.



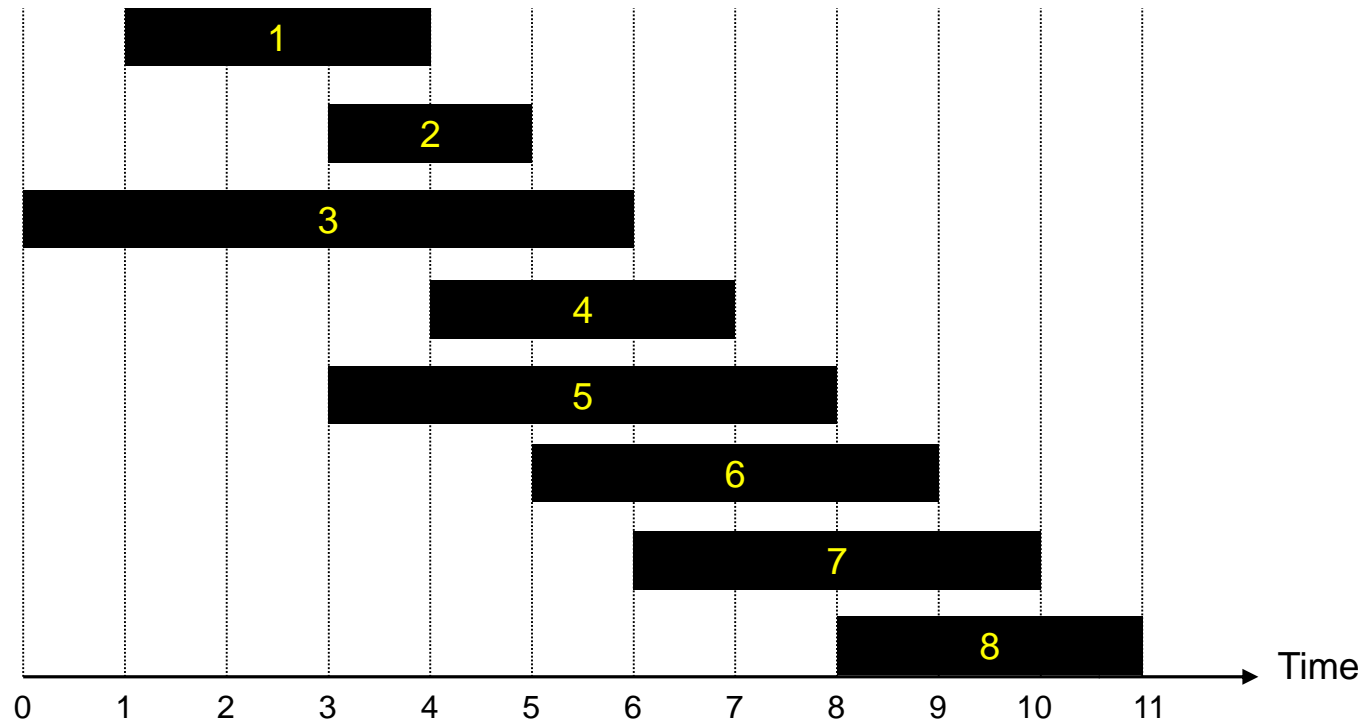
- Observation. Greedy algorithm can fail spectacularly (令人吃惊地) if arbitrary weights are allowed.



Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \dots \leq f_n$.

Def. $p(j)$ = largest index $i < j$ such that job i is compatible with j .
Ex: $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.





Dynamic Programming: Binary Choice

- Notation. $OPT(j)$ = value of optimal solution to the problem consisting of job requests $1, 2, \dots, j$.
 - Case 1: OPT selects job j .
 - can't use incompatible jobs $\{ p(j) + 1, p(j) + 2, \dots, j - 1 \}$
 - must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \dots, p(j)$ optimal substructure
 - Case 2: OPT does not select job j .
 - must include optimal solution to problem consisting of

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \{ v_j + OPT(p(j)), OPT(j-1) \} & \text{otherwise} \end{cases}$$



Weighted Interval Scheduling:

- Algorithm 1.

Input: $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p(1), p(2), \dots, p(n)$

```
Compute-Opt(j) {  
    if (j = 0)  
        return 0  
    else  
        return max( $v_j + \text{Compute-Opt}(p(j))$ ,  
        Compute-Opt(j-1))  
}
```

[illegible]

-





Weighted Interval Scheduling: Bottom-Up



- Bottom-up dynamic programming. Unwind recursion.

Input: $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p(1), p(2), \dots, p(n)$

```
Iterative-Compute-Opt {  
     $M[0] = 0$   
    for  $j = 1$  to  $n$   
         $M[j] = \max(v_j + M[p(j)], M[j-1])$   
}
```



Weighted Interval Scheduling: Running Time

- Claim. Memorized version of algorithm takes $O(n \log n)$ time.
 - Sort by finish time: $O(n \log n)$.
- Remark. $O(n)$ if jobs are pre-sorted by start and finish times.





Weighted Interval Scheduling: Finding a Solution

- Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?
- A. Do some post processing

Run M-Compute-Opt(n)

Run Find-Solution(n)

```
Find-Solution(j) {  
    if (j = 0)  
        output nothing  
    else if ( $v_j + M[p(j)] > M[j-1]$ )  
        print j  
        Find-Solution(p(j))  
    else  
        Find-Solution(j-1)  
}
```

– # of recursive calls $\leq n \Rightarrow O(n)$.



动态规划简介

学习动态规划最重要的是“一种思想方法和解决问题的过程”。

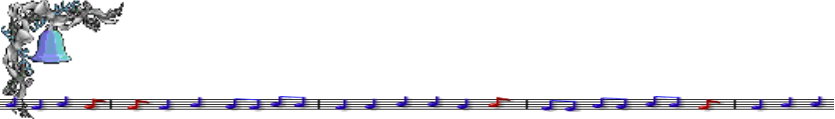




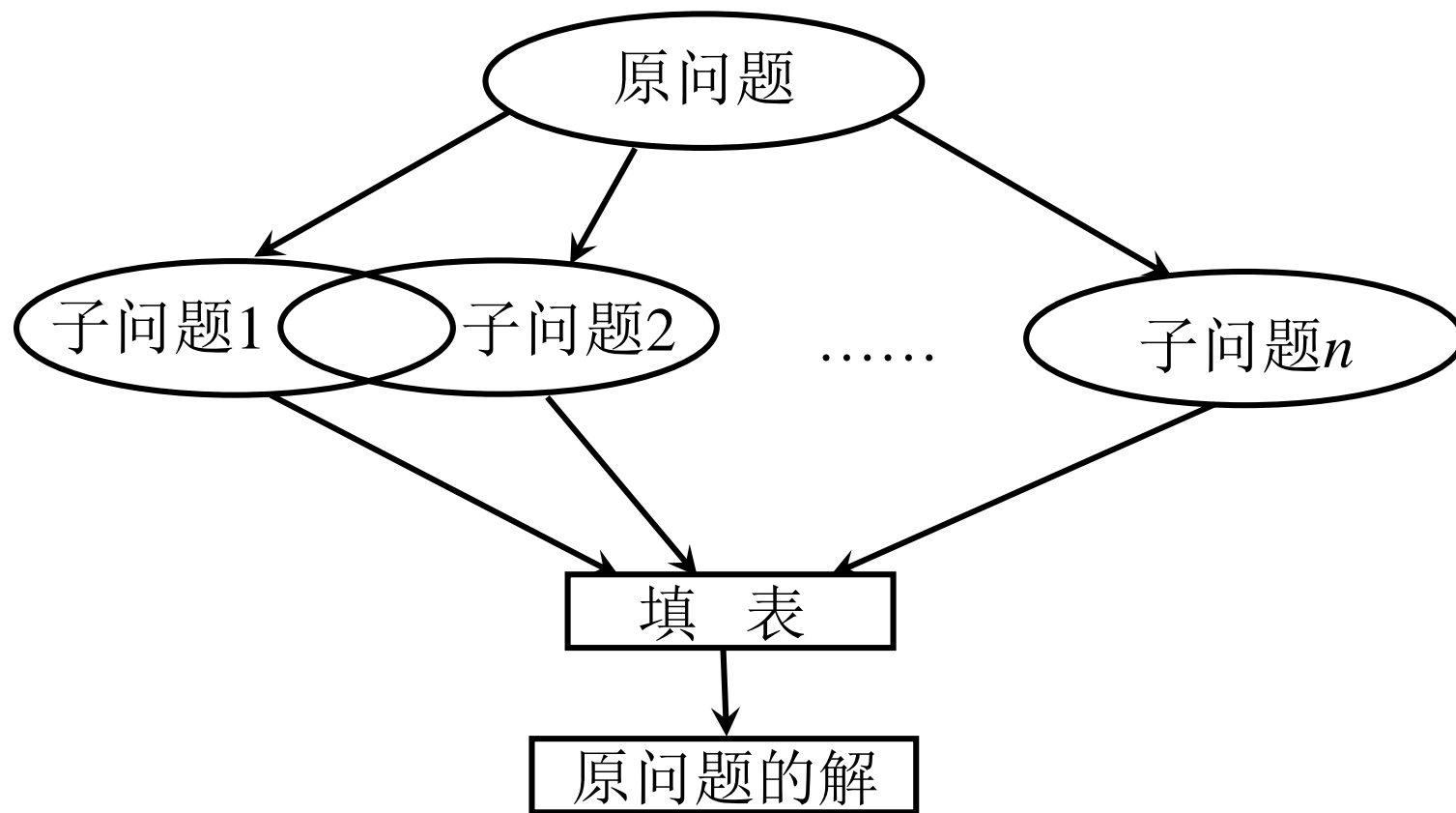
动态规划法的设计思想

动态规划法将待求解问题分解成若干个**相互重叠**的子问题，每个子问题对应决策过程的一个**阶段**，一般来说，子问题的重叠关系表现在对给定问题求解的**递推关系**（也就是动态规划函数）中，将子问题的解求解一次并**填入表**中，当需要再次求解此子问题时，可以通过查表获得该子问题的解而不用再次求解，从而避免了大量重复计算。





动态规划法的求解过程





用动态规划法求解的问题具有特征：

- ✓ 能够分解为相互重叠的若干子问题；
- ✓ 满足最优性原理（也称最优子结构性质）：该问题的最优解中也包含着其子问题的最优解。

（用反证法）分析问题是否满足最优性原理：

1. 先假设由问题的最优解导出的子问题的解不是最优的；
2. 然后再证明在这个假设下可构造出比原问题最优解更好的解，从而导致矛盾。





- 动态规划法看起来比较复杂而难以理解。
这是因为：在实际中动态规划的构造依赖于具体的问题。
- 是一种“依赖于序列决策问题的特定结构的艺术性智力活动”





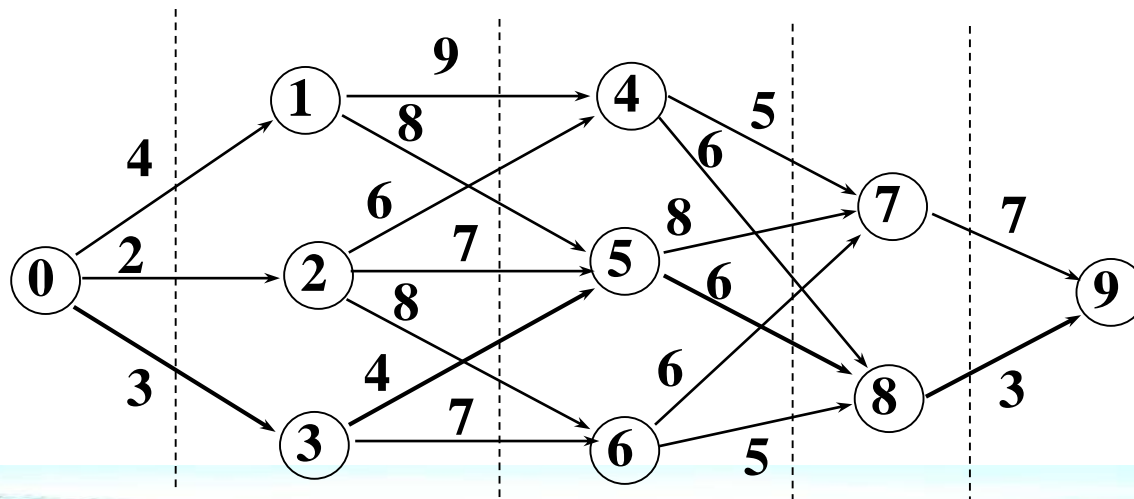
图问题中的动态规划法

多段图的最短路径问题



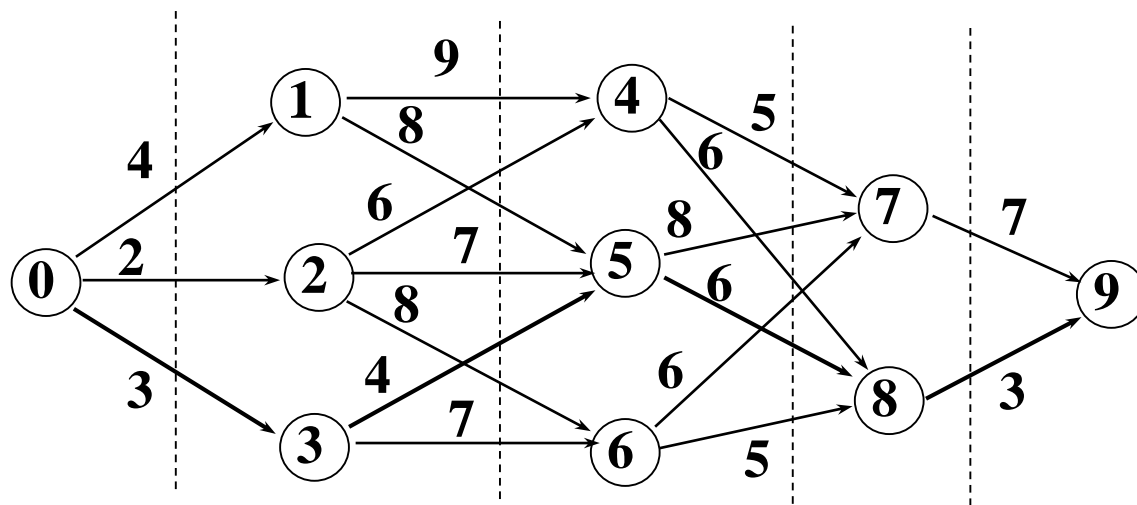
多段图的最短路径问题

设图 $G=(V, E)$ 是一个带权有向连通图，如果把顶点集合 V 划分成 k 个互不相交的子集 V_i ($2 \leq k \leq n$, $1 \leq i \leq k$)，使得 E 中的任何一条边 (u, v) ，必有 $u \in V_i$, $v \in V_{i+m}$ ($1 \leq i < k$, $1 < i+m \leq k$)，则称图 G 为多段图，称 $s \in V_1$ 为源点， $t \in V_k$ 为终点。多段图的最短路径问题是求从源点到终点的最小代价路径。



一个多段图

由于多段图将顶点划分为 k 个互不相交的子集，所以，多段图划分为 k 段，每一段包含顶点的一个子集。不失一般性，将多段图的顶点按照段的顺序进行编号，同一段内顶点的相互顺序无关紧要。假设图中的顶点个数为 n ，则源点 s 的编号为 0 ，终点 t 的编号为 $n-1$ ，并且，对图中的任何一条边 (u, v) ，顶点 u 的编号小于顶点 v 的编号。



一个多段图



证明多段图问题满足最优性原理

设 $s, s_1, s_2, \dots, s_p, t$ 是从 s 到 t 的一条最短路径，从源点 s 开始，设从 s 到下一段的顶点 s_1 已经求出，则问题转化为求从 s_1 到 t 的最短路径，显然 s_1, s_2, \dots, s_p, t 一定构成一条从 s_1 到 t 的最短路径，如若不然，设 $s_1, r_1, r_2, \dots, r_q, t$ 是一条从 s_1 到 t 的最短路径，则 $s, s_1, r_1, r_2, \dots, r_q, t$ 将是一条从 s 到 t 的路径且比 $s, s_1, s_2, \dots, s_p, t$ 的路径长度要短，从而导致矛盾。所以，多段图的最短路径问题满足最优性原理。





对多段图的边 (u, v) ，用 c_{uv} 表示边上的权值，将从源点 s 到终点 t 的最短路径记为 $d(s, t)$ ，则从源点0到终点9的最短路径 $d(0, 9)$ 由下式确定：

$$d(0, 9) = \min\{c_{01} + d(1, 9), c_{02} + d(2, 9), c_{03} + d(3, 9)\}$$

这是最后一个阶段的决策，它依赖于 $d(1, 9)$ 、 $d(2, 9)$ 和 $d(3, 9)$ 的计算结果，而

$$d(1, 9) = \min\{c_{14} + d(4, 9), c_{15} + d(5, 9)\}$$

$$d(2, 9) = \min\{c_{24} + d(4, 9), c_{25} + d(5, 9), c_{26} + d(6, 9)\}$$

$$d(3, 9) = \min\{c_{35} + d(5, 9), c_{36} + d(6, 9)\}$$

这一阶段的决策又依赖于 $d(4, 9)$ 、 $d(5, 9)$ 和 $d(6, 9)$ 的计算结果：





$$d(4, 9) = \min\{c_{47} + d(7, 9), c_{48} + d(8, 9)\}$$

$$d(5, 9) = \min\{c_{57} + d(7, 9), c_{58} + d(8, 9)\}$$

$$d(6, 9) = \min\{c_{67} + d(7, 9), c_{68} + d(8, 9)\}$$

这一阶段的决策依赖于 $d(7, 9)$ 和 $d(8, 9)$ 的计算，而 $d(7, 9)$ 和 $d(8, 9)$ 可以直接获得（括号中给出了决策产生的状态转移）：

$$d(7, 9) = c_{79} = 7(7 \rightarrow 9)$$

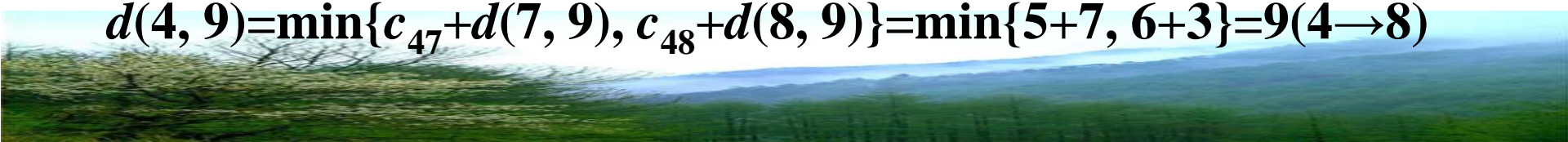
$$d(8, 9) = c_{89} = 3(8 \rightarrow 9)$$

再向前推导，有：

$$d(6, 9) = \min\{c_{67} + d(7, 9), c_{68} + d(8, 9)\} = \min\{6+7, 5+3\} = 8(6 \rightarrow 8)$$

$$d(5, 9) = \min\{c_{57} + d(7, 9), c_{58} + d(8, 9)\} = \min\{8+7, 6+3\} = 9(5 \rightarrow 8)$$

$$d(4, 9) = \min\{c_{47} + d(7, 9), c_{48} + d(8, 9)\} = \min\{5+7, 6+3\} = 9(4 \rightarrow 8)$$





$$d(3, 9) = \min\{c_{35} + d(5, 9), c_{36} + d(6, 9)\} = \min\{4 + 9, 7 + 8\} = 13(3 \rightarrow 5)$$

$$d(2, 9) = \min\{c_{24} + d(4, 9), c_{25} + d(5, 9), c_{26} + d(6, 9)\} = \min\{6 + 9, 7 + 9, 8 + 8\} = 15(2 \rightarrow 4)$$

$$d(1, 9) = \min\{c_{14} + d(4, 9), c_{15} + d(5, 9)\} = \min\{9 + 9, 8 + 9\} = 17(1 \rightarrow 5)$$

$$d(0, 9) = \min\{c_{01} + d(1, 9), c_{02} + d(2, 9), c_{03} + d(3, 9)\} = \min\{4 + 17, 2 + 15, 3 + 13\} = 16(0 \rightarrow 3)$$

最后，得到最短路径为 $0 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 9$ ，长度为16。





下面考虑多段图的最短路径问题的填表形式。

用一个数组 $\text{cost}[n]$ 作为存储子问题解的表格， $\text{cost}[i]$ 表示从顶点 i 到终点 $n-1$ 的最短路径，数组 $\text{path}[n]$ 存储状态， $\text{path}[i]$ 表示从顶点 i 到终点 $n-1$ 的路径上顶点 i 的下一个顶点。则：

$$\text{cost}[i] = \min\{c_{ij} + \text{cost}[j]\} \quad (i \leq j \leq n \text{ 且 顶点 } j \text{ 是 顶点 } i \text{ 的 邻接点})$$

(式7)

$$\text{path}[i] = \text{使 } c_{ij} + \text{cost}[j] \text{ 最小的 } j$$

(式8)



算法——多段图的最短路径

1. 初始化：数组 $cost[n]$ 初始化为最大值，数组 $path[n]$ 初始化为-1；
2. for ($i=n-2$; $i \geq 0$; $i--$)
 - 2.1 对顶点 i 的每一个邻接点 j ，根据式6.7计算 $cost[i]$;
 - 2.2 根据式6.8计算 $path[i]$;
3. 输出最短路径长度 $cost[0]$;
4. 输出最短路径经过的顶点：
 - 4.1 $i=0$
 - 4.2 循环直到 $path[i]=n-1$
 - 4.2.1 输出 $path[i]$;
 - 4.2.2 $i=path[i]$;

算法6.2主要由三部分组成：第一部分是初始化部分，其时间性能为 $O(n)$ ；第二部分是依次计算各个顶点到终点的最短路径，由两层嵌套的循环组成，外层循环执行 $n-1$ 次，内层循环对所有出边进行计算，并且在所有循环中，每条出边只计算一次。假定图的边数为 m ，则这部分的时间性能是 $O(m)$ ；第三部分是输出最短路径经过的顶点，其时间性能是 $O(n)$ 。所以，时间复杂性为 $O(n+m)$ 。

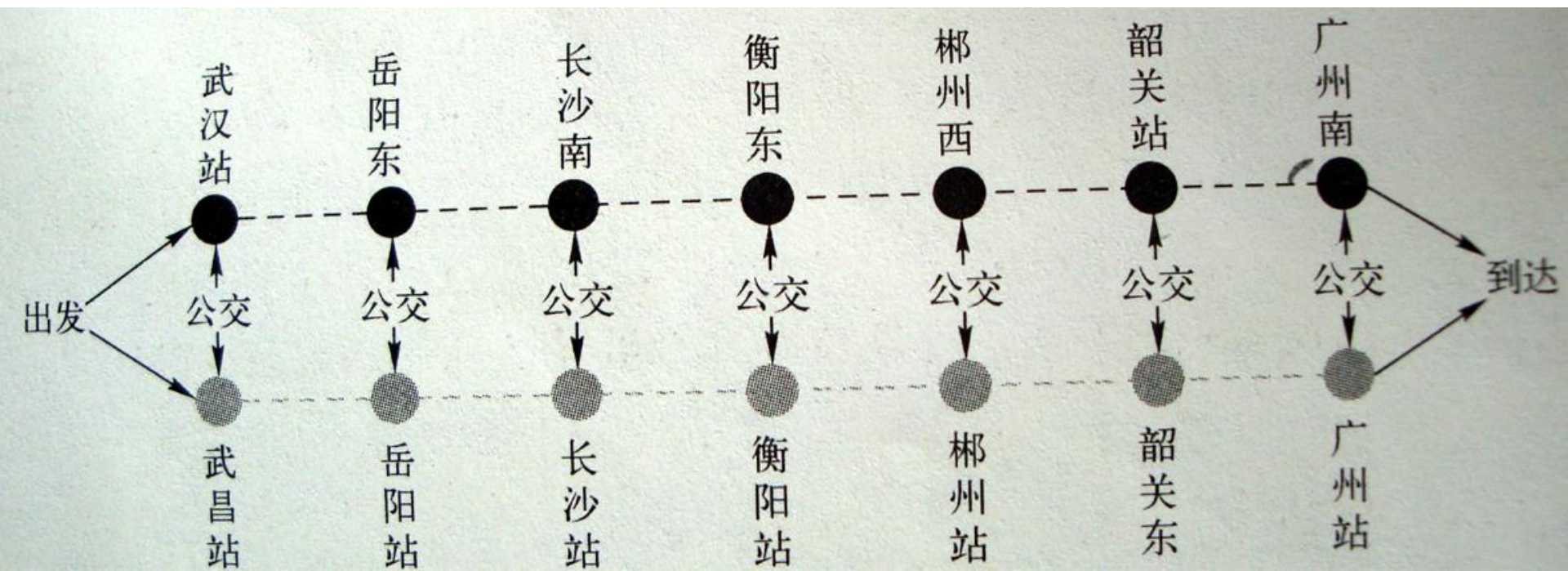


2009年12月26日，耗时4年半，耗资1200亿元的武广客运专线（即高铁）开通营运，该线路沿着原来的京广铁路武汉——广州段途径的城市行走，并沿途新建所有车站。这样沿途的每座城市就有两个车站，每座城市的两个车站间都有公交系统相连接。这样，乘客从武汉到广州就可以选择两条线路的任意一条，并且可在任意中间站选择换到另一条线路上。

显然，走高铁更快。但由于速度快，窗外的风景难以欣赏，而原京广线可以欣赏，如果欣赏风景那令人愉悦的感觉可以与时间相抵，那么走原京广线反而更快。（例如：长沙→衡阳，原：90分钟，新：50分钟，但美景抵消45分钟，则原：45分钟）

如果每对相邻城市间走高铁和普铁的时间已知，走普铁的心理时间抵消值及公交车的时间也已知，乘客要做的决策是如何选择行走线路而获得最佳的心理时间成本。如何规划？

若用蛮力法求解，每个城市有2种选择， 2^6 种可能。如果n很大呢？动态规划策略。





流水装配线问题

本章最前面的例子通常以一个更为一般化的流水线问题来表述：一家汽车装配厂有两条流水线(这个问题也可以扩展到 3 条或 3 条以上流水线的情况)，每条流水线有 n 个工作站，分别为 $S_{1,1}, \dots, S_{1,n}$ 和 $S_{2,1}, \dots, S_{2,n}$ 。两条流水线上相应位置上的工作站 $S_{1,j}$ 、 $S_{2,j}$ 所完成的任务一样，但完成的时间不同，分别为 $c_{1,j}$ 、 $c_{2,j}$ 。部件发射到流水线的时间分别为 $c_{1,0}$ 、 $c_{2,0}$ ，流出流水线的时间分别为 x_1 、 x_2 。如图 4-3 所示。

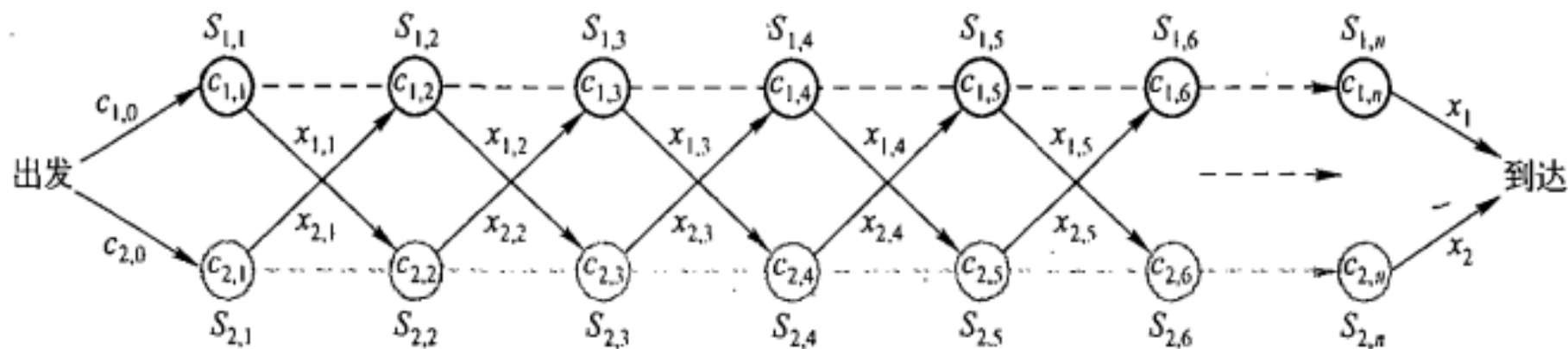


图 4-3 汽车装配厂流水线



在经过一个工作站 $S_{i,j}$ 后，一个部件可以：

- 1) 前进到同一条流水线上的下一个工作站，不产生任何流动成本。
- 2) 转移到另一条流水线上的下一个工作站，产生流动成本 $x_{i,j}$ 。

我们的问题是，给定上述所有的成本，我们应该选择流水线 1、2 上的哪些工作站才能使部件从进入流水线到流出流水线的时间最短？

一种很直接的解答是尝试所有的可能，然后选择时间最短的方案。显然这种方式的时间复杂性为指数级 2^n （以工作站数量作为输入参数）。一般的分而治之的办法也不行

那么办法是什么呢？你猜到了吗？动态规划。

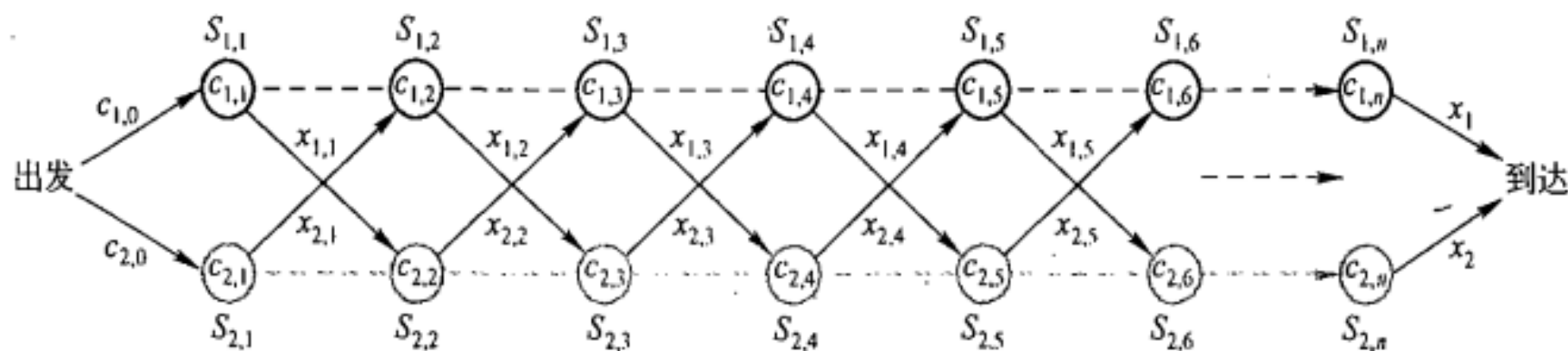


图 4-3 汽车装配厂流水线



下面我们看一下这个问题是否适合使用动态规划，首先考虑最优解的结构。我们考虑从出发到站点 $S_{1,j}$ 的最快的路线，可以分为两种情况：

- 1) 如果 $j = 1$ ，则只需要决定经过工作站 $S_{1,1}$ 需要使用多长时间即可。
- 2) 如果 $j \geq 2$ ，则到达 $S_{1,j}$ 又有两条路线选择：
 - 从 $S_{1,j-1}$ 过来，直接进入 $S_{1,j}$ 。
 - 从 $S_{2,j-1}$ 过来，经转换进入 $S_{1,j}$ 。

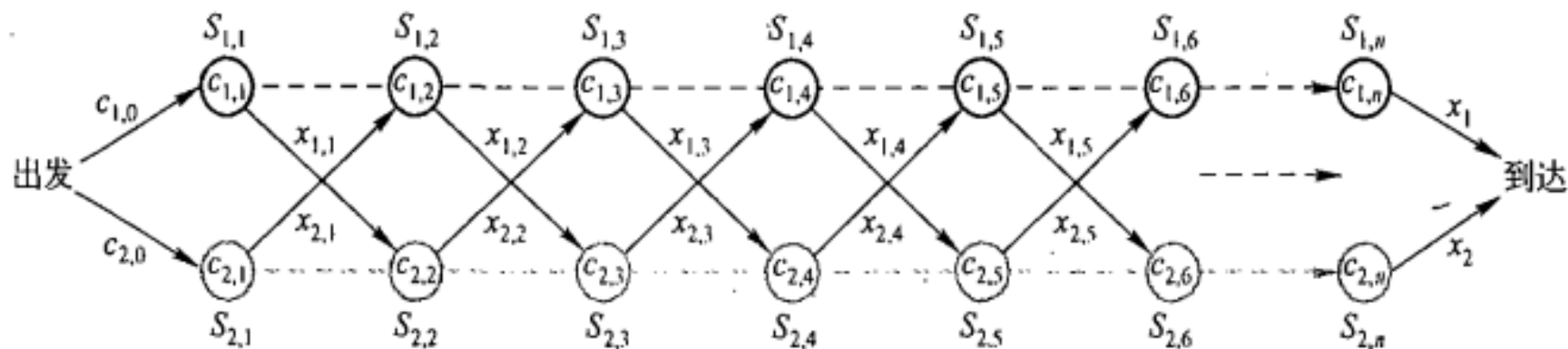


图 4-3 汽车装配厂流水线

假定最快的路线是由 $S_{1,j-1}$ 过来，则我们注意到：从出发到 $S_{1,j-1}$ 的路线必须是最快的路线；否则，我们可以使用另一条更快的路线到达 $S_{1,j-1}$ 再进入 $S_{1,j}$ ，从而获得一条更快的从出发到站点 $S_{1,j}$ 路线。

假定最快的路线是由 $S_{2,j-1}$ 过来，则我们注意到：从出发到 $S_{2,j-1}$ 的路线必须是最快的路线；否则，我们可以使用另一条更快的路线到达 $S_{2,j-1}$ 再转换进入 $S_{1,j}$ ，从而获得一条更快的从出发到站点 $S_{1,j}$ 路线。

一般来说，一个问题的最优解决方案里面包含了对子问题的最优解，即从出发到 $S_{1,j}$ 的最优方案包括了从出发到 $S_{1,j-1}$ 或者 $S_{2,j-1}$ 的最优方案。这就是我们所说的最优子结构。

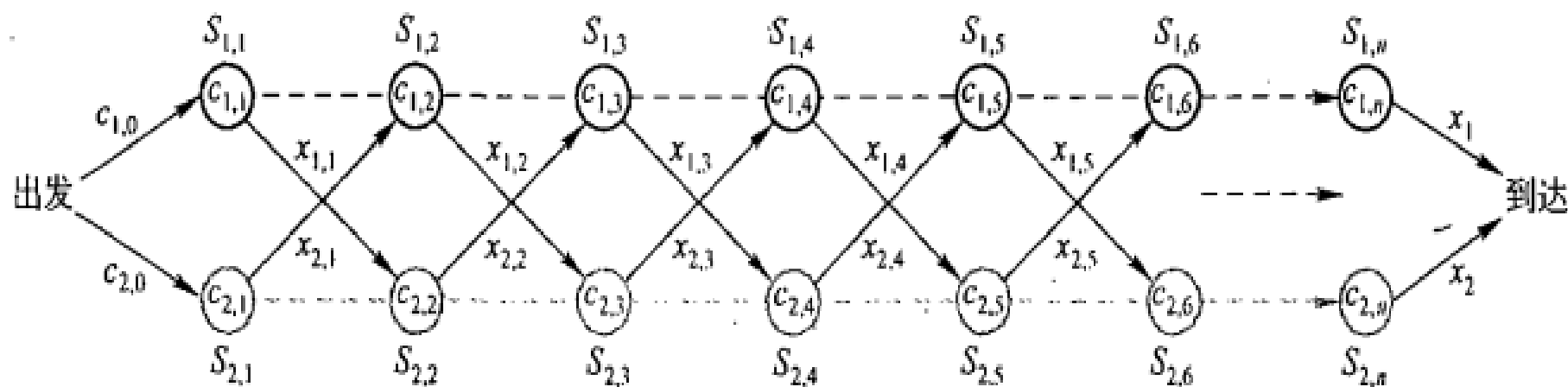


图 4-3 汽车装配厂流水线



有了最优子结构，我们就可以从子问题的解答构建原问题的解答。

到达 $S_{1,j}$ 最快的路线为以下二居其一：

- 到达 $S_{1,j-1}$ 最快的路线，然后直接进入 $S_{1,j}$ 。
- 或者到达 $S_{2,j-1}$ 最快的路线，然后从线 2 转换到线 1 进入 $S_{1,j}$ 。

对称地我们有到达 $S_{2,j}$ 最快的路线必为以下的二居其一：

- 到达 $S_{2,j-1}$ 最快的路线，然后直接进入 $S_{2,j}$ 。
- 或者到达 $S_{1,j-1}$ 最快的路线，然后从线 1 转换到线 2 进入 $S_{2,j}$ 。

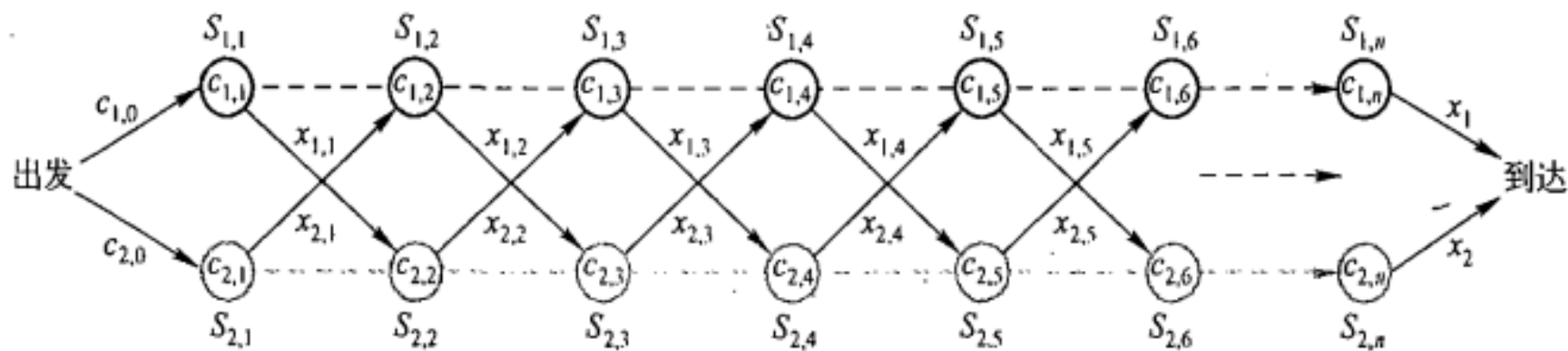


图 4-3 汽车装配厂流水线

因此，寻找到达 $S_{1,j}$ 和 $S_{2,j}$ 的最快路线就转换为递归寻找到达 $S_{1,j-1}$ 和 $S_{2,j-1}$ 的最快路线。

如此，我们获得递归解决方案如下：设 $t_i[j]$ = 通过 $S_{i,j}$ ($i = 1, 2$ 和 $j = 1, \dots, n$) 的最短时间， t^* = 通过整个流水线的最短时间，则有：

$$t^* = \min(t_1[n] + x_1, t_2[n] + x_2)$$

$$t_1[1] = c_{1,0} + c_{1,1}$$

$$t_2[1] = c_{2,0} + c_{2,1}$$

并且，对于 $j = 2, \dots, n$ ，我们有：

$$t_1[j] = \min(t_1[j-1] + c_{1,j}, t_2[j-1] + x_{2,j-1} + c_{1,j})$$

$$t_2[j] = \min(t_2[j-1] + c_{2,j}, t_1[j-1] + x_{1,j-1} + c_{2,j})$$

这里 $t_i[j]$ 给出的就是最优解决方案的值（最优方案需要的最短时间）。

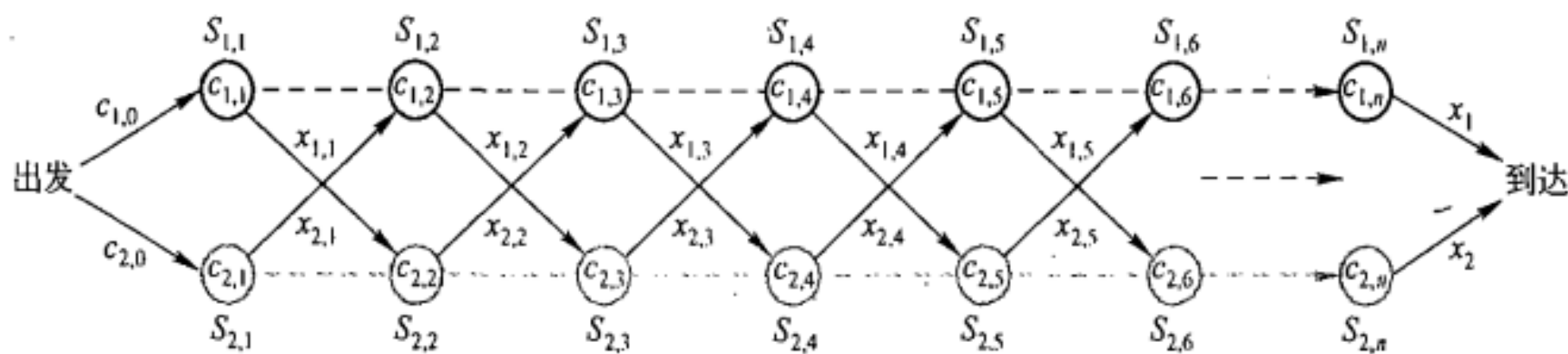


图 4-3 汽车装配厂流水线



但显然，我们不仅要知道最优方案需要多少时间通过整条流水线，还需要知道这个最优方案到底是什么，到底通过哪些工作站。那么如何构建这个最优方案呢？

很简单，只需要再增加一个变量即可。设 $l_i[j]$ = 到达 $S_{i,j}$ 最快路线上工作站 $j-1$ 所处的线路 (1 或 2)，即在通过 $S_{i,j}$ 的最快路线上，工作站 $S_{l_i[j], j-1}$ 在工作站 $S_{i,j}$ 之前，这里的 $i = 1, 2, j = 2, \dots, n$ 。再设 l^* = 最后站点的线路号。

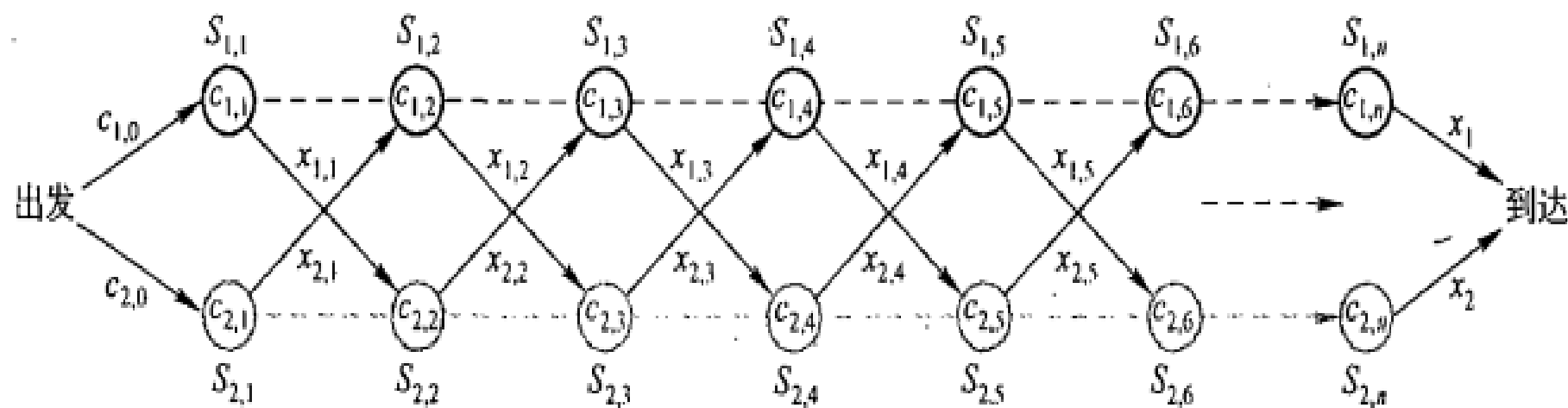


图 4-3 汽车装配厂流水线

表 4-1 通过工作站 $S_{i,j}$ ($i = 1, 2, j = 1, \dots, n$) 的最短时间

| j | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----------|---|---|----|----|----|----|----|
| $t_1[j]$ | 4 | 9 | 17 | 24 | 22 | 24 | 30 |
| $t_2[j]$ | 6 | 8 | 14 | 17 | 20 | 23 | 31 |

表 4-2 最快通过流水线使用的工作站

| j | 2 | 3 | 4 | 5 | 6 | 7 |
|----------|---|---|---|---|-----|-----|
| $t_1[j]$ | 1 | 1 | 1 | 2 | 1/2 | 1/2 |
| $t_2[j]$ | 2 | 2 | 2 | 2 | 2 | 2 |

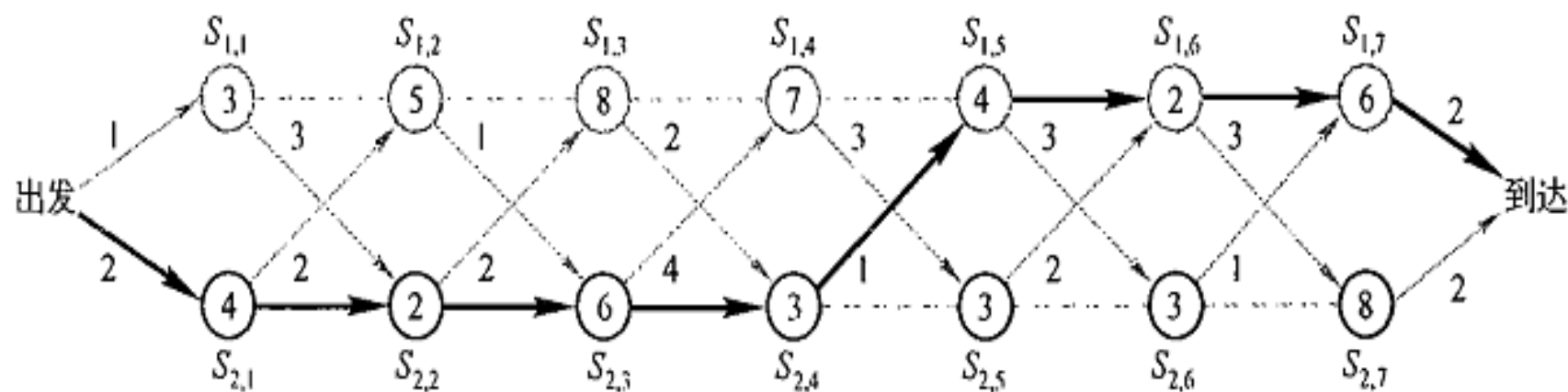


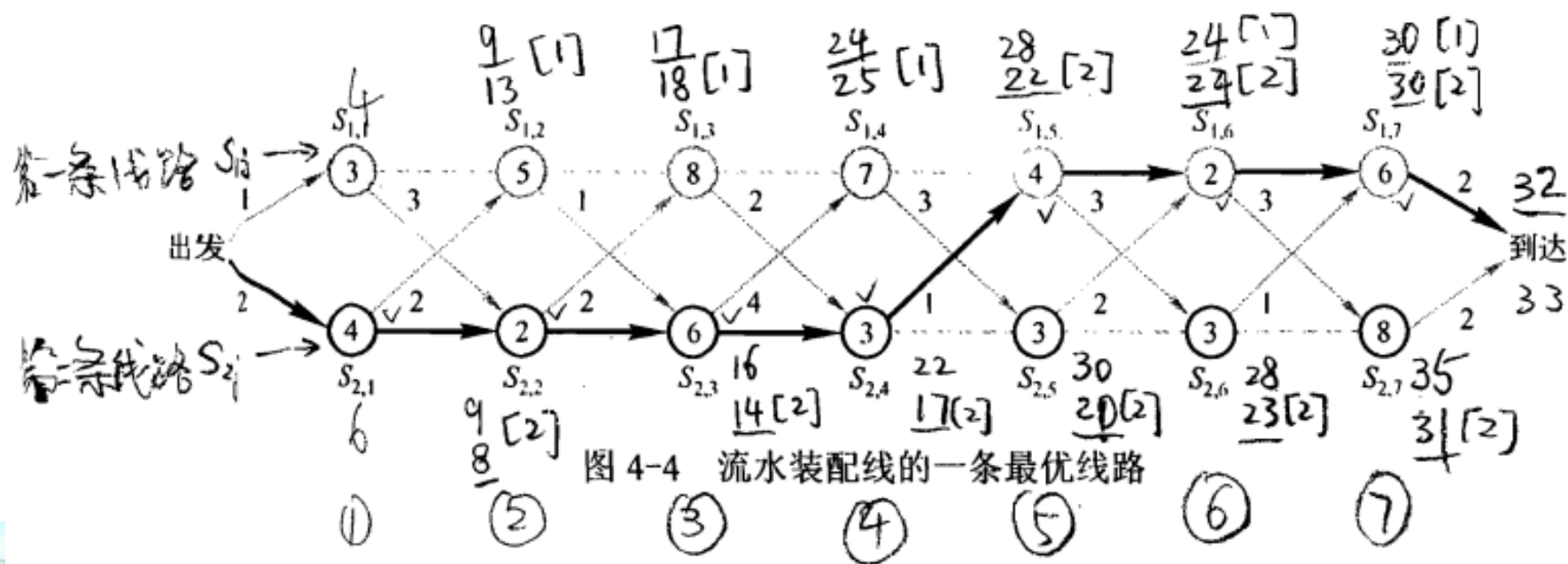
图 4-4 流水装配线的一条最优线路

表 4-1 通过工作站 $S_{i,j}$ ($i=1,2, j=1,\dots,n$) 的最短时间

| j | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----------|---|---|----|----|----|----|----|
| $t_1[j]$ | 4 | 9 | 17 | 24 | 22 | 24 | 30 |
| $t_2[j]$ | 6 | 8 | 14 | 17 | 20 | 23 | 31 |

表 4-2 最快通过流水线使用的工作站

| j | 2 | 3 | 4 | 5 | 6 | 7 |
|----------|---|---|---|---|-----|-----|
| $i_1[j]$ | 1 | 1 | 1 | 2 | 1/2 | 1/2 |
| $i_2[j]$ | 2 | 2 | 2 | 2 | 2 | 2 |





组合问题中的动态规划法

0/1背包问题



0/1背包问题

在0/1背包问题中，物品*i*或者被装入背包，或者不被装入背包，设 x_i 表示物品*i*装入背包的情况，则当 $x_i=0$ 时，表示物品*i*没有被装入背包， $x_i=1$ 时，表示物品*i*被装入背包。根据问题的要求，有如下约束条件和目标函数：

$$\begin{cases} \sum_{i=1}^n w_i x_i \leq C \\ x_i \in \{0,1\} \quad (1 \leq i \leq n) \end{cases} \quad (\text{式9})$$

$$\max \sum_{i=1}^n v_i x_i \quad (\text{式10})$$

于是，问题归结为寻找一个满足约束条件式6.9，并使目标函数式6.10达到最大的解向量 $X=(x_1, x_2, \dots, x_n)$ 。

证明0/1背包问题满足最优性原理。

设 (x_1, x_2, \dots, x_n) 是所给0/1背包问题的一个最优解，则 (x_2, \dots, x_n) 是下面一个子问题的最优解：

$$\begin{cases} \sum_{i=2}^n w_i x_i \leq C - w_1 x_1 \\ x_i \in \{0,1\} \quad (2 \leq i \leq n) \end{cases} \quad \max \sum_{i=2}^n v_i x_i$$

如若不然，设 (y_2, \dots, y_n) 是上述子问题的一个最优解，则

$$\sum_{i=2}^n v_i y_i > \sum_{i=2}^n v_i x_i \quad w_1 x_1 + \sum_{i=2}^n w_i y_i \leq C$$

因此，

$$v_1 x_1 + \sum_{i=2}^n v_i y_i > v_1 x_1 + \sum_{i=2}^n v_i x_i = \sum_{i=1}^n v_i x_i$$

这说明 (x_1, y_2, \dots, y_n) 是所给0/1背包问题比 (x_1, x_2, \dots, x_n) 更优的解，从而导致矛盾。



0/1背包问题可以看作是决策一个序列 (x_1, x_2, \dots, x_n) ，对任一变量 x_i 的决策是决定 $x_i=1$ 还是 $x_i=0$ 。在对 x_{i-1} 决策后，已确定了 (x_1, \dots, x_{i-1}) ，在决策 x_i 时，问题处于下列两种状态之一：

- (1) 背包容量不足以装入物品 i ，则 $x_i=0$ ，背包不增加价值；
- (2) 背包容量可以装入物品 i ，则 $x_i=1$ ，背包的价值增加了 v_i 。

这两种情况下背包价值的最大者应该是对 x_i 决策后的背包价值。令 $V(i, j)$ 表示在前 i ($1 \leq i \leq n$) 个物品中能够装入容量为 j ($1 \leq j \leq C$) 的背包中的物品的最大值，则可以得到如下动态规划函数：

$$V(i, 0) = V(0, j) = 0 \quad (\text{式11})$$

$$V(i, j) = \begin{cases} V(i-1, j) & j < w_i \\ \max\{V(i-1, j), V(i-1, j-w_i) + v_i\} & j > w_i \end{cases} \quad (\text{式12})$$

这个方程非常重要，基本上所有跟背包相关的问题的方程都是由它衍生出来的。所以有必要将它解释一下：“将前 i 件物品放入容量为 v 的背包中”这个子问题，若只考虑第 i 件物品的策略（放或不放），那么就可以转化为一个只牵扯前 $i-1$ 件物品的问题。如果不放第 i 件物品，那么问题就转化为“前 $i-1$ 件物品放入容量为 v 的背包中”，价值为 $f[v]$ ；如果放第 i 件物品，那么问题就转化为“前 $i-1$ 件物品放入剩下的容量为 $v-c$ 的背包中”，此时能获得的最大价值就是 $f[v-c]$ 再加上通过放入第 i 件物品获得的价值。



例如，有5个物品，其重量分别是{2, 2, 6, 5, 4}，价值分别为{6, 3, 5, 4, 6}，背包的容量为10。

根据动态规划函数，用一个 $(n+1) \times (C+1)$ 的二维表V， $V[i][j]$ 表示把前i个物品装入容量为j的背包中获得的最大价值。

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | | |
|-----------------|---|---|---|---|---|---|---|----|----|----|----|----|---------|--|
| 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| $w_1=2 \ v_1=6$ | 1 | 0 | 0 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | $x_1=1$ | |
| $w_2=2 \ v_2=3$ | 2 | 0 | 0 | 6 | 6 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | $x_2=1$ | |
| $w_3=6 \ v_3=5$ | 3 | 0 | 0 | 6 | 6 | 9 | 9 | 9 | 9 | 11 | 11 | 14 | $x_3=0$ | |
| $w_4=5 \ v_4=4$ | 4 | 0 | 0 | 6 | 6 | 9 | 9 | 9 | 10 | 11 | 13 | 14 | $x_4=0$ | |
| $w_5=4 \ v_5=6$ | 5 | 0 | 0 | 6 | 6 | 9 | 9 | 12 | 12 | 12 | 15 | 15 | $x_5=1$ | |

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





例如，有5个物品，其重量分别是{2, 2, 6, 5, 4}，价值分别为{6, 3, 5, 4, 6}，背包的容量为10。

根据动态规划函数，用一个 $(n+1) \times (C+1)$ 的二维表V， $V[i][j]$ 表示把前i个物品装入容量为j的背包中获得的最大价值。

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|----|
| $w_1=2 \ v_1=6$ $w_2=2 \ v_2=3$ $w_3=6 \ v_3=5$ $w_4=5 \ v_4=4$ $w_5=4 \ v_5=6$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 0 | 0 | | | | | | | | | |
| | 2 | 0 | 0 | | | | | | | | | |
| | 3 | 0 | 0 | | | | | | | | | |
| | 4 | 0 | 0 | | | | | | | | | |
| | 5 | 0 | 0 | | | | | | | | | |

Diagram illustrating the dynamic programming table V for the knapsack problem. The table has 6 rows (i=0 to 5) and 13 columns (j=0 to 10). The first row (i=0) contains all zeros. The subsequent rows (i=1 to 5) show the maximum value for each capacity j. Arrows indicate the recurrence relation: for $j < w_i$, the value is $V(i-1, j)$; for $j > w_i$, the value is $\max\{V(i-1, j), V(i-1, j-w_i) + v_i\}$. The diagram shows the calculation of $V(5, 10)$ using the recurrence relation, with arrows pointing to the relevant cells in the previous row and the current row.

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



按下述方法来划分阶段:


第一阶段, 只装入前1个物品, 确定在各种情况下的背包能够得到的最大价值;


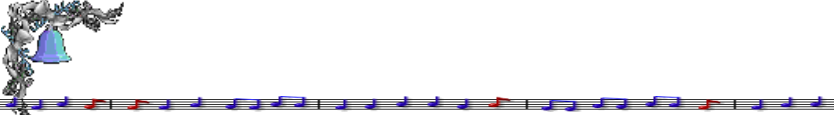
第二阶段, 只装入前2个物品, 确定在各种情况下的背包能够得到的最大价值;

依此类推...直到第 n 个阶段。

最后, $V(n, C)$ 便是在容量为 C 的背包中装入 n 个物品时取得的最大价值。


为了确定装入背包的具体物品, 从 $V(n, C)$ 的值向前推, 如果 $V(n, C) > V(n-1, C)$, 表明第 n 个物品被装入背包, 前 $n-1$ 个物品被装入容量为 $C - w_n$ 的背包中; 否则, 第 n 个物品没有被装入背包, 前 $n-1$ 个物品被装入容量为 C 的背包中。

$$x_i = \begin{cases} 0 & V(i, j) = V(i-1, j) \\ 1, & j = j - w_i \quad V(i, j) > V(i-1, j) \end{cases}$$





设 n 个物品的重量存储在数组 $w[n]$ 中，价值存储在数组 $v[n]$ 中，背包容量为 C ，数组 $V[n+1][C+1]$ 存放迭代结果，其中 $V[i][j]$ 表示前 i 个物品装入容量为 j 的背包中获得的最大价值，数组 $x[n]$ 存储装入背包的物品，动态规划法求解0/1背包问题的算法如下：

算法6.3——0/1背包问题



```
int KnapSack(int n, int w[ ], int v[ ])
{
    for (i=0; i<=n; i++) //初始化第0列
        V[i][0]=0;
    for (j=0; j<=C; j++) //初始化第0行
        V[0][j]=0;
    for (i=1; i<=n; i++) //计算第i行，进行第i次迭代
        for (j=1; j<=C; j++)
            if (j<w[i])
```





算法6.3——0/1背包问题

C++描述

```
V[i][j]=V[i-1][j];  
    else  
V[i][j]=max(V[i-1][j], V[i-1][j-w[i]]+v[i]);  
    j=C; //求装入背包的物品  
    for (i=n; i>0; i--)  
    {  
        if (V[i][j]>V[i-1][j]) {  
            x[i]=1;  
            j=j-w[i];  
        }  
        else x[i]=0;  
    }  
    return V[n][C]; //返回背包取得的最大价值  
}
```

在算法6.3中，第一个for循环的时间性能是 $O(n)$ ，第二个for循环的时间性能是 $O(C)$ ，第三个循环是两层嵌套的for循环，其时间性能是 $O(n \times C)$ ，第四个for循环的时间性能是 $O(n)$ ，所以，算法6.3的时间复杂性为 $O(n \times C)$ 。



适合于动态规划法的标准问题

- 整个问题的求解可以划分为若干个阶段的一系列决策过程；
- 每个阶段有若干可能状态；
- 一个决策将你从一个阶段的一种状态带到下一个阶段的某种状态；
- 在任一个阶段，最佳的决策序列和该阶段以前的决策无关；
- 各阶段状态之间的转换有明确定义的费用，而且在选择最佳决策时有递归关系


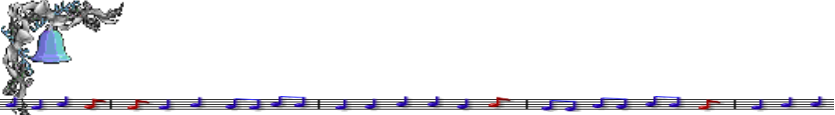





Algorithm

- Greedy. Build up a solution incrementally (逐渐地), myopically (目光短浅地) optimizing some local criterion.
- Divide-and-conquer. Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.
- Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.





标准分治、动态规划和贪婪选择的比较

- 标准分治、动态规划和贪婪选择可以说是孙子兵法里面的下、中、上三策。
 - 标准分治虽然将大问题分解为小问题，但每个小问题都需要一个一个解决，相当于逢城必攻，属于下策；
 - 动态规划则聪明地发现很多子问题相同，并不需要重新解决，即不对每个城池进行攻克，从而节省兵力和精力，但仍然需要攻克子问题中的相当部分，属于中策；
 - 而贪婪算法将子问题限于一个，将攻城数量减少到了最低，属于上策。
- 



- 从另一方面来看，三种策略都是为了在求解问题时使成本尽量低。因此，三种策略的目标一致。
- 一般说来，动态规划和贪婪选择要求获得一个最优解，因此，二者之间有很大的相似性。





贪婪选择与动态规划

- 贪婪选择属性是指：一个全局最优解决方案可以通过一个局部最优的选择（贪婪选择）获得。动态规划进行了贪婪选择了吗？
- 动态规划的每一步都需要做出一个选择，只不过此时我们的选择不是在不知道子问题解的情况下做出的，而是在子问题的解已经解出来的基础上进行的。我们选择的是最好的分解，因此是贪婪选择。





- 动规法是先解决子问题，自底向上，每次选择都是正确的；
- 而贪心法实则解决子问题之前做出选择，希望做出的选择正确，是自顶向下。
- 不管是先做出选择还是后做出选择，都是试图做出最好的选择！





标准分治、动态规划、贪婪选择三种策略的比较



| | 标准分治 | 动态规划 | 贪婪选择 |
|----------|-----------|-----------|-----------|
| 适用类型 | 通用问题 | 很多子问题重复 | 只有一个子问题 |
| 子问题结构 | 每个子问题不同 | 很多子问题重复 | 只有一个子问题 |
| 最优子结构 | 不用 | 必须满足 | 必须满足 |
| 子问题数 | 全部子问题都要解决 | 全部子问题都要解决 | 只要解决一个子问题 |
| 子问题在最优解里 | 全部 | 部分 | 部分 |
| 选择与求解次序 | 先选择后解决子问题 | 先解决子问题后选择 | 先选择后解决子问题 |



思考

- 假设你正在管理一条公路的广告牌建设，这条路从西到东 M 英里。广告牌可能的地点假设为 $x_1, x_2, x_3 \dots x_n$ ，处于 $[0, M]$ 中。若在 x_i 放一块广告牌，可以得到 $r_i > 0$ 的收益。
- 国家公路局规定，两块广告牌相对不能小于或等于5英里之内。
- 如何找一组地点使你的总收益达到最大？





解答

- 假设 $M=20$, $n=4$, $\{x_1, x_2, x_3, x_4\}=\{6, 7, 12, 14\}$
且 $\{r_1, r_2, r_3, r_4\}=\{5, 6, 5, 1\}$
最优解: $\{x_1, x_3\}$
- 令 $e(j)$ 表示编号比 j 小且距 x_j 大于 5 英里的最东边的地点
- 令 $OPT(j)$ 表示从 x_1, \dots, x_j 中地点的最优子集得到的收益
- $OPT(j)=\max(r_j + OPT(e(j)), OPT(j-1))$





假设 $M=20$, $n=4$, $\{x_1, x_2, x_3, x_4\} = \{6, 7, 12, 14\}$ 且
 $\{r_1, r_2, r_3, r_4\} = \{5, 6, 5, 1\}$

$$OPT(j) = \max (r_j + OPT(e(j)), OPT(j-1))$$

| 地点 编号 | 0 | 1 | 2 | 3 | 4 |
|-----------|---|---|---|----|----|
| 位置 x | 0 | 6 | 7 | 12 | 14 |
| 收益 r | 0 | 5 | 6 | 5 | 1 |
| $e(j)$ | 0 | | | | |
| $OPT(j)$ | 0 | | | | |
| 最优解 位置 | 0 | | | | |



习题

- 有3个物品，其体积分别为 $(4, 3, 2)$; 价值分别为 $(36, 24, 20)$, 背包容量为6, 请写出求最优解的过程。
- 分别用蛮力法、贪心法（按单位价值最大的贪心策略解）、动态规划法、回溯法、分支限界法解如上的0/1背包问题：

