第六章　动态规划

Fibonacci数列

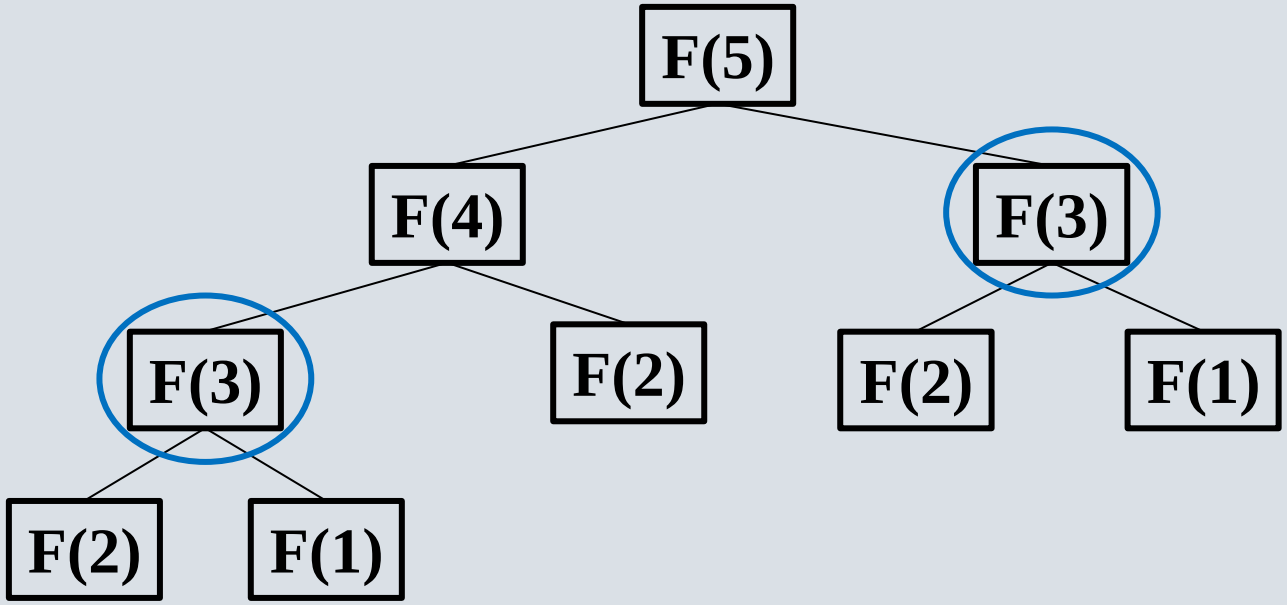
$$F(n)=\begin{cases}1 & n=1,2\\ F(n-1)+F(n-2) & n\geq 3\end{cases}$$

递归算法的伪代码：

```
F(n)
1.    if n=1 or n=2  then return 1
2.    else return F(n-1)+F (n-2)
```

- 二叉树高度是n-1
- 高度为k的二叉树最多由2^k-1个子结点
- 时间复杂度：$O(2^n)$

计算F(5)



多次重复计算！
如何避免？

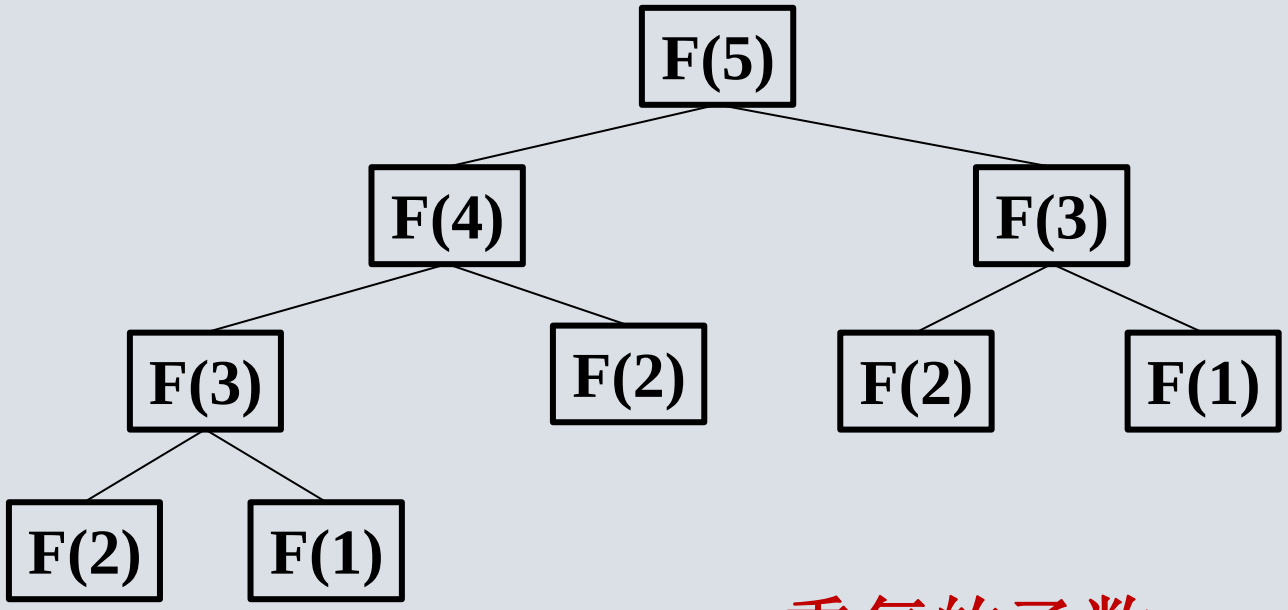
改进的想法

备忘录：

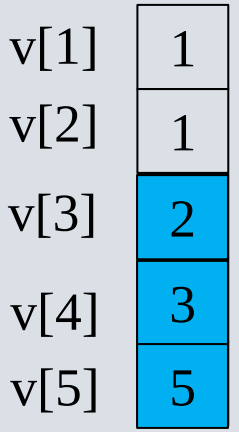
- 当$F_1(i)$被计算后，保存它的值；
- 当再次计算$F_1(i)$时，只需要从内存中取出即可。

```
F1(n)
1.  if v[n] <0, then
2.      v[n] ← F1(n-1)+F1(n-2)
3.  return v(n)
```

```
Main()
1.  v[1] = v[2]←1
2.  for i ← 3 to n do
3.      v[i] = -1
4.  output F1(n)
```

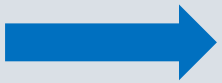


重复的函数调用



自顶向下
Top-down

- 递归



自底向上
Top-down

- 递推

递推方式节约大量

无用递归调用

```
F(n)
1.      A[1]=A[2] ← 1
2.      for i ← 3 to n  do
3.              A[i] ← A[i -1]+A[i -2]
4.      return A[n]
```

$O(n)$

分治递归：

```
F(n)
1.      if n=1 or n=2  then
2.              return 1
3.      else
4.              return F(n-1)+F (n-2)
```

效率低

动态规划：

```
F(n)
1.      A[1]=A[2] ← 1
2.      for i ← 3 to n  do
3.              A[i] ← A[i -1]+A[i -2]
4.      return A[n]
```

高效！
时间复杂性$O(n)$

用空间换时间

动态规划方法总结

- 写出一个递归公式，这个公式给出了问题与其子问题的解之间的关系；
- 用表（通常用数组记录子问题的解，以便保存和以后的检索）从最简单问题的解填起，以自底向上的方式填表。
  - ✓ 这就保证了，求解一个子问题的时候，所有与此相关的子问题，都可以从表中直接取出，而不必重新计算。

由于20世纪40年代末期，还没有出现计算机，这个动态填表的方法称为动态规划。

- 动态规划（dynamic programming）是运筹学的一个分支，20世纪50年代初美国数学家 R. E. Bellman等人在研究多阶段决策过程multistep decision process的优化问题时，提出了著名的最优化原理，把多阶段过程转化为一系列单阶段问题，利用各阶段之间的关系，逐个求解，创立了解决这类过程优化问题的新方法——动态规划。
- 多阶段决策问题：求解问题可以划分为一系列互相联系的阶段，在每个阶段都需要做出决策，且一个阶段决策的选择会影响下一个阶段的决策，从而影响整个过程的活动路线，求解的目标是选择各个阶段的决策使整个过程达到最优。

- 动态规划主要用于求解以时间划分阶段的动态过程的优化问题，但一些与时间无关的静态规划（如线性规划、非线性规划），只要人为地引进时间因素，把它视为多阶段决策过程，也可以用动态规划方法方便地求解。
- 动态规划是考察问题的一种途径，或者求解某类问题的一种方法。
- 动态规划问世以来，在经济管理、生产调度、工程技术和最优控制等方面得到了广泛的应用，例如最短线路、库存管理、资源分配、设备更新、排序、装载等问题，用动态规划方法比用其他方法求解更方便。

动态规划总体思想

- 动态规划算法于分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，再从子问题的解得到原问题的解；
- 但是经分解得到的子问题往往不是互相独立的，不同子问题的数目常常只有多项式量级，在用分治法求解时，有些子问题被重复计算了很多次；
- 保存已解决的子问题的答案，而在需要时再找出已求得的答案，就可以避免大量重复计算，从而得到多项式时间算法；
- 动态规划用一个表来记录所有已解决的子问题的答案，具体的动态规划算法多种多样，但他们具有相同的填表方式。

回顾分治法的适用条件

- 分治法所能解决的问题一般具有以下几个特征：
  - 该问题的规模缩小到一定的程度就可以容易地解决；
  - 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质；
  - 利用该问题分解出的子问题的解可以合并为该问题的解；

能否利用分治法，完全取决于问题是否具有这条特征，如果具备了前两条特征，而不具备第三条特征，则可以考虑贪心算法或动态规划。

回顾分治法的适用条件

- 分治法所能解决的问题一般具有以下几个特征：
  - 该问题的规模缩小到一定的程度就可以容易地解决；
  - 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质；
  - 利用该问题分解出的子问题的解可以合并为该问题的解；
  - 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

这条特征涉及到分治法的效率，如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然也可以用分治法，但一般用动态规划较好。

动态规划基本步骤

1. 找出最优解的性质，并刻画其结构特征；

2. 递归地定义最优值；

3. 以自底向上的方式计算出最优值；

4. 根据计算最优值时得到的信息，构造最优解。

- 步骤1-3是动态规划算法的基本步骤，对于只需要求出最优值的情形，步骤4可以省略；
- 若需要求出问题的一个最优解，则必须执行步骤4，步骤3中记录的信息是构造最优解的基础。

分治递归

- 递归与分治策略
- Top-down

贪婪算法

- 贪婪性质
- Top-down

动态规划

- 查表方式
- 最优子结构
- Bottom-up

6.1 动态规划的思想方法

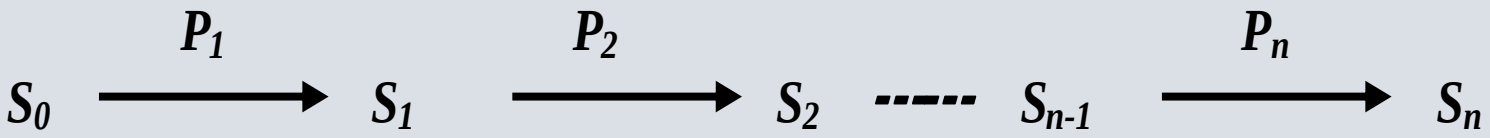
6.1.1 动态规划的最优决策原理

6.1.2 动态规划实例、货郎担问题

6.1.1 动态规划的最优决策原理

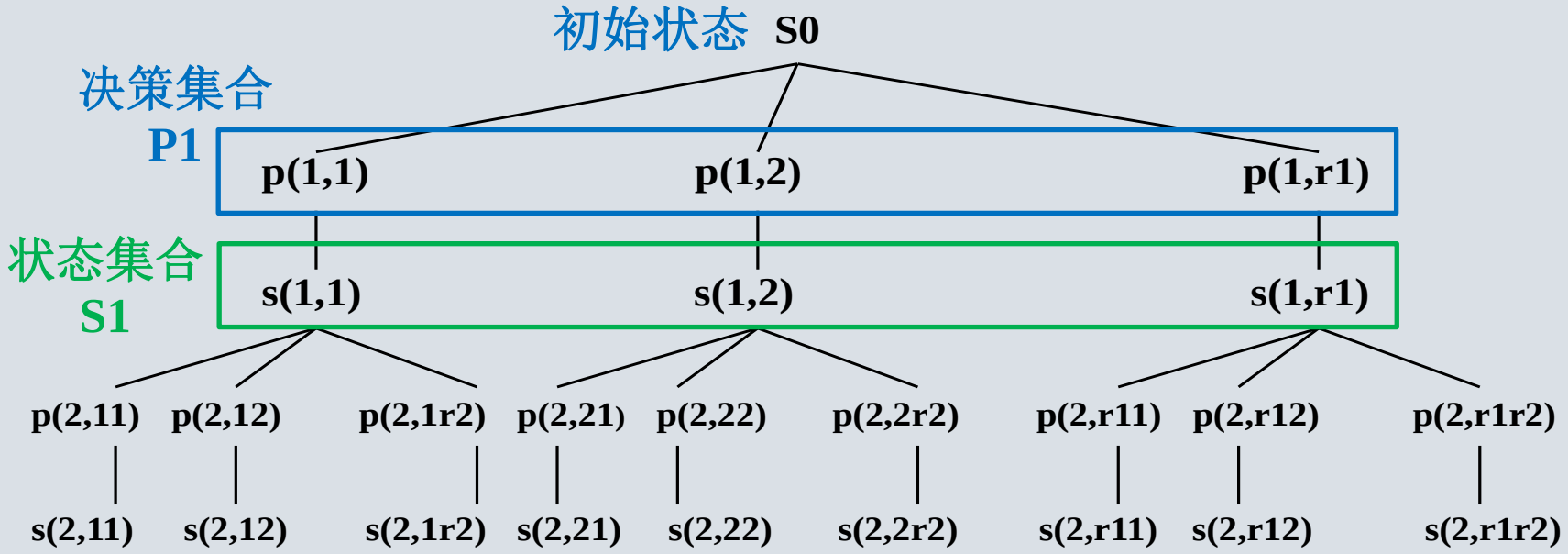
1. 活动过程的分阶段决策

把活动过程划分为若干个阶段，每一阶段的决策，依赖于前一阶段的状态，由决策所采取的动作，使状态发生转移，成为下一阶段的决策依据。

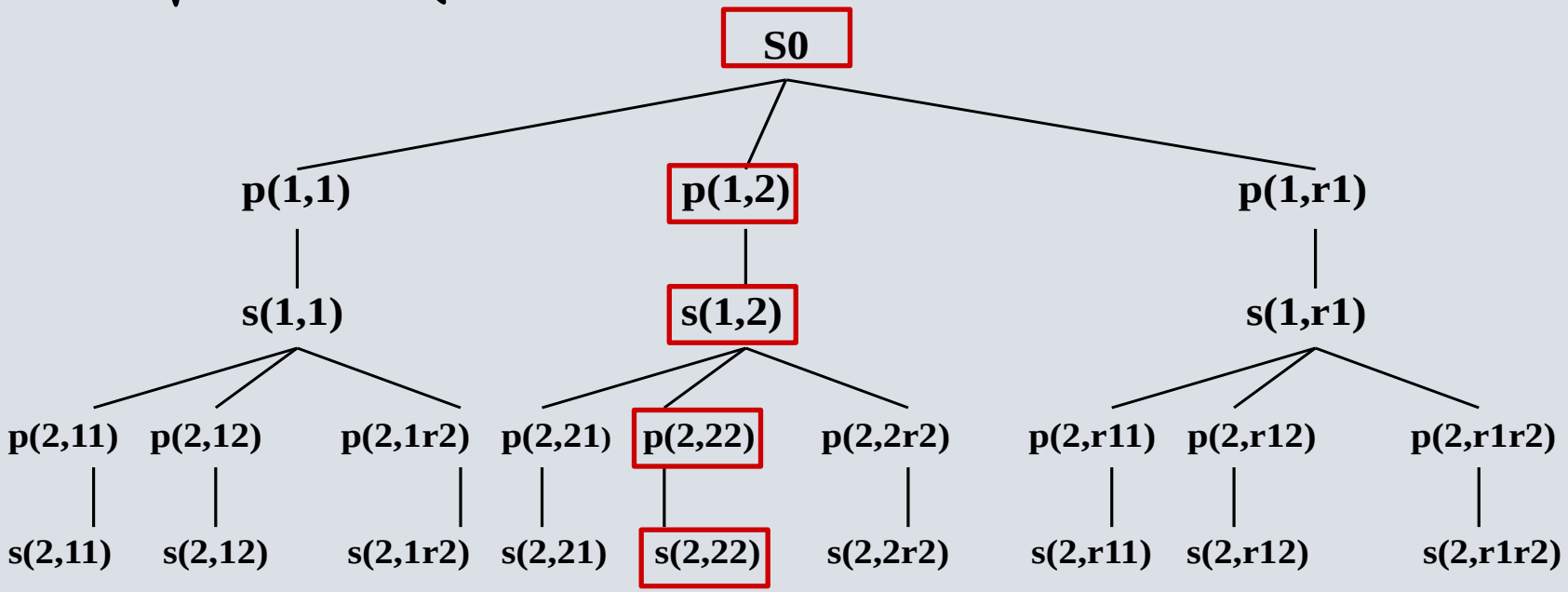


2. 最优性原理

无论过程的初始状态和初始决策是什么，其余决策都必须相对于初始决策所产生的状态，构成一个最优决策序列。



3. 最优决策的形成



令最优状态为 s(2, 22)，由此倒推：

$$s(2,22) \rightarrow p(2,22) \rightarrow s(1,2) \rightarrow p(1,2) \rightarrow s0$$

最优决策序列： $p(2,22) \rightarrow p(1,2)$

状态转移序列： $s0 \rightarrow s(1,2) \rightarrow s(2,22)$

1）最优决策在最后阶段形成，然后向前倒推，直到初始阶段；

2）决策的具体结果及所产生的状态转移，由初始阶段开始进行计算，然后向后递归或迭代，直到最终结果；

3）赖以决策的策略或目标，称为动态规划函数；

4）动态规划函数可以递归地定义，也可以用递推公式表达；

5）整个决策过程，可以递归地进行，或用循环迭代的方法进行。

6.1.2 动态规划实例——货郎担问题

1. 动态规划解货郎担问题的过程

1）货郎担问题实例：

4城市的费用矩阵

$$C=(c_{ij})=\begin{pmatrix} \infty & 3 & 6 & 7 \\ 5 & \infty & 2 & 3 \\ 6 & 4 & \infty & 2 \\ 3 & 7 & 5 & \infty \end{pmatrix}$$

2）动态规划函数：

$d(i, V')$：从顶点 i 出发，经 V' 中各个顶点一次，最终回到初始出发点的最短路径的长度。

设初始出发点为 i，定义动态规划函数：

$$d(i, V-\{i\})=d(i, V')=\min_{k\in V'}\{c_{ik}+d(k, V'-\{k\})\}$$

$$d(k, \emptyset)=c_{ki} \quad k\neq i$$

3）工作过程

①

$$C=(c_{ij})=\begin{pmatrix} \infty & 3 & 6 & 7 \\ 5 & \infty & 2 & 3 \\ 6 & 4 & \infty & 2 \\ 3 & 7 & 5 & \propto \end{pmatrix}$$

② 由城市 1 出发，经城市 2, 3, 4 ，然后返回1的最短路径长度：

$$d(1,\{2,3,4\})=min\{c_{12}+d(2,\{3,4\}),c_{13}+d(3,\{2,4\}),c_{14}+d(4,\{2,3\})\}$$

$$d(2,\{3,4\})=min\{c_{23}+d(3,\{4\}),c_{24}+d(4,\{3\})\}$$

$$d(3,\{2,4\})=min\{c_{32}+d(2,\{4\}),c_{34}+d(4,\{2\})\}$$

$$d(4,\{2,3\})=min\{c_{42}+d(2,\{3\}),c_{43}+d(3,\{2\})\}$$

③ $d(3,\{4\}) = c_{34} + d(4,\emptyset) = c_{34} + c_{41} = 2 + 3 = 5$

$d(4,\{3\}) = c_{43} + d(3,\emptyset) = c_{43} + c_{31} = 5 + 6 = 11$

$d(2,\{4\}) = c_{24} + d(4,\emptyset) = c_{24} + c_{41} = 3 + 3 = 6$

$d(4,\{2\}) = c_{42} + d(2,\emptyset) = c_{42} + c_{21} = 7 + 5 = 12$

$d(2,\{3\}) = c_{23} + d(3,\emptyset) = c_{23} + c_{31} = 2 + 6 = 8$

$d(3,\{2\}) = c_{32} + d(2,\emptyset) = c_{32} + c_{21} = 4 + 5 = 9$

④ $d(2,\{3,4\}) = min\{2+5, 3+11\} = 7$ 2,3,4,1

$d(3,\{2,4\}) = min\{4+6, 2+12\} = 10$ 3,2,4,1

$d(4,\{2,3\}) = min\{7+8, 5+9\} = 14$ 4,3,2,1

$$d(1,\{2,3,4\}) = min\{c_{12} + d(2,\{3,4\}), c_{13} + d(3,\{2,4\}), c_{14} + d(4,\{2,3\})\}$$

$$= min\{3+7, 6+10, 7+14\} = 10$$ 1,2,3,4,1

2. 复杂性分析

令N_i是从顶点 i 出发，返回顶点 i 所需计算的形式为$d(k, V'-\{k\})$ 的个数：

1）开始计算 $d(i, V'-\{i\})$ 时，集合 $V'-i$ 中有 $n-1$ 个顶点；

2）在计算 $d(k, V'-\{k\})$时，集合 $V'-\{k\}$的顶点数目，在不同的决策阶段，分别为 n-2, … , 0；

3）在整个计算中，需要计算大小为 j 的不同顶点集合的个数为C_{n-1}^{j}，j = 0, … , n - 1。总个数为：

$$N_i = \sum_{j=0}^{n-1} C_{n-1}^{j}$$

4）当集合 $V'-\{k\}$中的城市个数为 j 时，为计算$d(k, V'-\{k\})$，需 j 次加法，j - 1 次比较。

5）从顶点 i 出发，经其它城市再回到 i，总的运算时间为：

$$T_i = \sum_{j=0}^{n-1} j \cdot C_{n-1}^{j} < \sum_{j=0}^{n-1} n \cdot C_{n-1}^{j} = n \sum_{j=0}^{n-1} C_{n-1}^{j}$$

由二项式定理：

$$(x+y)^n = \sum_{j=0}^{n} C_n^j x^j y^{n-j}$$

令 $x = y = 1$, 得：$T_i < n \cdot 2^{n-1} = O(n2^n)$

6）动态规划方法求解货郎担问题，总花费 T 为：

$$T = \sum_{i=1}^{n} T_i < n^2 \cdot 2^{n-1} = O(n^2 2^n)$$

6.2 多段图的最短路径问题

6.2.1 多段图的决策过程

6.2.2 多段图动态规划算法的实现

6.2.1 多段图的决策过程

1. 多段图的最短路径问题

1）多段图的定义：

- 有向连通赋权图 $G = <V, E, W>$，顶点集合 V 划分成k 个不相交的子集$V_i (1 \le i \le k, k \ge 2)$，使得 E 中的任一边 (u, v)，必有 $u \in V_i$，$v \in V_{i+m}$，$m \ge 1$，称 G 为多段图。
- 令 $|V_1| = |V_k| = 1$，称 $s \in V_1$ 为源点，$t \in V_k$ 为收点。

2）多段图的最短路径问题：

求从源点到达收点的最小花费的通路

2. 多段图最短路径的决策过程

1）顶点编号：

① 根据多段图的 k 个不相交的子集V_i，把多段图划分为 k 段，每一段包含顶点的一个子集；

② 顶点集合 V 中的所有顶点，按照段的顺序编号：

⑴ 子集中的顶点互不邻接，它们之间的相互顺序无关紧要；

⑵ 顶点个数为 n，源点 s 的编号为 0，收点 t 的编号为 $n-1$；

⑶ 对 E 中的任何一条边 (u, v)，顶点 u 的编号小于顶点 v 的编号。

2）决策过程

① 第一阶段，确定第 $k-1$ 段的所有顶点到达收点 t 的花费最小的通路，及通路上该顶点的前方顶点编号；

② 第二阶段，用第一阶段的信息，确定第 $k-2$ 段的所有顶点到达收点 t 的花费最小的通路，及通路上该顶点的前方顶点编号；

③ 如此依次进行，直到最后确定源点 s 到达收点 t 的花费最小的通路，及通路上该顶点的前方顶点编号；

④ 最后，从源点的信息的前方顶点编号，递推地确定整条路径上的所有顶点编号。

3）动态规划函数

(1) 工作单元描述：

cost[i]：顶点 i 到达收点 t 的最小花费

path[i]：顶点 i 到达收点 t 的前方顶点编号

route[n]：源点到达收点 t 的最短通路上的顶点编号

(2) 动态规划函数：

$$\mathbf{cost}[i] = \min_{i<j\le n}\{c_{ij} + \mathbf{cost}[j]\} \qquad (1)$$

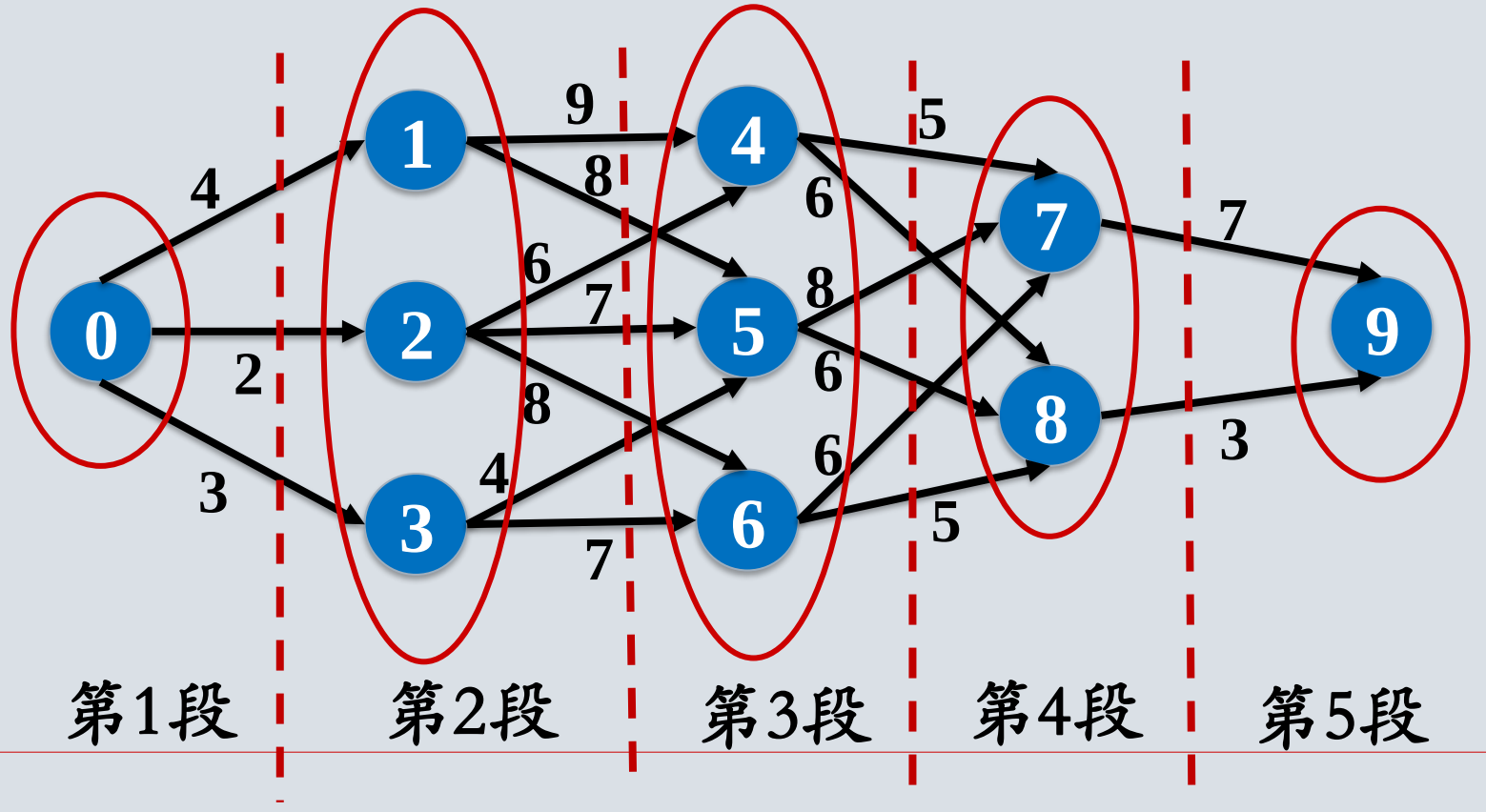
$$path[i] = c_{ij} + cost[j] \text{ 最小的 } j,\ i < j \le n \qquad (2)$$

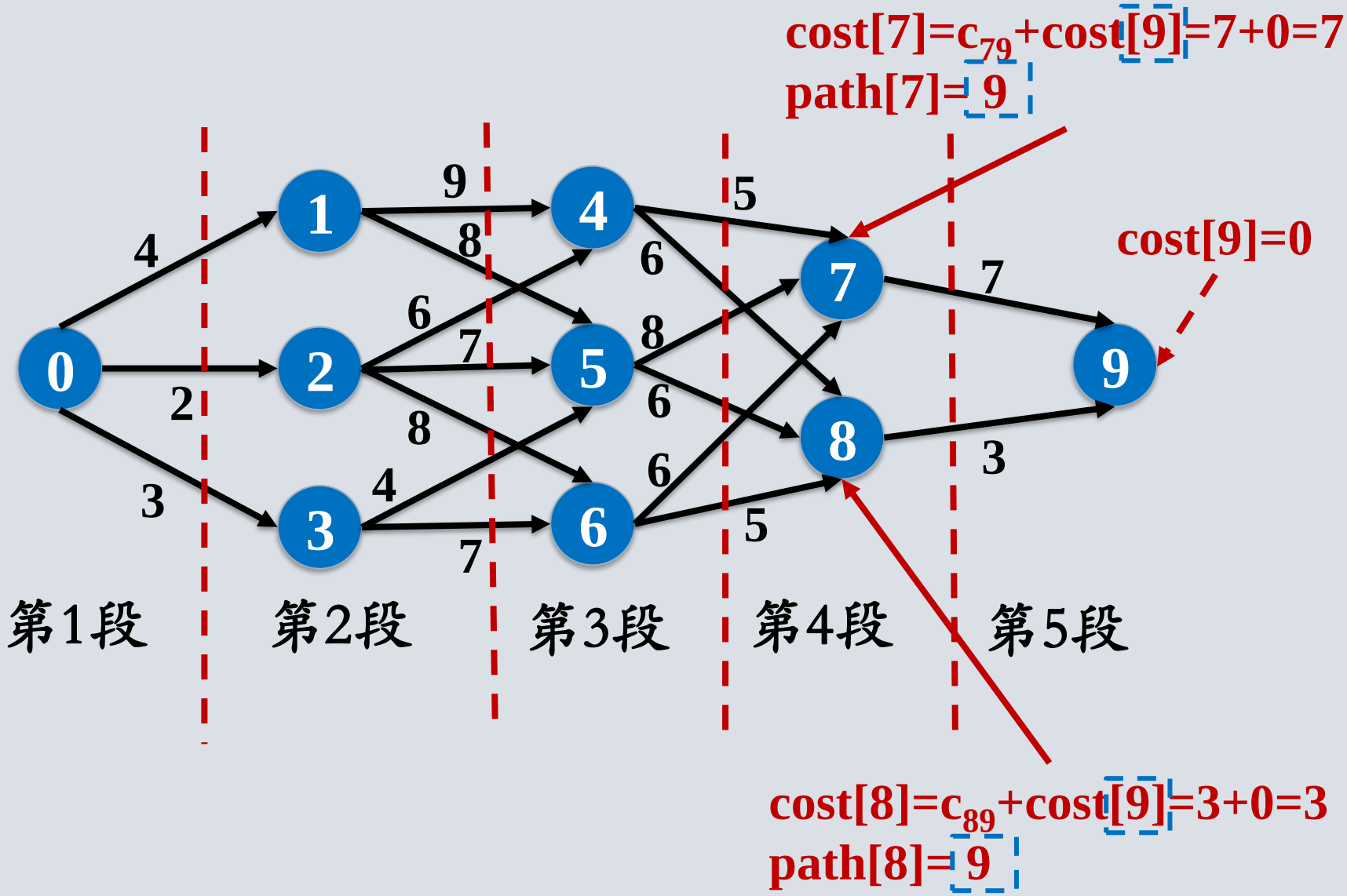
3. 例子：求解图所示的最短路径问题

- 逆向计算cost和path
- 正向确定最短路径

$$path[i] = arg \min_{i<j\le n}\{c_{ij} + \text{cost}[j]\}$$

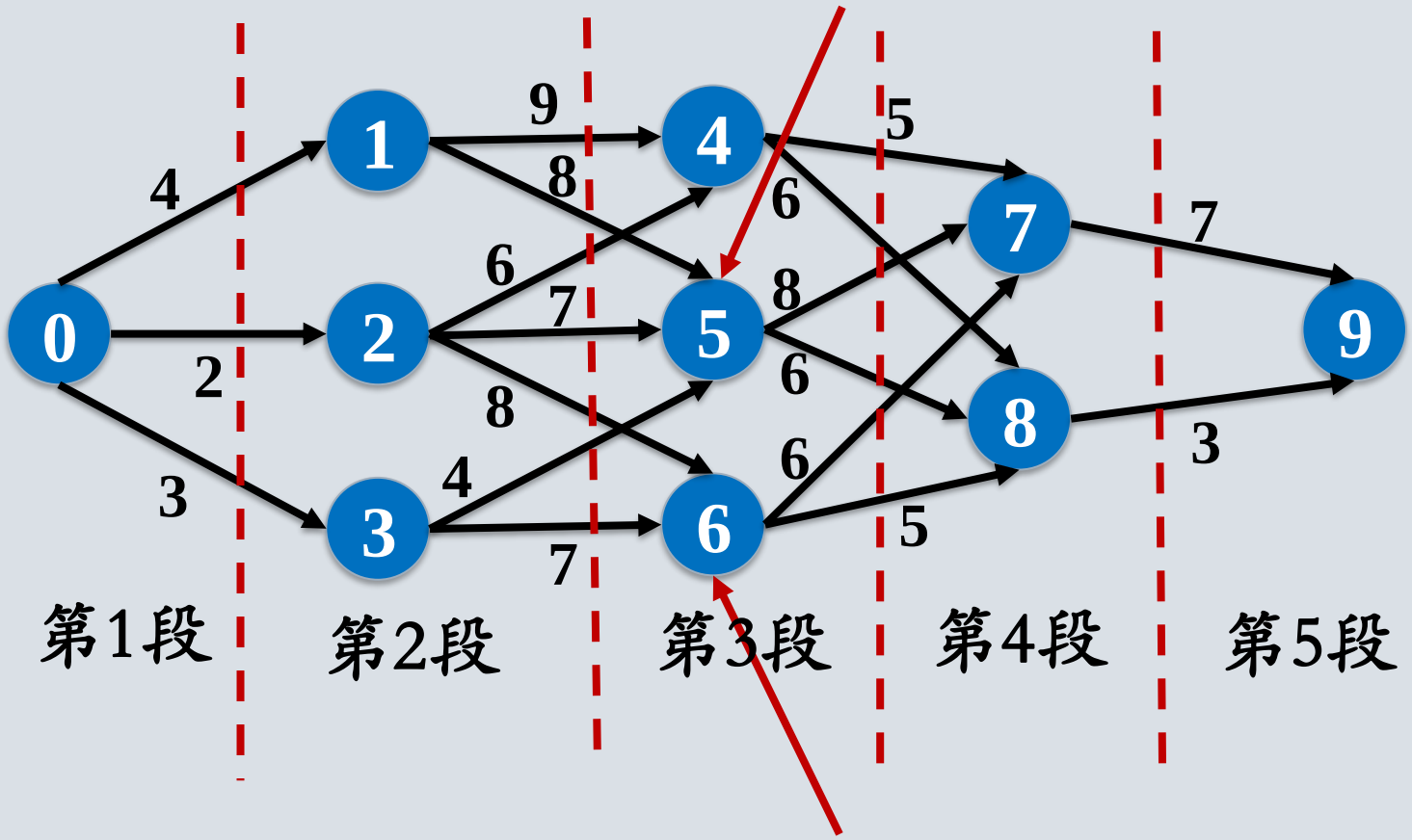
$$\text{cost}[i] = \min_{i<j\le n}\{c_{ij} + \text{cost}[j]\}$$




cost[7]=c79+cost[9]=7+0=7
path[7]=9
cost[9]=0
cost[8]=c89+cost[9]=3+0=3
path[8]=9
0
1
2
3
4
5
6
7
8
9
4
2
3
9
8
6
7
8
6
4
7
5
6
8
6
6
5
7
7
3
第1段
第2段
第3段
第4段
第5段

$cost[5]=\min\{c_{57}+cost[7], c_{58}+cost[8]\}=\{8+7, 6+3\}=9$

$path[5]=8$

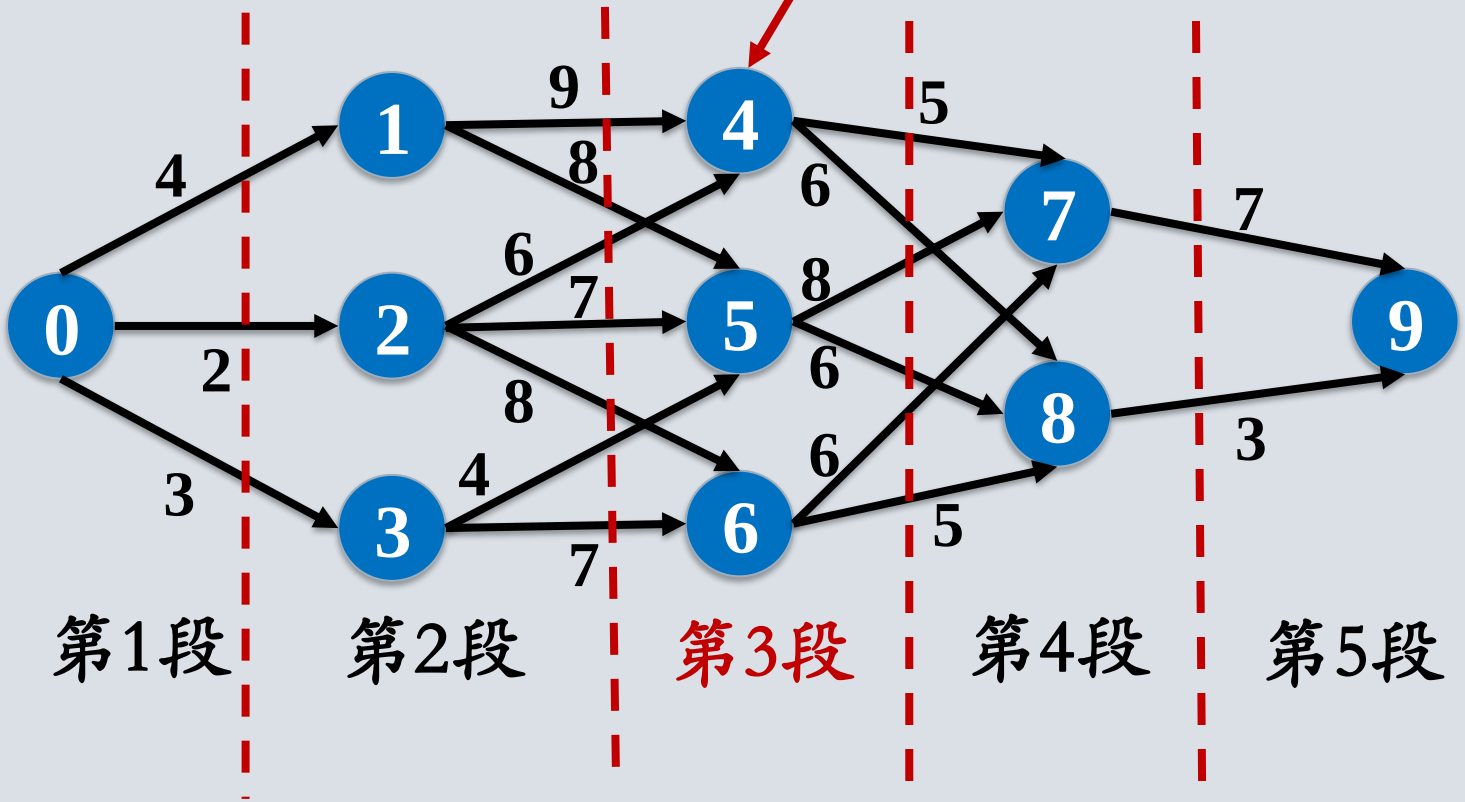


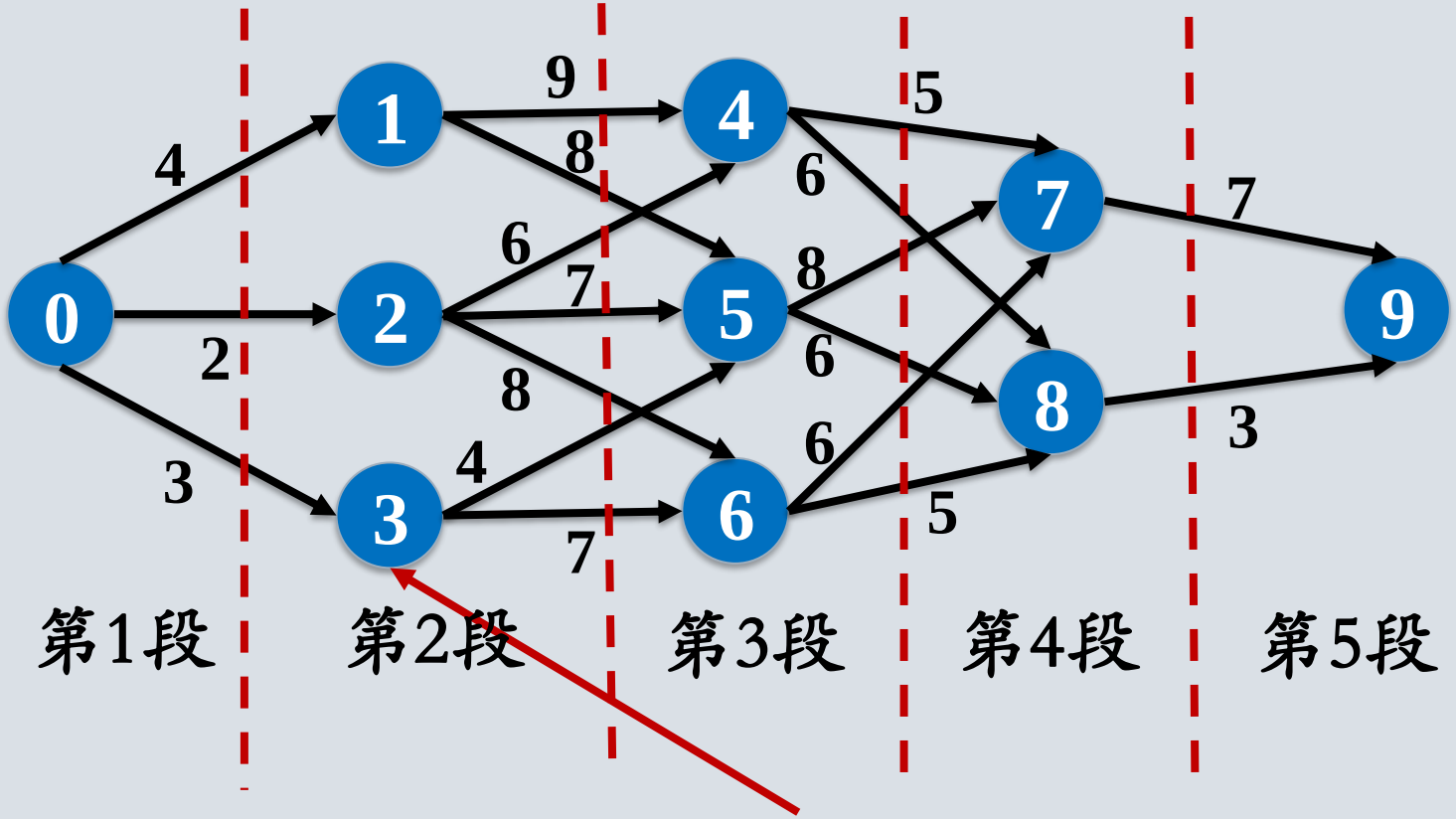
$cost[6]=\min\{c_{67}+cost[7], c_{68}+cost[8]\}=\{6+7, 5+3\}=8$

$path[6]=8$

cost[4]=min{c_{47}+cost[7], c_{48}+cost[8]}={ 5+7, 6+3 }=9

path[4]= 8



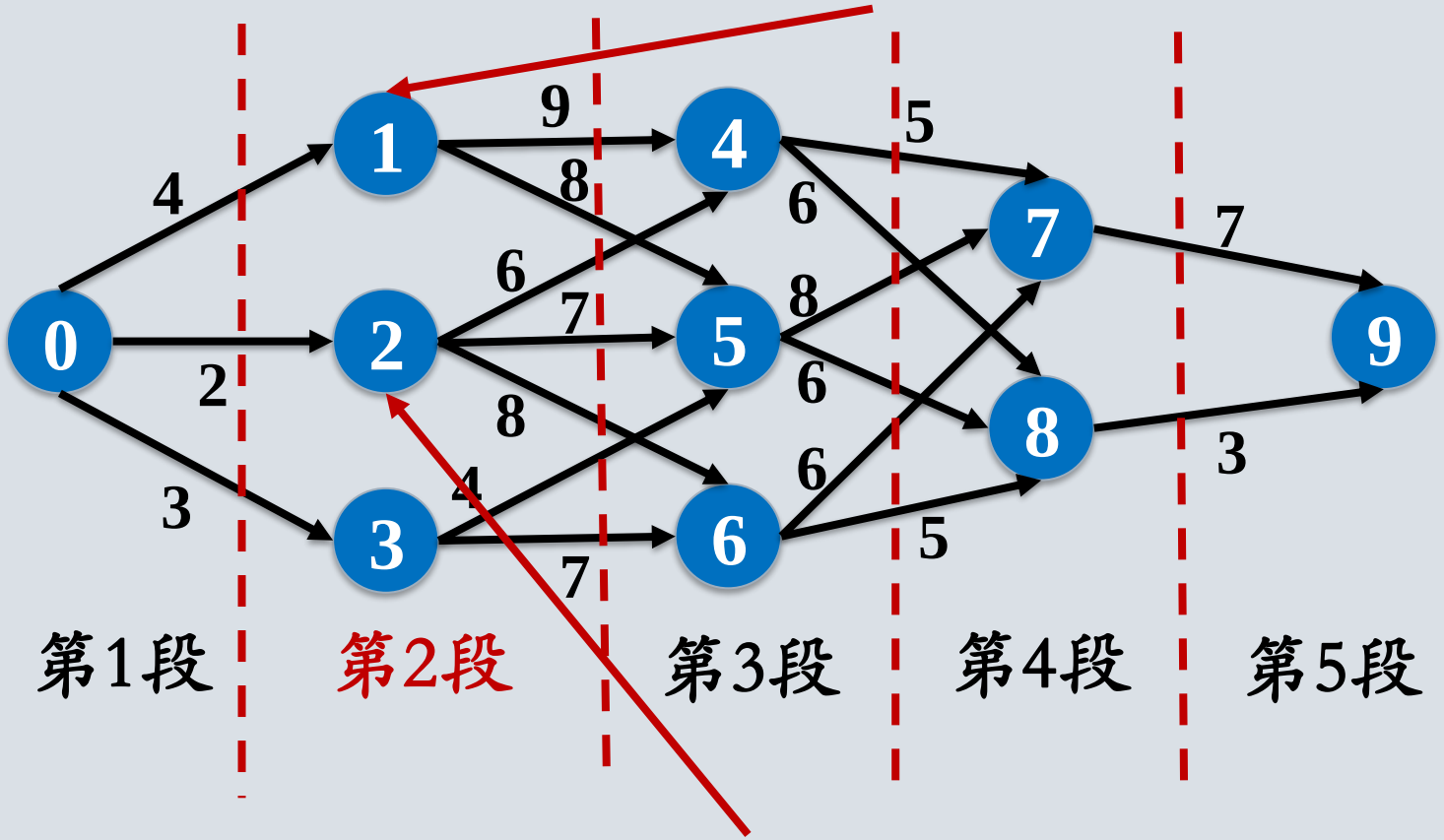


$cost[3]=\min\{c_{35}+cost[5], c_{36}+cost[6]\}=\{4+9, 7+8\}=13$

$path[3]=5$

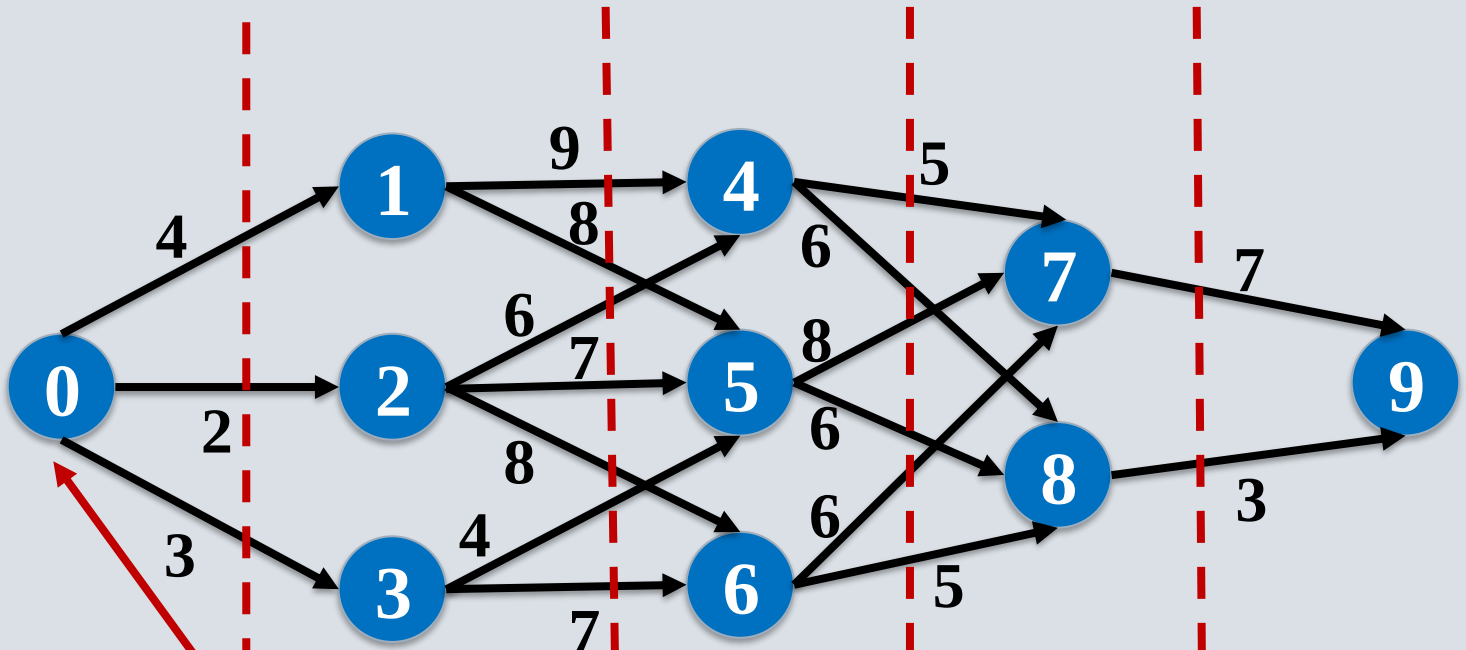
cost[1]=min{c_{14}+cost[4], c_{15}+cost[5]}={9+9, 8+9}=17

path[1]=5



cost[2]=min{c_{24}+cost[4], c_{25}+cost[5], c_{26}+cost[6]}={6+9, 7+9, 8+8}=15

path[2]=4

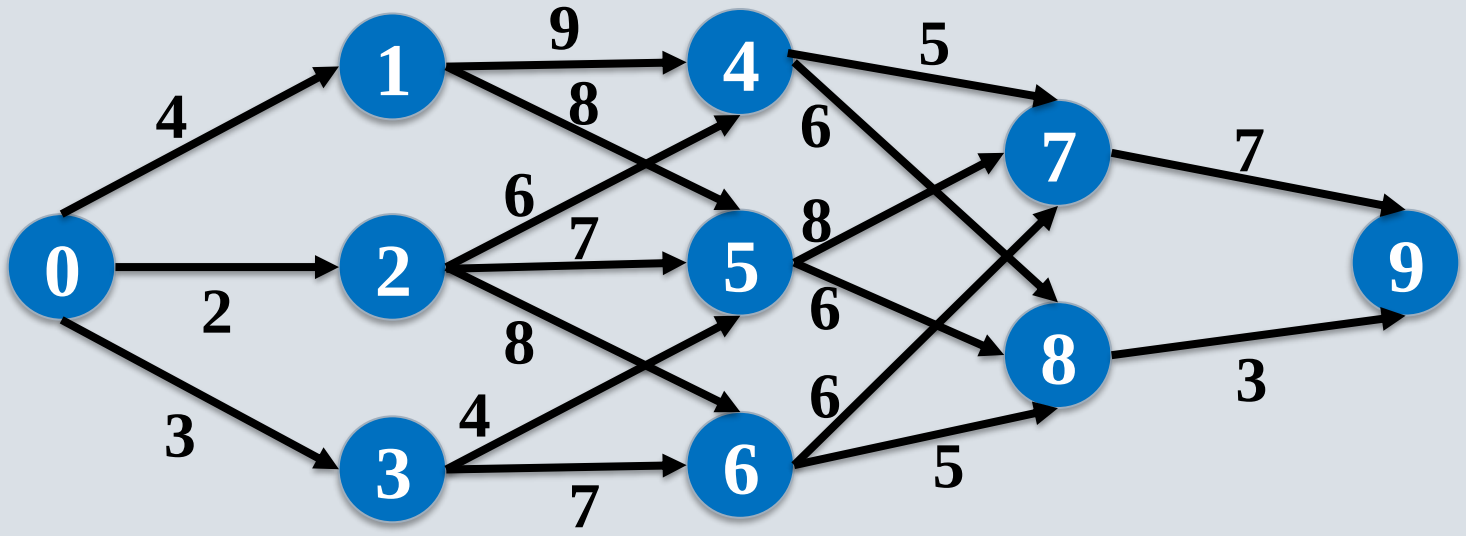


最短路径长度

$cost[0]=min\{c_{01}+cost[1],c_{02}+cost[2],c_{03}+cost[3]\}=\{4+17,2+15,3+13\}=16$

$path[0]=3$

➢ 从path[0]开始往后搜寻，可得具体路径



route[0] = 0

route[1] = path[route[0]] = path[0] = 3

route[2] = path[route[1]] = path[3] = 5

route[3] = path[route[2]] = path[5] = 8

route[4] = path[route[3]] = path[8] = 9

path[0]= 3

path[1]= 5

path[2]= 4

path[3]= 5

path[4]= 8

path[5]= 8

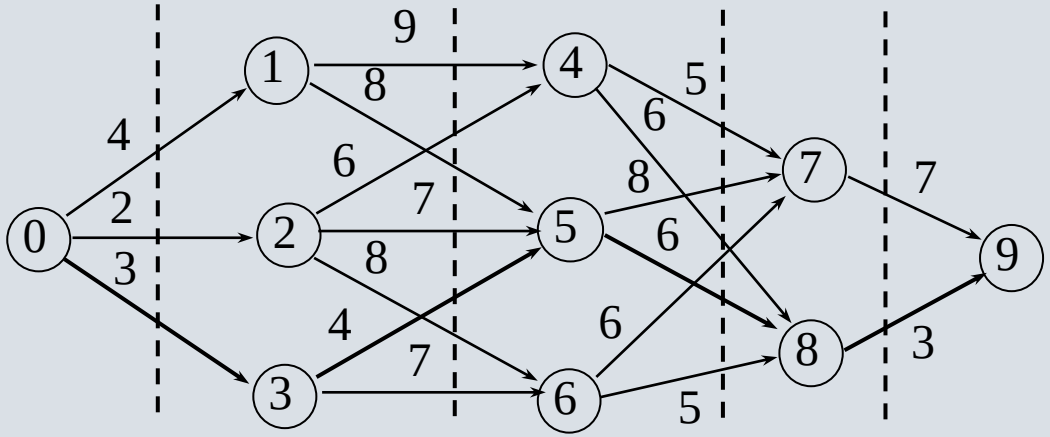
path[6]= 8

path[7]= 9

path[8]= 9

多段图最短路径：方法拓展

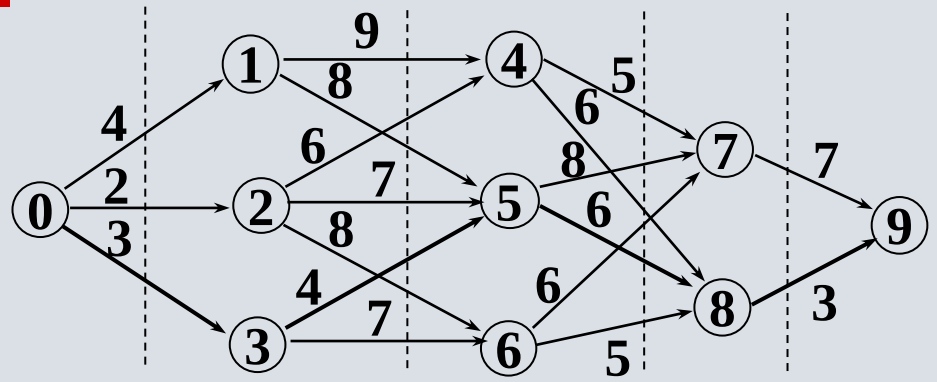
- 在前面的求解中，从收点出发，将求解过程分为了k-1个阶段，逆向推导，确定各个顶点到收点的最短路径;
- 也可以换一个思路，从源点出发，同样将求解过程分成k-1个阶段，正向推导，也可以确定各个顶点到收点的最短路径。



设c_{uv}表示多段图的有向边$<u, v>$上的权值，将从源点s到收点t的最短路径长度记为$d(s, t)$，考虑原问题的部分解$d(s, v)$，显然有下式成立：

$d(s, v) = c_{sv}$ （$<s, v> \in E$）

$d(s, v) = \min\{d(s, u) + c_{uv}\}$ （$<u, v> \in E$）



✓ 首先求解初始子问题，可直接获得：

$d(0,1)=c_{01}=4(0\to1)$

$d(0,2)=c_{02}=2(0\to2)$

$d(0,3)=c_{03}=3(0\to3)$ (0→3)

✓ 再求解下一个阶段的子问题，有：

$d(0,4)=\min\{d(0,1)+c_{14}, d(0,2)+c_{24}\}=\min\{4+9, 2+6\}=8(2\to4)$

$d(0,5)=\min\{d(0,1)+c_{15}, d(0,2)+c_{25}, d(0,3)+c_{35}\}=\min\{4+8, 2+7, 3+4\}$

$=7(3\to5)$ (3→5)

$d(0,6)=\min\{d(0,2)+c_{26}, d(0,3)+c_{36}\}=\min\{2+8, 3+7\}=10(2\to6)$

✓ 再求解下一个阶段的子问题，有：

$d(0,7)=\min\{d(0,4)+c_{47}, d(0,5)+c_{57}, d(0,6)+c_{67}\}=\min\{8+5, 7+8, 10+6\}$

$=13(4\to7)$

$d(0,8)=\min\{d(0,4)+c_{48}, d(0,5)+c_{58}, d(0,6)+c_{68}\}=\min\{8+6, 7+6, 10+5\}$

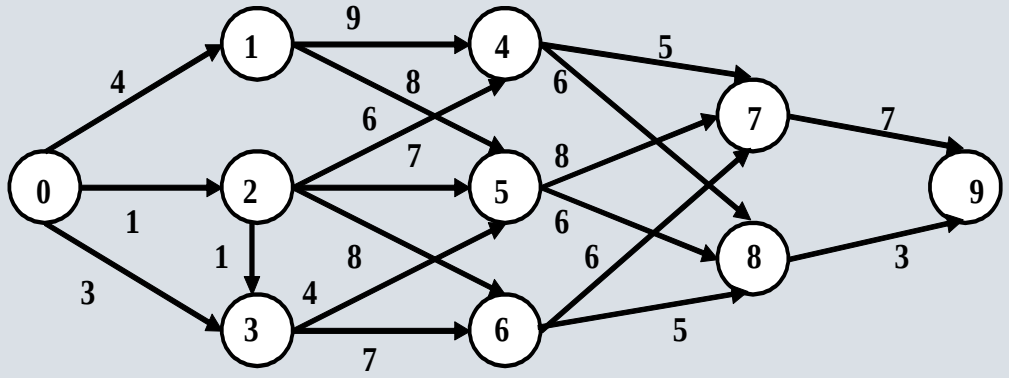
$=13(5\to8)$ (5→8)

✓ 直到最后一个阶段，有：

$d(0,9)=\min\{d(0,7)+c_{79}, d(0,8)+c_{89}\}=\min\{13+7, 13+3\}=16\ (8\to9)$ (8→9)

再将状态进行回溯，得到最短路径0→3→5→8→9，最短路径长度16。

例6.2：求解图所示的最短路径问题



$$cost[i]=\min_{i<j<n}\{c_{ij}+cost[j]\}$$

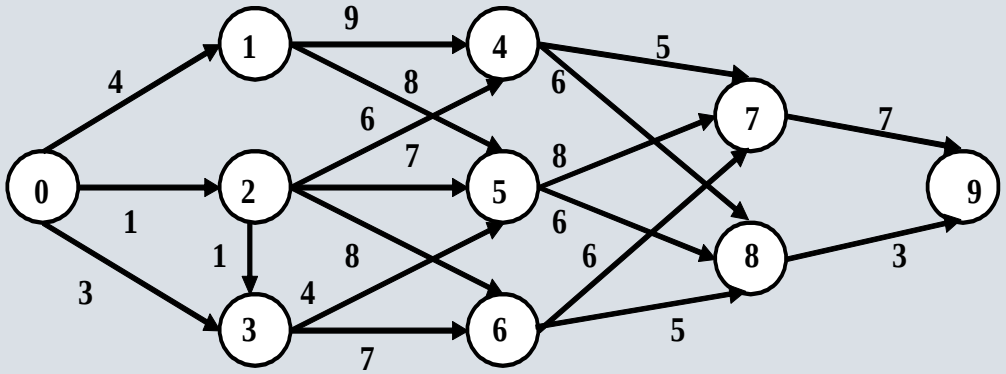
$path[i]=$ 使 $c_{ij}+cost[j]$ 最小的 j　　$i<j<n$

$i=8$：　$cost[8]=c_{89}+cost[9]=3+0=3$

$path[8]=9$

$i=7$：　$cost[7]=c_{79}+cost[9]=7+0=7$

$path[7]=9$



$$cost[i]=\min_{i<j<n}\{c_{ij}+cost[j]\}$$

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

$i=6$: $cost[6]=\min\{c_{67}+cost[7],c_{68}+cost[8]\}$

$=\min\{6+7,5+3\}=8$

$path[6]=8$

$i=5$: $cost[5]=\min\{c_{57}+cost[7],c_{58}+cost[8]\}$

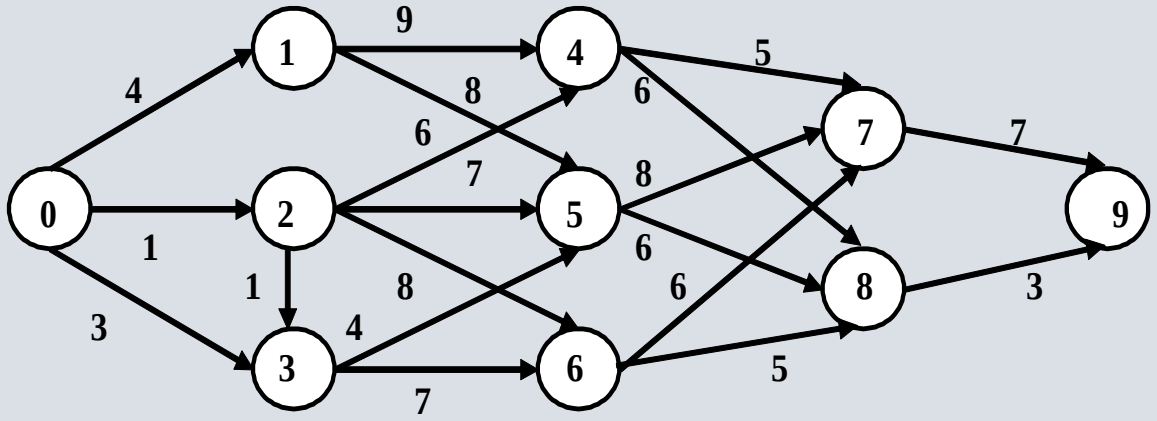
$=\min\{8+7,6+3\}=9$

$path[5]=8$

$i=4$: $cost[4]=\min\{c_{47}+cost[7],c_{48}+cost[8]\}$

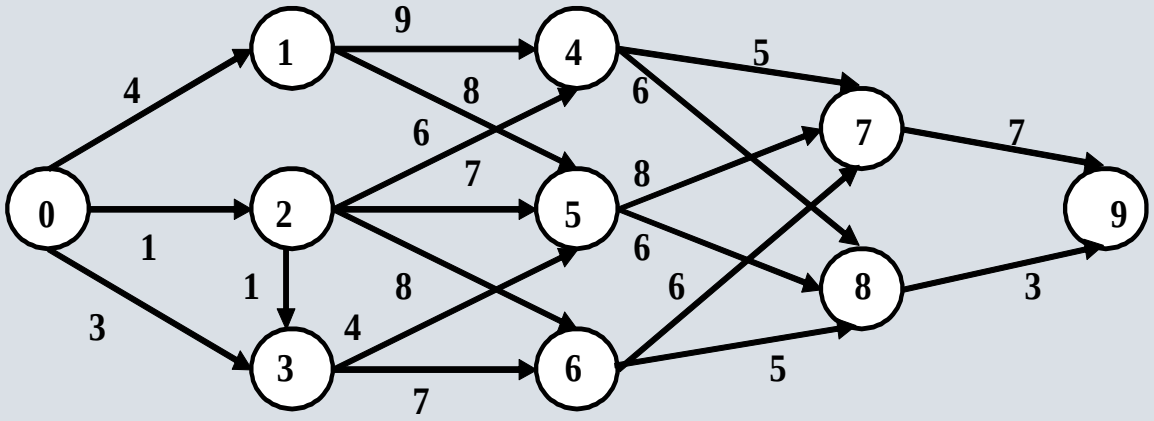
$=\min\{5+7,6+3\}=9$

$path[4]=8$



$$i = 3: \quad \cos t[3] = \min\{c_{35} + \cos t[5], c_{36} + \cos t[6]\}$$
$$= \min\{4 + 9, 7 + 8\} = 13$$

$$path[\,3\,] = 5$$

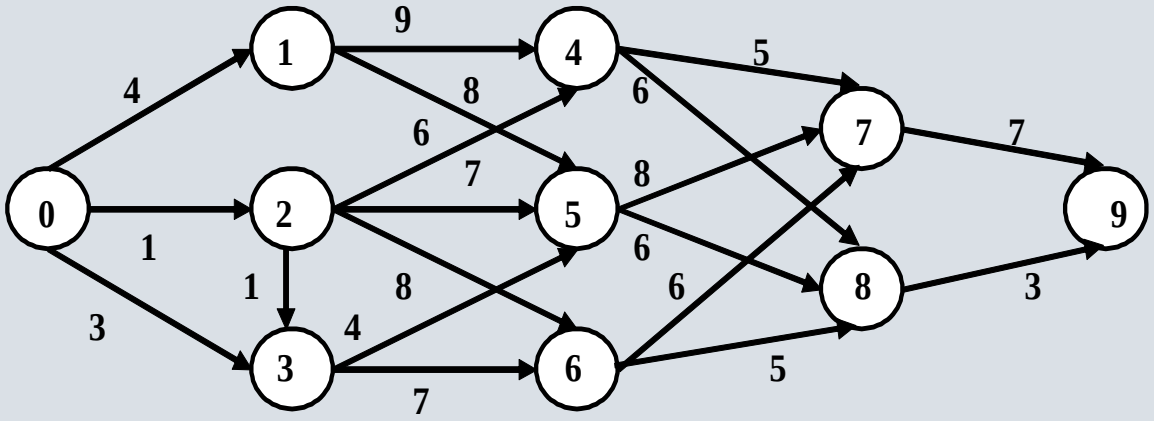


$$i = 2: \quad \cos t[2] = \min\{c_{23} + \cos t[3], c_{24} + \cos t[4], c_{25} + \cos t[5], c_{26} + \cos t[6]\} = 14$$

$$path[\,2\,] = 3$$

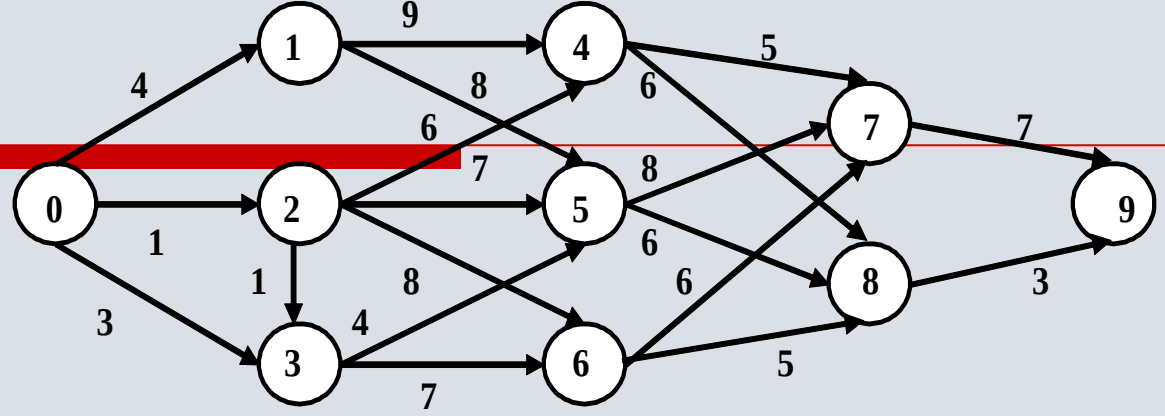
$$i = 1: \quad \cos t[1] = \min\{c_{14} + \cos t[4], c_{15} + \cos t[5]\} = \min\{9 + 9, 8 + 9\} = 17$$

$$path[\,1\,] = 5$$



$$i = 0: \quad \cos t[0] = \min\{c_{01} + \cos t[1], c_{02} + \cos t[2], c_{03} + \cos t[3]\} = 15$$

$$path[\,0\,] = 2$$



path[0] = 2　　path[1] = 5　　path[2] = 3

path[3] = 5　　path[4] = 8　　path[5] = 8

path[6] = 8　　path[7] = 9　　path[8] = 9

route[0] = 0

route[1] = path[route[0]] = path[0] = 2

route[2] = path[route[1]] = path[2] = 3

route[3] = path[route[2]] = path[3] = 5

route[4] = path[route[3]] = path[5] = 8

route[5] = path[route[4]] = path[8] = 9

cost[0] = 15

最短路径:
0,2,3,5,8,9

4. 多段图最短路径问题实现步骤

① 对所有的 i，$0 \le i < n$，置 cost[i] = ∞，path[i] = - 1，cost[$n-1$] = 0

② 令 i = n - 2

③ 根据动态规划函数(1), (2) 式，计算 cost[i], path[i]

④ i = i - 1，若 $i \ge 0$，转 ③；否则，转 ⑤

⑤ 令 i = 0，route[0] = 0

⑥ 如果 route[i] = $n-1$，算法结束；否则，转 ⑦

⑦ i = i + 1，route[i] = path[route[i – 1]]；转 ⑥

6.2.2 多段图动态规划算法的实现

1. 数据结构

```
struct NODE {                    // 邻接表结点的数据结构
    int       v_num;             // 邻接顶点的编号
    Type    len;                 // 邻接顶点与该顶点的费用
struct NODE *next;               // 下一个邻接顶点
};
struct NODE node[n];             // 多段图邻接表头结点
Type   cost[n];
int      route[n];
int      path[n];
```

2. 算法描述（步骤①）　初始化

```
4. Type fgraph(struct NODE node[ ], int route[ ], int n)
5. {   int i;
7.     struct NODE  *pnode;
8.     int  *path = new int[ n ];
9.     Type min_cost, *cost = new  Type[ n ];
10.    for (i=0; i<n; i++) {
11.        cost[ i ] = MAX_VALUE_TYPE;
11.        path[ i ] = -1;   route[ i ] = 0;
12.    }
13.    cost[ n-1 ] = ZERO_VALUE_OF_TYPE;
```

初始化：cost数组、path数组以及route数组初始化：

- cost数组表示当前顶点到收点的最小费用；
- path数组表示当前顶点到收点的最短路径上的前方顶点的编号；
- route数组存储从源点到收点的最短路径上的顶点编号。

编号n-1的顶点为收点

算法描述（步骤②③④） 分段决策

```
14.    for (i=n-2; i>=0; i--) {
15.        pnode = node[ i ]->next;
16.        while (pnode != NULL) {
17.            if (pnode->len + cost[pnode->v_num] < cost[ i ]) {
18.                cost[ i ] = pnode->len + cost[pnode->v_num];
19.                path[ i ] = pnode->v_num;
20.            }
21.            pnode = pnode->next;
22.        }
23.    }
```

从倒数第二个顶点开始，依次计算当前顶点到收点的最短路径，更新当前顶点i到收点的费用cost[i]，以及当前顶点i到收点的最短路径上的前一个顶点编号path[i]。

- pnode->len：当前顶点i到pnode所指向的邻接点的费用；
- cost[pnode->v_num]：从pnode所指向顶点到收点的最小费用。

算法描述（步骤⑤⑥⑦）

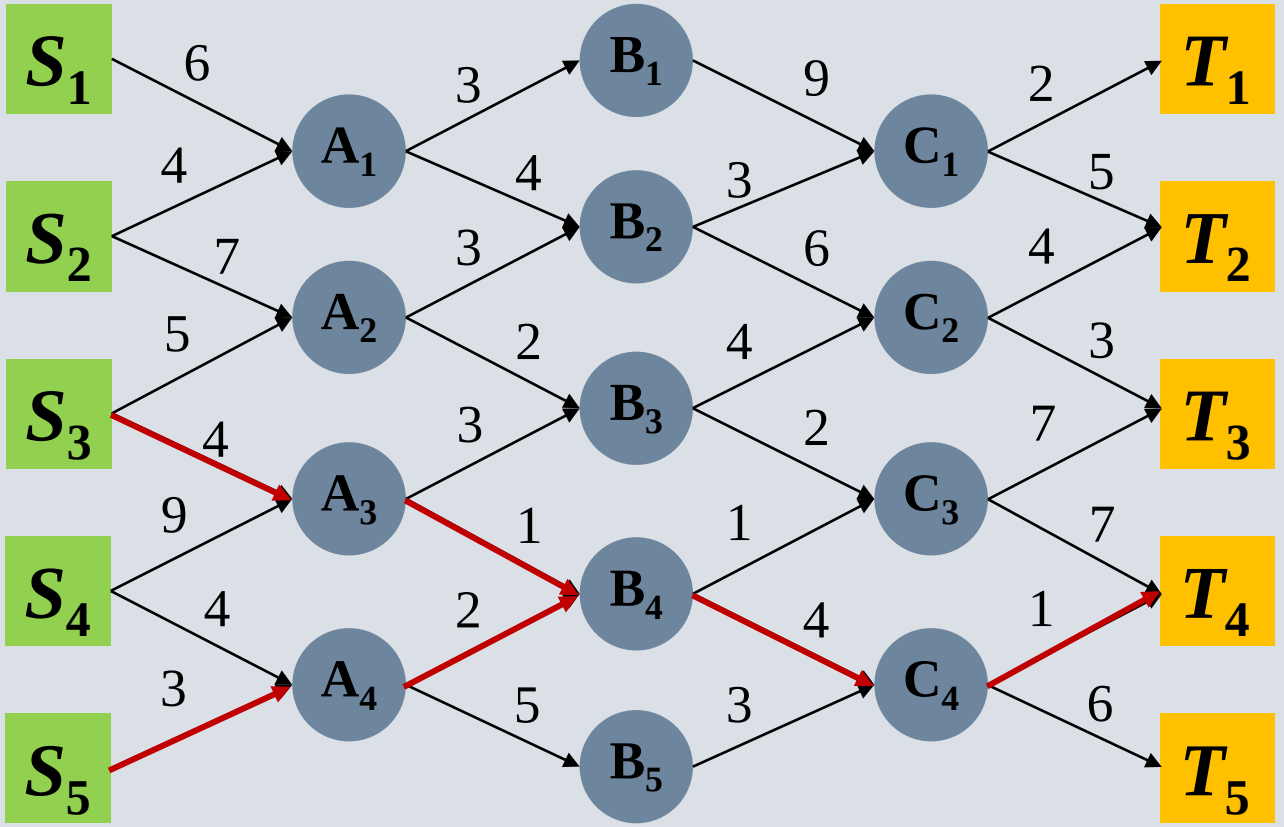
全局最优决策

根据cost数组和path数组，获得从源点到收点的最短路径。

```
24.     i = 0;
25.     while ((route[ i ] != n-1) && (path[ i ] != -1)) {
26.         i++;
27.         route[ i ] = path[ route[ i-1 ] ];
28.     }
29.     min_cost = cost[ 0 ];
30.     delete path;    delete cost;
31.     return min_cost;
32. }
```

从源点到收点的最短路径费用为cost[0]。

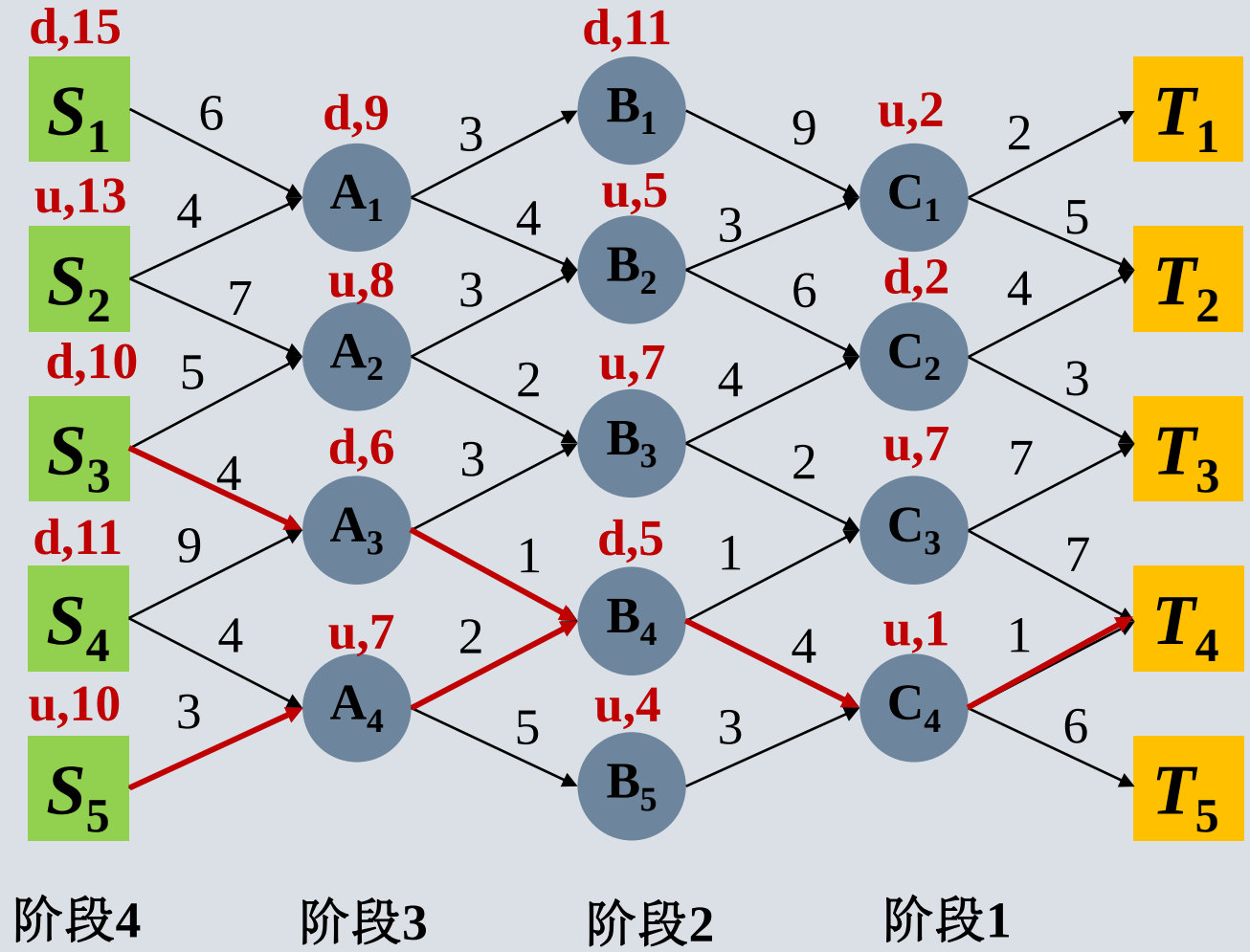
实例


S_1
S_2
S_3
S_4
S_5
A_1
A_2
A_3
A_4
B_1
B_2
B_3
B_4
B_5
C_1
C_2
C_3
C_4
T_1
T_2
T_3
T_4
T_5
6
4
7
5
4
9
4
3
3
4
3
2
3
1
2
5
9
3
4
6
5
4
2
3
4
2
1
4
3
7
7
1
6

算法设计

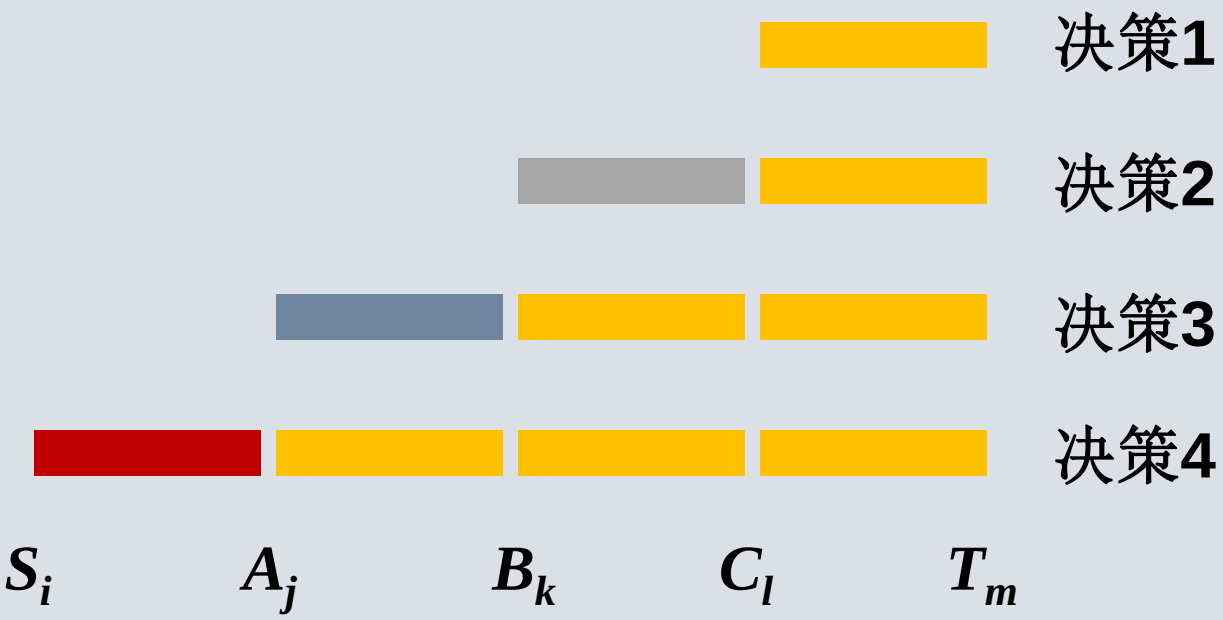
- 蛮力算法：考察每一条从某个起点到某个终点的路径，计算长度，从其中找出最短路径；
  - ✓ 在上述实例中，如果网络的层数为k，那么路径条数将近2^k。
- 动态规划：多阶段决策过程，每步求解的问题是后面阶段求解问题的子问题，每步决策将依赖于以前步骤的决策结果。

动态规划求解


d,15
S1
u,13
S2
d,10
S3
d,11
S4
u,10
S5
d,9
A1
u,8
A2
d,6
A3
u,7
A4
d,11
B1
u,5
B2
u,7
B3
d,5
B4
u,4
B5
u,2
C1
d,2
C2
u,7
C3
u,1
C4
T1
T2
T3
T4
T5
6
4
7
5
4
9
4
3
3
4
3
2
3
1
2
5
9
3
6
4
2
7
1
4
3
2
7
7
1
6
阶段4
阶段3
阶段2
阶段1

子问题的界定

后边界不变，前边界前移



最短路长的依赖关系

$$F(C_l) = \min_m\{C_l T_m\}$$ 决策1

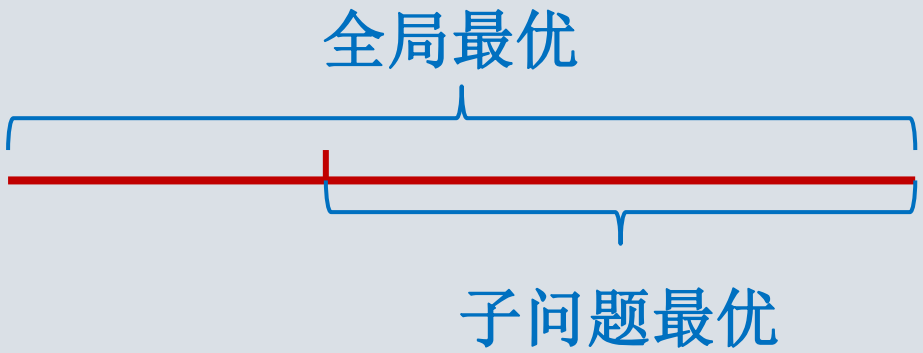
$$F(B_k) = \min_l\{B_k C_l + F(C_l)\}$$ 决策2

$$F(A_j) = \min_k\{A_j B_k + F(B_k)\}$$ 决策3

$$F(S_i) = \min_i\{S_i A_j + F(A_j)\}$$ 决策4

✓ 优化函数值之间存在依赖关系

- 优化原则：最优子结构性质
  - 优化函数的特点：任何最短路的子路径相对于子问题始、终点最短。

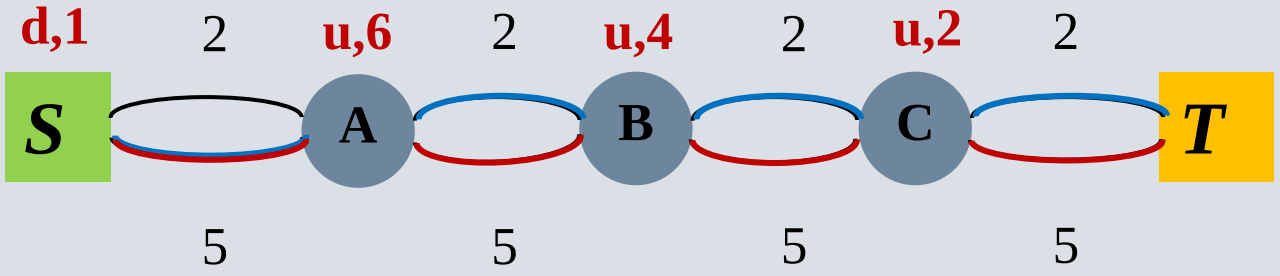


  - 优化原则：一个最优决策序列的任何子序列本身一定是相对于子序列的初始和结束状态的最优决策序列。

反例

求总长模10的最小路径

（总长度除以10，取余数，余数最小就是要求的最短路）



- 动态规划算法的解：下，上，上，上
- 最优解：下，下，下，下
- 不满足优化原则，不能用动态规划

小结

- 动态规划Dynamic Programming
  - 求解过程是多阶段决策过程，每步处理一个子问题，可用于求解组合优化问题；
  - 适用条件：问题要满足优化原则或最优子结构性质，即：一个最优决策序列的任何子序列本身一定是相对子序列的初始和结束状态的最优决策序列。

动态规划设计要素

1. 问题建模，优化的目标函数是什么？约束条件是什么？
2. 如何划分子问题（边界）？
3. 问题的优化函数值与子问题的优化函数值存在着什么依赖关系（递推方程）？
4. 是否满足优化原则？
5. 最小子问题怎样界定？其优化函数值，即初值等于什么？

6.3 资源分配问题

6.3.1 资源分配问题的决策过程

6.3.2 资源分配算法的实现

6.3.1 资源分配问题的决策过程

1. 资源分配问题

- 只有一种资源有待于分配到若干个活动，其目标是如何最有效地在各个活动中分配这种资源；
- 在建立任何效益分配问题的DP(Dynamic Programming)模型时，阶段对应于活动，每个阶段的决策对应于分配到该活动的资源数量；
- 任何状态的当前状态总是等于留待当前阶段和以后阶段的资源数量，即总资源量减去前面各阶段已分配的资源量。

2. 资源分配问题的建模

- 问题：m份资源，n项工程，$G_i(x)$：将x份资源分配给第i个项目的收益，求使得总收益最大的投资方案。
- 建模：
  - 问题的解是向量$<x_1, x_2, \ldots, x_n>$
  - x_i是投给项目i的资源份数，i=1,2,…,n.
  - 目标函数$max\{G_1(x_1) + G_2(x_2) + \cdots + G_n(x_n)\}$，$G(m)$
  - 约束条件$x_1 + x_2 + \cdots + x_n = m, x_i \in N$

- 子问题界定和计算顺序

➢ 子问题界定：由参数k和x界定

- k：考虑对项目1，2，…，k的投资
- x：投资资源数不超过x

➢ 原始输入：$k=n$, $x=m$

子问题计算顺序：$k=1,2,\ldots,n$

对于给定的k，$x=1,2,\ldots,m$

3. 决策过程

1）定义两组变量

$f_i(x)$：把 x 份资源分配给前面 i 个工程所得到的最大利润

$d_i(x)$：使$f_i(x)$最大时，分配给第 i 个工程的资源份额

2）阶段划分

n个工程划分为 n 个阶段:

第一阶段把 x ($x = 1, \ldots, m$) 份资源分配给第1个工程;

……

第 i 阶段把 x ($x = 1, \ldots, m$) 份资源分配给前面 i 个工程 ($i = 2, \ldots, n$)

3）分配份额为 x ($x = 1, \ldots, m$) 时，第 i 阶段的最大利润及第 i 工程的最优分配份额 ($i = 1, \ldots, n$)

第一阶段，把 x 份资源全部分配给第一个工程：

$$f_1(x) = G_1(x) \qquad 0 \le x \le m$$

$$d_1(x) = x$$

第二阶段，把 x 份资源分配给前面两个工程：

$$f_2(x) = \max\{ G_2(z) + f_1(x - z) \} \qquad 0 \le x \le m,\ 0 \le z \le x$$

$d_2(x)$ = 使 $f_2(x)$ 达最大的 z

在第 i 阶段，把 x 份资源分配给前面 i 个工程：

$$f_i(x) = \max\{ G_i(z) + f_{i-1}(x - z) \} \qquad 0 \le x \le m,\ 0 \le z \le x$$

$d_i(x)$ = 使 $f_i(x)$ 达最大的 z

4）分配份额为 m 时，第 i 阶段的最大利润 g_i 及前面 i 个工程的最优分配份额 q_i ($i = 1, \ldots, n$)

$$g_i = \max\{ f_i(1), \cdots, f_i(m)\}$$

$$q_i = \text{使 } f_i(x) \text{达最大的 } x$$

5）全局最大利润 $optg$ 及所分配工程项目的最大数目 k

$$optg = \max\{g_1, \cdots, g_n\}$$

$$k = \text{使 } g_i \text{ 最大的 } i$$

6）全局最大利润下 k 个工程的最优份额 $optx_k$

$$optx_k = \text{与最大的 } g_i \text{ 相对应的 } q_i$$

7）全局最大利润下分配给第 k 个工程的最优份额 $optq_k$

$$optq_k = d_k(optx_k)$$

8）分配给其余*k*-1个工程的剩余的最优份额:

$$optx_{k-1} = optx_k - d_k(optx_k)$$

9）回溯，得到分配给前面各个工程的最优份额的递推关系式

$$i = k, k-1, \ldots, 1$$

$$optq_i = d_i(optx_i)$$

$$optx_{i-1} = optx_i - optq_i$$

4. 实现步骤

1）对各个阶段、各个不同份额的资源计算$f_i(x)$ 及 $d_i(x)$;

2）计算各个阶段的最大利润 g_i 及获得此最大利润的分配份额 q_i;

3）计算全局的最大利润 $optg$，总的最优分配份额 $optx$及最大的工程项目数目 k;

4）递推计算各个工程的最优分配份额。

5. 例子

8 个份额的资源，分配给 3 个工程，其利润函数如下：

x	0	1	2	3	4	5	6	7	8
G_1	0	4	26	40	45	50	51	52	53
G_2	0	5	15	40	60	70	73	74	75
G_3	0	5	15	40	80	90	95	98	100

◆ 第1步：求各个阶段不同分配份额时的最大利润，及各个工程在此利润下的分配份额。

✓ 第一阶段：把8份资源分配给第1个工程

x	0	1	2	3	4	5	6	7	8
f_1	0	4	26	40	45	50	51	52	53
d_1	0	1	2	3	4	5	6	7	8

x	0	1	2	3	4	5	6	7	8
G_1	0	4	26	40	45	50	51	52	53
G_2	0	5	15	40	60	70	73	74	75
G_3	0	5	15	40	80	90	95	98	100

✓ 第二阶段：把8份资源分配给前面两个工程

										$f_2(x)$	$d_2(x)$
x=1		0, 1	1, 0								
	f_2	4	5							5	1
x=2		0, 2	1, 1	2, 0							
	f_2	26	9	15						26	0
x=3		0, 3	1, 2	2, 1	3, 0						
	f_2	40	31	19	40					40	0
x=4		0, 4	1, 3	2, 2	3, 1	4, 0					
	f_2	45	45	41	44	60				60	4

第二阶段（续）

x=5		0,5	1,4	2,3	3,2	4,1	5,0				$f_2(x)$	$d_2(x)$
	f_2	50	50	55	66	64	70				70	5
x=6		0,6	1,5	2,4	3,3	4,2	5,1	6,0				
	f_2	51	55	60	80	86	74	73			86	4
x=7		0,7	1,6	2,5	3,4	4,3	5,2	6,1	7,0			
	f_2	52	56	65	85	100	96	77	74		100	4
x=8		0,8	1,7	2,6	3,5	4,4	5,3	6,2	7,1	8,0		
	f_2	53	57	66	90	105	110	99	78	75	110	5

同样，计算出$f_3(x)$及$d_3(x)$

x	0	1	2	3	4	5	6	7	8	$g_i(x)$
f_1	0	4	26	40	45	50	51	52	53	53
d_1	0	1	2	3	4	5	6	7	8	
f_2	0	5	26	40	60	70	86	100	110	110
d_2	0	1	0	0	4	5	4	4	5	
f_3	0	5	26	40	80	90	106	120	140	140
d_3	0	1	0	0	4	5	4	4	4	

◆第2步：求各个阶段的最大利润，及在此利润下的分配份额

$$g_1 = 53 \qquad g_2 = 110 \qquad g_3 = 140$$
$$q_1 = 8 \qquad q_2 = 8 \qquad q_3 = 8$$

◆第3步：计算全局的最大利润 $optg$，总的最优分配份额 $optx_k$及最大的工程项目数目 k：

$$optg = \max(53, 110, 140) = 140,\quad k = 3,\quad optx_3 = 8$$

◆第4步：计算各个工程的最优分配份额

$$optq_3 = d_3(8) = 4 \qquad optx_2 = optx_3 - optq_3 = 8 - 4 = 4$$

$$optq_2 = d_2(4) = 4 \qquad optx_1 = optx_2 - optq_3 = 4 - 4 = 0$$

$$optq_1 = d_1(0) = 0$$

x	0	1	2	3	4	5	6	7	8	$g_i(x)$
f_1	0	4	26	40	45	50	51	52	53	53
d_1	0	1	2	3	4	5	6	7	8	
f_2	0	5	26	40	60	70	86	100	110	110
d_2	0	1	0	0	4	5	4	4	5	
f_3	0	5	26	40	80	90	106	120	140	140
d_3	0	1	0	0	4	5	4	4	4	

$optq_3 = d_3(8) = 4$ $optx_2 = optx_3 - optq_3 = 8 - 4 = 4$

$optq_2 = d_2(4) = 4$ $optx_1 = optx_2 - optq_3 = 4 - 4 = 0$

$optq_1 = d_1(0) = 0$

- 最终的决策结果：分配给第2、3个项目各4个份额，可得到最大利润140

6.3.2 资源分配算法的实现

1. 数据结构

```
int     m;          // 可分配的资源份额
int     n;          // 工程项目个数
Type    G[n][m+1];  // 工程的利润表
Type    optg;       // 最优的总利润
int     optq[n];    // 工程最优分配份额
Type    f[n][m+1];  // 各阶段不同资源份额的最大利润
int     d[n][m+1];  // f[i][x] 最大时,第i项工程的分配份额
Type    g[n];       // 各阶段的最大利润
int     q[n];       // 第 i 阶段第 i 工程最优分配份额
int     optx;       // 最优分配时的资源最优分配份额
int     k;          // 最优分配时的工程项目数
```

2. 算法描述（步骤①）　　数据初始化

```
2. Type alloc_res(int n, int m, Type G[ ][ ], int optq[ ])
3. {
4.     int  optx, k, i, z, x;
5.     int  *q = new int[ n ];    // 分配工作单元
6.     int (*d)[m+1] = new int[ n ][ m+1 ];
7.     Type (*f)[m+1] = new Type[ n ][ m+1 ];
8.     Type *g = new Type[ n ];
9.     for (x=0; x<=m; x++) {// 第一个工程的份额利润表
10.        f[ 0 ][ x ] = G[ 0 ][ x ];   d[ 0 ][ x ] = x;
11.    }
```

算法描述（步骤①）

设置前i个工程的资源份额利润表

```
12.   for (i=1; i<n; i++) {
13.       f[ i ][ 0 ] = G[ i ][ 0 ] + f[ i-1 ][ 0 ];     d[ i ][ 0 ] = 0;
15.       for (x=1; x <= m; x++) {
16.           f[ i ][ x ] = f[ i ][ 0 ];   d[ i ][ x ] = 0;
17.           for (z=0; z <= x; z++) {
18.               if (f[ i ][ x ]<G[ i ][ z ]+f[ i-1 ][ x-z ]) {
19.                   f[ i ][ x ] = G[ i ][ z ] + f[ i-1][ x-z ];
20.                   d[ i ][ x ] = z;
21.               }
22.           }
23.       }
24.   }
```

前i个工程的利润初值

第i个工程的资源份额

前i个工程在不同资源分配x下的最大利润

第i个工程的资源分配份额

算法描述（步骤②③）

第i阶段的最大利润；全局利润最大

```
25.   for (i=0; i<n; i++) {
26.       g[ i ] = f[ i ][ 0 ];    q[ i ] = 0;
27.       for (x=1; x<= m; x++)
28.           if (g[ i ] < f[ i ][ x ])
29.           {    g[ i ] = f[ i ][ x ];   q[ i ] = x;    }
30.   }
33.   optg = g[ 0 ];   optx = q[ 0 ];   k = 0;
34.   for (i=1; i < n; i++) {
35.       if (optg < g[ i ])
36.       {    optg = g[ i ];   optx = q[ i ];   k = i;    }
37.   }
```

第i阶段的最大利润g_i

全局最大利润opt$g$

算法描述（步骤④）

```
39.    if (k < n-1) {      // 最大编号之后的工程不分配份额
40.        for (i=k+1; i<n; i++)
41.            optq[ i ] = 0;
42.    for (i=k; i >= 0; i--) {   // 最大编号之前的工程分配份额
43.        optq[ i ] = d[ i ][ optx ];
44.        optx = optx – optq[ i ];
45.    }
46.    delete q;   delete d;
47.    delete f;   delete g;
48.    return optg;
49. }
```

时间复杂性分析(方法1)

递推方程来分析：

$$f_i(x) = \max\{ G_i(z) + f_{i-1}(x - z) \}$$
$$f_1(x) = G_1(x)$$

z有$m+1$种可能的取值，计算项 $f_i(x)$ $(2 \le i \le n, 1 \le x \le m)$，需要：

$x+1$次加法

x次比较

对备忘录中所有的项求和：

加法次数：$\sum_{i=2}^{n}\sum_{x=1}^{m}(x+1)=\frac{1}{2}(n-1)m(m+3)$

比较次数：$\sum_{i=2}^{n}\sum_{x=1}^{m}x=\frac{1}{2}(n-1)m(m+1)$

时间复杂度：$\Theta(nm^2)$

空间复杂度：$\Theta(nm)$

时间复杂性分析（方法2）

（1）各阶段在各种不同份额下得最优利润：

- 计算第一个阶段的份额利润表，花费时间为$\Theta(m)$
- 计算以后各阶段的份额利润表，花费时间为：

$$\frac{1}{2}(n-1)m(m+1)=\Theta(nm^2)$$

（2）计算各阶段的最大利润，及该阶段的最优分配份额：$\Theta(nm)$

（3）计算全局的最大利润：$\Theta(nm)$

（4）计算各个工程的最优分配份额：$\Theta(n)$

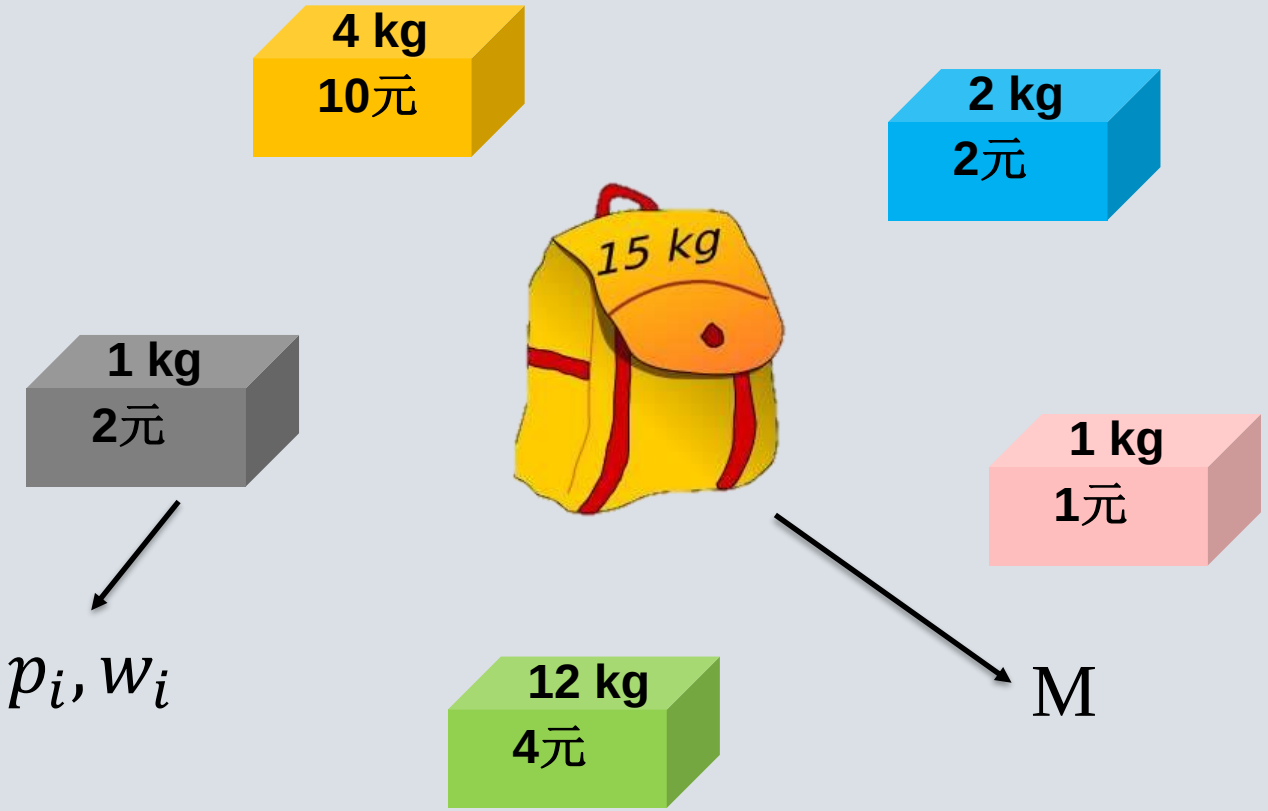
时间复杂度：$\Theta(nm^2)$

6.6 0/1 背包问题

6.6.1 0/1背包问题的求解过程

6.6.2 0/1背包问题的实现

问题引入



- 背包载重量为M
- n个价值为p_i, 重量为w_i的物品

6.6.1 0/1背包问题的求解过程

1. 背包问题的建模

- 0/1背包问题：物体或者被放入，或不被放入背包。
- 假设：x_i表示物体i被装入背包的情况，x_i =0,1

  - 约束方程：$\sum_{i=1}^{n} w_i x_i \leq M$
  - 目标函数：$optp = max \sum_{i=1}^{n} p_i x_i$

✓ 寻找一个满足上述约束方程、并使目标函数达到最大的解向量 $X = (x_1, x_2, \ldots, x_n)$。

子问题的界定和计算顺序

子问题界定：由参数k和y界定

- k：考虑对物品1,2,…,k的选择
- y：背包总重量不超过y

原始输入：$k=n, y=M$

子问题的计算顺序：

$k=1,2,\ldots,n$

对于给定的k，$y=1,2,\ldots,M$

2. 决策过程

1）定义变量

$optp_i(j)$ ：在前 i 个物体中，能够装入载重量为j的背包中物体的最大价值，j=1,2,…,m

2）阶段划分

n个物体划分为 n 个阶段:

第一阶段：装入一个物体，在载重量为 x (x = 1, …, m) 时的最大价值;

第 二 阶段：装入前两个物体时，在不同载重量的背包下得到的最大价值 (i = 2, …, n)

以此类推，直到第n个阶段。

3）求解过程：

- 确定各个阶段在各种不同背包载荷情况下的最大价值；

  令背包的载重量范围为0~m

  动态规划函数：

$$optp_i(0) = optp_0(j) = 0$$

$$optp_i(j) = \begin{cases} optp_{i-1}(j) & j < w_i \\ max\{optp_{i-1}(j), optp_{i-1}(j - w_i) + p_i\} & j \geq w_i \end{cases}$$

$optp_n(m)$: 在载重量为m的背包下，装入n个物体时能够取得的最大价值。

- 确定装入背包的具体物体：　从$optp_n(m)$的值向前倒推
  - $optp_n(m) > optp_{n-1}(m)$：表明第$n$个物体被装入背包，则前$n$-1个物体被装在载重量为$m - w_n$的背包中；
  - $optp_n(m) \leq optp_{n-1}(m)$：第$n$个物体未被装入背包，则前$n$-1个物体，被装入在载重量为$m$的背包中。
  - 以此类推，直到确定第一个物体是否被装入背包为止。
    - 递推关系：
      - ✓ $optp_i(j) \leq optp_{i-1}(j)$：则$x_i = 0$
      - ✓ $optp_i(j) > optp_{i-1}(j)$：则 $x_i = 1, j = j - w_i$

3. 例子

5个物体重量分别为2,2,6,5,4，价值分别为6,3,5,4,6，背包的载重量为10，求装入背包的物体及其总价值。

- ✓ 用一个$(n+1)(m+1)$的表，来存放把前面i个物体装入载重量为j的背包时所能取得的最大价值。

w	p	n	0	1	2	3	4	5	6	7	8	9	10
		0	0	0	0	0	0	0	0	0	0	0	0
2	6	1	0	0	6	6	6	6	6	6	6	6	6
2	3	2	0	0	6	6	9	9	9	9	9	9	9
6	5	3	0	0	6	6	9	9	9	9	11	11	14
5	4	4	0	0	6	6	9	9	9	10	11	13	14
4	6	5	0	0	6	6	9	9	12	12	15	15	15

- ✓ 装入背包的物体的最大价值为15，装入背包的物体为$x=\{1,1,0,0,1\}$

4. 实现步骤

1）初始化，对满足$0 \le i \le n, 0 \le j \le m$的$i$和$j$，令$optp_i(0) = 0, optp_0(j) = 0$

2）令i=1

3）对满足$1 \le j \le m$的j，计算$optp_i(j)$

4）$i = i + 1$, 若i>n ，转步骤5）；否则转步骤3）；

5）令$i = n, j = m$

6）按式求向量的第i个分量x_i

7）i=i-1，若i>0，转步骤6）；否则算法结束。

6.6.2 0/1背包问题的实现

1. 数据结构和变量：

```
int       w[n];          /* n个物体的重量 */
Type      p[n];          /* n个物体的价值 */
int       m;             /* 背包的载重量 */
BOOL  x[n];              /* 装入背包的物体,元素为TRUE */
                         /* 时,对应物体被装入 */
Type      v;             /* 装入背包中物体的最大价值 */
Type      optp[n+1][m+1];        /* i个物体装入载重量为 j的 */
                                 /* 背包中的最大价值 */
```

2. 算法描述：　　初始化表$optp_i(j)$的第0行和第0列

```
1. template <class Type>
2. Type knapsack_dynamic(int w[ ],Type p[ ], int n, int m, BOOL x[ ])
3. {
4.     int i,j,k;
5.     Type v,(*optp)[m+1] = new Type[n+1][m+1];    /* 分配工作单元 */
6.     for (i=0;i<=n;i++)    {                                  /* 初始化第0列 */
7.         optp[i][0] = ZERO_VALUE_OF_TYPE;
8.         x[i] = FALSE;                              /* 解向量初始化为FALSE */
9.     }
10.    for (i=0;i<=m;i++)                              /* 初始化第0行 */
11.        optp[0][i] = ZERO_VALUE_OF_TYPE;
```

计算$optp_i(j)$

```
12.    for (i=1;i<=n;i++) {                          /* 计算optp[i][j] */
13.        for (j=1;j<=m;j++) {
14.            optp[i][j] = optp[i-1][j];
15.            if ((j>=w[i])&&(optp[i-1,j-w[i]]+p[i]>optp[i-1][j])
16.                optp[i][j] = optp[i-1,j-w[i]]+p[i];
17.        }
18.    }
```

分阶段决策

从$optp_n(m)$向前递推，求出装入背包的物体

```
19.    j = m;                          /* 递推装入背包的物体 */
20.    for (i=n;i>0;i--) {
21.        if (optp[i][j]>optp[i-1][j]) {
22.            x[i] = TRUE;   j = j – w[i];
23.        }
24.    }
25.    v = optp[n][m];
26.    delete optp;                    /* 释放工作单元 */
27.    return v;                       /* 返回最大价值 */
28. }
```

算法时间复杂性

根据规划函数

$$optp_i(j)=\begin{cases} optp_{i-1}(j) & j<w_j \\ max\{optp_{i-1}(j),optp_{i-1}(j-w_i)+p_i\} & j\geq w_j \end{cases}$$

备忘录需计算nm项，每项常数时间，计算时间为$\Theta(nm)$

动态规划算法小结

- 与蛮力算法相比较，动态规划算法利用了子问题优化函数间的依赖关系，时间复杂性有所降低；
- 动态规划算法的递归实现效率不高，原因在于同一子问题多次重复出现，每次出现都需要重新计算一遍；
- 动态规划算法的递推（迭代）实现，采用空间换时间策略，记录每个子问题首次计算结果，后面再用时就直接取值，每个子问题只算一次。

- 迭代过程：
  - 从最小的子问题算起；
  - 考虑计算顺序，以保证后面用到的值前面已经计算好；
  - 存储结构保存计算结果——备忘录。
- 解的追踪
  - 设计标记函数标记每步的决策；
  - 考虑根据标记函数跟踪解的算法。

动态规划算法的要素

- 划分子问题，确定子问题边界，将问题求解转变成为多步判断的过程；
- 定义动态规划函数，以该优化函数极大（或极小）值作为依据，确定是否满足优化原则；
- 列出动态规划函数的递推方程和边界条件；
- 自底向上计算，设计备忘录（表格）；
- 考虑是否需要设立标记函数；
- 用备忘录估计时间复杂度。