第7章　动态规划

Fibonacci sequence

The definition of the Fibonacci sequence

1. procedure f(n)
2. if n=1 or n=2 then return 1
3. else return f(n-1)+f(n-2)

It is concise, easy to write and debug, and, most of all, its abstraction

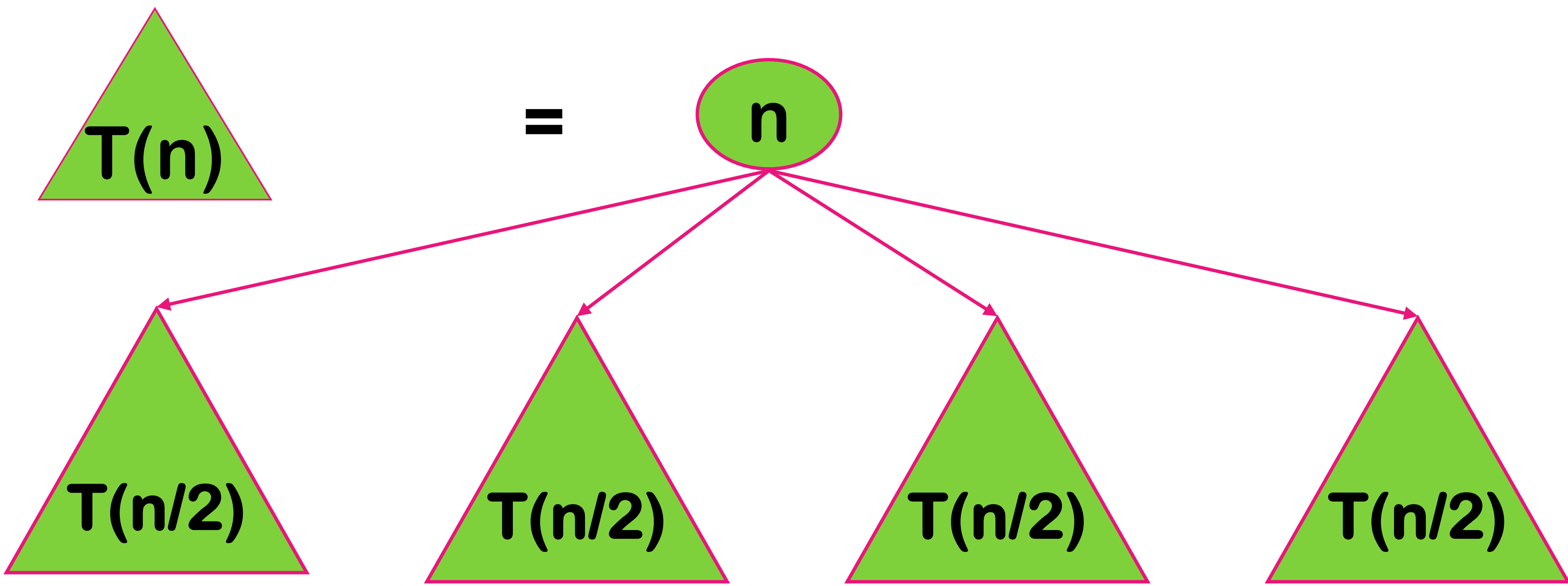
But it is far from being efficient.

Time complexity: $\Theta(\phi\text{\^{}}n)$

Why???

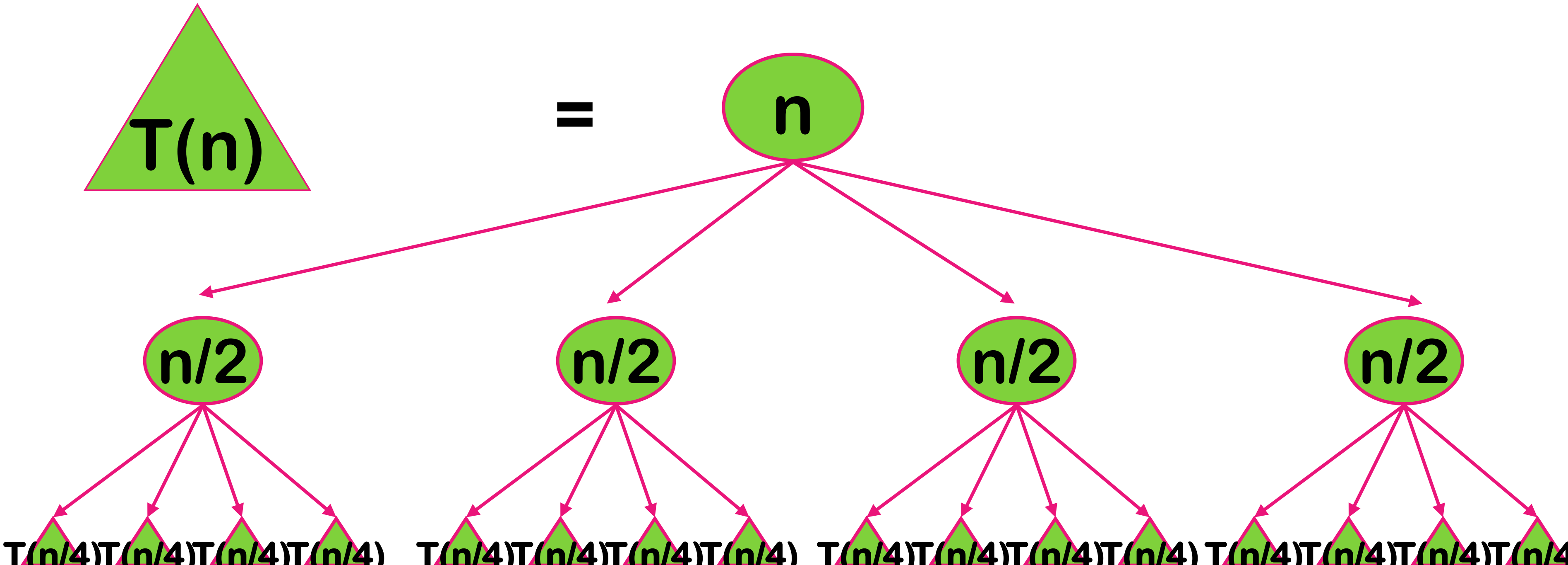
算法总体思想

- 动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题



算法总体思想

- 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许多次。



算法总体思想

- 如果能够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。

Those who cannot remember the past are doomed to repeat it.

-----George Santayana,
The life of Reason,
Book I: Introduction and
Reason in Common
Sense (1905)

动态规划基本步骤

- 找出最优解的性质，并刻划其结构特征。
- 递归地定义最优值。
- 以自底向上的方式计算出最优值。
- 根据计算最优值时得到的信息，构造最优解。

矩阵连乘问题

给定n个矩阵｛$A_1,A_2,\ldots,A_n$｝，其中A_i与A_i+1是可乘的，i=1,2 ,…,n-1。如何确定计算矩阵连乘积的计算次序，使得依此次序计算矩阵连乘积需要的数乘次数最少。

◆**穷举法**：列举出所有可能的计算次序，并计算出每一种计算次序相应需要的数乘次数，从中找出一种数乘次数最少的计算次序。

算法复杂度分析：
对于n个矩阵的连乘积，设其不同的计算次序为P(n)。
由于每种加括号方式都可以分解为两个子矩阵的加括号问题：$(A_1\ldots A_k)(A_k+1\ldots A_n)$可以得到关于P(n)的递推式如下：

$$P(n)=\begin{cases}1 & n=1\\ \sum_{k=1}^{n-1}P(k)P(n-k) & n>1\end{cases} \Rightarrow P(n)=\Omega(4^n/n^{3/2})$$

完全加括号的矩阵连乘积

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

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

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

16000, 10500, 36000, 87500, 34500

Assume we are given n+1 dimensions $r_1, r_2, \ldots, r_{n+1}$, where r_i and r_{i+1} are, respectively, the number of rows and columns in matrix M_i, $1 \le i \le n$.

We will write M_{ij} to denote the product of $M_i M_{i+1} \ldots M_j$. The cheapest cost of multiplying chain M_{ij}, denoted by C[i,j], is measured in terms of the number of scalar multiplications. Then

$$C[i, j] = \min_{i<k\le j}\{C[i, k-1] + C[k, j] + r_i r_k r_{j+1}\}$$

Particularly,

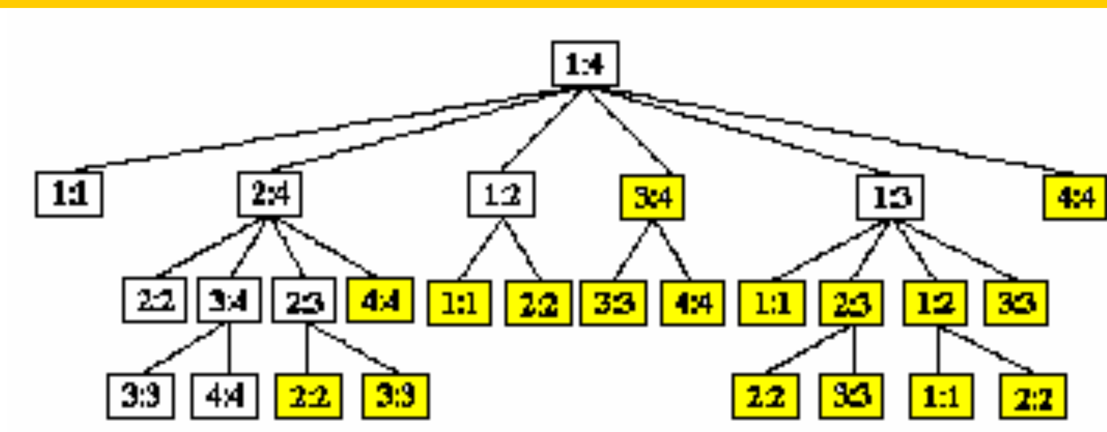
$$C[1, n] = \min_{1<k\le n}\{C[1, k-1] + C[k, n] + r_1 r_k r_{n+1}\}$$

分析最优解的结构

- 特征：计算A[i:j]的最优次序所包含的计算矩阵子链 A[i:k]和A[k+1:j]的次序也是最优的。
- 矩阵连乘计算次序问题的最优解包含着其子问题的最优解。这种性质称为**最优子结构性质**。问题的最优子结构性质是该问题可用动态规划算法求解的显著特征。

重叠子问题

- 递归算法求解问题时，每次产生的子问题并不总是新问题，有些子问题被反复计算多次。这种性质称为**子问题的重叠性质。**
- 动态规划算法，对每一个子问题只解一次，而后将其解保存在一个表格中，当再次需要解此子问题时，只是简单地用常数时间查看一下结果。
- 通常不同的子问题个数随问题的大小呈多项式增长。因此用动态规划算法只需要多项式时间，从而获得较高的解题效率。



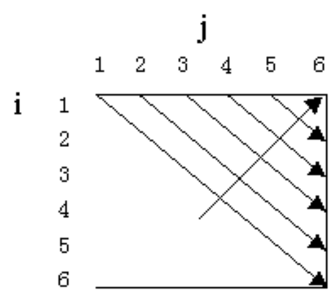
用动态规划法求最优解

```
public static void matrixChain(int [] p, int [][] m, int [][] s)
  {
    int n=p.length-1;
    for (int i = 1; i <= n; i++) m[i][i] = 0;
    for (int r = 2; r <= n; r++)
      for (int i = 1; i <= n - r+1; i++) {
        int j=i+r-1;
        m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j];
        s[i][j] = i;
        for (int k = i+1; k < j; k++) {
          int t = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
          if (t < m[i][j]) {
            m[i][j] = t;
            s[i][j] = k;}
          }
        }
  }
```

A1	A2	A3	A4	A5	A6
30×35	35×15	15×5	5×10	10×20	20×25

$$m[2][5]=\min\begin{cases} m[2][2]+m[3][5]+p_1p_2p_5 &= 0+2500+35\times15\times20=13000 \\ m[2][3]+m[4][5]+p_1p_3p_5 &= 2625+1000+35\times5\times20=7125 \\ m[2][4]+m[5][5]+p_1p_4p_5 &= 4375+0+35\times10\times20=11375 \end{cases}$$

算法复杂度分析：
算法**matrixChain**的主要计算量取决于算法中对r，i和k的3重循环。循环体内的计算量为O(1)，而3重循环的总次数为$O(n^3)$。因此算法的计算时间上界为$O(n^3)$。算法所占用的空间显然为$O(n^2)$。



(a) 计算次序

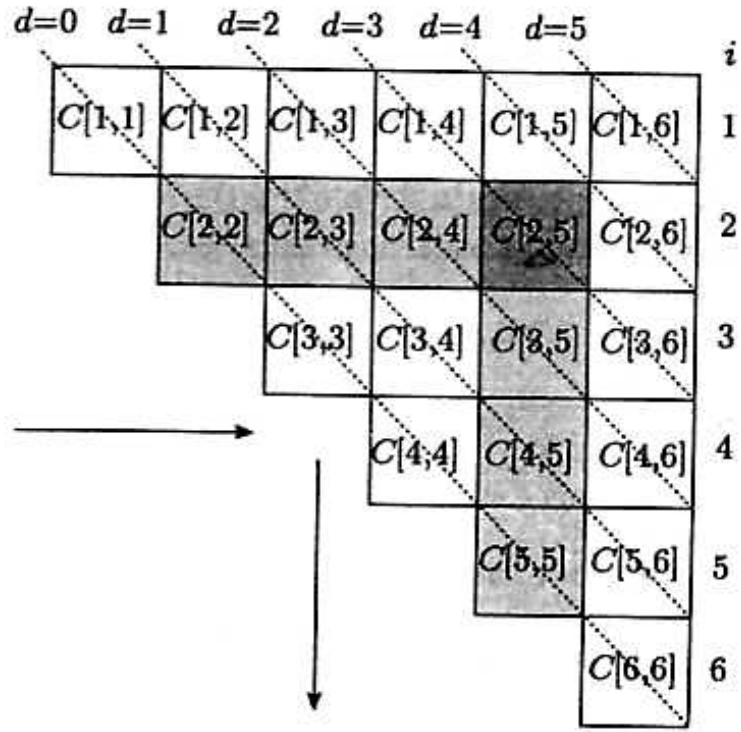
i \ j	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

(b) m[i][j]

i \ j	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

(c) s[i][j]

Illustration of matrix chain multiplication



Algorithm 7.2 MATCHAIN

Input: An array r[1..n+1] of positive integers corresponding to the dimensions of a chain of n matrices, where r[1..n] are the number of rows in the n matrices and r[n+1] is the number of columns in M_n

Output: The least number of scalar multiplications required to multiply the n matrices

```
for i←1 to n     {Fill in diagonal d_0}
    C[i,i]←0
end for
for d←1 to n-1     {Fill in diagonals d_1 to d_{n-1}}
    for i←1 to n-d     {Fill in entries in diagonal d_i}
        j←i+d
        comment: The next three lines computes C[i,j]
        C[i,j]←∞
        for k←i+1 to j
            C[i,j]←min{C[i,j],C[i,k-1]+C[k,j]+r[i]r[k]r[j+1]}
        end for
    end for
end for
return C[1,n]
```

Time: $\Theta(n^3)$

Space: $\Theta(n^2)$

The longest common subsequence problem

Given two strings A and B of length n and m, respectively, over an alphabet Σ, determine the length of the longest subsequence that is common to both A and B. Here a subsequence of $A=a_1a_2...a_n$ is a string of the form $a_{i_1}a_{i_2}...a_{i_k}$, where each i_j is between 1 and n and $1 \leq i_1 < i_2 < ... < i_k \leq n$

Brute-force method: enumerate all the 2^n subsequence of A, and for each subsequence determine if it is also a subsequence of B in $\Theta(m)$ time. The running time is $\Theta(m*2^n)$

Let $A=a_1a_2...a_n$ and $B=b_1b_2...b_m$

Let L[i,j] denote the length of a longest common subsequence of $a_1a_2...a_i$ and $b_1b_2...b_j$. Note that i or j may be zero, in which case one or both of $a_1a_2...a_i$ and $b_1b_2...b_j$ may be the empty string.

Then we have

$$L[i,j]=\begin{cases}0 & \text{if } i=0 \text{ or } j=0\\ L[i-1,j-1]+1 & \text{if } i>0, j>0 \text{ and } a_i=b_j\\ \max\{L[i,j-1],L[i-1,j]\} & \text{if } i>0, j>0 \text{ and } a_i\neq b_j\end{cases}$$

Algorithm 7.1 LCS

Input: Two strings A and B of length n and m, respectively, over an alphabet Σ

Output: The length of the longest common subsequence of A and B

```
for i←0 to n
    L[i,0]←0
end for
for j←0 to m
    L[0,j]←0
end for
for i←1 to n
    for j←1 to m
        if ai=bj then L[i,j]←L[i-1,j-1]+1
        else L[i,j]←max{L[i,j-1],L[i-1,j]}
        end if
    end for
end for
return L[n,m]
```

Theorem 7.1 An optimal solution to the longest common subsequence problem can be found in $\Theta(nm)$ time and $\Theta(\min\{m,n\})$ space

Algorithm 7.1pro LCS

```
for i←0 to n
    L[i,0]←0
end for
for j←0 to m
    L[0,j]←0
end for
for i←1 to n
    for j←1 to m
        if ai=bj then L[i,j]←L[i-1,j-1]+1, b[i,j]←"\"
        else
            if L[i-1,j]≥L[i,j-1] then
                L[i,j]←L[i-1,j], b[i,j]←"↑"
            else
                L[i,j]←L[i,j-1], b[i,j]←"←"
            end if
        end if
    end for
end for
return L[n,m] and b[n,m]
```

print-LCS(b,A,i,j)

```
if i=0 or j=0 then return
if b[i,j]= "\" then
    print-LCS(b,A,i-1,j-1)
    print ai
else
    if b[i,j]= "↑" then print-LCS(b,A,i-1,j)
    else print-LCS(b,A,i,j-1)
end if
```

最短路问题

Dijkstra algorithm, 1959

- 计算两节点之间或一个节点到所有节点之间的最短路

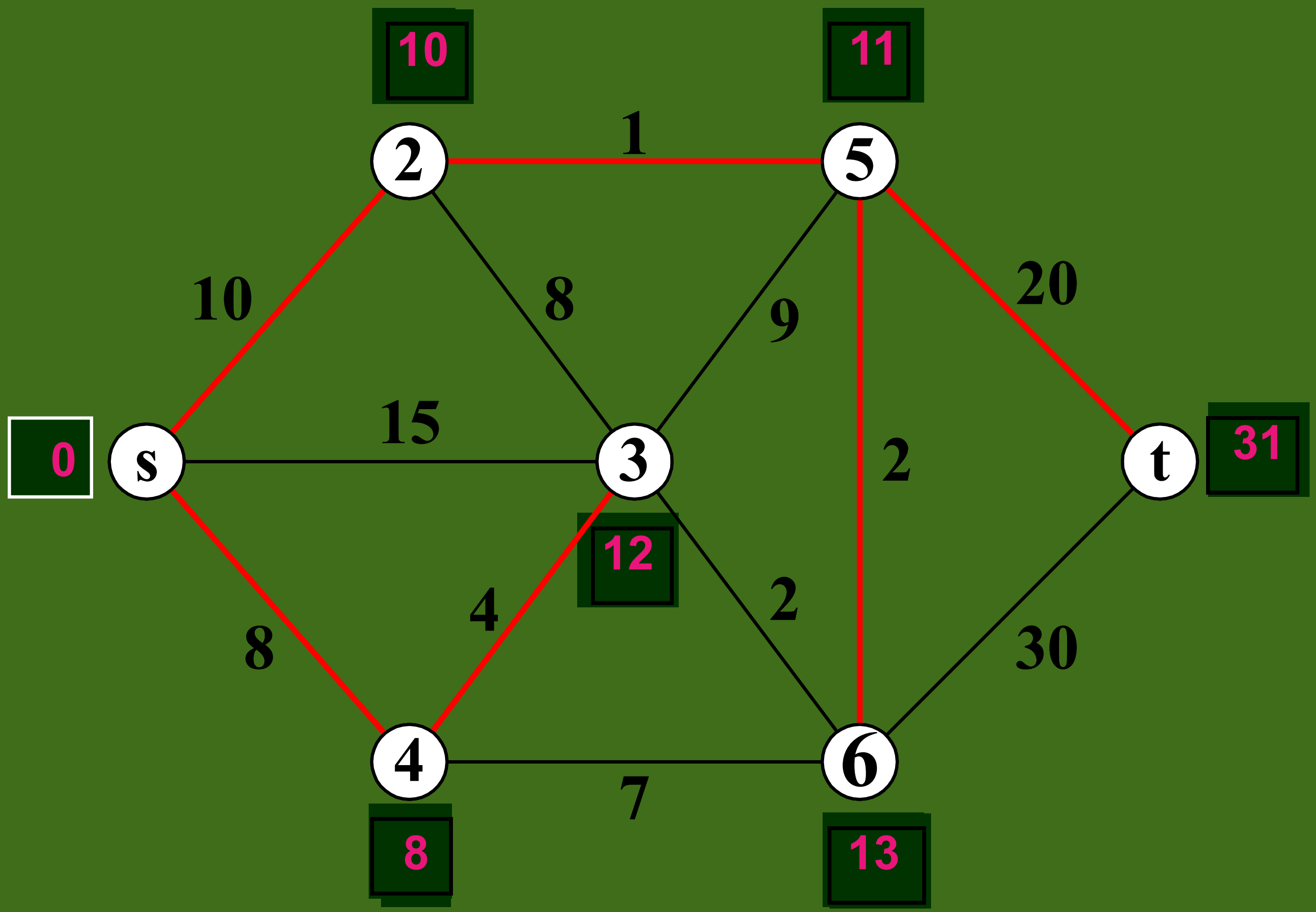
令 d_{ij} 表示 v_i 到 v_j 的直接距离(两点之间有边)，若两点之间没有边，则令 $d_{ij} = \infty$，若两点之间是有向边，则 $d_{ji} = \infty$；令 $d_{ii} = 0$，s 表示始点，t 表示终点

0、令始点T_s=0，并用□框住，所有其它节点临时标记 T_j=∞ ；

1、从 v_s 出发，对其相邻节点 v_{j1} 进行临时标记，有 $T_{j1}=d_{s,j1}$ ；

2、在所有临时标记中找出最小者，并用□框住，设其为 v_r 。若此时全部节点都永久标记，算法结束；否则到下一步；

3、从新的永久标记节点 v_r 出发，对其相邻的临时标记节点进行再标记，设 v_{j2} 为其相邻节点，则 $T_{j2}=\min\{T_{j2}, T_r+d_{r,j2}\}$，返回第2步。



10
11
1
2
5
10
8
9
20
0
s
15
3
2
t
31
12
4
2
8
30
4
6
7
8
13

*Dijkstra*最短路算法的特点和适应范围

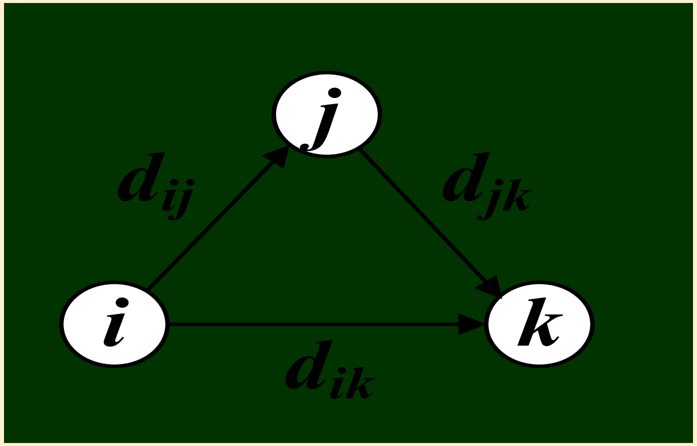
- 每次迭代只有一个节点获得永久标记，若有两个或两个以上节点的临时标记同时最小，可任选一个永久标记；总是从一个新的永久标记开始新一轮的临时标记，是一种深探法
- 被框住的永久标记 T_j 表示 v_s 到 v_j 的最短路，因此 要求 $d_{ij}\geq 0$ ，第 k 次迭代得到的永久标记，其最短路中最多有 k 条边，因此最多有$n-1$ 次迭代
- 可以应用于简单有向图和混合图，在临时标记时，所谓相邻必须是箭头指向的节点；若第 $n-1$ 次迭代后仍有节点的标记为 ∞，则表明 v_s 到该节点无有向路径
- 如果只求 v_s 到 v_t 的最短路，则当 v_t 得到永久标记算法就结束了；但算法复杂度是一样的
- 应用 *Dijkstra* 算法 $n-1$ 次 ，可以求所有点间的最短路
- v_s 到所有点的最短路也是一棵生成树，但不是最小生成树

Warshall-Floyd算法（1962)

- Warshall-Floyd算法可以解决有负权值边(弧)的最短路问题
- 该算法是一种整体算法，一次求出所有点间的最短路
- 该算法不允许有负权值回路，但可以发现负权值回路
- 该算法基于基本的三角运算

定义 对给定的点间初始距离矩阵$\{d_{ij}\}$，令$d_{ii}=\infty$，对所有 i。对一个固定点 j，运算 $d_{ik}=\min\{d_{ik}, d_{ij}+d_{jk}\}$, 对所有 $i, k \neq j$, 称为三角运算。(注意，这里允许 $i=k$)

定理 依次对 $j=1,2,\ldots,n$ 执行三角运算，则 d_{ik} 最终等于 i 到 k 间最短路的长度。



The all-pairs shortest path problems

Let G=(V,E) be a directed graph in which each edge (i,j) has a nonnegative length l[i,j]. If there is no edge from vertex i to vertex j, then l[i,j]=∞. The problem is to find the distance from each vertex to all other vertices, where the distance from vertex x to vertex y is the length of a shortest path from x to y.

Assume V={1,2,...,n}, let i and j be two different vertices in V. Define $d_k[i,j]$ to be the length of a shortest path from i to j that does not pass through any vertex in {k+1,k+2,...,k+n}. Then we have

$$d_k[i,j] = \begin{cases} l[i,j] & \text{if } k = 0 \\ \min\{d_{k-1}[i,j], d_{k-1}[i,k] + d_{k-1}[k,j]\} & \text{if } 1 \le k \le n \end{cases}$$

Algorithm 7.3 FLOYD

Input: An $n \times n$ matrix l[1..n,1..n] is the length of the edge (i,j) in a directed graph G=({1,2,...,n},E)

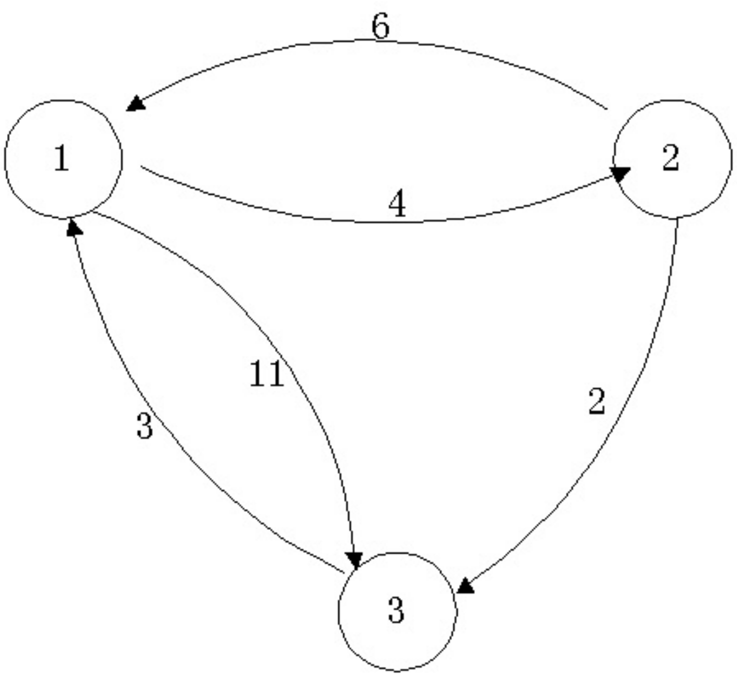
Output: A matrix D with D[i,j]=the distance from i to j

```
D←l     {copy the input matrix l into D}
for k←1 to n
    for i←1 to n
        for j←1 to n
            D[i,j]=min{D[i,j],D[i,k]+D[k,j]}
        end for
    end for
end for
```

Time: Θ(n^3)

Space: Θ(n^2)

最短路径的例子



$$A = \begin{pmatrix} \infty & 4 & 11 \\ 6 & \infty & 2 \\ 3 & \infty & \infty \end{pmatrix}$$

最短路径

$$C^{(0)} = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix}$$

$C^{(0)}(i, j)=c(i, j)$

$$C^{(1)} = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$$

$C^{(1)}(3,2)=\min\{C^{(0)}(3,2),C^{(0)}(3,1)+C^{(0)}(1,2)\}=7$

$$C^{(2)} = \begin{pmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$$

$C^{(2)}(1,3)=\min\{C^{(1)}(1,3), C^{(1)}(1,2)+C^{(1)}(2,3)\}$

$=\min\{11,4+2\}=6$

$$C^{(3)} = \begin{pmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$$

$C^{(3)}(1,2)=\min\{4,6+7\}=4$

$C^{(3)}(2,1)=\min\{6,2+3\}=5$

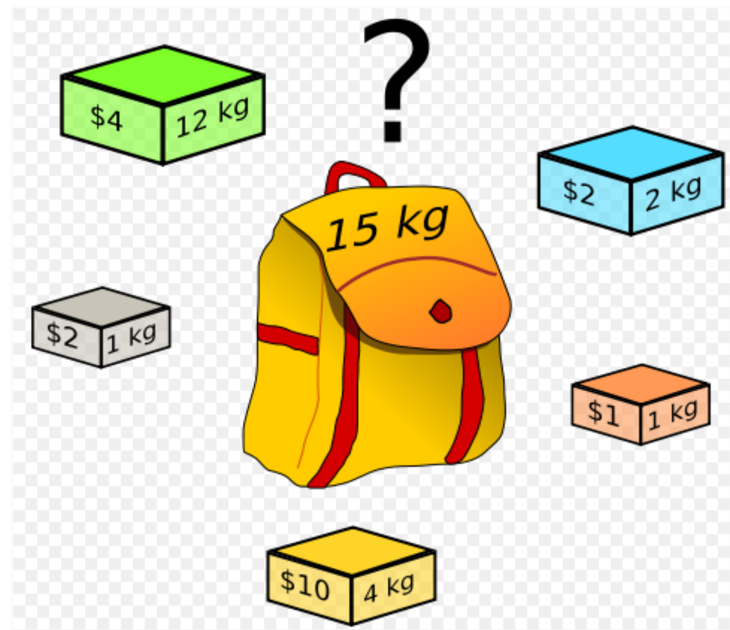
0-1背包问题

给定n种物品和一背包。物品i的重量是w_i，其价值为v_i，背包的容量为C。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大?

0-1背包问题是一个特殊的整数规划问题。

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

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le C \\ x_i \in \{0,1\}, 1 \le i \le n \end{cases}$$



Knapsack Problem问题类型

- Fractional Knapsack Problem:
  - 物品可以被任意分割
  - 一般采用贪婪算法(Greedy Approach)

- 0/1 Knapsack Problem:
  - 物品不可分割
  - 一般采用动态规划法(Dynamic Programming)

- 贪心法求解 最佳装载 背包问题
- 动态规划法求解 0/1 背包问题

2-动态规划法求解0/1背包问题

动态规划法设计算法一般分成三个阶段：

（1）分段：将原问题分解为若干个相互重叠的子问题；

（2）分析：分析问题是否满足最优性原理，找出动态规划函数的递推式；

（3）求解：利用递推式自底向上计算，实现动态规划过程。

❖ 动态规划法利用问题的最优性原理，以自底向上的方式从子问题的最优解逐步构造出整个问题的最优解。

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：

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le C \\ x_i \in \{0,1\} \quad (1 \le i \le n) \end{cases} \qquad \text{（式2.1）}$$

$$\max \sum_{i=1}^{n} v_i x_i \qquad \text{（式2.2）}$$

于是，问题归结为寻找一个满足约束条件式2.1，并使目标函数式2.2达到最大的解向量$X=(x_1, x_2, \ldots, x_n)$。

- [动态规划函数]:

3 第 i 物的重量比背包目前可承受之重量还重

1装入0个物品

2 背包容量为0

$$V(i,j)=\begin{cases}0 & \text{if } i=0 \text{ or } j=0 \\ V(i-1,j) & \text{if } j<w_i \\ \max(v_i+V(i-1,j-w_i),V(i-1,j)) & \text{if } j\ge w_i\end{cases}$$

（式2.3）

（式2.4.1）

（式2.4.2）

5 放入第 i 物后可得的价值

4 不放入第 i 物可得的价值

式2.3表明：把前面i个物品装入容量为0的背包和把0个物品装入容量为j的背包，得到的价值均为0。

式2.4的第一个式子表明：如果第i个物品的重量大于背包的容量，则装入前i个物品得到的最大价值和装入前i–1个物品得到的最大价值是相同的，即物品i不能装入背包；第二个式子表明：如果第i个物品的重量小于背包的容量，则会有以下两种情况：

（1）如果把第i个物品装入背包，则背包中物品的价值等于把前i–1个物品装入容量为$j-w_i$的背包中的价值加上第i个物品的价值v_i；

（2）如果第i个物品没有装入背包，则背包中物品的价值就等于把前i–1个物品装入容量为j的背包中所取得的价值。显然，取二者中价值较大者作为把前i个物品装入容量为j的背包中的最优解。

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。

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。

		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
$w_2=2$ $v_2=3$	2	0	0	6	6	9	9	9	9	9	9	9
$w_3=6$ $v_3=5$	3	0	0	6	6	9	9	9	9	11	11	14
$w_4=5$ $v_4=4$	4	0	0	6	6	9	9	9	10	11	13	14
$w_5=4$ $v_5=6$	5	0	0	6	6	9	9	12	12	15	15	15

如何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。依此类推，直到确定第1个物品是否被装入背包中为止。由此，得到如下函数：

$$x_i=\begin{cases} 0 & V(i,j)=V(i-1,j) \\ 1,\quad j=j-w_i & V(i,j)>V(i-1,j) \end{cases}$$ （式6.13）

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

Algorithm 7.4 KNAPSACK

Input: A set of items $U=\{u_1 \ldots u_n\}$ with sizes $s_1,\ldots,s_n$ and values $v_1,\ldots,v_n$ and a knapsack capacity C

Output: The maximum value of the problem

```
for i←0 to n
    V[i,0]←0
end for
for j←0 to
    V[0,j]←
end for
for i←1 to
    for j←1 to C
        V[i,j]←V[i-1,j]
        if s_i≤j then V[i,j]←max{V[i,j],V[i-1,j-s_i]+vi}
    end for
end for
return V[n,C]
```

算法复杂度分析：

从m(i，j)的递归式容易看出，算法需要O(nc)计算时间。当背包容量c很大时，算法需要的计算时间较多。例如，当$c>2^n$时，算法需要$\Omega(n2^n)$计算时间。

Time: $\Theta(nC)$

Space: $\Theta(C)$

But it is a pseudopolynomial time algorithm!

作业

- 7.7 7.9 7.21