# *Algoritmi e Strutture Dati*
# *03/05/2013*

## Esercizio 1

L'equazione di ricorrenza è la seguente:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(\lfloor n/2 \rfloor) + T(\lfloor n/3 \rfloor) + n/2 \end{cases}$$

Proviamo a dimostrare che $T(n) = \Theta(n)$. E' facile vedere che $T(n) = \Omega(n)$, in quanto:

$$T(n) = (\lfloor n/2 \rfloor) + T(\lfloor n/3 \rfloor) + n/2 \geq cn$$

L'ultima diseguazione è valida per $c \leq 1/2$.
Per quanto riguarda il limite superiore, proviamo con $T(n) = O(n)$. Dobbiamo quindi dimostrare che $\exists c > 0, \exists m \geq 0 : T(n) \leq cn, \forall n \geq m$. Proviamolo con il metodo di sostituzione.

**Caso base** $T(1) = 1 \leq c \cdot 1 \Rightarrow c \geq 1$

**Ipotesi induttiva** $\forall n' < n : T(n') \leq cn'$

**Passo induttivo**

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lfloor n/3 \rfloor) + n/2 \\ &\leq T(n/2) + T(n/3) + n/2 \\ &\leq cn/2 + cn/3 + n/2 \\ &= \frac{5}{6}cn + n/2 \\ &= n\left(\frac{5}{6}c + \frac{1}{2}\right) \\ &\leq cn \end{aligned}$$

L'ultima diseguazione è vera per $c \geq 3$.

## Esercizio 2

Proponiamo un algoritmo che opera utilizzando due stack $S_u$ e $S_v$. In ciascuno stack verranno inseriti i nodi presenti sul cammino da $u$ e $v$ (inclusi), rispettivamente, fino alla radice. Al termine di questa operazione, in cima a entrambi gli stack si troverà necessariamente un riferimento alla radice dell'albero. Estraiamo contemporaneamente un elemento da entrambi gli stack; l'ultimo elemento uguale estratto da entrambi sarà l'antenato comune. Occorre prestare attenzione al fatto che, durante questo processo, uno dei due stack potrebbe svuotarsi (in particolare ciò accade se uno dei due nodi è antenato dell'altro, e quindi è esso stesso l'antenato di entrambi).

La complessità è $O(n)$, caso pessimo che si ottiene quando l'albero in realtà è una catena di $n$ elementi.

```
TREE ancestor(TREE T, TREE u, TREE v)
    STACK S_u ← new Stack()
    STACK S_v ← new Stack()
    repeat
    |   S_u.push(u)
    |   u ← u.parent()
    until u ≠ nil
    repeat
    |   S_v.push(v)
    |   v ← v.parent()
    until u ≠ nil
    TREE Ret
    repeat
    |   x_u ← S_u.pop()
    |   x_v ← S_v.top()
    |   if x_u = x_v then
    |   |   Ret ← x_u
    until x_u ≠ x_v or S_u.isEmpty() or S_v.isEmpty()
    if x_u ≠ x_v then
    |   return Ret
    else if S_u.isEmpty() then
    |   return v
    else
    |   return u
```

## Esercizio 3

Questo problema si può risolvere efficientemente applicando l'algoritmo di visita in ampiezza (BFS) di un grafo non orientato. Non è necessario costruire esplicitamente il grafo da esplorare. Ciascuna cella della griglia è un nodo del grafo da esplorare; esiste un arco tra ogni coppia di nodi adiacenti tali che entrambi i nodi corrispondano a caselle libere (cioè non occupate).
Per semplificare il codice, sono stati creati due vettori ausiliari $di$, $dj$, di quattro elementi; presi a coppie, rappresentano gli scostamenti orizzontale e verticale delle quattro possibili mosse.
La matrice $D$ delle distanze non è altro che il vettore *erdos* utilizzato nella visita in ampiezza del libro; viene inizializzata a $+\infty$.

Il costo computazionale è pari a $n^2$, in quanto corrisponde alla visita in ampiezza di un grafo con $n^2$ nodi e meno di $4n^2$ archi.

```
integer gridDistance(integer[][] G, integer n)
    integer[][] D ← new integer[1...n][1...n]
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← +∞
    { Scostamento indici i, j delle quattro possibili mosse }
    integer[] di ← {0, 0, −1, +1}
    integer[] dj ← {−1, +1, 0, 0}
    D[1, 1] ← 0;
    QUEUE Q ← new Queue()
    Q.enqueue((1, 1))
    while not Q.isEmpty() do
        (i, j) ← Q.dequeue()                                   % Estrai inizio coda
        if i == n and j == n then
            return D[i, j]
        for k ← 1 to 4 do
            if G[i + di[k], j + dj[k]] = 0 and D[i + di[k], j + dj[k]] = +∞ then
                D[i + di[k], j + dj[k]] = D[i, j] + 1
                Q.enqueue(i + di[k], j + dj[k])
    return +∞                                                  % Non esiste alcun cammino
```

Un modo alternativo per risolvere il problema è quello di costruire il grafo ed eseguire la procedura erdos() a partire da $(1, 1)$. Utilizzando una funzione calcola$(i, j, n)$ che mappa gli indici bidemensionali $(i, j)$ in un indice lineare in $1 \dots n^2$, il codice si risolve in poche righe.

```
integer gridDistance(integer[][] G, integer n)
    GRAPH Gr ← new Graph()
    integer[] di ← {0, 0, −1, +1}
    integer[] dj ← {−1, +1, 0, 0}
    for i ← 1 to n do
        for j ← 1 to n do
            Gr.insertNode(calcola(i, j, n))
            for k ← 1 to 4 do
                if 1 ≤ i + di[k] ≤ n and 1 ≤ j + dj[j] ≤ n then
                    Gr.insertEdge(calcola(i, j, n), calcola(i + di[k], j + dj[k], n))
    integer[] D ← new integer[1...n^2]
    erdos(Gr, calcola(1, 1, n), D)                             % Non siamo interessati al vettore dei padri
    return D[calcola(n, n, n)]
```

```
integer calcola(integer i, integer j, integer n)
    return (i − 1) · n + j
```

## Esercizio 4

Sia $best[i, j]$ il costo minimo che si deve pagare per andare dal porto $i$ al porto $j$, con $i < j$. E' possibile descrivere ricorsivamente questo costo nel modo seguente:

$$best[i, j] = \begin{cases} C[i, j] & j = i + 1 \\ \min\{C[i, j], \min_{i<k<j}\{best[i, k] + best[k, j]\}\} & \end{cases}$$

Traducendo questo codice tramite memoization, si ottiene:

```
boat(integer[][] C, integer n, integer[][] best, integer i, j)
if j = i + 1 then
    return C[i, j]
if best[i, j] = ⊥ then
    best[i, j] ← C[i, j]
    for k = i + 1 to j − 1 do
        if best[i, k] + best[k, j] < C[i, j] then
            best[i, j] ← best[i, k] + best[k, j]
return best[i, j]
```

La complessità è $O(n^3)$.
Esiste una soluzione migliore, che utilizza un solo indice per la variabile $best$: $best[i]$ è il costo minimo per andare dal porto $i$ al porto $n$. Chiaramente, il caso base si ottiene con $best[n] = 0$. Altrimenti, si considera la possibilità di fermarsi in uno qualsiasi dei porti compresi fra $i$ (escluso) e $n$ (incluso).

$$best[i] = \begin{cases} 0 & i = n \\ \min_{i<j\leq n}\{C[i, j] + best[j]\} \end{cases}$$

Al termine, il valore cercato si troverà in $best[1]$.
Traducendo questo codice tramite programmazione dinamica, si ottiene il programma seguente

```
integer boat(integer[][] C, integer n)
integer[] best ← new integer[1 . . . n]
best[n] ← 0
for i ← n − 1 downto 1 do
    best[i] ← +∞
    for j ← i + 1 to n do
        if C[i, j] + best[j] < best[i] then
            best[i] ← C[i, j] + best[j]
return best[1]
```

La complessità è $O(n^2)$, per il doppio ciclo annidato.
Presento di seguito una versione leggermente modificata che invece di ritornare il costo minimo, stampa il percorso da seguire. Questo si ottiene utilizzando un secondo vettore di appoggio $next$, che memorizza per tutti i porti diversi da $n$, l'indice del prossimo porto da visitare. A questo punto è sufficiente visitare saltare dal porto $i$ al porto $next[i]$, dal porto $next[i]$ al porto $next[next[i]]$ e così via.

```
boat(integer[][] C, integer n)
integer[] best ← new integer[1 . . . n]
integer[] next ← new integer[1 . . . n]
best[n] ← 0
for i ← n − 1 downto 1 do
    best[i] ← +∞
    for j ← i + 1 to n do
        if C[i, j] + best[j] < best[i] then
            best[i] ← C[i, j] + best[j]
            next[i] ← j
integer i ← 1
while i ≠ n do
    print i
    i ← next[i]
print n
```

Infine, si può notare che si può risolvere il problema interpretando la matrice come una matrice di pesi di un grafo orientato pesato e risolverlo utilizzando Dijkstra.