Corso di Laurea in Informatica *A.A. 2008-2009*

Soluzione Problema 1.3: Tribonacci

Esame di Algoritmi 1 *Docente: Irene Finocchi*

Problema 1.3. Il Professor Tribonacci, dell'Università di Pizza, ha definito la sequenza dei numeri di Tribonacci:

$$X_k = \begin{cases} 0 & \text{se } k = 0 \\ 1 & \text{se } k = 1, 2 \\ X_k = X_{k-1} + X_{k-2} + X_{k-3} & \text{se } k \geq 3 \end{cases}$$

Il Prof. Tribonacci sostiene che l'algoritmo più efficiente per calcolare l'n-mo numero di Tribonacci X_n è dato dal seguente pseudocodice:

```
    algoritmo tribonacci(intero n) → intero
1.      if ( n = 0 ) then return 0
2.      else if ( n ≤ 2 ) then return 1
3.      else return tribonacci(n − 1) + tribonacci(n − 2) +
                    tribonacci(n − 3)
```

Sia $T(n)$ il tempo di esecuzione dell'algoritmo `tribonacci` su input n.

(a) Scrivere una relazione di ricorrenza per $T(n)$.

(b) Stimare $T(n)$ per n grande.

(c) Descrivere un algoritmo più efficiente per lo stesso problema, e dimostrare il suo tempo di esecuzione.

Soluzione. Illustriamo di seguito le soluzioni ai problemi dati.

(a) Una relazione di ricorrenza per $T(n)$ è data da

$$T(n) = \begin{cases} T(n-1) + T(n-2) + T(n-3) + c_1, & \text{se } n > 2 \\ c_2, & \text{se } n \leq 2 \end{cases}$$

dove c_1 e c_2 sono costanti positive opportune.

(b) Al fine di stimare $T(n)$ per n grande, ne diamo una delimitazione inferiore. Semplifichiamo preliminarmente la ricorrenza ottenuta in (a), utilizzando la proprietà che $T(n)$ è una funzione monotona crescente.

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + T(n-3) + c_1 \\ &\geq 3\ T(n-3) + c_1 \\ &\geq 3^2\ T(n-6) + 3c_1 + c_1 \\ &\geq 3^3\ T(n-9) + 3^2 c_1 + 3c_1 + c_1 \\ &\geq 3^k\ T(n-3k) + c_1\ \textstyle\sum_{i=0}^{k-1} 3^i \\ &\geq 3^{\lceil \frac{n}{3} \rceil}\ c, \end{aligned}$$

per una opportuna costante c, e quindi il tempo di esecuzione dell'algoritmo `tribonacci` cresce asintoticamente almeno come la funzione $3^{\frac{n}{3}}$, che è esponenziale in n.

(c) Un possibile algoritmo più efficiente per lo stesso problema è dato dal seguente pseudocodice:

```
    algoritmo tribonacci3(intero n) → intero
1.      Sia Fib un array di n interi
2.      Fib[0] ← 0
3.      Fib[1] ← Fib[2] ← 1
4.      for i = 3 to n do
5.          Fib[i] ← Fib[i − 1] + Fib[i − 2] + Fib[i − 3]
6.      return Fib[n]
```

Notiamo le analogie con l'algoritmo `fibonacci3` di Figura 1.6 a pagina 9 del libro. Ogni iterazione del ciclo richiede tempo costante. Ci sono $O(n)$ iterazioni, e quindi il tempo totale è $O(n)$. Esattamente come per l'algoritmo `fibonacci3`, il tempo di esecuzione di `tribonacci3` è quindi

$$T(n) = O(n)$$

Analogamene al caso dei numeri di Fibonacci, non è necessario mantenere un array di dimensione n per memorizzare i numeri di Tribonacci. Ispirandoci all'algoritmo `fibonacci4` descritto nella Figura 1.8 a pagina 12 del libro, otteniamo così l'algoritmo `tribonacci4`:

```
    algoritmo tribonacci4(intero n) → intero
1.      a ← 0, b ← 1, c ← 1
2.      for i = 3 to n do
3.          d ← a + b + c
4.          a ← b
5.          b ← c
6.          c ← d
7.      return c
```

Ancora una volta ci sono $O(n)$ iterazioni, ognuna delle quali richiede tempo costante. Anche l'algoritmo `tribonacci4`, come l'algoritmo `fibonacci4`, ha quindi tempo di esecuzione $O(n)$.

L'algoritmo `tribonacci4` può essere ancora sostanzialmente migliorato: seguendo l'algoritmo `fibonacci6` descritto nella Figura 1.11 di pagina 15 del libro, è infatti possibile progettare un algoritmo basato su potenze ricorsive. Consideriamo infatti la seguente uguaglianza:

$$\begin{pmatrix} X_{n+2} \\ X_{n+1} \\ X_n \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}^n \begin{pmatrix} X_2 \\ X_1 \\ X_0 \end{pmatrix}$$

che può essere facilmente dimostrata per induzione in n, seguendo la filosofia del Lemma 1.3 a pagina 13 del libro. Questo ci suggerisce di modificare l'algoritmo `fibonacci6` nel modo seguente:

```
    algoritmo tribonacci6(intero n) → intero
1.      A ← ( 1 1 1 )
            ( 1 0 0 )
            ( 0 1 0 )
2.      M ←potenzaDiMatrice(A, n − 1)
3.      return M[0][0]

    funzione potenzaDiMatrice(matrice A, intero k) → matrice
4.      if ( k ≤ 1 ) then M ← ( 1 0 0 )
                              ( 0 1 0 )
                              ( 0 0 1 )
5.      else
6.          M ←potenzaDiMatrice(A, k/2)
7.          M ← M · M
8.      if ( k è dispari ) then M ← M · A
9.      return M
```

Dato che la potenza n-esima di una matrice è calcolata in tempo $O(\log n)$ con il metodo dei quadrati ripetuti, il tempo totale di questo algoritmo è $O(\log n)$.