Esercizi sulla complessità di frammenti di pseudo-codice

Esercizio 1

Si determini la complessità temporale del seguente frammento di pseudo-codice in funzione di n.

```
integer i ← 2
while i ≤ n do
    b[j] ← a[i]
    j ← j + 1
    i ← i * i
```

Soluzione

Il ciclo contiene solo istruzioni elementari; con il criterio del costo costante, ognuna di queste ha costo Θ(k). La complessità del frammento di codice quindi dipende solo da quante volte viene ripetuto il ciclo while, che dipende dal valore di n e dalla crescita della variabile i.

i ad ogni iterazione del ciclo viene moltiplicata per se stessa; essa assume, ad ogni iterazione, i seguenti valori: 2, 4, 16, 256, …, cioè 2^1, 2^2, 2^4, 2^8,…, cioè 2^(2^0), 2^(2^1), 2^(2^2), 2^(2^3), …

Alla x-esima iterazione h vale quindi 2^(2^x).

Per determinare quante siano le iterazioni del ciclo, vogliamo sapere qual è il valore di x per cui 2^(2^x) = N+1. Noi vogliamo sapere il numero di iterazioni che porta i ad essere pari a n, cioè il valore di x per cui 2^(2^x)=n, quindi il numero di iterazioni è Θ(log log n).

La complessità del frammento di codice è quindi **Θ(log log n)**.

Esercizio 2

Calcolare la complessità del seguente frammento di pseudo-codice in funzione di N:

```
integer s ← 0, j ← 0, k ← 0
for integer i ← 0 to N do
    j ← 1
    while j < N do
        k ← 1
        while k < N do
            s ← s + 1
            k ← k * 3
        j ← j * 2
```

Soluzione

Indichiamo il primo ciclo con (1), il secondo con (2) e il terzo con (3). I tre cicli (1), (2) e (3) sono annidati ma non sono correlati quindi li possiamo considerare separatamente; in altre parole, il numero di iterazioni di un ciclo interno non dipende dalle iterazioni del ciclo esterno. E' possibile quindi determinare la complessità del frammento di codice in funzione del numero di iterazioni di ciascun ciclo in maniera separata.

Consideriamo quindi prima il **ciclo (1)**:

Il ciclo è un banale ciclo for in cui i viene incrementato di una unità.

Alla x-esima iterazione i vale quindi x. Noi vogliamo sapere il numero di iterazioni che porta i ad essere pari a N, cioè N.

Il numero di iterazioni del **ciclo (1)** è quindi **N**.

Analizziamo ora il **ciclo (2)**:

j, ad ogni iterazione del ciclo while, viene moltiplicato per 2. j cresce nel seguente modo: 1,2, 4, 8, 16, …, cioè 2^0, 2^1, 2^2, 2^3,…

Alla x-esima iterazione j vale quindi 2^x. Noi vogliamo sapere il numero di iterazioni che porta j ad essere pari a N, cioè il valore di x per cui 2^x=N, cioè log N.

Il numero di iterazioni del **ciclo (2)** è quindi **log N**.

Analizziamo ora il **ciclo (3)**:

k, ad ogni iterazione del ciclo while, viene moltiplicato per 3. k cresce nel seguente modo: 1, 3, 9, 27, 81, …, cioè 3^0, 3^1, 3^2, 3^3,…

Alla x-esima iterazione k vale quindi 3^x. Noi vogliamo sapere il numero di iterazioni che porta k ad essere pari a N, cioè il valore di x per cui 3^x=N, cioè log N.

Il numero di iterazioni del **ciclo (3)** è quindi **log N**.

La complessità del frammento di codice è proporzionale al numero totale di iterazioni del ciclo 3, che contiene due istruzioni con costo costante. Quindi la complessità del frammento di codice in esempio è **Θ(N log N log N), ossia Θ(N log^2 N)**.

Esercizio 3

Calcolare la complessità del seguente frammento di pseudo-codice in funzione di N.

```
integer a ← 0, j ← 1, k ← 0, h ← 0
while j < N do
    k ← k + 1
    for integer i ← 0 to k do
        h ← 2
        while h < 2^N do
            a ← a + 1
            h ← h * h
    j ← j * 2
```

Soluzione

Indichiamo il primo ciclo con (1), il secondo con (2) e il terzo con (3). Il ciclo (2) dipende dal ciclo (1), poichè il numero di iterazioni del secondo ciclo dipende dal valore di k, che viene incrementato ad ogni iterazione del primo ciclo. Il ciclo (3) è indipendente dai cicli esterni.

Procediamo quindi ad analizzare per primo il **ciclo (3)**:

h cresce nel seguente modo: 2, 4, 16, 256, …, cioè 2^1, 2^2, 2^4, 2^8,…, cioè 2^(2^0), 2^(2^1), 2^(2^2), 2^(2^3),…

Alla x-esima iterazione h vale quindi 2^(2^x). Noi vogliamo sapere il numero di iterazioni che porta h ad essere pari a 2^N, cioè il valore di x per cui 2^(2^x)=2^N, cioè log N.

Il numero di iterazioni del **ciclo (3)** è quindi **log N**.

Analizziamo ora i **cicli (1)** e **(2)**

j cresce nel seguente modo: 1, 2, 4, 8, 16,…, cioè 2^0, 2^1, 2^2, 2^3, 2^4,…

Alla x-esima iterazione j vale quindi 2^x. Noi vogliamo sapere il numero di iterazioni che porta j ad essere pari a N, cioè il valore di x per cui 2^x=N, cioè log N.

Ad ogni iterazione k viene incrementato di 1, quindi k varia, prima del ciclo (2), da 1 a logN con passo 1.

Ad ogni iterazione del ciclo (1), il ciclo (2) viene eseguito k volte, cioè il numero totale di operazioni svolte dai cicli (1) e (2) è 1+2+3+4+5+…+log N. Si noti che il numero di addendi nella sommatoria è quindi log N, perché log N è il numero di iterazioni che svolge il ciclo esterno (ciclo (1)).

Il numero totale di iterazioni è quindi Somma[i=0…logN](i) che è Θ(log^2 N).

Il numero di iterazioni del ciclo **(2)** nel complesso è log^2 N.

Visti i risultati parziali la complessità totale è del frammento di codice è **Θ(log^3 N)**.

Esercizio 3
Si cosideri il seguente frammento di codice:

```
integer sum ← 0, j ← 0
for integer i ← 0 to log(N) do
    j ← 2
    while j < 2^N do
        (*)
        j ← 2 * j
```

dove (*) è

```
1. sum++
2. sum <- sum + f(N);
3. sum <- sum + f(i);
```

con `f` definita nel seguente modo:

```
Procedure integer f(integer n)
    if n > 1 then
        return n + f(n/2)
    else
        return 1
```

Calcolare la complessità del frammento di codice, in funzione di N, in tutti e tre i casi.

Soluzione

Caso 1

I cicli (1) e (2) non sono correlati quindi li possiamo considerare separatamente. Consideriamo quindi prima il **ciclo (1)**:

i cresce nel seguente modo: 1, 2, 3, 4, ...

Alla x-esima iterazione i vale quindi x. Noi vogliamo sapere il numero di iterazioni che porta i ad essere pari a log N, cioè log N.

Il numero di iterazioni del **ciclo (1)** è quindi **log N**.

Analizziamo ora il **ciclo (2)**:

j cresce nel seguente modo: 2, 4, 8, 16, …, cioè 2^1, 2^2, 2^3,…

Alla x-esima iterazione j vale quindi 2^x. Noi vogliamo sapere il numero di iterazioni che porta j ad essere pari a 2^N, cioè il valore di x per cui 2^x=2^N, cioè N.

Il numero di iterazioni del **ciclo (2)** è quindi **N**.

La complessità del frammento di codice è in questo caso **Θ(N log N)**.

Casi 2 e 3

Prima di analizzare la complessità del frammento di codice, calcoliamo la complessità temporale Tf(n) della procedura f.

f è una procedura ricorsiva: la complessità può essere espressa dalla seguente equazione alle ricorrenze:

$Tf(n) = Tf(n/2) + \Theta(k)$ se $n > 1$, $Tf(1) = \Theta(k)$.

Applicando il Master Theorem si verifica facilmente che $Tf(n)$ è $\Theta(\log(n))$.

Analizziamo ora la complessità del frammento di pseudo-codice.
I cicli (1) e (2) non sono correlati come nel caso precedente e anche la chiamata di funzione all'interno del ciclo (2) non dipende da i e j. Per cui basta calcolare la complessità della funzione in relazione al parametro di ingresso.

Caso 2
f viene chiamata all'interno del ciclo (2) con parametro N, e non dipende quindi dalle iterazioni effettuate dai cicli in cui è contenuta. La complessità del frammento di codice può essere scritta come il prodotto tra le iterazioni del primo ciclo per le iterazioni del secondo ciclo per la complessità di f(N), ossia:

$T(n) = \Theta(N \log N \, Tf(N)) = \Theta(N \log N \log N) = \Theta(N \log^2 N)$.

Caso 3
La chiamata di funzione dipende dal ciclo (1), infatti all'interno del ciclo (2) f viene chiamata con i come parametro (dove i è l'indice del ciclo (1)), mentre il ciclo (2) è indipendente.

Possiamo scrivere la complessità del frammento di codice come:

$T(n) = N \, Tf(1) + N \, Tf(2) + \ldots + N \, Tf(\log N) = N \sum_{i = 1 \ldots \log N} Tf(i) = N \sum_{i = 1 \ldots \log N} \log(i) =$
$= N \log \left(\prod_{i = 1 \ldots \log N} i \right) = N \, (\log(\log(N)!))$

La complessità del frammento di codice è quindi in questo caso $\Theta(N \log N \log \log N)$.