Prova Intermedia 14 Aprile 2015 - soluzioni

http://twiki.di.uniroma1.it/twiki/view/Intro_algo/
bonacina@di.uniroma1.it

Esercizio 1

1. **(Traccia A)** Definire la notazione "O" e discuterne la relazione con la notazione "Θ".

   (Traccia B) Definire la notazione "Θ" e discuterene il significato.

   (Traccia C) Definire la notazione "Ω" e discutere la relazione con la notazione "Θ".

   (Traccia D) Definire la notazione "O". Esiste una relazione tra la notazione "O" e la notazione "Ω"? Motivare la risposta.

2. Si risponda ad ognuna delle seguenti domande, motivando la risposta:

   (a) **(Traccia A/C)** Dimostrare o confutare la seguente affermazione: se $f(n) = O(g(n))$ e $t(n) = O(h(n))$, allora $f(n) \cdot t(n) = O(g(n) \cdot h(n))$.

   (b) **(Traccia A/C)** Se si dimostra che un algoritmo ha tempo di esecuzione $\Omega(n^2)$ nel caso peggiore, è possibile che esistano istanze su cui l'algoritmo termina in $\Theta(n)$ passi?

   (c) **(Traccia A/D)** Se si dimostra che un algoritmo ha tempo di esecuzione $O(n^2)$, possiamo dedurne che nel caso migliore è in $\Omega(n^2)$?

   (d) **(Traccia B/D)** Dimostrare o confutare la seguente affermazione: se $f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n))$, allora $f(n) = \Theta(h(n))$.

   (e) **(Traccia B/C)** Se un algoritmo ha tempo di esecuzione $\Theta(n)$ nel caso migliore, posso dedurne che nel caso peggiore avrà una complessità in $\Omega(n)$?

   (f) **(Traccia B/D)** Se si dimostra che un algoritmo ha tempo di esecuzione $O(n^2)$ nel caso migliore, è possibile che esistano istanze su cui l'algoritmo termina in $\Omega(n^3)$ passi?

Soluzione

1. $\boldsymbol{f = O(g)}$ significa che esiste una costante c ed un intero n_0 tale che per ogni $n \geqslant n_0$, $f(n) \leqslant cg(n)$.

   $\boldsymbol{f = \Omega(g)}$ significa che esiste una costante c ed un intero n_0 tale che per ogni $n \geqslant n_0$, $f(n) \geqslant cg(n)$.

   $\boldsymbol{f = \Theta(g)}$ significa che vale che $f(n) = \Omega(g(n))$ e pure che $f(n) = O(g(n))$.

   Riguardo le relazioni, per esempio $O(n) \cap \Omega(n^2) = \emptyset$ e più in generale se $h/g \overset{n\to\infty}{\longrightarrow} \infty$ allora $O(g) \cap \Omega(h) = \emptyset$ (perchè?).

2. (a) Sì è vero. Se $f = O(g)$ e $t = O(h)$ allora, per definizione, esistono due costanti c_f e c_t, e due interi n_f e n_t tali che se $n \geqslant n_f$ allora $f(n) \leqslant c_f g(n)$ e se $n \geqslant n_t$ allora $t(n) \leqslant c_t h(n)$. Pertanto, per $n \geqslant \max\{n_f, n_g\}$ vale che

$$f(n) \cdot t(n) \leqslant c_f g(n) \cdot c_t h(n).$$

Quindi $f \cdot t = O(g \cdot h)$.

(b) Si, per esempio il seguente:

```
1: function ESEMPIETTO(intero n)
2:     if n è pari then
3:         esegui n passi
4:     else
5:         esegui n^2 passi
6:     end if
7: end function
```

(c) No, per esempio l'algoritmo potrebbe avere tranquillamente sempre tempo di esecuzione $O(1)$. E quindi avere tempo di esecuzione $O(1)$ anche nel caso migliore, cosa incompatibile con la possibilità che nel caso migliore il tempo di esecuzione sia $\Omega(n^2)$ (poichè $O(1) \cap \Omega(n^2) = \emptyset$).

(d) Sì è vero. Se $f = O(g)$ e $g = O(h)$ allora, per definizione, esistono due costanti c_f e c_g, e due interi n_f e n_g tali che se $n \geqslant n_f$ allora $f(n) \leqslant c_f g(n)$ e se $n \geqslant n_g$ allora $g(n) \leqslant c_g h(n)$. Pertanto, per $n \geqslant \max\{n_f, n_g\}$ vale che

$$f(n) \leqslant c_f g(n) \leqslant c_f c_g h(n).$$

Quindi $f = O(h)$. Il risultato per gli "Ω" è del tutto analogo.

(e) Si, nel caso migliore ha tempo di esecuzione $\Theta(n)$ e quindi a maggior ragione $\Omega(n)$. Poichè quello è il caso migliore significa che in ogni altra istanza il tempo di esecuzione è peggiore (o uguale). Quindi per ogni altra istanza è pure $\Omega(n)$, tra cui (in particolare) nel caso peggiore.

(f) Si, per esempio il seguente:

```
1: function ESEMPIETTO(intero n)
2:     if n è pari then
3:         esegui n^2passi (o n passi o anche solo 1...)
4:     else
5:         esegui n^3 passi
6:     end if
7: end function
```

□

Esercizio 2

Si considerino le seguenti funzioni:

```
1: function FUN(array A, int i, int f)          ▷ (Traccia A)
2:     n = f − i + 1;
3:     t = 1;
4:     while n ⩾ 1 do
5:         n = n/2;
6:         t++;
```

```
7:      end while
8:      if f − i + 1 < 4 then return t
9:      else return i + FUN(A, i, (i+f)/2) ∗ FUN(A, (i+f)/2 + 1, f)
10:     end if
11: end function
```

```
1: function FUN(array A, int i, int f)                    ▷ (Traccia B)
2:      n = f − i + 1;
3:      t = 1;
4:      while n ⩾ 1 do
5:          for k = 1 to n do t++;
6:          end for
7:          n = n/2;
8:      end while
9:      if f − i + 1 < 4 then return t;
10:     else5 return i + 3 ∗ FUN(A, i, (i+f)/2);
11:     end if
12: end function
```

```
1: function FUN(array A, int i, int f)                    ▷ (Traccia C)
2:      n = f − i + 1;
3:      t = k = 1;
4:      while t ⩽ n do
5:          t = 3 ∗ t;
6:          k++;
7:      end while
8:      if f − i + 1 < 9 then return k;
9:      else return i + FUN(A, i, i + n/4) − FUN(A, i + n/4 + 1, i + n/2) + FUN(A, i + n/2 + 1, i + 3n/4) −
        FUN(A, i + 3n/4 + 1, f);
10:     end if
11: end function
```

```
1: function FUN(array A, int i, int f)                    ▷ (Traccia D)
2:      n = f − i + 1;
3:      t = 1;
4:      while n ⩾ 1 do
5:          for k = 1 to n do t++;
6:          end for
7:          n = n/2;
8:      end while
9:      if f − i + 1 < 4 then return t;
10:     else return i + 2 ∗ FUN(A, i, f − (f−i+1)/4);
11:     end if
12: end function
```

Quale è il tempo di esecuzione di FUN in funzione di n? Si imposti e si risolva la relazione di ricorrenza.

Soluzione

Prima scriviamo tutte le relazioni di ricorrenza e poi vediamo come risolverle.

(Traccia A) Il running time del ciclo while (righe 4-7) è $O(\log n)$ e poi ci sono due chiamate ricorsive su input di lunghezza $n/2$, quindi la relazione di ricorrenza per il running time

complessivo T è:

$$T(n) = \begin{cases} 2T(n/2) + \Theta(\log n) & \text{se } n \geqslant 4 \\ \Theta(1) & \text{altrimenti.} \end{cases}$$

Possiamo provare che $\Omega(n) \leqslant T(n) \leqslant O(n \log n)$. Vediamo prima, per induzione, che $T(n) \geqslant n$. Sia d una costante tale che per n abbastanza grande $\Theta(\log n)$ è almeno $d \log n$

$$T(n) \geqslant 2T(n/2) + d \log n \geqslant 2\frac{n}{2} + d \log n \geqslant n.$$

Vediamo ora che $T(n)$ è $O(n \log n)$:

$$T(n) \leqslant \Theta(1) + \sum_{j=0}^{\log n - 2} 2^j \log \frac{n}{2^j} = \Theta(1) + \log n \sum_{j=0}^{\log n - 2} 2^j - \sum_{j=0}^{\log n - 2} j 2^j \leqslant \Theta(1) + \log n \sum_{j=0}^{\log n - 2} 2^j$$
$$= O(n \log n).$$

(Traccia B) Sia $N = f - i + 1$ il valore di input iniziale. Alla j-esima iterazione del ciclo while (righe 4-7) n è $N/2^{j-1}$. Pertanto sempre alla j-esima iterazione del ciclo while il running time del ciclo for (righe 5-6) è $O(n) = O(N/2^{j-1})$. Quindi il running time complessivo del ciclo while è:

$$\sum_{j=1}^{\log N} O(N/2^{j-1}) = O(N).$$

Poi si ha una chiamata ricorsiva con input $N/2$. Pertanto la relazione di ricorrenza è:

$$T(N) = \begin{cases} T(N/2) + \Theta(N) & \text{se } N \geqslant 4 \\ \Theta(1) & \text{altrimenti.} \end{cases}$$

$T(N) = \Theta(N)$. Sicuramente vale che $T(N) = \Omega(N)$, quindi ci basta mostrare che $T(N) = O(N)$. Mostriamo quindi, per induzione, che per N abbastanza grande esiste una costante c tale che $T(N) \leqslant cN$:

$$T(N) \leqslant T(N/2) + dN \leqslant cN/2 + dN,$$

ci basta pertanto scegliere $c = 2d$ per avere che vale l'ipotesi induttiva.

(Traccia C) Il running time del ciclo while (righe 4-7) è $O(\log n)$ e poi ci sono 4 chiamate ricorsive su input di lunghezza $n/4$, quindi la relazione di ricorrenza per il running time complessivo T è:

$$T(n) = \begin{cases} 4T(n/4) + \Theta(\log n) & \text{se } n \geqslant 9 \\ \Theta(1) & \text{altrimenti.} \end{cases}$$

Vale che $\Omega(n) \leqslant T(n) \leqslant O(n \log n)$ (vedi i conti in **(Traccia A)**).

(Traccia D) Sia $N = f - i + 1$ il valore di input iniziale. Alla j-esima iterazione del ciclo while (righe 4-7) n è $N/2^{j-1}$. Pertanto sempre alla j-esima iterazione del ciclo while il running time del ciclo for (righe 5-6) è $O(n) = O(N/2^{j-1})$. Quindi il running time complessivo del ciclo while è:

$$\sum_{j=1}^{\log N} O(N/2^{j-1}) = O(N).$$

Poi si ha una chiamata ricorsiva con input $3N/4$. Pertanto la relazione di ricorrenza è:

$$T(N) = \begin{cases} T(3N/4) + \Theta(N) & \text{se } N \geqslant 4 \\ \Theta(1) & \text{altrimenti.} \end{cases}$$

$T(N) = \Theta(N)$ (vedi i conti in **(Traccia B)**). □

Esercizio 3

Sia A un array di n valori reali.

> **(Traccia A)** Dato un valore k, si progetti un algoritmo per determinare se in A ci sono due elementi la cui differenza è uguale a k.
>
> **(Traccia B)** Un valore $x \in A$ si dice *speciale* se x è maggiore della somma di tutti i valori $y \in A$ tali che $y < x$. Si progetti un algoritmo per determinare se in A c'è almeno un elemento speciale *diverso da* $\min A$[1].
>
> **(Traccia C)** Dato un array B di n valori reali e un valore k. Si progetti un algoritmo per determinare se esistono un elemento $a \in A$ e un elemento $b \in B$ tali che $a \cdot b = k$.
>
> **(Traccia D)** Un valore $x \in A$ si dice *singolare* se x è uguale al prodotto di tutti i valori $z \in A$ tali che $x > z$. Si progetti un algoritmo per determinare se in A c'è almeno un elemento singolare.

Si descriva l'idea algoritmica prima di passare allo pseudocodice, e si analizzino poi correttezza e tempo di esecuzione dell'algoritmo proposto. Il tempo di esecuzione dovrebbe essere $O(n \log n)$.

Soluzione

Prima di tutto, siccome sono richiesti algoritmi con tempo di esecuzione $O(n \log n)$ abbiamo tempo per ordinare A col MERGESORT. Quindi ogni soluzione che proporremo inizia con l'ordinare l'array A, diciamo in maniera crescente. Poi, una volta ordinato A, le idee che possiamo usare sono le seguenti:

(Traccia A) Usiamo un indice j per scorrere ordinatamente A. Per ogni j usiamo la ricerca binaria su $A[j+1 : n]$ per individuare se c'è o meno un indice $\ell > j$ tale che

$$A[\ell] - A[j] = k.$$

Tempo di esecuzione di queste scansioni aggiuntive $O(n \log n)$.

```
1: function RICERCADIFFERENZA(array A, intero n, intero k) ▷ restituisce vero (YES) se due elementi di indici distinti sono a differenza k
2:     if k < 0 then k = −k;                                ▷ possiamo assumere k positivo
3:     end if
4:     MERGESORT(A);                                        ▷ running time Θ(n log n)
5:     j ← 1;
6:     while j ⩽ n − 1 do
7:         i ← j + 1;                                       ▷ (Re-)Inizializzo i parametri per la ricerca binaria
8:         f ← n;
9:         while i ⩽ f do                                   ▷ cerco se esiste un ℓ t.c. A[ℓ] − A[j] = k
```

[1] Nel testo originale dell'esercizio la condizione "***diverso da* $\min A$**" non era resente. Il che lo rendeva fattibile in $O(n)$. Vale infatti che $\min A$ è sempre *speciale* (perchè? come si trova $\min A$ in $O(n)$?).

```
10:             p ← (i+f)/2;
11:             if A[p] − A[j] = k then return YES;                ▷ l'ho trovato, è p
12:             end if
13:             if A[p] − A[j] > k then
14:                 f = p − 1;                                     ▷ cerco nella metà inferiore di A
15:             else
16:                 i = p + 1;                                     ▷ cerco nella metà superiore di A
17:             end if
18:         end while
19:         j++;                                                   ▷ incremento il contatore per scorrere A
20:     end while
21:     return NO;                                                 ▷ per ogni 1 ⩽ ℓ, j ⩽ n, A[j] − A[ℓ] ≠ k
22: end function
```

Controlliamo che il running time delle righe 4-18 è $O(n \log n)$. Alla iterazione j-esima del ciclo while a riga 4, le righe 5-16 implementano una ricerca binaria su $A[j : n]$ quindi il loro running time è $c \log(n - j)$, per qualche costante c. Pertanto il running time delle righe 4-18 è:

$$\sum_{j=1}^{n-1} c \log(n-j) = c \sum_{j=2}^{n-1} \log(j) \leqslant c \sum_{j=1}^{n-1} \log(n) \leqslant cn \log n.$$

(Traccia B) C'è da prestare attenzione a situazioni, per esempio, di questo tipo: $A = [-1, 1, 1]$. In questo caso abbiamo che $A[3] > \sum_{j<3} A[j]$ ma $A[3]$ *non* è speciale. Usiamo un indice j per scorrere A e una variabile $S_j = \sum_{k \leqslant j} A[k]$.

Se $S_j < A[j+1]$ e $A[j+1] > A[j]$ abbiamo finito. Altrimenti calcoliamo $S_{j+1} = S_j + A[j+1]$ e incrementiamo il contatore j. Tempo di esecuzione di questa seconda scansione (dopo aver ordinato A): $O(n)$.

```
1: function RicercaSpeciale(array A, intero n)▷ restituisce vero (YES) se in A c'è almeno
   un elemento speciale
2:     MergeSort(A);                                              ▷ running time Θ(n log n)
3:     j ← 1;
4:     S ← A[1];                                                  ▷ S conterrà Σ_{k⩽j} A[k]
5:     while j ⩽ n − 1 do
6:         j++;                                                   ▷ incremento il contatore per scorrere A
7:         if A[j] > S and A[j] > A[j − 1] then return YES;        ▷ j è speciale
8:         end if
9:         S ← S + A[j];
10:    end while
11:    return NO;                                                 ▷ nessun elemento in A è speciale
12: end function
```

(Traccia C) Usiamo un indice j per scorrere B (N.B.: B non serve che sia ordinato...) e usiamo la ricerca binaria su A per trovare se esiste un indice ℓ tale che $A[\ell] \cdot B[j] = k$. Tempo di esecuzione per queste scansioni aggiuntive: $O(n \log n)$.

```
1: function RicercaProdotto(array A, array B, intero n, intero k)▷ restituisce vero (YES)
   se ci sono indici j, ℓ tali che A[ℓ] ∗ B[j] = k
2:     MergeSort(A);                                              ▷ running time Θ(n log n)
3:     j ← 1;
4:     while j ⩽ n do
5:         i ← 1;                                                 ▷ (Re)-Inizializzo i parametri per la ricerca binaria
6:         f ← n;
```

```
7:        while i ⩽ f do                          ▷ cerco se esiste un ℓ t.c. B[j] * A[ℓ] = k
8:            p ← (i+f)/2;
9:            if B[j] * A[p] = k then return YES;          ▷ l'ho trovato, è p
10:           end if
11:           if B[j] * A[p] > k then
12:               f = p − 1;                          ▷ cerco ℓ nella metà inferiore di A
13:           else
14:               i = p + 1;                          ▷ cerco ℓ nella metà superiore di A
15:           end if
16:       end while                                   ▷ running time righe 5-16: O(log n)
17:       j++;                                        ▷ incremento il contatore j per scorrere B
18:   end while                                       ▷ running time righe 4-18: O(n log n)
19:   return NO;                                      ▷ per ogni 1 ⩽ ℓ, j ⩽ n, B[j] * A[ℓ] ≠ k
20: end function
```

(Traccia D) C'è da prestare attenzione a situazioni, per esempio, di questo tipo: $A = [1, 1, 1]$ oppure $A = [-1, -1, -1]$ oppure $A = [-1, 0, 0]$. In questo caso abbiamo che $A[3] = \prod_{j<3} A[j]$ ma $A[3]$ *non* è singolare (perchè?). Usiamo un indice j per scorrere A e una variabile $P_j = \prod_{k \leqslant j} A[k]$. Se $P_j = A[j+1]$ e $A[j+1] > A[j]$ abbiamo finito. Altrimenti calcoliamo $P_{j+1} = P_j \cdot A[j+1]$ e incrementiamo il contatore j. Tempo di esecuzione di questa seconda scansione (dopo aver ordinato A): $O(n)$.

```
1: function RICERCASINGOLARE(array A, intero n)   ▷ restituisce vero (YES) se in A c'è
   almeno un elemento singolare
2:     if n ⩽ 1 then return NO;                   ▷ escludiamo casi banali
3:     end if
4:     MERGESORT(A);                              ▷ running time Θ(n log n)
5:     j ← 1;
6:     P ← A[1];                                  ▷ P conterrà ∏_{k⩽j} A[k]
7:     while j ⩽ n − 1 do
8:         j++;                                   ▷ incremento il contatore per scorrere A
9:         if A[j] = P and A[j] > A[j − 1] then return YES;      ▷ j è singolare
10:        end if
11:        P ← P * A[j];
12:    end while
13:    return NO;                                 ▷ nessun elemento in A è singolare
14: end function
```

□