Introduzione agli Algoritmi

Esame Scritto a canali unificati con idee per la soluzione

docenti: T. Calamoneri, A. Monti
Sapienza Università di Roma
25 Ottobre 2022

Esercizio 1 (10 punti):
Si consideri la seguente funzione:

```
def es1(n):
    if n <= 1:  return 5
    a = es1(n//2)
    i = j = 1
    while j < n:
        j *= 2
        i += 1
    u, j = 1, n
    while j > 1:
        j -= i
        u += 1
    return a + es1(n//2) + u
```

a) Si imposti la relazione di ricorrenza che ne definisce il tempo di esecuzione giustificando dettagliatamente l'equazione ottenuta.

b) Qualora sia possibile, risolvere la ricorrenza utilizzando il **teorema principale** dettagliando il caso del teorema ed i passaggi logici. Se il teorema principale non è applicabile spiegarne il motivo.

a Il primo while del programma richiede tempo $\Theta(\log n)$ (ad ogni iterazione il valore di j raddoppia e si parte con $j = 1$ per arrivare a $j > n$). Il secondo while del programma richiede tempo $\Theta\left(\frac{n}{\log n}\right)$ (ad ogni iterazione al valore di j viene sottratto $\Theta(\log n)$ e si parte con $j = n$ per arrivare a $j \leq 1$). Vengono effettuate due chiamate alla funzione con input $\frac{n}{2}$. Per quanto detto si ottiene la ricorrenza: $T(1) = \Theta(1)$ e $T(n) = 2T\left(\frac{n}{2}\right) + \Theta\left(\frac{n}{\log n}\right)$.

b Il teorema principale non può essere applicato per risolvere l'equazione di ricorrenza. Infatti si ha $n^{log_b a} = n$ e $f(n) = \frac{n}{\log n}$. Vale dunque che $f(n)$ è più piccolo di $n^{log_b a}$ ma non possiamo applicare il primo caso perché $f(n)$ non è polinomialmente più piccolo di $n^{log_b a}$.

Esercizio 2 (10 punti):
Sia A un array di n elementi con n pari, contenente uno stesso numero di interi pari ed interi dispari. Un *riarrangiamento* degli interi in A è *valido* se nelle posizioni ad indice pari compaiono interi pari e in quelle ad indice dispari interi dispari (l'indice 0 è considerato pari).

Ad esempio, per $A = [7, 3, 1, 8, 8, 2, 1, 4]$ esistono diversi riarrangiamenti validi come: $[8, 7, 2, 3, 8, 1, 4, 1]$, oppure $[4, 1, 2, 1, 8, 3, 8, 7]$ o anche $[2, 3, 8, 1, 8, 7, 4, 1]$.

Progettare un algoritmo che, preso l'array A, produca un riarrangiamento valido.
L'algoritmo deve avere costo computazionale $O(n)$ e deve utilizzare uno spazio di lavoro costante (in altri termini, non è possibile utilizzare liste concatenate o array di appoggio).
Dell'algoritmo proposto:

a) si dia la descrizione a parole,

b) si scriva lo pseudocodice,

c) si giustifichi il costo computazionale.

a) L'algoritmo modifica l'idea della funzione `Partiziona` di QuickSort di usare due indici, ma invece di farli partire dagli estremi dell'array, li fa muovere sugli elementi di indice pari e dispari, rispettivamente. Più in dettaglio, usiamo p per scorrere le posizioni pari dell'array e

d per scorrere le posizioni dispari dell'array. All'inizio l' indice p 'e sulla posizione pari più a sinistra ($p = 0$) mentre l'indice d è sulla posizione dispari più a sinistra ($d = 1$). La posizione pari successiva sarà in $p + 2$ mentre la posizione dispari successiva sarà in $d + 2$. Scorriamo le posizioni pari e dispari dell'array iterando la seguente regola:

- **se il valore della posizione pari è pari: allora si passa ad esaminare la posizione pari successiva ($p+ = 2$)**
- **altrimenti, se il valore della posizione dispari è dispari: si passa a considerare la posizione dispari successiva ($d+ = 2$)**
- **altrimenti: i valori delle attuali posizioni pari e dispari vanno scambiati, si scambiano i valori contenuti in $A[p]$ e $A[d]$, e si passa ad esaminare le posizioni pari e dispari successive.**

b) di seguito una possibile implementazione dell'algoritmo in Python:

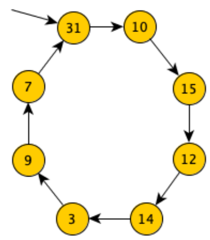
```python
def es2(A):
    p,d = 0,1
    while p<len(A) and d<len(A):
        if A[p]%2==0:
            p+=2
        elif A[d]%2==1:
            d+=2
        else:
            A[p],A[d]=A[d],A[p]
            p+=2
            d+=2
    return A
```

c) Nota che ad ogni iterazione del `while` almeno uno dei due indici viene incrementato di 2. L'indice p e l'indice d possono essere incrementati al più $n/2$ volte, questo significa che il numero di iterazioni del `while` è complessivamente limitato da n. Il costo dell'algoritmo è dunque $\Theta(n)$.

Esercizio 3 (10 punti):

Si consideri una lista a puntatori circolare L data tramite un puntatore p ad un suo elemento. In L ogni nodo ha 2 campi: il campo `key` contenente un intero ed il campo `next` con il puntatore al nodo seguente.

Sappiamo che gli interi dei vari nodi sono tutti distinti e bisogna trovare il valore minimo tra questi. Ad esempio per la lista circolare di seguito il valore cercato è 3:



Progettare un algoritmo **iterativo** che, dato il puntatore p ad un nodo della lista circolare, restituisce il valore cercato in tempo $\Theta(n)$ dove n è il numero di nodi della lista.
Dell'algoritmo proposto:

a) si dia la descrizione a parole,

b) si scriva lo pseudocodice,

c) si giustifichi il costo computazionale.

a) Grazie al campo `next`, si scorre la lista circolare fino ad incontrare di nuovo il nodo da cui si è partiti (quando il campo `key` contiene l'intero del nodo da cui si è partiti). Nello scorrere la lista, si tiene traccia del valore minimo x incontrato fino a quel momento. Alla fine si restituisce il valore x.

b) di seguito una possibile implementazione dell'algoritmo in Python:

```
def es3(p):
    a = x = p.key
    p = p.next
    while p.key != a:
        x = min(p.key, x)
        p = p.next
    return x
```

c) Il while richiede di scorrere l'intera lista e costa dunque $\Theta(n)$.