Introduzione agli Algoritmi
Esame Scritto a canali unificati con spunti per la soluzione

docenti: T. Calamoneri, A. Monti
Sapienza Università di Roma
20 Marzo 2023

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

```
def Es1(n):
    if n < 10:
        return n
    s = Es1(n//2) * Es1(n//2)
    for j in range(n):
        k = n
        while k > 1:
            s = s + 1
            k = k//2
    return s + Es1(n//2)
```

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 caso base si ha quando $n < 10$, ed ha costo costante.**
La funzione ricorsiva su input $n \geq 10$ richiama se stessa 3 volte su un'istanza di dimensione $n/2$. Il ciclo for viene eseguito $\Theta(n)$ volte mentre il ciclo $while$ al suo interno $\Theta(\log n)$ volte (in quanto si parte

con $k = n$ e ad ogni iterazione del while il valore di k si dimezza). Perciò, la relazione di ricorrenza da risolvere è:

$$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n \log n).$$

b) **Per applicare il metodo principale notiamo che $n^{\log_b a} = n^{\log_2 3} = n^{1+\epsilon}$ mentre $f(n) = \Theta(n \log n)$ che è dominato da $n^{1+\epsilon}$ siamo dunque nel primo caso del teorema principale e si ha $T(n) = \Theta(n^{\log_2 3})$. È necessario dettagliare tutti i passaggi per arrivare all'equazione di ricorrenza e per risolverla.**

Esercizio 2 (10 punti):

Dato un array A di n interi non negativi distinti, si vuole determinare se esistono almeno tre numeri consecutivi di valore inferiore a 100.
Ad esempio, se $A = [101, 5, 9, 31, 33, 10, 100, 4, 8, 32, 500, 11, 99]$, gli elementi 8, 9 e 10 così come gli elementi 31, 32 e 33 rispettano la proprietà mentre 99, 100 e 101 no.
Progettare un algoritmo che, dato A, in tempo $\Theta(n)$ restituisce il valore dell'elemento centrale della terna se questa è presente, -1 altrimenti. Se esistono più terne allora bisogna restituire l'elemento centrale di valore massimo (nell'esempio sopra, l'algoritmo dovrebbe restituire 32; se nell'array ci fossero 4 numeri consecutivi andrebbe restituito il terzo, ad esempio se l'array contenesse soltanto 5, 6, 7 e 8, andrebbe restituito il 7).
Dell'algoritmo proposto:

a) si scriva lo pseudocodice opportunamente commentato;

b) si giustifichi il costo computazionale.

È possibile procedere in molti modi, ma tutti si basano sul fatto che la soluzione va cercata tra i valori compresi tra 0 e 99.
Uno di questi consiste nell'utilizzare una variante dell'algoritmo di ordinamento per conteggio, con un array di lavoro C di dimensione 100 dove in posizione $C[i]$ viene inserito il numero di occorrenze in A dell'intero i, *solo se* $i < 100$. Con una singola scansione dell'array A si può riempire l'array C, e con al più una singola scansione dell'array C si può scoprire se è presente almeno una terna. Si noti che l'elemento centrale di una terna è un intero i tra 1 e 98 tale che $C[i-1]$, $C[i]$ e $C[i+1]$ hanno valore

diverso da zero e che l'elemento cercato è il primo elemento centrale di una terna scorrendo l'array C da destra, mentre se si scorre C da sinistra bisognerà tenere traccia dell'elemento centrale maggiore via via trovato. Al termine viene restituito -1 se nessuna terna è stata individuata o il valore centrale dell'ultima terna trovata.

Un'altra possibilità consiste nell'applicare una sorta di algoritmo di partizione che antepone tutti gli elementi ≤ 100 agli altri; si può poi procedere all'ordinamento di questi elementi in tempo costante (poiché sono tutti distinti non saranno più di 100) ed alla ricerca della prima terna che si incontra partendo da destra.

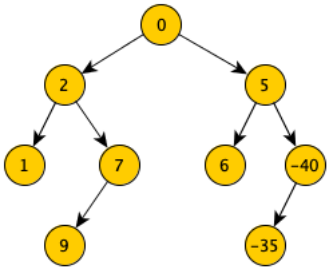
Si osservi che l'algoritmo naïf che consiste nello scorrere l'array per ciascun elemento alla ricerca dei due elementi che possano far parte della terna ha un tempo quadratico, mentre ordinare l'array e poi scorrerlo una sola volta alla ricerca di una eventuale terna richiederebbe tempo $\Omega(n \log n)$ (il costo esatto dipende dall'algoritmo di ordinamento usato).

Segue un possibile codice Python della prima idea proposta.

```
def terna(A):
    C = [0] * 100
    for i in range (len(A)):
        if A[i] < 100:   C[A[i]] += 1
    t = -1
    for i in range(1, 99):
        if C[i-1] > 0 and C[i] > 0 and C[i+1] > 0:   t = i
    return t
```

Il costo è ottenuto sommando il contributo del primo for (che è $\Theta(n)$ a quello del secondo for (che è $O(1)$). Ne segue un costo complessivo di $\Theta(n)$.

Esercizio 3 (10 punti): Un nodo di un albero a valori interi si dice *nodo valido* se la somma dei valori dei suoi antenati è uguale al valore del nodo. Ad esempio nell'albero binario in figura, risultano validi 3 nodi (quello con valore 0, quello con valore 9 e quello con valore -35).
Dato il puntatore r al nodo radice di un albero binario non vuoto, progettare un algoritmo *ricorsivo* che in tempo $\Theta(n)$ calcoli il numero di nodi validi dell'albero.
L'albero è memorizzato tramite puntatori e record a tre campi: il campo key contenente il valore ed i campi $left$ e $right$ con i puntatori al figlio sinistro e al figlio destro, rispettivamente (questi puntatori valgono $None$ in mancanza del figlio).



Dell'algoritmo proposto:

a) si scriva lo pseudocodice opportunamente commentato;

b) si giustifichi il costo computazionale.

NOTA BENE: nello pseudocodice dell'algoritmo ricorsivo **non** si deve far uso di variabili globali.

Si utilizzi una visita in pre-ordine dell'albero. Ogni nodo riceve la somma dei valori dei suoi antenati, (in questo modo è in grado di determinare se è valido o meno) e restituisce la somma dei nodi validi presenti nel suo sottoalbero.

Ecco un possibile codice Python dell'algoritmo che non utilizza variabili globali:

```
def es3(r, tot = 0):
    a = 0
    if r.key == tot:
        a+ = 1
    if r.left! = None:
        a+ = es3(r.left, tot + r.key)
    if r.right! = None:
        a+ = es3(r.right, tot + r.key)
  return a
```

Il costo computazionale è quello della visita di un albero con n nodi. L'equazione di ricorrenza relativa alla visita è:

- $T(n) = T(k) + T(n-1-k) + \Theta(1)$
- $T(0) = \Theta(1)$

dove k, è il numero di nodi presenti nel sottoalbero sinistro, $0 \leq k < n$. L'equazione si può risolvere con il metodo di sostituzione (che va

esplicitamente svolto) dando come soluzione $\Theta(n)$.
Di conseguenza il costo dell'algoritmo è, come richiesto, $\Theta(n)$.
Nel compito, ogni passaggio va dettagliato.