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

docenti: T. Calamoneri, A. Monti
Sapienza Università di Roma
31 Gennaio 2023

Esercizio 1 (10 punti):

Si consideri il seguente algoritmo ricorsivo che, dato un array A di dimensione n, verifica se esistono due indici diversi i e j compresi nell'intervallo $[0, n-1]$ tali che $A[i] = j$ e $A[j] = i$:

```
def IndiciValori(A, sx, dx):
    if (sx >= dx):  return False
    else:
        trovato = False
        centro = (sx + dx)//2
        for i in range(sx, centro + 1):
            for j in range(centro + 1, dx + 1):
                if (A[i] == j) and (A[j] == i):   trovato = True
    trovato1 = IndiciValori(A, sx, centro)
    trovato2 = IndiciValori(A, centro + 1, dx)
    return trovato or trovato1 or trovato2
```

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

b) Si risolva la ricorrenza usando **due metodi** a scelta, dettagliando i passaggi del calcolo e giustificando ogni affermazione.

La relazione di ricorrenza che descrive il costo dell'algoritmo è $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n^2)$ con $T(1) = \Theta(1)$. È necessario dettagliare come è stata ottenuta.
Applicando il teorema principale si ottiene $T(n) = \Theta(n^2)$. La soluzione con questo ed un altro metodo va dettagliata nel compito.

Esercizio 2 (10 punti): Scrivere un algoritmo `ElementoPiuFrequente` che, dato un array A di n interi, compresi tra 1 e $10n$ restiuisce il valore più presente all'interno dell'array, a parità di occorrenze va restituito il valore minimo.

Ad esempio, se $A = [2, 6, 8, 5, 2, 3, 6, 8, 9, 5, 8, 1, 2]$, allora la risposta è 2 in quanto 2 ed 8 sono gli unici valori che compaiono 3 volte all'interno dell'array, mentre gli altri valori compaiono al più 2 volte.

Il costo computazionale dell'algoritmo proposto deve essere $\Theta(n)$.

Dell'algoritmo proposto:

a) si scriva lo pseudocodice opportunamente commentato,

b) si giustifichi il costo computazionale.

Per ottenere un algoritmo efficiente, la cosa più semplice è utilizzare un algoritmo di ordinamento. Una volta che l'array è ordinato, basta scorrere l'array una volta alla ricerca del valore con il maggior numero di occorrenze. Importante notare che il range dei valori è lineare in n e quindi si può usare il Counting Sort, che garantisce costo lineare.

Ecco di seguito una possibile implementazione dell'algoritmo.

```
def ElementoPiuFrequente( A ):
  CountingSort(A)
  risultato=A[0]
  contatoreRis=1
  contatore=1
  for i in range(1,len(A))
     if A[i]==A[i-1] contatore+=1
     else:
          if contatore>contatoreRis:
               contatoreRis = contatore
               risultato=A[i-1]
          contatore=1
  if contatore>contatoreRis:  return A[len(A)-1]
  return risultato
```

Alternativamente, si può usare la prima parte dell'algoritmo di Counting Sort che, tramite un array ausiliario di dimensione $10n$, conta le occorrenze; scorrendo una sola volta questo array, si determina immediatamente il valore corrispondente al numero massimo di occorrenze.

Il costo è, ovviamente, lineare. Per ottenere il punteggio completo, è necessario dimostrarlo, cioè dettagliare i passaggi del calcolo.

Si osservi che l'utilizzo di un dizionario NON è consigliato per garantire il costo lineare, visto che -come studiato- le operazioni su di esso non sono costanti nel caso peggiore ma solo nel caso medio.

Esercizio 3 (10 punti): Sia L una lista concatenata semplicemente puntata data tramite il puntatore p alla sua testa e contenenti chiavi intere positive. Ogni record è composto da due campi: il campo *key* che contiene il valore del nodo ed il campo *next* che contiene il puntatore al nodo successivo della lista se questo esiste, il valore None altrimenti. Si progetti un **algoritmo ricorsivo** con costo computazionale $O(n)$ che restituisca un puntatore al primo elemento della lista la cui chiave sia esattamente uguale alla somma delle chiavi di tutti gli elementi precedenti; se un tale elemento non esiste, verrà ritornato None.

Ad esempio, per la lista $p \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 6$ verrà restituito un puntatore al record contenente l'informazione 3; si noti che anche il record contenente l'informazione 6 soddisfa la richiesta di avere la chiave pari alla somma dei precedenti, ma il record contenente 3 lo precede.
Dell'algoritmo proposto:

a) si scriva lo pseudocodice opportunamente commentato,

b) si giustifichi il costo computazionale trovando e risolvendo l'equazione di ricorrenza.

Possiamo progettare un algoritmo ricorsivo cui vengono passati due parametri: il puntatore ad un nodo di L e la somma `sum` dei contenuti dei nodi di L precedenti. La funzione viene invocata la prima volta sulla testa della lista e con `sum` pari a zero. Il passo base si ha quando `p==None`, ed in tal caso vuol dire che l'elemento non è stato trovato per cui viene restituito None. Nel caso generale, se `p.key==sum` allora vuol dire che è stato trovato l'elemento cercato e lo si restituisce, in caso contrario viene richiamata la funzione sul nodo successivo di L passando come nuova somma il valore di quella precedente a cui si aggiunge il valore del record corrente.
Un possibile algoritmo è il seguente, con chiamata `m=Somma(p,0)`:

```
def Somma(p,sum ):
  if p==None:
      return None
  if sum==p.key:
      return p
  return Somma(p.next, sum+p.key)
```

Si vede facilmente che il caso peggiore si verifica quando non si trova l'elemento cercato e bisogna scorrere l'intera lista; se questa contiene n elementi, il costo computazionale è quindi ovviamente $O(n)$. Per dimostrarlo formalmente bisogna determinare l'equazione di ricorrenza che è:

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

che dà come soluzione $\Theta(n)$.
Anche qui, è necessario dettagliare come si è ottenuta l'equazione e la sua risoluzione.