Esercizi

Esercizi di anali di frammenti di codice
Grado di popolarità
Calcolo massimo e minimo
Array come insiemi

Si consideri il seguente frammento di codice:

```
count = 0
n = A.length
ArraySort(A)
for  i = 1 to  n  do
    if (binarySearch(A, A[1]+A[i])) then count = count+1
```

Si analizzi asintoticamente il tempo di esecuzione del frammento di codice sia nell'ipotesi che ArraySort sia il SelectionSort, il bubbleSort, l'insertionSort, l'heapSort o il quicksort specificando il caso peggiore e il caso migliore.

Si consideri il seguente frammento di codice:

```
count = 0
n = A.length
ArraySort(A)
for  i = 1 to  n  do
    if (binarySearch(A, A[1]+A[i])) then count = count+1
```

Si analizzi asintoticamente il tempo di esecuzione del frammento di codice sia nell'ipotesi che ArraySort sia il SelectionSort, il bubbleSort, l'insertionSort, l'heapSort o il quicksort specificando il caso peggiore e il caso migliore.

Tempo di esecuzione di SelectionSort in tutti i casi in $\Theta(n^2)$, di insertionSort e BubbleSort caso peggiore in $\Theta(n^2)$, caso migliore in $\Theta(n)$, dell' heapSort in nel caso peggiore in $\Theta(n \lg n)$ e caso migliore in $\Theta(n)$. Il quickSort nel caso peggiore ha tempo di esecuzione in $O(n^2)$ e in $\Theta(n \lg n)$ nel caso migliore. Poiché binarySearch(A,A[1] + A[i]) può essere eseguita in tempo costante nel caso in cui tutti gli elementi di A siano uguali a 0, con insertionSort e BubbleSort il caso migliore è in $\Theta(n)$, con SelectionSort è in $\Theta(n^2)$ e con l' heapSort è in $\Theta(n)$, con il quikSort è in $\Theta(n \lg n)$.
Nel caso peggiore le n chiamate della binarySearch si eseguono in $\Theta(n \lg n)$, quindi l'intero frammento si esegue in $\Theta(n^2)$ nel caso di insertionSort, BubbleSort, SelectionSort e QuickSort visto che $\Theta(n \lg n) + \Theta(n^2) = \Theta(n^2)$, in $\Theta(n \lg n)$ nel caso dell'heapSort.

Si consideri il seguente frammento di codice e se ne analizzi il tempo di esecuzione:

```
for(i = 1 to n)
{
   for (j = 1 to 3)
    {
      2 operationi eseguite in tempo costante
      for (k = 1 to i)
          3 operationi eseguite in tempo costante
    }
}
```

Si consideri il seguente frammento di codice e se ne analizzi il tempo di esecuzione:

```
for(i = 1 to n)                          ——————————————————  n volte
{
   for (j = 1 to 3)                      ——————————————————  3 volte
    {
      2 operationi
      for (k = 1 to i)                   ——————————————————  i volte
          3 operationi
    }
}
```

Il ciclo for più interno è eseguito i volte e si trova all'interno di un ciclo for che viene eseguito 3 volte, quindi quello interno è eseguito 3i volte, i varia da 1 a n, quindi si deve considerare la somma di 3i Θ(1) per i =0 fino a n, e questa somma è in $\Theta(n^2)$.

Si consideri il seguente frammento di codice e se ne analizzi il tempo di esecuzione:

```
void function(int n)
{
    int count = 0;
    for (int i=n/2; i<=n; i++)
        for (int j=1; j<=n; j = 2 * j)
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

Si consideri il seguente frammento di codice e se ne analizzi il tempo di esecuzione:

```
void function(int n)
{
    int count = 0;
    for (int i=n/2; i<=n; i++)                    ——————————   n/2 volte
        for (int j=1; j<=n; j = 2 * j)            ——————————   lg n volte
            for (int k=1; k<=n; k = k * 2)        ——————————   lg n volte
                count++;
}
```

I cicli for più interni sono entrambi eseguiti lg n volte, quello più esterno n/2 volte, quindi in totale ho un tempo di esecuzione in $\Theta(n \lg^2 n)$.

Si consideri il seguente frammento di codice e se ne analizzi il tempo di esecuzione:

```
void function(int n)
{
    int count = 0;
      for (int i=0; i<n; i++)
            for (int j=0; j < i; j++)
          {  for (int k=1; k<n; k=k*2)
                printf("*");
           }
}
```

Si consideri il seguente frammento di codice e se ne analizzi il tempo di esecuzione:

```
void function(int n)
{
    int count = 0;
      for (int i=0; i<n; i++)                          ——————————      n   volte
            for (int j=0; j < i; j++)                  ——————————      i volte
                 {  for (int k=0; k<n; k=k*2)          ——————————      lg n volte
                       printf("*");
                }
}
```

I ciclo for più interno è eseguito lg n volte, quello più esterno n volte, quello intermedio i volte, quindi in totale bisogna considerare la somma $\sum_{i=0,\ldots,n} i \lg n = \lg n \sum_{i=0,\ldots,n} i = \lg n\,(n(n+1)/2) = \Theta(n^2 \lg n)$

grado di popolarità

Sia A[1;n] una sequenza contenente il grado di popolarità degli n amici di una persona P, ciascuno individuato dall'indice nell'array.
Definiamo il grado di popolarità di una persona P come il più grande numero h tale che P ha almeno h amici ciascuno con grado di popolarità almeno h.
L'array A quindi contiene n interi non-negativi.
Si scriva un algoritmo che calcoli il grado di popolarità di P e se ne analizzi la complessità.

Quindi in output vogliamo
h = max{j | P ha almeno j amici di popolarità almeno j, per 0≤j≤n}.
Si analizzi la complessità dell'algoritmo proposto.
Esempio:

A=

1	0	5	8	10	2	12	6	0
1	2	3	4	5	6	7	8	9

L'amico 1 ha grado di popolarità 1, il 2 0, il 3 5 e così via.

Esempio grado di popolarità

Quindi in output vogliamo
h = max{j | P ha almeno j amici di popolarità almeno j, per 0≤j≤n}.
Esempio:

A=

1	0	5	8	10	2	12	6	0
1	2	3	4	5	6	7	8	9

P ha 7 amici con grado di popolarità almeno 1
P ha 6 amici con grado di popolarità almeno 2
P ha 5 amici con grado di popolarità almeno 5
P ha 4 amici con grado di popolarità almeno 6
P ha 3 amici con grado di popolarità almeno 8
P ha 2 amici con grado di popolarità almeno 10
P ha un solo amico con grado di popolarità 12

Quindi la risposta è 5 che è il massimo valore di j per cui ci sono almeno j amici con grado di popolarità almeno j.

grado di popolarità: sol in O(n lg n)

Ordiniamo A in ordine decrescente e esaminiamo l'array da sinistra verso destra fintanto che A[i] ≥ i, incrementando i e dando in output il valore di i raggiunto. Infatti in A[i] ho il grado di popolarità e l'indice mi dice quanti elementi sto considerando.

A=

1	0	5	8	10	2	12	6	0
1	2	3	4	5	6	7	8	9

A=

12	10	8	6	5	2	2	0	0
1	2	3	4	5	6	7	8	9

Nell'esempio A[i] ≥ i per i =1,…,5, e A[6] = 2 per cui A[6] < 6 quindi la risposta è 5.
Infatti ogni entrata A[i] contiene il grado di popolarità di un amico, e P ha 5 amici il cui grado di popolarità è maggiore o uguale a 5. Mentre solo 4 con grado di popolarità 6.

Grado di popolarità: sol in O(n) e spazio in O(n)

Si può pensare di costruire, in analogia al counting sort, un array delle frequenze dei gradi di popolarità. Tale array però non dovrebbe avere più di n+1 elementi. Se ci fossero dei gradi di popolarità molto grandi, potremmo comunque ridurli a n e così l'array delle frequenze dei gradi di popolarità sarebbe di n+1 elementi in ogni caso.

A=

1	0	5	80	70	2	65	6	0
1	2	3	4	5	6	7	8	9

C=

2	1	1	0	0	1	1	0	0	3
0	1	2	3	4	5	6	7	8	9

In C[i] quindi ho il numero di amici con grado di popolarità i, se i ≤ n-1 e in C[n] il numero di amici con grado di popolarità ≥ n.

grado di popolarità: sol2

C=

2	1	1	0	0	1	1	0	0	3
0	1	2	3	4	5	6	7	8	9

Poiché a noi interessa determinare il numero di amici di P che ha almeno grado di popolarità pari a quel numero, costruiamo l'array con in C[i] il numero degli amici con grado di popolarità maggiore o uguale a i. Questo si ottiene sommando ogni entrata di C alla successiva partendo da destra:

C=

9	7	6	5	5	5	4	3	3	3
0	1	2	3	4	5	6	7	8	9

Per esempio in C[8] si ha il numero degli amici di P con grado di popolarità almeno 8, e in C[5] si ha quindi il numero degli amici di P con grado di popolarità almeno 5 …

grado di popolarità: sol2

C=

9	7	6	5	5	5	4	3	3	3
0	1	2	3	4	5	6	7	8	9

Ora basta scorrere C da destra fintanto che C[i] < i.
Il primo i per cui C[i] ≥ i mi dà l'output voluto i.
Infatti
i = max{j | P ha almeno j amici di popolarità almeno j, per 0≤j≤n}.

Quindi nel caso dell'esempio la risposta è 5.

Analisi

La costruzione dell'array delle frequenze C è in O(n),
e il suo aggiornamento é in O(n)
Infine la ricerca del risultato in C è ancora in O(n)
Quindi l'algoritmo è in O(n).

Calcolo minimo e massimo

Dato un array di n elementi se calcoliamo prima il massimo e poi il minimo otteniamo un algoritmo che esegue 2n-2 confronti.
Possiamo migliorare questo limite calcolando contemporaneamente il massimo e il minimo.

Calcolo minimo e massimo

Caso 1: il numero degli elementi è pari

A=

8	7	2	5	6	4	3	4
0	1	2	3	4	5	6	7

calcoliamo massimo e minimo sulle coppie:

max=8 min =7

max=5 min =2

max=6 min =4

max=8 min =7

calcoliamo massimo dei massimi e il minimo dei minimi.
Quanti confronti?
n/2 per il calcolo dei massimi e minimi sulle coppie,
n/2-1 per il calcolo dei massimi tra i massimi e ancora
n/2-1 per il calcolo dei minimi tra i minimi,
per un totale di 3/2n-2 confronti

Calcolo minimo e massimo

Caso 2: il numero degli elementi è dispari

calcoliamo massimo e minimo sulle prime coppie:

A=

8	7	2	5	6	4	3
0	1	2	3	4	5	6

max=8
min =7

max=5
min =2

max=6
min =4

Procediamo come prima trascurando l'ultimo elemento e infine confrontiamo il massimo e il minimo trovato sui primi n-1 elementi con l'ultimo.
Quanti confronti?
(n-1)/2 per il calcolo dei massimi e minimi sulle coppie,
(n-1)/2-1 per il calcolo dei massimi tra i massimi e ancora
(n-1)/2-1 per il calcolo dei minimi tra i minimi,
più 2 finali
per un totale di 3(n-1)/2 - 2 +2= 3(n-1)/2 confronti

Conclusione minimo e massimo

Se n è pari il numero dei confronti è 3/2n-2
se n è dispari il numero dei confronti è 3(n-1)/2

Quindi se n = 2m si ha 3/2*2m-2= 3m-2
se n= 2m+1 si ha 3(2m+1-1)/2 = 3m
m = ⌊n/2⌋

Quindi possiamo concludere che al più sono necessari 3 ⌊n/2⌋ confronti.

Calcolare con n-1 confronti prima il massimo e poi il minimo dà un numero complessivo di confronti pari a 2(n-1) ≥ 3 ⌊n/2⌋ per n > 2

Per esercizio si scriva l'algoritmo in pseudocodice

Array come insiemi

Siano U e V due array contenenti lo stesso numero n di elementi. Si progetti un algoritmo che determina se U e V individuanolo stesso insieme di elementi.
Esempi.
Gli array V=[1,2,1,1,2] ed U=[2,2,1,1,2] determinano l'insieme {1,2}, e quindi con questo input l'output dell'algoritmo deve essere true.

Gli array V=[1,1,2,2,3] ed U=[2,3,2,3,2] determinano rispettivamente gli insiemi {1,2,3} e {2,3}, e quindi con questo input l'output dell'algoritmo deve essere false.
L'algoritmo deve avere tempo di esecuzione O(n lg n) nel caso peggiore. Scrivere l'algoritmo ed analizzarne il tempo di esecuzione in modo dettagliato.

Array come insiemi: sol

Si ordinano U e V, poi si adatta la funzione di fusione di due array ordinati in un unico array ordinato in modo da verificare se gli elementi, a meno di ripetizioni, sono gli stessi nei due array.
Se si usa il mergesort o l'heapsort il tempo di esecuzione per l'ordinamento dei due array è Θ(n lg n) nel caso peggiore.
Il controllo finale prende tempo Θ(n).
Si scriva per esercizio la funzione di fusione modificata in modo da controllare se gli insiemi sono uguali.

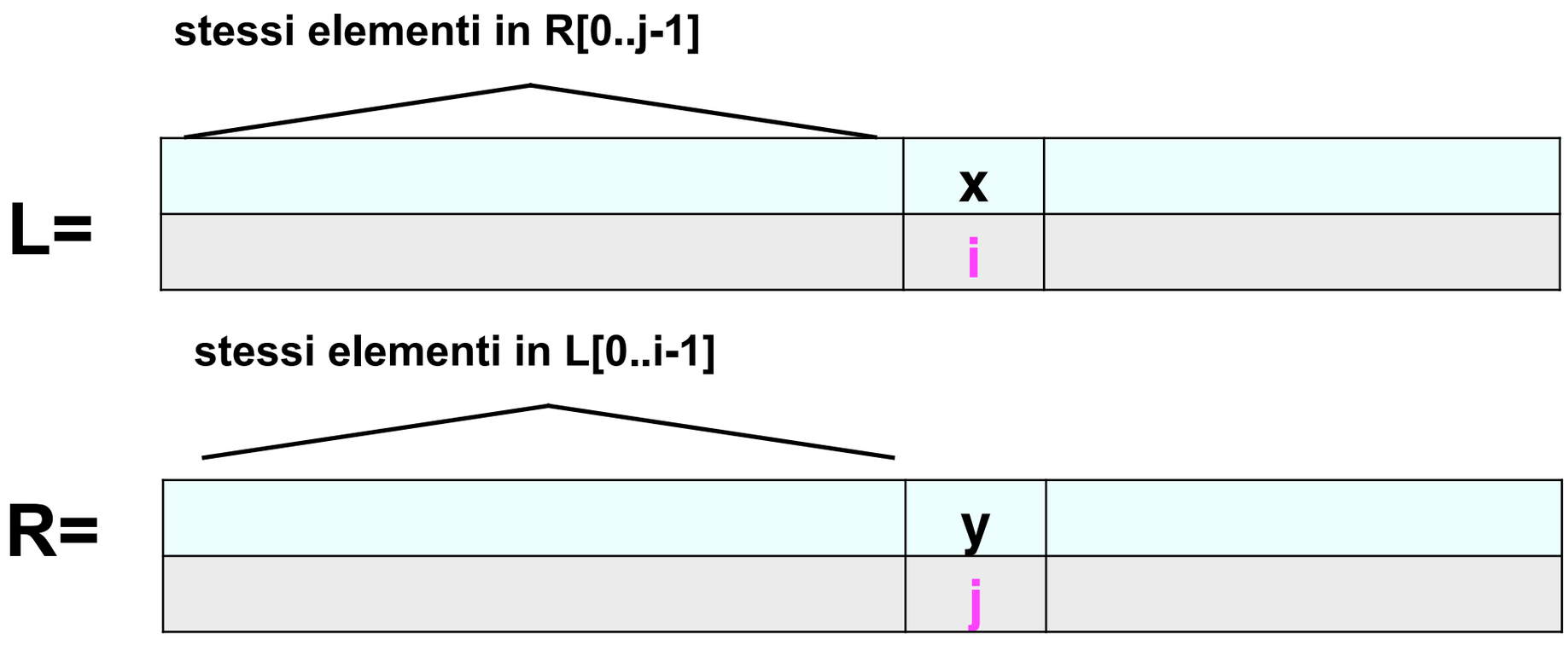
Lo pseudocodice

```
    insiemi(U,V)
input:  due array di interi
prec: U e V hanno lo stesso numero di elementi
output: 1 se i due array individuano lo stesso insieme e 0 altrimenti
HeapSort(U)
HeapSort(V)
return VerUguali(U,V)
```

La funzione Ver Uguali è così specificata:
VerUguali(L,R)

Input: L,R sono array di interi
prec: L e R hanno lo stesso numero di elementi e sono ordinati, cioè
L[1] ≤ ...≤ L[n-1] e R[1]≤…≤R[n-1]
postc: 1 se gli insiemi nei due array coincidono 0 altrimenti

Lo pseudocodice: progetto



Supponiamo che gli elementi in R[0..j-1] e in L[0..i-1] siano stati controllati e che siano uguali come insiemi, ci chiediamo cosa fare con x e y:

Se x = y bisogna incrementare entrambi gli indici per controllare gli elementi gli elementi successivi in entrambi gli array

Se x < y, controlliamo se x = R[j-1], se sì incrementiamo i altrimenti restituiamo 0, perchè, nel primo caso vuol dire che x è una occorrenza di un elemento già confrontato prima, nel secondo vuol dire che x non è presente in R, proprio perchè gli elementi sono gli stessi fino a quel momento e x dovrebbe essere presente a sinistra di y

L'altro caso è analogo.

Lo pseudocodice per verUguali

```
VerUguali(L,R)
Input: L,R  sono array n elementi
prec: L e R sono ordinati, cioè
L[1] ≤ ...≤ L[n-1] e R[1]≤…≤R[n-1]
postc: 1 se gli insiemi nei due array coincidono
 n =  L.size
  i=j=0
while  i < n and  j < n   do
        if (L[i] = R[j]) then i ++;j ++;
                         else
                    if L[i] < R[j]; then //deve essere uguale al precedente
                          if  L[i] = R[j-1] then i++ else return 0


                     else
                          if  R[j] = L[i-1] then j++ else return 0
while  i < n   do  //non è detto che terminino contemporaneamente
       if (L[i] = R[n-1]) then i ++; else return 0
while  j < n   do
      if (R[j] = L[n-1]) then j ++; else return 0
return 1
```