Esercizi 16 aprile 2020

1. Esercizi di analisi di algoritmi
2. Ordinare un array quasi ordinato
3. Contare le coppie a differenza k in un array

Prof.ssa E. Fachini

Esercizio di analisi algoritmi 1

```
Analizza(n)
input: un intero n
count = 0;
for i=1 to n do
        j = 1;
        while ( j+n/2 ≤ n ) do
                k=1;
                while (k ≤ n) do
                        count ++
                        k = 2*k;
                j++;
return count;
```

Soluzione

```
Analizza(n)
input: un intero n
count = 0;
for i=1 to n do
        j = 1;
        while ( j+n/2 ≤ n ) do
                k=1;
                while (k ≤ n) do
                        count ++
                        k = 2*k;
                j++;
return count;
```

Questo ciclo for è eseguito n volte.

Questo ciclo while è eseguito n/2 volte, perchè j varia da 1 a n, è incrementato di 1 a ogni passo ma quando è diventato n/2 si realizza la condizione di uscita dal ciclo.

Questo ciclo while è eseguito lg n volte, perchè k varia da 1 a n, ma viene raddoppiato ad ogni esecuzione.

Poiché i tre cicli sono annidati e le operazioni, diversi dai cicli, all'interno di essi sono eseguite in tempo costante possiamo concludere che Analizza è eseguita in tempo $\Theta(n^2/2 \lg n) = \Theta(n^2 \lg n)$

Esercizio di analisi algoritmi 2

```
Analizza(n)
input: un intero n
count = 0;
for i=1 to n do
      j = i;
      while(j≤i^2 )do
            if j è multiplo di i then
                  for k=1 to j-1 do
                        print(“*”)
            j++;
return;
```

Soluzione 2

```
Analizza(n)
input: un intero n
count = 0;
for i=1 to n do
      j = i;
      while(j≤i2 ) do
            if j è multiplo di i then
                  for k=1 to j-1 do
                        print(“*”)
            j++;
return;
```

Questo ciclo for è eseguito n volte.

Questo ciclo while è eseguito $i^2 - i$ volte, per i=1,…,n, perché j=i all'inizio e deve diventare i^2 perchè si esca dal ciclo

Questo ciclo for è eseguito j = pi volte, con p=1,…, i e i che varia da 1 a n

Quindi il ciclo for più interno è eseguito la prima volta i volte, poi 2i, poi 3i,…,fino a i*i, con i che va da 1 a n. Mettendo i in evidenza, questa somma si può scrivere $\Sigma_{i\in[1,n]}\, i(\Sigma_{j\in[1,i]}\, j) = \Sigma_{i\in[1,n]}\, i\,(i(i+1)/2) = \Sigma_{i\in[1,n]}\, i(i^2+i)/2) = 1/2(\Sigma_{i\in[1,n]}\, i^3 + \Sigma_{i\in[1,n]}\, i^2) = \Theta(n^4)$. Concludiamo che Analizza ha un tempo di esecuzione in $\Theta(n^4)$, perchè le esecuzioni del ciclo while, quando j non è un multiplo di i, sono eseguite in tempo costante e quindi costituiscono un termine assorbito dal conteggio sopra.

Esercizio di analisi algoritmi 3

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

Esercizio di analisi algoritmi 3

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

Questo ciclo for è eseguito n/2 volte.

Questo ciclo for è eseguito lg n volte.

Questo ciclo for è eseguito lg n volte.

Il ciclo for più interno è eseguito lg n volte circa, perché il parametro k parte da 1 e deve diventare la prima potenza di 2 maggiore o uguale a n per provocare l'uscita dal ciclo. Osservando che ogni intero è compreso tra due potenze di 2 consecutive si ha $2^h \leq n < 2^{h+1}$, è evidente che il ciclo viene eseguito h volte. Per la stessa ragione il ciclo immediatamente più esterno viene eseguito lg n volte e quindi i due annidati hanno un tempo di esecuzione pari a $\Theta(\lg^2 n)$. Infine il ciclo più esterno viene ripetuto n/2 volte e quindi il tutto ha tempo di esecuzione $\Theta(n \lg^2 n)$.

Ordinare un array quasi ordinato

Si vuole ordinare un array A di interi in cui ogni elemento è fuori posto di al più k posizioni.

a.Usando l'insertionSort il tempo è O(nk). Si spieghi perchè.

b.L'obiettivo è ottenere il risultato in O(n lg k). Suggerimento: si usi un min-heap di k+1 elementi

Esempio con k = 4:

A=

10	9	8	7	4	70	60	50
0	1	2	3	4	5	6	7

Esempio con k = 4:
4 è il minimo e il suo indice è 4, deve andare nella posizione 0 e 4-0 ≤ 4
7 è il secondo e il suo indice è 3, deve andare nella posizione 1 e 3-1 ≤ 4
8 è il terzo e il suo indice è 2, deve andare nella posizione 3 e 3-2 ≤ 4
9 è il quarto e il suo indice è 1, deve andare nella posizione 3 e 3-1 ≤ 4
50 è il quinto e il suo indice è 7, deve andare nella posizione 4 e 7-4 ≤ 4
60 è il sesto e il suo indice è 6, deve andare nella posizione 5 e 6-5 ≤ 4
70 è il settimo e il suo indice è 5, deve andare nella posizione 7 e 7-5 ≤ 4

Ordinare un array quasi ordinato: soluzione in O(nk)

```
InsertionSort(A)
n = len(A)
for (j = 1; j < n; j++)
%in ogni esecuzione del cici gli elementi di A[0:j-1]
sono una permutazione ordinata crescente
di quelli iniziali,  A[0] ≤A[1]≤...≤A[j-1]
     x = A[j]
     i = j-1
     while i ≥ 0 and x < A[i] do
          A[i+1] = A[i]
          i = i - 1
 A[i+1] = x
```

Se usassimo l'insertionSort, il ciclo più interno, il while, sarebbe eseguito al più k volte e quindi avremmo un tempo di esecuzione O(nk). Infatti nel ciclo più interno A[j] è spostato per essere inserito tra i primo j-1 elementi già ordinati, ma poiché dista al più di k posizioni dalla sua posizione giusta nell'ordinamento non sarà spostato più di k volte.
Nel caso peggiore abbiamo quindi un tempo in Θ(nk)

Esempio di esecuzione

```
InsertionSort(A)
n = len(A)
for (j = 1; j < n; j++)
%in ogni esecuzione del cici gli elementi di A[0:j-1]
sono una permutazione ordinata crescente
di quelli iniziali,  A[0] ≤A[1]≤...≤A[j-1]
      x = A[j]
      i = j-1
      while i ≥ 0 and x < A[i] do
            A[i+1] = A[i]
            i = i - 1
 A[i+1] = x
```

A=

10	9	8	7	4	70	60	50
0	1	2	3	4	5	6	7

Per j=1 i=0
Il ciclo è eseguito una volta

A=

9	10	8	7	4	70	60	50
0	1	2	3	4	5	6	7

Per j=2 i=1
Il ciclo è eseguito 2 volte

A=

8	9	10	7	4	70	60	50
0	1	2	3	4	5	6	7

Per j=3 i=2
Il ciclo è eseguito 3 volte

A=

7	8	9	10	4	70	60	50
0	1	2	3	4	5	6	7

Per j=4 i=3
Il ciclo è eseguito 4 volte

A=

8	9	10	7	4	70	60	50
0	1	2	3	4	5	6	7

Esempio di esecuzione: fine

```
InsertionSort(A)
n = len(A)
for (j = 1; j < n; j++)
%in ogni esecuzione del cici gli elementi di A[0:j-1]
sono una permutazione ordinata crescente
di quelli iniziali,  A[0] ≤A[1]≤...≤A[j-1]
     x = A[j]
     i = j-1
     while i ≥ 0 and x < A[i] do
          A[i+1] = A[i]
          i = i - 1
A[i+1] = x
```

A=

4	7	8	9	10	70	60	50
0	1	2	3	4	5	6	7

Per j=5 i=4
Il ciclo non è eseguito

A=

4	7	8	9	10	60	70	50
0	1	2	3	4	5	6	7

Per j=6 i=5
Il ciclo è eseguito 1 volta

A=

4	7	8	9	10	50	60	70
0	1	2	3	4	5	6	7

Per j=7 i=6
Il ciclo è eseguito 2 volte

Soluzione in O(n lgk)

Per ottenere l'ordinamento di A in O(n lg k) usiamo un min-heap di k+1 elementi, i primi k+1 dell'array, da cui estrarre il minimo e poi inserire un successivo elemento dell'array, di nuovo si estrae il minimo e si inserisce, fino ad esaurimento degli elementi di A, infine si rimuove il minimo dal min-heap fino a svuotarlo, copiando gli elementi in A.

.

Esempio con k = 4:

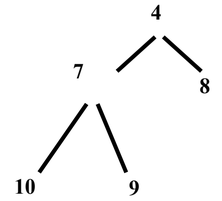
A=

10	9	8	7	4	70	60	50
0	1	2	3	4	5	6	7

Costruiamo un min-heap di 5 elementi:

H=

4	7	8	10	9
0	1	2	3	4



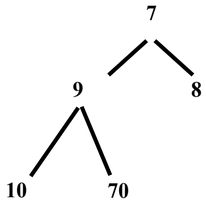
Rimuoviamo il minimo, copiandolo nella prima posizione in A e inseriamo 70 nel min-heap

A=

4	9	8	7	4	70	60	50
0	1	2	3	4	5	6	7

H=

7	9	8	10	70
0	1	2	3	4



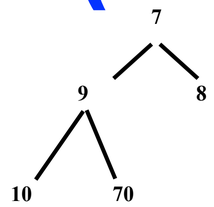
Esempio di esecuzione sol. in O(n lgk)

A=

4	9	8	7	4	70	60	50
0	1	2	3	4	5	6	7

H=

7	9	8	10	70
0	1	2	3	4



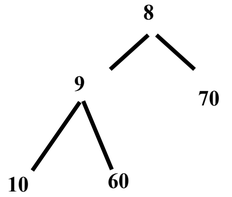
Rimuoviamo il minimo, copiandolo nella seconda posizione in A e inseriamo 60 nel min-heap

A=

4	7	8	7	4	70	60	50
0	1	2	3	4	5	6	7

H=

8	9	70	10	60
0	1	2	3	4



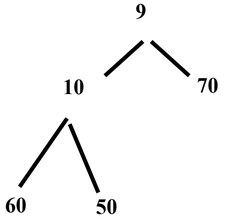
Rimuoviamo il minimo, copiandolo nella terza posizione in A e inseriamo 50 nel min-heap

A=

4	7	8	7	4	70	60	50
0	1	2	3	4	5	6	7

H=

9	10	70	60	50
0	1	2	3	4



Esempio di esecuzione sol. in O(n lgk)

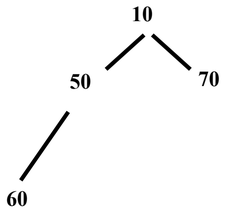
Rimuoviamo il minimo, copiandolo nella quarta posizione in A

A=

4	7	8	9	4	70	60	50
0	1	2	3	4	5	6	7

H=

10	50	70	60
0	1	2	3



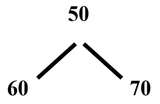
Rimuoviamo il minimo, copiandolo nella quinta posizione in A

A=

4	7	8	9	10	70	60	50
0	1	2	3	4	5	6	7

H=

50	60	70
0	1	2



Procedendo fino all'esurimento di H si ottiene

A=

4	7	8	9	10	50	60	70
0	1	2	3	4	5	6	7

Operazioni su min-heap necessarie

Le operazioni su min-heap che ci servono sono:

Heap-Extract-Min (A)
Input: A è un array

Prec: A è un min-heap

Output: dà in output il minimo, che viene rimosso da A rispristinando la proprietà del min-heap

Min-Heap-insert(A,key)

input: A è un array di interi e key un intero
prec: A è un min-heap
output: Inserisce key in A, ripristinando la proprietà del min-heap

Pseudocodice fine

```
OrdinaDistK(A, k)
Input: un array A di interi
prec: ogni elemento dista k posizione da quella che assumerebbe se l'array fosse ordinato, k < A.length
Output: A ordinato crescente
n = A.length
crea un array H di k+1 elementi
Copia i primi k+1 elementi di A in H
Build-Min-Heap(H)
j = 0;
for (i = k + 1; i < n; i++)
        A[j] = Heap-Extract-min(H)
        j++
        Min-Heap-insert(A,A[i])
 while (H.heapsize >0)
      A[j] = Heap-Extract-min(H)
      j++
```

Contare le coppie a differenza k in un array

Dato un array A di interi si vuole contare le coppie distinte di elementi la cui differenza è k.
Si vuole ottenere il risultato in O(nlg n)
Esempio con k = 4:

A=

8	12	16	4	0	20
0	1	2	3	4	5

Il risultato è 5, per via delle coppie (0, 4), (4, 8), (8, 12), (12, 16) and (16, 20)

A=

8	12	16	4	0	20	8
0	1	2	3	4	5	6

Il risultato è ancora 5, perchè si contano solo le coppie distinte.

Contare le coppie a differenza k in un array: soluzione

1) Ordina l'array A con un ordinamento in O(n lg n) //mergeSort o heapSort

2) inizializza a 0 un contatore

3) Per ogni elemento, A[i], tranne i duplicati

a)cerca con la ricerca binaria A[i] + k in A tra gli elementi di indice i+1 fino a n-1.

b) If A[i] + k viene trovato, incrementa il contatore

L'output è il contenuto del contatore.

Poiché il ciclo del passo 3 viene eseguito n volte e la ricerca binaria al più viene eseguita in O(lg n) si ottiene il risultato voluto: tutto l'algoritmo viene eseguito in O(n lg n).

Servirà dunque una funzione che implementa la ricerca binaria su una porzione di array:

binarySearch(A,i,j, key)

Input: un array A di interi e tre interi

Prec: A è ordinato crescente, $0 \leq i,j \leq n-1$

Output: l'indice di key, se presente nella porzione di array A[i..j], -1 altrimenti

Contare le coppie a differenza k in un array: soluzione

```
ContaCoppieDiffK(A, k)
Input: un array A di interi e un intero positivo k
Output: il numero di coppie di elementi di A la cui differenza è k
cont = 0
heapSort(A)
i = 0
while (i < n-1)
        if (binarySearch(A, i+1, n-1, A[i] + k) != -1)
            cont++
        while (A[i+1]=A[i]) i++
         i++
return cont
```