Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Esercizio 1.

Progettare un algoritmo che, dato un vettore A con n interi ed un intero k, determina se nel vettore A esistono due interi la cui somma è k.
L'algoritmo deve avere costo $O(n \log n)$ e restituire la coppia di elementi se questi esistono, $None$ altrimenti.

Esempio:
$A = [0, -1, 2, -3, 1]$ e $k = -2$ l'algoritmo restituisce la coppia $(-3, 1)$.

IDEA1: Per ogni elemento x di A possiamo scorrere tutti gli elementi che lo seguono in A alla ricerca di un elemento y tale che $x + y = k$.
Ecco di seguito l'algoritmo in python:

```
def es(A, k):
    n= len(A)
    for i in range(n-1):
        for j in range(i+1,n):
            if A[i]+ A[j] == k:
                return A[i],A[j]
    return Nane
```

La complessità di questo algoritmo è $\Theta(n^2)$. Infatti Per il numero totale di iterazioni dei due for si ha

$$\sum_{i=0}^{n-2} (n-1-i) = \sum_{j=1}^{n-1} j = \frac{(n-1)n}{2} = \Theta(n^2)$$

IDEA2: Posso ordinare il vettore A e poi scorrerlo e per ogni elemento x di A posso con la ricerca binaria verificare se in A è presente $k-x$.

L'ordinamento di A mi costa $\Theta(n \log n)$ utilizzando un qualunque algoritmo d'ordinamento efficiente mentre la ricerca dell'elemento x per cui esiste in A anche l'elemento $k-x$ richiederà al più n passi ciascuno di costo al più $\log n$ per un costo $O(n \log n)$.
La complessità di questo algoritmo è dunque $\Theta(n \log n)$.

IDEA3: dopo aver ordinato l'array A utilizziamo due indici i e j, inizializzati al primo ed all'ultimo elemento dell'array, rispettivamente.
I due indici scorrono il vettore da sinistra a destra il primo e da destra a sinistra il secondo, fino ad incontrarsi o a trovare la coppia cercata.

Ad ogni iterazione con $i < j$ si considera $A[i] + A[j]$ e:
se $A[i] + A[j] = x$ l'algoritmo termina restituendo la coppia (A[i], A[j])
se $A[i] + A[j] < x$ viene incrementato i
se $A[i] + A[j] > x$ viene decrementato j

La prima parte dell'algoritmo richiede tempo $\Theta(n \log n)$, utilizzando un qualunque algoritmo d'ordinamento efficiente.
La seconda parte esegue al più n passi costanti, e richiede dunque tempo $O(n)$.
Costo totale: $\Theta(n \log n)$

Ecco il codice in python:

```python
def es(A, k):
    A.sort()
    i, j = 0, len(A) - 1
    while i<j:
        if k == A[i] + A[j]:
            return A[i], A[j]
        if k< A[i] + A[j]:
            j -= 1
        else:
            i += 1
```

Esercizio 2.

Siano dati due array A e B, composti da $n \geq 1$ ed $m \geq 1$ numeri interi, rispettivamente. Gli array sono entrambi ordinati in senso crescente. A e B non contengono valori duplicati; tuttavia, uno stesso valore potrebbe essere presente una volta in A e una volta in B. Progettare un algoritmo iterativo efficiente che stampi i valori che appartengono all'unione di A e di B; l'unione va intesa in senso insiemistico, quindi gli eventuali valori presenti in entrambi i vettori devono essere stampati solo una volta.

Ad esempio, se $A = [1,3,4,6]$ e $B = [2,3,4,7]$, l'algoritmo deve stampare $1, 2, 3, 4, 6, 7$.

IDEA 1:

- considera gli elementi di *A* uno dopo l'altro e stampa solo quelli che non compaiono in *B*.
- considera gli elementi di *B* uno dopo l'altro e stampali tutti.

Ecco di seguito il codice python:

```
def es(A,B):
    for x in A:
        if x not in B:
            print(x)
    for x in B:
        print(x)
```

Per ogni elemento di *A*, scorro tutto *B* per vedere se è presente: Costo computazionale: $\Theta(nm)$.
ma non stiamo usando l'ipotesi che i due array siano ordinati!

IDEA2 Sfruttiamo il fatto che l'array B è ordinato.
considera gli elementi di *A* uno dopo l'altro e stampa solo quelli che non compaiono in *B*.
considera gli elementi di *B* uno dopo l'altro e stampali tutti.

eseguiamo, per ciascun elemento di *A*, una ricerca binaria su *B*. Costo computazionale di questo algoritmo è $O(n \log m + m)$

*A*ncora non stiamo sfruttando tutte le ipotesi perché così abbiamo usato il fatto che *B* sia ordinato ma l'ordinamento su *A* non viene sfruttato.

IDEA3 sfruttiamo il fatto che entrambi gli array sono ordinati e utilizziamo due indici: un indice *i* che scorre *A* ed un indice *j* che scorre *B* che partono entrambi da 0. Confrontiamo gli elementi $A[i]$ e $B[j]$ e operiamo in modo diverso a seconda dei casi:

se $A[i] < A[j]$ si stampa $A[i]$ e si incrementa i
se $A[i] > A[j]$ si stampa $A[j]$ e si incrementa j
se $A[i] = A[j]$ si stampa $A[i]$ e si incrementano sia i che j.

Appena uno dei due array termina, stampiamo tutti gli elementi dell'altro array.

Il ragionamento è simile a quello della fusione del Mergesort, in cui si trascrive solo uno degli elementi uguali.
Osserviamo che, a differenza della funzione di fusione del MergeSort, non abbiamo qui bisogno di un array ausiliario.
Un tale approccio ci porta ad un costo computazionale di $\Theta(n+m)$ in quanto sostanzialmente si effettua una scansione di entrambi i vettori esaminando ciascun elemento una e una sola volta in tempo $\Theta(1)$.

Ecco il codice in python:

```
def es(A, B):
    n, m = len(A), len(B)
    i = j = 0
    while i < n and j <m:
        if A[i] < B[j]:
            print( A[i])
            i += 1
        elif A[i] > B[j]:
            print(B[j])
            j += 1
        else:
            print( A[i]);
            i += 1
            j += 1
    while i < n: # stampa la parte finale di A
        print(A[i]); i+=1
    while j < m: # stampa la parte finale di B
        print(B[j]); j+=1
```

Esercizio 3.

Dato un vettore che contiene solo numeri negativi e positivi (nessun valore pari a zero), riorganizzarlo in modo che tutti i numeri negativi stiano a sinistra di quelli positivi.

IDEA1

Ordinare il vettore risolve il problema. Costo dell'algoritmo $\Theta(n \log n)$ utilizzando un efficiente algoritmo di ordinamento.

IDEA2 Utilizzo un vettore d'appoggio B. Scorro A e inserisco in coda a B tutti i valori negativi che incontro. Riscorro A e inserisco in coda a B tutti i valori positivi che incontro. Sposto in A tutti i valori che si trovano in B mantenendo l'ordine.

Si tratta di scorrere per tre volte vettori lunghi $n = len(A)$ Il costo computazionale dell'algoritmo è dunque $\Theta(n)$.

Ecco il codice python:

```
def es(A):
    B = [ ]
    for x in A:
        if x < 0:
            B.append(x)
    for x in A:
        if x > 0:
            B.append(x)
    for i in range(len(A)):
        A[i] = B[i]


>>> A= [3, 4, -1, 5, 6, -2, -7, -8, 9]
>>> es(A)
>>> A
[-1, -2, -7, -8, 3, 4, 5, 6, 9]
```

La precedente soluzione utilizzava tempo $\Theta(n)$ ma anche spazio $\Theta(n)$. La soluzione che proponiamo ora richiede spazio di lavoro $\Theta(1)$.

IDEA3. posso utilizzare due indici che scorrono l'array A. L'indice i che parte da zero e si incrementa e l'indice j che parte da $len(A)-1$ e si decrementa.
Utilizziamo l'invariante che i valori più piccoli di i sono negativi mentre i valori più grandi di j sono positivi; j su valori positivi si decrementa mentre i su valori negativi si incrementa se $i<j$ e i è su un valore positivo mentre j su un valore negativo allora avviene uno scambio e i si incrementa mentre j si decrementa.

I due puntatori si incontreranno dopo al più n incrementi decrementi e gli scambi hanno costo costante. Il costo computazionale dell'algoritmo è dunque $\Theta(n)$

Ecco il codice python:

```
def es(A):
    i, j = 0, len(A)-1
    while i < j:
        while A[i] < 0 and i <= j:
            i += 1
        while A[j] > 0 and j >= i:
            j -= 1
        if i < j:
            A[i], A[j] = A[j], A[i]
            i+=1; j-=1


>>> A = [3, 4, -1, 5, 6, -2, -7, -8, 9]
>>> es(A)
>>> A
[-8, -7, -1, -2, 6, 5, 4, 3, 9]
```

Esercizio 4.

Sia data una funzione f che prende un intero e restituisce un intero, la funzione è strettamente crescente (vale a dire $f(x)<f(x+1)$) e vogliamo trovare il primo intero n non negativo per cui la funzione assume un valore non negativo.

Ad esempio, per $f(x) = -100 + 3 \cdot x$ il valore da trovare è 34.

Assumendo che il calcolo di un valore di f costi $\Theta(1)$ *progettare* un algoritmo che trova questo valore in tempo $O(\log n)$.

IDEA1: Una semplice soluzione consiste nel cominciare a calcolare $f(0)$ e, se il valore è negativo, via via incrementare x fermandosi al primo x che dà un valore non negativo.

Nel caso dell'esempio:

$f(0) = -100$
$f(1) = -97$
$\ldots\ldots$
$f(33) = -1$
$f(34) = +2$

L'algoritmo è corretto ma la sua complessità è $\Theta(n)$.

IDEA2: Possiamo applicare la ricerca binaria ma per poterlo fare prima dobbiamo ricavare un limite superiore all'intervallo in cui ricercare:

1. Raddoppiamo ripetutamente il valore x su cui calcolare $f(x)$ fino a che non giungiamo ad un x' per cui $f(x') \geq 0$.

3. Applichiamo la ricerca binaria nell'intervallo $\left[\frac{x}{2}, \quad x'\right]$ alla ricerca della soluzione (vale a dire il minimo intero n per cui si ha $f(n) \geq 0$) .

nota che $f(x) \geq 0$ mentre $f\left(\frac{x}{2}\right) < 0$ quindi l'intero n che cerchiamo è proprio tra i valori dell'intervallo $\left[\frac{x}{2}, x\right]$

1. Raddoppiamo ripetutamente il valore x su cui calcolare $f()$ fino a che non giungiamo ad un x per cui $f(x) \geq 0$.
2. Applichiamo la ricerca binaria nell'intervallo $\left[\frac{x}{2}, x\right]$ alla ricerca della soluzione

Complessità dell'algoritmo è $O(\log n)$ infatti:

1. Il passo 1 richiede tempo $\Theta(\log n)$. La prima fase termina dopo $\lceil \log n \rceil$ invocazioni della funzione f infatti a quel punto si avrà $x = 2^{\lceil \log n \rceil} \geq n$ e quindi $f(x) \geq 0$.

2. Il passo 2 richiede tempo $O(\log n)$. Infatti l'intervallo $\left[\frac{x}{2}, x\right]$ su cui applichiamo la ricerca binaria ha estensione $x - \frac{x}{2} = \frac{x}{2} \leq \frac{2^{\log n+1}}{2} = n$.

Ecco il codice python:

```
def es(f) :
    if f(0) >= 0:
        return 0
    i = 1
    while f(i) <= 0:
        i = i * 2
    return ricercaBinaria(i//2, i, f)
```

```
def ricercaBinaria(i, j, f):
    m = (i + j)//2
    if f(mid) < 0 :
        # f(mid) e' negativo
        return ricercaBinaria(mid + 1, j, f)
    if f(mid) >= 0 and f(mid-1) < 0 :
        #f(mid) e' il primo non negativo
        return m
    else :
        # f(mid) è non negativo ma non il primo
        return ricercaBinaria(i, mid - 1, f)
```

```
def f(x): return -100+3*x

>>> es(f)
34
```