Algoritmi greedy

Progettazione di Algoritmi a.a. 2021-22
Matricole congrue a 1
Docente: Annalisa De Bonis

1

1

Scelta greedy

Un algoritmo greedy è un algoritmo che effettua ad ogni passo la scelta che in quel momento sembra la migliore (localmente ottima) nella speranza di ottenere una soluzione globalmente ottima.

Domanda. Questo approccio porta sempre ad una soluzione ottima?
Risposta. Non sempre ma per molti problemi sì.
Esempio:
Voglio andare in auto dalla città a alla città b. La distanza tra a e b e di 1500 km. Lungo la strada ci sono n punti in cui posso fermarmi a fare rifornimento di carburante (a e b compresi). Voglio minimizzare il numero di volte in cui devo fermarmi per fare rifornimento considerando che con un pieno posso percorrere 450 km e che alla partenza il serbatoio è vuoto.
Strategia greedy: faccio il pieno in a e arrivo al più lontano punto di rifornimento che riesco a raggiungere (distanza da a ≤ 450 km). Da questo punto raggiungo il punto di rifornimento più lontano che si trova ad al più 450 km da esso, e così via fino a che non arrivo in b. Si può dimostrare che questa strategia fornisce la soluzione ottima.

2

Scelta greedy

- Esempio di problema per cui questo approccio non porta alla soluzione ottima
- Problema dello zaino
- Input
  - n oggetti ed uno zaino
  - L'oggetto i pesa $w_i > 0$ chili e ha valore $v_i > 0$.
  - Lo zaino può trasportare fino a W chili.
- Obiettivo: riempire lo zaino in modo da massimizzare il valore totale degli oggetti inseriti tenendo conto che lo zaino trasporta al più W chili
- Esempio: { 3, 4 } ha valore 40.

W = 11

Oggetto	Valore	Peso
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

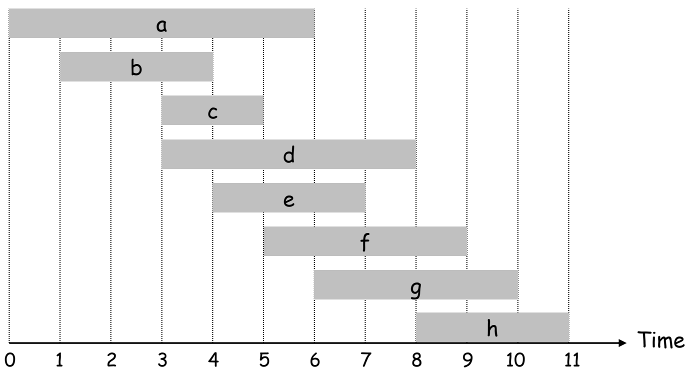
Proviamo diverse strategie greedy

1. seleziona ad ogni passo l'oggetto con valore v_i più grande in modo che il peso totale dei pesi selezionati non superi w → {5,2,1} valore=35
2. seleziona ad ogni passo l'oggetto con peso w_i più piccolo in modo che il peso totale dei pesi selezionati non superi w → {1,2,3} valore=25
3. seleziona ad ogni passo l'oggetto con il rapporto v_i/w_i più grande in modo che il peso totale dei pesi selezionati non superi w → { 5,2,1 }, valore = 35

Interval Scheduling

Interval scheduling.

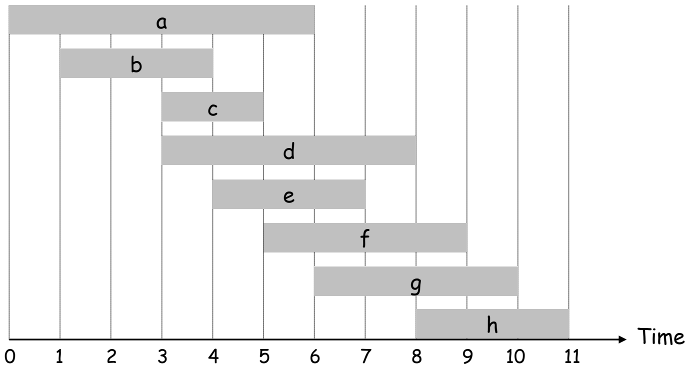
- Input: un insieme di attività ognuna delle quali inizia ad un certo instante s_j e finisce ad un certo istante f_j. Può essere eseguita al più un'attivita alla volta.
- Obiettivo: fare in modo che vengano svolte quante più attività è possibile.



Interval Scheduling

Due job i e j si dicono compatibili se $f_i \leq s_j$ oppure $f_j \leq s_i$.

- Possiamo riformulare l'obiettivo del problema nel modo seguente.
- Obiettivo: trovare un sottoinsieme di cardinalità massima di job a due a due compatibili.



5

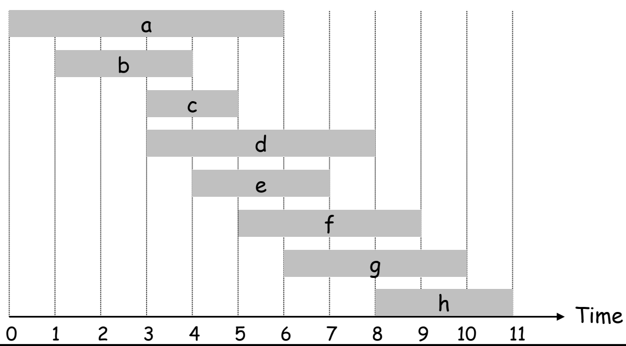
Interval Scheduling

Se all'inizio scegliamo a poi possiamo scegliere o g o h.

- Dopo aver scelto a e g oppure a e h, non possiamo scegliere nessun altro job. Totale = 2

Se all'inizio scegliamo b poi possiamo scegliere uno tra e, f, g e h.

- Se dopo b scegliamo e poi possiamo scegliere anche h. Totale = 3
- Se dopo b scegliamo f poi non possiamo scegliere nessun altro job. Totale = 2



6

Interval Scheduling: Algoritmi Greedy

Schema greedy. Considera i job in un certo ordine. Ad ogni passo viene esaminato il prossimo job nell'ordinamento e se il job è compatibile con quelli scelti nei passi precedenti allora il job viene inserito nella soluzione.

L'ordinamento dipende dal criterio che si intende ottimizzare localmente.

Diverse strategie basate su diversi tipi di ordinamento

- [Earliest start time] Considera i job in ordine crescente rispetto ai tempi di inizio s_j. Scelta greedy consiste nel provare a prendere ad ogni passo il job che inizia prima tra quelli non ancora esaminati.
- [Earliest finish time] Considera i job in ordine crescente rispetto ai tempi di fine f_j. Scelta greedy consiste nel provare a prendere ad ogni passo il job che finisce prima tra quelli non ancora esaminati.
- [Shortest interval] Considera i job in ordine crescente rispetto alle lloro durate $f_j - s_j$. Scelta greedy consiste nel provare a prendere ad ogni passo il job che dura meno tra quelli non ancora esaminati.
- [Fewest conflicts] Per ogni job, conta il numero c_j di job che sono in conflitto con lui e ordina in modo crescente rispetto al numero di conflitti. Scelta greedy consiste nel provare a prendere ad ogni passo il job che ha meno conflitti tra quelli non ancora esaminati.

7

7

Interval Scheduling: Algoritmi Greedy

La strategia "Earliest Start Time" sembra la scelta più ovvia ma…

Problemi con la strategia "Earliest Start Time". Può accadere che il job che comincia per primo finisca dopo tutti gli altri o dopo molti altri.

PROGETTAZIONE DI ALGORITMI A.A. 2021-22
A. De Bonis

8

8

Interval Scheduling: Algoritmi Greedy

Ma se la lunghezza dei job selezionati incide sul numero di job che possono essere selezionati successivamente perché non provare con la strategia "Shortest Interval"?

Questa strategia va bene per l'input della slide precedente ma…

Problemi con la strategia "Shortest Interval". Può accadere che un job che dura meno di altri si sovrapponga a due job che non si sovrappongono tra di loro. Se questo accade invece di selezionare due job ne selezioniamo uno solo.

Interval Scheduling: Algoritmi Greedy

Visto che il problema sono i conflitti, perché non scegliamo i job che confliggono con il minor numero di job?

Questa strategia va bene per l'input nella slide precedente ma…

Problemi con la strategia "Fewest Conflicts". Può accadere che un job che genera meno conflitti di altri si sovrapponga a due job che non si sovrappongono tra di loro. Se applichiamo questa strategia all'esempio in questa slide, invece di selezionare 4 job ne selezioniamo solo 3.

Interval Scheduling: Algoritmo Greedy Ottimo

L'algoritmo greedy che ottiene la soluzione ottima è quello che usa la stategia "Earliest Finish Time".

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
f ← 0
A ← φ
for j = 1 to n {
   if (sj ≥ f)
      A ← A ∪ {j}
      f ← fj
}
return A
```

Analisi tempo di esecuzione. $O(n \log n)$.

- Costo ordinamento $O(n \log n)$
- Costo for $O(n)$: mantenendo traccia del tempo di fine f dell'ultimo job selezionato, possiamo capire se il job j è compatibile con A verificando che $s_j \geq f$

11

11

Interval Scheduling: Ottimalità soluzione greedy

Teorema. L'algoritmo greedy basato sulla strategia "Earliest Finish Time" è ottimo.

Dim.

- Denotiamo con $i_1, i_2, \ldots i_k$ l'insieme di job selezionati dall'algoritmo greedy nell'ordine in cui sono selezionati, cioè in ordine non decrescente rispetto ai tempi di fine.
- Denotiamo con $j_1, j_2, \ldots j_m$ l'insieme di job nella soluzione ottima, disposti in ordine non decrescente rispetto ai tempi di fine.

1. Dimostriamo prima che l'esecuzione dei job $i_1, i_2, \ldots i_k$ termina non più tardi di quella dei job $j_1, j_2, \ldots j_k$
2. Usiamo il punto 1 per dimostrare che non è possibile che k sia minore di m e che quindi la soluzione greedy è ottima.

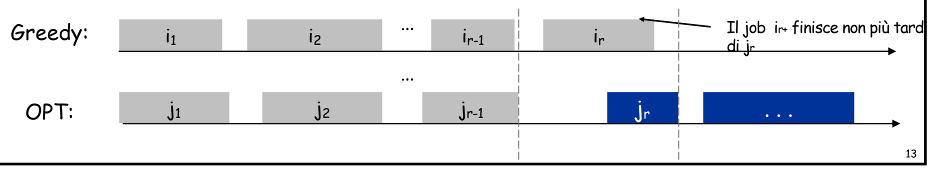
12

12

Interval Scheduling: Ottimalità soluzione greedy

Dimostriamo il punto 1.:

- Dimostriamo per induzione che per ogni indice r, $1 \le r \le k$, si ha che il tempo di fine di i_r è **non** più grande di quello di j_r .
- **Base.** r=1 : Banalmente vero perchè la prima scelta greedy seleziona la richiesta con il minimo tempo di fine.
- **Passo induttivo. Ipotesi induttiva:** per $r-1 \ge 1$, il tempo di fine di i_{r-1} è **non** più grande di quello di j_{r-1} .
- Il job j_r è compatibile con j_{r-1} e quindi $f(j_{r-1}) \le s(j_r)$ e siccome per ipotesi induttiva $f(i_{r-1}) \le f(j_{r-1}) \rightarrow f(i_{r-1}) \le s(j_r) \rightarrow j_r$ è compatibile con i_{r-1} .
- I tempi di fine di $i_1, i_2, \dots i_{r-2}$ sono non più grandi di quello di $i_{r-1} \rightarrow f(i_p) \le s(j_r)$ per ogni $1 \le p \le r-1 \rightarrow j_r$ è compatibile con $i_1, i_2, \dots i_{r-2}\ i_{r-1}$.
- All'r-esimo passo l'algoritmo greedy sceglie $i_r \rightarrow i_r$ finisce non più tardi di tutti i job compatibili con $i_1, i_2, \dots i_{r-1}$ e quindi anche non più tardi di j_r.



Interval Scheduling: Ottimalità soluzione greedy

- Abbiamo dimostrato che, per ogni indice $r \le k$, il tempo di fine di i_r è **non** più grande di quello di j_r .

  → il tempo di fine di i_k è non più grande di quello di j_k .
- Poiche' i job $i_1, i_2, \dots i_k$ sono ordinati in base ai tempi di fine

  → i_k è il job che finisce piu\` tardi tra i job $i_1, i_2, \dots i_k$
- → e → insieme implicano che tutti i job $i_1, i_2, \dots i_k$ terminano non piu\` tardi di $j_k$ e di conseguenza non piu\` tardi della sequenza di job $j_1, j_2, \dots j_k$

Continua nella prossima slide

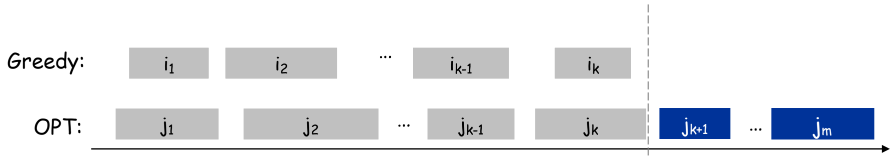
Interval Scheduling: Ottimalità soluzione greedy

Dimostriamo il punto 2.

• Supponiamo per assurdo che la soluzione greedy non sia ottima e quindi che $k < m$. Quindi la sequenza $j_1, j_2, \dots j_m$ conterrà il job j_{k+1}

• Per il punto 1, l'esecuzione di $i_1, i_2, \dots i_k$ termina **non** più tardi dell'esecuzione di $j_1, j_2, \dots j_k$. e si ha quindi che $i_1, i_2, \dots i_k$ sono compatibili con j_{k+1}

• L'algoritmo greedy, dopo aver inserito $i_1, i_2, \dots i_k$ nella soluzione, passa ad esaminare i job con tempo di fine maggiore o uguale a quelli di $i_1, i_2, \dots i_k$. Tra questi job vi è j_{k+1} che è compatibile con $i_1, i_2, \dots i_k$ per cui è impossibile che l'algoritmo greedy inserisca nella soluzione solo k job. Siamo giunti ad una contraddizione e quindi è impossibile che $k < m$.



Provare l'ottimalità della soluzione greedy

Come abbiamo provato l'ottimalità della soluzione greedy?

- Abbiamo prima di tutto dimostrato che la soluzione greedy "sta sempre avanti" a quella ottima.
  - Cosa vuol dire "sta sempre avanti"?
  - L'idea alla base della strategia greedy Earliest Finish Time è la seguente: quando si usa la risorsa è bene liberarla il prima possibile perchè ciò massimizza il tempo a disposizione per eseguire le restanti richieste
  - In questa ottica, una soluzione per il problema dell'interval scheduling "sta sempre avanti" ad un'altra se ad ogni passo seleziona una richiesta che termina non più tardi della corrispondente richiesta della soluzione ottima.
- Abbiamo usato il fatto che la soluzione greedy "sta sempre avanti" a quella ottima per provare che la soluzione greedy non può contenere un numero di richieste inferiore a quello della soluzione ottima

Partizionamento di intervalli

In questo caso disponiamo di più risorse identiche tra di loro e vogliamo che vengano svolte tutte le attività in modo tale da usare il minor numero di risorse e tenendo conto del fatto che due attività non possono usufruire della stessa risorsa allo stesso tempo.

- Durante il suo svolgimento, ciascuna attività necessita di un'unica risorsa.
- Una risorsa può essere allocata ad al più un'attività alla volta

Nell'interval scheduling avevamo un'unica risorsa e volevamo che venissero svolte il massimo numero di attività

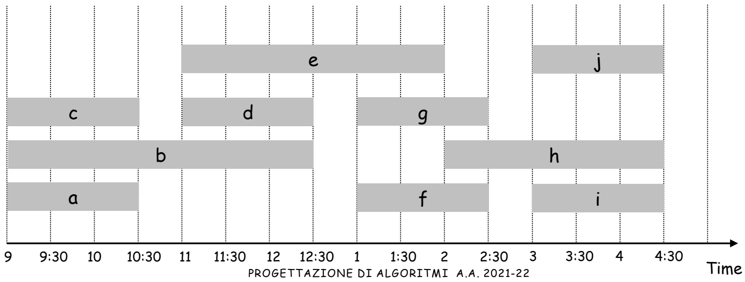
Un'istanza del problema (Input) consiste in un insieme di n intervalli $[s_1,f_1],\dots,[s_n,f_n]$ che rappresentano gli intervalli durante i quali si svolgono le n attività.
Obiettivo: far svolgere le n attività utilizzando il minor numero possible di risorse e in modo che ciascuna risorsa utilizzata venga allocata ad al più un'attività alla volta.

Partizionamento di intervalli

Esempio. Attività = lezioni da svolgere; Risorse= aule

- La lezione j comincia ad s_j e finisce ad f_j.
- Obiettivo: trovare il minor numero di aule che permetta di far svolgere tutte le lezioni in modo che non ci siano due lezioni che si svolgono contemporaneamente nella stessa aula.

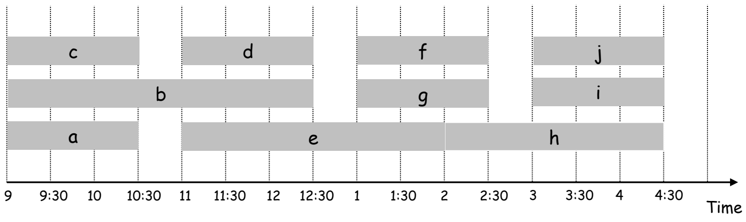
Esempio: Questo schedule usa 4 aule (una per livello) per 10 lezioni: e, j nella prima aula; c,d,g nella seconda aula; b, h nella terza aula; a, f, i nella quarta aula



Partizionamento di intervalli

Esempio. Questo schedule usa solo 3 aule per le stesse attività: {c,d,f,j},{b,g,i},{a,e,h} .

Si noti che la disposizione delle lezioni lungo l'asse delle ascisse è fissato dall'input mentre la disposizione lungo l'asse delle y è fissato dall'algoritmo.



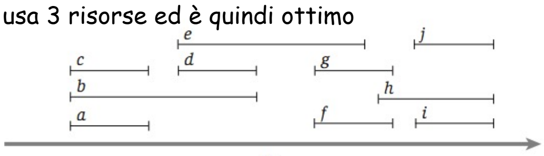
Partizionamento di intervalli: limite inferiore alla soluzione ottima

Def. Immaginiamo di disporre gli intervalli lungo l'asse delle ordinate in modo da non avere due intervalli che si sovrappongono alla stessa altezza (un qualsiasi schedule fa al nostro caso) . La profondità di un insieme di intervalli è il numero massimo di intervalli intersecabili con una singola linea verticale che si muove lungo l'asse delle ascisse.

Osservazione. Numero di risorse necessarie $\geq$ profondità.

Esempio. L'insieme di intervalli in figura (a) ha profondità 3. Lo schedule in figura (b) usa 3 risorse ed è quindi ottimo

Una risorsa per livello. Per un totale di 4 risorse



(a)

Una risorsa per livello. Per un totale di 3 risorse

(b)

Partizionamento di intervalli: soluzione ottima

Domanda. E' sempre possibile trovare uno schedule pari alla profondità dell'insieme di intervalli?
Osservazione. Se così fosse allora il problema del partizionamento si ridurrebbe a constatare quanti intervalli si sovrappongono in un certo punto.

- Questa è una caratteristica locale per cui un algoritmo greedy potrebbe essere la scelta migliore:
  - alloco una nuova risorsa solo se in un certo momento quelle gia` allocate sono tutte impegnate

Nel seguito vedremo un algoritmo greedy che trova una soluzione che usa un numero di risorse pari alla profondità e che quindi è ottimo
Idea dell'algoritmo applicata all'esempio delle lezioni:
- Considera le lezioni in ordine non decrescente dei tempi di inizio
- Ogni volta che esamina una lezione controlla se le può essere allocata una delle aule già utilizzate per qualcuna delle lezioni esaminate in precedenza. In caso contrario alloca una nuova aula.

Partizionamento di intervalli: Algoritmo greedy

```
Input: s1,s2, ... ,sn , f1,f2, ... ,fn
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤ sn.
d ← 0
S ← Ø
for j = 1 to n {
   if (interval j can be assigned an already allocated resources v)
      assign resource v to interval j
      S ← S U {(j,v)}
   else
      allocate a new resource d + 1
      assign the new resource d+1 to interval j
      S ← S U {(j,d+1)}
      d ← d + 1
}
return S
```

- Osservazione. L'algoritmo greedy non assegna mai la stessa risorsa a due intervalli incompatibili

- Dimostreremo che alla j-esima iterazione del for il valore di d è pari alla profondità dell'insieme ordinato di intervalli {1,2,…,j}
- Il valore finale di d è quindi pari alla profondità dell'insieme di intervalli {1,…,n}

Partizionamento di intervalli: Ottimalità soluzione greedy

Lemma. Alla j-esima iterazione del for, il valore di d in quella iterazione è pari alla profondità dell'insieme di intervalli {1,2,….,j}.

Dim.

Caso in cui alla j-esima iterazione del for viene allocata una nuova risorsa

- Supponiamo che, alla j-esima iterazione del ciclo di for, d venga incrementato.
- La risorsa d è stata allocata perchè ciascuna delle altre d-1 risorse già allocate è assegnata al tempo s_j ad un intervallo che non è ancora terminato.
  → $f_i > s_j$ per ciascuna attività i che impegna una delle d-1 risorse
- **Siccome l'algoritmo considera gli intervalli in ordine non decrescente dei tempi di inizio**, i d-1 intervalli a cui sono assegnate le d-1 risorse iniziano non più tardi di s_j → $s_i \le s_j$ e $f_i > s_j$, per ciascuna attività i che impegna una delle d-1 risorse
- Di conseguenza, questi d-1 intervalli e l'intervallo $[s_j,f_j]$ si sovrappongono al tempo s_j (sono cioè tutti intersecabili da una retta verticale che passa per s_j). Per definizione di profondità, d ≤ profondità di {1,2,….,j}
- Abbiamo osservato che il numero di risorse allocate per intervallo è maggiore uguale della sua profondità per cui d ≥ profondità di {1,2,….,j}
- Dagli ultimi due punti segue che d = profondità di {1,2,….,j}

23

23

Partizionamento di intervalli: Ottimalità soluzione greedy

Caso in cui alla j-esima iterazione del for non viene allocata una nuova risorsa:

- Sia j' l'ultima iterazione prima della j-esima in cui viene allocata una nuova risorsa.
- Per quanto dimostrato nella slide precedente, d= profondità {1,2,….,j'}.
- Siccome {1,2,….,j'} è contenuto in {1,2,….,j} allora si ha che profondità {1,2,….,j'} ≤ profondità di {1,2,….,j} e quindi per il punto precedente d ≤ profondità di {1,2,….,j}.
- Usiamo di nuovo il fatto che il numero di risorse allocate per un insieme di intervalli è maggiore o uguale della profondità dell'insieme. Si ha quindi d ≥profondità di {1,2,….,j}.
- Gli ultimi due punti implicano che d=profondità di {1,2,….,j}.

PROGETTAZIONE DI ALGORITMI A.A. 2021-22
A. De Bonis

24

24

Partizionamento di intervalli: Ottimalità soluzione greedy

Teorema. L'algoritmo greedy usa esattamente un numero di risorse pari alla profondità dell'insieme di intervalli {1,2,…,n}

Dim. Per il lemma precedente, quando j=n il numero di risorse allocate è uguale alla profondità dell'intervallo {1,2,…,n}

Partizionamento di intervalli: Analisi Algoritmo greedy

Implementazione. O(n log n).

- Per ogni risorsa p, manteniamo il tempo di fine più grande tra quelli degli intervalli assegnati fino a quel momento a p. Indichiamo questo tempo con k_p
- Usiamo una coda a priorità di coppie della forma (p, k_p), dove p è una risorsa già allocata e k_p è l'istante fino al quale è occupata.
  - In questo modo l'elemento con chiave minima indica la risorsa v che si rende disponibile per prima
- Se $s_j \geq k_v$ allora all'intervallo j può essere allocata la risorsa v. In questo caso cancelliamo v dalla coda e la reinseriamo con chiave $k_v=f_j$. In caso contrario allochiamo una nuova risorsa e la inseriamo nella coda associandole la chiave f_j
- Se usiamo una coda a priorità implementata con heap binario ogni operazione sulla coda richiede O(log m), dove m è la profondità dell'insieme di intervalli. Poiché vengono fatte O(n) operazioni di questo tipo, il costo complessivo del for è O(n log m).
- A questo va aggiunto il costo dell'ordinamento che è O(n log n). Siccome m ≤ n il costo dell'algoritmo è O(n logn)