Corso di laurea in Informatica
Progettazione d'algoritmi

Gli algoritmi greedy

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Algoritmi Greedy

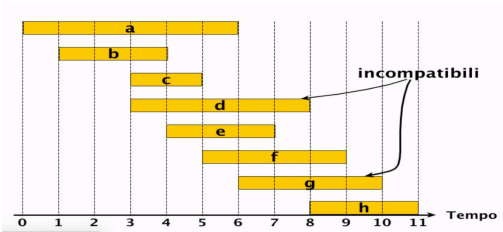
Per illustrare il progetto e l'analisi di un algoritmo *greedy* consideriamo ora un problema piuttosto semplice chiamato selezione di attività.

Abbiamo una lista di n attività da eseguire:

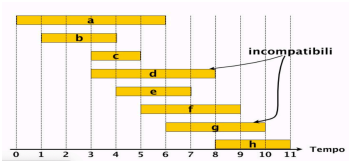
- ciascuna attività è caratterizzata da una coppia con il suo tempo di inizio ed il suo tempo di fine
- due attività sono compatibili se non si sovrappongono.

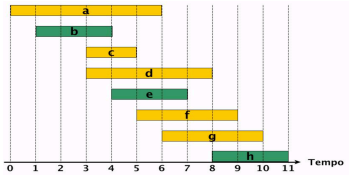
Vogliamo trovare un sottoinsieme di attività compatibili di massima cardinalità.

ESEMPIO: Istanza del problema con ***n = 8***.



ESEMPIO: Istanza del problema con ***n = 8***.





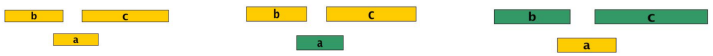
Per le 8 attività l'insieme da selezionare è {b,e,h}. In questo caso è facile convincersi che non ci sono altri insiemi di 3 attività compatibili e non c'è alcun insieme di 4 attività compatibili. In generale possono esistere diverse soluzioni ottime.

Volendo utilizzare il paradigma *greedy* dovremmo trovare una regola, semplice da calcolare, che ci permetta di effettuare ogni volta la scelta giusta.

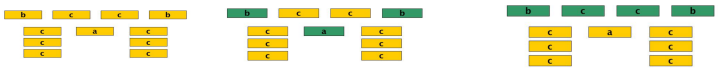
Per questo problema ci sono diverse potenziali regole di scelta:

- prendi l'attività compatibile che inizia prima:

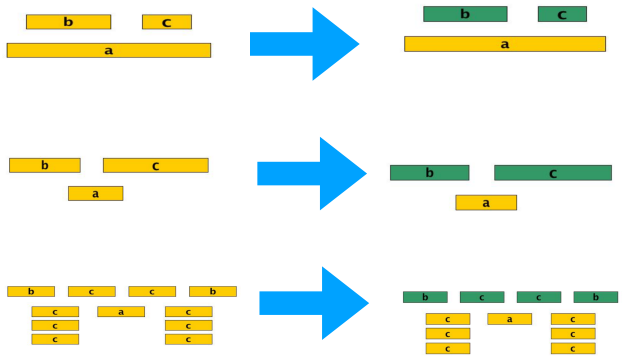
- prendi l'attività compatibile che dura meno:



- prendi l'attività compatibile che ha meno confltti con le rimanenti:



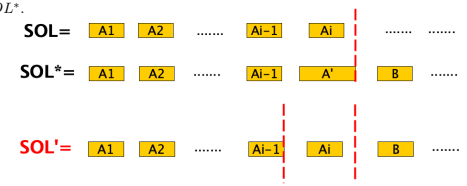
La regola giusta sta nel prendere sempre l'attività compatibile che finisce prima



La regola giusta sta nel prendere sempre l'attività compatibile che finisce prima

PROVA: Supponiamo per assurdo che la soluzione greedy SOL trovata con questa regola non sia ottima. Le soluzioni ottime differiscono dunque da SOL. Nel caso ci fossero più di una soluzione ottima prendiamo quella che differisce nel minor numero di attività da SOL, sia SOL^*. Dimostreremo ora che esiste un'altra soluzione ottima SOL' che differisce ancora meno da SOL il che è assurdo.

Siano $A1$, $A2$, ... le attività nell'ordine in cui sono state selezionate dal greedy sia Ai la prima attività scelta dal greedy e non dall'ottimo (questa attività deve esistere perché tutte le attività scartate dal greedy erano incompatibili con quelle prese dal greedy e se la soluzione avesse preso tutte le attività scelte dal greedy non potrebbe averne prese di più). Nell'ottimo deve esserci un'altra attività A' che va in conflitto con A_i (altrimenti SOL^* non sarebbe ottima in quanto potrei aggiungervi l'attività A_i). A questo punto posso sostituire in SOL^* l'attività A' con l'attività A_i senza creare conflitti (perché in base alla regola di scelta greedy A_i termina prima di A'). Ottengo in questo modo una soluzione ottima SOL' (infatti le attività in SOL' sono tutte compatibili e la cardinalit'a di SOL' è la stessa di SOL^*). Ma SOL' differisce da SOL di un'attività in meno rispetto a SOL^*.



Implementazione:

$$
\begin{array}{l}
\textbf{def}\ \ selezione_a(lista):\\
\quad Soluzione\ =\ set()\\
\quad \textbf{while}\ \ lista:\\
\qquad estrai\ \ da\ \ lista\ \ l'attivita'\ \ (a,b)\ \ che\ \ termina\ \ prima\\
\qquad \textbf{if}\ \ (a,b)\ \ non\ \ va\ \ in\ \ conflitto\ \ con\ \ le\ \ attivita'\ \ in\ \ Soluzione:\\
\qquad\qquad Soluzione.add((a,b))\\
\quad \textbf{return}\ \ Soluzione
\end{array}
$$

Idee

- cercare in *lista* l'attività che finisce prima scorrendo la lista costerebbe $O(n)$ ad ogni iterazione del *while*. Conviene fare un preprocessing ordinando le attività in *lista* per tempo di fine crescente. In questo modo paghiamo $O(n \log n)$ per l'ordinamento ma poi l'estrazione dell'attività ad ogni iterazione del file costerà $O(1)$
- per eseguire efficientemente il test richiesto dall'if conviene tenere una variabile con il tempo di fine dell'ultima attività inserita in **soluzione** in questo modo il test richiederà $O(1)$

Implementazione:

```
def selezione_a(lista):
    lista.sort(key=lambda x:x[1])
    libero,  sol = 0, set()
    for x,y in lista:
        if libero<x:
            sol.add((x,y))
            libero=y
    return sol

>>> lista=[(1,5),(6,11),(12,17),(18,22),(2,9),(10,13),(14,21),(3,8),(15,20),(4,7),(16,19)]
>>> attivita(lista)
{(6, 11), (1, 5), (18, 22), (12, 17)}
```

Complessità:

Ordinare la lista delle attività costa $O(n \log n)$. Il for viene eseguito n volte e il costo di ogni iterazione è costante. Il costo finale è $O(n \log n)$.

Consideriamo ora questo nuovo problema noto come assegnazione di attività:

Abbiamo una lista di attività, ciascuna caratterizzata da un tempo di inizio ed un tempo di fine. **Le attività vanno tutte eseguite** e vogliamo assegnarle al minor numero di persone tenendo conto che una persona non può eseguire più attività in parallelo.

$$lista = [(1,4),\ (1,6),\ (7,8),\ (5,10)]$$

possibile assegnazione

assegnazione ottima

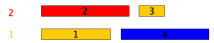
Un possibile algoritmo *greedy* si basa sull'idea assumere operai fino a che ci sono attività da assegnare e ad ogni operaio, una volta assunto, assegnare il maggior numero di attività non ancora assegnate che è in grado di svolgere.

```
def assegnazione_a(lista) :
    i = 0
    while lista :
        i += 1
        lista1 = selezione_a(lista)
        elimina da lista i lavori in lista1
    return D
```

L'algoritmo proposto non è corretto: $lista = [(1,4),\ (1,6),\ (7,8),\ (5,10)]$

soluzione prodotta:

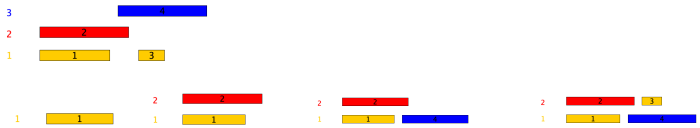
soluzione ottima:



Algoritmo greedy alternativo:

```
def assegnazione_a(lista) :
    inizializza la soluzione con una persona senza lavori assegnati
    while lista :
        estrai da lista l'attivita' (a,b) che inizia prima
        if c'e' nella soluzione un operaio in grado di eseguirla:
            assegna (a,b) a quell' operaio
        else :
            inserisci nella soluzione un nuovo operaio ed assegnagli (a,b)
    return la soluzione con gli operai e i lavori assegnati
```

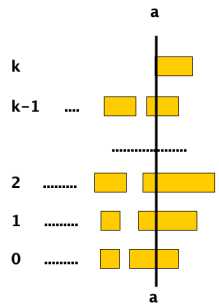
$$lista = [(1,4),\ (1,6),\ (7,8),\ (5,10)]$$




```
def partiziona(lista) :
    inizializza la soluzione con un operaio senza lavori assegnati
    while lista :
        estrai da lista l'attivita' (a,b) che inizia prima
        if c'e' nella soluzione un operaio in grado di eseguirla:
            assegna (a,b) a quell' operaio
        else :
            inserisci nella soluzione una nuovo operaio ed assegnagli (a,b)
    return la soluzione con gli operai e i lavori loro assegnati
```

CORRETTEZZA: Sia k il numero di operai utilizzati dalla soluzione. Faremo vedere che ci sono nella lista k attività incompatibili a coppie e questo ovviamente implica che k operai sono necessari.

Sia (a,b) l'attività che ha portato all'introduzione nella soluzione del k-mo operaio. In quel momento gli altri $k-1$ operai erano impossibilitate ad eseguire l'attività (a,b) ma per il criterio di scelta greedy posso anche dire che nell'istante di tempo a erano impegnati (le attività con cui ciascuno di loro era impegnato iniziavano prima del tempo a e non erano ancora finite) quindi nell'istante di tempo a a due a due tutte queste k attività sono incompatibili.



Implementazione

def *partiziona(lista)* :

inizializza la soluzione con una persona senza lavori assegnati

while *lista* :

estrai da lista l'attivita' (a,b) che inizia prima

if *c'e' nella soluzione una persona in grado di eseguirla*:

assegna (a,b) a quella persona

else :

inserisci nella soluzione una nuova persona ed assegnale (a,b)

return *la soluzione con le persone e i lavori assegnati*

IDEA:

1) per individuare efficientemente l'attività che inizia prima effettuo un preprocessing in cui ordino le attività per tempo di inizio.

2) per individuare efficientemente se uno degli operai in soluzione è in grado di eseguire un certo lavoro (a,b) uso un *heap* minimo in cui metto le coppie $(libero, i)$ dove *libero* indica il tempo in cui si libera l'operaio i. In questo modo in tempo $O(1)$ posso sapere qual'è la persona che si libera prima semplicemente verificando se $H[0]$

Se l'operaio che si libera prima può eseguire il lavoro allora dovrò assegnarglielo e aggiornare il valore *libero* della coppia che la rappresenta nell'heap. Se al contrario non può eseguire il lavoro allora nessuno degli altri operai sarà in grado di farlo e bisognerà assegnare l'attività ad una nuova persona ed inserire nell'heap la coppia che rappresenta questa nuova persona.

Inserimenti e cancellazioni dall'heap costeranno $O(\log n)$

Implementazione

```
def assegnazione_a(lista):
    from heapq import heappop, heappush
    sol=[[ ]]
    H=[(0,0)]
    lista.sort()
    for x,y in A:
        libero, operaio = H[0]
        if libero < x:
            sol[operaio].append((x,y))
            heappop(H)
            heappush(H,(y,operaio))
        else:
            sol.append([(x,y)])
            heappush(H,(y,len(sol)-1))
    return sol

>>>lista=[(1,4),(1,6),(7,8),(5,10)]
>>> assegnazione_a(lista)
[
[(1, 4), (5, 10)],
[(1, 6), (7, 8)]
]
```

```
>>> lista=[(1,5),(6,11),(12,17),(18,22),(2,9),(10,13),(14,21),(3,8),(15,20),(4,7),(16,19)]
>>> assegnazione_a(lista)
[
[(1, 5), (6, 11), (15, 20)],
[(2, 9), (14, 21)],
[(3, 8), (12, 17), (18, 22)],
[(4, 7), (10, 13), (16, 19)]
]
```

Complessità:

- ordinare la lista costa $O(n \log n)$
- il *for* viene eseguito n volte ed all'interno del *for* al caso pessimo può essere eseguita una estrazione seguita subito dopo da un inserimento nell'*heap* H, operazioni di costo $O(logn)$. Il *for* richiederà quindi tempo $O(n \log n)$.

La complessità dell'algoritmo è $O(n \log n)$.

ESERCIZIO: Abbiamo n file di dimensioni $d_0, d_1, \ldots, d_{n-1}$ che vorremmo memorizzare su un disco di capacità k. Tuttavia la somma delle dimensioni di questi file eccede la capacità del disco. Vogliamo dunque selezionare un sottoinsieme degli n file che abbia cardinalità massima e che possa essere memorizzato sul disco.

Descrivere un algoritmo *greedy* che risolve il problema in tempo $O(n \log n)$ e provarne la correttezza.

Ad esempio per $D = [5, 6, 3, 5, 4, 7, 3]$ **e** $k = 11$
la risposta deve essere la tripla di file $2,4$ **e** 6 **che occupa spazio** 10

- Un algoritmo *greedy* per questo problema è quello che si presenta naturalmente: consideriamo i file per dimensione crescenti e se c'è spazio per memorizzare il file su disco allora facciamolo.

- Un algoritmo *greedy* per questo problema è quello che si presenta naturalmente: consideriamo i file per dimensione crescente e se c'è spazio per memorizzare il file su disco allora facciamolo.

```
def file(D,k):
    lista=[(D[i],i) for i in range(len(D))]
    lista.sort()
    spazio, sol = 0, []
    for d,i in lista:
        if spazio+d<=k:
            sol.append(i)
            spazio+=d
        else: return sol


>>> D=[5,6,3,5,4,7,3]
>>> file(D,11)
[2, 6, 4]
```

La complessità dell'algoritmo è $O(n \log n)$

L'algoritmo greedy proposto è corretto

PROVA: Assumiamo per assurdo che la soluzione *sol* prodotta dal greedy non sia ottima, devono quindi esistere insiemi con più file di *sol* che rispettano la capacità del disco. Tra questi insiemi prendiamo quello con più elementi in comune con *sol* e denotiamolo con sol^*. Nota che

- esiste un file a che appartiene a sol^* e non a *sol* e che occupa più spazio di qualunque file in *sol* (questo perché tutti gli elementi in *sol* occupano meno spazio di quelli non presenti in *sol*)
- esiste un file b che appartiene a *sol* e non a sol^* (questo perche' $sol \not\subset sol^*$ infatti l'aggiunta di un qualunque elemento a *sol* porterebbe a superare la capacità del disco)

Posso dunque eliminare da sol^* il file a e inserire il file b ottenendo un nuovo insieme di file che rispetta ancora le capacità del disco ed ha un elemento in più in comune con *sol* contraddicendo la nostre ipotesi (il file sol^* è quello con più elementi in comune con *sol*)

Esercizio: Abbiamo due liste di interi A e B ciascuna con $2 \cdot n$ elementi. Dobbiamo selezionare n elementi dalla lista A ed n elementi dalla lista B in modo che la somma dei $2 \cdot n$ elementi selezionati sia massima. Non è possibile selezionare elementi che nelle due liste occupino la stessa posizione.

Ad esempio: per $A = [10, 2, 4, 6, 1, 7, 3, 4]$ e $B = [6, 6, 1, 0, 3, 8, 5, 7]$ la selezione ottima si ottiene selezionando
$A = [10, _, 4, 6, _, 7, _, _]$
$B = [_, 6, _, _, 3, _, 5, 7]$

Progettare un algoritmo che prende come parametro le due liste A e B e in tempo $O(n \log n)$ restituisce la somma massima che si può ottenere.
Motivare la correttezza e la complessità del vostro algoritmo.

E' facile vedere che scegliere gli n elementi di A a valore massimo non porta necessariamente ad una soluzione ottima:

Ad esempio:
$A = [20, 10]$ e
$B = [28, 15]$ ha come soluzione ottima

$A = [_, 10]$
$B = [28, _]$

IDEA: la scelta di un elementi $A[i]$ significa anche perdere la possibilità di scegliere $B[i]$ quindi conviene massimizzare non tanto $A[i]$ quanto $A[i] - B[i]$

Si propone quindi il seguente algoritmo:

```
def scelta(A, B):
    sol = 0
    for _ in range(n):
        sia i la posizione per cui risulta massimo A[i] − B[i]
        sol += A[i]
        estrai da A e da B gli elementi in posizione i
    for x in B:
        sol += x
    return sol
```

L'algoritmo greedy proposto è corretto

PROVA: Sia Sol^* il valore ottimo e SOL il valore prodotto dall'algoritmo.

Assumiamo per assurdo che SOL e Sol^* non coincidano e che quindi le scelte fatte dall'algoritmo e quelle che hanno portato a Sol^*differiscono. Sia Sol^* la sequenza di scelte che porta all'ottimo e che differisce meno dalle scelte fatte dall'algoritmo. Faremo vedere che esiste una sequenza di scelte SOL' che porta all'ottimo e nelle scelte fatte differisce dall'algoritmo greedy meno di quanto faccia Sol^*.

- Sia i un elemento in A scelto dal greedy ma non in Sol^* (un tale elemento deve esistere perché la soluzione ottima differisce dalla soluzione greedy)
- deve anche esistere in A un elemento j preso dalla soluzione ottima ma non da quella greedy (perché entrambe le soluzioni pescano in ciascuno degli insiemi lo stesso numero n di elementi)
- deve aversi $A[i] - B[i] \geq A[j] - B[j]$ (grazie alla regola greedy che ha portato a scegliere l'elemento i e non l'elemento j). Da cui deduciamo $A[i] - B[i] - A[j] + B[j] \geq 0$
- Considera ora la soluzione SOL' che si ottiene a partire da Sol^* scambiando la scelta fatta per i e j. Inserendo cioè gli elementi i di B al posto di quello di A e l'elemento j di B al posto di quello di A.
- la soluzione SOL' differisce meno da quella greedy di quanto faccia Sol^*. Resta da dimostrare che è ottima (vale ha dire che produce lo stesso valore di quella prodotta da Sol^*)
  - la variazione di valore tra SOL' e Sol^* è data da $A[i] - B[i] - A[j] + B[j]$ e noi sappiamo che questo valore è positivo o nullo. Positivo non può essere (avremmo altrimenti trovato una soluzione che vale più dell'ottimo) deve quindi essere nulla. Questo significa che SOL' e SOL^* hanno lo stesso valore.

Implementazione:

$$
\begin{array}{l}
\textbf{def}\ \ scelta(A,B):\\
\quad sol = 0\\
\quad \textbf{for}\ \ _\ \ in\ \ range(n):\\
\qquad sia\ \ i\ \ la\ \ posizione\ \ per\ \ cui\ \ risulta\ \ massimo\ \ A[i]-B[i]\\
\qquad sol\ +=\ \ A[i]\\
\qquad estrai\ \ da\ \ A\ \ e\ \ da\ \ B\ \ gli\ \ elementi\ \ in\ \ posizione\ \ i\\
\quad \textbf{for}\ \ x\ \ in\ \ B:\\
\qquad sol\ +=\ \ x\\
\quad \textbf{return}\ \ sol
\end{array}
$$

IDEA: una implementazione diretta produrrebbe una complessità $O(n^2)$(infatti costerebbe $O(n)$ la ricerca dell'elemento i per cui è massimo $A[i] - B[i]$) e questa operazione andrebbe fatta n volte all'interno del for. Per ottenere $O(n \log n)$ effettuo un pre-processing riordinando gli elementi di A in modo che risulti decrescente il valore $A[i] - B[i]$. Nota che fatto questo ordinamento gli elementi da selezionare in A saranno gli n che risultano al primo posto della lista.

```
def scelta(A,B):
    lista=sorted([ (A[i]-B[i],i) for i in range(len(A))],reverse=True)
    sol=0
    for i in range(len(lista)//2):
        sol+=A[lista[i][1]]
    for i in range(len(lista)//2,len(lista)):
        sol+=B[lista[i][1]]
    return sol

>>> A=[10,2,4,6,1,7,3,4]
>>> B=[ 6,6,1,0,3,8,5,7]
>>> scelta(A,B)
48
```

La complessità è $O(n \log n)$

ESERCIZIO: abbiamo una lista con le posizioni $p_0 < p_2 < \ldots < p_{n-1}$ in cui si trovano n case lungo una strada rettilinea. Lungo la strada bisogna posizionare dei cassonetti e si vuole che ciascuna casa abbia almeno un cassonetto ad una distanza che non superi k.

Ad esempio:
se $lista = [2, 5, 7, 11, 14, 16, 18]$ e $k = 3$ la soluzione ottima richiede 3 cassonetti, una possibile dislocazione è $[4, 11, 15]$

Progettare un algoritmo $greedy$ che, data la lista con le n posizioni e k, restituisca il minimo numero di cassonetti necessari a soddisfare il requisito e una lista con le posizioni in cui localizzarli. La complessità dell'algoritmo deve essere $O(n)$. Motivare correttezza e complessità del vostro algoritmo.

Un algoritmo $greedy$ piuttosto naturale prende in esame le case una dopo l'altra e se una casa in posizione p non ha un cassonetto a distanza al più k alla sua sinistra allora viene posizionato un cassonetto in posizione $p + k$. In pratica il cassonetto viene posizionato in modo che risulti il più a destra possibile ma ancora in grado di soddisfare la casa in esame.

$$
\begin{array}{l}
\textbf{def}\ \ cassonetto(lista,k):\\
\quad tot,\ sol\ \ =\ \ 0,\ [\ ]\\
\quad \textbf{for}\ \ p\ \ in\ \ lista:\\
\qquad \textbf{if}\ \ p\ \ dista\ \ piu'\ \ di\ \ k\ \ dalle\ \ posizioni\ \ in\ \ sol:\\
\qquad sol.append(p+k)\\
\qquad tot+=1\\
\quad \textbf{return}\ \ tot,\ sol
\end{array}
$$

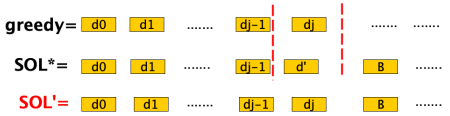
IDEA: per questioni di efficienza tengo traccia in una variabile x della posizione del cassonetto localizzato più a destra

```
def cassonetto(lista,k):
    tot,x,sol=0,-(k+1),[]
    for p in lista:
        if p> x+k:
            sol.append(p+k)
            tot+=1
            x=p+k
    return tot, sol

>>> lista1=[2,5,7,11,14,16,18]
>>> cassonetto(lista1,3)
3,[5, 14, 21]
```

La complessità è $O(n)$

L'algoritmo greedy proposto è corretto

PROVA: Sia tot^* il numero minimo di cassonetti necessari a soddisfare il vincolo, tot è invece il numero di cassonetti prodotti dall'algoritmo *greedy* e $d_0, d_1 \ldots$ le localizzazioni di questi tot cassonetti.
Assumiamo per assurdo che $tot > tot^*$ e che quindi le scelte fatte dall'algoritmo e quelle che portano all'ottimo tot^* differiscano. Sia SOL^* la sequenza di scelte che porta all'ottimo e che coincide col greedy nel numero massimo di localizzazioni iniziali. Faremo vedere che esiste una sequenza di scelte SOL' che porta all'ottimo e nelle scelte fatte coincide con quelle del *greedy* in una scelta in più di quanto faccia SOL^*.

- Sia d_j la prima localizzazione di cassonetto da sinistra scelta dal *greedy* ma non in SOL^* (una tale localizzazione deve esistere perché abbiamo assunto che la soluzione ottima differisce dalla soluzione greedy). Per la scelta greedy dell'algoritmo deve aversi $d_j = p_i + k$ per una qualche posizione di casa p_i che i cassonetti localizzati in $d_0, \ldots d_{j-1}$ non erano in grado di soddisfare.
- deve anche esistere in SOL^* un cassonetto alternativo localizzato in d' con $d' < d_j$ (in caso contrario la casa in posizione p_i non risulterebbe soddisfatta.
- Considera ora la soluzione SOL' che si ottiene a partire da SOL^* spostando il cassonetto localizzato in d' verso destra in posizione d_j. Nota che lo spostamento di cassonetto verso destra lascia ancora tutte le case soddisfatte: le case nelle posizioni da p_0 a p_{i-1} resteranno soddisfatte dai cassonetti localizzati in d_0 fino a $d_j - 1$ e le case con posizione $p \geq p_i$ saranno soddisfatte dai cassonetti localizzati da d_j in poi.
- la soluzione SOL' ha le in comune col *greedy* le prime j scelte, la soluzione SOL^* coincideva col greedy solo nelle prime $j-1$ scelte.