Algoritmi greedy

Progettazione di Algoritmi a.a. 2018-19
Matricole congrue a 1
Docente: Annalisa De Bonis

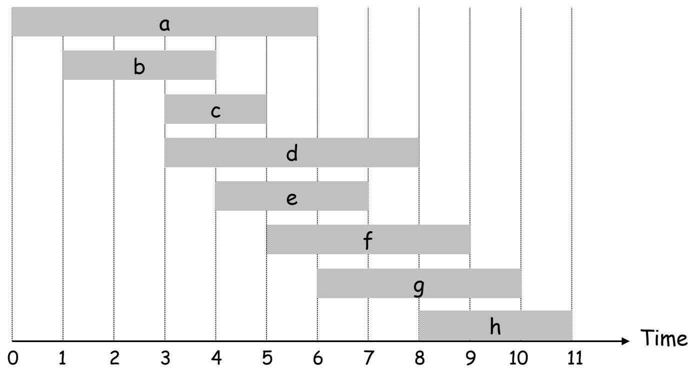
Scelta greedy

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 citta` a alla citta` 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 percorrrere 450 km e che alla partenza il serbatoio e` vuoto.
Strategia greedy: faccio il pieno in a e arrivo al piu` lontano punto di rifornimento che riesco a raggiungere (distanza da a ≤ 450 km). Da questo punto raggiungo il punto di rifornimento piu` lontano che si trova ad al piu` 450 km da esso, e cosi` via fino a che non arrivo in b. Si puo` dimostrare che questa strategia fornisce la soluzione ottima.

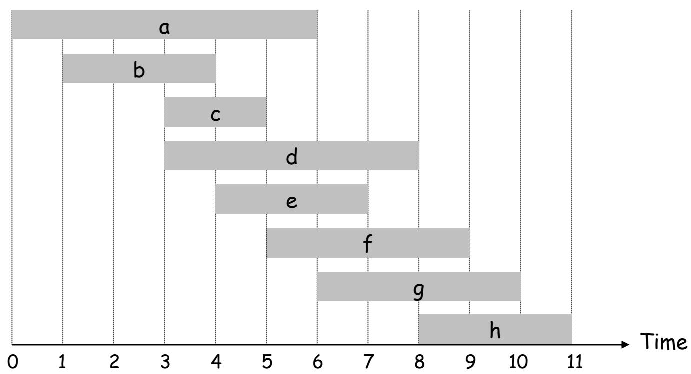
Interval Scheduling

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.



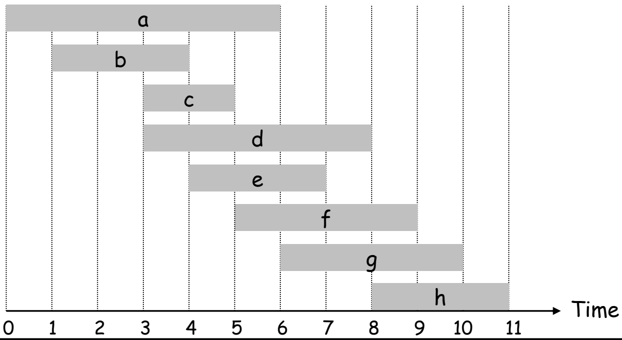
Interval Scheduling

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



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.

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.

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.

PROGETTAZIONE DI ALGORITMI A.A. 2018-19
A.DE BONIS

9

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 $s_j \geq f$

10

Interval Scheduling: Ottimalità soluzione greedy

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

Dim.

- Denotiamo con i_1, i_2, ... 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, ... j_m l'insieme di job nella soluzione ottima, disposti in ordine non decrescente rispetto ai tempi di fine.

1. Dimostriamo prima l'esecuzione dei job i_1, i_2, ... i_k termina non più tardi di quella dei job j_1, j_2, ... 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.

Continua nella prossima slide

Interval Scheduling: Ottimalità soluzione greedy

Dimostriamo il punto 1.:

- Dimostriamo per induzione che per ogni indice r, $1 \leqq r \leqq k$, 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. Supponiamo per ipotesi induttiva che per $r-1 \geqq 1$, il tempo di fine di i_{r-1} sia **non** più grande di quello di j_{r-1}. Di conseguenza il job j_r è compatibile con i_{r-1}. Poichè i tempi di fine di i_1, i_2, ... i_{r-2} sono non più grandi di quello di i_{r-1} allora j_r è compatibile con i_1, i_2, ... i_{r-2} i_{r-1}. Poichè all'r-esimo passo l'algoritmo greedy sceglie i_r, vuol dire che i_r finisce non più tardi di tutti i job compatibili con i_1, i_2, ... i_{r-1} e quindi anche non più tardi di j_r.

Continua nella prossima slide

Interval Scheduling: Ottimalità soluzione greedy

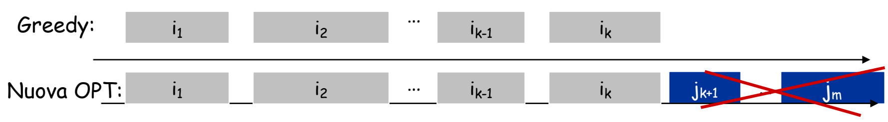
- Abbiamo dimostrato che, per ogni indice $r \leq k$, il tempo di fine di i_r è **non** più grande di quello di j_r.
- Di conseguenza il tempo di fine di i_k è non più grande di quello di j_k.
- Dal momento che $i_1, i_2, \ldots i_k$ sono ordinati in base ai tempi di fine allora si ha che anche i tempi di fine di $i_1, i_2, \ldots i_{k-1}$ sono **non** maggiori di quello di j_k.
- Quindi l'esecuzione della sequenza di job $i_1, i_2, \ldots i_k$ termina non più tardi di j_k e quindi possiamo affermare che l'esecuzione della sequenza di job $i_1, i_2, \ldots i_k$ termina non piu` tardi di quella dei job $j_1, j_2, \ldots j_k$

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 $j_1, j_2, \ldots j_m$ conterra` il job j_{k+1}
- Per il punto 1, l'esecuzione di $i_1, i_2, \ldots i_k$ termina **non** più tardi dell'esecuzione di $j_1, j_2, \ldots j_k$. e si ha quindi che $i_1, i_2, \ldots i_k$ sono compatibili con j_{k+1}
- L'algoritmo greedy, dopo aver inserito $i_1, i_2, \ldots i_k$ nella soluzione, passa ad esaminare i job con tempo di fine maggiore o uguale a quelli di $i_1, i_2, \ldots i_k$. Tra questi job vi è j_{k+1} che è compatibile con $i_1, i_2, \ldots i_k$ per cui è impossibile che l'algoritmo greedy inserisca nella soluzione solo k job. Siamo giunti ad una contraddizione e quindi e` imposibile 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 piu` 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

15

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 attivita` necessita di un'unica risorsa.
- Una risorsa puo` essere allocata ad al piu` un'attivita` alla volta

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

Un'istanza del problema consiste in un insieme di n intervalli $[s_1,f_1],\ldots,[s_n,f_n]$ che rappresentano gli intervalli durante i quali si svolgono le n attivita`.

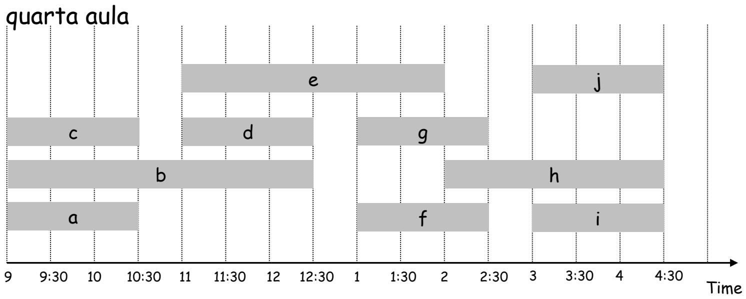
Obiettivo: far svolgere le n attivita` utilizzando il minor numero possible di risorse e in modo che ciascuna risorsa utilizza venga allocata ad al piu` un'attivita` alla volta.

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.

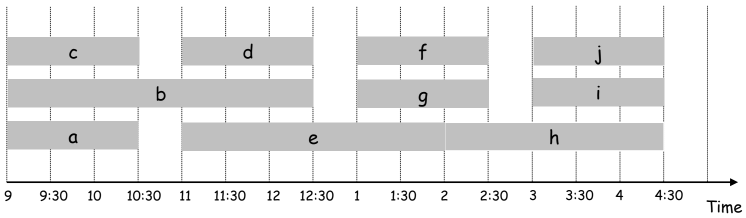
Esempio: Questo schedule usa 4 aule (una per livello) per 10 lezioni: e, j nella prima aula; c,d,g nella seconda aula; b, h nella terza aula; a, f, i nella quarta aula



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.



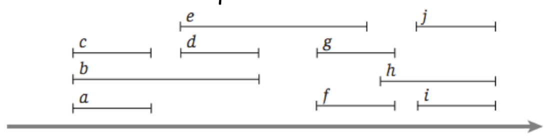
Partizionamento di intervalli: limite inferiore alla soluzione ottima

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

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

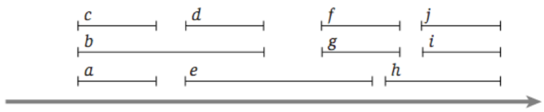
Esempio. L'insieme di intervalli in figura (a) ha profondità 3. Lo schedule in figura (b) usa 3 risorse ed è quindi ottimo

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.

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

```
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤ sn.
d ← 0

for j = 1 to n {
   if (interval j can be assigned an already allocated resources v)
      assign resource v to interval j
   else
      allocate a new resource d + 1
      assign the new resource d+1 to interval j
      d ← d + 1
}
```

- 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 profondita 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 attivita` 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 attivita` 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}

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}.
- Abbiamo osservato 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}.

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 $s_j > k_v$ 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 ogni operazione sulla coda richiede 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(nlog n). Siccome m≤n il costo dell'algoritmo è O(nlogn)

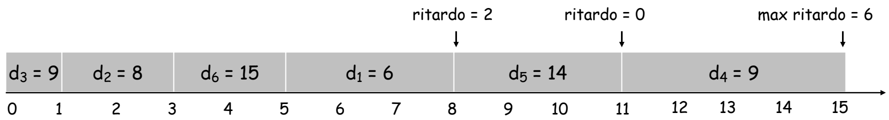
Scheduling per Minimizzare i Ritardi

Problema della minimizzazione dei ritardi.

- Una singola risorsa in grado di elaborare un unico job.
- Il job j richiede t_j unità di tempo e deve essere terminato entro il tempo d_j (scadenza).
- Se j comincia al tempo s_j allora finisce al tempo $f_j = s_j + t_j$.
- Def. Ritardo è definito come $\ell_j = \max\{0, f_j - d_j\}$.
- Obiettivo: trovare uno scheduling di tutti i job che minimizzi il ritardo massimo $L = \max \ell_j$.

Esempio:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Minimizzare il ritardo: Algoritmo Greedy

Schema greedy. Considera i job in un certo ordine.

- [Shortest processing time first] Considera i job in ordine non decrescente dei tempi di elaborazione t_j.
- [Earliest deadline first] Considera i job in ordine non decrescente dei tempi entro i quali devono essere ultimati d_j.
- [Smallest slack] Considera i job in ordine non decrescente degli scarti d_j - t_j.

Minimizzare il ritardo: Algoritmo Greedy

- [Shortest processing time first] Considera i job in ordine non decrescente dei tempi di elaborazione t_j.

	1	2
t_j	1	10
d_j	100	10

controesempio

Viene eseguito prima il job 1. Ritardo massimo è 11-10=1. Se avessimo eseguito prima il job 2 avremmo avuto ℓ_1 = max{0, 10-10}=0 e ℓ_2 = max{0,11-100}=0 per cui il ritardo massimo sarebbe stato 0.

- [Smallest slack] Considera i job in ordine non decrescente degli scarti d_j – t_j.

	1	2
t_j	1	10
d_j	2	10

controesempio

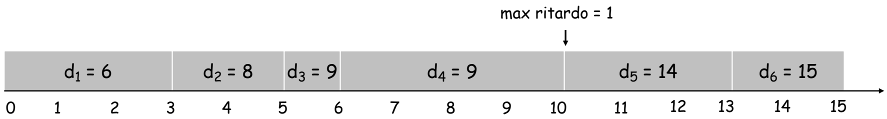
Viene eseguito prima il job 2. Ritardo massimo è 11-2=9. Se avessimo eseguito prima il job 1 il ritardo massimo sarebbe stato 11-10=1

Minimizzare il ritardo: Algoritmo Greedy

Algoritmo greedy. Earliest deadline first: Considera i job in ordine non decrescente dei tempi d_j entro i quali devono essere ultimati.

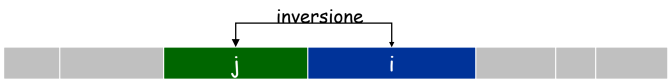
```
Sort n jobs by deadline so that d1 ≤ d2 ≤ … ≤ dn

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + tj]
   sj ← t, fj ← t + tj
   t ← t + tj
output intervals [sj, fj]
```



Minimizzare il ritardo: Inversioni

Def. Un' inversione in uno scheduling S è una coppia di job i e j tali che: $d_i < d_j$ ma j viene eseguito prima di i.



Ottimalità soluzione greedy

La dimostrazione dell'ottimalità si basa sulle seguenti osservazioni che andremo poi a dimostrare

1. La soluzione greedy ha le seguenti due proprietà:

a. Nessun idle time. Non ci sono momenti in cui la risorsa non è utilizzata
b. Nessuna inversione. Se un job j ha scadenza maggiore di quella di un job i allora viene eseguito dopo i

2. Tutte le soluzioni che hanno in comune con la soluzione greedy le caratteristiche a e b, hanno lo stesso ritardo massimo della soluzione greedy.

3. Ogni soluzione ottima può essere trasformata in un'altra soluzione ottima per cui valgono la a e la b

Dimostrazioni delle osservazioni 1. 2. e 3.

1. La soluzione greedy ha le seguenti due proprietà:

a. Nessun idle time. Non ci sono momenti in cui la risorsa non è utilizzata
b. Nessuna inversione. Se un job j ha scadenza maggiore di quella di un job i allora viene eseguito dopo i

Dim.

Il punto a discende dal fatto che ciascun job comincia nello stesso istante in cui finisce quello precedente.

Il punto b discende dal fatto che i job sono esaminati in base all'ordine non decrescente delle scadenze.

Dimostrazioni delle osservazioni 1. 2. e 3.

Prima di dimostrare il punto 2 consideriamo i seguenti fatti

Fatto I. In uno scheduling con le caratteristiche a e b i job con una stessa scadenza d sono disposti uno di seguito all'altro.

Dim.

Consideriamo i e j con $d_i=d_j=d$ e assumiamo senza perdere di generalità (da ora in poi s.p.d.g.) che i venga eseguito prima di j .

Supponiamo **per assurdo** che tra i e j venga eseguito il job q con $d \neq d_q$.

Se $d < d_q$ allora la coppia j, q è un'inversione . Se $d > d_q$ allora la coppia i, q è un'inversione. Ciò contraddice la proprietà b.

Ne consegue che tra due job con una stessa scadenza d non vengono eseguiti job con scadenza diversa da d e poiché lo scheduling non ha idle time, i job con una stessa scadenza vengono eseguiti uno di seguito all'altro.

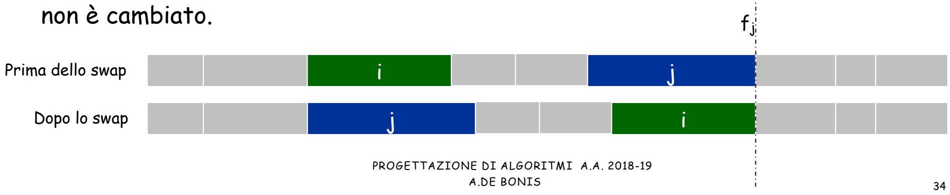
Dimostrazioni delle osservazioni 1. 2. e 3.

Fatto II. Se in uno scheduling con le caratteristiche a e b scambiamo due job con la stessa scadenza, il ritardo massimo non cambia.

Dim.

Consideriamo due job i e j con $d_i=d_j$ e supponiamo s.p.d.g. che i preceda j in S.
Per il fatto I, tra i e j vengono eseguiti solo job con la stessa scadenza di i e j.
Ovviamente il ritardo di j è maggiore o uguale del ritardo di i e dei ritardi di tutti i job eseguiti tra i e j perché j finisce dopo tutti questi job e ha la loro stessa scadenza.

Se scambiamo i con j in S otteniamo che il ritardo di j non può essere aumentato mentre quello di i è diventato uguale a quello che aveva prima j in quanto i finisce nello stesso istante in cui finiva prima j e la scadenza di i è la stessa di j. Il ritardo dei job compresi tra i e j potrebbe essere aumentato ma non può superare il ritardo che aveva prima j. Di conseguenza i ritardo massimo non è cambiato.



Dimostrazioni delle osservazioni 1. 2. e 3.

2. Tutte le soluzioni che hanno in comune con la soluzione greedy le caratteristiche a e b, hanno lo stesso ritardo massimo della soluzione greedy

Dim.

Dimostriamo che dati due scheduling S ed S' di n job entrambi aventi le caratteristiche a e b, S può essere trasformato in S' senza che il suo ritardo massimo risulti modificato.

Osserviamo che S ed S' possono differire solo per il modo in cui sono disposti tra di loro job con la stessa scadenza altrimenti o S o S' conterrebbero un'inversione.

Di conseguenza S può essere trasformato in S' scambiando tra di loro di posto coppie di job con la stessa scadenza.

Per il fatto II, scambiando coppie di job con la stessa scadenza il ritardo max non cambia. Di conseguenza possiamo trasformare S in S' senza che cambi il ritardo max. In altre parole S ed S' hanno lo stesso ritardo max. Prendendo S uguale ad un qualsiasi scheduling con le caratteristiche a e b ed S' uguale allo scheduling greedy si ottiene la tesi.

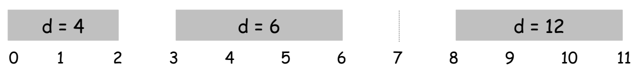
Dimostrazioni delle osservazioni 1. 2. e 3.

Prima di dimostrare il punto 3 consideriamo i seguenti fatti.

Fatto III. Una soluzione ottima può essere trasformata in una soluzione soluzione con nessun tempo di inattività (idle time).

Dim. Se tra il momento in cui finisce l'elaborazione di un job e quello in cui inizia il successivo vi è un gap, basta shiftare all'indietro l'inizio del job successivo in modo che cominci non appena finisce il precedente. Ovviamenti i ritardi dei job non aumentano dopo ogni shift

Esempio: Soluzione ottima con idle time



Esempio: Soluzione ottima con nessun idle time

.

Dimostrazioni delle osservazioni 1. 2. e 3.

Fatto IV. Se uno scheduling privo di idle time ha un'inversione allora esso ha una coppia di job invertiti che cominciano uno dopo l'altro.

Dim.

- Consideriamo tutte le coppie di job i e j tali che $d_i < d_j$ e j viene eseguito prima di i nello scheduling. S
- Supponiamo per assurdo tutte le coppie di questo tipo siano separate da un job.
- Tra tutte le coppie che formano un'inversione prendiamo la coppia (i,j) piu` vicina nello scheduling.
- Per l'ipotesi assurda, deve esistere un job k≠i eseguito subito dopo j che non forma un'inversione né con i né con j altrimenti i e j non formerebbero l'inversione piu` vicina. Deve quindi essere $d_j \leq d_k$ e $d_k \leq d_i$. Le due diseguaglianze implicano $d_j \leq d_i$ il che contraddice il fatto che la coppia i, j e un'inversione.

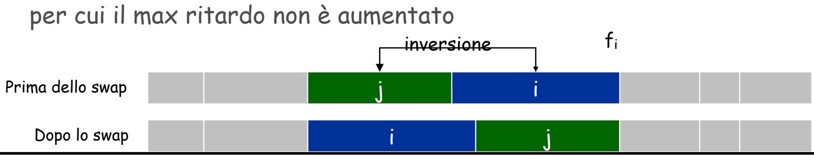
Dimostrazioni delle osservazioni 1. 2. e 3.

Fatto V. Scambiare due job adiacenti invertiti i e j riduce il numero totale di inversioni di uno e non fa aumentare il ritardo massimo.
Dim. Supponiamo $d_i < d_j$ e che j precede i nello scheduling.
Siano $\ell_1,\ldots,\ell_n$ i ritardi degli n job e siano $\ell'_1,\ldots,\ell'_n$ i ritardi degli n job dopo aver scambiato i e j di posto. Si ha che

- $\ell'_k = \ell_k$ per tutti i $k \neq i, j$
- $\ell'_i \leq \ell_i$ perchè viene anticipata la sua esecuzione.
- Vediamo se il ritardo di j è aumentato al punto da far aumentare il ritardo max. Ci basta considerare il caso in cui $\ell'_j > 0$ altrimenti vuol dire che il ritardo di j non è aumentato. Si ha quindi

$$\begin{aligned} \ell'_j &= f'_j - d_j && \text{(per la definizione di ritardo)} \\ &= f_i - d_j && \text{(dopo lo swap, j finisce al tempo } f_i\text{)} \\ &< f_i - d_i && \text{(in quanto } d_i < d_j\text{)} \\ &\leq \ell_i && \text{(per la definizione di ritardo)} \end{aligned}$$

per cui il max ritardo non è aumentato



Dimostrazioni delle osservazioni 1. 2. e 3.

- 3. Ogni soluzione ottima D può essere trasformata in un'altra soluzione ottima per cui valgono la a e la b

- Dim.
- Il fatto III implica che la soluzione ottima S può essere trasformata in una soluzione ottima per cui non ci sono idle time.

- Il fatto IV implica che se la soluzione ottima S contiene inversioni allora S contiene una coppia di job **adiacenti** invertiti. Il fatto V implica che se scambiamo le posizioni di questi due job otteniamo ancora una soluzione ottima con un numero inferiore di inversioni. Quindi possiamo scambiare di posto coppie di job adiacenti invertiti fino a che non ci sono più inversioni nella soluzione ottima.