Algoritmi greedy

Progettazione di Algoritmi a.a. 2015-16

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.

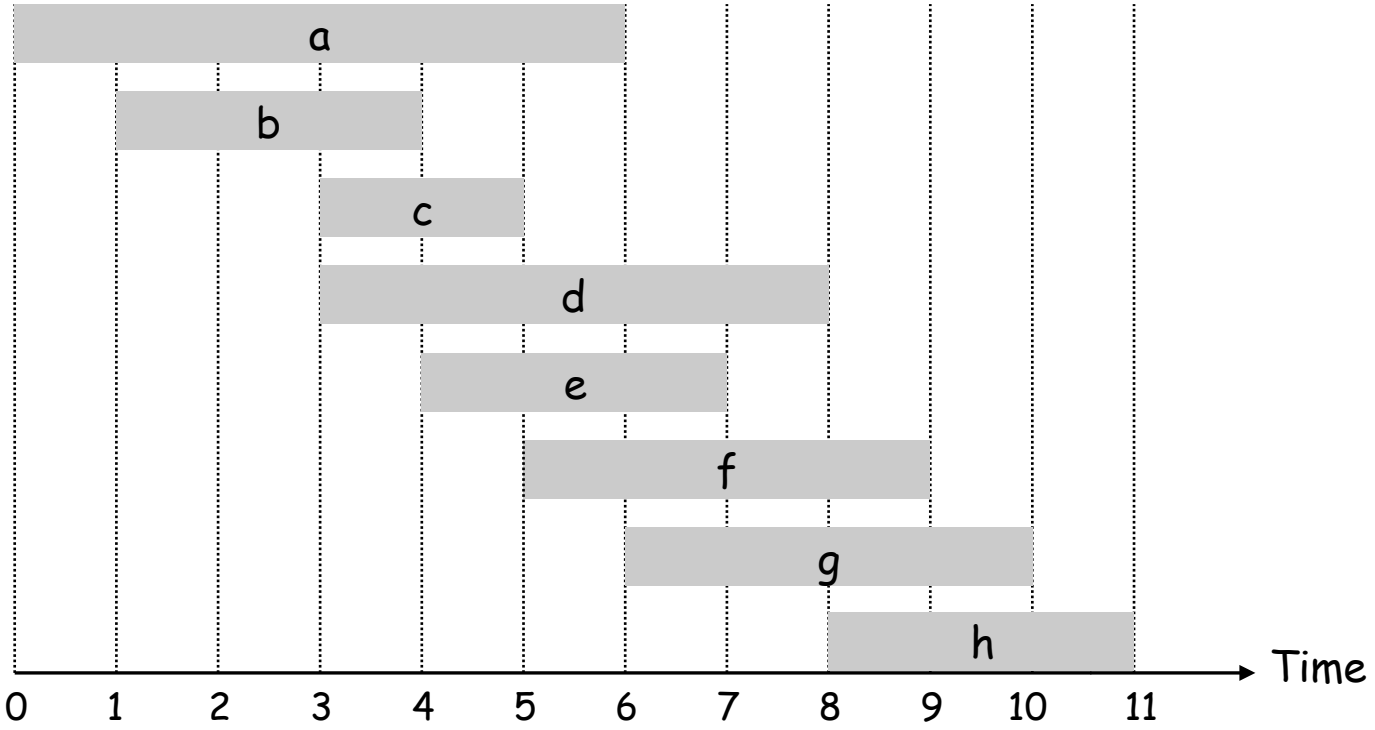
Domanda. Questo approccio porta sempre ad una soluzione ottima?

Risposta. Non sempre ma per molti problemi sì.

Interval Scheduling

Interval scheduling.

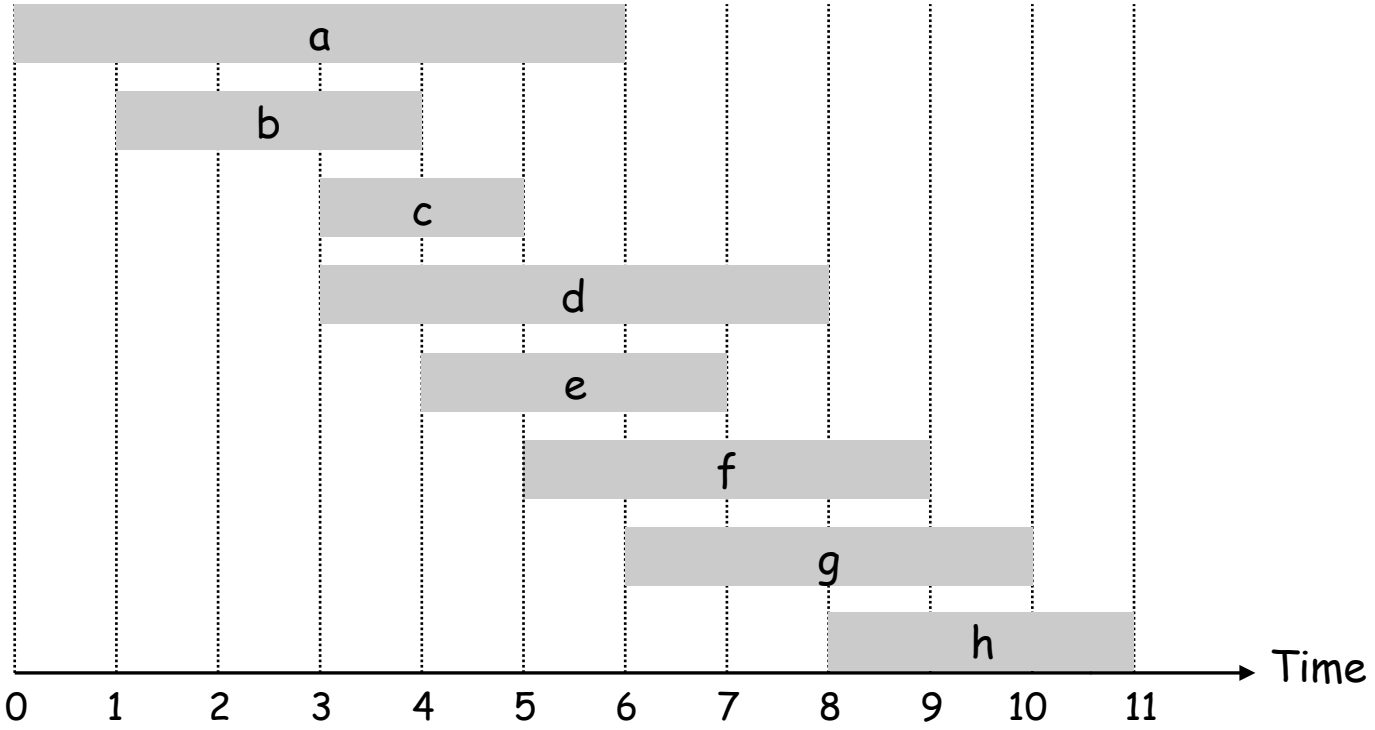
- Input: un insieme di attività ognuna delle quali inizia ad un certo instante s_j e finisce ad un certo istante f_j. Può essere eseguita al più un'attivita alla volta.
- Obiettivo: fare in modo che vengano svolte quante più attività è possibile.



Interval Scheduling

Interval scheduling.

- Il job j comincia nell'instante s_j e finisce all'istante f_j.
- Due job sono compatibili se non si sovrappongono
- Obiettivo: trovare un sottoinsieme di cardinalità massima di job a due a due compatibili.



Interval Scheduling

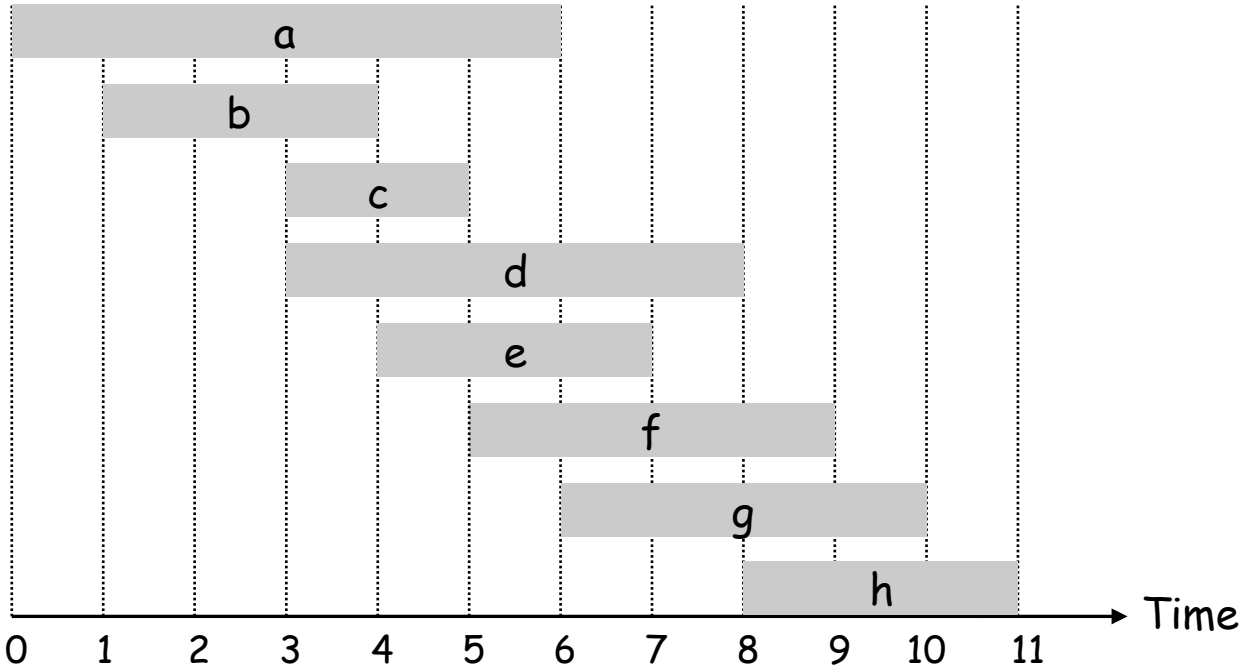
Se all'inizio scegliamo a poi possiamo scegliere o g o h.

- 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 possiamo non possiamo scegliere nessun altro job. Totale = 2

Se all'inizio scegliamo uno tra f, g e h, non possiamo selezionare nessun altro job. Totale = 1



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 questo accade 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 f_1 ≤ f_2 ≤ ... ≤ f_n.

A ← φ
for j = 1 to n {
   if (job j is compatible with A)
      A ← A ∪ {j}
}
return A
```

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

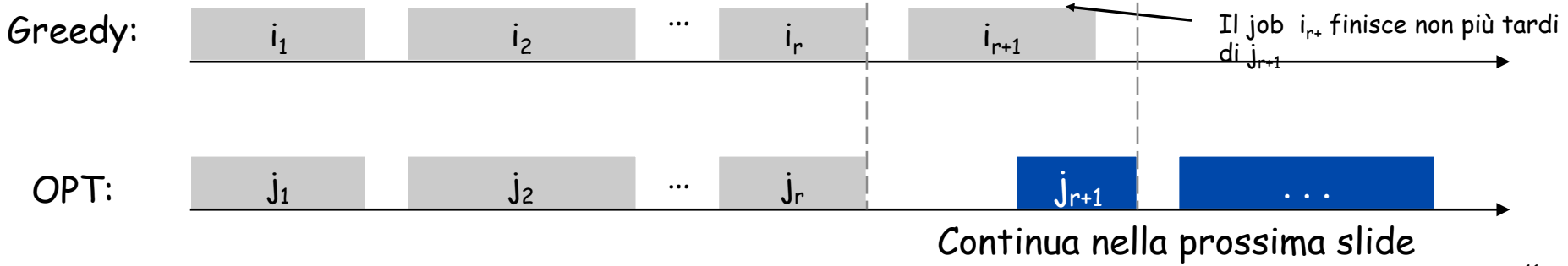
- Costo ordinamento $O(n \log n)$
- Costo for $O(n)$: se si mantiene traccia del tempo di fine f dell'ultimo job selezionato, per capire se il job j è compatibile con A basta verificare che $s_j \geq f$

Interval Scheduling: Ottimalità soluzione greedy

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

Dim. (per assurdo)

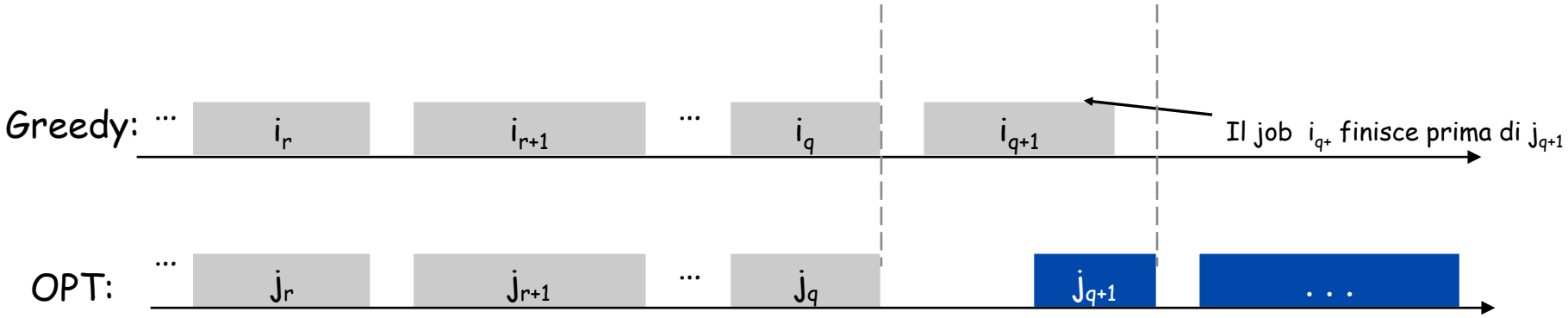
- Supponiamo che la tesi non sia vera, cioè che l'algoritmo non sia ottimo.
- Denotiamo con $i_1, i_2, \ldots i_k$ l'insieme di job selezionati dall'algoritmo nell'ordine in cui sono selezionati, cioè in ordine non decrescente rispetto ai tempi di fine.
- Denotiamo con $j_1, j_2, \ldots j_m$ l'insieme di job nella soluzione ottima, disposti in ordine non decrescente rispetto ai tempi di fine.
- Sia r il più grande indice per cui $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$. In altre parole la soluzione ottima e quella ottenuta dall'algoritmo coincidono nei primi r job e differiscono per la prima volta nell'(r+1)-esimo
- Poichè all'(r+1)-esimo passo l'algoritmo greedy sceglie i_{r+1} ,si ha che i_{r+1} finisce non più tardi di j_{r+1}. Quindi i_{r+1} è compatibile con gli altri job nella soluzione ottima e posso sostituire j_{r+1} con i_{r+1} nella soluzione ottima.



Continua nella prossima slide

Interval Scheduling: Ottimalità soluzione greedy

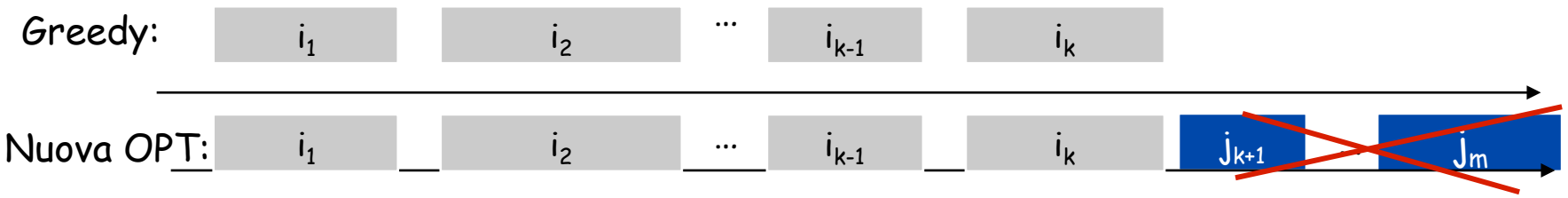
- La soluzione ottima con cui confrontiamo quella del nostro algoritmo ora è $i_1, i_2, \ldots i_{r+1}, j_{r+2}, \ldots, j_m$
- Sia ora q (q>r+1) il più grande indice tale che $i_{r+2}=j_{r+2}$, $i_{r+3}= j_{r+3}$, ... ,$i_q=j_q$. Si ha che i_{q+1} finisce non più tardi di j_{q+1} e quindi posso sostituire j_{q+1} con i_{q+1}.
- Posso continuare in questo modo fino a che la sequenza $j_1, j_2, \ldots j_k$ è stata trasformata in $i_1, i_2, \ldots i_k$.



Continua nella prossima slide

Interval Scheduling: Ottimalità soluzione greedy

- Siccome stiamo supponendo che $i_1, i_2, \ldots i_k$ non è ottima allora deve essere k<m.
- Di conseguenza, la soluzione ottima $i_1, i_2, \ldots i_k\, j_{k+1}, \ldots j_m$ contiene m-k intervalli $j_{k+1}, j_{k+2}, \ldots j_m$ che non fanno parte della soluzione dell'algoritmo greedy.
- Questi m-k job sono sicuramente compatibili con $i_1, i_2, \ldots i_k$ perchè fanno della soluzione ottima insieme a $i_1, i_2, \ldots i_k$.
- Inizialmente abbiamo assunto che gli intervalli $j_1, j_2, \ldots j_m$ nella soluzione ottima fossero ordinati in base ai tempi di fine per cui $j_{k+1}, \ldots j_m$ hanno tempi di fine maggiori o uguali dei precedenti $j_1, \ldots j_k$
- Per passare dalla soluzione ottima $j_1, j_2, \ldots j_m$ alla soluzione ottima $i_1, i_2, \ldots i_k, j_{+1}, \ldots j_m$ abbiamo sostituito alcuni dei $j_1, j_2, \ldots j_k$ con intervalli della soluzione greedy con tempi di fine minori o uguali di quelli degli intervalli sostituiti.
- Ne consegue che $j_{k+1}, \ldots j_m$ hanno tempi di fine maggiori o uguali di $i_1, i_2, \ldots i_k$.
- Se così fosse, l'algoritmo greedy dopo aver selezionato $i_1, i_2, \ldots i_k$ avrebbe esaminato $j_{k+1}, \ldots, j_m$ e li avrebbe selezionati. Abbiamo quindi contraddetto che k<m.



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.

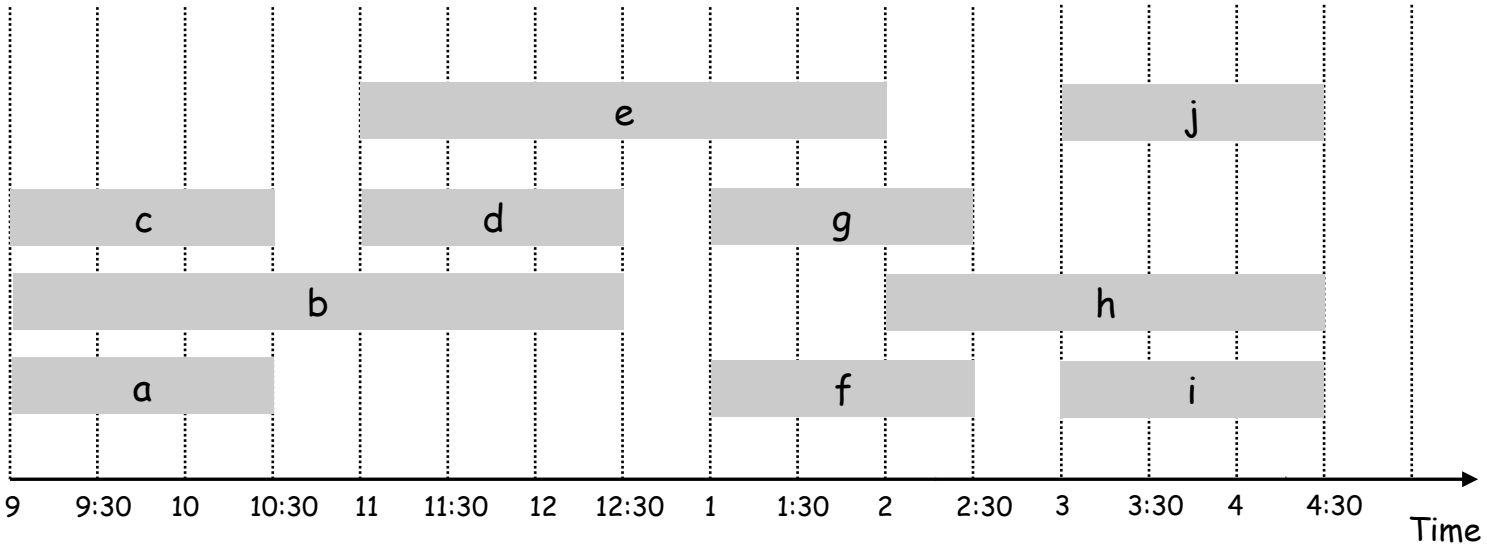
Nell' interval scheduling avevamo un'unica risorsa e volevamo che venissero svolte il massimo numero di attività

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 contemporaneamte nella stessa aula.

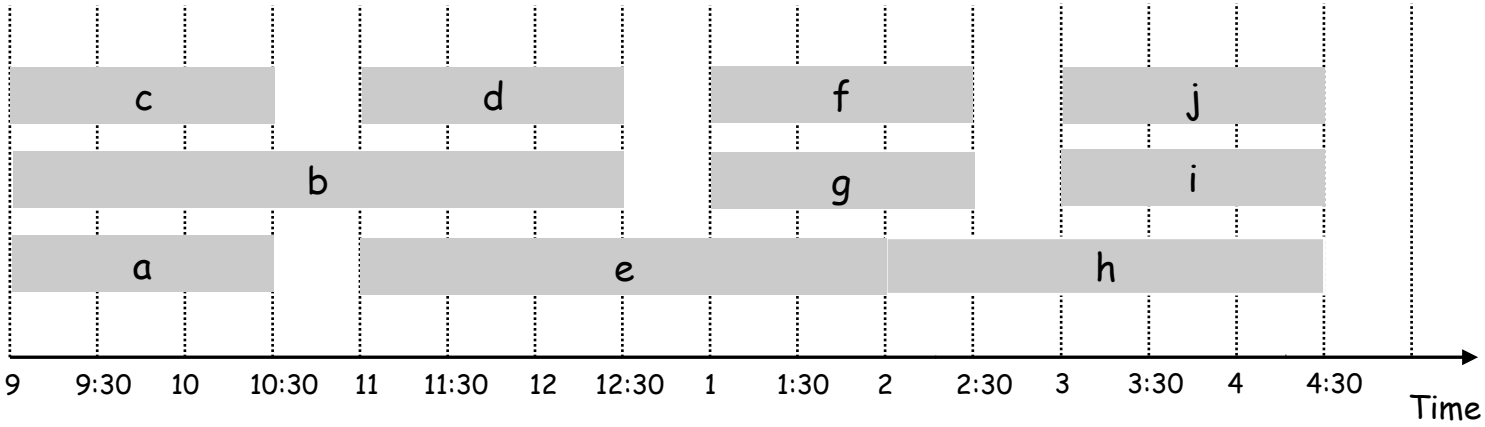
Esempio: Questo schedule usa 4 aule per 10 lezioni. (e, j nella stessa aula; c,d,g nella stessa aula; b, h nella stessa aula; a, f, i nella stessa aula)



Partizionamento di intervalli

Esempio. Questo schedule usa solo 3 aule: {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.

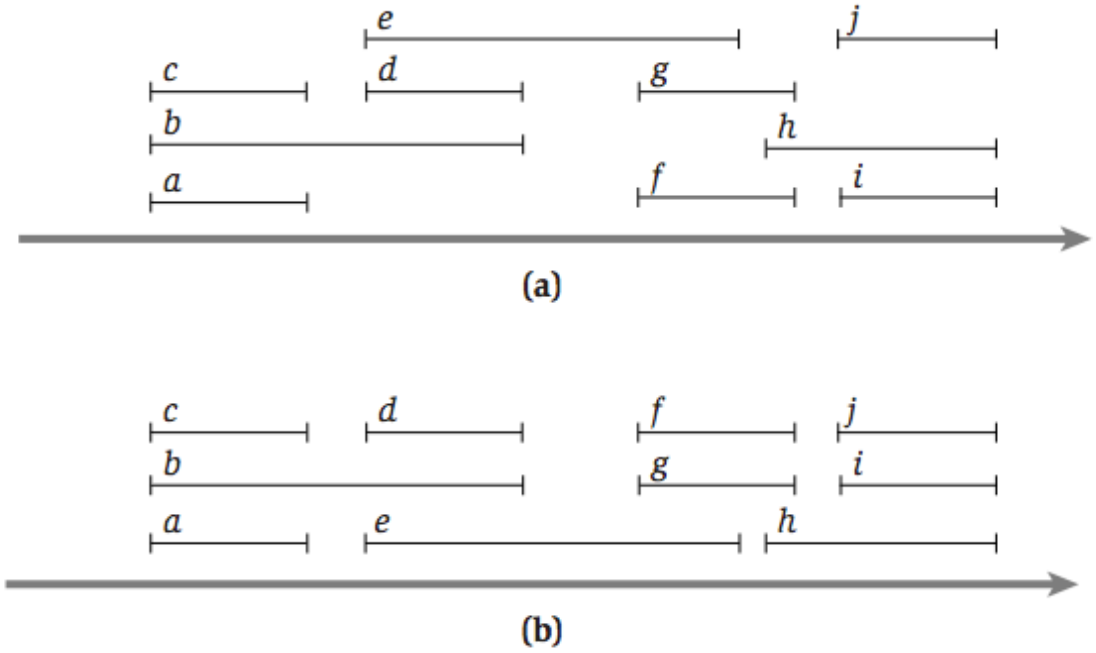


Partizionamento di intervalli: limite inferiore alla soluzione ottima

Def. La profondità di un insieme di intervalli è il numero massimo di intervalli intersecabili con una singola linea verticale

Osservazione. Numero di classi necessarie ≥ profondità.

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



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
}
```

Partizionamento di intervalli: Ottimalità soluzione greedy

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

Teorema. L'algoritmo greedy usa esattamente un numero di risorse pari alla profondità dell'insieme di intervalli.

Dim.

- Supponiamo che alla j-esima iterazione del ciclo di for d venga incrementato.
- La risorsa d è stata allocata perchè ciascuna delle altre risorse già allocate è assegnata al tempo s_j ad un intervallo incompatibile con l'intervallo j.
- Siccome l'algoritmo considera gli intervalli in ordine non decrescente dei tempi di inizio, tutti gli intervalli che generano incompatibilità iniziano non più tardi di j.
- Di conseguenza, ci sono d intervalli (uno per risorsa allocata) che si sovrappongono al tempo s_j.
- Ciò è vero ogni volta che viene allocata una nuova risorsa per cui se m è il numero totale di risorse allocate vuol dire che in un certo istante si sovrappongono m intervalli per cui m ≤ profondità. Il minore non può valere per il limite inferiore m ≥ profondità per cui m=profondità.

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. 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 $m \leq n$ il costo dell'algoritmo è $O(n \log n)$

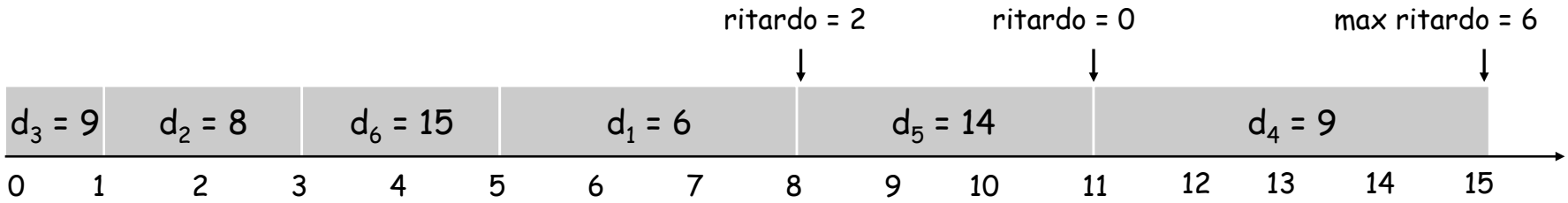
Scheduling per Minimizzare i Ritardi

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 ritarno 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 il ritardo massimo sarebbe stato max{0, 11-100}=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

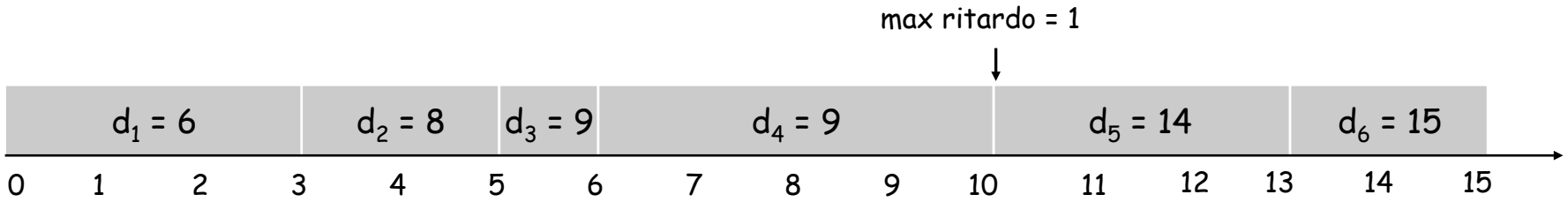
Viene eseguito prima il job 2. Ritardo massimo è 11-2=9. Se avessimo eseguito prima il job 1 il ritardo massimo sarebbe stato 11-10=1

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.

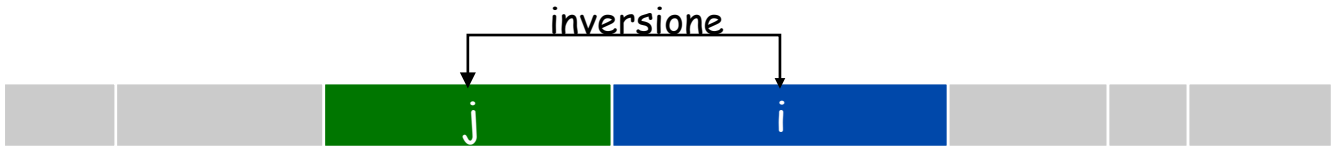
```
Sort n jobs by deadline so that d_1 ≤ d_2 ≤ … ≤ d_n

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + t_j]
   s_j ← t, f_j ← t + t_j
   t ← t + t_j
output intervals [s_j, f_j]
```



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 1. 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 $d_i \neq d_q$.

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.

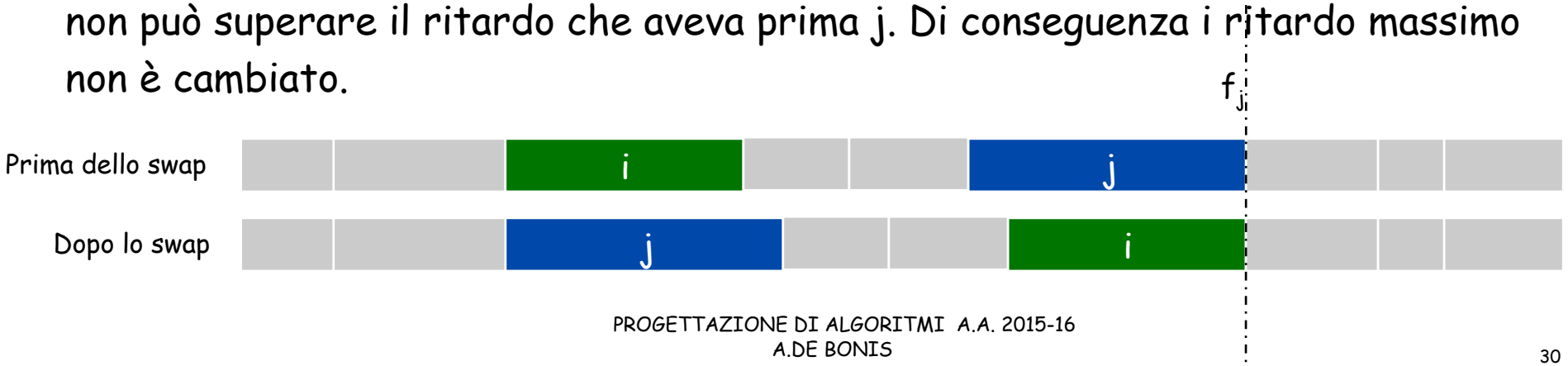
Ne consegue che tra due job con una stessa scadenza d non vengono eseguiti job con scadenza diversa da d e poiché lo scheduling non ha idle time, i job con una stessa scadenza vengono eseguiti uno di seguito all'altro.

Dimostrazioni delle osservazioni 1. 2. e 3.

Fatto 2. 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 1, 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. Infatti se un job in S dovesse essere scambiato con un job con scadenza diversa dalla sua, si avrebbe un'inversione.

Si ha quindi che S ed S' possono differire solo per il modo in cui sono ordinati tra di loro job con la stessa scadenza.

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

Per il fatto 2, 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.

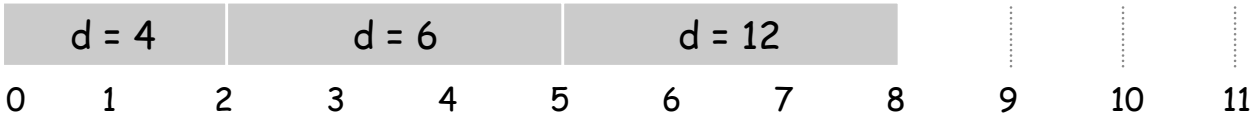
Prima di dimostrare il punto 3 consideriamo i seguenti fatti.

Fatto 3. 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 4. 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. Se nello scheduling tutte le coppie di job invertiti i e j ($d_i < d_j$ e j eseguito prima di i) sono separate da un job allora vuol dire che, per ogni coppia siffatta deve esistere un job $k \neq i$ eseguito subito dopo j che non fa parte di nessuna inversione. Deve quindi essere $d_k \geq d_j$ 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 5. Scambiare due job adiacenti invertiti i e j riduce il numero totale di inversioni di uno e non fa aumentare il ritardo massimo.

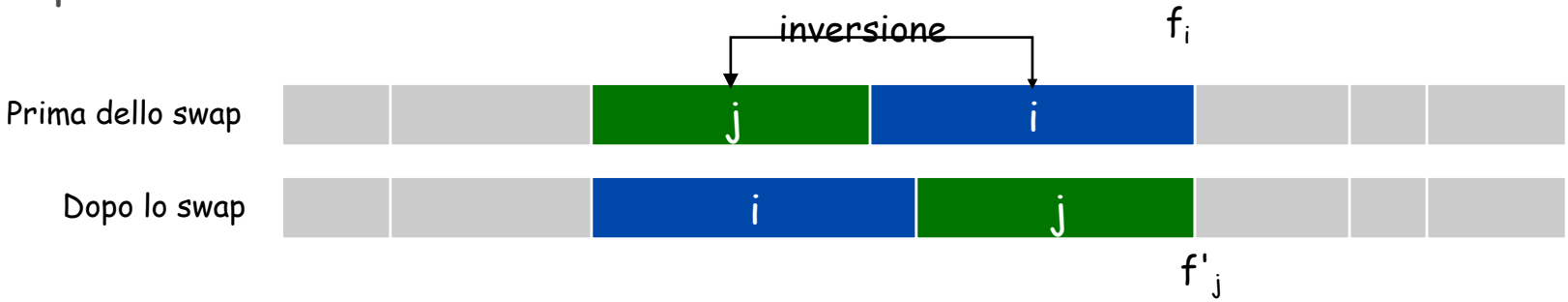
Dim. Supponiamo $d_i < d_j$ e che j precede i nello scheduling.

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

- Dim.
- Il fatto 3 implica che ogni soluzione ottima può essere trasformata in una soluzione ottima per cui non ci sono idle time.

- I fatti 4 e 5 implicano che ogni soluzione ottima contenente inversioni, contiene una coppia di job adiacenti invertiti e che se scambiamo le posizioni di questi job otteniamo ancora una soluzione ottima. Quindi possiamo scambiare di posto coppie di job adiacenti invertiti fino a che non ci sono più inversioni nella soluzione ottima.