**xAlgoritmi greedy**

Gli algoritmi greedy sono algoritmi di ottimizzazione: cercano la migliore soluzione possibile. Ciò che fa un algoritmo greedy ad ogni passo è effettuare la scelta che in quel momento sembra la migliore possibile nella speranza che procedendo in questo modo arrivi ad una soluzione globalmente ottima

Ho n job. Ognuno comincia ad un istante s\_j e finisce ad un istante f\_j. Ho un'unica risorsa (processore) per eseguirli, e posso eseguirne uno alla volta. Obiettivo: Eseguire quanti più job possibile, trovare il più grande insieme di job compatibili. Due job si dicono compatibili se non si sovrappongono i corrispondenti intervalli di esecuzione. Due job, i e j si dicono compatibili se f\_i<=s\_j || f\_j <= s\_i. L'unico caso in cui io potrei eseguire tutti i job è il caso in cui siano tutti compatibili (non c'è sovrapposizione tra di essi).

["Ragazzi, voi questo nel compito mi dovete scrivere:"]

**Input del problema:**

n job

Ogni job è caratterizzato da:

un tempo di inizio s\_i

un tempo di fine f\_i

Quindi:

**s\_1, s\_2, ..., s\_n**

**f\_1, f\_2, ..., f\_n**

**Obiettivo del problema:**

Trovare un sottoinsieme dei job quanto più grande possibile tale che i job dell'insieme siano tutti compatibili tra di loro

[Fine parentesi]

L'idea è di utilizzare una strategia greedy. Io posso basare la mia scelta su varie cose:

- Considerare i tempi di inizio s\_j in ordine crescente **[Earliest start time]**

- Considerare i tempi di fine f\_j in ordine crescente **[Earliest finish time]**

- Considerare i tempi di durata in ordine crescente **[Shortest interval]**

- Considerare il numero di job in conflitto in ordine crescente **[Fewest conflicts]**

**L'Earliest finish time è la migliore.**

Sort jobs by finish times so that f1 f2 ... fn.

f <- 0

A <- insieme vuoto

for j = 1 to n

{

if(s\_j >= f)

{

A <- A U {j}

f <- f\_j

}

return A

**Spiegazione**

Sort jobs by finish times so that f1 f2 ... fn.

Ordina i job in base al loro tempo di fine. Posso ribattezzare i job con degli interi progressivi tali che 1, 2, ..., n sia ora la mia lista di processi. Da cui f\_1 sarà il tempo di fine del primo processo, che sarà minore o uguale a f\_2, tempo di fine del secondo processo, e così via.

f <- 0

*Perché ancora non ho selezionato nessun job*

A <- insieme vuoto

*Insieme che contiene la soluzione, quindi i job selezionati, inizialmente è vuoto*

for j = 1 to n

*Scandisco i job in ordine dei tempi di fine*

{

*Sono già sistemati nel modo giusto, l'unica cosa che devo fare quando considero un nuovo job j è vedere se è compatibile con i precedenti*

if(s\_j >= f)

*Non devo far altro che confrontare il tempo di inizio di j con il tempo dell'ultimo job eseguito: tutti gli altri finiranno prima di lui.*

{

*Per provare la compatibilità di j con quelli già selezionati devo solo verificare che il job j cominci dopo quello selezionato in precedenza*

A <- A U {j}

*Il tempo di fine di questo job selezionato in precedenza sta in f, quindi controllo che s\_j >= f. Se lo è, allora scelgo j. Metto quindi j in A*

f <- f\_j

*e metto f\_j in f, in tal modo aggiornandolo.*

}

}

*Il ciclo finisce qui.*

return A

*Alla fine restituisco A, che è la fine del problema.*

**Analizziamone la complessità computazionale:**

Sort jobs by finish times so that f1 f2 ... fn.

*O(nlog(n))*

f <- 0

*Tempo costante*

A <- insieme vuoto

*Tempo costante*

for j = 1 to n

*Tempo lineare O(n)*

{

if(s\_j >= f)

*Tempo costante*

{

A <- A U {j}

*Tempo costante*

f <- f\_j

*Tempo costante*

}

}

return A

*Tempo costante*

*Quindi in totale, O(nlog(n)) + O(n) = O(nlog(n))*

**Ora dobbiamo dimostrarne l'ottimalità**, vediamone l'idea:

Supponiamo di eseguire il suddetto algoritmo e supponiamo che tra gli n job in input, riusciamo a selezionarne k. Chiamiamo questi job i\_1, i\_2, ..., i\_k e supponiamo che siano già nell'ordinamento giusto, ossia che i relativi tempi di fine f siano ordinati in questo modo: f\_1 <= f\_2 <= ... <= f\_k. Per dimostrarne l'ottimalità decido di confrontarla con una soluzione che so già essere ottima, formata dai job j\_1, j\_2, ..., j\_m. Per facilitare il confronto supponiamo che la soluzione ottima sia ordinata in base ai tempi di fine. Per dimostrare che la nostra soluzione sia anch'essa ottima devo dimostrare che il numero di job delle due soluzioni è lo stesso, cioè che k = m. Lo dimostro in due passi.

**Prima dimostro che la sequenza i\_1, i\_2, ..., i\_k termina non più tardi della corrispondente sottosequenza ottima j\_1, j\_2, ..., j\_k.**

Dimostreremo per induzione che per ogni r comrpeso tra 1 e k, il job r-esimo della soluzione greedy i\_r termina non più tardi del job r-esimo della soluzione ottima j\_r.

Caso base: Per r = 1 l'algoritmo greedy seleziona quel job essendo l'unico job.

Passo induttivo: i\_(r-1) termina non più tardi di j\_(r-1)

Osserviamo quel che accade per i\_r e j\_r. j\_r è stato selezionato dalla soluzione ottima e ciò significa che deve essere compatibile con gli altri della soluzione ottima, in particolare con j\_(r-1). Siccome i\_(r-1) termina non più tardi di j\_(r-1) per ipotesi induttiva, allora se j\_r è compatibile con j\_(r-1), a maggior ragione lo è con i\_(r-1). Ora, poiché gli altri job i\_1, i\_2, ..., i\_(r-1) per come sono stati scelti terminano prima di i\_r, sono quindi anch'essi compatibili con j\_r. Quindi, j\_r è un job compatibile con tutti i job selezionati fino al passo r-1-esimo dall'algoritmo greedy. L'algoritmo greedy, tra tutti i job compatibili con quelli già selezionati fino a quel momento prende quello con tempo di fine più piccolo. Ora, tra i job compatibili tra i\_1, i\_2, ..., i\_(r-1) ci sono sia i\_r che j\_r. Per il funzionamento dell'algoritmo greedy (ossia che sceglie sempre il job con il tempo di fine più piccolo), i\_r avrà sicuramente tempo di fine minore o uguale a j\_r. Da ciò si evince che l'intera sequenza i\_1, i\_2, ..., i\_r termina non più tardi della sequenza j\_1, j\_2, ..., j\_r.

Poi uso il primo punto per **dimostrare l'impossibilità che k < m**. Siccome k non può essere minore di m ed m è il massimo numero di job possibili, k = m.

Dal punto 1 posso concludere che i\_1, i\_2, ..., i\_k termina non più tardi della sequenza j\_1, j\_2, ..., j\_k, ma la sequenza di i termina sicuramente con il job k, ciò che dobbiamo **dimostrare è che termini anche la sequenza j.**

Supponiamo per assurdo che la sequenza ottima contenga più di k job, e che quindi dopo il k-esimo ci sia un altro job di indice k+1. Li stiamo sempre considerando in ordine dei tempi di fine, quindi j\_(k+1) è il k+1-esimo job nell'ordinamento dei job selezionati dall'algoritmo ottimo in base ai tempi di fine. Ripetiamolo, l'algoritmo ottimo non li seleziona in quest'ordine, siamo noi che li disponiamo così. Ora, j\_(k+1), se è stato selezionato è compatibile con j\_k, ma noi abbiamo dimostrato al punto 1 che l'intera sequenza i\_1, i\_2, ..., i\_k termina non più tardi della sequenza j\_1, j\_2, ..., j\_k. Quindi, se j\_(k+1) è compatibile con i job j\_1, j\_2, ..., j\_k, lo è anche con i\_1, i\_2, ..., i\_k, e quindi sarebbe stato selezionato anche dall'algoritmo greedy. Ma greedy non ha selezionato altri job e ciò significa che nessun'altro job era compatibile, **siamo arrivati ad un assurdo, k deve essere uguale ad m.**

L'idea di questa dimostrazione è:

Abbiamo dimostrato per induzione che greedy "sta sempre avanti" alla soluzione ottima. Il concetto di "stare sempre avanti" cambia a seconda del problema che stiamo considerando. Qui ciò che ce lo fa capire è che greedy seleziona un job che ad ogni passo seleziona un job che libera la risorsa prima o comunque non più tardi del corrispondente job della soluzione ottima. Ricordiamo ancora una volta che la soluzione ottima non seleziona i job nello stesso ordine in cui lo fa l'algoritmo greedy, e per poter utilizzare la parola "corrispondente" abbiamo ordinato la soluzione della soluzione ottima in base ai tempi di fine, e questa è un'idea che sfrutteremo spesso insieme a quella della sostituzione.

**Partizionamento degli intervalli**

Ora cambiamo problema:

Il partizionamento degli intervalli. Consiste in una serie di n attività, ognuna di esse caratterizzata da un tempo di inizio ed uno di fine. Qui io non voglio selezionare un sottoinsieme di attività bensì voglio eseguirle tutte. Ovviamente alcune di queste attività potrebbero sovrapporsi, quindi l'unica possibilità di eseguirle tutte è di avere più risorse identiche, ossia che ognuna di essi va bene per eseguire un'attività qualsiasi. Obiettivo? Minimizzare il numero di risorse utilizzate.

Il numero di sovrapposizioni di attività si chiama profondità, o, più rigorisamente, dato un insieme di attività in modo tale che sullo stesso livello non compaiano attività sovrapposte, definisco profondità il massimo numero di intervalli intersecabili da una retta che parte da 0 va a +inf spostandosi sull'asse delle ascisse.

Ci chiediamo: è sempre possibile trovare una soluzione che include un numero di risorse pari alla profondità? Proviamo a costruire un algoritmo greedy: controlliamo la profondità degli intervalli già osservati e allochiamo un numero di risorse pari alla profondità osservata. Questa è la caratteristica locale che mi permette di stabilire un criterio di ottimizzazione locale che è proprio ciò che fanno gli algoritmi greedy.

**Algoritmo**

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

}

**Spiegazione**

Sort intervals by starting time so that s1 s2 ... sn.

*Ordiniamo i job in base al loro tempo di inizio s\_i*

d <- 0

*d conta le risorse*

for j = 1 to n

*Il calcolo della profondità non viene fatto esplicitamente ma è intrinseco nel comportamento dell'algoritmo*

{

if (interval j can be assigned an already allocated resources v)

*Se all'intervallo che sto considerando in questo momento posso allocare una risorsa v già allocata in precedenza*

assign resource v to interval j

*Allora alloco una di quelle risorse v a j.[Nel compito i ragazzi che cosa dicono? Se all'intervallo j può essere assegnata una risorsa già allocata in precedenza, alloca al job j l'ultima risorsa utilizzata, perché c'è quel v che conta il numero di risorse. Ma ciò è sbagliato: v potrebbe essere occupata ma potrebbero essercene altre libere tra quelle già utilizzate in precedenza.]*

else

*Altrimenti[d viene incrementato se abbiamo bisogno di una risorsa perché le risorse utilizzate in precedenza sono ancora tutte occupate]*

allocate a new resource d + 1

*Allochiamo una nuova risorsa d + 1*

assign the new resource d+1 to interval j

*Assegnamo la nuova risorsa d + 1 a j*

d <- d + 1

*Incremento il numero di risorse d (d conta il numero di risorse allocate fino a quel momento)*

}

**Cosa dobbiamo dimostrare? Che l'algoritmo utilizza, ad ogni passo, un numero di risorse pari alla profondità dell'insieme di intervalli osservati fino a quel momento**, quindi in ogni punto d è la profondità dell'insieme di intervalli già considerati, e quindi, che alla fine d è uguale alla profondità dell'intero insieme di intervalli.

Sia d il numero di risorse usate al j-esimo passo dell'algoritmo, quindi alla j-esima iterazione del for. Questo d della dimostrazione corrisponde proprio alla variabile d dell'algoritmo.

Considero due casi:

Caso in cui alla j-esima iterazione alloco una nuova risorsa, quindi d è stato incrementato. Perché alloco una nuova risorsa? Perché tutte quelle allocate fino a quel momento erano occupate. Siccome li sto considerando in base ai tempi di inizio, ogni job i considerato fino a quel momento ha tempo di inizio minore di s\_j. Quindi tutti i job a cui sono assegnate le risorse già allocate devono ancora terminare, quindi ci sono d-1 attività che sono cominciate e devono terminare, quindi hanno tempo di fine maggiore del tempo di inizio della j-esima attività che sto considerando in questo momento, quindi per ogni i considerato fino a quel momento f\_i >= s\_j, e ciò significa che tutti gli intervalli (che iniziano prima si s\_j e finiscono dopo s\_j) includono il punto s\_j. Se traccio una retta che parte da 0 va a +inf spostandosi sull'asse delle ascisse, che è proprio la definizione di profondità, intersecherò d intervalli. Questo d sarà proprio minore o uguale alla profondità (perché non sappiamo se è il massimo). Noi abbiamo anche detto però che il numero di risorse utilizzabili non può essere più piccolo della profondità, quindi questo d sarà maggiore o uguale alla profondità. Da queste due affermazioni, d è proprio uguale alla profondità.

Caso in cui alla j-esima iterazione non alloco una nuova risorsa, quindi d non è stato incrementato. Vado a controllare qual è l'ultimo istante in cui io ho allocato una nuova risorsa. Chiamo j' l'ultima iterazione del for in cui è stata allocata una nuova risorsa. Per quello che ho dimostrato fino a questo momento io so che il valore di d è uguale alla profondità dell'insieme di intervalli 1, 2, ..., j'. Siccome l'insieme di intervalli 1, 2, ..., j' è un sottoinsieme dell'insieme di intervalli 1, 2, ..., j se io calcolo un massimo in un insieme più piccolo, questo massimo è minore o uguale al massimo dell'insieme più grande, quindi la profondità dell'insieme di intervalli 1, 2, ..., j' è sicuramente minore o uguale della profondità dell'insieme di intervalli 1, 2, ..., j. Quindi io ho d = profondità dell'insieme di intervalli 1, 2, ..., j' che è minore o uguale della profondità dell'insieme di intervalli 1, 2, ..., j. Quindi d <= della profondità dell'insieme di intervalli 1, 2, ..., j. Ma abbiamo detto però che il numero di risorse utilizzabili non può essere più piccolo della profondità, quindi questo d sarà maggiore o uguale alla profondità dell'insieme di intervalli. Da queste due affermazioni, d è proprio uguale alla profondità. dell'insieme di intervalli 1, 2, ..., j.

Ciò vale per un qualsiasi j, quindi anche j = n. Ho dimostrato che alla fine dell'esecuzione dell'algoritmo il valore della variabile d è proprio uguale alla profondità dell'insieme di intervalli 1, 2, ..., n; e siccome meglio della profondità non posso fare, quel valore di d è ottimo, il più piccolo possibile.

**Analizziamo l'algoritmo**

Analizzare l'algoritmo non è così semplice, perché non abbiamo informazioni su come si svolge la condizione dell'if. Potremmo ogni volta dover scandire tutte le risorse, e questo per ogni iterazione del for ci porterebbe ad un tempo quadratico. Utilizziamo una coda a priorità contenente le risorse già allocate. Ad ogni risorsa associerò una chiave che sarà il tempo di fine dell'attività a cui è allocata la risorsa in quel momento. A me serve estrarre la risorsa che si libera per prima, quindi sarà una minimum priority queue. Una volta estratta la risorsa devo verificare che il tempo riportato sia minore o uguale del tempo di inizio dell'attività che sto considerando.

Come inizializziamo e aggiorniamo la coda a priorità? All'inizio non ci sono risorse allocate.

[Promemoria: Quando scriviamo dello pseudocodice dobbiamo dire "la mia <inserire struttura dati> ha queste funzionalità: <f\_1>, <f\_2>, ..., <f\_n>; in questo modo la prof userà quelle funzionalità ed è meno probabile che sbaglierà a correggere l'algoritmo]

Ordiniamo i processi in base al loro tempo di inizio

Prima iterazione:

coda vuota, entro nel for, se la coda è vuota vado ad incrementare d e ad allocare una nuova risorsa.

Generica iterazione:

estraggo il minimo, confronto la chiave con s\_j

Se è minore o uguale alloco la risorsa ad s\_j, e questo a livello implementativo significa estrarre il minimo, cancellandolo, e reinserendolo con chiave f\_j.

Se è maggiore dobbiamo allocare una nuova risorsa p, creare la coppia (p, f\_j) e metterla all'interno della coda a priorità (incrementando d)

Quindi, in sostanza:

Partiamo con la coda vuota

Iniziamo a ciclare

Se la coda è vuota oppure e = getMin().key > s\_j

incrementa d

alloca una nuova risorsa d (che è stato incrementato, quindi si riferisce ad una nuova risorsa)

inserisci nella coda la coppia (d, f\_j) (che è stato incrementato, quindi si riferisce ad una nuova risorsa)

Altrimenti

update(queue, e, f\_j) //Aggiorno la coppia risorsa-chiave aggiornandone la chiave assegnano così la risorsa all'intervallo j.

Se utilizziamo un heap (ossia una coda a priorità implementata con un array i cui nodi sono disposti in un modo tale da rappresentare un albero binario) le operazioni fondamentali richiedono tempo O(log(m)), dove m è il numero di elementi all'interno della coda a priorità, che sono al massimo la profondità dell'insieme di intervalli, che nel caso pessimo è proprio il numero di job. Il ciclo for lo eseguo n volte, per un totale di O(nlog(m)), a cui va aggiunto il tempo dell'ordinamento O(nlog(n)). siccome m <= n, O(nlog(n)) prevale e in totale il tempo è O(nlog(n)).

**Algoritmo:**

Sort degli intervalli secondo il tempo di inizio quindi s1 s2 … sn

d <- 0

Inizializzazione coda delle risorse Q

For j = i to n

{

If(isEmpty(Q))

{

Allocare una nuova risorsa v

Assegnare la risorsa v a j

Enqueue (Q,v,f\_j)

d <- d + 1

}

Min <- getMin(Q)

If(Min < s\_j)

{

V = DeQueue(Q)

Assegno la risorsa v all’intervallo j

Enqueue (Q,V,f\_j)

}

Else

{

Allocare nuova risorsa w

Assegnare risorsa w all’intervallo j

EnQueue (Q, w, f\_j)

d <- d + 1

}

}

**Minimizzazione dei ritardi**

Altro problema: Minimizzazione dei ritardi.

["Quando i ragazzi descrivono il problema all'esame è un incubo: mettono l'output al posto dell'input"]

Ho un' insieme di attività che voglio eseguire. la scadenza , che è il tempo entro il quale l'attività j dovrebbe concludersi. L'obiettivo è minimizzare i ritardi

**Input:**

n richieste, ognuna ha:

un tempo di esecuzione

una scadenza per essere soddisfatta

[Il fatto che sia lì dentro non significa che faccia parte dell'input, non me lo date come input. input sono solo e . Gli dipendono dall'algoritmo, cioè dipendono dai tempi che ha stabilito l'algoritmo, perché per ogni attività l'algoritmo decide in quale istante cominciare quell'attività. è una conseguenza di quell' scelto dall'algoritmo e dell'input . Infatti è scritto lì come

**Obiettivo:**

Minimizzare il ritardo massimo, ma definiamo cos'è questo "ritardo massimo": il ritardo l\_j è la differenza del tempo in cui la richiesta è soddisfatta, chiamiamolo tempo di fine e la deadline . Però potrebbe darsi che la richiesta venga soddisfatta prima della deadline, e non possiamo ammettere un ritardo negativo, quindi in quel caso poniamo il ritardo = 0. Il ritardo massimo invece è il massimo tra tutti quegli l\_j.

Quindi, per minimizzare il ritardo massimo ci possono essere varie tecniche:

- Schedulare i job in base alle durate **[Shortest processing time first]**

- Schedulare i job in base alle scadenze **[Earliest deadline first]**

- Schedulare i job in base alla differenza tra tempi (durate) e scadenze **[Smallest slack]**

La soluzione corretta è la **Earliest deadline first**: cerco di eseguire prima quelli che dovrebbero terminare per prima.

**Algoritmo**

Sort n jobs by deadline so that d\_1 d2 … d\_n

t <- 0

for j = 1 to n

{

Assign job j to interval [t, t + ]

<- t

<- t +

t <- t +

}

output intervals [, ]

**Spiegazione**

Sort n jobs by deadline so that …

*Ordino le deadline in base al loro valore, dal più piccolo al più grande*

t <- 0

*t rappresenta l'istante in cui finisce il job precedente e inizia il job successivo. Più genericamente, l'istante in cui inizia il job j e termina il job j-1.*

for j = 1 to n

{

Assign job j to interval [t, t + ]

<- t

*Tempo di inizio stabilito dall'algoritmo*

<- t +

*Tempo di fine stabilito dall'algoritmo*

t <- t +

*Aggiorno t aggiungendogli*

}

output intervals [sj,]

*Restituisco tutti gli intervalli nella forma [, ]*

Questo algoritmo viene in genere mischiato ad interval scheduling.

**Dimostriamo l’ottimalità**

Per dimostrarne l'ottimalità dobbiamo introdurre il **concetto di inversione**

Diremo che due job i,j sono invertiti in uno scheduling se i viene eseguito prima di j ma la deadline di i è più grande di quella di j (quindi d\_i > d\_j). Nello scheduling prodotto dall'algoritmo greedy non possono esserci inversioni perché vengono schedulati in ordine di deadline. Un'altra caratteristica di greedy è che non ci sono tempi di idle.

I **tre concetti** su cui si basa la dimostrazione sono:

**1)La soluzione greedy non ha nè idle-time nè inversioni.**

**2)Ogni soluzione che gode delle proprietà espresse nel primo concetto ha lo stesso ritardo massimo della soluzione greedy.**

**3)Ogni soluzione ottima può essere trasformata in una soluzione, sempre ottimale, che soddisfa i primi due concetti quindi senza aumentare il ritardo massimo (per ricalcare il concetto 2).**

[Io all'esame raramente chiedo la dimostrazione di questi tre punti anche se semplice, quel che chiedo spesso è di utilizzare questi 3 punti per dimostrare che la soluzione (greedy) è ottima e sorprendentemente nessuno risponde.]

**Dimostrazione che greedy ha lo stesso ritardo massimo della soluzione ottima**:

Parto dalla soluzione ottima, che potrebbe anche contenere idle time e inversioni. Utilizzo il terzo concetto per trasformarla in una soluzione ottima senza idle-time e inversioni. Ora utilizzo il secondo concetto il quale stabilisce che una soluzione che rispetta il primo concetto (ossia che non ha idle-time nè inversioni) ha lo stesso ritardo massimo della soluzione greedy, quindi la soluzione ottima, siccome è stata trasformata in una soluzione ugualmente ottima che soddisfa il primo concetto, ha lo stesso tempo massimo della soluzione greedy.

**Dimostrazione primo concetto:**

**Si divide in due proprietà:**

**a) Nessun idle-time**

Un job comincia nello stesso istante in cui finisce il job precedente.

**b) Nessuna inversione**

Siano i,j due job. Se d\_i > d\_j, verrà eseguito prima j e poi i.

**Dimostrazione secondo concetto**

**Si divide in due proprietà:**

**I) In una soluzione priva di idle-time e inversioni, i job con la stessa scadenza vengono eseguiti consecutivamente.** [Se i job che hanno la stessa scadenza vengono eseguiti uno dopo l'altro in tutti gli scheduling che verificano le proprietà a,b; vuol dire che tutti gli scheduling che verificano quelle due proprietà sono uguali a meno della posizione dei job con la stessa scadenza, solo quella può variare. Quindi possiamo trasformare facilmente uno scheduling con quelle due proprietà in un altro scheduling con quelle due proprietà scambiando tra di loro i job con uguale deadline, e dobbiamo far vedere che nel fare questo il ritardo rimane inalterato: il ritardo massimo non viene modificato.]

La prima cosa che mi serve è verificare che in uno scheduling che verificano le proprietà a,b , i job con la stessa scadenza vengono eseguiti uno dopo l'altro. Prendiamo due job i,j e supponiamo che d\_i = d\_j e lo assegno alla variabile d. Supponiamo per assurdo che tra i e j venga eseguito un job q che ha scadenza diversa da d. Supponiamo, senza perdere di generalità che i venga eseguito prima di j. Siccome d\_q =/= d, ci possono essere due casi:

* **d\_q < d**

Siccome q viene eseguito dopo i, allora q formerà un'inversione con i.

* **d\_q > d**

Siccome q viene eseguito prima di j, allora q formerà un'inversione con j.

Quindi sono giunto ad un assurdo siccome per ipotesi le proprietà a,b sono verificate.

**II) Se io scambio due job con la stessa deadline il ritardo massimo non cambia.**

Prendiamo due job i,j e supponiamo che d\_i = d\_j e lo assegno alla variabile d. Senza perdere di generalità supponiamo che nello scheduling considerato, i viene eseguito prima di j. Il ritardo di j sicuramente non è aumentato, tutt'al più è diminuito dato che j viene eseguito prima. Il ritardo di i ora è identico a quello che prima aveva j perché la scadenza è la stessa e f\_j è quello che prima era f\_i. Quindi il ritardo accumulato da i è esattamente identico a quello che prima era occupato da j. Potrebbe accadere che tra i e j ci siano altri processi, e il loro ritardo potrebbe cambiare, ma sicuramente il ritardo di ciascuno di questi job non supera il ritardo che prima aveva j perché finiscono tutti quanti prima del tempo f\_j, quindi sicuramente il ritardo attuale è minore o uguale del ritardo che prima aveva j, perché le deadline sono uguali.

Ora, utilizziamo le due proprietà per dimostrare il secondo concetto.

Siano s\_1 e s\_2 due soluzioni che verificano le due proprietà a,b. Per il II fatto dimostrato, io posso trasformare s\_1 in s\_2 o viceversa senza modificare il ritardo massimo. [Per trasformare s\_1 in s\_2 non devo fare altro che effettuare un'inversione tra job con la stessa deadline in modo da ricondurmi allo scheduling di s\_2.] Ne consegue che s\_1 ed s\_2 sin dall'inizio avevano lo stesso ritardo massimo. Scegliendo s\_1 = una generica soluzione che soddisfa le proprietà a,b, ed s\_2 = soluzione greedy, posso trasformare s\_1 nella soluzione greedy senza modificare il ritardo massimo, il che vuol dire che s\_1 ha lo stesso ritardo massimo della soluzione greedy.

**Dimostrazione terzo concetto**

[Presa questa soluzione ottima, cosa devo eliminare? gli idle-time e le inversioni. Gli idle-time li elimino molto facilmente, le inversioni no.]

**Eliminazione idle-time:**

Supponiamo che la soluzione ottima contenga degli idle-time. A questo punto shiftiamo a sinistra le loro esecuzioni, anticipandoli; in questo modo la soluzione non può che migliorare, e si eliminano gli idle-time.

**Eliminazione inversioni, Si divide in più proprietà:**

**III)Se uno scheduling privo di idle time ha inversioni, ce n'è sicuramente una tra job adiacenti.** [Vi faccio vedere che quest'inversione può essere eliminata senza aumentare il ritardo massimo e quindi a questo punto, se nella selezione ottima da cui parto ci sono inversioni, eliminandole via via ottengo una soluzione ottima che soddisfa anche la proprietà b.]

Supponiamo che non sia vero, quindi che per ogni coppia di job invertiti, c'è un job che viene eseguito tra di loro. Siano i e j, tra tutte le inversioni, la coppia dei job più vicini nello scheduling che forma un'inversione. Siccome deve esserci un altro job k nello scheduling tra i e j. Assumiamo, senza perdere di generalità, che d\_i < d\_j, e quindi j viene eseguito prima di i. Siccome, però, la coppia dei job i,j forma l'inversione più piccola, k non deve formare un inversione nè con i, nè con j e questo significa che d\_j <= d\_k <= d\_i. Questo però contraddice l'ipotesi secondo la quale d\_i < d\_j, cioè che la coppia i,j sia un'inversione.

**IV)Se in uno scheduling privo di idle-time ci sono due job invertiti che cominciano uno dopo l'altro** (e per la proprietà III so per certo che esistono) **e li scambio tra di loro,il ritardo massimo non cambia.**

Siano i,j due job invertiti, quindi d\_i < d\_j e j precede i nello scheduling. Siano l\_1, l\_2, ..., l\_n i vari ritardi nello scheduling attuale. Ora scambiamo i e j, ottenendo così un nuovo scheduling. Siano quindi l\_1', l\_2', ..., l\_n' i ritardi del nuovo scheduling ottenuto. [Devo dimostrare che quest'inversione non mi ha fatto aumentare il ritardo massimo]. Per i job k non coinvolti dallo scambio si ha che l\_k = l\_k'. Per quanto riguarda i,j: l'esecuzione di i è stata anticipata, e il ritardo di i non può essere aumentato, per cui l\_i' < l\_i; il ritardo di j invece può essere aumentato. [Consideriamo il caso in cui il ritardo di j nel nuovo scheduling sia strettamente maggiore di 0, perché se fosse uguale a 0 non sarebbe un ritardo. Per cui assumiamo l\_j' > 0.] l\_j' lo calcolo come la differenza tra f\_j' e d\_j', ma f\_j' è esattamente il punto in cui finiva i, quindi posso sostituire f\_i a f\_j'. Osserviamo che siccome i,j era un'inversione d\_j > d\_i quindi f\_i - d\_j < f\_i - d\_i. Per definizione di ritardo ho che l\_i >= f\_i - d\_i. [Perché l\_i è il massimo tra questa differenza e 0.] Questa catena di disuguaglianze mi dice che l\_j' <= l\_i, quindi, riassumendo, il ritardo di i non può essere aumentato e il ritardo di j è minore o uguale a quello che prima aveva i, il ritardo degli altri job è rimasto invariato, quindi il ritardo massimo tra i due scheduling non è cambiato.

**Ora, utilizziamo le due proprietà per dimostrare il terzo concetto.**

Sia s una soluzione ottima. Se ci sono idle-time li posso eliminare senza far aumentare il ritardo massimo. Sfrutto la proprietà III secondo la quale se non ci sono idle time ma ci sono inversioni, sicuramente ci sono due job invertiti adiacenti nello scheduling. A questo punto sfrutto la proprietà IV secondo la quale se io inverto due job invertiti adiacenti, il ritardo massimo dello scheduling non cambia. Fino a che ci sono inversioni, elimino tutte le coppie di job invertiti adiacenti essendo certo che il ritardo massimo non aumenta. Alla fine mi ritroverò con uno scheduling senza inversioni [perché so che se ci sono inversioni ce n'è sicuramente una tra job adiacenti, e questo vale sempre] e il ritardo massimo sarà identico a quello di partenza.