Algoritmi greedy
III parte

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

Problema del caching offline ottimale

- Caching. Una cache è un tipo di memoria a cui si può accedere molto velocemente. Una cache permette accessi più veloci rispetto alla memoria principale ma ha dimensioni molto più piccole.

- Possiamo pensare ad una cache come ad un posto in cui possiamo tenere a portata di mano le cose che ci servono ma che è di dimensione limitata per cui dobbiamo riflettere bene su cosa mettervi e su cosa togliere per evitare che ci serva qualcosa che non abbiamo a portata di mano.

  - Cache hit: elemento già presente nella cache quando richiesto.
  - Cache miss: elemento non presente nella cache quando richiesto: occorre portare l'elemento richiesto nella cache e se la cache è piena occorre espellere dalla cache alcuni elementi per fare posto a quelli richiesti.

Problema del caching offline ottimale

Caching. Formalizziamo il problema come segue:

- Memoria centrale contenente un insieme U di n elementi
- Cache con capacità di memorizzare k elementi.
- Sequenza di m richieste di elementi d_1, d_2, …, d_m fornita in input in modo offline (tutte le richieste vengono rese note all'inizio). Non molto realistico!
- Assumiamo che inizialmente la cache sia piena, cioè contenga k elementi

Def. Un **eviction schedule ridotto** è uno scheduling degli elementi da espellere, cioè una sequenza che indica quale elemento espellere **quando c'è bisogno di far posto ad un elemento richiesto** che non è in cache.

Un eviction schedule non ridotto è uno scheduling che può decidere di inserire in cache un elemento che non è stato richiesto

Problema del caching offline ottimale

Un eviction schedule ridotto inserisce in cache un elemento solo nel momento in cui è richiesto e se non è presente già in cache al momento della richiesta.

Osservazione. In un eviction schedule ridotto il numero di inserimenti in cache è uguale al numero di cache miss.

Obiettivo. Un eviction schedule ridotto che minimizzi il numero di inserimenti (o equivalentemente di cache miss).

.

Eviction Schedule ridotto

Uno schedule non ridotto

a	a	b	c
a	a	x	c
c	a	d	c
d	a	d	b
a	a	c	b
b	a	x	b
c	a	c	b
a	a	c	b
a	a	c	b

Uno schedule ridotto

a	a	b	c
a	a	b	c
c	a	b	c
d	a	d	c
a	a	d	c
b	a	d	b
c	a	c	b
a	a	c	b
a	a	c	b

caching offline ottimale: Farthest-In-Future

Farthest-in-future. Quando viene richiesto un elemento che non è presente in cache, espelli dalla cache l'elemento che sarà richiesto più in là nel tempo o che non sarà più richiesto.

Cache in questo momento: a b c d e f

Richieste future: g a b c e d a b b a c d e a **f** a d e f g h ...

↑ cache miss (g) — ↑ Espelli questa (f)

Teorema. [Belady, 1960s] Farthest-in-future è uno schedule (ridotto) ottimo.

Dim. La tesi del teorema è intuitiva ma la dimostrazione è sottile.

Problema del caching offline ottimale

Esempio.

Cache di dimensione k = 2,
Inizialmente la cache contiene ab,
Le richieste sono a, b, c, b, c, a, a, b.
Usiamo farthest-in-future:
Quando arriva la prima richiesta di c viene espulso a perchè a verrà richiesto più in là nel tempo rispetto a b.
Quando arriva la seconda richiesta di a viene espulso c perchè c non viene più richiesto

Scheduling ottimo: 2 cache miss.

richieste	cache	
a	a	b
b	a	b
c	c	b
b	c	b
c	c	b
a	a	b
a	a	b
b	a	b

Implementazione dell'algoritmo di Belady

- Siano $d_1, d_2, \ldots, d_m$ le richieste in ordine dei tempo di arrivo

- Per ogni elemento d nella sequenza delle m richieste, l'algoritmo mantiene la lista L[d] contenente le posizioni in cui d appare nella sequenza. Ad esempio, se le richieste sono a, b, c, b, c, a, a, b allora L[a]=‹1,5,7›, L[b]=‹2,4,8›, L[c]=‹3,5›.

- L'algoritmo mantiene inoltre una coda a priorita` Q.
  - Per ogni elemento d in cache, la coda a priorita` Q contiene un'entrata (k,d), dove la chiave k è un intero che indica il punto della sequenza in cui verra` richiesto nuovamente l'elemento d. Se k=m+1 allora vuol dire che d non verra` piu` richiesto,

Implementazione dell'algoritmo di Belady

- Ogni volta che l'a esamina una nuova richiesta, l'algoritmo si comporta come segue.
- Se l'elemento richiesto d_j non è in cache, l'algoritmo
  - estrae da Q l'entrata (h,d) con chiave h piu` grande
  - espelle d dalla cache e inserisce. d_j in cache.
  - rimuove il primo elemento di $L[d_j]$ in modo che in testa venga a trovarsi la prossima posizione della sequenza in cui verra richiesto d_j.
  - se $L[d_j]$ non è vuota, inserisce in Q l'entrata (p, d_j) dove p è l'elemento che si trova ora in testa a $L[d_j]$; altrimenti inserisce in Q l'entrata (m+1,d_j)
- Se l'elemento richiesto d_j è in cache, l'algoritmo
  - rimuove il primo elemento di $L[d_j]$ in modo che in testa venga a trovarsi la prossima posizione della sequenza in cui verra richiesto d_j.
  - se $L[d_j]$ non è vuota, rimpiazza la chiave di d_j in Q con p, dove p è l'elemento che si trova ora in testa a $L[d_j]$; altrimenti rimpiazza questa chiave con m+1.

52

52

```
Input: requests d1,d2,…,dm arranged in ascending order of arrival time
For each element d, let L[d] the list of positions j s.t.dj=d;
initially L[d]=Ø
Let Q be a priority queue //entries associated with elements in cache
for j = 1 to m {                                          O(m+k log k )
  append j to list L[dj]                                  k = dimensione
  if(list L[dj] is empty and dj is in the cache)          cache
      insert (j,dj) in Q     //j is the key
}
for j = 1 to m {
   if (dj is in the cache){
        remove first element from L[dj]
        if(L[dj] is empty)                                    O(m log k )
          replace key of dj with m+1 in Q                     k = dimensione
        else                                                  cache
          {p ← first element of L[dj]
           replace key of dj with p in Q }
      }
   else{       //dj needs to be brought into the cache
       (h,dh) ← ExtractMax(Q)
        evict dh from the cache and bring dj to the cache
        remove first element from L[dj]
        if(L[dj] is NOT empty) {
            p ← first element of L[dj]
            insert (p,dj ) in Q  }
       else insert (m+1,dj) in Q
  }
```

53

Implementazione dell'algoritmo di Belady

- Per ogni elemento d_j della sequenza di richieste, il primo for inizializza la lista $L[d_j]$ e se d_j è gia` presente in cache inserisce d_j in Q con chiave uguale alla prima posizione in cui d_j appare nella sequenza delle richieste.
  - Ad esempio se inizialmente a e b sono in cache e la sequenza delle richieste è a, b, c, b, c, a, a, b allora dopo il primo for Q contiene le entrate (a,1) (b,2)
- Nella j-esima iterazione del secondo for, l'if-else gestisce i due seguenti casi.
  - d_j è presente in cache. In questo viene rimosso l'intero in testa a $L[d_j]$ e viene aggiornata la chiave di d_j con l'intero che si trova ora in testa a $L[d_j]$, sempre che $L[d_j]$ non sia vuota . Se $L[d_j]$ è vuota allora la chiave di d_j viene sostituita con m+1.
  - d_j non è presente in cache. In questo caso viene estratta da Q l'entrata (h, d_h) con chiave massima e h viene espulso dalla cache. Viene poi rimosso l'intero che si trova in testa a $L[d_j]$. Se dopo questa rimozione $L[d_j]$ non è vuota allora viene inserita in Q l'entrata (p, d_j), dove p indica l'intero che ora si trova in testa a $L[d_j]$. Se $L[d_j]$ è vuota allora in Q viene inserita l'entrata $(m+1, d_j)$

54

54

Analisi dell'algoritmo di Belady

L'algoritmo nella slide precedente richiende tempo O(m log k) se

- Ad ogni elemento è associato un flag che è true se e solo l'elemento è in cache
- Usiamo un heap binario come coda a priorità
  - assumiamo che l'heap supporti l'operazione changeKey che consente di modificare la chiave di un'entrata arbitraria dell'heap e l'operazione di remove che consente di cancellare un'entrata arbitaria. Queste operazioni possono essere implementata in modo da richiedere tempo O(log k).
- Consideriamo costante il tempo per espellere e inserire ciascun elemento in cache

55

55

Farthest-In-Future: ottimalità

La dimostrazione dell'ottimalità si basa sui seguenti fatti che andremo a dimostrare

1. Ogni schedule può essere trasformato in uno schedule ridotto senza aumentare il numero di inserimenti

2. Ogni schedule ridotto può essere trasformato nello schedule FF senza aumentare il numero di cache miss

- Per la 1 possiamo trasformare uno schedule ottimo S in uno schedule ridotto S' senza aumentare il numero di inserimenti
- Per la 2 possiamo trasformare S' nello schedule FF senza aumentare il numero di cache miss (= numero inserimenti)

→ FF va incontro allo stesso numero di inserimenti dell'algoritmo ottimo ed è quindi anch'esso ottimo

56

56

Farthest-In-Future: ottimalità

1. Un qualsiasi eviction schedule S può essere trasformato in un eviction schedule ridotto S' senza aumentare il numero di inserimenti nella cache.

Dim. Costruiamo S' a partire da S come segue:

- Caso 1. Se ad un certo tempo t, viene richiesto un elemento d che non è presente nella cache di S (ed è quindi portato in cache da S) allora S' non fa niente, se d è gia` nella sua cache, o inserisce d in cache in caso contrario. In quest'ultimo caso, se l'elemento q espulso da S è presente anche nella cache di S' allora S' espelle q altrimenti espelle un elemento tra quelli non presenti nella cache di S (deve essercene almeno uno).
- Caso 2. Se ad un certo tempo t, viene richiesto un elemento che è sia nella cache di S che in quella di' S e S porta in cache un elemento d (ovviamente senza che d sia stato richiesto), S' non inserisce niente in cache.
- Caso 3a. Se ad un certo tempo t, viene richiesto un elemento d presente nella cache di S ma non in quella di S' ed S non inserisce niente allora S' inserisce d in cache ed espelle un elemento tra quelli non presenti in S (deve essercene almeno uno)
- Caso 3b. Se ad un certo tempo t, viene richiesto un elemento d presente nella cache di S ma non in quella di S' ed S inserisce un elemento x ed espelle y allora S' inserisce d ed espelle y se questo e` presente nella sua cache oppure un qualsiasi elemento diverso da x.

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

57

Farthest-In-Future: ottimalità

- In realta` c'è anche il caso in cui al tempo t viene richiesto d che è presente in entrambe le cache ed S non fa niente. Questo caso è irrilevante ai fini della dimostrazione dal momento che S' resta banalmente ridotto e non viene modificata nessuna delle due cache e non aumenta il numero di inserimenti di nessuno dei due scheduling.
- Lo scheduling S' per come è stato costruito è uno schedule ridotto. Ci resta da dimostrare che fa un numero di inserimenti non superiore a quello di S.
- **Osservazione ✷: L'unico caso in cui S' inserisce un elemento ed S non inserisce niente è il caso 3a. Inoltre nel caso 2, S inserisce un elemento e S' non fa niente.**
  - Nel caso 1, S effettua un inserimento mentre S' potrebbe effettuarne anch'esso uno o non fare niente; nel caso 3b entrambi gli scheduling inseriscono un elemento.
- Faremo vedere che il numero di volte in cui si verifica il caso 3a è minore o uguale di quello in cui si verifica il caso 2. Questo ci garantisce che il numero di inserimenti fatti da S' non supera quello degli inserimenti effettuati da S.

Farthest-In-Future: ottimalità

- Abbiamo bisogno anche della seguente osservazione.

Osservazione ✷✷:

- Se al tempo t si verifica il caso 1 o il caso 3b allora dopo il tempo t la cache di S e quella di S' hanno un elemento in comune in piu` o esattamente lo stesso numero di elementi in comune di prima.
- Se al tempo t si verifica il caso 3a allora dopo il tempo t la cache di S e quella di S' hanno un elemento in comune in piu` .
- Se al tempo t si verifica il caso 2 allora dopo il tempo t le due cache potrebbero avere un elemento in comune in meno (l'elemento d inserito da S non è nella cache di S' e S toglie un elemento presente in cache di S')

Farthest-In-Future: ottimalità

- Indichiamo con cS e cS' il contenuto della cache di S e di S', rispettivamente.

- Si ha che:
  1. Affinche' si verifichi il caso 3a è necessario che la cache di S abbia un elemento non presente in quella di S', cioe` $cS\text{-}cS' \neq \emptyset$.
  2. Per l'osservazione ✷✷, l'unico caso che puo` far aumentare il numero di elementi di cS-cS' è il caso 2.
  3. Sempre per l'osservazione ✷✷, ogni volta che si verifica il caso 3a il numero di elementi in cS-cS' diminuisce di uno.
- La 1,2, e 3 insieme → il numero di passi in cui si verifica il caso 3a è minore o uguale del numero di passi in cui si verifica che si verifica il caso 2.

- Quest'ultima implicazione insieme all'osservazione ✷ nella slide precedente →il numero volte in cui S' effettua un inserimento ed S non fa niente è minore o uguale del numero di volte in cui S effettua un inserimento ed S' non fa niente

Farthest-In-Future: ottimalità

Teorema. Sia S uno **scheduling ridotto** che fa le stesse scelte dello scheduling S_{FF} di farthest-in-future per i primi j elementi, per un certo $j \geqq 0$. E' possibile costruire uno scheduling ridotto S' che fa le stesse scelte di S_{FF} per i primi j+1 elementi e determina un numero di cache miss non maggiore di quello determinato da S.

Dim.

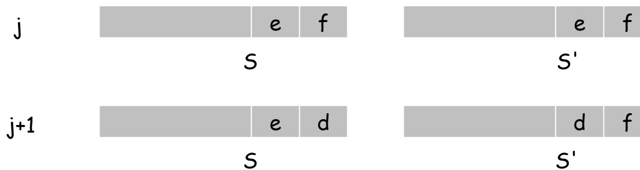
Produciamo S' nel seguente modo.

- Consideriamo la (j+1)-esima richiesta e sia $d = d_{j+1}$ l'elemento richiesto,
- Siccome S e S_{FF} hanno fatto le stesse scelte fino alla richiesta j-esima, quando arriva la richiesta (j+1)-esima il contenuto della cache per i due scheduling è lo stesso.
  - Caso 1: d è già nella cache. In questo caso sia S_{FF} che S non fanno niente perché entrambi sono ridotti.
  - Caso 2: d non è nella cache ed S espelle lo stesso elemento espulso da S_{FF} .
- In questi due casi basta porre S'=S visto che S ed S_{FF} hanno lo stesso comportamento anche per la (j+1)-esima richiesta.

Continua nella prossima slide

Farthest-In-Future: ottimalità

- Caso 3: d non è nella cache e S_{FF} espelle e mentre S espelle f ≠ e.
  - Costruiamo S' a partire da S modificando la (j+1)-esima scelta di S in modo che S' espella e invece di f.



  - ora S' ha lo stesso comportamento di S_{FF} per le prime j+1 richieste. Occorre dimostrare che S' riesce ad effettuare successivamente delle scelte che non determinano un numero di cache miss maggiore di quello di S.

Continua nella prossima slide

Farthest-In-Future: ottimalità

- Dopo la (j+1)-esima richiesta facciamo fare ad S' le stesse scelte di S fino a che, ad un certo tempo j', accade per la prima volta che non è possibile che S ed S' facciano la stessa scelta.
- A questo punto S' deve fare necessariamente una scelta diversa da quella di S. Facciamo però in modo che la scelta di S' renda il contenuto della cache di S' identico a quello della cache di S.
- Da questo punto in poi il comportamento di S' sarà identico a quello di S per cui andrà incontro allo stesso numero di cache miss.

Continua nella prossima slide

Farthest-In-Future: ottimalità

Notiamo che siccome i due scheduling fino al tempo j' si sono comportati in modo diverso un'**unica** volta (al passo j+1), il contenuto della cache nei due scheduling differisce in un singolo elemento che è uguale ad e in S ed è uguale a f in S'.

S | e   S' | f

Indichiamo con g l'elemento richiesto al tempo j'.

I casi che al tempo j' avrebbero permesso ad S' di fare la stessa scelta di S sono:

- g ≠ e, g ≠ f, g è presente nella cache di S: in questo caso g è presente anche nella cache di S' ed S' non fa niente come S.
- g ≠ e, g ≠ f, g non è presente nella cache di S ed S espelle un elemento diverso da e: in questo caso g non è neanche nella cache di S' ed S' può espellere lo stesso elemento espulso da S.

Nella prossima slide vediamo i casi in cui S' non può fare la stessa scelta di S.

Farthest-In-Future: ottimalità

Caso 3.1: g ≠ e, g ≠ f, g non è nella cache di S ed S espelle e. In questo caso g non è neanche nella cache di S'. Facciamo in modo che S' espella f. In questo modo dopo il tempo j' il contenuto della cache di S è uguale a quello della cache di S'. Il numero di cache miss di S è lo stesso di S'.

S | g   S' | g

Caso 3.2: g = f ed S espelle e. In questo caso S' non fa niente e da quel momento in poi le cache di S è uguale a quello di S'. Il numero di cache miss di S' è minore di quello di S.

S | f   S' | f

Caso 3.3: g = f ed S espelle e' ≠ e. In questo caso e' è presente anche nella cache di S' . Facciamo in modo che, al tempo j', S' espella e' ed inserisca e. Da questo momento la cache di S e quella di S' hanno lo stesso contenuto e il numero di cache miss in cui incorreranno i due scheduling sarà lo stesso. Il teorema non è ancora dimostrato per questo caso in quanto S' non è ridotto. Abbiamo però dimostrato che possiamo rendere S' ridotto senza aumentare il numero di inserimenti. Lo scheduling ridotto farà le stesse scelte di S_{FF} per i primi j+1 elementi in quanto sara identico ad S' fino al tempo j'-1.

S | f | e   S' | e | f

Farthest in Future: ottimalità

Resterebbe il caso g=e.

- Notiamo che al tempo j' non può accadere che g=e. Vediamo perché.
  - Al tempo j+1 S_{FF} ha espulso e al posto di f per cui, dopo il tempo j+1, e viene richiesto più tardi di f o non viene richiesto affatto.
  → Se dopo il tempo j+1 vi è una richiesta di e allora questa richiesta deve essere preceduta da una richiesta di f.

  - Come abbiamo visto nella slide precedente (casi 3.2 e 3.3) la richiesta di f in un tempo successivo al tempo j+1 porterebbe S' a fare una scelta diversa da S ma ciò non è possibile perché stiamo assumendo che j' è il primo momento (successivo al tempo j+1) in cui accade che S' non può fare la stessa scelta di S.

Farthest-In-Future: ottimalità

2. Ogni schedule ridotto può essere trasformato nello schedule FF senza aumentare il numero di cache miss

Dim.

- Consideriamo un eviction schedule ridotto S.
- Applicando il teorema precedente con j=0, si ha che possiamo trasformare S in uno schedule ridotto S_1 che per la prima richiesta si comporta come S_{FF} e determina un numero di cache miss non maggiore del numero di cache miss di S.
- Applicando il teorema con j=1, si ha che possiamo trasformare S_1 in uno schedule ridotto S_2 che per la prime due richieste si comporta come S_{FF} e determina un numero di cache miss non maggiore del numero di cache miss di S_1 e quindi di S.
- Continuando in questo modo, applicando cioè il teorema precedente per j=0,1,…,m-1, arriviamo ad uno schedule S_m che effettua esattamente le stesse scelte di S_{FF} (S_m= S_{FF}) e determina un numero di cache miss non maggiore del numero di cache miss di S.

Farthest-In-Future: ottimalità

Teorema. Farthest-in-future produce un eviction schedule S_{FF} ottimo.
Dim.

- Sia S^* uno schedule ridotto ottimo. Per il punto 2 (slide precedente) si ha che S^* può essere trasformato nello schedule S_{FF} senza aumentare il numero di cache miss. Di conseguenza S_{FF} determina lo stesso numero di cache miss di S^* ed è quindi uno schedule ridotto ottimo.

- Osserviamo che S_{FF} è ottimo non solo se restringiamo la nostra attenzione agli schedule ridotti ma è ottimo se consideriamo tutti i tipi di schedule (ridotti e non ridotti) perchè per il punto 1 possiamo trasformare uno schedule ottimo in uno schedule ridotto che effettua lo stesso numero di inserimenti.
  - NB: in questo caso parliamo di numero di inserimenti

Il problema del caching nella realtà

- Il problema del caching è tra i problemi più importanti in informatica.
- Nella realtà le richeste non sono note in anticipo come nel modello offline.
- E' più realistico quindi considerare il modello online in cui le richieste arrivano man mano che si procede con l'esecuzione dell'algoritmo.
- L'algoritmo che si comporta meglio per il modello online è l'algoritmo basato sul principio *Least-Recently-Used* o su sue varianti.
- *Least-Recently-Used* (LRU). Espelli la pagina che è stata richiesta meno recentemente
  - Non è altro che il principio Farthest in Future con la direzione del tempo invertita: più lontano nel passato invece che nel futuro
  - E' efficace perché in genere un programma continua ad accedere alle cose a cui ha appena fatto accesso (locality of reference). E' facile trovare controesempi a questo ma si tratta di casi rari.

Esercizio taglio tubo

- Abbiamo un tubo metallico di lunghezza L. Da questo tubo vogliamo ottenere al piu` n segmenti più corti, aventi rispettivamente lunghezze lung[1], lung[2], …, lung[n]. Il tubo viene segato sempre a partire da una delle estremità, quindi ogni taglio riduce la sua lunghezza della misura asportata. Descrivere un algoritmo greedy per determinare il numero massimo di segmenti che è possibile ottenere.
- Input: valori positivi L , lung[1], lung[2], …, lung[n].
- Obiettivo: selezionare il piu` grande sottoinsieme S di {1,2,…,n} in modo che la somma dei valori lung[i] per i in S non superi L.

Soluzione.

- Ordiniamo lung[1], lung[2], …, lung[n] in ordine non decrescente. Siano lung'[1], lung'[2], …, lung'[n] le lunghezze cosi` ordinate. Tagliamo prima un segmento di lunghezza lung'[1], poi quello di lunghezza lung'[2] e cosi` via fino a che ad un certo punto non riusciamo ad ottenere altri segmenti.
- Possiamo dimostrare che questa strategia greedy è ottima con la tecnica dello scambio. Siano $g_1,…,g_p$ i segmenti tagliati da greedy e $o_1,…,o_q$ quelli della soluzione ottima. Entrambe le sequenze sono ordinate in base all'ordine non decrescente delle lunghezze dei segmenti. Supponiamo che fino ad un certo $j\geq 0$ si abbia $g_1=o_1,…,g_j= o_j$. Facciamo vedere che se rimpiazziamo o_{j+1} con g_{j+1} in $o_1,…,o_q$ otteniamo ancora una soluzione ottima.

continua

70

Esercizio taglio tubo

- Se si ha gia` $g_{j+1}= o_{j+1}$ allora la dimostrazione è conclusa. In caso contrario, mostriamo che se rimpiazziamo o_{j+1} con g_{j+1} in $o_1,…,o_q$, otteniamo ancora una soluzione al problema senza ridurrre il numero di segmenti della soluzione. Notiamo prima di tutto che g_{j+1} non è uno dei $o_{j+1},…,o_q$ per cui puo` essere messo al posto di $o_{j+1}$ nella soluzione ottima. Infatti, per come funziona greedy, $g_{j+1}$ è la lunghezza del segmento piu` corto tra quelli di lunghezza maggiore o uguale di $g_1,…,g_j$. Ne consegue che se g_{j+1} è diverso da o_{j+1} allora g_{j+1} puo` essere solo minore di o_{j+1} e di conseguenza è minore (diverso) anche dai successivi $o_{j+2},…,o_q$. Se rimpiazziamo o_{j+1} con g_{j+1}, la parte che resta da tagliare del tubo è maggiore rispetto a quella che si aveva prima della sostituzione per cui anche le scelte successive $o_{j+2},…,o_q$ saranno possibili.

- Abbiamo dimostrato che per ogni $j\geq 0$, un algoritmo ottimo che fa le stesse prime j scelte di greedy puo` essere trasformato in un algoritmo ottimo che fa le stesse j+1 scelte di greedy.
- A partire da j=0 e fino ad arrivare a j=p possiamo quindi rimpiazzare man mano ogni scelta o_j con g_i in modo da ottenere dopo ogni scambio una soluzione ancora ottima (cioe` di q segmenti). Da cio ` si deduce che il numero q di segmenti della soluzione ottima non puo` essere maggiore del numero di segmenti p della soluzione greedy perche' se cosi` fosse l'algoritmo greedy si fermerebbe dopo aver tagliato p segmenti mentre l'algoritmo ottimo taglierebbe almeno un altro segmento. Cio` è impossibile in quanto dopo aver tagliato il p-esimo segmento, gli algoritmi hanno a disposizione una porzione di tubo della stessa lunghezza.

71

Esercizio 13 Cap. 4

- Una piccola ditta che si occupa di fotocopiare documenti ogni giorno riceve le richieste di n clienti. La ditta dispone di un'unica fotocopiatrice e fotocopiare i documenti dell'i-esimo cliente richiede tempo t_i. A ciascun cliente i è associato un peso w_i che indica l'importanza del cliente i.
- Indichiamo con C_i il tempo in cui viene terminata la copia dei documenti del cliente i. Se i documenti del cliente i vengono fotocopiati per primi allora $C_i=t_i$. Se i documenti di i vengono fotocopiati dopo quelli di j allora $C_i=C_j+t_i$
- Vogliamo eseguire le fotocopie in un ordine che minimizzi $\sum_{i=1}^{n} w_i C_i$

Soluzione.

- Ordiniamo le richieste in modo non crescente rispetto ai valori w_i/t_i (questo lo avrei suggerito in un'eventuale prova d'esame)
- Dimostriamo che la soluzione greedy è ottima utilizzando la tecnica dello scambio.
- Sia O l'ordinamento greedy e O' un ordinamento ottimo. Supponiamo $O' \neq O$
- Esistono due richieste j e k tali che j precede k in O e k precede j in O'. Inoltre devono esistere due richieste siffatte disposte una dopo l'altra in O'.
  - Consideriamo le richieste k e j piu` vicine in O' per cui risulta k precede j in O' e j precede k in O. Se esistesse una richiesta i che in O' viene eseguita tra k e j questa dovrebbe essere eseguita dopo k anche in O altrimenti k e i sarebbero due richieste eseguite in ordine diverso in O e sarebbero tra di loro piu` vicine di k e j . Per la stessa ragione i dovrebbe essere eseguita prima di j in O. Cio` è impossibile visto che j viene eseguita prima di k in O.

continua

Soluzione esercizio 13 Cap. 4

- Siano k e j due richieste adiacenti in O' eseguite in ordine inverso in O.
- Se in O' scambiamo k con j otteniamo che la somma $\sum_{i=1}^{n} w_i C'_i$ cambia come segue:
- Osserviamo che i valori C_i' delle richieste diverse dalla k e la j non cambiano
- Sia C' il momento in cui viene soddisfatta la richiesta del cliente che precede k in O'.
- Dopo aver scambiato k con j la somma $\sum_{i=1}^{n} w_i C'_i$ è modificata di una quantità pari a

$$- w_k (C'+t_k)+ w_k (C'+t_k+t_j)- w_j (C'+t_k+t_j)+w_j (C'+t_j)= w_k t_j- w_j t_k$$

- Siccome $w_j/t_j \geq w_k/t_k$ allora $w_j t_k \geq w_k t_j$ e di conseguenza la somma $\sum_{i=1}^{n} w_i C'_i$ risulta minore o uguale del valore che aveva prima dello scambio.
- Possiamo quindi scambiare in O' tutte coppie adiacenti che risultano invertite rispetto ad O fino a trasformare O' in O. Nel fare questi scambi il valore di $\sum_{i=1}^{n} w_i C'_i$ non aumenta per cui anche O è ottimo.