5
Algoritmi Online

5.1 Introduzione

Normalmente un algoritmo viene progettato per risolvere un problema per il quale l'input è completamente disponibile quando l'esecuzione dell'algoritmo inizia. Per completamente disponibile intendiamo che è disponibile nella sua interezza. Ci sono molte situazioni però dove l'input viene fornito *durante* l'esecuzione e quindi non è completamente noto all'inizio dell'esecuzione, ed è necessario fornire un output (parziale) man mano che si ricevono ulteriori pezzi dell'input. In tali situazioni si parla di problemi e algoritmi *online*. Quindi un algoritmo online riceve una serie di richieste (ognuna è un pezzo dell'input) ad ognuna delle quali deve fornire una risposta (ognuna è un pezzo dell'output) senza conoscere le richieste successive.

Ad esempio, supponiamo che l'input sia una sequenza di interi $x_1, x_2, \ldots, x_n$ che vengono forniti uno per volta. Quindi possiamo immaginare che l'algoritmo proceda per iterazioni, e alla i-esima iterazione viene fornito il valore di x_i. Immaginiamo di voler trovare il massimo fra gli x_i. Un algoritmo offline esaminerebbe tutti gli x_i e individuerebbe il massimo. Nella versione online del problema, però, dobbiamo stabilire se l'elemento x_i è il massimo alla i-esima iterazione, quindi senza conoscere i successivi elementi della sequenza di input.

Dovrebbe essere evidente che un problema online è più difficile del relativo problema offline: in quest ultimo caso possiamo prendere delle decisioni "conoscendo il futuro", mentre nel primo caso dobbiamo prenderle senza sapere quali saranno le successive richieste. Quindi è ovvio che un algoritmo offline sarà sempre migliore di un algoritmo online: se per un algoritmo offline proprio non riusciamo a fare meglio possiamo fare esattamente quello che fa un algoritmo online; un algoritmo online, invece, non può fare quello che fa un algoritmo offline in quanto ha meno informazioni a disposizione.

Per valutare quanto è buono un algoritmo online si utilizza l'*analisi competitiva*: cioè si valuta la soluzione trovata dall'algoritmo online confrontandola con quella di un algoritmo offline ottimo. Un algoritmo offline ottimo è un algoritmo che trova la soluzione ottima al problema conoscendo tutto l'input in partenza. Un algoritmo online è tanto più competitivo, quanto più esso approssima la soluzione ottima. In particolare diremo che un algoritmo online è c-competitivo se il valore della soluzione che produce si differenzia al massimo per un fattore moltiplicativo c dal valore della soluzione

ottima.

Il fattore di competitività per gli algoritmi online è l'analogo di quello di approssimazione per gli algoritmi approssimati. La differenza è che per l'analisi dell'approssimazione confrontiamo il comportamento di due algoritmi, quello approssimato e quello ottimo, sullo stesso problema, mentre per l'analisi della competitività confrontiamo il comportamento di algoritmi che operano su due problemi diversi, quello online e quello offline. Il fattore di competitività considera il caso pessimo; nella sezione 5.7 vedremo un'approccio alternativo che permette di considerare il caso medio.

Formalmente un problema online può essere descritto nel seguente modo. L'input da fornire all'algoritmo è dato da una sequenza di richieste $R = r_1, r_2, r_3, \ldots$. Le richieste devono essere soddisfatte nell'ordine in cui vengono ricevute: questo significa che l'algoritmo per ogni richiesta r_i, dovrà fornire un output o_i senza conoscere r_j per $j > i$. Produrre l'output o_i ha un costo. Il costo totale dell'algoritmo è la somma dei costi di tutti gli output.

Data una sequenza R di richieste denotiamo on $\textsc{opt}(R)$ il valore della soluzione fornita da un algoritmo offline ottimo. Sia A un algoritmo online che risolve il problema e sia $A(R)$ il costo della soluzione prodotta dall'algoritmo A. Consideriamo problemi di minimizzazione (per questo parliamo di costi). L'algoritmo A è detto c-competitivo se esiste una costante a tale che

$$Costo_A(R) \leq c \cdot Costo_{\text{OPT}}(R) + a,$$

per tutte le possibili sequenze di input R.

5.2 *Il problema dell'affitto degli sci*

Iniziamo con un esempio semplice: il problema dell'affitto degli sci. Supponiamo di voler andare a sciare ma di non avere gli sci. Possiamo o fittarli, pagando 1 per ogni volta che andiamo a sciare, opppure comprarli, pagando s. Se sapessimo a priori il numero t di volte che andremo a sciare potremmo facilmente decidere se conviene fittare gli sci o se conviene comprarli: se $t < s$ conviene fittarli se $t \geq s$ conviene comprarli. Nella versione online di questo problema non conosciamo apriori t, ma ogni volta dovremo decidere se fittare gli sci o comprarli (ovviamente dopo averli comprati non ci porremo più la domanda). In questo problema la richiesta r_i è sempre la stessa ("procurati gli sci") mentre i possibili output dell'algoritmo sono 3: (1) fitta, (2) compra, (3) usa gli sci comprati in precedenza. Il terzo output è possibile solo dopo aver comprato gli sci. Gli output hanno costo, rispettivamente 1, s e 0. Chiaramente un qualsiasi algoritmo sensato non farà altro che far fittare gli sci per le prime k volte (con $k \geq 0$), comprare gli sci alla k-esima volta, e usare gli sci comprati dalla $(k+1)$-esima volta. Quindi il costo può essere o solo k, quando $k < n$, oppure $k - 1 + s$, quando $k \geq n$.

In altre parole per stabilire quale algoritmo online usare dobbiamo solo decidere un valore per k.

Consideriamo una sequenza di n richieste. Il costo ottimale dell'algoritmo offline è $\min\{n, s\}$.

Il costo dell'algoritmo online è n se $n < k$, e $k - 1 + s$ se $n \geq k$.

Consideriamo l'algoritmo online con $k = s$. Dobbiamo confrontare tale costo con quello dell'algoritmo offline ottimo. Analizziamo il fattore di competitività separando i due casi

- Caso $n < s$. In questo caso entrambi gli algoritmi hanno un costo pari a n. Quindi l'algoritmo online è ottimo!
- Caso $n \geq s$. In questo caso l'algoritmo offline ottimo ha costo pari a s, mentre l'algoritmo online ha costo pari a $2s - 1$, quindi ha un fattore di competitività $2\frac{s-1}{s}$ che tende a 2 al tendere di s a ∞.

Questa strategia è la migliore possibile (Esercizio 1). Quindi, in pratica, il migliore algoritmo online è quello che fa fittare gli sci fino a che non raggiunge la cifra che sarebbe bastata a comprarli. A quel punto decide che era meglio comprarli e quindi li fa comprare. Anche se non ci saranno più utilizzi si è speso al massimo il doppio di quello che si sarebbe potuto spendere con il senno di poi.

5.3 *Problema del paging*

Un esempio classico di problema online è il problema della gestione delle pagine di memoria in un sistema operativo che supporta la memoria virtuale. Con il meccanismo della memoria virtuale si mette a disposizione dei programmi degli utenti una quantità di memoria RAM, divisa in cosiddette *pagine*, più grande di quella fisicamente disponibile. Questo è possibile in quanto non tutte le pagine di memoria devono essere fisicamente presenti in RAM in ogni singolo istante; pertanto il sistema operativo sfrutta l'hard disk per memorizzare le pagine che non sono necessarie facendo posto in RAM per quelle necessarie. Le pagine che devono essere presenti in RAM dipendono dai programmi che vengono eseguito dell'utente e quindi non si può sapere a priori quali saranno. Tuttavia ogni qualvolta il sistema operativo è costretto a cancellare una pagina dalla RAM (che verrà salvata nell'hard disk per il suo successivo recupero) deve decidere quale pagina deve essere cancellata. Chiaramente lo *scambio* delle pagine fra RAM e hard disk costa tempo, pertanto la strategia dovrebbe minimizzare il numero di scambi delle pagine. Conoscendo a priori l'intera sequenza di pagine necessarie rende il problema più facile. Tuttavia questo è un problema online in quanto successive pagine da caricare in RAM saranno note solo dopo aver scelto quali pagine cancellare.

Assumiamo che la memoria RAM possa contenere k pagine di memoria (simultaneamente) e che la memoria di massa (hard disk) possa invece contenere N pagine, con $k < N$. Ogni richiesta r_i è una specifica pagina che si vuole nella memoria RAM. Per soddisfare la richiesta r_i l'algoritmo deve far in modo che la pagina r_i sia presente in memoria. Se la pagina richiesta è già in memoria allora e non si deve far nulla. Se invece la pagina richiesta non è in RAM, allora occorre leggerla dalla memoria di massa e scriverla in una delle pagine della memoria RAM. Questa situazione viene detta in gergo *page fault*. Un page fault comporta un costo, che quantificheremo in un'unità, e richiede la scelta di una pagina che dovrà essere cancellata dalla RAM per far posto alla pagina r_i.

Consideriamo un esempio con $k = 4$, $N = 8$, e

$$\sigma = 1, 6, 1, 3, 4, 2, 1, 3, 2, 4, 5, 6, 8, 2, 5, 6, 4, 2.$$

Assumendo che la memoria cache sia inizialmente vuota,

$$C = \langle \cdot, \cdot, \cdot, \cdot \rangle,$$

le prime k diverse pagine richieste genereranno k page faults, ma non ci sarà bisogno di eliminare nessuna pagina dalla cache in quanto abbiamo k posizioni libere. Nel nostro esempio le prime due richieste $1, 6$ generano due page fault e la cache sarà

$$C = \langle 1, 6, \cdot, \cdot \rangle.$$

La terza richiesta è per una pagina già presente quindi non ci sarà un page fault. Le successive due richieste per 3 e 4 generano altri due page fault che faranno riempiere la cache

$$C = \langle 1, 6, 3, 4 \rangle.$$

Denotiamo con C_0 tale cache. Nella Figura 5.1 sono riportati i successivi cambiamenti.

Quando la cache è piena per gestire un page fault dovremo togliere una pagina presente dalla cache per far posto alla pagina richiesta. Ad esempio la successiva richiesta per la pagina 2 genera un page fault in quanto 2 non è in C e per far posto alla pagina 2 dovremo togliere una delle pagine presenti in C. Un algoritmo di paging deve decidere quale pagina togliere. Per questo esempio utilizziamo una regola qualsiasi, come ad esempio, scegliere la pagina nella posizione i-esima, partendo da $i = 1$, e incrementando i di 1 (modulo k) ad ogni page fault. Questa regola, chiaramente non ha nessuno criterio che la giustifichi e quindi la stiamo usando semplicemente per capire cosa deve fare un algoritmo di paging. Indicheremo con una sottolineatura la posizione che sarà usata per inserire il prossimo elemento e quindi cancellare quella presente in tale posizione:

$$C_0 = \langle \underline{1}, 6, 3, 4 \rangle.$$

Dunque, la pagina 2 verrà messa nella prima posizione, il che significa che la pagina 1 viene cancellata, e quindi si avrà C_1. Procedendo con la sequenza di richieste troviamo una richiesta per 1, che non è più nella cache quindi genera un page fault. La pagina 1 verrà caricata e messa nella posizione 2 cancellando la pagina 6; la cache diventa C_2. Le successive tre richieste, per $3, 2, 4$, non generano nessun page fault. Mentre le tre richieste che troviamo dopo, per $5, 6, 8$ generano 3 page faults e il corrispondente contenuto della cache sarà C_3, C_4 e C_5. La successiva richiesta è per la pagina 2 che è stata cancellata e quindi anche questa richiesta genera un page fault e ora la cache diventerà C_6. Le successive richieste per $5, 6$ non generano nessun page fault, mentre la richiesta per 4 sì, e la cache diventerà C_7. L'ultima richiesta per 2 non genera page fault. Il costo totale è, dunque, di 4 page fault iniziali per riempire la cache e poi di 7 page fault, quindi 11 in totale.

L'obiettivo di un algoritmo di paging è quello di minimizzare tale costo totale. L'algoritmo che abbiamo usato è abbastanza "stupido". Un algoritmo offline può

$C_0 = \langle \underline{1}, 6, 3, 4 \rangle$
$\sigma_6 = 2$, page fault
$C_1 = \langle 2, \underline{6}, 3, 4 \rangle$
$\sigma_7 = 1$, page fault
$C_2 = \langle 2, 1, \underline{3}, 4 \rangle$
$\sigma_{11} = 5$, page fault
$C_3 = \langle 2, 1, 5, \underline{4} \rangle$
$\sigma_{12} = 6$, page fault
$C_4 = \langle \underline{2}, 1, 5, 6 \rangle$
$\sigma_{13} = 8$, page fault
$C_5 = \langle 8, \underline{1}, 5, 6 \rangle$
$\sigma_{14} = 2$, page fault
$C_6 = \langle 8, 2, \underline{5}, 6 \rangle$
$\sigma_{17} = 4$, page fault
$C_7 = \langle 8, 2, 4, \underline{6} \rangle$

Figura 5.1: Esempio di esecuzione

$C = \langle 1, 6, 3, 4 \rangle$.
$\sigma_6 = 2$, page fault
$C = \langle 1, \underline{6}, 3, 4 \rangle$.
$C = \langle 1, 2, 3, 4 \rangle$.
$\sigma_{11} = 5$, page fault
$C = \langle \underline{1}, 2, 3, 4 \rangle$.
$C = \langle 5, 2, 3, 4 \rangle$.
$\sigma_{12} = 6$, page fault
$C = \langle 5, 2, \underline{3}, 4 \rangle$.
$C = \langle 5, 2, 6, 4 \rangle$.
$\sigma_{13} = 8$, page fault
$C = \langle 5, \underline{2}, 6, 4 \rangle$.
$C = \langle 5, 8, 6, 4 \rangle$.
$\sigma_{17} = 4$, page fault
$C = \langle 5, \underline{8}, 2, 6 \rangle$.
$C = \langle 5, 4, 6, 2 \rangle$.

Figura 5.2: Esecuzione di LFD

operare in modo più intelligente guardando le richieste future e decidendo in base a tali richieste quale pagine eliminare: non conviene eliminare pagine che saranno richieste nell'immediato futuro. Pertanto una possibile strategia è quella che decide di eliminare la pagina che sarà richiesta più tardi. Tale strategia è denominata LFD.

- LFD: (Longest Forward Distance) ad ogni page-fault espelli la pagina la cui prossima richiesta è più lontano nel futuro.

La Figura 5.2 mostra il comportamento di LFD sulla sequenza di input usata per l'esempio precedente. Il numero totale di page faults, $4 + 6 = 10$, è diminuito. Quindi, almeno in questo caso LFD è migliore. In realtà LFD è un algoritmo ottimo.

Teorema 5.3.1 *L'algoritmo* LFD *è un algoritmo ottimo per il problema del paging.*

Omettiamo la prova del precedente teorema. Il lettore interessato potrà trovarla, ad esempio, in S2012 [29]. L'algoritmo LFD non è un algoritmo online: ha bisogno di esaminare l'intero input per poter decidere quale pagina cancellare. Gli algoritmi online possono utilizzare solo la porzione di input nota fino al momento della decisione. I seguenti algoritmi, invece, sono online.

- LRU: (Least Recently Used) quando si verifica un page fault, selezionare come pagina da cancellare quella che è stata richiesta meno recentemente.
- FIFO: (First-In-First-Out) quando si verifica un page fault, selezionare come pagina da cancellare quella che sta da più tempo nella memoria RAM.
- LIFO: (Last-In-First-Out) quando si verifica un page fault, selezionare come pagina da cancellare quella che sta da meno tempo nella memoria RAM.

5.3.1 *Algoritmo* LRU

Teorema 5.3.2 *L'algoritmo LRU è k-competitivo.*

DIMOSTRAZIONE. Dobbiamo provare che per una qualunque sequenza $\sigma = \sigma_1\sigma_2 \dots \sigma_m$ di richieste, risulta che

$$Costo_{\text{LRU}}(\sigma) \leq k \times Costo_{\text{OPT}}(\sigma).$$

Partizioniamo la sequenza di richieste σ in fasi

$$\sigma = F_0F_1 \dots F_i \dots F_s$$

dove la fase F_0 è composta da una sottosequenza di richieste di σ su cui l'algoritmo LRU ha *al più* k page faults e le successive fasi F_i sono composte da sottosequenze con *esattamente* k page faults. La suddivisione di σ in fasi è effettuabile esaminando la sequenza partendo dalla fine.

Se riuscissimo a dimostrare che in ogni fase l'algoritmo ottimo ha almeno un page fault avremmo dimostrato il teorema in quanto si avrebbe che

$$Costo_{\text{OPT}} \geq s + 1$$

e quindi che

$$\begin{aligned} Costo_{\text{LRU}}(\sigma) &= \text{numero di fault che LRU genera su } \sigma \\ &\leq k \times \text{numero di fasi} \\ &= k \times (s+1) \\ &\leq k \times Costo_{\text{OPT}}. \end{aligned}$$

E ciò dimostrerebbe il teorema.

Per dimostrare che l'algoritmo ottimo avrà almeno un page fault in ognuna delle fasi procediamo come segue. Senza perdita di generalità assumiamo che la cache iniziale sia la stessa sia per l'algoritmo LRU che per quello ottimo off-line. Quindi la prima richiesta di σ che provoca un page fault lo provoca sia per LRU che per quello ottimo (a meno che l'algoritmo ottimo non decida di caricare la pagina prima anche in assenza di un page fault ma questo equivale a pagare il costo di un page fault prima del page fault, quindi il discorso funzionerebbe comunque). Ovviamente ciò avviene nella fase F_0. Quindi abbiamo provato che nella fase F_0 l'algoritmo ottimo paga il costo di almeno un page fault. Adesso proveremo che per ogni fase F_i, $i \geq 1$, un qualunque algoritmo (e quindi anche quello ottimo) deve necessariamente avere almeno un page fault. Consideriamo la fase F_i:

$$F_i = \sigma_{t_i}\sigma_{t_i+1} \dots \sigma_{t_{i+1}-1}$$

dove t_i è l'indice della prima richiesta della fase i. Sia P l'ultima pagina richiesta nella fase F_{i-1}. Essendo l'ultima pagina caricata nella fase precedente, P è sicuramente presente nella memoria all'inizio di F_i.

Dimostriamo adesso che F_i contiene almeno k richieste a pagine diverse da P. Sappiamo che in F_i ci sono k fault per l'algoritmo LRU. Questo significa che ci sono state k richieste a pagine non presenti nella memoria di LRU. Siano $A_1, A_2, \dots, A_k$ tali pagine.

Consideriamo ora queste due possibilità:

1. non ci sono duplicati in $\{P, A_1, A_2, \dots, A_k\}$.

   Questo è il caso più semplice in quanto $A_1, A_2, \dots, A_k$ sono tutte diverse e sono diverse anche da P. Poichè le pagine A_i sono pagine richieste nella fase F_i abbiamo dimostrato l'asserzione.

2. ci sono duplicati in $\{P, A_1, A_2, \dots, A_k\}$.

   Consideriamo due sottocasi:

   - P è duplicato; cioè $P = A_i$ per un qualche i
     Poiché P è la pagina più recente all'inizio della fase, l'unico modo per avere un page fault con $A_i = P$, è quello di epurare P dalla cache prima della richiesta A_i. Ma questo significa che P deve diventare la pagina più "vecchia" nella cache, e per questo servono $k-1$ richieste diverse da P e poi ci deve essere un page fault che toglie P e quindi serve una ulteriore richiesta per una pagina non presente nella cache e diversa da P. Quindi anche in questo caso abbiamo che nella fase F_i ci sono k richieste per pagine diverse tra loro e diverse da P (in questo caso potrebbero non essere $A_1, A_2, \dots, A_k$, ma altre pagine presenti in cache).

Esempio: $k = 3$, $\sigma = \underbrace{1,2,3}_{F_0},\underbrace{1,2,4,5,\dots}_{F_1}$ Per avere un fault sulla pagina $P = 3$ è necessario prima epurarla dalla cache e per fare questo deve diventare la pagina più vecchia; servono almeno $k-1$ richieste diverse da P, nell'esempio 1 e 2. Poi occorre una ulteriore richiesta diversa da P e dalle pagine in memoria per generare il page fault che toglie P dalla cache, nell'esempio 5. Le pagine $1, 2, 5$ sono k pagine diverse da P richieste nella fase.

- P non è duplicato; quindi $P \neq Q = A_i = A_j$ per due indici $i < j$.
  In realtà questo caso è molto simile al precedente, forse anche più semplice. Infatti per avere una pagina $Q = A_i = A_j$, $i < j$, duplicata fra le pagine che generano un page fault è necessario che fra il primo page fault A_i e il secondo page fault A_j ci siano altri k page fault; e poichè P è in memoria, questi page fault devono essere tutti per pagine diverse da P; prima di essere epurato dalla memoria P deve diventare la pagina più vecchia e non può farlo prima di k richieste a pagine diverse da P.

  Esempio: $k = 3$, $\sigma = \underbrace{1,2,3}_{F_0},\underbrace{4,2,3,1,4}_{F_1}$. Per avere un doppio fault sulla pagina $Q = 4$ è necessario avere almeno $k = 3$ richieste per pagine diverse da $P = 3$, nell'esempio $1, 2, 4$.

Quindi, in ogni caso, si ha che la fase F_i contiene almeno k richieste a pagine distinte e diverse da P.

A questo punto è facile concludere in quanto se nella fase F_i ci sono state richieste per k pagine diverse tra loro e diverse da P poichè P è in memoria all'inizio della fase le k richieste distinte necessariamente generano almeno un page fault. E questo è vero per un qualsiasi algoritmo, anche per quello ottimo. Pertanto il costo dell'algoritmo ottimo deve essere pari ad almeno il numero di fasi, cioè almeno $s + 1$. □

5.3.2 *Algoritmo* FIFO

Abbiamo dunque dimostrato che LRU è k-competitivo. In maniera analoga si può dimostrare che l'algoritmo FIFO è k-competitivo (Esercizio 5.)

5.3.3 *Algoritmo* LIFO

Si può anche dimostrare (Esercizio 8) che l'algoritmo LIFO *non* è c-competitivo per nessun valore di c. In altre parole è sempre possibile trovare una sequenza di input per la quale l'algoritmo LIFO genera un numero di page faults più grande di $c \cdot Costo_{OPT}$, per qualunque valore di c.

5.3.4 *Ottimalità algoritmi online*

Gli algoritmi LRU e FIFO sono algoritmi online ottimi, nel senso che non esistono algoritmi con competitività migliore. Infatti si ha il seguente risultato.

Teorema 5.3.3 *Non esistono algoritmi per il problema del paging che siano k'-competitivi per un qualsiasi $k' < k$.*

DIMOSTRAZIONE. Consideriamo l'algoritmo LFD e proviamo che per ogni sequenza $\sigma = \sigma_1\sigma_2\ldots$ di pagine nell'insieme $\{P_1, \ldots, P_k, P_{k+1}\}$ vale che

$$Costo_{\text{LFD}}(\sigma) \leq \frac{|\sigma|}{k} \tag{5.1}$$

dove $|\sigma|$ è il numero di richieste della sequenza. Notiamo innanzitutto che ogni richiesta σ_i si riferisce o ad una pagina già nella memoria cache o all'unica pagina correntemente al di fuori della memoria cache. Ciò a causa dell'ipotesi che σ è una sequenza di richieste a pagine nell'insieme $\{P_1, \ldots, P_k, P_{k+1}\}$. Supponiamo ora che ad un certo istante, in corrispondenza alla richiesta σ_i, l'algoritmo LFD espella la pagina P. Ricordiamo che l'algoritmo LFD espelle sempre la pagina che verrà richiesta più lontanamente nel futuro, tra tutte le pagine attualmente risiedenti nella memoria cache. Quindi, se P viene espulsa da LFD in conseguenza della richiesta σ_i vi saranno, dopo la richiesta σ_i, necessariamente almeno $k-1$ successive richieste alle altre pagine diverse da P, che per ipotesi stanno già in memoria, e quindi non creano alcun page fault. In altri termini, per ogni k richieste consecutive della sequenza σ, possiamo avere al più un page fault. Ovvero, il numero di page fault di LFD è minore o al massimo uguale a $|\sigma|/k$, e quindi la (5.1) è dimostrata.

Consideriamo ora un qualsiasi algoritmo online ALG. Costruiamo la seguente sequenza di richieste $\sigma = \sigma_1 \ldots \sigma_n$. Le prime k richieste $\sigma_1, \ldots, \sigma_k$ sono per k diverse pagine e senza perdere in generalità assumiamo che non ci siano page faults (siamo stati così fortunati da trovare nella memoria le k pagine richieste; visto che stiamo provando un lower bound questo non è un problema nel senso che se ci fossero dei page fault andrebbero ad avvalorare la tesi). La successiva richiesta σ_{k+1} sarà per una pagina non presente nella cache. Questo genera per l'algoritmo ALG, qualunque esso sia, un page fault. Sia P_j la pagina che ALG decide di espellere. La successiva richiesta σ_{k+2} sarà per P_j ed ovviamente genererà un page fault. Sia P'_j la pagina che ALG decide di espellere per questo nuovo page fault. La successiva richiesta σ_{k+3} sarà per P'_j. E così via. Cioè ogni nuova richiesta della sequenza sarà per la pagina appena espulsa.

Questa particolare sequenza che abbiamo costruito genera $n-k$ page faults (per le prime k richieste non ci sono page-fault).

Si ha che

$$Costo_{\text{ALG}}(\sigma) = n - k.$$

Se per assurdo l'algortimo ALG fosse k'-competitivo per un $k' < k$ si avrebbe che

$$Costo_{\text{ALG}}(\sigma) \leq k' \cdot Costo_{\text{OPT}} + a$$

per una qualche costante a ed una qualsiasi sequenza di input σ. D'altra parte abbiamo appena visto che $Costo_{\text{ALG}}(\sigma) = n - k$.

D'altra parte, sfruttando l'equazione (5.1) si avrebbe che[1]

$$Costo_{\text{OPT}} \leq Costo_{\text{LFD}} \leq \frac{n}{k}.$$

[1] In realtà noi sappiamo che LFD è ottimo, ma ai fini di questa dimostrazione possiamo ignorare questa conoscenza e sfruttare l'ovvia relazione $Costo_{\text{OPT}} \leq Costo_{\text{LFD}}$.

Pertanto se ALG fosse k'-competitivo si avrebbe che

$$\begin{aligned} n-k &= Costo_{\text{ALG}} \\ &\leq k' \cdot Costo_{\text{OPT}}(\sigma) + a \\ &\leq k' \cdot \frac{n}{k} + a. \end{aligned}$$

il che implicherebbe la disuguaglianza

$$n - k \leq k' \cdot \left(\frac{n}{k}\right) + a$$

che è impossibile per valori di n sufficientemente grandi. □

5.4 *Aggiornamento di liste*

Consideriamo il seguente problema. Dobbiamo mantenere una lista (non ordinata) di elementi; gli elementi possono essere inseriti o cancellati e possiamo ricevere richieste per recuperare un elemento presente nella lista. Più formalmente abbiamo una sequenza di richieste $\sigma = \sigma_1, \ldots, \sigma_m$ ognuna delle quali è una delle seguenti operazioni.

1. *inserimento* di un elemento nella lista;
2. *accesso* ad un elemento della lista;
3. *cancellazione* di un elemento dalla lista.

Per soddisfare la richiesta σ_i, un algoritmo online per questo problema deve prima di tutto attraversare la lista partendo dalla prima posizione fino a trovare l'elemento (se esiste). Il costo per fare ciò è proporzionale alla posizione j dell'elemento nella lista (se non c'è è proporzionale alla lunghezza n della lista). L'operazione di inserimento è consentita solo quando l'elemento non c'è, e l'elemento viene inserito in fondo alla lista. L'algoritmo può riorganizzare la lista in qualsiasi momento operando scambi di posizione fra elementi consecutivi[2]. Ogni scambio ha un costo pari a 1. Permetteremo anche di spostare un elemento, per il quale si è appena eseguita un'operazione di accesso, in una qualsiasi posizione più vicina alla testa della lista senza costi aggiuntivi (vedi Esercizio 9). Nel prosieguo useremo l'espressione "*scambi pagati*" per far riferimento agli scambi che fanno incorrere l'algoritmo nel costo dello scambio.

[2] L'esercizio 7 chiede di mostrare che questa non è una limitazione in quanto usando solo scambi fra elementi consecutivi possiamo ottenere un qualsiasi ordinamento.

Dato un algoritmo A il costo di $A(\sigma)$ su una sequenza di richieste σ è dato dalla somma dei costi delle singole richieste: $Costo(A(\sigma)) = \sum_{i=1}^{m} Costo(\sigma_i)$, dove il $Costo(\sigma_i)$ è dato dalla posizione j dell'elemento cercato più il costo degli scambi pagati operati dall'algoritmo. L'obiettivo è quello di avere un algoritmo online A che minimizza tale costo $Costo(A(\sigma))$.

5.4.1 *Strategia* MTF

La strategia MTF è quella più utilizzata: ogni volta che si accede ad un elemento lo si sposta in testa alla lista (Move-To-Front).

Teorema 5.4.1 *La strategia* MTF *è 2-competitiva.*

DIMOSTRAZIONE. Sia $\sigma_1, \ldots, \sigma_m$ una qualunque sequenza di input. Per provare che MTF è 2-competitivo dobbiamo provare che $Costo(\text{MTF}(\sigma)) \leq 2Costo(\text{OPT}(\sigma))$.

Per confrontare il comportamento di MTF rispetto al comportamento di OPT assumeremo che entrambi gli algoritmi partono dalla stessa lista. Quindi possiamo immaginare di eseguire parallelamente i due algoritmi e osservare le modifiche che fanno alla propria lista. Poichè gli algoritmi potrebbero comportarsi in modo diverso è possibile che durante la loro esecuzione gli elementi della lista assumano posizioni diverse in uno rispetto all'altro. Se due elementi x e y presenti nella lista hanno posizioni relative invertite, cioè x compare prima di y nella lista di MTF mentre nella lista di OPT x compare dopo di y, diremo che x e y formano una *inversione*.

Indicheremo con $Costo(\text{MTF}(\sigma_i))$ il costo in cui incorre MTF nel servire la richiesta σ_i e analogamente con $Costo(\text{OPT}(\sigma_i))$ il costo in cui incorre OPT nel servire la richiesta σ_i. Definiamo ora la funzione $\Phi(i)$ come il numero di inversioni che si hanno dopo aver servito la richiesta σ_i. Inizialmente non ci sono inversioni (gli algoritmi partono dalla stessa lista), quindi $\Phi(0) = 0$.

Proveremo che

$$\forall i, Costo(\text{MTF}(\sigma_i)) + \Phi(i) - \Phi(i-1) \leq 2Costo(\text{OPT}(\sigma_i)) - 1. \tag{5.2}$$

Prima di provare l'equazione (5.2), osserviamo che essa implica il teorema. Infatti si ha che

$$\begin{aligned} Costo(\text{MTF}(\sigma)) &= \sum_{i=1}^{m} Costo(\text{MTF}(\sigma_i)) \\ &\leq \sum_{i=1}^{m} (2Costo(\text{OPT}(\sigma_i)) - 1 - \Phi(i) + \Phi(i-1)) \\ &= 2Costo(\text{OPT}(\sigma)) - m - \Phi(m) + \Phi(0) \\ &\leq 2Costo(\text{OPT}(\sigma)) \end{aligned}$$

dove l'ultima disuguaglianza è dovuta al fatto che $\Phi(0) = 0$ e $\Phi(m) \geq 0$. Dunque la (5.2) implica il teorema. Procediamo a provare che essa è vera.

Consideriamo la richiesta σ_i per un qualunque i. Essa può essere una di tre operazioni: inserimento, cancellazione o accesso. Consideriamo prima il caso in cui σ_i è un richiesta di accesso ad un elemento k_i della lista. Denotiamo con t_i la posizione dell'elemento k_i nella lista di MTF prima che la richiesta σ_i venga servita e analogamente, denotiamo con s_i la posizione dell'elemento k_i nella lista di OPT prima che la richiesta σ_i venga servita.

Sia inoltre P_i il numero di scambi pagati in cui incorre OPT per servire σ_i. Si ricordi che MTF non usa scambi pagati ma solo spostamenti senza costi. Pertanto si ha che

$$Costo(\text{MTF}(\sigma_i)) = t_i \text{ e che } Costo(\text{OPT}(\sigma_i)) = s_i + P_i.$$

Distiguiamo ora due possibili casi.

- Caso 1: $t_i > s_i$.
  Considereremo due sottocasi: $P_i = 0$ e $P_i > 0$. Consideriamo prima $P_i = 0$. Questo

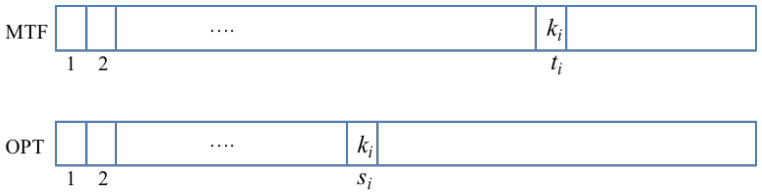

Figura 5.3: Posizioni di k_i nelle due liste (caso $t_i > s_i$).

significa che OPT non effettua scambi pagati, pertanto l'unico elemento che può cambiare di posizione è k_i. Ci chiediamo, quante inversioni ci sono prima di servire σ_i? Prima che k_i cambi posizione nella lista di MTF, poichè $t_i > s_i$, ci sono *almeno* $t_i - s_i$ inversioni in cui in elemento x si trova dopo k_i nella lista di OPT e prima di k_i nella lista di MTF. Sia X il numero di inversioni rimanenti, cioè quelle che non coinvolgono k_i. Abbiamo dunque che

$$\Phi(i-1) \geq t_i - s_i + X.$$

Per definizione MTF sposta k_i in testa alla lista, mentre OPT potrà spostare k_i in una qualunque delle posizioni $1, 2, \ldots, s_i - 1$ oppure lasciarlo nella posizione s_i. Pertanto dopo aver servito la richiesta σ_i ci saranno *al massimo* $s_i - 1$ inversioni che coinvolgono l'elemento k_i. Il numero X di inversioni che non coinvolgono k_i non è cambiato in quanto l'unico elemento che è stato spostato è k_i quindi la posizione relativa degli altri elementi non è cambiata. Quindi

$$\Phi(i) \leq s_i - 1 + X.$$

Pertanto abbiamo che

$$\Phi(i) - \Phi(i-1) \leq (s_i - 1 + X) - (t_i - s_i + X) = 2s_i - t_i - 1.$$

Ricordando che $Costo(\text{MTF}(\sigma_i)) = t_i$, e che $P_i = 0$ quindi $Costo(\text{OPT}(\sigma_i)) = s_i$, si ha che

$$\begin{aligned} Costo(\text{MTF}(\sigma_i)) + \Phi(i) - \Phi(i-1) &= t_i + \Phi(i) - \Phi(i-1) \\ &\leq t_i + (2s_i - t_i - 1) \\ &= 2s_i - 1 \\ &= 2Costo(\text{OPT}(\sigma_i)) - 1 \end{aligned}$$

e quindi la (5.2) è provata quando $P_i = 0$. Ora consideriamo $P_i > 0$. Ognuno degli P_i scambi pagati può far aumentare la differenza $\Phi(i) - \Phi(i-1)$ di al più 1, in quanto uno scambio di due elementi consecutivi può creare una sola nuova inversione. Pertanto abbiamo che

$$\Phi(i) - \Phi(i-1) \leq 2s_i - t_i - 1 + P_i.$$

Quindi si ha che

$$\begin{aligned} Costo(\text{MTF}(\sigma_i)) + \Phi(i) - \Phi(i-1) &= t_i + \Phi(i) - \Phi(i-1) \\ &\leq t_i + (2s_i - t_i - 1) + P_i \\ &= 2s_i + P_i - 1 \\ &\leq 2s_i + 2P_i - 1 \\ &= 2Costo(\text{OPT}(\sigma_i)) - 1. \end{aligned}$$

Dunque per il caso $t_i > s_i$ la (5.2) vale sempre.

- Caso 2: $t_i \leq s_i$.
  Possiamo procedere con un ragionamento analogo. Prima di servire la richiesta σ_i ci sono *almeno* $s_i - t_i$ inversioni che coinvolgono k_i e quindi, denotando con X quelle che non coinvolgono k_i si ha che $\Phi(i-1) \geq s_i - t_i + X$. Analogamente al caso precedente, dopo aver servito la richiesta σ_i si ha che MTF ha spostato k_i in testa alla lista mentre OPT può aver spostato k_i in una qualsiasi delle posizioni $1, 2, \ldots, s_i - 1$ e quindi ci possono ora essere *al massimo* $s_i - 1$ inversioni che coinvolgono k_i. Pertanto, come nel caso precedente, si ha che $\Phi(i-1) \leq s_i - 1 + X$. Considerando anche gli scambi pagati di OPT potremo avere P_i ulteriori inversioni in più, pertanto

  $$\Phi(i) - \Phi(i-1) \leq (s_i - 1 + X) - (s_i - t_i + X) + P_i = t_i - 1 + P_i.$$

  E quindi

  $$\begin{aligned} Costo(\text{MTF}(\sigma_i)) + \Phi(i) - \Phi(i-1) &\geq t_i + t_i - 1 + P_i \\ &= 2t_i + P_i - 1 \\ &\leq 2s_i + P_i - 1 \\ &\leq 2s_i + 2P_i - 1 \\ &= 2Costo(\text{OPT}(\sigma_i)) - 1. \end{aligned}$$

  Dunque anche per il caso $t_i \leq s_i$ la (5.2) è vera.

  Per concludere la prova del teorema, occorre considerare i casi in cui σ_i è un inserimento o una cancellazione. Consideriamo il caso in cui la richiesta è un inserimento. Detta n la lunghezza della lista, si ha che

  $$Costo(\text{MTF}(\sigma_i)) = n + 1 \text{ e } Costo(\text{OPT}(\sigma_i)) = n + 1.$$

  Poichè l'inserimento di un elemento in fondo alla lista non crea nessuna nuova inversione si ha che $\Phi(i) - \Phi(i-1) = 0$, quindi se la (5.2) valeva prima di σ_i, vale anche dopo.

  Rimane da considerare il caso della cancellazione. Questo caso è molto simile a quello dell'accesso. Basta osservare che una cancellazione non crea nuove inversioni ma può solo eliminarne alcune. Quindi, analogamente a quanto fatto per l'accesso, dette t_i e s_i le posizioni dell'elemento da cancellare rispettivamente nelle liste di MTF e OPT, potremo procedere esattamente come nel caso dell'accesso.

□

5.4.2 *Limite competitività*

La strategia MTF è, in pratica, la migliore possibile, nel senso che si può provare che non esistono algoritmi online con un fattore di competitività sostanzialmente migliore di 2. Per provare questa asserzione analizzeremo il costo di un generico algoritmo online e mostreremo che ci sono casi in cui l'algortimo deve necessariamente avere un costo che è quasi il doppio del costo di OPT.

Consideriamo un generico algoritmo online $\mathcal{A}$. Sia $\sigma = \sigma_1 \dots \sigma_m$ la sequenza di richieste che richiede sempre l'ultimo elemento della lista mantenuta da $\mathcal{A}$. Quindi, qualunque sia $\mathcal{A}$, il suo costo è

$$Costo(\mathcal{A}(\sigma)) = nm$$

dove n è la lunghezza della lista. Detto in altre parole, abbiamo osservato che per ogni algoritmo online esiste una particolare sequenza di richieste che causa un costo di nm.

Per confrontare tale costo con quello di un algoritmo ottimo considereremo il seguente algoritmo offline $\mathcal{B}$. Eseguiamo una fase preliminare in cui si analizza la sequenza σ per calcolare le frequenze con cui gli elementi compaiono; siano $f_1 \geq f_2, \dots \geq f_n$ tali frequenze ordinate dall'elemento più frequente a quello meno frequente. Riorganizziamo la lista ordinando gli elementi in base a tali frequenze, cioè mettendo nella prima posizione l'elemento più frequente e nell'ultima quello meno frequente. Per effettuare l'ordinamento sono sufficenti al più $n(n-1)/2$ scambi, ad esempio usando BubbleSort. Dopo l'ordinamento si potranno servire le m richieste, che faranno incorrere in un costo pari a $\sum_{i=1}^{n} i f_i$ e quindi si ha che

$$Costo(\mathcal{B}(\sigma)) \leq \frac{n(n+1)}{2} + \sum_{i=1}^{n} i f_i.$$

Osserviamo ora che la condizione $f_1 \geq f_2, \dots \geq f_n$, implica

$$\sum_{i=1}^{n} i f_i \leq \frac{m(n+1)}{2}. \tag{5.3}$$

La prova dell'equazione (5.3) è lasciata come esercizio (vedi Esercizio 10). Dunque si ha che

$$Costo(\text{OPT}(\sigma)) \leq Costo(\mathcal{B}(\sigma)) \leq \frac{n(n-1)}{2} + \frac{m(n+1)}{2}$$

e quindi che

$$\frac{m(n+1)}{2} \geq Costo(\text{OPT}(\sigma)) - \frac{n(n-1)}{2}$$

e pertanto

$$\begin{aligned} Costo(\mathcal{A}(\sigma)) &= mn \\ &= \frac{2n}{(n+1)} \cdot \frac{m(n+1)}{2} \\ &\geq \frac{2n}{(n+1)} \left[Costo(\textsc{opt}(\sigma)) - \frac{n(n-1)}{2} \right] \\ &= \left(2 - \frac{2}{n+1}\right) Costo(\textsc{opt}(\sigma)) - \frac{n}{n+1} n(n-1) \\ &\geq \left(2 - \frac{2}{n+1}\right) Costo(\textsc{opt}(\sigma)) - n(n-1) \end{aligned}$$

Osserviamo ora che il termine $\frac{2}{n+1}$ può essere reso piccolo a piacere considerando n grandi e che scegliendo m molto più grande di n, m diventa il fattore predominante nel costo degli algoritmi. Dunque per un n grande e un m molto più grande si ha che il costo di $\mathcal{A}$, cioè di un qualsiasi algoritmo online, deve essere più grande del (o più precisamente, di quasi il) doppio del costo di un algoritmo ottimo.

5.5 *Problema load balancing*

Il problema del load balancing che abbiamo visto nella sezione 3.3 può essere anche un problema online, quando l'input viene fornito un compito alla volta e bisogna assegnare il compito ad una delle macchine prima di vedere il compito successivo. L'algoritmo ApproxLoadInOrder è in realtà un algoritmo online! Il fattore di approssimazione è l'equivalente del fattore di competitività, quindi abbiamo già visto un algoritmo online 2-competitivo per questo problema.

5.6 *Problema Bin Packing*

La sequenza di richieste $a_1, a_2, \ldots, a_m$ rappresenta una sequenza di oggetti di peso a_i. Gli oggetti devono essere inseriti in dei contenitori $B_1, B_2, \ldots, B_b$ con $b \geq m$; ogni contenitore B_i ha capacità c, quindi la somma dei pesi degli oggetti assegnati a B_i non può superare c. Gli oggetti vanno assegnati in maniera online e si vuole minimizzare il numero totale di contenitori. Assumeremo che ogni singolo oggetto possa essere messo da solo in un qualsiasi conenitore (cioè che $\max_i a_i \leq c$). Infine, quando si decide di usare un nuovo contenitore, occorre "chiudere" il precedente, il che significa che non potremo più inserire altri oggetti in quel contenitore; questo vincolo, ad esempio, codifica il fatto che i contenitori debbano essere spediti e quindi "chiuderli" equivale a spedirli.

Come per il problema del load balancing, anche per questo problema l'approccio greedy fornisce un algoritmo sufficientemente buono. L'algoritmo è il seguente.

Algorithm 20: NextFit

```
j ← 1
for i ← 1 to m do
    if a_i non può essere inserito in B_j then
        j ← j + 1
    Inserisci a_i in B_j
```

Questo algoritmo fornisce un fattore di competitività pari a 2. Ossserviamo che l'algoritmo ottimo non può usare meno di $s = \frac{\sum_i^m a_i}{c}$ contenitori: questo caso limite si ha quando tutti i contenitori usati sono completamente pieni. L'algoritmo NextFit usa al massimo s contenitori. Infatti per ogni coppia di contenitori consecutivi, si ha che il carico totale dei due contenitori è superiore a c, in quanto si passa al secondo contenitore proprio perchè il carico del primo contenitore più l'oggetto considerato fa diventare il peso totale maggiore di c. Dunque l'algoritmo NextFit non userà mai più di $2s$ contenitori.

5.7 *Analisi probabilistica*

Gli algoritmi online che abbiamo visto in precedenza sono stati valutati in base alla loro competitività, cioè confrontandoli con i corrispondenti algoritmi offline ottimi. Questo tipo di analisi considera il caso pessimo ed è quella più utilizzata. Tuttavia ci sono dei casi nei quali vogliamo valutare l'algoritmo con un'analisi probabilistica facendo delle assunzioni sulla distribuzione dell'input. Questo tipo di analisi fornisce una misura del comportamento dell'algoritmo nel caso medio. L'analisi può essere difficile da fare in generale, ma se possiamo fare delle forti assunzioni sulla distribuzione dell'input, come ad asempio assumere che tutti i possibili input hanno la stessa probabilità di verificarsi, allora l'analisi può essere sufficientemente facile. Inoltre ci possono essere dei problemi per i quali nel caso pessimo, che è quello considerato dall'analisi competitiva, l'algoritmo online è estremamente poco competitivo mentre potrebbe comportarsi bene in media; in questi casi può avere più senso fornire un'analisi probabilistica. Per esemplificare l'analisi probabilistica riprendiamo il problema del massimo usato come esempio nell'introduzione al capitolo.

Potremmo formularlo nella seguente versione alternativa. Supponiamo di dover vendere una casa e supponiamo che n persone hanno contattatto l'agenzia di vendita per un appuntamento per presentarci un'offerta di acquisto. Siamo vincolati ad incontrare i potenziali acquirenti uno per volta e dopo avere ricevuto l'offerta dobbiamo decidere se declinarla e quindi passare alla prossima oppure accettarla e quindi non considerare più le prossime offerte (che ancora non conosciamo). Ovviamente il nostro obiettivo è quello di vendere al maggior offerente! La strategia online deve selezionare un elemento x_j, sulla base della conoscenza delle prime j offerte $x_1, x_2, \ldots, x_j$ e senza conoscere $x_{j+1}, \ldots, x_n$. Abbiamo già osservato che qualunque strategia online potrebbe non portare all'individuazione del massimo, che può essere determinato solo nel caso offline. Per facilitare l'analisi assumeremo che le offerte siano ordinate in modo casuale,

cioè che i possibili ordinamenti delle offerte hanno tutti la stessa probabilità di essere la sequenza delle offerte. Questa è un'assunzione abbastanza ragionevole se le offerte vengono fatte in modo indipendente.

Consideriamo ora i seguenti algoritmi, parametrizzati dall'indice r, $0 \leq r \leq n$: rifiutiamo incondizionatamente le prime $r-1$ offerte, ma ne calcoliamo il massimo M_{r-1}; accettiamo la prima delle successive offerte che è maggiore o uguale a M_{r-1} e se tutte le successive offerte sono minori di M_{r-1} accettiamo l'ultima offerta.

Facciamo un esempio, supponiamo che l'insieme delle offerte sia $\{10, 12, 18, 20\}$. Ci sono 24 possibili sequenze di questi 4 elementi:

#	x_1	x_2	x_3	x_4	A_1	A_2	A_3	A_4
1	10	12	18	20	($M_0 = 0$) 10	($M_1 = 10$) 12	($M_2 = 12$) 18	($M_3 = 18$) 20
2	10	12	20	18	($M_0 = 0$) 10	($M_1 = 10$) 12	($M_2 = 12$) 20	($M_3 = 20$) 18
3	10	18	12	20	($M_0 = 0$) 10	($M_1 = 10$) 18	($M_2 = 18$) 20	($M_3 = 18$) 20
4	10	18	20	12	($M_0 = 0$) 10	($M_1 = 10$) 18	($M_2 = 18$) 20	($M_3 = 20$) 12
5	10	20	12	18	($M_0 = 0$) 10	($M_1 = 10$) 20	($M_2 = 20$) 18	($M_3 = 20$) 18
6	10	20	18	12	($M_0 = 0$) 10	($M_1 = 10$) 20	($M_2 = 20$) 12	($M_3 = 20$) 12
7	12	10	18	20	($M_0 = 0$) 12	($M_1 = 12$) 18	($M_2 = 12$) 18	($M_3 = 18$) 20
8	12	10	20	18	($M_0 = 0$) 12	($M_1 = 12$) 20	($M_2 = 12$) 20	($M_3 = 20$) 18
9	12	18	10	20	($M_0 = 0$) 12	($M_1 = 12$) 18	($M_2 = 18$) 20	($M_3 = 18$) 20
10	12	18	20	10	($M_0 = 0$) 12	($M_1 = 12$) 18	($M_2 = 18$) 20	($M_3 = 20$) 10
11	12	20	10	18	($M_0 = 0$) 12	($M_1 = 12$) 20	($M_2 = 20$) 18	($M_3 = 20$) 18
12	12	20	18	10	($M_0 = 0$) 12	($M_1 = 12$) 20	($M_2 = 20$) 10	($M_3 = 20$) 10
13	18	10	12	20	($M_0 = 0$) 18	($M_1 = 18$) 20	($M_2 = 18$) 20	($M_3 = 18$) 20
14	18	10	20	12	($M_0 = 0$) 18	($M_1 = 18$) 20	($M_2 = 18$) 20	($M_3 = 20$) 12
15	18	12	10	20	($M_0 = 0$) 18	($M_1 = 18$) 20	($M_2 = 18$) 20	($M_3 = 18$) 20
16	18	12	20	10	($M_0 = 0$) 18	($M_1 = 18$) 20	($M_2 = 18$) 20	($M_3 = 20$) 10
17	18	20	10	12	($M_0 = 0$) 18	($M_1 = 18$) 20	($M_2 = 20$) 12	($M_3 = 20$) 12
18	18	20	12	10	($M_0 = 0$) 18	($M_1 = 18$) 20	($M_2 = 20$) 10	($M_3 = 20$) 10
19	20	10	12	18	($M_0 = 0$) 20	($M_1 = 20$) 18	($M_2 = 20$) 18	($M_3 = 20$) 18
20	20	10	18	12	($M_0 = 0$) 20	($M_1 = 20$) 12	($M_2 = 20$) 12	($M_3 = 20$) 12
21	20	12	10	18	($M_0 = 0$) 20	($M_1 = 20$) 18	($M_2 = 20$) 18	($M_3 = 20$) 18
22	20	12	18	10	($M_0 = 0$) 20	($M_1 = 20$) 10	($M_2 = 20$) 10	($M_3 = 20$) 10
23	20	18	10	12	($M_0 = 0$) 20	($M_1 = 20$) 12	($M_2 = 20$) 12	($M_3 = 20$) 12
24	20	18	12	10	($M_0 = 0$) 20	($M_1 = 20$) 10	($M_2 = 20$) 10	($M_3 = 20$) 10

Poichè ci sono 4 elementi, i possibili valori di r sono 4, cioè ci sono 4 possibili algoritmi online A_r, ognuno dei quali seleziona un elemento dall'r-esimo all'ultimo in base alla strategia descritta. Consideriamo ad esempio l'input $(18, 20, 10, 12)$. L'algoritmo A_1, per il quale si ha che $M_0 = 0$, seleziona il primo elemento 18 in quanto questo valore è maggiore di M_0. L'algoritmo A_2, per il quale si ha che $M_1 = 18$, seleziona il secondo elemento 20 in quanto questo valore è maggiore di M_1. L'algoritmo A_3, per il quale si ha che $M_2 = 20$, seleziona l'ultimo elemento 12 in quanto i valori del terzo e quarto elemento sono più piccoli di M_2. Analogamente l'algoritmo A_4 seleziona anch'esso l'ultimo elemento.

Analizzando gli output forniti dai 4 algoritmi, si ha che A_1 e A_4 forniscono la soluzione ottima, cioè il massimo, in 6 casi su 24, quindi forniscono la soluzione ottima con probabilità 0.25, mentre A_2 ed A_3 in 11 casi su 24, quindi con una probabilità di circa 0.458. Per cui, per il caso di $n = 4$, l'algoritmo migliore è A_2 (o anche A_3). La seguente tabella riporta le probabilità di ottenere l'ottimo dei miglior algoritmi per i $n \leq 9$.

n	r	p
3	2	0.5
4	2	$\simeq 0.458$
5	3	$\simeq 0.433$
6	3	$\simeq 0.428$
7	3	$\simeq 0.414$
8	4	$\simeq 0.410$
9	4	$\simeq 0.406$

Procedendo più analiticamente, possiamo calcolare la probabilità P_r che l'algoritmo A_r, per un dato valore di r calcoli il massimo:

$$\begin{aligned}
P_r &= \sum_{i=1}^{n} Pr[A_r \text{ seleziona l'offerta } i \wedge \text{l'offerta } i \text{ è il massimo}] \\
&= \sum_{i=r}^{n} Pr[A_r \text{ seleziona l'offerta } i \wedge \text{l'offerta } i \text{ è il massimo}] \\
&\quad \text{(perchè per } i < r, A_r \text{ non può selezionare il massimo)} \\
&= \sum_{i=r}^{n} Pr[A_r \text{ seleziona l'offerta } i | \text{l'offerta } i \text{ è il massimo}] \cdot Pr[\text{l'offerta } i \text{ è il massimo}] \\
&= \frac{1}{n} \sum_{i=r}^{n} Pr[A_r \text{ seleziona l'offerta } i | \text{l'offerta } i \text{ è il massimo}] \\
&= \frac{1}{n} \sum_{i=r}^{n} \frac{r-1}{i-1} \\
&= \frac{r-1}{n} \sum_{i=r}^{n} \frac{1}{i-1}.
\end{aligned}$$

Si noti che $Pr[A_r \text{ seleziona l'offerta } i \mid \text{l'offerta } i \text{ è il massimo}]=\frac{r-1}{i-1}$ perché, dato che l'offerta i è il massimo, essa verrà selezionata se e solo se non ne viene selezionata una prima, cioè se e solo se il massimo fra le prime $i-1$ offerte si trova tra le prime $r-1$ offerte. Più formalmente se M_{i-1}, cioè il massimo fra i primi $i-1$ elementi, che è anche il valore più grande dopo il massimo (che per ipotesi si trova nella posizione i), si trova nelle prime $r-1$ posizioni, allora l'algoritmo selezionerà il massimo in quanto i valori nelle posizioni $r, r+1, \ldots, i-1$ saranno tutti più piccoli di $M_{r-1} = M_{i-1}$. Se invece M_{i-1} si trova dopo la posizione $r-1$, allora l'algoritmo non selezionerà il massimo in quanto esiste almeno un valore (M_{i-1}) prima di arrivare al massimo, più grande di M_{r-1}. Poichè M_{i-1} può trovarsi in una qualsiasi delle prime $i-1$ posizioni, ci sono $r-1$ casi favorevoli su $i-1$ casi totali. Facciamo un esempio con $r = 5$, $i = 9$ per chiarire meglio. Consideriamo la sequenza

$$2, 15, 7, 8, 6, 4, 12, 13, 18, 10.$$

In questo caso il massimo, 18, si trova nella posizione $i = 9$, $M_{i-1} = M_{r-1} = 15$. Pertanto A_r scarterà $6, 4, 12$ e 13 perchè tutti minori di 15 e selezionerà il 18 che è maggiore di 15. Questo è un caso favorevole in cui l'algoritmo sceglie il massimo.

Consideriamo adesso la sequenza

$$2, 12, 7, 8, 6, 4, 15, 13, 18, 10.$$

In questo caso $M_{i-1} = 15$, e $M_{r-1} = 12$. Pertanto A_r scarterà 6 e 4 perchè tutti minori di 12 ma selezionerà il 15 che è maggiore di 12. Questo è un caso sfavorevole in cui l'algoritmo non sceglie il massimo.

La funzione $\frac{r-1}{n} \sum_{i=r}^{n} \frac{1}{i-1}$ è massimizzata per $r = n/e$.

5.8 *Note bibliografiche*

Il meteriale presentato in questo capitolo è tratto da varie fonti.

5.9 *Esercizi*

1. Consideriamo il problema dell'affitto degli sci per il quale abbiamo analizzato l'algoritmo online con $k = s$. Mostrare che le strategie con $k > s$ e $k < s$ portano ad algoritmi online peggiori rispetto a quello per $k = s$.

2. Consideriamo il seguente problema: abbiamo investito una somma di denaro comprando N azioni di una società quotata in borsa. Ora dobbiamo rivenderle e, semplificando molto ciò che accade in realtà, assumiamo che ogni giorno il valore delle azioni cambi, quindi abbiamo una sequenza $v_1, v_2, \ldots, v_n$ di valori. Per ogni giorno $i = 1, 2, \ldots, n$ dobbiamo decidere quante delle azioni rimanenti vendere. Fornire e analizzare un algoritmo online per tale problema.

3. Nella prova della k-competitività dell'algoritmo LRU abbiamo dimostrato che una generica fase $F_i, i > 0$, quindi dalla seconda in poi, è necessario avere almeno k richieste a k pagine $\{A_1, A_2, \ldots, A_k\}$ non presenti nella memoria. La prova richiedeva di dimostrare che ci sono sempre k richieste a pagine diverse da P, dove P è l'ultima pagina chiesta nella fase precedente F_{i-1}. Il caso in cui $\{P, A_1, A_2, \ldots, A_k\}$ non contiene duplicati è semplice. Il caso in cui invece $\{P, A_1, A_2, \ldots, A_k\}$ contiene duplicati è diviso in due sottocasi:

   (a) P è duplicato, cioè $P = A_i$ per un qualche i.

   (b) P non è duplicato, quindi $P \neq Q = A_i = A_j$, per $i \neq j$.

   Si faccia un esempio di queste due situazioni. Si consideri $k = 4$.

4. Si consideri il seguente algoritmo di paging FWF (Flush When Full): quando c'è un page fault, se c'è un posto vuoto nella cache lo si utilizza, altrimenti si svuota la cache generando k posti liberi. Si commenti la strategia FWF e si fornisca un'analisi dell'algoritmo.

5. Provare che la strategia FIFO è k-competitiva.

6. L'algoritmo FIFO soffre della seguente anomalia: esistono delle sequenze di input per le quali il numero di page fault con una cache più grande è maggiore di quello che si avrebbe con una cache più piccola. Fornire una sequenza di input in cui tale anomalia si verifica passando da una cache di grandezza $k = 3$ and una cache di grandezza $k = 4$.

7. Mostrare che data una lista di elementi, è possibile ottenere un qualsiasi ordinamento effetuando esclusivamente scambi di elementi in posizioni adiacenti.

8. Provare che la strategia LIFO non è c-competitiva per nessun valore di c.

9. Per gli algoritmi di gestione di una lista online abbiamo considerato delle operazioni senza costo: quando si accede ad un elemento possiamo spostarlo in una qualsiasi posizione più vicina alla testa della lista. Si dia una giustificazione. Perchè non permettiamo di spostarlo verso la coda della lista?

10. Siano $f_1 \geq f_2 \geq \ldots \geq f_n$ delle frequenze di n elementi tali che $\sum_{i=1}^{n} f_i = m$. Per semplicità assumiamo che m sia un multiplo di n. Provare che $\sum_{i=1}^{n} i f_i \leq \frac{m(n+1)}{2}$.

11. Si consideri il problema del bin packing in una versione che non richiede di "chiudere" un contenitore prima di usarne un altro. Si dia un algoritmo online per questa versione che si comporti meglio di NEXTFIT.