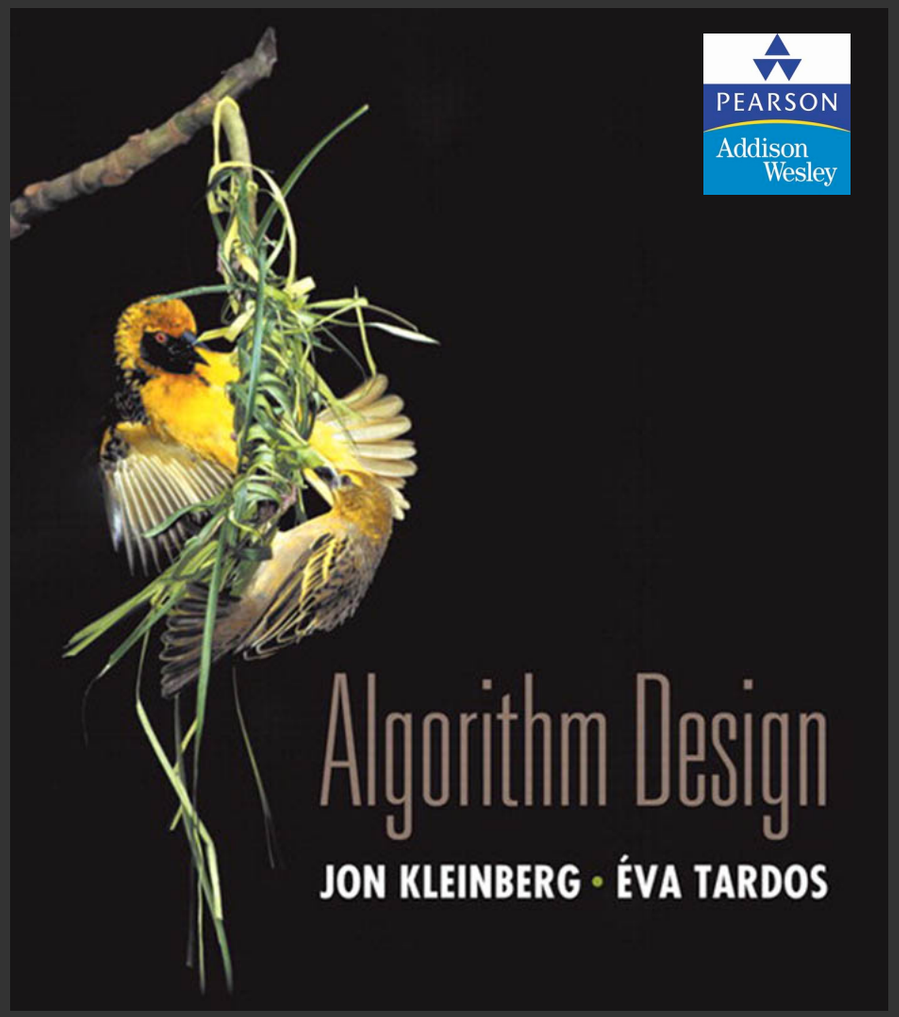


4. Algoritmi Avidi I

- *resto in monete*
- *schedulazione di intervalli*
- *partizionamento di intervalli*
- *schedulazione per minimizzare il ritardo*
- *caching ottimo*

Traduzione e adattamento di Vincenzo Bonifaci
Original Lecture slides by Kevin Wayne

http://www.cs.princeton.edu/~wayne/kleinberg-tardos

Last updated on 01/04/2020 16:26

4. ALGORITMI AVIDI I

- *resto in monete*
- *interval scheduling*
- *interval partitioning*
- *scheduling to minimize lateness*
- *optimal caching*

Problema del resto in monete

Scopo. Dati i valori delle monete statunitensi { 1, 5, 10, 25, 100 }, escogitare un metodo per pagare un dato ammontare al cliente utilizzando il minor numero possibile di monete.

Es. 34¢.

Algoritmo del cassiere. Ad ogni iterazione, aggiungiamo la moneta di valore massimo che non ci porta oltre l'ammontare da pagare.

Es. $2.89.

Algoritmo del cassiere

Ad ogni iterazione, aggiungiamo la moneta di valore massimo che non ci porta oltre l'ammontare da pagare.

ALGORITMO-CASSIERE $(x, c_1, c_2, \ldots, c_n)$

ORDINA gli n valori di moneta in modo che $0 < c_1 < c_2 < \ldots < c_n$.

$S \leftarrow \emptyset$. ← multiinsieme di monete selezionate

WHILE $(x > 0)$

  $k \leftarrow$ moneta a valore massimo c_k tale che $c_k \leq x$.

  IF (tale k non esiste)

    RETURN *"non c'è soluzione."*

  ELSE

    $x \leftarrow x - c_k$.

    $S \leftarrow S \cup \{ k \}$.

RETURN S.

Algoritmi avidi I: quiz 1

L'algoritmo del cassiere è ottimo?

A. Sì, gli algoritmi avidi sono sempre ottimi.

B. Sì, qualunque siano i valori delle monete $c_1 < c_2 < \ldots < c_n$ purché $c_1 = 1$.

C. Sì, a causa di particolari proprietà dei valori delle monete USA.

D. No.

Algoritmo del cassiere (per valori di moneta arbitrari)

D. L'algoritmo del cassiere è ottimo per qualunque valore delle monete?

R. No. Si considerino i francobolli USA: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

- Algoritmo del cassiere: 140¢ = 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1.
- Ottimo: 140¢ = 70 + 70.

R. No. Può non generare una soluzione ammissibile se $c_1 > 1$: 7, 8, 9.

- Algoritmo del cassiere: 15¢ = 9 + ?.
- Ottimo: 15¢ = 7 + 8.

Proprietà di qualunque soluzione ottima (per monete USA)

Proprietà. Numero di centesimi ≤ 4.

Dim. 5 centesimi sono sostituibili da 1 nichelino.

Proprietà. Numero di nichelini ≤ 1.

Proprietà. Numero di quartini ≤ 3.

Proprietà. Numero di nichelini + numero di decini ≤ 2.

Dim.

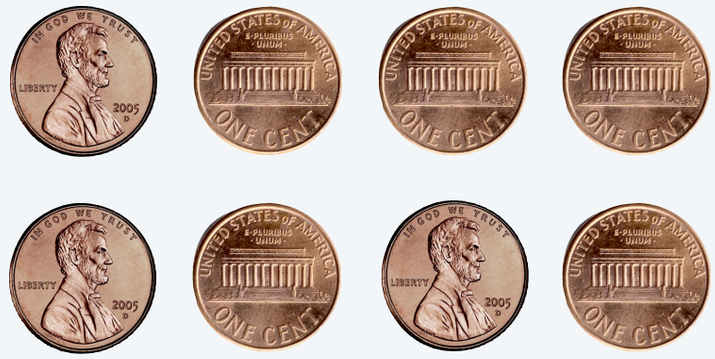
- Ricordiamo: ≤ 1nichelino.
- Cambiamo 3 decini e 0 nichelini con 1 quartino e 1 nichelino;
- Cambiamo 2 decini e 1 nichelino con 1 quartino.

dollari (100¢)

quartini (25¢)

decini (10¢)

nichelini (5¢)



centesimi (1¢)

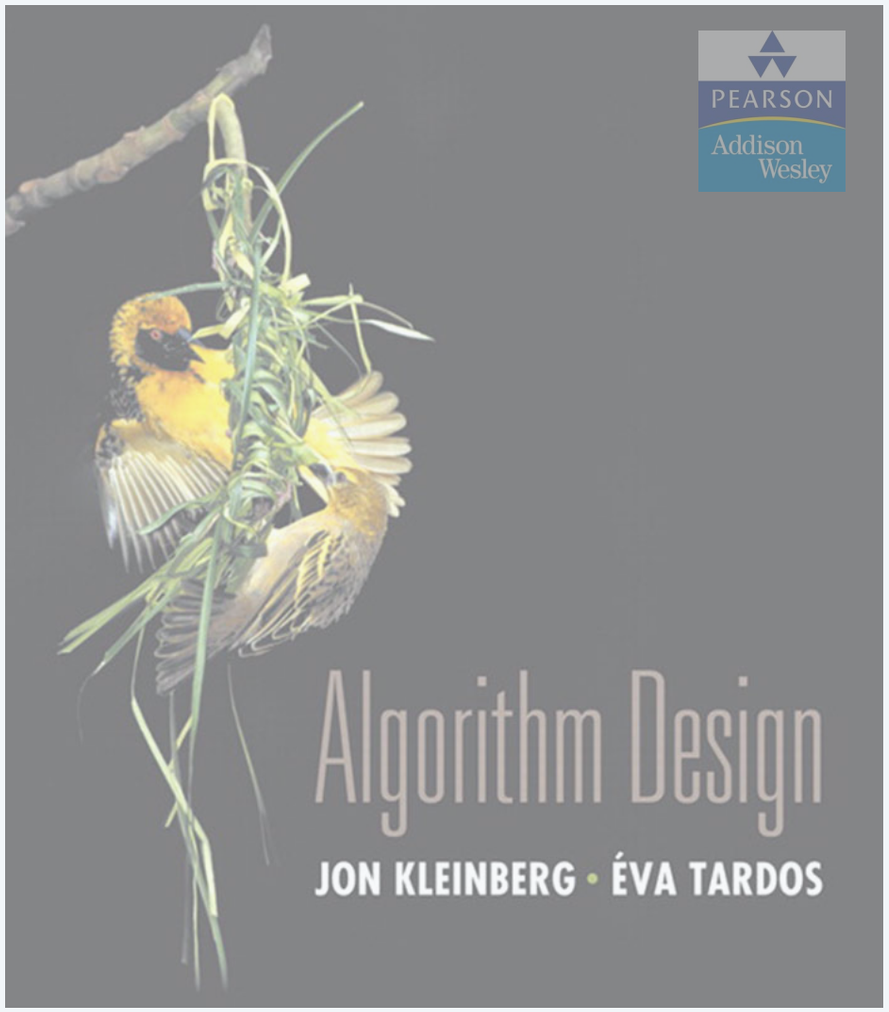
Ottimalità dell'algoritmo del cassiere (per monete USA)

Teorema. L'algoritmo è ottimo per le monete USA { 1, 5, 10, 25, 100 }.

Dim. [per induzione sull'ammontare da pagare x]

- Consideriamo modo ottimo per cambiare $c_k \le x < c_{k+1}$: l'approccio avido sceglie la moneta k.
- Affermiamo che ogni soluzione ottima deve usare la moneta k.
  - se no, richiede abbastanza monete $c_1, \ldots, c_{k-1}$ che sommino ad x
  - la tabella sottostante mostra che nessuna soluzione ottima può farlo
- Il problema diventa quello di cambiare $x - c_k$ centesimi, che, per induzione, è risolto in modo ottimo dall'algoritmo del cassiere. ▪

k	c_k	tutte le soluzioni ottime soddisfano	massimo valore totale delle monete $c_1, c_2, \ldots, c_{k-1}$ in ogni soluzione ottima
1	1	$C \le 4$	–
2	5	$N \le 1$	4
3	10	$N + D \le 2$	4 + 5 = 9
4	25	$Q \le 3$	20 + 4 = 24
5	100	*nessun limite*	75 + 24 = 99



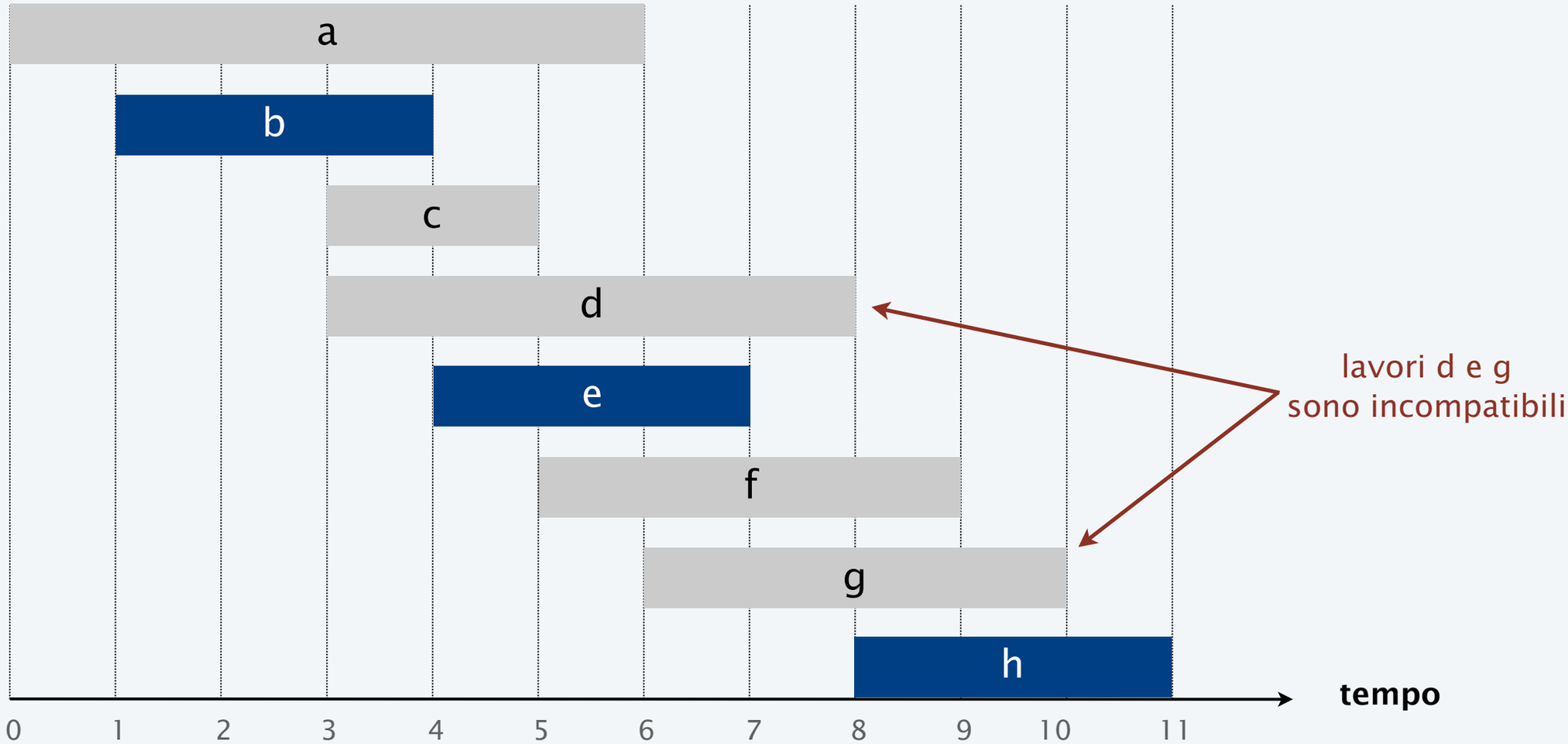
SECTION 4.1

4. ALGORITMI AVIDI I

- *coin changing*
- *schedulazione di intervalli*
- *interval partitioning*
- *scheduling to minimize lateness*
- *optimal caching*

Schedulazione di intervalli

- Lavoro j inizia ad s_j e finisce a f_j.
- Due lavori sono compatibili se non si sovrappongono.
- Scopo: trovare un sottoinsieme a cardinalità massima di lavori mutuamente compatibili.



Algoritmi avidi I: quiz 2

Considerati i lavori in qualche ordine, accettiamo ciascun lavoro se esso è compatibile con quelli già selezionati. Quale regola è ottimale?

A. [Minimo tempo di inizio] Consideriamo i lavori in ordine crescente di s_j.

B. [Minimo tempo di fine] Consideriamo i lavori in ordine crescente di f_j.

C. [Intervallo più breve] Consideriamo i lavori in ordine crescente di $f_j - s_j$.

D. Nessuna delle precedenti.

Schedulazione di intervalli: algoritmo del minimo tempo di fine

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

ORDINA lavori per tempo di fine in modo che $f_1 \leq f_2 \leq \ldots \leq f_n$.

$S \leftarrow \emptyset$. ⟵ insieme dei lavori scelti

FOR $j = 1$ TO n

  IF (lavoro j è compatibile con S)

    $S \leftarrow S \cup \{ j \}$.

RETURN S.

Proposizione. Possiamo implementare l'algoritmo in tempo $O(n \log n)$.

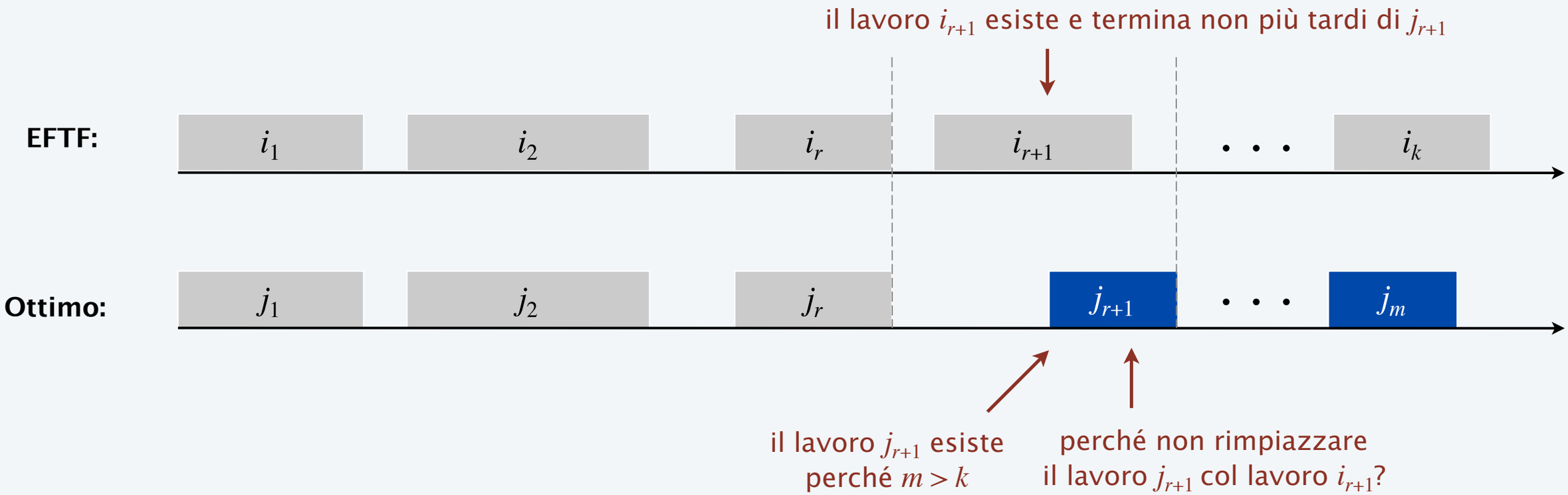
- Teniamo traccia dell'ultimo lavoro j^* aggiunto ad S.
- Il lavoro j è compatibile con S sse $s_j \geq f_{j^*}$.
- L'ordinamento in base al tempo di fine prende tempo $O(n \log n)$.

Schedulazione di intervalli: analisi di Earliest-Finish-Time-First

Teorema. L'algoritmo earliest-finish-time-first è ottimo.

Dim. [per assurdo]

- Assumiamo che non lo sia.
- Siano i_1, i_2, ... i_k i lavori selezionati dall'algoritmo.
- Siano j_1, j_2, ... j_m i lavori selezionati in una soluzione ottima con $i_1 = j_1, i_2 = j_2, ..., i_r = j_r$ per un valore di r il più grande possibile.

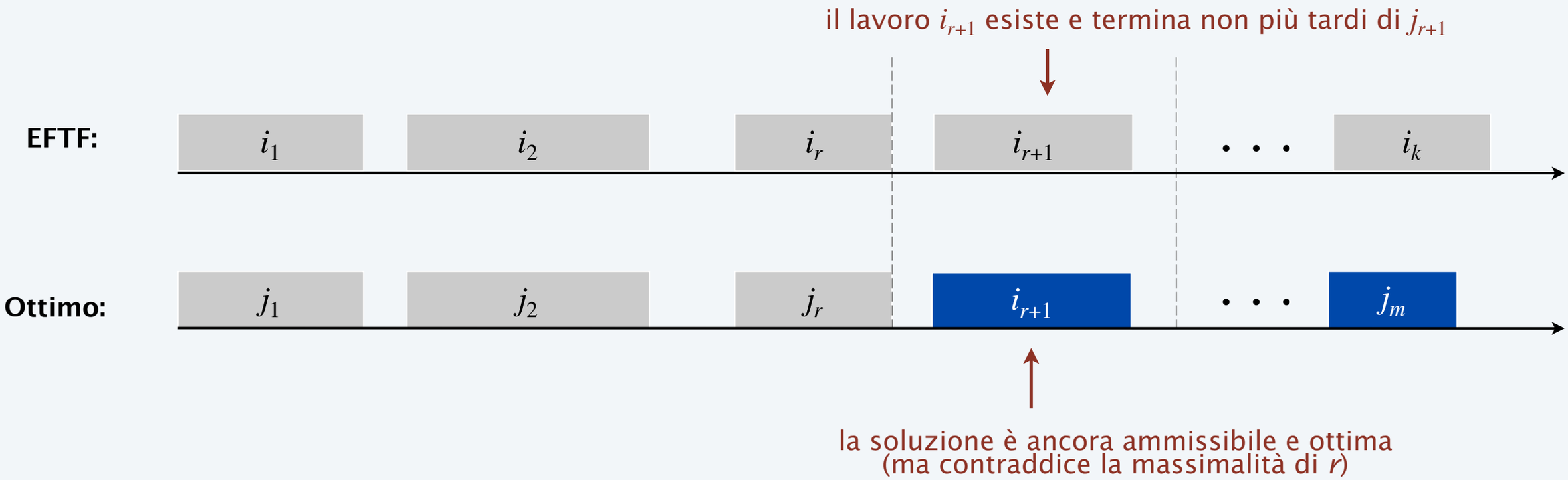


Schedulazione di intervalli: analisi di Earliest-Finish-Time-First

Teorema. L'algoritmo earliest-finish-time-first è ottimo.

Dim. [per assurdo]

- Assumiamo che non lo sia.
- Siano i_1, i_2, ... i_k i lavori selezionati dall'algoritmo.
- Siano j_1, j_2, ... j_m i lavori selezionati in una soluzione ottima con $i_1 = j_1, i_2 = j_2, ..., i_r = j_r$ per un valore di r il più grande possibile.



Algoritmi avidi I: quiz 3

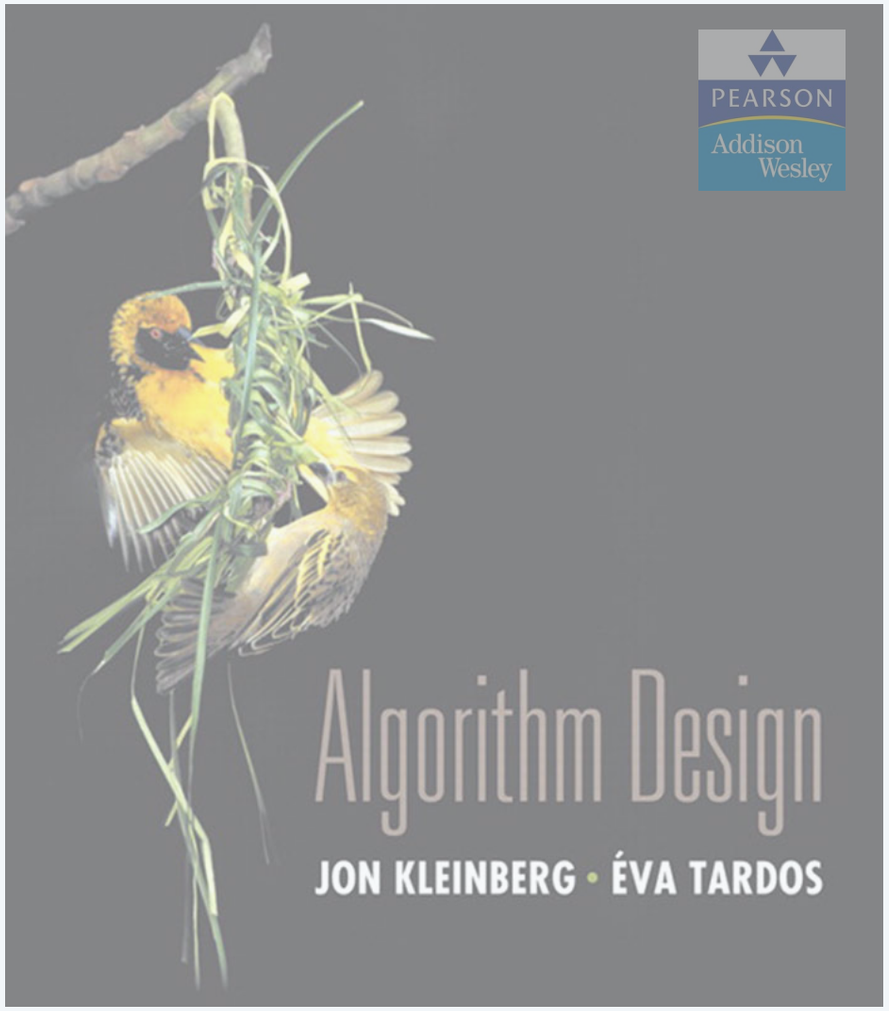
Supponiamo che ogni lavoro abbia anche un peso positivo e lo scopo sia di trovare un sottoinsieme a peso massimo di intervalli mutuamente compatibili. L'algoritmo earliest-finish-time-first è ancora ottimo?

A. Sì, perché gli algoritmi avidi sono sempre ottimi.

B. Sì, perché vale la stessa dimostrazione di correttezza.

C. No, perché la stessa dimostrazione di correttezza non è più valida.

D. No, perché si può assegnare un peso enorme ad un lavoro che si sovrappone con quello dal tempo di fine minimo.



Intermezzo: Coda di Priorità

Section 2.5

Collezioni: pile, code, code di priorità

Collezioni. Inserimento e cancellazione di elementi. Quale elemento si può cancellare?

Pila. Rimuovi l'elemento inserito per ultimo.
Coda. Rimuovi l'elemento inserito per primo.

Coda di priorità. Rimuovi l'elemento massimo (in alternativa: l'elemento minimo).

Generalizza sia una pila che una coda.

operation	*argument*	*return value*
insert	P	
insert	Q	
insert	E	
remove max		Q
insert	X	
insert	A	
insert	M	
remove max		X
insert	P	
insert	L	
insert	E	
remove max		P

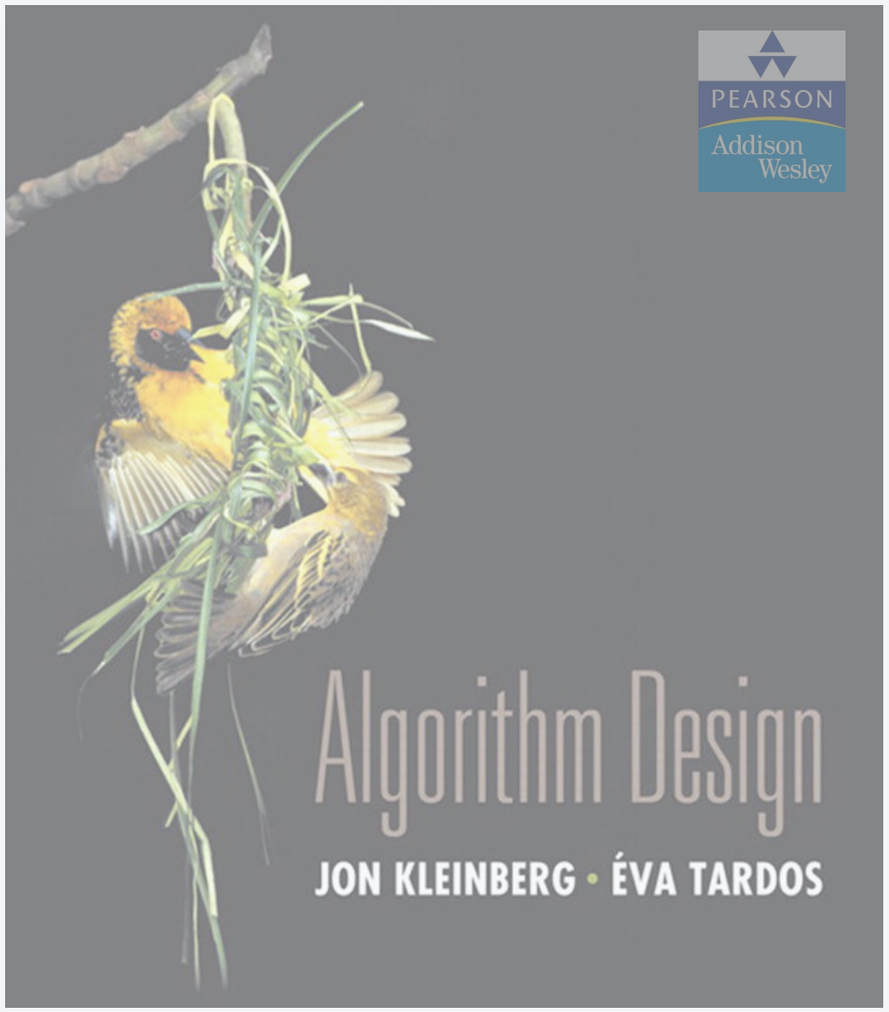
Coda di priorità: operazioni supportate

Requisito. Gli elementi della coda devono poter esser confrontati tra loro (attraverso una relazione d'ordine).

operazione	descrizione	tempo di esecuzione per una coda di n elementi (implementazione tramite heap binario)
IS-EMPTY()	la coda è vuota?	$O(1)$
FIND-MIN()	restituisci l'elemento minimo	$O(1)$
INSERT(x)	inserisci un elemento x	$O(\log n)$
DELETE-MIN()	restituisci l'elemento minimo e cancellalo dalla coda	$O(\log n)$

Nota. Elementi duplicati sono permessi; DELETE-MIN() cancella un elemento minimo non meglio specificato.

Nota. A volte sono richieste ulteriori operazioni (p.e.: modifica degli elementi), ma non tutte le implementazioni delle code di priorità le supportano.



SECTION 4.1

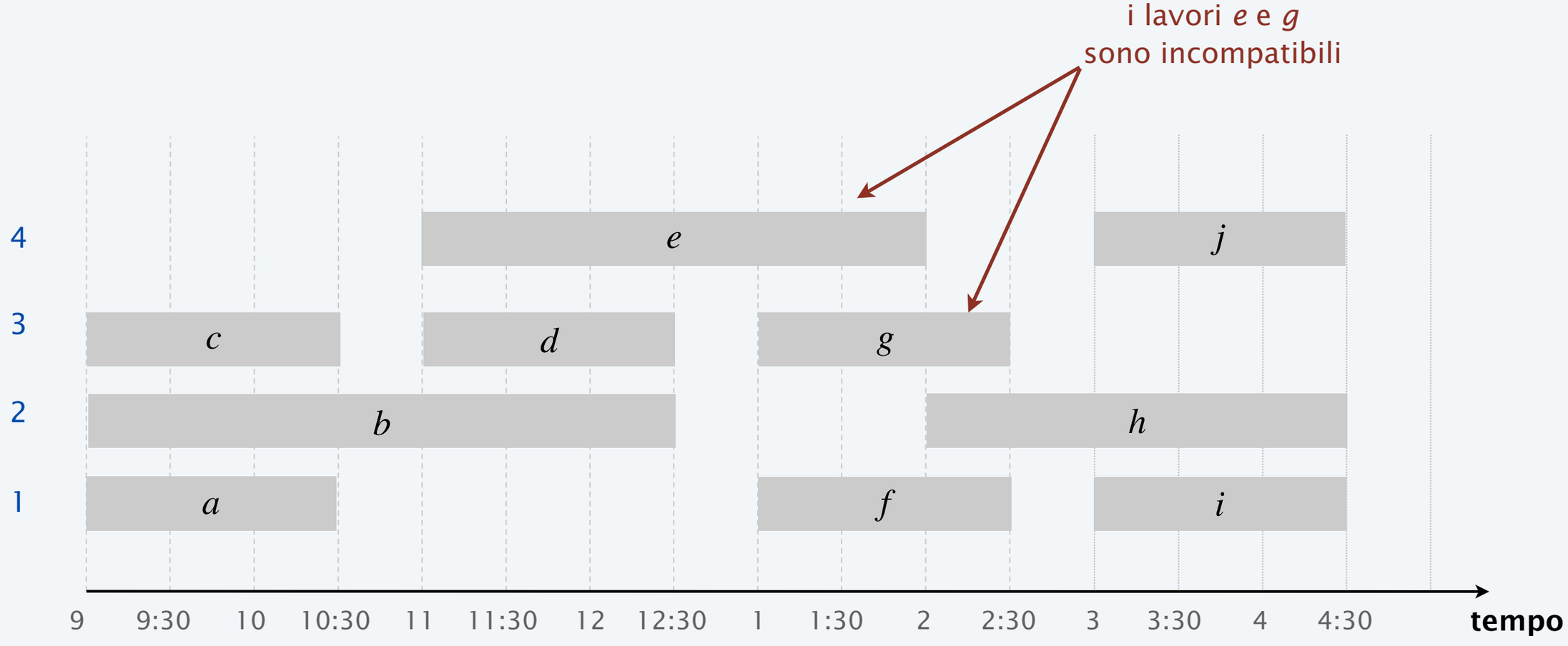
4. ALGORITMI AVIDI I

- *coin changing*
- *interval scheduling*
- *partizionamento di intervalli*
- *scheduling to minimize lateness*
- *optimal caching*

Partizionamento di intervalli

- La lezione j inizia alle ore s_j e finisce alle ore f_j.
- Scopo: trovare il minimo numero di aule per programmare tutte le lezioni in modo che nessun paio di lezioni impegni la stessa aula nello stesso momento.

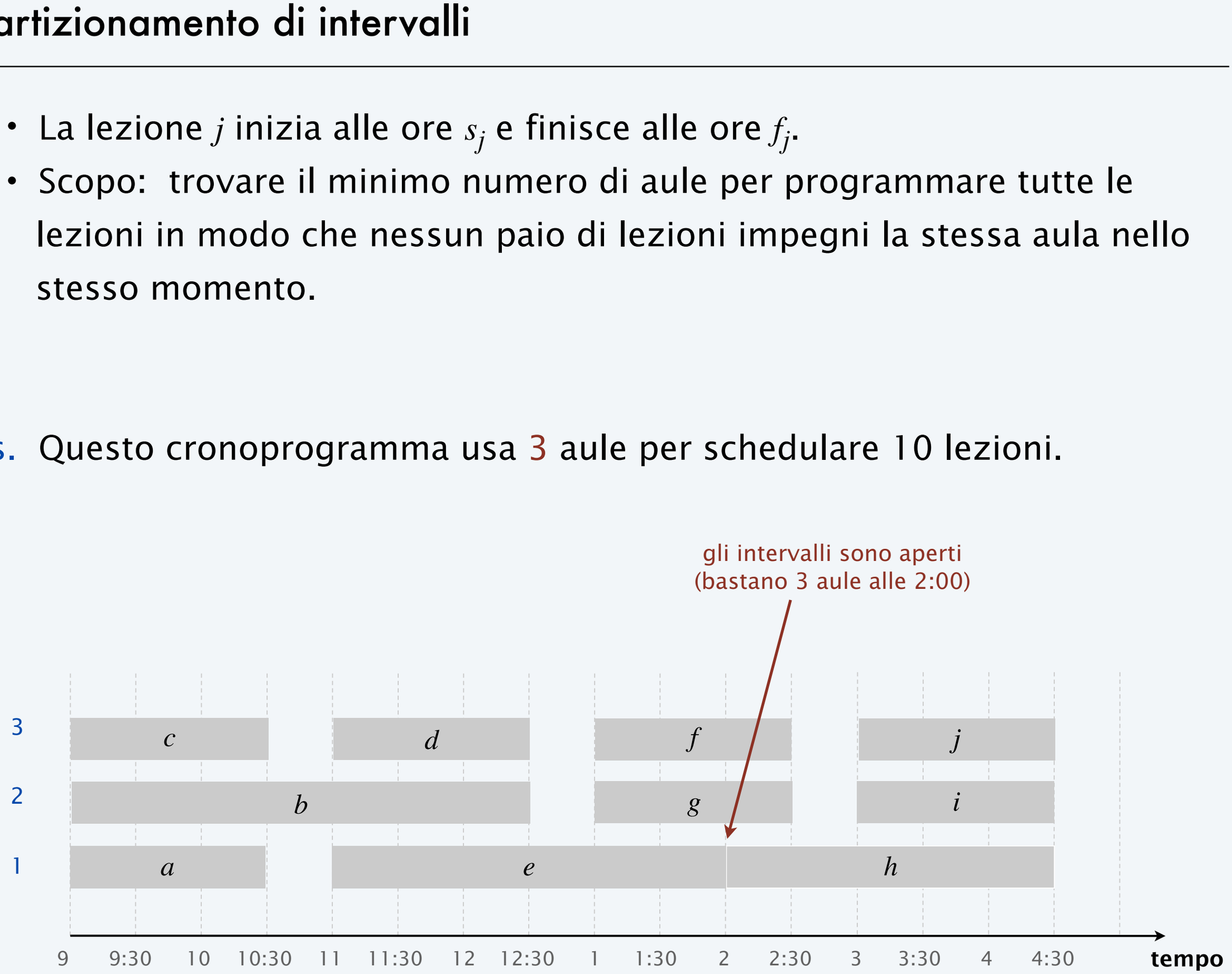
Es. Questo cronoprogramma usa 4 aule per schedulare 10 lezioni.



Partizionamento di intervalli

- La lezione j inizia alle ore s_j e finisce alle ore f_j.
- Scopo: trovare il minimo numero di aule per programmare tutte le lezioni in modo che nessun paio di lezioni impegni la stessa aula nello stesso momento.

Es. Questo cronoprogramma usa 3 aule per schedulare 10 lezioni.



Algoritmi avidi I: quiz 4

Consideriamo le lezioni in un ordine appropriato, e assegnamo ogni lezione alla prima aula disponibile (allocando una nuova aula se nessuna è disponibile). Quale regola è ottima?

A. [Minimo tempo di inizio] Ordine crescente di s_j.

B. [Minimo tempo di fine] Ordine crescente di f_j.

C. [Intervallo più breve] Ordine crescente di $f_j - s_j$.

D. Nessuna delle precedenti.

Partizionamento di intervalli: algoritmo earliest-start-time-first

EARLIEST-START-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

ORDINA lezioni per tempo di inizio in modo che $s_1 \leq s_2 \leq \ldots \leq s_n$.

$d \leftarrow 0$. ← numero di aule allocate

FOR $j = 1$ TO n

  IF (la lezione j è compatibile con qualche aula)

    Programma la lezione j in una aula siffatta k.

  ELSE

    Alloca una nuova aula $d + 1$.

    Programma la lezione j nell'aula $d + 1$.

    $d \leftarrow d + 1$.

RETURN il cronoprogramma.

Partizionamento di intervalli: algoritmo earliest-start-time-first

Proposizione. L'algoritmo earliest-start-time-first può essere implementato in tempo $O(n \log n)$ time.

Dim. Manteniamo le aule in una coda di priorità (chiave = tempo di fine dell'ultima lezione).

- Per sapere se la lezione j è compatibile con qualche aula, confrontiamo s_j con la chiave dell'aula k in cima alla coda.
- Per aggiungere la lezione j all'aula k, impostiamo la chiave dell'aula k a f_j.
- Il numero totale di operazioni sulla coda di priorità è $O(n)$.
- L'ordinamento in base ai tempi di inizio prende tempo $O(n \log n)$ time. ∎

Nota. Questa implementazione sceglie un'aula k il cui tempo di fine dell'ultima lezione è il minimo.

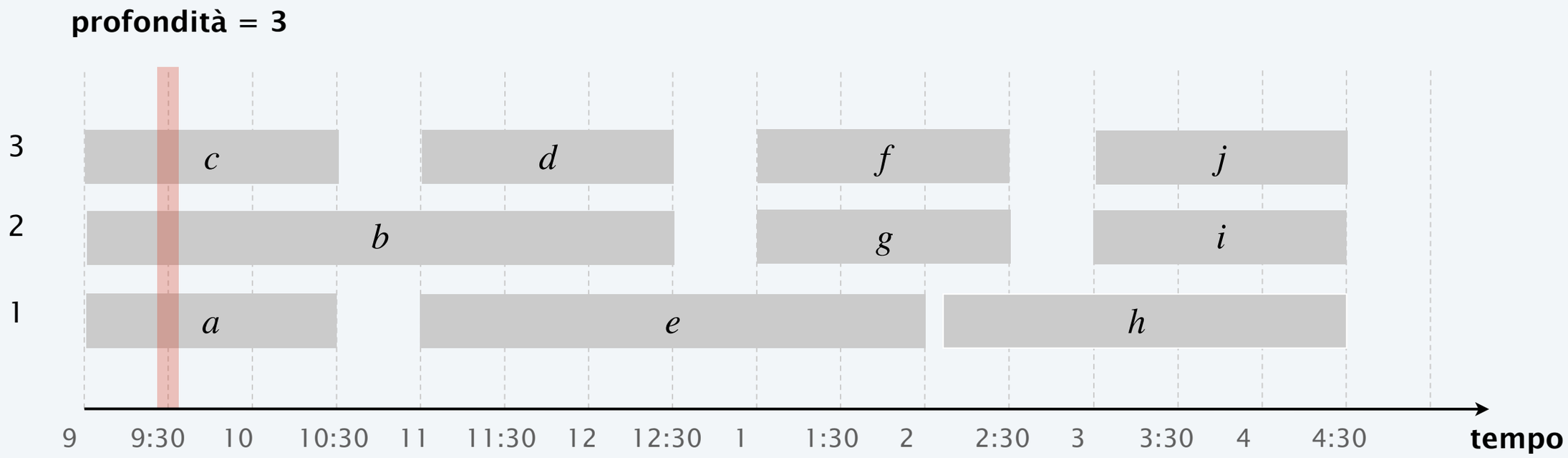
Partizionamento di intervalli: minorazione della soluzione ottima

Def. La profondità di un insieme di intervalli aperti è il massimo numero di intervalli che contengono un qualunque punto.

Osservazione chiave. Numero di aule necessarie $\geq$ profondità.

D. Il numero minimo di aule necessarie è sempre uguale alla profondità?

R. Sì! Inoltre, l'algoritmo earliest-start-time-first determina un cronoprogramma in cui il numero di aule utilizzate è pari alla profondità.



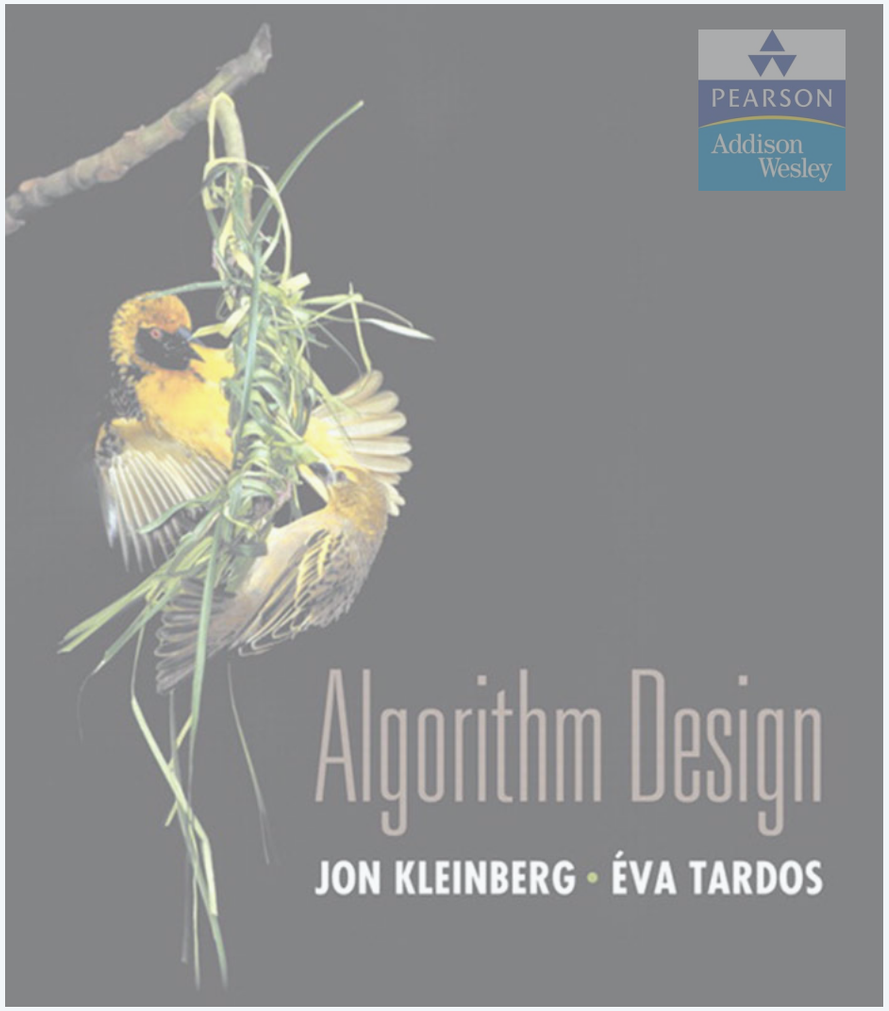
Partizionamento di intervalli: analisi di earliest-start-time-first

Osservazione. L'algoritmo earliest-start-time first non programma mai due lezioni incompatibili nella stessa aula.

Teorema. L'algoritmo earliest-start-time-first è ottimo.

Dim.

- Sia d = numero di aule allocate dall'algoritmo.
- L'aula d è stata allocata per programmare una lezione, sia essa j, incompatibile con una lezione in ciascuna delle $d - 1$ altre aule.
- Quindi, queste d lezioni finiscono tutte dopo s_j.
- Per l'ordinamento sui tempi, ciascuna di queste lezioni incompatibili inizia non più tardi di s_j.
- Quindi, abbiamo d lezioni sovrapposte all'istante $s_j + \varepsilon$.
- Osservazione chiave $\Rightarrow$ qualunque soluzione usa $\geq d$ aule. ▪



SECTION 4.2

4. ALGORITMI AVIDI I

- *coin changing*
- *interval scheduling*
- *interval partitioning*
- *schedulazione per minimizzare il ritardo*
- *optimal caching*

Schedulazione per minimizzare il ritardo

- Una risorsa che processa un lavoro alla volta.
- Il lavoro j richiede t_j unità di tempo e va consegnato entro il tempo d_j.
- Se j inizia all'istante s_j, finirà all'istante $f_j = s_j + t_j$.
- *Lateness [ritardo]*: $\ell_j = \max \{ 0, f_j - d_j \}$.
- Scopo: schedulare tutti i lavori minimizzando la massima lateness $L = \max_j \ell_j$.

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

Algoritmi avidi I: quiz 5

Scheduliamo i lavori secondo un ordine appropriato. Quale ordine minimizza la lateness massima?

A. [minimo tempo di processamento] Ordine crescente di t_j.

B. [minima scadenza] Ordine crescente di scadenza d_j.

C. [minimo slack] Ordine crescente di slack: $d_j - t_j$.

D. Nessuna delle precedenti.

ıto

Minimizzazione della lateness: earliest deadline first

EARLIEST-DEADLINE-FIRST $(n, t_1, t_2, \ldots, t_n, d_1, d_2, \ldots, d_n)$

ORDINA i lavori per scadenza in modo che $d_1 \leq d_2 \leq \ldots \leq d_n$.

$t \leftarrow 0$.

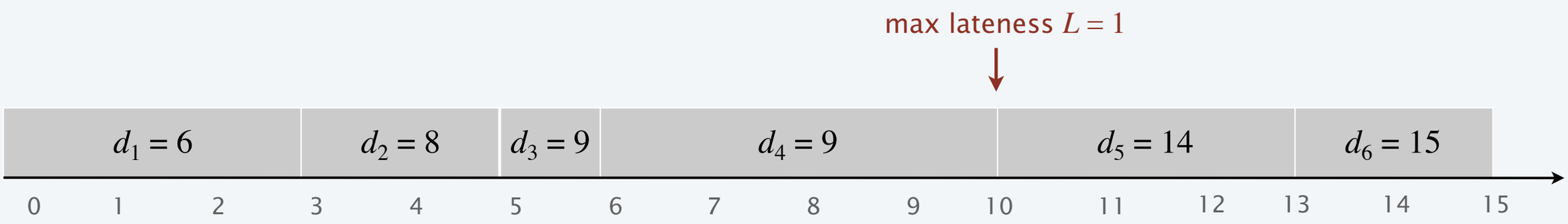
FOR $j = 1$ TO n

  Assegna il lavoro j all' intervallo $[t, t + t_j]$.

  $s_j \leftarrow t$; $f_j \leftarrow t + t_j$.

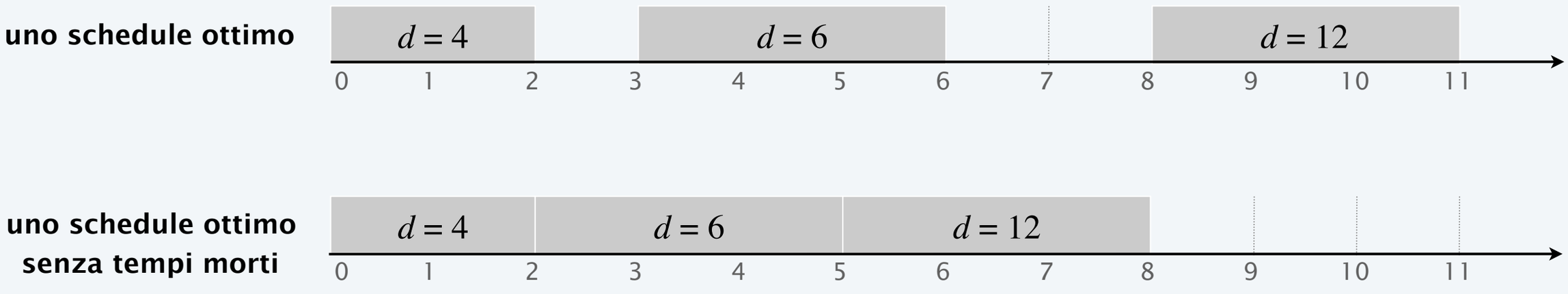
  $t \leftarrow t + t_j$.

RETURN intervalli $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$.



Minimizzazione della lateness: assenza di tempi morti

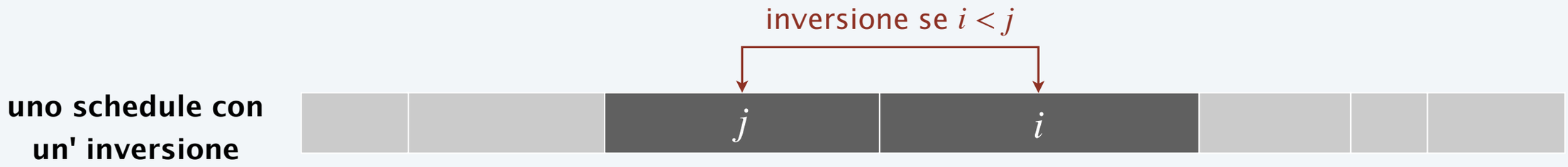
Osservazione 1. Esiste uno schedule ottimo senza tempi morti.



Osservazione 2. Lo schedule costruito da earliest-deadline-first non ha tempi morti.

Minimizzazione della lateness: inversioni

Def. Dato uno schedule S, un' inversione è una coppia di lavori i e j tale che: $i < j$ ma j è schedulato prima di i.



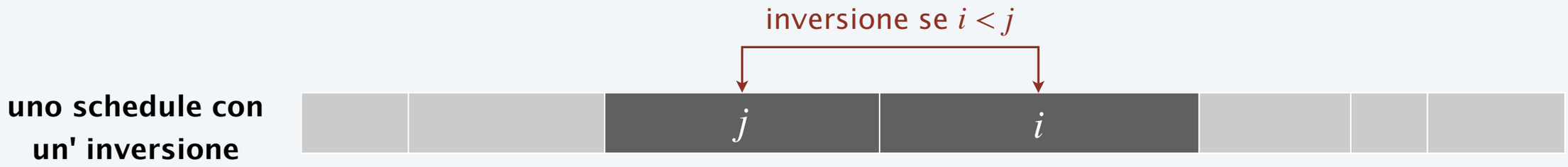
ricordare: i lavori sono stati numerati in modo che $d_1 \leq d_2 \leq \ldots \leq d_n$

Osservazione 3. Lo schedule di earliest-deadline-first è l'unico schedule senza tempi morti che non ha inversioni.

1	2	3	4	5	6	…	n

Minimizzazione della lateness: inversioni

Def. Dato uno schedule S, un' inversione è una coppia di lavori i e j tale che: $i < j$ ma j è schedulato prima di i.



ricordare: i lavori sono stati numerati in modo che $d_1 \leq d_2 \leq \ldots \leq d_n$

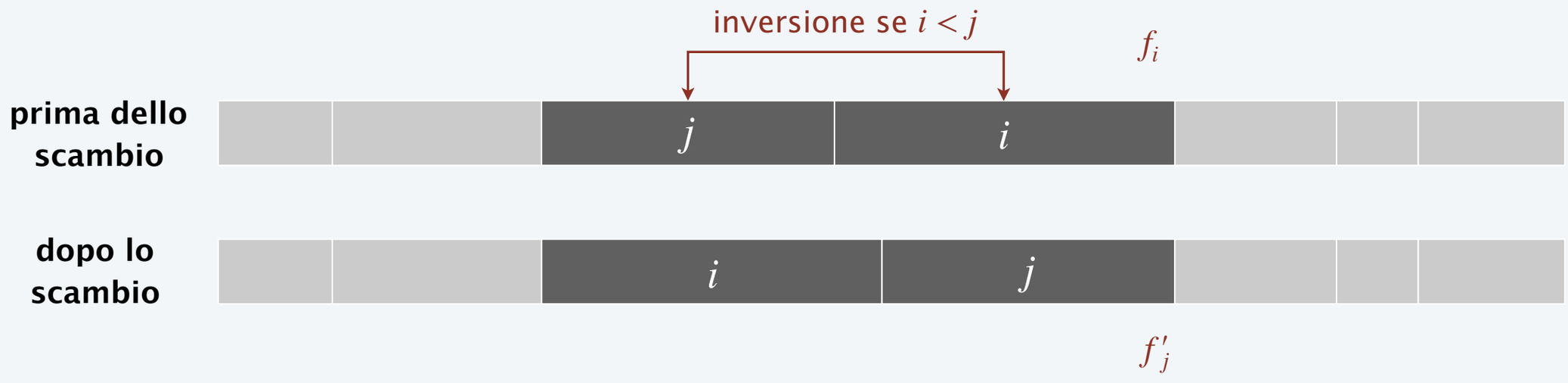
Osservazione 4. Se uno schedule senza tempi morti ha un'inversione, allora esso ha un'inversione adiacente.

Dim. due lavori invertiti schedulati uno dopo l'altro

- Sia i– j un'inversione con lavori il più vicini possibile.
- Sia k un elemento immediatamente alla destra di j.
- Caso 1. [$j > k$] Allora j–k è un' inversione adiacente.
- Caso 2. [$j < k$] Allora i–k è un' inversione più vicina poiché $i < j < k$. ※

Minimizzazione della lateness: inversioni

Def. Dato uno schedule S, un' inversione è una coppia di lavori i e j tale che: $i < j$ ma j è schedulato prima di i.



Enunciato chiave. Scambiando due lavori invertiti adiacenti i e j si riduce il numero di inversioni di 1, senza aumentare la lateness massima.

Dim. Sia ℓ la lateness prima dello scambio, e ℓ' quella dopo lo scambio.

- $\ell'_k = \ell_k$ per ogni $k \neq i, j$.
- $\ell'_i \leq \ell_i$.
- Se j è in ritardo,

$$\begin{aligned} \ell'_j &= f'_j - d_j && \longleftarrow \text{definizione} \\ &= f_i - d_j && \longleftarrow j \text{ ora finisce all'istante } f_i \\ &\leq f_i - d_i && \longleftarrow i < j \Rightarrow d_i \leq d_j \\ &\leq \ell_i . && \longleftarrow \text{definizione} \end{aligned}$$

Analisi dell'algoritmo earliest-deadline-first

Teorema. Lo schedule S di earliest-deadline-first è ottimo.

schedule ottimi possono avere inversioni

Dim. [per contraddizione]

Sia S^* uno schedule ottimo col numero minimo di inversioni.

- Possiamo assumere che S^* non abbia tempi morti. ← Osservazione 1
- Caso 1. [S^* non ha inversioni] Allora $S = S^*$. ← Osservazione 3
- Case 2. [S^* ha almeno un' inversione]
  - sia i–j un' inversione adiacente ← Osservazione 4
  - scambiare i lavori i e j diminuisce il numero di inversioni di 1 senza aumentare la lateness massima ← enunciato chiave
  - ciò contraddice la definizione di S^* come schedule ottimo con un numero minimo di inversioni. ※

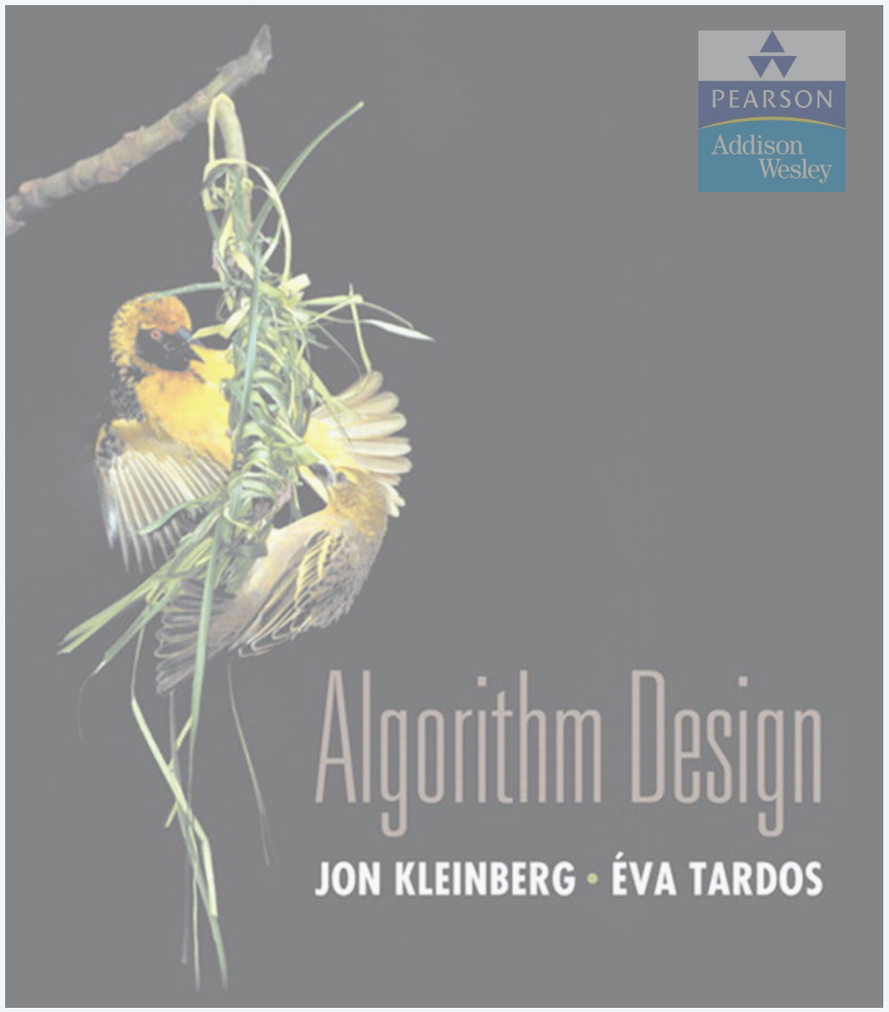
Tipi di analisi di strategie avide

Algoritmo sempre in vantaggio. Mostrare che dopo ogni passo dell'algoritmo avido, la sua soluzione è almeno buona quanto quella di qualunque altro algoritmo.

Strutturale. Scoprire una limitazione "strutturale" che dica che ogni possibile soluzione deve avere (almeno/al più) un certo valore. Poi mostrare che l'algoritmo avido in esame ha questo valore.

Argomentazione basata su scambi. Trasformare gradualmente qualunque soluzione in quella trovata dall'algoritmo avido senza danneggiare la sua qualità.

Altri algoritmi avidi. Gale–Shapley, Kruskal, Prim, Dijkstra, Huffman, ...



SECTION 4.3

4. ALGORITMI AVIDI I

- *coin changing*
- *interval scheduling*
- *interval partitioning*
- *scheduling to minimize lateness*
- *caching ottimo*

Caching offline ottimo

Caching.

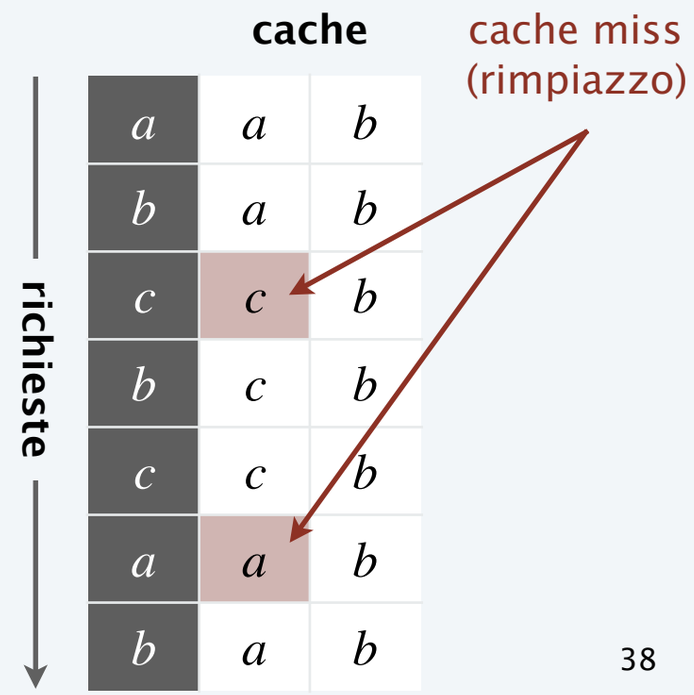
- Memoria cache con la capacità di mantenere k elementi.
- Sequenza di m richieste di elementi $d_1, d_2, \ldots, d_m$.
- *Cache hit*: elemento è nella cache quando è stato richiesto.
- *Cache miss*: elemento non è nella cache quando è stato richiesto.
  (occorre eliminare qualche altro elemento dalla cache e portare l'elemento richiesto nella cache)

Applicazioni. CPU, RAM, disco rigido, web, browser, ...

Scopo. Schedulare i rimpiazzi per minimizzare il numero di miss.

Es. $k = 2$, cache iniziale = ab, richieste: a, b, c, b, c, a, b.

Schedule ottimo. 2 rimpiazzi.

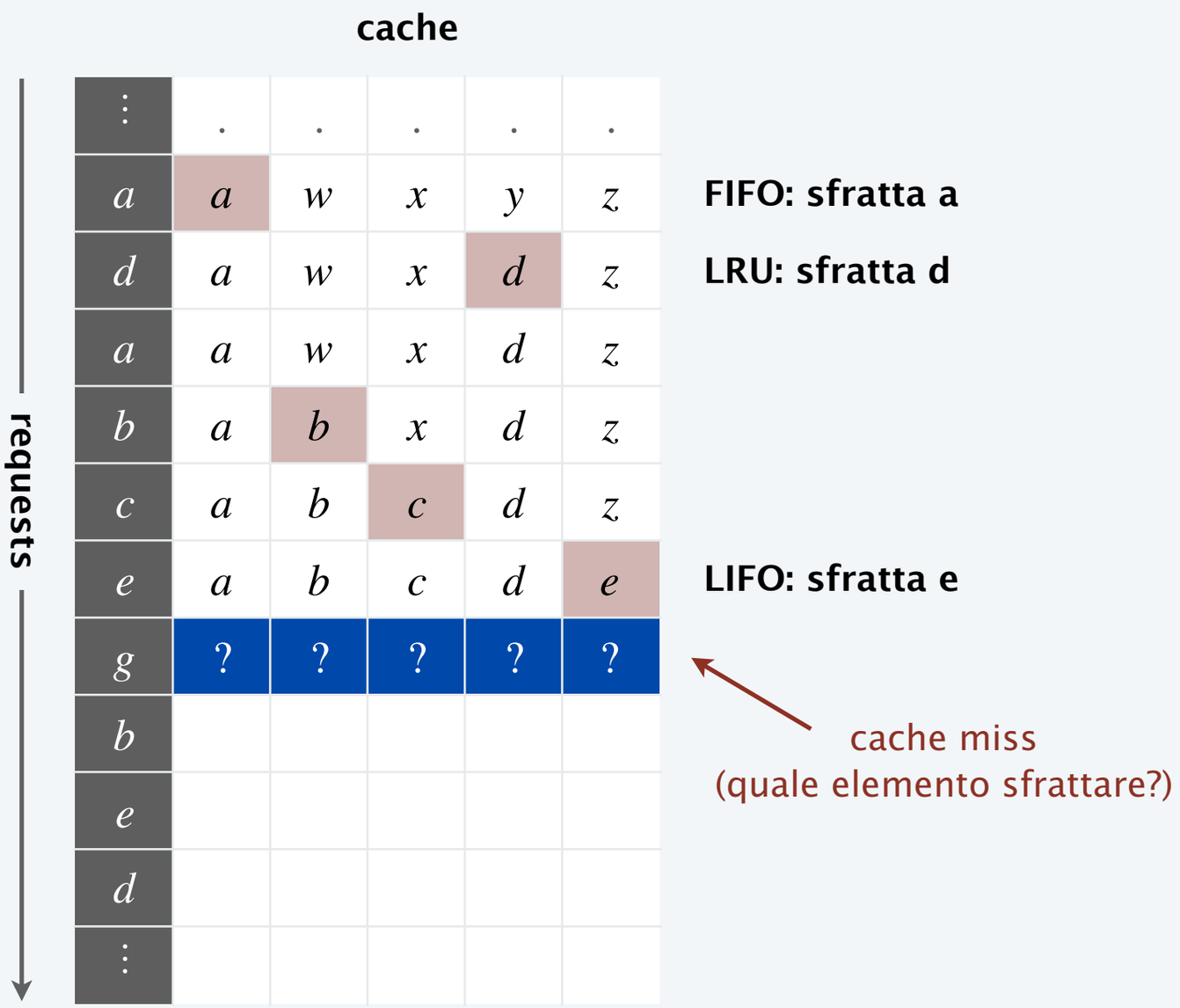


Caching offline ottimo: algoritmi avidi

LIFO/FIFO. Rimpiazza l'elemento aggiunto più (meno) di recente.

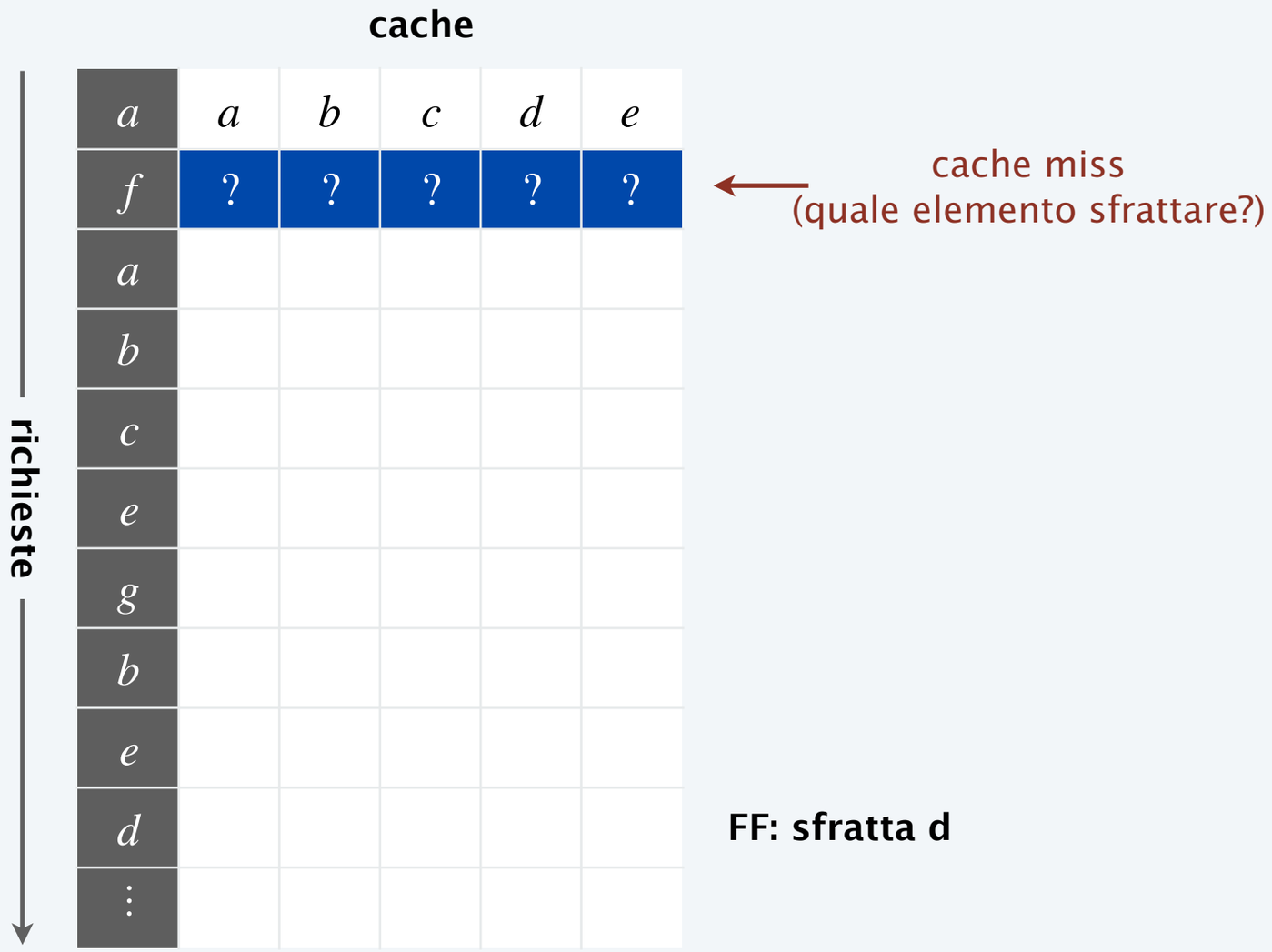
LRU. Rimpiazza l'elemento il cui accesso più recente è il più vecchio possibile.

LFU. Rimpiazza l'elemento richiesto meno di frequente.



Caching offline ottimo: farthest-in-future (algoritmo "chiaroveggente")

Farthest-in-future. Rimpiazza l'elemento nella cache che non sarà richiesto fino ad un momento il più in là possibile nel futuro.



Teorema. [Bélády 1966] FF è un algoritmo di rimpiazzo ottimo.

Dim. L'algoritmo e il teorema sono intuitivi; la dimostrazione è sottile.

Algoritmi avidi I: quiz 6

Quale elemento viene rimpiazzato nello schedule di farthest-in-future?

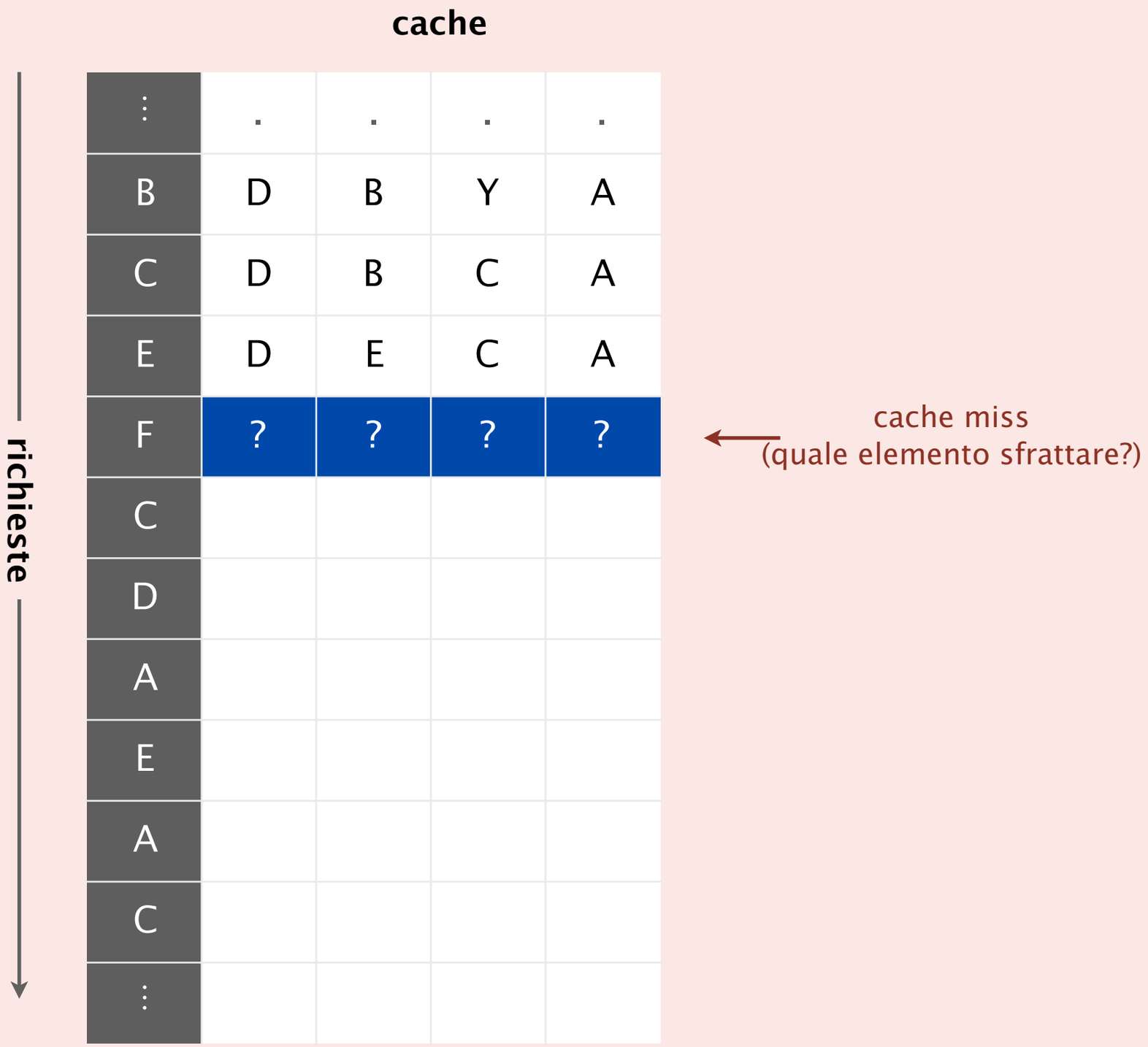
A.

B.

C.

D.

E.



Schedule ridotti

Def. Uno schedule ridotto è uno schedule che porta un elemento d nella cache al passo j solo se c'è una richiesta per d al passo j e d non è già nella cache.

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

d è inserito in cache senza una richiesta

d è inserito in cache pur essendo già presente

uno schedule non ridotto

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

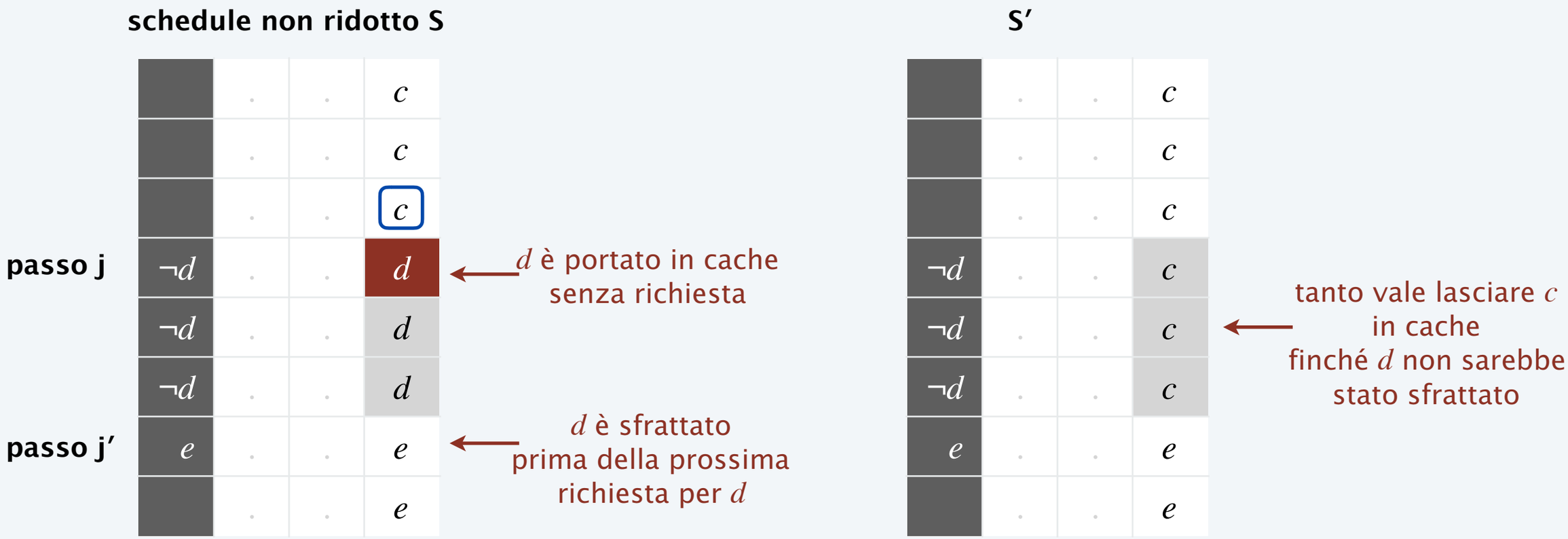
uno schedule ridotto

Schedule ridotti

Prop. Ogni schedule non ridotto S può essere essere trasformato in uno schedule ridotto S' senza aumentare il numero dei rimpiazzi.

Dim. [per induzione sul numero di passi j]

- Supponiamo che S porti d in cache al passo j senza che vi sia richiesta.
- Sia c l'elemento sfrattato da S quando d viene portato in cache.
- Caso 1a: d rimpiazzato prima della successiva richiesta per d.

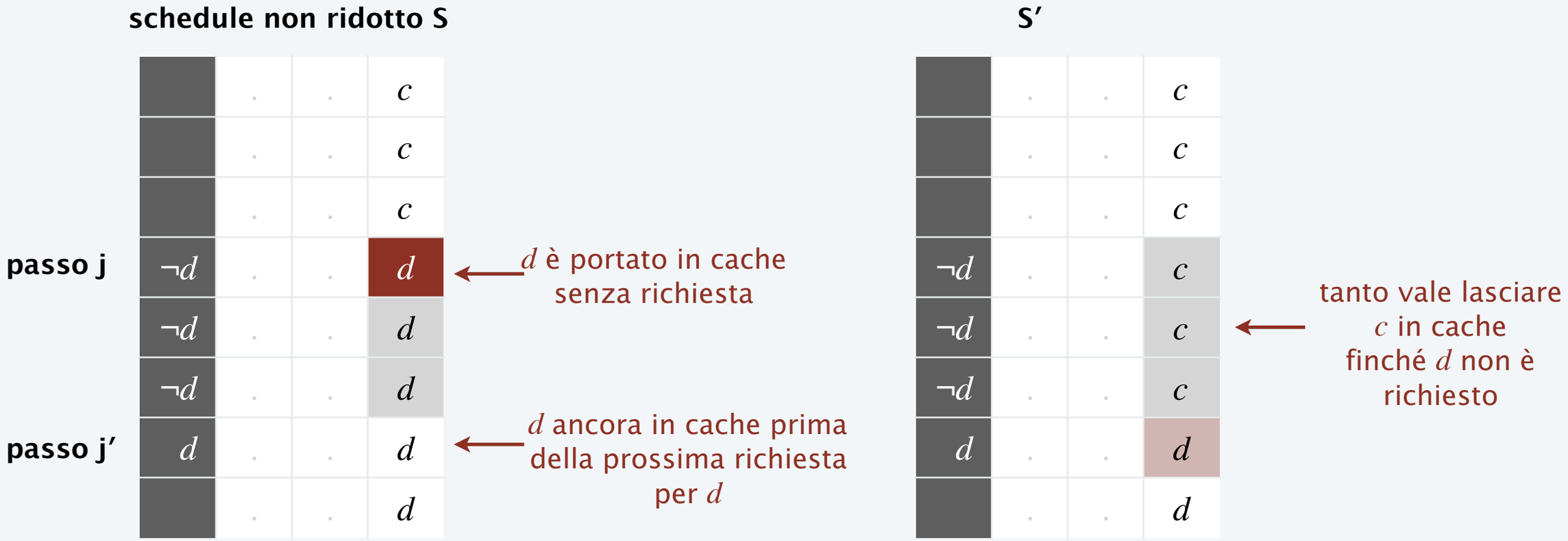


Schedule ridotti

Prop. Ogni schedule non ridotto S può essere essere trasformato in uno schedule ridotto S' senza aumentare il numero dei rimpiazzi.

Dim. [per induzione sul numero di passi j]

- Supponiamo che S porti d in cache al passo j senza che vi sia richiesta.
- Sia c l'elemento sfrattato da S quando d viene portato in cache.
- Caso 1a: d rimpiazzato prima della successiva richiesta per d.
- Caso 1b: la prossima richiesta per d occorre prima che d sia sfrattato.



Schedule ridotti

Prop. Ogni schedule non ridotto S può essere essere trasformato in uno schedule ridotto S' senza aumentare il numero dei rimpiazzi.

Dim. [per induzione sul numero di passi j]

- Caso 1: S porta d nella cache al passo j senza una richiesta. ✔
- Caso 2: S porta d nella cache al passo j sebbene d sia già in cache. ✔

  (può essere dimostrato in modo analogo) ▪

Analisi di Farthest-in-future

Teorema. L'algoritmo di rimpiazzo FF è ottimo.

Dim. Conseguenza immediata del seguente invariante.

Invariante. Esiste uno schedule ottimo ridotto S che ha lo stesso schedule dei rimpiazzi di S_{FF} per tutti i primi j passi.

Dim. [per induzione sul numero di passi j]

Caso base: $j = 0$.

Sia S uno schedule ridotto che soddisfa l'invariante per i primi j passi.

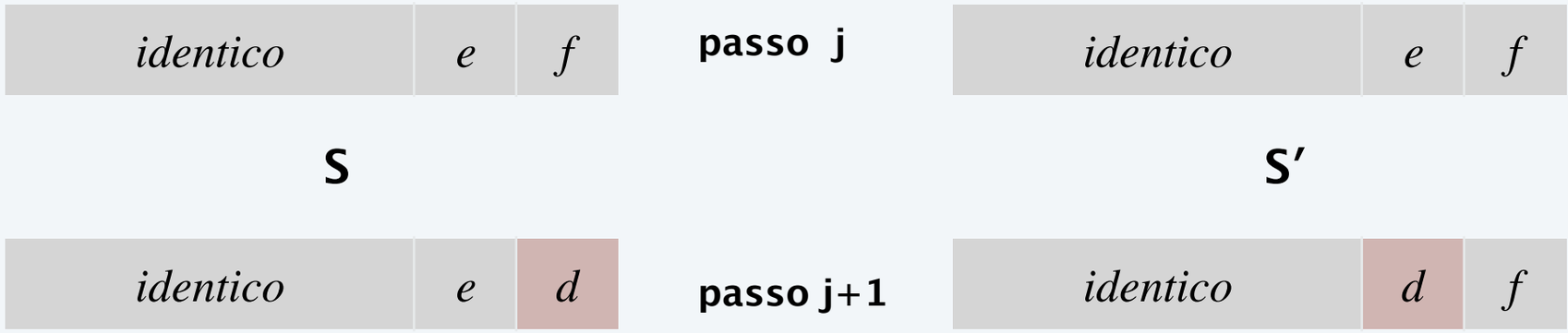
Costruiamo uno schedule S' che soddisfa l'invariante per $j + 1$ passi.

- Sia d l'elemento richiesto al passo $j + 1$.
- Poiché S e S_{FF} sono uguali fino al passo j, il contenuto delle loro cache è identico prima del passo $j + 1$.
- Caso 1: d è già nella cache.
  $S' = S$ soddisfa l'invariante.
- Caso 2: d non è nella cache e S ed S_{FF} sfrattano lo stesso elemento.
  $S' = S$ soddisfa l'invariante.

Analisi di Farthest-in-future

Dim. [continua]

- Caso 3: d non è nella cache; S_{FF} sfratta e; S sfratta $f \neq e$.
  - iniziamo a costruire S' da S sfrattando e invece di f



- ora S' è uguale a S_{FF} per i primi $j+1$ passi; mostriamo che avere f nella cache non è peggio che avere e nella cache

- il resto di S' è uguale ad S fino a che S' non deve effettuare un'azione diversa (o perché S sfratta e; o perché e od f vengono richieste)

Analisi di Farthest-in-future

Sia j' il primo passo dopo $j+1$ in cui S' deve fare un'azione diversa da S; sia g l'elemento richiesto al passo j'.

coinvolge e od f (o entrambe)

- Caso 3a: $g = e$.

  S' concorda con S_{FF} per i primi $j+1$ passi

  Non può capitare in FF poiché f verrà richiesto prima di e.

- Caso 3b: $g = f$.

  L'elemento f non può essere nella cache di S; sia e' l'elemento sfrattato da S.

  - se $e' = e$, S' trova f nella cache; ora S ed S' hanno cache identiche
  - se $e' \neq e$, S' fa sfrattare e' e porta e nella cache;
    ora S ed S' hanno la stessa cache

  S' è definito uguale ad S per le richieste rimanenti.

S' non è più ridotto, ma può essere trasformato in uno schedule ridotto che coincide con FF per i primi $j+1$ passi

Analisi di Farthest-in-future

Sia j' il primo passo dopo $j+1$ in cui S' deve fare un'azione diversa da S;
sia g l'elemento richiesto al passo j'.

coinvolge e od f (o entrambe)

altrimenti S' avrebbe potuto imitare S

- Caso 3c: $g \neq e, f$. S sfratta e.
  - facciamo sì che S' sfratti f.

  - ora S ed S' hanno cache identiche
  - S' è definito uguale ad S per le richieste rimanenti. ▪

Prospettiva sul caching

Algoritmi online vs. offline.

- Offline: l'intera sequenza di richieste è nota a priori.
- Online (realtà): le richieste non sono note a priori.
- Il caching è uno tra i problemi online più studiati in informatica.

LIFO. Rimpiazza l'elemento portato in cache più di recente.

LRU. Rimpiazza l'elemento il cui ultimo accesso è il più vecchio possibile.

FF con una direzione del tempo rovesciata!

Teorema. FF è un algoritmo di rimpiazzo ottimo.

- Fornisce una base per la comprensione e l'analisi di algoritmi online.
- LIFO può essere arbitrariamente più costoso dell'ottimo.
- LRU è k-competitivo: per ogni sequenza di richieste σ, $LRU(\sigma) \leq k\, FF(\sigma) + k$.
- Esiste un algoritmo online (marcatura aleatoria) $O(\log k)$-competitivo.

vedi SEZIONE 13.8