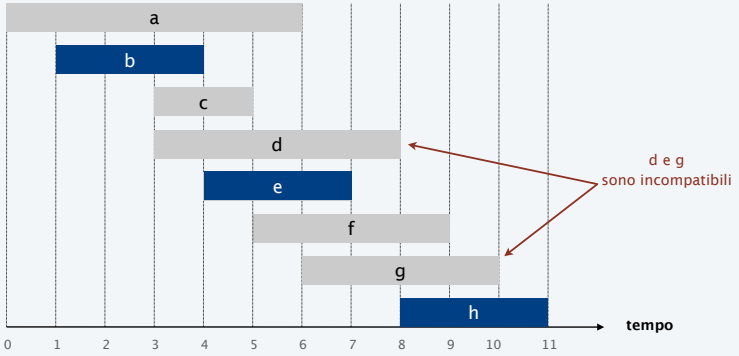
Scheduling di intervalli/attività

- Intervallo j inizia a $s(j)$ e termina a $f(j)$.
- Due intervalli sono compatibili se non si sovrappongono.
- Goal: trovare un insieme di attività compatibili di taglia massima.



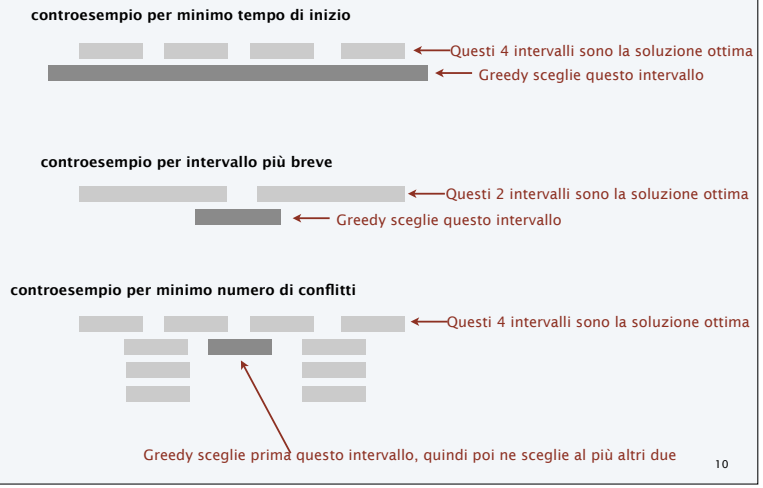
Scheduling di intervalli: algoritmi greedy

Schema Greedy generale. (i) Scegli un intervallo; (ii) elimina tutti gli intervalli incompatibili con quelli già scelti; (iii) ripeti da (i) se ci sono ancora intervalli

- [Minimo tempo di inizio] Considera intervalli in ordine crescente di $s(j)$.
- [Minimo tempo di fine] Considera intervalli in ordine crescente di $f(j)$.
- [Intervallo più breve] Considera intervalli in ordine crescente di $f(j) - s(j)$.
- [Minimo numero di conflitti] Per ogni intervallo j, calcola $c(j)$ = numero di intervalli con cui non è compatibile. Considera intervalli in ordine crescente di $c(j)$.

Scheduling di intervalli: algoritmi greedy

Schema Greedy generale. (i) Scegli un intervallo; (ii) elimina tutti gli intervalli incompatibili con quelli già scelti; (iii) ripeti da (i) se ci sono ancora intervalli



Scheduling di intervalli: algoritmo basato sul minimo tempo di fine

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

ORDINA gli intervalli per tempo di fine: $f_1 \le f_2 \le \ldots \le f_n$

$A \leftarrow \phi$ ← insieme di intervalli selezionati

LAST $\leftarrow 0$ ← tempo di fine dell'ultimo intervallo in A

FOR $j = 1$ TO n

  IF $s(j) \ge$ LAST ← se j è compatibile con A

    $A \leftarrow A \cup \{ j \}$

    LAST $\leftarrow f(j)$ ← ora j è l'intervallo che finisce per ultimo in A

RETURN A

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

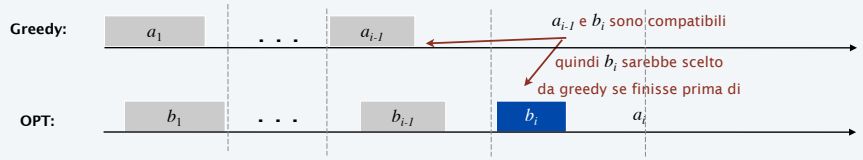
- LAST ricorda il tempo dell'ultimo intervallo inserito in A.
- quindi j è compatibile con A se e solo se $s(j) \ge$ LAST.
- Ordinare n elementi (per tempo di fine) richiede tempo $O(n \log n)$.

Scheduling di intervalli: algoritmo basato sul minimo tempo di fine

Teorema. L'algoritmo trova una soluzione ottima (massimo # intervalli).

Dim.

- Siano $a_1, a_2, \ldots a_k$ gli intervalli selezionati da greedy.
- Siano $b_1, b_2, \ldots b_m$ gli intervalli in una soluzione ottima.
- Per ogni $i = 1,\ldots, k$ abbiamo $f(a_i) \le f(b_i)$
  - Per induzione su i:
    Base: $i = 1,\ f(a_1) \le f(b_1)$ perchè greedy sceglie quello che finisce prima
    Ipotesi induttiva: supponiamo vero per $i - 1$, cioè $f(a_{i-1}) \le f(b_{i-1})$
    Passo induttivo: se (per assurdo) valesse $f(b_i) \le f(a_i)$ greedy avrebbe *scelto* b_i perchè vale $f(a_{i-1}) \le f(b_{i-1}) \le s(b_i)$ quindi b_i sarebbe compatibile con le scelte di greedy.

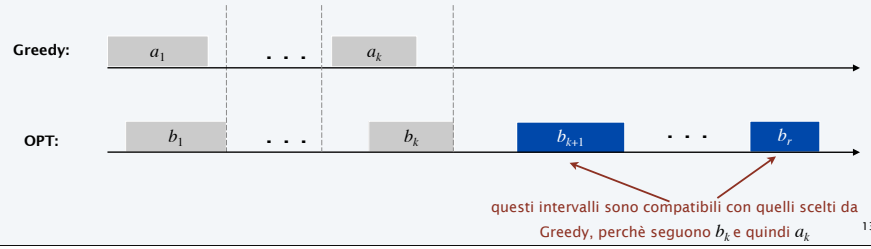


Scheduling di intervalli: algoritmo basato sul minimo tempo di fine

Teorema. L'algoritmo trova una soluzione ottima (massimo # intervalli).

Dim - parte 2. [per assurdo]

- Siano $a_1, \ldots a_k$ gli intervalli selezionati da greedy e $b_1, \ldots b_r$ quelli di una soluzione ottima, ordinati per tempo di fine crescente
- Assumiamo che greedy non sia ottimale, cioè $k < r$
- Poiché $f(a_k) \le f(b_k) \le s(b_{k+1})$ segue che $b_{k+1}, b_{k+2}, \ldots b_r$ sono compatibili con le scelte di greedy e quindi greedy non si sarebbe fermato

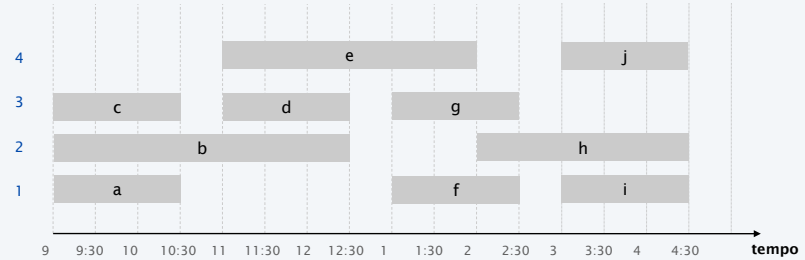


Partizionamento di Intervalli

Suddivisione degli intervalli.

- Lezione j comincia al tempo $s(j)$ e finisce a $f(j)$.
- Goal: Trova il minimo numero di aule in cui tenere le lezioni in modo che in ogni aula le lezioni siano compatibili.

Es. Questa soluzione usa 4 classi per organizzare 10 lezioni.

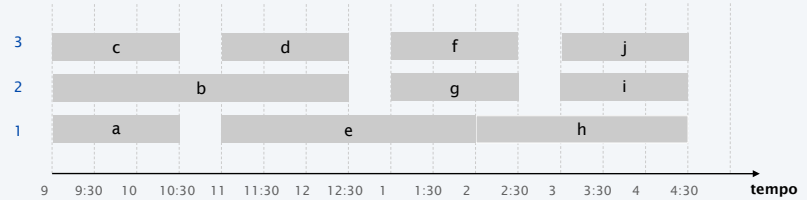


Partizionamento di Intervalli

Suddivisione degli intervalli.

- Lezione j comincia al tempo $s(j)$ e finisce a $f(j)$.
- Goal: Trova il minimo numero di aule in cui tenere le lezioni in modo che in ogni aula le lezioni siano compatibili.

Es. Questa soluzione usa 3 classi per organizzare le stesse 10 lezioni



Partizionamento di Intervalli: algoritmo greedy

Schema Greedy generale. (i) Scegli una lezione; (ii) Assegnala ad una classe disponibile se è compatibile con le lezioni da tenere lì; (iii) se non esiste una tale aula, allocane una nuova.

- [Minimo tempo di inizio] Considera intervalli in ordine crescente di $s(j)$.

Partizionamento di Intervalli: algoritmo per minimo tempo di inizio

EARLIEST-START-TIME-FIRST (n, s_1, s_2, …, s_n, f_1, f_2, …, f_n)

ORDINA lezioni per tempo di inizio: $s_1 \le s_2 \le \ldots \le s_n$.

$d \leftarrow 0$ ⟵ numero di classi allocate

FOR $j = 1$ TO n

  IF lezione j è compatibile con quelle in una qualche classe k

    assegna la lezione j in tale classe k.

  ELSE

    Alloca una nuova classe $d + 1$.

    assegna la lezione j alla nuova classe $d + 1$.

    $d \leftarrow d + 1$

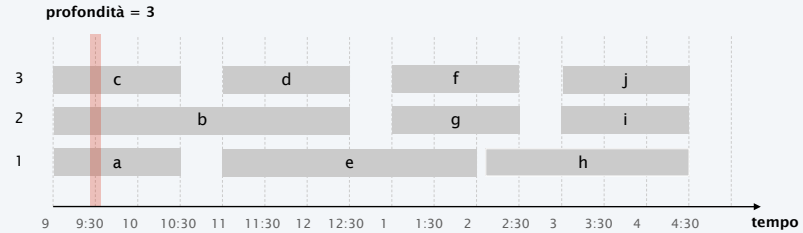
RETURN gli assegnamenti scelti.

Partizionamento di Intervalli: lower bound sull'ottimo

Def. La profondità di un insieme di lezioni è il massimo numero di lezioni che "contengono" uno stesso istante.

Osservazione chiave. Numero di classi necessarie ≥ profondità.

Q. Riusciamo sempre ad usare un numero di classi pari alla profondità?
A. sì! Inoltre, l'algoritmo EARLIEST-START-TIME-FIRST trova una tale soluzione.



Partizionamento di Intervalli: analisi dell'algoritmo

Osservazione. L'algoritmo greedy non schedula mai due lezioni incompatibili nella stessa classe.

Teorema. L'algoritmo greedy EARLIEST-START-TIME-FIRST che sceglie sempre la lezione con il tempo di inizio minimo è ottimale.

Dim.

- sia d = il numero di classi che greedy alloca.
- La classe d è allocata la prima volta perché dobbiamo schedulare la lezione j che è incompatibile con tutte le altre $d - 1$ classi.
- Quindi in ognuna di queste d-1 classi c'é una lezione che finisce dopo $s(j)$
- Poiché queste d-1 lezioni sono già state considerate da greedy e greedy considera in ordine di tempo di inizio, vuol dire che tutte queste lezioni non cominciano più tardi di $s(j)$.
- Quindi, includendo anche j, abbiamo d lezioni che devono essere attive subito dopo s_j. Quindi la profondità è almeno d
- Per l' Osservazione chiave (slide precedente) ⇒ tutti gli algoritmi usano ≥ d classi. ▪

Partizionamento di Intervalli: algoritmo minimo tempo di inizio

Proposizione. L'algoritmo può essere implementato in tempo $O(n \log n)$.

- questo non lo dimostriamo per il momento

Esercizio. Dimostrare che l'algoritmo può essere implementato con tempo di esecuzione $O(n\, d)$, dove d è la profondità dell'istanza, che è anche uguale al numero finale di classi usate.