Progettazione di Algoritmi **Anno Accademico 2023–2024**

Note per la Lezione 21

Ugo Vaccaro

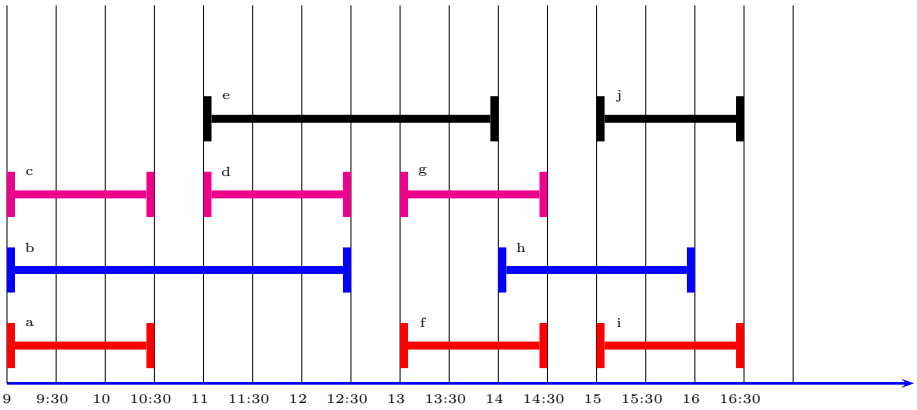
In questa lezione vedremo esercizi relativi all'applicazione della tecnica Programmazione Dinamica e Greedy.

Partizionamento di attività

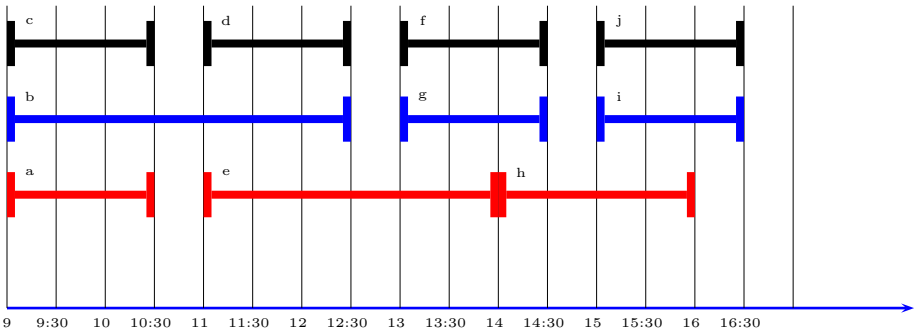
Input: n corsi, ciascuno con tempo di inizio s_i e tempo di fine f_i

Output: minimo numero di aule cui assegnare i corsi, in modo che nessuna coppia di corsi si svolga allo stesso tempo nella stessa aula

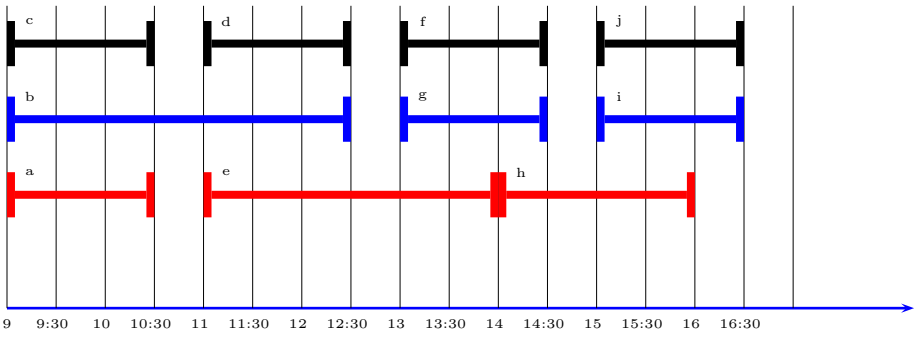
Esempio: questa assegnazione utilizza 4 aule per 10 corsi



Quest'altra assegnazione utilizza solo 3 aule per 10 corsi.



Possiamo fare meglio (cioé usare *meno* di 3 aule)? **No**, Infatti nell'intervallo dalle 9:30 alle 10:00 vi sono 3 corsi in contemporanea, che necessitano evidentemente di 3 aule distinte



L'ultima osservazione può essere generalizzata nella seguente:

Osservazione chiave: se in un dato istante ci sono d corsi in contemporanea, allora ogni partizionamento di attività richiede almeno d aule

Algoritmo Greedy per partizionare i corsi tra il minor numero di aule

Idea: Esamina i corsi nell'ordine in cui essi iniziano, e assegnali ad aule in modo che corsi di una stessa aula non si sovrappongano. **Solo** se questo non é possibile, utilizza una nuova aula.

```
Partizionamento_greedy (s_1 ... s_n, f_1, ..., f_n)
1.  Ordina i corsi in modo che s_1 ≤ s_2 ≤ ... ≤ s_n
2.  d = 0   %(d é il numero di aule utilizzate finora)
3.  FOR(j=1, j<n+1, j=j+1) {
4.      IF (corso j non si sovrappone ai corsi di qualche
              aula k ∈ {1, ..., d},) {
              assegna il corso j all'aula k
      } ELSE {
5.      apri l'aula d + 1
6.      assegna il corso j all'aula d + 1
7.      d = d + 1
    }  }
```

Analisi: sia D il numero di aule usate dall'algoritmo

L'algoritmo **non** assegna mai due corsi che si svolgono in uno stesso momento alla stessa aula. L'aula D é stata usata in quanto esisteva un corso j che **non** poteva essere inserito in **nessuna** delle aule $1, \ldots, D-1$

$\Longrightarrow$ poiché i corsi sono ordinati per tempo di inizio, questa impossibilitá é causata da $D-1$ corsi che iniziano a tempi $\leq s_j$ $\Longrightarrow$ al tempo $s_j + \Delta$ ci sono D corsi che si sovrappongono $\Longrightarrow$ dalla Osservazione Chiave, **ogni** assegnazione di corsi ad aule richiede D aule, quindi l'algoritmo utilizza il minor numero possibile di aule.

Complessitá:

1. richiede $O(n \log n)$. Il `FOR 3.` viene eseguito n volte. Per ogni aula, possiamo ricordarci il tempo di fine dell'ultimo corso assegnatoli (cosicché per verificare che il corso j non si sovrappone ai corsi giá inseriti in un' aula basta solo controllare che s_j sia maggiore di tale tempo di fine) Il numero di aule é $O(n)$, quindi l'istruzione `4.` richiede tempo $O(n)$.

Il resto richiede tempo $O(1)$, quindi in totale l'algoritmo greedy richiede tempo $O(n^2)$ (vedremo in seguito che un'implementazione più furba dell'algoritmo richiederá tempo $O(n \log n)$)

$\diamond$

Input: Vettore di numeri $A = A[1]...A[n]$
Output: piú lunga sottosequenza (non necessariamente contigua) di elementi crescenti di A

Detto in altri termini, cerchiamo il piú grande valore di k per cui esistono interi $1 \leq i_1 < \ldots < i_k \leq n$ tali che $A[i_1] < \ldots < A[i_k]$.

Ad esempio se abbiamo la sequenza

$$0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15,$$

una tra le piú lunghe sottosequenze sarebbe 0, 2, 6, 9, 11, 15.

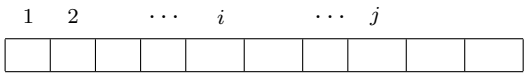
Al fine di derivare una equazione di ricorrenza per la soluzione al problema, effettuiamo le seguenti considerazioni. Sia

$$L(j) = \text{lunghezza della piú lunga sottosequenza crescente che termina nella posizione} j.$$

Noi siamo ovviamente interessati alla lunghezza della più lunga sottosequenza crescente che finisce in una qualunque posizione, cioé al valore

$$\max\{L(1), L(2), \ldots L(n)\}.$$

Per il calcolo di $L(j)$ chiedamoci: Quale puó essere il penultimo elemento (sia esso $A[i]$) nella sottosequenza di lunghezza massima $L(j)$? (l'ultimo é ovviamente $A[j]$)



Sicuramente varrá che $A[i] < A[j]$, dato che la sottosequenza deve essere composta da elementi crescenti Inoltre, nella posizione i terminerá la più lunga sottosequenza crescente con ultimo elemento $A[i]$. Pertanto

$$L(j) = \max\{L(i) : i < j \text{ e } A[i] < A[j]\}+1.$$

Valutiamo la complessitá di un tale algoritmo. Ci basterá un tempo $O(n)$ per il calcolo di ciascun $L(j)$, per $j = 1, \ldots, n$, ed un tempo $O(n)$ per il calcolo di $\max\{L(1), L(2), \ldots L(n)\}$. In totale, ci basta tempo $O(n^2)$.

L'algoritmo sará:

```
MaxSottosequenzaCrescente(A)
1. FOR (j=1, j<n+1, j=j+1) {
2.   L(j)=1
   }
3. FOR (j=2, j<n+1, j=j+1) {
4.   FOR (i=1, i<j, i=i+1) {
5.     IF ((A[i]<A[j])&&(L(j)<L(i)+1))
6.       L(j)=L(i)+1
        }
     }
7. max=L(1)
8. FOR (j=2, j<n+1, j=j+1) {
9.   IF (L(j)>max) {
10.   max=L(j)
   }
11. RETURN max
```

Un utile esercizio consiste nel trovare la sequenza crescente piú lunga, e non solo la sua lunghezza.

$\diamond$

Vediamo un altro esempio, per la cui soluzione useremo la tecnica Greedy.

Disponiamo di un tubo metallico di lunghezza L. Da questo tubo vogliamo ottenere al piú n segmenti di lunghezza minore, aventi rispettivamente lunghezza $S[1], S[2], \ldots, S[n]$.

Il tubo viene segato sempre a partire da una delle estremitá, quindi ogni taglio riduce la sua lunghezza della misura asportata. Vogliamo determinare il numero massimo di segmenti che é possibile ottenere.

Questo problema si puó risolvere con un algoritmo greedy. Ordiniamo le sezioni in senso non decrescente rispetto alla lunghezza, in modo che il segmento 1 abbia lunghezza minima e il segmento n lunghezza massima. Procediamo quindi a segare prima il segmento più corto, poi quello successivo e cosi via finché possibile (cioé fino a quando la lunghezza residua ci consente di ottenere almeno un'altro segmento).

Lo pseudocodice puó essere descritto in questo modo:

```
MassimoNumero(L, S[1...n])
1.  Ordina S in senso crescente
2.  i=1
3.  WHILE(i<n+1&&L ≥ S[i]){
4.  L = L − S[i]
5.  i=i+1
   } return i-1
```

L'operazione di ordinamento puó essere fatta in tempo $\Theta(n \log n)$. Il successivo ciclo while ha costo $\Theta(n)$ nel caso peggiore. Il costo complessivo dell'algoritmo risulta quindi $\Theta(n \log n)$.

Proviamo che l'algoritmo ritorna una soluzione ottima, ovvero per cui il numero di segmenti ritornati é il massimo possibile.

Osserviamo che se $\min\{S[1], \ldots, S[n]\} > L$, l'algoritmo ritorna il valore 0, e questo é ottimo in quanto non esistono pezzi che si possono ottenere, con le lunghezze date dal vettore $S[1 \ldots n]$.

Supponiamo quindi che $\min\{S[1], \ldots, S[n]\} \leq L$. Sia k il massimo numero di segmenti che é possibile ottenere, e siano $S[i_1], \ldots, S[i_k]$ le lunghezze di tali segmenti di una soluzione ottima. Per tali lunghezze vale ovviamente che $\sum_{j=1}^{k} S[i_j] \leq L$. Proviamo che esiste una soluzione della stessa cardinalità k della forma $(\{S[i_1], \ldots, S[i_k]\} \setminus \{S[i_1]\}) \cup \{S[1]\}$, dove $S[1]$ é la prima scelta effettuata dall'algoritmo Greedy. Infatti, poiché $S[1] = \min\{S[1], \ldots, S[n]\}$, segue che $S[1] \leq S[i_1]$, da cui $S[1] + \sum_{j=2}^{k} S[i_j] \leq \sum_{j=1}^{k} S[i_j] \leq L$ (quindi $(\{S[i_1], \ldots, S[i_k]\} \setminus \{S[i_1]\}) \cup \{S[1]\}$ é una soluzione corretta) con un numero di segmenti pari a $(k-1)+1 = k$, cioé il massimo possibile.

A questo punto, possiamo procedere per induzione sulla lunghezza $L - S[1]$ e provare che l'algoritmo éottimo.

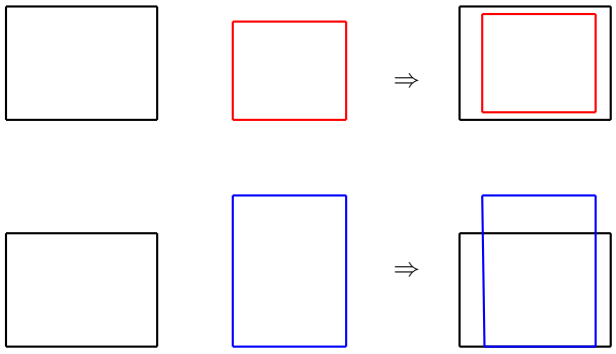
$\diamond$

Vediamo un altro esempio, per la cui risoluzione useremo la tecnica Greedy.

Siano $R = \{1, 2, \ldots, n\}$ n rettangoli. Il rettangolo i ha base $b(i)$ e altezza $h(i)$. Diremo che il rettangolo i contiene il rettangolo j se e solo se sia la base che l'altezza del rettangolo i sono maggiori della base ed altezza del rettangolo j, ovvero se e solo se

$$b(i) \geq b(j) \quad \text{e} \quad h(i) \geq h(j).$$

Ad esempio, il rettangolo rosso é contenuto nel nero, mentre il blu non lo é:



Diremo che il sottoinsieme di rettangoli $S \subseteq R = \{1, 2, \ldots, n\}$ contiene tutto l'insieme R se $\forall$ rettangolo $j \in R$ esiste un rettangolo $i \in S$ che contiene j.

Problema:
Input: Insieme di rettangoli $R = \{1, 2, \ldots, n\}$
Output: $S \subseteq R$ di cardinalità minima che contiene tutto R.

Supponiamo che i rettangoli in R siano ordinati per base decrescente, ovvero se $i < j$ allora $b(i) \geq b(j)$ (se $b(i) = b(j)$ allora $h(i) \geq h(j)$). Se non lo sono, li ordiniamo noi in tempo $O(n \log n)$.

Idea per un algoritmo greedy:
Passo 1: 1 é il rettangolo con base maggiore di tutti, quindi conviene metterlo in S,
Passo $i \geq 2$: sia j l'ultimo rettangolo inserito in S. Se il rettangolo in esame i é contenuto in j ignoriamo i, altrimenti mettiamo i in S.

L'algoritmo e sua analisi

```
1. S={1}
2. j=1
3. FOR (i=2, i<n+1, i=i+1) {
4.   IF (h(i)> h(j)) {
5.     S= S∪{i}
6.     j= i
        }
     }
7. RETURN S
```

Al fine di provare che l'algoritmo sopra descritto restituisce una soluzione ottima, ovvero di cardinalitá minima, osserviamo innazitutto che esiste una soluzione di cardinalitá minima (ottima) che contiene il rettangolo 1.

Sia pertanto S una soluzione ottima, se $1 \in S$ allora non vi é nulla da provare. Se $1 \notin S$, allora S deve possedere un rettangolo $i \geq 1$ che contiene 1, ovvero per cui $b(i) \geq b(1)$ e $h(i) \geq h(1)$. Ricordando che i rettangoli erano ordinati per basi decrescenti, abbiamo anche che $b(i) \leq b(1)$, da cui discende necessariamente che $b(i) = b(1)$. Ricordiamo anche che i rettangoli di base uguale erano ordinati per altezza decrescente, quindi $h(i) \leq h(1)$, e di nuovo ne segue che $h(i) = h(1)$. Pertanto i rettangoli i e 1 hanno le stesse dimensioni, da cui si ha che anche $S' = S - \{i\} \cup \{1\}$ copre tutto R, con $1 \in S'$ e S' ottimo (in quanto $|S'| = |S|$).

Sia k il rettangolo scelto al secondo passo dall'algoritmo Greedy. Proviamo che esiste una soluzione di cardinalità minima (ottima) che contiene 1 e k.
Se k é la seconda scelta effettuata, k é il più piccolo intero per cui $h(k) > h(1)$. Sia S una soluzione ottima che contiene 1. Se $k \in S$, allora non c'é nulla da provare. Se $k \notin S$, allora S deve possedere un rettangolo j che contiene k, ovvero per cui $b(j) \geq b(k)$ e $h(j) \geq h(k) > h(1)$. Essendo k il più piccolo intero per cui $h(k) > h(1)$, abbiamo che $j > k$. A causa dell'ordinamento decrescente delle basi, abbiamo $b(j) \leq b(k)$ e quindi $b(j) = b(k)$. A causa dell'ordinamento decrescente delle altezze di rettangoli con basi uguali, si ha che $h(j) \leq h(k)$, da cui, di nuovo otteniamo che $h(j) = h(k)$. Pertanto, i rettangoli k e j hanno le stesse dimensioni. Ne segue che anche $S' = S - \{j\} \cup \{k\}$ copre tutto R, con $1, k \in S'$ e S' ottimo (in quanto $|S'| = |S|$).

Iterando... possiamo concludere che esiste una soluzione ottima che contiene tutte le scelte effettuate dall'algoritmo Greedy, ergo, l'algoritmo Greedy produce una soluzione ottima al problema.

Complessitá: $O(n)$ se i rettangoli sono già ordinati come richiesto, alrimenti $O(n \log n)$.

$\diamond$

Input: n brani musicali $\{1, 2, \ldots, n\}$ di durata $d_1, d_2, \ldots, d_n$, ed un CD di dimensione D.
Output: il maggior numero di brani che possiamo memorizzare sul CD.

Supponiamo per semplicità che le durate siano distinte e ordinate in senso crescente $d_1 < d_2 < \ldots < d_n$. Un possibile semplice algoritmo consiste nel memorizzare i brani nell'ordine dal più piccolo al più grande, fin quando il CD non ne può contenere più.

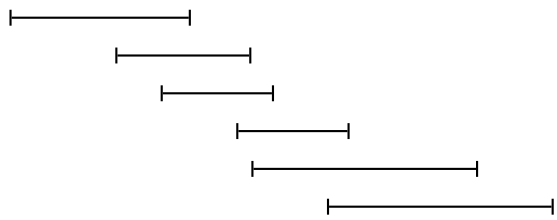
Siano $d_1, \ldots, d_k$ le durate dei brani memorizzati dall'algoritmo. É ovvio che esiste una soluzione ottima che contiene il brano di durata d_1. Infatti, se S é un insieme di brani ottimo per cui $1 \notin S$, allora detto $j > i$ il primo brano per cui $j \in S$, allora $d_j > d_1$, e posso sostituire j con 1, rispettando il vincolo sulla somma delle durate dei brani. Otteniamo quindi un'altro insieme ottimo di $|S|$ brani che contiene questa volta il brano musicale 1. Il resto per esercizio....

$\diamond$

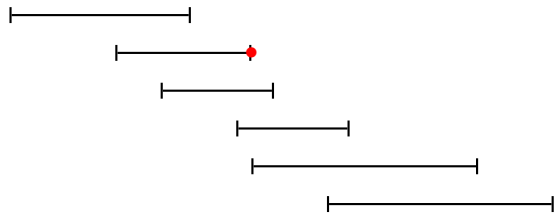
Supponiamo di avere n oggetti $\{1, \ldots, n\}$. Ognuno di essi deve essere immagazzinato e tenuto nell'opportuno range di temperatura. In altri termini, l'oggetto i deve essere conservato ad una temperatura che deve essere compresa nell'intervallo $[s_i, f_i]$, $i = 1, \ldots n$. Determinare il minimo numero di celle frigorifere in cui possiamo immagazzinare i nostri oggetti.

Un pó di riflessione porta all'opportuna formalizzazione del problema, nel modo seguente:
Input: n segmenti $[s_1, f_1], \ldots, [s_n, f_n]$;
Output: un insieme $S = \{x_1, \ldots, x_m\}$ di cardinalità minima tale che per ogni segmento $[s_i, f_i]$ esiste un elemento $x \in S$ per cui $s_i \leq x \leq f_i$.

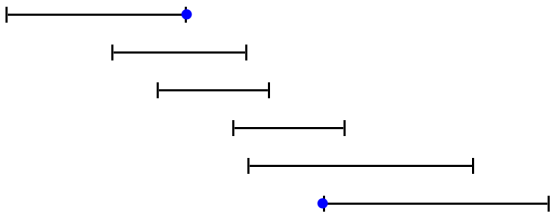
Diciamo che x copre un intervallo generico $[s, t]$ se $s \leq x \leq f$. Un primo algoritmo Greedy che potremmo progettare per il problema potrebbe essere quello che sceglie i punti x da inserire in S in modo tale che, ad ogni scelta, x sia il punto che copre il maggior numero di intervalli non ancora coperti. Purtoppo, ciò non porterebbe ad una soluzione di cardinalità minima. Vediamo il seguente esempio



in questo esempio, se scegliessimo il punto che copre il maggior numero di intervalli, esso necessariamente dovrebbe essere il punto **rosso**, che copre 4 intervalli



Avremmo poi bisogno di altri due punti per coprire il primo ed ultimo intervallo, per un totale di 3 punti. Che questa non sia una soluzione ottima lo si evince dal fatto che i due punti in **blu** coprono tutti gli intervalli



Un algoritmo Greedy più furbo potrebbe essere quindi il seguente.

1. Ordina gli intervalli in base ai valori f_i, dal più piccolo al più grande
2. Inserisci nella soluzione S il punto f_1.
3. Elimina tutti gli intervalli coperti da f_1.
4. Itera sugli intervalli rimanenti.
5. Restituisci S.

L'algoritmo si puó implementare in modo che abbia complessitá $O(n \log n)$.

Per la prova che l'algoritmo sopra descritto produce una soluzione di cardinalità minima, effettuiamo le seguenti considerazioni. Proviamo innanzitutto che esiste una soluzione di cardinalità minima che contiene il punto f_1. Sia S una generica soluzione di cardinalitá minima. Se essa contine f_1, non c'é null'altro da provare. Se $f_1 \notin S$, allora poiché ci deve essere un punto x in S che copre l'intervallo $[s_1, f_1]$, deve necessariamente valere che $x < f_1$. L'insieme $S' = (S \setminus \{x\}) \cup \{f_1\}$ ha la stessa cardinalitá di S e chiaramente copre gli stessi intervalli (cioé, tutti) di S. Il resto per esercizio...