Progettazione di Algoritmi **Anno Accademico 2021–2022**

Note per la Lezione 14

Ugo Vaccaro

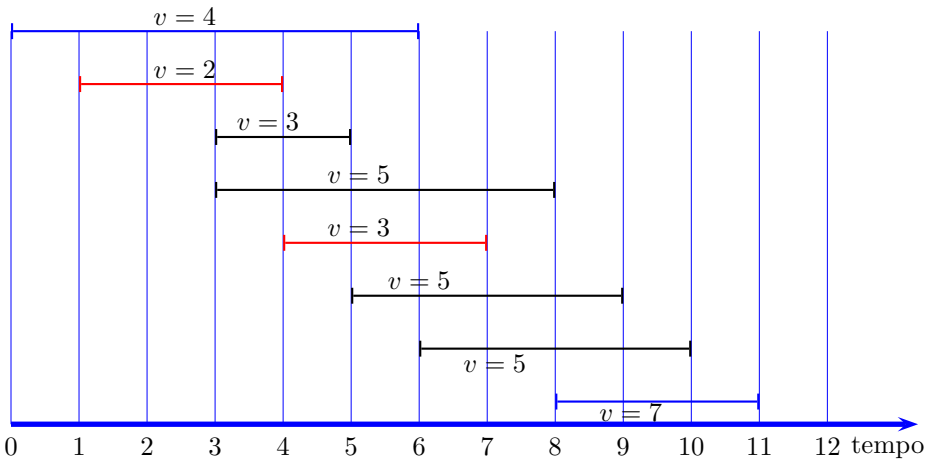
Applichiamo la tecnica di Programmazione Dinamica al seguente problema.

Input del problema: Supponiamo di avere un insieme $A = \{A_1, A_2, \ldots, A_n\}$ di *attività* , dove ciascuna attività A_i ha un tempo di *inizio* s_i, un tempo di *fine* f_i, con $s_i < f_i$ (in altre parole, l'attività A_i deve essere svolta nell'intervallo temporale $[s_i, f_i]$), ed un certo valore $v(A_i) = v_i$. Le attività in A devono essere eseguite da un *server*, sotto la condizione che A_i ed A_j possono essere entrambe eseguite se e solo se $[s_i, f_i] \cap [s_j, f_j] = \emptyset$ (in tal caso, diremo che l'attivitá A_i ed A_j sono *compatibili*). In altri termini, possono essere eseguite dal server solo attività il cui svolgimento temporale non si sovrappone (si pensi, ad esempio, alle attività come dei job che un sistema operativo deve far eseguire da una CPU che può eseguire una sola attività alla volta ed una volta iniziata un'attività questa non può essere interrotta ma bensì eseguita fino alla sua terminazione).

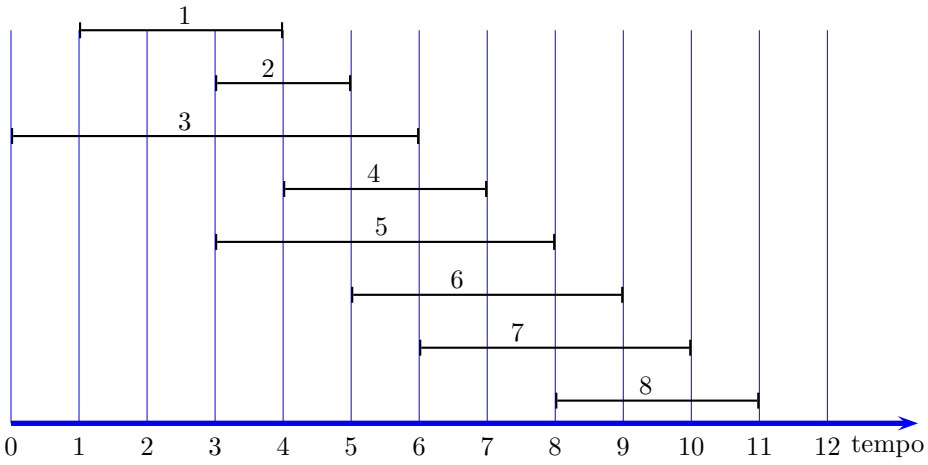
Output del problema: Calcolare un sottoinsieme di $S \subseteq A$ di attività a due a due compatibili, di valore totale $\sum_{A \in S} v(A)$ massimo. Ovvero, vogliamo calcolare

$$\max_{\substack{S \subseteq A : S \text{ è composto da attività} \\ \text{mutualmente compatibili}}} \sum_{A \in S} v(A).$$

Per semplicità, indichiamo l'insieme delle attività $\{A_1, A_2, \ldots, A_n\}$ semplicemente con l'insieme $\{1, 2, \ldots, n\}$. Vediamo un esempio.



Si vede che la prima e l'ultima attivita (di colore blu) sono compatibili, di valore totale 11. Però, anche la seconda, quinta (di colore rosso) e l'ultima attività sono compatibili, di valore totale 12. Al fine di progettare un algoritmo per il problema dello Scheduling di Attività, rinomiano innanzitutto le attività in ordine di terminazione (ovvero ordiniamole in base ai numeri f_i) per cui varrà: $f_1 \leq f_2 \leq \ldots \leq f_n$. Inoltre, per ogni attività j, sia $p(j) =$ il più grande indice $i < j$ tale che attività i è compatibile con l'attività j ($p(j) = 0$ se tale indice non esiste) Nell'esempio precedente, avremmo che dopo aver ordinato la situazione è quella riportata nella figura di sotto. Inoltre, avremmo che $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.



Per semplicità, denotiamo le attività con l'insieme $A = \{1, \ldots, n\}$, ordinate per tempo di fine. Sia $\mathcal{O}$ una soluzione di valore ottimo al problema in questione, ovvero sia $\mathcal{O}$ un sottoinsieme delle attività $A = \{1, \ldots, n\}$ composta da attività a due a due compatibili, di valore totale $\sum_{A \in \mathcal{O}} v(A)$ *massimo* possibile. Sicuramente, o vale che n (l'ultima attività) $\in \mathcal{O}$, oppure vale che $n \notin \mathcal{O}$

- Se $n \in \mathcal{O}$ allora *tutte* le attività $p(n)+1, p(n)+2, \ldots, n-1$ intersecano n, quindi esse *non* possono essere in $\mathcal{O}$. Inoltre, se $n \in \mathcal{O}$ allora $\mathcal{O} - \{n\}$ è una soluzione ottima per le attività $\{1, 2, \ldots, p(n)\}$ (che non intersecano l'attività n). Perchè? Perchè se $\mathcal{O} - \{n\}$ **non** fosse ottima relativamente all'insieme delle attività $\{1, 2, \ldots, p(n)\}$, allora si potrebbe trovare una soluzione in $\{1, 2, \ldots, p(n)\}$ *migliore* di $\mathcal{O} - \{n\}$. Tale soluzione, unita all'attività n sarebbe una soluzione relativamente all'insieme delle attività $\{1, 2, \ldots, n\}$ globalmente migliore di $\mathcal{O}$ stessa! Ciò è contro l'ipotesi di partenza che $\mathcal{O}$ è un sottoinsieme di A composta da attività a due a due compatibili, di valore totale $\sum_{A \in \mathcal{O}} v(A)$ *massimo* possibile.
- Se invece l'ultima attività non è presente nella soluzione ottima $\mathcal{O}$, allora $\mathcal{O}$ è chiaramente anche una soluzione ottima per l'insieme delle attività $\{1, 2, \ldots, n-1\}$

Applichiamo ora il primo passo per la risoluzione del problema in questione, utilizzando la tecnica Programmazione Dinamica, ovvero formuliamo la soluzione del prooblema in termini ricorsivi, cioè scriviamo una formula per la soluzione all'intero problema che sia una combinazione di soluzioni a sottoproblemi di taglia minore.

$\forall 1 \leq j \leq n$, sia $\mathcal{O}_j$ una soluzione ottima per il sottoproblema costituito dalle attività $\{1, \ldots, j\}$, e sia $\texttt{OPT}(j)$ il valore di $\mathcal{O}_j$ (noi cerchiamo $\texttt{OPT}(n)$).

Da quanto detto prima, o vale che $j \in \mathcal{O}_j$ (ed in tal caso $\mathcal{O}_j$ non può contenere le attività $p(j)+1, \ldots, j-1$). Inoltre $\mathcal{O}_j - \{j\}$ è una soluzione ottima (ovvero di valore $\texttt{OPT}(j)$) per le attivitá $\{1, 2, \ldots, p(j)\}$ In simboli

$$\texttt{OPT}(j) = v_j + \texttt{OPT}(p(j)).$$

Oppure vale che $j \notin \mathcal{O}_j$, ed in tal caso vale che

$$\texttt{OPT}(j) = \texttt{OPT}(j-1).$$

Tutto ciò vuol dire che

$$\texttt{OPT}(j) = \max\{v_j + \texttt{OPT}(p(j)), \texttt{OPT}(j-1)\}.$$

In sintesi

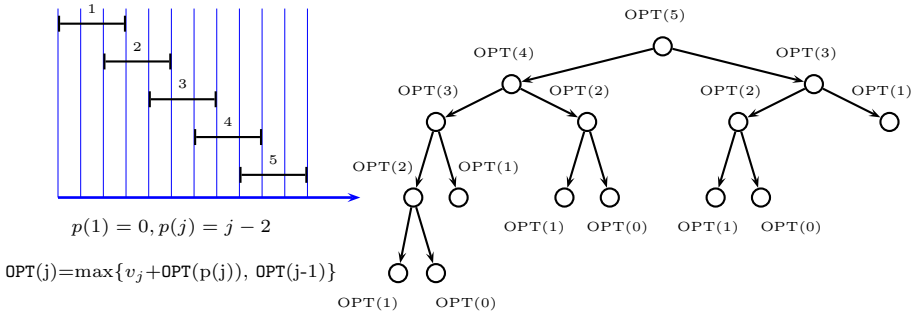
$$\texttt{OPT}(j) = \begin{cases} 0 & \text{se } j = 0, \\ \max\{v_j + \texttt{OPT}(p(j)), \texttt{OPT}(j-1)\} & \text{se } j \geq 1 \end{cases}$$

Il che ci suggerisce il seguente algoritmo

Input: $n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n$
1. `Ordina le attività in modo che` $f_1 \leq \ldots \leq f_n$
2. `Calcola` $p(1), \ldots, p(n)$
3. `Calcola-OPT`(j)
4. `IF`$(j == 0)$ {
5. `RETURN` 0
6. } `ELSE` {
7. `RETURN` $\max\{v_j +$ `Calcola-OPT`$(p(j))$, `Calcola-OPT`$(j-1)\}$
8. }

Noi siamo interessati alla esecuzione di `Calcola-OPT`(n). Purtroppo, l'algoritmo `Calcola-OPT`(n) ha complessità esponenziale nel caso peggiore. Supponiamo infatti che i tempi di inizio e fine di ogni attività siano tali che ogni attività A_j inizia prima che l'attività A_{j-1} sia terminata, ma dopo la terminazione di A_{j-2}, per $j = 2, \ldots, n$. Ciò comporta che $p(j) = j - 2$, per $j = 2, \ldots, n$. Ma se questo è il caso, si vede dalla linea 7. dell'algoritmo prima riportato, che `Calcola-OPT`(n) effettua due chiamate ricorsive a se stesso, ovvero a `Calcola-OPT`$(n-2)$ e `Calcola-OPT`$(n-1)$. Per cui, detta $T(n)$ la complessità di `Calcola-OPT`(n), ciò comporta che $T(n) = T(n-2) + T(n-1) + d$. Già sappiamo che questa equazione di ricorrenza ha soluzione esponenziale, purtroppo!

Perchè l'algoritmo di D&I `Calcola-OPT`(n) è esponenziale? Perchè risolve stessi problemi più volte, come si può vedere, as esempio, guardando l'albero delle chiamate ricorsive di `Calcola-OPT`(5).



É il momento, quindi, di usare la Programmazione Dinamica. Ricordiamo che esistono due approcci per trasformare un inefficiente algoritmo di Divide et Impera in un efficiente algoritmo.

La prima, basata sulla tecnica della Memoization, che aggiunge all'algoritmo una tabella in cui vengano memorizzate le soluzioni ai sottoproblemi giá risolti. Viene altresí addottata l'addizionale accortezza che prima di ogni chiamata ricorsiva dell'algoritmo su di un particolare sottoproblema, debba essere effettuato un controllo sulla tabella per verificare se la soluzione a quel sottoproblema è stata già calcolata in precedenza.

La seconda tecnica risolve semplicemente tutti i sottoproblemi del problema di partenza, in maniera iterativa ed in modo "bottom-up", ovvero risolvendo prima i sottoproblemi di taglia piccola e poi via via quelli di taglia maggiore fino a risolvere l'intero problema di partenza

Quale approccio è migliore? Dipende... Entrambi hanno i loro meriti.

L'approccio basato sulla memorizzazione preserva la struttura ricorsiva tipica degli algoritmi basati su Divide et Impera (che sono in generale semplici ed eleganti). Per contro, vi è un'aggiunta di lavoro, tipo gestione stack, etc., che in certe situazioni può diventare significativo.

L'approccio iterativo "bottom-up" è in generale efficiente. Tuttavia, gli algoritmi basati su questo approccio tendono a calcolare la soluzione a *tutti* i sottoproblemi del problema originale, anche quelli che potrebbero non concorrere alla soluzione ottima del problema di partenza. Ciò non accade per gli algoritmi basati sul primo approccio, che risolvono solo i sottoproblemi "strettamente necessari".

Vediamo in dettaglio l'applicazione della tecnica Programmazione Dinamica mediante il primo approccio, ovvero usando la Memoization. Ovvero, memorizziamo le soluzioni di ciascun sottoproblema, e le ri-leggiamo all'occorrenza.

Input: $n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n$ % fà uso di un'array M
1. `Ordina le attività in modo che` $f_1 \leq \ldots \leq f_n$
2. `Calcola` $p(1), \ldots, p(n)$
3. `M_Calcola_OPT`(j)
4. `IF`$(j == 0)$ {
6. `RETURN` 0
7. } `ELSE` {
8. `IF`($M[j]$ non è definito) {
9. $M[j] = \max\{v_j +$ `M_Calcola_OPT`$(p(j))$, `M_Calcola_OPT`$(j-1)\}$
}
10. `RETURN` $M[j]$

L'istruzione 1. richiede tempo $O(n \log n)$ per ordinare. L'istruzione 2. richiede tempo $O(n \log n)$ (una volta aver ordinato).

`M_Calcola_OPT`(n) effettua chiamate al suo interno a `M_Calcola_OPT`(j), $j < n$. Ogni chiamata richiede tempo $O(1)$ e o ritorna un valore $M[j]$ già calcolato, oppure calcola un nuovo valore $M[j]$ facendo due chiamate a valori già calcolati. Il numero totale di chiamate sarà quindi al più pari a $2n$, da cui segue che il tempo impiegato da `M_Calcola_OPT`(n) è $O(n)$. Sommando tutto otteniamo che l'algoritmo ha complessità $O(n \log n)$.

E se vogliamo trovare la soluzione ottima? (e non solo il suo valore). Il seguente algoritmo lo fà.

1. `M_Calcola_OPT`(n)
2. `Trova_Soluzione`(n)

dove

```
Trova_Soluzione(j)
1. IF(j==0) {
2.   RETURN nulla
3.   } ELSE {
4.     IF(v_j + M[p(j)] > M[j − 1]) {
5.       stampa j
6.     Trova_Soluzione(p(j))
7.   } ELSE {Trova_Soluzione(j − 1)
      }
   }
```

Il numero di chiamate ricorsive è $\leq n$. Di conseguenza, la complessità dell'algoritmo è $O(n)$.

$\diamond$

Applichiamo ora la tecnica Programmazione Dinamica al seguente problema. Consideriamo un gioco in cui vi sono n monete di valori $v_1, \ldots, v_n$, rispettivamente, disposte su di una riga da sinistra a destra (nell'ordine indicato), con n pari, e due giocatori: Alice e Bob. All'istante 1 Alice sceglie una delle due monete all'estremità della riga (quindi, all'inizio del gioco Alice può scegliere una tra le monete di valore v_1 e v_n), e la rimuove dalla riga. All'istante 2 Bob sceglierà una delle due monete all'estremità della riga *rimanente* (quindi, Bob può scegliere una tra le monete di valore v_2 e v_n, se Alice ha scelto nel turno precedente la moneta di valore v_1, oppure Bob potrà scegliere una tra le monete di valore v_1 e v_{n-1}, se Alice ha scelto nel suo turno la moneta di valore v_n), e la rimuove dalla riga. Il gioco procede in questo modo, a turni alterni, fino a quando le monete sono esaurite. Vince il giocatore che, alla fine del gioco, ha scelto le $n/2$ monete di valore totale *massimo*.

Il problema algoritmico che vogliamo risolvere è quello di determinare il valore massimo che il primo giocatore può ottenere (sotto l'ipotesi che anche il suo opponente giochi in maniera razionale).

Per farci un pò di intuizione, consideriamo l'esempio in cui si hanno 4 monete, di valore $2, 4, 8, 10$ e disposte nell'ordine indicato. Non è difficile vedere che il massimo valore che il primo giocatore può ottenere è pari a 14 (scegliendo le monete di valore 10 e 4).

Se le monete fossero invece $8, 20, 3, 2$ allora il massimo valore che il primo giocatore può ottenere è pari a 22 (scegliendo le monete di valore 2 e 20). Notiamo che la semplice strategia di scegliere ad ogni passo la moneta disponibile di valore maggiore non ci porta alla soluzione migliore. Infatti, se il primo giocatore seguisse questa strategia, sceglierebbe al primo passo la moneta di valore 8, l'avversario sceglierà la moneta di valore 20, ed il primo giocatore sceglierebbe la moneta di valore 3, per un valore totale $=11 < 22$.

Il fenomeno per cui effettuare la scelta della moneta disponibile di valore maggiore (ovvero la moneta di valore massimo disposta nelle due estremità della sequenza di monete) può non portarci alla soluzione migliore possibile, può essere generalizzato. Ad esempio, si consideri la sequenza di valori $\frac{v}{2} - 1, \frac{v}{2} - 1, \frac{3v}{2}, \frac{v}{2} + 1$. La strategia che sceglie sempre la moneta di maggior valore disponibile porterebbe il primo giocatore a scegliere le monete di valore $\frac{v}{2} + 1$ e $\frac{v}{2} - 1$, di valore totale v. Invece, la strategia migliore consiste nello scegliere prima la moneta di valore $\frac{v}{2} - 1$ e poi quella di valore $\frac{3v}{2}$, per un valore totale pari a $\frac{5v}{4} - 1$, che è maggiore di v per *tutti* i valori di $v > 4$.

Studiamo ora il caso generale. Sia $n > 0$ e sia data una sequenza di valori di monete $v_1, \ldots, v_n$. Assumiamo che Alice giochi per prima. Indichiamo con $f(i, j)$ il *massimo* valore che Alice può ottenere sulla sottosequenza

di valore $v_i, \ldots, v_j$ (assumendo che l'avversario Bob si comporti in maniera razionale, ovvero che cerchi di *massimizzare* il suo profitto, e quindi di *minimizzare* il profitto di Alice). Chiaramente, se $i = j$ varrà $f(i, j) = v_i$. Un'altro caso semplice è il calcolo di $f(i, i + 1)$ che ovviamente è pari a $\max\{v_i, v_{i+1}\}$. Consideriamo quindi il caso generale in cui $i < j - 1$. Varrà

$$f(i,j) = \max\Big(v_i + \min\{f(i+2,j), f(i+1,j-1)\}, v_j + \min\{f(i+1,j-1), f(i,j-2)\}\Big). \tag{1}$$

Per giustificare la (1), consideriamo distintamente i due casi possibili, ovvero che Alice scelga la moneta di valore v_i o quella di valore v_j.

- Alice sceglie la moneta di valore v_i. In questo caso, Bob sceglierà o la moneta di valore v_{i+1} (e quindi al passo successivo Alice dovrà poi scegliere tra le monete $v_{i+2} \ldots v_j$, ovvero risolvere $f(i + 2, j)$) oppure la moneta di valore v_j (e quindi al passo successivo Alice dovrà poi scegliere tra le monete $v_{i+1} \ldots v_{j-1}$, ovvero risolvere $f(i + 1, j - 1)$), a seconda di quale massimizza il suo profitto (ovvero *minimizza* il profitto di Alice). Detto in altri termini, Bob per massimizzare il suo profitto effettuerà una scelta che porterà al sottoproblema che ottiene il $\min\{f(i + 2, j), f(i + 1, j - 1)\}$. Tale sottoproblema è quello che poi dovrà risolvere Alice nel suo successivo turno.
- Alice sceglie la moneta di valore v_j. La discussione è analoga.

Di conseguenza, Alice sa che se sceglierà la moneta di valore v_i *allora* Bob effettuerà la scelta che ottiene il $\min\{f(i + 2, j), f(i + 1, j - 1)\}$, mentre se sceglierà la moneta di valore v_j *allora* Bob effettuerà la scelta che ottiene il $\min\{f(i+1, j-1), f(i, j-2)\}$. Di conseguenza, per massimizzare il suo profitto, tra le monete di valore v_i e v_j Alice sceglierà quella che ottiene il max indicato nella (1).

Per esercizio, si derivi l'algoritmo di Programmazione Dinamica che calcola $f(1, n)$ usando la ricorrenza (1) e le condizioni iniziali $f(i, i) = v_i$ e $f(i, i + 1) = \max\{v_i, v_{i+1}\}$. Inoltre, se ne valuti la complessità.