13 Programmazione Dinamica

La *programmazione dinamica* è caratterizzata da 4 fasi principali:

1. caratterizzare *matematicamente* la struttura di una soluzione ottima;
2. definire ricorsivamente il valore di una soluzione ottima;
3. calcolare il valore di una soluzione ottima con approccio bottom-up
   - utilizzando quindi una tabella per memorizzare la soluzione dei sottoproblemi ed utilizzarla per evitare di ripetere i calcoli più volte.
4. Ricostruire la soluzione ottima.

13.1 Numeri di fibonacci

```
int fibonacciRic(int n)
    se n ⩽ 1 allora
        ritorna 1
    allora
        ritorna fibonacciRic(n − 1) + fibonacciRic(n − 2)
```

$$T(n) = \begin{cases} 1 & n \leqslant 1 \\ T(n-1) + T(n-2) + 1 & n > 1 \end{cases}$$

Complessità É una ricorrenza lineare di ordine costante:

- $a_1 = 1, a_2 = 1, a = a_1 + a_2 = 2, \beta = 0$
- Complessità: $\Theta(a^n \cdot n^\beta) = \Theta(2^n)$

Dall'albero di ricorsione possiamo notare che molti sottoproblemi vengono ripetuti.

13.1.1 Come evitare di risolvere un problema più di una volta

Quando risolviamo un (sotto)problema, memorizziamo il risultato che otteniamo in una tabella (che può essere implementata come un vettore, una matrice, un dizionario...). La tabella dovrà contenere un elemento per ogni sottoproblema che dobbiamo risolvere. I casi base possono essere memorizzati nelle posizioni relative. Dopodiché l'iterazione è bottom-up, ovvero si parte dai casi base e si va verso problemi via via sempre più grande.

```
int fibonacciIter(int n)
    DP ← new int[0 . . . n]
    DP[0] ← DP[1] ← 1
    da i ← 2 fino a n fai
        DP[i] ← DP[i − 1] + DP[i − 2]
    ritorna DP[n]
```

Analisi delle complessità La complessità in tempo è lineare, come la complessità spaziale. Possiamo ridurre lo spazio utilizzato.

```
int fibonacci(int n)
    int DP_0 = 1
    int DP_1 = 1
    int DP_2 = 1
    da i = 2 fino a n fai
        DP_0 = DP_1
        DP_1 = DP_2
        DP_2 = DP_0 + DP_1
    ritorna DP_2
```

Analisi della complessità Questa implementazione ha costo costante nello spazio.

Funzione	Complessità	
	tempo	spazio
ricorsiva	$\mathcal{O}(n2^n)$	$\mathcal{O}(n^2)$
iterativa	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
finale	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$

13.2 Hateville

Questo problema è stato presentato la prima volta nel compito del

$$HV(i) = \begin{cases} 0 & i = 0 \\ \{1\} & i = 1 \\ highest(HV(i-1), HV(i-2) \cup \{i\}) & i \geqslant 2 \end{cases}$$

$$HV(i) = \begin{cases} 0 & i = 0 \\ \{1\} & i = 1 \\ max(DP(i-1), DP(i-2) \cup \{i\}) & i \geqslant 2 \end{cases}$$

13.2.1 Soluzione con linguaggi di programmazione

Analisi della complessità

13.3 Zaino

Definizione informale del problema

Definizione formale del problema

Analisi della complessità Non si può fare meglio di così.

Osservazione. *Non tutti gli elementi della matrice sono necessari alla risoluzione del nostro problema.*

Ad esempio

Nota. *Questo ci porta ad una nuova tecnica.*

13.4 Memoization

La *memoization* (annotazione) è una tecnica che fonde l'approccio di *memorizzazione* della programmazione dinamica con l'approccio *top-down* di divide-et-impera.
Quando devo risolvere un sottoproblema controllo prima se l'ho già risolto guardando nella cella corrispondente della tabella dove memorizzo i risultati, altrimenti lo calcolo on-the-fly (al momento) chiamando ricorsivamente i sottoproblemi e scrivo i rispettivi risultati nella tabella.
In questo modo mi assicuro di fare il calcolo una ed una volta sola ed evito di calcolare valori che non verranno mai usati.
La tabella viene inizializzata esternamente, nella funzione *wrapper*, con un valore che non viene usato durante la procedura (-1 nel caso vengano usati *solo* valori positivi, $-\infty$ nel caso vengano usati sia valori positivi che valori negativi)

Scelta della struttura dati É meglio scegliere una tabella o un dizionario?
Il costo di esecuzione è $\mathcal{O}(min(2^n, nC))$.

Nota. *La ricostruzione della soluzione è lasciata come esercizio.*

13.5 Zaino senza limiti (nella scleta degli oggetti)

Definizione informale del problema

- $c == 0$
- $pos[c] < 0$

Analisi della complessità Ognuno degli elementi costa $\Theta(n)$ per essere riempito, mal che vada devo riempire tutte le C caselle delle tabelle. Quindi nel caso pessimo ho una complessità di $\mathcal{O}(nC)$

13.6 Sottosequenza comune massimale

Defizione formale del problema Date due sequenze P e T di lunghezza n e m, rispettivamente, trovare la più lunga sottosequenza comune di P e T.
Una soluzione banale è quella di cercare tutte le sottosequenze e di prendere quella di lunghezza maggiore.
La quale ha una complessità che non vogliamo apgare inq uanto ci sono soluzioni più efficienti.

13.6.1 Descrizione matematica della soluzione ottima

13.7 String matching approssimato

- Una stringa $P = p_1 \dots p_m$ (*pattern*)
- Una stringa $T = p_1 \dots p_n$ (*testo*), con $m \leqslant n$

Definizione 13.1 (occorrenza k-approssimata)**.** *Un'occorrenza k-approssimata di un* pattern*(P) in un* testo*T, con $0 \leqslant k \leq m$, è una copia di P in T in cui sono ammessi k "errori" (o differenze) tra caratteri di P e caratteri di T, del seguente tipo:*

1. i corrispondenti caratteri in P e in T sono diversi (*sostituzione*);
2. un carattere in P non è incluso in T (*inserimento*);
3. un carattere in T non è incluso in P (*cancellazione*).

L'obiettivo è trovare un'occorrenza k-approssimata di P in T per cui k sia minimo.

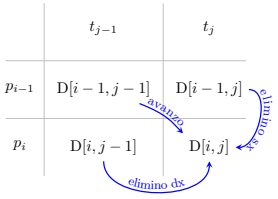
Definizione 13.2 (tabella di programmazione dinamica)**.** *Sia $DP[0 \dots m, 0 \dots n]$ una tabella di programmazione dinamica tale che $DP[i][j]$ contiene il minimo valore k per cui esiste un'occorrenza k-approssimata di $P(i)$ in $T(j)$, che termina nella posizione j.*

Esistono quattro possibilità:

1. $DP[i-1][j-1]$, se $p_i = t_j$ avanza su entrambi i caratteri;
2. $DP[i-1][j-1]+1$, se $p_i \neq t_j$ avanza su entrambi i caratteri (*sostituzione*);
3. $DP[i-1][j]+1$ avanza sul pattern (*inserimento*);
4. $DP[i][j-1]+1$ avanza sul testo (*cancellazione*).

Ricostruzione della soluzione finale:

- $DP[m][j] = k$ se e solo se esiste un'occorrenza k-approssimata di P in $T(j)$ che termina nella posizione j;
- La soluzione del problema è data dal più piccolo valore $DP[m][j]$, per $0 \leqslant j \leqslant n$.



(a) figura 1

P/T		A	B	A	B	A
	0	0	0	0	0	0
B	1	1	0	1	0	1
A	2	1	1	0	1	0
B	3	2	1	1	0	1

(b) figura 2

```
int stringMatching(ITEM[ ] P, ITEM[ ] T, int m, int n)
    // inizializzo i casi base
    int[ ][ ] DP ← new int[0...m][0...n]
    da j ← 0 fino a n fai                                   // Caso base: j ← 0
        DP[0][j] ← 0
    da i ← 1 fino a m fai                                   // Caso base: i ← 0
        DP[i][0] ← i

    // riempio la tabella
    da i ← 1 fino a m fai                                   // Caso generale
        da j ← 1 fino a n fai
            int temp ← DP[i − 1][j − 1] + iif(P[i] ← T[j], 0, 1)   // 0: uguali, 1:
             sostituzione
            temp ← min(temp, DP[i − 1][j] + 1)              // DP[i − 1][j] inserimento
            temp ← min(temp, DP[i][j − 1] + 1)              // DP[i][j − 1] rimozione
            DP[i][j] ← temp

    // cerco la posizione del minimo
    int min ← DP[m][0] // minimo
    int pos ← 0 // posizione del minimo

    da j ← 1 fino a n fai // trova il minimo sull'ultima riga
        se DP[m][j] < min allora
            min ← DP[m][j]
            pos ← j

    ritorna pos // potrei anche restituire DP[m][pos] che è il valore min associato
```

Nota. *Non è detto che la "soluzione finale" si trovi nella casella "in basso a destra" ; è invece possibile che la soluzione debba essere ricercata all'interno della tabella.*

13.8 Catena di moltiplicazione di matrici

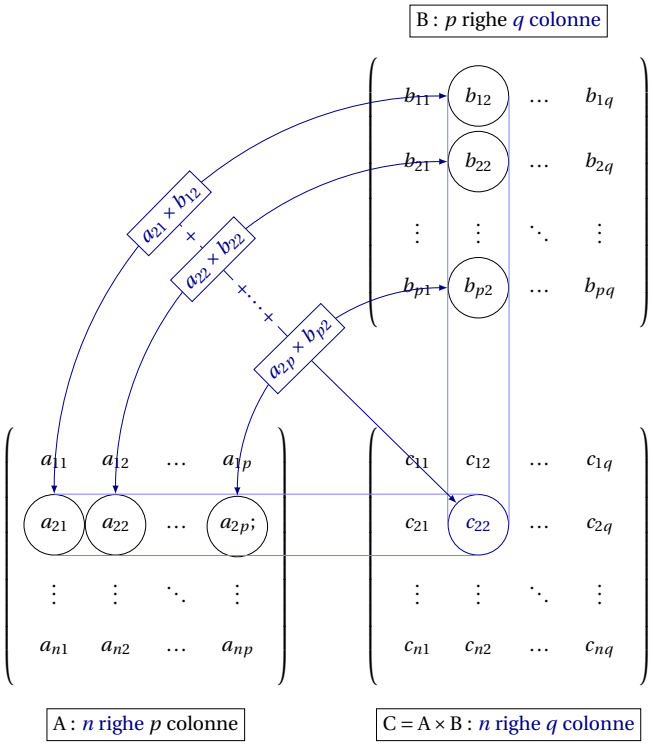


Figura 2: Moltiplicazione fra matrici

Il prodotto fra matrici non è *commutativo* (in quanto le matrici devono essere compatibili righe per colonne) però è *associativo*. Vogliamo quindi calcolare il prodotto di una catena di matrici impiegando il minor numero possibile di moltiplicazioni scalari.
Iniziamo caratterizzando matematicamente il nostro problema definendo la parentesizzazione $P[i \dots j]$ del prodotto $A_i \cdot A_{i+1 \dots A_j}$ come:

- la matrice A_i se gli indici corrispondono $(i = j)$;
- il prodotto di due parentesizzazioni $P_{i,k} \cdot P_{k+1,j}$ altrimenti.

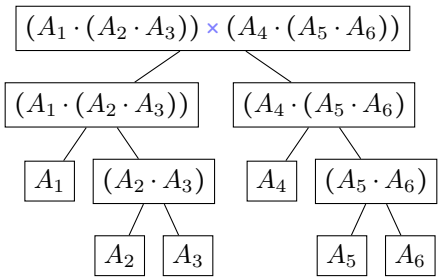


Figura 3: In questo caso $k = 3$ e il prodotto evidenziato è detto *ultimo prodotto*.

13.8.1 Quantificare il numero di parentesizzazioni possibili

Per trovare la soluzione ottima, nel nostro caso la parentesizzazione ottima (cieè la parentesizzazione che richiede il minor numero di moltiplicazioni scalari per essere completata, fra tutte le parentesizzazioni possibili) dobbiamo essere in grado di quantificarle e calcolarle empiricamente.
La motivazione alla base è che conviene Vale la pena preprocessare i dati per cercare la parentesizzazione migliore, per risparmiare tempo dopo nel calcolo vero e proprio.
Cercare di enumerare tutte le possibili combinazioni (approcciando il problema con la forza bruta) per poi scegliere la parentesizzazione migliore *non è possibile*, in quanto le combinazioni crescono come i numeri catalani (esponenzialmente) il che non ci permette di scrivere un algoritmo polinomiale, ma bensì esponenziale.

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{i=1}^{n-1} P(k)P(n-k) & n > 1 \end{cases}$$

Figura 4: numero di parentesizzazioni per n matrici $A_1 \cdot \ldots \cdot A_n$. $P(n) = \Omega(2^n)$

Aggiungiamo un po' di notazione matematica per semplificare la scrittura:

notazione...	... indica
$A_1 \cdot A_2 \cdot A_3 \ldots A_n$	il prodotto di n matrici da ottimizzare
c_{i-1}	il numero di *righe* della matrice A_i
c_i	il numero di *colonne* della matrice A_i
$A[i \ldots j]$	il sottoprodotto $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$
$P[i \ldots j]$	una parentesizzazione per $A[i \ldots j]$

Tabella 1: Definizioni matematiche

Nota. *$P[i \ldots j]$ non dev'essere necessariamente una parentesizzazione ottima.*

Indipendentemente dalle operazioni che facciamo il risultato avrà sempre il numero di righe della prima matrice ed il numero di colonne dell'ultima matrice.

Teorema 1 (sottostruttura ottima)**.** *Se $P[i \ldots j] = P[i \ldots k] \cdot P[k+1 \ldots j]$ è una parentesizzazione ottima del prodotto $A[i \ldots j]$, allora:*

- *$P[i \ldots k]$ è una parentesizzazione ottima del prodotto $A[i \ldots k]$;*
- *$P[k+1 \ldots j]$ è una parentesizzazione ottima del prodotto $A[k+1 \ldots k]$.*

Dimostrazione per assurdo. Supponiamo esista un parentesizzazione ottima (diversa) $P'[i \ldots k]$ di $A[i \ldots k]$ con costo inferiore a $P[i \ldots k]$.
Allora, la moltiplicazione $P'[i \ldots k] \cdot P'[k+1 \ldots j]$ sarebbe una parentesizzazione di $A[i \ldots j]$ con costo inferiore a quella ottima $P[i \ldots j]$, ma siccome quella ottima deve avere un costo minimo questo è assurdo. □

In altre parole il teorema afferma che esiste una *sottostruttura ottima*. Ogni soluzione ottima al problema della parentesizzazione contiene al suo interno le soluzioni ottime dei due sottoproblemi.
L'esistenza di sottostrutture ottime ci indica che in questo problema, *la programmazione dinamica è applicabile*.
Proseguiamo quindi defininendo ricorsivamente il costo di una soluzione ottima.

13.8.2 Valore della soluzione ottima

Sia $DP[i][j]$ il minimo numero di moltiplicazioni scalari necessarie per calcolare il prodotto $A[i \dots j]$.

- caso base: $i = j$ Allora $DP[i][j] = 0$ (non è necessario effettuare moltiplicazioni)
- caso ricorsivo $i < j$ Esiste una parentesizzazione ottima

$$P[i \dots j] = P[i \dots k] \cdot P[k+1 \dots j]$$

che può essere definita (sfruttando la ricorsione)

$$DP[i][j] = DP[i][k] + DP[k+1][j] + \underbrace{c_{i-1} \cdot c_k \cdot c_j}_{\text{costo ultima moltiplicazione}}$$

Nota. *$c_{i-1} \cdot c_k \cdot c_j$ è il costo della moltiplicazione delle c_{i-1} righe della prima matrice per le c_k colonne della seconda matrice (che corrispondono alle righe della terza matrice), il cui risultato viene a sua volta moltiplicato per le c_j colonne della quarta matrice.*

Non posso sapere a priori quale sia il valore ottimale per k quindi li provo tutti prendendo in considerazione che può assumere tutti i valori fra i e $j-1$. Ad esempio se ho tre matrici A_1, A_2, A_3 le due parentesizzazioni possibili sono $((A_1 \cdot A_2) \cdot A_3)$ e $(A_1 \cdot (A_2 \cdot A_3))$ quindi k può assumere valori da 1 a 2.

$$DP[i][j] = \begin{cases} 0 & i = j \\ \min\limits_{i \leqslant k < j} \{DP[i][k] + DP[k+1][j] + c_{i-1} \cdot c_k \cdot c_j\} & i < j \end{cases}$$

13.8.3 Dalla formula al codice

Le frecce uniscono i costi che devono essere sommati fra di loro.

Approccio top-down

```
int recPar(int[ ] c, int i, int j)
    se i ← j allora // se gli indici corrispondono
        ritorna 0 // non devo fare operazioni
    allora
        min ← +∞
        // Tutta la logica dell'algoritmo
        da int k ← i fino a j − 1 fai
            int q ← recPar(c, i, k) + recPar(c, k + 1, j) + c[i − 1] · c[k] · c[j]
            se q < min allora // se q è più piccolo del minimo
                min ← q // aggiorniamo il minimo
    ritorna min
```

Complessità Un'approccio ricorsivo top-down alla risoluzione del problema è illustrato nell'algoritmo recPar. É possibile dimostrare che l'algoritmo è $\Omega(2^n)$, che non è poi tanto meglio dell'approccio basato su forza bruta. Questa complessità deriva dal fatto che molti sottoproblemi vengono risolti più volte. É proprio in casi come questi che la programmazione dinamica ci viene incontro.

Approccio bottom-up

Utilizziamo quindi due tabelle per contenere i risultati intermedi:

1. $DP[i][j]$ contiene il numero di moltiplicazioni scalari necessarie per moltiplicare le matrici $A[i \dots j]$;
2. $last[i][j]$ contiene il valore k dell'ultimo prodotto che minimizza il costo per il sottoproblema.

```
computePar(int stuff)
    DP ← new int[1...n][1...n]
    last ← new int[1...n][1...n]

    // riempi diagonale principale
    da i ← 1 fino a n fai
        DP[i][i] ← 0

    // Tutta la logica dell'algoritmo
    da h ← 2 fino a n fai                                   // h: indice diagonale
        da i ← 1 fino a n − h + 1 fai                       // i: riga
            int j ← i + h − 1                               // j: colonna
            DP[i][j] ← +∞
            da k ← i fino a j − 1 fai
                int temp ← DP[i][k] + DP[k + 1][j] + c_{i−1} · c_k · c_j
                se temp < DP[i][j] allora
                    // aggiorna l'ultimo prodotto
                    DP[i][j] ← temp
                    last[i][j] ← k
```

L'algoritmo lavora per riempiendo una diagonale alla volta tramite la variabile h il cui valore varia sulle diagonali sopra quella principale, mentre i e j assumono valori delle celle nella diagonale h. Infatti la variabile i assume valori che variano da 1 a $n - h + 1$. Ad esempio al primo ciclo $i = 1$, $h = 2$ e la variabile i assumerà il valore di $6 - 2 + 1 = 5$ evitando così di toccare la diagonale principale. Discorso analogo per la variabile j.
L'algoritmo calcola tutti i possibili valori e conserva solo il più piccolo. A differenza dell'algoritmo precedente non svogliamo più chiamate ricorsive ma riutilizziamo valori già calcolati.

- Un vettore $c[0 \dots n]$ contenente le dimensioni delle matrici;
  - $c[0]$ è il numero di righe della prima matrice
  - $c[i-1]$ è il numero di righe della matrice A_i
  - $c[i]$ è il numero di colonne della matrice A_i
- Due indici i, j che rappresentano l'intervallo di matrici da moltiplicare;
- Il costo della funzione si trova nella posizione $DP[1, n]$.

i	$c[i]$
0	7
1	8
2	4
3	2
4	3
5	5
6	6

DP	**1**	**2**	**3**	**4**	**5**	**6**
1	0	224	176	218	276	350
2		0	64	112	174	250
3			0	24	70	138
4				0	30	90
5					0	90
6						0

LAST	**1**	**2**	**3**	**4**	**5**	**6**
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

$$
\begin{aligned}
DP[1][4] &= min_{1\le k<4}\{DP[1,k] + DP[k+1,4] + c_0\cdot c_k\cdot c_4\} \\
&= min \quad \{DP[1,1] + DP[2,4] + c_0\cdot c_1\cdot c_4, \\
&\qquad\; DP[1,2] + DP[3,4] + c_0\cdot c_2\cdot c_4, \\
&\qquad\; DP[1,3] + DP[4,4] + c_0\cdot c_3\cdot c_4\} \\
&= min \quad \{0 + 112 + 7\cdot 8\cdot 3, \\
&\qquad\; 224 + 24 + 7\cdot 4\cdot 3, \\
&\qquad\; 176 + 0 + 7\cdot 2\cdot 3\} \\
&= min \quad \{280 , 332 , 218\}
\end{aligned}
$$

Complessità Il costo computazionale è $\mathcal{O}(n^3)$ in quanto ogni cella richiede, nel caso peggiore, un tempo $\mathcal{O}(n)$ per essere riempita.

13.8.4 Ricostruzione della soluzione ottima

É inoltre necessario mostrare la soluzione trovata, per questo motivo abbiamo registrato informazioni sulla soluzione nella matriche *last*.
Possiamo quindi definire un algoritmo che ricostruire l'informazione a partire dall'informazione calcolata: $last[i,j]$ contiene il valore k su cui dobbiamo spezzare il prodotto $A[i\dots j]$. Ci suggerisce quindi che per calcolare $A[i\dots j]$ dobbiamo prima calcolare $A[i\dots k]$ e $A[k{+}1\dots j]$ e poi moltiplicarle fra di loro. Questo processo è facilmente realizzabile tramite un algoritmo *ricorsivo*.
Modifichiamo leggermente la funzione sopra nel modo seguente:

```
computePar(int c, int i, int n)
    [...]
    stampaPar (last, 1, n)
```

```
stampaPar(int[ ][ ] last, int i, int j)
    se i == j allora
        stampa "A["; stampa i; stampa "]"
    allora
        stampa "("                                          // aperta
        stampaPar(last, i, last[i...j])          // parentesizzazione ottima fino a k
        stampa "."                                  // segno di moltiplicazione
        stampaPar(last, last[i...j]+1, j)        // parentesizzazione ottima da k+1
        stampa ")"                                          // chiusa
```

13.8.5 Calcolo effettivo

La funziona sopra si limita a stampare la soluzione ma noi vogliamo effettuare il calcolo effettivo della soluzione.

```
int[ ][ ] multiply(int[ ][ ] A, int[ ][ ] S, int i, int j)
    se i == j allora
        ritorna A[i]
    allora
        int[ ][ ] X = multiply(last, i, last[i][j])
        int[ ][ ] Y = multiply(last, last[i][j] + 1, j)
        ritorna multiplyMatrices(X, Y)
```

Facendo un esempio

$A[1 \dots 6] = A[1 \dots 3] \cdot A[4 \dots 6]$
$A[1 \dots 3] = A_1 \cdot A[2 \dots 3]$
$A[4 \dots 6] = A[4 \dots 5] \cdot A_6$
$A[2 \dots 3] = A_2 \cdot A_3$
$A[4 \dots 5] = A_4 \cdot A_5$

LAST	**1**	**2**	**3**	**4**	**5**	**6**
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

Che risulta in $A = ((A_1 \cdot (A_2 \cdot A_3)) \cdot ((A_4 \cdot A_5) \cdot A_6))$

Nota. *A volte, bisogna fare attenzione a come riempire la tabella – non è detti che riempire una riga dopo l'altra sia possibile.*

13.9 Insieme indipendente di intervalli pesati

Nota. *Questo problema non è risolvibile solo tramite la programmazione dinamica.*

13.9.1 Definizione del problema

Siano dati n intervalli distinti $[a_1, b_1[, \dots, [a_n, b_n[$ della retta reale, aperti a destra, dove all'intervallo i è associato un profitto $w_i, 1 \leqslant i \leqslant n$.

Definizione 13.3 (Intervalli disgiunti)**.** *Due intervalli i e j si dicono* disgiunti *se: $b_j \leqslant a_i$ oppure $b_i \leqslant a_j$.*

L'obiettivo è trovare un *insieme indipendente di peso massimo*, ovvero un sottoinsieme di intervalli disgiunti tra loro tale che la somma dei loro profitti sia la più grande possibile. La una prenotazione di una sala conferenze è un esempio di questo genere di problemi. Riesci a trovare la soluzione ottima? La soluzione si trova sulla pagina successiva.

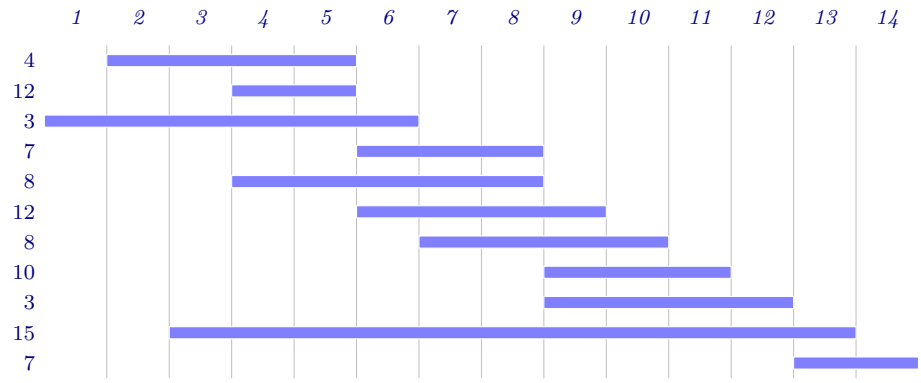


Figura 5: Un esempio di intervalli disgiunti: prenotazione di una sala conferenze, a sinistra sono indicati i *pesi* di ogni intervallo

13.10 Pre-elaborazione

13.10.1 Implementazione naive

Per usare la programmazione dinamica è necessario effetuare una pre-elaborazione: ordinare gli intervalli per estremi finali crescenti (per ordine di fine) $b_1 \leqslant b_2 \leqslant \cdots \leqslant b_n$.
$DP[i]$ contiene il profitto massimo ottenibile con i primi i intervalli.

$$DP[i] = \begin{cases} 0 & i = 0 \\ \max(DP[i-1],\ \max\{DP[j] + w_i : 1 \leqslant j < i \wedge b_j \leqslant a_i\}) & i > 0 \end{cases}$$

Figura 6: Prima versione

Devo compiere una scelta (e la scelta migliore è rappresentato dal fatto che scelgo quella con *profitto massimo*), le possibilità sono due:

- Non prendo l'intervallo, quindi tolgo un elemento ($DP[i-1]$);
- Prendo l'elemento, posso compiere questa scelta solo se l'intervallo è *compatibile* (ovvero se $j < i$ e (devono valere entrambe le proprietà) se l'intervallo finisce prima dell'inizio dell'intervallo considerato ($b_j \leqslant a_i$)).

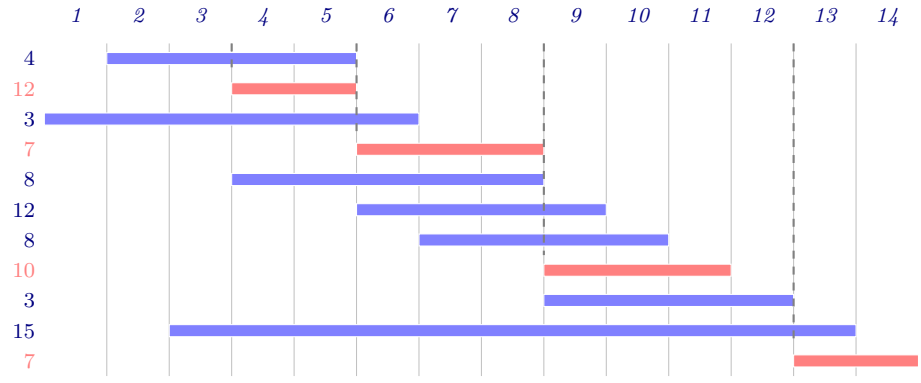


Figura 7: Soluzione ottima

Complessità Il costo computazionale associato a questa formula è $\mathcal{O}(n^2)$ perché per ognuno degli intervalli devo

13.10.2 Pre-calcolo dei predecessori

Una seconda possibile pre-elaborazione consiste nel pre-calcolare il *predecessore* $pred_i$ di i, dove j si riferisce all'intervallo subito antecedente a quello considerato ($j < i$ è il massimo indice tale che $b_j \leqslant a_i$)

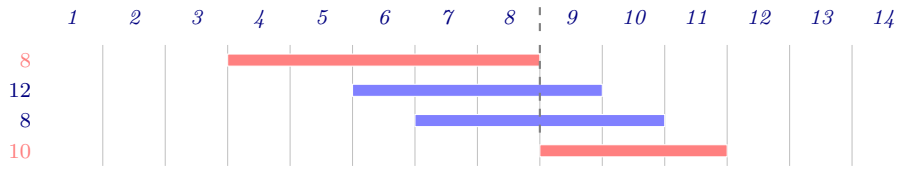


Figura 8: Predecessore

$$DP[i] = \begin{cases} 0 & i = 0 \\ \max(DP[i-1],\ DP[pred_i] + w_i) & i > 0 \end{cases}$$

Figura 9: Seconda versione

L'algoritmo di calcolo dei predecessori è illustrato di seguito

```
// Pre-computa i predecessori
int[][] computePredecessor(int[][] a, int[][] a, int n)
    int[][] pred ← new int[0...n]
    pred[0] ← 0
    da i ← 1 fino a n fai
        j ← i − 1
        finché j > 0 and b[j] > a[i] fai
            j ← j − 1
        pred[i] ← j
    ritorna pred
```

Complessità Pre-calcolare i predecessori viene a costare $\mathcal{O}(n^2)$, ma si può fare meglio di così.

```
SET maxinterval(int a, int b, int w, int n)
    { ordina gli intervalli per estremi di fine crescenti }
    int[ ][ ] pred ← computePredecessor(a, b, n)
    int[ ][ ] DP ← new int[0...n]
    DP[0] ← 0
    da i ← 1 fino a n fai
        DP[i] ← max(DP[i − 1], w[i] + DP[pred[i]])
    // Costruisco l'insieme dei predecessori
    i ← n
    SET S ← Set
    finché i > 0 fai // fintanto che ci sono intervalli disponibili
        se DP[i − 1] > w(i) + DP[pred[i]] allora // commento
            i ← i − 1 // non considerarlo
        allora
            S.insert(i) // inseriscilo nell'insieme
            i ← pred[i] // scorri gli interalli
    ritorna S // ritorna l'insieme di intervalli ordinati
```

Complessità La complessità di questa funzione è la sommatoria asintotica di tutte le sue parti

Funzione	**Costo**
Ordinamento intervalli	$\mathcal{O}(n \log n)$
Calcolo predecessori	$\mathcal{O}(n \log n)$
Riempimento tabella DP	$\mathcal{O}(n)$
Ricostruzione soluzione	$\mathcal{O}(n)$
	$\mathcal{O}(n \log n)$

Nota. *Talvolta, può essere necessario pre-processare l'input per poter applicare nella maniera più efficiente possibile la programmazione dinamica.*

13.11 Conclusioni

La programmazione dinamica non è la soluzione di tutti i vostri problemi. Esistono altre tecniche che possono fare "meglio di così". Inoltre, è possibile che soluzioni ad-hoc possano essere migliori.