

# Analisi e Progetto di Algoritmi

UniShare

Davide Cozzi  
@dlcgold

Gabriele De Rosa  
@derogab

Federica Di Lauro  
@f\_dila

# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
<b>2</b>	<b>Introduzione al corso</b>	<b>3</b>
2.1	Argomenti . . . . .	3
2.2	Ripasso Algoritmi 1 . . . . .	4
2.2.1	Equazioni di Ricorrenza . . . . .	4
<b>3</b>	<b>Programmazione Dinamica</b>	<b>9</b>
3.0.1	Programmazione Dinamica . . . . .	15
3.0.2	Un Problema di Sequenze . . . . .	15
3.0.3	Longest Common Substring . . . . .	16
3.0.4	Il Problema delle Scatole . . . . .	18
3.1	Longest Common Subsequence . . . . .	19

# Capitolo 1

## Introduzione

Questi appunti sono presi a lezione. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: <https://github.com/dlsgold/Appunti>.

Grazie mille e buono studio!

# Capitolo 2

## Introduzione al corso

### 2.1 Argomenti

Si hanno diversi tipi di problemi:

- **problemi di ottimo** dove si cercano singole soluzioni efficienti (massimi o minimi) tra molte soluzioni possibili. Si usa anche la **programmazione greedy**, dove si sceglie in base ai costi locali per ottenere massimi e minimi senza però guardare i costi complessivi.
- **problemi non risolubili in tempi accettabili**, per i quali si usa la **programmazione dinamica**, che cerca di individuare sotto-strutture ottime per risolvere il problema, cercando la soluzione migliore memorizzando le altre soluzioni e utilizzandole. Si cerca comunque la soluzione meno dispendiosa in termini di tempo.
- **problemi NP-completi**, ovvero problemi per cui non si può trovare un algoritmo o non si può trovare un algoritmo con una complessità asintotica polinomiale. Si useranno anche tecniche non deterministiche. Si cercherà di studiare uno dei 10 problemi più difficili della matematica:  $P \subseteq NP$ ?

Studieremo poi i grafi non pesati con gli algoritmi *BFS* (per cercare in ampiezza) e *DFS* (per cercare in profondità). Studieremo anche i grafi pesati con problemi di cammino minimo.

## 2.2 Ripasso Algoritmi 1

Innanzitutto due algoritmi con lo stesso scopo si possono confrontare in base a tempo e spazio, scegliendo anche in base alle esigenze hardware. Per lo spazio si calcola quanto spazio viene richiesto da variabili e strutture dati, soprattutto queste ultime che dipendono dalla dimensione dell'input. Per quanto riguarda il tempo si usano le tecniche di conto soprattutto basate sui cicli e, in generale, su tutte operazioni da effettuare. Il tempo si basa sull'input  $n$  e si indica con  $T(n)$  e si esprime in forma asintotica, interessandoci quindi unicamente all'ordine di grandezza. Si hanno il caso peggiore, indicato con l'O-grande e quello migliore indicato con l'o-piccolo a seconda di  $n$ .

Si ricorda poi la tecnica della ricorsione con algoritmi che si muovono su se stessi mediante dei "passi" arrivando ad un caso base di uscita. Per calcolare i tempi di un algoritmo ricorsivo si ha  $T(n) = F(n) + T(n - 1)$  con  $F$  che rappresenta le istruzioni delle subroutines. Questa equazione di ricorrenza non è facilmente calcolabile ma può essere espansa muovendosi sui passi fino a che non si arriva a qualcosa di calcolabile grazie al caso 0, questo è il metodo iterativo (anche se si ha anche il metodo per sostituzione). Per gli algoritmi ricorsivi si hanno anche i divide et impera (dove il problema  $P$  è diviso in sottoproblemi risolti separatamente, con la divide, e poi combinati alla fine, con la combina) dove i tempi non sono sempre calcolabili ma se lo sono si usa il metodo dell'esperto (studiando le tre possibili casistiche).

### 2.2.1 Equazioni di Ricorrenza

Le equazioni di ricorrenza hanno solitamente la seguente forma:

$$\begin{cases} T(n) = T(n - 1) + f(n) \\ T(1) = \Theta(1) \end{cases}$$

Esistono tre metodi per risolvere le equazioni di ricorrenza:

- Iterativo (detto anche Albero di ricorsione)
- Sostituzione
- Esperto (detto anche Principale)

**Metodo Iterativo**

Si può usare sia per algoritmi ricorsivi e per Divide et Impera. Ad ogni passo si prende il valore a destra dell'uguaglianza e lo si sostituisce, arrivando, dopo  $k$  passi ad una formula generale. Sempre  $k$  ci darà il caso base. Posso rappresentare questo metodo con l'albero delle chiamate ricorsive, guardando quanto è alto l'albero e quanto impiega ad ogni livello

**Esempio 1.** *Calcolo i tempi di:*

$$\begin{cases} T(N) = T(n-1) + 8 \\ T(1) = 6 \end{cases}$$

*procedo nella seguente maniera:*

$$\begin{aligned} T(n) &= T(n-1) + 8 = [T(n-2) + 8] + 8 = T(n-2) + 2 \cdot 8 \\ &= [T(n-3) + 8] + 2 \cdot 8 = T(n-3) + 3 \cdot 8 \\ &= [T(n-4) + 8] + 3 \cdot 8 = T(n-4) + 4 \cdot 8 \\ &= T(n-k) + k \cdot 8 \end{aligned}$$

*per  $k = n-1$  si ha:*

$$T[n - (n-1)] + (n-1) \cdot 8 = T(1) + (n-1) \cdot 8 = 6 + (n-1) \cdot 8 = \Theta(n)$$

*Altri esempi su sito e appunti di Chiodini*

**Metodo per Sostituzione**

Si ipotizza un tempo di calcolo (si possono usare gli asintotici con  $O$  e  $\Omega$  lo si dimostra per induzione

**Esempio 2.**

$$\begin{cases} T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n & n > 1 \\ T(1) & n = 1 \end{cases}$$

*Ipotizzo  $O(n \cdot \log n)$  e dimostro per induzione:*

$$T(n) = O(n \cdot \log n) \leq c \cdot n \cdot \log n$$

*Serve una dimostrazione forte: ipotizzo  $T(m)$  vera per  $1 \leq m \leq n-1$  quindi si ha:*

$$\begin{aligned} T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + n \leq 2 \cdot \left[c \cdot \frac{n}{2} \cdot \log \frac{n}{2}\right] + n \\ &= c \cdot n \cdot \log \frac{n}{2} + n = c \cdot n \cdot (\log_2 n - \log_2 2) + n \\ &= c \cdot n \cdot \log_2 n - c \cdot n + n \leq c \cdot n \cdot \log n \text{ se } c \geq 1 \end{aligned}$$

*Analizzo ora il caso base:*

$T(1) = 1$  quindi voglio  $1 \leq c \cdot \log_2 1$  ovvero  $1 \leq c \cdot 0$  ovvero mai. testo fino a che non trovo  $T(3) = 2 \cdot T(1) + 3 = 2 \cdot 1 + 3 = 5$  che mi va bene, infatti  $5 \leq c \cdot 3 \cdot \log_2 3$

### Metodo dell'Esperto

Posso usare questo metodo solo nel caso di un'equazione di ricorrenza di questo tipo:

$$\begin{cases} T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \\ T(1) = \Theta(1) \end{cases}$$

dove:

- $a \cdot T\left(\frac{n}{b}\right)$  è l'Impera ed è  $\sim n^{\log_b a}$
- $f(n)$  è il divide e il combina (ovvero la parte iterativa)

Si definiscono tre casi:

- **caso 1:**  $n^{\log_b a} > f(n)$  quindi  $T(n) \sim n^{\log_b a}$ . Si hanno le seguenti condizioni necessarie:  $f(n) = O(n^{\log_b a - \epsilon})$  (con  $\epsilon > 0$ ) e quindi  $T(n) = \Theta(n^{\log_b a - \epsilon})$
- **caso 2:**  $n^{\log_b a} \cong f(n)$  quindi  $T(n) \sim f(n) \cdot \log n$ . Si hanno le seguenti condizioni necessarie  $f(n) = \Theta(n^{\log_b a})$  e quindi  $T(n) = \Theta(n^{\log_b a})$
- **caso 3:**  $n^{\log_b a} < f(n)$  quindi  $T(n) \sim f(n)$ . Si hanno le seguenti condizioni necessarie:  $f(n) = \Omega(n^{\log_b a + \epsilon})$  (con  $\epsilon > 0$ ) e  $a \cdot f\left(\frac{n}{b}\right) \leq k \cdot f(n)$  (con  $k < 1$ ) quindi  $T(n) = \Theta(f(n))$

**Esempio 3.** *Risolve:*

$$T(n) = 9 \cdot T\left(\frac{n}{3}\right) + n$$

Si ha:  $f(n) = n$ ,  $a = 9$  e  $b = 3$ .

Ho che  $n^{\log_3 9} = n^2$  quindi ho il primo caso:

$f(n) = O(n^{\log_b a - \epsilon}) = O(n^{2 - \epsilon})$  Posso dire che  $\exists \epsilon : O(n^{2 - \epsilon}) = n$ ?

Si  $\forall \epsilon < 1$ , per esempio  $\epsilon = \frac{1}{2}$ . Quindi il Metodo dell'esperto è applicabile (nel primo caso) e si ha quindi  $T(n) = \Theta(n)$

**Esempio 4.** Si può analizzare meglio il MergeSort:

$$T(n) \cong 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$$

Si ha:  $f(n) = \Theta(n)$  e  $n^{\log_b a} = n^{\log_2 2} = n$

Posso applicare il Metodo dell'esperto nel secondo caso avendo così:

$$T(n) = \Theta(n \cdot \log n)$$

**Esempio 5.**

$$T(n) = 3 \cdot T\left(\frac{n}{4}\right) + n \cdot \log n$$

Si ha:  $f(n) = n \cdot \log n$  e  $n^{\log_b a} = n^{\log_4 3}$  e siamo nel terzo caso:

$$f(n) = \Omega(n^{\log_4 3 + \epsilon})$$

se pongo  $\epsilon = 1 - \log_4 3$  ottengo  $n$ . Il terzo caso richiede una doppia verifica:

$$3 \cdot \frac{n}{4} \cdot \log \frac{n}{4} \leq k \cdot n \log n$$

che vale per  $k = \frac{3}{4}$  infatti si ha:

$$\frac{3}{4} \cdot n \cdot \log \frac{n}{4} \leq \frac{3}{4} \cdot n \cdot \log n$$

Si hanno quindi entrambi i requisiti e si può asserire che  $T(n) = \Theta(n \cdot \log n)$

**Esempio 6.** Calcolo i tempi di:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n \cdot \log n$$

Si ha:  $n^{\log_b a} = n^{\log_2 2} = n$  e  $f(n) = n \cdot \log n$ . Provo a procedere col terzo caso, dimostrando che:

$$n \cdot \log n = \Omega(n^{\log_b a + \epsilon}) = \Omega(n^{1+\epsilon}) = \Omega(n \cdot n^\epsilon)$$

Ma tale  $\epsilon$  non esiste in quanto  $n^\epsilon > \log n$  infatti:

$$\lim_{n \rightarrow \infty} \frac{n \cdot \log n}{n \cdot n^\epsilon} = 0, \quad \forall \epsilon > 0$$

Bisogna quindi applicare un altro metodo per risolvere l'equazione di ricorrenza



**Esempio 7.** *Calcolo la seguente equazione di ricorrenza:*

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

*Quindi avrò un albero binario di soli 1 di profondità  $2^k$ . Quindi  $T(n) = \sum_{i=0}^k 2^i = 2^{k+1} - 1$  con  $k = \log n$  in quanto si avranno in totale  $n = 2^k = 2 \cdot 2^k - 1$ . Quindi ottengo  $2n - 1$  quindi avrò  $\Theta(n)$ .*

Abbiamo poi visto alcune strutture dati: *array*, *list*, *stack*, *queue*, *tree* (e *binary-tree*) e *heap*.

## Capitolo 3

# Programmazione Dinamica

Partiamo dall'algoritmo che calcola la lista di Fibonacci:

```
function FIB(n)  
  if n = 1 then  
    return n  
  else  
    return FIB(n - 1) + FIB(n - 2)  
  end if  
end function
```

Si vede che non sappiamo calcolarne la complessità, che non è polinomiale ma magari esponenziale o addirittura fattoriale.

Sia  $T(n)$  il costo della chiamata alla funzione. Se  $n = 0$  o  $n = 1$  ho  $T(n) = 1$ . Andando avanti avrò  $T(n) = 1 + T(n-1) + T(n-2)$  che non è risolvibile con le tecniche che conosciamo. Vediamo come risolverla: riscriviamo l'equazione non omogenea:

$$T(n) - T(n-1) - T(n-2) = 1$$

e facciamo una piccola approssimazione:

$$T(n) - T(n-1) - T(n-2) = 0$$

ottenendo un'equazione lineare omogenea a cui sommerò qualcosa per ottenere il risultato della non omogenea. Quindi risolvo l'omogenea ipotizzando un valore per  $T(n)$ , per esempio  $T(n) = r^n$ , e testiamolo, diventa:

$$r^n - r^{n-1} - r^{n-2} = 0$$

moltiplico da entrambe le parti per  $r^2$  perché posso:

$$r^2 \cdot r^n - r \cdot r^n - r^n = 0$$

$$r^2 - r - 1 = 0$$

che è un'equazione di secondo grado con soluzioni  $r = \frac{1 \pm \sqrt{5}}{2}$  Quindi

$$T(n) - T(n-1) - T(n-2) = 0$$

ha due soluzioni:

$$C_1 \left( \frac{1 + \sqrt{5}}{2} \right)^n$$
$$C_2 \left( \frac{1 - \sqrt{5}}{2} \right)^n$$

quindi:

$$T_0(n) = C_1 \left( \frac{1 + \sqrt{5}}{2} \right)^n + C_2 \left( \frac{1 - \sqrt{5}}{2} \right)^n$$

Ora cerco la soluzione particolare, sostituisco in:

$$T(n) - T(n-1) - T(n-2) = 1$$

Tutte le  $T(\cdot)$  con  $k$  ottenendo  $k - k - k = 1 \rightarrow k = -1$ .

Quindi la soluzione finale è:

$$T(n) = C_1 \left( \frac{1 + \sqrt{5}}{2} \right)^n + C_2 \left( \frac{1 - \sqrt{5}}{2} \right)^n - 1 = \Theta \left( \left( \frac{1 + \sqrt{5}}{2} \right)^n \right)$$

*che è la sezione aurea*

Miglioriamo l'algoritmo introducendo un array di  $n$  celle  $F$ , inizializzarlo

```
 $F[1 \dots n]$   
for  $i \leftarrow 1$  to  $n$  do  
     $F[i] \leftarrow \text{empty}$   
end for
```

e procedere con la ricorsione con annotazione, che scrive i vari step su un array (spreca quindi memoria) e modificando fibonacci per ottenere la versione con annotazione:

```
function FIBANN(n)  
  if f[i] == empty then  
    if  $n \leq 1$  then  
       $F[n] \leftarrow n$   
    else  
       $F[n] = \text{FIBANN}(n - 1) + \text{FIBANN}(n - 2)$   
    end if  
    return F[n]  
  end if  
end function
```

Quindi se si richiede qualcosa di già usato lo si ritorna prendendolo dall'array. Questo è esponenziale  
Iterativamente sarebbe:

```
function FIBIT(n)  
   $F[0] \leftarrow 0$   
   $F[1] \leftarrow 1$   
  for  $i \leftarrow 2$  to  $n$  do  
     $F[i] \leftarrow F[i - 1] + F[i - 2]$   
  end for  
  return F[n]  
end function
```

questo è polinomiale

### Un Nuovo Problema

Abbiamo una serie di task che possono essere svolte con un certo costo  $v_i$ , che partono in tempi diversi e non possono essere svolte contemporaneamente:



Per ogni attività si ha quindi un  $s_i$ , tempo di inizio, un  $f_i$ , tempo di fine, e un  $v_i$ , il costo, con  $i \dots n$  che indica l'attività. Definiamo  $A \subseteq \{1, \dots, n\}$  come l'insieme che contiene attività mutualmente compatibili sse:

$$\forall i, j \in A \ [s_i, f_i) \cap [s_j, f_j) \neq \emptyset$$

definiamo anche  $comp(A)$ :

$$\begin{cases} \text{true} & \text{se mutualmente compatibili} \\ \text{false} & \text{altrimenti} \end{cases}$$

che verifica se dei task sono mutualmente compatibili.

Inoltre  $V(A) = \sum_{i \in A} v_i$  per vedere il costo totale di una serie di task, con:

$$P(\{1, \dots, n\}) \rightarrow \{true, false\}$$

$$V : P(\{1, \dots, n\}) \rightarrow \mathbb{R}$$

Quindi la soluzione sarà un insieme di task tale che:

$$S \subseteq \{1, \dots, n\} \rightarrow comp(S) = true, \ V(S) = \max\{v(A)\}$$

Definiamo in maniera formale: l'istanza è una  $n \in \mathbb{N}$  e  $X_n \in \{1, \dots, n\}$ . Ad ogni attività  $i \in X_n$  ci sono associate il tempo di inizio  $s_i$ , di fine  $f_i$  e il valore  $v_i$ .

La soluzione è un sottoinsieme  $S \subseteq X_N$  di attività compatibili secondo la funzione  $comp$  e tale che  $v(S) = \max_{A \subseteq X_n, comp(A)=TRUE} \{v(A)\}$ . La soluzione basata sulla forza bruta (calcolo tutte le combinazioni) è  $\Omega(2^n)$  nel caso migliore; non va bene.

Usiamo quindi la programmazione dinamica devo però prima cercare la soluzione in termini ricorsivi, cercando i sottoproblemi. La struttura da "far

calare" è  $X_n$  a step di  $n - 1$  lavorando quindi su  $S_{n-1} \subseteq X_{n-1}$  tale che valga quanto sopra.

Il sottoproblema  $i$ -esimo sarà  $X_i = \{1, \dots, i\}$  con  $S_i \subseteq X_i$  con le condizioni di sopra.

Il sottoproblema con  $i = 1$  avrà  $X_1 = \{1\}$  fatto solo dalla prima attività e quindi avrà  $S_1 = \{1\}$  e quindi  $v(S_1) = v_1$ , ovvero 10 nel nostro caso.

Inoltre se  $i = 0$  avrò l'insieme vuoto che ha valore 0.

Quindi  $S_1 = \{1\}$ , con  $V(S_1) = 1$ , e  $S_0 = \emptyset$ , con  $v(s_0) = 0$  e questi potrebbero essere i miei casi base.

Ragioniamo su 6, se 6 è soluzione allora sarà insieme alla soluzione delle altre attività compatibili, ovvero quella di 4 essendo la prima compatibile in ordine, o meglio  $s_6 = \{6\} \cup S_4$ . Per la 5 sarà 1, per la 4 sarà 2, per la 3 sarà 1 e per la 2 non sarà nessuna, come per 1, e quindi sarà 0. Ho appena definito  $p(i) = \max\{j | j < i, j \text{ compatibile con } i\}$  assumendo che  $\max \emptyset = 0$ , quindi l'attività con indice maggiore compatibile precedente a quella in studio.

**Questo schema funziona per attività ordinate sull'ordine di fine.**

Quindi:

$$S_i = \begin{cases} \{i\} \cup S_{p(i)} & \text{se } i \in S_i \\ S_{i-1} & \text{se } i \notin S_i \end{cases}$$

e:

$$v(S_i) = \begin{cases} v_i + v(S_{p(i)}) \\ v(S_{i-1}) \end{cases}$$

Il modo di procedere consiste nel calcolare i valori e aggiornare il massimo:

$$v(S_i) = \max\{v_i + v(S_{p(i)}), v(S_{i-1})\}, i > 0$$

sapendo  $v(S_1) = 1$  e  $v(s_0) = 0$ .

Ma in realtà  $v(S_1) = 1$  è ricavabile da  $v(S_0)$  con la formula quindi avremo solo un caso base:  $v(S_0) = 0, i = 0$ .

Quindi abbiamo trovato la ricorrenza, considerando anche che:

$$S_i = \begin{cases} \{i\} \cup S_{p(i)} & \text{se } v_i + v(S_{p(i)}) \geq v(S_{i-1}) \\ S_{i-1} & \text{se } v_i + v(S_{p(i)}) < v(S_{i-1}) \end{cases}$$

Quindi ottengo l'algoritmo ricorsivo per calcolare i vari  $v_i$ :

```
function  $R - OPT(i)$ 
  if  $i == 0$  then
    return 0
  else
    return  $\max(v_i + R - OPT(p[i]), R - OPT(i - 1))$ 
  end if
end function
```

Ma è ingestibile dal punto di vista della complessità. Utilizzo quindi un vettore  $M[0..n]$ , introducendo la programmazione dinamica, dove in  $M[i]$  memorizzo il valore della soluzione del problema di taglia  $i$ , ovvero  $v(S_i)$ . Facciamo quindi la ricorsione con l'annotazione:

```
for  $j \leftarrow 0$  to  $n$  do
   $M[j] \leftarrow empty$ 
end for
function  $AR - OPT(i)$ 
  if  $M[i] == empty$  then
    if  $i == 0$  then
       $M[i] \leftarrow 0$ 
    else
       $M[i] \leftarrow \max(v_i + AR - OPT(p[i]), AR - OPT(i - 1))$ 
    end if
  end if
  return  $M[i]$ 
end function
```

che resta una tecnica *top-down*. Ma solitamente si lavora *bottom-up* nella programmazione dinamica. Quindi non abbiamo di caricare l'array e non abbiamo bisogno di if/else:

```
function  $PD - OPT(i)$ 
   $M[0] \leftarrow 0$ 
  for  $j \leftarrow 1$  to  $i$  do
     $M[j] \leftarrow \max(v_j + M[P[j]], M[j - 1])$ 
  end for
  return  $M[i]$ 
end function
```

ovviamente prima di tutto ho il caricamento di  $p[i]$ , come è stato descritto sopra.

Ora si ha una complessità pari a  $\Theta(n)$  per quest'ultimo algoritmo ma bisogna sommare  $\Theta(n \log n)$ , in quanto bisogna ordinare le attività sul tempo di fine.

### 3.0.1 Programmazione Dinamica

Un problema di decisione prevede unicamente 2 tipi di risultato: vero e falso. Si ha una distinzione dei problemi:

- **problemi intrattabili**, che potrebbero avere una risposta calcolabile in un tempo idnefinito o addirittura che non possono essere dimostrati
- **problemi di ricerca**, che si occupano di trovare una soluzione positiva ad una certa istanza (per esempio dei problemi che trattano i percorsi)
- **problemi di ottimo**, dove si cerca una e una sola soluzione che massimizza o minimizza una certa funzione costo

Si parla di **algoritmi euristici** quando si ha un algoritmo che ci da una soluzione che magari non è la migliore. A questi si aggiungono **algoritmi di approssimazione** che si occupano di cercare l'ordine di una soluzione rispetto a quella "migliore".

### 3.0.2 Un Problema di Sequenze

Prendiamo una stringa  $X = \langle x_1, \dots, x_n \rangle$ . Una sottosequenza di  $X$  è un insieme di indici con  $i_1 \dots i_k$  con  $k \leq n$  e indici strettamente crescenti ma non



necessariamente consecutivi. Quindi data una sequenza  $X = \langle x_1, \dots, x_n \rangle$  e una sottosequenza  $Z = \langle z_1, \dots, z_n \rangle$  diciamo che:

$$\exists i_1, \dots, i_k \rightarrow x_{i_1} < x_{i_2} < \dots < x_{i_k}, \quad i_i > i_{i-1}$$

Si assuma che gli indici partano da 1.

Quindi, per esempio, per la stringa  $X = \langle A, B, C, B, D, A, B \rangle$  si possono avere le sottosequenze  $A_1 = \langle B, C, D, B \rangle$  (con indici 2, 3, 5, 7),  $A_2 = \langle A, B, A, B \rangle$  (con indici 1, 2, 6, 7 oppure 1, 4, 6, 7) etc. . . .

### 3.0.3 Longest Common Substring

Si definisce una **sottosequenza comune**  $Z$  a due sequenze  $X$  e  $Y$  se  $Z$  è sottosequenza sia di  $X$  che di  $Y$  (non è necessario che gli indici siano nello stesso ordine).

Cerchiamo ora un algoritmo che trovi la più grande sottostringa comune, appunto **long common substring (LCS)**, con però elementi ordinati in grandezza, quindi *longest increasing subsequence (LIS)*.

Con una soluzione iterativa avremmo  $O(2^n)$  quindi pensiamo ad una soluzione con la programmazione dinamica.

Cerchiamo quindi un problema associato, cercando la sottosequenza di  $X$  più lunga che termina in una certa posizione  $i$ . In questo studio delle sequenze gli indici aumentano solo se il valore che indicizzano è superiore a quello precedente, altrimenti diminuiscono di una unità (a meno che non sia l'indice 1 che resta uguale), quindi, per esempio, la stringa  $X = \langle A, B, C, B, D, A, B \rangle$  avrà indici 1, 2, 3, 2, 4, 3, 4.

Definiamo  $L[i]$  la lunghezza massima della LIS che termina col carattere in posizione  $i$ .

Procedo salvando la lunghezza della sottosequenza più lunga fino a  $i$ .

Si definisce  $X_i$  la restrizione della sequenza considerando solo i primi  $i$  caratteri. Chiamiamo  $Z_i$  la più lunga sottosequenza di  $X$  che termina con  $X_i$ .

Salviamo le varie  $Z$  in un array  $L[1..N]$  con  $L[i]$  che è la lunghezza massima della sottosequenza di  $X$  che termina con  $X_i$ .

Si ha il caso base:

$$L[1] = 1$$

e il caso generale:

$$L[i] = 1 + \max_{1 \leq j \leq i-1} \{L[j] \mid X_j < X_i\}, \quad 1 < i \leq N$$

ricordando che  $\max \emptyset = 0$ .

La soluzione sarà quindi:

$$\max_{1 \leq i \leq N} \{L[i]\}$$

Scriviamo quindi l'algoritmo:

```
function LIS( $X[1..N]$ )  
   $L[1..N]$   
   $X[0] = -1$   
   $L[1] = 1$   
   $L[0] = 0$   
  for  $i \leftarrow 2$  to  $N$  do  
     $R \leftarrow \text{maxAcc}(l[], i, i - 1, X[i])$   
     $L[i] = R + 1$   
  end for  
   $RT \leftarrow \max(L[i], 1, N)$   
  return  $RT$ 
```

Con la funzione *maxAcc* che calcola il massimo degli accettabili. Questo algoritmo è  $O(n^2)$ .

calcolo  $L[i]$  che è la sequenza più lunga che termina col carattere i:

$$L[i] = \begin{cases} 0 & i = 0 \\ 1 & i = 1 \\ 1 + \max(L[j] \mid c[j] < c[i], j = 1 \dots i - 1) & i > 1 \end{cases}$$

Scriviamo il vero algoritmo:

```
function LIS( $S[1..n]$ )  
   $L[1] \leftarrow 1$   
   $\text{maxTot} \leftarrow 1$   
  for  $i \leftarrow 2$  to  $n$  do  
     $\text{max} \leftarrow 0$   
    for  $j \leftarrow 1$  to  $i - 1$  do  
      if  $S[j] < S[i]$  AND  $L[i] > \text{max}$  then  
         $\text{max} \leftarrow j$ 
```

```

    end if
     $l[i] \leftarrow L[max] + 1$ 
     $P[i] \leftarrow max$ 
    if  $L[i] > maxTot$  then
         $maxTot \leftarrow i$ 
    end if
end for
end for
return  $maxTot$ 
end function

```

Con  $maxTot$  salva il massimo dell'array delle lunghezze. Che ha tempo di esecuzione pari a  $T(n) = 3c + 5cn + 3c \sum_1^n i = O(n^2)$ . Questa funzione verrà chiamata in una ciclo *for*. Per rendere il tutto più efficiente quindi memorizzo, anzichè memorizzare tutti i valori degli array che si vengono a creare contenenti le sequenze buone, l'indice del  $maxTot$  precedente, nella lista  $P$ , avendo una complessità di salvataggio sulla memoria pari a  $O(n)$ .

### 3.0.4 Il Problema delle Scatole

Ho una tripla  $B_i = (a_i, b_i, c_i)$  rappresentanti lunghezza, larghezza e altezza di una scatola.  $B_i$  è quindi un insieme di scatole che non possono essere ruotate. Voglio sapere a lunghezza più lunga di scatole che possono essere contenute una dentro l'altra. Cerco quindi il massimo valore di  $k$  tale per cui, per una sequenza  $B_1, \dots, B_i$ :

$$\exists x \rightarrow B_1, \dots, B_i \mid B_{i1} \subset B_{i2} \subset \dots \subset B_{ik}, \quad i_1 < \dots < i_k$$

Si introduce un vettore  $z$  con  $n$  elementi tale che  $z[i]$  sia la lunghezza massima di una sottosequenza crescente di elementi  $B_1, \dots, B_i$ . Quindi:

$$z[i] = \begin{cases} 1 + \max\{z[j] \mid 1 \leq j < i, a_j < a_i, b_j < b_i, c_j < c_i\} & i > 1 \\ 1 & i = 1 \end{cases}$$

inoltre  $\max\{\emptyset\} = 0$ . Abbiamo quindi il seguente algoritmo:

```
function MAXBOX(B[1..n])
  z[1] ← 1
  for i ← 2 to n do
    max ← 0
    for j ← 1 to i - 1 do
      if aj < ai AND bj < bi AND cj < ci AND z[j] < max then
        max ← z[j]
      end if
    end for
    z[i] ← max
  end for
  return z
end function
```

Si ha complessità pari a  $O(n^2)$  mentre lo spazio richiesto è quello per memorizzare il vettore, ovvero  $\Theta(n)$ .

Può anche essere scritto ricorsivamente ma risulta troppo esoso in termini di spazio.

## 3.1 Longest Common Subsequence

Ritorniamo al problema di inizio capitolo. Abbiamo due stringhe,  $X = \langle x_1, \dots, x_n \rangle$  e  $Y = \langle y_1, \dots, y_n \rangle$  con  $Z$  sottosequenza comune a  $X$  e  $Y$ .