# Appunti di
# Analisi e Progettazione di Algoritmi


**A cura di:**
Francesco Refolli
Matricola 865955


**Anno Accademico 2022-2023**

# Chapter 1

# Note sul Corso

**todo: segnare delle note**

# Chapter 2

# Problema LCS

## 2.1 Introduzione

### Definizioni

**Alfabeto** Un alfabeto $\Sigma$ e' un insieme finito e non vuoto di simboli.
Si usano le prime lettere dell'alfabeto minuscolo per identificare i simboli generici.

**Stringa** Una stringa (o parola) $w$ e' una giustapposizione (o concatenazione) di simboli dell'Alfabeto $\Sigma$.
Si usano le ultime lettere dell'alfabeto minuscolo per identificare il nome di una stringa. Esempio: una stringa $X = a \cdot b \cdot c \cdot d$. Sempre sara' indicata in futuro senza l'operatore per semplicita'.

**Sequenza** Una sequenza $S$ e' un concetto analogo alla stringa, ma e' generalmente concepita come una elencazione di simboli di un alfabeto. La principale differenza sta nel fatto che questa puo' avere elementi di altri alfabeti.
Si usano le ultime lettere dell'alfabeto maiuscolo per identificare il nome di una sequenza. Si indica con $X = < a, b, c, d >$
L'i-esimo elemento si indica con $x_i$. La lunghezza puo' essere indicata tramite $|X|$.

**Sottostringa** Una sottostringa $S$ di una stringa (o sequenza) e' una stringa che sia la concatenazione di elementi consecutivi della stringa o sequenza di partenza. Di solito prodotta tagliando un pezzo di lunghezza $m$ dal capo, un pezzo di lunghezza $n$ dalla coda.

**Sottosequenza** Una sottosequenza $S$ di una stringa (o sequenza) e' una sequenza di elementi della stringa o sequenza di partenza che mantenga l'ordine degli stessi. (Non per forza la consecutivita'!). In questo senso una sottostringa e' analoga ad una sottosequenza che mantenga la consecutivita'. Di solito prodotta tramite la cancellazione di $k$ elementi.

**Prefisso** Un prefisso di lunghezza $i$ e' una sottosequenza composta dai primi $i$ elementi consecutivi di una stringa o sequenza. Detta X una stringa, si indica con $X_i$.

**Suffisso** Un suffisso di indice $i$ e' una sottosequenza composta da tutti gli elementi consecutivi a partire dall'indice $i$.

**LCS**

**Introduzione** Il problema LCS (Longest Common Subsequence) consiste nel cercare in tempo ragionevole la sottosequenza piu' lunga comune a due sequenze o stringhe.

**Esempi**

1. $X = <$ S, C, O, I, A, T, T, O, L, O $>$
   $Y = <$ B, A, R, A, T, T, O, L, O $>$
   La LCS e' $Z = <$ A, T, T, O, L, O $>$ **nota: puo' essere usata sia la prima che la seconda 'A' di** $Y$.

2. $X = <$ M, A, G, I, C, O $>$
   $Y = <$ M, A, N, T, E, N, E, R, E $>$
   La LCS e' $Z = <$ M, A $>$

3. $X = <$ M, A, I, A, L, E $>$
   $Y = <$ N, A, T, A, L, E $>$
   La LCS e' $Z = <$ A, A, L, E $>$

## 2.2 Algoritmo Banale

**Ragionamento** Detti $X$, $Y$ due sequenze, $i$, $j$ i rispettivi indici e $n$, $m$ le rispettive lunghezze. Detti $Z = LCS(X, Y)$ e $k$ il suo indice. Si inizializzano gli indici alle lunghezze. (si parte dal fondo!)

Si puo' ragionevolmente pensare che:

1. A) Se $x_i = y_j$ allora $z_k = x_i$ e $Z_{k-1} = LCS(X_{i-1}, Y_{j-1})$

2. B) Se $x_i \neq y_j$ e $z_k = x_i$ allora $Z_k = LCS(X_i, Y_{j-1})$

3. C) Se $x_i \neq y_j$ e $z_k = y_j$ allora $Z_k = LCS(X_{i-1}, Y_j)$

4. D) Se $z_k \neq y_j$ e $z_k \neq x_i$ allora $Z_k = LCS(X_{i-1}, Y_{j-1})$

Si nota che il caso D e' direttamente traducibile in codice, ma viene comunque compreso dai casi B e C, di conseguenza non avrebbe senso ripeterlo. In particolare si osserva che $D \Rightarrow (B \cdot C \Leftrightarrow C \cdot B)$.

**Procedura** Possiamo quindi sviluppare uno pseudo-codice basato su questi teoremi. Non potendo sapere nei casi B) e C) quale sia tra i due il risultato, l'unico modo per verificarlo e' esplorare entrambi i casi. Gli indici delle sequenze appartengono all'insieme $[1, length]$.

**Algorithm** Algoritmo Banale per LCS

**procedure** LCS($X_i, Y_j$)
  **if** $X_i = <>$ or $Y_j = <>$ **then**
    **return** $<>$
  **else**
    **if** $x_i = y_j$ **then**
      **return** append(LCS($X_{i-1}$, $Y_{j-1}$), $x_i$)
    **else**
      B $\leftarrow$ LCS($X_i$, $Y_{j-1}$)
      C $\leftarrow$ LCS($X_{i-1}$, $Y_j$)
      **if** len(B) $\geq$ len(C) **then**
        **return** B
      **else**
        **return** C
      **end if**
    **end if**
  **end if**
**end procedure**

**Complessita'** Abbiamo appurato che questo algoritmo (TOP-DOWN) puo' portare a problemi quando l'input e' troppo grande, tuttavia e' possibile porvi rimedio

Uno dei problemi e' per esempio la ripetizione di lavoro gia' svolto in ricorsioni "parallele". Si potrebbe immaginare di memorizzare i risultati delle chiamate e usare quelli memorizzati all'occorrenza.
Resta pero' l'alto numero di chiamate: tendenzialmente $O(2^{n+m})$. Per questo si usa la programmazione dinamica: si risolve iterativamente un procedimento ricorsivo.

## 2.3 Usare la Programmazione Dinamica

**Ragionamento** L'Approccio Bottom-Up si basa sul partire dal risultato piu' piccolo, caso base, e costruire pr$>$essivamente i risultati delle chiamate "precedenti". Questo significa in pratica calcolare tutte le chiamate per tutte le combinazioni possibili d'input attiente al problema. Potrebbe sembrare un procedimento dispendioso ma in alcuni casi puo' essere quello piu' efficiente.

Supponendo di avere a disposizione due matrici $b$, $c$ di dimensione $(n+1)(m+1)$, queste possono essere usate per simulare l'algoritmo ricorsivo nei vari passi. Attenzione, gli indici della matrice partono con 0!

Caso base:

$$\begin{cases} c[0][j] = 0 & 0 \leq j \leq n \\ c[0][j] = 0 & 0 \leq i \leq m \end{cases}$$

Caso ricorsivo:

$$c[i][j] = \begin{cases} c[i][j] = c[i-1][j-1] + 1 & x_i = y_j \\ c[i][j] = c[i-1][j] & c[i-1][j] \geq c[i][j-1] \\ c[i][j] = c[i][j-1] & c[i-1][j] < c[i][j-1] \end{cases}$$

$$b[i][j] = \begin{cases} b[i][j] = " \nwarrow " & x_i = y_j \\ b[i][j] = " \uparrow " & c[i-1][j] \geq c[i][j-1] \\ b[i][j] = " \leftarrow " & c[i-1][j] < c[i][j-1] \end{cases}$$

Questo algoritmo riempie due matrici:

- $c$ contiene le lunghezze dei vari $LCS(X_i, Yj)$
- $b$ contiene il percorso da seguire per ottenere l'ottimale

Per ottenere $LCS(X, Y)$ non restera' da fare che seguire il percorso indicato da $b$ con l'accortezza di raccogliere in coda gli $x_i$ tali che $b[i][j] = Diagonal$.

**Procedura** Avremo bisogno di tre algoritmi:

- una funzione che inizializza la matrice generando il caso base
- una funzione che riempie la matrice con i casi ricorsivi
- una funzione che legge il percorso della matrice $b$
- una funzione che stampa la LCS

**Algorithm** Inizializza Matrice

```
procedure InizializzaMatrice(m, n)
    for i ← 0 to m do
        c[i][0] ← 0
    end for
    for j ← 0 to n do
        c[0][j] ← 0
    end for
end procedure
```

**Algorithm** Riempi Matrice

```
procedure RiempiMatrice(X, Y)
    for i ← 1 to m do
        for j ← 1 to n do
            if x_i = y_j then
                c[i][j] ← c[i − 1][j − 1] + 1
                b[i][j] ← "↖"
            else
                if c[i − 1][j] ≥ c[i][j − 1] then
                    c[i][j] ← c[i − 1][j]
                    b[i][j] ← "↑"
                else
                    c[i][j] ← c[i][j − 1]
                    b[i][j] ← "←"
                end if
            end if
        end for
    end for
end procedure
```

**Algorithm** Leggi LCS

```
procedure LeggiLCS(b, X, i, j)
    if i = 0 ∨ j = 0 then
        return <>
    end if
    if b[i][j] = "↖" then
        return append(LeggiLCS(b, X, i − 1, j − 1), x_i)
    else
        if b[i][j] = "↑" then
            return LeggiLCS(b, X, i − 1, j)
        else
            return LeggiLCS(b, X, i, j − 1)
        end if
    end if
end procedure
```

**Algorithm** Stampa LCS

```
procedure StampaLCS(b, X, i, j)
    if i = 0 ∨ j = 0 then
        return <>
    end if
    if b[i][j] = " ↖ " then
        StampaLCS(b, X, i − 1, j − 1)
        print(x_i)
    else
        if b[i][j] = " ↑ " then
            StampaLCS(b, X, i − 1, j)
        else
            StampaLCS(b, X, i, j − 1)
        end if
    end if
end procedure
```

**Complessita'**

- La complessita' temporale di InizializzaMatrice e' $O(n + m)$.
- La complessita' temporale di RiempiMatrice e' $O(n * m)$.
- La complessita' temporale di LeggiLCS e' $O(n + m)$.

**LeggiLCS** La complessita' di questo algoritmo (analogo e identico a StampaLCS), e' data in termini di upper-bound rispetto al caso peggiore, ovvero quando le due sequenze hanno solo il primo simbolo in comune. In quel caso il percorso seguito dall'algoritmo seguira' una traiettoria angolata, facendo quindi tutti i "piccoli passi" (l'unico tassello obliquo e' il target $b[1][1]$), per un totale di $n + m$ iterazioni.
LeggiLCS non fa parte del programma d'esame, e' riportato qua per completezza.

**Conclusione** Quindi la complessita' temporale totale e' $O(n * m)$, molto migliore dell'algoritmo "banale" presentato nel paragrafo precedente.
* E' pero' da tenere in considerazione che questo richiede spazio in memoria con una complessita' di $O(n * m)$, visto che deve mantenere due matrici $n * m$.
Certamente superiore al $O(n + m)$ dell'algoritmo banale, ma resta pur sempre un costo che vale la pena pagare per usufruire della strategia Bottom-Up.

# Chapter 3

# Applicazione di PD

## 3.1 Distanza di Edit

**Introduzione** E' definita come numero minimo di **cancellazioni**, **sostituzioni**, **inserimenti** che trasformano una stringa $X$ in una seconda stringa $Y$. E' un problema simmetrico. $D_{edit}(X,Y) = D_{edit}(Y,X)$.

**Esempio** $X =< 2,4,10,3,1 > Y =< 2,4,2,1 >$

- Cancellazione di 10.
- Sostituisco 3 con 2.

Quindi $D_{edit}(X,Y) = 2$.

- Inserimento di 10.
- Sostituisco 2 con 3.

Quindi $D_{edit}(Y,X) = 2$.

- Cancellazione di 3.
- Sostituisco 10 con 2.

Possono non esistere soluzioni uniche.

**Problemi**

**PR** Date due sequenze X, Y, trovare la distanza di edit di X in Y.

**P** Date due sequenze X, Y, trovare il minimo insieme di operazioni di cancellazione, inserimento, sostituzione che trasformano X in Y.

**Soluzione**

**Sottoproblema di PR** Trovare la distanza di edit dei prefissi $X_i$ e $Y_j$. Il numero di sottoproblemi e' $(m+1) \cdot (n+1)$. $d_{i,j} = distanzadieditdeiprefissiX_iY_j$. Soluzione 'e $d_{m,n}$

**Casi base di PR**

- $i = 0 \wedge j = 0 \Rightarrow d_{0,0} = 0$
- $i > 0 \wedge j = 0 \Rightarrow d_{i,0} = i$
- $i = 0 \wedge j > 0 \Rightarrow d_{0,j} = j$

**Caso ricorsivo di PR** Con $i > 0 \wedge j > 0$:

$$d_{i,j} = \begin{cases} d_{i-1,j-1} & x_i = y_j \\ min(d_{i-1,j-1}, d_{i,j-1}, d_{i-1,j}) + 1 & altrimenti \end{cases}$$

Il caso "altrimenti" si spiega in questo modo:

- $Sostituzione(x_i \to y_j) + d_{i-1,j-1}$
- $Inserimento(y_j) + d_{i,j-1}$
- $Cancellazione(x_i) + d_{i-1,j}$

**Complessita'** La complessita' dell'algoritmo ricorsivo e' $(3^(n+m))$.

```
procedure ED-RIC(X, Y)
    if j = 0 then
        return i
    else if i = 0 then
        return j
    else
        A ← ED − RIC(X_{i−1}, Y_j) + 1
        B ← ED − RIC(X_i, Y_{j−1}) + 1
        C ← ED − RIC(X_{i−1}, Y_{j−1}) + 1
        if A ≤ B ∧ A ≤ C then
            return A
        else if B ≤ A ∧ B ≤ C then
            return B
        else
            return C
        end if
    end if
end procedure
```

**Algoritmo Ricorsivo**

```
procedure ED-ITER-RM(m, n)
    for i = 0 to m do
        M[i][0] ← i
        B[i][0] ← < " ↑ ", Null >
    end for
    for j = 0 to n do
        M[0][j] ← j
        B[0][j] ← < " ← ", Null >
    end for
end procedure
```

```
procedure ED-ITER(X, Y)
    ED − ITER − RM(m, n)
    for i = 1 to m do
        for j = 0 to n do
            if x_i = y_j then
                M[i][j] ← M[i − 1][j − 1]
                B[i][j] ← < "", Null >
            else if M[i − 1][j] ≤ M[i][j − 1] ∧ M[i − 1][j] ≤ M[i − 1][j − 1] then
                M[i][j] ← M[i − 1][j] + 1
                B[i][j] ← < " ↑ ", Delete >
            else if M[i][j − 1] ≤ M[i − 1][j] ∧ M[i][j − 1] ≤ M[i − 1][j − 1] then
                M[i][j] ← M[i][j − 1] + 1
                B[i][j] ← < " ← ", Insert >
            else
                M[i][j] ← M[i − 1][j − 1] + 1
                B[i][j] ← < " ↖ ", Change >
            end if
        end for
    end for
    return M[m][n]
end procedure
```

**Alrgotimo Iterativo PD** Preparo una Matrice $B[m][n]$ che contiene la ricostruzione del percorso iterativo. Ogni cella $B[i][j] =< Direction, Operation >$

**Complessita' di** $ED - ITER(X, Y)$ L'algoritmo e' formato da due cicli innestati, quindi $T(n) = \Theta(n \cdot m)$

# Part I

# Laboratorio

# Chapter 4

# 11-10-2022

## 4.1 Scheduling di Attivita'

Un esempio di Programmazione per Intervalli Pesati.
Siano date $n$ attivita' da svolgersi nello stesso spazio fisico. Determinare un sottoinsieme di attivita' che non si sovrappongono e che sia il massimo possibile.

| i | p(i) | v |
|---|---|---|
| 1 | 0 | 10 |
| 2 | 1 | 2 |
| 3 | 2 | 8 |
| 4 | 2 | 1 |
| 5 | 1 | 1 |
| 6 | 4 | 3 |

Ad occhio si ricava $< 1, 3, 6 >$.

L'algoritmo naive e' quello combinatorio, ma e' estremamente inefficiente. Il tempo e' $T(n) = \Omega(2^n)$.

**Soluzione PD**

- $n \leftrightarrow X = 1, .., n$
- $\forall i \in 1, .., n,\ s_i$ e' il tempo di inizio dell'attivita' $i$
- $\forall i \in 1, .., n,\ f_i$ e' il tempo di fine dell'attivita' $i$
- $\forall i \in 1, .., n,\ v_i$ e' il valore dell'attivita' i

Definisco la funzione:
$COMP : \P(1, .., n) \rightarrow true, false$
$\forall i, j \in A \mid i \neq j \vee attivitaCompatibili(A_i, A_j) \Rightarrow COMP(A) = true$

Dette poi $i, j$ due attivita' si dice:

$$attivitaCompatibili(i, j) = \begin{cases} true & [s_i, f_i) \cap [s_j, f_j) = \emptyset \\ false & altrimenti \end{cases}$$

Quindi si definisce:
$V : \mathbb{P}(1,..,n) \to \mathbb{R}$

$$V(i,j) = \begin{cases} \Sigma_{i \in A} v_i & A \neq \emptyset \\ 0 & A = \emptyset \end{cases}$$

La Soluzione e' $S \subseteq X \mid COMP(S) = true \wedge \forall A \subset XV(S) \leq V(A)$

**Processo** Detto $S_n \Leftrightarrow sol(X_n)$, e quindi $S_{n-k} \Leftrightarrow sol(X_{n-k})$. Nella soluzione di $S_n$ si assume di conoscere:

- $\forall k \in 1,..,nS_{n-k}$
- $\forall k \in 1,..,nsol(X_{n-k})$

Detto $OPT(i) = V(S_i)$. Dividendo in sottoproblemi e potendo disponere a piacimento di ognuno di questi, riesco facilmente a individuare il caso base. E' immediato sapere $sol(\emptyset)$ e $sol(1)$, quindi questi possono essere i casi base.

**Ragionamento**

**Caso Base** $S_0 \Leftrightarrow X_0 = \emptyset \wedge V(X_0) = 0$
$S_1 \Leftrightarrow X_1 = 1 \wedge V(X_1) = 1$

**Caso Ricorsivo** Voglio risolvere $S_i$ e $OPT(i)$, assumo di avere gia' risolto $\forall j \in X \mid j < iS_j$.

Se sapessi che $i \in S_i$ il sotto problema da considerare sarebbe $sol(\forall j \in X_{i-1} \mid attivitaCompatibile(i,j$
Questo si traduce nel risolvere $S_j$, dove $j$ e' il massimo indice di una attivita' compatibile con $i$. Se sapessi al contrario che $i \in S_i$ allora dovrei risolvere $S_{i-1}$.

Detto $p(i) : \mathbb{R} \to \mathbb{R}$, la funzione che associa ad ogni attivita' l'attivita' compatibile precedente piu' vicina. Quindi mi riduco a:

$$S_i = \begin{cases} 0 & i = 0 \\ 1 & i = 1 \\ S_{p(i)} \cup i & V(p(i)) + v_i \geq V(i-1) \\ S_{i-1} & V(p(i)) + v_i < V(i-1) \end{cases}$$

```
procedure WIS-OPT(i)
    if i = 0 then
        return 0
    else if i = 1 then
        return 1
    else
        Z1 ← append(WIS − OPT(p(i)), x_i)
        Z2 ← WIS − OPT(i − 1)
        if OPT(Z1) ≥ Z2 then
            return Z1
        else
            return Z2
        end if
    end if
end procedure
```

**Procedura TOP-DOWN**

---

**procedure** INIZIALIZZA-VETTORI
    $OPT[0] \leftarrow 0$
    $OPT[1] \leftarrow 1$
    $WIS[0] \leftarrow X_0$
    $WIS[1] \leftarrow X_1$
**end procedure**

---

**procedure** WIS-OPT-ITER($i$)
    $INIZIALIZZA - VETTORI()$
    **for** $i = 2$ to $n$ **do**
        $Z1 \leftarrow append(WIS[p(i)], x_i)$
        $Z2 \leftarrow WIS[i-1]$
        **if** $OPT[p(i)] < OPT[i-1]$ **then**
            $OPT[i] \leftarrow OPT[p(i)] + v_i$
            $WIS[i] \leftarrow Z1$
        **else**
            $OPT[i] \leftarrow OPT[i-1]$
            $WIS[i] \leftarrow Z2$
        **end if**
    **end for**
    **return** WIS[n]
**end procedure**

---

**Procedura BOTTOM-UP** Il vettore $WIS$ e' implementabile ragionevolmente con una matrice di booleani che caratterizzano la presenza di un elemento nell'insieme. Visto che questo richiederebbe una quantita' di spazio non indifferente, una cosa comoda potrebbe essere codificare le righe o le colonne in un numero intero decimale.

**Complessita'** La procedura $WIS - OPT - ITER$ comprende un solo ciclo di $\Theta(n)$ iterazioni. Il calcolo di $p(i)$ richiede $O(n)$, perche' e' un ciclo inverso semplice che dipende dalla disposizione degli elementi in $X$. Quindi l'algoritmo e' $T(n) = (n^2)$ nel caso medio.

**Osservazioni** E' possibile scrivere una procedura che esplori linearmente l'array $OPT$ per verificare i passi che sono stati effettuati per costruire OPT. Quindi si puo' risparmiare lo spazio occupato da $WIS$.