

# 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  $X = \langle a, b, c, d \rangle$

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.  $X = \langle S, C, O, I, A, T, T, O, L, O \rangle$   
 $Y = \langle B, A, R, A, T, T, O, L, O \rangle$   
 La LCS e'  $Z = \langle A, T, T, O, L, O \rangle$  **nota:** puo' essere usata sia la prima che la seconda 'A' di  $Y$ .
2.  $X = \langle M, A, G, I, C, O \rangle$   
 $Y = \langle M, A, N, T, E, N, E, R, E \rangle$   
 La LCS e'  $Z = \langle M, A \rangle$
3.  $X = \langle M, A, I, A, L, E \rangle$   
 $Y = \langle N, A, T, A, L, E \rangle$   
 La LCS e'  $Z = \langle A, A, L, E \rangle$

## 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  $X_i = \langle \rangle$  or  $Y_j = \langle \rangle$  then
    return  $\langle \rangle$ 
  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 progressivamente i risultati delle chiamate "precedenti". Questo significa in pratica calcolare tutte le chiamate per tutte le combinazioni possibili d'input attinte 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:

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

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  $LCS(X_i, Y_j)$
- $b$  contiene il percorso da seguire per ottenere l'ottimale

Per ottenere  $LCS(X, Y)$  non resterà 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  $i \leftarrow 0$  to  $m$  do
     $c[i][0] \leftarrow 0$ 
  end for
  for  $j \leftarrow 0$  to  $n$  do
     $c[0][j] \leftarrow 0$ 
  end for
end procedure

```

---



---

**Algorithm** Riempi Matrice

---

```

procedure RIEMPIMATRICE( $X, Y$ )
  for  $i \leftarrow 1$  to  $m$  do
    for  $j \leftarrow 1$  to  $n$  do
      if  $x_i = y_j$  then
         $c[i][j] \leftarrow c[i-1][j-1] + 1$ 
         $b[i][j] \leftarrow "$   $\nwarrow$   $"$ 
      else
        if  $c[i-1][j] \geq c[i][j-1]$  then
           $c[i][j] \leftarrow c[i-1][j]$ 
           $b[i][j] \leftarrow "$   $\uparrow$   $"$ 
        else
           $c[i][j] \leftarrow c[i][j-1]$ 
           $b[i][j] \leftarrow "$   $\leftarrow$   $"$ 
        end if
      end if
    end for
  end for
end procedure

```

---



---

**Algorithm** Leggi LCS

---

```

procedure LEGGILCS( $b, X, i, j$ )
  if  $i = 0 \vee j = 0$  then
    return  $\langle \rangle$ 
  end if
  if  $b[i][j] = "$   $\nwarrow$   $"$  then
    return append(LeggiLCS( $b, X, i-1, j-1$ ),  $x_i$ )
  else
    if  $b[i][j] = "$   $\uparrow$   $"$  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  $i = 0 \vee j = 0$  then
    return  $\langle \rangle$ 
  end if
  if  $b[i][j] = "\nwarrow"$  then
    StampaLCS( $b, X, i - 1, j - 1$ )
    print( $x_i$ )
  else
    if  $b[i][j] = "\uparrow"$  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.