UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laboratorio di Algoritmi e Strutture Dati

Autore:
Francesca Sani

N° Matricola:
7025023

Corso principale:
Algoritmi e Strutture Dati

Docente corso:
Simone Marinai

Indice

1 Introduzione

Questa relazione ha l'obiettivo di confrontare vari modi per calcolare la **Longest Common Subsequence** (*LCS*) tra due stringhe, in modo da comprendere i vantaggi e gli svantaggi delle diverse implementazioni.

Verranno analizzati gli algoritmi di **"forza-bruta"**, la versione **ricorsiva**, la versione ricorsiva con **memoization** e infine la versione **bottom-up**.

1.1 Ambiente di test

Il codice per eseguire gli esperimenti è stato realizzato in Python (interprete versione 3.9). L'ambiente di sviluppo è l'IDE PyCharm 2023.1.2.

Il computer su cui è stato scritto il codice ha le seguenti specifiche: MacBook Air Chip Apple M2 con CPU 8-core, 16GB di memoria unificata e sistema operativo macOS Ventura 13.6.

La relazione è stata scritta in LaTeX tramite l'utilizzo dell'editor online *Overleaf*.

1.2 Librerie utilizzate

Il programma Python scritto utilizza le seguenti librerie:

- **numpy**: permette di lavorare su tempi calcolati e metterli nella tabella generata;
- **timeit**: permette di misurare i tempi di esecuzione dei vari algoritmi;
- **matplotlib**: permette di generare grafici e tabelle, in modo da poter analizzare al meglio le prestazioni degli algoritmi;
- **random** e **string**: permettono di generare sequenze di stringhe casuali;
- **intertools** e **combinations**, permette di ottenere tutte le sottosequenze della sequenza passata.

2 Cenni teorici

2.1 Problema della Longest Common Subsequence

Una sottosequenza di una data sequenza è la sequenza stessa alla quale tolgo zero o più elementi. Formalmente, data una sequenza $X =< x_1, x_2, ..., x_m >$, un'altra sequenza $Z =< z_1, z_2, ..., z_k >$ è una **sottosequenza** di X se $\exists$ una sequenza di indici $I =< i_1, i_2, ..., i_k >$ tale che $\forall j = 1, 2, ..., k$ si ha $x_{ij} = z_j$.

Date due sequenze X e Y, diciamo che una sequenza Z è una **sottosequenza comune** di X e Y se Z è una sottosequenze di X e Y.

Nel problema della **LCS**, date due sequenze $X =< x_1, x_2, ..., x_m >$ e $Y =< y_1, y_2, ..., y_n >$, si desidera trovare una sottosequenza comune di lunghezza massima comune a X e Y.

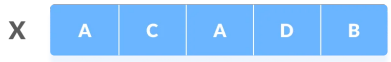

Figura 1: Sequenza X data in ingresso

Figura 2: Sequenza Y data in ingresso

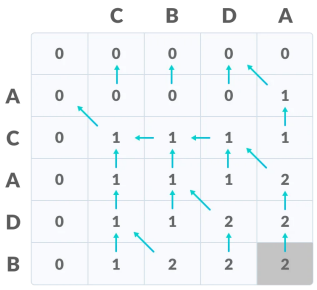

Figura 3: Tabella generata per calcolare la LCS

Figura 4: Sottosequenza comune delle sequenze X e Y

2.2 Algoritmo "forza-bruta"

In un approccio di **forza-bruta** per risolvere il problema LCS, enumeriamo tutte le sottosequenze di X e controlliamo ogni sottosequenza per vedere se è anche una sottosequenza di Y, tenendo traccia della sottosequenza più lunga che troviamo.
Dato che X ha 2^m sottosequenze, questo approccio richiede un *tempo esponenziale*, rendendolo impraticabile per sequenze lunghe.

2.2.1 Teorema - Sottostruttura ottimale di un LCS

Date le sequenze $X =< x_1, x_2, ..., x_m >$ e $Y =< y_1, y_2, ..., y_n >$.
Sia $Z =< z_1, z_2, ..., z_k >$ una qualsiasi LCS di X e Y.

- Se $x_m = y_n$, allora $z_k = x_m = y_n$ e Z_{k-1} è una LCS di X_{m-1} e Y_{n-1}.
- Se $x_m \neq y_n$, allora $z_k \neq x_m$ implica che Z è una LCS di X_{m-1} e Y.
- Se $x_m \neq y_n$, allora $z_k \neq y_n$ implica che Z è una LCS di X e Y_{n-1}.

2.3 Versione ricorsiva

Il teorema implica che ci siano uno o due sottoproblemi da esaminare per trovare una LCS di X e Y.
Se $x_m = y_n$, abbiamo trovato *una* LCS di X_{m-1} e Y_{n-1}.
Se $x_m \neq y_n$, allora dobbiamo risolvere *due* sottoproblemi: trovare una LCS di X_{m-1} e Y, e trovare una LCS di X e Y_{n-1}. La più lunga di queste due LCS è una LCS di X e Y.

La soluzione **ricorsiva** richiede la definizione di una ricorrenza per il valore di una soluzione ottima.
Definiamo $c[i, j]$ come la lunghezza di una LCS delle sequenze X_i e Y_j. Se $i = 0$ o $j = 0$, una delle sequenze ha lunghezzza 0. La sottostruttura ottima del problema della LCS consente di scrivere la seguente formula ricorsiva:

$$c[i,j] = \begin{cases} 0 & se\ i = 0\ o\ j = 0 \\ c[i-1, j-1] + 1 & se\ i, j > 0\ x_i = y_j \\ max(c[i, j-1], c[i-1, j]) & se\ i, j > 0\ x_i \neq y_j \end{cases}$$

Anche questa implementazione, nonostante sia semplice, è inefficiente a causa della sovrapposizione nei sottoproblemi, portando a una complessità esponenziale.

2.4 Versione ricorsiva con memoization

Per evitare la sovrapposizione dei sottoproblemi, attraverso la **memoization**, è possibile memorizzare i risultati ottenuti in una tabella, in modo da non doverli calcolare più volte.

Questo permette di migliorare notevolmente le prestazioni dell'algoritmo (rispetto alla versione di forza-bruta e alla versione ricorsiva), ma a causa della presenza della ricorsione può non essere molto efficiente quando si ha input di grandi dimensioni.

2.5 Versione bottom-up

Un'altra soluzione, dato che ci sono $\Theta(mn)$ sottoproblemi distinti, è quella di utilizzare la *programmazione dinamica* per calcolare le soluzioni con un metodo **bottom-up**.

Date due sequenze in input, memorizza i valori $c[i][j]$ in una tabella $c[0...m][0...n]$, le cui posizioni sono calcolate secondo l'ordine delle righe. Il tempo di esecuzione è $O(mn)$, perché il calcolo di ogni posizione della tabella richiede un tempo $\Theta(1)$.

2.6 Prestazioni attese

Questa relazione, come detto in precedenza, ha l'obbiettivo di confrontare le prestazioni di quattro diverse implementazioni di "calcolo" per trovare la più lunga sottosequenza comune tra due stringhe.
Quello che ci si aspetta dai test eseguiti, e commentati nelle sezioni successive, è che il metodo migliore per ricavare la LCS sia l'approccio *bottom-up*.

Le varie complessità temporali si possono distinguere come segue:

- **Approccio di forza-bruta**, poiché si esaminano tutte le sottosequenze di X e si verifica se sono sottosequenze di Y, ci aspettiamo una complessità esponenziale $O(2^m * n)$;
- **Approccio ricorsivo**, poiché vengono considerate tutte le possibili sottosequenze di X e Y, ci aspettiamo una complessità esponenziale $O(2^{m+n})$;
- **Approccio ricorsivo con memoization**, grazie alla memorizzazione dei sottoproblemi già analizzati permette di ridurre notevolmente la complessità, che non sarà più esponenziale $O(mn)$;
- **Approccio bottom-up**, le lunghezze delle sottosequenze analizzate vengono memorizzate in una tabella, avendo così una complessità $O(mn)$.

Algoritmo	**Complessità**
Forza-Bruta	$O(2^m * n)$
Ricorsivo	$O(2^{m+n})$
Ricorsivo con Memoization	$O(mn)$
Bottom-Up	$O(mn)$

Tabella 1: Complessità temporale per LCS

3 Documentazione del codice

Per svolgere i vari test è stato scritto del codice in Python, ovvero sono stati definiti vari metodi che permettono di calcolare la LCS con i diversi algoritmi in esame, generare grafici e tabelle, etc.

3.1 Descrizione dei metodi implementati

In questa sezione vengono spiegate le funzionalità di ogni metodo che è possibile trovare nel file **main.py**.

- $lcs\_bruteForce(s1,\ s2)$: metodo per l'algoritmo di forza-bruta, in cui vengono create tutte le sottosequenze della stringa s1 e per ciascuna si effettua un confronto per carattere con l'altra stringa s2, questo metodo ritorna la lunghezza della LCS trovata;
- $allSubsequences(s)$: metodo che viene chiamato da $lcs\_bruteForce(s1,\ s2)$ per generare tutte le sottosequenze possibili della stringa s;
- $lcs\_recursive(s1,\ s2)$ e $recursive(s1,\ s2,\ m,\ n)$: metodi per l'algoritmo ricorsivo, $lcs\_recursive(s1,\ s2)$ date due stringhe permette di entrare nella vera e propria ricorsione, invece $recursive(s1,\ s2,\ m,\ n)$ date le due stringhe e le loro dimensioni calcola la LCS secondo la formula ricorsiva definita precedentemente;
- $lcs\_memoization(s1,\ s2)$ e $memoization(s1,\ s2,\ c,\ m,\ n)$: metodi per l'algoritmo ricorsivo con memoization, $lcs\_memoization(s1,\ s2)$ date due stringhe permette di entrare nella vera e propria ricorsione e permette di generare la matrice di memorizzazione, invece $memoization(s1,\ s2,\ c,\ m,\ n)$ date le due stringhe, la matrice di memorizzazioni e le loro dimensioni calcola la LCS secondo la formula ricorsiva e verificando se un certo risultato è già stato calcolato;
- $lcs\_bottomUp(s1,\ s2)$: metodo per l'algoritmo bottom-up, in cui viene calcolata la LCS partendo dai primi caratteri delle stringhe e memorizzando i risultati in una matrice;
- $stringGenerator(length)$: metodo per generare delle sequenze di stringhe casuali;
- $measureTime(function, s1, s2)$: metodo per calcolare i tempi medi della funzione data;
- $drawPlots()$: metodo per genera grafici;
- $drawTable(columns, headers, title)$: metodo per generare tabelle;
- $test()$: metodo chiamato dal $main()$ per eseguire i test, il quale a sua volta chiama quasi tutti i metodi definiti in precedenza.

4 Descrizione dei test

Per eseguire i test sono state generate sequenze di stringhe casuali da 1 a 13 (attraverso la funzione *stringGenerator(length)*), pertanto sono stati eseguiti 12 test e per rendere più attendibili i risultati ottenuti, ciascun test è stato eseguito 100 volte e poi è stata calcolata una media dei tempi di esecuzione.

I tempi di esecuzione sono stati calcolati attraverso l'uso del metodo fornito dal modulo di Python *timeit*: ***timeit(stmt, number)***. Questo metodo, chiamato nella funzione *measureTime(function, s1, s2)* restituisce il tempo totale ottenuto per l'esecuzione dei test.

4.1 Grafici dei tempi di esecuzione

Di seguito sono riportati i grafici che, al variare della lunghezza delle stringhe confrontate, mostrano l'andamento dei tempi di esecuzione per ciascun algoritmo in esame. Inoltre, è riportato un grafico che racchiude i 4 grafici singoli, in modo da poter avere un confronto più accurato.

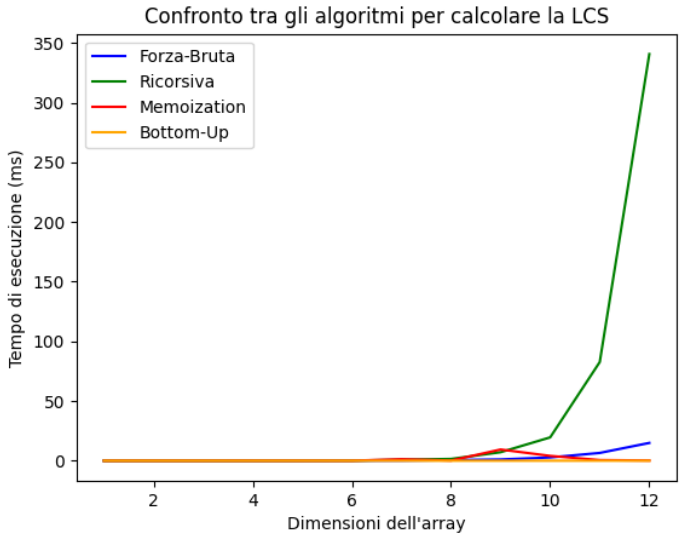

Figura 5: Tempi di esecuzione per calcolare la LCS

4.1.1 Algoritmo forza-bruta

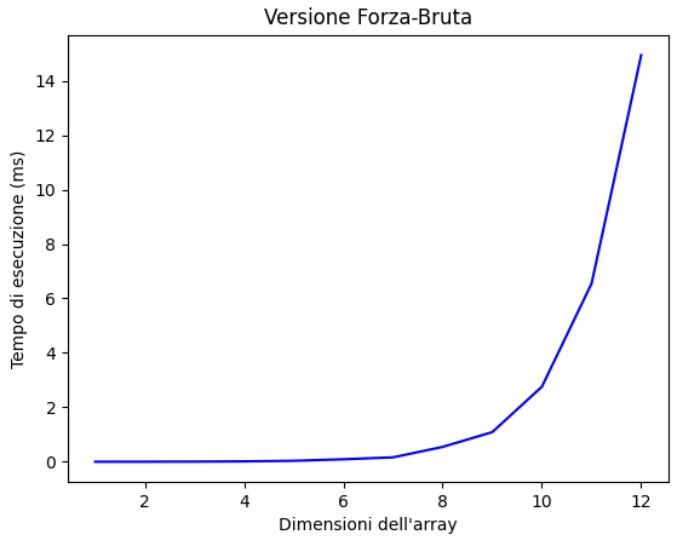

Figura 6: Tempi di esecuzione dell'algoritmo Forza-Bruta

4.1.2 Algoritmo ricorsivo

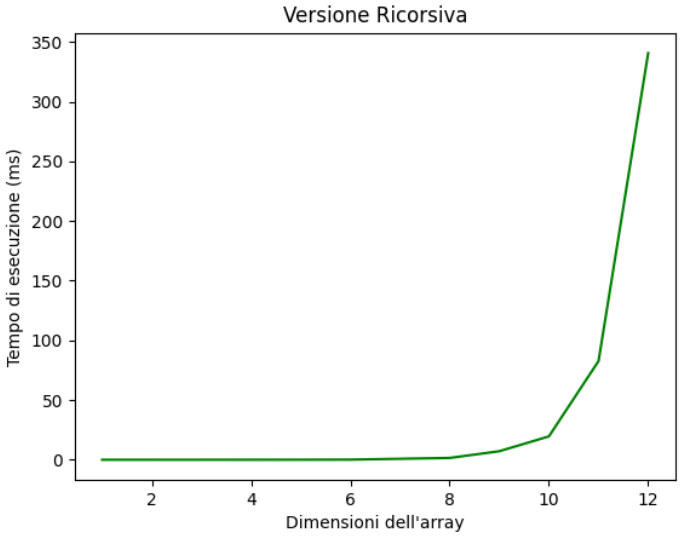

Figura 7: Tempi di esecuzione dell'algoritmo Ricorsivo

4.1.3 Algoritmo ricorsivo con memoization

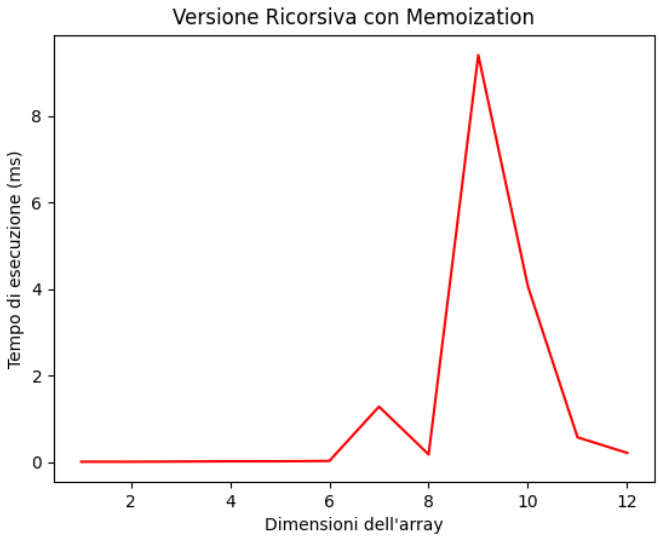

Figura 8: Tempi di esecuzione dell'algoritmo ricorsivo con Memoization

4.1.4 Algoritmo bottom-up

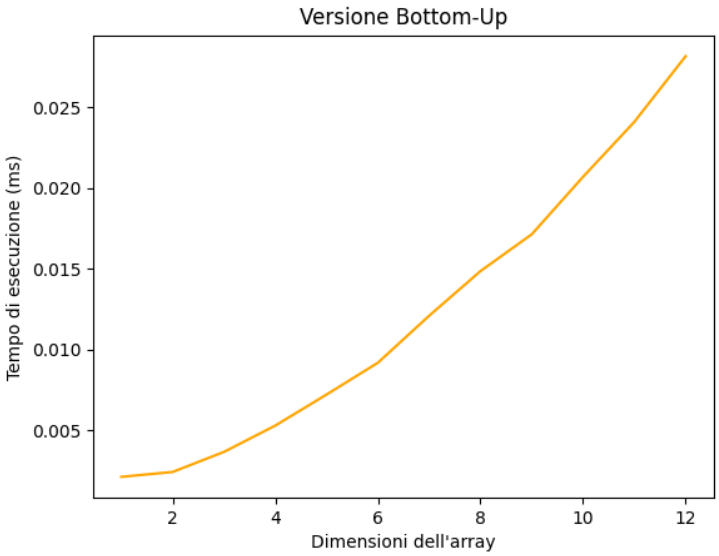

Figura 9: Tempi di esecuzione dell'algoritmo Bottom-Up

4.2 Tabelle dei tempi di esecuzione

Di seguito è riportata la tabella contenete i tempi di esecuzione dei quattro algoritmi analizzati, grazie ai quali è stato possibile generare i grafici della sezione precedente.

Dimensioni sequenza	Forza-Bruta	Ricorsivo	Memoization	Bottom-Up
1	1.118e-03	5.075e-04	3.462e-03	2.128e-03
2	3.120e-03	1.612e-03	3.185e-03	2.430e-03
3	6.172e-03	4.427e-03	7.856e-03	3.670e-03
4	1.527e-02	1.311e-02	1.374e-02	5.296e-03
5	3.645e-02	3.767e-02	1.415e-02	7.218e-03
6	9.287e-02	1.053e-01	2.183e-02	9.183e-03
7	1.624e-01	8.193e-01	1.279e+00	1.209e-02
8	5.472e-01	1.514e+00	1.725e-01	1.485e-02
9	1.089e+00	7.141e+00	9.415e+00	1.713e-02
10	2.760e+00	1.957e+01	4.062e+00	2.068e-02
11	6.547e+00	8.269e+01	5.677e-01	2.408e-02
12	1.494e+01	3.407e+02	2.083e-01	2.815e-02

Figura 10: Tabella dei tempi di esecuzione per calcolare la LCS

5 Considerazioni finali

In conclusione, osservando i risultati ottenuti dai vari test ed analizzando i grafici e la tabella, è stato possibile raggiungere l'obiettivo di questa relazione: confrontare quattro diverse implementazioni per calcolare la più lunga sottosequenza comune (LCS) tra due stringhe.

Dal grafico in *figura 5* è possibile trarre subito una prima conclusione, ovvero che il metodo migliore per calcolare la LCS è quello bottom-up, invece l'approccio ricorsivo è quello peggiore.

Il grafico in *figura 6* e il grafico in *figura 7*, i quali rappresentano rispettivamente l'andamento dei tempi di esecuzione per l'algoritmo di forza-bruta e per quello ricorsivo, confermano che la loro complessità temporale è esponenziale.
È interessante notare che nonostante siano entrambi esponenziali uno è migliore dell'altro. Infatti, anche se il metodo di forza-bruta a primo impatto possa sembrare peggiore, poiché esamina tutte le sottosequenze di X e controlla se esse sono presenti in Y, non è stato così.

Analogamente, il grafico in *figura 8* e il grafico in *figura 9*, i quali rappresentano rispettivamente l'andamento dei tempi di esecuzione per l'algoritmo ricorsivo con memoization e per quello bottom-up, confermano che la loro complessità temporale è quadratico.

Andando adesso ad analizzare la tabella in *figura 10* è possibile confrontare direttamente i tempi di esecuzione dei quattro approcci in esame. Nonostante le dimensioni delle sequenze generate aumentino, i tempi di esecuzione di bottom-up sono quasi sempre i più piccoli. Infatti per piccole dimensioni delle sequenze (1/2 caratteri), gli approcci migliori sono quelli ricorsivo e di forza-bruta.
È inoltre importante osservare che l'algoritmo ricorsivo con memoization, rispetto a quello ricorsivo e quello di forza-bruta, mostra notevoli vantaggi, nonostane il picco temporale che si verifica con dimensioni maggiori delle sequenze (circa 8/9 caratteri), osservabile anche nei grafici in *figura 5 e 8*.

In sintesi, è stato possibile affermare che grazie l'analisi condotta, ciò che era stato descritto nella sezione teorica, è confermato.

6 Bibliografia

Figura 1: Programmiz - LCS

Figura 2: Programmiz - LCS

Figura 3: Programmiz - LCS

Figura 4: Programmiz - LCS