Longest Common Subsequence(LCS)

Shehabul Islam Sawraz

Department of Computer Science & Engineering
Bangladesh University of Engineering and Technology

August 30, 2022

Problem Statement

1. **Input:** Two strings S and T.
2. **Output:** Longest common subsequence of S and T.

Problem Statement

1. **Input:** Two strings S and T.
2. **Output:** Longest common subsequence of S and T.

Subsequence

1. **Definition:** Given two sequences $X = \langle x_1, x_2,...,x_m \rangle$ and $Z = \langle z_1, z_2,...,z_k \rangle$, we say that Z is a subsequence of X if there is a strictly increasing sequence of k indices $i_1, i_2,...,i_k$ $(1 \leq i_1 < i_2 < ... < i_k \leq m)$ such that $Z = \langle x_{i_1}, x_{i_2},..., x_{i_k} \rangle$.
2. **Example:** Suppose $X = \langle$LIFESUCKS$\rangle$. Then $\langle$LFS$\rangle$, $\langle$ISUCKS$\rangle$, $\langle$FES$\rangle$ etc. can be subsequences of X. But $\langle$FIC$\rangle$, $\langle$ARA$\rangle$, $\langle$ERS$\rangle$ etc. are not subsequences of X.

Subsequence

1. **Definition:** Given two sequences $X = \langle\ x_1, x_2,...,x_m\ \rangle$ and $Z = \langle\ z_1, z_2,...,z_k\ \rangle$, we say that Z is a subsequence of X if there is a strictly increasing sequence of k indices $i_1, i_2,...,i_k$ $(1 \leq i_1 < i_2 < ... < i_k \leq m)$ such that $Z = \langle\ x_{i_1}, x_{i_2},..., x_{i_k}\ \rangle$.
2. **Example:** Suppose $X = \langle$LIFESUCKS$\rangle$. Then $\langle$LFS$\rangle$, $\langle$ISUCKS$\rangle$, $\langle$FES$\rangle$ etc. can be subsequences of X. But $\langle$FIC$\rangle$, $\langle$ARA$\rangle$, $\langle$ERS$\rangle$ etc. are not subsequences of X.

An Example

1. **Input:** $S = \langle \text{BDCB} \rangle$, $T = \langle \text{BACDB} \rangle$
2. **Common Subsequences:** $\langle \text{B} \rangle$, $\langle \text{BD} \rangle$, $\langle \text{BCB} \rangle$ etc.
3. **Longest Common Subsequence:** $\langle \text{BCB} \rangle$

An Example

1. **Input:** $S = \langle \text{BDCB} \rangle$, $T = \langle \text{BACDB} \rangle$
2. **Common Subsequences:** $\langle \text{B} \rangle$, $\langle \text{BD} \rangle$, $\langle \text{BCB} \rangle$ etc.
3. **Longest Common Subsequence:** $\langle \text{BCB} \rangle$

An Example

1. **Input:** $S = \langle \text{BDCB} \rangle$, $T = \langle \text{BACDB} \rangle$
2. **Common Subsequences:** $\langle \text{B} \rangle$, $\langle \text{BD} \rangle$, $\langle \text{BCB} \rangle$ etc.
3. **Longest Common Subsequence:** $\langle \text{BCB} \rangle$

Another Example

1. **Input:** $S = \langle \text{DABKC} \rangle$, $T = \langle \text{APBCK} \rangle$
2. **Common Subsequences:** $\langle \text{A} \rangle$, $\langle \text{AB} \rangle$, $\langle \text{ABK} \rangle$, $\langle \text{ABC} \rangle$ etc.
3. **Longest Common Subsequence:** $\langle \text{ABK} \rangle$, $\langle \text{ABC} \rangle$

Another Example

1. **Input:** $S = \langle \text{DABKC} \rangle$, $T = \langle \text{APBCK} \rangle$
2. **Common Subsequences:** $\langle \text{A} \rangle$, $\langle \text{AB} \rangle$, $\langle \text{ABK} \rangle$, $\langle \text{ABC} \rangle$ etc.
3. **Longest Common Subsequence:** $\langle \text{ABK} \rangle$, $\langle \text{ABC} \rangle$

Another Example

1. **Input:** $S = \langle \text{DABKC} \rangle$, $T = \langle \text{APBCK} \rangle$
2. **Common Subsequences:** $\langle \text{A} \rangle$, $\langle \text{AB} \rangle$, $\langle \text{ABK} \rangle$, $\langle \text{ABC} \rangle$ etc.
3. **Longest Common Subsequence:** $\langle \text{ABK} \rangle$, $\langle \text{ABC} \rangle$

Longest Common Subsequence Problem

Back to the Problem

How do we find the longest common subsequence of two strings?

Brute Force Algorithm

1. Enumerate all subsequences of S, and check if they are subsequences of T.

Brute Force Algorithm Complexity

1. Suppose S has length n and T has length m.
2. Then there are 2^n subsequences of S. Checking each of them if they are subsequence of T can be done greedily in $O(m)$.
3. So overall complexity is $O(m \cdot 2^n)$.
4. Unfortunately even the fastest computer today can't complete calculations in thousands of years if n, $m \geq 100$. So we need faster algorithm. Dynamic programming in the rescue!

Brute Force Algorithm Complexity

1. Suppose S has length n and T has length m.
2. Then there are 2^n subsequences of S. Checking each of them if they are subsequence of T can be done greedily in $O(m)$.
3. So overall complexity is $O(m \cdot 2^n)$.
4. Unfortunately even the fastest computer today can't complete calculations in thousands of years if n, $m \geq 100$. So we need faster algorithm. Dynamic programming in the rescue!

Brute Force Algorithm Complexity

1. Suppose S has length n and T has length m.
2. Then there are 2^n subsequences of S. Checking each of them if they are subsequence of T can be done greedily in $O(m)$.
3. So overall complexity is $O(m \cdot 2^n)$.
4. Unfortunately even the fastest computer today can't complete calculations in thousands of years if n, $m \geq 100$. So we need faster algorithm. Dynamic programming in the rescue!

Brute Force Algorithm Complexity

1. Suppose S has length n and T has length m.
2. Then there are 2^n subsequences of S. Checking each of them if they are subsequence of T can be done greedily in $O(m)$.
3. So overall complexity is $O(m \cdot 2^n)$.
4. Unfortunately even the fastest computer today can't complete calculations in thousands of years if n, $m \geq 100$. So we need faster algorithm. Dynamic programming in the rescue!

A Far Better Approach

1. Notice that there is a optimal substructure.
   Suppose $S = \langle s_1 s_2 .. s_n \rangle$, $T = \langle t_1 t_2 ... t_m \rangle$ and $Z = \langle z_1 z_2 ... z_k \rangle$ is their longest commmon subsequence.
   1. If $s_n = t_m$ then $z_k = s_n = t_m$. So Z_{k-1} is an LCS of S_{n-1} and T_{m-1}.
   2. If $s_n \neq t_m$ and $z_k \neq s_n$ then Z_k is an LCS of S_{n-1} and T_m.
   3. If $s_n \neq t_m$ and $z_k \neq t_m$ then Z_k is an LCS of S_n and T_{m-1}.
2. So the recursion will be:

$$dp[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ dp[i-1, j-1] & \text{if } i, j > 0 \text{ and } s_i = t_j \\ max(dp[i-1, j], dp[i, j-1]) & \text{if } i, j > 0 \text{ and } s_i \neq t_j \end{cases}$$

Here $dp[i,j]$ is the LCS of S and T till $i'th$ and $j'th$ index.

A Far Better Approach

1. Notice that there is a optimal substructure.
   Suppose $S = \langle s_1 s_2 .. s_n \rangle$, $T = \langle t_1 t_2 ... t_m \rangle$ and $Z = \langle z_1 z_2 ... z_k \rangle$ is their longest commmon subsequence.
   1. If $s_n = t_m$ then $z_k = s_n = t_m$. So Z_{k-1} is an LCS of S_{n-1} and T_{m-1}.
   2. If $s_n \neq t_m$ and $z_k \neq s_n$ then Z_k is an LCS of S_{n-1} and T_m.
   3. If $s_n \neq t_m$ and $z_k \neq t_m$ then Z_k is an LCS of S_n and T_{m-1}.
2. So the recursion will be:

$$dp[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ dp[i-1, j-1] & \text{if } i, j > 0 \text{ and } s_i = t_j \\ max(dp[i-1, j], dp[i, j-1]) & \text{if } i, j > 0 \text{ and } s_i \neq t_j \end{cases}$$

Here $dp[i,j]$ is the LCS of S and T till $i'th$ and $j'th$ index.

A Far Better Approach

1. Notice that there is a optimal substructure.
   Suppose $S = \langle s_1 s_2 .. s_n \rangle$, $T = \langle t_1 t_2 ... t_m \rangle$ and $Z = \langle z_1 z_2 ... z_k \rangle$ is their longest commmon subsequence.
   1. If $s_n = t_m$ then $z_k = s_n = t_m$. So Z_{k-1} is an LCS of S_{n-1} and T_{m-1}.
   2. If $s_n \neq t_m$ and $z_k \neq s_n$ then Z_k is an LCS of S_{n-1} and T_m.
   3. If $s_n \neq t_m$ and $z_k \neq t_m$ then Z_k is an LCS of S_n and T_{m-1}.
2. So the recursion will be:

$$dp[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ dp[i-1, j-1] & \text{if } i, j > 0 \text{ and } s_i = t_j \\ max(dp[i-1, j], dp[i, j-1]) & \text{if } i, j > 0 \text{ and } s_i \neq t_j \end{cases}$$

Here $dp[i,j]$ is the LCS of S and T till $i'th$ and $j'th$ index.

A Far Better Approach

1. Notice that there is a optimal substructure.
   Suppose $S = \langle s_1 s_2 .. s_n \rangle$, $T = \langle t_1 t_2 ... t_m \rangle$ and $Z = \langle z_1 z_2 ... z_k \rangle$ is their longest commmon subsequence.
   1. If $s_n = t_m$ then $z_k = s_n = t_m$. So Z_{k-1} is an LCS of S_{n-1} and T_{m-1}.
   2. If $s_n \neq t_m$ and $z_k \neq s_n$ then Z_k is an LCS of S_{n-1} and T_m.
   3. If $s_n \neq t_m$ and $z_k \neq t_m$ then Z_k is an LCS of S_n and T_{m-1}.
2. So the recursion will be:

$$dp[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ dp[i-1, j-1] & \text{if } i, j > 0 \text{ and } s_i = t_j \\ max(dp[i-1, j], dp[i, j-1]) & \text{if } i, j > 0 \text{ and } s_i \neq t_j \end{cases}$$

Here $dp[i,j]$ is the LCS of S and T till $i'th$ and $j'th$ index.

An Example Using Dynamic Programming Algorithm

1. Suppose S = ⟨ADAPT⟩, T = ⟨DBPT⟩.
2. Then the DP table will according to the recursion (1-based indexing)

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	0	0	0	0
2	D	0	1	1	1	1
3	A	0	1	1	1	1
4	P	0	1	1	2	2
5	T	0	1	1	2	3

An Example Using Dynamic Programming Algorithm

1. Suppose S = ⟨ADAPT⟩, T = ⟨DBPT⟩.
2. Then the DP table will according to the recursion (1-based indexing)

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	0	0	0	0
2	D	0	1	1	1	1
3	A	0	1	1	1	1
4	P	0	1	1	2	2
5	T	0	1	1	2	3

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x					
1	A					
2	D					
3	A					
4	P					
5	T					

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	0			
2	D	0				
3	A	0				
4	P	0				
5	T	0				

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	0	0		
2	D	0				
3	A	0				
4	P	0				
5	T	0				

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	0	0	0	
2	D	0				
3	A	0				
4	P	0				
5	T	0				

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ **0**	↑ 0
2	D	0				
3	A	0				
4	P	0				
5	T	0				

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ **0**	↑ 0
2	D	0	↖ 1			
3	A	0				
4	P	0				
5	T	0				

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↑ 0
2	D	0	↖ 1	← 1		
3	A	0				
4	P	0				
5	T	0				

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ **0**	↑ 0
2	D	0	↖ 1	← 1	← 1	
3	A	0				
4	P	0				
5	T	0				

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ **0**	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0				
4	P	0				
5	T	0				

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0	↑ 1			
4	P	0				
5	T	0				

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ **0**	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0	↑ 1	↑ 1		
4	P	0				
5	T	0				

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0	↑ 1	↑ 1	↑ 1	
4	P	0				
5	T	0				

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ **0**	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0	↑ 1	↑ 1	↑ 1	↑ 1
4	P	0				
5	T	0				

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ **0**	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0	↑ 1	↑ 1	↑ 1	↑ 1
4	P	0	↑ 1			
5	T	0				

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	0	0	0	0
2	D	0	1	1	1	1
3	A	0	1	1	1	1
4	P	0	1	1		
5	T	0				

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ **0**	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0	↑ 1	↑ 1	↑ 1	↑ 1
4	P	0	↑ 1	↑ 1	↖ 2	
5	T	0				

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ **0**	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0	↑ 1	↑ 1	↑ 1	↑ 1
4	P	0	↑ 1	↑ 1	↖ 2	← 2
5	T	0				

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ **0**	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0	↑ 1	↑ 1	↑ 1	↑ 1
4	P	0	↑ 1	↑ 1	↖ 2	← 2
5	T	0	↑ 1			

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0	↑ 1	↑ 1	↑ 1	↑ 1
4	P	0	↑ 1	↑ 1	↖ 2	← 2
5	T	0	↑ 1	↑ 1		

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0	↑ 1	↑ 1	↑ 1	↑ 1
4	P	0	↑ 1	↑ 1	↖ 2	← 2
5	T	0	↑ 1	↑ 1	↑ 2	

Construction of LCS Table

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ **0**	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0	↑ 1	↑ 1	↑ 1	↑ 1
4	P	0	↑ 1	↑ 1	↖ 2	← 2
5	T	0	↑ 1	↑ 1	↑ 2	↖ 3

Construction of Solution

1. Start from $dp[n, m]$
2. From each cell go to the direction in that cell.
3. In our path from $dp[i, j]$ compare $S[i]$ and $T[j]$. If $S[i] = T[j]$ include this character in our LCS.
4. Stop when reached in $dp[0, 0]$. Now we have our LCS in reverse order.

Construction of Solution

1. Start from $dp[n, m]$
2. From each cell go to the direction in that cell.
3. In our path from $dp[i, j]$ compare $S[i]$ and $T[j]$. If $S[i] = T[j]$ include this character in our LCS.
4. Stop when reached in $dp[0, 0]$. Now we have our LCS in reverse order.

Construction of Solution

1. Start from $dp[n, m]$
2. From each cell go to the direction in that cell.
3. In our path from $dp[i, j]$ compare $S[i]$ and $T[j]$. If $S[i] = T[j]$ include this character in our LCS.
4. Stop when reached in $dp[0, 0]$. Now we have our LCS in reverse order.

Construction of Solution

1. Start from $dp[n, m]$
2. From each cell go to the direction in that cell.
3. In our path from $dp[i, j]$ compare $S[i]$ and $T[j]$. If $S[i] = T[j]$ include this character in our LCS.
4. Stop when reached in $dp[0, 0]$. Now we have our LCS in reverse order.

Construction of Solution

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ **0**	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0	↑ 1	↑ 1	↑ 1	↑ 1
4	P	0	↑ 1	↑ 1	↖ 2	← 2
5	T	0	↑ 1	↑ 1	↑ 2	↖ 3

Construction of Solution

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ **0**	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0	↑ 1	↑ 1	↑ 1	↑ 1
4	P	0	↑ 1	↑ 1	↖ 2	← 2
5	T	0	↑ 1	↑ 1	↑ 2	↖ 3

Construction of Solution

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ **0**	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0	↑ 1	↑ 1	↑ 1	↑ 1
4	P	0	↑ 1	↑ 1	↖ 2	← 2
5	T	0	↑ 1	↑ 1	↑ 2	↖ 3

Construction of Solution

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0	↑ 1	↑ 1	↑ 1	↑ 1
4	P	0	↑ 1	↑ 1	↖ 2	← 2
5	T	0	↑ 1	↑ 1	↑ 2	↖ 3

Construction of Solution

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	0	0	0	0
2	D	0	1	1	1	1
3	A	0	1	1	1	1
4	P	0	1	1	2	2
5	T	0	1	1	2	3

Construction of Solution

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0	↑ 1	↑ 1	↑ 1	↑ 1
4	P	0	↑ 1	↑ 1	↖ 2	← 2
5	T	0	↑ 1	↑ 1	↑ 2	↖ 3

Construction of Solution

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0	↑ 1	↑ 1	↑ 1	↑ 1
4	P	0	↑ 1	↑ 1	↖ 2	← 2
5	T	0	↑ 1	↑ 1	↑ 2	↖ 3

Complexity

1. Each entity of the dp table can be computed in $O(1)$.
2. There are $|S| \cdot |T|$ entities to be filled. So total time complexity is $O(|S| \cdot |T|)$.
3. Similarly total memory complexity is also $O(|S| \cdot |T|)$.

Complexity

1. Each entity of the dp table can be computed in $O(1)$.
2. There are $|S| \cdot |T|$ entities to be filled. So total time complexity is $O(|S| \cdot |T|)$.
3. Similarly total memory complexity is also $O(|S| \cdot |T|)$.

Complexity

1. Each entity of the dp table can be computed in $O(1)$.
2. There are $|S| \cdot |T|$ entities to be filled. So total time complexity is $O(|S| \cdot |T|)$.
3. Similarly total memory complexity is also $O(|S| \cdot |T|)$.

Longest Common Subsequence Problem

Further Optimization

Can we do better?

Further Optimization

1. Unfortunately, unless something is said explicitly about input alphabet, no known optimization on time complexity is possible. Some optimizatios,
    1. For problems with a bounded alphabet size, the Method of Four Russians can be used to reduce the running time of the dynamic programming algorithm by a logarithmic factor.
    2. For problems where S and T has no repeated alphabet, Longest common subsequence problem can be reduced to Longest increasing subsequence. Thus solving the problem in $O(n \cdot logn)$.
2. But memory optimization is certainly possible. In each dp state we only need last two rows. So we can keep last two rows using even, odd sequence. Thus giving memory complexity $O(2 \cdot |T|)$

Further Optimization

1. Unfortunately, unless something is said explicitly about input alphabet, no known optimization on time complexity is possible. Some optimizatios,
   1. For problems with a bounded alphabet size, the Method of Four Russians can be used to reduce the running time of the dynamic programming algorithm by a logarithmic factor.
   2. For problems where S and T has no repeated alphabet, Longest common subsequence problem can be reduced to Longest increasing subsequence. Thus solving the problem in $O(n \cdot logn)$.
2. But memory optimization is certainly possible. In each dp state we only need last two rows. So we can keep last two rows using even, odd sequence. Thus giving memory complexity $O(2 \cdot |T|)$

Further Optimization

1. Unfortunately, unless something is said explicitly about input alphabet, no known optimization on time complexity is possible. Some optimizatios,
   1. For problems with a bounded alphabet size, the Method of Four Russians can be used to reduce the running time of the dynamic programming algorithm by a logarithmic factor.
   2. For problems where S and T has no repeated alphabet, Longest common subsequence problem can be reduced to Longest increasing subsequence. Thus solving the problem in $O(n \cdot logn)$.
2. But memory optimization is certainly possible. In each dp state we only need last two rows. So we can keep last two rows using even, odd sequence. Thus giving memory complexity $O(2 \cdot |T|)$

Further Optimization

1. Unfortunately, unless something is said explicitly about input alphabet, no known optimization on time complexity is possible. Some optimizatios,
    1. For problems with a bounded alphabet size, the Method of Four Russians can be used to reduce the running time of the dynamic programming algorithm by a logarithmic factor.
    2. For problems where S and T has no repeated alphabet, Longest common subsequence problem can be reduced to Longest increasing subsequence. Thus solving the problem in $O(n \cdot logn)$.
2. But memory optimization is certainly possible. In each dp state we only need last two rows. So we can keep last two rows using even, odd sequence. Thus giving memory complexity $O(2 \cdot |T|)$

Summary

- Longest common subsequence is a well known problem that can be solve using dynamic programming approach.
- Dymanic programming solution has time complexity $O(|S| \cdot |T|)$.
- Dymanic programming solution can further be optimized using constraints on input alphabet.

For Further Reading I

Thomas H. Cormen, C.E. Leiserson, R. L. Rivest, and C. Stein.
Introduction to Algorithms.
The MIT Press, 1989.

Masek, William J., Paterson, Michael S.
A faster algorithm computing string edit distances
Journal of Computer and System Sciences, , 20 (1): 18–31,1980.