Big Data Programming 1

Coding Challenge 1

Exercise 1: Common Subsequence of two strings

A. The approach & steps followed towards the solution

I started with the intuition of linear string matching approach. Though a brute-force method, it was promising in the sense that it would get me a basic solution to start with.

In my solution, I took the first string as pivot, and used the other string to match its characters in the pivot.

For each character in the pivot string, the algorithm looks for a matching character in the second string. If there is a match, the *LastMatchIndex* variable is set to the index of the character in the second string. Otherwise, it keep scanning it for that character upto the end.

LastMatchIndex helps us keep track of the index till which we have successfully found a match.

Please note that the algorithm needs to consider both the strings as pivot, so as to determine the Longest Common Subsequence. Following this way, we have 2 Common Subsequences; and subsequently we can easily determine the Longest CS using their length. This is the reason we have 2 calls to the function **CommonSubsequence()** - one with String 1 as pivot and the other call with String 2 as pivot.

Once I had this solution in place, I looked for other solutions and suggestions for finding the LCS, on the internet. The majority of solutions that I got across used recursion.

Why I chose to submit this solution

- First and foremost – This is my original.
- The solutions on the internet had the similar complexity, with an overhead of recursive calls, thereby consuming larger set of resources without taking much advantage.

B. Pseudocode (For finding LCS)

GetLCS(String *str1*, String *str2*)
1. String *strCS1*, *strCS2*
2. Integer *nIndex1* = **CommonSubsequence**(*str1*, *str2*, *strCS1*)
3. Integer *nIndex2* = **CommonSubsequence**(*str2*, *str1*, *strCS2*)

4. **if** *nIndex1* <= 0 **and** *nIndex2* <= 0
5. **then print** "No LCS Found"
6. **return**

7. **if** Length(*strCS1*) >= Length(*strCS2*)
8. **then print** *strCS1*
9. **else**
10. **print** *strCS2*

CommonSubsequence(String *str1*, String *str2*, String *strLCS*)

1. Set *strLCS* as an empty string.
2. *LastMatchIndex* = -1
3. **for** i = 0 up to (Length of str1) - 1
4. **for** j = (LastMatchIndex + 1) up to (Length of str2)
5. **if** *str1*[i] == str2[j]
6. *strLCS* = *strLCS* + str[i]
7. *LastMatchIndex* = j
8. **break** for the current value of 'i'

Computation Time of the solution:

- Best Case (Identical strings): N
- Worst Case (When there is no LCS found): N^2

Notes:

- This solution follows case-sensitivity.
- This solution assumes that the input strings can contain any printable character - including the numbers.

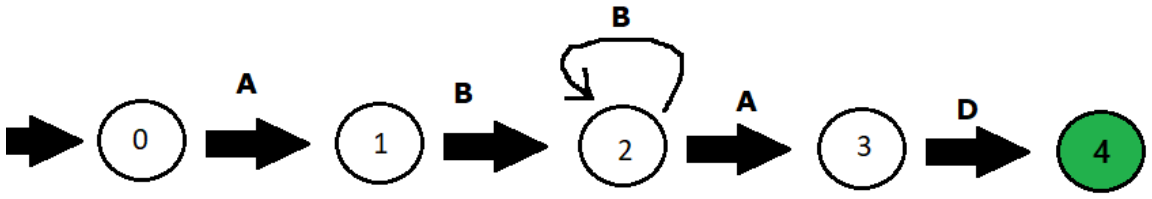
Exercise 2: Pattern Matching

A. The approach & steps followed towards the solution

I used the DFA – Deterministic Finite Automata principle to determine whether the given text string fits into the pattern.

In order to accomplish the task of matching the pattern, the pattern is required to be translated into a Transition Diagram and Transition Table.

For example, the Transition Diagram for a pattern **AB*AD** is shown as follows:



Here, the initial state is denoted by '0', and the acceptance state is '4' (shaded with green). Using this, we can now, generate a transition table that looks like:

State \ Input	A	B	C	D	E	F	.	.	.	.	Z
0	1	-	-	-	-	-	-	-	-	-	-
1	-	2	-	-	-	-	-	-	-	-	-
2	3	2	-	-	-	-	-	-	-	-	-
3	-	-	-	4	-	-	-	-	-	-	-
4	-	-	-	-	-	-	-	-	-	-	-

Each row in the table shows the current state, and each column shows the next character input. Using this table, we can determine the state as each character (in the text) is read in sequence.

If, after reading the last character in the text, we are in the State 4, we can say that the text matches the pattern. Otherwise, it's a mismatch.

In my approach, I have defined a transition table, using the pattern string. And subsequently, the text string is run through the transition table, to determine if the output should be TRUE or FALSE, as defined in the problem statement.

In this solution, I have made the following **assumptions**:

1. In case of '*', only the '*' is replaced with zero(or more) occurences of previous single character. In other words, the sub-string 'A*' will have at least one A in order to match the pattern.

2. Only {A-Z} (in upper case) are allowed to be entered for the Pattern as well as Text string.

B. Pseudocode (For Pattern Matching)

MatchPattern(String *strText*, String *strPattern*)

1. *Acceptance_State =* **ConstructTransitionTable***(strPattern)*;
2. **if MatchText**(*strText*, *Acceptance_State*))
3. Print “True”
4. else
5. Print “False”

ConstructTransitionTable*(*String *argPattern)*

1. int TransitionTable[200][26]; // 2-dimentional array for 26 characters(A-Z) and upto 200 length
2. if *argPattern* == “.*”
3. **for** i = 0 up to 199
4. **for** i = 0 up to 25
5. *TransitionTable*[i][j] = 1
6. return 1

7. Initialise the TransitionTable
8. **for** i = 0 up to 199
9. **for** i = 0 up to 25
10. *TransitionTable*[i][j] = -1

11. *current_state = 0*
12. prev_char = ‘\0’

13. Set the state transitions
14. **for** i = 0 up to Length(argPattern)
15. **if**(argPattern[i] >= 'A' **and** *argPattern*[i] <= 'Z' **and** *TransitionTable*[current_state][argPattern[i]] == -1)
16. TransitionTable[current_state][argPattern[i]] = current_state+1;
17. current_state++
18. prev_char = argPattern[i]
19. Skip the current interation

20. **if**(argPattern[i] == '*' **and** prev_char != '\0')
21. TransitionTable[current_state][prev_char] = *current_state*
22. Skip the current interation

23. **if**(argPattern[i] == '.')
24. for (int j = 0; j < 26; j++) // For 26 character-set {'A' to 'Z'}
25. TransitionTable[*current_state*][*j*] = *current_state* + 1;

26. prev_char = argPattern[i]
27. current_state++
28. Skip the current interation

29. **return** *current_state*
30. ----End of ConstructTransitionTable()---

MatchText(string *argText*, int *argAcceptanceState*)

```
1. current_state = 0;
2. for(int i = 0; i < argText.length(); i++)
   {
      current_state = TransitionTable[current_state][argText[i]];
   }

   return (current_state == argAcceptanceState); //Return true if the final state is the AccpetanceState
}
```

C. GitHub link to repository

https://github.com/arungupta21/BD_Programming_CodeChallenge1.git

---

Submitted by:

Arun Gupta
BDBA, April'19 batch
Enrolment ID: 11012482