CS 570: Analysis of Algorithms – H9

Submitted by: Indronil Bhattacharjee

Problem 15-2

Q- Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input '*character*', your algorithm should return '*carac*'. What is the running time of your algorithm?

```
Longest-Palindromic-Subsequence(s):
1    n = length of s
2    Initialize a 2D array dp of size n x n initialized to all 0s
3    for i = 0 to n-1:
4         dp[i][i] = 1
5    for l = 2 to n:
6         for i = 0 to n-l:
7             j = i + l - 1
8             if s[i] == s[j]:
9                 dp[i][j] = dp[i+1][j-1] + 2
10            else:
11                 dp[i][j] = max(dp[i+1][j], dp[i][j-1])
12   lps_length = dp[0][n-1]
13   Initialize an array lps of size lps_length
14   i = 0
15   j = n-1
16   k = 0
17   while i < j:
18        If s[i] == s[j]:
19            lps[k] = s[i]
20            lps[lps_length-1-k] = s[j]
21            Increment k, i, and decrement j
22       else if dp[i+1][j] > dp[i][j-1]:
23            Increment i
24       else:
25            Decrement j
26   if i == j:
27        lps[k] = s[i]
28   return lps
```

The running time of this algorithm is **O(n^2)**, where n is the length of the input string. Let's break down the time complexity of the algorithm outlined in the pseudocode line by line:

1. `n = length of s`: This line has a time complexity of **O(1)** as it simply assigns the length of the input string s to the variable n.

2. `Initialize a 2D array dp of size n x n initialized to all 0s`: Initializing the 2D array takes **O(n^2)** time since it involves creating an array of size n x n and setting all its elements to zero.
3. `for i = 0 to n-1`: This loop runs n times, so its time complexity is **O(n).**
4. `dp[i][i] = 1`: This assignment operation takes **O(1)** time since it just assigns a constant value to a specific cell in the array.
5. `for l = 2 to n`: This loop runs from 2 to n, so it executes n-1 times. Hence, its time complexity is **O(n).**
6. `for i = 0 to n-l`: This loop runs (n - l + 1) times for each value of l. Since l varies from 2 to n, the total number of iterations of this loop is the sum of the first n-1 integers, which is **O(n^2).**
7. `j = i + l - 1`: This line has a time complexity of **O(1)** as it simply calculates the value of j based on i and l.
8. `if s[i] == s[j]`: This comparison operation takes **O(1)** time.
9. `dp[i][j] = dp[i+1][j-1] + 2`: This assignment operation takes **O(1)** time.
10. `else`: This branch of the if-else statement takes **O(1)** time.
11. `dp[i][j] = max(dp[i+1][j], dp[i][j-1])`: This assignment operation takes **O(1)** time.
12. `lps_length = dp[0][n-1]`: This line has a time complexity of **O(1)** as it simply accesses a specific cell in the 2D array.
13. `Initialize an array lps of size lps_length`: This operation takes O(lps_length) time, which is **O(n)** in the worst case.
14. `i = 0`: These assignment take **O(1)** time.
15. `j = n-1`: These assignment take **O(1)** time.
16. `k = 0`: These assignment take **O(1)** time.
17. `while i < j`: This loop runs at most n/2 times in the worst case, so its time complexity is **O(n).**
18. `if s[i] == s[j]`: This comparison operation takes **O(1)** time.
19. `lps[k] = s[i]`: This assignment operation takes **O(1)** time.
20. `lps[lps_length-1-k] = s[j]`: This assignment operation also takes **O(1)** time.
21. `Increment k, i, and decrement j`: These operations take **O(1)** time.
22. `else if dp[i+1][j] > dp[i][j-1]`: This comparison operation takes **O(1)** time.
23. `Increment i`: This operation takes **O(1)** time.
24. `else`: This branch of the if-else statement takes **O(1)** time.
25. `Decrement j`: This operation takes **O(1)** time.
26. `if i == j`: This comparison operation takes **O(1)** time.
27. `lps[k] = s[i]`: This assignment operation takes **O(1)** time.
28. `return lps`: This operation takes **O(1)** time.

Overall, the time complexity of the algorithm is dominated by the initialization of the 2D array dp in line 2 and loop iterations in the line 6, which is **O(n^2).**

Hence, the overall time complexity of the algorithm is O(n^2)