Sabancı University
Faculty of Engineering and Natural Sciences

CS301 – Algorithms

Homework 4

Due: April 2, 2024 @ 23.55
(upload to SUCourse)

PLEASE NOTE: Ayşe Elif Aktaş - 27802

- Provide only the requested information and nothing more. Unreadable, unintelligible, and irrelevant answers will not be considered.
- Submit only a PDF file. (-20 pts penalty for any other format)
- Not every question of this homework will be graded. We will announce the question(s) that will be graded after the submission.
- You can collaborate with your `TA/INSTRUCTOR ONLY` and discuss the solutions of the problems. However, you have to write down the solutions on your own.
- Plagiarism will not be tolerated.

Late Submission Policy:

- Your homework grade will be decided by multiplying what you normally get from your answers by a "submission time factor (STF)".
- If you submit on time (i.e. before the deadline), your STF is 1. So, you don't lose anything.
- If you submit late, you will lose 0.01 of your STF for every 5 mins of delay.
- We will not accept any homework later than 500 mins after the deadline.
- SUCourse's timestamp will be used for STF computation.
- If you submit multiple times, the last submission time will be used.

Question 1

A palindrome is a sequence of characters that reads the same backward as forward. In other words, if you reverse the sequence, you get the same sequence. For example, "a", "bb", "aba", "ababa", "aabbaa" are palindromes. We also have real words which are palindromes, like "noon", "level", "rotator", etc.

We also have considered the concept of a subsequence in our lectures. Given a word A, B is a subsequence of A, if B is obtained from A by deleting some symbols in A. For example, the following are some of the subsequences of the sequence "*abbdcacdb*": "*da*", "*bcd*", "*abba*", "*abcd*", "*bcacb*", "*bbccdb*", etc.

The Longest Palindromic Subsequence (LPS) problem is finding the longest subsequences of a string that is also a palindrome. For example, for the sequence "*abbdcacdb*", the longest palindromic subsequence is "*bdcacdb*", which has length 7. There is no subsequence of "*abbdcacdb*" of length 8 which is a palindrome.

One can find the length of LPS of a given sequence by using dynamic programming. As common in dynamic programming, the solution is based on a recurrence.

Given a sequence $A = a_1a_2 \dots a_n$, let $A[i, j]$ denote the sequence $a_ia_{i+1} \dots a_j$. Hence it is part of the sequence that starts with a_i and ends with a_j (including these symbols). For example, if $A = abcdef$, $A[2, 4] = bcd$, $A[1, 5] = abcde$, $A[3, 4] = cd$, etc.

For a sequence $A = a_1a_2 \dots a_n$, let us use the function $L[i, j]$ to denote the length of the longest palindromic subsequence in $A[i, j]$.

(a) If we have a sequence $A = a_1a_2 \dots a_n$, for which values of i and j, $L[i, j]$ would refer to the length of the longest palindromic subsequence in A?

$i = 1 \quad j = n$

(b) Write the recurrence for $L[i, j]$.

$$L[i,j] = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ L[i+1, j-1] + 2 & \text{if } i < j \text{ and } a_i = a_j \\ \max(L[i+1, j], L[i, j-1]) & \text{if } i < j \text{ and } a_i \neq a_j \end{cases}$$

(c) What would be the worst case time complexity of the algorithm in Θ notation? Why?

$O(n^2)$

(d) Considering a memoization-based approach, complete the following pseudocode for the dynamic programming solution to the LPS problem. Assume the existence of a 2D array 'dp' of size $n \times n$ initialized to -1, where n is the length of sequence A, and a function 'LPS(i, j)' that computes the length of the longest palindromic subsequence in $A[i, j]$.

```
function LPS(i, j):
    if dp[i][j] != -1:
        return dp[i][j]
    if i > j:
        dp[i][j] = 0
    elif i == j:
        dp[i][j] = 1
    elif A[i] == A[j]:
        dp[i][j] = LPS(i+1, j-1) + 2
    else:
        dp[i][j] = max(LPS(i+1, j), LPS(i, j-1))
    return dp[i][j]
```

Question 2

To compute the depth of a given node in an RBT in constant time, consider augmenting the depths of nodes as additional attributes in the nodes of the RBT. Please explain by an example why this augmentation increases the asymptotic time complexity of inserting a node into an RBT in the worst case.

When augmenting the depths of nodes as additional attributes in RBT we are essentially keeping extra piece of information for each node which represents the depth of that node. This depth value calculated based on the node's position in the tree relative to its parent.
While augmenting the tree with this information allows us to compute the depth of a given node in constant time it also introduces an additional step in the process of inserting a node into the RBT. When inserting a node now we not only need to perform standart RBT insertion such as tree rotations and color adjustments we also need to update the depth attributes of affected nodes. To update we need to traverse the path from inserted node to the root updating their depth value along the way. This traversal operation takes time proportional to the height of the tree. In worst case the height of an RBT logarithmic in the number of nodes resulting in logarithmic time complexity for the depth update operation Therefore augmenting the depths of nodes in an RBT increases asymptotic time complexity of inserting a node in the worst case. The standart RBT insertion has time complexity of $O(\log n)$ and additional depth update operation takes an additional $O(\log n)$ time resulting in time complexity of $O(\log n + \log n) = O(\log n)$. While the increase in worst might not change the worst case complexity of $O(\log n)$ for each insert operation, the need to update the depths of many nodes after each insertion adds overhead. This overhead in practice increases the constant involved in the $O(\log n)$ complexity making insert operation more costly.

Question 3

Write pseudocode for LEFT-ROTATE that operates on nodes in an interval tree and updates the max attributes in $O(1)$ time.

```
LEFT_ROTATE (T, x)
    y = x.right
    x.right = y.left
    if y.left != NIL
        y.left.parent = x
    y.parent = x.parent
    if x.parent == NIL
        T.root = y
    elseif x == x.parent.left
        x.parent.left = y
    else
        x.parent.right = y
    y.left = x
    x.parent = y
    x.max = max(x.interval.high, x.left.max, x.right.max)
    y.max = max(y.interval.high, y.left.max, y.right.max)
```

→ T: interval tree

→ x: node around which the left rotation is performed

→ y: right child of x, which becomes the new root of the subtree initially rooted at x

→ each node has 'interval' attribute representing the interval stored at this node, with interval.high being the higher endpoint of the interval

→ each node also has 'max' attribute which is the maximum 'high' value of any interval in the subtree rooted at this node