Foundations of Algorithms Fall 2024
Programming Assignment 2

Rong Fan

1. Give an efficient algorithm that takes strings s, x, and y and decides if s is an interweaving of x and y. Derive the computational complexity of your algorithm.

   Since we don't know where does the interweave start and whether it exists, we have to check every starting point until we find a interweave. So the following algorithm is run $O(|s|)$ times. So let's say the starting point is s[a]
   To check whether a starting point has a corresponding interweave, we use a dynamic programming method.
   We create a 2d array dp, where each element stores a boolean:
   A true at $dp[i][j]$ represents the sub-string of s from $s[a]$ to $s[a+i+j-1]$ (inclusive) is a interweave of x and y with s' is a repetition of x of length i and s" is a repetition of y of length j
   So that if $i >= |x|$ and $j >= |y|$, and $dp[i][j]$ is true, we found our interweave: from $s[a]$ to $s[a+i+j-1]$, and return true
   Otherwise we keep going updating dp until we reach the end of s, and continue onto the next starting point.
   We initialize dp by setting $dp[0][0] = true$
   then for each loop, we update dp diagonally, from $dp[k][0]$, $dp[k-1][1]$ to $dp[0][k]$ where k is the length of the sub-string of s we are checking, and k=i+j.
   When we are at $dp[i][j]$, if ($dp[i-1][j]$ is true and $s[a+k-1] = x[(i-1)\%|x|]$) or ($dp[i][j-1]$ is true and $s[a+k-1] = y[(j-1)\%|y|]$) we put $dp[i][j]$ as true, otherwise we put false.
   We can reduce average run time by checking whether a whole diagonal is false, if so, we can stop checking this starting point since there is no way for future elements to be true again. It indicates that the sub-string is not possible to be a beginning of a interweave.

Algorithm 1. Check interweaving problem

Input: three strings made of "1"s and "0"s: s, x, y
Output: a boolean indicating whether s is an interweave of x and y

```
1: function CHECKINTERWEAVING(s, x, y)
2:     for a ← 0 to |s| − 1 do
3:         if |s| − a < |x| + |y| then                ▷ If the sub-string doesn't contain enough elements
4:             return false
5:         create 2d array dp that store boolean
6:         dp[0][0] = true
7:         for k ← 1 to |s| − a − 1 do                ▷ k is the length of the sub-string
8:             isAllFalse = true                      ▷ check if the whole diagonal is false
9:             if dp[k − 1][0] and s[a + k − 1] = x[(k − 1)%|x|] then   ▷ We check when i=0 and j=0 directly so no
   index error or extra checks
10:                dp[k][0] = true
11:                isAllFalse = false
12:            else
13:                dp[k][0] = false
14:            if dp[0][k − 1] and s[a + k − 1] = y[(k − 1)%|y|] then
15:                dp[0][k] = true
16:                isAllFalse = false
17:            else
18:                dp[0][k] = false
19:            for i ← 1 to k − 1 do                  ▷ i,j are the index of dp, but we access x[i-1] at i
20:                if (dp[i − 1][k − i] and s[a + k − 1] = x[(i − 1)%|x|] )
21: or (dp[i][k − i − 1] and s[a + k − 1] = y[(k − i − 1)%|y|]) then
22:                    dp[i][k − i] = true                                          ▷ j=k-i
23:                    if i >= |x| and k − i >= |y| then
24:                        return true                ▷ we found an interweave from s[a] to s[a+k-1]
25:                    isAllFalse = false
26:            if isAllFalse then
27:                break          ▷ we skip future checks with this starting point since they can never be true again
28:    return false
```

The worst case time complexity of this algorithm is $O(|s|^3)$. We can see that we go through the whole s array $O(|s|)$ times by finding the starting point. And within each loop, we iterate through the dp table which has a size of at most $O(|s|)$ by $O(|s|)$ which is $O(|s|^2)$. The average case should be much faster as we usually don't need to go through the whole string. The current algorithm, with enough pruning, makes it look more like $O(|s|^{3/2})$, but I can't give an exact average time complexity. And there is no clean way to demonstrate the worst case time complexity. Therefore a graph is unnecessary to demonstrate the time complexity as the worst case time complexity can't be demonstrated by it.

2. The code is modifying this pseudocode by adding counting comparison in between. It's hard to come up with a case that satisfies the worst case that is long enough to show the asymptotic complexity. It depends on not only the length of the strings but also the structure.