Logan Griffin

EN.605.621.82.SP25

Foundations of Algorithms

Programming Assignment 2

4/15/2025

(a) [50 points] 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.

Algorithm:

Algorithm 1. Untangling

Input: A sequence of symbols S, a signal string x, and a signal string y.
Output: Message stating whether or not S contains an interweaving of x and y.

```
1:  function NEWSTATE(char, x, y, seen, entries)
2:      length_x = len(x)
3:      length_y = len(y)
4:
5:      if char == x[0] and char == y[0] then
6:          stateX = (char, ySubstring, xExpected, yExpected)
7:          ID = (char, xSubstring, ySubstring, len(xSubstring), len(ySubstring))
8:          checkNewState = (ID, stateX, seen, char, entries, x, y)
9:
10:         stateY = (xSubstring, char, xExpected, yExpected)
11:         ID = (char, xSubstring, ySubstring, len(xSubstring), len(ySubstring))
12:         checkNewState = (ID, stateY, seen, char, entries, x, y)
13:
14:     else if char == x[0] then
15:         stateX = (char, ySubstring, xExpected, yExpected)
16:         ID = (char, xSubstring, ySubstring, len(xSubstring), len(ySubstring))
17:         checkNewState = (ID, stateX, seen, char, entries, x, y)
18:
19:     else if char == y[0] then
20:         stateY = (xSubstring, char, xExpected, yExpected)
21:         ID = (char, xSubstring, ySubstring, len(xSubstring), len(ySubstring))
22:         checkNewState = (ID, stateY, seen, char, entries, x, y)
```

```
23:
24: function updateStates(prefix, newPrefix, char, x, y, seen, entries)
25:     if prefix in entries and newPrefix not in entries then          ▷ get valid states to update
26:         for states in entries for prefix do                  ▷ go through states and add new char
27:             xNext = state[xExpected]                              ▷ char the state needs for x
28:             yNext = state[yExpected]                              ▷ char the state needs for y
29:
30:             if xNext == char and yNext == char then                              ▷ if matches both
31:                 stateX = (state[xSubstring] + char, ySubstring, xExpected, yExpected)    ▷ add
   char to x repetition
32:                 ID = (char, xSubstring, ySubstring, len(xSubstring), len(ySubstring))
33:                 checkNewState = (ID, stateX, seen, char, entries, x, y)
34:
35:                 stateY = (xSubstring, state[ySubstring] + char, xExpected, yExpected)    ▷ add
   char to y repetition
36:                 ID = (char, xSubstring, ySubstring, len(xSubstring), len(ySubstring))
37:                 if checkNewState = (ID, stateY, seen, char, entries, x, y) then
38:                     return True                                   ▷ if check found solution
39:
40:             else if xNext == char then                                       ▷ if matches only x
41:                 stateX = (state[xSubstring] + char, ySubstring, xExpected, yExpected)
42:                 ID = (char, xSubstring, ySubstring, len(xSubstring), len(ySubstring))
43:                 if checkNewState = (ID, stateX, seen, char, entries, x, y) then
44:                     return True                                   ▷ if check found solution
45:
```

```
46:             else if yNext == char then                                       ▷ if only matches y
47:                 stateY = (xSubstring, state[ySubstring] + char, xExpected, yExpected)
48:                 ID = (char, xSubstring, ySubstring, len(xSubstring), len(ySubstring))
49:                 if checkNewState = (ID, stateY, seen, char, entries, x, y) then
50:                     return True                                   ▷ if check found solution
51:     return False                                          ▷ if no solution has been found
52:
53: function checkNewState(ID, state, seen, key, entries, x, y)
54:     if state not in seen then                                                ▷ if state is new
55:         if ID[3] == len(x) and ID[4] == len(y) then                          ▷ check for solution
56:             return True
57:         else
58:             seen.add(ID)                                                      ▷ add to seen
59:
60:             if key not in entries then                          ▷ check if prefix has been seen
61:                 entries[key] = []
62:             entries[key].append(state)                          ▷ add states to prefix list
63:         return False                                                  ▷ if no solution found
64:
```

```
65: function UNTANGLE(S, x, y, entries)
66:     length_x = len(x)
67:     length_y = len(y)
68:     seen = set()                                   ▷ new set to store seen states
69:
70:     for i, 0:len(S) do                             ▷ iterate through S
71:         char = S[i]
72:         newState(char, x, y, seen, entries)        ▷ make new state from current character
73:
74:         for j, i-1 down to -1 do                   ▷ consider all prefixes up to i
75:             prefix = S[j:i]                        ▷ characters already considered in outer loop
76:             newPrefix = S[j:i+1]                   ▷ add new char to prefix
77:
78:             if updateState(prefix, newprefix, char, x, y, seen, entries) then
79:                 return True                        ▷ if solution found
80:     return False                                   ▷ if no solution found
81:
82: function MAIN
83:     entries = {}                                   ▷ new dictionary to store entries
84:     if untangle(S, x, y, entries) then
85:         return True
86:     else
87:         return False
```

The algorithm works by iterating over each character in the sequence of symbols, S. Since there can be leading characters that may not be part of the interweaving of the signals x and y, the algorithm starts a new possible interweaving at each character in S. In addition to starting a new possible interweaving at each character, the algorithm considers each possible prefix up to each new character by backtracking. For example, if S was 10010, at the second "1" instance in position 3, the algorithm would consider the following substrings: first "1", then "01", then "001", then finally "1001". This dynamic programming approach ensures that all possible states are considered. The states are stored in a set to make sure that repeat states do not have to be stored and considered multiple times in future iterations. Each state stores strings representing the repetitions of x and y, as well as the next expected character for each of the strings in order to continue the repetition. That is how the states are checked and updated. The states are also stored in a dictionary that is keyed by the prefix they correspond to. Since a specific prefix can have multiple states (depending on x and y), each prefix has a list associated with it in the dictionary. Iterating through those lists allows for proper state updating. As the algorithm works through the sequence S and updates states, it checks to see if any updated state contains repetition strings long enough that a full x and y are present. If so, the algorithm returns a boolean true back through the function calls to let the main "Untangle" function know to

stop iterating as a solution has been found. This does not help in the worst case scenario, but does improve the average case.

Computational Complexity:

The nested for loops of the untangle function will give rise to $O(n^2)$ complexity, with n being the length of the input sequence of symbols, S. The calls to the newState, updateState, and, subsequently, checkNewState functions all execute in a constant amount of operations. CheckNewState utilizes a set in order to achieve fast lookups that are $O(1)$ on average in python. This ensures that searching the set doesn't add to the complexity of the algorithm. The same is true for when the check is done to see if that prefix and state already exist in the entries dictionary. Since the algorithm checks to see if it has found a possible solution, many interweavings will result in less than the $O(n^2)$ complexity due to an early termination.

The worst case scenario in terms of loop iterations and states stored would be when there is no interweaving present, many leading characters, and if x and y were very similar. No interweaving present guarantees that the nested loops will execute the maximum number of times. Leading characters (Although not known to be leading characters) are useless and require extra iterations. If x and y are very similar, it would cause more states to be created during each iteration as a new character could be allocated to the repetition of x or the repetition of y for a given state.

(b) [50 points] Implement your algorithm above and test its run time to verify your complexity analysis. Remember that CPU time is not a valid measure for testing run time. You must use something such as the number of comparisons.

To verify the complexity analysis, I implemented a counter that increments every time a unique substring of S was considered. This essentially tracks the amount of for loop iterations between the outer and inner loop. Even though the inner loop updates states and checks to see if the substring/state combination is unique for memoization purposes, that check only takes constant time due to set utilization. Every state update or creation also checks to see if the state contains long enough sequences to be a valid solution. This adds an extra check, but also runs in constant time. This check allows the total number of iterations to decrease depending on the input. Since the sequence S consists of only 1s and 0s, it limits the number of unique substrings that can exist for any length of substring.

Below are a few examples of input strings and signals with the number of iterations the algorithm ran:

S	x	y	Iterations
0010010 (len = 7)	101	0	19
00000000 (len = 8)	1	1	36
100010101 (len = 9)	101	0	25
100010101 (len = 9)	10001	101	45
10000100101010101 (len = 17)	00000	00000	153

As you can see, the number of iterations is greatly depending on the signals x and y. Overall, the iterations stays within a polynomial limit. If the state comparisons were added to the count, they would only add a multiple of n, not anything exponential.