Name: David Xie

ID: 40065595

Date: September 28, 2020

COMP352 – A1 Programming Part

Version 1

- Pseudocode:

```
Algorithm RevealStr(str, index, outputStream)
        Input: String str, starting at index, printed through PrintWriter outputStream
        Output: txt file of all possible String combinations of binary characters

        if index = str.length() then
                print str
                outputStream.println(str)

        else if str.charAt(index) = '*' then
                str0 ← str.substring(0, index) + '0' + str.substring(index + 1)
                RevealStr(str0, index + 1, outputStream)

                str1 ← str.substring(0, index) + '1' + str.substring(index+1)
                RevealStr(str1, index + 1, outputStream)

        else
                RevealStr(str, index + 1, outputStream)
```

- a) Algorithm details

This algorithm reveals all possible string combinations of a masked string of binary characters, recursively. To do so, it has a base case when the index is the same of the length of the string, in which case it will print the string to the console and to a txt file. If that is not the case and the character at the index is a '*', it will replace that mask to a '0' and call itself (recursion) and to a '1' and call itself again. For all other cases, the string is preserved and calls itself to continue verifying individual characters until it reaches the base case. Thus, recursion will reveal all possible combinations.

- a) Time complexity:

When analyzing running times, the algorithm grows exponentially, suggesting that in the worst-case scenario (all characters of the string are masked, i.e. '*') the time complexity is $O(2^n)$. This makes sense when analyzing the algorithm, as each time a masked character is encountered, there are two recursions. Assuming worst-case scenario, that means that every level of recursion, there will be 2, 4, 8, 16, … recursions. In other words, $2^1, 2^2, 2^3, 2^4, \dots, 2^n$. As this is the worst-case scenario, it will determine the time complexity, which concludes to be $O(2^n)$.

This solution is not acceptable as it will take a very long time to process large numbers of masked characters, if at all. For example, on Eclipse, the console stopped being able to run the algorithm at a reasonable time starting at 20 masked characters.

- a) Space complexity:

The space complexity can be estimated to be $O(n)$. That is because the space allocated to this algorithm is variable, as recursion varies depending on the String input. The recursion will require space allocation in a stack, with each recursion adding a new "sheet" onto that stack. The size of that stack will vary depending on the String input. As this is the only variable space, we can estimate the space complexity to be $O(n)$.

Version 2

- Pseudocode:

```
Algorithm RevealStr(str, outputStream)
        Input: String str, printed through PrintWriter outputStream
        Output: txt file of all possible String combinations of binary characters

        stack ← new Stack()
        stack.push(str)

        while !stack.empty() do
                current ← stack.pop()

                if (i ← current.indexOf('*')) != -1 then
                        for num ← '0' to '1' do
                                current ← current.substring(0, i) + num + current.substring(i + 1)
                                stack.push(current)

                else
                        print current
                        outputStream.println(current)
```

- a) Algorithm details

This algorithm reveals all possible string combinations of a masked string of binary characters, iteratively. This is done by implementing a stack and running it in a while loop. Each loop removes one element from the stack to examine through the if statement. Each time a '*' is encountered, this loop adds two new elements to the stack. If it does not encounter a '*', this loop prints the string to the console and to a txt file. Through iteration, it will thus print all the possible combinations strings of binary characters.

- a) Time complexity

Running time experiments show that this algorithm's growth is exponential. This seems to suggest that the time complexity of this algorithm is also $O(2^n)$. This makes sense when analyzing the algorithm. The worst-case scenario is once more when all characters of the string are masked. Every operation in the algorithm is constant except for the iterative while loop. This loop depends on the number of '*'. Each time a '*' is encountered, two new strings are added to the stack. The stack cannot be empty until all '*' are replaced. Thus, the while loop will be executed for 2, 4, 8,

16, … times, or in other words, $2^1, 2^2, 2^3, \ldots, 2^n$ times. This suggests that the time complexity of Version 2 can be estimated to $O(2^n)$.

- a) Space complexity

The space complexity is O(n). The stack object is of a variable size: it can contain a number of elements. This size depends on the number of masked characters of the string input. Thus, the space allocated to this algorithm is not constant. It varies depending on the input. As this is the only space variation to take into account, the space complexity can be estimated to O(n).

b) Compare complexities between versions

The complexities for both space and time are similar between versions 1 and 2. That is because design of the two are similar in that their operations both grow exponentially as the number of masked characters increase, and both use a "stack" to allocate variable space. Hence, the similarities between the two versions' complexities.