PROGRAMMING QUESTION V2

```
1    def UnHide(arr, writer):
2
3        INPUT:  arr - an array of characters
4            writer - to write the output to a .txt file
5
6    //Convert the char arr to a String by instantiating String
     object
7    stringToProcess = arr
8
9    //Create a stack of strings
10    stack = Stack<String>()
11
12    //Push the current string to process into the stack
13    stack.push(stringToProcess)
14
15    //Check if the stack is not empty
16    WHILE stack.empty() != TRUE
17
18        //Pop most recent entry
19        currString = stack.pop()
20
21        //Find the first occurence of a # in the string
22        hashIndex = currString.INDEXOF('#')
23
24        //If a hash exists
25        IF (hashIndex != -1) THEN
26
27            //Replace # with X and push then with O and push
28            FOR i ← 1 TO 2 DO
29
30                //Replace with X
31                IF (i == 1) THEN
32             currString = currString.REPLACE(hashIndex, "X")
33
34                //Replace with O
35                ELSE
36               currString = currString.REPLACE(hashIndex, "O")
37
38                ENDIF
39
40                //Push back into stack
41                stack.push(currString);
42
43            ENDFOR
44
45        ELSE
46            //If no #'s exist then print
47            writer.PRINT(currString)
```

~~~~~~~~~~~~~~~~~~ TIME ANALYSIS ~~~~~~~~~~~~~~~~~~

For this iterative implementation, the string is randomly generated
with another function called getRandomString(int numOfHash).

Below are the timings (in nanoseconds and seconds) of the iterative function UnHide when run with #'s from 2 to 24.

2 #'s
    Time Taken (in Nanoseconds): 266795
    Time Taken (in Seconds): 2.66795E-4
4 #'s
    Time Taken (in Nanoseconds): 364351
    Time Taken (in Seconds): 3.64351E-4
6 #'s
    Time Taken (in Nanoseconds): 1274339
    Time Taken (in Seconds): 0.001274339
8 #'s
    Time Taken (in Nanoseconds): 2700914
    Time Taken (in Seconds): 0.002700914
10 #'s
    Time Taken (in Nanoseconds): 6527061
    Time Taken (in Seconds): 0.006527061
12 #'s
    Time Taken (in Nanoseconds): 14920475
    Time Taken (in Seconds): 0.014920475
14 #'s
    Time Taken (in Nanoseconds): 50085044
    Time Taken (in Seconds): 0.050085044
16 #'s
    Time Taken (in Nanoseconds): 141357837
    Time Taken (in Seconds): 0.141357837
18 #'s
    Time Taken (in Nanoseconds): 294222430
    Time Taken (in Seconds): 0.29422243
20 #'s
    Time Taken (in Nanoseconds): 400380303
    Time Taken (in Seconds): 0.400380303
22 #'s
    Time Taken (in Nanoseconds): 1657680569
    Time Taken (in Seconds): 1.657680569
24 #'s
    Time Taken (in Nanoseconds): 7715609167
    Time Taken (in Seconds): 7.715609167
~~~~~~~~~~~~~~~~~~

Upon analysis of the iterative function we observe that the number of runs of each loop follows this pattern:

Note: Methods such as REPLACE and INDEXOF count for only O(n) in programming languages such as java, python etc. This complexity is dominated by that of the loop and so it isn't considered in the analysis below. We can rule it out at this point

WHILE loop: $2^{n+1}-1$ (2^(n+1) - 1)
FOR loop: $2^{n+1}-2$ (2^(n+1)-2)

Where n is the number of #'s

The above observation was made by running the iterative method with various values for n and recording the number of iterations of each loop.
Example:

n = 2
    WHILE loop = 7
    FOR loop = 6

n = 4
    WHILE loop = 31
    FOR loop = 30

n = 6
    WHILE loop = 127
    FOR loop = 126

n = 8
    WHILE loop = 511
    FOR loop = 510

Since the FOR loop is nested within the WHILE, the reason it runs $2^{n+1}-2$ (2^(n+1)-2) times is because of the WHILE loop. Thus making it the dominant loop in the algorithm.

When adding the two complexities and constants together we simplify it down and can conclude that the big O notation for the iterative method is:

$O(2^n)$ (O(2^n))

~~~~~~~~~~~~~~~~~~ SPACE ANALYSIS ~~~~~~~~~~~~~~~~~~

For the iterative algorithm variables are used which account for a constant amount of space.
The main contributor is a stack data structure that is used to push and pop values.

Since for every # found in the char array, 2 pushes and 1 pop is executed - If we have n hashtags then: 2n pushes and (-)1n pops are executed. See lines 19 and 41.

Summing the two gives us 2n + (-1n) = n
Therefore we can conclude that for n hashtags the maximum size a stack reaches is n

To conclude the **space complexity is O(n)**

~~~~~~~~~~~~~~~~~~ COMPARISONS ~~~~~~~~~~~~~~~~~~

When looking at both the UnHide methods we see that they both share the same time complexity of $O(2^n)$. Along with this they both share similar problems in that their time complexities are poor, and the file output size grows rapidly.

However, when looking at the timings of the iterative function as shown on page 2 we notice that the timings are significantly faster on the iterative method. This can be due to better efficiency when it comes to internal methods such as REPLACE and INDEXOF, or the absence of recursion which saves up computational resources.

If one of the two UnHide methods were to be picked for scalability the iterative method should be selected. Although it computes the UnHide method with the same time complexity its actual runtime speeds are faster.