PROGRAMMING QUESTION V1

```
1    def UnHide(arr, currIndex, writer):
2
3    INPUT: arr - an array of characters
4            currIndex -  the starting index of the array arr
5            writer - to write the output to a .txt file
6
7    //Checking for base case (checks if currIndex reaches start
     of the array arr)
8    IF currIndex == -1 THEN
9        writer.PRINT(arr)
10        return()
11    ENDIF
12
13   //Checks if current position is a #
14   IF arr[currIndex] == '#' THEN
15
16        //Replace # with an O
17        arr[currIndex] = 'O'
18
19        //Recursive call from currIndex-1
20        UnHide(arr, currIndex-1, writer)
21
22        //Replace the index changed on line 19 to an 'X'
23       arr[currIndex] = 'X'
24
25        //Recursive call from currIndex-1
26        UnHide(arr, currIndex-1, writer)
27
28        //Replace the index back with a # after changing it to
          X and O
29        arr[currIndex] = '#'
30
31    ELSE
32        //Proceed to next position if it isn't a #
33        UnHide(arr, currIndex-1, writer)
34    ENDIF
```

Zafir Khalid (40152164)

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

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

```
2 #'s
     Time Taken (in Nanoseconds): 16997
     Time Taken (in Seconds): 1.6997E-5
4 #'s
     Time Taken (in Nanoseconds): 42450
     Time Taken (in Seconds): 4.245E-5
6 #'s
     Time Taken (in Nanoseconds): 243417
     Time Taken (in Seconds): 2.43417E-4
8 #'s
     Time Taken (in Nanoseconds): 657828
     Time Taken (in Seconds): 6.57828E-4
10 #'s
     Time Taken (in Nanoseconds): 3075558
     Time Taken (in Seconds): 0.003075558
12 #'s
     Time Taken (in Nanoseconds): 5615948
     Time Taken (in Seconds): 0.005615948
14 #'s
     Time Taken (in Nanoseconds): 9271765
     Time Taken (in Seconds): 0.009271765
16 #'s
     Time Taken (in Nanoseconds): 26994311
     Time Taken (in Seconds): 0.026994311
18 #'s
     Time Taken (in Nanoseconds): 21770117
     Time Taken (in Seconds): 0.021770117
20 #'s
     Time Taken (in Nanoseconds): 108060295
     Time Taken (in Seconds): 0.108060295
22 #'s
     Time Taken (in Nanoseconds): 486568134
     Time Taken (in Seconds): 0.486568134
24 #'s
     Time Taken (in Nanoseconds): 2494622597
     Time Taken (in Seconds): 2.494622597
```

These timings are for a single run and may vary from different runs.
However, this still gives a good estimate as most values vary within a very
small range.

To find the time complexity of our algorithm we will refer to the psuedocode shown above.
Assuming we have the worst case input as all #'s (ie: "######.."), the recursive function will run with the following time function:

$T(n) = 2T(n-1) + C$

Where n = currIndex
Where C is some constant relating to the number of primitive calls.
T(n-1) is our recursive call and since we call it twice (once for O's and once for X's we have 2T(n-1) recursive calls)

$T(n) = 2T(n-1) + C$ --0
$T(n-1) = 2T(n-2) + C$ --1
$T(n-2) = 2T(n-3) + C$ --2

Sub 1 and 2 in 0

$T(n) = C + 2(C + 2(C + 2(T(n-3))))$
$T(n) = 7C + 8T(n-3)$

$T(n) = (2^3-1)C + 2^3T(n-3)$ //(2^3 -1)C + (2^3)T(n-3)
$T(n) = (2^k-1)C + 2^kT(n-k)$ //(2^k -1)C + (2^k)T(n-k)

Since the base case checks if currIndex = -1, it follows that for the base case:
n-k = -1
k = n+1

$T(n) = (2^{n+1}-1)C + 2^{n+1}T(n-(n+1))$ //(2^(n+1) -1)C + (2^(n+1))T(n-(n+1))
$T(n) = (2^{n+1}-1)C + 2^{n+1}T(1)$ //(2^(n+1) -1)C + (2^(n+1))T(1)

DROP CONSTANTS

$T(n) = 2^{n+1} + 2^{n+1}$ //(2^(n+1)) + (2^(n+1))
$T(n) = 2 * 2^{n+1}$ // 2 * (2^(n+1))
$T(n) = 2 * 2 * 2^n$ // 2 * 2 * (2^n))
$T(n) = 2^n$ //2^n

The time function T(n) of the recursive algorithm is $O(2^n)$ (O(2^n))

This time complexity is not acceptable or scalable.
Since its growth is of exponential order it takes alot of computational power for a single increase in the number of #s

With every increase in input size n, the time taken may roughly double. The number of combinations grows by 2^n (2^n) where n is the number of hashtags. With an exponential growth, processing 100 #'s may take roughly 2^{100} seconds. For each recursive call there may be upto 2 additional recursive calls, this is also a source of inefficieny in the algorithm as each call uses additional resources.

Moreover, another cause of inefficieny in the algorithm is writing to an output file. Line 9 of the algorithm causes great inefficiency due to the number of characters that need to be written to a file. In addition to this the size of the .txt file that is generated reaches the order of terabytes at ~30 hashtags. Managing this file size leads to further inefficiency as well.

Although the times are relatively short on page 2, as the UnHide method reaches ~30 #’s it slows down to an order of minutes and grows from then on.

Hence with these reasons, we can justify that this algorithm is not scalable or efficient.