**ECE 406**

**Teodor Mihai Tiuca**

**Assignment 2**



iii)

p: 1725583

q: 3503117

iv)

private\_key: 2417965565405

v)

message: 329415

encoded: 928896701129

vi)

decoded: 329415

1. Based on Master Theorem

Therefore:

1. , so
2. , so
3. , so

…

Therefore



Function code:

**indEqualsVal(A, index):**

if length(A) == 1:

return A[0] == index

if length(A) == 2:

return A[1] == index or A[0] == (index - 1)

center = floor(length(A) / 2)

index2 = index - ceil(length(A) / 2)

if A[center] == center:

return True

if A[center] > center:

return indEqualsVal(A[0: center], index2)

else:

return indEqualsVal(A[center+1:], index)

**indexEqualsValueWrapper(A):**

return indEqualsVal(A, length(A) - 1)

The base case is reached in levels. Since we are dividing in half every time, this means that . Since there is only one path taken by the algorithm, there is only 1 sub-problem at any level k. Therefore, . Finally, since the combination step is , .

According to Master Theorem,

Since, , or .

1. Since n is divided by 3 every time, base case is reached in levels, so . Since each execution calls itself 3 times, there are sub-problems at each level , so . Finally, it takes time to combine the results of each run, but each run prints “yes” n times. So, letting work done represent the number of times “yes” is printed, it is , . Since , according to Master Theorem,

Since the work counted is the number of times “yes” is printed, this means that it is printed times.

1. The sorting part of the algorithm designed sorts both arrays in time. To achieve this, the popular sorting function “merge sort” is used. The Pseudocode for this specific sorting algorithm can be found here <https://en.wikipedia.org/wiki/Merge_sort>.

function checkForMatchingValues(A, B):

arr = []

for i = 0 to n-1:

arr.push(A[i])

arr.push(B[i])

merge\_sort(arr)

for i = 0 to 2n-2:

if arr[i] == arr[i+1]:

return True

return False

There are 3 main parts: the first loop, merge sort, and the second loop.

The first loop runs n-1 times; therefore, it has runtime.

The second loop runs 2n-2 times; therefore, it has runtime.

The sorting algorithm recursively breaks the array down into 2 sub arrays, until each sub array is length 1. Sub problems are merged by iterating over each sorted sub array and pushing values in order to a larger array, then returning the larger sorted array.

Since arrays are split in 2, the base level is reached in levels, so . Since each sub problem spawns 2 more, at a given level k, there are sub problems, so . Finally, since each sub problem needs to iterate over 2 sorted arrays, runtime for merging is , so . Since , according to Master Theorem,

2. In this case, the swap() function swaps values at the two given indices.

**split(A, lo, hi):**

pivot = random A[i] where lo < i < hi

swap A[pivot] with A[hi]

i = lo - 1

for j = lo to hi – 1:

if A[j] < pivot:

i = i + 1

swap A[i] with A[j]

if A[hi] < A[i + 1]:

swap A[i + 1] with A[hi]

return i + 1

**quicksort(A, lo, hi):**

if lo < hi:

p = split (A, lo, hi)

quicksort(A, lo, p - 1 )

quicksort(A, p + 1, hi)

Quicksort definition and pseudocode template obtained from <https://en.wikipedia.org/wiki/Quicksort> . Edited to pick pivot as random. Pseudocode for the split() function is provided for completeness even though the question says it is given.

1. Quicksort splits the array into 2 unequally sized parts. The size of these parts depends on the partition value chosen. The closer this value is to the median value of the array, the more even the split. However, in the worst-case scenario, the array will only be split into sub arrays of size 1 and n-1. Therefore, it requires n steps to reach the base case in the worst-case scenario. Each sub problem makes a call to partition(), which iterates over the array form lo to hi. In the worst-case scenario, this iteration size scales with n since lo is the start index and hi is the end index. The time to combine the two sorted arrays is so it is negligible.

Therefore, n steps each loop n times, so the worst-case runtime of the quicksort algorithm is .

1. One call to the quicksort algorithm call quicksort() twice with varying parameters and loops over the array n times within the partition function. Therefore,

The runtime of each and depends on the value of q. This value is randomly chosen between 1 and n, with each q having an equal chance of being chosen. According to expected value theorem,

# of times a specific q is chosen = probability \* # of selections

Therefore, if the function is called n times,

# of times a specific q is chosen = = 1

Meaning that in n iterations, each q will be chosen once. The expected runtime can be calculated as the average of the total runtime of the n calls to quicksort().

1. Proving that if:

Then,

Using the definition of from part (iii) for the expected runtime is,

Since each value of q is used once, the first call’s parameter goes from 0🡪n-1 and the second call’s parameter goes from n-1🡪0. These are reciprocals of each other, so technically the function is called twice with each value from 0🡪n-1

Given ,

By shifting the limits of the summation, the equation can be modified as follows:

However, when , the value is 0. Therefore, the equation is:

Given ,

Converting each of these to Big O notation:

The dominant one is , therefore,