Data Structures and Algorithms Assignment 1

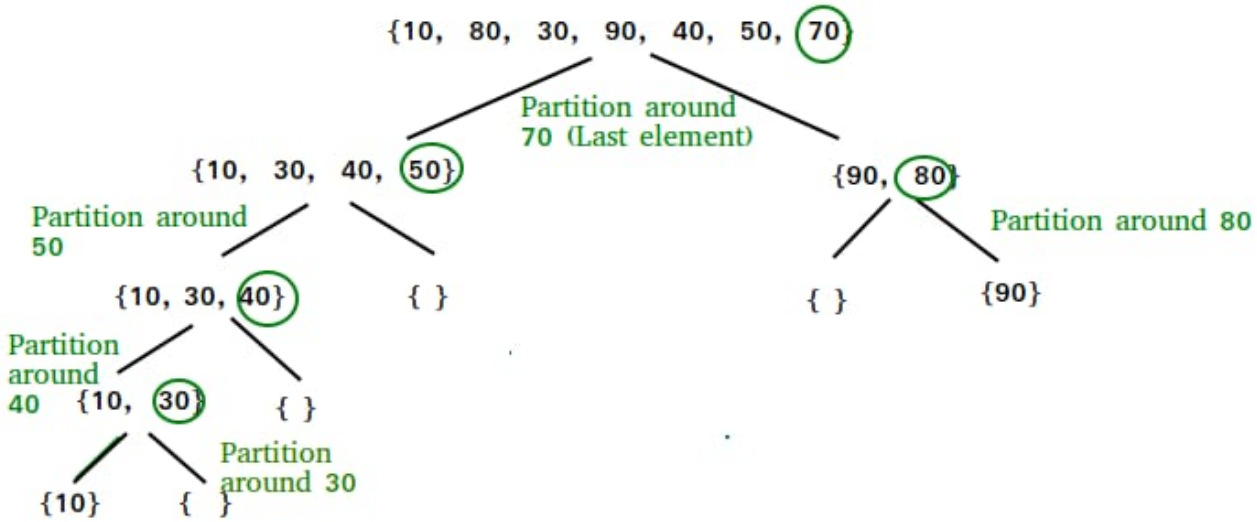
Minor
Akshay Iyer
190070006
h190070006 on hackerrank
Email id on hackerrank- 190070006@iitb.ac.in

I pledge on my honor that I have not given or received any unauthorized assistance on this assignment or any previous task.

Question-1 Stacking up on Quicksort

1. Why does it still consume stack space?

Each branch of the below diagram can take up a stack space only equal to its maximum depth as every time the function call exits a particular branch, it clears certain functions from the stack space. Thus maximum stack space will be the depth of a complete path from the topmost level to the end of a branch.
It still consumes stack space as even though we have caused branching in the sort space, there will still be cases when all the elements end up in a single branch of the sort space, for example when all the elements are arranged in descending order.



2. Proportional to number of items?
Irrespective of how the branches of the tree are formed due to the partitions created, in the worst case scenario, the maximum times a tree will be partitioned in the case where we are taking the lower half always, will be n. (This is when all the numbers are in descending order). Hence it is proportional to n as the partition function will be called a maximum of n times in one stack as the maximum depth of any one branch can only be n.

3. How much asymptotically?
Asymptotically the worst case size will be n itself in the unoptimised version. In my version as we are at least halving the stack space each time, the worst case stack size will be O(log n)

4. What is (asymptotically speaking) the worst case size of the stack in your version?
As every time we carry out the recursion function, we are only doing it in the smaller half, the total depth of the binary tree will always be less than log n.Hence we are dealing with a stack space of O(log n) in my worst case, which will be when the stack gets divided into exactly half at each step.

Observation:
The stack space decreased considerably by doing this. Also, as seen in the above diagram, irrespective of how we implement quick sort, every element is treated as a pivot once. Hence it would be best if every time it's treated as a pivot, it is compared to the least number of elements in the partition function. Hence we should keep searching in the lower sized half, as the pivots in those regions will be compared with less numbers, thus making it way faster. For example in the case where all the numbers are decreasing, we'll have to make n+n-1+n-2+……. Comparisons in the earlier case, but just 1+1+…..=n comparisons in my case.(as we check every pivot only with the last element). Plus it made sense to make it more

uniform and symmetric as always taking the lower half seemed kind of arbitrary.

This was a great question as it really cleared my concepts about stack space and time complexity of algorithms .

Compilation Successful
Input (stdin)
3 4 2 1 6 4 2
Your Output
1 2 2 3 4 4 6

Question-2 Go For EASY!!

This is a question that describes Catalan numbers. I couldn't find a very intuitive recursion formula myself. I researched Catalan numbers online, and found the recursion formula to them.
Even after seeing their recursion formula, I still wasn't fully convinced as the explanation that I had for their recursion formula, had repeated cases. So they had a different derivation, which I read through, and then implemented their recursion formula.
Here after finding out the recursion formula, the main problem was that the stack size was too massive for python to handle. It also took a lot of time. So instead of repeating the same function calls again and again, I just stored the net value returned by every function call in a list, when that particular function was called for the first time. Thus this prevented repeated unnecessary function calls, and the stack and time complexity of my algorithm greatly decreased.
Another problem was even after doing this my function just very slightly exceeded 1000 recursions which caused a runtime error when the input number was 999 or 1000. Hence to prevent this I pre loaded some of the initial values into my array as given in the sample inputs and outputs of the question, which then made my program run perfectly. I also removed some extra unnecessary

variables that I had introduced in my program to decrease the stack space taken by it.

Compilation Successful

Input (stdin)

```
7
```

Your Output

```
429
```

Compilation Successful

Input (stdin)

```
1000
```

Your Output

```
110961515
```

Question-3 Iterative Speaking, the Tower of Hanoi

I just implemented the algorithm discussed in class.
It also makes sense intuitively as we can move the smallest disc whenever we want, wherever we want, as it is always on top of one of the pole stacks. Hence moving the smallest disc twice is redundant as the same effect can be created by moving it in a single move. Hence the small disc needs to be moved in alternate steps only. Then we do the only remaining possible move to get the answer.
The important part here was to show when the base of the stack has been reached, so I initially stored a value 1000 in each of the three poles which signifies that the base of the pole stack.

Compilation Successful

Input (stdin)

```
3
```

Your Output

```
A B 1
A C 2
B C 1
A B 3
C A 1
C B 2
```

```
A B 1
```

Compilation Successful

Input (stdin)

```
5
```

Your Output

```
A B 1
A C 2
B C 1
A B 3
C A 1
C B 2
A B 1
A C 4
B C 1
B A 2
C A 1
B C 3
A B 1
A C 2
B C 1
A B 5
C A 1
C B 2
A B 1
C A 3
B C 1
B A 2
C A 1
C B 4
A B 1
A C 2
B C 1
A B 3
C A 1
C B 2
A B 1
```

Question-4 Is this Binary Search?

The name made me think of implementing a search algorithm similar to binary search. We will keep searching in the greater half as there will certainly be a maxima there.
I initially stored the xor values in an array and added an additional nested loop to compare the elements after each xor, which made my program take a lot more time than it should have. Hence I passed an argument to the function containing the value it has been xor'd with, and called the function as soon as every input xor number had been given, and before the next one was taken. Thus I did not store the xor values anywhere, instead xor'd them directly when comparing the numbers, and executed the function right there before taking the next xor input.

Compilation Successful

Input (stdin)

```
4 3
6 2 10 7
2
4
5
```

Your Output

```
3

3

3
```

Compilation Successful

Input (stdin)

```
4 3
10 2 6 7
2
4
5
```

Your Output

```
4

1

3
```