Saturday, February 5, 2022

Assignment 1

CSCI 411

1.

Considering n is an extremely large number close to infinity,

2 < 52! < n^(1/5) < ln (n^2) < ln^2 (n) < n < n ln(n) < (3/2)^n, log_2 ((4n)^n) < 2^n < n^3 < n!

2.

a)

```
A = list of real numbers
if (A.size is less than equal to 1)
        return

p = last element

index = low - 1
for (e in A not including last element) {
        if (e <= p) {
                append e in L
        } else {
                append e in R
        }
}
Result = func(L') + p + func(R')
return Result
```

b) O(n^2) is the worst-case asymptotic runtime considering the most unbalanced partitions possible, then the original call takes “cn” time for some constant c, the recursive call on “n - 1” elements takes “c(n - 1)” time, the recursive call Ono “n - 2” elements take “c(n - 2)” time, and so on.

c) Two recursive calls from L' and R': 2T
Both calls do the same amount of work resulting in: n/2
Loop iterates n times: O(n)

Hence, the recurrence relation is

$T(n) = O(1); \quad n <= 1,$
$\quad 2T(n/2) + O(n); \quad n > 1$

d) When the partitions are as evenly balanced as possible: their sizes either are equal or are within 1 of each other. The former case occurs if the subarray has an odd number of elements and the pivot is right in the middle after partitioning, and each partition has $(n - 1)/2$ elements. The latter case occurs if the subarray has an even number n of elements and one partition has $n/2$ elements with the other having $n/2 - 1$. In either of these cases, each partition has at most $n/2$ elements, and the tree of subproblem sizes looks a lot like the tree of subproblem sizes for merge sort, with the partitioning times looking like the merging times:

$T(n) = cn + 2c(n/2) + 4c(n/4) + \ldots + nc$, for some constant c

$= n \log_2 n$

Hence, the asymptotic runtime is $O(n \log_2 n)$

3.

a)

Considering the v as the nearest node to S,

Given,

if u has less than or same neighbors as v

Then, G does have a property of P with respect to s.

b)

```
let Q be queue.
Q.enqueue( s )
while ( Q is not empty)
        v  =  Q.dequeue( )
        for all neighbors w of v in Graph G
                if w is not visited
                        Q.enqueue( w )
                        If (u.NeighborSize less than equal to u.NeighborSize) {
                                return true;
                        }
                        return false;
```

c)

My pseudo code is correct because the code keeps track of the maximum number of neighbors for each level when traversing through the graph. We want to do this to see if the graph maintains the property P such that the maximum number of neighbors on a current level isn't greater than the previous level. It also meets the requirements $|N(u)| \leq |N(v)|$ and $d(s,u) > d(s,v)$. Hence, my pseudo code is correct.

d)

The time complexity of the is **O(V + E)** because we are using BFS, where V is the number of nodes and E is the number of edges.

4.

a) Use DFS, keep track of the color nodes, visit all the neighbors, if there is a target node increment the node, and at the end return the sizes of each colored nodes.

b) DFS(G, u)

```
u.visited = true
for each v ∈ G.Adj[u]
    if v.visited == false
        if (targetNode)
            size++
        DFS(G,v)
return size
```

c) My pseudo code is correct because the code keeps track of the target nodes, visit all the neighbors, and if there is a colored node increment the node, and at the end return the sizes of each colored nodes.
.

d) The time complexity of the is **O(V + E)** because we are using DFS, where V is the number of nodes and E is the number of edges.