Name: Girish Madnani
Student ID: 934130

Assignment 2: Analysis Problems

Problem 4. (Average-case complexity of Quicksort)

a) *Analysis: -*

Given, using Oracle algorithm pivot is chosen such that it lies in between top 25% to top 75% of the elements in the array. So, it is guaranteed that pivot divides the array in such a way that one side has at-least 1/4 elements.

```
randQuickSort(arr[], low, high)

1. If low >= high, then EXIT.

2. While pivot 'x' is not a Central Pivot.
  (i)   Choose uniformly at random a number from [low…high].
        Let the randomly picked number be x.
  (ii)  Count elements in arr[low…high] that are smaller
        than arr[x]. Let this count be sc.
  (iii) Count elements in arr[low…high] that are greater
        than arr[x]. Let this count be gc.
  (iv)  Let n = (high-low+1). If sc >= n/4 and
        gc >= n/4, then x is a pivot.

3. Partition arr[low..high] around the pivot x.

4. // Recur for smaller elements
   randQuickSort(arr, low, sc-1)

5. // Recur for greater elements
   randQuickSort(arr, high-gc+1, high)
```

How many times while loop runs before finding a pivot?

The probability that the randomly chosen element is pivot is 1/2.
Therefore, expected number of times the while loop runs are 2
Thus, the expected time complexity of step 2 is O(n).

What is overall Time Complexity in Worst Case?

In worst case, each partition divides array such that one side has n/4 elements and other side has 3n/4 elements. The worst case height of recursion tree is Log 3/4 n which is O(Log n).

$T(n) < T(n/4) + T(3n/4) + O(n)$
$T(n) < 2T(3n/4) + O(n)$

Solution of above recurrence is O(n Log n)

b) *Analysis: -*

If probability of success is p in every trial, then expected number of trials until success is 1/p

Proof: Let R be a random variable that indicates number of trials until success.

The expected value of R is sum of following infinite series
$E[R] = 1*p + 2*(1-p)*p + 3*(1-p)2*p + 4*(1-p)3*p + ...$

Taking 'p' out
$E[R] = p[1 + 2*(1-p) + 3*(1-p)2 + 4*(1-p)3 + ...]$ ---->(1)

Multiplying both sides with '(1-p)' and subtracting
$(1-p)*E[R] = p[1*(1-p) + 2*(1-p)2 + 3*(1-p)3 + ...]$ --->(2)

Subtracting (2) from (1), we get

$p*E[R] = p[1 + (1-p) + (1-p)2 + (1-p)3 + ...]$

Cancelling p from both sides
$E[R] = [1 + (1-p) + (1-p)2 + (1-p)3 + ...]$

Above is an infinite geometric progression with ratio (1-p).
Since (1-p) is less than, we can apply sum formula.
$E[R] = 1/[1 - (1-p)]$
$= 1/p$

C) *Analysis: -*

Using the above two points we get,

$T(n) < T(n/4) + T(3n/4) + O(n)$
$T(n) < 2T(3n/4) + O(n)$

Solution of above recurrence is O(n Log n) in average case .

Problem 5. (Lexicographical Optimisation with Paths)

```
function MinMaxWeightPath( 〈V,E〉 ,Vs, Ve)
    for each v ∈ V do
        dist[v] ← ∞
        prev[v] ← nil
    dist[Vs to Ve] ← ∞
    Set ← every set of every node in the graph
    while Set is non-empty do
        u ← sets with largest width
        delete u
        if dist[u] is ∞ then
            break
        w ← rest of Set
        for each w do
            alternate ← min(dist[w], max(dist[u], weight(u, w)))
            if alternate is less than dist[w] then
                dist[w] ← alternate
                prev[w] ← u
            path ← c
    return (w, dist[w])
```

Analysis: -

In this program first we define the unknown distance from every vertex to every other vertex and we define all previous vertex as undefined for every vertex. Then we define the distance from initial vertex to final vertex. We check for the largest distance if it exist, we remove the set and if it doesn't exist that means it is at the initial or same state. We use the rest of the set that wasn't removed and calculate the minimum maximum distance. The formula is the minimum distance of remaining sets and the maximum of distance of the set and its remaining paths. We then check if it is less than the distance of the remaining sets. If it is then the path is the minimum maximum distance and previous vertex is the next set.

Problem 6. (Weighted Graph Reduction)

```
function WeightedAPSP( 〈V,E〉 )
    Sets ← UnweightedAPSP( 〈V,E〉 )
    for i in len(Sets)
        if Sets.value() is equal to 1 then
            Sets.value() ← dist[Sets.keys()]
        else
            for i in (1 to N) do
                for j in (1 to N) do
                    if there is an edge from i to j then
                        dist[0][i][j] ← the length of the edge from i to j
                    else
                        dist[0][i][j] ← ∞

            for k in (1 to N)
                for i in (1 to N)
                    for j in (1 to N)
                        dist[k][i][j] ← min(dist[k-1][i][j], dist[k-1][i][k] +
                        dist[k-1][k][j])
            Sets.value() ← dist[k][i][j]
    return Sets
```

Analysis: -

In this algorithm, we take the sets and its values from the UnweightedAPSP algorithm. We then check if the value is 1 or more than 1. If it is 1 we take the weight of the set and overwrite it as the value. If it is more than 1, we use Floyd Warshall Algorithm to find the shortest distance of the elements in the set and put it as the value.

Problem 7.a. (Heap Algorithm Analysis)

```
function buildmaxheap(a[], n)
    heapsize ← n
    for j ∈ (n/2 to 0) do
        maxheapify(a, j, heapsize)
    return a

function maxheapify(a[], i , heapsize)
    left ← 2*i+1
    right ← (2*i) + 2
    if left is greater than or equal to heapsize then
        return
    else
        if left less than heapsize and a[left] greater a[i] then
            largest ← left
        else
            largest ← i
        if right less than heapsize and a[right] greater a[largest] then
            largest ← right
        if largest not equal to i then
            temp ← a[i]
            a[i] ← a[largest]
            a[largest] ← temp
            maxheapify(a, largest, heapsize)

function problem_1_a()
    allocate memory for 'a' pointer
    allocate memory for 'm' pointer
    input n
    for i in n do
        input a[i]
    m ← buildmaxheap(a, n)
    for t in n do
        output m[t]
    return
```

Analysis: -

In this algorithm we build a heap of n times and it is then check if its larger or small than the parent values.
Hence its runs at O(n) time.

Problem 7.b. (Right-Handed Heap Algorithm Analysis)

```
function buildmaxheap(a[], n)
    heapsize ← n
    for j ∈ (n/2 to 0) do
        maxheapify(a, j, heapsize)
    return a

function maxheapify(a[], i , heapsize)
    left ← 2*i+1
     right ← (2*i) + 2
     if left is greater than or equal to heapsize then
        return
     else
        if left less than heapsize and a[left] greater a[i] then
            largest ← left
        else
             largest ← i
        if right less than heapsize and a[right] greater a[largest] then
            largest ← right
        if largest not equal to i then
            temp ← a[i]
            a[i] ← a[largest]
            a[largest] ← temp

            maxheapify(a, largest, heapsize)

function buildmaxheap2(a[], n)
    heapsize ← n
    for j ∈ (0 to n/2) do
        maxheapify2(a, j, heapsize)
    return a

function maxheapify2(a[], i , heapsize)
    left ← 2*i+1
    right ← (2*i) + 2
    if a[left] is greater a[right] then
        temp ← a[right]
        a[right] ← a[left]
        a[left] ← temp
        maxheapify(a, left, heapsize)


function problem_1_b()
    allocate memory for 'a' pointer
    allocate memory for 'm' pointer
    input n
    for i in n do
        input a[i]
```

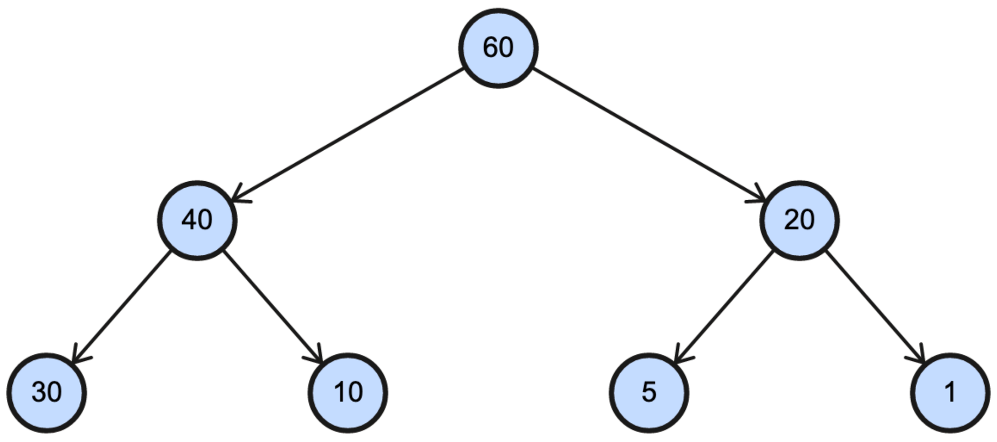
```
m ← buildmaxheap(a, n)
m ← buildmaxheap(m, n)
for t in n do
    output m[t]
return
```

Analysis: -

Similarly, to 7a. to build a heap of n times and it is then check if its larger or small than the parent values.
Hence its runs at O(n) time.

Problem 8. (Heap Top-k)

Professor dubious's argument are mostly correct other than the point where the top 3 elements aren't in the top 2 roots on the 1st element. The right root of the 1st element will indeed be smaller than the 1st element but may or may not be bigger than the 4th element of the heap as it is joined to the left root of the 1st element that is it is joined to the 2nd. This is also true for further down the heap tree



From the image we see that the 1st elements ('60') children are 40 (2nd element) and 20 (3rd element). The 2nd element ('40') children are 30 (4th element) and 10 (5th element). So, we see that the 4th element ('30') is bigger than the 3rd element ('20'). And we also see that the bigger element between the 2nd and 3rd which is the 2nd element is placed on the left side of the tree. Hence, we see that the top 3 elements aren't in the top 2 layers.