Assignment-02

1) Generally, the Quicksort uses an element as a Pivot to sort the array recursively. In general element at index which is at last is used as Pivot. The pseudo code for quick sort is as follows:

```
Quicksort (A,P,r)
 if P<r
     q= Partition(A,P,r)
     Quicksort(A,P,q-1)
     Quicksort(A,q+1,r).

Partition (A,P,r)
  x=A[r]
  i=P-1
  for J= P to r-1
      if A[J]≤x
        i=i+1
        exchange A[i] and A[J]
  exchange A[i+1] with A[r]
  return i+1
```

In the worst case the partitioning happens leaving (n-1) elements on one side and 0 elements on the other. The worst case scenario is when Array is in Descending order. The pivot will have all elements which are greater and 0 elements which are lesser. The recursive call of Array with 0 elements returns, T(0)=1. The rest for Partition costs $\Theta(n)$ times.

So,

$T(n) = T(n-1) + T(0) + \Theta(n)$

$T(n) = T(n-1) + cn$ $\quad \because T(n-1) = T(n-2) + c(n-1)$

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

Similarly after k times,

$T(n) = T(n-k) + c(n-(k-1)) + \dots + c(n-1) + c(n)$

When,

$n-k=0 \therefore n=k.$

it is the base case,

$T(n) = T(0) + c + 2c + \dots + c(n-1) + c(n)$

$= T(0) + c(1+2+3\dots n)$

$T(n) = T(0) + c\,\frac{n(n+1)}{2}$

So,

$\boxed{T(n) = \Theta(n^2)}$

So,

$\boxed{T(n^2)}$ is the worst case scenario for quicksort.

When the array is in descending order.

2) a) Quicksort'(A, P, r)

While p < r

do $q = \text{partition}(A, P, r)$

Quicksort(A, P, q-1).

$p = q + 1$

2A)

The above quicksort algorithm correctly sorts the Array.

To explain Consider an Array A and we have run Quick sort in it with call Quick sort (A, 1, length[A]).

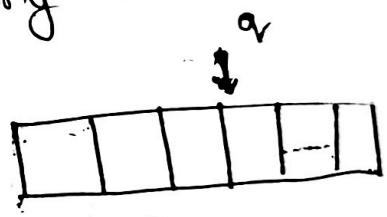
So,

from the code.

till p<r this algorithm runs without terminating.

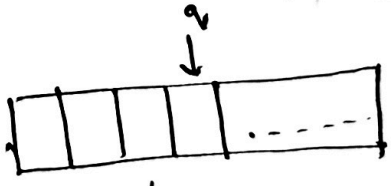
Consider for first run the algorithm gets a value of partition which is q from the following call.

→ `q = Partition(A, 1, length[A])`



Then this recursively calls quick sort for elements before pivot q.

→ `Quick sort(A, p, q-1)`



elements before q will be sorted.

When $p=q+1$, the elements to the right of q will be sorted while $p<r$ and tail recursion will execute for this part of the Array.

2)b)

If the elements of Array is already sorted, the pivot will be present in the right position which in turn results in 0 elements on the right side and n-1 elements on the leftside. The recursive calls happens to these n-1 elements while $p<r$. In this scenario we have $\Theta(n)$ Stack depth.

2) c). modified QuickSort (A, P, r)

```
while p<r
    q = partition (A,P,r)
    if q< floor( (P+r)/2)
        modified QuickSort (A, P, q-1)
        P = q+1
    else
        modified QuickSort (A, q+1, r)
        r = q-1.
```

In previous version of algorithm Quicksort we are always executing the tail recursion for elements after q (which is pivot). If the block of elements to right of q are more they will take more stack spaces in the memory.

In order to prevent this we want the smaller part of the array after split to execute rather than larger part to execute and call recursive function.

If we see in the above code (Modified Quicksort) we are calling Quicksort recursively if $q < \left(\frac{p+r}{2}\right)$ on the part modified Quicksort (A, p, q-1). and if $q > \left(\frac{p+r}{2}\right)$ Quicksort (A, q+1, r) in vice versa.

TO Conclude, we modified the algorithm to do tailrecursion always on the larger portion / partition.

As we are always executing the smaller portions first the stack space consumed can be atmost half of the array size. That means we can't push more than $\log n$ calls on stack. $\Rightarrow$ $\boxed{\Theta(\log n)}$

3)

Hoarie Partition Correctness :-

```
HOARE-PARTITION(A,p,r)
1) x ← A[p]
2) i ← p-1
3) j ← r+1
4) while TRUE
5)     do repeat j ← j-1
6)        until A[j] ≤ x
7)        repeat i ← i+1
8)        until A[i] ≥ x
9)        if i < j
10)       then exchange A[i] ↔ A[j]
11)       else return j.
```

3)a)

A = { 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 }

index = 1 2 3 4 5 6 7 8 9 10 11 12

Let Assume the Call : HOARE-PARTITION(A, 1, 12)

p = 1

r = 12,

So, x = A[p] = A[1] = 13.

Initial Case :-

x = 13,

i = Not Assigned

j = Not Assigned.

Array:	13	19	9	5	12	8	7	4	11	2	6	21
index:	1	2	3	4	5	6	7	8	9	10	11	12

After 1st Loop :-

x = 13 ,

i = 1
J = 11 (We swapped elements at 1 & 11 as i < J) from previous arrangement

Array:	6	19	9	5	12	8	7	4	11	2	13	21
Index:	1	2	3	4	5	6	7	8	9	10	11	12

After 2nd Loop:-

x = 13,
i = 2
J = 10 (we swapped elements at 2 & 10 as i < J) from previous arrangement.

Array:	6	2	9	5	12	8	7	4	11	19	13	21
Index:	1	2	3	4	5	6	7	8	9	10	11	12

After 3rd Loop:-

x = 13 [As i > J we have returned J]
i = 10
J = 9

Array:	6	2	9	5	12	8	7	4	11	19	13	21
Index:	1	2	3	4	5	6	7	8	9	10	11	12

The Algorithm will return 9 for the partition in this example
Moreover if we observe all the elements left to index 9 are less than 13.

3) b) The indexes i and J are initialized in the algorithm, such that the elements only in the subarray A[p.....r] and we never access an element outside A. This is because

We have initiated i as p-1 and J as r+1. As we go along the while loop we will increment i and decrease J. So, during the execution lets assume when, we found no element such that A[J] ≤ x J will not change and its value will be r, then assume i will increase and reaches value of J. Then programme terminates and returns J as i<J does not hold true. One more scenario where J is decreasing and i increased such that i>J the program terminates and returns J.

To conclude as we are initiating i & J in bounds of sub array A we won't go out of bound as we are terminating our program when i<J.

3) c)

As we can see from line 11 when i becomes greater than J i.e, when i>J it will return J.

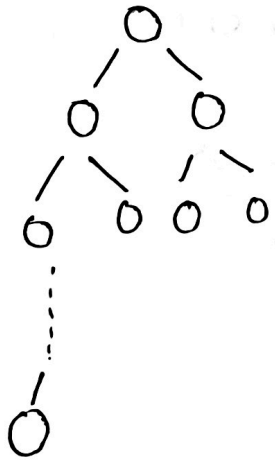
As we have initialized J as r+1 and decrement it, at most it can go till p as it is the lowest bound of the sub array, so when i>J it will return J whose range p≤J<r. When i is less than J usually it swaps the elements. At the end when program terminates all the elements before index J i.e, p≤J<r are less than A[p] which is pivot and element to the right of index J are greater.

3)d)

As we are initiating i and incrementing when we find any element less than pivot. We are actually checking whether the elements before A[p] are less than A[p] if they are not small we are swapping with element at j. Similarly when in case of j we are decreasing j from r+1 and we are decreasing only if $A[j] \geq A[p]$ which in turn gets swapped to the left side of pivot partition for $i \geq j$. When $i \geq j$ then the elements on the left side are all less than A[p] i.e, A[p...j] and all elements in A[j+1....r] are greater than these elements.

4)

The heap will have minimum elements when the last level has 1 node.



Level	Number of nodes
0	2^0
1	2^1
2	2^2
⋮	⋮
h-1	2^{h-1}
1	1

The number of nodes = $\underbrace{2^0}_{1} + \underbrace{2^1 + 2^2 + \cdots 2^{h-1}}_{h-1 \text{ elements}} + 1$

The series is in G.P with $n = h$, $r = 2$, $a = 1$

Sum of terms in G.P = $\frac{a(r^n-1)}{r-1}$

$= \frac{1(2^n-1)}{2-1}$

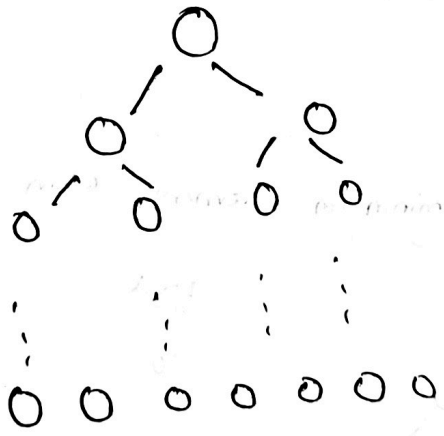
$= 2^n - 1$

Substituting this back,

So,
number of nodes $= 2^h - 1 + 1$

Number of nodes $= 2^h$

→ Maximum number of nodes are present if the number of nodes at last level is full.



Level	N. of nodes
0	2^0
1	2^1
2	2^2
⋮	⋮
h	2^h

Number of nodes = $2^0 + 2^1 + \cdots 2^h$

1 element h

The terms are in G.P.

$a = 1$

$r = 2$

$n = h+1$

Sum of term in G.P

$= \frac{1(2^{h+1}-1)}{(2-1)}$

substituting it back,

Number of nodes = $2^{h+1} - 1$

5) The Max Heapify algorithm that uses an iterative structure instead of recursion:

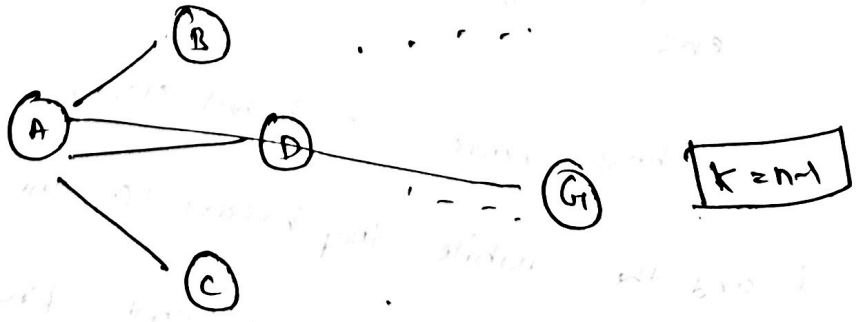
Pseudocode:-

```
MAX-HEAPIFY(A,i)
  while (true)
      l = left(i)
      r = Right(i)
      if l ≤ A.heapsize and A[l] > A[i]
          largest = l
      else largest = i
      if r ≤ A.heapsize and A[r] > A[largest]
          largest = r
      if largest = i
          break
      exchange A[i] with A[largest]
      i = largest
  end
```

→ If we have node as highest element then largest will be i and the while loop breaks (or) the root will get exchanged by the largest element present on the leftside (or) rightside of the node iteratively.

6) The smallest possible depth of a leaf in a decision tree will be n-1. Assume this scenario like insertion sort if we have a sorted array then the number of comparisons will be n-1 and the algorithm terminates after n-1 comparisons.

To speak probabilistically assume an element in array A which is A[i] and making comparison with other elements in array. If we assume the comparisons like a graph with n nodes, the minimum number of paths(k) to compare each element with the current one will be n-1 as. We cannot commit graph with $k < n-1$.



Graph for comparisons