CS-6363: ASSIGNMENT-1

Q.1>

a) $\Theta(\log n)$

b) $\Theta(n^2 \log n)$

c) Recursive Alg checks if there are any duplicate elements in the array.

Q.2>

(a). $\Theta(\sqrt{n})$

(b). $\Theta(2^n)$

(c). $\Theta(2^n)$

(d). $\Theta(n \log n)$

Q.3> Asymptotically Smallest to largest:

- $\log^{201} n$
- $1 + \lg\lg\lg n$
- $12 + \lfloor \lg\lg n \rfloor$
- $\lg^* n$
- $(\lg^*)^{\log n}$
- $\lg^*(2^{2^{2n}})$
- $n^{1/\log(\log n)}$
- $n \log^4 n$
- $\cos n + 2 \rightarrow (\log n)^{\log n}$
-
- $\sqrt{\log n}$
- $\log n$
- $n^{1/125}$
- $n^{\log(\log n)}$

- $n, 2^{\log n}$
- $(\sqrt{n})^{e}$
- $n^{3/2} \log n$
- $\sum_{i=1}^{n} i$
- $n^{2.1}$
- $4^{2 \log n}$
- $\sum_{i=1}^{n} i^2$
- $n^{7/2n}$
- n^5
- $\left(1 + \frac{1}{154}\right)^{15n}$

Q.4> Recurrence relation:

$$P(n) = P(n-1) + P(n-2) \quad n \geq 2$$

Base cases: $P(0) = 1$

$P(1) = 1$ ($\because$ when there is only one square, it can only be filled with a motorcycle)

Q.5>

(a). $T(n) = 2T(n/2) + n^4$

$a = 2, b = 2, c = 1, d = 4$

$b^d = 2^4 = 16$

$a = 2 < b^d$

$\therefore T(n) = O(n^4)$

Master theorem:

$f(n) = af(n/b) + cn^d$

where $a \geq 1$

$b > 1$

$c > 0, d \geq 0$

$f(n)$ is $\begin{cases} O(n^d), & \text{if } a < b^d \\ O(n^d \log n), & \text{if } a = b^d \\ O(n^{\log_b a}), & \text{if } a > b^d \end{cases}$

(b). $T(n) = 16T(n/4) + n^2$

$a=16,\ b=4,\ c=1,\ d=2$

$b^d = 4^2 = 16$

$a = 16 = b^d$

$\therefore T(n) = O(n^2 \log n)$

(c). $2T(n/3) + T(n/4) + n$

$a=2$

$b=3$

$c=1$

$d=1$

$b^d = 3^1 = 3$

$a = 2 < b^d$

$\therefore T(n) = O(n)$

(e). $T(n) = 3T(n/2) + 5n$

$a=3,\ b=2,\ c=5$

$d=1$

$a=3$

$b^d = 2^1 = 2$

$a > b^d$

$T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

Algorithm Peak_find (A[1...n])

```
{
  low = 0
  high = n-1
  While (low < high)
  {
     mid = low + (high-low)/2
     if (A[mid] < A[mid+1])
         low = mid + 1
     else
         high = mid
  }
  return A[low]
}
```

1). The time complexity of the algorithm is $O(\log n)$

2). The algorithm splits the array/subarray which it is currently scanning into 2 parts and based on comparisons then looks into only one of these 2 parts.

3) This process is continued until it rounds down to a single element which is the peak.

Correctness :

1) The algorithm splits the array elements only if $n > 1$ (or, low < high)

2> It then computes mid = low + (high-low)/2

Case-1: If A[mid] < A[mid+1], it implies the A[mid] cannot be the peak and therefore, the peak must be in the right part of the subarray.

Case-2: If A[mid] >= A[mid+1], it implies that A[mid] may or may not be the peak. The peak must thus, lie in the left portion of the current subarray.

3> When a single element is left, the index low is returned. By the principle of mathematical induction, the algorithm works correctly.

Running time:

Since the portion of the array it is looking at reduces by half in each iteration of the while loop, it runs in $O(\log n)$ time.
The comparisons inside and outside the while loop take $O(1)$ time and nothing is done at the end except return the (low) index of the peak element.

```
Algorithm   Count_inversions (A[1...n])
{
  if (n>1)
  {
    mid= ⌊n/2⌋
    return Count_inversions (A[1...mid]) + Count_inversions
                                            (A[mid+1...n])
              + Count (A[1...n], mid)

  }else return 0
}
Procedure Count (A[1...n], mid)
{
    i=1
    j=mid+1
    for k=1 to n
    {
        if (A[i] <= A[j])
            i= i+1

        else if (A[j] < A[i])
            j= j+1
            count=count+1;
        else if (i> mid)
            i=i+1

        else
            j=j+1
    }
    return count
}
```

1) We use a modified versions of the Merge-Sort.

2) At each stage, We divide the array recursively into 2 parts if (n>1).

3) We count the inversions in each subarray and add them up.

4) When we reach the bottom and the recursive calls begin to return, we add to the count the number of inversions across the 2 subarrays.

Running time:

1) Since this is a divide-and-conquer approach, the array is divided into 2 subproblems at each stage.

2) The count procedure uses the MERGE procedure's advantages of at most n comparisons.

3) Hence the complexity would be:

logn stages

Atmost n comparisons at each stage.

$T(n) = O(n\log n)$