Name: Yash Santosh Sawtare.
Roll No: 21102A0070
Sub: AOA
Div: CMPN A

Homework Assignment 5

PAGE No.
DATE / /

Q.1

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

$= T(n/2) + \frac{n}{2} + n$

$= T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + \frac{n}{2} + \frac{n}{2^0}$

$\vdots$

$= T\left(\frac{n}{2^k}\right) + \left[\frac{n}{2^{k-1}} + \cdots + \frac{n}{2} + \frac{n}{2^0}\right]$

$= T\left(\frac{n}{2^k}\right) + n\left(\frac{1-(1/2)^k}{1-1/2}\right)$

$= T\left(\frac{n}{2^k}\right) + 2n\left(1-\left(\frac{1}{2}\right)^k\right)$

Now, recursion bottoms out when $\frac{n}{2^k} = 2$

$\therefore k = \log\left(\frac{n}{2}\right)$

When this happens we get,

$T(n) = T(2) + 2n\left(1 - \frac{1}{2^{\log n/2}}\right)$

$= T(2) + 2n\left(1 - \frac{2}{n}\right)$

$= T(2) - 4 + 2n$

$T(2)$ will take $O(1)$ time i.e constant say 1.

$\therefore T(n) = 2n - 3$

PAGE No.
DATE / /

For upper bound:
we need to show that there exist c, n_0 such that $0 \leq T(n) \leq c \cdot g(n) \quad \forall n \geq n_0$.

$\therefore$ Here $g(n) = n$

$\therefore \quad 2n - 3 \leq cn$

setting $c = 3$ and $n_0 = 1$ proves the upper bound.

$\therefore \boxed{T(n) = O(n)}$

For lower bound:
we need to show that there exist c, n_0 such that $0 \leq c \cdot g(n) \leq T(n) \quad \forall n \geq n_0$.

$\therefore \quad 2n - 3 \geq cn$

This is same as showing

$$(2-c)n \geq 3$$

setting $c = 1$ and $n_0 = 4$ proves the lower bound.

$\therefore \boxed{T(n) = \Omega(n)}$

Hence $\boxed{T(n) = \Theta(n)}$

b. $T(n) = 2T(n/2) + 2n$

$= 2\left[2T\left(\frac{n}{2^2}\right) + n\right] + 2n$

$= 2^2 T\left(\frac{n}{2^2}\right) + 2(2n)$

$= 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2}\right] + 2(2n)$

$= 2^3 T\left(\frac{n}{2^3}\right) + 3(2n)$

$\vdots$

$= 2^k T\left(\frac{n}{2^k}\right) + k(2n)$

Recursion bottoms out when $\frac{n}{2^k} = 2$ ...

i.e $k = \log(n/2)$

when this happens, we get

$T(n) = 2^{\log(n/2)} T\left(\frac{n}{2^{\log(n/2)}}\right) + \log\left(\frac{n}{2}\right) \times (2n)$

$= \frac{n}{2} T(2) + 2n \log\left(\frac{n}{2}\right)$

$T(2)$ will take $O(1)$ time i.e constant say 1.

$T(n) = 2n \log\left(\frac{n}{2}\right) + \frac{n}{2} - 2n$

$T(n) = 2n \log\left(\frac{n}{2}\right) - \frac{3n}{2}$

PAGE No.
DATE / /

For upper bound:

$T(n) \leq c \cdot g(n) \quad \forall n \geq n_0$ for some c and n_0

here $g(n) = n\log(n)$

$\therefore \; en\log(n) - \frac{3n}{2} \leq c \cdot n\log(n)$

setting $c = 3$ and $n_0 = 2$ proves the upper bound.

~~$T(n) = O(n)$~~ $\quad \boxed{T_n = O(n\log(n))}$

For lower bound:

$T(n) \geq c \cdot g(n) \quad \forall n \geq n_0$ for some c and n_0

here $g(n) = n\log(n)$

$$n\log(n) - \frac{3n}{2} \geq c \cdot n\log(n)$$

this will be same as showing.

$$n\log(n)\left[2 - c\right] \geq \frac{3n}{2}$$

setting $c = \frac{3}{2}$ and $n_0 = 2^{15}$ proves the lower bound.

$$\boxed{T(n) = \Omega(n\log(n))}$$

Hence, $T(n) = \Theta(n\log(n))$

PAGE No.
DATE / /

c. $T(n) = T(\sqrt{n}) + 1$

→ $\therefore T(n) = T(n^{1/2}) + 1$

$= T(n^{1/2^2}) + 2$

$= T(n^{1/2^3}) + 3$

⋮

$T(n) = T(n^{1/2^k}) + k$

recursion bottoms out when, $n^{1/2^k} = 2$.

$\therefore k = \log_2(\log_2(n))$

when this happens, we get.

$T(n) = T(2) + \log_2(\log_2(n))$

$T(2)$ will take ~~cons~~ $O(1)$ time i.e constant say 1

$\therefore T(n) = 1 + \log_2(\log_2(n))$

Now,

for upper bound:

$T(n) \leq c \cdot g(n) \quad \forall n \geq n_0$ for some c and n_0

here

$g(n) = \log_2(\log_2(n))$

$\therefore 1 + \log_2(\log_2(n)) \leq c \cdot \log_2(\log_2(n))$

setting $c = 4$ and $n_0 = 4$ proves the upper bound.

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

PAGE No.
DATE / /

For lower bound:

$T(n) \geq c \cdot g(n) \quad \forall n \geq n_0$ for some c and n_0.

here $g(n) = \log_2 \log_2 (n)$

$$1 + \log_2(\log_2(n)) \geq c \log_2 \log_2 (n)$$
$$(1-c) \log_2(\log_2(n)) \geq -1$$

setting $c = \frac{1}{2}$ and $n_0 =$ proves the lower bound

$$\therefore T(n) = \Omega(\log_2(\log_2(n)))$$

Hence, $T(n) = \Theta(\log_2(\log_2(n)))$

d.

$$T(n) = 2T(n-1) + 1$$
$$= 2[2T(n-2)+1] + 1$$
$$= 2^2 T(n-2) + 2 + 2^0$$
$$= 2^2[2T(n-3)+1] + 2^1 + 2^0$$
$$= 2^3 T(n-3) + 2^2 + 2^1 + 2^0$$
$$\vdots$$
$$= 2^k T(n-k) + [2^{k-1} + \cdots + 2^1 + 2^0]$$

$$T(n) = 2^k T(n-k) + (2^k - 1)$$

Recursion bottoms out when $n-k = 2$

$\therefore \quad n = k+2 \quad \therefore \quad k = n-2$

when this happens, we get

$$T(n) = 2^{n-2} T(2) + 2^{n-2} - 1$$
$$= \frac{2^n T(2)}{4} + \frac{2^n}{4} - 1$$

$T(2)$ will take $O(1)$ time i.e constant say 1

$$\therefore T(n) = \frac{2^n}{4} + \frac{2^n}{4} - 1$$

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

For upper bound:

$T(n) \leq c.g(n) \quad \forall n \geq n_0$ for some c & n_0.

$$\therefore \frac{2^n}{2} - 1 \leq c.2^n$$

setting $c=2$ $n=2$ proves the upper bound.

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

PAGE No.
DATE / /

For lower bound:

$$T(n) \geq c \cdot g(n)$$

$$\frac{2^n}{2} - 1 \geq c \cdot 2^n$$

This will be same as showing

$$2^n \left(\frac{1}{2} - c\right) \geq 1$$

setting $c = \frac{1}{4}$ and $n_0 = 4$ proves the lower bound.

$$\therefore T(n) = \Omega(2^n)$$

Hence $T(n) = \Theta(2^n)$

Q.2

a) Algorithm:

```
1. int S[n], i, k;
2. mergesort ( S[n] )
3. for( i=0 ; i<n ; i++)
4.       k = BinarySearch ( S, z - S[i] )
5.       if ( k ≠ -1 && k ≠ i)
6.             return true.
7. return False
```

In above algorithm mergesort will take $\Theta(n \log n)$ and for loop will run for max n times.
and Binary search will take $\Theta(\log n)$ time
∴ sum of all above we will get

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

b) S ← sorted - (Given)

Algorithm:

```
1. int *q = 0 , int *e = n-1 , int sum
2. while (q<e)
3. { sum = (S[q] + S[e])
4.    if (sum == z)
            return true;
5.    if (sum < z)
            q++      // increase start pointer
6.    if ( sum > z)
            e--      // decrese end pointer
      if ( e>s)
7.          return false;
      }
```

above algorithm can take maximum n iterations
∴ it will take O(n) time.

Q.3)

Algorithm - 1:

Given → A[1...n] is sorted array.

```
1. int s = A[A.length - 1];
2. int check[s] = {0};
3. for( int i=0; i<n; i++)
   {
       check[A[i]]++;
   }
4. Merge sort (check, s)
5. if ( Binary Search (c, s, n/2) == 1)
   {
       // we will make Binary search return
          "1" if the is element greter than
          n/2 in c. orthurwise "0".

       printf( Majority element is present)
   }
6. else
       printf( Majority element is not present)
```

In above algorithm for loop will take max O(n) iterations, Merge sort will take O(n log(n)) time, and binary search will take O(log(n)) time and rest will take constant time.

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

$$\boxed{T(n) = O(n \log(n))}$$

PAGE No.
DATE / /

Better Algorithm.

Algorithm-2:

```
1. int count=0, majorityelement=0; int s = n/2;
2. for (int i=0; i<n; i++)
   {
      if (count==0)
         majorityelement = A[i];
      if ( A[i] == majorityelement);
         count = count+1;
      else
         count = count-1;
   }
3. count=0; // assign count=0;
4. for (int i=0; i<n; i++)
   {
      if (majorityelement == A[i])
      {
         count = count+1;
      }
   }
   if (count > s)
      return true;
   else
5.    return false.
```

In above algorithm, 1st for loop will run for maximum n time and so does the 2nd. and rest will take c time.

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

$\therefore \boxed{T(n) = O(n)}$