Assignment - 1

Neelansh Bhardwaj M.L 45

A-1) i) Big - Oh (O):

$f(n) = O(g(n))$

$g(n)$ is "tight" upper bound of $f(n)$

$\therefore f(n) <= (g(n))$

$\forall n \geq n_0$, some constant $c > 0$

Asymptotic Notations:

Asymptotic → towards infinity

- While defining complexity of our algo., we will use asymptotic notations, assuming our input size is very large.

Time Complexity	Space Complexity
No. of instructions to carry out an algorithm	Extra space req. by an algorithm except input

- While calculating complexities:
  - Constraints are ignored
  - Lower order terms are ignored in add" or subtract"
  - Take the highest order term.

ii) Big Omega (Ω):

$f(n) = \Omega(g(n))$

$g(n)$ is "tight" lower bound of $f(n)$

$\therefore f(n) \geq c \cdot g(n)$

$\forall n \geq n_0$ & $c > 0$

iii) Theta (Θ): Gives "tight" upper & lower bound both

$f(n) = \Theta(g(n))$

If $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

$\forall n \geq \max(n_1, n_2)$

for some constant c_1 & $c_2 > 0$

$\Rightarrow f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$

iv) Small - oh (o):

$f(n) = o(g(n))$

$f(n) < cg(n)$ $\forall n > n_0$ & $c > 0$

5) Small omega (ω):

$f(n) = \omega(g(n))$

$f(n) > cg(n)$

$\forall n > n_0$ & $c > 0$

Notes:- * $f(n) = O(g(n)) \rightarrow g(n) = \Omega(f(n))$

* $f(n) = \Omega(g(n)) \rightarrow g(n) = \omega(f(n))$

* $f(n) = \Theta(g(n)) \rightarrow f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$

	Reflexive	Symmetric	Transitive
O	✓	×	✓
Ω	✓	×	✓
Θ	✓	✓	✓
o	×	×	✓
ω	×	×	✓

A-2) for (i = 1; i <= n; i = i * 2) {

}

Values of i: 1, 2, 4, 8, ----, 2^k

which forms a G.P.

$a = 1, r = 2$

$\therefore t_k = a \cdot r^{k-1}$

$n = 1 \cdot 2^{k-1} = 2^k / 2$

$2n = 2^k$

$\therefore k \cdot \log_2 2 = \log_2(2n)$

$k \cdot \log_2 2 = \log_2(2) + \log_2(n)$

$k = 1 + \log_2(n)$

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

A-3) $T(n) = \{ 3T(n-1)$ if $n > 0$, otherwise $1 \}$

$T(0) = 1$

Using forward substitution:

$T(1) = 3T(0) = 3$

$T(2) = 3\,T(1) = 3 \times 3 = 9$

$T(3) = 3\,T(2) = 3 \times 9 = 27$

$\therefore T(n) = 3^n = O(3^n)$

Q-4) $T(n) = \{ 2\,T(n-1) - 1$ if $n > 0$, otherwise $1 \}$

$T(0) = 1$

Using backward substitution :

$T(n-1) = 2\,T(n-2) - 1$

$T(n-2) = 2\,T(n-3) - 1$

$\therefore T(n) = 2(2\,T(n-2) - 1) - 1$

$= 4\,T(n-2) - 2 - 1$

$= 8\,T(n-3) - 4 - 2 - 1$

$= 2^k \cdot T(n-k) - \sum_{i=0}^{k-1} (2^i)$

Put $k = n$

$\therefore T(n) = 2^n \cdot T(0) - \sum_{i=0}^{n-1} (2^i)$

$= 2^n - \left[\frac{1 \cdot (2^n - 1)}{(2-1)} \right]$

$= 2^n - 2^n + 1 = 1$

$\therefore T(n) = O(1)$

Q-5)

```
i = 1, s = 1;
while(s <= n){
    i++; s += i;
    printf("#");
}
```

Values of i & s will change as -

i : 1 → 2 → 3 → 4 → 5 ----

s : 1 → 3 → 6 → 10 → 15 ----

$s_i = s_{i-1} + i$, where $i = 1, 2, 3, \ldots, k$

Sum of A.P. as $n(n+1)/2 = k(k+1)/2$ $\therefore T(n) = \theta(\sqrt{n})$

A-6)

```
void function (int n) {
    int i, count = 0;
    for (i = 1; i * i <= n; i++)
        count++;
}
```

Values of i will be: 1, 2, ---, k

where $k = \sqrt{n}$

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

A-7)

```
void function (int n) {
    int i, j, k, count = 0;
    for (i = n/2; i <= n; i++) {
        for (j = 1; j <= n; j = 2) {
            for (k = 1; k <= n; k = k * 2) {
                count++;
            }
        }
    }
}
```

Values of i will change as: $n/2, (n/2)+1, (n/2)+2 \dots n$

$\Rightarrow (n/2)+1$ times $= O(n)$

Values of j & k change as: 1, 2, 4, 8, 16, ----

Time complexity of these loops will be $O(\log n)$

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

$= O(n \cdot (\log n)^2)$

8-8)
```
function (int n) {
    if (n == 1) return;
    for (i = 1 to n) {
        for (j = 1 to n)
            printf ("*");
    } }
    function (n - 3);
}
```

Values of n will be: $n, n-3, n-6, \ldots, 1 \Rightarrow O(n)$

Complexity of i loop $= O(n)$

Complexity of j loop $= O(n)$

$\therefore T(n) = O(n^3)$

8-9)
```
void function (int n) {
    for (i = 1 to n) {
        for (j = 1; j <= n; j += i)
            printf ("*");
    } }
```

Complexity of i loop $= O(n)$

Complexity of j loop $= O(\sqrt{n})$

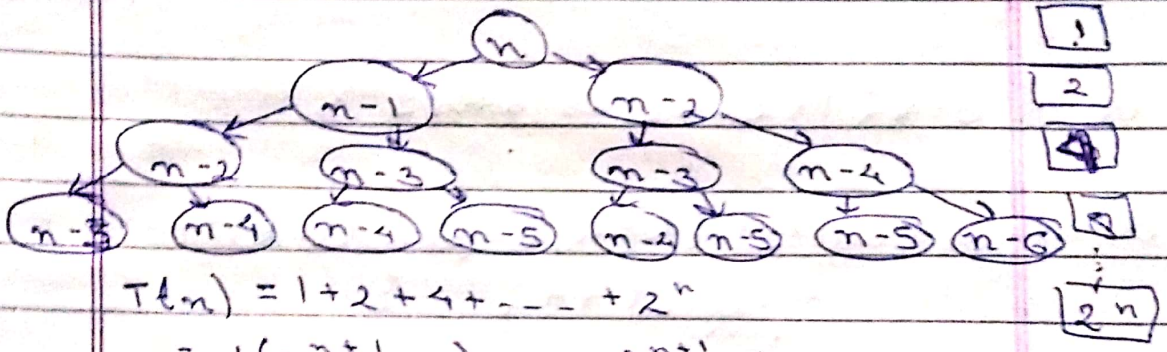
$\therefore T(n) = O(n \cdot \sqrt{n})$

8-11)
```
void function (int n) {
    int j = 1, i = 0;
    while (i < n) {
        i += j; j++; } }
```

Complexity $= O(\sqrt{n})$

A-12) int fib (int n){
if (n<=1) return n;
return fib (n-1) + fib (n-2); }

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



$T(n) = 1 + 2 + 4 + \dots + 2^n$

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

A-13) i) $n(\log n)$

```
for(int i = 0; i < n; i++){
    for (int j = 1; j <= n; j = j * 2) {
        count++; } }
```

ii) $n \wedge 3$

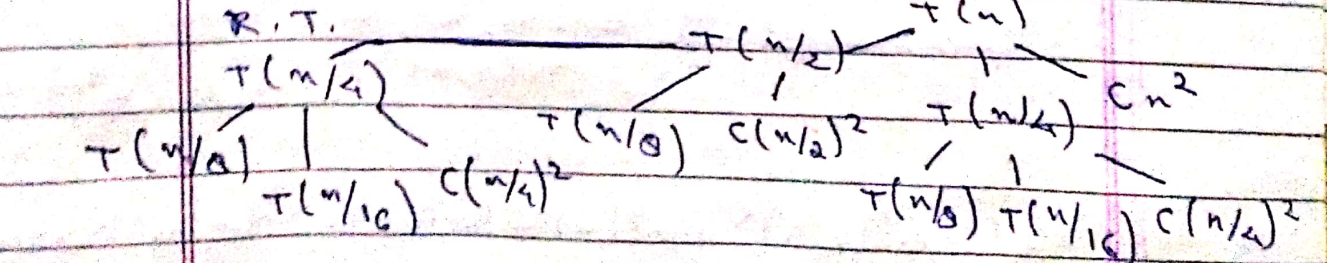
```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        for( int k=1; k<= n; k++)
            count++; } }
```

iii) $\log(\log n)$

```
for (int i = n; i >= 1; i = √i ) {
        count++; }
```

A-14) $T(n) = T(n/4) + T(n/2) + cn^2$

R.T.



No. of nodes per level are 1, 3, 6, 12, ---

∴ Total no. of nodes $= 1 + \frac{3 \cdot (2^{n/2} - 1)}{(2-1)}$

$= 1 + 3 \cdot (2^{n/2} - 1) = 2^n = O(2^n)$

A-15) $O(\sqrt{n})$, same as ans 5 & 11

A-16)
```
for (int i=2; i<=n; i=pow(i,k)){
    count++;
}
```

Values of i will be as follows:

$2, 2^k, 2^{k^2}, 2^{k^3}, \ldots, 2^{k^c}$

As $2^{k^c} <= n \Rightarrow k^c \cdot \log_2 2 = \log_2 n$

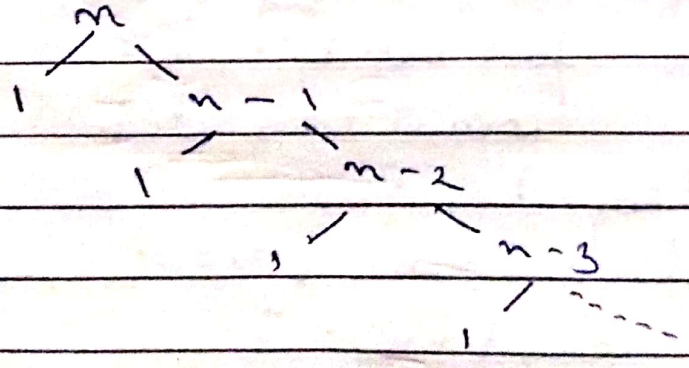
$\Rightarrow k^c = \log n \Rightarrow c \cdot \log_k k = \log_k (\log n)$

$\Rightarrow c = \log_k (\log n) \Rightarrow T(n) = O(\log_k (\log n))$

A-17) Quicksort will repeatedly divide array into 2 parts of 99% & 1%, i.e. when pivot chosen by partition is always either the smallest or largest elmt in array

$\therefore T(n) = T(n-1) + 1 \Rightarrow T(1) = 1$

R.T.



Using forward substitution

$T(1) = 1, T(2) = 2, T(3) = 3 \ldots T(n) = n$

∴ Time complexity $= 1 + 2 + 3 + \ldots + n$

$= n(n+1)/2 = O(n^2)$

A-18) $100 < \log(\log n) < \text{root}(n) \leq \log(n) < n < \log(n!) < n \log n < n!$

$2^n = n^2 < 2^{2n} = 4^n$

b) $1 < \sqrt{\log n} <= \log(\log n) < \log(n) < 2\log(n) < \log(2n)$
$< n < 2n < 4n < \log(n!) < n\log n < n^2 < 2(2^n) < n$

c) $96 < \log_8(n) < \log_2(n) < 5n < \log(n!) < n\log_6(n)$
$< n\log_2 n < 8n^2 < 7n^3 < 8^{2n} < n!$

A-19)
```
index = 0
while (index < n)
    if (arr[index] == key)
        break
    index++
if (index == n) // not found
else // found
```

A-23) Iterative Insertion sort

```
void insertion_sort(int arr[], int n){
  for(int i=0; i<n; i++){
    int temp = arr[i];
    int j = i-1;
    while(j>=0 && arr[j]>temp){
      arr[j+1] = arr[j];
      j--;
    }
    arr[j+1] = temp;
  }
}
```

Recursive Insertion sort

```
void insertion_sort(int arr[], int n){
  if(n<=1)
    return;
  insertion_sort(arr, n-1);
  int last = arr[n-1];
  int j = n-2;
  while(j>=0 && arr[j]>last){
    arr[j+1] = arr[j];
    j--;
  }
  arr[j+1] = last;
}
```

Insertion sort is called online sorting because we can add new element to array's and while sorting is being executed. At any given iteration, say iteration i, only first i elements of array participate in sorting. Therefore we can add more elements to array during sort

Q-21) Complexity of all algorithms discussed in class:

	Best	Average	Worst
• Bubble	$O(n^2)$	$O(n^2)$	$O(n^2)$
• Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
• Insertion	$O(n)$	$O(n^2)$	$O(n^2)$
• Merge	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$

Q-22) • Bubble sort: inplace, stable, offline
• Selection sort: inplace, unstable, offline
• Insertion sort: inplace, stable, online
• Merge sort: not inplace, stable, offline

Q-23&24) Recursive Binary Search -

```
int binary-search (int *arr, int l, int r){
if (l<=r){
    m = l + (r-l)/2;
    if (arr[m] == key)
        return m;
    else if (arr[m] < key)
        return binary-search (arr, m+1, r);
    else
        return binary-search (arr, l, m-1);
} }
```

• Recurrence relation for recursive binary search :-
$T(n) = T(n/2) + 1$
• Time complexity of recursive binary search :
$T(n) = O(\log n)$
• Time complexity of iterative binary search : $T(n) = O(\log n)$
• Time complexity of linear search → $T(n) = O(n)$
• Space complexity in all cases = $O(1)$