Name: Charu Wadhwa

Sec : A

Roll No: 18

Uni. Roll No: 2014620

Assignment - 1

Q1. What do you understand by Asymptotic notations. Define different asymptotic notations with examples.

Ans). Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis.

- **Θ Notation**: The theta notation bounds a function from above and below, so it defines exact asymptotic behaviour. A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For e.g. consider the following example:

  $3n^3 + 6n^2 + 6000 \Rightarrow \Theta(n^3)$.

  Dropping lower order terms is always fine because there will always be a no after which $\Theta(n^3)$ has higher values than $\Theta(n^2)$ irrespective of the constants involved.

  $\Theta(g(n)) = \{f(n)$: there exist positive constants c1, c2 and n0 such that $0 <= c1 * g(n) <= f(n) <= c2 g(n)$ for all $n >= n0\}$

- **Big O Notation**: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of insertion sort. It takes linear time in best case and quadratic time in worst case. We can safely say that Time complexity of Insertion sort is $O(n^2)$ and it also covers the case of linear time.

  $\Theta(n^2) \rightarrow$ worst case, $\Theta(n) \rightarrow$ best case, $O(n^2)$ is for overall case.

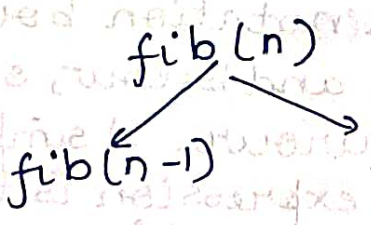
  $O(g(n)) = \{f(n)$: there exist positive constants c and n0 such that $0 <= c * g(n) <= f(n)$ for all $n >= n0\}$

- **Ω Notation:**

Just as Big O notation provides an asymptotic upperbound, Θ provides exact bound, Ω notation provides lower bound.
Least used.

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and no such that $0 <= cg(n) <= f(n)$ for all $n >= n_0\}$

e.g. For fib(n):



$\Rightarrow O(2^n)$
$\Rightarrow \Theta(1.68^n)$
$\Rightarrow \Omega(1.5^n)$

- **o notation:**

Loose upper bound of f(n), small-o notation
Let f(n) and g(n) be functions (strict)

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

$f(n) < c(g(n))$

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

e.g.
Merge sort
$\Rightarrow o(n^2)$

- **ω notation:** strict lower bound - small omega notation.

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

$f(n) > c(g(n))$

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

e.g.
Merge sort:
$\Rightarrow \omega(n)$

2) What should be the time complexity of:

```
for (i=1 to n)
{
  i = i * 2
}
```

value of i

1	2^0
2	2^1
4	2^2
8	2^3
16	2^4
⋮	⋮
n	2^k

(k+1)th iteration.

$2^k = n$

$\log 2^k = \log n$

$k \log_2 2 = \log n \quad (\log_2 2 = 1)$

$k = \log n$

$O(k) \Rightarrow O(\log n)$

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

Solution:

(using substitution)

$$T(n) = 3T(n-1)$$
$$= 3(3T(n-2))$$
$$= 3^2(T(n-2))$$
$$= 3^3(T(n-3))$$
$$= \text{.....}$$
$$= 3^n T(n-n)$$
$$= 3^n T(0)$$
$$= 3^n * 1 = 3^n$$
$$= O(3^n)$$

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

$$T(n) = 2T(n-1) - 1$$
$$= 2(2T(n-2) - 1) - 1$$
$$= 2^2(T(n-2)) - 2 - 1$$
$$= 2^3(2T(n-3) - 1) - 2 - 1$$
$$= 2^3(T(n-3)) - 4 - 2 - 1$$
$$= 2^n(T(n-n)) - 2^{n-1} - 2^{n-2} - 2^{n-3}$$
$$\Downarrow T(0)$$
$$= 2^n - 2^{n-1} - 2^{n-2} - \text{....} 2^2 - 2^1 - 2^0$$
$$= 2^n - (2^n - 1) \qquad \left[S_n = \frac{r^n - 1}{r - 1}\right] \text{ G.P.}$$
$$= 1 \qquad a = 1, \; r = 2$$

$T(n) = 1 \Rightarrow O(1)$.

(5) What should be the complexity of :-

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

3

i	s	
1	1	Initial
2	3	
3	6	
4	10	
5	15	

From this we can see that s at ith iteration is sum of first i positive integers.

$1+2+3+\cdots+K$ at Kth iteration

$$\frac{K(K+1)}{2} \geq n \Rightarrow \frac{K^2+K}{2} \geq n$$

$$K^2 \geq 2n-K$$

$$K^2 \geq n$$

$$K \geq \sqrt{n}$$

$$O(K) \Rightarrow O(\sqrt{n})$$

⑥ Time Complexity:

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

i
1
2^2
3^2
:
:
4^2
:
k^2

$k * k >= n$

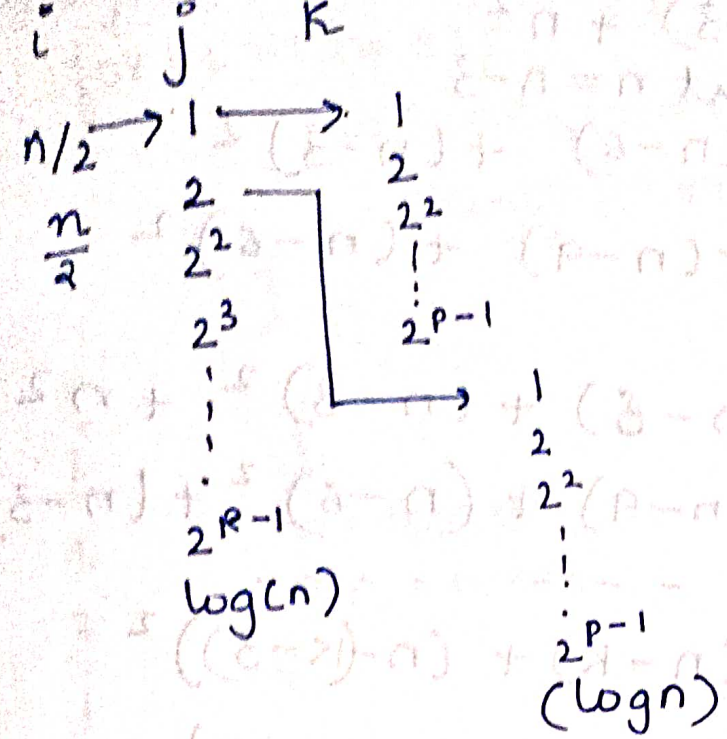
$k^2 >= n$

$k >= \sqrt{n}$

$O(\sqrt{n})$

⑦ Time complexity:

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



$$O\left(\frac{n}{2} * \log(n) * \log(n)\right)$$

$$\Rightarrow O(n * (\log n)^2) \Rightarrow O(n \log^2 n)$$

(8)

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

$$T(n) = T(n-3) + n^2$$

↓ using Backward substitution

$$T(n) = T(n-3) + n^2$$

$$T(1) = 1$$

$$T(n) = T(n-3) + n^2$$

Put $n = n-3$

$$T(n-3) = T(n-6) + (n-3)^2$$

$$T(n-6) = T(n-9) + (n-6)^2$$

$$T(n) = T(n-6) + (n-3)^2 + n^2$$

$$= T(n-9) + (n-6)^2 + (n-3)^2 + n^2$$

- - - - - - - -

$$= T(n-k) + (n-(k-3))^2 + (n-(k-6))^2 + \dots + n^2$$

put $n-k = 1 \Rightarrow k = n-1$

$$= T(1) + (-2)^2 + (-5)^2 + (-8)^2 + \dots + n^2$$

$n=1$
$3x1-1$

$$= 1 + 2^2 + 5^2 + 8^2 + \dots + n^2$$

$$= 1 + \sum (3n-1)^2$$

$$= \sum (3n-1)^2$$

$$= \sum (9n^2 + 1 - 6n)$$

$$= \frac{9(n)(n+1)(2n+1)}{6}$$

$$\approx n^3$$

$$O(n^3)$$

$\Rightarrow$ Time complexity

$O(n^3)$ Ans.

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

i	j
1	n - times
2	n/2 - times
.	.
.	.
n	1 time

For ith iteration n/i terms

$O(n \log n) \Rightarrow O(n \log n)$.

(10) $n^k = O(a^n)$

$c * g(n) \leq f(n)$

$c * a^n \leq n^k$

For $k >= 1$

$a > 1$

Let $k = 2$

$a = 2$

$c * 2^n \leq n^2$

$n = 2$

$c = 1$

$1 * 2^2 \leq 2^2$

$4 \leq 4$

Let $k = 1$

$a = 2$

$c * 2^n \leq n$

$n = 0$

$c = 0$

Let $k = 2$ $a = 2$

$f(n) = n^2$

$g(n) = 2^k$

$f(n) >= c * g(n)$

$n^2 >= c * 2^k$

for $k >= 2$ & $a >= 2$ condition satisfies.

taking log

$O(n) > O(\log n)$

(11)

```
void fun (int n)
{
 int j =1, i=0;
 while (i<n)
  {
   i= i+j ;
   j++;
  }
}
```

j	i
1	1
2	3
3	6
4	10

Initially
j=1 i=0

At ith iteration, i is calculation of sum of positive integers till j.

$1+2+3+\cdots\cdots K.$

$$\frac{K(K+1)}{2} > n$$

$$K(K+1) > n$$

$$K^2 > n$$

$$K > \sqrt{n}$$

T.C. → $O(\sqrt{n})$ Ans.

(12) Recurrence relation for Fibonnacci series:

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

$T(n-1) \simeq T(n-2)$

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

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

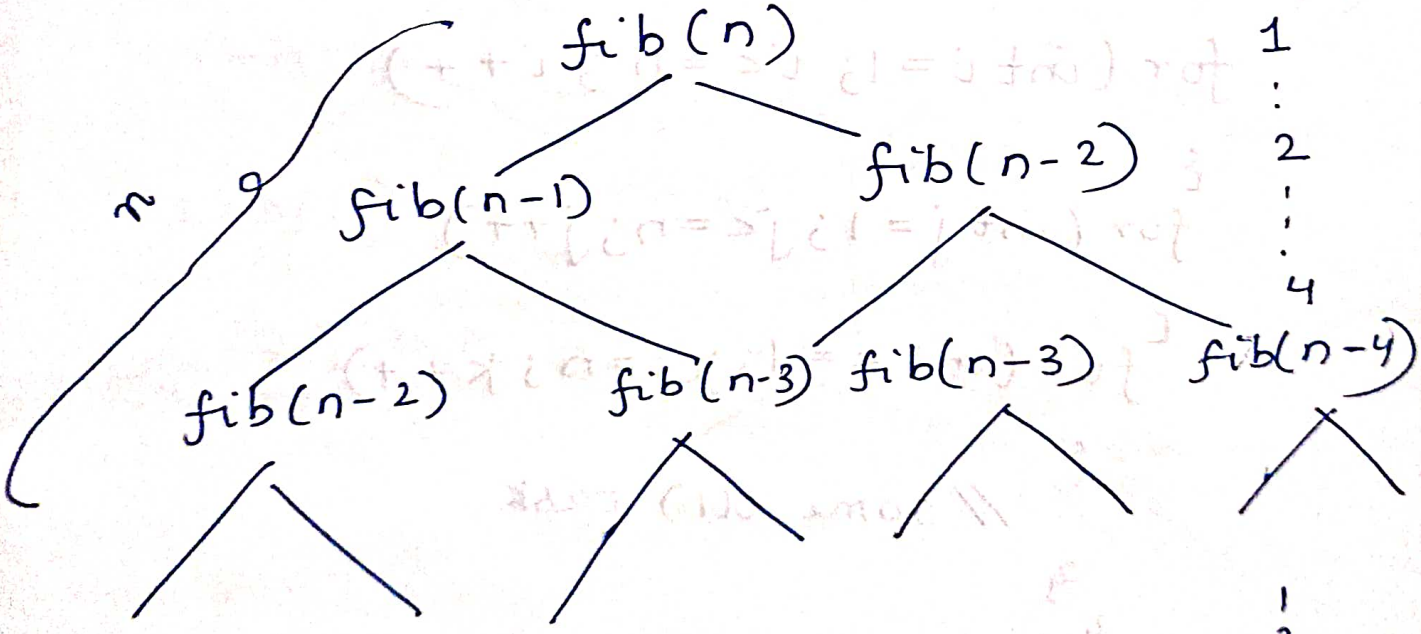
$$= 2^2 * T(n-2)$$

$$= \vdots$$

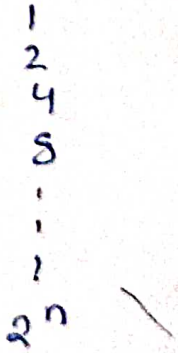
$$= 2^n * T(n-n)$$

$$= O(2^n)$$

$$= O(1.68^n)$$



Height of the recursion tree is n, and there are n stack frames created so space complexity is $O(n)$.



(13) Write programs which have complexity n(logn), n^3, log(log(n))

(i) n(logn)

```
for(int i= 1; i<=n; i++)
 {
 for(int j=1; j<=n; j=j*2)
    {
        // some O(1) task
    }
 }
```

(ii) n^3

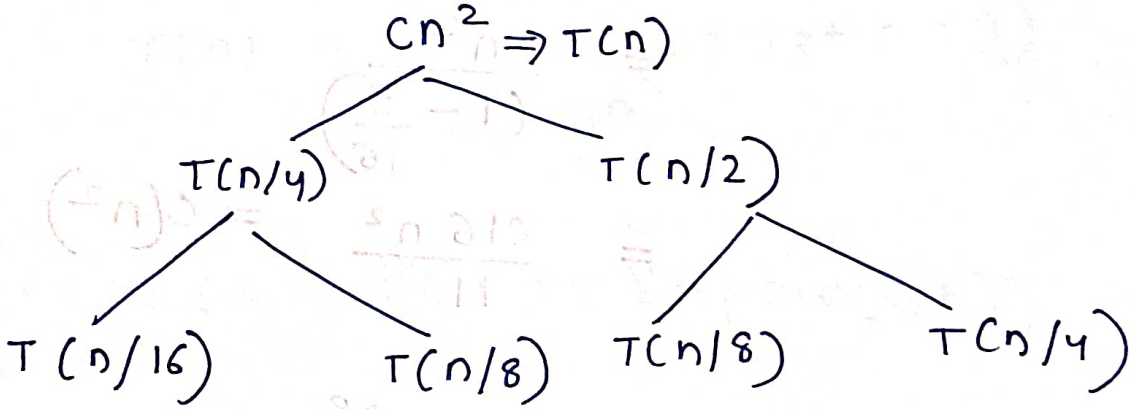
```
for(int i=1; i<=n; i++)
 {
 for(int j=1;j<=n;j++)
 {for(int k=1;k<=n;k++)
    {
        // some O(1) task
    }
 }
}
```

(iii) $O(\log(\log n))$

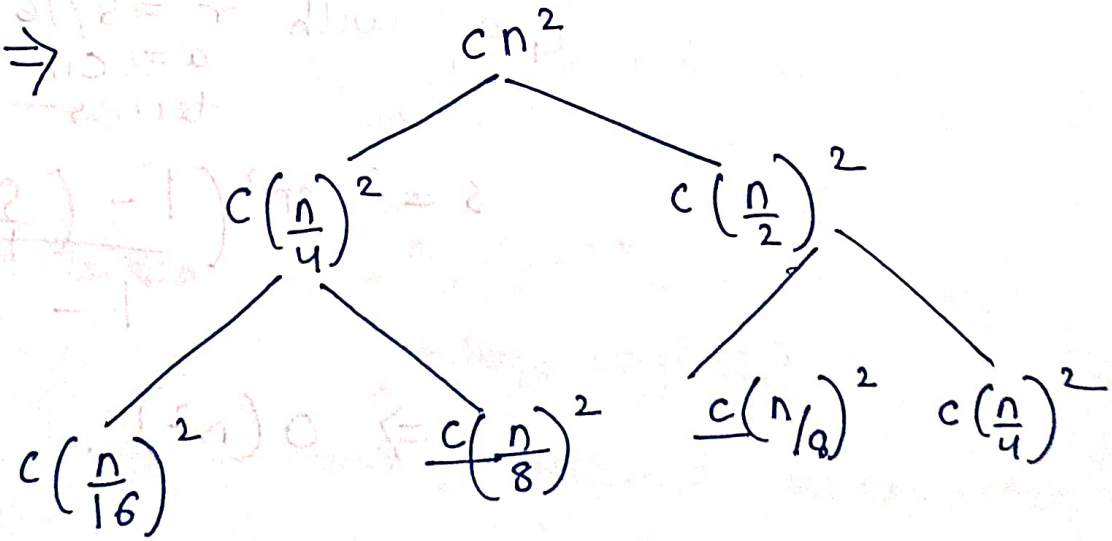
```
for (int i=2; i<=n; i = pow(i, c))
{
  // some O(1) task
}.
```

(14) Solve the following recurrence relation:

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



$T(n) \Rightarrow$



Sum of the three levels :

$$= cn^2 + c\left(\frac{n}{4}\right)^2 + c\left(\frac{n}{2}\right)^2 + c\left(\frac{n}{16}\right)^2 + c\left(\frac{n}{8}\right)^2 + c\left(\frac{n}{8}\right)^2 + c\left(\frac{n}{4}\right)^2$$

$$= c\left(n^2 + 5(n^2)/16 + 25n^2/256 + \cdots \infty\right)$$

$\Rightarrow$ G.P with $r = 5/16$

$a = cn^2$

$$S_n = \frac{a}{1-r}$$

$$= \frac{cn^2}{\left(1 - \frac{5}{16}\right)}$$

$$= \frac{c16n^2}{11} = O(n^2).$$

OR

G.P. with $r = 5/16$

$a = cn^2$

terms $\longrightarrow \log_2 n$

$$S = cn^2\left(\frac{1 - \left(\frac{5}{16}\right)^{\log_2 n}}{1 - 5/16}\right)$$

$\Rightarrow O(n^2)$

(15)
```
int fun(int n)
{
for (int i=1; i<=n; i++)
 {
 for(int j=1; j<n; j+=i)
  {
  // some O(1) task.
  }
 }
}
```

$i=1 \longrightarrow n$ times

ith iteration $\longrightarrow n/i$ times

$$T(n) = O(n(1+1/2+1/3+\cdots))$$
$$= O(n\log n).$$

(16)
```
for (int i=2; i<=n; i=pow(i,k))
 {
  //(some O(1) task)
 }
```

where k is constant

Ans i takes $2, 2^k, (2^k)^k, \ldots\ldots$

$$2^{k^{\log_k \log(n)}}$$

$$\Rightarrow 2^{\log_2 n} = n \rightarrow \text{Last term}$$

So T.C. $= O(\log\log n)$

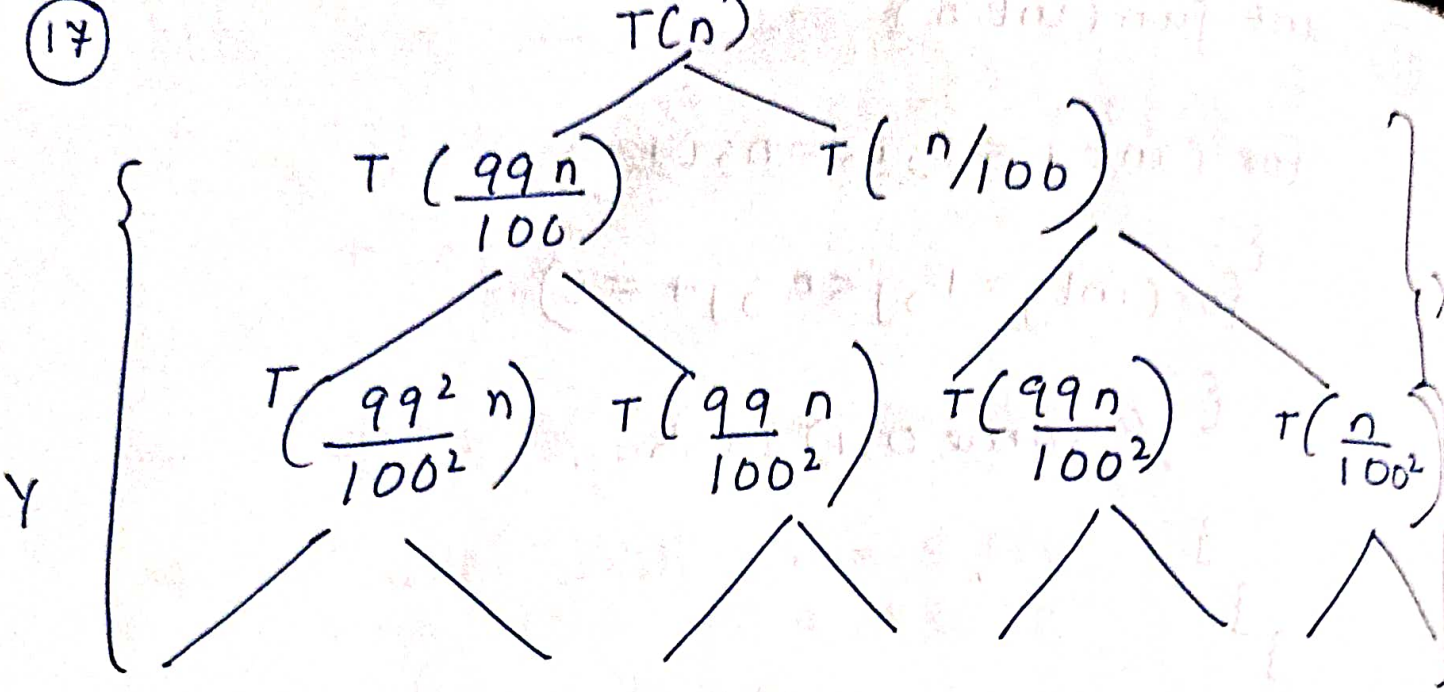
$2^{k^0}, 2^{k^1}, 2^{k^2} \ldots\ldots 2^{k^{c-1}}$

$2^{k^{c-1}} < n \Rightarrow k^{c-1} < \log_2 n$

$c-1 < \log_k \log_2 n$

Base does not matter $\leftarrow c < \log_k \log_2 n = O(\log\log n)$

(17)



$Y = \log_{\frac{100}{99}} n$ → height of left side

$x = \log_{100} n$ → height of right side

Difference b/w heights of extreme part $= \log_{100} n - \log_{\frac{100}{99}} n$

99 parts and 1 part.

$$T(n) = T\left(\frac{99n}{100}\right) + T\left(\frac{n}{100}\right) + n.$$

⟶ Recurrence Relation.

Analysis

Every level of tree has cost cn, until the recursion reaches a boundary condition at depth $\log_{100} n \Rightarrow O(\log n)$ and for the the right side log. and the levels have cost n. For left side the recursion terminates at $\log_{\frac{100}{99}} n = O(\log n)$. Total cost of quick-sort is therefore $n \log n \Rightarrow O(n \log n)$ since the split has constant proportionality it yields $O(n \log n)$.

(18)

a) $100 < \log(\log(n)) < \log(n)$
$< \sqrt{n} < n < \log(n!) < n \log n$
$< n^2 < 2^n < 2^{2n} < 4^n < n!$

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

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

(19)

```
linear_search ( a[], item , pos, n)
pos = -1
for (i=0; a[i] <= item && pos = -1 && i<n; i++)
{
if (a[i] = = item)
 pos =i
 }
 return (pos)
```

(20) Iterative Insertion sort

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

Recursive insertion sort

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

Insertion sort is online sorting algorithm

Because online algorithm is one that can process its input piece by piece in serial fashion, i.e. in order that the input is fed to the algorithm without having entire input available from beginning and insertion does not know the whole input.

Offline Algorithms	Online algo.
Bubble sort,	Insertion sort
Merge sort,	
selection sort,	
Quick sort.	

(21) Complexity of all sorting algorithms

	Best	Average	Worst
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Quick Sort	$\Omega(n \log(n))$	$\Theta(n \log n)$	$O(n^2)$ ~~$O(n \log n)$~~
Merge sort.	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$

(22) Sorting Algorithm

Sorting Algorithm	Inplace	Stable	Online
Selection Sort	✓	✗	✗
Bubble Sort	✓	✓	✗
Insertion sort	✓	✓	✓
Quick Sort	✓	✗	✗
Merge Sort	✗	✓	✗

(23) Recursive Binary Search

```
int Binary_search (int arr[], int l, int r, int x)
{
  if (r >= l) {
int mid = l + (r - l) / 2;
  if (arr[mid] == x)
   return mid;
   if (arr[mid] > x)
   return Binary_search(arr, l, mid-1, x);
   return Binary_search (arr, mid+1, r, x);
   }
   return -1;
   }
```

T.C. → O(logn)
S.C. → O(logn)

Iterative Binary Search

```
int binarysearch (int arr[], int l, int r, int x)
{
while (l<=r)
   {
   int m = l+(r-l)/2;
    if (arr[m] == x)
    return m;
    if(arr[m]<x)
    l = m+1;
    else
     r = m-1;
   }
return -1;
}
```

T.C. → O(logn)

S.C. → O(1).

(24) Recurrence Relation for Binary Search

$$T(n) = T(n/2) + O(1).$$