Name:- Mayank Joshi
Course :- B.Tech CSE
University Roll no:- 1961102
Section:- C
Semester:- 5

Assignment - 01

Q1. Sol Asymptotic notations are used to write fastest and slowest possible running time for an algorithm. These are also referred to as best case and worst case respectively.

There are three types of asymptotic notations :-

a. Big Theta (Θ)
b. Big Oh (O)
c. Big Omega (Ω)

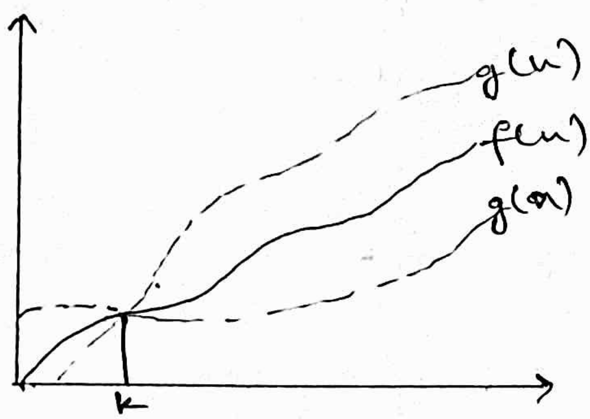
Big Θ

The time complexity represented by the Big Θ notation is like the average value or range within which the actual time of execution of the algorithm will be.

E.g:- $4n^2+6n$

We use the Big Θ notation to represent ~~tim~~ this, where the time complexity would be $\Theta(n^2)$ ignoring the constant coefficient and removing insignificant part, which is $6n$.

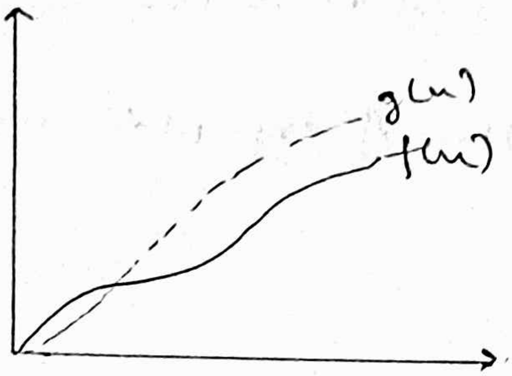
$\Theta(f(n)) = \{g(n)$ if and only if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$ for all $n > n_0\}$



Big Oh Notation → (O)

It is the formal way to express the upper bound of an algorithm running time. It measures the worst case time complexity or the longest amount of time an algorithm can possible take to complete.

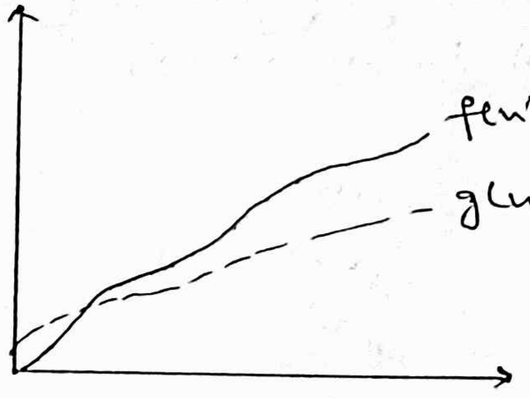
$$O(f(n)) = \{g(n): \text{there exists } c > 0 \text{ and } n_0 \text{ such that } f(n) \le g(n) \text{ for all } n > n_0\}$$



Omega Notation → (Ω)

The notation Ω(n) is the formal way to express the lower bound of an algorithm running time.
It measure the best case time an algorithm can possibly take to complete.

$$\Omega(f(n)) \ge \{g(n): \text{there exists } c > 0 \text{ \& } n_0 \text{ and such that } g(n) < c \cdot f(n) \text{ for all } n > n_0\}$$



Q2.
sol

for (i= 1; i≤n; i=i×2)

so, 1,2,3,4,8, ---.

$T(n) = O(\log_2 n)$

Q3.
sol

Let us solve this using substitution

$\boxed{T(n) = 3T(n-1)}$ —— (i)

Put n=n-1 in eq (i), we get

$T(n-1) = 3T(n-1-1)$

$\boxed{T(n-1) = 3T(n-2)}$ —— (ii)

Put the values of T(n-1) from (ii) in (i), we get

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

$\boxed{T(n) = 3^2(T(n-2))}$ —— (iii)

Put n=n-2 in eq(i), we get

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

$\boxed{T(n-2) = 3T(n-3)}$ —— (iv)

Put value of T(n-2) from (iv) to (iii), we get

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

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

So,

$T(n) = 3T(n-1)$

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

$= 3^2(T-2)$

$\vdots$

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

$= 3^n T(0)$

$= 3^n (1)$

$\boxed{T_n = 3^n}$

So, time complexity of this function is $O(3^n)$

(3)

5. Here we can define the term 's' according to relation

$$s_i = s_{i+1} + 1$$

Here, while loop can be terminated if k is total number of iterations taken by the program.

if $1+2+3+4 .... + k$

$$= \left[\frac{k(k+1)}{2}\right] > n$$

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

So, time complexity of the above function $O(\sqrt{n})$.

6. In this, if k is the total no. of iterations taken by a program.

∴ then the loop terminates

$$\Rightarrow (1)^2 + (2)^2 + (3)^2 + \text{———} (\sqrt{n})^2$$

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

7.

$$T(n) = O(n * \log_2 n * \log_2 n)$$

$$T(n) = O(n * (\log_2 n)^2)$$

So,

$$\boxed{T(n) = O(n(\log_2 n)^2)}$$

8.

$$T(n) = T(n-3) + n^2 \quad \text{———} (1)$$

$$T(n-1) = T(n-1-3) + (n-1)^2$$

~~T(n)~~.

~~T(n-1) = T(n~~

$$T(n) = T(n-4) + n^2 + (n-1)^2$$

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

|

|

$$T(n) = T(n-k) + (n^2 + (n-1)^2 + (n-2)^2 - — (k-2) \text{ terms}$$

$T(n-k)=1$

$k=n-1$

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

$T(n)=T(1)+(4^2+5^2+ \dots +n^2)$

$T(n)=T(1)=\left(\frac{(n-3)(n-2)(2n-5)}{6}\right)$

$T(n)=1+\left(\frac{2n^3+ \dots}{6}\right)$

$T(n)=n^3$

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

9. $i=1$ (is n times)

$i=2$ $(1,3,5, \dots n/2)$

$i=3$ $(1,4,7, \dots n, n/3)$

,

|

$i=n$ (0)

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

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

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

10. Given, ~~k~~ $\cdot n^k$ and ~~a~~ a^n

So, $k\geq 1$ and $a\geq 1$

Relation is n^k is $O(c^n)$.

11.

Here, $0, 3, 6, 10, 15, \dots n$.

k^{th} term $=\frac{k(k+1)}{2}=\frac{k^2+k}{2}$

$k=\sqrt{n}$

$T=\Theta(\sqrt{n})$

12. Here, recurrence relation of fabonacci series is

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

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

$T(n) = 4T(n-4) + 3$

$T(n) = 8T(n-6) + 7$

$T(n) = 16\,T(n-8) + 15$

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

for, $T(n-2k) = T(0)$

$n = 2k$

$k = n/2$

$T(n) = 2^{n/2}\,T(0) + (2^{n/2} - 1)$

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

$T(n) = O(2^n)$

So, the space complexity of fabonacci series is $O(n)$.

13. <u>for n(logn)</u>

```
for(int i=0; j<n; i++)
{ for (int j=0; i<n; i=i*o)
   {
     printf("*");
   }
}
void main()
{
   try();
}
```

for n^3

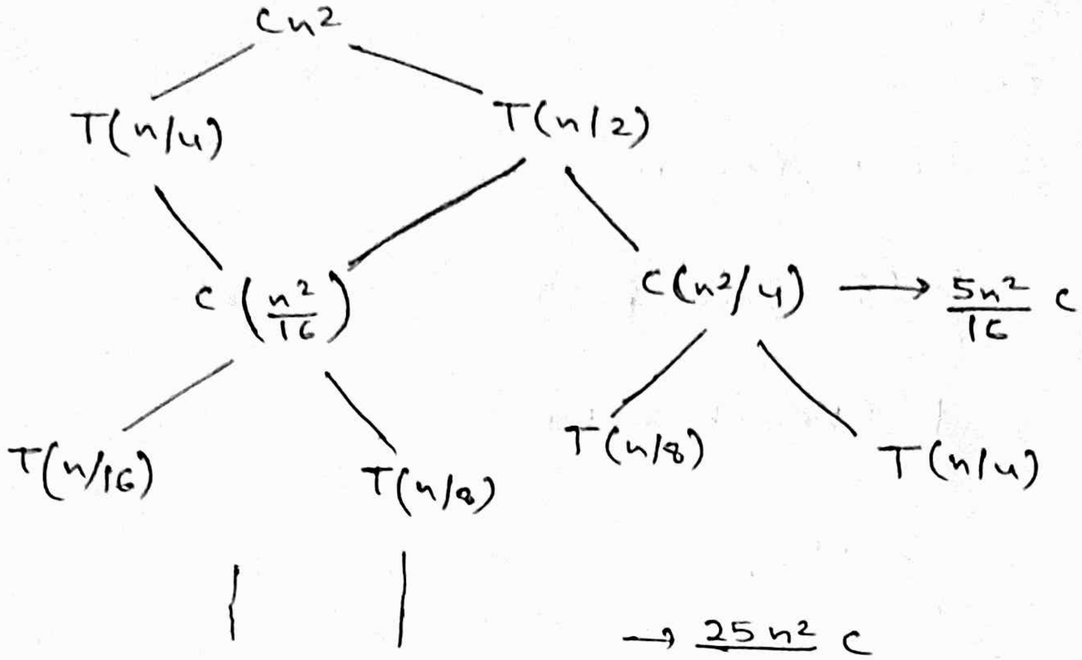
```
#include <stdio.h>
void main(){
int i, j, k;
int n;
cin >>n;
for (i= 0 ; i <n; i++)
 {
  for (j=0 ; j<n; j++)
  {
   for(k=0 ; k<n; k++)
   {
     n++;
   }
  }
 }
}
```

for log(logn)

```
#include <iostream>
void fun x (int n)
   {
    if (n==2)
     return 1;
    else .
     fun (sqrt(n));
   }
   void main()
   {
    fun(100);
   }
```

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

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



$\longrightarrow \frac{25n^2}{256}c$

$T(n) = cn^2 + \frac{5cn^2}{16} + \frac{25cn^2}{256} + \cdots$

Here, it is a G.P.

with $a = n^2$

$r = 5/16$

So, sum of GP

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

$= \frac{16cn^2}{11} = \frac{16cn^2}{11}$

$T(n) = O(n^2)$

15.

$\underbrace{n, \frac{n}{2}, \frac{n}{3}, \frac{n}{4}, \frac{n}{5}, \cdots\cdots}_{k\text{-times}}$

$k = \log_2 n$

$n(1, 1/2, 1/3, 1/4, \cdots 1/n)$

$(n(\log n))$

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

16.

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

It is a G.P

$a = 2$

$r = 2^k$

k^{th} term $= ar^{k-1}$

$n = 2(2^k)^{k-1}$

Let $k^{k-1} = x$

$k \log_k k = \log x$

$k = \log x$ ——— (i)

$n = 2^x$

$\log_2 n = x \log_2 2$

$x = \log_2 n$

$\log x = \log(\log n)$

from (i)

$k = \log(\log(n))$

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

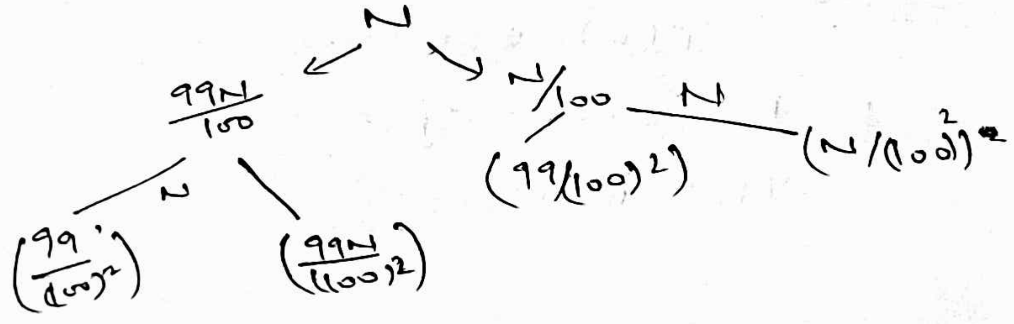
17.

Given,

Array is divided in 99% and 1%.

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

Now, so here we can use a extreme of a tree where initial point is n.



$$N\left(\frac{(99)(99)}{(100)(100)}\right) + \frac{(99)(1)}{(100)(100)}\Bigg) + \frac{100}{100\times 100}\,N$$

$$\Rightarrow \frac{99N}{100} + \frac{N}{100} = N$$

So cost of each level is N only.

Total cost = height * cost of each level.

so for 1st term = $N, \frac{99N}{100}, \left(\frac{99}{100}\right)^2 N$ ----

$$\left(\frac{99}{100}\right)^{n-1} N = 1$$

$$\left(\frac{99}{100}\right)^{n-1} = \frac{1}{N}$$

$$N = \left(\frac{100}{99}\right)^{n-1}$$

$$\log n = h \log(n)$$

$$h = \log n$$

$$h = \frac{\log N}{\log\left(\frac{99}{100}\right)} + 1$$

Height of 2nd tree.

$$N, \frac{N}{100}, \frac{N}{(100)^2}, \frac{N}{(100)^3} + \text{----} 1$$

$$N\left(\frac{1}{100}\right)^{n-1} = 1$$

$$N = (100)^{n-1}$$

$$(n-1)\log 100 = \log N$$

$$h = \frac{\log N}{\log 100} + 1 \quad 19N = \log(N)$$

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

so, time complexity is

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

Height of both extreme is $\frac{\log N}{\log 100} + 1$ of $\frac{1}{100}$

and $\frac{\log N}{\log(\frac{100}{99})} + 1$ of $\frac{99}{100}$

So, the conclusion is that if division is done more than height of tree will be more & more when division ratio is less than height is less.

18.

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

b) $O(1) < O(\log(\log(n))) < O(\log(n)) < O(\log 2n) < O(2\log n) < O(n) < O(n\log(n)) < O(\log(n!)) < O(2n) < O(4n) < O(n^2) < O(n!) < O(2(2^n))$.

c) $O(96) < O(\log_2(n)) < O\log n(n) < O(\log n!) < O(n\log(n)) < O(n\log_2(n)) < O(5n) < O(8n^3) < O(7n^3) < O(n!) < O(8^{\wedge}2^n)$.

19. ~~void insertion sort~~ (arr, n)

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

19. void ls (int arr[], int n, int n)

```
{
for(i=0 to i=n)
if arr[i]==key
  cout << "found";
else
 continue;
}
```

20. <u>Iterative insertion sort</u>

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

<u>Insertion Sort</u>

```
{ if n<=1
    return;
  insertion sort (arr, n-1);
  last = arr[n-1];
  j = n-2
  while (j >=0 and arr [j] > last)
     { arr [j+1] = arr [j]
       j--;
     }
    arr [j+1] = last
```

Insertion sort is called ~~one~~ line sorting because it don't know the whole input, it might make decision that later turn out to be not optimal ~~ot~~ other algorithm are off-line algorithms.

21.

	Time Complexity			Space Complexity
	Best	Avg	Worst	
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$ {recursion}
Quick Sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(n)$
Heap Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$

22.

	Inplace	Stable	on line Sorting
Bubble Sort	Yes	Yes	No
Selection Sort	Yes	No	No
Insertion Sort	Yes	Yes	Yes
Merge Sort	No	Yes	No
Quick Sort	Yes	No	No
Heap Sort	Yes	No	No

23. Binary Search (arr, int n, key)

```
{ start = 0
  end = n-1
  while (start <= end)
  mid = (start + end) / 2
  if [arr [mid] == key]
     found
  else if arr [mid] < key
       start = mid + 1
```

```
else
    end = mid - 1
  }
}
```

Time Complexity of linear search

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

Space complexity of linear search is $O(1)$

Time Complexity of binary search

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

Space complexity of binary search $= O(n)$

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