PAGE _____
DATE _____

# Design & Analysis of Algorithms
## Assignment: 1

1. Asymptotic Notations are methods / languages using which we can define the running time of the algorithm based on input size. These notations are used to tell the complexity of an algorithm when the input is very large.
Suppose we have an algorithm as a function of and n as the input size $f(n)$ will be the running time of the algorithm. Using this we make a graph with y-axis as running time ($f(n)$) and x-axis as input size ($n$)
The different asymptotic notations are —

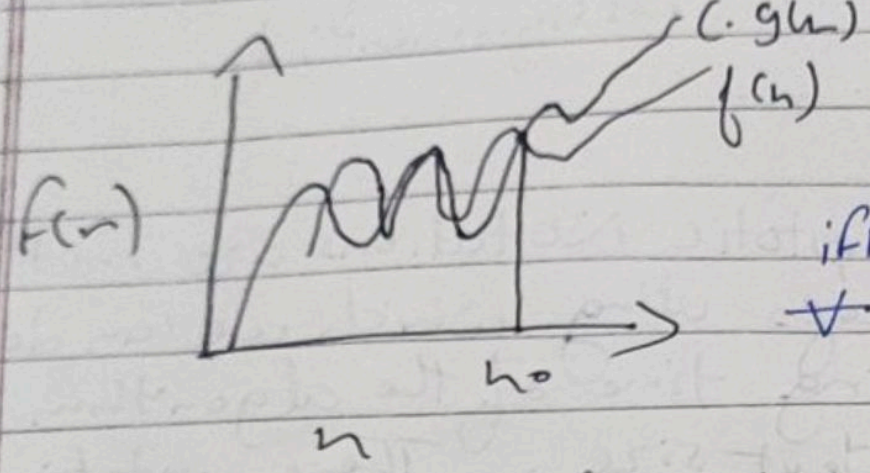
a) Big - O - notations
It is an asymptotic notation for the worst case or the ceiling growth for a given function.

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

PAGE

DATE

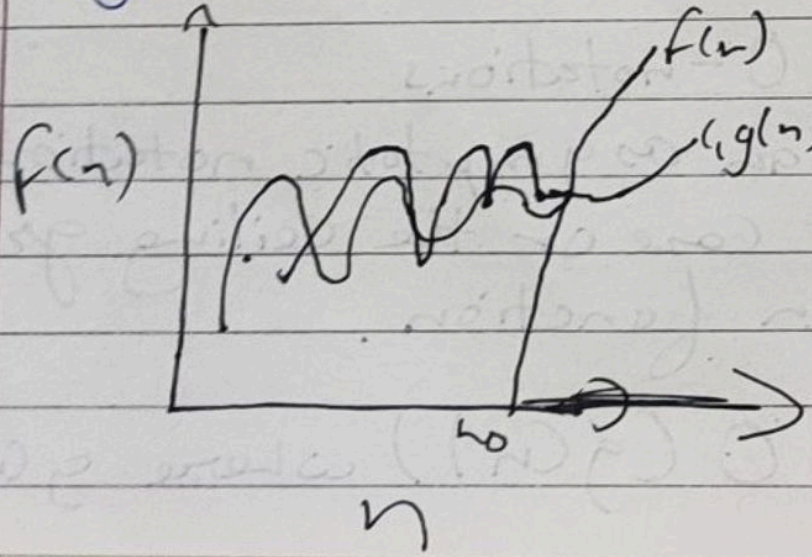
tight upper bound of $f(n)$




$f(n) = O(g(n))$
iff $f(n) \leq c \cdot g(n)$
$\forall\ n \geq n_0$, some constant $c > 0$

for eg- $f(n) = n^2 + 3n + 4$ | let $c = 100$
$g(n) = n^2$ | $n^2 + 3n + 4 \leq 100$
$n^2 + 3n + 4 = O(n^2)$ | $\forall\ n \geq 1$

b) Big - Omega ($\Omega$) :- It is the asymptotic notation for the best case or a floor growth rate for a given $f(n)$.
$f(n) = \Omega(g(n))$ where $g(n)$ is tight lower bound of $f(n)$

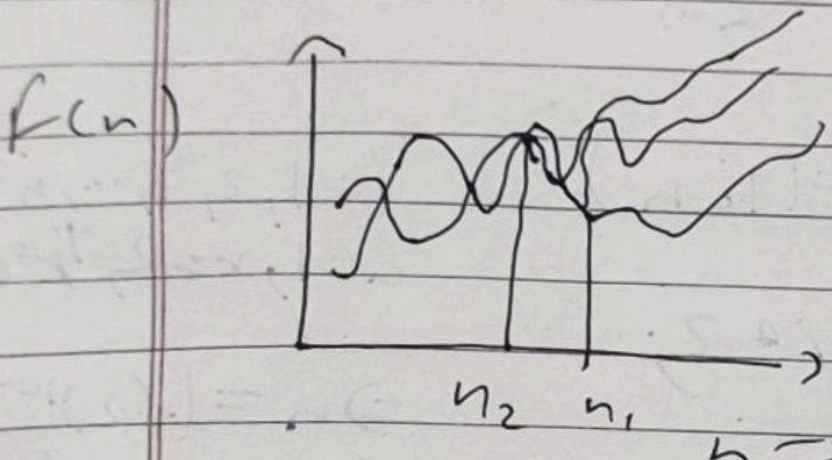



$f(n) = \Omega(g(n))$
iff $f(n) \geq c(g(n))$
$\forall\ n \geq n_0$ &
$c > 0$

PAGE
DATE

c) Theta ($\Theta$) is an asymptotic notation to denote the asymptotically tight bound on the growth rate of runtime of an algorithm.

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




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

iff $c_1 g(n) \leq f(n) \leq c_2 g(n)$

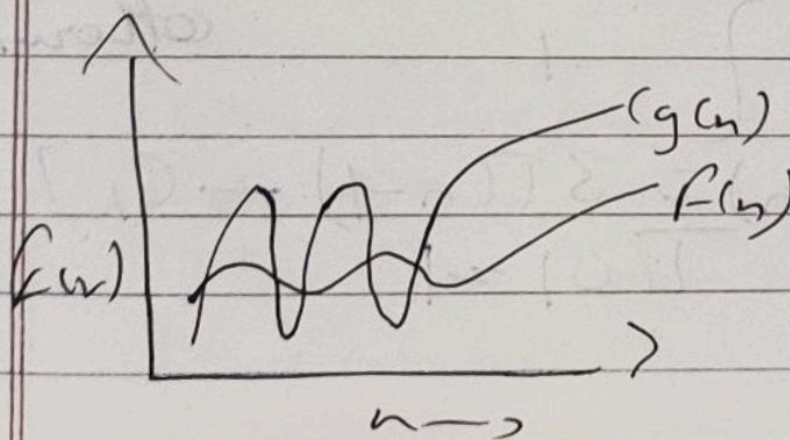
$\forall\ n \geq \max(n_1, n_2)$

$c_1, c_2 > 0$

d) Small-$\Theta$ denotes the upper bound (not tight) on the growth rate of runtime of an algorithm $f(n) = \Theta(g(n))$

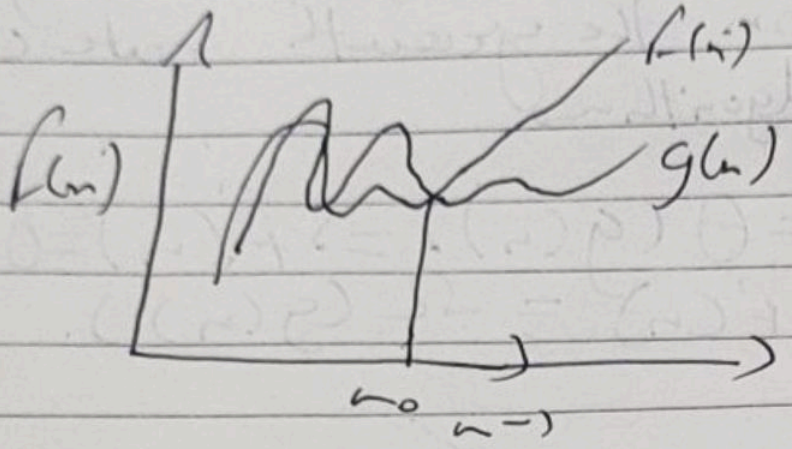
$\Rightarrow f(n) \leq c g(n)\ \forall\ n_1 \geq n_0$ & $c > c_0$

Eg. $n = O(n^2)$



PAGE
DATE

e) Small omega ($\omega$) denotes the lower bound




$f(n) = \omega(g(n))$
$f(n) > c(g(n))$
$\forall n > n_0 \& c$
eg. $n^2 = O(n)$

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

$i = 1, 2, 4, \ldots, n \to (1)$
$a = 1, r = 2, T_k = ar^{k-1}$

$\Rightarrow n = 1.(2)^{k-1}$
$\Rightarrow k = \log_2(2n)$
$= \log_2 n + 1$

T.C $= O(\log_2 n + 1)$
$= O(\log n)$

3. $T(n) = \begin{cases} 3T(n-1) & \text{if } n > 0 \\ 1 & \text{otherwise} \end{cases}$

$\Rightarrow T(n) = 3T(n-1) \quad -(1)$
$T(0) = 1$

PAGE _____
DATE _____

Using backward substitution,

Put $n = n-1$ in eq (1)

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

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

Put eq (2) in eq (1)

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

$T(n) = 9T(n-2) - (3)$

Put $n = n-2$ in eq (1)

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

$T(n-2) = 3T(n-3) - (4)$

Put in eq (3)

$T(n) = 9[3T(n-3)]$

$T(n) = 27T(n-3)$

$\Rightarrow T(n) = 3^k T(n-k)$

$n-k = 0 \Rightarrow n = k$

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

$= 3^n T(0) = 3^n$

$\Rightarrow$ Time Complexity $O(3^n)$

4. $T(n) = \begin{cases} 2T(n-1)-1, & \text{if } n>0 \quad -(1) \\ 1, & \text{otherwise} \end{cases}$

Using backward substitution,

Put $n = n-1$ in (1)

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

Put (2) in (1)

PAGE
DATE

$T(n) = 2[2T(n-2)-1]-1$ *
$= 4T(n-2)-2-1-(3)$

Put $n = n-2$ in eq (1)
$T(n-2) = 2T(n-3) - 1 - (4)$
Put in eq (3)

$T(n) = 4[2T(n-3)-1]-2-1$
$T(n) = 8T(n-3)-4-2-1$
$\Rightarrow T(n) = 2^n T(n-n) - 2^{n-1} - 2^{n-2}$
$\ldots - 2^2 - 2^1 - 2^0$ [$\because T(0)=1$]

$= 2^n - 2^{n-1} - 2^{n-2} \ldots - 2^1 - 2^0$
$= 2^n - (2^n - 1)$ [$2^{n-1} + 2^{n-2} + \ldots + 2^0 = 2^n - 1$]

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

$T(n) = 1$
$\Rightarrow$ Time complexity is $O(1)$

5.
```
int i=1, s=1;
while(s<=n){
    i++;
    s=s+i;
    printf("#");
}
```

PAGE
DATE

we can define the term 'S' according to the relation $S_i = S_{i-1} + i$
The value of i increases by 1 for each iteration. The value contained in 's' at the $i^{th}$ iteration is the sum of first i positive integers.

If K is the total no. of iterations taken by the program, then loop terminates if: $1+2+\ldots+K = \frac{K(K+1)}{2} > n$

$\Rightarrow$ so $K^2 + K > 2n$

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

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

6.
```
void function (int n)
{
    int i, count = 0;
    for (i=1; i*i<=n; i++)
        count++;
}
```

7.
```
void function (int n)
{ int i, j, k, count = 0;
    for (i=n/2; i<=n; i++)-
```

PAGE _____
DATE _____

```
for (j=1; j<=n; j=j*2)
    for (k=1; k<=n; k=k*2)
        count++;
```

executes log n times

Time complexity = $O(n \log^2 n)$

8.
```
void function (int n)
{
  if (n==1) return; -> Constant time
  for (i=1 to n) -> n times
  {
    for (j=1 to n) -> n times
    {
      printf("*");
    }
  }
  function (n-3);
}
```

Recurrence relation: $T(n) = T(n-3) + n^2$

$\Rightarrow T(n) = \Theta(n^3)$

9.
```
Void function (int n)
{
for (i=1 to n) -> this loop execute n times
{
  for (j=1; j<=n; j=j+i) -> this execute
                            j times with j increase by iterate of i
```

PAGE _____
DATE _____

= Inner loop executes $n/i$ times for each value of $i$.

Its running time is $n \times (\sum_{i=1}^{n} n/i)$

$= O(n \log n)$

10. The asymptotic relationship between these functions $n^k$ and $a^n$ is

$n^k = O(a^n)$ $\quad k >= 1, a > 1$

$n^k \leq c \cdot a^n \quad \forall \; n \geq n_0$

$\Rightarrow \frac{n^k}{a^k} \leq c$

11. Same as ques 5.
i is increasing at the rate of j

$\Rightarrow$ If $k$ is total no. of iterations, while loop terminates if, $0+1+\dots+k$

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

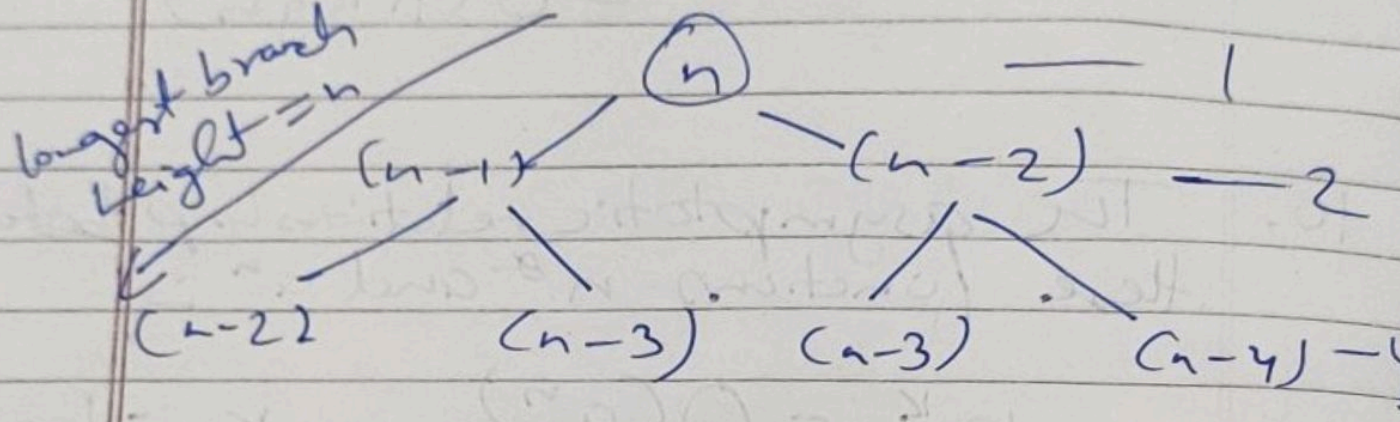
$\Rightarrow k = O(\sqrt{n})$

PAGE
DATE

12. The recurrence relation for the recursive method of fibonacci series is —

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

Solving using tree method —




$$T.C = 1 + 2 + 4 + \cdots + 2^n$$

$a = 1, r = 2$ $\quad S = \frac{a(r^{\text{terms}} - 1)}{r - 1}$

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

$$T.C = O(2^{n+1}) = O(2 \cdot 2^n)$$
$$= O(2^n)$$

Space Complexity $= O(n)$ [∵ Stack size never exceeds the depth of the calls tree shown above]

PAGE _____
DATE _____

13. Programs with Complexity –

i) $n \log n$ –

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

(ii) $n^3$

```
void function (int n)
{
    for(i=1; i<= n; i++)
    {
        for (int j=1; j<=n; j++)
        {
            for (int k=1; k<=n; k++)
            {  printf("#");
            }
        }
    }
```

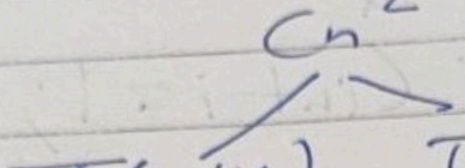
(iii) $\log(\log n)$ → `for (int i=2; i<=n; i = pow(i, k)) {} // O(1)`

Also, interpolation search has this complexity

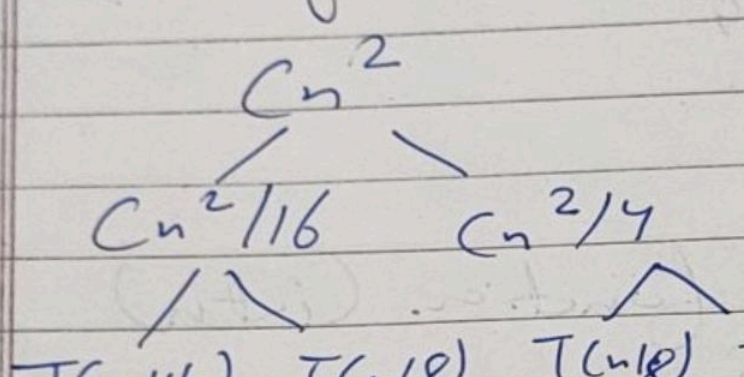
PAGE _____
DATE _____

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

Following is the initial recurssion tree;




On further breaking down,




To know the value of T(n) we need to calculate the sum of tree nodes level by level

$\Rightarrow T(n) = cn^2 + 5n^2/16 + 25n^2/256 + \ldots$

G.P with ratio 5/16

$$S_\infty = \frac{n^2}{1 - 5/16} \Rightarrow T.C = O(n^2)$$

15. Same as ques 9. ($O(n \log n)$)

PAGE _____
DATE _____

16. for (int i=2 ; i<=n ; i=pow(i,k))
{
// O(1) expression }

In this case i takes values $2, 2^k, (2^k)^k, (2^{k^2})^k = 2^{k^3}$ ...

$2^{k^{\log_k(\log_k(n))}}$. The last term must be less than or equal to n, we have

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

∴ There are total $\log_k(\log(n))$ many iterations & each iteration takes constant amount of time to run j.
Total time Complexity = $O(\log(\log n))$

17. The running time when in quick sort when the partition is putting 99% of elements on one side and 1% elements on another in each repetition

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

PAGE _____
DATE _____

20. Pseudo code for iterative insertion sort

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

Pseudo code for recursive insertion sort.

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

PAGE
DATE

```
        arr[j+1] = arr[j];
        j = j-1;
    }
    arr[j+1] = last;
}
```

An online sorting algorithm is one that will work, if the elements to be sorted are provided one at a time with the understanding that the algorithm must keep the sequence sorted as more & more elements are added in. Insertion sort considers one input element per iteration & produces a partial solution without considering future elements. Thus insertion sort is online. Other algorithms like selection sort repeatedly select the minimum element from the unsorted array & places it at the point which requires the entire input. Similarly bubble, quick & merge sorts also require the entire input. Therefore they are offline algorithms for sorting.

PAGE _____
DATE _____

21,22

| Sorting Algo | Best | Avg | Worst | Worst | Stable | Inplace |
|---|---|---|---|---|---|---|
| → Bubble | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | ✓ | ✓ |
| → selection | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | × | ✓ |
| → Insertion | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | ✓ | ✓ |
| → Merge | $O(n\log n)$ | $O(n\log n)$ | $O(n\log n)$ | $O(n)$ | ✓ | × |
| → Quick | $O(n\log n)$ | $O(n\log n)$ | $O(n^2)$ | $O(n)$ | × | × |
| → Heap | $O(n\log n)$ | $O(n\log n)$ | $O(n\log n)$ | $O(1)$ | × | ✓ |

23. Iterative pseudo code for binary search

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

PAGE
DATE

Time complexity :- Best case : O(1)
Avg, worst : O(log2 n)

Space O(1)

## Binary Search Recursive Code

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

    else if (arr[mid] > x)
    return binarySearch (arr, l, mid-1, x);

    else
      return binarySearch (arr, mid+1, r, x);
  }
  return -1;
}
```

T.C => Best : O(1) & Avg, Worst = O(log2 n)

PAGE _____
DATE _____

Space Complexity => Best : O(1)
↓ Avg & worst
programming O(log₂ n)
stack used

24. Recurrence relation for binary recursive search.

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