Design and Analysis of Algorithms Date :

Assignment - 1

Name : Ruchika Mehra University Roll No : 2014817

Section : A

Roll No : 43

1. Asymptotic notation are methods using which we can define the running time of an algorithm based on input size.

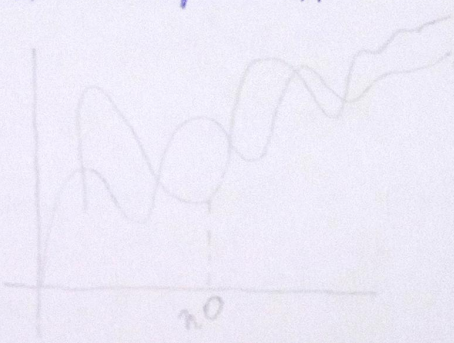
Types of Asymptotic Notation.

Big-O Notation :

Big-O commonly wrote as O, is an asymptotic notation for the worst case of growth for a given function.

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

$g(n)$ is the highest upper bound of $f(n)$.



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

Only if

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

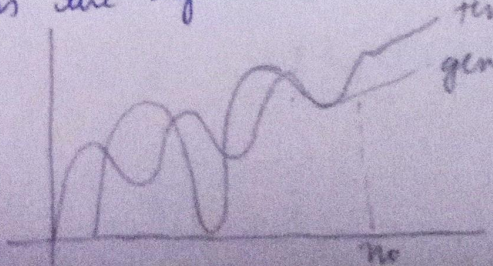
$\forall\ n \geq n_0$

Omega - Ω Notation :

It is the asymptotic notation for the best case for a given function. It gives the "tightest" lower bound of a function.

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

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



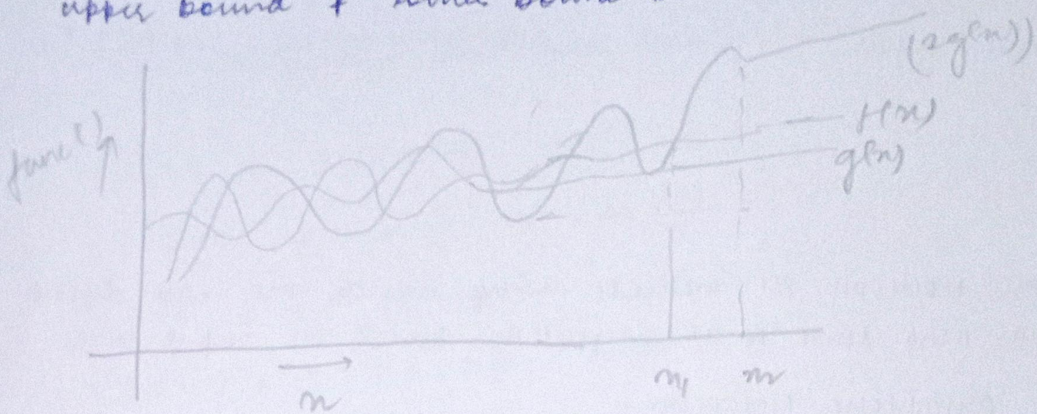
$f(n) = \Omega(g(n))$

if $f(n) \geq (-g(n))$

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

Theta (θ) :

It is an asymptotic notation to denote the tight between the upper bound & lower bound both.

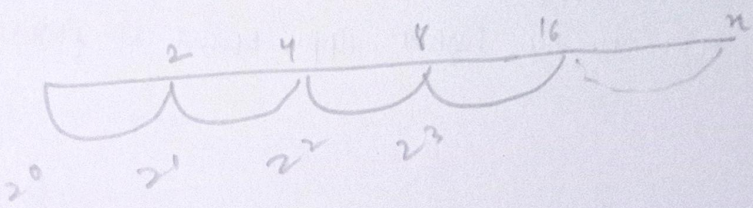


$f(n) = \theta(g(n))$

if $C_1 g(n) \leq f(n) \leq (2 \cdot g(n))$

$\forall \max(n_1, n_2)$

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



GP,

$T_k = a * r^{(k-1)}$

$a = 1$

$r = \frac{t_2}{t_1} = \frac{2}{1} = 2$

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

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

$n = \frac{2^k}{2^1}$

$2n = 2^k$

using log property

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

$1 \quad \boxed{k = \log(n)}$

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

Using ~~background~~ Backward Substitution,

$T(n) = 3T(n-1)$ —①

Put $n = n-1$ in eq ①

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

$T(n-1) = 3T(n-2)$ —②

putting the value of ② in ①

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

$T(n) = 9T(n-2)$ —③

putting $n = n-2$ in eq ①

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

$T(n-2) = 3T(n-3)$ —④

putting the eq ④ in eq ③

$T(n) = 9 \times 3\, T(n-3)$ —⑤

$n = n-4$ in eq ①

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

$T(n-3) = 3T(n-4)$ —⑥

$T(n) = \frac{27 \times 3\, T(n-4)}{3^4}$

$T(n-4)$ —ii

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

Let $k = (n-1)$

$$T(n) = 3^{(n-1)}\, T(n-(n-1))$$

$$T(n) = 3^n \cdot 3^{-1}\, T(n-n+1)$$

$$T(n) = \frac{3^n}{3}\, T(1)$$

$$T(1) = 1$$

$$\boxed{T(n) = 3^n}$$

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

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

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

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

now put eq (2) in eq (1)

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

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

$n = n-2$

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

$$= 2^3 T(n-4) - 2^2 - 2^1 - 2$$

$$= 2^k T(n-k) - 2^{k-1} - 2^{k-2} - 2^{k-3} \cdots 2^{k-k}$$

$$= 2^k - 2^{k-1} - 2^{k-2} \cdots$$

$$= 2^k - (2^k - 1)$$

$$= 1$$

time complexity = $O(1)$

5. $O(\sqrt{n})$

6. $O(\sqrt{n})$

7. $O(n(\log^2(n))$

8. $O(n)$

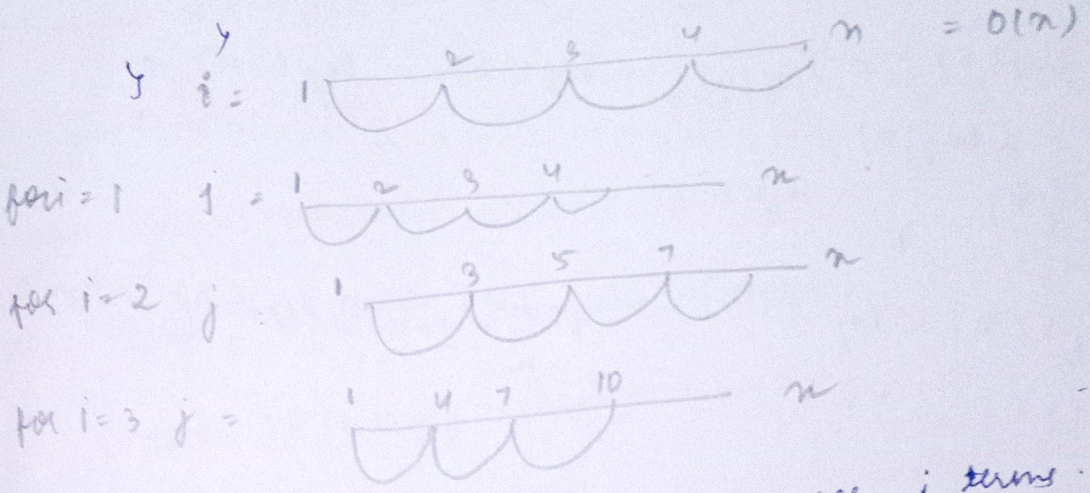
9. $O(n \log n)$

10. For functions n^k and a^n, the asymptotic relation is n^k is $O(a^n)$ ie time complexity of n^k is of order $O(n^k)$.

11.
```
void func(int n) {
int i = 1, i = 0;
while (i<n)
  {
  i = i+i;
  j++;
  }
}
```

ans 15.

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



each run of i loop forms an AP for j terms.

$O(n-i) = O(n)$

so

$O(n) \times O(n) = O(n^2)$

16. `for ( int i = 2 ; i <= n ; i = pow(i, k))`

```
{
  // some O(1) expressions or statements
}
```

where, k is constant:

$i = 2, 2^k, 2^{k^2} \ldots \underbrace{2^{k(\log_k(\log(n)))}}_{= 2^{\log n} = n}$

Total no. of iterations - $\log k(\log(n))$

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

18.

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

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

c) $9l < \log 8(n) < \log_2(n) < n\log_c(n) < n\log_2(n) < 5n < 8n^2 < \log(n!) < 7n^3 < n! < 8^{2n}$

20. Pseudo code for iterative insertion sort.

```
void insertionsort (int arr[], int n)
{
for i = 1 to n
{
temp = arr[i]
j = i - 1
while j >= 0 and arr[j] > temp
        arr[j+1] = arr[j]
        j = j - 1
End while
arr[j+1] = temp ;
End for.
```

22. Sorting Algorithm

Algorithm	Time complexity			Space complexity
	Best	Average	Worst	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	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$

22.

(Inplace)
1. selection sort
2. Bubble sort
3. Heap sort
4. Insertion sort
5. Quick sort

(Stable)
1. Merge sort
2. Insertion sort
3. Bubble sort

(Online Sorting Algorithm)
1. Insertion sort.

Recursive Insertion sort pseudo code

```
insertionsort ( arr[] , n)
{
  if (n <= 1)
    {return ;
    insertionsort ( arr, n-1)
      last = arr[n-1]
      j = n-2
  while ( j >=0 and arr[j] > last
    {
      arr[j+1] = arr[j]
        j = j-1
  End while
    arr[j+1] = last
}
End ~~loop~~ function
```

Insertion sort is also called online sorting because.

23.

Pseudo code for iterative Binary search.

```
int Binarysearch ( int A[], int n, int key)
{
    int low = 0;
    int high = n-1;
    while ( low <= high)
    {
     mid = low + (high - low)/2;
    if (A[mid] == key)
      {
       return mid;
      }
    else if ( A[mid] < key)
    {
      low = mid + 1;
    }
    else {
      high = mid - 1;
    }
    return -1;
  }
```

Time complexity of Linear search : $O(n)$

Time complexity of Binary search [iterative] : $O(\log(n))$

Time complexity of Binary search [recursive] : $O(\log(n))$

Space complexity of Linear search: $O(1)$

Space complexity of Binary search: $O(1)$

24. Recurrence relation for binary recursive search.

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

$$= T(n/2) + C$$

So, $T(n) = T(n/2) + C$

This can be solved using master's method.

case 2: time complexity : $O(\log n)$