PAGE No
DATE

Assignment-1

Deepak Melkani

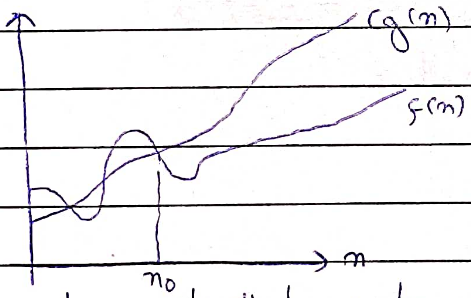
1). Asymptotic Notation

These notation are used to tell the complexity of an algo when the input is very large.

There are mainly 3 asymptotic notation :-

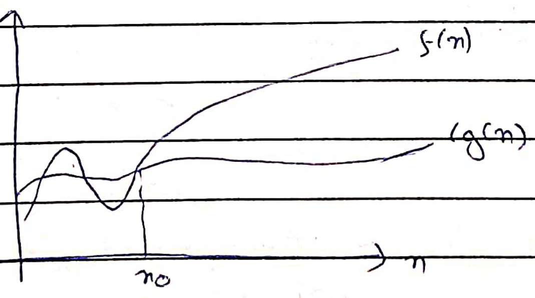
i). Big-O notation → it represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

$O(g(n)) = \{ f(n) :$ there exist +ve constants c & n_0 such that $0 \le f(n) \le c g(n)$ for all $n \ge n_0 \}$



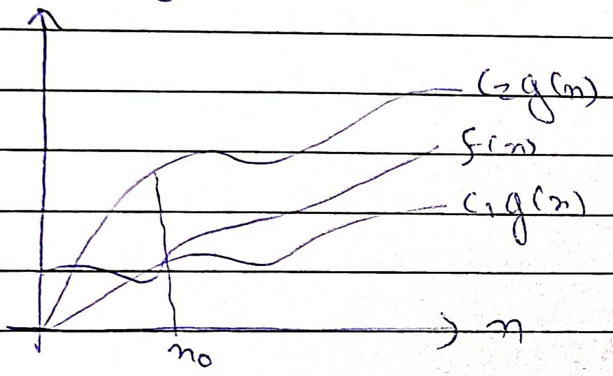
2). Omega notation → it represents the lower bound of the running time of an algo. Thus, it provides the best case complexity of an algorithm.

$\Omega(g(n)) = \{ f(n) :$ there exist +ve constants c & n_0 such that $0 \le c g(n) \le f(n)$ for all $n \ge n_0 \}$



1). Theta notation → it encloses the function from above & below. Since, it represents the upper & the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

$\Theta(g(n)) = \{ f(n)$: there exist +ve constants $c_1, c_2, \& n_0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0 \}$



2). $O(n \log_2 n)$

3). $T(n) = 3T(n-1)$
$T(0) = 1$

$T(n) = 3T(n-1)$
$T(n) = 3(3T(n-2)) = 3^2 \times T(n-2)$
$T(n) = 3^3 \times T(n-3)$
⋮
$T(n) = 3^n \times T(n-n)$
$= 3^n \times T(0)$
$= 3^n$
$\Rightarrow O(3^n)$

4). $T(n) = 2T(n-1) - 1$
$T(0) = 1$

$T(0) = 1$
$T(1) = 2T(1-1) - 1$
$= 2 - 1 = 1$
$T(2) = 2T(2-1) - 1 = 1$

$T(n) = O(1)$

5).
```
int i=1, s=1;
while (s <= n)
  { i++;
    s=s+i;
    print("#");
  }
```
[[illegible] i=1, s=1 & i=2, s=3 & i=3, s=6 & ...]

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

$O(\sqrt{n})$

6).
```
void function(int n)
{ int i, Count=0;
    for(i=1; i*i<=n; i++)
        Count++;
}
```
$\Rightarrow O(\sqrt{n})$

7).
```
void function (int n)
{ int i, j, k, count=0;
    for(i=n/2; i<=n; i++)                → O(n)
      for(j=1; i<=n; j=j*2)              → O(log_2 n)
        for(k=1; k<=n; k=k*2)            → O(log_2 n)
            Count++;
}
```
$\Rightarrow O(n(\log_2 n)^2)$

PAGE No
DATE

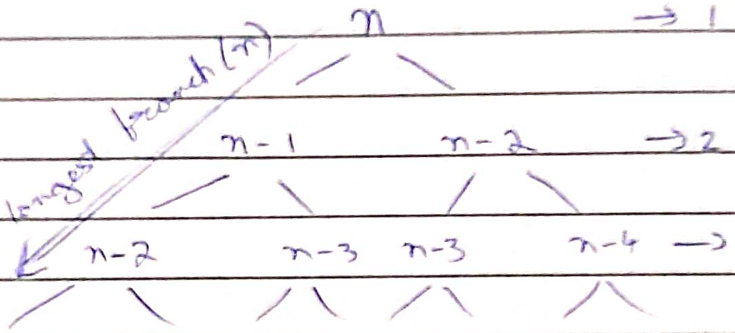
8).
```
function (int n)          → T(n)
{
    if (n == 1) return;
    for(i = 1 to n){          → n^2
        for(j = 1 to n){
            print ("*");                 ⇒ O(n^3)
        }
    }
    function (n-3);   → T(n-1)
}
```

9).
```
void function (int n)
{
    for(i=1 to n){
        for(j=1; j<=n; j=j+i)           ⇒ O(n log n)
            print ("*")
    }
}
```

10). n^k is O(c^n)

11).
```
void fun(int n){
    int j=1, i=0;              0, 3, 6, 10, 15, ---- n
    while (i<n){                                      ⇓
                                                      k-th term
        i = i+j;               k-th terms  k(k+1)/2
        j++; }                      k = √n
}
                               ⇒ O(√n)
```

12).
```
int fib (int n)                 ← T(n)
{
    if (n <= 1)                 ]→ constant time operation O(1)
        return n;
    return fib(n-1) + fib(n-2);
}
```



$1+2+4+ \ldots + 2^n$ (G.P.)

$a=1$

$r=2$

$\Rightarrow a\left[\frac{r^{\text{terms}}-1}{r-1}\right]$

$1 \times \left[\frac{2^n-1}{2-1}\right]$

$\Rightarrow O(2^n)$

→ For fibonacci recursive implementation / any recursive algo, the space required is proportional to the maximum depth of the recursion tree because that is the maximum number of elements that can be present in the implicit function call stack.

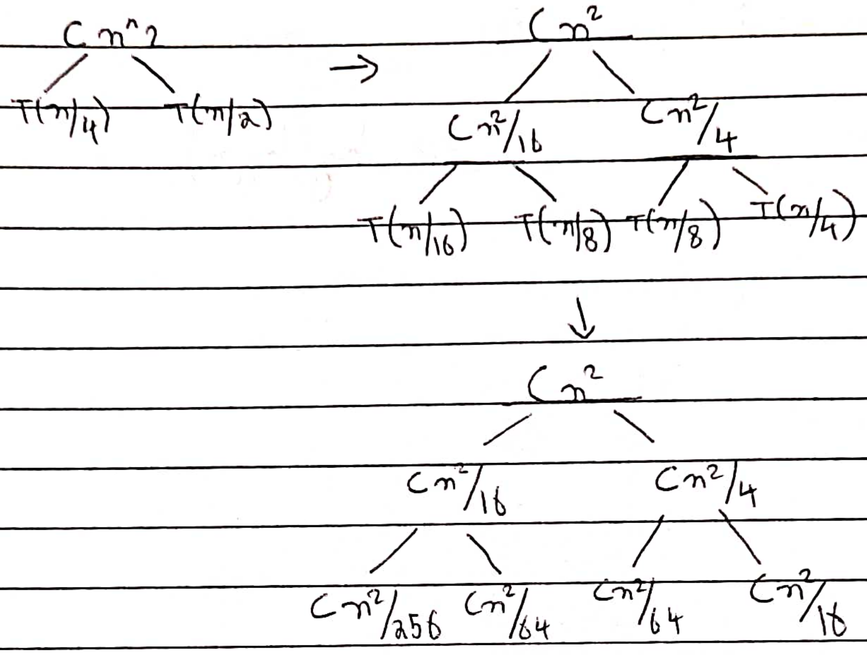
$O(n)$

13).
```
int count=0;
for(int i=0 to n, i++)
 {for(int j=0 to n, j++)
   { for(int k=0 to n, k++)
     { count++;
     }}}
```
$O(n^3)$

```
int count=0;
 for(int i=0 to n, i++)
  {for(int j=0 to n, j*=2)
    { count++;
    }}
```
$O(n \log_2 n)$

```
for (int i=n; i>1; i=fun(i))
  {  // some O(1)
  }
```
$\log(\log n)$

14). $T(n) = T(n/4) + T(n/2) + Cn\hat{}2$



$$T(n) = Cn^2 + \frac{5Cn^2}{16} + \frac{25Cn^2}{256} + \text{---}$$

$$= C\left[n^2 + \frac{5}{16}n^2 + \frac{25}{256}n^2 + \text{---}\right]$$

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

15). int fun (int n) $\rightarrow T(n)$

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

$i=1$, $j=n$ times

$i=2$, $j=n/2$ times [approx]

$\Rightarrow$ $i=3$, $j=n/3$ times ,,

$= n\left[1 + 1/2 + 1/3 + \text{---}\right]$

$- \log n$

$\Rightarrow O(n \log n)$

PAGE No

DATE

16) · [illegible]

```
for( int i=2; i <=n; i = pow(i,k))
{
    // Some O(1)                    ⇒ log(log n)
}.
```

17) ·

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). $O(96) < O(\log_8 n) < O(\log_2 n) < O(\log n!) < O(n\log_6 n) < O(n\log_2 n) < O(5n) <$

$O(8n^2) < O(7n^3) < O(n!) < O(8^{2n})$

19).
```
void linearSearch (int arr[], int n, int Key)
{ for(i=0 to n, i++)
        if (arr[i]==key)
            cout<<"found";
        else
            continue;
}
```

20). Iterative insertion Sort:
```
void InsertionSort (int arr, int n)
{ for(int i=1 to n, i++)
    { temp = arr[i];
        j=i-1;
        while(j>=0 && arr[j] > temp)
        { arr[j+1] = arr[j];
            j--;
        }
        arr [j+1]= temp;
}}
```

Recursive Insertion Sort :-

```
insertionSort( arr, n)
{
    if n <= 1
        return;
    insertionSort (arr, n-1);
    last = arr[n-1];
    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 it don't know the whole input, it might make decision that later turn out to be not optimal.

other algo, are off-line algo's that are discussed in lectures.

21).

	Time Complexity			Space Complexity
	Best	Average	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	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
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	online sorting
Bubble Sort	✓	✓	✗
Selection Sort	✓	✗	✗
Insertion Sort	✓	✓	✓
Merge Sort	✗	✓	✗
Quick Sort	✓	✗	✗
Heap Sort	✓	✗	✗

23). Iterative binary search:-

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

~~[illegible]~~	time Complexity			Space Complexity
	best	worst	average	
linear Search	O(1)	O(n)	O(n)	O(1)
binary Search	O(1)	O(log n)	O(log n)	O(1)

24) → $T(n) = T(n/2) + 1$