Algorithm Complexity and Main Concepts

Lecture No. 3

Performance Analysis

- Performance of an algorithm is a process of making evaluative judgement about algorithms.
- Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.
- The complexity of an algorithm is a function f (n) which measures the **time** and **space** used by an algorithm in terms of input size n.

Complexity

- Complexity of an algorithm is a measure of the amount of **time** and **space** required by an algorithm for an input of a given size (n).

Space Complexity	**Time Complexity**
It is the amount of space (or memory) taken by the algorithm to run as a function of its input length, *n*	It is the amount of time taken by the algorithm to complete its process as a function of its input length, *n*.

Space Complexity

Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.

Space Complexity = Fixed part + Variable part

where,

Fixed part :- It is a space required to store certain data and variables that are not dependent on the size of the problem (i.e. simple variables and constants)

Variable part:- It is a space required by variables, whose size is totally dependent on the size of the problem (Recursion stack space, dynamic memory allocation).

Examples of Space Complexity

Constant Space Complexity

```
int main()
{
 int a = 5, b = 5, c;
 c = a + b;
 printf("%d", c);
}
```

3 variables of integer type (4 bytes)
The total space occupied by the above-given program is 4 * 3 = 12 bytes

Linear Space Complexity

```
int main()
 {
  int n, i, sum = 0,  arr[n];
  scanf("%d", &n);
  for(i = 0; i < n; i++)
  {
    scanf("%d", &arr[i]);
    sum = sum + arr[i];
  }
 printf("%d", sum);
}
```

total space occupied by the program is 4n + 12 bytes or **O(n) i.e. linear**

Time Complexity

The time complexity is the number of operations an algorithm performs to complete its task.

Note:- Time Complexity of algorithm is **not** equal to the actual time required to execute a particular code but the number of times a statement executes.

1) Statements;

   example:- a= c+10;

   Time complexity is **Constant** i.e. **O(1)**

```
for (i=0 ; i<n ; i++)
{
 statement ;
}
```

```
for (i=n ; i>0 ; i--)
{
  statement ;
}
```

```
for(i=0 ; i<n ; i= i+2)
 {
    statement;
 }
```

The loop is directly proportional to n.

So, Time complexity will be **Linear** i.e **O(n)**

```
for (i=0 ; i<n ; i++)
{
  for (j=0 ; j<n ; j++)
  {
    statement;
  }
}
```

```
for (i=0 ; i<n ; i++)
{
   for (j=0 ; j<i ; j++)
   {
     statement;
   }
}
```

i	j	No. of times
0	0	0
1	0	1
	1	
2	0	1
	1	2
	2	
3	0	1
	1	2
	2	3
I		I
I		I
I		I
n		n

$1+2+3+4+\text{----}+n = \frac{n\,(n+1)}{2}$

$= \frac{n^2 + n}{2}$

Time complexity will be **Quadratic** i.e. **$O(n^2)$**

```
P =0;
for(i=1 ; P<=n ; i++)
{
  P=P+i;
}
```

Assume P>n

$P = k(k+1)/2$

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

$k^2 > n$

$k > \sqrt{n}$

i	No. of times (P)
1	0+1=1
2	1+2=3
3	3+3=6
4	6+4=10
I	I
I	I
I	I
k	k

$1+2+3+4+\text{----}+k = \frac{k(k+1)}{2}$

Time Complexity will be $O(\sqrt{n})$

```
for(i=1; i<n ; i= i*2)
{
   statement ;
}
```

Assume $i>=n$

w.k.t $i = 2^k$

$2^k >= n$

$2^k = n$

$k = \log_2 n$

Time Complexity will be $O(\log_2 n)$

i
1
$1*2=2$
$2*2=2^2$
$2^2*2=2^3$
$2^3*2=2^4$
I
I
I
2^k

```
P=0;
for(i=1; i<n; i=i*2)
{
 P++;
}
for(j=1; j<P; j=j*2)
{
 statement;
}
```

Time Complexity will be $O(\log \log_2 n)$

```
for(i=0; i<n; i++)
{
  for(j=1; j<n; j=j*2)
  {
     statement;
  }
}
```

Time complexity will be $O(n \log_2 n)$

Any Query ?