# Data Structures and Algorithms

Module-1

## Algorithm

An algorithm is a step-by-step procedure to solve a problem. A good algorithm should be optimized in terms of time and space.

Different types of problems require different types of algorithmic-techniques to be solved in the most optimized manner.

**Brute Force Algorithm:** This is the most basic and simplest type of algorithm. It is just like iterating every possibility available to solve a problem.

## Types of Algorithms

- Recursive Algorithm: In recursion, a problem is solved by breaking it into subproblems of the same type and calling own self again and again until the problem is solved with the help of a base condition.

  Ex: Factorial of a Number, Fibonacci Series, Tower of Hanoi, DFS

- Divide and Conquer Algorithm: The idea is to solve the problem in two sections, 1) divides the problem into subproblems of the same type. 2) solve the smaller problem independently and then add the combined result to produce the final answer to the problem.

  Ex: Binary Search, Merge Sort, Quick Sort

## Types of Algorithms

- Dynamic Programming Algorithms: The idea is to store the previously calculated result to avoid calculating it again and again. In Dynamic Programming, divide the complex problem into smaller overlapping subproblems and storing the result for future use.

  Ex: Knapsack Problem, Weighted Job Scheduling

- Greedy Algorithm: The solution is built part by part. The decision to choose the next part is done on the basis that it gives the immediate benefit.

  Ex: Dijkstra's Algorithm

## Types of Algorithms

- Randomized Algorithm: It uses a random number. It helps to decide the expected outcome. The decision to choose the random number so  it gives the immediate benefit.

  Ex: Quicksort

- Sorting Algorithm: The sorting algorithm is used to sort data in maybe ascending or descending order. Its also used for arranging data in an efficient and useful manner.

  Ex: Bubble sort, insertion sort, merge sort, selection sort

## Types of Algorithms

- Searching Algorithm: The searching algorithm is the algorithm that is used for searching the specific key in particular sorted or unsorted data.

  Ex: Binary search or linear search

- Hashing Algorithm: Hashing algorithms work the same as the Searching algorithm but they contain an index with a key ID. In hashing, we assign a key to specific data.

  Ex: Double Hashing, Quadratic Probing

## Analysis of Algorithms

What is the goal of analysis of algorithms?

- To compare algorithms mainly in terms of running time but also in terms of other factors (e.g., memory requirements, programmer's effort etc.)

What do we mean by running time analysis?

- Determine how running time increases as the size of the problem increases.

## Types of Analysis

Worst case

- Provides an upper bound on running time
- An absolute guarantee that the algorithm would not run longer, no matter what are the inputs.

Best case

- Provides a lower bound on running time.
- Input is the one for which the algorithm runs the fastest.

Average case

- Provides a prediction about the running time.
- Assumes that the input is random.

## Types of Analysis

Worst case

- Provides an upper bound on running time
- An absolute guarantee that the algorithm would not run longer, no matter what are the inputs.

Best case

- Provides a lower bound on running time.
- Input is the one for which the algorithm runs the fastest.

Average case

- Provides a prediction about the running time.
- Assumes that the input is random.

## Analysis of Algorithms

### Algorithm

- It is a finite set of instructions to solve a problem or to perform any computation.

### Analysis of algorithm

- Finding the best algorithm among all possible solutions.

### Factors to Analyse an algorithm

- Running time
- Memory space required
- Programmers effort

## Asymptotic Notations

- O notation: asymptotic "less than": (Big OH Notation)

  $f(n)=O(g(n))$ implies: $f(n)$ "$\leq$" $g(n)$

- Ω notation: asymptotic "greater than": (Big OMEGA Notation)

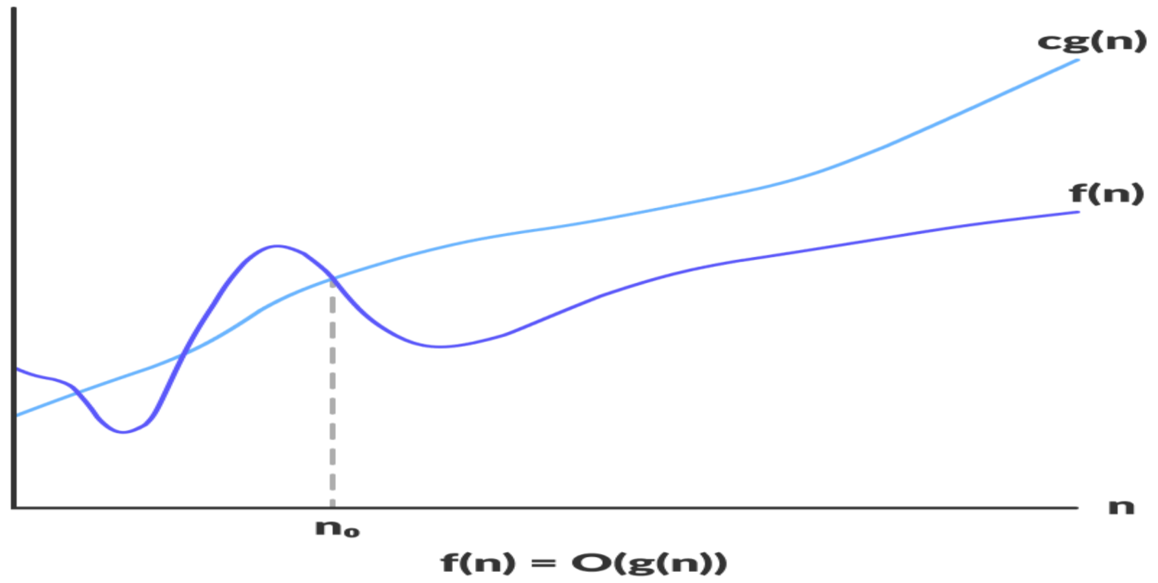
  $f(n)= \Omega (g(n))$ implies: $f(n)$ "$\geq$" $g(n)$

- Θ notation: asymptotic "equality": (Big THETA Notation)

  $f(n)= \Theta (g(n))$ implies: $f(n)$ "$=$" $g(n)$

## Big-Oh Notation

**Definition:** The function $f(n) = O\big(g(n)\big)\ iff\ \exists$ a positive constant $c\ and\ n_0$ such that $f(n) \leq c * g(n))\ \forall\ n \geq n_0$.



## Big-Oh Notation Example

Note: $1 < logn < \sqrt{n} < n < nlogn < n^2 < n^3 < \cdots < 2^n < 3^n < \cdots < n^n$

Ex: $f(n) = 2n + 3$

$2n + 3 \leq 5n$ where, $2n + 3$ is $f(n)$, $c\ is$ 5 and $g(n)\ is\ n\ \ \forall\, n \geq 1$.

Therefore $f(n) = Big\ Oh(n)$

We can also write

$2n + 3 \leq 5n^2$ which is also true $\forall\, n \geq 1$ then $f(n) = Big\ Oh(n^2)$.

Both the above cases are true but we just need to consider the nearest upper bound value for our function as it should be useful.

## Big-Oh Notation Examples

- Consider $f_A(n)=30n+8$. Represent it in Big-Oh Notation.
- Consider $f_B(n)=n^2+1$. Represent it in Big-Oh Notation.
- Consider $n^4 + 100n^2 + 10n + 50$. Represent it in Big-Oh Notation.
- Consider $10n^3 + 2n^2$. Represent it in Big-Oh Notation.

## Big-Oh Notation Examples

**Example 4.9:** $5n^2 + 3n\log n + 2n + 5$ *is* $O(n^2)$.

**Justification:** $5n^2 + 3n\log n + 2n + 5 \leq (5+3+2+5)n^2 = cn^2$, for $c = 15$, when $n \geq n_0 = 1$. ■

**Example 4.10:** $20n^3 + 10n\log n + 5$ *is* $O(n^3)$.

**Justification:** $20n^3 + 10n\log n + 5 \leq 35n^3$, for $n \geq 1$. ■

**Example 4.11:** $3\log n + 2$ *is* $O(\log n)$.

**Justification:** $3\log n + 2 \leq 5\log n$, for $n \geq 2$. Note that $\log n$ is zero for $n = 1$. That is why we use $n \geq n_0 = 2$ in this case. ■

## Big-Oh Notation Examples

**Example 4.12:** $2^{n+2}$ *is* $O(2^n)$.

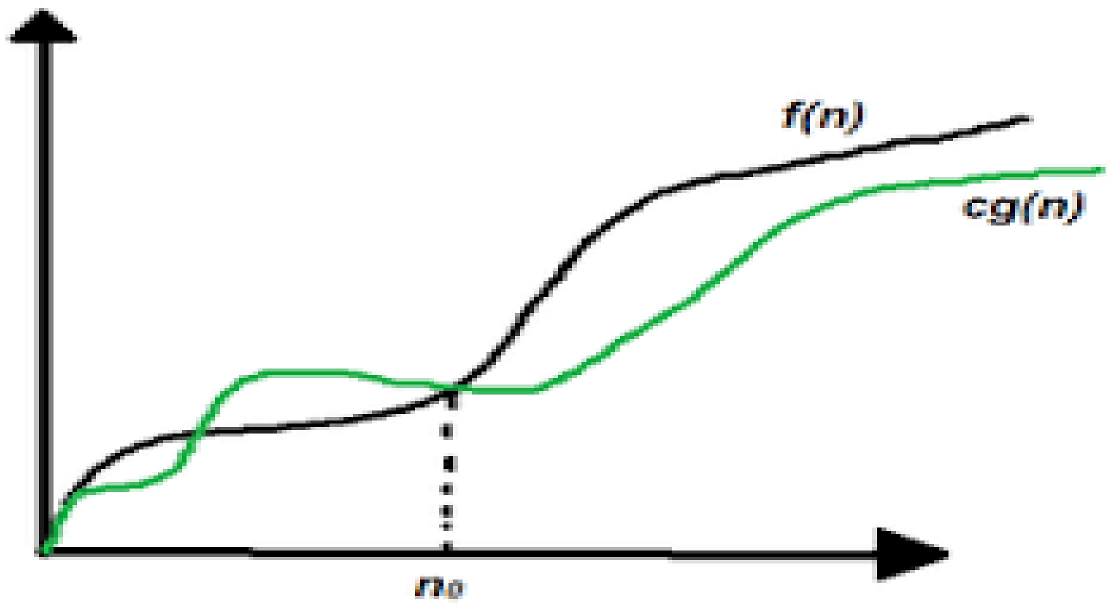
**Justification:** $2^{n+2} = 2^n \cdot 2^2 = 4 \cdot 2^n$; hence, we can take $c = 4$ and $n_0 = 1$ in this case. ■

**Example 4.13:** $2n + 100\log n$ *is* $O(n)$.

**Justification:** $2n + 100\log n \leq 102n$, for $n \geq n_0 = 1$; hence, we can take $c = 102$ in this case. ■

## Big-Omega Notation

- **Definition:** The function $f(n) = O\big(g(n)\big)\ iff$ $\exists$ a positive constant $c\ and\ n_0$ such that $f(n) \leq c * g(n))$ $\forall\, n \geq n_0$.



## Big-Omega Notation Example

- Prove that $3n + 2 = \Omega(n)$

Sol: As per the definition of Big Ω $cg(n) \leq f(n)$

$$cn \leq 3n + 2$$

$$cn - 3n \leq 2$$

$$n(c - 3) \leq 2$$

$$n \leq \frac{2}{c-3}$$

if we assume $c = 4$ then $n_0 = 2$

## Big-Omega Notation Example

- Prove that $5n^2 = \Omega(n)$

Sol: As per the definition of Big $\Omega$ $cg(n) \le f(n)$

$$cn \le 5n^2$$
$$c \le 5n$$

if we assume $n_0 = 1$ then $c = 5$

## Big-Omega Notation Example

- Prove that $5n^2 + 2n - 3 = \Omega(n^2)$

Sol: As per the definition of Big Ω $cg(n) \leq f(n)$

$$cn^2 \leq 5n^2 + 2n - 3$$

Need to prove $2n - 3 \geq 0$

$$2n \geq 3$$

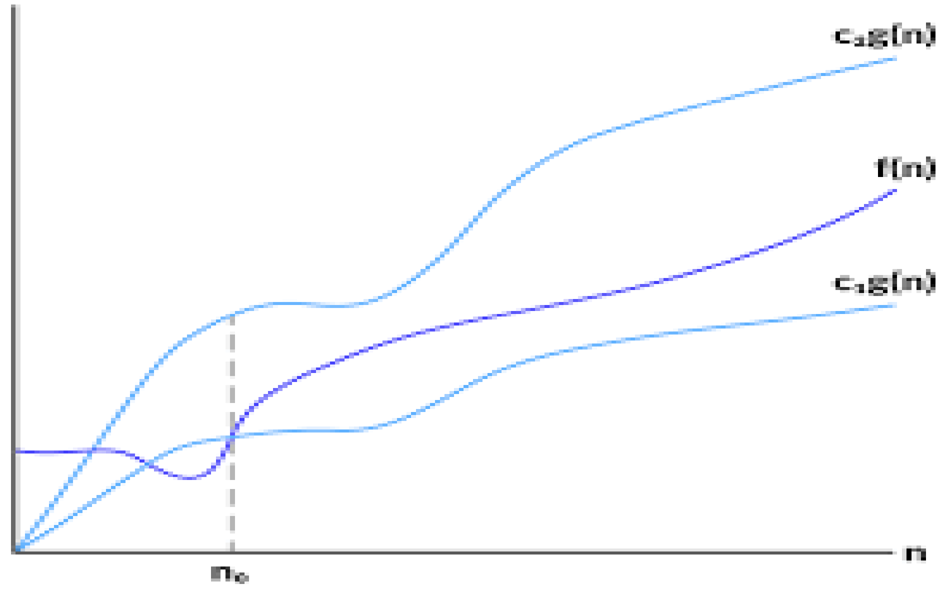
$$n \geq \frac{3}{2}$$

$$n \geq 2$$

if we assume $\mathrm{n}_0 = 2$ then $\mathrm{c} = 5$ hence it is proved.

# Theta Notation

**Definition:** The function $f(n) = \Theta\big(g(n)\big)$ $iff$ $\exists$ a positive constants $c_1, c_2$ $and$ $n_0$ such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)\ \forall\ n \geq n_0$.



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

## Theta Notation Example

**Ex:** Assume $f(n) = 3n^3 + 6n + 7$, prove that $f(n) = \Theta(n^3)$

Sol: As per the definition of $\Theta\ notation\ c_1 g(n) \leq f(n) \leq c_2 g(n)$

$$3n^3 \leq 3n^3 + 6n + 7 \leq 3n^3 + 6n^3 + 7n^3$$

$$3n^3 \leq 3n^3 + 6n + 7 \leq (3+6+7)n^3$$

$$3n^3 \leq 3n^3 + 6n + 7 \leq 16n^3$$

$As\ g(n) = n^3$ $\quad 3g(n) \leq f(n) \leq 16g(n)$

$$c_1 = 3, c_2 = 16, n_0 = 1$$

## Theta Notation Example

**Ex:** Assume $f(n) = 5n^3 + 16n^2 + 3n + 8$, prove that $f(n) = \Theta(n^3)$

Sol: As per the definition of $\Theta$ *notation* $c_1 g(n) \leq f(n) \leq c_2 g(n)$

$$5n^3 \leq 5n^3 + 16n^2 + 3n + 8 \leq 5n^3 + 16n^2 + 3n^3 + 8n^3$$

$$5n^3 \leq 5n^3 + 16n^2 + 3n + 8 \leq (5+16+3+8)n^3$$

$$5n^3 \leq 5n^3 + 16n^2 + 3n + 8 \leq 32n^3$$

*As* $g(n) = n^3$ $\quad 5g(n) \leq f(n) \leq 32g(n)$

$$c_1 = 5, c_2 = 32, n_0 = 1$$

## Theta Notation Example

**Ex:** Assume $f(n) = 6nlogn + 7n + 5$, prove that $f(n) = \Theta(nlogn)$

Sol: As per the definition of $\Theta\ notation\ c_1 g(n) \le f(n) \le c_2 g(n)$

$$6nlogn \le 6nlogn + 7n + 5 \le 6nlogn + 7nlogn + 5nlogn$$

$$6nlogn \le 6nlogn + 7n + 5 \le (6+7+5)nlogn$$

$$6nlogn \le 6nlogn + 7n + 5 \le 18nlogn$$

$As\ g(n) = nlogn$ $\qquad 6g(n) \le f(n) \le 18g(n)$

$$c_1 = 6, c_2 = 18, n_0 = 2$$

## Theta Notation Example

**Ex:** Assume $f(n) = 3nlogn + 4n + 5logn$, prove that $f(n) = \Theta(nlogn)$

Sol: As per the definition of $\Theta\ notation\ c_1 g(n) \leq f(n) \leq c_2 g(n)$

$$3nlogn \leq 3nlogn + 4n + 5logn \leq 3nlogn + 4nlogn + 5nlogn$$

$$3nlogn \leq 3nlogn + 4n + 5logn \leq (3+4+5)\,nlogn$$

$$3nlogn \leq 3nlogn + 4n + 5logn \leq 12nlogn$$

$As\ g(n) = nlogn$ $\qquad 3g(n) \leq f(n) \leq 12g(n)$

$$c_1 = 3, c_2 = 12, n_0 = 2$$

## Theta Notation Example

**Ex:** Assume $f(n) = 10n + 7$, prove that $f(n) = \Theta(n)$

Sol: As per the definition of $\Theta\ notation\ c_1 g(n) \leq f(n) \leq c_2 g(n)$

$$10n \leq 10n + 7 \leq 10n + 7n$$

$$10n \leq 10n + 7 \leq (10+7)n$$

$$10n \leq 10n + 7 \leq 17n$$

$As\ g(n) = n$ $\quad$ $10g(n) \leq f(n) \leq 17g(n)$

$$c_1 = 10, c_2 = 17, n_0 = 1$$

## Theta Notation Example

**Ex:** Assume $f(n) = 12n + 6$, prove that $f(n) = \Theta(n)$

Sol: As per the definition of $\Theta$ *notation* $c_1 g(n) \leq f(n) \leq c_2 g(n)$

$$12n \leq 12n + 6 \leq 12n + 6n$$

$$12n \leq 12n + 6 \leq (12+6)n$$

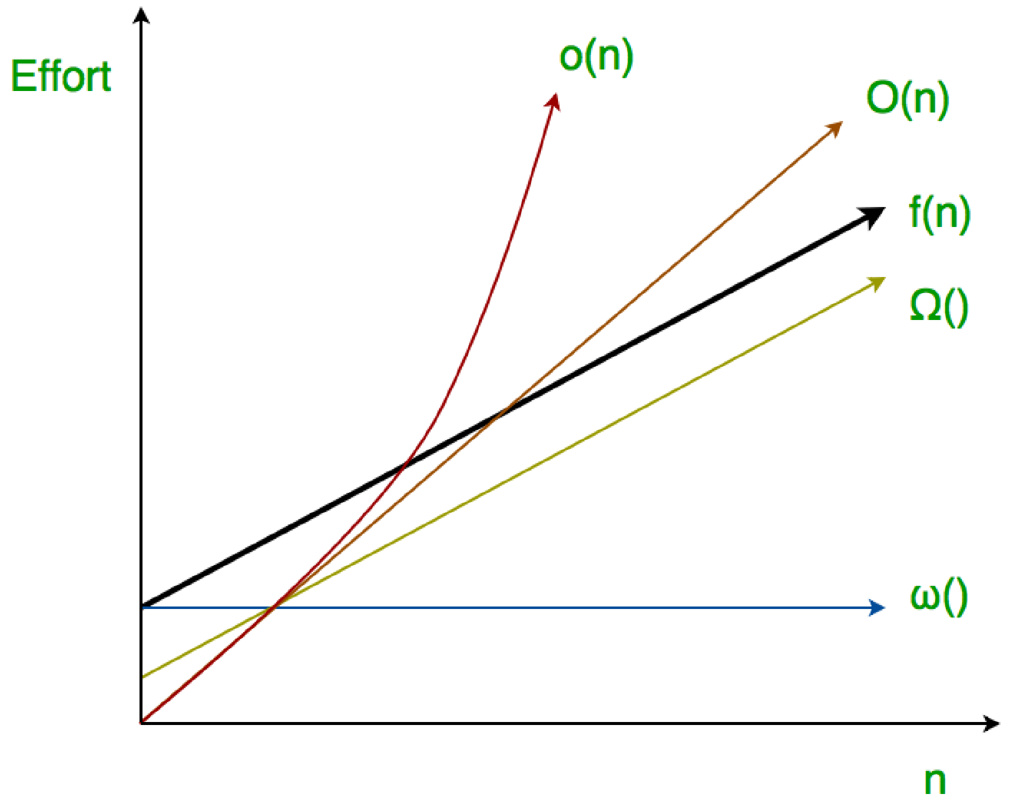
$$12n \leq 12n + 7 \leq 18n$$

$As\ g(n) = n$ $\qquad 12g(n) \leq f(n) \leq 18g(n)$

$$c_1 = 12, c_2 = 18, n_0 = 1$$

# Little Oh Notation

**Definition:** Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is o(g(n)) (or f(n) E o(g(n))) if for **any real** constant c > 0, there exists an integer constant n0 ≥ 1 such that 0 ≤ f(n) < c*g(n)



# Little Oh Notation

Thus, little o() means **loose upper-bound** of f(n). Little o is a rough estimate of the maximum order of growth whereas Big-O may be the actual order of growth. In mathematical relation, f(n) = o(g(n)) means lim  f(n)/g(n) = 0 n→∞

$f(n) = 7n + 8\ and\ g(n) = n^2$ prove that $f(n) = o(n^2)$

As per definition $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 0$

$$\lim_{n\to\infty} \frac{7n+8}{n^2}$$
$$\lim_{n\to\infty} \frac{7}{n} + \frac{8}{n^2}$$

When $n \to \infty$ then $\frac{f(n)}{g(n)} = 0$

# Little Omega Notation

- Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is ω(g(n)) (or f(n) ∈ ω(g(n))) if for any real constant c > 0, there exists an integer constant n0 ≥ 1 such that f(n) > c * g(n) ≥ 0 for every integer n ≥ n0.
- The relationship between Big Omega (Ω) and Little Omega (ω) is similar to that of Big-O and Little o except that now we are looking at the lower bounds.

- Little Omega (ω) is a rough estimate of the order of the growth whereas Big Omega (Ω) may represent exact order of growth.

In mathematical relation, $if\ f(n) =$ ω (g(n)) then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

## Little omega Notation Example

- **Prove that 4n + 6 ∈ ω(1)**

$$f(n) = 4n + 6 \ and \ g(n) = 1$$

As per definition $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

$$\lim_{n\to\infty} \frac{4n+6}{1} = \infty$$

When $n \to \infty$ then $\frac{f(n)}{g(n)} = \infty$

## Time complexity Calculation

- Time complexity is calculated by using step count in the algorithm.
- Step Count is measured in two ways.

➢ Count Method

➢ Frequency Method.

## Time complexity Calculation(Count Method)

- Globally initialize a variable count ← 0
- For each and every valid step of the algorithm increment the count variable by 1.
- Valid step means the step which is useful for execution.

## Time complexity Calculation(Count Method) Example

```
Algorithm sum(x , n)
{
Total = 0;
for i ← 1 to n do
Total = Total + i;
}
```

## Time complexity Calculation(Count Method) Example

```
Algorithm sum(x , n)
{
Total = 0;
count ← count + 1
for i ← 1 to n do
{
count ← count + 1
Total = Total + i;
count ← count + 1
}
count ← count + 1
}
```

- Outside for loop we have 2 Count variables. It will be iterated for 2 times
- Inside the for loop we have 2 Count variables.
- For loop will be iterated for n number of times
- Therefore inside the for loop time complexity is 2n

- Total time complexity is of 2n+2

## Time complexity Calculation(Count Method) Example

```
Algorithm add(a , b)
{
for i ← 1 to n do
{
for j ← 1 to n do
{
 c[i , j] = a[i , j] + b[i, j]
}
}
}
```

## Time complexity Calculation(Count Method) Example

```
Algorithm add(a , b)
{
for i ← 1 to n do
count ← count + 1
{
for j ← 1 to n do
count ← count + 1
{
 c[i , j] = a[i , j] + b[i, j];
count ← count + 1
}
count ← count + 1
}
count ← count + 1
}
```

## Time complexity Calculation(Count Method) Example

- Outside of the for loops we have only 1 count variable and it takes only 1 unit of execution time.
- In the outer for loop we have 2 count variables and it takes n times to iterate for loop and therefore it takes 2n units of time to execute outer for loop.
- For the inner for loop we have 2 count variables and as it is a nested for loop it takes $n^2$ times to iterate inner for loop and therefore it takes $2n^2$ times to execute inner for loop.
- Total time taken for the above algorithm is $2n^2+ 2n+1$.

## Time complexity Calculation(Frequency Method)

- To calculate Step Count we need a table format.
- Table Consists of s/e i.e. Steps per execution- number of steps required for executing a statement
- Frequency i.e. number of times a statement is executed.
- **Total = s/e * frequency**

## Time complexity Calculation(Frequency Method) Example

```
Algorithm sum(x , n)
{
Total = 0;
for i ← 1 to n do
{
Total = Total + i;
}
}
```

| s/e | Frequency | Total = s/e * frequency |
|---|---|---|
| --- | --- | --- |
| --- | --- | --- |
| 1 | 1 | 1 |
| 1 | n+1 | n+1 |
| --- | --- | --- |
| 1 | n | n |
| --- | --- | --- |
| --- | --- | --- |
| **Total time complexity** | | **2n+2** |

## Time complexity Calculation(Frequency Method) Example

```
Algorithm reverse(n)
{
rev = 0;
while(n>0)
{
r = n %10;
rev = rev * 10 + r;
n = n/10;
}
}
```

| s/e | Frequency | Total = s/e * frequency |
|---|---|---|
| -- | -- | -- |
| -- | -- | -- |
| 1 | 1 | 1 |
| 1 | n+1 | n+1 |
| -- | -- | -- |
| 1 | n | n |
| 1 | n | n |
| 1 | n | n |
| -- | -- | -- |
| -- | -- | -- |
| **Total time complexity** | | **4n+2** |

## Time complexity calculation

$for(i = 0; i < n; i++)$ ------- iterated for n+1 times.

{

Statement; ------- iterated for n times.

}

Total time complexity is 2n+1 but it is written as O(n).

## Time complexity calculation

```
$for(i = n; i > 0; i--)$
{
Statement;                    ------- iterated for n times.
}
```

Time complexity is written as O(n).

## Time complexity calculation

$for(i = 1; i < n; i = i + 2)$

{

Statement;                                        ------- iterated for n/2 times.

}

Even if it is iterated for n/2 times, Time complexity is written as O(n).

## Time complexity calculation

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

| i | j | No. of times |
|---|---|---|
| 0 | 0 x | 0 |
| 1 | 0<br>1 x | 1 |
| 2 | 0<br>1<br>2 x | 2 |
| 3 | 0<br>1<br>2<br>3 x | 3 |
| n |  | n |

**It is in the format of 1+2+3+….+n = n(n+1)/2**

**Therefore Time complexity can be represented as O($n^2$)**

## Time complexity calculation

```
p = 0;
 $for(i = 1; p \leq n; i++)$
{
p = p + i;
}
```

| i | p |
|---|---|
| 1 | 0 +1 =1 |
| 2 | 1 + 2= 3 |
| 3 | 1 + 2 + 3 |
| 4 | 1 + 2 + 3 +4 |
| | |
| K | 1 + 2 + 3 + 4 + ….+ k |
| | |

Assume $p > n$
Since $p = k(k+1)/2$
$k(k+1)/2 > n$
$k^2 > n$
$k > \sqrt{n}$
Time Complexity is mentioned as $O(\sqrt{n})$

## Time complexity calculation

$for(i = 1; i < n; i = i * 2)$

{

Statements;

}

| i |
|---|
| 1 |
| 1 *2 = 2 |
| 2 *2 = 4 = $2^2$ |
| $2^2$ * 2 = $2^3$ |
| |
| $2^k$ |
| |

Assume $i \geq n$

Since $i = 2^k$

$2^k \geq n$

Now if we take $2^k = n$

$K = \log_2 n$

Time Complexity is mentioned as $O(\log_2 n)$

## Time complexity calculation

$for(i = 1; i < n; i = i * 2)$

{

Statements;

}

| I when n=8 |
|---|
| 1 |
| 1 *2 = 2 |
| 2 *2 = 4 |
| 4 * 2 =8 X |
| |
| |
| |

| I when n=10 |
|---|
| 1 |
| 1 *2 = 2 |
| 2 *2 = 4 |
| 4 * 2 =8 |
| 8 * 2 =16 X |
| |
| |

When n= 8 it runs for 3 times
Therefore Time complexity can be written as O($\log_2 8$)= 3$\log_2 2$ = 3

When n =10 it runs for 4 times
Therefore Time Complexity can be written as O($\log_2 10$)= 3.23= 4 Which is a ceil value of logn.

## Time complexity calculation

```
$for(i = n; i \geq 1; i = i/2)$
{
Statements;
}
```

| i |
|---|
| n/2 |
| $n/2^2$ |
| $n/2^3$ |
| $n/2^4$ |
| |
| $n/2^k$ |
| |

Assume i <1

Since $i = n/2^k$

$n/2^k<1$

$n/2^k=1$

Now if we take $n = 2^k$

$K = \log_2 n$

Time Complexity is mentioned as $O(\log_2 n)$

## Time complexity calculation

$for(i = 0; i * i < n; i++)$

{

Statements;

}

i * i < n

Condition fails when i * i >= n

$i^2 = n$

$i = \sqrt{n}$

Time Complexity is mentioned as $O(\sqrt{n})$

## Time complexity calculation

```
$for(i = 0; i < n; i++)$
{
Statements;   --------------- n times
}$for(i = 0; i < n; i++)$
{
Statements;   --------------- n times
}
```

Time Complexity is mentioned as O(n)

## Time complexity calculation

```
 p=0;
for(i=1; i < n; i = i * 2)
{
p++;   --------------- logn
}
for( j = 1; j < p; j = j * 2)
Statements;   ---------------  logp
}
```

Time Complexity  is mentioned as O(loglogn)

## Time complexity calculation

```
for(i=0; i < n; i++)----------- n times
{
for( j = 1; j < n; j = j * 2)-------nlogn
Statements;   ---------nlogn
}
```

Time Complexity  is mentioned as O(nlogn)

## Time complexity calculation

```
i=0; ------------- 1 time
while(i < n) ---------- n+1 times
{
statements; --------- n times
i++; ------------- n times
}
```

Total Time complexity is 3n + 2.

It can be written as O(n)

## Time complexity calculation

```
 a=1;
While(a < b)
{
Statements;
a = a * 2;
}
```

Loop will terminate when a >= b

Since a = $2^k$

$2^k >= b$

$2^k = b$

K = $\log_2 b$. Time Complexity is O(logn)

| a |
|---|
| 1 |
| 1 * 2 = 2 |
| $2 * 2 = 2^2$ |
| $2^2 * 2 = 2^3$ |
| |
| $2^k$ |
| |

## Time complexity calculation

```
 i=n;
While(i  > 1)
{
Statements;
i = i/2;
}
```

Time Complexity is $O(\log_2 n)$

## Time complexity calculation

```
i = 1;
k = 1;
 $while(k < n)$
{
stmt;
k = k + i;
i++;
}
```

| i | k |
|---|---|
| 1 | 1 |
| 2 | 1+1 =2 |
| 3 | 2 + 2 |
| 4 | 2+ 2 + 3 |
| 5 | 2+ 2 + 3 +4 |
| | |
| m | 2+ 2 + 3 +4+…+ m |

2+ 2 + 3 +4+…+ m can be written as m(m+1)/2
Assume Condition fails at when $k \geq n$ then

$$m(m+1)/2 \geq n$$
$$m^2 \geq n$$
$$m = \sqrt{n}$$

Time Complexity can be written as O($\sqrt{n}$)

## Time complexity calculation

| m = 6 | n = 3 |
|---|---|
| 3 | 3 |

```
while(m!=n)
{
if(m > n)
m = m - n;
else
n = n – m;
}
```

It is executed only for one time

## Time complexity calculation

|  |  |
|---|---|
| m = 5 | n = 5 |

```
while(m!=n)
{
if(m > n)
m = m - n;
else
n = n – m;
}
```

It Won’t go inside the loop

## Time complexity calculation

```
while(m!=n)
{
if(m > n)
m = m - n;
else
n = n – m;
}
```

| m = 16 | n = 2 |
|---|---|
| 14 | 2 |
| 12 | 2 |
| 10 | 2 |
| 8 | 2 |
| 6 | 2 |
| 4 | 2 |
| 2 | 2 |

16/2 = n/2 times

Total time complexity is O(n)

- **Space Complexity** – Total amount of memory required by an algorithm.
- Space complexity of an algorithm calculated using by two parts.

  - Fixed part – variables which are having independent characteristics
  - Variable part- Variables which are having dependent characteristics

Algorithm abc (x, y, z)

return x * y * z + (x-y);

$s(p) = c + s_p$

Where c is fixed part and s(p) is variable part.

Since x, y, z are independent variables and they are not depends on other, then each variable takes 1 unit of time.

Total space complexity of this algorithm is 3

```
Algorithm sum(x,n)
{
Total = 0;
for i<-1 to n do
{
Total = Total + x[i];
}
}
```

$s(p) = c + s_p$

Where c is fixed part and s(p) is variable part.

Since x, n, Total  are independent variables which requires 1 unit of memory and the variable part is x[i] which depends on n value which requires n units of memory.

Total Space Complexity is 3+n

## Best, Worst and Average case analysis

Let us assume we have considered below elements in the list

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **8** | **6** | **12** | **5** | **9** | **7** | **4** | **3** | **16** | **18** |

Best case- Searching key element found at first index
Best case time – O(1)
B(n)= O(1)
Worst Case – searching the key at last index
Worst Case time = n
W(n)= O(n)
Average case = all possible case time/number of cases

Average Case = 1+2+3+…+n/n = n(n+1)/2/n = n+1/2