Unit 1

Introduction

Topics to be covered in lecture

- Analysis of algorithm
- frequency count and its importance in analysis of an algorithm
- Time complexity & Space complexity of an algorithm
- Big 'O', ‘Ω' and 'Θ' notations
- Best, Worst and Average case analysis of an algorithm.

- Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis −

- **Worst-case** − The maximum number of steps taken on any instance of size **a**.

- **Best-case** − The minimum number of steps taken on any instance of size **a**.

- **Average case** − An average number of steps taken on any instance of size **a**.

How to create programs

- Requirements
- Analysis: bottom-up vs. top-down
- Design: data objects and operations
- Refinement and Coding
- Verification
  - Program Proving
  - Testing
  - Debugging

Algorithm

- Definition
  An *algorithm* is a finite set of instructions that accomplishes a particular task.
- Criteria
  - input
  - output
  - definiteness: clear and unambiguous
  - finiteness: terminate after a finite number of steps
  - effectiveness: instruction is basic enough to be carried out

Data Type

- Data Type
  A *data type* is a collection of *objects* and a set of *operations* that act on those objects.
- Abstract Data Type
  An *abstract data type(ADT)* is a data type that is organized in such a way that the specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operations.

Specification vs. Implementation

- Operation specification
  - function name
  - the types of arguments
  - the type of the results
- Implementation independent

Measurements

- Criteria
  - Is it correct?
  - Is it readable?
- Performance Analysis (machine independent)
  - space complexity: storage requirement
  - time complexity: computing time
- Performance Measurement (machine dependent)

Analysis of Algorithm

- Time Complexity
- Space Complexity

- Suppose X=X+1
- Determine the amount of time required by the above Statement in terms of clock time is not possible because following is always dynamic.
  1. The Machine that is used to execute the programming statement
  2. Machine Language instruction set
  3. Time required by each machine instruction
  4. The Translation of compiler will make for this statement to machine language.
  5. The kind of operating system(multiprogramming or time sharing)
- The above information varies from machine to machine. Hence it is not possible to find out the exact figure. Hence the performance of the machine is measured in terms of frequency count.

Space Complexity
$S(P)=C+S_P(I)$

- Fixed Space Requirements (C)
  Independent of the characteristics of the inputs and outputs
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- Variable Space Requirements ($S_P(I)$)
  depend on the instance characteristic I
  - number, size, values of inputs and outputs associated with I
  - recursive stack space, formal parameters, local variables, return address

***Program:** Simple arithmetic function

```
float abc(float a, float b, float c)
{
return a + b + b * c + (a + b - c) / (a + b) + 4.00;
}
```

$$\boxed{S_{abc}(I) = 1}$$

***Program :** Iterative function for summing a list of numbers

```
float sum(float list[ ], int n)
{
float tempsum = 0;
int i;
for (i = 0; i<n; i++)
tempsum += list [i];
return tempsum;
}
```

Recall: pass the address of the first element of the array & pass by value

$$\boxed{S_{sum}(I) = n}$$

***Program :** Recursive function for summing a list of numbers

```
float rsum(float list[ ], int n)
{
if (n) return rsum(list, n-1) + list[n-1];
return 0;
}
```

$S_{sum}(I)=S_{sum}(n)=6n$

Assumptions:

***Figure :** Space needed for one recursive call of Program

Type	Name	Number of bytes
parameter: float parameter: integer return address:(used internally)	list [ ] n	2 2 2(unless a far address)
TOTAL per recursive call		6

Time Complexity

$$T(P)=C+T_P(I)$$

- Compile time (C)
  independent of instance characteristics
- run (execution) time T_P
- Definition
  A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.
- Example
  - abc = a + b + b * c + (a + b - c) / (a + b) + 4.0
  - abc = a + b + c

$$T_P(n)=c_aADD(n)+c_sSUB(n)+c_lLDA(n)+c_{st}STA(n)$$

Regard as the same unit
machine independent

Methods to compute the step count

- Introduce variable count into programs
- Tabular method
  - Determine the total number of steps contributed by each statement
    step per execution $\times$ frequency
  - add up the contribution of all statements

Iterative summing of a list of numbers

***Program :** Program with count statements

```
float sum(float list[ ], int n)
{
float tempsum = 0; count++; /* for assignment */
int i;
for (i = 0; i < n; i++) {
count++;            /*for the for loop */
tempsum += list[i]; count++;  /* for assignment */
}
count++;        /* last execution of for */
return tempsum;
count++;        /* for return */
}
```

$\boxed{2n + 3 \text{ steps}}$

***Program 1.13:** Simplified version of Program 1.12 (p.23)

```
float sum(float list[ ], int n)
{
    float tempsum = 0;
    int i;
    for (i = 0; i < n; i++)
        count += 2;
    count += 3;
    return 0;
}
```

2n + 3 steps

Recursive summing of a list of numbers

***Program :** Program with count statements added

```
float rsum(float list[ ], int n)
{
count++;      /*for if conditional */
  if (n) {
    count++;  /* for return and rsum invocation */
    return rsum(list, n-1) + list[n-1];
  }
  count++;
  return list[0];
}
```

$\boxed{2n+2}$

Matrix addition

***Program :** Matrix addition

```
void add( int a[ ] [MAX_SIZE], int b[ ] [MAX_SIZE],
          int c [ ] [MAX_SIZE], int rows, int cols)
{
    int i, j;
    for (i = 0; i < rows; i++)
      for (j= 0; j < cols; j++)
        c[i][j] = a[i][j] +b[i][j];
}
```

***Program :** Matrix addition with count statements

```
void add(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE],
         int c[ ][MAX_SIZE], int row, int cols )
{
  int i, j;
  for (i = 0; i < rows; i++){
    count++; /* for i for loop */
    for (j = 0; j < cols; j++) {
      count++; /* for j for loop */
      c[i][j] = a[i][j] + b[i][j];
      count++; /* for assignment statement */
    }
    count++;    /* last time of j for loop */
  }
  count++;       /* last time of i for loop */
}
```

2rows * cols + 2 rows + 1

Tabular Method

***Figure** Step count table for Program

Iterative function to sum a list of numbers

steps/execution

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

Recursive Function to sum of a list of numbers

***Figure :** Step count table for recursive summing function

Statement	s/e	Frequency	Total steps
float rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total			2n+2

Matrix Addition

***Figure :** Step count table for matrix addition

Statement	s/e	Frequency	Total steps
Void add (int a[ ][MAX_SIZE]. . .)	0	0	0
{	0	0	0
int i, j;	0	0	0
for (i = 0; i < row; i++)	1	rows+1	rows+1
for (j=0; j< cols; j++)	1	rows. (cols+1)	rows. cols+rows
c[i][j] = a[i][j] + b[i][j];	1	rows. cols	rows. cols
}	0	0	0
Total			2rows. cols+2rows+1

Exercise 1

***Program :** Printing out a matrix

```
void print_matrix(int matrix[ ][MAX_SIZE], int rows, int cols)
{
int i, j;
for (i = 0; i < row; i++) {
for (j = 0; j < cols; j++)
printf("%d",  matrix[i][j]);
printf( "\n");
}
}
```

Exercise 2

***Program :Matrix multiplication function**

```
void mult(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE], int c[ ][MAX_SIZE])
{
int i, j, k;
for (i = 0; i < MAX_SIZE; i++)
for (j = 0; j< MAX_SIZE;  j++) {
c[i][j] = 0;
for (k = 0; k < MAX_SIZE; k++)
c[i][j]  +=  a[i][k] * b[k][j];
}
}
```

Asymptotic Notations

- Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.
- Time function of an algorithm is represented by **T(n)**, where **n** is the input size.
- Different types of asymptotic notations are used to represent the complexity of an algorithm. Following asymptotic notations are used to calculate the running time complexity of an algorithm.
- **O** − Big Oh
- **Ω** − Big omega
- **θ** − Big theta
- **o** − Little Oh
- **ω** − Little omega

Asymptotic analysis :

Asymptotic Notations: To enable us to make meaningful (but inexact) statements about the time and space complexities of an algorithm , asymptotic notations (O, o, Ω, ω, θ) are used.

Big "Oh" : The function f(n) = O(g(n)), to be read as " f of n is Big Oh of g of n, if and only if there exist positive constants c and n_0 such that, $f(n) <= c*g(n)$ for all $n>=n_0$,C>0. for example,

i. $f(n) = 3n + 2 <= 5n$ for all $n >= 1$. Here c = 5, g(n) = n and $n_0 = 1$.

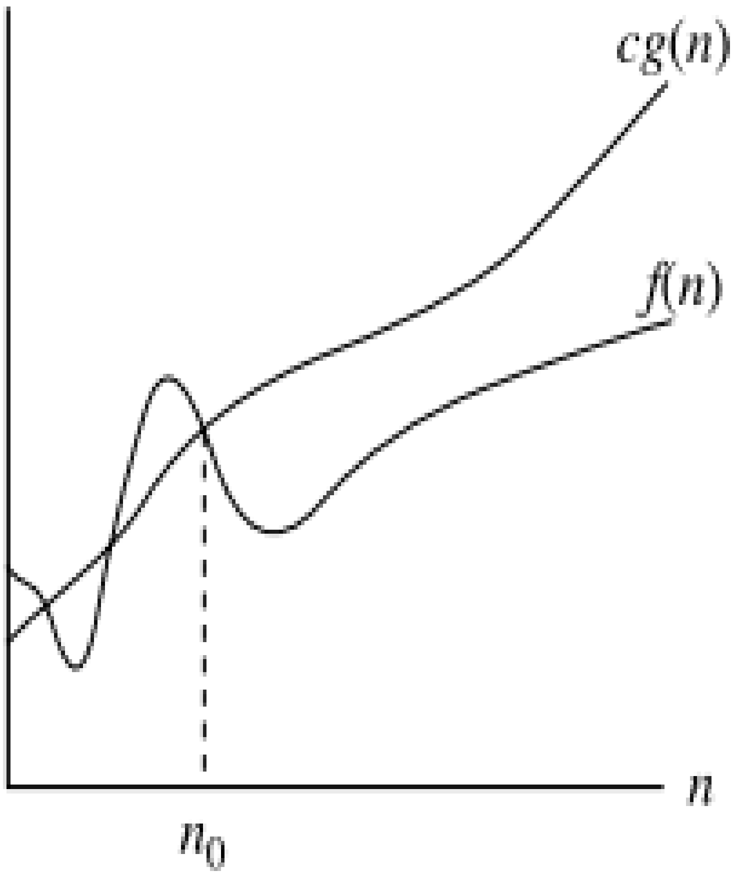
ii. $f(n) = (n^3 + 6n^2 + 11n +3) <= n^3$ for all $n >= 1$.

Here c = 1, $g(n) = n^3$ and $n_0 = 1$.

Thus f(n) = O(g(n)) states that g(n) is an ***upper bound*** on the value of f(n) for all $n>=n_0$.

O-notation

$$O(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Asymptotic analysis :

Omega (Ω): The function f(n) = Ω(g(n)), to be read as " f of n

is omega of g of n, if and only if there exist positive constants c and n0 such that, f(n) >= c*g(n) for all n>=n0,

for example,

i. f(n) = 3n + 2 >= 3n for all n >= 1.

Here c = 3, g(n) = n and n0 = 1.

ii. f(n) = (n3 + 6n2 + 11n +3) >= 21n3 for all n >= 1.

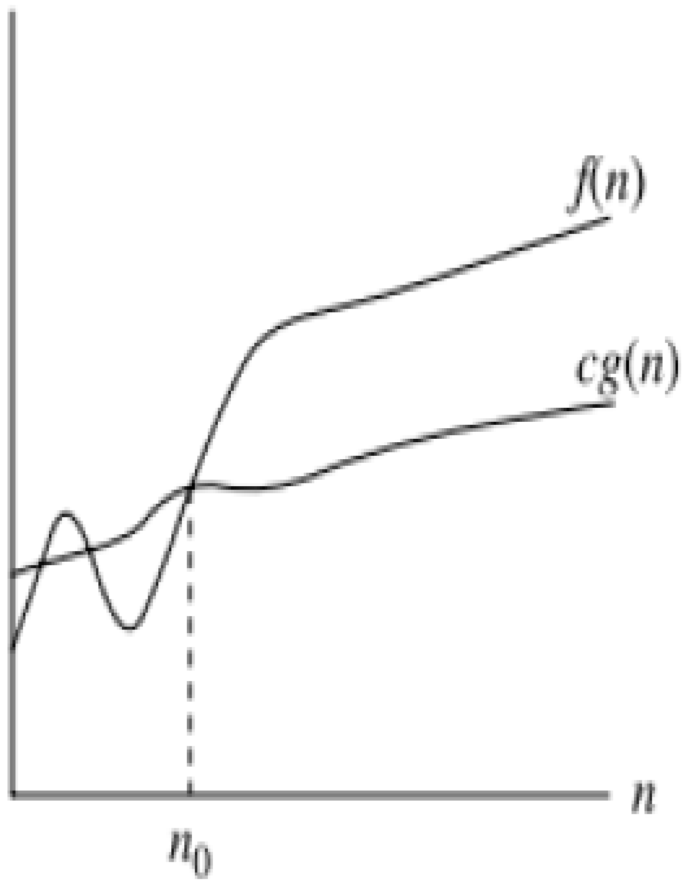
Here c = 21, g(n) = n3 and n0 = 1.

Thus f(n) = Ω(g(n)) states that g(n) is an ***lower bound*** on

the value of f(n) for all n>=n0.

Ω-notation

$$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \} .$$



Asymptotic analysis :

Theta (θ): The function $f(n) = \theta(g(n))$, to be read as " f of n is theta of g of n, if and only if there exist positive constants c_1 , c_2 and n_0 such that, $c_1 g(n) <= f(n) <= c_2 * g(n)$ for all $n >= n_0$, for example,

i. $f(n) = 3n + 2$ then

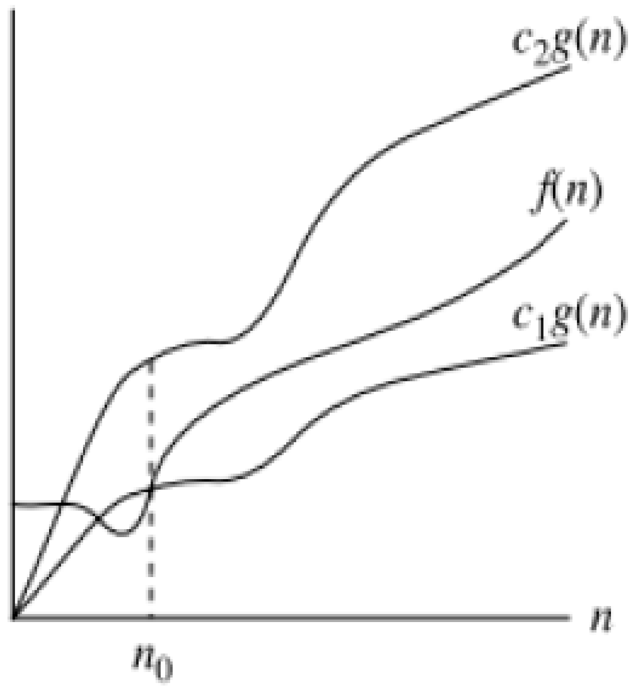
$3n <= 3n + 2 <= 5n$ for all $n>=1$

Here $c_1 = 3$, $c_2 = 5$, $g(n) = n$ and $n_0 = 1$.

Thus $f(n) = \theta(g(n))$ states that g(n) is both ***upper & lower bound*** on the value of f(n) for all $n>=n_0$.

Θ-notation

$$\Theta(g(n)) = \{ f(n) : \text{ there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \} .$$



$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Time complexity

```
#include <stdio.h>
int main()
{
   printf("Hello World");
}
```

```
#include <stdio.h>
void main()
{
   int i, n;
   for (i = 1; i <= n; i++) {          3C  n+1          3(n+1)+n = 3n+3+n
      printf("Hello Word !!!\n");  1C  n              4n+3 =O(n)
   }
```

```
Sum(a,b){
return a+b ;  or                    c=a+b;    cost 2 no of times 1
                                    return c   cost 1 no of time 1
}                                   O(1)
```

```
list_Sum(A,n){
total =0                    cost 1        no of times   1
for i=0 to n-1              cost 2        no of times n+1
sum = sum + A[i]            cost 2        no of times n
return sum                  cost 1        no of times 1
}
```

$1+2n+2+2n+1 = 4n+4 = O(n)$

for i=0 to n-1	cost = 2	no of times	n+1
for j=0 to n-1	2		n(n+1)
sum= sum+a[i]	2		nn
return sum	1		1

2(n+1)+2(nn+n)+2(nn)+1

2n+2+2n.n+2n+2n.n+1

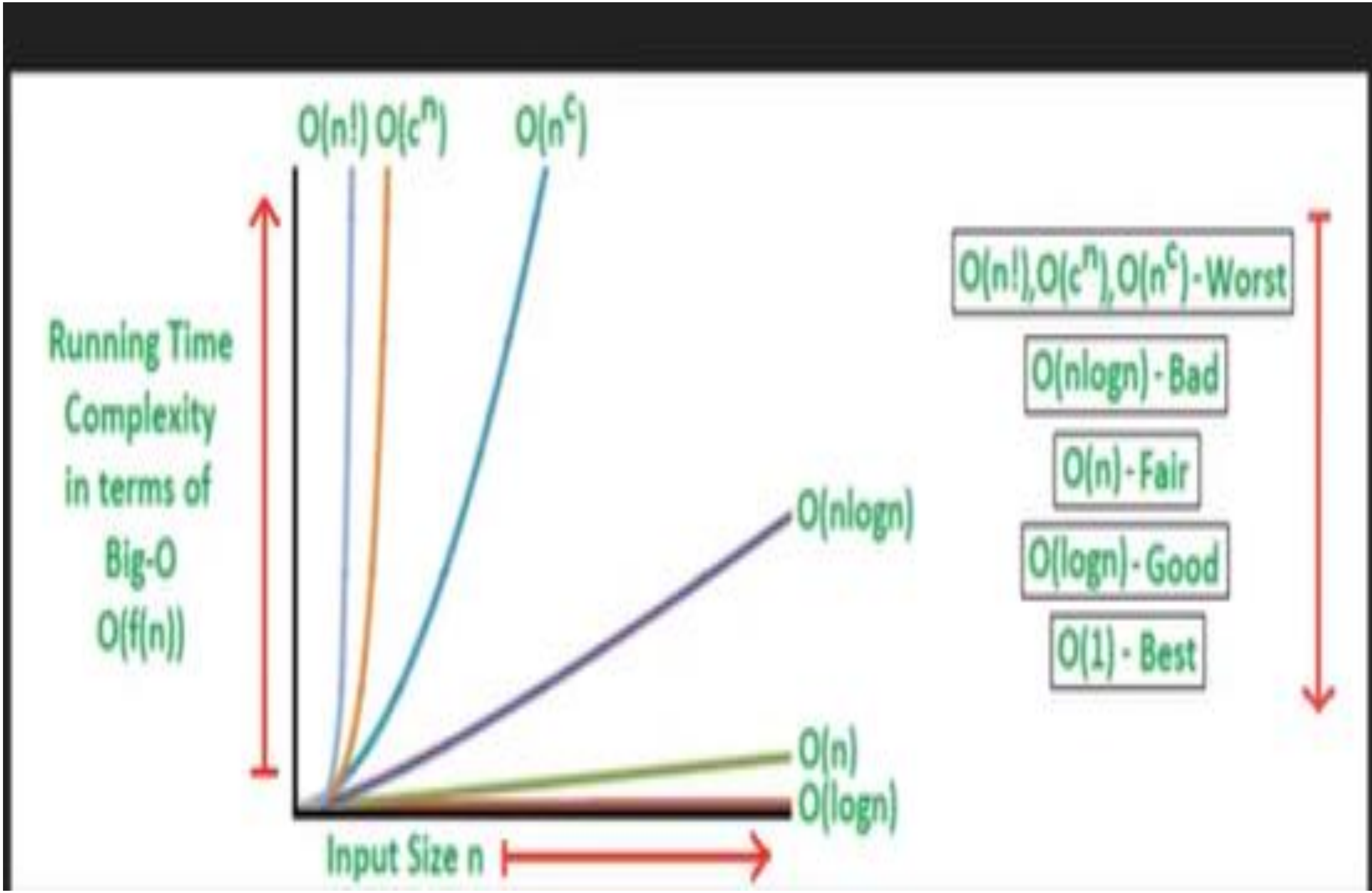
$4n^2+4n+3$

$O(n^2)$

-drop all lower order terms

-drop all constant terms

Big O Notation	Name	Example(s)
O(1)	Constant	# Odd or Even number, # Look-up table (on average)
O(log n)	Logarithmic	# Finding element on sorted array with **binary search**
O(n)	Linear	# Find max element in unsorted array,
O(n log n)	Linearithmic	# Sorting elements in array with **merge sort**
$O(n^2)$	Quadratic	# Duplicate elements in array **(naïve)**, # Sorting array with **bubble sort**
$O(n^3)$	Cubic	# 3 variables equation solver
$O(2^n)$	Exponential	# Find all subsets
O(n!)	Factorial	# Find all permutations of a given set/string



O(n!) O(c^n) O(n^c)
Running Time Complexity in terms of Big-O O(f(n))
O(nlogn)
O(n)
O(logn)
Input Size n
O(n!),O(c^n),O(n^c) - Worst
O(nlogn) - Bad
O(n) - Fair
O(logn) - Good
O(1) - Best

Frequency Count Method

	s/c	Frequency	Total
Function ArraySum(A , n)	0	0	0
Sum=0;	1	1	1
for(i=0; i<n; i++)	2	n+1	2n+2
{			
sum=sum+A[i];	2	n	2n
}			
return sum;	1	1	1
End Function			
		O(n)	4n+4

Frequency Count Method	
	Step Count
int sum(int n)	
{	
int sum=0;	1
for(i=0; i<n; i++)	1+(n+1)+n
sum+=i*i*i	4n
return sum;	1
}	
O(n)	6n+4

```
float sum(float list[ ], int n)
{
    float tempsum = 0; count++ ; /* for assignment */
    int i;
    for (i = 0; i < n; i++) {
        count++ ;          /*for the for loop */
        tempsum += list[i]; count++ ;  /* for assignment */
    }
    count++ ;       /* last execution of for */
    return tempsum;
    count++ ;       /* for return */
```

1
1
1+(n+1)+n **2n+2**
n
2n
n
1 **1** **1**
6n+7 =>O(n)

```
Algorithm add(a, b, n)
for(i=0; i<n; i++)                          n+1
      for(j=0; j<n; j++)                    n(n+1)
            c[i][j]=a[i][j]+b[i][j];        n. n
end
```

$O(n^2)$

Variables

a - n.n

b - n.n

c - n.n

```
Algorithm add(a, b, n)

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

          for(j=0; j<n; j++)

                    c[i][j]=0;

            for(k=0; k<n; k++)

               c[i][j]=c[i][j]+a[i][k]+b[k][j];
end
```

$O(n^3)$

Variables

a - n.n

b - n.n

c - n.n

n , i, j k,= $3n^2+4$ $O(n^2)$