Course Objective

1. To understand the concepts of data structures and abstract data type (ADT).
2. To acquire knowledge of linear and non-linear with their applications.
3. To choose the appropriate data structure and algorithm design method for a specified application.
4. To solve problems using linear and non-linear data structures and writing programs for these solutions.

Course Outcome

Graduate shall be able to:

1. describe the usage of various data structure
2. analyze, evaluate and choose appropriate abstract data types and algorithms to solve particular problems.
3. Compare and contrast the benefits of dynamic and static data structures implementation.
4. design and implement the learned data structure algorithm for problem solving.

Session Objectives

- To understand the concepts of
  - Data structures
  - Types of Data Structures
  - Applications
  - Algorithms
  - ADTs

Session Topics

- Algorithms
- ADTs
- Properties of an Algorithm
- Data structures
- Types of Data structures
- Problem Solving Phase

The Need for Data Structures

Data structures organize data

$\Rightarrow$ more efficient programs.

More powerful computers $\Rightarrow$ more complex applications.

More complex applications demand more calculations.

Complex computing tasks are unlike our everyday experience.

- More typically, a *data structure* is meant to be an *organization for a collection of data items*.
- Any organization for a collection of records can be searched, processed in any order, or modified.
- The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days. A data structure requires a certain amount of:
- space for each data item it stores
- time to perform a single basic operation
- programming effort.

Selecting a Data Structure

Select a data structure as follows:

1. Analyze the problem to determine the resource constraints a solution must meet.
2. Determine the basic operations that must be supported. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

Data Structures

DS includes

- Logical or mathematical description of the structure and Implementation of the structure on a computer
- Quantitative analysis of the structure, which includes determining the amount of memory needed to store the structure and the time required to process the structure.

Classification of Data Structures

"***Data Structures*** "deals with the study of how the data is organized in the memory, how efficiently the data can be retrieved and manipulated, and the possible ways in which different data items are logically related.

Types:

Primitive Data Structure: Ex. int,float,char

Non-primitive Data Structures:

Ex.Arrays,Structures,stacks

Linear Data Structures: Ex.Stacks,queues,linked list

Non-Linear Data Structures: Ex.Trees,Graphs.

persistent and ephemeral data structure:

Classification of Data Structures

1. Primary Data structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers.
2. All the basic constants (integers, floating point numbers, character constants, string constants)and pointers are considered as primary data structures
3. Secondary Data Structures are more complicated data structures derived from primary data structures
4. They emphasize on grouping same or different data items with relationship between each data item
5. Secondary data structures can be broadly classified as static data structures and dynamic data structures

Classification of Data Structures

- If a data structure is created using static memory allocation (ie. a data structure formed when the number of data items are known in advance), it is known as static data structure or fixed size data structure
- If a data structure is created , using dynamic memory allocation(ie. a data structure formed when the number of data items are not known in advance) it is known as dynamic data structure or variable size data structure
- Dynamic data structures can be broadly classified as linear data structures and non linear data structures
- Linear data structures have a linear relationship between its adjacent elements. Linked lists are examples of linear data structures.

Classification of Data Structures

- A linked list is a linear dynamic data structure that can grow and shrink during its execution time
- A circular linked list is similar to a linked list except that the first and last nodes are interconnected
- Non linear data structures don't have a linear relationship between its adjacent elements
- In a linear data structure , each node has a link which points to another node, whereas in a non linear data structure, each node may point to several other nodes
- A tree is a nonlinear dynamic data structure that may point to one or more nodes at a time
- A graph is similar to tree except that it has no hierarchical relationship between its adjacent elements

Abstract data type (ADTs)

- A data type that is defined entirely by a set of operations is referred to as Abstract data type or simply ADT
- Abstract data types are a way of separating the specification and representation of data types
- An ADT is a black box, where users can only see the syntax and semantics of its operations
- An ADT is a combination of interface and implementation The interface defines the logical properties of the ADT and especially the signatures of its operations
- The implementation defines the representation of data structure and the algorithms that implement the operations
- An abstract data type encapsulates data and functions into a named data type

Abstract data type (ADTs)

- It is similar to a structure in C, but can include functions in it
- The basic difference between ADTs and primitive data types is that the latter allow us to look at the representation, whereas former hide the representation from us
- An ADT consists of a collection of values and operations with the values derive their meaning solely through the operations that can be performed upon them
- Benefits of using ADTs:

  Code is easier to understand

  Implementations of ADTs can be changed without requiring changes to the program that uses the ADTs

Data Structures: *Data Collections*

- Linear structures
  - Array: Fixed-size
  - Linked-list: Variable-size
  - Stack: Add to top and remove from top
  - Queue: Add to back and remove from front
  - Priority queue: Add anywhere, remove the highest priority
- Tree: A branching structure with no loops
- Hash tables: Unordered lists which use a ‘hash function’ to insert and search
- Graph: A more general branching structure, with less stringent connection conditions than for a tree

ADTs Collection

- *ADT is a data structure and a set of operations which can be performed on it.*
  - *A class in object-oriented design is an ADT*
- *The pre-conditions* define a state of the program which the client guarantees will be true before calling any method,
- *post-conditions* define the state of the program that the object's method will guarantee to create for you when it returns.
- create Create a new collection
- add Add an item to a collection
- delete Delete an item from a collection find Find an item matching some criterion in the collection
- destroy Destroy the collection

Data Structures and ADTs

- A container in which data is being stored
  - Example: structure, file, or array
- An ADT is a data structure which does not exist within the host language, but rather must be created out of existing tools
- It is both a collection of data and a set of rules that govern how the data will be manipulated
- Examples: list, stack, queue, tree, table, and graph
- An ADT sits on top of the data structure, and the rules that govern the use of the data define the interface between the data structure and the ADT

Good Computer Program

- A computer program is a series of instructions to carry out a particular task written in a language that a computer can understand.
- The process of preparing and feeding the instructions into the computer for execution is referred as programming.
- There are a number of features for a good program

  Run efficiently and correctly

  Have a user friendly interface

  Be easy to read and understand

  Be easy to debug

  Be easy to modify

  Be easy to maintain

Good Computer Program

- Programs consists of two things: Algorithms and data structures
- A Good Program is a combination of both algorithm and a data structure
- An algorithm is a step by step recipe for solving an instance of a problem
- A data structure represents the logical relationship that exists between individual elements of data to carry out certain tasks
- A data structure defines a way of organizing all data items that consider not only the elements stored but also stores the relationship between the elements

Algorithms

- An algorithm is a step by step recipe for solving an instance of a problem.
- Every single procedure that a computer performs is an algorithm.
- An algorithm is a precise procedure for solving a problem in finite number of steps.
- An algorithm states the actions to be executed and the order in which these actions are to be executed.
- An algorithm is a well ordered collection of clear and simple instructions of definite and effectively computable operations that when executed produces a result and stops executing at some point in a finite amount of time rather than just going on and on infinitely.

Algorithm Properties

An algorithm possesses the following properties:

- It must be correct.
- It must be composed of a series of concrete steps.
- There can be no ambiguity as to which step will be performed next.
- It must be composed of a finite number of steps.
- It must terminate.
- It takes zero or more inputs
- It should be efficient and flexible
- It should use less memory space as much as possible
- It results in one or more outputs

Various steps in developing Algorithms

- Devising the Algorithm:

  It's a method for solving a problem. Each step of an algorithm must be precisely defined and no vague statements should be used. Pseudo code is used to describe the algorithm , in less formal language than a programming language.

- Validating the Algorithm:

  The proof of correctness of the algorithm. A human must be able to perform each step using paper and pencil by giving the required input , use the algorithm and get the required output in a finite amount of time.

Various steps in developing Algorithms

- Expressing the algorithm:

  To implement the algorithm in a programming language.

  The algorithm used should terminate after a finite number of steps.

Efficiency of an algorithm

- Writing efficient programs is what every programmer hopes to be able to do. But what kinds of programs are efficient? The question leads to the concept of generalization of programs.
- Algorithms are programs in a general form. An algorithm is an idea upon which a program is built. An algorithm should meet three things:

  It should be independent of the programming language in which the idea is realized

  Every programmer having enough knowledge and experience should understand it

  It should be applicable to inputs of all sizes

Efficiency of an algorithm

- Efficiency of an algorithm denotes the rate at which an algorithm solves a problem of size n.
- It is measured by the amount of resources it uses, the time and the space.
- The time refers to the number of steps the algorithm executes while the space refers to the number of unit memory storage it requires.
- An algorithm's complexity is measured by calculating the time taken and space required for performing the algorithm.
- The input size, denoted by n, is one parameter , used to characterize the instance of the problem.
- The input size n is the number of registers needed to hold the input (data segment size).

Time Complexity of an Algorithm

- Time Complexity of an algorithm is the amount of time(or the number of steps) needed by a program to complete its task (to execute a particular algorithm)
- The way in which the number of steps required by an algorithm varies with the size of the problem it is solving. The time taken for an algorithm is comprised of two times

  Compilation Time

  Run Time
- Compilation time is the time taken to compile an algorithm. While compiling it checks for the syntax and semantic errors in the program and links it with the standard libraries , your program has asked to.

Time Complexity of an Algorithm

- Run Time: It is the time to execute the compiled program. The run time of an algorithm depend upon the number of instructions present in the algorithm. Usually we consider, one unit for executing one instruction.
- The run time is in the control of the programmer , as the compiler is going to compile only the same number of statements , irrespective of the types of the compiler used.
- Note that run time is calculated only for executable statements and not for declaration statements
- Time complexity is normally expressed as an order of magnitude, eg $O(n^2)$ means that if the size of the problem n doubles then the algorithm will take four times as many steps to complete.

Time Complexity of an Algorithm

- Time complexity of a given algorithm can be defined for computation of function f() as a total number of statements that are executed for computing the value of f(n).
- Time complexity is a function dependent from the value of n. In practice it is often more convenient to consider it as a function from |n|
- Time complexity of an algorithm is generally classified as three types.

  (i) Worst case

  (ii) Average Case

  (iii) Best Case

Time Complexity

- Worst Case: It is the longest time that an algorithm will use over all instances of size n for a given problem to produce a desired result.

- Average Case: It is the average time(or average space) that the algorithm will use over all instances of size n for a given problem to produce a desired result. It depends on the probability distribution of instances of the problem.

- Best Case: It is the shortest time (or least space) that the algorithm will use over all instances of size n for a given problem to produce a desired result.

Space Complexity

- Space Complexity of a program is the amount of memory consumed by the algorithm (apart from input and output, if required by specification) until it completes its execution.
- The way in which the amount of storage space required by an algorithm varies with the size of the problem to be solved.
- The space occupied by the program is generally by the following:

  A fixed amount of memory occupied by the space for the program code and space occupied by the variables used in the program.

  A variable amount of memory occupied by the component variable whose size is dependent on the problem being solved. This space increases or decreases depending upon whether the program uses iterative or recursive procedures.

Space Complexity

- The memory taken by the instructions is not in the control of the programmer as its totally dependent upon the compiler to assign this memory.
- But the memory space taken by the variables is in the control of a programmer. More the number of variables used, more will be the space taken by them in the memory.
- Space complexity is normally expressed as an order of magnitude, eg $O(n^2)$means that if the size of the problem n doubles then four times as much working storage will be needed.
- There are three different spaces considered for determining the amount of memory used by the algorithm.

Space Complexity

- Instruction Space is the space in memory occupied by the compiled version of the program. We consider this space as a constant space for any value of n. We normally ignore this value , but remember that is there. The instruction space is independent of the size of the problem
- Data Space is the space in memory , which used to hold the variables , data structures, allocated memory and other data elements. The data space is related to the size of the problem.
- Environment Space is the space in memory used on the run time stack for each function call. This is related to the run time stack and holds the returning address of the previous function. The memory each function utilises on the stack is a constant as each item on the stack has a return value and pointer on it.

Iterative Factorial Example

```
fact ( long n)
{
  for (i=1; i<=n; i++)
x=i*x;
return x;
}
```

- Space occupied is
- Data Space: i, n and x
- Environment Space: Almost nothing because the function is called only once.
- The algorithm has a complexity of O(1) because it does not depend on n. No matter how big the problem becomes ,the space complexity remains the same since the same variables are used , and the function is called only once.

Recursive Factorial Example

```
long fact (long x)
{
if (x<=1)
return(1);
else
return (x * fact(x-1));
}
```

- Space occupied is
- Data space : x
- Environment Space: fact() is called recursively , and so the amount of space this program used is based on the size of the problem

Recursive Factorial Example

- The space complexity is

O(n)= (x + function call) * n

= x + function call + memory needed for fact(x-1)

x+function call + x+ function call + ……+

x+ function call n(n-1)+…….+1

- Note that in measuring space complexity, memory space is always allocated for variables whether they are used in the program or not.
- Space Complexity is not as big of an issue as time complexity because space can be reused, whereas time cannot.

Problem Solving Phase

- A problem/Project needs programs to create ,append, update the database, print data, permit online enquiry and so on.
- A Programmer should identify all requirements to solve the problem. Each problem should have the following specifications

  Type of Programming language

  Narration of the program describing the tasks to be performed

  Frequency of Processing (hourly, daily, weekly etc)

  Output and input of the program

  Limitations and restrictions for the program

  Detailed Specifications

Method 1: Algorithm Analysis

Code each algorithm and run them to see how long they take.

Problem: How will you know if there is a better program or whether there is no better program?

What will happen when the number of inputs is twice as many? Three? A hundred?

Method 2: Algorithm Analysis

Develop a model of the way computers work and compare how the algorithms behave in the model.

Goal: To be able to predict performance at a coarse level. That is, to be able to distinguish between good and bad algorithms.

Another benefit: when assumptions change, we can predict the effects of those changes.

Why algorithm analysis?

As computers get faster and problem sizes get bigger, analysis will become *more* important.

Why? The difference between good and bad algorithms will get bigger.

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;
}
```

$S_{abc}(I) = 1$

***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;
}
```

$S_{sum}(I) = n$

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

***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 */
}
```

2n + 3 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];
                    }
```

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]. . .)
{
int i, j;
for (i = 0; i < row; i++)
for (j=0; j< cols; j++)
c[i][j] = a[i][j] + b[i][j];
}
	0
	0
	0
	1
	1
	1
	0
		0
		0
		0
		rows+1
		rows. (cols+1)
		rows. cols
		0
			0
			0
			0
			rows+1
			rows. cols+rows
			rows. cols
			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 $n_0 = 1$.

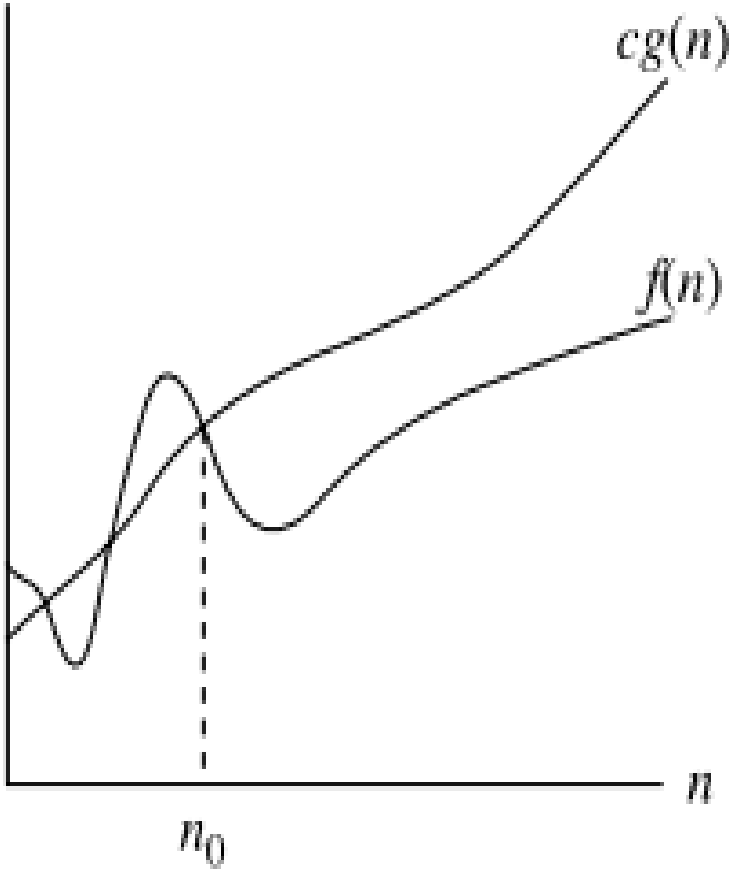
ii. f(n) = ($n^3 + 6n^2 + 11n + 3$) <= n^3 for all n >= 1.

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

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



$g(n)$ 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.

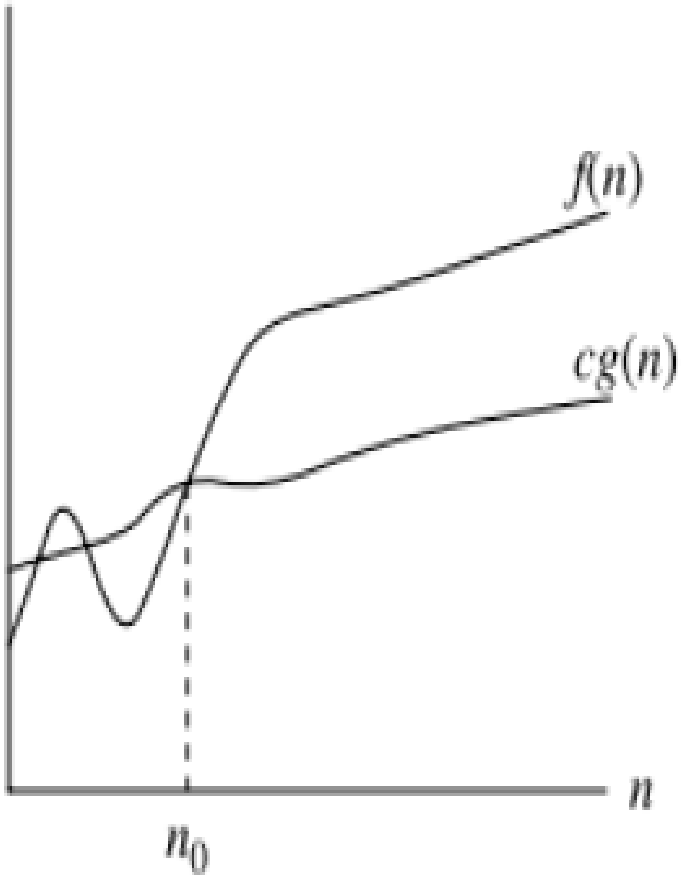
ii. f(n) = (n3 + 6n2 + 11n +3) >= 21n3 for all n >= 1.

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) = θ(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

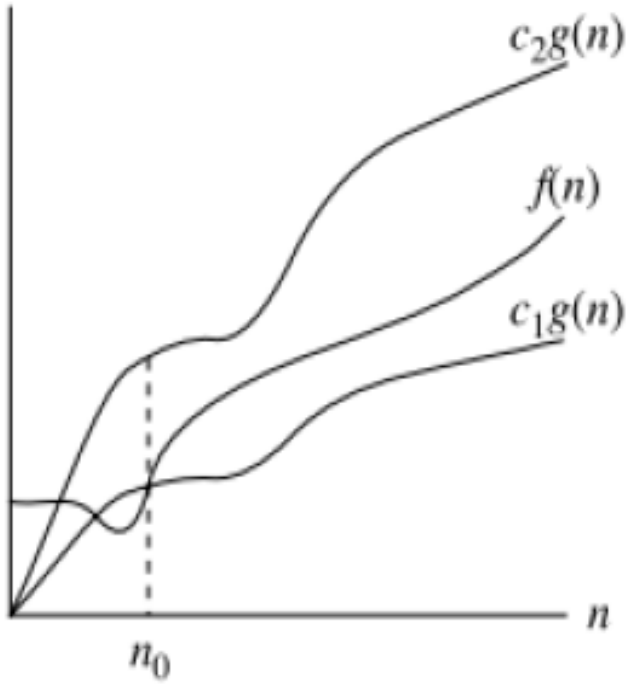
3n <= 3n + 2 <= 5n for all n>=1

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

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

Θ-notation

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

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

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

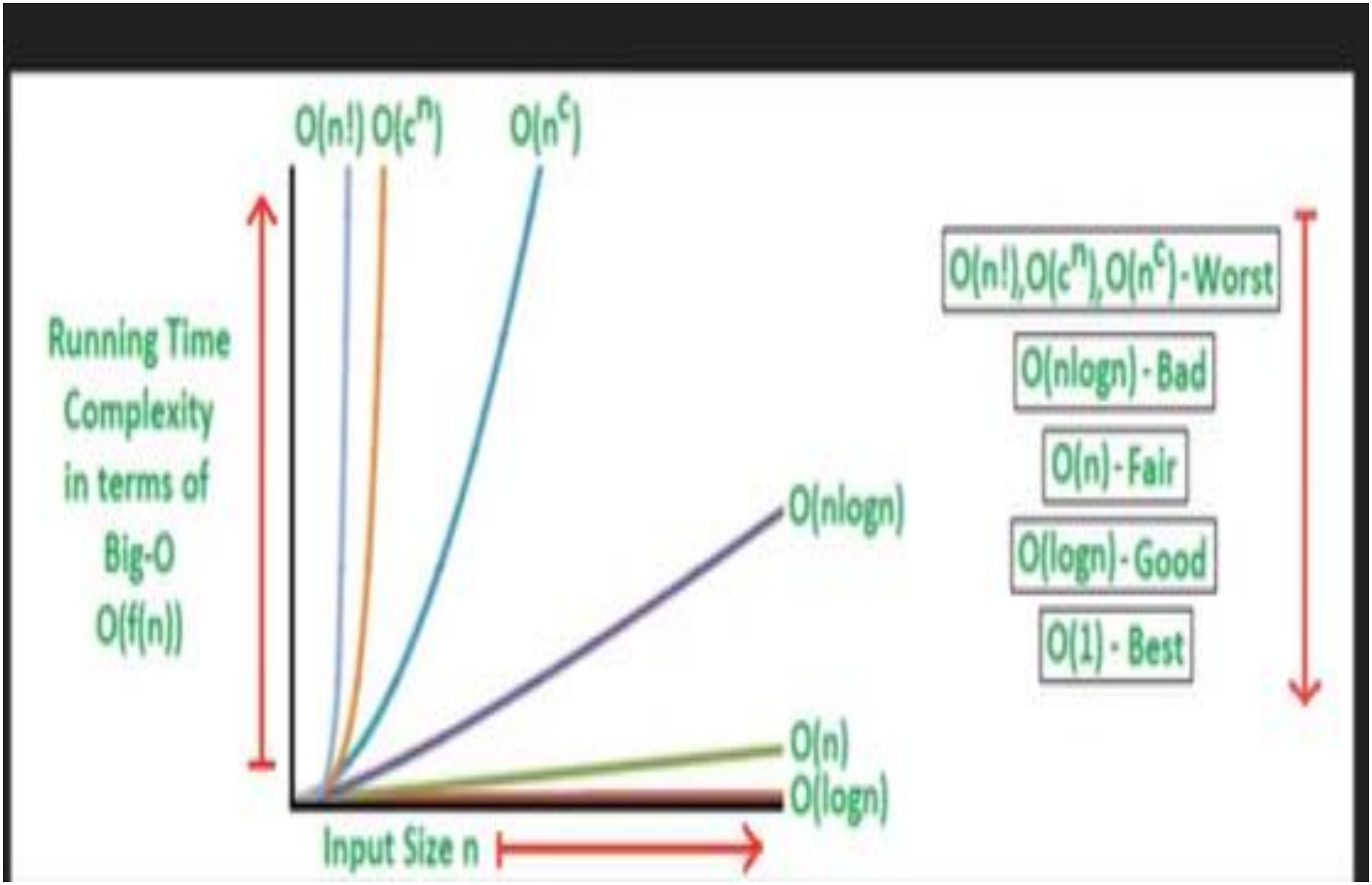
$4n^2+4n+3$

$O(n^2)$

-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)
Sum=0;
for(i=0; i<n; i++)
{
sum=sum+A[i];
}
return sum;
End Function
	0
	1
	2
	
	2
	
	1
		0
		1
		n+1
		
		n
		
		1
			0
			1
			2n+2
			
			2n
			
			1

		O(n)	4n+4

Frequency Count Method	
	Step Count

int sum(int n)
{
int sum=0;
for(i=0; i<n; i++)
sum+=i*i*i
return sum;
}
	
	1
	1+(n+1)+n
	4n
	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,= $3n^2+4$ $O(n^2)$