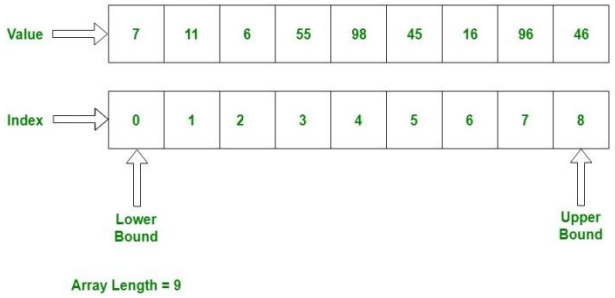
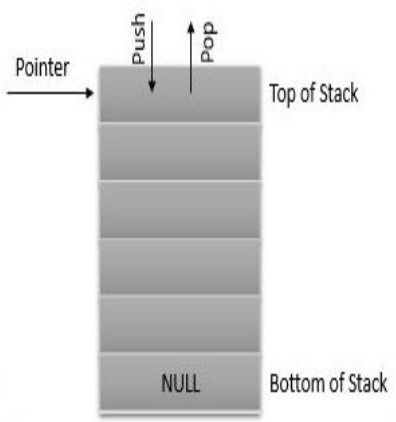
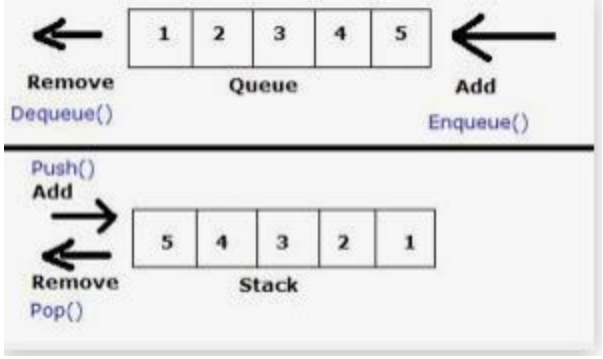
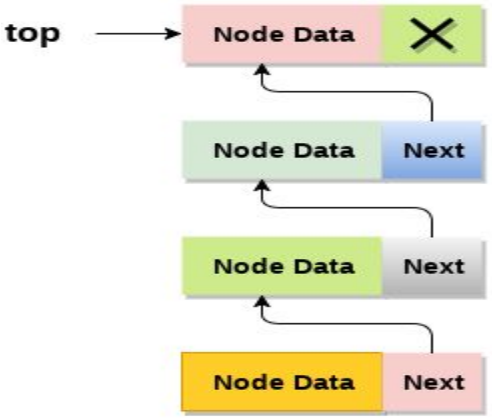
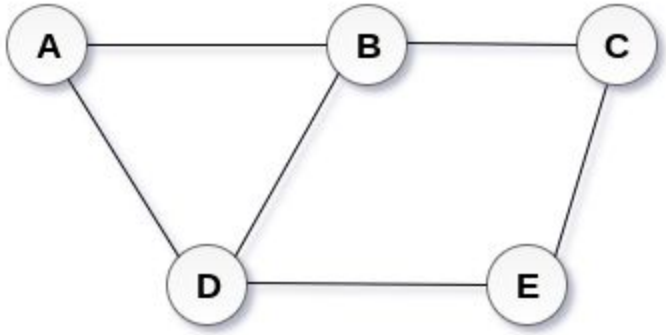


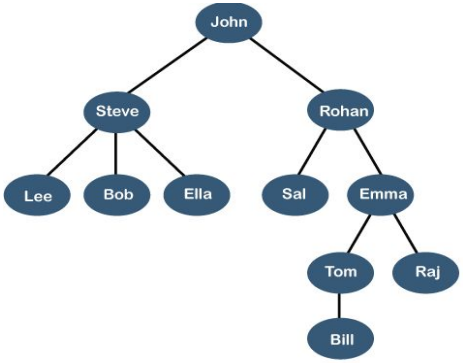




Introduction to Data Structure







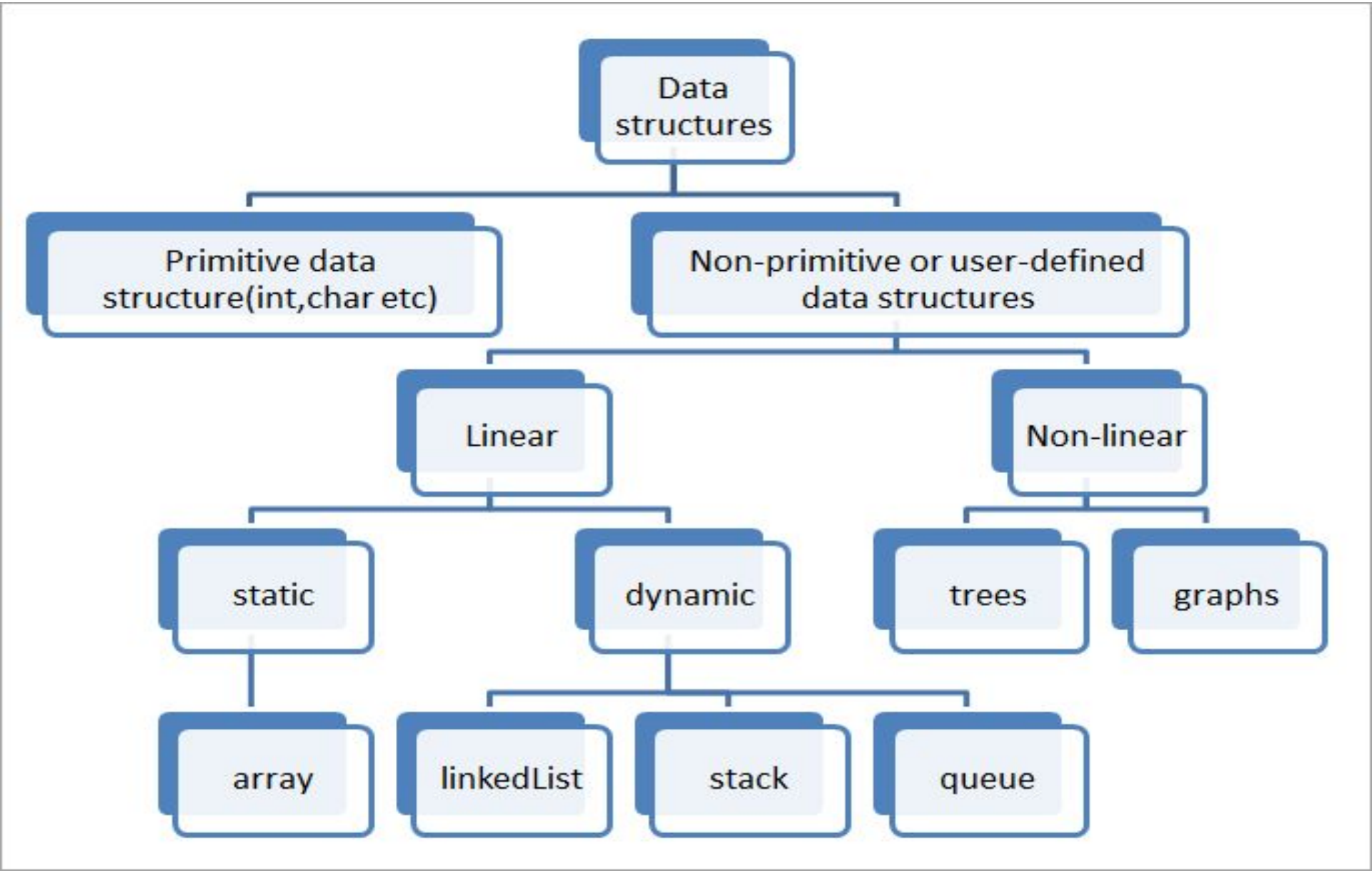
Outline

- Data Structures
- Classification of Data Structures
- Operations of Data Structures
- Define Abstract Data Types
- Introduction to Algorithms
- Approaches of Designing an Algorithm
- Control Structures used in Algorithms
- Understanding Basics of Time Complexity
- Introduction to Asymptotic Notation
- Rate of Growth in Algorithm
- Basics of Storage Management.

What is Data and Data Structure?

- Data structure study covers the following points
  - Amount of memory require to store.
  - Amount of time require to process.
  - Representation of data in memory.
  - Operations performed on that data.

Classification of Data Structure



Primitive/Non-primitive data structures

- **Primitive data structures**
  - Primitive data structures are basic structures and are directly operated upon by machine instructions
  - ***Integers***, ***floats***, ***character*** and ***pointers*** are examples of primitive data structures
- **Non primitive data structure**
  - These are derived from primitive data structures
  - The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items
  - Examples of Non-primitive data type are ***Array***, ***List***, and ***File***

Classification data structures

- **Primitive and Non-Primitive Data Structures**
  - *Primitive data*: Primitive data structures are those data structures that can be directly operated upon the machine instructions. These data structures include int, char, float type of data structures.
  - *Non-Primitive data*: Non-Primitive data structures are those data structures that are derived directly from the primitive data structures. Examples of Non-Primitive data structures include class, structure, array, linked lists.

Classification data structures

- **Homogeneous and Heterogeneous Data Structures**
  - *Homogeneous data*: Homogeneous data structures are those data structures that contain only similar type of data e.g. like a data structure containing only integral values or float values. The simplest example of such type of data structures is an Array.
  - *Heterogeneous Data*: Heterogeneous Data Structures are those data structures that contains a variety or dissimilar type of data, for e.g. a data structure that can contain various data of different data types like integer, float and character. The examples of such data structures include structures, class etc.

Classification data structures

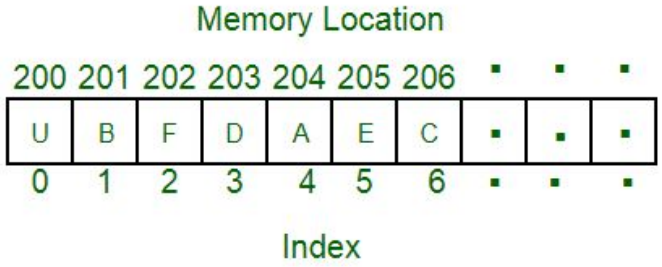
- **Static and Dynamic Data Structures**
  - *Static data*: Static data structures are those data structures to which the memory is assigned at the compile time, means the size of such data structures has to be predefined in the program and their memory can be increased or decreased during the execution. The simplest example of static data structure is an array.
  - *Dynamic data*: Dynamic Data Structures are those data structures to which memory is assigned at the runtime or execution time and hence their size can be increased or decreased dynamically as per requirement of the program. The example of such data structures include class, linked lists etc.

Classification data structures

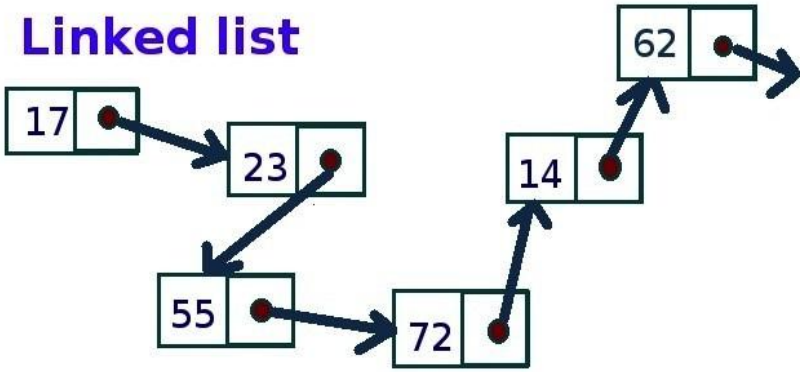
- **Linear and Non-Linear Data Structures**
  - *Linear data*: In linear data structures the data is kept in the sequential manner which means that the previous data has knowledge of next data or we can say that the data is interrelated. The example such data structures include arrays, linked lists etc.
  - *Non-linear data*: Non-linear data structures does not maintain any kind of relationship among the data. The data is stored in non-sequential manner and examples of these types of data structures includes graphs, trees etc.

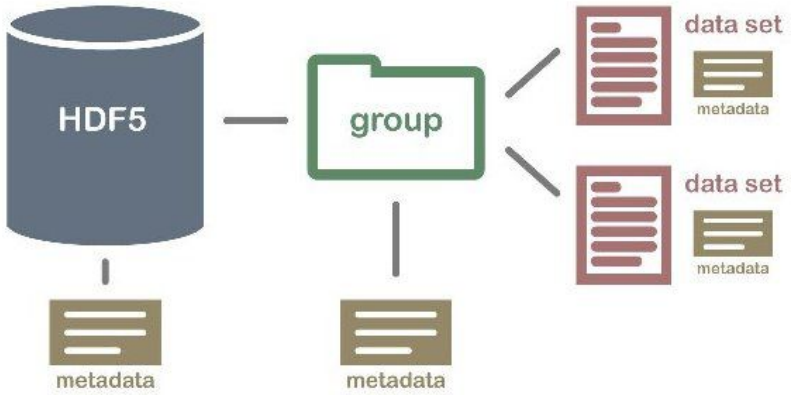
Non-primitive data structures

- **Array:** An array is a fixed-size sequenced collection of elements of the same data type.



- **List:** An ordered set containing variable number of elements is called as Lists.

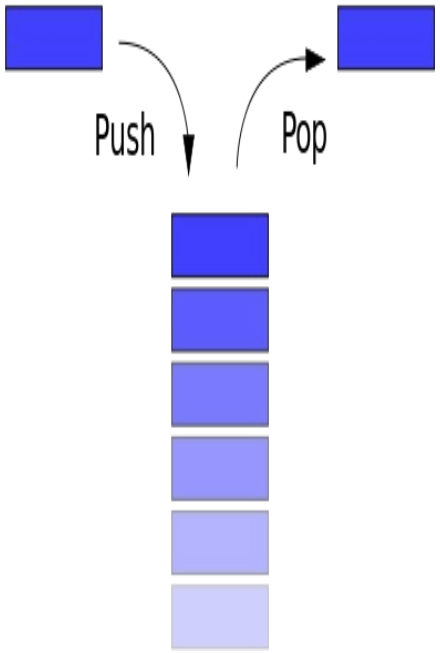


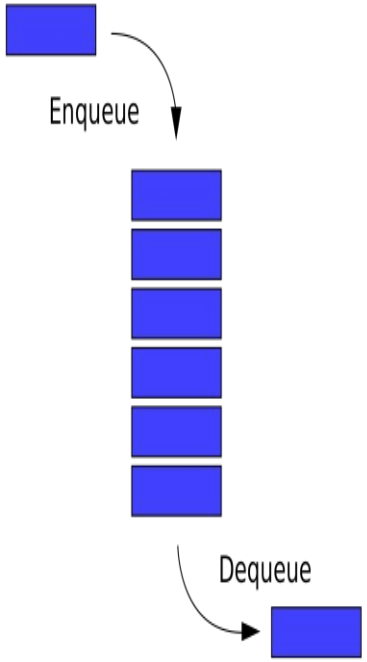


- **File:** A file is a collection of logically related information. It can be viewed as a large list of records consisting of various fields.

Linear / Non-Linear data structure

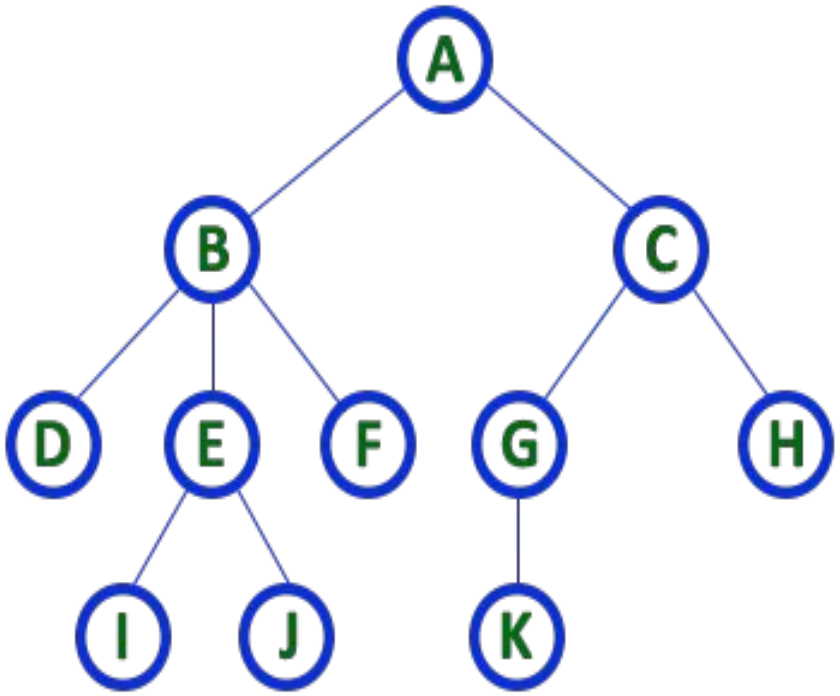
- **Linear data structures**
  - A data structure is said to be Linear, if its elements are connected in linear fashion by means of logically or in sequence memory locations
  - Examples of Linear Data Structure are ***Stack [LIFO]*** and ***Queue[FIFO]***.

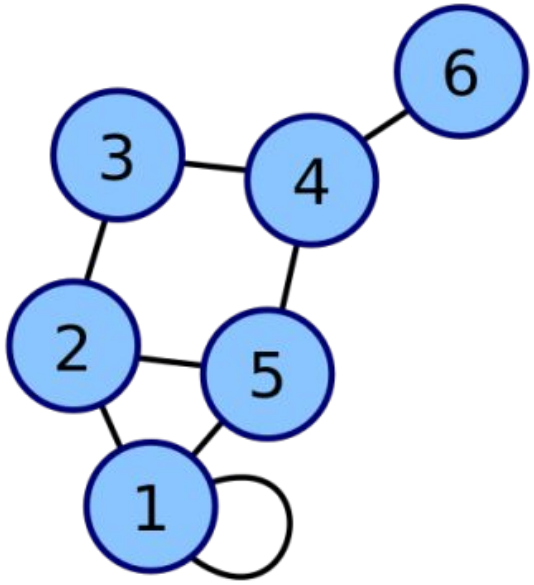




Linear / Non-Linear data structure

- **Nonlinear data structures**
  - Nonlinear data structures are those data structure in which data items are not arranged in a sequence
  - Examples of Non-linear Data Structure are ***Tree*** and ***Graph***





Operations of Data Structure

- **Create**: it results in reserving memory for program elements
- **Destroy**: it destroys memory space allocated for specified data structure
- **Selection**: it deals with accessing a particular data within a data structure
- **Updating**: it updates or modifies the data in the data structure
- **Searching**: it finds the presence of desired data item in the list of data items
- **Sorting**: it is a process of arranging all data items in a data structure in a particular order. For example, preparing roll number students are sorted on surname
- **Merging**: it is a process of combining the data items of two different sorted list into a single sorted list

Operations of Data Structure

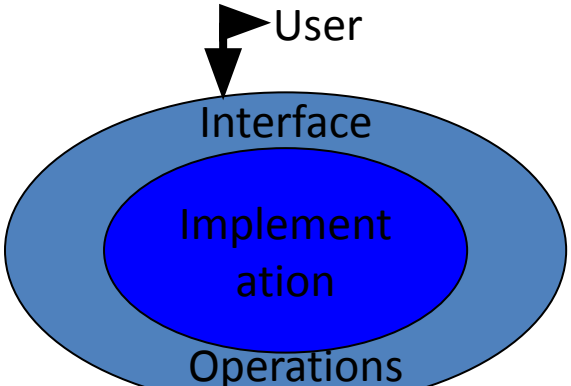
- **Splitting**: it is a process of partitioning single list to multiple list
- **Traversal**: it is a process of visiting each and every node of a list in systematic manner exactly once. For example, to print the names of all the students in a class.
- ***Inserting:*** It is used to add new data items to the given list of data items. For example, to add the details of a new student who has recently joined the course.
- ***Deleting:*** It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.

ABSTRACT DATA TYPE

- An abstract data type (ADT) is the way we look at a data structure, focusing on what it does and ignoring how it does its job.
- For example, stacks and queues are perfect examples of an ADT.
- We can implement both these ADTs using an array or a linked list.
- 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.

ABSTRACT DATA TYPE

- In C, an Abstract Data Type can be a structure considered without regard to its implementation.
- It can be thought of as a "description" of the data in the structure with a list of operations that can be performed on the data within that structure.
- The end user is not concerned about the details of how the methods carry out their tasks.
- They are only aware of the methods that are available to them and are concerned only about calling those methods and getting back the results but not HOW they work.



Algorithm

- An ***algorithm*** is a finite set of instructions that accomplishes a particular task.
  - Criteria

    input: zero or more quantities that are externally supplied
    output: at least one quantity is produced
    definiteness: clear and unambiguous
    finiteness: terminate after a finite number of steps
    effectiveness: instruction is basic enough to be carried out
  - A program does not have to satisfy the **finiteness** criteria.
  - Representation
    - A natural language, like English or Chinese.
    - A graphic, like flowcharts.
    - A computer language, like C. [Sequential search vs. Binary search]

Algorithm

- Algorithms are mainly used to achieve software reuse.
- Once we have an idea or a blueprint of a solution, we can implement it in any high-level language like C, C++, or Java.
- An algorithm is basically a set of instructions that solve a problem.
- It is not uncommon to have multiple algorithms to tackle the same problem, but the choice of a particular algorithm must depend on the time and space complexity of the algorithm.
- Good Algorithm : Run in less time , Consume less memory. But computational resources (time complexity) is usually more important

Time and space analysis of algorithms

- Sometimes, there are more than one way to solve a problem.
- We need to learn how to compare the performance different algorithms and choose the best one to solve a particular problem.
- While analyzing an algorithm, we mostly consider time complexity and space complexity
- ***Time complexity*** of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input
- ***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

Time and space analysis of algorithms

- Time & space complexity depends on lots of things like hardware, operating system, processors, etc.
- However, we don't consider any of these factors while analyzing the algorithm. We will only consider the execution time of an algorithm

What is Space complexity?

- When we design an algorithm to solve a problem, it needs some computer memory to complete its execution.
- For any algorithm, memory is required for the following purposes…
  - To store program instructions.
  - To store constant values.
  - To store variable values.
  - And for few other things like function calls, jumping statements etc,.
- Space complexity of an algorithm can be defined as follows…
  - Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.

What is Space complexity?

- To calculate the space complexity, we must know the memory required to store different datatype values (according to the compiler). For example, the C Programming Language compiler requires the following…
  - 2 bytes to store Integer value.
  - 4 bytes to store Floating Point value.
  - 1 byte to store Character value.
  - 6 (OR) 8 bytes to store double value.

What is Time complexity?

- Every algorithm requires some amount of computer time to execute its instruction to perform the task.
- This computer time required is called time complexity.
- The time complexity of an algorithm can be defined as follows..
  - The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

What is Time complexity?

- Generally, the running time of an algorithm depends upon the following…
  - Whether it is running on **Single** processor machine or **Multi** processor machine.
  - Whether it is a **32 bit** machine or **64 bit** machine.
  - **Read** and **Write** speed of the machine.
  - The amount of time required by an algorithm to perform **Arithmetic** operations, **logical** operations, **return** value and **assignment** operations etc.,
  - **Input** data

Calculating Time Complexity

Calculate Time Complexity of Sum of elements of List (One dimensional Array)

A is array, n is no of elements in array

```
SumOfList(A,n)
{
Line 1   total = 0
Line 2   for i = 0 to n-1
Line 3     total = total + A[i]
Line 4   return total
}
```

Line	Cost	No of Times
1	1	1
2	2	n + 1
3	2	n
4	1	1

TSumOfList = 1 + 2 (n+1) + 2n + 1

We can neglate constant 4

= 4n + 4

= n

Time complexity of given algorithm is ***n*** unit time

What is Time complexity?

- Determine the frequency counts of each statements of the following codes

1. x = x + y
2. for i = 1 to n do
       x = x + y
3. for i = 1 to n do
       for j = 1 to n do
           x = x + y

What is Time complexity?

- Determine the frequency counts of each statements of the following codes

1. The frequency count is 1
2. The frequency count is n
3. The frequency count is n^2

How to calculate Time complexity?

- Now the most common metric for calculating time complexity is *Big O* notation. This removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity. In general you can think of it like this : *statement;*
- Above we have a single statement. Its Time Complexity will be ***Constant***. The running time of the statement will not change in relation to N.

- `for(i=0; i < N; i++) {statement;}`
- The time complexity for the above algorithm will be ***Linear***. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

How to calculate Time complexity?

- for(i=0; i < N; i++){
  for(j=0; j < N;j++){
  statement;}
  }
- This time, the time complexity for the above code will be ***Quadratic***. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.

How to calculate Time complexity?

```
while(low <= high){
 mid = (low + high) / 2;
 if (target < list[mid])
   high = mid – 1;
 else if (target > list[mid])
   low = mid + 1;
 else
 break;}
```

- This is an algorithm to break a set of numbers into halves, to search a particular field. Now, this algorithm will have a ***Logarithmic*** Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2(N is high-low here).This is because the algorithm divides the working area in half with each iteration.

How to calculate Time complexity?

- void quicksort(int list[], int left, int right){
  int pivot = artition(list, left, right);
  quicksort(list, left, pivot - 1);
  quicksort(list, pivot + 1, right);
  }
- Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration N times(where N is the size of list). Hence time complexity will be **N*log(N)**. The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

What is Asymptotic Notation?

- Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm.
- But when we calculate the complexity of an algorithm it does not provide the exact amount of resource required.
- We use that general form (Notation) for analysis process.
- Asymptotic notation of an algorithm is a mathematical representation of its complexity.
- **Note -** In asymptotic notation, when we want to represent the complexity of an algorithm, we use only the most significant terms in the complexity of that algorithm and ignore least significant terms in the complexity of that algorithm (Here complexity can be Space Complexity or Time Complexity).

What is Asymptotic Notation?

- For example, consider the following time complexities of two algorithms…
  - **Algorithm 1 : $5n^2 + 2n + 1$**
  - **Algorithm 2 : $10n^2 + 8n + 3$**
- When we analyze an algorithm, we consider the time complexity for larger values of input data (i.e. **'n'** value). In above two time complexities, for larger value of **'n'** the term **'2n + 1'** in algorithm 1 has least significance than the term **'$5n^2$'**, and the term **'8n + 3'** in algorithm 2 has least significance than the term **'$10n^2$'**.
- For larger value of **'n'** the value of most significant terms (**$5n^2$** and **$10n^2$**) is very larger than the value of least significant terms (**2n + 1** and **8n + 3**).

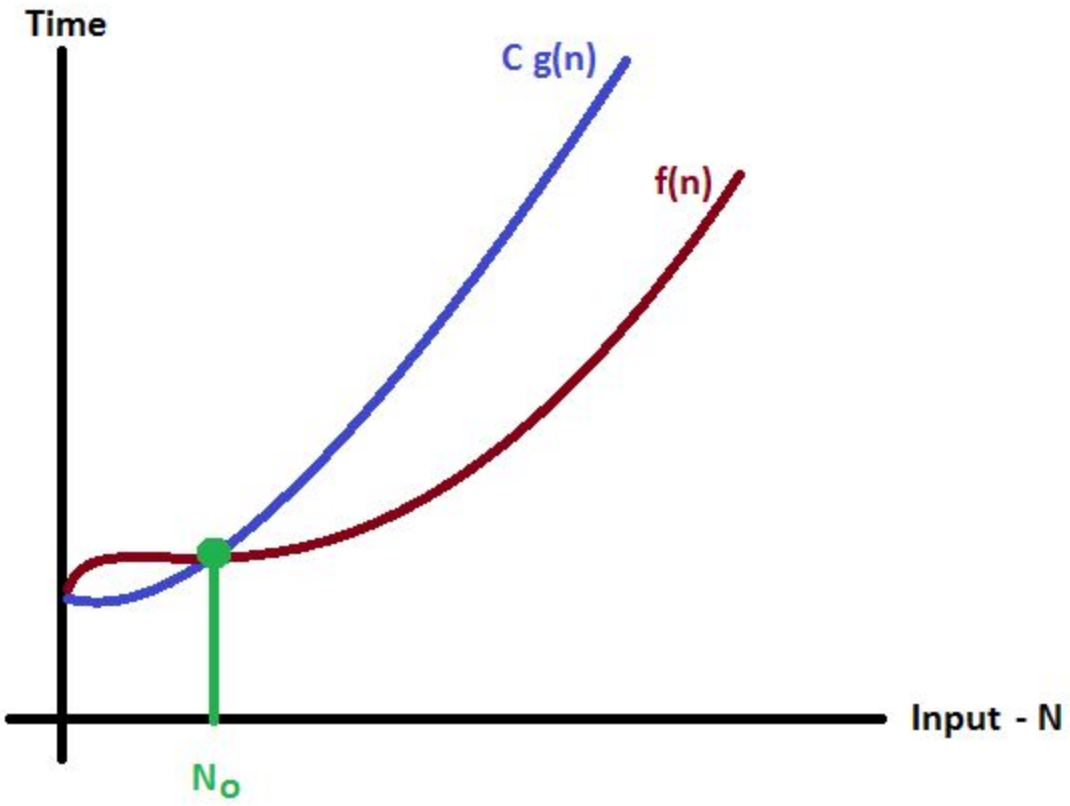
What is Asymptotic Notation?

- So for larger value of **'n'** we ignore the least significant terms to represent overall time required by an algorithm. In asymptotic notation, we use only the most significant terms to represent the time complexity of an algorithm.
- Majorly, we use THREE types of Asymptotic Notations and those are as follows…
  - **Big - Oh (O)**
  - **Big - Omega (Ω)**
  - **Big - Theta (Θ)**

Big - Oh Notation (O)

- Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.
- That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values.
- That means Big - Oh notation describes the worst case of an algorithm time complexity.
- Big - Oh Notation can be defined as follows…
  - Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as O(g(n)).
  - f(n) = O(g(n))
- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis

Big - Oh Notation (O)



- In above graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.

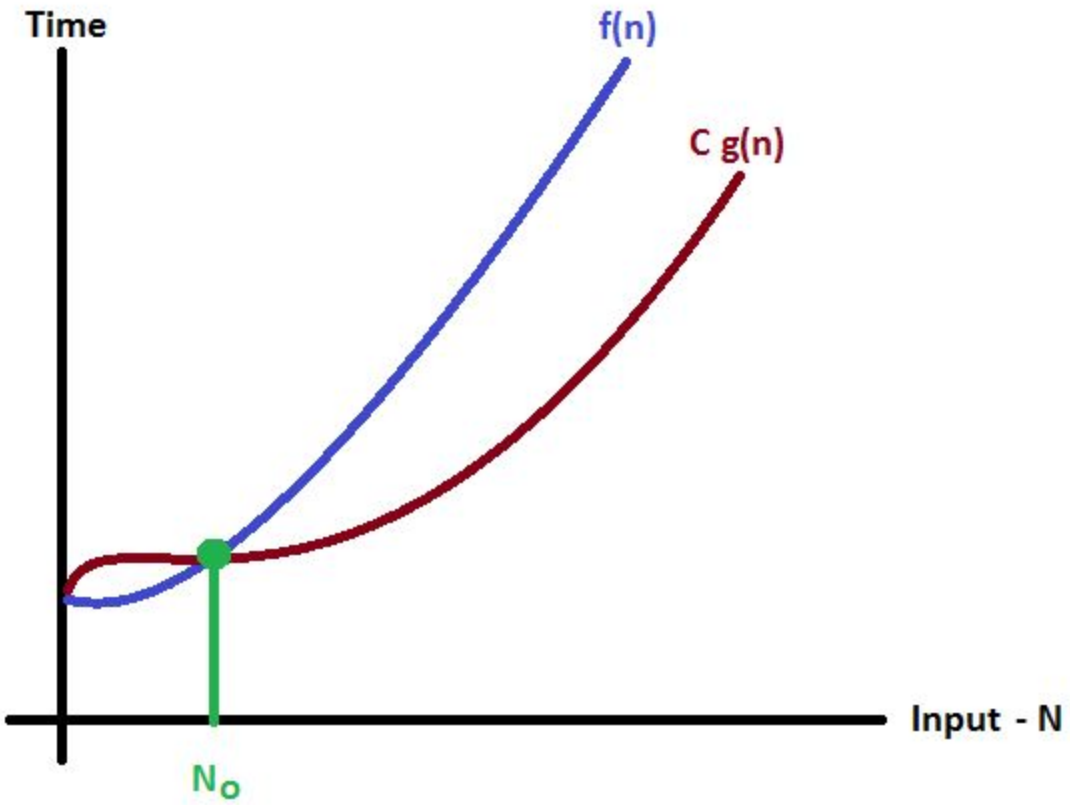
Big - Oh Notation (O)

- Example
  Consider the following f(n) and g(n)…
  f(n) = 3n + 2
  g(n) = n
- If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C g(n)** for all values of **C > 0** and **n_0>= 1**
- f(n) <= C g(n)
  ⇒3n + 2 <= C n
- Above condition is always TRUE for all values of **C = 4** and **n >= 2**.
- By using Big - Oh notation we can represent the time complexity as follows...
  3n + 2 = O(n)

Big - Omege Notation (Ω)

- Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.
- That means Big-Omega notation always indicates the minimum time required by an algorithm for all input values.
- That means Big-Omega notation describes the best case of an algorithm time complexity.
- Big - Omega Notation can be defined as follows…
  - Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term.
- If f(n) >= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as Ω(g(n)).
  f(n) = Ω(g(n))
- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis

Big - Omege Notation (Ω)



- In above graph after a particular input value n_0, always C g(n) is less than f(n) which indicates the algorithm's lower bound..

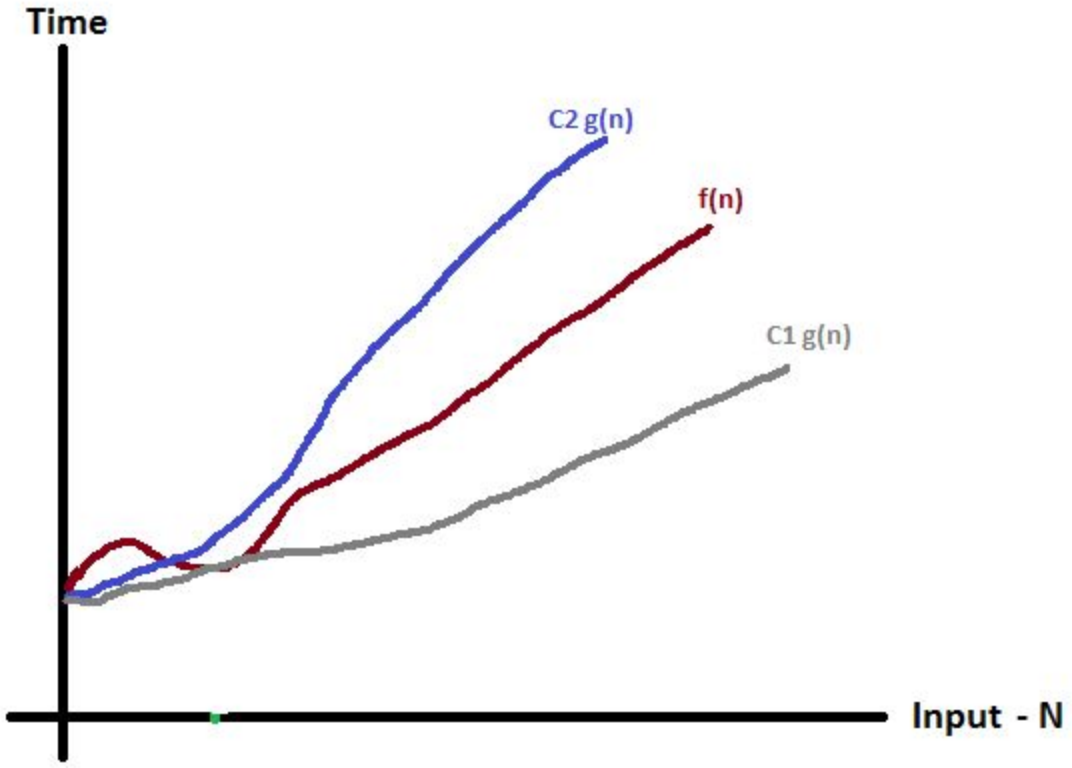
Big - Omege Notation (Ω)

- Example
  Consider the following f(n) and g(n)…
  f(n) = 3n + 2
  g(n) = n
- If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**
  f(n) >= C g(n)
  ⇒3n + 2 >= C n
  Above condition is always TRUE for all values of **C = 1** and **n >= 1**.
- By using Big - Omega notation we can represent the time complexity as follows...
  3n + 2 = Ω(n)

Big - Theta Notation (Θ)

- Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.
- That means Big - Theta notation always indicates the average time required by an algorithm for all input values.
- That means Big - Theta notation describes the average case of an algorithm time complexity.
- Big - Theta Notation can be defined as follows…
  - Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If C_1 g(n) <= f(n) <= C_2 g(n) for all n >= n_0, C_1 > 0, C_2 > 0 and n_0 >= 1. Then we can represent f(n) as Θ(g(n)).
  - f(n) = Θ(g(n))
- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis

Big - Theta Notation (Θ)



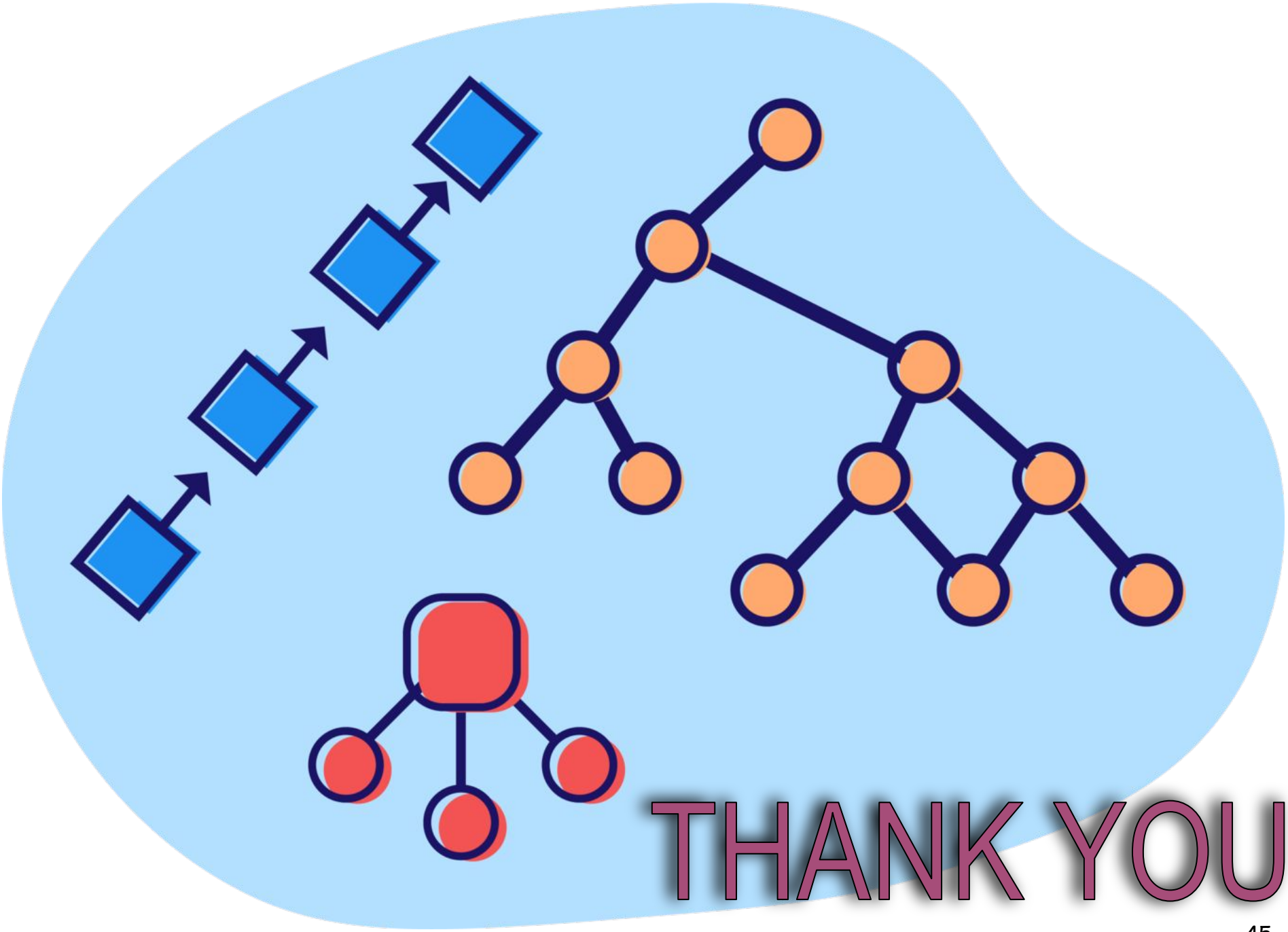
- In above graph after a particular input value n_0, always C_1 g(n) is less than f(n) and C_2 g(n) is greater than f(n) which indicates the algorithm's average bound.

Big - Theta Notation (Θ)

- Example
  Consider the following f(n) and g(n)…
  f(n) = 3n + 2
  g(n) = n
- If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy C_1 **g(n) <= f(n) <=** C_2 **g(n)** for all values of C_1 **> 0,** C_2 **> 0** and n_0**>= 1**
  C_1 g(n) <= f(n) <= C_2 g(n)
  ⇒C_1 n <= 3n + 2 <= C_2 n
  Above condition is always TRUE for all values of C_1 **= 1,** C_2 **= 4** and **n >= 2**.
- By using Big - Theta notation we can represent the time complexity as follows...
  3n + 2 = Θ(n)

Simple Example : Growth of 5n+3

- Estimated running time for different values of N:
  - N = 10 => 53 steps
  - N = 100 => 503 steps
  - N = 1,000 => 5003 steps
  - N = 1,000,000 => 5,000,003 steps
  - As N grows, the number of steps grow in *linear* proportion to N for this function *"Sum"*
  - What about the +3 and 5 in 5N+3?
    - As N gets large, the +3 becomes insignificant
    - 5 is inaccurate, as different operations require varying amounts of time and also does not have any significant importance
  - What is fundamental is that the time is *linear* in N.
- Asymptotic Complexity: As N gets large, concentrate on the highest order term:
  Drop lower order terms such as +3
  Drop the constant coefficient of the highest order term i.e. N


THANK YOU