MODULE – I

Introduction

1.1. Data Structures Introduction:

A data structures is a internal representation of the ordered state that exists between respective data components. In opposite words, data structures describes a method of handle all data items that takes into account not just the pieces contained but also their relationships to one another. The phrase data structures refers to the method by which data is accumulation.

To create an algorithm or program, we must first choose an acceptable data structure for that algorithm. As a result, the data structure is represented as:

Algorithms + Data Structures = Programmes

The elements of a data structures are said to be linear if they form a sequence or a linear list. Linear data structures, such as arrays, stack, queue, and linked list, manage data in a additive fashion. A data structures is said to be nonlinear if its elements create a hierarchic classification with data items appearing at different levels.

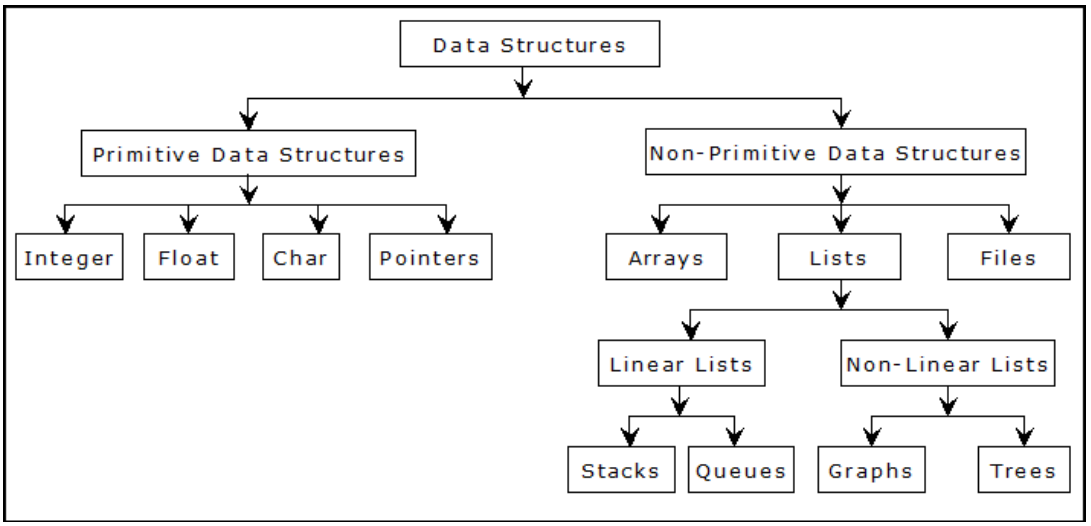
Non-linear data structure such as tree and graph are commonly employed. The hierarchical relationship between individual data pieces is represented by tree and graph structures. Graphs are simply trees with some constraints removed.

There are two kinds of data structures:

1. Data structure that are fundamental (Primitive).
2. Data structure that are not primitive (Non primitive).

Primitive Data Structures are the fundamental data structures that act directly on computer instructions. On different computers, they have different representations. This category includes integers, floating point numbers, character constants, string constants, and pointers.

Non-primitive data structures are more sophisticated than primitive data structures. They place emphasis on grouping similar or dissimilar data items with relationships between them. Arrays, lists, and files all fall within this category.

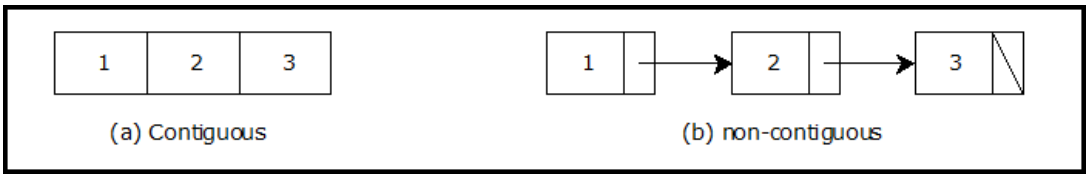


Data Structure Classification

Data structures: Organization of data

A program's data collecting has some form of structure or order to it. No matter how complicated your data structures are, they can be divided into two basic types: Non-contiguous vs. contiguous.

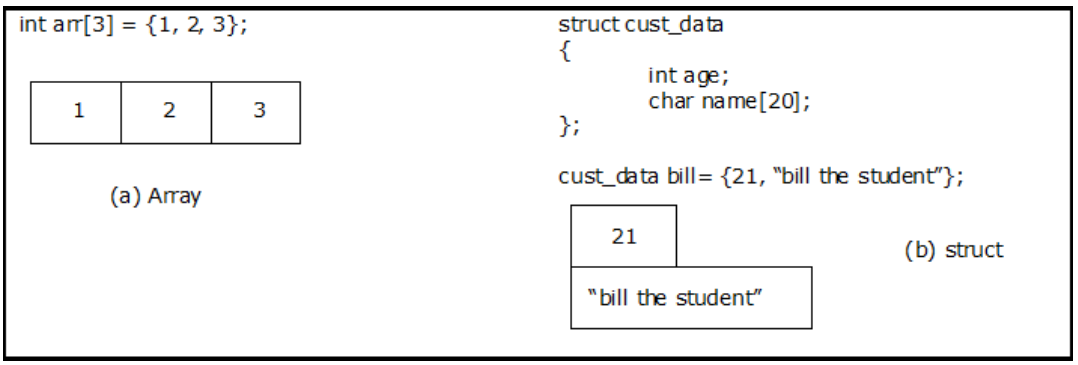
Terms of data are saved together in memory (either RAM or a file) in contiguous structures. A continuous structure is something like an array. Because each array element is adjacent to other elements. Items in a non immediate structure and disconnected throughout memory, on the other hand, were linked to each other in some way. Non-contiguous data structures include linked lists. The list's nodes are linked together via pointers contained in each node.



Contiguous structures:

Contiguous structures are further classified into two types: those containing data item of all the identical size and those containing data items of varying sizes. The first type is known as an arrays. For each one element in an array is of the corresponding type and so has the identical size.

Structure is the second form of contiguous structure. Elements in a struct might be of different data types and thus have varied sizes.



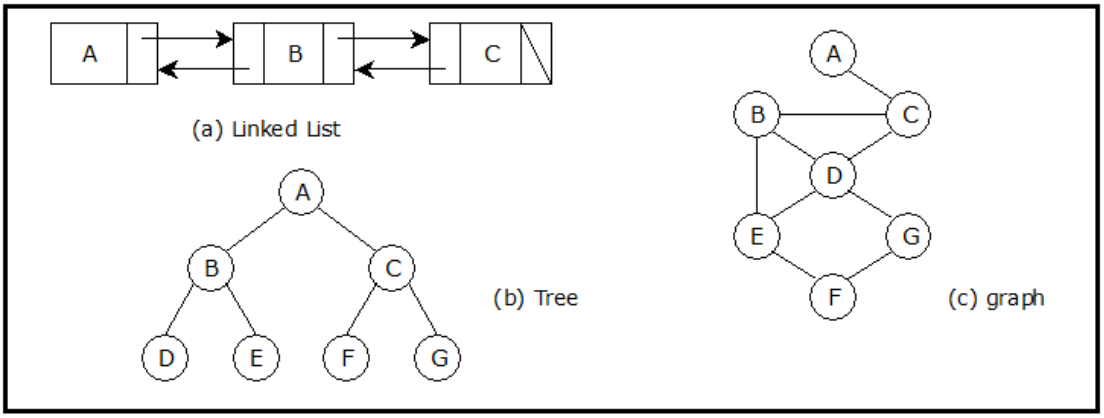
Examples of continuous structure

Non contiguous structures:

Non-contiguous structure is implemented as a set of data-items called nodes, with each node pointing to one or more other nodes in the set. The linked list is the most basic type of non-contiguous structure.

A linked list is a linear, one-dimensional type of non-contiguous structure using only backwards and forwards notation. There is the concept of up and down, as well as left and right.

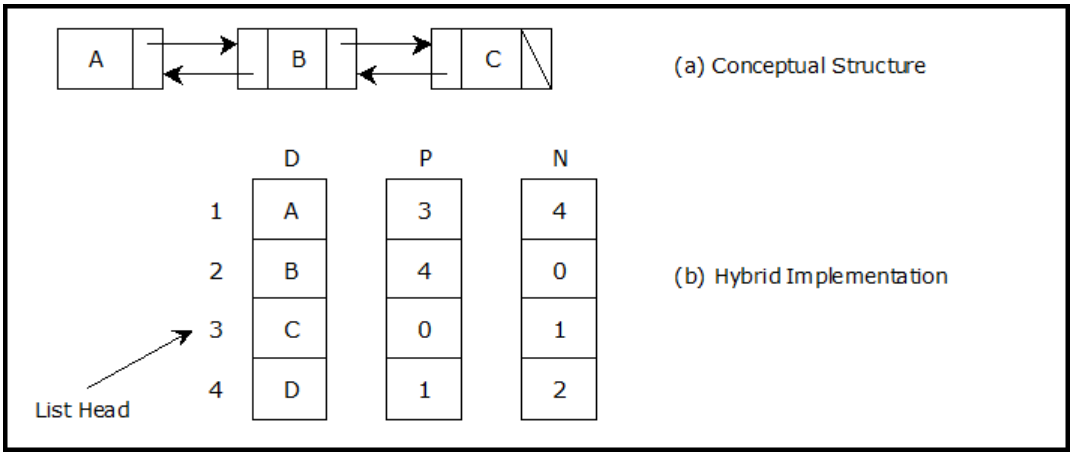
Each node in a trees has only one link that leads into it, and link can only go down the tree. A graph, the most unspecific sort of non contiguous structure, has no such constraints.



Examples of non continuous structure

Hybrid structures:

A hybrid form is formed when two basic types of structure are combined. Then there's a contiguous section and a non contiguous section.



Data Structure type of Hybrid

The array D holds the list's data, while the arrays P and N hold the previous and next "pointers." The pointers are basically just indexes into the D array. For example, D[i] carries the data for node i, but p[i] holds the index to the node before i, which may or may not be at position i-1. N[i] similarly holds the index to the next node in the list.

1.2 Basic Concepts and Notation:

There is a need for displaying higher level data from basic information and structures available at the machine level while constructing a solution in the problem solving process.

There is also a need for algorithm synthesis from fundamental machine-level operations to modify higher-level representations. These two are crucial in achieving the intended result. Data structures are required for data representation, whereas algorithms are required for data operations to yield correct results.

Data

These are truths that have meaning in a certain context and are represented by values, numbers, letters, and symbols.

Information

These are pieces of information that have value to the user or have been interpreted by the user.

When data is analyzed in a given circumstance or utilized to solve a problem, it becomes information.

- Data must be represented (named) or saved in order to be retrieved later.
- Data must be arranged so that it may be accessed selectively and efficiently.

-Data must be processed and displayed in such a way that it effectively supports the user's surroundings.

- Data must be safeguarded and handled in order to keep their worth.

1.3. Abstract Data Type (ADT):

A data structure design entails more than just its organization. You must also plan for how the data will be accessed and processed - that is, how the data will be interpreted. Non-contiguous structures, such as lists, trees, and graphs, can be implemented either contiguously or non-contiguously. Similarly, structures that are normally treated as contiguously, such as arrays and structures, can also be implemented non-contiguously.

The abstract concept of a data structure must be regarded differently than whatever is utilized to create the structure. The abstract concept of a data structure is determined by the operations we intend to execute on the data.

Abstract Data types = Data structures + Methods on the structures

Consideration of both data arrangement and expected operations on data leads to the concept of an abstract data type. An abstract data type is a theoretical construct that includes data as well as operations to be done on the data while concealing implementation.

A stack, for example, is an example of an abstract data type. Stack items can only be added and removed in a specific order - the last thing added is the first item removed. These operations are known as pushing and popping. We haven't indicated how objects are stored on the stack or how they are pushed and popped in this description. Only the valid operations that can be done have been mentioned.

To read a file, for example, we built the code to read the physical file device. That is, we may have to write the same code several times. As a result, we developed what is now known as an ADT. We built the code to read a file and stored it in a library for future use by programmers.

Another example is the code for reading from a keyboard, which is an ADT. It has a data structure, a character, and a set of operations for reading that data structure.

An abstract data type (such as stack) must be implemented before it can be used, which is where data structure comes into play. For example, we might use an array to represent the stack and then specify the relevant indexing methods to perform pushing and popping.

1.5. Algorithm

An algorithm is a finite sequence of instructions, each with a distinct meaning and capable of being completed with a finite amount of work in a finite period of time. An algorithm finishes after processing a finite amount of instructions, regardless of the input values. Furthermore, every algorithm must meet the following requirements:

Input: There are zero or more externally supplied quantities.

Output: At least one amount is produced as an output.

Definiteness: Each instruction must be precise and unambiguous.

Finiteness: If we trace out an algorithm's instructions, the method will always terminate after a finite number of steps.

Effectiveness: Every instruction must be simple enough that it can be carried out by a person utilizing only a pencil and paper. It is not enough for each operation to be specific; it must also be doable.

A distinction is made in formal computer science between an algorithm and a program. The fourth criteria is not always met by a program. One significant example of such a program for a computer is its operating system, which never ends (unless in the case of a system crash) but instead continues in a wait loop until new jobs are entered.

We represent an algorithm using pseudo language, which is a blend of computer language constructs and informal English remarks.

1.6. Analysis and Efficiency of Algorithms:

The selection of an efficient algorithm or data structure is only one aspect of the design process. Following that, we will look at some bigger design challenges. In a program, we should strive for three core design goals:

1. Time complexity: Try to save time.
2. Space complexity: Try to save space.
3. Try to have face.

A faster program is a better program, thus saving time is an apparent goal. A program that saves space over a rival software is also beneficial. We want to "save face" by preventing the application from freezing or producing massive amounts of distorted data.

If 'n' denotes the number of data items to be processed, the degree of polynomial, the size of the file to be sorted or searched, the number of nodes in a graph, and so on.

1: Most programs' next instructions are only executed once or a few times. If all of a program's instructions have this quality, the program's running time is said to be constant.

Log n: When a program's execution time is logarithmic, the program becomes slightly slower as n increases. This running time is frequent in programs that solve a large problem by changing it into a smaller problem, reducing the size by a constant fraction. When n is a million, log n is a constant that doubles every time n doubles, but log n does not double until n reaches n2.

n: When a program's execution time is linear, only a tiny amount of processing is performed on each input piece. This is the ideal circumstance for an algorithm with n inputs to process.

n.logn: This occurs for algorithms that solve a problem by dividing it down into smaller sub-problems, solving them separately, and then combining the solutions. The running time more than doubles when n doubles.

n^2: When an algorithm's execution time is quadratic, it can only be used on relatively modest problems. Quadratic running times are common in algorithms that process all pairs of data items (perhaps in a double nested loop); when n doubles, the running time quadruples.

n^3: Similarly, a method that processes triples of data items (perhaps in a triple-nested loop) has a cubic running time and is only applicable to modest tasks. When n doubles, the running time increases by an order of magnitude.

2^n: Few algorithms with exponential running times are likely to be useful in practice; such algorithms naturally emerge as "brute-force" solutions to issues.
The running time squares whenever n doubles.

1.7. Time and Space Complexity:

A program's performance is defined as the amount of computer memory and time required to run it. To assess a program's performance, we employ two methods. The first is analytical, whereas the second is experimental. We employ analytical methods in performance analysis and experiments in performance assessment.

Time Complexity:

The time required by an algorithm expressed as a function of issue size is referred to as the algorithm's TIME COMPLEXITY. A program's time complexity is the amount of computer time required to complete it.

Asymptotic temporal complexity refers to the limiting behavior of complexity as size grows. The asymptotic complexness of an algorithm eventually dictates the property of issues that the algorithm can solve.

Space Complexity:

A program's space complexity is the quantity of memory required to test it to ending. A program's space requirements include the following elements:

Instruction space is the amount of space required to hold the compiled version of the program instructions.

Data space is the amount of space required to store all constant and variable values. Data space is divided into two parts:

1. space required by constants and simple variables in programs.

2.Dynamically allocated objects, such as arrays and class instances, require space.

Environment stack space is used to save information required to resume execution of partially completed functions.

The quantity of instructions space required is determined by factors such as:

1. The compiler used to convert the program into machine code.
2. The compiler options that were in effect at the time of compilation.

Complexity of Algorithms

An algorithm's complexity M is the function f(n) that gives the algorithm's execution time and/or storage space need in terms of the size 'n' of the input data. Typically, an algorithm's storage space need is just a multiple of the data size 'n'. The running time of the algorithm is referred to as its complexity.

An algorithms asymptotic analytic thinking pertains to defining the numerical boundation / framing of its run time performance. We may very well deduce the best case, average case, and worst case scenarios of an algorithm using asymptotic analysis.

Asymptotic analysis is input bound, which means that if the algorithm receives no input, it is assumed to work in a constant time. Except for the "input," all other elements are assumed to be constant.

The computations of the run time of any activity in mathematical units of computation is referred to as asymptotic analysis. For representation, the run time of one activity

may be computed as f(n), whereas the run time of another activity may be computed as g(n2).

This means that the first operation's running time will increase linearly as n increases, while the second operation's running time will increase exponentially as n increases. Similarly, if n is really small, the running time of both operations will be virtually the same.

There are three sorts of time required by an algorithm.

Minimum time necessary for program execution in the best-case scenario.

The average amount of time taken for program execution in the Average case scenario.

Maximum time necessary for program execution in the worst-case scenario.

Asymptotic Notations

The below are some common asymptotic-notations for calculating an algorithms run time complexity.

Big -O- Notation
Big- Ω -Notation
Big- θ -Notation

Big-Oh-Notation (O):

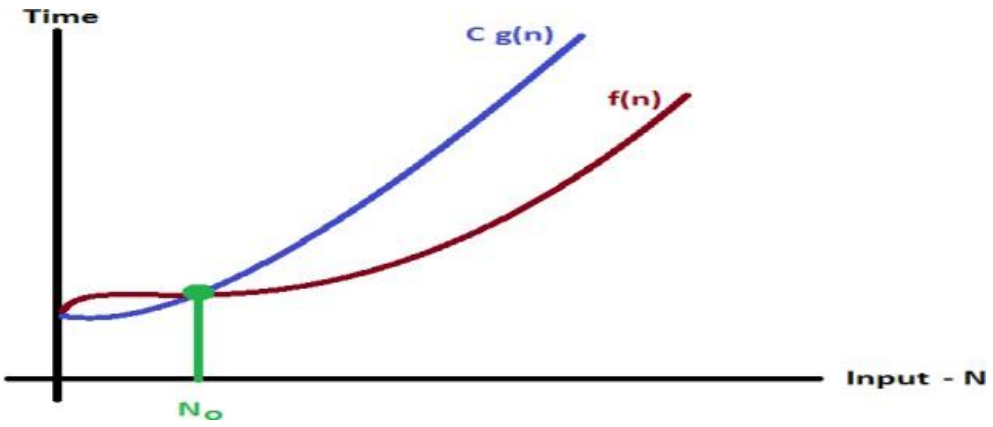
1.Big - Oh notation is used to define an algorithm's upper bound in terms of Time Complexity.

2.Big - Oh notation always represents the maximum time an algorithm requires for all input values.

3.Big - Oh notation describes the worst-case time complexity of an algorithm.

Big - Oh Notation is defined as follows... The graph below depicts the values of f(n) and C g(n) for input (n) value on the X-Axis and time required on the Y-Axis time complexity of an algorithm.

In the graph below, for a certain input value n0, C g(n) is always bigger than f(n), indicating the algorithm's upper bound.



Example

Consider the f(n) and g(n) expressions...

f(n) = 3n + 2

g(n) = n

If we want to describe f(n) as O(g(n)), it must satisfy f(n) = C g(n) for any C > 0 and n0 > 1 values.

f(n) <= C g(n)

⇒3n + 2 <= C n

For all values of C = 4 and n >= 2, the above condition is always TRUE.

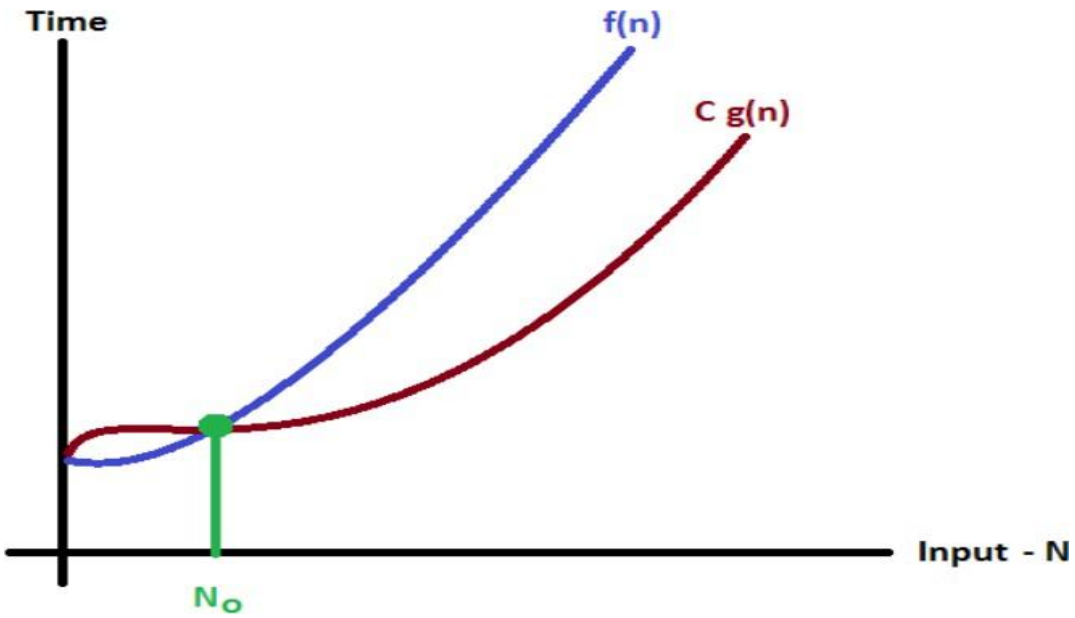
We can represent the time complexity as follows using Big - Oh notation-

3n + 2 = O(n).

Big - Omege-Notation (Ω):

1.Big - Omega notation is used to define an algorithm's lower bound in terms of Time Complexity.
2. Big-Omega notation always represents the shortest time an algorithm requires for all input values.
3. Big-Omega notation defines the best-case temporal complexity of an algorithm.

The graph below depicts the values of f(n) and C g(n) for input (n) values on the X-Axis and time required on the Y-Axis.



After a specific input value n0, C g(n) is always less than f(n), indicating the algorithm's lower bound.

Example
Consider the f(n) and g(n) expressions...
3n + 2 f(n) = n g(n) = n
If we want to describe f(n) as (g(n)), it must meet the following conditions: f(n) >= C g(n) for all values of C > 0 and n0 >= 1 f(n) >= C g(n) 3n + 2 >= C n

For all values of C = 1 and n >= 1, the above condition is always TRUE.
The temporal complexity can be represented using Big - Omega notation as 3n + 2 = (n).

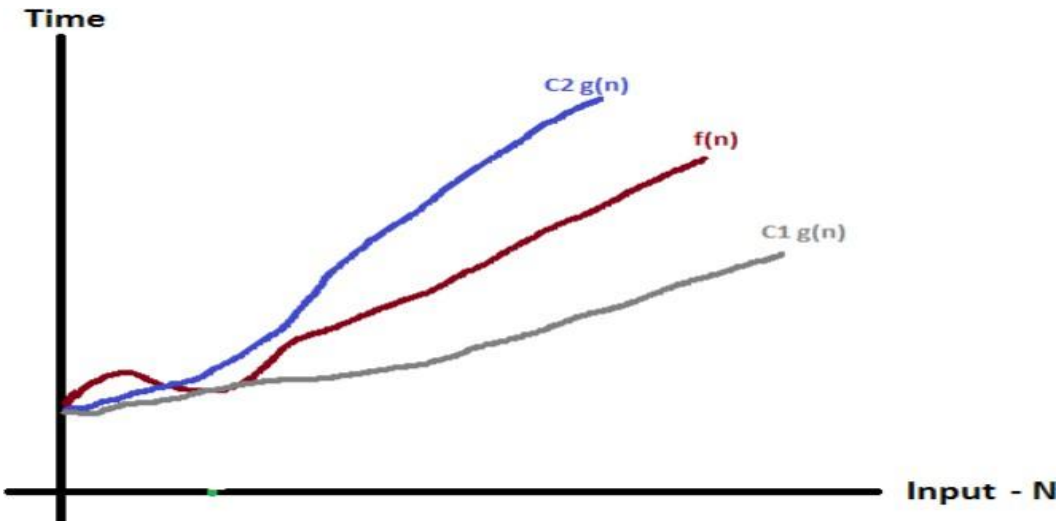
Big - Theta-Notation (Θ):

1.Big - Theta notation is used to define an algorithm's average bound in terms of Time Complexity.

2.Big - Theta notation always reflects an algorithm's average time for all input values.

3.Big - Theta notation describes an algorithm's average time complexity.

The graph below depicts the values of f(n) and C g(n) for input (n) values on the X-Axis and time required on the Y-Axis.



After a certain input value n0, C1 g(n) is always less than f(n), and C2 g(n) is always bigger than f(n), indicating the algorithm's average bound.

Example

Consider the f(n) and g(n) expressions...

3n + 2 f(n) = n g(n) = n

If we want to describe f(n) as (g(n)), it must meet C1 g(n) = f(n) = C2 g(n) for all values of C1 > 0, C2 > 0 and n0 >= 1 C1 g(n) = f(n) = C2 g(n)

For all values of C1 = 1, C2 = 4, and n >= 2, the above condition is always TRUE.

We can represent the time complexity as follows using Big - Theta notation…

3n + 2 = Θ(n).

Searching Techniques

1. Linear Search:

- **Definition:** Linear Search is a simple search algorithm that sequentially checks each element in a list or array until a match is found or the entire list has been traversed.
- **Explanation:** Linear Search is like searching for a specific item in an unordered list by checking each element one by one from the beginning until you either find the target element or reach the end of the list. It's a straightforward but not very efficient way to search for an item in a collection.

- **Algorithm for Linear Search:**

```
#include <stdio.h>

int linearSearch(int arr[], int n, int target)
   { for (int i = 0; i < n; i++) {
      if (arr[i] == target) {
         return i; // Return the index if the target element is found
      }
   }
   return -1; // Return -1 if the target element is not in the array
}
int main() {
   int arr[] = {1, 2, 3, 4, 5};
   int n = sizeof(arr) / sizeof(arr[0]);
   int target = 3;
   int result = linearSearch(arr, n, target);
   if (result != -1) {
      printf("Element found at index: %d\n", result);
   } else {
      printf("Element not found in the array.\n");
   }
   return 0;
}
```

- **Linear Search Time Complexity Analysis:**

  Time Complexity: O(n) - Linear Search checks each element in the array exactly once in the worst case, so its time complexness is linear with respect to the size of the array (n).

2. Binary-Search:

- **Definition:** Binary Search is a fast search algorithm that works on sorted lists or arrays. It repeatedly divides the search interval to half until the target element is found or the search interval is empty.
- **Explanation:** Binary Search is an efficient algorithm for finding a specific item in a sorted list or array. It starts by comparing the target value with the middle element and narrows down the search range by half in each iteration.

Algorithm for Binary Search:

```
#include <stdio.h>
int binarySearch(int arr[], int left, int right, int target)
  { while (left <= right) {
    int mid = left + (right - left) / 2;
    if (arr[mid] == target) {
      return mid; // Return the index if the target element is found
    }
    if (arr[mid] < target)
      { left = mid + 1;
    } else {
      right = mid - 1;
    }
  }
  return -1; // Return -1 if the target element is not in the array
}

int main() {
  int arr[] = {10, 21, 32, 43, 54};
  int n = sizeof(arr) / sizeof(arr[0]);
  int target = 32;
  int result = binarySearch(arr, 0, n - 1, target);
  if (result != -1) {
    printf("Element found at index: %d\n", result);
  } else {
    printf("Element not found in the array.\n");
  }
  return 0;
}
```

- **Binary-Search Time Complexity Analysis:**

  Time Complexness: O(log n) - Binary-Search repeatedly divides the search interval in half, so its time complexity is logarithmic with respect to the size of the array (n).

3. Fibonacci Search:

- **Definition:** Fibonacci Search is an algorithm for searching a sorted array using a divide & conquer approach. It uses Fibonacci series numbers to determine the split points in the array.
- **Explanation:** Fibonacci Search divides the search interval into smaller sub intervals using Fibonacci numbers. It is similar to Binary Search but uses a more complex formula for selecting the split point.

Algorithm for Fibonacci Search:

```
#include   <stdio.h>

int min(int a, int b) {
  return (a < b) ? a : b;}
int fibonacciSearch(int arr[], int n, int target)
  { int fibM_minus_2 = 0;
  int fibM_minus_1 = 1;
  int fibCurrent = fibM_minus_1 + fibM_minus_2;
//Finding Fibonacci number that is greater than or equal to
  n while (fibCurrent < n) {
     fibM_minus_2 = fibM_minus_1;
     fibM_minus_1 = fibCurrent;
     fibCurrent = fibM_minus_1 + fibM_minus_2;
  }
  int offset = -1;
  while (fibCurrent > 1) {
     int i = min(offset + fibM_minus_2, n - 1);
    if (arr[i] < target) {
       fibCurrent = fibM_minus_1;
       fibM_minus_1 = fibM_minus_2;
       fibM_minus_2 = fibCurrent - fibM_minus_1;
       offset = i;
    } else if (arr[i] > target)
       { fibCurrent =
       fibM_minus_2;
       fibM_minus_1 = fibM_minus_1 - fibM_minus_2;
       fibM_minus_2 = fibCurrent - fibM_minus_1;
    } else {
       return i; // Return the index if the target element is found
    }}
```

```
    if (fibM_minus_1 == 1 && arr[n-1] == target)
        { return n-1; // Check the last element
    }
    return -1; // Return -1 if the target element is not in the array
}
int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 3;
    int result = fibonacciSearch(arr, n, target);
    if (result != -1) {
        printf("Element found at index: %d\n", result);
    } else {
        printf("Element not found in the array.\n");
    }
    return 0;
}"
```

Example: let n=11, x=85

i	1	2	3	4	5	6	7	8	9	10	11
arr[i]	10	22	35	40	45	50	80	82	85	90	100

Following is the Fibonacci numbers table for acknowledgment.

	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
Fib.no	0	1	1	2	3	5	8	**13**	21	34	55

Smallest Fibonacci number greater than or equal to 11 is 13.
Therefore fibM = 13, fibMm1 = 8, and fibMm2 = 5

offset=-1

fibM	fibMm1	fibMm2	offset	i	Arr[i]	Inference
13	8	5	-1	4	45	Move down by one, reset offset
8	5	3	4	7	82	Move down by one, reset offset
5	3	2	7	9	90	Move down by two
2	1	1	9	8	85	return i

- **Fibonacci Search Time Complexity Analysis:**

  Time Complexness: O(log n) - Fibonacci-Search also has a logarithmic time complexness similar to Binary-Search.

Each of the algorithms for Linear Search, Binary Search, and Fibonacci Search has its own characteristics and use cases, with Binary Search being the most efficient for sorted data, while Linear Search is simple but less efficient. Linear-Search has a linear time complexness, while Binary-Search and Fibonacci-Search both have logarithmic time complexity. Binary Search is generally preferred for searching in sorted arrays due to its simplicity and efficiency. Fibonacci Search is less commonly used.

Sorting Techniques:

1. Bubble Sort Algorithm

Bubble Sort is one of the simplest sorting algorithms. It works by repeatedly stepping through the list to be sorted, comparing adjacent elements and swapping them if they are in the wrong order. The process continues until no more swaps are needed, indicating that the list is sorted.

Steps to Implement Bubble Sort:

1. **Start from the first element** of the array.
2. **Compare each pair of adjacent elements**.
    - If the pair is in the wrong order (i.e., the first element is greater than the second), swap them.
3. **Move to the next pair** and repeat the comparison.
4. After each pass through the array, the largest unsorted element "bubbles" up to its correct position.
5. **Repeat the process** for the remaining unsorted elements.
6. Stop when no swaps are needed during a new pass through the list.

Pseudocode:

plaintext
CopyEdit

```
BubbleSort(arr)
   n = length of arr
   for i = 0 to n-1
      for j = 0 to n-i-2
         if arr[j] > arr[j+1]
            swap arr[j] and arr[j+1]
```

Example:

Let's sort the array: [5, 3, 8, 4, 2]

1. **First Pass:**
    - Compare 5 and 3: swap → [3, 5, 8, 4, 2]
    - Compare 5 and 8: no swap → [3, 5, 8, 4, 2]
    - Compare 8 and 4: swap → [3, 5, 4, 8, 2]
    - Compare 8 and 2: swap → [3, 5, 4, 2, 8]
    - After the first pass, the largest element 8 is at the end.
2. **Second Pass:**
    - Compare 3 and 5: no swap → [3, 5, 4, 2, 8]
    - Compare 5 and 4: swap → [3, 4, 5, 2, 8]
    - Compare 5 and 2: swap → [3, 4, 2, 5, 8]
    - After the second pass, the second-largest element 5 is now at its correct position.
3. **Third Pass:**
    - Compare 3 and 4: no swap → [3, 4, 2, 5, 8]
    - Compare 4 and 2: swap → [3, 2, 4, 5, 8]
    - After the third pass, 4 is in its correct position.
4. **Fourth Pass:**
    - Compare 3 and 2: swap → [2, 3, 4, 5, 8]

- After the fourth pass, the list is completely sorted.

Sorted Array: [2, 3, 4, 5, 8]

Time Complexity:

- **Best case:** O(n)O(n)O(n) when the list is already sorted.
- **Average case:** O(n2)O(n^2)O(n2), because we compare and swap each pair of adjacent elements for each element.
- **Worst case:** O(n2)O(n^2)O(n2), when the array is sorted in reverse order.

Space Complexity:

- **Space Complexity:** O(1)O(1)O(1), as Bubble Sort is an in-place sorting algorithm (it does not require additional space).

Advantages of Bubble Sort:

1. **Simple to understand and implement.**
2. **In-place sorting** (does not require extra space).
3. It is **adaptive** in the best case (if the array is already sorted).

Disadvantages of Bubble Sort:

1. **Inefficient for large datasets** because its average and worst-case time complexity is O(n2)O(n^2)O(n2).
2. It requires many comparisons and swaps, making it slower compared to more efficient algorithms like Quick Sort or Merge Sort.

2. Selection-Sort:

- **Definition:** Selection Sort is a simple comparison-based sorting algorithm. It repeatedly selects the minimum (or maximum) element from the unsorted portion of the array and places it at the beginning (or end) of the sorted portion.
- **Explanation:** Selection Sort divides the array into two parts: the sorted part on the left and the unsorted part on the right. It repeatedly finds the minimum (or maximum) element from the unsorted part and swaps it with the first element of the unsorted part. This process continues until the entire array is sorted.
- **Time Complexity:** O(n^2) in the worst and average cases, where n is the number of elements in the array.

- **Algorithm for Selection Sort:**

```
#include <stdio.h>
void selectionSort(int arr[], int n)
   { int i, j, minIndex, temp;
   for (i = 0; i < n - 1; i++) {
      minIndex = i; // Assume the current index contains the minimum element
      // Find the index of the minimum element in the unsorted part of the array
      for (j = i + 1; j < n; j++) {
         if (arr[j] < arr[minIndex])
            { minIndex = j;
         }}
      // Swap the minimum element with the element at the current index
      temp = arr[i];   arr[i] = arr[minIndex]; arr[minIndex] = temp;
   }}
int main() {
   int arr[] = {64, 34, 25, 12, 22, 11, 90};
   int n = sizeof(arr) / sizeof(arr[0]);
   printf("Unsorted array: ");
    for (int i = 0; i < n; i++)
     { printf("%d ", arr[i]);
   }
   selectionSort(arr, n);
   printf("\nSorted array: ");
    for (int i = 0; i < n; i++)
     { printf("%d ", arr[i]);
   }return 0;
}
```

In this code:

- Selection-Sort is a function that takes an `integral` array arr and its length n as parameters and sorts the array using the Selection-Sort Algorithm.
- The outer loop iterates through each element of the array from left to right, considering them as the minimum element initially.
- The inner loop finds the index of the minimum element in the unsorted part of the array.
- When the minimum element is found, it is swapped with the element at the current index of the outer loop.
- This process continues until the entire array is sorted.

The main utility shows the usage of the selection-Sort function on an example array.

After sorting, the array will be in ascending order.

3. Insertion Sort:

- **Definition:** Insertion Sort is a simple comparison-based sorting algorithm. It builds the sorted portion of the array one element at a time by repeatedly taking the next unsorted element and inserting it into its correct position within the sorted part.
- **Explanation:** Insertion Sort starts with a single element (the first element) considered as the sorted portion. It then iterates through the unsorted part, taking one element at a time and inserting it into the correct position within the sorted portion. This process continues until the entire array is sorted.
- **Time Complexity:** O(n^2) in the worst and average cases, making it suitable for small datasets or nearly sorted datasets.

Example:

Original	34	8	64	51	32	21	Positions Moved
After p=1	8	34	64	51	32	21	1
After p=2	8	34	64	51	32	21	0
After p=3	8	34	51	64	32	21	1
After p=4	8	32	34	51	64	21	3
After p=5	8	21	32	34	51	64	4

The Insertion Sort algorithm is a simple and intuitive sorting algorithm that is particularly efficient for small data sets. It works by building a sorted array one element at a time, by repeatedly taking the next element from the unsorted part of the array and inserting it into its correct position in the sorted part. This process continues until the entire array is sorted.

When you run this code, it will output the original array followed by the sorted array.

- **Algorithm for Insertion Sort:**

```
"#include <stdio.h>

void insertionSort(int arr[], int n)
   { int i, temp, j;
   for (i = 1; i < n; i++)
      { temp = arr[i];
      for(j=i; j > 0 && arr[j-1] > temp; j--)
         { arr[j] = arr[j-1];
       }
      arr[j] = temp;
   }
}

int main() {
   int arr[] = {34, 8, 64, 51, 32, 21};
   int n = sizeof(arr) / sizeof(arr[0]);

   insertionSort(arr, n);

   printf("Sorted array: \n");
    for (int i = 0; i < n; i++)
     { printf("%d ", arr[i]);
   }
   printf("\n");
   return 0;
}"
```

4. Bubble-Sort:

- **Definition:** Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly compares adjacent elements in the array and swaps them if they are in the wrong order, gradually pushing the largest elements to the end of the array.
- **Explanation:** Bubble Sort repeatedly traverses the array, comparing adjacent elements, and swapping them if they are out of order. This process continues until no more swaps are needed, indicating that the array is sorted.
- **Time Complexity:** O(n^2) in the worst and average cases, making it suitable for small datasets.

- **Algorithm for Bubble Sort:**

```
#include <stdio.h>
void bubbleSort(int arr[], int n)
  { int temp;
  int swapped;
  for (int i = 0; i < n - 1; i++) {
    swapped = 0; // Flag to optimize when the array is already sorted
    for (int j = 0; j < n - i - 1; j++) {
      // Compare adjacent elements and swap them if they are in the wrong order
      if (arr[j] > arr[j + 1]) {
        temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
        swapped = 1; // Set the flag to indicate a swap occurred
      }}
    // If no two elements were swapped in the inner loop, the array is already sorted
    if (swapped == 0) {
      break;
    }}}
    int main() {
  int arr[] = {64, 34, 25, 12, 22, 11, 90};
  int n = sizeof(arr) / sizeof(arr[0]);
  printf("Original Array: ");
   for (int i = 0; i < n; i++)
    { printf("%d ", arr[i]);
  }
  bubbleSort(arr, n);
  printf("\nSorted Array: ");
  for (int i = 0; i < n; i++) {
    printf("%d ", arr[i]);
  }return 0;}
```

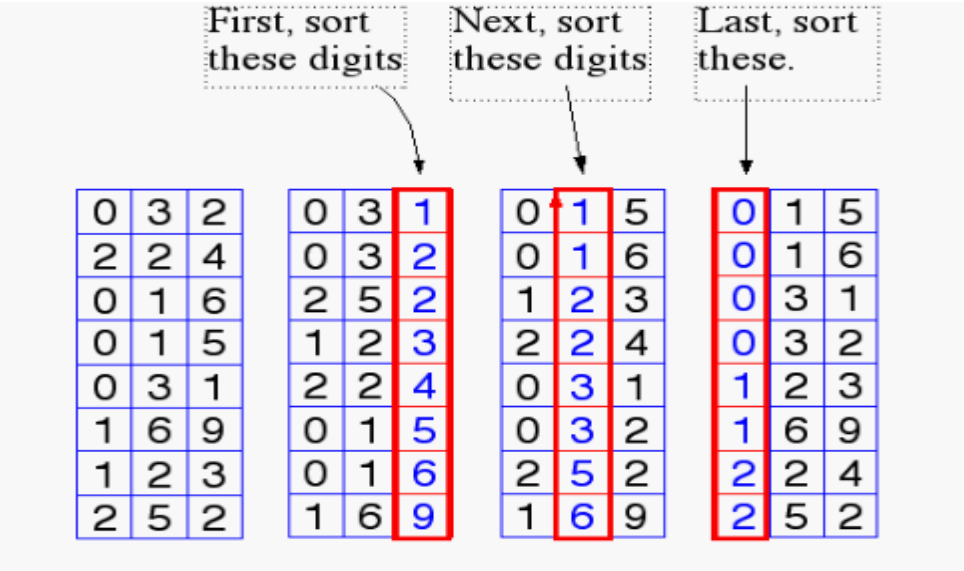
This code defines the bubble-Sort function, which takes an array of integers arr and its length , and sorts the array in ascending order using the Bubble Sort algorithm. The main function demonstrates how to use this sorting function on an example array.

When you run this code, it will output the original array followed by the sorted array.

5. Radix Sort or Bin Sort or Bucket Sort:

- **Definition:** Radix-Sort is a non comparable sorting algorithm that works by forwarding individual digits of the numbers in the array, from the (Minimum)Least significant digit (LSD) to the (Maximum)Most significant digit (MSD) or vice versa.
- **Explanation:** Radix Sort processes the array by considering the digits of the numbers. It starts by sorting based on the minimum significant digit and proceeds to the maximum significant digit.
- **Time Complexity:** O(n * k) in the worst, average, and best cases, where N is the number of components & K is the number of digits in the maximal number. It is efficient when k is small compared to n.

Radix Sort Example:



Radix-Sort Algorithm

The radix sort algorithm makes use of the counting sort algorithm while sorting in every phase. The detailed steps are as follows −

Step 1 − Check whether all the input elements have same number of digits. If not, check for numbers that have maximum number of digits (say k) in the list and add leading zeroes to the ones that do not.

Step 2 − Take the least significant digit of each element.

Step 3 − Sort these digits using counting sort logic and change the order of elements based on the output achieved. For example, if the input elements are decimal numbers, the possible values each digit can take would be 0-9, so index the digits based on these values.

Step 4 − Repeat the Step 2 for the next least significant digits until all the digits in the elements are sorted.

Step 5 − The final list of elements achieved after kth loop is the sorted output.

```
#include <stdio.h>
#include <stdlib.h>

// Function to get the maximum value in the array
int getMax(int arr[], int n) {
   int max = arr[0];
   for (int i = 1; i < n; i++) {
      if (arr[i] > max)
         max = arr[i];
   }
   return max;
}

// Function to perform counting sort based on the digit represented by exp
void countingSort(int arr[], int n, int exp) {
```

```
    int output[n]; // output array to store sorted numbers
    int count[10] = {0}; // count array to store the frequency of digits (0-9)

    // Count the occurrences of each digit
    for (int i = 0; i < n; i++) {
        count[(arr[i] / exp) % 10]++;
    }

    // Modify the count array such that count[i] contains the actual position of this digit in output[]
    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    // Build the output array by placing the elements in the correct position
    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // Copy the sorted output array to arr[], so that arr[] now contains the sorted numbers
    for (int i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}

// Main function to implement Radix Sort
void radixSort(int arr[], int n) {
    // Find the maximum number to determine the number of digits
    int max = getMax(arr, n);

    // Apply counting sort for every digit. The exp is 10^i where i is the current digit we are sorting by.
    for (int exp = 1; max / exp > 0; exp *= 10) {
        countingSort(arr, n, exp);
    }
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Main function
voids main() {
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original Array: \n");
    printArray(arr, n);

    radixSort(arr, n);

    printf("Sorted Array: \n");
    printArray(arr, n);

    }
```