

I. INTRODUCTION TO DATA STRUCTURES AND ALGORITHMS

1. What is a data structure

A data structure is a way of organizing and storing data in a computer so that it can be accessed and modified efficiently. Data structures are used to represent real-world entities such as lists, stacks, queues, trees, graphs, and more. They provide a way to store and manipulate data in an organized manner, allowing for efficient retrieval and modification of the data.

2. What is an algorithm

An algorithm is a step-by-step procedure for solving a problem. Algorithms can be written in a variety of programming languages and can be used to solve a variety of problems.

An algorithm is a set of instructions or steps followed to solve a problem or complete a task. Algorithms are used in computer programming to perform specific tasks, such as sorting data or calculating a result. They are also used in everyday life, such as when following a recipe or playing a game.

2.1. Characteristics of an Algorithm

An algorithm is a set of instructions or steps that are used to solve a problem or accomplish a task. Characteristics of an algorithm include:

- **Input:** An algorithm must have at least one input.
- **Output:** An algorithm must have at least one output.
- **Definiteness:** An algorithm must be precise and unambiguous.
- **Finiteness:** An algorithm must terminate after a finite number of steps.
- **Effectiveness:** An algorithm must be able to produce a result in a finite amount of time.

2.2. Representation of an Algorithm

Algorithms can be represented using:

- Flowcharts
- Pseudocode

a. Flowchart

A flowchart is a graphical representation of an algorithm.



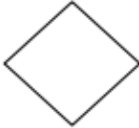



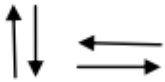
A flowchart is a visual representation of the steps involved in a process. It can be used to help people understand the process and to ensure that the process is followed correctly.

A flowchart is a diagram that shows the steps in a process. It is used to visually represent a sequence of activities or events in a process. Flowcharts are often used to document and analyze processes, troubleshoot problems, and plan projects. They can also be used to show how different parts of a system interact with each other.

Symbols are used to represent different activities in a flowchart.

The symbols include:

- Start/End: Oval
- Process: Rectangle
- Decision: Diamond
- Input/Output: Parallelogram
- Connector: Arrow

Symbol	Name	Function
	Process	Indicates any type of internal operation inside the Processor or Memory
	input/output	Used for any Input / Output (I/O) operation. Indicates that the computer is to obtain data or output results
	Decision	Used to ask a question that can be answered in a binary format (Yes/No, True/False)
	Connector	Allows the flowchart to be drawn without intersecting lines or without a reverse flow.
	Predefined Process	Used to invoke a subroutine or an Interrupt program.
	Terminal	Indicates the starting or ending of the program, process, or interrupt program
	Flow Lines	Shows direction of flow.

b. Pseudocode

Pseudocode is a plain language description of the steps in an algorithm or another system. Pseudocode often uses structural conventions of a normal programming language but is intended for human reading rather than machine reading.

3. Types of Data Structure

- **Arrays:** An array is a data structure that stores a collection of elements of the same type in a contiguous block of memory.

- **Linked Lists:** A linked list is a data structure that consists of a sequence of nodes, each containing data and a pointer to the next node in the list.
- **Stacks:** A stack is a data structure that stores data in a Last-In-First-Out (LIFO) manner.
- **Queues:** A queue is a data structure that stores data in a First-In-First-Out (FIFO) manner.
- **Trees:** A tree is a data structure that consists of nodes connected by edges

II. BASIC DATA STRUCTURES

a. Array

An array is a data structure that stores a collection of items. Each item in an array is identified by an index, which is a number that indicates the position of the item in the array. Arrays can store items of any data type, including numbers, strings, objects, and even other arrays. Arrays are commonly used to store collections of data that can be easily accessed and manipulated.

It is a set of data items of the same type. Items in an array are called array elements. Each array element has a unique identifying index, which is used for accessing a particular element in the array. Array index begins from zero. The array allows us to put many variables in one unique variable.

Consider the array Price which has values (100, 150, 400, 490). An array element may be accessed separately by stating its position in the array, Price(3) will refer to 490. The position of an element in an array is an Indices. A one-dimensional array is called a vector while a 2-dimensional array is called a matrix.

b. Stacks

This is a data structure in which elements are removed in the reverse order from which they were entered. They are often referred to as LIFO (Last In First Out) data

structure. Stacks are easier to implement than queues because you only work at one end. There two main operations applicable to a stack are push and pop. An operation that adds an element to a stack is called push. The operation that removes an element from a stack is know as pop

c. Queue

This is a data structure in which elements are removed in the same order they were entered. Queues are often referred to as FIFO (First In First Out) data structures.

d. Linked List

Linked lists are a type of data structure that stores elements in a linear fashion. Each element is connected to the next element via a pointer. Linked lists are used to store dynamic data structures, such as stacks and queues.

A linked list is a data structure that allows nodes to be accessed in any order. The first node in the list is called the head, and the last node is called the tail. Nodes in a linked list are linked by a pointer, which is a variable that points to the next node in the list.

A linked list is a data structure that consists of a sequence of nodes. Each node contains a value and a pointer to the next node in the list. The first node in the list is known as the head, and the last node is known as the tail. Linked lists are useful for storing data that needs to be accessed in a specific order, such as a list of items or a queue. They are also used to implement other data structures, such as stacks and tree.

Types of Linked Lists

➤ Singly Linked List:

A singly linked list is a type of linked list in which each node contains a pointer that points to the next node in the list.

➤ **Doubly Linked List:**

A doubly linked list is a type of linked list in which each node contains two pointers, one pointing to the previous node and one pointing to the next node in the list.

➤ **Circular Linked List:**

A circular linked list is a type of linked list in which the last node points back to the first node, thus forming a loop.

The following Operation can be performed on a linked list

- **Insertion:** Adding a new node to the linked list.
- **Deletion:** Removing an existing node from the linked list.
- **Traversal:** Accessing each node of the linked list in a sequential manner.
- **Searching:** Finding a particular node in the linked list.
- **Reversal:** Reversing the order of the nodes in the linked list.

III. ADVANCED DATA STRUCTURES

a. **Tree:**

Trees are a type of data structure that stores elements in a hierarchical fashion. Each element is connected to one or more other elements via a parent-child relationship

A tree data structure is a data structure that organizes data into a hierarchy. The root of the tree is at the top, and the leaves are the lower levels. Each node in the tree contains information about one or more children nodes.

b. **Graph**

A graph data structure is a data structure that allows nodes to be interconnected in a graph. Graphs are commonly used in computer science and data mining to model relationships between objects.

c. Heaps

A heap data structure is a data structure that uses a heap to store its elements. Heaps are efficient data structures for storing large amounts of data, because they allow the elements to be stored in a contiguous area.

IV. ALGORITHM DESIGN TECHNIQUES

a. Greedy Algorithms:

Greedy algorithms are a type of algorithm that make decisions by selecting the best option available at each step. They are used to solve optimization problems, such as finding the shortest path between two points or the most efficient way to allocate resources.

b. Divide and Conquer:

Divide and conquer algorithms break a problem into smaller sub-problems, solve each sub-problem, and then combine the solutions to the sub-problems to get a solution to the original problem. This technique is often used for sorting algorithms, such as quicksort and merge sort.

c. Dynamic Programming:

Dynamic programming is a technique for solving complex problems by breaking them down into smaller sub-problems and then combining the results.

V. Algorithm Analysis

a. Time Complexity

Time complexity is a measure of how long it takes to solve a problem.

The time complexity is the number of operations an algorithm performs to complete its task (considering that each operation takes the same amount of time). The

algorithm that performs the task in the smallest number of operations is considered the most efficient one in terms of time complexity.

There are different types of time complexity.

- **Constant time $O(1)$**

An algorithm is said to have constant time with order $O(1)$ when it is not dependent on the input size n . Irrespective of the input size n , the runtime will always be the same

- **Linear time $O(n)$**

An algorithm is said to have a linear time complexity when the running time increases linearly with the length of the input. When the function involves checking all the values in input data, such function has Time complexity with this order $O(n)$.

- **Logarithmic time $O(\log n)$**

An algorithm is said to have a logarithmic time complexity when it reduces the size of the input data in each step. This indicates that the number of operations is not the same as the input size. The number of operations gets reduced as the input size increases. Algorithms with Logarithmic time complexity are found in binary trees or binary search functions

- **Quadratic time $O(n^2)$**

An algorithm is said to have a non-linear time complexity where the running time increases non-linearly (n^2) with the length of the input. Generally, nested loops come under this time complexity order where one loop takes $O(n)$ and if the function involves a loop within a loop, then it goes for $O(n)*O(n) = O(n^2)$ order.

Similarly, if there are 'm' loops defined in the function, then the order is given by $O(n^m)$, which are called as polynomial time complexity functions.

- The **worst-case** complexity of the algorithm is the function defined by the maximum number of steps taken on any instance of size n .
- The **best-case** complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size n .
- Finally, the **average-case** complexity of the algorithm is the function defined by the average number of steps taken on any instance of size n .

b. Space Complexity

The space complexity of an algorithm is the size of the space required to store the algorithm's input and output.

The space Complexity of an algorithm is the total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

Auxiliary Space is the extra space or temporary space used by an algorithm.

Space complexity is a parallel concept to time complexity. If we need to create an array of size n , this will require $O(n)$ space. If we create a two-dimensional array of size $n \times n$, this will require $O(n^2)$ space.

In recursive calls stack space also counts.

VI. SORTING AND SEARCHING ALGORITHMS

1. Searching algorithm

A search algorithm is an algorithm for finding an item with specified properties among a collection of items.

The main purpose of searching algorithms is to check an element or retrieve it from any data structure. These searching algorithms are classified into two different parts generally based on the type of searching.

- **Sequential search:** List or array is traversed sequentially and every element is checked. e.g: Linear Search
- **Interval search:** Designed for sorted data structures and more efficient than sequential search algorithms as these repeatedly target the center of the data structure and divide the search space in half. eg: Binary Search

Example of the Search algorithm

a. Linear Search

Here, a sequential search is made throughout every element in the data structure one by one. If the match is found, it is returned otherwise searching process continues until the end of the data structure. It has a time complexity of $O(n)$

Consider we want to find the value x in array A .

Linear Search (Array A , Value x)

Step 1: Set i to 0

Step 2: if $i \geq n$ then go to step 7

Step 3: if $A[i] = x$ then go to step 6

Step 4: Set i to $i + 1$

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

procedure linear_search (list, value)

for each item in the list

if match item == value

return the item's location

b. Binary Search

The data collection should be in the **sorted form** in order to work this algorithm correctly. Binary search looks for a particular item by comparing “**the middlemost**” (element in the middle) item of the collection. If a match occurs, then the index of the item is returned. If the match does not occur, it checks whether the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. Until the subarray size reduces to zero this process continues on the sub-array as well. It has a time complexity of $O(\log n)$

Binary Search

Search 23

0	1	2	3	4	5	6	7	8	9
2	5	8	12	16	23	38	56	72	91

23 > 16
take 2nd half

L=0	1	2	3	M=4	5	6	7	8	H=9
2	5	8	12	16	23	38	56	72	91

23 > 56
take 1st half

0	1	2	3	4	L=5	6	M=7	8	H=9
2	5	8	12	16	23	38	56	72	91

Found 23,
Return 5

0	1	2	3	4	L=5, M=5	H=6	7	8	9
2	5	8	12	16	23	38	56	72	91

Consider we want to find the value x in sorted array A.

Binary Search (Array A, Value x)

```
Step 1: Set R=0 and L=n-1
Step 2: if L > R then go to step 7
Step 3: Set m (the position of the middle element) to the floor of
(L+R)/2
Step 4: If A[m] < x, set L to m+1 and go to Step 2
Step 5: If A[m] > x, set R to m-1 and go to Step 2
Step 6: Now A[m]= x, return m, and go to step 8
Step 7: Print element not found
Step 8: Exit
```

```
Procedure binary_search
  A ← sorted array
  n ← size of array
  x ← value to be searched

  Set lowerBound = 0
  Set upperBound = n-1

  while x not found
    if upperBound < lowerBound
      EXIT: x does not exists.

    set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

    if A[midPoint] < x
      set lowerBound = midPoint + 1

    if A[midPoint] > x
      set upperBound = midPoint - 1

    if A[midPoint] = x
      EXIT: x found at location midPoint
  end while

end procedure
```

2. Sorting Algorithm

It is an algorithm used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

Types of Sorting Algorithm

a. Bubble Sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is a comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

Performance Analysis:

Time Complexity :

- **Worst-case:** $O(n^2)$ - Since we loop through n elements n times, n being the length of the array, the time complexity of Bubble sort becomes $O(n^2)$.
- **Best-case:** $O(n^2)$ - Even if the array is sorted, the algorithm checks each adjacent pair, and hence the best-case time complexity will be the same as the worst-case.

Space Complexity: $O(1)$.

Since we are not making use of any extra data structure apart from the input array, the space complexity will be $O(1)$.

b. Selection Sort

The algorithm works by finding the smallest element, swapping it with the value in the first position, and repeating these steps for the remainder of the list. It has a complexity of $O(n^2)$, making it inefficient on large lists.

The algorithm maintains two subarrays in a given array.

- ✓ The subarray which is already sorted.
- ✓ Remaining subarray which is unsorted.

Time Complexity:

1. **Worst-case:** $O(n^2)$ - Since for each element in the array, we traverse through the remaining array to find the minimum, the time complexity will become $O(n^2)$
2. **Best-case:** $O(n^2)$ - Even if the array is already sorted, our algorithm looks for the minimum in the rest of the array, and hence best-case time complexity is the same as the worst-case.

Space Complexity: $O(1)$

Since like previous algorithms, we are not making use of any extra data structure apart from the input array, the space complexity will be $O(1)$.

c. Insertion Sort

It works by taking elements from the list one by one and inserting them in their position into a new sorted list. The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array).

Performance Analysis:**Time Complexity:**

1. **Worst-case:** $O(n^2)$ In the worst case, our array is sorted in descending order. So for each element, we have to keep traversing and swapping elements to the left.
2. **Best-case :** $O(n)$ - In the best case, our array is already sorted. So for each element, we compare our current element to the element at the left only once. Since the order is correct, we don't swap and move on to the next element. Hence the time complexity will be $O(n)$.

Space Complexity: $O(1)$

Since we are not making use of any extra data structure apart from input array, the space complexity will be $O(1)$.

d. Merge Sort

Merge sort is a sorting technique based on the divide and conquer technique. With worst-case time complexity being $O(n \log n)$, it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Performance Analysis:

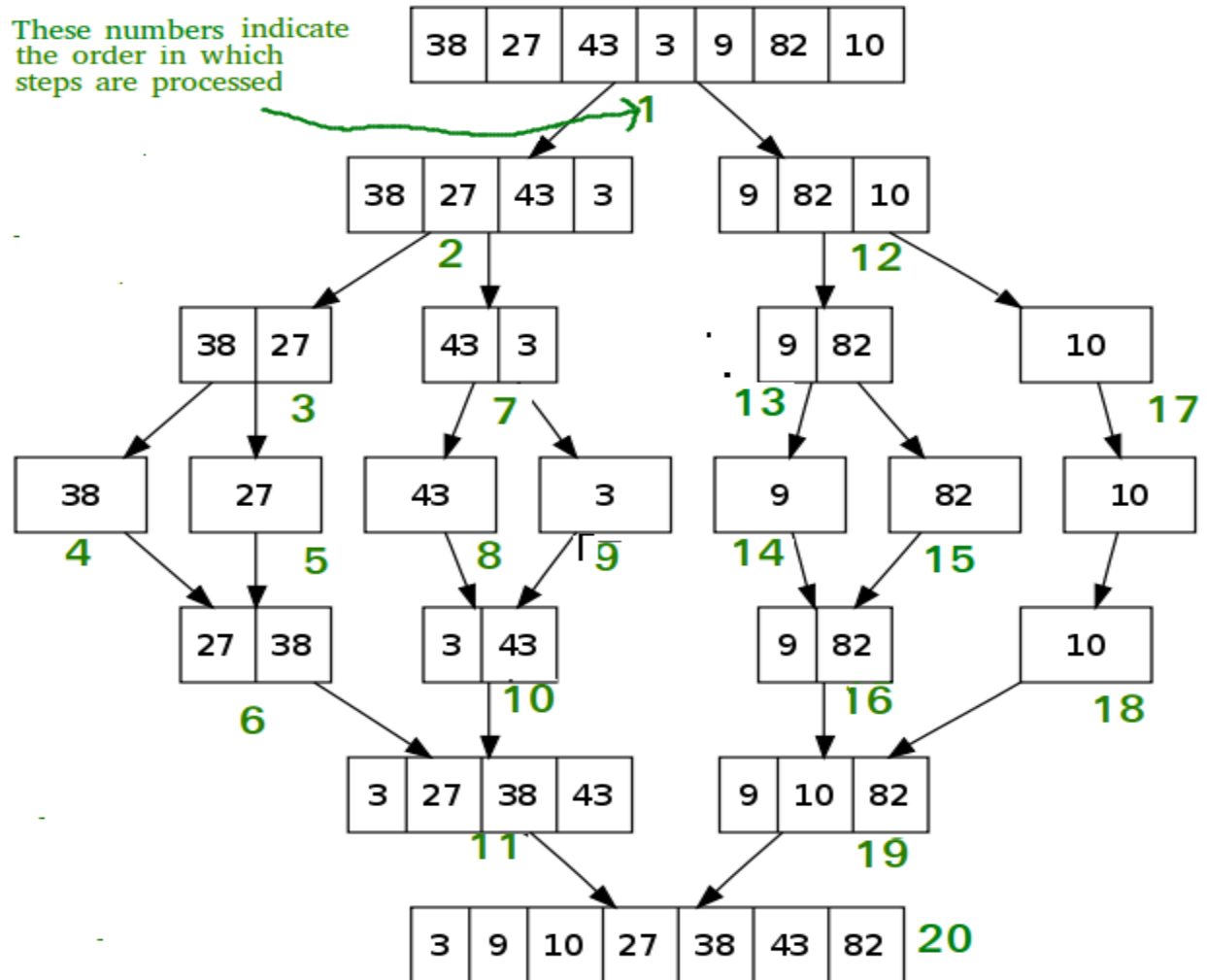
Time Complexity:

- **Best-case:** $O(n \log n)$ - First, we are dividing the array into two subarrays recursively which will cost a time complexity of $O(\log n)$. For each function call, we are calling the partition function which costs $O(n)$ time complexity. Hence the total time complexity is $O(n \log n)$.
- **Worst-case:** $O(n \log n)$ - The worst-case time complexity is the same as the best-case.

Space Complexity: $O(n)$

Since we are recursively calling the MergeSort function, an internal stack is used to store these function calls. There will be at most N calls in the stack and hence the space complexity will be $O(n)$.

These numbers indicate the order in which steps are processed



e. Quick Sort

Quick Sort is also known as Partition Sort. This sorting algorithm is faster than the previous algorithms because this algorithm uses the concept of Divide and Conquer.

First, we decide a pivot element. We then find the correct index for this pivot position and then divide the array into two subarrays. One subarray will contain elements that are lesser than our pivot and other subarrays will contain elements that are greater than our pivot. We then recursively call these two subarrays and the process goes on until we can't further divide the array.

Performance Analysis:

Time Complexity:

- **Best-case:** $O(N \log N)$ - First, we are dividing the array into two subarrays recursively which will cost a time complexity of $O(\log n)$. For each function call, we are calling the partition function which costs $O(n)$ time complexity. Hence the total time complexity is $O(n \log n)$.
- **Worst-case:** $O(n^2)$ - When the array is sorted in descending order or all the elements are the same in the array, the time complexity jumps to $O(n^2)$ since the subarrays are highly unbalanced.

Space Complexity: $O(n)$

Since we are recursively calling the quicksort function, an internal stack is used to store these function calls. There will be at most n calls in the stack and hence the space complexity will be $O(n)$