DATA STRUCTURES AND ALGORITHMS

(Introduction to Algorithms)

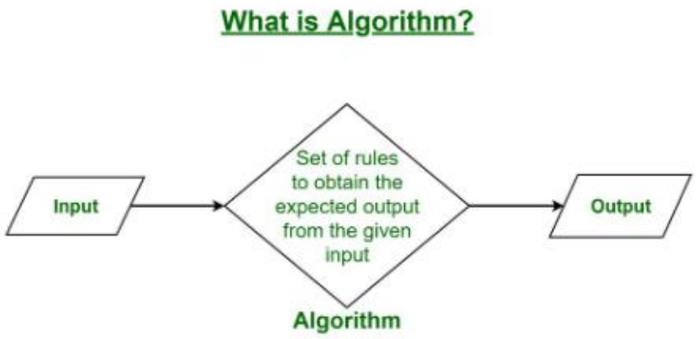
Algorithm

- An algorithm is simply a **set of steps used to complete a specific task**. They're the building blocks for programming, and they allow things like computers, smartphones, and websites to function and make decisions.
- *The word Algorithm means "A set of finite rules or instructions to be followed in calculations or other problem-solving operations"*
- *" A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operations".*

How do Algorithms Work?

Algorithms typically follow a logical structure:

- **Input:** The algorithm receives input data.
- **Processing:** The algorithm performs a series of operations on the input data.
- **Output:** The algorithm produces the desired output.



Use of the Algorithms:

Algorithms play a crucial role in various fields and have many applications. Some of the key areas where algorithms are used include:

1. **Computer Science:** Algorithms form the basis of computer programming and are used to solve problems ranging from simple sorting and searching to complex tasks such as artificial intelligence and machine learning.
2. **Mathematics:** Algorithms are used to solve mathematical problems, such as finding the optimal solution to a system of linear equations or finding the shortest path in a graph.
3. **Operations Research**: Algorithms are used to optimize and make decisions in fields such as transportation, logistics, and resource allocation.
4. **Artificial Intelligence:** Algorithms are the foundation of artificial intelligence and machine learning, and are used to develop intelligent systems that can perform tasks such as image recognition, natural language processing, and decision-making.
5. **Data Science:** Algorithms are used to analyze, process, and extract insights from large amounts of data in fields such as marketing, finance, and healthcare.

Characteristics of an Algorithm:

- **Clear and Unambiguous:** The algorithm should be unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-defined Inputs:** If an algorithm says to take inputs, it should be well-defined inputs. It may or may not take input.
- **Well-defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well. It should produce at least 1 output.
- **Finiteness:** The algorithm must be finite, i.e. it should terminate after a finite time.
- **Feasible:** The algorithm must be simple, generic, and practical, such that it can be executed using reasonable constraints and resources.
- **Language Independent:** Algorithm must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

Properties of Algorithm:

- It should terminate after a finite time.
- It should produce at least one output.
- It should take zero or more input.
- It should be deterministic means giving the same output for the same input case.
- Every step in the algorithm must be effective i.e. every step should do some work.

What is the Need for Algorithms?

Algorithms are essential for solving complex computational problems efficiently and effectively. They provide a systematic approach to:

- **Solving problems:** Algorithms break down problems into smaller, manageable steps.
- **Optimizing solutions:** Algorithms find the best or near-optimal solutions to problems.
- **Automating tasks:** Algorithms can automate repetitive or complex tasks, saving time and effort.

How to Write an Algorithm?

To write an algorithm, follow these steps:

- **Define the problem:** Clearly state the problem to be solved.
- **Design the algorithm:** Choose an appropriate algorithm design paradigm and develop a step-by-step procedure.
- **Implement the algorithm:** Translate the algorithm into a programming language.
- **Test and debug:** Execute the algorithm with various inputs to ensure its correctness and efficiency.
- **Analyze the algorithm:** Determine its time and space complexity and compare it to alternative algorithms.

Advantages of Algorithms:

- It is easy to understand.
- An algorithm is a step-wise representation of a solution to a given problem.
- In an Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Disadvantages of Algorithms:

- Writing an algorithm takes a long time so it is time-consuming.
- Understanding complex logic through algorithms can be very difficult.
- Branching and Looping statements are difficult to show in Algorithms**(imp)**.

How to express an Algorithm?

1. **Natural Language:-** Here we express the Algorithm in the natural English language. It is too hard to understand the algorithm from it.
2. **Flowchart:-** Here we express the Algorithm by making a graphical/pictorial representation of it. It is easier to understand than Natural Language.
3. **Pseudo Code:-** Here we express the Algorithm in the form of annotations and informative text written in plain English which is very much similar to the real code but as it has no syntax like any of the programming languages, it can't be compiled or interpreted by the computer. It is the best way to express an algorithm because it can be understood by even a layman with some school-level knowledge.

Basic Algorithm Approaches:

a. Greedy Algorithm

A Greedy algorithm is an approach to solving a problem that selects the most appropriate option based on the current situation. This algorithm ignores the fact that the current best result may not bring about the overall optimal result. Even if the initial decision was incorrect, the algorithm never reverses it.

Greedy algorithms are a class of algorithms that make locally optimal choices at each step with the hope of finding a global optimum solution. In these algorithms, decisions are made based on the information available at the current moment without considering the consequences of these decisions in the future. The key idea is to select the best possible choice at each step, leading to a solution that may not always be the most optimal but is often good enough for many problems. It operates on the principle of "taking the best option now" without considering the long-term consequences.

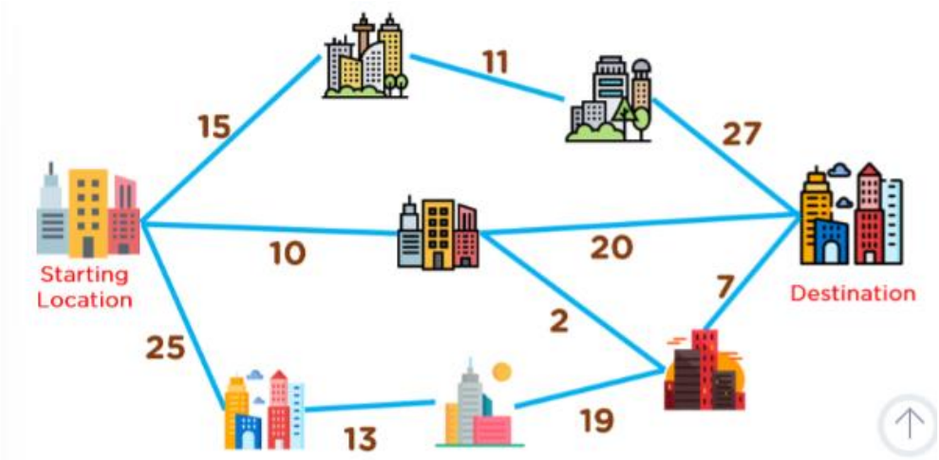
Steps for Creating a Greedy Algorithm

By following the steps given below, you will be able to formulate a greedy solution for the given problem statement:

- Step 1: In a given problem, find the best substructure or subproblem.
- Step 2: Determine what the solution will include (e.g., largest sum, shortest path).
- Step 3: Create an iterative process for going over all subproblems and creating an optimum solution.

Example of Greedy Algorithm

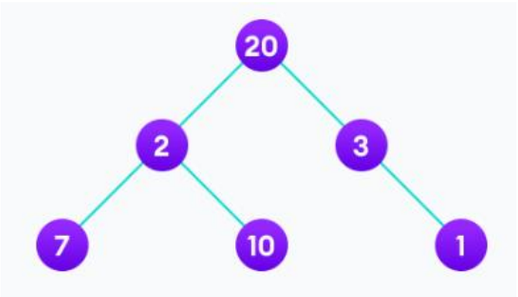
Problem Statement: Find the best route to reach the destination city from the given starting point using a greedy method.



Greedy Solution: In order to tackle this problem, we need to maintain a graph structure. And for that graph structure, we'll have to create a tree structure, which will serve as the answer to this problem. The steps to generate this solution are given below:

- Start from the source vertex.
- Pick one vertex at a time with a minimum edge weight (distance) from the source vertex.
- Add the selected vertex to a tree structure if the connecting edge does not form a cycle.
- Keep adding adjacent fringe vertices to the tree until you reach the destination vertex.

For example, suppose we want to find the longest path in the graph below from root to leaf. Let's use the greedy algorithm here.



Greedy Approach

1. Let's start with the root node **20**. The weight of the right child is **3** and the weight of the left child is **2**.
2. Our problem is to find the largest path. And, the optimal solution at the moment is **3**. So, the greedy algorithm will choose **3**.
3. Finally the weight of an only child of **3** is **1**. This gives us our final result 20 + 3 + 1 = 24.

However, it is not the optimal solution. There is another path that carries more weight (20 + 2 + 10 = 32) as shown in the image below.

Example - Greedy Approach

Problem: You have to make a change of an amount using the smallest possible number of coins.
Amount: $18

Available coins are
$5 coin
$2 coin
$1 coin
There is no limit to the number of each coin you can use.

Solution:

1. Create an empty solution-set = { }. Available coins are {5, 2, 1}.
2. We are supposed to find the sum = 18. Let's start with sum = 0.
3. Always select the coin with the largest value (i.e. 5) until the sum > 18. (When we select the largest value at each step, we hope to reach the destination faster. This concept is called **greedy choice property**.)
4. In the first iteration, solution-set = {5} and sum = 5.
5. In the second iteration, solution-set = {5, 5} and sum = 10.
6. In the third iteration, solution-set = {5, 5, 5} and sum = 15.
7. In the fourth iteration, solution-set = {5, 5, 5, 2} and sum = 17. (We cannot select 5 here because if we do so, sum = 20 which is greater than 18. So, we select the 2nd largest item which is 2.)
8. Similarly, in the fifth iteration, select 1. Now sum = 18 and solution-set = {5, 5, 5, 2, 1}.

One of the most famous examples of the greedy method is the knapsack problem. In this problem, we are given a set of items, each with a weight and a value. We want to find the subset of items that maximizes the value while minimizing the weight. The greedy method would simply take the item with the highest value at each step. However, this might not be the best solution. For example, consider the following set of items:

Item 1: Weight = 2, Value = 6
Item 2: Weight = 2, Value = 3
Item 3: Weight = 4, Value = 5

The greedy method would take Item 1 and Item 3, for a total value of 11. However, the optimal solution would be to take Item 2 and Item 3, for a total value of 8. Thus, the greedy method does not always find the best solution.

There are many other examples of the greedy method. The most famous one is probably the Huffman coding algorithm, which is used to compress data. In this algorithm, we are given a set of symbols, each with a weight. We want to find the subset of symbols that minimizes the average length of the code. The greedy method would simply take the symbol with the lowest weight at each step. However, this might not be the best solution. For example, consider the following set of symbols:

Symbol 1: Weight = 2, Code = 00
Symbol 2: Weight = 3, Code = 010
Symbol 3: Weight = 4, Code =011

The greedy method would take Symbol 1 and Symbol 3, for a total weight of 6. However, the optimal solution would be to take Symbol 2 and Symbol 3, for a total weight of 7. Thus, the greedy method does not always find the best solution.
The greedy method is a simple and straightforward way to solve optimization problems. However, it is not always guaranteed to find the best solution and can be quite slow. When using the greedy method, it is important to keep these disadvantages in mind.

Limitations of Greedy Algorithm

Factors listed below are the limitations of a greedy algorithm:

1. The greedy algorithm makes judgments based on the information at each iteration without considering the broader problem; hence it does not produce the best answer for every problem.
2. The problematic part for a greedy algorithm is analyzing its accuracy. Even with the proper solution, it is difficult to demonstrate why it is accurate.
3. Optimization problems (Dijkstra's Algorithm) with negative graph edges cannot be solved using a greedy algorithm.

Disadvantages of Using Greedy Algorithms

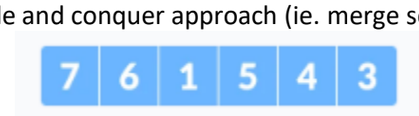
The main disadvantage of using a greedy algorithm is that it may not find the optimal solution to a problem. In other words, it may not produce the best possible outcome. Additionally, greedy algorithms can be very sensitive to changes in input data — even a small change can cause the algorithm to produce a completely different result. Finally, greedy algorithms can be difficult to implement and understand.

b. Divide and Conquer Algorithm

This algorithm breaks a problem into sub-problems, solves a single sub-problem, and merges the solutions to get the final solution. It consists of the following three steps:

- Divide - Divide the given problem into sub-problems using recursion.
- Solve - Solve the smaller sub-problems recursively. If the subproblem is small enough, then solve it directly.
- Combine - Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

Let us understand this concept with the help of an example.
Here, we will sort an array using the divide and conquer approach (ie. merge sort).



Divide the array into two halves.
Again, divide each subpart recursively into two halves until you get individual elements.
Now, combine the individual elements in a sorted manner. Here, **conquer** and **combine** steps go side by side.

Advantages of Divide and Conquer Algorithm:

- **Solving difficult problems:** Divide and conquer technique is a tool for solving difficult problems conceptually. e.g. Tower of Hanoi puzzle. It requires a way of breaking the problem into sub-problems, and solving all of them as an individual cases and then combining sub- problems to the original problem.
- **Algorithm efficiency:** The divide-and-conquer algorithm often helps in the discovery of efficient algorithms. It is the key to algorithms like Quick Sort and Merge Sort, and fast Fourier transforms.

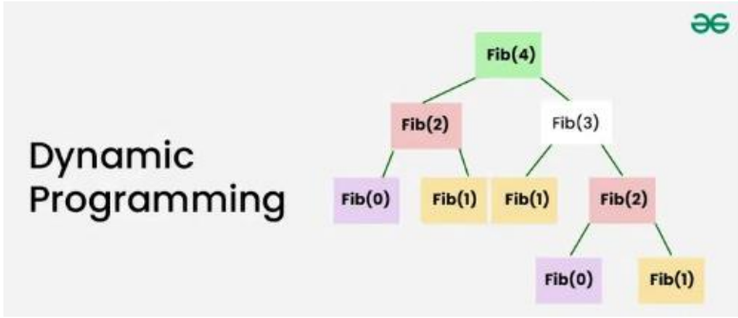
- **Parallelism:** Normally Divide and Conquer algorithms are used in multi-processor machines having shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed on different processors.
- **Memory access:** These algorithms naturally make an efficient use of memory caches. Since the subproblems are small enough to be solved in cache without using the main memory that is slower one. Any algorithm that uses cache efficiently is called cache oblivious.

Disadvantages of Divide and Conquer Algorithm:

- **Overhead:** The process of dividing the problem into subproblems and then combining the solutions can require additional time and resources. This overhead can be significant for problems that are already relatively small or that have a simple solution.
- **Complexity:** Dividing a problem into smaller subproblems can increase the complexity of the overall solution. This is particularly true when the subproblems are interdependent and must be solved in a specific order.
- **Difficulty of implementation:** Some problems are difficult to divide into smaller subproblems or require a complex algorithm to do so. In these cases, it can be challenging to implement a divide and conquer solution.
- **Memory limitations:** When working with large data sets, the memory requirements for storing the intermediate results of the subproblems can become a limiting factor.

c. Dynamic Programming Algorithm

This algorithm uses the concept of using the already found solution to avoid repetitive calculation of the same part of the problem. It divides the problem into smaller overlapping subproblems and solves them.



How Does Dynamic Programming (DP) Work?

- **Identify Subproblems:** Divide the main problem into smaller, independent subproblems.
- **Store Solutions:** Solve each subproblem and store the solution in a table or array.
- **Build Up Solutions:** Use the stored solutions to build up the solution to the main problem.
- **Avoid Redundancy:** By storing solutions, DP ensures that each subproblem is solved only once, reducing computation time.

Fibonacci Series using Dynamic Programming

- **Subproblems:** F(0), F(1), F(2), F(3), …
- **Store Solutions:** Create a table to store the values of F(n) as they are calculated.
- **Build Up Solutions:** For F(n), look up F(n-1) and F(n-2) in the table and add them.
- **Avoid Redundancy:** The table ensures that each subproblem (e.g., F(2)) is solved only once.

Advantages of Dynamic Programming (DP)

Dynamic programming has a wide range of advantages, including:

- Avoids recomputing the same subproblems multiple times, leading to significant time savings.
- Ensures that the optimal solution is found by considering all possible combinations.
- Breaks down complex problems into smaller, more manageable subproblems.

Applications of Dynamic Programming (DP)

Dynamic programming has a wide range of applications, including:

- **Optimization:** Knapsack problem, shortest path problem, maximum subarray problem
- **Computer Science:** Longest common subsequence, edit distance, string matching
- **Operations Research:** Inventory management, scheduling, resource allocation

Divide and Conquer Vs Dynamic approach

The divide and conquer approach divides a problem into smaller subproblems; these subproblems are further solved recursively. The result of each subproblem is not stored for future reference, whereas, in a dynamic approach, the result of each subproblem is stored for future reference.

Use the divide and conquer approach when the same subproblem is not solved multiple times. Use the dynamic approach when the result of a subproblem is to be used multiple times in the future.

Let us understand this with an example. Suppose we are trying to find the Fibonacci series. Then,

Divide and Conquer approach:

```
fib(n)
   If n < 2, return 1
   Else , return f(n - 1) + f(n -2)
```

Dynamic approach:

```
mem = []
fib(n)
   If n in mem: return mem[n]
   else,
      If n < 2, f = 1
      else , f = f(n - 1) + f(n -2)
      mem[n] = f
      return f
```

In a dynamic approach, mem stores the result of each subproblem.

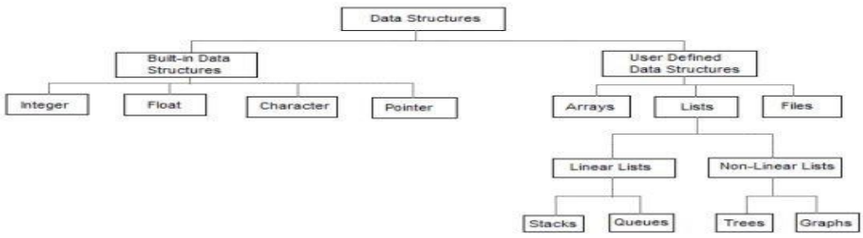
DATA STRUCTURES AND ALGORITHMS

Week 3 (Data Structures Basics)

DATA STRUCTURES

- Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage.
- In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily. It represents the knowledge of data to be organized in memory. It should be designed and implemented in such a way that it reduces the complexity and increases the efficiency.

BASIC TYPES OF DATA STRUCTURES

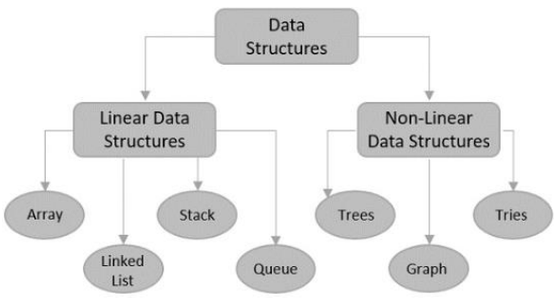


1. **Primitive Data Structures-** are basic data structures provided by programming languages to represent single values, such as integers, floating-point numbers, characters, and booleans.
2. **Complex/Abstract Data Structures-** are higher-level data structures that are built using primitive data types and provide more complex and specialized operations. Some common examples of abstract data structures include arrays, linked lists, stacks, queues, trees, and graphs.

CHARACTERISTICS OF A DATA STRUCTURE

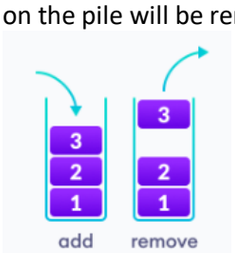
- **Correctness** − Data structure implementation should implement its interface correctly.
- **Time Complexity** − Running time or the execution time of operations of data structure must be as small as possible.
- **Space Complexity** − Memory usage of a data structure operation should be as little as possible.

TYPES OF DATA STRUCTURE



1. **Linear data structures-** elements are accessed in a sequential order but may be stored unsystematically.
    - **Array data structure-** elements in memory are arranged in continuous memory. All the elements of an array are of the same type. And, the type of elements that can be stored in the form of arrays is determined by the programming language.

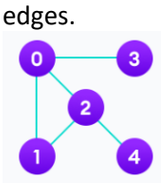
    - **Stack data structure -** In stack data structure, elements are stored in the LIFO principle. That is, the last element stored in a stack will be removed first. It works just like a pile of plates where the last plate kept on the pile will be removed first.



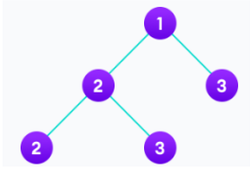
- **Queue data structure-** Unlike stack, the queue data structure works in the FIFO principle where first element stored in the queue will be removed first. It works just like a queue of people in the ticket counter where first person on the queue will get the ticket first.

- **Linked list data structure-** In linked list data structure, data elements are connected through a series of nodes. And, each node contains the data items and address to the next node. They are arranged in a hierarchical manner where one element will be connected to one or more elements.

2. **Non-linear data structures-** elements are stored and accessed in a non-sequential order.
   - **Graph data structure-** each node is called vertex and each vertex is connected to other vertices through edges.



   - **Trees data structure-** Similar to a graph, a tree is also a collection of vertices and edges. However, in tree data structure, there can only be one edge between two vertices.



Linear vs Non-Linear Data Structures

Linear Data Structures	Non Linear Data Structures
The data items are arranged in sequential order, one after the other.	The data items are arranged in non-sequential order (hierarchical manner).
All the items are present on the single layer.	The data items are present at different layers.
It can be traversed on a single run. That is, if we start from the first element, we can traverse all the elements sequentially in a single pass.	It requires multiple runs. That is, if we start from the first element it might not be possible to traverse all the elements in a single pass.
The memory utilization is not efficient.	Different structures utilize memory in different efficient ways depending on the need.
The time complexity increase with the data size.	Time complexity remains the same.
Example: Arrays, Stack, Queue	Example: Tree, Graph, Map

DATA TYPES

Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. There are two data types –

- **Built-in Data Type-** data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provide the following built-in data types.
  - **Integer-** stores both positive numbers and negative numbers along with zero. The integer data type uses all these integers to maintain precision. All the arithmetic operations can be efficiently done through integer data types. If the value of data is beyond the numerical range of the integer, then it is the case that the database server cannot store the value. However, integer data type consumes four (4) bytes of storage per value. *A whole number that can be positive, negative, or zero. Examples of integers include -100, 0, and 100.*
  - **Boolean-** In computer programs, there are three types of data such as numbers, text and booleans. Boolean is a data type, and the value in the boolean can be either false or true or positive or negative. A boolean type of data proves whether the data is valid or invalid. Boolean

values have two possible states such as True or False. In the binary system, the values 0 and 1 are used. Boolean expressions deal with algebra, logical values and binary variables.

- **Floating-Point Numbers-** A floating-point data type approximates real values using a formula in order to allow a trade-off between range and precision. Due to the need for quick processing speeds, systems with extremely small and very large real numbers are frequently found to use floating-point calculation. An exponent in a fixed base is typically used to scale a number and approximate its representation to a specific number of significant digits. *A real number that can contain a fractional component. Floating-point numbers represent currency values. Examples of floating-point numbers include 12.50 and -0.01.*
- **Fixed-Point Numbers-** Binary words are used to store numbers in digital electronics. A fixed-length string of bits (1s and 0s) is a binary word. The data type determines how these 1s and 0s are interpreted by hardware elements and software processes. There are two different data types for binary numbers: fixed-point and floating-point. Both signed and unsigned fixed-point data formats are available. There is no sign bit. Therefore, the binary word does not typically explicitly express whether a fixed-point value is signed or unsigned. In contrast, the architecture of the computer implicitly defines the sign information.
- **Character-** Character information is stored in a fixed-length field by the CHAR data type. Data can be a string of letters, integers, and other characters that are supported by the code set of your database locale, whether they are single-byte or multibyte characters.
- **Pointers-** Blocks of memory that are dynamically allocated are managed and stored using pointers. Data objects or arrays of objects are stored in such alliances. The heap or free store, which is a memory space provided by the majority of structured and object-oriented languages, is where objects are dynamically allocated.
- **String-** A string data type consists of a series of characters, either in the form of a literal constant or a variable. The latter can either be constant in length or allow its elements to alter (after creation). An array data structure of bytes (or words) is frequently used to create a string, which is typically thought of as a sort of data and contains a succession of items, typically characters, using some kind of character encoding.

- **Derived Data Type-** data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built-in data types and associated operations on them.
  >List, Array, Stack, Queue

BASIC OPERATIONS

Data structure operations are the methods used to manipulate the data in a data structure. The most common data structure operations are:

- **Traversal** are used to visit each node in a data structure in a specific order. This technique is typically employed for printing, searching, displaying, and reading the data stored in a data structure.
- **Insertion-** add new data elements to a data structure. You can do this at the data structure's beginning, middle, or end.
- **Deletion-** remove data elements from a data structure. These operations are typically performed on nodes that are no longer needed.
- **Search-** are used to find a specific data element in a data structure. These operations typically employ a compare function to determine if two data elements are equal.
- **Sort-** are used to arrange the data elements in a data structure in a specific order. This can be done using various sorting algorithms, such as insertion sort, bubble sort, merge sort, and quick sort.
- **Merge-** are used to combine two data structures into one. This operation is typically used when two data structures need to be combined into a single structure.
- **Copy-** are used to create a duplicate of a data structure. This can be done by copying each element in the original data structure to the new one.

How to choose a data structure

Now that we have gone over the basics of data structures, it is time for you to learn when to use each type of data structure. The choice of data structure depends on the following factors:

- **The operations that will be performed:** The choice of data structure should be based on the operations performed. For example, you should use a linked list if you need to perform insertions and deletions. If you need to perform indexing, then you should use an array.

- **The time complexity of the operations**: The choice of data structure should be based on the time complexity of the operations that will be performed. For example, if you need to perform searches frequently, you should use a binary search tree.

- **The space complexity of the operations:** The choice of data structure should be based on the space complexity of the operations that will be performed. For example, if you need to store a lot of data, you should use an array.

- **Memory usage:** The choice of data structure should be based on the amount of memory used. For example, if you need to store a lot of data in memory, you should use a linked list.

Algorithms are like verbs and Data Structures are like nouns. *An Algorithm is just a method of doing something on a computer, while a Data Structure is a layout for memory that represents some sort of data.* ***- Om Singh***

DATA STRUCTURES AND ALGORITHMS

Week 4 (Array Data Structure)

- An array is a linear data structure that collects elements of same data type and stores them in a contiguous and adjacent memory locations.

- An **array** is a collection of items of the same variable type that are stored at contiguous memory locations. It's one of the most popular and simple data structures and is often used to implement other data structures. Each item in an array is indexed starting with **0**. Each element in an array is accessed through its index.

Following are the important terms to understand the concept of Array.

- **Element** − Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

Creating an array in **Java** programming language

```
data_type[] array_name = {elements separated by commas}
or,
data_type array_name = new data_type[array_size];

1D Arrays: int arr[n];
2D Arrays: int arr[m][n];
3D Arrays: int arr[m][n][o];
```

How to Initialize an Array?

- Method 1:

```
int a[6] = {2, 3, 5, 7, 11, 13};
```

- Method 2:

```
int arr[]= {2, 3, 5, 7, 11};
```

- Method 3:

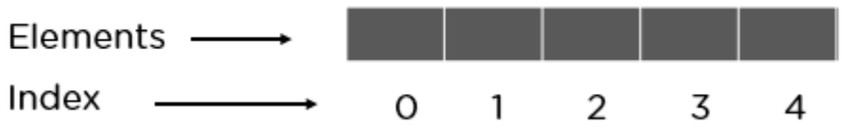
```
int n;
scanf("%d",&n);
int arr[n];
for(int i=0;i<5;i++)
{
scanf("%d",&arr[i]);
}
```

- Method 4:

```
int arr[5];
arr[0]=1;
arr[1]=2;
arr[2]=3;
arr[3]=4;
arr[4]=5;
```

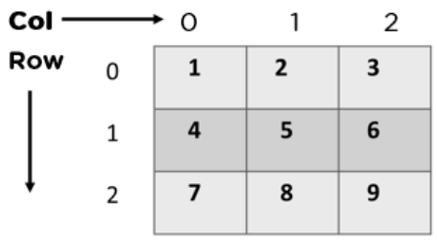
Types of Array:

- **One-dimensional arrays:** These arrays store a single row of elements.

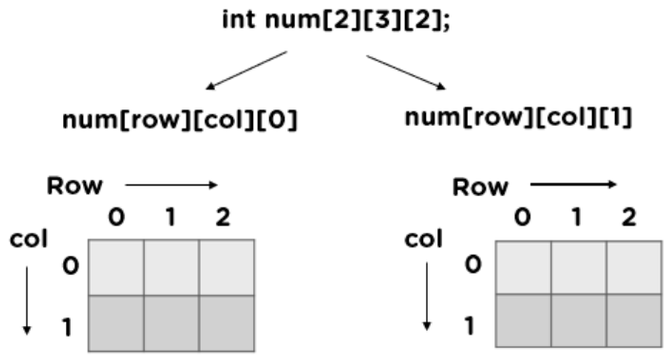


- **Multidimensional arrays:** These arrays store multiple rows of elements.

Two-Dimensional Arrays



Three-Dimensional Arrays

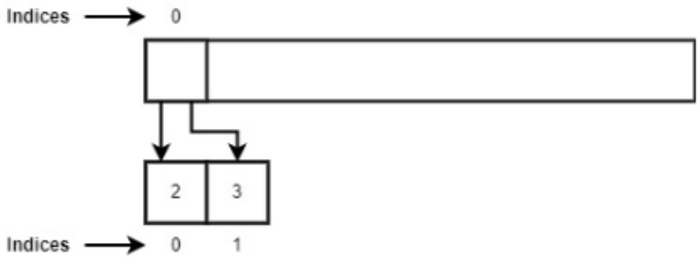


Array Representation

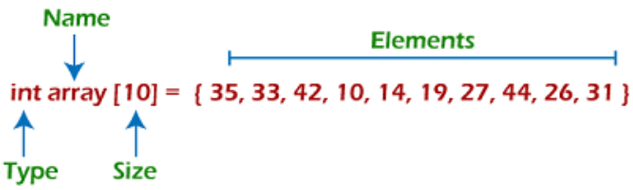
Arrays are represented as a collection of buckets where each bucket stores one element. These buckets are indexed from '0' to 'n-1', where n is the size of that particular array. For example, an array with size 10 will have buckets indexed from 0 to 9.

This indexing will be similar for the multidimensional arrays as well. If it is a 2-dimensional array, it will have sub-buckets in each bucket. Then it will be indexed as array_name[m][n], where m and n are the sizes of each level in the array.

Single Dimensional Array



Multi Dimensional Array



As per the above illustration, there are some of the following important points -

- Index starts with 0.
- The array's length is 10, which means we can store 10 elements.
- Each element in the array can be accessed via its index.

Basic Operations in Arrays

- **Traverse:** Visiting each element of an array in a specific order (e.g., sequential, reverse).
- **Insertion:** Adding a new element to an array at a specific index.
- **Deletion:** Removing an element from an array at a specific index.
- **Search**: Searches an element using the given index or by the value.
- **Update** – Updates an element at the given index.
- **Display:** Displays the contents of the array.

Traversing operation:

This operation is performed to traverse through the array elements. It prints all array elements one after another.

Algorithm

1 Start
2. Initialize an Array of certain size and datatype.
3. Initialize another variable ‘i’ with 0.
4. Print the ith value in the array and increment i.
5. Repeat Step 4 until the end of the array is reached.
6. End

Example

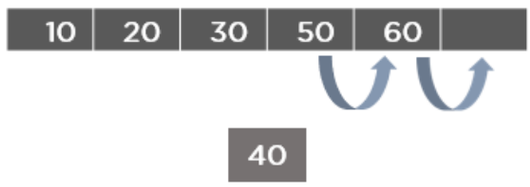
```
public class ArrayDemo {
  public static void main(String []args) {
    int LA[] = new int[5];
    System.out.println("The array elements are: ");
    for(int i = 0; i < 5; i++) {
      LA[i] = i + 2;
      System.out.println("LA[" + i + "] = " + LA[i]);
    }
  }
}
```

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
```

Insertion operation

This operation is performed to insert one or more elements into the array. As per the requirements, an element can be added at the beginning, end, or at any index of the array.



Algorithm

1. Start
2. Create an Array of a desired datatype and size.
3. Initialize a variable 'i' as 0.
4. Enter the element at ith index of the array.
5. Increment i by 1.
6. Repeat Steps 4 & 5 until the end of the array.
7. Stop

Example

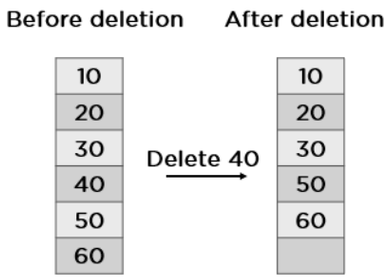
```
public class ArrayDemo {
  public static void main(String []args) {
    int LA[] = new int[3];
    System.out.println("Array Before Insertion:");
    for(int i = 0; i < 3; i++)
      System.out.println("LA[" + i + "] = " + LA[i]); //prints empty array
    System.out.println("Inserting Elements..");

    // Printing Array after Insertion
    System.out.println("Array After Insertion:");
    for(int i = 0; i < 3; i++) {
      LA[i] = i+3;
      System.out.println("LA[" + i + "] = " + LA[i]);
    }
  }
}
```

Output

Array Before Insertion:
LA[0] = 0
LA[1] = 0
LA[2] = 0
Inserting elements..
Array After Insertion:
LA[0] = 2
LA[1] = 3
LA[2] = 4
LA[3] = 5
LA[4] = 6

Deletion operation

This operation removes an element from the array and then reorganizes all of the array elements.



Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to delete an element available at the K^{th} position of LA.

1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop

Example

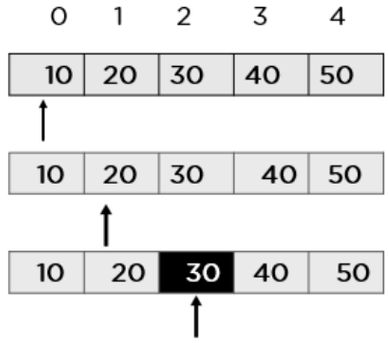
```
public class ArrayDemo {
  public static void main(String []args) {
    int LA[] = new int[3];
    int n = LA.length;
    System.out.println("Array Before Deletion:");
    for(int i = 0; i < n; i++) {
      LA[i] = i + 3;
      System.out.println("LA[" + i + "] = " + LA[i]);
    }
    for(int i = 1; i<n-1; i++) {
      LA[i] = LA[i+1];
      n = n - 1;
    }
    System.out.println("Array After Deletion:");
    for(int i = 0; i < n; i++) {
      System.out.println("LA[" + i + "] = " + LA[i]);
    }
  }
}
```

Output

Array Before Deletion:
LA[0] = 1
LA[1] = 3
LA[2] = 5
Array After Deletion :
LA[0] = 1
LA[1] = 5

Search operation

This operation is performed to search an element in the array based on the value or index.



Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to find an element with a value of ITEM using sequential search.

1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop

Example

```
public class ArrayDemo{
  public static void main(String []args){
    int LA[] = new int[5];
    System.out.println("Array:");
    for(int i = 0; i < 5; i++) {
      LA[i] = i + 3;
      System.out.println("LA[" + i + "] = " + LA[i]);
    }
    for(int i = 0; i < 5; i++) {
      if(LA[i] == 6)
        System.out.println("Element " + 6 + " is found at index " + i);
    }
  }
}
```

Output

```
Array:
LA[0] = 1
LA[1] = 3
LA[2] = 6
LA[3] = 7
LA[4] = 8

Element 6 is found at index 2
```

Update operation

This operation is performed to update an existing array element located at the given index.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to update an element available at the Kth position of LA.

1. Start
2. Set LA[K-1] = ITEM
3. Stop

Example

```
public class ArrayDemo {
  public static void main(String []args) {
    int LA[] = new int[5];
```

```
    int item = 15;
    System.out.println("The array elements are: ");
    for(int i = 0; i < 5; i++) {
      LA[i] = i + 2;
      System.out.println("LA[" + i + "] = " + LA[i]);
    }
    LA[3] = item;
    System.out.println("The array elements after updation are: ");
    for(int i = 0; i < 5; i++)
      System.out.println("LA[" + i + "] = " + LA[i]);
  }
}
```

Output

```
The array elements are::
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after updation are:
LA[0] = 1
LA[1] = 3
LA[2] = 10
LA[3] = 7
LA[4] = 8
```

Display Operation

This operation displays all the elements in the entire array using a print statement.

Algorithm

Consider LA is a linear array with N elements. Following is the algorithm to display an array elements.

1. Start
2. Print all the elements in the Array
3. Stop

Example

```
public class ArrayDemo {
  public static void main(String []args) {
    int LA[] = new int[5];
    System.out.println("The array elements are: ");
    for(int i = 0; i < 5; i++) {
      LA[i] = i + 2;
      System.out.println("LA[" + i + "] = " + LA[i]);
    }
  }
}
```

Output

```
The array elements are:
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
```

Advantages of Array

- Array provides the single name for the group of variables of the same type. Therefore, it is easy to remember the name of all the elements of an array.
- Traversing an array is a very simple process; we just need to increment the base address of the array in order to visit each element one by one.
- Any element in the array can be directly accessed by using the index.

Disadvantages of Array

- Array is homogenous. It means that the elements with similar data type can be stored in it.
- In array, there is static memory allocation that is size of an array cannot be altered.
- There will be wastage of memory if we store less number of elements than the declared size.

Two-Dimensional Array

2D array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.

The Need for Two-Dimensional Arrays

Using 2d arrays, you can store so much data at one moment, which can be passed at any number of functions whenever required.

Picture this, a class consists of 5 students, and the class has to publish the result of all those students.

- You need a table to store all those five students' names, subjects' names, and marks.
- For that, it requires storing all information in a tabular form comprising rows and columns.
- A row contains the name of subjects, and columns contain the name of the students.
- That class consists of four subjects, namely English, Science, and Mathematics and the names of the students are first, second, third, fourth, and fifth.

	First	Second	Third	fourth	Fifth
English	10	55	90	70	60
Science	50	60	75	65	95
Mathematics	95	75	55	85	45

How do we access data in a 2D array

Due to the fact that the elements of 2D arrays can be random accessed. Similar to one dimensional arrays, we can access the individual cells in a 2D array by using the indices of the cells. There are two indices attached to a particular cell, one is its row number while the other is its column number.

Declaration of Two-Dimensional Arrays

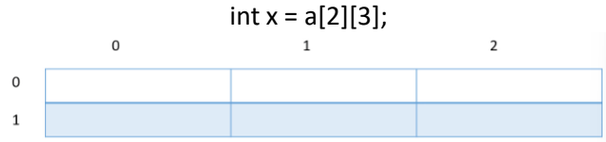
```
Data_type name_of the array[rows][index];

int x = a[i][j];
```

where i and j is the row and column number of the cell respectively.

We can assign each cell of a 2D array to 0 by using the following code:

```
1.  for ( int i=0; i<n ;i++)
2.  {
3.      for (int j=0; j<n; j++)
4.      {
5.          a[i][j] = 0;
6.      }
7.  }
```

int x = a[2][3];



There are two methods to initialize two-dimensional arrays.

Method 1

```
int multi_dim[4][3]={10,20,30,40,50,60,20,80,90,100,110,120};
```

Method 2

```
int multi_dim[4][3]={{10,20,30},{40.50,60},{70,80,90},{100,110,120}};
```

Accessing Two-Dimensional Arrays

Accessing two-dimensional arrays can be done using row index value and column index value.

```
Name_of_the arrays[row_index][column_index];
int multi_dim[4][3]={{10,20,30},{40,50,60},{70,80,90},{100,110,120}};
```

Suppose, in this example, you want to access element 80.

```
Multi_dim[2][1];
```

	0	1	2
0	10	20	30
1	40	50	60
2	70	80	90
3	100	110	120

Note: indexing always starts with zero.

Example : Storing User's data into a 2D array and printing it.

```
1.  import java.util.Scanner;
2.  publicclass TwoDArray {
3.  publicstaticvoid main(String[] args) {
4.     int[][] arr = newint[3][3];
5.     Scanner sc = new Scanner(System.in);
6.     for (inti =0;i<3;i++)
7.     {
8.        for(intj=0;j<3;j++)
9.        {
10.          System.out.print("Enter Element");
11.          arr[i][j]=sc.nextInt();
12.          System.out.println();
13.       }
14.    }
15.    System.out.println("Printing Elements...");
16.    for(inti=0;i<3;i++)
17.    {
18.       System.out.println();
19.       for(intj=0;j<3;j++)
20.       {
21.          System.out.print(arr[i][j]+"\t");
22.       }
23.    }
24. }
25. }
```