Module-1: Introduction to data structures

Introduction to data structures, ADT, Algorithm analysis, asymptotic notations, Recursion, Arrays operations, Searching (linear, binary), Sorting (bubble, selection, insertion, merge, quick).

---

1. Introduction to data structures:

i. Definition

A data structure is a systematic way of organizing, processing, retrieving, and storing data in a computer's memory so it can be accessed and used efficiently. The choice of data structure depends on the problem requirements, data size and expected operations.

Data structures are used in almost every program or software system. Some common examples of data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables.

Data structures are widely applied in the following areas:

- Compiler design
- Operating system
- Statistical analysis package
- DBMS
- Numerical analysis
- Simulation
- Artificial intelligence
- Graphics

ii. Importance of Data Structures

Data structures are one of the fundamental building blocks of computer science and software engineering. They provide an organized way of storing and managing data so that it can be processed efficiently. The selection of an appropriate data structure significantly influences the performance, scalability, and maintainability of software applications. Modern computing systems process enormous volumes of data every second, making efficient data organization essential for achieving high performance and reliability.

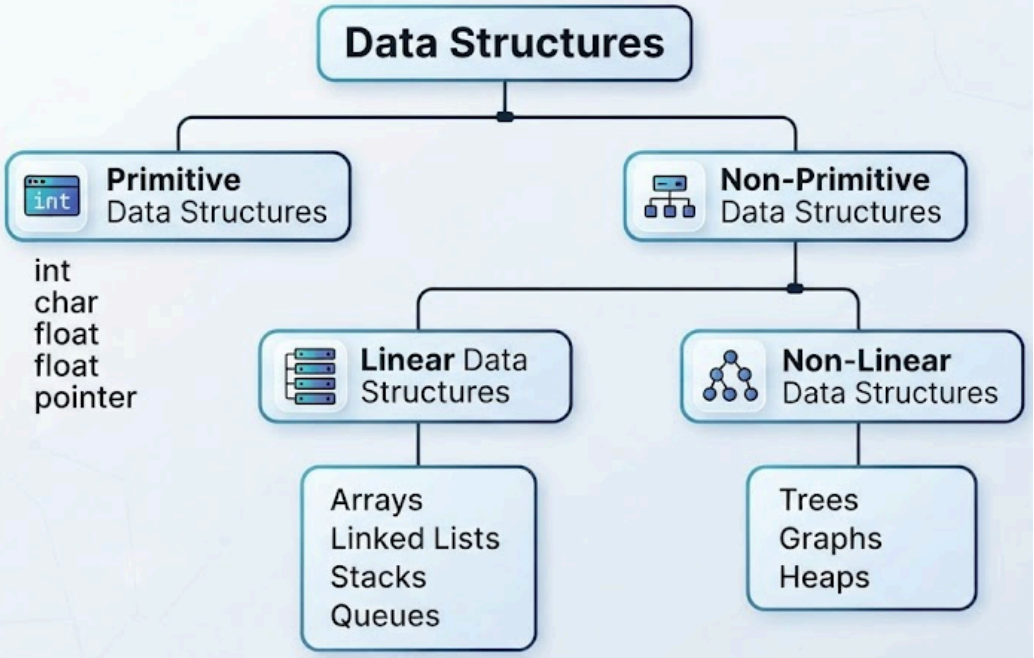
iii. Purpose of Data Structures

The primary purpose of a data structure is to organize and store data in a manner that enables efficient processing, retrieval, and manipulation. As the volume of data handled by modern applications continues to grow, efficient data organization becomes increasingly important. Data structures provide systematic methods for managing information, allowing software systems to perform operations quickly while minimizing the use of computational resources.

iv. Classification of Data Structures

Data structures are classified based on the manner in which data is organized, stored, and accessed in computer memory. This classification helps programmers select the most appropriate structure for solving a particular problem efficiently. The choice of a data structure depends on factors such as the nature of the data, the operations to be performed, memory availability, and execution speed.

Broadly, data structures are classified into **Primitive Data Structures** and **Non-Primitive Data Structures**. Non-primitive data structures are further divided into **Linear Data Structures** and **Non-Linear Data Structures**.



1. Primitive Data Structures

Definition

Primitive data structures are the **basic or fundamental data types** that are directly supported by the programming language and computer hardware. These data types store only a single value and are used as the building blocks for creating more complex data structures.

Primitive data structures occupy a fixed amount of memory and allow basic operations such as arithmetic calculations, logical comparisons, and data assignment.

2. Non-Primitive Data Structures

Definition

Non-primitive data structures are **derived from primitive data types** and are designed to store collections of related data. Unlike primitive structures, they can hold multiple values and establish relationships among data elements.

These structures are used to solve complex computational problems efficiently.

Characteristics

- Store multiple data items.
- Represent relationships among elements.
- Support dynamic operations.
- Improve searching and sorting efficiency.
- Suitable for large-scale applications.

Classification of Non-Primitive Data Structures

Non-primitive data structures are divided into two categories:

- Linear Data Structures
- Non-Linear Data Structures

2.1 Linear Data Structures

Definition

A linear data structure is one in which data elements are arranged **sequentially**. Every element has a unique predecessor and successor except the first and last elements.

The elements are accessed one after another in a single sequence.

Characteristics

- Sequential organization.
- Single level arrangement.
- Easy traversal.
- Simple implementation.
- Memory may be contiguous or non-contiguous depending on the structure.

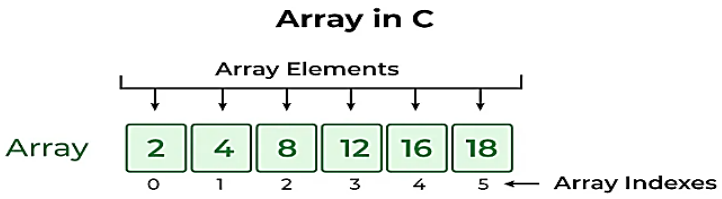
Types of Linear Data Structures

2.1.1 Array

An array stores elements of the same data type in **contiguous memory locations**.

Applications

- Student marks
- Employee salaries
- Image processing
- Matrix operations



2. Abstract Data Type:

i. Definition

An Abstract Data Type (ADT) is a logical model of a data structure that defines what data is stored and what operations can be performed, without specifying how those operations are implemented.

ii. Types of ADTs

The most common types of ADTs used in data structures can be broadly classified into **Linear ADTs** and **Non-Linear ADTs**.

1. Linear Abstract Data Types:

Elements in these ADTs are logically organized sequentially, one after another.

List ADT: An ordered collection of elements that allows duplicates and positional operations.

Core Operations: insert(), remove(), get(), size().

Common Implementations: Dynamic Arrays or Linked Lists.

Stack ADT: A linear structure following the Last-In, First-Out (LIFO) paradigm.

Core Operations: push() (add to top), pop() (remove from top), peek() (view top).

Queue ADT: A linear structure following the First-In, First-Out (FIFO) paradigm.

Core Operations: enqueue() (add to rear), dequeue() (remove from front)

2. **Non-Linear Abstract Data Types:**

Elements in these ADTs are organized hierarchically or interconnectively rather than in a straight sequence

Tree ADT: A hierarchical model consisting of root and child nodes with no cycles.

Core Operations: insert(), delete(), traverse()

Graph ADT: A network of nodes (vertices) connected by links (edges).

Core Operations: addVertex(), addEdge(), getAdjacent()

Priority Queue ADT: A collection where each element has a priority, and the element with the highest priority is served first.

Core Operations: insert(), extractMax() or extractMin()

3. Algorithm Performance analysis

i. Algorithm

An **algorithm** is a finite, well-defined sequence of step-by-step instructions used to solve a specific problem or perform a computation. When working with data structures, algorithms are used to perform operations on the stored data. In simpler terms, it acts like a precise recipe that takes specific inputs, processes them through a structured logical sequence, and delivers a predictable output.

ii. Key Characteristics of an Algorithm

To qualify as a true algorithm, a procedure must possess several fundamental properties

Unambiguous (Definiteness): Every instruction must be completely clear. Each step should have only one interpretation.

Input Specified: It must accept zero or more well-defined external inputs to process.

Output Specified: It must produce at least one clear, intended output or result.

Finiteness: It must eventually stop. The process must terminate after a countable number of execution steps.

Effectiveness (Feasibility): Each step must be basic enough to be performed practically with available resources and time.

Language Independent: The underlying logic must be purely mathematical or structural. It should work the same regardless of the programming language used for implementation.

iii. Performance Analysis

Performance analysis of an algorithm involves determining the amount of computer resources, such as time and memory, needed to execute it.

1. Space Complexity:

Space complexity is the amount of computer memory required during program execution as a function of the input size.

The space needed by a program consists of two primary components:

1.1 Fixed Part

This part varies from problem to problem and remains constant regardless of the specific program's execution flow.

It includes the space required for:

Instructions: The compiled code of the program.

Constants: Fixed values defined within the code.

Variables: Simple variables used for basic data storage.

Structured Variables: Data structures like arrays and structures whose size is determined at compile-time.

1.2 Variable Part

This part varies from program to program and depends on the specific logic and runtime requirements of the algorithm.

It includes space needed for:

Recursion Stack: Memory used to store return addresses and local variables during recursive function calls.

Dynamic Allocation: Structured variables that are allocated space dynamically on the heap during the runtime of a program (e.g., using malloc() or calloc()).

2. Time Complexity:

Time complexity is a key concept used to evaluate the performance of algorithms. It is fundamentally defined as the running time of a program expressed as a function of its input size (n). Specifically, it refers to the number of machine instructions an algorithm executes during its run.

Step Count and **Operation Count** are two fundamental structural techniques used to manually calculate the exact time complexity of an algorithm.

2.1 Step Count Method (Frequency Count)

In this method, you look at every statement in the algorithm, assign a **Step Per Execution (s/e)** value to it, determine its **Frequency** (how many times it runs), and multiply them together to get the total step count.

Standard Assignment Rules:

2.1.1 0 Steps: Comments, variable declarations, and opening/closing brackets ({ }).

2.1.2 1 Step: Simple assignments (x = 5), return statements (return sum), or basic conditionals.

2.1.3 Loop Statements: The efficiency of an algorithm containing loops is determined by the number of iterations performed, which is directly proportional to the loop factor.

The primary goal is to derive a function f(n) that predicts the growth rate of operations as the input size n increases.

The following cases describe how efficiency is calculated for different loop structures:

2.1.3.1 Linear Loops

In a single linear loop, the efficiency is directly proportional to the number of iterations.

Standard Increment: For a loop like for(i=0; i<100; i++), the iteration count matches the loop factor, expressed as f(n)=n.

Variable Increment: If the loop variable increments by more than one (e.g., i+=2), the iterations are reduced by that factor, expressed as f(n)=n/2.

2.1.3.2 Logarithmic Loops

Logarithmic efficiency occurs when the loop-controlling variable is multiplied or divided during each iteration (e.g., i*=2 or i/=2).

Because the value of the controlling variable doubles (or halves) each time, the number of iterations is a function of the multiplier or divisor. For an input size n, the efficiency is expressed as f(n)=log n.

2.1.3.3 Nested Loops

For loops containing other loops, the total efficiency is the product of the iterations of the inner loop and the outer loop. There are three common types of nested loop efficiency:

Linear Logarithmic Loop: This occurs when a linear outer loop contains a logarithmic inner loop (e.g., the inner variable is multiplied by 2 each step). Its efficiency is expressed as f(n)=n log n.

Quadratic Loop: This occurs when both the inner and outer loops execute n times. The generalized formula is $f(n)=n^2$.

Dependent Quadratic Loop: In this case, the inner loop's iterations depend on the current value of the outer loop's index (e.g., for(j=0; j<=i; j++)).

The total number of iterations is the sum of 1+2+3+...+n.

The efficiency is calculated as f(n)=n(n+1)/2

2.2 Operation Count Method

Popularized by computer scientists like Donald Knuth, this method focuses exclusively on **primitive or dominant operations**. Instead of auditing every line of code, you identify the operation that grows the fastest relative to the input size and count only that specific interaction

Standard Rules:

- It skips tracking programmatic scaffolding (like loop overhead, assignments, or return paths).
- It directly counts data-level interactions: data comparisons in sorting or mathematical transformations in matrix algebra.

Example Analysis (Linear Search):

- If you are searching an array of size n for a specific element, the **dominant operation** is the comparison check: if arr[i] == target.
- **Best Case**: The item is at index 0. **Operation Count = 1**.
- **Worst Case**: The item is at the end or missing. **Operation Count = n**.
- **Time Complexity**: **O(n)**.

Key Comparison

Feature	Step Count Method	Operation Count Method
Core Metric	Measures execution based on programmatic "steps per execution" (s/e) per line.	Measures execution based on specific "dominant" or primitive hardware/logical operations.
What is Counted	Assignments, returns, loop declarations, and function invocations.	Arithmetic operations (+, -, *, /) or comparisons (<, >, ==).
Granularity	Coarser: a full statement often counts as just 1 step.	Finer: a single complex statement may contain multiple operations.
Goal	Finds total programmatic instruction frequency.	Finds the behavior of the most computationally expensive operation.

4. Asymptotic notations:

The efficiency analysis framework concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency.

To compare and rank such orders of growth, computer scientists use three notations, they are:

1. O - Big oh notation
2. Ω - Big omega notation
3. Θ - Big theta notation

Let t(n) and g(n) can be any nonnegative functions defined on the set of natural numbers. The algorithm's running time t(n) usually indicated by its basic operation count C(n), and g(n) will be some simple function to compare with the count.

Notation	Represents	Property
Big O (O)	Upper bound	Used for worst-case analysis; indicates the maximum time an algorithm can take.
Omega (Ω)	Lower bound	Used for best-case analysis; indicates the minimum time an algorithm can take.

Theta (Θ)	Tight bound	Used for average-case analysis; indicates that the algorithm's performance is bounded by both a lower and upper limit, growing at the same rate.

(i) O - Big oh notation

Big-O notation represents the upper bound of the running time of an algorithm. For any value of n, the maximum time required by the algorithm is given by Big-O O(g(n)).

A function t*(n)* is said to be in *O(g(n))*, denoted $t\ (n) \in O(g(n))$, if *t (n)* is bounded above by some constant multiple of *g(n)* for all large *n,* i.e., if there exist some positive constant *c* and some nonnegative integer n_0 such that,

$$t\ (n) \leq cg(n) \text{ for all } n \geq n_0.$$

The definition is illustrated in Figure 1 where, for the sake of visual clarity, n is extended to be a real number.

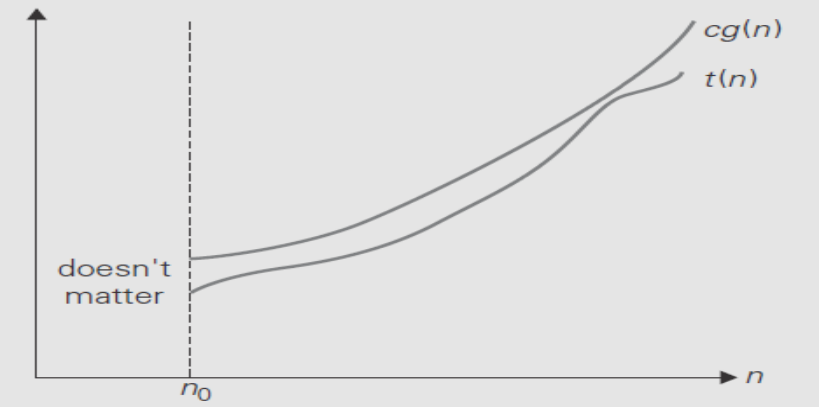


FIGURE 1 Big-oh notation: $t\ (n) \in O(g(n))$.

Example 1: Prove the assertions $100n + 5 \in (n^2)$.

Proof: $100n + 5 \leq 100n + n$ (for all $n \geq 5$)

$= 101n \leq 101n^2$

Since, the definition gives us a lot of freedom in choosing specific values for constants **c** and $\mathbf{n_0}$ we have c=101 and n_0=5.

Example 2: Prove the assertions $100n + 5 \in (n)$.

Proof: $100n + 5 \leq 100n + 5n$ (for all $n \geq 1$) $= 105n$

i.e., $100n + 5 \leq 105n$

i.e., $t(n) \leq cg(n)$

$100n + 5 \in (n)$ with c=105 and n_0=1.

(ii) Ω - Big omega notation

Omega notation represents the lower bound of the running time of an algorithm. For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n)).

A function *t(n)* is said to be in *Ω(g(n))*, denoted *t(n)* ∈ *Ω(g(n))*, if *t(n)* is bounded below by some positive constant multiple of *g(n)* for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that,

t(n) ≥ cg(n) for all n ≥ n0.

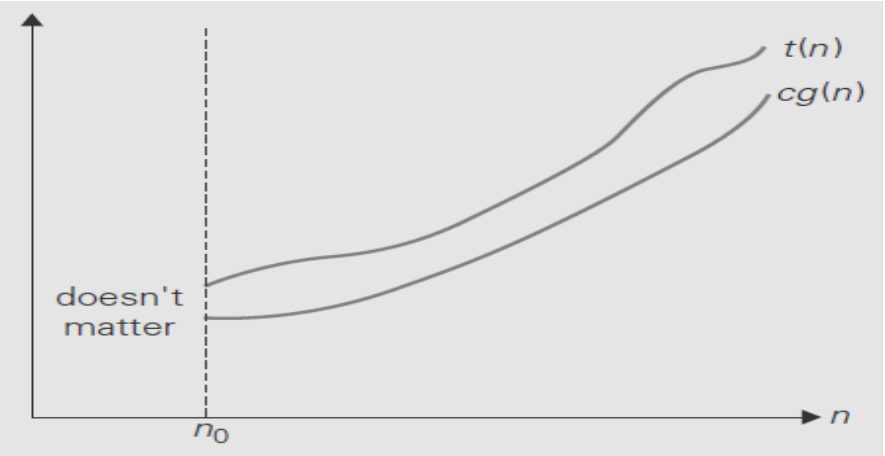


FIGURE 2 Big-omega notation: t (n) ∈ Ω (g(n)).

Example 1: Prove the assertions $n^3 \in \Omega(n^2)$.

Proof: $n^3 \geq n^2$ (for all $n \geq 0$)

i.e., by definition t(n) ≥ cg(n), where c=1 and n_0=0.

(iii) Θ - Big theta notation

Theta notation encloses the function from above and below, since it represents the upper and the lower bound of the running time of an algorithm. For any value of n, the average time required by the algorithm is given by Theta Θ(g(n)).

A function t(n) is said to be in *Θ(g(n))*, denoted t(n) ∈ Θ(g(n)), if t(n) is bounded both above and below by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that,

c_2g(n) ≤ t (n) ≤ c_1g(n) for all n ≥ n0.

If a function t(n) lies anywhere in between c_1g(n) and c_2g(n) for all n ≥ n0, then t(n) is said to be tight bound.

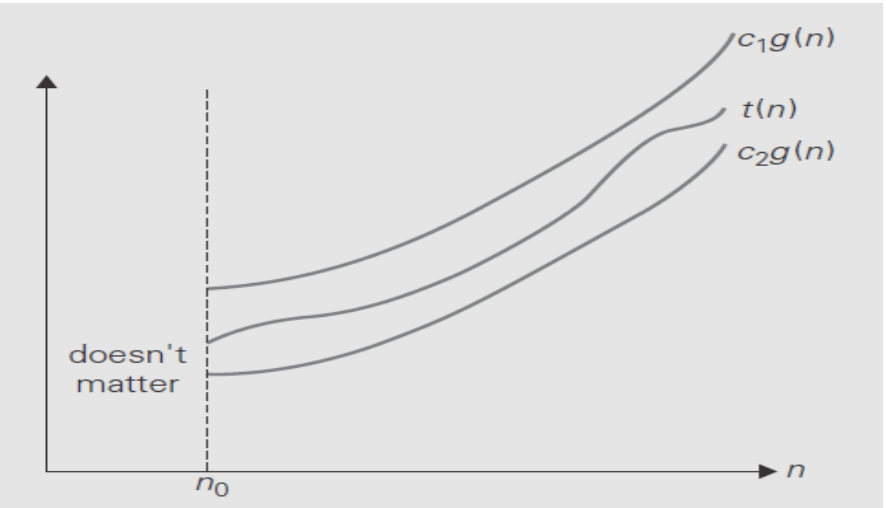


FIGURE 3 Big-theta notation: t (n) ∈ Θ(g(n)).

Example 1: Prove the assertions, $\frac{1}{2}n(n-1) \in \Theta(n^2)$

Proof: First, we prove the right inequality (the upper bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \quad \text{for all } n \geq 0.$$

Second, we prove the left inequality (the lower bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \text{ (for all } n \geq 2) = \frac{1}{4}n^2.$$

Hence, we can select $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$, and $n_0 = 2$.

Worst-Case, Best-Case, and Average-Case Efficiencies:

Worst-case efficiency:

The maximum amount of time or resources an algorithm can ever consume. In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed. This is the most commonly used analysis of algorithm. Most of the time we consider the case that causes maximum operations.

Best case efficiency:

The absolute fastest an algorithm can run. In the best-case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed. The number of operations in the best case is constant (not dependent on n). So the order of growth of time taken in terms of input size is constant.

Average case efficiency:

In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs. The average case analysis is not easy to do in most practical cases and it is rarely done. In the average case analysis, we need to consider every input, its frequency and time taken by it which may not be possible in many scenarios.

5. Recursion:

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.

- A recursive algorithm takes one step toward solution and then recursively call itself to further move. The algorithm stops once we reach the solution.

- Since called function may further call itself, this process might continue forever. So it is essential to provide a base case to terminate this recursion process.

Example 1: Factorial of a Number

The factorial of a number n (where n >= 0) is the product of all positive integers from 1 to n. To compute the factorial recursively, we calculate the factorial of n by using the factorial of (n-1). The base case for the recursive function is when n = 0, in which case we return 1.

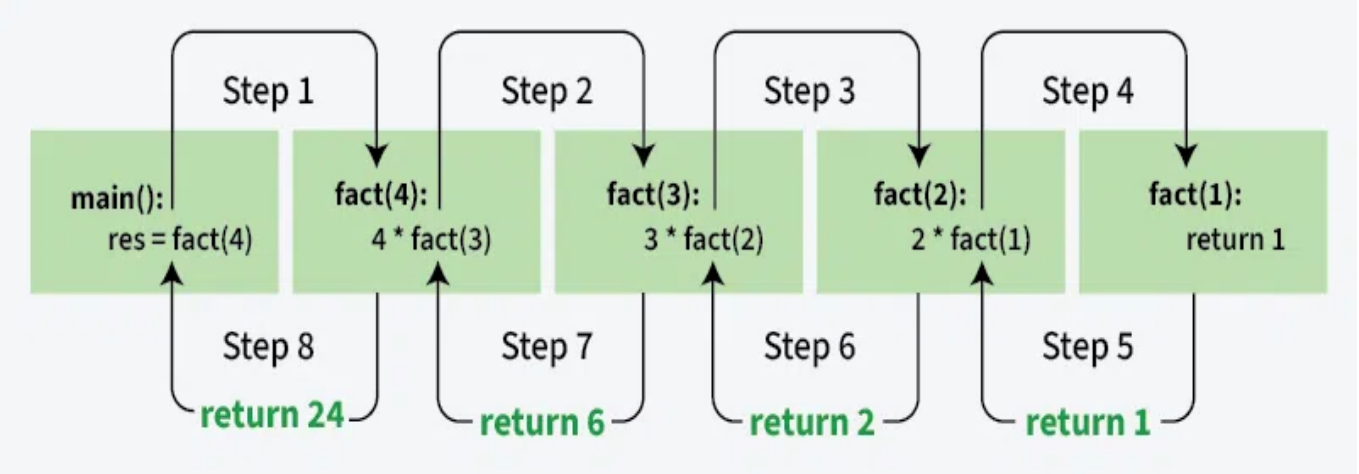
Program:

```
#include <stdio.h>
int fact(int n) {
     // Base Condition
   if (n == 0)
      return 1;
   return n * fact(n - 1);
}
int main() {
   printf("Factorial of 5 : %d\n", fact(5));
   return 0;
}
```

Output:

```
Factorial of 5: 120
```

Illustration of the above code:



Example 2: Fibonacci with Recursion

Write a program and recurrence relation to find the Fibonacci series of n where n >= 0.

Mathematical Equation:

$$n \text{ if } n == 0, n == 1;$$

$$fib(n) = fib(n-1) + fib(n-2) \text{ otherwise;}$$

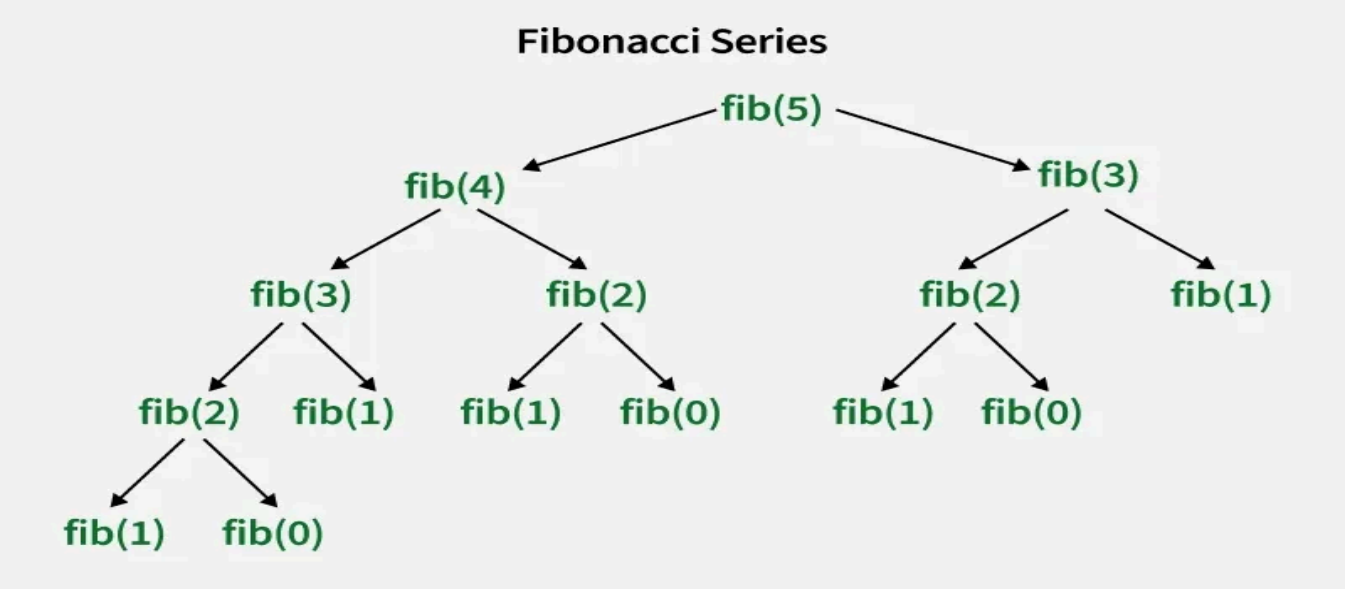
Program:

```
#include <stdio.h>
int fib(int n)                          // Function for fibonacci
{
   if (n == 0)
      return 0;
   if (n == 1 || n == 2)
      return 1;
   else
      return (fib(n - 1) + fib(n - 2));        // Recursion function
}
int main()
{
   int n = 5;
   printf("Fibonacci series of %d number is: ",n);
   for (int i = 0; i < n; i++)                  // for loop to print the fibonacci series.
   {
      printf("%d ", fib(i));
   }
   return 0;
}
```

Output

```
Fibonacci series of 5 numbers is: 0 1 1 2 3
```

Recursion Tree for the above Code:



Example 3: Tower of Hanoi

The **Tower of Hanoi** is a classic mathematical puzzle involving three rods (**A, B, and C**) and **n** disks of different sizes. Initially, all disks are stacked on rod A in decreasing order of diameter - the largest disk at the bottom and the smallest at the top. Goal is to move the entire stack to another rod (rod C) while following these rules:

- Move only one disk at a time.
- At each step, you can take the top disk from any rod and place it on another rod.
- A disk can only be moved if it is the topmost disk of a rod.
- No larger disk may be placed on top of a smaller disk.

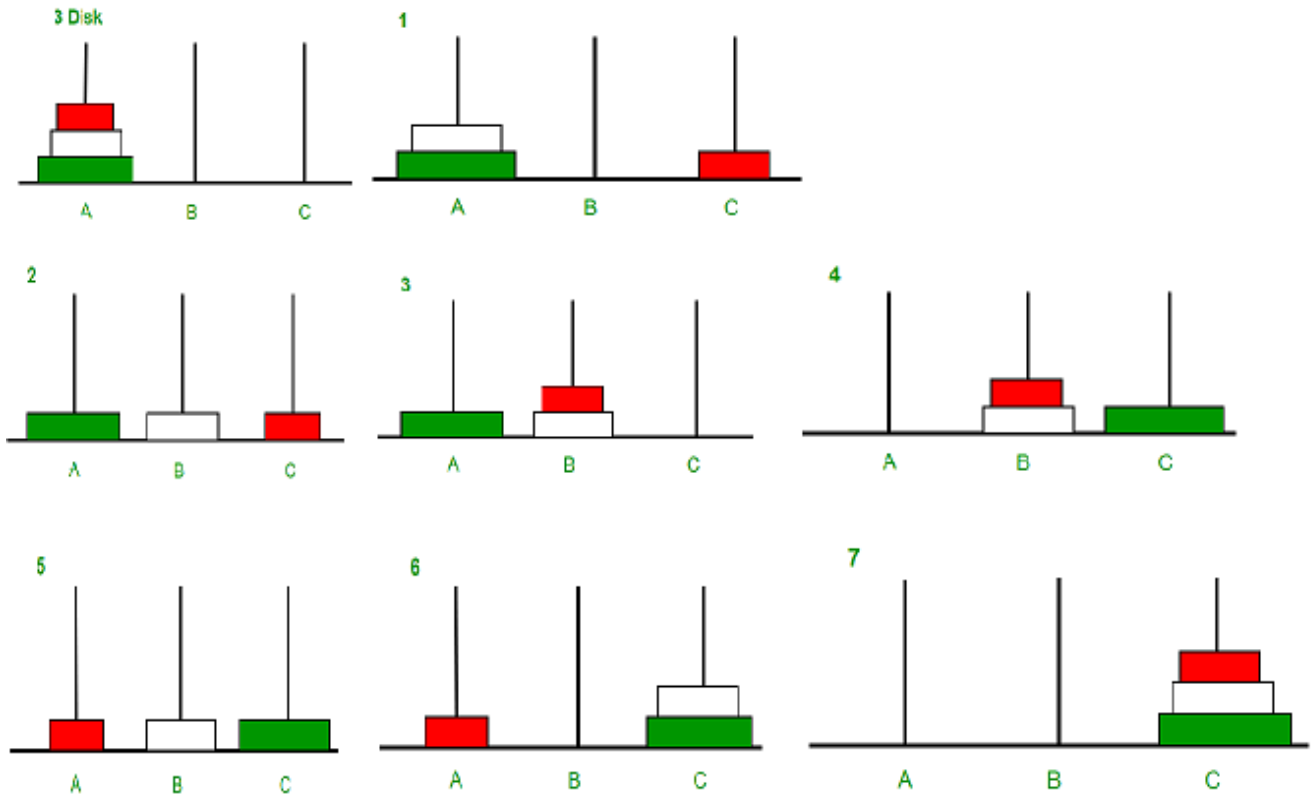
The idea is to use the helper node to reach the destination using recursion. Below is the pattern for this problem:

- *Shift 'n-1' disks from 'A' to 'B', using C.*
- *Shift last disk from 'A' to 'C'.*
- *Shift 'n-1' disks from 'B' to 'C', using A.*

Follow the steps below to solve the problem:

- Create a function towerOfHanoi where pass the n (current number of disk), fromRod, toRod, auxRod.
- Make a function call for n - 1 th disk.
- Then print the current the disk along with from_rod and to_rod
- Again make a function call for n - 1 th disk.

Illustrations:



Program:

```
#include <stdio.h>
void towerOfHanoi(int n, char fromRod, char toRod, char auxRod) {
    if (n == 0) {
        return;
    }
    towerOfHanoi(n - 1, fromRod, auxRod, toRod);
    printf("Disk %d moved from %c to %c\n", n, fromRod, toRod);
```

```
    towerOfHanoi(n - 1, auxRod, toRod, fromRod);
}
int main() {
    int n = 3;
    // A, B and C are names of rods
    towerOfHanoi(n, 'A', 'C', 'B');
    return 0;
}
```

Output:

```
Disk 1 moved from A to C
Disk 2 moved from A to B
Disk 1 moved from C to B
Disk 3 moved from A to C
Disk 1 moved from B to A
Disk 2 moved from B to C
Disk 1 moved from A to C
```

6. Arrays

i. Definition

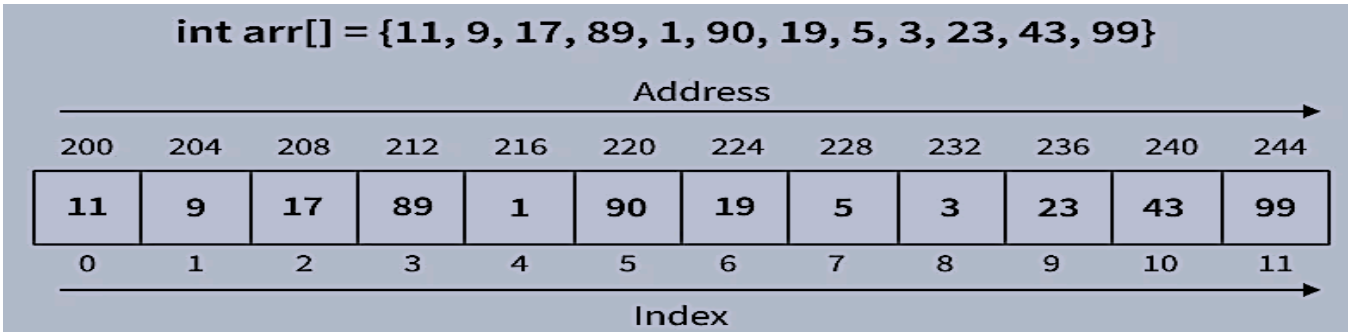
An array is a fundamental and linear data structure that stores collection of data elements having the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an index (also known as the *subscript*). The subscript is an ordinal number which is used to identify an element of the array.

Basic terminologies of Array

- **Array Element:** Elements are items stored in an array.
- **Array Index:** Elements are accessed by their indexes. Indexes in most of the programming languages start from 0.

Memory representation of Array

In an array, all the elements or their references are stored in contiguous memory locations. This allows for efficient access and manipulation of elements.



Declaration of Array

Arrays can be declared in various ways in different languages. For better illustration, below are some language-specific array declarations:

```
int arr[5];        // This array will store integer type element
char arr[10];      // This array will store char type element
float arr[20];     // This array will store float type element
```

Initialization of Array

Arrays can be initialized in different ways in different languages. Below are some language-specific array initialization:

```
int arr[] = { 1, 2, 3, 4, 5 };
char arr[5] = { 'a', 'b', 'c', 'd', 'e' };
float arr[10] = { 1.4, 2.0, 24, 5.0, 0.0 };
```

ii. Operations

1. Traversal in Array

Traversal in an array refers to the process of accessing each element in the array sequentially, typically to perform a specific operation, such as searching, sorting, or modifying the elements. Traversing an array is essential for a wide range of tasks in programming, and the most common methods of traversal include iterating through the array using loops like for, while, or foreach. Efficient traversal plays a critical role in algorithms that require scanning or manipulating data.

Example:

Input: *arr[] = [10, 20, 30, 40, 50]*

Output: *"10 20 30 40 50 "*

Explanation: *Just traverse and print the numbers.*

Types of Array Traversal

There are mainly two types of array traversal:

1.1 Linear Traversal

Linear traversal is the process of visiting each element of an array sequentially, starting from the first element and moving to the last element. During this traversal, each element is processed (printed, modified, or checked) one after the other, in the order they are stored in the array. This is the most common and straightforward way of accessing the elements of an array.

1.2 Reverse Traversal

Reverse traversal is the process of visiting each element of an array starting from the last element and moving towards the first element. This method is useful when you need to process the elements of an array in reverse order. In this type of traversal, you begin from the last index (the rightmost element) and work your way to the first index (the leftmost element).

Program:

```
#include <stdio.h>
int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Linear Traversal: ");
    for(int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    printf("Reverse Traversal: ");
    for(int i = n - 1; i >= 0; i--) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Output:

```
Linear Traversal: 1 2 3 4 5
Reverse Traversal: 5 4 3 2 1
```

2. Insert Element at a Given Position in an Array

Given an array of integers, the task is to insert an element at a **given position** in the array.

Example:

Input: *arr[] = [10, 20, 30, 40], pos = 2, ele = 50*

Output: *[10, 50, 20, 30, 40]*

To add an element at a given position in an array, shift all the elements from that position one index to the right, and after shifting insert the new element at the required position.

Program:

```
#include <stdio.h>
#include <string.h>
int main() {
    int n = 4;
    int arr[] = {10, 20, 30, 40, 0};
    int ele = 50;
    int pos = 2;
    printf("Array before insertion\n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    for(int i = n; i >= pos; i--)                    // Shifting elements to the right
```

```
        arr[i] = arr[i - 1];
    arr[pos - 1] = ele;                    // Insert the new element at index pos - 1
    printf("\nArray after insertion\n");
    for (int i = 0; i <= n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Output:

```
Array before insertion
10 20 30 40
Array after insertion
10 50 20 30 40
```

3. Delete an Element from a Given Position in an Array

Given an array of integers, the task is to delete an element from a **given position** in the array.

Example:

Input: *arr[] = [10, 20, 30, 40], pos = 1*

Output: *[20, 30, 40]*

The idea is to shift all the elements occurring after the given position, one index to the left and reduce the size of the array by 1.

Program:

```
#include <stdio.h>
int main() {
    int arr[] = { 10, 20, 30, 40 };
    int n = sizeof(arr)/sizeof(arr[0]);
    int pos = 2;
    printf("Array before deletion\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    // Delete the element at the given position
    for (int i = pos; i < n; i++) {
        arr[i - 1] = arr[i];
    }
    if (pos <= n)
        n--;
```

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

Output

```
Array before deletion
10 20 30 40
Array after deletion
10 30 40
```

iii. Searching

Searching in data structures is the algorithmic process of locating a specific target element or key within a collection of stored data. A search operation is successful if the element is found and returns its position, index, or value; otherwise, it is deemed unsuccessful.

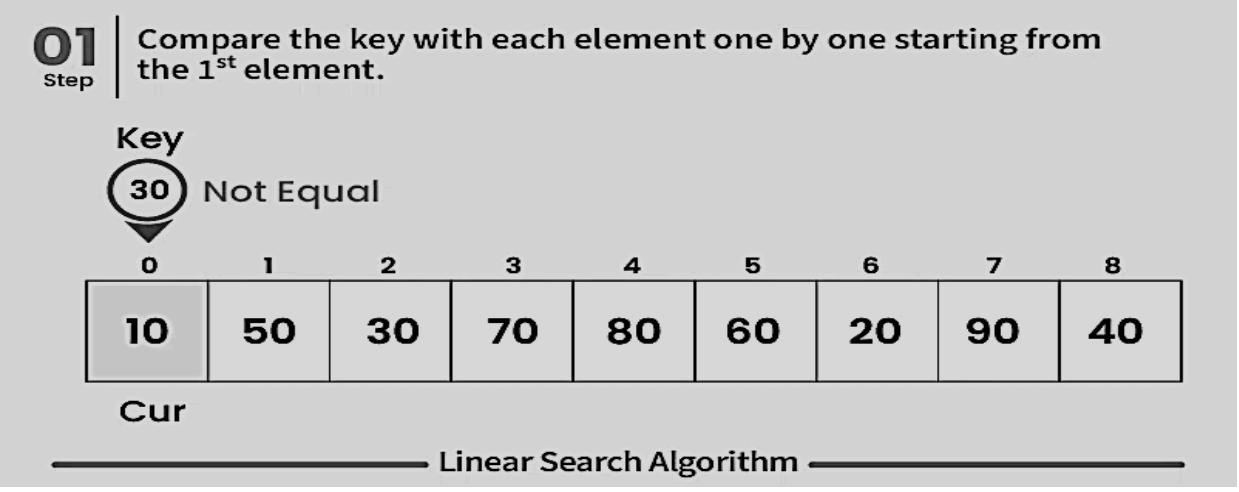
When we search an item in an array, there are two most common algorithms used based on the type of input array.

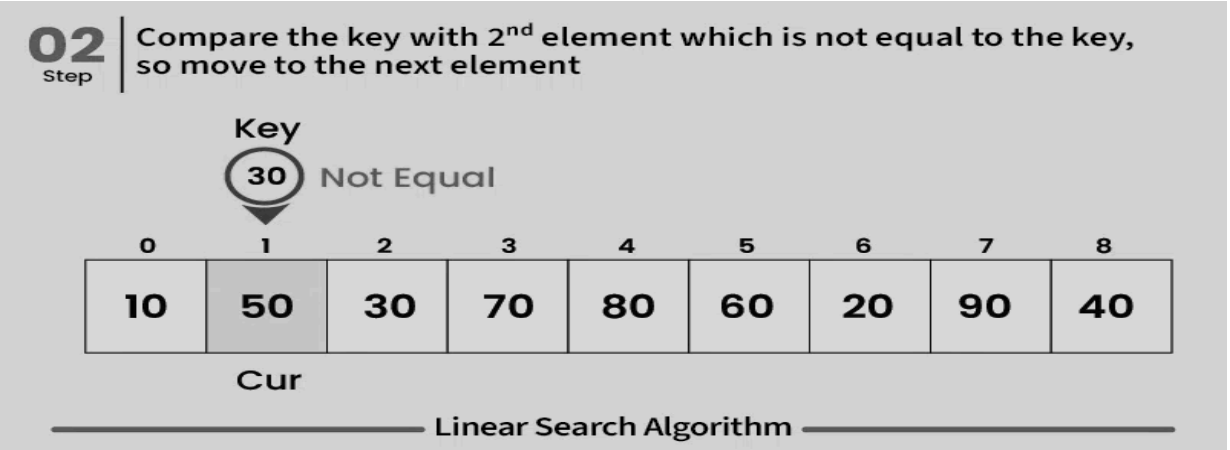
1. Linear Search:

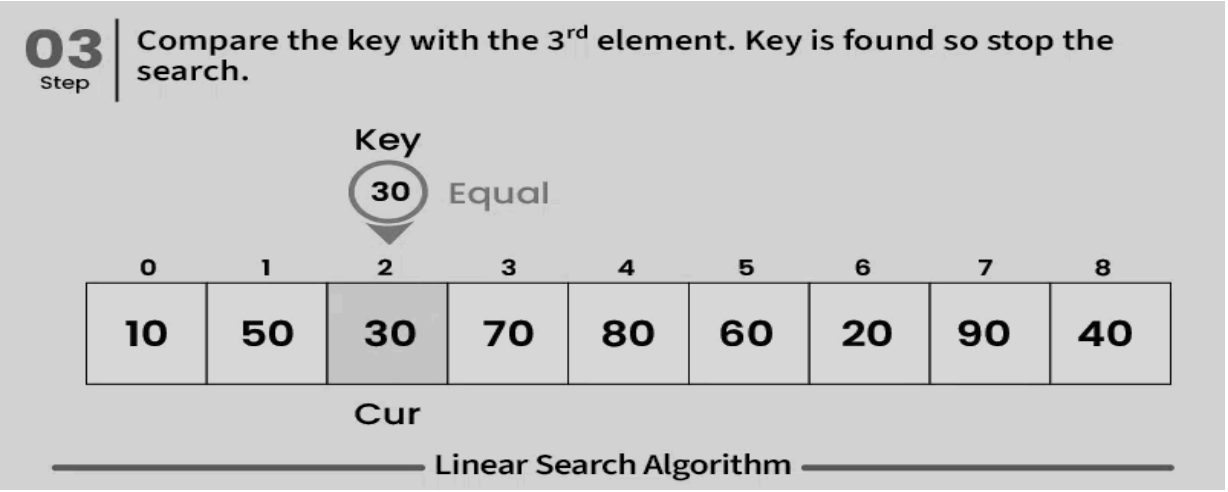
It is used for an unsorted array. It mainly does one by one comparison of the item to be search with array elements. It takes linear or O(n) Time.

In Linear Search, we iterate over all the elements of the array and check if it the current element is equal to the target element. If we find any element to be equal to the target element, then return the index of the current element. Otherwise, if no element is equal to the target element, then return -1 as the element is not found. Linear search is also known as **sequential search**.

For example: Consider the array **arr[] = {10, 50, 30, 70, 80, 20, 90, 40}** and **key** = 30







Program:

```
#include <stdio.h>
int search(int arr[], int n, int x) {
    // Iterate over the array in order to
   // find the key x
   for (int i = 0; i < n; i++)
      if (arr[i] == x)
         return i;
   return -1;
}
int main(void) {
   int arr[] = { 2, 3, 4, 10, 40 };
   int x = 10;
   int n = sizeof(arr) / sizeof(arr[0]);
   // Function call
   int result = search(arr, n, x);
   (result == -1)
```

```
        ? printf("Element is not present in array")
        : printf("Element is present at index %d", result);
    return 0;
}
```

Output:

```
Present at Index 3
```

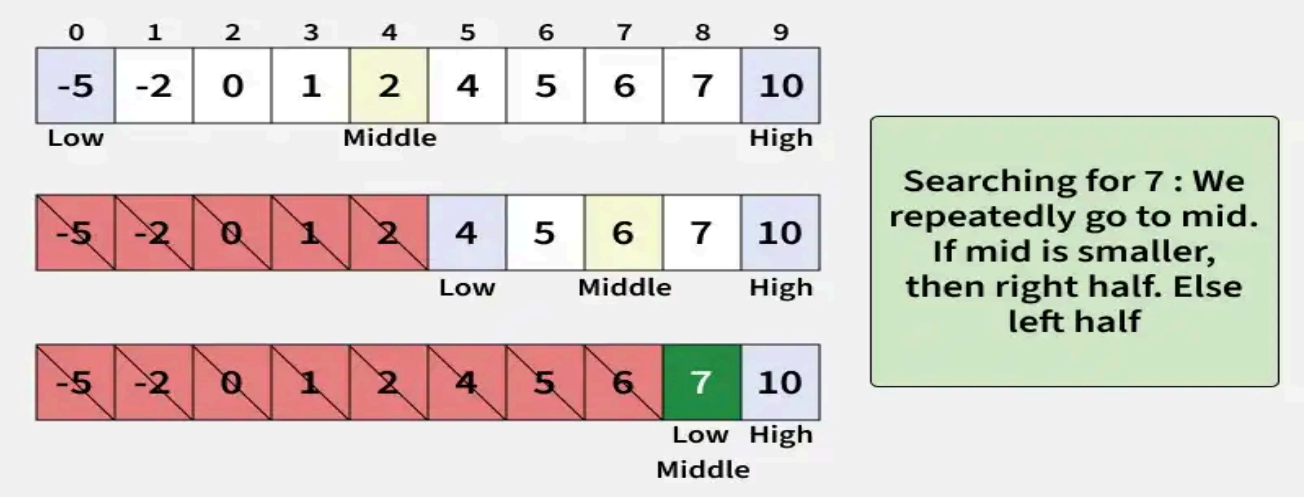
Time and Space Complexity of Linear Search Algorithm:

Time Complexity:

- **Best Case:** In the best case, the key might be present at the first index. So the best case complexity is O(1)
- **Worst Case:** In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst-case complexity is O(N) where N is the size of the list.
- **Average Case:** O(N)

2. Binary Search

Binary Search is a searching algorithm that operates on a sorted or monotonic search space, repeatedly dividing it into halves to find a target value or optimal answer in logarithmic time O (log N). It mainly compares the array's middle element first and if the middle element is same as input, then it returns. Otherwise it searches in either left half or right half based on comparison result (Whether the mid element is smaller or greater).



Conditions to apply Binary Search Algorithm in a Data Structure

- The data structure must be sorted.
- Access to any element of the data structure should take constant time.

Binary Search Algorithm

- Divide the search space into two halves by **finding the middle index "mid"**.

- Compare the middle of the search space with the **key**.
- If the **key** is found at middle, the process is terminated.
- If the **key** is not found at middle, choose which half will be used as the next search space.

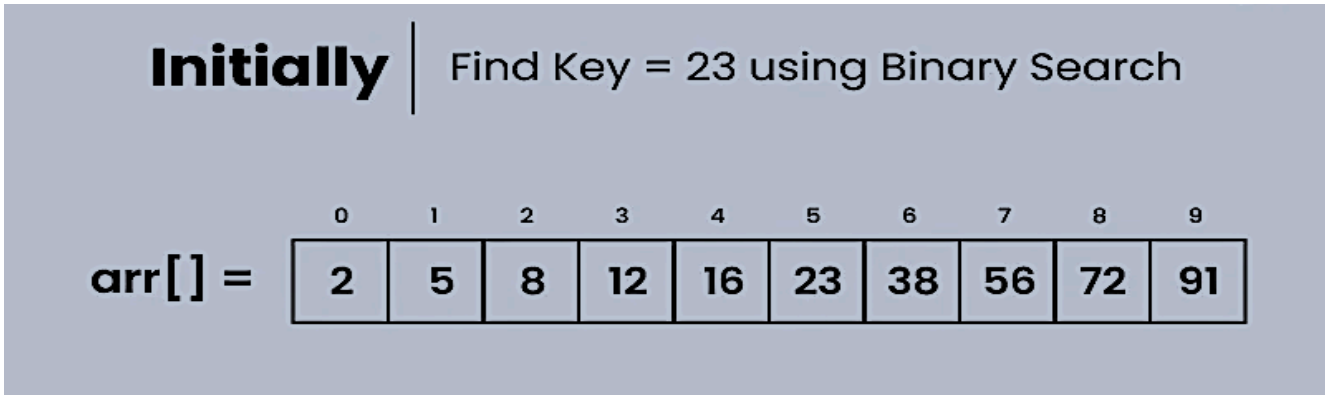
  -> If the **key** is smaller than the middle, then the **left** side is used for next search.

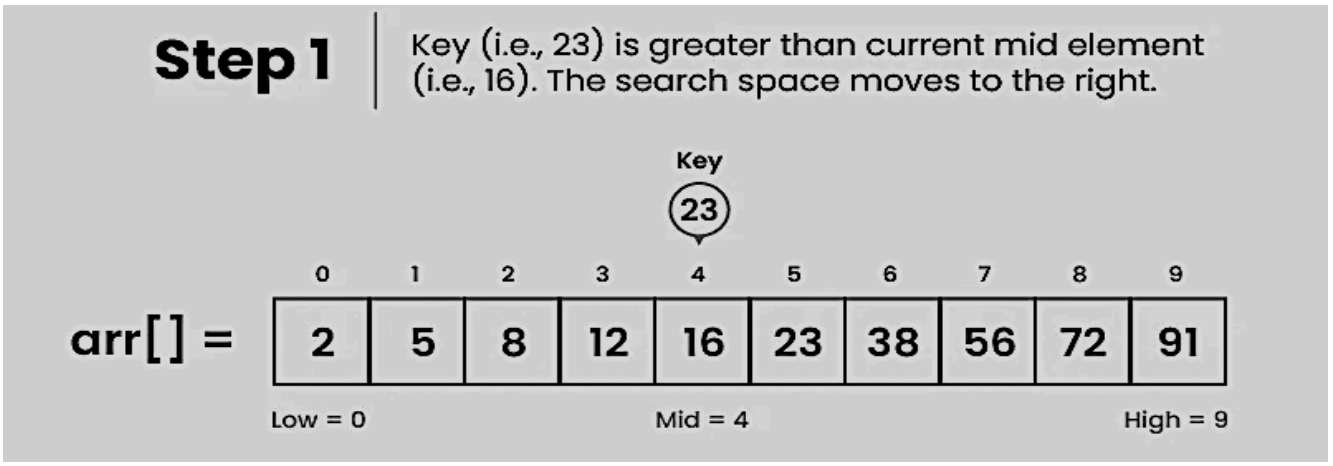
  -> If the **key** is larger than the middle, then the **right** side is used for next search.
- This process is continued until the **key** is found or the total search space is exhausted.

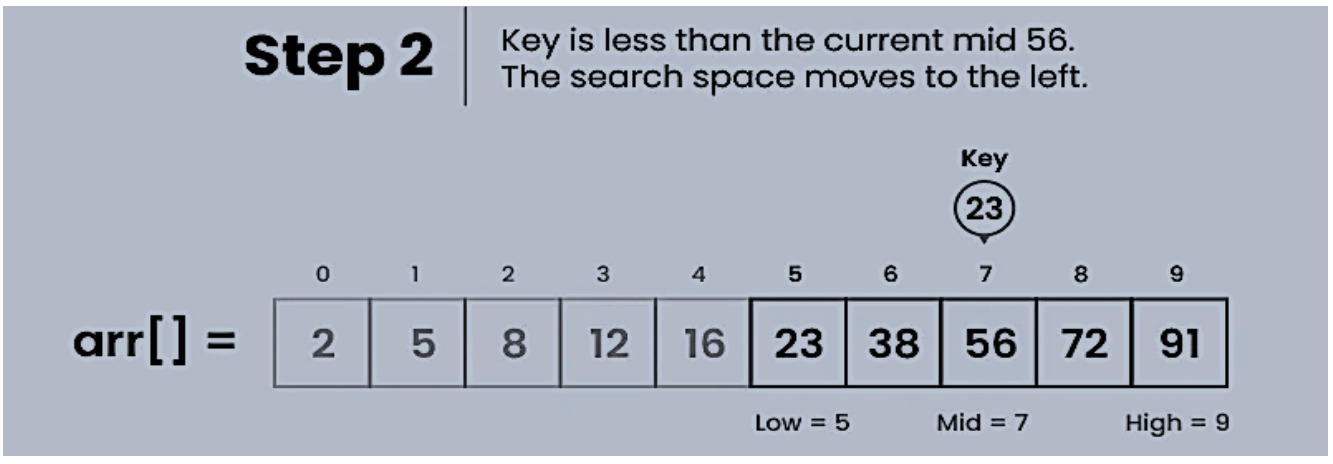
How does Binary Search Algorithm work?

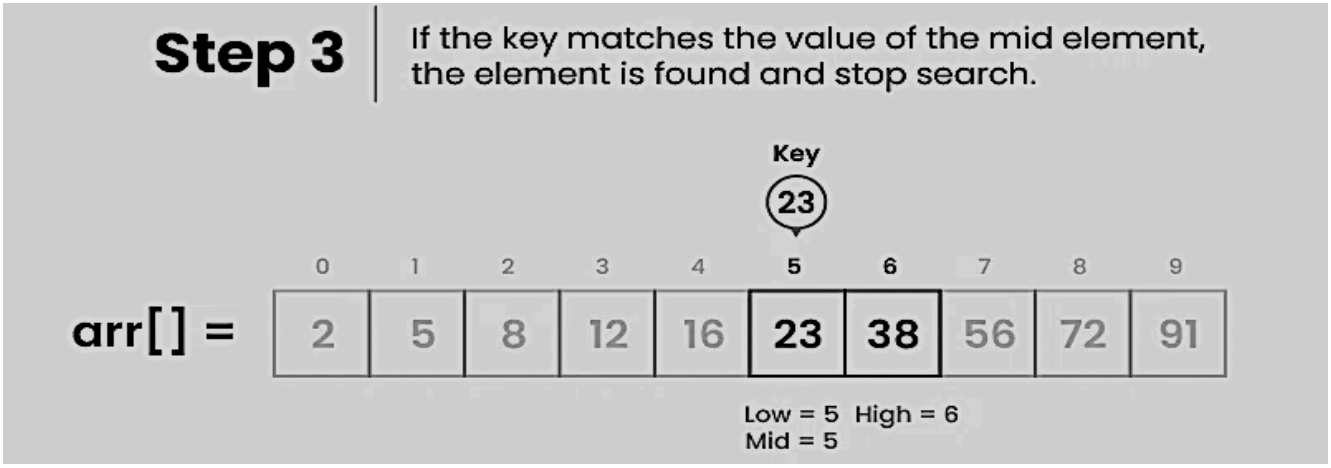
To understand the working of binary search, consider the following illustration:

Consider an array **arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}**, and the **target = 23**.









How to Implement Binary Search?

It can be implemented in the following two ways

- Iterative Binary Search Algorithm
- Recursive Binary Search Algorithm

2.1 Iterative Algorithm: O(log n) Time and O(1) Space

Here we use a while loop to continue the process of comparing the key and splitting the search space in two halves.

Program:

```
#include <stdio.h>
int binarySearch(int arr[], int n, int x) {
   int low = 0;
   int high = n-1;
   while (low <= high) {
      int mid = low + (high - low) / 2;
      // Check if x is present at mid
      if (arr[mid] == x)
         return mid;
      // If x greater, ignore left half
      if (arr[mid] < x)
         low = mid + 1;
      // If x is smaller, ignore right half
      else
         high = mid - 1;
   }
   // If we reach here, then element was not present
   return -1;
}
```

```
int main() {
   int arr[] = { 2, 3, 4, 10, 40 };
   int x = 10;
   int n = sizeof(arr) / sizeof(arr[0]);
   int result = binarySearch(arr, n, x);
   if(result == -1) printf("Element is not present in array");
   else printf("Element is present at index %d",result);
}
```

Output:

```
Element is present at index 3
```

2.2 Recursive Algorithm: O(log n) Time and O(Log n) Space

Create a recursive function and compare the mid of the search space with the key. And based on the result either return the index where the key is found or call the recursive function for the next search space.

Program:

```
#include <stdio.h>
// A recursive binary search function. It returns
// location of x in given array arr[low..high] is present,
// otherwise -1
int binarySearch(int arr[], int low, int high, int x) {
   if (high >= low) {
      int mid = low + (high - low) / 2;
      // If the element is present at the middle
      // itself
      if (arr[mid] == x)
         return mid;
      // If element is smaller than mid, then
      // it can only be present in left subarray
      if (arr[mid] > x)
         return binarySearch(arr, low, mid - 1, x);
      // Else the element can only be present
      // in right subarray
      return binarySearch(arr, mid + 1, high, x);
   }
   // We reach here when element is not
```

```
    // present in array
    return -1;
}
int main()
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    if (result == -1) printf("Element is not present in array");
    else printf("Element is present at index %d", result);
    return 0;
}
```

Output:

```
Element is present at index 3
```

Time Complexity:

-> Best Case: O(1)

-> Average Case: O(log N)

-> Worst Case: O(log N)

iv. Sorting

Sorting is the process of arranging data elements in a specific logical order, most commonly numerical or alphabetical (lexicographical). Organizing data efficiently optimizes the performance of other foundational procedures, such as reducing lookup times via binary search.

1. Bubble Sort

Bubble Sort:

Bubble sort is a simple and well-known sorting algorithm. Bubble sort belongs to $O(n^2)$ sorting algorithms, which makes it quite inefficient for sorting large data volumes. Bubble sort is **stable** and **adaptive**.

Technique:

1. Compare each pair of adjacent elements from the beginning of an array and, if they are in reversed order, swap them.
2. If at least one swap has been done, repeat step 1.

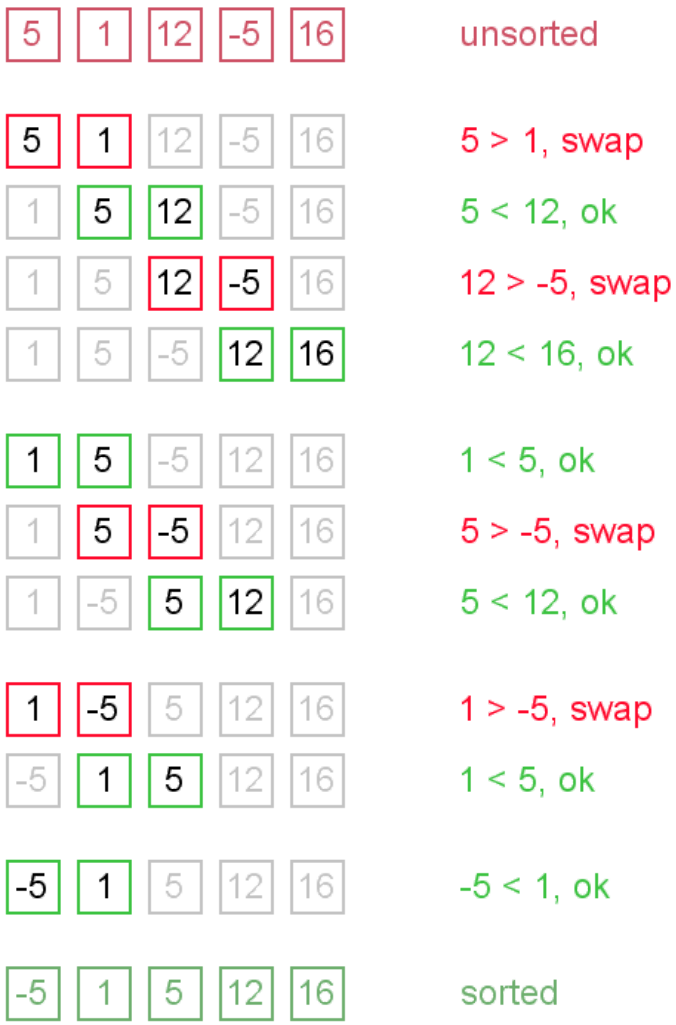
Algorithm:

```
for (k = 0; k < eff_size - 1; k++)

if (x[k] > x[k + 1])
        swaparay(x, k, k + 1);
```

Example:

Example. Sort {5, 1, 12, -5, 16} using bubble sort.



Complexity of Bubble Sort:

The complexity of any sorting algorithm depends upon the number of comparisons. In bubble sort, we have seen that there are N–1 passes in total. In the first pass, N–1 comparisons are made to place the highest element in its correct position. Then, in Pass 2, there are N–2 comparisons and the second highest element is placed in its position. Therefore, to compute the complexity of bubble sort, we need to calculate the total number of comparisons. Therefore, the complexity of bubble sort algorithm is $O(n^2)$. It means the time required to execute bubble sort is proportional to n^2, where n is the total number of elements in the array.

2. Insertion sort:

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Algoritm:

Algorithm: INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for i = 1 to N – 1

Step 2: SET TEMP = ARR[i]

Step 3: SET J = i - 1

Step 4: Repeat while TEMP <= ARR[J]

SET ARR[J + 1] = ARR[J]

SET J = J - 1

[END OF INNER LOOP]

Step 5: SET ARR[J + 1] = TEMP

[END OF LOOP]

Step 6: EXIT

Example:

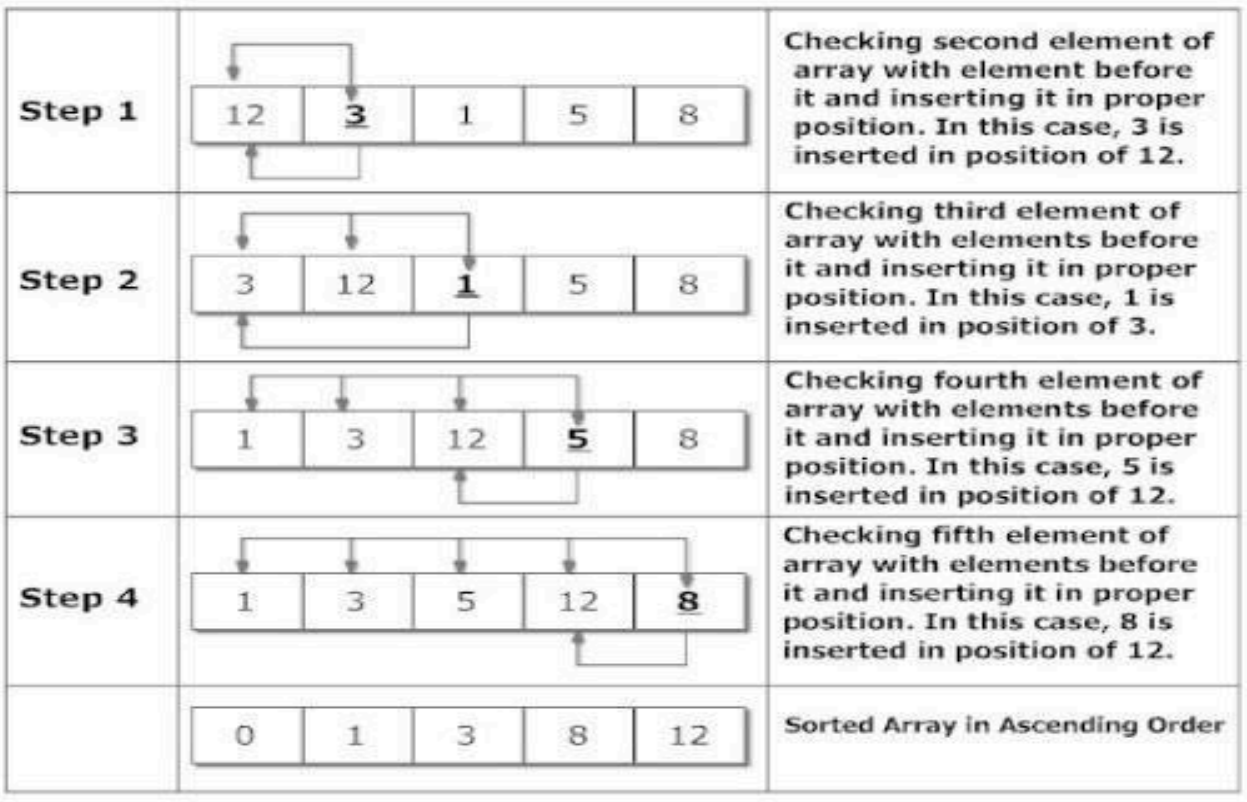


Figure: Sorting Array in Ascending Order Using Insertion Sort Algorithm

Complexity of Insertion Sort:

For insertion sort, the best case occurs when the array is already sorted. In this case, the running time of the algorithm has a linear running time (i.e., O(n)). This is because, during each iteration, the first element from the unsorted set is compared only with the last element of the sorted set of the array.

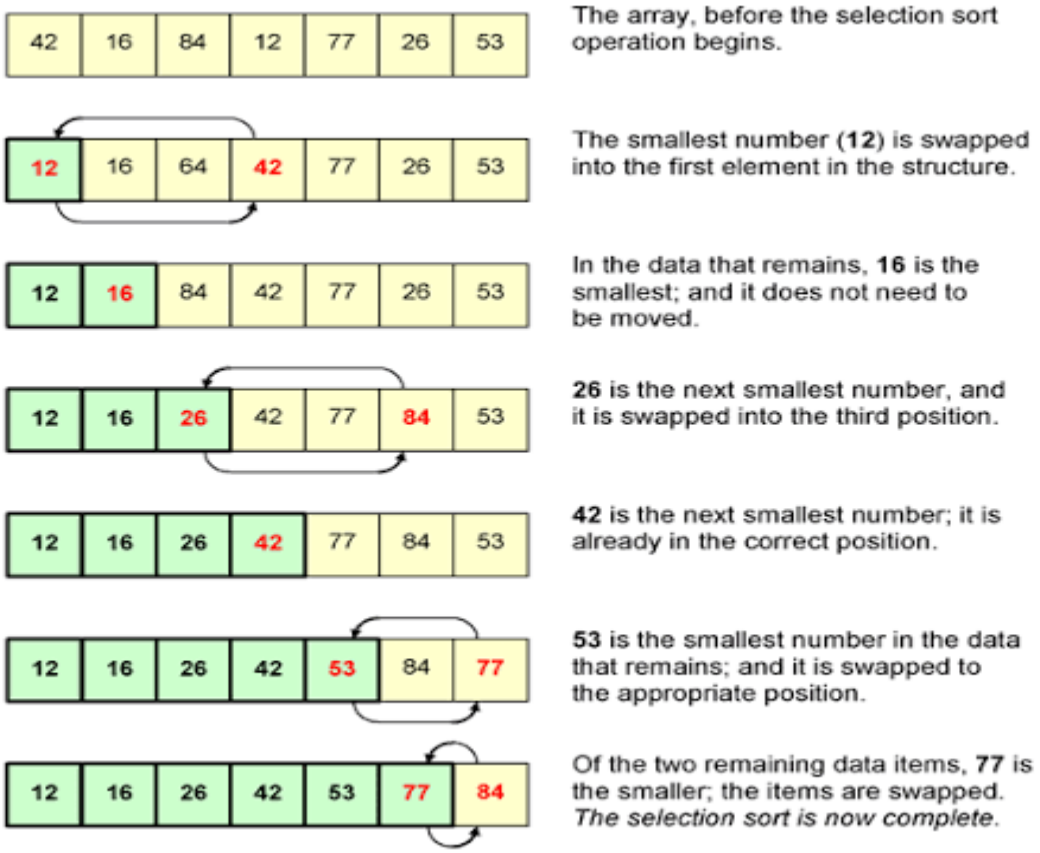
Similarly, the worst case of the insertion sort algorithm occurs when the array is sorted in the reverse order.

In the worst case, the first element of the unsorted set has to be compared with almost every element in the sorted set. Furthermore, every iteration of the inner loop will have to shift the elements of the sorted set of the array before inserting the next element. Therefore, in the worst case, insertion sort has a quadratic running time (i.e., $O(n^2)$).

Even in the average case, the insertion sort algorithm will have to make at least (K–1)/2 comparisons. Thus, the average case also has a quadratic running time.

3. Selection Sort

Selection Sort is a comparison-based sorting algorithm. It sorts by repeatedly selecting the smallest (or largest) element from the unsorted portion and swapping it with the first unsorted element. This process effectively divides the array into a **sorted region** (on the left) and an **unsorted region**



ALGORITHM SelectionSort(A[0..n − 1])

//Sorts a given array by selection sort

//Input: An array A[0..n − 1] of orderable elements

//Output: Array A[0..n − 1] sorted in increasing order

for $i \leftarrow 0$ *to* $n - 2$ **do**

$min \leftarrow i$

for j ← $i + 1$ to $n - 1$ do

if A[j]<A[min] $min \leftarrow j$

swap A[i] and A[min]

The analysis of selection sort is straightforward. The input size is given by the number

of elements n; the basic operation is the key comparison [j] < $A[min]$. The number of times it is executed depends only on the array size and is given by the following sum:

Thus, selection sort is a **$\Theta(n^2)$** algorithm on all inputs.

Note:

- One loop to select an element of Array one by one = O(n)
- Another loop to compare that element with every other Array element = O(n)
- Therefore overall complexity = O(n) * O(n) = O(n*n) = $O(n^2)$

4. Quick Sort:

Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

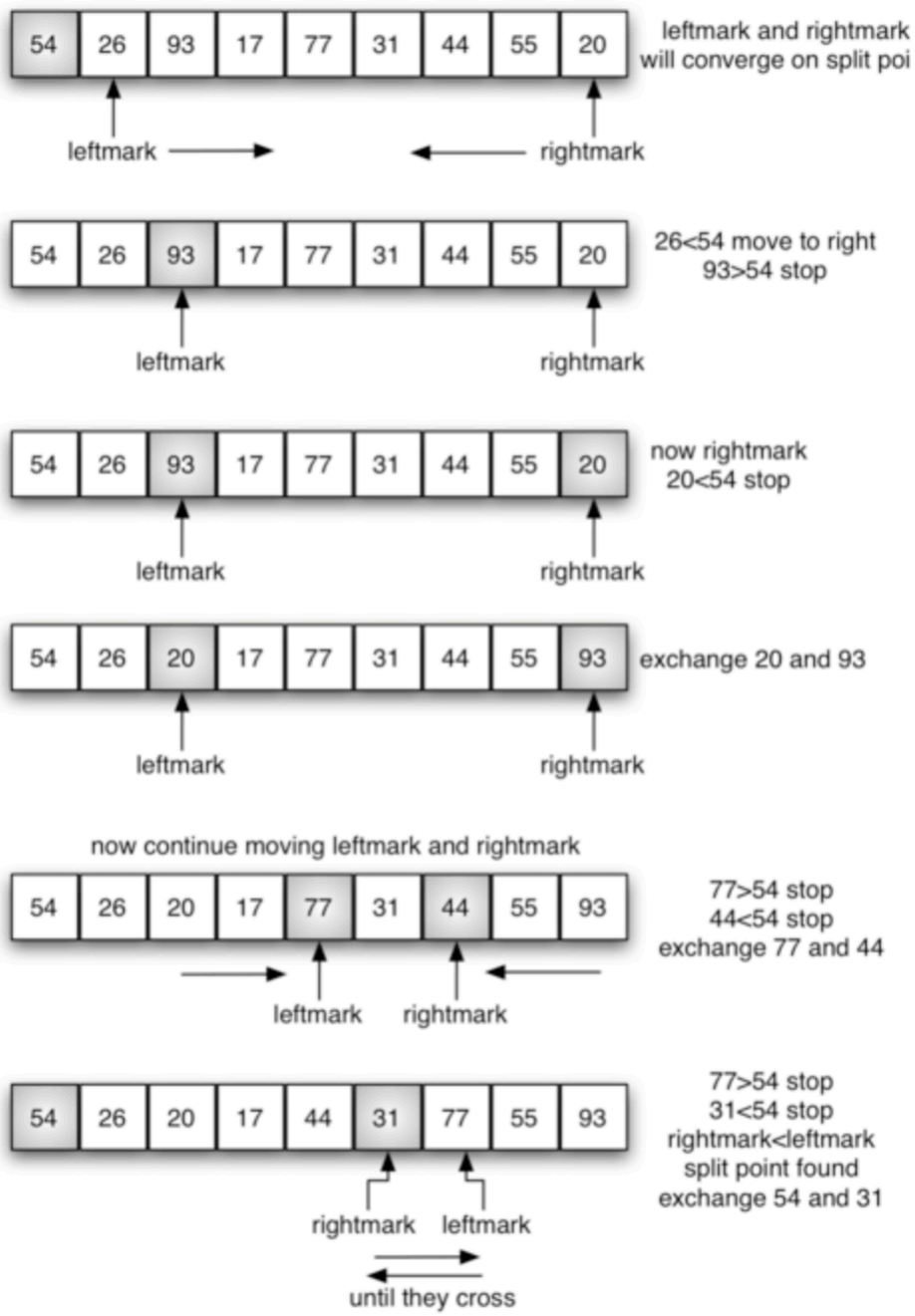
The steps are:

1. Pick an element, called a **pivot**, from the array.
2. Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

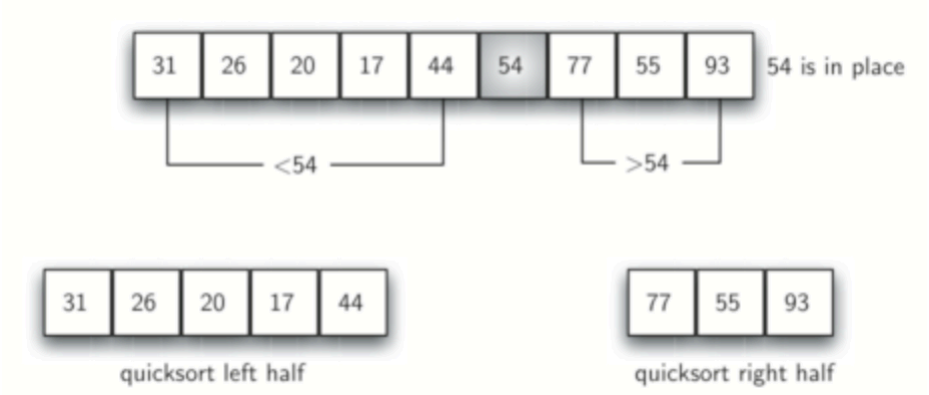
Algorithm:

```
      function quicksort(array)
  if length(array) > 1
      pivot := select any element of array
      left := first index of array
      right := last index of array
      while left ≤ right
          while array[left] < pivot
              left := left + 1
          while array[right] > pivot
              right := right - 1
          if left ≤ right
              swap array[left] with array[right]
       swap array[pivot] with array[right]
      quicksort(array from first index to pivot-1)
      quicksort(array from pivot+1 to last index)
```

Example:



The list can now be divided at the split point and the quick sort can be invoked recursively on the two halves.



Time Complexity:

Best Case: Ω(n log n), Occurs when the pivot element divides the array into two equal halves.

Average Case: θ(n log n), On average, the pivot divides the array into two parts, but not necessarily equal.

Worst Case: $O(n^2)$, Occurs when the smallest or largest element is always chosen as the pivot (e.g., sorted arrays).

5. Merge Sort

Merge Sort is a highly efficient, sorting algorithm based on the **Divide and Conquer** paradigm. It works by recursively breaking down a list into sub-lists until each sub-list consists of a single element, and then merging those sub-lists in a sorted manner.

```
Algorithm: MERGESORT(A, low, high)
{
if(low < high)
{
mid = (low + high) / 2
MERGESORT(A, low, mid)
MERGESORT(A, mid + 1, high)
MERGE(A, low, mid, high)
} }
```

```
MERGE(A, LOW, MID, HIGH)

1. Initialize:
    i ← LOW
    j ← MID + 1
    k ← LOW
2. Repeat while i ≤ MID and j ≤ HIGH
    If A[i] ≤ A[j]
        TEMP[k] ← A[i]
        i ← i + 1
    Else
        TEMP[k] ← A[j]
        j ← j + 1
    End If
    k ← k + 1
3. Copy remaining elements of the left subarray
    While i ≤ MID
        TEMP[k] ← A[i]
        i ← i + 1
```

k ← k + 1

4. Copy remaining elements of the right subarray

While j ≤ HIGH

TEMP[k] ← A[j]

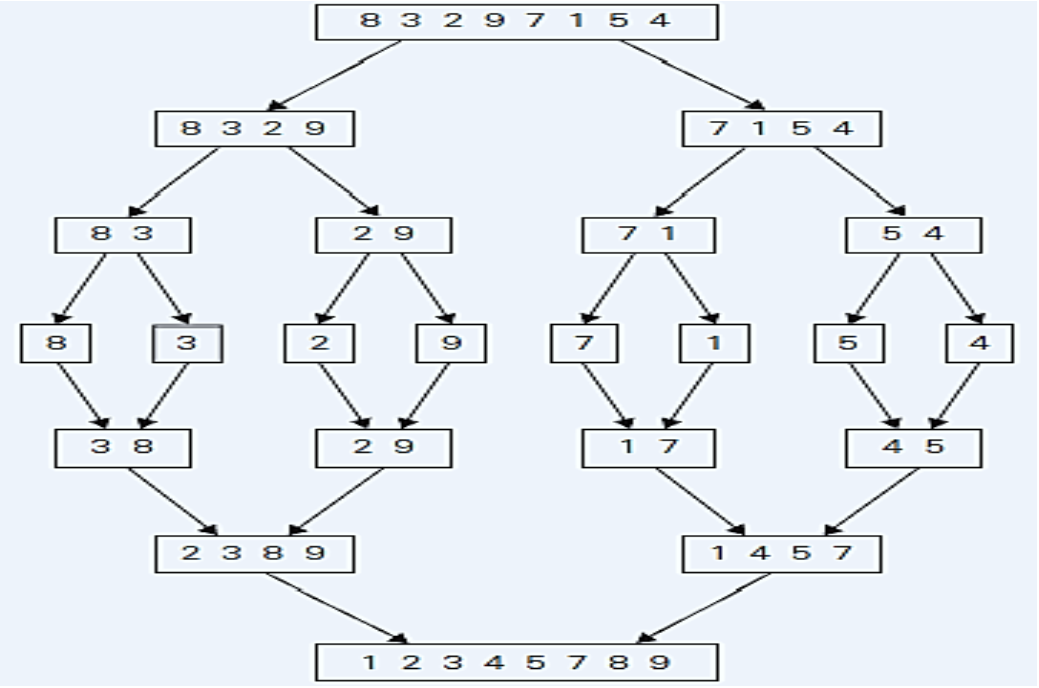
j ← j + 1

k ← k + 1

5. Copy TEMP[LOW…HIGH] back to A[LOW…HIGH]

6. Stop

The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 is illustrated in the Figure



Time Complexity:

- **Best Case:** O(n log n), When the array is already sorted or nearly sorted.
- **Average Case:** O(n log n), When the array is randomly ordered.
- **Worst Case:** O(n log n), When the array is sorted in reverse order.

First, the algorithm can be implemented bottom up by merging pairs of the array's elements, then merging the sorted pairs, and so on. This avoids the time and space overhead of using a stack to handle recursive calls. Second, we can divide a list to be sorted in more than two parts, sort each recursively, and then merge them together. This scheme, which is particularly useful for sorting files residing on secondary memory devices, is called ***multiway mergesort***.