

MODULE 2

ARRAYS

Operations on arrays; Arrays as abstract data types (ADT); Representation of Linear Arrays in memory. Traversing linear arrays; Inserting and deleting elements; Sorting – Selection sort, Bubble sort, Quick sort, Merge sort, Insertion sort; Searching - Sequential Search, Binary search; Iterative and Recursive searching.

Definition of Arrays

An **array** is a **collection of elements of the same data type** stored in **contiguous memory locations**. It allows **multiple values** to be stored under a **single variable name**, with each element accessible by its **index**.

Key Characteristics:

- Fixed size (defined at creation time).
- Elements are stored in consecutive memory blocks.
- All elements must be of the same data type (e.g., all integers or all floats).
- Indexing usually starts at 0.

Syntax (C language):

```
data_type array_name[size];
```

Operations on arrays:

- I. Traversal** – Visit each element.
- II. Insertion** – Add an element at a specific index.
- III. Deletion** – Remove an element from a specific index.
- IV. Searching** – Find an element (linear or binary).
- V. Sorting** – Arrange elements in order (ascending/descending).

1. Traversal

- **Definition:**

Traversal means **visiting each element** of a data structure exactly once to process it (e.g., display it, search for an element, or perform a computation).

- **Example:**

- Traversing an array: visiting elements one by one.
- Traversing a linked list: starting from the head and visiting each node.

- **Importance:**

Needed for operations like searching, updating, or displaying the contents.

- **Example Program (Array Traversal in C):**

```
#include<stdio.h>

int main() {

    int arr[] = {10, 20, 30, 40, 50};

    for(int i = 0; i < 5; i++) {

        printf("%d ", arr[i]);

    }

    return 0;

}
```

Output: 10 20 30 40 50

2. Insertion:

- **Definition:**
Insertion refers to **adding a new element** to a data structure at a specific position.
- **Where?**
 - At the beginning, end, or any random position in arrays, linked lists, etc.
 - Pushing an element onto a stack.
- **Importance:**
Used to update data dynamically.
- **Example Program (Array Insertion: in C):**

```
#include <stdio.h>

int main() {
    int arr[10] = {10, 20, 30, 40, 50};
    int n = 5; // Current number of elements
    int element = 25, position = 2; // Insert 25 at index 2 (3rd position)

    // Shift elements to right
    for (int i = n; i > position; i--) {
        arr[i] = arr[i-1];
    }
    arr[position] = element;
    n++; // Increase size
}
```

Output:

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

return 0;
}
```

javascript

Array after insertion:
10 20 25 30 40 50

3. Deletion

- **Definition:**
Deletion means **removing an existing element** from a data structure.
- **Where?**
 - Deleting an element at a specific index in an array.
 - Deleting a node in a linked list.
 - Popping from a stack.
- **Importance:**
Frees memory and manages space efficiently.
- **Example Program (Array Deletion in C):**

```

#include <stdio.h>

int main() {
    int arr[10] = {10, 20, 30, 40, 50};
    int n = 5;
    int position = 2; // Delete element at index 2 (30)

    // Shift elements to left
    for (int i = position; i < n-1; i++) {
        arr[i] = arr[i+1];
    }
    n--; // Reduce size

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

    return 0;
}

```

Output:

javascript

```

Array after deletion:
10 20 40 50

```

4. Searching

- **Definition:**
Searching is **finding the location** or **existence** of a particular element in a data structure.
- **Types:**
 - **Linear Search:** Check each element one by one.
 - **Binary Search:** Divide and conquer (only on sorted arrays).
- **Importance:**
Used in applications like finding users, records, or files.

Example Program (Array Searching in C):

```
#include <stdio.h>

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int n = 5;
    int key = 30;
    int found = 0;

    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            printf("Element %d found at index %d\n", key, i);
            found = 1;
            break;
        }
    }

    if (!found) {
        printf("Element not found\n");
    }

    return 0;
}
```

Output:

pgsql

Element 30 found at index 2

5. Sorting

- **Definition:**
Sorting arranges elements of a data structure in a **particular order** (ascending or descending).
- **Types of Sorting Algorithms:**
 - **Bubble Sort**
 - **Selection Sort**
 - **Insertion Sort**
 - **Quick Sort**

- **Merge Sort**
- **Importance:**
Sorting improves the efficiency of other operations like searching (especially binary search).
- **Example Program (Array Searching in C):**

```
#include <stdio.h>

int main() {
    int arr[5] = {40, 10, 30, 20, 50};
    int n = 5;
    int temp;

    // Bubble sort
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {

                // Swap
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }

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

    return 0;
}
```

Output:

```
php
```

```
Sorted array:
```

```
10 20 30 40 50
```

Arrays as Abstract Data Types (ADTs)

What is an Abstract Data Type (ADT)?

An **Abstract Data Type (ADT)** is a **logical description** of a way to organize data and a set of operations on that data, without focusing on how they are implemented in memory or in hardware.

Key idea: What operations are supported, not **how** they're implemented.

Why Arrays Are ADTs in C

In C, arrays are considered **Abstract Data Types** because:

Concept	Explanation
Encapsulation	Arrays allow you to store and access multiple values using indices, hiding the actual memory handling logic.
Interface-Oriented	You interact with arrays using <code>arr[i]</code> , without knowing how the address is computed internally (<code>base_address + i * size</code>).
Fixed set of operations	C arrays support operations like access, traversal, and updating values – a defined interface, similar to an ADT.
Logical structure	The array represents a logical data collection even if it's implemented using low-level memory.

Example Program:

```
#include <stdio.h>

int main() {
    int arr[5] = {10, 20, 30, 40, 50};

    printf("Traversing the array:\n");
    for (int i = 0; i < 5; i++) {
        printf("Element at index %d = %d\n", i, arr[i]);
    }
}
```

```
// Update operation
arr[2] = 100;
printf("\nAfter updating index 2:\n");
for (int i = 0; i < 5; i++) {
    printf("Element at index %d = %d\n", i, arr[i]);
}

return 0;
}
```

Output:

```
Traversing the array:
Element at index 0 = 10
Element at index 1 = 20
Element at index 2 = 30
Element at index 3 = 40
Element at index 4 = 50

After updating index 2:
Element at index 0 = 10
Element at index 1 = 20
Element at index 2 = 100
Element at index 3 = 40
Element at index 4 = 50
```

Explanation:

- You are accessing elements using a simple syntax `arr[i]`, hiding how C actually performs **pointer arithmetic** like `*(arr + i)`.
- You don't worry about how the memory is managed internally.
- You're using a **defined interface** (indexing, updating) without knowing the **implementation details** — this is abstraction.
- Hence, **arrays in C behave like an Abstract Data Type**.

Note :

In C, although arrays are implemented using low-level memory constructs, the **user interacts with them abstractly** — through indices and known operations. This separation of **interface and implementation** is what makes arrays an **Abstract Data Type (ADT)**.

Representation of Linear Arrays in memory:

- In C, a linear array, also known as a **one-dimensional array**, is represented in memory as a contiguous block of memory locations.
- Each element of the array is stored sequentially, one after the other, in this block.
- The size of each memory location depends on the data type of the array elements.

For example, if an integer array `int arr[5]` is declared, and an integer typically occupies 4 bytes of memory, then the array will occupy $5 * 4 = 20$ bytes of contiguous memory. The address of the first element `arr[0]` is considered the base address of the array. The address of any element `arr[i]` can be calculated using the formula:

$$\text{address}(\text{arr}[i]) = \text{base_address} + i * \text{element_size}$$

Where:

- `base_address` is the memory address of the first element `arr[0]`.
- `i` is the index of the element.
- `element_size` is the size of each element in bytes (e.g., 4 bytes for an integer).

```
#include <stdio.h>

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int i;

    printf("Array elements and their addresses:\n");
    for (i = 0; i < 5; i++) {
        printf("arr[%d] = %d, Address: %p\n", i, arr[i], (void*)&arr[i]);
    }
    return 0;
}
```

This code snippet demonstrates how array elements are stored contiguously in memory. When the program is executed, it will print the value of each element along with its memory address. The difference between the addresses of consecutive elements will be equal to the size of an integer (usually 4 bytes), confirming the contiguous memory allocation.

Explanation:

int arr[5] = {10, 20, 30, 40, 50};

Declares an array of 5 integers and initializes it with values.

Loop (for (i = 0; i < 5; i++))

Iterates over each index of the array.

printf(...)

Prints:

The index (arr[%d])

The value stored at that index (= %d)

The memory address of the element (Address: %p)

Note: (void*)&arr[i] is used to cast the address to void* which is the proper type for %p.

Output:

Array elements and their addresses:

arr[0] = 10, Address: 0x7ffee9c30180

arr[1] = 20, Address: 0x7ffee9c30184

arr[2] = 30, Address: 0x7ffee9c30188

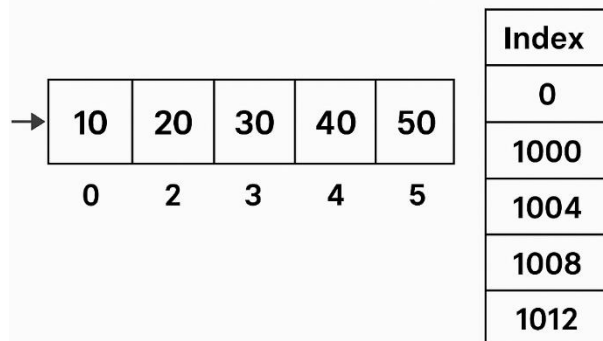
arr[3] = 40, Address: 0x7ffee9c3018c

arr[4] = 50, Address: 0x7ffee9c30190

Note:

- Each address is 4 bytes apart (assuming int is 4 bytes on your system).
- This helps demonstrate how arrays are stored in **contiguous memory**.
- The actual addresses will vary based on your system's memory layout and stack frame.

Linear Arrays in Memory in C



Iterative Searching

Iterative searching involves using loops (such as for or while) to traverse the data structure until the target element is found or the entire structure has been examined.

```
#include <stdio.h>

// Function to perform iterative linear search
int iterative_linear_search(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i; // Return the index if found
        }
    }
    return -1; // Return -1 if not found
}
```

```
int main() {
    int data[] = {34, 78, 12, 9, 55, 67, 100};
    int size = sizeof(data) / sizeof(data[0]);
    int target = 55;

    int result = iterative_linear_search(data, size, target);

    if (result != -1) {
        printf("Element %d found at index %d.\n", target, result);
    } else {
        printf("Element %d not found in the array.\n", target);
    }

    return 0;
}
```

Output:

pgsql

Element 55 found at index 4.

Recursive Searching

Recursive searching, on the other hand, involves defining a function that calls itself with a smaller subset of the data until a base case is met (e.g., the target is found or the subset is empty).

```
// Recursive binary search function
int recursive_binary_search(int arr[], int low, int high, int target) {
    if (low > high) {
        return -1; // Base case: not found
    }
    int mid = low + (high - low) / 2;
    if (arr[mid] == target) {
        return mid; // Base case: found
    } else if (arr[mid] < target) {
        return recursive_binary_search(arr, mid + 1, high, target); // Search right half
    } else {
        return recursive_binary_search(arr, low, mid - 1, target); // Search left half
    }
}
```

```
int main() {
    int data[] = {3, 8, 15, 23, 42, 56, 78, 91}; // Sorted array
    int size = sizeof(data) / sizeof(data[0]);
    int target = 42;

    int result = recursive_binary_search(data, 0, size - 1, target);

    if (result != -1) {
        printf("Element %d found at index %d.\n", target, result);
    } else {
        printf("Element %d not found in the array.\n", target);
    }

    return 0;
}
```

Output:

pgsql

Element 42 found at index 4.

Linear Search Algorithm:

Linear search is also called as **sequential search algorithm**. It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted. The worst-case time complexity of linear search is **$O(n)$** .

The steps used in the implementation of Linear Search are listed as follows -

- First, we have to traverse the array elements using a **for** loop.
- In each iteration of **for loop**, compare the search element with the current array element, and -
 - If the element matches, then return the index of the corresponding array element.
 - If the element does not match, then move to the next element.
- If there is no match or the search element is not present in the given array, return - **1**.

Now, let's see the algorithm of linear search.

Algorithm

```
Linear_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search
```

```
Step 1: set pos = -1
```

```
Step 2: set i = 1
```

```
Step 3: repeat step 4 while i <= n
```

```
Step 4: if a[i] == val
```

```
set pos = i
```

```
print pos
```

```
go to step 6
```

```
[end of if]
```

```
set i = i + 1
```

```
[end of loop]
```

```
Step 5: if pos = -1
```

```
print "value is not present in the array "
```

```
[end of if]
```

```
Step 6: exit
```

Working of Linear search

Now, let's see the working of the linear search Algorithm.

To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.

Let the elements of array are -

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

Let the element to be searched is $K = 41$

Now, start from the first element and compare K with each element of the array.

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 70$

The value of K , i.e., **41**, is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 40$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 30$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 11$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 57$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K = 41$

Now, the element to be searched is found. So algorithm will return the index of the element matched.

Linear Search complexity

Now, let's see the time complexity of linear search in the best case, average case, and worst case. We will also see the space complexity of linear search.

1. Time Complexity

Best Case	$O(1)$
Average Case	$O(n)$
Worst Case	$O(n)$

- **Best Case Complexity** - In Linear search, best case occurs when the element we are finding is at the first position of the array. The best-case time complexity of linear search is **$O(1)$** .
- **Average Case Complexity** - The average case time complexity of linear search is **$O(n)$** .
- **Worst Case Complexity** - In Linear search, the worst case occurs when the element we are looking is present at the end of the array. The worst-case in linear search could be when the target element is not present in the given array, and we have to traverse the entire array. The worst-case time complexity of linear search is **$O(n)$** .

The time complexity of linear search is **$O(n)$** because every element in the array is compared only once.

2. Space Complexity

Space Complexity	$O(1)$
------------------	--------

- The space complexity of linear search is $O(1)$.

Implementation of Linear Search

Now, let's see the programs of linear search in different programming languages.

Program: Write a program to implement linear search in C language.

```
1. #include <stdio.h>
2. int linearSearch(int a[], int n, int val) {
3.     // Going through array sequentially
4.     for (int i = 0; i < n; i++)
5.     {
6.         if (a[i] == val)
```

```

7.     return i+1;
8. }
9. return -1;
10.}
11. int main() {
12.     int a[] = {70, 40, 30, 11, 57, 41, 25, 14, 52}; // given array
13.     int val = 41; // value to be searched
14.     int n = sizeof(a) / sizeof(a[0]); // size of array
15.     int res = linearSearch(a, n, val); // Store result
16.     printf("The elements of the array are - ");
17.     for (int i = 0; i < n; i++)
18.         printf("%d ", a[i]);
19.     printf("\nElement to be searched is - %d", val);
20.     if (res == -1)
21.         printf("\nElement is not present in the array");
22.     else
23.         printf("\nElement is present at %d position of array", res);
24.     return 0;
25.}

```

Output

```

The elements of the array are - 70 40 30 11 57 41 25 14 52
Element to be searched is - 41
Element is present at 6 position of array

```

Binary Search Algorithm

Binary Search is tailored for lists, as its name implies. Therefore, before utilizing this method, ensure your list is properly organized.

This Binary Search approach follows the strategy of "**divide and conquer**." It divides the list into two parts and compares the target item with the Element. If there's a match, congratulations! You've hit the jackpot. If not, the Search proceeds to the right side based on the comparison outcome.

Remember that Binary Search is effective on lists. If your list is unsorted, sorting it beforehand is essential before employing this method

NOTE: Binary search can be implemented on sorted array elements. If the list elements are not arranged in a sorted manner, we have first to sort them.

Now, let's see the algorithm of Binary Search.

Algorithm

1. Binary_Search(a, lower_bound, upper_bound, val) // 'a' is the given array, 'lower_bound' is the index of the first array element, 'upper_bound' is the index of the last array element, 'val' is the value to search


```
Step 1: set beg = lower_bound, end = upper_bound, pos = - 1
Step 2: repeat steps 3 and 4 while beg <= end
Step 3: set mid = (beg + end)/2
Step 4: if a[mid] = val
    set pos = mid
    print pos
    go to step 6
else if a[mid] > val
    set end = mid - 1
else
    set beg = mid + 1
[end of if]
[end of loop]
Step 5: if pos = -1
    print "value is not present in the array"
[end of if]
Step 6: exit
```

Now, let's see the working of algorithm of Binary Search.

So, let's dive into how this Binary Search technique operates. It might not be as complex as it appears.

Picture this scenario; you're dealing with an array named 'a'. You aim to locate a value 'val' within it. The array consists of a starting element (lower_bound) and an ending element (upper_bound).

To begin with, you assign 'beg' to the lower_bound, 'end' to the upper_bound and 'pos' to -1 (simply a temporary marker, for now).

Then, you enter a loop where you keep repeating the following steps until 'beg' is greater than 'end':

1. Calculate the middle index 'mid' by taking the average of 'beg' and 'end'.
2. Check if the value at 'mid' equals 'val'. If it is, you've struck gold! Set 'pos' to 'mid' and print its position. Then, you can break out of the loop.
3. If the value at 'mid' is greater than 'val', it means 'val' might be in the left half of the array. So, you update 'end' to 'mid-1'.
4. If the value at 'mid' is smaller than 'val', it means 'val' could be in the right half. Update 'beg' to 'mid + 1'.

After the loop ends, if 'pos' is still -1, the value you were looking for isn't in the array. Otherwise, you've found its position

Working of Binary search

Now, let's see the working of the Binary Search Algorithm.

To understand the working of the Binary search algorithm, let's take a sorted array. It will be easy to understand the working of Binary search with an example.

There are two methods to implement the binary search algorithm -

- Iterative method
- Recursive method

The recursive method of binary search follows the divide and conquer approach.

Let the elements of array are -

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

Let the element to search is, **K = 56**

We have to use the below formula to calculate the **mid** of the array -

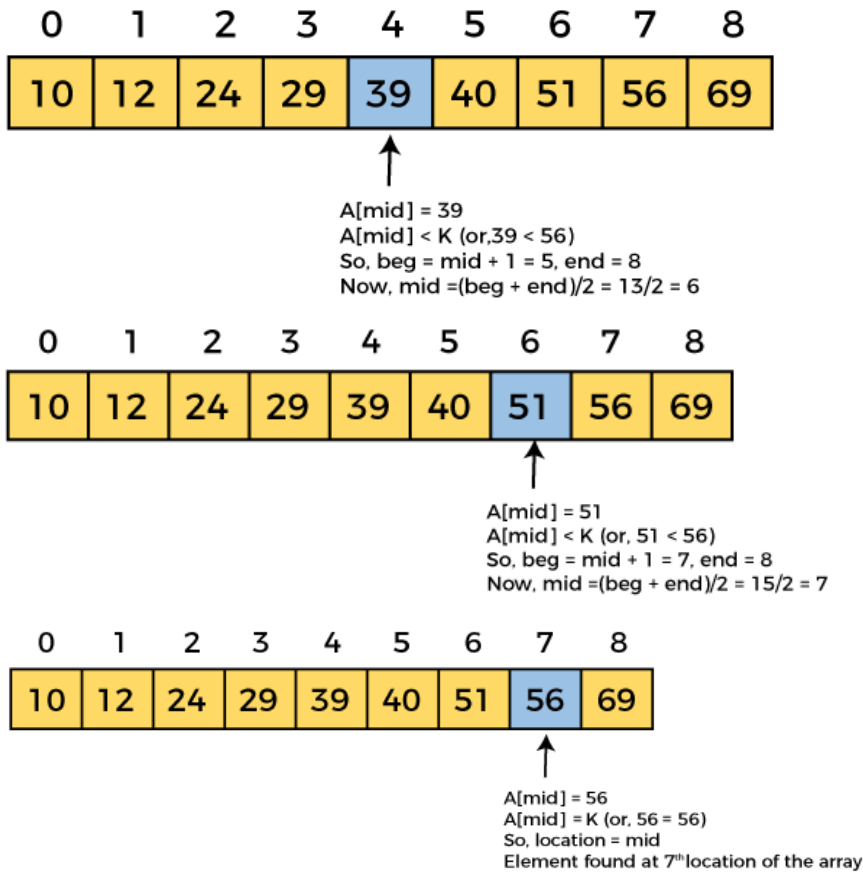
1. **mid** = (beg + end)/2

So, in the given array -

beg = 0

end = 8

mid = $(0 + 8)/2 = 4$. So, 4 is the mid of the array.



Now, the element to search is found. So algorithm will return the index of the element matched.

Binary Search complexity

Now, let's see the time complexity of Binary search in the best case, average case, and worst case. We will also see the space complexity of Binary search.

1. Time Complexity

Best Case	$O(1)$
Average Case	$O(\log n)$
Worst Case	$O(\log n)$

- **Best Case Complexity** - In Binary search, best case occurs when the element to search is found in first comparison, i.e., when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is **$O(1)$** .
- **Average Case Complexity** - The average case time complexity of Binary search is **$O(\log n)$** .

- **Worst Case Complexity** - In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is **$O(\log n)$** .

2. Space Complexity

Space Complexity	$O(1)$
------------------	--------

- The space complexity of binary search is $O(1)$.

Implementation of Binary Search

Now, let's see the programs of Binary search in different programming languages.

Program: Write a program to implement Binary search in C language.

```
1. #include <stdio.h>
2. int binarySearch(int a[], int beg, int end, int val)
3. {
4.     int mid;
5.     if(end >= beg)
6.     {
7.         mid = (beg + end)/2;
8.         /* if the item to be searched is present at middle */
9.         if(a[mid] == val)
10.        {
11.            return mid+1;
12.        }
13.        /* if the item to be searched is smaller than middle, then it can only be in left subarray */
14.        else if(a[mid] < val)
15.        {
16.            return binarySearch(a, mid+1, end, val);
17.        }
18.        /* if the item to be searched is greater than middle, then it can only be in right subarray */
19.        else
20.        {
21.            return binarySearch(a, beg, mid-1, val);
22.        }
23.    }
24.    return -1;
25. }
26. int main() {
27.     int a[] = {11, 14, 25, 30, 40, 41, 52, 57, 70}; // given array
28.     int val = 40; // value to be searched
29.     int n = sizeof(a) / sizeof(a[0]); // size of array
30.     int res = binarySearch(a, 0, n-1, val); // Store result
31.     printf("The elements of the array are - ");
```

```

31. for (int i = 0; i < n; i++)
32. printf("%d ", a[i]);
33. printf("\nElement to be searched is - %d", val);
34. if (res == -1)
35. printf("\nElement is not present in the array");
36. else
37. printf("\nElement is present at %d position of array", res);
38. return 0;
39. }

```

Output

```

The elements of the array are - 11 14 25 30 40 41 52 57 70
Element to be searched is - 40
Element is present at 5 position of array

```

Advantages of Binary Search

1. It's fast; Binary Search is quicker than other searching methods, especially when dealing large lists. The time it takes to find an element grows logarithmically, not linearly, with the size of the list.
2. It's highly efficient. Since Binary Search eliminates half of the remaining elements after each step, it's incredibly efficient and doesn't waste time checking unnecessary elements.
3. It's simple to implement. The algorithm is straightforward to understand, making it a breeze to code and implement in various programming languages.
4. It works like well sorted lists. As long as the list is sorted, Binary Search can find the target element quickly and reliably.
5. It's space-efficient. Unlike other searching algorithms, Binary Search doesn't require additional data structures or memory allocation. It operates directly on the given list, making it space-efficient.
6. It's versatile. Binary Search can be used not only to find a specific element but also to find the first or last occurrence of an element in a sorted list or even to find the closest Element to a target value.
7. It's adaptable. The basic idea behind Binary Search can be extended and adapted to solve various other problems, such as finding the square root of a number or searching in a rotated sorted array.
8. It's reliable and predictable. With Binary Search, you know exactly how many steps will take to find the target element (or determine that it's not present), making it a reliable and predictable algorithm.

Overall, the Binary Search algorithm is a powerful tool combining speed, efficiency, simplicity, and versatility, making it a go-to choice for searching sorted lists across various domains and applications.

Selection Sort Algorithm

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted into its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm.

In this algorithm, the array is divided into two parts; the first is the sorted part, and the other is the unsorted part. Initially, the sorted part of the array is empty, and the unsorted part is the given array. The sorted part is placed at the left, while the unsorted part is placed at the right.

Selection sort is generally used when:

- A small array is to be sorted
- Swapping cost does not matter
- It is compulsory to check all elements

Now, let's see the algorithm of selection sort.

```
// Function to do selection sort on the array 'a' of size 'n'
function selectionSort(a, s)
    // Outer loop: for iterating through the array
    // The last element is not included
    for j = 0 to s - 1
        // Assuming that the current element is the minimum
        // storing its index.
        minIdx = j
        // Inner loop: for finding the minimum element in the unsorted portion
        for k = j + 1 to s - 1
            // If a smaller element is found
            if a[k] < a[minIdx]
                // Updating the index of the element, which is the minimum
                minIdx = k
        // Swapping the minimum element found with the current element
        if minIdx != j
            swap(a[j], a[minIdx])
    // The array is now sorted. return it
    return a
```

Steps of Selection Sort Algorithm

Step 1: Find the smallest element and swap it with the first element of the input array. If the first element is the smallest, then swapping is not required.

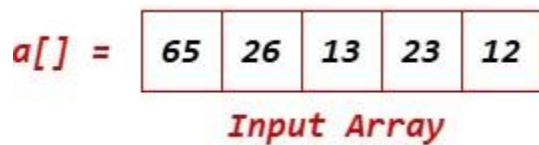
Step 2: Find the smallest element from the remaining elements (the second smallest element) and swap it with the second element. Again, here also swapping is not required if the second smallest element is the second element of the array.

Step 3: Repeat the process for the remaining elements of the input array until all elements achieve the correct position.

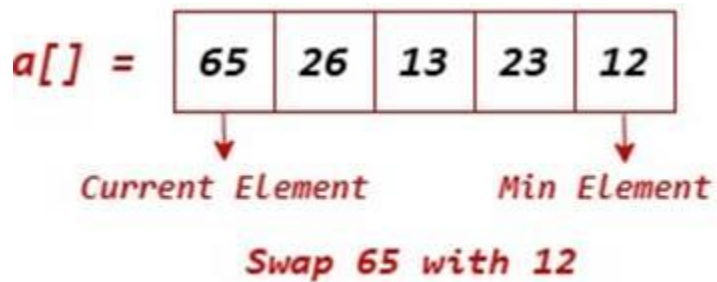
Working of Selection Sort Algorithm

Now, let's see the working of the Selection sort Algorithm.

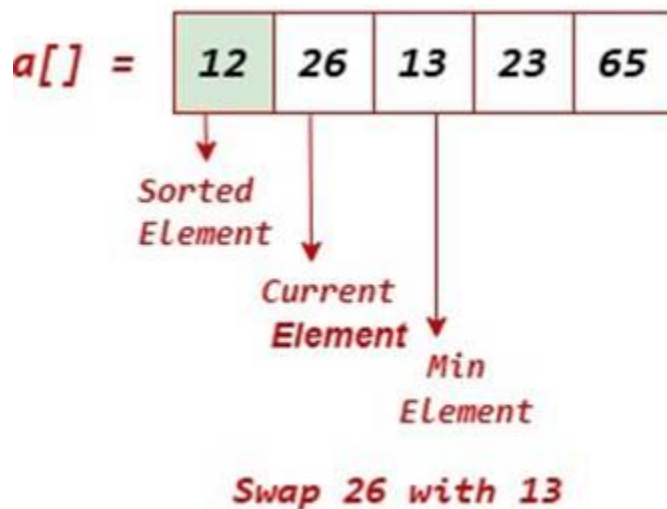
To understand the workings of the Selection sort algorithm, let's take an unsorted array.



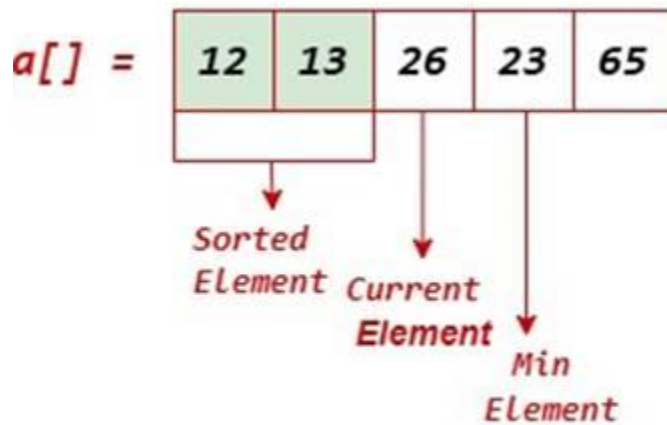
Start traversing the array from left to right. So, we encounter 65 as the current element. Now, find the smallest element from the array. We see 12 as the minimum element.



Now, swap the minimum element with the current element. Thus, the minimum element finds the current position in the sorted array.

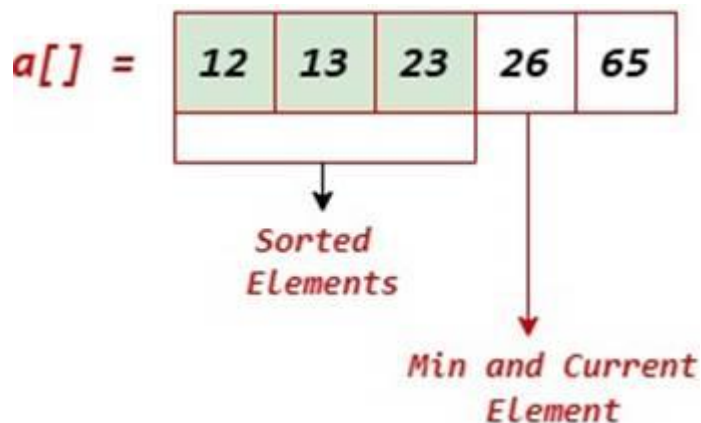


Again, find the next current element, which is 26. Also, look for the smallest element in the remaining elements of the array (the second smallest element overall). We see that the smallest element is 13. Do the swapping of 26 with 13.

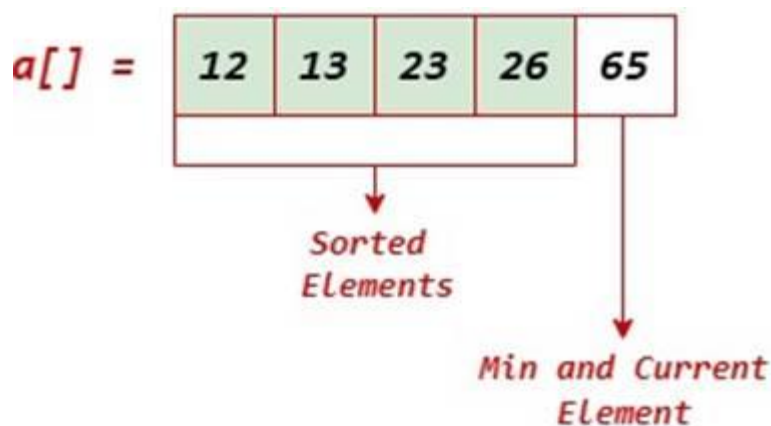


Swap 26 with 23

At this point, the two smallest elements of the array are placed at the correct positions. Now, we do the same process again, i.e., find the minimum element from the remaining and do the swapping with the current element. We find that the minimum element is 23, and the current element is 26. Thus, after swapping, we get the following.



Again, repeat the same process. At this time, we find that the minimum and the current element are the same. Hence, swapping is not required. Thus, we have sorted four elements of the array. Now, we move to the next element.



Here, we also see that the minimum and the current element are the same. Hence, no swapping is required. Thus, we have sorted all the elements of the array.



Implementation of Selection Sort in C

Example

```
1. #include <stdio.h>
2. void selection(int arr[], int n)
3. {
4.     int i, j, small;
5.     for (i = 0; i < n-1; i++) // One by one move boundary of unsorted subarray
6.     {
7.         small = i; //minimum element in unsorted array
8.         for (j = i+1; j < n; j++)
9.             if (arr[j] < arr[small])
10.                small = j;
11.        // Swap the minimum element with the first element
12.        if (small != i)
13.        {
14.            int temp = arr[small];
15.            arr[small] = arr[i];
16.            arr[i] = temp;
17.        }
18.    }
19. }
20. void printArr(int a[], int n) /* function to print the array */
21. {
22.     int i;
23.     for (i = 0; i < n; i++)
24.         printf("%d ", a[i]);
25. }
26. int main()
27. {
28.     int a[] = {65, 26, 13, 23, 12};
29.     int n = sizeof(a) / sizeof(a[0]);
30.     printf("Before sorting array elements are: \n");
31.     printArr(a, n);
32.     selection(a, n);
33.     printf("\nAfter sorting array elements are: \n");
34.     printArr(a, n);
```

```
35. return 0;
36. }
```

Output

Before sorting array elements are:

65 26 13 23 12

After sorting array elements are:

12 13 23 26 65

Complexity Analysis

Now, let's see the time complexity of selection sort in the best case, the average case, and the worst case. We will also see the space complexity of the selection sort.

Time Complexity

Best Case Complexity: It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of the selection sort is **$O(n^2)$** .

Average Case Complexity: It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of the selection sort is **$O(n^2)$** .

Worst Case Complexity: It occurs when the array elements are required to be sorted in reverse order. That means we have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of the selection sort is **$O(n^2)$** .

Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

Space Complexity

The space complexity of the selection sort is **$O(1)$** . It is because, in the selection sort, an extra variable is required for swapping.

Application of Selection Sort Algorithm

1. With the help of this sorting algorithm, we can sort the list of students by their grades or names in a small class.
2. We can also organize the files by their creation date or size in a directory.

3. Also, with the help of this algorithm, we can sort the deck of cards in ascending or descending order.
4. We can also able to arrange the list of items by price in a small e-commerce store.

Advantages of Selection Sort Algorithm

There are some benefits of using selection sort. These are as follows.

1. This sorting technique is very simple and easy to understand and implement.
2. It is also efficient for small data sets or nearly sorted data.
3. There is no need for extra space during the sorting of elements.

Disadvantages of the Selection Sort Algorithm

There are also some limitations to using selection sort. These are as follows.

1. This sorting technique is inefficient for large data sets, with a worst-case time complexity of $O(n^2)$.
2. Selection sort has a lot of comparisons, which can make it slow on modern computers.
3. This is an unstable sorting algorithm. Here, instability means it may not maintain the relative order of equal elements in the input array.

Conclusion

The average and worst-case complexity of the selection sort is $O(n^2)$, where n is the number of items. Due to this, it is not suitable for large data sets. It is an unstable algorithm. Selection sort is easy and simple to implement.

Bubble Sort Algorithm

The **Bubble Sort algorithm** is one of the simplest sorting algorithms in computer science. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the list is sorted. Its name comes from the way smaller elements "bubble" to the top of the list. It is not suitable for large data sets.

Bubble sort is majorly used where:

- Complexity does not matter
- Simple and short code is preferred

Here's a step-by-step explanation:

- **Compare Adjacent Elements:** Start from the first element. Compare it with the next one.
- **Swap if Necessary:** If the current element is greater than the next, swap them.
- **Repeat for All Pairs:** Do this for all adjacent elements in the list. After the first pass, the largest element will "bubble up" to its correct position.
- **Ignore the Last Sorted Elements:** Repeat the process for the remaining unsorted elements, ignoring the last sorted ones.
- **Stop When No Swaps are Made:** If in a complete pass no elements are swapped, the list is sorted.

Algorithm

```
bubbleSort(array)
n = length(array)
repeat
  swapped = false
  for i = 1 to n - 1
    if array[i - 1] > array[i], then
      swap(array[i - 1], array[i])
      swapped = true
    end if
  end for
  n = n - 1
until not swapped
end bubbleSort
```

Steps of Bubble Sort

Step 1: Compare the first two elements of the input array.

Step 2: If the first element is greater than the second element, swap them; otherwise, swapping is not required.

Step 3: According to step 2, do the comparison and swapping of the next pair of elements of the input array (the second and the third element), i.e. if the second element is greater than the third element, do the swapping; otherwise, not.

Step 4: Continue this process till the end of the input array is reached. At this point, the largest element will be "bubble up" and get the end of the list.

Step 5: Repeat the steps from 1 to 4, excluding the last element, as it has already achieved its correct position.

Working of Bubble Sort Algorithm

We have taken the following array as the input array.

a[] =

15	16	11	13	14
----	----	----	----	----

Input Array

First Pass

Compare the first two elements. We see that the first element is less than the second element. Hence, swapping is not required.

$a[] =$

15	16	11	13	14
----	----	----	----	----

Compare 15 with 16

After that, compare the second element with the third element. Observe that the second element is larger than the third element. Hence, swap element 16 with 11.

$a[] =$

15	16	11	13	14
----	----	----	----	----

Compare and swap 16 with 11

After swapping, compare the third element with the fourth element. Observe that the third element is larger than the fourth element. Hence, swap element 16 with 13.

$a[] =$

15	11	16	13	14
----	----	----	----	----

Compare and swap 16 with 13

After swapping, compare the fourth element with the fifth element. We see that the fourth element is larger than the fifth element. Hence, swap element 16 with 14.

$a[] =$

15	11	13	16	14
----	----	----	----	----

Compare and swap 16 with 14

At this point, we have the largest element of the array at its correct position (at the end) in the sorted array.

$a[] =$

15	11	13	14	16
----	----	----	----	----

Sorted Element

Again, compare the first two elements. We see that the first element is larger than the second element. Hence, swap element 15 with 11.

Second Pass

$a[] =$

15	11	13	14	16
----	----	----	----	----

Compare and Swap 15 with 11

After that, compare the second element with the third element. Observe that the second element is larger than the third element. Hence, swap element 15 with 13.

$a[] =$

11	15	13	14	16
----	----	----	----	----

Compare and Swap 15 with 13

After swapping, compare the third element with the fourth element. Observe that the third element is larger than the fourth element. Hence, swap element 15 with 14.

$a[] =$

11	13	15	14	16
----	----	----	----	----

Compare and Swap 15 with 14

At this point, we have got the two elements at their correct position in the sorted array. The elements are the first and the second largest elements of the array.

$a[] =$

11	13	14	15	16
----	----	----	----	----

Sorted Elements

Third Pass

Again, compare the first two elements. We see that the first element is less than the second element. Hence, swapping is not required.

$a[] =$

11	13	14	15	16
----	----	----	----	----

Compare 11 with 13

After that, compare the second element with the third element. Observe that the second element is less than the third element. Hence, swapping is not required.

$a[] =$

11	13	14	15	16
----	----	----	----	----

Compare 13 with 14

Since no swapping is done, further checks are not required as the array is sorted.

$a[] =$

11	13	14	15	16
----	----	----	----	----

Sorted Array

Implementation of Bubble Sort

C Program

```
1. #include<stdio.h>
2. #include<stdbool.h>
3. void print(int a[], int n) //function to print array elements
4. {
5.     int i;
6.     for(i = 0; i < n; i++) {
7.         printf("%d ",a[i]);
8.     }
9. }
10. void bubble(int a[], int n) // function to implement bubble sort
11. {
12.     int i, j, temp;
13.     bool isSwapped = false;
14.     for(i = 0; i < n; i++) {
15.         isSwapped = false;
16.         for(j = 0; j < n - i - 1; j++) {
17.             if(a[j] > a[j + 1]) {
18.                 isSwapped = true;
19.                 temp = a[j];
20.                 a[j] = a[j + 1];
21.                 a[j + 1] = temp;
22.             }
23.         }
24.         if(!isSwapped) {
25.             // no swapping is done. Hence, break the loop
26.             break;
27.         }
28.     }
29. }
30. void main() {
31.     int i, j, temp;
32.     int a[5] = { 15, 16, 11, 13, 14};
33.     int n = sizeof(a)/sizeof(a[0]);
34.     printf("Before sorting array elements are: \n");
35.     print(a, n);
36.     bubble(a, n);
37.     printf("\nAfter sorting array elements are: \n");
38.     print(a, n);
39. }
```

Output:

```
Before sorting array elements are:  
15 16 11 13 14  
After sorting array elements are:  
11 13 14 15 16
```

Complexity Analysis

Let's see the time and space complexity of Bubble sort for the best case, average case, and worst case.

Time Complexity

Best Case Complexity: It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is **$O(n)$** .

Average Case Complexity: It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is **$O(n^2)$** .

Worst Case Complexity: It occurs when the array elements are required to be sorted in reverse order. That means one has to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is **$O(n^2)$** .

Best Case	$O(n)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

Space Complexity

The space complexity of bubble sort is $O(1)$. This is because, in bubble sort, an extra variable is required for swapping.

Applications of Bubble Sort

- **Educational Tool:** Bubble sort is a straightforward algorithm. Hence, it is suitable for teaching the beginners the basics of sorting.

- **Nearly Sorted Data:** Bubble sort performs well when the data is almost sorted, as it can quickly spot and fix minor misplacements.

Advantages of Bubble Sort

- It is easy to implement and understand.
- It does not need any extra memory space.
- It is a stable sorting algorithm. It means that in the sorted output, the elements that share the same key value keep their relative order.

Disadvantages of Bubble Sort

- Bubble sort is slow for large data sets as it has a time complexity of $O(n^2)$.
- Real world applications are none or very limited for the Bubble sort. It is generally used in academics to teach various ways of sorting.

Conclusion

The bubble sort algorithm is a reliable sorting algorithm that is simple to understand. It works by repeatedly comparing and swapping the adjacent elements till the entire array is sorted.

Merge Sort Algorithm

Merge Sort is similar to the Quick Sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient Sorting algorithms. It divides the given list into two halves, calls itself the two halves, and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be split further. Then, we combine the pair of one-element lists into two-element lists, sorting them in the process. The sorted two-element pairs are merged into the four-element lists, and so on, until we get the Sorted list.

Let's see the algorithm of merge Sort.

Steps of Merge Sort

Step 1: Divide the input array into two halves and keep dividing it until further division is not possible.

Step 2: Sort each subarray individually with the help of the Merge Sort algorithm.

Step 3: Merge the sorted subarrays in the sorted order to form a bigger array. Keep doing it until all the elements of both subarrays have been merged.

Working of the Merge Sort Algorithm

To understand the workings of the Merge Sort algorithm, let's take an unsorted array as input.

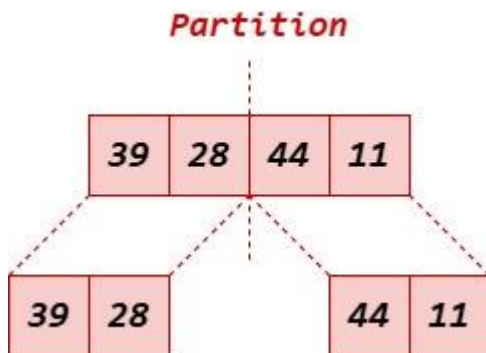
$a[] =$

39	28	44	11
----	----	----	----

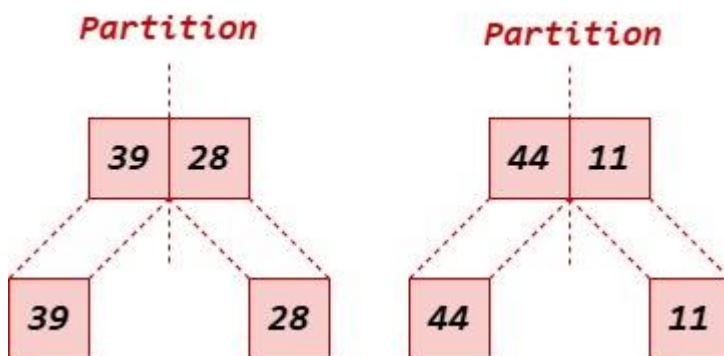
Input Array

Divide

The input array $[39, 28, 44, 11]$ is split into two halves $[39, 28]$ and $[44, 11]$.



Again, we divide the subarray $[39, 28]$ further into two halves $[39]$ and $[28]$. Similarly, we divide the subarray $[44, 11]$ into two halves $[44]$ and $[11]$.

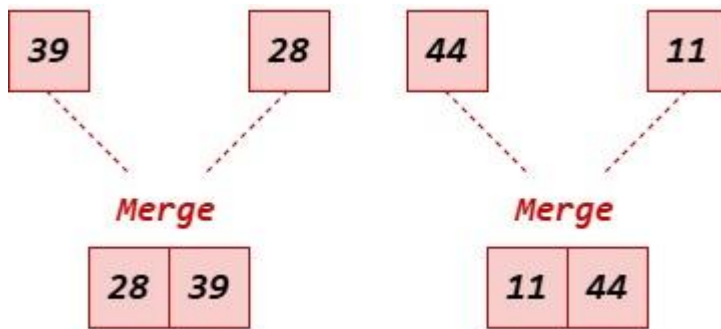


Conquer

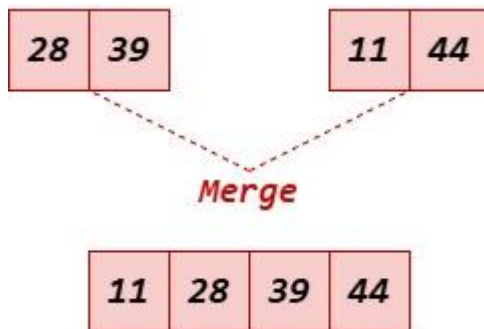
We know that the array with a single element is already sorted. Thus, array elements $[39]$, $[28]$, $[44]$, and $[11]$ are already sorted.

Merge

Start merging the Sorted subarrays in sorted order to form bigger arrays. We merge $[39]$ with $[28]$ to form $[28, 39]$. Similarly, we merge $[44]$ with $[11]$ to form $[11, 44]$.



Now, we merge the subarrays $[28, 39]$ and $[11, 44]$ in the sorted order to get the sorted array $[11, 28, 39, 44]$.



Implementation of Merge Sort in C

Example

```

1. #include <stdio.h>
2. /* Function to merge the subarrays of a[] */
3. void merge(int a[], int beg, int mid, int end)
4. {
5.     int i, j, k;
6.     int n1 = mid - beg + 1;
7.     int n2 = end - mid;
8.     int LeftArray[n1], RightArray[n2]; //temporary arrays
9.     /* copy data to temp arrays */
10.    for (int i = 0; i < n1; i++)
11.        LeftArray[i] = a[beg + i];
12.    for (int j = 0; j < n2; j++)
13.        RightArray[j] = a[mid + 1 + j];
14.    i = 0; /* initial index of first sub-array */
15.    j = 0; /* initial index of second sub-array */
16.    k = beg; /* initial index of merged sub-array */
17.    while (i < n1 && j < n2)
18.    {
19.        if(LeftArray[i] <= RightArray[j])
20.        {
21.            a[k] = LeftArray[i];
22.            i++;
23.        }
24.        else

```

```

25.     {
26.         a[k] = RightArray[j];
27.         j++;
28.     }
29.     k++;
30. }
31. while (i<n1)
32. {
33.     a[k] = LeftArray[i];
34.     i++;
35.     k++;
36. }
37. while (j<n2)
38. {
39.     a[k] = RightArray[j];
40.     j++;
41.     k++;
42. }
43. }
44. void mergeSort(int a[], int beg, int end)
45. {
46.     if (beg < end)
47.     {
48.         int mid = (beg + end) / 2;
49.         mergeSort(a, beg, mid);
50.         mergeSort(a, mid + 1, end);
51.         merge(a, beg, mid, end);
52.     }
53. }
54. /* Function to print the array */
55. void printArray(int a[], int n)
56. {
57.     int i;
58.     for (i = 0; i < n; i++)
59.         printf("%d ", a[i]);
60.     printf("\n");
61. }
62. int main()
63. {
64.     int a[] = {39, 28, 44, 11};
65.     int n = sizeof(a) / sizeof(a[0]);
66.     printf("Before sorting array elements are: \n");
67.     printArray(a, n);
68.     mergeSort(a, 0, n - 1);
69.     printf("After sorting array elements are: \n");
70.     printArray(a, n);
71.     return 0;
72. }

```

Output:

Before sorting array elements are:

39 28 44 11

After sorting array elements are:

11 28 39 44

Complexity Analysis

Time Complexity

Best Case Complexity: It occurs when there is no Sorting required, i.e. the array is already sorted. The best-case time complexity of Merge Sort is **$O(n \cdot \log n)$** .

Average Case Complexity: It occurs when the array elements are in a jumbled order that is neither properly ascending and not properly descending. The average case time complexity of Merge Sort is **$O(n \cdot \log n)$** .

Worst Case Complexity: It occurs when the array elements are required to be sorted in reverse order. That means if we have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of Merge Sort is **$O(n \cdot \log n)$** .

Case	Time Complexity
Best Case	$O(n \cdot \log n)$
Average Case	$O(n \cdot \log n)$
Worst Case	$O(n \cdot \log n)$

Space Complexity

Space Complexity	$O(n)$
Stable	YES

The space complexity of Merge Sort is $O(n)$. It is because, in Merge Sort, an extra variable is required for swapping.

Applications of Merge Sort

- It is used for sorting large datasets.
- It is also used for external Sorting (when the size of the dataset is too large to fit in memory).
- It is used for the [Inversion counting](#).
- In library methods of different programming languages, Merge Sort is used.
- The variation of Merge Sort is [Tim Sort](#), which is used in Java, Android, Python, and Swift. Merge Sort is stable. Therefore, it is preferred to sort non-primitive types.
- [Sort](#) uses Merge Sort.

- It is used for Sorting [Linked lists](#).

Advantages of Merge Sort

- **Stability:** Merge Sort is a stable Sorting algorithm. Hence, it maintains the relative order of elements that are equal in the array.
- **Guaranteed worst-case performance:** Merge Sort has a worst-case time complexity of $O(n \log n)$. Therefore, it performs well even when datasets are large.
- **Easier to implement:** The divide-and-conquer approach is easy to implement.
- **Naturally Parallel:** Since the subarrays are handled independently, it is suitable for parallel processing.

Disadvantages of Merge Sort

- **Space complexity:** Merge Sort takes additional memory for keeping the merged sub-arrays during the process of Sorting.
- **Not in-place:** It is not an in-place Sorting algorithm, which means there is a requirement for additional memory to store the Sorted data. This can be a disadvantage in those applications where the usage of memory is a concern.
- As compared to Quick Sort, Merge Sort is Slower than [Quick Sort](#). This is because Quick Sort is more cache-friendly. After all, it works in-place.

Conclusion

The merge Sort algorithm is known for its efficiency, stability, and predictable performance. These characteristics make Merge Sort a robust choice for various sorting tasks, especially when working with large datasets.

Quick Sort Algorithm

Quick sort is a sorting algorithm that uses the divide and conquer technique. It picks a pivot element and puts it in the appropriate place in the sorted array.

Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

- **Divide:** In divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element, and each element in the right sub-array is larger than the pivot element.
- **Conquer:** Recursively sort two subarrays with Quicksort.
- **Combine:** Combine the already sorted array.

Steps

- **Step 1:** Pick the pivot element. Usually, the first, the last, or the median element is chosen as the pivot element.
- **Step 2:** Do the array partition. For this, rearrange the elements of the array around the pivot. After partitioning, the elements on the left side of the pivot (left sub-array) are less than the pivot element, and the elements on the right side of the pivot (right sub-array) are larger as compared to the pivot element.

- **Step 3:** Repeat the above steps for the left sub-array and the right sub-array. It can be done recursively. The recursion stops when the subarray has only one element left. It is because a single element is already sorted.

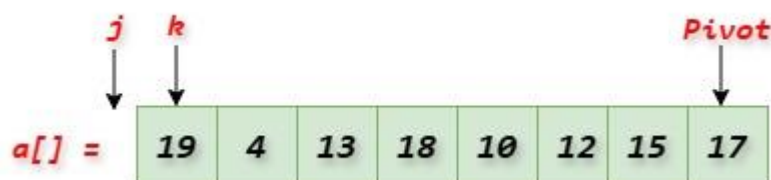
Algorithm

```
function quickSort(arr, low, high):
    if low < high:
        pivotIndex = partition(arr, low, high)
        quickSort(arr, low, pivotIndex - 1) // Sort left half
        quickSort(arr, pivotIndex + 1, high) // Sort right half
function partition(arr, low, high):
    pivot = arr[high] // Choose the last element as the pivot
    i = low - 1 // Pointer for greater element
    for j from low to high - 1:
        if arr[j] ≤ pivot:
            i = i + 1
            swap(arr[i], arr[j])
    swap(arr[i + 1], arr[high]) // Move pivot to its correct position
    return i + 1
```

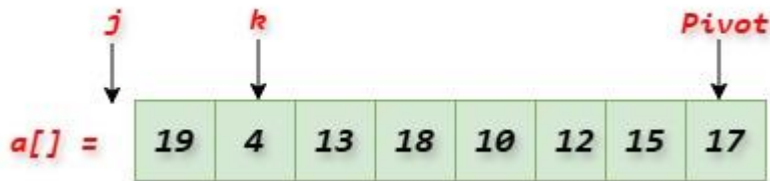
Working of Quick Sort Algorithm

The simple steps of achieving the quick sort are listed as follows.

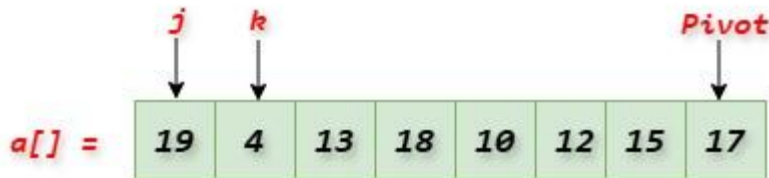
Step 1: Pick the pivot element. Usually, the first, the last, or the median element is chosen as the pivot element. In our case, we have taken the last element as a pivot. Also, take two pointers, j and k , where j has the value -1 , and k points to the 0 th index of the array. Move the j pointer in such a way that the element pointed by the j pointer and the elements left of the j pointer are smaller than the pivot element.



Step 2: Compare $a[k]$, which is 19, with the pivot element. We see that $a[k] > \text{pivot}$ element. Therefore, increment k by 1. Now, k points to the 1^{st} index.



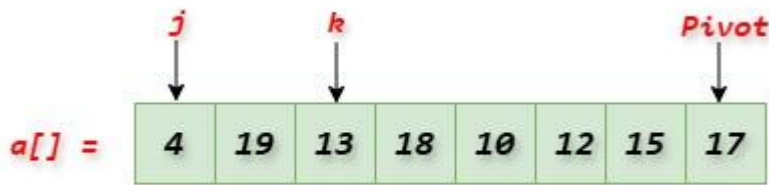
Step 3: Compare $a[k]$, which is 4, with the pivot element. We see that $a[k] < \text{pivot}$ element. Therefore, increment j by 1. Now, j points to the 0th index.



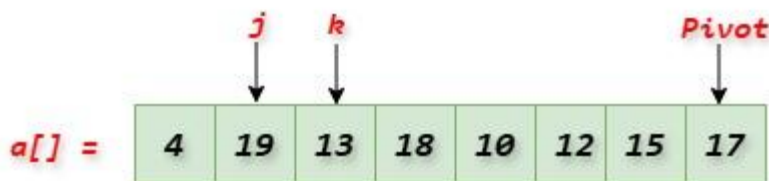
Swap 19 with 4

Step 4: Swap $a[j]$ with $a[k]$,

and increment k by 1. Now, k points to the 2nd index.

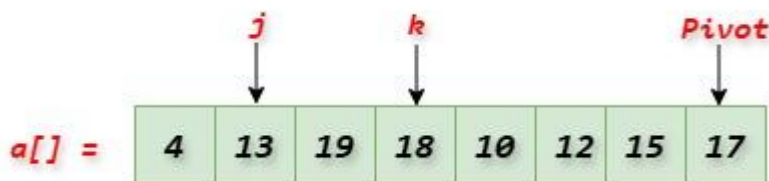


Step 5: Compare $a[k]$, which is 13, with the pivot element. We see that $a[k] < \text{pivot}$ element. Therefore, increment j by 1. Now, j points to the 1st index.

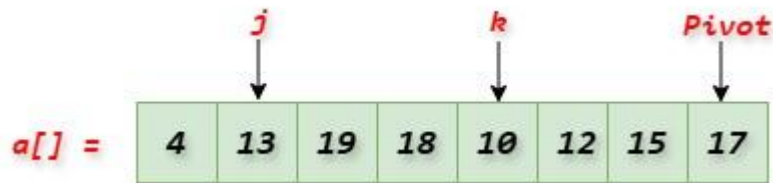


Swap 19 with 13

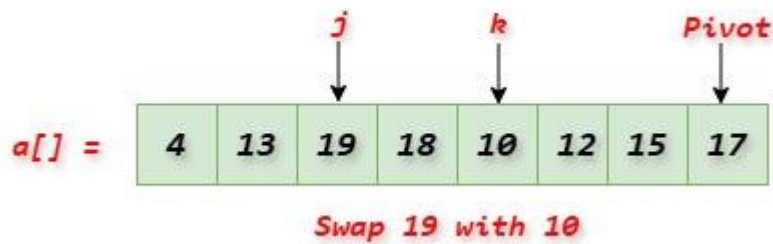
Step 6: Swap $a[j]$ with $a[k]$, and increment k by 1. Now, k points to the 3rd index.



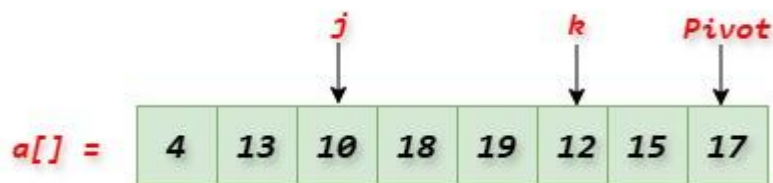
Step 7: Compare $a[k]$, which is 18, with the pivot element. We see that $a[k] > \text{pivot}$ element. Therefore, increment k by 1. Now, k points to the 4th index.



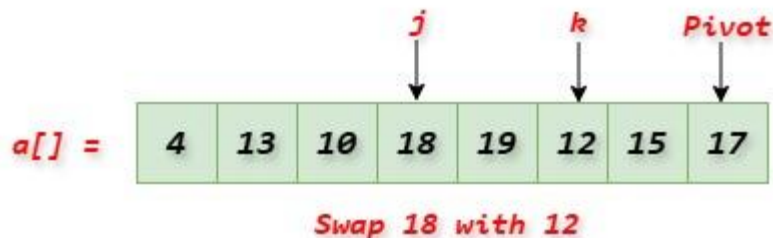
Step 8: Compare $a[k]$, which is 10, with the pivot element. We see that $a[k] < \text{pivot element}$. Therefore, increment j by 1. Now, j points to the 2nd index.



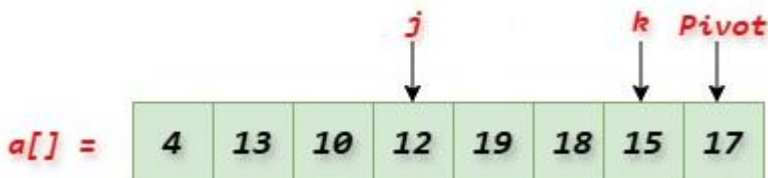
Step 9: Swap $a[j]$ with $a[k]$, and increment k by 1. Now, k points to the 5th index.



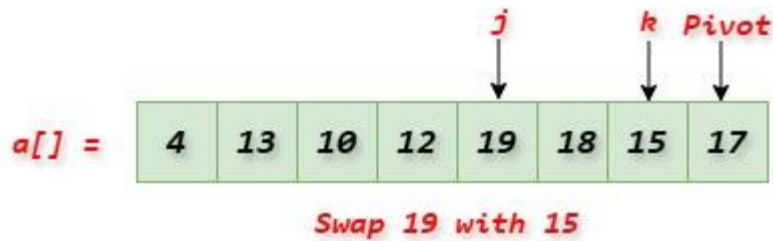
Step 10: Compare $a[k]$, which is 10, with the pivot element. We see that $a[k] < \text{pivot element}$. Therefore, increment j by 1. Now, j points to the 3rd index.



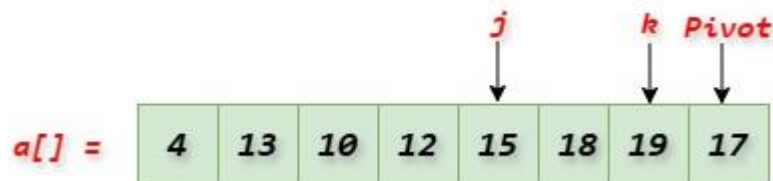
Step 11: Swap $a[j]$ with $a[k]$, and increment k by 1. Now, k points to the 6th index.



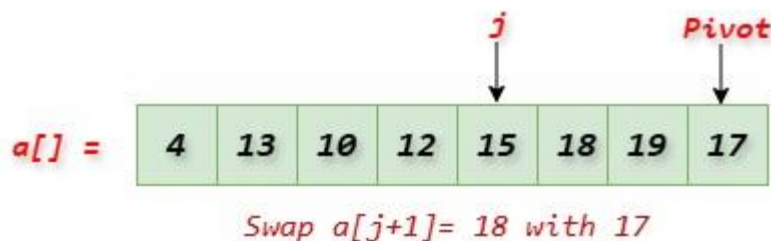
Step 12: Compare $a[k]$, which is 15, with the pivot element. We see that $a[k] < \text{pivot element}$. Therefore, increment j by 1. Now, j points to the 4th index.



Step 13: Swap $a[j]$ with $a[k]$. Since k is pointing to the last element, the further checks and swaps stop here. Observe that all the elements that are left of the j^{th} index and the element at the j^{th} index are smaller than the pivot element.

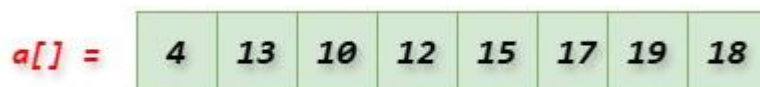


Step 14: Swap $a[j + 1]$ with the pivot element (swap 18 with 17).

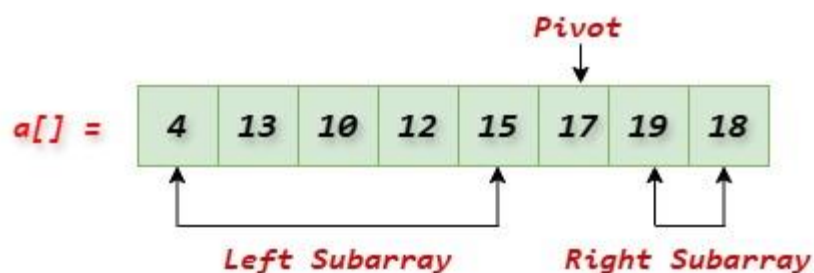


We see that the pivot element is placed at the correct position in the sorted array.

Step 15: Divide the array into left and right subarrays on the basis of the pivot element. The elements that are present on the left of the pivot element are part of the left subarray, and the elements present on the right side of the pivot element are part of the right subarray.



Step 16: Repeat all the steps for the left subarray and the right subarray.



Implementation of Quick Sort

```
1. #include <stdio.h>
2. // function used for swapping elements
3. void swapEle(int *e, int *g) {
4.     int s = *e;
5.     *e = *g;
6.     *g = s;
7. }
8. // function that returns the position of the partition
9. int partition(int a[], int l, int h) {
10.    // picking the rightmost element as the pivot element
11.    int pvt = a[h];
12.    // pointer for larger element
13.    int j = (l - 1);
14.    // traversing each element of the array
15.    // and comparing them with the pivot element
16.    for (int k = l; k < h; k++) {
17.        if (a[k] <= pvt) {
18.            // If the element is less than the pivot is found
19.            // Swap it with the larger element pointed by j
20.            j = j + 1;
21.            // swapping element at j with the element at k
22.            swapEle(&a[j], &a[k]);
23.        }
24.    }
25.    // Swapping the pivot element
26.    // with the larger element located at j + 1
27.    swapEle(&a[j + 1], &a[h]);
28.    // return the point of partition
29.    return (j + 1);
30. }
31. void qckSort(int a[], int l, int h) {
32.    if (l < h) {
33.        // finding the pivot element in such a way that
34.        // elements less than the pivot element are on the left of the pivot
35.        // elements larger than the pivot element are on the right of the pivot
36.        int p = partition(a, l, h);
37.        // recursive call on the left of the pivot
38.        qckSort(a, l, p - 1);
39.        // recursive call on the right of pivot
40.        qckSort(a, p + 1, h);
41.    }
42. }
43. // function to print array elements
44. void displayArray(int a[], int s) {
45.    for (int j = 0; j < s; ++j) {
46.        printf("%d ", a[j]);
47.    }
48.    printf("\n");
```

```

49. }
50. // main function
51. int main() {
52.     int a[] = {10, 7, 8, 9, 1, 5};
53.     int s = sizeof(a) / sizeof(a[0]);
54.     printf("The array before sorting is: \n");
55.     displayArray(a, s);
56.     // perform quicksort on the array
57.     qckSort(a, 0, s - 1);
58.     printf("The array after sorting is: \n");
59.     displayArray(a, s);
60. }

```

Output :

```

The array before sorting is:
10 7 8 9 1 5
The array after sorting is:
1 5 7 8 9 10

```

Complexity Analysis

Best Case	($\Omega(n \log n)$): The best case occurs when the pivot element splits the array into two halves.
Average Case	($\theta(n \log n)$): On average, the pivot element splits the array into two parts, but not necessarily in equal parts.
Worst Case	($O(n^2)$) Occurs when either the largest or smallest element is taken as the pivot element for the ex-arrays that are sorted.

Space Complexity: The space complexity is **$O(n)$** because of the recursive call stack, where n is the total number of elements present in the input array.

Application of Quick Sort

- Numerous government and private entities employ Commercial Computing to organize different types of data, such as arranging files by name/date/price, sorting students by their roll numbers, and organizing account profiles by specified IDs, as it is the quickest general-purpose algorithm for large datasets.
- At those places where stable sort is not needed, one can use quick sort. For stability, merge sort is used.
- To sort arrays in the library function of programming languages, quick sort is used. For example, `Arrays.sort()` in Java, `qsort` in C, and `sort` in C++ use a Hybrid Sorting where QuickSort is the primary sorting.

- Different versions of Quicksort are used to identify the Kth largest or smallest elements.

Advantages of Quick Sort

- The quick sort uses the divide-and-conquer algorithm, which makes it easier to solve problems.
- Quick sort works well on large data sets.
- Overheads are low in quick sort has a low. This is because only a small amount of memory is required to work.
- Quick sort is cache-friendly. It is because the work is done on the same array to sort. It does not copy data from any auxiliary array.
- Quick sort is the fastest general-purpose algorithm for large data where stability is not an issue.
- Quick sort uses tail recursion. Therefore, all of the tail call optimizations can be done.
- One can do parallel processing using quick sort, as two subarrays can be processed independently.

Disadvantages of Quick Sort

- Quick sort has a time complexity of $O(n^2)$, which occurs when the pivot is chosen poorly.
- Quick sort is a bad choice for data sets that are small.
- Quick sort is not a stable sort. It means that the relative order will not be maintained (as a result of a quick sort) of the two elements whose keys are the same. Observe the code where the swapping of elements is done without taking into consideration their original positions.

Conclusion

Due to its speed and simplicity, Quick Sort is one of the most widely used and efficient sorting algorithms. Its in-place sorting capability, combined with the divide-and-conquer approach, makes it suitable for systems with memory restrictions and large datasets.

Merge Sort Algorithm

Merge Sort is similar to the Quick Sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient Sorting algorithms. It divides the given list into two halves, calls itself the two halves, and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be split further. Then, we combine the pair of one-element lists into two-element lists, sorting them in the process. The sorted two-element pairs are merged into the four-element lists, and so on, until we get the Sorted list.

Let's see the algorithm of merge Sort.

Algorithm

In the following algorithm, **arr** is the given array, **beg** is the starting element, and **end** is the last element of the array.

```
1. function mergeSort(arr)
2.   if length(arr) <= 1
```

```

3.   return arr
4.   mid = length(arr) / 2
5.   leftArr = mergeSort(first half of arr)
6.   rightArr = mergeSort(second half of arr)
7.   return mergeArr(leftArr, rightArr)
8. function mergeArr(leftArr, rightArr)
9.   resArr = []
10.  j = 0
11.  k = 0
12.  while j < length(leftArr) and k < length(rightArr)
13.    if leftArr[j] <= rightArr[k]
14.      append leftArr[j] to resArr
15.      j = j + 1
16.    else
17.      append rightArr[k] to resArr
18.      k = k + 1
19.    while j < length(leftArr)
20.      append leftArr[j] to resArr
21.      j = j + 1
22.    while k < length(rightArr)
23.      append rightArr[k] to resArr
24.      k = k + 1
25.  return resArr

```

Steps of Merge Sort

Step 1: Divide the input array into two halves and keep dividing it until further division is not possible.

Step 2: Sort each subarray individually with the help of the Merge Sort algorithm.

Step 3: Merge the sorted subarrays in the sorted order to form a bigger array. Keep doing it until all the elements of both subarrays have been merged.

Working of the Merge Sort Algorithm

To understand the workings of the Merge Sort algorithm, let's take an unsorted array as input.

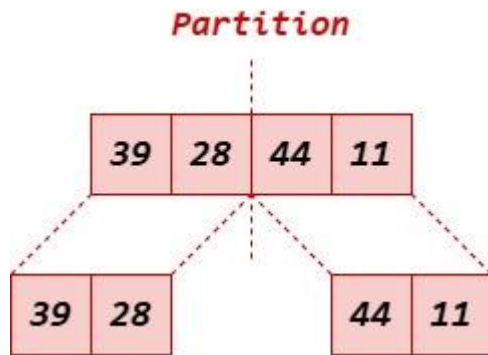
a[] =

39	28	44	11
----	----	----	----

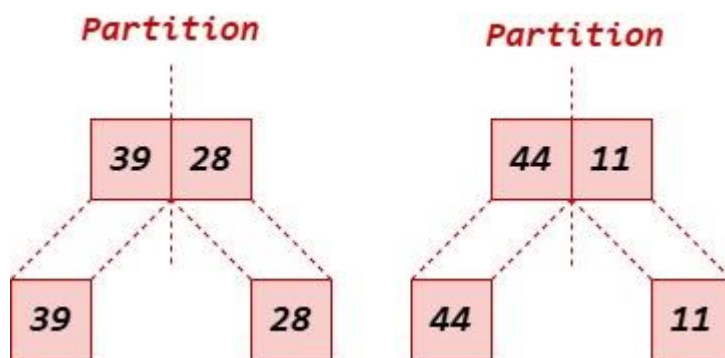
Input Array

Divide

The input array **[39, 28, 44, 11]** is split into two halves **[39, 28]** and **[44, 11]**.



Again, we divide the subarray $[39, 28]$ further into two halves $[39]$ and $[28]$. Similarly, we divide the subarray $[44, 11]$ into two halves $[44]$ and $[11]$.

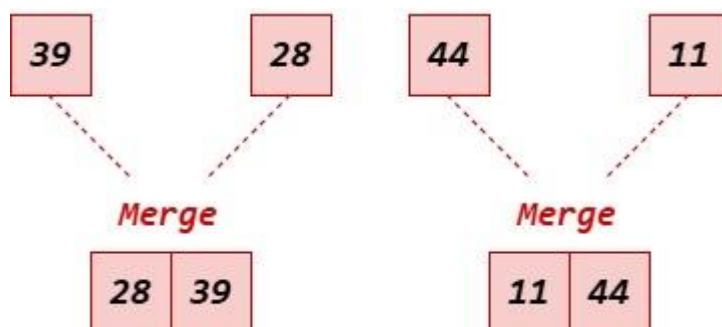


Conquer

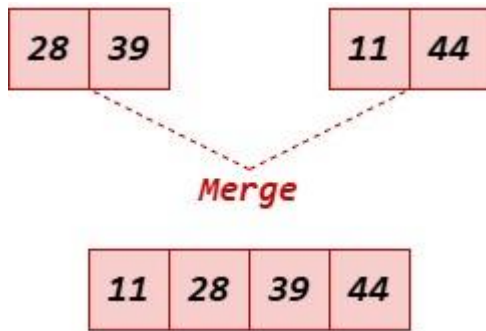
We know that the array with a single element is already sorted. Thus, array elements $[39]$, $[28]$, $[44]$, and $[11]$ are already sorted.

Merge

Start merging the Sorted subarrays in sorted order to form bigger arrays. We merge $[39]$ with $[28]$ to form $[28, 39]$. Similarly, we merge $[44]$ with $[11]$ to form $[11, 44]$.



Now, we merge the subarrays $[28, 39]$ and $[11, 44]$ in the sorted order to get the sorted array $[11, 28, 39, 44]$.



Implementation of Merge Sort in C

Example

```

1. #include <stdio.h>
2. /* Function to merge the subarrays of a[] */
3. void merge(int a[], int beg, int mid, int end)
4. {
5.     int i, j, k;
6.     int n1 = mid - beg + 1;
7.     int n2 = end - mid;
8.     int LeftArray[n1], RightArray[n2]; //temporary arrays
9.     /* copy data to temp arrays */
10.    for (int i = 0; i < n1; i++)
11.        LeftArray[i] = a[beg + i];
12.    for (int j = 0; j < n2; j++)
13.        RightArray[j] = a[mid + 1 + j];
14.    i = 0; /* initial index of first sub-array */
15.    j = 0; /* initial index of second sub-array */
16.    k = beg; /* initial index of merged sub-array */
17.    while (i < n1 && j < n2)
18.    {
19.        if(LeftArray[i] <= RightArray[j])
20.        {
21.            a[k] = LeftArray[i];
22.            i++;
23.        }
24.        else
25.        {
26.            a[k] = RightArray[j];
27.            j++;
28.        }
29.        k++;
30.    }
31.    while (i < n1)
32.    {
33.        a[k] = LeftArray[i];
34.        i++;
35.        k++;

```



```

36. }
37. while (j<n2)
38. {
39.     a[k] = RightArray[j];
40.     j++;
41.     k++;
42. }
43.}
44. void mergeSort(int a[], int beg, int end)
45. {
46.     if (beg < end)
47.     {
48.         int mid = (beg + end) / 2;
49.         mergeSort(a, beg, mid);
50.         mergeSort(a, mid + 1, end);
51.         merge(a, beg, mid, end);
52.     }
53.}
54. /* Function to print the array */
55. void printArray(int a[], int n)
56. {
57.     int i;
58.     for (i = 0; i < n; i++)
59.         printf("%d ", a[i]);
60.     printf("\n");
61.}
62. int main()
63. {
64.     int a[] = {39, 28, 44, 11};
65.     int n = sizeof(a) / sizeof(a[0]);
66.     printf("Before sorting array elements are: \n");
67.     printArray(a, n);
68.     mergeSort(a, 0, n - 1);
69.     printf("After sorting array elements are: \n");
70.     printArray(a, n);
71.     return 0;
72.}

```

Output:

```

Before sorting array elements are:
39 28 44 11
After sorting array elements are:
11 28 39 44

```

Insertion Sort Algorithm

Insertion sorting works similarly to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed on the right

side; otherwise, it will be placed on the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first, it takes one element and iterates it through the sorted array. Although it is simple to use, it is not appropriate for large data sets, as the time complexity of insertion sort in the average case and worst case is $O(n^2)$, where n is the number of items. Insertion sort is less efficient than other sorting algorithms like heap sort, quick sort, and merge sort etc. Insertion sort is a stable sorting algorithm.

Let's see the algorithm of insertion sort.

Algorithm

```
1. INSERTION-SORT(A)
2. for i ← 1 to length(A) - 1 do
3.   key ← A[i]
4.   j ← i - 1
5.   while j ≥ 0 and A[j] > key do
6.     A[j + 1] ← A[j]
7.     j ← j - 1
8.   A[j + 1] ← key
```

The simple steps of achieving the insertion sort are listed as follows -

Step 1: If the element is the first element, assume that it is already sorted. Return 1.

Step 2: Pick the next element and store it separately in a **key**.

Step 3: Now, compare the **key** with all elements in the sorted array.

Step 4: If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

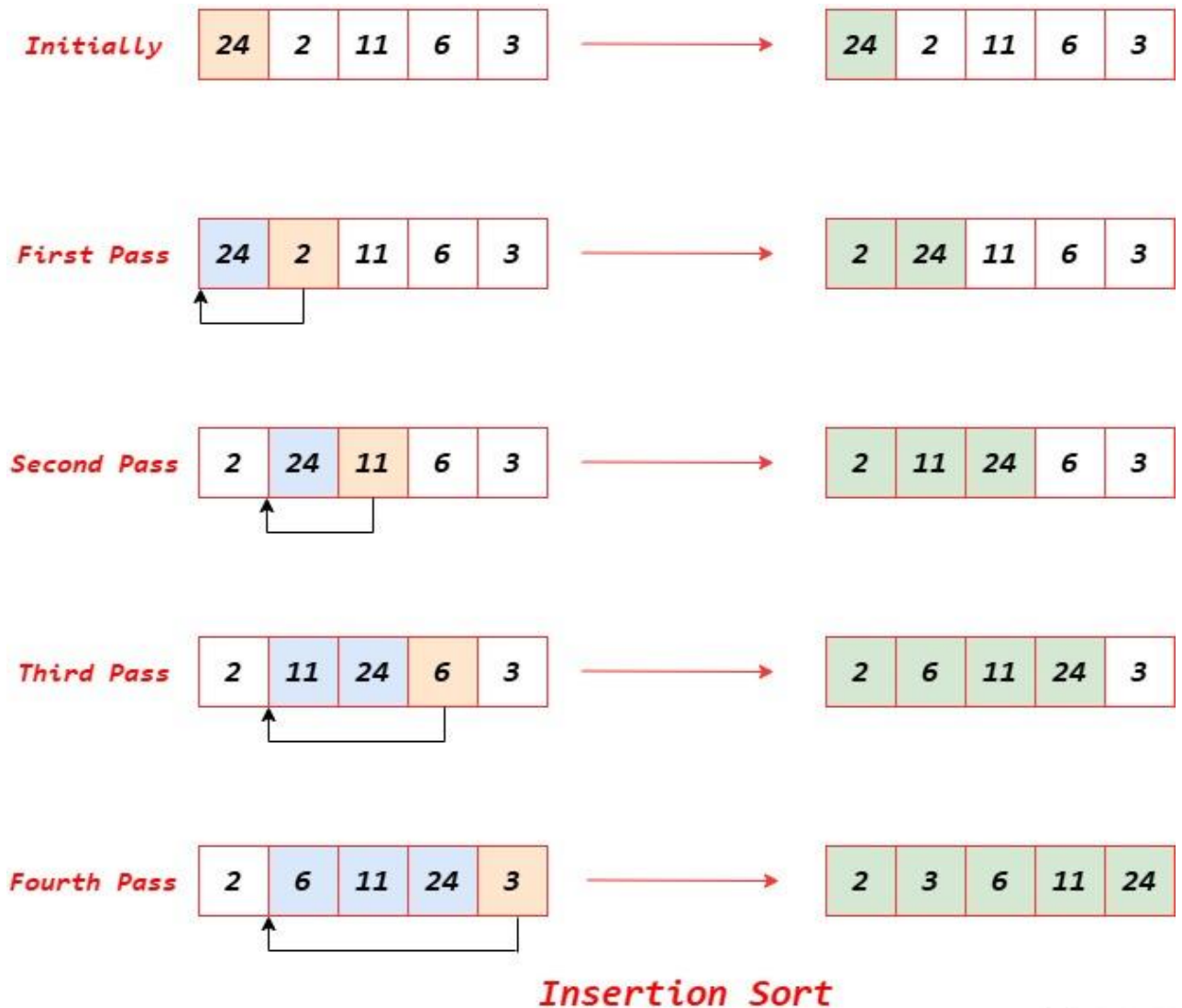
Step 5: Insert the value.

Step 6: Repeat until the array is sorted.

Working of the Insertion Sort Algorithm

Now, let's see the working of the insertion sort Algorithm.

To understand the workings of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.



$a[] = \{24, 2, 11, 6, 3\}$

Initial

We take the first element. The first element of the array is assumed to be sorted. Thus, the sorted part till the 0th index is [24].

First Pass

The current element is 2 (at the 1st index). Compare it with all the elements present on the left side of it. In this case, the element is 24. Since 2 is smaller than 24, insert element 2 before 24. Thus, the sorted part till the 1st index is [2, 24].

Second Pass

Now, the current element is 11 (at the 2nd index). Compare it with all the elements present on the left side of it. In this case, the elements are 2, 24. Since 11 is smaller than 24 but

larger than 2, insert element 11 between 2 and 24. Thus, the sorted part till the 2nd index is [2, 11, 24].

Third Pass

Now, the current element is 6 (at the 3rd index). Compare it with all the elements present on the left side of it. In this case, the elements are 2, 11, and 24. Since 6 is smaller than 11 but larger than 2, insert element 6 between 2 and 11. Thus, the sorted part till the 3rd index is [2, 6, 11, 24].

Fourth Pass

Now, the current element is 3 (at the 4th index). Compare it with all the elements present on the left side of it. In this case, the elements are 2, 6, 11, 24. Since 3 is smaller than 6 but larger than 2, insert element 3 between 2 and 6. Thus, the sorted part till the 4th index is [2, 3, 6, 11, 24].

As you can see here, the array is completely sorted.

Implementation of Insertion Sort

```
1. #include <stdio.h>
2. void insert(int a[], int n) /* function to sort an aay with insertion sort */
3. {
4.     int i, j, temp;
5.     for (i = 1; i < n; i++) {
6.         temp = a[i];
7.         j = i - 1;
8.         while(j >= 0 && temp <= a[j]) /* Move the elements greater than temp to one position
           ahead from their current position */
9.             {
10.                a[j+1] = a[j];
11.                j = j-1;
12.            }
13.        a[j+1] = temp;
14.    }
15.}
16. void printArr(int a[], int n) /* function to print the array */
17. {
18.     int i;
19.     for (i = 0; i < n; i++)
20.         printf("%d ", a[i]);
21. }
22. int main()
23. {
24.     int a[] = {70, 15, 2, 51, 60};
25.     int n = sizeof(a) / sizeof(a[0]);
26.     printf("Before sorting array elements are: \n");
27.     printArr(a, n);
28.     insert(a, n);
```

```
29. printf("\nAfter sorting array elements are: \n");
30. printArr(a, n);
31. return 0;
32. }
```

Output:

Before sorting array elements are:

70 15 2 51 60

After sorting array elements are:

2 15 51 60 70

Complexity Analysis

Let's see the time and space complexity of insertion sort for the best case, average case, and worst case.

Time Complexity

Best Case	$O(n)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

Best Case Complexity: It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is **$O(n)$** .

Average Case Complexity: It occurs when the array elements are in a jumbled order. The average case time complexity of insertion sort is **$O(n^2)$** .

Worst Case Complexity: It occurs when the array elements are required to be sorted in reverse order. Suppose one has to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is **$O(n^2)$** .

Space Complexity

The space complexity of insertion sort is **$O(1)$** . It is because, in insertion sort, an extra variable is required for swapping.

Application of Insertion Sort

Insertion sort comes in handy in situations where:

1. The list is small or nearly sorted.
2. Stability or simplicity is an important factor.
3. Since the Insertion sort is suitable for arrays that are smaller in size, it is used in hybrid sorting algorithms along with some other sorting algorithms like quick sort and Merge

sort. Whenever the subarray size is small, Insertion sort is used in these recursive algorithms. For example, [TimSort](#) and IntroSort use Insertion sort.

Advantages of Insertion Sort

1. Easy and simple to implement.
2. It is a stable sorting algorithm.
3. It is good for nearly sorted lists and small lists.
4. It is an in-place sorting algorithm, making it space-efficient.
5. Adaptive as the number of inversions is directly proportional to the number of swaps. For example, the number of swaps is zero in a sorted array, as the number of inversions is also zero.

Disadvantages of Insertion Sort

1. The Insertion sort is inefficient for large data sets.
2. It does not perform better than other, more advanced sorting algorithms.

Conclusion

Insertion sort is a modest, stable, and adaptive sorting algorithm. It is suitable for small or almost sorted datasets. Worst-case time complexity makes it inefficient for large datasets. As we have discussed that it is an in-place sorting that ensures minimal memory usage.

Overall, insertion sort is valuable for scenarios where simplicity, stability, and handling closely sorted data matter.