Design and Analysis of Algorithm

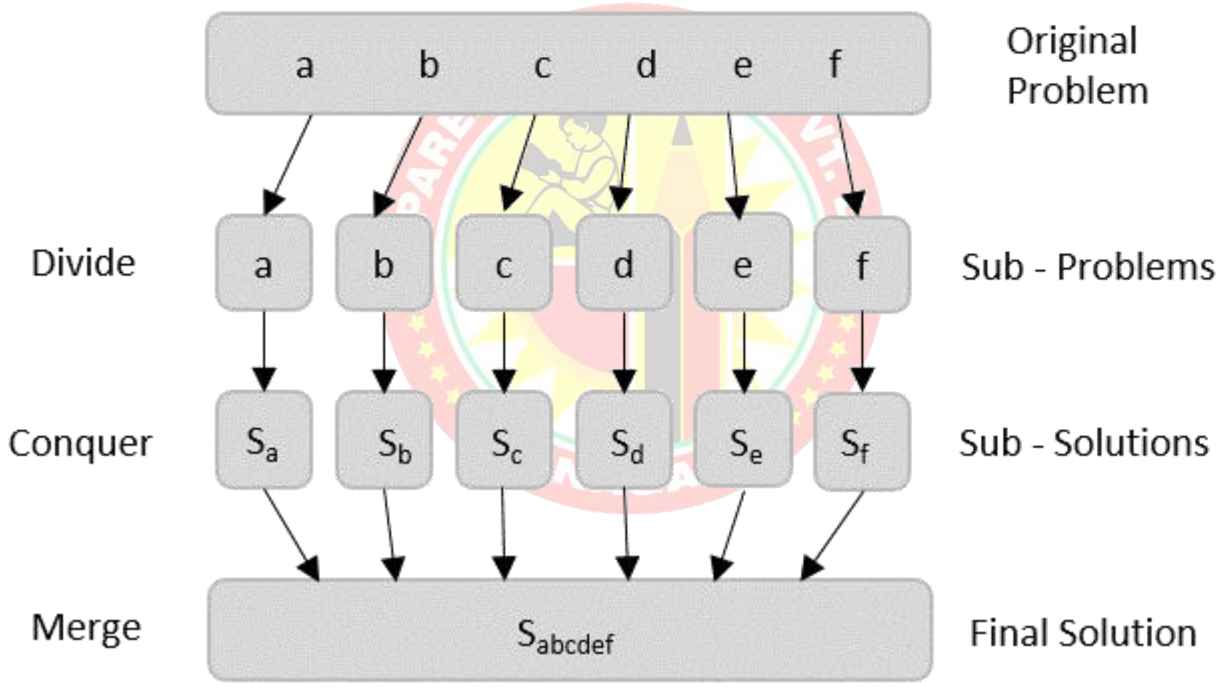
UNIT – 2 : Divide and Conquer Algorithm

Divide and Conquer Introduction

Using divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently.

When we keep dividing the sub-problems into even smaller sub-problems, we may eventually reach a stage where no more division is possible.

Those smallest possible sub-problems are solved using original solution because it takes lesser time to compute. The solution of all sub-problems is finally merged in order to obtain the solution of the original problem.



Broadly, we can understand **divide-and-conquer** approach in a three-step process.

Divide/Break

This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in size but still represent some part of the actual problem.

Conquer/Solve

This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

Merge/Combine

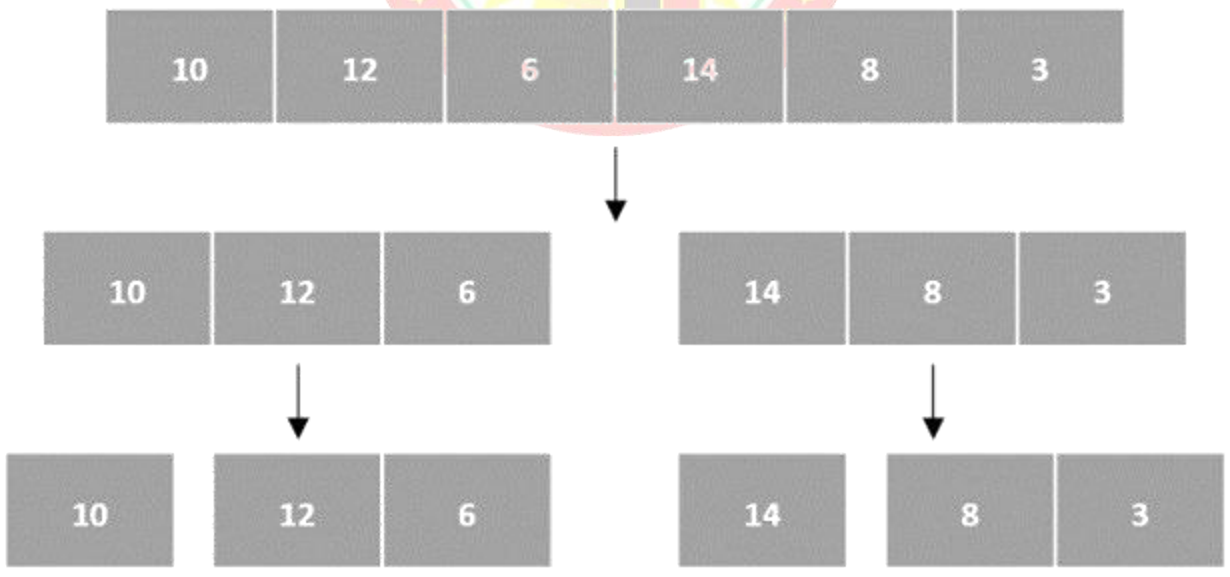
When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

Arrays as Input

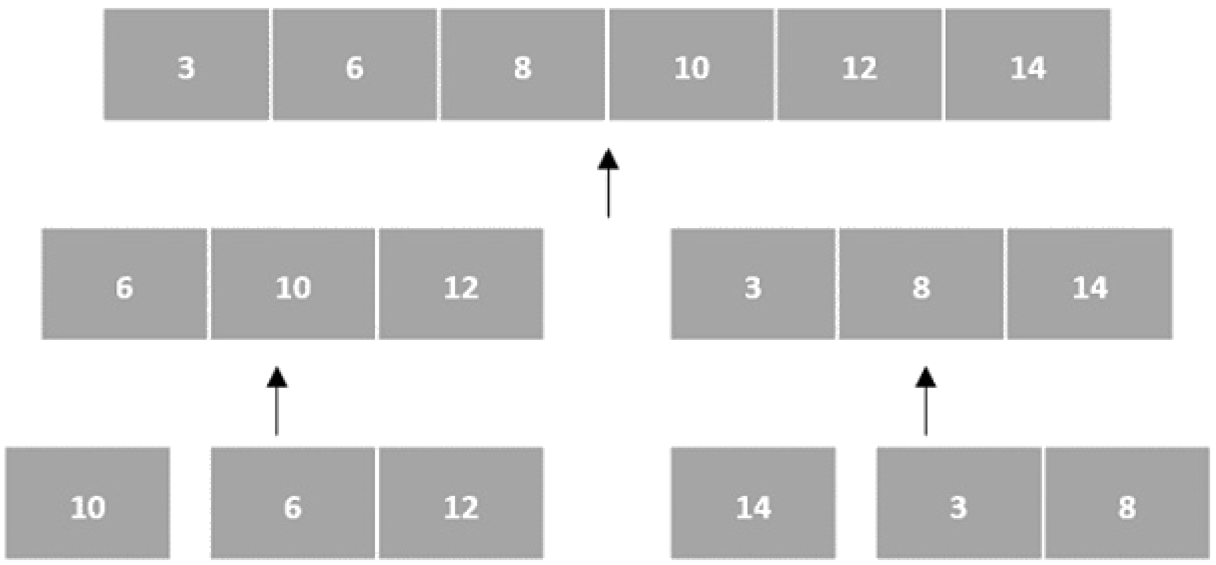
There are various ways in which various algorithms can take input such that they can be solved using the divide and conquer technique. Arrays are one of them.

In algorithms that require input to be in the form of a list, like various sorting algorithms, array data structures are most commonly used.

In the input for a sorting algorithm below, the array input is divided into subproblems until they cannot be divided further.



Then, the subproblems are sorted (the conquer step) and are merged to form the solution of the original array back (the combine step).

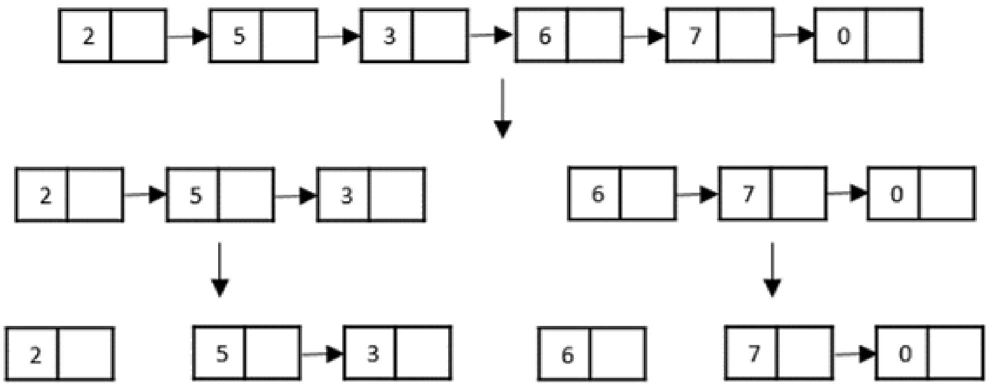


Since arrays are indexed and linear data structures, sorting algorithms most popularly use array data structures to receive input.

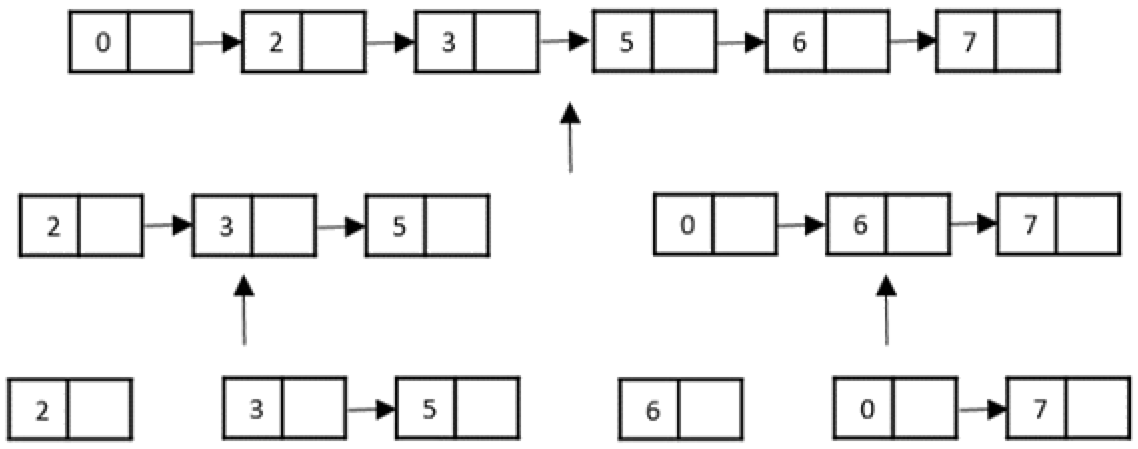
Linked Lists as Input

Another data structure that can be used to take input for divide and conquer algorithms is a linked list (for example, merge sort using linked lists). Like arrays, linked lists are also linear data structures that store data sequentially.

Consider the merge sort algorithm on linked list; following the very popular **tortoise and hare** algorithm, the list is divided until it cannot be divided further.



Then, the nodes in the list are sorted (conquered). These nodes are then combined (or merged) in recursively until the final solution is achieved.



Various searching algorithms can also be performed on the linked list data structures with a slightly different technique as linked lists are not indexed linear data structures. They must be handled using the pointers available in the nodes of the list.

Examples of Divide and Conquer Approach

The following computer algorithms are based on divide-and-conquer programming approach −

- Merge Sort
- Quick Sort
- Binary Search
- Strassen's Matrix Multiplication
- Closest pair (points)
- Karatsuba

There are various ways available to solve any computer problem, but the mentioned are a good example of divide and conquer approach.

Merge Sort:

Merge sort is a sorting technique based on divide and conquer technique. it is one of the most used and approached algorithms.

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

Merge sort keeps on dividing the list into equal halves until it can no more be divided.

By definition, if it is only one element in the list, it is considered sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

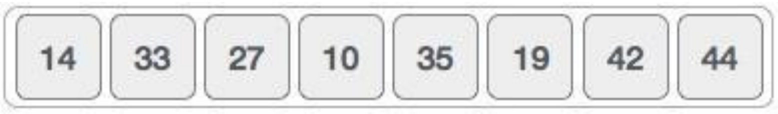
Step 1 − if it is only one element in the list, consider it already sorted, so return.

Step 2 − divide the list recursively into two halves until it can no more be divided.

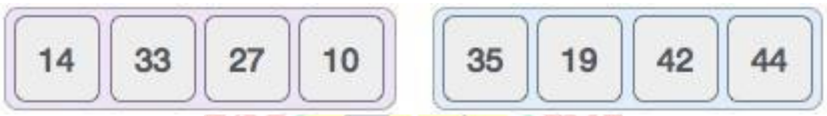
Step 3 − merge the smaller lists into new list in sorted order.

How Merge Sort Works?

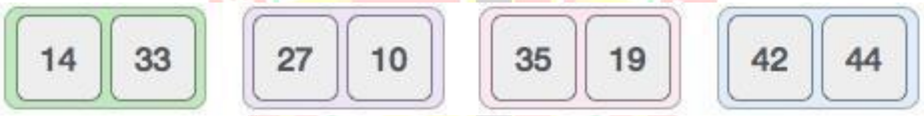
To understand merge sort, we take an unsorted array as the following –



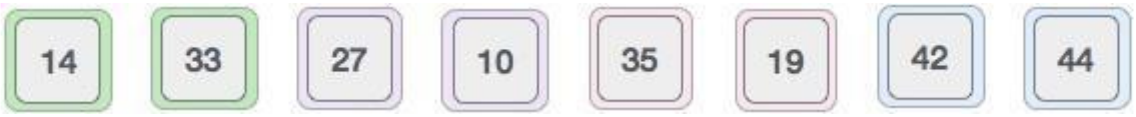
We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

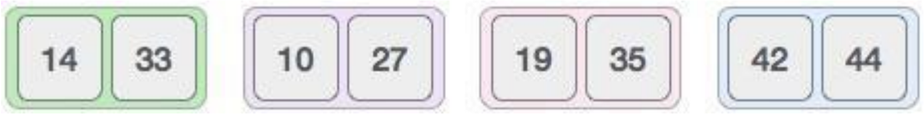


We further divide these arrays and we achieve atomic value which can no more be divided.

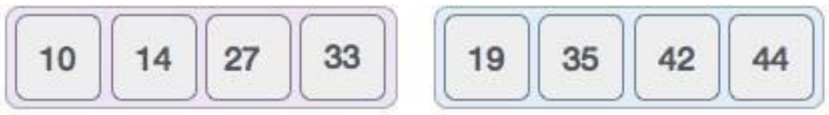


Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

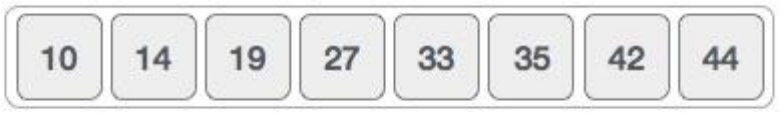
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list becomes sorted and is considered the final solution.



Example:

```
#include <stdio.h>
#define max 10
int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };
int b[10];
void merging(int low, int mid, int high)
{
        int l1, l2, i;
        for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++)
        {
                if(a[l1] <= a[l2])
                        b[i] = a[l1++];
                else
                        b[i] = a[l2++];
        }
        while(l1 <= mid)
                b[i++] = a[l1++];
        while(l2 <= high)
                b[i++] = a[l2++];
        for(i = low; i <= high; i++)
                a[i] = b[i];
}
void sort(int low, int high)
{
        int mid;
        if(low < high)
        {
                mid = (low + high) / 2;
                sort(low, mid);
                sort(mid+1, high);
                merging(low, mid, high);
        }
        else
        {
                return;
        }
}
int main()
{
        int i;
        printf("Array before sorting\n");
        for(i = 0; i <= max; i++)
                printf("%d ", a[i]);
        sort(0, max);
        printf("\nArray after sorting\n");
        for(i = 0; i <= max; i++)
                printf("%d ", a[i]);
}
```

Output:

```
Array before sorting
10 14 19 26 27 31 33 35 42 44 0
Array after sorting
0 10 14 19 26 27 31 33 35 42 44
```

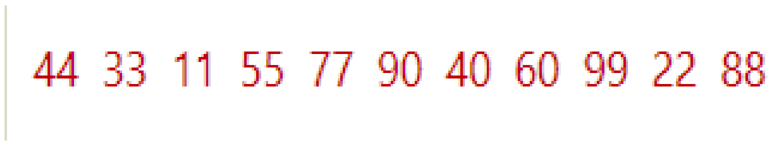
Quick Sort:

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays.

How Quick Sort Works?

To understand quick sort, we take an unsorted array as the following –



44 33 11 55 77 90 40 60 99 22 88

Let 44 be the Pivot element and scanning done from right to left

Comparing 44 to the right-side elements, and if right-side elements are smaller than 44, then swap it. As 22 is smaller than 44 so swap them.

22	33	11	55	77	90	40	60	99	**44**	88

Now comparing 44 to the left side element and the element must be greater than 44 then swap them. As 55 are greater than 44 so swap them.

22	33	11	**44**	77	90	40	60	99	**55**	88

Recursively, repeating steps 1 & steps 2 until we get two lists one left from pivot element 44 & one right from pivot element.

Swap with 77

22	33	11	**44**	77	90	40	60	99	**55**	88

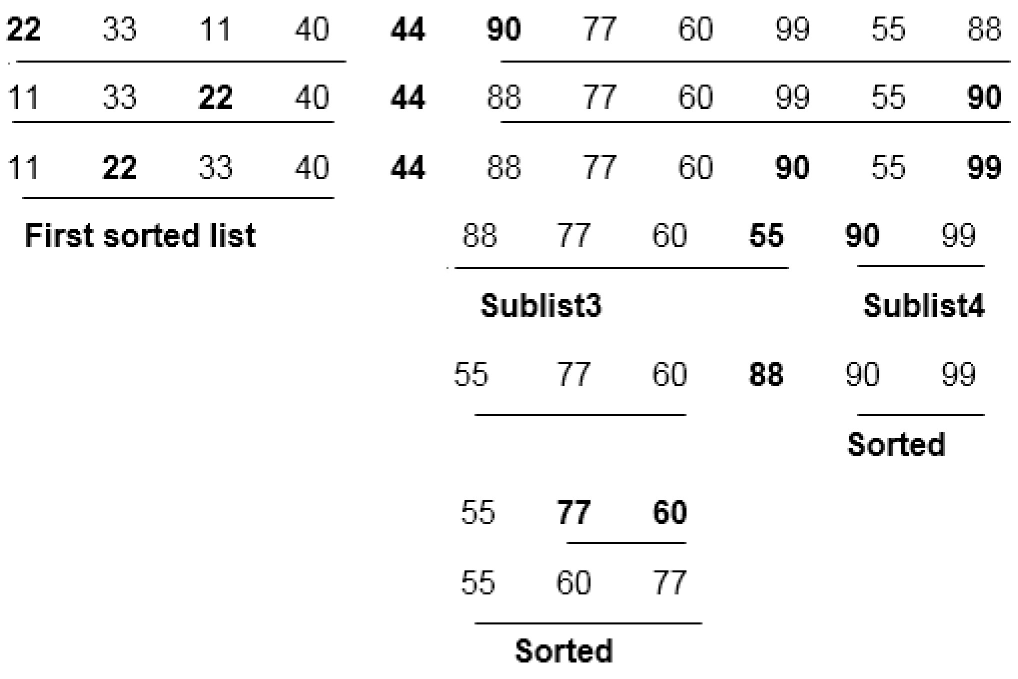
Now, the element on the right side and left side are greater than and smaller than 44 respectively.

Now we get two sorted lists

These two sorted sublists side by side.

$$\underbrace{22 \quad 33 \quad 11 \quad 40}_{\textbf{Sublist1}} \quad \mathbf{44} \quad \underbrace{90 \quad 77 \quad 66 \quad 99 \quad 55 \quad 88}_{\textbf{Sublist2}}$$

And these sublists are sorted under the same process as above done.
These two sorted sublists side by side.



Now merging sublists

11 22 33 40 44 55 60 77 88 90 99

Example:

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 7
int intArray[MAX] = {4,6,3,2,1,9,7};
void printline(int count)
{
        int i;
        for (i = 0; i < count - 1; i++)
        {
                Printf("=");
        }
        printf("=\n");
}
void display()
{
        int i;
        printf("[");
        // navigate through all items
        for (i = 0; i < MAX; i++)
        {
                printf("%d ", intArray[i]);
        }
        printf("]\n");
}
void swap(int num1, int num2)
{
        int temp = intArray[num1];
        intArray[num1] = intArray[num2];
        intArray[num2] = temp;
}
int partition(int left, int right, int pivot)
{
        int leftPointer = left - 1;
        int rightPointer = right;
        while (true)
        {
                while (intArray[++leftPointer] < pivot)
                {
                        //do nothing
                }
                while (rightPointer > 0 && intArray[--rightPointer] > pivot)
                {
                        //do nothing
                }
                if (leftPointer >= rightPointer)
                {
                        break;
                }
                else
                {
                        printf(" item swapped :%d,%d\n", intArray[leftPointer],
                        intArray[rightPointer]);
                        swap(leftPointer, rightPointer);
                }
        }
        printf(" pivot swapped :%d,%d\n", intArray[leftPointer], intArray[right]);
        swap(leftPointer, right);
        printf("Updated Array: ");
        display();
        return leftPointer;
}
void quickSort(int left, int right)
{
        if (right - left <= 0)
        {
                return;
        }
        else
        {
                int pivot = intArray[right];
                int partitionPoint = partition(left, right, pivot);
                quickSort(left, partitionPoint - 1);
                quickSort(partitionPoint + 1, right);
        }
}
int main()
{
        printf("Input Array: ");
        display();
        printline(50);
        quickSort(0, MAX - 1);
        printf("Output Array: ");
        display();
        printline(50);
}
```

Output:

```
Input Array: [4 6 3 2 1 9 7 ]
==================================================
 pivot swapped :9,7
Updated Array: [4 6 3 2 1 7 9 ]
 pivot swapped :4,1
Updated Array: [1 6 3 2 4 7 9 ]
 item swapped :6,2
 pivot swapped :6,4
Updated Array: [1 2 3 4 6 7 9 ]
 pivot swapped :3,3
Updated Array: [1 2 3 4 6 7 9 ]
Output Array: [1 2 3 4 6 7 9 ]
==================================================
```

Binary Search Algorithm:

Binary Search algorithm is an interval searching method that performs the searching in intervals only. The input taken by the binary search algorithm must always be in a sorted array since it divides the array into subarrays based on the greater or lower values. The algorithm follows the procedure below −

Step 1 − Select the middle item in the array and compare it with the key value to be searched. If it is matched, return the position of the median.

Step 2 − If it does not match the key value, check if the key value is either greater than or less than the median value.

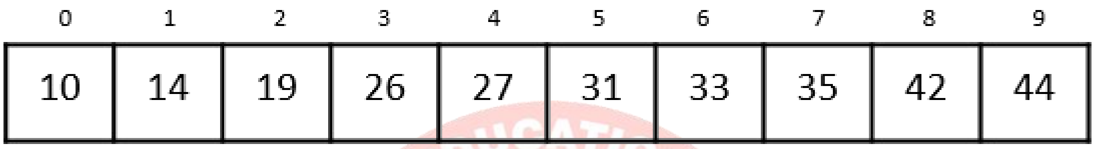
Step 3 − If the key is greater, perform the search in the right sub-array; but if the key is lower than the median value, perform the search in the left sub-array.

Step 4 − Repeat Steps 1, 2 and 3 iteratively, until the size of sub-array becomes 1.

Step 5 − If the key value does not exist in the array, then the algorithm returns an unsuccessful search.

How Binary Search Works?

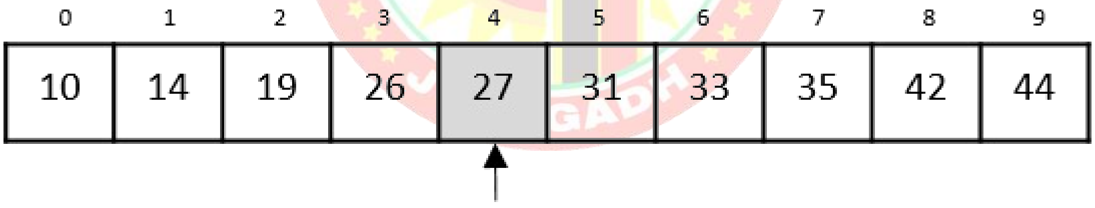
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



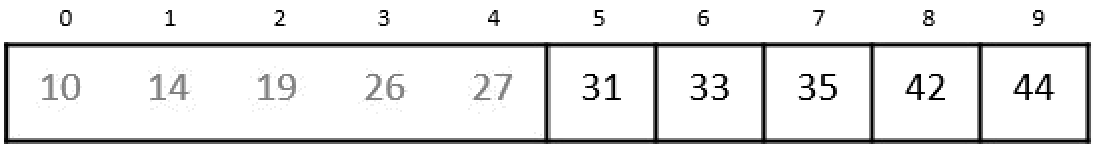
First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



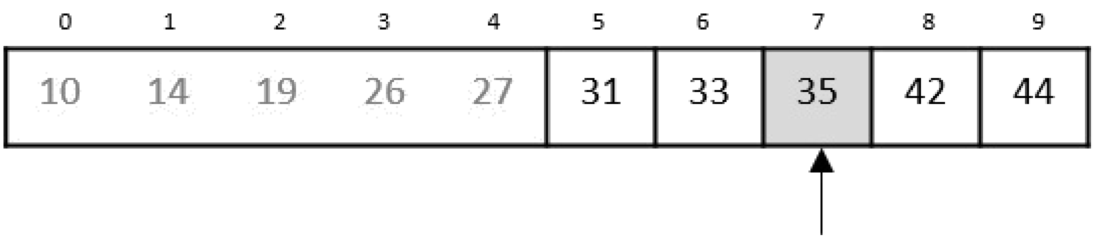
Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.



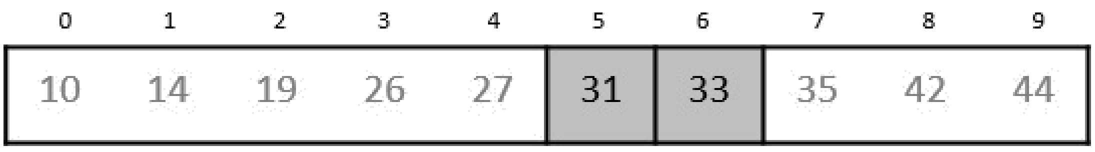
We change our low to mid + 1 and find the new mid value again.

low = mid + 1
mid = low + (high - low) / 2

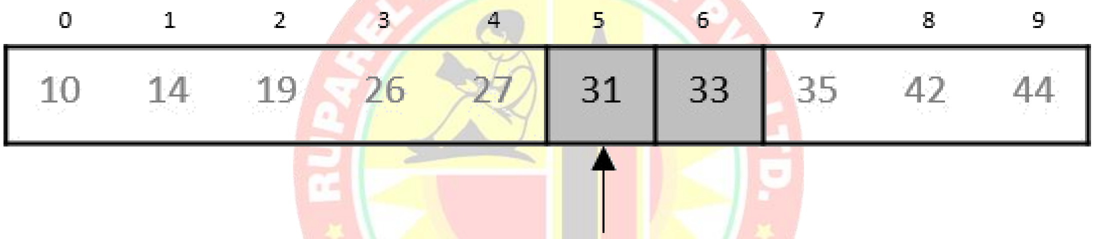
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



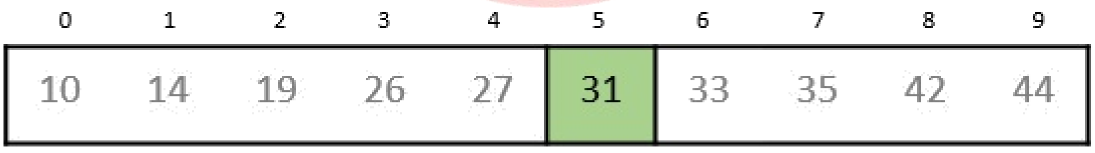
The value stored at location 7 is not a match, rather it is less than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Example:

```
#include<stdio.h>
void binary_search(int a[], int low, int high, int key)
{
        int mid;
        mid = (low + high) / 2;
        if (low <= high)
```

```
	{
		if (a[mid] == key)
			printf("Element found at index: %d\n", mid);
		else if(key < a[mid])
			binary_search(a, low, mid-1, key);
		else if (a[mid] < key)
			binary_search(a, mid+1, high, key);
	}
	else if (low > high)
		printf("Unsuccessful Search\n");
}
int main()
{
	int i, n, low, high, key;
	n = 5;
	low = 0;
	high = n-1;
	int a[10] = {12, 14, 18, 22, 39};
	key = 22;
	binary_search(a, low, high, key);
	key = 23;
	binary_search(a, low, high, key);
	return 0;
}
```

Output:

```
Element found at index: 3
Unsuccessful Search
```