Assignment No. 2

Aim: Implement Binary search program with Divide and Conquer design strategy for n numbers using C++. Discuss Best, Average and Worst time complexity.

Objectives:
1. To apply Binary search method and compute its time complexity.
2. To apply Divide and conquer strategy and find Best case, Worst case and Average case complexity.

Theory:

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

Divide and Conquer algorithm consists of a dispute using the following three steps.

1. **Divide** the original problem into a set of sub-problems.
2. **Conquer:** Solve every sub-problem individually, recursively.
3. **Combine:** Put together the solutions of the sub-problems to get the solution to the whole problem.

Binary Searching Technique:

This method is incorporates the divide and conquer strategy and uses its functionality to search a given element by dividing the array into two halves and then checking for the searching element from either the left or the right side of array, depending on the length of the array. Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

Binary Search = First element of the Array + Last element of the Array / 2

Working of the Binary searching technique:

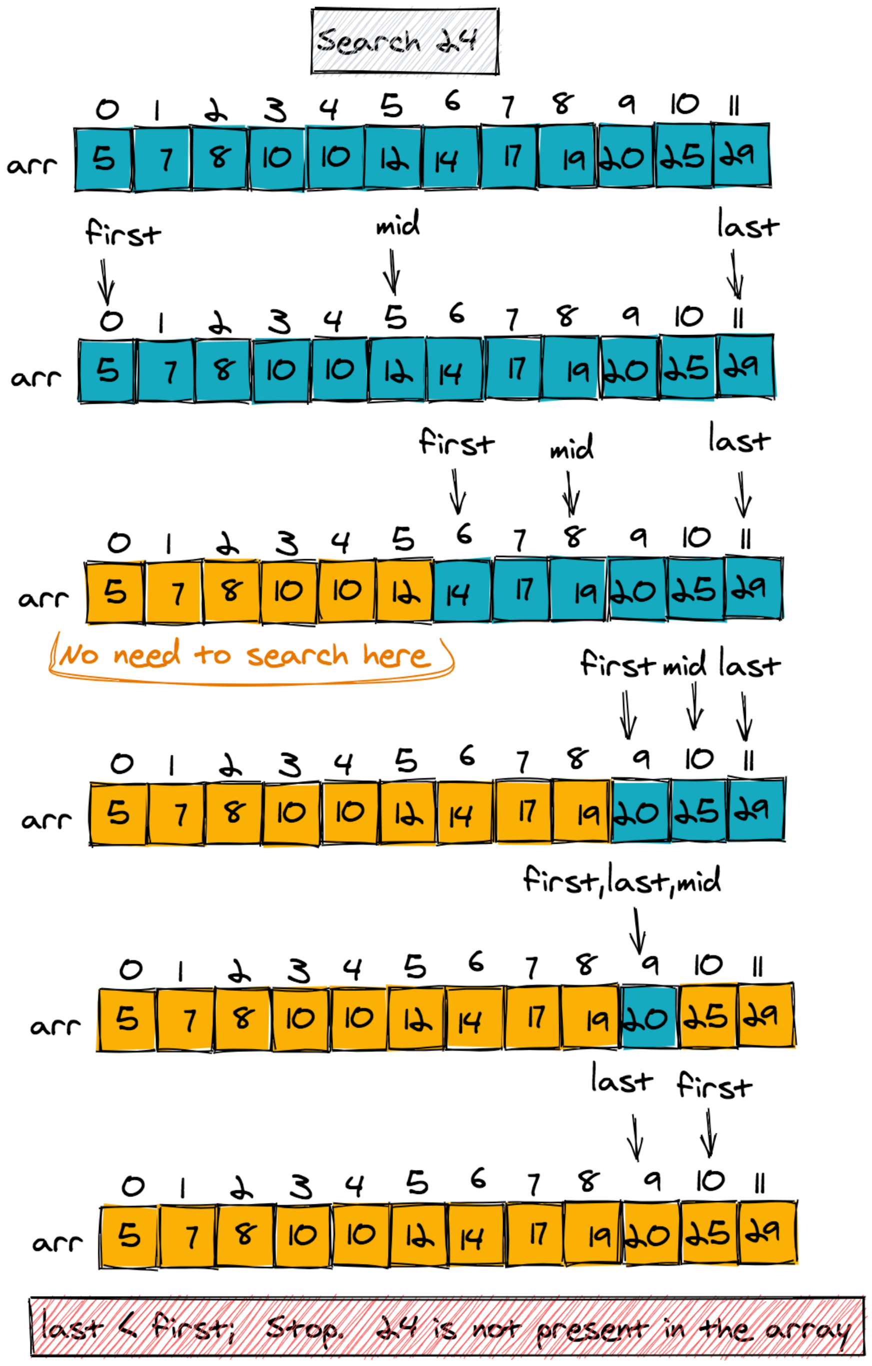
Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

Algorithm for Binary Search Technique:

```
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
```



Complexity of an Algorithm:

An algorithm's complexity is a measure of the amount of data that it must process in order to be efficient. Domain and range of this function are generally expressed in natural units. There are two methods of complexity namely

1. **Time complexity:**

   Time complexity is defined in terms of how many times it takes to run a given algorithm, based on the length of the input. Time complexity is not a measurement of how much time it takes to execute a particular algorithm because such factors as programming language, operating system, and processing power are also considered. Time complexity is a type of computational complexity that describes the time required to execute an algorithm. The time complexity of an algorithm is the amount of time it takes for each statement to complete. As a result, it is highly dependent on the size of the processed data. It also aids in defining an algorithm's effectiveness and evaluating its performance.

2. **Space complexity:**
   When an algorithm is run on a computer, it necessitates a certain amount of memory space. The amount of memory used by a program to execute it is represented by its space complexity. Because a program requires memory to store input data and temporal values while running, the space complexity is auxiliary and input space.

These are two of the important things needed for any algorithm as it can bring more information related to a given problem and thus can tell use which problem can be used whenever needed. Thus to show or represent these type of the problems we use the asymptotic notations.

What Are Asymptotic Notations?

Asymptotic Notations are programming languages that allow you to analyze an algorithm's running time by identifying its behavior as its input size grows. This is also referred to as an algorithm's growth rate. When the input size increases, does the algorithm become incredibly slow? Is it able to maintain its fast run time as the input size grows? You can answer these questions thanks to Asymptotic Notation.

You can't compare two algorithms head to head. It is heavily influenced by the tools and hardware you use for comparisons, such as the operating system, CPU model, processor generation, and so on. Even if you calculate time and space complexity for two algorithms running on the same system, the subtle changes in the system environment may affect their time and space complexity.

Asymptotic notations are classified into three types:

- Big-Oh (O) notation
- Big Omega (Ω) notation
- Big Theta (Θ) notation

These Complexities are expressed using these notations but in three specific situations, such as

1. **Best Case Complexity:**
   It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time. In this case, the execution time serves as a lower bound on the algorithm's time complexity.

2. **Average Case Complexity:**
   You add the running times for each possible input combination and take the average in the average case. Here, the execution time serves as both a lower and upper bound on the algorithm's time complexity.

3. **Worst Case Complexity:**
   It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time possible. In this case, the execution time serves as an upper bound on the algorithm's time complexity.

Program:

Output:

Conclusion: