Experiment No. 5
To implement Binary Search Algorithm
Date of Performance:29/03/2024
Date of Submission:07/03/2024

Experiment No. 5

Title: Binary Search Algorithm

Aim: To study and implement Binary Search Algorithm **Objective:** To introduce Divide and Conquer based algorithms

Theory:

Binary search is a highly efficient algorithm used to locate a target value within a sorted array. It works by repeatedly dividing the search interval in half until the target value is found or the search interval is empty.

Working Principle:

Binary search relies on the fact that the array is sorted. It compares the target value with the middle element of the array. If the target value matches the middle element, the search is successful. If the target value is less than the middle element, the search continues on the left half of the array. If the target value is greater, the search continues on the right half.

Steps of Binary Search:

Step 1:

Initialize two pointers, low and high, to the first and last indices of the array respectively.

Let low = 0 and high = n - 1 (where n is the size of the array)

Step 2:

Repeat the following steps until low is less than or equal to high.

Calculate the middle index: mid = (low + high) / 2.

Compare the target value with the middle element arr[mid].

If the target value equals arr[mid],

return mid

If the target value is less than arr[mid],

update high = mid - 1 (search the left half).

If the target value is greater than arr[mid],

update low = mid + 1 (search the right half).

Step 3:

If the search interval becomes empty (i.e., low exceeds high), the target value is not present in the array. Return a sentinel value (e.g., -1) to indicate that the value was not found.

Example:

Let's say we want to search for the value 12 in array [2, 4, 6, 8, 10, 12, 14, 16, 18, 20] using the binary search algorithm.

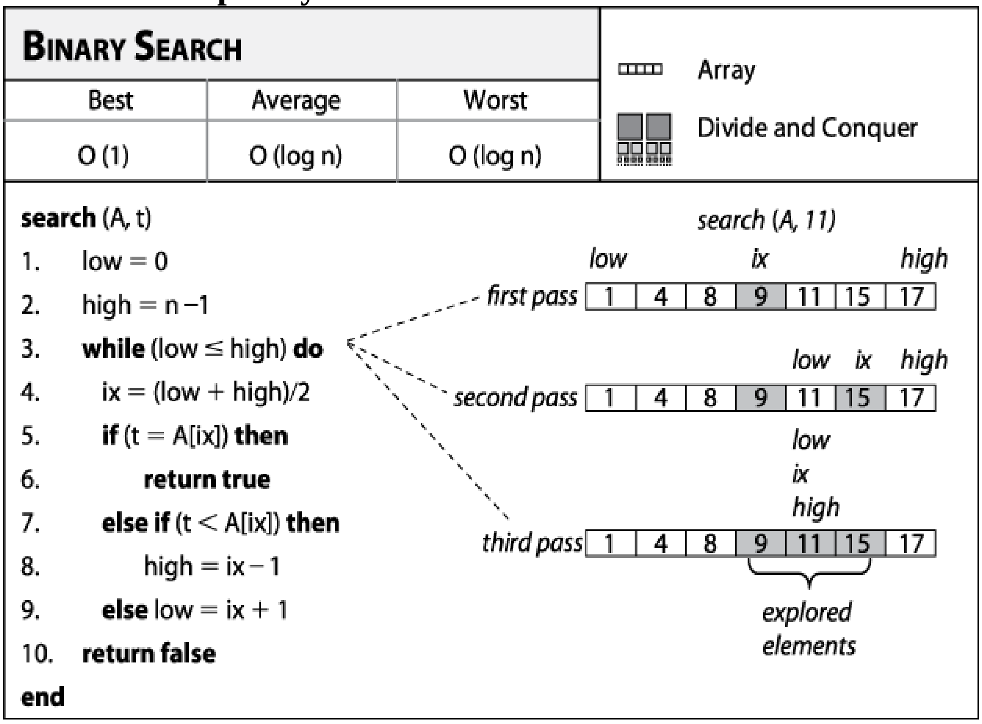
Initialize two pointers, low and high, to the first and last indices of the array.

low = 0, high = 9 (for an array of size 10).

Pass No.	Find Middle Element	Compare arr[mid] with the target value	Update Pointers
1	mid = (low+high)/2. mid = (0+9)/2 = 4.	Compare arr[mid] with the target value (12). arr[4] = 10 is less than 12, Indicating that target value in the right half of array.	Since the target value is greater than the middle element. Update low to mid + 1. low = mid + 1 = 4 + 1 = 5.
2	mid =(low+high)/2. mid =(5+9)/2=7.	arr[7] = 16 is greater than 12, indicating that the target value lies in the left half of the remaining array.	Update high to mid - 1. high = mid -1 =7-1 = 6.
3	mid = (low + high) / 2.	arr[5] = 12 matches the target value.	------------

	mid = (5 + 6) / 2 = 5.		
Result: Return the index of the found element (5 in this case). Binary search algorithm successfully located the target value 12 in the array.			

Algorithm and Complexity:



Best Case:

- In binary search, the key is initially compared to the array's middle element.
- If the key is in the center of the array, the algorithm only does one comparison, regardless of the size of the array.
- As a result, the algorithm's best-case running time is T(n) = 1.

Worst Case:

- Every iteration, the binary search, search space is decreased by half, allowing for maximum $\log_2 n$ array divisions.
- If the key is at the leaf of the tree or it is not present at all, then the algorithm does $\log_2 n$ comparisons, which is maximum.
- The number of comparisons increases in logarithmic proportion to the amount of the input. As a result, the algorithm's worst-case running time would be $T(n) = O(\log_2 n)$.
- The problem size is reduced by a factor of two after each iteration, and the method does one comparison.
- Recurrence of binary search can be written as $T(n) = T(n/2) + 1$. Solution to this recurrence leads to same running time, i.e. $O(\log_2 n)$. Detail derivation is discussed here:
- In every iteration, the binary search does one comparison and creates a new problem of size n/2.
- So, recurrence equation is,

  $T(n) = T(n/2) + 1$, if $n > 1$

  $T(n) = 1$, if $n = 1$

- Only one comparison is needed when there is only one element in the array.
- Let solve by iterative approach,

$$T(n) = T(n/2) + 1 \quad \ldots(1)$$

put n by n/2 in Equation (1) to find T(n/2)

$$T(n/2) = T(n/4) + 1 \quad \ldots(2)$$

put value of $T(n/2)$ in Equation (1),

$T(n) = T(n/2^2) + 1$ …(3)

put n by n/2 in Equation (2) to find $T(n/4)$,

$T(n/4) = T(n/8) + 1$

Use value of $T(n/4)$ in Equation (3),

$T(n) = T(n/2^3) + 3$

. . . . .

After k iterations,

$T(n) = T(n/ 2^k) + k$ --- (4)

Assume, $n/ 2^k = 1$ so, $n = 2^k$

Take log from both sides

$\text{Log } n = \log 2^k \Rightarrow \text{Log } n = k \log_2 2$

{ Using $\log_2 2 = 1$ }

Use $k = \log_2 n$ in Equation (4),

$T(n) = T(1) + \log n$

$= 1 + \log n$ {Ignore constant part}

$T(n) = O(\log_2 n)$

Average Case:

- The average case for binary search occurs when the key element is neither in the middle nor at the leaf level of the search tree.
- On average, it does half of the log2 n comparisons, which will turn out as
  T (n) = O(log2 n).

- The complexity of linear search and binary search for all three cases is compared in the following table.

Search Method	Best case	Average case	Worst case
Binary Search	O(1)	$O(\log_2 n)$	$O(\log_2 n)$
Linear Search	O(1)	O(n)	O(n)

Code:

```
#include <stdio.h>

#include <conio.h>


int binarySearch(int arr[],int low,int high,int x)

{

 while (low <= high)

 {

  int mid = low + (high - low) / 2;

  if (arr[mid] == x)

  {

   return mid;

  }

  else if (arr[mid] < x)

  {
```

```
   low = mid + 1;
  }
  else
  {
   high = mid - 1;
  }
 }
 return -1;
}

int main()
{
 int n,i,x;
 int result;
 int arr[50];

 printf("Enter the number of elements in the array: ");
 scanf("%d", &n);
 printf("Enter %d elements in  order:\n", n);
 for (i=0;i<n;i++)
 {
  scanf("%d",&arr[i]);
 }
 printf("Enter the element to search for: ");
 scanf("%d", &x);
```

```
 result = binarySearch(arr, 0, n - 1, x);
 (result == -1) ? printf("Element is not present"
                         " in array")
                : printf("Element is present at "
                         "index %d",
                         result);
 getch();
 return 0;
}
```

Output:

```
C:\Users\admin\Documents\b

Enter the number of elements in the array: 6
Enter 6 elements in  order:
7 89 5 6 3 56
Enter the element to search for: 8
Element is not present in array
```

```
C:\Users\admin\Documents\b

Enter the number of elements in the array: 5
Enter 5 elements in  order:
8 7 3 2 9
Enter the element to search for: 9
Element is present at index 4
--------------------------------
Process exited after 73.92 seconds with return value 0
Press any key to continue . . .
```

Conclusion:

In conclusion, Binary Search is a powerful algorithm for finding elements in a sorted array. Its efficiency stems from repeatedly halving the search space, resulting in a time complexity of O(log2 n). With its simplicity and effectiveness, Binary Search is widely used in diverse applications, providing fast and reliable search capabilities.