Experiment No. 4
Binary Search Algorithm
Date of Performance:
Date of Submission:

Experiment No. 4

Title: Binary Search Algorithm

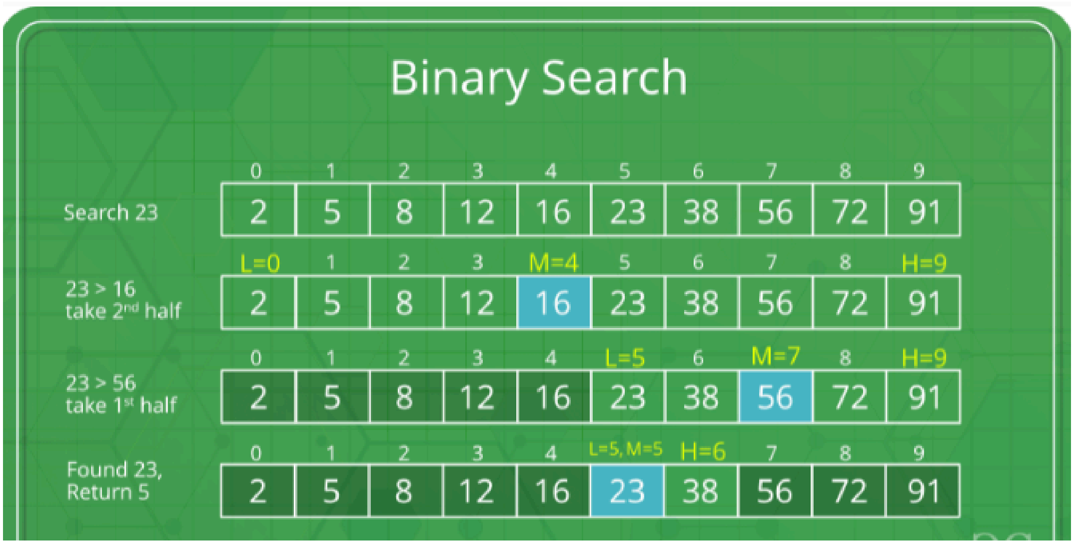
Aim: To study and implement Binary Search Algorithm

Objective: To introduce Divide and Conquer based algorithms

Theory:

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty

- Binary search is efficient than linear search. For binary search, the array must be sorted, which is not required in case of linear search.
- It is divide and conquer based search technique.
- In each step the algorithms divides the list into two halves and check if the element to be searched is on upper or lower half the array
- If the element is found, algorithm returns.



The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

- Compare x with the middle element.
- If x matches with the middle element, we return the mid index.
- Else If x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we recur for the right half.
- Else (x is smaller) recur for the left half.
- Binary Search reduces search space by half in every iterations. In a linear search, search space was reduced by one only.
- n=elements in the array
- Binary Search would hit the bottom very quickly.

	Linear Search	Binary Search
2^{nd} iteration	n-1	n/2
3^{rd} iteration	n-2	n/4

Example:

Algorithm BINARY_SEARCH(A, key)
// Description: Perform BS on array A
// I/P : array A of size n & key element to be searched.
// O/P : Success/failure.

low ← 1
high ← n
while low < high do
mid ← (low+high)/2
if A[mid] == key then
return mid
else if A[mid] < key then
low ← mid+1
else high ← mid-1
end
end
return 0

A{11, 22, 33, 44, 55, 66, 77, 88}
(indices 1 2 3 4 5 6 7 8)
key = 33

low = 1
high = 8
mid = 1+8/2
= 4
A[4] == 33 ×
A[4] < 33 ×
44
high = 4-1
high = 3

{11, 22, 33}
1 2 3

low = 1
high = 3
mid = 1+3/2
= 2
A[2] == 33 ×
22 < 33
low = 3

{33} mid = 3+3/2 = 3
A[3] = 33
A[mid] = 33
key = A[3]

Algorithm and Complexity:

The binary search

- Algorithm 3: the binary search algorithm

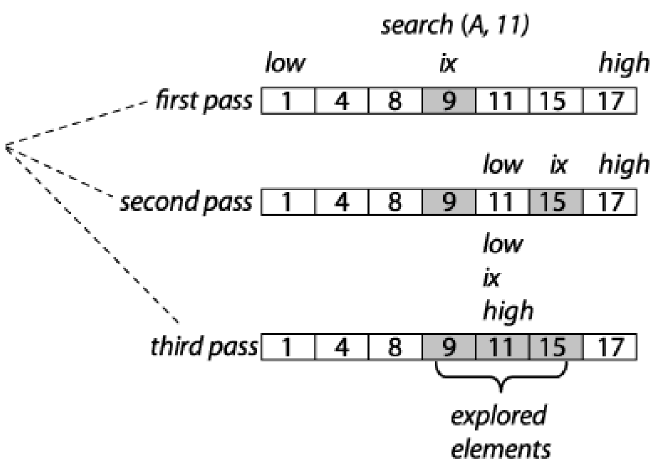
Procedure binary search (x: integer, a_1, a_2, …,a_n: increasing integers)
  i :=1 { i is left endpoint of search interval}
  j :=n { j is right endpoint of search interval}
While i < j
begin
  m := $\lfloor (i + j) / 2 \rfloor$
  if x > a_m **then** i := m+1
  else j := m
end
If x = a_i **then** *location* := i
else *location* :=0
{*location* is the subscript of the term equal to x, or 0 if x is not found}

2

Binary Search			
Best	Average	Worst	Array
O (1)	O (log n)	O (log n)	Divide and Conquer

search (A, t)
1. low = 0
2. high = n − 1
3. **while** (low ≤ high) **do**
4. ix = (low + high)/2
5. **if** (t = A[ix]) **then**
6. **return true**
7. **else if** (t < A[ix]) **then**
8. high = ix − 1
9. **else** low = ix + 1
10. **return false**

end



Best Case:
Key is first compared with the middle element of the array.
The key is in the middle position of the array, the algorithm does only one comparison, irrespective of the size of the array.
T(n)=1

Worst Case:
In each iteration search space of BS is reduced by half, Maximum log n(base 2) array divisions are possible.
Recurrence relation is
T(n)=T(n/2) +1
Running Time is O(logn).

Average Case:

Key element neither is in the middle nor at the leaf level of the search tree.
It does half of the log n(base 2).
Base case=O(1)
Average and worst case=O(logn)

Implementation:

```
#include <stdio.h>

int binarySearch(int arr[], int l, int r, int x)

{

while (l <= r) {

int m = l + (r - l) / 2; if (arr[m] == x)

return m; if (arr[m] < x)

l = m + 1;

else

r = m - 1;

}

return -1;

}

int main(void)

{

int arr[] = { 2, 3, 4, 10, 40 };
```

int n = sizeof(arr) / sizeof(arr[0]); int x = 10;

int result = binarySearch(arr, 0, n - 1, x); (result == -1) ? printf("Element is not present"

" in array")

: printf("Element is present at "

"index %d", result);

return 0;

}

Output:

```
Output
/tmp/jpAHhYVQ7Z.o
Element is present at index 3

=== Code Execution Successful ===
```

Conclusion: The Binary Search Algorithm divides a sorted array into halves, finding desired elements or exhausting search space. Its logarithmic time complexity is O(log n), ideal for large datasets. Careful handling of edge cases is crucial.