Vidyavardhini's College of Engineering & Technology

Department of Computer Science and Engineering (Data Science)

ACADEMIC YEAR: 2024-25

Course: Analysis of Algorithm Lab

Course code: CSL401

Year/Sem: SE/IV

Experiment No.: 03
Aim: To find the minimum and maximum &amp; implement a binary search Technique using DAC.
Name: SOHAM HEMENDRA RAUT
Roll Number: 24
Date of Performance: 30/01/2024
Date of Submission: 06/02/2025

Evaluation

Performance Indicator	Max. Marks	Marks Obtained
Performance	5	
Understanding	5	
Journal work and timely submission.	10	
Total	**20**	

Performance Indicator	Exceed Expectations (EE)	Meet Expectations (ME)	Below Expectations (BE)
Performance	5	3	2
Understanding	5	3	2
Journal work and timely submission.	10	8	4

Checked by

Name of Faculty : Mrs. Komal Champanerkar

Signature : **Date** :

❖ **Aim: To find the minimum and maximum &amp; implement a binary search Technique using DAC.**

❖ **Theory:**

- **Finding minimum and maximum:**
  To find the minimum and maximum values in an array using the Divide and Conquer (DAC) approach, we break the problem into smaller subproblems. We recursively divide the array into two halves. The minimum and maximum values for the entire array can be determined by combining the results from these two halves.

- **Binary Search Technique :**
  Binary search works by repeatedly dividing the search space in half. We start by comparing the target value to the middle element of the array. Based on this comparison, the search space is halved, and the algorithm proceeds with either the left or right half of the array. This process of recursively halving the search space continues until the target element is found or until there are no more elements to search.

- **Algorithms:**
  Algorithm finding MIN MAX ():
  Step 1: Split the array into two smaller subarrays by finding its midpoint.
  Step 2: If a subarray has one element, that element is both the minimum and maximum and If the subarray has two elements, compare them to find the minimum and maximum.
  Step 3: After finding the minimum and maximum values for both halves, combine them by making comparisons.

Complexity:

In all scenarios—best case, worst case, and average case, the time complexity is: O(n)

Algorithm for Binary Search ():

Step 1: Divide the array into two halves.

Step 2:

If the middle element is equal to the target, the search is successful.

If the middle element is greater than the target, continue the search in the left half.

If the middle element is less than the target, proceed to search in the right half.

Step 3: Repeat the substeps of step 2 till the target is achieved.

Complexity:

The complexity in the best case is: O(1)

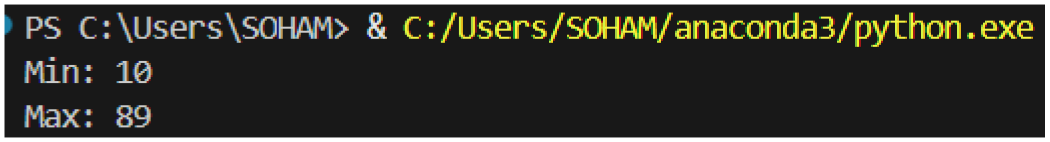
The complexity in the both average and worst case is: O(log n)

❖ **Program:**

- **Finding minimum and maximum:**

```
lt = [45, 10, 78, 23, 56, 12, 89, 34, 67, 18]

def minmax(low, high):
    # Base case: if the range has only one element
    if low == high:
        return [lt[low], lt[low]]  # Min and max are the same when only one element

    mid = (low + high) // 2
    # Recursively find the min and max in both halves
    left_minmax = minmax(low, mid)
    right_minmax = minmax(mid + 1, high)

    # Compare the results from both halves
    min_value = min(left_minmax[0], right_minmax[0])
    max_value = max(left_minmax[1], right_minmax[1])

    return [min_value, max_value]

result = minmax(0, len(lt) - 1)
print("Min:", result[0])
print("Max:", result[1])
```

❖ **Output:**

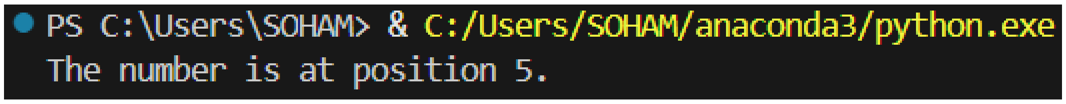


```
PS C:\Users\SOHAM> & C:/Users/SOHAM/anaconda3/python.exe
Min: 10
Max: 89
```

- **Binary Search:**

```
l = [12, 45, 23, 67, 34, 89, 21]  # New example list of numbers

def binary_search(number, lt, low, high):
    if low > high:
        return -1  # If low exceeds high, the number is not found
    mid = (low + high) // 2  # Find the middle index
    if number == lt[mid]:
        return mid  # Return the index if the number is found
    elif number > lt[mid]:
        return binary_search(number, lt, mid + 1, high)  # Search the right half
    else:
        return binary_search(number, lt, low, mid - 1)  # Search the left half


l.sort()                # Sort the list for binary search to work
# Searching for a number
inp = 67  # Number to be searched
position = binary_search(inp, l, 0, len(l) - 1)

if position == -1:
    print("The number is not in the list.")
else:
    print(f"The number is at position {position}.")
```

❖ **Output:**



```
PS C:\Users\SOHAM> & C:/Users/SOHAM/anaconda3/python.exe
The number is at position 5.
```

❖ **Conclusion:**

Binary Search is an efficient algorithm known for its time complexity of O(log n) and space complexity of O(log n) in recursive implementations, while iterative versions have a space complexity of O(1). This algorithm is particularly effective for large, sorted datasets because it reduces the search space by half with each iteration, making it faster than linear search. In contrast, the process of finding the minimum and maximum values using a divide-and-conquer approach has a time complexity of O(n) and a space complexity of O(log n) because of the depth of recursion involved.