Experiment 1

Introduction

The design and analysis of algorithms is a fundamental and essential aspect of computer science. Algorithms are step-by-step procedures or sets of rules that define how a task is to be performed. The goal of algorithm design is to develop efficient and effective solutions to computational problems. Algorithms aim to solve problems in the most efficient manner possible. Efficiency is often measured in terms of time complexity (how long an algorithm takes to run) and space complexity (how much memory an algorithm uses). Algorithms provide systematic approaches to problem-solving. They break down complex tasks into smaller, manageable steps, making it easier to understand and solve problems.

Key Concepts used

1. Efficiency: Efficiency is a critical aspect of algorithm design. This includes both time efficiency (how fast an algorithm runs) and space efficiency (how much memory it uses).
2. Scalability: Algorithms need to be scalable, meaning they can handle larger inputs without a significant increase in time or space complexity.
3. Divide and Conquer: A common strategy in algorithm design is the divide-and-conquer approach. It involves breaking a problem into smaller subproblems, solving them independently, and then combining their solutions.
4. Dynamic Programming: Dynamic programming is a technique where the solution to a problem is built by solving overlapping subproblems. It avoids redundant computations, improving efficiency.
5. Greedy Algorithms: Greedy algorithms make locally optimal choices at each step with the hope of finding a global optimum. They are often used for optimization problems.

Algorithm Analysis:

- Time Complexity: Time complexity describes the amount of time an algorithm takes to complete as a function of the size of the input. It is expressed using big-O notation.
- Space Complexity: Space complexity measures the amount of memory an algorithm uses, also expressed using big-O notation.
- Worst, Best, and Average Case: Algorithms may have different time complexities depending on the characteristics of the input. Analyzing worst, best, and average cases provides a comprehensive understanding of an algorithm's behavior.

Top to Down Approach

The top-down approach in algorithm design aligns well with the process of defining the problem at a high level, breaking it down into manageable subproblems, designing detailed solutions for each subproblem, implementing the algorithm, testing, and iteratively refining it for optimal performance. This systematic approach contributes to the development of efficient and well-designed algorithms.

1. Problem Definition: This step corresponds to clearly defining the problem that the algorithm aims to solve. For algorithm design, it involves understanding the input, desired output, and the constraints or requirements imposed by the problem.
2. Constrained and Conditions: Identifying any constraints or conditions in the problem is crucial. This may include limitations on time, space, or specific requirements for the algorithm's behavior. Considering these constraints guides the design process.

3. Design Strategy:Determining the overall strategy for solving the problem. This involves choosing algorithmic paradigms or approaches, such as divide and conquer, dynamic programming, or greedy algorithms, based on the nature of the problem.

4. Express and Develop the Algorithm: Expressing the algorithm involves breaking down the problem into subproblems and defining the steps for solving each subproblem. This step includes selecting appropriate data structures and algorithms for efficient problem-solving.

5. Validation (Dry Run): Before actual implementation, validate the algorithm through a dry run or simulation. This step helps ensure that the algorithm behaves as expected for various inputs and conditions.

6. Analysis: Assessing the time and space complexity of the algorithm. This step involves evaluating how the algorithm's performance scales with input size and identifying any potential bottlenecks or areas for improvement.

7. Coding: Translating the algorithmic design into actual code. This involves choosing a programming language, writing code that adheres to the algorithm's design, and considering best coding practices.

8. Testing: Creating test cases to validate the correctness and efficiency of the implemented algorithm. This includes checking edge cases, handling unexpected inputs, and ensuring the algorithm performs within specified constraints.

9. Installation: While "installation" might not directly apply to algorithm design, this step can be seen as the deployment and integration of the algorithm into the broader system or application where it will be used. Ensuring a smooth integration and addressing any issues that arise during deployment.

Implement binary search algorithm and compute its time complexity

Binary Search

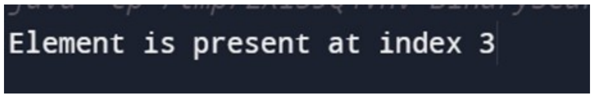
Working of Algorithm

- Initialize: Begin with the entire sorted array.
- Define Bounds: Set two pointers, low and high, to the start and end of the array, respectively.
- Middle Element: Calculate the middle index of the current search space using the formula: mid = (low + high) / 2.
- Compare: Compare the middle element with the target value.
- If the middle element is equal to the target, you have found the position, and the search is complete. If the middle element is less than the target, the target must be in the right half of the array. Update low to mid + 1 and repeat from step 3 for the right half. If the middle element is greater than the target, the target must be in the left half of the array. Update high to mid - 1 and repeat from step 3 for the left half.
- Repeat: Continue the process of dividing the search space in half until the target is found or the search space becomes empty.
- Termination: If the search space becomes empty (low > high), the target is not in the array.

CODE

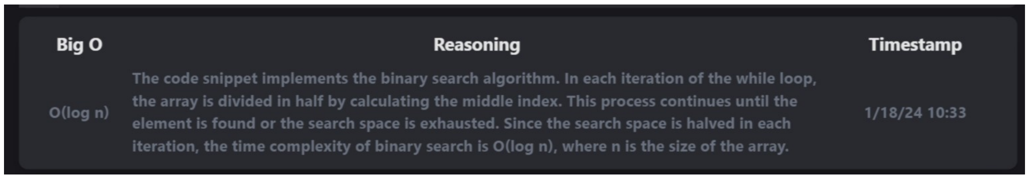
```
import java.io.*;
class BinarySearch
{
int binarySearch(int arr[], int x)
{
int l = 0, r = arr.length - 1;
while (l <= r)
{
int m = l + (r - l) / 2;
if (arr[m] == x) return m;
if (arr[m] < x) l = m + 1;
else r = m - 1;
}
return -1;
}
public static void main(String args[])
{
BinarySearch
ob = new BinarySearch();
int arr[] = { 2, 3, 4, 1 };
int n = arr.length;
int x = 10;
int result = ob.binarySearch(arr, x);
if (result == -1)
System.out.println( "Element is not present in array");
else
System.out.println("Element is present at " + "index " + result);
}
}
```

Output



```
Element is present at index 3
```

Time Complexity



Big O	Reasoning	Timestamp
O(log n)	The code snippet implements the binary search algorithm. In each iteration of the while loop, the array is divided in half by calculating the middle index. This process continues until the element is found or the search space is exhausted. Since the search space is halved in each iteration, the time complexity of binary search is O(log n), where n is the size of the array.	1/18/24 10:33

Experiment 2 : Implement merge sort algorithm and demonstrate divide and conquer technique.

Merge sort is a divide-and-conquer algorithm that works by recursively breaking down an array into smaller subarrays until they are trivially small (typically, arrays of size 1), sorting them, and then merging the sorted subarrays to produce a single sorted array. The key steps of the merge sort algorithm are as follows:

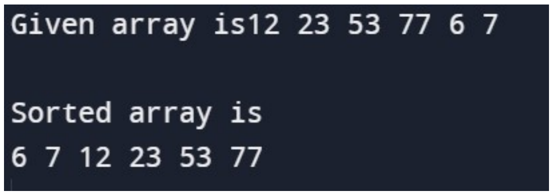
- Divide: The original array is divided into two (roughly) equal halves.
- Conquer: Recursively sort each of the subarrays created in the divide step. This process continues until the subarrays are of size 1 (trivially sorted).
- Merge: Merge the sorted subarrays to produce a single sorted array. The merging process involves comparing elements from the two subarrays and arranging them in sorted order.

The working of algorithm is as follows:
Base Case: If the array has only one element or is empty, it is considered already sorted, and no further action is needed.
Divide: Split the array into two halves. This can be done by finding the middle index of the array.
Conquer: Recursively apply the merge sort algorithm to each of the two halves created in the divide step.
Merge: Combine the two sorted halves back into a single sorted array. This involves comparing elements from the two halves and placing them in the correct order.

CODE:

```
import java.io.*;
class MergeSort
{
void merge(int arr[], int l, int m, int r)
{
int n1 = m - l + 1;
int n2 = r - m;
int L[] = new int[n1];
int R[] = new int[n2];
for (int i = 0; i < n1; ++i)
L[i] = arr[l + i];
for (int j = 0; j < n2; ++j)
R[j] = arr[m + 1 + j];
int i = 0, j = 0;
int k = l;
while (i < n1 && j < n2)
{
if (L[i] <= R[j])
{
arr[k] = L[i]; i++;
}
else
{ arr[k] = R[j]; j++; } k++; }
while (i < n1)
{
arr[k] = L[i]; i++; k++;
}
while (j < n2)
{ arr[k] = R[j]; j++; k++; }
}
void sort(int arr[], int l, int r)
{
if (l < r)
{ int m = l + (r - l) / 2;
sort(arr, l, m);
sort(arr, m + 1, r);
merge(arr, l, m, r);
}
}
static void printArray(int arr[])
{
int n = arr.length;
for (int i = 0; i < n; ++i)
System.out.print(arr[i] + " ");
System.out.println();
}
public static void main(String args[])
{
int arr[] = { 12,23, 53, 77, 6, 7 };
System.out.println("Given array is");
printArray(arr);
MergeSort ob = new MergeSort();
ob.sort(arr, 0, arr.length - 1);
System.out.println("\nSorted array is");
printArray(arr);
}
}
```

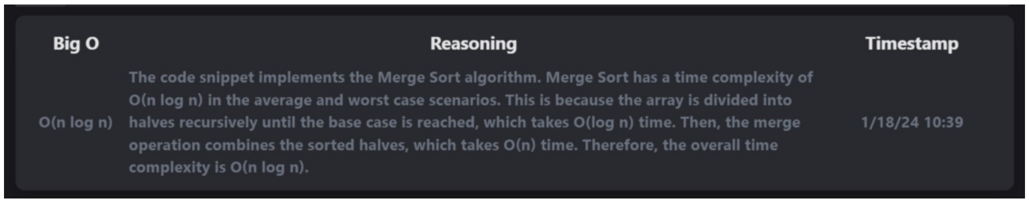
OUTPUT



```
Given array is12 23 53 77 6 7

Sorted array is
6 7 12 23 53 77
```

Time complexity



Big O	Reasoning	Timestamp
O(n log n)	The code snippet implements the Merge Sort algorithm. Merge Sort has a time complexity of O(n log n) in the average and worst case scenarios. This is because the array is divided into halves recursively until the base case is reached, which takes O(log n) time. Then, the merge operation combines the sorted halves, which takes O(n) time. Therefore, the overall time complexity is O(n log n).	1/18/24 10:39

Experiment 3: . Analyze the time complexity of Quick-sort algorithm.

QuickSort is a sorting algorithm that follows the divide and conquer paradigm. The key idea behind QuickSort is to choose a "pivot" element from the array and partition the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

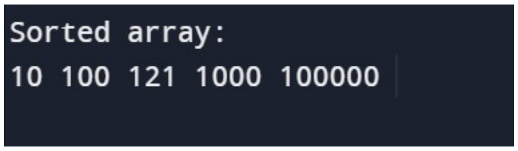
Algortihm working

1. Choose a Pivot: Select a pivot element from the array. The choice of pivot can affect the algorithm's efficiency.
2. Partitioning: Rearrange the elements in the array so that elements less than the pivot are on the left, and elements greater than the pivot are on the right. The pivot itself is in its final sorted position.
3. Recursion: Recursively apply the QuickSort algorithm to the sub-arrays formed by the partitioning step. This process is repeated on both the left and right sub-arrays.
4. Combine (No Operation): No combining step is needed in QuickSort as the elements are rearranged in place during the partitioning step.

CODE:

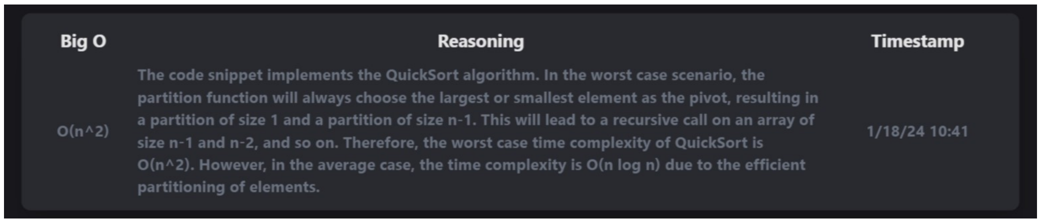
```
import java.io.*;
class GFG
{ static void swap(int[] arr, int i, int j)
{
int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
}
static int partition(int[] arr, int low, int high)
{
int pivot = arr[high];
int i = (low - 1);
for (int j = low;
j <= high - 1; j++)
{
if (arr[j] < pivot) { i++; swap(arr, i, j);
}
}
swap(arr, i + 1, high);
return (i + 1);
}
static void quickSort(int[] arr, int low, int high)
{
if (low < high)
{ int pi = partition(arr, low, high);
quickSort(arr, low, pi - 1);
quickSort(arr, pi + 1, high);
}
}
public static void printArr(int[] arr)
{
for (int i = 0; i < arr.length; i++)
{
System.out.print(arr[i] + " ");
}
}
public static void main(String[] args)
{
int[] arr = { 100, 10, 1000,100000,121};
int N = arr.length;
quickSort(arr, 0, N - 1);
System.out.println("Sorted array:");
printArr(arr);
}
}
```

OUTPUT



```
Sorted array:
10 100 121 1000 100000
```

TIME COMPLEXITY



Big O	Reasoning	Timestamp
O(n^2)	The code snippet implements the QuickSort algorithm. In the worst case scenario, the partition function will always choose the largest or smallest element as the pivot, resulting in a partition of size 1 and a partition of size n-1. This will lead to a recursive call on an array of size n-1 and n-2, and so on. Therefore, the worst case time complexity of QuickSort is O(n^2). However, in the average case, the time complexity is O(n log n) due to the efficient partitioning of elements.	1/18/24 10:41