Government College of Engineering, Jalgaon
(An Autonomous Institute of Government of Maharashtra)

Name :	**Semester :** VI	**PRN :**
Class : T. Y. B.Tech Computer	**Academic Year :** 2024-25	**Subject :** CO356U DAA Lab
Course Teacher : Mr. Vinit Kakde		
Date of Performance :	**Date of Completion :**	

Practical no. 2

Aim: Implement an algorithm using divide and conquer approach(by quick sort).

Theory: ***Quick Sort*** is a widely used sorting algorithm in computer science. It's an efficient, comparison-based sorting algorithm that follows a divide-and-conquer approach to sort a list or an array of elements.
The reason is that we humans love to simplify our work by dividing it into smaller parts. Keeping this fact in mind, people have developed computer algorithms that follow the divide and conquer principle. One of these algorithms is the quicksort algorithm used to sort the elements in an array.

How Does Quicksort Work?

Quicksort works by recursively sorting the sub-lists to either side of a given pivot and dynamically shifting elements inside the list around that pivot.

As a result, the Quicksort method can be summarized in three steps:

1. **Pick:** Select an element.
2. **Divide:** Split the problem set, move smaller parts to the left of the pivot and larger items to the right.
3. **Repeat and combine:** Repeat the steps and combine the arrays that have previously been sorted.

QuickSort Time Complexity Analysis

Now that we know about quicksort, our next job would be to analyze the time complexity.
To do that, let us again consider an array.

Best Case Complexity

Let us consider that each time, the partitioning occurs from the middle. We can visualize it like this:
Here, the number of iterations required to partition the array always sums up to n for each level.
To calculate the number of levels, we have to determine the number of times we divide the size of the initial array by two until we get 1.

Therefore,
$n / 2^k = 1$
$=> n = 2^k$
$=> \log_2(n) = \log_2(2^k)$

=> $k = \log_2(n)$

Hence, the total time complexity will be a function of nlogn i.e. O(nlogn)

Now, most of the time, the partitioning is not precisely from the middle. This is the ideal case, and so it gives the best case complexity.

Average Case Complexity

For average-case complexity, let us consider the partitioning to be in a 3:1 ratio.
The total number of iterations for partitioning in each level will again be n.
For the number of levels, we have to determine the number of times we divide the size of the initial array by four till we get 1.
Therefore,

$n / 4^k = 1$

=> $n = 4^k$

=> $\log_4(n) = \log_4(4^k)$

=> $k = \log_4(n)$

Hence the average case time complexity will again be O(nlogn).

Worst Case Complexity

In the worst-case scenario, we will be sorting an array that is already sorted. So, the partitioning diagram will be as follows:
Therefore, for large values of n, the time complexity will be,

$n + (n – 1) + (n – 2) + (n – 3) + \ldots + 2 + 1 = n(n – 1) / 2 = (n^2 – n) / 2$

Hence, the worst-case complexity is O(n^2).

Space Complexity of Quick Sort

1. Best & Average Case: O(logn)

- Quicksort is a **divide-and-conquer** algorithm.
- It works by partitioning the array and recursively sorting the left and right parts.
- It uses **in-place sorting**, meaning it doesn't require extra space for new arrays (except for recursive function calls).
- The **recursion depth** depends on how the array is divided.

Derivation:

- In the **best and average case**, Quicksort divides the array **roughly in half** at each step.
- The recursion depth will be **logarithmic** in terms of n.
- The space required for function calls (stack space) is O(logn).

2. Worst Case: O(n)

- The worst case occurs when the array is **already sorted** or **reverse sorted**, and we always pick the smallest or largest element as the pivot.
- This results in **unbalanced partitions** where one side has $n-1$ elements and the other has none.

- The recursion depth becomes **n**, leading to O(n) space complexity.

Algorithm of Quicksort

Step 1: Choose the pivot following any of the four methods (the last element is commonly used as the pivot).
Step 2: Assign the counter i to the leftmost element's index and the counter j to the rightmost element's index, excluding the counter.
Step 3: If i is less than j, proceed.
Step 4: If the element in position i is less than the pivot, increment i.
Step 5: If the element in position j is greater than the pivot, decrement j.
Step 6: If the element in i is greater than the pivot and the element in position j is less than the pivot, swap the elements in i and j.
Step 7: If i is greater than j, swap the element in position i and the pivot.
Step 8: Continue the process for the left and the right partitions formed.

With this, we have completed the partition.

Now, for the quicksort part,

Step 1: Declare a function with three parameters, an array (say arr) and two integer type variables (say i and j).

Step 2: If arr[i] < arr[j], partition the array between arr[i] and arr[j].

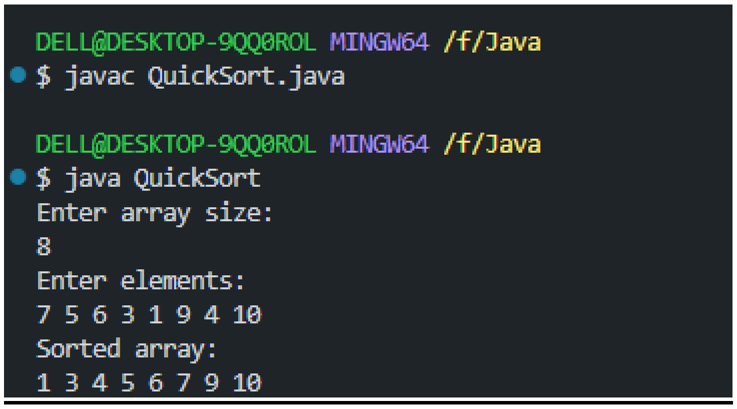
Step 3: Recursively call the function with three parameters, an array (arr) and two integers (i and (partition position - 1)).

Step 4: Recursively call the function with three parameters, an array (arr) and two integers ((partition position + 1), j).

Program:

```
import java.util.Scanner;

public class quick_sort {

    static void quickSort(int arr[], int low, int high) {

        if (low >= high) return;

        int pivot = partition(arr, low, high);

        quickSort(arr, low, pivot - 1);

        quickSort(arr, pivot + 1, high);

    }

    static int partition(int arr[], int low, int high) {

        int pivot = arr[high], i = low;

        for (int j = low; j < high; j++) {

            if (arr[j] < pivot) swap(arr, i++, j);

        }

        swap(arr, i, high);

        return i;

    }

    static void swap(int arr[], int i, int j) {

        int temp = arr[i];

        arr[i] = arr[j];

        arr[j] = temp;

    }

    public static void main(String args[]) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter array size: ");

        int n = sc.nextInt(), arr[] = new int[n];

        System.out.println("Enter elements:");
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        quickSort(arr, 0, n - 1);
        System.out.println("Sorted array:");
        for (int num : arr)
            System.out.print(num + " ");
    }
}
```

Output:



```
DELL@DESKTOP-9QQ0ROL MINGW64 /f/Java
$ javac QuickSort.java

DELL@DESKTOP-9QQ0ROL MINGW64 /f/Java
$ java QuickSort
Enter array size:
8
Enter elements:
7 5 6 3 1 9 4 10
Sorted array:
1 3 4 5 6 7 9 10
```

Conclusion: Quicksort remains one of the most efficient and widely used sorting algorithms due to its simplicity and speed.

Questions and answers

1. Why is Quicksort faster than other sorting algorithms?

Quicksort has a lower constant factor in its time complexity compared to other O(nlogn) algorithms like Merge Sort. It also sorts in place, reducing extra memory usage.

2. How does Quicksort choose the pivot?

Common pivot selection methods:

- Last element (Lomuto partition)
- First element
- Random element
- Median-of-three (first, middle, and last)

3. Is Quicksort stable?

No, Quicksort is not stable because it may swap elements far apart, changing their relative order.

4. When is Quicksort preferred over Merge Sort?

- When in-place sorting is needed
- When the dataset is not already sorted
- When extra memory usage should be minimized

5. How can we optimize Quicksort?

- Use randomized pivot selection to avoid worst-case scenarios
- Use tail recursion to reduce stack depth
- Switch to Insertion Sort for small sub arrays

Name & Sign of Course Teacher
Mr. Vinit Kakde