**Assignment 2**

**Name: Atharva Salitri**

**Roll No.: 029**

**Branch: TY CSAI-B**

**Batch: B2**

**PRN: 12310120**

**Title:**

Write a program to implement Quick Sort to sort an array of integers in ascending order. Find out Time and space complexity.

**Objective:**

* To understand the mechanism of comparison-based sorting.
* To implement Bubble Sort in C++ or Java.
* To analyze time and space complexity of Quick Sort.

**Software Requirements:**

* Operating System: Windows/Linux
* Language: C++ or Java
* Compiler: g++/javac

**Hardware Requirements:**

* Processor: 2 GHz or above
* RAM: 4 GB or more
* Disk Space: Minimum 500 MB

**Theory:**

Quick sort is a highly efficient **divide-and-conquer** sorting algorithm. It works by selecting a **pivot element** from the array and partitioning the other elements into two sub-arrays: those less than the pivot and those greater than the pivot. The sub-arrays are then recursively sorted. This process continues until the entire array is sorted**.**

Algorithm:

**Step 1 −** Choose a pivot element from the array (commonly the first, last, or middle element). **Step 2 −** Rearrange the array such that all elements less than the pivot are placed before it, and all greater elements are placed after it (this is called partitioning).

**Step 3 −** Place the pivot in its correct position in the sorted array.

**Step 4 −** Recursively apply the same process to the left sub-array (elements smaller than the pivot) and right sub-array (elements greater than the pivot).

**Step 5 −** Continue until the base case is reached (sub-array size ≤ 1).

**Pseudocode of Quick sort:**

Start

Function quickSort(arr, low, high):

if low < high:

pi = partition(arr, low, high)

quickSort(arr, low, pi - 1) // Recursively sort left sub-array

quickSort(arr, pi + 1, high) // Recursively sort right sub-array

Function partition(arr, low, high):

pivot = arr[high] // Choose last element as pivot

i = low - 1

for j = low to high-1:

if arr[j] < pivot:

i = i + 1

swap(arr[i], arr[j])

swap(arr[i + 1], arr[high])

return (i + 1)

End

**Time Complexity:**

| **Best Case** | O(n log n) |
| --- | --- |
| Average Case | O(n log n) |
| Worst Case | O(n²) |

**Space Complexity:**

Quick sort is an in-place sorting algorithm (no extra array is required). However, it uses the call stack due to recursion.

* Space Complexity = O(log n) for the recursion stack (best/average).
* In the worst case (highly unbalanced partitions), space can grow to O(n).

**CODE:**

import java.util.Scanner;

public class SortingDemo {

    public static void bubbleSort(int[] arr) {

        int n = arr.length;

        for (int i = 0; i < n - 1; i++) {

            for (int j = 0; j < n - i - 1; j++) {

                if (arr[j] > arr[j + 1]) {

                    int temp = arr[j];

                    arr[j] = arr[j + 1];

                    arr[j + 1] = temp;

                }

            }

        }

    }

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

        int pivot = arr[high];

        int i = low - 1;

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

            if (arr[j] < pivot) {

                i++;

                int temp = arr[i];

                arr[i] = arr[j];

                arr[j] = temp;

            }

        }

        int temp = arr[i + 1];

        arr[i + 1] = arr[high];

        arr[high] = temp;

        return i + 1;

    }

    public static void printArray(int[] arr) {

        for (int num : arr) {

            System.out.print(num + " ");

        }

        System.out.println();

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter no. of elements:");

        int n = sc.nextInt();

        int[] arr = new int[n];

        System.out.println("Enter elements:");

        for (int i = 0; i < n; i++) {

            arr[i] = sc.nextInt();

        }

        int[] arrBubble = arr.clone();

        int[] arrQuick = arr.clone();

        System.out.println("Bubble Sort:");

        bubbleSort(arrBubble);

        printArray(arrBubble);

        System.out.println("Quick Sort:");

        quickSort(arrQuick, 0, arrQuick.length - 1);

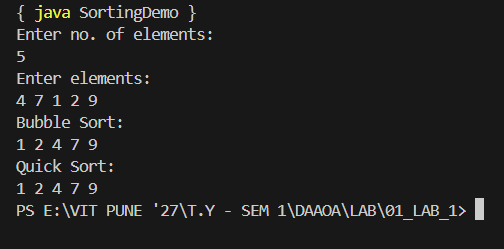
        printArray(arrQuick);

        sc.close();

    }

}

**OUTPUT:**

****

**Time and Space Complexity Analysis:**

* Best Case (balanced partitions): O(n log n)
* Average Case: O(n log n)
* Worst Case (already sorted or reverse sorted with bad pivot choice): O(n²)

**Space Complexity Analysis:**

* Quick Sort is in-place, meaning it does not need extra space for arrays.
* Uses O(log n) space for recursion stack in best/average cases.
* In worst case, recursive stack can grow to O(n).

FUNCTION bubbleSort(A, n) // Space: +1 for n (no new array)

FOR i = 0 TO n-2 // Time: +n-1 (≈ +n)

FOR j = 0 TO n-i-2 // Time: +(n-i-1) per i; total: \*n\*n

IF A[j] > A[j+1] // Time: +1 per check

temp ← A[j] // Space: +1 (temp), Time: +1

A[j] ← A[j+1] // Time: +1

A[j+1] ← temp // Time: +1

ENDIF

ENDFOR

ENDFOR

ENDFUNCTION

// Bubble Sort Total: Time O(n^2), Space O(1) (in-place)[web:12][web:11][web:15]

FUNCTION quickSort(A, low, high) // Space: O(log n) avg., O(n) worst (stack depth)

IF low < high // Time: +1

pi ← partition(A, low, high) // Time: +n (partitioning step)

quickSort(A, low, pi - 1) // Recursion, depth log n avg.

quickSort(A, pi + 1, high) // Recursion, depth log n avg.

ENDIF

ENDFUNCTION

FUNCTION partition(A, low, high) // Space: +1 (pivot), +1 (i), +1 (temp)

pivot ← A[high] // Time: +1

i ← low - 1 // Time: +1

FOR j = low TO high-1 // Time: +(high-low) per call

IF A[j] < pivot // Time: +1 per check

i ← i + 1 // Time: +1

temp ← A[i] // Space: +1, Time: +1

A[i] ← A[j] // Time: +1

A[j] ← temp // Time: +1

ENDIF

ENDFOR

temp ← A[i+1] // Space: +1, Time: +1

A[i+1] ← A[high] // Time: +1

A[high] ← temp // Time: +1

RETURN i + 1 // Time: +1 (return)

ENDFUNCTION

FUNCTION printArray(A, n) // Space: +1 (temp for num)

FOR each num IN A // Time: +n

PRINT num // Time: +1 per iteration

ENDFOR

PRINT newline // Time: +1

ENDFUNCTION

FUNCTION main

DECLARE n // Space: +1

INPUT n // Time: +1

DECLARE array A of size n // Space: +n

FOR i FROM 0 TO n-1 // Time: +n

INPUT A[i] // Time: +1 per iteration

ENDFOR

DECLARE arrBubble as clone of A // Space: +n

DECLARE arrQuick as clone of A // Space: +n

PRINT "Bubble Sort:"

bubbleSort(arrBubble, n) // Time: O(n^2), Space: +1

printArray(arrBubble, n) // Time: +n

PRINT "Quick Sort:"

quickSort(arrQuick, 0, n-1) // Avg Time: O(n log n), Space: O(log n)

printArray(arrQuick, n) // Time: +n

ENDFUNCTION

**Conclusion:**

Quick sort is one of the fastest sorting algorithms for large datasets due to its **divide-and-conquer** approach and average-case **O(n log n)** time complexity.  
However, in the **worst case** (unbalanced partitions), its performance can degrade to **O(n²)**.  
It is widely used in practice and often outperforms other simple algorithms like **Bubble Sort** and **Insertion Sort**.