**Name: Atharva Salitri**

**Roll No.: 029**

**Branch: TY CSAI-B**

**Batch: B2**

**PRN: 12310120**

**Experiment No. 1**

**Title:**

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

**CO-PO mapping**

|  |  |  |  |
| --- | --- | --- | --- |
| Title of Experiment | CO Mapping | CO Statements | PO Mapping |
| Assignment Based on Sorting strategy. (Implement Bubble Sort to sort an array in ascending order and analyze time & space complexity) | CO1, CO4 | CO2: To design efficient algorithms for computational problems using appropriate algorithmic paradigm. CO3: To analyze asymptotic complexity of the algorithm for a complex computational problem using suitable mathematical techniques. | PO1, PO2, PO3 |

**Objective:**

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

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order**.**

**Algorithm:**

1. Check if the first element in the input array is greater than the next element in the array.
2. If it is greater, swap the two elements; otherwise move the pointer forward in the array.
3. Repeat Step 2 until we reach the end of the array.
4. Check if the elements are sorted; if not, repeat the same process (Step 1 to Step 3) from the last element of the array to the first.
5. The final output achieved is the sorted array.

**Pseudocode of bubble sort:**

Start

Repeat for i = 0 to n-1

a. Repeat for j = 0 to n-i-1

- If arr[j] > arr[j+1], swap them

End

**Time Complexity:**

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

**Space Complexity:**

It sorts data directly within array without additional memory apart from few variables (counter and temp). The memory usage does not grow with the size of input. Regardless of whether you are sorting 10 elements or 10,000, fixed amount of memory is used for variables.

Hence **Space Complexity of bubble sort is O(1).**

**Conclusion:**

Bubble sort is easy to understand and implement. However, it is inefficient on large lists and is rarely used in practice for performance-critical applications.

**Source Code, with description and with Output Need to be Uploaded to the VOLP**

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);

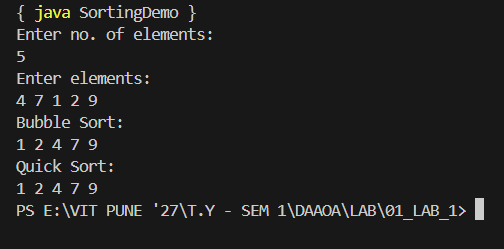
        printArray(arrQuick);

        sc.close();

    }

}

**OUTPUT:**

****

**Time and Space Complexity Analysis:**

**Bubble Sort:**

* Best Case Time Complexity: **O(n)** (when the array is already sorted)
* Average Case Time Complexity: **O(n²)**
* Worst Case Time Complexity: **O(n²)** (when the array is sorted in reverse order)
* Space Complexity: **O(1)** (in-place sorting with constant extra space)

**Quick Sort:**

* Best Case Time Complexity: **O(n log n)** (when partitioning divides array into nearly equal halves)
* Average Case Time Complexity: **O(n log n)**
* Worst Case Time Complexity: **O(n²)** (when partitioning is highly unbalanced, e.g., already sorted or reverse sorted with poor pivot choice)
* Space Complexity: **O(log n)** in best/average case (due to recursion stack)
* Space Complexity: **O(n)** in worst case (due to skewed recursion tree)

FUNCTION bubbleSort(A, n) // No new space; Space: +1 for n

FOR i FROM 0 TO n-2 // Time: +n-1 (≈+n)

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

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

temp ← A[j] // Space: +1, Time: +1

A[j] ← A[j+1] // Time: +1

A[j+1] ← temp // Time: +1

ENDIF

ENDFOR

ENDFOR

ENDFUNCTION

// Space complexity for bubbleSort: O(1), as swaps use a single temp variable only[web:16].

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

IF low < high // Time: +1

pi ← partition(A, low, high) // Time: +n on avg.

quickSort(A, low, pi - 1) // Recursive call; Time: depends on split

quickSort(A, pi + 1, high) // Recursive call; Time: depends on split

ENDIF

ENDFUNCTION

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

pivot ← A[high] // Time: +1

i ← low - 1 // Time: +1

FOR j FROM low TO high - 1 // Time: +(high-low)

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

ENDFUNCTION

FUNCTION printArray(A, n) // Space: +1 for temp (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: +n^2, Space: +1

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

PRINT "Quick Sort:"

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

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

ENDFUNCTION