|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Analysis of Algorithms** | **Semester:** | **IV** |
| **Date of Performance:** | **10 / 01 / 2023** | **Batch No:** | **A – 2** |
| **Faculty Name:** | **Dr. Aarti Phadke** | **Roll No.:** | **16014022050** |
| **Faculty Sign & Date:** |  | **Grade / Marks:** | **\_\_\_ / 25** |

**Experiment No.: 1**

**Title: Implementation of Insertion Sort**

|  |
| --- |
| **Aim and Objective of the Experiment:** |
| To analyze performance of sorting method. |

|  |
| --- |
| **COs to be achieved:** |
| **CO1:** Analyze the asymptotic running time and space complexity of algorithms. |

|  |
| --- |
| **Apparatus / Software Tools Used:** |
| 1. VS Code 2. Microsoft Excel |

|  |
| --- |
| **Theory:** |
| Given a function to compute on n inputs the divide-and-conquer strategy suggests splitting the inputs into k distinct subsets, 1< k ≤n, yielding k subproblems. These sub-problems must be solved and then a method must be found to combine sub-solutions into a solution of the whole. The divide-and-conquer strategy can be reapplied if the sub-problems are still relatively large.  Often the sub-problems resulting from a divide-and-conquer design are the same type as the original problem. For those cases, a recursive algorithm naturally expresses the reapplication of the divide-and-conquer principle. Now smaller and smaller subproblems of the same kind are generated until eventually subproblems that are small enough to be solved without splitting are produced. |

|  |
| --- |
| **Code:** |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  void insertion\_sort(int arr[], int n)  {      int i, j, temp;      for(i = 1; i < n; i++)      {          temp = arr[i];          j = i-1;          while((temp < arr[j]) && (j>=0))          {              arr[j+1] = arr[j];              j--;          }      arr[j+1] = temp;      }  }  int main()  {      clock\_t start, end;      double cpu\_time\_used;      // start = clock();      int n;      printf("\nenter the number of elements: ");      scanf("%d", &n);        int arr[n];      srand(time(NULL));      for (int i = 0; i < n; i++)      {          arr[i] = rand() % (n + 1);      }      printf("\nunsorted array: \n");      for (int i = 0; i < n; i++)      {          printf("%d ", arr[i]);      }      printf("\n");      start = clock();      insertion\_sort(arr, n);      end = clock();      printf("\nsorted array: \n");      for (int i = 0; i < n; i++)      {          printf("%d ", arr[i]);      }      printf("\n");      // end = clock();      cpu\_time\_used = ((double)(end - start)) / (CLOCKS\_PER\_SEC / 1000);      printf("\ntime taken: %f ms\n", cpu\_time\_used);      return 0;  } |

|  |
| --- |
| **Stepwise-Procedure / Algorithm:** |
| **Algorithm Insertion Sort:**  INSERTION\_SORT (A, n)  // The algorithm takes as parameters an array A[1.. n] and the length n of the array.  // The array A is sorted in place: the numbers are rearranged within the array  // A [1...n] of eletype, n: integer  FOR j ← 2 TO length[A]  DO key ← A [j]  {Put A[j] into the sorted sequence A [1... j − 1]}  i ← j − 1  WHILE i > 0 and A[i] > key  DO A [i +1] ← A[i]  i ← i − 1  A [i + 1] ← key |

|  |
| --- |
| **Observation Table:** |
| **Graphs for varying input sizes of Insertion Sort:** |

|  |
| --- |
| **Calculations:** |
| 1. **Space Complexity for Insertion sort:**          1. **Time Complexity for Insertion sort:** |

|  |
| --- |
| **Post Lab Subjective / Objective Type Questions:** |
| **Solve the problem theoretically which was implemented during the practical.** |

|  |
| --- |
| **Conclusion:** |
| In conclusion, this experiment helped us understand the insertion sort algorithm and how it efficiently organizes data. After careful analysis, we determined its time and space complexities, offering valuable insights into its practical effectiveness across different situations. |

**Signature of faculty in-charge with Date:**