|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Analysis of Algorithms** | **Semester:** | **IV** |
| **Date of Performance:** | **31 / 01 / 2024** | **Batch No:** | **A – 2** |
| **Faculty Name:** | **Prof. Aarti Phadke** | **Roll No.:** | **16014022050** |
| **Faculty Sign & Date:** |  | **Grade / Marks:** | **\_\_\_ / 25** |

**Experiment No.: 3**

**Title: Merge Sort Analysis and Quick Sort Analysis**

|  |
| --- |
| **Aim and Objective of the Experiment:** |
| To learn the divide and conquer strategy of solving the problems of different types. |

|  |
| --- |
| **COs to be achieved:** |
| **CO2:** Describe various algorithm design strategies to solve different problems. |

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

|  |
| --- |
| **Theory:** |
| **Historical Profile:**  Quicksort and merge sort are divide-and-conquer sorting algorithms in which division is dynamically carried out. They are one the most efficient sorting algorithms.  **New Concepts to be learned:**  Number of comparisons, Application of algorithmic design strategy to any problem, Classical problem-solving vs Divide-and-Conquer problem solving. |

|  |
| --- |
| **Stepwise-Procedure / Algorithm:** |
| **Algorithm Recursive Quick Sort:**  void quicksort(Integer[] A, Integer left, Integer right) {  if (left < right) {  Integer q = partition(A, left, right);  quicksort(A, left, q - 1);  quicksort(A, q + 1, right);  }  }  Integer partition(Integer[] AT, Integer left, Integer right) {  Integer pivot = AT[left];  Integer lo = left + 1;  Integer hi = right;  while (lo <= hi) {  while (AT[hi] > pivot) {  hi = hi - 1;  }  while (lo <= hi && AT[lo] < pivot) {  lo = lo + 1;  }  if (lo <= hi) {  swap(AT, lo, hi);  }  }  swap(AT, pivot, hi);  return hi;  }  **Algorithm Merge Sort:**  void mergeSort(A, p, r) {  if (p < r) {  int q = floor((p + r) / 2);  mergeSort(A, p, q);  mergeSort(A, q + 1, r);  merge(A, p, q, r);  }  }  void merge(A, p, q, r) {  int n1 = q - p + 1;  int n2 = r - q;  int[] L = new int[n1 + 1];  int[] R = new int[n2 + 1];  for (int i = 1; i <= n1; i++) {  L[i] = A[p + i - 1];  }  for (int j = 1; j <= n2; j++) {  R[j] = A[q + j];  }  L[n1 + 1] = infinity;  R[n2 + 1] = infinity;  int i = 1;  int j = 1;  for (int k = p; k <= r; k++) {  if (L[i] <= R[j]) {  A[k] = L[i];  i++;  } else {  A[k] = R[j];  j++;  }  }  } |

|  |
| --- |
| **Codes / Outputs / Graphs:** |
| 1. **Quick Sort:**   #include <stdio.h>  #include <stdlib.h>  #include <time.h>  void swap(int arr[], int i, int j)  {      int temp = arr[i];      arr[i] = arr[j];      arr[j] = temp;  }  void quickSort(int arr[], int left, int right)  {      if (left < right)      {          int q = partition(arr, left, right);          quickSort(arr, left, q - 1);          quickSort(arr, q + 1, right);      }  }  int partition(int arr[], int left, int right)  {      int pivot = arr[left];      int low = left + 1;      int high = right;      while (low <= high) {          while (high > left && arr[high] > pivot) {              high--;          }          while (low <= high && arr[low] <= pivot) {              low++;          }          if (low <= high) {              swap(arr, low, high);          }      }      swap(arr, left, high);      // printf("%d", &high);        return high;  }  int main()  {      clock\_t start, end;      double cpu\_time\_used;      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();      quickSort(arr, 0, n - 1);      end = clock();      printf("\nsorted array: \n");      for (int i = 0; i < n; i++)      {          printf("%d ", arr[i]);      }      printf("\n");      cpu\_time\_used = ((double)(end - start)) / (CLOCKS\_PER\_SEC / 1000000);      printf("\ntime taken: %f ms\n", cpu\_time\_used);      return 0;  }         1. **Merge Sort:**   #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #include <limits.h>  void merge(int arr[], int p, int q, int r)  {      int n1 = q - p + 1;      int n2 = r - q;      int L[n1 + 1], R[n2 + 1];      for (int i = 0; i < n1; i++)          L[i] = arr[p + i];        for (int j = 0; j < n2; j++)          R[j] = arr[q + j + 1];      L[n1] = R[n2] = INT\_MAX;      int i = 0, j = 0;        for (int k = p; k <= r; k++)      {          if (L[i] <= R[j])          {              arr[k] = L[i];              i++;          }          else          {              arr[k] = R[j];              j++;          }      }  }  void mergeSort(int arr[], int p, int r)  {      if (p < r)      {          int q = (p + r) / 2;          mergeSort(arr, p, q);          mergeSort(arr, q + 1, r);          merge(arr, p, q, r);      }  }  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();      mergeSort(arr, 0, n - 1);      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;  } |

|  |
| --- |
| **Observation Table:** |
| 1. **The space complexity of Quick Sort):**      1. **Derivation of best case and worst-case time complexity (Quick Sort):**        1. **The space complexity of Merge Sort:**      1. **Derivation of best case and worst-case time complexity (Merge Sort):** |

|  |
| --- |
| **Post Lab Subjective / Objective Type Questions:** |
| 1. **Write down algorithms for quicksort. Show that the best case of the quick sort occurs when the split has constant proportionality (Prove it using any other choice of splits rather than equal subdivision i.e., n/2).**   **Sort the following list of elements in ascending order using Quick sort technique. Give the output of each pass.**  **29 23 17 57 34 89 65 27**         1. **Sort the following list of elements in ascending order using Merge sort technique. Give the output of each pass.**   **29 23 17 57 34 89 65 27** |

|  |
| --- |
| **Conclusion:** |
| In conclusion, this experiment helped us understand the Quick Sort and Merge Sort algorithms and how it efficiently organizes/sorts 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:**