|  |  |
| --- | --- |
| **Name** | Siddiqui Hawaiza Sabeel |
| **UID no.** | 2022701010 |
| **Experiment No.** | 2 |
| **SUBJECT** | DAA |

|  |  |
| --- | --- |
| **AIM:** | Experiment based on divide and conquer approach. |
| **Program 1** | |
| **PROBLEM STATEMENT :** | For this experiment, you need to implement two sorting algorithms namely Quicksort and Merge sort methods. Compare these algorithms based on time and space complexity. Time required for sorting algorithms can be performed using high\_resolution\_clock::now() under namespace std::chrono. You have to generate 1,00,000 integer numbers using C/C++ Rand function and save them in a text file. Both the sorting algorithms uses these 1,00,000 integer numbers as input as follows. Each sorting algorithm sorts a block of 100,200,300,...,100000 integer numbers with array indexes numbers A[0..99], A[100..199], A[200..299],…, A[99900..99999]. You need to use high\_resolution\_clock::now() function to find the time required for 100, 200, 300…. 100000 integer numbers. Finally, compare two algorithms namely Quicksort and Merge sort by plotting the time required to sort integers using LibreOffice Calc/MS Excel. The x-axis of 2-D plot represents the block no. of 1000 blocks. The y-axis of 2-D plot represents the tunning time to sort 1000 blocks of 100,200,300,...,100000 integer numbers. Note – You have to use C/C++ file processing functions for reading and writing randomly generated 100000 integer numbers |
| **ALGORITHM/**  **THEORY:** | 1. **Merge Sort:**   **Algorithm:**   * 1. Start   2. If the input array has length less than or equal to 1, return the array as it is already sorted.   3. Divide the input array into two halves.   4. Recursively sort the left half by calling merge sort on it.   5. Recursively sort the right half by calling merge sort on it.   6. Merge the two sorted halves into a single sorted array.   7. Return the final sorted array.   8. End  1. **Quick Sort:**   **Algorithm:**   * 1. Start   2. If the input array has length less than or equal to 1, return the array as it is already sorted.   3. Select a pivot element from the array.   4. Partition the other elements into two sub-arrays, based on whether they are less than or greater than the pivot.   5. Recursively sort the left sub-array by calling QuickSort on it.   6. Recursively sort the right sub-array by calling QuickSort on it.   7. Concatenate the left sub-array, the pivot, and the right sub-array to get the final sorted array.   8. Return the final sorted array..   9. End |
| **PROGRAM:** | #include <stdio.h>  #include <math.h>  #include <conio.h>  #include <stdlib.h>  #include <time.h>  void getInput()  {    FILE \*fp;    fp = fopen("inputexp2.text","w");    for(int i=0;i<100000;i++)    fprintf(fp,"%d ",rand()%100000);    fclose(fp);  }  void merge(int arr[], int p, int q, int r) {    // Create L ← A[p..q] and M ← A[q+1..r]    int n1 = q - p + 1;    int n2 = r - q;    int L[n1], M[n2];    for (int i = 0; i < n1; i++)      L[i] = arr[p + i];    for (int j = 0; j < n2; j++)      M[j] = arr[q + 1 + j];    // Maintain current index of sub-arrays and main array    int i, j, k;    i = 0;    j = 0;    k = p;    while (i < n1 && j < n2) {      if (L[i] <= M[j]) {        arr[k] = L[i];        i++;      } else {        arr[k] = M[j];        j++;      }      k++;    }    while (i < n1) {      arr[k] = L[i];      i++;      k++;    }    while (j < n2) {      arr[k] = M[j];      j++;      k++;    }  }  void mergeSort(int arr[], int l, int r) {    if (l < r) {    int m = l + (r - l) / 2;      mergeSort(arr, l, m);      mergeSort(arr, m + 1, r);      merge(arr, l, m, r);    }  }  int partition(int A[], int low, int high)  {      int pivot = A[low];      int i = low + 1;      int j = high;      int temp;      do      {          while (A[i] <= pivot)          {              i++;          }          while (A[j] > pivot)          {              j--;          }          if (i < j)          {              temp = A[i];              A[i] = A[j];              A[j] = temp;          }      } while (i < j);      temp = A[low];      A[low] = A[j];      A[j] = temp;      return j;  }  void quickSort(int A[], int low, int high)  {      int partitionIndex; // Index of pivot after partition      if (low < high)      {          partitionIndex = partition(A, low, high);          quickSort(A, low, partitionIndex - 1);  // sort left subarray          quickSort(A, partitionIndex + 1, high); // sort right subarray      }  }    int main(){      getInput();      FILE \*rt, \*tks;      int a=99;      int arrNums[100000];      clock\_t t;      rt = fopen("exp2.text", "r");      tks = fopen("merge.txt", "w");      for(int i=0; i<1000; i++){          for(int j=0; j<=a; j++){              fscanf(rt, "%d", &arrNums[j]);          }          t = clock();          mergeSort(arrNums,0, a+1);          t = clock() - t;          double time\_taken = ((double)t)/CLOCKS\_PER\_SEC;          fprintf(tks, "time taken for %d iteration is %Lf\n", (i+1), time\_taken);          printf("%d\t%lf\n", (i+1), time\_taken);          a = a + 100;          fseek(rt, 0, SEEK\_SET);      }      fclose(tks);      tks = fopen("quick.txt", "w");      a=99;      for(int i=0; i<1000; i++){          for(int j=0; j<=a; j++){              fscanf(rt, "%d", &arrNums[j]);          }          t = clock();          quickSort(arrNums,0, a+1);          t = clock() - t;          double time\_taken = ((double)t)/CLOCKS\_PER\_SEC;          fprintf(tks, "time taken for %d iteration is %Lf\n", (i+1), time\_taken);          printf("%d\t%lf\n", (i+1), time\_taken);          a = a + 100;          fseek(rt, 0, SEEK\_SET);      }      fclose(tks);      fclose(rt);      return 0;  } |
|  | |
| **RESULT:**      **GRAPH:** | |
|  |  |
| **CONCLUSION:** | we have used two algorithm techniques i.e mergesort  and quicksort to sort the random no.s . both the  algorithms have less time complexity. i have seen  behaviour of the algorithms with time using of graph . it is seen that quick has better time complexity than merge sort. |