|  |  |
| --- | --- |
| **NAME** | **Ms.NEERJA DOSHI** |
| **UID** | **2021300029** |
| **DIV** | **SE – CE -A** |
| **DAA EXPT 2** | |
| **DATE OF PERFORMANCE** | **14-02-2023** |
|  |  |
|  |  |
| **AIM :** | **Experiment Based on Divide and Conquer Techniques 1) MERGE SORT 2) QUICK SORT** |
| **ALGORITHM :** | **MERGE SORT step 1: start step 2: declare array and left, right, mid variable step 3: perform merge function. if left &gt; right return mid= (left+right)/2 mergesort(array, left, mid) mergesort(array, mid+1, right) merge(array, left, mid, right) step 4: Stop QUICK SORT FOR QUICK SORT : QUICKSORT (array A, start, end)      1. {   2.  1 if (start &lt; end)      3.  2 {   4. 3 p = partition(A, start, end)   5. 4 QUICKSORT (A, start, p - 1) 6. 5 QUICKSORT (A, p + 1, end) 7. 6} 8. } FOR PARTIONING ALGORITHM PARTITION (array A, start, end)      1. {   2.  1 pivot ? A[end]      3.  2 i ? start-1      4.  3 for j ? start to end -1 {   5.  4 do if (A[j] &lt; pivot) {     6.  5 then i ? i + 1      7.  6 swap A[i] with A[j]     8.  7  }}    9.  8 swap A[i+1] with A[end]      10.  9 return i+1   11. }** |
| **CODE :** | **#include <stdio.h> #include <stdlib.h> #include <time.h> void merge(int a[], int beg, int mid, int end) {  int i, j, k;  int n1 = mid - beg + 1;  int n2 = end - mid;  int LeftArray[n1], RightArray[n2];  for (int i = 0; i < n1; i++)  LeftArray[i] = a[beg + i];  for (int j = 0; j < n2; j++)  RightArray[j] = a[mid + 1 + j];  i = 0,  j = 0;  k = beg;  while (i < n1 && j < n2)  {  if (LeftArray[i] <= RightArray[j])  {  a[k] = LeftArray[i];  i++;  }  else  {  a[k] = RightArray[j];  j++;  }  k++;  }  while (i < n1)  {  a[k] = LeftArray[i];  i++;  k++;  }  while (j < n2)  {  a[k] = RightArray[j];  j++;  k++;  } } void mergeSort(int a[], int beg, int end) {  if (beg < end)  {  int mid = (beg + end) / 2;  mergeSort(a, beg, mid);  mergeSort(a, mid + 1, end);  merge(a, beg, mid, end);  } } void printArray(int a[], int n) {  int i;  for (i = 0; i < n; i++)  printf("%d ", a[i]);  printf("\n"); } int partition(int a[], int start, int end) {  int pivot = a[end];  int i = (start - 1);  for (int j = start; j <= end - 1; j++)  {  if (a[j] < pivot)  {  i++;  int t = a[i];  a[i] = a[j];  a[j] = t;  }  }  int t = a[i + 1];  a[i + 1] = a[end];  a[end] = t;  return (i + 1); } void quick(int a[], int start, int end) {  if (start < end)  {  int p = partition(a, start, end);  quick(a, start, p - 1);  quick(a, p + 1, end);  } } void printArr(int a[], int n) {  int i;  for (i = 0; i < n; i++)  printf("%d ", a[i]); } void main() {  int n = 0;  for (int k = 0; k < (100000 / 100); k++)  {  n = n + 100;  int num[n];  int quicksort[n];  int merge[n];  int j, min;  clock\_t start\_t, end\_t;  double total\_t;  printf("%d\t", n);  for (int i = 0; i < n; i++)  {  num[i] = rand() % 10;  merge[i] = num[i];  quicksort[i] = num[i];  }  start\_t = clock();  mergeSort(merge, 0, n - 1);  end\_t = clock();  total\_t = (double)(end\_t - start\_t) / CLOCKS\_PER\_SEC;  printf("%f\n", total\_t);  start\_t = clock();  quick(quicksort, 0, n - 1);  end\_t = clock();  total\_t = (double)(end\_t - start\_t) / CLOCKS\_PER\_SEC;  printf("%f\n", total\_t);  } }** |
| **GRAPHICAL REPRESENTATION :** |  |
| **OBSERVTIONS :** | 1) Merge sort follows a linear graph 2) Quick sort has a parabolic garph 3) Approximately 3 lakh swaps are required to sort the given Input of 1 lakh number in Quick Sort method |
| **CONCLUSION :** | Merge sort is more efficient as its worst case time complexity is O(logn) while in case of quick sort, it remains constant throughout all operations as we can see from its graph which is linear in nature. |