**Manav Bhanushali**

**CSE-DS D1**

**2021700008**

**DAA Exp :- 2**

**AIM:Sorting Methods-Divide and Conquer**

**Theory :-**

Merge SortMerge sort is a divide-and-conquer algorithm that sorts an array or list by breaking it down into smaller and smaller subarrays, sorting those subarrays, and then merging them back together in the correct order. The algorithm has a worst-case time complexity of O(n log n) and a space complexityof O(n).Here are the steps to perform a merge sort:Divide the array or list into two halves.Recursively sort each half.Merge the two sorted halves back together.

Quick sort is also a divide-and-conquer algorithm that sorts an array or list by selecting a "pivot" element and partitioning the array or list around that pivot. Elements smaller than the pivot are moved to its left, and elements larger than the pivotare moved to its right. The pivot is then placed in its correct position and the process is repeated for the two subarrays on either side of the pivot. The algorithm has a worst-case time complexity of O(n^2) and a space complexity of O(log n).MergesortmergeSort(arr[], l, r)if l < r1. Find the middle point to divide the array into two halves:middle = l + (r -l) / 22. Call mergeSort for the first half:mergeSort(arr, l, middle)3. Call mergeSort for the second half:mergeSort(arr, middle + 1, r)4. Merge the two halves sorted instep 2 and 3:merge(arr, l, middle, r)

Quick SortquickSort(arr[], low, high)if low < high1. partitionIndex = partition(arr, low, high)2. Call quickSort for the first partition:quickSort(arr, low, partitionIndex -1)3. Call quickSort for the second partition:quickSort(arr, partitionIndex + 1, high)

**Code :-**

#include <stdio.h>

#include <stdlib.h>

#include <time.h> // required to use clock function if executing program on linux

#ifdef \_WIN32 // to check if OS is windows (\_WIN32 is a macro defined on every windows based gcc compiler)

#include <windows.h> // required to use windows api for time measurement if executing on windows

#endif

long int cm=0,cq=0;

/\* clock function of time.h on linux provides precision upto microseconds,

but on windows precision upto only milliseconds is supported by clock function

on windows, the windows api has much better functions for measuring time. \*/

// prototypes:

void mergeSort(int \*, int);

void quickSort(int \*, int);

void mergeSortActual(int \*, int, int);

void merge(int \*, int, int, int);

void quickSortActual(int \*, int, int);

int partition(int \*, int, int);

void deepCopy(int \*, int \*, int);

void printArray(int \*, int);

void generateRandomNumbers();

void LinuxMain();

#ifdef \_WIN32

void WindowsMain();

#endif

void printArray(int \*arr, int len)

{

for (int i = 0; i < len; i++)

printf("%d ", arr[i]);

printf("\n");

}

void mergeSort(int \*arr, int len)

{

int mid = (len-1)/2;

mergeSortActual(arr, 0, mid);

mergeSortActual(arr, mid+1, len-1);

merge(arr, 0, mid, len-1);

}

void mergeSortActual(int \*arr, int low, int high)

{

if(low < high)

{

int mid = (low+high)/2;

mergeSortActual(arr, low, mid);

mergeSortActual(arr, mid+1, high);

merge(arr, low, mid, high);

}

}

void merge(int \*arr, int low, int mid, int high)

{

int len = high - low + 1;

int temp[len];

int i = low;

int j = mid+1;

int temp\_idx = 0;

while(i <= mid && j <= high)

{

if(arr[i] < arr[j])

{

temp[temp\_idx++] = arr[i];

i++;

cm++;

}

else

{

temp[temp\_idx++] = arr[j];

j++;

cm++;

}

}

if(i>mid)

{

while(j <= high)

temp[temp\_idx++] = arr[j++];

}

else if(j>high)

{

while(i <= mid)

temp[temp\_idx++] = arr[i++];

}

for(int i=high; i>=low; i--)

arr[i] = temp[--temp\_idx];

}

void quickSort(int \*arr, int len)

{

int part = partition(arr, 0, len-1);

quickSortActual(arr, 0, part);

quickSortActual(arr, part+1, len-1);

}

void quickSortActual(int \*arr, int low, int high)

{

if(low < high)

{

int part = partition(arr, low, high);

quickSortActual(arr, low, part);

quickSortActual(arr, part+1, high);

}

}

int partition(int \*arr, int low, int high)

{

int pivot = arr[low];

int i = low, temp;

for(int j=low+1; j<=high; j++)

{

if(arr[j] < pivot)

{

i++;

temp = arr[j];

arr[j] = arr[i];

arr[i] = temp;

cq++;

}

}

temp = arr[low];

arr[low] = arr[i];

arr[i] = temp;

return i;

}

int main()

{

//generateRandomNumbers();

#ifdef \_WIN32

WindowsMain();

#else

LinuxMain();

#endif

return 0;

// printf("Enter length of array: ");

// int size;

// scanf("%d",&size);

// int arr[size];

// for(int i=0; i<size; i++)

// {

// printf("arr[%d] = ",i);

// scanf("%d",&arr[i]);

// }

// printf("Input array: ");

// printArray(arr, size);

// quickSort(arr, size);

// printf("After sorting: ");

// printArray(arr, size);

}

void deepCopy(int \*source, int \*dest, int len)

{

for (int i = 0; i < len; i++)

dest[i] = source[i];

}

void generateRandomNumbers()

{

FILE \*fptr = fopen("rand\_num.txt", "w");

time\_t cur\_time;

srand((unsigned int)time(&cur\_time));

for (int i = 0; i < 100000; i++)

fprintf(fptr, "%d\n", rand());

fclose(fptr);

}

#ifdef \_WIN32

void WindowsMain()

{

FILE \*rand\_num = fopen("rand\_num.txt", "r");

FILE \*dest = fopen("output.txt", "w");

fprintf(dest, "size | selection-sort-time | insertion-sort-time\n");

double time1, time2;

LARGE\_INTEGER clock\_freq, start, end;

QueryPerformanceFrequency(&clock\_freq);

for (int size = 100; size <= 100000; size += 100)

{

int arr1[size];

int arr2[size];

for (int j = 0; j < size; j++)

fscanf(rand\_num, "%d", &arr1[j]);

fseek(rand\_num, 0, SEEK\_SET);

deepCopy(arr1, arr2, size);

QueryPerformanceCounter(&start);

mergeSort(arr1, size);

QueryPerformanceCounter(&end);

time1 = (double)(end.QuadPart - start.QuadPart) \* 1.0 / clock\_freq.QuadPart;

QueryPerformanceCounter(&start);

quickSort(arr2, size);

QueryPerformanceCounter(&end);

time2 = (double)(end.QuadPart - start.QuadPart) \* 1.0 / clock\_freq.QuadPart;

fprintf(dest, "%6d | %19f | %19f\n", size, time1, time2);

printf("Size %d done!\n", size);

}

fclose(rand\_num);

fclose(dest);

}

#endif

void LinuxMain()

{

FILE \*rand\_num = fopen("rand\_num.txt", "r");

FILE \*dest = fopen("output.txt", "w");

fprintf(dest, "size\tmerge-sort-time\tquick-sort-time\tCount Merge\tCount Quick\n");

double time1, time2;

clock\_t start, end;

for (int size = 100; size <= 100000; size += 100)

{

int arr1[size];

int arr2[size];

for (int j = 0; j < size; j++)

fscanf(rand\_num, "%d", &arr1[j]);

fseek(rand\_num, 0, SEEK\_SET);

deepCopy(arr1, arr2, size);

start = clock();

mergeSort(arr1, size);

end = clock();

time1 = (double)(end - start) \* 1.0 / CLOCKS\_PER\_SEC;

start = clock();

quickSort(arr2, size);

end = clock();

time2 = (double)(end - start) \* 1.0 / CLOCKS\_PER\_SEC;

fprintf(dest, "%d\t%f\t%f\t%ld\t%ld\n", size, time1, time2,cm,cq);

printf("Size %d done!\n", size);

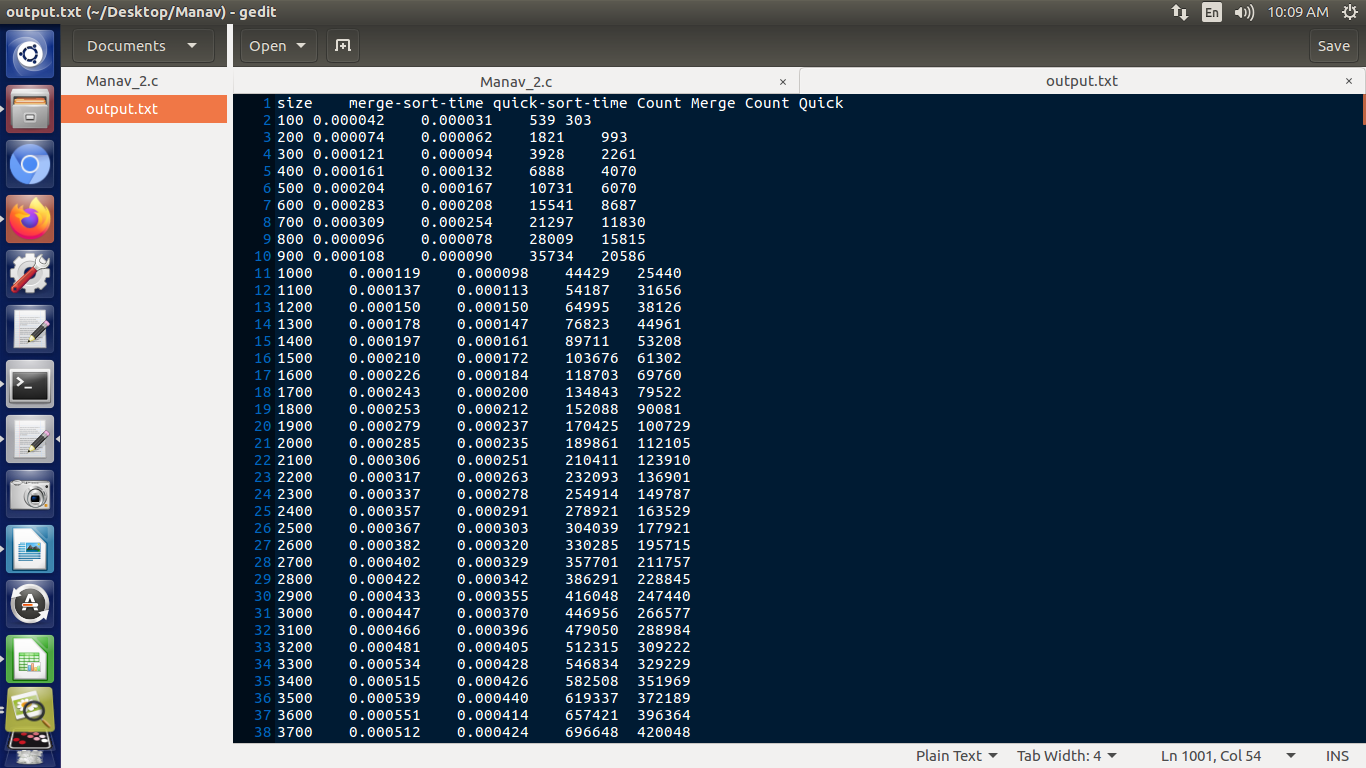
}

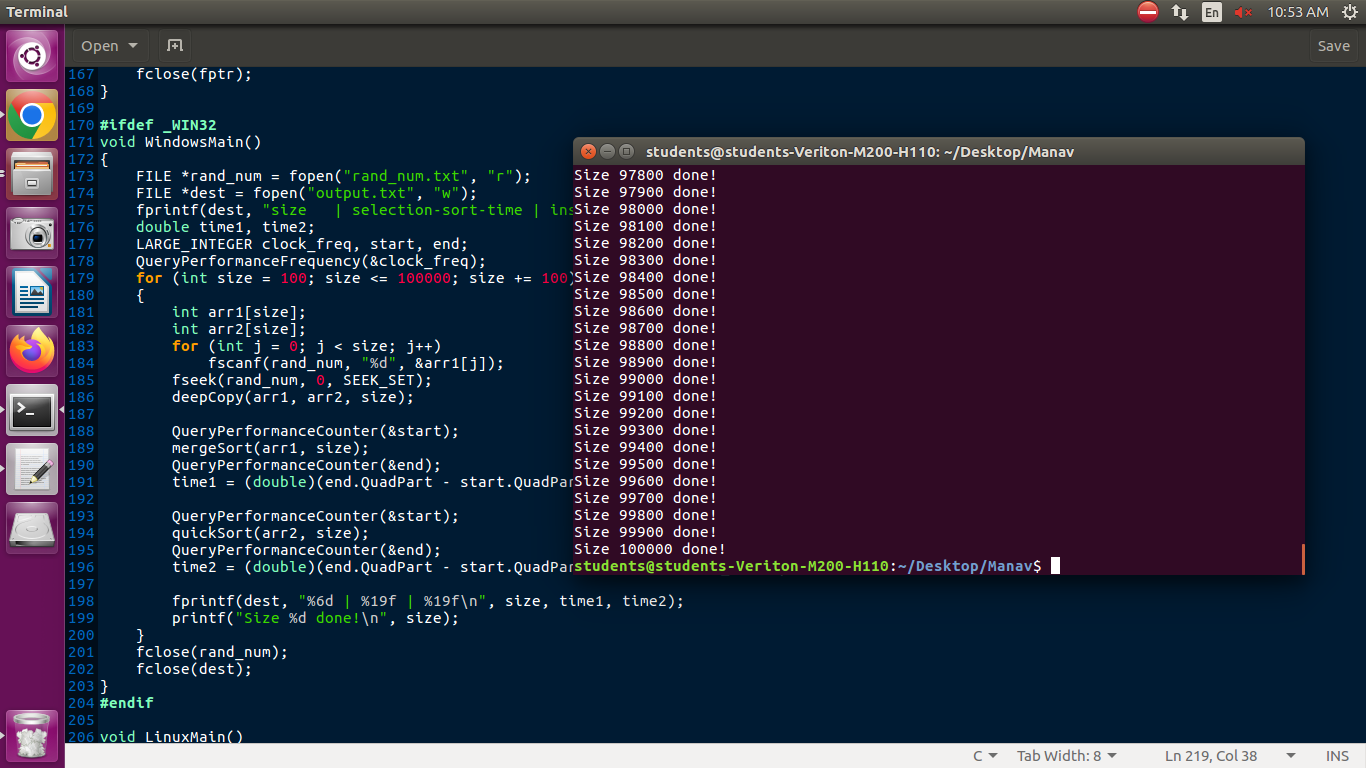
fclose(rand\_num);

fclose(dest);

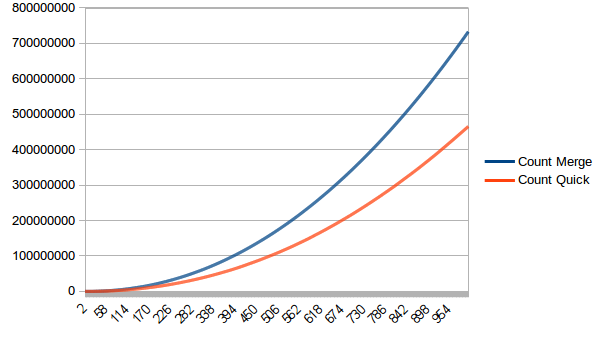
}

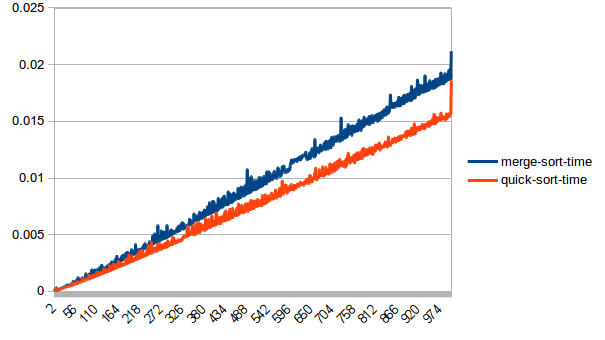
**Output :-**





**Result:**





**Observations:**Both merge sort and quick sort have an average-case time complexity of O(n log n), which makes them highly efficient for sorting large datasets. However, quick sort has a worst-case time complexity of O(n^2) if the pivot element is chosen poorly, while merge sort has a consistent worst-case time complexity of O(n log n). This means that merge sort is a safer choice for datasets with unpredictable distribution, while quick sort can be faster for datasets witha known distribution.No of Comparisons in mergesort are far less thanquick sort

**CONCLUSION:**.Inthis experiment Iunderstood divide and conquer algorithms like merge sort and quick sort and the basic principles behind them .

Also Iwas able to compare their efficient by plotting their graphs