**Shubhan Singh**

**SE-Comps B/Batch C**

**2022300118**

DAA Experiment 1

**Aim** – Experiment based on divide and conquer approach.

**Details** – Divide and conquer algorithm is a strategy of solving a large problem by breaking the problem into smaller sub-problems.

**Input** –Each student have to generate random 100000 numbers using rand() function and use this input as 1000 blocks of 100,200,300,...,100000 integer numbers to Quicksort and Merge sorting algorithms.

**Output** –

1) Store the randomly generated 100000 integer numbers to a text file.

2) Draw a 2D plot of both sorting algorithms such that the x-axis of 2-D plot represents the block no. of 1000 blocks. The y-axis of 2-D plot represents the running time to sort 1000 blocks of 100,200,300,...,100000 integer numbers.

3) Comment on Space complexity for two sorting algorithms.

**Source code(C language):**

* I have printed the output in a csv file so that the data can be easily imported into excel.

**(pseudocode given later)**

**#include<stdio.h>**

**#include<stdlib.h>**

**#include<time.h>**

**#define inf 0x7fffffff**

**#define SIZE 100000**

**FILE\* file;**

**void swap(int\* a, int\* b){**

**int temp=\*b;**

**\*b=\*a;**

**\*a=temp;**

**}**

**void merge(int\* arr,int start, int mid, int end){**

**int L[mid-start+2],R[end-mid+1];**

**L[mid-start+1]=R[end-mid]=inf;**

**for(int i=start;i<=mid;i++){**

**L[i-start]=arr[i];**

**}**

**for(int i=mid+1;i<=end;i++){**

**R[i-mid-1]=arr[i];**

**}**

**int i=0,l=0,r=0;**

**for(int i=start;i<=end;i++){**

**if(L[l]<R[r]){**

**arr[i]=L[l];**

**l++;**

**}**

**else{**

**arr[i]=R[r];**

**r++;**

**}**

**}**

**}**

**void mergesort(int\* arr, int start, int end){**

**if(end==start){**

**return;**

**}**

**int mid=(end+start)/2;**

**mergesort(arr,start,mid);**

**mergesort(arr,mid+1,end);**

**merge(arr,start,mid,end);**

**}**

**void quicksort(int\* arr, int start, int end){**

**if(end-start<2){**

**return;**

**}**

**int pivot=start;**

**int lower=end-1,higher=start+1;**

**while(1){**

**while(arr[higher]<=arr[pivot]){**

**higher++;**

**}**

**while(lower>start && arr[lower]>=arr[pivot]){**

**lower--;**

**}**

**if(lower<higher){**

**swap(&arr[pivot],&arr[lower]);**

**pivot=lower;**

**break;**

**}**

**swap(&arr[lower],&arr[higher]);**

**lower--;**

**higher++;**

**}**

**quicksort(arr,start,pivot);**

**quicksort(arr,pivot+1,end);**

**}**

**void runmergesort(int \*arr, int len){**

**int temp[len];**

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

**temp[i]=arr[i];**

**}**

**clock\_t time=clock();**

**mergesort(temp,0,len-1);**

**time=clock()-time;**

**double tme=(double)time/CLOCKS\_PER\_SEC;**

**fprintf(file,"%lf,",tme);**

**}**

**void runquicksort(int\* arr, int len){**

**int temp[len+1];**

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

**temp[i]=arr[i];**

**}**

**clock\_t time=clock();**

**temp[len]=inf;**

**quicksort(temp,0,len);**

**time=clock()-time;**

**double tme=(double)time/CLOCKS\_PER\_SEC;**

**fprintf(file,"%lf,",tme);**

**}**

**void runquicksortworst(int\* arr, int len){**

**int temp[len+1];**

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

**temp[i]=arr[i];**

**}**

**clock\_t time=clock();**

**temp[len]=inf;**

**quicksort(temp,0,len);**

**time=clock()-time;**

**double tme=(double)time/CLOCKS\_PER\_SEC;**

**fprintf(file,"%lf\n",tme);**

**}**

**int main(){**

**if(fopen("input.txt","r")==NULL){**

**srand(time(NULL));**

**file=fopen("input.txt","w");**

**for(int i=0;i<SIZE;i++){**

**fprintf(file,"%d ",rand());**

**}**

**fclose(file);**

**}**

**file=fopen("input.txt","r");**

**int arr[SIZE];**

**for(int i=0;i<SIZE;i++){**

**fscanf(file,"%d ",&arr[i]);**

**}**

**int best\_case\_merge[SIZE];**

**int worst\_case\_quick[SIZE];**

**int worst\_case\_merge[SIZE];**

**for(int i=0;i<SIZE;i++){**

**best\_case\_merge[i]=i+1;**

**worst\_case\_quick[SIZE-i-1]=i+1;**

**}**

**for(int i=0;i<SIZE/2;i++){**

**worst\_case\_merge[i]=2\*i+1;**

**}**

**for(int i=SIZE/2;i<SIZE;i++){**

**worst\_case\_merge[i]=2\*(i-SIZE/2+1);**

**}**

**fclose(file);**

**file=fopen("Exp\_2\_timings.csv","w");**

**fprintf(file,"merge sort best case,merge sort average case, merge sort worst case,quick sort average case, quick sort worst case\n");**

**for(int i=1000;i<=SIZE;i+=1000){**

**runmergesort(best\_case\_merge,i);**

**runmergesort(arr,i);**

**runmergesort(worst\_case\_merge,i);**

**runquicksort(arr,i);**

**runquicksortworst(worst\_case\_quick,i);**

**}**

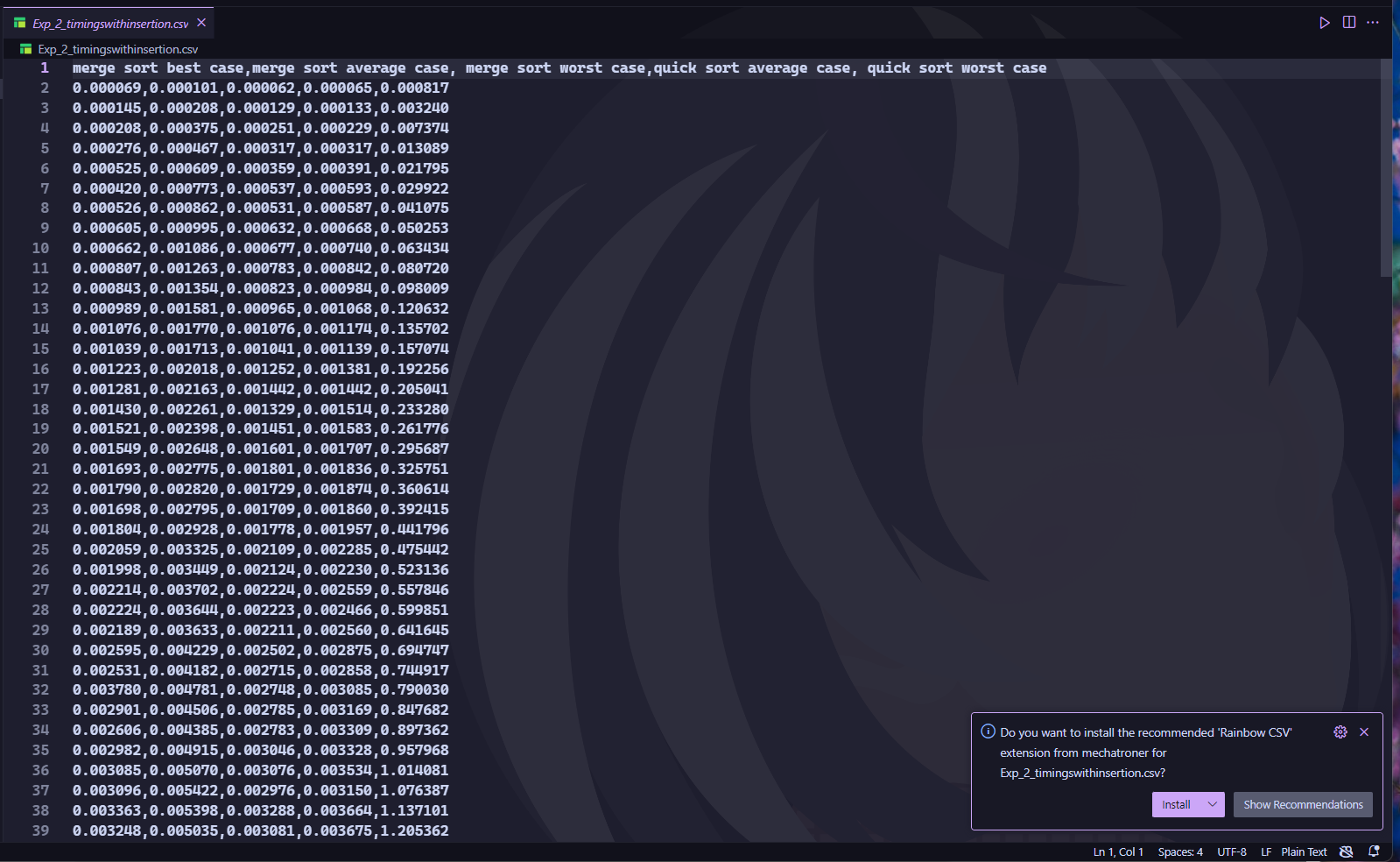
**fclose(file);**

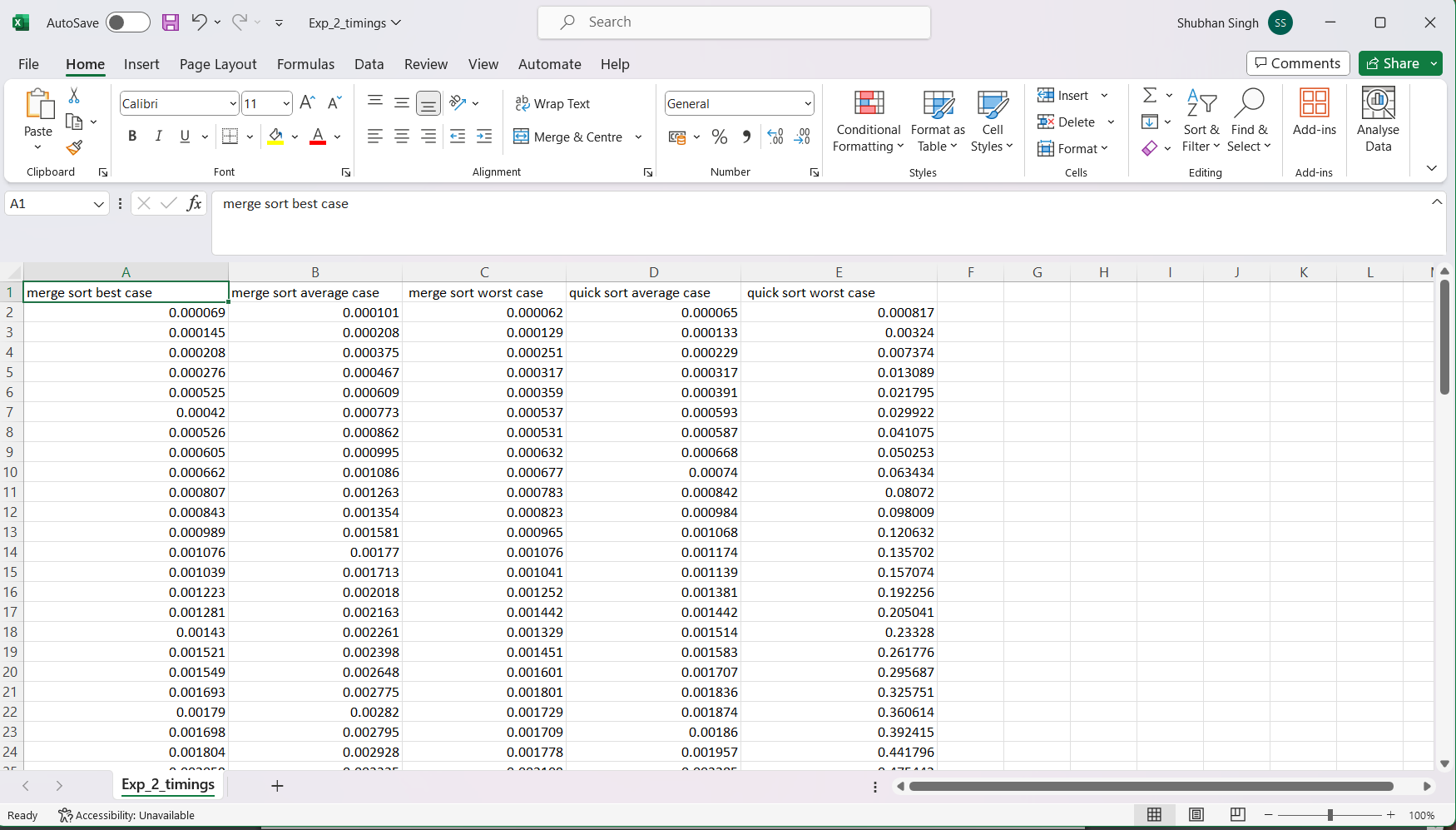
**return 0;**

**}**

**Output:**

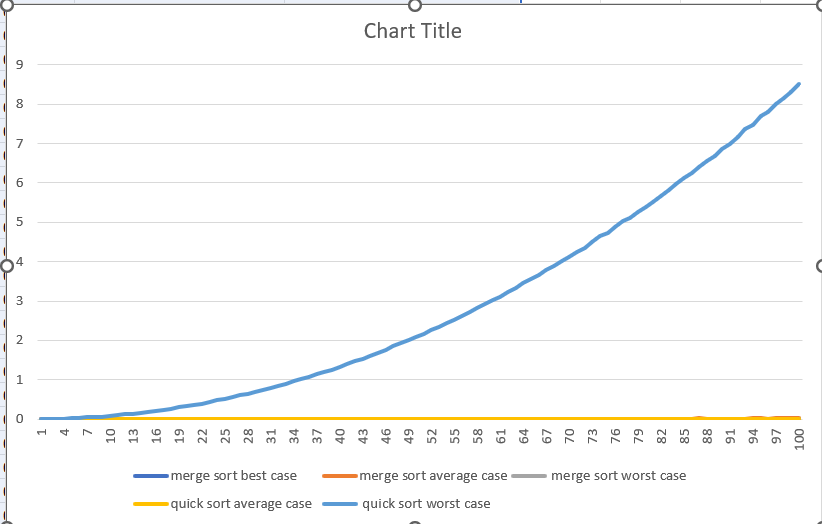
File created





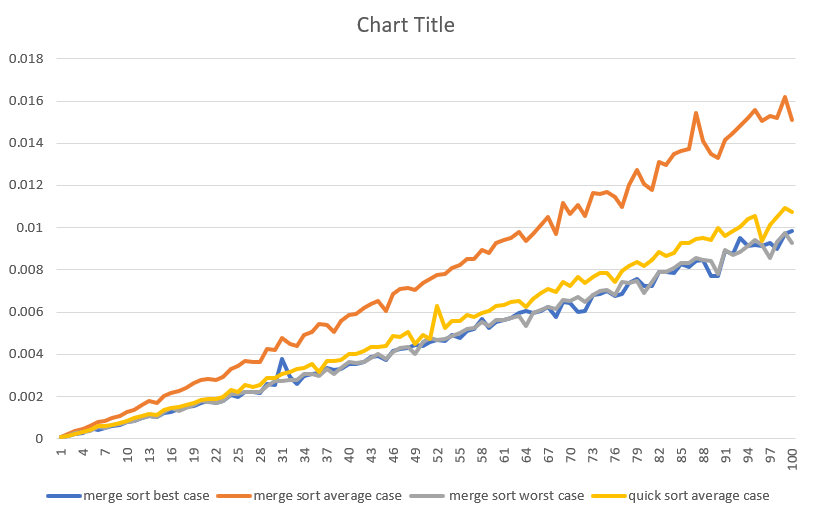
CSV file

**PLOTS:**

****

**Graph of time taken by both algorithms to sort 100 blocks of 1000,2000…100000 elements.**

**The worst case of Quicksort is polynomially slower than the other cases, thus the others just appear as a flat line.**

****

**The other graphs, apart from worst case of quicksort.**

**All are of O(nlogn) time complexity.**

**Conclusion:**

* Since the graphs of the running times average cases of the sorting algorithms with respect to the input size are similar to graphs of function nlogn, it can be said that insertion and selection sort have O(nlogn) average case time complexity.
* Merge sort has O(nlogn) best, worse and average case time complexity.
* Quicksort has O(n^2) time complexity.

**Pseudocode:**

1. Merge function:

Create two temporary arrays L and R

Copy the elements from the original array to these temporary arrays

Initialize two pointers l and r to the start of L and R respectively

For each element from start to end in the original array

If the current element in L is less than the current element in R

Assign the current element in L to the original array

Increment the pointer l

Else

Assign the current element in R to the original array

Increment the pointer r

1. Mergesort function:

If the start index is not equal to the end index (i.e., the array has more than one element)

Calculate the mid index

Recursively call mergesort for the left half of the array

Recursively call mergesort for the right half of the array

Merge the two halves using the merge function

1. Quicksort function:

If the array has more than one element

Choose the first element as the pivot

Initialize two pointers, lower and higher, to the start and end of the array respectively

While the lower pointer is less than the higher pointer

Increment the higher pointer while the current element is less than or equal to the pivot

Decrement the lower pointer while the current element is greater than or equal to the pivot

If the lower pointer is still less than the higher pointer

Swap the elements at the lower and higher pointers

Else

Swap the pivot with the element at the lower pointer and update the pivot index

Recursively call quicksort for the two halves divided by the pivot

**Theory:**

**Quicksort**– It picks an element called as pivot, and then it partitions the given array around the picked pivot element. It then arranges the entire array in two sub-array such that one array holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.The Divide and Conquer steps of Quicksort perform following functions.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort

Combine: Combine the already sorted array.

**Merge sort**– Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the merge() function to perform the merging.The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into twoelement lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

**Time and space complexities of insertion and selection sort:**

**Merge Sort:**

Time Complexity:

* Best, Average, and Worst Case: O(n log n)

Space Complexity: O(n) as it requires temporary arrays during the merge step

Here’s a simple visualisation of the process behind merge sort:

A paper with writing on it

Description automatically generated

**Quick Sort:**

Time Complexity:

* Best and Average Case: O(n log n) when the pivot element is near the median on average
* Worst Case: O(n^2) when the pivot element is the smallest or largest element (e.g., in a sorted or reverse sorted array)
* The worst case can be avoided by using methods like the median of three method to choose the pivot.

Space Complexity: O(log n) for the recursive stack space. It's an in-place sorting algorithm, but recursive calls add to the space complexity.

But it does not require extra arrays.

Here is a simple visualisation of quick sort:

A piece of paper with writing on it

Description automatically generated

As can be seen, for both algorithms, the number of levels of splitting, merging, or partitioning (which is proportional to the time complexity) increases logarithmically with the size of the input, hence the O(n log n) time complexity for both algorithms in the average case. The space complexity for Merge Sort is O(n) because of the temporary arrays used during merging, while Quick Sort has a space complexity of O(log n) due to the recursive stack space.