<table>
<tr><td>NAME:</td><td>Pratham Jain</td></tr>
<tr><td>UID:</td><td>2021300051- COMPS A (C-batch)</td></tr>
<tr><td>SUBJECT</td><td>DAA</td></tr>
<tr><td>EXPERIMENT NO :</td><td>2</td></tr>
<tr><td>DATE OF PERFORMANCE</td><td>13-02-23</td></tr>
<tr><td>DATE OF SUBMISSION</td><td>20-02-23</td></tr>
<tr><td>AIM:</td><td>Compare running time of Merge sort & Quick sort</td></tr>
<tr><td></td><td></td></tr>
<tr><td>THEORY</td><td>
<table>
<tr><th>Basis for comparison</th><th>Quick Sort</th><th>Merge Sort</th></tr>
<tr><td>The partition of elements in the array</td><td>The splitting of a array of elements is in any ratio, not necessarily divided into half.</td><td>In the merge sort, the array is parted into just 2 halves (i.e. n/2).</td></tr>
<tr><td>Worst case complexity</td><td>O(n^2)</td><td>O(nlogn)</td></tr>
<tr><td>Works well on</td><td>It works well on smaller array</td><td>It operates fine on any size of array</td></tr>
<tr><td>Speed of execution</td><td>It work faster than other sorting algorithms for small data set like Selection sort etc</td><td>It has a consistent speed on any size of data</td></tr>
</table>
</td></tr>
</table>

Additional storage space requirement	Less(In-place)	More(not In-place)
Efficiency	Inefficient for larger arrays	More efficient
Sorting method	Internal	External
Stability	Not Stable	Stable
Preferred for	for Arrays	for Linked Lists
Locality of reference	good	poor
Major work	The major work is to partition the array into two sub-arrays before sorting them recursively.	Major work is to combine the two sub-arrays after sorting them recursively.
Division of array	Division of an array into sub-arrays may or may not be balanced as the array is partitioned around the pivot.	Division of an array into sub array is always balanced as it divides the array exactly at the middle.
Method	Quick sort is in- place sorting method.	Merge sort is not in – place sorting method.
	Quicksort does not need explicit merging	Merge sort performs explicit merging of

	arrays; rather the sub-arrays rearranged properly during partitioning.	
Space	Quicksort does not require additional array space.	For merging of sorted sub-arrays, it needs a temporary array with the size equal to the number of input elements.

ALGORITHM

MERGE SORT :

MergeSort(arr[], l, r)
If r > l
Find the middle point to divide the array into two halves:
middle m = l + (r – l)/2
Call mergeSort for first half:
Call mergeSort(arr, l, m)
Call mergeSort for second half:
Call mergeSort(arr, m + 1, r)
Merge the two halves sorted in steps 2 and 3:
Call merge(arr, l, m, r)

QUICK SORT :

partition (arr[], low, high)
```
{
// pivot (Element to be placed at right position)
pivot = arr[high];

i = (low – 1)  // Index of smaller element and indicates the
// right position of pivot found so far

for (j = low; j <= high- 1; j++){

// If current element is smaller than the pivot
if (arr[j] < pivot){
```

```
i++;    // increment index of smaller element
swap arr[i] and arr[j]
}
}
swap arr[i + 1] and arr[high])
return (i + 1)
}

quickSort(arr[], low, high) {

    if (low < high) {

        /* pi is partitioning index, arr[pi] is now at right place */

        pi = partition(arr, low, high);

        quickSort(arr, low, pi – 1);  // Before pi

        quickSort(arr, pi + 1, high); // After pi

    }

}
```

PROGRAM:

QUICK SORT -

```
#include <stdio.h> #include<stdlib.h> #include<time.h> int count=0;

void swap(int *a, int *b) { int t = *a;
*a = *b;
*b = t;
}

// function to find the partition position int partition(int array[], int low, int
high) {
// select the rightmost element as pivot int pivot = array[high];

// pointer for greater element int i = (low - 1);

// traverse each element of the array
// compare them with the pivot for (int j = low; j < high; j++) {
if (array[j] <= pivot) {
```

```
// if element smaller than pivot is found
// swap it with the greater element pointed by i i++;

// swap element at i with element at j swap(&array[i], &array[j]);

count++;
}
}

// swap the pivot element with the greater element at i swap(&array[i + 1],
&array[high]);

// return the partition point return (i + 1);
}

void quickSort(int array[], int low, int high) { if (low < high) {

// find the pivot element such that
// elements smaller than pivot are on left of pivot
// elements greater than pivot are on right of pivot int pi = partition(array,
low, high);

// recursive call on the left of pivot quickSort(array, low, pi - 1);

// recursive call on the right of pivot quickSort(array, pi + 1, high);
}
}

int main()
{
FILE* ptr;
int arr[100000];
// file in reading mode
ptr = fopen("inputFile.txt", "r");

if (NULL == ptr)
{
printf("file can't be opened \n");
}
```

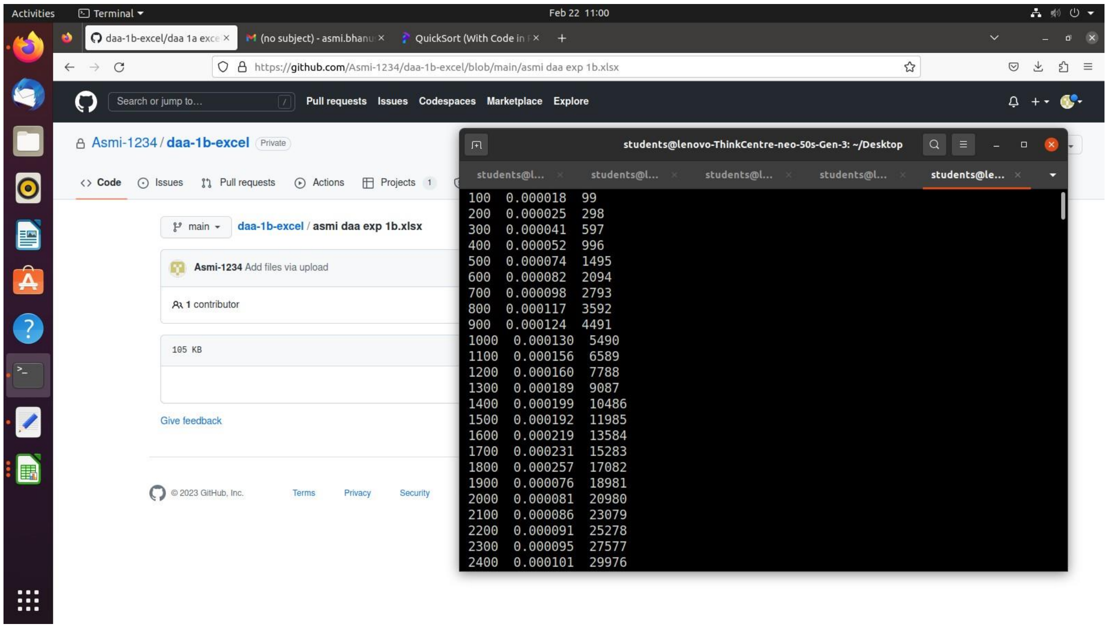
	int block=1; int size=100;
	while(block<=1000)
	{
	int data[size];
	for(int i=0;i<size;i++)
	{
	fscanf(ptr,"%d ",&data[i]);
	//printf("%d ",data[i]);
	
	}
	
	clock_t t; t = clock();
	quickSort(data,0,size-1);
	
	t = clock() - t;
	double time_taken = ((double)t)/CLOCKS_PER_SEC; printf("\n %d %f %d",size,time_taken,count);
	
	size=size+100; block++; fseek(ptr,0,SEEK_SET);
	
	}
	
	fclose(ptr);
	
	}
	
	MERGE SORT –
	
	#include <stdio.h> #include <stdlib.h> #include<time.h> int count=0;
	
	void merge(int arr[], int l, int m, int r)
	{
	int i, j, k;
	int n1 = m - l + 1; int n2 = r - m;

```
int L[n1], R[n2];


for (i = 0; i < n1; i++) L[i] = arr[l + i];
for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];


i = 0;
j = 0;
k = l;

while (i < n1 && j < n2) { if (L[i] <= R[j]) {
arr[k] = L[i]; i++;
}
else {
arr[k] = R[j]; j++;
} k++;
}


while (i < n1) { arr[k] = L[i]; i++;
k++;
}


while (j < n2) { arr[k] = R[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);
```

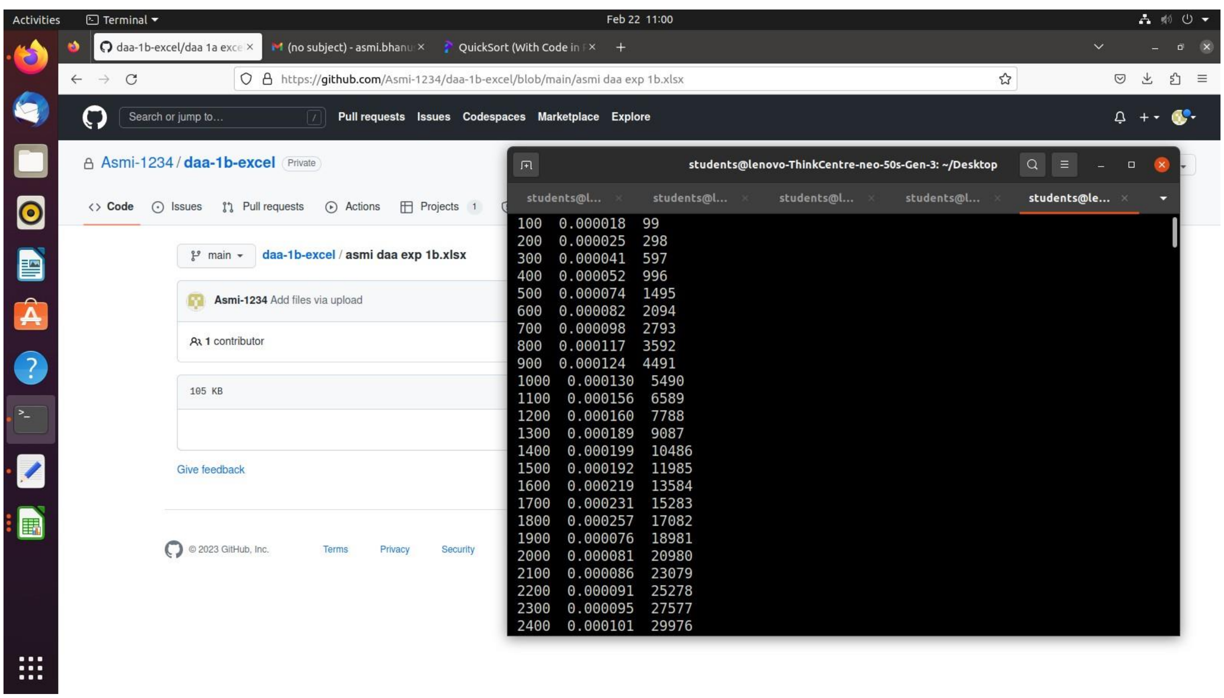
	merge(arr, l, m, r); count++;
	}
	}
	
	void printArray(int A[], int size)
	{
	int i;
	for (i = 0; i < size; i++)
	
	printf("%d ", A[i]); printf("\n");
	}
	
	int main()
	{
	FILE* ptr;
	int arr[100000]; ptr=fopen("inputFile.txt","r"); if(NULL==ptr)
	{
	printf("file cant be opened");
	}
	int block=1; int size=100;
	while(block<=1000)
	{
	int data[size];
	for(int i=0;i<size;i++)
	{
	fscanf(ptr,"%d",&data[i]);
	}
	clock_t t; t=clock();
	mergeSort(arr,0,size-1); t=clock() -t;
	double time_taken =((double)t)/CLOCKS_PER_SEC; printf("\n %d %f %d",block,time_taken,count); size=size+100;
	block++; fseek(ptr,0,SEEK_SET);
	}
	fclose(ptr);
	}

RESULT (SNAPSHOT)	

Merge Sort:



Quick SORT:



CONCLUSION:	I understood the divide and conquer approach with the help of sorting algorithms, namely merge sort and quick sort. I also got a better understanding of their time complexities by monitoring the run time measure. I also calculated the no of merges and no of swaps happened during the run time.