Name	**Bhagya Bijlaney**
UID no.	**2021700010**
Experiment No.	**2**

AIM:	**Experiment on finding the running time of an algorithm (Merge and Quick Sort)**
Program 1	
PROBLEM STATEMENT :	**Implement two sorting algorithms namely merge sort and quicksort methods. Compare these algorithms based on time and space complexity.** **You have to generate 10000 integer numbers using C/C++ Rand function and save them in a text file. Both the sorting algorithms uses these 10000 integer numbers as input as follows. Each sorting algorithm sorts a block of 100 integers numbers with array indexes numbers A[0..99], A[0..199], A[0..299],..., A[0..99999]. Finally, compare two algorithms namely merge sort and quicksort by plotting the time required to sort 100000 integers using** **LibreOffice Calc/MS Excel. The x-axis of 2-D plot represents the block no. of 100 blocks. The y-axis of 2-D plot** **represents the tunning time to sort 100 blocks of 100,200,300,...,10000 integer numbers.**
ALGORITHM/ THEORY:	**Theory:** **MergeSort:** Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array. In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process

is repeated until the entire array is sorted.

It uses the fact that an array with a single element can be assumed to be sorted.

QuickSort:
It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways. For eg, Picking the first/last/middle element as pivot.

The key process in **QuickSort** is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Algorithm:
Merge Sort

```
Step 1: start

Step 2: declare array and left, right, mid variable

Step 3: perform merge function.

    If left > right

        Return
    Mid= (left+right)/2
    Mergesort(array, left, mid)
    Mergesort(array, mid+1, right)
    Merge(array, left, mid, right)
```

Step 4: Stop

Quicksort:

```
quickSort(array, leftmostIndex, rightmostIndex)
  if (leftmostIndex < rightmostIndex)
    pivotIndex <- partition(array,leftmostIndex, rightmostIndex)
    quickSort(array, leftmostIndex, pivotIndex – 1)
    quickSort(array, pivotIndex, rightmostIndex)


partition(array, leftmostIndex, rightmostIndex)
  set rightmostIndex as pivotIndex
  storeIndex <- leftmostIndex – 1
  for i <- leftmostIndex + 1 to rightmostIndex
  if element[i] < pivotElement
    swap element[i] and element[storeIndex]
    storeIndex++
  swap pivotElement and element[storeIndex+1]
return storeIndex + 1
```

PROGRAM:

MergeSort:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

void merge(int arr[], int left, int mid, int right)
{
int n1 = mid - left + 1;
int n2 = right - mid;
int leftArr[n1], rightArr[n2];

for (int i = 0; i < n1; i++)
{
leftArr[i] = arr[left + i];
}
for (int i = 0; i < n2; i++)
{
rightArr[i] = arr[mid + 1 + i];
}

int i = 0, j = 0, k = left;

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

while (i < n1)
{
arr[k] = leftArr[i];
i++;
k++;
}
while (j < n2)
{
arr[k] = rightArr[j];
j++;
k++;
}
}

void mergeSort(int arr[], int left, int right)
{
```

```
if (left < right)
{
int mid = floor((left + right) / 2);
mergeSort(arr, left, mid);
mergeSort(arr, mid + 1, right);
merge(arr, left, mid, right);
}
}

void quickSort(int arr[], int low, int high)
{
if (low >= high)
return;

int left = low, right = high;
int pivotInd = floor((low + high) / 2);
int pivot = arr[pivotInd];

while (left <= right)
{
while (arr[left] < pivot)
left++;
while (arr[right] > pivot)
right--;
if (left > right)
break;

int temp = arr[left];
arr[left] = arr[right];
arr[right] = temp;

left++;
right--;
}

quickSort(arr, low, right);
quickSort(arr, left, high);
}
```

```c
void printArr(int arr[], int len)
{
for (int i = 0; i < len; i++)
{
printf("%d ", arr[i]);
}
printf("\n");
}

void generateRandomSeq(int numbers)
{
FILE *fp = fopen("numbers.txt", "w");
if (fp == NULL)
return;
for (int i = 0; i < numbers; i++)
{
fprintf(fp, "%d\n", rand());
}
fclose(fp);
}

int main()
{
srand(time(NULL));
generateRandomSeq(500000);

FILE *fp = fopen("numbers.txt", "r");

printf("\n\nMerge Sort\n\n");
printf("Size\t\tTime(ms)\n");
for (int i = 100; i <= 100000; i += 100)
{
int arr[i];
for (int j = 0; j < i; j++)
{
fscanf(fp, "%d\n", &arr[j]);
}

clock_t start = clock();
mergeSort(arr, 0, i - 1);
clock_t end = clock();
double time = ((double)(end - start)) /
CLOCKS_PER_SEC;

printf("%6d\t\t%6.3f\n", i, time * 1000);
}
rewind(fp);
fclose(fp);

return 0;
}
```

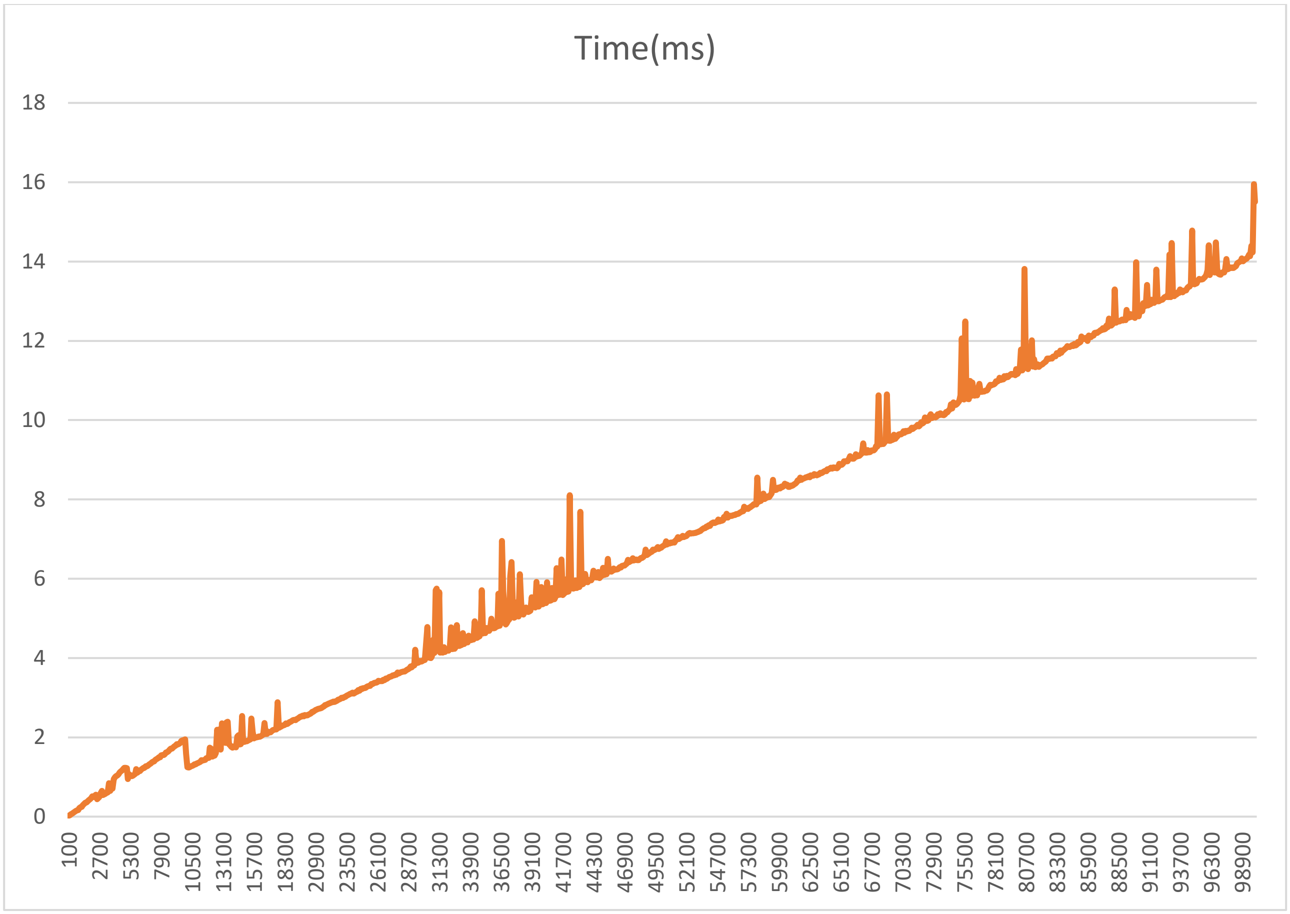
Quicksort:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

void quickSort(int arr[], int low, int high)
{
if (low >= high)
return;

int left = low, right = high;
int pivotInd = floor((low + high) / 2);
int pivot = arr[pivotInd];

while (left <= right)
{
while (arr[left] < pivot)
left++;
while (arr[right] > pivot)
right--;
if (left > right)
break;
int temp = arr[left];
arr[left] = arr[right];
arr[right] = temp;

left++;
right--;
}

quickSort(arr, low, right);
quickSort(arr, left, high);
}

void printArr(int arr[], int len)
{
for (int i = 0; i < len; i++)
{
printf("%d ", arr[i]);
}
printf("\n");
}

void generateRandomSeq(int numbers)
{
FILE *fp = fopen("numbers.txt", "w");
if (fp == NULL)
return;
for (int i = 0; i < numbers; i++)
{
fprintf(fp, "%d\n", rand());
}
fclose(fp);
}

int main()
{
srand(time(NULL));
generateRandomSeq(500000);

FILE *fp = fopen("numbers.txt", "r");
```

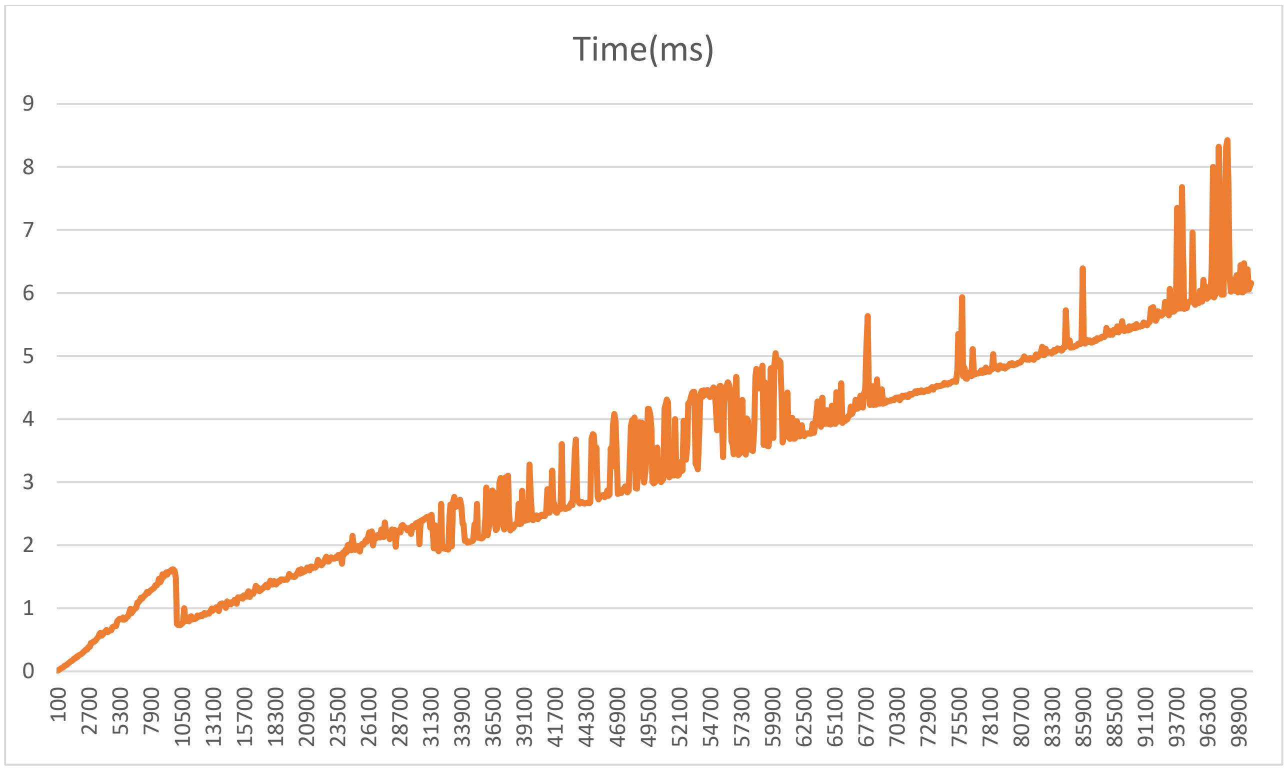
```
rewind(fp);
fclose(fp);

fp = fopen("numbers.txt", "r");

printf("\n\nQuick Sort\n\n");
printf("Size\t\tTime(ms)\n");
for (int i = 100; i <= 100000; i += 100)
{
int arr[i];
for (int j = 0; j < i; j++)
{
fscanf(fp, "%d\n", &arr[j]);
// arr[j] = rand();
}

clock_t start = clock();
quickSort(arr, 0, i - 1);
clock_t end = clock();
double time = ((double)(end - start)) /
CLOCKS_PER_SEC;

printf("%6d\t\t%6.3f\n", i, time * 1000);
}
fclose(fp);

return 0;
}
```

RESULT:

Mergesort performance:



Quicksort performance:



Analysis:	1. It is observed that quicksort is faster than mergesort in this case. Quicksort is faster than merge sort for a completely random dataset. 2. Quicksort performs better here since it is an in place sort so it has log n space complexity(height of recursion tree) while merge sort has O(n) space complexity due to the overhead of merging. 3. Quicksort is also faster as it has better cache locality. 4. Mergesort is a stable sort, so it is better in that regard. 5. Mergesort is better than quicksort when all elements are already sorted, or all are identical. 6. Following are the time & space complexites: Quicksort: Time Best case: O(nlogn), Avg case: O(nlogn), Worst

	case: O(n^2) Space complexity: O(logn) MergeSort: Time Best, avg, worst case: O(nlogn) Space complexity: O(n)
CONCLUSION:	In this experiment, I learnt to sort 2 or more large arrays using merge and quick sort methods.