NAME:	Bhavesh Prashant Chaudhari
UID:	2021300018
SUBJECT	DAA
EXPERIMENT NO :	2
DATE OF PERFORMANCE	9/2/23
DATE OF SUBMISSION	14/2/23
AIM:	Experiment on finding running time of mergesort and quicksort
PROBLEM STATEMENT 1:	
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. QuickSort- Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the

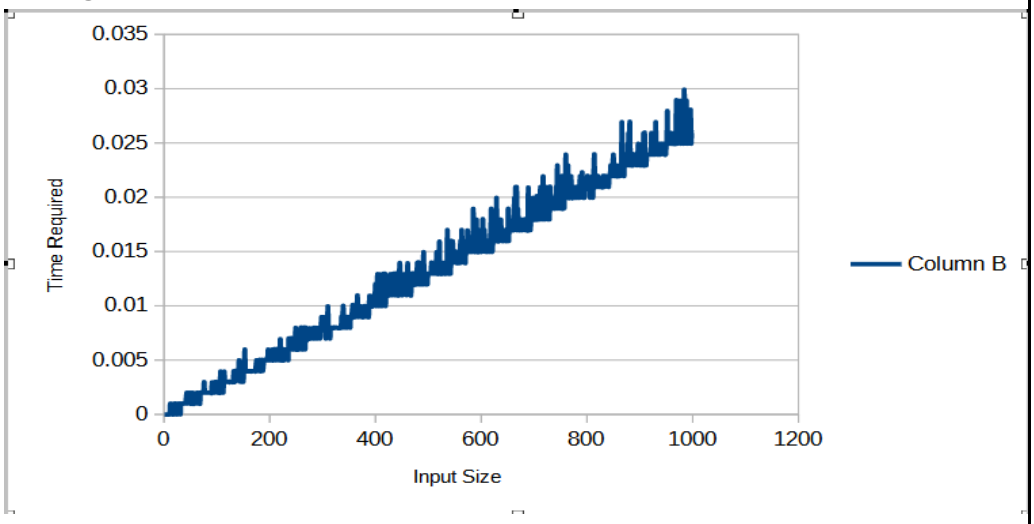
	given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways. Always pick the first element as a pivot. Always pick the last element as a pivot (implemented below) Pick a random element as a pivot. Pick median as the pivot. The key process in quickSort 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	MergeSort- MERGE(A,p,q,r): nL = q-p+1 nR = r - q let L[0:nL-1] and R[0:nR-1] be new arrays for i=0 to nL-1 L[i] = A[p+i] for j=0 to nR-1 R[j] = A[q+j+1] i=0 j=0 k=p while i < nL and j < nR

```
    if L[i] < R[j]
      A[k] = L[i]
      i = i+1
    else
      A[k] = R[j]
      j = j+1
  k = k+1
  while i < nL
    A[k] = L[i]
    i = i+1
    k = k+1
  while j < nR
    A[k] = R[j]
    j = j+1
    k = k+1

MERGE-SORT(A,p,r):
  if p >= r
    return
  q = (q+r)/2
  MERGE-SORT(A,p,q)
  MERGE-SORT(A,q+1,r)
  MERGE(A,p,q,r)



QuickSort-
PARTITION(A,low,high):
  x = A[high]
  i = low-1
```

```
  for j=p to r-1
    if A[j] <= x
      i = i+1
      swap A[i] with A[j]
  swap A[i+1] with A[r]
  return i+1

QUICKSORT(A,p,r):
  if q < r
    q = PARTITION(A,p,r)
    QUICKSORT(A,p,q-1)
    QUICKSORT(A,q+1,r)
```

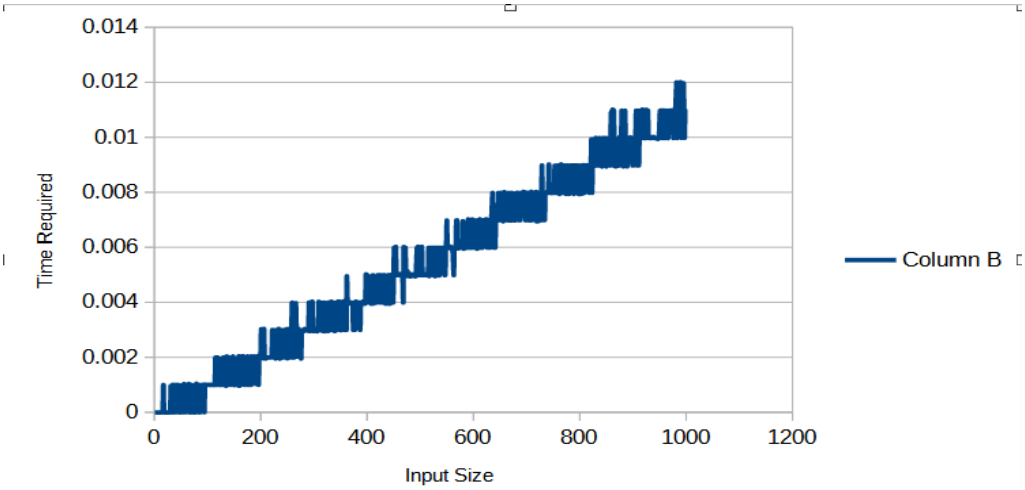
RESULT:

MergeSort-



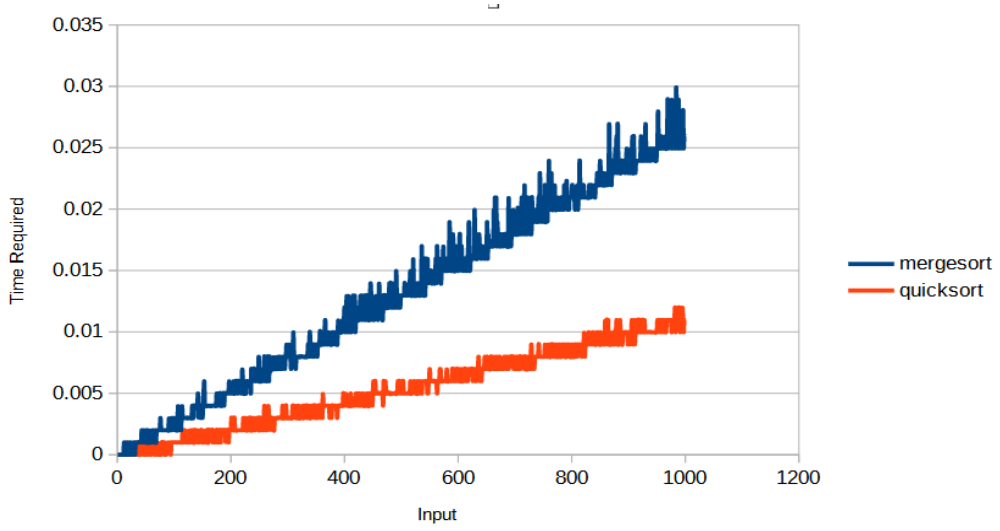
From graph it can be seen that as the size of input increases the time taken for sorting increase in a kind of linear manner.Comparing with the 2 sorting algorithms which we have seen before i.e insertion sort and selection sort merge sort is effecient for high input size.
The time complexity for merge sort is O(nlogn).

QuickSort-



From graph it can be seen that as the size of input increases the time taken for sorting increase in a kind of linear manner.Comparing with the 2 sorting algorithms which we have seen before i.e insertion sort and selection sort merge sort is effecient for high input size.
The time complexity for merge sort is O(nlogn).

MergeSort Vs QuickSort-



	From above graph it can be seen quicksort is bit faster than mergesort.

PROGRAM:	#include<iostream>
	#include<time.h>
	#include<cstdlib>
	#include<fstream>
	#include<iomanip>
	#include<numeric>
	#include<chrono>
	
	using namespace std;
	
	void genNumbers(){
	ofstream fptr("input.txt");
	for(int i=0;i<100000;i++)
	fptr << rand() << endl;
	fptr.close();
	}
	
	void readInput(int *arr,int size){
	ifstream fptr("input.txt");
	int i = 0;
	while(i<=size){
	fptr >> arr[i];
	i++;
	}
	fptr.close();
	}
	
	void merge(int *arr,int low,int high,int mid){
	int left_size = mid-low+1;
	int right_size = high-mid;
	int *left_arr = new int[left_size];

```
        int *right_arr = new int[right_size];
        for(int i=0;i<left_size;i++)
                left_arr[i] = arr[low+i];
        for(int i=0;i<right_size;i++)
                right_arr[i] = arr[mid+i+1];
        int i=0,j=0;
        int k=low;
        while(i<left_size && j<right_size){
                if(left_arr[i] < right_arr[j]){
                        arr[k] = left_arr[i];
                        i++;
                }else{
                        arr[k] = right_arr[j];
                        j++;
                }
                k++;
}
        while(i<left_size){
                arr[k] = left_arr[i];
                i++;
                k++;
}
        while(j<right_size){
                arr[k] = right_arr[j];
                j++;
                k++;
}
        delete[] left_arr;
        delete[] right_arr;
        return;
}

void mergeSort(int *arr,int low,int high){
        if(low < high){
```

```
                int mid = (low+high)/2;
                mergeSort(arr,low,mid);
                mergeSort(arr,mid+1,high);
                merge(arr,low,high,mid);
}
        return;
}

int partition(int *arr,int pivot,int low,int high){
        int i = low-1;
        int j = low;
        while(j<=high){
                if(arr[j] <= arr[pivot]){
                        i++;
                        int temp = arr[i];
                        arr[i] = arr[j];
                        arr[j] = temp;
                }
                j++;
        }
        int temp = arr[i+1];
        arr[i+1] = arr[pivot];
        arr[pivot] = temp;
        return i+1;
}

void quickSort(int *arr,int low,int high){
        if(low < high){
                int pivot = high;
                pivot = partition(arr,pivot,low,high-1);
                quickSort(arr,low,pivot-1);
                quickSort(arr,pivot+1,high);
        }
        return;
```

```
}


void print(int *arr,int size){
        ofstream fp;
        fp.open("sorted.txt");
        for(int i=0;i<size;i++)
                fp << arr[i] << endl;
        fp.close();
}

int main(){
        int *arr = new int[100000];
        ofstream fp;
        fp.open("quickSort_time.txt");
        genNumbers();
        for(int i=1;i<=1000;i++){
                int size = (i*100) - 1;
                readInput(arr,size);
                auto start =
std::chrono::high_resolution_clock::now();
                quickSort(arr,0,size);
                std::chrono::duration<double> time =
std::chrono::high_resolution_clock::now() - start;
        fp << time.count() << endl;
        cout << time.count() << endl;
        }
        print(arr,100000);


return 0;
}
```

CONCLUSION:	Through this experiment I understood about mergesort and quicksort and how important and efficient they are when compared with other sorting algorithm like insertion sort and selection sort.