Name	Parth Gandhi
UID No.	2021300033
Class & Division	S.E. COMPS A (BATCH B)
Experiment No.	2

Aim: Experiment on finding the running time of an algorithm(merge sort and quick sort)

Theory:

Merge sort

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.
One thing that you might wonder is what is the specialty of this algorithm. We already have a number of sorting algorithms then why do we need this algorithm? One of the main advantages of merge sort is that it has a time complexity of O(n log n), which means it can sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort.

Quick sort

Like Merge Sort, Quick sort 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:

Merge sort:

1. If the array “b” has only one element, return the array as it is already sorted.
2. Calculate the middle index of the array “b” using "mid = (beg + end) / 2".
3. Call the "mergesort" function recursively for the first half of the array "a[beg, mid]"
4. Call the "mergesort" function recursively for the second half of the array "a[mid+1, end]"
5. Call the "merge" function to merge the two sorted arrays obtained from the previous steps back into the original array “b”.
6. The "merge" function takes in two arrays, the first half and the second half, and sorts the elements in both arrays and stores them back into the original array “b”.
7. Repeat the above steps until all elements of the array “b” are sorted in ascending order.

Quick sort:

1. If the array “b” has zero or one element, return the array as it is already sorted.
2. Choose the first element of the array “b” as the pivot.
3. Initialize two variables “low” and “high” to keep track of the elements to be swapped. Set “low” to the first position and “high” to the last position in the array “b.”
4. While “low” is less than “high,” repeat the following steps: a. Increment “low” while the element at

"low" is less than or equal to the pivot. b. Decrement "high" while the element at "high" is greater than the pivot. c. If "low" is less than "high", swap the elements at "low" and "high".

5. Swap the pivot with the element at "high" to place the pivot in its correct position in the sorted array.
6. Call the quick sort algorithm recursively for the two sub-arrays "b[beg, high-1]" and "b[high+1, end]".
7. Repeat the above steps until all elements of the array "b" are sorted in ascending order

Code:

```
#include<bits/stdc++.h>
#include<chrono>
#include<fstream>
using namespace std;

void merge(int a[],int beg,int mid,int end)
{
   int l1=mid-beg+1;
   int l2=end-mid;
   int left[l1],right[l2];
   for(int i=0;i<l1;i++)
   left[i]=a[beg+i];
   for(int i=0;i<l2;i++)
   right[i]=a[mid+1+i];
   int leftind=0,rightind=0,ind=beg;
   while(leftind<l1 && rightind<l2)
   {
      if(left[leftind]<=right[rightind])
      a[ind]=left[leftind++];
      else
      a[ind]=right[rightind++];
      ind++;
   }
   while(leftind<l1)
   {
      a[ind]=left[leftind++];
      ind++;
   }
   while(rightind<l2)
   {
      a[ind]=right[rightind++];
      ind++;
   }
}

void mergesort(int a[],int beg,int end)
{
   if(beg<end)
   {
      int mid=(beg+end)/2;
```

```
        mergesort(a,beg,mid);
        mergesort(a,mid+1,end);
        merge(a,beg,mid,end);
    }
}

void quick(int b[],int beg,int end)
{
    if(beg>end)
    return;
    int pivot=beg,low=beg+1,high=end;
    while(pivot!=high)
    {
        if(b[low]<=b[pivot])
        low++;
        else
        {
            if(b[high]>b[pivot])
            high--;
            else
            {
                if(low<high)
                swap(b[low],b[high]);
                else
                {
                    swap(b[pivot],b[high]);
                    break;
                }
            }
        }
    }
    quick(b,beg,high-1);
    quick(b,high+1,end);
}

int main()
{
    cout<<"n\tMergeSort\tQuickSort";
    for(int n=100;n<=100000;n+=100)
    {
        int a[n],b[n];
        for(int i=0;i<n;i++)
        a[i]=rand()%n;
        copy(a,a+n,b);

        //Merge Sort
        auto start = chrono::high_resolution_clock::now();
        mergesort(a,0,n-1);
```

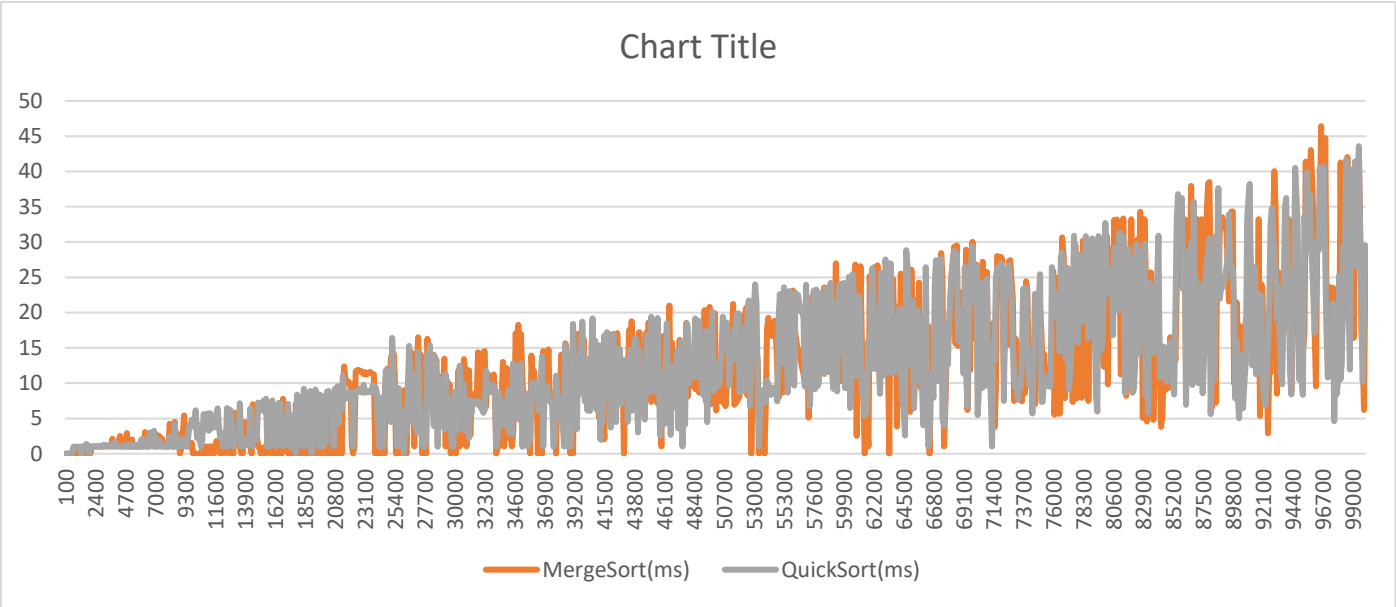
```
        auto end = chrono::high_resolution_clock::now();
        cout<<"\n"<<n<<"\t"<<chrono::duration_cast<chrono::nanoseconds>(end - start).count()<<"\t";

        //Quick Sort
        start = chrono::high_resolution_clock::now();
        quick(b,0,n-1);
        end = chrono::high_resolution_clock::now();
        cout<<chrono::duration_cast<chrono::nanoseconds>(end - start).count();
    }
    return 0;
}
```

Observation table:

n	MergeSort(ms)	QuickSort(ms)
100	0	0
200	0	0
300	0	0
400	0	0
500	0	0
600	0	0
700	0	0.985
800	0.441	1
900	0	0.999
1000	0	0.989
1100	0	1.007
1200	0	0.989
1300	1.008	0
1400	1.037	1.018
1500	0	1.034
1600	1.037	0.997
1700	0	1.395
1800	1.037	0.991
1900	1.03	0
2000	0	0.971
2100	1.031	1.042
2200	1.037	0.991
2300	0.996	1.046
2400	1.047	0.983
2500	1.03	1.043
2600	1.002	1.046
2700	0.995	1.135
2800	1.038	1.032
2900	1.031	1.038
3000	1.002	1.24
3100	1.038	1.203
3200	1.038	1.04
3300	1.039	1.203
3400	1.037	1.157
3500	1.038	1.206
3600	1.038	1.298

3700	2.1	1.038
3800	1.042	1.038
3900	1.037	1.689
4000	1.041	1.037
4100	1.04	1.04
4200	1.845	1.042
4300	2.518	1.041
4400	1.038	1.034
4500	1.039	1.035
4600	1	1.234
4700	1.036	1.037
4800	2.95	1.247
4900	1.001	1.027
5000	1.001	1.237



Observation: Merge sort takes less time to sort than quick sort.

Conclusion: Successfully wrote a program to implement merge and quick sort. Made relevant observations and found the running time of both sorting methods. We can conclude that merge sort is faster.