NAME:	Eshan Bhuse
UID:	2021300013
SUBJECT	Design and analysis of algorithm
EXPERIMENT NO:	2
AIM:	**To find out running time of 2 sorting algorithms like Merge sort and Quick sort.**
ALGORITHM	Main function: step 1: start Step2: call generate_numbers() function Step 2: call operation()function Step 3: end generate_numbers() function: step 1: start step 2: create the file pointer step 3: open the file in writing mode step 3: starts the loop from 0 to 100000 step 4: insert the 100000 random numbers in the file step 5: close the file handle step 6: end operation function(): step 1: start step 2: open the file in reading mode step 3: start the loop from 0 to 100000 and increment it with 100 step 4: create two arrays step 5: start the loop from 0 to j and scan the data from file step 6: before sorting store the time step 7: perform selection sort step 8: check the time after after the sorting step 9: calculate the time taken by the algorithm step 10: before sorting store the time step 11: perform selection sort

	step 12: check the time after after the sorting
	step 13: calculate the time taken by the algorithm
	
	Merge Sort:
	MERGE_SORT (arr, beg, end)
	
	if beg < end
	1. set mid = (beg + end)/2
	2. MERGE_SORT(arr, beg, mid)
	3. MERGE_SORT(arr, mid + 1, end)
	4. MERGE (arr, beg, mid, end)
	end of if
	
	END MERGE_SORT
	
	void merge (int a[], int beg, int mid, int end)
	1. int i, j, k;
	2. int n1 = mid - beg + 1;
	3. int n2 = end - mid;
	4. int LeftArray[n1], RightArray[n2];
	5. for (int i = 0; i < n1; i++)
	6. LeftArray[i] = a[beg + i];
	7. for (int j = 0; j < n2; j++)
	8. RightArray[j] = a[mid + 1 + j];
	9. Declare: i = 0,
	j = 0;
	k = beg;
	10. while (i < n1 && j < n2)
	11. if(LeftArray[i] <= RightArray[j])
	
	Perform:
	
	a[k] = LeftArray[i];
	increment i
	else
	
	Perform:
	
	a[k] = RightArray[j];
	increment j
	12. increment k
	13. while (i<n1)

	a[k] = LeftArray[i];
	increment I and k
	14. while (j<n2)
	a[k] = RightArray[j];
	increment j and k
	Quick Sort:
	
	QUICKSORT (array A, start, end)
	
	if (start < end)
	p = partition(A, start, end)
	recursively call,
	QUICKSORT (A, start, p - 1)
	QUICKSORT (A, p + 1, end)
	
	PARTITION (array A, start, end)
	pivot = A[end] OR a[start] OR random index OR Median
	i = start-1
	for j = start to end -1 {
	do if (A[j] < pivot) {
	then i = i + 1
	swap A[i] with A[j]
	swap A[i+1] with A[end]
	return i+1

THEORY:	**Merge sort Algorithm:** **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. Merge sort is a popular choice for sorting large datasets because it is relatively efficient and easy to implement. It is often used in conjunction with other algorithms, such as quicksort, to improve the overall performance of a sorting routine **Advantages of Merge Sort:** • Merge sort has a time complexity of O(n log n), which means it is relatively efficient for sorting large datasets. • Merge sort is a stable sort, which means that the order of elements with equal values is preserved during the sort. • It is easy to implement thus making it a good choice for many applications. • It is useful for external sorting. This is because merge sort can handle large datasets, it is often used for external sorting, where the data being sorted does not fit in memory. • The merge sort algorithm can be easily parallelized, which means it can take advantage of multiple processors or cores to sort the data more quickly. • Merge sort requires relatively few additional resources (such as memory) to perform the sort. This makes it a good choice for systems with limited resources.

Drawbacks of Merge Sort:

- Slower compared to the other sort algorithms for smaller tasks. Although efficient for large datasets its not the best choice for small datasets.
- The merge sort algorithm requires an additional memory space of 0(n) for the temporary array. This is to store the subarrays that are used during the sorting process.
- It goes through the whole process even if the array is sorted.
- It requires more code to implement since we are dividing the array into smaller subarrays and then merging the sorted subarrays back together.

Merge sort complexity:

Now, let's see the time complexity of merge sort in best case, average case, and in worst case. We will also see the space complexity of the merge sort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n*logn)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **O(n*logn)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **O(n*logn)**.

Quick Sort Algorithm:

In this article, we will discuss the Quicksort Algorithm. The working procedure of Quicksort is also simple. This article will be very helpful and interesting to students as they might face quicksort as a question in their examinations. So, it is important to discuss the topic.

Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

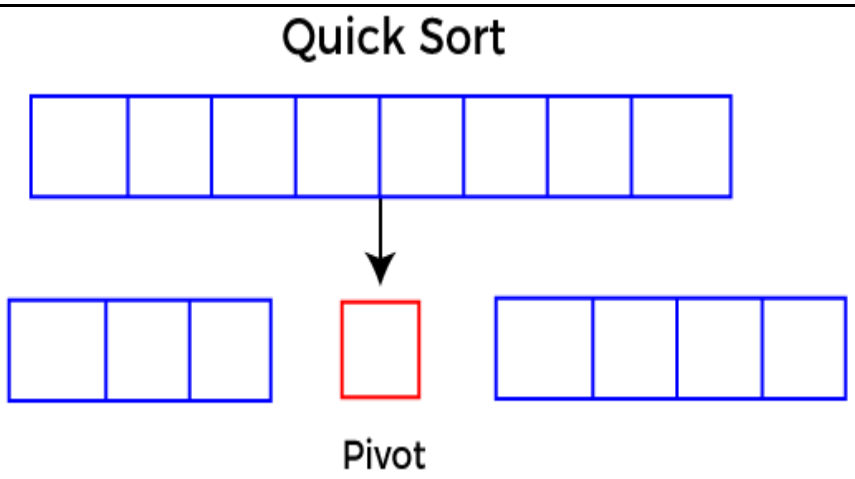
Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Advantages of Quick Sort:

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- It has a low overhead, as it only requires a small amount of memory to function.
-

Disadvantages of Quick Sort:

- It has a worst-case time complexity of O(n^2), which occurs when the pivot is chosen poorly.
- It is not a good choice for small data sets.
- It can be sensitive to the choice of pivot.
- It is not cache-efficient.
- It is not stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we swapping of elements according to pivot's position (without considering their original positions).

Quicksort complexity

Now, let's see the time complexity of quicksort in best case, average case, and in worst case. We will also see the space complexity of quicksort.

1. Time Complexity

Case	Time Complexity
Best Case	O (n* logn)
Average Case	O (n* logn)
Worst Case	$O(n^2)$

- **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **O(n*logn)**.
- **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **$O(n^2)$**.

Though the worst-case complexity of quicksort is more than other sorting algorithms such as **Merge sort** and **Heap sort**, still it is faster in practice. Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quicksort can be avoided by choosing the right pivot element.

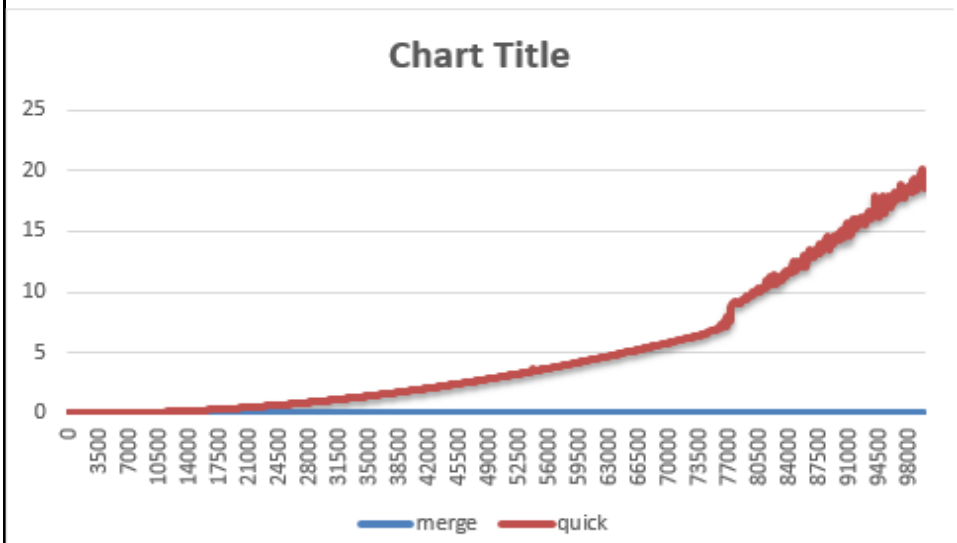
PROGRAM:

```c
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<time.h>
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];
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;

}
}
void quicksort(int arr[],int first,int last)
{
int i, j, pivot, temp;
if(first<last)
{
pivot=first;
while(i<j)
{
    while(arr[i]<=arr[pivot]&&i<last)
    i++;
while(arr[j]>arr[pivot])
j--;
if(i<j)
{
temp=arr[i];
arr[i]=arr[j];
arr[j]=temp;
}
}
arr[pivot]=arr[j];
arr[j]=temp;
}

}
```

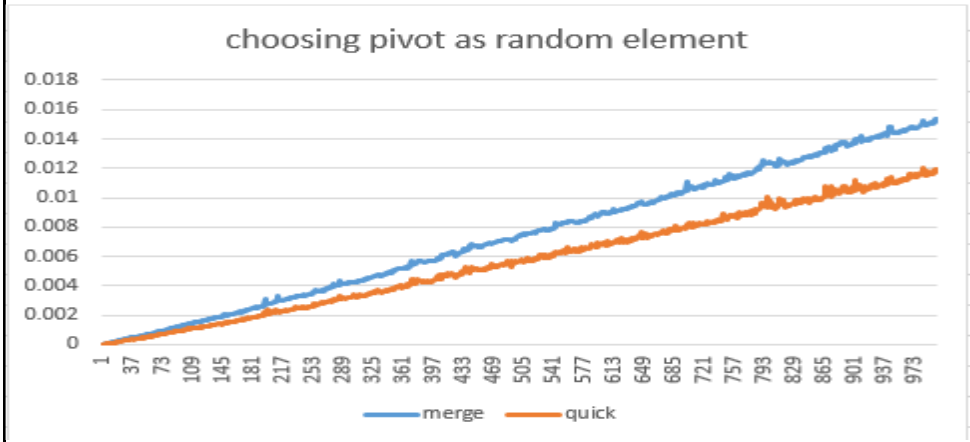
```
}

void generate_arr()
{

FILE *ptr;
ptr=fopen("number.txt","w");
{
fprintf(ptr,"%d\n",rand() % 100000);
}
fclose(ptr);
}
void call()
{
FILE *ptr;
int i=1;
ptr=fopen("number.txt","r");
for(int j=0;j<100000;j+=100)
{
int arr1[j]; int arr2[j];
for(int i=0;i<j;i++)
{
fscanf(ptr,"%d\n",&arr1[i]);
}
for(int i=0;i<j;i++)
{
arr2[i]=arr1[i];
}
clock_t s=clock();
double currs=(double)(clock()-s)/CLOCKS_PER_SEC;

quicksort(arr2,0,j);
double curri=(double)(clock()-i)/CLOCKS_PER_SEC;
printf("\n%d %d %f %f",i++,j,currs,curri);
}
}
int main()
{
generate_arr(); call();
return 0;
}
```

OBSERVATION:

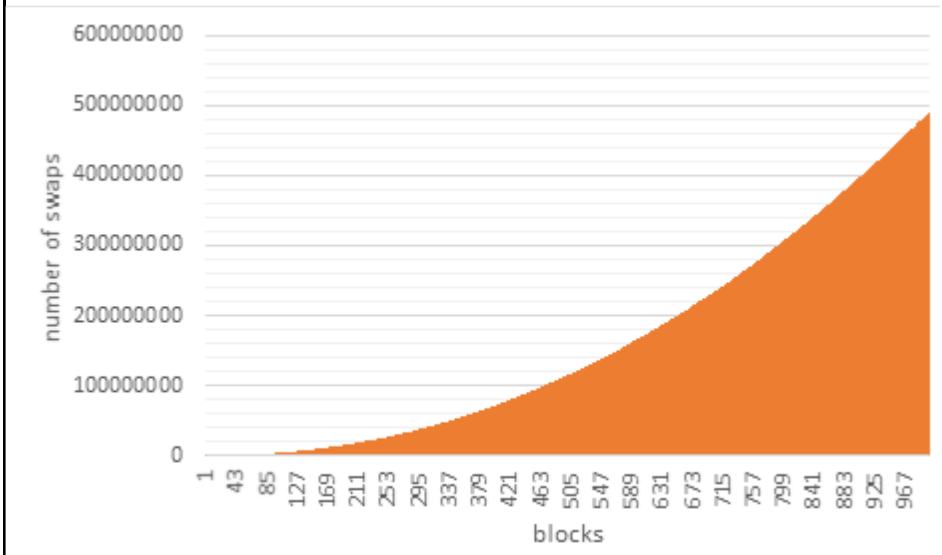


Quick sort: The graph of quick sort is a slightly linear and increases after the given block and time taken for the output of the program is more than the merge sort.

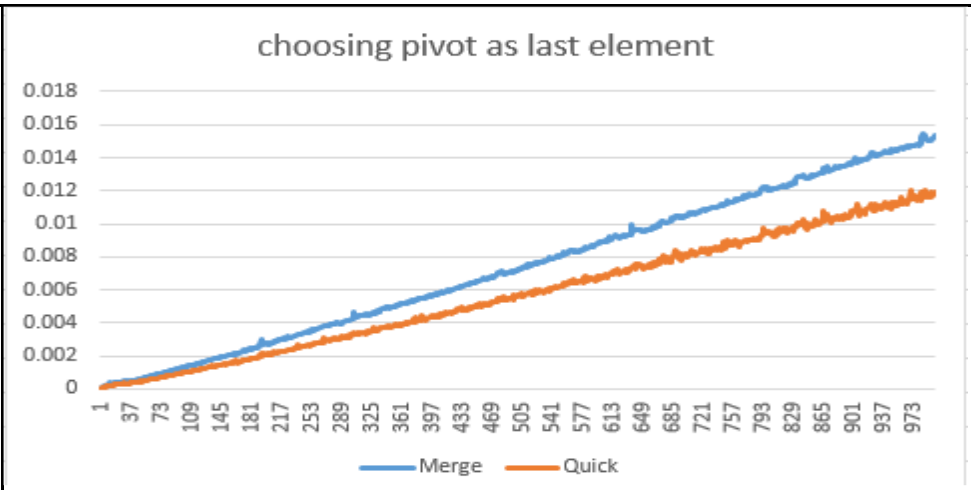
Merge sort: The graph of merge sort is linear which is a straight line and time taken for the output of the program is less than quick sort and the graph can vary depend on the upon hardware configuration, operating system, memory, cpu speed etc. Overall both take about the same time with negligible difference but on my system, merge sort performs slightly better.



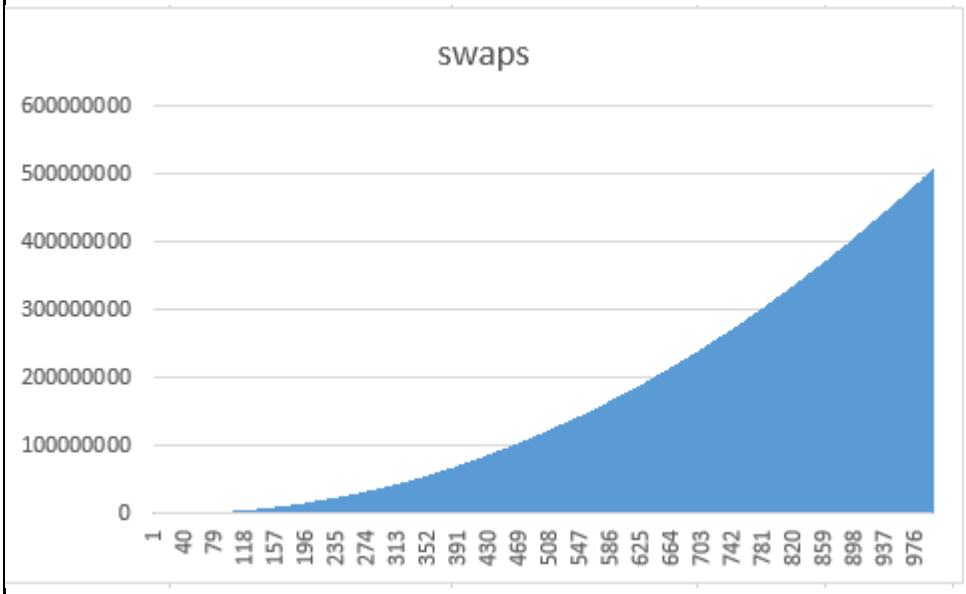
In the above graph, it is seen that merge sort takes take time around 0 seconds and as input size increases time taken to do sorting also increases. But as we took pivot as random element so, time taken drastically decreases. If we take pivot as lowest or highest element or list is already sorted or reversed, then it will be worst case situation for the quick sort. Because, In partition, only sub-array will be weighted at only 1 side. So, it best to take pivot as middle element or random element. In above case, quick sort is seen that it is more efficient than merge sort. This is not possible in every case.



As we observe as we add more numbers at each iteration to the block the number of swaps required increase significantly reaching hundreds of millions. Linear increase is observed.



In above case, quick sort is seen that it is more efficient than merge sort. This is not possible in every case. The order of growth of increase in time taken is little bit linear, both algorithm's time increases and as input size increases merge sort takes more and more time.



In above bar graph, number of swaps increases as input size increases, number of swaps are almost same in the merge sort and quick sort and quick sort takes in place swapping where extra space not required while swapping.

CONCLUSION:	Successfully performed the experiment of merge sort and quick sort in C Language. And with the help of the graph and as well as through the output of the code, merge sort is more efficient than quick sort.