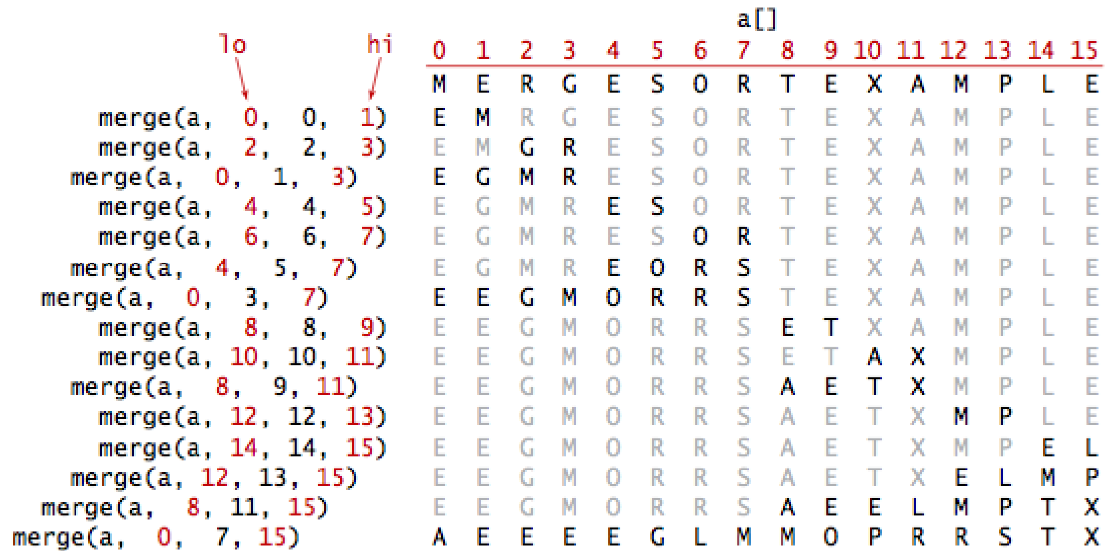
Name	**Advait Ravi Sapkal**
UID No.	**2021700055**
Experiment No.	**1 B**

AIM:	Experiment based on the divide and conquer approach.
Program 1	
PROBLEM STATEMENT:	**Details** – Divide and conquer algorithm is a strategy of solving a large problem by breaking the problem into smaller sub-problems. The divide-and-conquer strategy solves a problem by: **Quicksort**– It picks an element called as pivot, and then it partitions the given array around the picked pivot element. It then arranges the entire array in two sub-array such that one array holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.The Divide and Conquer steps of Quicksort perform following functions. 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. **Merge sort**– Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the merge() function to perform the merging.The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list. -- **Problem Definition & Assumptions** – For this experiment, you need to implement two sorting algorithms namely Quicksort and Merge sort methods. Compare these algorithms based on time and space complexity. Time required for sorting algorithms can be performed using high_resolution_clock::now() under namespace std::chrono. You have to generate 1,00,000 integer numbers using C/C++ Rand function and save them in a text file. Both the sorting algorithms uses these 1,00,000 integer numbers as input as follows. Each sorting algorithm sorts a block of 100,200,300,…,100000 integer numbers with array indexes numbers A[0..99], A[100..199], A[200..299],…, A[99900..99999]. You need to use high_resolution_clock::now() function to find the time required for 100, 200, 300…. 100000 integer numbers. Finally, compare two algorithms namely Quicksort and Merge sort by plotting the time required to sort integers using LibreOffice Calc/MS Excel. The x-axis of 2-D plot represents the block no. of 1000 blocks. The y-axis of 2-D plot represents the tunning time to sort 1000 blocks of 100,200,300,…,100000 integer numbers. Note – You have to use C/C++ file processing functions for reading and writing randomly generated 100000 integer numbers.
ALGORITHM :	Merge and Quick sort are the 2 most important sorting algorithms. Both make use of the divide-and-conquer strategy of algorithm design. Merge sort is a stable sorting algorithm whereas Quick sort is unstable. Merge sort is not an in-place sort, whereas Quick sort is in-place. Quick sort is listed among the top 10 algorithms of the 20th century. **Merge Sort:** **Invariant:** Sorting an array means dividing the array into two parts, sorting each part and merging the two parts in a sorted manner. **Algorithm:**

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
1  n_1 = q − p + 1
2  n_2 = r − q
3  let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
4  for i = 1 to n_1
5      L[i] = A[p + i − 1]
6  for j = 1 to n_2
7      R[j] = A[q + j]
8  L[n_1 + 1] = ∞
9  R[n_2 + 1] = ∞
10 i = 1
11 j = 1
12 for k = p to r
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
```

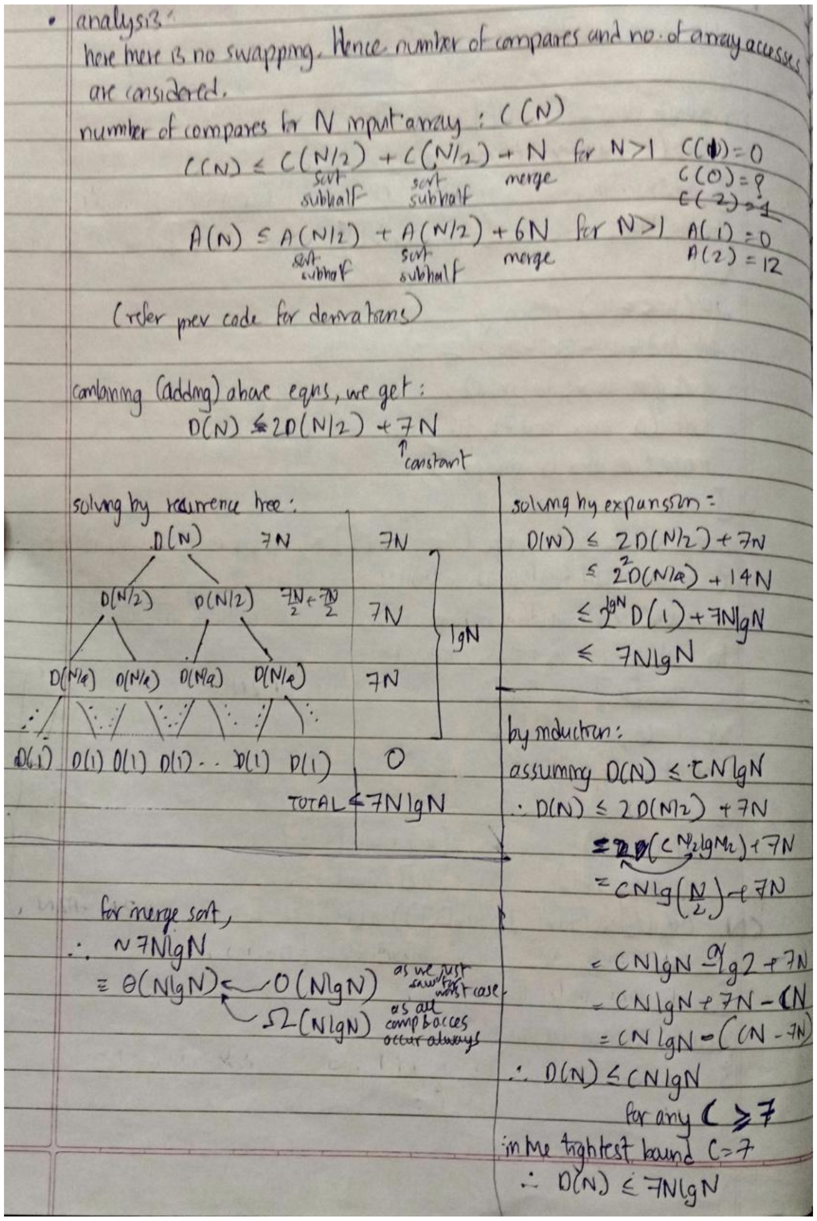
Trace:



```
                                                  a[]
                  lo        hi     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
                                   M  E  R  G  E  S  O  R  T  E  X  A  M  P  L  E
      merge(a,  0,  0,  1)         E  M  R  G  E  S  O  R  T  E  X  A  M  P  L  E
      merge(a,  2,  2,  3)         E  M  G  R  E  S  O  R  T  E  X  A  M  P  L  E
    merge(a,  0,  1,  3)           E  G  M  R  E  S  O  R  T  E  X  A  M  P  L  E
      merge(a,  4,  4,  5)         E  G  M  R  E  S  O  R  T  E  X  A  M  P  L  E
      merge(a,  6,  6,  7)         E  G  M  R  E  S  O  R  T  E  X  A  M  P  L  E
    merge(a,  4,  5,  7)           E  G  M  R  E  O  R  S  T  E  X  A  M  P  L  E
  merge(a,  0,  3,  7)             E  E  G  M  O  R  R  S  T  E  X  A  M  P  L  E
      merge(a,  8,  8,  9)         E  E  G  M  O  R  R  S  E  T  X  A  M  P  L  E
      merge(a, 10, 10, 11)         E  E  G  M  O  R  R  S  E  T  A  X  M  P  L  E
    merge(a,  8,  9, 11)           E  E  G  M  O  R  R  S  A  E  T  X  M  P  L  E
      merge(a, 12, 12, 13)         E  E  G  M  O  R  R  S  A  E  T  X  M  P  L  E
      merge(a, 14, 14, 15)         E  E  G  M  O  R  R  S  A  E  T  X  M  P  E  L
    merge(a, 12, 13, 15)           E  E  G  M  O  R  R  S  A  E  T  X  E  L  M  P
  merge(a,  8, 11, 15)             E  E  G  M  O  R  R  S  A  E  E  L  M  P  T  X
merge(a,  0,  7, 15)               A  E  E  E  E  G  L  M  M  O  P  R  R  S  T  X
```

Trace of merge results for top-down mergesort

Analysis:



- analysis:

here there is no swapping. Hence number of compares and no. of array accesses are considered.

number of compares for N input array : $C(N)$

$C(N) \le \underbrace{C(N/2)}_{\text{sort subhalf}} + \underbrace{C(N/2)}_{\text{sort subhalf}} + \underbrace{N}_{\text{merge}}$ for $N > 1$ $C(1) = 0$, $C(0) = ?$, ~~$C(2) = 1$~~

$A(N) \le \underbrace{A(N/2)}_{\text{sort subhalf}} + \underbrace{A(N/2)}_{\text{sort subhalf}} + \underbrace{6N}_{\text{merge}}$ for $N > 1$ $A(1) = 0$, $A(2) = 12$

(refer prev code for derivations)

combining (adding) above eqns, we get:

$D(N) \le 2D(N/2) + \underset{\uparrow \text{constant}}{7N}$

solving by recurrence tree:

```
              D(N)                 7N        |  7N   ⎤
            /      \                         |       ⎥
       D(N/2)       D(N/2)      7N/2 + 7N/2  |  7N   ⎥
       /    \       /    \                   |       ⎬ lgN
  D(N/4) D(N/4) D(N/4) D(N/4)                |  7N   ⎥
  ...                                        |       ⎦
D(1) D(1) D(1) D(1) .. D(1) D(1)             |  0
                                     TOTAL ≤ 7NlgN
```

for merge sort,

$\therefore \sim 7N\lg N$

$\equiv \Theta(N\lg N) \leftarrow O(N\lg N)$ as we just saw for worst case

$\leftarrow \Omega(N\lg N)$ as all comp & acces occur always

solving by expansion:

$D(N) \le 2D(N/2) + 7N$

$\le 2^2 D(N/4) + 14N$

$\le 2^{\lg N} D(1) + 7N\lg N$

$\le 7N\lg N$

by induction:

assuming $D(N) \le cN\lg N$

$\therefore D(N) \le 2D(N/2) + 7N$

$\le 2(c\frac{N}{2}\lg\frac{N}{2}) + 7N$

$= cN\lg\left(\frac{N}{2}\right) + 7N$

$= cN\lg N - cN\lg 2 + 7N$

$= cN\lg N + 7N - cN$

$= cN\lg N - (cN - 7N)$

$\therefore D(N) \le cN\lg N$

for any $c \ge 7$

in the tightest bound $c = 7$

$\therefore D(N) \le 7N\lg N$

Quick Sort:

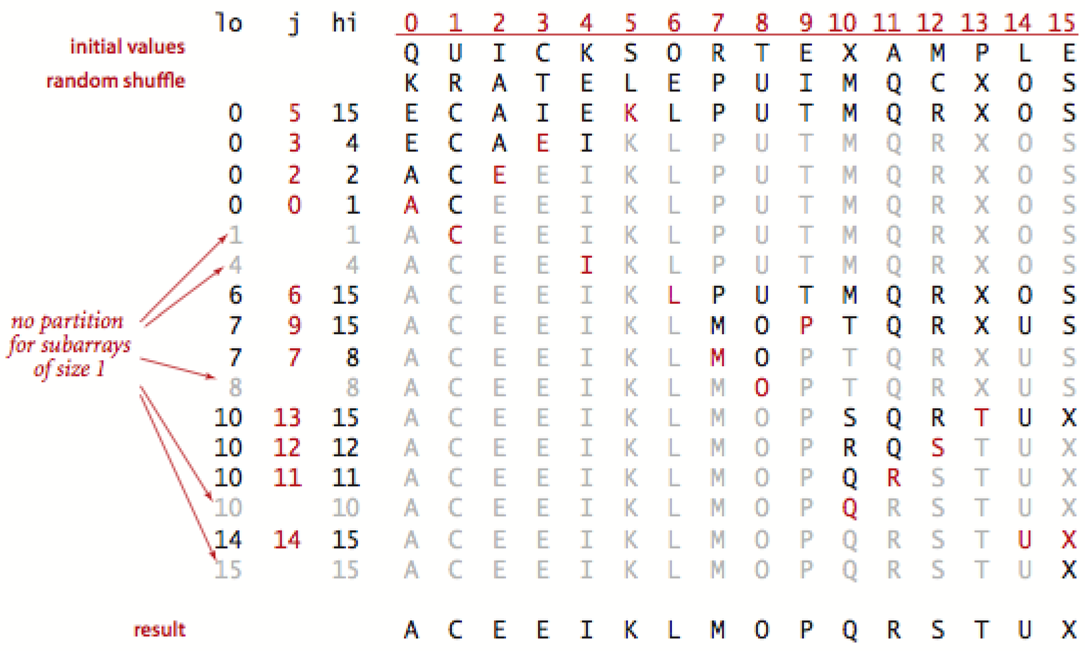
Invariant: Sorting an array means choosing a pivot, placing it such that all elements before it is smaller and all elements after it are bigger, and then sorting these preceding and succeeding subarrays.

Algorithm:

```
QUICKSORT(array $A$, int $p$, int $r$)
1  if ($p < r$)
2      then $q \leftarrow$ PARTITION($A, p, r$)
3           QUICKSORT($A, p, q-1$)
4           QUICKSORT($A, q+1, r$)
```

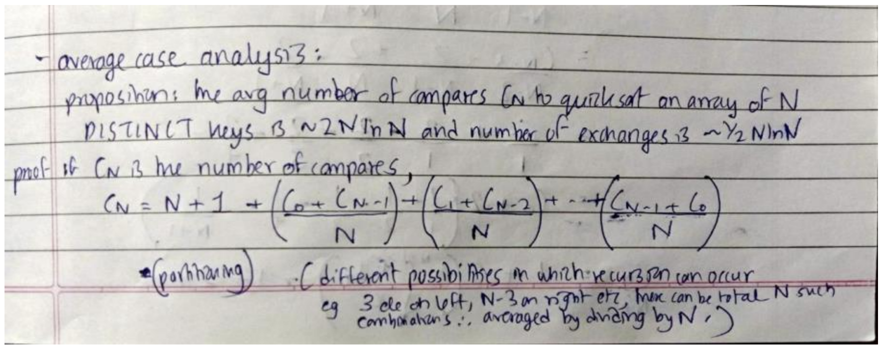
```
PARTITION(array $A$, int $p$, int $r$)
1  $x \leftarrow A[r]$                    ▷ Choose pivot
2  $i \leftarrow p-1$
3  for $j \leftarrow p$ to $r-1$
4      do if ($A[j] \leq x$)
5          then $i \leftarrow i+1$
6               exchange $A[i] \leftrightarrow A[j]$
7  exchange $A[i+1] \leftrightarrow A[r]$
8  return $i+1$
```

Trace:



	lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
random shuffle				K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	1		1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	4		4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	8		8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X
	10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X
	10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	10		10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	15		15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result				A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Analysis:



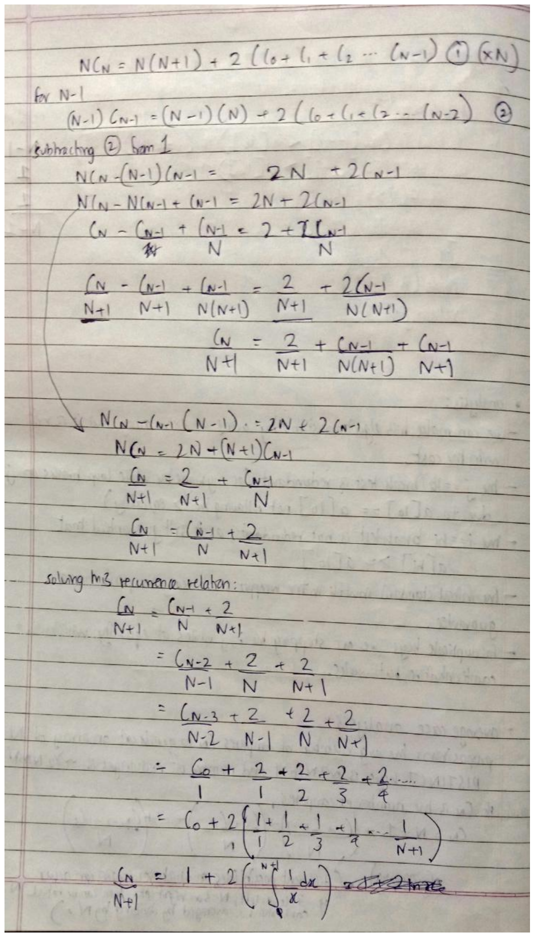
- average case analysis:

proposition: the avg number of compares C_N to quicksort an array of N DISTINCT keys is $\sim 2N\ln N$ and number of exchanges is $\sim \frac{1}{2} N\ln N$

proof: if C_N is the number of compares,

$$C_N = \underbrace{N+1}_{\text{(partitioning)}} + \left(\frac{C_0 + C_{N-1}}{N}\right) + \left(\frac{C_1 + C_{N-2}}{N}\right) + \cdots + \left(\frac{C_{N-1} + C_0}{N}\right)$$

(different possibilities in which recursion can occur eg 3 ele on left, N-3 on right etc, there can be total N such combinations ∴ averaged by dividing by N.)



$$N C_N = N(N+1) + 2(C_0 + C_1 + C_2 \cdots C_{N-1}) \quad \textcircled{1} \quad (\times N)$$

for N-1

$$(N-1)C_{N-1} = (N-1)(N) + 2(C_0 + C_1 + C_2 \cdots C_{N-2}) \quad \textcircled{2}$$

subtracting ② from 1

$$NC_N - (N-1)C_{N-1} = 2N + 2C_{N-1}$$

$$NC_N - NC_{N-1} + C_{N-1} = 2N + 2C_{N-1}$$

$$C_N - \frac{C_{N-1}}{\cancel{N}} + \frac{C_{N-1}}{N} = 2 + \frac{2C_{N-1}}{N}$$

$$\frac{C_N}{N+1} - \frac{C_{N-1}}{N+1} + \frac{C_{N-1}}{N(N+1)} = \frac{2}{N+1} + \frac{2C_{N-1}}{N(N+1)}$$

$$\frac{C_N}{N+1} = \frac{2}{N+1} + \frac{C_{N-1}}{N(N+1)} + \frac{C_{N-1}}{N+1}$$

$$NC_N - C_{N-1}(N-1) = 2N + 2C_{N-1}$$

$$NC_N = 2N + (N+1)C_{N-1}$$

$$\frac{C_N}{N+1} = \frac{2}{N+1} + \frac{C_{N-1}}{N}$$

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}$$

solving this recurrence relation:

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}$$

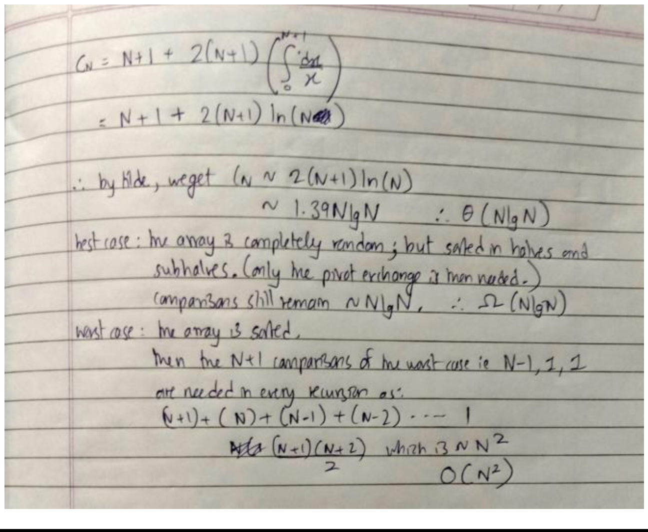
$$= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1}$$

$$= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1}$$

$$= \frac{C_0}{1} + \frac{2}{1} + \frac{2}{2} + \frac{2}{3} + \frac{2}{4} \cdots$$

$$= C_0 + 2\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} \cdots \frac{1}{N+1}\right)$$

$$\frac{C_N}{N+1} = 1 + 2\left(\int_0^{N+1} \frac{1}{x}\,dx\right) ~~= 1 + 2\ln x~~$$



$$C_N = N+1 + 2(N+1)\left(\int_0^{N+1} \frac{dx}{x}\right)$$

$$= N+1 + 2(N+1)\ln(N)$$

∴ by tilde, we get $C_N \sim 2(N+1)\ln(N)$

$\sim 1.39 N\lg N \quad ∴ \Theta(N\lg N)$

best case: the array is completely random; but sorted in halves and subhalves. (only the pivot exchange is then needed.) comparisons still remain $\sim N\lg N$, $∴ \Omega(N\lg N)$

worst case: the array is sorted.

Then the N+1 comparisons of the worst case ie N-1, 1, 1 are needed in every recursion as:

$$(N+1) + (N) + (N-1) + (N-2) \cdots 1$$

$$\frac{(N+1)(N+2)}{2} \text{ which is } \sim N^2$$

$$O(N^2)$$

PROGRAM:

```c
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<time.h>

void merge(int*arr,int* aux,int lo,int mid,int hi);
void merge_sort_inner(int* arr,int* aux,int lo, int hi);
void merge(int*arr,int* aux,int lo,int mid,int hi);
void swap(int *arr, int i, int j);
void quick_sort(int* arr,int len);
void quick_sort_inner(int* arr, int lo, int hi);
int partition(int* arr, int lo, int hi);

//merge sort
void merge_sort(int* arr, int len){
    int aux[len];
    merge_sort_inner(arr,aux,0,len-1);
}
void merge_sort_inner(int* arr,int* aux,int lo, int hi){

    if(hi<=lo)return;
    int mid=lo+(hi-lo)/2;
    merge_sort_inner(arr,aux,lo, mid);
    merge_sort_inner(arr,aux,mid+1,hi);
    merge(arr,aux,lo,mid,hi);

}
void merge(int*arr,int* aux,int lo,int mid,int hi){

    for(int i=lo;i<=hi;i++)aux[i]=arr[i];

    int i=lo,j=mid+1;
    for(int k=lo;k<=hi;k++){
        if     (i>mid)               arr[k]=aux[j++];
        else if(j>hi)                arr[k]=aux[i++];
        else if(aux[j]<aux[i])       arr[k]=aux[j++];
        else                         arr[k]=aux[i++];
    }
}

//quick sort
void swap(int *arr, int i, int j)
{
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
void quick_sort(int* arr,int len){
    quick_sort_inner(arr,0,len-1);
```

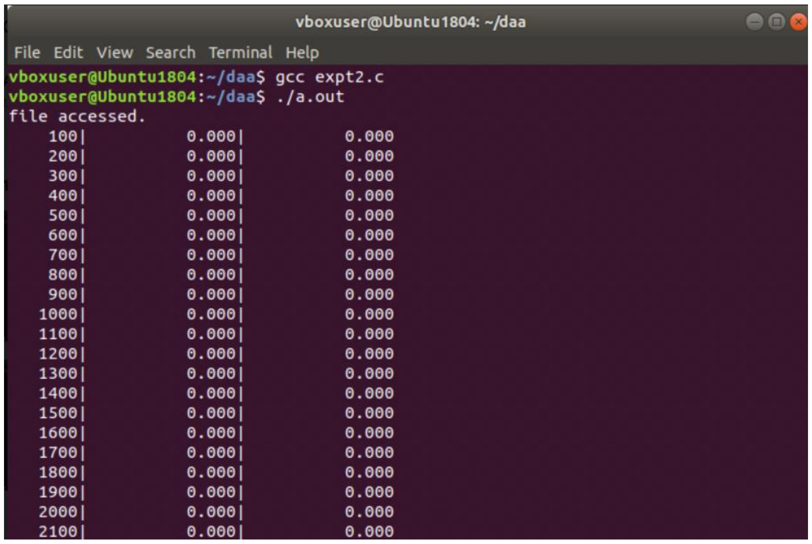
```
}
void quick_sort_inner(int* arr, int lo, int hi){
    if(lo>=hi)return;
    int j=partition(arr,lo,hi);
    quick_sort_inner(arr,lo,j-1);
    quick_sort_inner(arr,j+1,hi);
}
int partition(int* arr, int lo, int hi){
    int i=lo,j=hi+1;
    while(true){
        while(arr[++i]<arr[lo])if(i==hi)break;
        while(arr[--j]>arr[lo])if(j==lo)break;
        if(i>=j)break;
        swap(arr,i,j);
    }
    swap(arr,lo,j);
    return j;
}


int main(){

    int capacity=100000;
    //intitializing an array from the input file
    int buff_arr[capacity];
    FILE *fptr=fopen("inputfinal.txt","r");

    if(fptr!=NULL)printf("file accessed.\n");
    else {printf("error in accessing file.\n");return -1;}

    for(int i=0;i<capacity;i++)fscanf(fptr,"%d",&buff_arr[i]);
    fclose(fptr);

    //the subject array and clock variables
    int arr[capacity];
    clock_t start_t;
    clock_t end_t;

    for(int i=100;i<=capacity;i+=100){

        //filling for merge sort
        for(int j=0;j<i;j++)arr[j]=buff_arr[j];

        start_t=clock();
        merge_sort(arr,i);//merge sort
        end_t=clock();
        printf("%7d|%15.3lf|",i,((double)(end_t-start_t))/CLOCKS_PER_SEC);
//time
        //filling for quick sort
        for(int k=0;k<i;k++)arr[k]=buff_arr[k];
        start_t=clock();
        quick_sort(arr,i);//quick sort
        end_t=clock();
        printf("%15.3f\n",((double)(end_t-start_t))/CLOCKS_PER_SEC); //time
    }

}
```

Hence a text file is created with 1 lakh random integers and the experiment is carried upon this file.

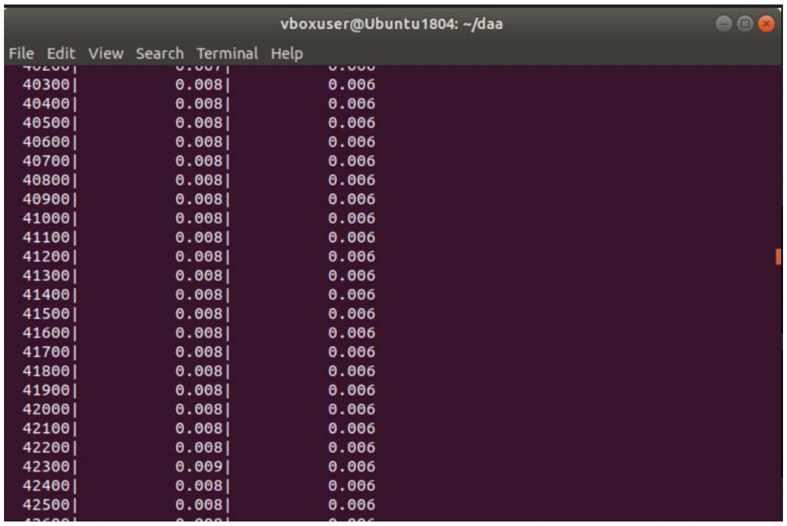
RESULT:

Outputs:

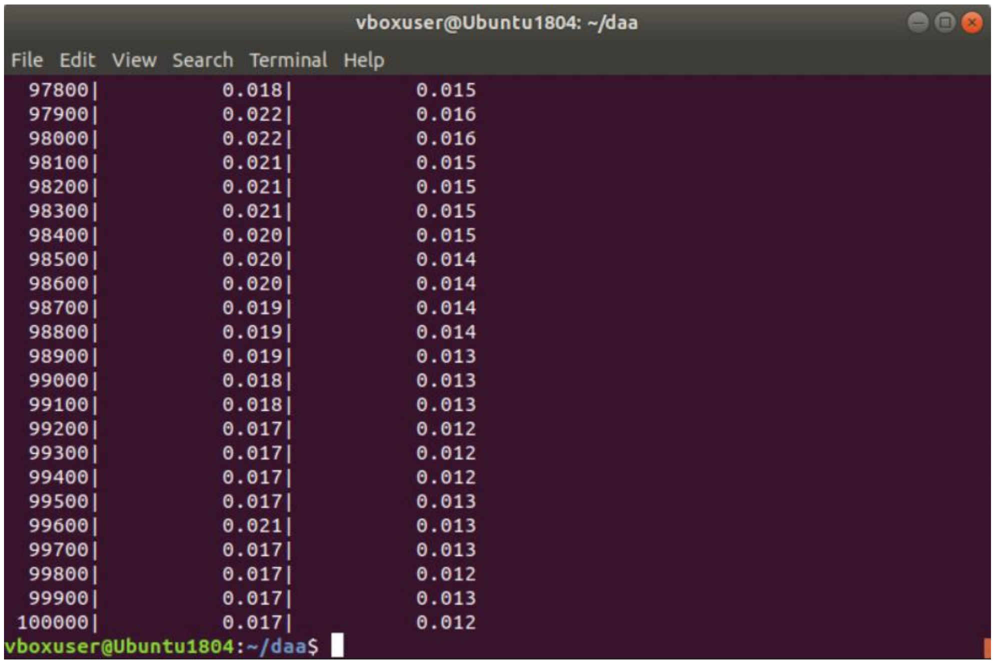
Column 1 is input size, column 2 is merge sort running time, and column 3 is quick sort running time. All running times are in terms of seconds. Input size was varied from 100 to 100000.



```
vboxuser@Ubuntu1804:~/daa$ gcc expt2.c
vboxuser@Ubuntu1804:~/daa$ ./a.out
file accessed.
    100|          0.000|          0.000
    200|          0.000|          0.000
    300|          0.000|          0.000
    400|          0.000|          0.000
    500|          0.000|          0.000
    600|          0.000|          0.000
    700|          0.000|          0.000
    800|          0.000|          0.000
    900|          0.000|          0.000
   1000|          0.000|          0.000
   1100|          0.000|          0.000
   1200|          0.000|          0.000
   1300|          0.000|          0.000
   1400|          0.000|          0.000
   1500|          0.000|          0.000
   1600|          0.000|          0.000
   1700|          0.000|          0.000
   1800|          0.000|          0.000
   1900|          0.000|          0.000
   2000|          0.000|          0.000
   2100|          0.000|          0.000
```



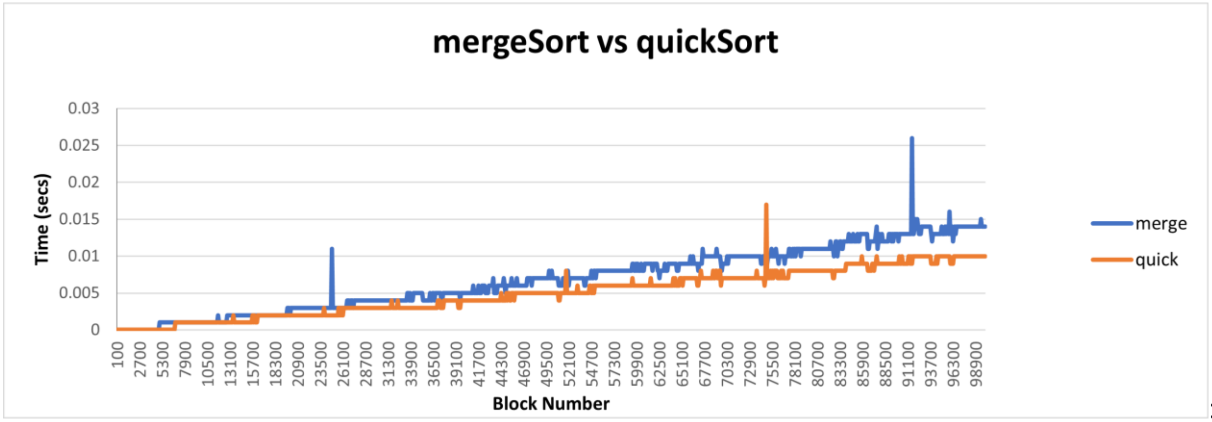
```
  40300|          0.008|          0.006
  40400|          0.008|          0.006
  40500|          0.008|          0.006
  40600|          0.008|          0.006
  40700|          0.008|          0.006
  40800|          0.008|          0.006
  40900|          0.008|          0.006
  41000|          0.008|          0.006
  41100|          0.008|          0.006
  41200|          0.008|          0.006
  41300|          0.008|          0.006
  41400|          0.008|          0.006
  41500|          0.008|          0.006
  41600|          0.008|          0.006
  41700|          0.008|          0.006
  41800|          0.008|          0.006
  41900|          0.008|          0.006
  42000|          0.008|          0.006
  42100|          0.008|          0.006
  42200|          0.008|          0.006
  42300|          0.009|          0.006
  42400|          0.008|          0.006
  42500|          0.008|          0.006
```



```
 97800|          0.018|          0.015
 97900|          0.022|          0.016
 98000|          0.022|          0.016
 98100|          0.021|          0.015
 98200|          0.021|          0.015
 98300|          0.021|          0.015
 98400|          0.020|          0.015
 98500|          0.020|          0.014
 98600|          0.020|          0.014
 98700|          0.019|          0.014
 98800|          0.019|          0.014
 98900|          0.019|          0.013
 99000|          0.018|          0.013
 99100|          0.018|          0.013
 99200|          0.017|          0.012
 99300|          0.017|          0.012
 99400|          0.017|          0.012
 99500|          0.017|          0.013
 99600|          0.021|          0.013
 99700|          0.017|          0.013
 99800|          0.017|          0.012
 99900|          0.017|          0.013
100000|          0.017|          0.012
vboxuser@Ubuntu1804:~/daa$
```

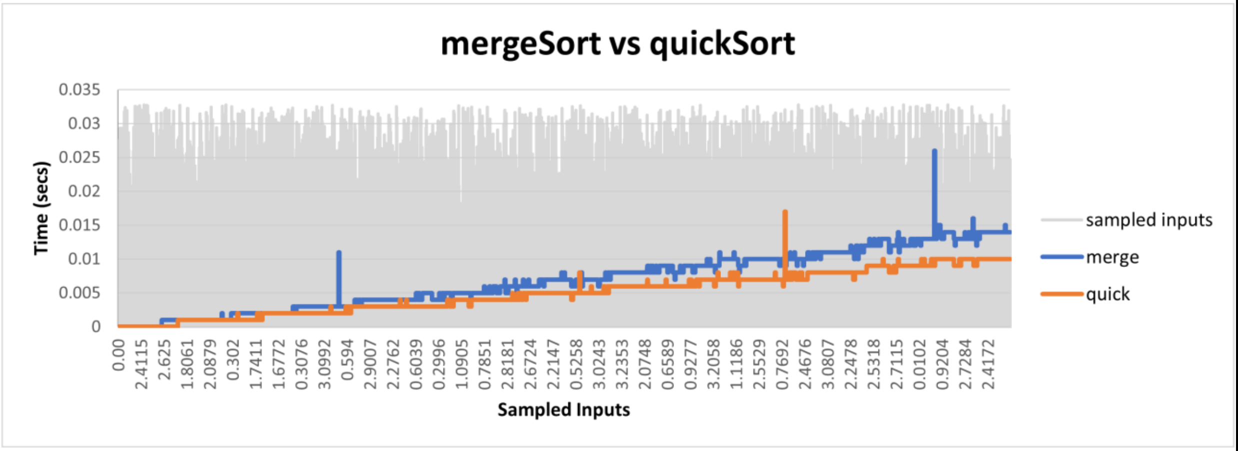
Plots:

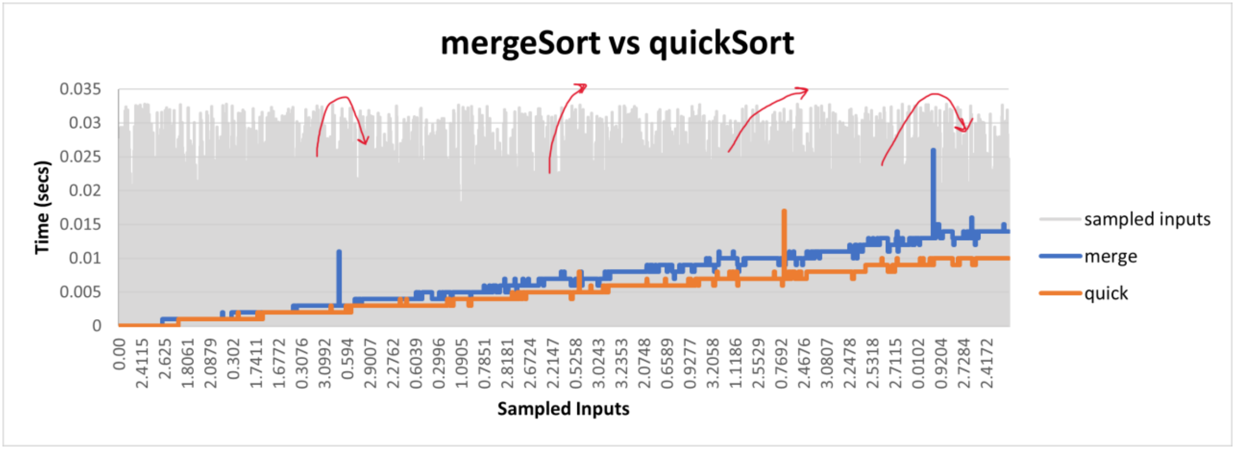
The above output was exported to Excel via a text file and the line graph was plotted.



:

Plots with the input values to examine spikes,drops etc:





The trends in the input were analysed and observed:

Conclusions:

1. Quick sort is better than Merge sort in terms of time taken.
2. Both the algorithms are undergoing an almost linear increase with input which is in line with their θ(NlgN) classification.
3. Merge Sort is always NlgN in avg, best and worst case. Quick Sort is NlgN in avg and best cases but becomes N^2 in the worst case. The worst case for quick sort is when the input is already sorted, which is quite rare.
4. Merge sort is undergoing a sharp increase in some places due to highly-varying input. This causes a bias on one side of the divided subarray, leading to more time.
5. Quick sort is not favorable when the input is already sorted. Hence it is going a steep increase when a partially sorted block is added.
6. Merge sort is not an in-place sort it makes a copy of the array given. Hence its space taken is N + c where N is the number of inputs and c is some constant space. Hence the space complexity is θ(N).
7. Quick sort makes a certain number of local variables say c in every recursive call. The total cost is proportional to the height of the recursive tree produced. Hence the space required is C*lgN which is θ(lgN), where C is some constant depending on c.