Experiment No.	**2 A**
Aim	**Experiment based on divide and conquers approach**
Name	**Rucha Sudhir Kulkarni**
UID No.	**2021300067**
Class & Division	**SE Computer Engineering (Div:A)(Batch:D)**
Date of Performance:	**08.02.2023**
Date of Submission:	**14.02.2023**

Aim:
Each student have to generate random 100000 numbers using rand() function and use this input as 1000 blocks of 100 integer numbers to Merge sort and Quick sort algorithms.

Algorithm:

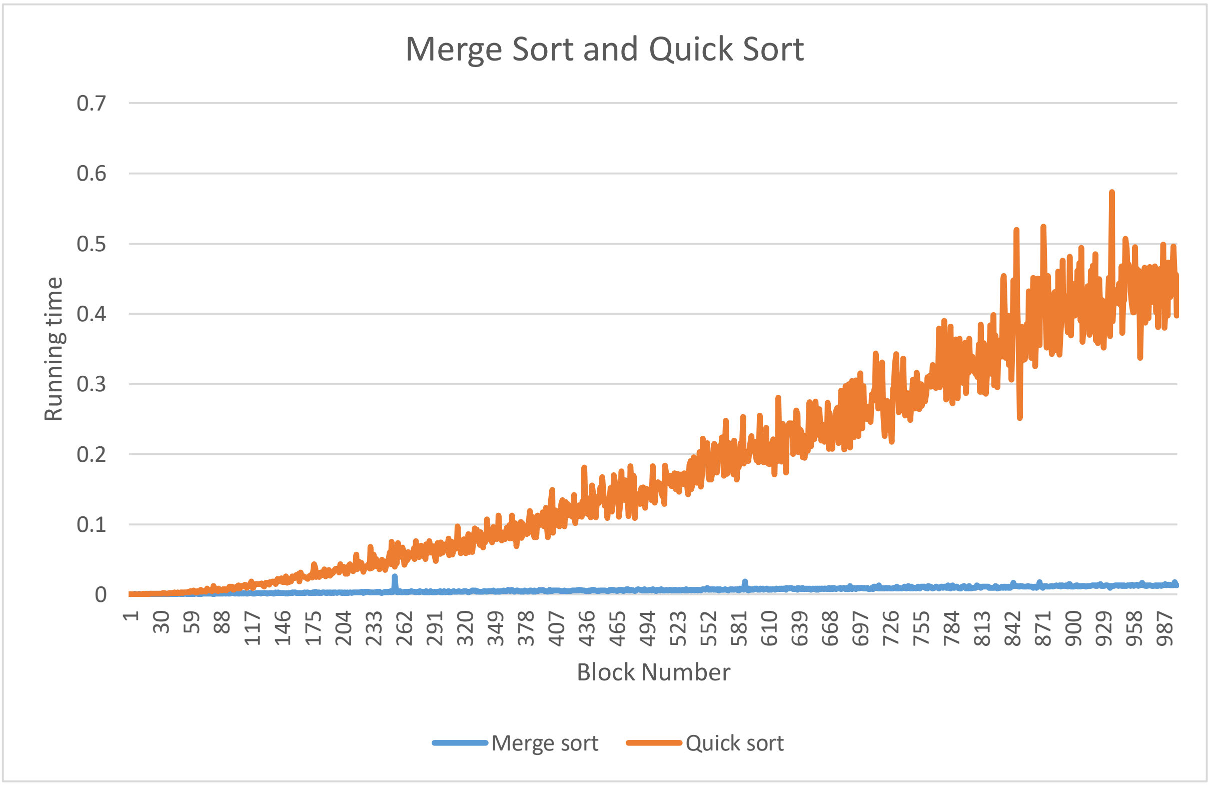
```
1. Start
2. Create 2 integer arrays of length 100000.
3. Input 100000 random integers into both the arrays using : rand()%10000
4. Store the generated random numbers in a text file.
5. Perform merge sort and quick sort of all the elements in groups of 100, then 200,
   then 300 ..so on till end.
6. Print the time taken for each sorting using clock() function.
7. Stop
8. Void mergesort(int arr[],int l, int r):
   8.1.  If r > l
   8.2.  Find the middle point to divide the array into two halves:
   8.3.  middle m = l + (r – l)/2
   8.4.  Call mergesort for first half:
   8.5.  Call mergesort(arr, l, m)
   8.6.  Call mergesort for second half:
   8.7.  Call mergesort(arr, m + 1, r)
   8.8.  Merge the two halves sorted in steps 2 and 3:
   8.9.  Call merge(arr, l, m, r)
9. Void Quicksort(int arr[],int n):
   9.1.  Make the left-most index value pivot
   9.2.  Partition the array using pivot value
   9.3.  Quicksort left partition recursively
   9.4.  Quicksort right partition recursively
10. Int partition(int a[],int lb,int ub)
   10.1. Choose the lowest index value as pivot
   10.2. Take two variables to point left and right of the array.
   10.3. left points to the low index
   10.4. right points to the high
   10.5. while value at left is less than pivot move right
```

10.6. while value at right is greater than pivot move left
10.7. if left<high , swap left and right values.
10.8. if left ≥ right swap pivot and high values.

Program:

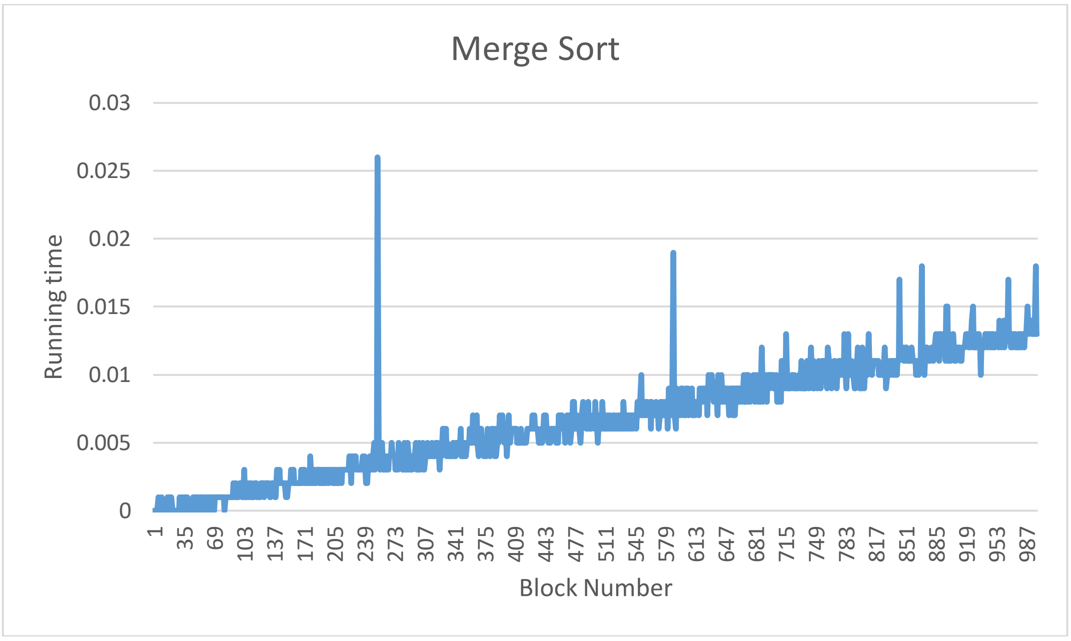
```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>


void merge(int arr[],int leftp,int midp,int rightp){
int leftlen=midp-leftp+1;
int rightlen=rightp-midp;

int left[leftlen],right[rightlen];

for(int i=0;i<leftlen;i++){
left[i]=arr[leftp+i];
}
for(int i=0;i<rightlen;i++){
right[i]=arr[midp+1+i];
}

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

while(i<leftlen && j<rightlen){
if(left[i]<=right[j]){
arr[k]=left[i];
i++;
}
else{
arr[k]=right[j];
j++;
}
k++;
}

while(i<leftlen){
arr[k]=left[i];
k++;
i++;
}

while(j<rightlen){
arr[k]=right[j];
k++;
j++;
}

}

void mergesort(int arr[],int l,int r){
if(l<r){
int m=l + (r-l)/2;

mergesort(arr,l,m);
mergesort(arr,m+1,r);
merge(arr,l,m,r);
}
}

int partition(int a[],int lb, int ub){
 int Pivot=a[lb];
 int st=lb,end=ub;
 while(st<end){
    while(a[st]<=Pivot){
        st++;
    }
    while(a[end]>Pivot){
        end--;
    }
    if(st<end)
    swap(&a[st],&a[end]);
 }
 swap(&a[end],&a[lb]);

 return end;
}

void Quicksort(int a[],int lb,int ub){
 if(lb<ub){
    int loc=partition(a,lb,ub);
    Quicksort(a,lb,loc-1);
    Quicksort(a,loc+1,ub);
 }
}
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

int main(){
int arr[100000],arr2[100000];
FILE *fptr;
FILE *sortans;
FILE *timeans;
FILE *sortans2;
FILE *timeans2;
fptr = fopen("lab2.txt","w");
sortans = fopen("lab2sort.txt","w");
timeans = fopen("lab2time.txt","w");
sortans2 = fopen("lab2sort2.txt","w");
timeans2 = fopen("lab2time2.txt","w");

clock_t start, end;
for(int i=0;i<100000;i++){
int x=rand()%10000;
arr[i]=x;
arr2[i]=x;
}

for(int i=0;i<100000;i++){
//printf("%d\n",arr[i]);
fprintf(fptr,"%d\n",arr[i]);
}

for(int i=1;i<=1000;i++){
    start = clock();
mergesort(arr,0,i*100);
end = clock();
//printf("Array %d after mergesort:\n",i);
double time_taken = (double)(end - start) / (double)(CLOCKS_PER_SEC);
printf("Time taken for %d elements to sort using merge sort:%f
s  \n",i*100,time_taken);
fprintf(timeans,"%f\n",time_taken);
fprintf(sortans,"Array %d after mergesort:\n",i);
for(int j=0;j<i*100;j++){
//printf("%d\n",arr[j]);
fprintf(sortans,"%d\n",arr[j]);
}
}

for(int i=1;i<=1000;i++){
    start = clock();
Quicksort(arr2,0,i*100);
end = clock();
//printf("Array %d after mergesort:\n",i);
double time_taken = (double)(end - start) / (double)(CLOCKS_PER_SEC);
printf("Time taken for %d elements to sort using Quick sort:%f
s  \n",i*100,time_taken);
fprintf(timeans2,"%f\n",time_taken);
fprintf(sortans2,"Array %d after Quick sort:\n",i);
for(int j=0;j<i*100;j++){
//printf("%d\n",arr[j]);
fprintf(sortans2,"%d\n",arr2[j]);
}
}

// printf("Array after mergesort:");
// for(int i=0;i<1000;i++){
// printf("%d\n",arr[i]);
// }

return 0;
}
```

Graph & observation:

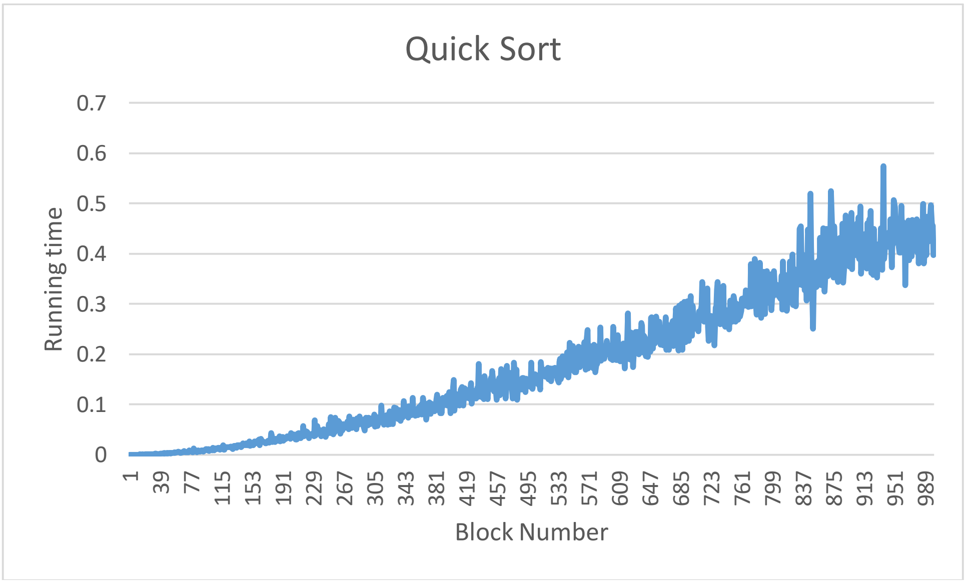


From the graph we can observe that quick sort takes more time compared to merge sort. This implies that merge sort is faster.

To get more clarity on their running times, individual running time graphs of both the algorithm have been constructed.



This graph shows that initially running time of merge sort is little higher when number of elements are less.



This graph shows that initially running time of quick sort is less when number of elements are less.

Hence, I observed that for small number of elements quick sort is faster while for large number of elements merge sort is faster.

Conclusion:

After performing the above experiment, I got to know how to perform merge sort and quick sort algorithms by using the divide and conquer strategy. I was also able to find their running times.