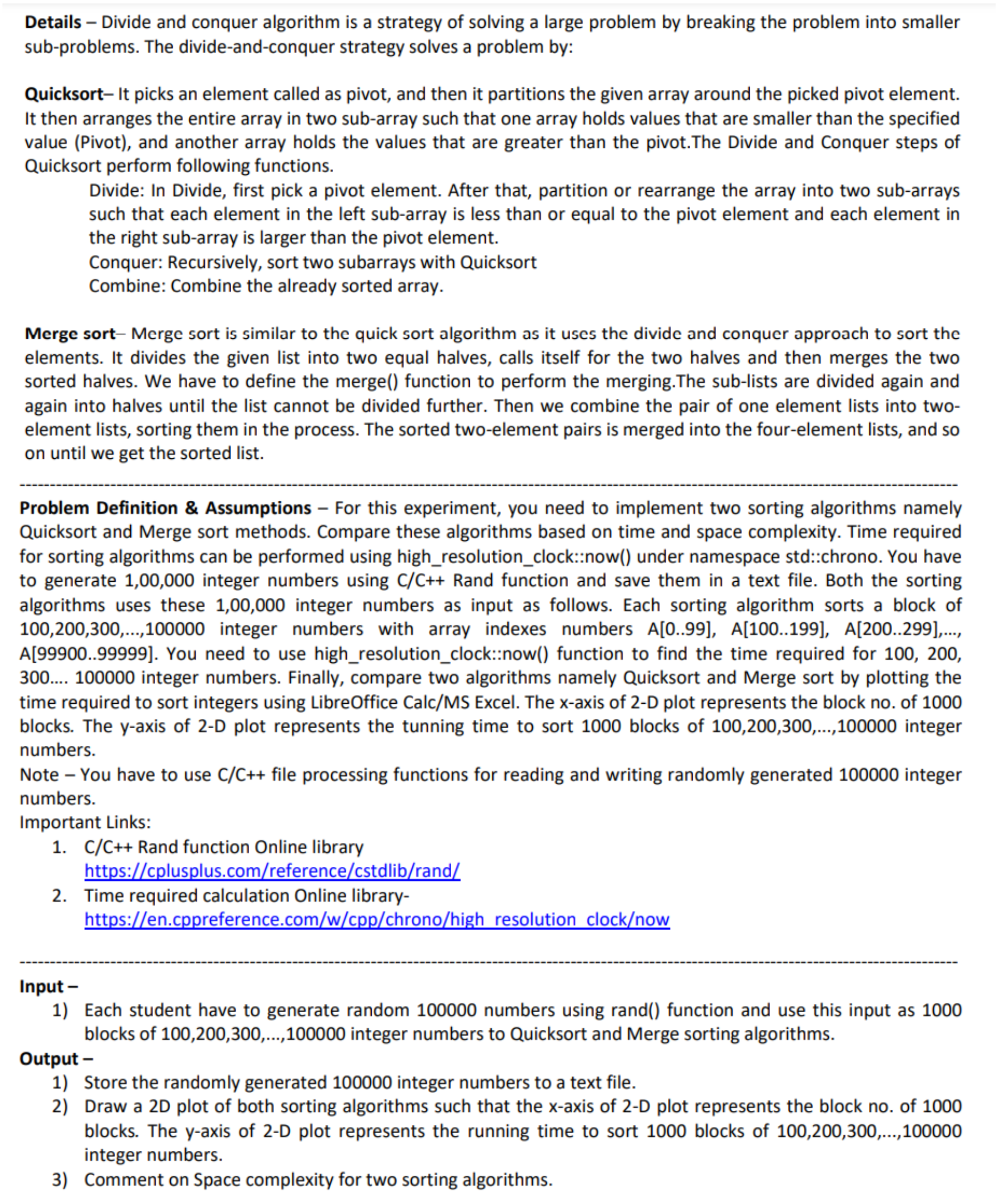
Name	Kinnari Shah
UID no.	2021700058
Experiment No.	2

AIM:	– Experiment based on 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.
Important Links:

1. C/C++ Rand function Online library
   https://cplusplus.com/reference/cstdlib/rand/
2. Time required calculation Online library-
   https://en.cppreference.com/w/cpp/chrono/high_resolution_clock/now

---

Input –

1) Each student have to generate random 100000 numbers using rand() function and use this input as 1000 blocks of 100,200,300,...,100000 integer numbers to Quicksort and Merge sorting algorithms.

Output –

1) Store the randomly generated 100000 integer numbers to a text file.
2) Draw a 2D plot of both sorting algorithms such that the x-axis of 2-D plot represents the block no. of 1000 blocks. The y-axis of 2-D plot represents the running time to sort 1000 blocks of 100,200,300,...,100000 integer numbers.
3) Comment on Space complexity for two sorting algorithms.

ALGORITHM:	To create a text file containing 100000 random numbers Step 1: Start. Include stdlib.h library to use the rand function. Step 2:Initialise an array of size 100000 number[100000] Step 3:Run a for loop from i=0 to 100000 Step 4: Call the rand() function and store the value in number[i] array . Step 5:Create a input.txt file and save the 100000 randomly generated values. Stop. **Quick Sort** 1. An array is divided into subarrays by selecting a **pivot element** (element selected from the array). While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot. 2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element. 3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array. **Merge Sort** 1. **I**f it is only one element in the list it is already sorted, return. 2. Divide the list recursively into two halves until it can no more be divided. 3. Merge the smaller lists into new list in sorted order.

PROGRAM:

Quick Sort:

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

void quicksort(int number[],int first,int last){
   int i, j, pivot, temp;
   if(first<last){
      pivot=first;
      i=first;
      j=last;
      while(i<j){
         while(number[i]<=number[pivot] && i<last)
         i++;
         while(number[j]>number[pivot])
         j--;
         if(i<j){
            temp=number[i];
            number[i]=number[j];
            number[j]=temp;
         }
      }
      temp=number[pivot];
      number[pivot]=number[j];
      number[j]=temp;
      quicksort(number,first,j-1);
      quicksort(number,j+1,last);
   }
}

int main()
{
   //to generate 1000000 random numbers
    /*int numbers[100000];
    for(int i=0;i<100000;i++)
    {

         numbers[i]=rand()%100000;
         printf("%d \n",numbers[i]);
    }*/

   FILE* ptr;
```

```
 // file in reading mode
 ptr = fopen("input.txt", "r");

 if (NULL == ptr)
 {
     printf("file can't be opened \n");
 }

 int block=1;
 int size=100;
 while(block<=1000)
 {
   int data[size];
   for(int i=0;i<size;i++)
  {
     fscanf(ptr,"%d ",&data[i]);
     //printf("%d ",data[i]);
  }

  clock_t t;
  t = clock();
  quicksort(data,0,size-1);
  if(block<3)  //this prints sorted first 2 blocks only
  {
    printf("\n\nafter sorting block %d:\n",block);
     for(int i=0;i<size;i++)
     {
       printf("%d \n",data[i]);
     }
  }


 t = clock() - t;
 double time_taken = ((double)t)/CLOCKS_PER_SEC;
 printf("\nRuntime for block number %d : %f\n",block,time_taken);
//printf("%f\n",time_taken);
 size=size+100;
 block++;
 fseek(ptr,0,SEEK_SET); //moving cursor again to start pointer of txt file

 }
```

```
    fclose(ptr);

}
```

Merge Sort

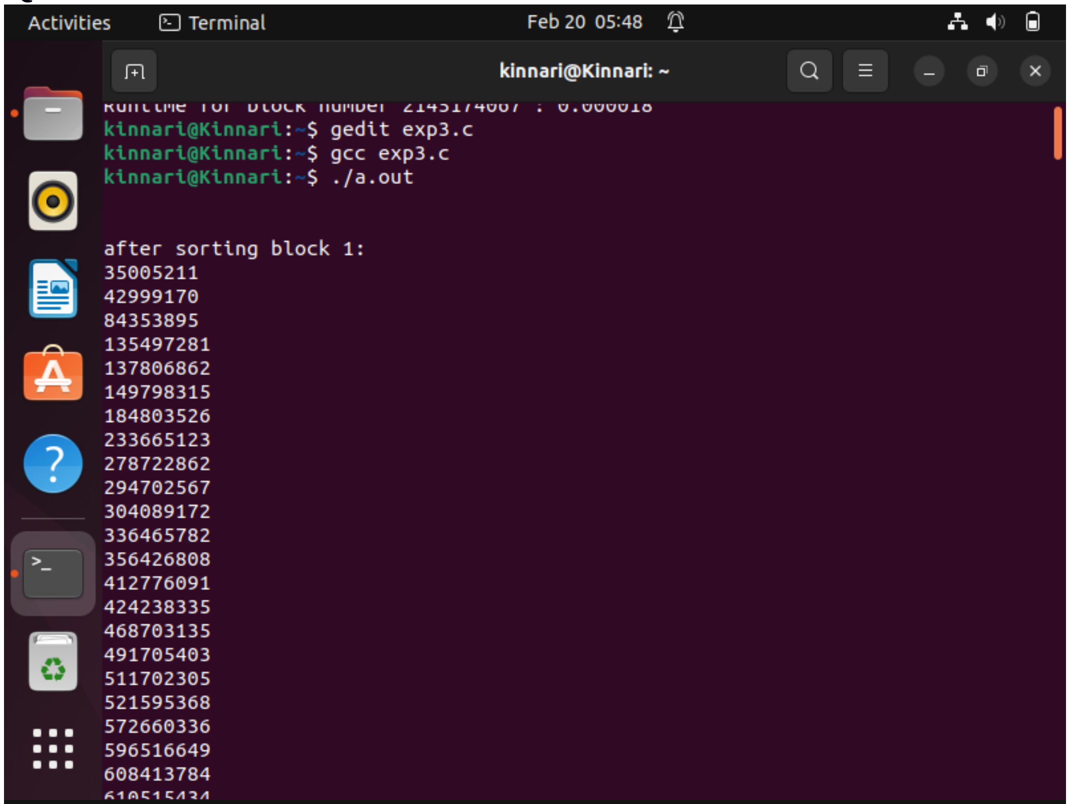
```
#include <stdio.h>
#include<stdlib.h>
#include<time.h>

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

void merge(int a[], int beg, int mid, int end)
{
    int i, j, k;
    int n1 = mid - beg + 1;
    int n2 = end - mid;

    int LeftArray[n1], RightArray[n2]; //temporary arrays

    /* copy data to temp arrays */
    for (int i = 0; i < n1; i++)
    LeftArray[i] = a[beg + i];
    for (int j = 0; j < n2; j++)
    RightArray[j] = a[mid + 1 + j];

    i = 0; /* initial index of first sub-array */
    j = 0; /* initial index of second sub-array */
    k = beg;  /* initial index of merged sub-array */

    while (i < n1 && j < n2)
    {
        if(LeftArray[i] <= RightArray[j])
        {
            a[k] = LeftArray[i];
            i++;
        }
        else
        {
            a[k] = RightArray[j];
            j++;
        }
        k++;
    }
    while (i<n1)
    {
        a[k] = LeftArray[i];
        i++;
        k++;
    }

    while (j<n2)
    {
        a[k] = RightArray[j];
        j++;
        k++;
    }
}

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);
    }
}

int main()
{
   //to generate 1000000 random numbers
    /*int numbers[100000];
    for(int i=0;i<100000;i++)
    {

         numbers[i]=rand()%100000;
         printf("%d \n",numbers[i]);
    }*/

   FILE* ptr;
```

	(code below)

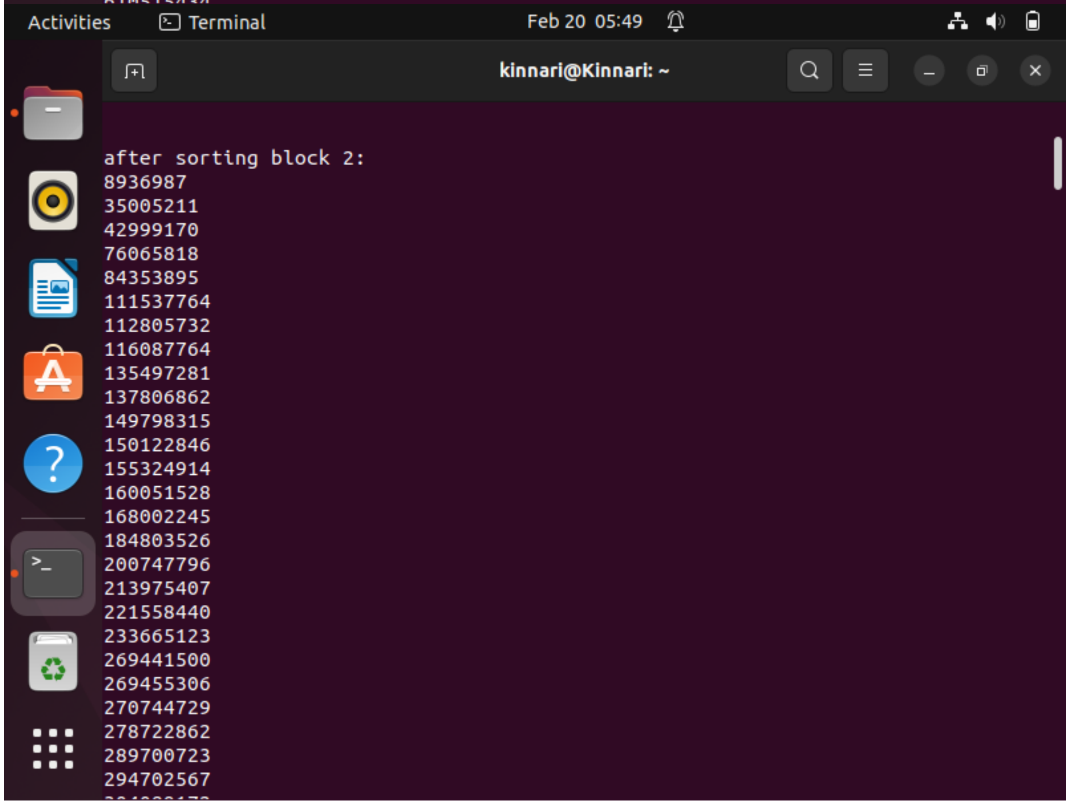
```
    // file in reading mode
    ptr = fopen("input.txt", "r");

    if (NULL == ptr)
    {
        printf("file can't be opened \n");
    }

    int block=1;
    int size=100;
    while(block<=1000)
    {
      int data[size];
      for(int i=0;i<size;i++)
     {
        fscanf(ptr,"%d ",&data[i]);
        //printf("%d ",data[i]);
     }

     clock_t t;
     t = clock();
     mergeSort(data,0,size-1);
     if(block<3)  //this prints sorted first 2 blocks only
     {
       printf("\n\nafter sorting block %d:\n",block);
        for(int i=0;i<size;i++)
        {
          printf("%d \n",data[i]);
        }
     }


    t = clock() - t;
    double time_taken = ((double)t)/CLOCKS_PER_SEC;
    printf("\nRuntime for block number %d : %f\n",block,time_taken);
   //printf("%f\n",time_taken);
    size=size+100;
    block++;
    fseek(ptr,0,SEEK_SET); //moving cursor again to start pointer of txt
file

    }

     fclose(ptr);

}
```

RESULT:

Quick Sort

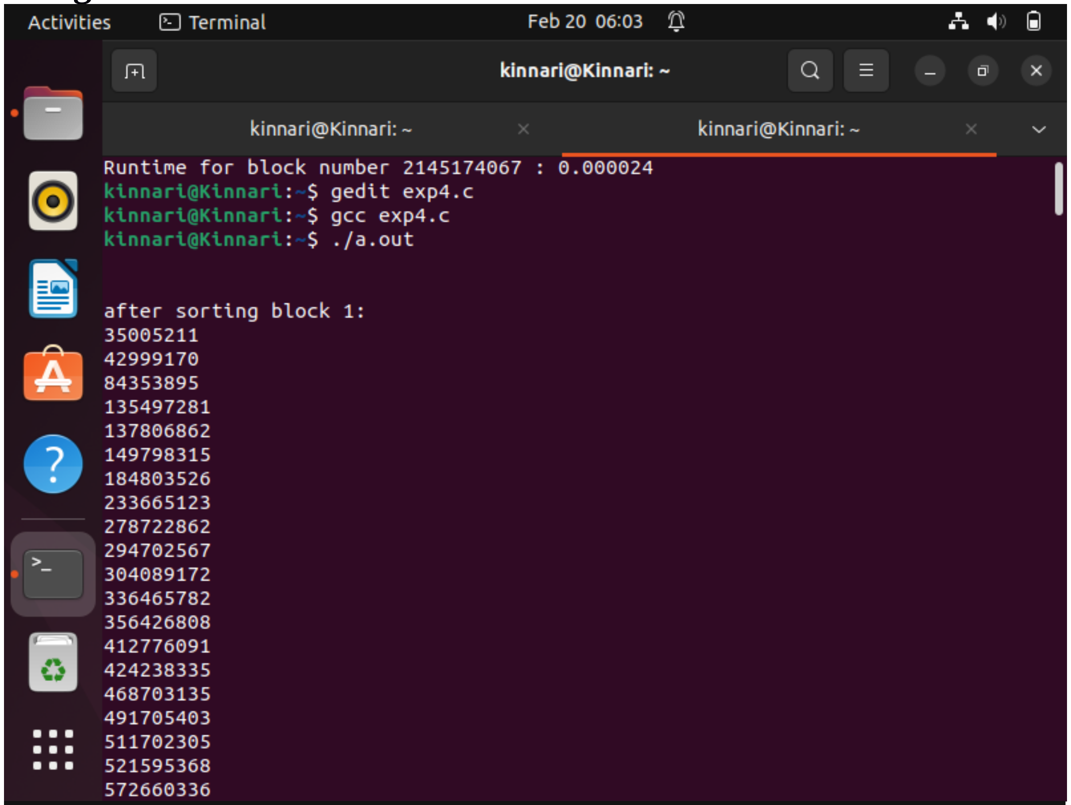


```
Runtime for block number 2143174067 : 0.000018
kinnari@Kinnari:~$ gedit exp3.c
kinnari@Kinnari:~$ gcc exp3.c
kinnari@Kinnari:~$ ./a.out


after sorting block 1:
35005211
42999170
84353895
135497281
137806862
149798315
184803526
233665123
278722862
294702567
304089172
336465782
356426808
412776091
424238335
468703135
491705403
511702305
521595368
572660336
596516649
608413784
610515434
```

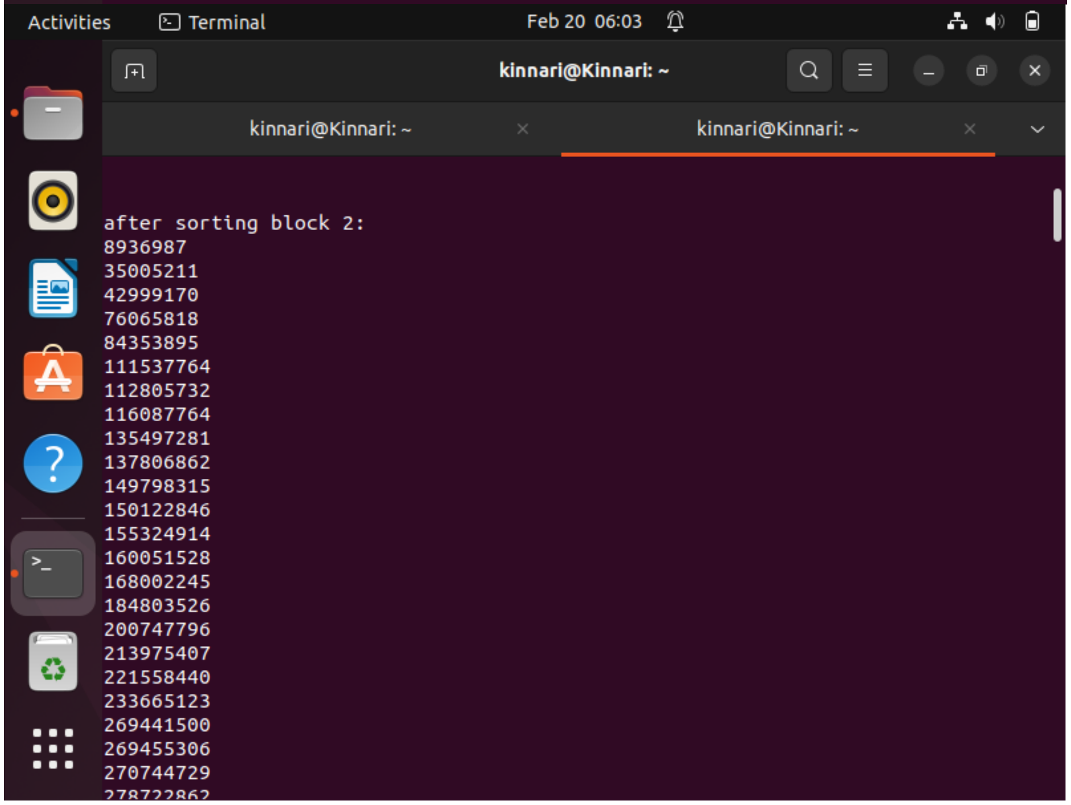


```


after sorting block 2:
8936987
35005211
42999170
76065818
84353895
111537764
112805732
116087764
135497281
137806862
149798315
150122846
155324914
160051528
168002245
184803526
200747796
213975407
221558440
233665123
269441500
269455306
270744729
278722862
289700723
294702567
```

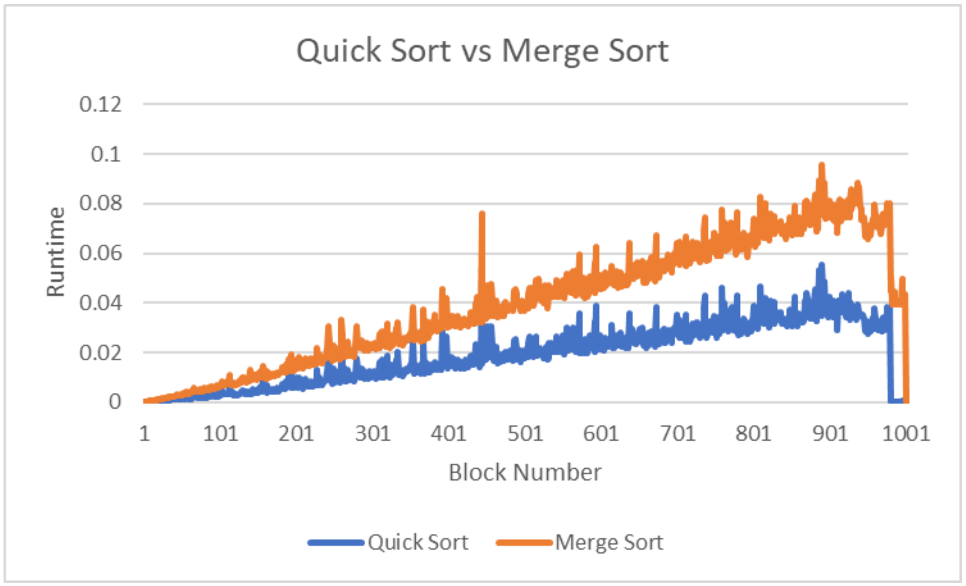
Merge Sort



```
Runtime for block number 2145174067 : 0.000024
kinnari@Kinnari:~$ gedit exp4.c
kinnari@Kinnari:~$ gcc exp4.c
kinnari@Kinnari:~$ ./a.out


after sorting block 1:
35005211
42999170
84353895
135497281
137806862
149798315
184803526
233665123
278722862
294702567
304089172
336465782
356426808
412776091
424238335
468703135
491705403
511702305
521595368
572660336
```



```


after sorting block 2:
8936987
35005211
42999170
76065818
84353895
111537764
112805732
116087764
135497281
137806862
149798315
150122846
155324914
160051528
168002245
184803526
200747796
213975407
221558440
233665123
269441500
269455306
270744729
```



OBSERVATION:	From above graph ,it is clearly visible that the running time for Merge sort algorithm representing orange line is more than the running time for Quick sort algorithm. There are certain reasons due to which quicksort is better especially in case of arrays: ➢ Quick sort is an in-place sorting algorithm. In-place sorting means no additional storage space is needed to perform sorting. Merge sort requires a temporary array to merge the sorted arrays and hence it is not in-place giving Quick sort the advantage of space. ➢ The worst case of quicksort **$O(n^2)$** can be avoided by using randomized quicksort. It can be easily avoided with high probability by choosing the right pivot. Obtaining an average case behavior by choosing right pivot element makes it improvise the performance and becoming as efficient as Merge sort. ➢ Quicksort in particular exhibits good cache locality and this makes it faster than merge sort in many cases like in virtual memory environment.
CONCLUSION:	In this experiment, we performed merge sort and quick sort on randomly generated 100000 numbers and observed that quick sort is faster than merge sort.