

<b>Name</b>	RITIK SINGH
<b>UID no.</b>	20201700061
<b>Experiment No.</b>	2

<b>AIM:</b>	Experiment based on divide and conquer approach.
-------------	--

### Program 1

<b>PROBLEM STATEMENT :</b>	<p>– 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 <code>high_resolution_clock::now()</code> under namespace <code>std::chrono</code>. You have to generate 1,00,000 integer numbers using C/C++ <code>Rand</code> 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 <code>A[0..99]</code>, <code>A[100..199]</code>, <code>A[200..299]</code>,..., <code>A[99900..99999]</code>. You need to use <code>high_resolution_clock::now()</code> 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 tuning time to sort 1000 blocks of 100,200,300,...,100000 integer numbers.</p>
<b>ALGORITHM/ THEORY:</b>	<p><b>What is Time complexity?</b></p> <p>Time complexity is defined as the amount of time taken by an algorithm to run, as a function of the length of the input. It measures the time taken to execute each statement of code in an algorithm. It is not going to examine the total execution time of an algorithm. Rather, it is going to give information about the variation (increase or decrease) in execution time when the number of operations (increase or decrease) in an algorithm. Yes, as the definition says, the amount of time taken is a function of the length of input only.</p> <p><b>What is Quick Sort?</b></p> <p>Quicksort is based on a divide and conquer strategy. It works in the following steps:</p>

1. It selects an element from within the array known as the **pivot element**.
2. Then it makes use of the **partition algorithm** to divide the array into two sub-arrays. One sub-array has all the values less than the pivot element. The other sub-array has all the values higher than the pivot element.
3. In the next step, the quicksort algorithm calls itself recursively to sort these two sub-arrays.
4. Once the sorting is done, we can combine both the sub-arrays into a single sorted array.

The most important part of quicksort is the partition algorithm. The partition algorithm puts the element into either of the two subarrays depending on the pivot point. We can choose pivot point in many ways:

- Take the first element as the pivot point
- Take the last element of the array as the pivot point
- Take the middle element of the array as the pivot element.
- Take random element as the pivot element in every recursive call

#### **PARTITION ALGORITHM:-**

Step 1: Choose the highest index value i.e. the last element of the array as a pivot point

Step 2: Point to the 1st and last index of the array using two variables.

Step 3: Left points to the low index and Right points to the high

Step 4: while Array[Left] < pivot

Move Right

Step 5: while Array[Right] > pivot

Move Left

Step 6: If no match found in step 5 and step 6, swap Left and Right

Step 7: If Left  $\geq$  Right, their meeting point is the new pivot

#### **QUICKSORT ALGORITHM:-**

Step 1 – Array[Right] = pivot

Step 2 – Apply partition algorithm over data items using pivot element

Step 3 – quicksort(left of pivot)

Step 4 – quicksort(right of pivot)

#### **What is Merge Sort?**

The merge sort algorithm is an implementation of the divide and conquer technique. Thus, it gets completed in three steps:

**1. Divide:** In this step, the array/list divides itself recursively into sub-arrays

	<p>until the base case is reached.</p> <p><b>2. Recursively solve:</b> Here, the sub-arrays are sorted using recursion.</p> <p><b>3. Combine:</b> This step makes use of the <b>merge( ) function</b> to combine the sub-arrays into the final sorted array.</p> <p><b>MERGESORT ALGORITHM:-</b></p> <p>Step 1: Find the middle index of the array.  <math>\text{Middle} = 1 + (\text{last} - \text{first})/2</math></p> <p>Step 2: Divide the array from the middle.</p> <p>Step 3: Call merge sort for the first half of the array  MergeSort(array, first, middle)</p> <p>Step 4: Call merge sort for the second half of the array.  MergeSort(array, middle+1, last)</p> <p>Step 5: Merge the two sorted halves into a single sorted array.</p>
<p><b>PROGRAM:</b></p>	<pre> #include &lt;stdio.h&gt; #include &lt;math.h&gt; #include &lt;conio.h&gt; #include &lt;stdlib.h&gt; #include &lt;time.h&gt;  void getInput() {     FILE *fp;     fp = fopen("inputexp2.txt","w");     for(int i=0;i&lt;100000;i++)         fprintf(fp,"%d ",rand()%100000);     fclose(fp); }  void merge(int arr[], int p, int q, int r) {      // Create L ← A[p..q] and M ← A[q+1..r]     int n1 = q - p + 1;     int n2 = r - q;      int L[n1], M[n2];      for (int i = 0; i &lt; n1; i++)         L[i] = arr[p + i];     for (int j = 0; j &lt; n2; j++)         M[j] = arr[q + 1 + j]; </pre>

```

// Maintain current index of sub-arrays and main array
int i, j, k;
i = 0;
j = 0;
k = p;

// Until we reach either end of either L or M, pick larger
among
// elements L and M and place them in the correct position at
A[p..r]
while (i < n1 && j < n2) {
    if (L[i] <= M[j]) {
        arr[k] = L[i];
        i++;
    } else {
        arr[k] = M[j];
        j++;
    }
    k++;
}

// When we run out of elements in either L or M,
// pick up the remaining elements and put in A[p..r]
while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
}

while (j < n2) {
    arr[k] = M[j];
    j++;
    k++;
}
}

// Divide the array into two subarrays, sort them and merge
them
void mergeSort(int arr[], int l, int r) {
    if (l < r) {

        // m is the point where the array is divided into two
        subarrays

```

```

    int m = l + (r - l) / 2;

    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);

    // Merge the sorted subarrays
    merge(arr, l, m, r);
}
}

int partition(int A[], int low, int high)
{
    int pivot = A[low];
    int i = low + 1;
    int j = high;
    int temp;

    do
    {
        while (A[i] <= pivot)
        {
            i++;
        }

        while (A[j] > pivot)
        {
            j--;
        }

        if (i < j)
        {
            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    } while (i < j);

    // Swap A[low] and A[j]
    temp = A[low];
    A[low] = A[j];
    A[j] = temp;
    return j;
}

```

```

void quickSort(int A[], int low, int high)
{
    int partitionIndex; // Index of pivot after partition

    if (low < high)
    {
        partitionIndex = partition(A, low, high);
        quickSort(A, low, partitionIndex - 1); // sort left
subarray
        quickSort(A, partitionIndex + 1, high); // sort right
subarray
    }
}

int main(){

    getInput();
    FILE *rt, *tks;
    int a=99;
    int arrNums[100000];
    clock_t t;
    rt = fopen("inputexp2.text", "r");
    tks = fopen("mTimes.txt", "w");
    for(int i=0; i<1000; i++){
        for(int j=0; j<=a; j++){
            fscanf(rt, "%d", &arrNums[j]);
        }
        t = clock();
        mergeSort(arrNums, 0, a+1);
        t = clock() - t;
        double time_taken = ((double)t)/CLOCKS_PER_SEC;
        fprintf(tks, "time taken for %d iteration is %Lf\n",
(i+1), time_taken);
        printf("%d\t%lf\n", (i+1), time_taken);
        a = a + 100;
        fseek(rt, 0, SEEK_SET);
    }
    fclose(tks);
    tks = fopen("qTimes.txt", "w");
    a=99;
    for(int i=0; i<1000; i++){
        for(int j=0; j<=a; j++){

```

```

        fscanf(rt, "%d", &arrNums[j]);
    }
    t = clock();
    quickSort(arrNums, 0, a+1);
    t = clock() - t;
    double time_taken = ((double)t)/CLOCKS_PER_SEC;
    fprintf(tks, "time taken for %d iteration is %Lf\n",
(i+1), time_taken);
    printf("%d\t%f\n", (i+1), time_taken);
    a = a + 100;
    fseek(rt, 0, SEEK_SET);
}
fclose(tks);
fclose(rt);
return 0;
}

```

## RESULT:

The screenshot shows the Visual Studio Code interface with the following components:

- EXPLORER:** Displays 'NO FOLDER OPENED' with a message 'You have not yet opened a folder.' and an 'Open Folder' button.
- EDITOR:** Shows the C code from the previous block. The file name is 'exp2.c'.
- TERMINAL:** Displays the output of the program, which is a list of iteration numbers and time taken values. The output is as follows:

Iteration	Time Taken
129	0.002000
130	0.001000
131	0.001000
132	0.002000
133	0.001000
134	0.001000
135	0.002000
136	0.001000
137	0.002000
138	0.001000
139	0.002000
140	0.002000
141	0.001000
142	0.001000
143	0.002000
144	0.002000
145	0.001000
146	0.001000
147	0.003000
148	0.001000
149	0.002000
150	0.001000
151	0.002000
152	0.002000
153	0.001000
154	0.001000
155	0.002000
156	0.002000
157	0.002000
158	0.001000
159	0.002000
160	0.002000
161	0.001000
162	0.002000
163	0.002000
164	0.003000
165	0.001000
166	0.001000
167	0.002000

The screenshot shows the Visual Studio Code interface. The Explorer pane on the left indicates 'NO FOLDER OPENED'. The main editor displays a C file named 'exp2.c' with the following code:

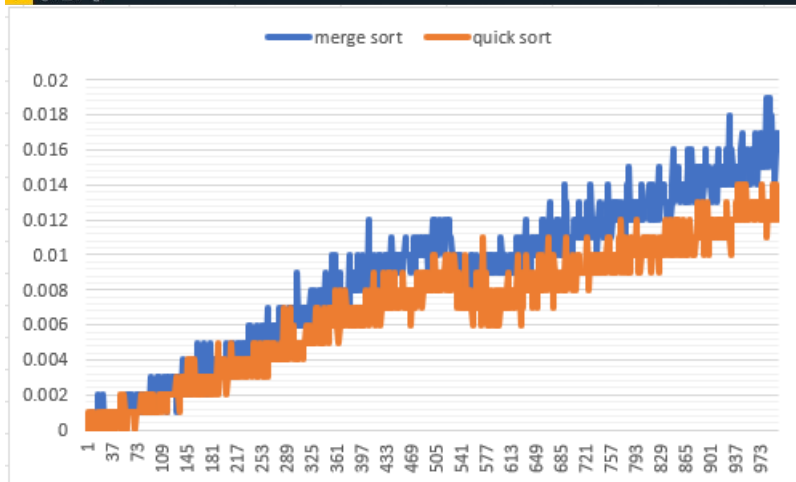
```

164 ...fclose(rt);
165 ...return 0;
166 }

```

The TERMINAL pane at the bottom shows the output of the program, which consists of a list of numbers and their corresponding time values in seconds:

Index	Value
996	0.017000
997	0.017000
998	0.015000
999	0.018000
1000	0.017000
1	0.000000
2	0.000000
3	0.000000
4	0.001000
5	0.000000
6	0.000000
7	0.000000
8	0.000000
9	0.000000
10	0.000000
11	0.000000
12	0.000000
13	0.000000
14	0.001000
15	0.000000
16	0.000000
17	0.000000
18	0.000000
19	0.000000
20	0.000000
21	0.000000
22	0.000000
23	0.002000
24	0.000000
25	0.001000
26	0.000000
27	0.000000
28	0.000000
29	0.000000
30	0.000000
31	0.000000
32	0.000000
33	0.001000
34	0.000000



## CONCLUSION:

WE HAVE USED TWO ALGORITHM TECHNIQUES i.e MERGESORT AND QUICKSORT TO SORT THE RANDOM NO.s . BOTH THE ALGORITHMS HAVE LESS TIME COMPLEXITY. I HAVE SEEN BEHAVIOUR OF THE ALGORITHMS WITH TIME USING OF GRAPH . IT IS SEEN THAT QUICK HAS BETTER TIME COMPLEXITY THAN MERGE SORT.