Name	Stephen David Vaz
UID no.	2021700070
Experiment No.	2

AIM:	Experiment based on divide and conquer approach.
PROBLEM STATEMENT :	This experiment requires implementing Quicksort and Merge sort algorithms to compare their time and space complexity. 100,000 integer numbers are generated using C/C++ Rand function and saved in a text file. The sorting algorithms will sort a block of 100, 200, 300, ..., 100,000 integer numbers, and the time required for sorting each block is recorded using high_resolution_clock::now() function. The time required to sort the integers is then plotted against the block number using LibreOffice Calc/MS Excel, with the x-axis representing the block number and the y-axis representing the time taken to sort 1000 blocks of integer numbers.
ALGORITHM/ THEORY:	**Quicksort** is a popular sorting algorithm that works by dividing an array into two sub-arrays and recursively sorting each of them. It chooses a pivot element from the array and partitions the other elements into two sub-arrays based on whether they are less than or greater than the pivot, and then recursively sorts the sub-arrays. Steps: 1. Choose last element as pivot 2. Partition the array into two sub-arrays based on whether the elements are less than or greater than the pivot. 3. Recursively apply steps 1 and 2 to the left and right sub-arrays. 4. Repeat until the sub-arrays have length 0 or 1. **Merge sort** works by dividing an array into two halves, recursively sorting each half, and then merging the sorted halves into a single sorted array. It uses a divide-and-conquer approach and has a time complexity of O(n log n), which makes it an efficient algorithm for large datasets. It is also stable, meaning it preserves the relative order of equal elements in the input array. Steps: 1. Divide the unsorted array into two halves, left and right. 2. Recursively sort the left half by dividing it into two sub-arrays and merging them using the same approach.

3. Recursively sort the right half by dividing it into two sub-arrays and merging them using the same approach.
4. Merge the two sorted sub-arrays to form a single sorted array.

PROGRAM:

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

double populate(int a[], int b[], int n) {
   clock_t start, end;
   double cpu_time_used;
   start = clock();
   for(int i = 0; i < n; i++)
   {
       int r = rand();
       a[i] = b[i] = r;
   }
   end = clock();
   FILE *fp = fopen("./random.txt", "w+");
   if(!fp) {
       printf("Error opening file\n");
       return -1;
   }
   for(int i = 0; i < n; i++) {
       fprintf(fp, "%d\n", a[i]);
   }
   cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
   return cpu_time_used;
}

void merge(int a[], 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] = a[l + i];
   for (j = 0; j < n2; j++)
       R[j] = a[m + 1 + j];
   i = j = 0;
   k = l;
```

	`` while(i < n1 && j < n2) { if(L[i] <= R[j]) { a[k] = L[i]; i++; } else { a[k] = R[j]; j++; } k++; } while (i < n1) { a[k] = L[i]; i++; k++; } while (j < n2) { a[k] = R[j]; j++; k++; } } void mergeSort(int a[], int l, int r) { if(l<r) { int m = (l+r)/2; mergeSort(a, l, m); mergeSort(a, m+1, r); merge(a, l, m, r); } } double mergeCalc(int a[], int n) { FILE *fp = fopen("./mergeSort.csv", "w+"); double totalTime = 0; if(!fp) { printf("Error opening file\n"); return -1; } fprintf(fp, "n, time\n"); for (int i = 99; i <= n; i+=100) ```

	(code below)

```
    {
        clock_t start, end;
        double cpu_time_used;
        start = clock();
        mergeSort(a, 0, i);
        end = clock();
        cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
        totalTime += cpu_time_used;
        fprintf(fp, "%d, %f\n", i+1, cpu_time_used);
        printf("Sorted from 0 to %d in %.2fs\n", i, cpu_time_used);
    }
    fclose(fp);
    fp = fopen("./mergeSort.txt", "w+");
    for(int i = 0; i < n; i++) {
        fprintf(fp, "%d\n", a[i]);
    }
    fclose(fp);
    return totalTime;
}

void swap(int *x, int *y) {
    int t = *x;
    *x = *y;
    *y = t;
}

int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int a[], int low, int high)
```

	(code below)

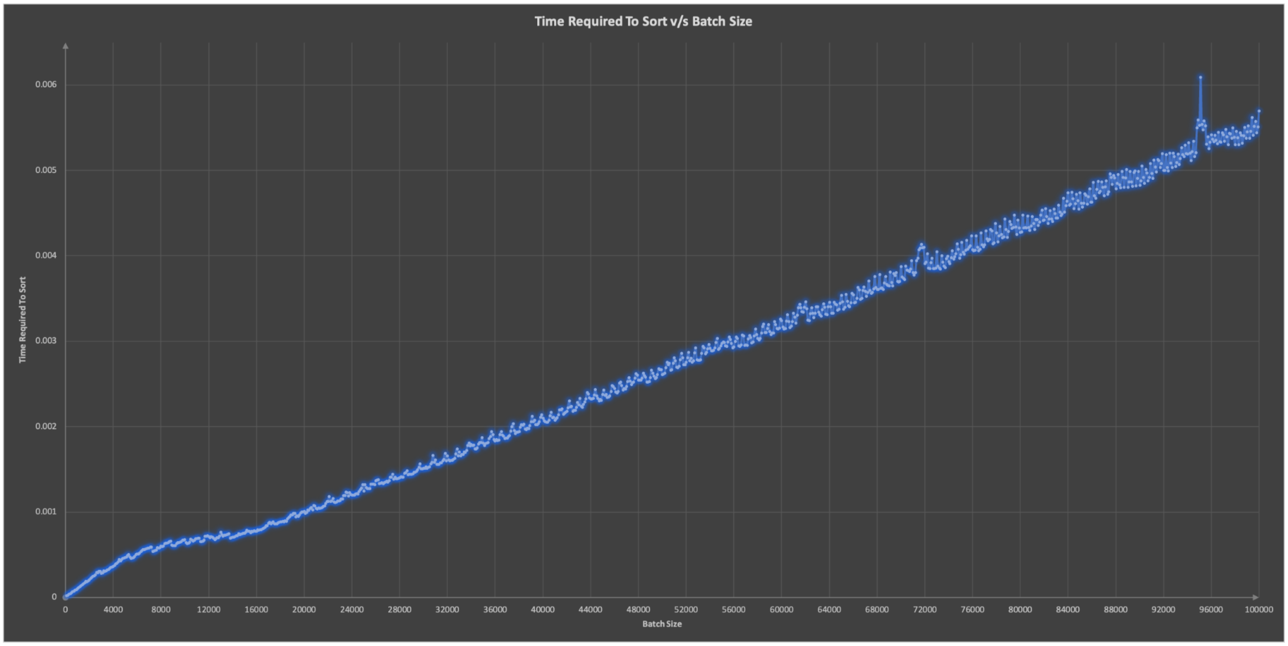
```
{
   if (low < high) {
       int pi = partition(a, low, high);
       quickSort(a, low, pi - 1);
       quickSort(a, pi + 1, high);
   }
}

double qC(int a[], int n) {
   FILE *fp = fopen("./quickSort.csv", "w+");
   double totalTime = 0;
   if(!fp) {
       printf("Error opening file\n");
       return -1;
   }
   fprintf(fp, "n, time\n");
   for (int i = 99; i <= n; i+=100)
   {
       clock_t start, end;
       double cpu_time_used;
       start = clock();
       quickSort(a, 0, i);
       end = clock();
       cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
       totalTime += cpu_time_used;
       fprintf(fp, "%d, %f\n", i+1, cpu_time_used);
       printf("Sorted from 0 to %d in %.2fs\n", i, cpu_time_used);
   }
   fclose(fp);
   fp = fopen("./quickSort.txt", "w+");
   for(int i = 0; i < n; i++) {
       fprintf(fp, "%d\n", a[i]);
   }
   fclose(fp);
   return totalTime;
}

void printArr(int a[], int n) {
   for (int i = 0; i <=n; i++)
       printf("%d\n", a[i]);
}
```

```c
int main()
{
   int n = 100000;
   int a[n],b[n];
   double timeToPopulate = populate(a, b, n);
   printf("Time taken to populate: %f\nSorting...\n",
timeToPopulate);
   double mergeT = mergeCalc(a, n);
   double quickT = qC(b, n);
   printf("Time taken by  Merge Sort: %f\n", mergeT);
   printf("Time taken by  Quick Sort: %f\n", quickT);
   return 0;
}
```

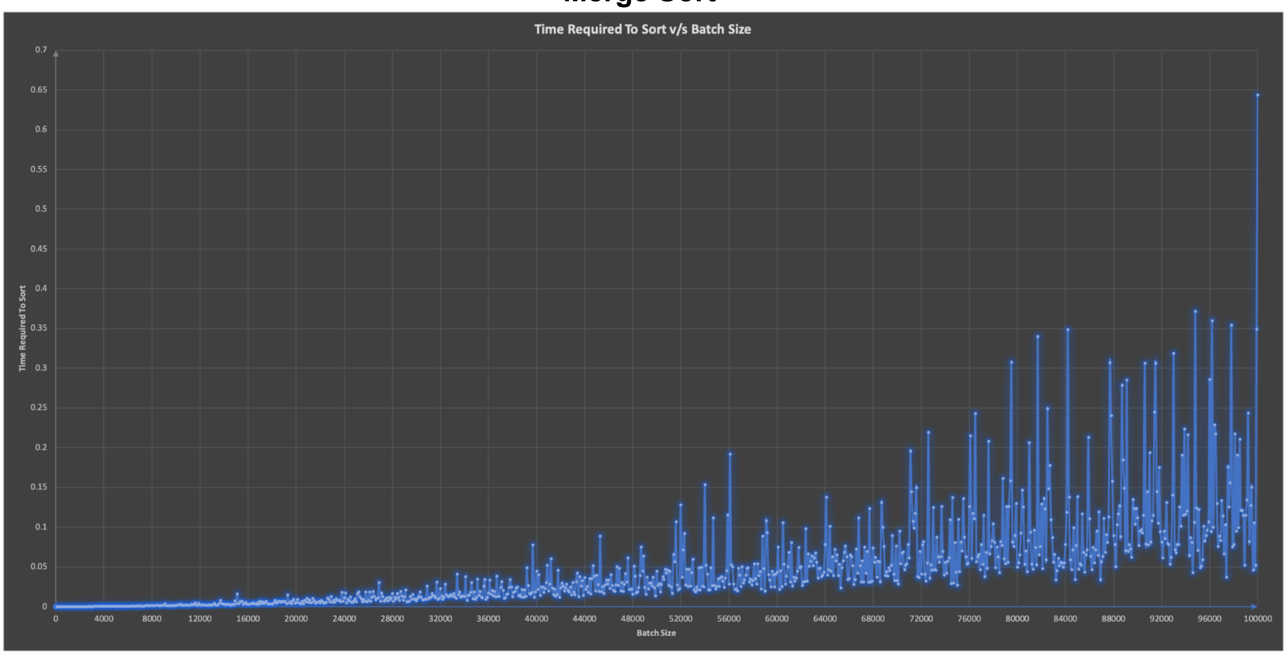
RESULT:

```
Sorted from 0 to 98199 in 0.09s
Sorted from 0 to 98299 in 0.19s
Sorted from 0 to 98399 in 0.10s
Sorted from 0 to 98499 in 0.21s
Sorted from 0 to 98599 in 0.12s
Sorted from 0 to 98699 in 0.12s
Sorted from 0 to 98799 in 0.11s
Sorted from 0 to 98899 in 0.05s
Sorted from 0 to 98999 in 0.11s
Sorted from 0 to 99099 in 0.13s
Sorted from 0 to 99199 in 0.24s
Sorted from 0 to 99299 in 0.08s
Sorted from 0 to 99399 in 0.13s
Sorted from 0 to 99499 in 0.15s
Sorted from 0 to 99599 in 0.05s
Sorted from 0 to 99699 in 0.11s
Sorted from 0 to 99799 in 0.05s
Sorted from 0 to 99899 in 0.35s
Sorted from 0 to 99999 in 0.64s
Time taken by  Merge Sort: 2.707178
Time taken by  Quick Sort: 47.935444
 *  Terminal will be reused by tasks, press any key to close it.
```

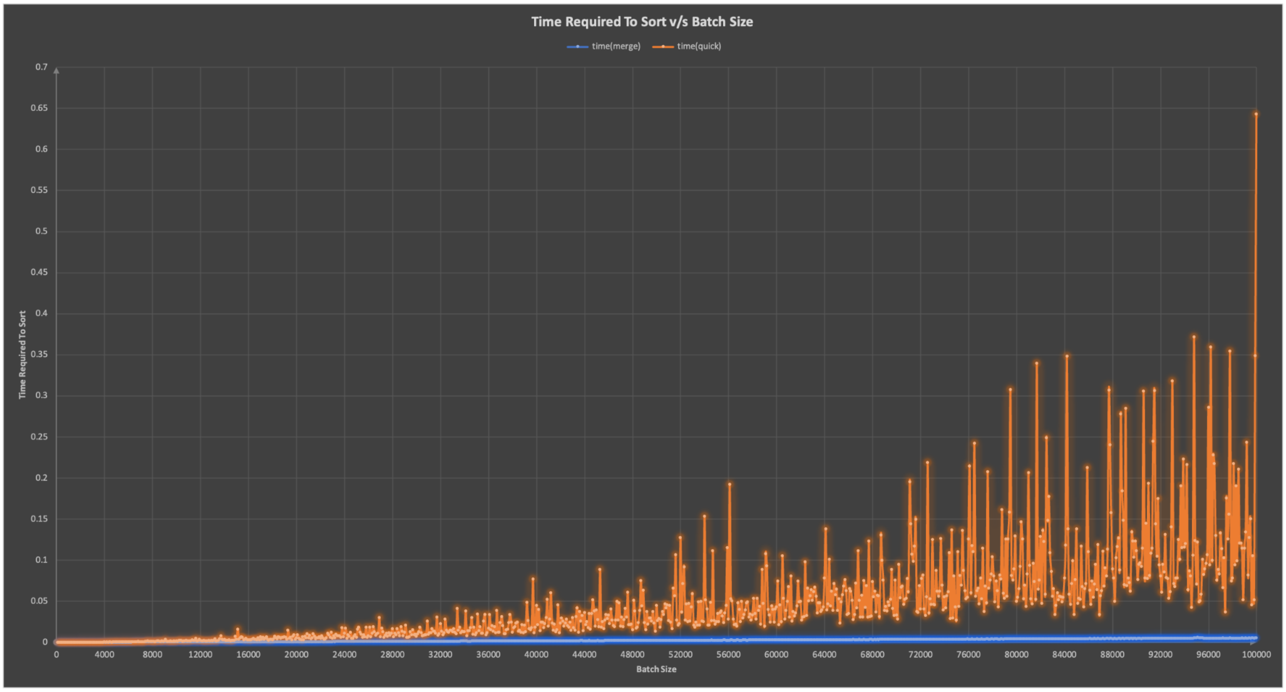
Truncated Terminal Output



Merge Sort



Quick Sort



Comparison between Merge sort and Quicksort

Merge Sort Complexity:-

- Time
  - Best: O(n log n)
  - Worst: O(n log n)
  - Average: O(n log n)
- Space
  - Best: O(1)
  - Worst: O(n)
  - Average: O(n)

Quick Sort Complexity:-

- Time
  - Best: O(n log n)
  - Worst: O(n^2)
  - Average: O(n log n)
- Space
  - Best: O(log n)
  - Worst: O(n)
  - Average: O(log n)

Quick sort can be faster than merge sort for small or nearly sorted lists, but merge sort is generally considered to be more efficient for large or unsorted lists.

CONCLUSION:	Successfully understood the Divide and Conquer approach with the help of sorting algorithms; namely merge sort and quicksort. Also got a better understanding of their time complexities by monitoring the run time measure in batches while sorting.