Bharatiya Vidya Bhavan's

Sardar Patel Institute of Technology

Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India

(Autonomous College Affiliated to University of Mumbai)

Name	Tejas Jadhav
UID No.	2022301006
Class	COMPS A (B batch)
Experiment No.	02

Aim: Experiment based on divide and conquers approach

Theory:

❖ **Divide and Conquer Approach**

Divide and conquer is a problem-solving approach that involves breaking a problem down into smaller sub-problems. This process goes on until we encounter a base case subproblem that can be easily solved.

- The problem is divided into smaller sub-problems, which are easier to solve.
- Each sub-problem is solved independently, either by solving it directly or by breaking it down further.
- The solutions to the sub-problems are combined to arrive at the solution for the original problem.
- This approach is often used for problems that can be broken down into smaller, independent sub-problems.
- Common algorithms that use divide and conquer include merge sort, quicksort, and binary search.
- The divide and conquer approach can help reduce the complexity of a problem and make it easier to solve.

The Divide and Conquer approach have 3 important steps:

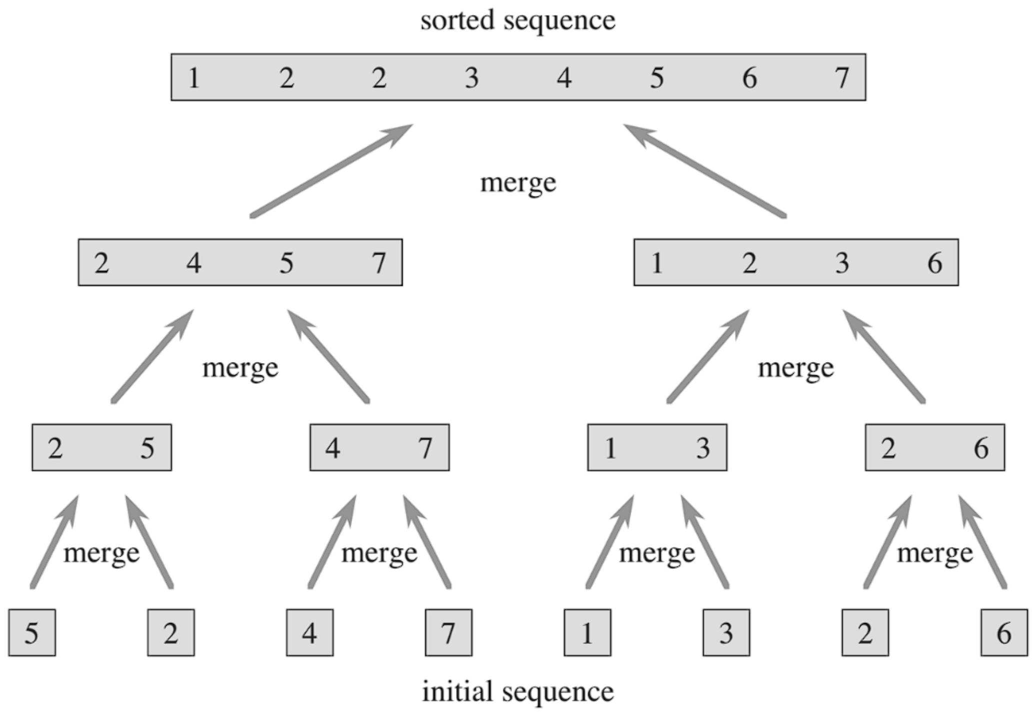
1. **Divide:** The problem is divided into smaller subproblems that are easier to solve. This can involve breaking the problem down into smaller, independent parts or identifying patterns in the problem that can be exploited to simplify the solution.

2. **Conquer:** Each subproblem is solved independently, either by solving it directly or by applying the divide and conquer approach recursively to break it down further. This step typically involves applying the same algorithm or problem-solving strategy to each subproblem.
3. **Combine:** The solutions to the subproblems are combined to arrive at the solution for the original problem. This step may involve merging sorted lists, aggregating results, or using the solutions to the subproblems to guide further processing. The goal is to use the solutions to the subproblems to derive the solution to the original problem.

❖ Merge Sort

Merge sort is a divide and conquer algorithm that sorts an array by breaking it down into smaller subarrays, sorting those subarrays, and then merging them back together.

- The algorithm works by recursively dividing the input array in half until each subarray contains only one element, which is already sorted.
- The algorithm then merges each pair of adjacent subarrays into a single, sorted array using a helper function called "merge".
- The "merge" function takes two sorted subarrays and combines them into a single sorted array by comparing the smallest elements in each subarray and placing them in order.
- The merge operation is repeated until all subarrays have been merged into a single sorted array, which is the final output of the algorithm.
- Merge sort is a stable sort algorithm, meaning that it maintains the relative order of equal elements in the input array.
- Although merge sort has a time complexity of O(n log n), it requires additional memory space to store the subarrays during the recursive calls, making it less memory-efficient than some other sorting algorithms.



Merge Sort

❖ Quick Sort

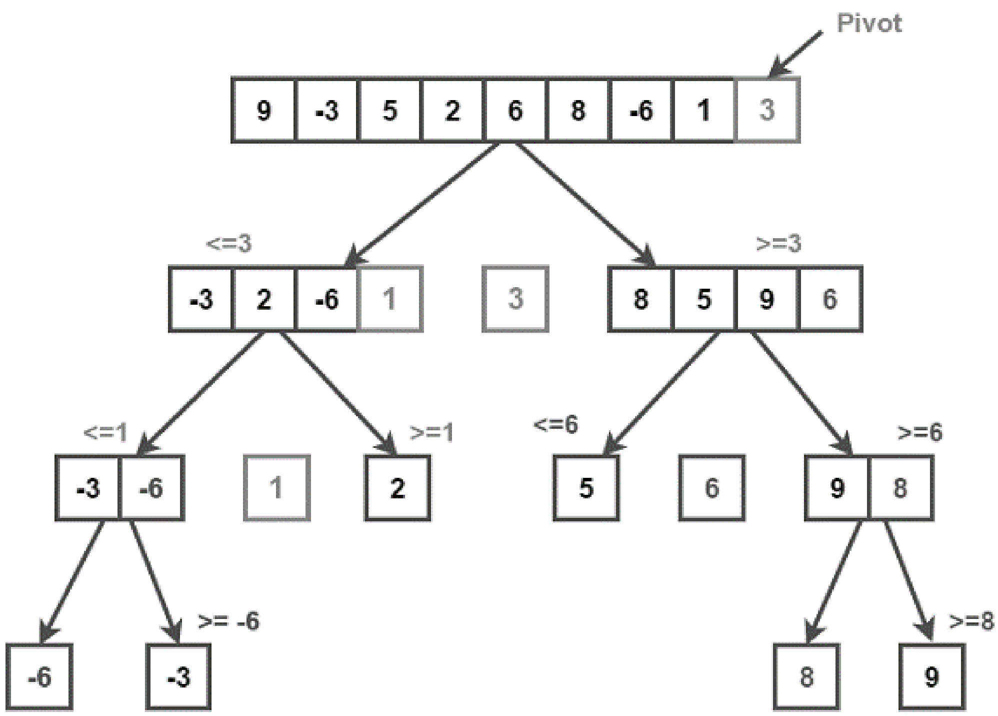
Quick sort is a divide and conquer algorithm that sorts an array by selecting a "pivot" element, partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot, and then recursively sorting the sub-arrays.

- The algorithm starts by selecting a pivot element from the input array.

Some Pivot Selection strategy:

1) First Element
2) Last Element
3) Random Element
4) Median

- The selected pivot is then placed in its correct position in the sorted array by partitioning the array into two sub-arrays, with one containing elements smaller than the pivot, and the other containing elements larger than the pivot.
- The partitioning is typically done by using two pointers, one that starts at the left end of the array and moves right, and another that starts at the right end of the array and moves left. When the pointers find elements that are in the wrong sub-array, they are swapped.
- After the partitioning is complete, the algorithm recursively sorts the two sub-arrays using the same approach until the entire array is sorted.



Quick Sort (pivot selection : last element)

Analysis:

* Merge sort analysis.

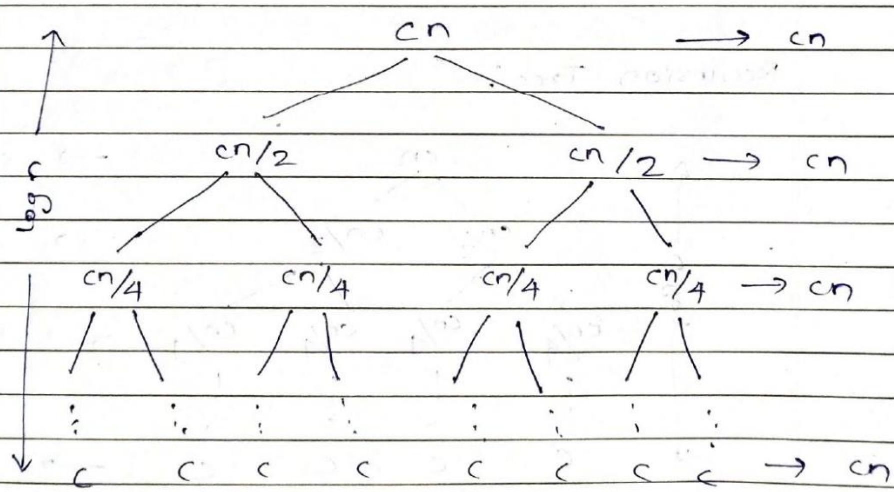
Recurrence (for all cases):

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(n/2) + c.n & n > 1 \end{cases}$$

consider $n > 1$.

$\therefore \quad T(n) = 2T(n/2) + c.n$

Using Recursive tree method :



Height of recursion tree $= \log n$

cost at each level $= cn$

$\therefore$ Total cost $= cn . \log n$

$\therefore$ Time complexity $= \Theta(n \log n)$

* Quick sort Analysis :

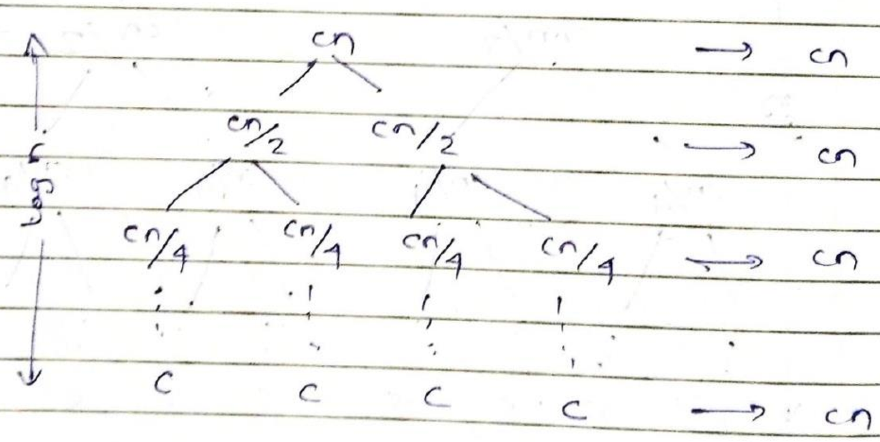
- Best case

In best case, pivot equally divides the subarrays into equal size. i.e. the 2 subarrays have same size for every recursion.

Recurrence :

$$T(n) = \underbrace{2T(n/2)}_{\text{Recursion}} + \underbrace{cn}_{\text{Partition()}}$$

Recursion Tree :



Total cost = $cn \cdot \log n$

Best Case Time Complexity = $\Omega(n \log n)$

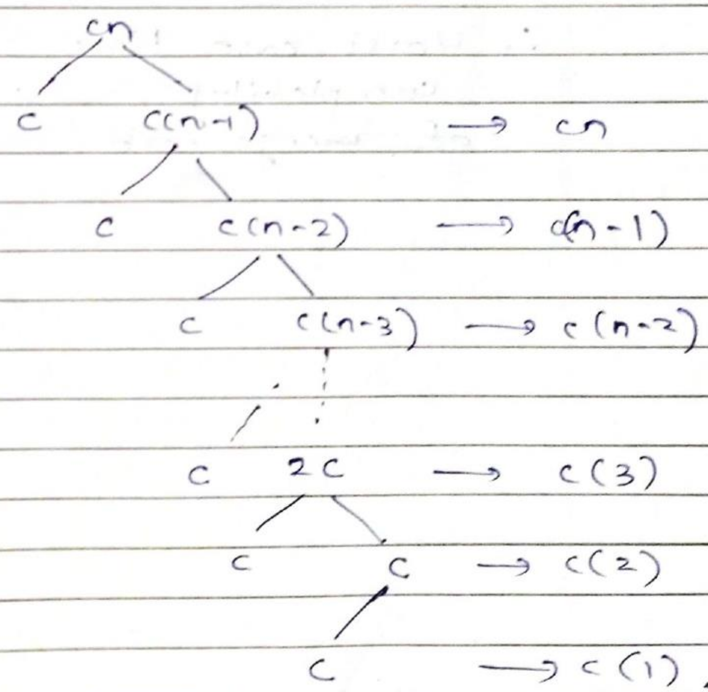
- Worst case

In worst case, pivot divides the subarray, such that one subarry has only 1 element and another subarray has (n-1) elements. For every recursion.

Recurrence :-

$$T(n) = T(n-1) + cn.$$

Recursion Tree :



Total cost = $cn + c(n-1) + c(n-2) + c(n-3) + \ldots \ldots 3c + 2c + c$

$= c[n + (n-1) + (n-2) + \ldots \; 3 + 2 + 1]$

$= c \dfrac{(n)(n+1)}{2}$ — sum of n natural numbers.

$T(n) = \dfrac{c\,n(n+1)}{2} = \dfrac{cn^2}{2} + \dfrac{cn}{2}$

$\therefore$ Worst case time complexity $= O(n^2)$.

Algorithm	Best case	Average Case	Worst Case
Merge Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
Quick Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$

Algorithm:

```
Merge(A, q, p, r)
        n1 = q – p + 1
        n2 = r – q
        let L[1..n1 + 1] and R[1..n2+1] be new arrays
        for i = 1 to n1
                L[i] = A[p + i – 1]
        For j = 1 to n2
                R[j] = A[q + j]
        L[n1 + 1] = ∞
        R[n2 + 1] = ∞
        i = 1
        j = 2
        for k = p to r
                if L[i] <= R[j]
                        A[k] = L[i]
                        i = i + 1
                else A[k] = R[j]
                        j = j + 1


MergeSort(A, p, r)
        if p < r
                q = (p + r) / 2
                MergeSort(A, p, q)
                MergeSort(A, q+1, r)
                Merge(A, p, q, r)
```

```
// considering last element as pivot

Partition(A, l, h)
        pivot = h
        low = l
        high = h – 1
        while low < high
                while a[low] <= a[pivot] and low < h
                        low = low + 1
                while a[high] > a[pivot]  and high >= l
                        high = high – 1
                if low < high
                        swap(a[low], a[high])
        if a[low] > a[high]
                swap(a[low], a[pivot]
        return low


QuickSort(a, l, h)
        if l < h
                pivot = partition(a, l, h)
                QuickSort(a, l, pivot – 1)
                QuickSort(a, pivot + 1, h)
```

Code:

```
#include <bits/stdc++.h>
#include <stdio.h>
#include <iostream>

#include <chrono>
#include <fstream>
using namespace std;

double merge_comparision = 0.0;
double quick_comparision = 0.0;

void print_array(int* a, int size) {
    for (int i = 0; i < size; i++) {
        cout << a[i] << " ";
    }
    cout << endl;
}
void merge(int* arr, int l, int m, int h) {
    int left_size = m - l + 1;
    int right_size = h - m;

    int left[left_size];
    int right[right_size];

    int x = 0;
    int y = 0;
    int z = l;

    for (int i = l; i <= m; i++) {
        left[x++] = arr[i];
    }
    for (int i = m + 1; i <= h; i++) {
        right[y++] = arr[i];
    }

    x = y = 0;
    while (x < left_size && y < right_size) {
        arr[z++] = left[x] <= right[y] ? left[x++] : right[y++];
        merge_comparision++;
```

```
    }

    while (x < left_size) {
        arr[z++] = left[x++];
        merge_comparision++;
    }
    while (y < right_size) {
        arr[z++] = right[y++];
        merge_comparision++;
    }
}
void merge_sort(int* arr, int l, int h) {
    if (l < h) {
        int mid = (l + h) / 2;
        merge_sort(arr, l, mid);
        merge_sort(arr, mid + 1, h);
        merge(arr, l, mid, h);
    }
}

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

int partition(int* a, int l, int h) {
    // pivot selection: last element
    int pivot = h;
    int low = l;
    int high = h - 1;

    while (low < high) {
        while (a[low] <= a[pivot] && low < h) {
            quick_comparision++;
            low++;
        }

        while (a[high] > a[pivot] && high >= l) {
            quick_comparision++;
            high--;
```

```
        }

        if (low < high) {
            quick_comparision++;
            swap(a[low], a[high]);
        }
    }
    if (a[low] > a[pivot])
        swap(a[low], a[pivot]);
    return low;
}

void quick_sort(int* a, int l, int h) {
    if (l < h) {
        int pivot = partition(a, l, h);
        quick_sort(a, l, pivot - 1);
        quick_sort(a, pivot + 1, h);
    }
}

int digits(int num) {
    return num == 0 ? 1 : floor(log10(abs(num))) + 1;
}

int main() {
    int arr_mer[100000];
    int arr_qui[100000];

    ifstream nums("random_numbers.txt");
    ofstream output("../csv/sort_analysis.csv");
    ofstream comparison("../csv/comparison.csv");

    output << "block_size,merge,quick\n";
    comparison << "block_no,merge,quick\n";

    for (int i = 1; i <= 100000; i++) {
        nums >> arr_mer[i];
        arr_qui[i] = arr_mer[i];
    }

    for (int i = 1; i <= 1000; i++) {
```

```
        // print 10 values at index 10000, 20000, ...
        int index = i * 100;
        if (index % 10000 == 0 && index != 100000) {
            cout << "\nPrinting 10 values from index " << index << endl;
            for (int t = 0; t < 10; t++) {
                cout << index + t << " : " << arr_mer[index + t] << "\n";
            }
        }

        // merge sort
        auto merge_start = chrono::high_resolution_clock::now();
        merge_sort(arr_mer, 0, i * 100 - 1);
        auto merge_end = chrono::high_resolution_clock::now();
        chrono::duration<double> merge_time = (merge_end - merge_start);

        // quick sort
        auto quick_start = chrono::high_resolution_clock::now();
        quick_sort(arr_qui, 0, i * 100 - 1);
        auto quick_end = chrono::high_resolution_clock::now();
        chrono::duration<double> quick_time = (quick_end - quick_start);

        output << i * 100 << "," << merge_time.count() << ","
               << quick_time.count() << "\n";

        comparison << i << "," << merge_comparision << "," <<
quick_comparision << "\n";
    }

    cout << "\nSorting completed !" << endl;

    cout << "\nSmallest Number = " << arr_mer[0] << "\tDigits = " <<
digits(arr_mer[0]) << endl;
    cout << "Largest Number = " << arr_mer[99999] << "\tDigits = " <<
digits(arr_mer[99999]) << endl;

    printf("Merge sort comparision count: %.0lf\n", merge_comparision);
    printf("Quick sort comparision count: %.0lf\n", quick_comparision);

    return 0;
}
```

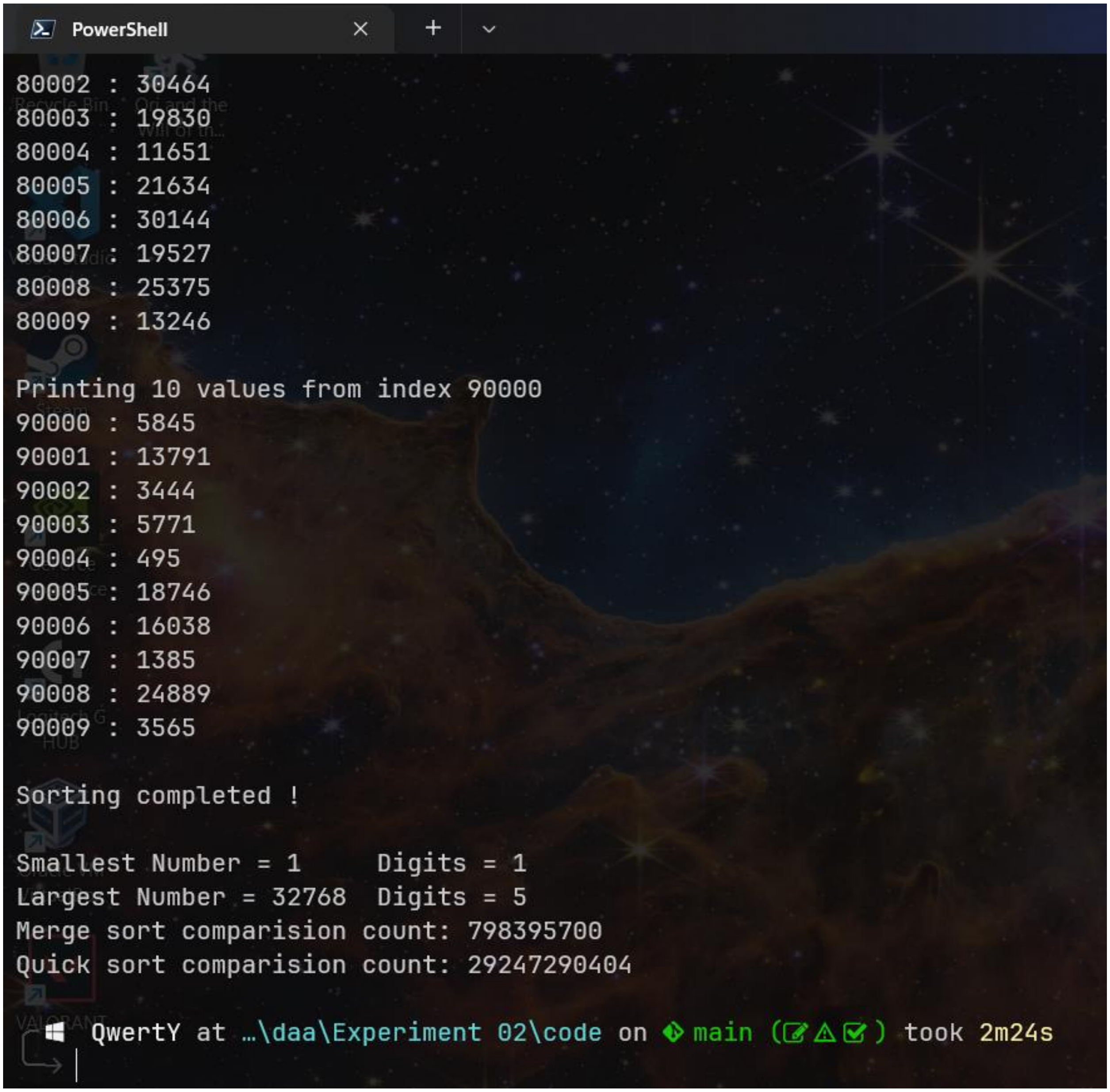
Output:

```
PowerShell

QwertY at …\daa\Experiment 02\code on main
> g++ sort_analysis.cpp

QwertY at …\daa\Experiment 02\code on main
> ./a

Printing 10 values from index 10000
10000 : 18797
10001 : 19485
10002 : 22694
10003 : 17195
10004 : 10606
10005 : 17060
10006 : 6839
10007 : 20557
10008 : 15180
10009 : 18297

Printing 10 values from index 20000
20000 : 6052
20001 : 3389
20002 : 16113
20003 : 3148
20004 : 28421
20005 : 20456
20006 : 5575
20007 : 16257
20008 : 29056
20009 : 10862

Printing 10 values from index 30000
30000 : 20861
30001 : 8971
30002 : 8321
30003 : 14253
```

PowerShell

```
30003 : 14253
30004 : 20913
30005 : 31361
30006 : 6955
30007 : 10444
30008 : 11839
30009 : 23377

Printing 10 values from index 40000
40000 : 21003
40001 : 4168
40002 : 14612
40003 : 17941
40004 : 13077
40005 : 29056
40006 : 8622
40007 : 7111
40008 : 26302
40009 : 3715

Printing 10 values from index 50000
50000 : 14500
50001 : 2805
50002 : 10919
50003 : 1421
50004 : 3351
50005 : 6964
50006 : 31326
50007 : 6475
50008 : 15474
50009 : 29607

Printing 10 values from index 60000
60000 : 14556
60001 : 3155
```

```
PowerShell
60003 : 8544
60004 : 16860
60005 : 9520
60006 : 8329
60007 : 9073
60008 : 19728
60009 : 17819

Printing 10 values from index 70000
70000 : 27273
70001 : 24804
70002 : 32768
70003 : 12821
70004 : 27593
70005 : 5188
70006 : 2490
70007 : 19858
70008 : 653
70009 : 23171

Printing 10 values from index 80000
80000 : 6592
80001 : 14439
80002 : 30464
80003 : 19830
80004 : 11651
80005 : 21634
80006 : 30144
80007 : 19527
80008 : 25375
80009 : 13246

Printing 10 values from index 90000
90000 : 5845
90001 : 13791
```



```
PowerShell
80002 : 30464
80003 : 19830
80004 : 11651
80005 : 21634
80006 : 30144
80007 : 19527
80008 : 25375
80009 : 13246

Printing 10 values from index 90000
90000 : 5845
90001 : 13791
90002 : 3444
90003 : 5771
90004 : 495
90005 : 18746
90006 : 16038
90007 : 1385
90008 : 24889
90009 : 3565

Sorting completed !

Smallest Number = 1     Digits = 1
Largest Number = 32768  Digits = 5
Merge sort comparision count: 798395700
Quick sort comparision count: 29247290404

QwertY at …\daa\Experiment 02\code on main took 2m24s
```

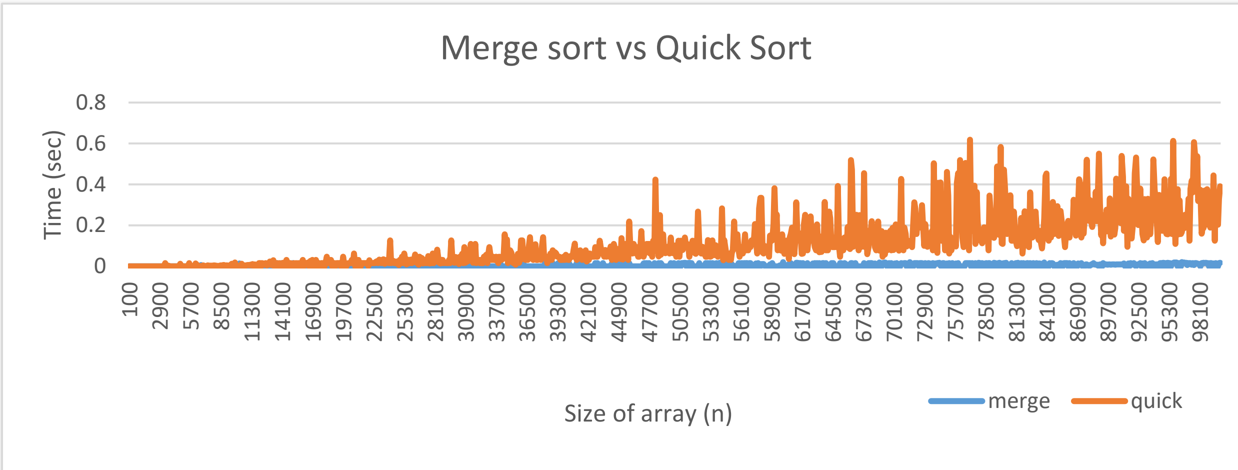
Bharatiya Vidya Bhavan's

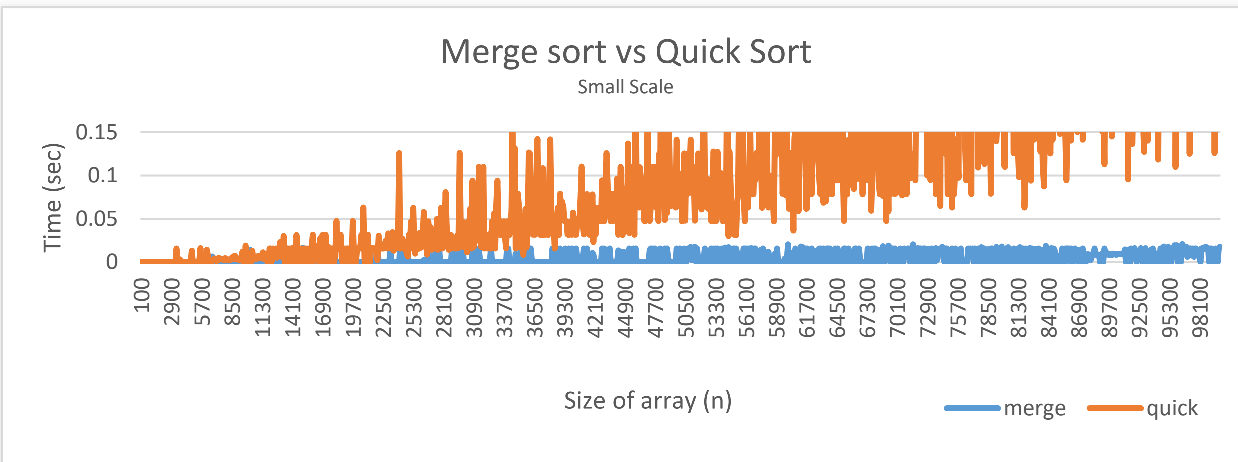
Sardar Patel Institute of Technology

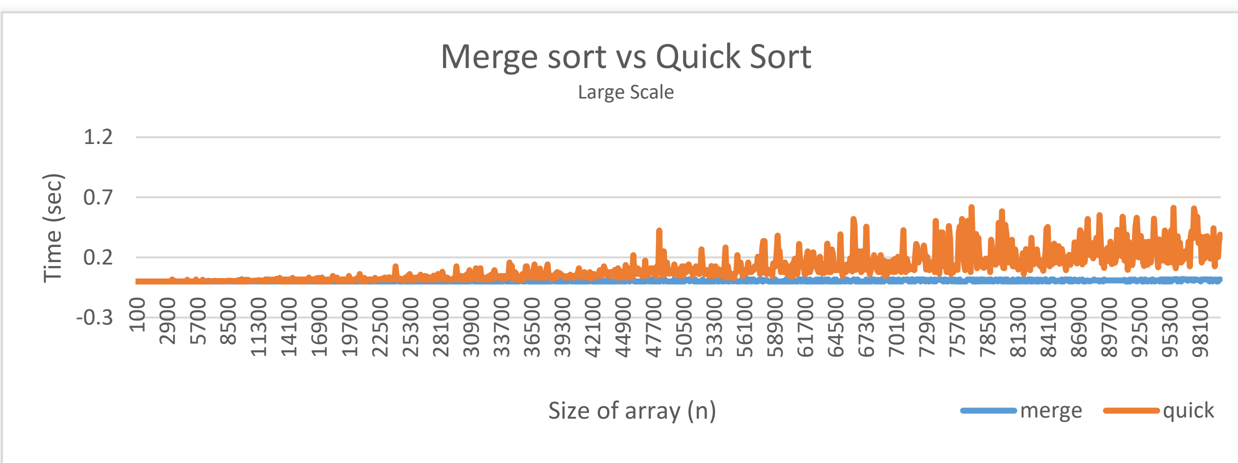
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India

(Autonomous College Affiliated to University of Mumbai)

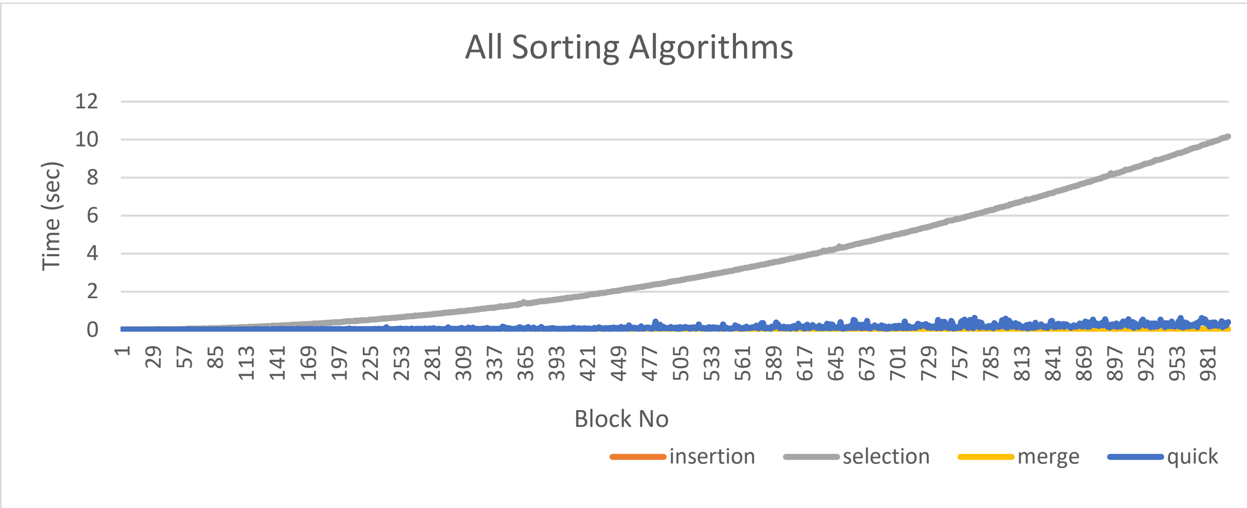
Chart:

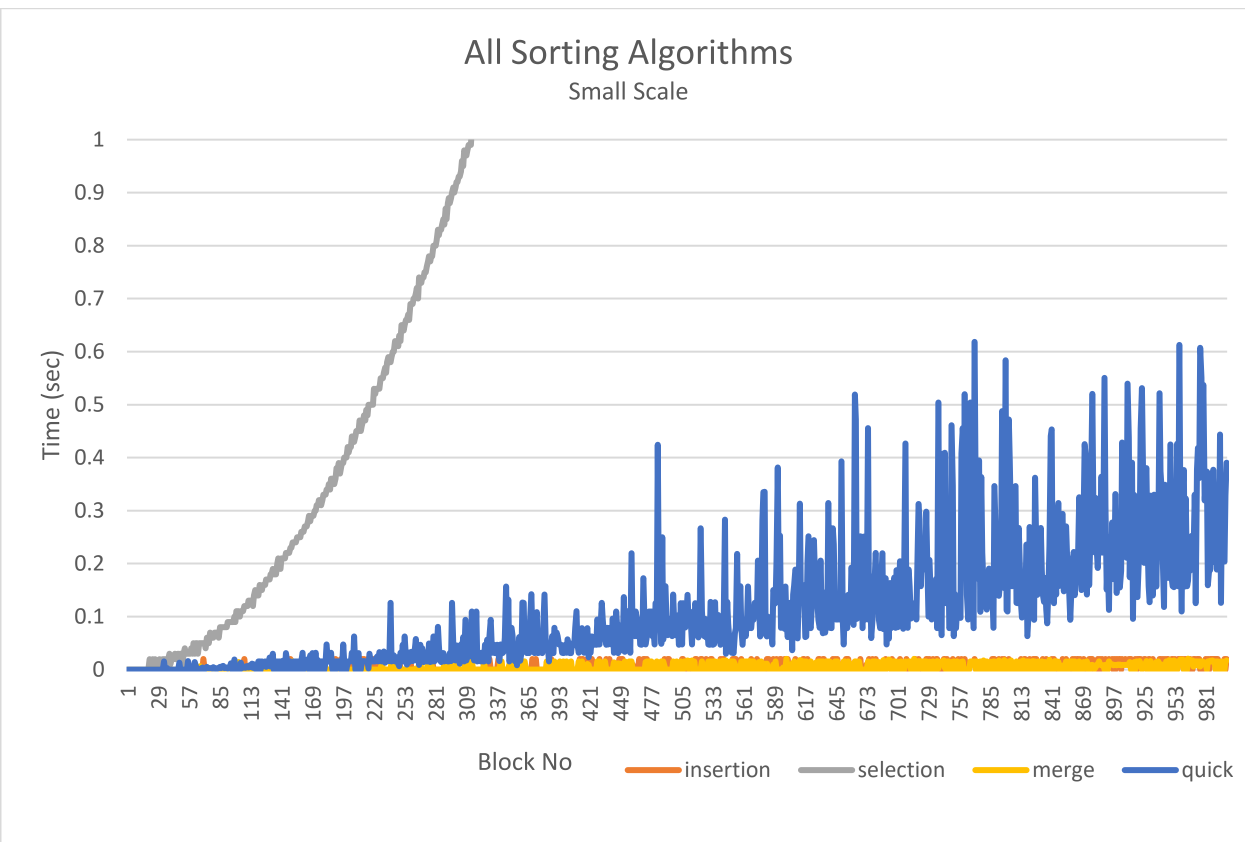


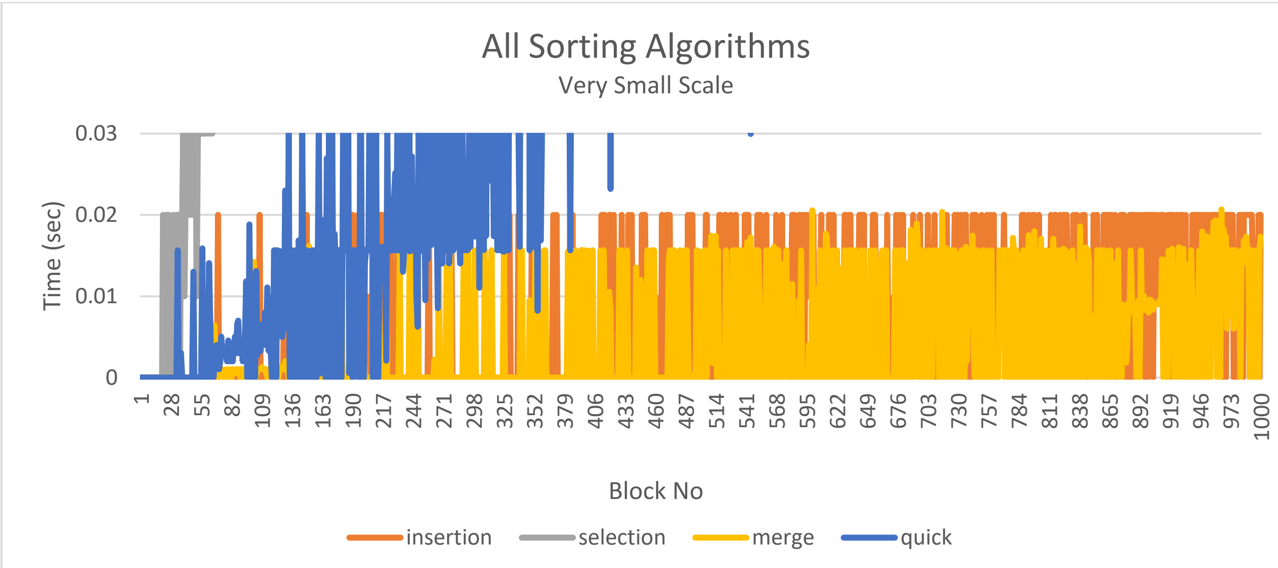




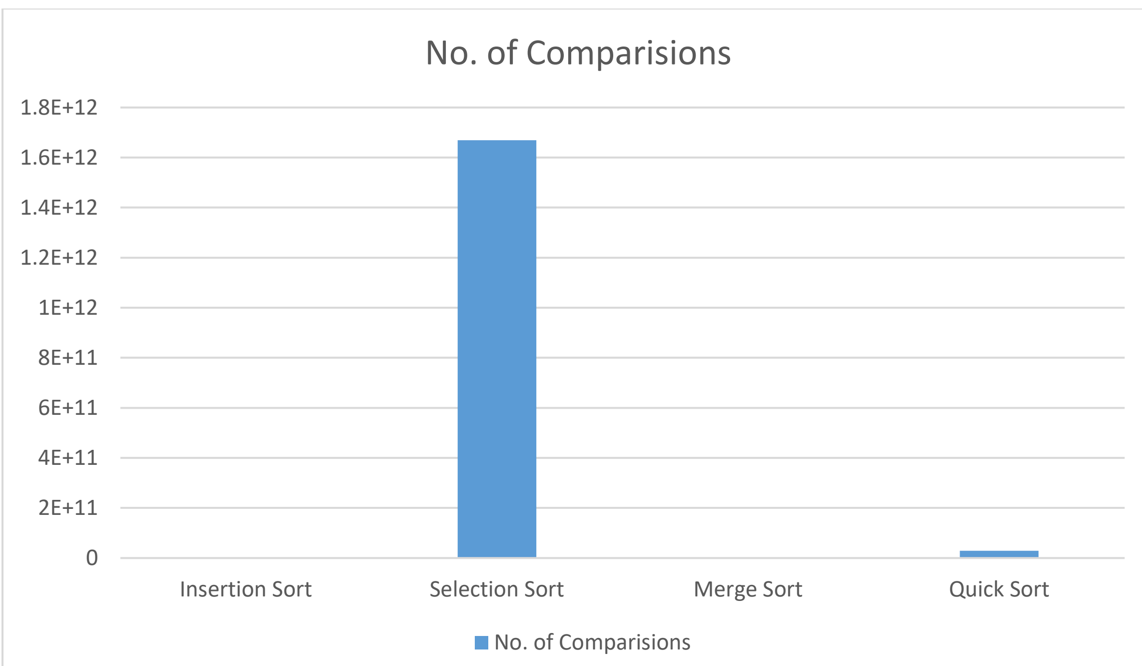
All Sorting Algorithm Comparison

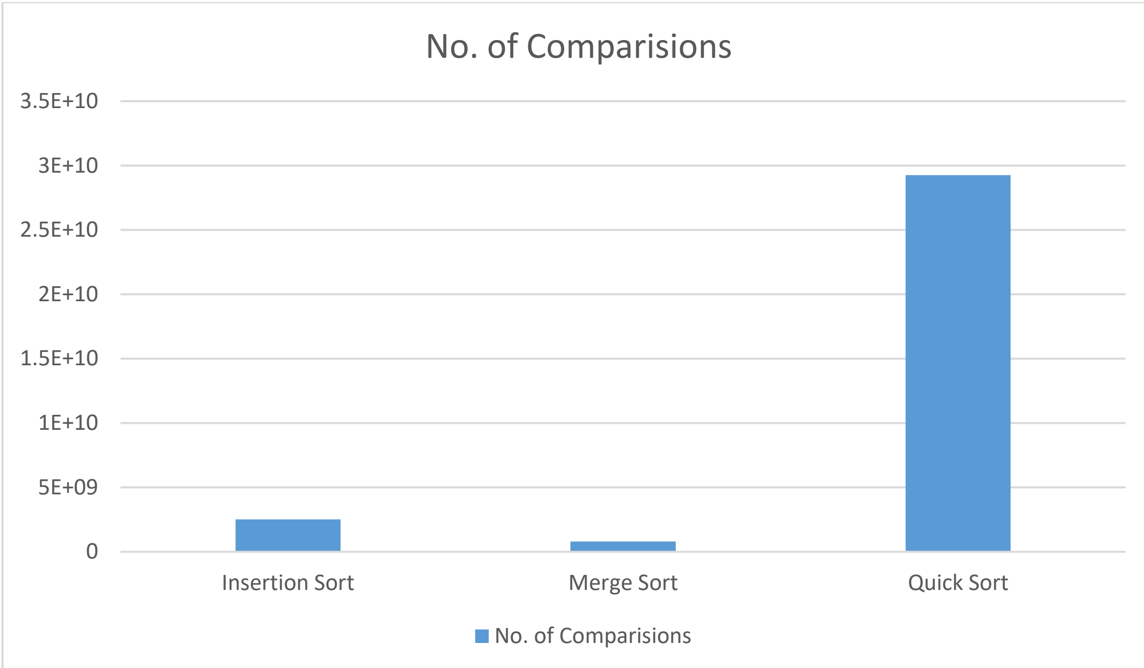


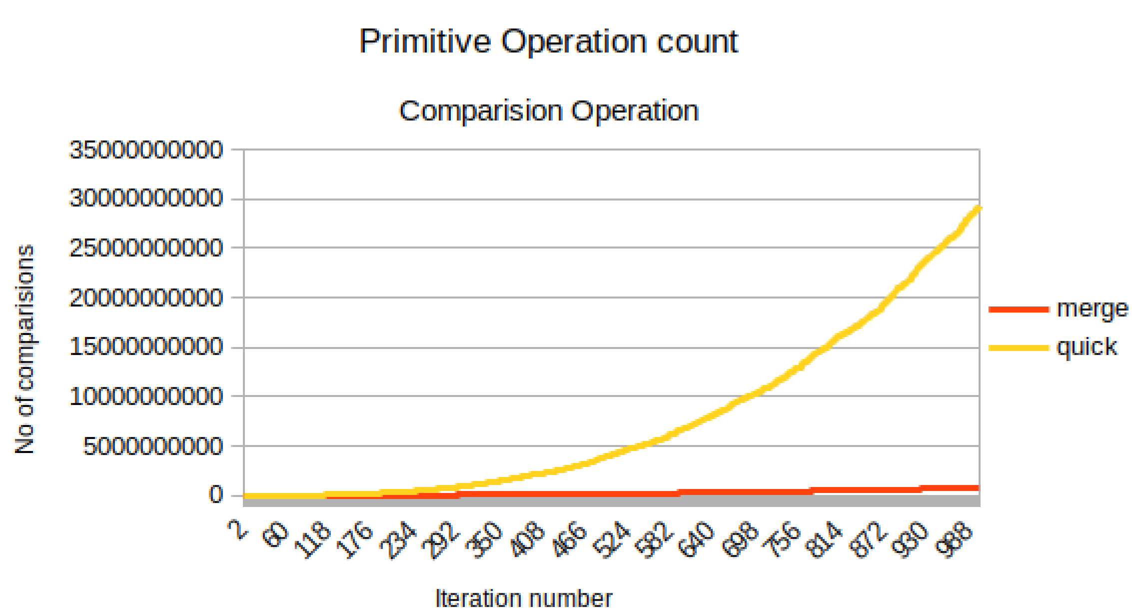




Comparisons:




No. of Comparisions
3.5E+10
3E+10
2.5E+10
2E+10
1.5E+10
1E+10
5E+09
0
Insertion Sort
Merge Sort
Quick Sort
No. of Comparisions


Primitive Operation count
Comparision Operation
No of comparisions
35000000000
30000000000
25000000000
20000000000
15000000000
10000000000
5000000000
0
2
60
118
176
234
292
350
408
466
524
582
640
698
756
814
872
930
988
Iteration number
merge
quick

Observations:

1) Initially both Quick sort and merge sort algorithm took the same time.
2) However, as data size increased, Quick sort became slower.
3) Merge Sort time increased slowly but in a predictable manner.
4) Time required by quick sort was unpredictable as in during one iteration quick sort would take less time and in just the next iteration it would take more time than usual.
5) It is observed that above stated behavior occurs due to how the pivot divides the subarray.
6) Comparing insertion sort, selection sort, merge sort and quick sort, Merge sort is observed to be the fastest.
7) T(merge) < T(insertion) < T(quick) < T(selection)
8) The average time complexity of quick sort is $\Theta(n \log n)$ and average case time complexity of insertion sort is $\Theta(n^2)$ However, Insertion sort compared better than Quick sort.

   Reasoning:

   1) In problem statement, we are sorting in chunks A[0..99], A[0..199], A[0..299], etc
   2) It is observed that only the new additional 100 elements are random and the elements of previous chunk are already sorted.
      Ex.
      Consider that just now A[0..67899] has been sorted
      Next is A[0..67999]
      Only the new elements from 67900 to 67999 are random and unsorted meanwhile all other element are sorted.
      (Even if the new 100 element lie in the sorted part, they only cause upto 100 changes in sorted array)
   3) Insertion Sort, whose best case is $\Omega(n)$ favors heavily here as a major chunk is already sorted
   4) Quick Sort, whose worst case is $O(n^2)$ has a major disadvantage as a major chunk is already sorted

Conclusion:

After conducting this experiment and analyzing the time for all sorting techniques, I conclude the following:

- Divide and Conquer strategy is an elegant and effective way to solve problems.
- I have learnt to analyze the complexity of divide and conquer algorithms and computing their time complexity by solving recurrences.
- The nature of problem statement and initial state of data affects greatly on the runtime of a sorting algorithm.
- An sorting algorithm must be selected on the basis of the current state of data. If most of the data is already sorted, then insertion sort will perform better than quick sort and also merge sort (merge sort requires space O(n))