

|  |  |
| --- | --- |
| Name | Vedant Deshmukh |
| UID No. | 2021300025 |
| 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.



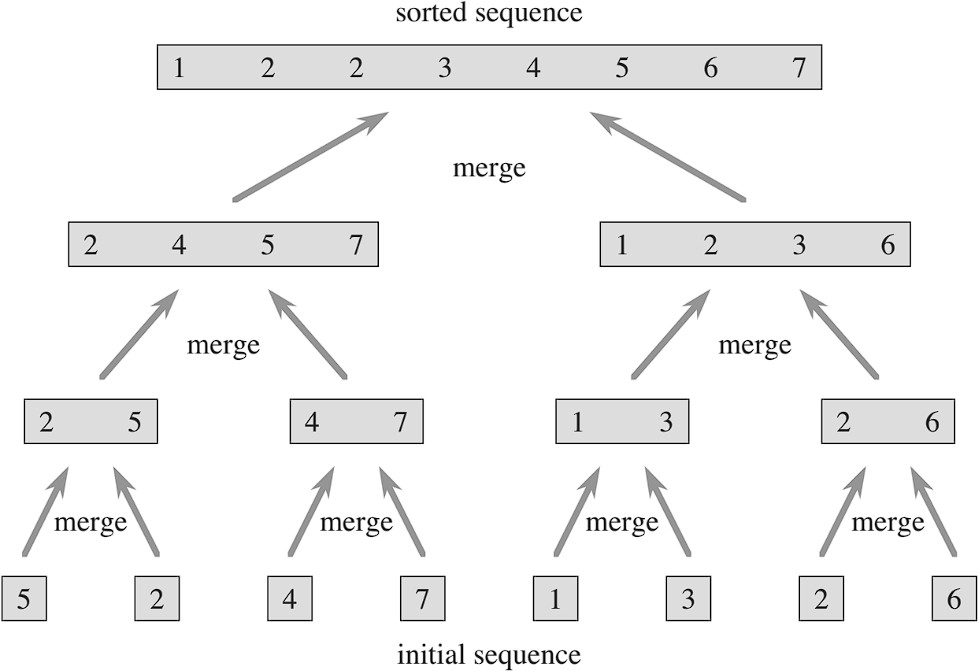
1. **Conquer:** Each subproblem is solved independently, either by solving it directly or by applying the divide and conquer approach recursively to break it down further. This step typically involves applying the same algorithm or problem-solving strategy to each subproblem.
2. **Combine:** The solutions 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

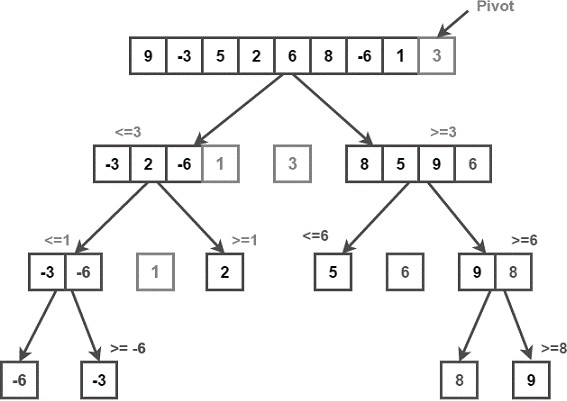
Quick sort is a divide and conquer algorithm that sorts an array by selecting a "pivot" element, partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot, and then recursively sorting the sub-arrays.

* 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



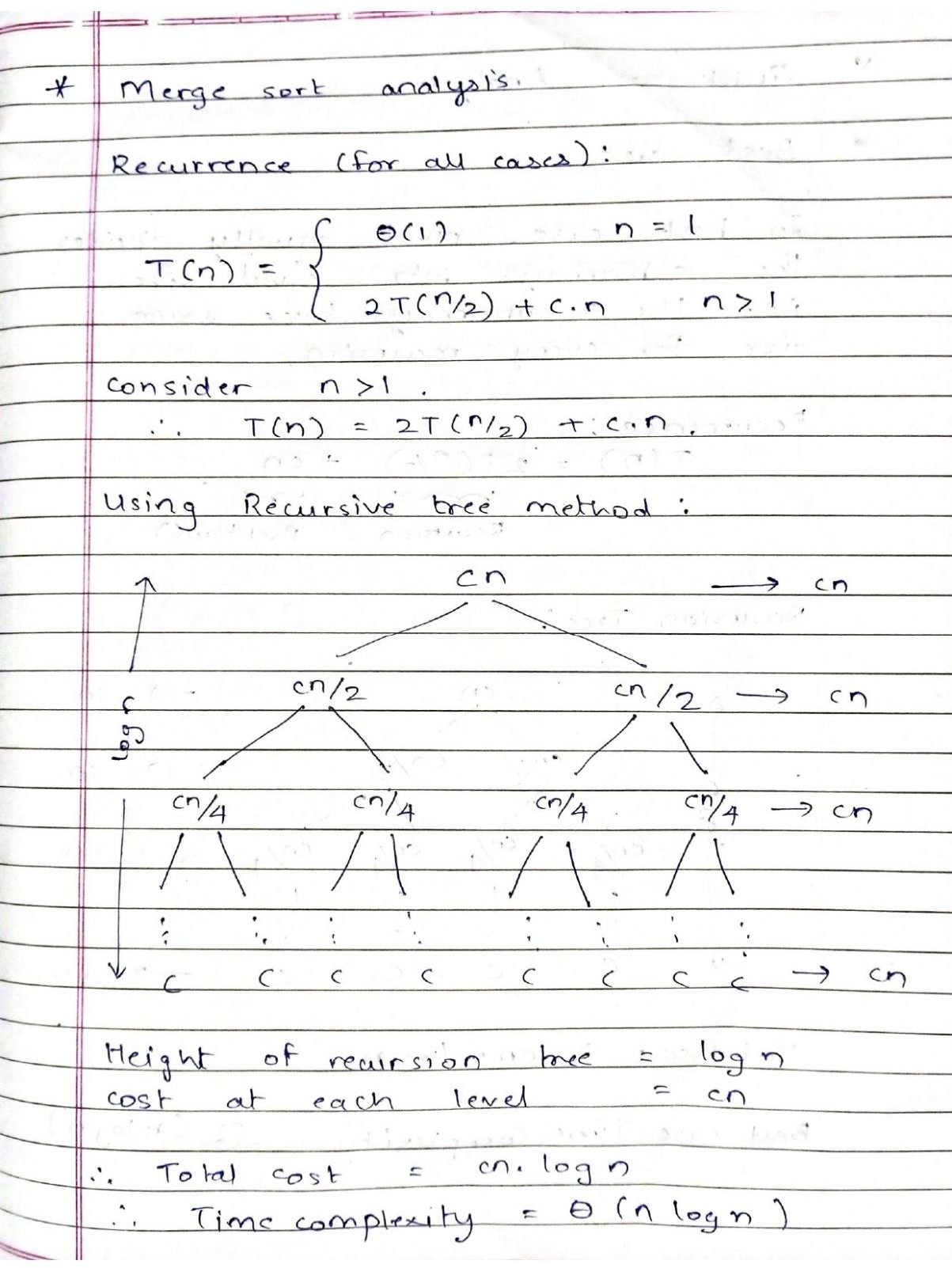
* The selected pivot is then placed in its correct position in the sorted array by partitioning the array into two sub-arrays, with one containing elements smaller than the pivot, and the other containing elements larger than the pivot.
* The partitioning is typically done by using two pointers, one 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.



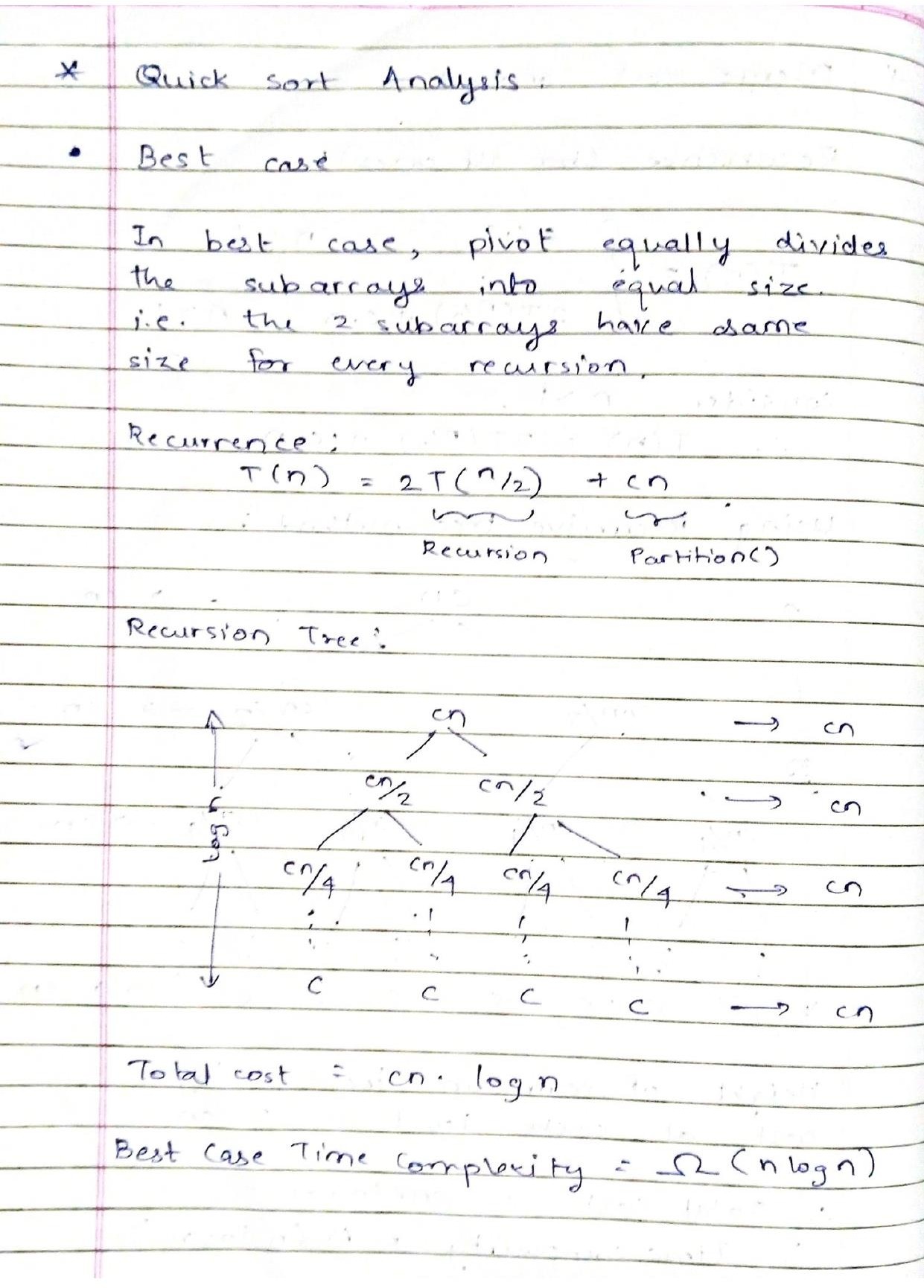
**Quick Sort (pivot selection : last element)**



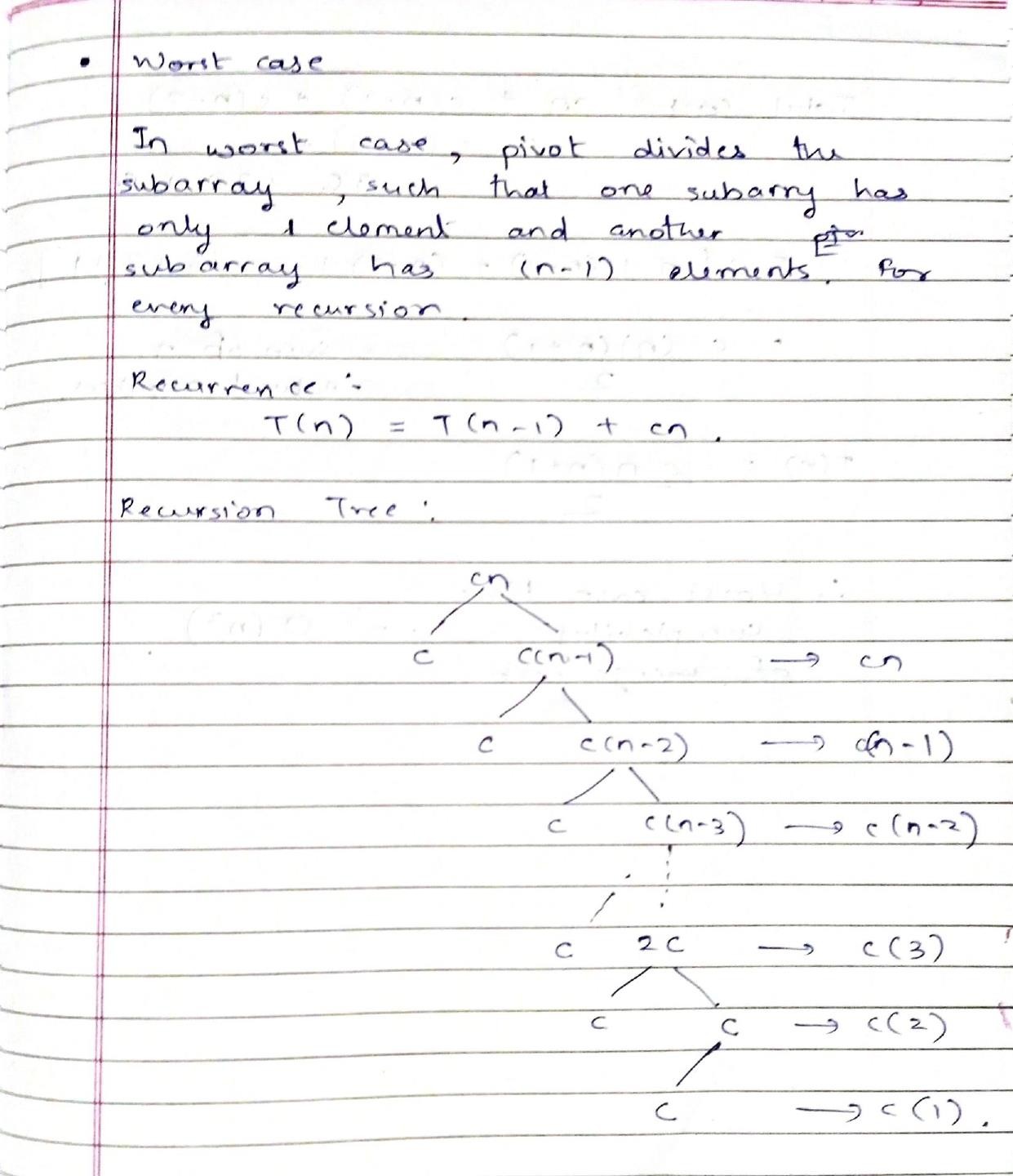
## Analysis:



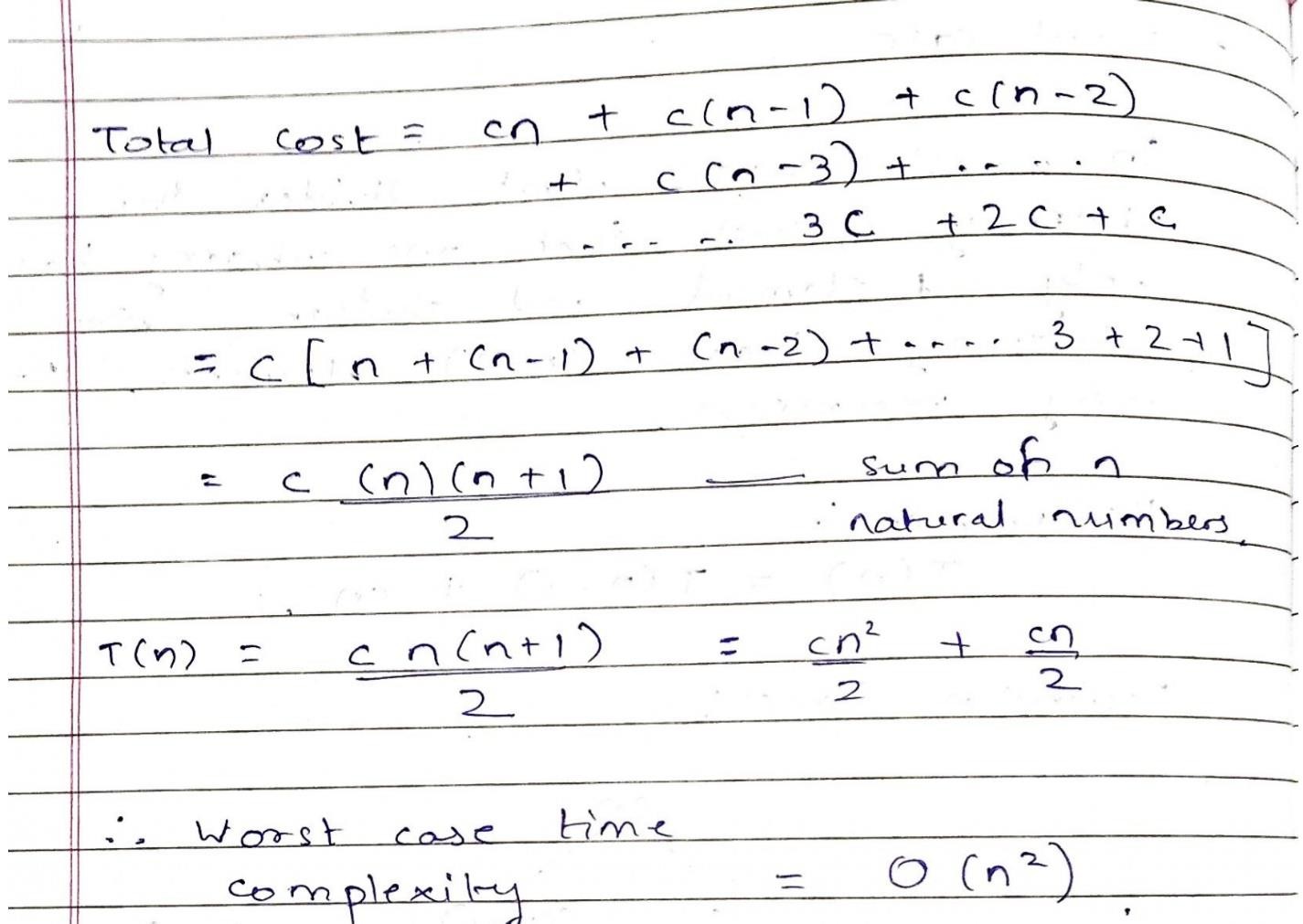












|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Best case** | **Average Case** | **Worst Case** |
| Merge Sort | Ω(*n log n*) | ϴ*(n log n)* | O*(n log n)* |
| Quick Sort | Ω(*n log n*) | ϴ*(n log n)* | O*(n2)* |



**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;

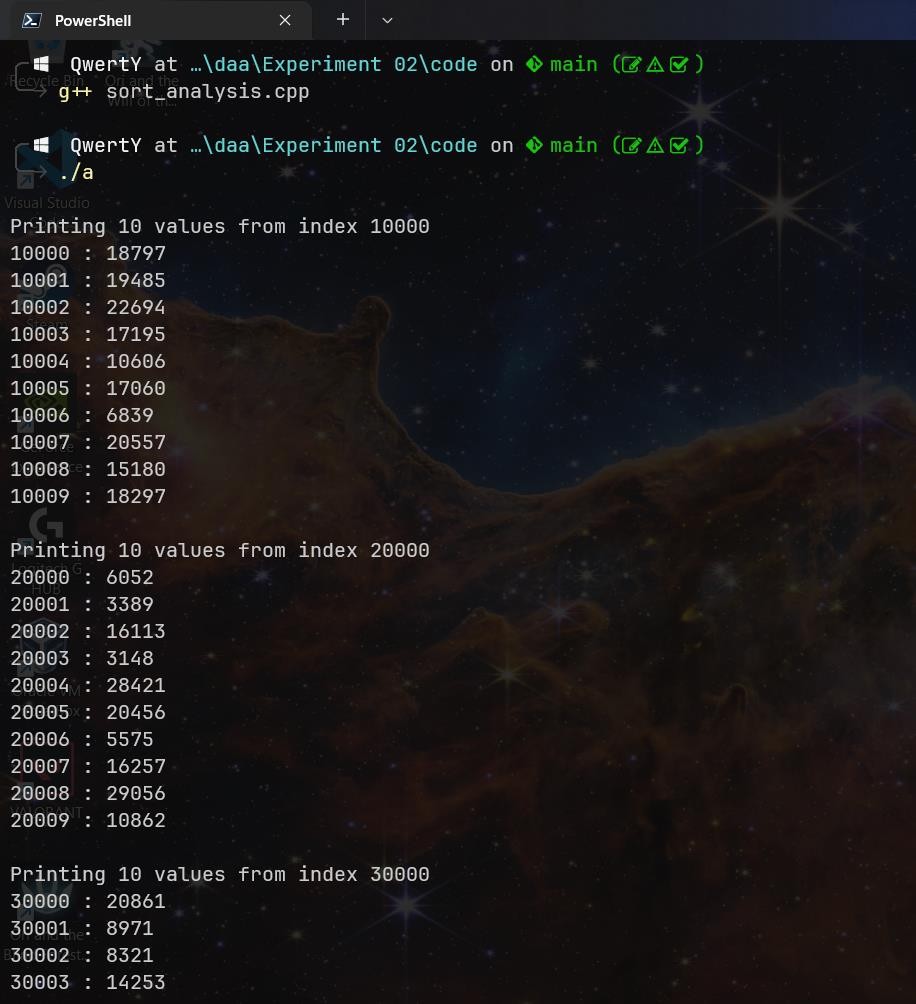
printf("Merge sort comparision count: %.0lf\n", merge\_comparision); printf("Quick sort comparision count: %.0lf\n", quick\_comparision);

return 0;

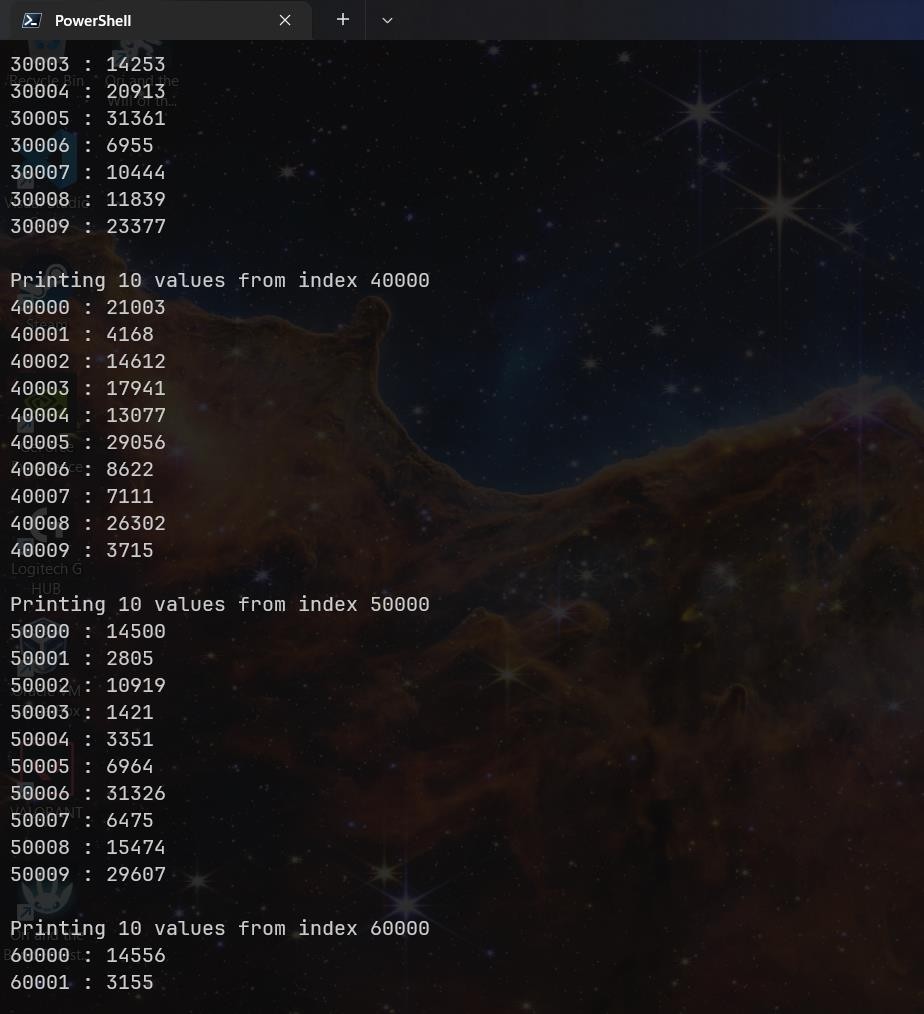
}



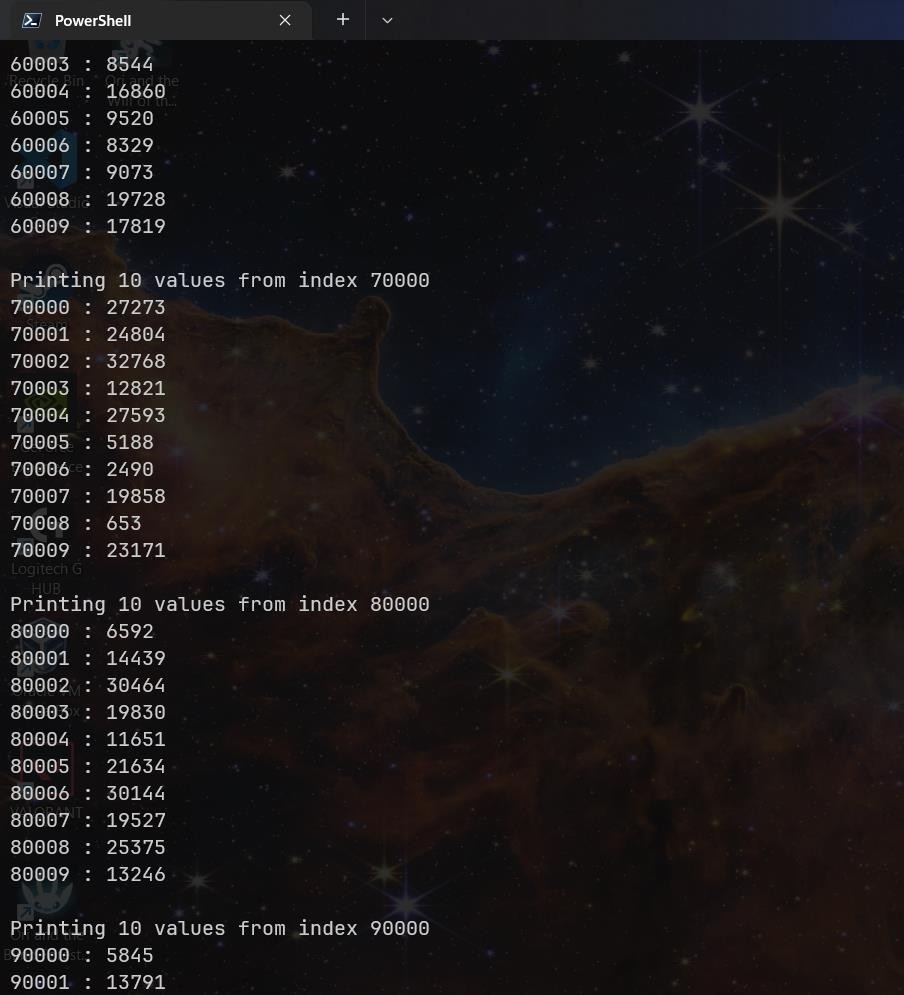
## Output:



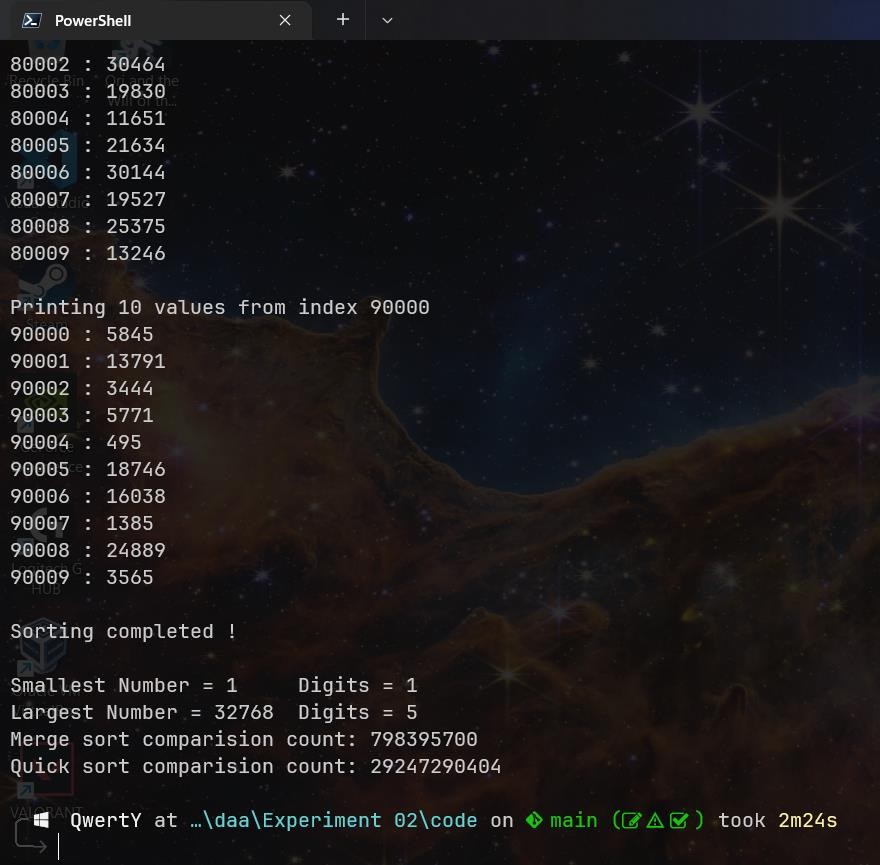


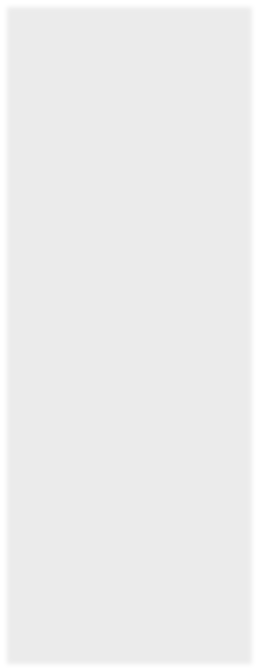
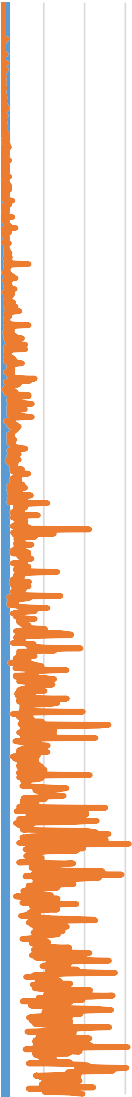
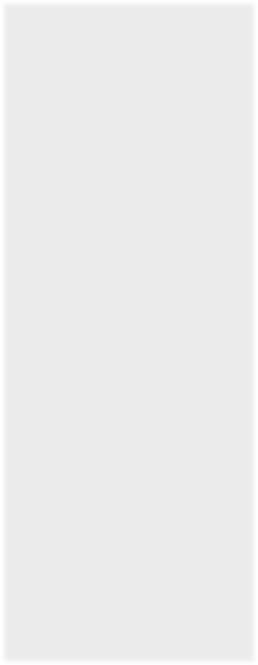












Time (sec)

100

2900

5700

8500

11300

14100

16900

19700

22500

25300

28100

30900

33700

36500

39300

42100

44900

47700

50500

53300

56100

58900

61700

64500

67300

70100

72900

75700

78500

81300

84100

86900

89700

92500

95300

98100

Time (sec)

100

2900

5700

8500

11300

14100

16900

19700

22500

25300

28100

30900

33700

36500

39300

42100

44900

47700

50500

53300

56100

58900

61700

64500

67300

70100

72900

75700

78500

81300

84100

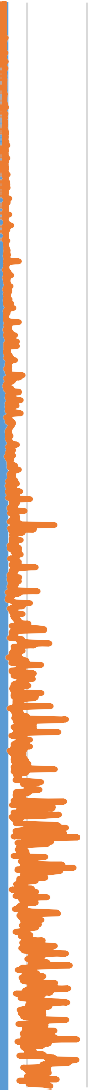
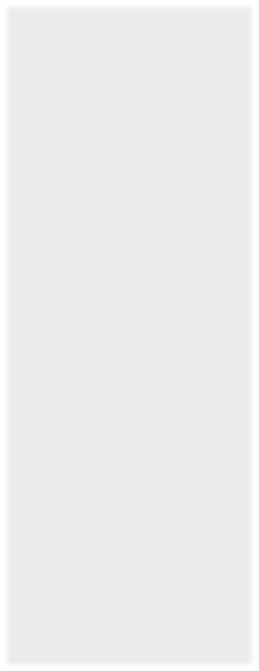
86900

89700

92500

95300

98100



Time (sec)

100

2900

5700

8500

11300

14100

16900

19700

22500

25300

28100

30900

33700

36500

39300

42100

44900

47700

50500

53300

56100

58900

61700

64500

67300

70100

72900

75700

78500

81300

84100

86900

89700

92500

95300

98100

**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:**

Merge sort vs Quick Sort

0.8

0.6

0.4

0.2

0

merge

quick

Size of array (n)

Merge sort vs Quick Sort

Small Scale

0.15

0.1

0.05

0

Size of array (n)

merge

quick

Merge sort vs Quick Sort

Large Scale

1.2

0.7

0.2

-0.3

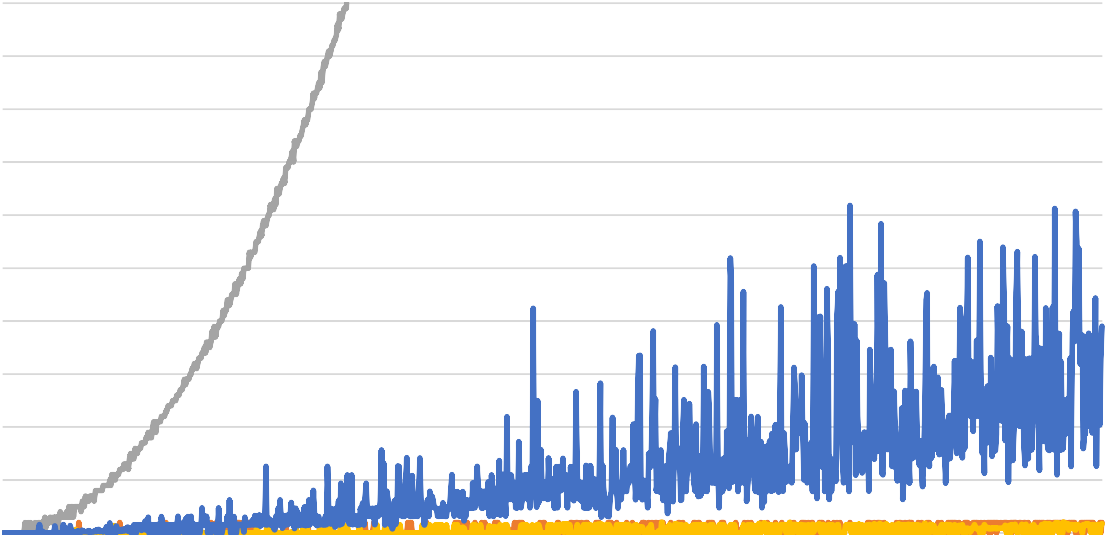
Size of array (n)

merge

quick



**Bharatiya Vidya Bhavan’s**



All Sorting Algorithms

Small Scale

1

0.9

0.8

0.7

0.6

0.5

0.4

0.3

0.2

0.1

0

Block No

insertion

selection

merge

quick

Time (sec)

1

29

57

85

113

141

169

197

225

253

281

309

337

365

393

421

449

477

505

533

561

589

617

645

673

701

729

757

785

813

841

869

897

925

953

981

1

29

57

85

113

141

169

197

225

253

281

309

337

365

393

421

449

477

505

533

561

589

617

645

673

701

729

757

785

813

841

869

897

925

953

981

# Sardar Patel Institute of Technology

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

(Autonomous College Affiliated to University of Mumbai)

## All Sorting Algorithm Comparison

All Sorting Algorithms

12

10

8

6

4

2

0

Block No

insertion

selection

merge

quick

Time (sec)

All Sorting Algorithms

Very Small Scale

0.03

0.02

0.01

0

insertion

Block No

selection merge

quick

Time (sec)



**Comparisons:**

1

28

55

82

109

136

163

190

217

244

271

298

325

352

379

406

433

460

487

514

541

568

595

622

649

676

703

730

757

784

811

838

865

892

919

946

973

1000

No. of Comparisions

1.8E+12

1.6E+12

1.4E+12

1.2E+12

1E+12

8E+11

6E+11

4E+11

2E+11

0

Insertion Sort

Selection Sort

Merge Sort

Quick Sort

No. of Comparisions



No. of Comparisions

3.5E+10

3E+10

2.5E+10

2E+10

1.5E+10

1E+10

5E+09

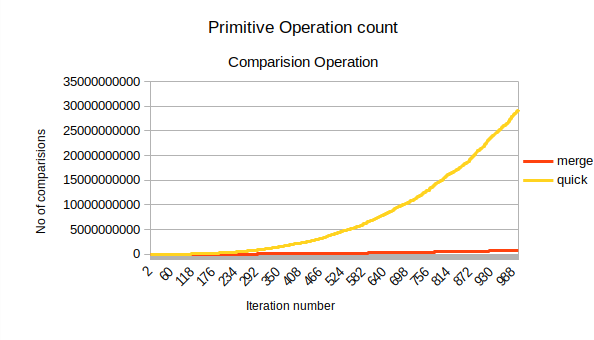
0

Insertion Sort

Merge Sort

Quick Sort

No. of Comparisions





## 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 ϴ*(n log n)* and average case time complexity of insertion sort is ϴ*(n2)* 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)

* + 1. Insertion Sort, whose best case is Ω(*n*) favors heavily here as a major chunk is already sorted
    2. Quick Sort, whose worst case is O(*n2*) 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))