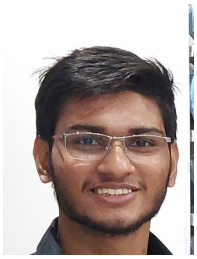
STUDENT PORTFOLIO

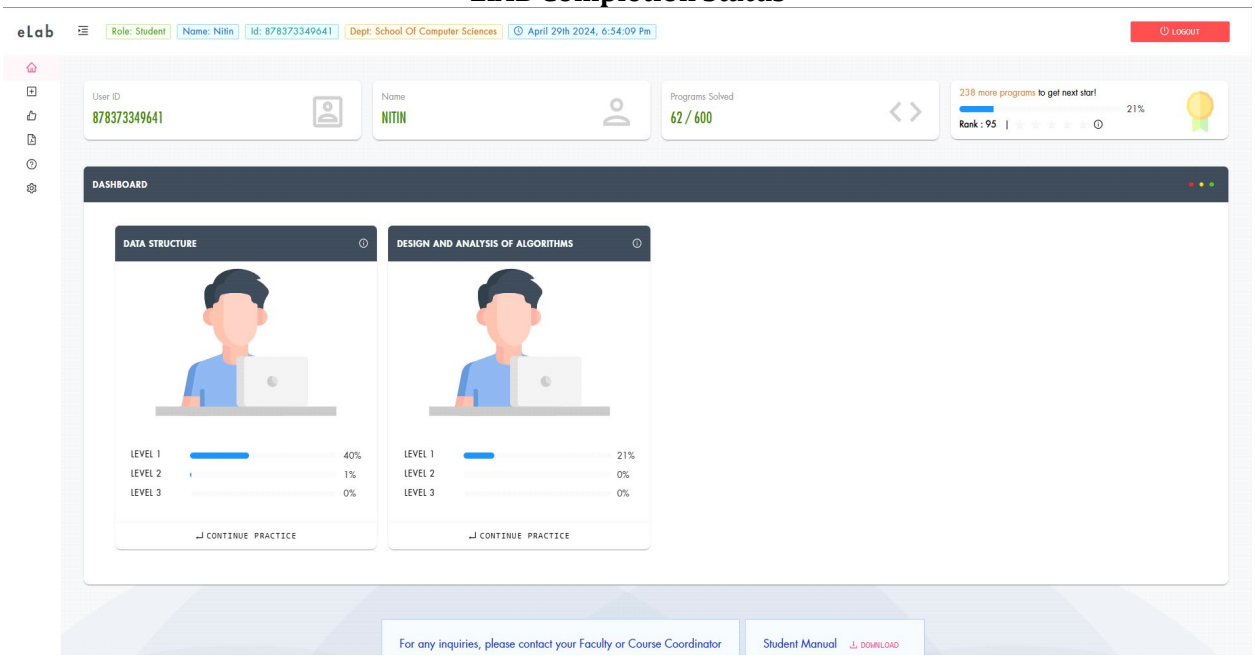


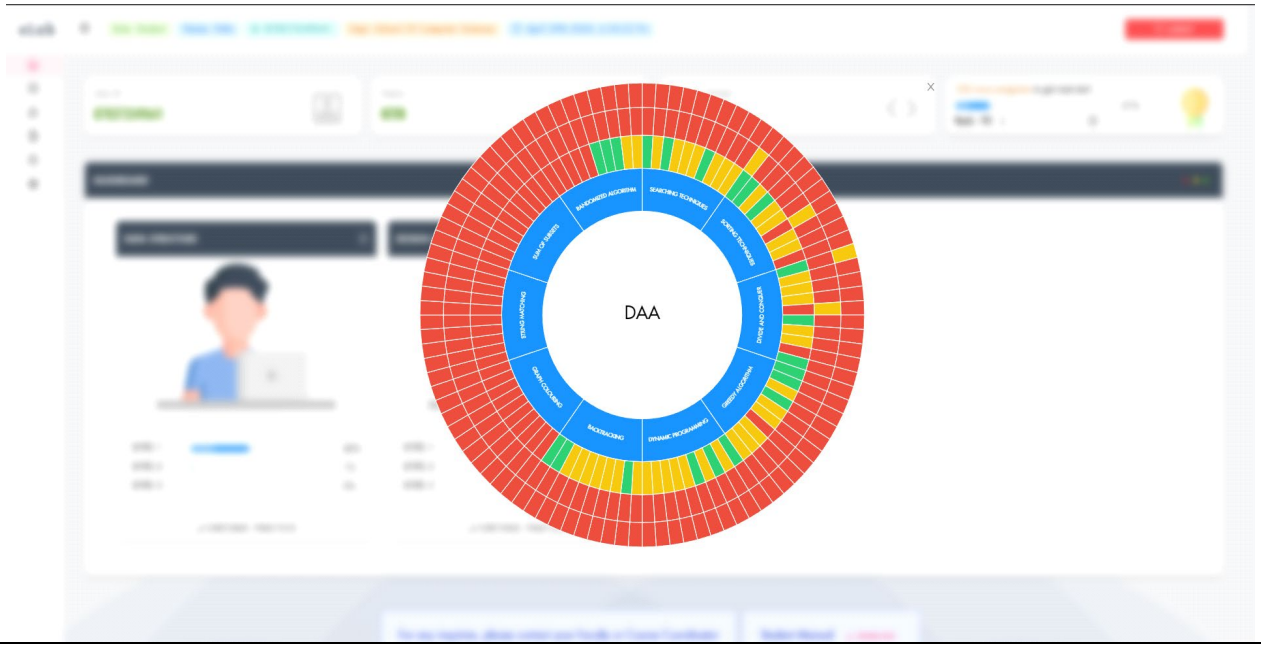
Name: NITIN KUMAR
Register Number: RA2211028010034
Mail ID: nk0865@srmist.edu.in
Department: COMPUTING TECHNOLOGIES
Semester: IV

Subject Title: 21CSC204J Design and Analysis of Algorithm

Handled By: Dr. G SUJATHA

ElAB Completion Status





Lab Experiment Completion status

SRM Institute of Science and Technology, Kattankulathur

(ACADEMIC YEAR AY 2023 - 24 EVEN)

21CSC204J - Design and Analysis of Algorithms

Observation Note Book Rubrics

Exp. No	Date	Title	Aim & Algorithm (1)	Program Implementation (10 Marks)					Time complexity analysis (3)	Dry run and Result (1)	Viva (5)	Total Marks (20)
				Basic Solution (2)	Modularity (2.5)	Readability (2.5)	Validation (2)	Scalability (1)				
1	30/01/24	1.a. Simple Algorithm-Insertion sort 1.b Bubble Sort	1	1	2.5	2.5	2	1	3	1	4	18
2	6/02/24	Linear search, Binary search	1	1	1.5	2.5	2	1	3	1	5	18
3	13/02/24	Merge sort,	1	2	1.5	1.5	2	1	3	1	5	18
4	20/02/24	Quick sort	1	1	2.5	2.5	2	1	3	1	4	18
5	27/02/24	Strassen Matrix multiplication	1	2	2.5	2.5	1	1	3	1	4	18
6	2/03/24	Finding Maximum and Minimum in an array, Convex Hull problem	1	2	2	2.5	2	1	3	1	4	18
7	5/03/24	7.a.Huffman coding 7.b.Knapsack using greedy	1	2	2.5	2.5	2	1	3	1	5	20
8	12/03/24	Longest common subsequence	1	2	2.5	2.5	2	1	3	1	5	20
9	19/03/24	N queen's problem	1	2	2.5	2.5	2	1	3	1	4	19
10	24/4/24	Travelling salesman problem	1	2	2.5	2.5	2	1	3	1	5	20
11	24/4/24	Randomized quick sort	1	2	2.5	2.5	2	1	3	1	5	20
12	24/4/24	String matching algorithms	1	2	2.5	2.5	2	1	3	1	4	19

Completed → 2/5/24

Music Streaming Platform: Playlist Creation and Song Sorting.

A comparative study of the available sorting techniques to identify the qptimised approach.

Table of contents

01

Problem Statement

For every music streaming platform, the ability of the platform to sort the songs based on various parameters is a primary feature. The users create personalized playlists based on their preferences and the application also needs to recommend songs to the user. This can be achieved by efficient sorting of songs on attributes like artist name, genre, or release date.

Introduction to Sorting.

- Sorting refers to arranging a set of data in a predefined logical order.
- The sorting algorithms need to be efficient for them to be usable in the real world problems.
- The complexity of a sorting algorithm measures the running time of a function in which n number of items are sorted.

Sorting algorithms are essential components of any music platform's functionality. They enable the efficient organization and retrieval of vast amounts of music data. In this presentation, we will explore various sorting techniques and their effectiveness for music platforms, with a focus on why Merge sort stands out as the most efficient choice.

02

Comparitive Study

There are various different sorting techniques available which can be utilised. Some are the brute force methods like Bubble Sort and Insertion Sort which are not ideal. Other techniques include Quick Sort and Merge Sort which are far more ideal due to the utilisation of Divide and Conquer approach.

Various Sorting Algorithms

Bubble Sort

Insertion Sort

Merge Sort

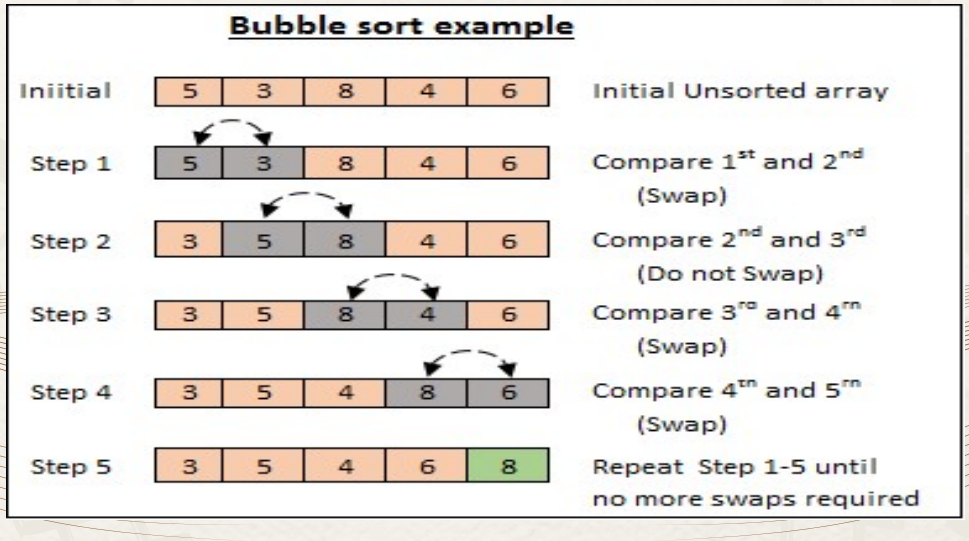
Quick Sort

Bubble Sort:

- Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjace elements, and swaps them if they are in the wrong order.
  In Bubble Sort algorithm,
  - Traverse from left and compare adjacent elements and the higher one is placed at right side.
  - In this way, the largest element is moved to the rightmost end at first.
  - This process is then continued to find the second largest and place it and so on until the data i sorted.

Complexity Analysis:

- Time Complexity: O(N2)
- The time complexity is conceptually defined O(N2) as it compares each element with every other element giving n comparisons. This process is repeated for n steps.
- This time complexity makes bubble sort inefficient for large datasets and a music application wil have millions of songs by thousands of artists.

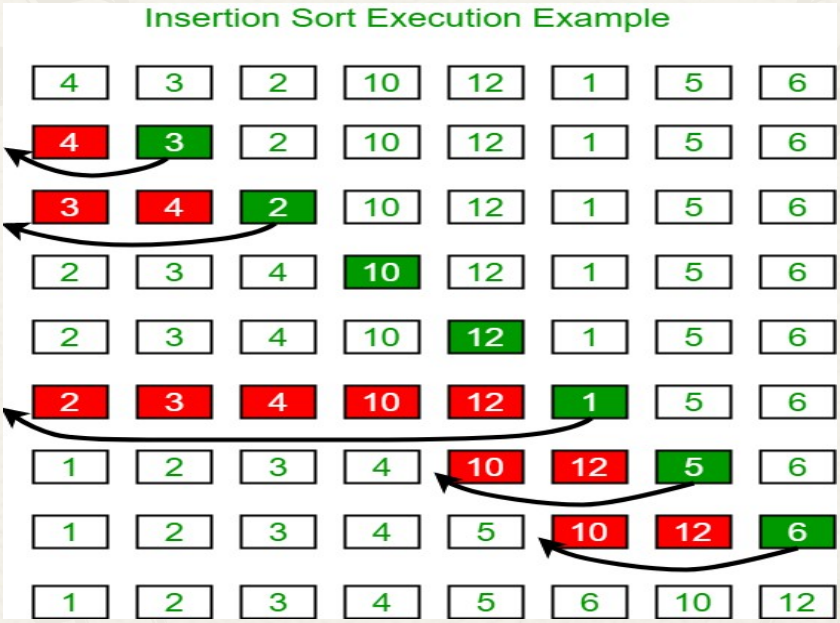


Insertion Sort:

- Insertion Sort builds the final sorted array one item at a time by iteratively removing elements from the input data and inserting them into their correct position.
- Insertion sort is a simple sorting algorithm that works similarly to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed in the correct position in the sorted part.

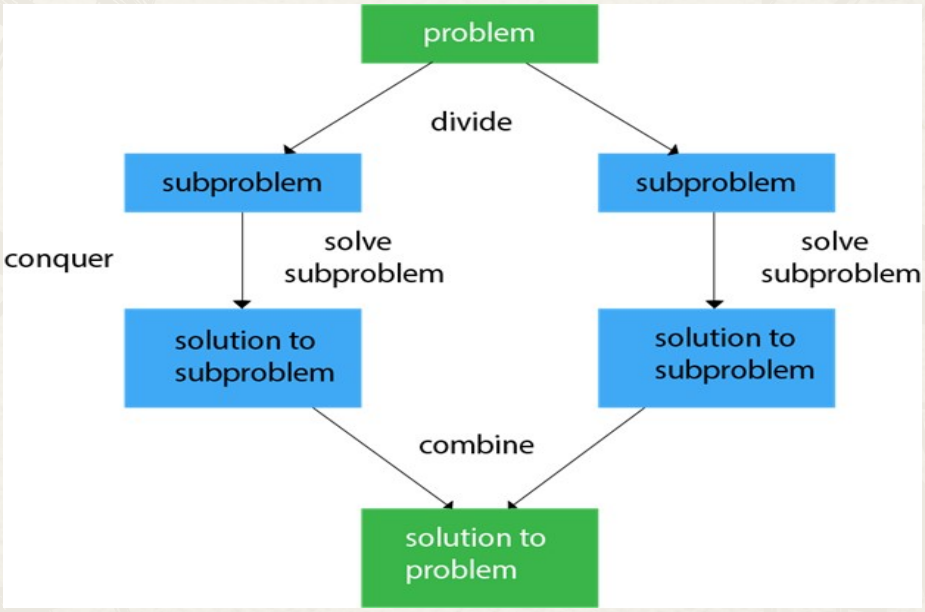
Complexity Analysis:

- Time Complexity: O(N2)
- The time complexity is conceptually defined O(N2) as it utilises two pointers and compares the inserted element with the previously sorted subarray. This process is repeated for n steps.
- While more efficient than Bubble Sort with a time complexity of O(n^2), Insertion Sort still struggle with large datasets due to its quadratic time



Divide and Conquer Approach:

- Divide and conquer, breaks a problem into sub problems that are similar to the original proble recursively solves the subproblems and finally combines the solutions of the sub problems to solve the original problem.
- Divide and conquer algorithm has three parts:
  - **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
  - **Conquer** the subproblems by solving them recursively. If, they are small enough, treat them as base cases.
  - **Combine** the solutions to the subproblems into the solution of the original problem.



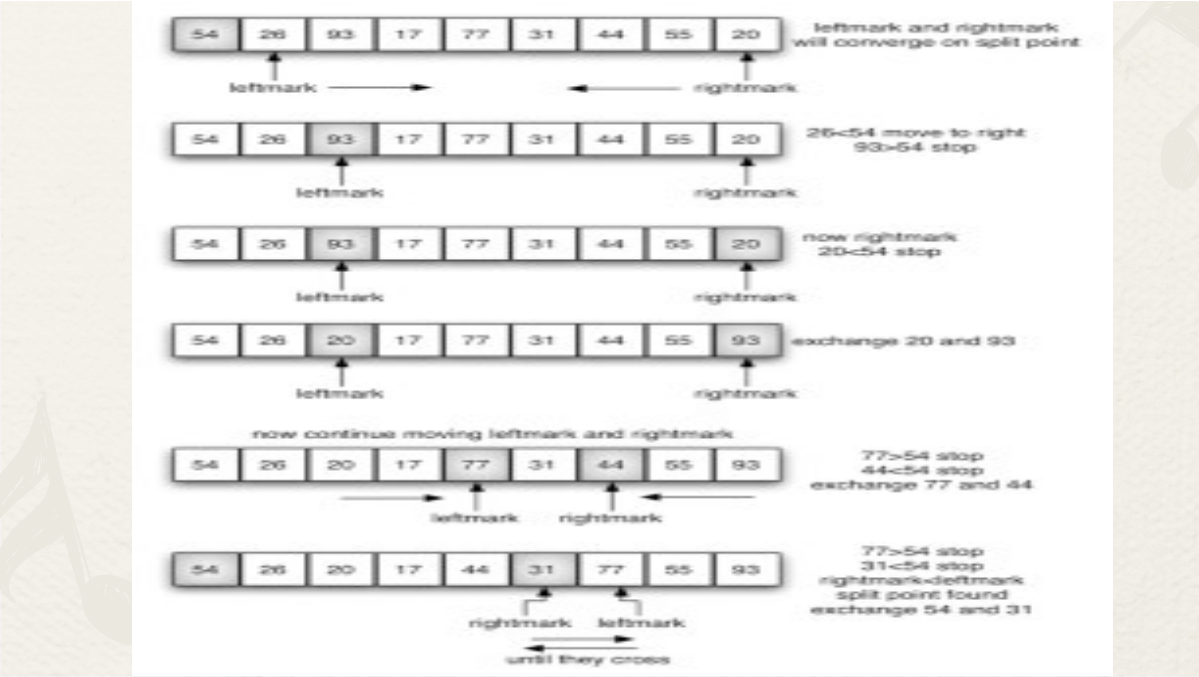
Quick Sort:

- Quick Sort employs a divide-and-conquer strategy to recursively partition the input array into smaller sub-arrays around a chosen pivot element.
- Quick Sort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.
- The key process in Quick Sort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.
- The choice of the pivot element is subject to the user or the algorithm. There can be a choice of pivot element from various options:
  - Always pick the first element as a pivot
  - Always pick the last element as a pivot (implemented below)
  - Pick a random element as a pivot
  - Pick the middle as the pivot.

- The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal) elements as i. While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise, we ignore the current element.
- As the partition process is done recursively, it keeps on putting the pivot in its actual position in the sorted array. Repeatedly putting pivots in their actual position makes the array sorted.

Complexity Analysis:

- Time Complexity:
  Best Case: Ω (N log (N)) and Worst Case: O(N2)
- The best-case scenario for quicksort occur when the pivot chosen at each step divides the array into roughly equal halves.
  In this case, the algorithm will make balanced partitions, leading to efficient Sorting.
- The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element.
- The reliance on randomization make it less predictable and potentially problematic for certain use ca

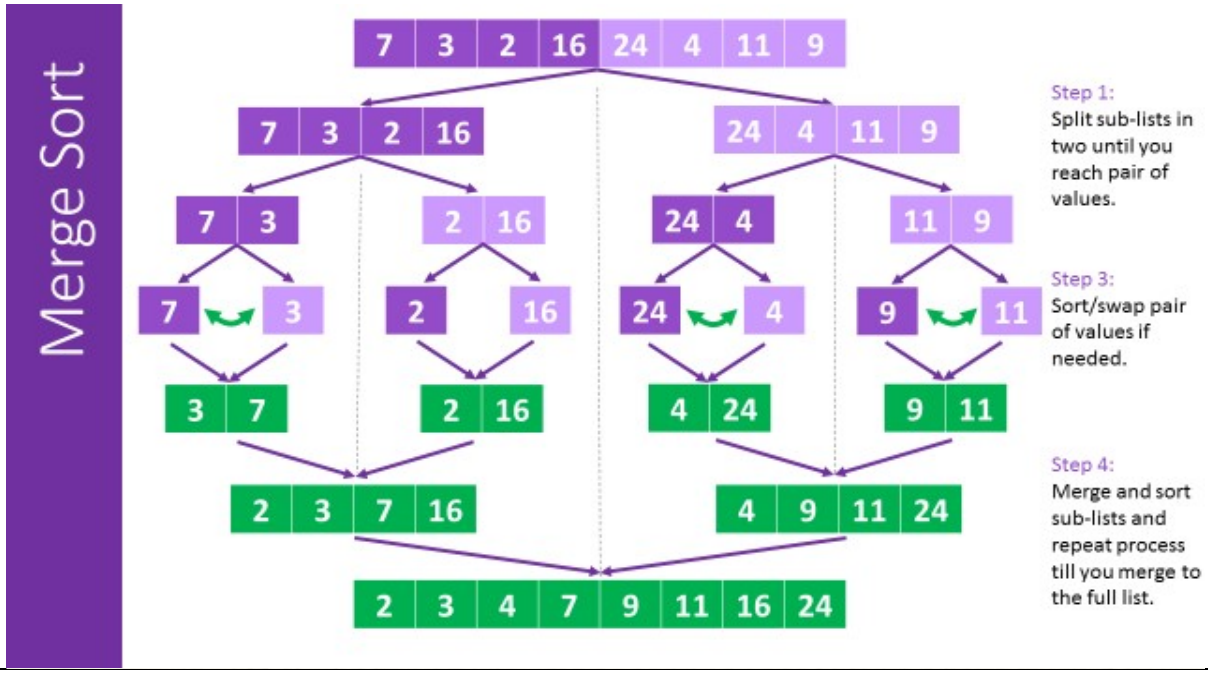


Merge Sort:

- Merge sort is a divide-and-conquer algorithm that divides the input array into smaller sub-arrays until each sub-array contains only one element. It then merges these arrays back together in sorted order.
- In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.
- Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

Complexity Analysis:

- Time Complexity: O(N log(N))
- Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation T(n) = 2T(n/2) + θ(n)
- As merge sort guarantees a time complexity of O(n log n) in all cases, making it highly efficient for sorting large datasets. Its stability and consistent performance make it an ideal choice for sorting music data efficiently.



03

Optimised Solution

As analysed in the previous slides, Merge Sorting algorithm is the most optimal solution for a music platform to perform sorting of songs on different parameters. In this section we take a look at how it can be achieved.

- The algorithm will be divided into two main functions: `merge_sort` and `merge`.
- `merge_sort`: This function will be responsible for dividing the input list into two halves, recursively calling the `merge_sort` function for each half, and then merging the two sorted halves using the merge function.
- `merge`: This function will take two sorted halves and merge them into a single sorted list.

Steps to sort the songs:

1. First, let's write a function to compare two songs based on a given attribute. This function, `compare_songs`, will take two songs and the chosen attribute as input and return a value based on the comparison of the attribute values.

```
def compare_songs(song1, song2, attribute):
    return getattr(song1, attribute) <= getattr(song2, attribute)
```

2. Now, let's implement the `merge` function. It will take three arguments: the two sorted halves `left` and `right`, and the chosen attribute A. The function will iterate through both halves, comparing the elements based on the attribute, and create a new sorted list by merging the elements in the correct order.

3. Finally, let's implement the `merge_sort` function. It will take the input list L and the chosen attribute A. If t length of the list is less than or equal to 1, the list is already sorted, and we return the list. Otherwis the list into two halves, recursively call the `merge_sort` function for each half, and then merge the two sorted halves using the merge function.

```
def merge_sort(L, attribute):
    if len(L) <= 1:
        return L

    mid = len(L) // 2
    left = merge_sort(L[:mid], attribute)
    right = merge_sort(L[mid:], attribute)

    return merge(left, right, attribute)
```

4. To demonstrate the sorting functionality, let's define a `Song` class with the necessary attributes, create a list songs, and call the `merge_sort` function with the desired sorting attribute.

- The `merge_sort` function is responsible for splitting the input list into smaller sublists and recursively calling itself to sort those sublists. Once the sublists are sorted, the merge function is used to merge these sorted sublists in the correct order based on the chosen attribute.
- The `compare_songs` function is used to compare two songs based on the given attribute, making the sor process more flexible and allowing users to sort their playlists based on different criteria.

Merge Function:

```
def merge(left, right, attribute):
    merged_list = []
    i = j = 0

    while i < len(left) and j < len(right):
        if compare_songs(left[i], right[j], attribute):
            merged_list.append(left[i])
            i += 1
        else:
            merged_list.append(right[j])
            j += 1

    while i < len(left):
        merged_list.append(left[i])
        i += 1

    while j < len(right):
        merged_list.append(right[j])
        j += 1

    return merged_list
```

```
class Song:
    def __init__(self, title, artist, album, release_date):
        self.title = title
        self.artist = artist
        self.album = album
        self.release_date = release_date

    def __repr__(self):
        return f"{self.title} by {self.artist} ({self.album}, {self.release_date})"

song1 = Song("Song 1", "Artist 2", "Album 3", 2019)
song2 = Song("Song 3", "Artist 1", "Album 1", 2021)
song3 = Song("Song 2", "Artist 3", "Album 2", 2020)

playlist = [song1, song2, song3]
sorted_playlist = merge_sort(playlist, "title")
```

Implementation Example:

- Imagine you have a playlist of unsorted songs:
- Song 1: "Bohemian Rhapsody" by Queen
- Song 2: "Imagine" by John Lennon
- Song 3: "Hotel California" by Eagles
- Song 4: "Hallelujah" by Leonard Cohen
- Song 5: "Stairway to Heaven" by Led Zeppelin

1. Divide:

- Split the playlist into smaller sublists, each containing just one song initially.

[Queen - Bohemian Rhapsody]
[John Lennon - Imagine]
[Eagles - Hotel California]
[Leonard Cohen - Hallelujah]
[Led Zeppelin - Stairway to Heaven]

2. Conquer and Combine:

Recursively call the merge_sort function on each sublist (individual songs in this case) until they are sorted by artist name (one-element lists are already sorted).

- Combine: Merge the sorted sublists back together in the correct order based on artist name.
- Merge the first two sublists:

[John Lennon - Imagine, Queen - Bohemian Rhapsody]
[Eagles - Hotel California]
[Leonard Cohen - Hallelujah]
[Led Zeppelin - Stairway to Heaven]

- Merge the next two sublists:

[John Lennon - Imagine, Queen - Bohemian Rhapsody]
[Eagles - Hotel California, Leonard Cohen - Hallelujah]
[Led Zeppelin - Stairway to Heaven]

- Merge the remaining two sublists:

[John Lennon - Imagine, Queen - Bohemian Rhapsody, Eagles - Hotel California, Leonard Cohen - Hallelujah]
[Led Zeppelin - Stairway to Heaven]

- Finally, merge the last two merged sublists:

[John Lennon - Imagine, Queen - Bohemian Rhapsody, Eagles - Hotel California, Leonard Cohen - Hallelujah, Led Zeppelin - Stairway to Heaven]

The final playlist is now sorted by artist name:
1. John Lennon - Imagine
2. Queen - Bohemian Rhapsody
3. Eagles - Hotel California
4. Leonard Cohen - Hallelujah
5. Led Zeppelin - Stairway to Heaven

Therefore, merge sort algorithm is implemented to sort the songs depending on the attribute- "Artist Name". As visible, the names are in alphabetical order and in large datasets, this makes selection of songs efficient.

04 Conclusion

In conclusion, the choice of sorting algorithm significantly impacts the performance of a music platform, particularly when handling large datasets. While several sorting techniques offer varying degrees of efficiency, Mergesort emerges as the most reliable and scalable option. Its consistent O(n log n) time complexity ensures optimal performance, making it the preferred choice for sorting music data efficiently and effectively.

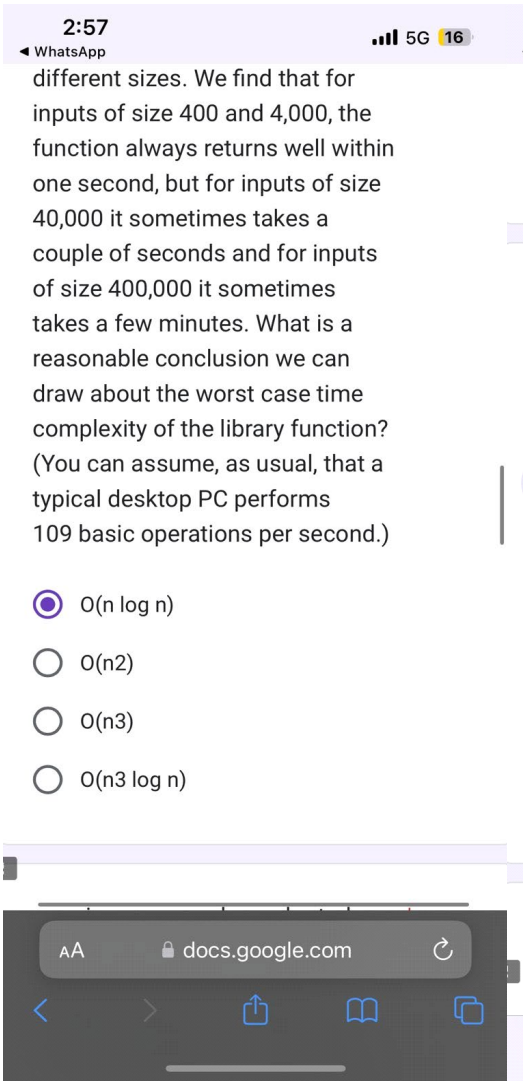
- While other sorting algorithms may work for smaller playlists, Merge Sort shines for large music libraries on a music platform.
- Its divide-and-conquer approach breaks down the playlist into manageable chunks, efficiently sorting them independently. This makes Merge Sort especially scalable, handling massive datasets without significant performance degradation.
- Unlike some sorting methods that require multiple passes through the data, Merge Sort is guaranteed to sort the songs in a single pass after the initial partitioning.

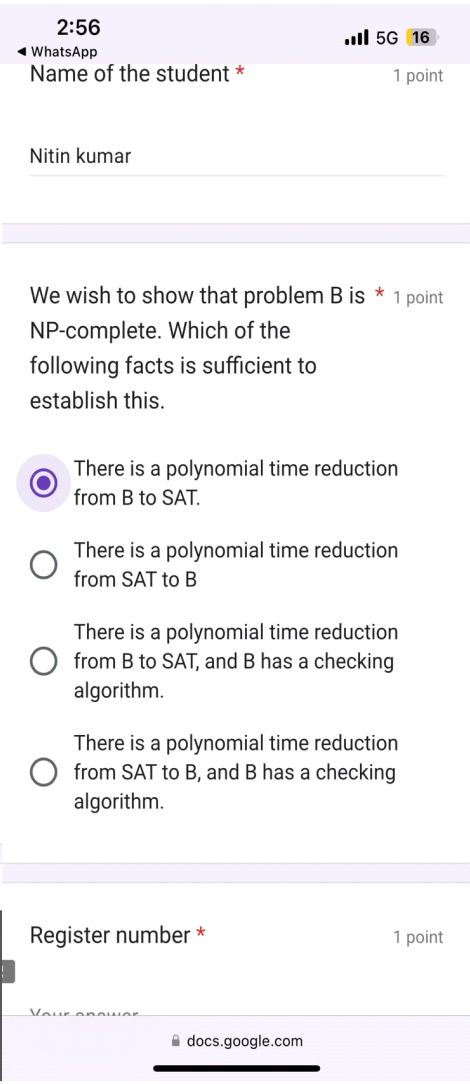
Thanks

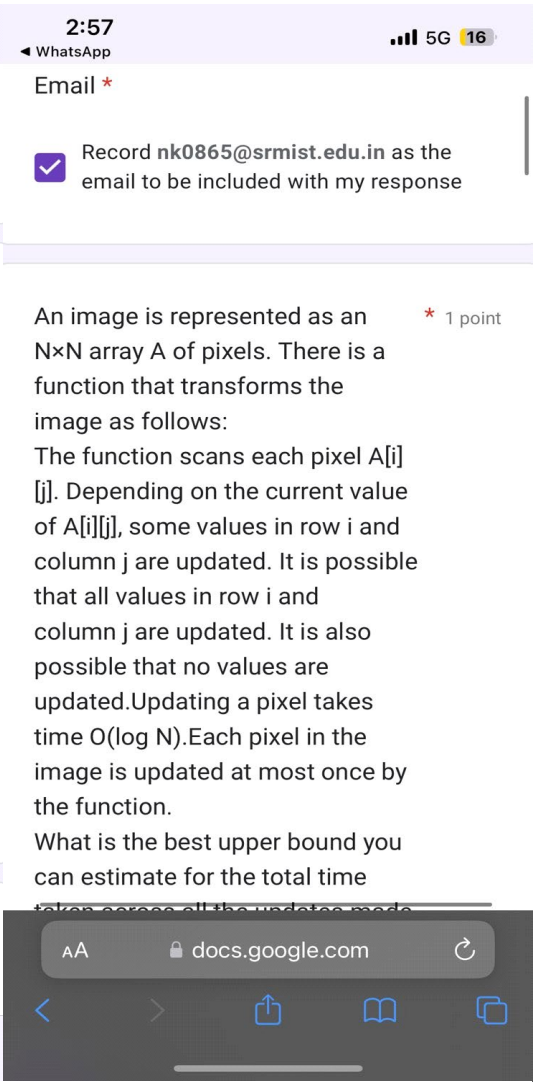
Presented By:
Nitin Kumar (RA2211028010034)
Chirag Jain (RA2211028010047)
Akshit Labh (RA2211028010049)

Any other
(Write if you registered or practice apart from Hacker rank, Leetcode, GitHub
E.g.: Certification Programs related to DAA)

NPTEL/HOTS Questions Solution.







Signature

Note: Enclose the assignment and relevant certificates along with the profile