Koneru Lakshmaiah Education Foundation
(Deemed to be University estd. u/s. 3 of the UGC Act, 1956)
Off-Campus: Bachupally-Gandimaisamma Road, Bowrampet, Hyderabad, Telangana - 500 043.
Phone No: 7815926816, www.klh.edu.in

Department of Computer Science and Engineering

2024-2025
Even Semester

DESIGN AND ANALYSIS ALGORITHMS (24CS2203)

ALM – PROJECT BASED LEARNING

QuickSort in Online Gaming Leaderboards

P SHASHINDRA REDDY **2420030360**
M YESESHWINI **2420090080**

COURSE INSTRUCTOR

Dr. J Sirisha Devi
Professor
Department of Computer Science and Engineering

TITLE OF THE PBL

Introduction:

In today's fast-paced gaming world, millions of players compete simultaneously in online games such as BGMI, Valorant, Free Fire, and Chess.com. These games maintain real-time leaderboards that continuously update player ranks after every match or score change. Efficient ranking systems are crucial to ensure that players see their positions updated instantly and accurately.

Sorting algorithms play a fundamental role in this process. Each time a player's score changes, the system needs to reorder the leaderboard based on scores in descending order. This must happen quickly to maintain a smooth and fair gaming experience for all users.

Among all sorting algorithms, QuickSort provides the best balance of speed, efficiency, and low memory usage for large, dynamic datasets. It uses the divide and conquer approach to partition the dataset around a selected pivot and recursively sort the partitions. This makes it highly suitable for real-time gaming environments where quick updates and scalability are essential.

Problem Statement:

The challenge in an online gaming leaderboard is to **sort millions of player scores efficiently** whenever updates occur.
If sorting is slow or inefficient, players might experience:

- Delayed rank updates
- Incorrect leaderboard positions
- Poor overall gaming experience

Therefore, the problem can be summarized as:

"Design a sorting mechanism that can handle frequent score updates efficiently, maintaining real-time accuracy and scalability for online gaming leaderboards."

QuickSort is chosen for this case study because of its **average-case performance of O(n log n)** and **in-place sorting mechanism**, which makes it ideal for systems where both **speed** and **memory** are critical.

How QuickSort Fits the Use Case:

When a player finishes a match, their score changes and the leaderboard must reflect it immediately. QuickSort divides the list of players into subgroups based on a **pivot score**:

- Players with **higher or equal scores** than the pivot go to the **left partition**.
- Players with **lower scores** go to the **right partition**.

These subgroups are then recursively sorted until the entire leaderboard is ordered correctly.

In practice:

- **Pivot selection** can be randomized to avoid worst-case performance.
- The sorting can occur **in-memory** for active leaderboards and **periodically persisted** to a database.

For example, consider a simplified leaderboard:

Player	Score
P1	500
P2	700
P3	300
P4	900
P5	600

After applying QuickSort (descending order), the sorted leaderboard will be:

Rank	Player	Score
1	P4	900
2	P2	700
3	P5	600
4	P1	500
5	P3	300

This ensures that leaderboard updates are instantaneous and fair, providing a smooth gaming experience.

ALGORITHM / PSEUDO CODE

Algorithm: QuickSortLeaderboard(arr, low, high):

```
QuickSortLeaderboard(arr, low, high):
   # arr is an array of (player_id, score) pairs
   if low < high:
      pivot_index = PartitionDesc(arr, low, high)
      QuickSortLeaderboard(arr, low, pivot_index - 1)
      QuickSortLeaderboard(arr, pivot_index + 1, high)
```

Partition Function: PartitionDesc(arr, low, high):

```
PartitionDesc(arr, low, high):
   # Choose pivot as arr[high] (can be randomized)
   pivot_score = arr[high].score
   i = low - 1
   for j = low to high - 1:
      if arr[j].score >= pivot_score:      # >= for descending order
         i = i + 1
         swap(arr[i], arr[j])
   swap(arr[i + 1], arr[high])
   return i + 1
```

Step-by-Step Explanation:

1. Pivot Selection:
   The last element of the array (or a random one) is chosen as the pivot.
   Example: pivot = player with score 700.
2. Partitioning Process:
   - All elements with a score greater than or equal to the pivot move to the left.
   - All elements with a score less than the pivot move to the right.
3. Recursive Sorting:
   - The process repeats for both left and right partitions until the array is fully sorted.
4. In-place Sorting:
   - QuickSort does not require additional arrays. It swaps elements directly in the existing array.

Example Execution

Input:
[(P1,500), (P2,700), (P3,300), (P4,900), (P5,600)]

Step 1: Choose pivot (e.g., 600).

Step 2: Partition around 600 → left: [700, 900], right: [500, 300].

Step 3: Recursively sort left and right.

Step 4: Final output: [900, 700, 600, 500, 300].

TIME COMPLEXITY

Function: QuickSortLeaderboard(arr, low, high):

Line	Statement	Time	Explanation
1	if low < high:	O(1)	Constant comparison
2	pivot_index = PartitionDesc(arr, low, high)	O(n)	Partition scans all elements once
3	QuickSortLeaderboard(arr, low, pivot_index - 1)	T_1	Recursive call for left partition
4	QuickSortLeaderboard(arr, pivot_index + 1, high)	T_2	Recursive call for right partition

Total Recurrence Relation:

$T(n) = T_1 + T_2 + O(n)$
If pivot splits evenly → $T(n) = 2T(n/2) + O(n)$

Case	Recurrence	Time Complexity	Explanation
Best Case	Balanced splits	O(n log n)	Equal halves each time
Average Case	Random pivot	O(n log n)	Most common in real leaderboards
Worst Case	Sorted data (bad pivot)	$O(n^2)$	All elements go to one side

Example:
For 8 players → ~24 comparisons (≈ n log n)

Final Result:

Time Complexity:

- **Best: O(n log n)**
- **Average: O(n log n)**
- **Worst: $O(n^2)$**

SPACE COMPLEXITY

Function: QuickSortLeaderboard(arr, low, high):

Line	Statement	Space	Explanation
1	if low < high:	O(1)	Constant space for comparison
2	pivot_index = PartitionDesc(arr, low, high)	O(1)	In-place partition uses same array
3	QuickSortLeaderboard(arr, low, pivot_index - 1)	Stack +1	Recursive call adds 1 stack frame
4	QuickSortLeaderboard(arr, pivot_index + 1, high)	Stack +1	Another recursive branch

Recursion Depth:

- Balanced splits: $\log_2(n) \rightarrow O(\log n)$
- Worst-case splits: $n \rightarrow O(n)$

◈ PartitionDesc Function:

Line	Statement	Space	Explanation
1–7	Variable assignments, swaps	O(1)	All operations done in-place, no new arrays