Design and Analysis of Algorithms (24CS2203)

Quick Sort in Online Gaming Leaderboards

A . Sahithya :2420030369

Tejaswi Chowdary :2420030517

Course Instructor

Dr. J Sirisha Devi

Professor

Department of Computer Science and Engineering

Case study - statement

- **Problem:** The core challenge is the need for **fast, real-time sorting and ranking** of millions of player scores in online gaming leaderboards, a list that is constantly updated and queried.
- **Goal:** Implement and analyze the **Quick Sort algorithm** to efficiently manage this dynamic dataset, minimizing latency for rank updates and retrieval.
- **Scale:** Leaderboards can involve N users, where N is very large (e.g., 106 to 108).

Algorithm /Pseudo code

- **Title: Quick Sort Algorithm: Divide and Conquer**
- **Key Concept:** Quick Sort is a recursive, in-place sorting algorithm based on the **Divide and Conquer** paradigm.
- **Steps:**

1.Divide (Partition): Select a **Pivot** element (e.g., Median-of-Three). Rearrange the array so elements ≤ Pivot are on one side and elements ≥ Pivot are on the other. The pivot is placed at its final sorted position.

2.Conquer (Recurse): Recursively call Quick Sort on the sub-array to the left and the sub-array to the right of the pivot.

3.Combine: The sorting is done in-place during the partitioning phase, so the combine step is trivial.

Algorithm /Pseudo code

- Present the two main functions: QUICK_SORT (the recursive wrapper) and PARTITION (the workhorse).

- **QUICK_SORT(A, low, high):**

Code snippet

```
IF low < high THEN
 pi = PARTITION(A, low, high)
 QUICK_SORT(A, low, pi - 1)
 QUICK_SORT(A, pi + 1, high)
 END IF
```

- **PARTITION(A, low, high):** Briefly explain the logic—selecting the pivot and using an index to track the boundary between smaller and larger elements. (Use a simplified pseudocode or flow diagram here to maximize slide space).

Time Complexity

- **Title: Time Complexity: The Speed of Ranking**
- **Variable N**: Number of players in the leaderboard.
- **Best Case: O(NlogN)**
- **When:** Pivot always splits the array into two equal halves.
- **Recurrence:** T(N)=2T(N/2)+O(N).
- **Relevance:** Achieved with good pivot selection (e.g., Median-of-Three).
- **Average Case: O(NlogN)**
- **When:** The most likely scenario with random data distribution.
- **Why Quick Sort is preferred:** It has a smaller hidden constant factor than Merge Sort, making it faster in practice for leaderboards.
- **Worst Case: O(N2)**
- **When:** Pivot always selects the smallest/largest element (e.g., already sorted data with fixed pivot).
- **Mitigation:** The worst case is almost entirely avoided in real-world applications by using a randomized pivot selection.

Space Complexity

- Auxiliary space is primarily due to the **Recursion Call Stack**.
- **Analysis:**
- **Best/Average Case: O(logN)**
  - **Reason:** The array is divided roughly in half at each step, limiting the depth of recursion to logN.
- **Worst Case: O(N)**
  - **Reason:** Unbalanced partitioning leads to a recursion depth of N.
  - **Mitigation:** Can be reduced to O(logN) even in the worst case by implementing **tail recursion optimization** (always recursing on the smaller sub-array first).
- **Conclusion:** Quick Sort is an **in-place** sort, making it highly memory efficient compared to O(N) auxiliary space algorithms like Merge Sort.

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

Quick Sort in Online Gaming Leaderboards

A.sahithya **2420030369**

Tejaswi Chowdary **2420030517**

COURSE INSTRUCTOR

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

Quick Sort in Online Gaming Leaderboards

In online gaming platforms, maintaining accurate and real-time leaderboards is essential for ensuring fair competition and enhancing the gaming experience. Leaderboards are used to display player rankings based on performance metrics such as scores, win rates, or levels achieved. As millions of players participate simultaneously, sorting and updating these scores efficiently becomes a computational challenge.

To address this, sorting algorithms play a crucial role, and among them, Quick Sort is one of the most efficient comparison-based sorting algorithms. Quick Sort works by selecting a pivot element, partitioning the array into two subarrays based on the pivot, and recursively sorting the subarrays. This makes it ideal for dynamically updating leaderboards in online gaming environments.

In a typical gaming scenario, when a player completes a game session, their score is added to the database. The leaderboard must then be updated to reflect their new position without introducing noticeable lag. Quick Sort, due to its average-case time complexity of $O(n \log n)$ and in-place sorting nature, allows for fast and memory-efficient ranking updates.

For instance, consider a multiplayer online battle game where players earn points after each round. The server maintains a list of player scores. When a player's score changes, Quick Sort can quickly re-sort the list to display the updated ranks on the leaderboard. Its partitioning mechanism ensures that sorting large datasets remains fast and stable even under heavy player load.

This case study demonstrates how Quick Sort can be implemented to maintain real-time accuracy and responsiveness in online leaderboards, contributing to improved user satisfaction and system performance.

ALGORITHM / PSEUDO CODE

Quick Sort Algorithm

Step 1: Start
Step 2: Choose a pivot element from the array (commonly the last or middle element).
Step 3: Partition the array such that:

- Elements less than the pivot are placed on the left side.
- Elements greater than the pivot are placed on the right side.
  Step 4: Recursively apply Quick Sort to the left and right subarrays.
  Step 5: Combine the sorted subarrays and the pivot to form the final sorted array.
  Step 6: Stop.
- Pseudo code

```
QuickSort(arr, low, high)
  if low < high then
    pivotIndex = Partition(arr, low, high)
    QuickSort(arr, low, pivotIndex - 1)
    QuickSort(arr, pivotIndex + 1, high)
  end if
end procedure


Partition(arr, low, high)
  pivot = arr[high]
  i = low - 1
  for j = low to high - 1 do
    if arr[j] <= pivot then
      i = i + 1
      swap arr[i] and arr[j]
    end if
```

```
    end for
    swap arr[i + 1] and arr[high]
    return i + 1
end procedure
```

Example

Consider an array of player scores:
[2400, 1800, 3200, 2100, 2700]

Step 1: Choose pivot = 2700
Partition array: [2400, 1800, 2100, 2700, 3200]
Step 2: Recursively sort [2400, 1800, 2100] and [3200]
Step 3: Final sorted leaderboard: [1800, 2100, 2400, 2700, 3200]

This sorted array represents the leaderboard from lowest to highest score, which can be easily reversed for descending order display.

TIME COMPLEXITY

Time Complexity

Case	Description	Time Complexity
Best Case	When the pivot divides the array into two equal halves	O(n log n)
Average Case	Typical performance with random data	O(n log n)
Worst Case	When the pivot is the smallest or largest element (already sorted array)	$O(n^2)$

Explanation:
In the best and average cases, each partition roughly divides the array into two subarrays, and the number of comparisons per level is proportional to *n*. Since there are *log n* levels of partitioning, the total comparisons are *O(n log n)*.
However, if the pivot repeatedly partitions the array unevenly (e.g., always selecting the smallest element as the pivot), the recursion depth increases to *n*, leading to $O(n^2)$ time complexity.

SPACE COMPLEXITY

Space Complexity

Parameter	Description	Complexity
Auxiliary Space	Stack space used for recursive calls	O(log n)
In-place Operation	Sorting is performed within the same array	O(1)

Explanation:
Quick Sort is an in-place algorithm because it rearranges elements within the same array without requiring additional memory for another data structure.
However, recursive calls consume stack space proportional to the depth of recursion, which is *O(log n)* for balanced partitions and *O(n)* in the worst case.

Conclusion

Quick Sort offers a powerful and efficient approach for maintaining dynamic leaderboards in online gaming environments. Its ability to handle large data efficiently, combined with its low memory usage, makes it well-suited for real-time applications. By integrating Quick Sort into leaderboard management systems, gaming platforms can ensure faster ranking updates, reduced server load, and an enhanced competitive experience for players.