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 Hospital Patient Prioritization

Ch.Sri Ram Sanjeev	2420090051
Dhanush	2420090083

COURSE INSTRUCTOR

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

Quick Sort in Hospital Patient Prioritization

Brief description of the case study with examples. Minimum 2-3 pages

1.1 The Challenge: Managing Patient Influx in Emergencies

In a hospital Emergency Department (ED), the patient inflow is unpredictable and frequently overwhelming. Patients arrive continuously with a broad spectrum of conditions — from minor injuries to critical, life-threatening emergencies. A traditional *first-come, first-served* system fails to account for severity, often resulting in dangerous delays for patients who require immediate attention.

In high-pressure environments like emergency care units, doctors, and triage staff must make rapid and accurate decisions with limited time and resources. The lack of an efficient prioritization mechanism can lead to inefficiencies, extended waiting times, and compromised patient safety. Manual sorting or reassessment of patient urgency during busy hours is not only impractical but also highly error-prone.

The main challenge, therefore, is to design an algorithmic solution that can dynamically organize the patient queue in real time — ensuring that the most critical patients receive care first while optimizing the hospital's limited resources.

1.2 The Solution: Triage and Algorithmic Prioritization

The healthcare domain already uses a structured prioritization process known as **triage**, which assigns patients an **urgency score** based on their condition. A common model is the **Emergency Severity Index (ESI)** — a five-level system where lower values represent higher medical urgency:

ESI Level	Description	Example
1	Resuscitation – Immediate life-saving intervention	Cardiac arrest

ESI Level	Description	Example
2	Emergent – Rapid intervention needed	Stroke symptoms
3	Urgent – Serious but not life-threatening	Severe abdominal pain
4	Less Urgent – One resource required	Simple fracture
5	Non-Urgent – Minor issue	Suture removal

As each patient is triaged and assigned an ESI score, their details are entered into a digital queue. To ensure continuous prioritization, this queue must be dynamically **sorted based on urgency**. Here, the **Quick Sort algorithm** becomes an ideal choice. It efficiently reorders the patient list by comparing urgency scores and guarantees that the patient requiring the most immediate care (lowest ESI score) always remains at the top.

This computational method transforms a complex human task into a rapid, automated process, enabling real-time decision-making during emergencies.

1.3 Core Principle of Quick Sort

Quick Sort is a **divide-and-conquer** algorithm designed to handle sorting problems with high efficiency. It operates in three major steps:

1. **Pivot Selection:**
   Choose one element (the pivot) from the list. The pivot acts as a reference point for sorting the rest of the data.
2. **Partitioning:**
   Rearrange elements so that those smaller (or higher priority) than the pivot appear before it, and those larger (lower priority) appear after it.
3. **Recursion:**
   Recursively apply the same steps to the sublists created on either side of the pivot.

This process continues until every sublist contains only one element — at which point the entire list is sorted. Quick Sort's in-place sorting capability means it requires very little additional memory, making it optimal for critical systems like hospital dashboards.

1.4 Application in an ED Triage System

In a hospital’s emergency software system, Quick Sort can maintain an always-sorted list of patients according to urgency level. Each event updates the list instantly:

- **Event 1: Patient Arrival**
  When a new patient is triaged, their urgency score is entered and Quick Sort runs to insert them into the correct position.
- **Event 2: Doctor Becomes Available**
  The system simply assigns the topmost patient in the sorted queue (highest priority) to the doctor.
- **Event 3: Re-Triage (Condition Worsens)**
  If a patient’s condition worsens, their score is updated and Quick Sort is invoked again, repositioning them higher on the list.

This approach minimizes manual intervention, ensures fairness, and guarantees that critical cases are never overlooked.

PSEUDO CODE

Step-by-step representation of the algorithm/pseudo code in detail. Minimum 2-3 pages

Pseudocode :

```
 // Quick Sort for patient prioritization
QuickSort(arr, low, high)
{
    if (low < high)
    {
        pi = Partition(arr, low, high);
        QuickSort(arr, low, pi - 1);   // Sort higher urgency patients
        QuickSort(arr, pi + 1, high);  // Sort lower urgency patients
    }
}

Partition(arr, low, high)
{
    pivot = arr[high];
    i = (low - 1);

    for (j = low; j <= high - 1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            swap(arr[i], arr[j]);
        }
    }

    swap(arr[i + 1], arr[high]);
    return (i + 1);
}
```

3. Algorithm Trace

Input List (Urgency Scores): [3, 5, 1, 4, 2]

Output (Sorted by Urgency): [1, 2, 3, 4, 5]

This trace demonstrates how patients with lower ESI values (more critical) are brought to the front. Each recursive call progressively sorts smaller sections until the entire list is ordered by priority.

TIME COMPLEXITY

Time complexity calculation with detailed explanation.

Let **T(n)** be the time (number of basic operations, e.g., comparisons + swaps) taken to Quick Sort an array of size **n**.
The **partition** step scans the subarray once, so its cost is **$\Theta(n)$** for that call (we'll write it as **$c \cdot n$** for some constant $c > 0$).

When you partition around a pivot, you split the array into two subarrays of sizes **k** and **$n - 1 - k$** (because the pivot occupies one position).
So, the recurrence is:

$$T(n) = T(k) + T(n - 1 - k) + c \cdot n$$

The different cases come from how large **k** is.

---

1) Best Case

Assumption: The pivot always splits the array exactly in half (or nearly half).
So, **$k \approx n/2$**.
The recurrence becomes:

$$T(n) = 2T(n/2) + c \cdot n$$

This matches the standard **Master Theorem** form:

$T(n) = aT(n/b) + f(n)$
where **$a = 2$**, **$b = 2$**, and **$f(n) = c \cdot n$**

Now, compare **f(n)** with **$n^{\log_b a} = n^{\log_2 2} = n$**

Since **$f(n) = \Theta(n^{\log_b a})$**, this falls under **Case 2 of the Master Theorem**, so the solution is:

$T(n) = \Theta(n \log n)$

Intuitive "Level-by-Level Work" View

- The top level does **$\Theta(n)$** work (one full partition).
- Each of the two subproblems at the next level (each size ≈ n/2) together process every element once again — another **$\Theta(n)$** total.
- This continues for **$\log_2 n$** levels (because dividing by 2 repeatedly reaches size 1 after $\log_2 n$ steps).

So, total work ≈ **$\Theta(n) \times \log n = \Theta(n \log n)$**

---

2) Worst Case

Assumption: The pivot always gives the most unbalanced split — one side has size **$n - 1$**, the other side has size **0**.
This happens, for example, when the array is already sorted and you choose the last element as the pivot.

The recurrence becomes:

$T(n) = T(n - 1) + c \cdot n$

(plus maybe an O(1) for selecting the pivot)

Now, unfold the recurrence step-by-step:

$$\begin{aligned}
T(n) &= T(n-1) + c\cdot n \\
&= [T(n-2) + c\cdot(n-1)] + c\cdot n \\
&= T(n-2) + c\cdot(n-1) + c\cdot n \\
&= T(n-3) + c\cdot(n-2) + c\cdot(n-1) + c\cdot n \\
&\ldots \\
&= T(1) + c\cdot(2 + 3 + \ldots + n)
\end{aligned}$$

Sum of first n integers = $n(n + 1)/2$, so:

$$T(n) = \Theta(n(n + 1)/2) = \Theta(n^2)$$

Intuitive Explanation

At each stage, the algorithm reduces the problem size by 1 but still performs about **$\Theta(n)$** comparisons per stage.
So total work = **$n + (n-1) + (n-2) + ... + 1 = \Theta(n^2)$**

Thus, the **worst-case time complexity = $\Theta(n^2)$**

---

3) Average Case (Random Pivot)

Now assume the pivot is chosen randomly and is equally likely to be any element (so splits are uniform on average).

Let **$E[T(n)]$** denote the *expected* running time for input size **n**.

Then:

$$E[T(n)] = (1/n) \Sigma [E[T(k)] + E[T(n - 1 - k)]] + c \cdot n,$$
where the sum is from $k = 0$ to $n - 1$.

Since the distribution of **k** is uniform from 0 to $n - 1$,
we can simplify the summation term:

$$E[T(n)] = (2/n) \Sigma E[T(k)] + c \cdot n \qquad \text{(Equation *)}$$

This is a standard recurrence for average-case Quick Sort.
Solving it using known techniques or induction (and knowing $E[T(0)] = E[T(1)] = \Theta(1)$) gives:

$$E[T(n)] = \Theta(n \log n)$$

Expected Number of Comparisons (Concrete Formula)

A stronger known result states that:

$E[\text{Comparisons}] = 2(n + 1)H_n - 4n \approx 2n \ln n + O(n)$

where **$H_n = 1 + 1/2 + 1/3 + \ldots + 1/n$** is the *n-th Harmonic Number,* and **$H_n \approx \ln n + \gamma$** ($\gamma$ = Euler's constant ≈ 0.577).

This still simplifies to **$\Theta(n \log n)$** (the difference between log base 2 and natural log is just a constant factor).

Intuition:
On average, each pair of elements is compared only a constant number of times across the sorting process.
Summing up all probabilities gives the harmonic sum, which grows as log n.

SPACE COMPLEXITY

Space complexity calculation with detailed explanation.

Let S(n)S(n)S(n) denote the **auxiliary space** used by Quick Sort to sort an array of size nnn.
Quick Sort uses extra space primarily for:

1. **Recursive function calls (call stack)**
2. **Temporary variables for partitioning** (pivot, indices, swaps)

Step 1: Space for Partitioning

- The partition function rearranges elements **in-place** within the same array.
- Requires only a few temporary variables:
    - pivot → the pivot element
    - i, j → loop indices
    - temp → for swapping

$$\mathbf{Spartition=\Theta(1)}$$

Constant memory, independent of array size.

Step 2: Space for Recursion (Call Stack)

Quick Sort is recursive. Each recursive call creates a **stack frame** storing:

- Local variables (low, high, pivotIndex)
- Return address

The **total recursion depth** determines the total stack memory.

Case 1: Best Case (Balanced Partition)

- Pivot always splits array roughly in half: sizes ≈ n/2n/2n/2

- Recursion depth: **Rbest=log2n**
- Total stack space: Sbest=Rbest·Θ(1)=Θ(logn)

Case 2: Average Case (Random Pivot)

- Pivot position is random → recursion tree is reasonably balanced
- Expected recursion depth ≈ log⁡2n\log_2 nlog2n

Savg=Θ(logn)

Case 3: Worst Case (Unbalanced Partition)

- Pivot always splits extremely unbalanced: one side = n−1n-1n−1, other = 0
- Recursion depth = nnn (linear chain of calls)

Sworst=n·Θ(1)=Θ(n)

Step 3: Total Space Complexity

Case	Recursion Depth	Partition Memory	Total Auxiliary Space
Best	$\log_2 n$	$\Theta(1)$	$\Theta(\log n)$
Average	$\log_2 n$	$\Theta(1)$	$\Theta(\log n)$
Worst	n	$\Theta(1)$	$\Theta(n)$

Step 4: Intuition / Level-by-Level

1. **Best/Average Case**:
   - At any moment, the stack holds frames only along a path from the root to a leaf of the recursion tree.
   - Balanced splitting → stack grows logarithmically → O(log n)
2. **Worst Case**:
   - Linear recursion chain → all n frames on stack → O(n)

Step 5: Example for n = 8

Best Case (Balanced Splits)

Level 0: size = 8
Level 1: two calls of size 4 each
Level 2: four calls of size 2 each
Level 3: eight calls of size 1 each
Max stack depth = 3 → O(log n)

Worst Case (Unbalanced Splits)

Level 0: size = 8
Level 1: size = 7
Level 2: size = 6
...
Level 7: size = 1
Max stack depth = 8 → O(n)

Step 6: Conclusion

- **Quick Sort is in-place** → partition uses constant extra memory
- **Auxiliary space dominated by recursion stack**
- **Final Space Complexity**:

This low memory footprint is ideal for hospital triage systems, ensuring stable operation even under heavy patient load.

GITHUB LINK : https://github.com/Sanjeev82210/DAA-Case-Study-2420090051