	Pimpri Chinchwad Education Trust's **Pimpri Chinchwad College of Engineering**
Assignment 2	

1. Problem Statement:

Movie Recommendation System Optimization A popular OTT platform, StreamFlix, offers personalized recommendations by sorting movies based on user preferences, such as IMDB rating, release year, or watch time popularity. However, during peak hours, sorting large datasets slows down the system. As a backend engineer, you must: ● Implement Quicksort to efficiently sort movies based on various user-selected parameters. ● Handle large datasets containing of movies while maintaining fast response times

2. Course Objective:

1. To know the basics of computational complexity of various algorithms.
2. To select appropriate algorithm design strategies to solve real-world problems.

3. Course Outcome:

1. Analyze the asymptotic performance of algorithms
2. Solve computational problems by applying suitable paradigms of Divide and Conquer or Greedy methodologies

4. Theory:

QuickSort is one of the most widely used sorting algorithms because of its efficiency, simplicity in concept, and adaptability. It is based on the divide-and-conquer paradigm, which makes it highly effective in sorting large datasets. Unlike some other sorting algorithms, QuickSort is an in-place algorithm, meaning it requires very little extra memory, and it avoids the overhead of merging, as seen in Merge Sort.

The working of QuickSort can be explained in three major steps:

1. Divide: A pivot element is selected from the array. The choice of pivot can significantly affect performance. Common strategies include choosing the last element, the first element, a random element, or using the median-of-three method (the median of the first, middle, and last elements).
2. Partition: The array is rearranged so that all elements smaller than the pivot are placed to its left and all elements larger are placed to its right. The pivot itself is placed in its correct sorted position.
3. Conquer: The same process is applied recursively to the left and right subarrays created by the partition step.

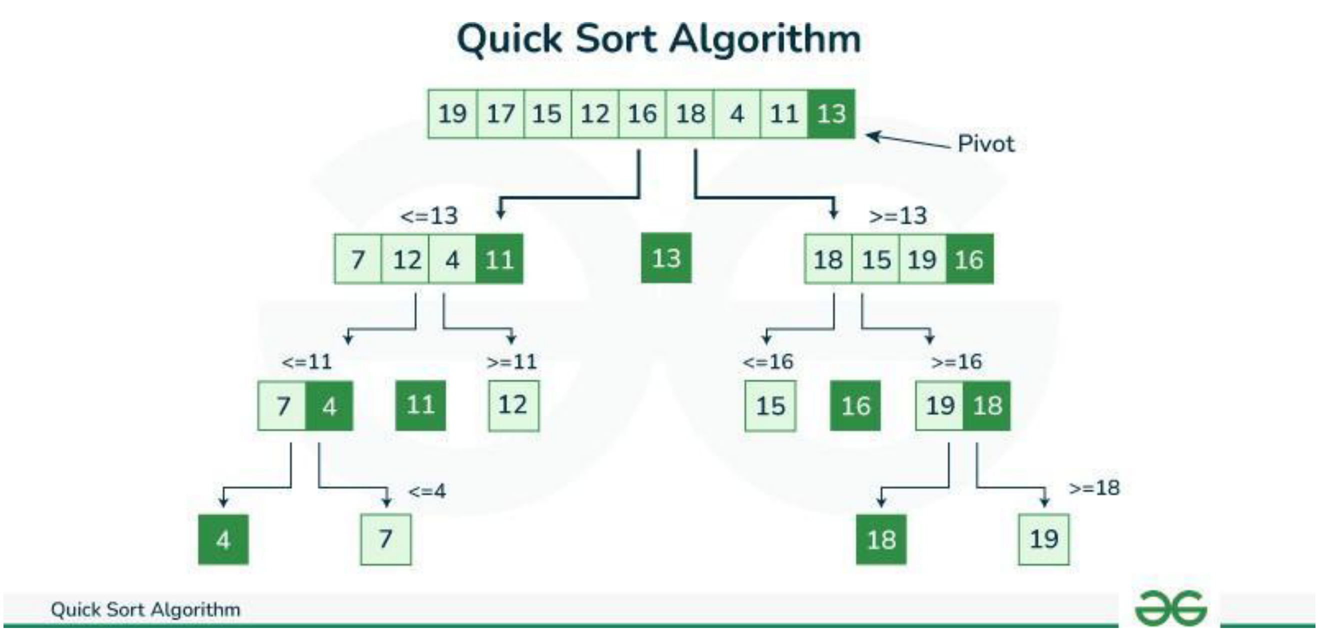
Unlike some other divide-and-conquer algorithms, QuickSort does not require a separate combine step. Once all recursive calls complete, the array is already sorted due to the placement of pivots at each stage.

The efficiency of QuickSort lies in its average-case time complexity of $O(n \log n)$, which makes it suitable for a wide range of applications. However, in the worst case—when the pivot selection is poor and the partitioning is unbalanced—its performance can degrade to $O(n^2)$. To minimize the likelihood of this scenario, several optimizations are often applied:

● Random Pivot Selection: Choosing a random pivot reduces the chance of encountering consistently unbalanced partitions.
● Median-of-Three Method: Using the median of the first, middle, and last elements as the pivot leads to more balanced partitions in practice.
● Tail Recursion Elimination: By converting the tail-recursive calls into iterative steps, the recursion depth can be reduced, improving space efficiency.
● Hybrid Approach: For small subarrays, switching from QuickSort to a simpler algorithm like Insertion Sort can improve performance since the overhead of recursion becomes significant for small input sizes.

Time and Space Complexity

Case	**Time Complexity**	Description
Best Case	**O(n log n)**	Balanced partitions
Average Case	O(n log n)	Typical for random input
Worst Case	**O(n²)**	When pivot is poorly chosen
Space	**O(log n)**	For recursive stack calls



5. Implementation:

QuickSort Algorithm

1. Start with an unsorted array.
2. Choose a pivot element (last element, first element, random, or median-of-three).
3. Partition the array so that:
   - o All elements smaller than the pivot are placed before it.
   - o All elements greater than the pivot are placed after it.
4. Place the pivot in its correct sorted position.
5. Recursively apply QuickSort on the left subarray and right subarray.
6. Continue until subarrays have size 0 or 1 (base case).
7. End when the entire array is sorted.

Pseudo Code for QuickSort

Procedure QUICK_SORT(arr, low,
high): if low < high then
pivotIndex ← PARTITION(arr, low, high)
// Recursively sort elements before and after partition
QUICK_SORT(arr, low, pivotIndex - 1) QUICK_SORT(arr,
pivotIndex + 1, high)
end if
End Procedure

Procedure PARTITION(arr, low, high):
pivot ← arr[high] // choose last element as
pivot i ← low - 1 // index of smaller element
for j ← low to high - 1 do
if arr[j] ≤ pivot then i
← i + 1
SWAP(arr[i], arr[j]) end
if
end for

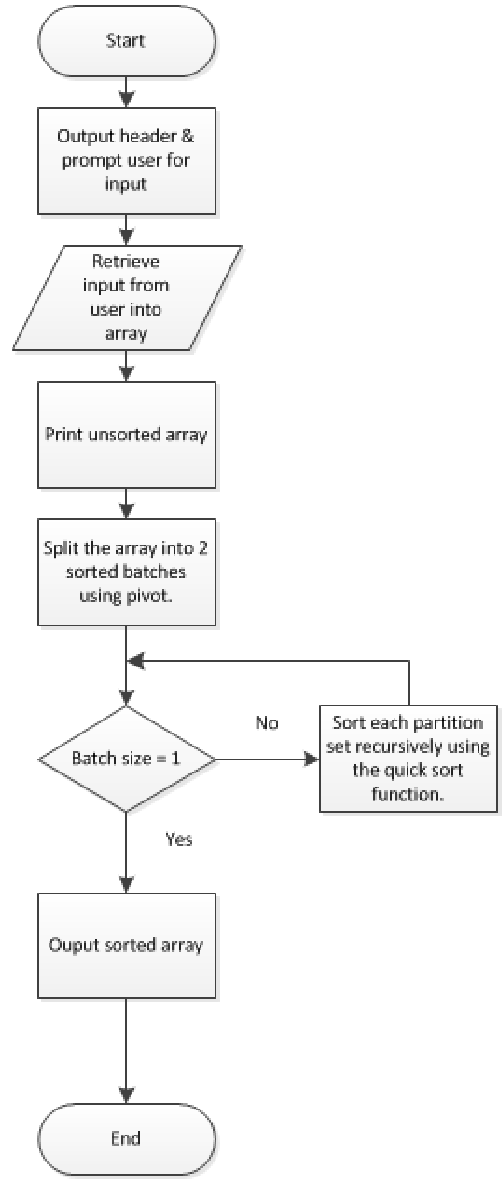
SWAP(arr[i + 1], arr[high]) // place pivot at correct position
return (i + 1) // pivot index
End Procedure

example to clearly see how QuickSort works.
Example Array:

[3, 1, 4, 2]

Step 1: Choose pivot
We choose the last element → pivot = 2. Step 2: Partition around pivot Compare each element with 2:
3 > 2 → goes to right
1 < 2 → goes to left
4 > 2 → goes to right

After partition: [1, 2, 4, 3]
Pivot 2 is in its correct sorted position.
Step 3: Recursive calls
Left subarray: [1] → already sorted
Right subarray: [4, 3]
Step 4: QuickSort on [4, 3]
Pivot = 3 Compare:
4 > 3 → goes right
After partition: [3, 4]
Final Sorted Array
Combine results: [1, 2, 3, 4]



6. Code:

```
#include <iostream>
#include <vector>
#include <string>
#include <fstream>
#include <sstream>
#include <chrono>
#include <algorithm>
#include <cmath>

using namespace std;
using namespace std::chrono;

struct Movie {
   string title;
   float rating;
   int releaseYear;
   int views;

   void display() const {
      cout << title << " | Score: " << rating
           << " | Year: " << releaseYear
           << " | Popularity: " << views << endl;
   }
};

void quickSorter(vector<Movie> &data, int low, int
high, bool (*compare)(const Movie &, const Movie
&)) {
   if (low >= high) return;

   int pivotPos = low;
   Movie pivotElement = data[high];

   for (int j = low; j < high; j++) {
      if (compare(data[j], pivotElement)) {
         swap(data[j], data[pivotPos]);
         pivotPos++;
      }
   }

   swap(data[pivotPos], data[high]);
   quickSorter(data, low, pivotPos - 1, compare);
   quickSorter(data, pivotPos + 1, high, compare);
}

bool byRatingAsc(const Movie &x, const Movie &y) {
return x.rating < y.rating; }
bool byRatingDesc(const Movie &x, const Movie &y)
{ return x.rating > y.rating; }
bool byYearAsc(const Movie &x, const Movie &y) {
return x.releaseYear < y.releaseYear; }
bool byYearDesc(const Movie &x, const Movie &y) {
return x.releaseYear > y.releaseYear; }
bool byViewsAsc(const Movie &x, const Movie &y) {
return x.views < y.views; }
bool byViewsDesc(const Movie &x, const Movie &y)
{ return x.views > y.views; }

vector<Movie> readMovieCSV(const string &path) {
   vector<Movie> movieList;
   ifstream input(path);

   if (!input.is_open()) {
       cerr << "Error: Unable to open file " << path <<
endl;
      return movieList;
   }

   string line;
   getline(input, line);

   while (getline(input, line)) {
      stringstream ss(line);
      string titleStr, ratingStr, yearStr, viewsStr;

      if (!getline(ss, titleStr, ',') ||
         !getline(ss, ratingStr, ',') ||
         !getline(ss, yearStr, ',') ||
         !getline(ss, viewsStr, ',')) {
         continue;
      }

      try {
         Movie m;
         m.title = titleStr;
         m.rating = stof(ratingStr);
         m.releaseYear = stoi(yearStr);
         m.views = stoi(viewsStr);
         movieList.push_back(m);
      } catch (...) {
         continue;
      }
   }
   return movieList;
}

int main() {
   string filePath = "movies_real_titles.csv";
   vector<Movie> movies = readMovieCSV(filePath);

   if (movies.empty()) {
      cout << "No movie records found!\n";
      return 0;
   }
```

```
    cout << "Sort by which attribute?
(rating/year/views): ";
    string userChoice;
    cin >> userChoice;
    transform(userChoice.begin(), userChoice.end(),
userChoice.begin(), ::tolower);

    cout << "Sort order? (asc/desc): ";
    string orderChoice;
    cin >> orderChoice;
    transform(orderChoice.begin(), orderChoice.end(),
orderChoice.begin(), ::tolower);

    bool (*cmp)(const Movie &, const Movie &);

    if (userChoice == "rating") {
        cmp = (orderChoice == "desc") ? byRatingDesc :
byRatingAsc;
    } else if (userChoice == "year") {
        cmp = (orderChoice == "desc") ? byYearDesc :
byYearAsc;
    } else if (userChoice == "views") {
        cmp = (orderChoice == "desc") ? byViewsDesc :
byViewsAsc;
    } else {
        cout << "Invalid input! Default sorting by rating
ascending.\n";
        cmp = byRatingAsc;
    }

    auto begin = high_resolution_clock::now();
    quickSorter(movies, 0, movies.size() - 1, cmp);
    auto end = high_resolution_clock::now();

    cout << "\n--- Top 10 Movies Sorted by " <<
userChoice
         << " (" << orderChoice << ") ---\n";

    int lastIdx = max(0, (int)movies.size() - 10);
    for (int i = movies.size() - 1; i >= lastIdx; --i) {
        movies[i].display();
    }

    duration<double> totalTime = end - begin;
    cout << "\nProcessed " << movies.size() << " entries
in "
         << totalTime.count() << " seconds.\n";

    return 0;
}

}
```

7. Output:

```
Top 10 movies sorted by year:
"The Wolf of Wall Street 16" -> Rating: 7.9, Year: 2024, Popularity: 281063
"Avengers: Infinity War 12" -> Rating: 6.1, Year: 2024, Popularity: 786173
"Gladiator 9" -> Rating: 4.9, Year: 2024, Popularity: 145947
"Parasite 5" -> Rating: 6.1, Year: 2024, Popularity: 401513
"WALL·E 4" -> Rating: 1.2, Year: 2024, Popularity: 422627
"Back to the Future 5" -> Rating: 8, Year: 2024, Popularity: 575837
"A Beautiful Mind 16" -> Rating: 5.7, Year: 2024, Popularity: 540292
"The Big Short 5" -> Rating: 2.1, Year: 2024, Popularity: 684093
"Saving Private Ryan 7" -> Rating: 6.1, Year: 2024, Popularity: 901489
"Pulp Fiction 16" -> Rating: 1.5, Year: 2024, Popularity: 792116

Sorted 1000 movies in 0.00161994 seconds.
```

8. Conclusion:

QuickSort is an efficient and memory-friendly algorithm ideal for sorting large movie datasets based on user preferences like rating, release year, or popularity. By using QuickSort, StreamFlix can ensure fast and responsive recommendations even during peak hours, improving performance and user experience.