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	**$O(n^2)$**	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:
    - All elements smaller than the pivot are placed before it.
    - 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. Output:

Example Dataset (sorted by IMDB rating)

Movies with ratings: [8.2, 6.5, 9.0, 7.3, 8.7]

Step 1: Divide

Choose pivot (last element) → pivot = 8.7

Partition:

- Elements ≤ 8.7 → [8.2, 6.5, 7.3]
- Elements > 8.7 → [9.0]

Array after partition: [8.2, 6.5, 7.3, 8.7, 9.0]

Step 2: Recursively Apply QuickSort

Left subarray: [8.2, 6.5, 7.3]
Pivot = 7.3 → Partition → [6.5, 7.3, 8.2]

Right subarray: [9.0] (already sorted)

Step 3: Combine Results

Final sorted array by IMDB rating → [6.5, 7.3, 8.2, 8.7, 9.0]

Final Output

Movies sorted efficiently based on user-selected parameter (IMDB rating in this case):
[6.5, 7.3, 8.2, 8.7, 9.0]

Code:

```
#include <iostream>
#include <vector>
#include <string>
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <chrono>

using namespace std;
using namespace std::chrono;

struct Movie {
  string title;
  float rating;
  int release_year;
  int popularity;

  void display() const {
    cout << title << " | Rating: " << rating
         << " | Year: " << release_year
         << " | Popularity: " << popularity << endl;
  }
};

// QuickSort with comparator function pointer
void quickSort(vector<Movie> &movies, int low, int high, bool (*compare)(const Movie &, const Movie &)) {
  if (low < high) {
    int pivotIndex = low;
    Movie pivot = movies[high];
    for (int i = low; i < high; i++) {
      if (compare(movies[i], pivot)) {
        swap(movies[i], movies[pivotIndex]);
        pivotIndex++;
      }
    }
    swap(movies[pivotIndex], movies[high]);
    quickSort(movies, low, pivotIndex - 1, compare);
```

```
        quickSort(movies, pivotIndex + 1, high, compare);
    }
}

// Comparators
bool compareByRating(const Movie &a, const Movie &b) {
    return a.rating < b.rating;
}
bool compareByYear(const Movie &a, const Movie &b) {
    return a.release_year < b.release_year;
}
bool compareByPopularity(const Movie &a, const Movie &b) {
    return a.popularity < b.popularity;
}

// Random movie generator
vector<Movie> generateMovies(int n) {
    vector<Movie> movies;
    for (int i = 0; i < n; i++) {
        Movie m;
        m.title = "Movie " + to_string(i + 1);
        m.rating = static_cast<float>(rand() % 90 + 10) / 10.0f; // 1.0 to 10.0
        m.release_year = rand() % 45 + 1980; // 1980 to 2024
        m.popularity = rand() % 1000000 + 1000; // 1,000 to 1,000,000
        movies.push_back(m);
    }
    return movies;
}

int main() {
    srand(time(0));
    int num_movies = 100000;
    vector<Movie> movies = generateMovies(num_movies);

    cout << "Sort movies by (rating/year/popularity): ";
    string sort_by;
    cin >> sort_by;

    bool (*compare)(const Movie &, const Movie &);
    if (sort_by == "rating")
        compare = compareByRating;
    else if (sort_by == "year")
        compare = compareByYear;
    else if (sort_by == "popularity")
        compare = compareByPopularity;
    else {
        cout << "Invalid choice. Defaulting to rating.\n";
        compare = compareByRating;
    }

    auto start = high_resolution_clock::now();
    quickSort(movies, 0, movies.size() - 1, compare);
    auto end = high_resolution_clock::now();
    duration<double> diff = end - start;

    // Show top 10 in descending order
    cout << "\nTop 10 movies by " << sort_by << ":\n";
    for (int i = num_movies - 1; i >= num_movies - 10; --i) {
        movies[i].display();
    }

    cout << "\nSorted " << num_movies << " movies in " << diff.count() << " seconds.\n";

    return 0;
}
```

Output:

```
Sort movies by (rating/year/popularity): rating

Top 10 movies by rating:
Movie 53421 | Rating: 9.9 | Year: 1999 | Popularity: 437069
Movie 55397 | Rating: 9.9 | Year: 1994 | Popularity: 61724
Movie 7314 | Rating: 9.9 | Year: 2005 | Popularity: 201797
Movie 52041 | Rating: 9.9 | Year: 2012 | Popularity: 228692
Movie 156 | Rating: 9.9 | Year: 1986 | Popularity: 889400
Movie 47624 | Rating: 9.9 | Year: 2018 | Popularity: 744989
Movie 63765 | Rating: 9.9 | Year: 2014 | Popularity: 684955
Movie 87169 | Rating: 9.9 | Year: 2003 | Popularity: 123922
Movie 16456 | Rating: 9.9 | Year: 2006 | Popularity: 869661
Movie 17881 | Rating: 9.9 | Year: 1993 | Popularity: 215094

Sorted 100000 movies in 0.456534 seconds.


=== Code Execution Successful ===
```

7.**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.

8.Questions: