Analysis of Algorithms

BLG 335E

Project 1 Report

ÖMER YILDIZ
yildizom20@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 21.11.2023

1. Implementation

1.1. Implementation of QuickSort with Different Pivoting Strategies

1.1.1. Implementation Details

- Partitioning Strategies
  Last Element (partitionLast)

  Chooses the last element as the pivot. Iterates through the array, placing elements smaller than the pivot to its left. Swaps the pivot into its correct position.

  Random Element (partitionRandom)

  Randomly selects a pivot for partitioning. Swaps the randomly chosen pivot with the last element. Proceeds with the same partitioning process as in the last element strategy.

  Median of Three (partitionMedianOfThree)

  Selects the median of three elements (first, middle, and last) as the pivot. Swaps the chosen pivot with the last element. Follows the same partitioning process as in the last element strategy.

- Sorting Function
  QuickSort (quickSort)

  Recursively applies the chosen pivot strategy to partition the array. Sorts the subarrays on either side of the pivot until the base case is reached.

- Logging and Verbose Output
  Logs each pivot and the corresponding array configuration to a log file when verbose output is enabled.

1.1.2. Related Recurrence Relation for QuickSort

The recurrence relation for the time complexity of the QuickSort algorithm can be expressed as follows:

$$T(n) = \begin{cases} 0 & \text{if } n \leq 1, \\ T(q) + T(n-q-1) + O(n) & \text{otherwise,} \end{cases}$$

where:

n is the size of the input array,

q is the index obtained from the partitioning process.

In the recurrence relation, $T(n)$ represents the time complexity to sort an array of size n. The recurrence relation is recursive, as the time complexity of sorting an array of size n is expressed in terms of sorting two subarrays of sizes q and $n - q - 1$, respectively. The partitioning process takes $O(n)$ time.

1.1.3. Time and Space Complexity

- Best Case
  O(nlogn) (when the pivot consistently divides the array into two nearly equal halves)
- Average Case
  O(nlogn)
- Worst Case
  O(n^2) (when the pivot consistently selects the smallest or largest element)

	Population1	**Population2**	**Population3**	**Population4**
Last Element	57419212208	xxx	278025993088	17020327744
Random Element	15277517201	19095254384	17061450619	17184319952
Median of 3	14707056410	16780947743	16504268077	16124783826

Table 1.1: Comparison of different pivoting strategies on input data.

In my analysis, i observed distinct performance characteristics among different pivot selection strategies in the context of QuickSort. Notably, the last element strategy exhibited inferior performance compared to the Median of 3 and Random Element strategies.

For the ordered datasets, Population2 and Population3, the last element strategy resulted in a time complexity of $O(n^2)$, indicating a suboptimal performance for nearly sorted arrays. Even for Population1, which is not entirely ordered, the last element strategy performed worse than the Random Element strategy, especially considering that Population4, a predominantly ordered dataset, showcased better performance.

The Median of 3 and Random Element strategies displayed greater variability in outcomes. Our effort to populate the table with average values revealed consistent advantages for these strategies compared to the last element strategy, particularly when applied to Population1, Population2, and Population3. It's noteworthy that, unexpectedly, the Random Element strategy outperformed the Median of 3 in the case of Population2.

In summary, my analysis emphasizes the importance of pivot selection strategies in QuickSort, showcasing that strategies like Median of 3 and Random Element can offer improved performance, especially in scenarios involving nearly sorted datasets.

1.2. Hybrid Implementation of QuickSort and Insertion Sort

1.2.1. Implementation Details

The hybrid implementation of QuickSort and Insertion Sort is designed to take advantage of the strengths of both algorithms based on a threshold k. For arrays larger than k, QuickSort with a specified pivot strategy is used. When the array size is smaller than or equal to k, Insertion Sort is applied.

Listing 1.1: Hybrid QuickSort and Insertion Sort

```
#include <iostream>
#include <vector>

// Structure to represent city data
struct CityData {
    std::string city;
    int population;
};

// Function to perform Insertion Sort
void insertionSort(std::vector<CityData>& arr, int low, int high) {
    // Implementation of Insertion Sort
    // ...
}

// Function to perform QuickSort
void quickSort(std::vector<CityData>& arr, int low, int high, char pivotS
    // Implementation of QuickSort
    // ...
}

// Hybrid QuickSort function
void hybridQuickSort(std::vector<CityData>& arr, int low, int high, char
    if (low < high) {
        if (high - low + 1 <= k) {
            insertionSort(arr, low, high);
// Apply Insertion Sort for small subarrays
        } else {
            int partitionIndex;
            // Choose partition strategy based on pivotStrategy
            switch (pivotStrategy) {
                case 'l':
                    partitionIndex = partitionLast(arr, low, high);
                    break;
                case 'r':
                    partitionIndex = partitionRandom(arr, low, high);
                    break;
                case 'm':
                    partitionIndex = partitionMedianOfThree(arr, low, hig
                    break;
                default:
                    return;
            }
            // Recursive calls for QuickSort
            hybridQuickSort(arr, low, partitionIndex − 1, pivotStrategy,
            hybridQuickSort(arr, partitionIndex + 1, high, pivotStrategy,
        }
    }
}

int main() {
    // Example usage of hybridQuickSort
    // ...
    return 0;
}
```

1.2.2. Related Recurrence Relation

The recurrence relation for the hybrid implementation depends on the input size and the threshold k. Assuming $T(n)$ is the time complexity, it can be expressed as:

$$T(n) = \begin{cases} O(n \log n) & \text{if } n > k \\ O(n^2) & \text{if } n \leq k \end{cases}$$

The first case corresponds to the time complexity of QuickSort, and the second case corresponds to the time complexity of Insertion Sort.

1.2.3. Time and Space Complexity

The time complexity of the hybrid implementation is influenced by the choice of the threshold k. For larger arrays, the time complexity is dominated by QuickSort, which is $O(n \log n)$. For smaller arrays (size $\leq k$), the time complexity is $O(n^2)$ due to the use of Insertion Sort.

The space complexity is determined by the recursive calls in QuickSort and the auxiliary space used by Insertion Sort. In the worst case, the space complexity is $O(n)$.

Threshold (k)	1	100	1000
Population4	822465708	555566542	262321250

Table 1.2: First Half of the Comparison of different thresholds on the hybrid QuickSort algorithm on input data.

Threshold (k)	10000	100000	1000000
Population4	102782959	106585500	105998208

Table 1.3: Second Half of the Comparison of different thresholds on the hybrid QuickSort algorithm on input data.

I would like to emphasize that the observed results vary across different executions, and the values presented here are based on the most commonly observed outcomes. In instances where the dataset size is large, such as 10,000, a value close to the total number of records, the system tends to exhibit behavior akin to $O(n^2)$. This occurs because the advantages of QuickSort with large-sized arrays diminish.

It is challenging to determine strict upper and lower bounds that significantly impact performance. However, through multiple runs, a threshold of 10,000 consistently yielded optimal results in the majority of cases. This threshold is particularly close to our dataset size, providing a balance between the advantages of QuickSort with larger arrays and the challenges associated with near $O(n^2)$ behavior.