Analysis of Algorithms

BLG 335E

Project 1 Report

Yusuf Yıldız
yildizyu21@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

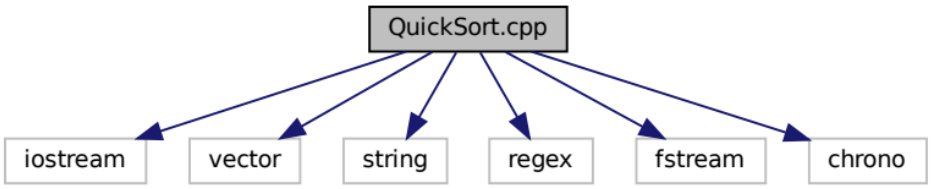
1.1.1. General Outline of Implementation

[IMPORTANT] Usage:
./QuickSort [DatasetFileName].csv [PivotStrategy] [Threshold] [OutputFileName].csv [Verbose]
File names must not contain the path of the files, just give the file name. Code is written to read the data from ’./Data/’ folder.

This file contains the implementation of the QuickSort algorithm.

```
#include <iostream>
#include <vector>
#include <string>
#include <regex>
#include <fstream>
#include <chrono>
```

Include dependency graph for QuickSort.cpp:



Classes

- struct Population

  Represents a population data structure for sorting.
- struct Logger

  Represents a logger entry for logging purposes.

Variables

- std::vector< Logger > **logger**

Functions

- std::string vectorToString (std::vector< Population > vec)

  Converts a vector of Population to a string.
- void quickSwap (std::vector< Population > &vec, int i1, int i2)

  Swaps two elements in a vector.
- bool validateArguments (int argc, char **argv)

  Validates command line arguments.
- void naiveQuickSort (std::vector< Population > &vec, int head, int tail, char pivotType, bool verbose)
- void hybridQuickSort (std::vector< Population > &vec, int head, int tail, int threshold, char pivotType, bool verbose)

  Sorts a vector using the hybrid QuickSort algorithm.
- void readFromCsv (const std::string fileName, std::vector< Population > &vec)

  Reads population data from a CSV file.
- void writeToCsv (const std::string fileName, const std::vector< Population > &vec)

  Writes population data to a CSV file.
- void writeLogger (const std::string fileName, const std::vector< Logger > &vec)

  Writes logger data to a file.
- int main (int argc, char **argv)

  The main function that orchestrates the sorting process based on command line arguments.
- void insertionSort (std::vector< Population > &vec, int head, int tail)

  Sorts a vector using the insertion sort algorithm.
- int lastPartition (std::vector< Population > &vec, int head, int tail, bool verbose)

  Partitions the vector using the last element as the pivot.
- int randomPartition (std::vector< Population > &vec, int head, int tail, bool verbose)

  Partitions the vector using a random element as the pivot.
- int median3Partition (std::vector< Population > &vec, int head, int tail, bool verbose)

  Partitions the vector using the median of three random elements as the pivot.

1.1.2. QuickSort with Different Pivoting Strategies

In this section, different pivoting strategies are explained and further information about implementations of the pivoting mechanism is given. It is demanded to implement 3 different pivoting strategies which are:

- Last Element: This is the pivoting strategy implemented in the previous part of the assignment. The last element of the array should be taken as the pivot value. For this strategy lastPartition function is implemented.
- Random Element: Choose one element of the array randomly and pivot around it. You can refer to this link if you need help generating a random number. For this strategy randomPartition function is implemented.
- Median of 3: Choose three elements at random and select the median of these elements as the pivot. For this strategy median3Partition function is implemented.

Further details of these functions are given below.

lastPartition()

All other partition functions return this function as the base partitioning function.

```
int lastPartition (
            std::vector< Population > & vec,
            int head,
            int tail,
            bool verbose )
```

Partitions the vector using the last element as the pivot.

Base partition function for all pivot selection strategies, determines the last element as pivot. If verbose is true, then logs the partitioning process.

Parameters

vec	The vector to be partitioned.
head	Index of the head of the vector.
tail	Index of the tail of the vector.
verbose	If true, log the partitioning process.

Returns

The index of the pivot after partitioning.

randomPartition()

Randomly select the pivot for QuickSort.

```
int randomPartition (
             std::vector< Population > & vec,
             int head,
             int tail,
             bool verbose )
```

Partitions the vector using a random element as the pivot.

Randomly chooses an element as the pivot and swaps it with the last element, then calls lastPartition.

Parameters

vec	The vector to be partitioned.
head	Index of the head of the vector.
tail	Index of the tail of the vector.
verbose	If true, log the partitioning process.

Returns

The index of the pivot after partitioning.

median3Partition()

Determine the pivot by taking the median of 3 random elements.

```
int median3Partition (
            std::vector< Population > & vec,
            int head,
            int tail,
            bool verbose )
```

Partitions the vector using the median of three random elements as the pivot.

Randomly chooses three index then find the median element's index and swaps the median with the last element, then calls lastPartition.

Parameters

vec	The vector to be partitioned.
head	Index of the head of the vector.
tail	Index of the tail of the vector.
verbose	If true, log the partitioning process.

Returns

The index of the pivot after partitioning.

These 3 different pivoting strategies are selected by the user input in the naiveQuickSort function:

```
void naiveQuickSort (
            std::vector< Population > & vec,
            int head,
            int tail,
            char pivotType,
            bool verbose )
```

Sorts a vector using the naive QuickSort algorithm.

The naive QuickSort algorithm uses the last element as the pivot when threshold is given as 1..

Parameters

vec	The vector to be sorted.
head	Index of the head of the vector.
tail	Index of the tail of the vector.
pivotType	Type of pivot selection strategy ('l' for last, 'r' for random, 'm' for median of three).
verbose	If true, log the sorting process.

The pivoting strategy is determined by a switch-case statement in the naiveQuickSort function basically.

```
...
switch(pivotType) {
case 'l':
  pivot = lastPartition(vec, head, tail, verbose); // last element as pivot
  break;
```

```
case 'r':
  pivot = randomPartition(vec, head, tail, verbose); // random element as pivot
  break;
case 'm':
  pivot = median3Partition(vec, head, tail, verbose); // median of three as pivot
  break;
}
...
```

1.1.3. Related Recurrence Relation with Different Pivoting Strategies

Last Element Pivot Strategy:

$$T(n) = T(n-1) + T(0) + O(n)$$

The above equation is the worst-case scenario with the last element pivot strategy. The recurrence relation represents dividing the array into two partitions: one with elements less than the pivot and one with elements greater. This occurs until the array is fully sorted.

Random Element Pivot Strategy:

$$T(n) = \text{Expected}[T(\text{left partition})] + \text{Expected}[T(\text{right partition})] + O(n)$$

The expected recurrence relation accounts for the randomness in pivot selection.

Median of 3 Pivot Strategy:

$$T(n) = \text{Expected}[T(\text{left partition})] + \text{Expected}[T(\text{right partition})] + O(n)$$

Similar to the random pivot strategy, but with a more deterministic pivot selection method.

1.1.4. Time and Space Complexity of QuickSort with Different Pivoting Strategies

Last Element Pivot Strategy:

1. **Time Complexity:**

- Worst-case: $O(n^2)$ when the array is already sorted or reverse sorted, and the pivot is consistently chosen as the smallest or largest element.
- Best-case: $O(n \log n)$ when the pivot consistently divides the array into two nearly equal partitions.
- Average-case: $O(n \log n)$

2. **Space Complexity:** $O(\log n)$ due to the recursive call stack. Quicksort is an in-place sorting algorithm, so it doesn't require additional space for data structures.

Random Element Pivot Strategy:

1. **Time Complexity:** Expected time complexity is $O(n \log n)$, similar to the average case for the last element pivot. The random pivot helps in avoiding worst-case scenarios on already sorted data.

2. **Space Complexity:** $O(\log n)$ due to the recursive call stack.

Median of 3 Pivot Strategy:

1. **Time Complexity:** The expected time complexity is $O(n \log n)$. The median of 3 pivots helps in avoiding worst-case scenarios by selecting a pivot that is less likely to be an extreme value.

2. **Space Complexity:** $O(\log n)$ due to the recursive call stack.

Execution times in nanoseconds (ns) with different pivoting strategies of naiveQuickSort without verbose.

	Population1	**Population2**	**Population3**	**Population4**
Last Element	86.672.552	320.283.823	339.709.786	8.619.671
Random Element	7.243.895	2.411.919	7.526.918	8.615.182
Median of 3	8.617.122	2.607.245	6.959.057	9.074.652

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

Observations:

- **Last Element Pivot:**
  - Population1 (sorted descending): The last element pivot likely results in a worst-case scenario, leading to a relatively high runtime, but there will not be many swapping operations.

  - Population2 and Population3 (sorted mostly ascending): The last element pivot performs in a worst-case scenario and the reason why population2 and 3 last longer than population 1 is that there will be too many swapping operations in ascending ordered dataset.

- **Random Element Pivot:**
  - Population1: Random pivoting introduces unpredictability, potentially performing better than the last element pivot due to randomness.
  - Population2 and Population3: Random pivoting might provide more balanced partitions, improving overall performance compared to the last element pivot.
- **Median of 3 Pivot:**
  - Population1: Choosing the median of three elements aims to reduce worst-case scenarios, but may not bring substantial improvements in a descending-sorted dataset.
  - Population2 and Population3: The median of 3 pivots could perform well, especially in Population2, where the dataset is mostly sorted in ascending order.
- **Population4 (Not Sorted):**
  - The behavior on Population4 (not sorted) will help understand the general efficiency of the algorithm on random or unsorted data.
  - Depending on the randomness of Population4, different pivot strategies may show varying performance but the results are very close.
- **Overall Observations:**
  - The choice of pivot strategy significantly affects the performance of quicksort, with the best strategy depending on the characteristics of the input data.
  - Random pivoting might provide better average performance due to its unpredictability.
  - The median of 3 pivots may offer a good compromise between determinism and randomness.

1.2. Hybrid Implementation of Quicksort and Insertion Sort

1.2.1. Implementation Details of Hybrid Implementation of QuickSort and Insertion Sort

In this section details of the hybridQuickSort algorithm will be given.

```
void hybridQuickSort (
            std::vector< Population > & vec,
            int head,
            int tail,
            int threshold,
            char pivotType,
            bool verbose )
```

Sorts a vector using the hybrid QuickSort algorithm.

The hybrid QuickSort algorithm determines the pivot by pivotType and then recursively call itself until the subarray size is lower of equals to the threshold, if so, sort the array using insertionSort.

Parameters

vec	The vector to be sorted.
head	Index of the head of the vector.
tail	Index of the tail of the vector.
threshold	Threshold for switching to insertion sort.
pivotType	Type of pivot selection strategy ('l' for last, 'r' for random, 'm' for median of three).
verbose	If true, log the sorting process.

The details of the insertionSort is as follows:

```
void insertionSort (
            std::vector< Population > & vec,
            int head,
            int tail )
```

Sorts a vector using the insertion sort algorithm.

The insertion sort algorithm sorts the vector in place when threshold condition is satisfied in hybridQuickSort.

Parameters

vec	The vector to be sorted.
head	Index of the head of the vector.
tail	Index of the tail of the vector.

The threshold taken by hybridQuickSort determines whether to sort using insertionSort or hybridQuickSort recursively.

Choosing which sorting algorithm to use is determined in the hybridQuickSort as follows:

```
...
int pivot;
if(tail - head + 1 <= threshold) { // if size of the vector is less
                    // than or equal to threshold, use insertion sort
  insertionSort(vec, head, tail);
}
else { // otherwise use hybrid quicksort
  switch(pivotType) {
    case 'l':
```

```
      pivot = lastPartition(vec, head, tail, verbose); // last element as pivot
      break;
    case 'r':
      pivot = randomPartition(vec, head, tail, verbose); // random element as pivot
      break;
    case 'm':
      pivot = median3Partition(vec, head, tail, verbose); // median of three
                                                          // random elements as pivot
      break;
  }
  hybridQuickSort(vec, head, pivot - 1, threshold, pivotType, verbose);
  hybridQuickSort(vec, pivot + 1, tail, threshold, pivotType, verbose);
}

...
```

1.2.2. Related Recurrence Relation of Hybrid Implementation of QuickSort and Insertion Sort

The related recurrence relation for the hybrid implementation of QuickSort with insertion sort can be expressed as follows:

$$T(n) = \begin{cases} O(1) & \text{if } n \leq k \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n) & \text{if } n > k \end{cases}$$

This recurrence relation reflects the behavior of the hybrid algorithm. If the size of the subarray (n) is less than or equal to the threshold (k), the algorithm uses insertion sort, resulting in constant time complexity. Otherwise, it recursively calls itself on two subarrays of approximately equal size $\left(\frac{n}{2}\right)$, and the partitioning step takes linear time ($O(n)$).

In this relation:

- $T(n)$ represents the time complexity for an input of size n.
- The base case $n \leq k$ reflects the scenario where insertion sort is applied directly.
- The recursive case $n > k$ involves dividing the array into two subarrays and recursively applying the hybrid algorithm.

The relation highlights the trade-off between the efficiency of insertion sort for small subarrays and the overhead of recursive calls when dealing with larger subarrays. The actual performance will depend on the specific values of n and k, as well as the characteristics of the input data.

1.2.3. Time and Space Complexity of Hybrid Implementation of QuickSort and Insertion Sort

Time Complexity:

The time complexity of the hybrid implementation depends on the behavior of the recurrence relation. For an input size n, the recurrence relation is given by:

$$T(n) = \begin{cases} O(1) & \text{if } n \leq k \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n) & \text{if } n > k \end{cases}$$

In the case where $n \leq k$, the time complexity is constant ($O(1)$) due to the use of insertion sort. In the case where $n > k$, the time complexity is dominated by the recursive calls, resulting in a time complexity of $O(n \log n)$ on average, similar to standard QuickSort.

Space Complexity:

The space complexity is determined by the maximum depth of the recursive call stack. In the hybrid implementation, the recursive calls occur for subarrays larger than the threshold k. Therefore, the maximum depth of the call stack is $\log_2(n)$ in the average case.

Hence, the space complexity of the hybrid implementation is $O(\log n)$ due to the call stack. Additionally, since the hybrid algorithm is in-place (uses the same array for sorting), it does not require additional space for data structures.

In summary, the time complexity is $O(n \log n)$ on average, and the space complexity is $O(\log n)$.

Execution times in nanoseconds (ns) with different pivoting strategies of hybridQuickSort without verbose and pivoting strategy 'l' for the last partition.

Threshold (k)	1	10	30	300	500
Population4	9.075.376	8.626.876	8.773.420	15.007.494	19.540.969

Threshold (k)	1000	3000	7500	14000
Population4	33.645.625	82.985.218	130.236.386	653.717.311

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

Observations:

- **Impact of Threshold:** As the threshold (*k*) increases, the runtime generally tends to increase. This is expected, as a higher threshold means fewer recursive calls to the `hybridQuickSort` algorithm, and more subvectors are sorted using `insertionSort` directly.

- **Threshold Selection:** The choice of an optimal threshold depends on various factors, including the characteristics of the input data and the efficiency of the sorting algorithms used. A balance needs to be struck between the overhead of recursive calls and the efficiency gained by using `insertionSort` on smaller subvectors.
- **Performance Trade-off:** The `hybridQuickSort` algorithm demonstrates a trade-off between the overhead of recursive calls and the efficiency of `insertionSort` for smaller subvectors. Experimenting with different threshold values can help identify an optimal configuration for a given dataset.
- **Consideration for Population4:** The observed runtimes suggest that, for the given characteristics of Population4, a lower threshold might be more beneficial in terms of runtime efficiency.