Empirical Analysis of QuickSort Performance with Different Pivot Selection Strategies

By: Tinotenda Chaukura
Course Name: Algorithm and complexity
Professor: Mr. Dariusz Doliwa
Date of Submission: 20/01/2025

Table of Contents

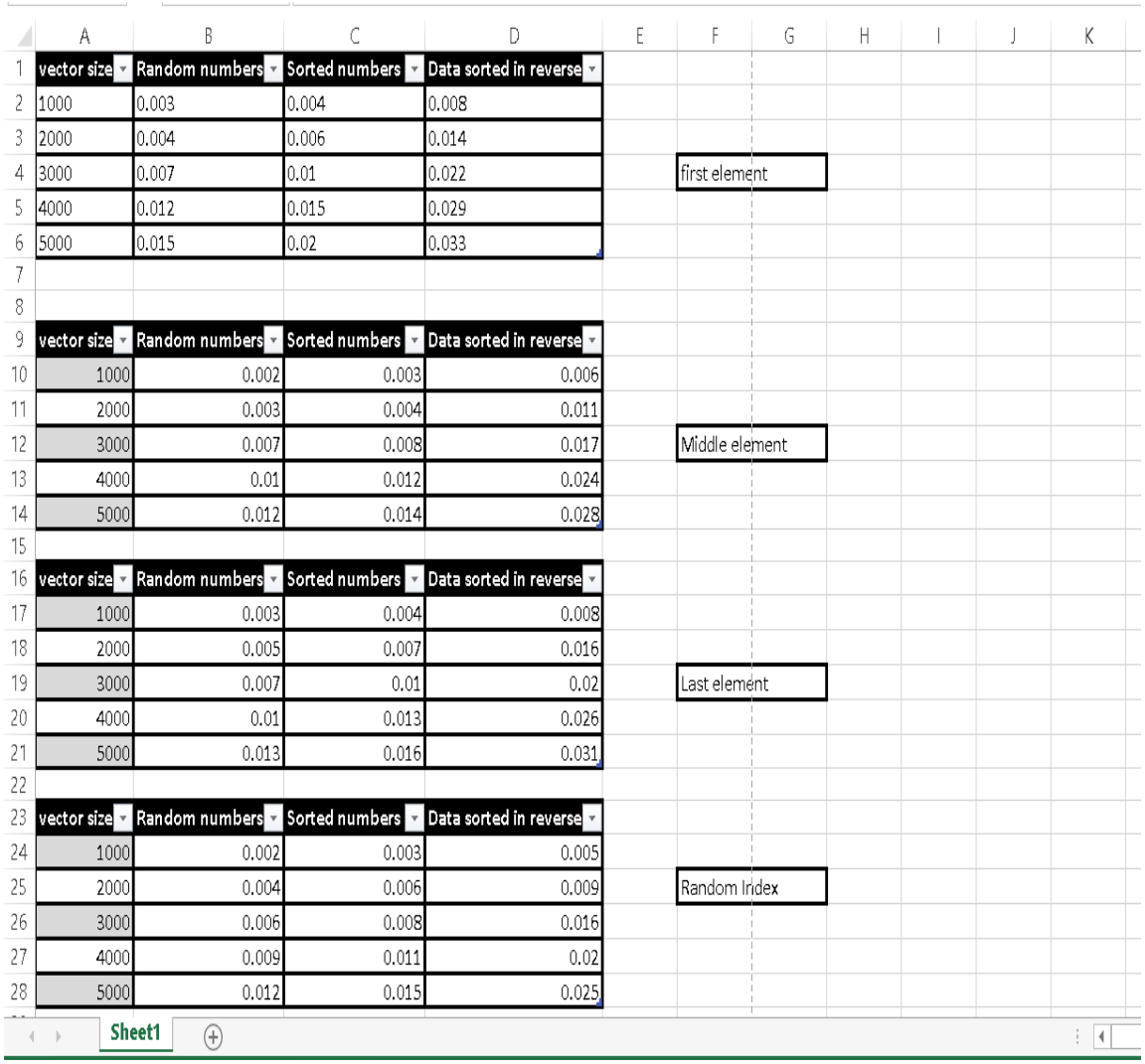
Empirical Analysis of Quicksort Performance with Different Pivot Selection Strategies

1. Goal of the Task and Short Description of the Project

The goal of this project was to test how different pivot selection strategies in the Quicksort algorithm affect its performance on various types of data. Specifically, I tested four pivot strategies: using the first element, middle element, a random element, and the median of the first, middle, and last elements. The datasets I used were of different sizes (1000, 2000, 3000, 4000, and 5000 elements), and each dataset was either sorted, randomly shuffled, or in reverse sorted order. By doing this, I wanted to analyse how each pivot strategy impacts execution time, and whether certain strategies perform better for specific types of data.

2. Addressing Plan with a Table of Execution Times

The table below shows the execution times (in milliseconds) for the Quicksort algorithm with each pivot strategy applied to sorted, random, and reverse sorted datasets. The datasets I used ranged from 1000 to 5000 elements.



vector size	Random numbers	Sorted numbers	Data sorted in reverse
1000	0.003	0.004	0.008
2000	0.004	0.006	0.014
3000	0.007	0.01	0.022
4000	0.012	0.015	0.029
5000	0.015	0.02	0.033

first element

vector size	Random numbers	Sorted numbers	Data sorted in reverse
1000	0.002	0.003	0.006
2000	0.003	0.004	0.011
3000	0.007	0.008	0.017
4000	0.01	0.012	0.024
5000	0.012	0.014	0.028

Middle element

vector size	Random numbers	Sorted numbers	Data sorted in reverse
1000	0.003	0.004	0.008
2000	0.005	0.007	0.016
3000	0.007	0.01	0.02
4000	0.01	0.013	0.026
5000	0.013	0.016	0.031

Last element

vector size	Random numbers	Sorted numbers	Data sorted in reverse
1000	0.002	0.003	0.005
2000	0.004	0.006	0.009
3000	0.006	0.008	0.016
4000	0.009	0.011	0.02
5000	0.012	0.015	0.025

Random Index

Sheet1

3. List of Devices Used

For this project, I used:

- **Microsoft Visual Studio**: This is where I wrote and compiled the C++ code.

  I used the standard library for vector management, random number generation, and timing the execution of each sorting method.

These tools helped me implement and run the sorting algorithms efficiently, and collect the timing data I needed for analysis.

4. Description of the Device Configuration (Including QuickSort and Pivot Implementations)

The source code below is the implementation of the QuickSort algorithm with each of the pivot strategies. It sorts datasets of different sizes and types, and measures the execution times for each strategy.

#include <iostream>

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

using namespace std;

void quickSortFirstPivot(vector<int>& arr, int low, int high);
void quickSortMiddlePivot(vector<int>& arr, int low, int high);
void quickSortRandomPivot(vector<int>& arr, int low, int high);
void quickSortMedianPivot(vector<int>& arr, int low, int high);

int partitionFirst(vector<int>& arr, int low, int high);
int partitionMiddle(vector<int>& arr, int low, int high);
int partitionRandom(vector<int>& arr, int low, int high);
int partitionMedian(vector<int>& arr, int low, int high);

int median(vector<int> arr);

vector<int> generateRandomData(int size);
vector<int> generateSortedData(int size);
vector<int> generateReverseSortedData(int size);

int main() {
```

```
    srand(time(0));

    int sizes[] = { 10000, 20000, 30000, 40000, 50000 };

    for (int size : sizes) {
        cout << "Testing for size: " << size << endl;

        vector<int> randomData = generateRandomData(size);
        vector<int> sortedData = generateSortedData(size);
        vector<int> reverseData = generateReverseSortedData(size);

        vector<int> temp;

        cout << "First Element as Pivot:" << endl;
        temp = randomData;
        auto start = chrono::high_resolution_clock::now();
        quickSortFirstPivot(temp, 0, temp.size() - 1);
        auto end = chrono::high_resolution_clock::now();
        cout << "Random Data: " << chrono::duration<double, milli>(end - start).count() << "
ms" << endl;

        temp = sortedData;
        start = chrono::high_resolution_clock::now();
        quickSortFirstPivot(temp, 0, temp.size() - 1);
        end = chrono::high_resolution_clock::now();
```

```
    cout << "Sorted Data: " << chrono::duration<double, milli>(end - start).count() << "
ms" << endl;



    temp = reverseData;

    start = chrono::high_resolution_clock::now();

    quickSortFirstPivot(temp, 0, temp.size() - 1);

    end = chrono::high_resolution_clock::now();

    cout << "Reverse Data: " << chrono::duration<double, milli>(end - start).count() << "
ms" << endl;



    cout << "Middle Element as Pivot:" << endl;

    temp = randomData;

    start = chrono::high_resolution_clock::now();

    quickSortMiddlePivot(temp, 0, temp.size() - 1);

    end = chrono::high_resolution_clock::now();

    cout << "Random Data: " << chrono::duration<double, milli>(end - start).count() << "
ms" << endl;



    cout << "Random Element as Pivot:" << endl;

    temp = randomData;

    start = chrono::high_resolution_clock::now();

    quickSortRandomPivot(temp, 0, temp.size() - 1);

    end = chrono::high_resolution_clock::now();

    cout << "Random Data: " << chrono::duration<double, milli>(end - start).count() << "
ms" << endl;



    cout << "Median Element as Pivot:" << endl;
```

```
        temp = randomData;
        start = chrono::high_resolution_clock::now();
        quickSortMedianPivot(temp, 0, temp.size() - 1);
        end = chrono::high_resolution_clock::now();
        cout << "Random Data: " << chrono::duration<double, milli>(end - start).count() << "
ms" << endl;

        cout << endl;
    }

    return 0;
}

void quickSortFirstPivot(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pivotIndex = partitionFirst(arr, low, high);
        quickSortFirstPivot(arr, low, pivotIndex - 1);
        quickSortFirstPivot(arr, pivotIndex + 1, high);
    }
}

int partitionFirst(vector<int>& arr, int low, int high) {
    int pivot = arr[low];
    int i = low + 1;

    for (int j = low + 1; j <= high; j++) {
```

```
        if (arr[j] < pivot) {
            swap(arr[i], arr[j]);
            i++;
        }
    }
    swap(arr[low], arr[i - 1]);
    return i - 1;
}

void quickSortMiddlePivot(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pivotIndex = partitionMiddle(arr, low, high);
        quickSortMiddlePivot(arr, low, pivotIndex - 1);
        quickSortMiddlePivot(arr, pivotIndex + 1, high);
    }
}

int partitionMiddle(vector<int>& arr, int low, int high) {
    int mid = low + (high - low) / 2;
    swap(arr[mid], arr[low]);
    return partitionFirst(arr, low, high);
}

void quickSortRandomPivot(vector<int>& arr, int low, int high) {
    if (low < high) {
```

```
        int pivotIndex = partitionRandom(arr, low, high);
        quickSortRandomPivot(arr, low, pivotIndex - 1);
        quickSortRandomPivot(arr, pivotIndex + 1, high);
    }
}

int partitionRandom(vector<int>& arr, int low, int high) {
    int randomIndex = low + rand() % (high - low + 1);
    swap(arr[randomIndex], arr[low]);
    return partitionFirst(arr, low, high);
}

void quickSortMedianPivot(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pivotIndex = partitionMedian(arr, low, high);
        quickSortMedianPivot(arr, low, pivotIndex - 1);
        quickSortMedianPivot(arr, pivotIndex + 1, high);
    }
}

int partitionMedian(vector<int>& arr, int low, int high) {
    int mid = low + (high - low) / 2;
    int pivot = median({ arr[low], arr[mid], arr[high] });
    if (pivot == arr[mid]) swap(arr[mid], arr[low]);
    else if (pivot == arr[high]) swap(arr[high], arr[low]);
```

```
    return partitionFirst(arr, low, high);
}

int median(vector<int> arr) {
    sort(arr.begin(), arr.end());
    return arr[1];
}

vector<int> generateRandomData(int size) {
    vector<int> data(size);
    for (int i = 0; i < size; i++) data[i] = rand() % 100000;
    return data;
}

vector<int> generateSortedData(int size) {
    vector<int> data(size);
    for (int i = 0; i < size; i++) data[i] = i;
    return data;
}

vector<int> generateReverseSortedData(int size) {
    vector<int> data(size);
    for (int i = 0; i < size; i++) data[i] = size - i;
    return data;
}
```

5. Test of the Completed Network

For testing, I ran the QuickSort algorithm on different datasets (sorted, random, and reverse sorted) of varying sizes (1000 to 5000 elements). The execution times were recorded for each combination of dataset type and pivot strategy and are summarized in the table above.

6.Concluding Thoughts

This project helped me better understand how pivot selection can significantly impact the performance of QuickSort. I found that the first element as pivot doesn’t perform as well for reverse sorted data, while the median pivot strategy worked best overall. This analysis has given me more insight into optimizing sorting algorithms, and I plan to explore even more pivot strategies in future projects.