Applied Algorithms 2020: Final Project

Emelie Elisabeth Louise Skörvald (emsk@itu.dk)
Nikol Ventsislavova Shakleva (nikv@itu.dk)
Szilvia Gaspar (szga@itu.dk)

December 2020

1 Part 1: Speeding up Binary Search using Lookup Tables

The following section contains implementation and performance test for two different binary search algorithms.

1.1 Task 1

Task one describes the implementation and construction of the two binary search algorithms. A Search interface containing search and pred method was implemented by both algorithms. Both algorithms are constructed to exclusively use 32 bit integers. The classic binary search is called on an array A, but for the sake of this project both classes should be instantiated and construct the array.

1.1.1 Binary Search

The standard Binary search algorithm used for this project uses R. Sedgewicks' implementation, with a few modifications. The original implementation returns the index position of A[i]. This version will also keep track of the closest array index for A[j] <A[i]. The closest array index of A[j] that will be returned if A[i] does not exist.

1.1.2 Binary Search with Tabulation

Binary search with tabulation speeds up the algorithm by scanning A and storing the index references of certain elements from A into a lookup table.
The lookup table is a matrix containing 2^k indexes each holding an array of size 2 that stores index references to elements in A.
The indexes of the lookup table are determined by the an integer's first k-bits. To find the integer's index the algorithm shift its first k-bits to 32 - k positions to the right.

```
 int shift = 32 - k;
int res = x >> shift;
```

To decide which range of numbers a specific index holds the algorithm calculates a min and a max value. At index max value is the number where all bits following the first k are set, while the min value are the first k bits followed by non-set bits.
After creating the main array A, the lookup table is made by scanning A from left to right. The index position of an A[i] is placed in a specific bucket as either the min or max index for this bucket.
By keeping track of the A[i] and A[i+1] index, the algorithm can check if the min and max top k-bits

are skipped and fill out the corresponding table entries with the right boundary at i. By making sure that the whole look-up table is filled out the search should be quicker. If a bucket contains more than one index the algorithm will perform a binary search on the sub array that is defined by min and max value in the look-up table.

1.2 Task 2

The following section will present and discuss the performance of the above algorithms with an experiment. The initial hypothesis was that the Binary search with Tabulation will perform better as the size of the array A increases.

1.2.1 Design

The Experiment consists of a producer, seed generator, timer, benchmark and experiment class.
The producer creates the input with four different input cases. The cases were created to examine the algorithms' weaknesses and strengths. For each array A of size N a query predictions of size 2N was also generated. The modes are:

- All integers in A end up in the same bucket.
- All numbers in A and the prediction are random.
- A contains small numbers while all the queries are larger integers
- Non of the queries are smaller than the items in A

The benchmark class is timing the performance by calling the timer class. For each test the benchmark executes a correctness test on the two algorithms by comparing the output results of the two algorithms.

The Experiment class defines the experiment and manages all parameters defined for the test. It calls the benchmark as shown in the pseudo code below.

```
Experiment run:
   for all modes
       for all sizes
           for all seed
               for all K
                   do 3 times
                       create input
                       execute benchmark
```

1.2.2 Execution

To prevent the compiler from optimizing the algorithms in the middle of the test a warm up is performed. The warm up is called by the Experiment class and calls the warm up as shown in the pseudo code below. The warm up is not measured, however the same correctness test as explained earlier is performed.

```
Warm-up run:
   for all modes
       for 2 sizes
           Execute 5.000 times
```

The Experiment contains 4 modes, 7 sizes of n, 4 seeds and 3 K. The benchmark did each benchmark 3 times. Each Benchmark runs one test case 50 times before finishing.

The experiment is executed automatically by calling the Experiment class. As the experiment runs, sub-results are printed in the terminal and once the experiment ends the results are saved in a file for each mode and algorithm.

1.2.3 Conclusion

The result clearly show that the Tabulation algorithm is considerable faster than the Binary Search for all the modes. this confirms the hypothesis of the experiment. The running time of the algorithms gets closer as the size N of the array increases.
It can be seen from the figure that both algorithms perform best with the non-existent input type. The worst performance is for input type random.

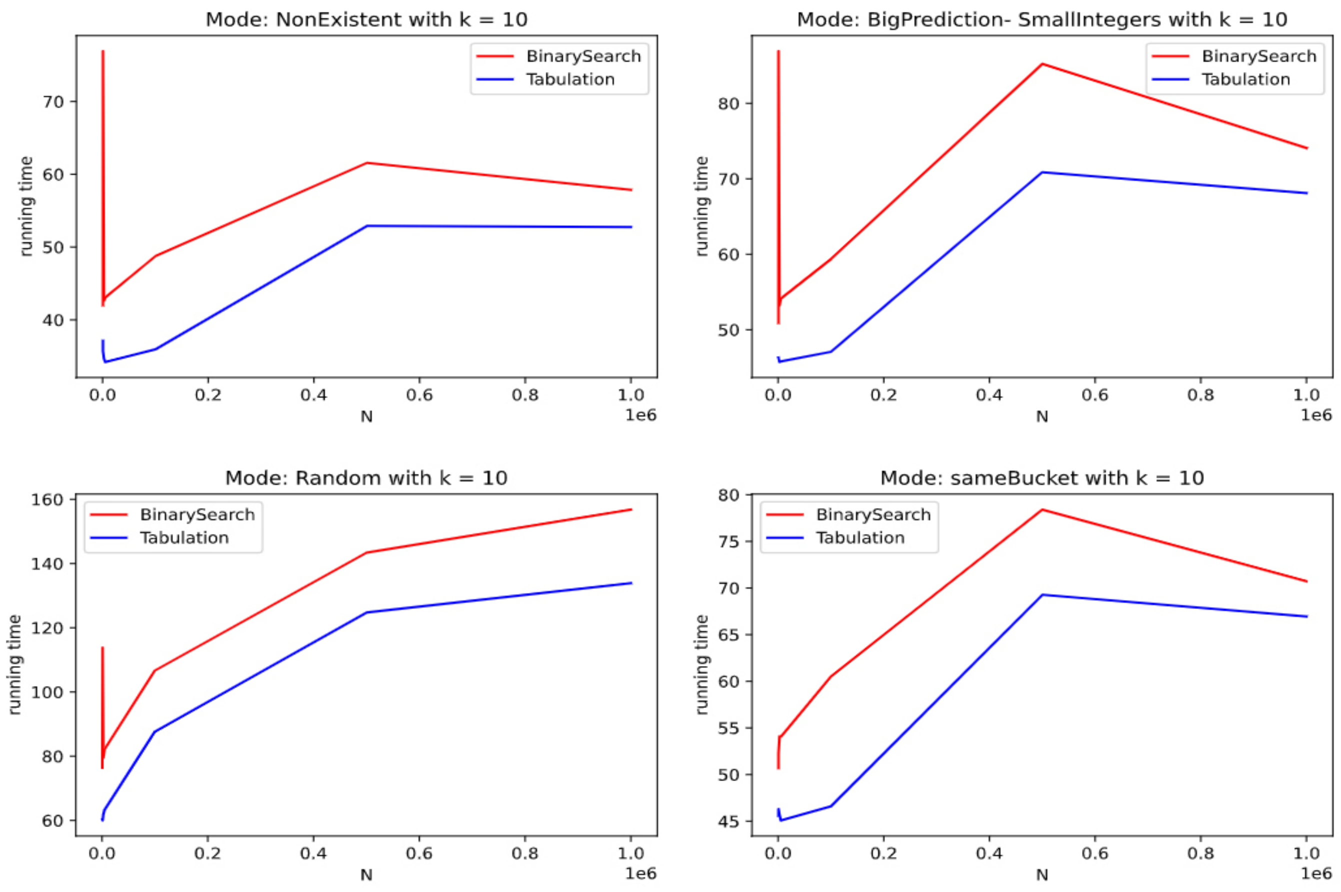

Figure 1: Result for each mode and algorithm

As shown in the graph below the size of K for the tabulation algorithms depends on the size N of the array A. As N grows the sub-arrays in the lookup table will increase. Looking at the graph the optimal K for the sizes N is 10, hence this value was used in the experiments.

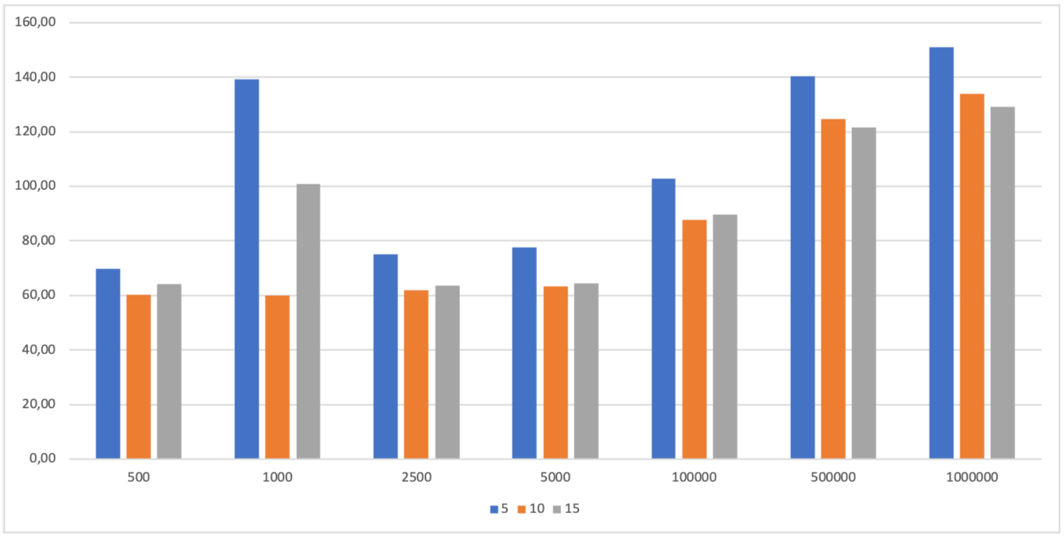

Figure 2: Relationship between the parameter k and the average query times for N

2 Part 2: Implementing Quicksort

This part implements and compares the performance of 3 different QuickSort algorithms. All the algorithms can be found in the folder called Part 2.

2.1 Task 3 Implementing the algorithms

An interface called Sort was created with two static methods swap and sort. These methods were implemented by all of the sorting algorithms. All of the algorithms were implemented based on the pseudo code from "Average Case Analysis of Java 7's Dual Pivot Quicksort" by Wild and Nebel [WN12], and "Multi-Pivot Quicksort: Theory and Experiments" from Kushagra et al. [K+13].

2.1.1 QuickSortClassic

This algorithm holds the invariant that A[left] to A[i] <p <A[i+1] to A[right].

2.1.2 DualPivotQuickSort

The DualPivotQuickSort functions almost the same way as QuickSortClassic, but using two partitioning elements p and q. The invariant here is that p <q has to be true at all times.

2.1.3 ThreePivotQuickSort

In the implementation of ThreePivotQuickSort the invariant p <q <r holds true for the partitioning elements.

2.2 Task 4 Correctness Test

2.2.1 Design and Execution

The Correctness test consists of a producer, seed generator and CorrectnessTest class. 6 different input types, 4 different seeds, 3 input sizes were used to determine the correctness of each algorithm. The correctness of the algorithms was determined by comparing the array A sorted by the quicksort implementations from Task 3 against the same array A sorted by the standard Java library sort implementation.

```
Correctness test run:
    for all modes
        for all seeds
            for all algorithms
                for all N
                    do  create input
                        compare algorithm sort against library sort
```

The 6 different input types were generated in the Producer.java class:

- Increasing: an input array sorted in increasing order
- Decreasing: an input array sorted in decreasing order
- Same: an input array where all the elements are the same
- Random: n random numbers generated with Java's Random class
- Semi-sorted: every second element in the generated input array is bigger than the previous one by 1
- Equal: Every One tenth elements in the array are the same

All the implementations of the Quick sort for all the input types passed the correctness test. Therefore, it can be concluded that all the implementations were correct.
The testing was run in an automatic fashion, by creating a Part2.java class from where the correctness test was run.

2.2.2 Boundary check

After removing the boundary check for DualPivotQuickSort, it failed the correctness test on input types random, equal, and semi-sorted. Which clearly shows that the implementation without the boundary check is not correct.
However, removing the boundary check for ThreePivotQuickSort did not fail any of the tests.

2.3 Task 5

2.3.1 Design

The Experiment consists of a producer, seed generator, timer, benchmark and experiment class.
The producer creates 6 different input types to test the average running time of the algorithms. These are the same as the input types in Task 4.
3 different input sizes were used: 20 000, 100 000 and 500 000. They were chosen according to the size of processor caches. Bigger input sizes were considered, but due to the limited time for running the experiments they were not included in the test. For each input type 4 different input arrays were generated based on different seeds.

The hypothesis before running the test was that the algorithms' running times will be the worst for input types increasing and decreasing. It is expected that the running times will be quadratic. For the increasing input type the algorithm has to perform a lot of swaps even though the elements are already sorted. For the decreasing input type the algorithm has to swap all the elements from the end of the array to the beginning. Moreover, the equal input type is also expected to perform poorly. The assumption was that the best case is going to be for random input type.

2.3.2 Execution

To make sure that the compiler does not optimize the algorithm's execution a warm-up was performed.

```
Warm-up run:
   for 2 modes
       for all algorithms
           for 1 sizes
                   do    create input
                     run correctness test
                       Execute 10.000 times
```

For each input type and each algorithm a total of 10 000 iterations were run.
After the warm-up was completed, the test was run with all 6 input types and 3 different input size (20 000, 100 000, 500 000). The test was run 5 times for each seed and with 30 iterations per benchmark run.

```
Experiment run:
   for all modes
       for all algorithm
           for all sizes
               for all seeds
               do    create input
                     run benchmark
                       Execute 30 times
```

The Experiment outputs the average mean and average standard deviation in nanoseconds for each input type counting the algorithm the input size and every seed. The output result was saved into a file for each mode.

2.3.3 Conclusion

It is evident from the graphs on Figure 3 that the Quicksort Classic algorithm is the slowest, whereas the ThreePivot Quick sort is the fastest. It can also be seen that the running time difference between Dual Pivot QuickSort and Three Pivot Quicksort is not that significant. The hypothesis of the experiment is being proven by the graphs. The worst case scenario is with the increasing input type, whereas the best performance is with a random input type.

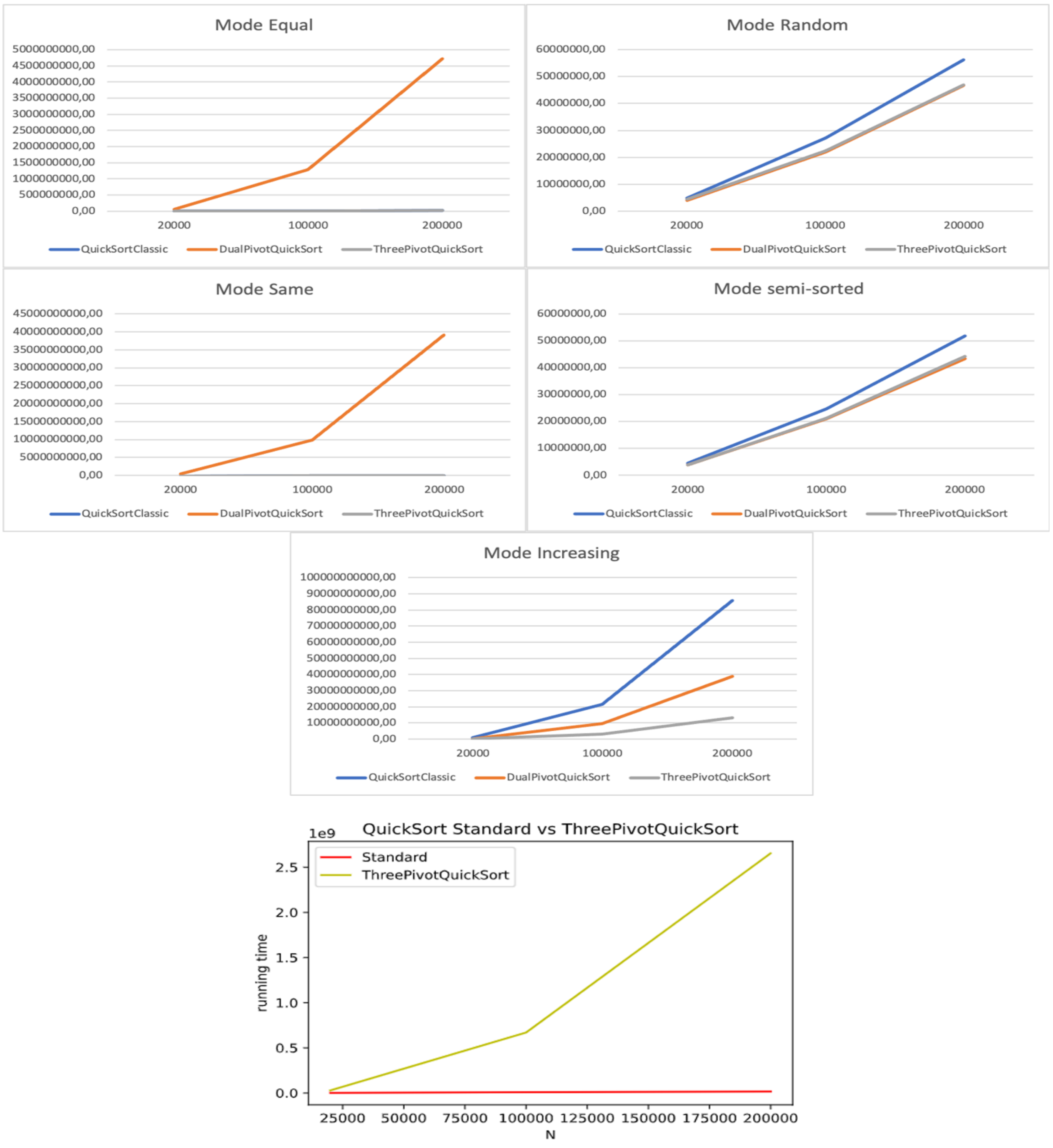

Figure 3:
Top: Performance of the QuickSort implementations
Bottom: Performance of Standard and ThreePivot QuickSort

2.3.4 InsertionSort

To optimize the running times of QuickSorts, insertion sort was implemented for small array sizes. A test was conducted to find the input size for array A where Insertion sort outperforms QuickSort.

InsertionSort was run against all 3 QuickSort implementations with small N ranging from 10 to 30. The results showed that InsertionSort is the fastest algorithm while N is smaller than 26 -28. See Figure below.

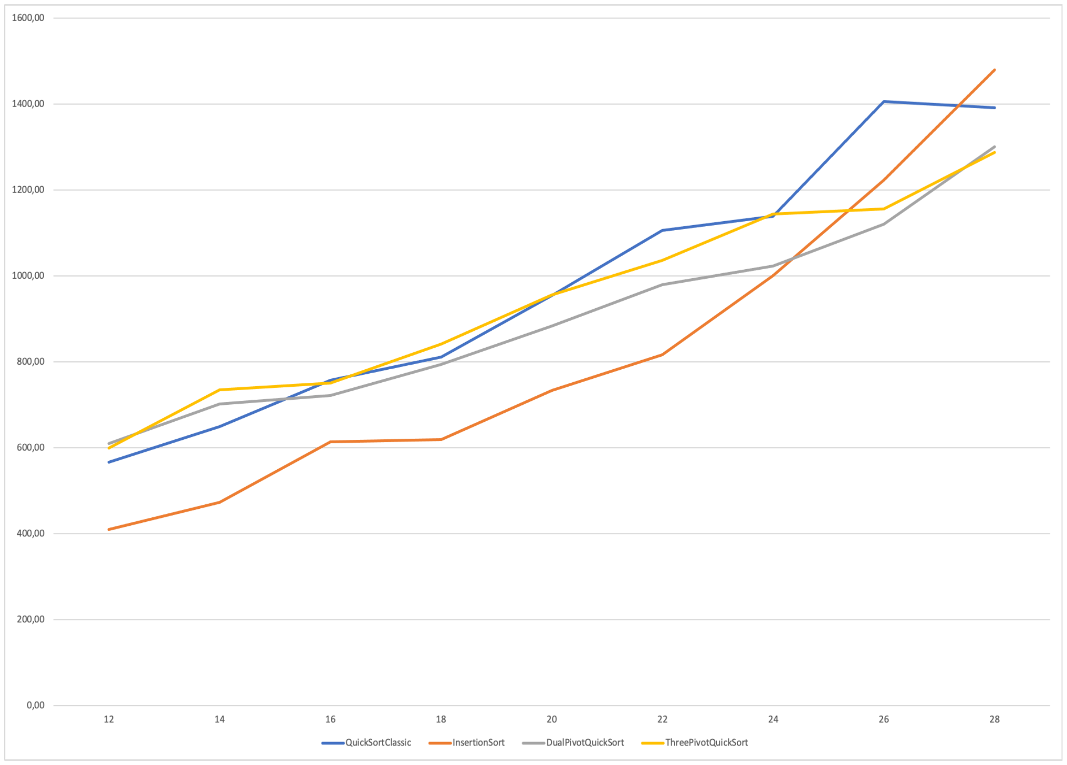

Figure 4: Finding optimal input size for insertionSort

2.3.5 Implementing insertionSort

The optimized versions of the QuickSorts, QuickSortClassicI.java, DualPivotQuickSortI.java and Three-PivotQuickSortI.java. can be found under folder Part2. The following condition was added to the algorithms: If the size of array A is smaller than 30, then Insertion Sort is called to sort the array (See the code snippet below).

```java
public static void sort(int[] A, int left, int right){
        int size = A.length;
        if (size < 30){
            InsertionSort.sort(A);
        }
        else {...}      // the respective QuickSort implementation
}
```

It was decided that all of the optimized algorithms will use the same array size for calling insertion. That was because the optimal values were very close to each other and not enough tests were run to determine the exact number for the different implementations.