https://doi.org/10.31449/inf.v48i4.4752 Informatica **48** (2024) 543–548 **543**

Empirical Evaluation of Algorithm Performance: Addressing Execution Time Measurement Challenges

Tomaž Dobravec
Faculty of Computer and Information Science
University of Ljubljana, Slovenia
E-mail: tomaz.dobravec@fri.uni-lj.si

Keywords: Empirical algorithm analysis, time measuring, accuracy, reliability, Java, C, x86 assembly

Received: March 21, 2023

In this paper, we investigate the influence of various factors, such as programming language, testing environment, and input data, on the accuracy of algorithm execution measurements. To conduct this study, we used the BubbleSort algorithm as a test case and implemented it in Java, C, and x86 assembly languages. We executed these implementations with various inputs and performed an empirical evaluation of the results using the ALGator system. We showed that the influence of the chosen programming language is negligible, since the Java and C implementations gave very similar results, while the assembler implementation differed only by a constant factor. Furthermore, our analysis emphasized the importance of repeating tests to obtain precise timing measurements - the more tests we do, the more accurate the measured result will be. We also discuss the impact of the input data type which can significantly affect the execution time due to the increased number of mispredictions of the branch predictor.

Povzetek: Narejena je empirična študija vpliva programskega jezika, ponavljanja testov in tipa vhodnih podatkov na točnost meritev časa izvajanja algoritmov.

1 Introduction

The analysis of algorithmic complexity is a crucial component of the algorithm design process [3]. This analysis is primarily concerned with estimating the amount of resources (such as time or memory usage) that an algorithm will require during its execution [10]. The outcome of this analysis is dependent upon the chosen computation model [4], which encompasses the execution environment and its limitations. Generally, the results of theoretical complexity analysis are used to differentiate between fast (i.e., polynomial) and slow (exponential) algorithms. However, the practical value of these results is limited, especially when two algorithms have the same (theoretical) time complexity. This is because the theoretical model used in the analysis does not take into account all the intricacies of the actual execution environment, such as memory caching, paging, or branch prediction [6, 7], which are only revealed during the execution of the algorithm on a real computer. Therefore, in order to make a practical comparison of algorithms, the theoretical analysis must be supplemented with empirical measurements of resource utilization during algorithm execution on various input data types [5, 8, 11].

To obtain accurate and reliable results, these measurements must be performed with great care, as numerous factors can impact the data being measured. This paper focuses on some of these factors and presents the results of our measurements that highlight their significance. Specifically, we employ three programming languages and demonstrate the impact of language selection on the speed of execution. Furthermore, we emphasize the importance of repeating tests, especially when the size of the input (and consequently, the execution time) is small. In addition, we discuss how the input data type can affect the rankings of algorithm quality. By thoroughly examining these factors and presenting our results, we hope to contribute to a more comprehensive understanding of the practical implications of algorithmic complexity analysis.

2 Testing environment setup

For all our tests in this research we will use the BubbleSort [1] algorithm for sorting arrays of integers. Since this is a very well-known and simple algorithm we are able to perform a precise theoretical analysis and provide a very accurate (theoretical) forecast for the time complexity of its implementations. The algorithm is so simple that we can count the number of operations performed during the execution for different inputs. Thus we will be able to compare theoretical predictions with the empirical results.

One of the goals of this research was to analyze the impact of the selected programming language on the efficiency of algorithm execution. Therefore we used three programming languages (namely Java, C, and the x86 assembler) to implement BubbleSort. Due to the simplicity of the algorithm, we managed to write the three implementations in such a way that they provide semantically identical code (see listings in Fig. 1). For further reference, we

named implementations `BubbleJ`, `BubbleC` and `BubbleA`, where the last letter denotes the programming language used (J for Java, C for C, and A for x86 assembler). Executing these implementations on the same inputs will result in an equal number of each programming-language-dependant atomic operations. Any differences in the execution speed will thus reflect the differences in the execution speed of these operations in the selected programming language. The C implementation was compiled with the `gcc` compiler in two ways: without optimization (the `-O0` flag) and with full optimization (the `-O3` flag). In this way, we got two distinct implementations (namely `BubbleC0` and `BubbleC3`). In the following, we will analyze the impact of this optimization on the speed of execution.

To facilitate the empirical evaluation in our research we used the ALGator system [2]. We used its tools to configure the Sorting project, to provide the test sets of input data and implementations, and to execute the algorithms' implementations in a controlled environment. For the execution machine, we used the Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz computer with 32GB RAM and with the Linux Ubuntu operating system installed.

The inputs for our algorithms consist of arrays of integers prearranged in three different orders: random order (RND), sorted order (SOR), and inversely sorted order (INV). These three distributions of input data are well manageable from a theoretical point of view since we know for all three the number of operations that will be performed during the sorting process. In all three case BubbleSort will perform exactly $n(n-1)/2$ comparisons, and $n^2/4$, 0, $n(n-1)/2$ swaps for RND, SOR, INV respectively. Note that all the numbers of operations are exact, except for the number of swaps in the RND case - here we only have the expected (instead of exact) number of swaps, since the sequence is randomly mixed. Since BubbleSort performs only comparisons and swaps (and some auxiliary increments of indices to maintain the loops) we could expect that, for example, sorting the RND array will be faster than sorting the INV array of the same size. But as we will see in the following this is not the case.

In the ALGator project inputs (i.e. test cases) are grouped into test sets. Each test case has its own identifier (Test ID), so the results can also be compared on a test basis. To provide accurate results each test case is executed several times (each execution of the test case has its identifier, Repetition ID). Besides a list of all execution times of a test case, ALGator provides two pieces of information, the time of the first execution (Tfirst) and the time of the fastest execution (Tmin) of this test case. The first execution is usually much slower than other executions - as we will see in the following the Tfirst time can even be twice as big as the Tmin time. This behavior is more noticeable in the java environment since the JVM needs to warm up before it can operate at full speed [9].

To measure the time in Java we can only use the wall clock (Java does not provide any processor usage information). To minimize the unreliability of the measured time (which is due to the fact that the process may spend time waiting for I/O or for other processes that are also using the CPU) we use a "clean" computer which is dedicated only to the execution of the algorithms. Besides that, we usually take the Tmin time as reference data, since this is a time in which the computer is capable of solving the problem (the number of disturbing factors is minimal). For the algorithms implemented in the C programming language, we use the CPU time obtained by the `clock()` function (which returns the number of clock ticks used by the process). By calling this function before and after the algo-

```
void bubbleJ(int data[]) {
  int n = data.length;
  for (int i=0; i<n-1; i++) {
    for (int j=0; j<n-i-1; j++) {
      int a = data[j];
      int b = data[j+1];
      if (a > b) {
        data[j]   = b;
        data[j+1] = a;
      }
    }
  }
}
```

(a) BubbleJ (Java)

```
void bubbleC(int data[], int n) {

  for (int i=0; i<n-1; i++) {
    for (int j=0; j<n-i-1; j++){
      int a = data[j];
      int b = data[j+1];
      if (a > b) {
        data[j]   = b;
        data[j+1] = a;
      }
    }
  }
}
```

(b) BubbleC (C)

```
;esi=@data, ecx=n
    xor eax, eax
loop1:
    cmp eax, ecx
    je  lend
    mov ebx, 1
    sub ecx, eax
loop2:
    cmp ebx, ecx
    je  eloop1
    mov edx, [esi + 4*ebx-4]
    mov edi, [esi + 4*ebx]
    cmp edx, edi
    jle eloop2
    mov [esi + 4*ebx-4],edi
    mov [esi + 4*ebx],  edx
eloop2:
    inc ebx
    jmp loop2
eloop1:
    add ecx, eax
    inc eax
    jmp loop1
lend:
```

(c) BubbleA (assembler x86)

Figure 1: The tree implementations of the BubbleSort algorithm

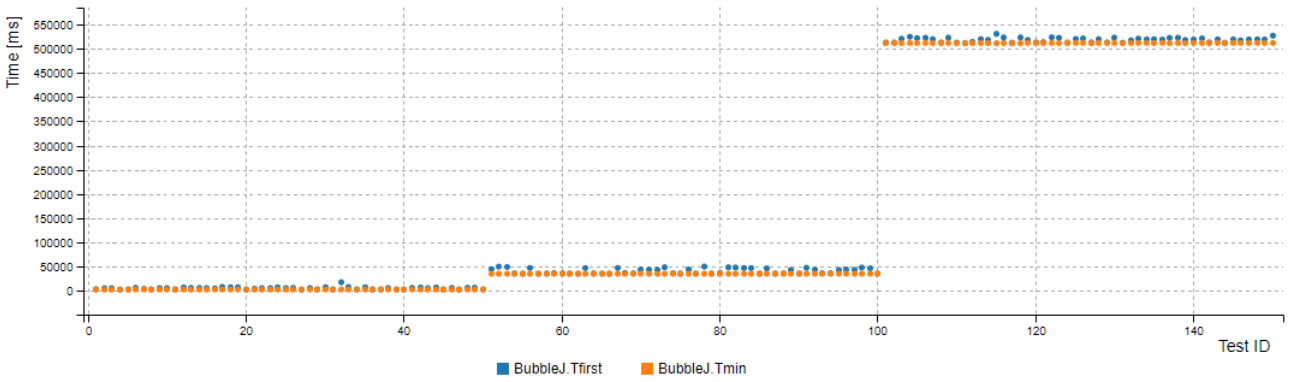

Figure 2: The Tmin and Tfirst measured times for `BubbleJ` implementation on inputs of size n=500, n=5000 and n=20000.

rithm execution and subtracting the returned values we get the total amount of time a process has actively used a CPU. The time measured this way is a much more reliable and accurate quality indicator.

3 The meaning of test repetition

In our first experiment, we would like to find out the meaning of several repetitions of a given test case execution. For this, we used a test set consisting of three groups of test cases: in each group, there are 50 identical tests of sizes 500, 5000, and 20000. All the input arrays in these test cases were ordered in inverse order (to ensure an identical number of operations during the sorting process). We executed each test case 50 times.

The graph in Fig. 2 depicts the times Tfirst and Tmin for all 150 tests. The Tfirst times (blue dots) in this graph are a little bit bigger than the Tmin times.

By analyzing the results we noticed that the (absolute) difference between Tfirst and Tmin is approximately the same for all three groups of test cases. The relative difference is therefore smaller for bigger measured times. We can conclude that the measurement of both Tfirst and Tmin is important for small inputs and that the importance of distinguishing between Tfirst and Tmin decreases with increasing input size. Measurements have shown that something similar to Java's "Tfirst phenomenon" also happens with C, except that in this case "warming up the machine" adds significantly less to the overall time complexity, so the differences in speed between Tfirst and Tmin are noticeable only in experiments that take very little time. From Table 1, which shows the relationship between the average first and the minimum execution time of a test case, $f = \frac{\overline{T}_{first}}{T_{min}}$ it can be seen that for small n the ratio is similar in both implementations, but for larger n the difference between Tfirst and Tmin is almost negligible for the `BubbleC3`, while for the `BubbleJ` the value decreases significantly more slowly. At $n = 20000$ the difference is still more than 5%.

The difference in measured times of multiple executions of the `BubbleJ` and `BubbleC3` implementations is depicted in Fig. 3. Here we used 50 inversely ordered arrays of size 5000, each test case was repeated 50 times. On the graph, the time of the first execution of the test case is shown in gray (Tfirst, Repetition ID=0), the first 20 repetitions are shown in orange and the next 30 in red.

With `BubbleJ`, we see that the first times (Tfirst) deviate considerably from the other measured times; the Tfirst times are somewhere between 18k and 22k, and the other times are much smaller (between 12k and 14k), which corresponds to the factor of 1.4 from Table 1. Other measured times on this graph do not show much fluctuation, as the scale of the display is reduced due to the large Tfirst times; we see that some Tfirst times are almost 100% larger than the smallest measured times. With `BubbleC3`, all times are quite similar to each other; the graph shows some variations, but everything is between 14.6k and 15.6k; the differences between measured times are relatively small (approx. 6%).

In conclusion: is it important to repeat the algorithm execution for several times to find the minimum time? As the measurements show, the answer depends on the size of the input - the smaller the input, the more measurements are unreliable, so we need to take more measurements to get a good result.

Bar charts in Fig. 4 depict the proportion of measurements that differ from the smallest measurement by the given percentage range. The measurements on small inputs for the `BubbleJ` vary a lot. More than 36% of all measurements differ from the minimal time of more than 10%. For the `BubbleC3` on the other hand only 17% of the measurements are that bad. When increasing the size of the input the results for both algorithms improve. For n=20000, for example, more than 73% (98%) of measurements differ from the minimal measurements for less than 1% for `BubbleJ`

N	*BubbleJ*	*BubbleC3*
500	1.98	1.80
5000	1.40	1.02
20000	1.05	1.01

Table 1: The ratio $f = \frac{\overline{T}_{first}}{T_{min}}$ between the average of the first and the minimal measured times

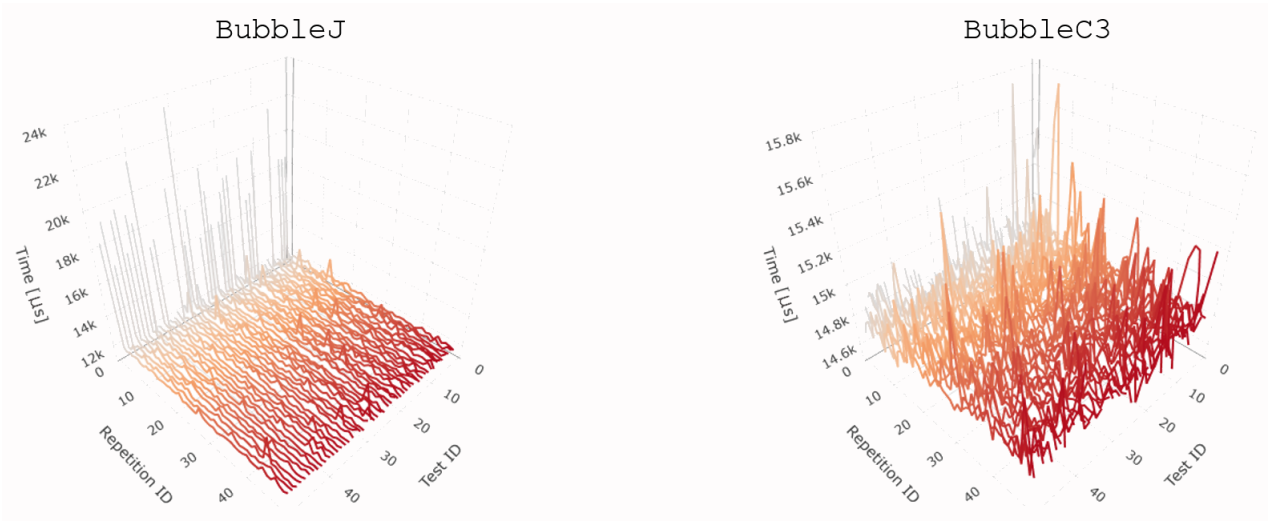

Figure 3: Times of execution of 50 identical test cases (50 repetitions of each test case) with `BubbleJ` and `BubbleC3`

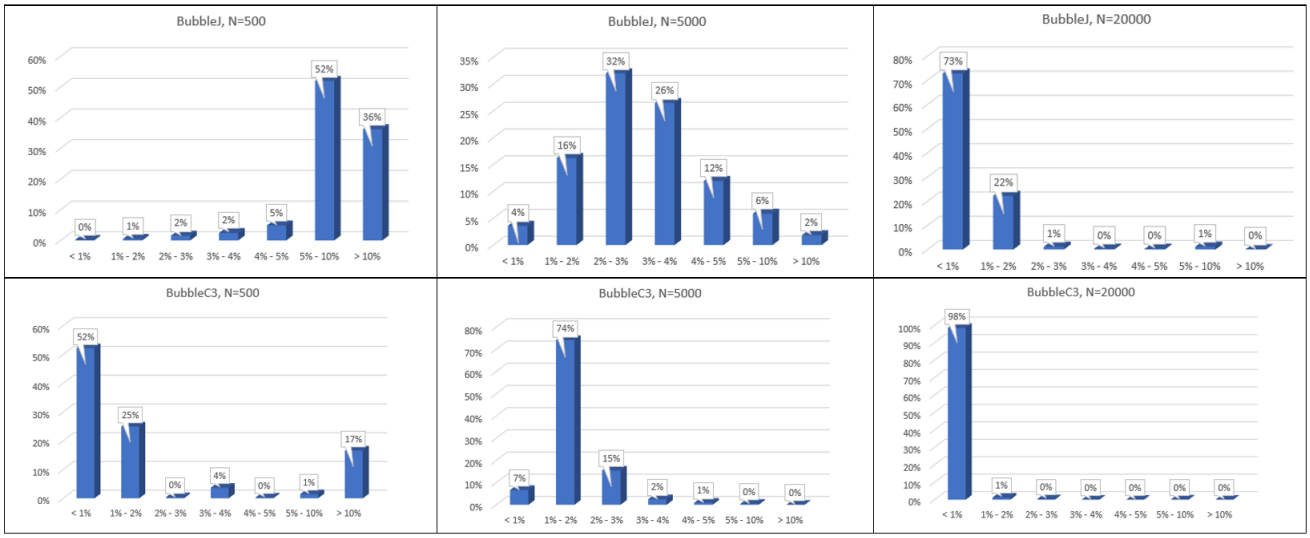

Figure 4: The proportion of measurements that differ from the smallest measurement by the given percentage range.

(`BubbleC3`) implementation.

The relative standard deviations of all measured times for `BubbleJ` are 21%, 7%, and 1% for n=500, 5000, and 20.000 respectively. This confirms the claim that as the size of the input increases, the importance of multiple tests decreases. Since the relative standard deviations are even smaller for `BubbleC3` (namely 15%, 1%, 0.24%), the importance of a large number of measurements is even smaller here.

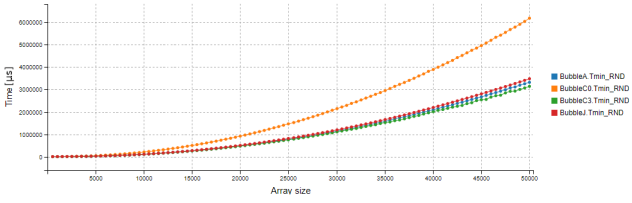

Figure 5: Tmin, RND data, $n = 500, ..., 50000$

4 The impact of the programming language

We compared the times of execution of four implementations (`BubbleJ`, `BubbleA`, `BubbleC0` and `BubbleC3`) on randomly ordered sequences (RND) of length 500 to 50000 (step 500). Each test was executed 30 times. Fig. 5 shows the minimum measured times Tmin of all four algorithms.

We expected the `BubbleC3` to be the best, which was also proven with the measurements. The difference between `BubbleC0` and `BubbleC3` is somewhat surprising. Since BubbleSort is a simple algorithm, one would expect that the speedup resulting from the optimization would not be that

great. But this is not the case, the difference is almost 2 times for large n. An interesting observation is that the factor of 2 appears to approximate the ratio between the sizes of machine code produced by optimized and non-optimized compilation. Specifically, the former contains 28 machine code instructions, while the latter contains 53.

The relationship between `BubbleJ` and `BubbleA` is interesting. In a battle between fast implementations, Java turned out to be the slowest, although the differences in speed are not so great. Fitting all measurements with quadratic functions results in the following:

`BubbleC0`: $T_{min}(n) = 2.438n^2\ \mu s$
`BubbleJ`: $T_{min}(n) = 1.372n^2\ \mu s$
`BubbleA`: $T_{min}(n) = 1.311n^2\ \mu s$
`BubbleC3`: $T_{min}(n) = 1.246n^2\ \mu s$

The ratio between the best (`BubbleC3`) and the worst (`BubbleC0`) implementation is 1 : 1.956, which we also noticed from the graph. More interesting is the ratio between the optimized C3 and Java implementation: `BubbleC3` : `BubbleJ` = 1 : 1,101. This means that for sorting random sequences Java is 10% slower than C. To find out, how good this conclusion is, let us calculate and depict the relative error

$$Error = \frac{|BubbleC3.T_{min} - 1.1 * BubbleJ.T_{min}|}{BubbleJ.T_{min}}$$

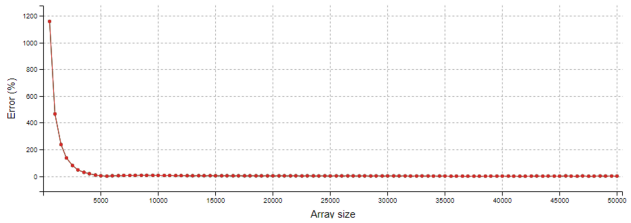

Figure 6: Relative error of estimating `BubbleJ`.Tmin with 1.1*`BubbleC3`.Tmin

Fig. 6 shows that for small inputs ($n < 5000$) the error is very big (as big as 1200%), but for larger inputs ($n > 10000$) the error is always less than 5% and it seems that it decreases when n increases.

5 The impact of the input data type

The input for a sorting algorithm is not always a randomly ordered array - sometimes the input data is already partially sorted. To find out if the (partial) pre-ordering of data impacts the execution time we used two special cases of the input data type - already sorted (SOR) and inversely sorted (INV) data. The results for both types show similar trends. Fig. 7 depicts the results on inversely ordered sequences (INV) of length 500 to 50,000 (step 500). The quality ranking of algorithms when sorting INV data changes compared to the ranking on RND data (Figures 5 and 7). While `BubbleC0` remains the worst implementation, in the first place there is a swap - `BubbleC3` gives way to `BubbleA` and `BubbleJ`. Something similar happens with the sorted (SOR) data. This change in ranking is hard to explain, but according to the research results presented in the following, we could speculate that the code generated by JVM is less suitable for branch prediction: with INV and SOR data the branch predictor is always correct, which could reflect better performance. Anyway, the results unequivocally show that the type of input has a great impact on the quality of implementation. While with random data `BubbleC3` implementation was faster than `BubbleJ`, for inversely ordered and already ordered data the Java implementation is the fastest.

The importance of data type is demonstrated by the results of the following experiment in which we ran the same algorithm (BubbleJ) on three different types of data: randomly ordered (RND), reverse-ordered (INV), and already sorted (SOR) data. With this experiment, we compared the number of swaps needed to sort an array with the time of execution and we discovered a strange behavior that can be explained only by the presence of the processor's branch predictor. The BubbleJ algorithm is a simple algorithm composed of three parts: loop administration, data comparison, and data exchange (data swap). Loop administration takes the same amount of time regardless of the input data type. Likewise, the input data type does not affect the number of comparisons performed by the algorithm, which is always precisely $n * (n-1)/2$. The input data type only affects the number of swaps performed, which is exactly $n * (n-1)/2$ in the case of INV data, 0 in the case of SOR data, and approximately $n^2/2$ in the case of RND data. The exact number of swaps in our test is shown in the left graph in Figure 8. Since the number of swaps is the only variable quantity during algorithm execution (the number of all the other operations is the same for all input data types), one would expect that graphs depicting the experimental time complexity for these three data types would be similar, but this is not the case. The right graph in Figure 8 shows that running the algorithm on randomly sorted data is much slower than running it on reverse-ordered data, even though the algorithm in the former case performs fewer operations than in the latter case. The only reasonable explanation for this phenomenon is in the influence of the processor's branch prediction mechanism, which optimizes data preparation for the processor and enables faster execution. This mechanism is particularly effective when successfully predicting the future, which it apparently does

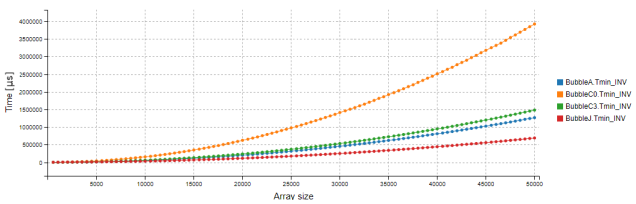

Figure 7: Tmin, INV data, $n = 500, ..., 50000$

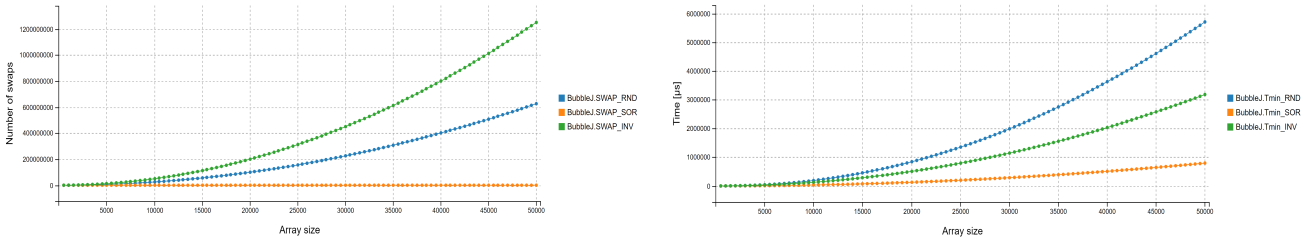

Figure 8: The number of swaps and the time of execution of BubbleJ algorithm on three different types of data: randomly sorted, already sorted and inversely sorted data.

well in the case of reverse-ordered data - because a swap follows each comparison, the branch predictor makes fewer mistakes than when swaps occur only occasionally. Properly prepared data makes the processor's work easier, resulting in a shorter total execution time, even though more operations are performed. This result clearly shows that the execution time of an algorithm is also influenced by factors that are not usually considered in theoretical analysis, which leads to significantly different results between theoretically predicted and empirically measured times.

6 Conclusions

The findings of this research paper demonstrate that the performance of algorithms is affected by numerous factors. Despite our efforts to maintain a controlled environment, we observed variations in our measurements. These deviations were particularly noticeable in Java, where the measurement of time is more sensitive to environmental influences than in C. We discovered that repeated execution of algorithms is particularly important for small inputs. Furthermore, we compared the performance of algorithms implemented in different programming languages. Our findings revealed that the difference between Java and C is not significant and that it depends on the type of input data. For randomly sorted arrays, the C implementation outperformed Java, while for inversely ordered and already sorted data, Java was superior.

In the future, we could apply similar methods to investigate other problems and determine if these results can be generalized. Additionally, we could explore the use of other popular programming languages such as Python, and examine in detail the real impact of the branch predictor on the final results. By conducting further research in this field, we can gain a better understanding of the factors that impact algorithm performance, and ultimately improve the efficiency and effectiveness of computer programs.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[2] T. Dobravec. Algator — an automatic algorithm evaluation system. *Advances in Computers*, 116(1):65–131, 2019. https://doi.org/10.1016/bs.adcom.2019.07.002.

[3] T. Dobravec. Exact time measuring challenges. In *Proceedings of the 25th International Multiconference Information Society, IS MATCOS*, volume I, pages 21–24, Koper, 13-14 October 2022.

[4] M. Fernández. *Models of Computation, An Introduction to Computability Theory*. Springer, 2009. https://doi.org/10.1007/978-1-84882-434-8.

[5] D. Johnson. A theoretician's guide to the experimental analysis of algorithms. 12 2001. https://doi.org/10.1090/dimacs/059/11.

[6] R. Kumar. *Instruction Level Parallelism: Branch Prediction and Optimization*. LAP LAMBERT Academic Publishing, 2012.

[7] C. C. McGeoch. Experimental methods for algorithm analysis. *Encyclopedia of Algorithms*, 2008. https://doi.org/10.1007/978-0-387-30162-4_135.

[8] B. Moret. Towards a discipline of experimental algorithmics. *Monograph in Discrete Mathematics and Theoretical Computer Science*, 2002. https://doi.org/10.1090/dimacs/059/10.

[9] M. Price. Hot code is faster code - addressing jvm warm-up. *QCon*, April 2016.

[10] B. Swathi. A comparative study and analysis on the performance of the algorithms. *International Journal of Computer Science and Mobile Computing*, 5(1):91–95, Januar 2016. ijcsmc.com/docs/papers/January2016/V5I1201621.pdf.

[11] M. Tedre and N. Moisseinen. Experiments in computing: A survey. *The Scientific World Journal*, (1):1–11, 2014. https://doi.org/10.1155/2014/549398.