Project 0: Ancient Multiplications Efficiency

Binjie Liang, *2023064642*, bliang@estudiantec.cr and Esteban Secaida, *2019042589*, esecaida@estudiantec.cr

***Abstract*—The quest for efficient algorithms has led to revisiting historical methods like Russian Peasant multiplication, which uses a binary-like process of doubling and halving. Despite its ancient origins, its performance relative to conventional multiplication algorithms remains underexplored. This paper investigates the implementation and computational efficiency of ancient multiplications methods compared to standard methods.**

***Index Terms*—Bits, Binary, Least Significant Bit**

I. Introduction

Addition, subtraction, multiplication, and division—these four operations form the basic arithmetic of mathematics, serving as the foundation for more advanced mathematical concepts and applications. Among these, multiplication is particularly crucial, over time, various multiplication techniques have emerged, each with unique historical and cultural significance. One such method, known as *Russian Peasant Multiplication* [1], finds its roots in ancient *Egyptian mathematics* [2] and employs a binary-like process of iterative doubling and halving.

The primary objective of this research is to analyze the computational efficiency of the Russian Peasant Multiplication algorithm. By implementing both this algorithm and the standard multiplication function in a controlled environment, we aim to measure and compare their execution times. This comparison will shed light on the practical implications of utilizing historical algorithms in contemporary computing, particularly in scenarios where computational efficiency is paramount.

II. Development of the Russian Peasant Multiplication Algorithm

A. What's Russian Peasant Method?

In the present day, most people learn multiplication using standard methods, such as the long multiplication algorithm (example in Fig.1), which involves multiplying digits and summing intermediate products. However, historical methods like the Russian Peasant Multiplication offer an alternative approach with a unique process based on iterative doubling and halving. This ancient technique, originating from ancient Egyptian mathematics, leverages a binary-like procedure that can be quite effective.

The Russian Peasant Multiplication algorithm operates by repeatedly halving one of the multiplicands and doubling the other until the former reaches zero. At each step, if the halved value is odd, the corresponding doubled value is added to the result.

Fig. 1. An example of the standard "long multiplication".

B. How this method works?

Given the numbers `A` and `B`

1) Initialize `result` as 0.
2) While `B` is not equal to 0:
    a) If `B` is odd, add `A` to `result`.
    b) Double `A`.
    c) Divide `B` by 2.
3) At the end, `result` will contain the product of `A` and `B`.

This approach not only highlights the algorithm's simplicity but also its potential efficiency in certain computational contexts. By exploring this historical method, we can gain insights into its practical applications and performance compared to modern multiplication algorithms.

In the following figures, we will demonstrate a short example of how to perform the Russian Peasant Multiplication algorithm.

Fig. 2. Initially, we have numbers A and B.

Initially, as shown in Fig. 2, we have number A (13) and number B (24). The result is initialized to 0.

The next step is to check if B is greater than 0. If it is, we check if B is an odd number. If B is odd, we add A to the result. Then, we double the value of A and halve the value of B. This process repeats until B equals 0 when rounded down.

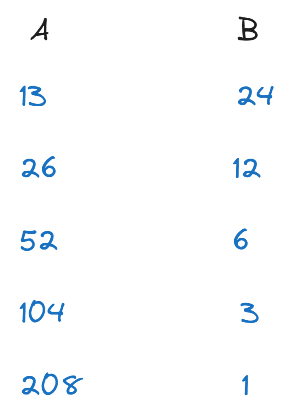


Fig. 3. Doubling A and halving B.

As shown in Fig. 3, after each step, we double the value of A and halve the value of B.

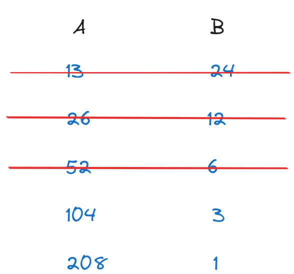


Fig. 4. Ruling out the rows where B is even.

After repeating the process, we rule out the rows where B is even, as illustrated in Fig. 4.

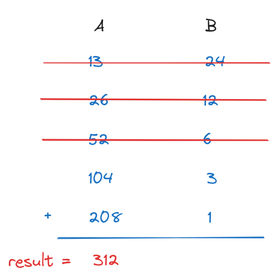


Fig. 5. Result of the product between A and B.

Finally, as shown in Fig. 5, we sum all the remaining values in the A column. This sum gives us the result of the product of A and B.

C. Relationship with binary system

You may be curious about the connection between this method and the binary system. The Russian Peasant Multiplication algorithm inherently utilizes the principles of the binary number system through its process of iterative doubling and halving.

In this method, when A is doubled, in binary it represents a left shift. While on the other hand when we halve B, we are actually doing a right shift in bits. To know if our number in B is odd, we just need to make a AND operation with the least significant bit, if the LSB is equal to 1, then is 1.

For a better understanding, we will revisit the example previously shown, but using binary representation.

Fig. 6. Figure 2 numbers in binary representation

Converting 13 and 24 into binary, we get 1101 and 11000, respectively. The next step involves performing a left shift on A and a right shift on B.

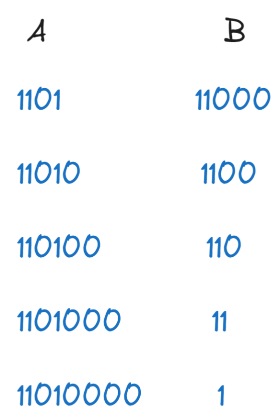


Fig. 7. Doubling and halving using left and right shifts

After shifting both numbers, the next step is to exclude the rows where the least significant bit (LSB) of B is 0.

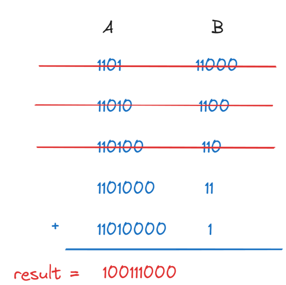


Fig. 8. Binary result after excluding even numbers

By excluding the rows where B's LSB is 0, the remaining task is to sum all the remaining numbers in the A column. This results in a binary value of 100111000, which is the final product.

Check out that the process is exactly the same, demonstrating that the Russian Peasant Multiplication algorithm leverages the principles of binary arithmetic, further underscoring its efficiency and relevance in computational contexts.

III. Hands-On Application

Algorithm efficiency is key in computer science. As we tackle more complex problems, evaluating even basic operations like multiplication becomes crucial. Efficiency isn't just about speed, it also involves memory use and how well methods scale. We'll examine two multiplication approaches: the conventional method and Russian Peasant Multiplication, comparing their performance across various metrics.

The standard multiplication method, taught in schools and widely used, relies on basic arithmetic. It's fast and efficient, especially on modern processors. However, techniques like Russian Peasant Multiplication offer interesting alternatives. This method, based on binary principles, might be more efficient in certain computational contexts.

1) Methodology: To compare these methods, we set up a thorough experiment. We use eight random numbers to create a 4x4 table, covering various multiplication scenarios. Each operation is repeated N times to get accurate average execution times. We also swap the order of multiplication (a×b and b×a) to check if it affects performance.

This comparison aims to reveal the pros and cons of each method, offering insights into their practical uses by analyzing the timestamps of their performances.

IV. Empty Version

To better understand the project before analyzing the results, it is important to consider why there is an empty return version of the multiplication algorithm.

A. What's the purpose of the empty method?

Beyond being a requirement, it serves as a benchmark to determine the minimal time it takes for a function to receive two arguments and return a result. By using the empty version, which returns 0, we can establish the quickest time for a function to receive two numbers and produce an output. This benchmark allows us to compare it with the time taken to return the result of a multiplication using both the normal operation and the ancient algorithm.

After understanding the methodology and why are we using an empty version, it is time to run the program and see the timestamps of the different versions of the algorithms so we can go on and see which is the fastest and answer why by analyzing the results.

B. Empty Version in C

First, we have the results of the empty version. We have timestamps that range from 30.051ms ms in the 40 * 605 case to 37.815ms in the 424 * 798 case for the individual times, with the function being called 50000 times per operation.

Fig.9 version presents a time of 491.896 ms in the A * B order using C language.

And Fig.10 presents a time of 522.372 ms in the B * A order using C language.

The total time (including the prints and variables initialization from both tables) for this version is 2027.123 ms.

```
N: 50000
nullMul(A * B)
         |                           38|                           40|                          424|                          585|
590      | 590 *  38 =    0 / 30.152000 ms | 590 *  40 =    0 / 30.324000 ms | 590 * 424 =    0 / 31.004000 ms | 590 * 585 =    0 / 30.061000 ms |
767      | 767 *  38 =    0 / 30.213000 ms | 767 *  40 =    0 / 30.132000 ms | 767 * 424 =    0 / 30.061000 ms | 767 * 585 =    0 / 30.124000 ms |
798      | 798 *  38 =    0 / 30.111000 ms | 798 *  40 =    0 / 30.227000 ms | 798 * 424 =    0 / 34.240000 ms | 798 * 585 =    0 / 34.141000 ms |
605      | 605 *  38 =    0 / 30.154000 ms | 605 *  40 =    0 / 30.163000 ms | 605 * 424 =    0 / 30.609000 ms | 605 * 585 =    0 / 30.180000 ms |

Execution time multiplying A * B returning 0 in C: 491.896 ms
```

Fig. 9. Null multiplication between A and B using C

```
nullMul(B * A)
         |                          590|                          767|                          798|                          605|
38       |  38 * 590 =    0 / 30.186000 ms |  38 * 767 =    0 / 30.433000 ms |  38 * 798 =    0 / 30.897000 ms |  38 * 605 =    0 / 30.083000 ms |
40       |  40 * 590 =    0 / 34.794000 ms |  40 * 767 =    0 / 30.260000 ms |  40 * 798 =    0 / 30.130000 ms |  40 * 605 =    0 / 30.051000 ms |
424      | 424 * 590 =    0 / 32.696000 ms | 424 * 767 =    0 / 30.193000 ms | 424 * 798 =    0 / 37.815000 ms | 424 * 605 =    0 / 33.949000 ms |
585      | 585 * 590 =    0 / 30.175000 ms | 585 * 767 =    0 / 37.485000 ms | 585 * 798 =    0 / 36.108000 ms | 585 * 605 =    0 / 37.117000 ms |

Execution time multiplying B * A returning 0 in C: 522.372 ms

Execution time taken from both tables: 2027.123 ms
```

Fig. 10. Null multiplication between B and A using C

C. Empty version in Assembly

Using NASM Assembler, we implement an empty version of the multiplication function. This function is called from the main C program. In terms of performance, using Assembly didn't result in a significant difference compared to the C implementation.

```
nullMulASM(A * B)
         |                           38|                           40|                          424|                          585|
590      | 590 *  38 =    0 / 33.151000 ms | 590 *  40 =    0 / 32.708000 ms | 590 * 424 =    0 / 30.467000 ms | 590 * 585 =    0 / 30.241000 ms |
767      | 767 *  38 =    0 / 30.295000 ms | 767 *  40 =    0 / 30.364000 ms | 767 * 424 =    0 / 31.459000 ms | 767 * 585 =    0 / 30.613000 ms |
798      | 798 *  38 =    0 / 30.307000 ms | 798 *  40 =    0 / 30.669000 ms | 798 * 424 =    0 / 30.374000 ms | 798 * 585 =    0 / 30.322000 ms |
605      | 605 *  38 =    0 / 30.874000 ms | 605 *  40 =    0 / 30.475000 ms | 605 * 424 =    0 / 30.324000 ms | 605 * 585 =    0 / 30.427000 ms |

Execution time multiplying A * B returning 0 in ASM: 493.070 ms
```

Fig. 11. Null multiplication between A and B using ASM

Comparing the results of Fig. 9 and Fig. 11, both representing functions that return 0 for multiplications between A and B, we observe only a marginal difference in performance. The Assembly version executed all multiplications merely 2ms faster than its C counterpart.

While in the other-hand in Fig.12, multiplying B and A in Assembly returning 0, was 34ms faster than C, but these results are not always present, this minimal performance difference suggests that for simple operations like our null multiplication, the overhead of function calls and basic register operations dominates the execution time, leaving little room for optimization through assembly programming.

V. Standard Version

A. Standard version in C

Next, we have the standard version, with individual times that range from 30.009ms in the 590 * 585 case to 38.348ms in the 40 * 590 case. These times reflect the duration of performing the operation also 50000 times.

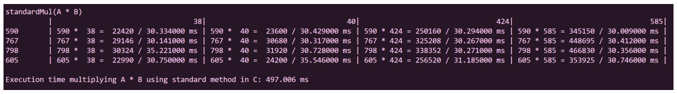
Fig.13 version presents a time of 497ms in the A * B order.

And Fig.14 presents a time of 517ms in the B * A order.

The total time for this version is 2020.89ms.

```
nullMulASM(A * B)
         |                           38|                           40|                          424|                          585|
590      | 590 *  38 =    0 / 33.151000 ms | 590 *  40 =    0 / 32.708000 ms | 590 * 424 =    0 / 30.467000 ms | 590 * 585 =    0 / 30.241000 ms |
767      | 767 *  38 =    0 / 30.295000 ms | 767 *  40 =    0 / 30.364000 ms | 767 * 424 =    0 / 31.459000 ms | 767 * 585 =    0 / 30.613000 ms |
798      | 798 *  38 =    0 / 30.307000 ms | 798 *  40 =    0 / 30.669000 ms | 798 * 424 =    0 / 30.374000 ms | 798 * 585 =    0 / 30.322000 ms |
605      | 605 *  38 =    0 / 30.874000 ms | 605 *  40 =    0 / 30.475000 ms | 605 * 424 =    0 / 30.324000 ms | 605 * 585 =    0 / 30.427000 ms |

Execution time multiplying A * B returning 0 in ASM: 493.070 ms

nullMulASM(B * A)
         |                          590|                          767|                          798|                          605|
38       |  38 * 590 =    0 / 30.249000 ms |  38 * 767 =    0 / 31.011000 ms |  38 * 798 =    0 / 30.245000 ms |  38 * 605 =    0 / 32.433000 ms |
40       |  40 * 590 =    0 / 32.593000 ms |  40 * 767 =    0 / 30.231000 ms |  40 * 798 =    0 / 30.217000 ms |  40 * 605 =    0 / 30.056000 ms |
424      | 424 * 590 =    0 / 30.140000 ms | 424 * 767 =    0 / 30.320000 ms | 424 * 798 =    0 / 30.224000 ms | 424 * 605 =    0 / 30.044000 ms |
585      | 585 * 590 =    0 / 30.180000 ms | 585 * 767 =    0 / 30.298000 ms | 585 * 798 =    0 / 30.229000 ms | 585 * 605 =    0 / 30.032000 ms |

Execution time multiplying B * A returning 0 in ASM: 488.502 ms

Execution time taken from both tables: 1960.887 ms
```

Fig. 12. Null multiplication between B and A using ASM

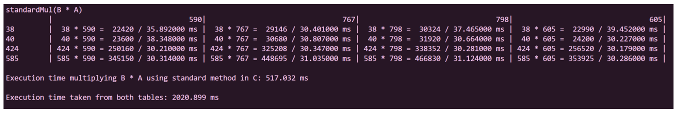


standardMul(A * B)

	38	40	424	585
590	590 * 38 = 22420 / 30.334000 ms	590 * 40 = 23600 / 30.429000 ms	590 * 424 = 250160 / 30.294000 ms	590 * 585 = 345150 / 30.009000 ms
767	767 * 38 = 29146 / 30.141000 ms	767 * 40 = 30680 / 30.317000 ms	767 * 424 = 325208 / 30.267000 ms	767 * 585 = 448695 / 30.412000 ms
798	798 * 38 = 30324 / 35.221000 ms	798 * 40 = 31920 / 30.728000 ms	798 * 424 = 338352 / 30.271000 ms	798 * 585 = 466830 / 30.356000 ms
605	605 * 38 = 22990 / 30.750000 ms	605 * 40 = 24200 / 35.546000 ms	605 * 424 = 256520 / 31.185000 ms	605 * 585 = 353925 / 30.746000 ms

Execution time multiplying A * B using standard method in C: 497.006 ms

Fig. 13. Standard version A * B in C



standardMul(B * A)

	590	767	798	605
38	38 * 590 = 22420 / 35.892000 ms	38 * 767 = 29146 / 30.401000 ms	38 * 798 = 30324 / 37.465000 ms	38 * 605 = 22990 / 39.452000 ms
40	40 * 590 = 23600 / 38.348000 ms	40 * 767 = 30680 / 30.807000 ms	40 * 798 = 31920 / 30.664000 ms	40 * 605 = 24200 / 30.227000 ms
424	424 * 590 = 250160 / 30.210000 ms	424 * 767 = 325208 / 30.347000 ms	424 * 798 = 338352 / 30.281000 ms	424 * 605 = 256520 / 30.179000 ms
585	585 * 590 = 345150 / 30.314000 ms	585 * 767 = 448695 / 31.035000 ms	585 * 798 = 466830 / 31.124000 ms	585 * 605 = 353925 / 30.286000 ms

Execution time multiplying B * A using standard method in C: 517.032 ms

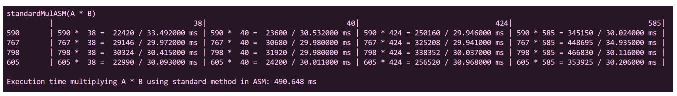
Execution time taken from both tables: 2020.899 ms

Fig. 14. Standard version B * A in C

B. *Standard Assembly version*

Now, lets compare it with Assembly implementation.

In Fig.15 we get an execution time of 490ms after executing all the multiplications, this is a difference of 7ms from Fig.13.



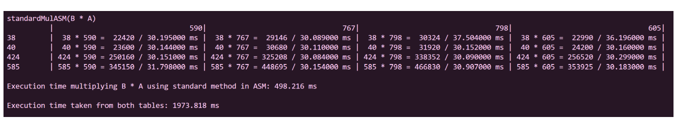
standardMulASM(A * B)

	38	40	424	585
590	590 * 38 = 22420 / 33.492000 ms	590 * 40 = 23600 / 30.532000 ms	590 * 424 = 250160 / 29.946000 ms	590 * 585 = 345150 / 30.024000 ms
767	767 * 38 = 29146 / 29.972000 ms	767 * 40 = 30680 / 29.980000 ms	767 * 424 = 325208 / 29.941000 ms	767 * 585 = 448695 / 34.935000 ms
798	798 * 38 = 30324 / 30.415000 ms	798 * 40 = 31920 / 29.980000 ms	798 * 424 = 338352 / 30.037000 ms	798 * 585 = 466830 / 30.116000 ms
605	605 * 38 = 22990 / 30.093000 ms	605 * 40 = 24200 / 30.011000 ms	605 * 424 = 256520 / 30.968000 ms	605 * 585 = 353925 / 30.206000 ms

Execution time multiplying A * B using standard method in ASM: 490.648 ms

Fig. 15. Standard version A * B in Assembly

While multiplying B * A using Assembly, it performs 19ms better than its C counterpart. we can see it in the comparison between Fig.14 and Fig.16. This indicates that modern compilers are highly efficient at optimizing simple C code, often producing machine code that's comparable to hand-written assembly in performance.



standardMulASM(B * A)

	590	767	798	605
38	38 * 590 = 22420 / 30.195000 ms	38 * 767 = 29146 / 30.089000 ms	38 * 798 = 30324 / 37.504000 ms	38 * 605 = 22990 / 36.196000 ms
40	40 * 590 = 23600 / 30.144000 ms	40 * 767 = 30680 / 30.110000 ms	40 * 798 = 31920 / 30.152000 ms	40 * 605 = 24200 / 30.160000 ms
424	424 * 590 = 250160 / 30.151000 ms	424 * 767 = 325208 / 30.084000 ms	424 * 798 = 338352 / 30.090000 ms	424 * 605 = 256520 / 30.299000 ms
585	585 * 590 = 345150 / 31.798000 ms	585 * 767 = 448695 / 30.154000 ms	585 * 798 = 466830 / 30.907000 ms	585 * 605 = 353925 / 30.183000 ms

Execution time multiplying B * A using standard method in ASM: 498.216 ms

Execution time taken from both tables: 1973.818 ms

Fig. 16. Standard version B * A in Assembly

VI. Ancient Version

A. *Ancient C Version*

Finally, we have the ancient version, with times for individual runs ranging from the lowest, 30.628ms in the 424 * 767 case, to the highest, 39.13ms in the 585 * 605 case.

This version presents a time of 510ms in the A * B order.

And it presents a time of 523ms in the B * A order.

The total time for this version is 2045ms.

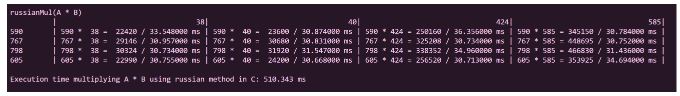
From this test case, we could infer that the ancient algorithm doesn't perform better than its counterpart, which is the standard method, there's a average of 4 ms of difference between them. This can be proved with other different test cases.

B. *Ancient Assembly Version*

While with Assembly, results are barely the same.

In Fig.19 the multiplication between AxB gives a execution time of 499ms, from the 510ms of Fig.17.

While on the other way, BxA gives an execution time of 521ms, just 2ms better than Fig.18.

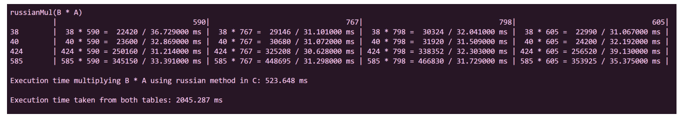


russianMul(A * B)

	38	40	424	585
590	590 * 38 = 22420 / 33.548000 ms	590 * 40 = 23600 / 30.874000 ms	590 * 424 = 250160 / 36.356000 ms	590 * 585 = 345150 / 30.784000 ms
767	767 * 38 = 29146 / 30.957000 ms	767 * 40 = 30680 / 30.831000 ms	767 * 424 = 325208 / 30.734000 ms	767 * 585 = 448695 / 30.752000 ms
798	798 * 38 = 30324 / 30.734000 ms	798 * 40 = 31920 / 31.547000 ms	798 * 424 = 338352 / 34.960000 ms	798 * 585 = 466830 / 31.436000 ms
605	605 * 38 = 22990 / 30.755000 ms	605 * 40 = 24200 / 30.668000 ms	605 * 424 = 256520 / 30.713000 ms	605 * 585 = 353925 / 34.694000 ms

Execution time multiplying A * B using russian method in C: 510.343 ms

Fig. 17. AxB using C Russian Peasant Method



russianMul(B * A)

	590	767	798	605
38	38 * 590 = 22420 / 36.729000 ms	38 * 767 = 29146 / 31.101000 ms	38 * 798 = 30324 / 32.041000 ms	38 * 605 = 22990 / 31.067000 ms
40	40 * 590 = 23600 / 32.869000 ms	40 * 767 = 30680 / 31.072000 ms	40 * 798 = 31920 / 31.509000 ms	40 * 605 = 24200 / 32.192000 ms
424	424 * 590 = 250160 / 31.214000 ms	424 * 767 = 325208 / 30.628000 ms	424 * 798 = 338352 / 32.303000 ms	424 * 605 = 256520 / 39.130000 ms
585	585 * 590 = 345150 / 33.391000 ms	585 * 767 = 448695 / 31.298000 ms	585 * 798 = 466830 / 31.729000 ms	585 * 605 = 353925 / 35.375000 ms

Execution time multiplying B * A using russian method in C: 523.648 ms

Execution time taken from both tables: 2045.287 ms

Fig. 18. BxA using C Russian Peasant Method

VII. Conclusion - Performance Analysis

After seeing the timestamps of the program and different versions, results are:

1. Assembly implementations were consistently faster, but only by small margins (2-34ms).

2. The standard method slightly outperformed the Russian Peasant method in both C and Assembly.

3. A*B operations were generally slightly faster than B*A operations, but this is not always the case. More tests cases can be seen in the annexes section. Although there's no clear reason why this occur.

These results demonstrate that for these algorithms on modern hardware:

1. The performance gap between C and Assembly is minimal.

2. Modern C compilers produce highly optimized code, reducing the impact of hand-written Assembly optimizations.

3. The standard multiplication method slightly outperforms the Russian Peasant method.

4. The order of operands (A*B vs B*A) can affect performance, but not significantly.

A. *How good is the Russian Peasant Method?*

The Russian Peasant method was slightly slower than the standard multiplication method in both C and Assembly implementations.

The method didn't show any significant performance advantages over the standard multiplication algorithm for the tested range of numbers.

It performed consistently across different input values, suggesting stability in its execution time.

VIII. Annexes

This section provides different tests cases done during results recompilation.

The application performed less quickly when it ran on a Linux computer as opposed to a virtual machine. There are several possible reasons for this speed disparity, including variations in the distribution of resources, configurations of the system, and hardware optimizations at play. The virtual machine's environment was personalized for the particular application, which resulted in quicker execution speeds despite Linux's reputation for efficiency. This highlights the significance of customized configurations for obtaining optimal performance. It was necessary to bring up this distinction even if it isn't very pertinent to the project as a whole.

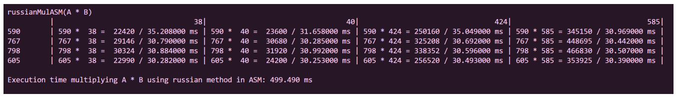


Fig. 19. AxB using Assembly Russian Peasant Method

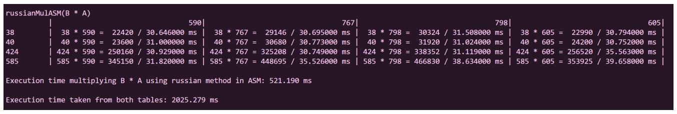


Fig. 20. BxA using Assembly Russian Peasant Method

REFERENCES

[1] "The Russian Peasant Method of Multiplication," Available: https://www.jstor.org/stable/27949747?seq=2. [Accessed: Jul. 26, 2024].

[2] "Russian Peasant Multiplication: How and Why," Available: https://www.themathdoctors.org/russian-peasant-multiplication-how-and-why/. [Accessed: Jul. 26, 2024].

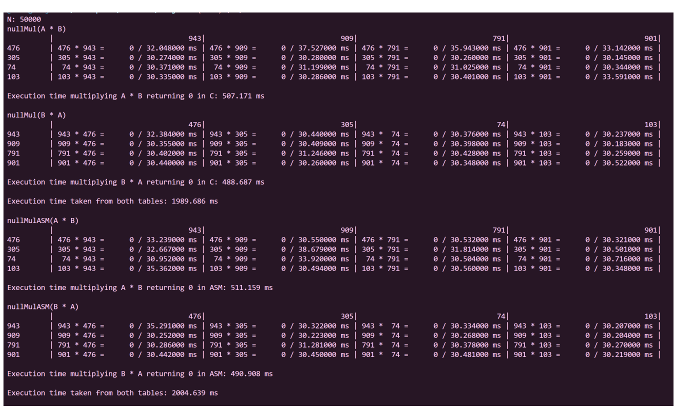


Fig. 21. C outperforms Assembly in Null method

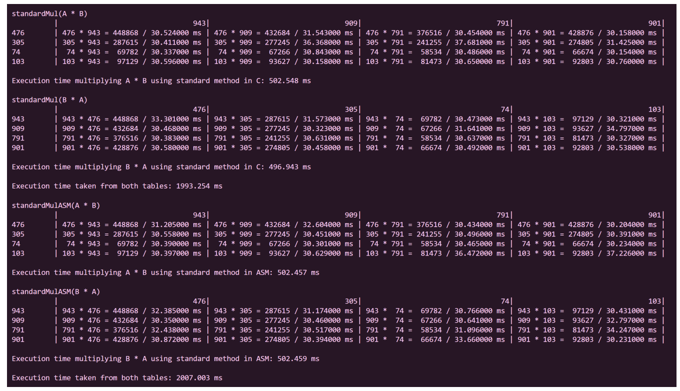


Fig. 22. C outperforms Assembly in Standard method

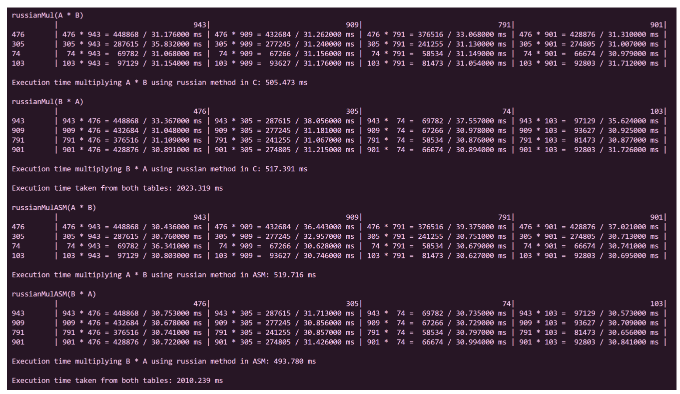


Fig. 23. Inverse performance in Russian Peasant Method

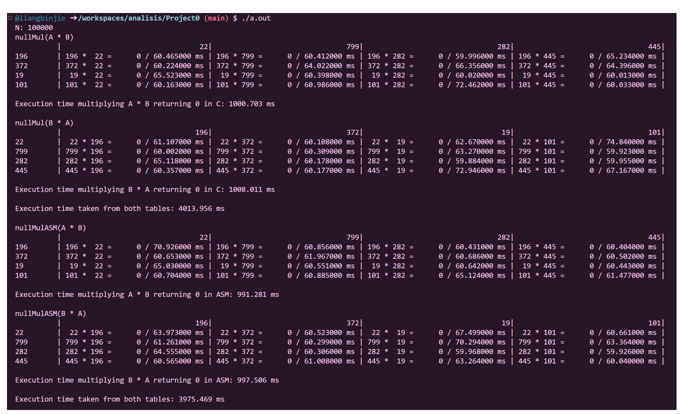


Fig. 24. Better performance in Assembly when N=100000, using null method

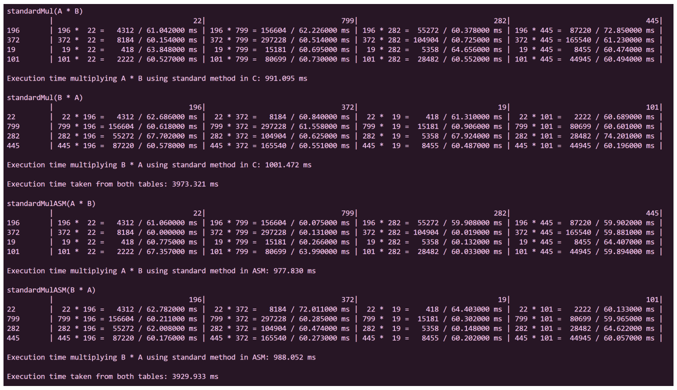


Fig. 25. Better performance in Assembly when N=100000, using standard method

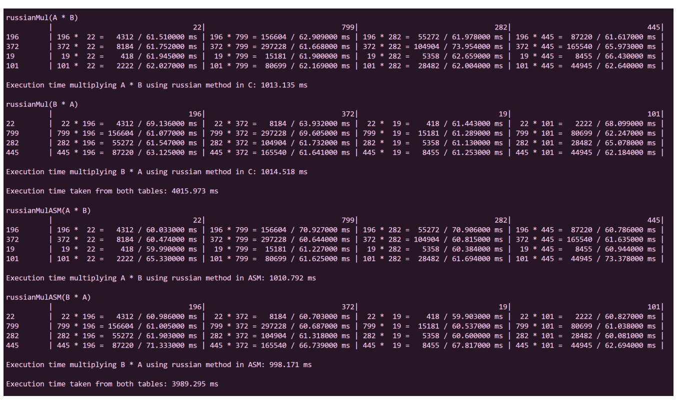


Fig. 26. Minimal difference using Russian Method, when N=100000

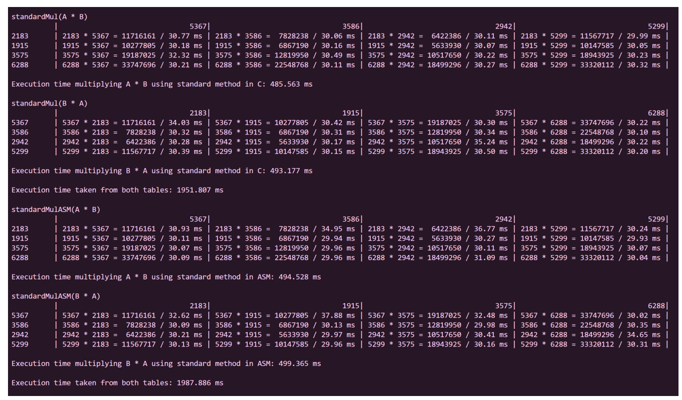


Fig. 28. C outperforms Assembly in standard method

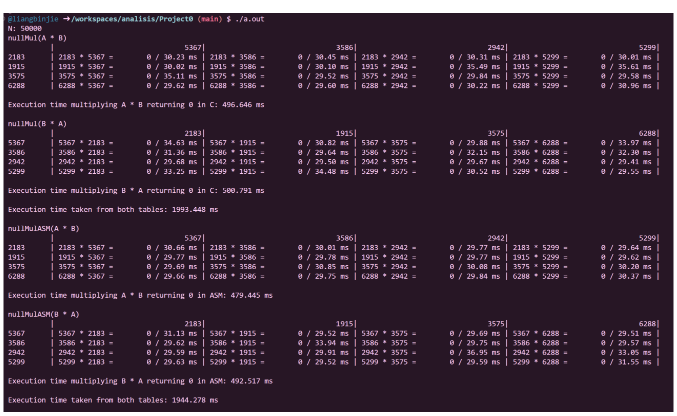


Fig. 27. Assembly outperforms C in null method