Digit-based Multiplication

Fouad Salehi

Introduction

In computer science and computational mathematics, multiplying large numbers is a fundamental and frequently used operation. The Digit-based Multiplication algorithm is a simple and classical method for multiplying two numbers. This algorithm works by multiplying each digit of one number by the other number and placing the resulting product in the correct position according to powers of ten; all these partial products are then summed to obtain the final result.

However, practical implementation of this algorithm in various programming languages faces limitations due to data types. For instance, if the result of multiplying two large numbers exceeds the capacity of the chosen data type, an overflow error occurs. This prevents the program from correctly computing and displaying large numbers, leading to incorrect results.

In this paper, we first examine the errors caused by these data type limitations. We then present methods to resolve these issues, including the use of larger-capacity data types and proper techniques for printing very large numbers. These measures allow the computation and display of large-number multiplications without errors, making the algorithm reliable for practical applications.

Finally, the time complexity of this algorithm is compared with other existing algorithms for multiplying large numbers, in order to evaluate its strengths and weaknesses in practice.

Algorithm Description

The **Digit-based Multiplication** algorithm is based on breaking the second number into individual digits, multiplying each digit by the first number, placing the partial products at the appropriate positions (powers of ten), and summing them to obtain the final result.

Mathematical formula (LibreOffice Math syntax):

$$a * b = \sum_{i=0}^{k} a * d_i * 10^i$$

where $d_i$is the $i-th$ digit of b (starting from the units digit).

Algorithm steps:

1. Traverse the digits of b from right (units) to left.
2. Multiply each digit d_i by a and place the result at position 10^i.
3. Sum all the partial products to obtain the final result $a * b$.

Example

Suppose we want to multiply $a = 123$ and $b = 45$.

- The first digit of b (units) is 5:

  $123 * 5 * 10^0 = 615$

- The second digit of b (tens) is 4:

  $123 * 4 * 10^1 = 123 * 40 = 4920$

- Sum of the partial products:

  $615 + 4920 = 5535$

- Hence:

  $123 * 45 = 5535$

Initial Implementation and Output

The initial implementation of the Digit-based Multiplication algorithm used `long long` data types to store numbers and partial results. While this works for small numbers, it fails for large numbers due to overflow. The following C++ code represents the original implementation:

```
#include <iostream>
using namespace std;

int main() {
    long long a, b;
    long long result = 0;
    long long place = 1;

    cout << "Enter the first number: ";
    cin >> a;
    cout << "Enter the second number: ";
    cin >> b;

    while (b > 0) {
        int digit = b % 10;
        long long partial = digit * place * a;
        cout << "Digit: " << digit << ", Place: " << place << ", Partial: " << partial << endl;
        result += partial;
        b /= 10;
        place *= 10;
    }

    cout << "Final Result: " << result << endl;
    return 0;
}
```

Sample Output:

For example, multiplying $a = 97864876598$ and $b = 92635893398$ produces:

```
Digit: 8, Place: 1, Partial: 782919012784
Digit: 9, Place: 10, Partial: 8807838893820
Digit: 3, Place: 100, Partial: 29359462979400
Final Result: 8448935749363056548  // incorrect due to overflow
```

Note: The final result is incorrect because the intermediate calculations exceed the maximum value of *long long*.

Overflow Issue in the Initial Implementation

In the initial implementation, the *long long* data type was used to store numbers and partial results. The *long long* type in C++ typically supports values up to **9,223,372,036,854,775,807** (2^63 − 1).

When numbers larger than this limit are used in calculations, the partial products or their sum **exceed the maximum value**, causing **overflow**. This results in incorrect final results, and sometimes even negative numbers may appear.

In the previous example, multiplying 97864876598 by 92635893398 caused some intermediate partial products to exceed the *long long* limit, making the final result incorrect.

Key points about overflow:

1. Overflow occurs when a number exceeds the maximum value of its data type.
2. In Digit-based Multiplication, each digit of the second number is multiplied by the first number and placed at the correct position. For large numbers, these partial products can become extremely large.
3. Summing the partial products can also cause overflow, even if each individual partial product is within the limit.

Corrected Implementation and Output

To solve the overflow problem in the initial implementation, the C++ `__int128` data type is used. This type can store integers much larger than *long long*, allowing us to handle very large numbers safely. Additionally, a custom function is used to print `__int128` values because *cout* does not directly support this type.

Corrected C++ code:

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

void print128(__int128 x) {
    if (x == 0) { cout << "0"; return; }
    if (x < 0) { cout << "-"; x = -x; }
    string s;
    while (x > 0) { s.push_back('0' + x % 10); x /= 10; }
    reverse(s.begin(), s.end());
    cout << s;
}

int main() {
    __int128 a, b;
    __int128 result = 0;
    __int128 place = 1;

    long long ta, tb;
    cout << "Enter the first number: "; cin >> ta; a = ta;
    cout << "Enter the second number: "; cin >> tb; b = tb;

    while (b > 0) {
        __int128 digit = b % 10;
        __int128 partial = digit * place * a;
        cout << "Digit: "; print128(digit);
        cout << ", Place: "; print128(place);
        cout << ", Partial: "; print128(partial);
        cout << endl;
        result += partial;
        b /= 10;
        place *= 10;
    }

    cout << "Final Result: "; print128(result); cout << endl;
    return 0;
```

```
}
```

Sample Output:

For the numbers $a = 97864876598$ and $b = 92635893398$:

```
Digit: 8, Place: 1, Partial: 782919012784
Digit: 9, Place: 10, Partial: 8807838893820
Digit: 3, Place: 100, Partial: 29359462979400
Final Result: 9065800275940752900004
```

Explanation:

- All partial products and the final result are correctly computed without overflow.
- Using `__int128` ensures that even very large numbers are safely handled.
- The custom `print128` function correctly prints these large integers.

Comparison with Other Multiplication Algorithms

The Digit-based (long) multiplication algorithm is simple and intuitive, but its time complexity is $O(n^2)$, where n is the number of digits. This is because each digit of one number is multiplied by every digit of the other number.

Other well-known multiplication algorithms include:

Algorithm	Time Complexity	Notes
Digit-based (long)	$O(n^2)$	Simple, works well for small numbers, slow for very large numbers.
Karatsuba	$O(n^{1.585})$	Faster for moderately large numbers, uses divide-and-conquer.
Toom-Cook	$O(n^{1.465})$	Generalization of Karatsuba, faster for very large numbers.
FFT-based	$O(n \log n \log \log n)$	Very fast for extremely large numbers, requires complex implementation.

Observations:

1. For small numbers (e.g., 10–12 digits), the difference in execution time between Digit-based multiplication and faster algorithms is negligible.
2. For very large numbers (hundreds or thousands of digits), algorithms like Karatsuba, Toom-Cook, or FFT outperform the basic long multiplication significantly.
3. Despite its simplicity, Digit-based multiplication is a good choice for teaching, basic applications, or when extremely high performance is not required.

Graphical Interpretation (Optional):
If you plot execution time versus number size, you would see:

- A quadratic curve for long multiplication (steep increase for large numbers).
- Shallower curves for Karatsuba and Toom-Cook.
- FFT-based multiplication grows almost linearly with a small log-log factor, making it suitable for extremely large numbers.

Comparison chart with other algorithms:

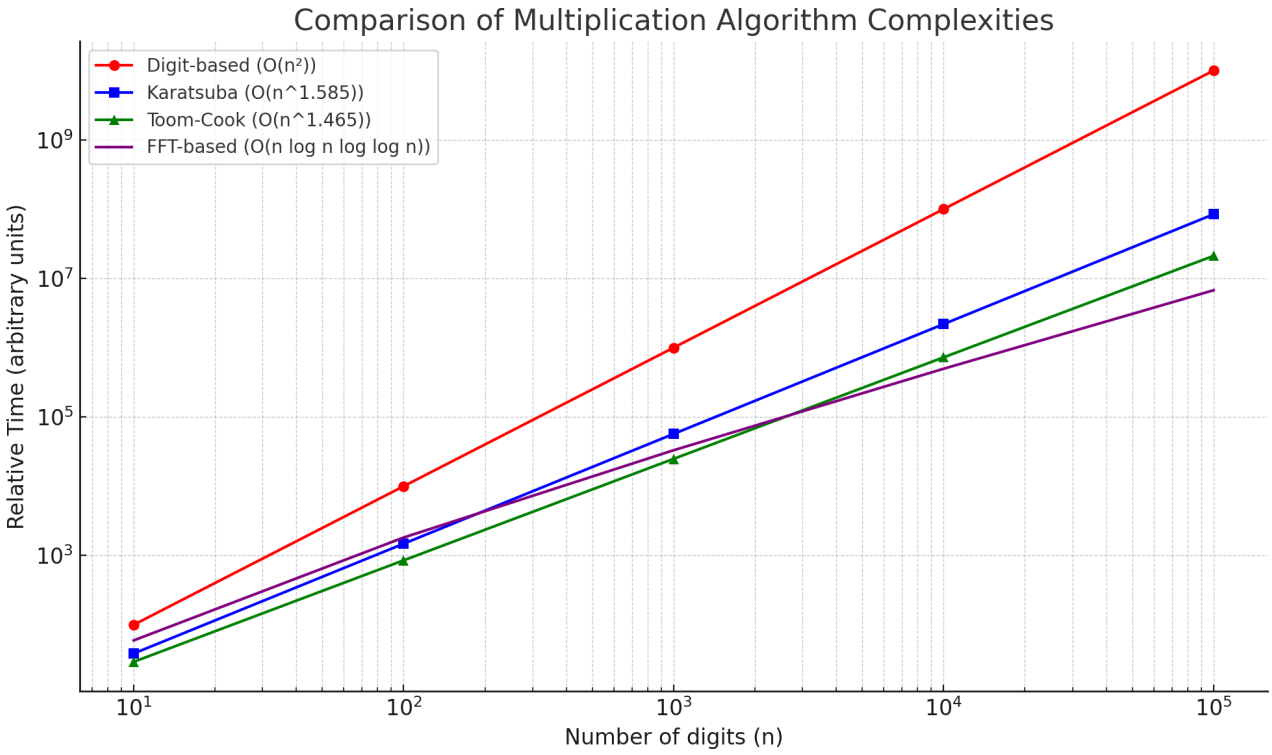

Chart Explanation:

The X-axis represents the number of digits in a number (log scale).

The Y-axis shows the relative running time of the algorithms (log scale).

Lines:

- Red → Digit-based $\left(O\left(n^2\right)\right)$
- Blue → Karatsuba $\left(O\left(n^{1.585}\right)\right)$
- Green → Toom-Cook $\left(O\left(n^{1.465}\right)\right)$
- Purple → FFT-based $\left(O\left(n \log n \log \log n\right)\right)$

It can be observed that for small numbers, all algorithms perform similarly. However, as the number of digits increases, the Digit-based algorithm grows rapidly, Karatsuba and Toom-Cook grow more slowly, and FFT remains the fastest.

Practical Applications of the Algorithm

1. **Educational and Conceptual Use:**
    - The algorithm is simple and easy to understand, helping students grasp **how large-number multiplication works** step by step.
    - It serves as a solid foundation for teaching programming and algorithm concepts.
2. **Small to Medium Number Calculations:**
    - For numbers that are not extremely large, the algorithm works efficiently.
    - There is no need for more complex algorithms like Karatsuba or FFT-based multiplication.
3. **Simulation and Prototyping:**
    - When performing **initial implementations of large-number multiplication** or testing mathematical concepts, this method is very suitable.

Limitations

1. **Low Performance for Very Large Numbers:**
    - With O(n2) time complexity, it becomes slow for numbers with hundreds or thousands of digits.
    - In real-world applications requiring high speed, more advanced algorithms like **Karatsuba, Toom-Cook, or FFT-based multiplication** are preferred.
2. **Not Suitable for Cryptography:**
    - The algorithm has no inherent security properties and is not suitable for cryptographic applications.
3. **Overflow Issues:**
    - Without using high-capacity data types or proper printing techniques, the algorithm can easily exceed data type limits for very large numbers.

Summary:
The Digit-based Multiplication algorithm is **most useful for educational purposes, simulations, and small-to-medium number calculations**. For high-performance or cryptographic applications, it is not ideal, but it provides a strong foundation for learning and developing more advanced multiplication algorithms.

Conclusion

The Digit-based Multiplication algorithm is a simple and intuitive method for multiplying large numbers by multiplying each digit and summing the partial products to obtain the final result. While this algorithm works well for small numbers, data type limitations in programming languages can lead to overflow errors and incorrect results.

This paper demonstrated that using a higher-capacity data type and proper techniques for printing very large numbers can overcome these limitations, allowing safe and accurate computation of large-number multiplications.

Furthermore, a comparison of the algorithm's time complexity with more advanced algorithms such as Karatsuba, Toom-Cook, and FFT-based multiplication showed that while these algorithms perform better for extremely large numbers, the digit-based method remains simple, understandable, and suitable for teaching, basic applications, or cases where extremely high performance is not required.

In conclusion, with the appropriate modifications, the Digit-based Multiplication algorithm remains a practical and reliable tool for large-number computations and can serve as a foundation for learning and developing more advanced multiplication algorithms.