Mikhail Kornilov, Shreyas Mayya, Barrett Ruth, Vincent Trang
28 April 2025
CS 4501
Floryan

Executive Summary

Introduction

Large integer multiplication is relevant to many fields and technologies that impact our day-to-day lives, including cryptography, scientific and financial computing, computation number theory, and digital signal processing. Naive integer multiplication of two *n*-digit numbers is $\Theta(n^2)$, which is utterly impractical for the size and scale that the aforementioned application requires. In the early 1960s, Karatsuba multiplication $\Theta(n^{log_2 3})$ was widely used for multiplication until Andrei Toom and Stephen Cook created the Toom-Cook multiplication algorithm in the mid 1960s, which presented a more practical runtime for large numbers. The algorithm runs in $\Theta(n^{log_k(2k-1)})$ where *k* is an implementation-defined parameter, the number of segments to split each *n*-digit number into.

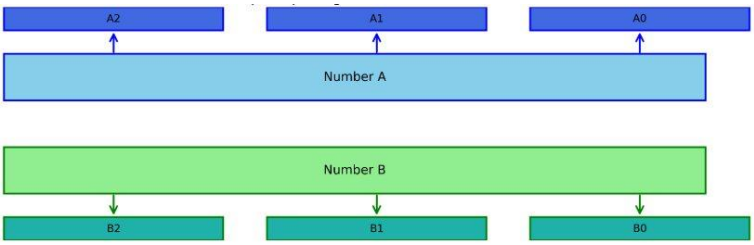
While the theoretical asymptotic bound of integer multiplication is $O(nlog(n))$, the primary challenge in creating an efficient fast multiplication algorithm is minimizing the "constant factor," i.e. how long it takes to run in practice. Toom-Cook primarily leverages the fact that integer multiplication can be reduced to polynomial multiplication. At a finer grain, this means numbers and polynomials must be represented efficiently in program memory, a choice of *k* must be appropriate given the input, and big-integer types must be efficiently implemented at a language-level for the algorithm to be practical.

Overall Intuitive Approach/Solution

Toom-Cook multiplication is a divide-and-conquer algorithm heavily correlated to the Karatsuba algorithm. The key observation is that, by splitting the input integers into pieces of size *k*, the product of such pieces can recreate the integer product, all while using less multiplications than the naive approach.

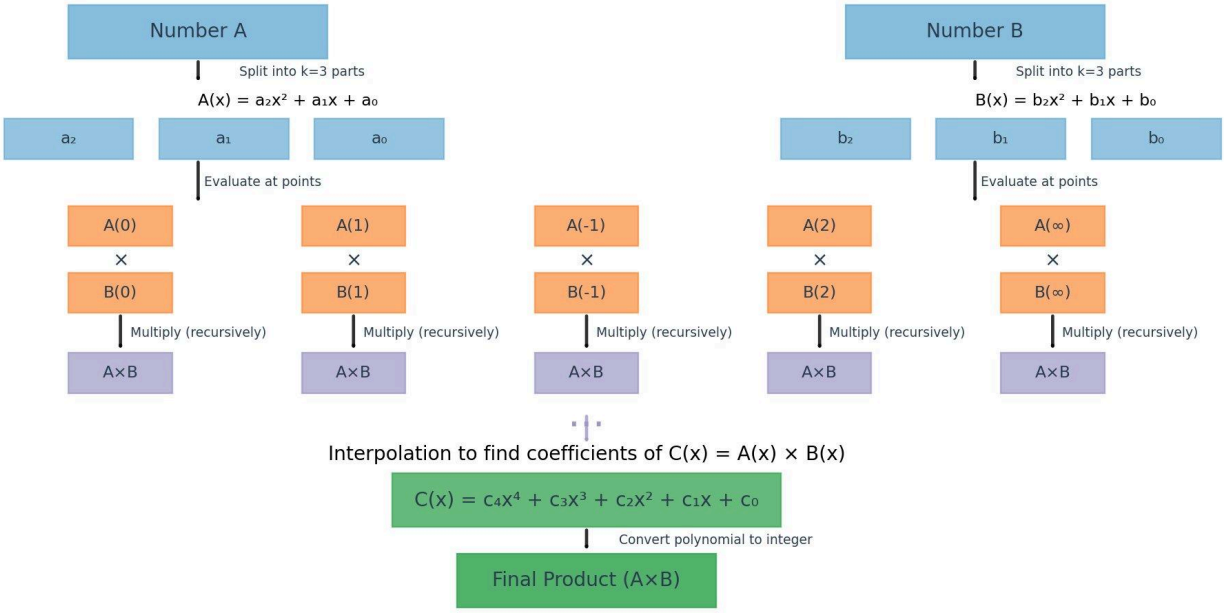
Specifically, Toom-Cook rests on the fact that a base *Z* integer can be represented as a polynomial. Take the familiar $Z = 10$—any base 10 number *N* can clearly be represented as the polynomial $N = a_0 + 10 \cdot a_1 + 10^2 \cdot a_2 +$... Therefore, integer multiplication reduces to polynomial multiplication, i.e. $N_1 \cdot N_2 = (\Sigma\, b^i N_{1i}) \cdot (\Sigma\, 1b^i N_{2i})$ in base *b*.

A simple example of Toom-Cook offers the best insight into how it works. Let Toom-*n* mean k, the number of parts to split each input integer into, equals *n*. Consider Toom-3 on numbers *A* and *B*. After splitting them as seen below, two corresponding coefficient polynomials $A(x) = A_2x^2 + A_1x + A_0$ and $B(x) = B_2x^2 + B_1x + B_0$ are implicitly formed:



In this case, finding degree 4 polynomial $C(x) = A(x) \cdot B(x)$ is the goal. Mathematically, it has been shown that $A(x)$ and $B(x)$ only need to be evaluated at 5 points (one more than the degree of $C(x)$) are needed to find $C(x)$. Naively, $O(5^2)$ multiplications need to be performed to find the coefficients of $C(x)$—however, with some fancy math, not all of these multiplications are necessary—*only the five* $A_i \cdot B_i$ *are*. The coefficients are intelligently reconstructed using "interpolation," and the points of interpolation should be carefully chosen.

When it comes to multiplying $A_i \cdot B_i$, note that the first blocks of digits are size n/k. The task reduces to multiplying various pairs integers of size n/k, which can be done recursively (in the base case, direct multiplication is used). This is a core reduction in work that leads to Toom-Cook's asymptotic improvement over the naive algorithm. For simplicity's sake, the Toom-3 algorithm is visualized below:



Implementation

The following implementation explanation will describe implementing Toom-*k* but code/methods is/are specialized for $k = 3$.

To implement Toom-*k* with input $x, y \in \mathbb{Z}$, recall the first step of the algorithm is to find an appropriate base that can "split" the input integers into digit groups of size *k* (this will implicitly form polynomials $A(x), B(x)$. So, to split up x and y, see how many digits-per-group

the larger of x and y requires —namely, $b := max\{\lfloor log_{10}(x) \rfloor + 1, \lfloor log_{10}(y) \rfloor + 1\} \div k$ (recall that in base 10, n has $\lfloor log_{10}(n) \rfloor + 1$ digits). Since we need b digits per group, the base is 10^b.

Next, split the number into k parts. To "extract out" the first b digits of a decimal number, take, modulo 10^b. Then, (integer) divide the number by the base and repeat the process. For Toom-3, repeat this 3 times and store the 3 separate parts for x and y.

Now for the beast: the multiplication itself. As Toom-Cook is a divide and conquer recursive algorithm, consider the base and recursive case separately. For the base cases, consider the following:

a) When the input is appropriately sized (i.e. fast enough to perform naive/builtin multiplication), simply return the product $x \cdot y$. For the sake of simplicity, if $|x| \leq 10$ or $|y| \leq 10$, return this product, although profiling a fine-tuned range is more realistic.
b) If $x = 0$ or $y = 0$, return 0
c) If $x = 1$ or $y = 1$, return $max\{x, y\}$

Now, to execute the divide-and-conquer given non-trivially sized x and y. The two parts implicitly form degree 2 polynomials, with some product $C(x) = c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$ —we seek the coefficients.

1. Split x and y using the aforementioned subroutines for calculating a base and performing the split—this is $A(x), B(x)$.
2. Evaluate the polynomials at $2k - 1$ distinct points. For Toom-3, commonly used points are $0, 1, -1, 2, \infty$. For example, with $x_0 = 0$, we take a_0 and b_0, the 0th degree terms of $A(x)$ and $B(x)$ respectively. For each point x_i, *recursively* compute the product of the size n/k integers $A(x_i) \cdot B(x_i)$.
3. Interpolate the results—namely, recombine the products computed from the previous step to find each c_i. In the case of Toom-3 with the aforementioned interpolation points:

$$c_0 = TCMul(A(x_0), B(x_0))$$
$$c_4 = TCMul(A(x_4), B(x_4))$$
$$c_2 = \lfloor (TCMul(A(x_1), B(x_1)) + TCMul(A(x_2), B(x_2))/2 \rfloor - c_0 - c_4$$
$$c_3 = \lfloor (TCMul(A(x_3), B(x_3)) - 2 \cdot TCMul(A(x_1), B(x_1)) + c_0 - 2 \cdot c_2 - 14 \cdot c_4) \rfloor / 6$$
$$c_1 = TCMul(A(x_1), B(x_1) - c_0 - c_2 - c_3 - c_4$$

While this may seem complicated, only 5 (or, generally, $2k - 1$) recursive calls are actually made on block sizes that are divided by k per call.

Assuming the parts have been multiplied correctly, the penultimate step is to transform the final broken-up number into its integer form. To do this, perform the reverse of the

aforementioned “split” procedure: simply multiply the ith part/polynomial coefficient (right-to-left) by b^i and sum the parts together—this is the final answer.

Summary of Programming Challenge

This programming challenge is based on a clever application of multiplication in non-integer bases. In this programming challenge, you are given a list of coins with unknown values, but you do know that their values satisfy a certain recurrence. Your task is to find another set of coins whose total value has been multiplied by a certain integer.

The learning objectives of this programming challenge are to get the solver to think about Toom-Cook in a more general way, to understand its details intricately, and to understand how it can be applied as a subroutine in other algorithms. Our programming challenge contains multiple twists that require highly nontrivial applications and modifications to the Toom-Cook algorithm. For example, one aspect of the programming challenge requires a conversion of an integer between different bases—Toom-Cook needs to be used to ensure this conversion runs in subquadratic time. Overall, the sample cases and extensive explanations for them should give enough insight into the problem that a smart solver should be able to make progress.

The constraints are chosen such that Toom-Cook multiplication significantly outperforms the naive approach: it gives a nearly 10x speedup on the largest test cases. Some of the largest test cases may take several minutes to run with Toom-Cook—but testing has shown that they would take even longer without it.

Key Ideas For Solving the Programming Challenge

The first main difficulty in the programming challenge is just figuring out how to approach it, and reduce it to Toom-Cook multiplication.

The problem statement reduces to the following: there's a hidden sequence of numbers $\ldots c_{-2}, c_{-1}, c_0, c_1, c_2, \ldots$, corresponding to the values of the coins, where all you know is that $c_i = c_{i+1} + c_{i+2}$ for all integers i. Then, you are given a list of distinct integer indices $a_1, a_2, \ldots$, corresponding to which specific coins are in the pile, and a multiplier k. You need to find the shortest possible list of distinct integer indices $b_1, b_2, \ldots$ such that $k(c_{a_1} + c_{a_2} + \ldots) = c_{b_1} + c_{b_2} + \ldots$, representing which specific coins are to be in the pile you output.

The key idea is to consider the sequence $c_i = \varphi^i$, where $\varphi = \frac{1+\sqrt{5}}{2}$. Then we need to solve the equation $k(\varphi^{a_1} + \varphi^{a_2} + \ldots) = \varphi^{b_1} + \varphi^{b_2} + \ldots$, which is just a multiplication in base φ:

provided that we can convert k into a sum of powers of φ, all we need to do is multiply the two base-φ numbers on the left-hand side (since k and $a_1, a_2, \ldots$ are given to us), and this will give us a base-φ number that we can use to find the right-hand side. With a bit of work (the math isn't too involved, but would clutter up this explanation), you can show that this approach gives us a solution that works for *all* possible values of the hidden starting sequence!

In short, to solve the programming challenge, we interpret our list of given coin values as a number in base φ, our multiplier k as another integer in base φ, and use Toom-Cook to multiply these numbers together (since they can be large).

To get the *shortest possible* list that works, the idea is to start with any working solution, then improve it. Since we know that $c_i = c_{i+1} + c_{i+2}$ for all integers i, if there are two coins with consecutive indices, let's say $v + 1$ and $v + 2$, they can be replaced with a single coin v and reduce the number of coins you need to use. This process repeats until no two coins have consecutive indices. It can be shown that there is always only one solution where no two coins have consecutive indices, so this approach is guaranteed to give the correct (optimal) solution. This can be done with a single pass from the highest-labeled coin to the lowest-labeled coin, so this step can be done very efficiently.

Now that the approach to solving this problem is clear, there are some more implementation challenges to tackle, but which will force anyone attempting the Programming Challenge to fully understand the finer details of how Toom-Cook multiplication works.

The most complex step of Toom-Cook is the interpolation step: the matrix inverse can be hardcoded, but this step still requires a matrix-vector multiplication regardless, so to get the subquadratic time complexity that warrants using Toom-Cook in the first place, we need to be able to add, subtract, and multiply by constants in linear time. If the matrix inverse were to contain fractions, we would also need to be able to divide by constants in linear time.

It turns out that if the interpolation points are chosen carefully, fractions in the inverse can be completely avoided! One such set of interpolation points that work is $0, 1, \varphi, \varphi^2, \infty$. It is actually required to use ∞ if you want this property with Toom-3, so this forces you to understand how ∞ works as an interpolation point. (In particular, p(∞) is defined as the leading coefficient of p, and since p(∞)q(∞) = (pq)(∞), this is suitable as an interpolation point as defined).

Not have to implement division is very useful, but implementing base-phi arithmetic is still a challenge. The main difficulty is that, in contrast to typical positive integer bases, carries can propagate in both directions (consider $2 = \varphi + \varphi^{-2}$, which has both positive and negative powers). The way to deal with this is to first compute the sum or difference digit-by-digit (for

example, $101.010 + 1.001 = 102.011$), then passing through the array several times and applying simplifications (for example, any 2 can be replaced with 10.01). Once base-phi addition has been implemented, schoolbook long multiplication comes easily. It's a good thing that having to implement division was avoided, because subtracting a small number from the start of a large one can still take O(n) because of the bidirectional carry propagation, which would have greatly increased the time complexity.

The final hurdle is converting our integer k from base 10 into base φ, and doing so quickly. The standard algorithm requires n multiplications of the form $(number\ with\ up\ to\ O(n)\ digits) \times (constant)$, which leads to a time complexity of $O(n^2)$, which is too slow. However, we have a fast multiplication algorithm. It shouldn't take too much thinking to figure out how to use it to get a faster base conversion algorithm: the idea is if you have a number in base b, you can combine two digits at a time to convert that number to base b^2. Two consecutive digits x and y can be combined into a single digit $xb + y$ in base b^2, and you can repeat this process until you only have one digit left. The key is to express all the digits in the target base (in this case, base φ), so that when we only have one digit left, that digit is just the base-φ representation of the original number. The runtime of this algorithm is dominated by the computation of xb and b^2 when b becomes large, and if Toom-3 multiplication is used, this algorithm also has complexity $O(n^{log_3 5})$. Leveraging all of the aforementioned algorithms, for the main task of multiplying the two base-φ numbers, a complexity of $O(n^{log_3 5})$ can be achieved.

In summary, the main twists that this programming challenge offers beyond just a naive implementation of Toom-Cook are:

1. Mathematical ingenuity
    a. Since this is a harder algorithm, this seems warranted: it shows how the need for Toom-Cook can arise naturally out of a wider range of problems.
2. Toom-Cook in a Non-Standard Base
    a. This shows anyone solving this programming challenge that Toom-Cook multiplication is more flexible than one might think, and forces them to understand more deeply what's going on under the hood since you do have to make slight changes (i.e. changing the interpolation points).
3. Toom-Cook as a Subroutine for Base Conversion
    a. Again, this shows how the need for fast multiplication can arise "naturally" out of a wider range of problems. It also requires anyone solving the programming challenge to think outside the box a little bit, which is a good feature.

Conclusion

The Toom-Cook multiplication algorithm is a large significant advancement in large integer multiplication, offering an $O(n^{\log_k(2k-1)})$ complexity for multiplying n-digit numbers, an asymptotic improvement over the naive $O(n^2)$ method. This is done by elegantly reducing integer multiplication to polynomial multiplication along with clever interpolation techniques to reduce the needed multiplications.

Generally, efficient multiplication is vital for real-world applications through not only raw calculations (i.e. for scientific computing) but also in optimizing initially infeasible algorithms. The programming challenge reinforces this aspect of the Toom-Cook algorithm's significance by requiring students to apply it in a non-standard base (φ) while solving a sequence-based problem—much more than a "plug-and-play" application or trivial reduction. Lastly. While Toom-Cook is not optimal for the largest scale multiplications, its use in both the Programming Challenge and real-world applications indicate it as a quick and effective subroutine for algorithmic speedups.