COMP 363 - Algorithms: Assignment 2: Karatsuba Multiplication

Due on 30 January, 2026

Leo Irakliotis

Nathan Hogg

1 Introduction

In this assignment, I implemented the Karatsuba multiplication algorithm to demonstrate its efficiency improvements over standard recursive multiplication. While standard multiplication operates in $\mathcal{O}(n^2)$ time, Karatsuba optimizes this to approximately $\mathcal{O}(n^{1.58})$ by reducing the number of recursive calls from four to three.

2 Verification

To ensure the correctness of the Karatsuba implementation, I ran the algorithm against a suite of test cases, comparing the results to Python's built-in integer multiplication. As shown in the output below, the implementation passed all sanity checks, including edge cases with zeros and large numbers.

```
12 * 34 = 408  (ok=True)
99 * 99 = 9801  (ok=True)
0123 * 0456 = 56088  (ok=True)
1234 * 5678 = 7006652  (ok=True)
0000 * 0000 = 0  (ok=True)
1111 * 0001 = 1111  (ok=True)
1234567890123456 * 9876543210123456 = 1219326311263526... (ok=True)
```

3 Performance Analysis

3.1 Benchmark Data

The algorithm was benchmarked against inputs of size $n = 4$ to $n = 2048$. The table below summarizes the execution time in seconds.

n	Simple (s)	Karatsuba (s)
4	0.000006	0.000019
8	0.000022	0.000030
16	0.000081	0.000101
32	0.000307	0.000335
64	0.001252	0.000939
128	0.004738	0.002732
256	0.019185	0.008245
512	0.077341	0.024642
1024	0.499853	0.073690
2048	1.433454	0.225801

3.2 Visual Comparison

The graph below visualizes the performance crossover. Note the use of a log-log scale.

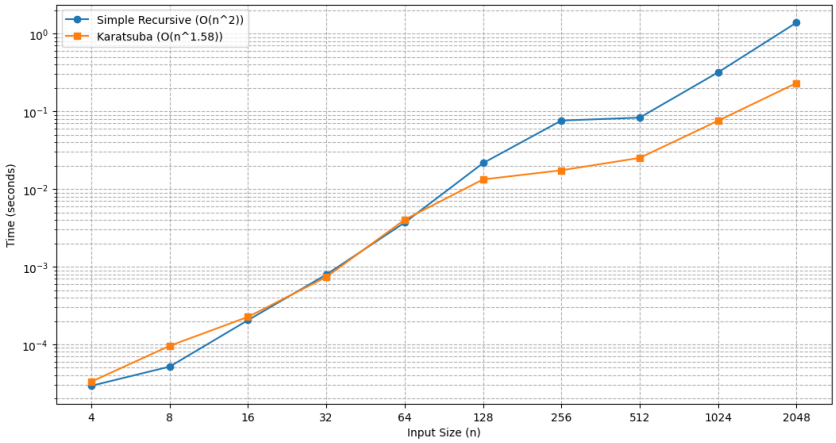

Figure 1: Comparison of execution time between Simple Recursive Multiplication and Karatsuba Multiplication.

3.3 Discussion of Results

The benchmarking results clearly demonstrate the theoretical difference in time complexity between the Simple Recursive Multiplication ($O(n^2)$) and the Karatsuba algorithm ($O(n^{\log_2 3} \approx n^{1.58})$).
For small input sizes ($n < 64$), the Simple Recursive method was marginally faster. This is expected due to the overhead associated with the Karatsuba implementation, which involves additional string conversions, padding, and arithmetic operations before the recursive calls can be made. However, a crossover point was observed around $n = 64$. Beyond this point, Karatsuba became significantly more efficient.
The difference in scaling became dramatic at larger input sizes. At $n = 1024$, the Simple method took approximately 0.50 seconds, while Karatsuba took only 0.07 seconds. At $n = 2048$, the gap widened further, with the Simple method taking 1.43 seconds compared to Karatsuba's 0.23 seconds. This aligns with the theoretical prediction that Karatsuba's performance degrades much more slowly than the quadratic degradation of the simple approach.

4 Implementation Details

A key implementation challenge involved handling strings of different lengths. The standard Karatsuba algorithm assumes inputs split evenly into $n/2$ parts. However, during recursion, the intermediate sums (e.g., $a+b$) can result in strings of different lengths due to carry-over digits (e.g., adding two 2-digit numbers can result in a 3-digit number).
If these differing lengths are passed directly into the recursive function, the split point m is calculated incorrectly for the shorter string, breaking the logic of the power-of-10 shifts. To solve this, I implemented a padding step before the third recursive call. The code calculates the maximum length of the two sums (‘max_len‘) and uses Python's ‘zfill‘ method to pad the shorter string with leading zeros. This ensures that both inputs to the recursive call have the same length, preserving the symmetry required for the divide-and-conquer approach to function correctly.

Appendix: Source Code

```
1
2  def k_multiply(x: str, y: str) -> str:
3      """
4      Karatsuba recursive multiplication for nonnegative integer strings.
5
6      Uses the optimized formula with only 3 recursive calls:
7      xy = ac*10^n + ((a+b)(c+d) - ac - bd)*10^(n/2) + bd
8
9      Includes padding logic using .zfill to handle cases where recursive
10     sums (a + b) and (c + d) result in strings of unequal length.
11     """
12     n = len(x)
13
14     if n == 1:
15         return str(int(x) * int(y))
16
17     m = n // 2
18
19     a = x[:m]
20     b = x[m:]
21     c = y[:m]
22     d = y[m:]
23
24     half_len = len(b)
25
26     a_plus_b = str(int(a) + int(b))
27     c_plus_d = str(int(c) + int(d))
28
29     max_len = max(len(a_plus_b), len(c_plus_d))
30
31     a_plus_b = a_plus_b.zfill(max_len)
32     c_plus_d = c_plus_d.zfill(max_len)
33
34     a_c = k_multiply(a, c)
35     b_d = k_multiply(b, d)
36
37     a_b_times_c_d = k_multiply(a_plus_b, c_plus_d)
38
39     term1 = a_c + ("0" * (2 * half_len))
40     term2 = str(int(a_b_times_c_d) - int(a_c) - int(b_d)) + ("0" * half_len)
41
42     return str(int(term1) + int(term2) + int(b_d))
```

Listing 1: Karatsuba Implementation