NATIONAL RESEARCH
UNIVERSITY

REPORT ON THE COMPARISON OF VARIOUS INTEGER MULTIPLICATION ALGORITHMS

Anton Ivanov, under supervision of George Piatsky

DSBA, Group 191-1

April 2020

1 The Problem

The objective of this research is to estimate theoretical complexity of three different multiplication algorithms, namely **School-grade multiplication**, **Divide and Conquer multiplication**, and **Karatsuba multiplication**, as well as to empirically prove or disprove possible hypotheses concerning the respective complexities of these algorithms

1.1 School-grade multiplication

The algorithm is based on the notions that are usually taught to people at school, this is where the name comes from. Basically, in order to multiply two integers of size n, one is ought to multiply each possible pair of integers (that is n^2 pairs) and then add this products to the resulting number, minding the positions to which add the products.
Remark: from now on term 'problem size' will correspond to the length of both numbers that are going to be multiplied.

One of the possible implementations may look like this

Source Code 1: School-grade multiplication

```
1  Number SchoolMultiplicator::multiply(const Number &n1, const Number &n2) const
2  {
3      std::string result(n1.size() + n2.size(), char(0));
4
5      for (size_t i = 0; i < n1.size(); ++i)
6      {
7          int carry = 0;
8          for (size_t j = 0; j < n2.size(); ++j)
9          {
10             result[i + j] += (n2.at(j) * n1.at(i) + carry);
11             carry = result[i + j] / 10;
12             result[i + j] %= 10;
13         }
14         result[i + n2.size()] = carry;
15     }
16     Number product = Number(std::move(result));
17
18     return product;
19 }
```

Here numbers n1, n2 are stored as objects of class Number, that is based on std::string, where the element at index 0, char(last digit of the number). So number 123 is stored as [char(3), char(2), char(1)]

The basic intuition that we perform almost fixed number of operations n^2 times, suggest that $T(n) \in \Omega(n^2)$, if we denote number of seconds to needed to multiply two n-digit integers using this algorithm as $T(n)$. Moreover, our implementation also suggests that

$T(n)$ is a quadratic function. Therefore, a **hypothesis** can be drawn.

Hypothesis: time complexity of **School-grade multiplication** $= \Theta(n^2)$.

1.2 Divide and Conquer multiplication

This recursive algorithm is based on the fact that $\forall A \in \mathbb{N} \;\; \exists t \in \mathbb{N} : A = A_1 \cdot 10^t + A_2$ and therefore, product of two numbers A and B can be expressed as follows:

$$AB = (A_1 \cdot 10^t + A_2)(B_1 \cdot 10^t + B_2) = A_1B_1 \cdot 10^{2t} + (A_1B_2 + A_2B_1) \cdot 10^t + A_2B_2$$

This expression allows us to compute product of two n-digit numbers by instead computing 4 products of $\frac{n}{2}$-digit numbers as well as some additions.

Source Code 2: Divide and Conquer multiplication

```
Number CaesarMultiplicator::multiply(const Number &n1, const Number &n2) const
{
    const size_t min_value = 1; // modify to optimize
    if (n1.size() <= min_value || n2.size() <= min_value)
    {
        std::unique_ptr<Multiplicator> fallback = std::make_unique<SchoolMultiplicator>();
        return fallback->multiply(n1, n2);
    }
    size_t m = std::min(n1.size(), n2.size()) / 2;

    //split numbers into two, one of which is of length m
    std::pair<Number, Number> a = n1.split(m);
    std::pair<Number, Number> b = n2.split(m);
    Number result;

    // A = A1 * t + A2
    // t = 10^m
    // AB = A1B1*t^2 + (A2B1 + A1B2)*t + A2B2 - 4 multiplications

    Number a1b1 = this->multiply(a.first, b.first);
    Number a2b2 = this->multiply(a.second, b.second);
    Number a1b2 = this->multiply(a.first, b.second);
    Number a2b1 = this->multiply(a.second, b.first);

    a2b1.shift(m);
    a1b2.shift(m);
    a1b1.shift(2 * m);

    return a1b1 + a2b1 + a1b2 + a2b2;
}
```

From this equality we can estimate the time complexity $T(n)$

$$T(n) = 4T(n/2) + O(n)$$

The formula can be 'explained' as following: at each recursive call we have 4 subproblems, of size $n/2$, and addition can be done in linear time. Applying **Master Theorem** [Thomas H. Cormen. Charles E. Leiserson. Ronald L. Rivest. Clifford Stein 2009. Introduction to algorithms. Third edition, 94.] one receives $T(n) = \Theta(n^2)$

Hypothesis: time complexity of **Divide and Conquer multiplication**$= \Theta(n^2))$

1.3 Karatsuba multiplication

This approach was proposed by Soviet mathematician Anatoly Karatsuba in 1960 and named after him. The algorithm is quite similar to **Divide and Conquer multiplication**, as it also recursive and bases on dividing main problem into many smaller subproblems, however here we have a different mathematical basis for it.

$$AB = A_1B_1 \cdot 10^{2t} + ((A_1 + A_2)(B_1 + B_2) - A1B1 + A_2B_2) \cdot 10^t + A_2B_2$$

This expression allows us to compute product of two n-digit numbers by computing 3 products of $\frac{n}{2}$-digit numbers, instead of 4 as in **Divide and Conquer algorithm**, at a cost of performing more additions and substractions along the way.

Source Code 3: Karatsuba multiplication

```
Number KaratsubaMultiplicator::multiply(const Number &n1, const Number &n2) const
{
    const size_t min_value = 1;// modify to optimize
    if (n1.size() <= min_value || n2.size() <= min_value)
    {
        std::unique_ptr<Multiplicator> fallback = std::make_unique<SchoolMultiplicator>();
        return fallback->multiply(n1, n2);
    }

    size_t m = std::min(n1.size(), n2.size()) / 2;

    //split numbers into two, one of which is of length m
    std::pair<Number, Number> a = n1.split(m);
    std::pair<Number, Number> b = n2.split(m);

    // A = A1 * t + A2
    // t = 10^m
    // AB = A1B1*t^2 + ((A1 + A2)(B1 + B2) - A1B1 - A2B2)*t + A2B2  - 3 multiplications
    Number a1b1 = this->multiply(a.first, b.first);
    Number a2b2 = this->multiply(a.second, b.second);
    Number a1_a2b1_b2 = this->multiply(a.first + a.second, b.first + b.second);

```

```
23      Number intermediate = a1_a2b1_b2 - a2b2 - a1b1;
24      a1b1.shift(2 * m);
25      intermediate.shift(m);
26
27      return a1b1 + intermediate + a2b2;
28  }
```

$$T(n) = 3T(n/2) + O(n)$$

An 'explanation' is similar as well: at each recursive call we have 3 subproblems, of size $n/2$, while additions and substractions can be done in a linear time. Applying **Master Theorem** one receives $T(n) = O(n^{\log_2 3}) \approx O(n^{1.6})$

Hypothesis: time complexity of **Karatsuba multiplication**= $\Theta(n^{\log_2 3})$

All these hypotheses are also supported by the materials of the ADS course [ADS course, Lecture 19, S. Shershakov, 2020]

2 Conditions of the experiment

2.1 A little bit about the implementation

The algorithms were implemented using the C++ programming language, with use of an auxilary class Number and a class Experimentator that was used to test to efficiency of the given multiplication algorithms.

Later in this page, **Divide and Conquer** algorithm will be simply called 'Caesar' to shorten quite a clumsy name.

Here notion of dashed lined should be read as 'mainly depends on', while inclusion of one area into another means that one is derived from another. All classes can be found at this link: https://github.com/Olenek/dsba-ads2020-hw1

2.2 General statements

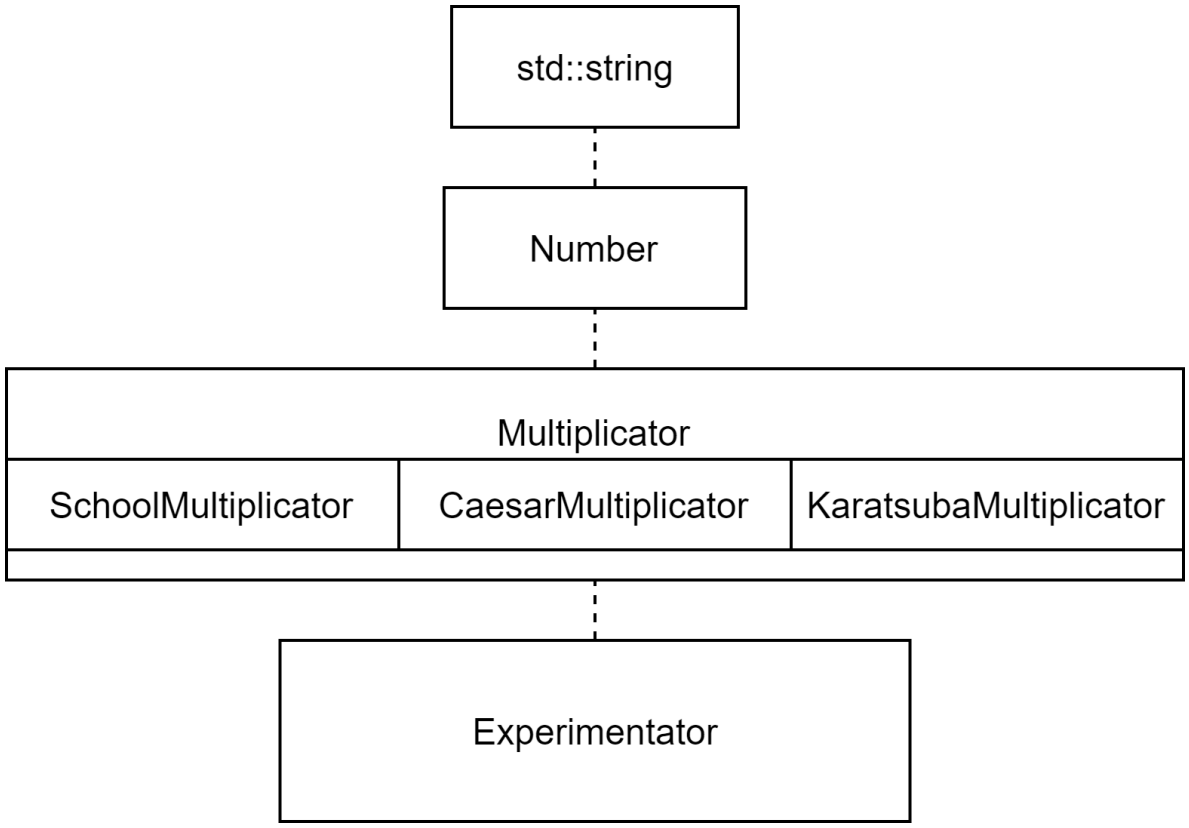
Every algorithm has been tested using the same procedure consisting of four steps:

1. Generate a pair of two random n-digit numbers
2. Run the tested algorithm on this pair of numbers, while measuring the running time, k times, in order to reduce the random error.
3. Write the mean running time of the algorithm to the output csv file.
4. Increment n and redo steps 1, 2, and 3.

For the experiment it was chosen to increment n in a pattern such that for all orders of numbers, n is changing in a way such that it iterates through an order in 10 steps. This procedure was executed with k = 3 with use of two additional methods: one to generate pseudorandom n-digit number and one to measure run-time of a function.

Source Code 4: Time measurement method

Figure 1: The class diagram




```
double Experimentator::measure_time
  (const std::function<Number(const Number &, const Number &)> &f,
   const Number &n1,
   const Number &n2)
{
    auto start = std::chrono::high_resolution_clock::now();
    f(n1, n2);
    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);

    return duration.count();
}
```

In this method, package <chrono> is used to measure time in milliseconds between the beginning of the function execution and its end.

Source Code 5: Random number generator

```
void Number::generate_random(size_t k, int time_seed)
{
    std::string randStr;
```

```
    std::srand(std::time(0) + time_seed);
    randStr.reserve(k);
    for (size_t i = 0; i < (k - 1); i++)
    {
        randStr.push_back(char(rand() % 10));
    }
    randStr.push_back(char((rand() % 9) + 1)); // adding the leading non-zero digit
    Number ans = Number(std::move(randStr));
    *this = ans;
}
```

In this function a random number is generated by creating a vector consisting of random digits, (with restriction on leading zero). Parameter **time_seed** ensures that **rand()** will be unpredictable from call to call.

Remark: multiple measurements on the same problem size are not shown on the diagram to make it more readable.

2.3 In brief

Finally, it was decided to perform calculations of the running time of all three algorithms on problem sizes varying from 1 digit to 10001 digits with total of 73 data points. And then perform calculations of the running times of **Karatsuba** algorithm and **School-grade** algorithm on problem sizes varying from 1 digit to 100001 digits with total of 47 data points, in order to empyrically find when **Karatsuba** algorithm 'beats' **School-grade** algorithm in terms of the running time.

3 Processing of the results

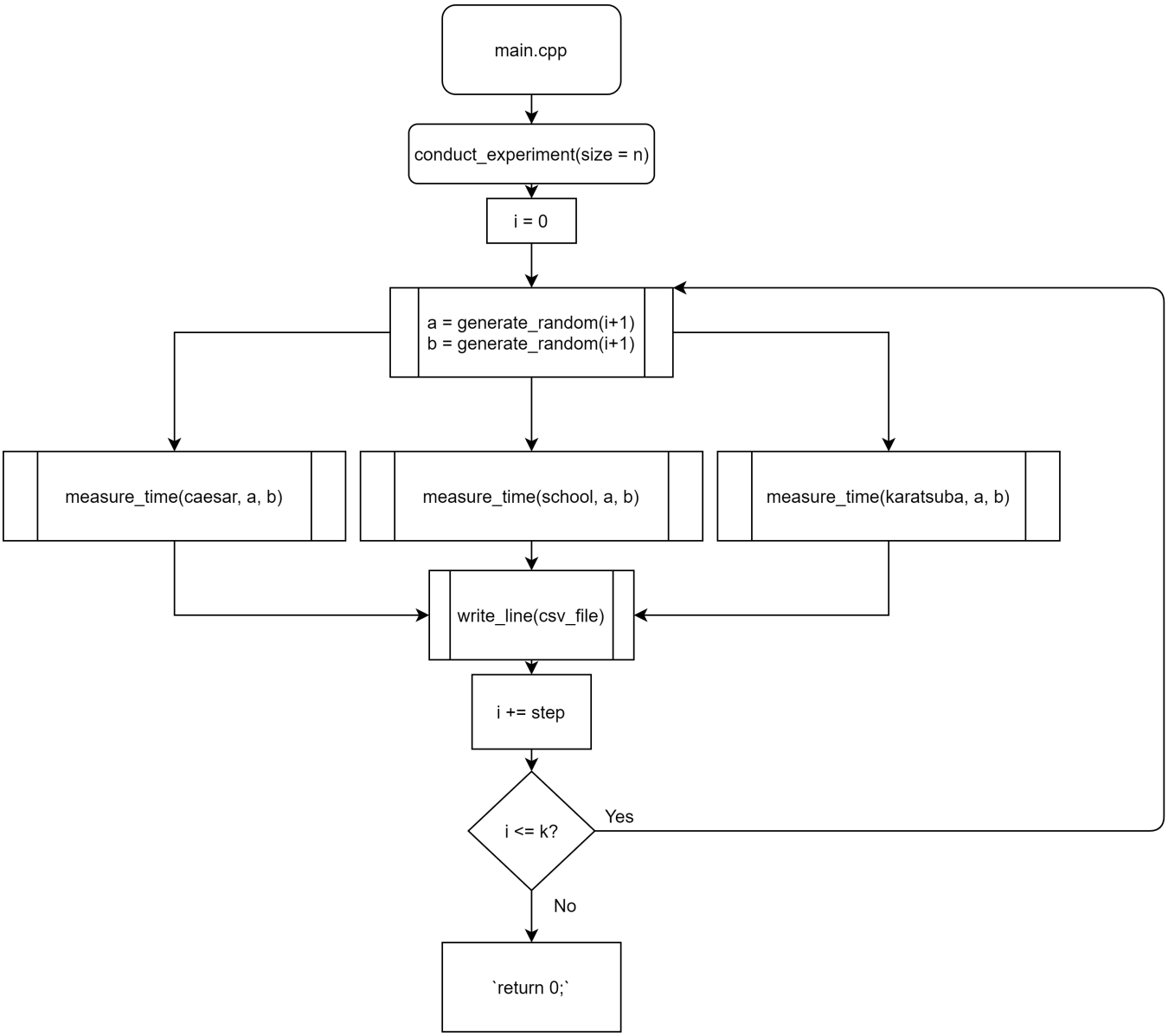
The graphs were made with use of Python and its module Pandas. All results of the experiment can be found in GitHub repository in directories '/result' and '/graphs'.

The graph illustrates that **Divide and Conquer** (line 'Caesar (ms)') is always outperformed by its opponents. It can be explained as following: Divide and Conquer algorithm's depth of recursion is quite big. It is logarithmic and on each layer of recursion it calls itself 4 times, even when multiplying quite small numbers. One of the possible ways to optimize this algorithm is to allow it to multiply smaller numbers like **School-grade** multiplication does, while leaving multiplication of bigger numbers the same: calling itself 4 times with smaller problem size. This topic will be addressed at the end of this article.

From this graph it is impossible to tell whether **Karatsuba** multiplication is more efficient or no. However another graph may help do this:

This graph was made from file 'intersection.csv' that was made with a different step-mechanism to make more frequent measurements. Details can be found in the repository. Here it can be seen that starting from around 90000 digits in a number, **Karatsuba** performs better than **School-grade**, however it does not mean that out hypotheses regarding time complexities are correct. We need another way to prove

Figure 2: The diagram of the testing process



them.

$$f(n) \in \Theta(g(n)) \iff \exists k_1, k_2 \in \mathbb{R}_+ : \exists n_0 : \forall n > n_0 \quad k_1 g(n) \leq f(n) \leq k_2 g(n)$$

Figure 3: Overall picture

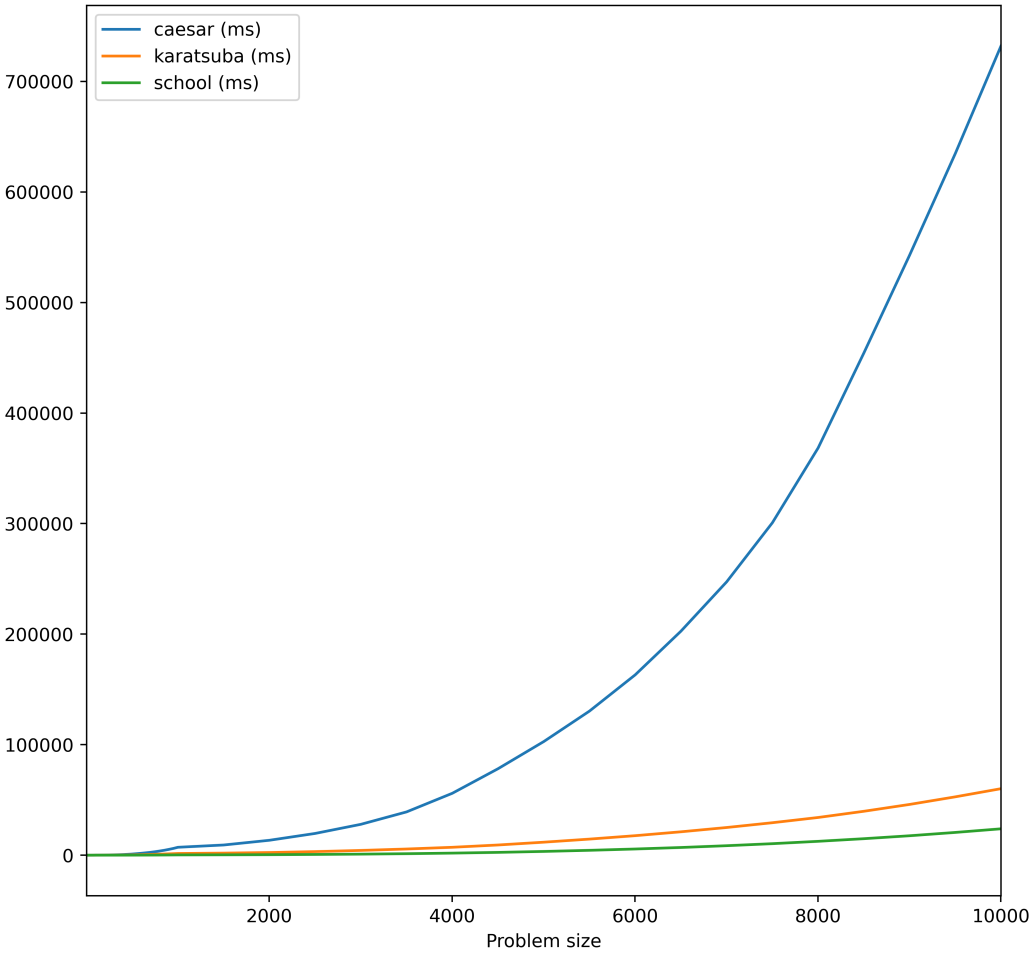


Figure 4: Intersection point

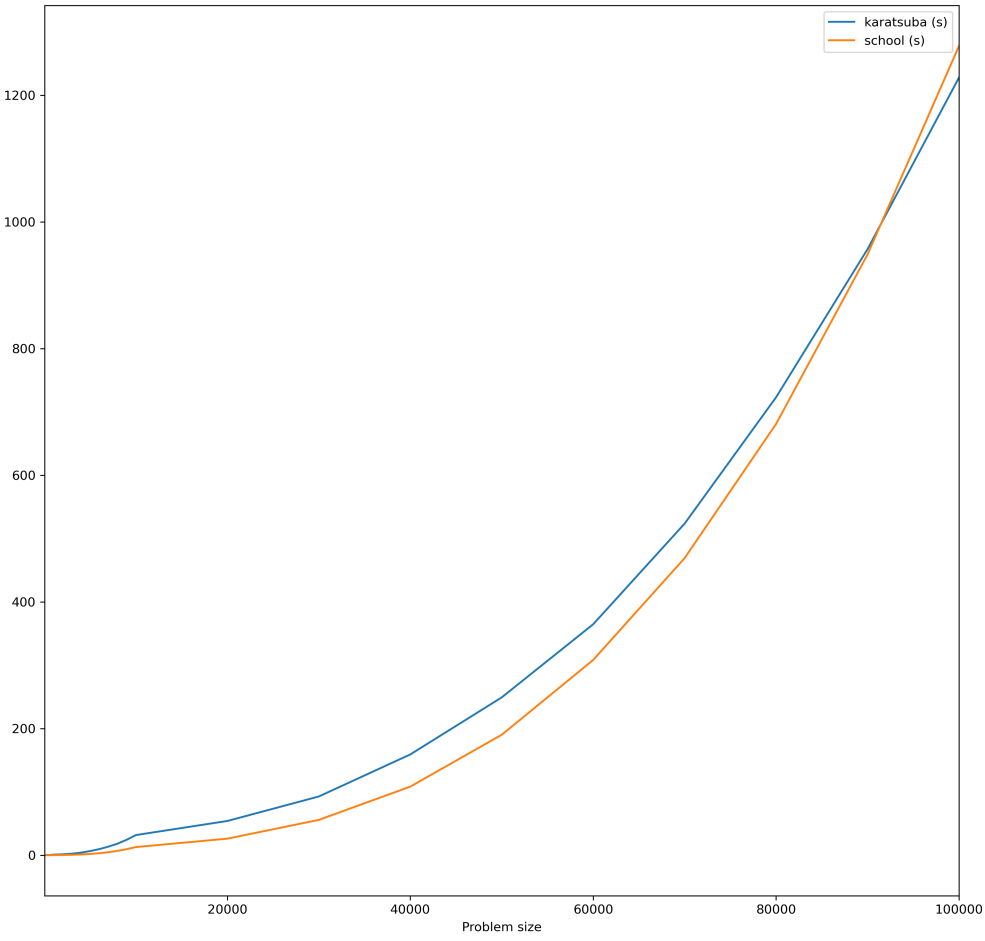


Figure 5: School-grade $\in \Theta(n^2)$

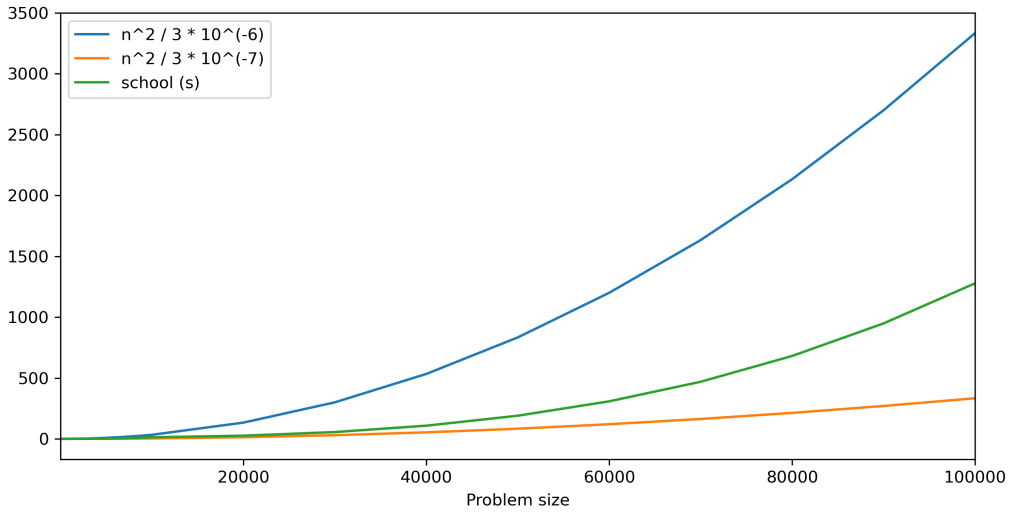


Figure 6: Karatsuba $\in \Theta(n^{\log_2 3})$

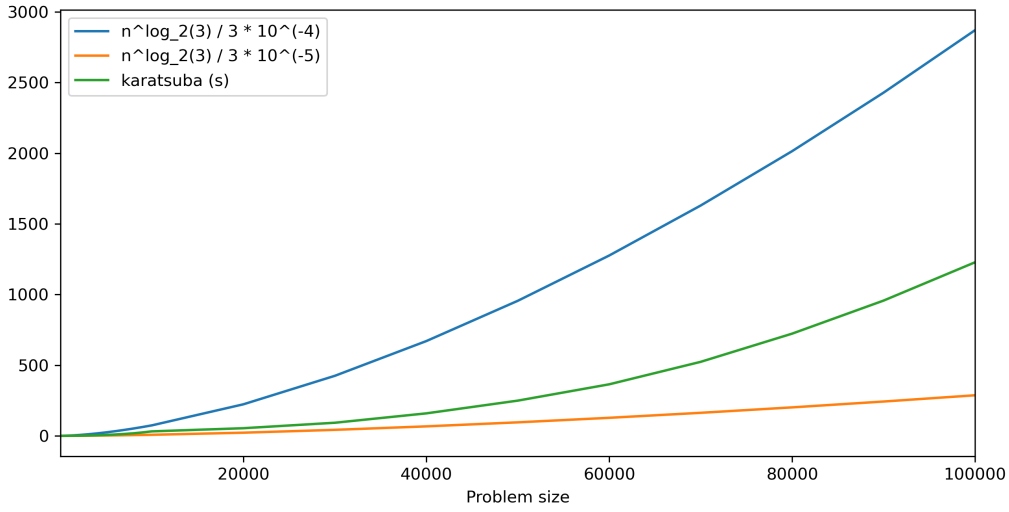
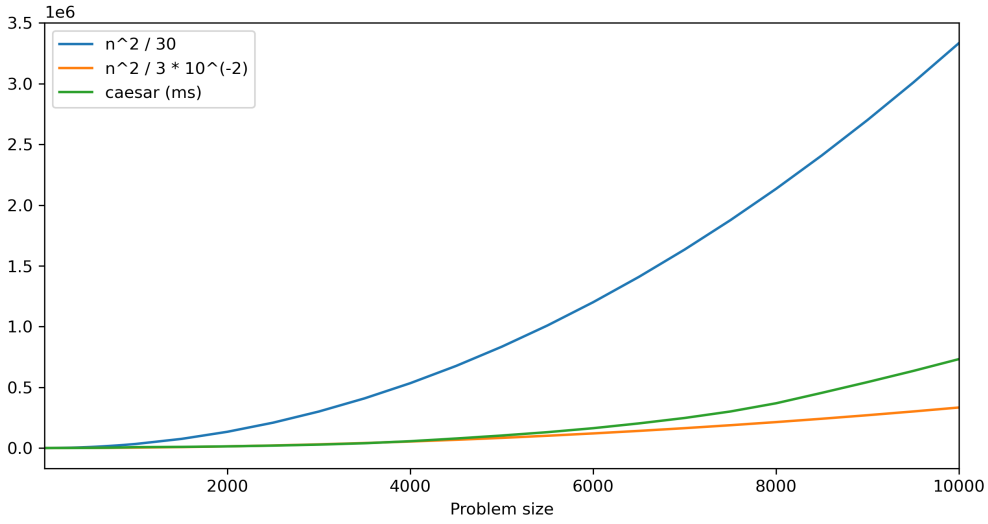


Figure 7: Divide and Conquer (Caesar) $\in \Theta(n^2)$



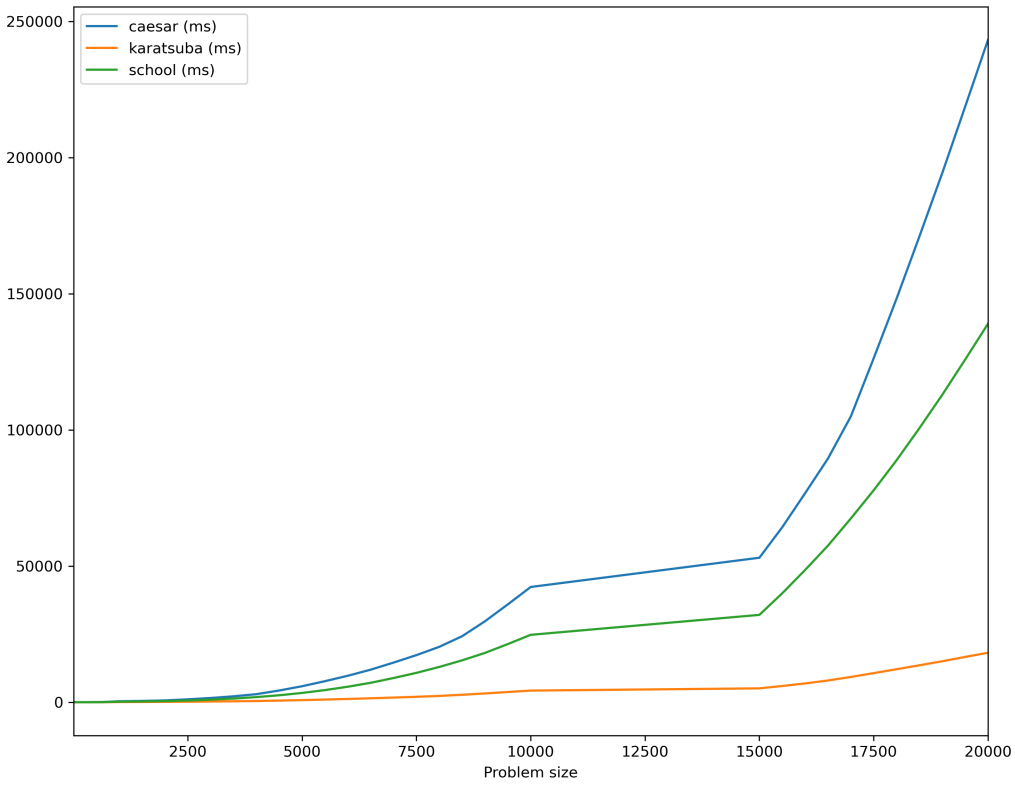
These graphs allow us to say that our initial hypotheses about time complexities of the given algorithms indeed hold (at least for the ranges examined in this article), as the curves are squezeed between x^2 or $x^{\log_2 3}$ with some coefficents that are displayed in the graphs.

4 Conclusion and optimisations

4.1 Possible optimisation

As it was previously mentioned, for big numbers **Divide and Conquer** algorithm fails to be effective(in terms of constant multiplier of complexity, not asymptotical values), however this can be improved by changing scheme: 'divide numbers into equal parts' $\rightarrow$ 'compute their products' $\rightarrow$ 'sum up the results' to 'if numbers are small' $\rightarrow$ 'compute their product'; 'else' $\rightarrow$ 'do the previous scheme'. This logic can also be applied to the Karatsuba algorithm also and in theory make them work faster. In the graph below it can be seen that **Divide and Conquer** actually appears to be viable, while **Karatsuba** algorithm works faster than **School-grade** on the whole range from 1-digit numbers to 20000-digit numbers.

Figure 8: Algorithms with fallback from 16 digits



4.2 Conclusion

What this article has established:

- **Divide and Conquer** multiplication: $T(n) \in \Theta(n^2)$
- **School-grade** multiplication: $T(n) \in \Theta(n^2)$
- **Karatsuba** multiplication: $T(n) \in \Theta(n^{\log_2 3})$, works faster for $n > 90000$, when not optimised.

In the end, it is possible to make all algorithms work better, for instance, now class 'Number' is based on std::string and its notions, however starting from C++17 std::string_view was introduced that would probably allow Number::operator+= and Number::split work faster significantly. Other direction of improvement may be to run all algorithms not 3 times to reduce error but more, for example 10, as well as to choose a wider range of problem sizes and make measurements more frequently.