Comparison of Multiplication Algortihms

Oleg Vasyutin 191-1
Supervised by George Piatskiy

April 26, 2020

1 Purpose of the task

Estimate the time complexity of three different multiplication algorithms experimentally. The problem might be divided into several steps:

1. Get the asymptotic estimates for recurrence relations corresponding to an algorithm by applying master theorem and any other appropriate means of estimation.
2. Implement given algorithms by using C++ language. Conduct an experiment and write the data to the .csv file for further research.
3. Draw conclusion. Examine the received data by creating a plot with graphs for visual representation and compare the runtime of the algortihms with the expected runtime complexities.

Algorithms description

To begin with, let us observe a detailed description of the given algorithms to understand what needs to be done, in order to implement them.

Grade-School Multiplication[1]

Classic algorithm of multiplication taught at schools. Given two numbers. Start from the last digits of numbers (obtained by division by modulo base of a number), multiply the digits and divide by modulo base. Write the result down the imaginable bar and "memorize" one if the result of multiplication is more than 10. Then, in the first number, move to the left, multiply by the same digit in the second number and repeat the process again. When no digits are located on the left, move to the last digit of the first number and pick a digit to the left of the initial digit in the second number. Start writing the result one digit to the left under the previous result. Obtain a collection of numbers, which should be later added to each other to receive the result.

[1] hereinafter referred to as "School Multiplication"

Divide and Conquer Multiplication[2]

A more complex approach to miltiply two numbers. Given two numbers: a and b of equal lengths of n.

1. Divide number into 2 parts of lengths $\frac{n}{2}$, such that:

$$a_1^{n/2} + a_2 = a$$
$$b_1^{n/2} + b_2 = b$$

2. Therefore, we can easily miltiply the numbers as follows:

$$a * b = (a_1^{n/2} + a_2) * (b_1^{n/2} + b_2) =$$
$$= (a_1 * b_1)^n + (a_1 * b_2 + a_2 * b_1)^{\frac{n}{2}} + a_2 * b_2,$$

   where operation of multiplication, denoted as *, is implemented as another recursive call of DaC Multiplication.

3. When the case of multiplying single digit numbers is reached, basic knowledge of miltiplication table is applied to receive the result.

Karatsuba Multiplication

An even more complex approach for multiplication. Throwbacks to DaC Multiplication algortihm with some advencements.
Let us observe that:

$$(a_1 + a_2) * (b_1 + b_2) = a_1 * b_1 + a_1 * b_2 + a_2 * b_1 + a_2 * b_2$$
$$\implies a_1 * b_2 + a_2 * b_1 = (a_1 + a_2) * (b_1 + b_2) - a_1 * b_1 - a_2 * b_2$$

Therefore, final formula for applying Karatsuba algorithm might be presented in this form:

$$a * b =$$
$$= (a_1 * b_1)^n + ((a_1 + a_2) * (b_1 + b_2) - a_1 * b_1 - a_2 * b_2)^{\frac{n}{2}} + a_2 * b_2,$$

where operation of multiplication, denoted as *, is implemented as another recursive call of Karatsuba Multiplication. Even though five recursive calls may be observer, there are only three of them which are different from the others.

2 Estimation

1. To determine time complexity for **School Multiplication** let us inspect the operations, which are done. For every miltiplication of a digit by digit a constant number of operations k_1 is done, which includes division with

[2] hereinafter referred to as "DaC Multiplication"

remainder, addition and multiplication. In total, the number of operations equals $k_1 * n^2$. As for final addition of n^2 numbers, the number of operations k_2, including the addition of two digits, is constant too. These are the same division with remainder and one more addition of 1 or 0, depending on the previous addition. So, the total number of operations done for the algorithm does not exceed $(k_1+k_2)*n^2$. Hence, the estimated time complexity for School Multiplication:

$$T(n) = \mathcal{O}(n^2)$$

2. In case of **DaC Multiplication**, the estimation is done by constructing a recurrence relation. As we know, in order to mupltiply by this algorithm, one is ought to split two numbers into four parts and make four recursive calls. Basically, one divides the problem into two sub-prblems of a two times size and every times one does it four times. In addition, the number of extra operations which is done on each step is $\mathcal{O}(n)$. Hence, we obtain a relation:

$$T(n) = 4T(\frac{n}{2}) + \mathcal{O}(n)$$

And this will satisfy case 1 of Master Theorem, therefore the runtime complexity of this algorithm is

$$T(n) = \Theta(n^2) \implies T(n) = \mathcal{O}(n^2)$$

3. Finally, the **Karatsuba Multiplication**. Everything is really similar to the DaC Multiplication, with the only exception being the number of calls that thi algorithm makes, which is three. So, we obtain:

$$T(n) = 3T(\frac{n}{2}) + \mathcal{O}(n),$$

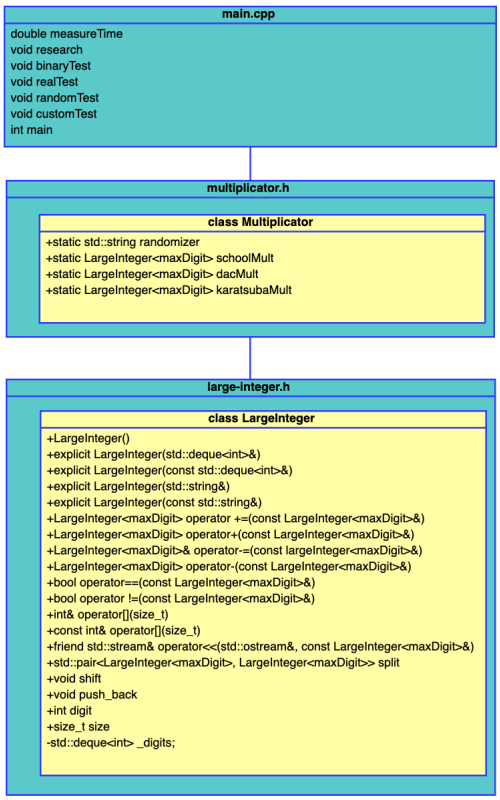
which satisfies case 1 of Master Theorem. Hence, the runtime complexity of Karatsuba Multiplication is:

$$T(n) = \Theta(n^{1.59}) \implies T(n) = \mathcal{O}(n^{1.59})$$

Now it is time to conduct an experiment and see, if the results would fulfill out expectations!

3 Implementation

Here is a basic representation of the structure of the project:



Integers in C++ language have their own limits, as they do not exceed huge length of numbers, such as 1000 of digits. That is why it was decided to implement a new class for storing large numbers in *large-integer.h*. In order to store numbers if various bases it was considered to use a template which exactly defines the base. It does not overcomplicate the class as it only effects the process of sum, where one is needed to divide by the base. The container for storing data std::deque due to the reasons explained later. All the numbers in the container are stored in reversed order for further convenice of iterating over the deque in operations like plus or minus.

Most of functions are basic getters, setters or overloaded operators, which do not contain any complex idea behind them. The exceptions here are *shift()* and *split()* methods. Both of them are used in algorithms that use divide and conquer approach. The second one is dedicated to divide the problem into subproblems for performing needed operations. As for the *shift()* function, it substitutes the multiplication by 10 to any power. Instead, it writes zeroes in the beginning, as digits are stored in reversed order. The operation is done in constant time in std::deque, but needs a lot more to perform in case of

using std::vector, for instance. In addition, deque already had internal function *push_front* to add elements.

In a different header file one may find the class Multiplicator, which contains implementations of all the previously discussed algorithms and a method randomizer() that produces an std::string, which is further used to be pushed into a number.

One major thing that can be done to make the algorithm faster is choosing appropriate base case, as Karatsuba Multiplication and DaC show really slow results when multiplying small numbers by the corresponding method. So, it one decides to boost their performance, they might conduct an experiment to determine the best possible case.

Apart from that, the implementations are really to similar to their descriptions, yet School Multiplication is functioning in a different way. Instead of creating n numbers to sum them up, the implemented schoolMult instantly adds the result of multiplying digits to the resulting number. Which increases the perofrmance drastically.

Finally, file *main.cpp* contains various tests, including the *realTest*, which uses a special .csv with 1000 randomly generated tests from python script. It is really beneficial for testing, as in python integers don't have size limits, tests are claimed to be true.

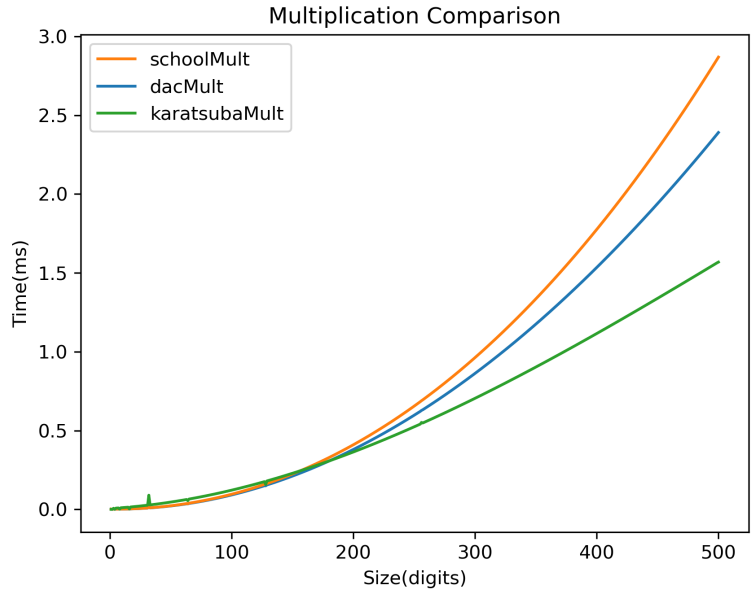
One more feature is the use of QueryPerformance from windows library. It was discovered during tests that standard *clock()* has really bad precision, so, at small number of digits the time measured could be zero. But such would still holf for cases of up to 80 due to the algorithms being so fast. That is why QueryPerformance was used.

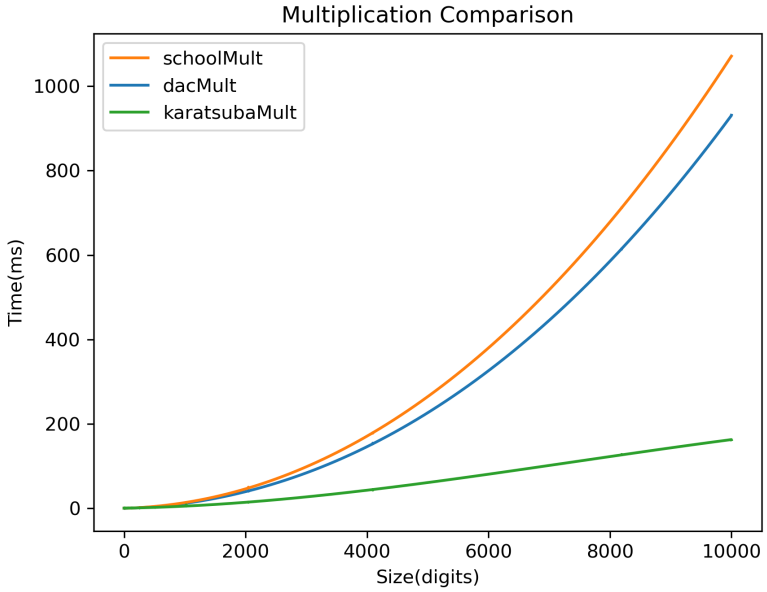
The most important function that was contained in *main.cpp* is *research()*, as it wrote the results to the specific .csv file, which was later used for creating a plot in python by using library matplotlib. In order to decrease the duration of the experiment and to later interpolate the graph for obtaining a more precise result the step between the numbers was logarithmic with the inclusion of the last number.

The graphs were interpolated using the polynomial of degree three, so that the graph would go through the recieved points, rather then being really close to the points, as if one was interpolating using Bézier curves.

4 Conclusion

Two general tests were done. As it can be seen from the graph, DaC Multiplication and School Multiplication are really simillar, as predicted. And Karatsuba Multiplication becomes faster than School Multiplication at around 160, being way faster at larger scale of numbers, as it can be seen on the second graph. Which fits the predictions as we wanted it to be


Multiplication Comparison
schoolMult
dacMult
karatsubaMult
Time(ms)
Size(digits)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
100
200
300
400
500


Multiplication Comparison
schoolMult
dacMult
karatsubaMult
Time(ms)
Size(digits)
0
200
400
600
800
1000
0
2000
4000
6000
8000
10000