Analysing multiplication algorithms

Student: Nikolay Kukarkin

Supervisor: Kirill Rudakov

Submission date: 26.04.2020

Group: DSBA 192-2

Problem statement:

I need to measure the runtime of three distinct algorithms: Grade School Multiplication, Divide and Conquer, Karatsuba Multiplication and to compare the results with theoretical estimations of these very algorithms.

Theory:

1. Grade School Multiplication:

This is the most simple and most popular method of multiplication. Easy to be used when multiplying small numbers. Commonly used in middle classes.
Complexity: O(n^2)

2. Divide and Conquer Multiplication:

Algorithm, based on recursion tree (T(n) = 2 T(n / 2) + O(n)). The idea is to continue splitting a given number into two smaller ones, until we obtain a “base case”, which will be computed by a Grade School Multiplication.
Complexity: O(nlog(n))

3. Karatsuba Multiplication:

Also a recurrent algorithm (3T(2n) + O(n)). Better then Divide and Conquer, as during splitting, in each cycle Karatsuba only makes 3 multiplications, while Divide and Conquer need 4.
Complexity: O(n^log(3))

Research plan:

1. Create a custom class “Number”. The internal representation is a vector (An element of vector is a single digit in the given number. Then, I need to provide it with all necessary methods and operations.

2. Create a custom class Multiplication. Equip it with three methods, one per multiplication algorithms. It also must have a method, creating a number of class “Number” with k digits.

3. Then, in the main function, test all algorithms if they all return correct products. Create time test and store them in a csv file.

4. Finally, using some python IDE make a visual representation of obtained results with a library “matplotlib”.

Implementation details

1. I have a class Number, with a protected field “data” (given number). For this class I have:

- 4 overloaded operators (“+”, “-”, “[]”, “constant []”) ;
- 8 methods, needed for representing Number like a vector (e.g. “*push back*”, “*size*”, “*reverse*”, etc)
- 2 methods needed directly for algorithms. The first one is “*Split*”, which splits an input Number and returns a pair of Numbers. Second is “*ten_pow*”, which simply adds zeroes to the end of number.

2. In the class Multiplicator, I have four static method mentioned before. All, except for the first one, have two numbers in the input (*num1, num2*):

- Randomizer

Has an integer k input. Creates a number Number with k digits in it. Uses *uniforim_int_distribution* and *rand().*

- Grade_School_multiplication

I created two versions of this algorithm.

The first one, a simpler one, multiplies all digits 1 by 1, and adds zeroes, corresponding to the size of given numbers, then with an overloaded operator “+” adds all possible product together and returns result;

The second one (commented in the code), much more complicated, it also multiplies digits 1 by 1, but instead of adding zeroes, it the keeps track of custom indexes (*pr1, pr2*), then to add corresponding digit in result number (*res*), so it never loses any remainders after dividing by 10. Then it reverses the number, cleans up unwanted zeroes and returns the final result (*fin*)

- DAC (Divide and Conquer)

After splitting, if we get two pairs, one for each input number. Elements of this pair are either the same size, or the second one is bigger. It depends on the size of the input number. For instance, if we have “1234”, it will split it into “12” and “34”, but if we have “123”, we will obtain “12” and “3”.

Then, by the following formula, we continue splitting both of these elements and adding zeroes with (*ten_pow*) until getting a base case (In my case, it is equal to 1), which will return Grade school multiplication algorithm.

- $\mathbf{a} = a_1 a_0$ and $b = b_1 b_0$
- $\mathbf{c} = a * b$
  $= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$
  $= c_2 10^n + c_1 10^{n/2} + c_0$

- Karatsuba

Almost the same algorithm as the previous one, but before splitting, I add zeroes, in case size is odd, then there won't be any unexpected mistakes.

- $T_0 = A_0 B_0$
- $T_1 = (A_1 + A_0)(B_1 + B_0)$
- $T_2 = A_1 B_1$
- $C = T_2 10^{2k} + (T_1 - T_0 - T_2)10^k + T_0$

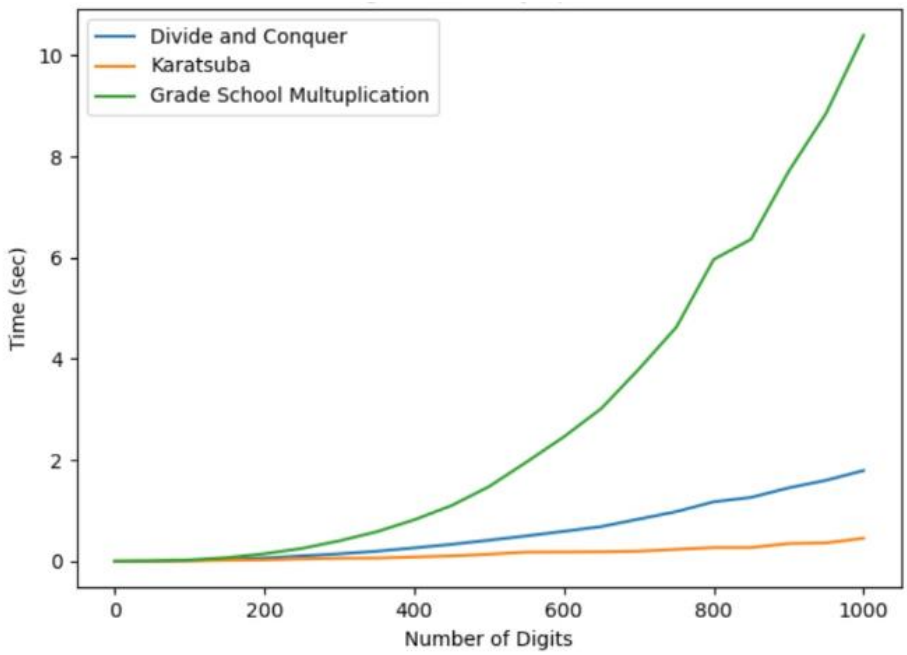
Also the formula is a bit different. (3 multiplications instead of 4)

(You can find my code here: https://github.com/Kukarkin/dsba-ads2020-hw1)

Results

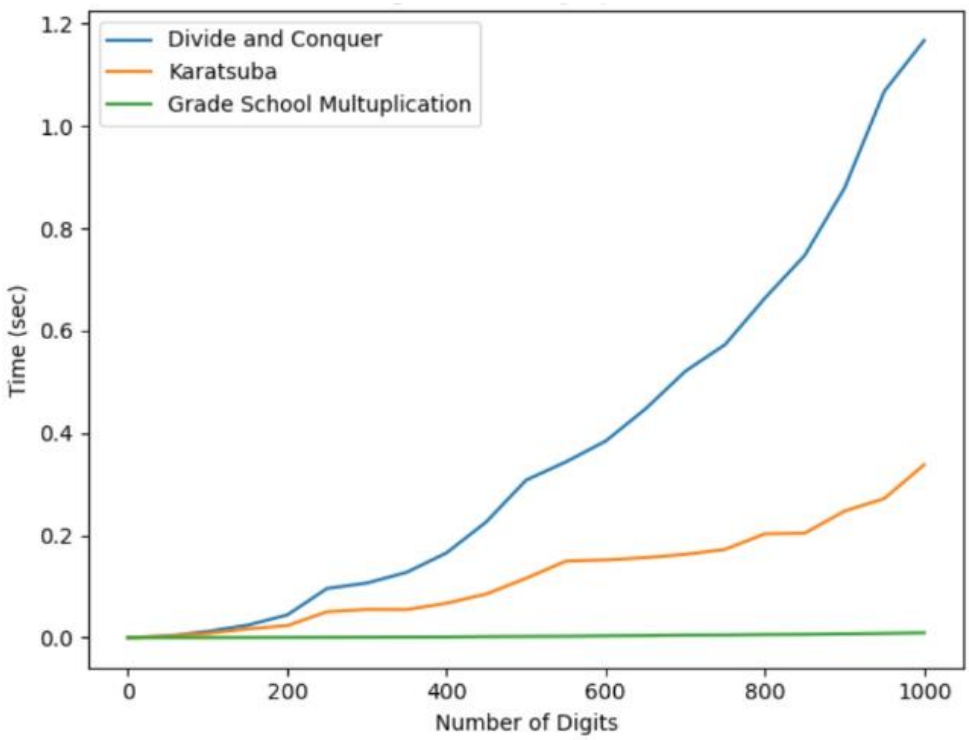
I made a 3 different plots, depending on the base cases and the type of grade school multiplication.

Base case 1, standard GSM



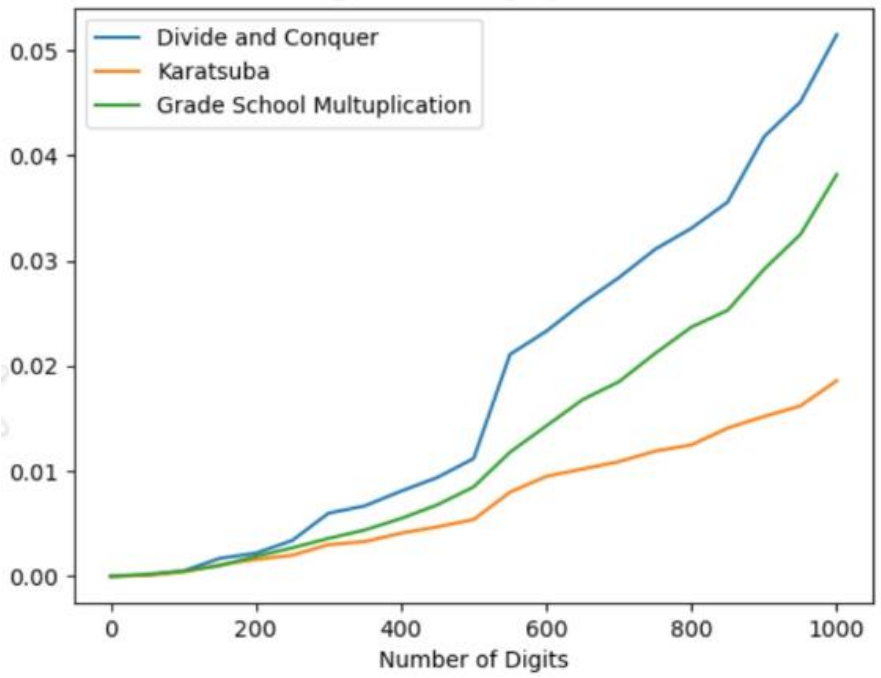
Here, we see that Grade School Multiplication algorithm is the worst one. Karatsuba is the best and Divide and Conquer is in between. Considering the complexity, it looks as the most realistic variant, relatively to each other.

Base case 1, advanced GSM



Here, Grade School Multiplication works the fastest, but this seems ridiculous, as the complexity is O(n^2), but time spent is almost 0 sec. (Still, all calculations are correct) Nevertheless, this may take place, as complexity doesn't take into account all other methods considered before. (Karatsuba can exceed GSM at about 50.000 digits, but it takes too long to compute such great number)

Base case 64, advanced GSM



The solution to the problem mentioned just before is pretty simple, but not obvious, and not honest, in terms of this task. If we change base case of Karatsuba from 1 to 64, then it will outrun any other algorithm in each point in time. But even in this case, Divide and Conquer is still the worst (why it may happen is already explained).

Conclusion

The results are not very clear. Depending on the base case of Karatsuba and Divide and conquer, and the chosen realization of Grade School Multiplication, runtime can differ very much.

Obviously, it is not very fair to use any other base case, but 1 (According to the task, of course). Still, for any algorithm the most efficient base case is 128. To improve the project, it is possible to try to get rid of such methods as “pop_front” and “Split”, because they may consume much time, while managing zeroes.