BIG HOMEWORK

Comparison of time complexities of different multiplication algorithms computing (Grade School Multiplication, Divide-and-Conquer and Karatsuba's algorithms).

Author name: Sharara N.

Supervisor name: Shershakov S.A

We need to estimate the time complexity of different multiplication algorithms.

Grade School Multiplication takes about n^2 one-digit operations (at most n^2 multiplications, at most n^2 additions, and then one need to add n different 2n numbers), so this algorithm scales like n^2.

Divide and Conquer Approach is based on breaking a big problem into smaller ones (sub-problems), then solve those sub-problems individually and, finally, bring the solutions of sub-problems into a solution of the problem. For the Multiplication we use the formula xy = (a * 10^(n/2 + b)(c * 10^(n/2) + d) = ac * 10^n + (ad + bc)*10^(n/2) + bd (basically, we take one n-digit multiply and break it into four (n/2)-digit multiplies, and break those multiplies more, if needed). The running time of DaC is at least n^2 as every pair of digits is multiplied together separately.

Karatsuba's algorithm also is based on breaking a big problem into smaller ones. We actually still use the formula from DAC algorithm xy = (a * 10^(n/2 + b)(c * 10^(n/2) + d) = ac * 10^n + (ad + bc)*10^(n/2) + bd. But (ac+bc) we compute as just one number without finding ac and bc independently, as (ad+bc) = (a+b)(c+d) – ac -bd. In this case we just break the "big problem" of size n into three smaller ones of size n/2 and so on, if needed. We need log2(n) times to cut the problem n to get down to problems of size 1. So, we get 3^(log2(n))=n^(log2(3)) problems of size 1, which is close to n^1.6. As we can see, Karatsuba's algorithm scales better comparing to DAC and GSM.

Implementation details.

First, I created class Number to represent an integer consisted of a large number of digits (I used std::vector for storing digits).

We were required to generate a random number but in case if we use just std::rand, we will get always the same result. Computers are generally incapable of generating random numbers, they, instead, simulate randomness. I used function std::srand() that seeds the pseudo-random number generator used by std::rand(), but each time rand() is seeded with the same seed, it will produce the same product. So, basically, we need to change the value every time. If the we use std::srand(time(NULL)) seed will be the current time.

GradeSchoolMultiplication

First, we make Numbers Equal in case if their sizes are different just my adding 0 in the beginning of the number with the least size.

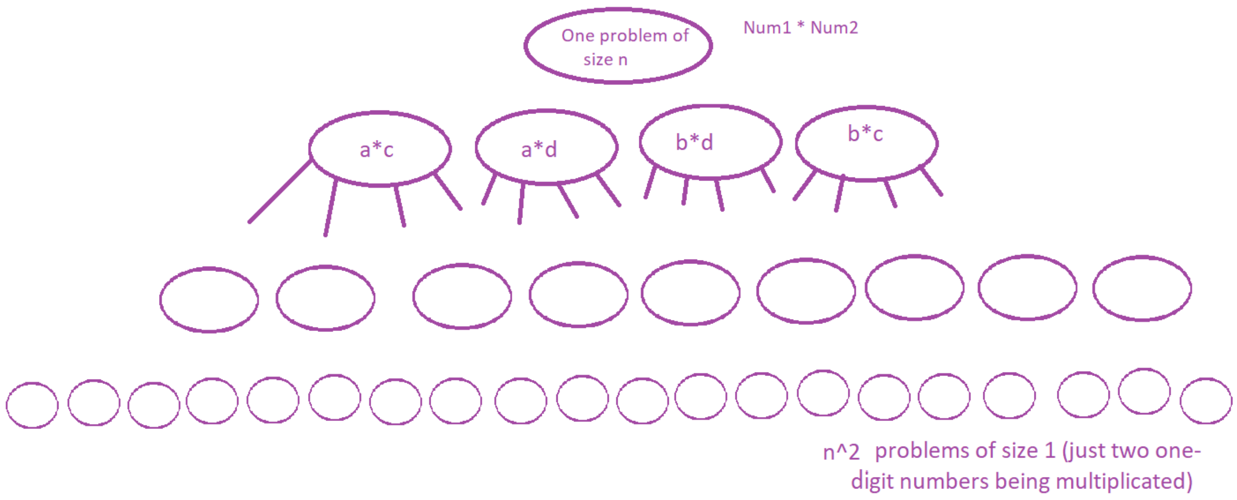
When multiplying two numbers of the same size, we can for sure say that the product will be at maximum the size of 2n (ex., 99*99 = 9801).

So, I created a vector "answer" of size 2n, which is filled of 0's.

The algorithm:

As in school while multiplying two numbers, we start from the end of one number and then multiply it by all digits of second number and so on. So, used two for cycles going from the last digits of the numbers to the first and multiply two one-digit numbers. But sometimes multiplying two digit numbers gives us a two-digit number, but in our vector (which stores every digit) every element of it should be just one digit. So, we need to check whether the product is greater than 9, and if so just like in school add the remainder to the following digit of the answer.

Divide-and-Conquer


One problem of size n
Num1 * Num2
a*c
a*d
b*d
b*c
n^2 problems of size 1 (just two one-digit numbers being multiplicated)

We break both n-digit integers using:

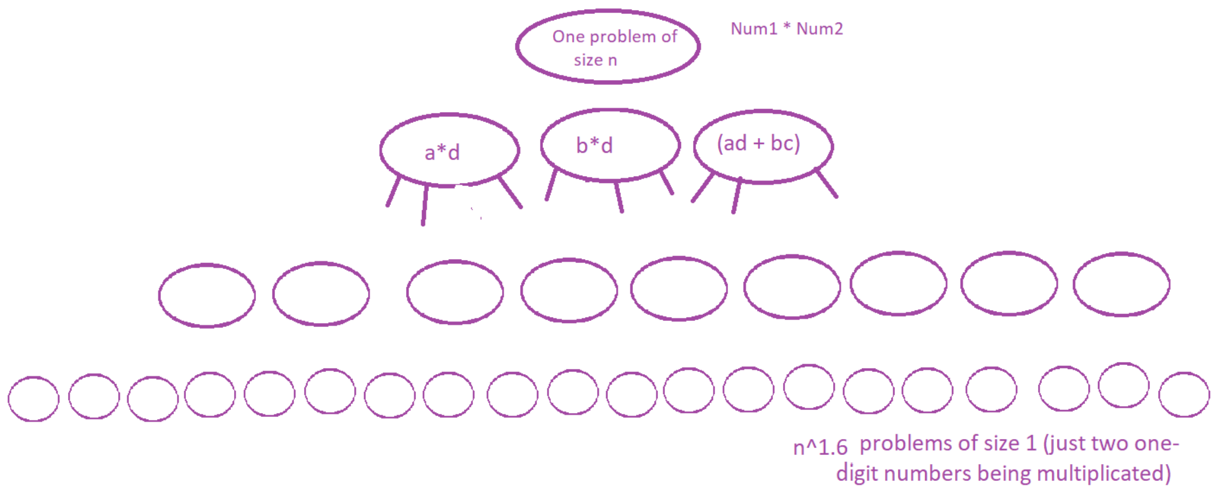
$$[x_1 x_2 \cdots x_n] = [x_1 x_2 \cdots x_{n/2}] \times 10^{n/2} + [x_{n/2+1} x_{n/2+2} \cdots x_n]$$

So, we represent the multiplication like x*y = (a*10^(n/2) + b)(c*10(n/2) + d) = **ac***10^(n) + (**ad** + **bc**)*10(n/2) + **bd**. Then, we get 4 problems (ac, bc, ad, bd) and we will also divide them into using the previous formula until we get just n^2 small problems (each of them is just multiplication of two one-digit numbers) that we can multiply by GradeSchoolMultiplication.

So, the base case will be, when the size of the problem is 1 and compute it by GSM(so we can still put it in the vector). We recursively find ac, bc, ad and bd, and, finally, put everything in the formula.

Karatsuba algorithm

Implementation of this method is similar to the DAC one. We just break our problem into tree and so on.


One problem of size n
Num1 * Num2
a*d
b*d
(ad + bc)
n^1.6 problems of size 1 (just two one-digit numbers being multiplicated)

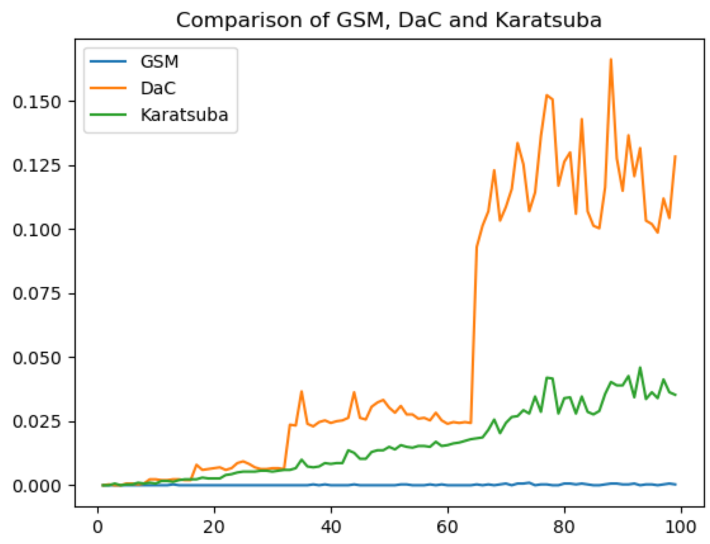
The base case in also 1. After recursively "dividing" we will get n^1.6 small subproblems of size 1, which we solve by GSM.

Code: https://github.com/NinCha1

Results

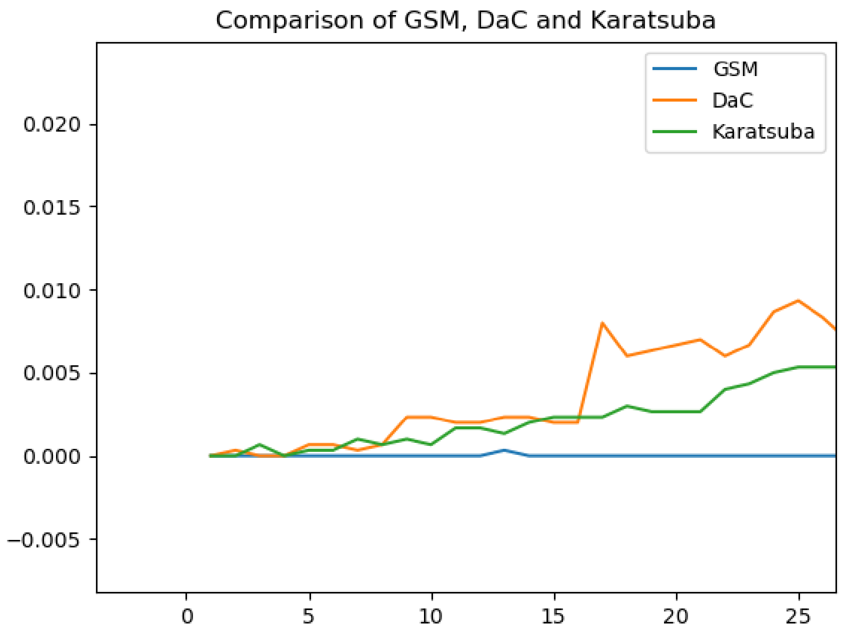
Using the results that method Average (counts the average time of algorithms and puts it in the csv-file) I've constructed a table (using matplotlib). My GSM algorithm had the best time complexity and the difference in runtime with smaller or bigger inputs is the least.

Size of problem	GSM	DAC	Karatsuba
1	0	0	0
2	0	0.000333333	0
3	0	0	0.000666667
4	0	0	0
5	0	0.000666667	0.000333333
6	0	0.000666667	0.000333333
7	0	0.000333333	0.001
8	0	0.000666667	0.000666667
9	0	0.00233333	0.001
10	0	0.00233333	0.000666667
11	0	0.002	0.00166667
12	0	0.002	0.00166667
13	0.000333333	0.00233333	0.00133333
14	0	0.00233333	0.002
15	0	0.002	0.00233333
16	0	0.002	0.00233333
17	0	0.008	0.00233333
18	0	0.006	0.003
19	0	0.00633333	0.00266667
20	0	0.00666667	0.00266667
21	0	0.007	0.00266667
22	0	0.006	0.004
23	0	0.00666667	0.00433333
24	0	0.00866667	0.005
25	0	0.00933333	0.00533333
26	0	0.00833333	0.00533333
27	0	0.007	0.00533333
28	0	0.00633333	0.00566667
29	0	0.00633333	0.00566667
30	0	0.00666667	0.00533333
31	0	0.00666667	0.00566667
32	0	0.00633333	0.006
33	0	0.0236667	0.006
34	0	0.0233333	0.00666667
35	0	0.0366667	0.01
36	0	0.024	0.00733333
37	0.000333333	0.023	0.007
38	0	0.0246667	0.00733333
39	0.000333333	0.0253333	0.00866667
40	0	0.0243333	0.00833333
50	0	0.0303333	0.015
51	0	0.0283333	0.014
52	0.000333333	0.031	0.0156667
53	0.000333333	0.0276667	0.015
54	0	0.0276667	0.0146667
55	0	0.026	0.0153333
56	0	0.0263333	0.0153333
57	0.000333333	0.0253333	0.015
58	0	0.0283333	0.017
59	0.000333333	0.0253333	0.0153333
60	0	0.024	0.0156667
61	0	0.0246667	0.0163333
62	0	0.0243333	0.0166667
63	0	0.0246667	0.0173333
64	0	0.0243333	0.018
65	0.000333333	0.093	0.0183333
66	0	0.101333	0.0186667
67	0.000333333	0.107	0.0216667
68	0	0.123	0.0256667
69	0.000333333	0.103333	0.0203333
70	0.000666667	0.108667	0.0243333
72	0.000666667	0.133667	0.027
73	0.000666667	0.125333	0.0293333
74	0.001	0.107	0.028
75	0	0.114333	0.0346667
76	0.000333333	0.136333	0.0286667
77	0.000333333	0.152333	0.042
78	0	0.150667	0.0416667
79	0	0.117	0.028
80	0.000666667	0.126333	0.034
81	0.000666667	0.13	0.0343333
82	0.000333333	0.106	0.028
83	0.000666667	0.143	0.0346667
84	0.000333333	0.107	0.0286667
85	0	0.101333	0.0276667
86	0	0.100333	0.029
87	0.000333333	0.116333	0.0356667
88	0.000666667	0.166333	0.0403333
89	0.000666667	0.127667	0.039
90	0.000333333	0.115	0.039



41	0	0.025	0.00866667	91	0.000333333	0.136667	0.0426667
42	0	0.0253333	0.00866667	92	0.000666667	0.120667	0.0343333
43	0	0.0263333	0.0136667	93	0	0.131667	0.046
44	0.000333333	0.0363333	0.0126667	94	0.000333333	0.103333	0.0336667
45	0	0.0263333	0.0103333	95	0.000333333	0.102	0.0363333
46	0	0.0256667	0.0103333	96	0	0.0986667	0.034
47	0	0.0306667	0.013	97	0.000333333	0.112	0.0413333
48	0	0.0323333	0.0136667	98	0.000666667	0.104333	0.0363333
49	0	0.0333333	0.0136667	99	0.000333333	0.128333	0.0353333

Divide and Conquer Algorithm shows the worst time complexity. Even with small inputs it is slower comparing to GSM and Karatsuba. After the size 65 and more the runtime increases radically, and then with bigger sizes of numbers the run time changes intermittently.



The time complexity of Karatsuba's algorithm both theoretically and in practice have better time complexity.

Overall, the algorithm that performed the best was the GSM algorithm, while the worst was DAC. Theoretically, Karatsuba method should have better time complexity, as in GSM scales like n^2, while Karatsuba – n^1.6. So, in my code the runtime of those algorithm differs quite a lot comparing to how those algorithms scale in theory. I "blame" it on the makeEqual method, as everytime in Divide-and-Conquer and Karatsuba method we input a number with an odd number of digits, the makeEqual method inputs zeros in the number that is smaller and after running the recursion we will have a number that has some amount of zeros in the beginning of it and I had to create a loop that deletes them, which, of course, takes some time. In GSM we can observe the similar situation, I create a number with only 0, and then I have to delete 0’s that were “useless”. So, this points better be improved.