General information

You can open an interactive version of this document at this URL:

https://jhub-node01.inf.ethz.ch/aud2021/user-redirect/lab/tree/aud2021/AD21-01_karatsuba.ipynb

You can use your nethz account to log into the server (same account as for your email at ETH).

The documented is hosted in a directory `/aud2021/` that is shared with all students. You can run the code in the document and edit the document, but you cannot save it in this shared directory. If you wish to save your edits, you can do so in the root path `/` (e.g., using the keyboard `Ctrl+Shift-S` to invoke the command `Save notebook as ...`). This path corresponds to the home directory of your user on the server `jhub-node01.inf.ethz.ch`. Files in this home directory are not visible to other users of the server.

Disclaimer

The material in this document is completely optional and not considered to be part of the course "Algorithmen und Datenstrukturen". The aim of this document is to give you an idea how to translate high-level algorithmic ideas discussed in class to actual Java code. This Java code may use programming concepts that have not yet been covered in the course "Einführung in die Programmierung" at this time. The slides and recorded lectures for the "Vorkus" may serve as a useful reference.

At the above URL, you can interact with this Java code by modifying the inputs, running the code, and potentially modifying the code. You can always return to the original version of the document by closing the document, confirming that you wish to discard your changes, and opening the document again.

Hidden code cells

This document contains several hidden code cells. The code in these cells can safely be ignored.

Interacting with this document in jupyterlab

If you view the interactive version of this document at the above URL, you first need to run all code cells before you can interact with the document. In order to run all code cells invoke the command `Run -> Run All Cells` in the menu bar.

Problem Statement

input	output	operations
integers $a, b \in \mathbb{N}$	product $c = a \cdot b$	add / multiply digits

Grade-school Algorithm

The grade-school algorithm for integer multiplication proceeds in two steps:

1. compute partial products by multiplying every digit of a with every digit of b,
2. sum up these partial products moved to the appropriate decimal position.

The following table illustrates for a concrete example how these two steps work.

a_1	a_0		b_1	b_0	
6	5	$\cdot$	2	1	
				5	$a_0 \cdot b_0$
			6		$a_1 \cdot b_0$
		1	0		$a_0 \cdot b_1$
	1	2			$a_1 \cdot b_1$
	1	3	6	5	

The following mathematical identity justifies that for the case of two 2-digit numbers, this algorithm computes the correct output.

$$(10 \cdot a_1 + a_0) \cdot (10 \cdot b_1 + b_0) = a_1 b_1 \cdot 100 + (a_0 b_1 + a_1 b_0) \cdot 10 + a_0 b_0 \,.$$

(Note that the left-hand side is the output we desire and the right-hand side is the sum of partial products computed by the algorithm.)

Can we do better?

What does it even mean for an algorithm to do better for this problem?

For starters, we will count the number of single-digit multiplications performed by the algorithm.

For the above example, we perform 4 single-digit multiplications (one operation for every partial product).

How many single-digit multiplications would this algorithm perform to multiply two 4-digit numbers?

Answer: 16 (again one operation for every partial product)

How many single-digit multiplications would the algorithm carry out to multiply two n-digit numbers?

Answer: n^2 (there are n^2 partial products because there are n choices of digits for a and n choices of digits for b).

Is there a way to multiply two n-digit numbers with (significantly) fewer than n^2 single-digit multiplications?

Karatsuba's Algorithm

Let us first consider the case $n = 2$.

If we look at the mathematical identity that we used to justify the correctness of the grade-school algorithm, we see that we only need to know the following three numbers

$$a_0 \cdot b_0, \quad a_1 \cdot b_1, \quad (a_0 \cdot b_1 + a_1 \cdot b_0)$$

Naively, it takes two single-digit multiplications to compute the third number.

Could we compute the third number using just one single-digit multiplication?

In 1960, the Russian mathematician Anatoly Karatsuba discovered an algorithm to achieve that feat. His algorithm works by negating numbers and reusing the previously computed single-digit multiplications $a_0 \cdot b_0$ and $a_1 \cdot b_1$.

The following table illustrates how Karatsuba's algorithm multiplies two 2-digit numbers using just three single-digit multiplications.

a_1	a_0		b_1	b_0	
6	5	$\cdot$	2	1	
				5	$a_0 \cdot b_0$
	1	2			$a_1 \cdot b_1$
		1	7		$a_1 \cdot b_1 + a_0 \cdot b_0$
			-1		$-(a_1 - a_0) \cdot (b_1 - b_0)$
	1	3	6	5	

This computation carries out exactly three single-digit multiplications, namely

$$a_0 \cdot b_0, \quad a_1 \cdot b_1, \quad (a_1 - a_0) \cdot (b_1 - b_0)\,.$$

Similarly to the grade-school algorithm, we can illustrate the correctness of this computation by a mathematical identity. Let us call the above products u, v, and w (so that $u = a_0 \cdot b_0$, $v = a_1 \cdot b_1$, and $w = (a_1 - a_0) \cdot (b_1 - b_0)$). Then, the following mathematical identity illustrates the correctness of this computation,

$$(10 \cdot a_1 + a_0) \cdot (10 \cdot b_1 + b_0) = u + (u + v - w) \cdot 10 + v \cdot 100\,.$$

The following Java code is a complete implementation of Karatsuba's algorithm to multiply two 2-digit numbers.

```java
// Karatsuba's algorithm for n=2

// input values (feel free to change these values to any other valid input values)
int a1 = 6; int a0 = 5;
int b1 = 2; int b0 = 1;

// perform single digit multiplications
int u = a0 * b0;
int v = a1 * b1;
int w = (a1 - a0) * (b1 - b0);

// compute c0,c1,c2 such that a*b = c0 + 10*c1 + 100*c2
int c0 = u;
int c1 = u + v - w;
int c2 = v;

// extract decimal digits of c0 + 10*c1 + 100*c2
int carry0 = 0;
int d0 = (c0 + carry0)%10; int carry1 = (c0 + carry0)/10;
int d1 = (c1 + carry1)%10; int carry2 = (c1 + carry1)/10;
int d2 = (c2 + carry2)%10; int carry3 = (c2 + carry2)/10;
int d3 = carry3;

// display table with all input values, intermediate values, and output values
// (this code is just for illustration and not part of the algorithm)
displayKaratsuba(a0,a1,b0,b1,u,v,w,c0,c1,c2,carry0,carry1,carry2,carry3,d0,d1,d2,d3);
```

	3	2	1	0
a			6	5
b			2	1
u				5
v		12		
u			5	
v			12	
$-w$			-1	
c		12	16	5
`carry`	1	1	0	0
d	1	3	6	5

Activity
Change the input values for `a1`,`a0`,`b1`,`b0` to any other single-digit numbers. Re-run the above cell by pressing `Shift-Enter`. Observe how the intermediate values and the final result changes.

Question
How large a value can you achieve for `carry3` by adjusting the input values? How about `carry2` and `carry1`?

Illustration of Karatsuba's algorithm for more digits

The following Java code serves to illustrate how Karatsuba's algorithm works for inputs with more than 2 digits. We emphasize that this Java code is not a proper implementation of Karatsuba's algorithm. In fact, this Java code turns out to be limited to inputs with at most 8 digits.

Additional explanation
Why is this implementation limited to inputs with a small number of digits and not a proper implementation of Karatsuba's algorithm?
The main reason is that this implementation uses `long`, one of Java's primitive data types, to represent the input and the output values. A `long` data type can represent only values between -2^{63} and $2^{63}-1$. For example, if we choose input values $a = 2^{32}$ and 2^{32} (10 decimal digits), then the output value 2^{64} could not be represented as a `long`.
In order to be able to deal with larger numbers, we could represent numbers as a list of digits, where the length of the list is not restricted a priori (except by the amount of available memory). Java provides such a representation in form of the `BigInteger` class in the `java.math` package. We choose not to use this class here because we want to restrict ourselves to more basic Java features at this point.

```java
// partial implementation of Karatsuba's algorithm for illustration purposes
long Karatsuba(long a,long b,long k, int depth) {
    // assertion: the decimal representations of a and b each have at most 2^k digits

    // print the values of a and b;
    // indentation indicates depth of recursion
    System.out.printf("%s%d*%d\n","  ".repeat(depth),a,b);

    // base case: a and b are single-digit numbers
    if (k==0) {
        return a*b;
    }

    // compute p = 10^{2^{k-1}}
    long p = power(10,power(2,k-1));

    // compute highest 2^{k-1} digits of a and b
    long a1 = a / p; long b1 = b / p;
    // compute lowest 2^{k-1} digits of a and b
    long a0 = a % p; long b0 = b % p;

    // recursively compute the products a0*b0, a1*b1, (a1-a0)*(b1-b0)
    long u = Karatsuba(a0,b0,k-1,depth+1);
    long v = Karatsuba(a1,b1,k-1,depth+1);
    long w = Karatsuba(a1 - a0,b1 - b0,k-1,depth+1);

    // combine the results to obtain the product a*b
    return u + p*(u+v-w) + p*p*v;
}
```

```
// illustration of Karatsuba's algorithm

// inputs (limite to 8 digit numbers)
long a = 67223345L;
long b = 84343592L;

// number of digits of inputs
long n = (long)Math.ceil(Math.max(Math.log10(a),Math.log10(b)));
// smallest integer k such that both a and b have at most 2^k digits
long k = (long)(Math.log(n)/Math.log(2));

// run Karatsuba's algorithm to multiply a and b
long result = Karatsuba(a,b,k,0);

System.out.printf("Output of Karatsuba's algorithm: %d\n", result);

System.out.println("Check standard multiplication has same output:");

result == a*b;
```

```
67223345*84343592
  3345*3592
    45*92
      5*2
      4*9
      -1*7
    33*35
      3*5
      3*3
      0*-2
    -12*-57
      -2*-7
      -1*-5
      1*2
  6722*8434
    22*34
      2*4
      2*3
      0*-1
    67*84
      7*4
      6*8
      -1*4
    45*50
      5*0
      4*5
      -1*5
  3377*4842
    77*42
      7*2
      7*4
      0*2
    33*48
      3*8
      3*4
      0*-4
    -44*6
      -4*6
      -4*0
      0*-6
Output of Karatsuba's algorithm: 5669858383555240
Check standard multiplication has same output:
```

```
true
```

Activity

Change the input values for `a`,`b` to any other 8-digit numbers. Re-run the above cell by pressing `Shift-Enter`. Observe how the intermediate values and the final result changes. Observe how the printed output represents the nested recursive calls of the `Karatsuba` method. (You can see that a single multiplication of 8-digit numbers is broken up into 3 multiplications of 4-digit numbers. Each multiplication of 4-digit numbers is broken up into 3 multiplications of 2-digit numbers. Finally, each multiplication of 2-digit numbers is broken up into 3 multiplications of 1-digit numbers. In total, 27 multiplications of 1-digit numbers are carried out.)