Divide and Conquer

Andres Mendez-Vazquez

January 10, 2018

Contents

1 Improving Algorithms **3**
1.1 Using Multiplication of Imaginary Numbers 3

2 Using Recursion to Find Complexities **5**

3 Asymptotic Notation **7**
3.1 Relations between Θ and O and Ω 8
3.2 Some examples about little o and ω 9
3.3 Exercises 9

4 Solving Recurrences **10**
4.1 Substitution Method 10
4.1.1 Subtleties of the Substitution Method 10
4.2 The Reduced Version of the Master Theorem 11
4.3 Exercises 13

List of Algorithms

1 Recursive Multiplication 5
2 Merge Sort Algorithm 7
3 Binary Search algorithm 10
4 Shell Sort algorithm 10

List of Figures

1 The Recursion Tree Of Merge Sort when n is even 7
2 The branching and depth of a recursion $T(n) = aT\left(\frac{n}{b}\right) + n$ . . . 12

1 Improving Algorithms

Although, we saw already a simple way of reducing the complexity of an algorithm by avoiding as much as possible the recursion by memorizing previous smaller values (Iterative Fibonacci). It is clear that neater tricks must exist in the realm of possible solutions while solving problems for algorithms. Thus, looking a those tricks is quite important to be able to learn how to improve on many different algorithms.

A clever strategy for improving algorithms is the well known Divide and Conquer which consists of the following steps:

1. **Divide**. Here the algorithm must split the large problem into smaller versions, thus they can be solved recursively by using new algorithm instantiation and the smaller inputs.

2. **Conquer**. At this moment, we use the idea of induction by going first backwards to the basis case where the easiest solution can be find, then the procedure is ready to go forward.

3. **Combine**. Here, we use the previously calculated answers to build the an answer for each subproblem. For example in the case of Fibonacci:

$$F_n = F_{n-1} + F_{n-2} \tag{1}$$

Then, it is essential to build clever algorithms that can build efficient answers for given problems. Thus, the importance of looking at several possible tricks to be able to obtain a basic intuitions for them!!! In order to avoid exponential number of steps while calculating the answer.

The next example of a clever trick that we will be studying is coming from the Prince of Mathematics, Frederich Carl Gauss (1777-1855) [3]. This improvement was developed by Anatoly Alexeevitch Karatsuba (1937 - 2008) at the Faculty of Mechanics and Mathematics of Moscow State University [5].

1.1 Using Multiplication of Imaginary Numbers

Even when we could have a efficient strategy to do the divide and conquer, the most difficult part is always the combine one because the need of not only doing simple combinations of subproblems, but clever ones. For example, given that the numbers could be represented as $x = x_L \circ x_R = 2^{n/2}x_L + x_R$ by assuming that n (Size of each of string of bits) is even. As an example of this, $x = 10110110$ can be seen as the left bits concatenated to right bits: $x_L = 1011$ and $x_R = 0110$. Then, $x = 1011 \times 2^4 + 0110$.

Thus, the multiplication between two numbers, x and y, can be found using the following equation:

$$\begin{aligned} xy &= \left(2^{n/2}x_L + x_R\right)\left(2^{n/2}y_L + y_R\right) \\ &= 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

Making possible to use the recursion to obtain the final multiplication between both numbers (Algorithm 1). This algorithm has the following total time:

$$T(n) = 4T\left(\frac{n}{2}\right) + Somework. \tag{2}$$

Where *Somework* correspond to the addition of the results obtained by calling the function in the left and right sections of both numbers. Although at the beginning, the recursive function looks quite complex, it is actually one the easiest to solve by means of the recursive three to guess the complexity. Thus, once you are understand the three main methods of basic complexity calculation:

- The Recursion-Tree Method.
- The Substitution Method.
- The Master Theorem Method.

It is possible to see that this recursive function is bounded by cn^2 time for some constant c.

An immediate question for anybody trying to improve this naive algorithm is the following one: Can be possible to have an algorithm able to perform the multiplication in shorter time? The answer is actually more convoluted than one can imagine. First, for many problems, it is possible to obtain absolute lower bounds that have not been reached by any actual algorithms used to solve them. An example of this situation is the algorithms used for matrix multiplication [6]. Therefore, from the philosophical point of view of algorithms, we first develop an initial solution and later on, we try to develop a faster solution.

Thus, What will allow us to improve the naive multiplication? While looking through the history of mathematics (It is always a good starting point for any developer of algorithms) Karatsuba [5] noticed that it is possible to have the following representation of imaginary numbers in a computer:

100100100101	010101110110
Bits for imaginary	Bits for real

He was able to see that imaginary numbers could be seen as binary numbers with $\frac{n}{2} + \frac{n}{2}$ bits. Thus, binary number can be seen as imaginary number where

$$x = 2^{\frac{n}{2}} \times \underbrace{x_L}_{\text{Imaginary}} + \underbrace{x_R}_{\text{Real}} \tag{3}$$

Algorithm 1 Recursive Multiplication

function *Naive-Recursive-Multiplication*

Input: x and y integers in binary representation, the number of bits n for both integers

Output: The multiplication $x \times y$

1. if $n == 1$
2. return $x\&y$
3. $t_1 =$*Naive-Recursive-Multiplication*$\left(x_L, y_L, \frac{n}{2}\right)$
4. $t_2 =$*Naive-Recursive-Multiplication*$\left(x_L, y_R, \frac{n}{2}\right)$
5. $t_3 =$*Naive-Recursive-Multiplication*$\left(x_R, y_L, \frac{n}{2}\right)$
6. $t_4 =$*Naive-Recursive-Multiplication*$\left(x_R, y_R, \frac{n}{2}\right)$
7. return $2^n \times t_1 + 2^{\frac{n}{2}} \left[t_2 + t_3\right] + t_4$

Thus, it could be possible to use the Gauss trick to decrease the number of multiplications from four to three by realizing the following. Given the equation of multiplication of imaginary numbers:

$$(a + ib)(c + id) = ac + (ad + bc)i - bd \tag{4}$$

Thus, Gauss noticed that $ad + bc =$. Therefore,

$$(a + ib)(c + id) = ac + [(a + b)(c + d) - ac - bd]i - bd \tag{5}$$

Then, if we use the Gauss's trick, we only need $x_L y_L$, $x_R y_R$, $(x_L + x_R)(y_L + y_R)$ to calculate the multiplication because $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$. This allows to generate a new algorithm that only requires to calculate three recursive multiplications. Thus, the time complexity of the new matrix multiplication is:

$$T(n) = 3T(n/2) + Somework \tag{6}$$

This equation can be proved to have an upper bound of $cn^{1.59}$. Therefore, it is not only to use of divide, but also the clever use of that conquer and combine.

2 Using Recursion to Find Complexities

After this clear example on how to improve a basic algorithm, we turn our heads to the merge sort algorithm (Algorithm 2). The merge sort algorithm exemplify

the classic way of using the recursion tree to obtain an initial guess for the time complexity.

Definition 1. Given an input as a string where the problem is being encoded using an alphabet Σ, the **time complexity** quantifies the amount of time taken by an algorithm to run as a function on the length of such string.

For this, look at the recursion tree in Merge sort (Fig. 1). There, we can see how the data is split in two parts in the following way

1. At the top we have n elements to be sorted
2. At the second level we split the data, using the recursion, into two subproblems of size $\frac{n}{2}$ and $\frac{n}{2}$.
3. At the third level the data is split into subproblems of size $\frac{n}{4}$.
4. And so on.

Thus, we can define the function to calculate the time complexity of steps using a recursion defined in the following way (Eq.).

$$T(n) = T\left(\tfrac{n}{2}\right) + T\left(\tfrac{n}{2}\right) + Somework.$$

Here, *Somework* can be seen as the merging that the MERGE needs to do i.e. cn time complexity. Formally, we can define the recursion for the time complexity as:

$$T\left(n\right) = \begin{cases} 1 & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases} \tag{7}$$

Algorithm 2 Merge Sort Algorithm

function *Merge-Sort*

Input: an array of integers A, the left end index p, the right end index r

Output: the array A with the integers in increasing order

1. if $p < r$ then
2. $\quad q \leftarrow \lfloor \frac{p+r}{2} \rfloor$
3. $\quad$ Merge-Sort(A,p,q)
4. $\quad$ Merge-Sort(A,q+1,r)
5. $\quad$ MERGE(A,p,q,r)

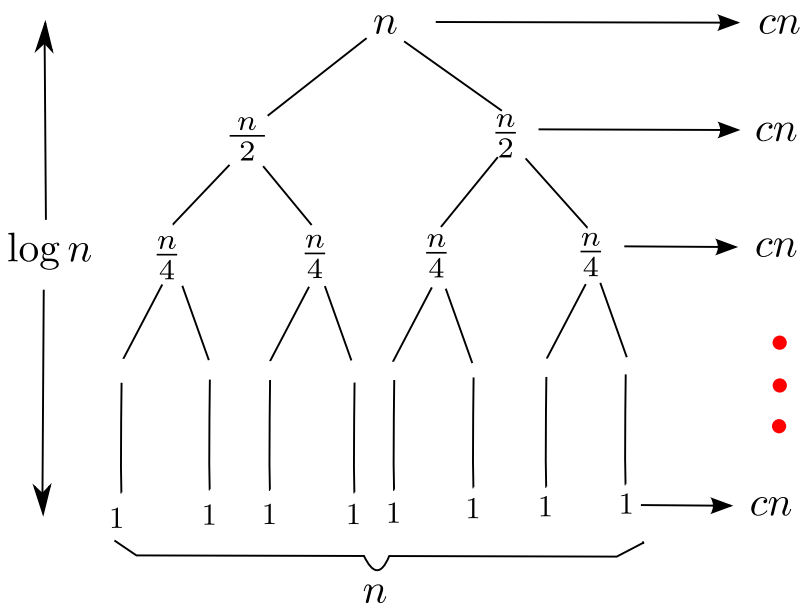

Figure 1: The Recursion Tree Of Merge Sort when n is even

3 Asymptotic Notation

We are ready to define one of the main tools for the calculation of complexities, the asymptotic notation [1, 4]. Asymptotic notation is a way to express the main component of the cost of an algorithm by using idealized units of work.

The main families of asymptotic notation are the Big O, the Big Ω, the Θ, the little o and the little ω.

First, let us to examine the famous big O.

Definition 2. For a given function $g(n)$

$$O(g(n)) = \{f(n) | \text{ There exists } c > 0 \text{ and } n_0 > 0 \\ \text{s.t. } 0 \leq f(n) \leq cg(n) \ \forall n \geq n_0\}$$

We have the following example for this notation.

Example. For example, we know that $n \leq n^2$ for $n \geq 1$, thus if we select $c = 1$. Then, we have that $n \in O\left(n^2\right)$ or $n = O\left(n^2\right)$.

Next, we have the lower bound for the complexity functions, the Ω set.

Definition 3. For a given function $g(n)$

$$\Omega(g(n)) = \{f(n) | \text{ There exists } c > 0 \text{ and } n_0 > 0 \\ \text{s.t. } 0 \leq cg(n) \leq f(n) \ \forall n \geq n_0\}$$

Finally, we get a combination of the previous sets as the Θ set.

Definition 4. For a given function $g(n)$

$$\Theta(g(n)) = \{f(n) | \text{ There exists } c_1 > 0, c_2 > 0 \text{ and } n_0 > 0 \\ \text{s.t. } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \ \forall n \geq n_0\}$$

3.1 Relations between Θ and O and Ω

The first relation is the transitivity.

Proposition 5. *(Transitivity) Given $f(n)$ and $g(n)$, if $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$*

It is easy to prove the proposition, you simply look at the constants provided by the equalities. The remaining only requires a correct interpretation.

1. $f(n) = \Theta(f(n))$ (Reflexivity).
2. $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$ (Symmetry).
3. $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$(Transpose Symmetry).

3.2 Some examples about little o and ω

The following examples clarify more the ideas about this "little" notation.

Example. For the little o, we have that $2n = o(n^2)$, but $2n^2 \neq o(n^2)$. In the case of the first part, it is easy to see that for any given c exist a n_0 such that $\frac{1}{\frac{n_0}{2}} < c$. In addition, $n > n_0$ implies that $\frac{1}{n_0} > \frac{1}{n}$. Then,

$$2 < cn \iff 2n < cn^2 \tag{8}$$

This gives us the proof for the first part. In the second part, if we assume $c = 2$ and a certain value n_0 that makes true the inequality, we have that

$$2n_0^2 < 2n_0^2$$

which is clearly a contradiction.

Example. A similar situation can be seen in little ω. For example $\frac{n^2}{2} = \omega(n)$, but $\frac{n^2}{2} \neq \omega(n^2)$. In the first case, a similar argument can be done such that

$$cn < \frac{n^2}{2} \tag{9}$$

However, if we assume that the inequality holds for the second case we can chose $c = 2$, we again obtain a contradiction.

3.3 Exercises

1. Please try to solve the following examples, i.e. find the c_1, c_2 and/or n_0
   (a) $n^2 - 5n = \Theta(n^2)$.
   (b) $\sqrt{n} = O(n)$
   (c) $n^2 = \Omega(n)$
2. Given the following codes can you devise their recursions by converting the iterative versions into recursive and provide the recursions
   (a) For Binary Search (Algorithm 3).
   (b) For Shell Sorting (Algorithm 4)

Algorithm 3 Binary Search algorithm

function *Binary Search*

Input: An array $A\,[1...n]$ of n elements sorted in nondecreasing order and an element x.

Output: j if $x = A\,[j]$, $1 \leq j \leq n$ and 0 otherwise

1. $low \leftarrow 1;\ high \leftarrow n;\ j \leftarrow 0$
2. while $(low \leq high)$ and $(j == 0)$
3. $\quad mid \leftarrow \left\lfloor \frac{(low+high)}{2} \right\rfloor$
4. $\quad$ if $x == A\,[mid]$ then $j \leftarrow mid$
5. $\quad$ if $x < A\,[mid]$ then $high \leftarrow mid - 1$
6. $\quad$ else $low \leftarrow mid + 1$
7. return j

Algorithm 4 Shell Sort algorithm

function *Shell Sort*

Input: An array $A\,[1...n]$ of n elements and a decreasing integer gap sequence $G\,[1..m]$.

Output: The array $A\,[1...n]$ sorted in increasing order.

1. for $g \leftarrow G\,[1]$ to $G\,[m]$
2. $\quad$ for $i \leftarrow g$ to n
3. $\quad\quad temp \leftarrow A\,[i]$
4. $\quad\quad j \leftarrow i$
5. $\quad\quad$ while $g \leq j$ and $A\,[j - g]$
6. $\quad\quad\quad A\,[j] \leftarrow A\,[j - g]$
7. $\quad\quad\quad j \leftarrow j - g$
8. $\quad\quad A\,[j] \leftarrow temp$

4 Solving Recurrences

4.1 Substitution Method

The main method is explained in the slides, but if you want more information please take a look at Cormen et al. [2].

4.1.1 Subtleties of the Substitution Method

Guess that $T(n) = O(n)$, then you can have something like

$$\begin{aligned} T(n) \leq & c\left\lfloor \frac{n}{2} \right\rfloor + c\left\lceil \frac{n}{2} \right\rceil + 1 \\ = & cn + 1 \\ = & O(n) \end{aligned}$$

Which is not the correct induction, after all $cn+1$ is not cn. We can overcome this problem by assuming a $d \geq 0$ and then "guessing" $T(n) \leq cn - d$. Then

$$\begin{aligned} T(n) \leq & \left(c\left\lfloor \frac{n}{2} \right\rfloor - d\right) + \left(c\left\lceil \frac{n}{2} \right\rceil - d\right) + 1 \\ = & cn - 2d + 1 \end{aligned}$$

Then, if we select $d \geq 1 \Rightarrow 0 \geq 1 - d$. This means that $cn - 2d + 1 \leq cn - d$. Therefore, $T(n) \leq cn - d = O(n)$.

4.2 The Reduced Version of the Master Theorem

Here, we discuss a somewhat reduced version of the Master Theorem from Dasgupta et al. [3], and we show the equivalence with the one at Cormen et. al [2].

Theorem 1. ***Master Theorem.*** *If* $T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O\left(n^d\right)$ *for some constants* $a > 0$, $b > 1$, *and* $d \geq 0$ *then*

$$T(n) = \begin{cases} O\left(n^d\right) & \text{if } d > \log_b a \\ O\left(n^d \log n\right) & \text{if } d = \log_b a \\ O\left(n^{\log_b a}\right) & \text{if } d < \log_b a \end{cases}$$

Proof. First, for convenience assume $n = b^p$. Now we can notice that the size of the subproblems are decreasing by a factor of b at each recursive step. This means that the size of each subproblems is $\frac{n}{b^i}$ at level i. Thus, in order to reach the bottom you need to have subptoblems of size 1. Therefore:

$$\frac{n}{b^i} = 1 \Rightarrow i = \log_b n \tag{10}$$

where i = height of the recursion three. Now, given that the branching factor is a, we have at the k^{th}level a^ksubproblems, each of size $\frac{n}{b^k}$. Then, the work at level k is:

$$a^k \times O\left(\frac{n}{b^k}\right)^d = O(n^d) \times \left(\frac{a}{b^d}\right)^k \tag{11}$$

Then, the total work done by the recursion is

$$T(n) = O(n^d) \times \left(\frac{a}{b^d}\right)^0 + O(n^d) \times \left(\frac{a}{b^d}\right)^2 + ... + O(n^d) \times \left(\frac{a}{b^d}\right)^{log_b n} \tag{12}$$

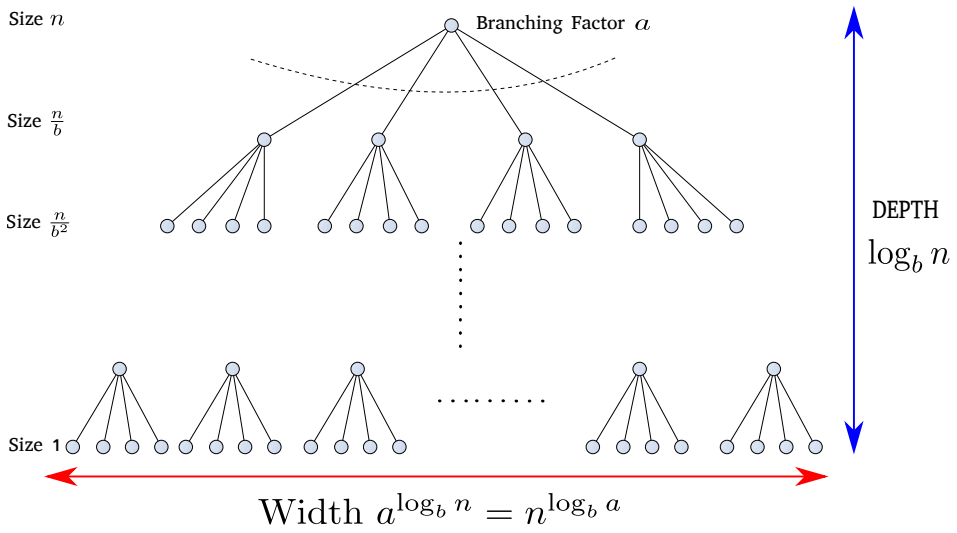

Figure 2: The branching and depth of a recursion $T(n) = aT\left(\frac{n}{b}\right) + n$

For the next step we are going to use the following facts of the Θ notation. For a $g(m) = 1 + c + c^2 + ... + c^m$, we have the following cases:

1. if $c < 1$ then $g(m) = \Theta(1)$
2. if $c = 1$ then $g(m) = \Theta(m)$
3. if $c > 1$ then $g(m) = \Theta(c^m)$

Now, we can see the different cases:

1. If $\frac{a}{b^d} < 1$, then we have that $a < b^d$ or $\log_b a < d$ (Case one of the theorem). Then, $T(n) = O(n^d)$.
2. If $\frac{a}{b^d} = 1$, then we have that $a = b^d$ or $\log_b a = d$ (Case two of the theorem). Then, we have that $g(n) = \left(\frac{a}{b^d}\right)^0 + \left(\frac{a}{b^d}\right)^2 + ... + \left(\frac{a}{b^d}\right)^{log_b n}$ is $\Theta(\log_b n)$. Then, $T(n) = O(n^{\log_b a} \log_b n) = O\left(n^{\log_n a} \log_2 n\right)$ because b can only be greater or equal to two.
3. If $\frac{a}{b^d} > 1$, then we have that $a > b^d$ or $\log_b a > d$ (Case three of the theorem). Then. we have $n^d \times \left(\frac{a}{b^d}\right)^{log_b n} = n^d \times \left(\frac{a^{\log_b n}}{\left(b^{\log_b n}\right)^d}\right) = a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a}$. Thus $T(n) = O(n^{\log_b a})$

□

Now, if we look at the version at Cormen et al. [2].

Theorem. *Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where we interpret $\frac{n}{b}$ as $\lfloor \frac{n}{b} \rfloor$ or $\lceil \frac{n}{b} \rceil$. Then $T(n)$ can be bounded asymptotically as follows:

1. *If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$. Then $T(n) = \Theta\left(n^{\log_b a}\right)$.*
2. *If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$.*
3. *If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$ and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

Thus, for the following cases:

1. If $d > \log_b a$, then it is possible to say $d = \log_b a + \epsilon$, thus $f(n) = O\left(n^{\log_b a + \epsilon}\right)$. Then, $T(n) = O\left(n^d\right) = O(f(n))$.
2. If $d = \log_b a$, thus $f(n) = O\left(n^{\log_b a}\right)$. Then, $T(n) = O\left(n^d \log n\right) = O\left(n^{\log_b a} \lg n\right)$.
3. If $d < \log_b a$, then $d + \epsilon = \log_b a$, thus $f(n) = O\left(n^{\log_b a - \epsilon}\right)$. Then, $T(n) = O\left(n^{\log_b a}\right)$.

This almost prove the equivalence between the two Master Theorems, we leave the other parts to you.

4.3 Exercises

1. Use the substitution method to find an upper bound for the recurrence

$$T(n) = \begin{cases} T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\frac{2n}{4}\right) & \text{if } n \geq 4 \\ 4 & \text{if } n < 4 \end{cases}$$

2. Use the Tree method to solve the recurrence

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + T\left(\frac{n}{5}\right) + \sqrt{n} & \text{if } n \geq 4 \\ 4 & \text{if } n < 4 \end{cases}$$

3. Prove that the solution to the recurrence

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + g(n) & \text{if } n \geq 2 \\ 1 & \text{if } n = 1 \end{cases}$$

is $T(n) = O(n)$ whenever $g(n) = o(n)$.

References

[1] Paul Bachmann. *Die Analytische Zahlentheorie.* 1894.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[3] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani. *Algorithms.* May 2006.

[4] N.G. de Bruijn. *Asymptotic Methods in Analysis.* Bibliotheca mathematica. Dover Publications, 1970.

[5] A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. *Proceedings of USSR Academy of Sciences*, 145(7):293–294, 1962.

[6] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 887–898, New York, NY, USA, 2012. ACM.