Faculty of Science and Engineering

Bachelor Thesis

Convolution Algorithms

for Integer data types

Author:
Nicolae Ghidirimschi

Supervisors:
dr. Arnold Meijster
dr. Sietse van Netten

12th July 2021

Abstract

In this bachelor thesis we discuss and implement several convolution algorithms on integer data types and define the scenarios in which each of these algorithms can perform the best in terms of running time. We investigate the individual limitations and bottlenecks of these algorithms, to which we propose potential solutions and optimizations. We also present several novel data-driven convolution algorithms. Lastly, we conduct running time measurements and evaluate the potential of these algorithms and the suggested extensions to be used in practice.

Contents

Chapter 1

Introduction

1.1 Convolution definition

Convolution is a mathematical operation that has various and important applications in many domains, such as image and audio processing, signal filtering, statistics and Artificial Intelligence. Given such a wide domain of application, it is not surprising that there exist both multiple informal and formal definitions for convolution, and as will be exemplified in the following sections, the homogeneity of these definitions is not always straightforward.

Informally, convolution is often defined as the operation that blends two inputs, measures the amount of overlap between these, or quantifies how the shape of one input is affected by the other. Figure 1 illustrates the result of such a "blending", where one of the inputs is a colored image and the other is a blur kernel.

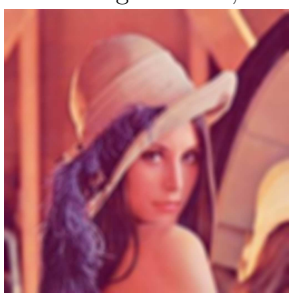

Figure 1.1: Blurred Lenna[1] using convolution

Similarly, multiple formal definitions of convolution may be distinguished based on the nature and properties of the inputs. In this thesis, we will limit ourselves to algorithms for the discrete convolution of integer typed data. In particular, for two discrete 1-dimensional time series x and h, their convolution $y = x * h$ is defined as:

$$(x * h)[n] = y[n] = \sum_{k=0}^{|h|-1} h[k] \cdot x[n-k] \qquad \text{(Def. 1.1)}$$

Here the notation $|h|$ denotes the number of elements in the time series h. We will often refer to $|h|$ as the *'length of h'*, but the reader is warned that this is not the Euclidean length of h.

Throughout this thesis, we will also assume that indexing of time series start from zero. Moreover, a time series is assumed to be zero valued for indices outside its domain (i.e. negative indices and indices greater or equal to the 'length' of the series).

As a particular example, consider $x = [1, 2, 2]$ and $h = [2, 3, 1]$. According to the definition above:

$y[0] = \sum_{k=0}^{2} h[k] \cdot x[0-k] = h[0] \cdot x[0] + h[1] \cdot x[-1] + h[2] \cdot x[-2] = 2 \cdot 1 + 3 \cdot 0 + 1 \cdot 0 = 2$
$y[1] = \sum_{k=0}^{2} h[k] \cdot x[1-k] = h[0] \cdot x[1-0] + h[1] \cdot x[1-1] + h[2] \cdot x[1-2] = 2 \cdot 2 + 3 \cdot 1 + 1 \cdot 0 = 7$
$y[2] = \sum_{k=0}^{2} h[k] \cdot x[2-k] = h[0] \cdot x[2-0] + h[1] \cdot x[2-1] + h[2] \cdot x[2-2] = 2 \cdot 2 + 3 \cdot 2 + 1 \cdot 1 = 11$

By computing $y[3]$ and $y[4]$ in a similar fashion, we obtain $x * h = y = [2, 7, 11, 8, 2]$.

Now consider the two integers that are obtained by 'concatenating' the time series above, that is, $a = 122$ and $b = 231$. Their integer multiplication is equal to $122 \cdot 231 = 28182$, which correspond to the values of y, given a carry shift is performed on its third value (being 11). As will be shown

[1] Lenna is a picture of Swedish model Lena Forsén that is often used in the field of image processing.

in this thesis, this example can be generalized to time series of any type and length. In particular, by concatenating the digits of the resulting convolution of two time series whose samples are all positive integers smaller than 10, after performing a carry shift, the same number is obtained as by integer multiplication of the two integers that are constructed from the values of the two given time series. It therefore follows that in the context of integer time series, convolution can be defined as long multiplication without carry shifting.

Similarly, consider the two polynomials $x^2 + 2x + 2$ and $2x^2 + 3x + 1$, which are formed by making $a_k = x[k]$ and $h[k]$ respectively in the general polynomial form given by $\sum_{k=0}^{n} a_k x^k$. Their product is the polynomial $2x^4 + 7x^3 + 11x^2 + 8x + 2$, whose coefficients a_k are equal to $y[k]$. This too, can be generalized to any time series and its corresponding polynomial.

These three distinct definitions will serve as the foundations of various algorithms that we will discuss in this project. It turns out that each such definition motivates significantly different implementations, each with different advantages and disadvantages. The various properties of each definition, together or combined, can also be exploited to achieve further optimizations and improvements. This will be discussed in much more detail in the following chapters.

1.2 Research goal

Our research will be centered on the following question:

What algorithms are most time efficient for computing the convolution (according to Def. 1.1) of two time-series on a modern machine and in which scenarios do they perform best?

To answer this, we will first describe the current state-of-affairs of convolution algorithms from a theoretical standpoint. A particular emphasis will be put on clarifying how and why these algorithms work, as well as their asymptotic time complexity. This will help us identify their limitations and bottlenecks, improving which will be our next milestone. We will also investigate whether specific configurations of the inputs might justify novel algorithms. Finally, we will implement all the discussed ideas and conduct running time measurements with various configurations of the inputs. All the collected information will then be used to answer the question above.

Chapter 2

Convolution Algorithms

The earliest study of the discrete convolution operation dates as early as 1821, and was performed by Cauchy in his book *"Cours d'Analyse de l'École Royale Polytechnique"* [4]. Although statisticians first used convolution for practical purposes as early as 19th century [6], the term "convolution" did not enter wide use until 1950-60. These years correspond to the period when important applications of discrete convolution started to emerge (some of which are mentioned in the previous section), which in turn motivated significant research into this area, particularly resulting in many important algorithms that allow for an efficient computation of the convolution of two 1-dimensional integer-valued time series. Albeit these algorithms already facilitate such computations by huge factor, they also lay the groundwork for further optimizations and algorithms.

This serves as our main incentive to dedicate this section to the investigation and elucidation of some of the most eminent discrete convolution algorithms that are known at the moment of writing of this project. These will include both optimal ($\mathcal{O}(n \log n)$) and sub-optimal algorithms (with a higher time complexity), as the latter not only employ relevant optimization techniques, but might also outperform the former for specific inputs.

2.1 Output-side algorithm

We begin by introducing the most naive and straightforward algorithm that implements convolution. The algorithm is called the "Output-side algorithm" because it computes the individual samples of the output time series one-by-one [12]. This only requires the knowledge of how such a sample can be computed in terms of the input time series, which is directly given by Def. 1.1.

Table 2.1 illustrates an example of convolution in a way which makes the visualization of Def. 1.1 more intuitive. In particular, to compute sample $y[n]$, one can retrieve the values of the cells on column n in the table by calculating the values of the product specified by each row number, and then adding them together. Note that in our example we have left blanks for cells that are to be computed using a value outside the time series (*e.g.* in column $n = 0$, that are the values of both rows $h[1]x[n-1]$ and $h[2]x[n-2]$, as both $x[n-1]$ and $x[n-2]$ are outside the domain of x). These values can be avoided altogether by ensuring that k starts at $n - |x| + 1$ if this value is greater than 0, which avoids samples of x with an index higher than $|x|$, and that it stops at n when $n < |h| - 1$, which avoids samples of x with a negative index. In other words, under the assumption that samples outside the domain of a time series are equal to 0, Def. 1.1 is equivalent to $\sum_{k=\max(0,n-|x|+1)}^{\min(n,|h|-1)} h[k] \cdot x[n-k]$. This is reflected in the pseudo-code below.

n	0	1	2	3	4
x[n]	4	2	-1		
h[n]	-2	4	2		
h[0]x[n-0]	-8	-4	2		
h[1]x[n-1]		16	8	-4	
h[2]x[n-2]			8	4	-2
y[n]	-8	12	18	0	-2

Table 2.1: $[4, 2, -1] * [-2, 4, 2]$

Algorithm 1 Output-side algorithm

Input: x and h are two integer-valued time series
Output: y is the convolution of x and h

```
1: function CONVOLVE(x, h)
2:     y ← ZEROES(|x| + |h| − 1)  ▷ creates a time series of the specified length filled with zeroes
3:     for n ← 0 to |y| − 1 do                ▷ Loop intervals are closed (upper bound is included)
4:         lwb ← MAX(0, n − |x| + 1)
5:         upb ← MIN(n, |h| − 1)
6:         for k ← lwb to upb do
7:             y[n] ← y[n] + h[k] · x[n − k]
8:         end for
9:     end for
10:    return y
11: end function
```

The complexity of this algorithm is $\mathcal{O}(|y| \cdot |h|) = \mathcal{O}(|x| \cdot |h| + |h|^2)$. Let $n > |x|, |h|$ then this can be further expressed as $\mathcal{O}(n^2)$.

2.2 Input-side algorithm

The Output-side algorithm has the downside of requiring to compute lwb and upb for each output sample (see Alg. 1) in order to not index the input arrays out of bounds. Moreover, this conditional test in each iteration of the outer loop reduces the possibilities for a compiler to vectorize it fully for CPUs that have vector instructions.

The Input-side algorithm avoids this problem. Instead of computing an individual sample of the output time series one at a time, it calculates the contribution of each combination of samples from the input time series and incrementally updates the samples in the output to reflect them. As a concrete example, during the first iteration of the outer loop, instead of computing the value of $y[0]$, as does **Algorithm 1**, the Input-side algorithm will add $h[0] \cdot x[0] = -8$, $h[0] \cdot x[1] = -4$ and $h[0] \cdot x[2] = 2$ to $y[0]$, $y[1]$ and $y[2]$ respectively. The algorithm will then proceed to compute further contributions in a similar fashion, adding them to the corresponding samples of the output (which will be shifted by 1 with respect to the previous iteration). In this regard, the algorithm is thus more kin to the long multiplication definition described in Section 1 rather than to Def. 1.1.

Algorithm 2 Input-side algorithm

Input: x and h are two integer-valued time series
Output: y is the convolution of x and h

```
1: function CONVOLVE(x, h)
2:     y ← ZEROES(|x| + |h| − 1)  ▷ creates a time series of the specified length filled with zeroes
3:     for k ← 0 to |h| − 1 do
4:         for i ← 0 to |x| − 1 do
5:             y[k + i] ← y[k + i] + h[k] · x[i]
6:         end for
7:     end for
8:     return y
9: end function
```

The time complexity of the Input-side algorithm is $\mathcal{O}(|x| \cdot |h|)$, which makes the algorithm slightly more efficient than the Output-side one. This is because $|x| < |x| + |h| - 1 = |y|$, and thus $|x| \cdot |h| < |y| \cdot |h|$. More informally, it is more efficient because it avoids any sample that lies outside the domain of either input time series. Nevertheless, under the previous assumption: $n > |h|, |x|$, the algorithm also belongs to $\mathcal{O}(n^2)$ and is therefore sub-optimal.

It is worth noting that albeit the Input-side algorithm is arguably less straightforward than the

Output-side one and both algorithms are sub-optimal, it is generally more preferred, as it has higher potential to benefit from compiler optimizations (mainly vectorization) due to lack of conditional statements.

2.3 Karatsuba (adapted) algorithm

The Karatsuba algorithm (cf. [7]) achieves fast multiplication of two numbers by reducing the number of elementary operations required to perform a traditional long multiplication. The long multiplication algorithm is similar to the Input-side trivial algorithm discussed in the previous section, with time series being represented by integer numbers and samples by digits. It also involves an extra step of carry shifting, which, takes linear time in terms of the length of the output. Thus, assuming that both input factors of the multiplication have at most n digits, the traditional long multiplication algorithm belongs to $\mathcal{O}(n^2 + 2n) \cong \mathcal{O}(n^2)$ as well.

In 1960, Andrey Kolmogorov conjectured that the quadratic complexity was asymptotically optimal for the problem of multiplication, but was soon proven wrong by Anatoly Karatsuba, then a 23-year-old student. Karatsuba resorted to the following trick: given two n-digit numbers x and y in base B, he rewrote them as follows:

$$x = x_1 B^m + x_0,\ y = y_1 B^m + y_0$$

He then expressed their product $x \cdot y$ as:

$$x \cdot y = (x_1 B^m + x_0)(y_1 B^m + y_0) = z_2 B^{2m} + z_1 B^m + z_0$$

where

$$z_2 = x_1 \cdot y_1,\ z_1 = x_1 \cdot y_0 + x_0 \cdot y_1,\ z_0 = x_0 \cdot y_0$$

It might seem that $x \cdot y$ can only be computed using 4 multiplications, this being the number of multiplication required to calculate z_i. However, Karatsuba observed that they can be computed with just 3 multiplications, at the cost of few extra additions (subtractions) as follows:

$$z_1 = x_1 \cdot y_0 + x_0 \cdot y_1 = (x_0 - x_1) \cdot (y_1 - y_0) + z_0 + z_2$$

This way, the multiplication of two (potentially large) numbers is performed by first computing z_0 and z_2 (requiring two multiplications), followed by the computation of $z_1 = (x_0 - x_1) \cdot (y_1 - y_0) + z_0 + z_2$ (requiring one multiplication, 2 subtractions and two additions)[1]. Additions and subtractions can be computed in time linear to n (the number of digits), while the 3 multiplications can be recursively calculated using the same Karatsuba trick, until the multiplication becomes trivial (i.e. when x and y become smaller than B, their multiplication is performed in a single machine multiplication operation). For the special case when $m = \lceil n/2 \rceil$, i.e. when the numbers are split into (near) equal halves, the number of operations performed by the algorithm will be given by the following recurrence:

$$\begin{cases} T(n) = 3T(\lceil n/2 \rceil) + c \cdot n \\ T(1) = 1 \end{cases} \tag{2.3.1}$$

Using the Master Theorem (cf. [3]) it can be shown that $T(n) = \Theta(n^{log_2 3}) \approx \mathcal{O}(n^{1.58})$.

We have already alluded to the relatedness of convolution and multiplication and will now show that Karatsuba's trick can also be used to compute the convolution of two input time series x and h. We will look at the special case where $m = \lfloor n/2 \rfloor$ and n is the length of the shorter signal, viz. $n = \frac{\min(|h|,|x|)}{2}$, but this can be generalized for arbitrary m and n.

Let $\delta[k]$ be a time series containing the value 1 at index k and 0 elsewhere. Then it's convolution $x * \delta[k]$ with an arbitrary time series x results in time series x shifted by k samples. Moreover, for time series operands, let $+(-)$ operator be defined as sample-wise addition (subtraction). Similar to the integer multiplication algorithm, we can now rewrite x and h as follows:

$$x = x_1 * \delta[-\lfloor n/2 \rfloor] + x_0,\ h = h_1 * \delta[-\lfloor n/2 \rfloor] + h_0$$

[1] One can also compute $z_1 = (x_0 + x_1) \cdot (y_1 + y_0) - z_0 - z_2$, resulting in the same number of basic operations.

such that $|x_1| = |h_1| = \lfloor n/2 \rfloor$ and . The convolution of x and h is then given by:

$$x * h = (x_1 * \delta[-\lfloor n/2 \rfloor] + x_0) * (h_1 * \delta[-\lfloor n/2 \rfloor] + h_0) = z_2 * \delta[-2\lfloor n/2 \rfloor] + z_1 * \delta[-\lfloor n/2 \rfloor] + z_0$$

where

$$z_2 = x_1 * h_1,\ z1 = x_1 * h_0 + x_0 * h_1,\ z_0 = x_0 * h_0$$

All the above hold because convolution is, like multiplication, both associative and distributive over vector addition, and because $\delta[x] * \delta[y] = \delta[x + y]$. Hence, we can compute z_1 as before:

$$z_1 = (x_0 - x_1) * (h_1 - h_0) + z_2 + z_0$$

As in the case of multiplication, the addition and shifting operations can be performed in time linear to n, while for our choice of m the original convolution is split into three smaller ones. This results in the same recurrence $T(n)$ as in Equation 2.1 above, which entails that the new convolution algorithm also belongs to $\mathcal{O}(n^{1.58})$ class.

The pseudo-code of the algorithm is given in Alg. 3.

Algorithm 3 Karatsuba convolution algorithm

Input: x and h are two integer-valued time series.

```
1: function CONVOLVE(x, h)
2:     if |h| > |x| then
3:         return CONVOLVE(h, x)
4:     end if
5:     n ← |h|
6:     if n = 1 then
7:         return h[0] · x                          ▷ Scaling of x by h[0]
8:     end if
9:     x_1, x_0 ← SPLITAT(x, ⌊n/2⌋)                 ▷ Split x into two halves at index ⌊n/2⌋
10:    h_1, h_0 ← SPLITAT(h, ⌊n/2⌋)                 ▷ Split h into two halves at index ⌊n/2⌋
11:    z_0 ← CONVOLVE(x_0, h_0)
12:    z_1 ← CONVOLVE(x_0 − x_1, h_1 − h_0)
13:    z_2 ← CONVOLVE(x_1, h_1)
14:    return z_2 * δ[−2⌊n/2⌋] + (z_1 + z_2 + z_0) * δ[−⌊n/2⌋] + z_0
15: end function
```

2.4 Overlap-Add Method

We have seen in the Karatsuba algorithm how it is possible to split the input time series into two parts, perform smaller convolutions on the resulting splits and use these results to further construct the convolution of the bigger time series using point-wise addition. The Overlap-Add method operates in a similar fashion by splitting one of the inputs into k smaller parts (where $k \geq 2$), performing convolution between these and the other input, and then merging these results to form the convolution of the original time series. Following the conventions used in Signal Processing, we split x (called the input signal) into k smaller signals of equal length: $x_0, x_1, x_2, \ldots, x_{k-1}$ with $|x_0| = |x_1| = \cdots = |x_{k-1}| = |x|/k = l$, under the assumption that $|x|$ is divisible by k. The convolution of x by h (known as finite impulse response filter in Sig. Processing) is illustrated in Figure 2.1 and is given by the following equation:

$$\begin{aligned} x * h &= (x_0 + x_1 * \delta[l] + x_2 * \delta[2 \cdot l] + \cdots + x_{k-1} * \delta[(k-1) \cdot l]) * h \\ &= x_0 * h + (x_1 * h) * \delta[l] + (x_2 * h) * \delta[2 \cdot l] + \cdots + (x_{k-1} * h) * \delta[(k-1) \cdot l] \end{aligned} \tag{2.4.1}$$

The formula above entails that the entire convolution can be reconstructed from k convolutions between x_i and h after performing appropriate shifts and point-wise additions. Unlike Karatsuba, the Overlap-Add Method does not improve the time complexity of convolution. Nevertheless, it may yield speed improvements due to improved cache locality, and as will be shown, also guaranteed speed-ups for particular type of input time series.

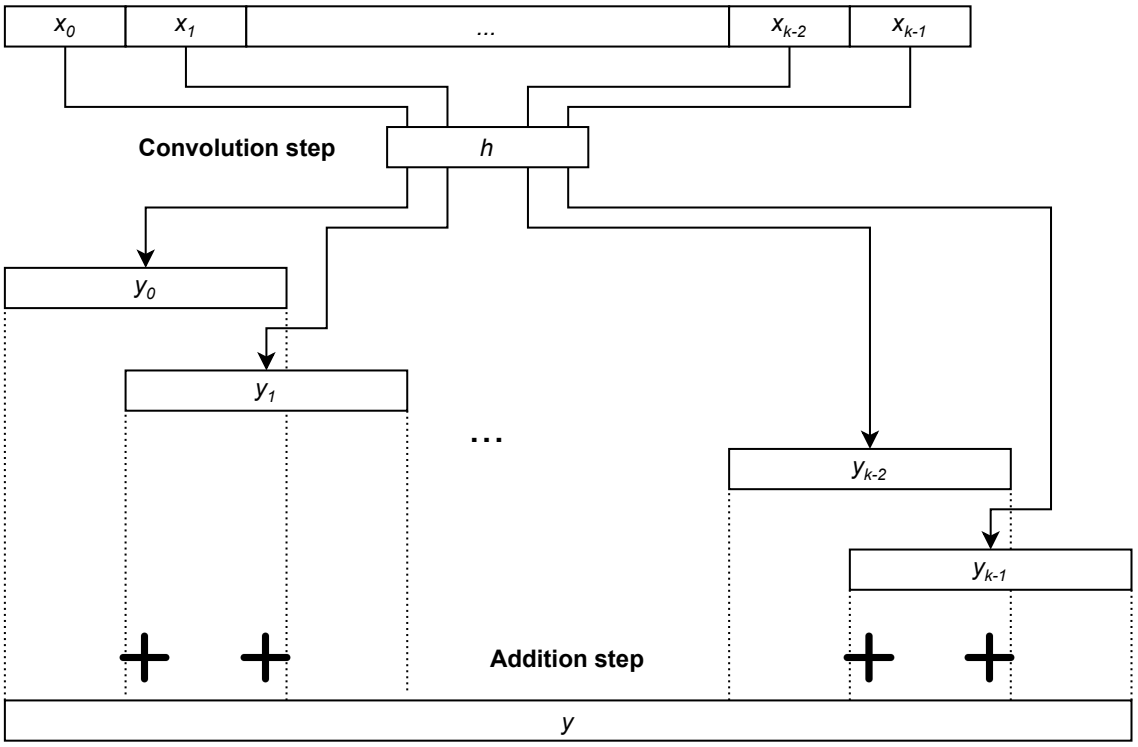

Figure 2.1: The illustration of the Overlap-Add Method

2.5 Toom-Cook algorithm

The Toom-Cook algorithm expands Karatsuba's idea in a way similar to the Overlap-Add Method. In particular, instead of splitting two n-digit numbers x and y in base B into two halves, it splits them into k parts, such that:

$$x = x_{k-1} \cdot B^{i(k-1)} + x_{k-2} \cdot B^{i(k-2)} + \cdots + x_1 \cdot B^i + x_0,\ y = y_{k-1} \cdot B^{i(k-1)} + y_{k-2} \cdot B^{i(k-2)} + \cdots + y_1 \cdot B^i + y_0$$

Where i is the number of digits per part and depends on the number of digits of the biggest input. We can then define the following two polynomials:

$$p(z) = x_{k-1} \cdot z^{k-1} + x_{k-2} \cdot z^{k-2} + \cdots + x_1 \cdot z + x_0,\ q(z) = y_{k-1} \cdot z^{k-1} + y_{k-2} \cdot z^{k-2} + \cdots + y_1 \cdot z + y_0$$

Let $r(z) = p(z) \cdot q(z)$. We then have that $x \cdot y = p(B^i)q(B^i) = r(B^i)$. To compute the coefficients of r, we can use the interpolation theorem, which says:

Theorem 2.5.1 (Interpolation theorem)**.** *Given $d + 1$ distinct points $a_0, a_1, \ldots, a_n$ and corresponding values $b_0, b_1, \ldots, b_n$, there exists a unique polynomial of degree at most d that interpolates $\{(a_0, b_0), \ldots, (a_n, b_n)\}$.*

Let $d = deg(r)$ be the degree of r. We compute d from the degree of p and q: $d = deg(r) = deg(p) + deg(q) - 1 = k + k - 1 = 2k - 1$. By the interpolation theorem, we can find the coefficients r_i of r if we have d distinct points a_i and corresponding values b_i, where $0 \leq i < d$ such that $r(a_i) = b_i$. Since $r(a_i) = p(a_i) \cdot q(a_i)$ and $p(z)$ and $q(z)$ are known, a pair (a_i, b_i) can be computed by evaluating both p and q at a_i and taking the product of these result. The a_i points can be picked arbitrary and only need to be distinct from one another.

Using matrix notation, we can evaluate a_i at p and q and then interpolate the coefficients of r:

$$\begin{pmatrix} p(a_0) \\ p(a_1) \\ \vdots \\ p(a_{d-1}) \end{pmatrix} = \begin{pmatrix} a_0^0 x_0 & a_0^1 x_1 & \dots & a_0^{d-1} x_{d-1} \\ a_1^0 x_0 & a_1^1 x_1 & \dots & a_1^{d-1} x_{d-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{d-1}^0 x_0 & a_{d-1}^1 x_1 & \dots & a_{d-1}^{d-1} x_{d-1} \end{pmatrix} = \begin{pmatrix} a_0^0 & a_0^1 & \dots & a_0^{d-1} \\ a_1^0 & a_1^1 & \dots & a_1^{d-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{d-1}^0 & a_{d-1}^1 & \dots & a_{d-1}^{d-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{d-1} \end{pmatrix}$$

Let $\mathcal{M} = \begin{pmatrix} a_0^0 & a_0^1 & \dots & a_0^{d-1} \\ a_1^0 & a_1^1 & \dots & a_1^{d-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{d-1}^0 & a_{d-1}^1 & \dots & a_{d-1}^{d-1} \end{pmatrix}$, then $\begin{pmatrix} q(a_0) \\ q(a_1) \\ \vdots \\ q(a_{d-1}) \end{pmatrix} = \mathcal{M} \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{d-1} \end{pmatrix}$, $\begin{pmatrix} r(a_0) \\ r(a_1) \\ \vdots \\ r(a_{d-1}) \end{pmatrix} = \mathcal{M} \begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_{d-1} \end{pmatrix}$

The last equation entails: $\begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_{d-1} \end{pmatrix} = \mathcal{M}^{-1} \begin{pmatrix} r(a_0) \\ r(a_1) \\ \vdots \\ r(a_{d-1}) \end{pmatrix} = \mathcal{M}^{-1} \left(\begin{pmatrix} p(a_0) \\ p(a_1) \\ \vdots \\ p(a_{d-1}) \end{pmatrix} \cdot \begin{pmatrix} q(a_0) \\ q(a_1) \\ \vdots \\ q(a_{d-1}) \end{pmatrix} \right)$.

The Toom-Cook algorithm normally picks the smallest possible points a_i. For example, a typical choice for $k = 3$ would be the points $\{-2, -1, 0, 1, 2\}$. By doings so, the algorithm ensures that all values in $\mathcal{M}$ and $\mathcal{M}^{-1}$ are as small as possible as well, and all multiplications involved in the computations of the $(p(a_0), p(a_1), \dots, p(a_{d-1}))$ and $(q(a_0), q(a_1), \dots, q(a_{d-1}))$ vectors are multiplications with a small constant factor, and can be therefore reduced to a constant number of additions, which can be performed in linear time. It is worth noting that both $\mathcal{M}$ and it's inverse $\mathcal{M}^{-1}$ only need to be pre-computed once and can be reused for the same instance of k and therefore add no time overhead. Similarly, the computation of $r(B^i) = x \cdot y$ can be done in linear time as it only involves additions and multiplications with a power of the base, as the later can be achieved in constant time using shifts. The only calculation that requires non-linear time is the vector dot-product in the computation of $(r_0, r_1, \dots, r_{d-1})$. This requires d multiplications of big numbers, which are smaller instances of our original problem. Summing everything up, we need to perform d instances of the original problem reduced by a factor of k, plus a constant linear time. This yields the following recurrence:

$$\begin{cases} T(n) = dT(n/k) + c \cdot n = (2k-1) \cdot T(n/k) + c \cdot n \\ T(1) = 1 \end{cases} \tag{2.5.1}$$

For all $k > 1$ it holds that $d = 2k-1 > k$. Since every recursive step will require linear time to merge the intermediate results, by the third case of Master Theorem: $T(n) = \Theta(n^{\log_k d}) = \Theta(n^{\log_k 2k-1})$. For the special case where $k = 1$, the algorithm simply degenerates to long hand multiplication, which is $\mathcal{O}(n^2)$, while for $k = 2$ and $\mathcal{M} = \left(\begin{smallmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{smallmatrix}\right)$ it is equivalent to Karatsuba algorithm.

The result above implies that the higher the value of k is (the more parts we choose to split the input in), the lower the time complexity will be. In particular, for $k = 3$, $T(n) = \Theta(n^{\log_3 5}) \approx \mathcal{O}(n^{1.46})$, for $k = 4$, $T(n) = \Theta(n^{\log_4 7}) \approx \mathcal{O}(n^{1.40})$ and so on. In theory, as k reaches infinity $T(n) = \Theta(n)$ since $\lim_{k \to \infty} \log_k (2k-1) = 1$.

Unfortunately all these asymptotic complexities fail to reflect an important property which has a critical significance for practical purposes - the growth of the constant factor. In 2.5.1 this constant factor is denoted as c, which does not express the fact that it grows significantly as k increases. To illustrate this growth, consider the case where $k = 1024$, which will yield an algorithm with a nice asymptotic complexity of $\Theta(n^{log_{1024} 2043}) \approx \mathcal{O}(n^{1.10})$. In this case, not only will $\mathcal{M}$ be a 2043×2043 matrix, but it's biggest element will be at least 1024^{2043}, a number with more than 6000 digits. Although the computations involving $\mathcal{M}$ are independent of n, they are clearly unfeasible. For this reason, it is rarely the case that values of $k > 4$ are used in practice [15].

So far, we have discussed the Toom-Cook algorithm in the context of n-digit integers, which might seem digressing from the topic of convolution. Nonetheless, using similar arguments as in Section 2.3 the integer algorithm described above can be adapted to the convolution of time series of length n. Alternatively, the time series can be converted to/from integers in some base $\mathcal{B}$ which is equal

to the difference between the maximum and the minimum elements found in both time series and the multiplication algorithm can be used.

Unfortunately, in practice, the constant factor in the time complexity of the convolution adapted version tends to scale even worse as k increases. A particular bottleneck is the memory required for the storage of intermediate results. Albeit there exist solutions that make this memory requirements constant, as will be shown later in this thesis, the overhead is still big enough to make even the Karatsuba convolution (or Toom-Cook with $k = 2$) of short time series slower than other convolution algorithms. There also exist methods that further reduce the asymptotic time complexity and the constant factor [9]. Nevertheless, convolutions based on the Toom-Cook algorithm seem to be rarely used in practice.

Although it might seem counter-intuitive, Toom-Cook algorithm is also sub-optimal for any value of k. As mentioned previously, as k increases the time complexity will tend to $\mathcal{O}(n^{1+\epsilon})$, with ϵ becoming closer and closer but never reaching 0. We can prove that for any $\epsilon > 0$, $n \log n = \mathcal{O}(n^{1+\epsilon})$ as follows:

$$\lim_{n\to\infty} \frac{n \log n}{n^{1+\epsilon}} = \lim_{n\to\infty} \frac{\log n}{n^{\epsilon}} = \lim_{n\to\infty} \frac{(\log n)'}{(n^{\epsilon})'} = \lim_{n\to\infty} \frac{\frac{1}{n}}{\epsilon n^{\epsilon-1}} = \lim_{n\to\infty} \frac{1}{\epsilon n^{\epsilon}} = 0$$

More informally, even for an infinitesimal ϵ, as n will go to infinity $n^{1+\epsilon}$ will be greater than $n \log n$. This proves that for a big value of k, not only is Toom-Cook algorithm practically unfeasible, but still remains theoretically sub-optimal.

2.6 Fast Fourier transform convolution

We have now reached the point where we can introduce the fastest practical convolution algorithm that is currently known. The algorithm is based on the so-called 'convolution theorem' that states that

$$(\mathcal{F}\{x * h\})[n] = (\mathcal{F}\{x\} \odot \mathcal{F}\{h\})[n]$$

Here, $\mathcal{F}\{x\}$ denotes the *discrete Fourier transform* (DFT) of the vector x, and $x \odot y$ is the vector that is obtained by pairwise multiplication of the elements of x and y (i.e. $[a, b, c] \odot [d, e, f] = [a \cdot d, b \cdot e, c \cdot f]$).

For inputs that consist of a number of elements that is a power of two, the DFT can be computed with a specialised algorithm called the '*Fast Fourier transform*' (abbreviated as FFT), which has an asymptotic time complexity of $\mathcal{O}(n \log n)$. In particular, we will look at the recursive *radix-2* algorithm, which is part of a large family of FFT algorithms based on the seminal paper of James Cooley and John Tukey published in 1965 [5].

The algorithm uses the properties of a *primitive N-th root of unity* - ω which is defined as the complex number that satisfies the following two conditions:

$$\omega^N = 1 \textbf{ and } \omega^m \neq 1 \text{ for } 1 \leq m < N$$

For an even $N > 2$, clearly ω^2 is also a primitive $(N/2)$-th root of unity because $\omega^N = (\omega^2)^{N/2} = 1$ and $\omega^{2k} = (\omega^2)^k$ unique for $0 \leq k < N/2$. Moreover, using geometric series equality $\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$ for $x \neq 1$ we can prove *cancellation property*:

$$\sum_{j=0}^{N-1} \omega^{kj} = \frac{(\omega^k)^N - 1}{\omega^k - 1} = \frac{1^k - 1}{\omega^k - 1} = \frac{0}{\omega^k - 1} = 0 \text{ for } 0 \leq k < N$$

For the special case $k = N/2$, by the property above we have:

$$0 = \sum_{j=0}^{N-1} \omega^{(N/2)j} = 1 + \omega^{N/2} + \omega^N + \omega^{N+N/2} + \cdots + 1 + \omega^{N/2} = (N/2)(1 + \omega^{N/2})$$

Since $N > 0$ it follows that $\omega^{N/2} = -1$. This, in turn, entails that for any k we have $(\omega^{k+N/2})^2 = \omega^{2k+N} = \omega^{2k} \cdot \omega^N = \omega^{2k} = (\omega^k)^2$. (2.6.1)

For two input time series x and h of length N, where N is a power of two ($N = 2^k$ for some $k \in \mathbb{N}$), the first step of the *radix*-2 algorithm is to compute their Discrete Fourier Transform (abbr. DFT).

For an arbitrary time series $a = [a_0, a_1, \ldots, a_{N-1}]$ and ω a primitive N-th root of unity, it's DFT is given by:

$$\mathrm{DFT}(a) = [X_0, X_1, \ldots, X_{N-1}] \text{ where } X_k = \sum_{n=0}^{N-1} a_n \omega^{nk}$$

In other words, the DFT contains the results of evaluation of the polynomial corresponding to a at all powers of ω smaller than N. As already mentioned, for any time series a, it's corresponding polynomial can be constructed as follows: $p(x) = \sum_{n=0}^{N-1} a_n x^n$. Therefore $X_k = p(\omega^k)$, and thus the DFT can be defined in a similar fashion to the polynomial evaluation in the Toom-Cook algorithm using matrix notation:

$$\begin{aligned}\mathrm{DFT}(a) &= \begin{pmatrix} p(\omega^0) \\ p(\omega^1) \\ \vdots \\ p(\omega_{N-1}) \end{pmatrix} = \begin{pmatrix} (\omega^0)^0 a_0 & (\omega^0)^1 a_1 & \ldots & (\omega^0)^{N-1} a_{N-1} \\ \omega^0 a_0 & \omega^1 a_1 & \ldots & \omega^{N-1} a_{N-1} \\ \vdots & \vdots & \ddots & \vdots \\ (\omega^{N-1})^0 a x_0 & (\omega^{N-1})^1 a_1 & \ldots & (\omega^{N-1})^{N-1} a_{d-1} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 1 & \ldots & 1 \\ 1 & \omega^1 & \ldots & \omega^{d-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & (\omega^{N-1})^1 & \ldots & (\omega^{N-1})^{N-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{N-1} \end{pmatrix}\end{aligned}$$

The matrix $\mathcal{M}$ obtained in the second equation is a Vandermonde matrix and can be proven to be invertible for any ω that is a primitive N-th root of unity. By denoting the inverse of DFT as $\mathrm{IDFT}(a) = \mathcal{M}^{-1}[X_0, X_1, \ldots, X_{N-1}]$, it is easy to conclude, based on the *Interpolation theorem*, that for any two time series x and h, their convolution will be equal to:

$$x * h = \mathrm{IDFT}(\mathrm{DFT}(x) \cdot \mathrm{DFT}(h))$$

The formula above requires at least $N = |x| + |h| - 1$ points to be evaluated correctly. Contrary to the Toom-Cook algorithm, it also involves multiplications of matrices and vectors whose size are proportional to N and can not be computed in constant time. It's asymptotic time complexity is therefore influenced by 3 $N \times N$ matrix-vector multiplications and N value multiplications, which together result in $\mathcal{O}(3N^2 + N) = \mathcal{O}(N^2)$, which is sub-optimal if computed directly from the definition. The radix-2 FFT algorithm computes the products by $\mathcal{M}$ and $\mathcal{M}^{-1}$, or the DFT and it's inverse in $\mathcal{O}(N \log N)$ time by exploiting the previously proven properties of roots of unity - 2.6.1. To do so, we first have to rewrite $p(x)$ as follows:

$$p(x) = \sum_{n=0}^{N-1} a_n x^n = p_0(x^2) + x p_1(x^2) \text{ where } p_0(x) = \sum_{n=0}^{N/2-1} a_{2n} x^n \text{ and } p_1(x) = \sum_{n=0}^{N/2-1} a_{2n+1} x^n$$

Therefore, for $k < N/2$, $X_k = p(\omega^k) = p_0((\omega^k)^2) + \omega^k p_1((\omega^k)^2)$ and $X_{k+N/2} = p(\omega^{k+N/2}) = p_0((\omega^{k+N/2})^2) + \omega^{k+N/2} p_1((\omega^{k+N/2})^2) = p_0((-\omega^k)^2) + (-\omega^k) p_1((-\omega^k)^2) = p_0((\omega^k)^2) - \omega^k p_1((\omega^k)^2)$. This entails that by having ω^k, $p_0((\omega^k)^2)$ and $p_1((\omega^k)^2)$ for all $k < N/2$, we can compute all X_k in $\mathcal{O}(N)$ time.

We recall that, under the assumptions of the *radix-2* algorithm, N will be a power of two and ω will be a N-th root of unity. Therefore, if $N > 1$, $N/2$ will also be a power of two and ω^2 will be a $(N/2)$-th root of unity. This invariant allows us to apply the trick described above to recursively compute, for all $k < N/2$, $p_0((\omega^k)^2) = p_0(\omega_{N/2}^k)$ and $p_1((\omega^k)^2) = p_1(\omega_{N/2}^k)$ where $\omega_{N/2}^k = (\omega^k)^2$ is a $(N/2)$-th root of unity. If $N = 1$ (base case), computing p is trivial and can be done in constant $\mathcal{O}(1)$ time.

To compute the IDFT, it can be proven that for any entry $\mathcal{M}[i, j] = \omega^{ij}$ the corresponding one in the inverse matrix will be $\mathcal{M}^{-1}[i, j] = \omega^{-ij}/N$. Therefore the same recursive procedure can be

used to compute the multiplication by $\mathcal{M}^{-1}$, except the inverse of ω is used and the results scaled by $1/N$. This is because $\omega^{-ij} = (\omega^{-1})^{ij} = \text{inv}(\omega)^{ij}$.

The last step is to compute the primitive N-th root of unity - ω. This can be done easily in complex plane by using the formula $\omega = e^{i2\pi/N}$.

Summing everything up, to compute the DFT/IDFT of length N, we recursively split the problem into two sub-problems that both are half the size of the original problem, and require cN operations, where c is a constant, to reconstruct the final result. This yields $T(N) = 2T(N/2)+cN$, and according to the master theorem this results in $T(N) = \mathcal{O}(N \log N)$. The asymptotic time complexity of computing the convolution of 2 time series of length N is then $\mathcal{O}(3(2N \log 2N)+2N) = \mathcal{O}(N \log N)$.

The pseudo-code of the FFT-based convolution algorithm is given in Alg. 4.

Algorithm 4 Radix-2 recursive convolution algorithm

Input: x and h are two integer-valued time series.

function CONVOLVE(x, h)
  $N \leftarrow$ MINGEQPOWTWO($|h| + |x| - 1$)) ▷ Minimum power of two greater or equal
  $x \leftarrow$ ZEROPAD(x, N) ▷ Zero pads to the specified length
  $h \leftarrow$ ZEROPAD(h, N) ▷ Zero pads to the specified length
  $\omega \leftarrow e^{2i\pi/N}$
  return IFFT(FFT(h, ω) $\cdot$ FFT(x, ω), ω)
end function

function FFT(a, ω)
  $N \leftarrow |a|$
  if $N = 1$ **then**
    return $y = a$
  end if
  $x \leftarrow \omega^0$
  $a^{\text{even}} \leftarrow [a_0, a_2, \ldots, a_{N-2}]$
  $a^{\text{odd}} \leftarrow [a_1, a_3, \ldots, a_{N-1}]$
  $y^{\text{even}} \leftarrow \text{FFT}(a^{\text{even}}, \omega^2)$
  $y^{\text{odd}} \leftarrow \text{FFT}(a^{\text{odd}}, \omega^2)$
  for $i \leftarrow 0$ to $N/2 - 1$ **do**
    $y_i \leftarrow y_i^{\text{even}} + x \cdot y_i^{\text{odd}}$
    $y_{i+N/2} \leftarrow y_i^{\text{even}} - x \cdot y_i^{\text{odd}}$
    $x \leftarrow x \cdot \omega$
  end for
  return y
end function

function IFFT(a, ω)
  return $1/N \cdot \text{FFT}(a, \overline{\omega})$ ▷ $1/N$ is sample-wise scaling, $\overline{\omega}$ is complex conjugate
end function

2.7 Number-theoretic transform convolution

The Number-theoretic transform (NTT) convolution is similar to the FFT convolution described in the previous section, except that all the operations take place in a different algebraic ring structure. The ring used for the standard FFT is the ring of complex numbers with their corresponding standard addition and multiplication operators. The NTT uses the ring of integers modulo a number M, and the addition and multiplication operations are taken modulo that number M. In the context of number theory this ring is usually denoted as $\mathbb{Z}/M\mathbb{Z}$. Using NTTs has the benefit of not requiring to move to/from the complex plane, in which the algebraic operations can be more expensive to perform and where round-off errors might result as a consequence of floating-point

arithmetic.

As in the FFT algorithm, ω must be an N-th root of unity, in other words $\omega^N \equiv 1 \pmod{M}$.[2] As opposed to the FFT however, ω being a *primitive* N-th root of unity is an insufficient condition, as it is not the case anymore that in an arbitrary ring $\omega^{N/2} = -1$. As an example, consider $N = 256, M = 65535, \omega = 7$. Although 7 is a 256-th root of unity modulo 65535, $\omega^{N/2} = 7^{128} \equiv 256 \not\equiv -1 \equiv 65534 \pmod{65535}$. This has to do with the fact that in a ring, division by another integer a is defined as multiplication with the modular inverse a^{-1}, where $a \cdot a^{-1} = 1$ which does not always have a solution a^{-1}. This is also the case for the example above, for which $(6)^{-1}$ does not exist, and therefore the cancellation property can not hold for $k = 1$, as the inverse of the denominator in the expanded fraction $\omega^k - 1 = 7^1 - 1 = 6$ does not exist. These cases can be avoided by requiring that ω is a *principal root of unity* instead. An ω is a principal root of unity if

$$\omega^N \equiv 1 \pmod{M} \textbf{ and } \sum_{j=0}^{N-1} \omega^{jk} \equiv 0 \pmod{M} \text{ for } 1 \leq k < N$$

Another alternative, which seems to be the more preferred choice in practice, is to require that the modulus M is prime. This ensures that for all $1 \leq a < M$, $\gcd(a, M) = 1$ and therefore a^{-1} exists. For such M, the division in the cancellation property is defined for any k and therefore it is sufficient that ω is a *primitive* N-th root of unity, as in the case of FFT.

Another necessary condition for the NTT convolution to work is the existence of the modular inverse of N [1]. This guarantees that the matrix $\mathcal{M}$, which is now a modular matrix, is invertible, and therefore the IDFT exists. Having a prime M ensures this is the case as well.

In principle, these are all the conditions that need to be satisfied to be able to derive the same recurrences as in FFT convolution and therefore implement a working NTT convolution that has an optimal time complexity.

The same pseudo-code for *recursive radix-2 convolution* as in **Algorithm 4** can then be used, with a few modifications:

1. **[line 5]** In general, given a modulo M, there is no straightforward way to compute the N-th *principal* (*primitive*, for prime M) root of unity. Usually, the N-th roots of unity are pre-computed for powers of two and are stored in a look-up table. Finding the *primitive* N-th root of unity on the spot using brute-force can be done in $\mathcal{O}(M(\log N + N))$ time.
2. **[line 26]** Instead of $1/N$ and $\overline{\omega}$ we take N^{-1} and ω^{-1}. These can be pre-computed as well.
3. All arithmetic operations are taking place modulo M.

Despite its advantages, the NTT has a big disadvantage compared to the FFT. Let $N_{\max}$ denote the maximum length of a time series for which it is possible to compute the NTT and its inverse (INTT), then $N_{\max} < M$ holds trivially. This is because, there exist at most M distinct elements in our ring, and therefore it is impossible to apply the interpolation theorem in this case. Assuming that $M = p_1^{r_1} p_2^{r_2} \dots p_l^{r_l}$, it can be proven [1] that the claim can further be strengthened to $N_{\max} = O(M)$, where:

$$O(M) = \gcd\{p_1 - 1, p_2 - 1, \dots, p_l - 1\}$$

This implies that the length of the input time series is limited by the choice of M, and in general is least restrictive when M is prime, as $M = p_1$ and $N_{\max} = O(M) = \gcd\{M - 1\} = M - 1$. This is not the case for the FFT algorithm, which is only limited by the scale of the floating point numbers round-off errors.

Summing everything up, NTT convolution is a similar but less error-prone algorithm, whose main limitations are the choice of ω and M, and the length of input time series.

[2] Note that we will assume that all algebraic operations in this section, besides the ones in the exponent, are taking place modulo M

Chapter 3

Improving convolution algorithms

In this section we look at possible improvements and optimizations that could benefit the algorithms described in the previous section. We will also present several novel sub-optimal algorithms and investigate the conditions that warrant their use.

3.1 Practical NTT

In the last section of the previous chapter we have provided the general blueprint for building a working recursive *radix-2* NTT convolution algorithm. As opposed to the FFT, for which there exists a direct and easy way to compute an N-th primitive root of unity ω for any value of N in the complex plane, in the case of NTT the choice of ω, as well as the integer modulo M are left at the discretion of the programmer. In general, few choices of M have *principal* (*primitive*, if M is prime) N-th roots of unity and existing inverses for high values of N in the corresponding ring. Not only that, given N and M it is also computationally expensive to find such a value, and it might not even exist in the first place. For this reason, a practical NTT implementation would require a preceding wise choice of M and ω, where the later is a *principal* root of unity for a sufficiently big power of two. This is because, given such a value, principal roots of unity for smaller powers of two can be easily computed by repeatedly squaring ω. Lastly, the chosen M should also be sufficiently large to contain both the maximum value of the desired convolution and its length in order to produce a correct result.

A possible choice for the modulus [10] is $M = 9223372036737335297 = 54975513881 \cdot 2^{24} + 1$, which is a prime slightly smaller than the maximum value of a signed 64 bit integer. Not only is this value big enough to contain the result of a convolution involving big numbers, but also a 2^{24}-th primitive root of unity is known to exist - $\omega = 2419180138865645092$. It therefore also allows the computation of a sufficiently large convolution (up to lenght 2^{24}). Moreover, since it can be represented using less than 63 bits, the modular addition operation can be implemented using unsigned 64 bits integers, which is implemented in hardware on most machines and is therefore efficient.

To further improve the running time, all N-th primitive roots of unity, as well as inverses N^{-1} can be pre-computed and stored in a lookup table for all N that are powers of two equal or smaller than 2^{24}. This table is given in Appendix A.

3.2 Barrett reduction

Despite its advantages, the modulus proposed above has a major bottleneck - modular multiplication. Since $M < 2^{63}$, in order to compute a modular product we will need to take an intermediate $2 \cdot 63 = 126$ bit number modulo M - a 63 bit number, which is not an operation supported by

many architectures. Compiling and executing such a code on a modern machine using 128 (which might not even be supported in the first place) and 64 bits variables, would most likely result in long division being performed under the hood, which is extremely inefficient.

Fortunately there exists a solution [10] that mitigates this problem by exploiting the fact that our modulo M is fixed, which is known as Barrett reduction. In particular, given an integer a, we can compute its remainder modulo M, $x \equiv a \pmod{M}$ using long division as follows: $x = a - a/M \cdot M$, where division operator represents integer division. The Barrett reduction replaces integer division by M with two less costly operations: multiplication by a constant integer m and division by a power of two - 2^k (which can be implemented using bit shift operations), where $\frac{m}{2^k} \approx \frac{1}{M}$. Since the former operations are only an approximation, there is an error of $\epsilon = \frac{1}{M} - \frac{m}{2^k}$, and therefore the reduction will only work if $x < \frac{1}{\epsilon}$.

For our previous 63 bit modulus, it can be proven [2] that to be able to correctly reduce the product of any two 63 bit numbers smaller than M, it is sufficient to choose any $k \in \mathbf{N}$ such that $2^k > M$ and perform division by $2^{2k} = 4^k$. From $\frac{m}{4^k} \approx \frac{1}{M}$, we can then compute $m = \left\lfloor \frac{4^k}{M} \right\rfloor$. In particular, by picking the smallest valid value $k = 63$, we can find $m = 9223372036972216319$. Then $x \equiv a = b \cdot c \pmod{M}$, where $0 \leq b, c < M$ can be calculated as follows $x = a - \lfloor \frac{am}{4^k} \rfloor m$, which only requires an addition, multiplication and division by a power of two. Because we chose to round down $\frac{4^k}{M}$ when computing m, x is guaranteed to be in the range $[0, 2M)$. This can be easily corrected to $[0, M)$ by checking if $x \geq M$ and subtracting M if this is the case.

Using bitwise manipulations, it is possible to implement the computation of the above formula using only 64 bit intermediate variables, which can generate further speedup. In particular. in a similar fashion to Karatsuba, we can write two 64 bit integers a and b as $a = a_h \cdot 2^{32} + a_l$ and $b = b_h \cdot 2^{32} + b_l$. Their product is then given by $x = a \cdot b = a_h \cdot b_h \cdot 2^{64} + (a_h \cdot b_l + a_l \cdot b_h) \cdot 2^{32} + a_l \cdot b_l = z_0 \cdot 2^{64} + z_1 \cdot 2^{32} + z_2$ where $z_0, z_1, z_2 < 2^{64}$. We can then compute x_l by summing the lowest 32 bits of z_1 times 2^{32} and z_2, and in a similar fashion x_h by summing the higher 32 bits of z_1 and z_0 times 2^{32}, plus one if there is a carry from x_l. Then $x = x_h \cdot 2^{64} + x_l$.

C code that implements the Barrett reduction using 64 bit variables is given in Appendix B.

3.3 NTT without an inverse of N and Chinese Remainder Theorem

So far, it has been suggested that in order to compute a correct convolution $y_c = x * h$ using the recursive *radix-2* NTT algorithm with modulo M, all samples in the output time series need to be smaller than M and $N^{-1} \pmod{M}$ needs to exist, where N is the smallest power of two greater than the length of the convolution $|y| = |x| + |h| - 1$. It turns out that computing a convolution that disregards these conditions still yields meaningful and potentially useful results.

Let y_c be the result of the correct convolution and y the one of the convolution that disregards the two assumptions. Then in the case the original output contains samples greater than the modulus M, it is easy to conclude that the final convolution using NTT modulo M will yield $y \equiv y_c \pmod{M}$. In other words, all samples in the output will be taken modulo M. For the second case, when $N^{-1} \pmod{M}$ does not exist, the scaling by N^{-1} can be omitted from each element of $\mathcal{M}^{-1}$, and the final result of the convolution will yield $y \equiv N y_c \pmod{M}$. In this case, by further assuming that all samples in the resulting convolution are smaller than $\lfloor \frac{M}{N} \rfloor$, then $y_c = y/N$, where division operator represents sample-wise integer division by length N. To put it informally, when no scaled sample in the resulting convolution overflows M, the original sample can be computed by dividing the former by the length N. This particular observation allows us to extend our choice of M to even ones as well. This was not possible under the previous assumption because for any M divisible by 2, $\gcd(M, N) = \gcd(M, 2^k) \geq 2 \neq 1$ and therefore N^{-1} does not exist.

Since operations modulo a power of two are easy to compute, this raises the question whether such moduli have N-th *principal* roots of unity for lengths N that are themselves powers of two.

Unfortunately our findings lead to a negative conclusion. For various N that are powers of two, we were only able to find N-th *principal* roots of unity for moduli that were powers of two smaller than or equal to N. This motivates us to conjecture the following:

> For any $N = c \cdot 2^k$, $c, k \in \mathbb{N}^+$ and $2 \nmid c$, there does not exist any N-th *principal* root of unity modulo M, where $M = 2^l$ and $l > k$.

This implies that convolutions using power of two moduli always lead to null time series, since all samples are scaled by a factor which is a power of two and taken modulo another equal or smaller power of two.

Let us now investigate the case where k convolutions $y_0, y_1, \ldots, y_{k-1}$ with moduli $M_0, M_1, \ldots, M_{k-1}$ contain samples congruent to samples in y_c that are bigger than the moduli. To reconstruct such samples, given that all moduli are pairwise coprime ($gcd(M_i, M_j) = 1$ for all $0 \leq i < j < k$) and the samples are smaller than their product, the following constructive proof of the *Chinese Remainder Theorem*[1] can be used:

> Let $x \equiv a_0 (\text{mod } M_0), x \equiv a_1 (\text{mod } M_1), \ldots, x \equiv a_{k-1} (\text{mod } M_{k-1})$ be a system of congruences, where moduli are pairwise coprime and P be the product of all moduli $P = \prod_{i=0}^{k-1} M_i$. Then a solution to the system is given by:
>
> $$x \equiv \sum_{i=0}^{k-1} a_i \mathcal{I}_i N_i \text{ (mod } P), \text{ where } P_i = P/M_i \text{ and } \mathcal{I}_i \equiv P_i^{-1} \text{ (mod } M_i)$$
>
> This follows directly from the fact that $P_i \equiv 0 \text{ (mod } M_j)$ for all $i \neq j$, and therefore:
>
> $$x \equiv a_i \mathcal{I}_i P_i \equiv a_i P_i^{-1} P_i \equiv a_i \text{ (mod } M_i)$$

According to the Chinese Remainder Theorem, we can obtain a new convolution y' modulo $P = \prod_{i=0}^{k-1} M_i$, such that $y' \equiv y_c \text{ (mod } P)$. Assuming that the maximum sample in y_c is smaller than P, it will follow that $y' = y_c$. An example of convolution with small moduli and reconstruction of the correct convolution using Chinese Remainder Theorem is given in Table 3.1.

x	4	2			
h	3	2	1		
y_0	0	2	2	2	mod 3
y_1	2	4	3	2	mod 5
y_c	12	14	8	2	mod 15

Table 3.1: An example of convolution with moduli 3 and 5 and reconstruction using CRT.

The use of Chinese Remainder Theorem in the context of NTT is particularly appealing when convolutions yielding big samples are required. Since infinitely many primes exist, we can perform an arbitrary number of NTT convolutions using distinct primes (which are coprime by definition) as moduli, such that their product is greater than the biggest sample in the output time series. Then the original convolution can be reconstructed using a CRT solving algorithm. Let k be the number of convolutions performed using k coprime numbers, then a constant amount of time is required to compute each sample of the output after solving k instances of NTT convolution. Assuming the convolution has N samples, the running time of the extended algorithm is $\mathcal{O}(kN \log N + kN) = \mathcal{O}(N \log N)$, which entails that the extended algorithm preserves the optimal time complexity.

Note that for each NTT convolution to compute a correct result, N must be smaller than the modulus. This implies that we need $\min\{M_0, M_1, \ldots, M_k\} > N$ for the extended algorithm to work. Moreover, the scaling by N^{-1} can be omitted from all k convolutions given that the biggest sample is smaller than $\left\lfloor \frac{M'}{N} \right\rfloor$. In this case $y' = N y_c$ and the final result can be obtained by doing sample-wise integer division by N afterwards. This allows for a single modulus to be even, but not more than one since there can not exist a pair of even coprime numbers.

[1] Note that the original theorem only asserts the existence of x, and does not provide a way to compute it.

3.4 NTT with a Mersenne number modulus

Although, as conjectured above, we were unable to find any N-th *principal* roots of unity for both convolution length N and modulo M being powers of two, this has also motivated us to search for other moduli that would be efficient to compute. As a result, we have identified that computing modulo operation under a modulus M which is a Mersenne number, that is $M = 2^k - 1$ for some $k \in \mathbb{N}$, or one less than a power of two, would also be efficient. In particular, we can rewrite any number $0 \leq a < M^2 = (2^k - 1)^2$ as $a = 2^k a_0 + a_1$ where $a_0, a_1 <= 2^k$. Then, by denoting a **mod** M as the remainder of Euclidean division of a by M, it holds that a **mod** $M \equiv (2^k a_0 + a_1)$ **mod** $M = ((2^k a_0)$ **mod** $M + a_1$ **mod** $M)$ **mod** $M = ((2^k$ **mod** $M \cdot a_0$ **mod** $M)$ **mod** $M + a_1$ **mod** $M)$ **mod** M. Since $M = 2^k - 1$, 2^k **mod** $M = 1$, it follows that a **mod** $M = ((1 \cdot a_0$ **mod** $M)$ **mod** $M + a_1$ **mod** $M)$ **mod** $M = (a_0$ **mod** $M + a_1$ **mod** $M)$ **mod** M. Because $a_0, a_1 <= 2^k$, computing the last formula is trivial.

Unfortunately, as in the case of powers of two, although we were able to find *primitive* N-th roots of unity for various moduli that were composite Mersenne numbers, these turned out to be insufficient. In particular, as mentioned above, for composite M, the roots of unity need to be *principal*. This is because the cancellation property might not hold anymore, and as a consequence, $\omega^2 \not\equiv -1 \pmod{M}$, which is a necessary condition for the algorithm to work. In fact, the example given in Section 2.7 has a modulus which is a Mersenne number.

For $N > 2$, our searches of a N-th *principal* root of unity modulo a Mersenne number M did not yield any results, which makes us conjecture that such values do not exist. For this reason, we believe that Mersenne numbers and the trick to compute the modulo such a number described above can unfortunately not be used in the context of NTT.

3.5 Overlap-add FFT

A big shortcoming of the FFT and NTT *radix-2* convolution algorithms is the need to pad both inputs x and h to a length $N = 2^k \geq |x| + |h| - 1$, for some $k \in \mathbb{N}$. This implies that for the case where $|h| \ll |x|$ (or vice-versa) the running time of the algorithm will be significantly influenced by the longest time series. By extending the idea described in Section 2.4 we can potentially reduce the running time for these cases at the cost of some additional overhead [11].

In particular, Equation 2.4.1 can be extended as following:

$$\begin{aligned} x * h &= x_0 * h + (x_1 * h) * \delta[l] + (x_2 * h) * \delta[2 \cdot l] + \cdots + (x_{k-1} * h) * \delta[(k-1) \cdot l] \\ &= \text{IDFT}(\text{DFT}(x_0) \odot \text{DFT}(h)) + \ \text{IDFT}(\text{DFT}(x_1) \odot \text{DFT}(h)) * \delta[l] + ... \\ &\quad ... + \text{IDFT}(\text{DFT}(x_{l-1}) \odot \text{DFT}(h)) * \delta[(k-1) \cdot l] \end{aligned}$$

Let N_s be the length of a smaller convolution involving any slice x_i and h such that $|h| + l - 1 = N_s$ and N_s is a power of two: $N_s = 2^k$ for some $k \in \mathbb{N}$. Then it follows that $l = N_s - |h| + 1$ and $k = |x|/l = |x|/(N_s - |h| + 1)$. Since we perform k convolutions of complexity $N_s \log N_s$ and k additions of size N_s, the total running time of the entire algorithm is $k \cdot N_s \log N_s + k \cdot N_s = k \cdot N_s(\log N_s + 1)$. By substituting k with the equation above and since $|x| = \mathcal{O}(N)$ we obtain:

$$k \cdot N_s(\log N_s + 1) = \mathcal{O}(\frac{N}{(N_s - |h| + 1)} \cdot N_s \log N_s) = \mathcal{O}(\underbrace{\frac{N_s}{(N_s - |h| + 1)}}_{\text{constant}} \cdot N \log N_s) = \mathcal{O}(N \log N_s)$$

Note that based on the lengths $|x|$ and $|h|$ we can pick a value of N_s such that the constant overhead is minimised. A good approximation is $N_s = 8 \cdot 2^{\lceil \log_2 |h| \rceil}$, that is 8 times the smallest power of two bigger than $|h|$. Moreover, all k convolutions require $\text{IDFT}(h)$, which can be computed once and cached for repeated use. This idea is illustrated in Figure 3.1.

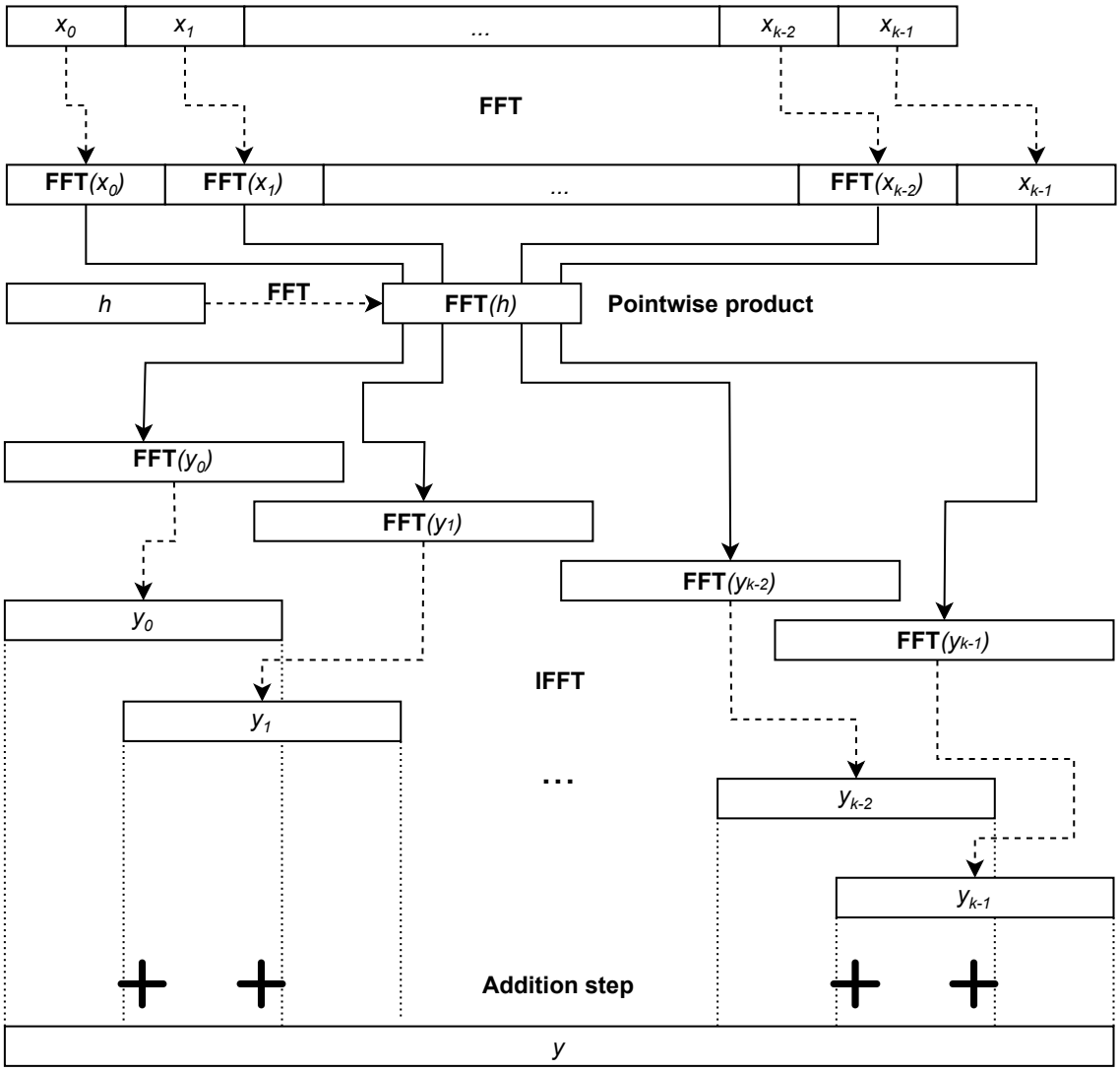

Figure 3.1: The illustration of the Overlap-Add Fast Fourier Transform

3.6 Duplicate convolution

Consider the case where $h[a] = h[b]$ for some $a \neq b$. Then both the a and b-th iterations of the outer loop at row 3 in Input-side Algorithm will compute the same products $h[a] \cdot x[i] = x[b] \cdot x[i]$ for all $0 \leq i < |x|$. In other words, both iterations will scale x by the same value. Although these values need to be added to different samples of the output, they can only be computed once.

This can be achieved by creating an array/list with all samples of h and their corresponding position in the time series. As in the Input-side algorithm, we first need to initialize all samples of y to zero. Following this, we iterate the sorted array and for each element that has a sample value different than the one at the previous position in the sorted array - compute and update some cache vector to contain x scaled by this value. Then for the current and following elements that have the same sample value, we only need to use the stored position in the original time series to compute the position in y from which the cached vector needs to be added. This translates to the pseudo-code given in Algorithm 5, which we will call the *Duplicate convolution* algorithm.

Algorithm 5 Duplicate convolution algorithm

Input: x and h are two integer-valued time series
Output: y is the convolution of x and h

```
1:  function DuplicateConvolve(x, h)
2:      y ← zeroes(|x| + |h| − 1)   ▷ creates a time series of the specified length filled with zeroes
3:      s ← emptyArray(|x| + |h| − 1)   ▷ creates an empty array of the specified size
4:      for k ← 0 to |h| − 1 do
5:          s[k] ← ⟨h[k], k⟩
6:      end for
7:      s ← sortTuples(s)   ▷ sorts an array of tuples based on the first element of each tuple
8:      p ← null   ▷ stores the value of the previous sample
9:      c ← zeroes(|x|)   ▷ the cache variable for scaled x
10:     start ← 0
11:     while first(s[start]) == 0 do
12:         start ← start + 1   ▷ Skip the samples that are zero
13:     end while
14:     for k ← start to |h| − 1 do
15:         if first(s[k]) ≠ p then   ▷ The current sample value is different than the previous
16:             p ← first(s[k])
17:             for i ← 0 to |x| − 1 do
18:                 c[i] ← first(s[k]) · x[i]   ▷ Update cache
19:             end for
20:         end if
21:         j ← second(s[k])
22:         for i ← 0 to |x| − 1 do
23:             y[i + j] ← y[i + j] + c[i]   ▷ Add scaled time series at appropriate position
24:         end for
25:     end for
26:     return y
27: end function
```

The algorithm can be further be optimized by noticing that scaling x by 0 results in a zero time series which does not add any contribution to the convolution regardless of the corresponding shift. This is implemented in the while loop at row 11. Another possible optimization is sorting on the absolute value of the sample values (with positive followed by negative ones) and performing subtraction instead of addition on row 23. This optimization is not reflected in Algorithm 5.

Overall, the resulting algorithms performs at most $|x| \cdot \Upsilon^+$ multiplications, where Υ^+ represents the number of unique non-zero samples in h. In the worst case where all samples in h are distinct and non-zero $\Upsilon^+ = |h|$ and the number of multiplications is $|h| \cdot |x|$, which is the same as the number of multiplications performed by the Input-side algorithm.

Concerning the running time, let $|x|, |h| < N$, $|h^+|$ be the number of non-zero samples in h and $\mathcal{O}(\mathsf{sort})$ be the asymptotic complexity of sorting the samples of h. The former can range from linear $\mathcal{O}(|h|)$ - in case non-comparison sorts can be used - to quadratic - when a simple sort is used. Assuming that an optimal comparison sorting algorithm is used (like Merge Sort), $\mathcal{O}(\mathsf{sort}) = \mathcal{O}(N \log N)$. The first loop performs at most $|h| - |h^+|$ iterations, while the second $|h^+|$. The inner if statements performs $|x|$ operations, same as the following for loop. Therefore, the second loop runs in $\mathcal{O}(|h^+| \cdot |x|)$. Summing everything up, the tightest asymptotic time complexity can be summarized by $\mathcal{O}(\mathsf{sort}) + \mathcal{O}(|h^+| \cdot |x|)$, which under previous assumption, regardless of the chosen sorting algorithm (assuming no algorithm with a time complexity worse than quadratic is chosen), is still equivalent to $\mathcal{O}(N^2)$ and remains sub-optimal. Nonetheless, the described algorithm has the potential to outperform its siblings in the following three scenarios:

1. $|h^+|$ is small, i.e. input time series that have a small number of non-zero elements.
2. the convolution needs to be performed on a machine with a high cost for integer multiplication operation and Υ^+ is small, i.e. inputs that have few unique non-zero samples.
3. The dynamic range of h is (much) smaller than its length.

3.7 Derivative convolution

Let us have a look at the discrete derivative h' of an arbitrary discrete time series h of length $|h| = l$. Informally, the derivative at some point n reflects the change from the previous to the current sample, that is $h'[n] = h[n] - h[n-1]$. Therefore, under our assumption that samples outside the domain are equal to zero, the first sample of the derivative should be equal to the first sample of the input: $h'[0] = h[0] - h[-1] = h[0]$, while the last sample of the derivative must reflect the change from the last sample of h to 0, and therefore is equal to it's negation: $h'[l+1] = h[l+1] - h[l] = -h[l]$, this also being the reason why the length of the derivative is equal to $|h'| = l + 1$. Based on this, it can be observed that the derivative of discrete time series can be expressed as the following convolution: $h' = h * d$, where $d = [1, -1]$.

Conversely, let $\int h$ denote the discrete integral (i.e. prefix sum of data from left to right) of the same arbitrary time series. Then, informally, we would like the integral at a given sample n to reflect the area formed by all samples up to and including n. Since our data is discrete, formally, this can be defined as $\int h[n] = \sum_{i=0}^{n} h[i]$, or recursively: $(\int h)[n] = \sum_{i=0}^{n-1} h[i] + h[n] = \int h[n-1] + h[n]$. Contrary to taking the derivative, the integration can not be expressed as a convolution.

Using the definitions given above, we will now prove that they represent inverse operations.

$$\begin{aligned}\int (h')[n] &= \int (h * [1, -1])[n] = \sum_{i=0}^{n} (h[i] - h[i-1]) = \sum_{i=0}^{n} h[i] - \sum_{i=0}^{n} h[i-1] \\ &= \sum_{i=0}^{n-1} h[i] + h[n] - \sum_{i=-1}^{n-1} h[i] = h[n] + h[-1] = h[n]\end{aligned}$$

Similarly:

$$(\int h)'[n] = (\int h * [1, -1])[n] = (\int h[n] - \int h[n-1]) = h[n] + \int h[n-1] - \int h[n-1] = h[n]$$

Consider the case where we convolve a time series x with the derivative h' of another time series h. Their convolution is then given by $x * h' = x * (h * d)$. Assuming the convolution of x and h is $y = x * h$, since convolution is associative, it then holds that $x * h' = (x * h) * d = y * d = y'$. In other words, this proves that by taking the convolution of x with the derivative h' of another input h, the result of their convolution is equal to the derivative of the convolution of x and h. By using the inverse property proven above: $y = \int (y') = \int (x * h')$. This can be generalized to:

$$y = \underbrace{\int \cdots \int}_{d+t} (x^{(d)} * h^{(t)}) = x * h \tag{3.7.1}$$

Assume two consecutive samples $h[i]$ and $h[i+1]$ for $0 < i < l$ have the probability of being the same $P(h[i] = h[i-1]) = p$. By the definition of derivative, it will then hold that $P(h'[i] = 0) = p$. In other words, around $p \cdot l$ samples in the derivative will be zero samples. With $|l| = 1024$ and $p = 0.9$, we will have around 921 zero samples in h'. Since computing the derivative and integral of a time series following their definition is not computationally expensive, while Duplicate convolution algorithm is efficient for inputs that have many zero samples, this motivates the following algorithm, which computes the convolution using Equation 3.7.1 with $d = 0$:

Algorithm 6 Derivative convolution algorithm

Input: x and h are two integer-valued time series, t is the order of derivation of h
Output: y is the convolution of x and h

```
1: function DerivativeConvolve(x, h, t)
2:     for k ← 1 to t do
3:         h ← Derivative(h)
4:     end for
5:     y ← DuplicateConvolve(x, h)
6:     for k ← 1 to t do
7:         y ← Integral(y)
8:     end for
9:     return y
10: end function
11:
12: function Derivative(x)
13:     x' ← Zeroes(|x| + 1)
14:     l ← |x|
15:     x'[0] ← x[0]
16:     for i ← 1 to l − 1 do
17:         x'[i] ← x[i] − x[i − 1]
18:     end for
19:     x'[l] ← −x[l − 1]
20:     return x'
21: end function
22:
23: function Integral(x)
24:     ∫x ← Zeroes(|x| + 1)
25:     l ← |x|
26:     ∫x[0] ← x[0]
27:     for i ← 1 to l − 1 do
28:         ∫x[i] ← ∫x[i − 1] + x[i]
29:     end for
30:     return ∫x
31: end function
```

For the scenario above, and the information that is given about the samples of the inputs, the best choice for the derivation parameters would be $d = 1$. This will compute the convolution of x and h' using the *Duplicate* algorithm, where the second input will have 90% of the samples equal to zero.

Note that the suggested algorithm only computes the derivative(s) of one of the inputs, when according to Equation 3.7.1 it is possible to do this for both. This is because the *Duplicate* convolution algorithm only benefits from having a big number of zero samples in only one of the inputs, and one only. Assume that we are given two probabilities $P(x^{(d)}[i] = 0) = p_x$ and $P(h^{(t)}[j] = 0) = p_h$ for all $0 \leq i < |x| + d$, $0 \leq j < |h| + t$ and some given $0 \leq d, t$. It follows that the number of non-zero samples in $x^{(d)}$ and $h^{(t)}$ will be given by $|{x^{(d)}}^+| \approx |x| + d - p_x \cdot (|x| + d) = (1 - p_x) \cdot (|x| + d)$ and $|{h^{(t)}}^+| \approx (1 - p_h) \cdot (|h| + t)$. Then to perform the least number of operations (excluding computing the derivative and integral), the algorithm can be invoked with parameters DerivativeConvolve(h, x, d) when $|{x^{(d)}}^+| \cdot |h| < |{h^{(t)}}^+| \cdot |x|$ and with

DerivativeConvolve(x, h, t) otherwise.

In general, the algorithm will perform $t \cdot |h| + 1 + 2 + \cdots + t = t \cdot |h| + \frac{t(t+1)}{2} \approx t \cdot |h|$ operations to compute the derivative and the integral, and $|h^{(t)^+}| \cdot |x|$ operations to compute the *Duplicate* convolution. Obviously, for $N > |h|, |x|$ the algorithm has an asymptotic time complexity of $\mathcal{O}(N^2)$ and remains sub-optimal. Nevertheless, as in the previous case, it has the potential to outperform other algorithms when the derivative of one of the inputs contains many zero samples.

3.8 Longest non-overlapping pattern convolution

Consider the case where for a given time series h, for some $0 \leq i < |h| - 2l$, $i + l \leq j < |h| - l$ and $1 \leq l < |h|/2$ it holds that $h[i + k] = h[j + k] = h_p[k]$ for all $0 \leq k < l$. Put informally, a non-overlapping pattern h_p of length l occurs at positions i and j in h. Then the convolution of h and another arbitrary time series x can be expressed as:

$$\begin{aligned} x * h &= x * (h_0 + h_p * \delta[i] + h_1 * \delta[i + l] + h_p * \delta[j] + h_2 * \delta[j + l]) \\ &= (x * h_0) + (x * h_p) * (\delta[i] + \delta[j]) + (x * h_1) * \delta[i + l] + (x * h_2) * \delta[j + l] \end{aligned}$$

In the formula above we assume that $0 \leq |h_k|$ for $0 \leq k < 3$ and for the special case where $|h_k| = 0$ it holds that $x * h_k = x * [\,] = [\,]$. It can be easily deduced that $|h_0| = i$, $|h_1| = j - i - l$ and $|h_2| = |h| - j - l$.

The new idea is very similar to the previously described Overlap-Add Method, except the convolution is split into k (where $1 \leq k < 6$) smaller convolutions of possibly different lengths, and $k + 1$ shifts and pointwise additions are performed. This is achieved on the grounds that the computation of the convolution of the repeating pattern only needs to be performed once.

Let us now investigate how, given an input h, we can find a recurring non-overlapping pattern. First of all, it should be noted that given a pattern of length k exists in h at positions i and j, it follows trivially that there will also exist patterns of length t, where $1 \leq t < k$ at the same positions. This property enables us to binary search over the length of the longest non-overlapping pattern.

To find whether a non-overlapping pattern of length k exists in h and it's starting positions, an idea similar to the Rabin-Karp algorithm can be used [8]. In particular, using a *rolling hash* function, compute the hash of all sub-sequences of length k in h. When two sub-sequences starting at i and $j > i$ have the same hash and $i + l < j$, assuming the hash function yielded no collision, a pattern of length k occurs at positions i and j. This can be implemented in two ways (both assume that no hash collision occurs):

1. compute and store all hashes in a list. After sorting the list on the value of the hashes and starting positions, take the first and last element in the list having a particular hash value. Assuming their starting positions are i and j, we have the same non-overlapping pattern at i and j when $i + l < j$. If no elements of the list satisfy this condition, no pattern of length k occurs in h.

2. for a given sub-sequence starting at j, compute its hash and insert it in a hash map. When insertion tries to override an occupied hash corresponding to position i, check whether $i + l < j$. If this is the case, we have a non-overlapping pattern at i and j. Otherwise, continue to the next sub-sequence and repeat. If all sub-sequences have been covered, no pattern of length k occurs in h.

The pseudo-code of the resulting algorithm is given below:

Note that in Algorithm 7, the LongestNonOverlappingPattern() function can be implemented using one of the two methods described above, or a potentially better adapted solution to the *longest non-overlapping sub-string* problem, which we could not establish at the moment of

Algorithm 7 Longest non-overlapping pattern convolution algorithm

Input: x and h are two integer-valued time series
Output: y is the convolution of x and h

function LnopConvolve(x, h)
  $l, i, j \leftarrow$ LongestNonOverlappingPattern(h)
  if $l < 1$ **then**
    return Convolve(x, h)
  end if
  $h_0, h_p, h_1, h_p, h_2 \leftarrow$ SplitAt($h, i, i + l, j, j + l$)
  $y_p \leftarrow$ LnopConvolve(x, h_p)
  for $k \leftarrow 0$ **to** 2 **do**
    $y_k \leftarrow$ Convolve(x, h_k)
  end for
  return $y_0 + y_p * \delta[i] + y_1 * \delta[i + l] + y_p * \delta[j] + y_2 * \delta[j + l]$
end function

doing our research. Moreover, the smaller convolutions can be computed using any convolution algorithm, including recursively with LnopConvolve(). In our pseudo-code, we have chosen to do exactly this for the convolution of x and pattern h_p, as this will work nicely for the scenario where a pattern occurs continually, and thus forms a "recursive" pattern.

Let us now look at the time complexity of the algorithm. Finding whether a pattern of length $k < |h|/2$ occurs would require $(|h| - k) < |h|$ times constant time to compute the hashes of all sub-sequences of length k. Using the first method (based on sorting), this will be followed by $\mathcal{O}(|h| \log |h|)$ operations (assuming an optimal comparison sort algorithm is used) and at most $\mathcal{O}(|h|)$ instructions to find whether a pattern occurs and where. The second approach, on the other hand, under the assumption that a good hashing function is used, on average would only require constant time for the insertion of a hash, and thus $\mathcal{O}|(h|)$ operations in total to detect whether a pattern occurs. Therefore, on average, the sorting method will run in $\mathcal{O}(|h| \log |h|)$ and the hash map approach in $\mathcal{O}(|h|)$.

Computing the longest non-overlapping pattern would take $\log |h|$ computations for the binary search of the length of the longest pattern. Assuming that we use the hash map approach this results in $\mathcal{O}(|h| \log |h|)$ for the overall computation of LongestNonOverlappingPattern(h). Splinting h into h_0, h_1, h_2 and h_p takes $|h|$ operations while the smaller convolutions will take at most $\mathcal{O}(N \log N)$ time ($N > |h|, |x|$) when an optimal convolution algorithm is used.

As can be noticed, even with additional assumptions, it is difficult to reduce the time complexity to a practical tight bound. In practice, as patterns occur in h with less noise in between, the smaller the non-pattern sub-sequences h_0, h_1, h_2 are and the faster the smaller convolutions can be performed. Moreover, less noise also implies a higher chance h is formed by exactly two non-overlapping patterns, which will result in less time spent on the binary search and finding the longest such pattern.

Chapter 4

Implementations and time measurements

In this chapter we discuss the concrete implementations of the algorithms described so far, as well as provide concrete measurements in relation to the already provided theoretical time complexities.

We have chosen to implement all algorithms in the `C` programming language because of its relative low level compared to other general purpose programming languages. Manual memory management, pointer (arithmetic) support and static typing of variables are only a few features of the language that, despite making the coding process more complex, reduce the processing overhead and make programs that are written in `C` faster than programs written in many other programming languages. Since speeding up convolution is the main goal of our research, it is only natural that we also use an efficient and powerful programming language to implement the convolution algorithms. This also ensures that our time measurements are more accurate, as we have no variables such as garbage collection, virtual tables or other run-time overhead to account for.

To compile our programs we have used the `gcc` compiler version 10.2 with optimization level `-O2`, as well as the `valgrind` tool to ensure that we have properly managed dynamic memory. A corresponding `Makefile` has been created to invoke the compilation in an easy way.

Each algorithm is implemented in a separate `.c` file. Each file contains a function with the following prototype, where `XXX` represents the name of the convolution algorithm:

```
void XXXConvolution(int lenH, int *h, int lenX, int *x, int *y);
```

The function accepts five parameters: the length of h, a pointer to the samples of the array h, the length of x, a pointer to the samples of the array x and a pointer to an array y where the result of the convolution of x and h will be stored. Unless explicitly specified, all implementations use 32 bit signed integers to store samples of the inputs and of the result of the convolution.

To avoid cluttering this document, only the most relevant functions for each convolution algorithm are provided in the Appendix. The full source code for each implementations is stored and available on the following `GitHub` repository: https://github.com/ghidirimschi/Convolution-Algorithms.

All measurements have been performed on a 'standard' node of the Peregrine HPC cluster of the University of Groningen. The node runs an `Intel(R) Xeon(R) E5-2680 v3` CPU operating at 2.5 GHz on a `CentOS Linux v7` Operating System.

4.1 Measuring and testing framework

All measurements of the running time and testing of algorithms have been done using an extended version of a framework provided by dr. Arnold Meijster, the source code of which can be found in `main.c`. This section will explain how the framework is implemented.

First of all, two arrays, one containing function pointers to the convolution algorithms, and another containing their corresponding ASCII names are hard-coded with the appropriate values. The framework program then parses 2 mandatory and 2 optional command line parameters in the following order: length of x, length of h, probability that two consecutive samples are the same in h, or, in case the fourth argument is supplied - probability that a non-overlapping pattern will repeat throughout h, and the length of the non-overlapping pattern in h. Using a pseudo-random number generator seeded with the current time, the framework then creates and fills arrays x and h with samples from an interval $[l, u]$ (where `MIN_INT` $= -2^{31} < l < u < 2^{31} - 1 =$ `MAX_INT` are hard-coded) that satisfy the supplied parameters. For the majority of convolution algorithms, the choice of l and u are irrelevant for the running time of computing the convolution, as on most modern CPUs all arithmetic operations on 32 bit signed integers take constant time, regardless of their value. Nevertheless, these values need to be chosen carefully when the length of the resulting convolution is big, as to avoid overflows and underflows. As a rough approximation, which follows directly from Def. 1.1, for two inputs x and h and $N = \min(|x|, |h|)$, the minimum and maximum possible values in their resulting convolution will be given by $l^2 N$ and $u^2 N$.

If a hard-coded macro value `CHECK_ALGORITHMS` is set to true, the framework will proceed to test all algorithms. To do so, it computes the convolution of x and h using the Output-Side algorithm and checks that all other algorithms compute the identical result. If any difference is observed, the program halts with an appropriate error message.

If no error occurred, or the value of `CHECK_ALGORITHMS` is set to false, the framework proceeds to measure the running time it takes for each convolution function to compute its output. To provide an accurate result, each algorithm is executed and its running time is measured 50 times, and the minimum such time is outputted. To limit the interference of caching on the measurements, an optional value `LIMIT_CACHING` can be set to true. This will enable the framework to perform several redundant computations before taking each measurement. Although this will hopefully fill the CPU cache with useless values, it will also increase the overall time to perform the measurements.

4.2 Benchmark algorithms

As our running time benchmarks we used the *radix-2* FFT and Input-Side algorithms, as they are generally the faster and most prominent representatives of the classes of optimal and sub-optimal convolution algorithms.

Their implementation has been provided to us by dr. Arnold Meijster together with the framework, and can be found in `fft.c` and `direct.c` (the latter file also contains the implementation of the Output-Side convolution). The code is a verbatim translation of the pseudo-codes provided in Chapter 2 and mostly differs in explicit types, type conversions and some syntactic differences.

A particular important and not yet discussed implementation detail in the radix-2 *FFT* is the memory management of $a^{\mathsf{even}}, a^{\mathsf{odd}}$ and $y^{\mathsf{even}}, y^{\mathsf{odd}}$ arrays. These arrays correspond to the phases of correctly ordering and propagating the samples to the recursive calls, as well as returning intermediate results to previous calls, which are the main bottlenecks of the algorithm. Consequently, various algorithms, such as Pease or Stockham algorithms, or extensions of Cooley-Tukey that require constant additional memory, were developed to make these computations more efficient for various architectures [13].

Our FFT implementation relies on the simple observation that to shuffle a into even and odd-indexed samples at every recursive call we only need an auxiliary (work-space) array wsp of the same length as a, the halves of which will be used as storage for the even and odd-indexed samples.

Once the data is shuffled in *wsp*, its halves play the role of a in the next recursive call, while a can be used as the auxiliary array in the next recursive call. Once the recursive calls are finished, the data in a is still irrelevant, and because the array is not local to the function, it can now be used to compute and return the result of the merge phase, thus yielding an *in-place* implementation. Here, in-place means overwriting the input with output and using at most $\mathcal{O}(N)$ auxiliary memory. Note that this is not the same as a so-called *in-situ* algorithm which uses only O(1) auxiliary memory. Since every recursive call uses a distinct memory region of the auxiliary array as the new main array, this ensures that no relevant data will be overwritten by another call. It therefore suffices to pre-allocate a single auxiliary array of the same size as the original main array before the first call of the function, and then propagate and interchange these array appropriately. This circumvents the need to allocate and free intermediate arrays for $a^{\mathsf{even}}, a^{\mathsf{odd}}$ and $y^{\mathsf{even}}, y^{\mathsf{odd}}$ at every recursive call, which makes our implementation significantly faster.

To illustrate some important properties of these algorithms, we have performed several measurements, of which the results are illustrated in Figure 4.1 and Figure 4.2.

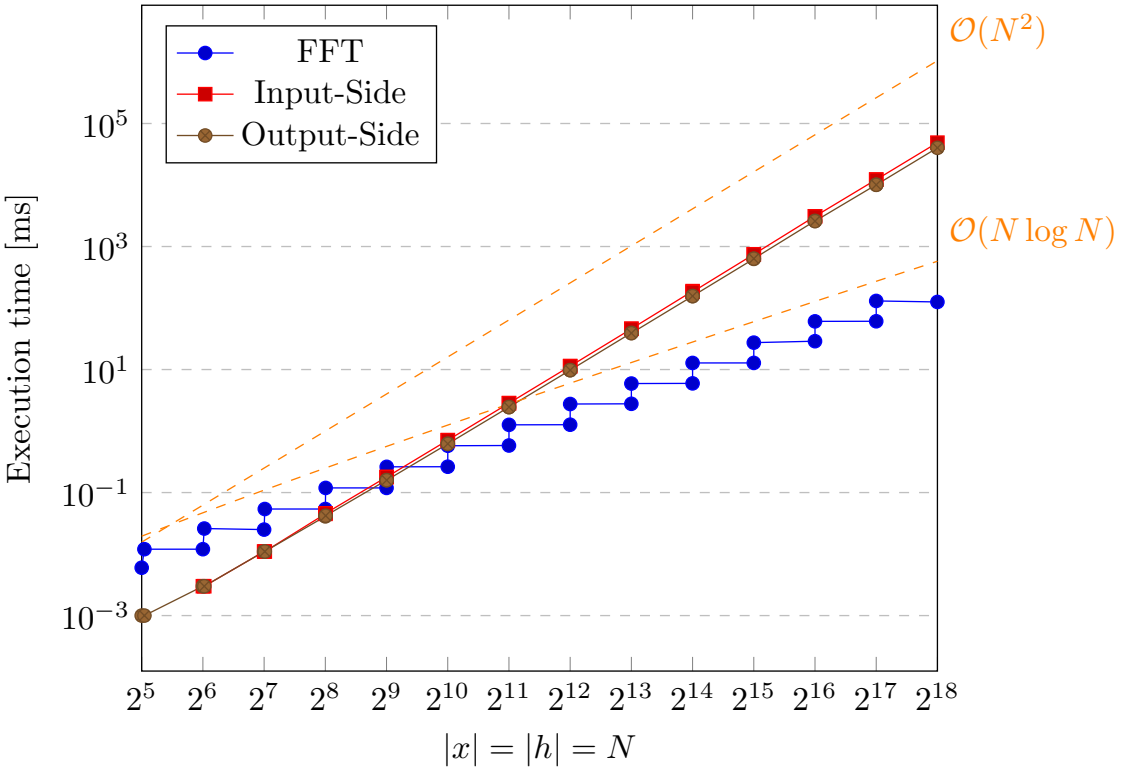

Figure 4.1: Running times at $N = 2^k$ and $N + 1$

In Figure 4.1 we have measured the running time of the FFT, Input-Side and Output-Side convolution algorithms for arbitrary inputs both having length N, where N is either a power of two $N = 2^k, k \in \{5..19\}$, or the successor of such a number (i.e. $N = 2^k + 1$). To better illustrate the data, both axes were taken on a logarithmic scale. The graph illustrates several important properties:

1. On our cluster machine, FFT convolution is generally slower than Input and Output-side convolution for inputs of length smaller than $2^9 - 2^{10}$. Conversely, as the lengths of the inputs go beyond 2^{10}, Input and Output-Side convolutions become slower than FFT convolution.

2. The "stair" pattern observed in the plot of the running time of the FFT convolution reflects the fact that there is negligible difference in the time necessary to compute the convolution of two inputs of length $N_1 = 2^k + 1$ and a convolution of two inputs of length $N_2 = 2^{k+1}$. This is because the lengths of the resulting convolutions in both cases, which are $2N_1 - 1 = 2^{k+1} + 1$ and $2N_2 - 1 = 2^{k+2} - 1$, have the same smallest power of two greater or equal to them, namely

2^{k+2}, and therefore both cases require a nearly equal amount of computations. Conversely, there is significant difference between computing the convolution of two inputs of length $N_1 = 2^k + 1$ and another of two inputs of length $N_0 = 2^k$, as in the former case, the smallest equal or greater power of two is 2^{k+1} which entails an approximately twice as small amount of required computations.

This is not the case in neither Input or Output-Side algorithms, as their amount of computations are independent of the smallest power of two greater or equal than the length of their convolution. Consequently, for these algorithms there is little difference between the running time of $N = 2^k$ and $N + 1$ and therefore the corresponding measurements can hardly be distinguished in the plot.

3. For all values of N, Output-Side is slightly faster than Input-Side algorithm, which contradicts our assumptions in Chapter 2. Some possible causes are that Output-Side algorithm is more suitable for the particular architecture of the machine on which the measurements are made or that the used compiler is generating slightly more efficient code for Output-Side convolution. Indeed, our tests confirm that measuring these implementations on different machines or using binaries generated by different compilers can make Input-Side algorithm faster.

4. The two additional orange dashed plots represent a quadratic N^2 and another $N \log N$ function, both scaled by some constant. As can be noticed, the plots of the Input and Output-side algorithms execution times intersect the logarithmic and are parallel to and always below that of the quadratic function. Conversely, the plot corresponding to the measurements of the FFT algorithm is always below both functions, and the growth in the "stair" pattern is equivalent to that observed in the lower dashed plot. This shows that the first two algorithms indeed belong to the class of sub-optimal algorithms - $\mathcal{O}(N^2)$, while the latter is an optimal $\mathcal{O}(N \log N)$ algorithm.

Figure 4.2 on the other hand illustrates time measurements for a fixed length of x (being $|x| = 2^{15}$) and a varying length of h (being $|h| = 2^k$ for $k \in \{5..18\}$). These measurements clearly indicate that the running time of the FFT convolution algorithm is highly correlated with the length of the longest input, as the running time stays constant when $|h| < |x|$ and starts growing otherwise. This is not the case for the direct convolution algorithms, which are clearly faster when length of $|h|$ is small, and take linearly more time as $|h|$ increase.

4.3 Karatsuba and Toom-Cook algorithms

The implementation of the adapted version of the Karatsuba algorithm can be found in `karatsuba.c`. It is a translation of pseudo-code provided in Algorithm 3, with a minor modification for computing z_1. In particular, as mentioned in Section 2.3, z_1 can also be expressed as $z_1 = (x_0 + x_1) * (h_0 + h_1) - z_2 - z_0$. This only requires swapping point-wise additions and subtractions in several places of the algorithm. We have chosen this approach as it allows to easily compute $x_0 + x_1$ and $h_0 + h_1$ directly in arrays x and h.

Similar to the FFT convolution algorithm, of particular interest in the implementation is the memory management of the intermediate computations, namely z_0, z_1 and z_2. First of all, using $2\lfloor x/2 \rfloor \leq x$ for all $x \in \mathbb{N}$, we note that $|z_0| = |h| - \lfloor |h|/2 \rfloor + |x| - \lfloor |h|/2 \rfloor - 1 = |h| + |x| - 1 - 2\lfloor |h|/2 \rfloor \leq |x| - 1$ and $|z_2| = 2\lfloor |h|/2 \rfloor - 1 \leq |h| - 1$. Since $|y| = |x| + |h| - 1 > |x| - 1 + |h| - 1 \geq |z_0| + |z_2|$, the results of intermediate convolutions z_0 and z_2 can be computed and stored directly in array y, at positions corresponding to the shifts specified by the sum in the return statement. Since no samples overlap, their values can also be retrieved for future computations. Unfortunately, we could not determine a way to manipulate the available input and output arrays to also compute z_1 without additional memory, where $|z_1| = |z_0|$. Although implementations requiring only additional space linear in the lengths of the inputs exist [14], we are sceptical whether the saved memory and the added complexity will result in significant (if any) running time improvements. For this reason we decided to resort to a simpler implementation that requires an auxiliary array of size $|z_1| = |h| + |x| - 1 - 2\lfloor |h|/2 \rfloor \leq |h| + |x| - 1 - (|h| - 1) = |x|$ at every recursive call.

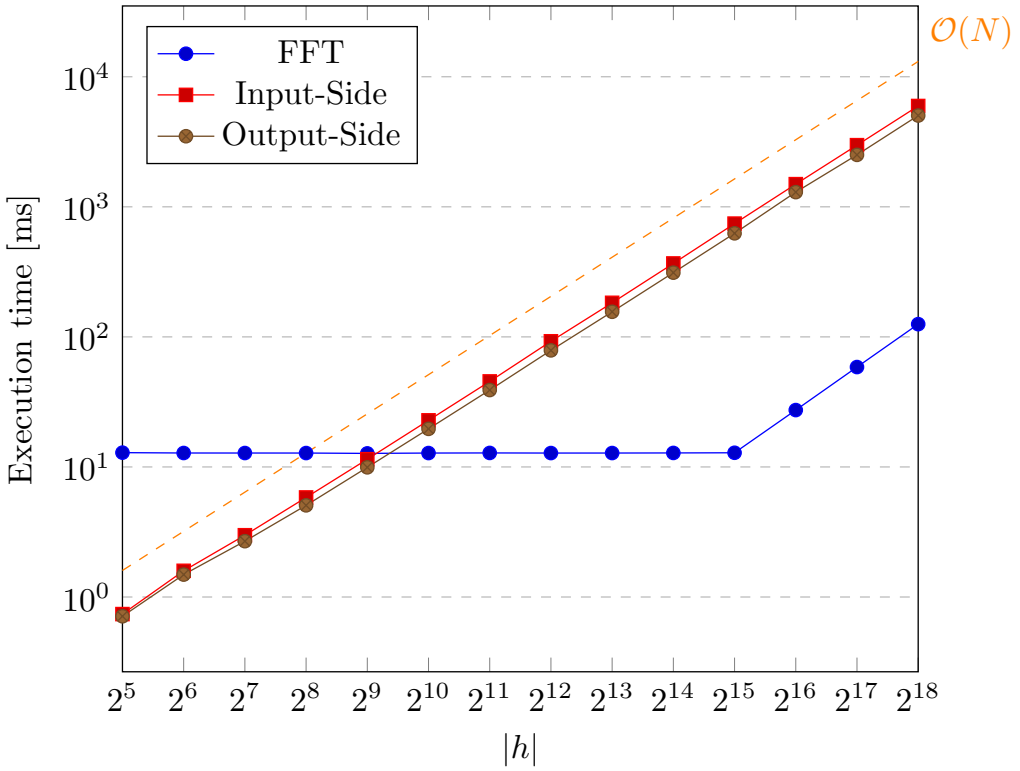

Figure 4.2: Running times at $|h| = 2^k$ for $k \in \{5..18\}$ and fixed $|x| = 2^{15}$

The most straightforward way to implement our idea would be to allocate and release the necessary memory at every recursive call, but this would result in memory fragmentation and will undoubtedly slow down the performance of our implementation. For this reason, we instead pre-allocate a sufficiently big work-space array which is propagated throughout the recursion, similarly to the idea used in the implementation of the FFT algorithm. The size can be easily computed by using the above fact that $|z_1| \leq |x|$ and that the depth of the recursion is smaller than $\lceil \log_2 |h| \rceil + 1$ (since every level halves h). It is therefore sufficient to use a work-space array of size $|x| \cdot (\lceil \log_2 |h| \rceil + 1) = \mathcal{O}(N \log N)$ assuming that $|h|, |x| < N$. As opposed to the FFT, there is no swapping of the arrays - a portion of the work-space memory is used to compute and store the smaller convolution of $(x_0 + x_1) * (h_0 + h_1)$, while the rest is used as the work-space in the recursive calls. The same memory storing the intermediate convolution can then be used to compute z_1, by point-wise subtracting from it z_0 and z_2 (available in the first and second halves of y). This result can then be point-wise summed over y at position $\lfloor n/2 \rfloor$, thus yielding the correct result of the convolution of x and h.

Figure 4.3 shows our measurements for arbitrary inputs, where both x and h have the same length (being $|x| = |h| = 2^k$ for $k \in \{5..17\}$). The growth of all three plots show that the proven asymptotic time complexity of the Karatsuba algorithm is indeed smaller than that of the Input-Side algorithm and bigger than that of the FFT convolution algorithm. The orange dashed plot, corresponding to $N^{1.58}$ function scaled by a constant factor, is indeed above and parallel to the plot of our measurements for the Karatsuba algorithm, which indicates that the algorithm belongs to $\mathcal{O}(N^{1.58})$ class. Since $N = 2^5$ is a small number, the measurements at this point mostly reflect the constant factor, which is unsurprisingly the biggest for the FFT and smallest for the Input-Side convolution. Overall, the adapted Karatsuba convolution algorithm is in a rather unfortunate position: it is faster than the FFT when it is slower than the Input-Side convolution, and it becomes slower than the FFT when it surpasses the Input-Side algorithm. This renders the algorithm impractical for computing convolutions on the used target architecture, as it is surpassed by one of the benchmark algorithms for all input lengths.

Figure 4.3 does not reflect an important property of the Karatsuba convolution algorithm: the smaller the difference in lengths of the inputs - the smaller is the constant factor. This is because

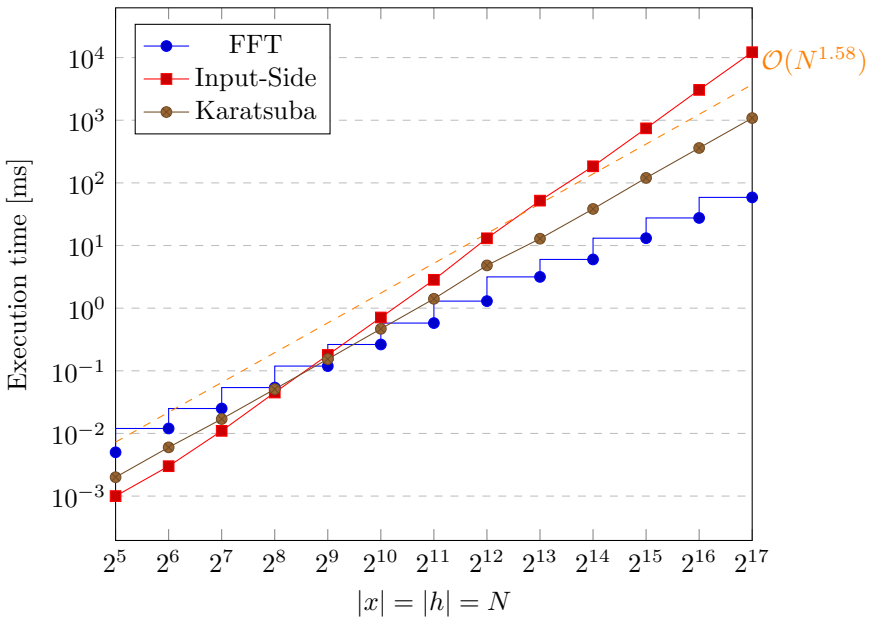

Figure 4.3: Running times at $|h| = |x| = N = 2^k$ for $k \in \{5..17\}$

at the last level of recursion, in the worst case, we will need to scale an array of size $|x| - 1/2|h| - 1/4|h| - \cdots - \frac{1}{\lceil \lg_2 |h| \rceil}|h| < |x| - |h| + 1$, which will tend to 1 as $|h|$ will tend to $|x|$. This implies that in fact, Figure 4.3 only illustrates the best case scenario where the constant factor is minimal for Karatsuba convolution. To show this property, we fixed the length of x at $|x| = 2^{15}$ and measured the running time of the algorithms for lengths of h that were smaller powers of two, or were close to the length of x. The results are illustrated in Figure 4.4.

Given practical considerations, we did not implement an adapted Toom-k convolution algorithm for any $k > 2$. Although the asymptotic time complexity function $N^{log_k 2k-1}$ outgrows the optimal $N \log N$ at bigger and bigger values of N as k increases, the constant factor also grows significantly. To solve this discrepancy, possible optimizations resulting in the reduction of the constant factor in Toom-Cook k algorithms need to be investigated and implemented. Unfortunately, this is outside the scope of this research.

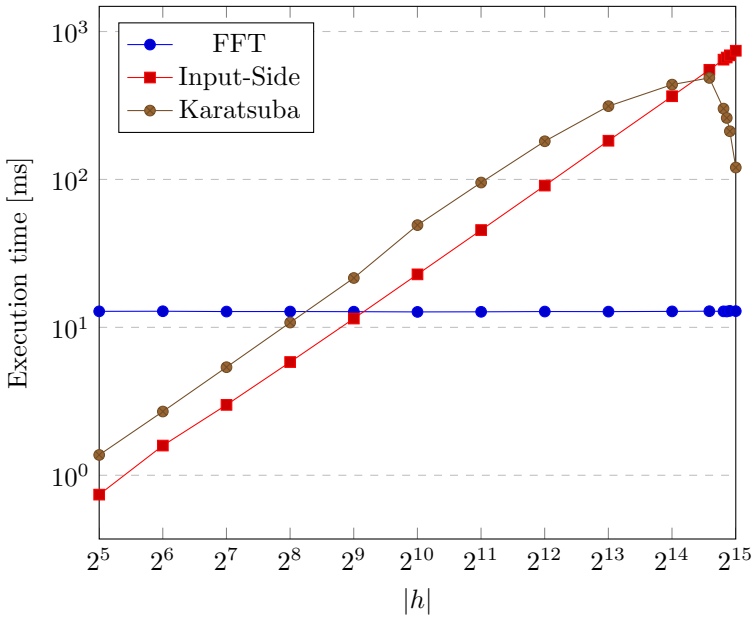

Figure 4.4: Running times at $|h| < |x| = 2^{15}$

4.4 NTT

In Section 3 we discussed a practical NTT using a big modulus $M = 9223372036737335297$, the Barrett reduction as a potential optimization, and the use of the Chinese Remainder Theorem as a way to reconstruct the convolution containing samples bigger than the moduli of two or more NTT convolutions. The concrete implementations of these algorithms/techniques can be found in the following files:

1. `nttsoftware.c` contains the implementation of the practical NTT discussed in Section 3.1. The code is an adaptation of the *radix-2* FFT implementation, using the same memory management approach described earlier. To accommodate for the 64 bit modulus, `double complex` are replaced by 64 bit typed integers, and the modular addition and multiplication operations are implemented trivially. In particular, for modular multiplication, an 128 bit integer is used to store the result of the intermediate multiplication and then its remainder under the modulo is computed using the built-in modulo operator. Since the modulo of a 128 bit integer can not be possibly computed directly on a mainstream CPU, this operation is emulated in software by the compiler, hence the name of the file. The N-th primitive roots of unity, their inverses and inverses of N, are pre-computed and stored in look-up arrays for all $N = 2^k$ for $k \in \{0..24\}$.

2. `nttbarrett.c` extends the code of `nttsoftware.c` by replacing the software modular multiplication by the Barrett reduction technique and bitwise manipulations described in Section 3.2.

3. `nttcrt.c` implements a convolution modulo $M = M_0 \cdot M_1 = 65537 \cdot 114689 = 6369482993$ by performing two NTT convolutions with moduli M_0 and M_1 and reconstructing all samples using the formula given by the constructive proof of the Chinese Remainder Theorem in Section 3.3. To further speed this computation, P and the products $\mathcal{I}_i P_i$ are pre-computed for all $0 \leq i < 2$. Note that since M_1 requires 17 bits to be stored, the multiplication must use a 64 bit intermediate variable because $2 \cdot 17 = 34$ bits do not fit in a standard 32 bits integer.

4. `nttcrtbarrett.c` extends the code of `nttcrt.c` by replacing the modular multiplication operations with Barrett reductions in both convolutions.

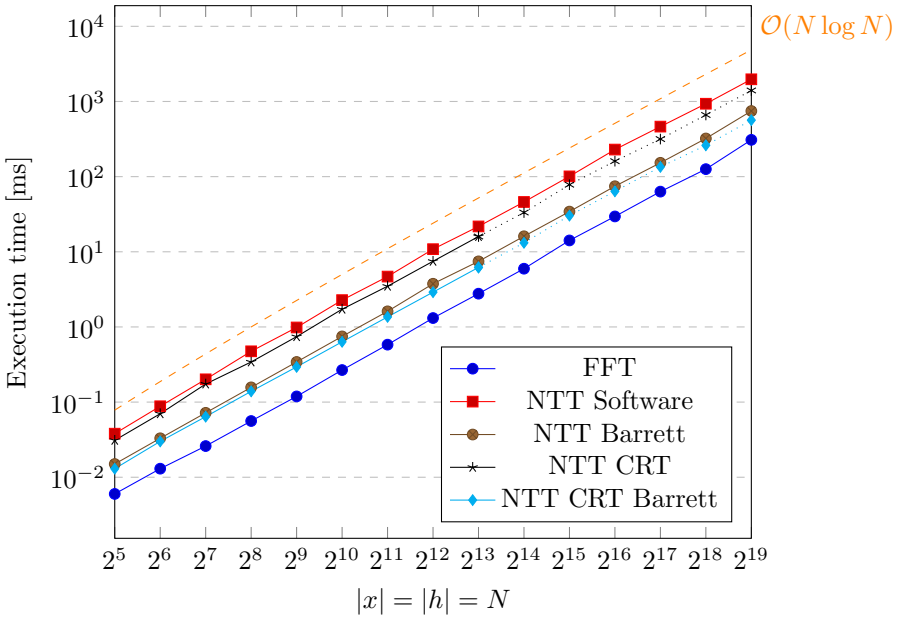

Figure 4.5: Running times at $N = 2^k$ for $k \in \{5..19\}$

Figure 4.5 illustrates the measurements of the running times of the FFT and the various NTT convolution algorithms for inputs of equal size (being $|x| = |h| = N = 2^k$ for $k \in \{5..19\}$). We deliberately omit measurements at successors of these powers of two, as to not congest the figure with overlapping "stair" plots. Note that in reality, all these running times would exhibit a "stair" pattern, as all algorithms pad their inputs to the smallest power of two equal to or greater than the sum of their lengths. Overall, the illustration leads to the following observations:

1. All plots are parallel and below the orange dashed line, which represents the $N \log N$ function scaled by a constant. This indicates that all implementations have an optimal running time $\mathcal{O}(N \log N)$. The plots corresponding to the NTT algorithms are parallel and above the blue line representing the FFT algorithm, which indicates that all NTT implementations are slower, and thus have a bigger constant factor.
2. On the target architecture, computing 2 NTT convolutions with smaller moduli and reconstructing the result using CRT (which requires 64 bits at most to reconstruct samples using the CRT) is slightly faster than computing the NTT using a big modulus (requiring 128 bits for storing intermediate data).
3. The Barrett reduction technique reduces the running time of both the NTT convolution using a big modulus, and the CRT algorithm.
4. Both variations of the NTT CRT algorithms begin to yield incorrect results after lengths of the inputs $N = 2^{13}$ (reflected by dotted lines in both plots). This corresponds to the fact that there does not exist an 2^{13}-th primitive root of unity modulo $M_1 = 114689$ and therefore the corresponding NTT based algorithms fail for these lengths.

4.5 Overlap-Add FFT

The implementation of the Overlap-Add FFT (OAM FFT) convolution algorithm can be found in `oamfft.c`. For the most part, it is a direct translation of the description in Section 3.5: compute N_s using the given approximation, compute once the FFT of h padded to this length, point-wise multiply it with the computed FFT of each slice of x padded to the same length, and finally point-wise sum the IFFT of the result at the appropriate positions in array y containing the convolution. There is only one extension, and that is an additional convolution to account for the situation where x can not be split into slices of length $N_s - (|x| - 1)$ exactly. Then the remaining samples can form an incomplete slice for which another convolution is computed and point-wise summed at the appropriate location in y.

Figure 4.6 illustrates the running times of the Overlap-Add FFT, the FFT and Input-Side convolution algorithms when the lengths of the inputs are both equal to powers of two or their successors. The results show a consistent and constant worse performance of the OAM FFT compared to standard FFT, which is not surprising given that inputs of equal lengths cause the algorithm to degenerate to a simple FFT of inputs of length 8 times as bigger. Despite the Figure picturing the worst case scenario for the OAM FFT algorithm, it confirms that it belongs to $\mathcal{O}(N \log N)$.

Figure 4.7 shows the running times of the OAM and simple FFT algorithms for a fixed length of x at $|x| = 2^{19}$. It depicts the fact that the smaller the length $|h|$ is in relation to the length $|x|$, the faster the algorithm computes the convolution. For $|x| = 2^{19}$, the OAM FFT algorithm runs faster than the standard FFT for all $|h| < 2^{17} = |x|/4$.

Conversely, Figure 4.8 displays the running time measurements for a fixed length $|h|$. Similar to the previous figure, it demonstrates that when x is not at least 4 times as long, the standard FFT runs faster than the OAM FFT algorithm, as well as that as x becomes longer in relation to h, OAM FFT runs faster in relation to the standard FFT. Another property that can be observed in the figure is the linear growth of the running time of OAM FFT as $|h|$ is fixed and $|x|$ increases. This follows from the orange dashed plots that correspond to a linear and another $N \log N$ function, both scaled by some constant. In particular, the OAM FFT measurements form a plot that is parallel and below the linear function, which supports the previously proven tightest

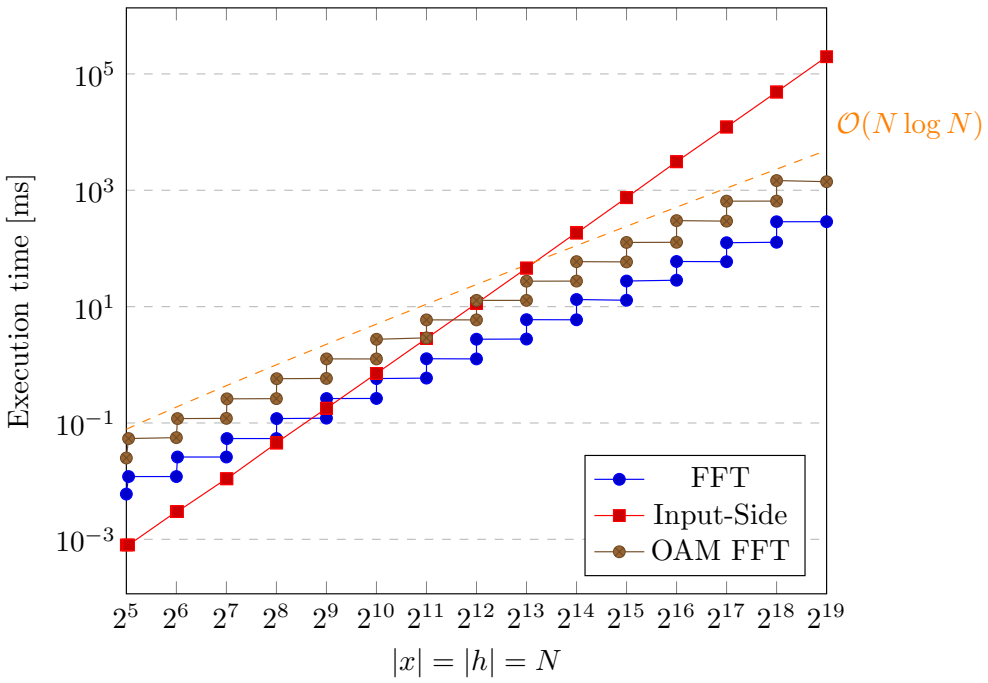

Figure 4.6: Running times at $N = 2^k$ and $N + 1$ for $k \in \{5..19\}$

asymptotic time complexity for the OAM FFT of $\mathcal{O}(N \log N_s)$ which equals $\mathcal{O}(N \log |h|)$ under the approximation $N_s = 8 \cdot 2^{\lceil \log_2 |h| \rceil}$, and therefore $\mathcal{O}(N)$ when $|h|$ is fixed. On the other hand, even with fixed $|h|$, standard FFT intersects the linear plot and is only below and parallel the $N \log N$ function. This exemplifies the fact that standard FFT asymptotic time complexity only depends on the length of the longest input and therefore does not change when one of the inputs is fixed.

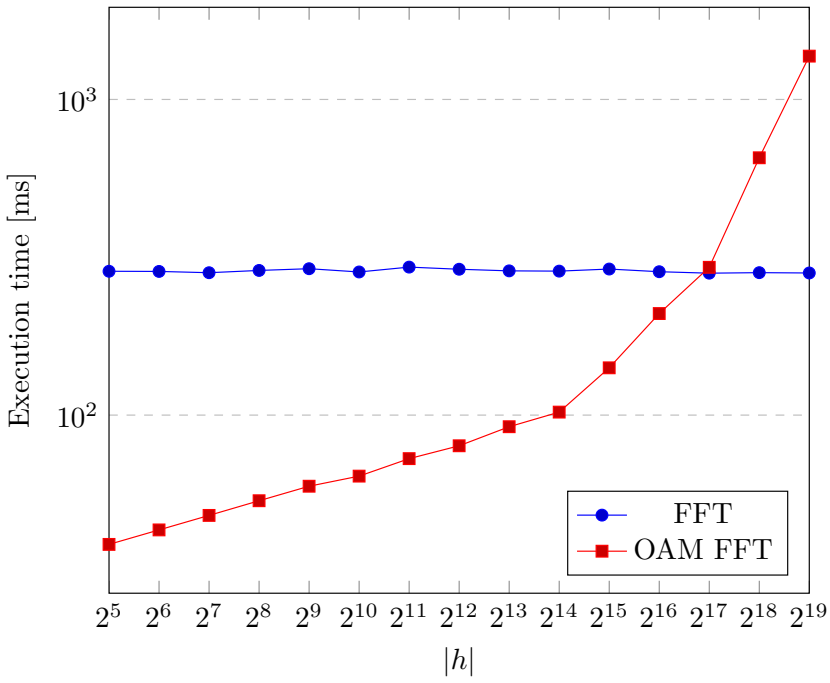

Figure 4.7: Running times at $2^5 \leq |h| = 2^k < |x| = 2^{19}$ for $k \in \mathbb{N}$

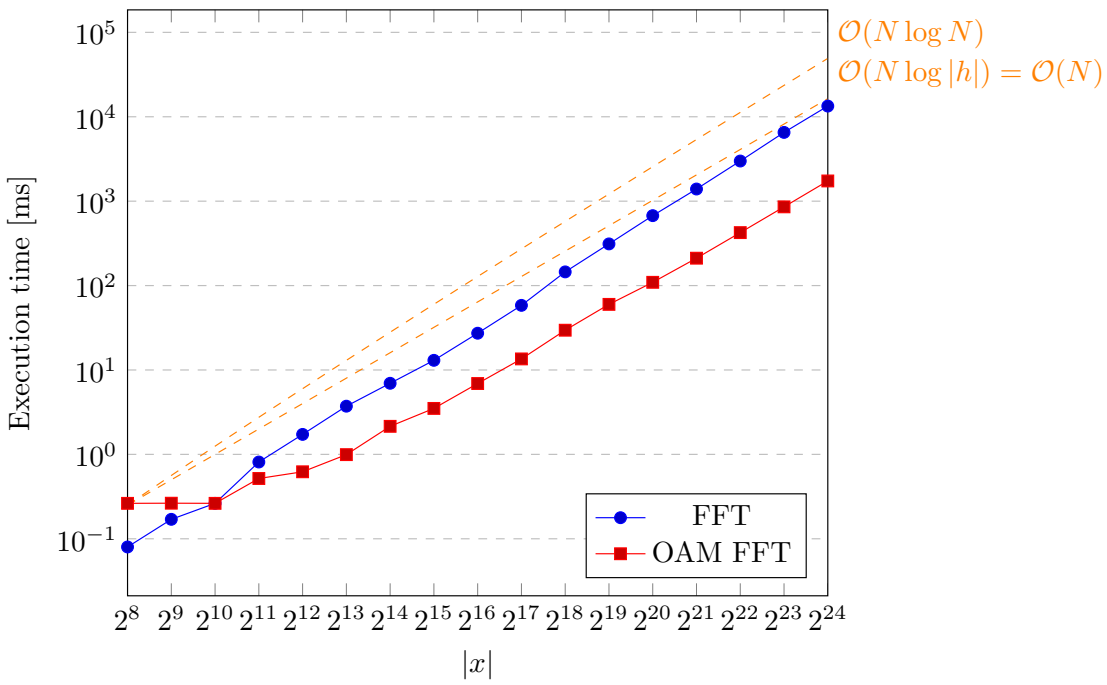

Figure 4.8: Running times at $|h| = 2^8 \leq |x| = 2^k < 2^{19}$ for $k \in \mathbb{N}$

4.6 Data-driven algorithms

So far we have discussed algorithms of which the performance depends exclusively on the lengths of the inputs. This means that these algorithms will take an equal amount of time to perform convolutions that are either hard or trivial to compute manually (e.g. $x * \delta[k]$, for some large k and $|x|$, which is just shifting x by k samples). In the second half of Chapter 2 we have introduced three convolution algorithms that exploit the properties of the inputs to potentially speed-up the computations at the cost of some overhead. The implementations of these algorithms can be found in the following files:

1. `duplicate.c` contains the implementation of the *duplicate convolution* algorithm. The algorithm is a verbatim translation of the pseudo-code provided in Section 3.6 except the samples of h are sorted on their absolute value and the same cache is used for both a value and its opposite. For sorting, the built-in `qsort` function is used when the dynamic range is big, and a hand-written histogram sort otherwise.

2. `derivative.c` implements the first derivative case of the *derivative convolution* algorithm ($t = 1$).

3. `lrnop.c` contains the implementation of the *longest non-overlapping pattern convolution* algorithm (LRNOP) using the hash-map approach for finding the pattern. For smaller convolutions, it uses the Input-Side algorithm when the length of one of the inputs is shorter than 1000, and the FFT convolution otherwise.

Figure 4.9 displays the running time measurements for benchmark and data-driven algorithms for arbitrary inputs of equal power of two lengths, where the samples in h range from 0 to 100. This leads to the following observations:

1. For $N > 2^9$, the duplicate and derivative algorithms are faster than the Input-Side convolution. This shows that for a small dynamic range of h and a sufficiently big length $|h|$, the overhead for sorting h and reusing a cache is smaller than recomputing multiplications. Although this makes the Duplicate and Derivative algorithms faster than Input-Side algorithm by a constant factor, the algorithms are still slower than FFT convolution for $N > 2^8$.

2. The duplicate algorithm is only by a negligible factor faster than the derivative algorithm. This shows that derivation and integration add very little overhead.

3. The derivative and duplicate algorithms are clearly sub-optimal. The plot corresponding to LRNOP intersects the plot of the scaled $N \log N$ function, but has a smaller growth rate than the quadratic function, which implies that it has a tighter asymptotic time complexity than $\mathcal{O}(N^2)$.

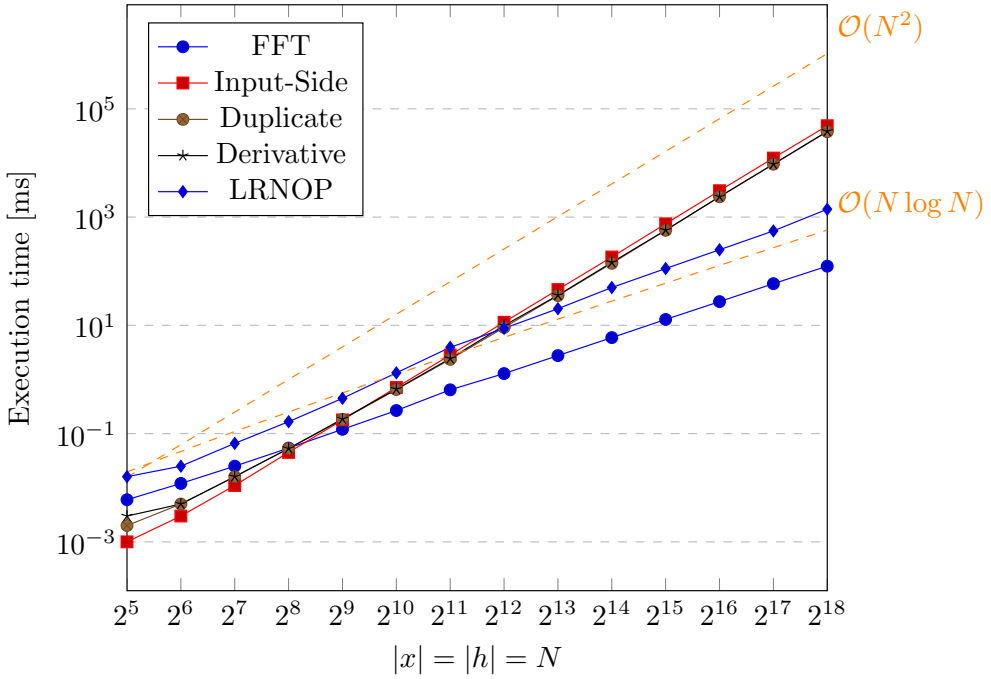

Figure 4.9: Running times at $N = 2^k$

Figure 4.10 illustrates how the performance of the *derivative* algorithm for lengths of inputs $|x| = |h| = 2^{13}$ increases as the probability that two consecutive samples in h are equal increases too. Unfortunately, for these lengths, the algorithm is only able to outrun FFT convolution for $p = P(h[i] = h[i-1]) > 0.9$. In general, as the lengths of the inputs increase, p must increase too for the algorithm to outrun FFT. More precisely, we need to limit the number of non-zero samples in $|h'^{+}| = (1-p) \cdot |h|$ to some constant c such that the algorithm has a linear instead of a quadratic growth in relation to $N > |x|, |h|$. For this we need that $1 - p < c/|h|$ and therefore $p > 1 - c/|h|$, which will approach 1 as $|h|$ grows. As a concrete example, to generate measurements on our machine where *duplicate* algorithm outrun FFT for inputs of lenght $|x| = |h| = 2^{15}$, we needed a $p \geq 0.99$. Nonetheless, this proves that for inputs where p is known to be high, the *duplicate* algorithm has the potential to outrun FFT.

Lastly, we have tried to cherry pick scenarios where LRNOP algorithm would be faster than FFT, OAM FFT and Input-Side algorithm. One of the examples we managed to find is when a pattern of length $|p| = 2^k$ occurred throughout $|h| = 2^4|p|$ without any break (noise), and when $|x| = 2^9|p|$, for $k \in \{0..8\}$. The running time measurements are illustrated in Figure 4.11. Such a configuration enables few invocations of the Rabin-Karp search of the longest non-overlapping pattern due to a shallow depth of the recursion and the pattern occurring at exactly half of the input at each level of recursion except the last (which causes binary search to return a result at the first iteration). This indicates that in general the overhead of searching repeating patterns using Rabin-Karp algorithms outweighs the potential benefit of computing the convolution of a pattern only once. A particular bottleneck in our implementation are hash collisions, which might be avoided by using a better (rolling) hash function. Similarly, a better solution to the *longest repeating non-overlapping substring* problem (e.g. running in linear time) could drastically diminish the overhead and therefore improve the performance of the LRNOP convolution. Unfortunately, we could not identify such an algorithm at the time of our research.

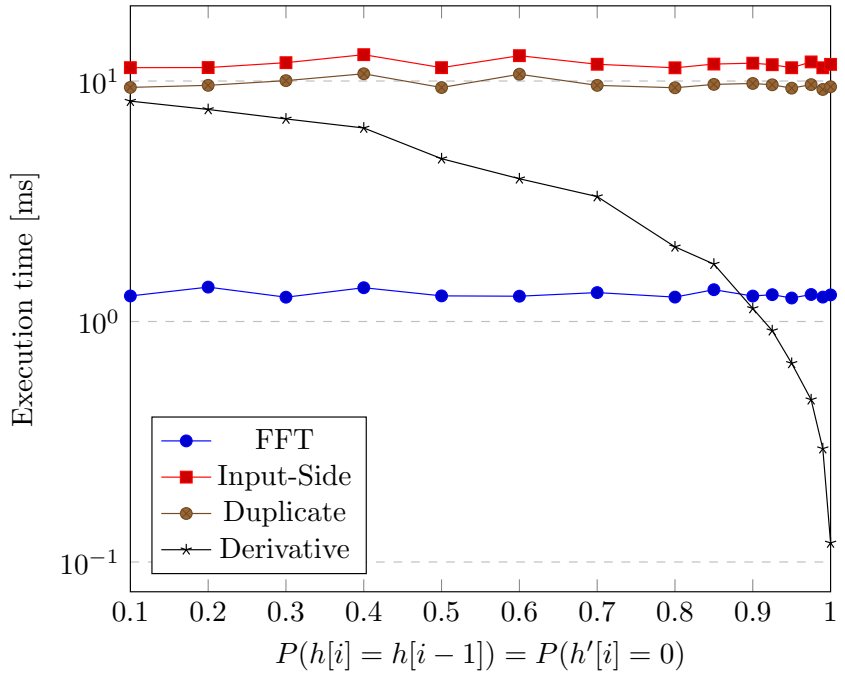

Figure 4.10: Running times at $|x| = |h| = 2^{13}$

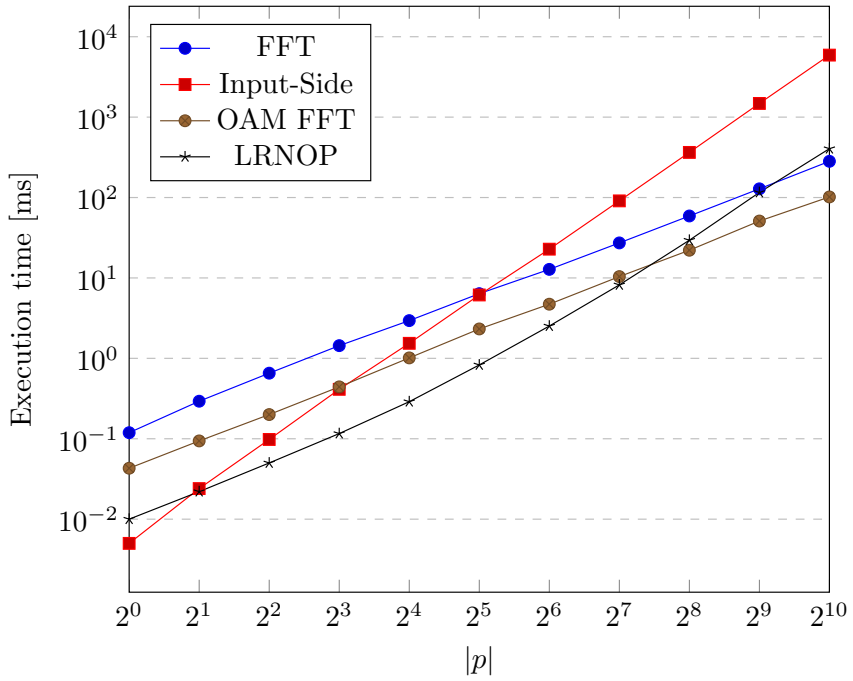

Figure 4.11: Running times at $|p| = 2^k$, $|h| = 2^4|p|$ and $|x| = 2^9|p|$ for $k \in \{0..10\}$

Chapter 5

Conclusions

In this research project we have discussed several convolution algorithms and provided implementations for them. These have been extensively tested using our framework, which makes us confident that these implementations and the underlying theory are correct. This includes the direct, Karatsuba and FFT algorithms, which we have analyzed in Chapter 2, as well as the NTT with big modulus, Overlap-Add FFT and novel *duplicate*, *derivative* and *longest repeating non-overlapping pattern* convolution algorithms that are discussed in Chapter 3. Separate NTT implementations using (combinations of) the Barrett reduction technique and the Chinese Remainder Theorem (presented in the same chapter) have also been tested. The results support their correctness, as well as their viability to improve the running time of a standard NTT using a straightforward modular multiplication. We have conducted tests with various configurations of the inputs, during which running times of all algorithms have been measured and stored. We consider that the conducted research and gathered data is sufficient to give a conclusive answer to our research question stated in Section 1.2.

Summing everything up, the FFT convolution is the algorithm that has both an optimal asymptotic time complexity and has the fastest practical running time for arbitrary integer time series of arbitrary lengths on the target machine that we used for this research. Unfortunately, as the length of and the maximum value in the inputs grow, the precision of the FFT convolution starts to decrease due to the use of floating point arithmetic and errors might appear in the result. The convolution algorithm based on the NTT circumvents this problem by using integer arithmetic. Although it also has an optimal time complexity, our measurements show that it has a bigger overhead than FFT based convolutions. This can be reduced by using the Barrett reduction technique and the CRT to reconstruct the result from NTT convolutions that use smaller data types, but on the architecture that we used these optimizations were insufficient to surpass FFT based algorithms. Moreover, in the case of NTT based algorithms, primitive (principal) roots of unity can not be computed in a trivial way and generally require some preliminary calculations.

We also investigated the Overlap-Add FFT convolution algorithm which, in our tests, has managed to outrun the standard FFT when the length of one input was at least four times bigger than the length of the other, but was slower otherwise. This makes it a good candidate for computing convolution where it is known in advance that one input is sufficiently shorter than the other. This is typically the situation where one of the two inputs are the filter coefficients of a Finite Impulse Response (FIR) filter.

In general, the direct convolution algorithms were slower than the optimal ones. Nevertheless, our measurements show that when the lengths of the inputs are smaller than some threshold ($2^9 - 2^{10}$ on our machine) they are faster and therefore viable options. Although we have proven a slightly tighter asymptotic bound for the *input-side algorithm*, our tests have shown that depending on the architecture of the machine and the used compiler, the *output-side* algorithm may perform slightly better.

We have also examined whether convolutions can be computed more efficiently for specific inputs

and have reached positive results. In particular, the *duplicate* convolution algorithm was able to achieve a minor speed-up for inputs with a small dynamic range compared to the input-side convolution algorithm. Despite the fact that this speedup is insignificant for practical purposes, the algorithm serves as the foundation for the *derivative* convolution, which was demonstrated to excel in terms of execution time when there is a high probability that two consecutive samples are the same in one of the inputs. Another data-driven algorithm focused on optimizing convolutions with recurrent patterns was devised and tested. Although it is able to outrun other algorithms for some very specific configurations of the inputs, its application for practical convolutions is currently limited by the rather inefficient solution to finding the *longest repeating non-overlapping pattern*. A faster way to identify the recurring patterns could enable the algorithm to become a better candidate for computing convolutions.

Lastly, we have also investigated the application of the Karatsuba algorithm and its generalization - Toom-Cook, originally devised for computing multiplications, in the context of convolution. We have shown theoretically, and in the case of Karatsuba also practically, that the same algorithms can be adapted to calculate the convolution with the same asymptotic time complexity as in the case of multiplication. Unfortunately, in practice, our implementation of Karatsuba was unable to achieve any advantage over other algorithms due to a high overhead. Nonetheless, similar to the LRNOP convolution, Karatsuba and Toom-k algorithms have the potential to become viable candidates for computing convolutions if methods that decrease their overhead are identified and implemented.

In conclusion, there does not exist a universally best convolution algorithm for integer data types that would perform best in all possible scenarios. We have shown that multiple factors can influence one convolution algorithm to perform better than the others in terms of running time. These factors include the underlying architecture of the machine carrying out the computations, the used compiler and its optimizations, (relative) lengths of the inputs and constraints on their data such as dynamic range, equal consecutive samples or non-overlapping patterns. Moreover, limitations of each algorithms need to be juxtaposed with the possible inputs to ensure that the correct result of the convolution can be returned (e.g. precision errors in the FFT, maximum lengths of the inputs in the NTT etc.). For this reason, we consider that in the case of systems that need to compute time-critical and infallible convolutions, all these elements need to be taken into consideration to decide the best convolution algorithm. To simplify this decision process, a tool like our framework can be used to compare the efficiency and reliability of convolution algorithms on the particular system.

Appendix A

NTT lookup table

N	$N^{-1} \pmod M$	$\omega \pmod M$	$\omega^{-1} \pmod M$
2^0	1	1	1
2^1	4611686018368667649	9223372036737335296	9223372036737335296
2^2	6917529027553001473	5821608562908302673	3401763473594151602
2^3	8070450532145168385	6183202344127710422	4637966906727796075
2^4	8646911284441251841	8232954128065750752	5263173472767262291
2^5	8935141660589293569	2506374561187543290	2022096071146522440
2^6	9079256848663314433	5175679149588297482	2345448648911858112
2^7	9151314442700324865	4529680260184893260	7628857276699979704
2^8	9187343239718830081	4693975314133565746	7164255507103637110
2^9	9205357638228082689	250208342857496214	8429686828039810538
2^{10}	9214364837482708993	4469250536659403476	7714104868670012898
2^{11}	9218868437110022145	1890336861624766990	3124841710688946557
2^{12}	9221120236923678721	935815637929945223	4634860353951294624
2^{13}	9222246136830507009	1236403532507684466	2872752945965363591
2^{14}	9222809086783921153	2949988598460004147	2751275275295523387
2^{15}	9223090561760628225	3207215361743254657	5067346905827978431
2^{16}	9223231299248981761	3793264300966149197	514542188249434779
2^{17}	9223301667993158529	5841790927866575890	7348949947088264256
2^{18}	9223336852365246913	2514491450482062217	1108796241822858026
2^{19}	9223354444551291105	3993053986736158084	7387127451862881604
2^{20}	9223363240644313201	6123203078656286151	3990408202089798765
2^{21}	9223367638690824249	7818225862990107593	7306313761062697635
2^{22}	9223369837714079773	4224198908362084779	3276901088803885951
2^{23}	9223370937225707535	4556310295883022695	6050139748279733103
2^{24}	9223371486981521416	2419180138865645092	6543991442235915766

Table A.1: The look-up table for inverses N^{-1} and N-th primitive roots of unity for powers of two up to and including 2^{24} (modulo $M = 9223372036737335297$)

The values in the table were computed on a local machine using simple modular inverse and exponentiation programs.

Appendix B

64 bit Barrett reduction

```
int64_t multiplyHighUnsigned(int64_t x, int64_t y) {
    int64_t x_high = (uint64_t) x >> 32;
    int64_t y_high = (uint64_t) y >> 32;
    int64_t x_low = x & 0xFFFFFFFFL;
    int64_t y_low = y & 0xFFFFFFFFL;

    int64_t z2 = x_low * y_low;
    int64_t t = x_high * y_low + ((uint64_t) z2 >> 32);
    int64_t z1 = t & 0xFFFFFFFFL;
    int64_t z0 = (uint64_t) t >> 32;
    z1 += x_low * y_high;
    return x_high * y_high + z0 + ((uint64_t) z1 >> 32);
}

int64_t multiply(int64_t a, int64_t b) {
  int64_t xh = multiplyHighUnsigned(a, b);
  int64_t xl = a * b;
  int64_t BARR_R = -M;
  int64_t xrh = multiplyHighUnsigned(xh, BARR_R);
  int64_t xrm = multiplyHighUnsigned(xl, BARR_R);
  int64_t add = xh * BARR_R;
  if((add ^ (1L << 63)) > ((xrm ^ (1L << 63)))) {
    xrh++;
  }

  int64_t t = xl - ((xrh << 2) | ((uint64_t) xrm >> 62)) * M - M;
  t += (t >> 63) & M;
  return t;
}
```

This code represents a `C` translation of the code discussed and provided in [10].

Appendix C

Source code

Note that the source code provided in this Appendix omits auxiliary functions and repeating code. The full compilable source code for each implementations can be found on the following `GitHub` repository: https://github.com/ghidirimschi/Convolution-Algorithms.

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <limits.h>
#include "timer.h"

#define CHECK_ALGORITHMS 1
#define LIMIT_CACHING   1


int *generateArray(int lowerLimit, int upperLimit, int size, double prob, int period) {
  int *array = malloc(size * sizeof(int));
  double rProb;
  for (int i = 0; i < period; ++i) {
    array[i] = (rand() % (upperLimit - lowerLimit + 1)) + lowerLimit;
  }
  if (!period) {
    array[0] = (rand() % (upperLimit - lowerLimit + 1)) + lowerLimit;
  }
  for(int i = !period ? 1 : period; i < size; i++) {
    rProb = (double)rand() / RAND_MAX;
    if(rProb <= prob) {
      if (period) {
        for (int j = 0; j < period && i+j < size; ++j) {
          array[i+j] = array[j];
        }
        i += period - 1;
      } else {
        array[i] = array[i - 1];
      }
    } else {
      array[i] = (rand() % (upperLimit - lowerLimit + 1)) + lowerLimit;
    }
  }
  return array;
}

int *readSignal(int *len) {
  int *x;
  char c;
```

```
42    scanf("%d:", len);
43    x = calloc(*len, sizeof(int));
44    do c = getchar(); while (c != '[');
45    int temp;
46    if (*len > 0) {
47      scanf("%d", &temp);
48      x[0] = temp;
49      for (int i=1; i < *len; i++) {
50        scanf(",%d", &temp);
51        x[i] = temp;
52      }
53    }
54    do c = getchar(); while (c != ']');
55    return x;
56  }
57
58  void printSignal(int len, int *x) {
59    if (len > 0) {
60      printf("[%d", x[0]);
61      for (int i = 1; i < len; i++) printf(",%d", x[i]);
62    }
63    printf("]");
64    return;
65  }
66
67  /************* Prototypes convolution algorithms **************/
68  void inputSideConvolution(int lenH, int *h, int lenX, int *x, int *y);
69  void outputSideConvolution(int lenH, int *h, int lenX, int *x, int *y);
70  void fftConvolution(int lenH, int *h, int lenX, int *x, int *y);
71  void karatsubaConvolution(int lenH, int *h, int lenX, int *x, int *y);
72  void duplConvolution(int lenH, int *h, int lenX, int *x, int *y);
73  void lrnopConvolution(int lenH, int *h, int lenX, int *x, int *y);
74  void lrnopConvolutionSort(int lenH, int *h, int lenX, int *x, int *y);
75  void oamfftConvolution(int lenH, int *h, int lenX, int *x, int *y);
76  void derivativeConvolution(int lenH, int *h, int lenX, int *x, int *y);
77  void nttConvolutionBarret(int lenH, int *h, int lenX, int *x, int *y);
78  void nttConvolutionSoftware(int lenH, int *h, int lenX, int *x, int *y);
79  void nttcrtConv(int lenH, int *h, int lenX, int *x, int *y);
80  void nttcrtBarrettConv(int lenH, int *h, int lenX, int *x, int *y);
81  /************* Timing routine ********************************/
82
83  double timeConvolutionAlgorithm(void (algorithm)(int,int*,int,int*,int*),
84          int lenH, int *h, int lenX, int *x, int *y) {
85    Timer timer;
86    double fastest = 999999999;
87    const int size = 20*1024*1024;
88    char *c = (char *)malloc(size);
89    for (int iteration=0; iteration<50; iteration++) {
90      #if LIMIT_CACHING
91      int i = 1;
92      for (int j = 0; j < size; j++)
93        c[j] = j * (++i*2);
94
95      #endif
96      startTimer(&timer);
97      algorithm(lenH, h, lenX, x, y);
98      stopTimer(&timer);
99      fastest = (seconds(timer) < fastest ? seconds(timer) : fastest);
100   }
101   free(c);
102   return fastest;
103 }
104
105 void clearHashMap();
106
107 int main(int argc, char *argv[]) {
108   void (*algorithm[])(int,int*,int,int*,int*) = {
109      fftConvolution, nttcrtConv, nttConvolutionBarret, nttcrtBarrettConv, ↩
           nttConvolutionSoftware, oamfftConvolution, lrnopConvolution, ↩
           derivativeConvolution, duplConvolution, inputSideConvolution, ↩
           outputSideConvolution, karatsubaConvolution
110   };
111
```

```
112   char name[][32] = {
113     "fftConvolution", "nttcrtConv", "nttConvolutionBarret", "nttcrtBarrettConv", ↩
          "nttConvolutionSoftware", "oamfftConvolution", "lrnopConvolution",  "↩
          derivativeConvolution", "duplConvolution", "inputSideConvolution", "↩
          outputSideConvolution", "karatsubaConvolution"
114   };
115
116   if ((argc < 3) || (argc > 5)) {
117     fprintf(stderr, "Usage: %s <lenX> <lenH> [probability from [0..1]] [period of↩
           h]\n", argv[0]);
118     return -1;
119   }
120
121   int lenX = atoi(argv[1]);
122   int lenH = atoi(argv[2]);
123   int lenY = lenX + lenH - 1;
124
125   double probability = (argc < 4 ? 0.0 : atof(argv[3]));
126   int period = (argc < 5 ? 0 : atoi(argv[4]));
127
128   int *h = generateArray(0, 100, lenH, probability, period);
129   int *x = generateArray(0, 100, lenX, probability, 0);
130   int *y = malloc(lenY*sizeof(int));
131
132 #if CHECK_ALGORITHMS
133   int *reference = malloc(lenY*sizeof(int));
134   outputSideConvolution(lenH, h, lenX, x, reference);
135
136   for (int alg=0; alg < sizeof(name)/32; alg++) {
137     algorithm[alg](lenH, h, lenX, x, y);
138     for (int i=0; i < lenY; i++) {
139       if (y[i] != reference[i]) {
140   printf("Fatal error: Algorithm %s produced wrong output (y[%d]=%d, while it ↩
          should have been %d)\n", name[alg], i, y[i], reference[i]);
141   exit(EXIT_FAILURE);
142       }
143     }
144   }
145   free(reference);
146 #endif
147
148   for (int i=0; i < sizeof(name)/32; i++) {
149     double elapsedTime = timeConvolutionAlgorithm(algorithm[i], lenH, h, lenX, x,↩
           y);
150     printf("%32s:    %lf miliseconds.\n", name[i], elapsedTime * 1000.0);
151     //printf("%lf\t", elapsedTime * 1000.0);
152
153   }
154   printf("\n");
155
156   free(y);
157   free(x);
158   free(h);
159   clearHashMap();
160   return 0;
161 }
```

direct.c

```
1  void inputSideConvolution(int lenH, int *h, int lenX, int *x, int *y) {
2    if (lenH > lenX) {
3      inputSideConvolution(lenX, x, lenH, h, y);
4      return;
5    }
6    int lenY = lenH + lenX - 1;
7    memset(y, 0, lenY*sizeof(int));
8    for(int i=0; i < lenX; i++) {
9      for(int j=0; j < lenH; j++) {
10       y[i+j] += x[i]*h[j];
```

```
    }
  }
}

void outputSideConvolution(int lenH, int *h, int lenX, int *x, int *y) {
  int lwb, upb, lenY = lenH + lenX - 1;
  for(int i = 0; i < lenY; i++) {
    lwb = (i < lenX ? 0: i-lenX+1);
    upb = (i < lenH ? i+1 : lenH);
    y[i] = 0;
    for (int j = lwb; j < upb; j++) {
      y[i] += h[j]*x[i-j];
    }
  }
}
```

fft.c

```
static void inplaceCooleyTukeyFFT(int length, double complex *a, double complex ↩
    omega, double complex *wsp) {.
  if (length < 2) return;
  length /= 2;
  double complex *even = wsp;
  double complex *odd = wsp + length;
  int idx = 0;
  for (int i=0; i < length; i++) {
    even[i] = a[idx++];
    odd[i] = a[idx++];
  }
  inplaceCooleyTukeyFFT(length, even, omega*omega, a);
  inplaceCooleyTukeyFFT(length, odd, omega*omega, a);
  double complex h, x = 1;
  for (int i=0; i < length; i++) {
    h = x*odd[i];
    a[i] = even[i] + h;
    a[i+length] = even[i] - h;
    x = x*omega;
  }
}

void inplaceFFT(int length, double complex *x) {
  double complex *wsp = malloc(length*sizeof(double complex)); y
  double complex omega = cexp(-2.0*PI*I/length);
  inplaceCooleyTukeyFFT(length, x, omega, wsp);
  free(wsp);
}

void inplaceInverseFFT(int length, double complex *x) {
  double complex *wsp = malloc(length*sizeof(double complex)); /
  double complex omega = cexp(2.0*PI*I/length);
  inplaceCooleyTukeyFFT(length, x, omega, wsp);
  free(wsp);
  for (int i=0; i < length; i++) x[i] /= length;
}

void fftConvolution(int lenH, int *h, int lenX, int *x, int *y) {
  int lenY = lenX + lenH - 1;
  int len2 = powerOfTwo(lenY);
  double complex *cx = calloc(2*len2, sizeof(double complex));
  double complex *ch = cx + len2;
  for (int i=0; i < lenX; i++) cx[i] = x[i];
  for (int i=0; i < lenH; i++) ch[i] = h[i];
  inplaceFFT(len2, ch);
  inplaceFFT(len2, cx);
  for (int i=0; i < len2; i++) cx[i] *= ch[i];
  inplaceInverseFFT(len2, cx);
  for (int i=0; i < lenY; i++) y[i] = (int)(cx[i]+0.5);
  free(cx);
}
```

karatsuba.c

```
void karatsubaConvolutionAux(int lenH, int *h, int lenX, int *x, int *y, int *wsp) {
  if (lenH == 1) {
    for (int i = 0; i < lenX; ++i) {
      y[i] = h[0] * x[i];
    }
    return;
  }
  int split = lenH/2;
  int *x0 = x + split;
  int *h0 = h + split;
  karatsubaConvolutionAux(split, h, split, x, y, wsp);
  y[2 * split - 1] = 0;
  karatsubaConvolutionAux(lenH - split, h0, lenX - split, x0, y + 2 * split, wsp);
  for (int i = 0; i < split; ++i) {
    x0[i] += x[i];
    h0[i] += h[i];
  }
  int z2Len = lenH - split + lenX - split - 1;

  int *z1 = wsp;

  karatsubaConvolutionAux(lenH - split, h0, lenX - split, x0, z1, z1 + z2Len);

  for (int i = 0; i < split; ++i) {
    x0[i] -= x[i];
    h0[i] -= h[i];
  }
  for (int i = 0; i < 2 * split - 1; ++i) {
    z1[i] -= y[i];
  }
  for (int i = 0; i < z2Len; ++i) {
    z1[i] -= y[2 * split + i];
    y[split + i] += z1[i];
  }
}

void karatsubaConvolution(int lenH, int *h, int lenX, int *x, int *y) {
  if (lenH > lenX) {
    karatsubaConvolution(lenX, x, lenH, h, y);
    return;
  }
  int wspSize = (32 - __builtin_clz(lenH)) * (lenX - 1);
  int *wsp = malloc(wspSize * sizeof(int));
  karatsubaConvolutionAux(lenH, h, lenX, x, y, wsp);
  free(wsp);
}
```

nttsoftware.c

```
static int64_t add(int64_t a, int64_t b) {
  a -= M;
  a += b;
  a += (a >> 63) & M;
  return a;
}

int64_t multiply(int64_t a, int64_t b) {
  return ((__int128) a * b) % M;
}

static void inplaceCooleyTukeyNTT(int length, int64_t *a, int64_t omega, int64_t *wsp) {
  if (length < 2) return;
  length /= 2;
```

```
  int64_t *even = wsp;
  int64_t *odd = wsp + length;
  int idx = 0;
  for (int i=0; i < length; i++) {
    even[i] = a[idx++];
    odd[i] = a[idx++];
  }
  inplaceCooleyTukeyNTT(length, even, multiply(omega, omega), a);
  inplaceCooleyTukeyNTT(length, odd, multiply(omega, omega), a);
  int64_t h, x = 1;
  for (int i=0; i < length; i++) {
    h = multiply(x, odd[i]);
    a[i] = add(even[i], h);
    a[i+length] = add(even[i], M - h);
    x = multiply(x, omega);
  }
}

static void inplaceNTT(int length, int64_t *x) {
  int64_t *wsp = malloc(length * sizeof(int64_t));
  int64_t omega = omegas[msb(length)][0];
  inplaceCooleyTukeyNTT(length, x, omega, wsp);
  free(wsp);
}

static void inplaceInverseNTT(int length, int64_t *x) {
  int64_t *wsp = malloc(length*sizeof(int64_t));
  int64_t omega = omegas[msb(length)][1];
  inplaceCooleyTukeyNTT(length, x, omega, wsp);
  free(wsp);
  for (int i=0; i < length; i++) x[i] = multiply(x[i], invN[msb(length)]);
}

void nttConvolutionSoftware(int lenH, int *h, int lenX, int *x, int *y) {
  int lenY = lenX + lenH - 1;
  int len2 = powerOfTwo(lenY);
  int64_t *cx = calloc(2*len2, sizeof(int64_t));
  int64_t *ch = cx + len2;
  for (int i=0; i < lenX; i++) cx[i] = x[i];
  for (int i=0; i < lenH; i++) ch[i] = h[i];
  inplaceNTT(len2, ch);
  inplaceNTT(len2, cx);
  for (int i=0; i < len2; i++) {
    cx[i] = multiply(cx[i], ch[i]);
  }
  inplaceInverseNTT(len2, cx);
  for (int i=0; i < lenY; i++) y[i] = (int)(cx[i]);
  free(cx);
}
```

nttbarrett.c

Replaces the `multiply`function in `nttsoftware.c` with the one presented in Appendix B.

nttcrt.c

```
static void chinese_remainder(int *n, int **a, int nrMods, long *y, int yLen)
{
  long prod = 7516372993, p[2] = {5010762410, 2505610584};

  for (int i = 0; i < nrMods; ++i) {
    for (int j = 0; j < yLen; ++j) {
      y[j] += ((long) a[i][j] * p[i]) % prod;
      if (y[j] > prod) {
        y[j] -= prod;
      }
```

```
    }
  }
}


void nttcrtConv(int lenH, int *h, int lenX, int *x, int *y) {
  int lenY = lenH + lenX - 1;
  int const nrMods = 2;
  int mods[] = {65537, 114689};

  int *nInv[] = {nInv65537, nInv114689};
  int (*omegas[])[] = {omegas65537, omegas114689};

  int **ym = malloc(nrMods * sizeof(int *));
  for (int i = 0; i < nrMods; ++i) {
    ym[i] = malloc(lenY * sizeof(int));
    nttConvolution(lenH, h, lenX, x, ym[i], mods[i], omegas[i], nInv[i]);
  }

  long *yL = calloc(lenY, sizeof(long));

  chinese_remainder(mods, ym, nrMods, yL, lenY);

  for (int i = 0; i < lenY; ++i) {
    y[i] = (int) yL[i];
  }

  for (int i = 0; i < nrMods; ++i) {
    free(ym[i]);
  }
  free(ym);
  free(yL);
}
```

nttcrtbarrett.c

Replaces the `multiply`function in `nttcrt.c` with the one presented in Appendix B.

oamfft.c

```
void oamfftConvolution(int lenH, int *h, int lenX, int *x, int *y) {
  if (lenH > lenX) {
    oamfftConvolution(lenX, x, lenH, h, y);
    return;
  }
  int N = 8 * powerOfTwo(lenH),
      step_size = N - (lenH - 1);

  double complex *ch = calloc(N, sizeof(double complex));
  double complex *cx = calloc(N, sizeof(double complex));
  for (int i=0; i < lenH; i++) ch[i] = h[i];
  inplaceFFT(N, ch);
  int position = 0;
  memset(y, 0, (lenX + lenH - 1) * sizeof(int));

  while (position + step_size <= lenX) {
    memset(cx, 0, N * sizeof(double complex));
    for (int i = 0; i < step_size; ++i) {
      cx[i] = x[position + i];
    }
    inplaceFFT(N, cx);
    for (int i = 0; i < N; i++) {
      cx[i] *= ch[i];
    }
    inplaceInverseFFT(N, cx);
    for (int i = 0; i < N; ++i) {
```

```
      y[position + i] += (int)(cx[i]+0.5);
    }
    position += step_size;
  }
  if (position != lenX) {
    step_size = lenX - position;
    memset(cx, 0, N * sizeof(double complex));
    for (int i = 0; i < step_size; ++i) {
        cx[i] = x[position + i];
    }
    inplaceFFT(N, cx);
    for (int i = 0; i < N; i++) {
      cx[i] *= ch[i];
    }
    inplaceInverseFFT(N, cx);
    for (int i = 0; i < lenX + lenH - position - 1; ++i) {
      y[position + i] += (int)(cx[i]+0.5);
  }
  free(ch);
  free(cx);
}
```

duplicate.c

```
#define MIN_SAMPLE -100
#define MAX_SAMPLE  100


#if (MAX_SAMPLE - MIN_SAMPLE < 10000) //use histogram sort for a limited dynamic range
void duplConvolution(int lenH, int *h, int lenX, int *x, int *y) {
  List *hist = calloc(MAX_SAMPLE - MIN_SAMPLE + 1, sizeof(List));
  memset(y, 0, (lenH + lenX - 1) * sizeof(int));
  int idx;
  for (int i = 0; i < lenH; ++i) {
    idx = h[i] - MIN_SAMPLE;
    hist[idx] = addItem(i, hist[idx]);
  }
  int *scaledX = malloc(lenX * sizeof(int));
  int shift;
  for (int i = 0; i < MAX_SAMPLE - MIN_SAMPLE + 1; ++i) {
    if (isEmptyList(hist[i])) {
      continue;
    }
    shift = i + MIN_SAMPLE;
    for (int j = 0; j < lenX; ++j) {
      scaledX[j] = shift * x[j];
    }
    while (!isEmptyList(hist[i])) {
      shift = firstItem(hist[i]);
      hist[i] = removeFirstNode(hist[i]);
      for (int j = 0; j < lenX; ++j) {
        y[shift + j] += scaledX[j];
      }
    }
  }
  free(scaledX);
  free(hist);
}
#else

typedef struct ElPos {
  int el;
  int pos;
} ElPos;

int cmpElPos(const void *a, const void *b) {
  int aEl = (*(ElPos*)a).el, bEl = (*(ElPos*)b).el;
```

```
44    if (aEl == -bEl) {
45      return bEl - aEl;
46    }
47    return abs(aEl) - abs(bEl);
48  }
49
50  void duplConvolution(int lenH, int *h, int lenX, int *x, int *y) {
51
52    ElPos *sortedH = malloc(lenH * sizeof(ElPos));
53    for (int i = 0; i < lenH; ++i) {
54      sortedH[i].el = h[i];
55      sortedH[i].pos = i;
56    }
57    qsort(sortedH, lenH, sizeof(ElPos), cmpElPos);
58    int *scaledX = malloc(lenX * sizeof(int));
59    memset(y, 0, (lenH + lenX - 1) * sizeof(int));
60
61    int i = 0,j;
62
63    int shift, scale;
64
65    while (!sortedH[i].el) {
66      ++i;
67    }
68
69    while (sortedH[i].el == 1) {
70      shift = sortedH[i].pos;
71      for (j = 0; j < lenX; ++j) {
72        y[j + shift] += x[j];
73      }
74      ++i;
75    }
76
77    while (sortedH[i].el == -1) {
78      shift = sortedH[i].pos;
79      for (j = 0; j < lenX; ++j) {
80        y[j + shift] -= x[j];
81      }
82      ++i;
83    }
84
85
86    while (i < lenH) {
87      shift = sortedH[i].pos;
88      scale = sortedH[i].el;
89      for (j = 0; j < lenX; ++j) {
90        scaledX[j] = scale * x[j];
91        y[j + shift] += scaledX[j];
92      }
93      ++i;
94      while (i < lenH && sortedH[i].el == scale) {
95        shift = sortedH[i].pos;
96        for (j = 0; j < lenX; ++j) {
97          y[j + shift] += scaledX[j];
98        }
99        ++i;
100     }
101     while (i < lenH && sortedH[i].el == -scale) {
102       shift = sortedH[i].pos;
103       for (j = 0; j < lenX; ++j) {
104         y[j + shift] -= scaledX[j];
105       }
106       ++i;
107     }
108   }
109   free(sortedH);
110   free(scaledX);
111 }
112 #endif
```

derivative.c

```c
void derivativeConvolution(int lenH, int *h, int lenX, int *x, int *y) {
  int lenY = lenH + lenX - 1;

  int *dH = malloc((lenH + 1) * sizeof(int));
  dH[0] = h[0];
  for(int i = 1; i < lenH; ++i) {
    dH[i] = h[i] - h[i - 1];
  }
  dH[lenH] = -h[lenH - 1];

  int *dY = malloc((lenY + 1) * sizeof(int));
  duplConvolution(lenH + 1, dH, lenX, x, dY);

  y[0] = dY[0];
  for (int i = 1; i < lenY; ++i) {
    y[i] = y[i - 1] + dY[i];
  }

  free(dH);
  free(dY);
}
```

lrnop.c

```c
int insertHash(int hash, int pos, int *str, int k) {
  List l = hm[hash];
  while (l != NULL) {
    if ((absll(pos - l->item) >= k) && !memcmp(str + pos, str + l->item, k * ↩
        sizeof(int))) {
      return l->item;
    }
    l = l->next;
  }
  hm[hash] = addItem(pos, hm[hash]);
  occupied = addItem(hash, occupied);
  return -1;
}

int substrings(int k, int *x, int lenX, int *other) {
  int powK = powMod(PRIME_BASE, k);
  int prevHash = hash(x, k);

  clearHashMap();
  insertHash(prevHash, 0, x, k);
  int tmpHash;
  int rtn;
  for (int i = 0; i < lenX - k; ++i) {
    tmpHash = (prevHash * PRIME_BASE + x[k + i]) % PRIME_MOD;
    tmpHash -= (powK * x[i]);
    tmpHash %= PRIME_MOD;
    if (tmpHash < 0) {
      tmpHash += PRIME_MOD;
    }
    rtn = insertHash(tmpHash, i + 1, x, k);
    if (rtn != -1) {
      *other = rtn;
      return i + 1;
    }
    prevHash = tmpHash;
  }
  return -1;
}

int findLongestSubstring(int *x, int lenX, int *s1, int *s2) {
  int l = 0, h = lenX, m;
  int ts1, ts2;
```

```
  int k = -1;

  while (l <= h && h >= MIN_REPEAT) {
    m = l + (h-l)/2;
    ts1 = substrings(m, x, lenX, &ts2);
    if (ts1 != -1) {
      *s1 = ts1 < ts2 ? ts1 : ts2;
      *s2 = ts2 > ts1 ? ts2 : ts1;
      l = m + 1;
      k = m;
    } else {
      h = m - 1;
    }
  }
  return k;
}

void lrnopConvolution(int lenH, int *h, int lenX, int *x, int *y) {
  int s1, s2, lenY = lenX + lenH - 1;
  int k = findLongestSubstring(h, lenH, &s1, &s2);
  memset(y, 0, lenY * sizeof(int));
  if (k == -1) {
    pickConvolution(lenH, h, lenX, x, y);
    return;
  }
  int *tY = malloc(lenY * sizeof(int));
  if (s1 != 0) {
    pickConvolution(s1, h, lenX, x, tY);
    for (int i = 0; i < s1 + lenX - 1; ++i) {
      y[i] += tY[i];
    }
  }



  lrnopConvolution(k, h + s1, lenX, x, tY);
  for (int i = 0; i < lenX + k - 1; ++i) {
    y[s1 + i] += tY[i];
    y[s2 + i] += tY[i];
  }


  if (s2 > s1 + k) {
    pickConvolution(s2 - s1 - k, h + s1 + k, lenX, x, tY);
    for (int i = 0; i < s2 - s1 - k + lenX - 1; ++i) {
      y[s1 + k + i] += tY[i];
    }
  }

  if (s2 + k < lenH) {
    pickConvolution(lenH - s2 - k, h + s2 + k, lenX, x, tY);
    for (int i = 0; i < lenH - s2 - k + lenX - 1; ++i) {
      y[s2 + k + i] += tY[i];
    }
  }
  free(tY);
}
```

Bibliography

[1] R. Agarwal and C. Burrus. 'Fast Convolution using fermat number transforms with applications to digital filtering'. In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 22.2 (1974), pp. 87–97. DOI: 10.1109/TASSP.1974.1162555.

[2] *Barrett reduction algorithm*. 2021. URL: https://www.nayuki.io/page/barrett-reduction-algorithm.

[3] Jon Louis Bentley, Dorothea Haken and James B. Saxe. 'A General Method for Solving Divide-and-Conquer Recurrences'. In: *SIGACT News* 12.3 (Sept. 1980), pp. 36–44. ISSN: 0163-5700. DOI: 10.1145/1008861.1008865. URL: https://doi.org/10.1145/1008861.1008865.

[4] Augustin-Louis Cauchy. *Cours d'analyse de l'Ecole royale polytechnique*. Cambridge University Press, 1821.

[5] James W. Cooley and John W. Tukey. 'An Algorithm for the Machine Calculation of Complex Fourier Series'. In: *Mathematics of Computation* 19.90 (1965). 297, pp. 297–301. ISSN: 0025-5718.

[6] Alejandro Dominguez. *Origin and history of convolution*. 2014. URL: https://www.slideshare.net/Alexdfar/origin-adn-history-of-convolution.

[7] Anatolii Karatsuba. 'The complexity of computations'. In: *Proceedings of the Steklov Institute of Mathematics* 211 (Jan. 1995), pp. 169–183.

[8] Richard M. Karp and Michael O. Rabin. *Efficient randomized pattern-matching algorithms*. 1987.

[9] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Third. Boston: Addison-Wesley, 1997, pp. 296–305.

[10] *Notes on FFT/NTT, and the "ultimate" NTT*. 2020. URL: https://codeforces.com/blog/entry/75326.

[11] Lawrence R. 1943- Rabiner and Bernard Gold. *Theory and application of digital signal processing*. English. Englewood Cliffs, N.J.: Prentice-Hall, 1975.

[12] Steven W Smith. *The scientist and engineer's guide to digital signal processing*. California Technical Pub., 2002, pp. 112–121.

[13] Paul N. Swarztrauber. 'FFT algorithms for vector computers'. In: *Parallel Computing* 1.1 (1984), pp. 45–63. ISSN: 0167-8191. DOI: https://doi.org/10.1016/S0167-8191(84)90413-7. URL: https://www.sciencedirect.com/science/article/pii/S0167819184904137.

[14] Emmanuel Thomé. 'Karatsuba multiplication with temporary space of size $\leq$ n'. working paper or preprint. Sept. 2002. URL: https://hal.archives-ouvertes.fr/hal-02396734.

[15] Alberto Zanoni. 'Toom-Cook 8-way for Long Integers Multiplication'. In: *2009 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 2009, pp. 54–57. DOI: 10.1109/SYNASC.2009.23.