Integer Multiplication in Time O(n log n) - A Study

by

Vighnesh Iyer

A dissertation submitted in partial fulfilment of the requirements for the degree of

B.Sc Honours in Mathematics

2020-2023

Supervised by: Divakaran D
Evaluated by: R Ramanujam

Integer Multiplication in $O(n \log n)$

Vighnesh Iyer

Abstract

Minimising running times for multiplication algorithms has been a quest for computer scientists in the field ever since Karatsuba's algorithm came out in 1960, shattering the perception that multiplication is an $\Omega(n^2)$ operation. The development of the Fast Fourier Transform has contributed significantly to the field of integer multiplication, allowing for lower bounds to be obtained. In 1971, Schönhage and Strassen used Fast Fourier Transforms to describe an algorithm that ran in time $O(n \log n \log \log n)$. They conjectured that $O(m \log n)$ was the optimal bound for integer multiplication, but an algorithm that achieved the bound followed by a proof of optimality were beyond their reach.

In 2019, David Harvey and Joris van der Hoeven described an algorithm for integer multiplication utilising Fast Fourier Transforms over multivariate polynomial rings that runs in time $O(n \log n)$. A proof of the optimality of this result (or indeed an algorithm that is faster) has not been described yet, however. In this report, we aim to study the algorithm and make it accessible to an undergraduate audience.

Dedication

To the APU Math community - for helping me discover beauty in a subject that scared me for so long;

and my family, who inspire me to be the best version of myself, every moment of every day.

Declaration

I hereby declare that the work in this thesis has been carried out by me, in the B. Sc (Honours) Program, under the supervision of Dr. Divakaran D, and in the partial fulfillment of the requirements for the award of the degree of B. Sc (Honours) at the Azim Premji University, Bangalore. I further declare that this work has not been the basis for the award of any degree, diploma or any other title elsewhere.

Acknowledgements

Any endeavour such as this one is not accomplished through the work of a single individual. I would like to thank my project supervisor Dr. Divakaran D for providing me with constant encouragement and support and humouring my ceaseless curiosity every step of the way. I would also like to thank my project mentor Prof. R. Ramanujam for providing extremely useful advice and support along the way.

I want to thank my friends, who were a source of consistent moral support throughout this journey. And lastly, I would like to thank my family, for patiently sitting listening to me narrate all the highs and lows of my project. It was all worth it.

Contents

Chapter 1

Introduction

For almost as long as we have been able to count, we have needed to multiply natural numbers. Most ancient civilisations had their own methods of multiplying integers. For the longest time though, people were only concerned with the ease with which human beings could compute these products, and making sure that the procedures they came up with were accurate. Eventually, the long multiplication all of us learnt in the third grade became the standard multiplication procedure taught in school, and this is something that most of us use to this day.

The advent of computers and development in computational thinking has, however, led many to ask the question: can we describe algorithms that give us the ability to calculate integer products quickly on a computer? This question is not a trivial one, since what is easy for a computer may not be particularly easy for a human being. Furthermore, it is a question that has pretty immediate practical importance. The functioning of most of the electronic devices that form such an inalienable part of our lives today - our phones, our laptops and so on - rely on the multiplication of large numbers. Answering this question satisfactorily, therefore, is pretty important in order for us to optimise the functioning of large parts of our lives.

Let us take as an example of multiplication algorithms the procedure we are taught in school. In order to multiply two $n-$ digit numbers, we first multiply each of the $n-$digits of the first number to each of the $n-$digits of the second number. This results in n^2 single-digit multiplications. We then have to add n numbers together. For the moment, we adopt the

convention that the multiplication of two single-digit numbers and the addition of two numbers are operations that may be conducted very quickly by any computer, and constitute a single 'step'. In general then, when we multiply two $n-$digit numbers we utilise about n^2 steps to do so (technically we utilise $n^2 + n$ steps to perform the multiplication, but as n grows large, the n term begins to appear quite small compared to the n^2 term, so n^2 serves as a good enough proxy for $n^2 + n$). Now as n increases, n^2 blows up pretty quickly, so this is not ideal. The question remains therefore: can we do better?

Indeed in 1960, renowned Russian mathematician Andrey Kolmogorov claimed that any multiplication algorithm must utilise a minimum of n^2 such steps. Within a week, a 23-year-old student by the name of Anatoly Karatsuba had come up with an algorithm that took about $n^{\log_2(3)}$ steps, clearly less than the n^2 bound that Kolmogorov had conjectured (admittedly, we are being a bit imprecise when we say that the algorithm takes 'about $n^{\log_2(3)}$ steps to run. One might even be confused as to what exactly constitutes a 'step' here. All of these are valid questions and will be cleared up in later chapters, so we shall deal with them then). With that, a race to find the fastest multiplication algorithm (in terms of time taken to multiply two $n-$digit integers) began, and it continues to this day. Among the defining moments of the quest for fast multiplication algorithms was in 1971. Two German mathematicians, Arnold Schönhage and Volker Strassen described an algorithm that for very large values of n, computed the products of $n-$digit integers in about $n \log n \log \log n$ steps. They conjectured that the best one could possibly do was $n \log n$ steps, but neither they, nor anyone else in the field had any idea how to get there.

Enter David Harvey and Joris van der Hoeven. In 2019, they described two separate algorithms, one conditional on a number theoretic result, and one unconditional (that's the one we'll be looking at) that achieved the hallowed $n \log n$ figure. A full 48 years after Schönhage and Strassen's conjecture, the bound they claimed could be reached was indeed reached. The same question that we've been asking all along still remains though: can we do better? And at the moment, the answer is a rather exciting: we don't know.

Another thing to be noted here is that the algorithms that Harvey and van der Hoeven have

come up with reach the $n \log n$ bound for integers that are so large that multiplying them has no practical importance whatsoever. The algorithms just demonstrate that the $n \log n$ bound may be reached. Why this is particularly interesting is because it demonstrates something very typical of most mathematics: a problem that starts off with some practical motivations is eventually loses all sense of those very same motivations in order to achieve an optimal solution. So, while Harvey and van der Hoeven have shown that Schönhage and Strassen's conjectured bound can be reached, don't expect to see their algorithm implemented any time soon!

In this report, we'll be examining Harvey and van der Hoeven's unconditional algorithm in detail described in Harvey and Hoeven 2021. We will establish most (if not quite all) of the machinery needed to understand the algorithm in the report itself, and hopefully by the end of it, the reader has an idea of how this algorithm works. More than that though, we hope that the reader is left with a feeling of wonderment just as we were at the fact that a problem that started out with an aim to multiply integers quickly has spawned so many brilliant and beautiful ideas, and has captivated the leading minds in computer science for over half a century.

Chapter 2

An Introduction to the Discrete Fourier Transform

2.1 The Aim of this Chapter

In this chapter, we will study an operation defined on certain kinds of rings known as a Discrete Fourier Transform. While it might appear a bit strange to introduce this concept right off the bat, suffice it to say that computing Discrete Fourier Transforms forms an integral part of Harvey and van der Hoeven's algorithm for fast integer multiplication. Indeed, since Schonhage and Strassen's algorithm (1971), most fast multiplication algorithms have utilised Discrete Fourier Transforms to achieve low running times. In most of these algorithms, we establish an equivalence between multiplying integers, and multiplying and then subsequently evaluating polynomials (see this for more details). There is a connection between Discrete Fourier Transforms and calculating polynomial products which will be established towards the end of this chapter when we prove a result called the Convolution Theorem.

2.2 Defining the Discrete Fourier Transform

In general, a Discrete Fourier Transform is a map defined from R^n to R^n, where R is a ring that has some specific properties. We describe these properties below.

2.2.1 Imposing Necessary Conditions on R

In order to define the transform on R^n, we require the following to be true of R:

- Multiplication among elements in R must be commutative.
- R must have a multiplicative identity element (called unity, represented by 1).
- There must exist an element ω in R, which is the primitive n^{th} root of unity. We call ω the primitive n^{th} root of unity when $\omega^n = 1$, but $\omega^j \neq 1$ for $j \in \{1, \ldots, n-1\}$.

We will be able to define the Fourier Transform for any ring with these conditions. However, in the paper itself, Harvey and van der Hoeven define Fourier Transforms on structures called $\mathbb{C}-$algebras. A $\mathbb{C}-$algebra is a ring that also has a vector space structure over $\mathbb{C}$. In the paper, Harvey and van der Hoeven deal specifically with finite-dimensional $\mathbb{C}-$algebras, that have a canonical basis. They use the term *coefficient ring* to describe such objects, and we shall be sticking to this nomenclature as well.

Harvey and van der Hoeven deal mainly with two kinds of coefficient rings, $\mathbb{C}$ itself and the coefficient ring $\mathscr{R} = \mathbb{C}[y]/\langle y^r + 1\rangle$. The canonical bases for these algebras are $\{1\}$ and $\{1, y, y^2, \ldots, y^{r-1}\}$ respectively. That the first two conditions mentioned above are satisfied by both these rings is pretty easy to see. What we must now define is what constitutes an n^{th} root of unity in such rings.

For $\mathbb{C}$, the n^{th} root of unity is known to be $e^{2\pi i/n}$. It is easy to see why this is the case, since raising this number to the n^{th} power gives us 1 and $e^{k(2\pi i)/n}$ when $1 \leq k < n$ is not equal to 1.

As for the second coefficient ring we consider, i.e., $\mathscr{R} = \mathbb{C}[y]/\langle y^r + 1\rangle$, we will only need to define the transform over $\mathscr{R}^n$ when n is a positive divisor of $2r$. We claim that $y^{2r/n}$ is a primitive n^{th} root of unity. Why? It is clear in the above ring that $y^r = -1$, which means that y^{2r} is equal to 1. Furthermore, for any integer $1 \leq k < n$, $y^{2rk/n} \neq 1$. (essentially what is happening here is that n is the smallest non-negative integer for which the denominator is completely 'cut-out', leaving us with a y^{2r} term. Try it if you aren't convinced!)

We now define the map on R^n, assuming that R is a coefficient ring satisfying the necessary conditions.

2.2.2 The Map Itself

Having imposed the necessary conditions on R, we now define the Discrete Fourier Transform map $F_\omega : R^n \to R^n$ as given below. Let $u = (u_0, \ldots, u_{n-1})$ be a vector in R^n. Then, the j^{th} coordinate of $F_\omega u$ is given by:

$$(F_\omega u)_j = \frac{1}{n} \sum_{k=0}^{n-1} \omega^{-jk} u_k$$

We can understand this map better by observing that:

$$F_\omega(u_0, \ldots, u_{n-1}) = \frac{1}{n} \cdot \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \ldots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \ldots & \omega^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \ldots & \omega^{-(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{n-1} \end{bmatrix}$$

There's something quite deep going on here which the reader should be mindful of. Notice that above, we didn't say j^{th} component, we said j^{th} coordinate. This is important since each u_j is itself a vector in R! That's why we're relying on R being a coefficient ring here, only then can we actually multiply elements of R together, and then add the products. This distinction doesn't really change notation though, so we shall proceed as before, keeping in mind the fact that u_j itself is not a complex number, but a vector in R.

We note that the jk^{th} element of the above matrix is $\frac{\omega^{-jk}}{n}$, provided the rows and columns of the matrix are indexed from 0 to $n-1$, instead of the standard 1 to n for an $n \times n$ matrix.

Suppose ω is a primitive n^{th} root of unity. We now aim to show that ω^{n-1} is a primitive n^{th} root of unity as well.

Lemma 1. *Let ω be a primitive n^{th} root of unity in a coefficient ring R. Then, $\omega^{n-1} = \omega - 1$ is also a primitive n^{th} root of unity as well.*

Proof:

It is clear that $\omega^{(n-1)n} = (\omega^n)^{n-1} = 1$. Now we consider $j \in \{1, \ldots, n-1\}$. $\omega^{(n-1)j} = \omega^{nj}\omega^{-j}$.

Since $\omega^{nj} = (\omega^n)^j$ and $\omega^n = 1$ (ω is a primitive root of unity), we can say that $\omega^{nj} = 1$. This means that $\omega^{(n-1)j} = \omega^{-j}$. Now $j \neq 0 \mod n$. This means that $-j \neq 0 \mod n$. This means that $-j = k \mod n$, where $k \in \{1, \ldots, n-1\}$. Since ω is a primitive n^{th} root of unity, $\omega^{-j} = \omega^k \neq 1$ when $k \in \{1, \ldots, n-1\}$. Therefore, ω^{n-1} is a primitive n^{th} root of unity as well, and we are done.

This means that we can use $\omega^{n-1} = \omega^{-1}$ to define the map $F_{\omega^{-1}} : R^n \to R^n$ appropriately.

Explicitly, we may define $F_{\omega^{-1}} : R^n \to R^n$ as below:

$$(F_{\omega^{-1}} u)_j = \frac{1}{n} \sum_{k=0}^{n-1} \omega^{jk} u_k$$

You might be wondering if $F_{\omega^{-1}}$ is an inverse for F_ω. If you guessed yes, then you'd be wrong, but not very far off the mark. We ilustrate why by proving the lemma below:

Lemma 2. *Let $w \in R^n$, and let ω be the n^{th} root of unity in R. Let F_ω and $F_{\omega^{-1}}$ be maps defined from $R^n \to R^n$ as above. Then,*

$$F_{\omega^{-1}}(F_\omega w) = \frac{1}{n} w$$

Proof. Consider $w = (w_0, \ldots, w_{n-1}) \in R^n$.

$$(F_{\omega^{-1}}(F_\omega w))_j = \frac{1}{n} \sum_{s=0}^{n-1} \omega^{sj} \left(\frac{1}{n} \sum_{t=0}^{n-1} \omega^{-st} w_t \right)$$

where we have used the fact that the s^{th} coordinate of $F_\omega w = \frac{1}{n} \sum_{t=0}^{n-1} \omega^{-st} w_t$. We collect the terms that each w_t is multiplied by. Doing so leaves us with a summation that looks like the one below:

$$(F_{\omega^{-1}}(F_\omega w))_j = \frac{1}{n^2} \sum_{t=0}^{n-1} \left(\sum_{s=0}^{n-1} \omega^{s(j-t)} \right) w_t$$

Now by the definition of ω, for any $i \in \{1, \cdots, n-1\}$, $\omega^i \neq 1$. Let us look at the sum $\sum_{j=0}^{n-1} (\omega^i)^j$. This is a geometric series with common difference ω^i, and first term $\omega^0 = 1$. Its sum works out to be $\frac{1(\omega^{ni}-1)}{\omega^i-1} = 0$, since $\omega^i \neq 1$.

What this means for our summation is the following: If $j = t \mod n$, since the quantity $s(j-t)$ works out to be some multiple of n, $\omega^{s(j-t)} = 1$. This means for $j = t$, we get the expression nw_j inside the first sum. However, if $j \neq t \mod n$, we see from above that this evaluates to 0. Since $j, t \in \{0, \ldots, n-1\}$, there is exactly one situation where $j = t \mod n$, and that is when $j = t$. This of course means that:

$$\begin{aligned}
(F_{\omega^{-1}}(F_\omega w))_j &= \frac{1}{n^2} \sum_{t=0}^{n-1} \left(\sum_{s=0}^{n-1} \omega^{s(j-t)} \right) w_t \\
&= \frac{1}{n^2} n w_j + \sum_{j \neq t \mod n} \left(\sum_{s=0}^{n-1} \omega^{s(j-t)} \right) w_t \\
&= \frac{1}{n} w_j + \sum_{j \neq t \mod n} 0 \cdot w_t \\
&= \frac{1}{n} w_j
\end{aligned}$$

We have proved that every coordinate of $F_{\omega^{-1}}(F_\omega w)$ is equal to the corresponding coordinate of $\frac{1}{n}w$. Therefore, $F_{\omega^{-1}}(F_\omega w) = \frac{1}{n}w$ and we are done. □

2.2.3 Defining Convolutions on R^n

We define a binary operation known as convolution on the coefficient ring R^n. Consider $u, v \in R^n$, where $u = (u_0, \ldots, u_{n-1})$ and $v = (v_0, \ldots, v_{n-1})$. Then, $u * v$ is another vector in R^n, such that

$$(u * v)_j = \sum_{k=0}^{n-1} u_k v_{j-k}$$

This definition raises a natural question. Consider $j = 0$, for instance. When we evaluate

$j-k$, for $k=n-1$ (say), we get $-(n-1)$. What does $v_{-(n-1)}$ even mean? We note that this problem is not unique to $j=0$, $k=n-1$. It happens every time we have values such that $j<k$. It is imperative, therefore, that we adjust our definition keeping this in mind. From here on out, unless specified otherwise, when we talk about u_j we will always mean $u_{j \mod n}$. Clearly then, it does not make a difference if $j-k$ takes a negative value in our above definition, since $j-k \mod n \in \{0,1,\ldots,n-1\}$, for every value of j and k.

That $(R^n,+,*)$ forms a commutative ring is a result whose proof is omitted here, but one the reader should try proving in case they are not convinced of it.

In fact, one can prove something even stronger. We can show that $(R^n,+,*) \cong R[x]/\langle x^n-1\rangle$. A proof of this fact is presented in brief below.

Proposition 3. $(R^n,+,*) \cong R[x]/\langle x^n-1\rangle$

Proof. We know that $R[x]/\langle x^n-1\rangle$ and R^n form vector spaces over $\mathbb{C}$, since they are $\mathbb{C}-$algebras. Let $A=(a_0,\ldots,a_{n-1})=(1,x,\ldots,x^{n-1})$ be an ordered basis for $R[x]/\langle x^n-1\rangle$. Let $B=(b_0,\ldots,b_{n-1})=((0,\ldots,0,1),(0,\ldots,1,0),\ldots,(1,\ldots,0,0))$ be an ordered basis for R^n. We define $\phi: A \to B$ as $\phi(a_i)=b_i$.

Since both the sets mentioned above are vector spaces, we extend this map linearly to form the map $\Phi: R[x]/\langle x^n-1\rangle \to R^n$, defined as below:

$$\Phi\left(\sum_{i=0}^{n-1}\lambda_i a_i\right)=\sum_{i=0}^{n-1}\lambda_i\phi(a_i)=\sum_{i=0}^{n-1}\lambda_i b_i$$

Let $A_1=\sum_{j=0}^{n-1}\lambda_j a_j$ and $A_2=\sum_{k=0}^{n-1}\mu_k a_k$. Let $\Phi(A_1)=B_1$, $\Phi(A_2)=B_2$. We must now show that

- $\Phi(A_1+A_2)=\Phi(A_1)+\Phi(A_2)$
- $\Phi(A_1A_2)=\Phi(A_1)*\Phi(A_2)$

We first prove that $\Phi(A_1+A_2)=\Phi(A_1)+\Phi(A_2)$.

$$
\begin{aligned}
\Phi(A_1 + A_2) &= \Phi\left(\sum_{j=0}^{n-1} \lambda_j a_j + \sum_{k=0}^{n-1} \mu_k a_k\right) \\
&= \Phi\left(\sum_{l=0}^{n-1} (\lambda_l + \mu_l) a_l\right) \\
&= \sum_{l=0}^{n-1} (\lambda_l + \mu_l) b_l \\
&= \sum_{j=0}^{n-1} \lambda_j b_j + \sum_{k=0}^{n-1} \mu_k b_k \\
&= \Phi(A_1) + \Phi(A_2)
\end{aligned}
$$

We now prove that $\Phi(A_1 A_2) = \Phi(A_1) * \Phi(A_2)$.

$$
\begin{aligned}
\Phi(A_1 A_2) &= \Phi\left(\left(\sum_{j=0}^{n-1} \lambda_j a_j\right)\left(\sum_{k=0}^{n-1} \mu_k a_k\right)\right) \\
&= \Phi\left(\sum_{i=0}^{n-1} a_i \left(\sum_{s=0}^{n-1} \lambda_s \mu_{i-s}\right)\right)
\end{aligned}
$$

One might be slightly confused what's happening in the last step above. We note that we're multiplying polynomials modulo $x^n - 1$ in the ring $R[x]/\langle x^n - 1\rangle$. This means that $x^n = 1$. What this means is that suppose $j + k = i \mod n$, the product of the coefficients of x^j in one of the polynomials and x^k in the other forms a part of the coefficient of x^i in the product polynomial. We have just collected all such pairwise products in the above step.

Now

$$\begin{aligned}\Phi(A_1A_2) &= \Phi\left(\sum_{i=0}^{n-1} a_i \left(\sum_{s=0}^{n-1} \lambda_s \mu_{i-s}\right)\right) \\ &= \sum_{i=0}^{n-1} b_i \left(\sum_{s=0}^{n-1} \lambda_s \mu_{i-s}\right)\end{aligned}$$

We know that $B_1 = \sum_{j=0}^{n-1} \lambda_j b_j$, and $B_2 = \sum_{k=0}^{n-1} \mu_k b_k$. We had mentioned that given $u = \sum_{i=0}^{n-1} u_i b_i$ and $v = \sum_{l=0}^{n-1} v_l b_l$, $u * v = \sum_{j=0}^{n-1} b_j \sum_{k=0}^{n-1} u_k v_{j-k}$. This means that

$$B_1 * B_2 = \sum_{i=0}^{n-1} b_i \left(\sum_{s=0}^{n-1} \lambda_s \mu_{i-s}\right) = \Phi(A_1A_2)$$

.

Therefore, we see that $\Phi(A_1A_2) = \Phi(A_1) * \Phi(A_2)$, and we are done.

□

2.3 The Convolution Theorem

Theorem 4. *For any $u, v \in R^n$, we have:*

$$\frac{1}{n} u * v = n F_{w^{-1}} \left(F_\omega u \cdot F_\omega v\right)$$

Proof. We prove the Convolution Theorem in two stages. Firstly, we prove that $F_\omega u \cdot F_\omega v = \frac{1}{n} F_\omega (u * v)$, for two vectors $u, v \in R^n$. We then use lemma 2, setting $u * v = w$ in order to obtain the result described above. (**Note:** This exact proof appears as lemma 2.3 in Harvey and van der Hoeven's paper).

Let $u = (u_0, \ldots, u_{n-1})$, $v = (v_0, \ldots, v_{n-1})$ be elements in R^n. We now consider the j^{th} coordinate of $F_\omega u \cdot F_\omega v$, and prove that this is equal to the j^{th} coordinate of $\frac{F_\omega (u*v)}{n}$. Since we assume nothing about the nature of j for the course of this proof, it follows that the two vectors are equal, and we will be done.

$$\begin{aligned}(F_\omega u)_j \cdot (F_\omega v)_j &= \left(\frac{1}{n}\sum_{s=0}^{n-1}\omega^{-sj}u_s\right)\cdot\left(\frac{1}{n}\sum_{t=0}^{n-1}\omega^{-tj}v_t\right)\\ &= \frac{1}{n^2}\sum_{s=0}^{n-1}\sum_{t=0}^{n-1}\omega^{-j(s+t)}u_s v_t\\ &= \frac{1}{n^2}\sum_{k=0}^{n-1}\omega^{-jk}\sum_{s+t=k \mod n} u_s v_t\end{aligned}$$

We note that in the last step above, all we are doing is collecting terms that are multiplied by the same power of ω^{-j}. We have used the variable k to represent the sum of s and t here, but this sum is represented modulo n since $\omega^n = 1$ (ω is the n^{th} root of unity). We note that the i^{th} coordinate of $u * v = \sum_{s=0}^{n-1} u_s v_{i-s}$. The expression $\sum_{s+t=k \mod n} u_s v_t$ may be rewritten as $\sum_{s=0}^{n-1} u_s v_{k-s}$, which means that the term ω^{-jk} is multiplied to the k^{th} coordinate of $u * v$. This means that:

$$\begin{aligned}(F_\omega u)_j \cdot (F_\omega v)_j &= \frac{1}{n^2}\sum_{k=0}^{n-1}\omega^{-jk}\sum_{s=0}^{n-1}u_s v_{k-s}\\ &= \frac{1}{n}\cdot\frac{1}{n}\sum_{k=0}^{n-1}\omega^{-jk}(u*v)_k\\ &= \frac{1}{n}(F_\omega(u*v))_j\end{aligned}$$

Therefore, we have shown that $F_\omega u \cdot F_\omega v = \frac{1}{n}F_\omega(u*v)$.

Now let us set $w = u * v$. We know that from lemma 2 that $F_{\omega^{-1}}(F_\omega(u*v)) = \frac{1}{n}u*v$. However, we also know that $F_\omega(u*v) = n(F_\omega u \cdot F_\omega v)$ from above. Using the fact that F_ω is linear, we get:

$$\frac{1}{n}u*v = nF_{w^{-1}}\left(F_\omega u \cdot F_\omega v\right)$$

□

Chapter 3

Understanding Running Time of Algorithms ft. the Fast Fourier Transform

3.1 The Aim of This Chapter

The title of Harvey and van der Hoeven's paper is 'Integer Multiplication in $O(n \log n)$'. The term integer multiplication needs no explanation for anyone who has passed the fourth grade, but what does $O(n \log n)$ mean? This is what we aim to give an intuitive idea of in this chapter. Understanding big-oh notation is key to understanding what is referred to as the running time of an algorithm. This is effectively the number of steps it takes for an algorithm to produce an output when given an input of a particular size.

The introduction to running times of algorithms is demonstrated in this chapter using a particularly beautiful algorithm known as the Fast Fourier Transform (FFT). It is an efficient algorithm that is used to calculate the Discrete Fourier Transform described above. The FFT algorithm was originally conceptualised by Gauss but was made popular by Cooley and Tukey when they rediscovered it and subsequently wrote a paper about it in 1965. The algorithm presented here follows the discussion in Section 2.4 of Harvey and Hoeven 2021.

3.2 Running Times of Algorithms

Algorithms are procedures that take an input and give us an output in a **finite** number of steps. The word finite is of paramount importance here, since anything that takes an infinite number of steps cannot be relied on to produce an output at all. The time it takes an algorithm to run, therefore, may be expressed as some constant multiplied by the number of steps it takes (the constant here represents the time taken to perform one step).

It is natural to question, however, what we mean by number of steps. What will this number be expressed in terms of? Indeed, one might ask: what constitutes a step at all? All of these are valid questions. The answer to the first is not too difficult to understand. Any algorithm will likely take a longer time to produce an output for a larger input. As a result, measuring the number of steps in terms of the 'size' of the input is something that would make sense. As for what constitutes a step, this is a question that has several possible answers. In this chapter alone, we shall see two possible answers to this question, and during the course of studying Harvey and van der Hoeven's multiplication algorithm, we shall see another.

We can make our understanding of the running times of algorithms clearer using a concrete example. Let us consider a simple algorithm for finding the maximum in an array of integers:

Algorithm 1 A Simple Algorithm for Sorting an Array

Input: An array $A[0, \ldots, n-1]$ containing n integers.
Output: An integer k such that k is the largest integer in A.

```
1: k ← A[0]
2: for i in [1, ... n − 1] do
3:     if A[i] ≥ k then:
4:         k ← A[i]
5:     end if
6: end for
7: return k
```

That this algorithm correctly produces the largest integer in a given array is not very hard to check. We assign the value of the first integer of the array to k and then compare this element with every other element of the array. If we find some element that is greater in value than k, we assign the value of this element to k and continue this process. Eventually, k will hold the value of the largest integer in the array since $\geq$ is a transitive relation on the integers.

Analysing the running time of this algorithm is pretty straightforward. We assign k the value of the first element of the array and then compare it with all the other $n-1$ elements of the array. We also assign k the value of the i^{th} element of the array, in case this element is greater in value than k. For our purposes, we will always deal with the worst-case analysis of algorithms. This means that we will understand how long it takes for the algorithm to correctly give an output on the worst-case input.

What we must answer then is: what is the worst-case input for this algorithm? For any algorithm, the worst-case input is one which makes the processor do the most amount of work. We see that we must compare k to $n-1$ other integers irrespective of how our array is organised. What does change depending on how the array is organised is the number of times k is reassigned, however. The worst-case input would, therefore, maximise the number of reassignments that occur. It is easy to see that any array in ascending order would have a total of n assignments (1 initial assignment, followed by $n-1$ consecutive reassignments since every element is greater than or equal to every element that came before it by definition). Say each reassignment takes c_1 time, where c_1 is a constant, and each comparison takes c_2 time. Then, for an array with $n-$elements arranged in ascending order, this algorithm would take $c_1(n) + c_2(n-1) = (c_1 + c_2)n - c_2$ time to complete running. Now since c_2 and c_1 are constants, we may say that the algorithm runs in 'linear' time, meaning that for an input of size n, the running time is some constant times the size of the input.

We note here that the algorithm described above is a really simple one. We will see algorithms that are more complicated, and their running time might not be expressed in such a straightforward manner. For instance, there might be an algorithm whose running time for an input of size n comes out to be $3n^4 + 6n^2 + 2^n$. How would we deal with something like this? We try and understand how to make sense of such expressions in the section below.

3.2.1 Big-Oh Notation

It is clear that we express the running time of any algorithm as some function $f : \mathbb{N} \to \mathbb{N}$ (the size of our input is always a positive integer). We now define what is referred to as big-oh

notation. Let $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$ be functions on the natural numbers. $f(n)$ is said to be $O(g(n))$ if there exists some constant c, and some natural number n_0, such that for all $n \geq n_0$, $|f(n)| \leq |g(n)|$. We work through some examples to build an intuition of how big-oh notation works.

Example 3.2.1. *$f : \mathbb{N} \to \mathbb{N}$ as $f(n) = n$ is $O(n)$.*

Proof. Take $c = 1$, $n_0 = 1$. We are done. □

Example 3.2.2. *$f : \mathbb{N} \to \mathbb{N}$ as $f(n) = n$ is $O(n^2)$.*

Proof. The same choice of c and n_0 above work here as well. □

Example 3.2.3. *Let $f(n) : \mathbb{N} \to \mathbb{N}$ as $f(n) = \sum_{i=0}^{m} a_i n^i$. Then, $f(n)$ is $O(n^m)$.*

Proof. Take $p = \max_i |a_i|$. Let $n_0 = 1$, and $c = (m+1)p$. This works since $\sum_{i=0}^{m} a_i n^i \leq \sum_{i=0}^{m} |a_i| n^i \leq \sum_{i=0}^{m} p n^i \leq (m+1) p n^m$ (where we have used the fact that for any natural number n, if $a \geq b$, $n^a \geq n^b$). □

We can similarly prove other results. For instance, if $f : \mathbb{N} \to \mathbb{N}$ is defined such that $f(n) = \sum_{i=0}^{m} a_i n^i$, then $f(n) = O(2^n)$. Another interesting result is given in the last example below.

Example 3.2.4. *Let $f : \mathbb{N} \to \mathbb{N}$ such that $f(n) = \log_2(n)$. Then $f(n) = O(\log_a(n))$, where a is any positive real number.*

Proof. Let $n_0 = 2$, and $c = \frac{1}{\log_a 2}$. We know that $\log_2(n) = \frac{1}{\log_a(2)} \log_a(n)$, and we are done. □

As a direct result of the last example, whenever we state that something is $O(\log n)$ we do so without specifying the base since it clearly does not matter.

3.3 The Fast Fourier Transform

As we had mentioned previously, the Fast Fourier Transform (FFT) is an efficient algorithm to compute the Discrete Fourier Transform of a vector described in the previous chapter. We take

a look at the In this section, we shall first motivate the algorithm, state it formally, and then analyse its running time.

Before starting off though, we very quickly go through the time it will take for us to calculate the DFT of a vector of length n naively. We know that a DFT over a coefficient ring may be represented as an $n \times n$ matrix. When we apply an $n \times n$ matrix to a vector of length n, we multiply the elements of each row to the corresponding components of the vector. Following this, we must also calculate the sum of $n-$terms to get each components of vector obtained by enacting our DFT map. This gives us n^2 total multiplications (each of the n components of the vector is multiplied to n entries in the matrix), and n^2 additions (adding n numbers n times over). Thus, the time taken to calculate the DFT of a vector of length n naively is $O(n^2)$. The FFT algorithm, as we will see, does significantly better than this.

3.3.1 Motivating the Algorithm

Let R be a coefficient ring as described previously, and let n be some power of two. We want to describe an efficient algorithm to compute the Discrete Fourier Transform of a vector $u = (u_0, \ldots, u_{n-1}) \in R^n$. While we may describe the FFT algorithm over R^n even when n is not a power of 2, doing so when n is a power of 2 is easier, and Harvey and van der Hoeven themselves only utilise the algorithm when $n = 2^k$ for some $k \in \mathbb{N}$, so studying these cases suffices for our purposes.

The j^{th}component of $F_\omega(u)$ represented as $(F_\omega(u))_j$ is equal to $\frac{1}{n}\sum_{k=0}^{n-1} \omega^{-jk} u_k$. We simplify this as below:

$$\begin{aligned}
(F_\omega(u))_j &= \frac{1}{n}\sum_{k=0}^{n-1} \omega^{-jk} u_k \\
&= \frac{1}{n}\sum_{k=0}^{n/2-1} \omega^{-jk} u_k + \frac{1}{n}\sum_{k=0}^{n/2-1} \omega^{-j\left(k+\frac{n}{2}\right)} u_{k+\frac{n}{2}} \\
&= \frac{1}{n}\sum_{k=0}^{n/2-1} \omega^{-jk}(u_k + \omega^{-jn/2} u_{k+\frac{n}{2}})
\end{aligned}$$

Now $(\omega^{n/2})^2 = 1$, but $\omega^{n/2} \neq 1$, since ω is a primitive n^{th} root of unity. This means that $\omega^{n/2}$ is that square root of 1 which is not equal to 1 itself. This of course means that $\omega^{n/2} = -1$. What this implies is that $\omega^{-n/2} = (-1)^{-1} = -1$. We may simplify the above expression as:

$$\begin{aligned}
(F_\omega(u))_j &= \frac{1}{n}\sum_{k=0}^{n/2-1} \omega^{-jk}(u_k + (-1)^j u_{k+\frac{n}{2}}) \\
&= \frac{2}{n}\sum_{k=0}^{n/2-1} \omega^{-jk}\frac{(u_k + (-1)^j u_{k+\frac{n}{2}})}{2} \\
&= \frac{1}{n/2}\sum_{k=0}^{n/2-1} \omega^{-jk}\frac{(u_k + (-1)^j u_{k+\frac{n}{2}})}{2}
\end{aligned}$$

From here, we split our analysis into two cases, one where j is even, and one where j is odd, since this has a bearing on how the sum above behaves (There is a $(-1)^j$ term in the sum which is affected by this).

Say $j = 2l$, for some l in $\{0, \ldots, \frac{n}{2} - 1\}$ (this follows from the fact that $j \in \{0, \ldots, n-1\}$, and $j \neq n-1$ since n is a power of two and $n-1$ is odd). Now substituting $2l$ in the place of j above, we get:

$$\begin{aligned}
(F_\omega(u))_{2l} &= \frac{1}{n/2}\sum_{k=0}^{n/2-1} \omega^{-2lk}\frac{(u_k + (-1)^{2l} u_{k+\frac{n}{2}})}{2} \\
&= \frac{1}{n/2}\sum_{k=0}^{n/2-1} \omega^{-2lk}\frac{(u_k + u_{k+\frac{n}{2}})}{2}
\end{aligned}$$

The eagle-eyed reader might notice something very interesting here. We know that ω is the n^{th} root of unity in R. We can use this to show that ω^2 is the $\frac{n}{2}^{th}$ root of unity in R. Firstly, it is very clear that $(\omega^2)^{(n/2)} = 1$. Now we take $(\omega^2)^k$, where $k \in \{1, \ldots, \frac{n}{2} - 1\}$. If $\omega^{2k} = 1$, then ω cannot be the n^{th} root of unity. Why? This is because $2k < n$, and we know that for any natural number $p < n$, $\omega^p \neq 1$.

We note that we have here a sum which appears to take the form of the DFT. We have a multiplier of $\frac{1}{n/2}$ outside the sum, $\frac{n}{2}$ terms inside the sum, each multiplied by a distinct power of the $\frac{n}{2}^{th}$ root of unity in a ring. What does this mean? This means that the $2l^{th}$ term of $F_\omega(u)$ may be given by the l^{th} term of $F_{\omega^2}v$, where $v \in R^{n/2}$ is a vector defined as:

$$v_k = \frac{1}{2}(u_k + u_{k+\frac{n}{2}})$$

We similarly analyse the case when $j = 2l + 1$, for some $l \in \{0, \ldots, \frac{n}{2} - 1\}$. Here,

$$\begin{aligned}(F_\omega(u))_{2l+1} &= \frac{1}{n/2}\sum_{k=0}^{n/2-1}\omega^{-(2l+1)k}\frac{(u_k + (-1)^{(2l+1)}u_{k+\frac{n}{2}})}{2} \\ &= \frac{1}{n/2}\sum_{k=0}^{n/2-1}\omega^{-2lk}\frac{\omega^{-k}(u_k - u_{k+\frac{n}{2}})}{2}\end{aligned}$$

Once again, we see a sum which appears like the Discrete Fourier Transform of a vector in $R^{n/2}$. In other words, we see that the $(2l + 1)^{th}$ term of $F_\omega u$ may be given by the l^{th} term of $F_{\omega^2}w$, where w is a vector defined as:

$$w_k = \frac{1}{2}\omega^{-k}(u_k - u_{k+\frac{n}{2}})$$

Immediately we see that a recursive algorithm pops out of this. As long as we can calculate transforms over $R^{n/2}$, we may calculate transforms over R^n (The significance of n being a power of two is clear now). All that is left for us to do is to understand what happens when $n = 1$, which will be the base case of our recursion.

When $n = 1$, $R^n = R$. The primitive n^{th} root of unity here will be 1 itself. A vector in R has only one component, and the Fourier transform of this vector is the vector itself scaled by the primitive first root of unity, i.e., 1. In other words, when $n = 1$, we just need to return the vector as it is.

Having understood all the necessary steps to formalise the FFT algorithm, we describe a

pseudocode for the algorithm below.

3.3.2 A Pseudocode for the FFT Algorithm

Algorithm 2 FFT(u, ω, n)

Input: A vector $u = (u_0, \ldots, u_{n-1}) \in R^n$; ω - the n^{th} root of unity in R; $n = 2^k$, where k is some non-negative integer.

Output: A vector $z \in R^n$, such that $z = F_\omega(u)$.

```
1: if n = 1 then
2:     return u
3: else
4:     for i in [0, ... n/2 − 1] do
5:         v_i ← (1/2)(u_i + u_{i+n/2})
6:         w_i ← (1/2)ω^{−i}(u_i + u_{i+n/2})
7:     end for
8:     x ← FFT(v, ω^2, n/2)
9:     y ← FFT(w, ω^2, n/2)
10:    for i in [0, ... n/2 − 1] do
11:        z_{2i} ← v_i
12:        z_{2i+1} ← w_i
13:    end for
14: end if
15: return z
```

3.3.3 Analysing the Running Time of the FFT

Before analysing the running time of the algorithm detailed above, we must first return to one of the most important questions of complexity analysis: what defines a step in our analysis? For the moment, we will operate in the **arithmetic complexity** model. What does this mean? It means that our basic arithmetic operations, like addition, subtraction, multiplication and assignment take $O(1)$ or constant time (why constant time is represented as $O(1)$ is something the reader should try figuring out if they are unsure of the notation! I promise that the contents of this chapter are enough to do this).

Now say we have an input of length n. We must first create two vectors of length $\frac{n}{2}$ out of it. We have a for loop the iterates $\frac{n}{2}$ times, and we have two additions and some scaling for every iteration of the for loop. Thus, we take $O(n)$ time to create these two vectors (since both

addition and multiplication take $O(1)$ time, and we do each of these operations about n times in the loop). We then recursively call the FFT procedure, and using a for loop, assign the values l^{th} component of the Fourier Transform of v to the $2l^{th}$ component of z and the l^{th} component of the Fourier Transform of w to the $(2l+1)^{th}$ component of z. Again, since we assumed that assignments take $O(1)$ time, this step takes us $O(n)$ time as well.

Say $T(n)$ is the time it takes for the algorithm to produce an output for an input of length n. It is clear from above that we take $O(n)$ time for various arithmetic operations in the algorithm. We also need to calculate two Fourier Transforms for vectors of length $\frac{n}{2}$. The time it takes for each of these is, by definition, $T\left(\frac{n}{2}\right)$. We are therefore left with the following relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

How do we get $T(n)$ from here, one might ask? It's actually very simple. We note that $O(n)$ may be given by cn for some constant c. This means that our recurrence is of the form $T(n) = 2T\left(\frac{n}{2}\right) + cn$. Now let us consult the diagram below, which is taken from Cormen et al. 2022.

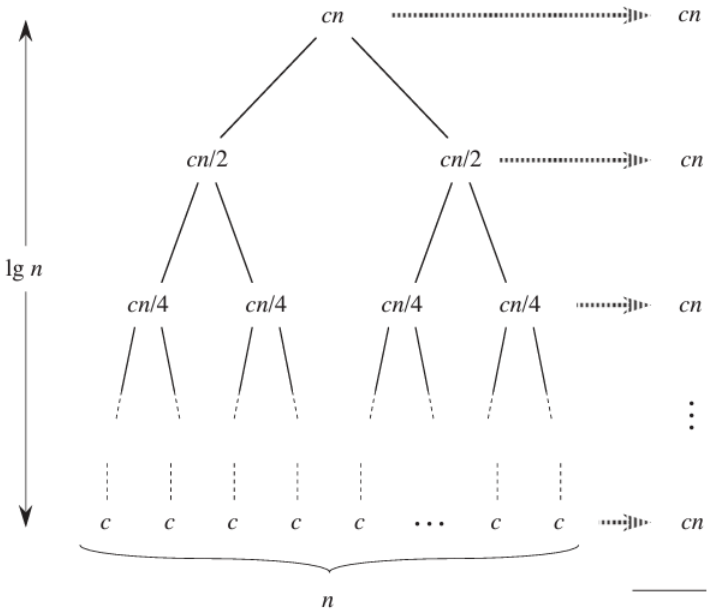

Figure 3.1: A Recursion Tree for the Given Recurrence

Such a figure is referred to as a recursion tree. The total cost of the algorithm may be understood as the sum of the time spent at each node of the tree. We see that when $n = 1$ in our algorithm, all we must do is return the input as it is, so we have a constant cost. Something interesting is happening in the tree though. The sum of the time spent at all nodes on a particular level remains constant at cn across all levels of the tree. The depth of the tree (i.e., the length of the path between the root at cn and the leaves, which have values c) is $\log_2(n)$. The total time taken for the algorithm to run, therefore, is $cn(\log_2(n) + 1)$ (here $\log_2(n) + 1$ represents the height of the tree including the root). This expression evaluates to $cn \log_2(n) + cn$ which is $O(n \log n)$. As we argued before, this is significantly better than the $O(n^2)$ bound we got for the naive computation of the DFT, and the advantage actually increases as n grows larger.

Chapter 4

An Introduction to Tensor Product Spaces

4.1 The Aim of This Chapter

The aim of this chapter is very simple: to give a very brief and concise introduction to tensor product spaces. We note that this particular chapter will not act as a comprehensive guide for tensor products. It is meant to introduce only those concepts that are necessary to understand the working of Harvey and van der Hoeven's algorithm. Keeping this in mind, while tensor product spaces are usually introduced using bilinear maps, we will endeavour to motivate them using multivariate polynomial rings.

4.2 Polynomials in Many Variables - A Brief Discussion

A polynomial in one variable (say x) is an expression of the form $\sum_{i=0}^{n} a_i x^i$, where the a_i's are called coefficients, and come from some ring. n is called the degree of the polynomial. These are concepts that we are all quite familiar with. We now aim to generalise this concept and extend it to understand what it means to have polynomials in multiple variables.

A question the reader might have, for instance, is: why is this generalisation required at all? In the algorithm that Harvey and van der Hoeven have developed, they aim to use Fast Fourier

Transforms in order to multiply polynomials quickly. We note that we had established that the Fast Fourier Transform over R^n may be used to efficiently compute the product of two polynomials in the ring $R[x]/\langle x^n - 1\rangle$. Now let us say we understand what polynomials in multiple variables are. The motivation Harvey and van der Hoeven have is the following: transforms in multivariate polynomial rings may be computed more efficiently than in univariate rings in some cases. In fact, in their concluding remarks in Nussbaumer and Quandalle 1978, Nussbaumer and Quandalle explicitly state this. Of course, their motivations to study multidimensional Fourier transforms differed from Harvey and van der Hoeven, but the point still holds: calculating convolutions in multivariate polynomial rings may be done quickly, and since we aim to minimise the running time of our algorithm, looking at these rings is a good idea.

The previous paragraph raises a lot of questions, most of which we aim to answer in this chapter. Chief among them are of course what these polynomials in multiple variables actually mean, and what it means to calculate transforms over rings of such polynomials.

4.2.1 Understanding Polynomials in Multiple Variables

Consider the expression $x^2 + 3xy - 6xy^2 + 3$. This is a polynomial in two variables, x and y. The maximum degree x has in this expression is 2, and the same goes for y as well. In general then, we may express a polynomial in two variables as follows:

$$P(x, y) = \sum_{i=0}^{n_1} \sum_{j=0}^{n_2} a_{ij} x^i y^j$$

where a_{ij} is the coefficient of the term that is expressed as a product x^i and y^j, n_1 is the maximum degree of x and n_2 is the maximum degree of y.

We generalise this to polynomials in $d-$variables as well, where d is some natural number greater than or equal to 2. Such polynomials may be expressed as:

$$P(x_1, \ldots, x_d) = \sum_{j_1=0}^{n_1} \cdots \sum_{j_d=0}^{n_d} a_{j_1,\ldots,j_d} x_1^{j_1} \ldots x_d^{j_d}$$

where $x_1, \ldots x_d$ are the variables and $n_1, \ldots, n_d$ are the maximum degrees of each of the

variables. As seen above, $a_{j_1,\ldots,j_d}$ is the coefficient of the term $\prod_{i=1}^{d} x_i^{j_i}$.

4.2.2 Understanding Rings of Polynomials in Multiple Variables

Having understood what a polynomial in multiple variables is, we move on to understanding rings of such polynomials. We note that the coefficients of the polynomials that Harvey and van der Hoeven consider are all complex numbers.

Once we had understood polynomials in one variable, something we tried while studying rings was to construct quotient rings of such polynomials. Say we had the ring of all polynomials with complex coefficients, i.e., $\mathbb{C}[x]$. Something we wished to do was to understand the set $\mathbb{C}[x]/\langle x^n - 1\rangle$, i.e., the set of all polynomials with coefficients in $\mathbb{C}$, whose degree is less than n. We noted that this set formed a ring under the operations of polynomial addition, and multiplication, where given two polynomials $p = \sum_{i=0}^{n-1} a_i x^i$ and $q = \sum_{j=0}^{n-1} b_i x^i$:

$$\begin{aligned} p+q &= \sum_{i=0}^{n-1} (a_i + b_i) x^i \\ pq &= \sum_{i=0}^{n-1} x^i \left(\sum_{k=0}^{n-1} a_k b_{i-k} \right) \end{aligned}$$

(**Note:** We note that as mentioned in the previous chapter, b_{i-k} is the coefficient of the term $x^{i-k \mod n}$ in q).

We extend this definition very neatly to polynomials of multiple variables. Let $\mathbb{C}[x_1, \ldots, x_d]$ be the set of polynomials in d variables. This forms a commutative ring with unity (the constant polynomial 1 is the multiplicative identity element) under the extensions of the operations of addition and multiplication seen on $\mathbb{C}[x]$. What do we mean by this? Given any two polynomials, their sum is the polynomial whose coefficients are the sum of coefficients of corresponding terms in both polynomials. Given two polynomials of the form $x_1^{k_1} \ldots x_d^{k_d}$ and $x_1^{j_1} \ldots x_d^{j_d}$, their product is $x_1^{k_1+j_1} \ldots x_d^{k_d+j_d}$ (this definition may be easily extended to any general polynomial in the ring).

We now consider $\mathbb{C}[x_1, \ldots, x_d]/\left\langle x_1^{k_1} - 1, \ldots, x_d^{k_d} - 1 \right\rangle$. Let p and q be elements of the afore-

mentioned ring, such that $p = \sum_{j_1=0}^{n_1-1} \cdots \sum_{j_d=0}^{n_d-1} a_{j_1,\ldots,j_d} x_1^{j_1} \ldots x_d^{j_d}$ and $q = \sum_{j_1=0}^{n_1-1} \cdots \sum_{j_d=0}^{n_d-1} b_{j_1,\ldots,j_d} x_1^{j_1} \ldots x_d^{j_d}$. We see that $\mathbb{C}[x_1, \ldots, x_d] / \langle x_1^{n_1} - 1, \ldots, x_d^{n_d} - 1 \rangle$ forms a commutative ring with unity (again, the multiplicative identity element is the constant polynomial 1) under the following operations:

$$\begin{aligned} p + q &= \sum_{j_1=0}^{n_1-1} \cdots \sum_{j_d=0}^{n_d-1} (a_{j_1,\ldots,j_d} + b_{j_1,\ldots,j_d}) x_1^{j_1} \ldots x_d^{j_d} \\ pq &= \sum_{j_1=0}^{n_1-1} \cdots \sum_{j_d=0}^{n_d-1} x_1^{j_1} \ldots x_d^{j_d} \left(\sum_{k_1=0}^{n_1-1} \cdots \sum_{k_d=0}^{n_d-1} a_{k_1,\ldots,k_d} b_{j_1-k_1,\ldots,j_d-k_d} \right) \end{aligned}$$

4.3 Introducing Tensors

4.3.1 The Vector Space Structure of Multivariate Polynomial Rings

The reader might well wonder, having read through the previous chapter and the beginning of this one: Does the ring above not form a vector space over $\mathbb{C}$? This was the case with $\mathbb{C}[x] / \langle x^n - 1 \rangle$, so it is natural to question if this is the case here as well. And indeed it is. Trying to prove that it is a vector space over $\mathbb{C}$ is not a very difficult exercise. Closure, associativity, commutativity and the existence of an additive identity and an additive inverse for every element are all guaranteed by the ring structure. The proofs of properties involving scalar multiplication by complex numbers are not very difficult and are identical to the proofs we have seen for the same results for rings of polynomials in one variable.

In fact, we can come up with an explicit basis for the above ring when treated as a vector space over $\mathbb{C}$, which we do below.

Let us look at the ring $\mathbb{C}[x_1, \ldots, x_d] / \langle x_1^{n_1} - 1, \ldots, x_d^{n_d} - 1 \rangle$. An element p in this ring is given by $p = \sum_{j_1=0}^{n_1-1} \cdots \sum_{j_d=0}^{n_d-1} a_{j_1,\ldots,j_d} x_1^{j_1} \ldots x_d^{j_d}$. We see for every choice of $j_1, \ldots, j_d$ that $a_{j_1,\ldots,j_d}$ is a complex number. It is easy to see therefore that the set is spanned by the set:

$$B = \{x_1^{k_1} \ldots x_d^{k_d} \mid 0 \leq k_1 < n_1; \ldots; 0 \leq k_d < n_d\}$$

Seeing that this set is a minimal spanning set is also not very difficult (try it if you're not convinced!). Therefore, this set forms a basis for the vector space $\mathbb{C}[x_1, \ldots, x_d]/\left\langle x_1^{n_1} - 1, \ldots, x_d^{n_d} - 1 \right\rangle$. This also means that $\mathbb{C}[x_1, \ldots, x_d]/\left\langle x_1^{n_1} - 1, \ldots, x_d^{n_d} - 1 \right\rangle$ forms a $\mathbb{C}-$algebra.

We note that we have seen that $\mathbb{C}[x_1, \ldots, x_d]/\left\langle x_1^{n_1} - 1, \ldots, x_d^{n_d} - 1 \right\rangle$ forms a vector space over $\mathbb{C}$, and we have specified a basis. But what is the dimension of this vector space? It is immediately clear that this forms a finite-dimensional vector space over $\mathbb{C}$. We see that the basis we specified above consists of elements of the form $x_1^{j_1} \ldots x_d^{j_d}$, where $0 \leq j_i < n_i$. This means that there are a total of $\prod_{i=1}^{d} n_i$ elements in the set, which is also the dimension of $\mathbb{C}[x_1, \ldots, x_d]/\left\langle x_1^{n_1} - 1, \ldots, x_d^{n_d} - 1 \right\rangle$ as a vector space over $\mathbb{C}$.

4.3.2 The Tensor Product

We now introduce a concept known as the tensor product. Let V and W be two finite-dimensional vector spaces over a field $\mathbb{F}$. Let $\dim(V) = m$, and $\dim(W) = n$. We now define the tensor product of V and W, denoted by $V \otimes W$. $V \otimes W$ is a vector space over $\mathbb{F}$ which has dimension mn.

Let $B_V = (v_0, \ldots, v_{m-1})$ be an ordered basis for the vector space V. Let $B_W = (w_0, \ldots, w_{n-1})$ be an ordered basis for W. Then, we define $V \otimes W$ to be the vector space that consists of all formal linear combinations of expressions of the form $v_i \otimes w_j$, where $1 \leq i \leq m$, $1 \leq j \leq n$.

A natural question to ask here is what is meant by formal linear combinations. And what exactly does $v_i \otimes w_j$ even represent? The answers to both questions are tied together. We answer the second one first. Think of $\otimes$ in the expression $v_i \otimes w_j$ as being analogous to multiplying xy in a multivariate polynomial ring. What exactly it represents is not something that we need to pin down exactly. It just represents the tensor product of two vectors, whatever that is, just as xy represents the product of two elements in a multivariate polynomial ring, whatever that is.

Now we try and understand what formal linear combinations are. Consider the expression $v_0 \otimes w_0 + v_1 \otimes w_1$. This represents the sum of two tensors in the tensor space, just as $xy + x^2y^3$ represents the sum of polynomials in a polynomial ring. Again, one can completely concretise this understanding, but we don't need to. What we will be seeing below is that tensor product

spaces can be made to behave exactly like multivariate polynomial rings. Therefore, in order to understand the former, it is enough to understand the latter, which we already have. We explicitly describe the relationship between the two below.

4.4 Isomorphisms with Multivariate Polynomial Rings

Let us start off with an example. Consider the ring $\mathbb{C}[x,y]/\left\langle x^2-1, y^3-1\right\rangle$. We know that this also forms a vector space over $\mathbb{C}$, and has the basis $\{1, x, y, y^2, xy, xy^2\}$. We consider the vector spaces $\mathbb{C}^2$ and $\mathbb{C}^3$, which have the following ordered bases: $B_1 = (v_0, v_1) = ((0,1),(1,0))$ and $B_2 = (w_0, w_1, w_2) = ((0,0,1),(0,1,0),(1,0,0))$.

Now by definition, $\mathbb{C}^2 \otimes \mathbb{C}^3$ is the space spanned by the set $B = ((0,1)\otimes(0,0,1),(1,0)\otimes(0,0,1),(0,1)\otimes(0,1,0),(0,1)\otimes(1,0,0),(1,0)\otimes(0,1,0),(1,0)\otimes(1,0,0))$. Clearly, since the cardinality of both basis sets are equal, we may define a bijection from one to the other. Let this bijection be the one that sends x^iy^j to $v_i \otimes w_j$. Then, since we had defined $\mathbb{C}^2 \otimes \mathbb{C}^3$ to be the set spanned by B, we see that $\mathbb{C}^2 \otimes \mathbb{C}^3 \cong \mathbb{C}[x,y]/\left\langle x^2-1, y^3-1\right\rangle$.

However, we note that $\mathbb{C}[x,y]/\left\langle x^2-1, y^3-1\right\rangle$ is a commutative ring with unity as well. This means that since we have a very natural vector space isomorphism between $\mathbb{C}[x,y]/\left\langle x^2-1, y^3-1\right\rangle$ and $\mathbb{C}^2 \otimes \mathbb{C}^3$, we may induce the same ring structure over it as well. We define operations of addition and multiplication on $\mathbb{C}^2 \otimes \mathbb{C}^3$ as below.

Given two tensors $\sum_{i=0}^{1}\sum_{j=0}^{2} a_{i,j}v_i\otimes w_j$ and $\sum_{i=0}^{1}\sum_{j=0}^{2} b_{i,j}v_i\otimes w_j$, their sum is $\sum_{i=0}^{1}\sum_{j=0}^{2}(a_{i,j}+b_{i,j})v_i\otimes w_j$. Their product is the tensor given by $\sum_{i=0}^{1}\sum_{j=0}^{2} v_i\otimes w_j\left(\sum_{k=0}^{1}\sum_{l=0}^{2} a_{k,l}b_{i-k,j-l}\right)$. Since the operations are defined exactly as they were on the multivariate polynomial rings, we see that $\mathbb{C}^2\otimes\mathbb{C}^3$ forms a commutative ring with unity as well, where the multiplicative identity $v_0 \otimes w_0$ is the image of the constant polynomial 1 under our vector space isomorphism.

More generally, therefore, we see that the multivariate polynomial ring

$$\mathbb{C}[x_1,\ldots,x_d]/\left\langle x_1^{n_1}-1,\ldots,x_d^{n_d}-1\right\rangle$$

is isomorphic as an algebra to $\otimes_{i=1}^{d}\mathbb{C}^{n_i}$ (this is just convenient notation to represent $\mathbb{C}^{n_1}\otimes$

$\cdots \otimes \mathbb{C}^{n_d}$). This is a result we will use heavily in our analysis of Harvey and van der Hoeven's algorithm.

4.5 Defining Discrete Fourier Transforms on Tensor Product Spaces

Let R be a coefficient ring as described in the previous chapter. Let this be a vector space of dimension R over $\mathbb{C}$. Then, R^n is a vector space of dimension kr over $\mathbb{C}$, and $\otimes_{i=1}^{d} R^{n_i}$ is a vector space of dimension $r \prod_{i=1}^{d} n_i$ over $\mathbb{C}$. We represent vectors in $\mathbb{R}^n$ as $n-$tuples of the form $(u_0, \ldots, u_{n-1})$, where $u_j \in R$, for every value of j. Similarly, a vector $u \in \otimes_{i=1}^{d} R^{n_i}$ is denoted by its coordinates stored in dictionary order as $(u_{0,\ldots,0}, u_{0,\ldots,1}, \ldots, u_{n_1-1,\ldots,n_d-1})$, where $u_{j_1,\ldots,j_d} \in R$, $0 \leq j_i < n_i$.

Let $\omega_1, \ldots, \omega_d$ be primitive roots of unity of order $n_1, \ldots, n_d$ respectively in R. Then, the $(j_1, \ldots, j_d)^{th}$ coordinate of the tensor obtained when the Fourier map is implemented on $u \in \otimes_{i=1}^{d} R^{n_i}$ is given by:

$$(\mathcal{F}_{\omega_1,\ldots,\omega_d} u)_{j_1,\ldots,j_d} = \frac{1}{n_1 \cdots n_d} \sum_{k_1=0}^{n_1-1} \cdots \sum_{k_d=0}^{n_d-1} \omega_1^{-j_1 k_1} \cdots \omega_d^{-j_d k_d} u_{k_1,\ldots,k_d}$$

There is an analogue of the convolution theorem for general tensor product spaces. Given u, v in $\otimes_{i=1}^{d} R^{n_i}$, we define $u * v$ as below:

$$(u * v)_{j_1,\ldots,j_d} = \sum_{k_1=0}^{n_1-1} \cdots \sum_{k_d=0}^{n_d-1} u_{k_1,\ldots,k_d} v_{j_1-k_1,\ldots,j_d-k_d}$$

We define pointwise products in $\otimes_{i=1}^{d} R^{n_i}$ in the same way we did for R^n:

$$(u \cdot v)_{j_1,\ldots,j_d} = u_{j_1,\ldots,j_d} \cdot v_{j_1,\ldots,j_d}$$

Then we have:

$$\frac{1}{n_1 \cdots n_d} u * v = n_1 \cdots n_d \mathcal{F}_{\omega_1^{-1},\ldots,\omega_d^{-1}} \left(\mathcal{F}_{\omega_1,\ldots,\omega_d} u \cdot \mathcal{F}_{\omega_1,\ldots,\omega_d} v \right)$$

We omit the proof of this generalised version of the convolution theorem simply because it works exactly the same way as the proof we saw earlier.

Having established the concepts we need in order to understand Harvey and van der Hoeven's algorithm, we now take a tour through their paper.

Chapter 5

Walking Through Harvey and van der Hoeven's Paper

5.1 The Aim of This Chapter

The aim of this chapter is very straightforward: here, we provide the necessary tools in order to understand the multiplication algorithm developed by David Harvey and Joris van der Hoeven in 2019. It is important to note that we will not be providing proofs of some of the major theorems established by Harvey and van der Hoeven in this chapter, for the simple reason that these proofs have been explained in an extraordinarily accessible fashion in the paper itself. The goal here is to try and fill in the gaps between the proofs in order to make the process of understanding the paper easier for undergraduate students. Having said that, we start off this chapter with a very brief description of the algorithm itself.

Note: Before we begin, we establish that we operate in the bit complexity model for the rest of our discussion. We explain this with the help of an example. In the arithmetic complexity model, we saw that adding two numbers was a 'basic operation' that took $O(1)$ time. Here, that isn't the case. What does represent a basic operation is the addition of one-bit numbers, i.e., 1s and 0s. Adding numbers such as 11 and 01 takes two such 'bit-operations', and as a result, adding $n-$bit numbers is not an operation that takes constant time, but one that takes linear

time (i.e., $O(n)$ time). This is a difference the reader should be mindful of.

5.2 The Algorithm Itself

Harvey and van der Hoeven do not provide just one algorithm: they describe a family of algorithms parametrised by a dimension factor $d \geq 2$. We won't discuss what exactly d is here (refer to the choosing parameters section of this chapter if you're interested in learning about this, that's linked here), but one should develop an intuitive picture in just a moment. In order to use the algorithm with dimension factor d, we must be multiplying numbers of size $n-$bits, where $n \geq 2^{d^{12}}$. (Note: by number of bits we mean the 'length' of the number itself, when expressed in binary. A bit is a binary digit. For instance, the number 1101 written has four bits, but has a value equal to 13 (the number is expressed in decimal here). It is clear that any number with $n-$bits can have value at most $2^n - 1$.). For numbers of smaller sizes, the authors say that any convenient multiplication algorithm may be utilised.

5.2.1 Multiplying Numbers is (Almost) Equivalent to Multiplying Polynomials

Let us define two quantities, which will help us later on. Let $b := \lceil \log_2(n) \rceil$. Let $N := \lceil \frac{n}{b} \rceil$. We now want to divide our $n-$bit numbers into N chunks of size b. Each of these N chunks then forms the coefficient of a polynomial of degree $N-1$, which when evaluated at 2^b gives us back our numbers.

There were a lot of things established in the previous paragraph, so let's try and see how this works with an example. Consider the number 1101. Here, $n = 4$, $b = N = 2$. The claim is that the polynomial $11x + 01$ evaluated at $x = 2^2$ gives us back 1101. (A small but important note: Things can get quite tricky here, since 2^2 is being expressed in decimal notation, while the coefficients are expressed in binary. We shall not run into such difficulties later, since it will be obvious which base we are using. For the purposes of this exercise though, we denote the base of a number by attaching it as a subscript to the number itself. For instance, 2_{10} denotes the number 2 represented in decimal, while 10_2 denotes the same number represented in binary).

We see that indeed this is true. $11_2(2^2_{10}) + 01_2 = 11_2(10^2_2) + 01_2 = 1100_2 + 01_2 = 1101_2$. The same is true in general for any n bit number as well.

We have therefore expressed our $n-$bit numbers in terms of polynomials of degree $N-1$. We claim that multiplying these two integers is the same as multiplying the two polynomials we have and then evaluating the product polynomial at 2^b. It is not difficult to see why this claim is true. Given any two polynomials, we know that if we evaluate the polynomials at some value k and then multiply the numbers so obtained, or we evaluate the product of these two polynomials at k, we will get the same answer. This is the principle that is being utilised here. All we are doing is setting $k = 2^b$. Since we know that evaluating the original polynomials at 2^b gives us the integers we started with, evaluating the product polynomial at 2^b must give us the integer product we are looking for.

5.2.2 Evaluating Polynomials at a Particular Value is a Quick Operation - So We Let it Be.

From above, it is clear that our algorithm consists of two parts: calculating the polynomial product, and evaluating the resultant polynomial at 2^b. The paper mainly deals with the former step. We see that the product polynomial would have degree $2N-2$, or $O(N)$, with each coefficient in the product polynomial being a sum of at most N terms of length $2b$.

How do we get here? Think of it this way. Each of our coefficients have $b-$bits. This means that each one of them is lesser than 2^b, which is the smallest number that has $b+1$ bits. This means that the product of any two such coefficients must be less than $2^b \cdot 2^b = 2^2b$. This in turn means that each of these terms has at most $2b$ bits.

The coefficients of the product polynomials are the sum of such terms. There are at most N such terms in a coefficient (this is the case for the coefficient of x^{N-1}, and the intuition behind this comes from the binomial theorem, which we assume the reader to be familiar with). The value of each coefficient of the product polynomial is thus less than $2^{2b} * N < 2^{3b}$ (This follows since $N < n \leq 2^b$). Since each coefficient is lesser in value that 2^{3b}, we see that we would only need $O(b)$ bits to store each coefficient. The degree of our polynomial is $O(N)$, and the value of

the largest coefficient is $O(b)$, which means that the time taken to evaluate this polynomial at 2^b is $O(bN) = O\left(\log_2(n) \cdot \frac{n}{\log_2(n)}\right) = O(n)$.

The main argument of the above paragraph can be summarised as follows: evaluation of the product polynomial at 2^b does not really take us too much time. Therefore, we should spend our energy on bringing down the time taken to actually calculate this product, which is what Harvey and van der Hoeven have done.

5.2.3 Why Stop at Polynomials in One Variable?

Since our polynomial product has a degree of $2N-2$, we would not lose any information about the product itself if we calculated it in the ring $\mathbb{Z}[x]/\left\langle x^S - 1\right\rangle$, where S is some positive integer greater than $2N-2$. We now impose another condition on S. Let us say S is a product of d primes, $s_1, \ldots s_d$, each unique, all greater than 2. Then, using the Chinese remainder theorem, we know that there is an isomorphism between the rings $\mathbb{Z}[x]/\left\langle x^S - 1\right\rangle$ and $\mathbb{Z}[x_1, x_2, \ldots, x_d]/\left\langle x_1^{s_1} - 1, x_2^{s_2} - 1, \ldots, x_d^{s_d} - 1\right\rangle$. It is clear that the first ring has polynomials with S terms. In the second ring, the polynomials have terms of the form $x_1^{k_1} \ldots x_d^{k_d}$, where $k_i \in \{0, 1, \ldots, s_i - 1\}$. From this, it is clear to us that there must be $s_1 s_2 \ldots s_d = S$ terms in the polynomials of the second ring as well. We note that the isomorphism between the rings sends x to $x_1 x_2 \ldots x_d$. That each distinct power of x goes to a unique term of the form $x_1^{k_1} \ldots x_d^{k_d}$ is a direct consequence of the Chinese Remainder Theorem, and a formal proof of this fact is omitted here. We utilise this isomorphism to study the product of the images of our two polynomials in the second ring.

5.2.4 Speed is Key, and Recursion is Fast. Let's get in some Powers of Two

What do we have so far? We've taken two $n-$bit integers, expressed them as polynomials in one variable of degree N, and then expressed them as polynomials in $d-$variables. We still need to figure out a way to multiply them efficiently though. We do so as follows. Firstly, we treat $\mathbb{Z}[x_1, x_2, \ldots, x_d]/\left\langle x_1^{s_1} - 1, x_2^{s_2} - 1, \ldots, x_d^{s_d} - 1\right\rangle$ as a subring of the following ring:

$$\mathbb{C}[x_1, x_2, \ldots, x_d] / \left\langle x_1^{s_1} - 1, x_2^{s_2} - 1, \ldots, x_d^{s_d} - 1 \right\rangle$$

.

We know from the section on tensors (see here for more details on the same) that

$$\left(\mathbb{C}[x_1, x_2, \ldots, x_d] / \left\langle x_1^{s_1} - 1, x_2^{s_2} - 1, \ldots, x_d^{s_d} - 1 \right\rangle, +, \times\right) \cong \left(\otimes_{i=1}^{d} \mathbb{C}^{s_i}, +, *\right)$$

where $*$ represents the convolution of two vectors in $\otimes_{i=1}^{d} \mathbb{C}^{s_i}$. Our aim is to utilise this isomorphism, and the Fast Fourier Transform in order to compute this convolution quickly. Harvey and van der Hoeven's main innovation comes up in the next step, however. Instead of using the Fast Fourier Transform map directly over $\otimes_{i=1}^{d} \mathbb{C}^{s_i}$, they have shown that calculating this is equivalent to sending the vectors in $\otimes_{i=1}^{d} \mathbb{C}^{s_i}$ to a slightly 'larger' space $\otimes_{i=1}^{d} \mathbb{C}^{t_i}$, computing the Fourier Transform over this space, and mapping the vector obtained back to the space $\otimes_{i=1}^{d} \mathbb{C}^{s_i}$. What exactly the space $\otimes_{i=1}^{d} \mathbb{C}^{t_i}$ is is a good question to ask at this juncture. Harvey and van der Hoeven define t_i to be 2^k for some natural number k, such that $t_i > s_i \ \forall \ i$. They show that if t_i and s_i are chosen appropriately, one may define maps $\mathcal{A} : \otimes_{i=1}^{d} \mathbb{C}^{s_i} \to \otimes_{i=1}^{d} \mathbb{C}^{t_i}$ and $\mathcal{B} : \otimes_{i=1}^{d} \mathbb{C}^{t_i} \to \otimes_{i=1}^{d} \mathbb{C}^{s_i}$ such that the Fourier Transform $\mathcal{F}_{s_1,\ldots,s_d}$ over $\otimes_{i=1}^{d} \mathbb{C}^{s_i}$ is equal to $c\mathcal{B}\mathcal{F}_{t_1,\ldots,t_d}\mathcal{A}$, where $\mathcal{F}_{t_1,\ldots,t_d}$ is the Fourier Transform map over $\otimes_{i=1}^{d} \mathbb{C}^{t_i}$. c here is a constant chosen appropriately (**Note:** We have deliberately been a little vague about the nature of t_i, s_i and c here. The exact choices that the authors make will be established clearly in the choosing parameters section. Make your way over there to understand this better).

The reason behind the choice of t_i being powers of two is, unsurprisingly, to ensure that we can calculate this Fourier Transform recursively. Recursion in any algorithm often helps us calculate quantities quickly (provided it is not wasteful, of course), and the same principle applies here. Head over here to find out how they implement a recursive algorithm to calculate the Fourier Transform specified above.

5.2.5 Going Over the Algorithm Again - With Some Questions to Think Over for Later

Finally, we are in a position to describe the major steps of the algorithm. Given two $n-$bit numbers, we first express them as polynomials of degree $N-1$ in one variable. We know that integer multiplication is equivalent to polynomial multiplication followed by evaluation (which is quick anyway), so it suffices for us to minimise the time we use to multiply two polynomials. We calculate this polynomial product modulo $x^S - 1$, where S is a product of d distinct primes $s_1, \ldots, s_d$ and $S > 2N - 2$. We then use the Chinese Remainder Theorem to express our polynomials as polynomials of $d-$variables in the ring $\mathbb{Z}[x_1, x_2, \ldots, x_d]/ < x_1^{s_1} - 1, x_2^{s_2} - 1, \ldots, x_d^{s_d} - 1 >$, which we treat as a sub-ring of $\mathbb{C}[x_1, x_2, \ldots, x_d]/ < x_1^{s_1} - 1, x_2^{s_2} - 1, \ldots, x_d^{s_d} - 1 >$. We know that this is isomorphic to $\otimes_{i=1}^{d} \mathbb{C}^{s_i}$, so we want to now figure out a way to calculate convolutions in the latter space quickly. We do so by sending the vectors in this space to $\otimes_{i=1}^{d} \mathbb{C}^{t_i}$, computing the convolution there, and sending the image back. Having obtained the image in $\otimes_{i=1}^{d} \mathbb{C}^{s_i}$, we then express it as a polynomial in one variable using our first isomorphism, and evaluate it at 2^b in order to get our integer product.

There are two things that remain unclear though. Firstly, operating in the complex numbers involves using arbitrarily precise arithmetic, something computers do not and will never have. So, we must approximate quantities in these spaces. This leads to an error though, and in order for this algorithm to work, we would need to keep the error low enough that eventually we know that we obtain a unique and correct answer.

Let us spend a little more time looking carefully at this question. Why do we know that if we keep the error bound low enough, we will get a correct and unique answer? The reasoning here is simple. We see that our polynomials have integer coefficients. This means that the polynomial products themselves must also have integer coefficients. While we calculate Fourier Transforms using approximations of elements in vector spaces over the complex numbers, we induce an error in our computations, but we only need this error (i.e., the maximum difference between the computed coefficient and the actual coefficient) to be less than $\frac{1}{2}$ in magnitude. If we ensure that this is the case, then we know that since any two integers differ by at least one,

for every coefficient that we have computed, there exists a unique integer that this coefficient is closest to. This unique integer is the actual coefficient of some term in the product polynomial. Harvey and van der Hoeven select parameters to ensure that the magnitude of the error in each coefficient does not exceed $\frac{1}{4}$ (see this for more details), so we can indeed conclude that this algorithm always gives us a correct and unique answer.

Secondly, we've just described an incredibly complicated and intricate algorithm. How do we know it runs in the time that we claim it does at all? Both these questions are important, and form the essence of the next sections of this chapter. To start off your journey on error analysis, and basic complexity bounds, you're in the right place! Read on. Want to find out what $\mathcal{A}$ and $\mathcal{B}$ are? Go here. Looking for a place to study how we calculate the transforms on $\otimes_{i=1}^{d}\mathbb{C}^{t_i}$? Make your way over here, and you'll find out. Lastly, want to see what those parameters are and understand the algorithm precisely? Go here. I hope you enjoy reading about this beautiful algorithm as much as I loved learning about it.

5.3 Approximating Complex Numbers and Other Basic Operations

We had noted in the previous section that working in the complex numbers necessitates making approximations. Of course making approximations induces errors in our working, and therefore, we need to find a way to ensure that these errors are low enough that we get an accurate answer. Before we get to that analysis however, it is important to establish a method of approximating complex numbers, so that we may appropriately define errors in them. The beginning of this section deals with procedures for doing exactly this. In the latter half, we establish some operations on complex vector spaces that we will need for performing various steps in the algorithm. At a glance, much of the discussion in this section (which is a reiteration of the results Harvey and van der Hoeven describe in section 2 of their paper) might seem like a formality when compared to the real meat of the paper itself, but as we found while reading the paper, these results are imperative in establishing bounds for every other major result of the paper.

5.3.1 Integer Arithmetic - Some Basic Assumptions

Harvey and van der Hoeven start off by stating some assumptions about basic arithmetic operations on **integers**. Let us say we have two integers a and b, which are p bits long (i.e., $|a|$, $|b|$ are both strictly less than 2^p), where $p \geq 1$. For the rest of the paper, the following are assumed:

- Computing $a + b$ and $a - b$ takes $O(p)$ bit operations (it is worth noting that we operate in the bit complexity model here. This should not be a surprise, for operating in the arithmetic complexity model negates the whole point of the paper).
- Computing ab requires $O(p^{1+\delta})$ bit operations. δ here is a small constant, and in order to use it concretely when we prove things later on, we assume $\delta < \frac{1}{8}$.
- Computing the quantities $\left\lfloor \frac{a}{b} \right\rfloor$ and $\left\lceil \frac{a}{b} \right\rceil$ takes $O(p^{1+\delta})$ bit operations.
- Given two integers x, y such that $x > 0$ and $y > 0$, we may compute $\left(\left\lfloor \frac{a}{b} \right\rfloor\right)^{x/y}$ and $\left(\left\lceil \frac{a}{b} \right\rceil\right)^{x/y}$ in $O(p^{1+\delta})$ bit operations.

5.3.2 Approximating Complex Numbers

An integral part of approximating any quantity is to understand how precise this approximation actually is. This helps us measure the error in our approximation, and keep track of it as we perform various operations on the quantity in question. Therefore, we fix what is called a precision parameter, henceforth referred to as p. In Harvey and van der Hoeven's paper, they assume $p \geq 100$, and we do the same here.

Consider the complex unit disc $\mathbb{C}_0$, which is the set of all vectors on the complex plane whose modulus is less than or equal to 1. Let $\mathbb{Z}[i]$ denote the set of Gaussian integers, and let $2^{-p}\mathbb{Z}[i]$ denote $\{2^{-p}(x + iy) \mid x + iy \in \mathbb{Z}[i]\}$. We then consider:

$$\tilde{\mathbb{C}_0} := 2^{-p}\mathbb{Z}[i] \cap \mathbb{C}_0$$

We note that every element of this set is of the form $2^{-p}(x + iy)$, where $x + iy$ is a Gaussian integer with modulus lesser than or equal to 2^p. Why? It is clear that every element of this

set must have a modulus less than or equal to 1 (since we are looking at a set that contains elements belonging to the complex unit disc). This means that it consists of Gaussian integers $x + iy$ such that $|2^{-p}(x + iy)| \leq 1$. This in turn gives us $|x + iy| \leq 2^p$.

It is important to understand how elements of $\tilde{\mathbb{C}}_0$ would be stored. Harvey and van der Hoeven say that elements of the approximation of the complex unit disc would always be stored as a tuple (x, y), where $|x| \leq 2^p$ and $|y| \leq 2^p$ (If this was not the case, and say $|x| > 2^p$, then $|x + iy| > 2^p$, and such a number lies outside our set). Storing these numbers therefore takes $O(p)$ bits, and the precision parameter is always known, so it need not be stored along with the tuple itself.

We now define what is known as the round towards zero function. Consider $\rho_0 : \mathbb{R} \to \mathbb{Z}$ as follows:

$$\rho_0(x) = \begin{cases} \lfloor x \rfloor & \text{if } x \geq 0 \\ \lceil x \rceil & \text{if } x < 0 \end{cases}$$

Why this function is called a round towards zero function should not be very hard to see, we round upwards in case of negative numbers, and downwards in case of non-negative numbers. We then define $\rho' : \mathbb{C} \to \mathbb{Z}[i]$ as follows:

$$\rho'(x + iy) = \rho_0(x) + i\rho_0(y)$$

From the definition of ρ_0, it is clear that $|\rho_0(x)| \leq |x|$ and $|\rho_0(y)| \leq y$. This in turn means that $(\rho_0(x))^2 \leq x^2$, for any real x. From here, we can see that $(\rho_0(x))^2 + (\rho_0(y))^2 \leq x^2 + y^2$, which means that $|\rho'(z)| \leq |z|$, for any complex number z. We know that $|\lfloor x \rfloor - x| < 1$, and that $|\lceil x \rceil - x| < 1$, for every real number x. This means that $|\rho'(z) - z| = \sqrt{(\rho_0(x) - x)^2 + (\rho_0(y) - y)^2} < \sqrt{1^2 + 1^2} = \sqrt{2}$ for any complex number z.

Finally, we define the approximating function $\rho : \mathbb{C} \to 2^{-p}\mathbb{Z}[i]$ as follows:

$$\rho(z) = 2^{-p}\rho'(2^p z)$$

Let us make sense of what this map does. We first shift both the real and imaginary parts of a complex number to the left by p places (think of multiplying by 2 when you store in binary as being equivalent to multiplying by ten when we store in decimal). We then round the numbers we get towards zero and shift them p bits back. When we represent real numbers in decimal notation, we know that they may have a non-terminating decimal expansion. This is true even when we represent real numbers in binary notation. The equivalent of a decimal point when we express a real number in binary is called the binary point. Since a complex number may be thought of as a tuple of real numbers, this same principle of non-terminating binary expansions applies here as well. What our function is doing here is approximating a complex number, with the error arriving at the p^{th} bit after the binary point or later. In essence, we obtain an approximation of the given complex number whose real and imaginary parts differ from those of the original complex number by most 2^{-p}. It is clear that $|\rho(z)| \leq |z|$. Furthermore, since the error in each part of the complex number occurs at the p^{th} bit after the binary point, it is also evident that $|\rho(z) - z| < \sqrt{2} \cdot 2^{-p}$.

We note here that because of the way we have defined ρ, $\rho(\mathbb{C}_0) \subset \tilde{\mathbb{C}}_0$. We quickly establish why this is the case. Let $z \in \mathbb{C}_0$. Then, $\rho(z) \in \mathbb{C}_0$, because $|\rho(z)| \leq |z|$. This means that $|\rho(z)| \leq 1$. We know that $\rho(z)$ must have at most p bits after the binary point, because all the other bits are eliminated when we round $2^p z$ towards 0. This means that $2^p(\rho(z)) \in \mathbb{Z}[i]$. Furthermore, since $|\rho(z)| \leq 1$, $|2^p \rho(z)| \leq 2^p$, which in turn means that $\rho(z) \in \tilde{\mathbb{C}}_0$.

5.3.3 Approximating Vector Spaces over $\mathbb{C}$

We recall that approximating vector spaces over $\mathbb{C}$ will help us since the Fourier maps we require for our algorithm will be defined over precisely such vector spaces. More specifically, they will be defined over finite dimensional vector spaces over $\mathbb{C}$. These vector spaces are assumed to have a 'privileged' choice of basis. This means that there is a canonical basis B_V for the vector space V, where $B_V = \{b_0, \ldots, b_{m-1}\}$ (the dimension of V is assumed to be some positive integer m). For instance, we know that $\mathbb{C}$ forms a vector space over itself. $B_{\mathbb{C}} = \{1\}$.

The definition of a norm on these vector spaces that Harvey and van der Hoeven use is

quite easy to understand, but it differs from the Euclidean norm that we are accustomed to, so it is worth going through it once. Given a vector $v \in V$, such that $v = \sum_{i=0}^{m-1} \lambda_i b_i$, $\|v\| := \max\{|\lambda_1|, \ldots, |\lambda_{m-1}|\}$. That this definition satisfies all the properties we require a norm to satisfy is a good exercise to try out if one is not convinced that this is indeed a norm, but we shall not be attempting that here (It is worth noting here that this definition of a norm on vector spaces is natural and common, and is referred to as the **sup norm**).

We extend our approximation procedure to all finite dimensional vector spaces over $\mathbb{C}$. We define the map $\rho : V \to V$ as $\rho(v) = \rho\left(\sum_{i=0}^{m-1} \lambda_i b_i\right) = \sum_{i=0}^{m-1} \rho(\lambda_i) b_i$. It follows immediately from the definition of ρ and the norm we have just defined on finite dimensional vector spaces over $\mathbb{C}$ that $\|\rho(v)\| \leq \|v\|$, and that $\|\rho(v) - v\| < \sqrt{2} \cdot 2^{-p}$, for any $v \in V$.

Intuitively, the unit ball in such vector spaces V is defined as $V_0 := \left\{ \sum_{i=0}^{m-1} \lambda_i b_i \mid \lambda_i \in \mathbb{C}_0 \right\}$ (Our definition of the unit ball in $\mathbb{C}$ agrees with this definition trivially. Such sanity checks are always worth doing, just to ensure that we are being consistent). The approximate unit ball would therefore be defined as $\tilde{V_0} := \left\{ \sum_{i=0}^{m-1} \lambda_i b_i \mid \lambda_i \in \tilde{\mathbb{C}_0} \right\}$. Again, as in the case with $\mathbb{C}_0$, we see that $\rho(V_0) \subset \tilde{V_0}$.

Storing elements of $\tilde{V_0}$ is accomplished in a fashion analogous to how we stored elements of $\tilde{C_0}$. Given that any element in $\tilde{V_0}$ may be expressed as $\sum_{i=0}^{m-1} \lambda_i b_i$, where $\lambda_i \in \tilde{\mathbb{C}_0}$, we may store such an element in the form of a list of m elements, containing each of the scalars $\lambda_1, \ldots, \lambda_{m-1}$ in order. Since $\Lambda_i \in \tilde{\mathbb{C}_0}$, it takes $O(p)$ bits to store one such scalar, a given vector may be stored in $O(mp)$ bits.

Finally, before proceeding to analyse errors in some basic operations on elements in vector spaces, we define what it means for a vector to have an error. Let us say that for some $v \in V_0$, we have computed an approximation $\tilde{v} \in \tilde{V_0}$. Then, the error in v is measured as follows:

$$\varepsilon(\tilde{v}) := 2^p \|\tilde{v} - v\|$$

We note that here the norm utilised is still the sup norm, so effectively, this becomes the maximum of the errors in approximating each component of the vector. The reason for multiplication by the scalar 2^p is to ensure that until otherwise specified, the error is in terms of this

particular quantity. In other words, unless specified, if we say that $\epsilon(\tilde{v}) = 1$, what we actually mean is that the error in approximating v when the approximation is $\tilde{v}$ is $1 \cdot 2^{-p}$.

5.3.4 Some Basic Results Concerning Operations on Elements in Vector Spaces over $\mathbb{C}$

In this section, we go through some basic results on operations on elements in vector spaces over $\mathbb{C}$. The lemmas and theorems described in this section are all mentioned in section 2 of Harvey and van der Hoeven's paper. We have described in detail the proof of Lemma 2.1 of Harvey and van der Hoeven's paper, which is the first lemma written below. We note that due to the paucity of time, we have omitted proofs of the other lemmas in this section. However, we do have described the results briefly, in order to give the reader an idea of what Harvey and van der Hoeven have proved in the section. The lemmas are numbered differently from Harvey and van der Hoeven's paper here, but the original numbering is present inside brackets in bold and underline characters. We follow this convention throughout the paper.

Lemma 5. *(**Lemma 2.1:**) Given as input $u, v \in \tilde{V_0}$, in time $O(mp)$ we may compute an approximation for $\tilde{w} \in \tilde{V_0}$ for $w := \frac{1}{2}(u \pm v)$ such that $\epsilon(\tilde{w}) < 1$.*

Proof. Harvey and van der Hoeven only prove this result for $m = 1$, i.e., $V = \mathbb{C}$. They argue that the proof for the general case follows immediately from here. We shall therefore outline the proof for $V = \mathbb{C}$, and briefly explore how we may generalise this argument.

Let $u, v \in \tilde{\mathbb{C}_0}$. This means that $u = 2^{-p}(x_1 + iy_1)$, $v = 2^{-p}(x_2 + iy_2)$, where $x_1 + iy_1, x_2 + iy_2 \in \mathbb{Z}[i]$. We now let $a = 2^p u$, and $b = 2^p v$, which implies $a, b \in \mathbb{Z}[i]$. We know that $2^p w = \frac{1}{2}(a \pm b) = \frac{1}{2}((x_1 \pm x_2) + i(y_1 \pm y_2))$. Now if $x_1 \pm x_2$ and $y_1 \pm y_2$ are even quantities, then since they are divisible by 2, we know that $\rho'(2^p w) - 2^p w = 0$. If any one of these quantities is an odd number, then we would have some error term. Let us try and estimate this error. If x is a non-negative odd number, then $\rho_0\left(\frac{x}{2}\right) = \frac{x-1}{2}$. Otherwise, $\rho_0\left(\frac{x}{2}\right) = \frac{x+1}{2}$. Either way we see that for a given integer x, $\left|\rho_0\left(\frac{x}{2}\right) - \frac{x}{2}\right| \leq \frac{1}{2}$. This in turn means that $|\rho'(2^p w) - 2^p w| \leq \sqrt{\left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2} = \frac{1}{\sqrt{2}}$.

Finally, we come to our approximation for w, denoted by $\tilde{w}$. Set $\tilde{w} := \rho(w)$ (which we recall is equal to $2^{-p}\rho'(2^p w)$). Now (question about using $\|$ and $||$ interchangeably here).

$$\begin{aligned}
\epsilon(\tilde{w}) &= 2^p \|\rho(w) - w\| \\
&= 2^p \|2^{-p}\rho'(2^p w) - 2^{-p}(2^p w)\| \\
&= 2^{p-p} \|\rho'(2^p w) - 2^p w\| \\
&\leq \frac{1}{\sqrt{2}} \\
&< 1
\end{aligned}$$

It is evident that $\tilde{(}w)$ satisfies the necessary error bound. Now, all that is left is ensuring that this may be calculated in $O(p)$ time. Now we know that since u and v are elements of $\tilde{\mathbb{C}}_0$, it takes $O(p)$ bits to store them. a and b may easily be computed in $O(p)$ time (we are just shifting the binary point p places to the right in the real and imaginary parts of u and v, which definitely takes $O(p)$ time). We had noted above that given any element in $\tilde{\mathbb{C}}_0$, its real and imaginary parts are lesser than or equal to 2^p. This means that the real and imaginary parts of a and b are either p or $p+1$ bit integers, and we may add and subtract these quantities in $O(p)$ time from the assumptions that we have made in the beginning. Dividing this quantity by half is done easily by placing a binary point before the last digit in $O(1)$ time. Rounding towards zero may be done in $O(1)$ time by simply deleting the bit after the binary point. Lastly, scaling by 2^{-p} may be done in $O(p)$ time, giving us a total cost of $O(p)$ bit operations to calculate $\tilde{w}$.

We complete our proof by arguing the same for any finite-dimensional vector space over $\mathbb{C}$. The argument extends very nicely because we have defined our function ρ component-wise. Therefore, the procedure for a finite-dimensional vector space over $\mathbb{C}$ is the same as the one described above, just repeated for each of the $m-$components of the vector. It is easy to see therefore that it takes $O(mp)$ time to compute the approximation. Furthermore, since we will measure error by taking the maximum of the component-wise error (because the norm defined on such vector spaces is the sup norm), the proof of correctness for the error bound remains identical. $\square$

Lemma 6. *(**Lemma 2.2**) Let $u \in V$ and $c \in \mathbb{Z}$ such that $1 \leq c \leq 2^p$. Assume that $\|u\| \leq c^{-1}$, and let $v := cu \in \tilde{V_0}$. Given as input c and an approximation $\tilde{u} \in \tilde{V_0}$, in time $O(mp^{1+\delta})$ we may compute an approximation $\tilde{v} \in \tilde{V_0}$ such that $\varepsilon(\tilde{v}) < 2c \cdot \varepsilon(\tilde{u}) + 3$*

In this lemma, we show that given a vector in the unit ball whose norm is less than some absolute constant c, we may construct an approximation for the vector scaled by this same absolute constant (this scaled vector also lies in the unit ball) in time $O(mp^{1+\delta})$, where m is the dimension of our vector space V, and p is the precision with which we store our vectors. The error, as is the case above, is expressed in terms of units of 2^{-p}. We see this lemma applied in some of the results in the paper that deal with scaling vectors obtained as part of Fourier transforms.

Lemma 7. *(**Lemma 2.4**) Given as input in $u, v \in \tilde{\mathbb{C}}_0$, in time $O(p^{1+\delta})$ we may compute an approximation $\tilde{w} \in \tilde{\mathbb{C}}_0$ for $w := uv \in \mathbb{C}$, such that $\varepsilon(\tilde{w}) < 2$.*

This lemma is pretty self-explanatory. It talks about the cost associated with multiplying two numbers in the approximate unit ball. Of course, we compute an approximation of the product once again simply because when we have two numbers in the unit ball stored as p bits, storing their product accurately would take $2p$ bits, which we do not have access to.

Lemma 8. *(**Lemma 2.5**) Let $\mathscr{R} = \mathbb{C}[y]/\langle y^r + 1 \rangle$. Assume r is a power of two, and that $r < 2^{p-1}$. Given as input $u, v \in \tilde{\mathscr{R}}_0$, in time $4M(3rp)+O(rp)$, we may compute an approximation $\tilde{w} \in \tilde{\mathscr{R}}_0$ for $w := uv/r \in \mathscr{R}_0$ such that $\varepsilon(\tilde{w}) < 2$.*

We note that $u = u_0 + \cdots + u_{r-1}y^{r-1}$, and $v = v_0 + \cdots + v_{r-1}y^{r-1}$. The product of these two polynomials modulo $y^r + 1$ is expressed as:

$$uv = \sum_{i=0}^{r-1} y^i \left(\sum_{j=0}^{r-1} u_j v_{i-j} \right)$$

The coefficient of each term in the above product is therefore a sum of exactly r terms of the form $u_i v_j$. Now since $u, v \in \tilde{\mathscr{R}}_0$, each $|u_i| \leq 1$ and $|v_i| \leq 1$ for every i. This means $|u_i v_j| \leq 1$ for every value of i and j. Since each coefficient is a sum of exactly r such terms, we may say that

the modulus of every coefficient is at most r. This is why the authors claim that $w := uv/r$ lies inside the unit ball of $\mathscr{R}$.

We now get to what the authors want to establish in the lemma itself. The claims are simple: they can compute an approximation of the product of two polynomials (scaled by $1/r$) in the given ring in a certain amount of time, with the specified error. We describe very quickly what $\mathrm{M}(3rp)$ is. This denotes the time it takes to multiply two integers of $3rp$ bits each. As we noted before we started the chapter, we will be operating in the bit complexity model, and the algorithm the authors utilise in order to prove this lemma requires the multiplication of integers of size $3rp$ bits, which is why we have this extra cost. The algorithm used by Harvey and van der Hoeven is a very famous recursive algorithm for calculating such polynomial products. They cite Corollary 8.27 in Gathen and Gerhard 2013 as a resource for the algorithm, and go on to describe its implementation. We will not be describing the algorithm at the moment (we intend to do so later), so for the moment, we assume this result.

5.3.5 Results Regarding Approximating Linear and Bilinear Maps on Vector Spaces Over $\mathbb{C}$

In this newt few lemmas, Harvey and van der Hoeven deal with approximations of linear and bilinear maps over $\mathbb{C}$. We are all familiar with what linear maps on vector spaces over $\mathbb{C}$ look like, but what is a bilinear map?

Let U, V and W be vector spaces over $\mathbb{C}$. Then, we define a bilinear map $\mathcal{A} : U \times V \to W$ as a function that satisfies the properties listed below. Here, $u, x \in U$, $v, y \in V$, and $\alpha \in \mathbb{C}$.

- $\mathcal{A}(u + x, v + y) = \mathcal{A}(u, v) + \mathcal{A}(u, y) + \mathcal{A}(x, v) + \mathcal{A}(x, y)$
- $\mathcal{A}(\alpha u, v) = \alpha \mathcal{A}(u, v)$
- $\mathcal{A}(u, \alpha v) = \alpha \mathcal{A}(u, v)$

Now let $\mathcal{A} : V \to W$ be a linear map over $\mathbb{C}$. We define something called an operator norm on $\mathcal{A}$ as below:

$$\|\mathcal{A}\| := \sup_{v \in V_0} \|\mathcal{A}v\|$$

An approximation $\tilde{\mathcal{A}}$ for $\mathcal{A}$ is a function defined from $\tilde{V_0}$ to $\tilde{W_0}$. The error in this approximation is measured as:

$$\varepsilon(\tilde{\mathcal{A}}) := \max_{v \in \tilde{V_0}} 2^p \|\tilde{\mathcal{A}}v - \mathcal{A}v\|$$

Again, the norm utilised here is still the sup norm, and this is something the reader should always keep in mind.

Having built up the necessary notation, we now move on to stating the lemmas for linear maps that Harvey and van der Hoeven go through.

Lemma 9. *(**Lemma 2.6:**) Let $\mathcal{A} : V \to W$ be a $\mathbb{C}-$linear map such that $\|\mathcal{A}\| \leq 1$, and let $v \in V_0$. Let $\tilde{A} : \tilde{V_0} \to \tilde{W_0}$ be a numerical approximation for $\mathcal{A}$ and let $\tilde{v} \in \tilde{V_0}$ be an approximation for v. Then, $\tilde{w} := \tilde{\mathcal{A}}\tilde{v} \in \tilde{W_0}$ is an approximation for $w := \mathcal{A}v \in W_0$, such that $\varepsilon(\tilde{w}) \leq \varepsilon(\tilde{\mathcal{A}}) + \varepsilon(\tilde{v})$*

The lemma is not very difficult to understand. It talks about the propagation of error in linear maps. If you look closely, one can see an argument that hinges on the triangle inequality pop out of the lemma almost instantly. Indeed, this is exactly the proof the authors suggest, and we do not discuss it here.

Corollary 10. *(**Corollary 2.7:**) Let $\mathcal{A} : U \to V$ and $\mathcal{B} : V \to W$ be $\mathbb{C}-$linear maps such that $\|\mathcal{A}\|, \|\mathcal{B}\| \leq 1$. Let $\tilde{\mathcal{A}} : \tilde{U_0} \to \tilde{V_0}$ be numerical approximations. Then, $\tilde{\mathcal{C}} := \tilde{\mathcal{B}}\tilde{\mathcal{A}} : \tilde{U_0} \to \tilde{W_0}$ is a numerical approximation for $\mathcal{C} := \mathcal{B}\mathcal{A} : U \to W$ such that $\varepsilon(\tilde{\mathcal{C}}) \leq \varepsilon(\tilde{\mathcal{B}}) + \varepsilon(\tilde{\mathcal{A}})$*

This corollary states that errors in linear maps add up when composed, and once again, te=he proof uses the triangle inequality.

We now move on to the results that they state for bilinear maps. We define the operator norm on bilinear maps the same way we did on linear maps, i.e., given a bilinear map $\mathcal{A} : U \times V \to W$:

$$\|\mathcal{A}\| := \sup_{u \in U_0, v \in V_0} \|\mathcal{A}(u, v)\|$$

Let $\|\mathcal{A}\| \leq 1$, and let $\tilde{\mathcal{A}} : \tilde{U_0} \times \tilde{V_0} \to \tilde{W_0}$ be an approximation for $\mathcal{A}$. Then, we define the error in this approximation as:

$$\varepsilon(\tilde{\mathcal{A}}) := \max 2^p \|\tilde{\mathcal{A}}(u,v) - \mathcal{A}(u,v)$$

The time it takes for us to compute $\tilde{\mathcal{A}}(u,v)$ from u and v is given as $\mathrm{C}(\tilde{\mathcal{A}})$.

We now look at the lemmas that are stated for bilinear maps in Harvey and van der Hoeven's paper.

Lemma 11. *(**Lemma 2.8:**)Let $\mathcal{A} : U \times V \to W$ be a $\mathbb{C}-$linear map such that $\|\mathcal{A}\| \leq 1$, and let $u \in U_0$, $v \in V_0$. Let $\tilde{A} : \tilde{U_0} \times \tilde{V_0} \to \tilde{W_0}$, $\tilde{u}$, $\tilde{v}$ be approximations. Then, $\tilde{w} := \tilde{\mathcal{A}}(\tilde{u}, \tilde{v}) \in \tilde{W_0}$ is an approximation for $w := \mathcal{A}(u,v) \in W_0$, such that $\varepsilon(\tilde{w}) \leq \varepsilon(\tilde{\mathcal{A}}) + +\varepsilon(\tilde{u}) + \varepsilon(\tilde{v})$*

We see that this is the bilinear analogue of Lemma 2.6 in Harvey and van der Hoeven's paper. The proof is completed the same way that the proof for Lemma 2.6 was (using the triangle inequality).

Corollary 12. *(**Corollary 2.9:**) Let $u, v \in \mathbb{C}_0$, and let $w := uv \in \mathbb{C}_0$. Given as input approximations $\tilde{u}$, $\tilde{v} \in \tilde{\mathbb{C}_0}$, in time $O(p^{1+\delta})$ we may compute an approximation $\tilde{w} \in \tilde{\mathbb{C}_0}$ such that $\varepsilon(\tilde{w}) = \varepsilon(\tilde{u}) + \varepsilon(\tilde{v}) + 2$.*

What we see here is the multiplication of complex numbers being expressed as a bilinear map (if you aren't convinced that it can be expressed as a bilinear map from $\mathbb{C} \times \mathbb{C} \to \mathbb{C}$, do try it out! It's a pretty simple proof). One might also be wondering: why is this even worth noting when we already have Lemma 2.4 (this is the one). This is a pretty important question to answer, and the answer is that in Lemma 2.4, we assume that the vectors u and v come from the approximate unit ball. Here, we don't, which is why the error term also accounts for the errors in these approximations. The proof of this corollary follows directly from Lemma 2.8 and Lemma 2.4.

Now, we get to the biggest result in section 2. This result is concerned with approximating tensor products of linear maps.

Lemma 13. *(**Lemma 2.10:**) Let R be a coefficient ring of dimension r, and let $m_1, \ldots, m_d \geq 1$, $n_1, \ldots, n_d \geq 1$. Put $M := \prod_i \max(m_i, n_i)$, and assume $M \geq 2$. For $i \in \{1, \ldots, d\}$, let $\mathcal{A}_i : R^{m_i} \to R^{n_i}$ be a linear map over R, with $\|\mathcal{A}_i\| \leq 1$, and let $\tilde{\mathcal{A}}_i : \tilde{R}_0^{m_i} \to \tilde{R}_0^{n_i}$ be a numerical approximation. Let $\mathcal{A} := \otimes_i \mathcal{A}_i : \otimes_i R^{m_i} \to \otimes_i R^{n_i}$ (note that automatically, $\|\mathcal{A}\| \leq 1$).*

Then we may construct a numerical approximation $\tilde{\mathcal{A}} : \otimes_i \tilde{R}^{m_i} \to \otimes_i \tilde{R}^{n_i}$ such that $\varepsilon(\tilde{\mathcal{A}}) \leq \sum_i \varepsilon(\tilde{\mathcal{A}}_i)$ and

$$C(\tilde{\mathcal{A}}) \leq M \sum_i \frac{C(\tilde{\mathcal{A}}_i)}{m_i} + O(Mrp \log M)$$

We must understand what tensor products of linear maps means here. Let U, V, W and Z be vector spaces, and let $A_1 : U \to W$, $A_2 : V \to Z$ be linear maps. We already know what $U \otimes V$ and $W \otimes Z$ represent. $A_1 \otimes A_2$ is a map from $U \otimes V \to W \otimes Z$. This map is implemented in the following way. Let $u \in U$, and $v \in V$. Then, $(A_1 \otimes A_2)(u \otimes v) = A_1 u \otimes A_2 v$.

The proof of this lemma is slightly complicated, but it involves breaking down $\mathcal{A}$ into a finite sequence of maps. Think of it this way. We may implement $A_1 \otimes A_2$ above as a composition of two maps, one from $U \otimes V \to W \otimes V$, and another from $W \otimes V \to W \otimes Z$. The first map, $B_1 : U \otimes V \to W \otimes V$ is given by:

$$B_1 := A_1 \otimes I_V$$

where I_V is the identity map on V. Similarly, the second map $B_2 : W \otimes V \to W \otimes Z$ is given by:

$$B_2 := I_W \otimes A_2$$

, where I_W is the identity map on W.

Harvey and van der Hoeven do this exact same thing, but more generally. They define $U^i : R^{n_1} \otimes \cdots \otimes R^{n_i - 1} \otimes R^{m_i} \otimes R^{m_{i+1}} \cdots \otimes R^{m_d}$. We can clearly see that $U^0 = \otimes_i R^{m_i}$, and $U^d = \otimes_i R^{n_i}$. Now $\mathcal{A}$ is a map from U^0 to U^d. They decompose it into a sequence of d linear maps $\mathcal{B}_d \ldots \mathcal{B}_1$, where $\mathcal{B}_i : U^{i-1} \to U_i$ is given by:

$$\mathcal{B}_i := I_{n_1} \otimes \cdots \otimes I_{n_{i-1}} \otimes \mathcal{A}_i \otimes I_{m_{i+1}} \otimes \cdots \otimes I_{m_d}$$

The proof then uses the linearity of these maps, and Lemma 2.7 to achieve its result. What is important here is that we understand the techniques being used in order to compute $\mathcal{A}$, since understanding the running time bounds after doing so becomes a matter of some not-so-difficult algebra.

5.3.6 Results Regarding Exponentiatial Functions

The last three lemmas described here are regarding computing approximations for exponential functions. The reason we need these is that we're interested in roots of unity (they are integral to describing Fourier transforms), and in the complex numbers, roots of unity are expressed in the form of $e^{2\pi i/n}$. Harvey and van der Hoeven themselves do not provide rigorous arguments for these results, so we shall state them without proof, and move on from there.

Lemma 14. *(**Lemma 2.11**) Let $k \geq 1$ and j be integers such that $0 \leq j < k$, and let $w := e^{2\pi ij/k} \in \mathbb{C}_0$. Given j and k as input, we may compute an approximation $\tilde{w} \in \tilde{\mathbb{C}}_0$ such that $\varepsilon(\tilde{w}) < 2$ in time $O(\max(p, \log k)^{1+\delta})$.*

Corollary 15. *(**Corollary 2.12**) Let $k \geq 1$ and $j \geq 0$ be integers, and let $w := e^{-\pi j/k} \in \mathbb{C}_0$. Given j and k as input, we may compute an approximation $\tilde{w} \in \tilde{\mathbb{C}}_0$ such that $\varepsilon(\tilde{w}) < 2$ in time $O(\max(p, \log(|j| + 1), \log k)^{1+\delta})$.*

Lemma 16. *(**Lemma 2.13**) Let $k \geq 1$, $j \geq 0$ and $\sigma \geq 0$ and assume that $e^{\pi j/k} \leq 2^\sigma$ and $\sigma < 2^p$. Let $w := 2^{-\sigma} e^{\pi j/k} \in \mathbb{C}_0$. Given j, k and σ as input, we may compute an approximation $\tilde{w} \in \tilde{\mathbb{C}}_0$ such that $\varepsilon(\tilde{w}) < 2$ in time $O(\max(p, \log k)^{1+\delta})$.*

With the statements of these three results, we end our journey through section 2 of Harvey and van der Hoeven's paper.

5.4 A Brief Discussion of the Ideas in Section 3 of the Paper

We state the main result that is proved in section 3 (This is referred to as Theorem 3.1 in the paper), and then go through the major ideas that are used in its proof. We do not aim to prove Theorem 3.1 here but aim to give the reader a rough idea of the strategy that is used in the paper to prove it.

5.4.1 Stating the Main Result of the Section

Theorem 17. *(**Theorem 3.1:**) Let p be the precision parameter, such that $p \geq 100$. Let $d \geq 2$, and let $t_1, \ldots, t_d$ be powers of two such that $t_d \geq \cdots \geq t_1 \geq 2$. Let $T = \prod_{i=1}^{d} t_i$ and assume that $T < 2^p$. Then, we may construct a numerical approximation $\tilde{\mathcal{F}}_{t_1,\ldots,t_d} : \otimes_i^d \tilde{\mathbb{C}}_0^{t_i} \to \otimes_{i=1}^d \tilde{\mathbb{C}}_0^{t_i}$ for $\mathcal{F}_{t_1,\ldots,t_d}$ such that $\varepsilon(\tilde{\mathcal{F}}_{t_1,\ldots,t_d}) < 8T \log_2 T$ and*

$$C(\tilde{\mathcal{F}}_{t_1,\ldots,t_d}) < \frac{4T}{t_d} M(3t_d p) + O(Tp \log T + Tp^{1+\delta})$$

Here, $\mathcal{F}_{t_1,\ldots,t_d}$ represents the Fourier map over $\otimes_{i=1}^d \mathbb{C}^{t_i}$. The t_i^{th} primitive root of unity in $\mathbb{C}$ is given by $e^{2\pi i/t_i}$.

For the rest of this section, we set $r := t_d$ and $\mathscr{R} = \mathbb{C}[y]/\langle y^r + 1 \rangle$.

5.4.2 The Main Ideas in the Section

The main idea of this section is as follows: We know that $\mathscr{R}$ is a coefficient ring. We know that we may describe transforms Fourier on $\mathscr{R}^n$ when n is a positive divisor of $2r$. We first understand the time it takes for us to compute this Fourier map. This forms the content of Lemma 3.2 in the paper, which is stated below:

Lemma 18. *(**Lemma 3.2**) Let $\mathscr{G}_t : \mathscr{R}^t \to \mathscr{R}^t$ denote the Fourier map defined appropriately on $\mathscr{R}^t$, where $t \in \{1, 2, 4, \ldots, 2r\}$. Then, we may construct a numerical approximation $\tilde{\mathscr{G}} : \tilde{\mathscr{R}}_0^t \to \tilde{\mathscr{R}}_0^t$ for $\mathscr{G}_t$ such that $\varepsilon(\tilde{\mathscr{G}}_t)$ and $C(\tilde{\mathscr{G}}_t) = O(trp \log t)$*

The procedure that is followed for calculating the Fourier Transform here is exactly the same as the one that we described in the chapter where we described the FFT algorithm (one might've

been able to guess that from the running time we see here). The reason we get $trp \log t$ is because we're operating in the bit complexity model, and each coordinate of $\mathscr{R}^t$ is a polynomial with r coefficients, each of whom have p bits.

We then use Lemma 2.10 to generalise this result in order to obtain transforms over $\otimes_{i=1}^{d-1}\mathscr{R}^{t_i}$. This forms the basis of Proposition 3.3, which is stated below:

Proposition 19. *(**Proposition 3.3**) Let $t_1, \ldots, t_d$ and T be as in Theorem 3.1. We may construct a numerical approximation $\tilde{\mathscr{G}}_{t_1,\ldots,t_{d-1}} : \otimes_i^{d-1}\tilde{\mathscr{R}}_0^{t_i} \to \otimes_{i=1}^{d-1}\tilde{\mathscr{R}}_0^{t_i}$ for $\mathscr{G}_{t_1,\ldots,t_{d-1}}$ such that $\varepsilon(\tilde{\mathscr{G}}_{t_1,\ldots,t_{d-1}}) < \log_2 T$ and $C(\tilde{\mathscr{G}}_{t_1,\ldots,t_{d-1}}) = O(Tp \log T)$*

Following this, we make use of the multidimensional analogue of the convolution theorem, which relates multiplication in $\mathscr{R}[x_1, \ldots x_d]/\left\langle x_1^{t_1-1}, \ldots, x_{d-1}^{t_{d-1}-1}\right\rangle$ (remember, this is just another way of looking at $\otimes_{i=1}^{d-1}\mathscr{R}^{t_i}$) to Fourier transforms defined over the same space. We note that given two elements $u, v \in \otimes_{i=1}^{d-1}\mathscr{R}^{t_i}$:

$$\frac{1}{t_1 \cdots t_{d-1}} u * v = t_1 \ldots t_{d-1}\mathscr{G}^*_{t_1,\ldots,t_{d-1}}(\mathscr{G}_{t_1,\ldots,t_{d-1}}u \cdot \mathscr{G}_{t_1,\ldots,t_d}v)$$

where $\mathscr{G}^*_{t_1,\ldots,t_{d-1}}$ is the appropriately defined inverse transform on $\otimes_{i=1}^{d-1}\mathscr{R}^{t_i}$.

In order to calculate the convolution of two vectors in $\otimes_{i=1}^{d-1}\mathscr{R}^{t_i}$, what we must do then is calculate the forward transforms, multiply them pointwise, and then calculate the inverse transforms. This is basically what the authors accomplish in Proposition 3.4 of their paper. We note that in this proposition, the authors do not approximate the convolution map itself, but the normalised convolution map (the norm of the map is less than or equal to 1. This fact is stated without proof here, but really the proof just involves some minor algebraic manipulation. One can find an argument for the same in the paper itself, but we do not present it here).

In fact, one may have realised that all of the approximate maps that we have been defining have only been over the unit ball of a given vector space. This is the convention followed in the paper, and we stick to it here as well.

The map is denoted by $\mathcal{M}_{\mathscr{R}}$ and is defined as shown. $\mathcal{M}_{\mathscr{R}} : \otimes_{i=1}^{d-1}\mathscr{R}^{t_i} \times \otimes_{i=1}^{d-1}\mathscr{R}^{t_i} \to \otimes_{i=1}^{d-1}\mathscr{R}^{t_i}$ is given by:

$$\mathcal{M}_{\mathscr{R}}(u,v) = \frac{1}{T} u * v$$

Proposition 20. *(**Proposition 3.4:**) Let $t_1, \ldots, t_d$ and T be as in Theorem 3.1. We may construct a numerical approximation $\tilde{\mathcal{M}}_{\mathscr{R}} : \otimes_{i=1}^{d-1} \tilde{\mathscr{R}}_0^{t_i} \times \otimes_{i=1}^{d-1} \tilde{\mathscr{R}}_0^{t_i} \to \otimes_{i=1}^{d-1} \tilde{\mathscr{R}}_0^{t_i}$ for $\mathcal{M}_{\mathscr{R}}$ such that $\varepsilon(\tilde{\mathcal{M}}_{\mathscr{R}}) < 3T \log_2 T + 2T + 3$ and*

$$C(\tilde{\mathcal{M}}_{\mathscr{R}}) < \frac{4T}{r} M(3rp) + O(Tp \log T + Tp^{1+\delta})$$

As noted above, this proof consists of three applications of Proposition 3.3 - i.e., two for the forward transforms, and one for the inverse transform. The pointwise products are handled using Lemma 2.5 (hence the $\frac{4T}{r}\mathrm{M}(3rp)$ bound). There is an extra cost of $O(Tp^{1+\delta})$ that is added when the authors conduct some appropriate scaling required by their algorithm (we intend to fully flesh this result out at a later time).

We now discuss the major step involved in proving Theorem 3.1, which is referred to as Proposition 3.5 in the paper. It talks about the approximation of the normalised convolution map from $\mathcal{M}_{\mathbb{C}} : \otimes_{i=1}^{d} \mathbb{C}^{t_i} \times \otimes_{i=1}^{d} \mathbb{C}^{t_i} \to \otimes_{i=1}^{d} \mathbb{C}^{t_i}$. The normalised convolution map is defined below:

$$\mathcal{M}_{\mathbb{C}} := \frac{1}{T} u * v$$

Proposition 21. *(**Proposition 3.5:**) Let $t_1, \ldots, t_d$ and T be as in Theorem 3.1. We may construct a numerical approximation $\tilde{\mathcal{M}}_{\mathbb{C}} : \otimes_{i=1}^{d} \tilde{\mathbb{C}}_0^{t_i} \times \otimes_{i=1}^{d} \tilde{\mathbb{C}}^{t_i}{}_0 \to \otimes_{i=1}^{d} \tilde{\mathbb{C}}_0^{t_i}$ for $\mathcal{M}_{\mathbb{C}}$ such that $\varepsilon(\tilde{\mathcal{M}}_{\mathbb{C}}) < 3T \log_2 T + 2T + 15$ and*

$$C(\tilde{\mathcal{M}}_{\mathbb{C}}) < \frac{4T}{r} M(3rp) + O(Tp \log T + Tp^{1+\delta})$$

We note that all this while, we had been approximating transforms over $\otimes_{i=1}^{d-1} \mathscr{R}^{t_i}$. Of note here is the fact that $1 \leq i \leq d-1$. One might have been wondering why we aren't varying i from 1 to d. The reason for this comes up in the proof for this proposition. What is happening

here is essentially this: We know that $(\mathbb{C}^r, +, *) \cong (\mathbb{C}[x]/\langle x^r - 1\rangle, +, \times)$. Now we have the ring $\mathbb{C}[x]/\langle x^r - 1\rangle$ and $\mathscr{R} = \mathbb{C}[y]/\langle y^r + 1\rangle$. In the first, we make use of the fact that x is the r^{th} primitive root of unity. In the second ring, y is the $2r^{th}$ root of unity. Setting $\zeta = e^{\pi i/r}$, we can see that if we send $x \to \zeta y$, we obtain a very nice isomorphism between the two rings (one may test how this works on both basis sets - remember, they're both algebras - and get an idea of how this isomorphism operates).

We already know that $\otimes_{i=1}^{d}\mathbb{C}^{t_i} \cong \mathbb{C}[x_1, \ldots, x_d]/\left\langle x_1^{t_1} - 1, \ldots, x_{d-1}^{t_d - 1}\right\rangle$. Now, let us take the ring $\mathscr{R}[x_1, \ldots, x_{d-1}]/\left\langle x_1^{t_1} - 1, \ldots, x_{d-1}^{t_{d-1}-1}\right\rangle$. This is the same as taking the ring

$$\mathbb{C}[x_1, \ldots, x_{d-1}, y]/\left\langle x_1^{t_1} - 1, \ldots, x_{d-1}^{t_{d-1}-1, y^r+1}\right\rangle$$

. From what we have seen above, there exists a natural isomorphism between the two rings, i.e, one that sends $(x_1, x_2, \ldots, x_{d-1}, x_d) \to (x_1, \ldots, x_{d-1}, \zeta y)$. The authors have already established how to understand Fourier transforms on the second ring. So, in this proposition, we utilise Lemma 2.11 (the result about exponentiation) to calculate approximations for complex roots of unity. We then send vectors in $\otimes_{i=1}^{d}\mathbb{C}^{t_i}$ to $\mathscr{R}[x_1, \ldots, x_{d-1}]/\left\langle x_1^{t_1} - 1, \ldots, x_{d-1}^{t_{d-1}-1}\right\rangle$ using this approximated root of unity, calculate their convolution there, and send the vectors back. We see quite easily how Proposition 3.4 figures in this result as well.

5.4.3 Sketching the Proof of the Main Idea

We give a brief sketch of the proof of Theorem 3.1 here. The theorem utilises something called Bluestein's trick. What the trick says is that we may construct a vector $a \in \otimes_{i=1}^{d}\mathbb{C}^{t_i}$, such that $v = \mathcal{F}_{t_1,\ldots,t_d}u$ may be rewritten as $v = \bar{a} \cdot (\frac{1}{T}a \cdot *(\bar{a} \cdot u))$. Here, $\bar{a}$ denotes the complex conjugate of the vector a.

a is constructed as below:

$$a_{j_1,\ldots,j_d} := e^{\pi i({j_1}^2/t_1 + \cdots + {j_d}^2/t_d}$$

For the time being, we assume Bluestein's trick works. Now a may be computed using

Lemma 2.11. Once we have obtained a (obtaining $\bar{a}$ after obtaining a is a process that does not take much time, since one simply has to invert the signs of all the entries storing imaginary values), we use Corollary 2.9 to understand the pointwise product of a and u. We then use Proposition 3.5 above to compute the convolution of a and the new vector we have just formed. We then utilise Corollary 2.9 again to multiply $\bar{a}$ to this convolution and obtain the vector v with the appropriate error.

5.5 The Gaussian Resampling Technique

Recall that Harvey and van der Hoeven's main idea was to come up with maps $\mathcal{A}$ defined from $\otimes_{i=1}^{d}\mathbb{C}^{s_i} \to \otimes_{i=1}^{d}\mathbb{C}^{t_i}$ and $\mathcal{B} : \otimes_{i=1}^{d}\mathbb{C}^{t_i} \to \otimes_{i=1}^{d}\mathbb{C}^{s_i}$ such that the Fourier transform $\mathcal{F}_{s_1,\ldots,s_d}$ may be expressed as $c\mathcal{B}\mathcal{F}_{t_1,\ldots,t_d}\mathcal{A}$, where c is an appropriately chosen constant. For the purposes of the algorithm, Harvey and van der Hoeven take each s_i to be some odd prime (all the s_i's are distinct), and each t_i to be some power of 2. They also assume that for each i, $s_i < t_i$. For the discussion in this section, however, we will not need to assume the first condition. It suffices to assume that $\gcd(s_i, t_i) = 1$, for each i.

5.5.1 The Main Result in This Section

We state the main result of this section here, which we assume for the rest of the report.

Theorem 22. *(**Theorem 4.1:**) Let $d > 1$, let $s_1, \ldots, s_d$ and $t_1, \ldots, t_d$ be integers such that $2 \leq s_i < t_i < 2^p$ and $\gcd(s_i, t_i) = 1$, and let $T := t_1 \ldots t_d$. Let α be an integer in the interval $2 \leq \alpha < p^{1/2}$. For each i, let $\theta_i := \frac{t_i}{s_1} - 1$, and assume that $\theta_i \geq \frac{p}{\alpha^4}$. Then there exist linear maps $\mathcal{A} : \otimes_{i=1}^{d}\mathbb{C}^{s_i} \to \otimes_{i=1}^{d}\mathbb{C}^{t_i}$ and $\mathcal{B} : \otimes_{i=1}^{d}\mathbb{C}^{t_i} \to \otimes_{i=1}^{d}\mathbb{C}^{s_i}$, with $\|\mathcal{A}\|, \|\mathcal{B}\| \leq 1$, such that*

$$\mathcal{F}_{s_1,\ldots,s_d} = 2^{\gamma}\mathcal{B}\mathcal{F}_{t_1,\ldots,t_d}\mathcal{A} \qquad \gamma := 2d\alpha^2$$

Moreover, we may construct numerical approximations $\tilde{\mathcal{A}} : \otimes_{i=1}^{d}\tilde{\mathbb{C}}_0^{s_i} \to \otimes_{i=1}^{d}\tilde{\mathbb{C}}_0^{t_i}$ and $\tilde{\mathcal{B}} : \otimes_{i=1}^{d}\tilde{\mathbb{C}}_0^{t_i} \to \otimes_{i=1}^{d}\tilde{\mathbb{C}}_0^{s_i}$ such that $\varepsilon(\tilde{\mathcal{A}}), \varepsilon(\tilde{\mathcal{B}}) < dp^2$ and

$$\mathsf{C}(\tilde{\mathcal{A}}),\ \ \mathsf{C}(\tilde{B}) = O(dTp^{3/2+\delta}\alpha + Tp\log T)$$

(***Note:*** The constant we had been representing as c all along is 2^γ here.)

The strategy for proving the above result is to prove it for $d = 1$, i.e., the one-dimensional case, for just one pair of values, s and t. We substitute $d = 1$ and $T = t$ in all the results mentioned in the above proposition to get the statement for the one-dimensional case. Harvey and van der Hoeven spend a significant amount of time proving the result for the one-dimensional case. After the results for the case when d is arbitrary are proved by a simple application of Lemma 2.10 (the one that is used to approximate tensor products of linear maps). We omit the proof of even the one-dimensional case of the Gaussian Resampling result since we lack enough time to do justice to something that is so technical. We aim to finish describing it in detail later.

5.6 Setting Parameters and Making the Algorithm Precise

Let's get down to business. We know how the algorithm operates, so we now establish all the parameters precisely in order to make sure that it runs in the time that Harvey and van der Hoeven claim that it does. Firstly, we remember that Harvey and van der Hoeven described a family of algorithms, parametrised by a dimension factor d. By definition, $d \geq 2$.

In the algorithm itself, we had to split an $n-$bit integer into several chunks of size b. We defined $b := \lceil \log n \rceil$. We note that in order to use the d^{th} algorithm, the integers we are multiplying must be $2^{d^{12}}$ bits. This means that $b \geq d^{12} \geq 2^{12} = 4096$.

We set the precision parameter $p := 6b$.

For the purposes of the Gaussian Resampling procedure, we required a parameter α, which was we said was an integer between 2 and $p^{\frac{1}{2}}$. We set $\alpha := \lceil (12d^2b)^{1/4} \rceil$. Now $b \geq 4096$, and $b^{1/4} \geq 4096^{1/4} = 8 > 2$, which means that $\alpha > b^{1/4} \geq 8 > 2$.

We know that $d \leq b^{1/12}$ (since $d^{12} \leq b$). From here, we can see that $\alpha = \lceil (12d^2b)^{1/4} \rceil \leq \lceil (12b^{2/12}b)^{1/4} \rceil = \lceil 12^{1/4}b^{14/48} \rceil = \lceil 12^{1/4}b^{7/24} \rceil$. Now $12^{1/4} < 1.87$, so we may say that $\alpha \leq 1.87b^{7/24} + 1$ (1 is added to ensure that the quantity is greater than α since the definition of α

contains a ceiling function in it). Now we utilise the fact that $b \geq 4096$ to say that $0.13b^{7/24} > 1$. This means that $1.87b^{7/24} < 2b^{7/24}$. Since $p = 6b$, $p^{1/2} = \sqrt{6}b^{1/2} = \sqrt{6}b^{12/24}$. It is clear, therefore, that $\alpha < p^{1/2}$.

We had defined the quantity γ to be $2d\alpha^2$. We just showed that $\alpha < 2b^{7/24}$, which means that $\alpha^2 < 4b^{7/12}$. This in turn means that $\gamma < (2b^{1/12})(4b^{7/12}) = 8b^{2/3}$. This quantity is less than $b - 13$, since $b \geq 4096$, and this is a fact that we will use later on in proofs.

We define a quantity T now. T is the unique power of 2 such that $\frac{4n}{b} \leq \frac{8n}{b}$. That there exists a unique power of 2 between these two numbers is not very difficult to prove, and the reader should try it if they aren't convinced. In fact, you can prove a more general claim. Let $x > 1$. Then, $\exists\ !\ k \in \mathbb{N}$, such that $2^k \in [x, 2x)$ (since $\frac{4n}{b} > 1$, we can see how this easily applies to the case we have here).

Let r be the unique power of two such that $T^{1/d} \leq r < 2T^{1/d}$. We also define another quantity, $r' := \log_2\left(\frac{r^d}{T}\right)$. Since $r^d \geq T$, and is also a power of two (if r is a power of two, any power of r is also a power of 2), $T \mid r^d$. Furthermore, the quotient must also be a power of 2 (think of this as being of the form $\frac{2^y}{2^x}$ and it should become clear immediately why this is the case). Therefore, r' is a positive integer. We note that since r look at a power of 2 between $T^{1/d}$ and $2T^{1/d}$, $T \leq r < 2^d T$. This means $\log_2\left(\frac{r^d}{T}\right) < \log_2\left(\frac{2^d T}{T}\right) = d$.

Recall that we wanted to calculate the Fourier Transform over $\otimes_{i=1}^{d}\mathbb{C}^{s_i}$ by injectively mapping the vectors to $\otimes_{i=1}^{d}\mathbb{C}^{t_i}$, computing the transform there, and mapping the vectors obtained back. We describe the t_i's now. We set $t_1 = \cdots = t_{r'} = \frac{r}{2}$, and $t_{r'+1} = \cdots = t_d$.

From here, it is easy to see that $\prod_{i=1}^{d} t_i = \frac{r^d}{2^{d'}} = T$.

Harvey and van der Hoeven then introduce a method to find the primes s_i we had mentioned earlier. The main step they utilise for this purpose is a prime counting argument (this forms **Lemma 5.1** in their paper). The lemma is stated as follows:

Lemma 23. *(**Lemma 5.1**:) Let $\eta \in \left(0, \frac{1}{4}\right)$ and let $x \geq e^{2/\eta}$. Then there are at least $\frac{\eta x}{2\log x}$ primes q in the interval $(1-2\eta)x < q \leq (1-\eta)x$.*

Having proved this lemma, they set $\eta := \frac{1}{4d} \leq \frac{1}{8}$. They then set $x = \frac{r}{2}$ first, and argue that there are at least $\frac{1}{4d} \cdot \frac{r/2}{2\log(r/2)}$ primes q in the interval $(1 - \frac{2}{4d})\frac{r}{2} < q \leq (1 - \frac{1}{4d})\frac{r}{2}$. Now

$$\begin{aligned}\frac{1}{4d}\cdot\frac{r/2}{2\log(r/2)} &= \frac{1}{16d}\cdot\frac{r}{\log(r/2)}\\ &\geq \frac{1}{16d}\cdot\frac{r}{\log(r)}\end{aligned}$$

Now $r \geq T^{1/d} = \left(\frac{4n}{b}\right)^{1/d}$. We know that $b \leq \log_2 n + 1$, and since n is so large, we may certainly say that $\log_2 n + 1 < 4n^{1/2}$. This means that $b \leq 4n^{1/2}$, which means that $r \geq \left(\frac{4n}{4n^{1/2}}\right)^{1/d} = n^{1/2d}$. Since $n \geq 2^{d^{12}}$, we have $r \geq (2^{d^{12}})^{1/2d} = 2^{d^{11}/2} \geq 2^{d^{10}}$, since $d \geq 2$.

This of course means that:

$$\begin{aligned}\frac{1}{4d}\cdot\frac{r/2}{2\log(r/2)} &\geq \frac{1}{16d}\cdot\frac{r}{\log(r)}\\ &\geq \frac{1}{16d}\cdot\frac{2^{d^{10}}}{\log(2^{d^{10}})}\\ &= \frac{2^{d^{10}}}{16d(d^{10})\log 2}\\ &= \frac{2^{d^{10}}}{16d^{11}\log 2}\end{aligned}$$

Here, Harvey and van der Hoeven use the following identity. We know that $2^{d^{10}}$ is a massive number. What would the numerator have to be for our fraction to evaluate to d? It would have to be $(16\log 2)(d^{12})$. Since $2^{d^{10}}$ is so large, the authors assume that it is greater than $(16\log 2)(d^{12})$. This shouldn't be too hard to see, exponents do grow much bigger than polynomials, and the constant $(16\log 2)$ isn't big enough to make a difference at this scale. Now we also know that $d' \leq d$. In sum, this means that we can find over d' distinct primes in the neighbourhood defined, so we do not need to worry about our algorithm not working because we don't have primes.

Similarly, we may find $d - d'$ many distinct primes in the interval $\left(\left(1-\frac{1}{2d}\right)r, \left(1-\frac{1}{4d}\right)r\right]$. Harvey and van der Hoeven claim that these primes may be found in $O(n)$ time (actually, they claim they can do this in $o(n)$ time, but we stick to $O(n)$ simply because this is what we have

been introduced to us in this report).

Harvey and van der Hoeven argue that the two collections of primes that they have are disjoint. This is because $\left(1 - \frac{1}{4d}\right) \frac{r}{2} < \frac{r}{2}$. Since $\frac{1}{4d} \leq \frac{1}{8}$, $\left(1 - \frac{2}{4d}\right) r \geq \frac{3r}{4} > \frac{r}{2}$ (What this shows us is that the upper bound for one set of primes is smaller than the lower bound for the other set, so the sets must be disjoint).

We have found primes s_i such that $(1 - 2\eta)t_i < s_i \leq (1 - \eta)t_i$, where $1 \leq i \leq d$. This means that $\frac{s_i}{t_i} > (1 - 2\eta)$, which in turn means that $\prod_{i=1}^{d} \frac{s_i}{t_i} > (1 - 2\eta)^d$. At this juncture, Harvey and van der Hoeven use the following identity: if $x \in (0, 1)$, then $(1 - x)^d > 1 - dx$. We know that $\prod_{i=1}^{d} s_i = S$, and $\prod_{i=1}^{d} t_i = T$. $\eta := \frac{1}{4d}$, which means we have:

$$\begin{aligned} \frac{S}{T} &> \left(1 - \frac{2}{4d}\right)^d \\ &= 1 - \frac{1}{2} \\ &= \frac{1}{2} \end{aligned}$$

This means that $S > \frac{2n}{b}$, by the definition of T. Since $N := \lceil \frac{n}{b} \rceil$, $N - 1 < \frac{n}{b}$. This in turn means that $2(N - 1) < \frac{2n}{b} < S$. We recall that initially, we had wanted to calculate our polynomial product modulo $x^S - 1$, where S is a positive integer greater than $2N - 2$. Now, we have obtained exactly this S.

We now go through the proof of Proposition 5.2 of the paper, which is instrumental in order to prove that the algorithm runs correctly, and in the time that it does.

The proposition is stated below:

Proposition 24. *(**Proposition 5.2:**) We may construct a numerical approximation* $\tilde{\mathcal{F}}_{s_1,\ldots,s_d} : \otimes_{i=1}^{d} \tilde{\mathbb{C}}_0^{s_i} \to \otimes_{i=1}^{d} \tilde{\mathbb{C}}_0^{s_i}$ *such that* $\varepsilon(\tilde{\mathcal{F}}_{s_1,\ldots,s_d}) < 2^{\gamma+5} T \log_2 T$ *and*

$$C(\tilde{\mathcal{F}}_{s_1,\ldots,s_d}) < \frac{4T}{r} M(3rp) + O(n \log n)$$

Proof. The proof implemented here basically utilises Theorem 3.1 and Theorem 4.1. We use Theorem 3.1 in order to calculate the approximation of the Fourier transform map over $\otimes_{i=1}^{d} \mathbb{C}^{t_i}$

(denoted by $\tilde{\mathcal{F}}_{t_1,\ldots,t_d}$) quickly, and we utilise Theorem 4.1 in order to generate approximations of the maps $\mathcal{A}$ and $\mathcal{B}$ (denoted by $\tilde{\mathcal{A}}$ and $\tilde{\mathcal{B}}$ respectively) where $\mathcal{F}_{s_1,\ldots,s_d} = 2^{\gamma}\mathcal{B}\mathcal{F}_{t_1,\ldots,t_d}\mathcal{A}$.

The hypotheses of Theorem 3.1 require each t_i to be a power of 2, with $t_{i+1} \geq t_i$ for each i. We also require their product $T < 2^p$. That the first hypothesis is verified is very easy to check (earlier t_is are $r/2$, later ones are r). The second one is also easy to check, since $T < n$, and $n \leq 2^b < 2^{6b} = 2^p$.

We now verify the hypothesis of Theorem 4.1. Each of the (s_i, t_i) pairs is clearly coprime (s_is are odd primes, t_is are powers of two). $\alpha < p^{1/2}$ is something we had established already. Now we defined $\theta_i := t_i/s_i - 1$. We know that $t_i/s_i \geq 1/(1-\eta)$ from before ($s_i \leq (1-\eta)t_i$ by definition). This means that $\theta_i \geq \frac{1}{1-\eta} - 1 = \frac{\eta}{1-\eta}$. Now since $\eta := 1/4d \leq 1/8$, we know that $\frac{\eta}{1-\eta} > \eta$. This means that $\theta_i \geq \eta = \frac{1}{4d}$.

Now $\alpha^4 \geq 12d^2b$ (since $\alpha := \lceil (12d^2b)^{1/4} \rceil$), which means $\alpha^4\theta_i \geq \frac{12d^2b}{4d} = 3db = \frac{d}{2}\cdot 6b = \frac{d}{2}p \geq p$, since $d \geq 2$. We have therefore established that $\alpha^4\theta_i \geq p$, implying that $\theta_i \geq \frac{p}{\alpha^4}$. This completes our verification of the hypotheses of Theorem 4.1.

Define $\tilde{\mathcal{F}'}_{s_1,\ldots,s_d} := \tilde{\mathcal{A}}\tilde{\mathcal{F}}_{t_1,\ldots,t_d}\tilde{\mathcal{B}}$. We get $\tilde{F}_{s_1,\ldots,s_d}$ by first computing $\tilde{\mathcal{F}'}_{s_1,\ldots,s_d}$ and then scaling it the output by 2^{γ}. All Fourier maps defined in the paper are normalised, which means that $\mathcal{F}_{s_1,\ldots,s_d}$ has a norm at most 1 as well. This means we that when we scale the result of $\tilde{\mathcal{F}'}_{s_1,\ldots,s_d}$, we must still get a vector in the unit ball of $\otimes_{i=1}^{d}\mathbb{C}^{s_i}$. We may therefore use the scaling lemma (i.e., Lemma 2.2) in order to compute the scaled vector in time $O(Sp^{1+\delta})$.

Now we may calculate $\tilde{\mathcal{F}}'_{s_1,\ldots,s_d}$ in time $\mathrm{C}(\tilde{\mathcal{A}}) + \mathrm{C}(\tilde{\mathcal{F}}_{t_1,\ldots,t_d}) + \mathrm{C}(\tilde{\mathcal{A}})$. Using the results of Theorem 3.1 and Theorem 4.1, we get:

$$\mathrm{C}(\tilde{\mathcal{F}}'_{s_1,\ldots,s_d}) \quad = \quad \frac{4T}{r}\mathrm{M}(3rp) + O(dTp^{3/2+\delta}\alpha + Tp\log T + Tp^{1+\delta})$$

Now the time required to scale the vector obtained from this was $O(Sp^{1+\delta})$, but since $S < T$, we can see that $O(Sp^{1+\delta}) = O(Tp^{1+\delta})$, which is already present in the above bound.

This means that $\mathrm{C}(\tilde{\mathcal{F}}'_{s_1,\ldots,s_d})$ is exactly the same as above, i.e., $\mathrm{C}(\tilde{\mathcal{F}}'_{s_1,\ldots,s_d}) = \frac{4T}{r}\mathrm{M}(3rp) + O(dTp^{3/2+\delta}\alpha + Tp\log T + Tp^{1+\delta})$.

Now we know that $d \leq b^{1/12}$, since $n = 2^{d^{12}}$, and $b = \lceil \log_2 n \rceil$. We have already set $p := 6b$. This means that $d < p^{1/12}$. We know that $\alpha = O(p^{7/24})$ from here. Utilising the fact that $\delta < 1/8$, we may say:

$$\begin{aligned} dp^{3/2+\delta}\alpha &= O(p^{1/12}p^{3/2+\delta}p^{7/24}) \\ &= O(p^{9/24}p^{3/2+\delta}) \\ &= O(p^{3/8}p^{12/8+1/8}) \\ &= O(p^2) \end{aligned}$$

We see that $O(dTp^{3/2+\delta}\alpha + Tp\log T + Tp^{1+\delta})$ therefore simplifies $O(Tp^2 + Tp\log T + Tp^{1+\delta}) = O(Tp^2 + Tp\log T) = O(Tp^2)$, since $\log T < \log n < p$. Now $T = O(n/\log n)$, and $p = O(\log n)$, which is why we get that $O(Tp^2) = O(n\log n)$.

As for the error bound, we see using additivity of errors in approximations of linear maps that $\varepsilon(\tilde{\mathcal{F}}'_{s_1,\ldots,s_d}) \leq \varepsilon(\tilde{\mathcal{A}}) + \varepsilon(\tilde{\mathcal{F}}_{t_1,\ldots,t_d})) + \varepsilon(\tilde{\mathcal{A}})$. Again, from the results of Theorem 3.1 and Theorem 4.1, we have:

$$\varepsilon(\tilde{\mathcal{F}}'_{s_1,\ldots,s_d}) < 2dp^2 + 8T\log_2 T$$

The error for $\tilde{\mathcal{F}}_{s_1,\ldots,s_d}$ is given by substituting $\varepsilon(\tilde{\mathcal{F}}'_{s_1,\ldots,s_d})$ in the place of $\varepsilon(\tilde{u})$ and 2^γ in the place of c in the error bound for Lemma 2.2. Doing so yields

$$\begin{aligned} \varepsilon(\tilde{\mathcal{F}}_{s_1,\ldots,s_d}) &< 2\cdot 2^\gamma(2dp^2 + 8T\log_2 T) + 3 \\ &< 2^{\gamma+1}(3dp^2 + 8T\log_2 T) \end{aligned}$$

where we have used the fact that $2^{\gamma+1}dp^2 > 3$ (this is pretty easy to see). Now $d \leq b^{1/12}$, and $p = 6b$, which means that $3dp^2 \leq 3(b^{1/12}\cdot 36b^2) = 108b^{1/12}b^2 = 108\lceil \log_2 n \rceil^{25/12}$.

Since n is so large, we can say that $108\lceil \log_2 n \rceil^{25/12} < n^{1/2}$. We know that the second function grows faster than the first, and by $n = 2^{2^{12}}$, we may verify that the second function has a higher value than the first (do so if you aren't convinced!). Furthermore, since $T \geq 4n/b$, we may say that $T > n^{1/2}$ (this is because $\log_2 n < n^{1/2}$, so $n/\log_n > n/n^{1/2}$. The difference in value is especially stark for large values of n). Clearly, $T < 8T \log_2 T$. From all of this, we can say:

$$\begin{aligned} \varepsilon(\tilde{\mathcal{F}}_{s_1,\ldots,s_d}) &< 2^{\gamma+1}(3dp^2 + 8T\log_2 T) \\ &< 2^{\gamma+1}(8T\log_2 T + 8T\log_2 T) \\ &= 2^{\gamma+5} T \log_2 T \end{aligned}$$

With that, we are done.

□

We now apply this result in order to calculate the normalised convolution map over $\otimes_{i=1}^{d} \mathbb{C}^{s_i}$. The normalised convolution map denoted by $\mathcal{M} : \otimes_{i=1}^{d} \mathbb{C}^{s_i} \times \otimes_{i=1}^{d} \mathbb{C}^{s_i} \to \otimes_{i=1}^{d} \mathbb{C}^{s_i}$ is defined as:

$$\mathcal{M}(u, v) := \frac{1}{S}(u * v)$$

The authors approximate this map using a result termed Proposition 5.3 in their paper. This result is described below:

Proposition 25. *(**Proposition 5.3:**) We may construct an approximation $\tilde{\mathcal{M}} : \otimes_{i=1}^{d} \tilde{\mathbb{C}}_0^{s_i} \times \otimes_{i=1}^{d} \tilde{\mathbb{C}}_0^{s_i} \to \otimes_{i=1}^{d} \tilde{\mathbb{C}}_0^{s_i}$ such that $\varepsilon(\tilde{\mathcal{M}}) < 2^{\gamma+8} T^2 \log_2 T$ and*

$$C(\tilde{\mathcal{M}}) < \frac{12T}{r} M(3rp) + O(n \log n)$$

We follow the same method that Harvey and van der Hoeven use in their paper to prove Proposition 3.4. We know that calculating a convolution involves computing two forward transforms, one pointwise product, and an inverse transform. From the multidimensional analogue

to the convolution theorem, we know that $\frac{1}{S}u * v = S\mathcal{F}^*_{s_1,\ldots,s_d}(\mathcal{F}_{s_1,\ldots,s_d}u \cdot \mathcal{F}_{s_1,\ldots,s_d}v)$.

We set $w' := \mathcal{F}_{s_1,\ldots,s_d}u \cdot \mathcal{F}_{s_1,\ldots,s_d}v$, and calculate its approximation using Proposition 5.2 thrice (this is why the constant associated with the M($3rp$) term changes from $\frac{4T}{r}$ to $\frac{12T}{r}$ here. The $O(n \log n)$ doesn't change even when we consider pointwise multiplications since the authors argue that this may be done in $O(Tp^{1+\delta})$ time - we have S total components, it takes $O(p^{1+\delta})$ to multiply each pair of such components, and $O(Sp^{1+\delta}) = O(Tp^{1+\delta})$).

We then set $w := Sw'$ and compute it using Lemma 2.2. The error term is greater than the one in Proposition 5.2 due to the application of the approximate Fourier map thrice, followed by the subsequent scaling.

We are now in a position to understand the running time of the algorithm, which we do below.

5.7 The Algorithm Runs in Time $O(n \log n)$, and it Runs Correctly. Why?

We describe the major steps of the algorithm and the cost associated with each one below:

1. Take two $n-$ bit integers, split them into polynomials in one variable with coefficients of size b of degree $N-1$. This takes $O(n)$ time, since we just have to mark out where each coefficient begins and ends.

2. In order to compute the integer product, it suffices to compute the polynomial product and evaluate it at 2^b.

3. Since the degree of the product polynomial is at most $2N-2$ and $S > 2N-2$, we calculate the polynomial product modulo $x^S - 1$. Expressing the polynomial as part of this ring takes no additional time, as they remain the same as before ($N-1 < S$, so the polynomials do not change when expressed in $\mathbb{Z}[x]/\langle x^S - 1\rangle$ instead of $\mathbb{Z}[x]$).

4. Using the Chinese Remainder Theorem, send the polynomials from $\mathbb{Z}[x]/\langle x^S - 1\rangle$ to $\mathbb{Z}[x_1,\ldots,x_d]/\langle x_1^{s_1} - 1,\ldots,x_d^{s_d} - 1\rangle$. The way Harvey and van der Hoeven intend to store

these polynomials is in the form of a $d-$dimensional array. The coefficients of the term $x_1^{j_1} \dots x_d^{j_d}$ in the polynomial u may be denoted as $u_{j_1,\dots,j_d} \in \mathbb{Z}$.The polynomial coefficients are stored in the array in lexicographic order. By this we mean that the constant coefficient $(u_{0,\dots,0})$ is stored first, followed by the coefficient of x_d $(u_{0,\dots,1})$, followed by the coefficient of x_d^2 $(u_{0,\dots,0,2})$, then x_d^3 $(u_{0,\dots,0,3})$, and so on until the final value stored is the coefficient of $x_1^{s_1-1}x_2^{s_2-1}\dots x_d^{s_d-1}$ $(u_{s_1-1,\dots,s_d-1})$. They argue that computing this isomorphism may be done in $O(n \log n)$ time.

5. Treating $\mathbb{Z}[x_1,\dots,x_d]/\left\langle x_1^{s_1}-1,\dots,x_d^{s_d}-1\right\rangle$ as a subring of $\mathbb{C}[x_1,\dots,x_d]/\left\langle x_1^{s_1}-1,\dots,x_d^{s_d}-1\right\rangle$, we use the isomorphism between $\mathbb{C}[x_1,\dots,x_d]/\left\langle x_1^{s_1}-1,\dots,x_d^{s_d}-1\right\rangle$ and $\otimes_{i=1}^d \mathbb{C}^{s_i}$ and express our polynomials as elements of the latter space. This takes no additional time, since tensors and elements of multivariate polynomial rings are stored the same way by Harvey and van der Hoeven.

6. We normalise the tensors by dividing them by 2^b (each coefficient is less than 2^b, since it is at most b bits long). Let the normalised tensors be u and v (they now belong in the unit ball of $\otimes_{i=1}^d \mathbb{C}^{s_i}$), and let $w := \frac{1}{S} u * v$. We now approximate w using Proposition 5.3. Let this approximation be $\tilde{w}$. Then, $\varepsilon(\tilde{w}) < 2^{\gamma+8}T^2 \log_2 T$, and we may compute $\tilde{w}$ in time $(12T/r)\mathrm{M}(3rp) + O(n \log n)$.

7. Provided the error bound is low enough (which we argue below), the polynomial product is $2^{2b}S\tilde{w}$. We are able to calculate this quantity given $\tilde{w}$ in time $O(Sp^{1+\delta})$. Since $S = O(T)$, $T = O(n/\log n)$, $p = O(\log n)$, $O(Sp^{1+\delta}) = O((n/\log n)(\log n)^{1+\delta}) = O(n(\log n)^{\delta}) = O(n \log n)$.

8. Having calculated our polynomial product successfully, we evaluate our polynomial at 2^b in $O(n)$ time to obtain our integer product.

5.7.1 Arguing that the Error Bound is Low Enough

We saw above that the only step that generates an error is the approximation of $\tilde{w}$. We know this error to be less than $2^{\gamma+8}T^2 \log_2 T$. Now when we want to get back the actual polynomial

itself, since we had computed the normalised convolution of two normalised vectors, we must multiply these constants back in our error. Overall, we had normalised by $2^{2b}S$, which means the error in the polynomial product is actually $2^{2b}S(2^{\gamma+8}T^2 \log T)$ (Remember, we are calculating error in a vector in $\otimes_{i=1}^{d}\mathbb{C}^{s_i}$, and this error is defined as the maximum component-wise error using the sup norm. So, the error in every component is at most this much). We note that this error has been expressed in terms of units of 2^{-p} all along, so the absolute error in the polynomial product is: $2^{\gamma+8+2b-p}ST^2 \log_2 T$.

What is our goal here? We know that we are computing the product of polynomials with integer coefficients. This means that the product polynomial itself must have integers coefficients. The difference between any two integers is at least 1. Now the approximation of our product polynomial may not have integer coefficients, but we would like the error to be small enough that this approximation has coefficients that are closest to a unique integer. Since any two integers differ by at least 1, we would like the absolute error to be less than $\frac{1}{2}$ in order for this to happen (otherwise, we cannot say which integer our approximate coefficient is closest to, and one can easily see why this would derail the algorithm completely).

We know that the error in each coefficient is less than $2^{\gamma+8+2b-p}ST^2 \log_2 T$. But $S < T$, so the error must be less than $2^{\gamma+8+2b-p}T^3 \log_2 T$. Now $T^2 < n^2 \leq 2^{2b}$ (since $b := \lceil \log_2 n \rceil$), and $T \log_T < T \log_2 n \leq Tb \leq 8n \leq 2^{b+3}$, since by definition $T \leq \frac{8n}{b}$. Substituting these in our error bound, we get that our error must be less than $2^{\gamma+8+2b-p}2^{2b}2^{b+3} = 2^{\gamma+5b+11-p}$.

We had defined $p := 6b$, so we get that our error must be less than $2^{\gamma+11-b}$. We had also argued that $\gamma < 8b^{2/3}$, and since b is large enough (remember, b is at least $2^{12} = 4096$), $8b^{2/3} < b - 13$. This means that our error is less than $2^{b-13+11-b} = 2^{-2} = \frac{1}{4}$.

We now see that our algorithm runs such that the absolute error in each coefficient is less than $\frac{1}{4}$. This means that there is a unique integer that this coefficient is closest to, and upon rounding to this integer, we get the accurate polynomial product. Thus, the algorithm runs correctly on all inputs.

5.7.2 Proving that the Algorithm Runs in $O(n \log n)$ for Large Enough n

Harvey and van der Hoeven prove this in two stages, one of which we assume, and the other which we go through. We know that the maximum time taken for any of the steps mentioned in the first part of this section is in the step where we calculate the approximate normalised convolutions. This step takes $12T/r\mathrm{M}(3rp) + O(n \log n)$ time, and since all other steps are absorbed by the $O(n \log n)$ term, we can say that $\mathrm{M}(n) = 12T/r\mathrm{M}(3rp) + O(n \log n)$ (where $\mathrm{M}(n)$ is the time taken to multiply two $n-$bit integers using this algorithm).

Harvey and van der Hoeven first define a quantity, $\mathrm{T}(n) := \frac{\mathrm{M}(n)}{n \log n}$. In a result we now assume (this involves a lot of symbolic manipulation of constants), they show that $\mathrm{T}(n) < \frac{1728}{d-\frac{1}{2}}\mathrm{T}(3rp) + O(1)$.

What they proceed to do is to set $d = 1729$. The motivation behind this is clearly to ensure that $\frac{1728}{d-\frac{1}{2}} < 1$. The reason we want this to happen is simply because by induction, we aim to prove that $T(n) = O(1)$. After this, it becomes instantly clear that $\mathrm{M}(n) = O(n \log n)$.

For $d = 1729$, we see that $\mathrm{T}(n) < 0.9998\mathrm{T}(3rp) + O(1)$. Since $O(1)$ represents some constant time, we set this constant to be A. This means that $\mathrm{T}(n) < \mathrm{T}(3rp) + A$. Now in order to multiply integers with the algorithm when $d = 1729$, the integers must be at least $n_0 = 2^{1729^{12}}$ bits long. For integers of smaller sizes, we use any convenient multiplication algorithm.

We set $B := \max\{T(n) \mid 2 \leq n < n_0\}$, and $C := \max\{B, 5000A\}$. By induction, we aim to prove that $T(n) \leq C$. For $n < n_0$, this is clear, since C is at least equal to B, and $B \geq T(n)$ for every $n < n_0$. So, the base case of our induction is complete. We now assume we have $n \geq n_0$, and that $T(m) \leq \mathbb{C}$, $\forall\ m < n$. We know our algorithm involves the multiplication of integers of size $3rp$ bits (in Proposition 5.3, this is what gives us the $(12T/r)\mathrm{M}(3rp)$ term), so it is a recursive algorithm. Now $3rp < 3(2T^{1/d})6b = 36T^{1/d}b$, since $r < 2T^{1/d}$ by definition. Now $T < n$, which means $T^{1/d} < n^{1/d}$, so we have $3rp < 36n^{1/d} \log n$. Now since $d = 1729$, $n^{1/d}$ is a pretty small number compared to n, as is $\log n$. Their product is certainly not close to n, and the constant 36 does little to change this since it is so small compared to n. This means that $3rp < n$, which means by the inductive hypothesis, $\mathrm{T}(3rp) \leq C$.

This in turn means that $\mathrm{T}(n) < 0.9998C + A$. Now $A \leq \frac{C}{5000} = 0.0002C$ by definition. This

means that $\mathrm{T}(n) < 0.9998C + 0.0002C = C$. This in turn means that $\mathrm{T}(n) = O(1)$. As argued above, since $\mathrm{T}(n) := \mathrm{M}(n)/n \log n$, and $\mathrm{T}(n) = O(1)$, we see that $\mathrm{M}(n) = O(n \log n)$.

Bibliography

Harvey, David and Joris van der Hoeven (2021). "Integer multiplication in time O(n log n)". In: *Annals of Mathematics* (cit. on pp. 3, 14).

Cormen, Thomas H et al. (2022). *Introduction to algorithms*. MIT press (cit. on p. 22).

Nussbaumer, H. J. and P. Quandalle (1978). "Computation of Convolutions and Discrete Fourier Transforms by Polynomial Transforms". In: *IBM Journal of Research and Development* 22.2, pp. 134–144. DOI: 10.1147/rd.222.0134 (cit. on p. 25).

Gathen, Joachim von zur and Jurgen Gerhard (2013). *Modern Computer Algebra*. 3rd. USA: Cambridge University Press. ISBN: 1107039037 (cit. on p. 46).