Utrecht University

Utrecht University

BSc Mathematics & Applications

The Strassen Algorithm and its Computational Complexity

Rein Bressers

Supervised by:

Dr. T. van Leeuwen

June 5, 2020

Abstract

Due to the many computational applications of matrix multiplication, research into efficient algorithms for multiplying matrices can lead to widespread improvements of performance. In this thesis, we will first make the reader familiar with a universal measure of the efficiency of an algorithm, its computational complexity. We will then examine the Strassen algorithm, an algorithm that improves on the computational complexity of the conventional method for matrix multiplication. To illustrate the impact of this difference in complexity, we implement and test both algorithms, and compare their runtimes. Our results show that while Strassen's method improves on the theoretical complexity of matrix multiplication, there are a number of practical considerations that need to be addressed for this to actually result in improvements on runtime.

Contents

Chapter 1

Introduction

Matrices and matrix multiplication know many computational applications. Take, for example, graphics in video games; matrices are used to project the three-dimensional scene onto a two-dimensional plane, and transformations on the scene like scaling and rotation are performed using matrix multiplication [1]. Other examples of applications of matrices and matrix multiplication include training neural networks with backpropagation [2] and solving systems of linear equations [3]. Because of the widespread use of matrices and matrix multiplication, researching efficient ways to find the product of two matrices could result in further optimization in a lot of these applications.

1.1 Computational complexity

When we multiply two 2×2 matrices, the resulting matrix is equal to

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}.$$

We see that there are $8 = 2^3$ multiplications and $4 = 2^2$ additions required to find this matrix. Generalizing for two $n \times n$ matrices, we find that calculating the resulting matrix requires n^3 multiplications and $(n-1) \cdot n^2$ additions. This means that as n increases, the number of arithmetic operations needed to find the product of two $n \times n$ matrices increases at a much higher rate than the size of the matrices.

The ratio between an algorithm's input and the resources it requires to find a solution is called the computational complexity of the algorithm. As we saw above, in the case of matrix multiplication the arithmetic computational complexity is equal to $n^3 + (n-1) \cdot n^2 = 2n^3 - n^2$. However, this "naive" approach is not optimal. In this thesis we will examine an algorithm for matrix multiplication of lower computational complexity, and conduct an experiment comparing the two approaches.

1.2 Structure

We will start this thesis by going into further detail on computational complexity, introducing the big-O notation and showing how an algorithm's computational complexity can be determined. After that, we will look into the Strassen algorithm, an algorithm used for matrix multiplication, and compare this to the naive approach. In order to illustrate the impact of the difference in computational complexity, we will implement both algorithms and test them on a number of matrices of different sizes. Lastly, after presenting and analyzing the results of the experiment, we will discuss the results found and state our conclusion.

Chapter 2

Computational Complexity

As stated in our introduction, an algorithm's computational complexity is the amount of resources required to execute the algorithm, relative to the size of its input. When we talk about computational complexity, we generally consider one of two resources, namely space and time. Space complexity concerns the amount of memory that is required to run an algorithm, whereas time complexity concerns the runtime of an algorithm. Usually, when (computational) complexity is mentioned without specifying the resource, it refers to the latter. This will also be the case for this thesis, as we will be examining the time complexity of matrix multiplication in particular.

2.1 Time complexity

Because the runtime of an algorithm can differ greatly from machine to machine, time complexity cannot just simply be expressed as the absolute runtime of the algorithm in seconds or minutes. Instead of using runtime, we look at the number of "steps" an algorithm takes, to express time complexity. Here, a step represents the execution of an operation whose runtime is not affected by the size of the input. In our introduction, we counted the number of arithmetic operations needed to find the product of two 2×2 matrices, and stated that multiplying two $n \times n$ matrices requires the execution of $2n^3 - n^2$ arithmetic operations. While the total number of operations depends on the dimension of the matrices n, the individual runtimes of these operations do not. Therefore, we say that these operations run in constant time, as their runtimes stay consistent for all values of n. By defining an algorithm's time complexity as the number of constant time operations that are executed when running the algorithm, we can formalize a function $T(n)$ for time complexity that is solely based on the input size n. This notion of time complexity is universal for all machines, as the number of steps needed to run the algorithm will be the same for every machine.

2.2 Determining an algorithm's complexity

To determine an algorithm's complexity, we need to define a function $T(n)$ that expresses the number of required constant time operations, or steps, in terms of the size of the input n. However, the number of required steps can differ greatly between inputs of the same size. Consider, for instance, an algorithm that determines whether an array of size n contains a particular value, by iteratively comparing the elements in the array to the wanted value. Comparing two values is a constant time operation, as the time required to do so is not affected by the size of the array. These comparisons are the only constant time operations, and thus we can find the complexity of this algorithm by counting the number of times the algorithm compares two values. If the wanted value is not present in the array, the algorithm will check every element in the array, requiring n comparisons, before concluding that none of the elements match the wanted value. However, if the array does contain the wanted value, the algorithm could need anywhere from 1 to n comparisons, as the value could be located anywhere in the array. Because the number of constant time operations required is not consistent over inputs of size the same size, it is unclear what $T(n)$ should be. This is why $T(n)$ is commonly defined as the maximum number of steps required over all

inputs of size n. This worst-case complexity is generally a more useful measure of overall complexity than best-case complexity, and while average-case complexity might be an even better alternative, it is often much harder to determine than worst-case complexity, making the latter a more practical choice. We can thus find $T(n)$ by identifying the constant time operations, and counting the number of times the algorithm executes these operations, for the worst-case input of size n.

In the case of naive matrix multiplication, the constant time operations consist of multiplying two matrix elements and adding up these products. If an element in either of the input matrices is equal to zero, we know that products of this element will also equal zero, and thus addition of this product is trivial. As this results in a decrease of constant time operations executed, the worst-case input is two matrices of which no element equals zero. To compute a single element in the product of these matrices, we need to take the dot product of a row in the first matrix and a column in the second matrix. As the rows and columns both contain n non-zero elements, this requires n multiplications and $n-1$ additions, adding up to $n+(n-1)=2n-1$ constant time operations. Because the output matrix contains n^2 elements, we find the complexity function of naive matrix multiplication to be $T(n)=n^2(2n-1)=2n^3-n^2$.

2.3 Asymptotic analysis

For small input sizes, a difference in complexity is often negligible. Take, for example, two algorithms with complexity functions $T_1(n)=4n^2+15n$ and $T_2(n)=n^3+5n$. When we run the algorithms on an input of size $n=10$, the difference between the $T_1(10)=550$ and $T_2(10)=1050$ operations required will be unnoticeable, as both algorithms will execute almost instantly. Furthermore, analyzing algorithms using small values of n may lead to incorrect conclusions on how the complexities of these algorithms compare. If we run the algorithms in the example above on an input of size $n=5$, we find $T_1(5)=175$ and $T_2(5)=150$, which might suggest that the first algorithm has a higher complexity than the second algorithm. However, as n grows larger and differences in complexity become more apparent, it is clear to see that T_2 will grow much faster than T_1. For these reasons, the asymptotic behavior of an algorithm's complexity, that is the behavior of the complexity function as n tends to infinity, generally gives a more useful insight into how the algorithm's complexity compares to others.

2.3.1 The big-O notation

To express the asymptotic behavior of complexity functions, we use a notation that gives an upper bound on the growth of a function, called the big-O notation. The big-O notation is defined as follows:

Definition 2.1. *The Big-O notation*

Given two real valued, non-negative functions f and g, defined on an unbound subset of the real non-negative numbers, we say that $f(x)=O(g(x))$ as x tends to infinity if there exist a positive constant c and a real number x_0 such that $f(x)\leq c\cdot g(x)$ for all x greater than or equal to x_0.

Because the big-O notation provides an upper bound on the growth of a function, we can use it to express the asymptotic behavior of complexity functions. To illustrate this, consider the complexity functions T_1 and T_2 mentioned in the previous paragraph. For $c = 5$ and $n_0 = 15$, we see that

$$T_1(n_0) = 4{n_0}^2 + 15n_0 = 5{n_0}^2 = c \cdot {n_0}^2 \text{ and } T_1(n) \leq c \cdot n^2 \text{ for all } n \geq n_0,$$

And thus follows that $T_1(n) = O(n^2)$. In the same manner, we find $T_2(n) = O(n^3)$, which makes it clear to see that second algorithm is of higher complexity than the first algorithm.

By only considering the most dominating term of the complexity function as n grows larger, the big-O notation makes it much easier to analyze algorithms and compare their complexities. However, ignoring the lower-order terms and constant factors means that it is possible that an algorithm of high order with small constants outperforms an algorithm of low order with large constants, for most, or all, inputs encountered in practice. Despite this, asymptotic analysis and the big-O notation remain the conventional method to discuss and compare algorithms and their respective complexities. As for naive matrix multiplication, we found a complexity function of $T(n) = 2n^3 - n^2$, and it is therefore that we say naive matrix multiplication has a complexity of $O(n^3)$.

2.4 Recurrent complexity functions

As we saw in section 2.2, we can find an algorithm's complexity function by counting the number of constant time operations it executes, and we can use this function to determine an algorithm's (asymptotic) complexity. However, this does not always suffice. Consider, for example, the following sorting algorithm, called Merge Sort:

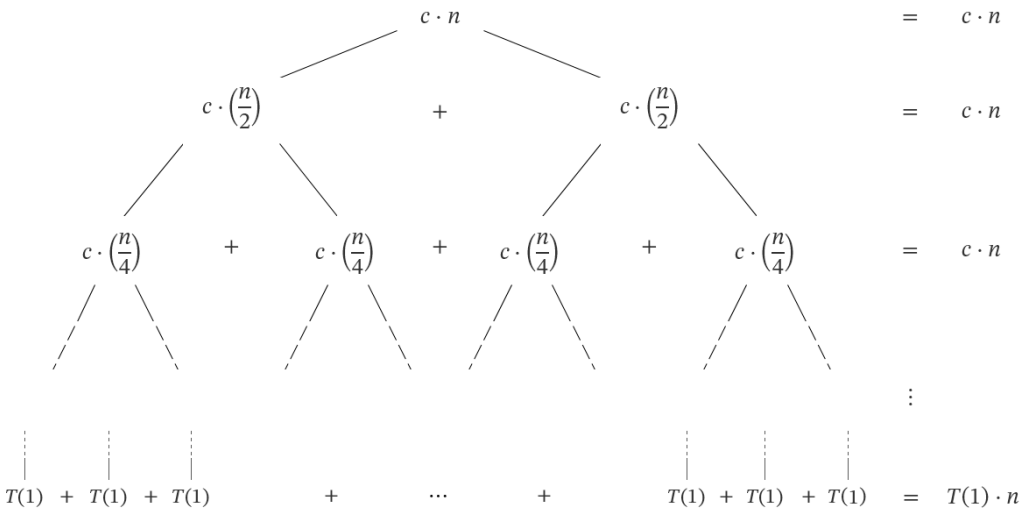
Algorithm 1. *Merge Sort*[1]

```
 1. function MergeSort(A)
 2.   n ← length(A)
 3.   if n ≤ 1 then
 4.     return A
 5.
 6.   left  ← A[0:n/2]
 7.   right ← A[n/2:n]
 8.
 9.   left  ← MergeSort(left)
10.   right ← MergeSort(right)
11.   return Merge(left, right)
```

[1]Here, `A[i:j]` represents the subarray consisting of the elements in `A` with indices i through $j-1$. The function `Merge(B, C)` merges two arrays into one, by iteratively taking the lowest value from `B` and `C`.

First, the algorithm checks for the base case. If the input array is empty or only contains one element, it is consequently sorted, and the algorithm returns the array unmodified. For input arrays containing more than one element, the algorithm splits the array in two, sorts the two halves separately, merges the sorted arrays, and returns the merged array. To find the complexity function $T(n)$ of this algorithm, we need to determine the complexities of these steps individually. For splitting and merging the array, this is fairly straightforward. Splitting the input array requires placing the n elements in two separate arrays, thus having a complexity of $O(n)$. The `Merge` function iterates over the two sorted subarrays, which results in a complexity of $O(n)$ as well.

Determining the complexity of sorting the two subarrays proves a bit more difficult. In order to sort the subarrays, the algorithm uses Merge Sort, thus recursively calling on itself. We do not know the complexity of Merge Sort, however, as this is what we are actually trying to formalize. Because of this, we can only define $T(n)$ recursively. We know that $T(n)$ is the complexity function for running Merge Sort on an input of size n, and therefore, the complexity function for sorting a subarray of size $\frac{n}{2}$ is equal to $T\left(\frac{n}{2}\right)$. Combining this with the complexities of splitting and merging the array, we find the recurrence relation $T(n) = 2T\left(\frac{n}{2}\right) + c \cdot n$, for some constant c, with $T(1) = O(1)$ for the base case. To better understand this expression, we can construct a recursion tree for $T(n)$:



The nodes in this tree represent the recursive calls to T, and for each node, the complexity of splitting the input array and merging the two sorted subarrays is specified. The leaves represent the recursive calls on arrays of size 1, i.e. the base case. Note that at any depth d, there are 2^d nodes representing a call on a subarray of size $\frac{n}{2^d}$, or a call on the base case. Since $\frac{n}{2^{\log_2 n}} = \frac{n}{n} = 1$, we find the leaves at a depth of $d = \log_2 n$, with a total complexity of $T(1) \cdot 2^d = T(1) \cdot 2^{\log_2 n} = T(1) \cdot n$. For depths d smaller than $\log_2 n$, the nodes have an associated complexity of $c \cdot \left(\frac{n}{2^d}\right)$, and thus the complexity at these depths is equal to $c \cdot \left(\frac{n}{2^d}\right) \cdot 2^d = c \cdot n$. The combined complexity of the $\log_2 n$ layers of internal nodes, paired with the complexity of the layer of leaves, therefore result in a complexity function of $T(n) = c \cdot n \cdot \log_2 n + T(1) \cdot n$.

As we have now explicitly defined $T(n)$, we can express Merge Sort's asymptotic behavior using the big-O notation. Since $T(1) \cdot n = O(1) \cdot n = O(n) = c \cdot n$, the dominating term in $T(n)$ is $c \cdot n \cdot \log_2 n$, and we find Merge Sort has a complexity of $O(n \cdot \log n)$. Here, the base of the logarithm is intentionally left unspecified, as it does not affect the function's asymptotic growth. This can be easily shown using the "change of base" rule for logarithms. This rule states that, given two real numbers a and b, the equality $\log_b n = \log_a n / \log_a b$ holds, from which follows that $\log_a n = \log_a b \cdot \log_b n$. Thus, we find that two logarithms of n, to any two bases, differ only by a constant factor $\log_a b$. This means that $O(\log_b n)$ is equivalent to $O(\log_a n)$ for any pair of real numbers a and b, and we conclude that the bases of the logarithmic terms in a function are irrelevant to the function's asymptotic growth.

2.4.1 The Master Theorem

Algorithms that solve problems by using recursion in the same manner as Merge Sort, are called *divide-and-conquer* algorithms. Instead of solving a problem iteratively, divide-and-conquer algorithms solve a problem by splitting it into subproblems, and combining the solutions of these subproblems, which are found by solving the subproblems recursively. Because of their recursive nature, the complexity functions of these algorithms typically consist of a recurrence relation of a generic form; for an algorithm that divides a problem of size n into $a \geq 1$ subproblems of size $\frac{n}{b}$, with $b > 1$, and $f(n)$ the complexity function for splitting and recombining the problem, we find a complexity function $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$. Like we saw in the case of Merge Sort, there are two factors that contribute to this complexity function, namely the complexity of splitting and recombining the problem, and the complexity of the calls on the base case, specified respectively at the internal nodes and the leaves of the recursion tree for the complexity function. As we are concerned with the asymptotic growth of complexity, we can distinguish the following three cases:

1. The complexity at the leaves is dominant over the complexity at the nodes,
2. The complexity at the leaves is comparable to the complexity at the nodes, (2.1)
3. The complexity at the leaves is dominated by the complexity at the nodes,

with a term $O(g(n))$ being dominant over a term $O(h(n))$ when $g(n)$ is polynomially larger than $h(n)$, i.e. there exist an $\varepsilon > 0$ such that $h(n) = O(g(n) \cdot n^{-\varepsilon})$. These three cases form the basis for a general method for finding the complexity of divide-and-conquer algorithms. The "Master Theorem", first introduced in 1980 by Jon Bentley, Dorothea Haken and James B. Saxe [4], provides asymptotic bounds on the complexities of divide-and-conquer algorithms, using the generality of their complexity functions.

Theorem 2.1. *The Master Theorem* [5]

Given constants $a \geq 1$, $b > 1$, and function $f(n)$, we distinguish the following three cases for the asymptotic bound on the recurrence relation of the form $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$, for some constant $\varepsilon > 0$, then $T(n) = O(n^{\log_b a})$,
2. If $f(n) = O(n^{\log_b a})$, then $T(n) = O(n^{\log_b a} \cdot \log n)$,
3. If $f(n) = O(n^{\log_b a + \varepsilon})$, for some constant $\varepsilon > 0$, and if $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ for some constant $c < 1$, and all sufficiently large n, then $T(n) = O(f(n))$.

Proof. To distinguish between cases (2.1), we need to find the complexities at both the nodes and the leaves of the recursion tree for $T(n)$. At a depth d in the recursion tree, there will be a^d nodes, with the non-leave nodes representing recursive calls on subproblems of size $\frac{n}{b^d}$. This is because the initial problem has then been divided d times, each time creating a subproblems and reducing the size of the problem by a factor b. After dividing the problem $\log_b n$ times, these subproblems will be of size $\frac{n}{b^{\log_b n}} = 1$, at which point they cannot be divided any further. This results in a maximum depth of the tree of $\log_b n$, and in the case where the entirety of leaves in the tree is located at this depth, we find the tree contains at most $a^{\log_b n}$ leaves, along with a^d non-leave nodes at depths d ranging from 0 through $\log_b n - 1$, therefore representing a maximum number of $a^{\log_b n}$ executions of the base case. Since the leaves represent calls on the base case, As calls on the base case only occur for (sub)problems of sizes below a certain threshold, the worst case complexity of a single call on the base case does not depend on the initial input size n, which means that execution of the base case has a complexity of $O(1)$. We therefore find the combined complexity at the leaves of the recursion tree to be equal to $a^{\log_b n} \cdot O(1) = O(a^{\log_b n}) = O(n^{\log_b a})$.

The complexity associated with each of the internal nodes is equal to that of splitting and recombining the subproblem at that node. Thus, each of the nodes at depth d, for $d \in \{0, \dots, \log_b n - 1\}$, will have an individual complexity function of $f\left(\frac{n}{b^d}\right)$. As the number of nodes at these depths is equal to a^d, we find a complexity function of

$$\sum_{d=0}^{\log_b n - 1} a^d \cdot f\left(\frac{n}{b^d}\right) \tag{2.2}$$

for the combined complexity of the internal nodes in the recursion tree. For each of the three cases of the Master Theorem, we will provide asymptotic bounds for this summation, by substituting the corresponding bound on $f(n)$ into equation (2.2). We can then combine these complexities with the complexity at the leaves of the recursion tree, to determine asymptotic bounds on $T(n)$, thereby proving each of the cases of the Master Theorem individually.

For case (1) of the Master Theorem, we substitute $f(n) = n^{\log_b a - \varepsilon}$ into (2.2), which results in the following expression:

$$\begin{aligned}
\sum_{d=0}^{\log_b n - 1} a^d \cdot \left(\frac{n}{b^d}\right)^{\log_b a - \varepsilon} &= n^{\log_b a - \varepsilon} \cdot \sum_{d=0}^{\log_b n - 1} \left(\frac{ab^\varepsilon}{b^{\log_b a}}\right)^d \\
&= n^{\log_b a - \varepsilon} \cdot \sum_{d=0}^{\log_b n - 1} (b^\varepsilon)^d \\
&= n^{\log_b a - \varepsilon} \cdot \left(\frac{b^{\varepsilon \cdot \log_b n} - 1}{b^\varepsilon - 1}\right) \\
&= n^{\log_b a - \varepsilon} \cdot \left(\frac{n^\varepsilon - 1}{b^\varepsilon - 1}\right).
\end{aligned}$$

As b and ε are constants, we find an asymptotic bound of $O\left(n^{\log_b a - \varepsilon}\right) \cdot O(n^\varepsilon) = O\left(n^{\log_b a}\right)$. Since this is equal to the complexity at the leaves of the tree, we find an asymptotic bound of $T(n) = 2 \cdot O\left(n^{\log_b a}\right) = O\left(n^{\log_b a}\right)$, equal to the bound provided in first case of the Master Theorem.

For second case of the Master Theorem, we substitute $f(n) = n^{\log_b a}$ into (2.2):

$$\begin{aligned}
\sum_{d=0}^{\log_b n - 1} a^d \cdot \left(\frac{n}{b^d}\right)^{\log_b a} &= n^{\log_b a} \cdot \sum_{d=0}^{\log_b n - 1} \left(\frac{a}{b^{\log_b a}}\right)^d \\
&= n^{\log_b a} \cdot \sum_{d=0}^{\log_b n - 1} 1 \\
&= n^{\log_b a} \cdot \log_b n.
\end{aligned}$$

This means that at every depth, the combined complexity of the nodes is $O\left(n^{\log_b a}\right)$, which is equal to the complexity at the leaves of the tree, which results in a total complexity of $O\left(n^{\log_b a}\right) \cdot (\log_b n + 1)$. Since we write $O(\log n)$ for the asymptotic bound on a logarithm, regardless of its base, we find $T(n) = O\left(n^{\log_b a}\right) \cdot O(\log n) = O(n^{\log_b a} \cdot \log n)$, thus proving case (2) of the Master Theorem.

The third case applies for $f(n) = O(n^{\log_b a + \varepsilon})$, and includes an additional condition. We assume this condition has been met, such that for some constant $c < 1$ and sufficiently large n, we have $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$. If we divide both sides by a, we find $f\left(\frac{n}{b}\right) \leq \frac{c}{a} \cdot f(n)$, and iterating this d times gives us $f\left(\frac{n}{b^d}\right) \leq \left(\frac{c}{a}\right)^d \cdot f(n)$ and $a^d \cdot f\left(\frac{n}{b}\right) \leq c^d \cdot f(n)$, assuming all values iterated over are sufficiently large. We substitute this into equation (2.2), and add $O(1)$ to cover the terms not covered by our assumption that n is sufficiently large:

$$\begin{aligned}\sum_{d=0}^{\log_b n - 1} a^d \cdot f\left(\frac{n}{b^d}\right) &\leq O(1) + \sum_{d=0}^{\log_b n - 1} c^d \cdot f(n) \\ &\leq O(1) + f(n) \cdot \sum_{d=0}^{\infty} c^d \\ &= O(1) + f(n) \cdot \left(\frac{1}{1-c}\right).\end{aligned}$$

Because c is a constant, we find an asymptotic (upper) bound of $O(f(n))$. This means that, since $f(n) = O(n^{\log_b a + \varepsilon})$, the complexity at the internal nodes of the tree dominates the complexity at the leaves. It follows that $T(n) = O(f(n)) + O\left(n^{\log_b a}\right) = O(f(n))$, which proves case (3) and completes our proof of the Master Theorem.

Because the summation in (2.2) is not defined for non-integer values of $\log_b n - 1$, this proof only considers the cases where n is an exact power of b. However, as the Master Theorem provides asymptotic upper bounds, floor and ceiling functions can be used to extend this proof, so that it includes all other cases as well [6].

Chapter 3

The Strassen Algorithm

In 1969, mathematician Volker Strassen published a paper [7] in which he provided an algorithm for matrix multiplication of sub-cubic complexity, thus proving that naive matrix multiplication, having a complexity of $O(n^3)$, is not optimal. The basis of the algorithm presented in this paper is a divide-and-conquer approach for matrix multiplication, where the two input matrices are split into submatrices, and the output matrix is constructed by recursively calculating and combining products of these submatrices. The most elementary way of doing this has a complexity of $O(n^3)$, which is no better than the iterative approach. Strassen, however, managed to devise a method to construct the output matrix that requires multiplying fewer pairs of submatrices, thus reducing the number of recursive calls and lowering the complexity of the algorithm. In this chapter, we will first expand on the "basic" divide-and-conquer approach for matrix multiplication, after which we will present Strassen's alternative, and determine and compare their respective complexities.

3.1 Divide-and-conquer approach

To define a divide-and-conquer algorithm for matrix multiplication, we need to break down the problem into subproblems, i.e. smaller instances of matrix multiplication, and construct the output matrix using the solutions to these subproblems. This can be done through a process called block-matrix multiplication.

3.1.1 Block-matrix multiplication

A block-matrix is a representation of a matrix that has been partitioned into submatrices, or "blocks". We can interpret the block-matrix representation of a matrix by imagining a grid being projected on the matrix, dividing it into blocks. These blocks, and their associated submatrices, can be referred to using the indices of the rows and columns in the grid. For a matrix A, and a block-matrix representation of A consisting of q rows and r columns of blocks, we write

$$A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1r} \\ A_{21} & A_{22} & \cdots & A_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ A_{q1} & A_{q2} & \cdots & A_{qr} \end{pmatrix}$$

where A_{xy} refers to the submatrix corresponding to the block in the xth row and yth column of the block-matrix.

Definition 3.1. *Block-matrix multiplication*

Let A and B be matrices with block-matrix representations of respective sizes $q \times r$ and $r \times p$. If these block-matrices are compatible for multiplication, their product C is defined as follows:

$$C = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1r} \\ A_{21} & A_{22} & \cdots & A_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ A_{q1} & A_{q2} & \cdots & A_{qr} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{r1} & B_{r2} & \cdots & B_{rp} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1p} \\ C_{21} & C_{22} & \cdots & C_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ C_{q1} & C_{q2} & \cdots & C_{qp} \end{pmatrix}$$

with $C_{xy} = \sum_{k=1}^{r} A_{xk} B_{ky}$ for $x \in \{1, \ldots, q\}$ and $y \in \{1, \ldots, p\}$.

To multiply block-matrices, we use products of their submatrices. Therefore, for two block-matrices to be compatible for multiplication, all corresponding submatrices need to be compatible for multiplication as well.

Theorem 3.1. *Let A and B be matrices with block-matrix representations compatible for multiplication. The product of these block-matrices is in turn a block-matrix, representing the product AB.*

Proof. For the purpose of this thesis, it is only necessary to prove this for the case of two square matrices of even dimension, partitioned into four blocks of equal size. Let A and B be matrices, both of dimension $2n \times 2n$, with elements a_{ij} and b_{ij} for $i, j \in \{1, \dots, 2n\}$. Partition both A and B into four submatrices of size $n \times n$, and let

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \text{ and } B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{3.1}$$

be the block-matrix representations for A and B corresponding to this partition. Without loss of generality, we can formalize the correspondence between A and its block-matrix representation using the equality $(A_{xy})_{st} = a_{ij}$ (3.2), where $i = n \cdot (x-1) + s$ and $j = n \cdot (y-1) + t$. For $x, y \in \{1, 2\}$ and $s, t \in \{1, \dots, n\}$, each of the $4n^2$ unique combinations of (x, y, s, t) represents an element in the block-matrix, and maps to a distinct pair (i, j), with $i, j \in \{1, \dots, 2n\}$, thus representing the elements of A.

We will prove that the product C of the block-matrices representing A and B (3.1) is in turn a block-matrix representation of the product of A and B, by showing that the equality (3.2) holds for the elements in C and AB as well. By the definition of block-matrix multiplication, we find the product C of the block-matrix representations for A and B to be equal to:

$$C = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{11}B_{12} + A_{12}B_{22} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Given $x, y \in \{1, 2\}$ and $s, t \in \{1, \dots, n\}$, the element $(C_{xy})_{st}$ in C is equal to

$$(C_{xy})_{st} = (A_{x1}B_{1y} + A_{x2}B_{2y})_{st} = (A_{x1}B_{1y})_{st} + (A_{x2}B_{2y})_{st}.$$

By the definition of matrix multiplication, we have

$$(C_{xy})_{st} = \textstyle\sum_{k=1}^{n} (A_{x1})_{sk} (B_{1y})_{kt} + \sum_{k=1}^{n} (A_{x2})_{sk} (B_{2y})_{kt},$$

and using equality (3.2), we find

$$\begin{aligned} (C_{xy})_{st} &= \sum_{k=1}^{n} a_{ik} b_{kj} + \sum_{k=1}^{n} a_{i(n+k)} b_{(n+k)j} \\ &= \sum_{k=1}^{n} a_{ik} b_{kj} + \sum_{k=n+1}^{2n} a_{ik} b_{kj} \\ &= \sum_{k=1}^{2n} a_{ik} b_{kj} \\ &= (AB)_{ij}, \end{aligned}$$

with $i = n \cdot (x-1) + s$ and $j = n \cdot (y-1) + t$.

3.1.2 Naive algorithm

Because block-matrix multiplication allows us to find the product of two matrices by using the products of smaller submatrices, we can now define a divide-and-conquer algorithm for the multiplication of two $2^k \times 2^k$ matrices:

Algorithm 2. *D&Q Matrix Multiplication* [5]

```
 1. function MatrixMultiplicationDAQ(A, B)
 2.   n ← dim(A)
 3.   C ← new n×n matrix
 4.   if n == 1 then
 5.     C ← (a11 · b11)
 6.     return C
 7.   else partition A and B into four (n/2)×(n/2) quadrants, as in (3.1)
 8.   C11 ← MatrixMultiplicationDAQ(A11, B11)
          + MatrixMultiplicationDAQ(A12, B21)
 9.   C12 ← MatrixMultiplicationDAQ(A11, B12)
          + MatrixMultiplicationDAQ(A12, B22)
10.   C21 ← MatrixMultiplicationDAQ(A21, B11)
          + MatrixMultiplicationDAQ(A22, B21)
11.   C22 ← MatrixMultiplicationDAQ(A21, B12)
          + MatrixMultiplicationDAQ(A22, B22)
12.   C   ← ( C11  C12 )
             ( C21  C22 )
12.   return C
```

The base case for this algorithm is the multiplication of two 1×1 matrices, in which case the algorithm returns a 1×1 matrix containing the product of the two elements in the input matrices. If the input matrices are of a higher dimension, the algorithm divides them into four equally sized quadrants, and uses products of these quadrants, which it finds recursively, to construct and combine the quadrants of the output matrix.

Because this is a divide-and-conquer algorithm, its complexity function is of the familiar form $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$, for an algorithm that divides the problem in a subproblems of size $\frac{n}{b}$, with $f(n)$ the complexity of splitting and recombining the problem. In the algorithm described above, each of the four quadrants in the output matrix is found by summing the results of two recursive calls on submatrices of dimension $(n/2) \times (n/2)$, thus dividing the problem of matrix multiplication into 8 subproblems of half the size of the input problem, and we find $a = 8$ and $b = 2$. Partitioning the input matrices, summing two products of submatrices, and constructing the output matrix all require iterating over the elements in the associated matrices, therefore having a complexity of $O(n^2)$. These actions are performed a constant number of times, regardless of the size of the input matrices; two matrices are divided into submatrices, four matrix additions are used to find the quadrants of the output matrix, and the output matrix is constructed only once. Therefore, the complexity of splitting and recombining the problem is equal to $f(n) = c \cdot O(n^2) = O(n^2)$, and we find a complexity function of $T(n) = 8 \cdot T\left(\frac{n}{2}\right) + O(n^2)$.

Note that for input matrices of dimensions not equal to a power of two, we can simply add rows and columns containing only zeros to these matrices, until it is possible to divide them into four equally sized submatrices. This "padding" of the input matrices allows us to use the above algorithm on inputs of non-square or uneven dimensions as well, whose product can be found after stripping the output matrix of the rows and columns padded with zeros.

3.2 Strassen's algorithm

Rather surprisingly, Strassen managed to reduce the number of recursive calls needed to construct the output matrix. In his paper, Strassen showed that the product of two 2×2 matrices can be found using 7 multiplications and 18 addition or subtractions, as opposed to the 8 multiplication and 4 additions used when multiplying the matrices in the "traditional" manner. Strassen realized that this method can be used to find the product of two 2×2 block-matrices as well, as block-matrix multiplication is equivalent to regular matrix multiplication, with the elements in the matrices representing the blocks in the block-matrices. Given matrices A and B, with block-matrix representations of the form described in (3.1), Strassen defined the following matrices:

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22}) \cdot B_{11} \\
M_3 &= A_{11} \cdot (B_{12} - B_{22}) \\
M_4 &= A_{22} \cdot (B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12}) \cdot B_{22} \\
M_6 &= (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22})
\end{aligned}
$$

He then used these matrices to construct a block-matrix representation for the product of A and B. For $C = AB$, Strassen found a corresponding block-matrix representation $C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$, where

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{aligned}
$$

Strassen modified the naive divide-and-conquer algorithm for matrix multiplication, described in section 3.1.2, by replacing the 8 multiplications and 4 additions used to construct the output matrix with the factoring scheme above. This resulted in the following algorithm for matrix multiplication, which later became known as the *Strassen algorithm:*

Algorithm 3. *The Strassen Algorithm*

```
1.  function Strassen(A, B)
2.    n ← dim(A)
3.    C ← new n×n matrix
4.    if n == 1 then
5.      C ← (a_11 · b_11)
6.      return C
7.    else partition A and B into four (n/2)×(n/2) quadrants, as in(3.1)
8.    M_1 ← Strassen(A_11 + A_22, B_11 + B_22)
9.    M_2 ← Strassen(A_21 + A_22, B_11)
10.   M_3 ← Strassen(A_11, B_12 − B_22)
11.   M_4 ← Strassen(A_22, B_21 − B_11)
12.   M_5 ← Strassen(A_11 + A_12, B_22)
13.   M_6 ← Strassen(A_21 − A_11, B_11 + B_12)
14.   M_7 ← Strassen(A_12 − A_22, B_21 + B_22)
15.   C_11 ← M_1 + M_4 − M_5 + M_7
16.   C_12 ← M_3 + M_5
17.   C_21 ← M_2 + M_4
18.   C_22 ← M_1 − M_2 + M_3 + M_6
19.   C ← ( C_11  C_12 )
          ( C_21  C_22 )
20.   return C
```

This algorithm finds the product of two matrices using 7 recursive calls on submatrices of dimension $(n/2)\times(n/2)$, along with 18 matrix additions or subtractions. This means that for its complexity function, likewise of the form $T(n) = a \cdot T\left(\frac{n}{b}\right) + g(n)$, we find $a = 7$ and $b = 2$, since the algorithm divides the problem of matrix multiplication into only 7 subproblems of size $\frac{n}{2}$. The 18 matrix additions and subtractions required by the Strassen algorithm, compared to the 4 matrix additions required by the naive algorithm, do result in a higher complexity for splitting and recombining the problem. However, as the number of matrix additions and subtractions required by the algorithms is not affected by the size of the inputs, these complexities differ by a constant factor c, and we therefore find $g(n) = c \cdot f(n) = c \cdot O(n^2) = O(n^2)$. Thus, the complexity function for the Strassen algorithm is equal to $T(n) = 7 \cdot T\left(\frac{n}{2}\right) + O(n^2)$.

3.3 Complexity

As both algorithms described above are divide-and-conquer algorithms, we can use the Master Theorem to determine their complexities. For the naive divide-and-conquer algorithm, we found a complexity function of $T(n) = 8 \cdot T\left(\frac{n}{2}\right) + O(n^2)$. Since $f(n) = O(n^2) = O(n^{3-1}) = O\left(n^{\log_2 8 - 1}\right)$, this corresponds to the first case of the Master Theorem,. We therefore find a complexity of $O\left(n^{\log_2 8}\right) = O(n^3)$, equal to complexity we found for the naive, iterative algorithm.

The Strassen algorithm manages to improve on this complexity, by finding the product of two matrices using only 7 recursive calls, instead of 8. We found a complexity function for the Strassen algorithm equal to $T(n) = 7 \cdot T\left(\frac{n}{2}\right) + O(n^2)$, and since $O(n^2) = O\left(n^{\log_2 4}\right) = O\left(n^{\log_2 7 - \varepsilon}\right)$ for $\varepsilon = \log_2\left(\frac{7}{4}\right) > 0$, this corresponds to the first case of the Master Theorem as well. However, because of the reduced number of recursive calls, we find a sub-cubic complexity for the Strassen algorithm, equal to $O\left(n^{\log_2 7}\right) \approx O(n^{2.81})$.

3.4 Practical considerations

Keep in mind that improving on an algorithm's complexity does not always imply an improvement of practical performance, since high constant factors could result in implementations of this "improved" algorithm only reducing runtimes for problems of sizes beyond those encountered in practice. Therefore, when evaluating the practical uses of the Strassen algorithm, it is important that, besides asymptotic complexity, we also consider the impact of the 18 matrix additions and subtractions performed at each recursive calls on the runtime of the algorithm.

Asymptotically, the complexity of matrix addition and subtraction is dwarfed by the complexity of the recursive calls on the Strassen algorithm, meaning that as the dimension of input matrices tends to infinity, the impact of these operations on the algorithm's performance will be negligible. However, for input matrices of sufficiently low dimensions, the decrease in complexity does not compensate for the additional arithmetic operations required by the 18 matrix additions and subtractions performed at these calls. This becomes apparent when we consider two input matrices of dimension $n = 8$; the $18 \cdot n^2 = 18 \cdot 8^2 = 1152$ arithmetic operations performed at the very initial call on the Strassen algorithm already surpasses the naive algorithm's total of $2n^3 - n^2 = 2 \cdot 8^3 - 8^2 = 960$ arithmetic operations, with subsequent recursive calls only further increasing this disparity. When multiplying matrices of low enough dimensions, this can result in a significant difference in efficiency in favor of naive matrix multiplication. Furthermore, since the algorithm recurses down to submatrices of dimension 1, these inefficient calls on low dimension inputs will inevitably occur for higher dimension inputs as well. Because the number of recursive calls is increased sevenfold at each level of recursion, the number of inefficient calls grows rapidly with the dimension of the input matrices, raising the point at which the Strassen algorithm outperforms naive matrix multiplication.

3.4.1 Hybrid variant

To make effective use of the lower complexity of the Strassen algorithm, it is common for practical implementations to use a "hybrid" of the two algorithms. Since there exist some threshold for which the naive approach is likely to be faster than the Strassen algorithm, we can replace the inefficient recursive calls, on submatrices of dimensions lower than this threshold, with calls on the naive algorithm. As mentioned in the previous section, the number of inefficient calls on the Strassen algorithm grows rapidly with the dimension of the input matrices, which means that a relatively small increase of the efficiency of these calls could result in a substantial improvement of the overall performance of the Strassen algorithm. By stopping recursion early and switching to naive matrix multiplication, we essentially extend the base-case of the Strassen algorithm to include all inputs of dimensions lower than some n_0, and for its execution run the naive algorithm on these inputs. Note that, therefore, the traditional Strassen approach and this variant differ solely in the definitions of their base-cases, and we can thus define the algorithm for this hybrid variant by altering only lines 4 and 5 in Algorithm 3. Using the multiplication operator to represent naive matrix multiplication, we this results in the following algorithm:

Algorithm 4. *Hybrid variant*

```
 1. function StrassenHV(A, B)
 2.   n ← dim(A)
 3.   C ← new n×n matrix
 4.   if n < n_0 then
 5.     C ← A·B
 6.     return C
 7.   else partition A and B into four (n/2)×(n/2) quadrants, as in (3.1)
 8.   M_1 ← StrassenHV(A_11 + A_22, B_11 + B_22)
 9.   M_2 ← StrassenHV(A_21 + A_22, B_11)
10.   M_3 ← StrassenHV(A_11, B_12 − B_22)
11.   M_4 ← StrassenHV(A_22, B_21 − B_11)
12.   M_5 ← StrassenHV(A_11 + A_12, B_22)
13.   M_6 ← StrassenHV(A_21 − A_11, B_11 + B_12)
14.   M_7 ← StrassenHV(A_12 − A_22, B_21 + B_22)
15.   C_11 ← M_1 + M_4 − M_5 + M_7
16.   C_12 ← M_3 + M_5
17.   C_21 ← M_2 + M_4
18.   C_22 ← M_1 − M_2 + M_3 + M_6
19.   C ← ( C_11  C_12 )
          ( C_21  C_22 )
20.   return C
```

Using this hybrid variant, we are able to reduce the recursion overhead of the Strassen algorithm, which is very likely to result in an improvement of runtime. The extent of this potential improvement, however, depends heavily on our choice of n_0. For high values of n_0, the algorithm might resort to naive matrix multiplication too early, not taking full advantage of the Strassen algorithm's lower complexity. For low values of n_0, on the other hand, the algorithm might use recursive calls on inputs for which naive matrix multiplication is faster, resulting in suboptimal runtimes. Thus, using an appropriate value of n_0 is essential for maximizing the improvement of runtime over the traditional Strassen algorithm. Note that, as we are concerned with runtime, appropriate values of n_0 can differ heavily between machines and implementations, and optimal values have to be determined empirically.

3.4.2 Cross-over point

A suitable value of n_0 would be the so called *cross-over* point. This is the input dimension n, for which performing one iteration of the Strassen algorithm, and subsequently using the naive algorithm for multiplying the submatrices, results in runtimes that are on par with those for naive matrix multiplication of $n \times n$ matrices. At this point, the improvement on complexity, compared to the naive algorithm, makes up for the additional matrix additions and subtractions used for one iteration of the Strassen algorithm. Due to their difference in complexity, the runtimes for naive matrix multiplication grow at a faster rate than those for the Strassen approach. Therefore, beyond this cross-over point, performing one iteration of the Strassen algorithm before applying naive matrix multiplication will result in lower runtimes compared to exclusively using the naive algorithm. This, in turn, means that if the dimension of the submatrices resulting from one iteration of the Strassen algorithm is again greater than this n, another recursive iteration would improve on runtime even further. Thus, we want to perform recursive iterations of the Strassen algorithm until the dimensions of the resulting submatrices fall below this point, after which we switch to the naive algorithm instead, which makes the cross-over point an ideal candidate for the value of n_0.

Chapter 4

Experiment

For the remainder of this thesis, we conduct an experiment to determine how the difference in complexity between Strassen's approach and the naive algorithm affects practical performance of computational matrix multiplication. We do so by implementing both algorithms, and comparing their respective runtimes for inputs of varying sizes. To illustrate the impact on performance brought on by the additional matrix additions and subtractions needed for the Strassen algorithm, we first compare runtimes for the variant of the Strassen algorithm described in section 3.4.1, using multiple values of n_0. Additionally, this provides us with a general idea of the cross-over point, which we can subsequently use for our comparison between naive matrix multiplication and the Strassen approach. In this chapter, we will first expand on our implementations of the algorithms, and specify the parameters of the experiment.

4.1 Implementations

For our experiment, we used *C#* to implement and test the matrix multiplication algorithms. We first implemented an object to represent our test matrices, which we defined using a jagged array, whose size corresponds to the dimensions of the matrix it represents. We can construct an object either representing a matrix containing only zeroes, or a randomly generated matrix. After construction, the values for individual elements can be set using row and column indices, or a jagged array can be used to set the entirety of the elements at once. We added methods for the partitioning of a matrix into equally sized quadrants, which returns objects representing these four submatrices, and made it possible to set the elements of a matrix object using such a block-matrix representation. We included definitions for the addition and subtraction operators, using conventional iterative methods for matrix addition and subtraction, and wrote an implementation of naive matrix multiplication for the multiplication operator.

By using this object to represent our test matrices, we were able to write an implementation for the Strassen algorithm that closely resembles the pseudo-code we presented in Algorithms 3 and 4. Note that for $n_0 = 2$, the hybrid variant of the Strassen algorithm is equivalent to the traditional algorithm, which allows us to use a single implementation of Algorithm 4 for testing both variants.

Exact descriptions of the code used for our implementations can be found at **https://git.science.uu.nl/r.j.bressers/thesismatrixmultiplication**.

4.2 Parameters

The first part of our experiment consists of a comparison between the performances of naive matrix multiplication, the traditional Strassen approach, and the hybrid variant of the Strassen algorithm, on input matrices of dimensions 32, 64, 96, 128 and 160. For the hybrid variant, we set n_0 to be equal to the dimension of the test matrices, to narrow down the cross-over point for our particular implementation and machine.

In the second part of our experiment, we test naive matrix multiplication and the Strassen algorithm on a greater range of input dimension for a more thorough comparison of practical performance, with input matrices of dimensions 64, 128, 256, 512, 1024 and 2048. We use the results found in the first part of the experiment to set the value of n_0, taking the dimension for which the difference in performance between the naive algorithm and the hybrid variant is minimal. If any of our intended test dimensions happens to be smaller than the value found for n_0, we will not include it in our experiment since the Strassen implementation will be equivalent to the naive algorithm for input dimensions lower than n_0.

We measure the performance of the algorithms by recording the runtimes of their corresponding implementations. In our first experiment, we run the implementations on 150 pairs of input matrices of size $n \times n$, for each test dimension n mentioned above, and average over their respective runtimes. For our second experiment, we use matrices of much greater dimension. As dimension grows, differences in complexity become more evident, which reduces the need for a large test set. Since runtimes can drastically increase with dimension, we have therefore chosen to use test sets of sizes 150, 100, 75, 50 and 25, respectively for the test dimensions in our second experiment. To ensure a fair comparison, we use the same set of test matrices for all implementations, consisting of randomly generated matrices with elements taking integer values ranging between 1 and 10.000.

Chapter 5

Results

In this chapter, we will present the results of our experiment and give an analysis of these results.

5.1 Experiment I

The table below shows the results of the first part of our experiment. Each column is headed by the dimension of the tested input matrices, under which the average runtimes on these inputs is stated. The furthermost left column of the table specifies the implementation for which we found the given averages. Because some of the runtimes were too low to accurately record, we did not include them in this table.

Table 5.1. *Results of experiment I*

		Average runtime (ms)				
Algorithm	n	32	64	96	128	160
Naive		-	3,29	10,34	25,36	48,6
Strassen	- $n_0 = 2$	20,8	82,35	518,94	542,02	3525,25
	- $n_0 = n$	-	3,54	10,24	23,97	45,34

As we can clearly see in Table 5.1, the traditional Strassen algorithm performs orders of magnitude worse than the naive and the hybrid algorithms on matrices of these sizes. To determine the point after which the Strassen algorithm outperforms naive matrix multiplication, we can define functions for the approximate runtimes of our implementations. Using the averages we found, we can determine coefficients for the asymptotic complexities of the algorithms, as well as coefficients for the additional quadratic and constant factors impacting runtime. These coefficients allow us to formulate for each of the algorithms a function, that, given a dimension n, returns the expected runtime on our machine for inputs of this dimension. This resulted in the following plot for the expected runtimes of the algorithms:

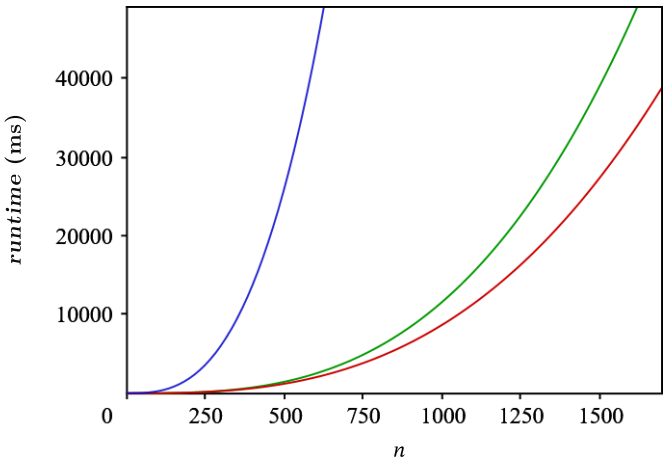


Figure 5.1 [8]. *Expected runtimes for the traditional Strassen algorithm* (blue), *naive matrix multiplication* (green), *and the hybrid variant using* $n_0 = n$ (red).

The functions we found for the Strassen algorithm and naive matrix multiplication intersected at a dimension of around $2{,}42 \cdot 10^9$, meaning that the traditional Strassen only outperforms naive matrix multiplication after a point way beyond practical use. The hybrid variant, however, does show improvement over the naive algorithm, and we can use its function to determine the cross-over point for our machine. The point at which this function and the function for the naive algorithm intersect, is the point where the hybrid variant, using $n_0 = n$, becomes faster than the naive algorithm. We found the functions intersecting between dimensions $n = 84$ and $n = 85$, meaning that from $n = 85$ onward, the hybrid variant outperforms naive matrix multiplication on the machine used for this experiment. In our next experiment, we will use this cross-over point to compare naive matrix multiplication to a more practical implementation of the Strassen algorithm, i.e. the hybrid variant with $n_0 = 85$.

5.2 Experiment II

The results for our seconds experiment are presented in the table below, in the same manner as we did in Table 5.1. As we found a cross-over point greater than 64, we did not test the algorithms on matrices of that dimension.

Table 5.2. *Results of experiment II*

		Average runtime (ms)				
Algorithm	*n*	128	256	512	1024	2048
Naive		25,36	215,37	1851,99	16136,2	180450,1
Strassen	- $n_0 = 85$	24,23	184,21	1322,68	9007,6	61915,8

Comparing the average runtimes in Table 5.2, we can see that as the matrices grow larger, the difference in complexity between the algorithms becomes more and more noticeable. Not only is the Strassen variant faster for every tested input dimension, the relative improvement over naive matrix multiplication grows with the sizes of the input matrices. This implies that, for large matrices, the difference in complexity has a significant impact on runtime and practicality. If we plot the runtime functions corresponding to these results, the impact of complexity becomes even more apparent:

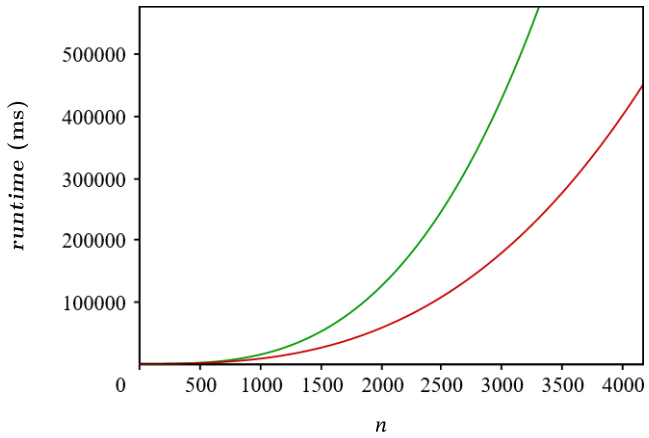


Figure 5.2 [9]. *Expected runtimes for the naive algorithm* (green) *and the hybrid variant using* $n_0 = 85$ (red).

Chapter 6

Conclusion

In this chapter, we will discuss the results of our experiment and reflect on possible limitations. To conclude this thesis, we will briefly discuss how our findings might apply to problems other than matrix multiplication.

6.1 Discussion

While Strassen's innovative method for matrix multiplication does improve on the asymptotic complexity of naive matrix multiplication, our results show that it is limited in its practicality, only improving on performance for matrices beyond practical use. To benefit from its lower complexity, practical implementations of the Strassen algorithm stop recursion early, resorting to the naive algorithm for multiplying the submatrices instead. Our results show that using this hybrid variant, the small difference in complexity between the Strassen algorithm and the naive approach can provide significant improvements in runtime.

However, performance is heavily dependent on the choice of the switching point n_0, and as optimal values for this point are highly system dependent, they to be determined empirically for distinct machines and implementations. Additional considerations regarding memory have to be made as well, due to the recursive nature of the Strassen algorithm. Constructing matrices at each point of recursion can result in substantial overhead, and the use of inefficient data structures can subsequently lead to significant losses in performance. Also, because we used only square matrices in our experiment, largely of dimensions 2^k, the padding required for the Strassen algorithm was minimized. When multiplying matrices of arbitrary sizes, efficient padding methods might be necessary to achieve competitive runtimes. Due to these practical concerns, the Strassen algorithm might not always be the preferred choice when deciding on an implementation of matrix multiplication, despite its lower complexity.

On the other hand, by dividing the problem into subproblems, Strassen's approach allows for parallel implementations that compute the products of submatrices concurrently. By using multiple processor cores simultaneously, this is likely to result in significantly reduced runtimes. Unfortunately, the implementation we used for our experiment does not take advantage of this, and it might be worthwhile to explore the possibilities of multi-threaded implementations in future works.

6.2 Conclusion

Analyzing and comparing algorithms proves to be a complicated matter. Asymptotic analysis and the big-O notation provide a universal notion of complexity, making it easier to determine and compare the complexities of algorithms. However, this is not a definitive measure, as there are often other factors affecting an algorithm's practical performance. This thesis illustrates this, by examining the Strassen algorithm for matrix multiplication, and comparing it to naive matrix multiplication. While Strassen's method improves on the theoretical complexity of matrix multiplication, implementations need to account for a number of practical considerations to benefit from this reduced complexity.

References

[1] N. Lončarić and M. Kraljić, "Matrices in Computer Graphics," *Tehnički glasnik,* pp. 120-123, 2018.

[2] C. Ionescu, O. Vantzos and C. Sminchisescu, "Matrix Backpropagation for Deep Networks with Structured Layers," in *The IEEE International Conference on Computer Vision*, 2015.

[3] K. Sood, "Solver Schemes for Linear Systems," 2016.

[4] J. L. Bentley, D. Haken and J. B. Saxe, "A General Method for Solving Divide-and-Conquer Recurrences," *SIGACT News,* vol. 12, no. 3, p. 36–44, 1980.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, in *Introduction to Algorithms*, The MIT Press, 2009, pp. 77, 94, 98-106.

[6] K. Bogart, S. Drysdale and C. Stein, in *Discrete Math for Computer Science Students*, 2004, pp. 156-157.

[7] V. Strassen, "Gaussian Elimination is not Optimal," *Numerische Mathematik,* vol. 13, pp. 354-356, 1969.

[8] M. Hohenwarter, "GeoGebra - Dynamic Mathematics for Everyone," [Online]. Available: https://www.geogebra.org/graphing/ewvgysfd.

[9] M. Hohenwarter, "GeoGebra - Dynamic Mathematics for Everyone," [Online]. Available: https://www.geogebra.org/graphing/mv3p2wn4.