Bachelor's Thesis Computing Science

Radboud University Nijmegen

# Visualising divide-and-conquer algorithms with self-similar fractals

*Author:*
Artur Wiadrowski
s1090597

*First supervisor/assessor:*
Dr. Engelbert Hubbers

*Second assessor:*
Dr. Wieb Bosma

June 9, 2024

## Abstract

Fractals, geometric shapes with the property of self-similarity, are objects the Hausdorff dimensions of which are of fractional values. Establishing connections between the number of self-similar copies and the number of recursive invocations of divide-and-conquer algorithms, the scaling factors of such copies and the scaling factors within the algorithms, and further, allows us to visualise divide-and-conquer algorithms with self-similar fractals. Working with computer animations provides us with the possibility of accounting in our visualisations for the operations done in any recursive call that are not recursive calls themselves. To design such animations, we make use of Iterated Function Systems, the application of which on a set of points infinitely many times yields fractals. We consider not only the final fractal, but also fractal approximations arising after finite application of the associated Iterated Function Systems. We utilize all of this and give visualisations of divide-and-conquer algorithms whereby the input size, the scaling factor for the input sizes passed to the recursive calls, the number of the recursive calls itself, and the complexity of the operations that are not recursive invocations, can all be inferred from our product.

# Contents

Chapter 1

# Introduction

Mandelbrot, commonly regarded as the originator of the concept, coined the term ‘fractal’ to mean a self-similar shape, necessarily of infinite detail [16]. Since then, there has been discussion on the definition as fractals might as well display the property of self-affinity rather than self-similarity, and Mandelbrot himself proposed using the term “without a pedantic definition” [17]. After introducing fractals as objects for study, there have been discoveries made in the field. For instance, a generalization of the Euclidean concept of dimensionality, one where dimensions can assume fractional values, has been shown to be connected to fractals [10]. The values of those dimensions, as it turns out for self-similar fractals composed of mutually disjoint parts, are dependent on the number of the self-similar parts composing the fractal and the scaling factors of those parts. That dimension of a fractal corresponds to the number and size of subproblems of divide-and-conquer algorithms, provided the fractal is self-similar.

Self-similarity, to give an intuitive explanation, is the property of an image such that upon continually zooming-in on a part of it, one will eventually end up with the same image. Since divide-and-conquer algorithms are recursive and each call looks similar, it is not surprising that there are connections between self-similar fractals and divide-and-conquer algorithms. There is also a connection between such fractals and the Master Theorem, discussed in Chapter 4, for deducing the complexity of divide-and-conquer algorithms, as is seen in Section 4.1. Given a fractal of the same number of self-similar copies as the number of recursive calls per one algorithm invocation, and the same scaling factors of those copies as those of the input sizes of the recursive calls, which case of the theorem is possible to apply depends on the asymptotic behavior of the complexity of the non-recursive operations of the algorithm compared to the Hausdorff measure of the fractal.

The research question of this thesis is “How can aspects of divide-and-conquer algorithms, including the number of recursive calls, the scaling factor for inputs, the size of the original input as well as the complexity of algorithmic operations, be visualised with fractals?" Drawing inspiration from Simant Dube's work in [6] and [5], we present an approach to visualising divide-and-conquer algorithms with computer animations. While Dube associates the work of an algorithm that is itself not a recursive invocation, with a part of an image, we deviate from this approach by introducing computer animations. We construct animations such that a fractal is approximated, and the animation time from a fractal approximation to the other, is the same as the time it takes for the associated divide-and-conquer algorithm to combine the results of all subproblems of certain depth.

In this thesis, we will state formal connections between divide-and-conquer algorithms and self-similar fractals. Graphics have been added to help readers understand fractals better and make it easier to see how fractals relate to divide-and-conquer algorithms. This thesis includes Chapter 2 on all the necessary preliminaries, such as the concept of Hausdorff dimension or the introduction of Iterated Function Systems. The preliminaries also make clear the terminology employed throughout this thesis. We then outline in Chapter 5 the portrayal of recursive algorithms with fractals. There, the number of recursive calls in a divide-and-conquer algorithm is represented with the number of self-similar copies composing the associated fractal. The parameter $b$, by the reciprocal of which input size is scaled, is associated with the contraction factor that gives the self-similar copies. We further associate the time of algorithmic computations that are not recursive invocations with the animation times of going from one fractal approximation to the next one. With this, we see yet another connection to the Master Theorem. As it turns out, the complexity of the animation visualising a divide-and-conquer algorithm, is necessarily the same as the complexity of the algorithm itself. All of this leads us to conclude that fractals can be used to visualise divide-and-conquer algorithms.

# Chapter 2

# Preliminaries

To tackle our research question, we give all the necessary preliminaries in this chapter.

## 2.1 Fractals

The term 'fractal' was coined by Mandelbrot [16]. The term was used to describe shapes exhibiting self-similarity. There are many ways of constructing fractals. We give examples below.

### 2.1.1 Examples

**Example 1** (Middle-thirds Cantor Set)**.** *To construct this Cantor Set, one starts with a line, divides it into three equal parts, and removes the middle part. Then, the same action is applied to the remaining components. The set is the result of infinite application of this action [10, p.77-81].*

Figure 2.1: Approximation of the middle-thirds Cantor set.

**Example 2** (Koch Snowflake)**.** *To construct the Koch Snowflake, one starts with an equilateral triangle, builds two more sides sticking out of the middle one-third piece of each side thus making those form a smaller equilateral triangle, and applies this action to all the sides again. The Koch Snowflake is the boundary of the outcome of this action applied infinitely many times [10, p.41-42].*

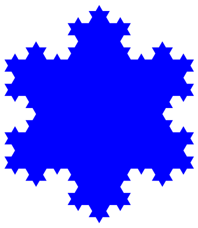

Figure 2.2: Approximation of the Koch Snowflake.

**Example 3** (Sierpinski Triangle)**.** *To construct the Sierpinski Triangle, one takes an equilateral triangle, divides it into four parts, and removes the part in the center, and then proceeds to apply the same action on the remaining three triangles.*

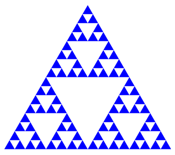

Figure 2.3: Approximation of the Sierpinski Triangle.

**Example 4** (Dragon Curve)**.** *The dragon curve can be constructed by recursively applying a rule onto a line segment [15].*

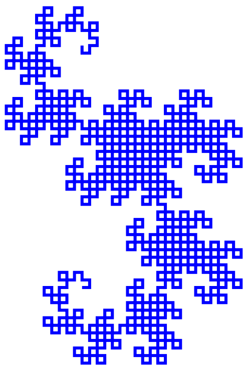

Figure 2.4: Approximation of the Dragon Curve.

**Example 5** (Mandelbrot Set)**.** *The Mandelbrot Set is described by points in the complex plane for which a recurrence relation converges [3].*

Example 1, Example 2, Example 3, and Example 4 are examples of fractals that can be constructed with Iterated Function Systems, which we outline

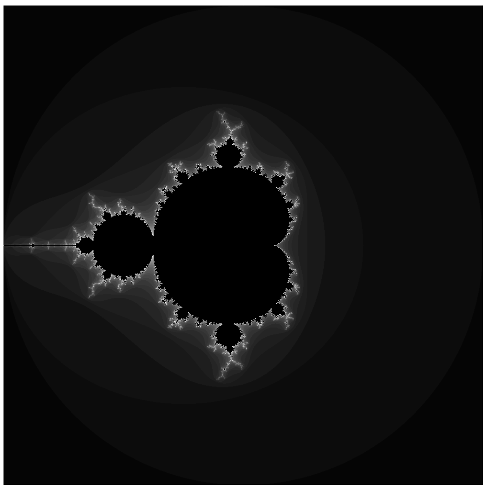
Figure 2.5: Mandelbrot Set.

in Section 2.1.3. This thesis concerns only fractals for which there exist such Iterated Function Systems. Other examples are shown for the sake of completeness. The given fractals are just examples, and the list is non-exhaustive. To formally define Iterated Function Systems, we first proceed to introduce geometric transformations.

### 2.1.2 Affine and linear transformations

Affine and linear transformations are bijections $f : V \to W$ of a set of points $V$ to a set of points $W$ that can be classified according to the properties they preserve. Affine transformations preserve lines and parallelism, but not necessarily Euclidean distances and angles. Linear transformations are affine transformations between vector spaces that preserve the operations of vector addition and scalar multiplication. Similarities are linear transformations. A specific form of a similarity is that of uniform scaling:

**Definition 1.** *Uniform Scaling is a linear transformation $f : V \to V$ of a vector space $V$ onto itself that shrinks or enlarges objects by a* scale factor *that is the same in all directions.*

Uniform scaling results in objects that are geometrically similar to their original. Of particular interest are transformations that result in a shrunken image. This is because those transformations allow us to construct fractals, which are composed of self-similar smaller copies.

**Definition 2** (Contraction)**.** *Let $X$ be a subset of $\mathbb{R}^n$, $n \geq 1$. The transformation $f : X \to X$ is called a contraction if there is some constant $c \in [0, 1)$ such that*

$$|f(x) - f(y)| \leq c|x - y| \textit{ for all } x, y \in X. \tag{2.1}$$

*The number $c$ is a contraction factor of $f$. If a set of points is scaled by the same value along every axis, we call $c$ the contraction factor of $f$.*

### 2.1.3 Iterated Function Systems

Iterated Function Systems are tools for building fractals. An Iterated Function System $\mathcal{F}$ of $t$ contractions is defined as a union of contractions

$$\mathcal{F} = \bigcup_{i=1}^{t} f_i, \tag{2.2}$$

where each contraction is a function from $X$ to $X$ for some compact metric space $(X, d)$ [8]. We say that any contraction function has an associated contractivity factor. This factor is the value by which any passed object is scaled in one of the directions. In this thesis, we only consider contractions scaling objects by the same factors in all directions. It is shown that for any Iterated Function System, given some set, there exists a unique fractal that is the result of applying the Iterated Function System on that set infinitely many times [8]. Hence, given an Iterated Function System and such a set, there must exist a fractal associated with it. In this thesis, we use the notation

$$\mathcal{F}^k(A) = \mathcal{F}(\mathcal{F}^{k-1}(A)) = \bigcup_{i=1}^{t} f_i(\mathcal{F}^{k-1}(A)) \tag{2.3}$$

for all $t \in \mathbb{N}$.

### 2.1.4 Fractal dimension

Observe that for $D$-dimensional figures, scaling the length of their sides by $r$ results in multiplying their $D$-dimensional measures by $r^D$. We notice that for the Sierpinski Triangle, scaling its sides by 2 produces three copies of the Triangle. We thus obtain $3 = 2^D$, which gives dimension $D = \log_2 3$. While this approach is not formal, we see that fractals necessitate the introduction of fractional dimensions. Dimensions are descriptors of how geometrical shapes fit in space. Extending the notion of dimension to fractional values can be done in many ways. The focus of this thesis for measuring the dimension of a fractal is the Hausdorff dimension, which we define in Definition 4. It is necessary to use an approach differing from that of Lebesgue. For example, trying to use the Lebesgue measure for fractals often yields no meaningful descriptors of fractals, as shown in Section 3.1.

#### Hausdorff dimension

The Hausdorff dimension of a fractal is a descriptor of how such a fractal fits in space. Definition 4 is the definition of the Hausdorff dimension. First, we define the Hausdorff measure.

**Definition 3.** *Let $X$ be a metric space. Let $d \in [0, \infty)$ and $S \subset X$. Then let*

$$H_\delta^d(S) = \inf \left\{ \sum_{i=1}^{\infty} diam^d(U_i) \;\middle|\; \bigcup_{i=1}^{\infty} U_i \supset S,\, diam(U_i) < \delta \right\}, \tag{2.4}$$

*where the infimum is the smallest element of the set. Given this, the Hausdorff measure is given by*

$$H^d(S) = \lim_{\delta \to 0} H_\delta^d(S). \tag{2.5}$$

This allows us to give a definition of the Hausdorff dimension as in [9].

**Definition 4.** *The Hausdorff dimension $dim_H$ of $S$ is given by*

$$dim_H(D) = \inf\{d \geq 0 \mid H^d(S) = 0\}. \tag{2.6}$$

Chapter 3 shows that the Hausdorff measure of a self-similar fractal must be positive and finite.

### 2.1.5 Considered fractals

The focus of this thesis remains on self-similar fractals for which certain conditions hold. Intuitively, we only consider fractals made up of copies of itself scaled by certain factors, where the fractal parts do not overlap. To give a formal definition, we first require that there must exist an Iterated Function System the infinite application of which on a set of points yields a fractal. From there, we can define the Open Set Condition.

**Definition 5.** *We say that the components $f_i(A)$ of $A$ satisfy the Open Set Condition if there exists a non-empty bounded open set $V$ such that*

$$V \supset \bigcup_{i=1}^{a} f_i(V) \tag{2.7}$$

*with the parts $f_i(X)$ being mutually disjoint. An open set is one where for every point in such a set, a neighbourhood of positive radius of this point is contained in this set.*

We only consider fractals that satisfy this condition. Such fractals are composed of mutually disjoint self-similar copies. That is to say, given some set of points $A$ and an Iterated Function System with contractions $f_1$, ..., $f_a$, the generated fractal $\mathcal{F}^\infty(X)$ is given by $\mathcal{F}^\infty(X) = f_1(\mathcal{F}^\infty(X)) \cup ... \cup f_a(\mathcal{F}^\infty(X))$ where for any $i,\ j,\ i \neq j$,

$$f_i(\mathcal{F}^\infty(X)) \cap f_j(\mathcal{F}^\infty(X)) = \emptyset. \tag{2.8}$$

With the Open Set condition, it is shown in [12] that for any fractal $A$, the Hausdorff dimension of the fractal $A$ is equal to $s$, where

$$\sum_{i=1}^{a} c_i^s = 1, \tag{2.9}$$

where $c_i$ is the contraction factor of $f_i$ and the parameter $a$ is the number of self-similar copies composing the fractal [12]. We mostly focus on fractals where all the scaling factors are the same. For such a fractal $A$, it holds that

$$number\ of\ copies = (scaling\ factor)^{-dim_H A}, \tag{2.10}$$

where the number of copies in question is the amount of self-copies composing the fractal. Now, the Hausdorff dimension of the fractal $A$ can be given explicitly as

$$dim_H(A) = -\frac{\log number\ of\ copies}{\log scaling\ factor}. \tag{2.11}$$

Unless stated otherwise, the fractals considered in this thesis have their Iterated Function Systems defined with the same contraction factors per fractal.

## 2.2 Notation

Throughout this thesis, the symbols $O$, $\Omega$, and $\Theta$ are used. They mean the following.

**Definition 6.** *If there exist constants $c,\ N \in \mathbb{R}$ and a function $g$ such that $\forall n > N \mid f(n) < cg(n)$, then $f(n) = O(g(n))$. In other words, $f$ is said to be upper-bounded by $g$.*

**Definition 7.** *If there exist constants $c,\ N \in \mathbb{R}$ and a function $g$ such that $\forall n > N \mid cg(n) < f(n)$, then $f(n) = \Omega(g(n))$. In other words, $f$ is said to be lower-bounded by $g$.*

**Definition 8.** *If there exist constants $c_1,\ c_2,\ N \in \mathbb{R}$ and a function $g$ such that $\forall n > N \mid c_1 g(n) < f(n) < c_2 g(n)$, then $f(n) = \Theta(g(n))$. In other words, $f$ is both upper- and lower-bounded by $g$.*

We also use $[\cdot]$ to indicate the rounded value of $\cdot$. We use $\lceil\cdot\rceil$ and $\lfloor\cdot\rfloor$ to indicate the rounded up and rounded down value of $\cdot$ respectively.

## 2.3 Divide-and-conquer algorithms

Divide-and-conquer algorithms are algorithms that invoke themselves recursively on parts of the original input [2, p.76]. For algorithms where the parts of the original input are of the same size, they must adhere to the template shown in Algorithm 1:

**Algorithm 1** Divide-and-Conquer

```
Data: $A$ of size $n$
Check whether $A$ is sufficiently small to apply a base case. If so, apply an action of complexity $O(1)$.

For every $i = 1, 2, ..., a$ do:
    Invoke this algorithm on a part of $A$ that is of size $\frac{n}{b}$.

Perform an action on $A$ of complexity $f(n)$.
```

Unless stated otherwise, we consider divide-and-conquer algorithms following the template in Algorithm 3. Throughout this thesis, we use the variable $a$ to describe the number of recursive calls of any divide-and-conquer algorithm that is invoked in an algorithmic call. Each of those calls is invoked on input of size scaled by the variable $\frac{1}{b}$ as compared to the original input. Throughout this thesis, $f(n)$ describes the complexity of all actions performed in a call of an algorithm to which input of size $n$ was passed, that themselves are not recursive invocations of the algorithm. The parameter $n$ always refers to the size of the input passed to a first call of any divide-and-conquer algorithm.

Here, we also introduce the notion of depth of a divide-and-conquer algorithm.

**Definition 9.** *Let the ratio of the size of the input to the immediate recursive call be $b$. A recursive call is said to be of depth $k$ if the divide-and-conquer algorithm was called on input of size $n$, and the recursive call is called on input of size $\frac{n}{b^k}$.*

### 2.3.1 Examples

Examples of divide-and-conquer algorithms are Binary Search [13, p.132-134], Merge Sort [13, p.120-122], and Strassen's Matrix Multiplication algorithm [14].

### 2.3.2 Complexity of divide-and-conquer algorithms

From the above template for divide-and-conquer algorithms, we deduce that the complexity function is given recursively by

$$T(n) = aT\left(\frac{n}{b}\right) + f(n). \tag{2.12}$$

This is because $T(n)$ gives the time it takes for the algorithm to process input of size $n$, and we see that such a call is comprised of other $a$ calls on input of size $\frac{n}{b}$, plus the non-recursive part that takes $f(n)$ time.

For relatively simple relations, one can continue replacing the $T\left(\frac{n}{b}\right)$ factor recursively to see what the pattern for that particular relation is. However, the Master Theorem defined in Theorem 1 allows one, under certain conditions, to deduce the complexity of such algorithms much more efficiently.

### 2.3.3 Master Theorem

This theorem states the bounds of the above recurrence relations for divide-and-conquer algorithms [2, p.112].

**Theorem 1.** *Let $a \geq 1$, $b > 1$. Then $T(n)$ has the following bounds:*

1. *If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.*
2. *If $f(n) = \Theta(n^{log_b a} \log^k n)$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$. This thesis only considers the $k = 0$ case.*
3. *If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$ and sufficiently large $n$, then $T(n) = \Theta(f(n))$.*

Throughout this thesis any references to the Master Theorem pertain exclusively to these three cases. Other cases are possible, as seen in [1].

### 2.3.4 Inadmissibility to the Master Theorem

There are inadmissible cases to the Master Theorem. Among others, they include the following constraints.

1. The variable $a$ must be a constant, hence a growing number of subproblems is not covered.
2. The $\epsilon$ constants in the theorem state that there must be a polynomial difference between $f(n)$ and $n^{\log_b a}$. Hence examples where it is not the case are inadmissible.
3. The time of computation must be positive, hence $f(n) > 0$.
4. One should also notice that $f$ should reflect the fact that computation of a divide-and-conquer algorithm must always be faster for smaller inputs. Hence examples where it is violated, for instance $f(n) = \frac{1}{n}$, are inadmissible.

The Master Theorem is further investigated in Chapter 4.

# Chapter 3

# Measure of fractals

In this chapter, we discuss ways of describing how much space a fractal occupies.

## 3.1 Lebesgue Measure

Consider the Sierpinski Triangle. Let the area of its fractal basis be $a_0$. The first fractal approximation's area is of value $\frac{3}{4}a_0$. In general, to find the area value $a_k$ of the $k$-th fractal approximation of the Sierpinski Triangle, we use the equation

$$a_k = \left(\frac{3}{4}\right)^k a_0. \tag{3.1}$$

The area of the Sierpinski Triangle is therefore 0. In this approach, we implicitly used Lebesgue measure. Definiton 10 gives the definition of Lebesgue measure taken from [17].

**Definition 10.** *Given a set $Y$ of finite size containing integers and an open set $S = \bigcup_{k\in Y}(a_k, b_k)$, its Lebesgue measure is defined by $\mu(S) = \sum_{k\in Y}(b_k - a_k)$. For any closed set $S' = [a, b] - \bigcup_{k\in Y}(a_k, b_k)$, its Lebesgue measure is given by $\mu(S) = b - a - \sum_{k\in Y}(b_k - a_k)$.*

Each approximation of the Sierpinski Triangle is defined as a sum of disjoint intervals. The lengths of those intervals are getting smaller for every approximation. For the fractal itself, the lengths are 0, hence the Lebesgue measure of this fractal is 0. This is to be expected, as this Lebesgue measure is taken implicitly in dimension 2, but the fractal itself is of dimension $\log_2 3$, which is lower. With this example we see that using Lebesgue measure is generally not a meaningful descriptor of how much space a fractal occupies.

## 3.2 Hausdorff Measure

The $D$-dimensional Hausdorff measure describes how a $D$-dimensional object fits in space. For self-similar fractals $\mathcal{F}^\infty(X)$ of dimension $D$ that we consider, the $D$-dimensional Hausdorff measure is always positive and finite. In other words,

$$0 < M < \infty, \tag{3.2}$$

where $M = H^D(\mathcal{F}^\infty(X))$.

Because of the imposed Open Set Condition given in Equation 2.7, our fractals are contained within some space that is not stretched infinitely in any direction. This already shows that our $D$-dimensional fractals cannot have infinite $D$-dimensional Hausdorff measures.

To show the strict positivity of the $D$-dimensional Hausdorff measures of any considered fractal of dimension $D$, we note the following. Each of those fractals is constructed by an Iterated Function System applied infinitely many times onto a non-empty set of points. Hence any such fractal is non-empty. By the Open Set Condition, there must exist in our fractal a point with neighbourhood of positive radius such that this neighbourhood is contained within this fractal. Hence the diameter of this fractal, which is the largest distance between any two points within this fractal, must be positive.

From there, it follows that the Hausdorff measure must be positive and finite. The $D$-dimensional Hausdorff measure of an object is the smallest element of the set of the sums of the diameters raised to the power of $D$ of some coverings of this set as the diameters of those coverings approach 0. This is by definition of the Hausdorff measure. The diameters are positive and finite by the property of self-similarity. We can cover our fractal with its self-similar copies. We always obtain a positive and finite $D$-dimensional Hausdorff measure if this $D$ matches the dimension of our fractal.

From this, we can derive Equation 2.9. The $D$-dimensional Hausdorff measure of a $D$-dimensional fractal is the sum of the $D$-dimensional Hausdorff measures of its $a$ copies, which are scaled versions of the original fractal. With $c_i$ being the contractivity factors, we obtain

$$M = Mc_1^D + Mc_2^D + ... + Mc_a^D \implies \sum_{i=1}^{a} c_i^D = 1. \tag{3.3}$$

It might be the case that the Hausdorff measure of a shape is unknown even if its Hausdorff dimension is known. This is the case for the Sierpinski Triangle, where currently only an interval of values containing the measure of the Triangle is known [11].

# Chapter 4

# Master Theorem

In this chapter, we give an explanation of the Master Theorem as well as its proof, and show a generalized version of the Master Theorem.

## 4.1 Explanation of the Master theorem

We give an explanation of the Master Theorem given in Section 2.3.3. In Section 4.1 we observe that the obtained asymptotic times include the term $n^{\log_b a}$, which is asymptotically the same as the $\log_b a$-dimensional Hausdorff measure of any self-similar fractal composed of $a$ copies of itself scaled by $\frac{1}{b}$.

### 4.1.1 First case of the Master Theorem

The condition for applying the first case is a polynomial difference between $f(n)$ and $n^{\log_b a}$ such that $f(n)$ must be the smaller function, that is to say,

$$\lim_{n\to\infty} \frac{f(n)}{n^{\log_b a}} = 0. \tag{4.1}$$

Because any divide-and-conquer algorithm calls itself recursively finitely many times, one will obtain a finite sum with terms each smaller than $n^{\log_b a}$, plus $n^{\log_b a}T(1)$ by expanding Equation 2.12. This is why this yields

$$T(n) = \Theta(n^{\log_b a}). \tag{4.2}$$

It should also be noted that the greatest value of $f(n), f\left(\frac{n}{b}\right)$, ... is $f(n)$ as per Section 2.3.4, hence it makes sense to compare the number of base cases to $f(n)$. The term determining the complexity here can either be the number of base cases or $f(n)$, and since in this case it is the number of base cases that grows faster than $f(n)$, the complexity is dependent only on $n^{\log_b a}$.

### 4.1.2 Second case of the Master Theorem

The penultimate case concerns $f$ that are polynomially the same as $n^{\log_b a}$. This means that

$$0 < \lim_{n\to\infty} \frac{f(n)}{n^{\log_b a}} < \infty. \tag{4.3}$$

If it is so, then we apply the following reasoning. We can sum up all the $f$ occurring in the recurrence-representing tree. Since the height of the tree is logarithmic, $n^{\log_b a}$ must be multiplied by the logarithm. Moreover, $f(n)$ being proportional to the number of base cases means that it is the sum of all the terms that determines the complexity, and hence $T(n)$ is described by a growing function dependent on the number of base cases. Thus it gives

$$T(n) = \Theta(n^{\log_b a} \log n). \tag{4.4}$$

### 4.1.3 Third case of the Master Theorem

This case concerns functions $f$ that are polynomially greater than $n^{\log_b a}$, meaning

$$\lim_{n\to\infty} \frac{f(n)}{n^{\log_b a}} = \infty. \tag{4.5}$$

If that is the case, one obtains a finite sum with terms greater than $n^{\log_b a}$, one of which will be $f(n)$, plus $n^{\log_b a}$ itself. Hence the $f(n)$ term will dominate and thus

$$T(n) = \Theta(f(n)) \tag{4.6}$$

in such cases. Moreover, since $n^{\log_b a}$ is the number of base cases, it means that all the base cases together are still less significant than $f(n)$, and hence $f$ determines the complexity in such cases.

### 4.1.4 Proof of the Master Theorem

Let $f(n)$ be bounded by $\Theta(Cn^d)$ for some $d$, $C$. Then

$$\begin{aligned}
T(n) &= aT\left(\frac{n}{b}\right) + Cn^d \\
&= a\left(aT\left(\frac{n}{b^2}\right) + C\left(\frac{n}{b}\right)^d\right) + Cn^d \\
&= a^2T\left(\frac{n}{b^2}\right) + Cn^d\left(1 + \frac{a}{b^d}\right) \\
&= a^3T\left(\frac{n}{b^3}\right) + Cn^d\left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2\right) \\
&\vdots \\
&= a^kT\left(\frac{n}{b^k}\right) + Cn^d\left(1 + \frac{a}{b^d} + ... + \left(\frac{a}{b^d}\right)^{k-1}\right).
\end{aligned} \tag{4.7}$$

Now, one can see that the value of this expression depends on what value $a$ assumes. Setting $k = \log_b n$, we obtain

$$T(n) = a^{log_b n} T(1) + \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(f(n) & \text{if } a > b^d. \end{cases} \tag{4.8}$$

which completes the proof. Setting $k = \log_b n$ is justified because $\frac{n}{b^k}$ is 1 at this $k$, thus indicating a base case in which the algorithm will cease calling itself recursively. It should be noted that taking $f$ to be bounded by a monomial is allowed because the Master Theorem case distinction is only made on the basis of polynomial differences as described in Sections 4.1.1, 4.1.2, and 4.1.3. For general $f$, one should choose a case depending on the value of the limit of the ratio of $f$ to $n^{\log_b a}$, just as described in those respective Sections.

### Impact of rounding

It should be noted that the Master Theorem is proven by comparing $f$ to monomials, as seen in Section 4.1.4. That is to say, to $Cn^d$ for some constants $C$, $d$. One should notice that either $C\lceil\frac{n}{b}\rceil^d = C\left\lfloor\frac{n}{b}\right\rfloor^d$ or

$$\begin{aligned} C\left\lceil\frac{n}{b}\right\rceil^d &= C\left(\left\lfloor\frac{n}{b}\right\rfloor + O(1)\right)^d \\ &= C\left\lfloor\frac{n}{b}\right\rfloor^d + O\left(\left\lfloor\frac{n}{b}\right\rfloor^{d-1}\right) \\ &= O\left(\left\lfloor\frac{n}{b}\right\rfloor^d\right) \end{aligned} \tag{4.9}$$

by the well-known binomial theorem [7, p.380]. Hence the rounding of the sizes of subproblems does not impact the complexity of divide-and-conquer algorithms.

# Chapter 5

# Construction

Settling for certain choices in interpretation, such as ascribing meaning to the function representing the time of work done in each call of a divide-and-conquer algorithm that itself is not an invocation of the algorithm, we will give, using fractals, a geometrical visualization of divide-and-conquer algorithms. Our visualisations are computer animations.

## 5.1 Parameters to visualise

Divide-and-conquer algorithms are defined by multiple parameters. There is the number of subproblems in a call, which we call $a$. Data of different sizes is passed to each of those calls. The data passed to the first algorithm call is said to be of size $n$. The ratio of the size of the input of a call to the size of the input to the immediate recursive call is $b$, the reciprocal of which is another parameter to visualise. The computation time of the work done in all subproblems of the same depth that are not recursive algorithm invocations also needs to be visualised. We proceed to give a visualisation that incorporates all of the above.

## 5.2 Outline of the approach

To visualise any divide-and-conquer algorithm, we first note that for computer visualisations, there is the process of animating, and the final result of such an animation. We first tackle visualising the parameters $a$, $\frac{1}{b}$, $n$. Because our approach is to give a visualisation, we can choose our animation times. The time of the entire animation per algorithm, as well as the intermediate animations that all constitute the entire animation, are chosen such that the animation time corresponds to the computation time of the associated algorithm.

For the purposes of constructing our animations, we introduce the notion of fractal bases and fractal approximations.

**Definition 11.** *A fractal basis is said to be a set of points to which no rule to transform it into a fractal was applied. We call this $X$.*

**Definition 12.** *The $k$-th fractal approximation is a fractal basis onto which an Iterated Function System $\mathcal{F}$ was applied $k$ number of times. We denote this as $\mathcal{F}^k(X)$.*

Our computer animations start with a fractal basis that is then approximated into a fractal. Our animations show the fractal basis $X$, then $\mathcal{F}(X)$, then $\mathcal{F}^2(X)$, and so on. Our animations include intermediate animations transforming one approximation into the next one. The choice of animation effects is arbitrary and is not covered by this thesis. Going from one fractal approximation to the next takes time, which we set in Section 5.4. How closely we are able to approximate any fractal depends on the constraints explained in Section 5.8.

## 5.3 Visualisation of the parameters $n$, $a$, $\frac{1}{b}$

In a single algorithmic call, there are $a$ invocations of the same divide-and-conquer algorithm that are of the next depth. The data passed to the original call is of size $n$. The size of the data passed to the recursive calls is scaled by $\frac{1}{b}$. In this section, we link these numbers to our visualisations.

### 5.3.1 Input size

In Definition 11, we defined $X$ as a set of points. To construct our animations, we choose $X$ to be a geometric figure of equal side lengths. We give an analog of the input size of an algorithm as the length of a side of a fractal basis. This fractal basis is the starting point of our animations.

### 5.3.2 Recursive calls

The analog of a recursive call of a divide-and-conquer algorithm shall be a self-similar copy that composes the associated fractal. The analog of the number of recursive calls of an algorithm shall be the number of self-similar copies composing the associated fractal. In line with Equation 2.2, this means that the Iterated Function System defining that fractal, such that infinite application of this iterated function system on a fractal basis gives that fractal, is given by

$$\mathcal{F}(X) = \bigcup_{i=1}^{a} f_i(X). \tag{5.1}$$

**Existence**

For any positive integer $a$, an iterated function system of mutually disjoint contractions as given in Equation 5.1 must, if applied infinitely many times onto a fractal basis, yield a fractal as explained in Section 2.1.3.

### 5.3.3 Size of subproblems

We portray the size of the subproblems of depth $k$ of a divide-and-conquer algorithm as the length of the sides of the scaled fractal bases composing the $k$-th fractal approximation $\mathcal{F}^k(X)$. We therefore give the contractivity factor $c_i$ of the contraction $f_i$ as

$$c_i = \frac{1}{b} \tag{5.2}$$

for all $f_i$ in Equation 5.1. This means that, for any fractal basis $X$, $\mathcal{F}^k(X)$ is composed of $a^k$ mutually disjoint scaled fractal bases, the sides of which are of length $\frac{n}{b^k}$.

## 5.4 Visualisation of the term $f$

The term $f(n)$ denotes the time of the computation it takes in an algorithm call with input size $n$ to combine the results of the subproblems. We give an analog to this in the form of the transformation time it takes for a set of points to transform, in a computer animation, into a closer approximation of a fractal. In this thesis, we operate on the assumption of the unit of time being the same for both the animations and the corresponding algorithms. From Definition 12, we can define the notion of transformations.

**Definition 13.** *Transformation is the process of applying $\mathcal{F}$ onto a fractal approximation.*

With this in mind, we proceed to define transformation time for the purpose of utilizing it in our computer animations.

**Definition 14.** *The transformation time of a fractal approximation is the time it takes in a computer animation to transform this fractal approximation into the next fractal approximation.*

In other words, in the computer animation used to portray a divide-and-conquer algorithm we have intermediate animations that start at a fractal approximation and result in the next one. Each of those takes some animation time, which we choose in Section 5.4.1.

### 5.4.1 Time

We give an analog of $f$ in the form of transformation time that depends on the depth of a recursive call as outlined in Definition 9.

Recall that any divide-and-conquer algorithm's time to process input of size $n$ is given by Equation 2.12. Expanding it, we obtain

$$T(n) = a^{\log_b n} T(1) + \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) = O\left(\sum_{i=0}^{\log_b n} a^i f\left(\frac{n}{b^i}\right)\right). \tag{5.3}$$

We give an analog of $f$ in the form of transformation time. For a fractal approximation $\mathcal{F}^k(X)$, its transformation time shall be $a^k f\left(\frac{n}{b^k}\right)$. This way, the transformation time of a $k$-th fractal approximation $\mathcal{F}^k(X)$ corresponds to combining the results of $a^k$ subproblems of depth $k$.

### 5.4.2 Overall complexity

Because our approach associates transformation times with the terms of the sum in Equation 5.3, and because a computer animations consist of fractal approximations with animations that take $a^k f\left(\frac{n}{b^k}\right)$ time to transform $\mathcal{F}^k(X)$ into $\mathcal{F}^{k+1}(X)$, we see that the asymptotic time it takes for a computer animation representing a divide-and-conquer algorithm to run is the same as the asymptotic time it takes for this algorithm to run. This is subject to the constraint described in Section 5.8.3. Moreover, the Master Theorem, as outlined in Section 2.3.3, gives the asymptotic time it takes to run the computer animation. This is because the Master Theorem gives the asymptotic time it takes to run the visualised algorithm, and this visualisation now has the same asymptotic running time.

## 5.5 Final animation product

Our computer animations consist of fractal approximations. Under our assumptions, such as the existence of an Iterated Function System generating that fractal, this fractal must always exist. By Equation 2.9, the fractal $\mathcal{F}^\infty(X)$ that is approximated is of Hausdorff dimension

$$dim_H(\mathcal{F}^\infty(X)) = \log_b a. \tag{5.4}$$

We note, also, that under our requirements, there are infinitely many fractals that can be approximated such that the animation still corresponds to the same divide-and-conquer algorithm. This is a feature of our design. This is because the fractal self-similar copies can always be arranged in different ways. For example, a fractal might consist of its self-similar copies stack on top of each other, or such copies placed horizontally. Also, the fractal self-similar copies do not have to stick to each other, they can be further apart. This leaves room under our approach for visualising divide-and-conquer algorithms in many ways while still adhering to the presented method.

## 5.6 Correctness

We outlined our requirements in Section 5.1. Of all of the listed parameters, we succeeded in visualising them. Divide-and-conquer algorithms can be represented with fractals such that their bases have sides that are of the same length as the size of the input for the algorithm. We see that fractals defined with Iterated Function Systems of $a$ contractions correspond to algorithms where there are $a$ recursive calls per call. We see that in divide-and-conquer algorithms where input size is scaled by $\frac{1}{b}$ before being passed onto the immediate recursive call, the parameter $\frac{1}{b}$ corresponds to the contractivity factors of the contractions composing the Iterated Function System that generates the associated fractal. By introducing the notion of transformation time in computer animations, we succeeded in visualising the asymptotic time of combining the results of subproblems of depth $k$ as the transformation time of the $k$-th fractal approximation $\mathcal{F}^k(X)$. We gave a visualisation of divide-and-conquer algorithms that inform the user of the parameters $n$, $a$, $\frac{1}{b}$, and $f$.

## 5.7 Argumentation for choices

In this section, we give reasons for the visualisation choices we made.

### 5.7.1 Association of parameter $n$

We chose to associate the size of the original input with the length of the sides of fractal bases. This is design choice. We could have decided that for fractals the bases of which are $D$-dimensional, the $D$-dimensional measures of those bases would have to be $n$. This would have entailed changing our approach to the contractivity factors.

### 5.7.2 Association of parameter $a$

We chose to associate the number of recursive calls invoked within a single algorithmic call with the number of the self-similar fractal copies composing the fractal that is approximated within our computer animations. The reason for this is the structure of divide-and-conquer algorithms. Each call, apart from the computation that is not recursive algorithm invocation, is necessarily composed of algorithmic calls of the same structure. Self-similar fractals have the same structure. By choosing to associate the parameter $a$ with the number of self-similar fractal copies composing it, we reflect in our visualisations the structure of the associated divide-and-conquer algorithms correctly.

### 5.7.3 Association of parameter $\frac{1}{b}$

We chose to associate the number that scales the input size before this input is passed to a recursive call of incremented depth, the parameter $\frac{1}{b}$, with the contractivity factors of the contractions composing the Iterated Function System that yields the associated fractal. We chose the self-similar copies of a fractal to represent recursive algorithmic calls. We chose the side lengths of the fractal basis of the fractal representing a divide-and-conquer algorithm of input size $n$, to also be of the value $n$. We observe that by choosing the contractivity factors for our computer animations to be $\frac{1}{b}$, we ensure that the self-similar fractal copies represent not only the same algorithmic structure, but also provide information on the size of the input passed to the represented algorithmic calls.

### 5.7.4 Association of computation time

We chose to associate the time of computation that is not itself recursive algorithmic invocation, with transformation times. Specifically, we chose the transformation time of the $k$-th fractal approximation to be $a^k f\left(\frac{n}{b^k}\right)$. By doing so, we made it possible to infer from the computer visualisations what the time of combining the results of subproblems of depth $k$ is. Note that we defined fractal approximations to be the results of repeated, finite applications of Iterated Function Systems on fractal bases. Our fractal approximations, therefore, will never be composed of figures that are fractal bases to which contractions were applied a different number of times, which would have symbolised processing some algorithmic calls before others. While this would have better reflected the actual order of algorithmic calls, we instead decided to omit this, as visualising the order of calls is not a goal given in Section 5.1. Within the constraint of working with only repeated applications of the entire Iterated Function Systems, we chose our transformation times well, as it now provides information on what the time of combining the results of subproblems of depth $k$ is.

### 5.7.5 Other possibilites

This section makes it clear that there could have been other choices taken to visualise divide-and-conquer algorithms. To list a few other possibilities, we note that we could have chosen the parameter $n$ to be represented by the $D$-measure of $D$-dimensional fractal bases. If we decided to have the fractal bases of the self-similar fractal copies to also have this property, we would have had to change the contractivity factors to $b^{-\frac{1}{D}}$. We could have decided the transformation times to be the same as the space complexity required for all depth $k$ subproblems instead. We could have decided to have non-recursive computation correspond to the angles by which fractal copies would have been rotated. We could have experimented with geometric figures composed of sets of points to which different contractions were applied a different amount of times. All of this leaves no doubt that what we presented is a possible approach out of many to visualising divide-and-conquer algorithms. This subsection provides ideas on possible future work.

## 5.8 Constraints

We notice that our computer animations are subject to the following constraints.

### 5.8.1 Rounding

There are Iterated Function Systems with contractions defined with numbers that have infinite decimal expansions. An example of this is the Sierpinski Triangle of side length 1. To construct it, we start with an equilateral triangle. Let it leftmost tip be at the point $(0, 0)$. It follows that its rightmost tip is at the point $(1, 0)$. The middle point between those two is the point $\left(\frac{1}{2}, 0\right)$. The points $(1, 0)$ and $\left(\frac{1}{2}, 0\right)$ are where two of the tips of one of the self-similar copies of the final fractal are. Because the bases of those copies are equilateral triangles as well, it follows that the third tip of the leftmost self-similar copy is $\left(\frac{1}{4}, \frac{\sqrt{3}}{4}\right)$.

What this tells us is that what follows scaling the fractal basis by $\frac{1}{2}$ is shifting the coefficients of the newly acquired set such that the components of the new fractal approximation are at the right place. For the Triangle, each point constituting the scaled fractal basis must be shifted by $\frac{1}{2}$ to the right to obtain the second of the three components of the first fractal approximation. To obtain the last component, we shift the points of the first scaled copy by $\frac{1}{4}$ to the right and up by $\frac{\sqrt{3}}{4}$. Hence, the Iterated Function System for the Triangle is the union of the contractions $f_1$, $f_2$, $f_3$ given by

$$f_1(x) = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} x, \tag{5.5}$$

$$f_2(x) = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} x + \begin{bmatrix} \frac{1}{2} \\ 0 \end{bmatrix}, \tag{5.6}$$

$$f_3(x) = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} x + \begin{bmatrix} \frac{1}{4} \\ \frac{\sqrt{3}}{4} \end{bmatrix}. \tag{5.7}$$

From this, we see that to get the exact position of some points, we have to tackle numbers with infinite decimal expansions. For fractal approximations generated with software, this necessitates rounding of the coordinates. This means that for some fractal approximations, the scaled fractal bases composing the approximations will not match the exact positions of their ideal counterparts. Designing representations of divide-and-conquer algorithms requires making sure that there is no overlap between the scaled fractal bases due to rounding.

### 5.8.2 Graphical constraint

Our computer animations approximate a fractal. Due to limitations of computers, we cannot show infinite approximation. Hence, we have to decide at what fractal approximation we could stop. There are multiple approaches available.

#### Bound on transformation time

For the cases where the transformation time is different for each fractal approximation, we can set a value such that if the transformation time is less than that value or exceeds that value, the computer animation would stop.

#### Bound on $\lceil D \rceil$-dimensional measure

In a computer animation, one will inevitably work with integer-valued dimensions. Spaces of non-integer dimensions are impossible to visualise. As shown in Section 3.1, fractals of any non-integer dimension, say $D$, have their $\lceil D \rceil$-dimensional measure equal to 0. This means that the $\lceil D \rceil$-dimensional measure of the respective fractal approximations tends to 0. One can, therefore, choose a value for such cases such that if the $\lceil D \rceil$-dimensional measure of a fractal approximation is smaller than that value, the computer animation stops. For example, we can choose an algorithm with 3 recursive invocations of itself at input scaled by $\frac{1}{2}$, with original input of size 16, to be visualised with animations approximating the Sierpinski Triangle. The area of the $k$-th fractal approximation is then $\left(\frac{3}{4}\right)^k 64\sqrt{3}$. We can decide to stop when the area of a fractal approximation is below 1. Then, we see that the area of $\mathcal{F}^{17}(X)$ is the first fractal approximation that satisfies this, because the area decreases, the area of $\mathcal{F}^{16}(X)$ is $\frac{43046721\sqrt{3}}{67108864}$, and the area of $\mathcal{F}^{17}(X)$ is $\frac{129140163\sqrt{3}}{268435456}$. We would then stop our computer animation at this approximation.

### 5.8.3 Transformation time constraint

The value $a^k f\left(\frac{n}{b^k}\right)$ is the transformation time of $\mathcal{F}^k(X)$. We want our computer animations to have the same asymptotic running time as that of the algorithms they visualise. We see therefore that depending on the parameters $a$, $\frac{1}{b}$ and $f$, we may have to stop at a specific fractal approximation. Taking Equation 4.7 for two different $k$ and evaluating the difference of those, we see that this difference is bounded by a constant if

$$\sum_{k=0}^{\infty} a^k f\left(\frac{n}{b^k}\right) < \infty. \tag{5.8}$$

This means that if Equation 5.8 holds, we can stop at any fractal approximation, save for the fractal basis itself, to achieve the same asymptotic running time of our animations as that of the associated algorithms. If Equation 5.8 does not hold, we have to stop at the $\log_b n$-th fractal approximation to have the same asymptotic running time. To see this, consider an algorithm with the recurrence relation $T(n) = T\left(\frac{n}{b}\right) + f(n)$ such that

$$f(n) = \frac{1}{1 + \ln\left(1 + \frac{1}{n}\right)}. \tag{5.9}$$

It should be noted that this function is a proper example as it does not violate any requirement of being admissible to the Master Theorem as outlined in Section 2.3.4.

$$\begin{aligned}\sum_{k=0}^{\infty} f\left(\frac{n}{b^k}\right) &= \sum_{k=0}^{\infty} \frac{1}{1 - \ln n + \ln(n + b^k)} \\ &\geq \sum_{k=0}^{\infty} \frac{1}{1 - \ln n + \ln((n+1)b^k)} \\ &= \sum_{k=0}^{\infty} \frac{1}{1 - \ln n + \ln(n+1) + k \ln b}.\end{aligned} \tag{5.10}$$

The last sum in Equation 5.10 is a shifted harmonic series, which is well known to diverge [7, p.345]. Provided Equation 5.8 holds, we obtain

$$O(T(n)) = O\left(a^{\log_b n} T(1) + \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)\right) = O\left(\sum_{i=0}^{\infty} a^i f\left(\frac{n}{b^i}\right)\right) \tag{5.11}$$

and as such, we are free in our computer animation to approximate the fractal beyond the first $\log_b n$ approximations.

## 5.9 Example of animation-algorithm relation

To show an example of a fractal-algorithm relation, we present Strassen's multiplication algorithm in Algorithm 2. Strassen described this approach to matrix multiplication in [14].

**Algorithm 2** Strassen's Multiplication of Matrices

```
Data: square matrices $A$ and $B$ with $n$ elements each

Check whether $A$ and $B$ are sufficiently small to perform a base case action.
If so, perform this base case action and return.

Split matrix $A$ into matrices $A_{11}, A_{12}, A_{21}, A_{22}$.
Split matrix $B$ into matrices $B_{11}, B_{12}, B_{21}, B_{22}$.

Let $P_1 = \text{Strassen}(A_{11} + A_{22}, B_{11} + B_{22})$.
Let $P_2 = \text{Strassen}(A_{21} + A_{22}, B_{11})$.
Let $P_3 = \text{Strassen}(A_{11}, B_{12} - B_{22})$.
Let $P_4 = \text{Strassen}(A_{22}, B_{21} - B_{11})$.
Let $P_5 = \text{Strassen}(A_{11} + A_{12}, B_{22})$.
Let $P_6 = \text{Strassen}(A_{21} - A_{11}, B_{11} + B_{12})$.
Let $P_7 = \text{Strassen}(A_{12} - A_{22}, B_{21} + B_{22})$.

Calculate auxiliary matrices as sums of $P_1, ..., P_7$, combine them and
return the result.
```

The algorithm follows the recursive relation

$$T(n) = 7T\left(\frac{n}{4}\right) + f(n), \tag{5.12}$$

where $f(n) = O(n)$ is the complexity of calculating the auxiliary matrices and combining them. Here, the parameter $a$ is 7, and the parameter $\frac{1}{b}$ is $\frac{1}{4}$. Hence, this algorithm can be visualised with a computer animation approximating a fractal made of 7 self-similar copies, each scaled by $\frac{1}{4}$. Figure 5.1 shows stages of our computer animation. We first start with a square of side lengths $n$, which is our fractal basis $X$. We then start our animation. At first, the first fractal approximation $\mathcal{F}(X)$ is arrived at in $n$ time. Then, $\mathcal{F}^2(X)$ is arrived at in $\frac{7}{4}n$ time. We continue approximating the fractal until the $\log_4 n$-th approximation, in line with constraint described in Section 5.8.3. The entire process runs for $\Theta(n^{\log_4 7})$ time, which is the same time as it takes for Algorithm 2 to run.

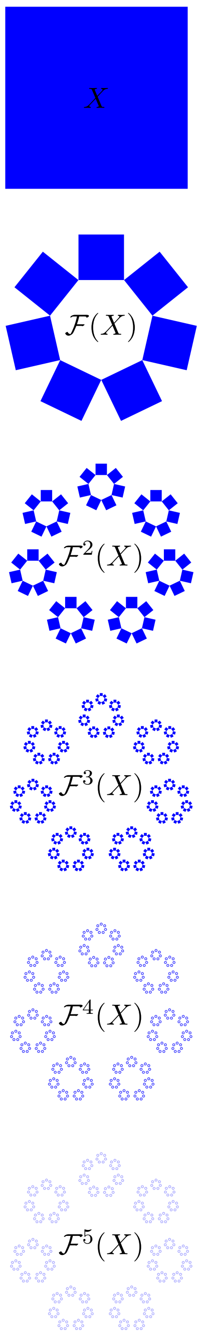

Figure 5.1: First 6 approximations of a fractal composed of 7 self-similar copies scaled by $\frac{1}{4}$.

# Chapter 6

# Algorithms with different scaling factors

In this chapter, we tackle algorithms following the template in Algorithm 3.

**Algorithm 3** Extended divide-and-conquer

```
Data: $A$ of size $n$

Check whether $A$ is sufficiently small to apply a base case. If so, apply an
action of complexity $O(1)$.

For every $i = 1, 2, ..., a$ do:
    Invoke this algorithm on a part of $A$ that is of size $nr_i$.

Perform an action on $A$ of complexity $f(n)$.
```

Algorithms following the template presented in Algorithm 3 have their recurrence relations defined as Equation 6.1.

$$T(n) = T(nr_1) + T(nr_2) + ... + T(nr_a) + f(n). \tag{6.1}$$

## 6.1 Generalized Master Theorem

As mentioned in Section 2.1.5, fractals may be composed of self-copies scaled by different factors. If a fractal is composed of $a$ copies of itself, each scaled by the factors $r_1, r_2, ..., r_a$, then its Hausdorff dimension $d$ is the solution to the equation

$$r_1^d + r_2^d + ... + r_a^d = 1 \tag{6.2}$$

as explained in Section 2.1.5. If all the contractivity ratios are $\frac{1}{b}$, this dimension $d$ is given by $d = \log_b a$.

In this section, we derive a Master theorem for divide-and-conquer algorithms the recurrence relations of which adhere to the template in Equation 6.1. We make use of comparing the function $f(n)$ to a monomial of the same degree as the dimension in Equation 6.2.

Take $f(n)$ to be bounded by $\Theta(Cn^d)$ for some $d$, $C$. Then

$$\begin{aligned} T(n) = Cn^d \left(1 + \sum_{i=1}^{N} r_i^d + \sum_{j=1}^{N}\sum_{i=1}^{N} r_j^d r_i^d + ... + \sum_{i_k=1}^{N} ... \sum_{i_1=1}^{N} r_{i_k}^d ... r_{i_1}^d \right) \\ + \sum_{i_{k+1}=1}^{N} ... \sum_{i_1=1}^{N} T(nr_{i_{k+1}}...r_{i_1}). \end{aligned} \tag{6.3}$$

We set $k$ to be bounded by $O(\log n)$. This is justified as the number of recursive algorithmic calls grows exponentially in the depth of any divide-and-conquer algorithm, and thus its depth must be bounded by a logarithm. Now, one can deduce the complexity just like for the standard Master Theorem.

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } D < d, \\ \Theta(n^D \log n) & \text{if } D = d, \\ \Theta(n^D) & \text{if } D > d. \end{cases} \tag{6.4}$$

## 6.2 Calls on input of differing size

We observe that under our approach to visualising divide-and-conquer algorithms, algorithms presented in Algorithm 3 can also be visualised with our animations. We argued in Section 5.3.2 that the number of self-similar copies can be made to correspond to the number of self-similar copies composing the corresponding fractal. This is the case also for algorithms presented in this section. One difference is that in Equation 5.1, the contractions $f_i,\ f_j$ have different contractivity factors $c_i,\ c_j,\ i \neq j$ such that

$$c_i \neq c_j \tag{6.5}$$

for some $i,\ j$. Specifically, those contractivity factors must be equal to $r_1,\ r_2, ...\ r_a$ respectively, as those are the numbers by which input size is scaled in Equation 6.1. As a result, the final product of our visualisations will be a figure approximating a fractal the parts of which are self-similar copies of itself, except scaled by differing factors. The approximations approach a fractal of dimension $s$, where $r_1^s + ... + r_a^s = 1$ as per Equation 2.9. Moreover, we can expand Equation 6.1 to obtain

$$\begin{aligned} T(n) = &\sum_{t=1}^{k}\left(\sum_{i_t=1}^{a}\dots\sum_{i_1=1}^{a} f(nr_{i_1}\dots r_{i_t})\right) \\ &+ \sum_{i_{k+1}=1}^{a}\dots\sum_{i_1=1}^{a} T(nr_{i_1}\dots r_{i_{k+1}}). \end{aligned} \tag{6.6}$$

We see therefore that the time of combining the results of subproblems of depth $k$ is given by $\sum_{i_1=1}^{a}\dots\sum_{i_k=1}^{a} f(nr_{i_1}\dots r_{i_k})$. We see that we can associate the transformation time of the $k$-th fractal approximation $\mathcal{F}^k(X)$ with $\sum_{i_1=1}^{a}\dots\sum_{i_k=1}^{a} f(nr_{i_1}\dots r_{i_k})$ for all $k$ to achieve the same transformation time of the $k$-th approximation as it takes for the algorithm to combine the results of all subproblems of depth $k$. In Section 6.1, we derive a generalized Master theorem for algorithms that follow the template in Algorithm 3. We observe that with our choice transformation of times, such animations run for the same time, asymptotically, as the corresponding divide-and-conquer algorithms. This is subject to the constraint that

$$\sum_{k=0}^{\infty}\left(\sum_{i_1=1}^{a}\dots\sum_{i_k=1}^{a} f(nr_{i_1}\dots r_{i_k})\right) < \infty, \tag{6.7}$$

which is analogous to the constraint outlined in Section 5.8. If not satisfied, we have to stop at a fractal approximation such that the last transformation time determined the asymptotic time of our computer animation to be the same as that of the associated algorithm. This is because any transformation time after that would change the asymptotic running time of our animation. Overall, we see that fractals are well-suited for representing also those divide-and-conquer algorithms that follow the template outlined in Algorithm 3.

# Chapter 7

# Related Work

This thesis presents an approach to visualising divide-and-conquer algorithms. This is one of many possible ways of doing so. There are more, the validity of which is dependent on what goals we assume. We see that, for example, we could have forgone of introducing animations and represent the function $f(n)$ with a single image, just like Simant Dube did in [6]. Within the animation approach, we could have associated the transformation times of fractal approximations with different values. For example, we could have chosen $a^k f\left(\frac{n}{b^k}\right)$ to indicate the amount of memory a divide-and-conquer algorithm requires to process the subproblems. There are many other possibilities, such as, if deciding $f$ to nonetheless represent the time to combine the results of subproblems, making only the overall asymptotic time of our animations be the same as that of the corresponding algorithm, thus disregarding what exactly the intermediate transformation times would be. This by itself shows that there are many approaches to visualising divide-and-conquer algorithms. We see that this thesis succeeds in showing one of those.

We used Iterated Function Systems to give a precise definition of the fractals we worked with. Iterated Function Systems are useful also in the cases of tackling possible relations of fractals to other topics of computing science. For example, the question of whether a fractal generated by an Iterated Function System intersects a diagonal line segment is undecidable [4]. In essence, fractals are infinitely complex and contain many details on arbitrarily small scales, whereas computers can only approximate fractals. Given any two points, one may require the final fractal, not approximations, to see if said fractal crosses the points. We see that fractals are related to the theory of computability as well, showing their many appearances in computing science.

Fractals also show up in approximation theory. Given any real-valued function $p$, one can approximate its root $x_\infty$ with the formula

$$x_{n+1} = x_n - \frac{p(x_n)}{p'(x_n)}, \tag{7.1}$$

which is known as Newton's Method [18]. This is known to produce what is known as Newton Fractals [18]. While Newton was unaware of the existence of those fractals, such fractals are nonetheless named after him to underline the fact of having been generated with the Newton approximation method. The fractal that is generated depends on the function the root of which we wish to approximate. The fractal itself then arises in the complex plane.

It should be noted that the generalized Master Theorem described in Section 6.1 is a special case of the Akra-Bazzi method as detailed in [1]. There, the authors accommodated all recursive relations of the form

$$T(n) = f(x) + \sum_{i=1}^{k} a_i T\left(nb_i + h_i(n)\right) \tag{7.2}$$

that have accompanying base cases and conditions. Crucially, a closer inspection of the relation between fractals and divide-and-conquer algorithms allowed us to rediscover a part of the Akra-Bazzi method. The rediscovery of a part of the Akra-Bazzi method testifies to the closeness of fractals and divide-and-conquer algorithms.

As can be seen, fractals are connected to many subfields of computing science. While this thesis focuses on the case of visualising divide-and-conquer algorithms with fractals, we note that fractals also appear in other topics of computing science, such as computability and approximation theory. Further work may show more connections between fractals and computing science.

# Chapter 8

# Conclusions

Fractals have been presented in this thesis alongside divide-and-conquer algorithms. Prior to the research section, we went on to describe the necessary knowledge relating to fractals, divide-and-conquer algorithms, and also the Master Theorem for deducing the complexity of divide-and-conquer algorithms. Once the basics were laid out, we went on to describe how divide-and-conquer algorithms can be visualised with computer animations approximating fractals.

We first argued that self-similar fractals can be used for visualising divide-and-conquer algorithms. In the recurrence relation $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ corresponding to a divide-and-conquer algorithm, $f(n)$ was associated with transformation time of a shape into a shape of yet more advanced detail, where an infinitude, under certain conditions, of consecutive transformations symbolize the work of the divide-and-conquer algorithm at hand. We associated the number of self-similar copies composing fractals with the number $a$, indicating the number of recursive calls per algporithm call. The parameter $\frac{1}{b}$ was associated with the scaling factors of those fractal copies. Establishing this relation allowed us to further deduce more correlations. This includes the observation that the exponent $\log_b a$ within the Master Theorem is the Hausdorff dimension of an associated self-similar fractal. The Master theorem itself was observed to give the same asymptotic running time of our animations as the illustrated divide-and-conquer algorithms. Furthermore, the $\log_b a$-dimensional measure of a fractal was mentioned. It has been argued that fractals are well-suited for representing divide-and-conquer algorithms.

Having noticed that the Hausdorff dimension applies also to fractals the parts of which are scaled by differing factors, we went on to take another look at the Master Theorem. We modified the Master Theorem. By inductive reasoning, we expected the generalized Hausdorff dimension to play a role in the generalized Master Theorem. This allowed us to discover a Master theorem for divide-and-conquer algorithms that sub-call themselves on data of different sizes within a single call. This result is proven.

This thesis makes it clear that fractals and divide-and-conquer algorithms are related. To answer the research question of this thesis, "“How can aspects of divide-and-conquer algorithms, including the number of recursive calls, the scaling factor for inputs, the size of the original input as well as the complexity of algorithmic operations, be visualised with fractals?”, one can observe that this can be done by using computer animations with transformation times the same as the time it takes to combine the results of $a^k f\left(\frac{n}{b^k}\right)$ algorithmic subproblems, where such animations approximate self-similar fractals composed of $a$ copies of itself, each scaled by the factor of $\frac{1}{b}$. There are more possibilities.

# Bibliography

[1] Mohamad A. Akra and Louay Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10:195–210, 1998.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, fourth edition.* MIT Press, 2022.

[3] Marius-F. Danca and Michal Feckan. Mandelbrot set and Julia sets of fractional order. page 2, 2022.

[4] Simant Dube. Undecidable problems in fractal geometry. *Complex Syst.*, 7:424, 1993.

[5] Simant Dube. Using fractal geometry for solving divide-and-conquer recurrences. *Journal of the Australian Mathematical Society*, 1993.

[6] Simant Dube. Geometrical Interpretation of the Master Theorem for Divide-and-conquer Recurrences. 2009.

[7] E.R. Fadell and A.G. Fadell. *Calculus.* University Series in Mathematics. Van Nostrand Reinhold Company, 1970. ISBN: 9780442023522.

[8] K. Falconer. *Fractal Geometry: Mathematical Foundations and Applications*, pages 123–126. 3rd edition, 2014.

[9] M. Fernández-Martínez, J.L.G. Guirao, M.Á. Sánchez-Granero, and J.E.T. Segovia. *Fractal Dimension for Fractal Structures: With Applications to Finance*, page 8. SEMA SIMAI Springer Series. Springer International Publishing, 2019.

[10] Benoit Mandelbrot. *Fractal Geometry of Nature.* W.H Freeman and Company, 1983. ISBN: 0-7167-1186-9.

[11] Péter Móra. Estimate of the Hausdorff measure of the Sierpinski triangle. *Fractals*, 17:137–148, 2009.

[12] P.A.P. Moran. Additive functions of intervals and Hausdorff measure. *Mathematical Proceedings of the Cambridge Philosophical Society*, 42(1):15–23, 1946.

[13] S.S. Skiena. *The Algorithm Design Manual.* Springer London, 2009. ISBN: 9781848000704.

[14] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[15] Serge Tabachnikov. Dragon Curves Revisited. *The Mathematical Intelligencer*, 36, 02 2014.

[16] T. Vicsek. *Fractal Growth Phenomena (2nd Edition)*, pages 9–11. World Scientific Publishing Company, 1992. ISBN: 9789814506199.

[17] Eric W. Weisstein. Lebesgue Measure. From MathWorld—A Wolfram Web Resource. Last visited on 2 June 2024.

[18] Figen Çilingir. Fractals Arising from Newton's Method. *AIP Conference Proceedings*, 1470:142–147, 08 2012.