Methods of Verifying Complexity Bounds of Algorithms using Dafny

Master Thesis in Computer Science

by

Shiri Morshtein

School of Computer Science

The Academic College Tel-Aviv Yaffo

Supervisors:

Dr. Ran Ettinger

Prof. Shmuel Tyszberowicz

September 2020

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of Master of Science in "Methods of Verifying Complexity Bounds of Algorithms using Dafny", is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed:_____ Date:_____

Acknowledgement

I would like to express my gratitude to my supervisors Prof. Shmuel Tyszberowicz and Dr. Ran Ettinger for their attention, guidance and encouragement. I am also thankful for their creative ideas, suggestions and moral support over the duration of this work. I would also like to extend my gratitude to all of my lecturers in the Academic College Tel-Aviv Yaffo, for their knowledge, skills and support they imparted to me.

Abstract

Formal program verification techniques are widely used to prove the functional correctness of programs according to formal specifications. However, nonfunctional properties such as time complexity are most commonly estimated manually by counting the number of elementary steps performed by the algorithm to finish execution. Since manual analysis is vulnerable to human errors, incorrect estimation of an algorithm's runtime may occur. This thesis presents methods for using the Dafny verification tool to specify and verify worst-case time complexity boundaries such as big-$\mathcal{O}$ as a nonfunctional properties specification of various algorithms. The approach presents with methods of calculating and exposing information about the algorithm's time complexity as part of its specification. Therefore, the software developer would calculate the aggregated complexity of a complex algorithm, which consists of several algorithms and adds it to its specification. If a change in any of the components is made, the component's postcondition specification must be changed as well, and the old definitions would not verify the complex algorithm time complexity. This approach is compatible with the complex structure of modern architecture. It will maintain the modularity of every complex algorithm, handling each of the algorithms that compose it separately and support frequent changes (such as refactoring), therefore preserving the program's complexity integrity. This approach can also be used for academic purposes, as an innovative methodology for teaching algorithms and complexity: this methodology supports the verification of both the algorithm's correctness and its time complexity. In this thesis, we define big-$\mathcal{O}$ for linear, logarithmic, quadratic, and exponential notations, and present implementations of each notation verified algorithm, functionally, and the specified time complexity.

תקציר

טכניקות לאימות פורמלי של תוכנה (Software Formal Verification) משמשות לרוב להוכחת נכונות פונקציונלית של מערכות תוכנה מורכבות בהתאם לאיפיון פורמלי. עם זאת, תכונות לא פונקציונליות כגון סיבוכיות זמן, מוערכות לרוב באופן ידני על ידי ספירת מספר הצעדים האלמנטריים שבוצעו על ידי האלגוריתם. ניתוח ידני של זמן הריצה חשוף לטעויות אנוש, וכיוצא מכך מתבצעת הערכה שגויה של זמן הריצה. תזה זו מציגה שיטות לשימוש בכלי אימות התוכנה Dafny, אשר משמש להוכחת נכונות פונקציונלית, להגדרת הדרישות לסיבוכיות הזמן של אלגוריתמים שונים והוכחתם. הגישה המוצגת בתזה מכילה שיטות לחישוב והוכחת סיבוכיות הזמן של האלגוריתם, ואופן חשיפת הממשק של האלגוריתם כחלק מהאיפיון שלו. לפיכך, ניתן יהיה לחשב את העלות המצטברת של אלגוריתם המורכב מכמה תתי-אלגוריתמים על בסיס ההצהרה הניתנת ע"י הממשק ומבלי צורך להריצם. אם נעשה שינוי באחד מתתי-האלגוריתמים, ההצהרה הניתנת על סיבוכיות הזמן שלו בממשק עשויה לדרוש שינוי, שכן אחרת היא לא תאומת ע"י Dafny. גישה זו תואמת את המבנה הארכיטקטוני של תוכנה מודרנית המבוססת על עקרונות המודולריות והמיקרו-שירותים וכן גם תומכת בשינויים תכופים (כגון refactoring), כיוון שכל הפרה של הסיבוכיות המוצהרת באחד מחלקי התוכנית תתגלה מיד. גישה זו יכולה לשמש גם למטרות אקדמיות, כמתודולוגיה חדשנית להוראת אלגוריתמים וסיבוכיות: מתודולוגיה זו תומכת באימות נכונות האלגוריתם ומורכבות הזמן שלו. בתזה זו, אנו מגדירים את הסימון האסימפטוטי "O" (big-O notation) עבור מחלקות סיבוכיות ליניאריות, לוגריתמיות, ריבועיות ואקספוננציאליות, ומציגים מימושים מאומתים של אלגוריתמים מכל מחלקה, הן מבחינה פונקציונלית והן מבחינת סיבוכיות הזמן שאופיינה.

Table Of Content

Abstract **i**

List of Figures **v**

1 Introduction **1**

1.1 The Problem Description 1

1.2 Contribution 2

1.3 Thesis Overview 2

2 Background **4**

2.1 Formal Program Verification 4

2.2 Dafny 5

2.3 Functional Specification and Verification Using Dafny 6

2.3.1 The Binary Search Algorithm 6

2.3.2 Formal Verification of Binary Search 7

3 Asymptotic Notations **10**

3.1 Definition of Big-$\mathcal{O}$ notations 11

3.1.1 Specification of Linear Complexity Class 11

3.1.2 Specification of Quadratic Complexity Class 12

3.1.3 Specification of Logarithmic Complexity Class 13

3.1.4 Specification of Exponential Complexity Class 15

3.2 Examples for Ω and Θ Notations 16

3.2.1 Specification of $\Omega(n)$ 16

3.2.2 Specification of $\Theta(n)$ 16

3.3 Verification of a Function for Big-$\mathcal{O}$, Ω, and Θ Notations 17

4 The Methodology **20**

4.1 Case Study 20

4.1.1 Specification of Binary Search Logarithmic Time Complexity with Dafny 20

4.1.2 Derivation of a tight Upper-Bound Function 22

4.1.3 Binary Search Upper-Bounding Function is Logarithmic 27

4.2 Definition of the Methodology 29

4.3 Demonstration of the Methodology 30

4.3.1 Fibonacci Algorithm - Iteration vs. Recursion 31

4.3.2 Insertion Sort 45

5 Proof engineering **54**

5.1 Example 1: Find Max 55

5.2 Example 2: Quick Sort 59

5.3 Example 3: Powerset Size 74

6 Conclusion **82**

6.1 Related Work 82

6.2 Conclusion 84

A About the Code **86**

List of Figures

2.1 A running example of binary search. 6

4.1 A Fibonacci sequence. 31

4.2 A running example of insertion sort. 45

5.1 A running example of quicksort. 60

Chapter 1

Introduction

Software is an integral part of every aspect of modern society, directly impacting human life, requiring a low defect rate. For safety-critical software, traditional quality assurance may not suffice in minimizing software defects. When traditional software quality assurance is not enough for defect removal, software engineering formal methods may help minimize defects. A formal method such as program verification is useful for proving correctness in real-time software.

The goal of program verification is to assure that software fully satisfies all the expected requirements. Requirements may include both functional and nonfunctional specifications. Formal verification of a program is the process of checking whether a design satisfies some requirements (properties), i.e., proving the correctness of a program with respect to its formal specification. In formal verification, we seek to examine the correctness in the program implementation's operation by mathematical proof. Dafny is a verification tool that is used to verify functional properties. This research demonstrates the time complexity verification methods for standard classes of complexity using Dafny.

1.1 The Problem Description

Since modern software development has embraced modular design [MW05], introducing us with software design styles as serverless architecture, micro-services and the concept of FaaS (Function as a Service), software applications are easier to understand, develop, test, and refactor [Che18]. Software modularity enables flexibility for small teams to develop, deploy, and scale their respective services quickly and independently. In modular

software products, software components can be combined to meet the varied purposes of a product. Each software component's functionality is presented with documentation, and interaction with each other is made through computing interfaces such as APIs. While the documentation attached to these APIs declares and specifies each component's functional capabilities and how to interact with it, nonfunctional properties specifications such as time complexity are not accessible or known to the developers. As software evolves, new components may be used, which might cause time complexity not to be maintained. Those changes can cause unauthorized changes in resource usage and performance issues, which their source would be hard to track. Software performance and data load tests attempt to detect run-time and excessive resource usage issues, yet only after each module is up and running. Detecting design flaws that influence cost and performance in the early stages of development is cheaper than making changes within a final product, thus the need for a methodology to support this kind of detection.

1.2 Contribution

The methodology of defining and proving the correctness of the time complexity with Dafny arms the user with a deeper understanding of an algorithm's time complexity and the big-$\mathcal{O}$ notation. These qualities can assist in implementing efficient programs. This methodology can also be used for academic causes by presenting to computer science students how to design and implement verified programs based on a functional and time complexity assessment. It is also practical since it generates fully verified code. Moreover, the algorithm's declaration also includes the time-complexity specification; hence it is guaranteed that the implementation also fulfills the runtime requirements.

1.3 Thesis Overview

This chapter presents a high-level explanation of program verification and time complexity specification. Furthermore, it describes the problem statement and thesis contribution according to the research motivation and goals. Chapter 2 provides the necessary background on program verification with Dafny, including a running example for functional

specification and several algorithms' verification. These algorithms are used as running examples for the thesis methodology implementation.

In Chapter 3 we display the definition, declaration, specification, and mathematical implementation with Dafny of the *big-*$\mathcal{O}$ notations for several complexity classes. It also contains a brief introduction to the Ω and Θ notations, carried on with a demonstration of complexity verification for a simple function.

The methodology of developing a verified algorithm, focused on time complexity verification using Dafny, is discussed in Chapter 4. This chapter includes a case study on the binary search algorithm and implementation of the generalized suggested methodology on several algorithms, implementation types, and complexity classes.

Then, in Chapter 5, we present the methodology for abstraction over modules of different complexity classes by related types, methods, and functions grouped and control the scope of declarations. We present the reuse of methods between algorithms and how to separate implementation from an interface. We also demonstrate the methodology for verifying other nonfunctional properties such as the powerset size in relation to the original set.

We conclude in Chapter 6, where we evaluate our work, present related work, and discuss future work.

Chapter 2

Background

This chapter introduces the background required for this thesis's perception and includes detailed implementations of functional software verification with Dafny. Those implementations are significant as they are the foundation of our work.

2.1 Formal Program Verification

An essential aspect of any algorithm is that it is correct: it always produces the expected output for the range of inputs, and it eventually terminates. As it turns out, it is not easy to prove that an algorithm is correct. Developers often use empirical analysis to find faults in an algorithm, yet the only way to prove an algorithm's correctness over all possible inputs is by formally reasoning about it. Formal program verification verifies that the implemented system meets design requirements (or specifications) using mathematical reasoning. The formal verification process starts with describing a specification for a program in some symbolic logic, followed by a proof (in some proof system) that the program meets the specification. If the proof system is sound, then this implies that the program meets its specification for all inputs. Formal specification and verification can help reduce bugs, aid in code maintenance, extension, and reuse. While formal specification is widespread in the industry, formal verification is most often applied in safety-critical situations (airplanes, cars, medical equipment, nuclear power plants) [LSP07],[BS93]. Researchers continue to develop systems to automate verification and develop programming methodologies in which proofs of correctness are produced along with programs.

2.2 Dafny

Dafny is an imperative language that supports formal specification using annotations. It builds on the Boogie intermediate language [Lei08], which uses the Z3 automated theorem prover for discharging proof obligations [MB08]. By defining theorems and using pre- and postconditions, loop invariants, assertions, and other constructs, we build fully verified programs. Cases where the theorem cannot be proved automatically, may require some help from the developer. The developer may assist Dafny, usually by providing *assertions* or *lemmas* [Rea20a]. Dafny is an auto-active program verifier [LM10b], which is more automated than proof assistants such as Coq [Kal+19]. It provides the platform to write a verified version of an algorithm, including the implementation and verification in the same method. Quoting from *Verified Calculations* [LP13]:

> Indeed, constructing proofs within interactive proof assistants (like Coq [BC04], Isabelle/HOL [Nip20], or PVS [ORS92]) has a reputation of being a demanding task. The purpose of these tools is to give the user maximum control over the proof process, which is why they favor predictability over automation. Interaction with such an assistant consists of issuing low-level commands, called tactics, that guide the prover through the maze of proof states. This mode of interaction is biased towards expert users with a good knowledge of the tool's inner workings. In contrast, auto-active program verifiers [LM10b], like Dafny [Lei10], VCC [Coh+09], or Veri- Fast [JSP10], take a different approach to mechanism reasoning: they provide automation by default, supported by an underlying SMT solver. All the interaction with such a verifier happens at the level of the programming language, which has the advantage of being familiar to the programmer. So far, these tools have been used mostly for verifying functional correctness of programs, but their domain is gradually expanding towards general mathematical proofs.

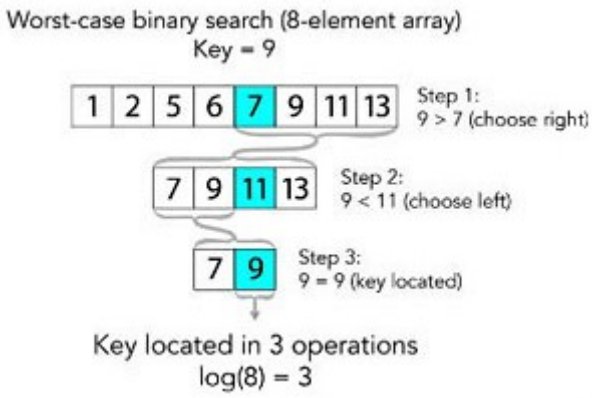

Figure 2.1: A running example of binary search.

2.3 Functional Specification and Verification Using Dafny

In this section, we present a full implementation process of functional verification of the binary search algorithm. The implementation of this case study will be used to demonstrate our methodology in the following chapter.

2.3.1 The Binary Search Algorithm

The binary search algorithm is a well-known example used in courses such as Algorithms and Data Structures to demonstrate the logarithmic time complexity search algorithm, searching a value in a sorted data structure of elements. We use a sorted sequence as that data structure where we search the key. In each iteration, the middle element of an interval is tested, and if it is not the required key, half of the sequence in which the key cannot lie is eliminated, and the search continues on the remaining half. It is repeated until either the key is found or the remaining half is empty, which means that it is not in the sequence. Since each iteration narrows the search range by half, after k steps, the algorithm reduces the range by 2^k. As soon as 2^k equals or exceeds n, the process terminates. Given that $n \leq 2^k$ is equivalent to $log_2 n \leq k$, the process terminates after at most $log_2 n$ steps. Figure 2.1 presents a running example of the algorithm.

It is not trivial to establish the correctness of the algorithm, because the index calculations might lead to programming errors [WS03a]. In our implementation of the algorithm, q is an input sequence, $|q|$ denotes its size, where the indices range from 0 to

$|q| - 1$. Initially, we assign the return value r the value -1. If the key is found, we change the value to the relevant index, which must range between 0 to $|q| - 1$; otherwise, -1 is returned. Hence, to specify the functional requirements of the algorithm, the following expressions must hold:

$$0 \leq r \Rightarrow (r < |q| \text{ and } q[r] = key) \tag{2.1}$$

$$r < 0 \Rightarrow \text{ key !in q} \tag{2.2}$$

2.3.2 Formal Verification of Binary Search

We now show a fully verified implementation of the binary search algorithm. The input sequence needs to be sorted to perform binary search:

```
predicate Sorted (q: seq<int>)
{
  ∀ i, j • 0≤i≤j<|q| ⟹ q[i]≤q[j]
}
```

For verifying the functional requirement of the binary search algorithm, we need to ensure that both expressions (2.1) and (2.2) are satisfied:

```
predicate BinaryPosts (q: seq<int>, r: int, key: int)
{
  (0≤r ⟹ (r<|q| ∧ q[r]=key)) ∧
  (r<0 ⟹ key ∉ q)
}
```

Now the binary search method can be declared with its functional specification:

```
method binarySearch (q: seq<int>, key: int) returns (r: int)
  requires Sorted(q)
  ensures BinaryPosts(q, r, key)
```

These properties can be used to prove the correctness of the search. Following is the body of the method:

```
r:= -1;
var lo, hi := 0, |q|;
while lo < hi
  invariant BinaryLoop(q, lo, hi, r, key)
  decreases hi-lo
{
  var mid := (lo+hi)/2;
  assert
```

```
  if key < q[mid]
  {
    hi := mid;
  }
  else if q[mid] < key
  {
    lo := mid + 1;
  }
  else
  {
    r := mid;
    hi := lo;
  }
}
```

Notice, Dafny uses mathematical integers, not the bounded integers found in most popular programming languages. It means that there is no issue of overflow in Dafny [LM10a] in the operation *(hi+lo)/2*, and we ensure it with the assertion:

```
assert (lo+hi)/2 = (lo+((hi-lo)/2));
```

Although, If we define:

```
newtype int32 = x | -0x8000_0000 ≤ x < 0x8000_0000
```

and now use (a+b)/2 we will have an exception.

A loop invariant is an expression that holds upon entering a loop and after every loop body execution. The loop invariants of the binary search algorithm are grouped into a predicate for a cleaner implementation. Note that in Dafny q[..lo] does not include lo index.

```
predicate BinaryLoop (q: seq<int>, lo: int, hi: int,
    r: int,key: int)
{
  (0≤lo≤hi≤|q| ∧
  (r<0 ⟹ key ∉ q[..lo] ∧ key ∉ q[hi..]) ∧
  (0≤r ⟹ r<|q| ∧ q[r]=key)
}
```

This predicate is composed of three expressions that must hold for every iteration. The first expression defines that indices' variables are within the input sequence. The second expression is concluded from the first postcondition—if r is -1, the key has not been found yet, so it does not lie in the part of the sequence that we already had eliminated. The third expression is the same as the second postcondition; if the returned value r is

non-negative, the key has been found, and r is assigned with its location. As we can see, when the loop invariant holds, the postcondition is satisfied; thus, the functional properties of the algorithm are verified.

Chapter 3

Asymptotic Notations

The traditional theoretical approach to algorithm analysis defines algorithm time complexity in counting the number of elementary operations performed by the algorithm. To express the time complexity of an algorithm, we use *asymptotic notation*s [Wil02]. Asymptotic notations are the mathematical notations used to describe an algorithm's runtime as a function of the input size.

Mathematical bounding for the time complexity of an algorithm is generally expressed as a function of the input size, which represents the asymptotic behavior of the complexity. There are mainly three asymptotic notations: the Ω notation, the Θ notation, and the big-$\mathcal{O}$ notation.

The big-$\mathcal{O}$ notation is a formal way describing the upper bound of an algorithm in terms of runtime Functions in Dafny define mathematical functions, i.e., pure functions without side-effects. We use *function*s to describe the mathematical functions, and *predicate*s (Boolean functions) to define conditions that those functions must fulfill. This section provides the definition, declaration, and specification of the *big-*$\mathcal{O}$ notation for linear, quadratic, logarithmic, and exponential functions implementation with Dafny. These functions are implemented respectively according to the mathematical definitions for O(n), O(n^2), O($log_2 n$), and O(2^n). These definitions are elaborated further in this chapter.

In this thesis, we follow the approach, steps, and techniques of the traditional methods and creates an automated environment for specifying and verifying its products. Therefore our methodology, which is presented in Chapter 4, relies on the same method:

counting the number of dominant steps of an algorithm's implementation, describing the runtime as a function of the input size that upper-bounds it, and proving that it belongs to a specific big-$\mathcal{O}$ notation. Since the thesis focused on big-$\mathcal{O}$ notations, Theta and Omega notations are presented only briefly.

3.1 Definition of Big-$\mathcal{O}$ notations

In this section we present the big-$\mathcal{O}$ notation of standard complexity classes. The big-$\mathcal{O}$ notation provides an upper bound for a function *f*.

3.1.1 Specification of Linear Complexity Class

The function $f(n)$, belongs to O(n) if:

$$\exists c > 0 \; and \; \exists n_0 > 0 \; s.t. \; \forall n. \; n \geq n_0 \Rightarrow f(n) \leq c * n \tag{3.1}$$

Now we specify O(n) with Dafny. The predicates and the *lemma* that define O(n) are:

```
predicate IsOn (n: nat ,t: nat)
{
  ∃ f: nat -> nat • IsLinear(f) ∧ t≤f(n)
}
predicate IsLinear (f: nat -> nat)
{
  ∃ c :nat, n0: nat • IsLinearFrom(c, n0, f)
}
predicate IsLinearFrom (c :nat, n0: nat, f: nat -> nat)
{
  ∀ n: nat • n0≤n ⟹ f(n)≤c*n
}
lemma linear (c :nat, n0: nat, f: nat -> nat)
  requires IsLinearFrom(c,n0,f)
  ensures IsLinear(f)
{}
```

The predicate *IsOn* is required to prove that such a function f which satisfies expression (3.1) and upper-bounds the the algorithm's runtime t indeed exists, where n is the size of the input. It defines the postcondition we strive to establish for any linear algorithm. The predicates and the lemma define the mathematical expression of O(n) as defined in expression (3.1): The *IsLinearFrom* predicate determines if for given positive

c, n_0, for every n larger than n0, $f(n)$ is upper-bounded by c*n (the second part of expression (3.1)). The *IsLinear* predicate determines if such positive c, n_0 that satisfies *IsLinearFrom* exist(the first part of expression (3.1)). The *linear* lemma is provided to merge the two predicates above it to the complete definition. In this case, the developer does not need to provide further proof to deduce that if the inputs *f,c,n0* satisfy the predicate *IsLinearFrom*, then f satisfies the predicate *IsLinear* and hence f is linear.

3.1.2 Specification of Quadratic Complexity Class

The function $f(n)$, belongs to O(n^2) if:

$$\exists c > 0 \; and \; \exists n_0 > 0 \; s.t. \; \forall n. \; n \geq n_0 \Rightarrow f(n) \leq c * n^2 \tag{3.2}$$

Now we specify O(n^2) with Dafny. The predicates and the lemma that define O(n^2) are:

```
predicate IsOn2 (n: nat ,t: nat )
{
  ∃ g: nat -> nat • IsQuadratic(g) ∧ t≤g(n)
}
predicate IsQuadraticFrom (c :nat, n0: nat, g: nat -> nat)
{
  ∀ n: nat • 0 < n0 ≤ n ⟹ g(n) ≤ c*n*n
}
predicate IsQuadratic(g: nat -> nat)
{
  ∃ c :nat, n0: nat • IsQuadraticFrom(c, n0, g)
}
lemma quadratic (c :nat, n0: nat, g: nat -> nat)
  requires IsQuadraticFrom(c,n0,g)
  ensures IsQuadratic(g)
{}
```

The predicate *IsOn2* is required to prove that such a function f which satisfies expression (3.2) and upper-bounds the the algorithm's runtime t indeed exists, where n is the size of the input. It defines the postcondition we strive to establish for any quadratic algorithm. The predicates and the lemma define the mathematical expression of O(n^2) as defined in expression (3.2): The *IsQuadraticFrom* predicate determines if for given positive c, n_0, for every n larger than n0, $f(n)$ is upper-bounded by $c * n^2$ (the second part of expression (3.2)). The *IsQuadratic* predicate determines if such positive c, n_0 that

satisfies *IsQuadraticFrom* exist(the first part of expression (3.2)). The *quadratic* lemma is provided to merge the two predicates above it to the complete definition. The developer does not need to provide further proof to deduce that if the inputs *f,c,n0* satisfy the predicate *IsQuadraticFrom*, then f satisfies the predicate *IsQuadratic* and hence f is quadratic.

3.1.3 Specification of Logarithmic Complexity Class

The function $f(n)$, belongs to O($log_2 n$) if:

$$\exists c > 0 \; and \; \exists n_0 > 0 \; s.t. \; \forall n. \; n \geq n_0 \Rightarrow f(n) \leq c * log_2 n \tag{3.3}$$

In this case, $log_2 n$ is a function that must be defined for Dafny.

```
function Log2 (n: nat): nat
  requires n>0
  decreases n
{
  natDivisionDecreases(n, 2);
  if n=1 then 0 else 1 + Log2(n/2)
}
```

The $log_2 n$ function must require a positive input, hence the n¿0 precondition. Also, the *decreases* annotation and the *lemma natDivisionDecreases* are used to assist Dafny to prove that the function terminates and calls itself with a valid value. The decreases annotation provides an expression that decreases with every recursive call and is bounded by 0.

The *natDivisionDecreases* lemma is an example of an additional proof that the developer provides to assist Dafny. The decreases expression of the function *Log2* holds since each recursive call divides the value by two, starting with a non-negative value. Thus, it necessarily becomes smaller with each call. Dafny does not automatically accept $n/2$ as a valid input (a positive integer) for the *Log2* function, thus this requirement is handled with the *natDivisionDecreases* lemma:

```
lemma natDivisionDecreases(a: nat, b: nat)
  requires b>0
  ensures a/b = (a as real / b as real).Floor

{
  assert a = (a/b)*b + (a%b);
```

```
  assert a as real = (a/b) as real * b as real + (a%b) as real;
  assert a as real / b as real =
         (a/b) as real + (a%b) as real / b as real;
}
```

This proof method is equivalent to a pen and paper mathematical proof, and in a few steps, it convinces Dafny that a division of a natural number with a positive integer is a natural number. Now that the $log_2 n$ function is defined, we specify the O($log_2 n$).

The predicates and the lemma that define O($log_2 n$) are:

```
predicate IsOLog2n (n: nat, t: nat )
{
  ∃ f: nat -> nat • IsLog2(f) ∧ t≤f(n)
}
predicate IsLog2 (f: nat -> nat)
{
  ∃ c :nat, n0: nat • IsLog2From(c, n0, f)
}
predicate IsLog2From (c :nat, n0: nat, f: nat -> nat)
{
  ∀ n: nat • 0 < n0 ≤ n ⟹ f(n) ≤ Log2(n)*c
}
lemma logarithmic (c :nat, n0: nat, f: nat -> nat)
  requires IsLog2From(c, n0, f)
  ensures IsLog2(f)
{}
```

The predicate *IsOLog2n* is required to prove that such a function f which satisfies expression (3.3) and upper-bounds the algorithm's runtime t indeed exists, where n is the size of the input. It defines the postcondition we strive to establish for any logarithmic algorithm. The predicates and the lemma define the mathematical expression of O($log_2 n$) as defined in expression (3.3): The *IsLog2From* predicate determines if for given positive c, n_0, for every n larger than n0, $f(n)$ is upper-bounded by $c * log_2 n$ (the second part of expression (3.3)). The *IsLog2* predicate determines if such positive c, n_0 that satisfies *IsLog2From* exist(the first part of expression (3.3)). The *logarithmic* lemma is provided to merge the two predicates above it to the complete definition. The developer does not need to provide further proof to deduce that if the inputs *f,c,n0* satisfy the predicate *IsLog2From*, then f satisfies the predicate *IsLog2* and hence f is logarithmic.

3.1.4 Specification of Exponential Complexity Class

The function $f(n)$, belongs to $O(2^n)$ if:

$$\exists c > 0 \; and \; \exists n_0 > 0 \; s.t. \; \forall n. \; n \geq n_0 \Rightarrow f(n) \leq 2^{n^c} \tag{3.4}$$

In this case, we use the power function, which must be defined for Dafny.

```
function Power(n: nat, c: nat): nat
  decreases c
{
  if c=0 then 1 else if n=0 then 0 else n * Power(n,c-1)
}
```

The predicates and the lemma that define $O(2^n)$ are:

```
predicate IsOExpn (n: nat ,t: nat )
{
  ∃ e: nat -> nat • IsExpo(e) ∧ t≤e(n)
}
predicate IsExpo (e: nat -> nat)
{
  ∃ c :nat, n0: nat • IsExpoFrom(c,n0,e)
}
predicate IsExpoFrom (c :nat, n0: nat, e: nat -> nat)
{
  ∀ n: nat • 0 < n0 ≤ n ⟹ e(n) ≤ Power(2,Power(n,c))
}
lemma exponential (c :nat, n0: nat, e: nat -> nat)
  requires IsExpoFrom(c,n0,e)
  ensures IsExpo(e)
{}
```

The predicate *IsOExpn* is required to prove that such a function f which satisfies expression (3.4) and upper-bounds the the algorithm's runtime t indeed exists, where n is the size of the input. It defines the postcondition we strive to establish for any exponential algorithm. The predicates and the lemma define the mathematical expression of $O(2^n)$ as defined in expression (3.4): The *IsExpoFrom* predicate determines if for given positive c, n_0, for every n larger than n0, $f(n)$ is upper-bounded by 2^{n^c} (the second part of expression (3.4)). The *IsExpo* predicate determines if such positive c, n_0 that satisfies *IsExpoFrom* exist(the first part of expression (3.4)). The *exponential* lemma is provided to merge the two predicates above it to the complete definition. The developer does not need to provide further proof to deduce that if the inputs *f,c,n0* satisfy the predicate *IsExpoFrom*, then f satisfies the predicate *IsExpo* and hence f is exponential.

3.2 Examples for Ω and Θ Notations

Since traditionally time complexity is commonly expressed using big-$\mathcal{O}$ notation, this thesis presented approach is focused on the verification of the big-$\mathcal{O}$ notation of an upper bounding function to an algorithm. Nevertheless, in the next sections, we demonstrate that our methods can be applied to Ω and Θ notations.

3.2.1 Specification of $\Omega(n)$

The Omega notation described as the lower bound of the function f permits us to bound the value of f from below. The function $f(n)$, belongs to $\Omega(n)$ if:

$$\exists c > 0 \; and \; \exists n_0 > 0 \; s.t. \; \forall n. \; n \geq n_0 \Rightarrow c * n \leq f(n) \tag{3.5}$$

The specification of $\Omega(n)$ with Dafny is:

```
predicate IsLinearOrUp(f: nat -> nat)
{
  ∃ c :nat, n0: nat • IsLinearOrUpFrom(c, n0, f)
}
predicate IsLinearOrUpFrom(c: nat ,n0: nat, f: nat -> nat)
{
  ∀ n: nat • n≥n0 ⟹ f(n)≥c*n
}
lemma linearOrUp(c :nat, n0: nat, f: nat -> nat)
  requires IsLinearOrUpFrom(c,n0,f)
  ensures IsLinearOrUp(f)
{}
```

These predicates and the lemma define the mathematical expression of $\Omega(n)$. The *linearOrUp* lemma is provided to merge the two predicates above it to the definition on expression (3.5).

3.2.2 Specification of $\Theta(n)$

Theta notation is used when a function can be bounded both from above and below. The function $f(n)$, belongs to $\Theta(n)$ if:

$$\exists c_1, c_2 > 0 \; and \; \exists n_0 > 0 \; s.t. \; \forall n. \; n \geq n_0 \Rightarrow c1 * n \leq f(n) \leq c2 * n \tag{3.6}$$

The specification of $\Theta(n)$ with Dafny is:

```
predicate IsEqLinear(f: nat -> nat)
{
  ∃ c1 :nat, c2 :nat, n0: nat • IsEqLinearFromTo(c1,c2,n0,f)
}

predicate IsEqLinearFromTo(c1: nat ,c2: nat, n0: nat, f: nat -> nat)
{
  ∀ n: nat • n≥n0 ⟹ c1*n≤f(n)≤c2*n
}

lemma linearEq(c1: nat ,c2: nat, n0: nat, f: nat -> nat)
  requires IsEqLinearFromTo(c1,c2,n0,f)
  ensures IsEqLinear(f)
{}
```

These predicates and the lemma define the mathematical expression of $\Theta(n)$. The *linearEq* lemma is provided to merge the two predicates above it to the definition of expression (3.6).

3.3 Verification of a Function for Big-$\mathcal{O}$, Ω, and Θ Notations

This section demonstrates the process of proving a function that belongs to a set of big-$\mathcal{O}$, Ω, and Θ Notations with a simple example.

If $f(n) = n + 2$, its Dafny definition is:

```
function f(n: nat): nat
{
  n + 2
}
```

We now prove that this expression is of $O(n)$, $\Omega(n)$ and $\Theta(n)$ using calculation lemmas which are replicas of the pen and paper complexity proofs, with a correctness guarantee by Dafny. For this type of implementation, we use Dafny's *calc* construct [Rea20c], a theorem established by a chain of formulas, each transformed into the next. The lemmas specifications are:

```
lemma OLinearCalcLemma(n: nat) returns(c: nat, n0: nat)
  ensures IsLinearFrom(c, n0, f)
  ensures IsLinear(f)

lemma OmegaLinearCalcLemma(n: nat) returns(c: nat, n0: nat)
  ensures IsLinearOrUpFrom(c,n0,f)
```

```
  ensures IsLinearOrUp(f)

lemma ThetaLinearCalcLemma(n: nat) returns(c1: nat, c2: nat, n0: nat
    )
  ensures IsEqLinearFromTo(c1,c2,n0,f)
  ensures IsEqLinear(f)
```

This requires us to find proper constants c (or *c1*, *c2*) and *n0* that satisfy the expressions in (3.1), (3.5), (3.6). The implementations of these lemmas for $f(n) = n + 2$ are:

```
lemma OLinearCalcLemma(n: nat) returns(c: nat, n0: nat)
  ensures IsLinearFrom(c, n0, f)
  ensures IsLinear(f)
{
  if n≥2
  {
    calc ⟸{
      f(n) ≤ c*n;
      f(n) ≤ 2*n ≤ c*n;
      c=3 ∧ n0=2 ∧ n0≤n ∧ (f(n) ≤ 2*n = c*n);
    }
  }

  c, n0 := 2, 2;
  linear(c, n0, f);
}
```

```
lemma OmegaLinearCalcLemma(n: nat) returns(c: nat, n0: nat)
  ensures IsLinearOrUpFrom(c,n0,f)
  ensures IsLinearOrUp(f)
{
  calc ⟸{
    f(n) ≥c*n;
    f(n) ≥ 1*n ≥ c*n;
    c=1 ∧ n0=1 ∧ n0≤n ∧ (f(n) ≥ n = c*n);
  }
  c, n0 := 1, 1;
  linearOrUp(c,n0,f);
}
```

```
lemma ThetaLinearCalcLemma(n: nat) returns(c1: nat, c2: nat, n0: nat
    )
  ensures IsEqLinearFromTo(c1,c2,n0,f)
  ensures IsEqLinear(f)
{
  var n10, n20;
  c1, n10 := OmegaLinearCalcLemma(n);
  c2, n20 := OLinearCalcLemma(n);
  if n10>n20
  {
```

```
    n0 := n10;
  }
  else
  {
    n0 := n20;
  }
  linearEq(c1,c2,n0,f);
}
```

This lemma works through the steps that are required to infer proper constants for $f(n) = n + 2$ to be of $O(n)$, $\Omega(n)$ and $\Theta(n)$ as required.

Chapter 4

The Methodology

We now provide the methodology that we have developed. In Section 4.1, we describe the required steps for the addition of the nonfunctional time complexity property to the specification of the binary search algorithm, in Section 4.2, we present our methodology. We demonstrate the methodology in Section 4.3.

4.1 Case Study

In Chapter 2 we have verified the functional properties of the binary search algorithm. Now we investigate the nonfunctional time complexity property that was added to the specification. Using Dafny, this property can then be verified alongside the functional properties. As described in Section 2.3.1, the binary search algorithm terminates after at most $log_2 n$ steps. To specify the requirement that binary search time complexity belongs to $O(log_2 n)$, we define $T(n)$—binary search time complexity, where T is the time complexity of binary search on input of size $n = |q|$.

$$T(n) = O(log_2 n) \tag{4.1}$$

i.e.,

$$\exists c > 0 \; and \; \exists n_0 > 0 \; s.t. \; \forall n. \; n \geq n_0 \Rightarrow T(n) \leq c * log_2 n \tag{4.2}$$

4.1.1 Specification of Binary Search Logarithmic Time Complexity with Dafny

To specify the binary search algorithm's runtime, we define the time complexity value with a *ghost variable t*. It counts the number of dominant operations performed by the

algorithm [McG08], and it is added to the algorithm's postcondition. The postcondition includes the expression that must hold for logarithmic time complexity as defined and detailed in Section 3.1.3. These *ghost* entities in Dafny are used only during verification and are excluded from the executable code.

```
method binarySearch (q: seq<int>, key: int)
    returns (r: int, ghost t: nat)
  requires Sorted(q)
  ensures BinaryPosts(q, r, key)
  ensures |q|>0 ⟹ IsOLog2n(|q|,t)
  ensures |q|=0 ⟹ t=0
{
  t := 0;
  r:= -1;
  if |q|>0
  {
    var lo, hi := 0, |q|;
    while lo < hi
      invariant BinaryLoop(q, lo, hi, r, key)
      decreases hi-lo
    {
      ...
      t := t+1;
    }
  }
}
```

Here the time complexity variables are inserted into the algorithm's specification. Notice that a condition of a non-empty sequence has been added. This addition is essential for using the sequence size as an input to the Log2 function that is used also for further proves in Section 4.1.3 (the *Log2* function domain consists only of positive numbers). Dafny fails to verify the postcondition

```
ensures|q|>0 ⟹ IsOLog2n(|q|,t)
```

complaining that the postcondition might not hold, due to the non-satisfied expression represented by the predicate *IsOLog2n*:

```
predicate IsOLog2n (n: nat ,t: nat )
{
  ∃ f: nat -> nat • IsLog2(f) ∧ t≤f(n)
}
```

As in traditional proofs, this predicate would be satisfied only when we prove such logarithmic function *f*, that upper-bounds the binary search algorithm's time complexity,

exists. $t \leq f(n)$, and $n > 0$ (the *Log2n* function requires it), this proof can be done by the following lemma:

```
lemma OLog2nProof (n: nat, t: nat)
  requires n>0
  requires t≤f(n)
  ensures IsOLog2n(n, t)
{
  var c, n0 := logarithmicCalcLemma(n);
  logarithmic(c, n0, f);
}
```

This lemma requires our assumed function *f*, as a global variable. The *logarithmicCalcLemma* is calculating the values of *c,n0* that are required to prove that *logarithmic(c,n0,f)*. The lemma specification is:

```
lemma logarithmicCalcLemma(n: nat) returns(c: nat, n0: nat)
  requires n>0
  ensures IsLog2From(c, n0, f)
```

The body of the lemma is written specifically for the function f in Section 4.1.3. The following sections engage in deriving this function f.

4.1.2 Derivation of a tight Upper-Bound Function

We now move from intuition to mathematics. To prove that t is bounded by a logarithmic function of the size of the algorithm's input, a suitable logarithmic function f must be provided. In order to define a function that upper-bounds binary search and is expressed with the *Log2* function, we analyze the behavior of the binary search algorithm's loop in relation to the *Log2* function:

```
var lo, hi := 0, |q|;
while lo < hi
  invariant BinaryLoop(q, lo, hi, r, key)
  decreases hi-lo
{
  var mid := (lo+hi)/2;
  if key < q[mid]
  {
    hi := mid;
  }
  else if q[mid] < key
  {
    lo := mid + 1;
  }
  else
```

```
  {
    r := mid;
    hi := lo;
  }
  t := t+1;
}
```

```
function Log2 (n: nat): nat
  requires n>0
  decreases n
{
  natDivisionDecreases(n, 2);
  if n=1 then 0 else 1 + Log2(n/2)
}
```

It is hard for developers and Dafny to perform the transition from an iterative implementation as in *binary search* to a recursive one as in *Log2*. That is why we reduce the problem as follows:

(a) Defining a recursive transition function that imitates the actions performed in the binary search method, returning the number of recursive calls rather than the key's index or -1.

(b) Proving, by adding loop invariants to the binary search method, that the transition function upper-bounds the value of *t*.

(c) Deriving a function that upper-bounds the transition function and is expressed with *Log2*.

Following the several steps, we transitively prove that the logarithmic function upper-bounds the binary search estimated runtime.

Step ((a)):

```
function TBS (q: seq<int>, lo: int, hi: int, key: int): nat
  requires 0≤lo≤hi≤|q|
  decreases hi-lo
{
  var mid := (lo+hi)/2;
  if hi-lo=0 ∨ |q|=0 then 0
  else if key=q[mid] ∨ hi-lo=1 then 1
  else if key<q[mid] then 1 + TBS(q, lo, mid, key)
  else 1 + TBS(q, mid+1, hi, key)
}
```

This function imitates the binary search algorithm's actions, with the same range of indices and decreases definition, implying the same number of steps as the binary search algorithm's loop. For now, we assume that *TBS* returns the same value as the number of steps the binary search algorithm's loop performs. Next, the focus is on the formal verification made by Dafny to prove the correctness of this assumption.

Step ((b)):

```
while lo < hi
  invariant BinaryLoop(q, lo, hi, r, key)
  invariant t ≤ TBS(q, 0, |q|, key) − TBS(q, lo, hi, key)
  decreases hi−lo
{
    ...
    t := t+1;
}
```

Here the time complexity variables are inserted into the loop's specification. To prove that *TBS* returns the same value as the number of steps the binary search algorithm's loop performs (*ghost var t*), the correctness of the assumption in step (a) must be proven in each loop iteration. Hence, the loop invariant $t \leq TBS(q, 0, |q|, key) - TBS(q, lo, hi, key)$ has been added. This loop invariant holds, thus each decision made during the loop body is also made by the *TBS* function. That is, for each iteration, the step counter t must be upper-bounded by the difference between the algorithm's runtime for the whole sequence to the current part of the sequence. The loop ends when $lo = hi$, where the *TBS* function returns 0. Therefore, the method terminates when the loop invariant holds for $t \leq TBS(q, 0, |q|, key)$, which gives us an upper-bound to the algorithm's time complexity. In the next section, we specify and verify several lemmas to ensure that *TBS* is logarithmic.

Step ((c)): This step we calculate the time complexity of the logarithmic function. As a pen and paper calculation would have been done, we analyze the *TBS* function in relation to the *Log2* function:

```
function TBS (q: seq<int>, lo: int, hi: int, key: int): nat
  requires 0≤lo≤hi≤|q|
  decreases hi−lo
{
```

```
  var mid := (lo+hi)/2;
  if hi-lo=0 ∨ |q|=0 then 0
  else if key=q[mid] ∨ hi-lo=1 then 1
  else if key<q[mid] then 1 + TBS(q, lo, mid, key)
  else 1 + TBS(q, mid+1, hi, key)
}
```

```
function Log2(n: nat): nat
  requires n>0
  decreases n
{
  natDivisionDecreases(n, 2);
  if n=1 then 0 else 1+Log2(n/2)
}
```

We start by analyzing the stop condition. The function *TBS* stops and returns 0 when the size inspected is 0 (whether the size of the sequence is 0 or the difference between hi and lo is 0). However, *Log2* stops and returns 0 when the input is 1. Therefore, 1 is chosen to be added to the mathematical expression of *TBS*. Continuing with the analysis of the value that is divided by 2, *TBS* takes the value of *lo+hi*, sum of 2 indices in the sequence. The value of this sum can be at most double of size of the sequence minus 1 (if hi is $|q|$ and lo is $|q| - 1$). Therefore the mathematical expression of *TBS* must be doubled. The last thing must be considered is that *Log2* input is equal or greater than 1. The behavior of the rest of the functions is similar. In each call only a part of the initial size is assigned and the result increases by 1. The result is the expression:

$$n > 0 \Rightarrow (TBS(q, lo, hi, key) \leq 2 * log_2(|q| + 1) + 1) \tag{4.3}$$

This analysis has also been done the same way as pen and paper analysis. That being so, we now provide a proof that the analysis holds with the following lemma:

```
lemma TBSisLog (q: seq<int>, lo: nat, hi: nat, key: int)
  requires |q|>0
  requires 0 ≤ lo < hi ≤ |q|
  decreases hi-lo
  ensures TBS(q, lo, hi, key) ≤ 2*Log2(hi-lo)+1
{
  var mid := (lo+hi)/2;
  if key<q[mid] ∧ 1<hi-lo
  {
    TBSisLog(q, lo, mid, key);
  }
  else if key>q[mid] ∧ hi-lo>2
```

```
  {
    log2Mono(hi-(mid+1), (hi-lo)/2);
    TBSisLog(q, mid+1, hi, key);
  }
}
```

This lemma adds the missing requirements for enabling the use of the log function: the precondition requires the sequence is non empty, and the *lo* index to be strictly smaller than the *hi* index. The *ensures* expression is the proof obligation as in expression (4.3). This lemma calls itself recursively. The recursive call is treated in accordance with programming rules: the precondition of the callee is checked, termination is checked, and the postcondition can be assumed. In effect, this sets up a proof by induction, where the recursive call to the lemma acts as an inductive step. The lemma performs the same actions *TBS* does, excluding cases where the new preconditions ($lo < hi$ *and* $0 < |q|$) are violated. The *log2Mono* lemma is for Dafny to understand that *Log2* is a monotonic function.

```
lemma log2Mono (x: nat, y: nat)
  requires x>0 ∧ y>0
  ensures y≥x ⟹ Log2(y)≥Log2(x)
  decreases x, y
{
  if x≠1 ∧ y≠1 {log2Mono(x-1,y-1);}
}
```

The *log2Mono* will be used again for the *Log2* function's requirements. Since a natural number can be of value 0 in our context, the upper-bounded function is increased from $(2 * log_2(|q|) + 1)$ to $(2 * log_2(|q| + 1) + 1)$. Now it has been proven transitively that the binary search algorithm has a mathematical function, expressed with *Log2*, that upper-bounds the variable t and verified by Dafny with the help of the lemmas *log2Mono* and *TBSisLog*. To make sure Dafny verifies this upper-bound for t, we insert a temporary *assertion* at the end of the method. This assertion will later be replaced by a lemma.

```
method binarySearch (q: seq<int>, key: int)
    returns (r: int, ghost t: nat)
  requires Sorted(q)
  ensures BinaryPosts(q, r, key)
  ensures |q|>0 ⟹ IsOLog2n(|q|, t)
  ensures |q|=0 ⟹ t=0
{
  t := 0;
```

```
  r:= -1;
  if |q|>0
  {
    var lo, hi := 0, |q|;
    while lo < hi
      invariant BinaryLoop(q, lo, hi, r, key)
      invariant t ≤ TBS(q, 0, |q|, key) -
                    TBS(q, lo, hi, key)
      decreases hi-lo
    {
      var mid := (lo+hi)/2;
      if key < q[mid]
      {
        hi := mid;
      }
      else if q[mid] < key
      {
        lo := mid + 1;
      }
      else
      {
        r := mid;
        hi := lo;
      }
      t := t+1;
    }
    TBSisLog(q,0,|q|,key);
    log2Mono(|q|,|q|+1);
  }
  assert t ≤ 2*log2(|q|+1)+1;
}
```

Dafny still fails to verify the postcondition $|q| > 0 \Rightarrow IsOLog2n(|q|, t)$, since it has not yet proven that $f(n) = 2 * log_2(n+1) + 1$ is logarithmic. For this proof, the lemmas that have been specified in Section 4.1.1 defined.

4.1.3 Binary Search Upper-Bounding Function is Logarithmic

So far it has been proven that the binary search runtime upper-bound is expressed with *Log2*. We now prove that this expression is $O(log_2 n)$ using the lemmas *OLog2nProof* and *logarithmicCalcLemma*. This requires us to find proper *c* and *n0* that satisfy the expression in (4.2). For implementing the body of *logarithmicCalcLemma* we use Dafny's *calc* construct [LP13].

```
lemma logarithmicCalcLemma (n: nat) returns (c :nat, n0:nat)
  requires n>0
```

```
  ensures IsLog2From(c,n0,f)
{
  calc {
    f(n);
  =
    2*log2(n+1) + 1;
  ≤
    2*log2(n+1) + log2(n+1);
  =
    3*log2(n+1);
  ≤  {assert n≥1; log2Mono(n+1,2*n);}
    3*log2(2*n);
  =
    3*(1+log2(n));
  }
  assert f(n)≤ 3*(1+log2(n));
  assert n≥2 ⟹ (f(n)≤6*log2(n));
  c, n0 := 6, 2;
}
```

This lemma works through the steps that are required to infer proper *c*, *n0* for $f(n) = 2 * log_2(n+1) + 1$. Since this lemma ensures that f is logarithmic, the conditions of the *OLog2nProof* lemma now hold.

```
function f (n: nat) : nat
{
  2*Log2(n+1) + 1
}
```

```
lemma OLog2nProof (n: nat, t: nat )
  requires n>0
  requires t≤f(n)
  ensures IsOLog2n(n, t)
{
  var c, n0 := logarithmicCalcLemma(n);
  logarithmic(c, n0, f);
}
```

This lemma can be used now, since the implementation has already been limited to non-empty inputs and proved that the steps counter t value is at most the value of the function f in Section 4.1.2(c). Also, $f(n)$ is proven to be logarithmic by the definitions that have been determined in Section 3.1.3.

Finally, the proof that the binary search algorithm's runtime is logarithmic is complete:

```
method binarySearch (q: seq<int>, key: int)
    returns (r: int, ghost t: nat)
```

```
  requires Sorted(q)
  ensures BinaryPosts(q, r, key)
  ensures |q|>0 ==> IsOLog2n(|q|, t)
  ensures |q|=0 ==> t=0
{
  t := 0;
  r:= -1;
  if |q|>0
  {
    var lo, hi := 0, |q|;
    while lo < hi
      invariant BinaryLoop(q, lo, hi, r, key)
      invariant t ≤ TBS(q, 0, |q|, key) -
                    TBS(q, lo, hi, key)
      decreases hi-lo
    {
      var mid := (lo+hi)/2;
      if key < q[mid]
      {
        hi := mid;
      }
      else if q[mid] < key
      {
        lo := mid + 1;
      }
      else
      {
        r := mid;
        hi := lo;
      }
      t := t+1;
    }
    TBSisLog(q,0,|q|,key);
    log2Mono(|q|,|q|+1);
    OLog2nProof(|q|,t);
  }
}
```

The postcondition *IsOLog2n* sums up all the required terms for having $O(log_2(n))$ time complexity as defined in expression 4.1. Now our process is complete.

4.2 Definition of the Methodology

Reflecting on the presented process in the previous section, it appears that the methodology for the specification and verification of time-complexity properties of algorithms involves the following steps:

1. *Defining big-$\mathcal{O}$ notation.* Define mathematical theorems with predicates to create expressions of the big-$\mathcal{O}$ notation for the desired complexity class. This definition has to be a function of both runtime and input size of the algorithm.

2. *Specifying the functional and time complexity requirements for the Algorithm Declaration* Specify the functional and time complexity requirements of the algorithm. Add it to the postcondition, using the definition of the big-$\mathcal{O}$ of Step 1.

3. *Verifying the correctness of the algorithm.* Implement a functionally verified version of the algorithm. If there is already a verified implementation, it can be used as long as it has only a single return point—at the end of the method. We implement the algorithm in this thesis with a single return point to simplify the calculation of the number of steps the algorithm performs. It would be interesting also to support algorithms with more than one return point.

4. *Count the number of elementary operations.* Add to the algorithm code, the time complexity using a ghost variable that counts the number of elementary operations performed by the algorithm.

5. *Deriving the function that upper-bounds the time complexity.* Using traditional algorithm analysis, derive a mathematical function to upper-bound the time complexity of the algorithm.

6. *Proving the function belongs to the target complexity class.* Specifying and proving lemmas with proofs by stepwise formula manipulation verifies that the function belongs to the complexity class defined with the big-$\mathcal{O}$ notation in Step 1.

7. *Integrating the time complexity elements with the code.* Adding the annotations provided by the products of Steps 4 and 5, to make the postcondition of Step 3 hold.

4.3 Demonstration of the Methodology

This section demonstrates the methodology's effectiveness by projecting it on iterative and recursive implementations of known algorithms of different complexity classes. These

$n =$	0	1	2	3	4	5	6	7	8	9
$x_n =$	0	1	1	2	3	5	8	13	21	34

Figure 4.1: A Fibonacci sequence.

implementations cover all the complexity classes that we have verified in Chapter 3 and consist of: (1) Iterative Fibonacci (2) Recursive Fibonacci (3) insertion Sort

4.3.1 Fibonacci Algorithm - Iteration vs. Recursion

Fibonacci series is defined as a sequence of numbers in which the first two numbers are 0 and 1, and each subsequent number is the sum of the previous 2. Thus, in this series, the nth term is the sum of $(n-1)^{th}$ term and $(n-2)^{th}$ term.

Mathematically, the nth term of the Fibonacci series can be represented as:

$$X_n = X_{n-1} + X_{n-2} \tag{4.4}$$

Fig 4.1 presents a Fibonacci sequence.

Fibonacci can be solved iteratively as well as recursively.

The iterative implementation, includes 2 variables that always stores the $(n-1)^{th}$ term and $(n-2)^{th}$ term (starting with 0 and 1), and a loop from 1 to n that sums them up, and assigns the n^{th} term as the new $(n-1)^{th}$ term and the $(n-1)^{th}$ term as the new $(n-2)^{th}$ term. A theoretical algorithm analysis results that the time complexity of the iterative implementation is linear, as the loop runs from 1 to n. i.e. it runs in O(n) time.

The recursive implementation takes n as an input, which will refer to the n^{th} term of the sequence that we want to be computed. If n is a number that is smaller than 2 (0 or 1), we return n. Else, we return the sum of the results of two recursive calls; a call with the $(n-1)^{th}$ term and $(n-2)^{th}$ term.

Here is a standard recursive implementation of the Fibonacci function (implemented with Dafny):

```
function Fib(n: nat): nat
{
  if n≤1 then n else Fib(n−1) + Fib(n−2)
}
```

A theoretical algorithm analysis results that the time complexity of the recursive implementation is described by the following recurrence relation:

$$T(n) = T(n-1) + T(n-2) + 4 \tag{4.5}$$

Where T(n) is the runtime of Fib(n) (number of elementary operations required to calculate the n^{th} term of the sequence), T(n-1) the runtime of Fib(n-1), T(n-2) the runtime of Fib(n-2) and the 4 stands for 1 comparison, 2 subtractions, 1 addition. Observing the behavior of the algorithm, it is easy to see that $T(n-1) \leq T(n-2)$. Since we are looking for an upper-bound, we can approximating that $T(n-1) \approx T(n-2)$ and now the time complexity can be described by:

$$T(n) = 2 * T(n-1) + 4 \tag{4.6}$$

For solving this recurrence relation, we iterate through the values of n until we find a pattern which can determine a closed formula.

$$T(n) = T(n-1) + T(n-2) + 4 \tag{4.7}$$

$$\leq 2T(n-1) + 4 \text{//from the approximation T(n-1)} \approx \text{T(n-2)}$$

$$= 2 * (2T(n-2) + 4) + 4 = 4T(n-2) + 3 * 4$$

$$= 8T(n-3) + 7 * 4$$

$$= 16T(n-4) + 15 * 4$$

$$\ldots$$

$$= 2^k * T(n-k) + (2^k - 1) * 4$$

When k==n-1 then:

$$T(n) = 2^{n-1} * T(1) + (2^{n-1} - 1) * 4 = 2^{n-1} + 2 * 2^{n-1} - 4 \leq 3 * 2^n \tag{4.8}$$

Hence the time taken by recursive Fibonacci is $O(2^n)$ or exponential.

4.3.1.1 Verification of the Iterative Fibonacci Algorithm Time Complexity

Now we use our methodology to implement a verified version of an iterative Fibonacci algorithm.

1. *Defining big-$\mathcal{O}$ notation*: This algorithm has been analyzed and assumed to be of O(n)—the O(n) definitions with Dafny detailed in Section 3.1.1.

2. *Specifying the functional and time complexity requirements for the Algorithm Declaration*: For verifying the functional requirement of Fibonacci as in expression (4.4), we use the function *Fib*:

```
function Fib(n: nat): nat
{
  if n≤1 then n else Fib(n−1) + Fib(n−2)
}
```

As presented by the case study in Section 4.1.1, we define the time complexity value with a *ghost variable t*. We use the *IsOn* predicate to ensure the algorithm is linear:

```
predicate IsOn (n: nat ,t: nat )
{
  ∃ f: nat −> nat • IsLinear(f) ∧ t≤f(n)
}
```

Now the iterative Fibonacci method can be declared with functional and time complexity specifications:

```
method ComputeFib(n: nat) returns (x: nat, ghost t: nat)
  ensures x = Fib(n)
  ensures IsOn(n,t)
```

3. *Verifying the correctness of the algorithm*: As described in Section 4.3.1, the method body is given below:

```
if n≤1
{
  x := n;
}
else
{
  var i,a:= 1,0;
  x := 1;
  while i < n
    invariant FibLoop(n,x,a,i)
  {
    a, x := x, a+x;
    i := i+1;
```

```
  }
}
```

The *decreases* annotation is not provide explicitly in this method, since Dafny is able to guess the right annotation (*which is decreases n-i*) and do this proof on its own. The Fibonacci algorithm's loop invariants are grouped into a predicate for a cleaner implementation:

```
predicate FibLoop(n: nat, x: nat, a: nat, i: nat)
{
  0<i≤n ∧ a=Fib(i−1) ∧ x=Fib(i)
}
```

This predicate is composed of three expressions that must hold for every iteration. The first expression defines the variable which holds the iteration number i runs from 1 to n. The second and third expressions makes sure that the variable a and the returned value x are always the $i-1^{th}$ and the n^{th} terms of the Fibonacci sequence respectively. If the loop invariant holds, the postcondition *ensures fib(n)==x* is satisfied when i==n. Thus, the algorithm's functional properties are verified.

4. *Count the number of elementary operations*: To measure the algorithm's runtime, we add the operations counter t to the algorithm's body:

```
if n≤1
{
  x := n;
  t := 1;
}
else
{
  var i,a:= 1,0;
  x := 1;
  t := 3;
  while i < n
    invariant 0<i≤n
    invariant FibLoop(n,x,a,i)
  {
    a, x := x, a+x;
    i := i+1;
    t := t+1;
  }
}
```

We assign t with the count of the comparison on each condition (1 for the if or 1 for the else), and the 2 variables creation in the else code. We add 1 for each iteration. Dafny

fails to verify the postcondition

```
ensures IsOn(n,t)
```

complaining that the postcondition might not hold, due to the non-satisfied predicate:

```
predicate IsOn (n: nat ,t: nat )
{
  ∃ f: nat -> nat • IsLinear(f) ∧ t≤f(n)
}
```

As in traditional proofs, this predicate would be satisfied only when we prove such linear function *f*, that upper-bounds the algorithm's time complexity, exists. This can be proved by the following lemma:

```
lemma OnProof(n: nat, t: nat)
  requires t≤f(n)
  ensures IsOn(n, t)
{
  var c,n0 := linearCalcLemma(n);
  linear(c, n0, f);
}
```

This lemma requires our assumed to exist function *f*, as a global variable. The *linearCalcLemma* is calculating the values of *c,n0* that are required to prove that *linear(c,n0,f)*. The lemma specification is:

```
lemma linearCalcLemma(n: nat) returns(c: nat, n0: nat)
  ensures IsLinearFrom(c,n0,f)
```

The body of the lemma is written specifically for the function *f* we derive in the next step.

5. *Deriving the function that upper-bounds the time complexity* In order to define a function that upper-bounds the algorithm, we analyze the worst-case scenario, which leads us to the else condition and requires to analyze the behavior of the algorithm's loop in relation to the input:

```
while i < n
  invariant 0<i≤n
  invariant FibLoop(n,x,a,i)
{
  a, x := x, a+x;
  i := i+1;
  t := t+1;
}
```

In this case deriving the function is quite easy, it is clear that the loop iterate from 1 to n, hence t is added with i-1 on each iteration. Since this path starting point is t==3, the loop invariant requires to verify that $t == i + 2$;

```
while i < n
  invariant 0<i≤n
  invariant FibLoop(n,x,a,i)
  invariant t=i+2
{
  a, x := x, a+x;
  i := i+1;
  t := t+1;
}
```

The method terminates when i==n, and the loop invariant holds for $t == i + 2$. Hence, our upper-bounding function is $f(n) = n + 2$.

6. *Proving the function belongs to the target complexity class* Dafny still fails to verify the postcondition *IsOn*, since it has not yet proven that $f(n) = n + 2$ is linear. For this proof, we implement the body of the *linearCalcLemma* and use the *OnProof* that were introduced in step 4. This requires us to find proper c and *n0* that satisfy the expression:

$$\exists c > 0 \; and \; \exists n_0 > 0 \; s.t. \; \forall n. \; n \geq n_0 \Rightarrow n + 2 \leq c * n \tag{4.9}$$

For implementing the body of *linearCalcLemma* we use Dafny's *calc*.

```
lemma linearCalcLemma(n: nat) returns(c: nat, n0: nat)
  ensures IsLinearFrom(c,n0,f)
{
  if n=0
  {
    assert ∀ k: nat • f(n) ≥ k * n;
  }
  else
  {
    assert 1≤n;
    calc ⟸{
      f(n)≤c*n;
      f(n)≤2*n+1≤c*n;
      f(n)≤2*n+1≤3*n≤c*n;
      c=3 ∧ n0=1 ∧ n0≤n ∧ (f(n)≤2*n+1≤3*n≤ c*n);
    }
  }

  c, n0 := 3, 1;
}
```

This lemma works through the steps that are required to infer proper *c*, *n0* for $f(n) = n + 2$. Since this lemma ensures that f is linear, the conditions of the *OnProof* lemma now hold. The *OnProof* lemma can be used now.

Step 7. *Integrating the time complexity elements with the code*

Finally, the functionally and time complexity verified version of the algorithm is complete:

```
method ComputeFib(n: nat) returns (x: nat, ghost t: nat)
  ensures x = Fib(n)
  ensures IsOn(n,t)
{
  if n≤1
  {
    x := n;
    t := 1;
  }
  else
  {
    var i,a:= 1,0;
    x := 1;
    t := 3;
    while i < n
      invariant FibLoop(n,x,a,i)
      invariant t=i+2
    {
      a, x := x, a+x;
      i := i+1;
      t := t+1;
    }
  }
  OnProof(n,t);
}
```

The postcondition *IsOn* sums up all the required terms for having $O(n)$ time complexity as defined in expression 4.9.

4.3.1.2 Verification of the Fibonacci Recursive Algorithm Time Complexity

Now we use our methodology to implement a verified version of a recursive Fibonacci algorithm.

1. *Defining big-$\mathcal{O}$ notation*: This algorithm has been analyzed theoretically and assumed to be of $O(2^n)$. The $O(2^n)$ definitions with Dafny detailed in Section 3.1.4.

2. *Specifying the functional and time complexity requirements for the Algorithm Decla-*

ration: Since the result of the recursive Fibonacci is identical to the result of the iterative Fibonacci, The functional verification leans on the same function *Fib*. We use the *ghost variable t* and the *IsOExpn* predicate (presented in 3.1.4) to ensure the algorithm is exponential:

```
predicate IsOExpn (n: nat ,t: nat )
{
  ∃ f: nat -> nat • IsExpo(f) ∧ t≤f(n)
}
```

Now the recursive Fibonacci method can be declared with functional and time complexity specifications:

```
method ComputeFib(n: nat) returns (b: nat, ghost t: nat)
  ensures b = Fib(n)
  ensures IsOExpn(n,t)
```

3. *Verifying the correctness of the algorithm*: For the recursive implementation of Fibonacci as described in Section 4.3.1, we use *ComputeFibRec* as the *ComputeFib* body:

```
method ComputeFib(n: nat) returns (x: nat, ghost t: nat)
  ensures x = Fib(n)
  ensures IsOExpn(n,t)
{
  x := ComputeFibRec(n);
}

method ComputeFibRec(n: nat) returns (x: nat)
  ensures x = Fib(n)
{
  if n≤1
  {
    x := n;
  }
  else
  {
    var x1, x2;
    x1 := ComputeFibRec(n-1);
    x2 := ComputeFibRec(n-2);
    x := x1+x2;
  }
}
```

Dafny easily verifies this implementation for the functional part. Since the method is recursive, the proof corresponds to a proof by induction. The recursive behavior of the method is similar to the behavior of the *Fib* function. The *decreases* annotation is not provided explicitly in this method too. Dafny guesses the right annotation, which is

(*decreases n*) and does this proof on its own. The algorithm's functional properties are verified.

4. *Count the number of elementary operations*: To measure the algorithm's runtime, we add the operations counter t to *ComputeFibRec* decleration and body:

```
method ComputeFibRec(n: nat) returns (x: nat, ghost t: nat)
  ensures x = Fib(n)
{
    if n≤1
    {
      x := n;
      t := 1;
    }
    else
    {
      var x1, x2;
      ghost var t1, t2;
      x1,t1 := ComputeFibRec2(n-1);
      x2,t2 := ComputeFibRec2(n-2);
      x := x1+x2;
      t := t1+t2+4;
    }
}
```

We count the runtime of the algorithm the same way as described in 4.5. Since we have not yet established a postcondition for t in *ComputeFibRec*, *ComputeFib* has no information on t and Dafny still fails to verify the *IsOExpn* postcondition. To proof this postcondition we must derive from *ComputeFibRec* a function f, that upper-bounds the algorithm's time complexity.

5. *Deriving the function that upper-bounds the time complexity* This algorithm includes two recursive calls. In this case, it is not easy to derive the function that upper-bounds the algorithm in contrast to the iterative implementation. Therefore, we turn to the methods of function derivation we presented on a more complex algorithm as in the case study, and adjust them to this algorithm's specific needs. We analyze the runtime of the algorithm to derive the function in 3 sub-steps:

(a) Defining a recursive transition function that imitates the method's actions, returning the number of recursive calls rather than the n^{th} term.

(b) Proving, by adding lemmas to the method, that the transition function upper-

bounds the value of t.

(c) Deriving a function that upper-bounds the transition function and is expressed with 2^n (an exponential function).

Following the respective sub-steps, we transitively prove that the exponential function upper-bounds the binary search estimated runtime. Step (a)

```
function FibTime(n: nat): nat
{
  if n≤1 then 1 else FibTime(n − 1)+FibTime(n − 2)+4
}
```

This function imitates the actions performed by the algorithm. For now, we assume that *FibTime* returns the same value as the number of steps the algorithm's loop performs. Next, we verify this assumption. Step (b)

```
method ComputeFibRec(n: nat) returns (x: nat, ghost t: nat)
  ensures x = Fib(n)
  ensures t = FibTime(n)
```

Here the time complexity variables are inserted into *ComputeFibRec* specification, which is verified automatically by Dafny. In the next section we specify and verify several lemmas to ensure that *FibTime* is exponential.

Step (c) As a pen and paper calculation would have been done, we analyze the *FibTime* function in relation to the 2^n function:

```
function Power(n: nat, c: nat): nat
  decreases c
{
  if c=0 then 1 else if n=0 then 0 else n * Power(n,c−1)
}
```

Where n=2 and c=n. We start by analyzing the stop condition. *FibTime* stops and returns 1 when n is 0 or 1. However, 2^n stops and returns 1 or 2 when the input is 0 or 1 respectively. Therefore, 2^n upper bounds *FibTime* for $n \leq 1$. Continuing with the analysis of the else path, *FibTime* takes the value of FibTime(n-1)+FibTime(n - 2)+4, as the recurrence relation in 4.5. Similar to the technique we used in the recurrence relation solution, we first prove that $FibTime(n-2) \leq FibTime(n-1)$ with Dafny:

```
lemma monoFib(t1: nat ,t2: nat , n1: nat ,n2: nat )
  requires n1≥n2
  requires t1=FibTime(n1) ∧ t2=FibTime(n2)
  ensures t1≥t2
{
  if n2<n1−1
  {
    var t1' := FibTimeCalc(n1−1);
    monoFib(t1',t2,n1−1,n2);
  }
}
```

Now we required to provide a process which leads us to derive the desired exponential function:

```
lemma fibTimeIsExp(n: nat)
  ensures FibTime(n)≤3∗Power(2,n)
{
  if n>1
  {
    var t1 :=FibTimeCalc(n−1);
    var t2 :=FibTimeCalc(n−2);
    monoFib(t1,t2,n−1,n−2);
    assert FibTime(n)≤ 2∗FibTime(n−1)+4;
    assert FibTime(n)≤ 6∗FibTime(n−1);
    fibTimeExp(n−1);
    expAttribute(n−1);
  }
}
```

This lemma defines and performs the process of deriving the exponential function. For calculating the values of FibTime(n-1) and FibTime(n-2) we use a *function method*. A function method, unlike a function that can only be use for specification, can be called from a real code. The *FibTimeCalc* function method structured exactly like *FibTime*, only returns a value which can be used as part of *fibTimeIsExp*. This function method describes the analysis stages which concludes that $FibTime(n) <= 3 * 2^n$. It required to provide an inductive proof and a lemma to establish a known attribute of the power function: $2 * 2^n = 2^{n+1}$.

```
lemma expAttribute(n: nat )
  ensures 2∗Power(2,n)=Power(2,n+1)
{}
```

Now it has been proven that the algorithm has a mathematical function, expressed with 2^n, that upper-bounds the variable t and verified by Dafny with the help of the lemmas

monoFib and *fibTimeIsExp*.

```
method ComputeFibRec(n: nat) returns (x: nat, ghost t: nat)
  ensures x = Fib(n)
  ensures t = FibTime(n)≤3*Power(2,n)
{
  if n≤1
  {
    x := n;
    t := 1;
  }
  else
  {
    var x1, x2;
    ghost var t1, t2;
    x1,t1 := ComputeFibRec(n-1);
    x2,t2 := ComputeFibRec(n-2);
    x := x1+x2;
    t := t1+t2+4;
    monoFib(t1,t2,n-1,n-2);
    fibTimeExp(n);
  }
}
```

6. *Proving the function belongs to the target complexity class* Dafny still fails to verify the postcondition *IsOExpn*, since it has not yet proven that $f(n) = 3*2^n$ is of $O(2^n)$. For this proof, we implement *expCalcLemma* and use *OExpnProof* that would be presented next. This requires us to find proper c and $n0$ that satisfy the expression: The condition for $f(n)$ with $0 \leq n$ to belong to $O(2^n)$ is:

$$\exists c > 0 \; and \; \exists n_0 > 0 \; s.t. \; \forall n. \; n \geq n_0 \Rightarrow 3 * 2^n \leq 2^{n^c} \tag{4.10}$$

For implementing the body of *expCalcLemma* we use Dafny's *calc*.

```
lemma expCalcLemma(n: nat) returns(c: nat, n0: nat)
  ensures IsExpoFrom(c, n0, f)
{

  calc ⟸{
    f(n)≤Power(2,Power(n,c));
    {calc1(n);}
    f(n)≤Power(2,n+2)≤Power(2,Power(n,c));
  }
  if n≥2
  {
    calc ⟸{
      f(n)≤Power(2,n+2)≤Power(2,Power(n,c));
      {calc2(n);}
```

```
        f(n)≤Power(2,n+2)≤Power(2,Power(n,2))≤Power(2,Power(n,c));
      }
    }
    c, n0 := 2,2;
    expLemmaHelper1 (c, n0);
    calc1(n);
    assert ∀ k: nat • f(k)≤Power(2,k+2);
    expLemmaHelper2 (c, n0);
}
```

This lemma works through the steps that are required to infer proper *c*, *n0* for $f(n) = 3 * 2^n$. The verification process for this exponential function is trickier for Dafny, hence the process consists of 3 parts:

(i) A Display of the calculation steps which proves that $f(n) \leq 2^{n+2}$ using the lemma *calc1*

```
lemma calc1(n: nat)
  ensures f(n)≤Power(2,n+2);
{
  calc{
      f(n);
    =
      3*Power(2,n);
    ≤
      4*Power(2,n);
    =
      2*Power(2,n+1);
    =
      Power(2,n+2);
    }
}
```

(ii) A Display of the calculation steps which proves that if $n \geq 2$ than $f(n) \leq 2^{n^2}$ using the lemma *calc2*.

```
lemma calc2(n: nat)
  requires f(n)≤Power(2,n+2)
  ensures n≥2 ⟹ f(n)≤Power(2,Power(n,2))
{
  if n≥2
  {
    calc≤{
      f(n);

      Power(2,n+2);
      {powerMono(n+2,n+n);}
      Power(2,n+n);
```

```
        Power(2,2*n);
        {mul_inequality(2,n);powerMono(2*n,n*n);}
        Power(2,n*n);

        Power(2,Power(n,2));
      }
    }
}
```

(iii) A Derivation of suitable c and n_0 for proving the expression in 4.10. The proof is aided by 2 lemmas: *expLemmaHelper1* and *expLemmaHelper2*.

```
lemma expLemmaHelper1 (c: nat, n0: nat)
  requires c=2 ∧ n0=2
  ensures IsExpoFrom(c,n0,g)
{
  assert ∀ n: nat ,m: nat • m>n>0 ⟹ Power(2,m)≥ 2*Power(2,n);
  assert ∀ n: nat • n≥n0 ⟹g(n)≤Power(2,2*n);
  assert ∀ n: nat • n≥n0 ⟹g(n)≤Power(2,n*n);
  assert ∀ n: nat • n≥n0 ⟹g(n)≤Power(2,Power(n,c));
}
lemma expLemmaHelper2 (c: nat, n0: nat)
  requires IsExpoFrom(c,n0,g)
  requires ∀ n: nat • n≥n0 ⟹f(n)≤g(n);
  ensures IsExpoFrom(c,n0,f)
{}
function g(n: nat) : nat
{
  Power(2,n+2)
}
```

Since this lemma ensures that f is exponential, the conditions of the *OExpnProof* lemma now hold.

```
lemma OExpnProof(n: nat , t: nat )
  requires t≤f(n)
  ensures IsOExpn(n,t)
{
  var c,n0 := expCalcLemma(n);
  exponential(c,n0,f);
}
```

Step 7. *Integrating the time complexity elements with the code* Finally, the functionally and time complexity verified version of the algorithm is complete:

```
method ComputeFib(n: nat) returns (x: nat, ghost t: nat)
  ensures x = Fib(n)
  ensures IsOExpn(n,t)
```

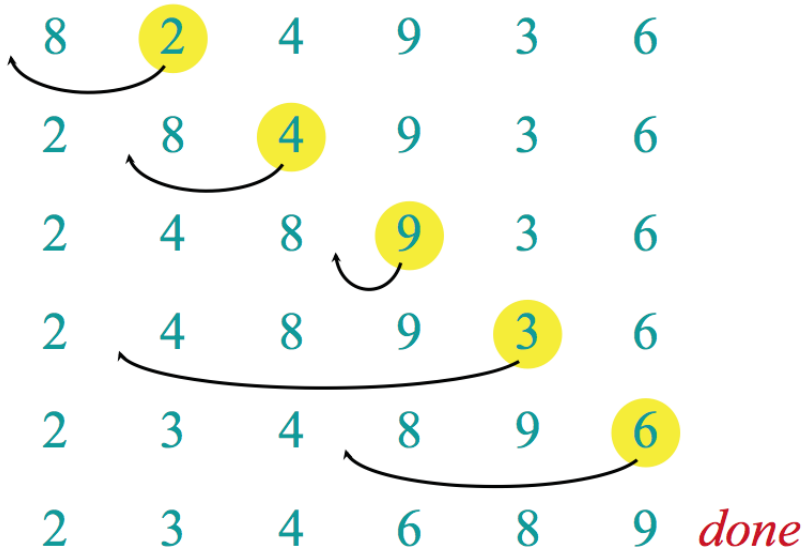

Figure 4.2: A running example of insertion sort.

```
{
  x, t := ComputeFibRec(n);
  OExpnProof(n, t);
}
```

The postcondition *IsOExpn* sums up all the required terms for having $O(2^n)$ time complexity as defined in expression 4.10.

4.3.2 Insertion Sort

The insertion sort algorithm iterates through an input array and removes one element per iteration, finds the place the element belongs in the array, and then places it there. This process results in a sorted list from left to right. Fig 4.2 presents a running example of the algorithm.

The worst case for insertion sort will occur when the input list is in decreasing order. To insert the last element, we need at most n-1 comparisons and at most n-1 swaps. To insert the second to last element, we need at most n-2 comparisons and at most n-2 swaps, and so on. We count the number of comparisons needed to perform in insertion sort worst case as the algorithm's upper-bounding runtime, which is:

$$T(n) = 2 * (1 + 2 + \cdots + n - 2 + n - 1) = (n - 1)(n - 2) = O(n^2) \tag{4.11}$$

We used the formula for a sum of integers between 1 to n-1:

$$1 + 2 + \cdots + n - 2 + n - 1 = \frac{(n-1)(n-2)}{2} \tag{4.12}$$

Now we use our methodology to implement a verified version of an insertion sort.

1. *Defining big-$\mathcal{O}$ notation*: This algorithm has been analyzed and assumed to be of O(n^2). The O(n^2) definitions with Dafny detailed in Section 3.1.2.

2. *Specifying the functional and time complexity requirements for the Algorithm Declaration*: For verifying the functional requirement of insertion sort, we need to ensure the array is sorted and that its content is not changed (i.e., the number of copies of each element is the same). For this causes we create a predicate for the postconditions

```
predicate SortPosts(q: seq<int>, oldq: seq<int>)
{
  Sorted(q) ∧ multiset(q)=multiset(oldq)
}
```

which contains the *Sorted* predicate we have already used in the binary search precondition,

```
predicate Sorted(q: seq<int>)
{
  ∀ i,j • 0 ≤ i ≤ j < |q| ⟹ q[i] ≤ q[j]
}
```

and the structure *Multiset* [Rea20c]. For verifying the number of copies of each element is the same in the pre and post sort array, we use the *old* expression. An old expression is used in postconditions where old(e) evaluates to the value expression e had on entry to the current method.

```
   ensures SortPosts(a[..],old(a[..]))
```

Next we define the time complexity value with a *ghost variable t*. We use the *IsOn2* predicate to ensure the algorithm is quadratic:

```
predicate IsOn2 (n: nat ,t: nat )
{
  ∃ f: nat -> nat • IsQuadratic(f) ∧ t≤f(n)
}
```

Now the insertion sort method can be declared with functional and time complexity specifications:

```
method insertionSort(a: array<int>) returns(ghost t: nat)
  modifies a
  ensures SortPosts(a[..],old(a[..]))
  ensures IsOn2(a.Length,t)
```

The *modifies* annotation is required to list which parts of memory we modify.

3. *Verifying the correctness of the algorithm*: The method body is given below.

```
if a.Length>0
{
  var i := 1;
  while(i < a.Length)
    invariant IsValidOuterloop(a[..],old(a[..]),i)
  {
    sortSubArr(a,a[..],i);
    i := i + 1;
  }
}
```

The *decreases* annotation is not provide explicitly in this method, since Dafny is able to guess the right annotation (*which is decreases a.Length-i*) and do this proof on its own. The algorithm's loop invariants are grouped into a predicate for a cleaner implementation:

```
predicate IsValidOuterloop(q: seq<int>, oldq: seq<int>, i: int)
{
  0≤i≤|q|=|oldq| ∧ Sorted(q[..i]) ∧ multiset(oldq) = multiset(q)
}
```

This predicate is composed of three expressions that must hold for every iteration. The first expression defines the variable which holds the iteration number i that runs from 1 to the array's size. It also makes sure that the array size does not change along the algorithm's run. The second expression verifies that the left part of the array, the one we already arranged by the algorithm, is indeed sorted. The third expression makes sure that the number of copies of each element is the same. If the loop invariant holds, the postconditions are satisfied when i==a.Length. Thus, the algorithm's functional properties are verified.

The loop contains an inner loop which is implemented by the method:

```
method sortSubArr(a: array<int>, olda: seq<int>, i: int)
  modifies a
  requires 0<i<a.Length=|olda| ∧ IsValidOuterloop(a[..], olda, i)
  ensures SortSubArrPosts(a[..],olda,i)
```

this method's specification ensures us that the following postconditions hold on each outer loop iteration:

```
predicate SortSubArrPosts(q: seq<int>, oldq: seq<int>, i: int)
{
  0<i<|q|=|oldq| ∧ Sorted(q[..i+1]) ∧ multiset(q)=multiset(oldq)
}
```

The *SortSubArr* method body performs the innerloop of the algorithm. This loop iterates from the index i we have reached to in the outerloop- backwards, and swaps it with elements that are greater. This ensures that all the elements left to the index if the element from the outerloop are sorted.

```
var j := i;
ghost var oldi := i;
ghost var q := a[..];
while(0<j ∧ a[j]<a[j-1])
  invariant IsValidInnerloop(a[..],q,olda,j,i,oldi)
  decreases j
{
  ghost var q' := a[..];
  swap(a, j-1, j);
  afterSwap(a[..],q,q',j,i);
  j := j - 1;
}
```

The ghost variables q and oldi are not compiled to the algorithm's code, but used for ensuring that the outerloop index and the number of copies of each element is the same respectively. We use the *IsValidInnerloop* to express the innerloop invariants:

```
predicate IsValidInnerloop(q: seq<int>, oldq: seq<int>, outerq: seq<
   int>, j: int , i: int, oldi: int)
{
   (0≤j≤i=oldi<|q|=|oldq|=|outerq|) ∧ (∀ k • j<k≤i ⟹ q[j]<q[k])
   ∧ (q[0..j]=oldq[0..j]) ∧ (q[j+1..i+1]=oldq[j..i])
   ∧ Sorted(q[..j]) ∧ Sorted(q[j+1.. i+1]) ∧ multiset(q) = multiset(
   oldq)
}
```

This predicate makes sure that the sub array between i and 0 stays sorted (except the element that moves to the correct spot in each iteration). The swaps are performed by another method:

```
method swap(a: array<int>, i:int, j:int)
  modifies a;
  requires a.Length>1 ∧ 0≤i≤j<a.Length
  ensures SwapPosts(a[..],old(a[..]),i,j)
```

```
{
  a[i], a[j] := a[j], a[i];
}
```

To ensure the swaps preserves the parts of the array that are not part of the swap, we created a lemma. this lemma helps Dafny to ensure the *IsValidInnerloop* holds:

```
lemma afterSwap(q: seq<int>, oldq: seq<int>,preSwapQ: seq<int>, j:
    int, i: int)
  requires AfterSwapPres(q,oldq,preSwapQ,j,i)
  ensures q[j..i+1]=oldq[j-1..i]
{}

predicate AfterSwapPres(q: seq<int>, oldq: seq<int>,preSwapQ: seq<
    int>,
     j: int, i: int)
{
  1<|q|=|oldq|=|preSwapQ| ∧ 0<j≤i<|q|
  ∧ oldq[0..j]=preSwapQ[..j] ∧  preSwapQ[j+1..i+1]=oldq[j..i]
  ∧ Swapped(q,preSwapQ, j-1,j) ∧ multiset(q)=multiset(oldq)=multiset
    (preSwapQ)
}
```

Now all the components for verifying insertion sort functionally are complete, and we can tie them up:

```
method insertionSort(a: array<int>) returns(ghost t: nat)
  modifies a
  ensures SortPosts(a[..],old(a[..]))
  ensures IsOn2(a.Length,t)
{
  if a.Length>0
  {
    var i := 1;
    while(i < a.Length)
      invariant IsValidouterloop(a[..],old(a[..]),i)
    {
      sortSubArr(a,a[..],i);
      i := i + 1;
    }
  }
}
```

As we can see, the algorithm is built in a modular manner, since each of its components is verified separately. The composed algorithm trusts components are verified as long as the specifications are well defined.

4. *Count the number of elementary operations*: To measure the algorithm's runtime, we add the operations counter t to the algorithm. Since the algorithm is composed of

outer algorithms, we start to measure the run time from the lowest to the highest module:

```
method swap(a: array<int>, i:int, j:int) returns(ghost t: nat)
  modifies a;
  requires a.Length>1 ∧ 0 ≤ i ≤ j < a.Length
  ensures SwapPosts(a[..],old(a[..]),i,j)
  ensures t=1
{
  a[i], a[j], t := a[j], a[i], 1;
}
```

First we count each swap as a single step. We add the count to the specification, so the next level can use it:

```
method sortSubArr(a: array<int>, olda: seq<int>, i: int)
    returns(ghost t: nat)
  modifies a
  requires 0<i<a.Length=|olda| ∧ IsValidouterloop(a[..], olda, i)
  ensures SortSubArrPosts(a[..],olda,i)
  ensures t ≤ i
{
  t := 0;
  var j := i;
  ghost var oldi := i;
  ghost var q := a[..];
  while(0<j ∧ a[j]< a[j-1])
    invariant IsValidInnerloop(a[..],q,olda,j,i,oldi)
    invariant 0 ≤ t ≤ (i-j)
    decreases j
  {
    ghost var q' := a[..];
    ghost var t2 := swap(a, j-1, j);
    afterSwap(a[..],q,q',j,i);
    j := j - 1;
    t := t + t2;
  }
}
```

Now, when Dafny calls *swap*, it does not care if *swap* is verified but only considers the value of t in its postcondition. Since *sortSubArr* loops backwards from i until it encounters a smaller element, it makes i swaps at most.

We now add t to the main algorithm's body:

```
t := 0;
if a.Length>0
{
  var i := 1;
  t := t+1;
  while(i < a.Length)
    invariant IsValidouterloop(a[..], old(a[..]), i)
```

```
  {
    ghost var t1 := sortSubArr(a,a[..],i);
    ghost var t' := t;
    i := i + 1;
    t := t + t1;
  }
}
```

Dafny still fails to verify the postcondition

```
 ensures IsOn2(n,t)
```

To proof this postcondition we must derive from *insertionSort* a function *f*, that upper-bounds the algorithm's time complexity.

5. *Deriving the function that upper-bounds the time complexity* In order to define a function that upper-bounds the algorithm, we analyze the worst-case scenario, which leads us to the else condition and requires to analyze the behavior of the algorithm's body. The insertion sort main algorithm calls *sortSubArr* for every i, which iterates from 1 to the size of the array. As a result, the outer loop makes each iteration i steps. Starting with t=1, the number of elementary steps is bounded by:

$$1 + \sum_{k=1}^{n} k = 1 + \frac{n(n+1)}{2} \tag{4.13}$$

We use a lemma and a function to define this sum,

```
function APSum(n: int) : int
  requires 0≤n
  decreases n
{
  if n=0 then 0 else n + APSum(n-1)
}

//proof of gauss sum
lemma Gauss(n: nat)
  ensures APSum(n) = n*(n+1)/2
{
  if(n≠0) {Gauss(n-1);}
}
```

and add it to the main method's body.

```
t := 0;
if a.Length>0
{
  var i := 1;
  t := t+1;
```

```
  while(i < a.Length)
    invariant IsValidouterloop(a[..], old(a[..]), i)
    invariant t ≤ 1+APSum(i)
  {
    ghost var t1 := sortSubArr(a,a[..],i);
    ghost var t' := t;
    i := i + 1;
    t := t + t1;
  }
}
Gauss(a.Length);
```

6. *Proving the function belongs to the target complexity class* Dafny still fails to verify the postcondition *IsOn2*, since it has not yet proven that $f(n) = 1 + \frac{n(n+1)}{2}$ is of $O(n^2)$. For this proof, we implement *quadraticCalcLemma* and use *On2Proof* that would be presented next. This requires us to find proper *c* and *n0* that satisfy the expression: The condition for $f(n)$ with $0 \leq n$ to belong to $\mathrm{O}(n^2)$ is:

$$\exists c > 0 \; and \; \exists n_0 > 0 \; s.t. \; \forall n. \; n \geq n_0 \Rightarrow 1 + \frac{n(n+1)}{2} \leq c * n * n \tag{4.14}$$

For implementing the body of *quadraticCalcLemma* we use Dafny's *calc*.

```
lemma quadraticCalcLemma(n: nat) returns(c: nat, n0: nat)
  ensures IsQuadraticFrom(c, n0, f)
{
  calc ⟸{
      f(n) ≤ c*n*n;
    }
    if n>0
    {
      calc ⟸{
        f(n) ≤ 1+(n*(2*n)/2) ≤ c*n*n;
        f(n) ≤ 1+(n*(2*n)/2) = 1+n*n ≤ c*n*n;
        f(n) ≤ 1+(n*(2*n)/2) = 1+n*n ≤ 2*n*n ≤ c*n*n;
        (c=2 ∧ n0=1 ∧ n0≤n) ∧(f(n) ≤ 1+(n*(2*n)/2) = 1+n*n ≤ 2*n*n ≤
    c*n*n);
      }
    }
    c, n0 := 2, 1;
}
```

This lemma works through the steps that are required to infer proper *c*, *n0* for $f(n) = 1 + \frac{n(n+1)}{2}$. Since this lemma ensures that f is quadratic, the conditions of the *On2Proof* lemma now hold.

```
lemma On2Proof(n: nat , t: nat )
  requires t≤f(n)
```

```
  ensures IsOn2(n,t)
{
  var c,n0 := quadraticCalcLemma(n);
  quadratic(c,n0,f);
}
```

Step 7. *Integrating the time complexity elements with the code* Finally, the functionally and time complexity verified version of the algorithm is complete:

```
method insertionSort(a: array<int>) returns(ghost t: nat)
  modifies a
  ensures SortPosts(a[..],old(a[..]))
  ensures IsOn2(a.Length,t)
{
  t := 0;
  if a.Length>0
  {
    var i := 1;
    t := t+1;
    while(i < a.Length)
      invariant IsValidouterloop(a[..], old(a[..]), i)
      invariant t ≤ 1+APSum(i)
    {
      ghost var t1 := sortSubArr(a,a[..],i);
      ghost var t' := t;
      i := i + 1;
      t := t + t1;
    }
  }
  Gauss(a.Length);
  On2Proof(a.Length,t);
}
```

The postcondition *IsOn2* sums up all the required terms for having $O(n^2)$ time complexity as defined in expression 4.14.

Chapter 5

Proof engineering

When writing a new program or software, the most common rookie mistake is to jump in directly, write all the necessary code into a single program, and later try to debug or extend later. This kind of approach is doomed to fail and would usually require re-writing from scratch. So in order to tackle this scenario, we can try to divide the problem into multiple sub-problems and then try to tackle it one by one. Doing so not only makes our task easier but also allows us to achieve abstraction from the high-level programmer and also promotes the reusability of code. In Dafny, this is accomplished via modules [Rea20b]. Modules may import each other for code reuse, and it is possible to abstract over modules to separate implementation from an interface. In this chapter, we use the concept of modules, abstraction, and refinement to present how the presented approach in Chapter 4 can be used to implement other verified algorithms (functionally and time complexity) without starting the methodology steps from scratch. It also creates a structure that separates the interface from the implementation of algorithms and proofs, allowing the developer to create a different implementation of the same declaration according to a specific requirement, and consume existing methods and proofs without understanding their bodies. We also use the concept of modules along our methodology to demonstrate that this methodology works on proving classic algorithms problems such as powerset size. This concept aims to project software engineering best practices of modularity and APIs usage on our proof methodology, hence the title *Proof Engineering*.

5.1 Example 1: Find Max

We start with a simple one, to implement a verified version of the find max algorithm - given a sequence, find the largest element in it. The solution is to initialize max as the first element, then traverse the given sequence from the second element till the end. For every traversed element, compare it with max, if it is greater than max, then update max. A theoretical algorithm analysis results that the time complexity is linear, as a loop runs from 0 to n-1. i.e., it runs in O(n) time.

1. *Defining big-$\mathcal{O}$ notation*: Since the definitions of O(n) exists, we wrap them in a module that would be reused for all the linear algorithms. A module body can consist of classes, datatypes, types, methods, and functions.

```
module LinearNotation
{
  predicate IsOn (n: nat ,t: nat )
  {
    ∃ f: nat -> nat • IsLinear(f) ∧ t≤f(n)
  }
  predicate IsLinear (f: nat -> nat)
  {
    ∃ c :nat, n0: nat • IsLinearFrom(c, n0, f)
  }
  predicate IsLinearFrom (c :nat, n0: nat, f: nat -> nat)
  {
    ∀ n: nat • n0 ≤ n ⟹ f(n) ≤ c * n
  }
  lemma linear (c :nat, n0: nat, f: nat -> nat)
    requires IsLinearFrom(c,n0,f)
    ensures IsLinear(f)
  {}
}
```

We use the *IsOn* predicate to ensure the algorithm is linear. This predicate would be satisfied only if we prove such linear function *f*, that upper-bounds the algorithm's time complexity exists. This proof and any linear proof is done by the *OnProof* lemma and a *linearCalcLemma*. For creating a non-specific implementation, all that is needed is a module that implements some interface. In that case, we use an abstract module, which uses the *LinearNotation* module:

```
abstract module LinearProofInterface
{
  import opened LinearNotation
```

```
  function f(n: nat) : nat

  lemma linearCalcLemma(n: nat) returns(c: nat, n0: nat)
    ensures IsLinearFrom(c,n0,f)

  lemma OnProof(n: nat , t: nat )
    requires t≤f(n)
    ensures IsOn(n,t)
  {
    var c,n0 := linearCalcLemma(n);
    linear(c,n0,f);
  }
}
```

This module imports the module *LinearNotation* as *opened*, which causes all of its members to be available. The function *f* and *linearCalcLemma* are bodyless, since their implementations are explicit to a specific algorithm.

2. *Specifying the functional and time complexity requirements for the Algorithm Declaration*: For verifying the functional requirement of the algorithm, we need to require that the input sequence has at least one cell, and ensure the return index is within the sequence and has the sequence's greatest value. For this causes we create *FindMaxPosts* a predicate for the postconditions. We also define the time complexity value with a *ghost variable t*, and create another abstract module for FindMax interface. This module is a *refinement* of *LinearProofInterface*. The *refinement* statement will allow us to implement the explicit *f* and *linearCalcLemma* for *findMax*.

```
abstract module FindMaxInterface refines LinearProofInterface
{
  predicate FindMaxPosts(q: seq<int>, max_ind: int)
  {
    0≤max_ind<|q| ∧ ∀ k • 0≤k<|q| ⟹ q[k]≤q[max_ind]
  }
  method findMax(q: seq<int>) returns (max_ind: int, ghost t: nat)
    requires |q|>0
    ensures  FindMaxPosts(q,max_ind)
    ensures IsOn(|q|,t)
}
```

Steps 3-7 are performed within the module which contains the specific implementation of *findMax*:

```
module FindMax refines FindMaxInterface
{
  function f(n: nat) : nat
```

```
  {
    n+1
  }
  lemma linearCalcLemma(n: nat) returns(c: nat, n0: nat)
  {
    if n=0
    {
      assert ∀ k: nat • f(n)≥k*n;
    }
    else
    {
      assert 1≤n;
      calc ⟸{
        f(n)≤c*n;
        f(n)≤2*n≤c*n;
        c=2 ∧ n0=1 ∧ n0≤n ∧ (f(n)≤2*n≤ c*n);
      }
    }

    c, n0 := 2, 1;
  }
  predicate FMLoop(q: seq<int>,i: int, max_ind: int)
  {
    0≤i≤|q| ∧ 0≤max_ind<|q| ∧ ∀ k • 0≤k<i ⟹ q[k]≤q[max_ind]
  }
  method findMax(q: seq<int>) returns(max_ind: int, ghost t: nat)
  {
    t := 1;
    var i := 0;
    max_ind := 0;
    while i < |q|
      invariant FMLoop(q,i,max_ind)
      invariant t = i+1
      decreases |q| - i
    {
      if q[i] ≥ q[max_ind]
        {max_ind := i;}
      i := i + 1;
      t := t + 1;
    }
    OnProof(|q|,t);
  }
}
```

This module contains the explicit implementation of *findMax* the explicit definition of *f* and the explicit implementation of *linearCalcLemma* which is adjusted to *f*. The implementation of the refined function, lemma and method is separated from the declaration. this concept can be define for the iterative Fibonacci as well.

```
module FIBHelper
{
  function Fib(n: nat): nat
  {
    if n≤1 then n else Fib(n − 1) + Fib(n − 2)
  }
}
abstract module FiboItInterface refines LinearProofInterface
{
  import opened FIBHelper
  method ComputeFib(n: nat) returns (x: nat, ghost t: nat)
     ensures x = Fib(n)
     ensures IsOn(n,t)
}
module FiboIt refines FiboItInterface
{
  function f(n: nat) : nat
  {
    n+2
  }
  lemma linearCalcLemma(n: nat) returns(c: nat, n0: nat)
  {
    if n=0
    {
      assert ∀ k: nat • f(n) ≥ k * n;
    }
    else
    {
      assert 1≤n;
      calc ⟸{
        f(n)≤c*n;
        f(n)≤2*n+1≤c*n;
        f(n)≤2*n+1≤3*n≤c*n;
        c=3 ∧ n0=1 ∧ n0≤n ∧ (f(n)≤2*n+1≤3*n≤ c*n);
      }
    }

    c, n0 := 3, 1;
  }
  predicate FibLoop(n: nat, x: nat, a: nat, i: nat)
  {
    0<i≤n ∧ a=Fib(i−1) ∧ x=Fib(i)
  }
  method ComputeFib(n: nat) returns (x: nat, ghost t: nat)
    ensures x = Fib(n)
    ensures IsOn(n,t)
  {
    if n≤1
    {
      x := n;
      t := 1;
```

```
    }
    else
    {
      var i,a:= 1,0;
      x := 1;
      t := 3;
      while i < n
        invariant FibLoop(n,x,a,i)
        invariant t=i+2
      {
        a, x := x, a+x;
        i := i+1;
        t := t+1;
      }
    }
    OnProof(n,t);
  }
}
```

The *FIBHelper* is defined separately Since it is a functional attribute used for the recursive implementation as well.

5.2 Example 2: Quick Sort

Quicksort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. This implementation takes the last the last element of the array. The key process in quicksort is the partition: given an array and an element x of the array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time. The sub-arrays are then sorted recursively. Fig 5.1 presents a running example of quicksort.

The worst case scenario occurs if the pivot happens to be the smallest element in the list. If this happens repeatedly in every partition, then each recursive call processes a list of size one less than the previous list. Consequently, we can make $n-1$ nested calls before we reach a list of size 1. This means that the call tree is a linear chain of $n-1$ nested calls. The i^{th} call does $O(n-i)$ work to do the partition, so the quick sort worst case upper-bounding runtime is:

$$T(n) = n-1+n-2+\cdots+2+1 = \frac{(n-1)(n-2)}{2} = O(n^2) \tag{5.1}$$

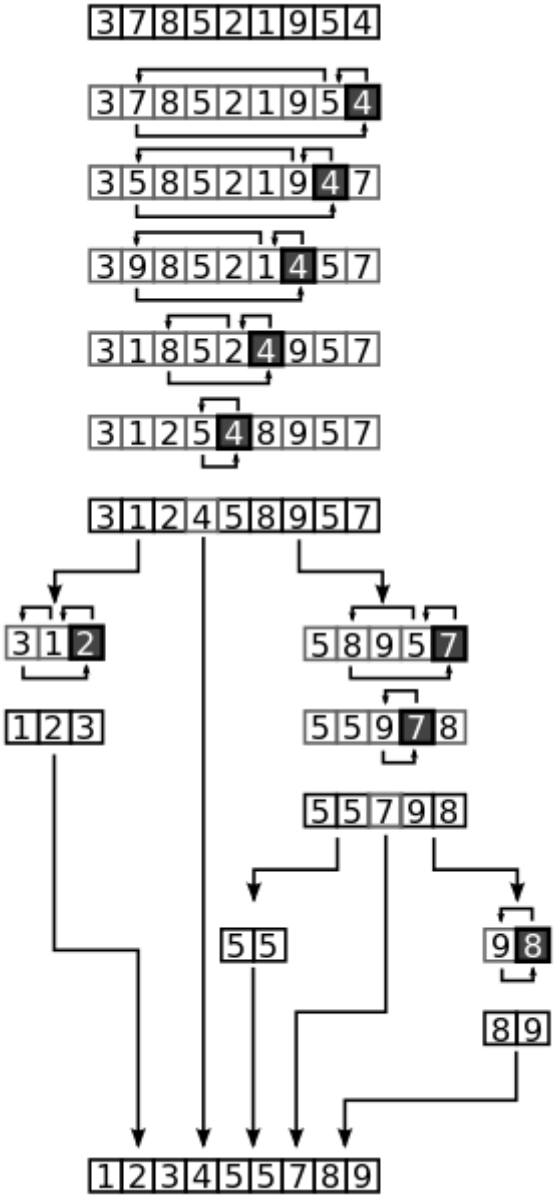

Figure 5.1: A running example of quicksort.

Since quick sort is a sorting algorithm, functional specifications and implementations that have been done for insertion sort can be reused. These *SortHelpers* should be gathered in a module, which can be *opened* int the implementation of any sorting algorithm:

```
module SortHelpers
{
  predicate Sorted(q: seq<int>)
  {
    ∀ i,j • 0 ≤ i ≤ j < |q| ⟹ q[i] ≤ q[j]
  }

  predicate SortPosts(q: seq<int>, oldq: seq<int>)
  {
    Sorted(q) ∧ multiset(q)=multiset(oldq)
  }

  predicate Swapped(q: seq<int>, oldq: seq<int>, i: int, j: int)
    requires |q|>1 ∧ 0≤i≤j<|q|=|oldq|
  {
    q = oldq[i:=oldq[j]][j:=oldq[i]]
  }

  function APSum(n: int) : int
    requires 0≤n
    decreases n
  {
    if n=0 then 0 else n + APSum(n-1)
  }

  lemma Gauss(n: nat)
    ensures APSum(n) = n*(n+1)/2
  {
    if(n≠0) {Gauss(n-1);}
  }

  method swap(a: array<int>, i:int, j:int) returns(ghost t: nat)
    modifies a;
    requires a.Length>1 ∧ 0 ≤ i ≤ j < a.Length
    ensures SwapPosts(a[..],old(a[..]),i,j)
    ensures t=1
  {
    a[i], a[j], t := a[j], a[i], 1;
  }

  predicate SwapPosts(q: seq<int>, oldq: seq<int>, i:int, j:int)
  {
    |q|>1 ∧ 0≤i≤j<|q|=|oldq| ∧ Swapped(q,oldq,i,j) ∧ multiset(q)=
   multiset(oldq) ∧ multiset(q[i..])=multiset(oldq[i..])
  }
```

```
}
```

Next, we elaborate on the implementation of quicksort with the help of our methodology and modulization:

1. *Defining big-$\mathcal{O}$ notation*: Since the definitions of $O(n^2)$ exist, we wrap them in a module that would be reused for all the quadratic algorithms.

```
module QuadraticNotation
{
  predicate IsOn2 (n: nat ,t: nat )
  {
    ∃ f: nat -> nat • IsQuadratic(f) ∧ t≤f(n)
  }
  predicate IsQuadraticFrom (c :nat, n0: nat, f: nat -> nat)
  {
    ∀ n: nat • 0 < n0 ≤ n ⟹ f(n) ≤ c*n*n
  }
  predicate IsQuadratic(f: nat -> nat)
  {
    ∃ c :nat, n0: nat • IsQuadraticFrom(c, n0, f)
  }
  lemma quadratic (c :nat, n0: nat, f: nat -> nat)
    requires IsQuadraticFrom(c,n0,f)
    ensures IsQuadratic(f)
  {}
}
```

For creating the non-specific function and lemmas for quadratic proofs, we use an abstract module, which uses the *QuadraticNotation* module:

```
abstract module QuadraticProofInterface
{
  import opened QuadraticNotation
  function f(n: nat) : nat

  lemma quadraticCalcLemma(n: nat) returns(c: nat, n0: nat)
    ensures IsQuadraticFrom(c, n0, f)

  lemma On2Proof(n: nat , t: nat )
    requires t≤f(n)
    ensures IsOn2(n,t)
  {
    var c,n0 := quadraticCalcLemma(n);
    quadratic(c,n0,f);
  }
}
```

2. *Specifying the functional and time complexity requirements for the Algorithm Declaration*: We define a module for the quick sort interface, which is similar to the specification

that was made for insertion sort:

```
abstract module InsertionSortInterface refines
    QuadraticProofInterface
{
  import opened SortHelpers
  method insertionSort(a: array<int>) returns(ghost t: nat)
    modifies a
    ensures SortPosts(a[..],old(a[..]))
      ensures IsOn2(a.Length,t)
}
```

The next steps of the methodology are implemented as part of

```
module QuickSort refines QuickSortInterface
{...}
and elaborated separately.
```

3. *Verifying the correctness of the algorithm*: For the recursive implementation of quick sort, we use *ComputeFibRec* as the *quickSortRec* body:

```
predicate QuickSortRecPosts(q: seq<int>, oldq: seq<int>,
    from: int, to: int)
{
  |q|=|oldq|>1 ∧ 0≤from≤to≤|q| ∧ multiset(q)=multiset(oldq) ∧
   SubMultisets(q,oldq,from,to) ∧ Sorted(q[from..to])
}

method quickSortRec(a: array<int>, from: int, to: int)
  modifies a
  requires a.Length>1 ∧ 0≤from≤to≤a.Length
  ensures QuickSortRecPosts(a[..],old(a[..]),from,to)
  decreases to − from
{
  ghost var q := a[..];
  if from<to
  {
    var r;
    r := partition(a,from,to);
    ghost var q' := a[..];
    quickSortRec(a,from,r);
    prove1 (a[..],q',q,from,to,r);
    ghost var q'' := a[..];
    quickSortRec(a,r+1,to);
    prove2(a[..],q'', q',q,from,to,r);
    subSorted(a[..], from, to, r);
  }
}
```

The partition part is made by the *partition* algorithm:

```
method partition(a: array<int>, from: int, to: int) returns (r: int)
```

```
  modifies a
  requires a.Length>1 ∧ 0≤from<to≤a.Length
  ensures Partitioned(a[..],old(a[..]),from,to,r)
{
  var le := from;
  r := to-1;
  var piv := a[r];
  while le<r
    invariant PartitionLoop(a[..],old(a[..]),from,to,r,le,piv)
    decreases r-le
  {
    if a[le]≤piv
    {
      le := le+1;
    }
    else
    {
      swap2(a,le,r-1);
      ghost var q' := a[..];
      swap2(a,r-1,r);
      bigSwap(a[..],q',to,r);
      r := r-1;
    }
  }
}
```

This method's specification ensures that the partition is made correctly, as describes at the beginning of this section. The returned value is the new index of the pivot, which is the pivot's location in a sorted array:

```
predicate Partitioned(q: seq<int>, oldq: seq<int>, from: int, to:
   int, r: int)
{
  |oldq|=|q|>1 ∧ multiset(q) = multiset(oldq)
  ∧ 0≤from≤r<to≤|q|
    ∧ SeqSmallerThanNum(q[from..r+1],q[r])
    ∧ SeqBiggerThanNum(q[r..to],q[r])
    ∧ SubMultisets(q, oldq, from, to)
}
```

The first line ensures the array before and after the partition has more than one element and that the number of copies of each element is the same. The second line ensures the boundaries of the partition and the returned index are within the array. *SeqSmallerThanNum* and *SeqBiggerThanNum* ensure that all the elements left to the pivot's new location are smaller than the pivot and all the elements right to the pivot's new location are larger than the pivot respectively:

```
predicate SeqSmallerThanNum(q: seq<int>, num: int)
{
  ∀ x • x in q ⟹ x ≤ num
}

predicate SeqBiggerThanNum(q: seq<int>, num: int)
{
  ∀ x • x in q ⟹ x ≥ num
}
```

SubMultisets ensures the array has changed order only where the elements are between from and to (to not included):

```
predicate SubMultisets(q: seq<int>, oldq: seq<int>, from: int, to:
    int)
  requires |q|>1 ∧ 0 ≤ from ≤ to ≤ |q|=|oldq|
  requires multiset(q) = multiset(oldq)
{
  q=oldq[..from]+q[from..to]+oldq[to..] ∧ q[..from]=oldq[..from] ∧
    multiset(q[from..to])=multiset(oldq[from..to]) ∧ q[to..]=oldq[to
    ..]
}
```

The *partition* method body loops (with *le*) from the first element (*from*) to the last(*to*). If the iterated value is bigger than the pivot we swap it with the pivot and than with the one before the pivot's old location for retaining the order.

```
predicate PartitionLoop(q: seq<int>, oldq: seq<int>, from: int, to:
    int, r: int, le: int, piv: int)
  requires |q|>1 ∧ 0 ≤ from < to ≤ |q|=|oldq|
{
  0≤from≤le≤r<to≤|q| ∧ multiset(q) = multiset(oldq)
   ∧ q[r]=piv ∧ SeqSmallerThanNum(q[from..le],q[r])
  ∧ SeqBiggerThanNum(q[r..to],q[r])
  ∧ SubMultisets(q, oldq, from, to)
}
```

The loop invariants ensure that the indices remain legit, the number of copies of the before and after each iteration are unchanged, and that the returned index r is the pivot's location. The invariants make sure that all elements that have been handled (left to *le*) are smaller than the pivot, all elements right to *r* are bigger, and the array changed order only where the elements are between from and to.

The swaps of the else case, where the iterated value is bigger than the pivot, are done by the swap method imported from the *SortHelpers* module and has been used by the

insertion sort method. The outcome of the swaps which we explained in the previous paragraph, is proven by the lemmas *bigSwap*:

```
predicate bigSwapPres(q: seq<int>, oldq: seq<int>, to: int, r: int)
{
  |oldq|=|q|>1 ∧ 0<r<to≤|q|
  ∧ Swapped(q,oldq,r−1,r) ∧ SeqBiggerThanNum(oldq[r..to],oldq[r])
  ∧ oldq[r−1]≥oldq[r] ∧ multiset(q[r−1..])=multiset(oldq[r−1..])
}
lemma bigSwap(q: seq<int>, oldq: seq<int>, to: int, r: int)
  requires bigSwapPres(q,oldq,to,r)
  ensures SeqBiggerThanNum(q[r−1..to],q[r−1])
{
  assert oldq[r−1..r] + oldq[r..to] = oldq[r−1..to];
  assert SeqBiggerThanNum(oldq[r−1..to],oldq[r]);
  multiBig(q[r−1..to],oldq[r−1..to],oldq[r]);
  assert SeqBiggerThanNum(q[r−1..to],oldq[r]);
}
lemma multiBig(q: seq<int>, oldq: seq<int>, num: int)
  requires |q|≥0 ∧ |q|=|oldq|
     ∧ multiset(q)=multiset(oldq)
     ∧ SeqBiggerThanNum(oldq,num)
  ensures SeqBiggerThanNum(q,num)
{
  assert ∀ x • x in q ⟹ x in multiset(q);
  assert ∀ x • x in multiset(q) ⟹ x in multiset(oldq);
  assert ∀ x • x in multiset(oldq) ⟹ x in oldq;
}
```

multiBig proves that if 2 multisets are equal, they are bigger from the same number.

Now after the partition is verified functionally, we move back to the *quickSortRec* body:

```
ghost var q := a[..];
if from<to
{
  var r;
  r := partition2(a,from,to);
  ghost var q' := a[..];
  quickSortRec(a,from,r);
  prove1 (a[..],q',q,from,to,r);
  ghost var q'' := a[..];
  quickSortRec(a,r+1,to);
  prove2(a[..],q'', q',q,from,to,r);
  subSorted(a[..], from, to, r);
}
```

We define several ghost variable for storing different stated of the array. q is for the 0 state, q' is the state after the partition , and q" is the state after the recursive call of the

method has been made for the part of the arrays left to the pivot. Theses states are used in the lemmas *prove1* and *prove2*:

```
predicate Prove1Pres(q'': seq<int>, q': seq<int>, q: seq<int>, from:
    int, to: int, r: int)
{
  1<|q''|=|q'|=|q| ∧ 0≤from≤r<to≤|q|
  ∧ multiset(q'')=multiset(q')=multiset(q)
  ∧ Partitioned(q',q,from,to,r)
  ∧ SubMultisets(q'',q', from,r)
  ∧ Sorted(q''[from..r])
}
predicate Prove1Posts(q'': seq<int>, q': seq<int>, q: seq<int>, from
   : int, to: int, r: int)
{
  Prove1Pres(q'',q',q, from,to,r) ∧ q''[r]=q'[r] ∧ Partitioned(q'',q
   ,from,to,r)
}
lemma prove1 (q'': seq<int>, q': seq<int>,q: seq<int>, from: int, to
   : int, r: int)
  requires Prove1Pres(q'',q',q, from,to,r)
  ensures Prove1Posts(q'',q',q, from,to,r)
{
  sameMultiRight(q'',q',from,r);
  multiSmall(q''[from..r+1], q'[from..r+1], q''[r]);
  assert q''[r..to] = q'[r..to];
  subMultiset(q'',q', from,to);
}
```

```
predicate Prove2Pres(q''': seq<int>, q'': seq<int>, q': seq<int>, q:
    seq<int>, from: int, to: int, r: int)
{
  1<|q''|=|q'|=|q|=|q'''| ∧ 0≤from≤r< to ≤ |q|
  ∧ multiset(q''')=multiset(q'')=multiset(q')=multiset(q)
  ∧ Partitioned(q'',q,from,to,r) ∧ SubMultisets(q''',q'',r+1,to)
  ∧ Sorted(q'''[r+1..to]) ∧ Sorted(q''[from..r])
  ∧ q''[r]=q'[r]
}
predicate Prove2Posts(q''': seq<int>, q'': seq<int>, q': seq<int>, q
   : seq<int>, from: int, to: int, r: int)
{
  Prove2Pres(q''',q'',q',q, from,to,r) ∧ q'''[r]=q''[r]=q'[r] ∧
   Partitioned(q''',q,from,to,r) ∧ Sorted(q'''[from..r])
}
lemma prove2 (q''': seq<int>,q'': seq<int>, q': seq<int>,q: seq<int
   >, from: int, to: int, r: int)
  requires Prove2Pres(q''',q'',q',q, from,to,r)
  ensures Prove2Posts(q''',q'',q',q, from,to,r)
{
  sameMultiLeft(q''',q'',to,r);
  multiBig(q'''[r..to], q''[r..to], q'''[r]);
```

```
  subMultiset(q'''',q'', from,to);
  assert q''''[from..r]=q''[from..r];
}
```

The lemma *prove1* proves that the first recursion call returns the left part of the array sorted, without touching the right part or the pivot. it is aided with the lemmas *sameMultiRight* that ensures the right of the array did not changed, *multiSmall* that ensures that if the left part of the arrys was smaler than the pivot before the call its is still after (equal multisets), and *subMultiset* that ensure that parts of the array outside the sort limits remain the same:

```
lemma sameMultiRight(q'': seq<int>,q': seq<int>, from: int,
    r: int)
  requires |q''|>1 ∧ 0≤from≤r<|q''|=|q'| ∧ multiset(q''[from..r])=
   multiset(q'[from..r]) ∧ q''[r]=q'[r]
  ensures multiset(q''[from..r+1]) = multiset(q'[from..r+1])
{
  assert q''[from..r]+[q''[r]] = q''[from..r+1];
  assert q'[from..r]+[q'[r]] = q'[from..r+1];
  assert multiset(q''[from..r]+[q''[r]]) = multiset(q''[from..r+1]);
  assert multiset(q'[from..r]+[q'[r]]) = multiset(q'[from..r+1]);
}

lemma multiSmall(q: seq<int>, oldq: seq<int>, num: int)
  requires |q|≥0 ∧ |q|=|oldq| ∧ multiset(q)=multiset(oldq) ∧
   SeqSmallerThanNum(oldq,num)
  ensures SeqSmallerThanNum(q,num)
{
  assert ∀ x • x in q ⟹ x in multiset(q);
  assert ∀ x • x in multiset(q) ⟹ x in multiset(oldq);
  assert ∀ x • x in multiset(oldq) ⟹ x in oldq;
}

lemma subMultiset(q: seq<int>, oldq: seq<int>, from: int, to: int)
  requires |q|>1 ∧ 0≤from≤to≤|q|=|oldq| ∧ multiset(q)=multiset(oldq)
    ∧ q=oldq[..from]+q[from..to]+oldq[to..]
  ensures SubMultisets(q,oldq,from,to)
  ensures to<|q|=|oldq| ⟹ (q[to]=oldq[to] ∧ q[to+1..]=oldq[to
   +1..])
  ensures from>0 ⟹ (q[from−1]=oldq[from−1] ∧ q[..from−1]=oldq[..
   from−1])

{
  assert multiset(q)−multiset(oldq[..from])−multiset(oldq[to..]) =
   multiset(q[from..to]);
  assert oldq=(oldq[..from]+oldq[from..to]+oldq[to..]);
  assert multiset(oldq)−multiset(oldq[..from])−multiset(oldq[to..])
   = multiset(oldq[from..to]);
```

```
  assert  multiset(q) = multiset(oldq[..from])+multiset(oldq[from..
   to])+multiset(oldq[to..]);
  assert q[..from] = oldq[..from];
  assert q[to..] = oldq[to..];
}
```

The same concept of proof is done by prove2 to the right and bigger part of the array with *sameMultiLeft*:

```
lemma sameMultiLeft(q'': seq<int>,q': seq<int>, to: int, r: int)
  requires |q''|>1 ∧ 0≤r<to≤|q''|=|q'| ∧ multiset(q''[r+1..to])=
   multiset(q'[r+1..to]) ∧ q''[r]=q'[r]
  ensures multiset(q''[r..to]) = multiset(q'[r..to])
{
  assert [q''[r]]+q''[r+1..to] =  q''[r..to];
  assert [q'[r]]+q'[r+1..to] =  q'[r..to];
  assert multiset([q''[r]] + q''[r+1..to]) = multiset(q''[r..to]);
  assert multiset([q'[r]]  + q'[r+1..to]) = multiset(q'[r..to]);
}
```

Also *multiBig* and *subMultiset* which was elaborated earlier.

Now all the components for verifying quick sort *functionally* are complete, and we can tie them up:

```
method quickSort(a: array<int>) returns(ghost t: nat)
  modifies a
  ensures SortPosts(a[..],old(a[..]))
  ensures IsOn2(a.Length,t)
{
  if a.Length > 1
  {
    quickSortRec(a,0,a.Length);
  }
}
method quickSortRec(a: array<int>, from: int, to: int)
  modifies a
  requires a.Length>1 ∧ 0≤from≤to≤a.Length
  ensures QuickSortRecPosts(a[..],old(a[..]),from,to)
  decreases to - from
{
  ghost var q := a[..];
  if from<to
  {
    var r;
    r := partition2(a,from,to);
    ghost var q' := a[..];
    quickSortRec(a,from,r);
    prove1 (a[..],q',q,from,to,r);
    ghost var q'' := a[..];
    quickSortRec(a,r+1,to);
```

```
    prove2(a[..],q'', q',q,from,to,r);
    subSorted(a[..], from, to, r);
  }
}
```

As we can see, the algorithm is built in a modular manner, since each of its components is verified separately. The composed algorithm trusts components are verified as long as the specifications are well defined.

4. *Count the number of elementary operations*: We now add the operations counter t to the algorithms that compose *quicksort*. First we count each swap as a single step. It is given to us effortlessly in this algorithm since it was already implemented and declared in the *SortHelpers* module. The next level uses this count to its own:

```
method partition(a: array<int>, from: int, to: int) returns (r: int,
    ghost t: nat)
  modifies a
  requires a.Length>1 ∧ 0 ≤ from < to ≤ a.Length
  ensures Partitioned(a[..],old(a[..]),from,to,r)
  ensures t ≤ 3*(to-from)
{
  var le := from;
  r := to-1;
  var piv := a[r];
  t := 1;
  while le<r
    invariant PartitionLoop(a[..],old(a[..]),from,to,r,le,piv)
    invariant t ≤ 1 + (2*((to-from)-(r-le)))
    decreases r-le
  {
    if a[le]≤piv
    {
      le := le+1;
      t := t+1;
    }
    else
    {
      ghost var t1 := swap(a,le,r-1);
      ghost var q' := a[..];
      ghost var t2 := swap(a,r-1,r);
      bigSwap(a[..],q',to,r);
      r := r-1;
      t := t+t1+t2;
    }
  }
}
```

Now, when Dafny calls *swap*, it does not care if *swap* is verified but only considers the

value of t in its postcondition. If we assign $n = to - from$, then the *partition* loops backwards from n-1 (the distance between r and le decreases) to 0. starting with t=1 before the loop and counting 2 elementary operations in each iteration, there are at most 1+2*n elementary operations in partition.

We now add t to *quickSortRec* body:

```
method quickSortRec(a: array<int>, from: int, to: int) returns(ghost
    t: nat)
  modifies a
  requires a.Length>1 ∧ 0 ≤ from ≤ to ≤ a.Length
  ensures QuickSortRecPosts(a[..],old(a[..]),from,to)
  decreases to − from
{
  ghost var q := a[..];
  t := 0;
  if from<to
  {
    ghost var n := to−from;
    var r;
    ghost var t';
    r, t' := partition(a,from,to);
    ghost var n1 := r−from;
    ghost var n2 := to−(r+1);
    ghost var q' := a[..];
    ghost var t1 := quickSortRec(a,from,r);
    prove1 (a[..],q',q,from,to,r);
    ghost var q'' := a[..];
    ghost var t2 := quickSortRec(a,r+1,to);
    prove2 (a[..],q'', q',q,from,to,r);
    subSorted(a[..], from, to, r);
    t := t' + t1 + t2;
  }
}
```

Dafny naturally fails to verify the postcondition

```
 ensures IsOn2(n,t)
```

To proof this postcondition we must derive from *quicksort* a function *f*, that upper-bounds the algorithm's time complexity.

5. *Deriving the function that upper-bounds the time complexity* A quick analysis on the method results that the number of elementary operations in *quickSortRec* is the sum of the number of elementary operations in partition *(t'=1+2*n)* and the number of elementary operations in *quickSortRec* of each part of the array between from and to (size n) minus the pivot in location r. Therefore, the size of the inputs to both recursive

calls is n-1. As we did superficially at the beginning of this section, the *qtime* lemma proves thoroughly; the recursion tree results a sum of natural numbers to n-1, and that if n¿1 and n-1=n1+n2, the sum of natural numbers to plus the sum of natural numbers to n2 is at most the sum of natural numbers to n-1. It is multiplied by 3 because the cost of the partition.

```
lemma qtime(n: nat, n1: nat, n2: nat)
  requires n≥1 ∧ n1+n2 = n−1
  ensures 3∗APSum(n1)+ 3∗APSum(n2) ≤ 3∗APSum(n−1)
{
  calc {
    APSum(n−1);
    ={Gauss(n−1);}
    ((n−1)∗n)/2;
    =
    ((n−1)∗(n−1)+(n−1))/2;
    ={Gauss(n1);Gauss(n2);}
    ((n1+n2)∗(n1+n2)+(n1+n2))/2;
    =
    ((n1∗n1 + 2∗n1∗n2 + n2∗n2)+(n1+n2))/2;
    ≥
    ((n1∗n1 + n2∗n2)+(n1+n2))/2;
    ≥
    (n1∗(n1+1))/2 + (n2∗(n2+1)/2);
    ≥{Gauss(n1); Gauss(n2);}
    APSum(n1)+ APSum(n2);
  }
  assert APSum(n1)+ APSum(n2) ≤ APSum(n−1);
  assert 3∗APSum(n1)+ 3∗APSum(n2) ≤ 3∗APSum(n−1);
}
```

Now we can declare that the number of elementary steps in *quickSortRec* is bounded by:

$$3 * \sum_{k=1}^{n} k = 3 * \frac{n(n+1)}{2} \tag{5.2}$$

We add the lemma *Gauss* and a function APSum which have been used with *insersionSort* and imported from the *SortHelpers* module to define this sum. Now *quickSortRec* is complete with a bounding function:

```
method quickSortRec(a: array<int>, from: int, to: int) returns(ghost
    t: nat)
  modifies a
  requires a.Length>1 ∧ 0 ≤ from ≤ to ≤ a.Length
  ensures QuickSortRecPosts(a[..],old(a[..]),from,to)
  ensures t ≤ 3∗APSum(to−from)
  decreases to − from
```

```
{
  ghost var q := a[..];
  t := 0;
  if from<to
  {
    ghost var n := to-from;
    var r;
    ghost var t';
    r, t' := partition(a,from,to);
    ghost var n1 := r-from;
    ghost var n2 := to-(r+1);
    ghost var q' := a[..];
    ghost var t1 := quickSortRec(a,from,r);
    prove1 (a[..],q',q,from,to,r);
    ghost var q'' := a[..];
    ghost var t2 := quickSortRec(a,r+1,to);
    prove2 (a[..],q'', q',q,from,to,r);
    subSorted(a[..], from, to, r);
    t := t' + t1 + t2;
    qtime(n,n1,n2);
    Gauss(n);
  }
}
```

And if we add it to the main method:

```
method quickSort(a: array<int>) returns(ghost t: nat)
  modifies a
  ensures SortPosts(a[..],old(a[..]))
  ensures IsOn2(a.Length,t)
{
  t := 1;
  if a.Length > 1
  {
    ghost var t1 := quickSortRec(a,0,a.Length);
    t := t+t1;
    Gauss(a.Length);
  }
  On2Proof(a.Length,t);
}
```

Now the upper bounding function of quick sort is $f(n) = 1 + 3 * \frac{n(n+1)}{2}$

6. *Proving the function belongs to the target complexity class* Dafny still fails to verify the postcondition *IsOn2*, since it has not yet proven that $f(n) = 1+3*\frac{n(n+1)}{2}$ is of $O(n^2)$. For this proof, we implement *quadraticCalcLemma* explicitly to this definition of f:

```
function f(n: nat) : nat
{
  1+3*(n*(n+1)/2)
}
```

```
lemma quadraticCalcLemma(n: nat) returns(c: nat, n0: nat)
  ensures IsQuadraticFrom(c, n0, f)
{
  calc ⟸{
      f(n) ≤ c*n*n;
    }
    if n>0
    {
      calc ⟸{
        f(n) ≤ 1+3*(n*(2*n)/2) ≤ c*n*n;
        f(n) ≤ 1+3*(n*(2*n)/2) = 1+3*n*n ≤ c*n*n;
        f(n) ≤ 1+3*(n*(2*n)/2) = 1+3*n*n ≤ 4*n*n ≤ c*n*n;
        (c=4 ∧ n0=1 ∧ n0≤n) ∧(f(n) ≤ 1+3*(n*(2*n)/2) = 1+3*n*n ≤ 4*n
   *n ≤ c*n*n);
      }
    }
    c, n0 := 4, 1;
}
```

This lemma works through the steps that are required to infer proper *c*, *n0* for $f(n) = 1+3*\frac{n(n+1)}{2}$ Since this lemma ensures that f is quadratic, the conditions of the *On2Proof* lemma now hold.

Step 7. *Integrating the time complexity elements with the code* Finally, the functionally and time complexity verified version of the algorithm is complete. As in the first example, *quickSort* and *insertionSort* are implemented as a *refinement* of *QuadraticProofInterface*. That allows us to implement the explicit f and *quadraticCalcLemma* for each algorithm.

5.3 Example 3: Powerset Size

The powerset is a set which includes all the subsets of a set. All subsets include the empty set and the original set itself. If set A = x,y,z is a set, then all its subsets , x, y, z, x,y, y,z, x,z and x,y,z are the elements of the powerset of, such as:

$$P(A) = \{\{\}, \{x\}, \{y\}, \{z\}, \{x, y\}, \{y, z\}, \{x, z\}, \{x, y, z\}\} \tag{5.3}$$

where P(A) denotes the powerset. We implement an iterative algorithm for generating the powerset of the set s of size n. The implementation consists of outer and inner loops. The outer loop iterate trough the set's elements; for each element *elem*, the inner loop iterates through the so far generated subsets in the returned powerset. For each subset

subs, we create a new subset which is the union between {*elem*} and subs. A superficial analysis results that every iteration of the outer loop doubles the subsets' number in the powerset. Since the number of iteration is the number of elements in the original group, the number of subsets in the powerset is 2^n. This analysis leads us to the conclusion that the size of the powerset is 2^n.

Next, we elaborate on the implementation of powerset with the help of our methodology and modulization.

1. *Defining big-$\mathcal{O}$ notation*: Since the definitions of O(2^n) exist, we wrap them in a module that would be reused for all the exponential algorithms. Since we must use the *Power* function to define them, we create two modules. A module for *Power* and power oriented rules and a module for the big-$\mathcal{O}$ notation.

```
module PowerRules
{
  function Power(n: nat, c: nat): nat
    decreases c
  {
    if c=0 then 1 else if n=0 then 0 else n * Power(n,c-1)
  }

  lemma mul_inequality(x: nat, y: nat)
    requires x ≤ y;
    ensures  ∀ z: nat • x*z ≤ y*z;
  {}
  lemma powerMono (x: nat ,y: nat )
    requires y≥x>0
    ensures ∀ n: nat • Power(n,y)≥Power(n,x)
    decreases x,y
  {
    if (x≠1 ∧ y≠1) { powerMono(x-1,y-1); }
  }
  function method PowerCalc(n: nat, c: nat): nat
    requires n>0
    ensures PowerCalc(n,c) = Power(n,c)
    ensures PowerCalc(n,c) ≥ 1
    decreases c
  {if (c=0) then 1 else n * PowerCalc(n,c-1)}
}

module ExpNotation
{
  import opened PowerRules
  predicate IsOExpn (n: nat ,t: nat )
  {
```

```
    ∃ f: nat -> nat • IsExpo(f) ∧ t≤f(n)
  }
  predicate IsExpo (f: nat -> nat)
  {
    ∃ c :nat, n0: nat • IsExpoFrom(c,n0,f)
  }
  predicate IsExpoFrom (c :nat, n0: nat, f: nat -> nat)
  {
    ∀ n: nat • 0 < n0 ≤ n ⟹ f(n) ≤ Power(2,Power(n,c))
  }
  lemma exponential (c :nat, n0: nat, f: nat -> nat)
    requires IsExpoFrom(c,n0,f)
    ensures IsExpo(f)
  {}

}
```

For creating the non-specific function and lemmas for exponential proofs, we use an abstract module, which uses the *ExpNotation* and *PowerRules* modules:

```
abstract module ExpProofInterface
{
  import opened ExpNotation
  import opened PowerRules
  function f(n: nat) : nat
  lemma expCalcLemma(n: nat)returns(c :nat, n0:nat)
    ensures IsExpoFrom(c, n0, f)
  lemma OExpnProof(n: nat , t: nat )
    requires t≤f(n)
    ensures IsOExpn(n,t)
  {
    var c,n0 := expCalcLemma(n);
    exponential(c,n0,f);
  }
}
```

2. *Specifying the functional and returned value size requirements for the Algorithm Declaration*: We define a module for the powerset size interface:

```
abstract module PowerSetSizeInterface refines
    ExpProofInterface
{
  method powersetSize(s: set<int>)
      returns (powers: set<set<int⟩)
    ensures IsPowerSet(s,powers)
    ensures IsOExpn(|s|,|powers|)
}
```

For the definition, creation, and verification of powerset, we need to specify several terms. These terms composed to a module:

```
module PowerSetHelpers
{
  function method PowerSetFunc<T>(s: set<T>) : set<set<T⟩
    ensures ∀ subs: set<T> • subs in PowerSetFunc(s) ⟹ subs≤s
    ensures ∀ subs: set<T> • subs≤s ⟹ subs in PowerSetFunc(s)
  {
    idLemma(s);
    set subset: set<T> | subset ≤ s ∧ subset=Id(subset)
  }

  lemma idLemma<T>(s: set<T>)
    ensures ∀ subs: set<T> • subs ≤ s ⟹ subs=Id(subs)
  {}

  function method Id<T>(x: T): T
  {
    x
  }
  predicate IsPowerSet<T>(s: set<T>, powers: set<set<T⟩)
  {
    PowerSetFunc(s)= powers
  }

  lemma onlyOne<T>(emp :set<T>)
    requires emp={} ∧ emp in PowerSetFunc(emp) ∧ ¬∃ elem • NotEmp(
   elem) ∧ elem≤emp
    ensures PowerSetFunc(emp) = {{}};
  {
    assert ∀ elem • NotEmp(elem) ⟹ elem ∉ PowerSetFunc(emp);
  }

  predicate NotEmp<T>(x: set<T>)
  {
    x≠{}
  }
}
```

This implementation uses Dafny's *sets* [Rea20d]. The function method *PowerSetFunc* generates a powerset according to the powerset definition. It uses *IDLemma* an *ID* function for Dafny's triggers to the *forall* and *exist* expressions. The *IsPowerSet* function is for verifying the algorithm's postcondition. The *onlyOne* lemma proves that a powerset of the empty set is a set with one element - the empty set and the *NotEmp* is for verifying a nonempty set.

Next we present *steps 3-5* which are implemented in the module :

```
module PSSize refines PowerSetSize
{...}
```

The method's body is:

```
var s_iter := s;
powers := {{}};
ghost var s_diff := {};
onlyOne(s_diff);
while (s_iter≠{})
  invariant PSOuter(s,s_diff,s_iter,powers)
  invariant |powers| = Power(2,|s_diff|)
  decreases s_iter
{
  var elem :| elem in s_iter;
  var powers_in := {};
  var powers_iter := powers;
  ghost var powers_diff := {};
  while (powers_iter ≠ {})
    invariant PSInner(powers,powers_diff,powers_iter,powers_in,
    s_diff,elem)
    decreases powers_iter
  {
    var subs :| subs in powers_iter;
    powers_diff := powers_diff + {subs};
    powers_in := powers_in + {subs + {elem}};
    powers_iter := powers_iter - {subs};
  }
  ghost var s_diff' := s_diff;
  s_diff := s_diff + {elem};
  powers := powers + powers_in;
  psLemma1(s_diff',powers_diff,s_diff,PowerSetFunc(s_diff),powers,
   elem);
  s_iter := s_iter - {elem};
}
OExpnProof(|s|,|powers|);
```

Since we use sets, the outer loop iteration is implemented with *s_iter* - a copy of *s* and *s_diff* - an empty set. Each iteration we choose an element from *s_iter* using the :— operator, which implicates for 'such as'. We remove the element from *s_iter*, and add it to *s_diff*. The variable *s_diff* is used to store the elements we already handled, for measuring the size of the powerset *powers* so far. the outer loop terminates when *s_iter* is empty. The inner loop is implemented with the same concept using the variables *powers_iter* a copy of *powers*, and *powers_diff* an empty set. Each inner loop iteration create a new subset adding the outerloop chosen element to the inner loop chosen subset. the new set stored in *powers_in*, which is united to powers in the end of the outer iteration.

The inner loop invariant ensures that *PSInner*

```
predicate PSInner<T>(powers: set<set<T>>,powers_diff: set<set<T>>,
```

```
    powers_iter: set<set<T>>, powers_in: set<set<T>>, s_diff: set<T>,
    elem: T)
{
  {}≤powers_iter≤powers ∧ {}≤powers_diff≤powers
  ∧ powers_diff=powers−powers_iter
  ∧ (∀ subset • subset in powers_diff ⟹ subset+{elem} in
    powers_in)
  ∧ (∀ subset • subset in powers_in ⟹ ¬∃ el • el in subset ∧ el
    ∉ s_diff+{elem})
  ∧ (∀ subset • subset in powers_iter ⟹ subset+{elem} ∉ powers_in
    )
  ∧ |powers_in|=|powers_diff| ∧ powers⁄powers_in
}
```

ensures that in each iteration *powers_iter*, *powers_diff* and *powers_in* contain the subsets as required in the previous paragraph, and that the sizes of the size of the new subsets stored in *powers_in* is equal to the size of the size of the handled subsets in *powers_diff*. Hence, in the end of the inner loop *powers* and *powers_diff* are the same set and *powers* is equal by size to *powers* but have no elements in common. This indicated that when the operation powers := powers + powers_in is performed, *powers* size equals twice of it's old size, therefore the derivation of the bounding function of the size of powerset results $2^{|s\_diff|}$. The lemmas *psLemma1* and *psLemma2*

```
predicate psLemma2Pres<T>(xPowers: set<set<T>>, yPowers: set<set<T>>,
    powers: set<set<T>>, elem: T)
{
  xPowers≤powers ∧ xPowers≤yPowers
  ∧ (∀ subs • (subs in yPowers ∧ elem ∉ subs) ⟹ subs in powers)
  ∧ (∀ subs • (subs in yPowers ∧ elem in subs) ⟹ (subs−{elem}) in
    xPowers)
  ∧ (∀ subs • subs in xPowers ⟹ (subs in powers ∧ subs+{elem} in
    powers))
}

lemma psLemma2<T>(xPowers: set<set<T>>, yPowers: set<set<T>>, powers:
    set<set<T>>, elem: T)
  requires psLemma2Pres(xPowers,yPowers,powers,elem)
  ensures yPowers≤powers
{
  assert ∀ subs • (subs in yPowers ∧ elem in subs) ⟹ ((subs−{elem
    })+{elem} in powers);
  assert ∀ subs • (subs in yPowers ∧ elem in subs) ⟹ (subs−{elem
    })+{elem}=subs;
}

predicate psLemma1Pres<T>(x: set<T>,  xPowers: set<set<T>>, y: set<T
```

```
    >, yPowers: set<set<T>>, powers: set<set<T>>, elem: T)
{
  IsPowerSet(x,xPowers) ∧ y=x+{elem} ∧ yPowers=PowerSetFunc(y)
  ∧ (∀ subs • subs in xPowers ⟹ (subs in powers ∧ subs+{elem} in
    powers))
  ∧ (∀ subs • subs in powers ⟹ ¬∃ el • el in subs ∧ el ∉ y)
}

lemma psLemma1<T>(x: set<T>, xPowers: set<set<T>>, y: set<T>,
    yPowers: set<set<T>>, powers: set<set<T>>, elem: T)//x=sdiff,
    xpowers=powersdiff y=powers
  requires psLemma1Pres(x,xPowers,y,yPowers,powers,elem)
  ensures IsPowerSet(y,powers)
{
  psLemma2(xPowers,yPowers,powers,elem);
}
```

prove that powers is the powerset of the so far handled elements stored in *s_diff*. The outer loop invariant *PSOuter*

```
predicate PSOuter<T>(s: set<T>, s_diff: set<T>, s_iter: set<T>,
    powers: set<set<T>>, powers': set<set<T>>)
{
  {}≤s_iter≤s ∧ {}≤s_diff≤s
  ∧ IsPowerSet(s_diff,powers) ∧ s_diff=s−s_iter
}
```

ensures that in each iteration *s_iter* and *s_diff* contain the elements as required in the previous paragraph, and that *powers* is the powerset of the so far handled elements stored in *s_diff*. The loop invariant *—powers— == Power(2,—s_diff—)* ensures that the size of the powerset of the so far handled elements is $2^{|s\_diff|}$.

6. *Proving the function belongs to the target complexity class* If we assign $|s| = n$ it has not yet proven to Dafny that $f(n) = 2^n$ is of $O(2^n)$. For this proof, we implement *expCalcLemma* explicitly to this definition of f:

```
lemma expCalcLemma(n: nat) returns(c: nat, n0: nat)
  ensures IsExpoFrom(c, n0, f)
{

  calc ⟸{
    f(n)≤Power(2,Power(n,c));
  }
  c, n0 := 1,1;
  expLemmaHelper1 (c, n0);
  assert ∀ k: nat • f(k)≤Power(2,k);
}
lemma expLemmaHelper1 (c: nat, n0: nat)
```

```
  requires c=1 ∧ n0=1
  ensures IsExpoFrom(c,n0,f)
{}
```

This lemma works through the steps that are required to infer proper c, $n0$ for $f(n) = 2^n$ Since this lemma ensures that f is exponential, the conditions of the *OExpnProof* lemma now hold.

Step 7. *Integrating the time complexity elements with the code* Finally, the functionally and the return value size verified version of the algorithm is complete. As in the previous examples, *powerset* and the recursive Fibonacci are implemented as a *refinement* of *ExpProofInterface*. That allows us to implement the explicit f and *expCalcLemma* for each algorithm.

Chapter 6

Conclusion

In this chapter, we discuss related work, in two aspects of our work; Software verification of time complexity and teaching with the use of formal verification. we conclude the thesis and discuss our contribution and further work.

6.1 Related Work

Formalization of teaching mathematics through formal verification by computers is introduced and researched in [BCM19],[Yan20]. The presented work in [BCM19] experiments on formalizing mathematical proofs using the Lean theorem prover [AMK17]. The experiment concludes that mathematicians with no computer science training can become proficient users of a proof assistant using formal verification. [Yan20] also presented an approach for teaching the material of a basic Logic course to undergraduate Computer Science students, tailored to the unique intuitions and strengths of this cohort of students. Our approach supports this objective and lifts it to more practical levels; by teaching the mathematics behind a complexity class and the methodology of proving that an algorithm belongs to the complexity class.

The use of formal methods in software development education is described, for example, in [CD15] and [Mor16]. Both approaches are dedicated to verifying functional correctness while our approach can illustrate the time complexity of algorithms to developers.

A verification of binary search that includes time complexity verification *VeriFun* [WS03b] has been demonstrated in [WS03a]. Since VeriFun is a semi-automated system

for verifying statements about programs written in a functional programming language, the loop's binary search implementation was written recursively. In our work, we use Dafny, which supports procedural and functional implementation.

The functional and time complexity analysis approach is presented in [WWC17], which is focused on creating a designated functional language. This approach applies only to functional programming implementation of algorithms. Since our approach using Dafny, we can generate the desired algorithm for several languages.

The motivation and concepts of the work of Gu'eneau, Pottier, and Chargu'eraud ([Gué19],[CP19],[GCP18]) are similar to ours: enabling robust and modular proofs of asymptotic complexity bounds along with their functional correctness. The methodology is built on specifying the intended behavior of a program, proving a theorem relating concrete code to the specification, and using the proof assistant *Coq* [BC04], [McC+18] to mechanically check every step of the proof. This methodology is based on the Coq proof assistant; it is highly expressive, requires more expertise in verification and less automated [LP13]. Coq can automatically extract executable programs from specifications to OCaml, which is functional and imperative, but the Coq specifications must be purely functional. Our methodology is based on Dafny, which supports functional and imperative programs [FGL18]. Furthermore, Dafny's program verifier is more automated than Coq [Kal+19] and provides the platform to write a verified version of an algorithm, including the implementation, the functional verification, and the time complexity in the same method. The target audience of our methodology is computer science students and algorithms developers. This audience is not necessarily an expert in verification, so we must provide a tool that can be easily assimilated.

The approach and methods presented in [McC+18] are based on the Coq proof assistant as well. The proof obligations establish the correctness of the functions and establish a basic running time result, but not an asymptotic one in terms of big-$\mathcal{O}$.

In [ZH18] a framework for verifying asymptotic time complexity of imperative programs is presented. It is done by extending Imperative HOL and its separation logic with time credits. The authors have stated that part of their future work goals is to include

reasoning about while and for loops (both functional correctness and time complexity) and build a single framework in which all deterministic algorithms typically taught in the undergraduate study can be verified straightforwardly. Our work fulfills both of these goals.

The methods of work in [Gué19],[CP19],[GCP18], [McC+18], [ZH18] presented above, implemented with proof assistants. As we mentioned earlier in Chapter 2, Dafny, as an auto-active program verifier [LM10b] provides automation by default, and the interaction with such a verifier happens at the level of the programming language, which has the advantage of being familiar to the programmer.

A use of a powerful tool for termination analysis and extending its approach for complexity analysis is demonstrated in [FG17]. This approach is limited to Java programs, has a steep learning curve, and involves more off-the-shelf solvers.

An approach for complexity analysis derived from modern architecture design is presented in [GMC09]. This approach is focused on yielding a bound on the total number of loop iterations of a respective program, with no reference to asymptotic analysis, which is essential to assimilate algorithm design methods to developers.

6.2 Conclusion

This thesis has presented methods for calculating, verifying, and exposing information about the algorithm's nonfunctional properties as part of its specification. The focus of the thesis is on asymptotic runtime. In addition, we have also demonstrated how we use this methodology to measure a powerset size. We formally defined big-$\mathcal{O}$ for linear, logarithmic, quadratic, and exponential notations and presented implementations for algorithms of each notation verified algorithm, functionally, and the specified time complexity. We have determined a methodology and demonstrated the effectiveness of implementing its steps on each of the presented algorithms and complexity classes.

The examples we chose are algorithms commonly taught in computer science courses such as data structures, computing, and complexity. The methodology presented in this thesis may be part of a new approach to teaching these courses; through formal

verification and not only based on pen and paper proofs. This teaching approach can make the homework assignments grading process more efficient; today, these assignments are submitted in writing, hence checking them must be performed manually. It sometimes requires hiring another person, specifically for this job, since the lecturer is not always available to do this work. This person must be qualified in the studied profession and available for this time consuming but not a well-paid job, which is very hard to find. Our approach enables us to automate part of the process.

This thesis is also explores the possibility of exposing nonfunctional properties as part of an algorithm's API. These features can extend the effectiveness of modern development methods based on microservices and serverless architectural design patterns to the next level; not only the developer gets all the functional information through APIs, but also nonfunctional information as time complexity is enabled.

Our initial investigations showed that this methodology could be adjusted to other algorithms. Algorithms such as BFS, DFS, and mergesort with the respective complexity classes O(n+m) and nlog(n), These algorithms can be implemented with the same methodology guidelines, but requires more effort from the user.

Implementing the asymptotic of time complexity and output size, as demonstrated with the powerset algorithm, could suggest that an implementation for space complexity is applicable using the same methodology. However, the challenge in space complexity is mostly in how to measure it and verify the size correctly; for example, the space of a variable can change several times inside the body of a loop, but the loop invariant verifies its condition only in the beginning and end of each iteration.

A more ambitious future work is the specification and verification of average-case time complexity, such as the quicksort average-case is nlog(n). This work includes the definition and a methodology of calculation probabilistic time complexity.

Appendix A

About the Code

All the presented code is arranged in modules as we elaborated on Chapter 6. The code is available at: `https://github.com/shirimo/thesis_files` The content and references to the thesis is detailed in the README file: `https://github.com/shirimo/thesis_files/blob/master/README.md`

Bibliography

[AMK17] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem Proving in Lean*. https://leanprover.github.io/theorem_proving_in_lean/. Last visited October 1st, 2020. 2017.

[BC04] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development Coq'Art: The Calculus of inductive constructions*. Jan. 2004. ISBN: 3540208542. DOI: 10.1007/978-3-662-07964-5.

[BCM19] Kevin Buzzard, Johan Commelin, and Patrick Massot. "Formalising perfectoid spaces". In: *arXiv e-prints*, arXiv:1910.12320 (Oct. 2019), arXiv:1910.12320. arXiv: 1910.12320.

[BS93] Jonathan Bowen and Victoria Stavridou. "Safety-Critical Systems, Formal Methods and Standards". In: *Software Engineering Journal* 8 (July 1993), pp. 189–209. DOI: 10.1049/sej.1993.0025.

[CD15] Dipak Chaudhari and Om Damani. "Introducing Formal Methods via Program Derivation". In: *Innovation and Technology in Computer Science Education (ITiCS)*. Ed. by Valentina Dagiene, Carsten Schulte, and Tatjana Jevsikova. June 2015, pp. 266–271.

[Che18] Lianping Chen. "Microservices: Architecting for Continuous Delivery and DevOps". In: *International Conference on Software Architecture (ICSA)*. IEEE, 2018, pp. 39–46.

[Coh+09] Ernie Cohen et al. "VCC: A Practical System for Verifying Concurrent C". In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 23–42. ISBN: 978-3-642-03359-9.

[CP19] Arthur Charguéraud and François Pottier. "Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits". In: *Journal of Automated Reasoning* 62.3 (Mar. 2019), pp. 331–365. URL: https://doi.org/10.1007/s10817-017-9431-7.

[FG17] Florian Frohn and Jürgen Giesl. "Complexity Analysis for Java with AProVE". In: *Integrated Formal Methods*. Ed. by Nadia Polikarpova and Steve Schneider. Springer, 2017, pp. 85–101. ISBN: 978-3-319-66845-1.

[FGL18] Ismael Figueroa, Bruno García, and Paul Leger. "Towards progressive program verification in Dafny". In: Sept. 2018, pp. 90–97. DOI: 10.1145/3264637.3264649.

[GCP18] Armaël Guéneau, Arthur Charguéraud, and François Pottier. "A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification". In: *Programming Languages and Systems*. Ed. by Amal Ahmed. Cham: Springer International Publishing, 2018, pp. 533–560. ISBN: 978-3-319-89884-1.

[GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. "SPEED: Precise and Efficient Static Estimation of Program Computational Complexity". In: *SIGPLAN Not.* 44.1 (Jan. 2009), pp. 127–139.

[Gué19] Armaël Guéneau. "Mechanized Verification of the Correctness and Asymptotic Complexity of Programs". PhD thesis. Dec. 2019.

[JSP10] Bart Jacobs, Jan Smans, and Frank Piessens. "A Quick Tour of the VeriFast Program Verifier". In: *Programming Languages and Systems*. Ed. by Kazunori Ueda. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 304–311. ISBN: 978-3-642-17164-2.

[Kal+19] F. Kalim et al. "Kaizen: Building a Performant Blockchain System Verified for Consensus and Integrity". In: *2019 Formal Methods in Computer Aided Design (FMCAD)*. 2019, pp. 96–104.

[Lei08] Rustan Leino. *This is Boogie 2*. `https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/`. Last visited October 1st, 2020. June 2008.

[Lei10] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370. ISBN: 978-3-642-17511-4.

[LM10a] K. Rustan M. Leino and Rosemary Monahan. "Dafny Meets the Verification Benchmarks Challenge". In: *Verified Software: Theories, Tools, Experiments*. Ed. by Gary T. Leavens, Peter O'Hearn, and Sriram K. Rajamani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 112–126. ISBN: 978-3-642-15057-9.

[LM10b] K. Rustan M. Leino and Michał Moskal. *Usable Auto-Active Verification*. 2010.

[LP13] Rustan Leino and Nadia Polikarpova. *Verified Calculations*. `https://www.microsoft.com/en-us/research/publication/verified-calculations`. Last visited October 1st, 2020. 2013.

[LSP07] Thierry Lecomte, Thierry Servat, and Guilhem Pouzancre. "Formal Methods in Safety-Critical Railway Systems". In: Aug. 2007.

[MB08] Leonardo de Moura and Nikolaj Bjørner. "Z3: an efficient SMT solver". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Apr. 2008, pp. 337–340.

[McC+18] Jay McCarthy et al. "A Coq library for internal verification of running-times". In: *Science of Computer Programming* 164 (2018). Special issue of selected papers from FLOPS 2016, pp. 49–65. URL: `http://www.sciencedirect.com/science/article/pii/S0167642317%20300941`.

[McG08] Catherine C. McGeoch. "Experimental Methods for Algorithm Analysis". In: *Encyclopedia of Algorithms - 2008 Edition*. Ed. by Ming-Yang Kao. Springer, 2008. DOI: 10.1007/978-0-387-30162-4_135.

[Mor16] Carroll Morgan. "(In-)Formal Methods: The Lost Art". In: *Engineering Trustworthy Software Systems*. Ed. by Zhiming Liu and Zili Zhang. Cham: Springer International Publishing, 2016, pp. 1–79. ISBN: 978-3-319-29628-9.

[MW05] Marc H. Meyer and Peter H. Webb. "Modular, layered architecture: the necessary foundation for effective mass customisation in software". In: *International Journal of Mass Customisation* 1.1 (2005), pp. 14–36. URL: https://www.inderscienceonline.com/doi/abs/10.1504/IJMASSC.2%20005.007349.

[Nip20] T. Nipkow. *Programming and Proving in Isabelle/HOL*. https://isabelle.in.tum.de/doc/prog-prove.pdf. Last visited October 1st, 2020. 2020.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. "PVS: A prototype verification system". In: *Automated Deduction—CADE-11*. Ed. by Deepak Kapur. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 748–752. ISBN: 978-3-540-47252-0.

[Rea20a] Micosoft Reaserch. *Lemmas*. https://rise4fun.com/Dafny/tutorial/Lemmas. Last visited October 1st, 2020. 2020.

[Rea20b] Micosoft Reaserch. *Modules*. https://rise4fun.com/Dafny/tutorial/Modules. Last visited October 1st, 2020. 2020.

[Rea20c] Micosoft Reaserch. *Multisets*. https://rise4fun.com/Dafny/tutorialcontent/Collections. Last visited October 1st, 2020. 2020.

[Rea20d] Micosoft Reaserch. *Sets*. https://https://rise4fun.com/Dafny/tutorial/Sets. Last visited October 1st, 2020. 2020.

[Wil02] Herbert S. Wilf. "Algorithms and Complexity". In: *Algorithms and Complexity - 2nd Edition*. New York: A K Peters/CRC Press, 2002. DOI: 10.1201/9780429294921.

[WS03a] Christoph Walther and Stephan Schweitzer. "A Verification of Binary Search". In: *Mechanizing Mathematical Reasoning: Techniques, Tools and Applications*. Ed. by D. Hutter and W. Stephan. Vol. 2605. Lecture Notes in Artificial Intelligence. Springer, 2003, pp. 1–18.

[WS03b] Christoph Walther and Stephan Schweitzer. "About VeriFun". In: *Automated Deduction – CADE-19*. Ed. by Franz Baader. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 322–327. ISBN: 978-3-540-45085-6.

[WWC17] Peng Wang, Di Wang, and Adam Chlipala. "TiML: A Functional Language for Practical Complexity Analysis with Invariants". In: *Proc. ACM Program. Lang. (PACMPL)* 1.OOPSLA (Oct. 2017), 79:1–79:26. URL: http://doi.acm.org/10.1145/3133903.

[Yan20] Noam Nisan Yannai A. Gonczarowski. *Mathematical Logic through Python*. https://https://www.logicthrupython.org/. Last visited October 1st, 2020. 2020.

[ZH18] Bohua Zhan and Maximilian P. L. Haslbeck. *Verifying Asymptotic Time Complexity of Imperative Programs in Isabelle*. 2018. arXiv: 1802.01336 [cs.LO].