Verifying Time Complexity of Binary Search using Dafny

Shiri Morshtein
School of Computer Science
The Academic College Tel-Aviv Yaffo
shirim66@gmail.com

Ran Ettinger
Department of Computer Science
Ben-Gurion University of the Negev
ranger@cs.bgu.ac.il

Shmuel Tyszberowicz
RISE - Centre for Research and Innovation in Software Engineering
Southwest University, Chongqing, China
Department of Software Engineering, Afeka Academic College of Engineering
Tel Aviv, Israel
tyshbe@tau.ac.il

Formal software verification techniques are widely used to specify and prove the functional correctness of programs. However, nonfunctional properties such as time complexity are usually carried out with pen and paper. Inefficient code in terms of time complexity may cause massive performance problems in large-scale complex systems. We present a proof of concept for using the Dafny verification tool to specify and verify the worst-case time complexity of binary search. This approach can also be used for academic purposes as a new way to teach algorithms and complexity.

1 Introduction

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 array 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 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 [3, 24]. 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:

$$0 \leq r \Rightarrow (r < |q| \text{ and } q[r] = key) \tag{1}$$

$$r < 0 \Rightarrow \text{ key !in q} \tag{2}$$

In addition, to specify the requirement that the binary search time complexity belongs to $O(log_2 n)$, we define $T(n)$—the binary search time complexity, where T is the time complexity of binary search on input of size $n = |q|$.

$$T(n) = O(log_2\, n) \tag{3}$$

A. Paskevich, J. Proença (Eds.): Formal Integrated
Development Environment 2021 (F-IDE 2021)
EPTCS 338, 2021, pp. 68–81, doi:10.4204/EPTCS.338.9

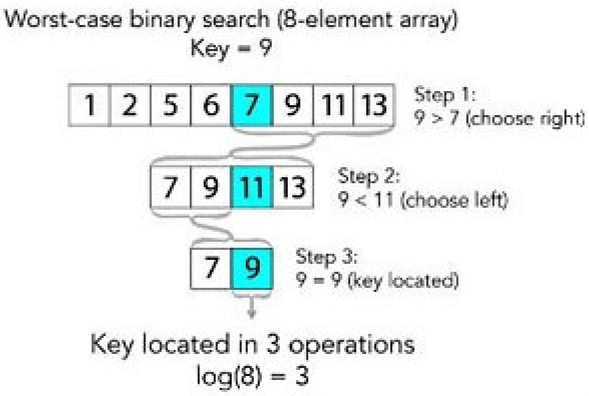

Figure 1: A running example of binary search.

i.e.,

$$\exists c > 0 \; and \; \exists n_0 > 0 \; s.t. \; \forall n. \; n \geq n_0 \Rightarrow T(n) \leq c \cdot log_2 n \tag{4}$$

In this paper we demonstrate the use of Dafny [13] for specifying and proving time complexity of the binary search algorithm; i.e., our main goal is to prove that the binary search time complexity belongs to $O(log_2 n)$. Dafny is an imperative language that supports formal specification using annotations. It builds on the Boogie intermediate language [14], which uses the Z3 automated theorem prover for discharging proof obligations [22]. By defining theorems and using pre- and postconditions (*requires* and *ensures*, respectively), loop invariants (*invariant*), assertions (*assert*), 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* [27]. Dafny is an auto-active program verifier [16], which is more automated than proof assistants such as Coq [12]. It provides the platform to write a verified version of an algorithm, including the implementation and verification in the same method.

Section 2 provides the definition, declaration, and specification of the *big-O* notation for logarithmic functions, implementing it by creating mathematical definitions for O($log_2 n$), followed by the verification of the functional correctness of the binary search algorithm. The challenges of using Dafny to create the nonfunctional specifications and proofs are presented in Section 3, that also illustrates a process and a methodology of proving an approximated time boundary for an algorithm, proving that the respective time boundary belongs to O($log_2 n$). This process presents an approach for computer science students and software developers to design and implement verified programs on a functional measure and for time complexity assessment. Our main contribution is the addition of the time complexity big-O definition to the specification and verification process. The methodology of defining and proving the correctness of the time complexity with Dafny (presented in the sections 4, 5, 6), arms the user with deeper understanding of time complexity big-O notation of the algorithm and assists with the implementation of efficient programs. The advantages of this approach compared to previous work are discussed in Section 7. The paper concludes in Section 8.

The code that is presented in this paper is available at:
https://github.com/shirimo/binary-search-verification and was run with Dafny 3.1.0: https://github.com/dafny-lang/dafny/releases/tag/v3.1.0 .

2 Specification of Logarithmic Complexity Class

The big-O notation is a formal way to express the upper bound of an algorithm's runtime. Since functions in Dafny define mathematical functions, *predicates* (Boolean functions) are used to define conditions that those functions must fulfill. The function f, belongs to O($log_2 n$) if:

$$\exists c > 0 \; and \; \exists n_0 > 0 \; s.t. \; \forall n. \; n \geq n_0 \Rightarrow f(n) \leq c \cdot log_2 n \tag{5}$$

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
{
  if n=1 then 0 else 1 + Log2(n/2)
}
```

The *nat* type stands for the natural numbers (non-negative integers), which is required by the $log_2 n$ function. Also, the $log_2 n$ function must require a positive input, hence the $n > 0$ precondition. The *decreases* annotation is used to assist Dafny to prove that the function terminates and calls itself with a valid value. It provides an expression that decreases with every recursive call and is bounded by 0. *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.

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 • t≤f(n) ∧ IsLog2(f)
}
predicate IsLog2 (f: nat -> nat)
{
  ∃ c :nat, n0: nat • IsLog2From(c, n0, f)
}
predicate IsLog2From (c :nat, n0: nat, f: nat -> nat)
{
  ∀ n: nat {:induction} • 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) and upper-bounds the algorithm's runtime 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 (5): 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 (5)). The *IsLog2* predicate determines if such positive c, n_0 that satisfies *IsLog2From* exist (the first part of expression (5)). 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 Functional Specification and Verification

We now show a fully verified implementation of the binary search algorithm. The use of binary search requires that the input sequence is sorted:

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

For verifying the functional requirement of binary search, as defined in Section 1, we need to ensure both expressions (1) and (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 the 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. The method body is given below:

```
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;
  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 [15] in the operation *(hi+lo)/2*.

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 the *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 defines that indices' variables are within the input sequence; the second 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 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.

4 Binary Search Logarithmic Time Complexity Specification

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 [19] (i.e., the number of loop iterations), 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 2. 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. 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 • t≤f(n) ∧ IsLog2(f)
}
```

This predicate would be satisfied when we prove such logarithmic function *f*, that upper-bounds the binary search algorithm's time complexity, exists. 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 lemma body is explicitly written for the function f in Section 6. The following sections are engaged in deriving this function f.

5 Derivation of a tight Upper-Bound Function

We now move from intuition to mathematics. To prove that $t \leq f(n)$, a suitable logarithmic function f must be provided. 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* [2] 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;
}
```

It is hard for the developer, and for 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 transition recursive function which 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 respective 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 actions performed by the binary search algorithm, 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 made also 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 $t \leq TBS(q,0,|q|,key)$, and that 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): In 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* [5] function in relation to the *Log2* [2] function:

We start by analyzing the stop condition. The function *TBS* stops and returns 0 when the inspected size is 0 (regardless if the size of the sequence is 0 or the difference between *hi* and *lo* is 0). However, *Log2* stops and returns 0 for the input 1. Therefore, 1 is added to the mathematical expression of *TBS*. Continuing with the analysis of the value divided by 2, *TBS* gets the value *lo+hi*, summing 2 sequence indices. This sum can at most be 2*$|q|$-1 (when hi is $|q|$ and lo is $|q|-1$). Therefore, the mathematical expression of *TBS* must be doubled. The last thing to consider is that the input for *Log2* is greater or equal 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:

$$n > 0 \Rightarrow (TBS(q,lo,hi,key) \leq 2 * log_2(|q|+1)+1) \tag{6}$$

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 (6). 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 been proven that $f(n) = 2*log_2(n+1)+1$ is logarithmic. The lemmas that have been specified in Section 4 are used to prove it.

6 Binary Search Upper-Bound is Logarithmic

So far we proved 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 expression (5). For implementing the body of *logarithmicCalcLemma* we use Dafny's *calc* construct [17], a theorem established by a chain of formulas, each transformed into the next. That is a replica of the pen and paper complexity proof, with a correctness guarantee by Dafny:

```
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 5(c). Also, $f(n)$ is proven to be logarithmic by the definitions that have been determined on Section 2.

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 $O(log(n)$ time complexity as defined in expression 3. Now our process is complete.

7 Related Work

Formalization of teaching mathematics through formal verification by computers is introduced and researched in [4],[9]. The presented work in [4] experiments on formalizing mathematical proofs using the Lean theorem prover [1]. The experiment concludes that mathematicians with no computer science training can become proficient users of a proof assistant using formal verification. 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, is presented in [9]. 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, e.g., in [6] and [20]. Both approaches are dedicated to verifying functional correctness while we can illustrate the time complexity of algorithms to developers.

A use of a powerful tool for termination analysis and extending its approach for complexity analysis is demonstrated in [8]. This approach is limited to Java programs, has a steep learning curve, and involves more off-the-shelf solvers.

A verification of binary search that includes time complexity verification *VeriFun* [23] has been demonstrated in [24]. 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. We use Dafny, which supports procedural and functional implementation.

The functional and time complexity analysis approach is presented in [25], which is focused on creating a designated functional language. This approach applies only to functional programming implementation of algorithms. Using Dafny, we can generate the desired algorithm for several languages.

The motivation and concepts of the work of Guéneau et al. ([11],[5],[10]) enable 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* [2], [4] 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 [17]. 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 [7]. Furthermore, Dafny's program verifier is more automated than Coq [12] 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 [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 [26] 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 [11],[5],[10], [18], [26] presented above, implemented with proof assistants. As we mentioned earlier in Section 1, Dafny, as an auto-active program verifier [16] 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.

8 Conclusion

We investigated the addition of the nonfunctional complexity property to the specification of algorithms. Using Dafny, this property can then be verified alongside the functional properties. The concept was demonstrated on an algorithm of a non-trivial time-complexity class, namely a logarithmic one.

Reflecting on the presented process, it appears that a methodology emerges for the specification and verification of time-complexity properties of algorithms. The methodology involves the following steps:

1. *Defining big-O notation.* This is a function of both runtime and input size of the algorithm.

2. *Verifying the correctness of the algorithm.* To simplify the calculation of the number of steps the algorithm performed, we implemented it with a single return point; It would be interesting to support algorithms with more than one return point.

3. *Adding specifications for time complexity verification.* This is done using a ghost variable that counts the number of elementary operations.

4. *Deriving a mathematical function that upper-bounds the time complexity.*

5. *Proving the function belongs to logarithmic complexity class.*

6. *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.

Our initial investigation presented a methodology that can be used for theoretical and practical measures. It refers to the complexity theory by using this process of specifying, implementing, and verifying algorithm complexity for academical causes. It is also practical since it generates fully verified code, and the algorithm's declaration includes the time-complexity specification as part of designing and writing the code processes.

This work is part of wider research on developing methods for specifying and proving time complexity of algorithms [21]. For example, given that *method A* calls *method B*, we explain how to find the time complexity of *A* using the calculated upper-bound of the ghost variable *t* in the postcondition of *B*.

References

[1] Avigad, J., de Moura, L., Kong, S.: Theorem proving in Lean. https://leanprover.github.io/theorem_proving_in_lean/ (2017), last visited April 30, 2021

[2] Bertot, Y., Castran, P.: Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions. Springer, 1st edn. (2010)

[3] Bloch, J.: Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken. https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html (2006), last visited March 6, 2021

[4] Buzzard, K., Commelin, J., Massot, P.: Formalising perfectoid spaces. arXiv e-prints arXiv:1910.12320 (Oct 2019)

[5] Charguéraud, A., Pottier, F.: Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. Journal of Automated Reasoning **62**(3), 331–365 (Mar 2019), https://doi.org/10.1007/s10817-017-9431-7

[6] Chaudhari, D., Damani, O.: Introducing formal methods via program derivation. In: Dagiene, V., Schulte, C., Jevsikova, T. (eds.) Innovation and Technology in Computer Science Education (ITiCS). pp. 266–271 (2015). https://doi.org/10.1145/2729094.2742628

[7] Figueroa, I., García, B., Leger, P.: Towards progressive program verification in Dafny. In: Proceedings of the XXII Brazilian Symposium on Programming Languages. pp. 90–97 (09 2018). https://doi.org/10.1145/3264637.3264649

[8] Frohn, F., Giesl, J.: Complexity analysis for Java with AProVE. In: Polikarpova, N., Schneider, S. (eds.) Integrated Formal Methods. pp. 85–101. Springer (2017). https://doi.org/10.1007/978-3-319-66845-1_6

[9] Gonczarowski, Y.A., Nisan, N.: Mathematical logic through python. https://www.logicthrupython.org/#:~:text=The%20textbook%20%22Mathematical%20Logic%20through,of%20this%20cohort%20of%20students. (2020), last visited April 30, 2021

[10] Guéneau, A., Charguéraud, A., Pottier, F.: A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In: Ahmed, A. (ed.) Programming Languages and Systems. pp. 533–560. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-89884-1_19

[11] Guéneau, A.: Mechanized Verification of the Correctness and Asymptotic Complexity of Programs. Ph.D. thesis, Inria, Paris, France (12 2019)

[12] Kalim, F., Palmskog, K., Mehar, J., Murali, A., Gupta, I., Madhusudan, P.: Kaizen: Building a performant blockchain system verified for consensus and integrity. In: Barrett, C.W., Yang, J. (eds.) Formal Methods in Computer Aided Design (FMCAD). pp. 96–104 (2019). https://doi.org/10.23919/FMCAD.2019.8894248

[13] Koenig, J., Leino, R.: Getting started with Dafny: A guide. In: Nipkow, T., Grumberg, O., Hauptmann, B. (eds.) Software Safety and Security - Tools for Analysis and Verification, NATO Science for Peace and

Security Series - D: Information and Communication Security, vol. 33, pp. 152–181. IOS Press (2012). https://doi.org/10.3233/978-1-61499-028-4-152

[14] Leino, R.: This is Boogie 2. https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/ (June 2008), last visited April 30, 2021

[15] Leino, R., Monahan, R.: Dafny meets the verification benchmarks challenge. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) Verified Software: Theories, Tools, Experiments (VSTTE). pp. 112–126. Springer Berlin Heidelberg (2010). https://doi.org/10.1007/978-3-642-15057-9_8

[16] Leino, R., Moskal, M.: Usable auto-active verification. In: T. Ball, L. Zuck, N.S. (ed.) Usable Verification Workshop (2010)

[17] Leino, R., Polikarpova, N.: Verified calculations. https://www.microsoft.com/en-us/research/publication/verified-calculations (2013), last visited April 30, 2021

[18] McCarthy, J., Fetscher, B., New, M.S., Feltey, D., Findler, R.B.: A Coq library for internal verification of running-times. Science of Computer Programming **164**, 49–65 (2018). https://doi.org/10.1016/j.scico.2017.05.001, http://www.sciencedirect.com/science/article/pii/S0167642317300941

[19] McGeoch, C.C.: Experimental methods for algorithm analysis. In: Kao, M. (ed.) Encyclopedia of Algorithms. Springer (2008). https://doi.org/10.1007/978-0-387-30162-4_135

[20] Morgan, C.: (in-)formal methods: The lost art. In: Liu, Z., Zhang, Z. (eds.) Engineering Trustworthy Software Systems (SETSS). pp. 1–79. Springer International Publishing (2014). https://doi.org/10.1007/978-3-319-29628-9_1

[21] Morshtein, S.: Methods of Verifying Complexity Bounds of Algorithms using Dafny. Master's thesis, School of Computer Science The Academic College Tel-Aviv Yaffo, Israel (2020), https://github.com/shirimo/thesis_files

[22] de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 4963, pp. 337–340 (04 2008). https://doi.org/10.1007/978-3-540-78800-3_24

[23] Walther, C., Schweitzer, S.: About VeriFun. In: Baader, F. (ed.) Automated Deduction (CADE). pp. 322–327. Springer Berlin Heidelberg (2003). https://doi.org/10.1007/978-3-540-45085-6_28

[24] Walther, C., Schweitzer, S.: A verification of binary search. In: Hutter, D., Stephan, W. (eds.) Mechanizing Mathematical Reasoning: Techniques, Tools and Applications. Lecture Notes in Artificial Intelligence, vol. 2605, pp. 1–18. Springer (2003)

[25] Wang, P., Wang, D., Chlipala, A.: TiML: A functional language for practical complexity analysis with invariants. Proc. ACM Program. Lang. (PACMPL) **1**(OOPSLA), 79:1–79:26 (Oct 2017), http://doi.acm.org/10.1145/3133903

[26] Zhan, B., Haslbeck, M.P.L.: Verifying asymptotic time complexity of imperative programs in isabelle. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) Automated Reasoning - International Joint Conference, (IJCAR). LNCS, vol. 10900, pp. 532–548. Springer (2018). https://doi.org/10.1007/978-3-319-94205-6_35

[27] Lemmas. https://rise4fun.com/Dafny/tutorial/Lemmas (2020), last visited April 30, 2021