DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

# Verifying Competitive-Programming Programs in a Rust Verifier

**Ahmed Rayan**

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

# Verifying Competitive-Programming Programs in a Rust Verifier

# Verifizierung von Programmen aus Competitive Programming in einem Rust Verifier

| | |
|---|---|
| Author: | Ahmed Rayan |
| Supervisor: | Prof. Dr. Susanne Albers |
| Advisor: | Vytautas Astrauskas, Prof. Dr. Peter Müller |
| Submission Date: | 15.04.2021 |

# Abstract

Ensuring the correctness of the created master solutions is one of the biggest challenges that people face in creating tasks for programming competitions such as ICPC and IOI. The task authors usually check the solutions' correctness by testing or peer review. However, that doesn't give sure guarantees on the solutions' correctness. In this thesis, we explore an alternative way to do that by using Prusti to formally verify the correctness of the Rust implementation to these master solutions. We investigate if we can find a general strategy to verify the solutions' correctness of the tasks belonging to each of the following competitive programming topics: Dynamic Programming (DP), DP on Trees, Greedy, and Segment Tree. Unlike the already existing published papers and archives that contain verification for the correctness of specific algorithms using other tools, we explore general strategies for whole classes of problems.

# Contents

# 1 Introduction

A big challenge in creating tasks for programming competitions such as the International Collegiate Programming Contest (ICPC) and the International Olympiad in Informatics (IOI) is ensuring that the created master solutions are correct. The task authors almost always provide an (informal) argument why a specific solution is correct. However, the correctness of the implementation is checked only by testing and peer-review. Using these techniques to ensure the correctness of the master solution doesn't guarantee that it is fully correct. On the other hand, human error can be eliminated by providing an automated way to verify the correctness of the implemented master solution.

Generally, the verification of program correctness is notoriously difficult. Moreover, this task for competitive programs is even harder because they are typically written in C and C++ languages, whose type-systems provide few memory safety guarantees.

Prusti [1] is a verification tool for Rust programs that tries to use the strong Rust type system to significantly simplify the verification. It allows developers to write contracts and annotations in the source code, which are then automatically checked to prove the functional correctness of the program. Internally, Prusti translates Rust programs to the Viper [2] intermediate verification language.

Since programs written in Rust have similar performance characteristics as the ones written in C and C++, Rust could be used as an alternative language to C and C++ in programming competitions for writing master solutions. Having the option to write master solutions in Rust enables the use of Prusti for verifying these solutions.

In competitive programming, there are clear classes of tasks based on what patterns can be used for solving them. It seems that these patterns also correspond to the vocabulary used by programmers in this field talking to each other. For example, saying that a problem will be solved using a minimum range segment tree gives the programmer an indication of the main structure of the solution program without even looking at the problem. However, two questions arise here. The first question is whether these solving patterns can indicate the verification strategy of the solution as it gives the main outline for the solution itself. The second question is whether Rust vocabulary matches that of C and C++, which might change the verification of these algorithms using Prusti. For example, tree representation in C/C++ is usually done by a parent array where the value in each cell represents the parent's index of the vertex with that index. However, the natural representation of a tree in Rust is done using a recursive Rust struct with the sub-trees as fields in the struct. We have investigated these two questions in this project.

**The main goal of this project is to find strategies to verify the correctness of the solutions implemented in Rust for competitive programming problems.**

## 1.1 Methodology

In this project, we aim to find a strategy for verifying a solution for some competitive programming problem based on its class. For this, we followed the following steps:

1. We selected a set of topics. Each of these topics represents one pattern or technique for solving tasks in the competitive programming community. The aim was to find a verification pattern for each of these topics. Mainly, the book Competitive Programming 3 [3] was used to choose that set of topics as it provides an organized explanation for most of the common topics in the field.

2. For each of the selected topics, which also represents a problem class, a set of problems was selected to verify their solutions. Competitive Programming 3 provides a large collection of problems for each topic contained in it. These problems also exist on uva online judge [4], which gives a way to test solutions against a set of prepared tests.

3. Translating the solution for each of the selected problems into Rust was performed. During the translation, we considered different implementation techniques, when feasible, for each of the selected topics and compared how different choices for implementation techniques affected the verification process.

4. Prusti was used to verify the implementation of the solution for each problem pointing out the relation between our choice of the implementation technique and how easy the verification is.

5. From the verified solutions of the selected problems from each topic, we tried to derive a general strategy for verifying the solutions' correctness of all problems in this topic.

## 1.2 Contributions

The main contributions of this thesis are the following:

1. We provide a general strategy to verify the correctness of any Bottom-Up DP solution by proving its equivalence to a simple Recursive Backtracking solution.

2. We provide three general strategies to verify any DP on Trees Solution's correctness and analyze the limitations and the use cases for each of them. All three strategies depend on proving the equivalence between the DP solution and a simple DFS algorithm.

3. We explain why it is hard (or infeasible) to find a general strategy to verify the correctness of an arbitrary Greedy solution.

4. We provide a general strategy to verify a static Segment Tree's correctness represented as a recursive Rust struct. Also, we explain the Prusti limitations preventing us from doing the same if we use an array structure to represent the segment tree.

## 1.3 Outline

In the next chapter, we present the necessary background on Rust and Prusti. Then, in the following chapters, we present the results of applying our methodology on different topics: Dynamic Programming (chapter 3), Dynamic Programming on Trees (chapter 4), Greedy (chapter 5), and Segment Tree (chapter 6). We conclude in chapter 7.

# 2 Background

In this chapter, we are giving the base knowledge that we need to understand about Rust and Prusti as we need them for the following chapters.

## 2.1 Rust

Rust[5] is a multi-paradigm programming language (concurrent, functional, generic, imperative, and structured) designed for performance and safety, especially safe concurrency. It aims to allow writing fast code like in C and C++ but with memory safety guarantees. Typically, achieving memory safety is a problem because it is done with a garbage collector or reference counting and both these methods have non-negligible performance impact. Rust achieves this by using an ownership type system, which is a compile time concept and has no run time overhead.

An example for an owned value is in the following code snippet:

```rust
let x = vec![1, 2, 3];  // Create a vector containing [1, 2, 3].
let y = x;              // Move the vector from x into y.
let z = x;              // Compiler error: x was already moved!
```

Having only owned values would be very inefficient because that would result in many memory copy operations. Therefore, Rust also supports references. The borrow checker checks that the program uses the references correctly and prevents various memory related mistakes. One example of the borrow checker preventing a mistake is in the following code snippet:

```rust
let mut x = vec![1, 2, 3];
let y = &mut x[1];  // Create a mutable reference into the second element.
x.push(4);          // Push 4 into the vector. This could cause the
                    // underlying memory block to be reallocated and
                    // in that case, y would become dangling.
*y = -2;            // Update the element of the vector via the reference.
```

In this example, the compiler reports an error and prevents us from creating two mutable references to the same object in order not to have a dangling pointer.

In addition to mutable references (`&mut T`), Rust has also shared references (`&T`) that have a guarantee that the referenced memory is immutable (if we ignore internal mutability, which is not relevant for our project).

## 2.2 Prusti

Formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics.[6]

Prusti is a verifier that allows its users to prove functional correctness of Rust programs. Rust Ownership plays a vital role in simplifying program verification using Prusti as owned values and mutable references give exclusive access to objects and shared references guarantee that referenced things are immutable. For example, if we take a look at the following Java function:

```java
void foo(Counter x, Counter y) {
    int v = x.getValue();
    y.increment();
    assert(v == x.getValue());  // Is this guaranteed to hold?
}
```

we don't have any guarantee on the value of the counter `x` as we might call `foo` function with the same object `z` for both arguments `x` and `y` (`foo(z, z)`). However, if we take a look at the Rust function that does the same work in the following code snippet:

```rust
fn foo(x: &Counter, y: &mut Counter) {
    let v = x.getValue();
    y.increment();
    assert(v == x.getValue());  // Is this guaranteed to hold?
}
```

we are sure that the value of the counter `x` would stay the same after the running of the function `foo`. For example, the Rust compiler would prevent us from calling foo in such a way: `foo(&z, &mut z)` because the mutable borrow `&mut z` guarantees that it is the unique reference to that memory location.

Prusti has multiple features that we can use to verify properties about the supplied Rust code. Here, we mention only the features which we used during our work on the project and will be encountered frequently in the next chapters. We are using the following example for Rust code that uses a loop to calculate the result of the addition of two non-negative integers to explain these features:

```rust
#[requires(x >= 0 && y >= 0)]
#[ensures(result >= 0 && result == x + y)]
fn add(x: isize, y: isize) -> isize {
    let mut currentResult = x;
    let mut addedValue = 0;
    while addedValue < y {
        body_invariant!(currentResult >= 0 && addedValue < y);
        body_invariant!(currentResult == x + addedValue);
        currentResult += 1;
        addedValue +=1;
    }
    currentResult
}
```

*#[requires(...)]* is a precondition that has to hold for the function arguments whenever it is called. *#[ensures(...)]* is a post-condition that has to hold for the function arguments and the return value after the function is fully executed (The return value is referred to by the keyword `result`]). There can be any number (including none) of pre-conditions and post-conditions attached to a function. When no precondition is specified, *#[requires(true)]* is assumed, and likewise for post-conditions. The expression inside the parentheses of requires or ensures should be a Prusti specification: Rust boolean expressions, quantifiers, and pure function calls.

As Rust is an imperative programming language that supports loops in its syntax, Prusti provides a way to deal with these loops in the verification process by allowing us to explicitly specify what properties should hold after any number of iterations for each loop in the program. To verify loops, including loops in which the loop condition has side effects, Prusti allows specifying the invariant of the loop body using the `body_invariant!(...);` statement. The expression inside the parentheses should be a Prusti specification. There may be any number of body invariants in any given loop, but they must all be written at the beginning of the loop body.

Another feature supported by Prusti is pure functions. Pure functions are functions that are deterministic and side-effect-free. In Prusti, such functions can be marked with the *#[pure]* attribute. They can take shared references as arguments, but they cannot take mutable references, because modifying the heap is considered a side effect. The main reason why we need pure functions is that, unlike the regular functions, we can use them in the Prusti specifications. An example of a pure function is in the following code snippet:

```
#[pure]
#[ensures(result == *a + *b)]
fn pure_example(a: &i32, b: &i32) -> i32 {
  *a + *b
}
```

# 3 Dynamic Programming (DP)

In this chapter, we explain how Dynamic Programming works in section 3.1, explore how to verify its implementation using Prusti in section 3.2, and conclude the whole chapter in section 3.3.

## 3.1 Explanation

In any programming contest, hammering every problem with Brute Force solutions doesn't allow the contestants to perform well due to the high complexity of these solutions. Higher complexity means higher running time, more memory allocation, or both. The Brute Force technique, which is also known as Complete Search, is a problem-solving method where we traverse the entire search space to obtain the required solution. During the search, pruning some parts of the search space can be helpful to optimize the solution if we are sure the required solution doesn't exist in these parts.

Usually, contestants tend to find a Complete Search solution for the problem they are facing as a first option. That is because a bug-free Complete Search solution should never give a wrong answer as it actually traverses the whole search space. Also, these solutions are the easiest to come by, develop and also debug. However, we need to pay attention to the input size and calculating the complexity of the solution making sure it has an acceptable expected running time.

Recursive Backtracking is one form to implement a Complete Search solution. This technique is helpful to solve problems by breaking them into smaller sub-problems that follow the optimal substructure property. The optimal sub-structure property means the solutions to the smaller sub-problems can be used to find the solution to the bigger problems.

The Fibonacci problem is one of the famous problems in the competitive programming community. It is considered a simple example of a problem that is solved using Recursive Backtracking. The first thing that needs to be done in order to develop a Recursive Backtracking solution is to define the problem *state*, which is the arguments defining each sub-problem. In the Fibonacci problem, the number that we are calculating the Fibonacci value for is enough. So, we need to calculate $fib(n)$. Then, we need to formulate the recurrence defining how to calculate the solution to a problem using the solutions to smaller sub-problems, which is known as *transitions*. For Fibonacci, it is defined as:

$$fib(n) = \begin{cases} 1 & if\ n = 0\ OR\ n = 1; \\ fib(n-1) + fib(n-2) & if\ n > 1 \end{cases}$$

This defined recurrence is then translated into a recursive backtracking function:

```rust
fn fib_rec(n: usize) -> i32 {
    if n <= 1 { 1 }
    else {
        fib_rec(n-1) + fib_rec(n-2)
    }
}
```

Complete Search is just one of the main $four$ problem-solving paradigms, which was mentioned in the book Competitive Programming 3 [3], needed to be mastered by Competitive Programming contestants in order to attack each problem with the appropriate tool for its solution. The other three problem-solving paradigms are Divide and Conquer (D&C), Greedy, and Dynamic Programming (DP). In this chapter, we focus on the fourth problem-solving paradigm which is Dynamic Programming (DP).

Dynamic Programming (DP) is primarily used to solve **optimization** problems and **counting** problems. Whenever we find some keyword in the problem description like "minimize this", "maximize that" or "count the ways to do that", there is a high chance that we are dealing with a DP problem.

Similar to Recursive Backtracking, the key skill required to master DP is the ability to determine the problem *states* and the relationships or *transitions* between the current problems and their sub-problems.

For any problem, we can consider it as a DP problem if it exhibits two main properties:

- **This problem has optimal sub-structures** which means the solution for the sub-problem is part of the solution of the original problem. This allows us to reconstruct the solution for big problems using the solutions of smaller problems. This property is the common thing between DP problems and Recursive Backtracking problems as shown in the Fibonacci problem.
- **This problem has overlapping sub-problems.** This property is the key we use to reduce the complexity of our Recursive Backtracking solution with a DP solution. The added concept is calculating everything only once. That means when we try to calculate the solution to a specific sub-problem that has been calculated before, we don't redo the calculations again but use the previously calculated and memoized value (this process is called **memoization**).

As shown in the figure 3.1, calculating the $fib(n)$ exhibits both properties of a DP problem. In order to calculate $fib(5)$, we calculate first both $fib(4)$ and $fib(3)$. Then, we use these calculated values to reconstruct the solution for $fib(5)$ and so on for all smaller sub-problems except $fib(1)$ and $fib(0)$ which have a known answer equals to 1. That demonstrates the **optimal sub-structures** property. The **overlapping sub-problems** is also obvious from the figure 3.1 as we see $fib(3)$ is calculated twice, once for calculating $fib(4)$ and another time for $fib(5)$. All smaller sub-problems are also calculated more than once (one time for each path from the node representing the main problem $fib(5)$ to the node representing the sub-problem).

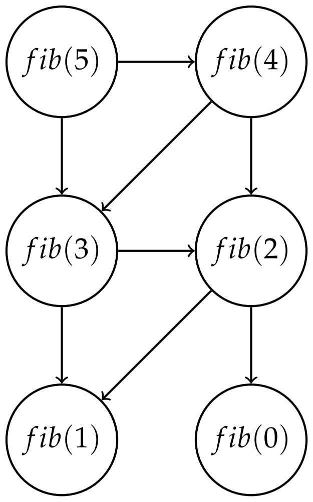

Figure 3.1: Fibonacci Recurrence

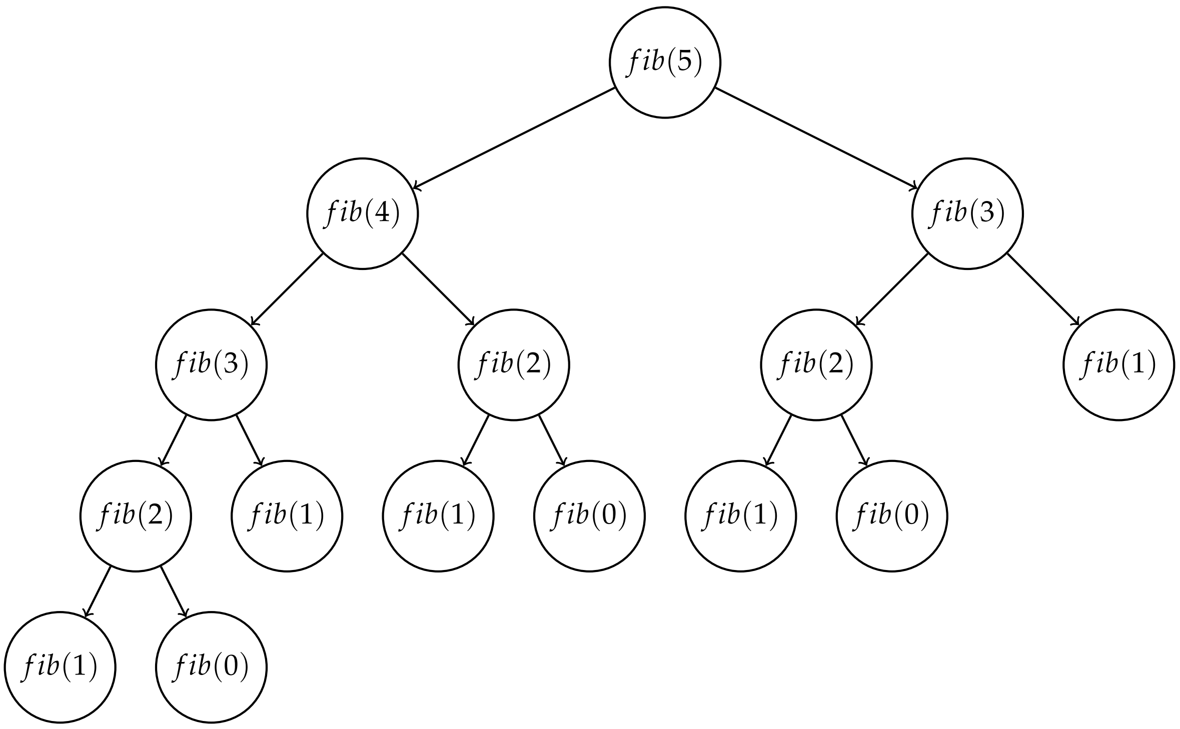

Figure 3.2: Fibonacci Recursive Backtracking Tree

Using Recursive Backtracking to solve the Fibonacci problem seems very natural. However, if we examine the complexity of such a program, we will find out it is really time-consuming as it takes exponential running time due to the **overlapping sub-problems** property. If we take a look at the Recursive Backtracking traversal tree in the figure 3.2, we will notice that in order to calculate the solution for $fib(n)$ in the root of the tree, we need to visit all nodes in an almost balanced binary tree with a height equal to $n$. That will result in running time complexity of $O(2^n)$.

Applying the **Memoizaion** concept to the Recursive Backtracking solution will result in a DP solution for the same problem. The first change is to add another argument to the recursive function which serves as a memoization table *memo*. This table should be initially filled with dummy values that cannot be a solution to any of the sub-problems. In the Fibonacci case, we might use $-1$ as we know all Fibonacci values are positive. In the function body, instead of just calling the recurrence directly, we first check if we already calculated the solution for that sub-problem before by checking the value in the memo table. In the case of having the answer in the table, we just return. Otherwise, we call the recurrence normally but we store the calculated value in the memoization table before returning it for future usages in other sub-problems. This idea is referred to as **Top-Down** DP. We start from the main top problem, then we move down calculating smaller sub-problems in order to find the solution to the bigger ones.

The following code snippet shows a **Top-Down** DP implementation to solve the Fibonacci problem:

```rust
fn fib(n: usize, memo: &VecWrapperI32) -> i32 {
    if n <= 1 { 1 }
    else if memo.lookup(n) != -1{
        memo.lookup(n)
    } else {
        let result = fib(n-1) + fib(n-2);
        memo.set(n, result);
        result
    }
}
```

The figure 3.3 shows the traversal tree for calculating $fib(5)$ using the Top-Down DP solution. By comparing it to the traversal tree for the Recursive Backtracking solution in the figure 3.2, we can see how much pruning is caused by the memoization. The tree is not balanced anymore with a significantly less number of nodes. This reflects on the running time complexity and reduces it to linear $O(n)$ instead of exponential.

In order to find a general time complexity for Top-Down DP programs, we should observe the number of times the memoization table is updated. We notice that we set a value to each cell representing a different sub-problem only once. This means the time complexity is $O(|memo|)$ which is usually polynomial in terms of the input.

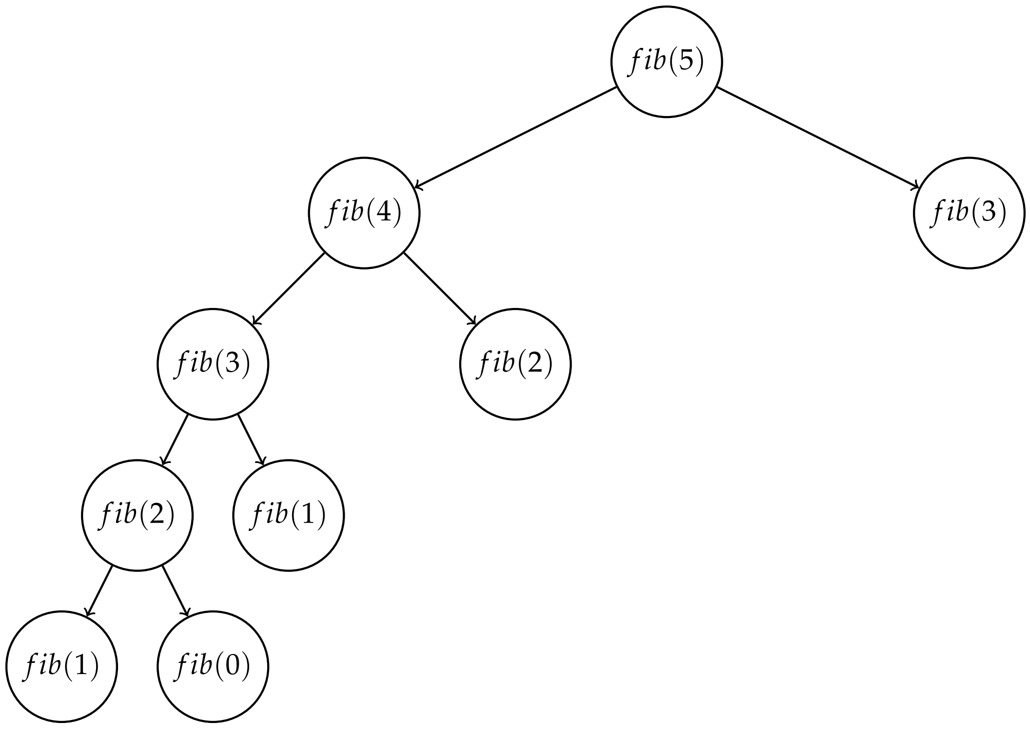

Figure 3.3: Fibonacci DP Tree

There is another way to implement DP solutions usually referred to as **Bottom-Up** DP. This is the true form of DP as DP was originally known as the tabular method which is a computation method involving a table. The general steps for **Bottom-Up** DP are as follows:

- Determine the required set of parameters that describe the problem uniquely, which is known as the problem **state**. This step is similar to what we discussed in Recursive Backtracking and **Top-Down** DP
- If there are $N$ parameters required to describe the problem state, an $N$ dimensional table will be initialized with one cell for every unique state. This table is somehow equivalent to the *memo* table used for **Top-Down** DP to store the calculated solutions for visited states. However, in **Bottom-Up** DP, we set the cells corresponding to the base cases in the DP table to the already known values of their solutions.
- After having all the base case cells already filled with the correct solutions, we determine what are the next cells that can be computed using only the already filled ones and set their value to the solutions for their corresponding state. This step is repeated until the whole table is complete. The whole process is usually done through iterations using loops.

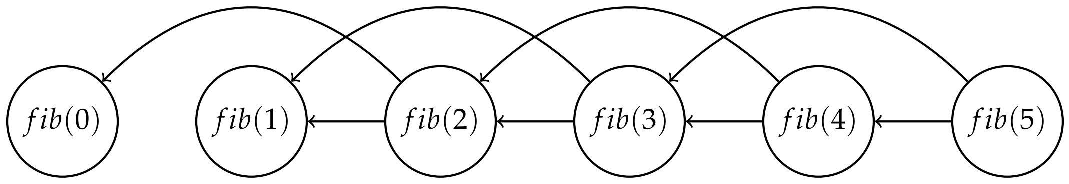

Figure 3.4: Flattened Fibonacci Recurrence

If we flatten the recurrence of Fibonacci as we can observe in Figure 3.4, we can see all the arrows are pointing to a left node. So, if we calculate the solutions for the states in that order from left to right, we will be sure that all solutions for the sub-problems needed to calculate the current problem are already computed and stored in the table. We can apply the three main steps for **Bottom-Up** DP as follows:

- We have only one parameter defining the problem state which is the number $n$ we are calculating the value $fib(n)$ for.
- As we have only one parameter in the problem state, we prepare a $1 - dimensional$ table $dp$, where $dp[n]$ will store the value $fib(n)$ by the end of the program. As we have $n = 0$ and $n = 1$ as base cases, we will set the values for the cells $dp[0]$ and $dp[1]$ to their known solution which is 1 for both cases.
- Now, we can move on to the next cell that is ready to be filled with its solution. In this case, $dp[2]$ is the only one as all of the smaller sub-problems have their solutions stored in the table. And by repeating this step for the next cells in order, we will have the whole table complete with the solutions for all problems stored in it.

We can see that explained procedure translated in the following code snippet:

```rust
fn fib(n: usize) -> i32 {
    if n <= 1 { 1 }
    else {
        let mut dp = VecWrapperI32::new(n + 1);
        dp.set(0, 1);
        dp.set(1, 1);
        let mut index = 2;
        while index <= n {
            let result = dp.lookup(index - 1) + dp.lookup(index - 2);
            dp.set(index, result);
            index += 1;
        }
        dp.lookup(n)
    }
}
```

## 3.2 Verification

In this chapter, we focus only on the verification of the **Bottom-Up DP** solutions for DP problems. The main goal here is to find a general strategy to verify the correctness of the Bottom-Up DP implementation for any general DP solution the same way we defined a general strategy for implementing these solution themselves.

As we have discussed in the previous section, Recursive Backtracking is the most obvious and the easiest to come by for any DP problem as it is a simple translation of the problem recurrence into code of the required programming language. Also, it is guaranteed for any DP problem to have a Recursive Backtracking solution because the only property needed in a problem to belong to the Recursive Backtracking class is having an **optimal substructure**, which is also a property needed in all DP problems along side with the **overlapping sub-problems** property.

Our main idea in verifying the correctness a Bottom-Up DP solution for any DP problem is proving the equivalence between that solution and the Recursive Backtracking solution for the same problem. The informal proof that is often done for master solutions typically explains why the recurrence is correct and the Recursive Backtracking solution is a direct translation of that recurrence into a programming language.

To demonstrate our verification strategy for DP problems, we will use the **SuperSale [7]** problem as an example. There is a SuperSale in some market. Every person can take only one object of each kind but for extra low price. Every person can take as many objects, as he/she can carry out from the SuperSale. We have given list of objects with prices and their weight. We also know, what is the maximum weight that a specific person can stand. What is the maximal value of objects we can buy at SuperSale?

The input of the SuperSale problem is given as a single integer number $N$ that indicates the number of objects $(1 \leq N \leq 1000)$, a list of integers $P$ containing $N$ values where $P[i]$ is the the price of the object $i$ $(1 \leq W[i] \leq 100)$, a list of integers $W$ containing $N$ values where $W[i]$ is the the weight of the object $i$ $(1 \leq W[i] \leq 30)$, and a single integer $MW$ describing the maximum weight a specific person can stand $(1 \leq MW \leq 30)$.

In the following part of this section we are showing how to modify the strategy for implementing a Bottom-Up DP Solution for a DP problem to verify the implementation's correctness along with the application of each step on the SuperSale problem. The strategy is as follows:

1. The first step is to determine the problem **state**, which is the required set of parameters that uniquely describe each of the sub-problems. In the SuperSale problem we have two parameters: $i$, which is the object we are deciding whether to take or leave, and $r$ indicating how much weight the buyer is still allowed to carry. According to these definitions, $solve(i, r)$ is the solution to the sub-problem described by $i$ and $r$.

2. Then, we need to define the problem **transitions**, which is the recurrence describing how to calculate the solution to a problem using the solutions to smaller sub-problems. That is usually described in a mathematical formula. The correctness of this recurrence

is usually demonstrated by an informal proof. The recurrence for the SuperSale problem is defined in the following formula:

$$solve(i, r) = \begin{cases} 0 & if\ i < 0\ OR\ r \leq 0; \\ solve(i-1, r) & if\ r \leq W[i]; \\ max(solve(i-1, r), P[i] + solve(i-1, r - W[i])) & otherwise \end{cases}$$

3. Implement the **Recursive Backtracking** solution as a **pure** function. This step should be a direct translation of the recurrence defined in the previous step into **Rust**. We will use this function as a trusted solution assuming its correctness to verify our Bottom-Up DP solution. This pure **Recursive Backtracking** solution function for the SuperSale problem is shown in the following code snippet:

```rust
#[pure]
#[requires(P.len() ==  W.len())]
#[requires(i >= 0 && i < P.len())]
fn solve_rec(
    P: &VecWrapperI32,
    W: &VecWrapperI32,
    i: isize,
    r: isize,
) -> isize {
    if index <= 0 || r <= 0 {
        0
    } else if r < W.lookup(i) {
        solve_rec(P, W, i - 1, r)
    } else {
        max(
            solve_rec(P, W, i - 1, r),
             P.lookup(i)
                + solve_rec(P, W, i - 1, r - W.lookup(index)),
            )
    }
}
```

In our implementation, we made the assumption that the object defined by index 0 is a dummy object which cannot be bought. This assumption will make it a lot easier to implement the DP solution as we don't have any sub-problem having a negative value for $i$. We will consider this assumption true for this step and all of the following ones as well and see how it simplified our further implementation.

We may also need to add some preconditions to the implemented pure function depending on the problem itself. In our case, we added two preconditions. The first one requires the length equality of the two input lists $p$ and $W$. The second one requires that the index $i$ falls within the acceptable range depending on our implementation.

4. Now we come to the step where we implement our **Bottom-Up DP** solution according to 3-steps procedure described in the previous section as follows:
   a) Defining the problem state is already done as the first step of the whole Verification process.
   b) The next step is to prepare the *dp* table with the same number of dimensions as the number of arguments in the problem state, where each dimension corresponds to one argument. This will result in a multi-dimensional table where each cell corresponds to a specific sub-problem. Then, we fill all cells corresponding to the base cases with the already known solution for them.
   c) The third step is to fill the dp table cells that can be filled as all sub-problems needed to calculate its value have been already calculated. This step is repeated until the whole table is complete. Usually, this process is implemented using nested loops where each loop is iterating over one dimension of the table (one problem state argument). That results in going through the cells with smaller sub-problems first going up to the bigger ones while being sure the needed values to calculate the current cell are already stored in the table. At the end, the value corresponding to the main problem is fetched from the table and returned as the answer for the whole function. We can see that done for the SuperSale problem as follows:

```
#[requires(P.len() ==  W.len())]
fn solve_dp(
    P: &VecWrapperI32,
    W: &VecWrapperI32,
    MW: isize,
) -> isize {
    // Initialize dp table
    let n = P.len();
    let mut dp = Matrix::new(n, MW + 1);

    // Fill base cases cells
    let mut w = 0;
    while w <= MW {
        dp.set(0, w, 0);
        w += 1;
    }

    // Fill cells in a bottom-up way
    let mut i = 1;
    while i < n { // LOOP0
        w = 0;
        while w <= MW { // LOOP1
```

```
            if w < W.lookup(i) {
                let result = dp.lookup(i - 1, w);
                dp.set(i, w, result);
            } else {
                let result = max(
                    dp.lookup(i - 1, w),
                    P.lookup(i)
                        + dp.lookup(i - 1, w - W.lookup(index)),
                )
                dp.set(i, w, result);
            }
            w += 1;
        }
        i += 1;
    }
    dp.lookup(n-1, MW)
}
```

5. Now we have finished the solution itself and we need to verify its correctness. As we discussed before, this task will be done by proving the equality between the result of the implemented Bottom-Up DP solution and the result of calling the pure Recursive Backtracking function with the arguments defining the main problem. Unfortunately, this cannot be done directly as we haven't verified the correctness of any values in the table, and hence, the returned value from the DP function.

| | | | | |
|---|---|---|---|---|
| 🟥 | 🟥 | 🟥 | 🟥 | 🟥 |
| 🟥 | 🟥 | 🟥 | 🟥 | 🟥 |
| 🟦 | 🟦 | 🟦 | | |
| | | | | |
| | | | | |

Table 3.1: A 2-dimensional DP table with $i = 2$ and $w = 3$

since the loop bodies are verified in isolation, we need to tell the verifier what is already computed and stored in the DP table. We can do that by providing the loop body invariants. The table 3.1 shows a 2-dimensional $5 \times 5$ DP table. The variable $i$ iterates over the rows in the outer loop of the DP solution and $w$ iterates over the columns in the inner loop. The tables shows its state when $i = 2$ and $w = 3$. At this point, all the coloured cells has the computed results for their corresponding sub-problem. The outer loop needs to have the information that all the red cell in the table have the correct values (all cells that have their first dimension index less than $i$), which can be done using one loop body invariant at the beginning of the loop. The inner loop needs to have the information that all the colored cells have correct values. The information about the red cells is added by a similar loop body invariant at the beginning of the inner loop as the one added at the beginning of the outer loop. We need an additional loop body invariant in the inner loop to verify the correctness of the blue cells (all cells that have their first dimension index as $i$ and a the second dimension index less than $w$).

For the general case, filling the table was implemented using nested loops where we had $K$ loops as $K$ is the number of arguments in the problem state and also the number of dimensions in the DP table. If we enumerate the loops as $LOOP_0, LOOP_1, .., LOOP_{K-1}$, we will have $LOOP_i$ responsible for iterating over the $i^{th}$ dimension of the table and it uses the variable $index_i$ as the iterating index with values in the range $[0, d_i[$[1].

For each of these nested loops $LOOP_i$, we will keep a set of cells in the dp table $SET_i$, where they are verified to be storing the correct solution for their corresponding sub-problems within the scope of that loop. The set $SET_i$ of verified cells by some specific loop $SET_i$ is the set containing all cells corresponding to the current indexes of all outer loops in relevance to that loop $(index_0, index_1, .., index_{i-1})$ and all indexes that is smaller than current index of the loop itself $(0, 1, .., index_i - 1)$.

The following formula describes $SET_i$ in a formal way where $SET_i = \varnothing$ for all $i < 0$.

$$\begin{aligned} SET_i = SET_{i-1} & \\ & \cup \{dp[index_0][index_1]..[t_i][t_{i+1}]..[t_{K-1}]; \\ & \quad where\ t_i \in [0, index_i[ \\ & \quad \wedge \forall j \in \mathbb{Z}\ (j \in ]i, K[ \implies t_j \in [0, d_j[)\} \end{aligned}$$

At this moment, we need to translate this mathematical formula to an actual Prusti annotation. We will be using `body_invariant!` statements as we need to provide conditions that hold for an iterative loop. For some $SET_i$, we will add the `body_invariant!` statements for $SET_{i-1}$ at the beginning of the $LOOP_i$ body, as we need to make sure the $LOOP_i$ doesn't violate any of the conditions assumed at the beginning of $LOOP_{i-1}$ body, followed by a new `body_invariant!` statement for the newly accumulated set of cells corresponding to this specific loop. However, in the most outer loop $LOOP_0$, we only add the second part. The general `body_invariant!` statement for a loop $LOOP_i$ should be as follows:

```
body_invariant!(forall(|v_i: isize, v_{i+1}: isize, .., v_{K-1}: isize|
    (v_i >= 0 && v_i < index_i
    && v_{i+1} >= 0 && v_{i+1} < d_{i+1} && .. && v_{K-1} >= 0 && v_{K-1} < d_{K-1})
        ==>  dp.lookup(index_0, index_1, .., v_i, v_{i+1}, .., v_{K-1})
            == solve_rec(index_0, index_1, .., v_i, v_{i+1}, .., v_{K-1})))
```

[1]For a mathematical range representation, we are using $[i, j]$ to represents a range from $i$ to $j$ where both ends are included. $[i, j[$ represents the range where $j$ is not included. $]i, j]$ represents the range where $i$ is not included. $]i, j[$ represents the range where both $i$ and $j$ are not included.

Applying this procedure to the SuperSale Bottom-up DP solution should update the nested loops part of the solutions as in the following code snippet:

```
let mut i = 1;
while i < n { // LOOP_0
    body_invariant!(forall(|k: isize, l: isize|
        (k >= 0 && k < i && l >= 0 && l <= MW)
            ==>  dp.lookup(k, l) == solve_rec(P, W, k, l)));
    w = 0;
    while w <= MW { // LOOP_1
        body_invariant!(forall(|k: isize, l: isize|
        (k >= 0 && k < i && l >= 0 && l <= MW)
            ==>  dp.lookup(k, l) == solve_rec(P, W, k, l)));
        body_invariant!(forall(|l: isize|
        (l >= 0 && l < w)
            ==>  dp.lookup(i, l) == solve_rec(P, W, i, l)));

        // Current cell calculation
        w += 1;
    }
    i += 1;
}
```

6. Now, we have all cells in the *dp* table verified to have the correct solution to its specific sub-problem, which is equal to the solution resulting from the pure Recursive Backtracking function. As we return the value stored in the cell corresponding to the main problem in the *dp* table to the solution function, we can now directly verify the correctness of the whole solution by adding a post-condition to it ensuring the result is equal to the one returned from the Recursive Backtracking function with the parameters of the main problem. Applying this to the SuperSale solution should give us the following:

```
#[requires(P.len() ==  W.len())]
#[ensures(result == solve_rec(P, W, P.len() - 1, MW))]
fn solve_dp(
    P: &VecWrapperI32,
    W: &VecWrapperI32,
    MW: isize,
) -> isize {
    // Solution body
}
```

For Some of the DP problems, the recurrence used to define the problem **transitions** might contain a universal quantifier. This results in an iterative work to calculate each state. The problem, here, is having a loop in the body of our Recursive Backtracking solution function, which is supposed to be pure in order to use in the **Prusti** annotations for verifying our Bottom-Up DP Solution.

The main idea to overcome the problem of iterative state work is to eliminate it by the replacement with an equivalent recursive work following these steps:

1. The first step is to implement an additional Recursive Backtracking function doing the same thing done by the original Recursive Backtracking solution except replacing the iterative work per state by a call to a helper function, that does the same work recursively.

2. Then, we verify the equality between the Recursive Backtracking function with the iterative work and the pure Recursive Backtracking function calling the helper. This is done by adding a post-condition to the first one ensuring that equality.

3. In the Bottom-up DP solution, we will use the pure recursive function in all the `body_invariant!` statements and post-conditions instead of the original recursive function with the iterative work. This should give us the same outcome, as by the transitivity of equality, the solution result will be also verified to be equal to the Recursive Backtracking solution with the iterative work.

4. We can perform the iterative work per state in the body of the most inner loop, as that is where we calculate the values for each cell. Additionally, we add `body_invariant!` statement to the iterative work loop, verifying the equality of the so far calculated result with the value returned from the pure recursive helper function corresponding to the same work. Also, we add all the `body_invariant!` statements for $SET_{K-1}$, that were already added to the most inner DP loop, to the iterative work loop.

## 3.3 Conclusion

Dynamic Programming(DP) is a technique to solve problems by breaking them down into overlapping sub-problems which follow the optimal substructure. The main goal of this chapter was to find a general strategy to verify the correctness of a Bottom-Up DP solution for an arbitrary problem with the DP properties. We managed to reach that goal by finding a strategy that works for a general problem by verifying the equivalence of the implementation of the Bottom-Up DP solution and a direct implementation of a Recursive Backtracking (Complete Search) solution to the same problem. This idea was practically feasible due to the huge similarities between the structure of both approaches.

# 4 Dynamic Programming on Trees

In this chapter, we explain how Dynamic Programming on Trees works in section 4.1, show how we represent trees in Rust in section 4.2, explore how to verify its implementation using Prusti in section 4.3, and conclude the whole chapter in section 4.4.

## 4.1 Explanation

Regular Dynamic Programming (DP), as explained in chapter 3, is a technique to solve problems by breaking them down into overlapping sub-problems which follow the optimal substructure. In this chapter, we are interested in the application of the DP technique to the tree structure.

A tree is a special connected graph the has the property $E = V - 1$, where $E$ is the number of edges and $V$ is the number of vertices in the graph. This results in the existence of one unique path between any pair of vertices in the tree. A rooted tree is a tree that has one special vertex called the root of the tree. In a usual tree traversal, we start from the root and follow the direct links (edges) to visit the rest of the nodes in the tree.

While discussing regular DP, we found a big similarity between the DP solution structure and the Recursive Backtracking solution structure of the same problem. If we consider an analogy with DP on trees solutions, the Depth First Search (DFS) traversal algorithm can be considered the equivalence to the Recursive Backtracking in the regular DP case. DFS on trees is a simple algorithm for traversing a tree. Starting from a distinguished source vertex (the root), DFS will traverse the tree **depth-first**. Every time DFS hits a node, DFS will choose one of the unvisited children and visit this vertex. DFS repeats this process and goes deeper until it reaches a vertex where it cannot go any deeper. When this happens, DFS will **backtrack** and explore another unvisited child, if any.

Both DP properties must occur in a problem that uses a tree structure in its solution to use the DP on that problem. The properties in the case of trees have some specific interpretation as follows:

- **The optimal sub-structures** property should be there in a tree problem in the form of the solution to the child nodes is a part of the solution to their parent node. That allows us to construct the solution for a specific node by combining the solutions of all its children.

  In DFS, we visit all the child nodes before we backtrack to the parent. If we modify the traversal algorithm to also calculate and return the solution value for the current node, we will have the solutions for all children nodes to the current one before backtracking and we can calculate the solution to the current one and return it. That allows any tree problem that has optimal sub-structures in the explained way to be solvable by DFS.

- **The overlapping sub-problem** property usually found in tree problems in two main forms:
    1. The first common form is using the calculated values for the tree nodes for further calculations in the main program. These further calculations can also be inside another DFS traversal.
    2. The second form is having multiple Values to calculate per node, where these values have interleaving formulas. That means to calculate one value $V_1$ for a specific node we need the values of $V_1$ as well as some other value $V_2$ for its children and the other way around as shown in the figure 4.1. In some cases, not all the arrows are there. However, we need some crossing between the two trees to have the overlapping sub-problems property.

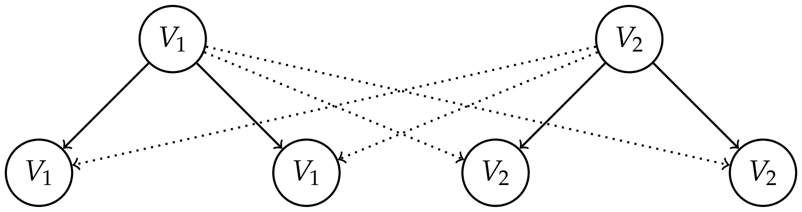

Figure 4.1: Tree Overlapping Sub-problems

Each of the two mentioned forms can exist alone in a tree problem. However, both of them can occur simultaneously by having multiple values with interleaving formulas that will be used later on for further calculations in the program. We will consider that the general case for DP on trees problems

Solving a tree problem that has only the optimal sub-structure property is usually done using a simple DFS algorithm in a linear time complexity $O(V)$. Each node is visited only once during the running of the Program. However, this complexity increases significantly when the overlapping sub-problems property exists. The extra cost differs according to which form of the overlapping sub-problems property is there. In the case of only reusing the DP values for further calculations, the complexity of the DFS increases to a quadratic linear time $O(V^2)$, if we assume using the DP value for each node a constant number of times. In order to calculate the DP value for one node, we have to traverse its whole sub-tree resulting in a linear complexity $O(V)$ for a single value. By multiplying the number of nodes and the complexity for calculating the value per node, we get the quadratic complexity for the whole program. However, the DFS algorithm complexity is a lot worse in case the overlapping sub-problems appeared in the second form of multiple values with interleaving formulas. The algorithm doesn't have a Polynomial complexity anymore but an exponential one $O(2^V)$ in the worst-case scenario. As we see in the figure 4.1 that shows only the first two levels of the tree, the DP values for the nodes in the second level are calculated twice for each node.

And, if we go deeper in the tree, each level doubles the number of times which we need to calculate the DP values for a node in that level resulting in the exponential complexity.

Using the DP technique on the tree problems that have both of the DP properties aims to solve them in a way similar to the simple DFS solution while keeping the solution complexity linear $O(V)$. The general idea is the same as using DP for a regular problem, which is storing the already calculated DP values for all sub-problems and reusing them whenever needed, Whether to calculate the solution for a bigger sub-problem or for latter usages in the program instead of entirely recalculating them.

The first step in solving any DP problem is determining the problem **state**. Fortunately, this step is quite easy for tree problems as the node for which we are calculating the DP values is, for most of the problems, sufficient to uniquely identify the sub-problem. Then, We usually try to determine the **transitions** between the problem states. In tree problems, the transitions come between a parent node and its children as we can define the solution to some node in terms of the solutions of its direct children.

In order to implement a DP solution on trees, we start by implementing a simple DFS algorithm that calculates the DP value (in case we have only one). When there are multiple DP values for each node, we implement a single DFS function that calculates all of them at the same time. The extra step we need to add to the simple DFS is storing the calculated values in some data structure, that we can use to fetch the stored values whenever needed. Also, the function fetches the solution values for the children of the current node in order to calculate its solutions.

In this chapter, we aim to provide a general strategy to verify the correctness of DP solutions to problems that use a tree structure.

## 4.2 Tree Representation

One of the main concerns, when we deal with tree problems, is how the tree will be represented in the programming language we are using to write our solutions. The tree representation affects the solution itself in major aspects such as the solution structure and its time and space complexity. However, dealing with the verification of tree problems in Rust increases the effect of the tree representation choice on the solution. The reason for that is mainly due to Prusti's limitations. In other words, some ways to represent a tree in Rust might make it more difficult or even prevent us from verifying the solution.

A straightforward way to represent a tree in Rust is by using the following struct:

```rust
pub struct Tree {
    data: isize,
    children: Vec<TreeNode>,
}
```

Here, `data` is the data we want to store at that tree node and `children` is the vector of children nodes of that tree node. However, due to Prusti technical limitations, we cannot use the regular `Vec` for `children`. One problem is that the `Vec` from the standard library does not have functional specification. This can be easily solved by providing a trusted wrapper `VecWrapperTree` that encapsulates `Vec<TreeNode>` and provides the specification. However, a larger problem is that we need to have a pure function that retrieves a child node based on its index in the following `lookup` function:

```
#[trusted]
#[requires (0 <= index && index < self.len())]
pub fn lookup(&self, index: isize) -> &Tree {
    self.v[index as usize]
}
```

However, we need this function to have `#[pure]` as an annotation, because we will be using it inside other pure functions and Prusti requires all functions used in a pure function to be also pure. Unfortunately, a pure function needs to have a primitive data type as a return type, which is not the case for the `Tree` data type. Due to these reasons, this way of representing trees in Rust has a big limitation when we try to verify the solutions as it makes it infeasible.

Instead of trying to give a strategy to verify DP solutions on general trees, we will use binary trees as a special case for trees. This is mainly to avoid the return of the non-primitive data type `Tree` in a function that needs to be pure. Representing a binary tree in Rust will be similar to how we represent the general tree. However, we will replace the array of trees with two fields, `left` and `right`, which are optional instances of other trees. We use `Option` to provide the cases of having only one child or even no children in case of a leaf node. The implementation of such binary tree representation is shown in the following code snippet:

```
pub struct Tree {
    v: isize,
    left: Option<Box<Tree>>,
    right: Option<Box<Tree>>,
}
```

Inside our solution, we can, now, access both left and right sub-trees directly in the pure and non pure functions exactly the same way. We can see the general way of doing so in the following code snippet:

```
fn access_sub_trees(node: &Tree) {
    match &node.left {
        None => {}
        Some(box l) => {} // Access the left sub-tree l in case it exists
    }
    match &node.right {
        None => {}
        Some(box r) => {} // Access the right sub-tree r in case it exists
    }
}
```

## 4.3 Verification

The main goal of this chapter is to find a general strategy to verify DP solutions for problems that use the tree structure. As mentioned in the section 4.1, any DP problem, that can be solved using DP, has a simpler solution with a DFS traversal algorithm with additional time complexity. These simple DFS solutions solutions are direct encoding of the recurrences, which are justified by the informal proof that is often done for master solutions. So, We will try to build our verification strategies to DP solutions by proving the equivalence between both types of solutions.

In this section, we will show three different implementation techniques for DP solutions on binary trees. All of these techniques aim to verify the correctness of a solution against the same simple DFS technique. Also, we will discuss the limitations of each technique along with the optimal use cases for each of them.

We will be using the problem **MaxCoinsInBinaryTree** to illustrate each technique with an example. The problem description is as follows: Given a tree $T$ of $N$ nodes, where each node $i$ has $C_i$ coins attached with it. You have to choose a subset of nodes such that no two adjacent nodes(i.e. nodes connected directly by an edge) are chosen and the sum of coins attached with nodes in the chosen subset is maximum.

When we try to solve this problem for a specific sub-tree, we try to make a decision on whether to include the root of this sub-tree in the selected set of nodes. If we take it, we cannot take any of its children but we can take grandchildren. However, if we leave this node, we can take all of its children if we want to. So, we define two values, $take(V)$ and $leave(V)$, for any node $V$, where $take(V)$ is the maximum number of coins we can take for the sub-tree rooted at the vertex $V$ if we take $V$ in the solution subset, and $take(V)$ is the same definition but if we don't take $V$ in the solution subset. Then, we can define the solution recurrences as:

$$take(V) = C_V + \sum_{i=1}^{n} leave(V_i)\ (where\ i\ iterates\ over\ the\ children\ of\ V)$$

$$leave(V) = \sum_{i=1}^{n} max(take(V_i), leave(V_i))\ (where\ i\ iterates\ over\ the\ children\ of\ V)$$

Both the tree representation and the DFS solution implementation will be the same in all of the three techniques. However, we might need to add extra fields in the tree structure in some of them, and we will mention that in place. But, the general tree representation for the MaxCoinsInBinaryTree problem is as follows:

```rust
pub struct Tree {
    coins: isize,
    left: Option<Box<Tree>>,
    right: Option<Box<Tree>>,
}
```

The implementation for the DFS solution to the problem is two pure DFS functions that directly translates the mentioned recurrences to Rust. Both functions are shown in the following two code snippets:

```rust
#[pure]
fn take(node: &Tree) -> isize {
    let mut result = node.coins;
    match &node.left {
        None => {}
        Some(box l) => {
            result += leave(&l);
        }
    }
    match &node.right {
        None => {}
        Some(box r) => {
            result += leave(&r);
        }
    }
    result
}
```

```rust
#[pure]
fn leave(node: &Tree) -> isize {
    let mut result = 0;
    match &node.left {
        None => {}
        Some(box l) => {
            result += max(take(&l), leave(&l));
        }
    }
    match &node.right {
        None => {}
        Some(box r) => {
            result += max(take(&r), leave(&r));
        }
    }
    result
}
```

If we consider the overlapping sub-problems in the definition of the recurrences of the MaxCoinsInBinaryTree problem, we will notice it appears in the form of having multiple values to calculate per node with interleaving sub-problems. That makes the complexity of the presented simple DFS solution exponential because of the reasons explained in the section 4.1. So, we will provide DP solutions to the same problems in the following parts of this section.

### 4.3.1 Using DP Tables

The original form of DP is the tabular method where we store the calculated values for each sub-problem in $dp$ tables and reuse them for later calculations of bigger problems. The first technique to implement and verify DP solutions on a tree structure is the closest to the tabular method and also to the implementation of DP solutions for regular DP problems. In this technique, we will provide a DFS function with an additional $dp$ table for each of the calculated values to transform it into a DP function. This table will be used to store the solutions and retrieve the stored values whenever needed.

The main strategy is described in the following steps:

1. The first step is to implement a pure DFS algorithm function for each value. These functions should be a direct translation for the problem recurrences to Rust. The implementation for the MaxCoinsInBinaryTree is the same as shown before.

2. The tree structure should be updated to include two additional fields. The first one is $N$ which is the number of nodes across the whole tree. The second field *index* is a unique index for each node that represents the position in the $dp$ tables that will hold the solutions for that node. Also, the index should be in the range $[0, N[$. For the MaxCoinsInBinaryTree, the tree structure should be as follows:

   ```
   pub struct Tree {
       coins: isize,
       N: isize,
       index: isize,
       left: Option<Box<Tree>>,
       right: Option<Box<Tree>>,
   }
   ```

   The implementation of the function that returns the index value for a node should ensure the correct range as in the following code snippet:

   ```
   #[pure]
   #[ensures (result == self.index)]
   #[ensures(result >= 0 && result < self.n)]
   fn index(&self) -> isize {
       self.index
   }
   ```

3. The value for the field $N$, which indicates the number of nodes in the whole tree, should be the same for all nodes in the tree. In order to ensure that, we will need a function that returns true whenever that condition is satisfied for the given tree. This function will be used as a precondition for all DP solution functions. We can implement that for all problems in the same way as follows:

```
#[pure]
pub fn same_n(node: &Tree) -> bool {
    let mut result = true;
    match &(*node).left {
        None => {}
        Some(box l) => {
            result &= same_n(&l);
            result &= (node.N == l.N);
        }
    }

    match &(*node).right {
        None => {}
        Some(box r) => {
            result &= same_n(&r);
            result &= (node.N == r.N);
        }
    }
    result
}
```

4. The next step is to implement a single DP function that takes the current node as an argument along with a *dp* table for each of the calculated values. The function calls itself with both children of the current node and uses the values stored for them in the *dp* tables to calculate the solutions for the current node. Finally, the function stores the calculated DP values in the correct positions corresponding to the current node in the *dp* tables.

5. The last step is to add a post-condition for each of the calculated values to the DP function ensuring the value stored in each *dp* table in the position indicated by the node index are equal to the values returned by the simple DFS solutions for the same node.

   Applying the last two steps to the MaxCoinsInBinaryTree problem should produce the function in the following code snippet:

```
#[requires(same_n(node))]
#[ensures(takeDP.lookup(node.index()) == take(node))]
#[ensures(leaveDP.lookup(node.index()) == leave(node))]
fn solve(
    node: &Tree,
    takeDP: &mut VecWrapperI32,
    leaveDP: &mut VecWrapperI32,
) {
    let mut takeResult = node.coins;
    let mut leaveResult = 0isize;
    match &node.left {
        None => {}
        Some(box l) => {
            solve(&l, takeDP, leaveDP);
            takeResult += leaveDP.lookup(l.index());
            leaveResult += max(takeDP.lookup(l.index()),
                                leaveDP.lookup(l.index()));
        }
    }
    match &node.right {
        None => {}
        Some(box r) => {
            solve(&r, takeDP, leaveDP);
            takeResult += leaveDP.lookup(r.index());
            leaveResult += max(takeDP.lookup(r.index()),
                                leaveDP.lookup(r.index()));
        }
    }

    takeDP.set(node.index(), takeResult);
    leaveDP.set(node.index(), leaveResult);
}
```

The limitation of this approach is that the calculated values for the root node (or the node being passed to the DP function in the main program) are the only values in the table verified to be equal to the results of the DFS functions. We lose the knowledge for all nodes deeper in the tree in the scope of the main problem. The main reason why this limitation exists is that calling the DP function on both left and right sub-trees recursively as a part of its body makes changes to the $dp$ tables. Also, because the indexes for the nodes don't have any structure with respect to the structure of the tree, we cannot specify the parts of the $dp$ tables the don't get mutated by some call. Then, we lose the information regarding the correctness of the calculated values corresponding to the nodes in the deeper levels.

This idea for implementing and verifying DP solutions for problems with a tree structure is ideal for those problems that calculate multiple DP values per node with interleaving formulas. However, if we need to use the calculated values for some nodes deeper in the tree in some later stages in the main program, the verification doesn't work anymore due to the limitation mentioned above.

### 4.3.2 Storing the DP Values in the Tree Nodes

The second proposed technique shares the same concept of DP to store the calculated DP values for the sub-problems and reuse them again whenever needed to calculate the solutions for the bigger problems instead of recalculating them again. However, there is a big difference in where the calculated values will be stored. This technique doesn't use the tabular method at all, as it stores the calculated DP values inside the tree nodes themselves instead of an additional table.

The main strategy is described in the following steps:

1. The first step is the same as in the previous technique. We need to implement a pure DFS algorithm function for each DP value.

2. For each DP value we need to calculate per node, we add an extra field to the tree structure that will hold the solution to that node once it is already calculated. The tree structure for the MaxCoinsInBinaryTree will be as follows:

```
pub struct Tree {
    coins: isize,
    take: isize,
    leave: isize,
    left: Option<Box<Tree>>,
    right: Option<Box<Tree>>,
}
```

   In this example, the `take` and `leave` fields will hold the values of $take(V)$ and $leave(V)$ when they get calculated, where $V$ is the vertex represented by the instance of the structure.

3. The next step is to implement a single DP function that takes the current node as a mutable argument. The function calls itself with both children of the current node and uses the values stored in the DP fields we added earlier to the tree structure to calculate the solutions for the current node. Finally, the function stores the calculated values in the DP fields of the current node. Applying this step to the MaxCoinsInBinaryTree problem should produce the function in the following code snippet:

```
fn solve(node: &mut Tree) {
    let mut takeResult = node.coins;
    let mut leaveResult = 0isize;
    match &node.left {
        None => {}
        Some(box l) => {
            solve(&mut l);
            takeResult += l.leave;
            leaveResult += max(l.take, l.leave);
        }
    }
    match &node.right {
        None => {}
```

```
            Some(box r) => {
                solve(&mut r);
                takeResult += r.leave;
                leaveResult += max(r.take, r.leave);
            }
        }

        node.take = takeResult;
        node.leave = leaveResult;
    }
```

4. In order to ensure that each node in the whole sub-tree hold the correct values in the dp variables, we need to implement another function that returns true when that condition holds and add it as a post-condition to the DP solution function. This function should return true if the values in the current node are equal to the values calculated by the DFS solutions and the same condition is true recursively for both of its children. Implementing this function for the MaxCoinsInBinaryTree problem will result in the following function:

```
#[pure]
fn dp_values_are_correct(node: &Tree) {
    let mut result = node.take == take(node) &&
                        node.leave == leave(node)
    match &node.left {
        None => {}
        Some(box l) => {
            result = result && dp_values_are_correct(l);
        }
    }
    match &node.right {
        None => {}
        Some(box r) => {
            result = result && dp_values_are_correct(r);
        }
    }
    result
}
```

The post-condition should be add to the DP solution function:

```
#[ensures(dp_values_are_correct(node))]
fn solve(node: &mut Tree) {
    // ...
}
```

Storing the DP values within the tree structure solves the limitation of the first technique that uses tables to store those values. It provides verification for the correctness of all stored values to be equal to the result of the DFS solution, not only for the root of the tree but also for all nodes in the deeper levels. However, this technique has its own limitations. The first one is that it makes changes to the input tree during the running time of the DP solution. Another limitation is the lack of providing random access to the calculated values for some arbitrary node. As the values are stored in the tree structure, we need to perform another tree traversal in order to fetch the DP values. As a result, the complexity of the whole solution to the problem might have increased complexity if it uses the calculated values in later stages.

This technique can be used for both use cases of DP on trees, whether we have multiple values with interleaving formulas or we need to use the calculated values later on in the solution program. However, we need to consider its limitations and make sure they would be acceptable for the problem we are solving.

### 4.3.3 Using Array Slices

In the third technique, we try to avoid the limitations of both the first and the second techniques. In this technique, we use the tabular method in order to provide random access to the calculated DP values. However, the way of using the tabular method is different from the first technique, in order to allow verification for the correctness of the DP values of all nodes in the tree.

The main concept is replacing the indexes given to the nodes in the tree, which doesn't have any relation to the tree structure, with another implicit indexing technique that gives the node in any sub-tree a continuous range of indexes. Having these continuous ranges of indexes for all sub-trees allows us to easily specify what cells in the DP table are changed and are changed and which are not by a specific function call.

The implicit indexing technique, that we will be using here, is defined by the preorder traversal. The preorder traversal will enumerate the nodes in the tree by giving the root the first index. Then, it will first traverse the left sub-tree followed by the right sub-tree, giving all nodes in the left sub-tree indexes smaller than all the nodes in the right one. The traversal is done recursively, making these indexing conditions valid for any sub-tree.

Another change to the tabular method explained in the first technique is building up the table during the running time of the DP solution instead of initializing it at the beginning and passing it as an argument to the solution function. This update will make it easier as we only care about the cells changed by the function call and nothing else.

The main strategy is described in the following steps:

1. The first step is the same as in the previous two techniques. We need to implement a pure DFS algorithm function for each DP value.

2. The next step is to implement the DP solution function, which takes the node as an argument and calculates all the DP values for that node at the same call. The function should return a tuple $(T_1, T_2, .., T_k)$ of array slices, each corresponding to one DP value. All of these array slices have an equal length, which is equal to the size of the sub-tree

rooted at the node passed to the function call. Each cell in any of the returned array slices contains the DP value of the node that is given the index of that cell by a preorder traversal starting at the root of that sub-tree.

In the body of the solution function, the function calls itself recursively on both the left and the right sub-trees. This step will result in two tuples of array slices $(L_1, L_2, .., L_k)$ and. $(R_1, R_2, .., R_k)$, each belonging to one of the child nodes of the current node. These array slices hold all the DP values for all nodes below the current node. In order to calculate the DP values $(v_1, v_2, .., v_k)$ for the current node, the function needs to access the DP values for both the left and the right children. This can be easily done as these values occur in position 0 in all of the array slices inside their corresponding tuples. Lastly, the function constructs its array slices tuple that contains the DP values for all the nodes in the sub-tree to return it as a result. In order to do that, the function constructs a tuple of array slices where every element is the result of appending the DP value of the passed node, the array slice from the left sub-tree, and the array slice from the right sub-tree. So, each $T_i$ should be equal to $[v_i]|L_i|R_i$, where the operator $|$ appends two array slices into a bigger one.

Applying this step to the MaxCoinsInBinaryTree problem results in the following function (due to Prusti limitations we are using Rust vectors instead of slices):

```rust
fn solve(node: &mut Tree) -> (VecWrapperI32, VecWrapperI32) {
    let mut takeResult = node.coins;
    let mut leaveResult = 0isize;

    let mut leftTake = VecWrapperI32::new(0);
    let mut leftLeave = VecWrapperI32::new(0);
    let mut rightTake = VecWrapperI32::new(0);
    let mut rightLeave = VecWrapperI32::new(0);

    match &node.left {
        None => {}
        Some(box l) => {
            let (a,b) = solve(&l);
            leftTake = a;
            leftLeave = b;
            takeResult += leftLeave.lookup(0);
            leaveResult += max(leftTake.lookup(0),
                                leftLeave.lookup(0));

        }
    }
    match &node.right {
        None => {}
        Some(box r) => {
            let (a,b) = solve(&r);
            rightTake = a;
            rightLeave = b;
            takeResult += rightLeave.lookup(0);
            leaveResult += max(rightTake.lookup(0),
                                rightLeave.lookup(0));
        }
    }

    let mut takeResultArray = VecWrapperI32::new(1);
    takeResultArray.sett(0, takeResult);
    let mut leaveResultArray = VecWrapperI32::new(1);
    leaveResultArray.sett(0, leaveResult);

    (takeResultArray.append(leftTake).append(rightTake),
        leaveResultArray.append(leftLeave).append(rightLeave))
}
```

3. The last step is ensuring that the values stored in the returned array slices from the DP solution function are equal to the values calculated by the DFS solution. In order to do that, we will implement a function that takes a node, a tuple of array slices of equal sizes, and a range inside these array slices having the length equal to the size of the sub-tree rooted at the passed node. The function returns true if the values stored in each of the array slices are equal to the results of the DFS solution for each node having the implicit index given by the preorder traversal inside the given range.

   Implementing this function for the MaxCoinsInBinaryTree problem should be as follows:

```
#[pure]
#[requires(e - b == size(node))]
fn dp_values_are_correct(
    node: &Tree,
    takeDP: &VecWrapperI32,
    leaveDP: &VecWrapperI32,
    b: isize,
    e: isize,
) {
    let mut result = takeDP.lookup(b) == take(node) &&
                        leaveDP.lookup(b) == leave(node);

    let mut leftIndex = b + 1;
    let mut rightIndex = b + 1;
    match &(*node).left {
        None => {}
```

```
            Some(box l) => {
                let leftSubSize = size(l);
                result = result &&
                    dp_values_are_correct(l, takeDP, leaveDP,
                                leftIndex, leftIndex + leftSubSize);
                rightIndex += leftSubSize;
            }
    }

    match &(*node).right {
        None => {}
        Some(box r) => {
            let rightSubSize = size(r);
            result = result &&
                dp_values_are_correct(r, takeDP, leaveDP,
                            rightIndex, rightIndex + rightSubSize);
        }
    }

    result
}
```

Here, the function `size` returns the number of nodes in the sub-tree rooted at the passed node. This can also be done efficiently using this technique of DP on trees as a pre-step.

Lastly, we add this function as a post condition for the DP solution function for all of the returned array slices. For the MaxCoinsInBinaryTree problem, it should be as follows:

```
#[ensures(dp_values_are_correct(node, &result.0, &result.1,
                                    0, size(node)))]
fn solve(node: &mut Tree) -> (VecWrapperI32, VecWrapperI32) {
    // ...
}
```

This approach provides random access to the calculated DP values for some node if we knew the implicit index given to that node by a preorder traversal. We can easily calculate the implicit indexes for all nodes in the tree by a single tree traversal which maps each node to its index. It also ensures that the calculated DP values for all nodes in the tree are correct. These two advantages of the proposed technique make it suitable for both use cases of DP on trees, whether having multiple values with interleaving formulas, or reusing the calculated DP values later in the program, or even both at the same time.

Prusti causes one limitation for that approach because it is missing native support for Rust slices. Using Rust slices, we can append two array slices in a constant time, which doesn't add any additional cost to our DP solution. However, if we use regular arrays and implement the append function in a linear complexity, the complexity of the whole DP solution becomes quadratic instead of linear in the worst-case scenario.

## 4.4 Conclusion

Applying the concept of Dynamic Programming (DP) to solve problems that use the tree structure in their solutions plays a big role in solving them efficiently. The main goal of this chapter was to find a general strategy to verify the correctness of DP solutions on trees for an arbitrary problem that uses trees and has the DP properties. We managed to reach that goal only for binary trees, due to Prusti limitations, by presenting three different techniques to implement and verify these solutions. All of the three presented techniques verify the correctness of the solution by proving the equivalence between the DP solution and the direct implementation of a DFS solution to the same problem.

# 5 Greedy

In this chapter, we explain what is Greedy in section 5.1, explore if we can verify its implementation using Prusti in section 5.2, and conclude the whole chapter in section 5.3.

## 5.1 Explanation

Greedy is another one of the four problem-solving paradigms mentioned in the competitive programming book. An algorithm is said to be greedy if it makes the locally optimal choice at each step with the hope of eventually reaching the globally optimal solution. Greedy solutions are short and they work in some cases. However, For many others, they do not.

A problem must exhibit these two properties in order for a greedy algorithm to work:

1. **The problem has optimal sub-structure**. This property also existed in the DP problems. The meaning of this property for greedy problems is that the optimal solution to the problem contains optimal solutions to the sub-problems.

2. **The problem has the greedy property**. If we make a choice that seems like the best at the moment and proceed to solve the remaining sub-problem, we reach the optimal solution. We will never have to reconsider our previous choices.

An Example of a greedy problem is The Frog Jumping problem: The frog begins at position 0 in the river. Its goal is to get to position $n$. There are lilypads at various positions. There is always a lilypad at position 0 and position $n$. The frog can jump at most $r$ units at a time. The goal is to find the minimum number of jumps the frog should make to reach the destination, assuming a solution exists.

We have a simple greedy algorithm for routing the frog home by jumping as forward as possible at each step. Here, the two previously explained properties hold as it has optimal substructure and the greedy property is choosing to jump to the farthest lilypad within the frog range. An implementation for that solution in Rust is in the following code snippet:

```rust
fn solve(positions: &VecWrapperI32, r: isize, currentIdx: isize) -> isize {
    if currentIdx == positions.len() - 1 {
        0
    } else {
        let nextIdx = currentIdx + 1;
        while nextIdx < positions.len() &&
                positions.lookup(nextIdx) -
                    positions.lookup(currentIdx) <= r {
```

```
            nextIdx += 1;
        }
        solve(positions, r, nextIdx - 1) + 1
    }
}
```

## 5.2 Verification

The goal of this chapter is to find a general strategy to verify Greedy solutions for competitive programming problems. In order to prove the correctness of some greedy algorithm, we need to prove both the feasibility and the optimality of the given algorithm. If we project these two characteristics on the Frog Jumping problem, we need to verify the provided solution produces a valid sequence of jumps from the start to the end positions without violating the constrain of the jump limit $r$. The optimality of the solution means the number of jumps resulting from the solution is optimal (there is no other valid sequence that has less number of jumps).

While trying to find the required strategy for verifying greedy algorithms, we faced the following problems:

- Despite having some common properties between all greedy problems, like the optimal sub-structure and the greedy property, the greedy property varies in a huge way from one problem to another. That makes the structure of each greedy solution different from the others. As we might notice in the Frog Jumping problem, the part where we calculate what greedy choice that we are taking at each step takes a big part of the solution. And, if that part completely changes between different greedy problems, finding one general strategy for verifying all greedy problems becomes really hard or even infeasible.

- If we aim to verify some greedy solution in a similar fashion that we followed when we tackled the verification of DP solutions, which is verifying the equivalence between that solution and a complete search solution, we will face a big problem. That problem is the big difference in the structure between a greedy solution and a complete search solution for the same problem.

  We can explain that by taking a look at the Frog Jumping problem. If we try to solve it in using Recursive Backtracking, which is feasible due to the optimal sub-structure property, we should have an algorithm like in the following code snippet:

  ```
  fn solve_rec(
      positions: &VecWrapperI32,
      r: isize,
      currentIdx: isize,
      nextIdx: isize,
  ) -> isize {
  ```

```
    if currentIdx == positions.len() - 1 {
        0
    } else if positions.lookup(nextIdx) -
                positions.lookup(currentIdx) > r {
        // INF is an a large number that is guaranteed to have
        // a value larger than the optimal solution
        INF
    } else {
        let take = solve_rec(positions, r, nextIdx, nextIdx + 1) + 1;
        let leave = solve_rec(positions, r, currentIdx, nextIdx + 1);
        min(take, leave)
    }
}
```

And if we compare this solution with the greedy one provided earlier, we will notice the huge difference between them. This solution explores everything in the search tree and gives the guaranteed optimal solution. However, the greedy solution calculates the greedy choice and relies on its correctness. It omits big parts of the search space. In order to verify the correctness of the greedy solution, we need to prove these omitted parts of the search space don't contain a better solution. However, proving this often requires intricate mathematical reasoning, for which the support is currently lacking in Prusti. On the other hand, Recursive Backtracking and DP solutions have the same structure but it does not revisit the parts of the search space that has been already visited before.

- Greedy solutions do some calculations on the run of the algorithm in order to calculate the greedy choice at each step, which adds extra hardness in proving the properties about that choice that might help to verify the correctness of the whole solution.

Due to these problems, we failed to achieve the goal of this chapter to find a general strategy to verify greedy solutions.

## 5.3 Conclusion

Greedy solutions are short and have an efficient running time in most cases. However, the big challenge is proving the correctness of these solutions. In this chapter, we tried to reach a general algorithm to verify the correctness of an arbitrary greedy solution. But, unfortunately, we failed to achieve that goal. We might consider the greedy technique being too broad class was one of the reasons that prevented us from reaching a general verification strategy. In that case, potentially, one could explore sub-classes to check whether then the verification strategies could be found.

# 6 Segment Tree

In this chapter, we explain what is Segment Tree in section 6.1, explore if we can verify its implementation using Prusti in section 6.2, and conclude the whole chapter in section 6.3.

## 6.1 Explanation

A Segment Tree is a data structure that can efficiently answer dynamic range queries. For dynamic problems, we need to frequently update and query the data. Due to time constraints, we focused on verifying tasks that only requires querying without updating the data. Verifying such algorithms is a prerequisite for handling the general version.

An example of such a range query that can be solved using a Segment Tree is the problem of finding the Range Minimum Query (RMQ). In this problem, we need to find the minimum element in an array within the range $[i, j]$. For example, given an array $A$ with the size $n = 7$ below, $RMQ(1, 3) = 13$ as 13 is the minimum value among $A[1]$, $A[2]$ and $A[3]$.

| values | 16 | 17 | 13 | 19 | 15 | 11 | 20 |
|---|---|---|---|---|---|---|---|
| indexes | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

There are multiple ways to implement a solution for that problem. The most naive way is to simply iterate the array from index $i$ to index $j$ and report the minimum value. However, this solution runs in $O(n)$ time per query. When $n$ is large and there are many queries, such an algorithm may be infeasible in the allowed running time constraints.

An alternative way to solve this problem efficiently is to use a Segment Tree, which is a way to arrange the data in a binary tree. Each node in the tree stores the result of the minimum range query for a specific range. The root of the tree represents the segment $[0, n-1]$. For each segment $[L, R]$ stored in some vertex in the tree where $L \neq R$, the range is split into $[L, \frac{L+R}{2}]$ and $[\frac{L+R}{2} + 1, R]$ for the left and right sub-trees, respectively. When $L = R$, the node corresponding to this range is a leaf and it stores the value $A[L]$ (or $A[R]$) as it is the only value in the range and hence, it is the minimum in that range. Otherwise, we build the segment tree recursively by comparing the minimum value of the left and right sub-segments and store the smaller one in the node of the current segment.

The whole build process creates $O(1 + 2 + 4 + ... + 2^{log_2 n}) = O(2n)$ segments, which is also the size of the tree. Therefore, the complexity to build the segment tree is $O(n)$ running time.

With the segment tree ready, we can answer a RMQ in $O(\log n)$. The answer for $RMQ(i, i)$ is trivial as we simply return $A[i]$. However, for a general query $RMQ(i, j)$, we can split it into two smaller ranges $[i, k]$ and $[k + 1, j]$, get the answer to both ranges $p1 = RMQ(i, k)$ and $p1 = RMQ(i + 1, j)$, then, construct the main solution as $RMQ(i, j) = min(p1, p2)$. We are showing how to use this idea along with the segment tree to answer a general RMQ using some examples below.

Take for example the query $RMQ(1,3)$. The process in figure 6.1 is as follows: Start from the root which represents segment $[0,6]$. We cannot use the stored minimum value of segment $[0,6] = 11$ as the answer for $RMQ(1,3)$ since it is the minimum value over a larger segment than the desired $[1,3]$. From the root, we only have to go to the left sub-tree as the root of the right sub-tree represents segment $[4,6]$ which is outside the desired range in $RMQ(1,3)$.

We are now at the root of the left sub-tree that represents segment $[0,3]$. This segment $[0,3]$ is still larger than the desired $RMQ(1,3)$. In fact, $RMQ(1,3)$ intersects both the left sub-segment $[0,1]$ and the right sub-segment $[2,3]$ of segment $[0,3]$, so we have to explore both sub-trees.

The left segment $[0,1]$ of $[0,3]$ is not yet inside the $RMQ(1,3)$, so another split is necessary. From segment $[0,1]$, we move right to segment $[1,1]$, which is now inside $[1,3]$. At this point, we know that $RMQ(1,1) = 17$ and we can return this value to the caller. The right segment $[2,3]$ of $[0,3]$ is inside the required $[1,3]$. From the stored value inside this vertex, we know that $RMQ(2,3) = 13$. We do not need to traverse further down.

Now, back in the call to segment $[0,3]$, we now have $p1 = RMQ(1,1) = 17$ and $p2 = RMQ(2,3) = 13$. Because p1 > p2, we now have $RMQ(1,3) = p2 = 13$. This is the final answer.

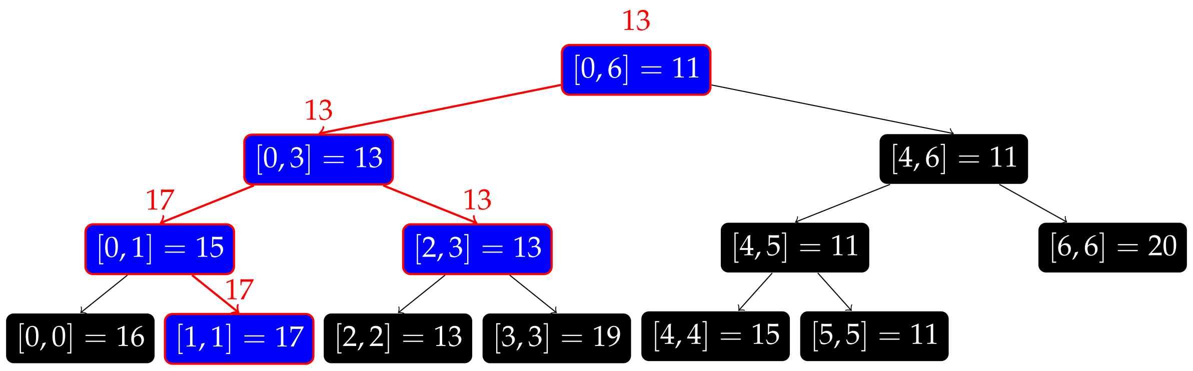

Figure 6.1: Segment Tree of Array $A = \{18, 17, 13, 19, 15, 11, 20\}$ and $RMQ(1,3)$

Now, let's take a look at another example: $RMQ(4,6)$. The execution in figure 6.2 is as follows: We again start from the root segment $[0,6]$. Since it is larger than the $RMQ(4,6)$, we move right to segment $[4,6]$ as segment $[0,3]$ is outside. Since this segment exactly represents $RMQ(4,6)$, we simply return the minimum element that is stored in this vertex, which is 11. Thus $RMQ(4,6) = 11$.

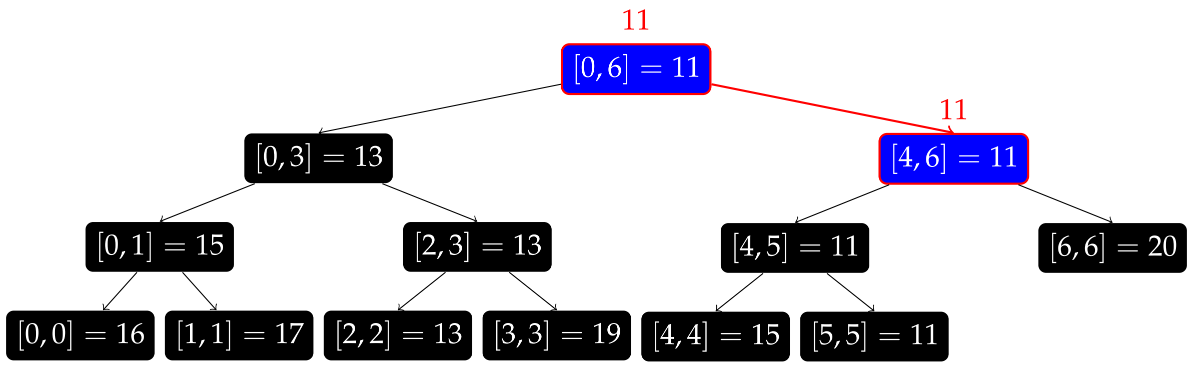

Figure 6.2: Segment Tree of Array $A = 18, 17, 13, 19, 15, 11, 20$ and $RMQ(4, 6)$

This data structure allows us to avoid traversing the unnecessary parts of the tree. In the worst case, we have two root-to-leaf paths which is just $O(2 \times log(2n)) = O(logn)$. Example: In $RMQ(3, 4) = 15$, we have one root-to-leaf path ($[0, 6] \rightarrow [0, 3] \rightarrow [2, 3] \rightarrow [3, 3]$) and another root-to-leaf path ($[0, 6] \rightarrow [4, 6] \rightarrow [4, 5] \rightarrow [4, 4]$).

## 6.2 Verification

In this chapter, we aim to find a general strategy to verify the correctness of a static segment tree that performs an arbitrary type of range queries. We will take a look at two different implementation strategies for segment trees in Rust and examine how the implementation choice affects the hardness and feasibility of verifying its correctness.

Our main idea to verify segment tree correctness is to prove the equivalence between the answer of the range query (RQ) we obtain from the segment tree and a naive recursive implementation for the same query over the input array we used to construct the segment tree.

In case of a RMQ segment tree, the simple recursive solution function that finds the minimum value within a given range in the input array should be implemented as in the following code snippet:

```
#[pure]
#[requires(lIdx >= 0 && rIdx <= array.len() && lIdx <= rIdx)]
#[ensures(forall(|i: isize| (i >= lIdx && i < rIdx) ==> result ==
    min(naive_rmq(array, lIdx, i), naive_rmq(array, i + 1, rIdx))))]
fn naive_rmq(
    array: &VecWrapperI32,
    lIdx: isize,
    rIdx: isize,
) -> isize {
    if lIdx == rIdx {
        array.lookup(lIdx)
    } else {
        min(array.lookup(lIdx), naive_rmq(array, lIdx + 1, rIdx))
    }
}
```

As we explained in how the segment tree works, we split the range that we are trying to find its answer into two smaller sub-segments. Then, we use their answers to construct the answer to the original range. The last post-condition added to this function plays a vital role in the verification process of the correctness of the segment tree. This post-condition ensures that if we split a range into two parts at any position, the answer for the whole range is the minimum of both answers of the two smaller parts. For a general static segment tree that performs the operation *op* (which has to be associative) over the range in the range query, we need to have the following post-condition added to the naive solution:

```
#[ensures(forall(|i: isize| (i > lIdx && i <= rIdx) ==> result ==
    op(naive_rq(array, lIdx, i), naive_rq(array, i, rIdx))))]
```

One assumption, that we will be making in this section, is the input array has a length that is a power of two. This assumption gives us the guarantee that the constructed segment tree is complete (all levels are full). Otherwise, we will pad the input array with dummy values until its length reaches the closest power of two.

### 6.2.1 Tree Structure

The first way of implementing the segment tree data structure is the most intuitive one. We use a simple tree structure with a `data` field that holds the answer to the range represented by it and two other optional trees representing its left and right children. This structure is as follows:

```
pub struct Tree {
    data: isize,
    left: Option<Box<Tree>>,
    right: Option<Box<Tree>>,
}
```

The first step we take when we deal with a segment tree is to build the segment tree from the input data array $A$. The build process is done recursively. If the range is $[i, i]$ with a length of 1, we just return a leaf node containing the answer to that range (which is the value of $A[i]$ in most cases). Otherwise, the range will be $[i, j]$ with a length greater than 1. In that case we split the range into two sub-segments $[i, \frac{i+j}{2}]$ and $[\frac{i+j}{2} + 1, j]$ of equal length, create their segment tree parts, and then use them as a left and right children of the node representing the whole range. The two sub-segments will always have equal length because the input array has a length which is power of two as we assumed before. This build process for a RMQ segment tree is implemented as the following code snippet:

```
#[requires(lIdx >= 0 && rIdx < array.len() && lIdx <= rIdx)]
#[requires(power_of_two(rIdx - lIdx))]
fn build(array: &VecWrapperI32, lIdx: isize, rIdx: isize) -> Tree {
    if lIdx == rIdx {
        Tree::new_leaf(array.lookup(lIdx));
    } else {
        let mid = (rIdx + lIdx) / 2;
        let left = build(array, lIdx, mid);
        let right = build(array, mid + 1, rIdx);
        let mut result = Tree::new_leaf(min(left.v(), right.v()));
        result.left = Some(box left);
        result.right = Some(box right);
        result
    }
}
```

We construct this segment tree in order to use it later to answer range queries over the input array. In order to achieve that and verify the correctness of these answers, we need to verify some properties about the structure and the correctness of the saved data at each node of the constructed segment tree. First we define the properties that describe the structure of the segment tree which is done exactly this way for all segment trees:

1. **Completeness**. This property is to ensure the completeness of constructed part of the segment tree which means all levels are full. This condition is true for some tree if both the left and right sub-trees are complete and have the same size. Also, the number of leaf nodes should be equal for both sub-trees in the case of a complete tree. We can implement this recursively in the following function:

```
#[pure]
fn is_complete(node: &Tree) -> bool {
    let mut result = true;
    let mut sizeL = 0;
    let mut leavesL = 0;

    let tL = &node.left;
    match tL {
        None => {}
        Some(box l) => {
            result &= is_complete(l);
            sizeL = size(l);
            leavesL = leaves_count(l);
        }
    }

    let tR = &node.right;
    match tR {
        None => {
        result &= sizeL == 0 && leaves_count(r) == 0;
        }
        Some(box r) => {
            result &= is_complete(r) && size(r) == sizeL &&
                leaves_count(r) == leavesL;
        }
    }

    result
}
```

2. **Tree to sub-tree size relation.** This property is to relate the size of a complete tree to the size of both of sub-trees rooted at the children of the root of the input tree. If a complete tree has the size $S$, then each of its left and right sub-trees should have the size $(S-1)/2$. Similarly, if a complete tree has $LN$ leaf nodes, each of its sub-trees should have $LN/2$ leaf nodes. We can implement this recursively in the following function:

```
#[pure]
#[ensures(is_complete(node) ==> result)]
fn correct_sizes(node: &Tree) -> bool {
    let mut result = true;
    let size = size(node);
    let leaves = leaves_count(node);
    let childrenSize = size - 1;

    let tL = &node.left;
    match tL {
        None => {}
        Some(box l) => {
            result &= size(l) * 2 == childrenSize;
            result &= leaves_count(l) * 2 == childLeaves;
        }
    }

    let mut sizeR = 0;

    let tR = &node.right;
    match tR {
        None => {}
        Some(box r) => {
            result &= size(r) * 2 == childrenSize;
            result &= leaves_count(r) * 2 == childLeaves;
        }
    }

    result
}
```

3. **The relation between the tree size and number of leaf nodes.** This property is the relation between the size of a complete tree and the number of leaf nodes in it. For any complete tree, we have $S = 2 \times LN - 1$. We can implement this in the following function:

```
#[ensures(is_complete(node) && correct_sizes(node) ==>
    size(node) == 2 * leaves_count(node) - 1)]
fn size_leaf_number_lemma(node: &Tree) {
    if is_leaf(node) {
    } else {
        let mut sizeL = 0;
        let mut leavesL = 0;

        let tL = &node.left;
        match tL {
            None => {}
            Some(box l) => {
                size_leaf_number_lemma(l);
            }
        }
    }
}
```

4. **Correct range length.** This property, that we need to ensure, is the relation between the range length represented by a node in the segment tree and the length of the range corresponding to that node. These two values should be equal for all nodes in the segment tree.

Then, we introduce the properties concerning the data themselves which needs some changes from a segment tree to another depending on the operation $op$ performed by the segment tree.

1. **Correct data calculation.** This property expresses that each non-leaf node stores the result of applying the segment tree operation $op$ on the values stared in its two child nodes. For an RMQ segment tree, we can implement such function recursively as the following:

```
#[pure]
fn correct_calculations(node: &Tree) -> bool {
    if is_leaf(node) {
        true
    } else {
        let mut range_min = INF;
        let mut result = true;

        let tL = &node.left;
        match tL {
            None => {}
            Some(box l) => {
                result &= correct_calculations(l);
                range_min = min(range_min, l.data());
            }
        }

        let tR = &node.right;
        match tR {
            None => {}
            Some(box r) => {
                result &= correct_calculations(r);
                range_min = min(range_min, r.data());
            }
        }

        result && range_min == node.data()
    }
}
```

2. **Array and tree equivalence.** This property is ensuring the equivalence between the resulting part of the segment tree and the corresponding range in the input array $A$. This means the segment tree stores the array values in its leaf nodes in the same order from left to right. This can be done recursively in the following function:

```
#[pure]
#[requires(lIdx >= 0 && rIdx < array.len() && lIdx <= rIdx)]
#[requires(power_of_two(rIdx - lIdx + 1))]
#[requires(is_complete(node))]
#[requires(correct_sizes(node))]
#[requires(leaves_count(node) == rIdx - lIdx + 1)]
fn array_tree_equivalent(
    node: &Tree,
    array: &VecWrapperI32,
    lIdx: isize,
    rIdx: isize,
) -> bool {
    if is_leaf(node) {
        node.v() == array.lookup(lIdx)
    } else {
        let mut result = true;
        let mid = (rIdx + lIdx) / 2;

        let tL = &node.left;
        match tL {
            None => {}
            Some(box l) => {
                result &=
                    array_tree_equivalent(l, array, lIdx, mid);
            }
        }

        let tR = &node.right;
        match tR {
            None => {}
            Some(box r) => {
                result &=
                    array_tree_equivalent(r, array, mid + 1, rIdx);
            }
        }
        result
    }
}
```

3. **Node values and naive solution results equivalence.** The last property is to ensure that each node in the segment tree the same value resulting from applying the naive range query on its range stored in its `data` field. For a RMQ segment tree, the following function shows the implementation for this property:

```
#[pure]
#[requires(lIdx >= 0 && rIdx < array.len() && lIdx <= rIdx)]
#[requires(power_of_two(rIdx - lIdx))]
#[requires(is_complete(node))]
#[requires(correct_sizes(node))]
#[requires(size_leaf_number_lemma(node))]
#[requires(correct_calculations(node))]
```

```
#[requires(leaves_count(node) == rIdx - lIdx)]
#[ensures(array_tree_equivalent(node, array, lIdx, rIdx) ==>
    node.v() == naive_rmq(array, lIdx, rIdx) && result)]
fn correct_data(
    node: &Tree,
    array: &VecWrapperI32,
    lIdx: isize,
    rIdx: isize,
) -> bool {
if is_leaf(node) {
    node.v()  == naive_rmq(array, lIdx, rIdx)
} else {
    let mut result = true;
    let mid = (rIdx + lIdx) / 2;

    let tL = &node.left;
    match tL {
        None => {}
        Some(box l) => {
            result &= correct_data(l, array, lIdx, mid);
        }
    }

    let tR = &node.right;
    match tR {
        None => {}
        Some(box r) => {
            result &= correct_data(r, array, mid + 1, rIdx);
        }
    }
    result && node.v() == naive_rmq(array, lIdx, rIdx)
}
```

These mentioned properties should be added as a post condition to the build function to ensure them on the resulting sub-tree that we will use later for answering range queries on the input array. Adding these properties for the RMQ segment tree build function should result in the following:

```
#[requires(lIdx >= 0 && rIdx < array.len() && lIdx <= rIdx)]
#[requires(power_of_two(rIdx - lIdx + 1))]
#[ensures(is_complete(&result))]
#[ensures(correct_sizes(&result))]
#[ensures(size_leaf_number_lemma(&result))]
```

```
#[ensures(leaves_count(&result) == rIdx - lIdx + 1)]
#[ensures(correct_calculations(&result))]
#[ensures(array_tree_equivalent(&result, array, lIdx, rIdx))]
#[ensures(correct_data(&result, array, lIdx, rIdx))]
fn build(array: &VecWrapperI32, lIdx: isize, rIdx: isize) -> Tree {
}
```

Having these properties for a segment tree should be enough to verify the correctness of the range query function of the segment tree against the naive range query function. The function implementation is implemented in the same way described in the description section as follows:

```
#[requires(nodeLIdx >= 0 && nodeRIdx < array.len() &&
    nodeLIdx <= nodeRIdx)]
#[requires(lIdx >= nodeLIdx && rIdx <= nodeRIdx && lIdx <= rIdx)]
#[requires(power_of_two(nodeRIdx - nodeLIdx + 1))]
#[requires(is_complete(node))]
#[requires(correct_sizes(node))]
#[requires(size_leaf_number_lemma(node))]
#[requires(leaves_count(node) == nodeRIdx - nodeLIdx + 1)]
#[requires(correct_calculations(&result))]
#[requires(array_tree_equivalent(&result, array, nodeLIdx, nodeRIdx))]
#[requires(correct_data(&result, array, nodeLIdx, nodeRIdx))]
#[ensures(result == naive_rmq(array, lIdx, rIdx))]
fn rmq(
    node: &Tree,
    array: &VecWrapperI32,
    lIdx: isize,
    rIdx: isize,
    nodeLIdx: isize,
    nodeRIdx: isize,
) -> isize {
    if lIdx == nodeLIdx && rIdx == nodeRIdx {
        node.v()
    } else {
        let mut result = INF;
        let midIdx = (nodeRIdx + nodeLIdx) / 2;

        let tL = &node.left;
        match tL {
            None => {}
            Some(box l) => {
                if lIdx < midIdx {
                result = min(result,
                    rmq(l, array, lIdx, min(rIdx, midIdx),
                        nodeLIdx, midIdx));
                }
            }
        }

        let tR = &node.right;
        match tR {
            None => {}
            Some(box r) => {
                if rIdx > midIdx {
                    result += min(result,
                        rmq(r, array, max(midIdx, lIdx), rIdx,
                            midIdx + 1, nodeRIdx));
                }
            }
        }

        result
    }
}
```

### 6.2.2 Array Structure

Using a tree structure to represent the segment tree data structure is considered intuitive as it is a direct translation to the natural concept of segment trees. However, it is not the most common way in the competitive programming community. Instead, we use a 1-based array *st* to represent the segment tree. This array is just rearranging to tree nodes into a linear data structure. The cell at index 1 (skipping index 0) is the root of the tree. The left and right child nodes of a node represented by index $p$ are the cells at index $2 \times p$ and $(2 \times p) + 1$ respectively.

Similar to the tree structure, each cell in *st* will hold the answer to a specific range. The root of the tree represents the segment $[0, n-1]$. For each segment $[L, R]$ stored in some cell in *st* where $L \neq R$, the range is split into $[L, \frac{L+R}{2}]$ and $[\frac{L+R}{2}+1, R]$ for the left and right children, respectively. When $L = R$, the cell corresponding to this range is a mapping from a leaf node in the binary tree. In the case of an RMQ segment tree, it stores the value $A[L]$ (or $A[R]$) as it is the only value in the range and hence, it is the minimum in that range.

Unfortunately, we currently cannot verify array-based segment trees in Prusti because the required computations lead to non-linear integer arithmetic, which is undecidable and Prusti does not have support for manual mathematical reasoning.

One of the computations that we need to represent and verify in the process of verifying the correctness of a segment tree is the length of the range represented by a node (or a cell in the case of array representation). For some cell with index $p$ in the segment tree $st$, we can calculate which level $l$ in the tree this cell should be in. In case we enumerate the level starting from 0 at the root node, $l = log_2 p$. And the length of the range covered by some node in some level $l$ is equal to $\frac{n}{2^l}$. This leads to having the length of the range covered by the cell with the index $p$ equal to $\frac{n}{2^{log_2 p}}$. However, using Prusti to verify that for all cells in the segment tree is infeasible as it doesn't support such complicated mathematical formulas.

## 6.3 Conclusion

A Segment Tree is a data structure that answers range queries on some input array of data. In this chapter, we aimed to reach a general strategy to verify the correctness of the segment tree in Rust. As a result of the work done here, we reached that strategy when we used a binary tree structure to represent segment trees. However, when we changed the choice of implementation to using array structure instead, we failed to achieve the goal of the chapter due to lack of support for non-linear arithmetic in Prusti.

# 7 Conclusion

We can conclude from this thesis that the solving pattern, that a competitive programming task belongs to, can indicate the verification strategy for the task's solution in a similar way to how it indicates a strategy for the solution itself. We have succeeded to provide a general strategy to verify the competitive programming tasks that belong to the Dynamic Programming (DP) class, DP on Trees class, or the Segment Tree class. However, we failed to achieve the same for the problems belonging to the Greedy class.

Also, we have shown that we can use Rust vocabulary to solve competitive programming tasks from the investigated classes. However, using the natural representation in Rust helps to make the verification of the solution correctness using Prusti more feasible or easier. For example, we managed to verify the correctness of segment trees when we used their natural Rust representation with a recursive Rust struct. However, we failed to do the same when we used an array for its representation.

## 7.1 Related Work

Ensuring the correctness of task solutions is one of the big challenges in the competitive programming field. There are a lot of online platforms (e.g. Codeforces [8], uva online judge [4], HackerRank [9] ..) that contain different tasks where each one is associated with a predefined set of tests. When a solution to a specific task is submitted on an online platform, it runs the solution on that set of tests in an attempt to ensure its correctness. Also, there are a lot of competitive programming contests, like ICPC and IOI, where the contestants are provided with a set of problems and they are required to provide correct solutions for them. In these competitions, the correctness of the provided solutions is also checked in the same way as in the online platforms. However, passing all these tests doesn't give any guarantee of the solution's correctness. On the other hand, the general strategies that we explained give a way to formally verify the correctness of the proposed solutions.

VerifyThis [10] is a series of program verification competitions, which have taken place annually since 2011. The competition offers a number of challenges presented in natural language and pseudo-code. Participants have to formalize the requirements, implement a solution, and formally verify the implementation for adherence to the specification. Both this competition and our thesis test the capabilities of verifiers. However, the difference is that this thesis tries to find general verification strategies.

Prusti is a young verification tool and we are the first to try using it to verify the correctness of known algorithms. However, there are more mature tools like Why3 [11] [12], Isabelle [13], and Coq [14]. Also, there are many attempts to verify algorithms and their implementations in different programming languages using these tools. One example of these attempts is verifying the correctness of Tarjan's strongly connected components algorithm in Why3, Coq, and Isabelle [15]. There are not only papers that use these tools for algorithms' correctness verification but also archives of verified algorithms like the Isabelle archive [16] and the Why3 archive [17]. Unlike these examples which formally verify the correctness of specific tasks, we presented general strategies to verify the correctness of a task solution depending on which competitive programming class this task belongs to.

## 7.2 Future Work

The work mentioned in this thesis can be extended in the following directions:

1. **Different Topics.** In this thesis, we explored only four competitive programming topics. In the future, more topics can be investigated to check whether we can find a general strategy to verify their solutions in Rust using Prusti or not.

2. **Testing Strategies.** The general strategies for verifying competitive programming solutions explained in this thesis can be put into more testing phases to prove if they function properly for general problems. To do that, we can try to apply these strategies to different problems belonging to each of the topics.

3. **Rust Slices Support.** As a future step, we can extend Prusti with the native support of Rust slices. We need this support for the third DP on Trees verification strategy that we mentioned in chapter 4.

4. **Greedy Sub-classes.** In chapter 5, we examined the Greedy class and failed to reach a general strategy to verify a Greedy solution. However, Greedy is a broad class that we might be able to divide into some smaller sub-classes. And, if we examine each of these sub-classes separately, we might reach a verification strategy for some of these sub-classes.

5. **Non-linear Mathematical support.** In chapter 6, we failed to provide a strategy to verify the correctness of a segment tree represented using an array structure due to the lack of non-linear mathematical support in Prusti. However, as a future step, we can try providing this support in Prusti and revisit the topic of verifying segment trees once again.

# List of Figures

# List of Tables

# Acronyms

| | |
|---|---|
| **DFS** | Depth First Search. 20 |
| **ICPC** | International Collegiate Programming Contest. 1 |
| **IOI** | International Olympiad in Informatics. 1 |
| **RMQ** | Range Minimum Query. 39 |

# Bibliography

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. "Leveraging Rust types for modular specification and verification". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–30.

[2] P. Müller, M. Schwerhoff, and A. J. Summers. "Viper: A verification infrastructure for permission-based reasoning". In: *International conference on verification, model checking, and abstract interpretation*. Springer. 2016, pp. 41–62.

[3] S. Halim, F. Halim, S. S. Skiena, and M. A. Revilla. *Competitive programming 3*. Citeseer, 2013.

[4] *uva online judge*. `https://onlinejudge.org/`.

[5] S. Klabnik and C. Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[6] A. Sanghavi. "What is formal verification?" In: *EE Times_Asia* (2010).

[7] *SuperSale Problem*. `https://onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=1071`.

[8] *Codeforces*. `https://codeforces.com/`.

[9] *HackerRank*. `https://www.hackerrank.com/`.

[10] G. Ernst, M. Huisman, W. Mostowski, and M. Ulbrich. "VerifyThis–verification competition with a human factor". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2019, pp. 176–195.

[11] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. "Why3: Shepherd your herd of provers". In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. 2011, pp. 53–64.

[12] J.-C. Filliâtre and A. Paskevich. "Why3 – Where Programs Meet Provers". In: *ESOP'13 22nd European Symposium on Programming*. Vol. 7792. LNCS. Rome, Italy: Springer, Mar. 2013. URL: `https://hal.inria.fr/hal-00789533`.

[13] L. C. Paulson. *Isabelle: A generic theorem prover*. Vol. 828. Springer Science & Business Media, 1994.

[14] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, et al. "The Coq proof assistant reference manual: Version 6.1". PhD thesis. Inria, 1997.

[15] R. Chen, C. Cohen, J.-J. Lévy, S. Merz, and L. Théry. "Formal Proofs of Tarjan's Strongly Connected Components Algorithm in Why3, Coq and Isabelle". In: *ITP 2019-10th International Conference on Interactive Theorem Proving*. Vol. 141. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2019, pp. 13–1.

[16] *Isabelle Archive*. `https://www.isa-afp.org/topics.html`.

[17] *Why3 Archive*. `http://toccata.lri.fr/gallery/why3.en.html`.