VERIFICACIÓN DE ALGORITMOS VORACES EN DAFNY

Trabajo de Fin de Máster

Curso 2021/2022

UNIVERSIDAD COMPLUTENSE
MADRID

FACULTAD DE INFORMÁTICA

MÁSTER INTERUNIVERSITARIO DE MÉTODOS FORMALES EN INGENIERÍA INFORMÁTICA

Autora: Paula Eugenia Pastor Pérez

Directores: Clara María Segura Díaz y Manuel Montenegro Montes

Madrid, 9 de Septiembre de 2022

Contents

Resumen

Verificación de Algoritmos Voraces en Dafny

La demostración de que un programa funciona de acuerdo a un comportamiento previamente establecido es lo que se conoce como verificación formal. El empleo de técnicas de verificación formal está incrementándose en los últimos años y en áreas de la industria del software. En este trabajo, se verificará formalmente el comportamiento de algunos algoritmos voraces en Dafny. Esto implicará llevar a cabo demostraciones matemáticas rigurosas en Dafny.

En el presente trabajo se introducirá Dafny en primer lugar. Es un lenguaje de programación que permite al usuario comprobar la corrección de su código. Para ello hará falta una especificación detallada del comportamiento esperado del código.

En segundo lugar, se explicarán con detalle los algoritmos voraces, se presentarán diferentes problemas donde se aplica el método voraz y se estudiará la estrategia que siguen para obtener una solución óptima. Se desarrollará la especificación de cada problema en Dafny y posteriormente se llevará a cabo la implementación del algoritmo voraz que lo resuelve, para finalmente poder verificarlo. Además, se diseñará una metología genérica con la idea de poder establecer unos pasos a seguir en la corrección de estos algoritmos voraces para ciertos problemas.

Palabras clave: verificación formal, algoritmos voraces, demostraciones matemáticas, Dafny, especificación, metodología de programación.

Abstract

Verification of Greedy Algorithms in Dafny

The proof that a program works according to a previously established behavior is known as formal verification. The use of formal verification techniques is increasing in recent years and in areas of the software industry. In this paper, the behavior of some greedy algorithms will be formally verified in Dafny. This will involve carrying out rigorous mathematical proofs in Dafny.

In this paper, Dafny will be introduced first. It is a programming language that allows the user to check the correctness of his/her code. This will require a detailed specification of the expected behavior of the code.

Secondly, greedy algorithms will be explained in detail, different problems where the greedy method is applied will be presented, and the strategy they follow to obtain an optimal solution will be studied. The specification of the code of each problem will be developed in Dafny and then the implementation of the greedy algorithm that solves it will be carried out in order to finally verify it. In addition, a generic methodology will be designed with the idea of being able to establish some steps to follow in the verification of these greedy algorithms for specific problems.

Keywords: formal verification, greedy algorithms, mathematical proofs, Dafny, specification, programming methodology.

Chapter 1

Introduction

It is not surprising to understand that programmers or algorithm designers want their algorithms to work as intended, and this requires formal verification techniques to ensure that programs work correctly. These techniques can be assisted by verification tools, such as Dafny. As the complexity of these algorithms and designs increase, the task of checking their correctness becomes more complicated. This is the main motivation for computer-assisted formal verification, to try to simplify this task.

Formal verification is a process of checking whether a design satisfies some properties or conditions; i.e. it is essentially concerned with identifying the correctness of a software or hardware design operation. Because mathematical proofs are used in verification, a mathematical model of the software design must be created. The growth in complexity of computer systems increases the importance of formal verification techniques.

One of the first areas to integrate the formal verification was the hardware industry. The complexity of the systems made it necessary to introduce this form of verification.

There are different approaches to formally verify a system. There is a wide range of tools available with different degrees of automation. Semi-automatic theorem provers have led to increased performance and automation of verification systems. Examples of this are SMT solvers like *Z3* [14]. Dafny [12], the language we will use during this project, is based on this solver. There also exist another verification systems like *HOL* [9], *Isabelle* [16] or *Why3* [1]. Both Why3 and Dafny work as *Verification condition generators* (VCG): from the code they produce verification conditions for an automated theorem prover, which then tries to prove that they are satisfied.

1.1 Goals

The main goal of this project is to be able to formally verify a selection of greedy algorithms in Dafny:

1. *Files* problem: we have a collection of files and we want to store the maximum number of them in a computer/device.
2. *Gas Station* problem: we have a collection of gas stations along a car route and we want to stop the least number of times to fill the tank.
3. *Chairlift* problem: we have a collection of people that have to travel in a chairlift and we want to pair them so that we use the minimum number of chairlifts.
4. *Hose* problem: we have a hose with holes and we want to put the minimum number of patches to fix it.

The greedy method is an approach for solving a problem in a number of steps. At each step, the algorithm takes the locally optimal choice. In some problems, this method is able to find an optimal solution, in others it is not. Our aim is to study some problems where the greedy strategy finds an optimal solution. We will specify and verify these problems in Dafny, with the aim of proving that the greedy solution is also optimal.

As a result, we have developed a methodology to prove the correctness of these algorithms. In this methodology we identify those proofs, concepts and definitions that are common to a wide range of greedy algorithms. In this way, we are able to establish the steps a programmer has to follow in order to verify his/her algorithm.

The developed code can be found at `https://github.com/PaulaPastorPerez/TFM`.

1.2 Work plan

This work has been carried out from February 2022 to August 2022. Let us see now the phases of this work:

1. First of all, the idea was to start with a simple problem where we could apply the greedy method and prove its correctness. The problem is the *files problem*, and it is explained in parallel to the development of the greedy algorithms in order to introduce them with a concrete example. This was carried out in March 2022.
2. After this example, we worked on more problems, as the *gas station problem* and *chairlift problem*. The idea was repeating something similar to what we did in the first case, but with more complex problems. This was carried out in April - May 2022.

In this stage we were able to begin to identify common elements in the specification and verification of these greedy algorithms, which served as the basis for developing a generic methodology to facilitate the verification of greedy algorithms.

3. In the last phase we must highlight the development of a generic methodology, which we elaborated once we had a few problems already done. We then applied this template to the previous problems and to a new one, *hose problem*. This was carried out in May - July 2022.

The structure of this work is divided in 5 chapters:

1. Dafny is the tool we use in this work. So, understanding how this tool works was essential to carry it out. Over the course of the months mentioned before, I took the subject *Computer-aided program verification*, where I was introduced to this new language and learned what is needed to address this project. An introduction to this language is given in Chapter 2.

2. The verification of greedy algorithms is the aim of this work. Being familiar with this kind of algorithms is necessary to understand it. That is the reason why our Chapter 3 is focused on introducing the Greedy method and how it works.

3. Chapter 4 explains the generic methodology we have developed. We looked for a way of generalising the steps to verify greedy algorithms, and once we obtained them we developed several problems using this methodology.

4. The last chapter is dedicated to the conclusions. A reflection on the work done is carried out, and aspects such as objectives achieved, difficulties encountered and possible future work are mentioned.

Chapter 2

Dafny: Main Features

Dafny is a program verification tool composed of a programming language and specification constructs. Dafny is designed to make it easy to write code and prove that it is correct [7, 13]. With that purpose, the Dafny user provides specifications with respect to which implementations need to be verified. What Dafny exactly checks is that programs terminate and satisfy the given specifications. It also checks that certain predefined operations that are partial, such as array index or division, are carried out under conditions that ensure that they are well defined.

The specifications of a Dafny method include preconditions and postconditions, which are properties that need to be verified respectively before and after calling that method. The implementors of these methods can assume that the preconditions hold when the method's execution starts and must prove that the postconditions hold at the end of the method's execution. On the other hand, the method caller has to ensure that the preconditions are fulfilled at the point of call and can then assume that the postconditions are fulfilled. These specification constructs are not only found in methods, but also in functions, predicates and lemmas. To further support specifications and verification, Dafny also offers ghost variables (all ghost elements do not compile), recursive functions, and types such as sets and sequences.

In this chapter we will introduce our readers into some important concepts of Dafny. To achieve the aim of this project, it will be necessary to focus on the specifications and on the verification of methods. We will continue presenting some Dafny types we have worked with, class types, and modules, which are used in the last part of our project to provide abstractions for greedy algorithms.

2.1 Specifications and functions

The aim of specifications is to describe properties of Dafny's methods, and functions. But in Dafny we cannot only write preconditions and postconditions, but also invariants, frames (i.e., memory locations that can be read or modified) and termination measures. Here is where we can find the real power of Dafny, when we are able to specify the behaviour of programs and then verify it with some help from the programmer.

Focusing on preconditions and postconditions, they must be declared with *requires* and *ensures* clauses, respectively. Both clauses are written below the header of a method, function, lemma or predicate. Dafny assumes that the preconditions hold at the beginning of the execution of the method, function, or predicate. However, when calling the method Dafny must check if these properties are met and, otherwise, we get an error. The *ensures* clauses can be found also in methods, functions, lemmas or predicates. The behaviour in this case is that Dafny assumes that the postconditions are met after a call as long as the preconditions are met. However, it must be ensured that the postconditions are met at the end of the method (or function, lemma or predicate). The absence of preconditions and postconditions corresponds to a *requires true* and *ensures true*, respectively. An *ensures true* clause requires termination.

Another concept we have not talked about, but necessary for our project, is the *decreases* clause. The clause is used to help verify termination. Sometimes we find these clauses in the specification of a function or a method, but where they are most commonly found is in loops. Although the *decreases* clauses are automatically inferred by Dafny in most cases, Dafny sometimes needs the user to write these expressions in order to prove that a recursive function or a while loop terminates. A *decreases* annotation is followed by an expression that is non-negative and decreases with every loop step or recursive call.

Example 1. We show a simple example to understand these three concepts:

```
function fib(n: nat) : nat
decreases n
{
    if (n = 0 ∨ n = 1) then n
    else fib(n−1) + fib(n−2)
}

method fibonacci(n: int) returns (f: int)
requires n ≥ 0
ensures f = fib(n)
```

In this sample there is no implementation, just verification-related code. We define a function *fib* that receives a natural number *n*. But also we give Dafny a *decreases* clause to help it prove the termination of *fib*. Non-recursive functions do not need any *decreases* clause. In this context, Dafny checks whether the expression given by the decreases clause is always non-negative and its value decreases at each recursive call. Later we will see an example where this clause is used in a loop. We use the *fib* function in the postcondition of the *fibonacci* method, to ensure that the value returned by the *fibonacci* method verifies the

condition `f = fib(n)`. □

Function *fib* takes a natural number and returns another natural number. The difference between methods and functions is that in the first case we can find all sorts of statements in the body; however, in a function body we just can have a side-effect free expression, with the specified result type. In our case, it must have natural type. To be able to implement the *fib* function, we need to use an *if expression*. The condition of this expression must be a boolean expression and both branches must have the same type. Another difference between functions and methods is that functions are transparent, while methods are opaque. That is to say, when applying a function during reasoning, the body of its definition is visible, while when calling a method, we only know its precondition and postcondition, but nothing about its internal implementation. The principal advantage of functions is that we can use them in specifications.

By default, a function is ghost. This means that it is used for verification purposes only, and it is never executed at runtime, so no machine code is generated for that function. If we want to be able to execute a function at runtime, we need to make it a *function method*.

2.2 Methods: specification and verification

The syntax of Dafny is similar to that of a typical imperative programming language. We can find methods, variables, types, loops, if statements, arrays, and more. One of the most important and basic elements of Dafny is the *method*. We have seen already an example of a method with its corresponding specification. Later, we will see the same example but with an implementation of the method (its body) and the elements that help in its verification. First of all, we have the header, where we have to specify each input parameter, and after that we must write *returns* followed by the output parameters. Note that the types of all the parameters must be specified after a colon separator. As before, we need to state the preconditions and postconditions with *requires* and *ensures* clauses.

The body of the method is enclosed inside curly braces. The body contains a series of statements, such as assignments, loops, calls to other methods or lemmas, etc.

Example 2. Let us see how the body of our *fibonacci* method is:

```
method fibonacci(n: int) returns (f: int)
requires n ≥ 0
ensures f = fib(n)
{
    var i, f, fsig := 0, 0, 1;
    while (i < n)
        decreases n − i
        invariant 0 ≤ i ≤ n ∧ f = fib(i) ∧ fsig = fib(i+1)
        {
            f, fsig := fsig, f + fsig;
            i := i + 1;
        }
}
```

□

First of all, to assign a value to a variable in Dafny we do not use the = symbol. Instead, we have to use the := symbol and all statements must be followed by semicolon, (;). Another important detail in the Dafny syntax is the simultaneous assignment, you can give value to several variables in the same line of code. In the example above you can see: `f, fsig := fsig, f + fsig;` where we are assigning to *f* the value of *fsig* and to *fsig*, the value of *f* + *fsig*. As this is done simultaneously, the entire right-hand assignment is calculated with the original values, not with the ones after the left-hand assignment is made. The first line of the body code is an assignment to a local variable. Let us explain this new concept. Local variables are declared inside the body of the method just adding `var` before it. For such variables there is no need to declare their type, since Dafny can infer it in almost all situations. The behaviour of these variables is just as we expect, we declare them whenever we need it and we can change their value any time we want. In our example, there are two local variables that we need to declare: *i* and *fsig*; in both cases the type annotation is not necessary.

To return a value from a method, we need to assign a value to the variable that denotes the result of the method, as defined in the method's header. In our example this is *f*, and the method will return its value at the end of the method.

Let us focus on the *while* statement. Verification of loops in Dafny require determining which properties hold throughout the iterations of the loop (called loop invariant), so that they can be asserted once the loop has terminated; and proving that the loop terminates.

Loop invariant is a property which holds between iterations; that is before entering the loop, between loop iterations, and after the last iteration of the loop. As the name says, it is an expression that is invariant, it remains true between loop iterations, although it might not hold temporarily in the middle of an iteration. Just like preconditions and postconditions, they are boolean expressions that must be verified. For example, the first invariant we see in our *fibonacci* loop is the expression $0 \leq i \leq n$; this means that i is always between those values, because it starts as 0, increases in one at each iteration and the loop finishes when its value is n, so it will never take values outside this range. The second invariant ensures some properties that the variables *f* and *fsig* must satisfy and this helps Dafny prove that the postcondition holds.

In some cases, Dafny cannot verify by itself the correctness of these invariants. To help Dafny with these proofs, we can add in the body of the loop some assertions. These can be declared in an *assert* clause. As invariants, they are boolean expressions that must be true when the execution reaches the point in which they are defined, and they can be placed anywhere in methods, *while* loops or lemmas.

Dafny is able now to prove that *fibonacci*'s code terminates. As we have seen before, the *decreases* clause helps Dafny know the *while* does not loop forever. In many situations Dafny does not need this as it can guess by itself the right annotations. However it is not a bad idea to made it explicit and write the *decreases* clause always. Dafny needs to prove, when given a decreases expression, that it actually gets smaller after every loop iteration and that it is

bounded. In most cases, a variable with an integral or natural value is the most appropriate candidate to be the decreases expression. In these cases, the bound is assumed to be zero.

Let us see this in our example. As decreases expression we have $n - i$, where n is a parameter given as input so we cannot modify it, then it is a constant; and i is a local variable that starts as 0 and increases with every loop iteration. Then it is easy to see that $n - i$ decreases with every loop iteration. Now let us prove that this expression is bounded. The invariant holds that $0 \leq i \leq n$, so it verifies that $n - i \geq 0$.

2.3 Types

Dafny types [11] can be classified as value types and reference types. The first ones are composed of basic scalar types (*int*, *nat*, *bool*...) and built-in collection types (*seq*, *set*, *multiset*...). These types are *immutable*.

For any type T, each value of type `set <T>` is a finite set of values of type T. A set can be formed of a collection of elements with no repetitions and with no order; the `{ }` expression denotes the empty set. Some examples of set expressions of type `set <int>` are: `{2, 7, 5, 3}`, `{4+2, 1+5, a*b}`, where a and b are variables of type *int*.

Another type is the *multiset*, whose behaviour is similar to *sets* but allowing repetitions. Multisets are specified in similar ways to sets, but preceded by the word *multiset*. Example of multiset expressions of type `multiset<int>` are: `multiset{ }`, `multiset {0,1,1,2,3,5}` and `multiset{ 4+2, 1+5, a*b}`, where a and b are variables of type *int*.

Despite all of the above, what we are most going to use in this project is the *sequence* type. For any type T, a value of type `seq<T>` denotes a sequence of elements of type T. In this case, it is ordered, so each element has an index. To illustrate this: `[ ]`, `[0,1,1,2,3,5]`, `[4+2, 1+5, a*b]` are sequence expressions of type `seq<int>`, where a and b are variables of type *int*. We can extract elements from the sequence using indices and slices: if `s = [0,1,1,2,3,5]` then `s[0] = 0` (Dafny takes the first position as the index 0) and `s[1..4] = [1,1,2]` (the element at position 1 is included but not the one at position 4). In addition, there is a notation for extracting the length of a sequence; this is $|s|$. Taking the sequence of our example, we know that $|s| = 6$. We should mention a way to construct a sequence of values using sequence comprehension. The following syntax `seq(k, n ⇒ n+1)` constructs a sequence of k elements whose values are obtained by evaluating the function on the sequence of indices (0,1,2,...). For example, `seq(3, n ⇒ n+2) = [2,3,4]`, because for element in position 0 it adds 2 and so on for the rest. We can also have functions that assign boolean values, for example: `seq(2, n ⇒ false) = [false, false]`.

There exist other immutable types such as *strings* and *maps*, which we are not using in this project.

Besides the *value types* explained above, Dafny offers *reference types*. In this group we

will focus on *classes* and *arrays*. These are *mutable* types.

A *class* C is a reference type, which must be declared as:

```
class C<T> extends J
{
    members
}
```

We must mention that <T> is an optional list of type parameters, so the classes of Dafny can be parametric.

The members of a class can be fields, functions, predicates, methods... These are invoked by referring first to a C instance. Ghost fields represent the abstract model of an instance of a class. This model is embodied in the concrete fields of the class, which are not ghost. A class usually has a *Valid* predicate that expresses the representation invariant of the class. This invariant can place restrictions on the values of the concrete attributes, and relates them to the abstract model.

Example 3. An example of a class with a *Valid* predicate is the following, a class for representing rectangles. This class has two variables, one with the value of each side, a predicate *Valid*, and two different methods. This time, the predicate *Valid* ensures that both variables, *side1* and *side2*, are greater or equal to 0, as the sides of a rectangle cannot be negative numbers. Then, we have two methods that calculate the perimeter and the area of a rectangle.

```
class Rectangle{
    var side1: int
    var side2: int

    predicate Valid()
    reads this
    {
        side1 ≥ 0 ∧ side2 ≥ 0
    }

    method perimeter() returns (p: int)
    requires Valid()
    ensures p ≥ 0
    {
        p := 2 * side1 + 2 * side2;
    }

    method area() returns (p: int)
    requires Valid()
    ensures p ≥ 0
    {
        p := side1 * side2;
    }
}
```

□

Array is another kind of *mutable* type. Dafny supports fixed-length array types with any

positive dimension. *Arrays* are reference types; that is, they are allocated in the heap. An array can be created as follows:

```
var a : array<T> := new T[n]
```

where T is the type of the elements of the array. To obtain the *length* of array a, we just have to write `a.Length` which returns a natural number. As in *sequences*, to obtain an element of the array we just have to use indices. However, we can have a subarray converted to a sequence, or the whole array converted to a sequence. If a is of type `array<T>`, the expression `a[i..j]` evaluates to a sequence of type `seq<T>` that contains the elements of a from index i (included) to index j (not included). If both indices are omitted, the resulting sequence contains all the elements of the array.

Let us see an example of this:

```
var a: array<int> := new int[4] [6, -5, 1, 3];
var s: seq<int> := a[1..3];
assert s = [-5, 1];
s := a[..];
assert s = [6, -5, 1, 3];
```

Before finishing this section, let us introduce user-defined immutable *datatypes*. There are different kinds of algebraic datatypes. We are going to focus on the inductive ones. A good example will be the standard datatype *List*, which has the form:

```
datatype List<T> = Nil | Cons(head: T, tail: List<T>)
```

where the first alternative is the empty list and the second one is the constructor of non-empty lists.

In the same way *tuples* can be declared. For example, a pair can be declared as:

```
datatype Pair<T,U> = Pair(0: T, 1: U)
```

Showing the use of a tuple destructor, here is a property that holds:

```
assert (5, true).1 = true
```

where the notation `(x,y)` is equivalent to `Pair(x,y)`.

2.4 Frames

Frame expressions are used in Dafny to denote the set of heap memory locations that a Dafny program element may read or write [4]. Note that framing only applies to heap allocated data, so it can only be applied to reference types, but not to immutable values, such as sequences, for example. We are going to focus on *reads* and *modifies* clauses.

The *reads frame* of a function or predicate is the set of all memory locations that a function is allowed to read. Framing is essential for verification when arrays are involved. Suppose we have a function f that has a *reads* x clause, where x is a parameter. When we apply that function several times throughout the code with the same argument, we know that several occurrences of the application $f(x)$ always evaluate to the same value as long as the value of x does not change between them. If more than one *reads* clause is given in a specification, the read set is the union of the sets specified. If there are no *reads* clauses the read set is assumed to be empty. If * is given in a *reads* clause it means any reference may be read.

Example 4. *Appears* is an example of a predicate which needs a *reads* clause for the array given as input parameter. Without it, Dafny would not accept the predicate definition.

```
predicate appears(v: array<int>, x:int)
reads v
{
    exists i :: 0 ≤ i < v.Length ∧ v[i] = x
}
```

□

Methods do not need *reads clause* as they are allowed to read any memory location reachable from their parameters but they do need to specify which parts of memory they may modify via the *modifies* clause. Better said, rather than identifying what is going to be changed, it serves to indicate what is not going to be changed for sure. They are almost identical to *reads* clauses, except they specify what could be changed; and they are also needed for verification. If more than one *modifies* clause is given in a specification, the modifies set is the union of the sets. If no *modifies* clause is given the modifies set is empty. A loop can also have a *modifies* clause. If none is given, the loop may modify anything the enclosing context is allowed to modify.

Example 5. Assume the following method declaration:

```
method ModifyArray(a: array<int>)
  modifies a
```

We call this method and check its effects on already existing arrays:

```
var c: array<int> := new int[3] [0, 0, 0];
var b: array<int> := new int[3] [0, 0, 0];

assert c[0] = 0;
assert b[0] = 0;

ModifyArray(c);

assert b[0] = 0; // The modifies clause of ModifyArray ensures that b is not modified
assert c[0] = 0; // Assertion violation: as ModifyArray can modify parameter c,
// We cannot ensure that it has the same value as before calling the method.
```

□

Another command we can use is *old*. This is used due to the fact that the postcondition may be expressed in terms of the state of the variables before and after method execution. Dafny allows this by making use of the *old* keyword, which when applied to a variable (*old(variable)*) refers to the value of the variable at the time the method was invoked. This is shown in the following example:

```
method AddOne(a: array<int>)
modifies a
ensures ∀ i | 0 ≤ i < a.Length :: a[i] = old(a[i]) + 1
```

In this *method*, we just have to modify the array a by adding one to each element. We do need to use the *modifies* clause, and to refer to the array elements at the time the method was invoked we use `old(a[i])`.

2.5 Class Types

We have already introduced *classes* in Dafny. Classes offer a way to dynamically allocate mutable data structures [10]; and, as we mentioned before, they contain two types of fields: physical fields and ghost fields, the latter of which represent the abstract model of an instance of a class. These two are related most of the times thanks to the representation invariant, specified by means of a predicate named *Valid*. This predicate is satisfied when an instance of the class is constructed, and its validity should be maintained before and after calling each of the methods of the class. Essentially, the *Valid* predicate ensures that the target instance is in a consistent state. *Constructor* is another component that we are going to find in most of our classes. It is a method that is executed when an instance of a class is created, and is used to initialise the fields of the class.

Example 6. Let us see an example from [6] which will help us relate all these concepts. The idea is to implement a queue with two stacks, here expressed as the *tail* and *head* sequences, which elements are only added or removed at the left end (top of the stack). The elements are added to the queue by stacking them on *tail* and removed from the queue by unstacking them from *head*. If *head* is empty, *tail* is inverted on top of *head*. The method *enqueue* will manage the process of adding data and the method *dequeue* will remove the data. The idea is that the data item stored first will be accessed first.

```
class Queue{
    var tail: seq<int>;
    var head: seq<int>;

    ghost var contents: seq<int>;

    function reverse(s:seq<int>): seq<int>
    decreases s
    ensures |s| = |reverse(s)|
    ensures ∀ i :: 0 < i < |s| ⟹ s[i] = reverse(s)[|s|-1-i]
    {
```

```
        if s = [] then []
        else reverse(s[1..]) + s[0..1]
    }

    predicate Valid()
    reads this
    {
        contents = head + reverse(tail)
    }

    constructor ()
    ensures Valid()
    ensures contents = []
    {
        head := [];
        tail := [];
        contents := [];
    }

    method enqueue(e: int)
    modifies this
    requires Valid()
    ensures contents = old(contents) + [e]
    ensures Valid()
    {
        tail := [e] + tail;
        contents := contents + [e];
    }

    method dequeue() returns (e: int)
    modifies this
    requires Valid()
    requires contents ≠ []
    ensures contents = old(contents)[1..]
    ensures e = old(contents)[0]
    ensures Valid()
    {
        if head = [] {
            moveFromIncoming();
        }
        e := head[0];
        head := head[1..];

        contents := contents[1..];
    }

    method moveFromIncoming()
    modifies this
    requires Valid()
    requires Valid()
    requires head = []
    ensures tail = []
    ensures head = reverse(old(tail))
    ensures Valid()
    {
        while tail ≠ []
            decreases |tail|
            invariant reverse(tail) + head = reverse(old(tail))
            {
                head := tail[0..1] + head;
                tail := tail[1..];
            }
        contents := head;
    }
}

method Main()
{
    var q := new Queue();
```

```
    assert q.contents = [];

    q.enqueue(1);
    assert q.contents = [1];

    q.enqueue(2);
    q.enqueue(3);
    q.enqueue(2);
    q.enqueue(4);
    assert q.contents = [1,2,3,2,4];

    var f := q.dequeue();
    assert f = 1 ∧ q.contents = [2,3,2,4];

    q.enqueue(5); assert q.contents = [2,3,2,4,5];
}
```

We have declared a class named *Queue*. To implement this class we have used two physical variables, *head* and *tail*, and one ghost variable, *contents*. As we have mentioned before, it is necessary to define a *Valid* predicate, which ensures our ghost variable *contents* is the union of the sequence *head* and the reverse of the sequence *tail*, for this we need an extra function that returns the reverse of a sequence. The *constructor* just initializes an instance of the class, where all the sequences, *head*, *tail* and *contents*, are empty.

In the example there are two methods: *Enqueue* and *Dequeue*, apart from *moveFromIncoming*, which we needed for *Dequeue*. With the preconditions and postconditions we are able to see what is the purpose of these methods. The first one adds the elements that receives as input to the *queue*, the reason why there is no output is because it just modifies the *queue* but it does not have to return anything. *Dequeue* deletes from the *queue* the element in the first position (index 0) and returns it as result.

Finally, note that the example has a *Main* method that can be used as a tester. First of all, it creates a new instance of the class, using the *constructor*. Then it calls *enqueue* a few times and we can observe the result in the following *assert*. This is the main function of the ghost variable, it lets us see the abstract model of the *queue*. And finally we have another example to see how *dequeue* works.

□

2.6 Modules

Let us finish this chapter introducing our last concept, which we used to improve our final project. When creating large programs, dividing it into parts helps structuring the program. This is what modules are used for. They help us with grouping together the elements (types, classes, methods, functions, predicates...) of our programs that are related in some way. Modules let us import their definitions with the purpose of reusing the code.

A new module is declared in Dafny [5] just by using the word *module*, followed by its

name, with the body of the module inside {}; similar to so many more structures we have already seen. In the body of the module we can find classes, datatypes, methods, lemmas, functions, etc. We can even have a module inside another module (something we did not need to do in our project), or having abstract modules, which we will explain later as it is an important part of our code.

All the elements defined inside a module are available just like in classes, with just the module name prefixing them. In case we are interested in using elements that we have inside *module A* in another module, *module B*, we do it via the *import* keyword. However we are more interested in *module abstraction*. It allows us to define different concepts with their own properties; but with no implementation. From these *abstract modules* we obtain *refinements* of the module. A refinement of a module is another module with all the properties, definitions, methods or lemmas from the first one applied to particular types and particular implementations.

Example 7. Let us see an example to illustrate these ideas. In group theory, a group is a set G that has an associated operation such that the following properties are satisfied: associativity, existence of a neutral element and existence of an inverse element. All of these properties are represented in an *abstract module* in Dafny, called *Group*:

```
abstract module Group {
    // The type which represent the elements of the group
    type G

    // An operation
    function Product(x: G, y: G): G

    // Associativity Lemma, without proof
    lemma Associativity(x: G, y: G, z: G)
    ensures Product(x, Product(y, z)) = Product(Product(x, y), z)

    // Existence of an neutral element
    function Identity(): G

    lemma IdentityIsNeutral(x: G)
    ensures Product(x, Identity()) = x
    ensures Product(Identity(), x) = x

    // Existence of an inverse element
    function Inverse(x: G): G

    lemma ProductInverse(x: G)
    ensures Product(x, Inverse(x)) = Identity()
    ensures Product(Inverse(x), x) = Identity()

    // The following lemma follows from the properties given above, and holds
    // regardless of the set G, so we can prove it here.
    // Every refinement module will inherit this lemma.

    lemma SimplifyLeft(a: G, b: G, c: G)
    requires Product(a, b) = Product(a, c)
    ensures b = c
    {
        calc {
                Product(a, b) = Product(a, c);
            ==>
                Product(Inverse(a), Product(a, b)) = Product(Inverse(a), Product(a, c));
            ==> { Associativity(Inverse(a), a, b); Associativity(Inverse(a), a, c); }
                Product(Product(Inverse(a), a), b) = Product(Product(Inverse(a), a), c);
```

```
            ==> { ProductInverse(a); }
                Product(Identity(), b) = Product(Identity(), c);
            ==> { IdentityIsNeutral(b); IdentityIsNeutral(c); }
                b = c;
        }
    }
}
```

Once we have the *abstract module*, *refinements* can be obtained, which are merely instances of the theory applied to specific groups and specific operations. For example, we can represent the group (Z, +) in Dafny via the *AdditiveInt* module:

```
module AdditiveInt refines Group {
    type G = int

    function Product(x: G, y: G): G {
        x + y
    }

    lemma Associativity(x: G, y: G, z: G)
    ensures Product(x, Product(y, z)) = Product(Product(x, y), z)
    {}

    function Identity(): G {
        0
    }

    lemma IdentityIsNeutral(x: G)
    ensures Product(x, Identity()) = x
    ensures Product(Identity(), x) = x
    {}

    function Inverse(x: G): G {
        -x
    }

    lemma ProductInverse(x: G)
    ensures Product(x, Inverse(x)) = Identity()
    ensures Product(Inverse(x), x) = Identity()
    {}
}
```

□

The proofs of the *Associativity*, *IdentityIsNeutral* and *ProductInverse* lemmas are empty because Dafny can prove them automatically. The lemma *SimplifyLeft* is inherited from the abstract module.

Chapter 3

Greedy Algorithms

In this chapter we will introduce greedy algorithms. The greedy method is an algorithmic scheme that can be used to solve some optimization problems, so it is essential to prove that greedy algorithms find an optimal solution when proving their correctness. There are several ways of proving this correctness, but we will dedicate one section of this chapter to focus on the technique we have used. At the same time we will gradually present a concrete problem to be solved with a greedy algorithm and prove that we do indeed obtain an optimal solution.

3.1 Method's description

The greedy method is an algorithmic scheme used to solve optimization problems that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit [2]. So, the problems where choosing the best option in every moment leads to an optimal solution are the ones that fit best for the greedy method.

Sometimes greedy strategies fail to find the optimal solution because their decisions are restricted by the ones they have already taken; this means, the choice made by a greedy algorithm depends on the choices it has already made. Let us see an example of this in Figure 3.1.

If we are trying to find the longest path (the number besides each edge represents its length), it seems logical that a greedy algorithm chooses in the first case the edge with 12 instead the one with 3. This is because the greedy algorithm is just considering the edges going out the first node, and it does not think about the rest of the edges. The globally optimal solution is not a path that starts like $12 \rightarrow ...$, it will be the path: $3 \rightarrow 1 \rightarrow 99$. This is clear to us because when taking a decision we consider all combination of edges. This is a case of a problem that cannot be solved by a greedy algorithm. In this example, a locally

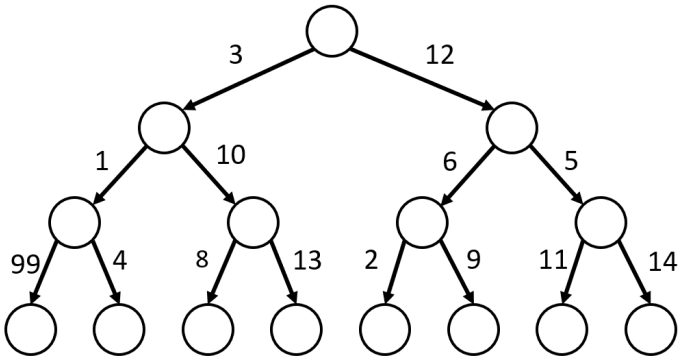

Figure 3.1: Find the longest path in this graph

optimal solution does not lead to a global optimal solution.

If an optimal solution can be found by taking the best choice at each step without reconsidering the decisions previous to that step, then we say the problem can be solved using a greedy algorithm. This property is called *greedy choice property*. If there is a greedy strategy that solves the problem, then it holds that the optimal solution to the problem contains optimal solutions to its subproblems. This property is called *optimal substructure* [8]. The greedy method is quite powerful and works well for a wide range of problems.

Greedy algorithms have some advantages and disadvantages:

- Advantage: they are often efficient and simple to implement algorithms.
- Disadvantage: the difficult part is that the correctness proof of greedy algorithms is more involved. It is not only necessary to prove that the algorithm returns a correct solution, but also that it is optimal.

There exist different ways to prove that a greedy algorithm is correct; however, all of them have some things in common. First of all, we need to prove that the algorithm returns a solution that meets all the constraints. For example, something as simple as checking that the type of the solution is correct, or that if the algorithm must return a sorted array, verifying that this is the case. Next, we must ensure that the algorithm returns an optimal solution, that is, a solution that optimizes an objective function.

Example 1. At this point we can present our running example in this chapter: the *files problem*. The idea is that we have a collection of n files, where each of them takes up a certain space s_i, and we have a computer or an electronic device with limited space s. The problem consists in choosing the maximum number of files that can be stored in the computer/device. The greedy strategy we will follow involves sorting the files from smallest to greatest, which

does not place any special constraints on the problem, and taking the files in ascending order of size until there is no more room for them. This way we will store in the computer/device as many files as possible. Then, the solution will be a sequence of booleans, where the boolean in position i represents if we store the i-th file or not. □

3.2 Correctness Proof

To prove correctness we have chosen a strategy called *exchange proof*, or *greedy exchange*. The idea for this proof is to compare two different solutions: an optimal solution and a greedy solution. Let us assume that $A = [a_1, a_2, ..., a_m]$ is our greedy solution obtained by the algorithm, and $\theta = [o_1, o_2, ..., o_k]$ is an optimal solution. We assume that our arbitrary optimal solution is not equal to our greedy solution, otherwise we would have proved that the greedy solution is also optimal. Then, we assume that up to but not including some position i, where $0 \leq i \leq m$ and $0 \leq i \leq k$, both solutions are identical and that $a_i \neq o_i$, i.e. i is the first index where they are different. We transform the optimal solution into another optimal solution θ' such that θ' and A are equal up to and including i. We have to repeat this process until all the positions are equal. If we are able to achieve this, then our greedy solution is optimal.

Let us look at an example to help us understand how the *exchange proof* works. We will do it with the *files problem*.

Example 2. Assume that we have our greedy solution A and an optimal solution θ; and we can assume that both solutions are equal up to, but not including, some position i, i.e. i is the first index where they are different. Let us look at the possible situations that we may encounter: $a_i = \mathit{false} \neq \mathit{true} = o_i$ or $a_i = \mathit{true} \neq \mathit{false} = o_i$.

1. When a_i is equal to *false* and o_i is equal to *true*, we know by the definition of greedy solution that for all positions j after i it is verified that $a_j = \mathit{false}$. Then, regardless the remaining positions of the optimal solution, θ has at least one more *true* value than A. This situation contradicts the definition of the greedy solution; in particular, it contradicts the property which ensures that if a file has not been stored in our greedy solution it is because there is no space to store it with the previously chosen. Given that in the case of the optimal solution it is possible to store file i, we know that in the greedy solution it is also possible to store that file and, therefore, this last mentioned property is not fulfilled. Consequently, this case is not possible.

2. When a_i is equal to *true* and o_i is equal to *false*, there must exist a position j after i in our optimal solution such that $o_j = \mathit{true}$. Otherwise it would contradict an essential property of an optimal solution: it must store at least the same number of files as the rest of solutions. Then, if there exists a position $j > i$ such that $o_j = \mathit{true}$, that is, $\theta = [o_1, ..., o_{i-1}, \mathit{false}, ..., o_{j-1}, \mathit{true}, ..., o_m]$; we can exchange positions i and j as follows: $\theta' = [o_1, .., o_{i-1}, \mathit{true}, ..., o_{j-1}, \mathit{false}, ..., o_m]$. This is allowed because files are

sorted by size, then the size of the file in position i is smaller than the size of the file in position j, then we can leave out file in j and store file in i.

This way we have achieved that our both solutions are equal up to, and including position, i; i.e. $A[..i+1] = \theta'[..i+1]$, and we repeat the process, so we have to compare the following positions until we find another two that are not equal. As the number of elements in the sequence is finite, we can apply this process that tries to match θ' with A without losing the optimality of θ', so that, at the end of the process, the optimal and the greedy solutions coincide, which proves the optimality of the greedy solution.

□

3.3 Other ways of proving optimality

There are other ways of proving optimality that we have not seen, as proving directly that the greedy solution is better than any other and is therefore optimal, or by induction.

Example 3. One famous problem that can be solved with a greedy algorithm and that can be proved applying the first way we have mentioned is the *Knapsack* problem. The idea is that there are n objects, each of them with a specific weight $p_i > 0$ and value $v_i > 0$. The objects can be broken and the knapsack has a maximum total weight $M > 0$. Our goal is to maximize the sum of the values of all the objects in the knapsack; what means that our objective function is

$$\sum_{i=1}^{n} x_i * v_i$$

where x_i represents the fraction of the object i that we are going to put inside the knapsack.

The strategy followed in this algorithm is similar to the one in the *files problem*. We have the objects decreasingly sorted by their value per weight unit, and we check for each object whether we can fit it inside the backpack in its entirety or whether we can only fit the fraction that fills the backpack. Once we have reached the maximum weight of the backpack, the rest of the objects are not put into the backpack.

The way of proving optimality here is the following: we assume $X = (x_1, ..., x_n)$ is our solution obtained from the greedy algorithm, where objects are decreasingly sorted by their value per weight unit and x_i can take values from 0 to 1. If all x_i are equal to 1, the solution is clearly optimal. Then let us say j is the smallest index where $x_j \neq 1$. Then $x_i = 1$ for all $1 \leq i < j$ and $x_i = 0$ for all $j < i \leq n$.

It can be proved that $\sum x_i v_i - \sum y_i v_i \geq 0$ for any other solution $Y = (y_1, ..., y_n)$. For that reason, X must be an optimal solution. The detailed proof is not included for the shake of brevity, but it can be found in [15]. We should mention that the proof of this problem can be

also done applying *exchange proof*, but we wanted to explain this other way of demonstrating optimality. □

Example 4. Now we are going to see another way of proving optimality. In second place, we mentioned the proof by induction. *Kruskal's algorithm* is a greedy algorithm that can be proved applying this technique. It is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

1. forms a tree that includes every vertex (i.e. a spanning tree)
2. has the minimum sum of weights among all spanning the trees that can be formed from the graph

The idea of this algorithm is to consider edges in order of increasing weight and add them to the tree if they do not create a cycle with those already selected. The invariant of the algorithm is that the set of all the edges is promising, which means that it can be extended to a minimum spanning tree. The proof of the correctness of this algorithm is based on induction on the number of edges. The base case is where our tree is empty and the induction hypothesis assumes that our tree T is promising before adding an edge a. Then we have to prove that $T \cup \{a\}$ is also promising. The detailed procedure for this proof can be found at [15]. □

There are more greedy algorithms that solve problems on graphs, such as *Prim's* and *Dijkstra's* algorithms. The correctness proof in these problems can be done by applying induction on the number of edges, as we just show, or in the number of steps of the algorithm. Detailed induction proofs can be found in [15]. Apart from proofs by induction on the number of edges, we can also guarantee the optimality of these algorithms in other ways, for example with proofs based on special properties, such as in the case of the algorithm of *Dijkstra's* algorithms where we can base the correctness proof on the *upper-bound* property. A detailed proof can be found in [3].

3.4 Verification Example: Implementation in Dafny

In this section we are going to verify in Dafny the greedy algorithm that solves the *files problem*.

3.4.1 Predicates and Algorithm

First of all, we have to establish the input and output types of our algorithm. As we have mentioned before, we have two input parameters: first, the different values representing the

space each file takes up; and the space available in the computer/device. The space values are saved in an array called *s_files*, where each element must have type *nat*; and it seems obvious that the length of the array tells us the number of files we have. The space available at the computer will be stored in a natural variable, called *s_disc*. Considering the output parameter, the solution will be also an array of the same length as the input, but this time the elements are going to be booleans which indicate if the concrete file has been stored or not. The header of our method will be:

```
method file_in_disc(s_files: array<nat>, s_disc: nat) returns (sol: array<bool>)
```

However, in the verification part we are going to use sequences instead of arrays, so the input parameter *s_files* and the output parameters *sol* will be sequences of naturals and booleans, respectively. These sequences represent the values of these arrays at each point of execution, and it is on these values that the different properties are defined. The reason why we use arrays in the implementation and sequences in the verification part is because arrays are mutable, while sequences are not. So, the array-based implementation is closer to what one would do in an imperative programming language, while the use of immutable sequences is more suitable for reasoning. Later in the next chapter we will see how to establish a relation between these two types, which ensures that the properties fulfilled by the immutable types are also fulfilled by the corresponding mutable types.

We should also mention the possible restrictions we can plan on the input parameters. We could differentiate these constraints into two types. Firstly, those that do not restrict, but impose an input format; and secondly, those that guarantee the existence of a solution. In the *files problem*, we will just find restrictions of the first type. As we have mentioned before, we will require that the *s_files* array (or sequence) is increasingly sorted. Existence of solution is always guaranteed: that one in which no file is stored.

One of the most important parts of the code is to define all the different types of solutions. This was made using predicates. Let us see, for example, the definition of being a solution. A solution to this problem must verify that it has the same length as *s_files* and that the sum of all the sizes of the files we have chosen to store is not greater than the maximum space available. The first condition must be verified, because for each file we need to know whether it is stored or not; then if the solution does not have the same length as the number of files, it cannot be a possible solution. The second condition is obvious, we cannot store more space that the one we have available in the device. This was represented in Dafny the following way:

```
// Given a vector with the size of the files (sorted from the smallest to the biggest),
// the size of the disk and a solution. A solution has to ensure that the sum
// of all the files that we choose is smaller or equal to the size of the disk.

predicate isSolution(s_files: seq<nat>, s_disc: nat, sol: seq<bool>)
requires sorted(s_files)
{
    |sol| = |s_files| ∧ sum(s_files, sol) ≤ s_disc
}
```

There is a *requires* clause we have not mentioned. This precondition ensures that the

sequence *s_files* is sorted from the smallest to the biggest file; this is the restriction we have talked about before and it will help us later in the algorithm. Besides that, in this predicate we see represented the two previously mentioned properties that any solution must meet. It is necessary the use of a function *sum*, which adds the sizes of files we have decided to store.

```
function sum(a:seq<nat>, b:seq<bool>):nat
requires |a| = |b| ≥ 0
{
    if (|a|= 0) then 0
    else if b[|b|−1] then sum(a[..|a|−1], b[..|b|−1]) + a[|a|−1]
         else sum(a[..|a|−1], b[..|b|−1])
}
```

Knowing the definition for solution, the next step is to define when one solution is better than another. This is defined by a predicate named *isBetterSol* and, as *isSolution*, it receives as inputs *s_files*, *s_disc* and two different solutions. With regard to the specification part, we must write as preconditions that both sequence of booleans we received as input are solutions, in other words, they must verify *isSolution*. Regarding the code, it is easy to see that if the number of files we have stored in s_1 is greater or equal to the number of files we have stored in s_2, then s_1 is better solution than s_2. For this, we need a function *value* that returns the number of files we have stored in a specific solution, what in reality means counting the number of *true* values in a sequence.

```
function value(sol: seq<bool>): nat
{
 if (|sol|= 0) then 0
    else if sol[|sol|−1] then value(sol[..|sol|−1]) + 1
         else value(sol[..|sol|−1])
}

predicate isBetterSol(s_files:seq<nat>, s_disc: nat, betterSol:seq<bool>, sol:seq<bool>)
requires sorted(s_files)
requires isSolution(s_files, s_disc, sol)
requires isSolution(s_files, s_disc, betterSol)
{
  value(sol) ≤ value(betterSol)
}
```

The other two important predicates we need are *isSolOptimal* and *isSolGreedy*. In both predicates, as inputs we have our parameters *s_files* and *s_disc* (as always), and a sequence of booleans named *sol*. So, as preconditions we must ensure that *sol* is a solution, and we will study if it is optimal or greedy solution (depending on the predicate). An optimal solution must be defined as a solution that is better than the rest of possible solutions; in other words, for all possible solutions the optimal one is always better or equal.

```
predicate isSolOptimal(s_files:seq<nat>, s_disc:nat, optimalSol:seq<bool>)
requires sorted(s_files)
requires isSolution(s_files, s_disc, optimalSol)
{
    ∀ sol | isSolution(s_files,s_disc,sol) :: isBetterSol(s_files,s_disc,optimalSol,sol)
}
```

This definition is the same for any optimization problem.

On the other hand, the properties that the greedy solution must verify change depending

on the problem, so we must adjust it to the *files problem*. We must ensure two conditions for a solution to be a greedy solution:

1. for all positions i and j, where $i < j$, if $sol[i] =$ false then $sol[j] =$ false. It would not make any sense to not store file i but store file j knowing that $s\_files[i] \leq s\_files[j]$.
2. for all files we have decided not to store, it is because it is not possible to store them.

This is represented in Dafny as follows:

```
predicate isSolGreedy(s_files:seq<nat>, s_disc:nat, sol:seq<bool>)
requires sorted(s_files)
requires isSolution(s_files, s_disc, sol)
{
    (∀ i, j | 0 ≤ i < j < |sol| :: (sol[i] = false ⟹ sol[j] = false)) ∧
    ∀ m | 0 ≤ m < |sol| ∧ sol[m] = false :: (sum(s_files, sol[m := true]) > s_disc)
}
```

Let us study now the algorithm. The idea is the following: we receive as entry two parameters, *s_files* and *s_disc*. We are going to have a loop which traverses our sequence and we will have to decide to store each file or not. As *s_files* is sorted, something we must specify as precondition, once we cannot store file i, the following files (files $i+1$ to file n) cannot be stored either. We will declare a local variable *accum*, which represents the amount of space we have already taken up from the computer/device. It is a variable that will help us know how much space we have left. As a condition in our loop, we have to ensure that $i < n$ and that it is possible to store the file. In case it is, we must update variables i and *accum*, and set $sol[i] =$ true. In case it cannot be stored, the first loop stops and another loop sets all the positions left to false. At the end of the algorithm it must verified that our variable *sol* is a greedy solution; something we specify in the postconditions.

```
method file_in_disc(s_files:array<nat>, s_disc:nat) returns (sol:array<bool>)
requires sorted(s_files[..])
ensures sol.Length = s_files.Length
ensures isSolution(s_files[..], s_disc, sol[..])
ensures isSolGreedy(s_files[..], s_disc, sol[..])
{
    var accum := 0;
    var i := 0;
    sol := new bool[s_files.Length];

    while (i < s_files.Length ∧ (accum + s_files[i] ≤ s_disc))
      decreases s_files.Length − i
      invariant 0 ≤ i ≤ s_files.Length
      invariant accum ≤ s_disc
      invariant accum = sum(s_files[..i], sol[..i])
      invariant isSolution(s_files[..i], s_disc, sol[..i])
      invariant isSolGreedy(s_files[..i], s_disc, sol[..i])
      {
        sol[i] := true;

        accum := accum + s_files[i];

        assert s_files[..i + 1][..i] = s_files[..i];
        assert sol[..i + 1][..i] = sol[..i];

        i:= i+1;
      }
```

```
    while (i < s_files.Length)
      decreases s_files.Length - i
      invariant 0 ≤ i ≤ s_files.Length
      invariant accum = sum(s_files[..i], sol[..i])
      invariant isSolGreedy(s_files[..i], s_disc, sol[..i])
      {
        sol[i] := false;

        assert s_files[..i + 1][..i] = s_files[..i];
        assert sol[..i + 1][..i] = sol[..i];

        i := i + 1;
      }
    assert s_files[..] = s_files[..s_files.Length];
    assert sol[..] = sol[..s_files.Length];

}
```

Once the code is ready, we must ensure that Dafny can prove the postconditions. For this, we need to add some invariants and assertions, that help Dafny with the verification. For example, in both loops there are invariants which ensure that the solution we are creating up to the current index is indeed a greedy solution. Considering the *accum* variable, there is another invariant which ensures that, during the loop, this variable represents the amount of space occupied by the selected files, and therefore *accum* must be smaller or equal than the available space, which is represented in another invariant. There are also one decreases clause for each loop that lets Dafny know that the loop finishes.

3.4.2 Proving correctness

Now we are going to prove that the greedy solution is optimal. To that purpose, we create a method that given two different solutions, one greedy and one optimal, it ensures that the greedy is also optimal. We apply the *exchange proof* we have studied before. So, the idea is to compare both solutions in a loop and in the following position i where $greedy[i] \neq optimal[i]$, we must study the situation. We will do that in a new method called *reduction_step*, and that method will return *optimal'*, another optimal solution which verifies that $optimal'[0..i+1] = greedy[0..i+1]$.

Assuming that *reduction_step* has already been defined, the following method reduction specifies the whole optimality proof by applying zero or more reduction steps until the greedy and optimal solutions become equal.

```
ghost method reduction(s_files:seq<nat>, s_disc:nat, greedy:seq<bool>, optimal:seq<bool>)
requires sorted(s_files)
requires isSolution(s_files, s_disc, greedy)
requires isSolution(s_files, s_disc, optimal)
requires isSolGreedy(s_files, s_disc, greedy)
requires isSolOptimal(s_files, s_disc, optimal)
ensures isSolOptimal(s_files,s_disc,greedy)
{
    var i := 0;
    var optimal' := optimal;
```

```
    while (i < |optimal|)
     invariant 0 ≤ i ≤ |greedy|
     invariant isSolution(s_files, s_disc, optimal')
     invariant isBetterSol(s_files, s_disc, optimal', optimal)
     invariant optimal'[..i] = greedy[..i]
    {
        if (greedy[i] ≠ optimal'[i])
        {
            optimal' := reduction_step(s_files, s_disc, greedy, optimal', i);
        }
        i := i+1;
    }

   endReduction(s_files, s_disc, greedy, optimal, optimal');
   betterThanOptimal(s_files, s_disc, greedy, optimal);
}
```

Lemmas *endReduction* and *betterThanOptimal* help us with the verification of *reduction*. The first one proves that, after the loop, *greedy* and *optimal'* are equal. Dafny can prove it without help. The second one proves that if *greedy* is better solution than *optimal*, which is true from the third invariant and the postcondition of *endReduction*, then *greedy* is an optimal solution, our main goal to prove. This second lemma is also proven without help.

```
lemma endReduction(s_files:seq<nat>, s_disc:nat, greedy:seq<bool>,
  optimal:seq<bool>, optimal':seq<bool>)
requires sorted(s_files)
requires isSolution(s_files,s_disc,greedy) ∧ isSolGreedy(s_files,s_disc,greedy)
requires isSolution(s_files,s_disc,optimal) ∧ isSolOptimal(s_files,s_disc,optimal)
requires isSolution(s_files,s_disc,optimal') ∧ isSolOptimal(s_files,s_disc,optimal')
requires isBetterSol(s_files,s_disc,optimal',optimal)
requires optimal'[..|optimal|] = greedy[..|optimal|]
ensures optimal' = greedy
{}

lemma betterThanOptimal(s_files:seq<nat>, s_disc:nat, sol:seq<bool>, optimalSol:seq<bool>)
requires sorted(s_files)
requires isSolution(s_files, s_disc, sol)
requires isSolution(s_files, s_disc, optimalSol) ∧ isSolOptimal(s_files, s_disc, optimalSol)
requires isBetterSol(s_files,s_disc,sol,optimalSol)
ensures isSolOptimal(s_files,s_disc,sol)
{}
```

Let us focus now on *reduction_step*. Assuming that $greedy[i] \neq optimal[i]$, the purpose for this method is to build *optimal'* so it verifies that $greedy[i] = optimal'[i]$. We must verify that $greedy[..i+1] = optimal'[..i+1]$, that it keeps being an optimal solution and that it is better solution than *optimal*. As we have seen in the previous section, where we explained the *exchange proof* from a more technical point of view, there are two possible situations: $greedy[i] = false$ (and $optimal[i] = true$) or $greedy[i] = true$ (and $optimal[i] = false$).

1. The first case is not possible, and we have to prove so by contradiction.

2. In the second case, where $greedy[i] = true$ and $optimal[i] = false$, there must be an index in the vector $optimal[i..]$ where $optimal[j] = true$, that is what the *existTrue* lemma ensures. Knowing the value of j, we must swap $optimal[i]$ with $optimal[j]$; we know we can do that because we change a file with greater size for another one with a smaller size. So, if the bigger one fits in the computer, then so does the smaller

one. That is the aim of method *swap*, to exchange those two positions in the optimal solution.

```
ghost method reduction_step(s_files:seq<nat>, s_disc:nat, greedy:seq<bool>,
                            optimal:seq<bool>, i:nat) returns (optimal':seq<bool>)
requires sorted(s_files)
requires isSolution(s_files, s_disc, greedy) ∧ isSolution(s_files, s_disc, optimal)
requires isSolGreedy(s_files, s_disc, greedy) ∧ isSolOptimal(s_files, s_disc, optimal)
requires 0 ≤ i < |greedy|
requires greedy[i] ≠ optimal[i] ∧ greedy[0..i] = optimal[0..i]
ensures isSolution(s_files, s_disc, optimal')
ensures isSolOptimal(s_files, s_disc, optimal')
ensures isBetterSol(s_files, s_disc, optimal', optimal)
ensures |optimal| = |optimal'|
ensures optimal'[0..i+1] = greedy[0..i+1]
{
  if (greedy[i]=false) //∧ optimal[i]=true
   {//Impossible. It follows that 'greedy' is not greedy which is a contradiction.

    {sumFalse(s_files, greedy, i);}
    assert sum(s_files, greedy[i := true]) = sum(s_files[..i+1], greedy[i := true][..i+1]);
    assert greedy[i:=true][..i+1] = optimal[..i+1];
    assert sum(s_files[..i+1], greedy[i := true][..i+1]) = sum(s_files[..i+1], optimal[..i+1]);
    {sumSmaller(s_files, optimal, i);}
    assert sum(s_files[..i+1], optimal[..i+1]) ≤ sum(s_files, optimal) ≤ s_disc;

    assert sum(s_files, greedy[i := true]) ≤ s_disc;
    assert !isSolGreedy(s_files, s_disc, greedy);
   }
  else
   { {existTrue(s_files, s_disc, greedy, optimal, i); }
     var j :| i < j < |optimal| ∧ optimal[j] = true;
     optimal' := swap(s_files, s_disc, optimal, i, j);
   }
}
```

Let us study the first case in more detail. If $greedy[i] = false$ and $optimal[i] = true$, then our *greedy* variable would not be a greedy solution because it is possible to store file i:

```
assert sum(s_files, greedy[i := true]) ≤ s_disc;
assert !isSolGreedy(s_files, s_disc, greedy);
```

The idea is to prove that it is possible to store the file in position i, because it fits (as it did in the optimal solution), and Dafny will be able to prove that *greedy* is not a greedy solution. To prove this, we are going to use two lemmas: *sumFalse* and *sumSmaller*. The postconditions of both lemmas are the *asserts* written just after them. To get an idea of these two lemmas let us assume we have a greedy solution where there is a *false* value in position i. We will call this solution A (this is what greedy verifies in our problem); and a solution B is A[i := true].

To understand what *sumFalse* proves, let us see first that the number of *true* values in our solution B is equal to the number of *true* values in $B[..i+1]$, i.e. that there are no *true* values from position $i+1$ onwards. This is because when there is a *false* value in a greedy solution, we know that in the following positions everything is false and there are no more *true* values. Then, the sum of the sizes in B is equal to the sum of sizes in $B[..i+1]$.

Lemma *sumSmaller* proves that if C is an optimal solution, then the amount of device

space taken by a partial solution of C is smaller or equal to the space taken by C, which is quite obvious.

Let us focus now in the second case. We know that if $greedy[i] = true$ and $optimal[i] = false$, then there must exist a position j after i where $optimal[j] = true$. Otherwise, optimal would not be an optimal solution, because it would have less *true* values than our greedy solution. That is the idea we followed in the lemma *existTrue*: prove that if for all positions j after i $optimal[j] = false$, then *optimal* is not optimal; and therefore there would be a contradiction with the preconditions imposed.

```
lemma existTrue(s_files:seq<nat>, s_disc:nat, greedy:seq<bool>, optimal:seq<bool>, i:nat)
requires sorted(s_files[..])
requires isSolution(s_files, s_disc, greedy) ∧ isSolution(s_files, s_disc, optimal)
requires isSolGreedy(s_files, s_disc, greedy) ∧ isSolOptimal(s_files, s_disc, optimal)
requires 0 ≤ i < |optimal|=|greedy|
requires greedy[0..i] = optimal[0..i]
requires optimal[i] = false ∧ greedy[i] = true
ensures exists j :: i < j < |optimal| ∧ optimal[j] = true
{
  if (∀ j | i < j < |optimal| :: optimal[j] = false) {
    falseValue(optimal,i);
    // assert value(optimal) = value(optimal[..i]);
    assert greedy[..i+1][..|greedy[..i+1]|-1] = greedy[..i];
    assert value(greedy[..i+1]) = 1 + value(greedy[..i]);
    monotone(greedy,i+1);
    assert value(optimal) = value(optimal[..i]) =
      value(greedy[0..i]) < value(greedy[..i+1]);
    assert !isSolOptimal(s_files, s_disc, optimal);}
}
```

Lemma *falseValue* proves the property in the next *assert*, which is commented out. The idea is that if there exists a position i in a sequence where that position and all the following ones are *false*, then the number of *true* values in the entire sequence is equal to the number of *true* values in that same sequence but considering only from position 0 until position i, not included.

Lemma *monotone* proves something similar to the property in *falseValue*. This time we do not need any precondition and it just ensures that the number of *true* values of a sequence x is greater or equal to the number of *true* values in that same sequence x until some position i, where $0 \leq i \leq |x|$.

Once we know that the position we are looking for exists, we give it a name because we need to put it as input parameter in *swap*. We call this position j. Let us see now the method *swap*. The idea, as we have already mentioned, is to change $optimal[i]$ from false to true, and $optimal[j]$ from *true* to *false*. We keep the same number of *true* values, so the solution keeps being optimal. We know we are able to do this because we are leaving out a file with greater size than the one we have decided to store. This way *optimal* keeps being a solution.

Our lemma *sameValue* proves that the number of *true* values stays the same, and *smallerSum* proves that the sum of spaces is smaller in *v_modified* than in our solution *v*.

```
ghost method swap(s_files:seq<nat>, s_disc:nat, v:seq<bool>, i: nat, j: nat)
                  returns (v_modified:seq<bool>)
requires sorted(s_files)
```

```
requires isSolution(s_files, s_disc, v)
requires 0 ≤ i < j < |v|
requires v[i] = false ∧ v[j] = true
ensures isSolution(s_files, s_disc, v_modified)
ensures isBetterSol(s_files,s_disc,v_modified,v)//they are equal
ensures v_modified = v[i := true][j := false]
{

  v_modified := v[i := true][j := false];
  //They have the same number of true values
  sameValue(s_files,v,i,j);
  assert value(v_modified) = value(v);

  //They take up less space because s_files[i] ≤ s_files[j]
  smallerSum(s_files,v,i,j);
  assert sum(s_files, v_modified) ≤ sum(s_files,v) ≤ s_disc;
}
```

This concludes the exchange proof, which specifies that any solution that follows the greedy strategy is optimal. Since the file's method defined above produces a solution that follows a greedy strategy (as indicated by its postcondition), we can deduce that the solution returned by the file's method is optimal.

3.4.3 Motivation for next chapter

After having solved several problems similar to this one, we realised that there were certain common elements in them. In all of them we needed the same predicates, as *isSolution*, *isSolGreedy* and *isSolOptimal*, among others. But also the verification part has some common elements, as the *reduction* and *reduction_step* methods. This was the main motivation for our next chapter, in an attempt to simplify the resolution of these problems in Dafny by creating a methodology that can encompass all of them.

Chapter 4

Implementation

4.1 GreedyAlgorithm Module

After having carried out several examples of greedy algorithms with their respective specifications, codes and verification, we became aware of the common points that all greedy algorithms share and those that are specific of the particular problem. Thus, the idea is to create a "template" for these greedy algorithms. In this generic methodology the intention is to add first all the definitions that are common to all the greedy algorithms (like *isSolution*, *isBetterSol*, *isSolOptimal* and *isSolGreedy*). Some definitions, like *isSolOptimal*, are independent of the particular problem and they can be defined in an abstract module. On the contrary, those definitions that depend on the particular problem must be refined in the implementation module. The same happens with methods that are common and that we will need in all our problems; for example, *greedy*, *reduction* or *reduction_step*, shown in the previous section.

Let us focus now in this abstract module. As we have seen before, we will have different types: those for the algorithm that solves the problem and another ones that we use to reason about its optimality, where the predicates and the verification part are located. This time, for the algorithm we will have type *InputA* and type *OutputA*. To represent these types we use classes, because, in general, the types used in the concrete problems may be and/or contain mutable types such as arrays. Each class has a ghost variable *Repr* and a predicate *Valid*. The *Repr* field (or footprint) represents the set of objects that are reachable from an instance of this class and that could be affected by any method that modifies the instance. Remember that this information is essential to be able to verify programs that work with objects. *Repr* objects should be in *reads*/*modifies* clauses of the predicates/methods as appropriate. The predicate *Valid* (or representation invariant) is the property that determines which instances of the class denote a valid value. Therefore this predicate must be able to read the object itself and its footprint.

The predicate *Valid* in the class *OutputA* has as input parameter an element of type *InputA*, which means that the representation invariant of an object of the class *OutputA* is defined in terms of an object of class *InputA* and therefore also has to read that object and its representation. The greedy algorithm will require as a precondition that the input is valid and will guarantee that the output will be valid with respect to the input, as we will see below.

The types used in the verification part, as we have already seen in the previous chapter, will be immutable types. However, as this is a general template, we cannot specify a concrete type because for each problem it might be different, so we need to add opaque type declarations, which will be refined in the concrete module. So, we define three: *Input*, *Elem* and *Solution*. Type *Input* will represent the type of the input parameters; for example, in the case of the *files problem* it would be the type of the pair (*s_files*,*s_disc*), whose components are a sequence of naturals and a natural, respectively. Unlike the type *Input*, which can represent any immutable type, the type *Output* must necessarily be a sequence type, because the technique of exchange proof requires that the solutions are represented by sequences of elements. For this reason, the type *Elem* represents the type of the elements of the solution, so the type *Solution* is defined as a sequence of elements with type *Elem*. For example, in the *files problem Elem* would be the type *bool*. Both types *Input* and *Elem* use the following modifier: `!new`. This is needed to let Dafny know that those types are non-reference values.

All of these types and classes will be represented in Dafny in the following way:

```
abstract module GreedyAlgorithm {

    type Input(!new, =) // Immutable input data
    type Elem(!new, =) // Immutable solution elements

    class InputA {
      ghost var Repr: set<object>

      predicate Valid()
      reads this, Repr
    }

    class OutputA {
      ghost var Repr: set<object>

      predicate Valid(x: InputA)
      reads this, Repr, x, x.Repr
      requires x.Valid()
    }

    type Solution = seq<Elem>
    (...)
}
```

The relation between the algorithm types, *InputA* and *OutputA*, and the immutable types used for verification, *Input* and *Solution*, is established through functions *inputTrans* and *outputTrans*. They are defined only for valid instances of *InputA* and *OutputA*, so they must be able to read their footprints.

```
function inputTrans(x: InputA) : Input
reads x, x.Repr
requires x.Valid()
```

```
function outputTrans(x: InputA, y: OutputA) : Solution
reads x, x.Repr, y, y.Repr
requires x.Valid() ∧ y.Valid(x)
```

However, these functions do not ensure in any way that the user defines them in a reasonable way. So, in order to claim that the transformations from *InputA* to *Input* and from *OutputA* to *Solution* is done correctly, we should guarantee that everything that is true for *Input* and *Solution* is also true for *InputA* and *OutputA*. In order to be able to guarantee such property, we need that *inputTrans* and *outputTrans* are inyective. In order to guarantee that we define two methods *inputRecover* and *outputRecover* that meet *inputRecover(inputTrans(x)) = x* and *outputRecover(outputTrans(x,y)) = y*. Once we know this verifies, all the properties that are true in the world of verification are also true in the concrete world. These methods should be developed in the specific module of each problem. However, we will only develop them in the *files problem*, as it is quite methodical and to avoid repeating ourselves.

```
method inputRecover(x: Input) returns (s: InputA)
ensures s.Valid()
ensures inputTrans(s) = x

method outputRecover(xA: InputA, y: Solution) returns (s: OutputA)
requires xA.Valid()
requires isValidInput(inputTrans(xA))
requires isSolution(inputTrans(xA), y)
ensures s.Valid(xA)
ensures outputTrans(xA, s) = y
```

Once we have our generic types for input and output parameters, we can focus on the predicates. All our predicates will need a precondition to ensure that the input parameters meet some requirements including both those that guarantee the existence of solution to the problem and those concerning a particular format on the input values (e.g., that they are sorted). This is done with the predicate *isValidInput*, which is not defined at the abstract level because it depends on the problem. In addition, in some predicates as *isBetterSol*, *isSolGreedy* and *isSolOptimal*, we have to add as preconditions that the parameter *s* is a solution. We already saw this in the *files problem*.

In the following we can see the predicates *isSolution*, *isBetterSol*, *isSolGreedy* and *isSolOptimal*. All of them represent what their names indicate. They are similar to what we have already seen.

```
predicate isValidInput(x: Input)

predicate isSolution(x: Input, s: Solution)
requires isValidInput(x)

predicate isBetterSol(x: Input, s1: Solution, s2: Solution)
requires isValidInput(x)
requires isSolution(x,s1) ∧ isSolution(x,s2)

predicate isSolGreedy(x: Input, s: Solution)
requires isValidInput(x)
requires isSolution(x,s)

predicate isSolOptimal(x: Input, s: Solution)
```

```
requires isValidInput(x)
requires isSolution(x,s)
{
  ∀ sol | isSolution(x,sol) :: isBetterSol(x,s,sol)
}
```

There are two new lemmas: *isBetterSolTrans* and *isBetterSolRefl*. They represent the two properties that the predicate *isBetterSol* must verify: transitivity and reflexivity. This is represented in Dafny as follows:

```
lemma isBetterSolTrans(x: Input, s1: Solution, s2: Solution, s3: Solution)
requires isValidInput(x)
requires isSolution(x,s1) ∧ isSolution(x,s2) ∧ isSolution(x,s3)
requires isBetterSol(x,s1,s2) ∧ isBetterSol(x,s2,s3)
ensures isBetterSol(x,s1,s3)

lemma isBetterSolRefl(x: Input, s: Solution)
requires isValidInput(x)
requires isSolution(x,s)
ensures isBetterSol(x,s,s)
```

These lemmas cannot be proved at the abstract level, since their validity depend on the specific definitions for the predicates given before. They have to be proved at a more concrete level.

Lemma *betterThanOptimal* ensures that if we have an optimal solution and another one that is better, then both are optimal. The proof for this lemma is generic for all problems:

```
lemma betterThanOptimal(x: Input, s: Solution, optimalS: Solution)
requires isValidInput(x)
requires isSolution(x,s) ∧ isSolution(x,optimalS)
requires isSolOptimal(x,optimalS) ∧ isBetterSol(x,s,optimalS)
ensures isSolOptimal(x,s)
{
  ∀ s' | isSolution(x,s') ensures isBetterSol(x,s,s')
  {
    assert isBetterSol(x, s, optimalS);
    assert isBetterSol(x, optimalS, s');
    isBetterSolTrans(x, s,optimalS, s');
  }
}
```

Now we show the header of the greedy algorithm that should be refined in the concrete module. As we have mentioned before, our input and output parameters have types *InputA* and *OutputA*, respectively. As they are classes, we have to ensure both verify the predicate *Valid*; also we have to check if the input we receive is correct with the predicate *isValidInput*. The aim of the *greedy* algorithm is to construct a greedy solution, so we must return a variable called *res* of type *OutputA*, which have to verify *isSolution* and *isSolGreedy* predicates. Here we are able to see the usage of *inputTrans* and *outputTrans* functions. As the predicates *isSolution* and *isSolGreedy* expect parameters of type *Input* and *Solution*, we need these functions to convert from *InputA* to *Input* and from *OutputA* to *Solution*.

```
method greedy(x: InputA) returns (res: OutputA)
requires x.Valid()
requires isValidInput(inputTrans(x))
ensures res.Valid(x)
```

```
ensures isSolution(inputTrans(x),outputTrans(x,res))
ensures isSolGreedy(inputTrans(x),outputTrans(x,res))
```

In order to prove that the greedy solution is optimal, we implement method *algorithm*, which calls *greedy* to build the solution and then calls *reduction* to ensure it is an optimal solution. However, for this method to be verified, we need the help of another extra lemma: *existsOptimalSolution*. Lemma *existsOptimalSolution* guarantees that if the input parameters are valid, then an optimal solution exists. We need this lemma because the exchange proof method requires an optimal solution as input parameter against which we can compare our greedy solution. And to do so, it must first ensure the existence of an optimal solution.

```
lemma existsOptimalSolution(x: Input)
requires isValidInput(x)
ensures exists sol :: isSolution(x,sol) ∧ isSolOptimal(x,sol)

method algorithm(x: InputA) returns (res: OutputA)
requires x.Valid()
requires isValidInput(inputTrans(x))
ensures res.Valid(x)
ensures isSolution(inputTrans(x),outputTrans(x,res))
ensures isSolOptimal(inputTrans(x),outputTrans(x,res))
{
  ghost var inputTr := inputTrans(x);
  res := greedy(x);
  assert inputTrans(x) = old(inputTrans(x));
  existsOptimalSolution(inputTr);

  ghost var sol :| isSolution(inputTr, sol) ∧ isSolOptimal(inputTr, sol);
  reduction(inputTr, outputTrans(x,res),sol);
}
```

Now we can focus on the verification part of this module. We are going to find three methods in all our problems: *reduction*, *endReduction* and *reduction_step*. Let us see what each method establishes.

Method *reduction* receives the input data from the problem, a greedy solution and an optimal solution. We must assume as a precondition the input data is valid; and as post-condition we ensure that greedy is better solution than the optimal and, for that reason, *greedy* is an optimal solution. The idea, as we saw in the previous chapter with the *files problem*, is to compare both solutions, *greedy* and *optimal*; and if we find the first position where *greedy* and *optimal* are not equal, then we must call *reduction_step*, where we will study all possible situations we can face depending on our problem. The aim is to maintain our optimal solution untouched and we will create a new one called *optimal'*, where we will make all the needed changes. It is an invariant of the loop which ensures that the solution *optimal'* we are modifying is better than *optimal* and consequently it is also optimal; as we are changing *optimal'* to be equal to *greedy*, it would not make any sense to change our solution to something worse. In order this invariant holds on entry we need to prove that relation *isBetterSol* is reflexive, so we invoke lemma *isBetterSolRefl* before entering to the loop. In order to prove that the invariant holds after each loop iteration we need to prove that relation *isBetterSol* is transitive, so at the end of the loop iteration we invoke lemma *isBetterSolTrans*. After the loop, we call *endReduction* which modifies *optimal'* in such way that it returns a solution equal to *greedy*. In the case of the *files problem*, for example, the

solution returned by *endReduction* was the same as the one it received, but in some other cases some additional changes may be necessary. Using again transitivity we can conclude that *optimal'* is optimal and hence so is *greedy*.

```
ghost method reduction(x: Input, greedy: Solution, optimal: Solution)
requires isValidInput(x)
requires isSolution(x, greedy) ∧ isSolution(x, optimal)
requires isSolGreedy(x, greedy) ∧ isSolOptimal(x, optimal)
ensures isBetterSol(x, greedy, optimal)
ensures isSolOptimal(x, greedy)
{
    var i := 0; var optimal' := optimal;
    isBetterSolRefl(x, optimal');

    while (i < |greedy| ∧ i < |optimal'|)
      invariant 0 ≤ i ≤ |optimal'|
      invariant 0 ≤ i ≤ |greedy|
      invariant isSolution(x, optimal')
      invariant isSolOptimal(x,optimal')
      invariant isBetterSol(x, optimal', optimal)
      invariant optimal'[..i] = greedy[..i]
      {
          if (greedy[i] ≠ optimal'[i])
          {
              ghost var oldOptimal := optimal';
              optimal' := reduction_step(x, greedy, optimal', i);
              betterThanOptimal(x, optimal', oldOptimal);
              isBetterSolTrans(x,optimal',oldOptimal,optimal);
          }
          i := i+1;
      }

    ghost var oldOptimal := optimal';
    optimal' := endReduction(x,greedy,optimal');
    isBetterSolTrans(x,optimal',oldOptimal,optimal);
    betterThanOptimal(x, greedy, optimal);
}
```

We should mention that the condition of the loop has been modified from the one we wrote in the *reduction* method of the *files problem*. In that code, the condition was `while (i < |optimal|);` however, in the *reduction* method of the abstract module it is necessary to write a more restrictive condition, where we make sure that i is smaller than $|greedy|$ and $|optimal'|$. This is done because, in general, the solutions to a problem are not necessarily of the same length.

Let us see now the methods *endReduction* and *reduction_step*. In the *Modules* files we will just see their specification, and they will have to be refined in each particular problem. As we have already seen in the previous chapter, *endReduction* method receives a greedy and an optimal solution, which are equal up to the length of the shortest solution; and returns another solution *optimal'*, which verifies two properties:

1. it is better solution than *optimal*, therefore it is also an optimal solution.
2. it is equal to greedy, therefore it is also a greedy solution.

The conclusion is that if an optimal and a greedy solution are equal up to the length of the shortest one, then the greedy solution is also optimal.

```
ghost method endReduction(x: Input, greedy: Solution, optimal: Solution)
  returns (optimal': Solution)
requires isValidInput(x)
requires isSolution(x, greedy) ∧ isSolution(x, optimal)
requires isSolGreedy(x, greedy) ∧ isSolOptimal(x, optimal)
requires |optimal| ≤ |greedy| ⟹ optimal[..|optimal|] = greedy[..|optimal|]
requires |optimal| > |greedy| ⟹ optimal[..|greedy|] = greedy[..|greedy|]
ensures isSolution(x, optimal')
ensures isBetterSol(x, optimal', optimal)
ensures optimal' = greedy
```

The *reduction_step* method receives two solutions, a greedy and an optimal one, which are equal up to some position i but not included; and then it returns the *optimal* solution modified so that both solutions are equal up to and including i. The idea is to apply the *exchange proof* applied to a particular location in the solution.

```
ghost method reduction_step(x: Input, greedy: Solution, optimal: Solution, i: nat)
  returns (optimal': Solution)
requires isValidInput(x)
requires isSolution(x, greedy) ∧ isSolution(x, optimal)
requires isSolGreedy(x, greedy) ∧ isSolOptimal(x, optimal)
requires 0 ≤ i < |greedy| ∧ 0 ≤ i < |optimal|
requires greedy[i] ≠ optimal[i] ∧ greedy[0..i] = optimal[0..i]
ensures isSolution(x, optimal')
ensures isBetterSol(x, optimal', optimal)
ensures |optimal| = |optimal'|
ensures optimal'[0..i+1] = greedy[0..i+1]
```

To recap, let's list everything that needs to be defined to refine this module:

1. Types *InputA* and *OutputA* that reflect the input and output of the algorithm, and may be mutable.
2. Type *Input* which represents the input to the algorithm as an immutable data type, and type *Elem* which denotes each component of a solution.
3. *inputTrans* and *outputTrans* which transform mutable types into immutable types, and the corresponding *inputRecover* and *outputRecover* methods.
4. *isSolution* predicate, which determines if a solution is valid.
5. *isBetterSol* predicate, which defines a pre-order relationship between solutions, and *isBetterSolRefl* and *isBetterSolTrans* lemmas, which prove its reflexivity and transitivity.
6. *existsOptimalSolution* lemma, which proves the existence of an optimal solution.
7. *greedy* method, which implements the greedy algorithm.
8. *endReduction* and *reductionStep* lemmas, used in the exchange proof.

4.2 GreedyAlgorithmValue Module

We have seen our first abstract module. However, we have defined another abstract module, called *GreedyAlgorithmsValue*, which refines our abstract module *GreedyAlgorithm*. The aim of this module is to provide a framework for a subclass of greedy algorithms which share common aspects regarding the existence of optimal solutions and the proof of reflexivity and transitivity of the optimality relation. In this subclass of greedy algorithms, the concept of optimality is based on finding a function that assigns a natural value to each solution. This is what we have called, in the previous chapter, the objective function.

```
function solutionValue(x: Input, s: Solution): nat
requires isValidInput(x)
requires isSolution(x, s)
```

In this way, we can consider that a solution is better than another one if it has a smaller value. So, in this case, we are minimising our objective function. For example, in the *files problem* this function would be $N - F$, where N is the total number of files and F is the number of files chosen to be stored.

```
module Files refines GreedyAlgorithmValue{
    ...
    function solutionValue(x: Input, s: Solution): nat
    {
        |x.0| - value(x,s)
    }
    ...
}
```

Back to the *GreedyAlgorithmsValue* module, the transitivity and reflexivity lemmas are proved by Dafny without any help; it means, without need of adding a body in the lemma.

Then, the predicate *isBetterSol* that we defined without body in *GreedyAlgorithm* module is completed at this level.

```
predicate isBetterSol(x:Input, s1:Solution, s2:Solution)
{
  solutionValue(x, s1) ≤ solutionValue(x, s2)
}
```

Not all problems can be based on assigning a value of optimality to each function. So, the idea was to create an abstract module for this specific type of problems. Then, all modules that refine *GreedyAlgorithmValue* must have:

- a definition for *solutionValue*.
- a proof of the existence of a solution (that it does not need to be optimal).

Let us focus now on the *existsOptimalSolution* lemma. If we can assume the existence of a solution, it is possible to prove the existence of an optimal solution, because otherwise it

would exist an infinite sequence of solutions, each one with a smaller value than the previous one. So assuming there exists a solution (a lemma we will prove in each problem), we create a variable *currentSol*, where we store a solution; and a variable *value*, where we store the value of such solution. Then, we have a loop that stops when the optimal solution is found. The idea is to keep checking if we can find a better solution than the current one until we cannot. If we cannot find a better one than the one we have, then we already have an optimal solution. The key point here is the *decreases value* clause, because it ensures that the loop finishes and therefore it finds an optimal solution. We could not prove this lemma in the *GreedyAlgorithm* module, because we did not have an objective function to compare solutions with.

```
lemma existsSolution(x: Input)
requires isValidInput(x)
ensures exists sol :: isSolution(x, sol)

lemma existsOptimalSolution(x:Input)
{
  existsSolution(x);
  var currentSol :| isSolution(x, currentSol);
  var value := solutionValue(x, currentSol);

  while (!isSolOptimal(x, currentSol))
    invariant isSolution(x, currentSol)
    invariant value = solutionValue(x, currentSol)
    decreases value
    {
        var betterSol :| isSolution(x, betterSol) ∧ !isBetterSol(x, currentSol, betterSol);
        assert solutionValue(x, currentSol) > solutionValue(x, betterSol);
        currentSol := betterSol;
        value := solutionValue(x, betterSol);
    }
  assert isSolution(x, currentSol) ∧ isSolOptimal(x, currentSol);
}
```

If we remember the list of things needed to refine the *GreedyAlgorithm* module, items 5 and 6 are refined in this new abstract module so we no longer need to refine them in the problem-specific modules. Instead, whoever refines the *GreedyAlgorithmValue* module, must indicate a new point that we can add to our list:

6'. *existsSolution* lemma, which proves the existencia of a solution, not necessarily optimal.

4.3 Examples

Once we understand how our modules work and the purpose for which we created them, let us see the *files problem* we have already studied but using the modules. We will also see other problems where we can apply greedy algorithms and their corresponding modules.

Each problem will be done in a specific module, refining *GreedyAlgorithmValue* module. All predicates, lemmas, methods or functions we have already defined in our modules shall not need to be repeated.

4.3.1 Files Problem

First of all, we need to start specifying the types of the algorithm's input and output. Class *InputA* contains two fields, an array *s_files* containing the space occupied by the files and a natural number *s_disc* that represents the total space of the computer/device. The constructor builds an instance of the class and assigns to the footprint the mutable components, i.e. the array *s_files*, and the object itself. The object will be *Valid* as long as the footprint remains the same, i.e. it does not capture any other memory locations. In this way, the refined greedy method knows which memory locations owns each object in scope and is able to prove properties about them. We do something similar in the *OutputA* class; however, we need two fields we did not need when doing it without the modules. First, *size* tells us the position of the solution up to which it has been filled. This field is used to express partial solutions, so *files* will represent the number of files that the solution has decided to store up to *size*.

```
class InputA ... {
  var s_files: array<nat>;
  var s_disc: nat;

  constructor (f: array<nat>, d: nat)
  ensures Valid()
  ensures s_files = f ∧ s_disc = d
  {
    s_files := f;
    s_disc := d;
    Repr := {this, s_files};
  }

  predicate Valid()
  {
    Repr = {this, this.s_files}
  }
}

class OutputA ... {
  var sol: array<Elem>;
  var files: nat;
  var size: nat; //partial output

  constructor (x: InputA)
  requires x.Valid()
  ensures ∀ z | z in Repr :: fresh(z)
  ensures Valid(x)
  ensures sol.Length = x.s_files.Length
  ensures size = 0
  {
    sol := new Elem[x.s_files.Length];
    Repr := {this, sol};
    files := 0;
    size := 0;
  }

  predicate Valid(x: InputA)
  {
    Repr = {this, sol} ∧
    sol.Length = x.s_files.Length ∧
    0 ≤ size ≤ sol.Length ∧
    files = value(inputTrans(x), sol[..size])
  }
}
```

Then, we need to specify the types of our data for the verification part. The entry is a sequence of naturals that represents the amount of space each file occupies and a natural number that tells us the space available on the computer/device where we want to store the files. This is stored in type *Input*. We have to specify also the type *Elem*, which in this case is the type of booleans. This will be represented in Dafny as follows:

```
module Files refines GreedyAlgorithmValue {
  type Input = (seq<nat>, nat)
  type Elem = bool
  ...
}
```

Regarding *inputTrans* and *outputTrans* functions, we need to transform from *InputA* (resp. *OutputA*) to *Input* (resp. *Solution*), which we achieve just by obtaining the sequence of elements of the corresponding array. This is represented in Dafny as:

```
function inputTrans(x:InputA) : Input
{
  (x.s_files[..],x.s_disc)
}

function outputTrans(x:InputA, y:OutputA) : Solution
{
  (y.sol)[..]
}
```

Once we have defined these functions, we can develop the methods *inputRecover* and *outputRecover*, so we ensure that the properties verified in *Input* or *Solution* are also verified in *InputA* or *OutputA*, respectively. In *inputRecover* method, we just need to convert back the sequence into an array. That is what is done in the loop of the method. In the second method, as we have a solution, we are able to calculate the number of files selected in the solution and the size of it. Also, we need to convert the sequence into an array.

```
method inputRecover(x: Input) returns (s: InputA)
{
    var i := 0;
    var a := new nat[|x.0|];
    while i < |x.0|
      invariant 0 ≤ i ≤ |x.0|
      invariant ∀ j | 0 ≤ j < i :: a[j] = x.0[j]
      {
        a[i] := x.0[i];
        i := i + 1;
      }

    assert ∀ j | 0 ≤ j < |x.0| :: a[j] = x.0[j];
    assert a[..] = x.0;
    s := new InputA(a, x.1);
    assert inputTrans(s) = (a[..], x.1);
    assert inputTrans(s) = x;
}

method outputRecover(xA: InputA, y: Solution) returns (s: OutputA)
{
    assert isValidInput(inputTrans(xA));
    assert isSolution(inputTrans(xA), y);
    assert |y| = |inputTrans(xA).0|;

    s := new OutputA(xA);
```

```
    assert s.sol.Length = |y|;
    assert s.Valid(xA);
    var i := 0;
    while i < |y|
      modifies s`files, s`size, s.sol
      invariant 0 ≤ i ≤ |y| ∧ i = s.size
      invariant s.sol.Length = |y|
      invariant ∀ j | 0 ≤ j < i :: s.sol[j] = y[j]
      invariant s.Valid(xA)
      invariant s.files = value(inputTrans(xA),s.sol[..s.size])
      {
        assert (s.sol)[..s.size + 1][..s.size] = (s.sol)[..s.size];

        s.sol[i] := y[i];
        if (y[i] = true) {
           s.files := s.files+1;
        }
        s.size := s.size + 1;
        i := i + 1;
      }
      assert outputTrans(xA, s) = y;
}
```

The objective function, as we have mentioned before, is the number of total files minus the number of files we have decided to store:

```
function solutionValue(x: Input, s: Solution): nat
{
  |x.0| - value(x,s)
}
```

The predicate *isValidInput* must ensure that the sequence of files is increasingly sorted. We need to require this in order to be able to apply the greedy strategy. This does not pose any constraint on the possible input data.

```
predicate isValidInput(x: Input) {
  sorted(x.0)
}
```

Predicates *isSolOptimal* or *isBetterSol* are already defined in the abstract module. Predicates *isSolution* and *isSolGreedy* are the same that we defined in the previous chapter. A sequence of booleans is a solution if the length of the solution is equal to the number of files we have and if all the files we have decided to store fit in our computer/device. For the definition of a greedy solution, we needed to ensure the two properties we already mentioned in the previous chapter:

1. for all positions i and j, where $i < j$, if $sol[i] =$ false then $sol[j] =$ false. It would not make any sense to not store file i but store file j knowing that $s\_files[i] \leq s\_files[j]$.
2. for all files we have decided not to store, it is because it is not possible to store them.

```
predicate isSolGreedy(x:Input, s:Solution)
{
    (∀ i,j | 0 ≤ i < j < |s| :: (s[i] = false ⟹ s[j] = false)) ∧
    ∀ m | 0 ≤ m < |s| ∧ s[m] = false :: (sum(x.0, s[m := true]) > x.1)
}
```

Before focusing on the algorithm, we must prove the existence of a solution. To do that we use a lemma that is specified in the *GreedyAlgorithmValue* module, but that must be proved in each problem. The idea is to try to find the simplest possible solution and prove that it is indeed a solution. In the case of the *files problem*, the simplest solution would be to store no files at all, so that the solution for sure verifies the property of not exceeding the maximum space of the computer/device. However, to prove this last property we need an extra lemma *sumAllFalse*, where we prove that if all elements of the solution are *false*, then sum of the weights of the selected files is 0.

```
lemma existsSolution(x: Input)
{
    var sol := seq(|x.0|, n ⇒ false);

    {sumAllFalse(x.0, sol);}
    assert sum(x.0, sol) = 0;
    assert isSolution(x, sol);
}

lemma sumAllFalse(a: seq<nat>, b: Solution)
requires |a| = |b| ∧ ∀ i | 0 ≤ i < |b| :: b[i] = false
ensures sum(a,b) = 0
decreases |b|
{
    if |a| = 0 {}
    else {
      assert sum(a,b) = sum(a[..|a|−1],b[..|b|−1]) + (if b[|b|−1] then a[|a|−1] else 0);
      {sumAllFalse(a[..|a|−1],b[..|b|−1]);}
    }
}
```

Concerning the algorithm, we just have to adapt our previous algorithm to the types *InputA* and *OutputA*. This time instead of having a variable i, which represents the file we were considering, we are going to use *res.size*, the parameter of the class *OutputA*. We will go through each position to check if we are able or not to store one more file; and in case there are some files we cannot store we have another loop to establish the rest of the positions to false.

```
method greedy(x: InputA) returns (res: OutputA)
{
    var accum := 0;
    res := new OutputA(x);
    res.files := 0;
    res.size := 0;

    while (res.size < x.s_files.Length ∧ (accum + (x.s_files)[res.size] ≤ x.s_disc))
      decreases (x.s_files).Length − res.size
      invariant 0 ≤ res.size ≤ (x.s_files).Length = (res.sol).Length
      invariant accum ≤ x.s_disc
      invariant accum = sum((x.s_files)[..res.size], (res.sol)[..res.size])
      invariant isSolution((x.s_files[..res.size],x.s_disc), (res.sol)[..res.size])
      invariant isSolGreedy((x.s_files[..res.size],x.s_disc), (res.sol)[..res.size])
      //To be able to fill the solution array
      invariant ∀ z | z in res.Repr :: fresh(z)
      invariant res.Valid(x)
      {
        res.sol[res.size] := true;
        accum := accum + (x.s_files)[res.size];

        assert (x.s_files)[..res.size + 1][..res.size] = (x.s_files)[..res.size];
        assert (res.sol)[..res.size + 1][..res.size] = (res.sol)[..res.size];
```

```
        res.files := res.files + 1;
        res.size := res.size + 1;
      }

    while (res.size < (x.s_files).Length)
      decreases (x.s_files).Length - res.size
      invariant 0 ≤ res.size ≤ (x.s_files).Length = (res.sol).Length
      invariant accum = sum((x.s_files)[..res.size], (res.sol)[..res.size])
      invariant isSolGreedy((x.s_files[..res.size],x.s_disc), (res.sol)[..res.size])
      //To be able to fill the solution array
      invariant ∀ z | z in res.Repr :: fresh(z)
      invariant res.Valid(x)
      {
        (res.sol)[res.size] := false;

        assert (x.s_files)[..res.size + 1][..res.size] = (x.s_files)[..res.size];
        assert (res.sol)[..res.size + 1][..res.size] = (res.sol)[..res.size];

        res.size := res.size + 1;
      }

    assert (x.s_files)[..] = (x.s_files)[..(x.s_files).Length];
    assert (res.sol)[..] = (res.sol)[..(x.s_files).Length];
}
```

The only thing we must emphasize, because it was not needed before and now it does due to the classes, it this two last invariants we needed to add in both loops:

```
invariant ∀ z | z in res.Repr :: fresh(z)
invariant res.Valid(x)
```

They let Dafny know that the footprints of *res* and x are disjoint and that therefore the modifications made to *res* do not affect x.

Method *reduction_step* is already defined in our abstract module, and the *swap* method was defined in the previous chapter. The only new thing in the verification part is that this time *endReduction* returns something. We must return a solution, which verifies it is better than *optimal* and equal to *greedy*. This is verified by *optimal*, without the need of changing anything. In some problems it will be necessary to make some changes to the optimal solution received as parameter.

```
ghost method endReduction(x: Input, greedy: Solution, optimal: Solution)
                          returns (optimal': Solution)
{
  optimal' := optimal;
}
```

4.3.2 Gas Station Problem

Let us present a new problem: *gas station problem*. We are on a trip and we have to refuel the car from time to time and each time we do it we fill the tank, but we want to refuel the car as few times as possible in order to make the minimum number of stops. So we have a list of all the gas stations and the distance in kilometres between them; and we also know the

number of kilometres we can drive without having to stop. Then, our problem is to figure out in which gas stations we have to stop to minimize the numbers of stops.

The greedy strategy we will follow is the following: we just stop when it is necessary. So, for each stop, we check if we can continue until the next gas station without running out of fuel and if not, we must stop. Otherwise, we continue and in the next gas station we check again.

First of all, we have to study the types of the algorithm. We must represent our two input parameters in the *InputA* class: *distance* is the array of naturals that represents the distance between the consecutive gas stations and K is the number of kilometers our car can drive without stopping. In *OutputA* we must define our solution (an array of booleans, that tells us if we stop in the gas station or not), the number of stops and the size of a partial solution. This is represented in Dafny as follows:

```
class InputA ... {
  var distance: array<nat>;
  var k: nat;

  constructor (d: array<nat>,l: nat)
  {
    distance := d;
    k := l;
    Repr := {this, distance};
  }

  predicate Valid()
  {
    Repr = {this, this.distance}
  }
}

class OutputA ... {
  var sol: array<Elem>;
  var stops: nat;
  var size: nat; //partial output

  constructor (x: InputA)
    requires x.Valid()
    ensures ∀ z | z in Repr :: fresh(z)
    ensures Valid(x)
  {
    sol := new Elem[x.distance.Length+1];
    Repr := {this, sol};
    stops := 0;
    size := 0;
  }

  predicate Valid(x: InputA)
  {
    Repr = {this, sol} ∧
    sol.Length = x.distance.Length+1 ∧
    0 ≤ size ≤ sol.Length ∧
    stops = value(inputTrans(x), sol[..size])
  }
}
```

Now, let us see the verification part. We receive as inputs two parameters: a sequence of naturals, which represents the distance between each gas station; and a natural, that represents the number of kilometers our car can do without stopping. This is stored in the

type *Input*. Then we have the type *Elem*, which tells us the type of the elements of the sequence *Solution*. As in the previous problem, they are booleans which represent if we stop in the gas station or not.

```
module GasStation refines GreedyAlgorithmValue {
  type Input = (seq<nat>, nat)
  type Elem = bool
  (...)
 }
```

Functions *inputTrans* and *outputTrans* which are quite similar to those of the *files problem*. They simply have to convert our sequence of naturals (that represents the distance between gas stations) to an array; and a solution (which is a sequence of booleans) to an array.

```
  function inputTrans(x:InputA) : Input {
    (x.distance[..],x.k)
  }

  function outputTrans(x:InputA,y:OutputA) : seq<Elem> {
    (y.sol)[..]
  }
```

This time we want to minimize the number of stops, then our objective function must be the number of trues in the solution. For this reason we create a function *value* that, given a sequence of booleans, calculates the number of true values in that sequence, and so our function *solutionValue* calls this other function.

```
    function solutionValue(x: Input, s: Solution): nat {
        value(x,s)
    }
```

Regarding the predicate *isValidInput*, we must ensure that there are at least two gas stations, the departure and arrival points, and that every distance between gas stations is smaller than the maximum distance allowed by the fuel deposit of the car. Otherwise, the problem could not be solved.

```
    predicate isValidInput(x:Input) {
        |x.0|>0 ∧ smaller(x)
    }
```

Let us focus now in what is new in this problem. Starting with the definition of being a solution, three conditions must be met. The first one is that our final solution has the same length as the sequence of distances plus one, because if for example we have 7 distance values, it means that there are 8 gas stations and we have to return for those 8 gas stations a boolean indicating whether we stopped or not. The second condition is that we must stop at both the first and the last gas station, as they are the origin and destination of the journey. And the last condition is that it must hold that it is a partial solution. And what we need to ensure in the *isPartialSolution* predicate is that for any two stops that are consecutive (which means that we do not stop at gas stations in between) we are able to drive between them.

```
predicate isPartialSolution(x: Input, s: Solution)
requires isValidInput(x)
requires |s| ≤ |x.0| + 1
{
  ∀ i, j | 0 ≤ i < j ≤ |s|−1 ∧ consecutive_stops(i,j,s) :: sum_distance(x.0[i..j]) ≤ x.1
}

predicate isSolution(x: Input, s: Solution)
{
  |s| = |x.0| + 1 ∧ s[0] = s[|s|−1] = true ∧ isPartialSolution(x,s)
}
```

The definition of *isSolGreedy* follows the same idea. A sequence of booleans is a partial greedy solution if for any pair of consecutive stops i and j, it is not possible to reach $j+1$ from i. Because what a greedy solution must ensure is that we stop as few times as possible; then if we stop without being necessary, it is not a greedy solution.

```
predicate isPartialSolGreedy(x: Input, s: Solution)
requires isValidInput(x) ∧ |s| < |x.0| + 1
requires isPartialSolution(x, s)
{
  ∀ i,j | 0 ≤ i < j ≤ |s|−1 ∧ consecutive_stops(i,j,s) :: sum_distance(x.0[i..j+1]) > x.1
}

predicate isSolGreedy(x: Input, s: Solution)
{
  |s| = |x.0| + 1 ∧ s[0] = s[|s|−1] = true ∧ isPartialSolGreedy(x,s[..|s|−1])
}
```

Before finishing the part related to the solutions, let us see the lemma that proves the existence of a solution. As commented above, the idea is to find the simplest possible solution; so in this problem, it would be to stop at all gas stations, thus ensuring the property of always arriving at the next one. However, we need an extra lemma *solAllTrue*, which proves that a solution where all the elements are *true* (which means, we stop in every gas station), verifies that we always can reach the next gas station.

```
lemma existsSolution(x: Input)
{
    var sol := seq(|x.0|+1, n ⇒ true);

    {solAllTrue(x,sol);}
    assert isSolution(x,sol);
}

lemma solAllTrue(x: Input, sol: Solution)
requires isValidInput(x)
requires |x.0|+1 = |sol| ∧ ∀ i | 0 ≤ i < |sol| :: sol[i] = true
ensures ∀ i,j | 0 ≤ i < j ≤ |sol|−1 ∧ consecutive_stops(i,j,sol) ::
        sum_distance(x.0[i..j]) ≤ x.1
{
    ∀ i,j | 0 ≤ i < j ≤ |sol|−1 ∧ consecutive_stops(i,j,sol) ensures i+1 = j
    {
      if (i+1 ≠ j) {
        assert sol[i+1] = true;
        assert !consecutive_stops(i,j,sol);
      }
      else {}
    }
}
```

The idea in the algorithm is to traverse the array of distances with a loop, and inside this loop we must study if it is necessary to stop or if we can wait for the next stop. To decide if we stop or not, we are going to declare a variable called *km*, which represents the kilometers we have driven since the last stop. If we are able to reach the next station, then we write a *false* in the current position of the solution, and add to *km* the distance to the next station. Otherwise, we write *true* in that position and restore our variable *km* to 0.

```
method greedy(x: InputA) returns (res: OutputA)
{
    var km := (x.distance)[0]; //kilometers walked since the last stop
    res := new OutputA(x);
    res.sol[0] := true;
    res.stops, res.size := 1, 1;

    var j := 0;

    while res.size < (x.distance).Length
      decreases (x.distance).Length - res.size
      invariant 0 ≤ res.size ≤ (x.distance).Length
      invariant  0 ≤ j < res.size
      invariant res.sol.Length = (x.distance).Length + 1 > 0 ∧ (res.sol)[0] = true
      //invariants for isPartialSolution
      invariant km = sum_distance((x.distance)[j..res.size]) ≤ x.k
      invariant (res.sol)[j] = true ∧ ∀ z | j < z < res.size :: (res.sol)[z] = false
      //invariant for the postconditions
      invariant isPartialSolution(inputTrans(x),(res.sol)[..res.size])
      invariant isPartialSolGreedy(inputTrans(x),(res.sol)[..res.size])
      //To be able to fill the solution array
      invariant ∀ z | z in res.Repr :: fresh(z)
      invariant res.Valid(x)

      {
        if (km + (x.distance)[res.size] ≤ x.k) { //it does not stop
          (res.sol)[res.size] := false;
          km := km + (x.distance)[res.size];

          propertyRestoreF(inputTrans(x),(res.sol)[..res.size],j);
          propertyRestoreF_2(inputTrans(x),(res.sol)[..res.size],j);
        }
        else { //it stops
          ghost var oldj := j;
          propertyRestoreT(inputTrans(x),(res.sol)[..res.size],oldj);
          propertyRestoreT_2(inputTrans(x),(res.sol)[..res.size],oldj);

          assert ∀ z | z in res.Repr :: fresh(z);
          (res.sol)[res.size] := true;

          assert isPartialSolution(inputTrans(x),(res.sol)[..res.size+1]);
          assert isPartialSolGreedy(inputTrans(x),(res.sol)[..res.size+1]);

          j := res.size;
          km := (x.distance)[j];
          res.stops := res.stops+1;
        }
        res.size := res.size + 1;
      }

    (res.sol)[res.size] := true;
    res.stops := res.stops + 1;
    assert res.stops = value(inputTrans(x),res.sol[..res.size+1]);
    assert isPartialSolution(inputTrans(x),(res.sol)[..res.size]);
    assert isPartialSolGreedy(inputTrans(x),(res.sol)[..res.size]);

    res.size := res.size + 1;

    assert isSolution(inputTrans(x),(res.sol)[..]);
```

```
    assert isSolGreedy(inputTrans(x),(res.sol)[..]);
}
```

Some of the invariants we had to add so Dafny could verify the method are:

1. the length of the solution is the number of distances plus one, i.e. the number of gas stations and we have to stop in the first gas station, so $res.sol[0] = true$.
2. as j represents the index of the last gas station where we have stopped and km represents the number of kilometers we have driven without stopping, the distance from the j to the gas station where we are at ($res.size$) is equal to km and always smaller than the maximum kilometers we can do without stopping.
3. as j represents the index of the last gas station where we have stopped, the solution in j must be $true$. And for all gas stations between that one and the one in $res.size$, we have not stopped, so the solution must contain $false$.

With those, Dafny is able to verify the next two invariants: *res.sol[..res.size]* is a partial solution and a partial greedy solution. However, for the verification part of the algorithm we need some auxiliary lemmas.

We need four different lemmas. In case we decide not to stop, we have *propertyRestoreF* and *propertyRestoreF_2*. The first one ensures that if we have a partial solution *sol* of which we know the last stop made, represented with the variable *lastT*, and we are able to reach the next station, then *sol+[false]* is also a partial solution. The second lemma is the same, but ensuring *sol+[false]* is a partial greedy solution knowing that *sol* is also a partial greedy solution. Let us see one of the lemmas, the other one is similar:

```
lemma propertyRestoreF(x:Input, sol:Solution, lastT:int)
requires isValidInput(x) ∧ |sol| < |x.0|
requires isPartialSolution(x,sol)
requires 0 ≤ lastT < |sol| ∧ sol[lastT] ==true ∧ ∀ z | lastT < z < |sol| :: sol[z] = false
requires sum_distance(x.0[lastT..|sol|+1]) ≤ x.1
ensures isPartialSolution(x,sol+[false])
{
  assert |sol+[false]| = |sol|+1;
  ∀ c,f | 0 ≤ c < f < |sol| + 1 ∧ consecutive_stops(c,f,sol+[false])
  ensures sum_distance(x.0[c..f]) ≤ x.1;
  {
    if (f < |sol|) {
      assert ∀ z | 0 ≤ z < |sol| :: (sol + [false])[z] = sol[z];
      assert consecutive_stops(c,f,sol);
      assert sum_distance(x.0[c..f]) ≤ x.1;
    }
    else {
      assert !consecutive_stops(c,|sol|,sol+[false]);}
    }
}
```

In case we have to stop, we need another two lemmas: *propertyRestoreT* and *propertyRestoreT_2*. The idea is the same as before but this time with a true instead of a false, because we stop in the gas station.

With these invariants, the lemmas and some additional assertions we do not show here, we are able to verify the algorithm. Let us finish with this problem by looking at how it was necessary to do the verification part to ensure that the greedy solution is optimal.

Let us remember we have already specified the *reduction* method in the modules which goes through both solutions, greedy and optimal, to compare them. In case we find a position i where $greedy[i] \neq optimal[i]$, we call *reduction_step* method. Let us start studying this method.

```
ghost method reduction_step(x: Input, greedy: Solution, optimal: Solution, i: nat)
returns (optimal': Solution)
{
    if (greedy[i]=true) //∧ optimal[i] = false
    {//Impossible

      //We are not allowing i to be 0 to make sure the existence of a previous TRUE.
      existTrueB(x,optimal,i);
      var lastT :| 0 ≤ lastT < i ∧ optimal[lastT] = true
          ∧ ∀ n | lastT < n < i :: optimal[n] = false;
      existTrueA(x,optimal,i);
      var nextT :| i ≤ nextT < |optimal|
          ∧ optimal[nextT] = true ∧ ∀ n | i < n < nextT :: optimal[n] = false;

      // Prove that as (lastT,i) are consecutive stops and the distance is  ≤ k,
      // then (lastT,i+1) has distance > k. So, (lastT,nextT) are consecutive
      // stops with distance > k. So, it is not sol.
      assert consecutive_stops(lastT,i,greedy) ∧ sum_distance(x.0[lastT..i]) ≤ x.1;
      assert x.1 < sum_distance(x.0[lastT..i+1]) ≤ sum_distance(x.0[lastT..nextT]);

      assert !isPartialSolution(x,optimal);
      assert !isSolution(x,optimal);
    }
    else //greedy[i] = false ∧ optimal[i] = true
    {
      existTrueB(x,optimal,i);
      var lastT :| 0 ≤ lastT < i
          ∧ optimal[lastT] = true ∧ ∀ n | lastT < n < i :: optimal[n] = false;

      existTrueA(x,greedy,i);
      var nextT :| i < nextT < |greedy|
          ∧ greedy[nextT] = true ∧ ∀ n | i < n < nextT :: greedy[n] = false;

      assert sum_distance(x.0[lastT..i+1]) ≤ sum_distance(x.0[lastT..nextT]) ≤ x.1;
      optimal' := postpone(x,optimal,i,lastT);
    }
}
```

For *reduction_step*, as it receives as input parameters two solutions, *greedy* and *optimal*, where $greedy[i] \neq optimal[i]$, and since in this problem the elements of a solution are booleans, we have two possibilities: $greedy[i] = true$ (and $optimal[i] = false$) or $greedy[i] = false$ (and $optimal[i] = true$).

Like in the *files problem*, the first case is not possible because otherwise *optimal* would not be a solution. Let us understand why. We know, by definition, that greedy only stops if it cannot reach the next station. Then, if $lastT$ is the last *true* before position i ($lastT$ is a common position for both solutions as they are equal until position i), we know that the distance between $lastT$ and $i+1$ is greater than the one we can drive. So, if in *optimal* there is a *false* in position i, wherever is the next *true* (let us say in $nextT$) we know that the

distance between *lastT* and *nextT* is greater than the one we can drive without stopping. As a conclusion, *optimal* is not a solution. To prove all of this we have needed help of two auxiliary lemmas: *existTrueB*, which proves that exists a *true* before position $i > 0$ in any solution; and *existsTrueA*, which proves that exists a true after position $i < |sol| - 1$ in any solution.

```
// Exist a TRUE to the left of i
lemma existTrueB(x: Input, sol: Solution, i: nat)
requires isValidInput(x)
requires isSolution(x,sol)
requires 0 < i ≤ |sol|
ensures exists j :: 0 ≤ j < i ∧ sol[j] = true ∧ ∀ n | j < n < i :: sol[n] = false


// Exist a TRUE to the right of i
lemma existTrueA(x: Input, sol: Solution, i: nat)
decreases |sol|-i
requires isValidInput(x)
requires isSolution(x,sol)
requires 0 ≤ i < |sol|-1 ∧ sol[i] = false
ensures exists j :: i < j < |sol| ∧ sol[j] = true ∧ ∀ n | i < n < j :: sol[n] = false
```

The other case verifies that $greedy[i] = false$ and $optimal[i] = true$. Again, we know by definition that if greedy does not stop it is because we can reach the next station from the last stop; so we can postpone the stop in optimal. The idea will be to establish $optimal[i] = false$ and $optimal[i + 1] = true$. In this case, all postconditions that *reduction_step* requires will be ensured by the *postpone* method.

```
ghost method postpone(x: Input, v: Solution, i: nat, lastT: nat) returns (v_modified: Solution)
requires isValidInput(x)
requires isSolution(x, v)
requires 0 ≤ lastT < i < |v|-1 //i and j cannot be 0 either |v|-1
requires v[i] = true ∧ v[lastT] = true
requires sum_distance(x.0[lastT..i+1]) ≤ x.1
requires consecutive_stops(lastT,i,v)
ensures isSolution(x, v_modified)
ensures isBetterSol(x, v_modified, v) //they are equal
ensures v_modified = v[i := false][i+1 := true]
{
    // Postpone the stop -> we made the needed change in v_modified
    v_modified := v[i:=false][i+1:=true];

    assert v_modified[0..i] = v[0..i] ∧ v_modified[i+2..|v|] = v[i+2..|v|];
    isSol(x,v,v_modified,i,lastT);

    // isBetterSol(distance,k,v_modified,v)
    lessStops(x,v,i,i+1);
    assert isBetterSol(x,v_modified,v);
}
```

As we have already mentioned, we just need to change $optimal[i]$ from *true* to *false* and $optimal[i + 1]$ must be true. But we have to ensure that this new optimal solution is still a solution and optimal. To verify it keeps being a solution we have a new lemma called *isSol*, and to prove it is optimal we need to prove that it has now less or the same number of stops than before, for this we use *lessStops* lemma.

```
lemma lessStops(a: Input,b: Solution,i: nat,j: nat)
requires 0 ≤ i < j < |b|=|a.0|+1
requires b[i] = true // Similarly the other way around
```

```
ensures value(a,b) ≥ value(a,b[i:=false][j:=true])

lemma isSol(x: Input, v: Solution, v_modified: Solution, i: nat, lastT: nat)
requires isValidInput(x)
requires 0 ≤ lastT < i < |v|−1
requires consecutive_stops(lastT,i,v)
requires isSolution(x,v)
requires sum_distance(x.0[lastT..i+1]) ≤ x.1
requires v_modified = v[i:=false][i+1:=true]
ensures isSolution(x,v_modified)
```

In addition, these lemmas require other simpler lemmas, which we will not show here.

Once we reach the end of the loop in *reduction*, where we have already *greedy* = *optimal*; then we call *endReduction* method. This must ensure that the solution it returns (called *optimal′*) is equal to *greedy*; but as *optimal* (the solution it receives) is already equal to *greedy*, we just have to establish that *optimal′* is optimal. The only difference with the *files problem*, where we also only needed to say that *optimal′* was equal to *optimal*, is that we have to write some *asserts* to help Dafny prove all the postconditions. Let us see it:

```
ghost method endReduction(x: Input, greedy: Solution, optimal: Solution)
returns (optimal': Solution)
{
  optimal' := optimal;
  assert optimal'[0..|greedy|] = optimal' ∧ greedy[0..|greedy|] = greedy;
  assert value(x,optimal') ≤ value(x,greedy);
  assert isBetterSol(x,greedy,optimal);
}
```

4.3.3 Chairlift Problem

Another typical problem that can be solved applying a greedy algorithm is the *chairlift problem*. We have a group of people who want to go on a chairlift where at most two people can sit on a chair, each of which has a maximum weight allowance. We want to sit this people so that we can occupy as few chairs as possible.

The greedy strategy will be to order the people from heaviest to lightest and in this way we will sit together the heaviest person with the lightest person, considering only those who have not yet sat down. If they cannot go together, we will put the heaviest person alone in the chair. And as the new heaviest person of the ones remaining will be less heavy, then we will try to sit together the new heaviest person with the lightest person.

Our input parameters for the algorithm will be represented as in the sections before. We have a class *InputA*, where we have an array of natural numbers, which represent the weights of the people, and a natural number, which represents the maximum weight allowed by the chair.

The class *OutputA* will have an array *sol* of pairs of natural numbers, which represents the pairs formed, a variable *chairs* for the number of chairlifts we have already used and a variable *size*, that will let us know the size of our final solution. Regarding the array *sol*, let

us explain what each number represents. If, for example, we have a pair (1,7), it means that the person 1, which has as weight the value given at position 1 in the weight's sequence, is paired with person 7, which has as weight the value given at position 7. However, there are some cases where people can travel alone, perhaps because there are no other people left to travel with or because they are too heavy to be paired with someone else without going over the permitted weight. So now the question is: how do we represent this in Dafny? We have to write a datatype which allows us to omit a natural number in one of the components. This is the data type Maybe <T>:

```
datatype Maybe <T> = Nothing | Just (val:T)
```

But this is not the only difficulty in this problem. It also happens that we do not know the number of pairs of the solution in advance. For that reason the array *sol* is initially of the same length as the array of weights (because, in the worst case, we would need as many chairs as people we want to sit), and once we have already sat all people, we can reduce the length of the *sol* array to the one we really need. To do this we need a variable *size* that will tell us the number of pairs we have. Another characteristic that makes this problem different from the others is that this time we only need one variable in the *OutputA* apart from *sol*. In previous cases, we found *size* and *files*, or *size* and *stops*, for example, and they were not equal. On this occasion, *size* is enough, since *size* and *chairs* (which would be the other variable in this problem) are the same; the size of the solution represents the number of chairlifts we are going to use.

```
class InputA ... {
    var weight: array<nat>;
    var k:nat

    constructor (w:array<nat>,l:nat)
    {
      weight := w;
      k := l;
      Repr := {this,weight};
    }

    predicate Valid()
    {
      Repr = {this, this.weight}
    }
}

class OutputA ... {
    var sol: array<Elem>;
    var size: nat; //partial output

  constructor (x:InputA)
    requires x.Valid()
    ensures ∀ z | z in Repr :: fresh(z)
    ensures Valid(x)
    ensures sol.Length = x.weight.Length
    {
        sol := new Elem[x.weight.Length];
        Repr := {this,sol};
        chairs := 0;
        size := 0;
    }

  predicate Valid(x:InputA)
  {
```

```
    Repr = {this, sol} ∧
    sol.Length ≤ x.weight.Length ∧
    0 ≤ size ≤ sol.Length
  }
}
```

The type *Input* will be a pair where the first component is a sequence of naturals, which represents the weights of the people, and the second component is a natural number, which represents the maximum weight allowed by the chair. Type *Elem*, which is the type of the elements of the sequence *Solution*, is different this time. It is a pair where the first element is a natural (which represents the index of the sequence of weights of the person that is in the chairlift) and the second element is of type `Maybe<nat>`, which means it can be another natural value, in case there are two persons travelling in the chairlift, or *Nothing*, in case there is only one person.

```
type Input = (seq<nat>, nat)
type Elem = (nat, Maybe<nat>)
```

The idea for functions *inputTrans* and *outputTrans* is the same as we have already seen in the previous problems: to convert arrays into sequences. However, this time in *outputTrans* we have to convert only the part of the solution we want; this means, only until the variable *size*, because the rest is not needed.

```
function inputTrans(x:InputA) : Input
{
    (x.weight[..],x.k)
}

function outputTrans(x:InputA, y:OutputA) : seq<Elem>
{
    (y.sol)[..y.size]
}
```

This time we want to minimize the number of chairlift trips we have to do, then our objective function will be the length of our solution, which represents the number of pairs we have made. This is what we have to write in our *solutionValue*, which tells us that a solution is better than another when the returned value is smaller.

```
function solutionValue(x: Input, s: Solution): nat
{
    |s|
}
```

Regarding the *isValidInput* predicate, we must ensure the weights sequence is decreasingly sorted (this will be used to build the greedy solution) and that all weights of persons are below the maximum weight allowed on the chairlift (otherwise persons who do not comply with this would not be able to take the chairlift and there would be no solution).

```
predicate isValidInput(x: Input)
{
    sorted(x.0) ∧ smaller(x)
}
```

Let us focus now in the definition of solution. In this case we are going to define it using a generalized function *isSolutionG*, where we have as input parameter *index*, which represents the set of people who has to be sat.

```
// The generalised version when all persons are sat
predicate isSolution(x:Input, s:Solution)
ensures isSolution(x,s) ⟹ ∀ p | p in s :: 0 ≤ p.0 < |x.0|
ensures isSolution(x,s) ⟹ ∀ p | p in s ∧ p.1 ≠ Nothing :: 0 ≤ p.1.val < |x.0|
{
 isSolutionG((set i | 0 ≤ i < |x.0|), x, s)
}
```

The idea is similar as in the *gas station problem*, where we had a partial solution. In *isSolutionG* we have to ensure that the input is valid and also that *index* is a subset of the set of natural numbers that represents the people (between 0 and the length of the sequence of weights).

```
// Generalised version of isSolution where index represents the indices of people to seat
// If index = (set i | 0≤i<|weight|) we have a full solution
predicate isSolutionG(index:set<nat>, x:Input, s:Solution)
requires isValidInput(x)
requires index ≤ set n | 0 ≤ n < |x.0|
ensures isSolutionG(index,x,s) ⟹ ∀ p | p in s :: p.0 in index
ensures isSolutionG(index,x,s) ⟹ ∀ p | p in s ∧ p.1 ≠ Nothing :: p.1.val in index
ensures isSolutionG(index,x,s) ⟹ ∀ i | i in index ::
  (exists p :: p in s ∧ (p.0=i ∨ (p.1≠Nothing ∧ p.1.val=i)))
{
  |s| ≤ |x.0| ∧ firstSmaller(s) ∧
  sortedFirst(s) ∧ allPerson(index,s) ∧
  (∀ z | z in s ∧ z.1 ≠ Nothing :: (x.0)[z.0] + (x.0)[(z.1).val] ≤ x.1)
}
```

Let us see now what a solution is. First of all, the length of the solution must be smaller or equal than the length of the weight's sequence. Then, we must ensure that the first element of every pair is smaller than the second one (in case there is any) and that the first elements are sorted from smaller to greater; these two properties will help to compare solutions. Secondly, we have the predicate *allPerson*, which ensures that only people from the set *index* are seated and that each person appears only once, without repetitions. And finally, we must ensure that the sum of both weights sitting at each chair is under the maximum allowed.

The definition of *isSolutionG* satisfies the following properties: all elements in the solution must be in the set *index*, this includes all first elements of the pairs (this is represented in the first *ensures*) and all second elements of the pairs that are not *Nothing* (this is represented in the second *ensures*). All elements in *index* must be in the solution, and viceversa.

Let us see know the definition for greedy solution. As in the definition for solution, we will need a generalised version that has as input parameters the set *index*. The predicate *isSolGreedy* is defined using *isSolGreedyG* for the set of all persons.

```
// When the algorithm ends everybody is seated and i=j+1=|sol|
predicate isSolGreedy(x:Input, s:Solution)
{
  isSolGreedyG(set n | 0 ≤ n < |x.0|,x,s,|s|,|s|-1)
}
```

The idea for *isSolGreedyG* is similar to the one in *isSolutionG*. In this predicate we have to ensure that the input is valid. The input parameters i and j are the indices that tell us which persons we have already seated, this means that the people who have already seated are the ones from $[0..i) + (j..|weight|)$.

```
// Generalised version of isSolGreedy where index is the set of people already seated
// that we know are in [0..i)+[j+1..|weight|)
predicate isSolGreedyG(index:set<nat>, x:Input, sol:Solution, i:nat, j:int) //j may be -1
requires isValidInput(x) ∧ |sol| = i
requires 0 ≤ i ≤ j+1 ≤ |x.0|
requires index = (set n | 0 ≤ n < i) + (set n | j < n < |x.0|)
requires isSolutionG(index,x,sol)
{
  values(|x.0|,sol,i,j) ∧ theSmallest(|x.0|,sol,i) ∧
  theBiggest(|x.0|,sol,i) ∧ notPostpone(x,sol,i,j)
}
```

The *isSolGreedyG* predicate has to verify four different properties. First, that all elements in the solution are in the correct range (between $[0..i)$ and (j..|weight|)). Secondly, the first element of a pair is always the smallest index (which means the highest weight) of the remaining people. In third place, something similar but with the biggest index; the second element (when it is different from *Nothing*) is always the biggest index (which means the least weight) of the remaining people. And lastly, if we have a pair whose second element is *Nothing*, it means the first component cannot be seated with any remaining person. This is expressed by the predicate *notPostpone*.

As *theSmallest*, *theBiggest* and *notPostpone* predicates follow the same idea, let us explain just the predicate *notPostpone*. The approach is to check that all the people we have already seated who are alone cannot be paired with any of the people who are not yet seated, otherwise we could reduce the number of chairs to be used.

```
//Second Nothing component means first component cannot seat with any remaining person
predicate notPostpone(x:Input, sol:Solution, i:nat, j:int)
requires 0 ≤ i ≤ j + 1 ≤ |x.0| ∧ |sol| = i
requires values(|x.0|,sol,i,j)
{
    var allIndex := set n | 0 ≤ n < |x.0|;
    ∀ y | 0 ≤ y < i ∧ sol[y].1 = Nothing ::
            cannotSeat(x,sol[y].0,allIndex − set_of(sol[..y+1]))
}
```

Regarding the existence of a solution, as it is enough with finding the simplest solution, we decided to define the variable *sol*, where all the elements are of the form *(n, Nothing)*, which means a solution where every person seats alone in a chairlift. This way we ensure the property that we do not exceed the maximum allowed weight. We need an extra lemma *everyoneAlone*, which ensures that if the solution is as the one we have mentioned, then every person is seated.

```
lemma existsSolution(x:Input)
{
    var sol := seq(|x.0|, n ⇒ (n,Nothing));

    {everyoneAlone(x,(set i | 0≤i<|x.0|),sol,|sol|);}
    assert isSolution(x,sol);
}
```

```
lemma everyoneAlone(x: Input, index: set<nat>, sol: Solution, i: nat)
requires isValidInput(x)
requires |x.0| = |sol| ∧ ∀ j | 0 ≤ j < |x.0| :: sol[j] = (j, Nothing)
requires index = (set j | 0 ≤ j < i)
requires 0 ≤ i ≤ |sol|
ensures set_of(sol[..i]) = index;
{
    if (i = 0) {}
    else {
      assert sol[i−1] = (i−1,Nothing);
      assert set_of(sol[..i]) = set_of(sol[..i−1]) + {i−1};
      assert index = (set j | 0 ≤ j < i−1) + {i−1};
      {everyoneAlone(x,(set j | 0 ≤ j < i−1),sol,i−1);}
    }
}
```

Let us study now the algorithm that builds our solution. The principal idea is similar to previous problems, we traverse the array of weights and try to make as few pairs as possible. This time we will need two variables, *res.size* and j, that will go through the array; *res.size* starts at the beginning and j starts at the end of the array, so we check if the person with greatest weight can sit with the one with smallest weight. If the sum of the weights of both positions are smaller than the maximum allowed by the chairlift, then they sit together, so sol[*res.size*] = (*res.size*, j) and we advance both indices in their respective directions. Otherwise, the person in position *res.size* will sit alone, so sol[*res.size*] = (*res.size*, *Nothing*) and we only increase *res.size*.

The loop will finish once *res.size* $\geq j$. In case *res.size* $= j$, there is one person left, and it has to sit alone in the chairlift. In case *res.size* $> j$, all persons have been seated.

```
method greedy(x: InputA) returns (res: OutputA)
{
  var j := (x.weight).Length−1;
  res := new OutputA(x);
  res.size := 0;

  while (res.size < j)
    decreases j−(res.size)
    invariant 0 ≤ res.size ≤ j+1 ∧ (res.size)−1 ≤ j < (x.weight).Length
    invariant 0 ≤ res.sol.Length = (x.weight).Length
    invariant set_of((res.sol)[..res.size])
        = (set n | 0 ≤ n < res.size) + (set n | j < n < (x.weight).Length)
    invariant isSolutionG(set_of((res.sol)[..res.size]),inputTrans(x),(res.sol)[..res.size])
    invariant isSolGreedyG(set_of((res.sol)[..res.size]),
            inputTrans(x),(res.sol)[..res.size],res.size,j)
    //To be able to fill the solution array
    invariant ∀ z | z in res.Repr :: fresh(z)
    invariant res.Valid(x)
 {
    assert res.size < j;
    if (x.weight)[res.size] + (x.weight)[j] ≤ x.k { //they can go together
      (res.sol)[res.size] := (res.size, Just (j));
      restoreGreedy1(inputTrans(x),(res.sol)[..res.size],res.size,j);
      assert (res.sol)[..res.size+1] = (res.sol)[..res.size] + [(res.size,Just (j))];
      j := j−1;
    }
    else { //they cannot go together
      (res.sol)[res.size] := (res.size, Nothing);
      restoreGreedy2(inputTrans(x), (res.sol)[..res.size],res.size,j);
      assert (res.sol)[..res.size+1] = (res.sol)[..res.size]+[(res.size,Nothing)];
    }
    res.size := res.size + 1;
```

```
  }

  if (res.size=j) //there is one person left, it has to go alone in the chairlift
  {
    (res.sol)[res.size] := (res.size, Nothing);
    assert set_of((res.sol)[..res.size+1]) = set_of((res.sol)[..res.size]) + {res.size};
    assert (res.sol)[..res.size+1] = (res.sol)[..res.size] + [(res.size,Nothing)];
    restoreGreedy3(inputTrans(x), (res.sol)[..res.size],res.size,j);

    res.size := res.size + 1;
  }

  assert isSolution(inputTrans(x),outputTrans(x,res));
  assert isSolGreedy(inputTrans(x),outputTrans(x,res));
}
```

Let us study now all the invariants, lemmas and assertions that were needed to verify this method. We have two important invariants:

1. The people with indices between 0 and (not including) *res.size*, and those between $j+1$ and (not including) the number of people (which is the length of the array of weights, this is *x.weight.Length*) have already been sat.
2. The partial solution *res.sol* satisfies predicates *isSolutionG* and *isSolGreedyG* for that group of people.

Apart from this, we need three different lemmas used to restore the invariant in the three situations we can encounter: *restoreGreedy1*, *restoreGreedy2* and *restoreGreedy3*. The three of them receive a partial solution and return a new partial solution by adding a new pair. The first one, *restoreGreedy1*, corresponds to the case in which the algorithm adds a new pair with two persons that travel together in the chairlift. The second one, *restoreGreedy2*, corresponds to the case in which the algorithm adds a pair with just one person because it cannot be paired with anyone else. And the last one, *restoreGreedy3*, corresponds to the case in which the algorithm adds a pair with just one person because it is the only one left. Let us see just one example, the other two are almost identical.

In the *restoreGreedy1* lemma the purpose is to prove that if *sol* is a greedy partial solution and the persons i and j can sit together, then when we add the pair `(i, Just (j))` to *sol* we obtain a greedy solution. The proof is divided in three different parts, one for each of the properties which define the greedy solution.

```
lemma restoreGreedy1(x:Input, sol:Solution, i:nat, j:nat)
requires 0 ≤ i < j ≤ |x.0|−1 ∧ |sol|=i
requires isValidInput(x)
requires set_of(sol) = (set n | 0 ≤ n < i) + (set n | j < n < |x.0|)
requires isSolutionG(set_of(sol),x,sol)
requires isSolGreedyG(set_of(sol),x,sol,i,j)
requires (x.0)[i] + (x.0)[j] ≤ x.1
ensures isSolutionG(set_of(sol)+{i,j},x,sol+[(i,Just (j))])
ensures isSolGreedyG(set_of(sol)+{i,j},x,sol+[(i,Just (j))],i+1,j−1)
{
  assert values(|x.0|,sol+[(i,Nothing)],i+1,j);

  var allIndex := set n | 0 ≤ n < |x.0|;
  ∀ y | 0 ≤ y < i+1
```

```
    ensures isSmaller((sol + [(i,Just (j))])[y].0,
              allIndex - set_of((sol + [(i,Just (j))])[..y+1]))
  {
    if (y < i) {
      assert (sol+[(i,Just (j))])[..y+1] = sol[..y+1];
    }
    else {
      assert set_of(sol+[(i,Just (j))]) = set_of(sol) + {i,j};
      assert (sol+[(i,Just (j))])[y].0 = i;
      assert (sol+[(i,Just (j))])[..y+1] = sol + [(i,Just (j))];
      assert isSmaller(i,allIndex-set_of(sol+[(i,Just (j))]));
    }
  }

  ∀ y | 0 ≤ y < i+1 ∧ (sol+[(i,Just (j))])[y].1 ≠ Nothing
    ensures isBigger((sol + [(i,Just (j))])[y].1.val,
              allIndex - set_of((sol + [(i,Just (j))])[..y+1]))
  {
    if (y < i) {
      assert (sol+[(i,Just (j))])[..y+1] = sol[..y+1];
    }
    else {
      assert (sol+[(i,Just (j))])[..y+1] = sol + [(i,Just (j))];
      assert set_of(sol+[(i,Just (j))]) = set_of(sol) + {i,j};
      assert set_of(sol+[(i,Just (j))]) = (set n | 0 ≤ n < i+1) + (set n | j ≤ n < |x.0|);
    }
  }

  ∀ y | 0 ≤ y < i+1 ∧ (sol+[(i,Just (j))])[y].1 = Nothing
    ensures cannotSeat(x,(sol + [(i,Just (j))])[y].0,
              allIndex-set_of((sol + [(i,Just (j))])[..y+1]))
  {
    if (y < i) {
      assert (sol+[(i,Just (j))])[..y+1] = sol[..y+1];
      assert (sol+[(i,Just (j))])[y].0 = sol[y].0;
    }
    else {}
  }
}
```

Once we have our algorithm working well and verified, we must prove that the greedy solution is optimal. We have to apply the *exchange proof*. We assume that $greedy[..i] = optimal[..i]$. The lemma *sameInI* allows us to know that also $greedy[i].0 = optimal[i].0$. This is true because as the first i elements are the same in both solutions ($greedy[..i] = optimal[..i]$), we know that the people who have not yet been seated in both cases are the same. As in both solutions (*greedy* and *sol*) the first component is the heaviest one and the first components are decreasingly sorted, this person will be the same one in both cases.

```
lemma sameInI(x: Input, sol1: Solution, sol2: Solution, i: nat)
requires isValidInput(x)
requires 0 ≤ i < |sol1| ≤ |sol2|
requires isSolution(x,sol1) ∧ isSolution(x,sol2)
requires sol1[..i] = sol2[..i]
ensures sol1[i].0 = sol2[i].0

ghost method reduction_step(x: Input, greedy: Solution, optimal: Solution, i: nat)
returns (optimal': Solution)
{
  var allIndex := set n | 0 ≤ n < |x.0|;
  assert set_of(greedy[i..]) = allIndex-set_of(greedy[..i]);
  assert set_of(optimal[i..]) = allIndex-set_of(optimal[..i]);
  assert set_of(greedy[i..]) = set_of(optimal[i..]);

  sameInI(x,optimal,greedy,i);
```

```
  assert greedy[i].0 = optimal[i].0 ∧ greedy[i].1 ≠ optimal[i].1;
  (...)
}
```

Then, the possible cases are:

1. *greedy*$[i].1 =$ *Nothing* (and *optimal*$[i].1 \neq$ *Nothing*)
2. *greedy*$[i].1 \neq$ *Nothing* and *optimal*$[i].1 =$ *Nothing*. In this case there exists $z > i$ such that
   (a) either *greedy*$[i].1 =$ *optimal*$[z].0$
   (b) or *greedy*$[i].1 =$ *optimal*$[z].1$
3. *greedy*$[i].1 \neq$ *Nothing* and *optimal*$[i].1 \neq$ *Nothing*. In this case there exists $z > i$ such that
   (a) either *greedy*$[i].1 =$ *optimal*$[z].0$
   (b) or *greedy*$[i].1 =$ *optimal*$[z].1$

because all the people has a seat.

The first situation, where *greedy*$[i].1 =$ *Nothing* $\neq$ *optimal*$[i].1$, is not possible because it contradicts the definition of greedy solution. The definition says that if a person travels alone on the chairlift that means that he/she cannot be paired with any of the remaining people. However, in the optimal solution *greedy*$[i].0 =$ *optimal*$[i].0$ is paired with *optimal*$[i].1$, which is one of the remaining people. In Dafny, first we need to prove that there exists a position z (from i to the end) in the greedy solution where one of the persons of the pair is *optimal*$[i].1$. Lemma *existsZ* proves this. So, we have that *greedy*$[z].0 =$ *optimal*$[i].1$ or *greedy*$[z].1 =$ *optimal*$[i].1$; and we let Dafny know that the sum of weights of *greedy*$[i].0$ and *greedy*$[z].0$ (or *greedy*$[z].1$) is smaller than the maximum allowed by the chairlift, so they could sit together.

```
lemma existsZ(x: Input, sol1: Solution, sol2: Solution, i: nat, j: nat)
requires isValidInput(x)
requires isSolution(x, sol1)
requires isSolution(x, sol2)
requires 0 ≤ i < |sol2| ∧ 0 ≤ i < |sol1| ∧ 0 ≤ j < |x.0|
requires sol1[i].1 ≠ Nothing ∧ j = sol1[i].1.val
requires sol2[..i] = sol1[0..i] ∧ sol1[i].0 = sol2[i].0 ∧ sol1[i].1 ≠ sol2[i].1
ensures exists z :: i < z < |sol2| ∧ ((sol2[z].0 = j) ∨ sol2[z].1 = Just(j))
```

Let us see now the first part of the proof, the one we just explained:

```
if (greedy[i].1 = Nothing) //∧ optimal[i].1 ≠ Nothing
{//Impossible

    assert x.0[greedy[i].0] + x.0[optimal[i].1.val] ≤ x.1;

    var j := optimal[i].1.val;
    existsZ(x,optimal,greedy,i,j);
```

```
    var z :| i < z < |greedy| ∧ (greedy[z].0 = j ∨ greedy[z].1 = Just (j));
    assert (x.0[greedy[i].0] + (x.0)[greedy[z].0] ≤ x.1) ∨
      (greedy[z].1 = Just (j) ∧ (x.0)[greedy[i].0] + (x.0)[greedy[z].1.val] ≤ x.1);
    assert !isSolGreedy(x,greedy); //contradicts the precondition
}
```

In the second case, $greedy[i].1 \neq Nothing = optimal[i].1$; in this situation we can distinguish another two cases: when $greedy[i].1$ is the first element of a pair of optimal or when it is the second one. In case $greedy[i].1 = optimal[z].0$, we have to prove first that $optimal[z].1 \neq Nothing$. We prove it by contradiction because if it were *Nothing* we could omit the pair $optimal[z]$ and replace $optimal[i]$ by $(optimal[i].0,\ optimal[z].0)$, which contradicts the fact that it is optimal. Then we have to change the pair in $optimal[i]$ from $(optimal[i].0,\ Nothing)$ to $(optimal[i].0,\ optimal[z].0)$ and the pair in $optimal[z]$ from $(optimal[z].0,\ optimal[z].1)$ to $(optimal[z].1,\ Nothing)$. This is done with the method *swapFirstSecond*.

Let us see now the other case, when $greedy[i].1 = optimal[z].1$; this time we call the method *move* that changes the pair $optimal[i]$ from $(optimal[i].0,\ Nothing)$ to $(optimal[i].0,\ optimal[z].1)$ and the pair $optimal[z]$ from $(optimal[z].0,\ optimal[z].1)$ to $(optimal[z].0,\ Nothing)$.

This proof is represented in Dafny as follows:

```
else {
    var j := greedy[i].1.val;
    existsZ(x,greedy,optimal,i,j);
    var z :| i < z < |optimal| ∧ (optimal[z].0 = j ∨ optimal[z].1 = Just (j));

    if (optimal[i].1 = Nothing) {
      if (optimal[z].0 = j) {
        //Prove first that optimal[z].1 ≠ Nothing by contradiction
        if optimal[z].1 = Nothing {
          lessChairlift(x,optimal,i,z);
          assert !isSolOptimal(x,optimal);
        }
        else {
          optimal' := swapFirstSecond(x,optimal,i,z);
          assert optimal'[0..i+1] = optimal'[0..i]+[optimal'[i]];
        }
      }
      else { //optimal[z] = (_,j)
        optimal' := move(x,optimal,i,z);
        assert optimal'[0..i+1] = optimal'[0..i]+[optimal'[i]];
      }
}
```

Let us now look at the methods *swapFirstSecond* and *move*, needed in this part of the poof. Earlier we mentioned that in this second case we have studied, both positions $optimal[i]$ and $optimal[z]$ must be modified. However, in the postconditions of our methods, the only changes that are ensured are the ones in $optimal[i]$ and not in $optimal[z]$. This is because changes in position z are only relevant for the purpose of staying as a solution without getting worse.

```
ghost method swapFirstSecond(x: Input, v: Solution, i: nat, z: nat) returns (v_modified: Solution)
requires isValidInput(x)
requires isSolution(x,v)
requires 0 ≤ i < z < |v| ≤ |x.0|
```

```
requires v[i].1 = Nothing ∧ v[z].1 ≠ Nothing
requires v[i].0 < v[z].0
requires (x.0)[v[i].0] + (x.0)[v[z].0] ≤ x.1
ensures |v_modified| = |v| ∧ v_modified[..i] = v[..i] ∧ v_modified[i] = (v[i].0, Just (v[z].0))
ensures isSolution(x, v_modified)
ensures isBetterSol(x, v_modified, v)//they are equal

ghost method move(x: Input, v: Solution, i: nat, z: nat) returns (v_modified: Solution)
requires isValidInput(x)
requires isSolution(x,v)
requires 0 ≤ i < z < |v|≤|x.0|
requires v[i].1 = Nothing ∧ v[z].1 ≠ Nothing
requires v[i].0 < v[z].1.val
requires (x.0)[v[i].0] + (x.0)[v[z].1.val] ≤ x.1
ensures |v_modified| = |v| ∧ v_modified[..i] = v[..i] ∧ v_modified[i] = (v[i].0,v[z].1)
ensures isSolution(x, v_modified)
ensures isBetterSol(x, v_modified, v) //they are equal
```

Finally, the last situation that we may encounter is when $greedy[i].1 \neq Nothing \neq optimal[i].1$. We must look for $greedy[i].1$ in $optimal[i..]$ and, as before, there are two different situations: if $greedy[i].1$ is the first component of a pair in the optimal solution or if it is the second component of a pair. The case $greedy[i].1 = optimal[z].0$ is quite similar to the case 2(a), but this time we must prove that $optimal[z].1 = Nothing$ by contradiction, and then we must change our solution optimal the following way. The pair $optimal[i]$ changes from $(optimal[i].0, optimal[i].1)$ to $(optimal[i].0, optimal[z].0)$ and the pair $optimal[z]$ changes from $(optimal[z].0, Nothing)$ to $(optimal[i].1, Nothing)$; all these changes are made by calling the method *swapSecondFirst*.

Let us see now the other case, when $greedy[i].1 = optimal[z].1$; this time we call the method *swapSecond* that changes the pair $optimal[i]$ from $(optimal[i].0, optimal[i].1)$ to $(optimal[i].0, optimal[z].1)$ and the pair $optimal[z]$ from $(optimal[z].0, optimal[z].1)$ to $(optimal[z].0, optimal[i].1)$.

This proof is represented in Dafny as follows:

```
else { //optimal[i].1 ≠ Nothing
    if (optimal[z].0 = j) {
        //Prove first that optimal[z].1 = Nothing by contradiction
        assert ∀ n | n in allIndex−set_of(greedy[..i+1]):: greedy[i].1.val > n;
        assert greedy[..i+1] = greedy[..i] + [greedy[i]];
        assert set_of(greedy[..i+1]) = set_of(greedy[..i]) + {greedy[i].0, greedy[i].1.val};
        if optimal[z].1 ≠ Nothing {
          assert optimal[z].1.val in allIndex − set_of(greedy[..i+1]);
          assert greedy[i].1.val = optimal[z].0 < optimal[z].1.val;
          assert !isSolution(x, optimal);
        }
        else {
          optimal' := swapSecondFirst(x, optimal, i, z);
          assert optimal'[0..i+1] = optimal'[0..i] + [optimal'[i]];
        }
    }
    else {
        optimal' := swapSecond(x, optimal, i, z);
        assert optimal'[0..i+1] = optimal'[0..i] + [optimal'[i]];
    }
}
```

Let us now look at the methods *swapSecondFirst* and *swapSecond*, needed in this part of

the proof.

```
ghost method swapSecondFirst(x: Input, v: Solution, i: nat, z: nat) returns (v_modified: Solution)
requires isValidInput(x)
requires isSolution(x,v)
requires 0 ≤ i < z < |v| ≤ |x.0|
requires v[i].1 ≠ Nothing ∧ v[z].1 = Nothing
requires v[i].0 < v[z].0
requires (x.0)[v[i].0] + (x.0)[v[z].0] ≤ x.1
ensures |v_modified| = |v| ∧ v_modified[..i] = v[..i] ∧ v_modified[i] = (v[i].0, Just (v[z].0))
ensures isSolution(x,v_modified)
ensures isBetterSol(x,v_modified,v) //they are equal

ghost method swapSecond(x: Input, v: Solution, i: nat, z: nat) returns (v_modified: Solution)
requires isValidInput(x)
requires isSolution(x,v)
requires 0 ≤ i < z < |v|≤|x.0|
requires v[i].1 ≠ Nothing ∧ v[z].1 ≠ Nothing
requires v[i].0 < v[z].1.val
requires (x.0)[v[i].0] + (x.0)[v[z].1.val] ≤ (x.1)
ensures |v_modified| = |v| ∧ v_modified[..i] = v[..i] ∧ v_modified[i] = (v[i].0,v[z].1)
ensures isSolution(x,v_modified)
ensures isBetterSol(x,v_modified,v) //they are equal
```

We can realise that at the end of all these cases (except the first where we do not make any change in our optimal solution) in position i of the optimal solution we have a pair equal to the one we found in position i of the greedy solution, which was our main objective. Obviously we have to ensure that all these changes we make are allowed, this means, the maximum weight allowed by the chairlift is still respected, but this is something we had to make sure in each of the methods we call to modify our optimal solution.

In the *endReduction* method we have to return the optimal solution, but this time we need an extra lemma to let Dafny know that the optimal and greedy solutions are equal. This lemma is called *sameSol*. This lemma proves that if two solutions verify that $sol1 = sol2\,[..|sol1|]$, then $sol1 = sol2$.

```
ghost method endReduction(x: Input, greedy: Solution, optimal: Solution)
                          returns (optimal': Solution)
{
    optimal' := optimal;
    sameSol(x,optimal',greedy);
    assert optimal' = greedy;
    assert isBetterSol(x,greedy,optimal);
}
```

Let us study now the functions that modify the optimal solution, which are invoked from methods *swapFirstSecond*, *swapSecondFirst*, *swapSecond* and *move*. We use three functions to modify a solution, in our case the optimal solution: to insert a new pair, to delete a pair and to replace a pair. We will focus on the *insert* function as it is the most complex, however the other two are quite similar regarding the specifications. We want to return a solution by adding a new pair, p. Then we have to make sure that all the properties of being a solution hold, so we cannot insert this new pair wherever we want. To respect the order we define a recursive function that also returns the right position where to insert p.

The preconditions must ensure the following properties:

1. *sol* is a solution
2. the components of the pair p are not already in the set *index*; otherwise, it would be already in the solution, and the pair p verifies that the sum of the weights is smaller than the maximum allowed.
3. the parameter i represents the index we are looking to insert the pair or not. Then for all the pairs previous to i, it must hold that the first component of the pair p is greater than all the first components of the previous pairs.

Regarding the postconditions, we must ensure that we return a solution with the pair inserted in the right place.

```
//Inserts a valid pair with components that were not in the sol.
//It returns the new solution and the position where it was inserted.
function insert(index:set<nat>, x:Input, p:Elem, sol:Solution, i:nat):(Solution,nat)
requires isValidInput(x) ∧ |sol| < |x.0|
requires index ≤ set n | 0 ≤ n < |x.0|
requires isSolutionG(index,x,sol)
requires 0 ≤ p.0 < |x.0| ∧ p.0 ∉ index
requires p.1 ≠ Nothing ⟹ p.1.val ∉ index ∧ 0 ≤ p.0 < p.1.val < |x.0| ∧
         (x.0)[p.0] + (x.0)[(p.1).val] ≤ x.1
requires 0 ≤ i ≤ |sol|
requires ∀ z | 0 ≤ z < i :: sol[z].0 < p.0
ensures |insert(index,x,p,sol,i).0| = 1 + |sol|
ensures 0 ≤ insert(index,x,p,sol,i).1 < |insert(index,x,p,sol,i).0| ∧
        insert(index,x,p,sol,i).0[insert(index,x,p,sol,i).1] = p
ensures ∀ z | 0 ≤ z < insert(index,x,p,sol,i).1 :: sol[z].0 < p.0
ensures ∀ z | insert(index,x,p,sol,i).1 ≤ z < |sol| :: sol[z].0 > p.0
ensures insert(index,x,p,sol,i).0[..insert(index,x,p,sol,i).1] =
        sol[..insert(index,x,p,sol,i).1]
ensures insert(index,x,p,sol,i).0[insert(index,x,p,sol,i).1+1..] =
        sol[insert(index,x,p,sol,i).1..]
ensures p.1 = Nothing ⟹ isSolutionG(index+{p.0},x,insert(index,x,p,sol,i).0)
ensures p.1 ≠ Nothing ⟹ isSolutionG(index+{p.0,p.1.val},x,insert(index,x,p,sol,i).0)
decreases |sol| − i
{
  if i = |sol| then
    assert p.1 = Nothing ⟹ isSolutionG(index+{p.0},x,sol+[p]);
    assert p.1 ≠ Nothing ⟹ |sol + [p]| ≤ |x.0|;
    assert p.1 ≠ Nothing ⟹ isSolutionG(index+{p.0,p.1.val},x,sol+[p]);
    (sol + [p],i)
  else
    assert 0 ≤ i < |sol|;
    if sol[i].0 > p.0 then
      (sol[..i] + [p] + sol[i..], i)
    else
      insert(index,x,p,sol,i+1)
}
```

Let us see how *swapSecondFirst* is implemented using these functions: *insert*, *delete* and *replace*, so that we can get a more clear idea of how all these methods work. We must emphasise that it is important to know in which order to apply the functions so that it is still a solution. This method receives a solution v, and two indices i and z. The idea is to change from our solution the pair $optimal[i]$ from ($optimal[i].0$, $optimal[i].1$) to ($optimal[i].0$, $optimal[z].0$) and the pair $optimal[z]$ from ($optimal[z].0$, *Nothing*) to ($optimal[i].1$, *Nothing*). So, the first thing we have to do is delete the pair in z, $optimal[z]$; so the function *delete* returns a partial solution. Then, we replace the pair in i and finally we insert the new pair ($optimal[i].1$,*Nothing*) in the correct place.

```
ghost method swapSecondFirst(x: Input, v: Solution, i: nat, z: nat) returns (v_modified: Solution)
requires isValidInput(x)
requires isSolution(x,v)
requires 0 ≤ i < z < |v| ≤ |x.0|
requires v[i].1 ≠ Nothing ∧ v[z].1 = Nothing
requires v[i].0 < v[z].0
requires (x.0)[v[i].0] + (x.0)[v[z].0] ≤ x.1
ensures |v_modified| = |v| ∧ v_modified[..i] = v[..i] ∧ v_modified[i] = (v[i].0, Just (v[z].0))
ensures isSolution(x, v_modified)
ensures isBetterSol(x, v_modified, v)//they are equal
{
    var allIndex := set n | 0 ≤ n < |x.0|;
    v_modified := delete(allIndex, x, v[z], z, v);
    var index := allIndex − {v[z].0};
    assert v_modified[..z] = v[..z];

    v_modified := replace(index, x, v_modified, i, Just (v[z].0));
    index := index − {v[i].1.val} + {v[z].0};
    assert v_modified[..i] = v[..i] ∧ v_modified[i] = (v[i].0, Just (v[z].0));

    var pair := insert(index, x, (v[i].1.val, Nothing), v_modified, 0);
    index := index + {v[i].1.val};

    assert i < pair.1 ≤ |v_modified| ≤ |pair.0| ;
    assert pair.0[..pair.1] = v_modified[..pair.1];
    equalSequences(v_modified, pair.0, pair.1, i);
    assert pair.0[..i] = v_modified[..i] = v[..i] ∧
           pair.0[i] = v_modified[i] = (v[i].0, Just (v[z].0));

    v_modified := pair.0;
    assert isSolutionG(allIndex, x, v_modified);
}
```

Let us give an example of how these reduction steps are applied. Let us imagine that we have a group of people with the following weights: [10,5,4,2,1], so this is our sequence $x.0$ sorted in descending order. And the maximum value allowed in the chairlift is 15, so this is our constant $x.1$. Then, following the definition of greedy solution, $greedy = [(0,4),(1,3),(2,Nothing)]$, knowing that the indices of the solution are the positions and not the weights. This means, that the person with weight 10 is paired with the one with weight 1, for example.

An optimal solution is $optimal = [(0,Nothing),(1,2),(3,4)]$. Checking both solutions and comparing the first elements of both, we find ourselves in the case *2(b)* where $i = 0$ and $z = 2$, because $greedy[0].1 \neq Nothing = optimal[0].1$ and $greedy[0].1 = optimal[2].1$. This means that we must first apply the method *move*, which exchanges positions $optimal[0].1$ and $optimal[2].1$. Now, our optimal solution will be a partial optimal solution that looks like $optimal = [(0,4),(1,2),(3,Nothing)]$; and in position i our both solutions, *greedy* and *optimal*, are equal.

Then, we move one position forward to $i = 1$. This time, $greedy[1].1 \neq Nothing \neq optimal[1].1$ and $greedy[1].1 = optimal[2].0$, so $z = 2$ and we are in the case *3(a)*, where we have to apply the method *swapSecondFirst*. As we have the code of this method just above, let us explain this case in a little more detail. First of all, we must call the function *delete*, which deletes the pair in z and we obtain a partial solution, that in this example will look like $[(0,4),(1,2)]$. After this, it calls the function *replace*, which replaces the pair in i from $[(0,4),(1,2)]$ to $[(0,4),(1,3)]$ and this way we already obtained that *greedy[i] = optimal[i]*.

At the end, we must add a new pair so we have again a solution and not a partial solution. So, the method calls the function *insert*, which inserts the pair $(2, Nothing)$ and we obtain the optimal solution $[(0, 4), (1, 3), (2, Nothing)]$. And so we end up with an optimal solution which is equal to the greedy one.

4.3.4 Hose Problem

Let us present our last problem: *hose problem*. We have a hose with holes and we have to cover up these holes so that the hose works properly. We have some patches available, all of them of the same length, and the idea is to minimize the number of patches that we need to fix the hose by covering all the holes.

The greedy strategy is the following: we traverse the holes in order from one end of the hose, and we add a new patch on the first hole that is not covered by the patches already placed on the hose.

First of all, we have to study the types of the algorithm. We have to represent the input parameters in the *InputA* class: *holes* will be an array of natural numbers, which represents the positions of the holes from one end of the hose; and *l_patch* is a natural number that represents the length of the patches. In the *OutputA* class we have a variable for the solution, *sol*, which represents the positions of the patches; another variable for the total number of patches, *patches*; and the size of a partial solution, *size*. As in the previous problem, we do not know the number of patches in advance. For that reason, the array *sol* starts with the same length as the array *holes*, which really represents the number of holes there are. A valid solution might have more patches than holes but none of those can be optimal, so we will not consider them. That is why we initially put the same length as the array *holes* and not a greater one.

```
class InputA ... {
   var holes: array<nat>;
   var l_patch: nat;

   constructor (h: array<nat>,p: nat)
   {
      holes := h;
      l_patch := p;
      Repr := {this, holes};
   }

   predicate Valid()
   {
      Repr = {this, this.holes}
   }
}

class OutputA ... {
   var sol: array<Elem>;
   var patches: nat;
   var size: nat;

   constructor (x: InputA)
   requires x.Valid()
   ensures ∀ z | z in Repr :: fresh(z)
```

```
    ensures Valid(x)
    {
        sol := new Elem[x.holes.Length];
        Repr := {this,sol};
        patches := 0;
        size := 0;
    }

    predicate Valid(x:InputA)
    {
        Repr = {this, sol} ∧
        sol.Length = x.holes.Length ∧
        0 ≤ size ≤ sol.Length ∧
        patches = size
    }
}
```

The types of the verification part are, as in the previous problems, equal but instead of arrays, we have now sequences. We have the type *Input*, which is a pair of a sequence of naturals, which represents the holes in the hose, and a natural number, which represents the length of the patches. Then, as our solution is the position where we are going to put the patches, the type *Elem* is a natural number, so a solution is a sequence of natural numbers.

```
type Input = (seq<nat>, nat)
type Elem = nat
```

Functions *inputTrans* and *outputTrans* are like the ones we have already seen, the aim is to convert an element from the class *InputA* to the type *Input*, or from the class *OutputA* to the type *Solution*. In the *outputTrans*, as in the *chairlift problem*, we only want to keep a subsequence of the sequence *sol*. This is why we have variable *size*, to know until what position we want to maintain. This is represented in Dafny as follows:

```
function inputTrans(x: InputA) : Input
{
    (x.holes[..],x.l_patch)
}

function outputTrans(x: InputA, y: OutputA) : Solution
{
    (y.sol)[..y.size]
}
```

As we have mentioned before, we want to minimize the number of patches we use. So, our objective function will be the length of the solution. Then, our *solutionValue* function will return a natural number, and the smaller it is, the better the solution will be.

```
function solutionValue(x: Input, s: Solution): nat
{
    |s|
}
```

Regarding the *isValidInput* predicate, we must ensure two properties: first of all, we need to have holes in the hose; otherwise, the problem would not have any sense. This is the type of restrictions we need to guarantee so there exists a solution. Secondly, we need to ensure that the sequence of the holes is strictly sorted (*ssorted*), which means that is increasingly

sorted and does not have repetitions, because it does not make sense to have more than one hole in the same position.

```
predicate isValidInput(x: Input)
{
    |x.0| > 0 ∧ ssorted(x.0)
}
```

Let us start now with the predicates. First of all, the predicate *isSolution* must ensure two properties: the solution is sorted and all the holes are covered with a patch. This second property is ensured thanks to the predicate *covered*, which must be verified for each hole. So this predicate receives as input parameter, apart from the data input and the solution, a position of a hole and ensures that there exists a position in the solution that covers this hole.

```
predicate isSolution(x:Input, s:Solution)
{
    sorted(s) ∧ (∀ i | i in x.0 :: covered(x,s,i))
}

predicate covered(x:Input, s:Solution, i:nat)
{
    exists j :: j in s ∧ (j ≤ i) ∧ (j + x.1 ≥ i)
}
```

Regarding the predicate *isSolGreedy*, we must ensure first that the patches start where there is a hole, so for all the elements in the solution they must be in the sequence of the holes. Additionally, in a greedy solution there is no hole covered by more than one patch, otherwise we could reduce the number of patches and the greedy solution would not be optimal.

```
predicate isSolGreedy(x:Input, s:Solution)
{
    (∀ i | i in s :: i in x.0) ∧
    (∀ i,j | 0 ≤ i < j < |s| :: s[i] + x.1 < s[j])
}
```

Before explaining the algorithm, we must prove the existence of a solution. This time, the proof of this lemma is simpler than in previous problems, as we just need to define a solution where we put patches on each hole. In this case we do not need an extra lemma to prove the properties of a solution, Dafny by itself can prove that every hole is covered and that the solution is sorted, as $x.0$ is also sorted.

```
lemma existsSolution(x:Input)
{
    var sol := x.0;
    assert isSolution(x,sol);
}
```

The algorithm of this problem will follow the strategy we mentioned before. The idea is to go through the array with the positions of the holes and check if a new patch is necessary or not. For this we use a new variable called *isCovered*, that represents until which position we cover with the last patch we have placed. If the current hole is covered by this last patch,

then we just advance to the next hole. Otherwise, if the current hole is not covered by the last patch, we put a new patch in our solution, we increase by one the size of the solution and the number of patches and we update the variable *isCovered*. This algorithm is implemented in Dafny as follows:

```
method greedy(x: InputA) returns (res: OutputA)
{
    res := new OutputA(x);
    res.sol[0] := x.holes[0];
    res.patches := 1;
    res.size := 1;
    var i := 1;
    var isCovered := res.sol[0] + x.l_patch;

    while (i < x.holes.Length)
      decreases (x.holes).Length - i
      invariant 0 < res.size ≤ i ≤ x.holes.Length = res.sol.Length
      invariant isCovered = res.sol[res.size-1] + x.l_patch
      //isSolution
      invariant sorted(res.sol[..res.size])
      invariant res.sol[res.size-1] in res.sol[..res.size]
      invariant res.sol[res.size-1] ≤ x.holes[i-1] ∧ isCovered ≥ x.holes[i-1]
      invariant ∀ m | m in x.holes[..i] :: covered(inputTrans(x),res.sol[..res.size],m)
      invariant isSolution((x.holes[..i],x.l_patch),res.sol[..res.size])
      invariant isSolGreedy((x.holes[..i],x.l_patch),res.sol[..res.size])
      //To be able to fill the solution array
      invariant ∀ z | z in res.Repr :: fresh(z)
      invariant res.Valid(x)
      {
        if x.holes[i] > isCovered { //We need a new patch
          res.sol[res.size] := x.holes[i];
          res.size := res.size + 1;
          res.patches := res.patches + 1;
          isCovered := res.sol[res.size-1] + x.l_patch;

          ∀ m | m in x.holes[..i+1]
           ensures covered(inputTrans(x),res.sol[..res.size],m)
           {
            if (m in x.holes[..i]) {
              assert covered(inputTrans(x),res.sol[..res.size-1],m);
              var j :| j in res.sol[..res.size-1] ∧ (j ≤ m) ∧ (j + inputTrans(x).1 ≥ m);
              assert j in res.sol[..res.size];
              assert covered(inputTrans(x),res.sol[..res.size],m);
            }
            else {
              assert x.holes[i] in res.sol[..res.size];
              //assert covered(inputTrans(x),res.sol[..res.size],x.holes[i]);
            }
           }
      }
      i := i+1;
    }

    assert x.holes[..i] = x.holes[..];

    assert isSolution(inputTrans(x),outputTrans(x,res));
    assert isSolGreedy(inputTrans(x),outputTrans(x,res));
}
```

This time the verification part did not need any lemmas, just with some invariants and assertions it was possible to prove that the solution returned is greedy.

Let us study now the correctness of this solution. We must verify that the greedy solution is also optimal. As in all previous problems, we apply the *exchange proof*. We have to compare

both solutions, and when we find a position i where $greedy[i] \neq optimal[i]$ the code calls the method *reduction_step*.

As in this problem, the elements of a solution are natural numbers. The possible situations that we have when $greedy[i] \neq optimal[i]$ are:

1. $greedy[i] < optimal[i]$
2. $greedy[i] > optimal[i]$

However, as in the previous problems, the first case is not possible. As we know that $greedy[..i] = optimal[..i]$, if $greedy[i] < optimal[i]$ then either $greedy[i]$ is not covered by *optimal* or $greedy[i]$ is not a position of a hole and then *greedy* would not verify all the properties of a greedy solution. In the second case, we must change the position i in the optimal solution from $optimal[i]$ to $greedy[i]$ and prove that this returns an optimal solution that has not become worse. The idea is that if in the greedy solution we do not have the element $optimal[i]$ it is because there is no hole there.

Let us see the *reduction_step*, where all of this is done:

```
ghost method reduction_step(x: Input, greedy: Solution, optimal: Solution, i: nat)
                            returns (optimal': Solution)
{
    if (greedy[i] < optimal[i])
    { //Impossible
      assert !covered(x,optimal[..i],greedy[i]);
      assert !covered(x,optimal,greedy[i]);
      assert greedy[i] ∉ x.0;

      assert !isSolGreedy(x,greedy);
    }
    else
    { //postpone where I put the patch
      optimal' := postpone(x,optimal,i,greedy[i]);
    }
}
```

Once we get that $greedy = optimal$, after checking every position to ensure that they are equal, we call *endReduction* method. We have already seen this methods previously, and in this problem is pretty similar to the one in *gasStation* problem. This method must ensure that the solution it returns is equal to the greedy one it receives. Let us see it:

```
ghost method endReduction(x: Input, greedy: Solution, optimal: Solution)
                          returns (optimal': Solution)
{
    optimal' := optimal;
    sameSol(x,optimal',greedy);
    assert optimal' = greedy;
    assert isBetterSol(x,greedy,optimal);
}
```

As the optimal solution this method receives is already equal to the greedy solution, we just have to establish that *optimal'* is equal to *optimal*. However, to ensure that *optimal'*

and *greedy* are equal, we need an extra lemma *sameSol*. Thanks to the preconditions of *endReduction*, we know that $|sol1| \leq |sol2|$ and $sol1 = sol2\,[..|sol1|]$. Knowing that, we can ensure that $sol1 = sol2$ because if $|sol1| < |sol2|$, there would be some positions in the hose that would not be covered by *sol1*.

```
lemma sameSol(x: Input, sol1: Solution, sol2: Solution)
requires isValidInput(x)
requires isSolution(x,sol1) ∧ isSolution(x,sol2)
requires isSolOptimal(x,sol1) ∧ isSolGreedy(x,sol2)
requires |sol1| ≤ |sol2|
requires sol1 = sol2[..|sol1|]
ensures |sol1| = |sol2|
ensures sol1 = sol2
{
    if |sol1| = |sol2| {}
    else {
      if (∀ i,j | 0 ≤ i < j < |sol2| :: sol2[i] + x.1 < sol2[j]) {
        assert ∀ i | i in x.0 :: covered(x,sol2[..|sol1|],i);
        assert ∀ i | i in x.0 :: !covered(x,sol2[|sol1|..],i);
        assert ∀ i | i in sol2[|sol1|..] :: i ∉ x.0;
        assert sol2[|sol1|] in sol2;
        assert exists i :: i in sol2 ∧ i ∉ x.0;
        //assert !(∀ m | m in sol2 :: m in x.0);
        //assert !isSolGreedy(x,sol2);
      }
      else {
        //assert !isSolGreedy(x,sol2);
      }
    }
}
```

In the second case of *reduction_step*, where we have to postpone a patch, we call the method *postpone*. Let us study now this. As we have mentioned, we have the optimal solution where $optimal[i] < greedy[i]$ and we want to change $optimal[i]$ to $greedy[i]$ because we know that all the holes in $[optimal[i]..greedy[i]]$ are covered by $optimal[i-1] = greedy[i-1]$. But, of course, we have to make sure that the returned solution is optimal and better than the original optimal solution. This method receives the position i that we want to change, and the value j, which is the new value that we want to put in i. We will return *v_modified*. The idea is to ensure both properties that a solution must verify: firstly, all the positions that were covered by $optimal[i]$ are still covered after the change; and secondly, that it is sorted. To guarantee the first property we use a new lemma, *changePos*. To make sure that *v_modified* is sorted, we created a loop so that all the positions after i with a value smaller than j are changed to j.

Here is an example where this situation may happen. The *Input* values are $x.0 = [3, 4, 6, 9, 15, 20, 22]$ (the positions where we can find holes) and $x.1 = 3$ (the length of the patches). Our greedy solution will be $[3, 9, 15, 20]$ and an optimal solution could be $[3, 6, 15, 20]$. In this example, it holds that $optimal[1] < greedy[1]$, so we call the method *postpone*. The value for i is 1 and the value for j is 9, because we want to return our optimal solution looking like the greedy solution. Then, this method returns the solution $optimal = [3, 9, 15, 20]$ making sure that it still verifies all the properties of a solution and that is better than the original optimal solution.

```
ghost method postpone(x: Input, v: Solution, i: nat, j: nat) returns (v_modified: Solution)
requires isValidInput(x)
```

```
requires isSolution(x,v) ∧ isSolOptimal(x,v)
requires 0 ≤ i < |v| ∧ v[i] < j
requires j in x.0
requires ∀ p | v[i] ≤ p < j ∧ p in x.0 :: i > 0 ∧  v[i−1] + x.1 ≥ p
ensures isSolution(x, v_modified)
ensures isBetterSol(x,v_modified,v)//they are equal
ensures |v_modified| = |v|
ensures v_modified[i] = j
ensures v_modified[..i] = v[..i]
{
    v_modified := v[i := j];
    changePos(x,v,i,j,i);
    assert ∀ m | m in x.0 :: covered(x,v_modified,m);

    var k := i+1;
    while (k < |v_modified| ∧ v_modified[k] < j)
      invariant i < k ≤ |v_modified| = |v|
      invariant sorted(v_modified[..k])
      invariant ∀ m | m in x.0 :: covered(x,v_modified,m)
      invariant v_modified[..i] = v[..i]
      invariant ∀ m | i ≤ m < k :: v_modified[m] = j
      invariant v_modified[k..] = v[k..]
      decreases |v_modified| − k
      {
        changePos(x,v_modified,i,j,k);
        v_modified := v_modified[k := j];
        k := k+1;
      }

    assert sorted(v_modified);
    assert ∀ i | i in x.0 :: covered(x,v_modified,i);

    //assert isSolution(x,v_modified);
    //assert isBetterSol(x,v_modified,v);
}
```

The method *changePos* receives as input parameters two values, i and j, which represent positions where we have patches, and a variable k, which verifies that $sol[k] = i$. The aim is to ensure that the solution $sol[k := j]$ covers all the holes, but we know this is guaranteed because all holes between $sol[i]$ and j are covered by $sol[i - 1]$. The proof was divided in three different cases:

1. the holes that where covered by the positions previous to $sol[i]$ are still covered by the new solution because those positions have not been changed.
2. all holes that are in position j or in any following position, which are covered by some element of the solution $sol[k..]$.
3. the holes in between, which are covered by the position $sol[i - 1]$.

These cases are represented in the following Dafny code:

```
lemma changePos(x: Input, sol: Solution, i: nat, j: nat, k: nat)
requires isValidInput(x)
requires 0 ≤ i ≤ k < |sol| ∧ sol[k] < j
requires sorted(sol[..k])
requires sorted(sol[k..])
requires ∀ m | sol[i] ≤ m < j ∧ m in x.0 ::
          i > 0  ∧ sol[i−1] ≤ m ∧ sol[i−1] + x.1 ≥ m
```

```
requires ∀ m | m < sol[i] ∧ m in x.0 ::
          (exists z :: 0 ≤ z < i ∧ sol[z] ≤ m ∧ sol[z] + x.1 ≥ m)
requires ∀ m | m ≥ j ∧ m in x.0 ::
          (exists z :: k ≤ z < |sol| ∧ sol[z] ≤ m ∧ sol[z] + x.1 ≥ m)
requires ∀ m | m in x.0 :: covered(x,sol,m)
ensures ∀ m | m in x.0 :: covered(x,sol[k := j],m)
{
    ∀ m | m in x.0
      ensures covered(x,sol[k := j],m)
      {
        if (m < sol[i]) {
          var z :| 0 ≤ z < i ∧ sol[z] ≤ m ∧ sol[z] + x.1 ≥ m;
          assert  sol[z] = sol[k:=j][z];
          assert sol[z] in sol[k:=j];
          //assert covered(x,sol[k := j],m);
        }
        else if (m ≥ j) {
          var z :| k ≤ z < |sol| ∧ sol[z] ≤ m ∧ sol[z] + x.1 ≥ m;
          if (z = k){
            assert sol[k:=j][k] in sol[k:=j];
            assert covered(x,sol[k := j],m);
          }
          else{
            assert sol[z] = sol[k:=j][z];
            assert sol[z] in sol[k:=j];
            assert covered(x,sol[k := j],m);
          }
        }
        else {
          assert sol[i-1] = sol[k:=j][i-1];
          assert sol[i-1] in sol[k:=j];
          assert covered(x,sol[k := j],m);
        }
      }
}
```

Chapter 5

Conclusions

This final chapter is a reflection on the work carried out in this master thesis, where conclusions are drawn about the initially proposed goals, the difficulties encountered and the future work that can be carried out.

Regarding the initially proposed goals, I think that most of them have been fulfilled. Not only have we managed to verify some greedy algorithms in Dafny, but we have also found a generic methodology that can be applied to verify other greedy algorithms. But to what kind of greedy problems can this methodology be applied? This is a question we have answered. There are different ways of telling when one solution is better than another one. In our methodology we use the *solutionValue* function, which returns a natural number and we know that a solution is better than another one when this natural number is smaller in the solution that is best. As we must be able to define this *solutionValue* function of the *GreedyAlgorithmValue* module, our methodology can be applied to problems whose optimality is based on a natural number. We focused on a total of four problems that have been verified applying this methodology. In order to do so, I have needed to learn about Dafny and how it works. I have had some surprises with this language, because sometimes its proof capacity is surprising, and other times, the task of finding the property that Dafny is not able to prove by itself is quite complicated or requires a lot of experience. On the other hand, I have also had to learn about greedy algorithms, how they work and their characteristics.

Before I continue, I must mention the experience of doing this work. The opportunity to research, develop and work for months on a subject that I have gradually found more and more interesting is an experience that I qualify as gratifying. In the first place, this work has allowed me to apply my mathematical knowledge acquired during my degree in the many proofs that have been carried out. But, on the other hand, I have been able to apply the knowledge I have acquired during the Master's degree, as I have been able to work with a language such as Dafny, where we find executable code and specifications. It is the union between these two areas that I liked the most from the beginning.

In this work, I have mainly learned about formal verification and Dafny. Before this project, what I had mostly seen was writing code without specification (or with a very small degree of specification). However, with this work, I have learned all that formal verification implies, such as specification. Preconditions, postconditions, invariants... all those properties that at some point are satisfied and must be specified and proved in case it is needed. However, it is well known that formal verification is not a simple task. I also learned to work with Dafny, which I had not worked with previously. It is a tool that allows you to do many things, but it also has its difficulties. Trying to prove some properties or lemmas could sometimes become an arduous task.

Finally, this work can still be developed. These are the possible ways to do it:

1. An idea is to try to apply our generic methodology to more complex algorithms, as for example some graph algorithms.
2. We have applied our methodology to problems where the existence of an optimal solution is guaranteed by the fact that the objective function is natural. It can be studied how to prove the existence of optimal solution in problems in which the function is not natural but a real number. In fact that is not necessarily a problem as far as the set of possible solutions is finite, as it happens for example in the minimum spanning tree problem. In that kind of problems it would be enough to traverse all the possible solutions in order to ensure the existence of an optimal one. But in other problems, like the *Knapsack* problem it is not clear to us yet, because the reasoning required is more complex and specific to this problem, if there is some general way of proving the existence and under which circumstances.
3. In this paper we focused on proving the optimality using the exchange proof, however we show in Chapter 3 that there exist other ways of proving greedy correctness as applying induction on the number of steps or basing the proof on special properties. So, another option is to verify greedy algorithms by using other techniques rather than exchange proof.

Bibliography

[1] François Bobot et al. *Why3*. Inria Saclay-Île-de-France / LRI Univ Paris-Sud 11 / CNRS., 2016. URL: http://why3.lri.fr/.

[2] *Basics of Greedy Algorithms - Tutorial*. URL: https://www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/tutorial/.

[3] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2022. Chap. 24.3.

[4] *Dafny Reference Manual*. URL: https://dafny.org/dafny/DafnyRef/DafnyRef.

[5] *Dafny: Modules*. URL: http://www.cse.unsw.edu.au/~se2011/DafnyDocumentation/.

[6] *Example Queue in Dafny*. URL: https://homepage.cs.uiowa.edu/~tinelli/classes/181/Fall17/Tools/Dafny/Examples/abstractQueue2.dfy.

[7] *Getting Started with Dafny: A Guide*. URL: https://info.usherbrooke.ca/mfrappier/IFT734/ref/dafny/documents/.

[8] *Greedy Algorithms*. URL: https://www.programiz.com/dsa/greedy-algorithm.

[9] *HOL Interactive Theorem Prover*. URL: https://hol-theorem-prover.org/.

[10] *K. Rustan M. Leino, Developing Verified Programs with Dafny*. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/krml233.pdf.

[11] *K. Rustan M. Leino, Types in Dafny*. 2015. URL: http://leino.science/papers/krml243.html.

[12] K. Rustan M. Leino. *Specification and Verification of Object-Oriented Software*. KIT Scientific Publishing, 2008.

[13] Paqui Lucio. *A Tutorial on Using Dafny to Construct Verified Software*. The University of the Basque Country (UPV/EHU), 2016. URL: https://doi.org/10.4204/EPTCS.237.1.

[14] Leonardo de Moura and Nikolaj Bjørner. *Z3: An effcient SMT solver*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[15] José Alberto Verdejo López Narciso Martí Oliet Yolanda Ortega Mallén. *Estructuras de datos y métodos algorítmicos: 213 ejercicios resueltos*. Ibergarceta Publicaciones S.L., 2013.

[16] *University of Cambridge and Technische Universität München. Isabelle.* URL: `https://isabelle.in.tum.de/`.