Dafny coursework exercises

John Wickerson

Autumn term 2021

There are six tasks and they are ordered in roughly increasing order of difficulty. Tasks labelled (⋆) are expected to be straightforward. Tasks labelled (⋆⋆⋆) are highly challenging; it is not expected that many students will complete these.

Marking principles. If you have completed a task, you will get full marks for it and it is not necessary to show your working. If you have not managed to complete a task, partial credit may be given if you can demonstrate your thought process. For instance, you might not be able to come up with *all* the invariants that are necessary to complete the verification, but perhaps you can confirm *some* invariants and express (in comments) some of the other invariants that you think are needed but haven't managed to verify.

Submission process. You are expected to produce a single Dafny source file called `YourName.dfy`. This file should contain your solutions to all of the tasks below that you have attempted. You are welcome to show your working on incomplete tasks by decorating your file with `/*comments*/` or `//comments`.

Plagiarism policy. You **are** allowed to consult the coursework tasks from previous years – the questions and model solutions for these are available. You **are** allowed to consult internet sources like Dafny tutorials. You **are** allowed to work together with the other student in your pair. Please **don't** submit these programs as questions on Stack Overflow! And please **don't** share your answers to these tasks outside of your own pair. If you would like to share your answers to these tasks publicly, e.g. on a public GitHub repo, you are welcome to do so, but please check with me first, because some students may still be working on the coursework with an extended deadline.

Task 1 (⋆) Write a Dafny method that meets the following specification.

```
method create_multiples_of_two(A:array<int>)
ensures forall n ::
  0 <= n < A.Length ==> A[n] == 2*n
modifies A
```

Instrument your code with enough loop invariants (and other assertions as you see fit) so that Dafny can prove that the postcondition is always met.

Task 2 (⋆⋆) Let us define the `sorted` predicate as follows:

```
predicate sorted(A:array<int>)
  reads A
{
  forall m,n ::
    0 <= m < n < A.Length ==> A[m] <= A[n]
}
```

Here is an implementation of 'exchange' sort, which is a variant of bubble sort. In exchange sort, each element is compared not just with the element that immediately follows it, but with *all* the elements that follow it.

```
method exchange_sort (A:array<int>)
ensures sorted(A)
modifies A
{
  var i := 0;
  while i < A.Length - 1 {
    var j := i + 1;
    while j < A.Length {
      if A[i] > A[j] {
        A[i], A[j] := A[j], A[i];
      }
      j := j + 1;
    }
    i := i + 1;
  }
}
```

Instrument this code with enough loop invariants (and other assertions as you see fit) so that Dafny can prove that the postcondition is always met.

Task 3 (⋆⋆) Here is a sorting method that was recently discovered by Stanley Fung from the University of Leicester.

```
method fung_sort (A:array<int>)
ensures sorted(A)
modifies A
{
  var i := 0;
  while i < A.Length {
    var j := 0;
    while j < A.Length {
      if A[i] < A[j] {
        A[i], A[j] := A[j], A[i];
      }
      j := j+1;
    }
    i := i+1;
  }
}
```

Instrument this code with enough loop invariants (and other assertions as you see fit) so that Dafny can prove that the postcondition is always met.

Task 4 (⋆⋆⋆) Here is an implementation of ‘odd/even’ sort, which is another variant of bubble sort. Odd/even sort splits each pass into two stages; in the first stage, odd-indexed elements are compared (and possibly swapped) with their successor, and in the second stage, the same is done for the even-indexed elements. In this implementation, repeated passes are made until a fixed point is reached.

```
method odd_even_sort(A:array<int>)
  ensures sorted(A)
  modifies A
  decreases *
{
  var is_sorted := false;
  while !is_sorted {
    is_sorted := true;
    var i := 0;
    while 2*i+2 < A.Length {
      if A[2*i+2] < A[2*i+1] {
        A[2*i+1], A[2*i+2] := A[2*i+2], A[2*i+1];
        is_sorted := false;
      }
      i := i+1;
    }
    i := 0;
    while 2*i+1 < A.Length {
      if A[2*i+1] < A[2*i] {
        A[2*i], A[2*i+1] := A[2*i+1], A[2*i];
        is_sorted := false;
      }
      i := i+1;
    }
  }
}
```

Instrument this code with enough loop invariants (and other assertions as you see fit) so that Dafny can prove that the postcondition is always met. Note that you are not required to prove that this method terminates, as indicated by the ‘`decreases *`’ clause. (As it happens, the method *does* terminate, but I think it is too difficult to prove this in Dafny.)

Remark. Usually, Dafny proves *total correctness*. A method is said to be ‘totally correct’ if it always terminates and it always gives the correct result when it does. In the task above, we are only proving what is called *partial correctness*. Partial correctness means that *if* the method terminates, it gives the correct result. (In particular, this implies that a method that always enters an infinite loop can always be considered ‘partially correct’, regardless of its postcondition!) The following equation provides a concise way to think about the relationship between the two versions of correctness:

$$\text{total correctness} \quad = \quad \text{partial correctness} \quad + \quad \text{termination.}$$

Task 5 (⋆⋆⋆) Here is yet another variant of bubble sort. Where ordinary bubble sort considers *pairs* of adjacent elements, this variant considers *triples* of adjacent elements.

```
method bubble_sort3(A:array<int>)
  ensures sorted(A)
  modifies A
  decreases *
{
  var stable := false;
  while !stable {
    stable := true;
    var j := 0;
    while 2*j+2 <= A.Length {
      if 2*j+2 == A.Length {
        if A[2*j+1] < A[2*j] {
          A[2*j+1], A[2*j] := A[2*j], A[2*j+1];
          stable := false;
        }
      } else {
        if A[2*j+1] < A[2*j] {
          A[2*j+1], A[2*j] := A[2*j], A[2*j+1];
          stable := false;
        }
        if A[2*j+2] < A[2*j+1] {
          A[2*j+2], A[2*j+1] := A[2*j+1], A[2*j+2];
          stable := false;
        }
        if A[2*j+1] < A[2*j] {
          A[2*j+1], A[2*j] := A[2*j], A[2*j+1];
          stable := false;
        }
      }
      j := j + 1;
    }
  }
}
```

Instrument this code with enough loop invariants (and other assertions as you see fit) so that Dafny can prove that the postcondition is always met. Note that you are not required to prove that this method terminates, as indicated by the '**decreases** *' clause. (As it happens, the method *does* terminate, but I think it is too difficult to prove this in Dafny.)

Task 6 (⋆) Consider a Dafny method with the following signature.

```
method sort3(a:int, b:int, c:int)
  returns (min:int, mid:int, max:int)
```

The method should take three integers and return those same three integers in ascending order.

1. Write a specification for this method that captures this behaviour.
2. Provide an implementation for this method. To make things interesting, your implementation is only permitted to use addition, subtraction, and the `get_min` function defined below. It is not allowed to use any if-statements or comparison operators.

```
function method get_min(a:int, b:int) : int
{
  if a <= b then a else b
}
```

3. Prove that your implementation meets your specification.