Dafny coursework exercises

John Wickerson

Autumn term 2020

Tasks are ordered in roughly increasing order of difficulty. Tasks labelled (⋆) are expected to be straightforward. Tasks labelled (⋆⋆) should be manageable but may require quite a bit of thinking, and it may be necessary to consult additional sources of information, such as an online Dafny tutorial or Stack Overflow. Tasks labelled (⋆⋆⋆) are challenging. It is not expected that many students will complete these, but partial credit will be given to partial answers.

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 not expected to complete all tasks. You *are* expected, however, to provide detailed annotations throughout your file (in the form of `/*comments*/` or `//comments`) that demonstrate the extent to which you have understood the software verification process. For instance, you might not be able to come up with *all* the invariants that are necessary to complete the verification, but partial credit will be given for writing down *some* invariants, and expressing (in comments) some of the other invariants that you think are needed but haven't managed to verify.

Plagiarism policy. You are allowed to consult the coursework tasks from last year – 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 predicate that determines whether every element of an array of integers is zero. Here is a template:

```
predicate zeroes(A:array<int>)
  reads A
{
  // ...
}
```

Here is a function that zeroes every element of an array.

```
method resetArray(A:array<int>)
  ensures zeroes(A)
  modifies A
{
  var i := 0;
  while i < A.Length {
    A[i] := 0;
    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 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 selection sort that works in a 'backward' manner. Where the traditional algorithm starts by swapping the minimal element into the first position of the array, this implementation starts by swapping the maximal element into the last position of the array.

```
method backwards_selection_sort(A:array<int>)
  ensures sorted(A)
  modifies A
{
  var i := A.Length;
  while 0 < i {
    var max := 0;
    var j := 1;
    while j < i {
      if A[max] < A[j] {
        max := j;
      }
      j := j+1;
    }
    A[max], A[i-1] := A[i-1], A[max];
    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 recursive implementation of selection sort.

```
method recursive_selection_sort_inner(A:array<int>, i:int)
  modifies A
{
  if i == A.Length {
    return;
  }
  var k := i;
  var j := i + 1;
  while j < A.Length {
    if A[k] > A[j] {
      k := j;
    }
    j := j+1;
  }
  A[k], A[i] := A[i], A[k];
  recursive_selection_sort_inner(A, i + 1);
}

method recursive_selection_sort(A:array<int>)
  ensures sorted(A)
  modifies A
{
  recursive_selection_sort_inner(A, 0);
}
```

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. You will need to annotate the `recursive_selection_sort_inner` function with a **`decreases`** clause, just like you do with loops, so that Dafny knows that the recursion will terminate. You will also need to add **`requires`** and **`ensures`** clauses to the `recursive_selection_sort_inner` function.

Task 4 (★★★) Here is an implementation of bubble sort that uses a Boolean flag (`sorted`) to enable early termination once the array is sorted.

```
method bubble_sort_with_early_stop(A:array<int>)
  ensures sorted(A)
  modifies A
{
  var stable := false;
  var i := 0;
  while !stable {
    var j := 0;
    stable := true;
    while j + 1 < A.Length - i {
      if A[j+1] < A[j] {
        A[j], A[j+1] := A[j+1], A[j];
        stable := false;
      }
      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 5 (★★★) Here is an implementation of cocktail-shaker sort. It is similar to bubble sort, but where bubble sort makes a series of ascending passes through the array, cocktail-shaker sort alternates between ascending and descending passes (like shaking a cocktail). This is intended to overcome the shortcoming of bubble sort whereby small values take a long time to descend to the start of the array.

```
method cocktail_shaker_sort(A:array<int>)
  ensures sorted(A)
  modifies A
{
  var i := 0;
  while i < A.Length / 2 {
    var j := i;
    while j < A.Length - i - 1 {
      if A[j+1] < A[j] {
        A[j], A[j+1] := A[j+1], A[j];
      }
      j := j+1;
    }
    while i < j {
      if A[j-1] > A[j] {
        A[j-1], A[j] := A[j], A[j-1];
      }
      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.