SENG 2011

Assignment 2

Proofs and Program Verification

Due Sunday 9pm, 20th November, 2022

Welcome to the second assignment. Some notes:

- The total number of marks of the exercises will be scaled to 25 marks for the course assessment.
- Please make sure your code verifies with the *CSE dafny* verifier.
- No compilation or execution is required in this assignment. No output is generated.
- Conform to the given method/function/predicate signatures exactly. The auto-marker is case-sensitive.
- You may include extra Dafny predicates and functions to improve structure and readability.
- Do not use the assume statement, function methods or predicate methods in this assignment.
- Warnings are treated as errors in the assessment.
- Specifications will be assessed for readability, conciseness, structure, range of testing and performance.
- Each of your submitted solutions should verify in between 10 and 20 seconds using *CSE dafny*. This includes all the testers in each exercise. There is a verification time limit of 60 seconds for each exercise. If your solution exceeds this, and there is justification, contact the course convenor.

---

ex1.dfy 1(i) + 7(ii) + 2(iii) marks

The following Dafny method `PropertyTester` asserts an arithmetic property. Unfortunately Dafny does not 'know' this property so the assertion fails.

```
// description of the property here
method PropertyTester(n: nat, k: nat)
requires n%2==1
{
   assert n*(n+2) != k*k;
}
```

i Replace the comment at the top of the tester with a one-line sentence that states the property. Use simple, conceptual English, with no technical jargon, no variable names and no Dafny-speak.

ii Write a level-3 induction lemma that proves the assertion. Use the signature:

```
lemma LemNPS(n: nat, k: nat)
```

iii Modify the tester to demonstrate that Dafny has 'learnt' the property.

Submit the file `ex1.dfy`, which should contain:

- lemma `LemNPS`
- your modified tester method `PropertyTester` that passes the assertion

It's important that your lemma has the same signature shown above.

ex2.dfy 10(bulls) + 20(cows) marks

Bulls and cows is a game of deduction that involves the system selecting a sequence of n digits, followed by the user trying to guess what the sequence is. Each digit in the sequence is unique, where the digits are 0,...,9 of course. Examples of sequences are [8], [9,1,3,0,2] and [0,1,2,3,4,5,6,7,8,9]. The sequence [1,4,3,1] is invalid as a digit repeats. Initially, the user knows only the length of the system's sequence.

The user can make as many guesses as he/she likes, but each guess must be a sequence that conforms to the same sequence requirements (same length, only digits and all unique). Each time, the user is told how many *cows* and *bulls* he/she has scored. A *bull* is defined as a correct digit in the correct position in the sequence, and a *cow* as a correct digit but in an incorrect position.

For example, if the system (secretly) selects the sequence [4,2,9,3,1], and the user guesses [1,2,3,4,5], then the user would be told that he/she has scored 1 bull and 3 cows, but not what they are of course. With this knowledge, the user makes another guess trying to improve his/her score. (*The bull is digit 2, and the cows are 1, 3 and 4 by the way.*) Eventually the user should be able to deduce what the sequence is, and in his/her last guess will score n bulls (and 0 cows of course). The aim is to make as few guesses as possible.

Bulls and cows is similar to *mastermind*, but predates it by many decades. You can play the game on `www.mathsisfun.com/games/bulls-and-cows.html` for a sequence of 4 digits.

The exercise: You do not have to code the game in Dafny. The exercise is to verify the calculation of the number of cows and bulls in the user's guess. Do this in two steps:

i Write two functions:

```
function bullspec(s:seq<nat>, u:seq<nat>): nat
function cowspec(s:seq<nat>, u:seq<nat>): nat
```

which specify the number of bulls and cows in the user sequence u for selection s. The sequences u and s must satisfy the requirements of the game of course.

ii Write a method with signature:

```
method BullsCows (s:seq<nat>, u:seq<nat>)  returns (b:nat, c:nat)
```

which computes the actual scores using Dafny code and verifies the code using the above functions.

Testing: Write a tester that provides a range of black-box tests for the functions and method. A sample testcase for a game of 5 digits is:

```
var sys:seq<nat> := [4,2,9,3,1];         // system selection
var usr:seq<nat> := [1,2,3,4,5];         // user guess
assert bullspec(sys, usr) == 1;          // verify the bulls
assert cowspec(sys, usr)  == 3;          // verify the cows

var b:nat, c:nat := BullsCows(sys, usr); // calculate the bulls and cows
assert b == 1 && c == 3;                 // verify the calculation of bulls and cows
```

I have included calls to the functions in the above tester to show how they work as well. You may write separate testers if you like. You may call your tester(s) anything you like, but not Main.

Submit the file `ex2.dfy`, which should contain:

- function `bullspec`
- function `cowspec`
- method `BullsCows`
- your tester(s)

Be careful that you conform to the given signatures in i and ii, and remember that you are not allowed to use predicate methods and function methods.

ex3.dfy 8(i) + 8(ii) + 24(iii) marks

In bio-chemistry, sequencing DNA means determining the order of the four chemical building blocks, called 'bases', that make up the DNA molecule. These bases are adenine (A), cytosine (C), guanine (G) and thymine (T). For the purpose of this work, assume that there may be any number of bases in a sequence (but greater than 0). (There are actually billions.)

i Write a verified method that exchanges the bases at positions x and y in a sequence. A requirement is there must be bases at these positions. The first base in a sequence is at position 0.

For example, if the sequence of bases is [A, C, A, T] then exchanging bases at positions 2 and 3 results in [A, C, T, A].

The signature of the method should be:

```
method Exchanger(s: seq<Bases>, x:nat, y:nat) returns (t: seq<Bases>)
```

where `Bases` is a datatype that represents the four DNA bases. The sequence t that is returned has the bases exchanged. The input sequence s is not altered.

Testing: Write a tester method that checks that `Exchanger` is working for a comprehensive range of testcases. A sample testcase is:

```
var a:seq<Bases> := [A, C, A, T];       // the testcase sequence
var b:seq<Bases> := Exchanger(a, 2, 3); // exchange bases at position 2 and 3
assert b == [A, C, T, A];               // assert the new sequence
```

You may call the tester method anything you like, but not Main.

ii Write a predicate with the following signature:

```
predicate bordered(s:seq<Bases>)
```

which returns true if the input sequence is *base-ordered*, and false otherwise. Base-order is alphabetical: that is `A`, `C`, `G`, `T`.

For example, the sequence s = [A, G, G, G, T] is in base-order so *bordered(s)* is true. The empty sequence is also base-ordered. The sequence t = [G, A, T] is not in base-order so *bordered(t)* is false.

iii Write a verified method with signature:

```
method Sorter(bases: seq<Bases>) returns (sobases:seq<Bases>)
```

which sorts the sequence `bases`, creating and returning a new sorted sequence `sobases`. There must be at least one base in `bases`. The ordering of bases should be as in part ii of course.

Some conditions:

- You should use predicate `bordered` to verify whether a sequence is ordered or not.
- The sorter must be linear in performance (quadratic sorts such as insertion sort are not suitable as they are too slow).
- You should not use an array anywhere in this exercise.

Testing: Write a tester method that checks that your sorter is working. Provide a comprehensive range of testcases. A sample testcase is:

```
var a:seq<Bases> := [G,A,T];       // the testcase
assert a == [G,A,T];               // assert the values
var b:seq<Bases> := Sorter(a);     // sort the bases in the sequence
assert bordered(b);                // test the new sequence is ordered
assert multiset(b) == multiset(a); // test the new sequence contains the same bases
```

For small testcases, such as the one above, you could have tested the new sequence is ordered simply using `assert b == [A,G,T]`. For technical reasons, however, this will not work in general, and instead you will need to use the code used above.

As usual, you may call the tester method anything you like (but not Main and not the same name as you used earlier in this exercise).

Submit the file `ex3.dfy`. This file should contain:

- method `Exchanger` and its tester
- predicate `bordered`
- method `Sorter` and its tester

In total, you should submit the files `ex1.dfy`, `ex2.dfy` and `ex3.dfy` on the course website. If you have made no attempt at a question, submit an empty file for that question. It is safest to submit all the files again if you change one and wish to re-submit. That keeps all the latest version together. Good luck.