CSC410

Assignment 5

Due on Monday October 2nd, 2017 at 11am

Assignment Format and Guidelines on Submission

This assignment is now complete. We will only update this PDF later to include precise submission information for Markus.

Problem 1 (20 points)

Below, you can see a function that sorts an array. This a simpler sorting routine called *insertion sort*. The goal is to verify partial correctness of this implementation. Naturally, we expect a sorting routine to return a *sorted* array. In case you are wondering, you do not have to verify that the resulting array and the original one contain the same set of elements.

```
method insertSort(a : array<int>)
{
  var i := 1;
  while (i < a.Length)
  {
    var j := i;
    var value := a[i];
    a[i] := a[i-1];
    while (j > 0 && a[j-1] > value)
    {
      a[j] := a[j-1];
      j := j - 1;
    }
    a[j] := value;
    i := i + 1;
  }
}
```

(a) What are the appropriate precondition and postcondition for this function? Naturally, we want sortedness as part of the postcondition!

(b) What are the (inductive) loop invariants? Note that there are two while loops in this example. You will submit a Dafny file containing the code with your pre/post-conditions and loop invariants that successfully runs through Dafny. Keep in mind that your assignment will be graded automatically. If it does not go through Dafny, you will get no marks.

Do not forget: no assignment will be accepted from a group of less than 3 students. Find your groupmates today! You can use pizza's feature for this or just talk to your classmates in class.

Problem 2 (50 points)

The following "elegant" sort algorithm gets its name from the *Three Stooges* slapstick routine in which each stooge hits the other two.

Hint: this is a bit tricky. Try to elaborate on "what" we know about the contents of each 1/3rd array after each recursive call, with respect to the content of the other two 1/3rd arrays.

```
method stoogeSort(a : array<int>, left : int, right : int)
{
```

```
  if (a[left] > a[right]) {
   // swap a[left] and a[right]
    var tmp := a[left];
    a[left] := a[right];
    a[right] := tmp;
  }
  if (left + 1 >= right)
     return;
  k := (right - left + 1) div 3;
  stoogeSort(a,left, right - k); // First two-thirds
  stoogeSort(a,left + k, right); // Last two-thirds
  stoogeSort(a,left, right - k); // First two-thirds again
}
```

- What are the appropriate precondition and postcondition for this function? Naturally, we want sortedness as part of the postcondition!

  You will submit a text file containing the code with your pre/post-conditions that successfully runs through Dafny. Naturally, this file will include functions, lemmas, and all the extra goodies you require to prove that the function correctly sorts. Like the previous example, the aspect of the proof that the sorted array is a permutation of the original array elements may be skipped as the post condition.

  Note: this is your hardest problem. The reason is that unlike Problem 3, we have not already provided you with the "intuitive" reason why the code above correctly sorts, and only left the formalization for you. You need to first conceptually find that reason, and then formalize it.

Problem 3 (50 points)

This solution is based on a known puzzle, "catch a spy". The purpose is for us to practice using a theorem prover, like Dafny, to do a formal proof that is not related to program. The puzzle and its solution are effectively provided to you (and you can consult resources online for this). We are not asking you to find a solution for the puzzle. We are asking you to use Dafny to prove formally that the provided solution for the puzzle is indeed correct.

The spy exists on a one-dimensional line. His location is described by the expression $a+t*b$, where $a, b \in \mathbb{N}$ are unknown constants and $t \in \mathbb{N}$ is the number of seconds since he pirated your work. Unfortunately, he has also stolen U of T's cloak of invisibility, so our only means to find him is to physically query a single location every second. That is, at every second, we may ask "is the spy at location n" and get back a "yes" or "no" answer.

We can catch the spy as follows. The set $\mathbb{N} \times \mathbb{N}$ of pairs of natural numbers is countable, so there exists a function unpair : $\mathbb{N} \to \mathbb{N} \times \mathbb{N}$ that covers all pairs of natural numbers. Our strategy is to check the location $x + t \times y$ at time t, where $(x, y) = \text{unpair}(i)$. Since there exists some t_0 such that $(a, b) = \text{unpair}(t_0)$, then at time t_0 we will check the location $a + t_0 * b$ and therefore catch the spy.

Complete the implementation of *unpair* according to our strategy above, and prove that the while loop terminates (i.e. that we eventually catch the spy).

```
function method unpair(i: nat): (nat, nat)
  { // TODO }

function method pick(i: nat): nat
{
  var (x, y) := unpair(i);
  x + i * y
}

method catchTheSpy(a: nat, b: nat)
{
```

```
var i := 0;
  while a + i * b != pick(i)
    { i := i + 1; }
}
```

You may use any valid implementation of unpair : $\mathbb{N} \to \mathbb{N} \times \mathbb{N}$, but the Cantor pairing function is suggested. The Cantor pairing function essentially lists all pairs who's sums are 0, then all pairs who's sums are 1, and so on.

```
unpair(0) = (0, 0)

unpair(1) = (0, 1)
unpair(2) = (1, 0)

unpair(3) = (0, 2)
unpair(4) = (1, 1)
unpair(5) = (2, 0)

...
```

On 2017/09/25: minor typo fixed above. The return type of unpair was mistakenly left as a single "`nat`" before. But, its use in `pick` already indicated that it is meant to return a pair of elements.