Master in Informatics and Computing Engineering (M.EIC)

# M.EIC037 | Formal Methods for Critical Systems 2021/22

## Exercises on Program Verification with arrays with Dafny (Resolution)

## 1. Binary Search [Selected]

Assume the following implementation of the binary search algorithm in Dafny:

```
// Finds a value 'x' in a sorted array 'a', and returns its index,
// or -1 if not found.
method binarySearch(a: array<int>, x: int) returns (index: int) {
    var low, high := 0, a.Length;
    while low < high {
        var mid := low + (high - low) / 2;
        if {
            case a[mid]  < x => low := mid + 1;
            case a[mid]  > x => high := mid;
            case a[mid] == x => return mid;
        }
    }
    return -1;
}
```

a) Identify adequate pre and post-conditions for this method, and encode them as "requires" and "ensures" clauses in Dafny. You can use the predicate below if needed.

```
// Checks if array 'a' is sorted.
predicate isSorted(a: array<int>)
  reads a
{
    forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
}
```

```
requires isSorted(a)
ensures (index < a.Length && a[index] == x) ||
           (index == -1 && forall i :: 0 <= i < a.Length ==> a[i] != x)
```

Or:

```
ensures (index < a.Length && a[index] == x) ||
           (index == -1 && x !in a[..])
```

b) Identify an adequate loop variant and loop invariant, and encode them as "decreases" and "invariant" clauses in Dafny.

```
decreases high - low
invariant 0 <= low <= high <= a.Length
invariant forall i :: 0 <= i < low || high <= i < a.Length ==> a[i] != x
```

Or:

```
    invariant 0 <= low <= high <= a.Length
```

```
    invariant x !in a[..low] && x !in a[high..]
```

## 2. Insertion Sort [Selected]

Assume the following implementation of the insertion sort algorithm in Dafny:

```
// Sorts array 'a' using the insertion sort algorithm.
method insertionSort(a: array<int>) {
    var i := 0;
    while i < a.Length {
        var j := i;
        while j > 0 && a[j-1] > a[j] {
            a[j-1], a[j] := a[j], a[j-1];
            j := j - 1;
        }
        i := i + 1;
    }
}
```

a) Identify adequate pre and post-conditions for this method, and encode them as "requires" and "ensures" clauses in Dafny. (Suggestion: See SelectionSort.dfy).
Solution:

```
 modifies a
 ensures isSorted(a, 0, a.Length)
 ensures multiset(a[..]) ==  multiset(old(a[..]))
```

b) Identify adequate variants and invariants for the two loops, and encode them as "decreases" and "invariant" clauses in Dafny.
Solution, outer Loop:

```
     invariant 0 <= i <= a.Length
     invariant isSorted(a, 0, i)
     invariant multiset(a[..]) ==  multiset(old(a[..]))
```

Solution, inner Loop:

```
      invariant 0 <= j <= i
      invariant forall l, r :: 0 <= l < r <= i && r != j ==> a[l] <= a[r]
      invariant multiset(a[..]) ==  multiset(old(a[..]))
```

## 3. Sorting algorithms

Implement and verify in Dafny one of the following sorting algorithms: bubble sort (easier), merge sort, quick sort.

```
/*
* Formal verification of the merge sort algorithm with Dafny.
* FEUP, MIEIC, MFES, 2020/21.
*/
```

```
type T = int

// Checks if array 'a' is sorted between positions 'from' (inclusive) and 'to' (exclusive).
predicate sorted(a: array<int>, from : int, to: int)
  reads a
{
  forall i, j :: 0 <= from <= i < j < to <= a.Length ==> a[i] <= a[j]
}

// Checks if sequences 'a' and 'b' are identical, except for elements between indices
// 'lo' and 'hi' (inclusive), that may be permutated.
predicate permutation(a: seq<T>, b: seq<T>, lo: int, hi: int)
  requires 0 <= lo <= hi + 1 <= |a| == |b|
{
  a[..lo] == b[..lo] && a[hi+1..] == b[hi+1..]
  && multiset(a[..]) == multiset(b[..])
}

// Clones an array
method clone(a: array<T>) returns (res: array<T>)
  ensures fresh(res)
  ensures res[..] == a[..]
{
    var b := new T[a.Length];
    var i := 0;
    while i < a.Length
      decreases a.Length - i
      invariant 0 <= i <= a.Length
      invariant forall k :: 0 <= k < i ==> b[k] == a[k]
    {
        b[i] := a [i];
        i := i +1;
    }
    return b;
}

// Sorts array using the merge sort algorithm.
method sort(a: array<T>)
  modifies a
  ensures sorted(a, 0, a.Length)
  ensures multiset(a[..]) == multiset(old(a[..]))
{
    mergeSortRec(a, 0, a.Length - 1);
}


// Sorts array 'a' between indices 'p' and 'r' (inclusive) using the merge sort algorithm.
method mergeSortRec(a: array<T>, p: int, r: int)
  requires 0 <= p <= r + 1 <= a.Length
  modifies a
  decreases r - p
  ensures sorted(a, p, r+1)
  ensures permutation(a[..], old(a[..]), p, r)
{
    if p < r
    {
        var q := (p + r) / 2;
        mergeSortRec(a, p, q);
        mergeSortRec(a, q + 1, r);
        var b := clone(a);
        merge(b, p, q, r, a);
    }
}


// Merges non-empty sorted subarrays a[p..q] (q inclusive) and a[q+1..r] (r inclusive) into a
single sorted
// subarray b[p..r] (r inclusive).
method merge(a: array<T>, p: nat, q: nat, r: nat, b: array<T>)
  requires 0 <= p <= q < r < a.Length
  requires a != b // to make sure 'a' is not modified!
  requires a[..] == b[..]
  requires sorted(a, p, q+1) && sorted(a, q+1, r+1)
```

```
  modifies b
  ensures sorted(b, p, r+1)
  ensures permutation(b[..], old(b[..]), p, r)
{
    var i := p; // index in a[p..q] (q inclusive)
    var j := q + 1; // index in a[q+1..r] (r inclusive)
    var k := p; // index in b[p..r] (r inclusive)
    // Repeatedly take the smallest element of the left and right subarrays
    while k <= r
      decreases r + 1 - k
      invariant p <= i <= q + 1 <= j <= r + 1
      invariant p <= k <= r + 1
      invariant sorted(b, p, k)
      invariant k > p && i <= q ==> b[k-1] <= a[i]
      invariant k > p && j <= r ==> b[k-1] <= a[j]
      invariant k - p == (i - p) + (j - (q+1))
      invariant multiset(b[p..k]) == multiset(a[p..i]) + multiset(a[q+1..j])
      invariant forall k :: 0 <= k < p || r < k < b.Length ==> b[k] == old(b[k])
    {
        if i <= q && (j > r || a[i] <= a[j])
        {
            b[k] := a[i];
            i := i+1;
        }
        else
        {
            b[k] := a[j];
            j:= j+1;
        }
        k := k+1;
    }

     // needed to convince Dafny that the poscondition with multisets follows from the invariants
...
    assert a[p..r+1] == a[p..q+1] + a[q+1..r+1];
    assert a[..p] == b[..p];
    assert a[r+1..] == b[r+1..];
    assert a[..] == a[..p] + a[p..r+1] + a[r+1..];
    assert b[..] == b[..p] + b[p..r+1] + b[r+1..];
}
```

```
/*
* Formal verification of the quick sort algorithm with Dafny.
* The algorithm is based on https://en.wikipedia.org/wiki/Quicksort#Lomuto_partition_scheme
* Illustrates the usage of advanced proof techniques.
* FEUP, MIEIC, MFES, 2020/21.
*/

/** Auxiliary predicates and definitions **/

type T = int // could also be real or other comparable type

// Checks if array 'a' is sorted between positions 'lo' and 'hi' (inclusive)
predicate sorted(a: array<T>, lo: int, hi: int)
  requires 0 <= lo <= hi + 1 <= a.Length
  reads a
{
  forall i, j :: 0 <= lo <= i < j <= hi < a.Length ==> a[i] <= a[j]
}

// Checks if sequences 'a' and 'b' are identical, except for elements between indices
// 'lo' and 'hi' (inclusive), that may be permutated.
predicate permutation(a: seq<T>, b: seq<T>, lo: int, hi: int)
  requires 0 <= lo <= hi + 1 <= |a| == |b|
{
  a[..lo] == b[..lo] && a[hi+1..] == b[hi+1..]
  && multiset(a[lo..hi+1]) == multiset(b[lo..hi+1])
}

// Checks if elements of subarray between indices 'lo' and 'hi (inclusive),
// are less than a value x.
```

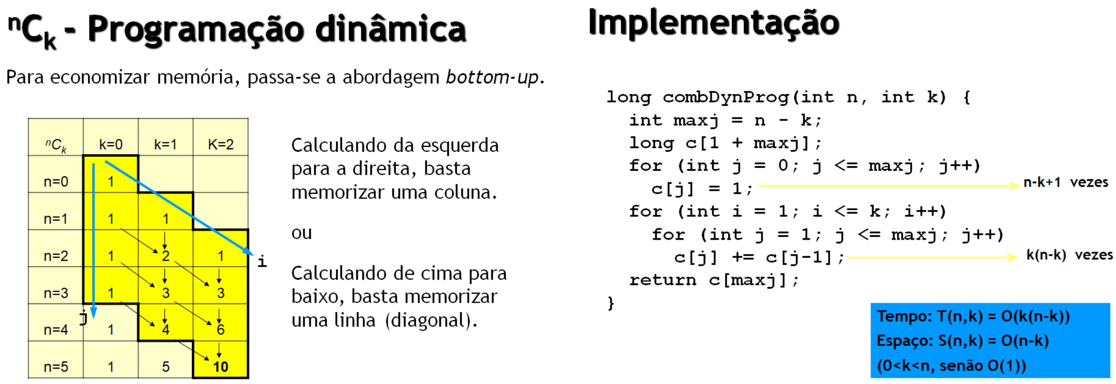
```
predicate subseqLt(a: seq<T>, lo: int, hi: int, x: T)
  requires 0 <= lo <= hi + 1 <= |a|
{
  forall k :: lo <= k <= hi ==> a[k] < x
}

// Checks if elements of subarray between indices 'lo' and 'hi (inclusive),
// are greater or equal than a value x.
predicate subseqGe(a: seq<T>, lo: int, hi: int, x: T)
  requires 0 <= lo <= hi + 1 <= |a|
{
  forall k :: lo <= k <= hi ==> a[k] >= x
}

/** Main algorithm  **/

// Sorts sorts elements at 'lo' through 'hi' (inclusive) of an array 'a'
// using the quicksort algorithm as described in
// https://en.wikipedia.org/wiki/Quicksort#Lomuto_partition_scheme
method quicksort(a: array<T>)
  modifies a
  ensures sorted(a, 0, a.Length-1)
  ensures permutation(a[..], old(a[..]), 0, a.Length-1)
{
  quicksortRec(a, 0, a.Length-1);
}

// Recursive method for quicksorting array 'a' between indices 'lo' and 'hi' (inclusive).
method quicksortRec(a: array<T>, lo: int, hi: int)
  requires 0 <= lo <= hi + 1 <= a.Length
  decreases hi - lo
  modifies a
  ensures sorted(a, lo, hi)
  ensures permutation(a[..], old(a[..]), lo, hi)
{
    if lo < hi {
      var p := partition(a, lo, hi);
      label L1: quicksortRec(a, lo, p  - 1);
        // Recall auxiliary lemmas to show that the post-conditions of partition are preserved...
        permutationInv(old(a[..]), old@L1(a[..]), a[..], lo, hi, lo, p - 1);
        subseqLtInv(old@L1(a[..]), a[..], lo, p-1, a[p]);
      label L2: quicksortRec(a, p + 1, hi);
        // Recall auxiliary lemmas to show that the post-conditions of partition are preserved...
        permutationInv(old(a[..]), old@L2(a[..]), a[..], lo, hi, p + 1, hi);
        subseqGeInv(old@L2(a[..]), a[..], p + 1, hi, a[p]);
    }
}

// Partitions a non-empty subarray 'a' between indices 'lo' and 'hi' (inclusive), with smaller
values
// to the left and higher or equal values to the right of the pivot, and returns the pivot
position.
method partition(a: array<T>, lo: int, hi: int) returns(pivot: int)
  requires 0 <= lo <= hi < a.Length
  modifies a
  ensures lo <= pivot <= hi
  ensures subseqLt(a[..], lo, pivot-1, a[pivot])
  ensures subseqGe(a[..], pivot+1, hi, a[pivot])
  ensures multiset(a[lo..hi+1]) == multiset(old(a[lo..hi+1]))
  ensures permutation(a[..], old(a[..]), lo, hi)
{
    pivot := hi; // pivot starts at end of array
    var i := lo;
    var j := lo;
    while  j < hi
      decreases hi - j
      invariant lo <= i <= j <= hi
      invariant subseqLt(a[..], lo, i-1, a[pivot])
      invariant subseqGe(a[..], i, j-1, a[pivot])
      invariant permutation(a[..], old(a[..]), lo, hi)
    {
      if a[j] < a[pivot] {
        a[i], a[j] := a[j], a[i];
```

```
        i := i + 1;
      }
      j := j+1;
    }
    a[hi], a[i] := a[i], a[hi];
    pivot := i; // pivot is swapped to the 'mid' of array
}

/** Auxiliary lemmas  **/

// States that permutation is invariant under (sub)permutation.
lemma permutationInv(a: seq<T>, b: seq<T>, c: seq<T>, lo1: int, hi1: int, lo2: int, hi2: int)
  requires 0 <= lo1 <= lo2 <= hi2 + 1 <= hi1 + 1 <= |a| == |b| == |c|
  requires permutation(a, b, lo1, hi1) && permutation(b, c, lo2, hi2)
  ensures permutation(a, c, lo1, hi1)
{
  // Intermmediate proof steps needed by Dafny (!) ...
  assert b[hi2 + 1 .. hi1 + 1] == c[hi2 + 1 .. hi1 + 1];
  assert b[lo1..lo2] + b[lo2..hi2 + 1] + b[hi2 + 1..hi1 + 1] == b[lo1..hi1 + 1];
  assert c[lo1..lo2] + c[lo2..hi2 + 1] + c[hi2 + 1..hi1 + 1] == c[lo1..hi1 + 1];
}

// States that subseqLt is invariant under permutation
lemma subseqLtInv(a: seq<T>, a': seq<T>, lo: int, hi: int, x: T)
  requires 0 <= lo <= hi + 1 <= |a| && subseqLt(a, lo, hi, x)
  requires |a| == |a'| && permutation(a, a', lo, hi)
  ensures subseqLt(a', lo, hi, x)
{
  // Intermmediate proof step needed by Dafny (!) ...
  assert forall k :: lo <= k <= hi ==> a'[k] in multiset(a'[lo..hi+1]);
}

// States that subseqGe is invariant under permutation
lemma subseqGeInv(a: seq<T>, a': seq<T>, lo: int, hi: int, x: T)
  requires 0 <= lo <= hi + 1 <= |a| && subseqGe(a, lo, hi, x)
  requires |a| == |a'| && permutation(a, a', lo, hi)
  ensures subseqGe(a', lo, hi, x)
{
  // Intermmediate proof step needed by Dafny (!) ...
  assert forall k :: lo <= k <= hi ==> a'[k] in multiset(a'[lo..hi+1]);
}
```

## 4. Combinations ($^nC_k$)

**a)** Encode in Dafny a function to define $^nC_k$ according to the Pascal rule $^nC_k = {}^{n-1}C_k + {}^{n-1}C_{k-1}$ $(0<k<n)$, with boundary cases $^nC_k=1$, if $k=0 \lor k=n$.

**b)** A method to calculate $^nC_k$ efficiently in terms of time and space using dynamic programming is reproduced below (from "Conceção e Análise de Algoritmos"). Encode it in Dafny and prove its correctness with respect to the definition in (a).




```
/*
* Formal specification and verification of a dynamic programming algorithm for
```

```
* calculating C(n, k).
* FEUP, MIEIC, MFES, 2020/21.
*/

// Initial recursive definition of C(n, k), based on the Pascal equality.
function method comb(n: nat, k: nat): nat
  requires 0 <= k <= n
{
  if k == 0 || k == n then 1 else comb(n-1, k) + comb(n-1, k-1)
}

// Calculates C(n,k) iteratively in time O(k*(n-k)) and space O(n-k),
// with dynamic programming.

method combDyn(n: nat, k: nat) returns (res: nat)
  requires 0 <= k <= n
  ensures res == comb(n, k);
{
  var maxj := n - k;
  var c := new nat[1 + maxj];
  forall i | 0 <= i <= maxj {
       c[i] := 1;
  }
  var i := 1;
  while i <= k
    decreases  k - i
    invariant 1 <= i <= k + 1
    invariant forall j :: 0 <= j <= maxj ==> c[j] == comb(j + i - 1, i - 1)
  {
    var j := 1;
    while j <= maxj
      decreases maxj - j
      invariant 1 <= j <= maxj + 1
      invariant forall j' :: 0 <= j' < j ==> c[j'] == comb(j' + i, i);
      invariant forall j' :: j <= j' <= maxj ==> c[j'] == comb(j' + i -  1, i - 1);
    {
      c[j] := c[j] + c[j-1];
      j := j+1;
    }
    i := i + 1;
  }
  return c[maxj];
}
```