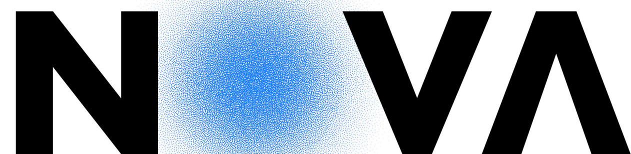
Construction and Verification of Software

2021 - 2022

MIEI - Integrated Master in Computer Science and Informatics
Consolidation block

Lecture 4 - Loop Invariants (cont.) and Sorting

Bernardo Toninho (btoninho@fct.unl.pt)
based on previous editions by **João Seco** and **Luís Caires**



NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Outline

- Loop Invariants (recap)
- Sorted Arrays
- Sorting algorithms
- Verification with bounded integers
- Abstract Data Types (intro)

Part I
Loop Invariants (recap + more)

Loop Invariants

- Loop invariant approximate state assertions before the loop, between iterations, and at the end of the loop.

```
method MaxArray(a:array<int>) returns (m:int)
    requires 0 < a.Length
    ensures forall k : int :: 0 <= k < a.Length ==> a[k] <= m
{
    m := a[0];
    var i := 1;
    while i < a.Length
        invariant 1 <= i <= a.Length
        invariant forall k : int :: 0 <= k < i ==> a[k] <= m
    {
        if m < a[i]
        { m := a[i]; }
        i := i + 1;
    }
}
```

Example: BinarySearch

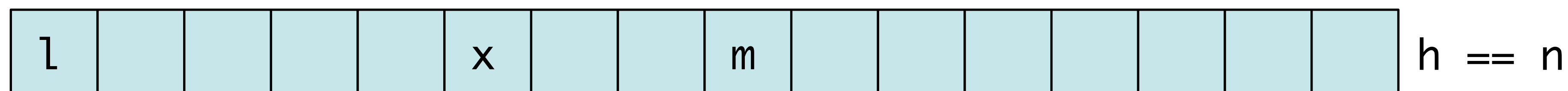
```
predicate sorted(a:array<char>, n:int)
    requires 0 <= n <= a.Length
    reads a
{   forall i, j:: (0 <= i < j < n) ==> a[i] <= a[j]  }
```

Example: BinarySearch

```
function sorted(a:array<char>, n:int):bool
    requires 0 <= n <= a.Length
    reads a
{   forall i, j:: (0 <= i < j < n) ==> a[i] <= a[j]  }

method BSearch(a:array<char>, n:int, value:char) returns (pos:int)
    requires 0 <= n <= a.Length && sorted(a, n)
    ensures …
    ensures …
{









}
```

Example: BinarySearch

```
function sorted(a:array<char>, n:int):bool
    requires 0 <= n <= a.Length
    reads a
{   forall i, j:: (0 <= i < j < n) ==> a[i] <= a[j]  }

method BSearch(a:array<char>, n:int, value:char) returns (pos:int)
    requires 0 <= n <= a.Length && sorted(a, n)
    ensures 0 <= pos ==> pos < n && a[pos] == value
    ensures pos < 0  ==> forall i :: (0<= i < n) ==> a[i] != value
{
```



```
}
```

Example: BinarySearch

```
predicate sorted(a:array<char>, n:int)
    requires 0 <= n <= a.Length
    reads a
{ forall i, j:: (0 <= i < j < n) ==> a[i] <= a[j] }

method BSearch(a:array<char>, n:int, value:char) returns (pos:int)
    requires 0 <= n <= a.Length && sorted(a, n)
    ensures 0 <= pos ==> pos < n && a[pos] == value
    ensures pos < 0  ==> forall i :: (0<= i < n) ==> a[i] != value
{
    var low, high := 0, n;
    while low < high
        decreases high - low
        invariant ????
        invariant ????
        invariant ????
    {
        var mid := low + (high-low) / 2;
        if a[mid] < value           { low := mid + 1; }
        else if value < a[mid]      { high := mid; }
        else /* value == a[mid] */ { return mid; }
    }
    return -1;
}
```

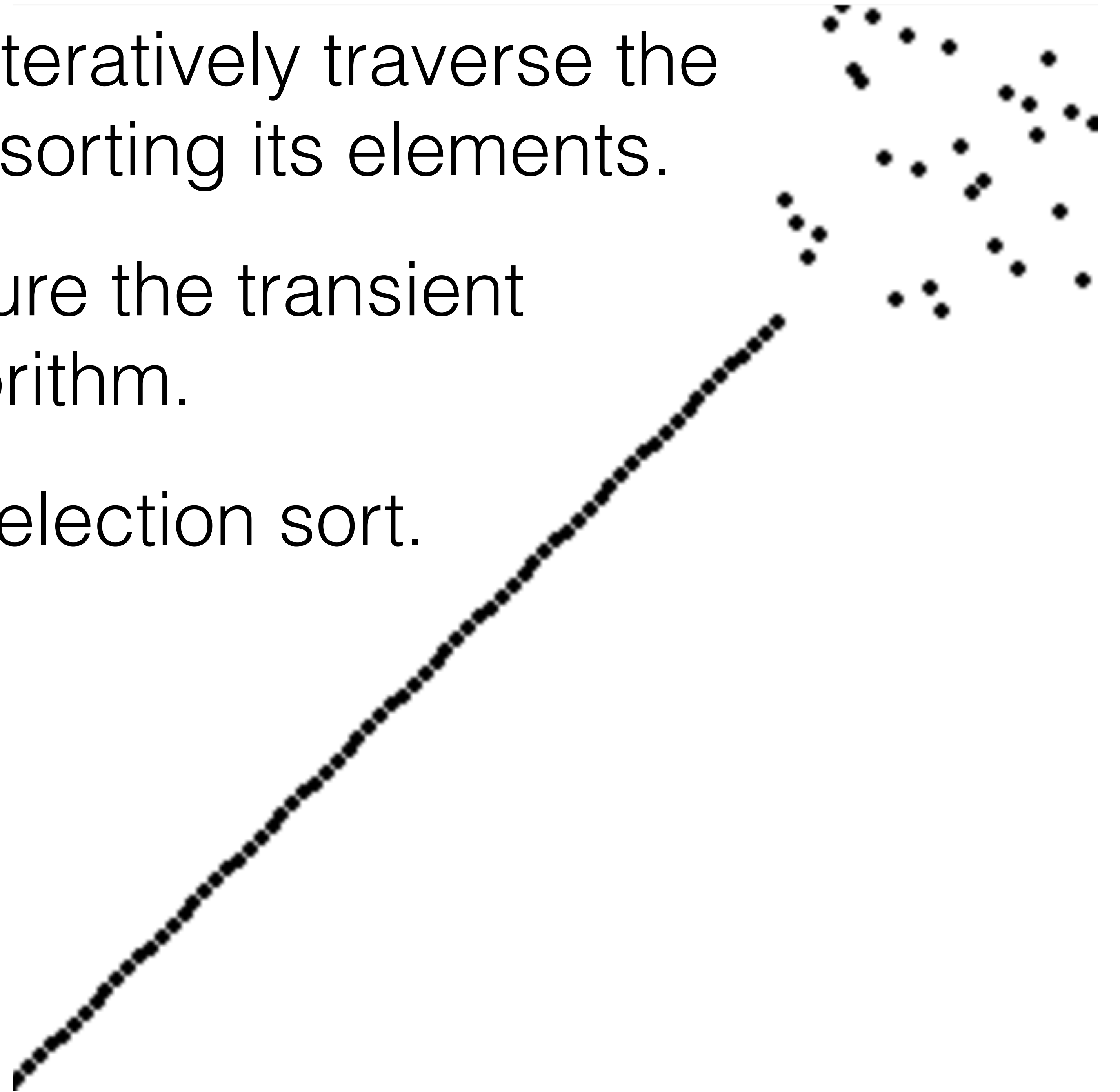
Example: Binary Search

```
predicate sorted(a:array<char>, n:int)
    requires 0 <= n <= a.Length
    reads a
{ forall i, j:: (0 <= i < j < n) ==> a[i] <= a[j] }

method BSearch(a:array<char>, n:int, value:char) returns (pos:int)
    requires 0 <= n <= a.Length && sorted(a, n)
    ensures 0 <= pos ==> pos < n && a[pos] == value
    ensures pos < 0  ==> forall i :: (0<= i < n) ==> a[i] != value
{
    var low, high := 0, n;
    while low < high
        decreases high - low
        invariant 0 <= low <= high <= n
        invariant forall i :: 0 <= i < n && i < low ==> a[i] != value
        invariant forall i :: 0 <= i < n && high <= i ==> a[i] != value
    {
        var mid := low + (high-low) / 2;
        if a[mid] < value         { low := mid + 1; }
        else if value < a[mid]    { high := mid; }
        else /* value == a[mid] */ { return mid; }
    }
    return -1;
}
```

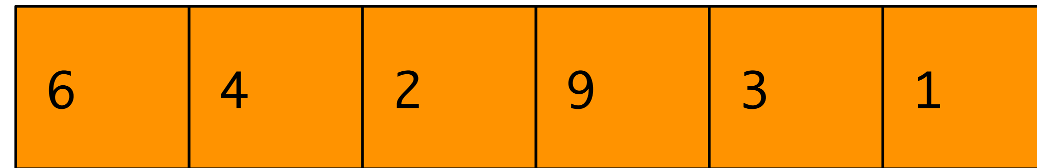
Part II
Sorting

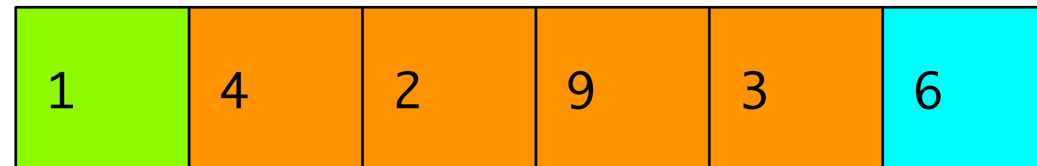
Sorting — Rationale

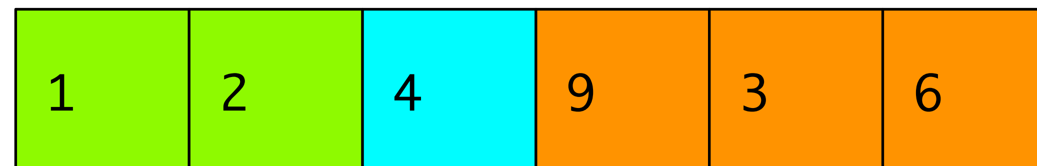
- Most sorting algorithms iteratively traverse the data structure gradually sorting its elements.
- The loop invariants capture the transient status of the sorting algorithm.
- We illustrate that using selection sort.

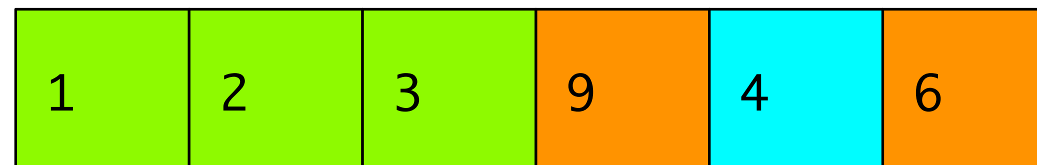


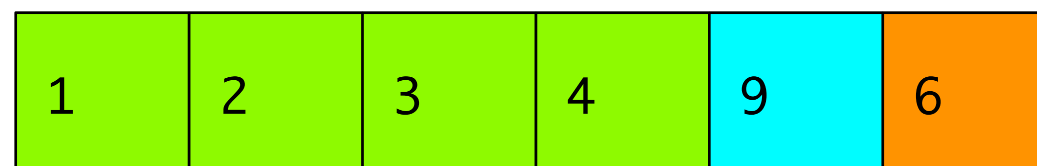
Selection Sort

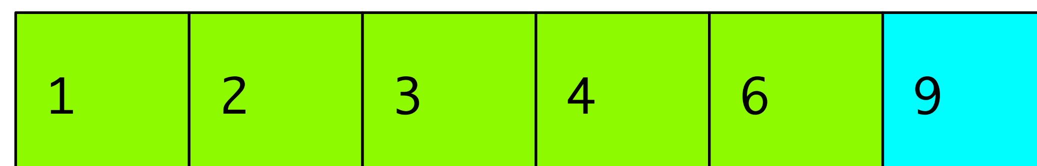


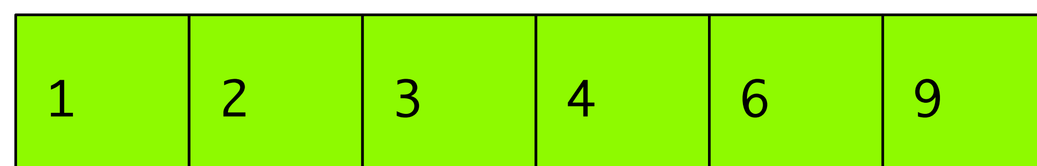








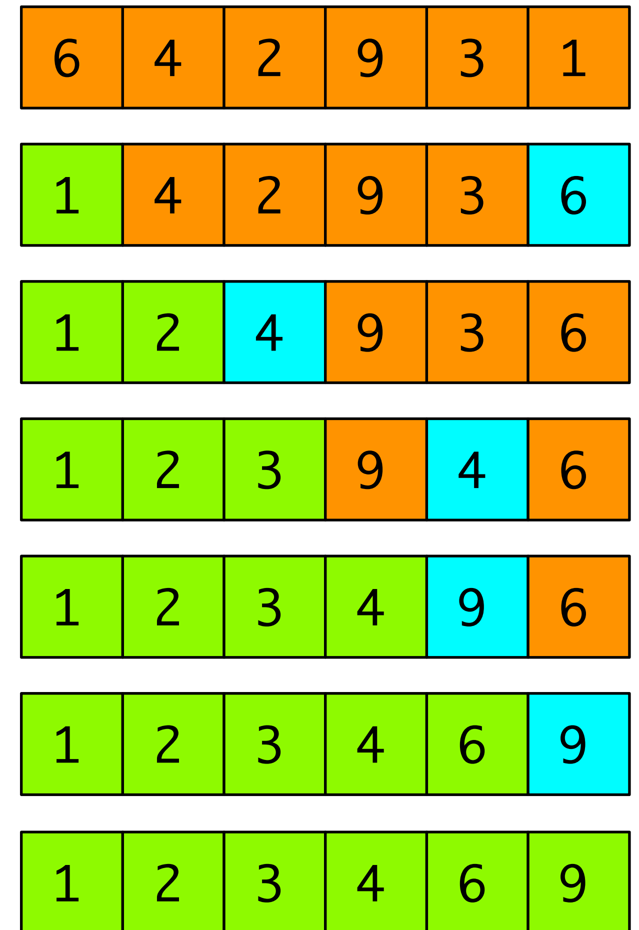




Selection Sort

- Some auxiliary predicates

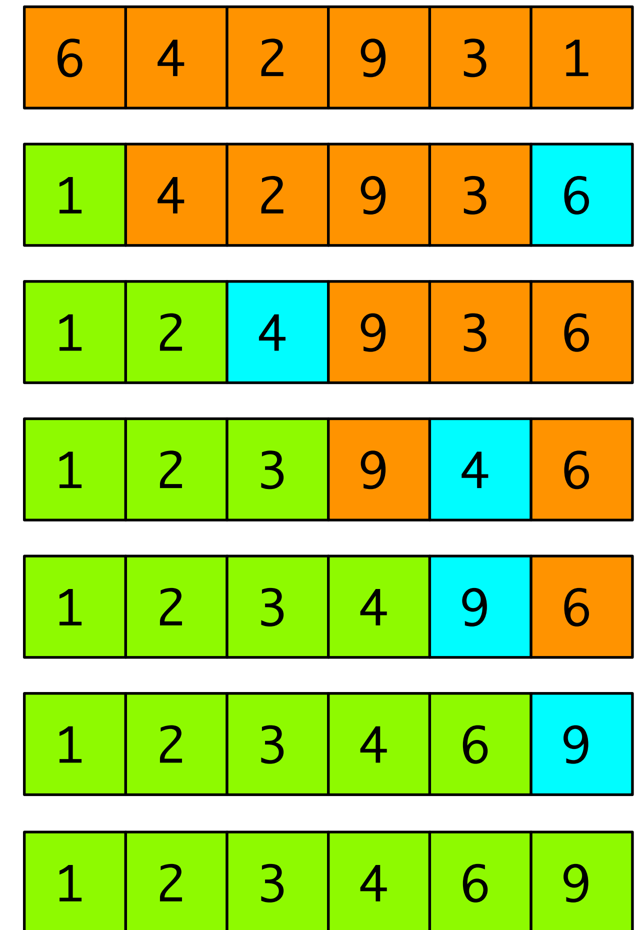
```
predicate sorted(a:array<char>, n:int)
    requires 0 <= n <= a.Length
    reads a
{ forall i, j:: (0 <= i < j < n) ==> a[i] <= a[j] }
```




```
predicate partitioned(a:array<char>,i:int,n:int)
    requires 0 <= n <= a.Length
    reads a;
{ forall k, l :: 0 <= k < i <= l < n ==> (a[k] <= a[l]) }
```

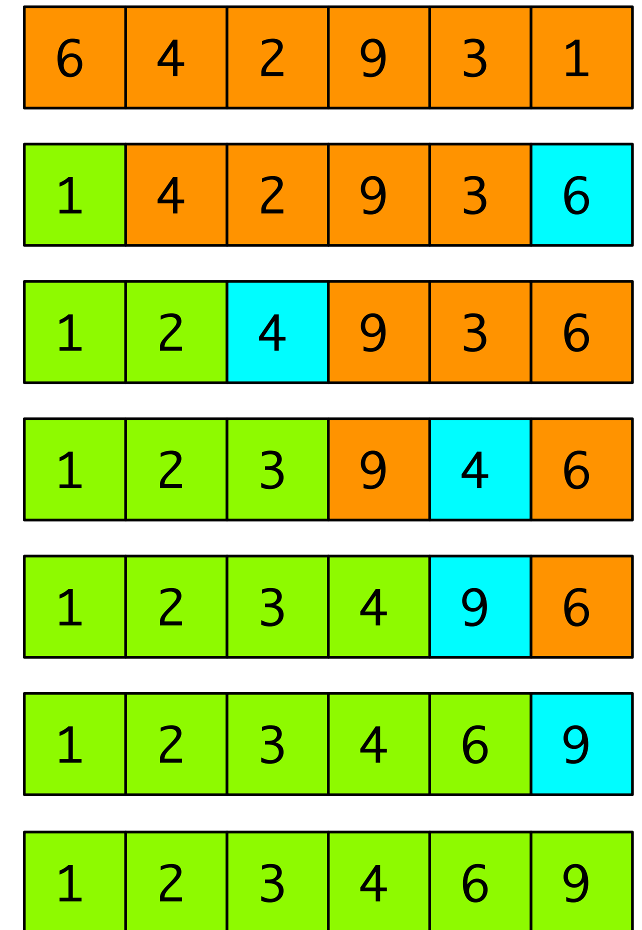
Selection Sort

```
method selectionSort(a:array<int>)
{
    var i := 0;
    while i < a.Length
    {
        selectSmaller(a,i);
        i := i+1;
    }
}
method selectSmaller(a:array<int>,i:int)
{
    var jMin := i;
    var j := i+1;
    while (j < a.Length)
    {
        if (a[j] < a[jMin]) {
            jMin := j;
        }
        j := j+1;
    }
    if (jMin != i) {
        a[i] , a[jMin] := a[jMin] , a[i];
    }
}
```



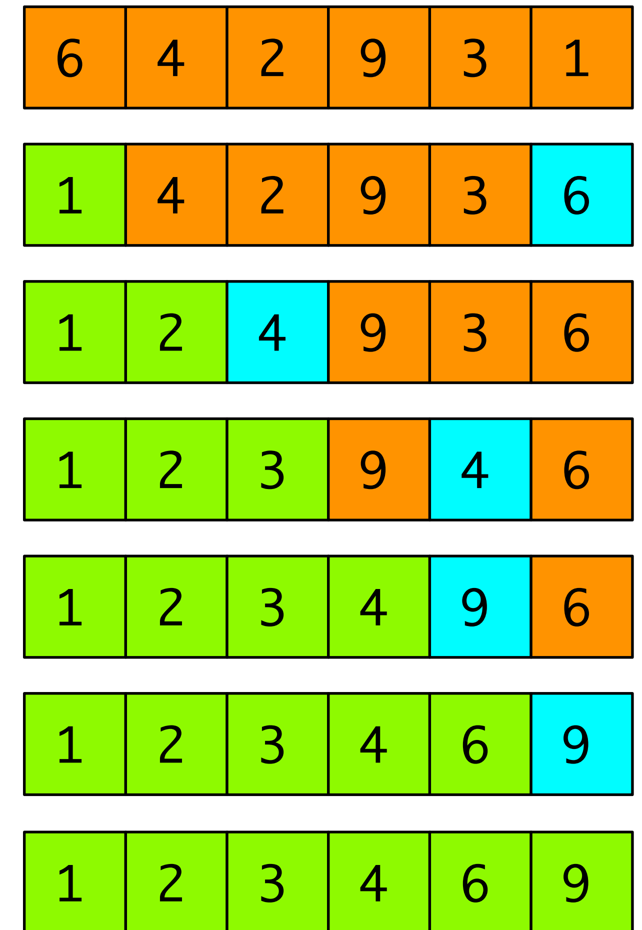
Selection Sort

```
method selectionSort(a:array<int>)
    ensures sorted(a,a.Length)
    modifies a
{
    var i := 0;
    while i < a.Length
        invariant 0 <= i <= a.Length
        invariant sorted(a,i)
        invariant partitioned(a,i,a.Length)
    {
        selectSmaller(a,i);
        i := i+1;
    }
}
method selectSmaller(a:array<int>,i:int)
    requires 0 <= i < a.Length
    requires sorted(a,i)
    requires partitioned(a,i,a.Length)
    modifies a
    ensures sorted(a,i+1)
    ensures partitioned(a,i+1,a.Length)
{ … }
```



Selection Sort

```
method selectSmaller(a:array<int>,i:int)
    requires 0 <= i < a.Length
    requires sorted(a,i)
    requires partitioned(a,i,a.Length)
    modifies a
    ensures sorted(a,i+1)
    ensures partitioned(a,i+1,a.Length)
{
    var jMin := i;
    var j := i+1;
    while (j < a.Length)
        invariant i+1 <= j <= a.Length
        invariant i <= jMin < j
        invariant forall k :: i <= k < j ==> a[jMin] <= a[k]
        invariant sorted(a, i)
        invariant partitioned(a, i, a.Length)
    {
        if (a[j] < a[jMin]) {
            jMin := j;
        }
        j := j+1;
    }
    if (jMin != i) {
        a[i] , a[jMin] := a[jMin] , a[i];
    }
}
```



Exercise

- orderedInsert

```
method insert(a:array<char>, i:int, elem:char, n:int)
  requires 0 <= n < a.Length
  requires sorted(a, n)
  modifies a
  ensures sorted(a, n+1)
```

Exercise

- orderedInsert

```
method insert(a:array<char>, i:int, elem:char, n:int)
  requires 0 <= n < a.Length
  requires sorted(a, n)
  modifies a
  ensures sorted(a, n+1)
```

- Problem: Any method that maintains the order of the elements and adds an extra one satisfies the specification! (a related issue arises in the spec. of sort from before)

Exercise

- orderedInsert

```
method insert(a:array<char>, i:int, elem:char, n:int)
  requires 0 <= n < a.Length
  requires sorted(a, n)
  modifies a
  ensures sorted(a, n+1)
{
   if (n > 0)
    { a[n] := a[n-1]; }
}
```

A better specification

- A more precise specification is needed:
  - Return the position of the inserted element.
  - State that position is valid and contains inserted element.

```
method insert(a:array<int>, n:int, e:int) returns (pos:int)
  requires 0 <= n < a.Length
  requires sorted(a, n)
  ensures sorted(a, n+1)
  ensures 0 <= pos <= n && a[pos] == e
```

A better specification

- A more precise specification is needed:
  - Return the position of the inserted element.
  - State that position is valid and contains inserted element.

```
method insert(a:array<int>, n:int, e:int) returns (pos:int)
  requires 0 <= n < a.Length
  requires sorted(a, n)
  ensures sorted(a, n+1)
  ensures 0 <= pos <= n && a[pos] == e
  modifies a
{
  var i := 0;

  while (i < n)
    invariant 0 <= i <= n
    invariant forall j :: 0 <= j < i ==> a[j] == e
    invariant sorted(a,i)
  {
      a[i] := e;
      i := i+1;
  }
  pos := i;
  a[pos] := e;

}
```

An even better specification

- A more precise specification is needed:
  - Return the position of the inserted element.
  - State that position is valid and contains inserted element.
  - All positions up to **pos** stay unchanged.
  - All positions beyond **pos** are shifted by one.

```
method insert(a:array<int>, n:int, e:int) returns (pos:int)
  requires 0 <= n < a.Length
  requires sorted(a, n)
  ensures sorted(a, n+1)
  ensures 0 <= pos <= n && a[pos] == e
  ensures forall k :: 0 <= k < pos ==> a[k] == old(a[k])
  ensures forall k :: pos < k <= n ==> a[k] == old(a[k-1])
  modifies a
{ … }
```

An even better specification

```
method insert(a:array<int>, n:int, e:int) returns (pos:int)
  requires 0 <= n < a.Length
  requires sorted(a, n)
  ensures sorted(a, n+1)
  ensures 0 <= pos <= n && a[pos] == e
  ensures forall k :: 0 <= k < pos ==> a[k] == old(a[k])
  ensures forall k :: pos < k <= n ==> a[k] == old(a[k-1])
  modifies a
{
  var i := n;
  if( n > 0 )
  { a[n] := a[n-1]; }
  while 0 < i && e < a[i-1]
    invariant 0 <= i <= n
    invariant sorted(a, n+1)
    invariant forall k :: i < k < n+1 ==> e <= a[k]
    invariant forall k :: 0 <= k < i ==> a[k] == old(a[k])
    invariant forall k :: i < k <= n ==> a[k] == old(a[k-1])
  {
    a[i] := a[i-1];
    i := i - 1;
  }
  a[i] := e;
  return i;
}
```

Part III
Verification with Bounded Arithmetic

Verification and Arithmetic

- Until now all our proofs have assumed mathematical integers.
- Our reasoning is sound, provided the code that executes using arbitrary precision integers.
  - Check the code generated by Dafny!
- What if we want to use 32-bit or 64-bit integers?
- What do we know about our implementations?
  - Overflow?
  - What happens? (Modulo arith., saturated arith., abrupt termination, etc.)

Verification with Machine Integers

- **Goal:** Prove that a program is safe with respect to overflows.
- 32-bit signed integers in two-complement representation: integers between -2^{31} and $2^{31}-1$.
- If the **mathematical** result of an operation fits in the range, that is the **computed** result.
- Otherwise, an **overflow** occurs:
    - Behavior depends on language and environment.
- A program is **safe** wrt overflow if no overflow can occur.

Verification with Machine Integers

- **Idea:** Replace all arith. operations by methods with preconditions.
- x+y becomes int32_add(x,y), and so on:

```
method int32_add(x:int,y:int) returns (r:int)
    requires -2_147_483_648 <= x+y <= 2_147_483_647
    ensures r == x+y
{
    return x+y;
}
```

- Not great: range constraints of integer must be added explicitly everywhere…

Verification with Machine Integers

- **Better Idea:** Replace int with a **refined** type int32

```
newtype int32 = x:int | -2_147_483_648 <= x <= 2_147_483_647
```

- Dafny provides projection back to **int**
- Can even replace operations with custom methods with appropriate spec:

```
method int32_add(x:int32,y:int32) returns (r:int32)
    requires -2_147_483_648 <= (x as int + y as int) <= 2_147_483_647
    ensures r == x+y
{
    return x+y;
}
```

Verification with Machine Integers

```
method BSearch(a:array<int>, n:int32, value:int) returns (pos:int32)
    requires 0 <= n as int <= a.Length && sorted(a, n as int) && a.Length <= 2_147_483_647
    ensures 0 <= pos ==> pos < n && a[pos] == value
    ensures pos < 0  ==> forall i :: (0<= i < n) ==> a[i] != value
{
    var low:int32, high:int32 := 0, n;
    while low < high
        decreases high - low
        invariant 0 <= low <= high <= n
        invariant forall i :: 0 <= i < n && i < low ==> a[i] != value
        invariant forall i :: 0 <= i < n && high <= i ==> a[i] != value
    {
        var mid := (high+low)/2;              ⓧ result of operation might violate newtype constraint for 'int32'  Verifier
        if a[mid] < value           { low := mid + 1; }
        else if value < a[mid]      { high := mid; }
        else /* value == a[mid] */ { return mid; }
    }
    return -1;
}
```

Verification with Machine Integers

```
method BSearch(a:array<int>, n:int32, value:int) returns (pos:int32)
    requires 0 <= n as int <= a.Length && sorted(a, n as int) && a.Length <= 2_147_483_647
    ensures 0 <= pos ==> pos < n && a[pos] == value
    ensures pos < 0  ==> forall i :: (0<= i < n) ==> a[i] != value
{
    var low:int32, high:int32 := 0, n;
    while low < high
        decreases high - low
        invariant 0 <= low <= high <= n
        invariant forall i :: 0 <= i < n && i < low ==> a[i] != value
        invariant forall i :: 0 <= i < n && high <= i ==> a[i] != value
    {
        var mid:int32 := low + (high-low)/2;
        if a[mid] < value          { low := mid + 1; }
        else if value < a[mid]     { high := mid; }
        else /* value == a[mid] */ { return mid; }
    }
    return -1;
}
```

Verification with Machine Integers

```
function gcd(x:int,y:int) : int
    requires y >=0
    decreases y
{
    if y == 0 then x else gcd(y, x % y)
}

method euclid(u:int32, v:int32) returns (r:int32)
    requires u >= 0 && v >= 0
    ensures r as int == gcd(u as int,v as int)
{
    r := u;
    var y := v;
    while y > 0
        invariant 0 <= y <= v
        invariant gcd(u as int,v as int) == gcd(r as int,y as int)
    {
        var tmp := y;
        y := r % y;
        r := tmp;
    }

}
```

Verification with Machine Integers

- **Caveat emptor:** Reasoning with machine integers in this way can be challenging to Dafny's automation.
- What if we want to reason **with** overflow?
- Dafny (and many related tools) support (unsigned) **bit-vectors**.
- Dafny includes a family of types bvN, where N denotes the bit vector length.

Verification with Bit-based integers

- Bit-vectors capture common “tricks”:

```
method multby2(a:bv32) returns (r:bv32)
    ensures r == a*2
{
    return a << 1;
}
method divby2(a:bv32) returns (r:bv32)
    ensures r == a/2
{
    return a >> 1;
}
method multby3(a:bv32) returns (r:bv32)
    ensures r == a*3
{
    return (a << 1)+a;
}
```

- Not very easy to use in practice / not automation friendly.
- Verification using bit-vectors is ongoing research.

Part IV
Abstract Data Types
(intro)

Abstract Data Types (Liskov, 78)

- ADTs are the building blocks for software construction
  - Consist of:
    - A description of the data elements of the type
    - A set of operations over the data elements of the ADT
  - A software system is a composition of ADTs
  - ADTs behave like regular types in a programming language
  - Promotes modularity, encapsulation, information hiding, and hence reuse, modifiability, and correctness.

ADTs (Liskov & Zilles,78)

PROGRAMMING WITH ABSTRACT DATA TYPES

Barbara Liskov
Massachusetts Institute of Technology
Project MAC
Cambridge, Massachusetts

Stephen Zilles
Cambridge Systems Group
IBM Systems Development Division
Cambridge, Massachusetts

Abstract

The motivation behind the work in very-high-level languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area. The programmer is then able to spend his effort in the right place; he concentrates on solving his problem, and the resulting program will be more reliable as a result. Clearly, this is a worthwhile goal.

Unfortunately, it is very difficult for a designer to select in advance all the abstractions which the users of his language might need. If a language is to be used at all, it is likely to be used to solve problems which its designer did not envision, and for which the abstractions embedded in the language are not sufficient.

This paper presents an approach which allows the set of built-in abstractions to be augmented when the need for a new data abstraction is discovered. This approach to the handling of abstraction is an outgrowth of work on designing a language for structured programming. Relevant aspects of this language are described, and examples of the use and definitions of abstractions are given.

Barbara Liskov (MIT)

BARBARA LISKOV
United States – **2008**

For contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing.

Abstract Data Type

Abstract types are intended to be very much like the built-in types provided by a programming language. The user of a built-in type, such as _integer_ or _integer array_, is only concerned with creating objects of that type and then performing operations on them. He is not (usually) concerned with how the data objects are represented, and he views the operations on the objects as indivisible and atomic when in fact several machine instructions may be required to perform them. In addition, he is not (in general) permitted to decompose the objects. Consider, for example, the built-in type _integer_. A programmer wants to declare objects of type _integer_ and to perform the usual arithmetic operations on them. He is usually not interested in an integer object as a bit string, and cannot make use of the format of the bits within a computer word. Also, he would like the language to protect him from foolish misuses of types (e.g., adding an integer to a character) either by treating such a thing as an error (strong typing), or by some sort of automatic type conversion.

Abstract Data Type (External View)

- External View
  - A public opaque data type (that clients will use)

    Note: opaque means = behaves as a primitive type
  - A set of operations on this data type
  - Operations must neither reveal, nor allow a client to invalidate the internal representation of the ADT
  - pre and post conditions on these operations must be expressed in terms of the abstract type (the only type known to the client)
  - This is why ADTs promote reuse, modifiability, and correctness: the developer can change the implementation anytime, without breaking contracts

Abstract Data Type (Internal View)

- Internal View
  - A **representation** data type (hidden from clients)
  - A set of operations on the representation data type
- ***Key remarks:***
  - A programmer must define the operations in such a way that the representation state (invisible to clients) is kept consistent with the intended abstract state
  - Pre-conditions on the public operations, expressed on the abstract state, must map into pre-conditions expressed in terms of the representation state
  - The same for post-conditions
  - At all times the concrete state must represent a well defined abstract state (otherwise something is wrong!)

Example (Positive Set ADT)

```
class PSet {
// an abstract set of positive numbers

    method new(sz:int) {…}
    // initializes the set ( e.g., Java constructor )

    method add(v:int) {…}
    // adds v to the set if space available

    function size() : int {…}
    // returns number of elems in the set

    function contains(v:int) : bool {…}
    // returns number of elems equal to v in the set

    function maxsize() : int {…}
    // returns max number of elems allowed in the set
}
```

Technical ingredients in ADT design

- The ***abstract state***
  - defines how client code sees the object
- The ***representation type***
  - chosen by the programmer to implement the ADT internals. The programmer is free to choose the implementation strategy (data-structures, algorithms). This is done at construction time.
- The ***concrete state***
  - in general, not all representation states are legal concrete states
  - a concrete state is a representation state that really represents some well-defined abstract state

Technical ingredients in ADT design

- The ***representation invariant***
  - The representation invariant is a condition that restricts the representation type to the set of (safe) concrete states
  - If the ADT representation falls outside the rep invariant, something is wrong (inconsistent representation state).
- The ***abstraction function***
  - maps every concrete state into some abstract state
- The ***operation pre-*** *and* ***post- conditions***
  - expressed for the representation type
  - also expressed for the abstract type (for client code)

Part V
Abstract Data Types (with objects)

Bank Account ADT

- Abstract State
  - the account balance (`bal`)
  - `bal` is of type `int` subject to the constraint (`bal >= 0`)

Bank Account ADT

- Representation type
  - an integer `bal`
  - in this simple case the representation type is the same as the abstract type
  - the true "meaning" of the representation and abstract types are different
  - not all operations on integers are valid on account balances (e.g., to multiply bank accounts)

Bank Account ADT

- Representation type
  - an integer `bal`
  - in this simple case the representation type is the same as the abstract type
  - the true “meaning” of the representation and abstract types are different
  - not all operations on integers are valid on account balances (e.g., to multiply bank accounts)
- Representation invariant
  - (`bal >= 0`)
  - this time, pretty simple

Example (Account)

```
class Account {
    var bal: int;

    predicate RepInv()
    // specifies the representation invariant
        reads this
    {
        bal >= 0
    }
   …
}
```

Example (Account)

```
class Account {
    var bal: int;

    predicate Valid()
    // specifies the representation invariant
      reads this
    {
      bal >= 0
    }

    constructor()
        ensures Valid()
    { bal := 0; }
    …
}
```

Example (Account)

```
class Account {
    var bal: int;
…
    // All operations must require the representation invariant
    // All operations must ensure the representation invariant
    method deposit(v:int)
        modifies this;
        requires Valid() && v >= 0
        ensures Valid()
    { bal := bal + v; }

    method withdraw(v:int)
        modifies this
        requires Valid() && v >= 0
        ensures Valid()
    { if (bal>=v) { bal := bal - v; } }
}
```

Example (Account)

```
class Account {
    var bal: int;
…
   function method getBal():int
        reads this
    { bal }

    method withdraw(v:int)
        modifies this;
        requires Valid() && 0 <= v <= getBal()
        ensures Valid()
    { bal := bal - v;  }
}
```

Set ADT

```
class ASet {
// an abstract Set of numbers

    constructor(sz:int) {}
    // initializes aset ( e.g., Java constructor )

    method add(v:int) {}
    // adds v to aset if space available )

    function size() : int
    // returns number of elems in aset

    function contains(v:int) : bool
    // check if v belongs to set

    function maxsize() : int
    // returns max number of elems allowed in aset

}
```

Set ADT

- Abstract State
  - a set of positive integers aset

Set ADT

- Representation type
  - an array of integers **store** with sufficient large size
  - an integer nelems counting the elements in **store**

Set ADT

- Representation type
  - an array of distinct integers **store**
  - an integer nelems counting the elements in **store**
- Representation invariant

```
(store != null) &&

(0 <= nelems <= store.Length) &&

forall k :: (0<=k<nelements) ==> forall j::(k<j<nelements)  ==> b[k] != b[j]
```

Set ADT

- Representation type
  - an array of **distinct** integers **store**
  - an integer nelems counting the elements in **store**
- Representation invariant

```
(store != null) &&

(0 <= nelems <= store.length) &&

forall k :: (0<=k<nelements) ==> forall j::(k<j<nelements)  ==> b[k] != b[j]
```

Set ADT

- Representation type
  - an array of distinct integers **store**
  - an integer nelems counting the elements in **store**
- Representation invariant

```
(store != null) &&
(0 <= nelems <= store.length) &&
forall k :: (0<=k<nelements) ==> forall j::(k<j<nelements)  ==> b[k] != b[j]
```

- Abstraction mapping
  - $\langle nelems=n, store=[v_0,v_1,\ldots v_{store.Length-1}]\rangle \rightarrow \{v_0,\ldots,v_{n-1}\}$
  - more later ....

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;

  constructor(SIZE:int)
    requires SIZE > 0
    ensures Valid()
  {
    a := new int[SIZE];
    size := 0;
  }

…
```

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;

  constructor(SIZE:int)
    requires SIZE > 0;
    ensures Valid()
  {
    a := new int[SIZE];
    size := 0;
  }

  predicate Valid()
    reads this,a;
  {
   …
  }
…
```

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;

…

  predicate RepInv()
    reads this,a
  {
    a!=null &&
    0 < a.Length &&
    0 <= size <= a.Length &&
    unique(a,0, size)
  }
…
```

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;

…
  predicate unique(b:array<int>, l:int, h:int)
  reads b
  requires b != null && 0<=l <= h <= b.Length
  {
    forall k::(l<=k<h) ==> forall j::(k<j<h)  ==> b[k] != b[j]
  }
…
```

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;

  function method count():int
  reads this,a;
  requires Valid()
  { size }

  function method maxsize():int
  reads this,a;
  requires Valid()
  { a.Length }

  method add(x:int)
  modifies this,a;
  requires Valid() && count() < maxsize()
  ensures Valid()
  {
    var f:int := find(x);
    if (f < 0) {
      a[size] := x;
      size := size + 1;
    }
  }
…
```

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;
…
  method find(x:int) returns (r:int)
  requires Valid();
  ensures -1 <= r < size;
  ensures r < 0 ==> forall j::(0<=j<size) ==> x != a[j];
  ensures r >=0 ==> a[r] == x;
  {
    var i:int := 0;
    while (i<size)
    decreases size-i
    invariant 0<=i<=size;
    invariant forall j::(0<=j<i) ==> x != a[j];
    {
      if (a[i]==x) { return i; }
      i := i + 1;
    }
    return -1;
  }
```

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;
…
  method contains(v:int) returns (f:bool)
  requires Valid()
  ensures  f <==> exists j::(0<=j<size) && v == a[j];
  ensures Valid()
  {
    var p:int := find(v);
    f := (p >= 0);
  }
}
```

Next week

- More on ADTs
- Soundness and Abstraction mapping
- Framing