Exponentiation
Exponentiation

- Computing the power of 2 can be tedious
- $2^{1024}$ requires 1023 multiplications
  - that's a lot of work
- We can use the fact that $2^{2n} = 2^n * 2^n$, and $2^{2n+1} = 2^{2n} * 2$
- $2^{1024}$ just needs 9 multiplications
  
  \[
  \begin{align*}
  a &:= 2 * 2; \\
  a &:= a * a; \quad = 2^2 * 2^2 \\
  a &:= a * a; \quad = 2^4 * 2^4 \\
  a &:= a * a; \quad = 2^8 * 2^8 \\
  a &:= a * a; \quad = 2^{16} * 2^{16} \\
  a &:= a * a; \quad = 2^{32} * 2^{32} \\
  a &:= a * a; \quad = 2^{64} * 2^{64} \\
  a &:= a * a; \quad = 2^{128} * 2^{128} \\
  a &:= a * a; \quad = 2^{512} * 2^{512}
  \end{align*}
  \]

  $2^9$ needs 4 multiplications
  
  \[
  \begin{align*}
  a &:= 2 * 2; \\
  a &:= a * a; \quad = 2^2 * 2^2 \\
  a &:= a * a; \quad = 2^4 * 2^4 \\
  a &:= a * 2; \quad = 2^8 * 2 \\
  a &:= a * a; \quad = 2^{16} * 2 \\
  a &:= a * a; \quad = 2^{32} \\
  a &:= a * a; \quad = 2^{64} \\
  a &:= a * a; \quad = 2^{128}
  \end{align*}
  \]

  The number of multiplications is the approx. \log of $n$, where $n$ is the exponent
  (e.g. $\log_2 1024 = 10$, $\log_2 9 \approx 3$)

function pow2()

The inefficient way of computing exponentials is as follows

```plaintext
function pow2(n:nat):int
{
    if (n==0) then 1 else 2 * pow2(n-1)
}
```

Note

- It requires the power $n$ to be a positive integer
- It does $n-1$ multiplications, using recursion
- Using this is ‘dumb’ for computation, but it is great for verification!
method Power2(): bare bones

- The ‘smart’ way of computing exponentials does log(n) multiplications
- It’s still recursive, calling itself twice, for even and odd cases

```java
method Power2(n:nat) returns (p:int)
{
    if n==0 {
        p := 1;
    } else if n%2 == 0 {
        p := Power2(n/2);
        p := p * p;
    } else {
        p := Power2(n-1);
        p := 2 * p;
    }
}
```

- Can we verify this?
method Power2(): specification

The postcondition is easy because we can use the function pow2().

- this function is the criteria of correctness, you could say

```daml
method Power2(n: nat) returns (p: int)
ensures p==pow2(n)
{
    if n==0 {
        p := 1;
    } else if n%2 == 0 {~ Dafny reports a postcondition violation here
        p := Power2(n/2);
        p := p * p;
    } else {
        p := Power2(n-1);
        p := 2 * p;
    }
}
```

But this does not verify: Dafny cannot prove that the postcondition is always true.
Dafny does not fail because of a logical flaw in the program. Dafny fails because it does not know the following law of exponentials:
\[ 2^a \times 2^b = 2^{a+b} \]

- in our case we have \( n \) is even, so substitute \( a := b := n/2 \)

Dafny allows users to add a new rule in the form of a lemma.
- it will try to prove the lemma correct

Here is a lemma that states the above property using the function \( \text{pow2()} \):

```plaintext
lemma lawofexponents(n:nat) requires n%2==0 ensures pow2(n) == pow2(n/2) * pow2(n/2) {
    if (n!=0) {lawofexponents(n-2);}
}
```

Dafny proves this lemma correct using induction.
method Power2(): use the lemma

You use lemmas by simply calling them at the right time

function pow2(n:nat):int { ... }  this function is called in the lemma and the method
lemma lawofexponents(n:nat) { ... }

method Power2(n: nat) returns (p: int)
ensures p==pow2(n)
{
    if n==0 {
        p := 1;
    } else if n%2 == 0 {
        lawofexponents(n);
        p := Power2(n/2);
        p := p * p;
    } else {
        p := Power2(n-1);
        p := 2 * p;
    }
}

Dafny program verifier finished with 5 verified, 0 errors.  yey!
To execute Power2()

```csharp
method Main()
{
    var power := 30;
    var number := Power2(power);
    print "2^", power," = ", number;
}
```

Dafny program verifier finished with 7 verified, 0 errors
Program compiled successfully
Running...

$2^{30} = 1073741824$

that’s about a billion
But there’s another way to provide Dafny with a new rule

- use the `assume` statement

The `assume` statement defines a rule. Dafny will not try to prove this rule.

```daml
method Power2(n: nat) returns (p: int)
ensures p==pow2(n)
{
    assume pow2(n) == pow2(n/2)*pow2(n/2);
    // notice this statement is the same as the postcondition of the lemma
    if n==0 {
        p := 1;
    } else if n%2 == 0 {
        p := Power2(n/2);
        p := p * p;
    } else {
        p := Power2(n-1);
        p := 2 * p;
    }
}
```

Dafny program verifier finished with 5 verified, 0 errors. great, we’ve avoided the lemma!
assume: other examples

What can assume do?

1. { var a:int; assume a==0; a:=a+1; assert a==1; }
   
a is assumed 0, a is incremented, and the assert is true.

2. { var a:int; assume a==2 || a==3; a:= a/2; assert a==1; }
   
both 2/2 and 3/2 equal 1, so the assert is true

3. { var a:int, b; assume a==b/2; assert b==8==>a==4; }
   
the assert is true because a is assumed to b/2

4. { var a:int; a:=0; assume a!=1; assert a==0; }
   
a is zero, a is assumed not equal to 1, so the assert is true
assume: another example

- Look for an arbitrary element in an arbitrary seq
  - Dafny will find it if we assume it’s there

```daml
method MysterySearch ()
{
  var s: seq<int>;
  var e: int;
  assume e in s;       // Dafny takes this as a ‘promise’ it’s there
  while (e != s[0])
  invariant e in s;   // why does this hold?
  decreases |s|       // needed to prove termination
  {
    s := s[1..];
  }
  assert e==s[0];    // found e (but what is it I wonder?)
}
```

HOWEVER, assumes only work for verification
If you try to run any code containing an \texttt{assume} stmt, you get

\textbf{Compilation error: an assume statement cannot be compiled (line 19)}

Do \texttt{assume} statements make life easy?

- Consider the program fragment:

  \begin{verbatim}
  var x:int := 1;
  assume x!=1;       // clearly wrong, a contradiction
  assert x==1 && x!=1; // clearly ridiculous
  \end{verbatim}

  Dafny program verifier finished with 2 verified, 0 errors

- \ldots{} from a contradiction, anything can be proved true
  - this is called \textquotedblleft\textit{ex contradictione quodlibet}\textquotedblright
From any contradiction we can prove anything, including “pigs can fly”

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Premise 1</td>
</tr>
<tr>
<td>2</td>
<td>!a</td>
<td>Premise 2</td>
</tr>
<tr>
<td>3</td>
<td>pigsfly</td>
<td>Aim: to prove this</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pigsfly</td>
<td>“Disjunction syllogism” 4 and 2</td>
</tr>
</tbody>
</table>

What does ExCG mean in Dafny?

```dsharp
method ExCQ()
{
    var x: int := -1;
    assume x==0; // a contradiction, but Dafny assumes it is true
    assert x==1; // pigsfly
    assert x==1 && x==0 && x==1; // pigsfly
}
```
Similarly, more generally, from false anything can be proved true

```
method ExCQ()
{
  var r: real;
  assume false;
  assert r==3.141592653; // supposed to be π
}
```

- and every assert after this assume is ‘assumed’ by Dafny to be true ...
- ... which is a disaster

If you assume something that is true, however, ...

```
method AssertViolation()
{
  var r: real;
  assume 1==1;  // anything that is true
  assert r==3.141592653; // this will generate an assert violation
}
```
.... so in summary

- If you assume something that really is true ... are you really sure?
  - ... they can be useful
- If you assume something that is actually false,
  - deliramentum (nonsense)

_caveat emptor_
Find
Find(): the informal specification

- We now want to write a method Find().
- The method looks for an element key in an array of integers a
- The informal spec is
  - return the index \( i \) of the element if it is found in the array
  - return \(-1\) if the element is not in the array

The spec tells us what we want.
Find() in Dafny

In Dafny, a spec consists of 2 things:

1. Preconditions
2. Postconditions

The preconditions are easy.

- Find() works on any array, ordered or unordered, so no restrictions
- The array may be empty, but it must not be null
- So must have
  
  requires a!=null
The spec said:
1. return the index \( i \) of the element if it is found in the array
2. return \(-1\) if the element is not in the array

So we can say that:
1. … if Find() returns a positive index then it is the key
   
   We’ll use the trick with short-circuiting (slide 42? Dafny Digest A)

   \[ i \geq 0 \implies i < a.\text{Length} \land a[i] = \text{key} \]

2. If Find() returns \(-1\), then the element is not in the array

   \[ i < 0 \implies (\forall k: 0 \leq k < a.\text{Length} \implies a[k] \neq \text{key}) \]
Find() specification

Putting the pre- and post-condition into the method:

```csharp
method Find(a: array<int>, key: int) returns (i:int)
requires a!=null
ensures i>=0 => i<a.Length && a[i]==key
ensures i<0 => forall k:: (0<=k<a.Length => a[k]!=key)
{
    // put some search algorithm here
}
```

- This is the spec we give the programmer
  - “give me an implementation that satisfies this spec”
- The spec is sometimes called “the contract”
Linear Search
Let’s choose linear search to implement Find()

Here we have the bare-bones code

```plaintext
i := 0;
while i < a.Length
    // need some invariants here
    {
        if a[i] == key { return; }
        i := i + 1;
    }
    i := -1;
```

We already have the pre- and post-conditions (of Find())

All we need now are the invariants
Find() using linear search

1. The invariant must say what the bounds are on \( i \) in while \( i < a.Length \)

\[
0 \leq i \leq a.Length
\]

Notice we go 1 past the loop condition (allow loop to terminate)

2. We also need an invariant that ties into the postcondition
   - The postcondition relates the index and the key
   - at all times, there is no key at values of the index lower than \( i \)
     - (if there was a key, the loop would have terminated earlier)
     - so, it must be true that ...

\[
\forall k :: 0 \leq k < i \implies a[k] \neq \text{key}
\]

   - Let’s check this is true the first time the loop is entered
     - if \( i := 0 \) then \( k < 0 \), so no \( k \) is possible
     - the implication is said to be vacuously true. Great.

maybe these 2 invariants are enough to satisfy Dafny
Find(), final version

method Find(a: array<int>, key: int) returns(i: int)

requires a != null
ensures 0<=i ==> i<a.Length && a[i]==key
ensures i<0  ==> forall k:: 0<=k<a.Length ==> a[k]!=key
{
    i:=0;
    while i<a.Length
    invariant 0<=i<=a.Length
    invariant forall k:: 0<=k<i ==> a[k]!=key
    {
        if a[i] == key { return; }
        i:=i+1;
    }
    i:=-1;
}
Binary Search
Binary search, bare bones

If the input array is sorted, then we can implement Find() using binary search. BinarySearch in Dafny is shown below (it won’t verify in this form though):

```daml
method BinarySearch (a:array<int>, key:int) returns (index:int)
{
    var low, high := 0, a.Length;
    while low < high
    {
        var mid := (low + high) / 2;
        if a[mid] < key
            {low := mid + 1;}
        else if key < a[mid]
            {high := mid;}
        else
            {return mid;}
    }
    return -1;
}
```

Binary search is what you use to find a ‘key’ in a telephone directory: open at the middle and see if the key is to the left or right, ...

Imagine using linear search on the Sydney phone directory!!!

obviously, e.g. there’s no null protector on the array.
Binary search: preconditions

- First we need to work out the preconditions
- These are different because BinarySearch has different requirements
  1. The first one is the same, protect against null \( a \neq \text{null} \)
  2. For this search, the array must be sorted
     - we’ve used a sorted predicate before. Here is one again.
       ```
predicate allsorted(a: array<int>)
requires a \neq \text{null}
reads a
{
  forall j,k:: 0\leq j<k<a.Length \implies a[j]\leq a[k]
}
```
So the preconditions are
```
requires a \neq \text{null} \&\& \text{allsorted}(a)
```
Binary search: postconditions

The rest of the spec of Find() does not need to change? It says:

1. return the index of the key if the key is in the array
2. return -1 if the key is not in the array

We convert the above into postconditions as before:

1. Use the trick with short-circuiting (slide 42? Dafny Digest A)
   \[ 0 \leq i \implies i < a.Length \land a[i] = key \]
2. If index is negative, there is no key in the whole array
   \[ i < 0 \implies \forall k:: 0 \leq k < a.Length \implies a[k] \neq key \]

So we have:

\[ \text{ensures } 0 \leq i \implies i < a.Length \land a[i] = key \]
\[ \text{ensures } i < 0 \implies \forall k:: 0 \leq k < a.Length \implies a[k] \neq key \]
Find() on an ordered array, using binary search

method Find(a: array<int>, key: int) returns(i: int)
requires a != null && allsorted(a)
ensures 0<=i ==> i<a.Length && a[i]==key
ensures i<0 ==> forall k:: 0<=k<a.Length ==> a[k]!=key
{
  var low, high := 0, a.Length;
  while low < high
    invariant 0 <= low <= high <= a.Length
    invariant forall i:: 0<=i<a.Length && !(low<=i<high) ==> a[i]!=key
    {
      var mid := (low + high) / 2;
      if a[mid] < key {low := mid + 1;}
      else if key < a[mid] {high := mid;}
      else {return mid;}
    }
  return -1;
}
Insertion Sort
Insertion sort

- **Basic idea**
  - Ordered part followed by unordered part
    - … initially ordered is 1 element
  - One by one move elements from unordered to ordered
    - … until no elements left
Insertion Sort: what’s really happening

shuffle

E -> D -> C -> B -> A

D -> E -> C -> B -> A

C -> D -> E -> B -> A

B -> C -> D -> E -> A

A -> B -> C -> D -> E

swap

E <-> D <-> C <-> B <-> A

D <-> E <-> C <-> B <-> A

C <-> D <-> E <-> B <-> A

B <-> C <-> D <-> E <-> A

A <-> B <-> C <-> D <-> E
Insertion Sort: looking more closely

shuffle: last ‘step’

Before loop:

B → C → D → E → A

Loop starts:

A → B → C → D → E

After loop:

B → A → B → C → D → E

swap: last ‘step’

Before loop:

B ↔ C ↔ D ↔ E ↔ A

Loop starts:

B ↔ C ↔ D ↔ A ↔ E

After loop:

A ↔ B ↔ C ↔ D ↔ E
(Shuffling) insertion sort: bare bones

method InsertionSort (a:array<int>)
modifies a;                                      // tell verifier ‘a’ will be modified
{
    var up:=1;                                    // start at first element
    while (up < a.Length) {                      // ... and iterate to the end
        var down := up-1;                        // up-1 is index of last sorted element
        var temp:=a[up];                         // up is index of first unsorted element
        a[up]:=a[down];                          // overwrite a[index]: a first shuffle
        while (down >= 0 && temp < a[down])     // count down & compare
        {
            a[down+1]:=a[down];                 // shuffle element up
            down:=down-1;                       // move counter down
        }
        a[down+1]:=temp;                       // reached right place, put temp back
        up:=up+1;                             // take next unsorted element
    }
}
(Shuffling) insertion sort: precondition

method InsertionSort (a:array<int>)
  requires a!=null && a.Length>1       // get rid of null && insist on >1 element
  modifies a;                       // tell verifier ‘a’ will be changed
  {
    var up:=1;                       // start at first element
    while (up < a.Length) {
      var down:=up-1;                // up-1 is index of last sorted element
      var temp:=a[up];               // up is index of first unsorted element
      a[up]:=a[down];                // overwrite a[index]: a first shuffle

      while (down >= 0 && temp < a[down]) // count down & compare
        {
          a[down+1]:=a[down];         // shuffle element up
          down:=down-1;               // move counter down
        }
      a[down+1]:=temp;               // reached right place, put temp back
      up:=up+1;                      // take next unsorted element
    }
  }
(Shuffling) insertion sort: postcondition

Need

- Pre-condition ✔
- Post-condition, ✔
  - is the same as the precondition of Binary Search! The array is sorted.

Invariants.

We’ll need a function to specify a sorted sub-array between given bounds

- It’s very similar to allsorted() from BinarySearch(). New is that it allows bounds.

```java
predicate sorted (a:array<int>, low:int, high:int)
requires a!=null
requires 0<=low<=high<=a.Length
reads a
{
    forall j,k: low<=j<k<high ==> a[j]<=a[k]
}
```

sorted(a, 0, a.Length) is the same as allsorted(a) of course.
(Shuffling) insertion sort: spec

- So we now have a complete spec

  ```java
  method InsertionSort (a:array<int>)
  requires a!=null & a.Length>1
  ensures sorted(a, 0, a.Length)
  modifies a
  {
    ...
  }
  ```

- Now we need to find invariants for the 2 while loops (outer and inner).
  - An invariant to fix bounds on the indices
  - One or more invariants to tie the loop condition and postcondition
(Shuffling) insertion sort: invariants

- **Outer** while (up < a.Length) { ... 
  - Use invariant 1<=up<=a.Length ✔

- **Inner** while (down >= 0 && temp < a[down]) { ... 
  - This is harder... both down and temp probably should appear
  - What does the loop actually do?
    - ... we decrement down and
    - ... ‘shuffle’ over elements that are bigger than temp
    - in fact, all the elements between down .. up are bigger than temp
    - that is: for all k between down and up ==> a[k]>temp
  - Use invariant forall k:: down<k<up ==> a[k]>temp ✔
(Shuffling) insertion sort: invariants

Now find invariants to tie the loop to the postcondition

- Outer \( \text{while} \ (\text{up} < a.\text{Length}) \ \{ \ ... \) 
  - When we complete an iteration, array between 0..up is sorted.
  - Use invariant \( \text{sorted}(a, 0, \text{up}) \)

- Inner \( \text{while} \ (\text{down} >= 0 \ \&\& \ \text{temp} < a[\text{down}]) \ \{ \ ... \)
  - We save \( a[\text{up}] \) in temp ...
    - shuffle each element up until correct location for temp is found
  - The lower part of array is sorted until location up+1
  - Use invariant \( \text{sorted}(a, 0, \text{up}+1) \) 
    - can you see why it's +1 more?
  - gee, \text{sorted}() \text{ came in handy}
verified (shuffling) insertion sort

method InsertionSort (a:array<int>)

requires a!=null && a.Length>1
ensures sorted(a, 0, a.Length) we defined this earlier
modifies a

{ var up := 1;
  while (up < a.Length) // outer loop

    invariant 1 <= up <= a.Length;
    invariant sorted(a, 0, up);

    { var down := up-1;
      var temp := a[up];
      a[up] := a[down];
      while (down >= 0 && temp < a[down]) // inner loop

        invariant forall k:: down < k < up => a[k]>temp;
        invariant sorted(a, 0, up+1);

        { a[down+1] := a[down];
          down := down-1;
        }
      a[down+1] := temp;
      up := up+1;
    }
}
Upfront
What’s the problem?

Re-arrange an array of natural numbers so all 0’s are upfront

- Must be in situ
- No other data structures are allowed
- The order of the non-zero elements may change

Example:
- input 2, 0, 1, 5, 3, 0, 4, 0 …
- output 3 and the rearranged array is 0, 0, 0, 5, 3, 2, 4, 1

Strategy:
- Traverse array with 2 index variables
- Swap elements at indices when a zero is found
- Return the number of zeros
method Upfront(a:array<int>) returns (nonz: int)
requires a!=null
modifies a;
{
    nonz := 0;
    var next := 0;
    while (next != a.Length)
        invariant 0<=nonz<=next<=a.Length
        {
            if (a[next]==0) {
                a[next], a[nonz] := a[nonz], a[next];
                nonz:=nonz+1;
            }
            next:=next+1;
        }
}
The postcondition must state:

- All elements could be zero, or none, so the returned number should be in range
- All elements below the index `nonz` are zeros
- All elements above and including the index `nonz` are non-zero

\[
\text{ensures } 0 \leq \text{nonz} \leq \text{a.Length}
\]
\[
\text{ensures } \forall i :: 0 \leq i < \text{nonz} \implies a[i] = 0
\]
\[
\text{ensures } \forall i :: \text{nonz} \leq i < \text{a.Length} \implies a[i] \neq 0
\]

The while loop is

\[
\text{while (next} \neq \text{a.Length)}
\]

- The indices must be in range, and `nonz` \(\leq\) `next`, and ...
- We need invariants to match the postcondition

\[
\text{invariant } 0 \leq \text{nonz} \leq \text{next} \leq \text{a.Length}
\]
\[
\text{invariant } \forall i :: 0 \leq i \leq \text{nonz} \implies a[i] = 0
\]
\[
\text{invariant } \forall i :: \text{nonz} \leq i \leq \text{next} \implies a[i] \neq 0
\]
method Upfront(a: array<int>) returns (nonz: int)
requires a!=null
ensures 0 <= nonz <= a.Length
ensures forall i:: 0 <=i <nonz ===> a[i]==0
ensures forall i:: nonz <=i <a.Length ===> a[i]!=0
modifies a;
{
    nonz := 0;
    var next := 0;
    while (next != a.Length)
    invariant 0 <= nonz<=next<= a.Length
    invariant forall i:: 0 <=i <nonz ===> a[i]==0
    invariant forall i:: nonz<=i <next ===> a[i]!=0
    {
        if (a[next]==0) {
            a[next], a[nonz] := a[nonz], a[next];
            nonz:=nonz+1;
        }
        next:=next+1;
    }
}
Testing Upfront

Executing the method upfront:
method Main()
{
    var a: array<int> := new int[8];
    a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7] := 2, 0, 1, 5, 3, 0, 4, 0;
    print a[..], '
';
    var n := Upfront(a);
    print "number of zeros =", n,'\n';
    print a[..], 'n';
}
results in ...

Running...
[2, 0, 1, 5, 3, 0, 4, 0]
number of zeros =3
[0, 0, 0, 5, 3, 2, 4, 1]
Het probleem van de Nederlandse nationale vlag
What’s the problem?

In his book *The Discipline of Programming*, Edsger Dijkstra proposed:

*Given a number of white, blue and red balls in random order, arrange them in the order of the colours of the Dutch national flag.*

Example:

![Ball arrangement example](image)

Also called the tri-partitioning problem
The best performing strategy:

- store the ‘unsorted balls’ in an array
- partition the array into 4 sections
  - Red, initially empty
  - White, initially empty
  - Unsorted, initially the whole array
  - Blue, initially empty
- use 3 pointers, indicating
  - Start of white balls
  - Start of unsorted balls
  - Start of blue balls

![Diagram of array with pointers]
Solution strategy

If next==BLUE  
then decrement blue & swap(A.next, A.blue)

If next==WHITE then increment next

If next==RED  
then swap(A.next, A.white) & increment next & increment white

The algorithm terminates when next == blue

Notice:

- It is a sorting algorithm
- The array is sorted in situ: no other data structure is used
- The performance is linear
datatype Colour = RED | WHITE | BLUE

method FlagSort(flag: array<Colour>) returns (white: int, blue: int)
requires flag!=null
modifies flag;
{
    white := 0;
    var next := 0;
    blue := flag.Length;  // colours between next and blue unsorted
    while (next != blue)  // when next==blue, no colours left to sort
        invariant 0 <= white<=next<=blue <= flag.Length;
        {
            match (flag[next]) {
                case WHITE => next:=next+1;
                case BLUE  => blue:=blue-1;
                    flag[next], flag[blue] := flag[blue], flag[next];
                case RED   => flag[next], flag[white] := flag[white], flag[next];
                    next:=next+1;
                    white:=white+1;
            }
        }
}

Even with no postcondition, Dafny checks array bounds. It reports index out-of-bounds if this ‘little’ invariant is not included.

This code can be executed
Verification flesh

- The postcondition includes variables white and blue ...
  - ... because these variables are returned by the method

ensures $0 \leq \text{white} \leq \text{blue} \leq \text{flag.Length}$
ensures $\forall i :: 0 \leq i < \text{white} \implies \text{flag}[i] = \text{RED}$
ensures $\forall i :: \text{white} \leq i < \text{blue} \implies \text{flag}[i] = \text{WHITE}$
ensures $\forall i :: \text{blue} \leq i < \text{flag.Length} \implies \text{flag}[i] = \text{BLUE}$

- The invariant of the loop while $(\text{next} \neq \text{blue})$ ... is almost identical
  - The invariant must also include next because it is in the condition
  - White balls only go to next: from next to blue is unsorted

invariant $0 \leq \text{white} \leq \text{next} \leq \text{blue} \leq \text{flag.Length}$
invariant $\forall i :: 0 \leq i < \text{white} \implies \text{flag}[i] = \text{RED}$
invariant $\forall i :: \text{white} \leq i < \text{next} \implies \text{flag}[i] = \text{WHITE}$
invariant $\forall i :: \text{blue} \leq i < \text{flag.Length} \implies \text{flag}[i] = \text{BLUE}$
datatype Colour = RED | WHITE | BLUE

method FlagSort(flag: array<Colour>) returns (white: int, blue: int)

requires flag!=null
ensures 0 <= white<=blue <= flag.Length
ensures forall i:: 0 <= i <white => flag[i]==RED
ensures forall i:: white <=i <blue => flag[i]==WHITE
ensures forall i:: blue <= i <flag.Length => flag[i]==BLUE
modifies flag;
{
    white := 0;
    var next := 0;
    blue := flag.Length; // colours between next and blue unsorted
    while (next != blue) // when next==blue, no balls left to sort
    invariant 0 <= white<=next<=blue <= flag.Length
    invariant forall i:: 0 <=i<white => flag[i]==RED
    invariant forall i:: white<=i<next => flag[i]==WHITE
    invariant forall i:: blue <=i<flag.Length => flag[i]==BLUE
    {
        match (flag[next])
        {
            case WHITE => next:=next+1;
            case BLUE  => blue:=blue-1;
                flag[next], flag[blue] := flag[blue], flag[next];
            case RED   => flag[next], flag[white] := flag[white], flag[next];
                next:=next+1;
                white:=white+1;
        }
    }
Testing FlagSort

Executing the method FlagSort:

```csharp
method Main()
{
    var flag: array<Colour> := new Colour[6];
    flag[0], flag[1], flag[2], flag[3], flag[4], flag[5]
    := BLUE, RED, WHITE, BLUE, RED, BLUE;
    var a, b := FlagSort(flag);
    print a, ' ', b,'\n';
    print flag[..], '\n';
}
```

results in ...

Running...
2 3
[Colour.RED, Colour.RED, Colour.WHITE, Colour.BLUE, Colour.BLUE, Colour.BLUE]
Partition
What’s the problem?

Partition an array into:
- those that are less than some number in the array (put on the left)
- those that are greater than (or equal to) some number (right)

For example
- 9 5 4 2 8 7 3 1 5 where we want to partition on element at end
- 1 3 4 2 5 7 5 9 8 notice numbers are partitioned, not sorted

Restrictions
- in place
- no other data structure may be used

Why is partitioning interesting?
- fundamental to quicksort, the most popular sort in the world
  - quicksort is fast, much faster than insertion sort for example
Strategy

- Pick the number at the end (at index $a.Length-1$) as pivot
- Introduce 2 index variables: $i$ and $j$
- $i$ starts at index 0, $j$ starts at index $a.Length-2$

1. Increment $i$ until $a[i] \geq$ pivot
2. Decrement $j$ until $a[j] <$ pivot
3. Swap $a[i]$ and $a[j]$
4. Repeat 1. to 3. until $i == j$
5. Swap $A[j]$ and pivot

\[1 \quad 5 \quad 3 \quad 4 \quad 2 \quad 8 \quad 5 \quad 7 \quad 3 \quad 5 \quad 1 \quad 9 \quad 5 \quad 8\]

\[
\uparrow \quad \ldots \quad \uparrow \quad i == j \quad \ldots \quad \uparrow \quad j
\]

- no incr or decr
- swap 9 and 1
- no incr or decr
- swap 5 and 3
- incr $i$ across 4 then 2
- decr $j$ across 7 then 8
- $i == j$
- swap 5 and 8
- voila!
Strategy continued

- Strategy is one of many to partition an array
- This strategy involves 3 (!) while loops
  - External loop: i and j dance toward each other (steps 1 to 3)
  - Internal loop for i to skip (l⇒r) across correctly positioned numbers
  - Internal loop for j to skip (l⇐r) across correctly positioned numbers
- For verification, 3 sets of invariants! ...gasp, sounds horrible

- This ‘strategy’ is linear:
  - the index variables i and j each traverse half the array (on average)
method Partition(a: array<int>) returns (i: int)
requires a!=null && a.Length>1
modifies a;
{
    var low := 0;
    var high := a.Length-1;
    var pivot := a[high];
    i := 0;
    var j := high-1;
    while (i < j)                      // external loop
        invariant low <= i<=j<= high-1
    {
        while (i<j && a[i]<pivot)       // internal loop i
            invariant low<=i<=j
            {i:=i+1;}
        while (i<j && a[j]>=pivot)      // internal loop j
            invariant i<=j<=high-1
            {j:=j-1;}
        if (i<j) {
            assert a[i]>=pivot>a[j];  // just making sure swap is required
            a[i], a[j] := a[j], a[i];
            i:=i+1;
        }
    }
    a[i], a[high] := pivot, a[i];
}
Specification

The postcondition says the returned index $i$ satisfies:

- ensures $0 \leq i < a.\text{Length}$
- ensures $\forall k :: 0 \leq k < i \implies a[k] < a[i]$ which says that everything to the left of $i$ is $<$ than the pivot
- ensures $\forall k :: i < k < a.\text{Length}-1 \implies a[k] \geq a[i]$ which says that everything to the right of $i$ is $\geq$ than the pivot

Together with the requires, we now have a complete specification.
Invariants

- **External loop invariant**
  - everything to left of \( i \) is < pivot
  - everything to right of \( j \) is \( \geq \) pivot

\[
\text{invariant low} \leq i \leq j \leq \text{high-1} \\
\text{invariant forall } k :: \text{low} \leq k < i \implies a[k] < \text{pivot} \\
\text{invariant forall } k :: j < k < \text{high} \implies a[k] \geq \text{pivot}
\]

- **Internal loop left to right**
  - same again, everything to left < pivot

\[
\text{invariant low} \leq i \leq j \\
\text{invariant forall } k :: \text{low} \leq k < i \implies a[k] < \text{pivot}
\]

- **Internal loop right to left**
  - same again, everything to right \( \geq \) pivot

\[
\text{invariant } i \leq j \leq \text{high-1} \\
\text{invariant forall } k :: j < k < \text{high} \implies a[k] \geq \text{pivot}
\]
while (i < j)                      // external loop
    invariant low <= i<=j<= high-1
    invariant forall k::: low<=k<i  ==>  a[k]<pivot
    invariant forall k::: j<k<high  ==>  a[k]>=pivot
{
    while (i<j && a[i]<pivot)       // internal loop i
        invariant low<=i<=j
        invariant forall k::: low<=k<i  ==>  a[k]<pivot
        {i:=i+1;}
    while (i<j && a[j]>=pivot)      // internal loop j
        invariant i<=j<=high-1
        invariant forall k::: j<k<high  ==>  a[k]>=pivot
        {j:=j-1;}
    if (i<j) {
        assert a[i] >= pivot > a[j];
        a[i], a[j] := a[j], a[i];
        i:=i+1;
    }
}
Quicksort
quicksort performance

- Recursive sort that can be very fast if number of items $n$ large
  - $n \log_e n$ (on average)
  - $n^2$ in the worst case (reverse ordered)
- Why?
  - **Partitioning** the array is linear, hence order $n$
  - Each **partition** halves the length of the array (on average)
  - Number of **partitions** to halve from $n$ to 1 is order $\log n$
  - Result is order $n \times \log n$
- What’s the difference in performance?
  - If there are 100,000,000 items
    - **Insertion sort** takes $(10^8)^2$ seconds $\approx 4$ months (on a ‘nano-computer’) 
    - **Quicksort** takes $10^8 \log_e 10^8$ seconds $\approx 2$ seconds
quicksort

quicksort calls a partitioning algorithm:

- partition puts the pivot element in its correct position
- left part: elements are smaller than pivot element
- right part: elements are equal to or larger than pivot element

quicksort calls itself on the smaller left and right parts

- left and right parts get smaller and smaller

**Q: Where exactly does sorting happen?**

**A: It’s subtle, but quicksort sorts in 2 ways:**

- putting $a[pivot]$ into its correct position
- producing ordered parts: left half $< a[pivot] \leq$ right half
method QuickSort (a:array<int>, low:int, high:int)
requires a!=null && a.Length>=1
requires 0<=low<=high<=a.Length
decreases high-low
modifies a
{  if (high-low>1)
      {  var pivot := partition(a, low, high); // partition and return pivot
         QuickSort (a, low, pivot);            // pivot is used here
         QuickSort (a, pivot+1, high);        // and here
      }  }
the element at this index is ‘sorted’!!
Partitioning is at the heart of quicksort

We saw a partitioning algorithm written in Dafny earlier
- we choose the last element in the array as pivot element
- 2 index variables move towards each other, swapping out-of-place elements …
  - … until all elements are in their correct part
  - last step is move the pivot element to its correct position

We use a different algorithm here
- we choose the first element as pivot element
- ‘shuffle’ the pivot to the right until it is in its correct position
  - … making sure along the way that elements are in the correct half
method partition (a:array<int>, low:int, high:int) returns (pivot:int)
requires a!=null && a.Length>0
requires 0<=low<high<=a.Length
ensures 0<=low<=pivot<=high<=a.Length
modifies a
{ pivot := low;
  var indx := low+1;
  while (indx < high)
    invariant low<=pivot<indx<=high
    { if (a[indx] < a[pivot])
      { var stor := a[indx];
        a[indx] := a[indx-1];
        var back := indx - 1;
        while (back > pivot)
          invariant a[pivot] > stor
          { a[back+1] := a[back];
            back := back-1
          }
        a[pivot+1] := a[pivot];
        a[pivot] := stor;
        pivot := pivot+1;
      }
    indx := indx+1;
  }
First 3 help methods

**predicate sorted** (a:array<int>, low:int, high:int)
requires a != null
requires 0<=low<=high<=a.Length
reads a

{ forall j,k::low <= j < k < high ==> a[j] <= a[k] }

**predicate sandwich** (a:array<int>, lo:int, hi:int)
requires a!=null
reads a

{ (0<lo<=hi<=a.Length ==> forall j:: lo<=j<hi ==> a[lo-1]<=a[j]) &&
  (0<=lo<=hi<a.Length ==> forall j:: lo<=j<hi ==> a[j]<a[hi]) }

**predicate pivotpart** (a:array<int>, lo:int, pivot:int, hi:int)
requires a!=null
requires 0<=lo<=pivot<hi<=a.Length
reads a

{ (forall j:: lo<=j<pivot ==> a[j]<a[pivot]) &&
  (forall j:: pivot<j<hi ==> a[pivot]<=a[j]) }
method QuickSort (a:array<int>, low:int, high:int)
requires a!=null && a.Length>=1
requires 0<=low<=high<=a.Length
requires sandwich (a, low, high)
ensures sandwich (a, low, high)
ensures sorted (a, low, high)
ensures forall j:: 0<=j<a.Length && !(low<=j<high) ==> a[j]==old(a[j]);
decreases high-low
modifies a
{
    if (high-low>1)
    {
        var pivot := partition(a, low, high);
        QuickSort(a, low, pivot);
        QuickSort(a, pivot+1, high);
    }
}
method partition (a:array<int>, low:int, high:int) returns (pivot:int)
requires a!=null && a.Length>0
requires 0<=low<high<=a.Length
requires sandwich (a, low, high)
ensures sandwich (a, low, high)
ensures 0<=low<=pivot<high<=a.Length
ensures pivotpart (a, low, pivot, high)
ensures forall j:: 0<=j<a.Length && !(low<=j<high) ==> a[j]==old(a[j]);
modifies a
pivot := low;
var indx := low+1;
while (indx < high)
invariant low<=pivot<indx<=high
invariant pivotpart (a, low, pivot, indx)
invariant sandwich (a, low, high)
invariant forall j:: 0<=j<a.Length && !(low<=j<high) => a[j]=old(a[j]);
{  if (a[indx] < a[pivot])
    { var stor := a[indx];
      a[indx] := a[indx-1];
      var back := indx - 1;
      while (back > pivot)
        invariant a[pivot] > stor
        invariant pivotpart(a, low, pivot, indx+1)
        invariant sandwich (a, low, high)
        invariant forall j:: 0<=j<a.Length && !(low<=j<high) => a[j]=old(a[j]);
        { a[back+1] := a[back];
          back := back-1;
        }
      a[pivot+1] := a[pivot];
      a[pivot] := stor;
      pivot := pivot+1;
    }
    indx := indx+1;
  }
}
method Main()
{
    var a := new int[7];
    a[0], a[1], a[2], a[3], a[4], a[5], a[6] := 3, 9, 1, 4, 2, 9, 0;
    print a[..], '\n';
    QuickSort(a, 0, 7);
    print a[..], '\n';
}

Running...
[3, 9, 1, 4, 2, 9, 0]
[0, 1, 2, 3, 4, 9, 9]
queues
Queue

- Typically, queues have 3 main operations:
  - getq (dequeue)
    - returns the first element put in the queue
  - putq (enqueue)
    - places an element on the queue
  - emptyq
    - returns true if the queue is empty, otherwise false

- A queue is a data structure, but a ‘higher order’ one
  - Implemented as another ‘lower order’ data structure
    - As an array (most commonly)
    - Linked list
    - Set of objects
What is verification of a higher-order data structure?
- shows that the mapping from higher to lower order in consistent
- the operations do what you expect

It involves using another data structure!

For example, with queues, we have
- the queue data structure and its operations
- an array data structure on which the queue is implemented
- and another data structure, a sequence, to verify the operations do what they are supposed to on the array
  - the sequence is there only for verification purposes
  - It therefore is defined as a ghost variable (not compiled into code)
queue verification set-up

class { :autocontracts } Queue<Data> { // a queue of data
    ghost var shadow: seq<Data>; // this is the verification data struct
    var a: array<Data>; // here's the implementation data struct
    var m:int, n:int; // really important index variables

    predicate Valid() { // to do with :autocontracts
        a!=null && a.Length!=0 && 0<=m<=n<=a.Length && shadow==a[m..n]
    }

    constructor()
    ensures shadow==[]; {
        a, m, n := new Data[10], 0, 0; // initialise the implementation
        shadow:=[ ]; // initialise the verification
    }
}
Empty() is a method that tests whether the array is ‘empty’
generates true if the implementation ‘says’ the array is empty
the verification must prove the sequence is empty

method Empty () returns (e:bool)
ensures e <=> |shadow|==0       // e==true  ➔ empty seq & the reverse
{
    e := m==n;                // e is true if m and n are the same
}
method Pop() returns (d: Data)
requires shadow != [];
ensures d == old(shadow)[0]  // ‘d’ must be same as first in seq
ensures shadow == old(shadow)[1..]  // remove first element from seq
{
    d, m := a[m], m+1;
    shadow := shadow[1..];
}

The ‘old’ in the postcondition is very important: it tells the verifier to use ‘shadow’ before it gets modified in this method. So the postcondition maps the ‘old’ to the ‘new’.
method Push (d: Data)
ensures shadow == old(shadow) + [d]  // easy to prove you would think
{
    a[n], n := d, n+1;                // add new data to the array
    shadow := shadow + [d];         // add new data to the old sequence
}

This method generates \textbf{Error: index out of range} on the first line of the method

- statement \( n := n+1 \) may overflow array causing \( a[n] \) to go out of bounds

What would happen if I put \( \text{if } n < \text{a.Length}\{ \ldots \} \) around the body?
method Push(d: Data)
    ensures shadow == old(shadow) + [d];
{
    if n == a.Length
    {
        var b := a;               // array copy
        if m == 0
        {                          // if really choc-a-bloc
            b := new Data[2*a.Length]; } // new array 2x as long
        forall (i | 0<=i<n-m )     // use instead of while-loop
            { b[i] := a[m+i]; }       // move data to start
        a, m, n := b, 0, n-m;      // copy back and adjust m,n
    }
    a[n], n := d, n+1;
    shadow := shadow + [d];
}
Aside : autocontracts

- The autocontracts parameter in the statement:
  class {::autocontracts} SomeDataStruct<Data>

  - is a time-saver: Dafny adds ‘things’ to the spec to help you
  - the predicate Valid() plays a special role here:
    - it links the implementation to the verification
    - it defines a ‘universal’ state that must always be valid
      - the statement
        ```
        ensures Valid()
        ```
        is added to the constructor and very method in the class
      - the statement
        ```
        requires Valid()
        ```
        is added to every method in the class
  - additionally, modifies clauses are handled automatically
queue: testing ... 

A test program is:

```csharp
method Main()
{
    var q := new Queue<int>();
    q.Push(123456789);
    var i := q.Pop();
    print "pushed and popped ", i,'\n';
}
```

Running...
pushed and popped 123456789
A more interesting test program is:

```csharp
method Main()
{
    var q := new Queue<char>();
    q.Push('A'); q.Push('B'); q.Push('#');
    var g:char := q.Pop(); assert g == 'A';
        g := q.Pop(); assert g == 'B';
        g := q.Pop(); assert g == '#';
    print "last pop = ", g, '\n';
    var e:bool := q.Empty();
    if e {print "test okay\n";} else {print "queue not empty!\n";}
}
```

Running...
last pop = #
test okay