Multiplication and Powers

The Russian peasant method for multiplication and raising to powers of integers and modular integers

```
// Integer multiplication using the Russian peasant algorithm
// in a recursive method.

// See IntPow.dfy.

// Dafny requires no additional reasoning to accept that
// this works.

method IntMulRecursive( x: int, y: int) returns ( p: int )
    requires y >= 0
    decreases y
    ensures p == x*y
{
    if y == 0 { return 0; }
    p := IntMulRecursive(x+x,y/2);
    if y%2 == 1 { p := p+x; }
}
```

```
// Integer multiplication using the Russian peasant method
// in a loop.
method IntMulLoop( x: int, y: int) returns ( p: int )
    requires y >= 0
    ensures p == x*y
{
    p := 0;
    var q, r := x, y;
    while r != 0
        invariant 0 <= r
        decreases r
        invariant x*y == p+q*r
    {
        if r%2 == 0 { r := r/2; q := q+q; }
        else        { r := r-1; p := p+q; }
    }
}
```

```
// Integer multiplication using the Russian peasant method
// in a loop. Alternative version.
method IntMulLoop( x: int, y: int) returns ( p: int )
    requires y >= 0
    ensures p == x*y
{
    p := 0;
    var q, r := x, y;
    while r != 0
        invariant 0 <= r
        decreases r
        invariant x*y == p+q*r
    {
        if r%2 == 1 { p := p+q; }
        r, q := r/2, q+q;
    }
}
```

```
// Define raising to a power
function IntPow( x: int, y: int ): int
    decreases y
    requires y >= 0
    requires x >= 0
    ensures IntPow(x,y) >= 0
{
    if y == 0 then
        1
    else
        x*IntPow(x,y-1)
}

// Define squaring
function Square( x: int ): int
{
    x*x
}
```

```
// Raising to a power efficiently
function IntPowEfficient( x: int, y: int ): int
    decreases y
    requires x >= 0
    requires y >= 0
    ensures IntPowEfficient(x,y) >= 0
    ensures x > 0 ==> IntPowEfficient(x,y) > 0
{
    if y == 0 then
        1
    else if y%2 == 0 then
        Square(IntPowEfficient(x,y/2))
    else
        x*Square(IntPowEfficient(x,y/2))
}
```

```
// Raising to a power efficiently – alternative version
function IntPowEfficient( x: int, y: int ): int
    decreases y
    requires x >= 0
    requires y >= 0
    ensures IntPowEfficient(x,y) >= 0
    ensures x > 0 ==> IntPowEfficient(x,y) > 0
{
    if y == 0 then
        1
    else if y%2 == 0 then
        IntPowEfficient(Square(x),y/2)
    else
        x*IntPowEfficient(Square(x),y/2)
}
```

```
// This lemma can be proven in Dafny using induction.
// See IntPow.dfy for the proof.

// We need a separate lemma to prove this because
// IntPowEfficient is a function rather than a method
// and we can not put assertions and suchlike inside
// the body of a function, but we can inside a lemma
// or a method.

lemma IntPowEfficientLemma( x: int, y: int )
    requires x >= 0
    requires y >= 0
    ensures IntPowEfficient(x,y) == IntPow(x,y)
    ensures x > 0 ==> IntPow(x,y) > 0
```

```
// x^(a+b) == (x^a)*(x^b)
lemma IntPowMulLemma( x: int, a: int, b: int )
    decreases a
    requires x >= 0 && a >= 0 && b >= 0
    ensures IntPow(x,a+b) == IntPow(x,a)*IntPow(x,b)
{
    if a == 0 { return; }
    IntPowMulLemma(x,a-1,b);
    assert IntPow(x,a+b) == x*IntPow(x,a+b-1);
}
```

```
// (x^a)^b == x^(a*b)
lemma IntPowPowLemma( x: int, a: int, b: int )
    decreases b
    requires x >= 0 && a >= 0 && b >= 0
    ensures IntPow(IntPow(x,a),b) == IntPow(x,a*b)
{
    if b == 0 { return; }
    IntPowPowLemma(x,a,b-1);
    IntPowMulLemma(x,a,a*(b-1));
    calc ==
    {
        IntPow(IntPow(x,a),b);
        IntPow(x,a)*IntPow(IntPow(x,a),b-1);
        IntPow(x,a)*IntPow(x,a*(b-1));
        IntPow(x,a+a*(b-1));
    }
}
```

```
// Integer raising to a power using the Russian peasant method in a loop.
method IntPowLoop( x: int, y: int) returns ( p: int )
    requires x >= 0 && y >= 0
    ensures p == IntPow(x,y)
{
    p := 1;
    var q, r := x, y;
    ghost var qpow, ppow := 1, 0;
    while r != 0
        invariant r >= 0 && qpow >= 1 && ppow >= 0
        decreases r
        invariant q == IntPow(x,qpow) && p == IntPow(x,ppow) && y == ppow+r*qpow
    {
        if r%2 == 0
        {
            IntPowPowLemma(x,2*qpow,r/2); IntPowMulLemma(x,qpow,qpow);
            q, qpow, r := q*q, 2*qpow, r/2;
        }
        else
        {
            IntPowMulLemma(x,ppow,qpow);
            p, ppow, r := p*q, ppow+qpow, r-1;
        }
    }
}
```

```
// Define modular powers
function ModPow( x: int, y: int, m: int ): int
    decreases y
    requires m > 1
    requires x >= 0
    requires y >= 0
    ensures 0 <= ModPow(x,y,m) < m
{
    if y == 0 then
        1
    else
        ModMul(x,ModPow(x,y-1,m),m)
}
```

```
// Define modular multiplication
function ModMul( x: int, y: int, m: int ): int
    requires x >= 0
    requires y >= 0
    requires m > 1
{ (x*y)%m }

//Define modular squaring
function ModSquare( x: int, m: int ): int
    requires m > 1
    requires 0 <= x < m
    ensures ModSquare(x,m) == (x*x)%m
    ensures ModSquare(x,m) == ModPow(x,2,m)
    ensures ModSquare(x,m) == IntPow(x,2)%m
    ensures ModSquare(x,m) == ModMul(x,x,m)
{ (x*x)%m }
```

```
// Fast modular power using recursion. See ModPow.dfy
method ModPowRecursive( x: int, y: int, m: int ) returns ( p: int )
    decreases y
    requires m > 1
    requires 0 <= x < m
    requires y >= 0
    ensures 0 <= p < m
    ensures p == IntPow(x,y)%m
{
    if y == 0 { return 1; }
    if y%2 == 0
    {
        p := ModPowRecursive(x,y/2,m);
        p := ModSquare(p,m);
    }
    else
    {
        p := ModPowRecursive(x,y/2,m);
        p := ModMul(p,ModMul(x,p,m),m);
    }
}
```

```
// Fast modular power using a loop
method ModPowLoop( x: int, y: int, m: int ) returns ( p: int )
    requires m > 1
    requires x >= 0
    requires y >= 0
    ensures 0 <= p < m
    ensures p == ModPow(x,y,m)
{
    p := 1;
    var q, r := x%m, y;
    while r != 0
    {
        // 0 <= r,p < m, x^y == p*q^r mod m
        if r%2 == 0 { r := r/2; q := ModSquare(q,m); }
        else        { r := r-1; p := ModMul(p,q,m);  }
    }
}
```

```
// Dafny loop invariant for ModPowLoop.
// qpow and ppow are ghost variables.
invariant r >= 0
invariant 0 <= q < m
invariant 0 <= p < m
invariant qpow > 0
invariant ppow >= 0
invariant q == ModPow(x,qpow,m)
invariant p == ModPow(x,ppow,m)
invariant y == ppow+qpow*r
```

```
// The following fundamental lemma can be proven in Dafny
// using an indirect proof (reductio ad absurdum, proof by
// contradiction). See ModPow.dfy and next slide.

// If x == q*m+r and 0 <= r < m then q == x/m and r == x%m.
lemma ModBasicLemma( x: int, m: int, q: int, r: int )
    requires x == q*m+r
    requires 0 <= r < m
    ensures q == x/m
    ensures r == x%m
```

```
// If x == q*m+r and 0 <= r < m then q == x/m and r == x%m.
var q2 := x/m;
var r2 := x%m;
if( q==q2 ) { return; }
// Given that different quotients are possible,
// we will now derive a contradiction.
assert m*(q-q2)+(r-r2) == 0;
assert -m < r-r2 < m;
if( q > q2 )
{
    assert q-q2 >= 1;
    assert m*(q-q2)+(r-r2) != 0;
}
else
{
    assert q-q2 <= -1;
    assert m*(q-q2)+(r-r2) != 0;
}
```

This version was accepted by previous versions of Dafny. The current Dafny is not able to verify this, but ModPow.dfy contains a modified proof that Dafny accepts.