RSA handout

Here, we will implement RSA using Tcl. Your assignment is to read this document through, and to type in the code examples in the gray boxes. If you have any difficulties, contact the instructor. To learn more, you can consult the Tcl command manual pages at http://www.tcl.tk/doc/, or the excellent and comprehensive Tcl Wikibook at http://en.wikibooks.org/wiki/Tcl_Programming/Print_version.

Getting started

This requires Tcl 8.5. Get to a tclsh prompt and type the following text:

```
your-prompt$ tclsh
% info tclversion
8.5
% expr 2**100
1267650600228229401496703205376        ←  (result from 8.5)
syntax error in expression "2**100"... ←  (result from 8.4)
```

If the version is 8.5 or above, Tcl can natively perform arbitrary precision integer arithmetic. If you're stuck with 8.4, there's a workaround, but it's not fun.

We will also use the Unix bc command, which performs arbitrary precision arithmetic. It has many useful features, including easy base conversion. Try from the Unix prompt:

```
your-prompt$ bc
bc 1.06 [stuff about free software foundation]

2^100
1267650600228229401496703205376
obase=2;2^100
10000000000000000000000000000000000000000000000000000000000000000000\
00000000000000000000000000000000
[Ctrl-D to Quit]
```

We can easily write a Tcl command to invoke `bc` whenever we need it, using the `exec` command. There are two complications: first, Tcl uses `**` for exponentiation, and `bc` uses `^`. Second, `bc` outputs long numbers by breaking them into lines with backslashes, and they need to be de-slashed and de-spaced. Try this:

```
proc tobc e {
        set e [string map {** ^} $e]
        join [subst [exec echo $e | bc]] ""
}
```

The first line swaps Tcl notation for bc notation; the second line removes the backslashes (`subst`) and spaces (`join`) to give you a single string. Try executing the command `tobc "obase=2;2**100"` to verify. Note the quotes: if you don't have the quotes, Tcl will be confused by the semicolon in the middle of a command.

The `bc` command is incredibly useful, but it has the disadvantage that Tcl must perform a system call to a Unix command, pass in input, and collect output each time it is called. That's much slower than the native arithmetic of Tcl 8.5. We'll only use it for base conversion, which we'll need for modular exponentiation, and for primality testing:

```
% proc tobinary  d { tobc "obase=2;$d" }
% proc todecimal b { tobc "ibase=2;$b" }
% tobinary 42
101010
% todecimal 100000000
256
```

RSA

The RSA algorithm has a very simple setup and encryption algorithm. Here's the setup:

```
1  Choose large primes p, q
2  Compute n=pq, $\varphi(n)=(p-1)(q-1)$
3  Compute exponents e, $d=e^{-1}$ (mod $\varphi$)
4  Publish <n,e> (public key), save <n,d> (private key)
```

The encryption and decryption rules are just

```
Encrypt:  compute $c=m^e$ (mod n)
Decrypt:  compute $m=c^d$ (mod n)
```

Tcl/bc is already capable of multiplying and division with remainder, and can perform exponentiation within reasonable limits. We then need fast modular exponentiation with very large exponents and moduli; computation of inverses mod n; and a primality test for generating the public key.

Let’s start with exponentiation. We will use the square-and-multiply algorithm, which requires that we translate our exponent into binary.

```
proc modexp {m e n} {
     set P 1
     foreach bit [split [tobinary $e] ""] {
          if $bit {
             set P [expr {($P*$P*$m)%$n}]
          } else {
             set P [expr {($P*$P)%$n}]
          }
     }
     set P
}
```

And test:

```
% modexp 37 12341345236234563767801 1000
37
```

Next, let’s try our hand at primality testing. Here is the Miller Rabin algorithm. To review: if n is prime, then we can write $\phi(n)=n-1=d2^k$, where d is odd. We choose a random value a and gradually compute a^{n-1} by computing a^d and squaring it k times. We know n is not prime if we observe one of two oddities: (1) something that isn’t -1 squares to 1, or (2) the final total is not congruent to 1. We quit early if a^d is 1 to start with, or if we see a -1---in those cases, neither oddity could occur in this trial.

```
proc millerRabinTrial {n d k} {
     set minusOne [expr {$n-1}]    ;# we need to compare to -1
     set a [randomlessthan $n]     ;# choose a random a
     set M [modexp $a $d $n]       ;# compute ad
     if {$M == 1} {return 1}     ;# quit if it’s already 1

     for {set i 0} {$i<$k} {incr i} {
         if {$M eq $minusOne} {return 1}
         set M [expr {($M*$M)%$n}]
         if {$M eq 1} {return 0}              ;# oddity (1)
     }
     if {$M == 1} {return 1} else {return 0}  ;# oddity (2)
}  -- that last line isn’t necessary!
```

This code requires us to write another function, `randomlessthan`, which generates a "random" value a. We'll break some major rules here and use the (insecure) built-in random generator to generate lousy (nonuniform) pseudo-random numbers. See exercise 1 for an explanation of why we shouldn't do this in practice:

```
proc randomlessthan n {
     set a 1
     foreach d [split $n ""] {append a [expr int(rand()*10)]}
     set result [expr {($a%$n)}]
     if {$result==0} {return 1} else {return $result}
}
```

Now to wrap this in the full test. We want to perform this test at most MAXTEST times, let's say 20. There are some tricks in this code we'll discuss in the exercises:

```
set MAXTEST 20
proc millerRabin n {
     regexp {^(.*1)(0*)$} [tobinary ($n-1)] -> first last
     set d [todecimal $first]
     set k [string length $last]

     for {set i 0} {$i<$::MAXTEST} {incr i} {
         if {[millerRabinTrial $n $d $k]==0} {return 0}
     }
     return 1       ;# probably a prime number
}
```

Finally, for prime generation:

```
proc randomnumber bits {
     for {set i 0} {$i<$bits-2} {incr i} {
         append s [expr int(rand()*2)]
     }
     todecimal 1${s}1   ;# extra 1s to make odd n-bit number
}

proc makeprime bits {
     while {![millerRabin [set p [randomnumber $bits]]]} {}
     set p
}
```

Back to RSA: let's do our key setup and everything:

```
proc RSAkey bits {
     set p [makeprime $bits]
     set q [makeprime $bits]
     set n [expr $p*$q]
     set phi [expr ($p-1)*($q-1)]

     for {set e 3} {[gcd $e $phi]!=1} {incr e} {}
     set d [invert $e $phi]

     list $n $e $d   ;# modulus, public, private exponent
}
```

Whoa, whoa whoa: we need to be able to invert a number, and compute a GCD. Let's implement the General Euclidean algorithm:

```
proc Euclid {a n} {
     set equations [list 1 0 $n \
                         0 1 $a]  ;# start of algorithm
     set r -1
     while {$r} {
         lassign $equations a b x c d y
         set q [expr {$x/$y}]
         set e [expr {$a-$q*$c}]
         set f [expr {$b-$q*$d}]
         set r [expr {$x-$y*$q}]
         set equations [list $c $d $y $e $f $r]
     }
     lrange $equations 0 2    ;# return A B GCD
}                             ;# with An + Ba = GCD
```

This gives us both the GCD of two numbers, and the inverse of one mod the other:

```
proc gcd {a n} {lindex [Euclid $a $n] end}
proc invert {a n} {
     set b [lindex [Euclid $a $n] 1]  ;# middle element
     expr {$b%$n}
}
```

Great. Finally, encryption, which doesn't really require a separate procedure:

```
proc encrypt {m e n} {modexp $m $e $n}
proc decrypt {c d n} {modexp $c $d $n}
```

Test:

```
% lassign [RSAkey 20] n e d
% set c [encrypt 31337 $e $n]
528770031468
% set m [decrypt $c $d $n]
31337
%
```

It works! Well, with 20 bits. Let's find out how horrible this is with 1000-bit primes:

```
% time {lassign [RSAkey 1000] n e d}
25788920 microseconds per iteration
% time {set c [encrypt 31337 $e $n]}
2798 microseconds per iteration
% time {set m [decrypt $c $d $n]}
146510 microseconds per iteration
% set m
31337
```