LAB 7

PowerMod and RSA

We have already made a function that computes exponents, mod m, extremely quickly. To implement RSA, we will also need multiplicative inverses.

PROBLEM 7.1. Write a recursive function `MyGCD1` that computes $\gcd(a, b)$ for any pair of integers a, b.

To implement the Extended Euclidean Algorithm, we will need to keep track of the quotients and remainders involved in computing $\gcd(a, b)$: the current a, b, q, and r.

PROBLEM 7.2. Write a while loop that creates a table of all the numbers for the GCD. It should look something like this (remember to copy-paste). Note: you can visualize a table more easily by adding `//TableForm`.

```
EuclideanTable[a_,b_]:=Module[{ETable={},i=1},
      AppendTo[ETable, (*starting row*)];
      While[(*remainder in the last row is non-zero*),
          (*add a new row to ETable*);
          i++;
      ];
      EETable
 ]
```

Problem 7.3. Make a function `MyGCD2` that passes its parameters to `EuclideanTable`, and then extracts the relevant cell from the table.

Hint: Tables are just lists of lists. To get the third element of a list `T`, use the command `T[[3]]`. To get the last one, use `T[[-1]]`.

Now let's extend the above function:

Problem 7.4. Write a `ExtendedEuclideanTable[a_,b_]` that creates the table we generate with the Extended Euclidean Algorithm.

It should look roughly like this:

```
ExtendedEuclideanTable[a_,b_]:=Module[{EETable={},i=1},
     AppendTo[EETable, (*starting row, with blank s and t*)];
     While[(*remainder in the last row is non-zero*),
         (*add a new row to EETable*);
         i++;
     ];
     (*In the second half, we fill in s and t values:*)
     (*Fill in the s and t values in the bottom row*)
     While[(*we have not returned to the top*),
         i--;
         (*fill in the i-th row of s and t values*);
     ];
     EETable
]
```

Problem 7.5. By accessing the correct cells inside the table produced by `ExtendedEuclideanTable`, write functions `MyGCD3[a_,b_]`, `MyBezout[a_,b_]` and `MyMultiplicativeInverse[a_,m_]`.

We have now built all the pieces of `PowerMod`: exponents and inverses.

Problem 7.6. What does `PowerMod[a_,b_,m_]` do? Try different values of a, b, m, including negative ones.

Problem 7.7. Make a `MakeRSAKeys[]` function that randomly chooses $\{e, d, n\}$. It should look something like:

```
MakeRSAKeys[a_,b_]:=Module[{p,q,e,n,m,d},,
     p=RandomPrime[10^9,10^10];
     (*Generate q and e, and compute n, m, and d*)
     {n,e,d}
]
```

Make sure you use `MyPowerMod` to compute d. Use `MakeRSAKeys[]` to generate RSA public and private keys, and post your public key on the whiteboard.

Problem 7.8. Use `MyPowerMod` to create the functions

```
RSAEncode[text_,e_,n_]:=MyPowerMod[
    FromDigits[
        ToCharacterCodes[text]-97,
        26
    ],e,n];


RSADecode[y_,d_,n_]:=
    FromCharacterCodes[
        FromIntegerDigits[
            MyPowerMod[y,d,n],
            26
        ]+97;
    ]
```

Use these to send messages to other teams, or sign your announcements. This time, we can't (easily) crack your encryption, so make sure you're writing the correct numbers on the board.