Intro to Discrete Math & Functional Programming— CSCI 54— Fall 2023
Instructor: Chen
Homework - Week 14 (28 points)
Due: 10:00PM on Sunday, December 3

- This is another two part assignment. Both parts are due at the same time but need to be uploaded to separate Gradescope assignments (week14-ps-written and week14-ps-coding).
- For each of the two parts you can choose to work either individually or with a partner of your choice from the class.
- As before, you are welcome to discuss the problems with your classmates, the TAs, and myself. You must write up your solutions/code on your own (or with your partner). You should never be looking at someone else's solution/proof/code (not even written up on a whiteboard!) nor should you show anyone else in the class your solution/proof/code. *Please* let me know if you have any questions about the collaboration policy!

Part 1: On modular arithmetic

This part of the problem set should be typeset in LaTeX and the pdf should be uploaded to the gradescope assignment `week14-ps-written`.

1. **[6 points] Working with the** mod **operator**

   Use a proof by cases to show that the following is true: $\forall x \in \mathbb{Z}^+ : (x^3 + 1) \bmod 3 = (x + 1)^3 \bmod 3$.

   case 1:

   consider $(x^3 + 1)$

   let k be an integer;

   $x = 3k$

   $((3k)^3 + 1) = 9k^3 + 1$

   $(3(3k^3) + 1) \bmod 3 = 1$

   Also,

   consider $(x + 1)^3$

   let k be an integer;

$x = 3k$

$(3k+1)^3 = 27k^3 + 27k^2 + 9k + 1$

$(3(9k^3 + 9k^2 + 3) + 1) \bmod 3 = 1$

case 2:

consider $(x^3 + 1)$

let k be an integer;

$x = 3k + 1$

$((3k+1)^3 + 1) = 27k^3 + 27k^2 + 9k + 2$

$3(9k^3 + 9k^2 + 3k) + 1 \bmod 3 = 2$

Also,

consider $(x+1)^3$

let k be an integer;

$x = 3k + 1$

$(3k+2)^3 = 27k^3 + 54k^2 + 36k + 8$

$(3(9k^3 + 18k^2 + 12k) + 8) \bmod 3 = 2$

case 3:

consider $(x^3 + 1)$

let k be an integer;

$x = 3k + 2$

$(3k+2)^3 + 1 = 27k^3 + 54k^2 + 36k + 9$

$3(9k^3 + 18k^2 + 12k + 3) \bmod 3 = 0$

Also,

Consider $(x+1)^3$

let k be an integer;

$x = 3k + 2$

$((3k+2)+1)^3 = 21k^3 + 81k^2 + 81k + 27$

$3(7k^3 + 27k^2 + 27k + 9) \bmod 3 = 0$

In all the three cases, $(x^3 + 1) \bmod 3 = (x+1)^3 \bmod 3$

Therefore we have proven that

$\forall x \in \mathbb{Z}^+ : (x^3 + 1) \bmod 3 = (x + 1)^3 \bmod 3$

2. **[4 points] Working with RSA**

   In class we sketched out the proof of why `decrypt(encrypt(m)) = m`. There were two restrictions in that proof: we needed $m < n$. And we needed $gcd(m, n) = 1$.

   (a) How many messages does this rule out? In other words, given $n = pq$, what is $|\{m \in \mathbb{Z}^+ \mid m < n \text{ and } gcd(m, n) \neq 1\}|$? You should be able to provide an exact formula. Briefly justify.

   To find the number of integers that satisfy $|\{m \in \mathbb{Z}^+ \mid m < n \text{ and } gcd(m, n) = 1\}|$, we find $\phi(n)$ which can be found by doing :

   $n = pq$

   let k_1 and k_2 be integers

   Any multiples of n other than p and q will have a gcd greater than 1.

   $k_1 < p$ and $k_2 < q$

   $p - 1$ counts the multiples of p that are less than one

   $q - 1$ counts multiples of q less than one

   1 being a multiple of all numbers and does not provide a meaningful encryption is therefore not considered

   the formula for the total number of messages that are ruled out is therefore: $p + q - 2$

   $|\{m \in \mathbb{Z}^+ \mid m < n \text{ and } gcd(m, n) = 1\}| = p + q - 2$

   (b) What percentage of the total number of possible messages are ruled out? Complete the following table:

p	q	n	# messages ruled out	% messages ruled out
3	7	21	8	40
181	431	78010	611	0.7832229
1543	2729	4210846	4272	0.1014523
11497	23143	266075070	34639	1.3018506442631046e-2

Part 2: Implementing RSA encryption/decryption

This part of the problem set should be done in Haskell and you should upload your `.hs` file to the gradescope assignment `week14-ps-coding`. The `week14-ps-template.hs` file on piazza has the signatures for each function. Make sure the version you turn in is commented (e.g. each function should have a comment describing what it does).

1. **[2 points] squareMod**

   Write a function called `squareMod` that takes two `Integers` b and n and returns $b^2 \bmod n$. You may assume n is non-negative.

2. **[2 points] powerMod**

   Write a function called `powerMod` that takes three `Integers` b, e, and n and that returns $b^e \bmod n$. Your function must compute $b^e \bmod n$ using the following recursive function:

$$b^e = \begin{cases} 1 & \text{if } e = 0, \\ (b^{e/2})^2 & \text{if } e \text{ is even, and} \\ (b^{e/2})^2 \cdot b & \text{otherwise.} \end{cases}$$

   The expression $e/2$ in the recursive step signifies integer division, with truncation.

   The order of the arguments is `b`, `e`, and then `n`. You may assume that `b` and `n` are positive, and `e` is non-negative. The function `squareMod` may be helpful. Be sure to take the remainder after *every* arithmetic operation to keep the size of the intermediate results under control.

   A comment on the algorithm: You will use `powerMod` again later in the assignment. It is very important that you follow the formula above. The more obvious strategy—of multiplying b by itself e times—could take centuries. The reason that the "repeated squaring" strategy is better is that the recursive step has the exponent $e/2$ instead of $e - 1$. The exponent e gets one bit shorter with each recursive call, so the number of recursive calls is the number of bits required to express e in binary. For example, if e is one million, about 2^{20}, the "repeated multiplying" strategy will do about a million multiplications, compared to at most 40 for the "repeated squaring" strategy.

3. **[2 points]** `block`

   To be able to translate arbitrarily long messages we need to be able to break a number that is larger than n into a collection of numbers that are all smaller than n (and then the reverse operation), all *in a way that is efficient in space usage.*

   Implement a function `block` that takes two integers n and m and that evaluates to a list of non-negative integers, each of which is less than n. If the returned list is $[x0, x1, \ldots, xk]$, then $m = x0 + x1 \cdot n + x2 \cdot n^2 + \ldots + xk \cdot n^k$. You can think of `block n m` as evaluating to the base$-n$ representation of the integer m. You may assume that both n and m are greater than 1.

   For example:

```
ghci> block 2 26
[0,1,0,1,1]
```

```
ghci> block 111 12345678910
[58,110,5,36,81]
```

4. **[2 points]** `unblock`

   Implement the inverse of `block`. In other words, `unblock n (block n m)` should evaluate to `m`. Again, you may assume that both parameters are greater than 1.

5. **[2 points]** `messageToInteger`

   RSA encrypts numbers. To encrypt a string, we need to be able to convert from any string into some corresponding number. One way to do this is to treat each character in a string as a digit. Characters correspond to integers between 0 and 255.

   Recall that the function `ord` maps a character to the corresponding integer. Its inverse is the function `chr`. This means a string can be represented as a list of values between 0 and 255. Once we have a list of numbers (representing the "digits") we can turn it into an `Integer` represented in base 256.

   Write a function `messageToInteger` that takes a string and converts it to an Integer. (Hint: think about how you can use `block` and/or `unblock`)

   As an example:

   ```
   ghci> messageToInteger "abc"
   6513249
   ```

   because $97 + 98 \cdot 256 + 99 \cdot 256^2 = 6513249$ (and `ord 'a' = 97`).

6. **[2 points]** `integerToMessage`

   Implement the inverse of `messageToInteger`. In other words, `integerToMessage (messageToInteger str)` should evaluate to `str`.

7. **[2 points]** `rsaEncode`

   Write a function `rsaEncode (e,n) m` that returns the message m encrypted using the public key (e, n). You may assume that m is non-negative and less than n. The function takes three `Integers` and evaluates to a fourth `Integer` which equals $m^e \bmod n$.

8. **[2 points]** `encodeString`

   We now have all of the pieces to support encryption and decryption, given a set of keys. Remember, to encrypt a string:

   - We turn the string into a number,
   - then break this number into chunks of size n.

- then encrypt each of these chunks using the public key.
- And finally, put it all back together into a single number.

Decryption is just these steps in reverse using the private key.

Write a function `encodeString` that takes a key (e, n) and a string, and produces a single `Integer` value that encrypts the message contained in the string. It should make sense to write this function as a combination of functions that you have already written.

```
ghci> encodeString (7,111) "CS54 is my favorite class!"
397205335758531275142249411863858662823428826090037031845358616
```

9. **[2 points]** `decodeString`

Write an analogous function `decodeString` that decrypts a message encoded using `encodeString`. Of course to decrypt the message you'll need to use the private key that corresponds to the public key.

Assuming everything works, you should be able to encode and then decode a message with a pair of keys (e.g., (7, 111) and (31, 111)):

```
ghci> decodeString (31,111) (encodeString (7,111) "CS54 is my favorite class!")
"CS54 is my favorite class!"
```