CSC110 Fall 2022 Assignment 4: Number Theory, Cryptography, and Algorithm Running Time Analysis

Yehyun Lee

November 23, 2022

Part 1: Proofs

1. Statement to prove: $\forall a, b, n \in \mathbb{Z},\ \big(n \neq 0 \wedge a \equiv b\ (\text{mod } n)\big) \Rightarrow \big(\forall m \in \mathbb{Z},\ a \equiv b + mn\ (\text{mod } n)\big)$

   Proof. (I will first show how I expanded the statement and then prove the statement.)

   Let a, b, n $\in \mathbb{Z}$. Assume $\big(n \neq 0 \wedge a \equiv b\ (\text{mod } n)\big)$ is true.

   By the definition of modular equivalence in course note 7.4,
   we say that "a is equivalent to b modulo n" or a $\equiv b\ (\text{mod } n)$, when n | a - b.
   So we know that n | a - b is true.
   And by using the definition of divisibility, we know that $\exists k_1 \in \mathbb{Z}$, a - b = k_1n is true.

   We want to show that $\forall m \in \mathbb{Z},\ a \equiv b + mn\ (\text{mod } n)$.

   By the definition of modular equivalence, we can express this as n | a - (b + mn).
   And by using the definition of divisibility, we know this is $\exists k_2 \in \mathbb{Z}$, a - b - mn = k_2n.
   So $\forall m \in \mathbb{Z},\ \exists k_2 \in \mathbb{Z}$, a - b - mn = k_2n is what we need to prove.

   When we fully expand the statement:
   $\forall a, b, n \in \mathbb{Z},\ \big(n \neq 0 \wedge \exists k_1 \in \mathbb{Z}, a - b = k_1 n\big) \Rightarrow \big(\forall m \in \mathbb{Z},\ \exists k_2 \in \mathbb{Z}$, a - b - mn = k_2n$\big)$

   Now, let's finish proof:
   Let a, b, n $\in \mathbb{Z}$
   By the assumption, we know $\big(n \neq 0 \wedge \exists k_1 \in \mathbb{Z}, a - b = k_1 n\big)$ is true.
   We want to prove $\forall m \in \mathbb{Z}$, $\exists k_2 \in \mathbb{Z}$, a - b - mn = k_2n.

   Let m $\in \mathbb{Z}$
   Take $k_2 = k_1 - m$.

   Take out k_2 and substitute $k_1 - m$ into a - b - mn = k_2n.
   a - b - mn = $(k_1 - m)$n, thus, a - b - mn = $k_1 n - mn$.

   Since we know that $\exists k_1 \in \mathbb{Z}$, a - b = k_1n,
   take out a - b and substitute k_1n into a - b - mn = $k_1 n - mn$.
   $(k_1 n) - mn = k_1 n - mn$.

   Since we get both sides equal, $\forall m \in \mathbb{Z}$, $\exists k_2 \in \mathbb{Z}$, a - b - mn = k_2n holds true. Thus, the statement: $\forall a, b, n \in \mathbb{Z},\ \big(n \neq 0 \wedge a \equiv b\ (\text{mod } n)\big) \Rightarrow \big(\forall m \in \mathbb{Z},\ a \equiv b + mn\ (\text{mod } n)\big)$ is true. □

2. Statement to prove: $\forall f, g : \mathbb{N} \to \mathbb{R}^{\geq 0},\ \Big(g \in \mathcal{O}(f) \wedge \big(\forall m \in \mathbb{N},\ f(m) \geq 1\big)\Big) \Rightarrow g \in \mathcal{O}(\lfloor f \rfloor)$

Proof. First, let's expand the statement using the definition of Big-O (From course note 9.3):
$\forall f, g : \mathbb{N} \to \mathbb{R}^{\geq 0},\ \Big((\exists c_1, n_1 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_1 \Rightarrow g(n) \leq c_1 \cdot f(n)) \wedge \big(\forall m \in \mathbb{N},\ f(m) \geq 1\big)\Big) \Rightarrow$
$(\exists c_2, n_2 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_2 \Rightarrow g(n) \leq c_2 \cdot \lfloor f(n) \rfloor)$.

Let f, g: $\mathbb{N} \to \mathbb{R}^{\geq 0}$.
Assume $(\exists c_1, n_1 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_1 \Rightarrow g(n) \leq c_1 \cdot f(n)) \wedge \big(\forall m \in \mathbb{N},\ f(m) \geq 1)$ is true.

We want to prove $\exists c_2, n_2 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_2 \Rightarrow g(n) \leq c_2 \cdot \lfloor f(n) \rfloor$.
Take $c_2 = 2 \cdot c_1$
Take $n_2 = n_1$
Let $n \in \mathbb{N}$
Since we assumed $n \geq n_1$, this implies $n \geq n_2$, so assume n $\geq n_2$.
We want to show that $g(n) \leq c_2 \cdot \lfloor f(n) \rfloor$.

Since we know that (according to instruction) for all $m \in \mathbb{N}, \lfloor f(m) \rfloor \geq 1$ and for all x $\in \mathbb{R}, \lfloor f(n) \rfloor \leq f(n) < \lfloor f(n) \rfloor + 1$, we also know that $\lfloor f(n) \rfloor \leq f(n) < \lfloor f(n) \rfloor + 1 \leq 2 \cdot \lfloor f(n) \rfloor$.

And since $c_1 \cdot \lfloor f(n) \rfloor \leq c_1 \cdot f(n) < c_1 \cdot \lfloor f(n) \rfloor + c_1 \leq c_1 \cdot 2 \cdot \lfloor f(n) \rfloor$, if we take $c_2 = 2 \cdot c_1$, it makes g(n) $\leq c_1 \cdot f(n) < c_2 \cdot \lfloor f(n) \rfloor$ possible because we know $c_2 \cdot \lfloor f(n) \rfloor$ is greater than $c_1 \cdot f(n)$, and since g(n) $\leq c_1 \cdot f(n)$, we know this makes $g(n) < c_2 \cdot \lfloor f(n) \rfloor$, and also we know this makes $g(n) \leq c_2 \cdot \lfloor f(n) \rfloor$ true.

Therefore, $\exists c_2, n_2 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_2 \Rightarrow g(n) \leq c_2 \cdot \lfloor f(n) \rfloor$ is true.

Thus, the statement is true. □

Part 2: Running-Time Analysis

1. Function to analyse:

```
def f1(n: int) -> int:
    """Precondition: n >= 0"""
    total = 0

    for i in range(0, n):  # Loop 1
        total += i ** 2

    for j in range(0, total):  # Loop 2
        print(j)

    return total
```

$RT_{f1}(n) = 1 + n + \frac{2n^3-3n^2+n}{6} + 1 = n + \frac{2n^3-3n^2+n}{6} + 2$, which is $\Theta(n^3)$.

"total = 0" takes 1 step, since it's constant time. Loop 1 takes n times because it takes n iterations and each iteration takes 1 step, thus n * 1 = n. We first want to find what total is to determine the Loop 2. Equation, $\frac{2n^3-3n^2+n}{6}$, is derived from using given equation "sum of consecutive squares" from instruction: $\sum_{i=0}^{n} i = \frac{n(n+1)(2n+1)}{6}$, from here I substitute n - 1 (since this is Python, index starts at 0 and does not include n for "range(0, n)"). Thus:
$\sum_{i=0}^{n-1} i = \frac{(n-1)(n)(2(n-1)+1)}{6} = \frac{2n^3-3n^2+n}{6}$. We now know how many times Loop 2 iterates! print(j) is 1 constant time, so multiplying that to equation, we get $\frac{2n^3-3n^2+n}{6}$ as a step. Next, since "return total" takes 1 step (constant time), that's 1. Theta in this case is n^3 since it dominates the rest.

Thus, $RT_{f1}(n) = n + \frac{2n^3-3n^2+n}{6} + 2$, which is $\Theta(n^3)$.

2. Function to analyse:

```
def f2(n: int) -> int:
    """Precondition: n >= 0"""
    sum_so_far = 0

    for i in range(0, n):  # Loop 1
        sum_so_far += i

        if sum_so_far >= n:
            return sum_so_far

    return 0
```

$RT_{f2}(n) = 1 + \lceil \frac{1+\sqrt{1+8n}}{2} \rceil + 1 = 2 + \lceil \frac{1+\sqrt{1+8n}}{2} \rceil$, which is $\Theta(\sqrt{n})$.

"sum_so_far = 0" takes 1 step (constant time), thus 1.
Next, let's determine Loop 1. This is very similar to what we did in lecture. Except, instead of while loop, this is if statement. Thus, when $i_k \geq n$, it will stop function by return statement. i_k in this case is sum_so_far. How can we find sum_so_far? Well since for loop iterates n times and sum_so_far += 1 every time it iterates, we can use "sum of consecutive numbers", i.e., $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$. Let's use this to find equation of i_k. Substituting k-1, we get $i_k = \frac{k^2-k}{2}$. Since $i_k \geq n$, so, $\frac{k^2-k}{2} \geq$ n. Now, we can move n to left side and apply quadratic formula to solve for k! (Since we're looking for smallest value of k, I had to add ceil into equation.) Thus, I get $\lceil \frac{1+\sqrt{1+8n}}{2} \rceil$. Now, we know how many times Loop 1 iterates! sum_so_far += i is 1 constant time, so multiplying that to how many times Loop 1 iterates, we get $\lceil \frac{1+\sqrt{1+8n}}{2} \rceil$ as step! Lastly, I have additional 1, because we will either return 0 or sum_so_far (inside Loop 1, if condition is satisfied). We cannot return both(As we return, function will stop). Thus, that takes another 1 step! Now, we have $RT_{f2}(n) = 1 + \lceil \frac{1+\sqrt{1+8n}}{2} \rceil + 1 = 2 + \lceil \frac{1+\sqrt{1+8n}}{2} \rceil$, which is $\Theta(\sqrt{n})$.

Part 3: Extending RSA

Complete this part in the provided `a4_part3.py` starter file. Do **not** include your solutions in this file.

Part 4: Digital Signatures

Part (a): Introduction

Complete this part in the provided `a4_part4.py` starter file. Do **not** include your solutions in this file.

Part (b): Generalizing the message digests

Complete most of this part in the provided `a4_part4.py` starter file. Do **not** include your solutions in this file, *except* for the following two questions:

3b.
```
def find_collision_len_times_sum(message: str) -> str:
    """Return a new message, not equal to the given message,
    that can be verified using the same signature
    when using the RSA digital signature scheme with the len_times_sum message digest.

    Preconditions:
    - len(message) >= 2
    - any({ord(c) < 1114111 for c in message})
    """
    change_to_ord = [ord(c) for c in message]
    if sum(change_to_ord) == 0:
        return message + chr(0)
    smallest, biggest = min(change_to_ord), max(change_to_ord)
    smallest_position, biggest_position = 0, 0        # To avoid PythonTA
    for find_smallest_position in range(0, len(change_to_ord)):
        if smallest == change_to_ord[find_smallest_position]:
            smallest_position = find_smallest_position
    change_to_ord[smallest_position] += 1
    for find_biggest_position in range(0, len(change_to_ord)):
        if biggest == change_to_ord[find_biggest_position]:
            biggest_position = find_biggest_position
    change_to_ord[biggest_position] -= 1
    final = ''.join([chr(o) for o in change_to_ord])
    if final == message:
        list.reverse(change_to_ord)
        return ''.join([chr(o) for o in change_to_ord])
    return final
```

First we know that—as Professor Liu stated in instruction—1114111 is the maximum possible ord value for Python character, and also—according to lecture note—we know that smallest ord value for Python is 0. Any negative ord value or ord value greater than 1114111 does not exist in Python.

Since the precondition is

```
any({ord(c) < 1114111 for c in message})
```

meaning every each character's ord value is LESS than 1114111, this makes sure that it is always possible to add 1 to the ord value of at least one character in the message to obtain a valid character.

Now, the goal is to come up with different message that result same value when we compute len_times_sum function. Since changing the length of the message can make big changes to the value, we should focus on maintaining sum of all ord values. We should have same length, while keeping sum of all ord values same. We can replace position of character(possibly can cause error), however, better and simplified way is to add 1 and subtract 1 to each biggest and smallest ord values. This will result different message while keeping the same length, and same sum of all ord values, in which, will result same value of len_times_sum function when computed.

Now, using this code,

```
change_to_ord = [ord(c) for c in message]
```

I converted all the characters in message to ord values. By using max() and min(), I could find max value and min value. From here, we can add 1 to min value and subtract 1 to the max value.

```
    for find_smallest_position in range(0, len(change_to_ord)):
        if smallest == change_to_ord[find_smallest_position]:
            smallest_position = find_smallest_position
    change_to_ord[smallest_position] += 1
    for find_biggest_position in range(0, len(change_to_ord)):
        if biggest == change_to_ord[find_biggest_position]:
            biggest_position = find_biggest_position
    change_to_ord[biggest_position] -= 1
```

This code does what I've said. First, it look for position of where min is located in change_to_ord and add 1. Then, it look for position of where max is located in change_to_ord and subtract 1. For example, if change_to_ord is [0, 1], we make it such that [1, 0]. We add 1 to 0, and subtract 1 to 1.

Since we add 1 to smallest ord value first, then find position of max value and subtract 1, even if we have same character, function works fine. For example, if message is '$\backslash x01\backslash x01$', where change_to_ord is [1, 1], we will first add 1, to make [1, 2], then subtract 1 to get [0, 2], which is '$\backslash x00\backslash x02$'. Now all the length is same, as well as, sum of all ord values are same while having different message!

The problem with this is when we have both min and max to be 0, this is when all the integers in change_to_ord is all 0. This is serious problem since we cannot subtract 1 to 0 because as I stated early in above, smallest ord value is 0, we cannot have negative. Thus, I implemented code for this special case(at the very top above for loop):

```
    if sum(change_to_ord) == 0:
        return message + chr(0)
```

Since this is all ord value of 0, when len_times_sum is computed it will still return 0 (Since for any length multiplied by sum of zero is just 0). While we won't have same length, it will still return same len_times_sum!

Other problem we have is if message is 'bbba', since it has ord values of [98, 98, 98, 97]. Once we add 1 to min, it will be [98, 98, 98, 98], then since max is 98, due to characteristic of my code, it will subtract 1 to 98 at last index, making [98, 98, 98, 97], which is same message. Thus, I implemented additional code:

```
final = ''.join([chr(o) for o in change_to_ord])
if final == message:
    list.reverse(change_to_ord)
    return ''.join([chr(o) for o in change_to_ord])
return final
```

What this does is, if our final message is same as message, then we will just reverse order of the message and return it. If final message is not same as message, we will just return our final message without reversing the message! For example, for 'bbba', it will now return 'abbb', since it reversed the order of change_to_ord. Now, this function handles all possible cases(when precondition is satisfied)!

4b.
```
def find_collision_ascii_to_int(public_key: tuple[int, int], message: str) -> str:
    """Return a new message, distinct from the given message, that can be
    verified using the same signature, when using the RSA digital signature
    scheme with the ascii_to_int message digest and the given public_key.

    The returned message must contain only ASCII characters, and cannot
    contain any leading chr(0) characters.

    Preconditions:
    - signature was generated from message using the algorithm in rsa_sign and
    digest ascii_to_int, with a valid RSA private key
    - len(message) >= 2
    - ord(message[0]) > 0
    """
    n = public_key[0]
    a = ascii_to_int(message)
    ascii_we_need = a + n
    base_we_get = a4_part3.int_to_base128(ascii_we_need)
    return ''.join([chr(ord_value) for ord_value in base_we_get])
```

Question is how can we find message that is different to given message, but generate the same digest modulo n. This digest modulo n appear in rsa_sign (also in rsa_verify).

`compute_digest(message) % n`

I will assume this to be a % n where 'a' represent compute_digest, or in this case, "ascii_to_" function. I can also express this as (a - b) % n, where 'b' is set to 0.

Now, in part 1 question 1, we've proven that $\forall a, b, n \in \mathbb{Z},\ \big(n \neq 0 \wedge a \equiv b\ (\text{mod}\ n)\big) \Rightarrow \big(\forall m \in \mathbb{Z},\ a \equiv b + mn\ (\text{mod}\ n)\big)$.
(Important condition for using this proved definiton is that we cannot have n as 0. However, for my code, since n is p times q, and p, q are prime numbers, they're not 0. Thus, we can use this definition.) Thus, if I express this as Python expression: $\forall a, b, n \in \mathbb{Z},\ \big(n \neq 0 \wedge (a - b)\%n \Rightarrow \big(\forall m \in \mathbb{Z},\ (a - (b + mn))\%n\big)$. From this expression, since we know (a - b) % n is true in our case if b is 0, we can also conclude that we can use (a - (b + mn)) % n, where b is 0. From here, if I make m = -1, the equation becomes (a + n)%n since b is 0. (This m can by any number like 1 and change the equation! However, we cannot have m as 0, if so, we will get same message. I took m to be -1, because it simplifies the equation!)

Since when we're digesting, we're using ascii_to_int, compute_digest(message) % n is basically ascii_to_int(message) % n, we know that this ascii_to_int function takes message and uses base128_to_int to return one integer. And as we know that because of (a + n)%n, we've proven, this returned integer can be a + n. Then, how can we find message that when we compute ascii_to_int(message), we get integer that is a + n?

Well, in part 3, we've already reverse engineered base128_to_int to make function 'int_to_base128'. We should use this function to get list of ord values, and then convert it to message.

Let's use it!

`ascii_we_need = a + n`

```
base_we_get = a4_part3.int_to_base128(ascii_we_need)
```

Note how when we compute int_to_base128, we get list of ord values, so we need additional work to convert this to new message.

```
return ‘’.join([chr(ord_value) for ord_value in base_we_get])
```

Thus, in this last line, I implemented comprehension that convert each ord values to character by using chr(). When we compute this comprehension, we get list that contains each string. We will combine all of them into one single string by using str.join(),

and finally... return it!