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

Jaeyong Lee

November 24, 2022

Part 1: Proofs

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

   Proof. Expanding the statement:

   $\forall a, b, n \in \mathbb{Z},\ \big(n \neq 0 \land n | a - b\big) \Rightarrow \big(\forall m \in \mathbb{Z},\ n | a - b - mn\big)$
   $\forall a, b, n \in \mathbb{Z},\ \big(n \neq 0 \land \exists k_1 \in \mathbb{Z},\ a - b = k_1 n\big) \Rightarrow \big(\forall m \in \mathbb{Z},\ \exists k_2 \in \mathbb{Z},\ a - b - mn = k_2 n\big)$

   Let $a, b, n \in \mathbb{Z}$. Assume $n \neq 0$ and assume $\exists k_1 \in \mathbb{Z},\ a - b = k_1 n$

   Let $m \in \mathbb{Z}$.
   We wish to show that $\exists k_2 \in \mathbb{Z},\ a - b - mn = k_2 n$

   Take $k_2 = k_1 - m$. Then,
   $a - b = k_1 n$
   $a - b - mn = k_1 n - mn$
   $a - b - mn = n(k_1 - m)$
   $a - b - mn = k_2 n$

   Therefore, we've shown that $\forall m \in \mathbb{Z},\ \exists k_2 \in \mathbb{Z},\ a - b - mn = k_2 n$, as required

   □

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. Expand the statement:
$\forall f, g : \mathbb{N} \to \mathbb{R}^{\geq 0},\ \Big(\exists c_1, n_1 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_1 \Rightarrow g(n) \leq f(n)c_1 \wedge (\forall m \in \mathbb{N}, f(m) \geq 1)\Big) \Rightarrow \Big(\exists c_2, n_2 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_2 \Rightarrow g(n) \leq c_2 \lfloor f(n) \rfloor\Big)$

Let $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$

Assume $\exists c_1, n_1 \in \mathbb{R}^+$ such that $\forall n \in \mathbb{N}$, $n \geq n_1 \Rightarrow g(n) \leq f(n)c_1$

Assume $\forall m \in \mathbb{N}, f(m) \geq 1$

Let $c_2 = 2c_1$ and Let $n_2 = n_1$

Let $n \in \mathbb{N}$ and assume $n \geq n_2$
We wish to show that: $g(n) \leq c_2 \lfloor f(n) \rfloor$

From our assumption, recall that $\forall n \in \mathbb{N}, f(n) \geq 1$. From this, we know that $\forall n \in \mathbb{N}, \lfloor f(n) \rfloor \geq 1$

Also consider the following property of the floor function which states that:
$\lfloor f(n) \rfloor \leq f(n) < \lfloor f(n) \rfloor + 1$

So, we know that $f(n) < \lfloor f(n) \rfloor + 1$
Since $\lfloor f(n) \rfloor \geq 1$, it is also true that $f(n) < \lfloor f(n) \rfloor + \lfloor f(n) \rfloor$

Then,
$f(n) < 2 \lfloor f(n) \rfloor$
$g(n) \leq c_1 f(n) < 2c_1 \lfloor f(n) \rfloor$
$g(n) < 2c_1 \lfloor f(n) \rfloor$
$g(n) < c_2 \lfloor f(n) \rfloor$, (since $2c_1 = c_2$)

Therefore, we have shown that $g(n) \leq c_2 \lfloor f(n) \rfloor$ as needed.

$\square$

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
```

Let n be the integer input

- The assignment statement total = 0 takes 1 step (constant time)
- The first for loop takes n iterations, and each iteration is 1 step.
  - We know that after each iteration of loop 1, the variable total is reassigned to be itself plus the value of i squared. In other words, the variable total is summing up consecutive squares starting from 0 up to but not including n. After loop 1 finishes, by the provided sum of consecutive squares summation formula, we know that total $= \frac{n(n+1)(2n+1)}{6}$.
- With this information, we know that the second for loop takes $\frac{n(n+1)(2n+1)}{6}$ iterations, and each iteration is one step.
- Lastly, the return statement takes one step (constant time).

So, the total running time of the first loop is n, which is $\theta(n)$
The total running time of the second loop is $\frac{n(n+1)(2n+1)}{6}$, which is $\theta(n^3)$

When we put it all together $RT_{f1}(n) = 1 + n + \frac{n(n+1)(2n+1)}{6} + 1$.

By the sum of functions theorem, we can conclude that $RT_{f1}(n) \in \theta(n^3)$

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
```

Let n be the integer input

Since the running time expression is different when $n < 5$ and when $n \geq 5$, we will analyze the running time for these two cases separately.

Case 1: $n < 5$

- The assignment statement sum_so_far = 0 takes 1 step (constant time)
- The for loop runs for n iterations with 1 step per iteration
- The return statement takes 1 step

So, for case 1, the total number of steps is 1 + n + 1, which is $\theta(n)$

Case 2: $n \geq 5$

- The assignment statement sum_so_far = 0 takes 1 step (constant time).
- The loop body takes 1 step (constant time) but the number of iterations must be calculated. Since at each iteration, the variable sum_so_far is reassigned to equal itself plus the current value of i, the sum of consecutive numbers summation formula may be useful to figure out the number of iterations it would take for sum_so_far to be greater than or equal to n since that is the condition that will stop the loop.

By the provided sum of consecutive numbers summation formula, we know that $i_k = \frac{k(k+1)}{2}$
We want $i_k \geq n$

$\frac{k(k+1)}{2} \geq n$
$k^2 + k \geq 2n$
$k^2 + k - 2n \geq 0$
$k \geq \frac{-1 \pm \sqrt{1^2 - 4(1(-2n))}}{2}$

So we need to find the smallest value of k such that $k \geq \frac{-1 \pm \sqrt{1^2 - 4(1(-2n))}}{2}$
This is exactly the definition of the ceiling function, and so the smallest value of k is $\lceil \frac{-1+\sqrt{1+8n}}{2} \rceil$
*Note in the step above, we took the positive of the quadratic equation as we must ensure that $k \in \mathbb{N}$. We also took the ceiling of this expression because we need the smallest value of k that satisfies this

inequality

Thus, the total number of steps for case 2 is $1 + \lceil \frac{-1+\sqrt{1+8n}}{2} \rceil + 1$, which is $\theta(\sqrt{n})$. Since the definition of Theta includes an n_0, we will take the final Theta expression from our $n \geq 5$ case.

So, $RT_{f2}(n) \in \theta(\sqrt{n})$

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})

    >>> rsa_sign((23, 59, 115), len_times_sum, 'Soccer is my favourite')
    365
    >>> rsa_verify((1357, 1043), len_times_sum,
  ↪ find_collision_len_times_sum('Soccer is my favourite'), 365)
    True
    """

    if all(x == chr(0) for x in message):
        return message + chr(0)
    else:
        digits = [ord(c) for c in message]
        found1, found2 = False, False
        i = 0

        while not found1 or not found2:
            if digits[i] >= 1 and not found1:
                digits[i] -= 1
                found1 = True

            elif not found2:
                digits[i] += 1
                found2 = True
            i += 1

        return ''.join([chr(x) for x in digits])
```

This algorithm works by first checking whether the given message is made up of all chr(0) characters. If such is the case, we will return the same message but with an additional chr(0) character appended to it, which would ensure that the new message is not equal to the input message (because they will have different lengths) and the len_times_sum message digest of the new message is equal to that of the input message (they will both be zero since the sum of both their ord values are zero). This ensures that the new message can be verified using the same signature as the input message when using the RSA digital signature scheme with the len_times_sum message digest.

If the given message is not made up of all chr(0) characters, we begin to iterate through the list *digits*, made up of the ord value of each character in the input message. As we iterate, we will mutate the list *digits* in two ways:

- Find one value in the list *digits* that is greater than or equal to one and subtract 1 from it
- Add 1 to one value in the list *digits*

Then, we will return a string that is the joined chr values of our mutated list *digits*. This process is correctly able to find a collision because the two mutations do not change the length of the list *digits*. They also do not change the sum of ord values since we are adding 1 to one of them but also subtracting 1 from one of them. So, since the len_times_sum message digest relies only on the length and sum of ord values of messages, the new message that we return can surely be verified using the same signature as the input message.

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

    NOTES:
        - Unlike the other two "find_collision" functions, this function takes in
↪   the public key
          used to generate signatures. Use it!
        - You may NOT simply add leading chr(0) characters to the message string.
          (While this does correctly produces a collision, we want you to think a
↪   bit harder
          to come up with a different approach.)
        - You may find it useful to review Part 1, Question 1.

    >>> rsa_sign((13, 41, 371), ascii_to_int, 'Pizza is so yummy!')
    419
    >>> rsa_verify((533, 251), ascii_to_int, find_collision_ascii_to_int((533,
↪   251), 'Pizza is so yummy!'), 419)
    True
    """

    n = public_key[0]
    target = ascii_to_int(message) + n
    return ''.join([chr(c) for c in a4_part3.int_to_base128(target)])
```

This algorithm works by first calculating a number and assigning it to the variable "target." We calculate this number to be the ascii_to_int message digest of the input message plus the n value of the public key. This is based on the following statement:

- $\forall a, n, m \in \mathbb{Z},\ \left(a \equiv a + mn \pmod{n}\right)$
  - So, ascii_to_int(message + n) % n == ascii_to_int(message) % n

This modular equivalence ensures that a message generated by taking the chr values of the list returned by the int_to_base128 conversion of our "target" integer can be verified using the same signature as the input message (the return statement returns just that, the list of chr values joined using the str.join method). This is so because rsa_sign with the ascii_to_int digest involves raising the message digest to a certain power (the d value of the private key) and then returning that value modulo n. The ascii_to_int digest of the original message and the ascii_to_int digest of the new message generated by this aglorithm have the same remainder on division by n, and therefore can both be verified using the same signature.