CSC110 Fall 2022 Assignment 4: Loops, Mutation, and Applications

Vrinda Subhash

November 23, 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. Let a, b, n $\in \mathbb{Z}$
   I can assume: $n \neq 0 \land a \equiv b(modn)$.
   By definition of modular equivalence, $a \equiv b(modn)$ means $n|a-b$.
   By definition of divisibility, $n|a-b$ means $\exists k_1 \in \mathbb{Z}, a-b = k_1 n$.
   I want to show: $\forall m \in \mathbb{Z}, a \equiv b + mn(modn)$.
   By definition of modular equivalence, $a \equiv b + mn(modn)$, means $n|(a-(b+mn))$.
   By definition of divisibility, $n|(a-(b+mn))$ means $\exists k_2 \in \mathbb{Z}, (a-(b+mn)) = k_2 n$.
   Let $m \in \mathbb{Z}$.
   By the assumption,
   $a - b = k_1 n$
   $a - b - mn = k_1 n - mn$
   $a - b - mn = (k_1 - m)n$
   Let $k_2 = k_1 - m$
   $k_2 \in \mathbb{Z}$ because $k_1, m \in \mathbb{Z}$.
   Then, $a - b - mn = k_2 n$ as required.
   Therefore, $\forall a, b, n \in \mathbb{Z},\ \big(n \neq 0 \land a \equiv b \pmod{n}\big) \Rightarrow \big(\forall m \in \mathbb{Z},\ a \equiv b + mn \pmod{n}\big)$ □

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

Proof. Want to show: $((\exists c_1, n_1 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_1 \Rightarrow g(n) \leq c_1 f(n)) \wedge (\forall m \in \mathbb{N}, f(m) \geq 1))$
$\Rightarrow (\exists c, n_0 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_0 \Rightarrow g(n) \leq c\lfloor f(n) \rfloor)$
Let $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$.
I can assume $g \in \mathcal{O}(f)$ and $\forall m \in \mathbb{N}, f(m) \geq 1$.
By these assumptions, there exists $c_1, n_1 \in \mathbb{R}^+$ such that for all $n \in \mathbb{N}$, if $n \geq n_1$, then $g(n) \leq c_1 f(n)$.
We want to prove that $g \in \mathcal{O}\lfloor f \rfloor$, which means proving there exists $c, n_0 \in \mathbb{R}^+$ such that for all $n \in \mathbb{N}$, if $n \geq n_0$ then $g(n) \leq c\lfloor f(n) \rfloor$.
Let $\lfloor f \rfloor$: $\mathbb{N} \to \mathbb{R}^{\geq 0}$
Let $n_0 = n_1$
Let $c = 2c_1$
Let $n \in \mathbb{N}$
Assume for all n, $n \geq n_0$
Want to prove $g(n) \leq c\lfloor f(n) \rfloor$.
Since $f(n) \geq 1$, we know $\lfloor f(n) \rfloor \geq 1$ (for all $n \in \mathbb{N}$).
Using the given property of the floor function, we know $f(n) < \lfloor f(n) \rfloor + 1$
Combining the inequalities, we get:
$1 + f(n) \leq \lfloor f(n) \rfloor + \lfloor f(n) \rfloor + 1$
$f(n) \leq 2\lfloor f(n) \rfloor$
$c_1 f(n) \leq 2c_1 \lfloor f(n) \rfloor$
$c_1 f(n) \leq c\lfloor f(n) \rfloor$
$g(n) \leq c_1 f(n) \leq c\lfloor f(n) \rfloor$
Therefore: $g(n) \leq c\lfloor f(n) \rfloor$ as required. $\square$

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
```

Running Time Analysis:
- Let n be the integer input to f1.
- The assignment statement ”total = 0” counts as 1 step. Its running time does not depend on how big the input integer is. $\rightarrow$ 1 step
- Loop 1 has n iterations and each iteration takes 1 step. $\rightarrow$ n $\times$ 1 = n steps
- Loop 2 has $\frac{n(n+1)(2n+1)}{6}$ iterations, and each iteration takes 1 step. $\rightarrow n^3 \times 1 = n^3$ steps
- The return statement ”return total” counts as 1 step. $\rightarrow$ 1 step

$RT_{f1}(n) = 1 + n + n^3 + 1$
$\in \theta(n^3)$

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
```

Running Time Analysis:
- Let n be the integer input to f2.
- The assignment statement "sum_so_far = 0" counts as 1 step. $\rightarrow$ 1 step
- Loop 1 iterates while:
sum_so_far < n
$\frac{i(i+1)}{2} < n$
$\frac{i^2+i}{2} < n$
$i^2 + i < 2n$
$\sqrt{i^2 + i} < \sqrt{2} \times \sqrt{n}$
Approximately becomes $\rightarrow i + \sqrt{i} < \sqrt{2} \times \sqrt{n}$
Can ignore the $\sqrt{i}$ on the left side of the inequality.
Can ignore the $\sqrt{2}$ on the right side of the inequality.
$i < \sqrt{n}$
So, iterates about $\sqrt{n}$ times and each iteration takes about 2 steps. $\rightarrow \sqrt{n} \times 2 = 2\sqrt{n}$ steps
- The return statement "return 0" counts as 1 step. $\rightarrow$ 1 step
$RT_{f2}(n) = 1 + 2\sqrt{n} + 1$
$\in \theta(\sqrt{n})$.

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
    """

    msg_list = list(message)

    to_swap = msg_list[0]

    # swapping two characters that are not the same
    for i in range(1, len(msg_list)):
        if msg_list[i] != to_swap:
            msg_list[0] = msg_list[i]
            msg_list[i] = to_swap
            return ''.join(msg_list)

    # if the code reaches here, it means the message has all the same characters
    # increasing the ord of the first character by 1 and decreasing the ord of the
    second character by 1
    ord_of_first = ord(msg_list[0])
    ord_of_second = ord(msg_list[1])
    msg_list[0] = chr(ord_of_first + 1)
    msg_list[1] = chr(ord_of_second - 1)
    return ''.join(msg_list)
```

Take the message and find two characters that aren't the same and swap them. This produces a different string than the input message, that can be verified using the same signature since it is still all the same original characters just in a new order. If all the characters in the string are the same, increase the ord value of the first character by 1 and decrease the ord value of the second character by 1. This will also produce a different string that can be verified by the same signature since the length of the new message will remain the same and the sum of the ord values of all the characters will be the same since 1 was added and 1 was subtracted.

4b.
```
def find_signature(public_key: tuple[int, int], message: str) -> int:
    """Brute force tries all possible signature values until the right one is found.
    """
    n, _ = public_key
    for i in range(n):
        if rsa_verify(public_key, ascii_to_int, message, i):
            return i
    # unable to find signature
    return 0


@check_contracts
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
    len_times_sum, 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.
    """

    signature = find_signature(public_key, message)
    if signature == 0:
        # failed to find signature
        return ''

    n, _ = public_key

    m_length = int(n / 2)
    tmp_list = ['a' for _ in range(m_length)]

    # brute force search
    for i in range(26):
        for j in range(m_length):
            next_char = ord('a') + i
            tmp_list[j] = chr(next_char)
```

```
            tmp_message = ''.join(tmp_list)
            if rsa_verify(public_key, ascii_to_int, tmp_message, signature):
                return tmp_message
            next_char = ord('A') + i
            tmp_list[j] = chr(next_char)
            tmp_message = ''.join(tmp_list)
            if rsa_verify(public_key, ascii_to_int, tmp_message, signature):
                return tmp_message

    # failed to find new message
    return ''
```

Discover the signature by using the public key and trying out all possible signature values until it is True. The signature is a value between 0 and n, so you try all the possible signature values until you find the right one. Once you know the signature, generate a lot of strings until you find a string that has the same signature. The algorithm for generating strings is as follows: Decide on a string length, I used string length of n/2. Found this by trial and error. For each position in the string, try all the characters 'a' to 'z' and 'A' to 'Z'. The string that creates a digest that matches the discovered signature, decrypted, will be returned.