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

Print this handout

In this assignment, you'll apply what you've learned in the past few weeks about number theory, cryptography, asymptotic function growth, and algorithm running time. In Part 1, you'll prove some mathematical statements involving familiar definitions. In Part 2, you'll analyse the running time of some functions where you'll need to take into account how the loop accumulator changes. And in Parts and 4, you'll extend the RSA public-key cryptosystem in two new ways that bring us closer to how RSA used in the real world.

Logistics

- Due date: Wednesday, November 23 before 12pm noon Eastern Time.
  - We are using **grace credits** to allow you to submit your work up to 30 hours after the due date. Please review the Policies and Guidelines: Assignments page (https://q.utoronto.ca/courses/278340/pages/policies-and-guidelines-assignments?module_item_id=3928858) carefully!
- This assignment must be done **individually**.
- You will submit your assignment solutions on MarkUs (see Submission instructions at the end of this handout).

Starter files

To obtain the starter files for this assignment:

1. Login to MarkUs (https://markus.teach.cs.toronto.edu/2022-09) and go to CSC110.
2. Under the list of assignments, click on **Assignment 4**.

3. On the assignment page, click on the **Download Starter Files** button (near the bottom of the page). This will download a zip file to your computer.
4. Extract the contents of this zip file into your `csc110/assignments/` folder.
5. You should see a new `a4` folder that contains the assignment's starter files!

You can then see these files in your PyCharm `csc110` project, and also upload the `a4.tex` file to Overleaf to complete your written work.

General instructions

This assignment contains a mixture of both written and programming questions. All of your written work should be completed in the `a4.tex` starter file using the LaTeX typesetting language. You went through the process of getting started with LaTeX in the Software Installation Guide (https://q.utoronto.ca/courses/160038/pages/setting-up-your-computer-start-here?module_item_id=1346385), but for a quick start we recommend using the online platform Overleaf (https://www.overleaf.com/) for LaTeX. Overleaf also provides many tutorials (https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes) to help you get started with LaTeX.

Tip: See this video tutorial (https://www.overleaf.com/learn/how-to/How_to_upload_files_to_a_new_project) for instructions on how to upload a `.tex` file to Overleaf.

Your programming work should be completed in the different starter files provided (each part has its own starter file). We have provided code at the bottom of each file for running doctest examples and PythonTA on each file. We are **not** grading doctests on this assignment, but encourage you to add som as a way to understand and test each function we've asked you to complete. We *are* using PythonTA to grade your work, so please run that on every Python file you submit using the code we've provided.

Warning: one of the purposes of this assignment is to evaluate your understanding and mastery of the concepts that we have covered so far. So on this assignment, you may only use parts of the Python programming language that we have covered in Chapters 1-9 of the Course Notes. Other parts are not allowed, and parts of your submissions that use them may receive a grade as low as **zero** for doing so.

Part 1: Proofs

Answer the following questions in `a4.tex`.

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

   You may **only** use the definitions of modular equivalence and divisibility in your proofs, and may not use any properties/theorems of modular equivalence or divisibility, including ones we have covered in the course.

2. The floor function $\lfloor \cdot \rfloor$ takes a real number x and returns the largest integer that is less than or equal to x. For each function $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, we define its *corresponding floor function*, $\lfloor f \rfloor$, to be

   $$(\lfloor f \rfloor)(n) = \lfloor f(n) \rfloor \quad \text{for all } n \in \mathbb{N}$$

   Prove that for all functions $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$, if $g \in \mathcal{O}(f)$ and if for all $m \in \mathbb{N}$, $f(m) \geq 1$, then $g \in \mathcal{O}(\lfloor f \rfloor)$.

   Notes:

   - Since $f(m) \geq 1$, you also know $\lfloor f(m) \rfloor \geq 1$ (for all $m \in \mathbb{N}$).
   - You may use the following property of the floor function:

     $$\forall x \in \mathbb{R},\ \ \lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$$

     Substituting $f(n)$ for x, this gives you the inequality $\lfloor f(n) \rfloor \leq f(n) < \lfloor f(n) \rfloor + 1$.

Part 2: Running-Time Analysis

NOTE: we will be introducing formal running-time analysis in lecture on Thursday, November 3. We also strongly recommend reading Section 9.5 (https://www.teach.cs.toronto.edu/~csc110y/fall/notes/09-runtime/05-basic-algorithm-analysis.html) of the Course Notes before completing this part.

In lecture, we began our study of running-time analysis with functions that contain only constant-time operations and for loops. In this part, you will analyse the running time of functions where the for loop have a number of iterations that is not immediately obvious, and you'll need to do some work in your analysis to determine the number of iterations.

You may use the following formulas (from [Appendix C.1 (https://www.teach.cs.toronto.edu/~csc110y/fall/notes/C-math-reference/01-summations-products.html)](https://www.teach.cs.toronto.edu/~csc110y/fall/notes/C-math-reference/01-summations-products.html) of the Course Notes).

For all $n \in \mathbb{N}$:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \qquad \text{(sum of consecutive numbers}$$

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \qquad \text{(sum of consecutive squares)}$$

Analyse the running time of each the following functions. Write your responses to this question in a4.tex`.

Note (added Nov 14) In your running-time analyses, you may go directly from an exact step count (e.g., $n^2 + 2n + 4$) directly to a Theta expression (e.g., $\Theta(n^2)$) without formal proof *or even justification*. This matches the style of running-time analysis we've seen in lecture.

0. *Warm-up, not to be handed in.*

```
def f0(n: int) -> int:
    """Precondition: n >= 0"""
    total = 0

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

    return total
```

1.
```python
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
```

Hint: in your analysis, first determine the value of `total` (in terms of `n`) after Loop 1 ends.

2.
```python
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
```

Hint: Loop 1 has an early return, and will iterate fewer than n times when $n \geq 5$. In your analysis find a formula for `sum_so_far` at iteration k, and then determine the smallest value of k (in term of `n`) when the if condition will be True.

You may use the quadratic formula in your analysis. You may also find it helpful to review this section (https://www.math.toronto.edu/preparing-for-calculus/2_inequalities/we_2_intervals.html) of the Department of Mathematics Precalculus prep.

Part 3: Block-Based RSA

In lecture we introduced the RSA cryptosystem, a public-key cryptosystem that uses ideas from modul arithmetic in its encryption and decryption algorithms. One of the limitations of the version of RSA we studied is that it encrypted strings one character at a time, making it vulnerable to various forms of cryptographic attacks like *frequency analysis* on the ciphertext.[1] So in practice, RSA is typically implemented to operate on *blocks* of characters rather than individual characters. You'll learn about o such approach in this part of the assignment!

Part (a): From strings to numbers

The key new idea required for block-based string encryption is converting a string containing more tha one character into a number. We already know how to convert between a single character and an integ representation, using the built-in `ord` and `chr` functions:

```
>>> ord('A')
65
>>> chr(65)
'A'
```

How do we generalize this to strings with more than one character? Read the framed text below to find out!

ASCII strings as base-128 numbers

Consider the string `'Hello'`. We can convert it into a list of integers by applying the `ord` function to each character:

```
>>> [ord(c) for c in 'Hello']
[72, 101, 108, 108, 111]
```

We then need to convert this list of integers into a single integer. However, our familiar built-in aggregation functions like `sum` and `max` aren't sufficient, since we need to ensure that each string get mapped to a unique integer (so that it is encrypted to a unique ciphertext message).

Here is our key simplifying assumption for this part: *we will assume the plaintext messages contain only ASCII characters*, which corresponds to the characters whose `ord` value is between `0` and `127`, inclusive.

This tells us that for any string `plaintext`, the list `[ord(c) for c in plaintext]` will only contain integers between `0` and `127`. This allows us to treat this list as a *base-128* number, and so turn the list into a single integer. But before talking about base-128 further, let's formally introduce the concept of a base-n number system.

We humans are familiar with the **base-10** number system, which represents numbers as a sequence of digits from 0 to 9. For example, when we write the number "324", we can break this down into the three digits $[3, 2, 4]$, with the numerical value $3 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$. In general, a number in base-10 consists of a sequence of digits $d_{k_1} d_{k-2} \dots d_1 d_0$, where each digit d_i is in $\{0, 1, \dots, 9\}$. The *value* of this number is calculated by the formula

$$\sum_{i=0}^{k-1} d_i \times 10^i$$

In words, the right-most digit is multiplied by 10^0, the next digit to the left is multiplied by 10^1, and so on. Each digit to the left has a multiplier that is 10 times the multiplier of the previous digit. In ou example "324", we have $d_2 = 3$, $d_1 = 2$, and $d_0 = 4$, and so the value is $3 \times 10^2 + 2 \times 10^1 + 4 \times 10$
. Let's now generalize this to bases other than 10.

Definition. Let $n \in \mathbb{Z}^+$ where $n \geq 2$. We define the **base-n** number system as follows:

- A number in base-n is a sequence of digits $d_{k-1} d_{k-2} \dots d_1 d_0$, where each digit d_i is in $\{0, 1, \dots, n-1\}$.
- We typically write this number as $(d_{k-1} d_{k-2} \dots d_1 d_0)_n$, so that it isn't confused with a base-10 number.
- The *value* of this number is $\sum_{i=0}^{k-1} d_i \times n^i$.

Let's apply this definition to base-128 (i.e., $n = 128$). Consider the number $(324)_{128}$, which has the same three digits as before ($d_2 = 3$, $d_1 = 2$, $d_0 = 4$), but now has a new numerical value because each of these digits is multiplied by a power of 128 instead of a power of 10:

$$(324)_{128} = 3 \times 128^2 + 2 \times 128^1 + 4 \times 128^0 = 49412$$

Note that "49412" is written in base-10, which we're using to tell you the value of $(324)_{128}$ in a way you can understand!

Now let's apply this to our string example. We left off by converting the string `'Hello'` into a list of integers:

```
>>> [ord(c) for c in 'Hello']
[72, 101, 108, 108, 111]
```

We can now treat this list of integers as digits in base-128, and thereby calculate a single integer from the list. To be consistent with the notation used above, we'll consider the *last* (or rightmost) element of the list to be d_0, the second-last element to be d_1, etc. Here is the calculation:

$$72 \times 128^4 + 101 \times 128^3 + 108 \times 128^2 + 108 \times 128^1 + 111 \times 128^0 = 19540948591$$

That's a pretty big number, from such a small string! Working in base-128 means that just five digits can represent all of the numbers between 0 and $128^5 - 1$, inclusive.

Now, open the `a4_part3.py` starter file.

1. Your first task for this part is to implement the function `base128_to_int`, in which you'll conve a list of digits (like `[72, 101, 108, 108, 111]`) into a single integer by performing the base-128 calculation.
2. Your second task is to implement `int_to_base128`, which does the reverse: convert a positive integer into its base-128 represention. This requires a more complex algorithm than the first question, so please pay attention to the hints we've provided in the function docstring!

Part (b): Encrypting and decrypting blocks

Now that you've completed the two conversion functions `base128_to_int` and `int_to_base128`, let's apply them to the RSA cryptosystem. As a first step, please review the character-based implementation of RSA's encryption algorithm (the decryption function is similar):

```
def rsa_encrypt_text(public_key: tuple[int, int], plaintext: str) -> str:
    """Encrypt the given plaintext using the recipient's public key.

    Preconditions:
        - public_key is a valid RSA public key (n, e)
        - all({ord(c) < public_key[0] for c in plaintext})
    """
    n, e = public_key

    encrypted = ''
    for letter in plaintext:
        encrypted = encrypted + chr((ord(letter) ** e) % n)

    return encrypted
```

Your task for this part will be to modify this algorithm so that it operations on larger *blocks* of `plaintext` rather than one character at a time.

The Block-based RSA cryptosystem

The **block-based RSA cryptosystem** is like the version of RSA we saw in lecture, with the following differences:

- **Key generation** is the same, *except* that the primes p and q must satisfy $pq \geq 128$.
- For a given public-private key pair with modulus $n = pq$, we define the **block length** to be the largest integer k such that $128^k \leq n$.
  - This definition ensures that all blocks will be converted to a unique integer modulo n during encryption.
- The **plaintexts** are strings consisting of only ASCII characters, with the additional restriction that *the length of the plaintext must be divisible by the block length*.
  - This ensures that there won't be a "short block" at the end of the message.
- The **ciphertexts** are now *lists of non-negative integers*, where each element is $< n$. (See the next point about encryption.)
- **Encryption** now works as follows:

  1. Divide the plaintext message into blocks of the calculated *block length*.

- For example, if the block length is 4 and the plaintext is `'Hello David and Tom!'`, there are five blocks:

```
'Hell'
'o Da'
'vid '
'and '
'Tom!'
```

2. Convert each block into an integer using the base-128 transformation.
3. Apply the standard RSA modular exponentiation, $x^e \ \% \ n$, to each integer (one integer per block).
4. The resulting list of integers is the encrypted ciphertext.

- **Decryption** reverses encryption.
  - This will be your responsibility to figure out and implement correctly!

There are three key implementation details that we'll note here:

1. *How do we iterate through blocks of plaintext (instead of just one character at a time)?* One Python feature that you will find useful for this is **slicing**, which allows you to extract a substring of a string based on start and end indexes. For example:

```
>>> plaintext = 'Hello David and Tom!'
>>> plaintext[0:4]  # Like range, start index is INCLUDED and stop index
      is EXCLUDED
'Hell'
>>> plaintext[4:8]
'o Da'
```

   For this given `plaintext` and a block length of 4, the blocks would be `plaintext[0:4]`, `plaintext[4:8]`, `plaintext[8:12]`, `plaintext[12:16]`, `plaintext[16:20]`. You will need to a use a for loop or while loop to iterate over the possible slice indexes (e.g., `0`, `4`, `8`, `12`, `16`

2. *What if `int_to_base128` returns a list that's shorter than the required block length?* For example, if the block length is 4 but you call `int_to_base128(1)`, just `[1]` will be returned. Yo need to check for this and write some code to *add leading zeros*, e.g. turning `[1]` into `[0, 0, 0 1]`.

3. *How can we test our code on larger primes?* You can use the `rsa_generate_key` function from lecture (which we've provided as a helper function in `a4_helpers.py`). You shouldn't add an import statement to `a4_part3.py`, but you can import `a4_helpers` into the Python console (or separate testing file) to test your work.

   For the arguments to pass in, we encourage you to use online known lists of primes, e.g. Wikipedia's list of the first 1000 primes (https://en.wikipedia.org/wiki/List_of_prime_numbers#The_first_1000_prime_numbers). As note, here are the first few powers of 128 (you will find this useful to pick primes that will result in a particular block length).

$$
\begin{aligned}
128^1 &= 128 \\
128^2 &= 16384 \\
128^3 &= 2097152 \\
128^4 &= 268435456 \\
128^5 &= 34359738368
\end{aligned}
$$

Part 4: Digital Signatures

We saw in lecture how public-key cryptosystems can be used to create *digital signatures* to provide evidence for the authenticity of a message. Intuitively, if Alice has a public key k_1 and corresponding private key k_2, and wants to send a message m, then she can use her private key k_2 to generate a signature for m, and send both m and the signature to Bob. Then, Bob can use Alice's public key to verify that the signature indeed matches the message.

Please read the following box for the definition of a *secure digital signature scheme*, which formalizes this idea.

Secure digital signature schemes

A **secure digital signature scheme** consists of the following components:

- A set $\mathcal{P}$ of possible messages, called **plaintext** messages. (E.g., a set of strings)
- A set $\mathcal{S}$ of possible **signatures** (E.g., a set of numbers)
- A set $\mathcal{K}_1$ of possible public keys and a set $\mathcal{K}_2$ of possible private keys.

- A subset $\mathcal{K} \subseteq \mathcal{K}_1 \times \mathcal{K}_2$ of possible **public-private key pairs**.
- Two functions $\mathit{Sign} : \mathcal{K}_2 \times \mathcal{P} \to \mathcal{S}$ and $\mathit{Verify} : \mathcal{K}_1 \times \mathcal{P} \times \mathcal{S} \to \{\mathit{True}, \mathit{False}\}$ that satisfy the following two properties:
  - (**correctness**) For all $(k_1, k_2) \in \mathcal{K}$ and $m \in \mathcal{P}$, $\mathit{Verify}(k_1, m, \mathit{Sign}(k_2, m)) = \mathit{True}$.

    That is, if you generate a signature for a message using a private key, anyone can verify that signature using the corresponding public key.
  - (**security**) For all $(k_1, k_2) \in \mathcal{K}$ and $m \in \mathcal{P}$, if an eavesdropper knows k_1, m, and $s = \mathit{Sign}(k_2, m)$, but doesn't know the private key k_2, it is computationally infeasible for the eavesdropper to compute a new $m' \in \mathcal{P}$ and $s' \in \mathcal{S}$ that satisfy both:

    1. $m \neq m'$ (i.e., the eavesdropper's message is different from the original message)
    2. $\mathit{Verify}(k_1, m', s') = \mathit{True}$ (i.e., the eavesdropper's signature s' can be verified for the chosen m').

    Note: this is only one (simple) definition of security for digital signature schemes. For more information, please see our Going further subsection at the end of this part.

Part (a): Introduction

In the `a4_part4.py` starter file under "Part (a)", we have provided an implementation of a digital signature scheme based on the RSA cryptosystem, in the two functions `rsa_sign_simple` and `rsa_verify_simple`.

For the RSA digital signature schemes we'll study on this assignment, the messages will be *strings*, anc the signatures will be *integers*. Because RSA is based on modular arithmetic, we'll need to convert our messages to integers in order to sign them. Given a message m, we define a **message digest** of m to b an integer that was computed from m using some pre-determined function.

Using this definition, let's formalize what an RSA digital signature scheme looks like.

RSA digital signature scheme

An RSA digital signature scheme has the following components:

- The plaintext messages are *strings*.

- The signatures are *natural numbers*.
- $\mathcal{K}$ is the set of RSA public-private key pairs (identical to the RSA cryptosystem from lecture). So the public keys are of the form (n, e), and the private keys are of the form (p, q, d).
- The $\mathit{Sign}((p, q, d), m)$ function is defined as follows:
  1. First, compute the message digest $a \in \mathbb{Z}$ for m.
  2. Compute $n = pq$.
  3. Then, compute $s = a^d \ \% \ n$. This is the signature for the message.
- The $\mathit{Verify}((n, e), m, s)$ function is defined as follows:
  1. First, compute the message digest $a \in \mathbb{Z}$ for m.
  2. Compute $a' = s^e \ \% \ n$.
  3. Return whether $a' = a$.

Comment: the Sign and Verify algorithms are essentially the same as the encryption and decryption algorithms from RSA, except the roles of the private key and public key are reversed.

Now, the only part of the above description that is underspecified is what exactly we mean by a messag digest. And indeed, different choices of the digest result in different signatures! In this part of the assignment, you'll explore how different choices of digests lead to different security (or "*non-security*" properties for the RSA digital signature scheme.

Now open `a4_part4.py` and complete the following questions:

1. Read through the two functions `rsa_sign_simple` and `rsa_verify_simple`. These two functions use the *length of the message* to compute the message digest.
2. Suppose we have the key pair $(n, e) = (1357, 1043)$ and $(p, q, d) = (23, 59, 115)$. Answer the following questions:
   a. Use the Python console to compute the signature for the message `'Cryptography is cool'` using the given private key. *Then, add a doctest example* to `rsa_sign_simple` to show this example.
   b. Use the Python console to verify that the signature from the previous step matches `'Cryptography is cool'` and the given public key. *Then, add a doctest example* to `rsa_verify_simple` to show this example.
   c. Finally, use the Python console to check that `rsa_verify_simple` returns `False` when given the same public key and same message as the example from (b), but a different

signature number. *Then, add this as another doctest example* to `rsa_verify_simple`.

3. Now, some analysis. Computing the message digest using the length of the message leads to a *correct* digital signature scheme, but not a secure one. This is because given one message m, it is *very easy* to find another message $m' \neq m$ that has the same digest as m!

   a. To illustrate this idea, first complete the test case `test_collision_simple`, for which you'll need to find a second string that has the same signature as `'Cryptography is cool'`.
   b. Then, you'll generalize this idea by completing the function `find_collision_simple`. Fo this function, you will play the role of an eavesdropper who has seen one valid message and signature, and will now be able to efficiently compute a new message that can be verified wit that same signature!

Part (b): Generalizing the message digests

Computing message digests using just the message length is not secure—big surprise. Now we'll explor two other candidate message digests. But before we do so, we'll present a generalized implementation the sign/verify functions in the two functions `rsa_sign` and `rsa_verify`, found under the "Part (b)" header of the `a4_part4.py` starter file.

These two functions are similar to `rsa_sign_simple` and `rsa_verify_simple`, except they now have an additional parameter called `compute_digest`. This parameter has a new type annotation: `Callable[[str], int]`. This type annotation indicates that `compute_digest` is a **function** that takes in a `str` and returns an `int`—in other words, `rsa_sign` and `rsa_verify` are functions that tak in another function as an argument!

This might seem a bit strange, so please read through the functions carefully, paying special attention t how `compute_digest` is used in the function body. Note that the doctest examples pass in the built-i `len` function as an argument, so that the behaviour illustrated is the same as `rsa_sign_simple`/`rsa_verify_simple`.

When you're ready to continue, complete the following tasks:

1. Implement the following two message digest functions:

   - `len_times_sum`: return the length of the message multiplied by the sum of the `ord` values each character in the message.

- `ascii_to_int`: given a message with *only* ASCII characters, interpret it as a base-128 representation and return the integer value corresponding to that representation. (This is very similar to what you did for Part 3, and you may import your work from `a4_part3.py` into this file.)

2. (*not to be handed in*) For each message digest function from Question 1, use the Python console t

   a. Compute the digest value of the message `'hello'`.
   b. Compute the *signature* of the message `'hello'` when signed with the RSA private key $(23, 59, 115)$.
   c. *Verify* the computed signature using the corresponding RSA public key $(1357, 1043)$.

3. Perhaps unsurprisingly, `len_times_sum` is *not* a good choice of message digest, and also leads t an unsecure digital signature scheme.

   a. Complete the test case `test_collision_len_times_sum` and "generalized" function `find_collision_len_times_sum`. These are analogous to the functions you completed i part (a), but now for the `len_times_sum` digest. You may, but are not required, to add doctest examples as part of your testing.

   Note (added Nov 14): to make this part a bit easier, we are adding the following *precondition* to `find_collision_len_times_sum`:

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

   Since 1114111 is the maximum possible `ord` value for a Python character, this precondition ensures 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.

   (Please note, however, that the message may contain all `chr(0)` characters, and so it won't necessarily be possible to *subtract* 1 from the ord value of at least one character in the message.)

   b. In `a4.tex`, provide a copy of your code for `find_collision_len_times_sum`, and expla (in English) your algorithm. You should both explain the steps involved and justify *why* it works, but a formal proof of correctness is *not* necessary.

   Note: do not simply restate each line of code in English. Your explanation should focus on explaining the ideas behind your code.

4. A bit more surprisingly, `ascii_to_int` is *also* an unsecure choice for message digest, though it i a bit trickier to see why.

   In Part 3, we used a base-128 representation to map blocks of characters to integers for RSA encryption. We chose the block length so that each block of ASCII characters would be mapped to a unique integer. However, with digital signatures we do not require that the message be a bounded length, as in practice message digests are computed on data from a few kilobytes to hundreds of gigabytes (and larger) in size. So for digital signature schemes, it is inevitable that **two different messages may generate the same digest modulo n**.

   The question of security is whether it is *computationally feasible* to find two such messages—and the answer is *yes* when using the `ascii_to_int` message digest.

   a. Complete the test case `test_collision_ascii_to_int` and "generalized" function `find_collision_ascii_to_int`. You may, but are not required, to add doctest example as part of your testing.

   b. In `a4.tex`, provide a copy of your code for `find_collision_ascii_to_int`, and explai (in English) your algorithm. You should both explain the steps involved and justify *why* it works, but a formal proof of correctness is *not* necessary.

      Note: do not simply restate each line of code in English. Your explanation should focus on explaining the ideas behind your code.

Going further

That's the end of this assignment! But of course, this might be a bit of an unsatisfying end—after all, w studied three possible message digests (`len`, `len_times_sum`, and `ascii_to_int`) that led to unsecure digital signature schemes. You might wonder, then, about what people actually do in practic and whether there really are message digests that are considered "secure".

Indeed there are: in practice, we use **cryptographic hash functions** to compute message digests. These functions take in arbitrary forms of data—including strings, but really arbitrary sequences of 0/ bits—and return integer digests. But unlike the digest functions we studied on this assignment, these cryptographic hash functions are much more complex. And due to their complexity, it is not computationally feasible to find digest or signature collisions when they are used.

To find out more, check out the Wikipedia page on cryptographic hash functions (https://en.wikipedia.org/wiki/Cryptographic_hash_function) and the Python `hashlib` module (https://docs.python.org/3/library/hashlib.html).

You might also find it interesting to learn more about the **different definitions of security** for digital signatures that go beyond what we've used on this assignment. For example, what if an eavesdropper has access to not just one message and signature, but hundreds or thousands? Or how do we distinguish between cases where the eavesdropper can generate a new message that matches an existing signature vs. where the eavesdropper has the ability to generate a valid signature for an arbitrary message of their choice (much worse!!)?

To find out more, check out the Wikipedia page on digital signatures (https://en.wikipedia.org/wiki/Digital_signature) or this seminal paper (https://people.csail.mit.edu/rivest/GoldwasserMicaliRivest-ADigitalSignatureSchemeSecureAgainstAdaptiveChosenMessageAttacks.pdf) by Shafi Goldwasser, Silvio Micali, and Ron Rivest. ("Rivest" as in the "R" in **R**SA!)

Submission instructions

Please **proofread** and **test** your work carefully before your final submission! As we explain in Runnin and Testing Your Code Before Submission (https://q.utoronto.ca/courses/278340/pages/running-anc testing-your-code-before-submission?module_item_id=3953751), it is essential that your submitted code not contain syntax errors. **Python files that contain syntax errors will receive a grade of on all automated testing components (though they may receive partial or full credit on any TA grading for assignments).** You have lots of time to work on this assignment and check you work (and right-click -> "Run in Python Console"), so please make sure to do this regularly and fix syntax errors right away.

1. Login to MarkUs (https://markus.teach.cs.toronto.edu/2022-09/courses/1).
2. Go to Assignment 4, then the "Submissions" tab.
3. Submit the following files: `a4.tex`, `a4.pdf` (which must be generated from your `a4.tex` file), `a4_part3.py`, and `a4_part4.py`. Please note that MarkUs is picky with filenames, and so your filenames must match these exactly, including using lowercase letters.

- Please check out [the Overleaf tutorial on exporting project files (https://www.overleaf.com/learn/how-to/Exporting_your_work_from_Overleaf)](https://www.overleaf.com/learn/how-to/Exporting_your_work_from_Overleaf) to learn how to download your `a4.tex` and `a4.pdf` files from Overleaf.

4. Refresh the page, and then *download each file* to make sure you submitted the right version.

Remember, you can submit your files multiple times before the due date. So you can aim to submit you work early, and if you find an error or a place to improve before the due date, you can still make your changes and resubmit your work.

After you've submitted your work, please give yourself a well-deserved pat on the back and go take a re or do something fun or look at some dynamical systems or eat some chocolate! *Holy cat, you've just completed your last assignment for CSC110.*

---

1. Frequency analysis looks at the number of times each character appears in a ciphertext, and base on these frequencies makes guess about the corresponding plaintext characters.↩