CSC110 / CSC111

Week 3

3.1 Propositional Logic

- A Preposition is a statement that is either TRUE or FALSE.

1. AND. OR. NOT.
2. The implication operator ⇒
    - false ⇒ true: Vacuous Truth Cases
    - A → B
        - NOT A OR B
        - NOT B → NOT A (Contrapositive)

          B → A (Converse)
3. Biconditional (if and only if)

3.2 Predicate Logic

- A Predicate is a statement whose truth value **depends** on variables.
-

1. Existential Quantifier

- there exist (”OR”)

2. Universal Quantifer

- for all (”AND”)

3. Python built-ins: `any` ***and*** `all`

```
>>> any([False, False, True])
True
>>> all([False, False, True])
False

>>> any([])
False # empty set has no True value.
```

```
# To check if a string has a char for its first letter
>>> strings = ['Hello', 'Goodbye', 'David']
>>> any([s[0] == 'D' for s in strings])

>>> strings = ['
```

4. Where to put commas?

- immediately after a variable quantification, or separating two variables with the same quantification
- separating arguments to a predicate function

- $\neg(\neg p)$ becomes p.
- $\neg(p \vee q)$ becomes $(\neg p) \wedge (\neg q)$.[7]
- $\neg(p \wedge q)$ becomes $(\neg p) \vee (\neg q)$.
- $\neg(p \Rightarrow q)$ becomes $p \wedge (\neg q)$.[8]
- $\neg(p \Leftrightarrow q)$ becomes $(p \wedge (\neg q)) \vee ((\neg p) \wedge q))$.
- $\neg(\exists x \in S,\ P(x))$ becomes $\forall x \in S,\ \neg P(x)$.
- $\neg(\forall x \in S,\ P(x))$ becomes $\exists x \in S,\ \neg P(x)$.

3.3 Filtering collections in Python

‘Only use FOR when we are iterating a collection! Not for a condition.”

```
# Filtering set comprehension
{<expression> for <variable> in <collection> if <condition>}

# Filtering list comprehension
[<expression> for <variable> in <collection> if <condition>]

# Filtering dictionary comprehension
{<key_expr> : <value_expr> for <variable> in <collection> if <condition>}
```

Examples:

```
>>> numbers = {1, 2, 3, 4, 5, 6, 7, 8}
>>> all({n ** 2 + n >= 20for nin numbersif n > 3})
True
```

```
>>> numbers = {1, 2, 3, 4, 5}  # Initial collection
>>> {n for n in numbers if n > 3}  # Pure filtering: only keep elements > 3
{4, 5}
>>> {n * n for n in numbers if n > 3}  # Filtering with a data transformation
{16, 25}
```

3.4 If Statements: Conditional Code Execution

- Skipped.

3.5 Simplifying If Statements

- Skipped

3.6 The Main Block: `if __name__ == '__main__'`

“Execute the following code if this module is being run, and ignore the following code if this module is being imported by another module”

Recap:

1. Doctest

- Nothing to be done: it will try the ‘code and output’ in t he Docstring!

```
if __name__ == '__main__':
    import doctest              # import the doctest library
    doctest.testmod(verbose=True)  # run the tests and display all results (pass or fail)
```

2. Pytest

- Define the function in the MAIN File.
- Write tests in a test_MAIN file.

1. Import MAIN.
2. Function tests start with prefix `test_` and ends with `_true` or `_false`
3. In test function, `assert MAIN.method()` / `assert not MAIN.method()`
4. Run `pytest`

```
# At the bottom of test_trues.pyif __name__ == '__main__':
import pytest
    pytest.main(['test_trues.py'])
```

3.7 Logical Statements with Multiple Quantifiers

REMEMBER: QUANTIFIERS are coded at the END! Prepositions FIRST.

```
>>> all({loves(a, b) for a in A for b in B})
False
```

```
>>> any({loves(a, b) for a in A for b in B})
True
```

Week 4

4.1 Specifying What a function should do

- A function specification 入咩出咩
    1. Preconditions: a set of predicates, where a valid input must satisfy all predicates.

    - parameter type
2. Postconditions: a set of predicates, where the return value must satisfy all predicates.
    - return type

- Function implementation is correct with respect to the function specification when: For all inputs that satisfy preconditions, the return value satisfy the postconditions.

- Simple specifications

```
def is_even(n: int) → bool:
"""Return whether n is even.
>>> is_even(1)
False
>> is_even(2)
True
"""
# Body omitted.
```

1. 'int' type annotation is the precondition
2. 'bool' type annotation and docstring description are the postconditions

- "Preconditions:"
    - In a docstring, written in python code.

```
def max_length(strings: set[str]) → int:
"""Return the maximum length of a string in the set of strings.
Preconditions:
- strings != set()
"""
return max({len(s)for sin strings})
```

4.2 Type Annotations Revisited

- **Homogenous** collection: every element has same data type

`set[T]`	A set whose elements all have type `T`. For example, `{'hi', 'bye'}` has type `set[str]`.
`list[T]`	A list whose elements all have type `T`. For example, `[1, 2, 3]` has type `list[int]`.
`dict[T1, T2]`	A dictionary whose keys all have type `T1` and whose associated values all have type `T2`. For example, `{'a': 1, 'b': 2, 'c': 3}` has type `dict[str, int]`.

- The More Specific Type Annotation Syntax

A precondition example:

```
def count_cancelled(flights: dict[str, list[int]]) -> int:
    """Return the number of cancelled flights for the given flight data.

    flights is a dictionary where each key is a flight ID,
    and whose corresponding value is a list of two numbers, where the first is
    the scheduled departure time and the second is the estimated departure ti
me.

    Precondition:
    - all({len(flights[k]) == 2 for k in flights})

    >>> count_cancelled({'AC110': [10, 12], 'AC321': [12, 19], 'AC999': [1, 1]})
    1
    """
    cancelled_flights = {id for id in flights
```

```
                    if get_status2(flights[id][0], flights[id][1]) == 'Cancelled'}
    return len(cancelled_flights)
```

- Any: a general type
  - import typing.
  - For example

    def identity(x: typing.Any) → typing.Any:
    - By using `from typing import Any`, we can directly use Any.
    - def identity(x: Any) → Any....

- Use direct import to access several functions from the module
- Use import from if you want to access only one or few.

4.3 Checking Function Specifications with `python_ta`

▼ Checking preconditions with assertions

```
def max_length(strings: set[str]) → int:
    """Return the maximum length of a string in the set of strings.

    Preconditions:
    - strings != set()
    """
    assert strings != set(), 'Precondition violated: max_length called on an e
mpty set.'
    return max(len(s) for s in strings)
```

```
>>> empty_set = set()
>>> max_length(empty_set)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 7, in max_length
AssertionError: Precondition violated: max_length called on an empty set.
```

▼ `python_ta`

1. Import code

```
from python_ta.contracts import check_contracts
```

2. Add:

@check_contract: decorator of the function, adding additional behaviour to the function

```
@check_contracts
def max_length(strings: set[str]) -> int:
"""Return the maximum length of a string in the set of strings.
Preconditions:
- strings != set()
"""
return max({len(s)for sin strings})
```

- What check_contract does:

1. Checks the precondition violations
2. Checks the type annotations (input and return)
3. Returns an `AssertionError` if something is wrong.

4.4 Testing Functions: `hypothesis`

- Unit test: cover all possible execution paths
- Property-based testing: tested with multiple inputs that `hypothesis` chooses for us
    - provides strategies to generate several values of a specific type of input
- In the test file,

1. Import given and strategies

```
from hypothesis import given# NEW
from hypothesis.strategies import integers # NEW
```

2. Add @given(x=integers())

```
# In file test_my_functions.py
from hypothesis import given
from hypothesis.strategies import integers

from my_functions import is_even


@given(x=integers())
def test_is_even_2x(x: int) -> None:
    """Test that is_even returns True when given a number of the form 2*x."""
    assert is_even(2 * x)


@given(x=integers())
def test_is_even_2x_plus_1(x: int) -> None:
    """Test that is_even returns False when given a number of the form 2*x +
1."""
    assert not is_even(2 * x + 1)


if __name__ == '__main__':
```

```
    import pytest
    pytest.main(['test_my_functions.py', '-v'])
```

i. integers() return a data type called strategy.

ii. given takes in form of (parameter = strategy).

iii. @given(x=integer()) automates process of running tests

3. Use `pytest` to run the test

```
if __name__ == '__main__':
import pytest
    pytest.main(['test_my_functions.py', '-v'])

# -v stands for verbose, giving information on failed test cases!
```

Example: list of integers

```
# In file test_my_functions.py
from hypothesis import given
from hypothesis.strategies import integers, lists  # NEW lists import

from my_functions import is_even, num_evens


@given(nums=lists(integers()), x=integers())  # NEW given call
def test_num_evens_one_more_even(nums: list[int], x: int) -> None:
    """Test num_evens when you add one more even element."""
    assert num_evens(nums + [2 * x]) == num_evens(nums) + 1
```

```
if __name__ == '__main__':
    import pytest
    pytest.main(['test_my_functions.py', '-v'])
```

4.5 Justifying Correctness (Beyond Using Test Cases)

- Same function specification with different implementations :)
- E.g.
- 1. (1 + 2 + 3 + .. + n)
- 2. n(n+1)//2

4.6 **Proofs and Programming I: Divisibility**

- Divisibility:
  - Definition

Definition. Let $n, d \in \mathbb{Z}$. We say that d divides n, or n is divisible by d, when there exists a $k \in \mathbb{Z}$ such that $n = dk$. In this case, we use the notation $d \mid n$ to represent "d divides n."

- Divisibility predicate: returns a boolean (4 | 10 is False)
- Horizontal Fraction bar: returns a number (10 / 4 is 2.5)

n = dk.

Proof Techniques:

1. A typical proof of an existential

a. Given statement to prove: there exist ... P(n)

   Proof.

   Let x = __

   Proof that P(n) is True,

2. A typical proof of a universal
   a. Given statement to prove: for all ... P(n)

      Proof.

      Let x in S.

      Proof that P(n) is True

3. A typical proof of an implication
   a. ASSUME that the hypothesis (if statement) is True
   b. Prove that the Conclusion is also True

- Order of introducing variables: In the order of the definition :)
  - x = 3
  - y = x # y = 3

4. A typical proof of a biconditional
   a. Given statement to prove: p ⇔ q.

      Proof. This proof is divided into two parts.

      Part 1: p ⇒ q: Assume p. Proof that q is True.

      Part 2: q ⇒ p: Assume q. Proof that p is True.

- Expressing divisibility in Python

```
def divides(d: int, n: int) -> bool:
"""Return whether d divides n."""
```

> **Theorem**. For all integers n and d, if $d \mid n$, then $-|n| \leq d \leq |n|$.
>
> This theorem tells us that for a given integer n, its set of possible divisors is $\{-|n|, -|n|+1, \ldots, |n|-1, |n|\}$ How do we represent such a set in Python? We can use the `range` data type:[15]

Therefore, we get

`possible_divisors = range(-abs(n), abs(n) + 1)` (you cannot have 5/6 to return a integer when 6 > 5!)

OK! Want to proof:

$$d \mid n \ : \ "\exists k \in \mathbb{Z},\ n = dk" \quad \text{where } n, d \in \mathbb{Z}$$

```
def divides(d: int, n: int) -> bool:
    """Return whether d divides n."""
    possible_divisors = range(- abs(n), abs(n) + 1)
    return any({n = dk for k in possible_divisors})
    # Remember: any / all only evaluates BOOLEANS!
    # WRONG: return any({k in possible_divisors if n = dk})
```

> **Theorem**. (Divisibility and Remainder Theorem) For all integers n and d, if $d \neq 0$, then d divides n if and only if $n \,\%\, d = 0$.

Theorem. (Quotient-Remainder Theorem) For all $n, d \in \mathbb{Z}$, if $d \neq 0$ there exist unique integers $q \in \mathbb{Z}$ and $r \in \mathbb{N}$ such that $n = qd + r$ and $0 \leq r < |d|$.

A Proof (Biconditional):

For all n in Z, 0 | n ⇒ n = 0 AND n = 0 ⇒ 0 | n.

Let n be in Z

1. (<=) Assume o | n, then let k in Z s.t. n = 0k..

   then n = 0.

2, (⇒) Assume n = 0, then let k = 0. Then n = 0k.

Therefore, 0 | n.

A fast implementation of divides:

```
def divides_v2(d: int, n: int) → bool:
"""Return whether d divides n."""if d == 0:
# This is the original definition.       possible_divisors = range(-abs(n), abs(n)
+ 1)
return any({n == k * d for k in possible_divisors})
else:
# This is a new but equivalent check.return n % d == 0
```

4.7 Proof and Programming (Prime Numbers)

IsPrime(p) :
p > 1 ∧ (∀d ∈ N, d ∣ p ⇒ d = 1 ∨ d = p)

where $p \in Z$

```
def is_prime(p: int) → bool:
    """Return whether p is prime."""
    possible_divisors = range(1, p + 1)
    return (
        p > 1 and
        all({d == 1 or d == p for d in possible_divisors if divides(d, p)})
    )
```

- A faster implementation

```
from math import floor, sqrt


def is_prime_v2(p: int) → bool:
    """Return whether p is prime."""
    possible_divisors = range(2, floor(sqrt(p)) + 1)
    return (
        p > 1 and
        all({not divides(d, p) for d in possible_divisors})
    )
```

Proof!

$$\forall p \in \mathbb{Z},\ IsPrime(p) \Leftrightarrow \big(p > 1 \wedge (\forall d \in \mathbb{N},\ 2 \leq d \leq \sqrt{p} \Rightarrow d \nmid p)\big)$$

Proof. Let $p \in \mathbb{Z}$.
Part 1: Prove that IF $IsPrime(p)$, THEN $p > 1$ and $\forall d \in \mathbb{N},\ 2 \leq d \leq \sqrt{p} \Rightarrow d \nmid p$.
...

Part 2: Prove that IF $p > 1$ and $\forall d \in \mathbb{N},\ 2 \leq d \leq \sqrt{p} \Rightarrow d \nmid p$, THEN $IsPrime(p)$.
...

IsPrime(p) :
p > 1 ∧ (∀d ∈ N, d ∣ p ⇒ d = 1 ∨ d = p)
where p ∈ Z

∀p ∈ Z, IsPrime(p) ⇔ (p > 1 ∧ (∀d ∈ N, 2 ≤ d ≤ √p ⇒ d ∤ p)

Proof: (CTRL Shift E for writing math symbols, \forall)

Let p ∈ Z.

(⇒)(Assume IsPrime(p).

a. p > 1 holds by the assumption
b. Prove ∀ d ∈ N, 2 ≤ d ≤ sqrt(p) ⇒ d not ∣ p.

   Let d ∈ N . Assume 2 ≤ d ≤ sqrt(p)

   Using the assumption from earlier, we obtain

   $d\ p => d = 1 \vee d = p$ as a new assumption

   we can take the contrapositive to get

   d =/= 1 ∧ d =/= p ⇒ d NOT ∣ p.

Since 2 ≤ d, d =/= 1

Since d ≤ sqrt(p) and p > 1, d =/= p

Using the implication, we get d NOT | p as desired.

(<=) Assume(p > 1 ∧ (∀d ∈ N, 2 ≤ d ≤ √p ⇒ d ∤ p)

a. Then p>1 by assumption.
b. Let d ∈ N. Assume d | p.
c. Using the first assumption, we obtain

- 2≤ d ≤ sqrt(p) ⇒ d NOT | p
- Taking the contapositive,
  - d | p ⇒ 2 > d ∨ sqrt(p) < d
  - Since we know d | p, then 2 > d ∨ sqrt(p) < d
    i. Assume 2 > d. Then d = 0 or d = 1.

    d =/= 0 since p > 1.

    d = 1 as desired.

    ii. Assume d > sqrt(p). Unfolding d | p, we some k ∈ Z where p = dk.

    Using the first assumption, we obtain

    2 ≤ k ≤ sqrt(p) ⇒ k NOT | p

    k | p ⇒ k < 2 OR k > sqrt(p)

    Taking the contraositive and only using k | p,

    we obtain k < 2 OR k > sqrt(p)

    k = p/d, k < p / sqrt(p) = sqrt(p).

    Therefore k < 2

    Since p and d are positive, then k = 1.

    Therefore d = p.

5.0 File Handling

Opening a File

```
with open('marriage_data.csv') as my_file:
    reader = csv.reader(file)
```

- **open:** a builtin function: looking in the same folder as the current python module.
- next(reader): to read just single line (reading the text consumes it)
- data = [row for row in reader] : to read the entire file
- csv: comma-separated values.
    - Each row is a list. Items are separated by commas.

```
ID,Civic Centre,Marriage Licenses Issued,Time Period
1657,ET,80,January 2011
1658,NY,136,January 2011
1659,SC,159,January 2011
```

```
>>> data = [row for row in reader]
>>> data[1]
['1657', 'ET', '80', 'January 2011']
>>> data[2]
['1658',' NY', '136', 'January 2011']

# Items are all in STRING format. Convert yourself using int(data[x][y]) etc...
```

```
with open('marriage_data.csv') as file:
    reader = csv.reader(file)
```

```
    headers = next(reader)
#heade['ID', 'Civic Centre', 'Marriage Licenses Issued', 'Time Period']
```

```
#Reading the text consumes it
>>> header = next(reader)
# Since the above line already read the header
# it's no longer read below:
>>> data = [row for row in reader]
>>> data[0]
['1657', 'ET', '80', 'January 2011']
# Since the above comprehension read the entire file,
# there is nothing left to read
>>> next(reader)
**Error occurs**
# Trying to read the whole file again gives nothing back
>>> data = [row for row in reader]
>>> data
[]
```

```
def read_csv_file(filename: str) → list:
    # Return the headers and data stored in a csv file with the given filename.
    with open(filename) as file:
        reader = csv.reader(file)
        return [next(reader), [row for row in reader]]
```

5.1 Tabular Data

2D Arrays!

Consider this dataset:

```
>>> import datetime
>>> marriage_data = [
```

```
...     [1657, 'ET', 80, datetime.date(2011, 1, 1)],
...     [1658, 'NY', 136, datetime.date(2011, 1, 1)],
...     [1659, 'SC', 159, datetime.date(2011, 1, 1)],
...     [1660, 'TO', 367, datetime.date(2011, 1, 1)],
...     [1661, 'ET', 109, datetime.date(2011, 2, 1)],
...     [1662, 'NY', 150, datetime.date(2011, 2, 1)],
...     [1663, 'SC', 154, datetime.date(2011, 2, 1)],
...     [1664, 'TO', 383, datetime.date(2011, 2, 1)]
... ]
>>> len(marriage_data)  # There are eight rows of data
8
>>> len(marriage_data[0])  # The first row has four elements
4
>>> [len(row) for row in marriage_data]  # Every row has four elements
[4, 4, 4, 4, 4, 4, 4, 4]
>>> marriage_data[0]
[1657, 'ET', 80, datetime.date(2011, 1, 1)]
>>> marriage_data[1]
[1658, 'NY', 136, datetime.date(2011, 1, 1)]
```

```
>>> marriage_data[0][0]
1657
>>> marriage_data[0][1]
'ET'
>>> marriage_data[0][2]
80
>>> marriage_data[0][3]
datetime.date(2011, 1, 1)
```

Some list comprehensions!

```
>>> {row[1] for row in marriage_data}
{'NY', 'TO', 'ET', 'SC'}
```

```
>>> [row for row in marriage_data if row[2] > 380]
[[1664, 'TO', 383, datetime.date(2011, 2, 1)]]
```

```
>>> [row for row in marriage_data if row[1] == 'TO']  # The 'TO' rows
[[1660, 'TO', 367, datetime.date(2011, 1, 1)], [1664, 'TO', 383, datetime.date(201
1, 2, 1)]]

>>> [row[2] for row in marriage_data if row[1] == 'TO']  # The 'TO' marriages i
ssued
[367, 383]

>>> issued_by_TO = [row[2] for row in marriage_data if row[1] == 'TO']
```

```
def average_licenses_issued(data: list[list], civic_centre: str) -> float:
    """Return the average number of marriage licenses issued by civic_centre i
n data.

    Return 0.0 if civic_centre does not appear in the given data.

    Preconditions:
      - all({len(row) == 4 for row in data})
      - data is in the format described in Section 5.1
    """
    issued_by_civic_centre = [row[2] for row in data if row[1] == civic_centre]

    if issued_by_civic_centre == []:
        return 0.0
    else:
        total = sum(issued_by_civic_centre)
        count = len(issued_by_civic_centre)

        return total / count
```

```
>>> average_licenses_issued(marriage_data, 'TO')
375.0
```

```
def average_licenses_by_centre(marriage_data: list[list]) -> dict[str, float]:
    """Return a mapping of the average number of marriage licenses issued at
each civic centre.

    In the returned mapping:
      - Each key is the name of a civic centre
      - Each corresponding value is the average number of marriage licenses is
sued at
        that centre.

    Preconditions:
      - marriage_data is in the format described in Section 5.1
    """
    names = {'TO', 'NY', 'ET', 'SC'}
    return {key: average_licenses_issued(marriage_data, key) for key in names}
```

5.2 Defining Our Own Data Types, Part 1

Defining a data class (INSTANCE ATTRIBUTES!) An object is an instance.

```
from dataclasses import dataclass


@dataclass
class Person:
    """A custom data type that represents data for a person.
    """
```

```
    given_name: str
    family_name: str
    age: int
    address: str
```

1. `from dataclasses import dataclass` is a Python import-from statement that lets us use `dataclass` below.
2. `@dataclass` is a Python *decorator*. We've already seen decorators for function definitions when using the `hypothesis` library for property-based testing. A decorator for a class definition works in the same way, acting as a modifier for our definition. In this case, `@dataclass` tells Python that the data type we're defining is a data class.
3. `class Person:` is the syntax for the start of a *class definition*. The name of the data class is `Person`.

   The rest of the code for the class is indented to put it inside of the class body.
4. The next line, `"""A custom data type...` is a docstring that describes the purpose of the data class.
5. Each remaining line (starting with `given_name: str`) defines a piece of data associated with the data class; each piece of data is called an **instance attribute** (often shortened to just **attribute**) of the data class.

   For each instance attribute, we write a name and a type annotation.

Create an object with attributes:

```
>>> david = Person('David', 'Liu', 100, '40 St. George Street')

#OR
# The below allows flexibility in the order of passing in argument
>>> david = Person(family_name='Liu', given_name='David', address='40 St.
George Street', age=100)

#OR
```

```
>>> david = Person(
...     family_name='Liu',
...     given_name='David',
...     address='40 St. George Street',
...     age=100
... )
```

Type the object:

```
>>> type(david)
<class Person>
```

Accessing instance attributes

```
>>> david
Person(given_name='David', family_name='Liu', age=100, address='40 St. Ge
orge Street')
```

Express in memory model

```
david   Person(
          given_name='David',
          family_name='Liu',
          age=100,
          address='40 St. George Street')
```

5.3 Defining Our Own Data Types, Part 2

Representation invariant: the constraints on the attributes! (a predicate describing how we represent values)

```
from dataclasses import dataclass
from python_ta.contracts import check_contracts


@check_contracts #must be above dataclass!
@dataclass
class Person:
    """A person with some basic demographic information.

    Representation Invariants:
      - self.age >= 0
    """
    given_name: str
    family_name: str
    age: int
    address: str
```

“Representatin Invariants” MUST BE CAPITALIZED.

@check_contract works as representation invariant is also a type of constraint ⇒ Return AssertionError if there is invalid attributes.

The Data Class Design Recipe

1. Write the class header. (@dataclass and class Name:)

2. Write the instance attributes for the data class. (age: int)

3. Write the data class docstring.

```
"""A data class representing a person.

    Instance Attributes:
      - given_name: the person's given name
      - family_name: the person's family name
      - age: the person's age
```

```
    - address: the person's address
    """
```

4. Write an example instance (optional).

```
    >>> david = Person(
    ...     'David',
    ...     'Liu',
    ...     40,
    ...     '40 St. George Street'
    ... )
    """
```

5. Document any additional representation invariants.

```
    Representation Invariants:
      - self.age >= 0
```

```
@dataclass
class Person:
    """A data class representing a person.

    Instance Attributes:
      - given_name: the person's given name
      - family_name: the person's family name
      - age: the person's age
      - address: the person's address

    Representation Invariants:
      - self.age >= 0

    >>> david = Person(
    ...     'David',
    ...     'Liu',
    ...     40,
```

```
...     '40 St. George Street'
... )
"""
given_name: str
family_name: str
age: int
address: str
```

An example:

```
from dataclasses import dataclass
import datetime


@dataclass
class MarriageData:
    """A record of the number of marriage licenses issued in a civic centre in a
given month.

    Instance Attributes:
      - id: a unique identifier for the record
      - civic_centre: the name of the civic centre
      - num_licenses: the number of licenses issued
      - month: the month these licenses were issued
    """
    id: int
    civic_centre: str
    num_licenses: int
    month: datetime.date
```

```
>>> marriage_data = [
...     MarriageData(1657, 'ET', 80, datetime.date(2011, 1, 1)),
```

```
...     MarriageData(1658, 'NY', 136, datetime.date(2011, 1, 1)),
...     MarriageData(1659, 'SC', 159, datetime.date(2011, 1, 1)),
...     MarriageData(1660, 'TO', 367, datetime.date(2011, 1, 1)),
...     MarriageData(1661, 'ET', 109, datetime.date(2011, 2, 1)),
...     MarriageData(1662, 'NY', 150, datetime.date(2011, 2, 1)),
...     MarriageData(1663, 'SC', 154, datetime.date(2011, 2, 1)),
...     MarriageData(1664, 'TO', 383, datetime.date(2011, 2, 1))
... ]
```

```
    issued_by_civic_centre = [
      row.num_licenses for row in data if row.civic_centre == civic_centre
    ]
# You can now use attributes instead of indexes :)
```

Summary: Why data classes?

1. Easier to remember

5.4 Repeated Execution: For Loops

Loop accumulator = ‘Sum’

Looping over dictionaries: it is looping over the KEY.

e.g. for item in menu

- item is the key
- menu[item] is the VALUE.

5.5 For Loop Variations

- Dark magic:

```
for char in s:
        if char in 'aeiouAEIOU':
            vowels_so_far = vowels_so_far + 1
# You can use 'char' in a string to check ;)

#Equivalent
return len([char for char in s if char in 'aeiouAEIOU'])
```

- Implementing Max

```
def my_max(numbers: list[int]) → int:
    """Return the maximum value of the numbers in numbers.

    Preconditions:
        - numbers != []

    >>> my_max([10, 20])
    20
    >>> my_max([-5, -4])
    -4
    """
    # ACCUMULATOR max_so_far: keep track of the maximum value
    # of the elements in numbers seen so far in the loop.
    max_so_far = numbers[0]

    for number in numbers:
        if number > max_so_far:
            max_so_far = number

    return max_so_far
```

- Existential Search

```
def starts_with_v2(strings: Iterable[str], char: str) → bool:
    """..."""
    # ACCUMULATOR starts_with_so_far: keep track of whether
    # any of the strings seen by the loop so far starts with char.
    starts_with_so_far = False

    for s in strings:
        if s[0] == char:
            starts_with_so_far = True

    return starts_with_so_far
```

- Universal Search

```
def all_starts_with_v2(strings: Iterable[str], char: str) → bool:
    """..."""
    # ACCUMULATOR starts_with_so_far: keep track of whether
    # all of the strings seen by the loop so far starts with char.
    starts_with_so_far = True

    for s in strings:
        if s[0] != char:
            starts_with_so_far = False

    return starts_with_so_far
```

5.6 Index-Based For Loops

Remember slicing: s[i:i+1] returns s[i] only!

The sequence of the index of numbers:

```
range(0, len(numbers)) #Remember len is exclusive :)
```

```
# If comparing char in string, then use len(numbers) - 1 to avoid out of bound error!
```

5.7 Nested For Loops

Cartesian Product

```
def product(set1: set, set2: set) -> set[tuple]:
    """Return the Cartesian product of set1 and set2.

    >>> result = product({10, 11}, {5, 6, 7})
    >>> result == {(10, 5), (10, 6), (10, 7), (11, 5), (11, 6), (11, 7)}
    True
    """
```

To append an element in a set... Use set.union(setA, setB)

```
def cartesian_product(set1: set, set2: set) -> set[tuple]:
    """Return the Cartesian product of set1 and set2.

    >>> result = cartesian_product({10, 11}, {5, 6, 7})
    >>> result == {(10, 5), (10, 6), (10, 7), (11, 5), (11, 6), (11, 7)}
    True
    """
    # ACCUMULATOR product_so_far: keep track of the tuples from the pairs
    # of elements visited so far.
    product_so_far = set()

    for x in set1:
        for y in set2:
            product_so_far = set.union(product_so_far, {(x, y)})
```

```
    return product_so_far
```

To append an element to a list, either use append, or use LIST + [x]

```
averages_so_far = averages_so_far + [course_average]
```

6.1 Variable Reassignment, Revisited

- Augmented assignment statements `+=`, `-=`, `*=`, `//=`, `%=`, `/=`, and `**=`

6.2 Objects and Object Mutation

An object contains: id, data-type, value

- **id** of an object is a unique `int` representation of the memory address of the object
- Object Mutation: is an operation that changes the **value** of an existing object.
- e.g. using list.append() rather than list1 + [num * num], much more efficient as the code is mutated, and no extra list objects are being created.
- After reassignments, the id of a variable is changed.
- e.g. `squares_so_far = squares_so_far + [16]` creates a *new list object* and assigns that to `squares_so_far`
-

```
>>> squares_so_far = [1, 4, 9]
>>> squares_so_far
```

```
[1, 4, 9]
>>> id(squares_so_far)
1920480441344

>>> squares_so_far = squares_so_far + [16]
>>> squares_so_far
[1, 4, 9, 16]
>>> id(squares_so_far)
1920484788736
```

6.3 Mutable Data Types

- Variable: the “names” / memory address referring to the values (objects)
- Object: the instance of data values we interact with

Below has NO RETURN value!!!

Mutating list:

- **list.append()**
- **list.insert(index, x) (x will become list[index])**
- **list.extend(collectible) # add it to the end of the original list, unpacks the list**

```
lst = [1, 2, 3]

lst.extend([4, 5])
print(lst)
# [1, 2, 3, 4, 5]
# [1, 2, 3, [4, 5]] using append
```

- list.pop(x) # delete the item with value x in list and returns x
- list[x] = ‘ ‘ # list index assignment (applies in the for-loop!)
- list += [x,y,z] # actually is mutation! (different from list = list + [x,y,z]

Mutating sets:

- set.add(x) #does not specify a position
- set.remove(x)
- NO: .intersection / .union

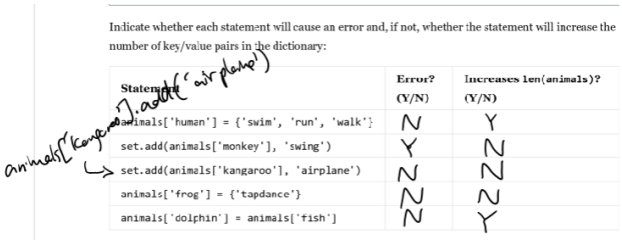
Mutating dictionaries

- dict['x'] = ' ' # dict KEY assignment to a value

```
>>> items = {'a': 1, 'b': 2}
>>> items['c'] = 3
>>> items
{'a': 1, 'b': 2, 'c': 3}

>>> items['a'] = 100
>>> items
{'a': 100, 'b': 2, 'c': 3}
```

```
def get_order_quantities(table_orders: dict[str, list[str]]) -> dict[str, int]:
    """Return a mapping from food item to the number of that item ordered.

    In the input dictionary table_orders:
        - Each key is the name of a person.
        - Each corresponding value is a list of the food items that person has ord
ered.
          Duplicates are allowed!

    In the returned dictionary:
        - Each key a a food item.
        - Each corresponding value is the number of times that food item was or
dered
          in table_orders, across all people.
```

```
    Use a for loop with a dictionary accumulator, and use mutating operations t
o update
    the accumulator in the loop body.

    >>> orders = {'David': ['Vegetarian stew', 'Poutine', 'Vegetarian stew'],\
                'Mario': ['Steak pie', 'Poutine', 'Vegetarian stew'],\
                'Jen': ['Steak pie', 'Steak pie']}
    >>> get_order_quantities(orders) == {'Vegetarian stew': 3, 'Poutine': 2, 'Ste
ak pie': 3}
    True
    """
    # TODO: complete this function body


    food_count = {}  # Accumulator dictionary to store food item counts

    for orders in table_orders:
        # Iterate over each food item in their list of orders
        for food in table_orders[orders]:
            # If the food item is already in the dictionary, increment its count
            if food in food_count:
                food_count[food] += 1
            else:
                # If the food item is not in the dictionary, add it with a count of 1
                food_count[food] = 1

    return food_count
```



Indicate whether each statement will cause an error and, if not, whether the statement will increase the number of key/value pairs in the dictionary:

Statement	Error? (Y/N)	Increases len(animals)? (Y/N)
animals['human'] = {'swim', 'run', 'walk'}	N	Y
set.add(animals['monkey'], 'swing')	Y	N
set.add(animals['kangaroo'], 'airplane')	N	N
animals['frog'] = {'tapdance'}	N	N
animals['dolphin'] = animals['fish']	N	Y

Mutating data classes

- Just assign to the attributes directly :)

```
>>> p = Person('David', 'Liu', 100, '40 St. George Street')
>>> p.age = 200
>>> p
Person(given_name='David', family_name='Liu', age=200, address='40 St. Ge
orge Street')
```

6.4 The Python Memory Model: Introduction

- We now know that it is more precise to say that evaluating any Python expression produces *an id of an object representing the value of the expression*.
- We cannot change the value of objects inside the objects — they are immutable. (in a list, set, dict, data class etc)

- If the expression is a literal, such as `176.4` or `'hello'`, Python creates an object of the appropriate type to hold the value.
- If the expression is a variable, Python looks up the variable. If the variable doesn't exist, a `NameError` is raised. If it does exist, the expression produces the id stored in that variable.
- If the expression is a binary operation, such as `+` or `%`, first Python evaluates the expression's two operands and applies the operator to the resulting values, creating a new object of the appropriate type to hold the resulting value. The expression produces the id of the new object.
- There are additional rules for other types of expression, but these will do for now.

Assignment statements. Second, we said earlier that an assignment statement is executed by first evaluating the right-hand side expression, and then storing it in the left-hand side variable. Here is a more precise version of what happens:

1. Evaluate the expression on the right-hand side, yielding the id of an object.
2. If the variable on the left-hand side doesn't already exist, create it.
3. Store the id from the expression on the right-hand side in the variable on the left-hand side.

6.5 Aliasing and "Mutation at a Distance"

-

```
>>> x = [1, 2, 3]
>>> y = [1, 2, 3]
>>> z = x
```

```
>>> id(x)
4401298824
>>> id(z)
4401298824
```

```
>>> x = [1, 2, 3]
>>> y = [1, 2, 3]
>>> z = x
>>> z[0] = -999
>>> x   # [-999, 2, 3]

>>> x = [1, 2, 3]
>>> y = [1, 2, 3]
>>> z = x
```

```
>>> y[0] = -999
>>> x   # [1, 2, 3]
```

- *Aliasing and loop variables*

  Reassignment for the loop variable:

  - it changes what the loop variable refers to, but does not change what the contents of the list 'numbers' refers to.
  - Remember! List reassignment through index is a mutation!

```
>>> numbers = [5, 6, 7]
>>> for number in numbers:
...     number = number + 1
...
>>> numbers
[5, 6, 7]



>>> numbers
[5, 6, 7]
>>> for i in range(0, len(numbers)):
...     numbers[i] = numbers[i] + 1
...
>>> numbers
[6, 7, 8]
```

- ***Two types of equality***
  - == is the value equality
  - `is` **: x is y**
    - the identity/reference equality

6.6 The Full Python Memory Model: Function Calls

Stack frames

- a special data type used by Python Interpreter to keep track of the functions and variables called in program.
- Function call stack: the entire collection of stack frames

1. Creates a new stack frame and add it to the call stack.
2. Evaluates the arguments in the function call, yielding the ids of objects (one per argument). Each of these ids is assigned to the corresponding parameter, as an entry in the new stack frame.
3. Executes the body of the function.
4. When a return statement is executed in the function body, the id of the returned object is saved and the stack frame for the function call is removed from the call stack.

Argument passing and aliasing

- def repeat(count, word), repeat(n, s) MEANS

```
n = count
s = word
```

Again, mutable vs immutable list operations!

```
def emphasize(words: list[str]) → None:
    """Add emphasis to the end of a list of words."""
    new_words = ['believe', 'me!']
    list.extend(words, new_words)
```

```
# In the Python console
>>> sentence = ['winter', 'is', 'coming']
>>> emphasize(sentence)
>>> sentence
['winter', 'is', 'coming', 'believe', 'me!']
```

```
def emphasize_v2(words: list[str]) -> None:
    """Add emphasis to the end of a list of words."""
    new_words = ['believe', 'me!']
    words = words + new_words


# In the Python console
>>> sentence = ['winter', 'is', 'coming']
>>> emphasize_v2(sentence)
>>> sentence
['winter', 'is', 'coming']
```

6.7 Testing Functions III: Testing Mutatoion

Property-based testing: No mutation

```
from hypothesis import given
from hypothesis.strategies import lists, integers


@given(lst=lists(integers()))
def test_squares_no_mutation_general(lst: list[int]) -> None:
    """Test that squares does not mutate the list it is given.
    """
    lst_copy = lst.copy()  # Create a copy of lst (not an alias!)
    squares(lst)
```

```
    assert lst == lst_copy
```

Property-based testing: Have mutation

```
@given(lst=lists(integers()))
def test_square_all_mutation_general(lst: list[int]) -> None:
    """Test that square_all mutates the list it is given correctly.
    """
    lst_copy = lst.copy()
    square_all(lst) # NOT result = square_all(lst)!

    assert all({lst[i] == lst_copy[i] ** 2 for i in range(0, len(lst))})
```

7.0 SUMMARY

7.1 Introduction to Number Theory

① GCD

- $\gcd(0,0) = 0$
- $\gcd(5,0) = 5$

② Euclidean ALG: $\gcd(a,b) = \gcd(b, a \% b)$ 24, 16 → 16, 8 → 8, 0

③ Congruent Modularity

a. Arithmetic

- $a \equiv b \pmod n$ (same remainder)
- $c \equiv d \pmod n$

Then $a+c \equiv b+d \pmod n$

$a-c \equiv b-d \pmod n$

$ac \equiv bd \pmod n$

b. Inverse

If $\gcd(a,n) = 1$

- $ab \equiv 1 \pmod 2$

c. Exponentiation

Any number.

$2^1 \equiv 2 \pmod 7 \Rightarrow 2^{1 \cdot ?} \equiv 2 \pmod 7$

$2^2 \equiv 4 \pmod 7 \Rightarrow 2^{2 \cdot ?} \equiv 4 \pmod 7$

$[\ 2^3 \equiv 1 \pmod 7 \Rightarrow 2^{3 \cdot ?} \equiv 1 \pmod 7$

Order of 2 mod 7 is 3.

(smallest power to get $a^k = 1 \bmod n$)

(cycle length; before it repeats)

Theorems and Algorithms

① GCD Characterization

$GCD(a,b) = ax + by$. (there exist)

② Quotient-Remainder Theorem

For any a and b,

$a = bq + r$

③ Divisibility of Linear Combinations

$d|a, d|b \Rightarrow d|ax+by$

Any x, y

④ Fermat's Little Theorem

- p is prime
- p does not divide a

$a^{p-1} \equiv 1 \pmod{p}$

⑤ Euler's Theorem

- $GCD(a,n) = 1$
- $a^{\phi(n)} \equiv 1 \pmod{n}$, where $\phi(n)$ = numbers that smaller than n. coprime with it.
- $\phi(pq) = (p-1)(q-1)$

Step 1: $a=3$, $n=10$. $GCD(3,10) = 1$

Step 2: $\phi(n) = 4$ (1, 3, 7, 9 are coprime with 10)

Step 3: $3^4 \equiv 1 \pmod{10}$

Divisibility, primality, and the greatest common divisor

- Divisibility:

$$d \mid n : \exists k \in \mathbb{Z},\ n = dk, \qquad \text{where } n, d \in \mathbb{Z}$$

- Prime numbers:

$$IsPrime(p) : p > 1 \wedge \left(\forall d \in \mathbb{N},\ d \mid p \Rightarrow d = 1 \vee d = p\right), \qquad \text{where } p \in \mathbb{Z}$$

- Common Divisor (d).

  For all integers d, n, m, d divides m and n.

  - gcd(-30, 18) = 6
  - gcd(5,0) = 5 # as 0 is divisble by all numbers! (n = dk)

- Coprime:
- 10 and 7 are **coprime** as gcd(10,7) = 1.

- Modular Equivalence
- 4 | 12-8
- 12 ≡ **8 (mod 4)**

Definition. Let $a, b, n \in \mathbb{Z}$, and assume $n \neq 0$. We say that a **is equivalent to** b **modulo** n when $n \mid a - b$. In this case, we write $a \equiv b \pmod{n}$.

Quotients, remainders, and modular arithmetic

- Quotient-Remainder Theorem
  - n = qd + r

- if a and b have the same remainder when divided by same number d, a - b can be divided completely by the same number d.
- The former one is a predicate that states the relationship between a,b,n
- % is a math operator that RETURNs an integer.
- EXAMPLE: 10 and 2 divided by 4 both has remainder 2.
- Then we say $10 \equiv \mathbf{2} \pmod 4$ is TRUE.
- i.e. $5 \mid 20 - 10$ is True.

$$a \equiv b \pmod n$$

$$\forall a, b, n \in \mathbb{Z},\ n \neq 0 \Rightarrow (a \equiv b \pmod n) \Leftrightarrow (a \,\%\, n = b \,\%\, n)$$

Theorem. Let $a, b, c, n \in \mathbb{Z}$ with $n \neq 0$. Then the following hold:

1. $a \equiv a \pmod n$.
2. If $a \equiv b \pmod n$ then $b \equiv a \pmod n$.
3. If $a \equiv b \pmod n$ and $b \equiv c \pmod n$ then $a \equiv c \pmod n$.

Theorem. Let $a, b, c, d, n \in \mathbb{Z}$ with $n \neq 0$. If $a \equiv c \pmod n$ and $b \equiv d \pmod n$, then the following hold:

1. $a + b \equiv c + d \pmod n$.
2. $a - b \equiv c - d \pmod n$.
3. $a \times b \equiv c \times d \pmod n$.

Comprehension Quiz (Week 7)

Review the definition of *divisibility* from lecture. Using this definition, select all of the **True** statements below.

- ✓ ☑ $\forall n \in \mathbb{Z}, \exists m \in \mathbb{Z},\ n \mid m$
- ☐ $\forall n \in \mathbb{Z},\ 0 \mid n$
- ☐ $\forall n \in \mathbb{Z},\ -1 \mid n$
- ✓ ☑ $\exists n \in \mathbb{Z},\ 0 \mid n$
- ☐ $\forall n, m \in \mathbb{Z},\ n \mid m$
- ✓ ☑ $\forall n \in \mathbb{Z},\ n \mid 0$

- Also, for all n in integers, -1 | n.

Review the definition of *prime* and select the correct representation in predicate logic.

- ○ $P(x) : \text{“}x > 1 \wedge (\forall d \in \mathbb{N}, d = 1 \vee d = x \Rightarrow d \mid x)\text{”}$, where $x \in \mathbb{Z}$
- ✓ ◉ $P(x) : \text{“}x > 1 \wedge (\forall d \in \mathbb{N}, d \mid x \Rightarrow d = 1 \vee d = x)\text{”}$, where $x \in \mathbb{Z}$
- ○ $P(x) : \text{“}x \geq 1 \wedge (\forall d \in \mathbb{N}, d \mid x \wedge (d = 1 \vee d = x))\text{”}$, where $x \in \mathbb{Z}$
- ○ $P(x) : \text{“}x \geq 1 \wedge (\forall d \in \mathbb{N}, (d \mid x \Rightarrow d = 1) \vee d = x)\text{”}$, where $x \in \mathbb{Z}$

Select the greatest common divisor of each of the following pairs of numbers.

gcd(-16, 28)	✓ 4
gcd(8, 90)	✓ 2
gcd(12, 13)	✓ 1
gcd(-14, -70)	✓ 14

4 1 / 1 point

Calculate $\gcd(189040, 820752)$. Enter the exact integer without any spaces or other characters.

Hint: Use the **math** module's **gcd** function in Python.

✓ 16

5 1 / 1 point

Match the following modulo to its equivalent definition using the divisibility predicate.

$20 \equiv 10 \pmod 5$

- $20 - 10 \mid 5$
- $10 \mid 20 - 5$
- $20 \,\%\, 5 = 10$
- ✓ $5 \mid 20 - 10$
- $5 \mid 20 + 10$

6 0.75 / 1 point

We know that $15 \equiv 1 \pmod 7$ and $20 \equiv 6 \pmod 7$. Select all the **True** statements below.

- ☐ $300 \equiv 30 \pmod 7$
- ✓ ☑ $35 \equiv 7 \pmod 7$
- ☐ $1500 \equiv 30 \pmod 7$
- ✓ ☑ $300 \equiv 6 \pmod 7$
- ✓ ☑ $5 \equiv 5 \pmod 7$
- ☐ $35 \equiv 5 \pmod 7$

7.2 Greatest Common Divisor GCD

Definitions MATHS

Saturday, 12 October 2024 7:53 PM

Divisibility: $d \mid n : \forall n, d, \exists k \in \mathbb{Z},\ n = dk$. (recall 0|0 is true, n|0 is true $n \in \mathbb{Z}$. $k = 0$ ☺)

IsPrime(p): $p > 1 \wedge (\forall d \in \mathbb{N},\ d \mid p \Rightarrow d = 1 \vee d = p)$

Faster version ↓

$\forall p \in \mathbb{Z}, \left((p > 1) \wedge (\forall d \in \mathbb{N},\ 2 \le d \le \sqrt{p} \Rightarrow d \nmid p) \right)$

GLD : $gcd(-30, 18) = 6$. $(\mathbb{Z} \times \mathbb{Z} \to \mathbb{N})$

Coprime : $gcd(10, 7) = 1$.

Quotient - remainder Theorem: $\forall n \in \mathbb{Z}, \forall d \in \mathbb{Z}^+, \exists q \in \mathbb{Z}, \exists r \in \mathbb{N}$

s.t. $n = qd + r$ AND $0 \le r < d$.

Modular equivalence : $\forall a, b, n \in \mathbb{Z}$,

$n \neq 0 \Rightarrow \left(a \equiv b \pmod n\right) \iff (a \,\%\, n = b \,\%\, n)$

$10 \equiv 2 \pmod 4$, as $10 \,\%\, 4 = 2 \,\%\, 4 = 2$. (remainder equiv.)

Properties; $a \equiv c \pmod n$ and $b \equiv d \pmod n$

Then $a + b \equiv c + d \pmod n$

$a - b \equiv c - d \pmod n$

$a \times b \equiv c \times d \pmod n$

Linear combinations and gcd.

- a is a linear combination of n, m

when $\exists\ p, q \in \mathbb{Z}$ s.t. $a = pm + qn$

- Divisibility of Linear Combinations

  If $d \mid m$ and $d \mid n$,

  $\Rightarrow$ d divides every linear combination of m and n.

  e.g. $42 = 3 \times 6 + 2 \times 12$ $\quad 3 \mid 6,\ 3 \mid 12.$

  Then $3 \mid 42$ as $3 \mid 6$ and $3 \mid 12$.

- GCD Characterization.

  $\gcd(m, n)$ is the smallest positive integer that is a linear combination of m and n.

Proving:

If $d \mid m$ and $d \mid n$,

then $d \mid \gcd(m, n)$.

$\gcd(m, n) = pm + qn$.

By GCD Characterization Theorem, $\exists\ p, q \in \mathbb{Z}$ s.t.

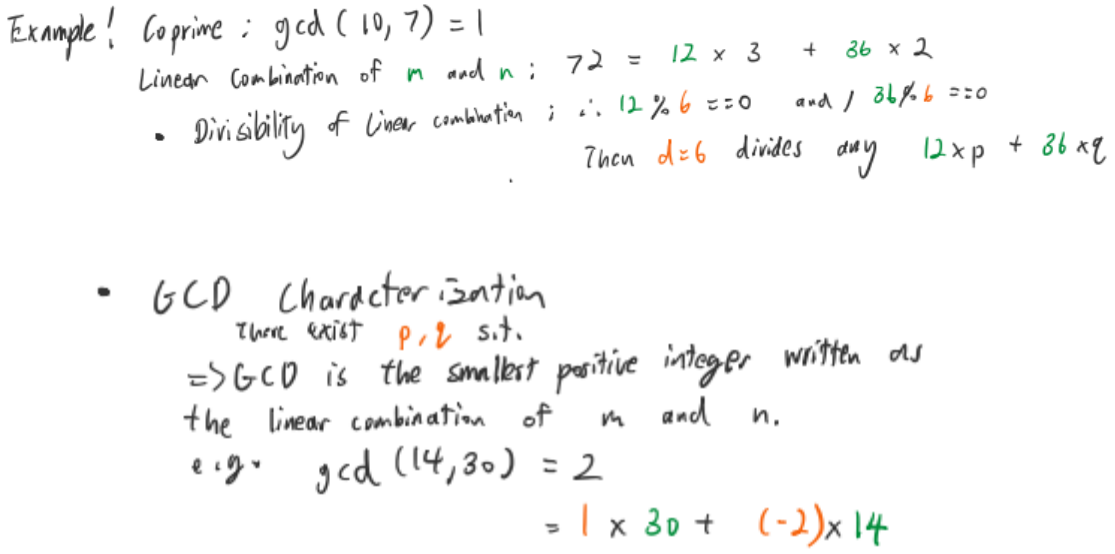
$\gcd(m, n) = pm + qn$

Assume $d \mid m$ and $d \mid n$.

Then d divides any linear combination of m, n.

$\Rightarrow d \mid pm + qn$

$\therefore d \mid \gcd(m, n)$



7.3 Proofs and Algorithms III: Computing the Greatest Common Divisor

- Divide

```
def divides(d: int, n: int) → bool:
    """Return whether d divides n."""
    if d == 0:
        return n == 0
    else:
        return n % d == 0
```

- Naive gcd(a,b)

```
def naive_gcd(m: int, n: int) → int:
    """Return the gcd of m and n."""
    if m == 0:
        return abs(n)
```

```
    elif n == 0:
        return abs(m)
    else:
        possible_divisors = range(1, min(abs(m), abs(n)) + 1)
        return max({d for d in possible_divisors if divides(d, m) and divides(d,
n)})
```

Parallel Assignment

```
while y != 0:
        r = x % y
        x, y = y, r
```

Loop invariant

- a property about loop variables that must be true at the start and the end

Euclidean Algorithm

$$\gcd(a, b) = \gcd(b, a \,\%\, b).^{2}$$

Given: non-negative integers `a` and `b`.

`gcd(a, b)`.

Example:

gcd(24,16)

⇒ gcd(16,8) (24%16 = 8)

⇒ gcd(8,0) (16%8 = 0)

```
def extended_euclidean_gcd(a: int, b: int) → int:
    """Return the gcd of a and b.

    Preconditions:
    - a >= 0
    - b >= 0
    """
    # Step 1: initialize x and y
    x, y = a, b

        # NEW: more loop variables
    px, qx = ..., ...
    py, qy = ..., ...

    while y != 0:  # Step 4: repeat Steps 2 and 3 until y is 0
        assert math.gcd(x, y) == math.gcd(a, b)  # (NEW) Loop invariant
                assert x == px * a + qx * b             # Loop invariant 2
        assert y == py * a + qy * b
        # Step 2: calculate the remainder of x divided by y
        r = x % y

        # Step 3: reassign x and y
        x, y = y, r

        # NEW: update the new loop variables
        px, qx, py, qy = ..., ..., ..., ...

    # Step 5: x now refers to the gcd of a and b
    return x, px, qx
```

▼ Explanation on why x and y are always linear combinations of a and b (rmb p,q can be negative)

In our implementation of the Euclidean Algorithm, each loop iteration makes the loop variables `x` and `y` smaller, while preserving the property `gcd(x, y) == gcd(a, b)`. The key mathematical insight is that **`x` and `y` are always linear combinations of `a` and `b`, at every loop iteration**! This might sound surprising, so let's double-check this informally:

- At the start, `x = a`, and `a` is certainly a combination of `a` and `b`: `a == a * 1 + b * 0`.
- At the start, `y = b`, and similarly `b == a * 0 + b * 1`.
- Inside the loop, `x` gets reassigned to `y`, and `y` gets reassigned to `r = x % y`.
    - If we assume `y` starts each loop iteration as a linear combination of `a` and `b`, then `x` will end each loop iteration as a linear combination as well.
    - But what about `x % y`? From the definition of remainder, we know that `x % y == x - q * y` for some integer `q` (the quotient `x // y` in Python syntax). So this tells us `x % y` is a linear combination of `x` and `y`, and if we assume that `x` and `y` are both linear combinations of `a` and `b`, we can conclude that `x % y` is also a linear combination of `a` and `b`.

- Keynotes:
    - each loop iteration makes the loop variables `x` and `y` smaller, while preserving the property `gcd(x, y) == gcd(a, b)`

extended_euclidean_gcd(100,13)

Iteration	x	px	qx	y	py	qy
0	100	1	0	13	0	1
1	13	0	1	9	1	-7
2	9	1	-7	4	-1	8
3	4	-1	8	1	3	-23
4	1	3	-23	0	-13	100

7.4 Modular Arithmetic and 7.5 Modular Exponentiation and Order

1. **Modular Equivalence**

- Definition: For integers a, b, and m, a≡b (mod m) if m divides a−b (i.e., a%m=b%m).

  aa

  bb

  mm

  a≡b (mod m)a \equiv b \ (\text{mod} \ m)

  mm

  a−ba - b

  a%m=b%ma \% m = b \% m

  - Example: 17≡5 (mod 6) since 17−5=12 is divisible by 6.

    17≡5 (mod 6)17 \equiv 5 \ (\text{mod} \ 6)

    17−5=1217 - 5 = 12

2. **Properties of Modular Arithmetic**

- **Addition**: If a≡b (mod m) and c≡d (mod m), then:

  a≡b (mod m)a \equiv b \ (\text{mod} \ m)

  c≡d (mod m)c \equiv d \ (\text{mod} \ m)

  - (a+c)≡(b+d) (mod m)(a + c) \equiv (b + d) \ (\text{mod} \ m)
    (a+c)≡(b+d) (mod m).
- **Subtraction**: (a−c)≡(b−d) (mod m).

  (a−c)≡(b−d) (mod m)(a - c) \equiv (b - d) \ (\text{mod} \ m)
- **Multiplication**: (a·c)≡(b·d) (mod m).

  (a·c)≡(b·d) (mod m)(a \cdot c) \equiv (b \cdot d) \ (\text{mod} \ m)

3. Modular Inverses

- For integers a and m, if gcd(a,m)=1, there exists an integer x such that:

  aa

  mm

  gcd(a,m)=1\gcd(a, m) = 1

  xx

  - a·x≡1 (mod m)a \cdot x \equiv 1 \ (\text{mod} \ m)a·x≡1 (mod m).
  - Example: The modular inverse of 3 modulo 7 is 5, since 3·5=15≡1 (mod 7).

    3·5=15≡1 (mod 7)3 \cdot 5 = 15 \equiv 1 \ (\text{mod} \ 7)

4. Modular Exponentiation

- **Exponentiation**: The powers of integers modulo m often repeat in cycles.

  mm

  - Example: 2k (mod 7) repeats every 3 steps: 2,4,1.

    2k (mod 7)2^k \ (\text{mod} \ 7)

    2,4,12, 4, 1
  - **Cycle Length**: The smallest integer k such that ak≡1 (mod m) is called the *order* of a modulo m.

kk

ak≡1 (mod m)a^k \equiv 1 \ (\text{mod} \ m)

aa

mm

5. Fermat’s Little Theorem

- If p is a prime and a is an integer such that gcd(a,p)=1, then:

  pp

  aa

  gcd(a,p)=1\gcd(a, p) = 1

  - ap−1≡1 (mod p)a^{p-1} \equiv 1 \ (\text{mod} \ p)ap−1≡1 (mod p).

6. Euler’s Theorem

- Generalizes Fermat’s Little Theorem:
  - For gcd(a,n)=1, aϕ(n)≡1 (mod n), where ϕ(n) is Euler's totient function (the number of integers less than n that are coprime to n).

    gcd(a,n)=1\gcd(a, n) = 1

    aϕ(n)≡1 (mod n)a^{\phi(n)} \equiv 1 \ (\text{mod} \ n)

    ϕ(n)\phi(n)

    nn

    nn

7. Examples

- **Addition**: 10+15 (mod 7)=(10%7+15%7)=3+1=4 (mod 7).

  10+15 (mod 7)=(10%7+15%7)=3+1=4 (mod 7)10 + 15 \ (\text{mod} \ 7) = (10 \% 7 + 15 \% 7) = 3 + 1 = 4 \ (\text{mod} \ 7)

- **Multiplication**: 4·3 (mod 5)=(4%5·3%5)=4·3=12 (mod 5)=2.

  4·3 (mod 5)=(4%5·3%5)=4·3=12 (mod 5)=24 \cdot 3 \ (\text{mod} \ 5) = (4 \% 5 \cdot 3 \% 5) = 4 \cdot 3 = 12 \ (\text{mod} \ 5) = 2

- **Fermat's Little Theorem**: For a=3 and p=7 (prime), 36≡1 (mod 7).

  a=3a = 3

  p=7p = 7

  36≡1 (mod 7)3^6 \equiv 1 \ (\text{mod} \ 7)

8.1 An Introduction to Cryptography

```
LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ '


def letter_to_num(c: str) -> int:
    """Return the number that corresponds to the given letter.

    Preconditions:
        - len(c) == 1 and c in LETTERS
    """
    return LETTERS.index(c)


def num_to_letter(n: int) -> str:
    """Return the letter that corresponds to the given number.

    Precondtions:
        - 0 <= n < len(LETTERS)
    """
    return LETTERS[n]
```

Encrypt and Decrypt in ASCII

```
def encrypt_ascii(k: int, plaintext: str) -> str:
    """Return the encrypted message using the Caesar cipher with key k.
```

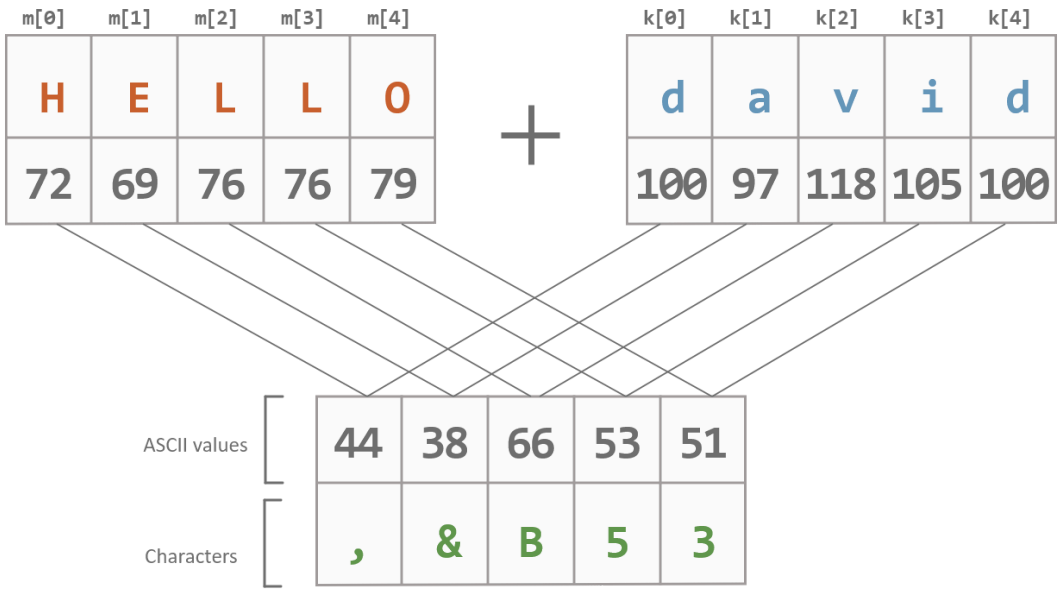
```
    Preconditions:
        - all({ord(c) < 128 for c in plaintext})
        - 1 <= k <= 127

    >>> encrypt_ascii(4, 'Good morning!')
    'Kssh$qsvrmrk%'
    """
    ciphertext = ''

    for letter in plaintext:
        ciphertext = ciphertext + chr((ord(letter) + k) % 128)

    return ciphertext


def decrypt_ascii(k: int, ciphertext: str) -> str:
    """Return the decrypted message using the Caesar cipher with key k.

    Preconditions:
        - all({ord(c) < 128 for c in ciphertext})
        - 1 <= k <= 127

    >>> decrypt_ascii(4, 'Kssh$qsvrmrk%')
    'Good morning!'
    """
    plaintext = ''

    for letter in ciphertext:
        plaintext += chr((ord(letter) - k) % 128)

    return plaintext
```

Correctness vs. Security

- While the Caesar cipher is correct (it encrypts and decrypts messages properly), it is **not secure** because attackers can easily guess the key by brute force due to a small key space.
- Vulnerable to **Brute-Force Exhaustive Key Search Attack**

8.2 The One-Time Pad and Perfect Secrecy

- One-time Pad
  - The shift is different for each character.



cryptText[i] = (m[i] + k[i]) % 128

decryptText[i] = (cryptText[i] - k[i]) % 128

  - Perfect Secrecy
    - Same CryptText with
    - HELLO + david
    - FUNNY + fQtGZ
    - So, is it HELLO or FUNNY? We don't know.

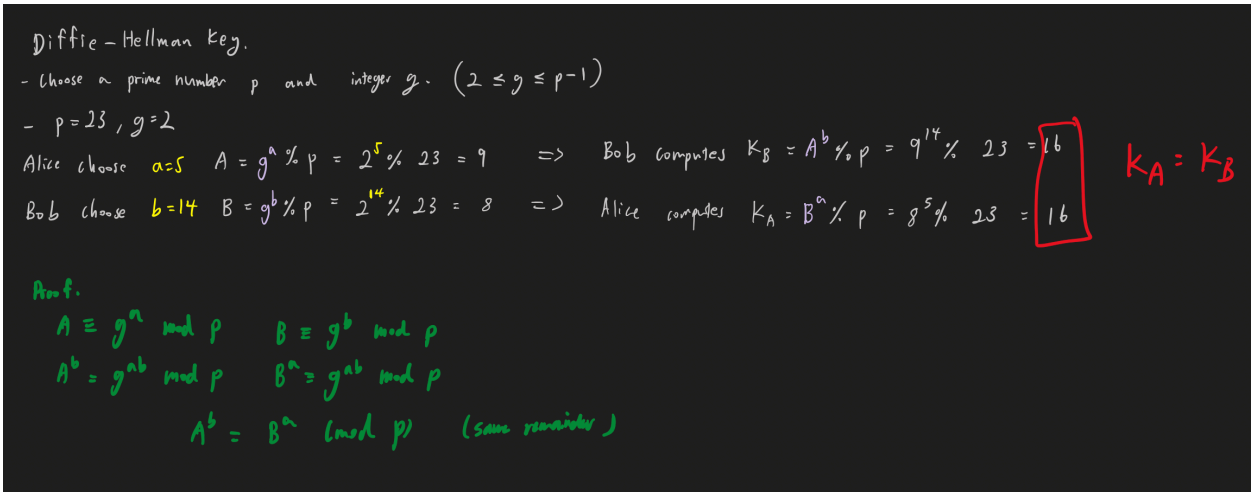
▼ Stream Ciphers: Use a smaller set of key and based on randomness, create a length of key same as the text and use one-time pad.

Stream Ciphers:

Stream ciphers are a type of **symmetric-key cryptosystem** that mimic the perfect secrecy of a one-time pad but use a **much smaller secret key**. The small key, combined with an algorithm, generates a continuous stream of pseudo-random characters for encryption.

- **Key Features**:
  - Small secret key shared between sender and receiver.
  - Generated characters, while not truly random, appear random, making decryption without the key computationally impossible.
  - Offers **"good enough" secrecy** instead of perfect secrecy.
- **Challenges**:
  - Security depends on the quality of the generating algorithm.
  - Poorly designed algorithms may reveal patterns or leak key information to attackers.

8.3 Computing Shared Secret Keys



Problem: In symmetric-key cryptosystems, establishing a shared secret key without meeting in person is challenging. The **Diffie-Hellman key exchange** solves this by allowing two parties to securely compute a shared key while communicating over a public channel.

Diffie-Hellman Key Exchange Steps:

1. **Public Setup**: Alice and Bob agree on a prime number p and a base g. pp gg
2. **Secret Numbers**: Alice and Bob each choose their own secret numbers (private).
3. **Exchange**: Alice sends Bob gamodp, and Bob sends Alice gbmodp.

   gamod pg^a \mod p

   gbmod pg^b \mod p
4. **Key Calculation**: Alice computes (gbmodp)a, and Bob computes (gamodp)b, both yielding the same secret key gabmodp.

   (gbmod p)a(g^b \mod p)^a

   (gamod p)b(g^a \mod p)^b

   gabmod pg^{ab} \mod p

Correctness:

Alice and Bob will always compute the same shared key because of the properties of modular arithmetic:

(gbmod p)a=(gamod p)b=gabmod p(g^b \mod p)^a = (g^a \mod p)^b = g^{ab} \mod p(gbmodp)a=(gamodp)b=gabmodp

Security:

An eavesdropper sees ppp, ggg, gamod pg^a \mod pgamodp, and gbmod pg^b \mod pgbmodp, but cannot compute the shared key without solving the **discrete logarithm problem**, which is computationally infeasible for large primes.

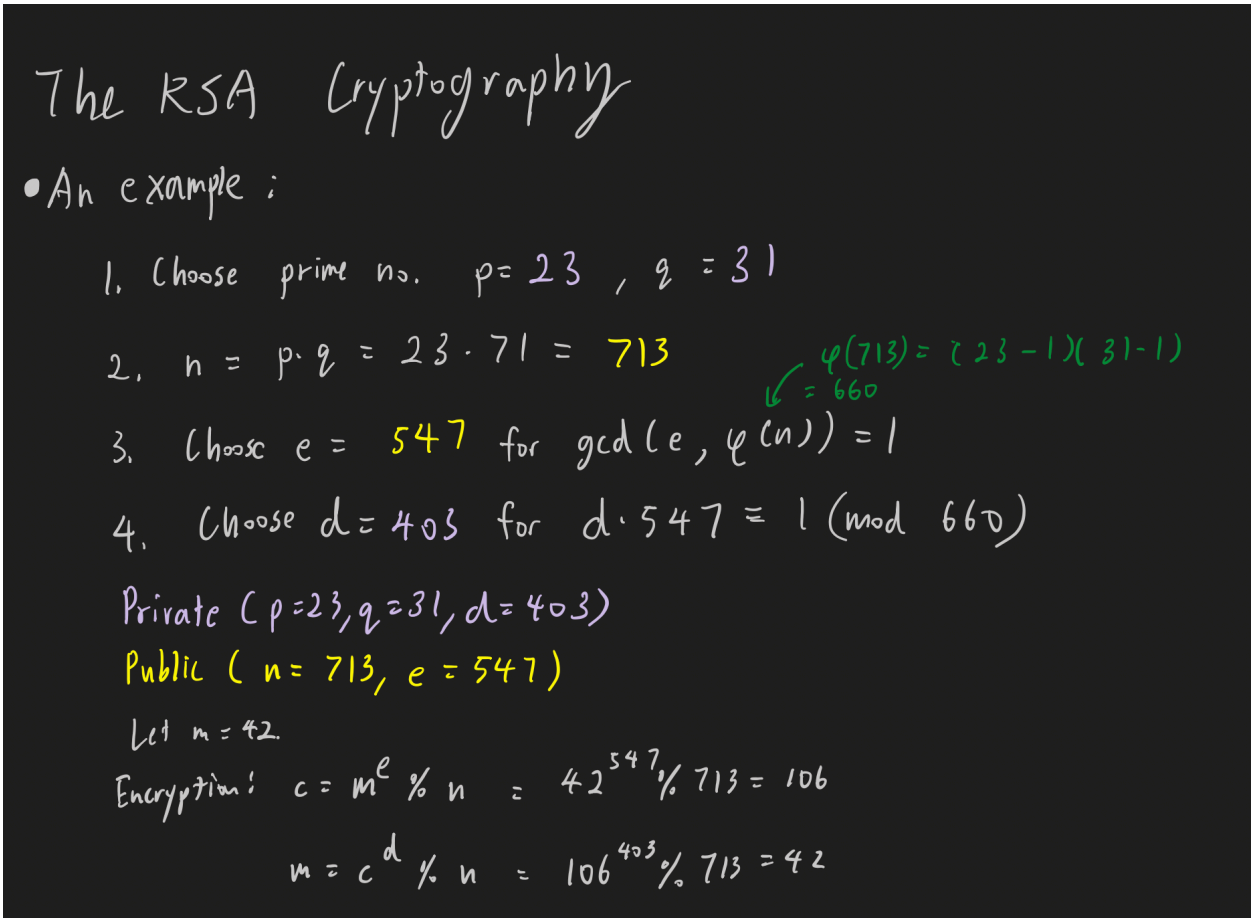
In practice, very large primes (600+ digits) are used, making the Diffie-Hellman exchange **computationally secure**.

8.4 The RSA Cryptosystem

RSA Cryptosystem is a **public-key cryptosystem**, where each person has two keys: a **public key** (shared with everyone) and a **private key** (kept secret). This system solves the issue of needing multiple shared keys between users in symmetric-key cryptosystems.

Key Features:

- **Public-Key Cryptography**:
  - Public keys are used for encryption, and private keys are used for decryption.
  - Unlike symmetric-key systems (where the same key is used for both encryption and decryption), RSA uses different keys, making it an **asymmetric system**.



RSA Security:

- RSA relies on the **hardness of factoring large integers**. While public keys are shared, private keys remain secure because it is computationally infeasible to factor large numbers (e.g., products of large primes).

RSA is widely used because it allows secure communication without the need for pre-shared keys, ensuring both correctness and security in encryption 【11†source】【10†source】.

8.5 Implementing RSA in Python

Key generation, decrypt and encrypt

```
def rsa_generate_key(p: int, q: int) -> \
        tuple[tuple[int, int, int], tuple[int, int]]:
    """Return an RSA key pair generated using primes p and q.

    The return value is a tuple containing two tuples:
      1. The first tuple is the private key, containing (p, q, d).
      2. The second tuple is the public key, containing (n, e).

    Preconditions:
        - p and q are prime
        - p != q
    """
    # Compute the product of p and q
    n = p * q

    # Choose e such that gcd(e, phi_n) == 1.
    phi_n = (p - 1) * (q - 1)

    # Since e is chosen randomly, we repeat the random choice
    # until e is coprime to phi_n.
    e = random.randint(2, phi_n - 1)
```

```
    while math.gcd(e, phi_n) != 1:
        e = random.randint(2, phi_n - 1)

    # Choose d such that e * d % phi_n = 1.
    # Notice that we're using our modular_inverse from our work in the last cha
pter!
    d = modular_inverse(e, phi_n)

    return ((p, q, d), (n, e))


def rsa_encrypt(public_key: tuple[int, int], plaintext: int) → int:
    """Encrypt the given plaintext using the recipient's public key.

    Preconditions:
        - public_key is a valid RSA public key (n, e)
        - 0 < plaintext < public_key[0]
    """
    n, e = public_key[0], public_key[1]

    encrypted = (plaintext ** e) % n

    return encrypted


def rsa_decrypt(private_key: tuple[int, int, int]  ciphertext: int) → int:
    """Decrypt the given ciphertext using the recipient's private key.

    Preconditions:
        - private_key is a valid RSA private key (p, q, d)
        - 0 < ciphertext < private_key[0] * private_key[1]
    """
    p, q, d = private_key[0], public_key[1], public_key[2]
    n = p * q

    decrypted = (ciphertext ** d) % n
```

```
    return decrypted


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({0 < ord(c) < public_key[0] for c in plaintext})
    """
    n, e = public_key

    encrypted = ''
    for letter in plaintext:
        # Note: we could have also used our rsa_encrypt function here instead
        encrypted = encrypted + chr((ord(letter) ** e) % n)

    return encrypted


def rsa_decrypt_text(private_key: tuple[int, int, int], ciphertext: str) -> str:
    """Decrypt the given ciphertext using the recipient's private key.

    Preconditions:
        - private_key is a valid RSA private key (p, q, d)
        - all({0 < ord(c) < private_key[0] * private_key[1] for c in ciphertext})
    """
    p, q, d = private_key
    n = p * q

    decrypted = ''
    for letter in ciphertext:
        # Note: we could have also used our rsa_decrypt function here instead
        decrypted = decrypted + chr((ord(letter) ** d) % n)
```

```
    return decrypted
```

8.6 Application: Securing Online Communications

Overview of HTTPS and TLS

- **HTTPS**: A combination of HTTP and TLS, HTTPS ensures secure communication over the Internet. The HTTPS icon (padlock) in web browsers indicates a secure connection.
- **HTTP**: Governs the format of data exchanged between the client (your computer) and the server.
- **TLS**: Governs the encryption of that data, ensuring that it remains private and cannot be easily intercepted.

Steps in the TLS Protocol

1. **Server Identity Verification**: The server sends a "proof of identity" (a digital certificate) to the client, which is verified without encryption.
2. **Key Exchange**: The client and server perform the Diffie-Hellman key exchange to establish a shared secret key, also without encryption.
3. **Encrypted Communication**: All subsequent data exchanged is encrypted using a symmetric-key cryptosystem.

Why Use Symmetric-Key Encryption?

- **Performance**: Symmetric-key encryption is faster than public-key encryption (like RSA), making it more suitable for the high data rates required by modern applications (e.g., video streaming).

Ensuring Server Authenticity

- **Digital Certificates**: The server sends a digital certificate containing its public key, signed by a certificate authority (CA). The client verifies this signature using the CA's public key.
- **Digital Signatures**: During the Diffie-Hellman key exchange, the server signs messages to ensure their authenticity, allowing the client to verify it is communicating with the correct server.

Vulnerabilities and Security Protocols

- **Logjam Attack**: A vulnerability discovered in 2015 showed that many servers used the same 512-bit prime numbers for the Diffie-Hellman algorithm, making them susceptible to attacks. As a result, more robust 2048-bit keys are now recommended.
- **Updates and Protocol Revisions**: Security protocols are constantly updated to address new vulnerabilities. Older versions of TLS (1.0 and 1.1) have been deprecated to ensure higher security standards.

Conclusion

As digital communication becomes increasingly integral to everyday life, understanding cryptography's role in securing these communications is crucial. TLS exemplifies how cryptography is used to protect data and ensure that users are communicating with the intended servers.

9.0 Running time analysis TRICKS

- We are talking about NUMBER of iterations :)
- range(0,10) is CONSTANT running time. range(0,n) has theta(n)
- Remember, running time of nested loops are MULTIPLIED.
- i < n BUT i *= 2, has theta(log2(n))
- x ** 2 for x in range(0,n) has theta(n)

- Why log? Because it is taking faster to reach n, than linear!
- List pop and insert at index i: THETA(n-i)a
- Sets and dictionaries:
  - Uses Constant-time lookup. insertion and removal of elements / key-value pairs. But its elements / key-value pair must all support hashing (list, set and dict DO NOT!). To use hash table, a set cannot contain another set.
  - We count all the sets, dict and dataclass operations all as ONE STEP.

```
for person in people: # n

    # the below count as ONE step
    if person.age > max_age_so_far:
        max_age_so_far = person.age

for course in new_grades:
    # The below count as ONE STEP!!!!!!
    if course in grades:
            grades[course].append(new_grades[course]) #THIS NOT TWO ST
EPS.
    else:
            grade[course] = [new_grades[course]] #THIS NOT TWO STEPS.
```

  - Data Class: benefit from constant-time operations (uses dictionary!) e.g. accessing david.age, mutating david.age = 99

```python
def while_function3(n: int) -> int:
    """Precondition: n >= 1"""
    count_so_far = 0
    i = 1

    while i < n:
        count_so_far += len({x ** 2 for x in range(0, n)})
        i *= 2

    return count_so_far
```

- $\Theta(1)$
- $\Theta(\log n)$
- $\Theta(n)$
- ✓ $\Theta(n \cdot \log_2(n))$
- $\Theta(n + \log_2(n))$
- $\Theta(n^2)$

- sum, max, min: linear running time theta n.
- len: theta(1)

MORE TRICKS.

While loops:

1. Express the value of the loop variable i in terms of k, after k iterations.
2. Plug in such value i such that i does not satisfy the while loop condition. Use ceiling function when necessary (almost always).

Exercise 1: Analysing running time of while loops

Your task here is to **analyse the running time of each of the following functions**. Recall the technique from lecture for calculating the number of iterations a while loop takes:

1. Find i_0, the value of the loop variable at the start of the first iteration. (You may need to change the notation depending on the loop variable name.)
2. Find a pattern for i_0, i_1, i_2, etc. based on the loop body, and a formula for a general i_k, the value of the loop variable after k loop iterations, assuming that many iterations occurs.
3. Find the *smallest* value of k that makes the loop condition False. This gives you the number of loop iterations.

   You'll need to use the floor/ceiling functions to get the correct exact number of iterations.

Exercise 2: Analysing nested loops

Remember, to analyse the running time of a nested loop:

1. First, determine an expression for the exact running time of the inner loop(s) for a fixed iteration of the outer loop. This may or may be the same for each iteration of the outer loop.
2. Then determine the total running time of the outer loop by adding up the steps of the inner loop(s) from Step 1. Note that if the number of steps of the inner loop(s) is the same for each iteration, you can simply multiply this number by the total number of iterations of the outer loop. Otherwise, you'll need to set up and simplify an expression involving summation notation (Σ).
3. Repeat Steps (1) and (2) if there is more than one level of nesting, starting from the innermost loop and working your way outwards. Your final result should depend *only* on the function input, not any loop variables.

You will also find the following formula helpful:

$$\forall n \in \mathbb{N},\ \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

Remember, when you have nested WHILE loop, there is a possibility to introduce summation, when your loop variable for outside while loop i is in an INNER FOR LOOP.

```python
def f5(n: int) -> None:
    """Precondition: n >= 4"""
    i = 4
    while i < n:
        j = 1
        while j < n:
            j = j * 3
        for k in range(0, i):
            print(k)
        i = i + 1
```

Rough Work

- Inner while loop
  - For a fixed iteration of the outer loop, the inner while loop takes $\lceil \log_3 n \rceil$ iterations
  - One step per iteration
  - So $\lceil \log_3 n \rceil$ steps for a fixed iteration of the outer loop.
- Inner for loop
  - For a fixed iteration of the outer loop, the inner for loop takes i iterations.
  - One step per iteration
  - So i steps for a fixed iteration of the outer loop
- Outer loop
  - $n - 4$ iterations
  - Steps per iteration: $1 + \lceil \log_3 n \rceil + i + 1$

Rough Analysis

So in total, the outer loop takes:

$$\begin{aligned}\sum_{i=4}^{n-1}(\lceil \log_3 n \rceil + i + 2) &= \sum_{i=4}^{n-1}(\lceil \log_3 n \rceil + 2) + \sum_{i=4}^{n-1} i \\ &= (n-4)(\lceil \log_3 n \rceil + 2) + \sum_{i=4}^{n-1} i\end{aligned}$$

Evaluating $\sum_{i=4}^{n-1} i$ is the hardest part. It is very close to a formula we know:

$$\sum_{i=0}^{m} i = \frac{m(m+1)}{2}$$

but it starts at 4 instead of 1. We can rewrite the summation by adding and subtracting the "missing" values:

$$\begin{aligned}\sum_{i=4}^{n-1} i &= \left(\sum_{i=4}^{n-1} i\right) + 1 + 2 + 3 - 1 - 2 - 3 \\ &= \left(\sum_{i=0}^{n-1} i\right) - 1 - 2 - 3 \\ &= \frac{(n-1)n}{2} - 6\end{aligned}$$

So putting this all together, the total running time is

$$RT_{f5} = (n-4)(\lceil \log_3 n \rceil + 2) + \frac{(n-1)n}{2} - 6 \in \Theta(n^2)$$

6.

```python
def f6(n: int) -> None:
    """Precondition: n >= 0"""
    i = 0
    while i < n:
        j = n
        while j > 0:
            for k in range(0, j):
                print(k)
            j = j - 1
        i = i + 4
```

Rough Work

- Innermost for loop
  - Iterates j times; depends on the inner while loop it is nested inside.
  - One step per iteration
  - So j steps for a fixed iteration of the outer loop.
- Inner while loop
  - Iterates n times
  - Steps per iteration: $\sum_{j=1}^{n}(j+1) = (\sum_{j=1}^{n} j) + n = \frac{n(n+1)}{2} + n$
- Outer loop
  - Iterates $\lceil \frac{n}{4} \rceil$ times
  - Steps per iteration: $2 + \frac{n(n+1)}{2} + n$

Rough Analysis

$$RT_{f6} = \left\lceil \frac{n}{4} \right\rceil \cdot (2 + \frac{n(n+1)}{2} + n) \in \Theta(n^3)$$

- Worst-Case Running Time Analysis

PUTTING IT TOGETHER

First, we proved that $WC_{\texttt{search}} \in \mathcal{O}(n)$.

Second, we found an input family (set of inputs, one for each $n \in \mathbb{N}$) whose running time is $\Theta(n)$. This told us that $WC_{\texttt{search}} \in \Omega(n)$.

Putting these two parts together, we can conclude that $WC_{\texttt{search}} \in \Theta(n)$.

1. Prove WCsearch in O(n) (find a tight upper bound BigO. tight means able to prove the same lower bound)
   - Just like normal runtime analysis. But use words like “AT MOST”.
   - WCsearch in O(...)

2. Find input family that the running time is theta(n), implying WCsearch is omega(n).
3. Putting both, we get WCsearch in theta(n)

Structure:

1. Upper bound
    - **Let n in N. Let nums1 and nums2 be ARBITRARY LISTS of length n.**
    - Each loop body / iterates **AT MOST** ? steps / times.
    - **At mos**t total steps.
    - Return takes **AT MOST** (1) steps.
    - Running time of **at most** ? steps.
    - WC in O(?)
2. Lower bound
    - **Let n in N. Take nums1 and nums2 = [1,2,...], [1,2,...]**
    - Each iteration = ?, Iterations = ?
    - total steps in theta(?) (NOT WC!)
    - **⇒ WC in OMEGA(?)**
3. Combine both to conclude a tight THETA bound.
    - Since WC in O(?) and WC in OMEGA(?)
    - Therefore WC in THETA(?)

10.1 An Introduction to Abstraction

- Abstraction: separation of two groups - Creators and Clients of the entity
- Interface: the public side of the entity. Part of the creator’s work that everyone can interact it.
  - Function: header and docstrings
  - dataclass: name, name and types of attributes, class docstring
  - module: the collection of interaces of functions and ata types (each python file we written, we create a module)
- The contract: creators have the responsibility to not affect the user when they make changes to the module

10.2 Defining Our Own Data Types, Part 3

```
class Person:
    """A custom data type that represents data for a person."""
    given_name: str
    family_name: str
    age: int
    address: str

    def __init__(self, given_name: str, family_name: str, age: int, address: str) →
None:
        """Initialize a new Person object."""
        self.given_name = given_name
        self.family_name = family_name
        self.age = age
        self.address = address
```

- This version, init is indented. Function being defined is a method for the Person class.
- “self” should always be the class that the initializer belongs to (e.g. Person object).

- We never have to pass a value for self. Python automatically sets it to the instance to be initialized.
- e.g. self.age = age (We are assigning the instance attribute to be the parameter!)
- What happens when we do david = Person(.....)?
    - Confusion: shouldn't init returns None. HUH?
        - Calling Person actually does 3 things.
            1. Create new Person object
            2. Call Person.__init__ with that Person object as 'self', with other arguments
            3. Return the new object (not directly from init)
        - Same for type conversion! e.g. int()
            1. Create int object
            2. Call int.__init__ with that int object as "self", with that int object as 'self', with other arguments
            3. Return the new int object

10.3 Defining Our Own Methods

```
class Person:
    """A custom data type that represents data for a person."""
    given_name: str
    family_name: str
    age: int
    address: str

    def __init__(self, given_name: str, family_name: str, age: int, address: str) →
None:
        """Initialize a new Person object."""
        self.given_name = given_name
```

```
        self.family_name = family_name
        self.age = age
        self.address = address

    def increase_age(self, years: int) -> None:
        """Add the given number of years to this person's age.

        >>> david = Person('David', 'Liu', 100, '40 St. George Street')
        >>> david.increase_age(10)
        >>> david.age
        110
        """
        self.age = self.age + years
```

```
class <ClassName>:
    """..."""
    <instance attributes/types omitted>

    def <method_name>(self, <param>: <type>, ...) -> <return type>:
        """Method docstring"""
        <statement> #e.g. self.xxx = self.xxx + yyy
        ...
```

10.4 Data Types, Abstract and Concrete

- Summary:
  - Abstract Data Types:
    - Mapping, Set, List, Iterable
  - Concrete Data Types:
    - dict, set, list...
  - We can use list to implement Mapping, Sets ADT...

- Key concern: time analysis.

10.5 Stacks

- First-in-last-out
  - Applications: function calling, check balanced paratheses
  - Implementing the Stack ADT using lists.
  - Idea of private instance attributes: No mentioning when call help. The variables can still be accessed. But implying the client code should NOT access this attribute.
  - Stack 1 Vs Stack 2: inserting and popping at the front of the list causes O(n), as it involves shifting of data.

```
class Stack1:
    """A last-in-first-out (LIFO) stack of items.

    Stores data in first-in, last-out order. When removing an item from the
    stack, the most recently-added item is the one that is removed.

    >>> s = Stack1()
    >>> s.is_empty()
    True
    >>> s.push('hello')
    >>> s.is_empty()
    False
    >>> s.push('goodbye')
    >>> s.pop()
    'goodbye'
    """
    # Private Instance Attributes:
    #   - _items: The items stored in the stack. The end of the list represents
    #     the top of the stack.
```

```
    _items: list

    def __init__(self) -> None:
        """Initialize a new empty stack.
        """
        self._items = []

    def is_empty(self) -> bool:
        """Return whether this stack contains no items.
        """
        return self._items == []

    def push(self, item: Any) -> None:
        """Add a new element to the top of this stack.
        """
        self._items.append(item)

    def pop(self) -> Any:
        """Remove and return the element at the top of this stack.

        Preconditions:
            - not self.is_empty()
        """
        return self._items.pop()
```

```
class Stack2:
    # Duplicated code from Stack1 omitted. Only push and pop are differen
t.

    def push(self, item: Any) -> None:
        """Add a new element to the top of this stack.
        """
        self._items.insert(0, item)

    def pop(self) -> Any:
```

```
        """Remove and return the element at the top of this stack.

        Preconditions:
            - not self.is_empty()
        """
        return self._items.pop(0)
```

10.6 Exceptions as a Part of the Public Interface

- Error message reveals private attributes. (e.g. calling pop on empty stack)
- Creating a custom exception!
-

```
class EmptyStackError(Exception):
    """Exception raised when calling pop on an empty stack."""

    def __str__(self) -> str:
        """Return a string representation of this error."""
        return 'pop may not be called on an empty stack'


class Stack1:
#OMITTED
    def pop(self) -> Any:
        """Remove and return the element at the top of this stack.

        Raise an EmptyStackError if this stack is empty.
        """
        if self.is_empty():
            raise EmptyStackError
        else:
```

```
            return self._items.pop()


>>> s = Stack()
>>> s.pop()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "...", line 60, in pop
    raise EmptyStackError
EmptyStackError: pop may not be called on an empty stack
# would just be EmptyStackError without the custom message.
```

- Pytest: Check if calling stack.pop raises the EmptyStackError

```
# Assuming our stack implementation is contained in a file stack.py.
from stack import Stack, EmptyStackError
import pytest


def test_empty_stack_error() -> None:
    """Test that popping from an empty stack raises an exception."""
    s = Stack()

    with pytest.raises(EmptyStackError):
        s.pop()
```

- try-except statement
  - code in ‘try’ is executed.
  - If error occurs and the exception has the type EmptyStackError,
    - run the code in except instead.

```
def second_from_top(s: Stack) -> Optional[str]:
    """Return the item that is second from the top of s.

    If there is no such item in the Stack, returns None.
    """
    try:
        hold1 = s.pop()
    except EmptyStackError:
        # In this case, s is empty. We can return None.
        return None

    try:
        hold2 = s.pop()
    except EmptyStackError:
        # In this case, s had only one element.
        # We restore s to its original state and return None.
        s.push(hold1)
        return None

    # If we reach this point, both of the previous s.pop() calls succeeded.
    # In this case, we restore s to its original state and return the second item.
    s.push(hold2)
    s.push(hold1)

    return hold2
```

10.7 Queues

-Assume using index 0 as front of queue.

Enqueue is O(1) (just add at the end)

Dequeue is O(n) (need to shift data)

```
class Queue:
    """A first-in-first-out (FIFO) queue of items.

    Stores data in a first-in, first-out order. When removing an item from the
    queue, the most recently-added item is the one that is removed.

    >>> q = Queue()
    >>> q.is_empty()
    True
    >>> q.enqueue('hello')
    >>> q.is_empty()
    False
    >>> q.enqueue('goodbye')
    >>> q.dequeue()
    'hello'
    >>> q.dequeue()
    'goodbye'
    >>> q.is_empty()
    True
    """
    # Private Instance Attributes:
    #   - _items: The items stored in this queue. The front of the list represents
    #             the front of the queue.
    _items: list

    def __init__(self) -> None:
        """Initialize a new empty queue."""
        self._items = []

    def is_empty(self) -> bool:
        """Return whether this queue contains no items.
        """
        return self._items == []

    def enqueue(self, item: Any) -> None:
```

```
        """Add <item> to the back of this queue.
        """
        self._items.append(item)

    def dequeue(self) -> Any:
        """Remove and return the item at the front of this queue.

        Raise an EmptyQueueError if this queue is empty.
        """
        if self.is_empty():
            raise EmptyQueueError
        else:
            return self._items.pop(0)
```

ADT Recipe:

```
class DataStruct:
    # Instance attributes:
    items: list
    # Private insstance attributes:
    _items: list

    #Methods
    def func(self, ...) -> ...:

    def funcErrorproof(self, ..) -> ...:
        if ...:
                raise ErrorExceptionClass
        else:
            return ...
```

```
class ErrorExceptionClass(exception):
    def __str__(self) -> str:
        """Return a string representation of this error."""
        return 'pop may not be called on an empty stack'
```

10.8 Priority Queues

- Data: a collection of items + their priorities
- Operations: determine whether the priority queue is empty, ENQUEUE an item with priority, DEQUEUE the highest priority item

```
from typing import Any

class PriorityQueue:
        """A collection items that are be removed in priority order.

    When removing an item from the queue, the highest-priority item is the one
    that is removed.

    >>> pq = PriorityQueue()
    >>> pq.is_empty()
    True
    >>> pq.enqueue(1, 'hello')
    >>> pq.is_empty()
    False
    >>> pq.enqueue(5, 'goodbye')
    >>> pq.enqueue(2, 'hi')
    >>> pq.dequeue()
    'goodbye'
    """

    _items: list[tuple[int, Any]]
```

```
    def __init__(self) -> None:
        """Initialize a new and empty priority queue."""
        self._items = []

    def is_empty(self) -> bool:
        """Return whether this priority queue contains no items.
        """
        return self._items == []

    def dequeue(self) -> Any:
        """Remove and return the item with the highest priority.

        Raise an EmptyPriorityQueueError when the priority queue is empty.
        """
        if self.is_empty():
            raise EmptyPriorityQueueError
        else:
            _priority, item = self._items.pop()
            return item

    def enqueue(self, priority: int, item: Any) -> None:
        """Add the given item with the given priority to this priority queue.
        """
        i = 0
        # if use <= priority, then the item would be added to that same priority nu
mber's last item.
        # if not, then would be the item right after the first one.
        while i < len(self._items) and self._items[i][0] < priority:
            # Loop invariant: all items in self._items[0:i]
            # have a lower priority than <priority>.
            i = i + 1

        self._items.insert(i, (priority, item))


class EmptyPriorityQueueError(Exception):
```

```
        def __str__(self) -> str:
            return 'You called dequeue on an empty priority queue'
```

10.9 Defining a Shared Public Interface with Inheritance

- An Abstract method: a method whose body raise NotImplementError
- An Abstract class: with at least one abstract method

Connections!

```
class Stack1(Stack):
    def ..
    def ..

sus = Stack1()

def f1(stack: Stack, item: Any) -> None:
    stack.push(item)

f1(sus, 4) # works because sus is an instance of Stack and Stack1
```

```
#Declare Stack1 and Stack2 as subclass of Stack (Superclass)
class Stack1(Stack):
    def __init__(self) -> None:
        """Initialize a new empty stack.
        """
        self._items = []

    def is_empty(self) -> bool:
        """..."""
        return self._items == []
```

```
    def push(self, item: Any) -> None:
        """..."""
        self._items.append(item)

    def pop(self) -> Any:
        """..."""
        return self._items.pop()


class Stack2(Stack):
    def __init__(self) -> None:
        """Initialize a new empty stack.
        """
        self._items = []

    def is_empty(self) -> bool:
        """..."""
        return self._items == []

    def push(self, item: Any) -> None:
        """..."""
        self._items.insert(0, item)

    def pop(self) -> Any:
        """..."""
        return self._items.pop(0)

#Polymorphic! Take inputs has different concrete data types
#below is OBJECT DOT NOTATION
def push_and_pop(stack: Stack, item: Any) -> None:
    """Push and pop the given item onto the stack stack."""
    stack.push(item)
    stack.pop()


>>> stack1 = Stack1()
```

```
>>> push_and_pop(stack1)
# Over here, s1.push(item) and s1.pop() is called (Stack1 methods)

 # This works! (Because stack1 is also an instance of Stack())
>>> stack2 = Stack2()
>>> push_and_pop(stack2)  # This also works!
# over here, s2.push(item) and s2.pop() is called (Stack2 methods)

# BELOW is CLASS DOT NOTATION (Not preferred!!)
# Does not work because Stack is not implemented
def push_and_pop_alt1(stack: Stack, item: Any) -> None:
    """Push and pop the given item onto the stack stack."""
    Stack.push(stack, item)
    Stack.pop(stack)


# Only works for Stack1 stacks.
def push_and_pop_alt2(stack: Stack, item: Any) -> None:
    """Push and pop the given item onto the stack stack."""
    Stack1.push(stack, item)
    Stack1.pop(stack)
```

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        """Calculate and return the area of the shape."""
        pass

    @abstractmethod
    def perimeter(self):
        """Calculate and return the perimeter of the shape."""
        pass
```

```
# Concrete subclass
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

# Concrete subclass
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius ** 2

    def perimeter(self):
        return 2 * 3.14159 * self.radius

# Usage
rect = Rectangle(4, 5)
print(f"Rectangle area: {rect.area()}")  # Output: Rectangle area: 20
print(f"Rectangle perimeter: {rect.perimeter()}")  # Output: Rectangle perimet
er: 18

circle = Circle(3)
print(f"Circle area: {circle.area()}")  # Output: Circle area: 28.27431
print(f"Circle perimeter: {circle.perimeter()}")  # Output: Circle perimeter: 18.8
4954
```

```
>>> type(my_stack) is Stack
False
>>> isinstance(my_stack, Stack)
True

#type(x) is t returns whether x is object of type t
#isinstance(x,t) returns whether. is an object of type t or any subclass of t
```

10.10 The object superclass

- every class implicitly inherits from built-in `object` class

```
class Stack:
  # is same as
class Stack(object): #automatically inherits object class

class Stack1(Stack): #Stack is superclass, Stack1 is subclass
```

Special methods in object

__init__(self, ...)

- called when object is initialized
- used to set up attributes or state for instance

```
class Donut:

# Inherits object.__init__ method, allowing us to create new
# donut instances.
>>> donut = Donut()
>>> type(donut)
<class ' __main__.Donut'>
```

__str__(self)

- returns a string representation of the object

```
>>> d = Donut()
>>> d.__str__()
'<__main__.Donut object at 0x7fc299d7b588>'

#Actually, when we call str(x), we call x.__str__().
# print function actually converts its arg into strings using their __str__ method
```

Method Overriding

- When we define our own __init__ for our own class, we have overridden the object.__init__ method.

```
class EmptyStackError(Exception):
    """Exception raised when calling pop on an empty stack."""

    def __str__(self) -> str:
        """Return a string representation of this error."""
        return 'pop may not be called on an empty stack'
```

```
class A:
        def m(self) -> int:
                return 1
class B(A):
        def m(self) -> int:
                return 100

>>> my_b = B()
>>> my_b.m()
100
```

1.9 Representations of Natural Numbers

Binary. Hex.

Floating Point

- Fractions:
  - 0.0111 in binary means

$$0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4} = 0/2 + 1/4 + 1/8 + 1/16 = 0.25 + 0.125 + 0.0625 = 0.4375$$

- 0.1 is infinite using binary.
- Lessons: Addition are not communicative. Avoid using type float variables as loop variables. Reducing number of float operations can produce a more accurate result.
- pytest.approx(expected) turns 0.3000004 into 0.3 (similar to math.isclose(1.233, 1.2330000001) which returns True)

> If you need to have a loop depend on a type **`float`** variable, best to reduce the number of type **`float`** operations involved to compute the variable.
>
> Compute the **k**th new value using e.g.,
> `b_so_far = 1.1 + k * 0.1`
> as there are only **two** **`float`** operations in the computation of the **`k+1`**st **`b_so_far`**.
>
> The first version using
> `b_so_far = b_so_far + 0.1`
> accumulates the result of **`k`** **`float`** operations in the computation of the **`k+1`**st **`b_so_far`**.

Important Definitions:

- **Variable reassignment:** only changes the immediate variable being reassigned, does not change any other variables or values
- Object: id, type, value (id and type never changes)
- Object Mutation: changes the value of an existing **object**
- Aliasing: 2 or more variables contain same id, refer to same object

Ch13 Linked List

- Optional[int] is equivalent to saying int or None.
- from __future__ import annotations: allow an attribute of a class to refer to itself

Templates:

```
#Traversal of Linked List
curr = my_linked_list._first   # 1. Initialize curr to the start of the list.
while curr is not None:        # 2. curr is None if we've reached the end of the li
st.
    ... curr.item ...          # 3. Do something with the current *element*, curr.item.
    curr = curr.next           # 4. "Increment" curr, assigning it to the next node.
```

```
# Get item using index
    def __getitem__(self, i: int) -> Any:
        """..."""
        # Version 1
        curr = self._first
```

```
        curr_index = 0

        while curr is not None:
            if curr_index == i:
                return curr.item

            curr = curr.next
            curr_index = curr_index + 1

        # If we've reached the end of the list and no item has been returned,
        # the given index is out of bounds.
        raise IndexError
```

```
from __future__ import annotations
from dataclasses import dataclass
from typing import Any, Optional
import math

@dataclass
class _Node:
    """A node in a linked list.

    Note that this is considered a "private class", one which is only meant
    to be used in this module by the LinkedList class, but not by client code.

    Instance Attributes:
      - item: The data stored in this node.
      - next: The next node in the list, if any.
    """
    item: Any
    next: Optional[_Node] = None  # By default, this node does not link to any o
ther node
```

```
class LinkedList:
    """A linked list implementation of the List ADT.
    """
    # Private Instance Attributes:
    #   - _first: The first node in this linked list, or None if this list is empty.
    _first: Optional[_Node]

    def __init__(self, items: Iterable = None) -> None:
        """Initialize an empty linked list.
        """
        self._first = None

    def to_list(self) -> list:
        """Return a built-in Python list containing the items of this linked list.

        The items in this linked list appear in the same order in the returned list.
        """
        items_so_far = []

        curr = self._first
        while curr is not None:
            items_so_far.append(curr.item)
            curr = curr.next

        return items_so_far

    def maximum(self) -> float:
        """Return the maximum element in this linked list.

        Preconditions:
            - every element in this linked list is a float
            - this linked list is not empty

        >>> linky = LinkedList()
        >>> node3 = _Node(30.0)
        >>> node2 = _Node(-20.5, node3)
```

```
        >>> node1 = _Node(10.1, node2)
        >>> linky._first = node1
        >>> linky.maximum()
        30.0
        """
        # Implementation note: as usual for compute maximums,
        # import the math module and initialize your accumulator
        # to -math.inf (negative infinity).
        max = -math.inf
        curr = self._first
        while curr is not None:
            if curr.item > max:
                max = curr.item
        return max

    def __getitem__(self, i: int) -> Any:
        """Return the item stored at index i in this linked list.

        Raise an IndexError if index i is out of bounds.

        Preconditions:
            - i >= 0
        """
        curr = self._first
        curr_index = 0

        while curr is not None:
            if curr_index == i:
                return curr.item
            curr = curr.next
            curr_index += 1

    def __append__ (self, item: Any) -> None:

        new_node = _Node(item)
```

```
        if self._first is None:
            self._first = new_node
        else:
            curr = self._first
            while curr is not None:
                curr = curr.next # This does not mutate
                #curr = new_node ###Would this mutate the list? (NO, the list is not
affected!)
            curr.next = new_node # This is mutated

    def insert(self, i: int, item: Any) -> None:
        """Insert the given item at index i in this list.

        Raise IndexError if i > len(self).
        Note that adding to the end of the list (i == len(self)) is okay.

        Preconditions:
            - i >= 0

        >>> lst = LinkedList([1, 2, 10, 200])
        >>> lst.insert(2, 300)
        >>> lst.to_list()
        [1, 2, 300, 10, 200]
        """

        new_node = _Node(item)
        if i == 0:
            new_node.next = self._first
            self._first = new_node

        else:
            curr = self._first
            curr.index = 0
            while curr is not None:
                if curr.index == i - 1:
                    curr.next, new_node.next = new_node, curr.next
```

```
def pop(self, i: int) -> Any:
    """Remove and return the item at index i.

    Raise IndexError if i >= len(self).

    Preconditions:
        - i >= 0

    >>> lst = LinkedList([1, 2, 10, 200])
    >>> lst.pop(1)
    2
    >>> lst.to_list()
    [1, 10, 200]
    >>> lst.pop(2)
    200
    >>> lst.pop(0)
    1
    >>> lst.to_list()
    [10]
    """
    if self._first is None:
        raise IndexError
    elif i == 0:
        item = self._first.item
        self._first = self._first.next
        return item
    else:
        curr = self._first
        curr_index = 0
        while not (curr is None or curr_index == i-1):
            curr = curr.next
            curr_index += 1

        if curr is None or curr.next is None:
            raise IndexError
```

```
        else:
            item = curr.next.item
            curr.next = curr.next.next
            return item
```

Ch14 Recursion

Function Design Recipe:

```
def f(nested_list: int | list) -> ...:
    if isinstance(nested_list, int):
        ...
    else:
        accumulator = ...

        for sublist in nested_list:
            rec_value = f(sublist)
            accumulator = ... accumulator ... rec_value ...

        return accumulator
```

```
def sum_nested_v1(nested_list: int | list) -> int:
    """Return the sum of the given nested list.

    This version uses a loop to accumulate the sum of the sublists.
    """
    if isinstance(nested_list, int):
        return nested_list
    else:
        sum_so_far = 0
```

```
        for sublist in nested_list:
            sum_so_far += sum_nested_v1(sublist)
        return sum_so_far


def sum_nested_v2(nested_list: int | list) -> int:
    """Return the sum of the given nested list.

    This version uses a comprehension and the built-in sum aggregation functi
on.
    """
    if isinstance(nested_list, int):
        return nested_list
    else:
        return sum(sum_nested_v2(sublist) for sublist in nested_list)
```

Recursive List

```
from __future__ import annotations
from typing import Any


class RecursiveList:
    """A recursive implementation of the List ADT.
    """
    # Private Instance Attributes:
    #   - _first: The first item in this list.
    #   - _rest: A list containing the items in this list that come after the first one.
    _first: Any
    _rest: RecursiveList
```

```
class RecursiveList:
    """A recursive implementation of the List ADT.
```

```
    Representation Invariants:
        - (self._first is None) == (self._rest is None)
    """
    # Private Instance Attributes:
    #   - _first: The first item in this list, or None if this list is empty.
    #   - _rest: A list containing the items in this list that come after the first one,
    #            or None if this list is empty.
    _first: Optional[Any]
    _rest: Optional[RecursiveList]

    def __init__(self, first: Optional[Any], rest: Optional[RecursiveList]) -> None:
        """Initialize a new recursive list."""
        self._first = first
        self._rest = rest
```

```
RecursiveList(
    1,
    RecursiveList(
        2,
        RecursiveList(
            3,
            RecursiveList(
                4,
                RecursiveList(
                    None,
                    None
                )
            )
        )
    )
)
```

```
# Built-in Python list
def sum_list(lst: list[int]) -> int:
```

```
    sum_so_far = 0
    for num in lst:
        sum_so_far += num
    return sum_so_far


# Linked list
class LinkedList:
    def sum(self) -> int:
        sum_so_far = 0
        curr = self._first

        while curr is not None:
            sum_so_far += curr.item
            curr = curr.next

        return sum_so_far


# Recursive list (Which is just linked list but adding them up using recursion)
class RecursiveList:
    def sum(self) -> int:
        if self._first is None:
            return 0
        else:
            return self._first + self._rest.sum()
```

▼ Recursive List Summary

Summary: Recursive Lists

Key Concepts:

1. **Recursive Definition of Lists**:

- A list can be recursively defined as:
  - The empty list `[]` is a list.
  - If `x` is a value and `r` is a list, then `+ r` is a list where `x` is the first element and `r` is the rest.
- Example: `[1, 2, 3, 4]` can be visualized recursively as:

```
[1, 2, 3, 4] ==
([1] + ([2] + ([3] + ([4] + []))))
```

2. **RecursiveList Class**:
   - A recursive implementation of a list using Python classes.
   - Each `RecursiveList` object contains:
     - `_first`: the first item or `None` if the list is empty.
     - `_rest`: another `RecursiveList` representing the rest of the list or `None` for an empty list.
   - Representation invariant: both `_first` and `_rest` are `None` for an empty list.
3. **Example Construction**:
   A RecursiveList for `[1, 2, 3, 4]`:

```
RecursiveList(1, RecursiveList(2, RecursiveList(3, RecursiveList(4, Rec
ursiveList(None, None)))))
```

4. **Recursive Functions**:
   - Recursive definitions naturally lead to recursive functions for operations on lists.
   - Example: Summing the elements of a list:

```
class RecursiveList:
    def sum(self) -> int:
        if self._first is None:  # Base case: empty list
            return 0
```

```
        else:  # Recursive case: non-empty list
            return self._first + self._rest.sum()
```

5. **Comparison to Other List Implementations**:
    - **Python Lists**: Use loops for traversal.
    - **Linked Lists**: Use iterative traversal via node links.
    - **Recursive Lists**: Use recursion to process elements without explicit loops.
6. **Relationship to Nodes**:
    - RecursiveList is conceptually similar to the `_Node` class in linked lists, with a recursive structure enabling different interpretations:
        - A `_Node` represents a single element with a link to the next node.
        - A `RecursiveList` represents the entire list, with the `_rest` representing the remaining list.

Takeaways:

- Recursive data structures like `RecursiveList` highlight the duality between node-based and recursive views.
- Recursive functions simplify list operations conceptually but may shift computational overhead to the interpreter.
- This recursive approach serves as a foundation for studying more complex structures like trees.

Ch15 Trees

Size: number of values in the tree (how many nodes?)

Leaf: a value with no subtree

Internal value: a value with at least one subtree

Height: Longest path from root to leaf

Children: all values directly connected under that value

Descendants: all values in the subtree under that value

Parent: immediately above that value (one parent)

```
from __future__ import annotations
from typing import Any, Optional


class Tree:
    """A recursive tree data structure.

    Representation Invariants:
        - self._root is not None or self._subtrees == []
    """
    # Private Instance Attributes:
    #   - _root:
    #       The item stored at this tree's root, or None if the tree is empty.
    #   - _subtrees:
    #       The list of subtrees of this tree. This attribute is empty when
    #       self._root is None (representing an empty tree). However, this attribute
    #       may be empty when self._root is not None, which represents a tree co
nsisting
    #       of just one item.
    _root: Optional[Any]
    _subtrees: list[Tree]

    def __init__(self, root: Optional[Any], subtrees: list[Tree]) -> None:
        """Initialize a new Tree with the given root value and subtrees.

        If root is None, the tree is empty.

        Preconditions:
            - root is not none or subtrees == []
        """
```

```
        self._root = root
        self._subtrees = subtrees

    def is_empty(self) -> bool:
        """Return whether this tree is empty.
        """
        return self._root is None



    def __len__(self):
        if self.is_empty():        # tree is empty
            return 0
        elif self._subtrees == []:  # tree is a single item
            return 1
        else:                      # tree has at least one subtree
            return 1 + sum(subtree.__len__() for subtree in self._subtrees)
```

```
#Tree code template
class Tree:
    def method(self) -> ...:
        if self.is_empty():        # tree is empty
            ...
        elif self._subtrees == []:  # tree is a single value
            ...
        else:                      # tree has at least one subtree
            ...
            for subtree in self._subtrees:
                ... subtree.method() ...
            ...
```

▼ Tree.__str__

The explanation below breaks down the provided examples and concepts step-by-step:

Tree.str and Representation Challenges

When creating a string representation of a tree, the main challenge lies in the **non-linear structure** of the tree. Unlike a list, which has a straightforward ordering of elements, a tree has:

1. A **root** value.
2. **Subtrees**, which themselves may contain further nested subtrees.

In a basic implementation of `Tree.__str__`:

- The **root** is printed first.
- The string representations of its subtrees are concatenated recursively.
- **Problem**: While this approach outputs all elements in the tree, the structure of the tree (i.e., the hierarchy and relationships between nodes) is lost.

Example:

```
t6 = Tree(6, [Tree(4, [Tree(1, []), Tree(2, []), Tree(3, [])]), Tree(5, [])])
print(t6)
```

Output:

```
6
4
1
2
3
5
```

Here, the hierarchy is unclear—it's not obvious which nodes are children or at what depth they reside.

Improved Representation with Indentation

To preserve the tree structure, **indentation** can be used to indicate the depth of each node:

- The root node has **no indentation**.
- Each level of the tree is indented by an additional **two spaces**.

Example:

```
6
  4
    1
    2
    3
  5
```

Implementation with Indentation

To implement this, a helper method `_str_indented` is defined with a `depth` **parameter**:

- **Depth** controls the number of spaces (indentation) added before a node's value.
- Recursive calls to `_str_indented` pass an incremented depth for subtrees.

Here's the implementation:

```
class Tree:
    def _str_indented(self, depth: int) -> str:
        """Return an indented string representation of this tree.

        The indentation level is specified by the <depth> parameter.
        """
        if self.is_empty():
            return ''
        else:
            # Add indentation proportional to the depth
            str_so_far = '  ' * depth + f'{self._root}\n'
            for subtree in self._subtrees:
                str_so_far += subtree._str_indented(depth + 1)  # Recurse with in
```

```
creased depth
            return str_so_far
```

The public `__str__` method calls `_str_indented` with an initial depth of 0:

```
class Tree:
    def __str__(self) -> str:
        return self._str_indented(0)
```

Traversal Orders

The order in which tree nodes are visited affects how the string is built. Two common traversal orders are:

1. **Preorder Traversal**:
    - Visit the root **before** traversing subtrees.
    - Root value appears at the start of its subtree representation.
    - Example:

      ```
      6
        4
          1
          2
          3
        5
      ```

      Implementation:

      ```
      def _str_indented(self, depth: int = 0) -> str:
          if self.is_empty():
              return ''
          else:
              str_so_far = '  ' * depth + f'{self._root}\n'
              for subtree in self._subtrees:
      ```

```
            str_so_far += subtree._str_indented(depth + 1)
        return str_so_far
```

2. **Postorder Traversal**:
    - Visit the root **after** traversing all subtrees.
    - Root value appears at the end of its subtree representation.
    - Example:

```
  4
    1
    2
    3
  5
6
```

    Implementation:

```
def _str_indented_postorder(self, depth: int = 0) -> str:
    if self.is_empty():
        return ''
    else:
        str_so_far = ''
        for subtree in self._subtrees:
            str_so_far += subtree._str_indented_postorder(depth + 1)
        str_so_far += '  ' * depth + f'{self._root}\n'
        return str_so_far
```

Key Concepts

1. **Recursive Helper Method**:
    - `_str_indented` uses recursion to process each subtree.
    - A `depth` parameter modifies behavior based on the tree level.
2. **Indentation**:

- `' ' * depth` adds spaces proportional to the depth.
- This creates a visual hierarchy in the string output.

3. **Traversal Orders**:
   - **Preorder**: Root → Subtrees.
   - **Postorder**: Subtrees → Root.
4. **Optional Parameters**:
   - Default values (e.g., `depth: int = 0`) simplify calling methods without explicitly passing arguments.

Practical Use

Using indentation in the tree's string representation helps visualize the hierarchy, making it easier to understand and debug complex tree structures. Traversal orders can be customized based on specific application needs.

15.3 Mutating Trees

- Insertion and Deletion
  - We have two .remove functions!
    - .remove(for subtrees) is a LIST remove function,
    - .remove(a value item) is a Tree REMOVE function

```
class Tree:
    def remove(self, item: Any) -> bool:
        """..."""
        if self.is_empty():
            return False
        elif self._root == item:
            self._delete_root()  # delete the root
            return True
        else:
```

```
        for subtree in self._subtrees:
            deleted = subtree.remove(item)
            if deleted and subtree.is_empty():
                # The item was deleted and the subtree is now empty.
                # We should remove the subtree from the list of subtrees.
                # Note that mutate a list while looping through it is
                # EXTREMELY DANGEROUS!
                # We are only doing it because we return immediately
                # afterwards, and so no more loop iterations occur.
                self._subtrees.remove(subtree)
                return True
            elif deleted:
                # The item was deleted, and the subtree is not empty.
                return True

        # If the loop doesn't return early, the item was not deleted from
        # any of the subtrees. In this case, the item does not appear
        # in this tree.
        return False



# If the node we want to delete has a subtree,
# we choose the last tree in the subtree list,
# and replace the root of the current tree with the chosen tree

def _delete_root(self) -> None:
    """..."""
    if self._subtrees == []:
        self._root = None
    else:
        # Get the last subtree in this tree.
        chosen_subtree = self._subtrees.pop()
```

```
            self._root = chosen_subtree._root
            self._subtrees.extend(chosen_subtree._subtrees)
```

```
from __future__ import annotations
from typing import Any, Optional


class Tree:
    """A recursive tree data structure.

    Note the relationship between this class and RecursiveList; the only major
    difference is that _rest has been replaced by _subtrees to handle multiple
    recursive sub-parts.

    Representation Invariants:
        - self._root is not None or self._subtrees == []
    """
    # Private Instance Attributes:
    #   - _root:
    #       The item stored at this tree's root, or None if the tree is empty.
    #   - _subtrees:
    #       The list of subtrees of this tree. This attribute is empty when
    #       self._root is None (representing an empty tree). However, this attribute
    #       may be empty when self._root is not None, which represents a tree co
nsisting
    #       of just one item.
    _root: Optional[Any]
    _subtrees: list[Tree]

    def __init__(self, root: Optional[Any], subtrees: list[Tree]) -> None:
        """Initialize a new Tree with the given root value and subtrees.

        If root is None, the tree is empty.

        Preconditions:
```

```
            - root is not none or subtrees == []
        """
        self._root = root
        self._subtrees = subtrees

    def is_empty(self) -> bool:
        """Return whether this tree is empty.

        >>> t1 = Tree(None, [])
        >>> t1.is_empty()
        True
        >>> t2 = Tree(3, [])
        >>> t2.is_empty()
        False
        """
        return self._root is None
```

15.4 Running-Time Analysis for Tree Operations

Step 1: find the number of non-recursive call in the function

Step 2: find the number of recursive calls

Step 3: Putting things together = multiply

15.6 Introduction to Binary Search Trees

- Multiset
  - an extension of the Set ADT, allows for duplicates
    - Data: unordered collection of values with duplicates
    - Operations:
      - get size
      - insert a value

- remove one occurrence of a specified value
- check membership in the multiset

```
class BinarySearchTree:
    """Binary Search Tree class.

    Representation Invariants:
      - (self._root is None) == (self._left is None)
      - (self._root is None) == (self._right is None)
      - (BST Property) if self._root is not None, then
          all items in self._left are <= self._root, and
          all items in self._right are >= self._root
    """
    # Private Instance Attributes:
    #   - _root:
    #       The item stored at the root of this tree, or None if this tree is empty.
    #   - _left:
    #       The left subtree, or None if this tree is empty.
    #   - _right:
    #       The right subtree, or None if this tree is empty.
    _root: Optional[Any]
    _left: Optional[BinarySearchTree]
    _right: Optional[BinarySearchTree]
    def __init__(self, root: Optional[Any]) -> None:
        """Initialize a new BST containing only the given root value.

        If <root> is None, initialize an empty BST.
        """
        if root is None:
            self._root = None
            self._left = None
            self._right = None
        else:
            self._root = root
```

```
        self._left = BinarySearchTree(None)   # self._left is an empty BST
        self._right = BinarySearchTree(None)  # self._right is an empty BST

    def is_empty(self) -> bool:
        """Return whether this BST is empty.
        """
        return self._root is None

    def __contains__(self, item: Any) -> bool:
        """Return whether <item> is in this BST.
        """
        if self.is_empty():
            return False
        elif item == self._root:
            return True
        elif item < self._root:
            return self._left.__contains__(item)
        else:
            return self._right.__contains__(item)

     def remove(self, item: Any) -> None:
        """Remove *one* occurrence of <item> from this BST.

        Do nothing if <item> is not in the BST.
        """
        if self.is_empty():
            pass
        elif self._root == item:
            self._remove_root()
        elif item < self._root:
            self._left.remove(item)
        else:
            self._right.remove(item)


        def _remove_root(self) -> None:
```

```
        """..."""
        if self._left.is_empty() and self._right.is_empty():
            self._root = None
            self._left = None
            self._right = None
        elif self._left.is_empty():
            # "Promote" the right subtree.
            self._root, self._left, self._right = \
                self._right._root, self._right._left, self._right._right
        elif self._right.is_empty():
            # "Promote" the left subtree.
            self._root, self._left, self._right = \
                self._left._root, self._left._left, self._left._right
        else:
            self._root = self._left._extract_max()


    def _extract_max(self) -> Any:
        """Remove and return the maximum item stored in this tree.

        Preconditions:
          - not self.is_empty()
        """
        if self._right.is_empty():
            max_item = self._root
            # Like remove_root, "promote" the left subtree.
            self._root, self._left, self._right = \
                self._left._root, self._left._left, self._left._right
            return max_item
        else:
            return self._right._extract_max()
```

Running time analysis for BST

- Searching, Inserting, Deletion = O(h) , where h is the height of the tree.

- Maximum number of nodes for a BST = 2^h - 1
- Minimum height: log2(n+1)
- Maximum height: n

e. Suppose we have a BST of height **111**. What's the maximum possible number of values in the BST? (*Don't* draw an example, but find a pattern from your previous answers.)

$1 + 2 + 2^2 + 2^3 + \ldots + 2^{110} = \sum_{i=0}^{111} 2^i = 2^{111} - 1$

geometric series

f. Suppose we have a BST of height h and that contains n values. Write down an inequality of the form $n \leq \ldots$ to relate n and h. (This is a generalization of your work so far.)

$n \leq 2^h - 1$

$n \leq 2^h - 1$

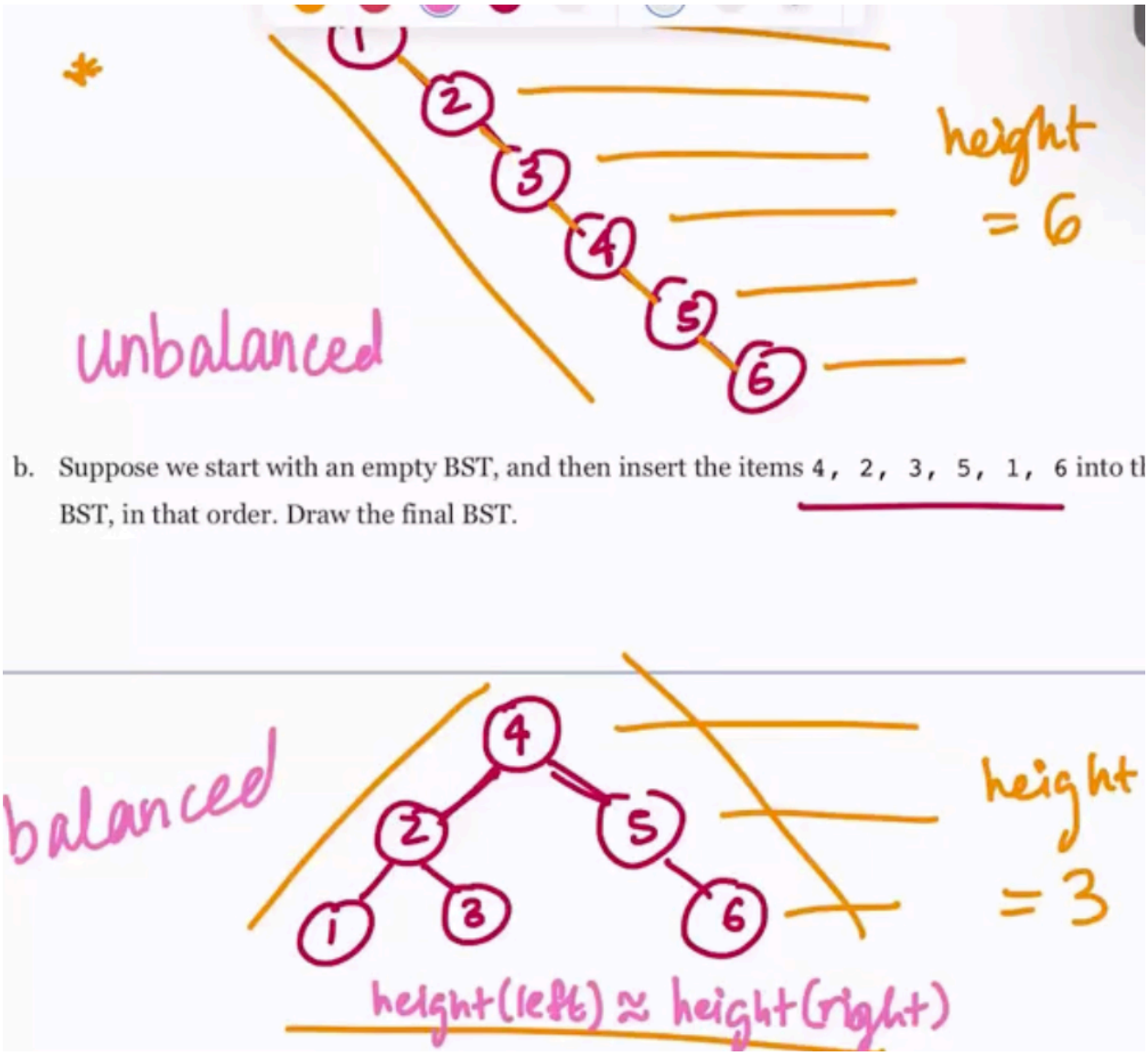
$n + 1 \leq 2^h$

$\log_2(n+1) \leq \log_2(2^h)$

$\log(n+1) \leq h$

g. Finally, take your inequality from the previous part and isolate h. This produces the answer to our original question: what is the minimum height of a BST with n values?

For a balanced BST, $\Theta(h) = \Theta(\log n)$



Ch16

Ch16.1 Abstract Syntax Tree

- “Abstract Syntax Tree”
  - tree-based structure of the program code, parsed by the interpreter

- The Expr class
  - We use different class to represent different kind of expression (e.g. int, string...) but we use inheritance to ensure they follow same fundamental interface.
-

```
class Expr:
    """An abstract class representing a Python expression.
    """
    def evaluate(self) -> Any:
        """Return the *value* of this expression.

        The returned value should be the result of how this expression would be
        evaluated by the Python interpreter.
        """
        raise NotImplementedError
```

`Num`: numerical literals

```
class Num(Expr):
        n: int | float
        def __init__(self, number: int | float) -> None:
                self.n = number
        def evaluate(self) -> Any:
                return self.n
```

- literals are like the **BASE CASE of AST.**

BinOp: arithmetic operations

- Consist 3 parts
  - left

- right
- operator

```
class BinOp(Expr):
    left: Expr
    op: str
    right: Expr

    def __init__(self, left: Expr, op: str, right: Expr) -> None:
        self.left = left
        self.op = op
        self.right = right

    def evaluate(self) -> Any:
        """Return the *value* of this expression.

        The returned value should the result of how this expression would be
        evaluated by the Python interpreter.

        >>> expr = BinOp(Num(10.5), '+', Num(30))
        >>> expr.evaluate()
        40.5
        """
        left_val = self.left.evaluate()
        right_val = self.right.evaluate()
                # self.left.evaluate can be BinOp.evalute, or Num.evaluate


        if self.op == '+':
            return left_val + right_val
        elif self.op == '*':
            return left_val * right_val
        else:
            # We shouldn't reach this branch because of our representation inv
ariant
```

```
            raise ValueError(f'Invalid operator {self.op}')

# 3 + 5.5 = BinOp(Num(3), '+', Num(5.5))

# ((3 + 5.5) * (0.5 + (15.2 * -13.3)))
BinOp(
    BinOp(Num(3), '+', Num(5.5)),
    '*',
    BinOp(
        Num(0.5),
        '+',
        BinOp(Num(15.2), '*', Num(-13.3)))
```

Ch16.2 Variables and the Variable Environment

From Ch16.1

- Dataclass: Expr
    - subclasses: Num, BinOp

- Variables and the `Name` class
    - x + 5.5 is expression but x is variable.
    - New subclass: Name
    - Use a dictionary to keep track of variable names and values
        - Variable environment = dict
        - Binding: Each key-value paiR

```
#Updated Expr, Num and BinOp
class Expr:
    def evaluate(self, env: dict[str, Any]) -> Any:
        """Evaluate this statement with the given environment.

        This should have the same effect as evaluating the statement by the
        real Python interpreter.
        """
        raise NotImplementedError


class Num(Expr):
    def evaluate(self, env: dict[str, Any]) -> Any:
        """..."""
        return self.n  # Simply return the value itself!

class BinOp(Expr):
    def evaluate(self, env: dict[str, Any]) -> Any:
        """..."""
        left_val = self.left.evaluate(env)
        right_val = self.right.evaluate(env)

        if self.op == '+':
            return left_val + right_val
        elif self.op == '*':
            return left_val * right_val
        else:
            raise ValueError(f'Invalid operator {self.op}')
```

```
class Name(Expr):
    def __init__(self, id_: str) -> None:
        self.id = id_

    def evaluate(self, env: dict[str, Any]) -> Any:
```

```
        """Return the *value* of this expression.

        The returned value should the result of how this expression would be
        evaluated by the Python interpreter.

        The name should be looked up in the `env` argument to this method.
        Raise a NameError if the name is not found.
        """
        if self.id in env:
            return env[self.id]
        else:
            raise NameError(f"name '{self.id}' is not defined")
>>> expr = Name('x')
>>> expr.evaluate({'x': 10})
10
>>> binop = BinOp(expr, '+', Num(5.5))
>>> binop.evaluate({'x': 100})
105.5
```

-

Ch16.3 From Expressions to Statements

1. The `Statement` Abstract Class
    - All expressions are statements, but not all statements are expressions.
    - Statement is a parent class of the expression class.

```
class Statement:
    """An abstract class representing a Python statement.

    We think of a Python statement as being a more general piece of code than
a
    single expression, and that can have some kind of "effect".
    """
```

```
    def evaluate(self, env: dict[str, Any]) -> Optional[Any]:
        """Evaluate this statement with the given environment.

        This should have the same effect as evaluating the statement by the
        real Python interpreter.

        Note that the return type here is Optional[Any]: evaluating a statement
        could produce a value (this is true for all expressions), but it might
        only have a *side effect* like mutating `env` or printing something.
        """
        raise NotImplementedError


class Expr(Statement):
    """An abstract class representing a Python expression.

    We've now modified this class to be a subclass of Statement.
    """
```

2. `Assign`: an assignment statement

```
class Assign(Statement):
    """An assignment statement (with a single target).

    Instance Attributes:
        - target: the variable name on the left-hand side of the equals sign
        - value: the expression on the right-hand side of the equals sign
    """
    target: str
    value: Expr

    def __init__(self, target: str, value: Expr) -> None:
        """Initialize a new Assign node."""
```

```
        self.target = target
        self.value = value

    def evaluate(self, env: dict[str, Any]) -> ...:
        """Evaluate this statement with the given environment.
        """
        env[self.target] = self.value.evaluate(env)
        # the variable assigned to the expression's value
```

- Example! y = x + 5.5

```
Assign('y', BinOp(Name('x'), '+', Num(5.5))
```

3. `Print`: displaying text to the user
    - Subclass of `Statement`

```
class Print(Statement):
    """A statement representing a call to the `print` function.

    Instance Attributes:
        - argument: The argument expression to the `print` function.
    """
    argument: Expr

    def __init__(self, argument: Expr) -> None:
        """Initialize a new Print node."""
        self.argument = argument

    def evaluate(self, env: dict[str, Any]) -> None:
        """Evaluate this statement.

        This evaluates the argument of the print call, and then actually
        prints it. Note that it doesn't return anything, since `print` doesn't
        return anything.
```

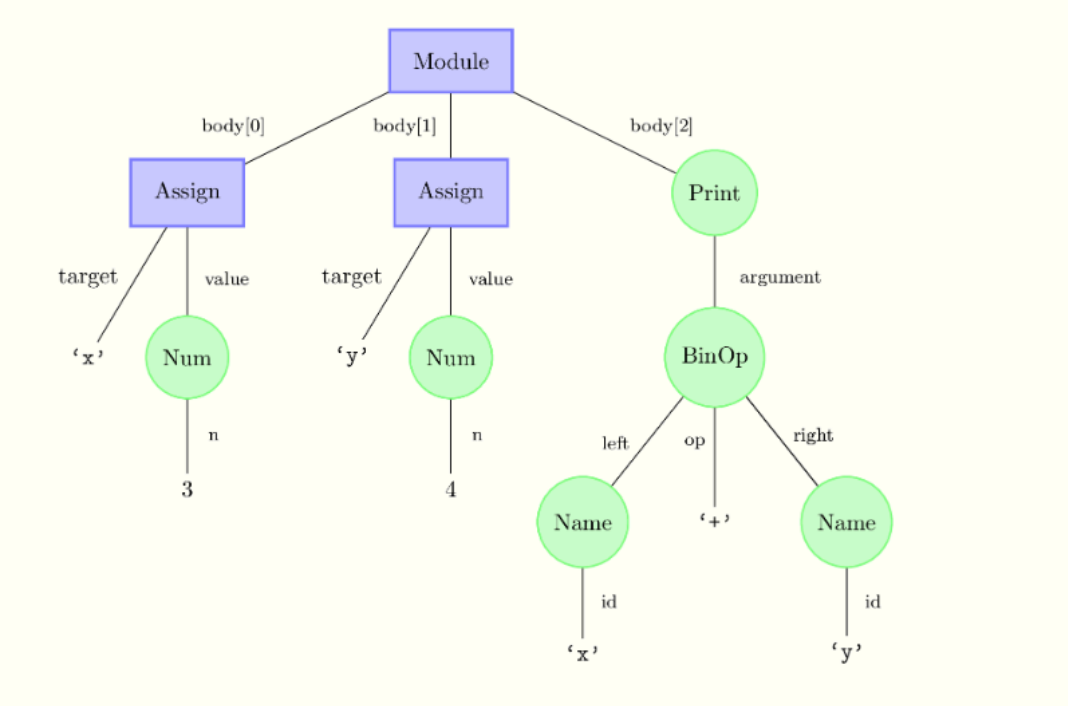
```
        """
        print(self.argument.evaluate(env))
```

4. `Module` : a sequence of statements
   - represents a full python program
   - To evaluate a module...
     1. Initialize an empty dictionary to represent env
     2. Iterate over each statement of module body, and evaluate it

```
class Module:
    """A class representing a full Python program.

    Instance Attributes:
        - body: A sequence of statements.
    """
    body: list[Statement]

    def __init__(self, body: list[Statement]) -> None:
        """Initialize a new module with the given body."""
        self.body = body

    def evaluate(self) -> None:
        """Evaluate this statement with the given environment.
        """
        env = {}
        for statement in self.body:
            statement.evaluate(env) # env can be mutated by assignments
```

```
# EG:
x = 3
y = 4
print(x + y)
```

```
# Can be represented as
Module([
    Assign('x', Num(3)),
    Assign('y', Num(4)),
    Print(BinOp(Name('x'), '+', Name('y')))
])
```



Control flow statements

1. If Statement

```
class If(Statement):

    test: Expr #True or False?
    body: list[Statement] # To be evaluated if True
    orelse: list[Statement] # To be evaluated if False
```

```
class ForRange(Statement):

    target: str
    start: Expr
    stop: Expr
    body: list[Statement]
```

Ch17 Graphs

17.1 Introduction to Graphs

- A graph: is a pair of sets (V, E)
  - V is set of objects
    - each element is called a vertex of graph, V itself is set of vertices of the graph
  - E is set of pairs of objects
    - v1 and v2 in V and v1 =/= v2 is called an edge
  - Adjacent / Neighbors: only one edge between them
  - Degree: number of neighbours
  - Path: sequence of distinct vertices (each consecutive pair of vertices are adjacent) (e.g. A to D: A, B, C, D) (can have 0 length paths)
  - Length: A to D, has length 3 (3 edges)
  - Connected: u, v are connected if exist a path between them

17.2 Some Properties of Graphs

- Maximum number of edges in a graph
  - 1st universal proof:

    - max. no of edges a graph can have

Example. Prove that for all graphs $G = (V, E)$, $|E| \leq \frac{|V|(|V|-1)}{2}$.

- Transitivity of connectedness.
    - **Example.** Let $G = (V, E)$ be a graph, and let $u, v, w \in V$. If v is connected to both u and w, then u and w are connected.[2]

- Proof by contradiction
    - **Example.** Prove that for all graphs $G = (V, E)$, if $|V| \geq 2$ then there exist two vertices in V that have the same degree.[3]

17.3 Representing Graphs in Python

- Vertex
    - item: any value
    - neighbours: a set of Vertex

```
from __future__ import annotations
from typing import Any
```

```
class _Vertex:
    """A vertex in a graph.

    Instance Attributes:
        - item: The data stored in this vertex.
        - neighbours: The vertices that are adjacent to this vertex.
    """
    item: Any
    neighbours: set[_Vertex]

    def __init__(self, item: Any, neighbours: set[_Vertex]) -> None:
        """Initialize a new vertex with the given item and neighbours."""
        self.item = item
        self.neighbours = neighbours


>>> v1 = _Vertex('a', set())
>>> v2 = _Vertex('b', set())
>>> v3 = _Vertex('c', set())
>>> v1.neighbours = {v2, v3}
>>> v2.neighbours = {v1, v3}
>>> v3.neighbours = {v1, v2}
```

- Graph
    - _vertices: a dictionary that maps the item value to the Vertex object

```
class Graph:
    """A graph.

    Representation Invariants:
    - all(item == self._vertices[item].item for item in self._vertices)
    """
    # Private Instance Attributes:
```

```
    #     - _vertices: A collection of the vertices contained in this graph.
    #                  Maps item to _Vertex instance.
    _vertices: dict[Any, _Vertex]

    def __init__(self) -> None:
        """Initialize an empty graph (no vertices or edges)."""
        self._vertices = {}

    def add_vertex(self, item: Any) -> None:
        """Add a vertex with the given item to this graph.

        The new vertex is not adjacent to any other vertices.

        Preconditions:
            - item not in self._vertices
        """
        self._vertices[item] = _Vertex(item, set())

    def add_edge(self, item1: Any, item2: Any) -> None:
        """Add an edge between the two vertices with the given items in this gra
ph.

        Raise a ValueError if item1 or item2 do not appear as vertices in this grap
h.

        Preconditions:
            - item1 != item2
        """
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            v2 = self._vertices[item2]

            # Add the new edge
            v1.neighbours.add(v2)
            v2.neighbours.add(v1)
        else:
```

```
            # We didn't find an existing vertex for both items.
            raise ValueError

    def adjacent(self, item1: Any, item2: Any) -> bool:
        """Return whether item1 and item2 are adjacent vertices in this graph.

        Return False if item1 or item2 do not appear as vertices in this graph.
        """
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            return any(v2.item == item2 for v2 in v1.neighbours)
        else:
            # We didn't find an existing vertex for both items.
            return False

    def get_neighbours(self, item: Any) -> set:
        """Return a set of the neighbours of the given item.

        Note that the *items* are returned, not the _Vertex objects themselves.

        Raise a ValueError if item does not appear as a vertex in this graph.
        """
        if item in self._vertices:
            v = self._vertices[item]
            return {neighbour.item for neighbour in v.neighbours}
        else:
            raise ValueError
```

17.4 Connectivity and Recursive Graph Traversal

```
    def connected(self, item1: Any, item2: Any) -> bool:
        """..."""
        if item1 in self._vertices and item2 in self._vertices:
```

```
            v1 = self._vertices[item1]
            return v1.check_connected(item2)
        else:
            return False

    def check_connected(self, target_item: Any, visited: set[_Vertex]) -> bool:
        """Return whether this vertex is connected to a vertex corresponding to t
he target_item,
        WITHOUT using any of the vertices in visited.

        Preconditions:
            - self not in visited
        """
        if self.item == target_item:
            # Our base case: the target_item is the current vertex
            return True
        else:
            visited.add(self)        # Add self to the set of visited vertices
            for u in self.neighbours:
                if u not in visited:  # Only recurse on vertices that haven't been visite
d
                    if u.check_connected(target_item, visited):
                        return True

            return False
```

17.6 Cycles and Trees

- Cycle: a sequence of vertices v0, ... , vk s.t.
  - k ≥ 3
  - v0 = vk (all other vertices are distinct from each other and v0)
  - each consecutive pair of vertices is adjacent

- If G is connected and e is in cycle, then G - e is connected
- If there is e in E s.t. G - e is connected, then e is in cycle in G, the original path

- Trees (Minimally-connected graphs)
  - connected graphs that have no cycles
  - in graph theory, no hierachy
  1. trees have no cycle but are connected
  2. trees are minimally connected: the graph which have the fewest number of edges possible to remain connected
  3. Removing any edge from a tree results in a graph that is NOT CONNECTED
  - Number of edges in tree = No. of Vertices - 1
  - G′ is the spanning tree of G when G′ is a tree (no cycle)
  - Spanning_tree (pseudo code)
    - all edges
    - if there is a cycle in all edges
      - remove edge in cycle from all the edges
    - return all the remaining edges
    -

```
class _Vertex:
    def spanning_tree(self, visited: set[_Vertex]) → list[set]:
        """Return a list of edges that form a spanning tree of all vertices t
hat are
        connected to this vertex WITHOUT using any of the vertices in vis
ited.

        The edges are returned as a list of sets, where each set contains
the two
        ITEMS corresponding to an edge.
```

```
        Preconditions:
            - self not in visited
        """
        edges_so_far = []

        visited.add(self)
        for u in self.neighbours:
            if u not in visited:  # Only recurse on vertices that haven't been
visited
                edges_so_far.append({self.item, u.item})
                edges_so_far.extend(u.spanning_tree(visited))

        return edges_so_far


>>> # Using same graph g as above
>>> g._vertices[0].spanning_tree(set())
[{0, 3}, {0, 1}, {1, 4}, {2, 4}]

class Graph:
    def spanning_tree(self) -> list[set]:
        """Return a subset of the edges of this graph that form a spannin
g tree.

        The edges are returned as a list of sets, where each set contains
the two
        ITEMS corresponding to an edge. Each returned edge is in this gr
aph
        (i.e., this function doesn't create new edges!).

        Preconditions:
            - this graph is connected
        """
        # Pick a vertex to start
        all_vertices = list(self._vertices.values())
```

```
        start_vertex = all_vertices[0]

        # Use our helper _Vertex method!
        return start_vertex.spanning_tree(set())
```

- Every tree has |E| = |V| - 1
  - If G is connected , then |E| ≥ |V| - 1
  - If |e| < |v| - 1, then G is not connected
- Every connected graph either is a tree, or can be made into a tree by removing edges from cycles.
- |E| ≥ (|V|-2)(|V|-1)/2 + 1, then G is connected

Ch18 Sorting

18.2 Selection Sort

> We “select” the smallest from the unsorted list, to be put into the sorted list.

- **in-place** when it sorts a list by mutating its input list, and without using any additional list objects
- ALWAYS take O(n^2) regardless of what input

```
def selection_sort(lst: list) -> None:
    """Sort the given list using the selection sort algorithm.

    Note that this is a *mutating* function.

    >>> lst = [3, 7, 2, 5]
    >>> selection_sort(lst)
    >>> lst
    [2, 3, 5, 7]
    """
    for i in range(0, len(lst)):
        # Loop invariants
        assert is_sorted(lst[:i])
        assert i == 0 or all(lst[i - 1] <= lst[j] for j in range(i, len(lst)))

        # Find the index of the smallest item in lst[i:] and swap that
        # item with the item at index i.
        index_of_smallest = _min_index(lst, i)
        lst[index_of_smallest], lst[i] = lst[i], lst[index_of_smallest]


def _min_index(lst: list, i: int) -> int:
    """Return the index of the smallest item in lst[i:].

    In the case of ties, return the smaller index (i.e., the index that appears firs
t).

    Preconditions:
        - 0 <= i <= len(lst) - 1
    """
    index_of_smallest_so_far = i

    for j in range(i + 1, len(lst)):
        if lst[j] < lst[index_of_smallest_so_far]:
            index_of_smallest_so_far = j
```

```
    return index_of_smallest_so_far
```

Running-time Analysis (Selection sort):

1. _min_index(lst, i):
    - Let n be length of input list.
        - Statement outside of loop: 1 step
        - Loop: iterates n - i - 1 times
            - Body is 1 step
        - Total: 1 + (n - i - 1) steps = Theta(n-i)
2. Selection sort:

- Let n be length of input list.
- Loop: runs for n times
    - Each iteration runs for (n - i step from _min_index) and (1 step from the swapping)

This gives us a total running time of:

$$\begin{aligned}\sum_{i=0}^{n-1} n - i + 1 &= n(n+1) - \sum_{i=0}^{n-1} i \\ &= n(n+1) - \frac{n(n-1)}{2} \\ &= \frac{n(n+3)}{2}\end{aligned}$$

Therefore the running time of `selection_sort` is $\Theta(n^2)$.

18.3 Insertion Sort

- insertion sort doesn't choose the smallest unsorted element to add to the sorted part.
- Instead, it always takes the next item in the list, `lst[i]`, and ***inserts it* into the sorted part by moving it into the correct location to keep this part sorted.**
-

```
# Version 1, using an early return
def _insert(lst: list, i: int) -> None:
    for j in range(i, 0, -1):  # This goes from i down to 1
        if lst[j - 1] <= lst[j]:
            return
        else:
            # Swap lst[j - 1] and lst[j]
            lst[j - 1], lst[j] = lst[j], lst[j - 1]


# Version 2, using a compound loop condition
def _insert(lst: list, i: int) -> None:
    j = i
    while not (j == 0 or lst[j - 1] <= lst[j]):
        # Swap lst[j - 1] and lst[j]
        lst[j - 1], lst[j] = lst[j], lst[j - 1]

        j -= 1

def insertion_sort(lst: list) -> None:
    """Sort the given list using the insertion sort algorithm.

    Note that this is a *mutating* function.
    """
    for i in range(0, len(lst)):
        assert is_sorted(lst[:i])

        _insert(lst, i)
```

- Has the early return. Don't need to compare with everything that come before it (since it is sorted)

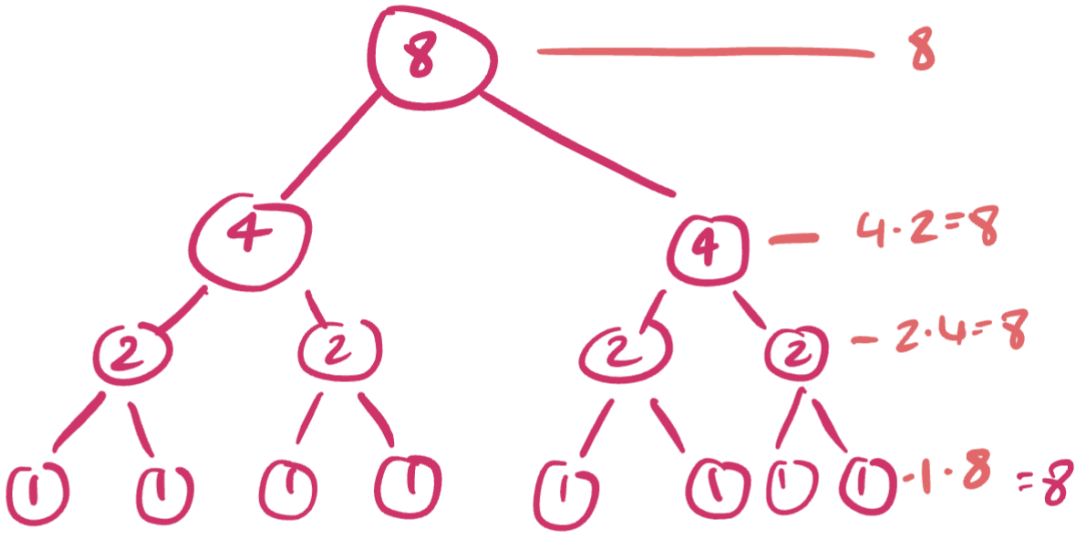
18.5 Merge sort

Running time analysis!

- Height of the tree: log2(n) + 1 (the one is to account for the root)
- log2(n) is because for n = 2^k, it can split k times.

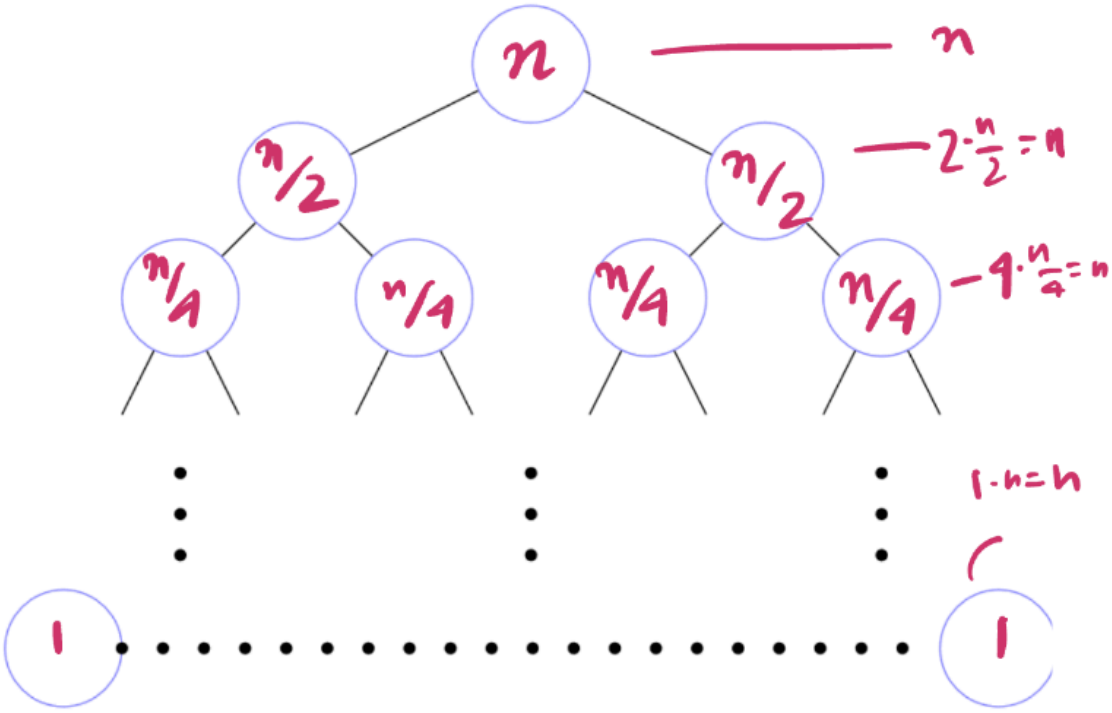
1. Suppose we call `mergesort` on a list of length **8**. Draw the corresponding recursion diagram, and inside each node write down the non-recursive running time of the call, which we count as equal the *size of the input list* for that call.

   (For example, the root of the tree should contain an "8", and its two children should be "4"s.)



2. Compute the total of the numbers in your above diagram. This gives you the total running time fc `mergesort` on a list of length 8.

$8 \cdot 4 = 32$



4. Compute the total of the numbers in your above diagram, again assuming n is a power of 2.

   Hint: consider the sum of the numbers in each *level* of the tree.

each level = n steps

# of levels = $\log_2 n + \underbrace{1}_{\text{root}}$

Total work:

$$n \times (\log_2 n + 1)$$

$$\Theta(n \log n)$$

```
def mergesort(lst: list) -> list:
    if len(lst) < 2:
        return lst.copy()  # Use the list.copy method to return a new list object
    else:
        # Divide the list into two parts, and sort them recursively.
        mid = len(lst) // 2
        left_sorted = mergesort(lst[:mid])
        right_sorted = mergesort(lst[mid:])

        # Merge the two sorted halves. Using a helper here!
        return _merge(left_sorted, right_sorted)


# The base lists are sorted as you merge into a sorted list!
def _merge(lst1: list, lst2: list) -> list:
    """Return a sorted list with the elements in lst1 and lst2.

    Preconditions:
        - is_sorted(lst1)
        - is_sorted(lst2)

    >>> _merge([-1, 3, 7, 10], [-3, 0, 2, 6])
    [-3, -1, 0, 2, 3, 6, 7, 10]
    """
    i1, i2 = 0, 0
    sorted_so_far = []

    while i1 < len(lst1) and i2 < len(lst2):
        # Loop invariant:
        # sorted_so_far is a merged version of lst1[:i1] and lst2[:i2]
        assert sorted_so_far == sorted(lst1[:i1] + lst2[:i2])

        if lst1[i1] <= lst2[i2]:
            sorted_so_far.append(lst1[i1])
            i1 += 1
```

```
        else:
            sorted_so_far.append(lst2[i2])
            i2 += 1

    # When the loop is over, either i1 == len(lst1) or i2 == len(lst2)
    assert i1 == len(lst1) or i2 == len(lst2)

    # In either case, the remaining unmerged elements can be concatenated to
sorted_so_far.
    if i1 == len(lst1):
        return sorted_so_far + lst2[i2:]
    else:
        return sorted_so_far + lst1[i1:]
```

Let's dry run your `mergesort` and `_merge` functions step by step on this example:

🧪 Input:

```
mergesort([6, 3, 8, 5, 2])
```

🔁 Step 1: `mergesort([6, 3, 8, 5, 2])`

- `mid = 2`
- Split into: `[6, 3]` and `[8, 5, 2]`

So we recursively do:

```
left_sorted = mergesort([6, 3])
right_sorted = mergesort([8, 5, 2])
```

🔁 Step 2: `mergesort([6, 3])`

- `mid = 1`
- Split into: `[6]` and `[3]`

- Base case: return copies of `[6]` and `[3]`

Now merge:

```
_merge([6], [3])
```

🔁 Step 3: `_merge([6], [3])`

Start:

- `i1 = 0`, `i2 = 0`, `sorted_so_far = []`

Compare:

- `6 > 3`, so add `3`: `sorted_so_far = [3]`, `i2 = 1`

Now `i2 == len(lst2)`, so we append the rest of `lst1`:

```
[3] + [6] = [3, 6]
```

Now we go back to Step 1 with:

```
left_sorted = [3, 6]
```

🔁 Step 4: `mergesort([8, 5, 2])`

- `mid = 1`
- Split into: `[8]` and `[5, 2]`

Base case:

- `[8]` → return copy

Now recurse:

```
mergesort([5, 2])
```

🔁 Step 5: `mergesort([5, 2])`

- `mid = 1`
- Split into: `[5]` and `[2]`
- Base case: return copies

Now merge:

```
_merge([5], [2])
```

🔁 Step 6: `_merge([5], [2])`

- Compare `5 > 2`, so add `2` → `sorted_so_far = [2]`
- Then append rest of `[5]` → `[2, 5]`

Back to Step 4:

```
_merge([8], [2, 5])
```

🔁 Step 7: `_merge([8], [2, 5])`

- Compare `8 > 2` → add `2`
- Compare `8 > 5` → add `5`
- Now append remaining `[8]`

Result: `[2, 5, 8]`

Now back to Step 1:

```
_merge([3, 6], [2, 5, 8])
```

🔁 Final Merge: `_merge([3, 6], [2, 5, 8])`

- Compare `3 > 2` → `[2]`
- Compare `3 < 5` → `[2, 3]`
- Compare `6 > 5` → `[2, 3, 5]`

- Compare `6 < 8` → `[2, 3, 5, 6]`
- Append remaining `[8]`

Result: `[2, 3, 5, 6, 8]`

✅ Final Output:

```
[2, 3, 5, 6, 8]
```

Would you like a visualization or diagram-style breakdown of how the merge tree looked?

18.6 quicksort

Remember! smaller: is SMALLER OR EQUAL to pivot

smaller is ≤ pivot

bigger is > pivot.

- in-place algorithm: (MARKING SCHEME)
    1. Original list is mutated
    2. No new list are created

Running-time analysis:

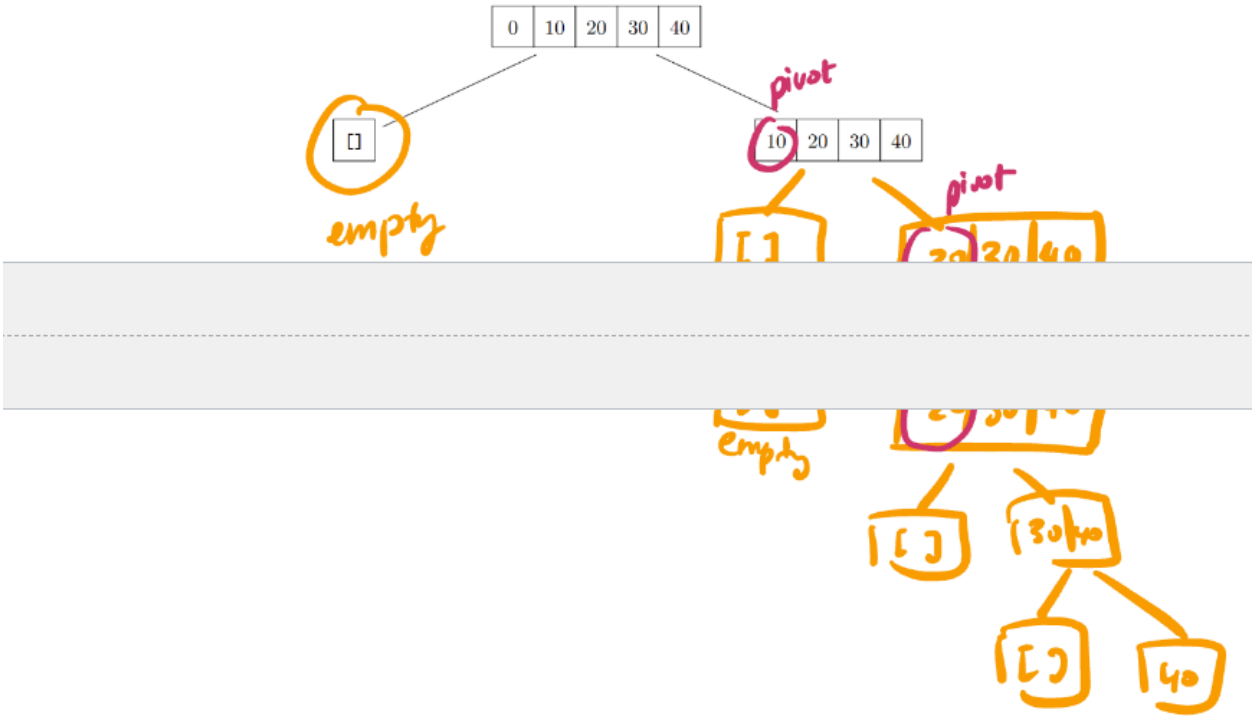
- Assume random pivot (median is the pivot),
    - then each partition splits smaller and bigger into even size
    - then it is similar to merge sort, with log2(n) + 1 height.
    - nlogn
- Quicksort takes less step than merge sort for O(nlogn) case on average. It is also more efficient.
- In-place quicksort: also more space-efficient to merge sort

1. Suppose we call `quicksort([0, 10, 20, 30, 40])`. After the `_partition` call, what are `smaller` and `bigger`?

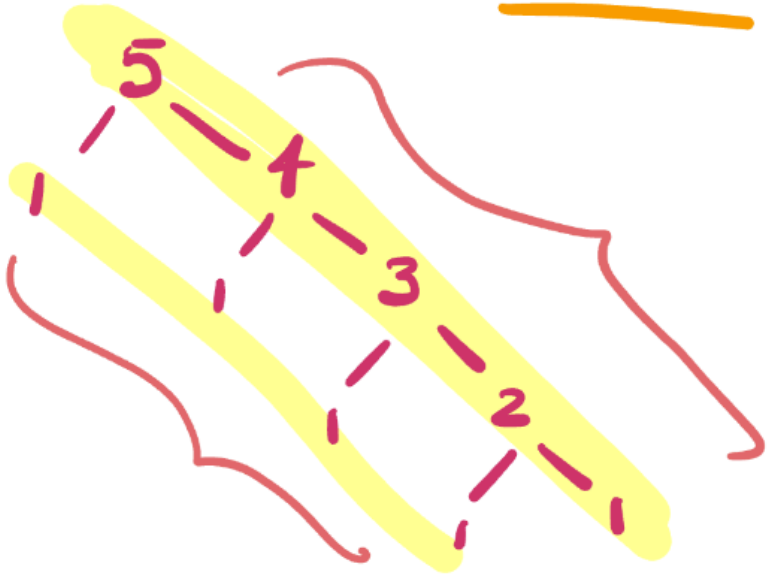
smaller = []

bigger = [10, 20, 30, 40]

2. Draw a recursion tree showing the *inputs* to each recursive call, when we call `quicksort([0, 10, 20, 30, 40])`. We've started the first two levels for you below. The node containing `[]` represents a recursive call on an empty list.



3. Now redraw the recursion tree, but with the *non-recursive running time* in each node.
    - For an empty list, the non-recursive running time is 1.
    - For a non-empty list, the non-recursive running time is the *length of the list*.



5. Generalize the above calculation for a call to quicksort with a list of length n, when the chosen pivot is always the *smallest* element in the list.

$$\sum_{i=1}^{n} i + (n-1)$$

$$= \frac{n(n+1)}{2} + (n-1)$$

$$\in \Theta(n^2)$$

```
def quicksort(lst: list) -> list:
    if len(lst) < 2:
        return lst.copy()
    else:
        pivot = lst[0]
        smaller, bigger = _partition(lst[1:], pivot)
        return quicksort(smaller) + [pivot] + quicksort(bigger)


def _partition(lst: list, pivot: Any) -> tuple[list, list]:
    """Return a partition of lst with the chosen pivot.

    Return two lists, where the first contains the items in lst
    that are <= pivot, and the second contains the items in lst that are > pivot.
    """
    smaller = []
    bigger = []

    for item in lst:
        if item <= pivot:
            smaller.append(item)
        else:
            bigger.append(item)

    return (smaller, bigger)
```

Absolutely! Let's do a **dry run** of your `quicksort` function step-by-step.

🧪 Input:

```
quicksort([6, 3, 8, 5, 2])
```

We'll assume that `_partition(lst, pivot)` returns two lists:

- `smaller`: all elements ≤ pivot
- `bigger`: all elements > pivot

🔁 Step-by-step Dry Run

1️⃣ First Call: `quicksort([6, 3, 8, 5, 2])`

- `pivot = 6`
- `lst[1:] = [3, 8, 5, 2]`
- `_partition([3, 8, 5, 2], 6)` → `smaller = [3, 5, 2]`, `bigger = [8]`

Now recursively call quicksort on both:

```
quicksort([3, 5, 2]) + [6] + quicksort([8])
```

2️⃣ Left Subtree: `quicksort([3, 5, 2])`

- `pivot = 3`
- `lst[1:] = [5, 2]`
- `_partition([5, 2], 3)` → `smaller = [2]`, `bigger = [5]`

Now:

```
quicksort([2]) + [3] + quicksort([5])
```

3️⃣ `quicksort([2])`

- Base case (`len < 2`) → returns `[2]`

4️⃣ `quicksort([5])`

- Base case (`len < 2`) → returns `[5]`

Now back to step 2:

```
[2] + [3] + [5] = [2, 3, 5]
```

5️⃣ Right Subtree: `quicksort([8])`

- Base case → returns `[8]`

🔚 Final Merge from Step 1:

```
[2, 3, 5] + [6] + [8] = [2, 3, 5, 6, 8]
```

✅ Final Output:

```
[2, 3, 5, 6, 8]
```

Would you like to dry run `_partition()` too or rewrite this to do it in-place (no extra lists)?

Recursive Runtime Analysis

1. Find the recursive call structure (tree diagram)
2. Analyse the non-recursive running time
3. Fill in the tree with the non-recursive running time of each call andcalculate the sum of allof the non-recursive running times

18.9 Generalized Sorting

Exercise 1: Sorting by length

- We want to sort it by length. But using the current insertion sort, it is going to do string comparison on the alphabets instead of the length.
- Approach 1: By len

```
def insertion_sort(lst: list) -> None:
    for i in range(0, len(lst)):
        _insert(lst, i)


def _insert(lst: list, i: int) -> None:
    for j in range(i, 0, -1):

        # if lst[j - 1] <= lst[j]: # This compares the value
        if len(lst[j - 1]) <= len(lst[j]): # This compares the length
            return
        else:
            lst[j - 1], lst[j] = lst[j], lst[j - 1]
```

- Approach 2: By Key
    - “calling key(’hello’) is SAME as len(’hello’). A Callable is just a complete placeholder for the name of the function.

```
def insertion_sort_by_key(lst: list,
                          key: Optional[Callable] = None) -> None:
    """Sort the given list using the insertion sort algorithm.

    If key is not None, sort the items by their corresponding
    return value when passed to key.
    """

    for i in range(0, len(lst)):
        _insert_by_key(lst, i, key)

def _insert_by_key(lst: list, i: int, key: Optional[Callable] = None) -> None:
```

```
    """Move lst[i] so that lst[:i + 1] is sorted.

    If key is not None, sort the items by their corresponding
    return value when passed to key.

    Precondition:
        - key is either None or it is a single-argument function that can be
         called on every element of lst without error
    """
    if key is None:
        # code omitted ... will be exact same as our original _insert method

    else: # `key` is a function that we should use to compare values
        for j in range(i, 0, -1):

            if key(lst[j - 1]) <= key(lst[j])########################
            # same as if len(lst[j - 1]) <= len(lst[j]):
                return
            else:
                lst[j - 1], lst[j] = lst[j], lst[j - 1]

>>> insertion_sort_by_key(['bumbly', 'is', 'soo', 'fluffyyyy'], len)
```

- Memoization: keeping track of values that we already computed → lead to more efficient code as we can re-use the values we saved, rather having to recompute the same result over and over
-

```
def insertion_sort_memoized(lst: list, key: Optional[Callable] = None) → None:
    """Sort the given list using the insertion sort algorithm.

    If key is not None, sort the items by their corresponding return value
    when passed to key. Use a dictionary to keep track of "key" values,
    so that the function is called only once per list element.
```

```
    Note that this is a *mutating* function.

    >>> lst = ['cat', 'octopus', 'hi', 'david']
    >>> insertion_sort_memoized(lst, key=len)
    >>> lst
    ['hi', 'cat', 'david', 'octopus']
    >>> lst2 = ['cat', 'octopus', 'hi', 'david']
    >>> insertion_sort_memoized(lst2)
    >>> lst2
    ['cat', 'david', 'hi', 'octopus']
    """
    # Use this variable to keep track of the saved "key" values
    # across the different calls to _insert.
    key_values = {}
    for i in range(0, len(lst)):
        _insert_memoized(lst, i, key, key_values)


# Define the _insert_memoized helper below.
# Hint: You'll need to modify the _insert_by_key helper function to use the
# additional dictionary argument.
def _insert_memoized(lst: list, i: int,
                     key: Optional[Callable] = None,
                     key_values: Optional[dict]) -> None:
    """Same as _insert_by_key, except that:

    When key(x) should computed, first look up x in key_values.

        - If x is in key_values, return the corresponding values
        - Otherwise, compute key(x), and then store the result in key_values
    """
        if key is None:
        # code omitted ... will be exact same as our original _insert method

    else: # `key` is a function that we should use to compare values
        for j in range(i, 0, -1):
```

```
                if not key_values[lst[j-1]]:#########
                    key_values[lst[j-1]] = key(lst[j-1]) ##########
                if not key_values[lst[j]]:##########
                    key_values[lst[j]] = key(lst[j])########
            if key_values[lst[j-1]] <= key_values[lst[j]]#########
            # same as if len(lst[j - 1]) <= len(lst[j]):
                return
            else:
                lst[j - 1], lst[j] = lst[j], lst[j - 1]
```

- Anonymous function

```
lambda x: x + 1 #taking x and returning x + 1
lambda lst1, lst2: len(lst1) * len(lst2)

>>> strings = ['david', 'is', 'amazing']
>>> sorted(strings, lambda s: s.count('a')) #Sort with key, with key being lam
bda
['is', 'david', 'amazing']
```

- Build in sorting: sorted and list.sort has the key argument built-in