12. Interlude: Nifty Python Features

12.1 Sequences Revisited: Ranges, Indexing, and Slicing

- `range`
    - step `range(start, stop, step)`
    - optional start - the `start` argument is <u>optional</u> and default <u>0</u>
        - `range(i)` is equivalent to `range(0, i)`

```
>>> [x for x in range(0, 10, 2)]
[0, 2, 4, 6, 8]

>>> [x for x in range(0,5)]
[0, 1, 2, 3, 4]

>>> [x for x in range(5)]
[0, 1, 2, 3, 4]
```

- negative indexing - access elements offset from the <u>end</u> of the sequence
    - `seq[-i]` is equivalent to `seq[len(seq)-1]`

```
>>> seq = [10, 20, 30, 40]
>>> seq[-1]
40
```

- sequence slicing `seq[i:j]`
    - has the same data type as `seq`
    - equivalent `[seq[x] for x in range(i, j)]`
    - `seq[start:stop:step]`
    - `seq[::1]` for reversing

```
>>> [10, 20, 30, 40][1:3]
[20, 30]

>>> 'Hello'[:3]
'Hel'

>>> 'David is cool'[10:0:-1]
'oc si diva'
```

12.2 String Interpolation with f-strings

- in an **f-string**, any text surrounded by `{...}` is interpreted as a Python expression

```
>>> family_name = 'Liu'
>>> given_name = 'David'
>>> student_number = 123456789

>>> f'{family_name}, {given_name} ({student_number})'
'Liu, David (123456789)'

>>> f'{family_name.upper()}'
'LIU'
```

12.3 Function with Optional Parameters

```
# parameter definition with default value syntax
def ...(<parameter_name>: <parameter_type> = <default_value>, ...) -> ...:
```

```
def increment(n: int, step: int = 1) -> int:
    """Return n incremented by step.

    If the step argument is omitted, increment by 1 instead.
    """
    return n + step

>>> increment(10, 2)  # n = 10, step = 2
12
>>> increment(10)     # n = 10
11
```

- in the function header, optional parameters must be written **after** mandatory parameters
- **WARNING** - do not use mutable objects as default values
- every default value is an object that s created when the function is **defined**, not when the function is **called**
- can use `None`

```
# wrong
def add_num(num: int, numbers: list[int] = []) -> list[int]:

# right
def add_num(num: int, numbers: Optional[list[int]] = None) -> list[int]:
```

13. Linked List

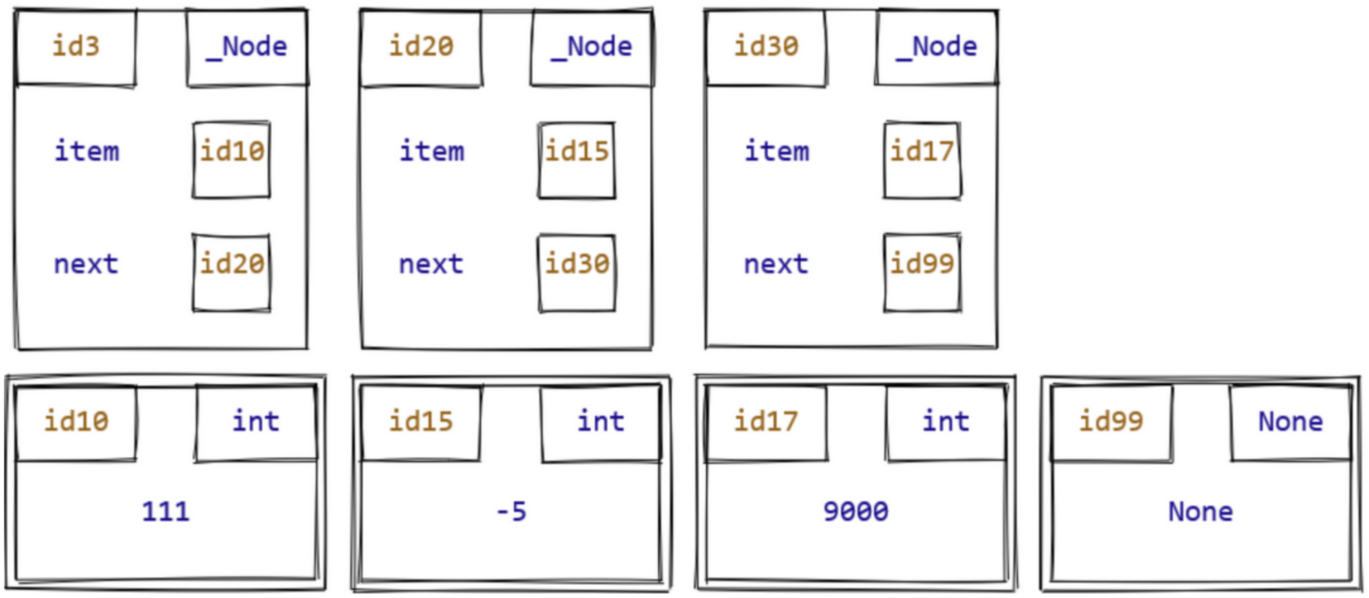
13.1 Introduction

- use `_<class_name>` to indicate that this entire class is private

```
from dataclasses import dataclass
from typing import Optional
```

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

    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 other node
```

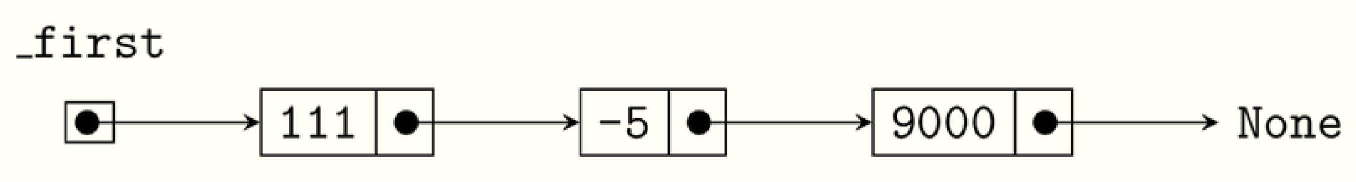
- _Node represents a single element of a list



```python
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) -> None:
        """Initialize an empty linked list.
        """
        self._first = None
```

```
>>> linky = LinkedList()  # linky is an empty linked list
>>> linky._first is None
True
>>> node1 = _Node(111)    # New node with item 111
>>> node2 = _Node(-5)     # New node with item -5
>>> node3 = _Node(9000)   # New node with item 900
>>> node1.item
111
>>> node1.next is None    # By default, new nodes do not link to another node
True
>>> node1.next = node2    # Let's set some links
>>> node2.next = node3
>>> node1.next is node2   # Now node1 links to node2!
True
>>> node1.next.item
-5
>>> node1.next.next is node3
True
>>> node1.next.next.item
9000
>>> linky._first = node1    # Finally, set linky's first node to node1
>>> linky._first.item       # linky now represents the list [111, -5, 9000]
111
>>> linky._first.next.item
-5
>>> linky._first.next.next.item
9000
```

- `node1` is an object `_Node`, `node.item` is the value stored in `node`



13.2 Traversing Linked List

```
# 1. Initialize curr to the start of the list.
```

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


class LinkedList:
    def print_items(self) -> None:
        """Print out each item in this linked list."""
        curr = self._first
        while curr is not None:
            print(curr.item)
            curr = curr.next

    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
```

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

@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 other 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) -> None:
        """Initialize an empty linked list.
        """
        self._first = None

    def sum_items(self) -> int:
        """Return the sum of the items in this linked list.
        Preconditions:
            - all items in this linked list are ints
        """
        sum_so_far = 0
        curr = self._first
        while curr is not None:  # or, while not (curr is None):
            sum_so_far = sum_so_far + curr.item
            curr = curr.next
        # Contrast with:
        # i = 0
        # while i < len(self):
        #     sum_so_far = sum_so_far + self[i]
        #     i = i + 1
        return sum_so_far
    ############################################################################
    # Exercise 1: Linked List Traversal
```

```
###########################################################################
    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_so_far = -math.inf
        # Comment: could also initialize to self._first.item
        curr = self._first

        while curr is not None:  # or, while not (curr is None):
            if curr.item > max_so_far:
                max_so_far = curr.item
            # Or,
            # max_so_far = max(max_so_far, curr.item)
            curr = curr.next
        return max_so_far

    def __contains__(self, item: Any) -> bool:
        """Return whether item is in this list.
        >>> linky = LinkedList()
        >>> linky.__contains__(10)
        False
        >>> node2 = _Node(20)
        >>> node1 = _Node(10, node2)
        >>> linky._first = node1
        >>> linky.__contains__(20)
        True
        """
        curr = self._first
        while curr is not None:
            # We should be comparing the node's item with item,
            # not the node itself.
            # As written, this comparison will always be False
            # (assuming item isn't a _Node).
            # if curr == item:
            if curr.item == item:
                # We've found the item and can return early.
                return True
            curr = curr.next
        # If we reach the end of the loop without finding the item,
        # it's not in the linked list.
        return False

    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
        raise IndexError
        # Version 2: not using an early return, but using a "compound loop
        # condition"
        # HOMEWORK: read about this in 13.2
        # curr = self._first
        # curr_index = 0  # the index of the current node
        #
        # # Idea: modify the loop condition so that we stop when EITHER:
        # #  1. we reach the end of the list (curr is None)
        # #  2. we reach the right index (curr_index == i)
        # while not (... or ...):
        #     curr = curr.next
        #     curr_index = curr_index + 1
        #
        # # Now, detect which of the two cases we're in (1 or 2)
        # # and handle each case separately.
        # if ...:
        #     ...
        # else:
        #     ...
    # version 2
    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 not (curr is None or curr_index == i):
            curr = curr.next
            curr_index += 1

        # Note: this is the *stopping condition*
        assert curr is None or curr_index == i

        if curr is None:
            raise IndexError
        else:
            return curr.item
```

13.3 Mutating Linked List

- "off-by-one" error - loop iterates one too many times

```
class LinkedList:

    def __init__(self, items: Iterable) -> None:
        """Initialize a new linked list containing the given items.
        """
        self._first = None
        for item in items:
            self.append(item)

    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.next is not None:
                curr = curr.next

            # After the loop, curr is the last node in the LinkedList.
            assert curr is not None and curr.next is None
            curr.next = new_node
```

- in `_init_`, removing the line `self._first = None`

  → error `AttributeError: 'LinkedList' object has no attribute '_first'` in `append`

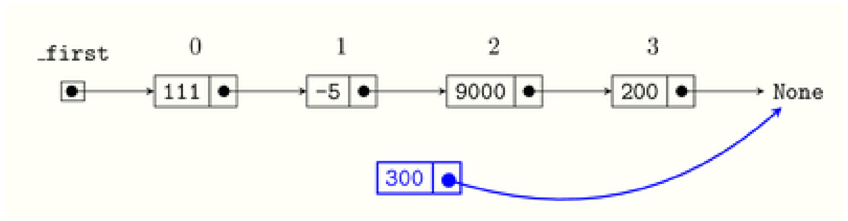
- `append` running time analysis

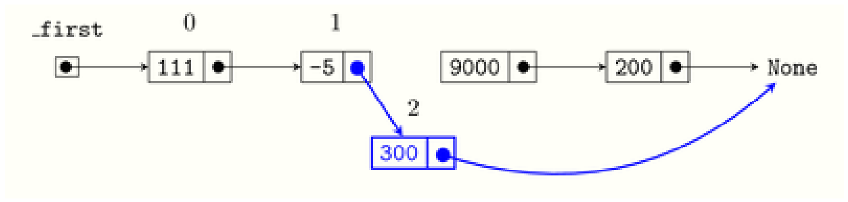
  > assume self.size = n
  >
  > if branch → 3
  >
  > else branch:
  >
  > while loop → $n-1$
  >
  > total → $1+1+1+(n-1)+1=n+3$ → $\Theta(n)$

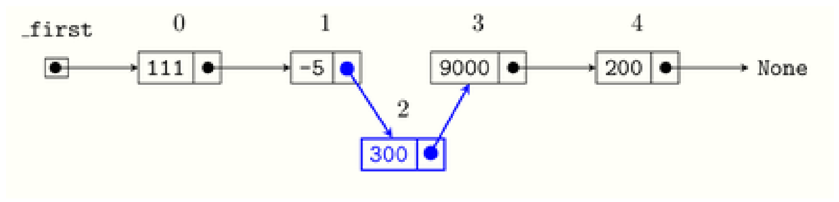
- `_init_` running time analysis

  > assume the size of `items` is n
  >
  > total:
  > $1+3+(1+3)+(2+3)+\ldots+(n-1+3)=4+\sum_{i=1}^{n-1}(i+3)=4+\frac{(n+6)(n-1)}{2}$
  > → $\Theta(n^2)$

13.4 Index-Based Mutation







```python
class LinkedList:
    def insert(self, i: int, item: Any) -> None:
        """..."""
        new_node = _Node(item)

        curr = self._first
        curr_index = 0

        while not (curr is None or curr_index == i - 1):
            curr = curr.next
            curr_index = curr_index + 1

        # After the loop is over, either we've reached the end of the list
        # or curr is the (i - 1)-th node in the list.
        assert curr is None or curr_index == i - 1

        if curr is None:
            # i - 1 is out of bounds. The item cannot be inserted.
            raise IndexError
        else: # curr_index == i - 1
            # i - 1 is in bounds. Insert the new item.
            new_node.next = curr.next
            curr.next = new_node
```

- **Common Error**
  - the **order** in which we update the links **really matters**

    ```
    curr.next = new_node
    new_node.next = curr.next

    # equivalent to new_node.next = new_node
    ```

  - use parallel assignments to avoid

    ```
        curr.next, new_node.next = new_node, curr.next
    ```

- corner case `i == 0`

  ```
  ...
  new_node = _Node(item)

          if i == 0:
              # Insert the new node at the start of the linked list.
              self._first, new_node.next = new_node, self._first
          else:
              ...
                  curr.next, new_node.next = new_node, curr.next
  ```

- index-based deletion

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

          Preconditions:
              - i >= 0
  ```

```
        Raise IndexError if i >= the length of self.

        >>> linky = LinkedList([1, 2, 10, 200])
        >>> linky.pop(2)
        10
        >>> linky.pop(0)
        1
        >>> linky.to_list()
        [2, 200]
        """
        # 1. If the list is empty, you know for sure that index is out of
bounds...
        curr = self._first
        if curr is None:
            raise IndexError

         # 2. Else if i is 0, remove the first node and return its item.
        else if i == 0:
            item = curr.item
            self._first = curr.next
            return item
        # 3. Else iterate to the (i-1)-th node and update links to remove
        # the node at position index. But don't forget to return the item!
        else:
            curr_index = 0
            while not (curr is None or curr_index == i - 1):
                curr = curr.next
                curr_index += 1

            assert curr is None or curr_index == i - 1

            if curr is None or curr.next is None:
                raise IndexError
            else:
                item = curr.next.item
                curr.next = curr.next.next
                return item
```

- remove

```python
class LinkedList:
    def remove(self, item: Any) -> None:
        """Remove the first occurrence of item from the list.
        Raise ValueError if the item is not found in the list

        >>> lst = LinkedList([10, 20, 30, 20])
        >>> lst.remove(20)
        >>> lst.to_lst()
        [10, 30, 20]
        """

        pre, curr = None, self._first
        if curr is None:
            raise ValueError

        while not (curr is None or curr.item == item):
            pre, curr = curr, curr.next

        assert curr is None or curr.item == item

        if curr is None:
            raise ValueError
        elif pre is None:
            self._first = curr.next
        else:
            pre.next = curr.next
```

13.5 Linked List Running-Time Analysis

- `LinkedList.insert` running time anaysis
    - Let n be the length of `self`
    - **Case 1: Assume** `i==0`.
        - the if branch executes, which takes constant time, so we'll count it as one step
    - **Case 2: Assume** `i>0`
        - The first two statements in the else branch take constant time, so we'll count them as 1 step.
        - The statements after the while loop all take constant time, so we'll count them as one step

        - The while loop iterates until either it reaches the end of the list or until it reaches the correct index
            - so happens after n iterations or $i-1$ iterations
            - the number of iterations taken is $min(n, i-1)$
            - each iteration takes 1 step, for a total of $min(n, i-1)$ steps
        - total running time of $1 + min(n, i-1) + 1 = min(n, i-1) + 2$ steps
    - In the first case, we have a running time of $\Theta(1)$. In the second case, we have a running time of $\Theta(min(n, i))$. The second expression also becomes $\Theta(1)$ when $i = 0$, and the overall running time is $\Theta(min(n, i))$
    - note: $min(n, i-1) \in \Theta(min(n, i))$ since $i - 1 \in \Theta(i)$
- assume that $0 \leq i < n$ in which case $min(i, n) = i$, and we get that the running time is $\Theta(i)$. That is a simplification under an additional assumption that $i < n$.
- essentially, we say that if we treat i as small with respect to the size of the list, then the running time of the algorithm does not depend on the size of the list.
    - the most extreme case of this is when `i == 0`, so we're inserting into the front of the linked list. This takes constant time, meaning it does not depend on the length of the linked list.

Operation (assuming $0 \leq i < n$)	**Running time (`list`)**	**Running time (`LinkedList`)**
Indexing(`lst[i]`)	$\Theta(1)$	$\Theta(i)$
Insert into index i	$\Theta(n-i)$	$\Theta(i)$
Remove item at index i	$\Theta(n-i)$	$\Theta(i)$

14. Induction and Recursion

14.1 Proof by Induction

- the principle of induction applies to universal statements over the natural numbers $\forall x \in \mathbb{N}, P(n)$.
    - **base case**: prove that $P(0)$ holds
    - **inductive step**: prove that $\forall k \in \mathbb{N}, P(k) \implies P(k+1)$

> Given statement to prove: $\forall n \in \mathbb{N}, P(n)$.
>
> **Base case:** Let $n = 0$.
>
> [prove that $P(0)$ is true]
>
> **Inductive step:** Let $k \in \mathbb{N}$, and assume that $P(k)$ is true.(induction hypothesis)
>
> [prove that $P(k+1)$ is true]

14.2 Recursively-Defined Functions

- recursion definition - f is defined in terms of itself
- `f` is a **recursively-defined function** when it contains a call to itself in its body
- **recursive call** - inner `f(n-1)` call
- **recursion** - the programming technique of defining recursive functions to perform computations and solve problems
  - **base case:** does not require any additional "breaking down" of the problem
  - **recursive step:** require the problem to be broken down into an instance of a smaller size
- Python: "When we call `f(_)`, the recursive call `f(_)`..." → extremely time-consuming and error-prone
- **inductive approach (partial tracing)** - assume that the recursive call returns the correct result, based on the function's specification, and without relying on having explicitly traced that call.

> 1. when we call `f(100)`, then the recursive call `f(100-1)=f(99)` is made. Assuming this call is correct, it returns `4950`
> 2. Then `4950+100==5050` is returned.

```python
def euclidean_gcd_rec(a: int, b: int) -> int:
    """Return the gcd of a and b (using recursion!).

    Preconditions:
        - a >= 0 and b >= 0
    """
    if b == 0:
        return a
    else:
        return euclidean_gcd_rec(b, a % b)
```

14.4 Nested Lists and Structural Recursion

- **nested list of integers** - as one of two types of values
  - For all $n \in \mathbb{Z}$, the single integer n is a nested list of integers.
  - For all $k \in \mathbb{N}$ and nested lists of integers $a_0, a_1, \dots, a_{k-1}$, the list $[a_0, a_1, \dots, a_{k-1}]$ is also a nested list of integers.
- $nested\_sum(x) = \begin{cases} x, & \text{if } x \in \mathbb{Z} \\ \sum_{i=0}^{k-1} nested\_sum\,(a_i), & \text{if } x = [a_0, a_1, \dots, a_{k-1}] \end{cases}$
  - The sum of a nested list that's an **integer** is simply the value of that integer itself.
  - The sum of a nested list of the form $[a_0, a_1, \dots, a_{k-1}]$ is equal to the sum of each of the a_i''s added together.
- `int | list` means `int` or `list`

```python
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
```

```python
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
function.
    """
    if isinstance(nested_list, int):
        return nested_list
    else:
        return sum(sum_nested_v2(sublist) for sublist in nested_list)
```

- suppose we want to trace the call

  ```
  >>> sum_nested_v1([1, [2, [3, 4], 5], [6, 7], 8])
  36
  ```

Iteratoin	`sublist`	`sum_nested_v1(sublist)`	**Accumulator**`sum_so_far`
0	N/A	N/A	`0`
1	`1`		
2	`[2,[3,4],5]`		
3	`[6,7]`		
4	`8`		

  - we will <u>assume</u> that each recursive call is correct (NOTE: this assumption depends only on the specification of `sum_nested_v1` written in its **docstring**, and not its implementation)

Iteratoin	`sublist`	`sum_nested_v1(sublist)`	**Accumulator**`sum_so_far`
0	N/A	N/A	`0`
1	`1`	`1`	`1`

Iteratoin	`sublist`	`sum_nested_v1(sublist)`	**Accumulator**`sum_so_far`
2	`[2,[3,4],5]`	`14(2+3+4+5)`	`15`
3	`[6,7]`	`13(6+7)`	`28`
4	`8`	`8`	`36`

- recursive function design recipe for nested lists
  1. Write a doctest example to illustrate the base case of the function, when the function is called on a single `int` value
  2. Write a doctest example to illustrate the recursive step of the function
     - pick a nested list with around 3 sublists, where at least one sublist is a single `int`, and another sublist is a `list` that **contains** other lists
     - your doctest should show the correct return value of the function for this input nested list
  3. Use the following nested list recursion code template to follow the recursive structure of nested lists
  4. **Implement the functions base case**, using your first doctest example to test. Most base cases are pretty straightforward to implement, though this depends on the exact function you're writing
  5. Implement the function's recursive step by doing two things:
     - Use your **second doctest example** to write down the relevant sublists and recursive function calls (these are the second and third columns of the loop accumulation table we showed above). Fill in the recursive call output based on the function specification, not any code you have written!
     - **Analyze the output of the recursive calls and determine how to combine them to return the correct value for the original call.** This will almost certainly involve some aggregation of the recursive call return values.
- Principles for debugging
  - check the base cases and recursive cases separately
  - if there is a bug in a recursive case, the problem is **not** the recursive calls. It's what the code is doing with the results to the calls
- nested list depth

- Let x be a nested list. The **depth** of x is defined as the maximum number of times a list is nested within another list in x.
- Base case example and implementation

  Let $x \in \mathbb{Z}$. The depth is 0.

```python
def depth(x: int | list) -> int:
    if isinstance(nested_list, int):
        return 0
    else:
        ...
```

- Create an example for the recursive step

```python
>>> nested_list = [[1], 2, [[3,4]], []]
>>> depth(nested_list)
3
```

- Generalize and implement

  Suppose we have sublists $[a_0, a_1, \ldots, a_{k-1}]$ and depths $d_0, d_1, \ldots, d_{k-1}$

  The depth of $[a_0, a_1, \ldots, a_{k-1}]$ is $1 + \max\{d_0, d_1, \ldots, d_k - 1\}$

  or just 1 if the list is empty

$$depth(x) = \begin{cases} 0, & \text{if } x \in \mathbb{Z} \\ 1, & \text{if } x = [] \\ 1 + \max\{depth(a_i) \mid a_i \in x\}, & \text{if } x = [a_0, a_1, \ldots, a_{k-1}], k > 0 \end{cases}$$

```python
def depth(nested_list: int | list) -> int
    if isinstance(nested_list, int):
        return 0
    elif nested_list == []:
        return 1
    else:
        return 1 + max(depth(sublist) for sublist in nested_list)
```

- contains

```python
def nested_list_contains(nested_list: int | list, item: int) -> bool:
    """Return whether the given item appears in nested_list.
    If nested_list is an integer, return whether it is equal to item.
    >>> nested_list_contains(10, 10)
    True
    >>> nested_list_contains(10, 5)
    False
    >>> nested_list_contains([[1, [30], 40], [], 77], 50)
    False
    >>> nested_list_contains([[1, [30], 40], [], 77], 40)
    True
    """
    if isinstance(nested_list, int):
        return nested_list == item
    else:
        # Version 1 (comprehension and any)
        return any(nested_list_contains(sublist, item) for sublist in
nested_list)
        # Version 2 (loop and early return)
        # for sublist in nested_list:
        #     if nested_list_contains(sublist, item):
        #         return True
        #
        # return False
```

- item at depth

```python
def items_at_depth(nested_list: int | list, d: int) -> list[int]:
    """Return the list of all integers in nested_list that have depth d.
    Preconditions:
        - d >= 0
    >>> items_at_depth(10, 0)
    [10]
    >>> items_at_depth(10, 3)
    []
    >>> items_at_depth([10, [[20]], [[30], 40]], 0)
    []
    >>> items_at_depth([10, [[20]], [[30], 40]], 3)
    [20, 30]
    """
    # Exercise: try rewriting this code without using nested if
    # statements, and instead using compound if/elif conditions.
    if isinstance(nested_list, int):
        if d == 0:
            return [nested_list]
        else:
            return []
    else:
        if d == 0:
            return []
        else:
            result_so_far = []
            for sublist in nested_list:
                result_so_far.extend(items_at_depth(sublist, d - 1))
            return result_so_far
```

14.5 Recursive Lists

- recursive definition of a list
  - the empty list `[]` is a list
  - If `x` is a value and `r` is a list, then we can construct a new list `lst` whose first element is `x` and whose other elements are the elements of `r`
  - call `x` the first element of `lst`, and `r` the rest of `lst`

```
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
```

- $sum(lst) = \begin{cases} 0, & \text{if } lst \text{ is empty} \\ (\text{first of } lst) + sum(\text{rest of } lst), & \text{if } lst \text{ is non-empty} \end{cases}$

```
class RecursiveList:
    def sum(self) -> int:
        """Return the sum of the elements in this list.

        Preconditions:
            - every element in this list is an int
        """
        if self._first is None:  # Base case: this list is empty
            return 0
        else:                    # Recursive case: this list is non-empty
            return self._first + self._rest.sum()
```

- `sum` comparison

```
# 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
class RecursiveList:
    def sum(self) -> int:
        if self._first is None:
            return 0
        else:
            return self._first + self._rest.sum()
```

- `RecursiveList` and `_Node` classes have essentially the same structure

  → `_Node` is technically a **recursive** class
- `_Node`
  - representing a single list elements
  - `next` is a "link" to another `_Node`
- `RecursiveList`
  - representing an entire sequence of elements
  - `_rest` is the rest of the list itself
  - focus on how to use the result of that call in our computation

14.6 Application: Fractals

- fractal - a geometric figure that has been defined recursively

```
import pygame

# Define some colours using their RGB values
FOREGROUND = (255, 113, 41)
```

```python
BACKGROUND = (46, 47, 41)

# The minimum number of pixels in the Sierpinski triangle
MIN_SIDE = 3


def sierpinski(screen: pygame.Surface, v0: tuple[int, int], v1: tuple[int,
int],
               v2: tuple[int, int]) -> None:
    """Draw a Sierpinski Triangle on the given screen, with the given vertices.

    Each of v0, v1, and v2 is an (x, y) tuple representing a vertex of the
triangle.
    v0 is the lower-left vertex, v1 is the upper vertex, and v2 is the lower-
right vertex.
    """
    if v2[0] - v0[0] < MIN_SIDE:
        pygame.draw.polygon(screen, FOREGROUND, [v0, v1, v2])
    else:
        pygame.draw.polygon(screen, FOREGROUND, [v0, v1, v2])

        mid0 = midpoint(v0, v1)
        mid1 = midpoint(v0, v2)
        mid2 = midpoint(v1, v2)

        # Draw centre "sub-triangle"
        pygame.draw.polygon(screen, BACKGROUND, [mid0, mid1, mid2])

        # Recursively draw other three "sub-triangles"
        sierpinski(screen, v0, mid0, mid1)
        sierpinski(screen, mid0, v1, mid2)
        sierpinski(screen, mid1, mid2, v2)


def midpoint(p1: tuple[int, int], p2: tuple[int, int]) -> tuple[int, int]:
    """Return the midpoint of p1 and p2."""
    return ((p1[0] + p2[0]) // 2, (p1[1] + p2[1]) // 2)


if __name__ == '__main__':
    # Initialize a pygame window
    pygame.init()
    window = pygame.display.set_mode((800, 800))
    window.fill(BACKGROUND)

    # Draw the Sierpinski Triangle!
    sierpinski(window, (100, 670), (400, 150), (700, 670))

    # Render the image to our screen
    pygame.display.flip()

    # Wait until the user closes the Pygame window
    pygame.event.clear()
    pygame.event.set_blocked(None)
    pygame.event.set_allowed(pygame.QUIT)
    pygame.event.wait()
    pygame.quit()
```

15. Trees

15.1 Introduction to Trees

- Tree
    - **recursive** data structure
    - empty, or
    - has a **root value** connected to any number of other trees, called the **subtrees** of the tree
- **depth** - the distance between the item and the root of the tree, inclusive (root: depth 0)
- **size** - the number of values in the tree
- **leaf** - a value with **no subtrees**
- **internal value** - a value that has **at least one** subtree
- **height** - the length of the **longest** path from its root to one of its leaves
- **children**- all values directly connected **underneath** that value
- **descendants** - its children, the children of its children
- **parent** - the value immediately **above** and connected to it
- **ancestors** - its parent, the parent of its 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 consisting 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
```

15.2 Recursion on Trees

- $size(T) = \begin{cases} 0, & \text{if } T \text{ is empty} \\ 1 + \sum_{i=0}^{k-1} size(T_i), & \text{if } T \text{ has subtrees } T_0, T_1, \ldots, T_{k-1} \end{cases}$

```
class Tree:
    def __len__(self) -> int:
        """Return the number of items contained in this tree.

        >>> t1 = Tree(None, [])
        >>> len(t1)
        0
        >>> t2 = Tree(3, [Tree(4, []), Tree(1, [])])
        >>> len(t2)
        3
        """
        if self.is_empty():
            return 0
        else:
            # return 1 + sum(subtree.__len__() for subtree in self._subtrees)
            size_so_far = 1
            for subtree in self._subtrees:
                size_so_far += subtree.__len__()
            return size_so_far
```

- **Tree recursion code template**

```
class Tree:
    def method(self) -> ...:
        if self.is_empty():
            ...
        else:
            ...
            for subtree in self._subtrees:
                ... subtree.method() ...
            ...
```

- **Tree recursion code template(with size-one case)**

```python
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() ...
            ...
```

```python
class Tree:
    def __str__(self) -> str:
        """Return a string representation of this tree.
        """
        return self._str_indented(0)

    def _str_indented(self, depth: int = 0) -> str:
        """Return an indented string representation of this tree.

        The indentation level is specified by the <depth> parameter.
        """
        if self.is_empty():
            return ''
        else:
            str_so_far = '  ' * depth + f'{self._root}\n'
            for subtree in self._subtrees:
        # Note that the 'depth' argument to the recursive call is modified.
                str_so_far += subtree._str_indented(depth + 1)
            return str_so_far


>>> t1 = Tree(1, [])
>>> t2 = Tree(2, [])
>>> t3 = Tree(3, [])
>>> t4 = Tree(4, [t1, t2, t3])
>>> t5 = Tree(5, [])
>>> t6 = Tree(6, [t4, t5])
>>> print(t6)
6
  4
    1
    2
    3
  5
```

- `depth` is an optional parameter that can either be included or not included when this method is called
- Traversal orders
  - **left-to-right preorder** - first it visits the **root** value, then it recursively visits each of its subtrees, in **left-to-right** order
  - **left-to-right postorder** - first it visits every value in its **subtrees**, then visits the **root** value

```python
class Tree:
    def _str_indented_postorder(self, depth: int = 0) -> str:
        """Return an indented *postorder* string representation of this tree.

        The indentation level is specified by the <depth> parameter.
        """
        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
```

- first at depth

```python
class Tree:
    def first_at_depth(self, d: int) -> Optional[Any]:
        """Retrun the leftmost value at depth d in this tree.

        Return None if there are No items at depth d in this tree.

        Preconditions:
            - d >= 0
        """
        if self.is_empty():
            return None
        elif self._subtrees == []:
            if d == 0:
                return self._root
            else:
                return None
        else:
            if d == 0:
                return self._root
            for sub in self._subtrees:
                result = sub.first_at_depth(d - 1)
                if result is not None:
                    return result
        return None
```

15.3 Mutating Trees

- problems
    1. **doesn't return anything**, violating this method's type contract
    2. If one of the recursive calls successfully finds and deletes the item, no further subtrees should be modified

```python
class Tree:
    def remove(self, item: Any) -> bool:
        """..."""
        if self.is_empty():
            return False
        else:
            if self._root == item:
                self._delete_root()   # Delete the root
                return True
            else:
                for subtree in self._subtrees:
                    subtree.remove(item)
```

- solved - using return values of the recursive calls to determine whether the item was deleted from the current subtree.

```python
class Tree:
    def remove(self, item: Any) -> bool:
        """..."""
        if self.is_empty():
            return False
        elif self._root == item:
            self._delete_root()
            return True
        else:
            for subtree in self._subtrees:
                deleted = subtree.remove(item)
                if deleted:
                    # One occurrence of the item was deleted, so we're done.
                    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
```

- problem - if this tree has subtrees, then we can't set the `_root` attribute to `None`

```python
class Tree:
    def _delete_root(self) -> None:
        """Remove the root item of this tree.

        Preconditions:
            - not self.is_empty()
        """
        self._root = None
```

- solved

```python
class 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)
```

- problem - the result of doing `Tree.remove` is an empty tree - so its parent will contain an empty tree in its subtrees list

```
>>> t = Tree(10, [Tree(1, []), Tree(2, []), Tree(3, [])])  # A tree with leaves
1, 2, and 3
>>> t.remove(1)
True
>>> t.remove(2)
True
>>> t.remove(3)
True
>>> [subtree.is_empty() for subtree in t._subtrees]
[True, True, True]
```

- solved - in general it is **extremely dangerous** to remove an object from a list as you iterate through it

```
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
```

- Implicit assumptions are bad - there is no guarantee that this assumption will always hold for our trees
- use representation invariants

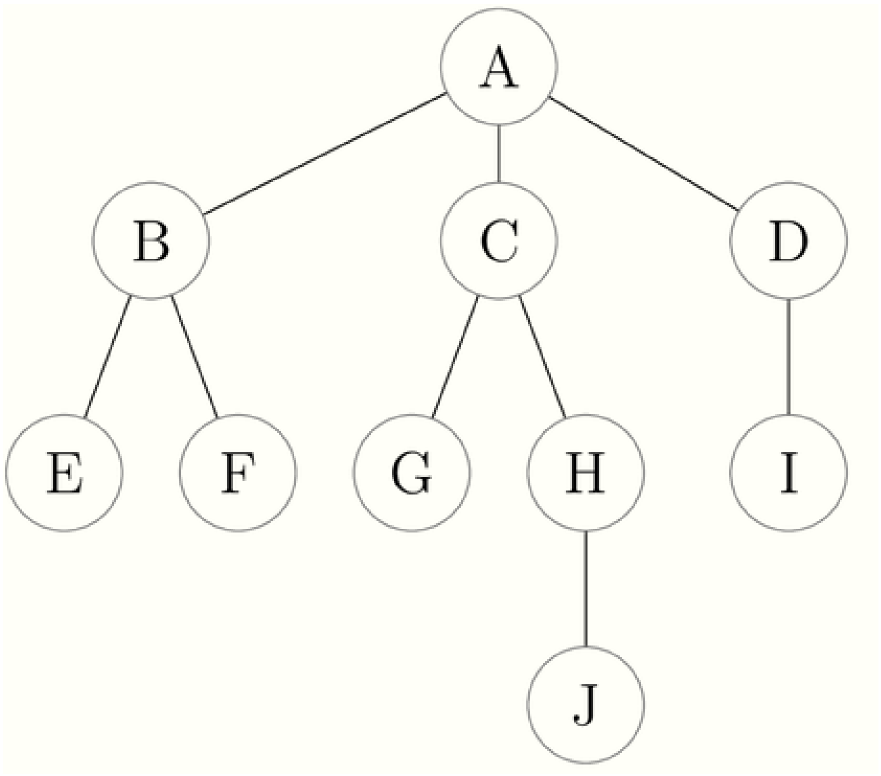
```
class Tree:
    """A recursive tree data structure.

    Representation Invariants:
        - self._root is not None or self._subtrees == []
        - all(not subtree.is_empty() for subtree in self._subtrees)  # NEW
    """
```

- delete root(version 2)

```
class Tree
    def _delete_root(self) -> None:
        """Remove the root of this tree.
        Preconditions:
            - not self.is_empty()
        """
        if self._subtrees == []:
            self._root = None
            # If self has size one, we turn it into an empty tree
        else:
            # Strategy 2: Move a leaf
            # Need to update this method as well to preserve the "no empty
            # subtrees" representation invariant!
            self._root = self._extract_leaf()

    def _extract_leaf(self) -> Any:
        """Remove and return the leftmost leaf in this tree.
        Preconditions:
            - not self.is_empty()
        """
        if self._subtrees == []:
            # Similar to _delete_root, but also returns the root value
            root = self._root
            self._root = None
            return root
        else:
            return self._subtrees[0]._extract_leaf()
            # Recurse on leftmost subtree
```

15.4 Runinning-Time Analysis for Tree Operations

- `_len_`

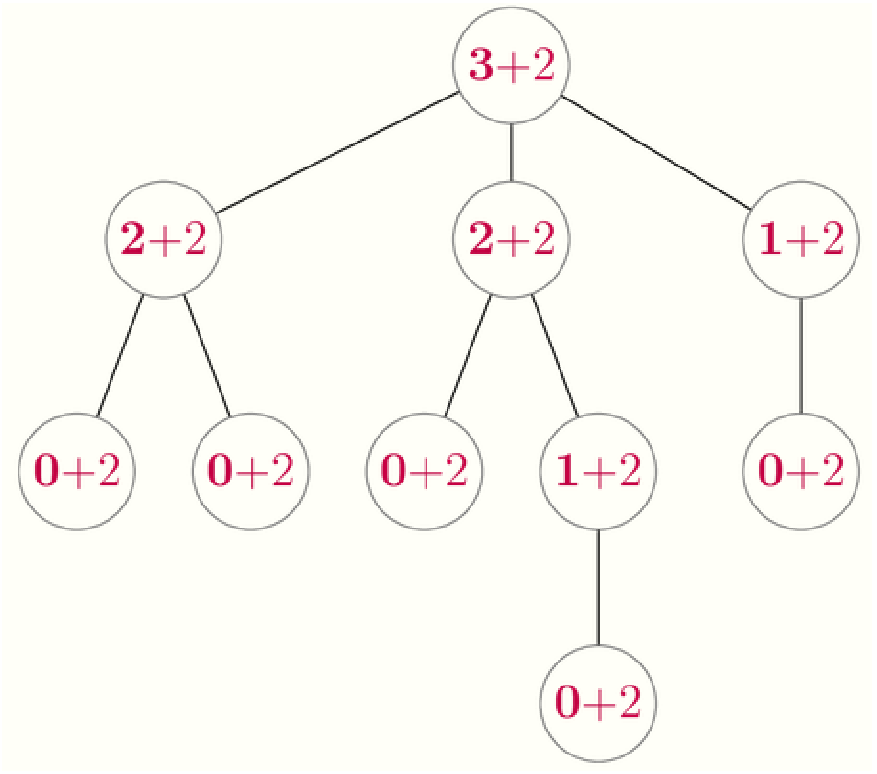


  - recursive calls
    - Initial call (A) makes **three recursive calls** on each of its subtrees (B),(C) and (D)
      - (B) - **two** on (E) and (F)
        - (E) and (F) - leaf - no more recursive calls - **zero**
      - (C) - **two** on (G) and (H)
        - (G) - leaf - no more recursive calls - **zero**
        - (H) - **one** on (J)

                - (J) - leaf - no more recursive calls - **zero**
            - (D) - **one** on (I)
                - (I) - leaf - no more recursive calls - **zero**
    - makes a recursive call on every subtree - **ten** calls, including the intial function call on the whole tree
    - <u>the structure of the recursive callls exactly follow the structure of the tree</u>
- "non-recursive" part of each call

    ```
            else:
                size_so_far = 1
                for subtree in self._subtrees:
                    size_so_far += subtree.__len__()
                return size_so_far
    ```

    - assuming each recursive call takes constant time
    - **one** step `size_so_far = 1`
    - k step for the loop, where k is **the number of subtrees in the tree** (NOTE: counting the loop body as just 1 step)
    - **one** step for the return statement
    - total $k + 2$ steps



- add up the "k" and "$+2$" values separately

$$\begin{aligned}
&\mathbf{3}+2 && \text{(A)}\\
+&\mathbf{2}+2 && \text{(B)}\\
+&\mathbf{2}+2 && \text{(C)}\\
+&\mathbf{1}+2 && \text{(D)}\\
+&\mathbf{0}+2 && \text{(E)}\\
+&\mathbf{0}+2 && \text{(F)}\\
+&\mathbf{0}+2 && \text{(G)}\\
+&\mathbf{1}+2 && \text{(H)}\\
+&\mathbf{0}+2 && \text{(I)}\\
+&\mathbf{0}+2 && \text{(J)}\\
\hline
=&\mathbf{9}+\mathbf{20}\\
=&\mathbf{29}
\end{aligned}$$

  - the sum of all the subtrees is 9, the 20 is the constant number of steps (2) multiplied by the number of recursive calls
- generlization
  - let $n \in \mathbb{Z}^+$, and suppose we have a tree of size n. We know that there will be n recursice calls made
    - the "constant time" parts will take $2n$ steps across all n recursive calls

- the total number of staps taken by the for loop across all recursive callls is equal to the sum of all of the numbers of children of each node, which is $n - 1$
- total running time $2n + (n - 1) = 3n - 1 \rightarrow \Theta(n)$

15.5 Introduction to Binary Search Trees

- **Multiset**
  - **Data** - an unordered collection of values, **allowing duplicates**
  - **Operations** - get size, insert a value, remove one occurrence of a specified value, check membership in the multiset
- **binary search** - searching $\Theta(\log n)$
- **binary tree** - a tree in which every item has **two** (possibly empty) subtrees, which are labeeled its left and right subtrees
- **binary search tree property** - the value of an item is **greater than or equal to** all items in its **left** subtree, and **less than or equal to** all items in its **right** subtree
- **binary search tree (BST)** - when every item in the tree satisfies the binary search tree property
- an empty BST - `_root` - `None`, `_left` and `_right` - `None` instead of `[]`
  - an empty BST is the **only** case where any of the attributes can be `None`
  - the attributes `_left` and `_right` of leaves should be **binary search trees** rather than `None`

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

      Representation Invariants:
        - (self._root is None) == (self._left is None)
        - (self._root is None) == (self._right is None)
      """
  ```

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

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

- searching
    - the initial comparison to the root tells you which subtree you need to check

    ```
    class BinarySearchTree:
        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)
    ```

15.6 Mutating Binary Search Trees

- Deletion

```
class BinarySearchTree:
    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:
        """Remove the root of this tree.
```

```
        Preconditions:
          - not self.is_empty()
        """
        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()
```

15.7 The Running Time of Binary Search Tree Operations

- contains

```python
class BinarySearchTree:
    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)  # or, item in self._left
        else:
            return self._right.__contains__(item)  # or, item in self._right
```

  - only recurses on **one** subtree
- Recursion diagram(upper bound) - at most $h+1$, h is the height
  - non-recursive running time is just **1** step
  - total(upper bound) - execute **at most** $h+1$ calls and each call has a non-recursive running time of 1 step → **at most** $h+1$ steps, which is $\mathcal{O}(h)$.
  - total(lower bound) - pick a tree of height h and search for the deepest leaf in the tree, using the same analysis, the results in h steps, which is $\Omega(h)$
- Let B be a binary search tree with height h and size n. Then $n \leq 2^h - 1$.
- All three BST operations (search, insert, delete) have a worst-case running time of $\Theta(h) = \Theta(\log n)$

Operation	Unsorted list	Sorted list	Tree	Binary search tree
search	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(h)$
insert	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(h)$
delete	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(h)$

16. Abstract Syntax Trees

16.1 Introduction to Abstract Syntax Trees

- as human, read and write code as text
- python interpreter - taking file and running it
- work directly with strings is hard
- strings are a **linear** structure (a sequence of characters)
- naturally recursive structure - if statements, for loops (potential to be arbitrarily nested)
- **Abstract Syntax Tree (AST)** - use a tree based data structure to represent recursice program

the `Expr` class

- basic building blocks of the language

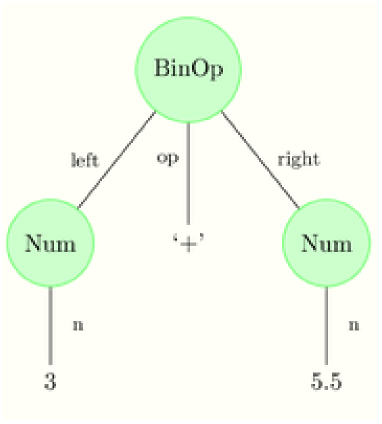
```python
class Expr:
    """An abstract class representing a Python expression.
    """
    def evaluate(self) -> Any:
        """Return the *value* of this expression.

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

class Num(Expr):
    """A numeric literal.

    Instance Attributes:
        - n: the value of the literal
    """
    n: int | float

    def __init__(self, number: int | float) -> None:
        """Initialize a new numeric literal."""
        self.n = number
    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 = Num(10.5)
        >>> expr.evaluate()
        10.5
        """
        return self.n  # Simply return the value itself!
```

- `BinOp` - arithmetic operation
  - left subexpression (operand of the expression)
  - right subexpression (operand of the expression)
  - operator

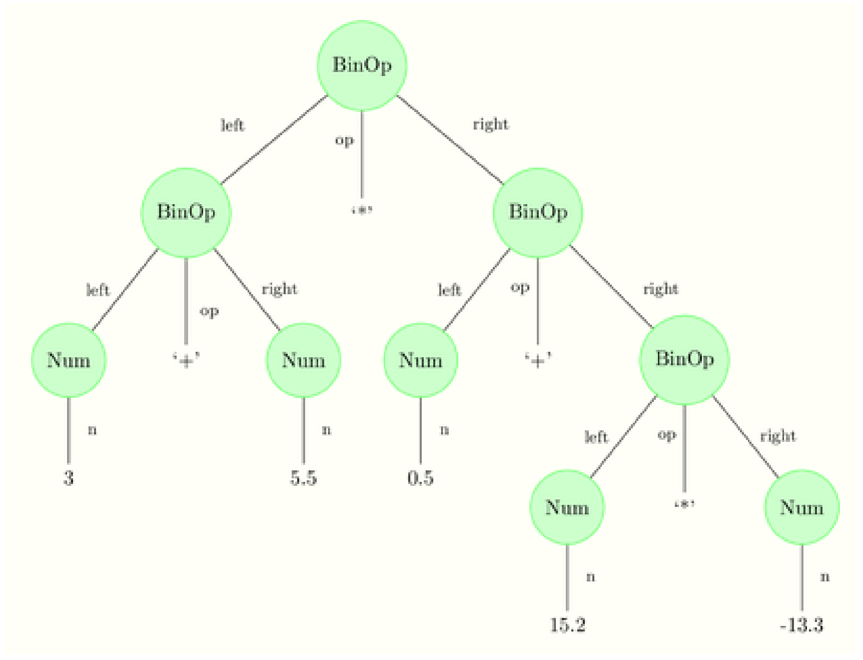
```
class BinOp(Expr):
    """An arithmetic binary operation.

    Instance Attributes:
        - left: the left operand
        - op: the name of the operator
        - right: the right operand

    Representation Invariants:
        - self.op in {'+', '*'}
    """
    left: Expr
    op: str
    right: Expr

    def __init__(self, left: Expr, op: str, right: Expr) -> None:
        """Initialize a new binary operation expression.

        Preconditions:
            - op in {'+', '*'}
        """
        self.left = left
        self.op = op
        self.right = right
```

- `BinOp` calss is basically a binary tree
  - its `left` and `right` attributes aren't `Num`s
  - make this data type recursive
  - the root is the name of its class

```
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)))
```



- to evaluate a binary operation, first evaluate its operands, and then combine them using the operater

```python
class BinOp:
    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()

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

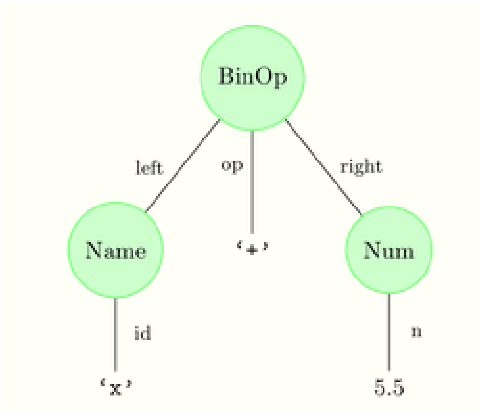
- base case - call `Num.evaluate`

16.2 Variables and the Variable Environment

- `Name` - class for the variables

```
class Name(Expr):
    """A variable expression.

    Instance Attributes:
        - id: The variable name.
    """
    id: str

    def __init__(self, id_: str) -> None:
        """Initialize a new variable expression."""
        self.id = id_
```

```
# x + 5.5
BinOp(Name('x'), '+', Num(5.5))
```



- requires a **mapping** between variable names and values (use a `dict`)
- **variable environment** - each key-value pair in the environment a **binding** between a variable and its current value

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

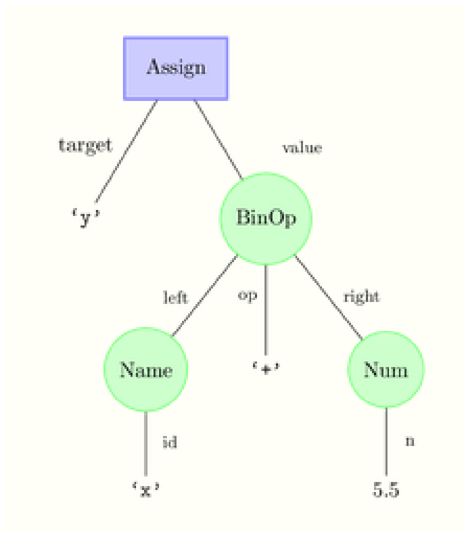
```python
        """Evaluate this statement with the given environment.

        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:
    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
```

16.3 From Expressions to Statements

```
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 mightonly 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.
    """


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.
        """
```

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

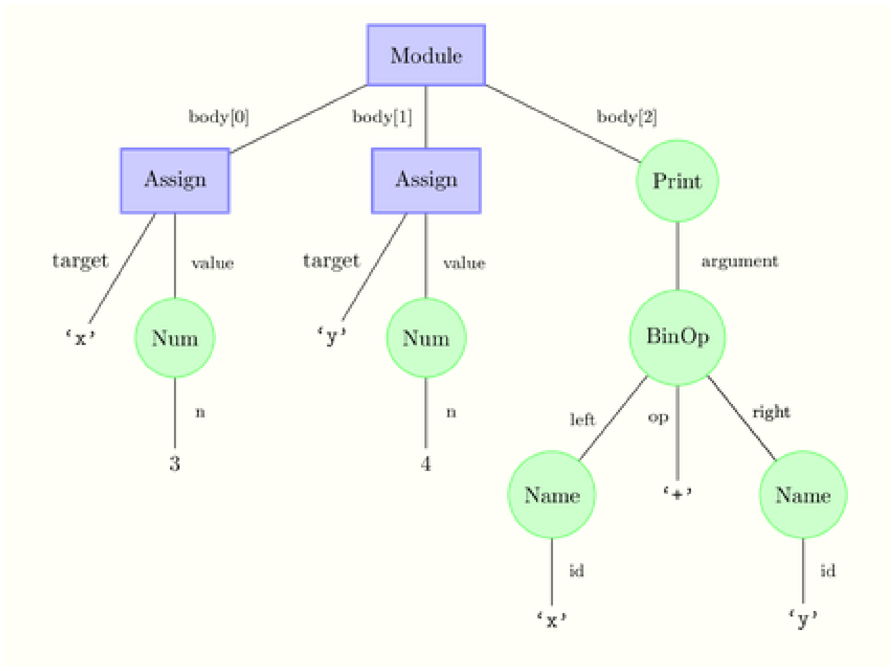


```
class Assign:
    def evaluate(self, env: dict[str, Any]) -> None:
        """Evaluate this statement with the given environment.
        """
        env[self.target] = self.value.evaluate(env)
```

```python
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))
```

```python
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
```

```
x = 3
y = 4
print(x + y)

Module([
    Assign('x', Num(3)),
    Assign('y', Num(4)),
    Print(BinOp(Name('x'), '+', Name('y')))
])
```



- `Module` is **not** a subclass of `Statement`, is the **root** of a complete abstract syntax tree

```
class Module:
    def evaluate(self) -> None:
        """Evaluate this statement with the given environment.
        """
        env = {}
        for statement in self.body:
            statement.evaluate(env)
```

- `If`
    - evaluate the if condition

- True - evaluate the body; False - evaluate the else

```python
class If(Statement):
    """An if statement.

    This is a statement of the form:

        if <test>:
            <body>
        else:
            <orelse>

    Instance Attributes:
        - test: The condition expression of this if statement.
        - body: A sequence of statements to evaluate if the condition is True.
        - orelse: A sequence of statements to evaluate if the condition is
False.
        (This would be empty in the case that there is no `else` block.)

    # if x < 100:
    #     print(x)
    # else:
    #     y = x + 2
    #     x = 1
    >>> If(
    ...     Compare(Name('x'), [('<', Num(100))]),
    ...     [Print(Name('x'))],
    ...     [Assign('y', BinOp(Name('x'), '+', Num(2))),
    ...     Assign('x', Num(1))]
    ... )
    """
    test: Expr
    body: list[Statement]
    orelse: list[Statement]

    def __init__(self, test: Expr, body: list[Statement],
                 orelse: list[Statement]) -> None:
        self.test = test
        self.body = body
        self.orelse = orelse
    def evaluate(self, env: dict[str, Any]) -> None:
    """Evaluate this statement.

    Preconditions:
        - self.test evaluates to a boolean

    >>> stmt = If(Bool(True),
    ...           [Assign('x', Num(1))],
    ...           [Assign('y', Num(0))])
    ...
    >>> env = {}
    >>> stmt.evaluate(env)
    >>> env
    {'x': 1}
    """
    # 1. Evaluate the test (if condition).
    test_val = self.test.evaluate(env)

    # 2. If it's true, evaluate the statements in the body (if branch).
    # Otherwise, evaluate the statements in orelse (else branch).
    if test_val:
        for statement in self.body:
            statement.evaluate(env)
    else:
        for statement in self.orelse:
            statement.evaluate(env)
```

- for loop over a range
  - evaluate `<start>` and `<stop>`
  - assign a new variable `<variable>` to the value of `<start>`
  - execute the `<body>` statements
  - reapeat steps 2 and 3 once for each number between the range

```
class ForRange(Statement):
    """A for loop that loops over a range of numbers.

        for <target> in range(<start>, <stop>):
            <body>
    Instance Attributes:
        - target: The loop variable.
        - start: The start for the range (inclusive).
        - stop: The end of the range (this is *exclusive*, so <stop> is not
        included in the loop).
        - body: The statements to execute in the loop body.

    # sum_so_far = 0

    #     for n in range(1, 10):
    #         sum_so_far = sum_so_far + n

    # print(sum_so_far)

    >>> assign = Assign('sum_so_far', BinOp(Name('sum_so_far'), '+',
Name('n')))
    >>> Module([
    ...     Assign('sum_so_far', Num(0)),
    ...     ForRange('n', Num(1), Num(10),
    ...             [assign]),
    ...     Print(Name('sum_so_far'))
    ... ])
    """
    target: str
    start: Expr
    stop: Expr
    body: list[Statement]

    def __init__(self, target: str, start: Expr, stop: Expr, body:
list[Statement]) -> None:
        """Initialize a new ForRange node."""
        self.target = target
        self.start = start
        self.stop = stop
        self.body = body

    def evaluate(self, env: dict[str, Any]) -> None:
        """Evaluate this statement.
        >>> statement = ForRange('x', Num(1), BinOp(Num(2), '+', Num(3)),
        ... [Print(Name('x'))])
        >>> statement.evaluate({})
```

```
1
2
3
4
"""
# 1. Evaluate start and stop
start_value = self.start.evaluate(env)
stop_value = self.stop.evaluate(env)

for i in range(start_value, stop_value):
    # 2. Assign a new variable <target> to the value of <start>.
    # env[self.target] = i
    Assign(self.target, Num(i)).evaluate(env)

    # 3. Execute the <body> statement(s).
    for statement in self.body:
        statement.evaluate(env)
```

17. Graphs

17.1 Introduction to Graphs

- A **graph** is a pair of sets (V, E), which are defined as follows:
  - V is a set of objects. Each element of V is called a **vertex** of the graph, and V itself is called the set of **vertices** of the graphs
  - E is a set of pairs of objects from V, where each pair $\{v_1, v_2\}$ is a set consisting of two distinct vertices - i.e., $v_1, v_2 \in V$ and $v_1 \neq v_2$ - and is called an **edge** of the graph.
  - Order does not matter in the pairs, and so $\{v_1, v_2\}$ and $\{v_2, v_1\}$ represent the same edge.
- **adjacent/neighbor** - $G = (V, E)$ and let $v_1, v_2 \in V$. There exists an edge between v_1 and v_2
- **degree/ d(v)** - v's number of neighbours. equivalently, how many edges v is a part of
- **path** - a sequence of *distinct* vertices $v_0, v_1, \ldots, v_k \in V$ which satisfy the following properties
  - $v_0 = u$ and $v_1 = u'$

    - each consecutive pair of vertices are adjacent
    - $k = 0$ → path just a single vertex
- **length** - the number of **edges** which are used by this sequence
- **connected**
    - vertices - there exists a path between the vertices
        - NOTE: single vertex is always connected to itself
    - graph - for all pairs of vertices $u, v \in V$, u and v are connected.

17.2 Some Properties of Graphs

> - For all graphs $G = (V, E), |E| \leq \frac{|v|(|v|-1)}{2}$
> $\forall n \in \mathbb{N}, \forall G = (V, E), |V| = n \implies |E| \leq \frac{n(n-1)}{2}$

Proof. We'll prove this statement by induction on n. Our predicate is

$$P(n) : \forall G = (V, E), |V| = n \implies |E| \leq \frac{n(n-1)}{2} \quad \text{where} n \in \mathbb{N}$$

Base case: Let $n = 0$

Let $G = (V, W)$ be an arbitrary graph, and assume that $|V| = 0$. In this case, the graph has no vertices, and so cannot have any edges. Therefore $|E| = 0$, and satisfies the inequality $|E| \leq \frac{0(0-1)}{2}$.

Induction step: let $k \in \mathbb{N}$ and assume that $P(k)$ holds: every graph with k vertices has at most $\frac{k(k-1)}{2}$ edges. We need to prove $P(k+1)$.

Let $G = (V, E)$ be an arbitrary graph, and assume that $|V| = k + 1$. We want to prove that $|E| = \frac{(k+1)k}{2}$.

Let v be a vertex in V. We can divide the edges of G into two groups:

- E_1, the set of edges that contain v. Since there are k other vertices in V that v could be adjacent to, $|E_1| \leq k$.
- E_2, the set of edges that do not contain v. To count these edges, suppose we remove v from the graph G, to obtain a new graph G'. Then E_2 is exactly the set of edges of G'.

  But since G' has one fewer vertex than G, we know G' has k vertices. By the induction hypothesis, we know that G' has at most $\frac{k(k-1)}{2}$ edges, so $|E_2| \leq \frac{k(k-1)}{2}$.

Putting this together, we have

$$\begin{aligned}|E| &= |E_1| + |E_2| \\ &\leq k + \frac{k(k-1)}{2} \\ &= \frac{(k+1)k}{2}\end{aligned}$$

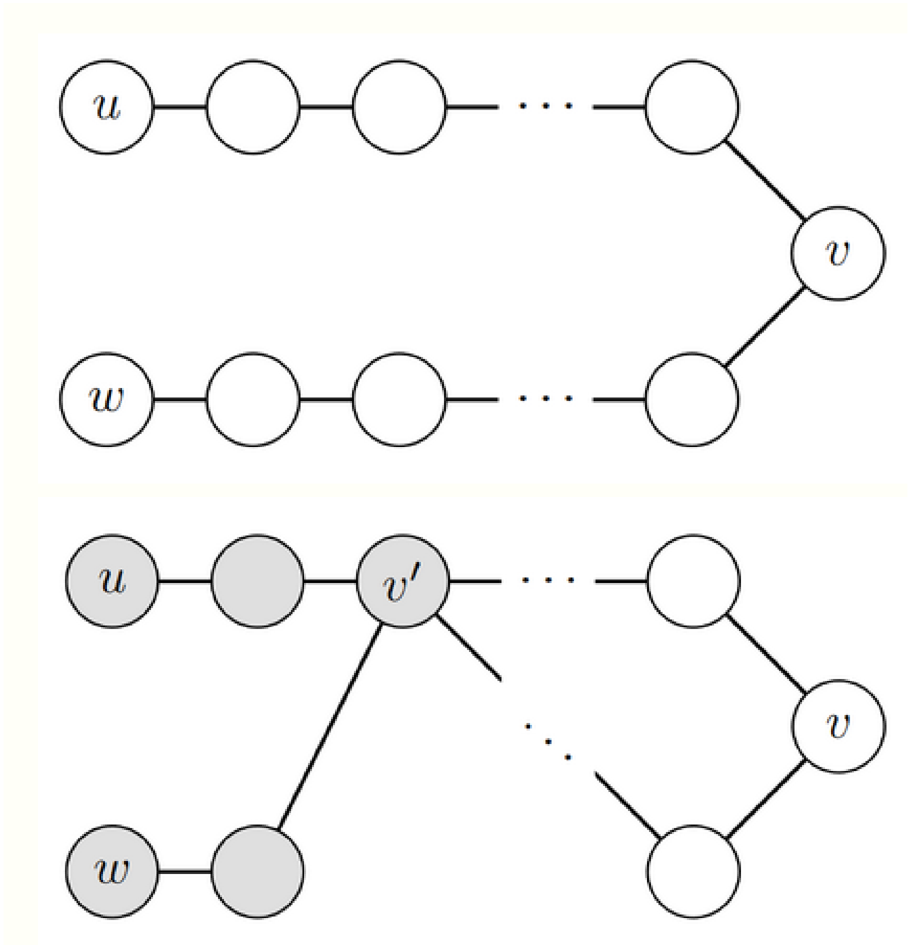
- > $Conn(G, u, v) =$ "u and v are connected vertices in G"
  >
  > $\forall G = (V, E), \forall u, v, w \in V, (Conn(G, u, v) \land Conn(G, v, w)) \implies Conn(G, u, w)$

Proof. Let $G = (V, E)$ be a graph, and $u, v, w \in V$. Assume that u and v are connected, and v and w are connected. We want to prove that u and w are connected.

Let P_1 be a path between u and v, and P_2 be a path between v and w.

Handling multiple shared vertices: Let $S \subseteq V$ be the set of all vertices which appear on both P_1 and P_2. Note that this set is not empty, because $v \in S$. Let v' be the vertex in S which is *closest* to u in P_1. This means that *no* vertex in P_1 between u and v' is in S, or in other words, is also in P_2.

Finally, let P_3 be the path formed by taking the vertices in P_1 from u to v', and then the vertices in P_2 from v' to w. Then P_3 has no duplicate vertices and is indeed a path between u and w.



- > For all graphs $G = (V, E)$, $\sum_{v \in V} d(v) = 2 \cdot |E|$.

proof. Let $G = (V, E)$ be an arbitrary graph.

For a vertex v, $d(v)$ is the number of edges that "touch" v.\

But each edge touches exactly 2 vertices.

If we sum up $d(v)$ for every vertex, each edge is counted exactly twice.

Therefore, the total degree is equal to twice the number of edges.

- > for all graphs $G = (V, E)$, if $|V| \geq 2$ then there exist two vertices in V that have the same degree.
  >
  > $\forall G = (V, E), |V| \geq 2 \implies (\exists v_1, v_2 \in V, d(v_1) = d(v_2))$

Proof. Assume for a contradiction that this statement is False, i.e., that there exists a graph $G = (V, E)$ such that $|V| \geq 2$ and all of the vertices in V have a different degree. We'll derive a contradiction from this. We also let $n = |V|$.

First, let v be an arbitrary vertex in V. We

know that $d(v) \geq 0$, and because there are $n-1$ other vertices not equal to v that could be potential neighbours of v, $d(v) \leq n-1$. So every vertex in V has degree between 0 and $n-1$, inclusive.

Since there are n different vertices in V and each has a different degree, this means that **every** number in $\{0, 1, \ldots, n-1\}$ must be the degree of some vertex. In particular, there exists a vertex $v_1 \in V$ such that $d(v_1) = 0$, and other vertex $v_2 \in V$ such that $d(v_2) = n-1$

Then on the one hand, since $d(v_1) = 0$, it is not adjacent to any other vertex, and so $\{v_1, v_2\} \notin E$.

But on the other hand, since $d(v_2) = n-1$, it is adjacent to every other vertex, and so $\{v_1, v_2\} \in E$.

So both $\{v_1, v_2\} \notin E$ and $\{v_1, v_2\} \in E$ are True, which gives us our contradiction.

17.3 Representing Graphs in Python

```python
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.
    Representation Invariants:
        - self not in self.neighbours
        - all(self in u.neighbours for u in self.neighbours)
    """
    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
```

```
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
graph.
           Raise a ValueError if item1 or item2 do not appear as vertices in
this graph.

        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

    def num_edges(self) -> int:
        """Return the number of edges in this graph."""
        # Calculate the sum of the vertex degrees
        sum_so_far = 0
        # for vertex in self._vertices.values():
        for item in self._vertices:
        vertex = self._vertices[item]
        sum_so_far += len(vertex.neighbours)
        return sum_so_far // 2
        # Or, using a comprehension
        # sum_degrees = sum(len(self._vertices[item].neighbours)
        # for item in self._vertic)

    def complete_graph(n: int) -> Graph:
        graph_so_far = Graph()
        for i in range(0, n):
        # Add new vertex for i
        graph_so_far.add_vertex(i)
        # Add edges to all previous vertices (0 <= j < i)
        for j in range(0, i):
        graph_so_far.add_edge(i, j)

        # Alternate (adding vertices and edges separately)
        # Add all vertices first
        for i in range(0, n):
        # Add new vertex for i
        graph_so_far.add_vertex(i)
        # Add all edges
        for i in range(0, n):
        # Add edges to all previous vertices (0 <= j < i)
        for j in range(0, i):
        graph_so_far.add_edge(i, j)

        return graph_so_far
```

17.4 Connectivity and Recursive Graph Traversal

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

        Return False if item1 or item2 do not appear as vertices in this
graph.

        >>> g = Graph()
        >>> g.add_vertex(1)
        >>> g.add_vertex(2)
        >>> g.add_vertex(3)
        >>> g.add_vertex(4)
        >>> g.add_edge(1, 2)
        >>> g.add_edge(2, 3)
        >>> g.connected(1, 3)
        True
        >>> g.connected(1, 4)
        False
        """
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            return v1.check_connected(item2)
        else:
            return False
```

- Definition (connectivity, recursive)

> Let $G = (V, E)$ be a graph, and let $v_1, v_2 \in V$. We say that v_1 and v_2 are **connected** when:
>
> - $v_1 = v_2$ or
> - there exists a neighbour u of v_1 such that u and v_2 are connected by a path that does not use v_1

```
class _Vertex:
    def check_connected(self, target_item: Any) -> bool:
        """Return whether this vertex is connected to a vertex corresponding to
the target_item.
        """
        if self.item == target_item:
            # Our base case: the target_item is the current vertex
            return True
        else:
            for u in self.neighbours:
                if u.check_connected(target_item):
                    return True

            return False
```

- problem
    - `RecursionError`
    - **infinite recursion** - a recursive computation that does not stop by reaching a base case
- solution
    - keep track of the items that have been already visited so that do not visit the same vertex more than once

```
class Graph:
    def connected(self, item1: Any, item2: Any) -> bool:
        """..."""
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            # Pass in an empty "visited" set
            return v1.check_connected(item2, set())
        else:
            return False
class _Vertex:
    def check_connected(self, target_item: Any, \
                        visited: set[_Vertex]) -> bool:
        """Return whether this vertex is connected to a vertex corresponding
to the 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:
            # Add self to the set of visited vertices
            new_visited = visited.union({self})
            for u in self.neighbours:
                # Only recurse on vertices that haven't been visited
                if u not in new_visited:
                    if u.check_connected(target_item, new_visited):
                        return True

            return False
```

- alternative version

```python
class _Vertex:
    def check_connected(self, target_item: Any, visited: set[_Vertex]) -> bool:
        """Return whether this vertex is connected to a vertex corresponding to
the 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:
            # Add self to the set of visited vertices
            visited.add(self)
            for u in self.neighbours:
                # Only recurse on vertices that haven't been visited
                if u not in visited:
                    if u.check_connected(target_item, visited):
                        return True

            return False
```

- once we've established that `u0` is not connected to the target item, we **shouldn't** recurse on it ever again
- WARNING - whenever you use recursion with a mutable argument, be very careful when choosing whether to mutate the argument or create a modifies copy - if you choose to mutate the argument, know that **all recursive calls will mutate it as well**

17.5 A Limit for Connectedness

- a graph is connected -

> $$\forall n \in \mathbb{Z}^+, n > 1 \implies (\exists G = (V, E), |V| = n \wedge |E| = \frac{(n-1)(n-2)}{2} \wedge G \text{ is not connected})$$

proof. Let $n \in \mathbb{Z}^+$, and assume $n > 1$.

Let $G = (V, E)$ be the graph defined as follows:

- $V = \{v_1, v_2, \ldots, v_n\}$
- $E = \{\{v_i, v_j\} i, j \in \{1, \ldots, n-1\} \text{ and } i < j\}$. That is E consists of all edges between the first $n-1$ vertices and has no edges connected to v_n

We need to show three things:

1. $|V| = n$
2. $|E| = \frac{(n-1)(n-2)}{2}$
3. G is not connected

For (1), we have labelled the n vertices in V, and so it is clear that $|V| = n$.

For (2), we have chosen all possible pairs of vertices from $\{v_1, v_2, \ldots, v_{n-1}\}$. There are exactly $\frac{(n-1)(n-2)}{2}$ edges.

For (3), because v_n is not adjacent to any other vertex, it cannot be connected to any other vertex. So G is not connected.

- filling in the blank

> Let $n \in \mathbb{Z}^+$. For all graphs $G = (V, E)$, if $|V| = n$ and $|E| \geq \frac{(n-1)(n-2)}{2} + 1$, then G is connected.

proof. We will proceed by induction on n. More precisely, define the following predicate over the positive integers:

$$P(n) : \forall G = (V, E), (|V| = n \land |E| \geq \frac{(n-1)(n-2)}{2} + 1) \implies G \text{ is connected.}$$

Base Case: Let $n = 1$.

$$P(1) : \forall G = (V, E), (|V| = 1 \land |E| \geq 1) \implies G \text{ is connected.}$$

This statement is vacuously true: no graph exists that has only one vertex and at least one edge, since an edge requires two vertices

Inductive Step: Let $k \in \mathbb{Z}^+$, and assume that $P(k)$ holds. We need to prove that $P(k+1)$ also holds.

Let $G = (V, E)$ and assume that $V = k + 1$ and $|E| \geq \frac{k(k-1)}{2} + 1$.

Case 1. Assume $|E| = \frac{(k+1)(k)}{2}$, i.e. G has all possible edges. In this case, G is certainly connected.

Case 2. Assume $|E| < \frac{(k+1)(k)}{2}$. We now need to prove the following claim.

G has a vertex in G with between one and $K - 1$ neighbours, inclusive.

Since G has fewer than the maximum number of possible edges, there exists a vertex (u, v) which is **not** an edge. Both u and v have at most $k - 1$ neighours, since there are $k - 1$ vertices in G other than these two.

Claim. both u and v have at least one neighbour.

Assume u and v has no neighbour.

Then, for the rest $k - 1$ vertices, there are at most $\frac{(k-1)(k-2)}{2}$ edges.

But we assume $|E| \geq \frac{k(k-1)}{2} + 1$ edges.

Therefore, we still need at least $\frac{k(k-1)}{2} + 1 - \frac{(k-1)(k-2)}{2} = k$ edges, which means that one of u and v should have neighbours.

Assume v has no neighbour. Then, u can have at most $k - 1$ neighbours, which can at most create $k - 1$ edges. We still need one more edge.

Therefore, v must have at least one neighbour.

Assume u has no neighbour, similarly, we will conclude that u must have at least one neighbour.

Continue. Use this claim, we let v be a vertex which has at most $k - 1$ neighbours.

Let $G' = (V', E')$ be the graph which is formed by taking G and removing v from V, and all edges in E which use v. Then $|V'| = |V| - 1 = k$.

$$\begin{aligned}
|E'| &= |E| - \text{number of removed edges} \\
&\geq |E| - (k-1) \\
&\geq \frac{k(k-1)}{2} + 1 - (k-1) \\
&= \frac{(k-2)(k-1)}{2} + 1
\end{aligned}$$

Therefore, by the induction hypothesis, G' is connected.

Since any two vertices not equal to v are connected in G because they are connected in G'. Since v has at least one neighbour, so call it w. Then v is connected to w, but because G' is connected, w is connected to every other vertex in G. By the **transitivity of connectedness**, we know that v must be connected to all of these other vertices.

17.6 Cycles and Trees

- > Any connected graph $G = (V, E)$ must have $|E| \geq |V| - 1$.

- Let $G = (V, E)$ be a graph. A **cycle** in G is a sequence of vertices $v_0, \ldots, v_k$ satisfying the following conditions:
  - $k \geq 3$
  - $v_0 = v_k$, and all other vertices are distinct from each other and v_0.
  - each consecutive pair of vertices is adjacent
  - Cycles are **connectedness redundancy** in a graph.

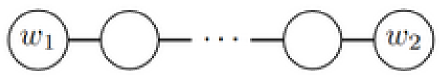
> Let $G = (V, E)$ be a graph and $e \in E$. If G is connected and e is in a cycle of G, then the graph obtained by removing e from G is still connected.
>
> $$\forall G = (V, E), \forall e \in E, (G \text{ is connected} \land e \text{ is in a cycle of } G \implies G - e \text{ is connected})$$

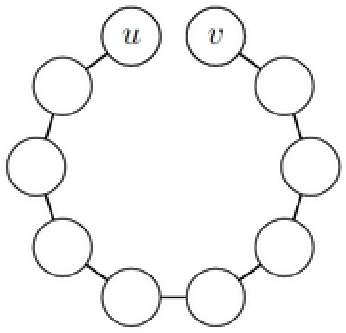
proof. Let $G = (V, E)$ be a graph, and $e \in E$ be an edge in the graph. Assume that G is connected and that e is in a cycle. Let $G' = (V, E\{e\})$ be the graph formed from G by removing edge e. We want to prove that G' is also connected.

Let $w_1, w_2 \in V$. By our assumption, we know that w_1 and w_2 are connected in G. We want to show that they are also connected in G'.

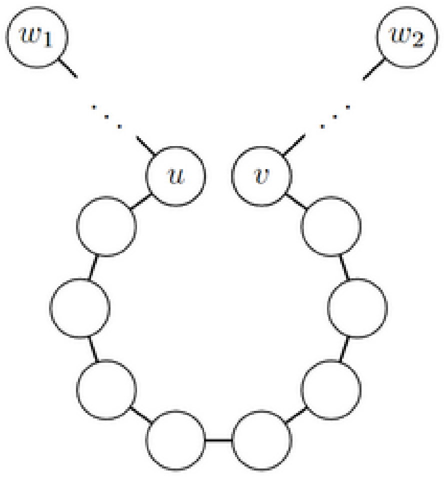
Let P be a path between w_1 and w_2 in G. We divide the path P into two cases

Case 1: P does not contain the edge e. Then P is a path in G' as well (since the only edge that was removed is e).





Case 2: P does contain the edge e. Let u be the endpoint of e which is closer to w_1 on the path P, and let v be the other endpoint.



This means that we can divide the path P into three parts:

- P_1, the part from w_1 to u, the edge $\{u, v\}$, and then
- P_2, the part from v to w_2

Since P_1 and P_2 cannot use the edge $\{u, v\}$ - no duplicates - they must be paths in G'.

So then w_1 is connected to u in G', and w_2 is connected to v in G'.

But we know that u and v are also connected in G' since they were part of the cycle

so by the transitivity of connectedness, w_1 and w_2 are connected in G'

- The graph $G = (V, E)$ is a **tree** when it is connected and has no cycles
- > If G doesn't have a cycle, then there does not exist an edge e in G such that $G - e$ is connected
  >
  > $\forall G = (V, E), G \text{ does not have a cycle } \implies \neg(\exists e \in E, G - e \text{ is connected}).$

> $\forall G=(V,E), (\exists e \in E, G-e \text{ is connected}) \implies G \text{ has a cycle.}$

proof. Let $G=(V,E)$ be a graph. Assume that there exists an edge $e \in E$ such that $G-e$ is still connected.

Let $G' = (V, E \setminus \{e\})$ be the graph obtained by removing e from G. Our assumption is that G' is connected.

Let u and v be the endpoints of e. By the definition of connectedness, there exists a path P in G' between $u$and v;
this path does not use e, since e isn't in G'.

Then taking the path P and adding the edge e o it is a cycle in G.

- > Let G be a tree. Then removing any edge from G results in a graph that is not connected.

proof. This follows directly from the previous lemma. By definition, G does not have any cycles, and so there does not exist an
edge that can be removed from G without disconnecting it.

- **Theorem (Number of edges in a tree)** Let $G=(V,E)$ be a tree. Then $|E| = |V| - 1$.

- > Let $G=(V,E)$ be a tree. If $|V| \geq 2$, then G has a vertex with degree one.
  > $\forall G=(V,E), (G \text{ is a tree } \wedge |V| \geq 2) \implies (\exists v \in V, d(v)=1)$

proof. Let $G=(V,E)$ be a tree. Assume that $|V| \geq 2$. We want to prove that there exists a vertex $v \in V$ which has exactly one neighbour

Let u be an arbitrary vertex in V. Let v be a vertex in G that is at the maximum possible distance from u.

Let P be the shortest path between v and u. We know that v has **at least** one neighbour: the vertex immediately before it on P.

v cannot be adjacent to any other vertex on P, as otherwise G would would have a cycle.

Also, v cannot be adjacent to any other vertex w **not** on P, as otherwise we could crate a longer path.

And so v has exactly one neighbour.

- > $\forall n \in \mathbb{Z}^+, \forall G = (V, E), (G \text{ is a tree} \wedge |V| = n) \implies |E| = n - 1.$

proof.

Case 1: Let $n = 1$. Let $G = (V, E)$ be an arbitrary graph, and assume that G is a tree with one vertex.

In this case, G cannot have any edges. Then $|E| = 0 = n - 1$.

Case 2: Let $k \in \mathbb{Z}^+$, and assume that $P(k)$ is true. We want to prove that $P(k+1)$ is also true.

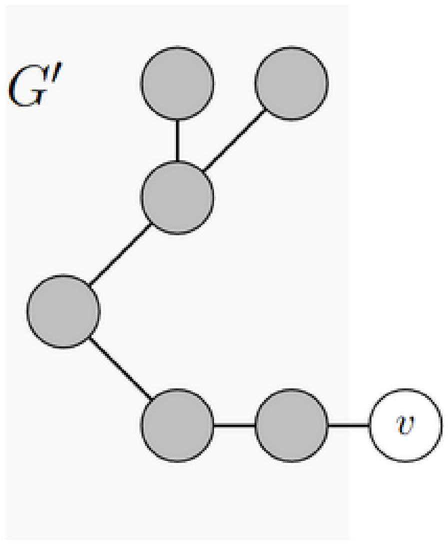
Let $G = (V, E)$ be a tree, and assume $|V| = k+!$. We want to prove that $|E| = k$.

By the previous three lemma, since $k + 1 \geq 2$, there exists a vertex $v \in V$ that has exactly one neighbour.

Let $G' = (V', E')$ be the graph obtained by removing v and the one edge on v from G. Then $|V'| = |V| - 1 = k$ and $|E'| = |E| - 1$.

We know that G' is also a tree. Then the induction hypothesis applies, and we can conclude that $|E'| = |V'| - 1 = k - 1$.

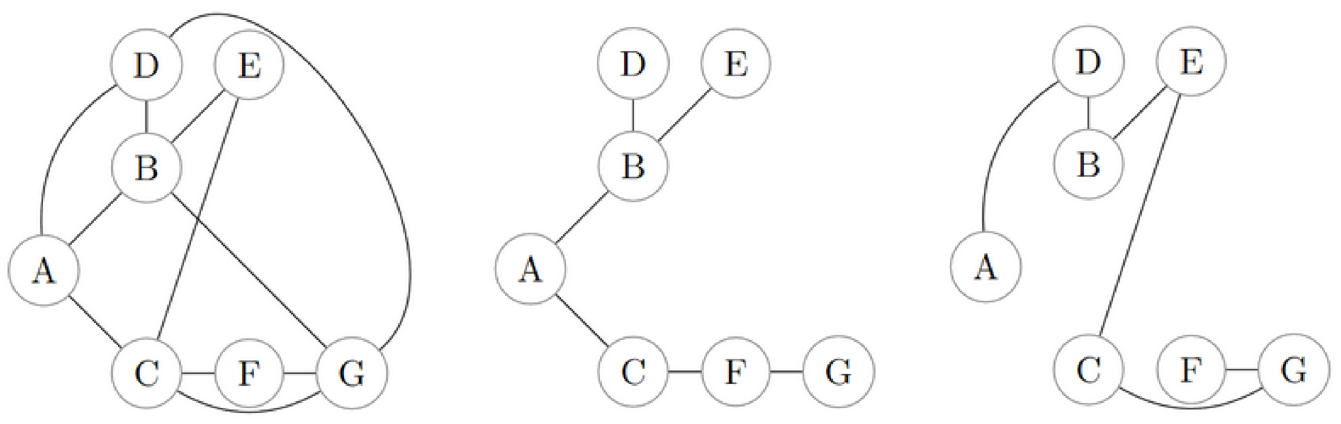
This means that $|E| = |E'| + 1 = k$ as required.



- **Theorem.** Let $G = (V, E)$ be a graph. If G is connected, then $|E| \geq |V| - 1$.

17.7 Computing Spanning Trees

- Let $G = (V, E)$ be a connected graph. Let $G' = (V', E')$ be another graph with the same vertex set as G, and where $E' \subseteq E$. We say that G' is a **spanning tree** of G when G' is a tree.



1. try to find a cycle → remove one of its edges → repeat until no more cycle **(inefficient)**
2. print connected vertices - drop the early return or are searching for a vertex that `self` isn't connected to.
   In this case, we expect that this method will recurse on **every vertex** that `self` is **connected to**.

```
class _Vertex:
    def print_all_connected(self, visited: set[_Vertex]) -> None:
        """Print all items that this vertex is connected to, WITHOUT
using any of the vertices in visited.

        Preconditions:
            - self not in visited
        """
        print(self.item)

        visited.add(self)
        for u in self.neighbours:
            # Only recurse on vertices that haven't been visited
            if u not in visited:
                u.print_all_connected(visited)
class Graph:
    def print_all_connected_indented(self, visited: set[_Vertex], d: int)
 -> None:
        """Print all items that this vertex is connected to, WITHOUT
using any of the vertices in visited.

        Print the items with indentation level d.

        Preconditions:
            - self not in visited
            - d >= 0
        """
        print('  ' * d + str(self.item))

        visited.add(self)
        for u in self.neighbours:
            # Only recurse on vertices that haven't been visited
            if u not in visited:
                u.print_all_connected_indented(visited, d + 1)
```

- Spanning tree algorithm
    1. each recursive call will now return a list of edges
    2. to "handle the root", we'll also need to add edges between `self` and each vertex where we make a recursive call.

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

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

        return edges_so_far

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

18. Sorting

18.1 Sorted Lists and Binary Search

- **binary search** - sorted list searched from the middle
    - need to keep track of the current search range
    - use `b` and `e` to represent the endpoints of this range
        - `lst[0:b]` less than the item being search
        - `lst[b:e]` current search range
        - `lst[e:len(lst)]` greater than the item being search
    - calculate the midpoint `m` of the current range
        - compare `lst[m]` against `item`

  - `item<lst[m]` → update `e`
  - `item > lst[m]` → update `b`

```python
def binary_search(lst: list, item: Any) -> bool:
    """Return whether item is in lst using the binary search algorithm.

    Preconditions:
        - lst is sorted in non-decreasing order
    """
    b = 0
    e = len(lst)

    while b < e:
        # Loop invariants
        assert all(lst[i] < item for i in range(0, b))
        assert all(lst[i] > item for i in range(e, len(lst)))

        m = (b + e) // 2
        if item == lst[m]:
            return True
        elif item < lst[m]:
            e = m
        else:  # item > lst[m]
            b = m + 1

    # If the loop ends without finding the item, the item is not in the list.
    return False
```

Running-time analysis

- focus on the quantity `e-b`
- `e-b` initially equals n the length of the input list
- the loop stops when `e-b <= 0`
- at each iteration, `e-b` decreases by at least a factor of 2
- at most $1 + \log_2 n$ iterations, with each iteration taking constant time
- **worst-case** $\mathcal{O}(\log n)$

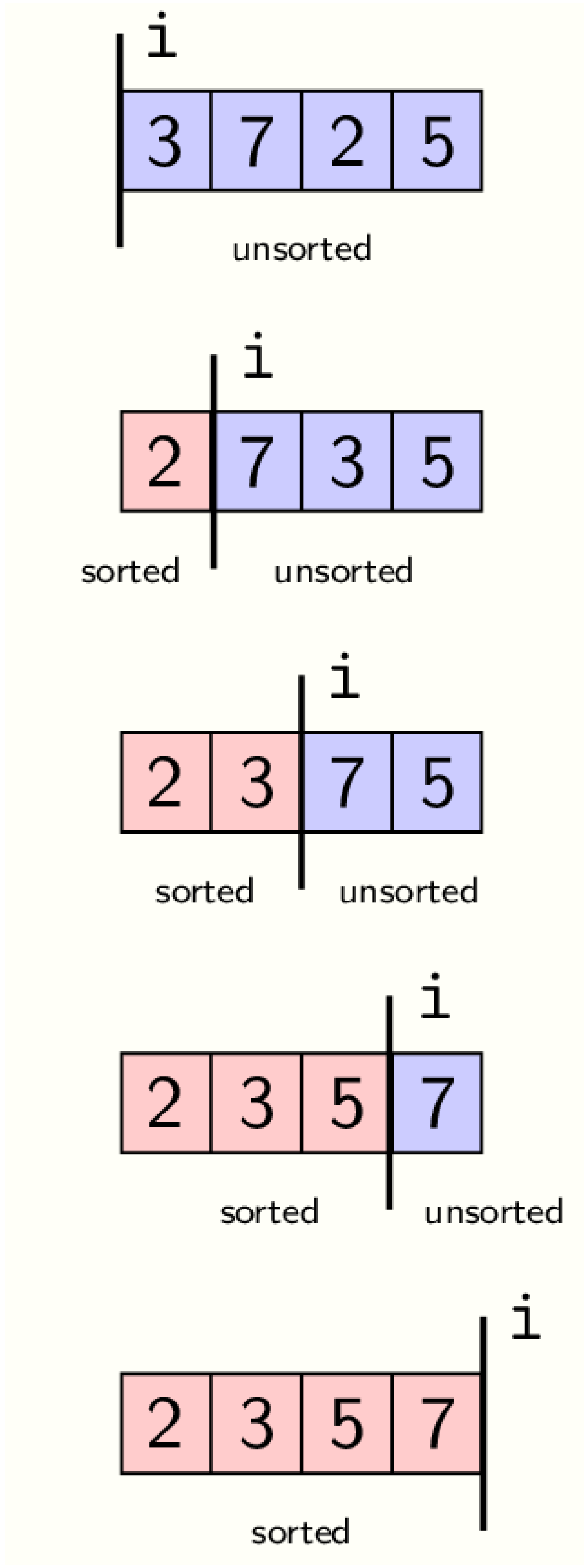
18.2 Selection Sort

- core idea
  - repeatedly extract the **smallest** element from the collection
  - building up a sorted list from these elements
  - (in-place) swap the items
- example `[3,7,2,5]`

items to be sorted	smallest element	sorted list
`[3,7,2,5]`	`2`	`[2]`
`[3,7,5]`	`3`	`[2,3]`
`[7,5]`	`5`	`[2,3,5]`
`[7]`	`7`	`[2,3,5,7]`
`[]`		`[2,3,5,7]`

```python
def selection_sort_simple(lst: list) -> list:
    """Return a sorted version of lst."""
    sorted_so_far = []

    while lst != []:
        smallest = min(lst)
        lst.remove(smallest)
        sorted_so_far.append(smallest)

    return sorted_so_far
```

- problem - mutate the input `lst`
- **in- place** - sorts a list by **mutating** its input list and without using any additional list objects
  - may use new memory to store, but this amount is **constant** with respect to the size of the input list ($\Theta(1)$)



- **swap** the smallest unsorted item with the item at index `i`

```
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
first).

    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

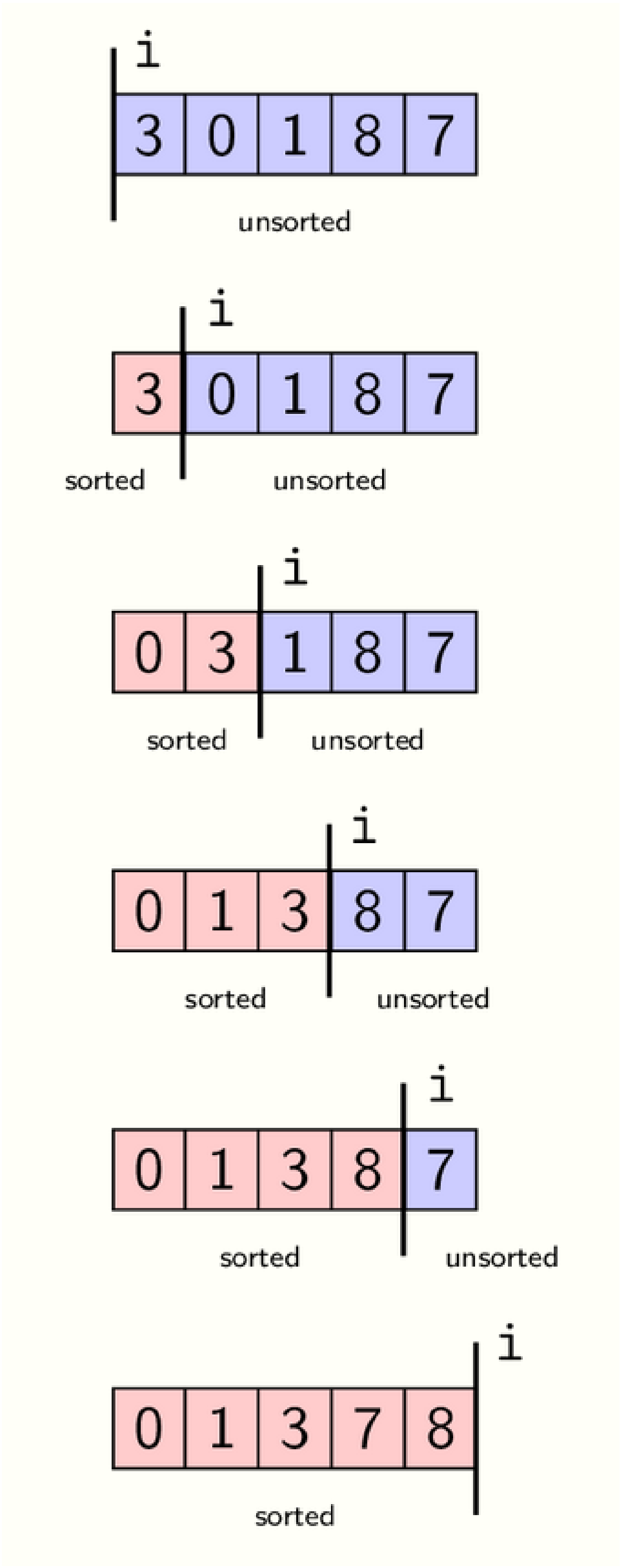
- helper function `_min_index`
  - outside of the loop take constant time. treat them as just a single step
  - the loop iterates $n-i-1$ times, the body takes constant time (1 step), the running time of the loop is $n-i-1$ steps
  - total : $(n-i-1)+1$, which is $\Theta(n-i)$
- `selection_sort`
  - call `_min_index`, takes $n-i$ steps (translate running time into an exact number of steps)
  - assignment takes constant time (1 step)
  - for one iteration - $n-i+1$

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

→ The running time of `selection_sort` is $\Theta(n^2)$

18.3 Insertion Sort

- core idea
  - takes the next item in the list `lst[i]`
  - **inserts** it into the sorted part by moving it into the correct location
- example `[3,0,1,8,7]`



- helper function `_insert`

```
def _insert(lst: list, i: int) -> None:
    """Move lst[i] so that lst[:i + 1] is sorted.
```

```python
    Preconditions:
        - 0 <= i < len(lst)
        - is_sorted(lst[:i])

    >>> lst = [7, 3, 5, 2]
    >>> _insert(lst, 1)
    >>> lst  # lst[:2] is sorted
    [3, 7, 5, 2]
    """

# 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
```

- insertion_sort

```python
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)
```

running-time analysis

- helper function `_insert` (early return version)
  - **Upper bound**
    - Let $n \in \mathbb{N}$ and `lst` be an arbitrary list of length n. Let $i \in \mathbb{N}$ and assume $i < n$
    - The loop runs **at most** i times and each iteration takes constant time (1 step)
    - total **at most** i steps → $\mathcal{O}(i)$
  - **Lower bound**
    - consider an input list `lst` where `lst[i]` is less than all items in `lst[:i]`
    - the expression `lst[j-1]<= lst[j]` will always be `False`
    - so the loop will only stop when `j==0`, which takes i iterations
    - total i steps → $\Theta(i)$
- `insertion_sort`
  - **Upper bound**
    - Let $n \in \mathbb{N}$ and let `lst` be an arbitrary list of length n. The loop iterates n times.
    - the call to `_insert(lst,i)` counts as **at most** i steps.
    - for one iteration - i
    - total $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$, which is $\mathcal{O}(n^2)$
  - **Lower bound**
    - let $n \in \mathbb{N}$ and let `lst` be the list `[n-1,n-2,...,1,0]`.
    - This input is an extension of the input family for `_insert`.

    - As we describe above, this causes each call to `_inset` to take i steps
    - total → $\Theta(n^2)$
- both selection sort and insertion sort are **iterative** sort - implemented by loop
- running insertion sort on list that's **already sorted** → running time is $\Theta(n)$

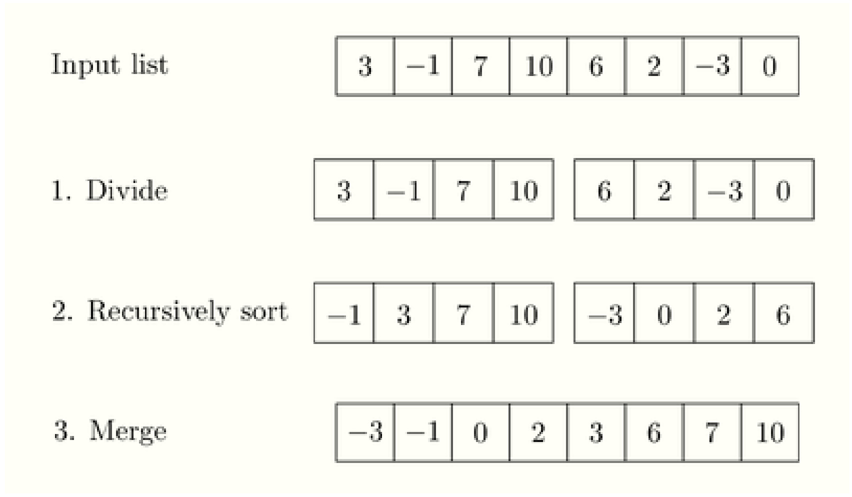
	Insertion sort	Selection sort
harder part	insert	pick item
easier part	pick item	insert
running time	$<= n^2$	always n^2

18.4 Introduction to Divide-and-Conquer Algorithms

- **divide-and-conquer**
    - given the problem input, **split** it up into two or more smaller subparts with the **same structure**
    - recursively run the algorithm on each subpart separately
    - **combine** the results of each recursive call into a single result, solving the original problem

18.5 Merge Sort

- core idea
    - divide the input into the **left** half and **right** half
    - recursive sort each half
    - merge each sorted half together

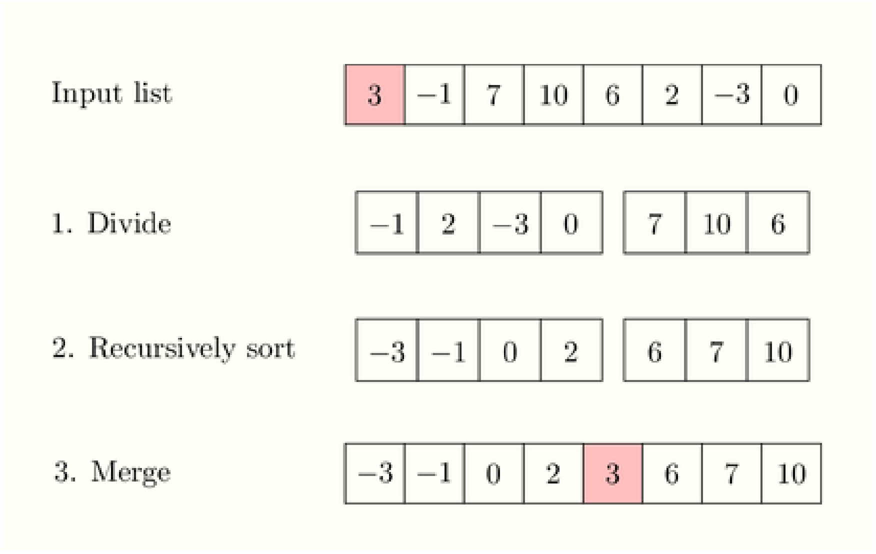


unmerged items in `lst1`	unmerged items in `lst2`	comparison	sorted list
`[-1, 3, 7, 10]`	`[-3, 0 ,2, 6]`	`-1` vs. `-3`	`[-3]`
`[-1, 3, 7, 10]`	`[0, 2, 6]`	`-1` vs. `0`	`[-3, -1]`
`[3, 7, 10]`	`[0, 2, 6]`	`3` vs. `0`	`[-3, -1, 0]`
`[3, 7, 10]`	`[2, 6]`	`3` vs. `2`	`[-3, -1, 0 ,2]`
`[3, 7, 10]`	`[6]`	`3` vs. `6`	`[-3, -1, 0, 2, 3]`
`[7, 10]`	`[6]`	`7` vs. `6`	`[-3, -1, 0, 2, 3, 6]`
`[7, 10]`	`[]`	N/A	`[-3, -1, 0, 2, 3, 6]`

```python
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:]


def mergesort(lst: list) -> list:
    """"Return a new sorted list with the same elements as lst.

    This is a *non-mutating* version of mergesort; it does not mutate the
    input 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)
```

18.6 Quick Sort

- core idea
  - pick one element - **pivot** (below always choose the **first**)
  - partitioning step - into two parts: **<=** to the pivot and **>** the pivot
  - sort each part recursively
  - concatenate the two sorted parts, putting the pivot in between them



```
def quicksort(lst: list) -> list:
    """Return a sorted list with the same elements as lst.

    This is a *non-mutating* version of quicksort; it does not mutate the
    input list.
    """
    if len(lst) < 2:
        return lst.copy()
    else:
        # Divide the list into two parts by picking a pivot and then
# partitioning the list.
        # In this implementation, we're choosing the first element as the
# pivot, but we could have made lots of other choices here
        # (e.g., last, random).
        pivot = lst[0]
        smaller, bigger = _partition(lst[1:], pivot)
```

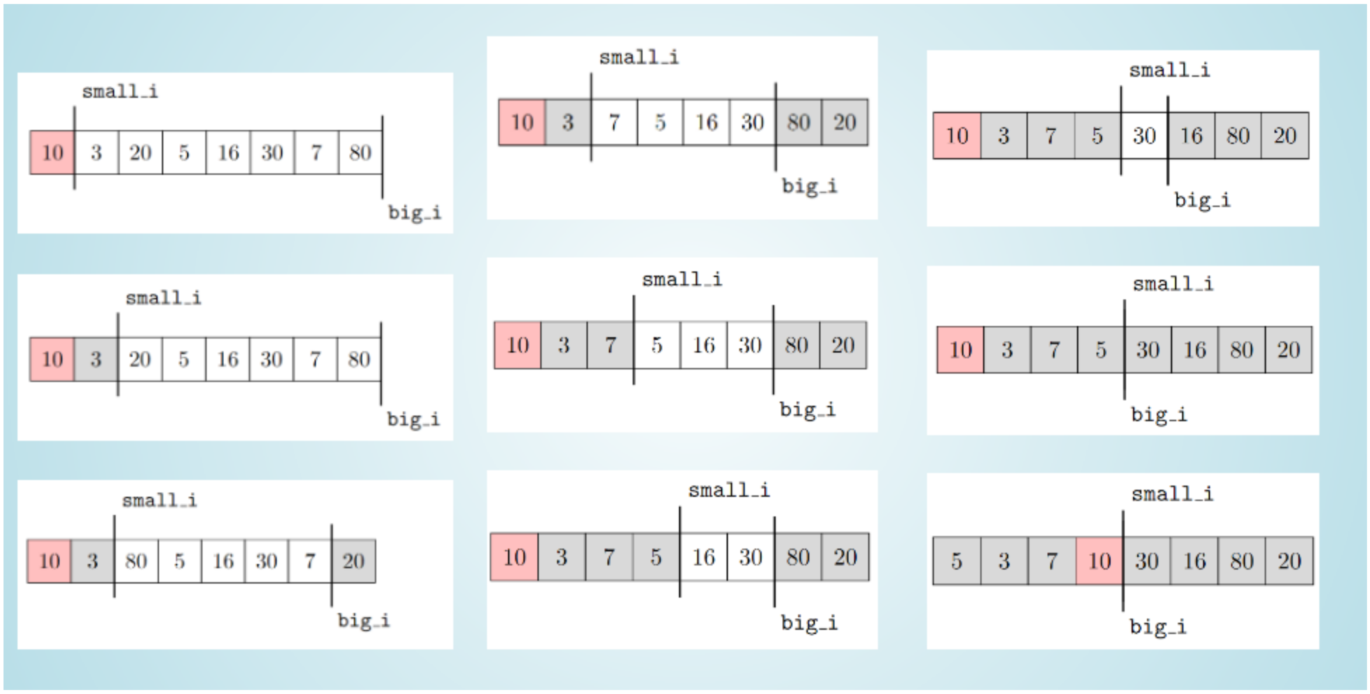
```python
        # Sort each part recursively
        smaller_sorted = quicksort(smaller)
        bigger_sorted = quicksort(bigger)

        # Combine the two sorted parts. No need for a helper here!
        return smaller_sorted + [pivot] + bigger_sorted


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)
```

- in place

```python
def in_place_quicksort(lst: list) -> None:
    """Sort the given list using the quicksort algorithm.
    """
    _in_place_quicksort(lst, 0, len(lst))


def _in_place_quicksort(lst: list, b: int, e: int) -> None:
    """Sort the sublist lst[b:e] using the quicksort algorithm.
    Preconditions:
    - 0 <= b <= e <= len(lst)
    """
    if e - b < 2:
    # Do nothing; lst[b:e] is already sorted
```

```
        pass
    else:
    # Partition lst[b:e]

        pivot_index = _in_place_partition_indexed(lst, b, e)

        # Recursively sort each partition
        _in_place_quicksort(lst, b, pivot_index) # smaller partition
        _in_place_quicksort(lst, pivot_index + 1, e) # bigger partition


def _in_place_partition_indexed(lst: list, b: int, e: int) -> int:
    """Mutate lst[b:e] so that it is partitioned with pivot lst[b].
    Return the new index of the pivot.

    Notes:
    - Only elements in lst[b:e] are rearranged.
    - _in_place_partition_indexed(lst, 0, len(lst)) is equivalent to
    _in_place_partition(lst), with an additional return value

    Preconditions:
    - 0 <= b < e <= len(lst)
    >>> my_lst = [10, 13, 20, 5, 16, 30, 7, 100]
    # pivot is 13
    >>> my_pivot_index = _in_place_partition_indexed(my_lst, 1, 6)
    >>> my_pivot_index # pivot is now at index 2
    2
    >>> my_lst[my_pivot_index]
    13
    >>> set(my_lst[1:my_pivot_index]) == {5}
    True
    >>> set(my_lst[my_pivot_index + 1:6]) == {16, 20, 30}
    True
    >>> my_lst[:1]
    [10]
    >>> my_lst[6:]
    [7, 100]
    """

    pivot = lst[b]
    small_i = b + 1
    big_i = e
```

```
while small_i < big_i: # while not (small_i == big_i):
# Loop invariants (homework: update loop invariants to use b and e)
# assert all(lst[j] <= pivot for j in range(1, small_i))
# assert all(lst[j] > pivot for j in range(big_i, len(lst)))

    if lst[small_i] <= pivot:
        # Increase the "smaller" partition
        small_i += 1

    else: # lst[small_i] > pivot
        # Swap lst[small_i] to back and increase the "bigger" partition
        lst[small_i], lst[big_i - 1] = lst[big_i - 1], lst[small_i]
        big_i -= 1
# Move the pivot to between the "smaller" and "bigger" parts
lst[b], lst[small_i - 1] = lst[small_i - 1], lst[b]
# Return the new index of the pivot
return small_i - 1
```



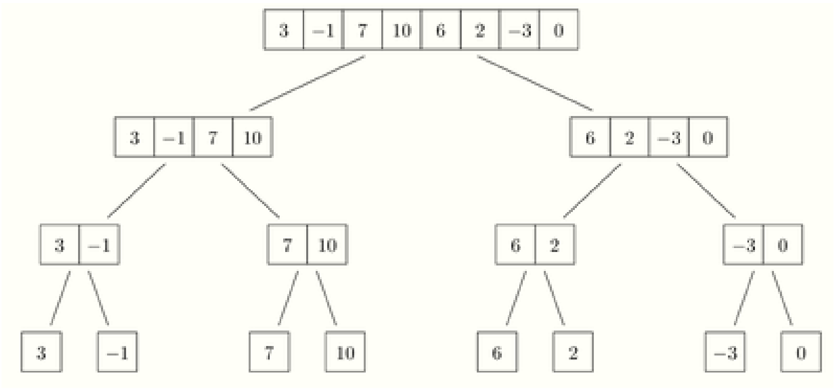
18.7 Running-Time Analysis for Merge Sort and Quick Sort

`_merge`

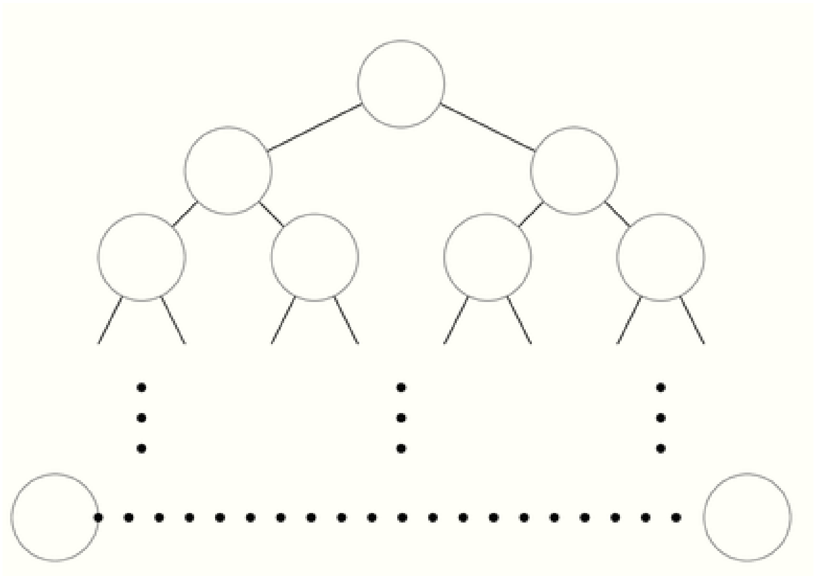
- **Upper Bound**
  - Let $n_1, n_2 \in \mathbb{N}$ and `lst1` and `lst2` be an arbitrary list of length n_1 and n_2.
  - Let $i_1, i_2 \in N$ and assume $i_1 < n_1$ and $i_2 < n_2$.
  - The loop runs **at most** $n_1 + n_2 - 1$ times and each iteration takes constant time (1 step)
  - When the loop stops, the length of `sorted_so_far` is $i_1 + i_2$.
  - Assume `i1 == len(lst1)` is true, then the list slicing runs $n_2 - i_2$ steps
  - the concatenations runs $i_1 + i_2 + n_2 - i_2 = n_1 + n_2$ steps
  - Total: $n_1 + n_2 - 1 + n_1 + n_2 = 2(n_1 + n_2) - 1$ → $\mathcal{O}(n_1 + n_2)$
- **Lower Bound**
  - Let $n \in \mathbb{N}$ be a power of 2 and let the list `lst1 = [1, ..., n-3, n-1]` and `lst2[2, ..., n-2, n]`.
  - Therefore, the length of `lst1` and `lst2`, n_1 and n_2, are $\frac{n}{2}$.
  - So the loop will stop when $i_1 == n_1$, at that time, $i_2 == n_2 - 1$, thus takes $n_1 + n_2 - 1$ ierations
  - The slicing `lst2[i_2:]` takes 1 steps
  - The concatenations takes $n_1 + n_2 - 1 + 1 = n_1 + n_2$ steps
  - Total: $n_1 + n_2 - 1 + n_1 + n_2 = 2(n_1 + n_2) - 1$ → $\Theta(n_1 + n_2)$

Merge Sort

- recursive call
  - a list of length n, where $n > 1$, assume that n is a power of 2
  - divide the list and obtain two lists of length $\frac{n}{2}$, and recurse on each one...
  - each call to `mergesort` taking as input a list of length $\frac{n}{2^k}$ and recursing on two lists of length $\frac{n}{2^{k+1}}$ → base case reached when list length is 1

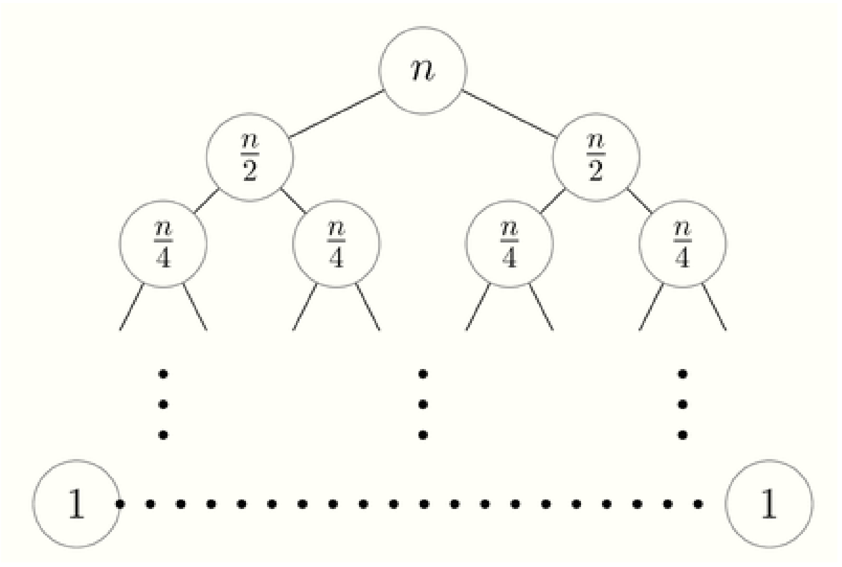


- recursive call diagram is a binary tree
- start with height n, where n is a power of 2, there are exactly $\log_2(n) + 1$ levels in this tree



- non-recursive
  - a list `lst` of length n, $n \geq 2$, assume n is a power of 2
  - if condition check (`len(lst) <2`) and calculation of `mid` take constant time
  - slicing operations `lst[:mid]` and `lst[mid:]` each take time proportional to the length of the slice, which is $\frac{n}{2}$

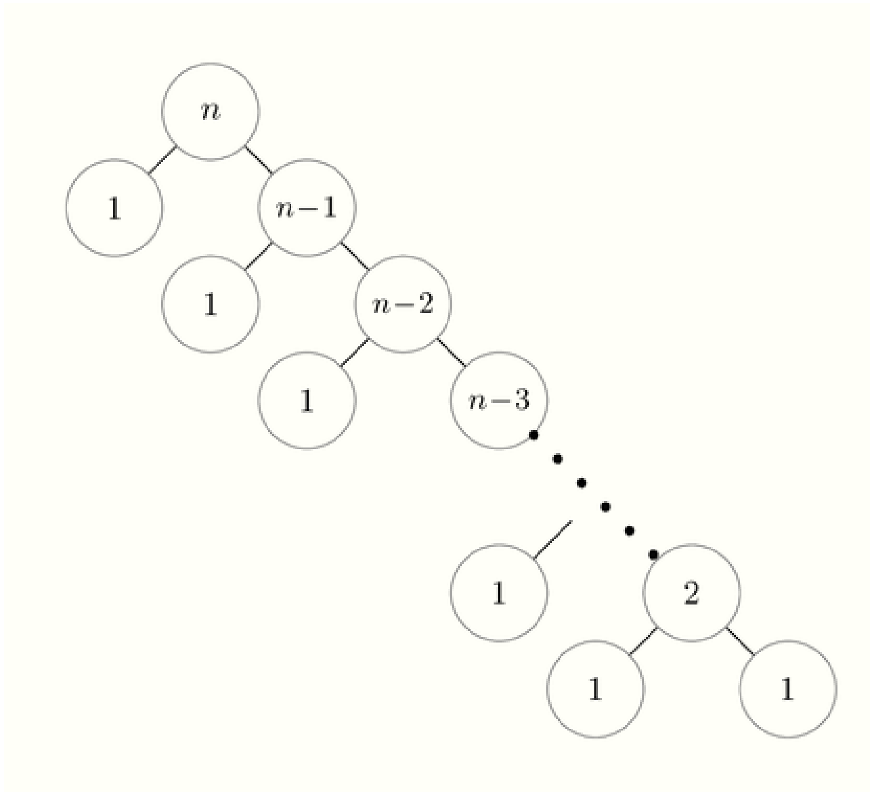
    NOTE: slicing occurs in the same line as a recursive call, the slicing itself is considered non-recursive, since it occurs **before** making the recursive calls
  - `_merge` takes $\frac{n}{2} + \frac{n}{2} = n$ steps
  - total: $1 + \frac{n}{2} + \frac{n}{2} + n = 2n + 1$, when $n \geq 2$
  - NOTE: when $n < 2$, the base case executes, which is cnstant time.
- use n instead of $2n + 1$ as the running time



- each **level** in the tree has nodes with the same running time
- at depth d in the tree, there are 2^d nodes, and each nodes contains the number $\frac{n}{2^d}$.
- add up the nodes at each depth, we get $2^d \cdot \frac{n}{2^d} = n$
- each **level** in the tree has the same total running time
- there are $\log_2(n) + 1$ levels → total $n \cdot (\log_2(n) + 1)$ → $\Theta(n \log n)$

Quick Sort

- non-recursive
  - let `lst` be a list with length of n, assume $n \geq 2$.
  - the slicing (`lst[1:]`) takes $n - 1$ steps
  - the list concatenation takes n steps
  - others takes constant time (1 step)
  - total: $1 + n - 1 + n = 2n$ → $\Theta(n)$.
- recursive call
  - makes two recursive calls, its recursion tree is also binary
  - the size of the partition depends on the choice of pivot
  - median - $\frac{n}{2}$, smallest - 1 and $n - 1$



- in this case, the height of the tree is n. The size of the `bigger` partition just decreases by 1 each call.
- There are $n-1$ recursive calls on empty partitions (`smaller`)
- total: $(\sum_{i=1}^{n} i) + (n-1) = \frac{n(n+1)}{2} + n - 1$ → $\Theta(n^2)$ (worst-case) $\Theta(n \log n)$ (best-case)

mergesort vs. quicksort

- the same simlification al has the effect of **flattening** reported running times → all $\Theta(n \log n)$ look the same
- the **in-place** quicksort can be significantly faster than merger sort
  - for most inputs, quick sort has "samller contants" than mergesort
  - performas fewer primitive machine operations
- the performance of quicksort on a **random** list of length n ⇔ choose a **random** elements to be the pivot
  - leads **average-case running time analysis**
  - quick sort has an average-case running time of $\Theta(n \log n)$
    - with smaller constant factors than merge sort's $\Theta(n \log n)$
    - indicates that the actual "bad" inputs for quicksort are quite rare

- linear-time alogithm for computing the median of a list of numbers → median of medians
- can always choose the median of the list as the pivot in the quicksort's partitioning step, making the worst-case running time $\Theta(n \log n)$
- better worst-case running time → calcute the median → slower running time for random input (rarely use)
- built-in sort - **Tim Sort** using the same basic idea of merging sorted sublists, smarter and more efficient way than our impliementation

Sorting algorithm	RT(worst-case)	RT(average-case)	RT(best-case)
selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
merge sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
quick sort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$
`list.sort`	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$

18.9 Generalized Sorting

- sorting by length
  - call `insertion_sort` - sort alphabetically (default behaviour)

```python
def sort_by_len(lst: list[str]) -> None:
    """Sort the given list of strings by their length.

    >>> lst = ['david', 'is', 'cool', 'indeed']
    >>> sort_by_len(lst)
    >>> lst
    ['is', 'cool', 'david', 'indeed']
    """
    insertion_sort_by_len(lst)


# Insertion sort implementation, for reference
def insertion_sort_by_len(lst: list) -> None:
    for i in range(0, len(lst)):
        _insert_by_len(lst, i)
```

```python
def _insert_by_len(lst: list, i: int) -> None:
    for j in range(i, 0, -1):
        if len(lst[j - 1]) <= len(lst[j]):  # This line has changed!
            return
        else:
            lst[j - 1], lst[j] = lst[j], lst[j - 1]
```

- `key` parameter
  - represents a **function** that specifies how to compute the values that will be compared

```python
from typing import Callable


def insertion_sort_by_key(lst: list, key: Callable) -> None:
    """Sort the given list using the insertion sort algorithm.

    The elements are sorted by their corresponding return values 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: Callable) -> None:
    """Move lst[i] so that lst[:i + 1] is sorted.
    """
    for j in range(i, 0, -1):
        if key(lst[j - 1]) <= key(lst[j]):  # This line has changed again!
            return
        else:
            lst[j - 1], lst[j] = lst[j], lst[j - 1]

def sort_by_len(lst: list[str]) -> None:
    """Sort the given list of strings by their length.

    >>> lst = ['david', 'is', 'cool', 'indeed']
    >>> sort_by_len(lst)
    >>> lst
    ['is', 'cool', 'david', 'indeed']
    """
    insertion_sort_by_key(lst, len)
```

- can define our functions and pass into `key`

  ```
  def count_a(s: str) -> int:
      """Return the number of 'a' characters in s.

      >>> count_a('david')
      1
      """
      return s.count(a)


  def sort_by_a_count(lst: list[str]) -> None:
      """Sort the given list of strings by their number of 'a' characters.

      >>> lst = ['david', 'is', 'amazing']
      >>> sort_by_a_count(lst)
      >>> lst
      ['is', 'david', 'amazing']
      """
      insertion_sort_by_key(lst, count_a)
  ```

- built-in `sorted` has **optional** `key` parameter

- **anonymous function** - defines a new function without giving it a name

  ```
  lambda <param> ... : <body>
  ```

  - can only have an **expression** as their body - cannot contain statements like assignment statements or loops

```
lambda x: x + 1
lambda lst1, lst2: len(lst1) * len(lst2)

>>> strings = ['david', 'is', 'amazing']
>>> sorted(strings, lambda s: s.count('a'))
['is', 'david', 'amazing']
```

- **memoization** - saving the return values of function calls so that they can be looped up later
  - **memoized** version of insertion sort uses a dictionary mapping a list element `x` to `key(x)`, so that `key` is only called once per `x` value
  - E.g., if `lst=['David', 'is', 'cool']` and `key = len`. the memoization dictionary looks like

```
{
    'David': 5,
    'is': 2,
    'cool': 4
}

def insertion_sort_memoized(lst: list, key: Optional[Callable] = None)
-> None:
    """..."""
    # Initialize a dictionary to store the results of calling `key`
    key_values = {}
    for i in range(0, len(lst)):
        _insert_memoized(lst, i, key, key_values)

def _insert_memoized(lst: list, i: int, key: Optional[Callable] = None,
        key_values: Optional[dict] = None) -> None:
    """Same as _insert_by_key, except that:
    When key(x) should be computed, first look up x in key_values.

    - If x is in key_values, return the corresponding values
    - Otherwise, compute key(x) and store the result in key_values
    """
    for j in range(i, 0, -1): # This goes from i down to 1
        if key is None:
            if lst[j - 1] <= lst[j]:
            # The element has been inserted into the correct position!
                return
            else:
                # Swap lst[j - 1] and lst[j]
                lst[j - 1], lst[j] = lst[j], lst[j - 1]
        else: # key is a function that we should use to compare values
            # Get the key values (this is the part you need to change
            k1 = key(lst[j - 1])
            k2 = key(lst[j])
            # Then do the swapping.
            if k1 <= k2:
            # The element has been inserted into the correct position!
                return
            else:
                 # Swap lst[j - 1] and lst[j]
                lst[j - 1], lst[j] = lst[j], lst[j - 1]
```