CSC111

Table of Contents

11.1 Introduction to Linked Lists

Intro

- Goal: create a new Python class that behaves exactly the same as the built-in list class

Specifying Order Through Links

- We can store along with each element a reference to the *next* element in the list
- *Node* – an element plus a reference to the next element

```
@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
```

- An instance of _Node represents a *single element* of a list
- Given a bunch of _Node objects, we can follow each next attribute to recover the sequence of items that these nodes represent

The LinkedList Class

- Represents the list itself
- Has a private attribute _first that refers to the first node in the list

```
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
```

Building Links

- _node3 is very different from _node3.item
- The former is a _Node object containing a value (say 111), and the latter is the value 111

11.2 Traversing Linked Lists

Intro

- For a Python list, we can manually use an index variable i to keep track of where we are in the list

```
i = 0
while i < len(my_list):
    … do something with my_list[i] …
    i += 1
```

  - Contains 4 parts
    - Initialize the loop variable i (0 refers to the starting index of the list)
    - Check if we've reached the end of the lit in the loop condition
    - In the loop, do something with the current element my_list[i]
    - Increment the index loop variable
- We don't have the indexing operation
- Our loop variable should refer to the _Node object we're currently on in the loop

```
curr = my_linked_list._first  # Initialize curr to the start of the list
while curr is not None  # curr is None if we've reached the end of the loop
    … curr.item …  # Do something with the current 'element', curr.item
    curr = curr.next  # 'Increment' curr, assigning it to the next node
```

Looping With an Index

- We can implement an indexing method, the equivalent of my_list[3]

```
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 = curr_index + 1

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

- By updating curr and curr_index together, we get a *loop invariant*
  - curr refers to the node at index curr_index in the linked list
- The above implementation uses an *early return* inside the loop, stopping as soon as we've reached the node at the given index i

- Another approach modifies the while loop condition so that the loop stops when it either reaches the end of the list or the correct index

```
        def __getitem__(self, i: int) -> Any:
            """..."""
            curr = self._first
            curr_index = 0

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

            assert curr is None or curr_index == i
            if curr is None:
                # index is out of bounds
                raise IndexError
            else:
                # curr_index == i, so curr is the node at index i
                return curr.item
```

- No early return
- Two cases to consider after the loop ends

__getitem__ and List Indexing Expressions

- The __getitem__ special method is called automatically by the Python interpreter when we use *list indexing expressions*
  - linky[0] is equivalent to linky.__getitem__(0)

11.3 Mutating Linked Lists

LinkedList.append

- We need to find the current last node in the linked list, and then add a new node to the end of that
- We need to stop the loop when it reaches the last node
- We also need a case where curr starts as None

```
def append(self, item: Any) -> None:
    """Add the given item to the end of this linked list. ""
    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
```

A More General Initializer

- With our append method in place, we can modify our linked list initializer to take in an iterable collection of values, which we'll then append one at a time to the linked list

```
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)
```

    - This code is inefficient

11.4 Index-Based Mutation

Intro

- We want to implement a method analogous to link.insert

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

        Raise IndexError if i is greater than the length of self.

        If i *equals* the length of self, add the item to the end of the linked list,
        which is the same as LinkedList.app

        Preconditions:
            - i >= 0

        >>> lst = Linked([111, -5, 9000, 200])
        >>> lst.insert(2, 300)
        >>> lst.to_list()
        [111, -5, 300, 9000, 200]
        """
```

- For our doctest, we want to insert item 300 at index 2 in the list
    - We need to modify the current node at index 1 to link to our new node
    - We also need to modify the new node so it links to the old node at index 2

Implementing LinkedList.insert

- If we want the node to be inserted into position i, we need to access the node at position i – 1

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

        curr = self._first
        curr_index = 0

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

        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
```

- When the loop is over
  - If curr is None, the then list doesn't have a node at position i – 1, and so i is out of bounds
  - If not, then we've reached the desired index, and can insert the new node
- Corner case: i == 0
  - We need an extra condition since it does not make sense to iterate to the -1th node

Common Error

- The following order of link updates in the final else branch doesn't work

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

  - On the second line, curr.next has already been updated, and its old value lost
- *Parallel assignment* is recommended

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

11.5 Linked List Running-Time Analysis

Running-Time Analysis of LinkedList.insert

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

    if i == 0:
        self._first, new_node.next = new_node, self._first
    else:
        curr = self._first
        curr_index = 0

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

        if curr is None:
            raise IndexError
        else:
            curr.next, new_node.next = new_node, curr.next
```

- *Running-time analysis*. Let n be the length (i.e. number of items) of self
    - Case 1: Assume i == 0. In this case, the if branch executes, which takes constant time, so we'll count it as 1 step
    - Case 2: Assume i > 0. In this case,
        - The first 2 statements in the else branch (curr = self._first, curr_index = 0) takes 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 1 step
        - The while loop iterates until either it reaches the end of the list (curr is None) or until it reaches the correct index (curr_index == i – 1)
            - The first case happens after n iterations, since curr advances by 1 _Node each iteration
            - The second case happens after $i-1$ iterations, since curr_index starts at 0 and increases by 1 each iteration
        - So the number of iterations taken is $\min(n, i-1)$
        - Each iteration takes 1 step, for a total of $\min(n, i-1)$ steps
        - This gives us a 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 so we can say that the overall running time of LinkedList.insert is $\Theta(\min(n, i))$.

Comparing LinkedList and List Running Times

- We can assume that $0 \leq i < n$, then $\min(n, i) = i$

- The running time of LinkedLIst.insert is $\Theta(i)$
    - 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 running times of LinkedList.__getitem__ and LinkedList.pop follow the same analysis, since they also involve using a loop to reach the node at the correct index

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

- For insertion and deletion, linked lists have the exact opposite running time as array-based lists
- If we want constant time indexing and mainly will add/remove elements at the end of the list, use list
- If we plan to mainly add/remove elements at the front of the list, use LinkedList

12.1 Proof by Induction

A Proof by Induction

- **Ex.** Let $f : \mathbb{N} \to \mathbb{N}$ be defined as $f(n) = \sum_{i=0}^{n} i$. Prove that for all $n \in \mathbb{N}, f(n) = \frac{n(n+1)}{2}$.

The Principle of Induction

- The principle of induction applies to *universal* statements over natural numbers
    - i.e. $\forall n \in \mathbb{N}, P(n)$
- Two steps to proving a statement $\forall n \in \mathbb{N}, P(n)$ by induction:
    - The **base case** is a proof that the statement is True for the first natural number $n = 0$
        - i.e. a proof that $P(0)$ holds
    - The **inductive step** is a proof that for all $k \in \mathbb{N}$, if $P(k)$ is True, then $P(k+1)$ is also True
        - i.e. $\forall k \in \mathbb{N}, P(k) \Longrightarrow P(k+1)$
- Once the base case and inductive steps are proven, we can conclude $\forall n \in \mathbb{N}, P(n)$
- Structure
    - Given statement to prove: $\forall n \in \mathbb{N}, P(n)$
    - $Proof$. We prove this by induction on n.
        - **Base case:** Let $n = 0$
            - [Proof that $P(0)$ is True]
        - **Inductive step:** Let $k \in \mathbb{N}$, and *assume* that $P(k)$ is True

            - *Induction hypothesis* – the assumption that $P(k)$ is True
            - [Proof that $P(k+1)$ is True]
    - QED

Back to Example

- $Proof$. We prove this statement by induction on n.
- **Base case:** Let $n = 0$
    - In this case, $f(0) = \sum_{i=0}^{0} i = 0$, and $\frac{0(0+1)}{2} = 0$. So the two sides of the equation are equal.
- **Inductive step:** Let $k \in \mathbb{N}$ and assume that $f(k) = \frac{k(k+1)}{2}$. We want to prove that $f(k+1) = \frac{(k+1)(k+2)}{2}$
    - $Discussion$. We need to determine how to use the induction hypothesis. $f(k+1) = \sum_{i=0}^{k+1} i$ can be broken down by "taking out" the last term: $\left(\sum_{i=0}^{k} i\right) + (k+1)$
    - We will start with the left side of the equation

      $f(k+1) = \sum_{i=0}^{k+1} i$ (Definition of f)

      $= \left(\sum_{i=0}^{k} i\right) + (k+1)$ (Taking out last term)

      $= f(k) + (k+1)$ (Definition of f)

      $= \frac{k(k+1)}{2} + (k+1)$ (By the I.H.)

      $= \frac{(k+1)(k+2)}{2}$ QED

How Does Induction Work?

- Rather than directly proving a predicate for an arbitrary n, we *assume* that the predicate is True for an arbitrary k instead
- The inductive step allows us to form a *chain of implications* going from 0 to every other natural number
- From base case, $P(0)$ is True. From induction step, $P(0) \Longrightarrow P(1), P(1) \Longrightarrow P(2)$, …, thus covering all natural numbers

12.2 Recursively-Defined Functions

Intro

- In the last section, we broke down the summation into one of the same form, but of a slightly smaller size:
    - $\sum_{i=0}^{k+1} i = \left(\sum_{i=0}^{k} i\right) + (k+1)$

- For all $k \in \mathbb{N}, f(k+1) = f(k) + (k+1)$
    - This relationship gives us a different way of *defining* f
    - $f(n) = \begin{cases} 0, \text{ if } n = 0 \\ f(n-1) + n, \text{if } n > 0 \end{cases}$
    - This is a *recursive* definition
        - i.e. f is defined in terms of itself
        - Another name: *self-referential definition*
    - We were able to manipulate the equation $f(k+1) = f(k) + (k+1)$ to prove the inductive step

Recursively-Defined Functions in Python

- We can represent the above function f in Python

```
def f(n: int) -> int:
    “””Return the sum of the numbers between 0 and n, inclusive.

    Preconditions:
        - n >= 0

    >>> f(4)
    10
    “””
    if n == 0:
        return 0
    else:
        return f(n – 1) + n
```

    - Inefficient comparing to return n * (n + 1) // 2
- Let f be a Python function. f is a *recursively-defined function* (or *recursive function*) when it contains a call to itself in its body
- We use the term *recursive call* to describe the inner f(n – 1) call
- *Recursion* – the programming technique of defining recursive functions to perform computations and solve problems
- Structure
    - The if branch, consisting of the statement return 0, is the *base case* of the function
    - The else branch, consisting of the statement return f(n – 1) + n, is the *recursive step* of the function, since it contains a recursive call
- The *base case* does not require any additional “breaking down” of the problem

- Both the *inductive step* of a proof and the *recursive step* of a function require the problem to be broken down into an instance of a smaller size
    - Either by using the inductive hypothesis or by making a recursive call

How Does Recursion Work?

- What the Python interpreter does:
    - When we call f(0), the base case executes, and 0 is returned
    - When we call f(1), the recursive call f(1 – 1) == f(0) is made, which returns 0. Then 0 + 1 == 1 is returned
    - When we call f(2), the recursive call f(2 – 1) == f(1) is made, which returns 1. Then 1 + 2 == 3 is returned
    - f(2) calls f(1), which calls f(0)
- Reasoning with the *inductive approach*:
    - We assume that the recursive call f(n – 1) returns the correct result
        - Analogous to the induction hypothesis
    - **Ex.** Reason inductively about the call f(100)
        - When we call f(100), the recursive call f(100 – 1) == f(99) is made. Assuming this call is correct, it returns 4950
        - Then 4950 + 100 == 5050 is returned
- The inductive approach technique is also called *partial tracing*
    - “Partial”: we don’t trace into any recursive calls, but instead assume that they work correctly

The Euclidean Algorithm

- Calculates the gcd of two numbers
- Relies on the fact that for all $a, b \in \mathbb{Z}$ with $b \neq 0, \gcd(a, b) = \gcd(b, a \% b)$
- Implementation using a while loop

```
def euclidean_gcd(a: int, b: int) -> int:
        “””Return the gcd of a and b.

        Preconditions:
                - a >= 0 and b >= 0
        “””
        x = a
        y = b
        while y != 0:
                r = x % y
                x = y
        y = r
    return x
```

- We also know that $\gcd(a, 0) = a$ for all $a \in \mathbb{N}$
- The recursive definition of the <u>gcd</u> function over the natural numbers:
    - $\gcd(a, b) = \begin{cases} a, \text{ if } b = 0 \\ \gcd(b, a \,\%\, b), \text{ if } b > 0 \end{cases}$
    - The recursive part decreases from b to $a \,\%\, b$ each time
    - We are not just limited to going "from n to $n - 1$"
- A recursive definition is valid as long as it always uses "smaller" argument values to the function in the recursive call
- Translating the recursive gcd definition into Python code:

```
def euclidian_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)
```

12.3 Introduction to Nested Lists

<u>A Motivating Example</u>

- Consider the problem of computing the sum of a list of numbers. It is very easy.
- If we make the input structure a list of lists of numbers, we would need to use a nested loop to process individual items in the nested list
- If we add another layer, the function would have a "nested nested loop"
- This can go on forever

<u>Simplifying Using Helpers</u>

- Though this works as a simplification, it does not generalize elegantly
- If we wanted to deal with 10th order nested sums, we would need to implement 10 functions

<u>Non-Uniform Testing</u>

- No function of the above form can handle nested lists with a non-uniform level of nesting among its elements
    - i.e. [[1, 2], [[[3]]], 4,]
- These functions operate on a list of a specific structure, requiring that each list element itself have the same level of nesting of its elements
- We need a better solution

12.4 Nested Lists and Structural Recursion

<u>A Recursive Definition for Nested Lists</u>

- We define a *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
    - Each of the a_i is called a *sublist* of the outer list
- It is very convenient to include single integers as nested lists in our definition

<u>A Recursive Definition for Nested List Sum</u>

- $nested\_sum(x) = \begin{cases} x, \text{if } x \in \mathbb{Z} \\ \sum_{i=0}^{k-1} nested\_sum(a_i), \text{if } x = [a_0, a_1, \dots, a_{k-1}] \end{cases}$
- The sum of a nested list that's an integer is 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

```
def sum_nested(nested_list: Union[int, list]) -> int:
        “””Return the sum of the given nested list.”””
        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
```

    - The base case is when <u>isinstance(nested_list, int)</u>
    - We can also use a comprehension utilizing <u>sum</u> instead of a loop
- Because a nested list can have an arbitrary number of sublists, we use a loop/comprehension call to make a recursive call on each sublist and aggregate the results

Inductive Reasoning

- If we want to trace the call:
  >>> sum_nested_v1([1, [2, [3, 4], 5], [6, 7], 8])
  - The recursive step executes, which is the else-branch
- We can set up a *loop accumulation table* to keep track of what's going on

Iteration	sublist	sum_nested(sublist)	Accumulator sum_so_far
0	n/a	n/a	0
1	1	1	1
2	[2, [3, 4], 5]	14	15
3	[6, 7]	13	28
4	8	8	36

  - We may assume that each recursive call is correct, and use this assumption to fill the third column of the table directly
  - The accumulator column shows the value of sum_so_far at the end of that row's iteration
  - The final entry shows the value of sum_so_far after the loop is complete

Recursive Function Design Recipe for Nested Lists

- 1. Write a doctest example to illustrate the *base case* of the function
  - i.e. 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 1 sublist is a single int, and another sublist is a list that contains other lists
  - Our 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

```
def f(nested_list: Union[int, list]) -> …:
    if isinstance(nested_list, int):
        …
    else:
        accumulator = …

        for sublist in nested_list:
            rec_value = f(sublist)
            accumulator = … accumulator … rec_value …
```

```
                return accumulator
```

- 4. Implement the function's base case, using the first doctest example to test.
- 5. Implement the function's recursive step by doing the following:
    - Use the second doctest example to write down the relevant sublists and recursive function calls (i.e. second and third columns of the loop accumulation table above). Fill in the recursive call output based on the function specification, not any code written
    - Analyse the output of the recursive calls and determine how to combine them to return the correct value for the original call
        - Involves *aggregation* of the recursive call return values

12.5 Recursive Lists

A Recursive Definition of Lists

- The 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
        - In this case, we call x the *first* element of lst, and r the *rest* of lst
- This definition decomposes a list into its "first" and "rest" parts, and is recursive because the "rest" part is another list
- We define a new *recursive* Python class to represent this recursive definition

```
from __future__ import annotations
from typing import Any

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 the list, or None if this list is empty.
    #     - _rest: A list containing the items 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
```

  - This RecursiveList data type is recursive, because its _rest instance attribute refers to another instance of RecursiveList
  - An empty list has no "first" or "rest" values, so we set both these attributes to None to represent an empty list
- To create a RecursiveList representing [1, 2, 3, 4]:
  - RecursiveList(1, RecursiveList(2, RecursiveList(3, RecursiveList(4, RecursiveList(None, None)))))

From Recursive Definition to Recursive Functions

- We can use our recursive list definition to design and implement recursive functions that operate on those lists
- We want to calculate the sum of a list of integers
  - Let lst be an empty list. In this case, its sum is 0
  - Let lst be a non-empty list oof integers. In this case, we can decompose it into its first element, x, and the rest of its elements, r. The sum of lst is equal to x plus the sum of the rest oof the elements.
  - i.e. if lst = [1, 2, 3, 4], then $sum(lst) = 1 + sum([2, 3, 4])$
- Recursive definition for list sum:
  - $sum(lst) = \begin{cases} 0, \text{ if } lst \text{ is empty} \\ (\text{first of } lst) + sum(\text{rest of } lst), \text{ if } lst \text{ is non-empty} \end{cases}$
- In Python:

```
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:
            return self._first + self._rest.sum()
```

    - This function calculates the sum of an arbitrary number of elements without using a built-in aggregation function or a loop

Nodes

- The RecursiveList and _Node classes have essentially the same structure

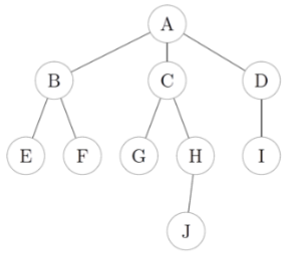
```
class RecursiveList:
    _first: Optional[Any]
    _rest: Optional[RecursiveList]

class _Node:
    item: Any
    next: Optional[_Node] = None
```

- _Node is technically a recursive class
- For a _Node, we think of it as representing a *single* list element
    - Its recursive attribute next is a "link" to another _Node, and we traverse these links in a loop to access each node one at a time
- For a RecursiveList, we think of it as representing an *entire* sequence of elements, not just one element
    - Its recursive attribute _rest is not a link, it is the rest of the list itself
- When computing on a RecursiveList, we don't try to access each item individually; instead, we make a recursive function call on the _rest attribute, and focus on how to use the result of that call in our computation

13.1 Introduction to Trees

Tree Definition and Terminology

- We use a *tree* data structure to represent hierarchical data
- Trees are *recursive*
- A tree is either:
    - Empty, or
    - Has a *root value* connected to any number of other trees, called the *subtrees* of the tree
- The *size* of a tree is the number of values in the tree
- A *leaf* is a value with no subtrees
- *Internal value* – opposite of a leaf, a value that has at least one subtree
- The *height* of a tree is the length of the *longest* path from its root to one of its leaves, counting the number of values on the path

- The *children* of a value are all values directly connected underneath that value
- The *descendants* of a value are its children, the children of its children, etc.
    - Can be defined recursively as "the descendants of a value are its children, and the descendants of its children
- The *parent* of a value is the value immediately above and connected to it
    - Each value in a tree has 1 parent, except the root, which has no parent
- The *ancestors* of a value are its parent, the parent of its parent, etc.
    - This can be defined recursively as "the ancestors of a value are its parent, and the ancestors of its parent"
- Sometimes, it is convenient to say that descendants/ancestors of a value include the value itself
- A value is *never* a child or parent of itself
- 

    - The root value of the above tree is A, it is connected to three subtrees
    - The size is of the above tree is 10
    - The leaves of the above tree are E, F, G, J, I
    - The internal values of the above tree are A, B, C, D, H
    - The height of the above tree is 4
    - The children of A are B, C, D

<u>A Tree Implementation</u>

```
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
```

- Our initializer here always creates either an empty tree (when root is None), or a tree with a value and the given subtrees
- A root that is not None and subtrees that is empty represents a tree with a single root value

13.2 Recursion on Trees

Tree Size

- Suppose we want to calculate the size of a tree. We can approach this problem by following the recursive definition of a tree, being either empty or a root connected to a list of subtrees
- Let T be a tree, and let $size$ be a function mapping of any tree to its size
    - If T is empty, the its size is 0: we can write $size(T) = 0$
    - If T is non-empty, then it consists of a root value and collection of subtrees $T_0, T_1, \dots, T_{k-1}$ for some $k \in \mathbb{N}$. In this case, the size of T is the sum of the sizes of its subtrees, plus 1 for the root:
        - $size(T) = 1 + \sum_{i=0}^{k-1} size(T_i)$
- We can combine the above observations to write a recursive mathematical definition of our $size$ function:

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

```
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)
```

- We need to recurse on each subtree
- Besides using the built-in aggregation function <u>sum</u>, we can also implement a custom aggregation operation using a loop

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

- Code template for recursing on trees:

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

- Often the <u>…</u> will use <u>self._root</u>

An Explicit Size-One Case

- We can handle the size-one case separately by adding an extra check

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

- Sometimes, this check is *necessary*
    - We want to do something different for a tree with a single item than for either an empty tree or one with at least one subtree
- Sometimes, this check is *redundant*
    - The action performed by this case is already handled by the recursive step
- In the case of __len__, the latter situation applies, because when there are no subtrees, the built-in sum function returns 0
- The three-case recursive Tree code template:

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

Example: Tree.__str__

- Trees have a non-linear ordering on the elements
- To implement Tree.__str__, we can start with the value of the root, then recursively add on the __str__ for each of the subtrees
    - The base case is when the tree is empty, and in this case the method returns an empty string

```
def _str_indented(self, depth: int) -> str:
    “””Return an indented string representation fo this tree.
```

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

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

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

  - We used *indentation* to differentiate between the different levels of a tree
  - We passed in an extra parameter for the *depth* of the current tree
    - Since we can't change the public interface of the __str__ method, we defined a helper method that has this extra parameter

Optional Parameters

- One way to customize the behaviour of functions is to make a parameter *optional* by giving it a *default value*
  - Can be done for any function

    i.e. def _str_indented(self, depth: int = 0) -> str:

            - depth becomes an optional parameter that can either be included or not included when this method is called
            - We can call t._str_indented()
                - No argument for depth given

- All optional parameters must appear *after* all of the required parameters in the function header
- Do *not* use mutable values like lists for optional parameters
    - If we do, the code will mysteriously stop working
    - Use optional parameters with immutable values like integers, strings, and None

Traversal Orders

- The __str__ implementation we gave visits the values in the tree in a fixed order:
    - 1. First it visits the root value
    - 2. Then it recursively visits each of its subtrees, in left-to-right order
        - By convention, we think of the _subtrees list as being ordered from left to right
- This visit order is known as the *(left-to-right) preorder* tree traversal
    - Root value is visited before any values in the subtrees
- Another common tree traversal is the *(left-to-right) postorder*
    - Visits the root value *after* it has visited every value in its subtrees
- Implementation of _str_indented in a postorder fashion:

```
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:
        s = ''
        for subtree in self._subtrees:
            s += subtree._str_indented(depth + 1)

        s += '  ' * depth + f'{self._root}\n'
        return s
```

13.3 Mutating Trees

<u>Value-Based Deletion</u>

```
class Tree:
    def remove(self, item: Any) -> bool:
        “””Delete *one* occurrence of the given item from this tree.

        Do nothing if the item is not in this tree.
        Return whether the given item was deleted.
        “””
        if self.is_empty():
            return False  # The item is not in the tree
        else:
            if self._root == item:
                self._delete_root()  # delete the root
                return False
            else:
                for subtree in self._subtrees:
                    subtree.delete_item(item)
```

- In the recursive step, we need to check whether the item is equal to the root
    - If so, then we only need to remove the root
    - If not, we need to recurse on the subtrees to look further for the item
- We need a helper method (Tree._delete_root)

- The inner <u>else</u> branch has two serious problems:
    - It doesn’t return anything, violating this method’s type contract
    - If one of the recursive calls successfully finds and deletes the item, no further subtrees should be modified (or even need to be recursed on)
- And so we need a boolean return value to tell us whether the item was deleted

```
            …
            else:
                for subtree in self._subtrees:
                    deleted = subtree.delete_item(item)
                    if deleted:
                        # One occurrence of the item was deleted, so 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
```

- We can move self._root == item check into an elif condition

```
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.delete_item(item)
            if deleted:
                return True
        return False
```

Deleting the Root

- We still need to implement Tree._delete_root
- If the tree has subtrees, then we can't set the _root attribute to None

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

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

- This implementation picks the rightmost subtree, and "promote" its root and subtrees by moving them up a level in the tree
- However, this implementation changed around some structure of the original tree just to delete a single element

The Problem of Empty Subtrees

- By the above implementation, after we delete an item, the result is an empty tree
    - The parent will contain an empty tree in its subtrees list
- Fixing the problem
    - If we detect that we deleted a leaf, we remove the now-empty subtree from its parent's subtree list

```
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.
            self._subtrees.remove(subtree)
            return True
        elif deleted:
            # The item was deleted, and the subtree is not empty.
            return True

    return False
```

    - In general it is *extremely dangerous* to remove an object from a list as we iterate through it
        - It interferes with the iterations of the loop that is underway
        - As soon as we remove the subtree, we stop the method by returning

Implicit Assumptions vs. Representation Invariants

- We want to make it explicit that each _subtrees list doesn't contain any empty trees

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

13.4 Running-Time Analysis for Tree Operations

Analysing Tree.__len__

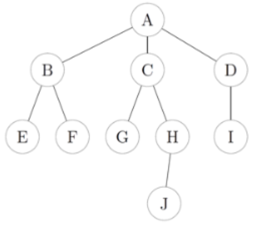
```
def __len__(self) -> int:
```

```
"""..."""
if self.is_empty():
    return 0
else:
    size_so_far = 1
    for subtree in self._subtrees:
        size_so_far += subtree.__len__()
    return size_so_far
```

- Let n be the size of self, i.e. the number of items in the tree
- We can ignore "small" values of n, so we assume $n > 0$, and the else branch executes
- We are making a call to subtree.__len__, but we are in the middle of trying to analyse the running time of subtree.__len__, which we don't know the running time

Analysing the Structure of Recursive Call

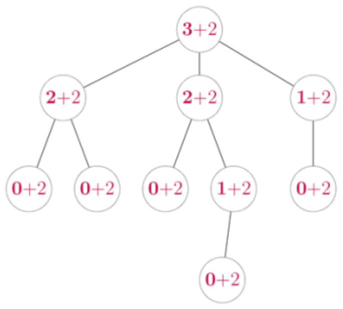
- Suppose we have the following tree



- We first identify all recursive calls that are made when we call __len__ on this tree
- Shorthand: "(A)" means "the tree rooted at A"
- When we make our initial call on the whole tree (rooted at A)
  - Initial call (A) makes three recursive calls on each of its subtrees (B), (C), and (D)
    - The recursive call on (B) makes two recursive calls on each of its subtrees, (E) and (F)
      - Each of (E) and (F) is a leaf, so no more recursive calls are made from them
    - The recursive call on (C) makes two recursive calls, on (G) and on (H)
      - The (G) is a leaf, so no more recursive calls happen for that tree
      - The recursive call on (H) makes one more recursive call on (J)
        - The (J) is a leaf, so no more recursive calls happen
    - The recursive call on (D) makes one recursive call on (I)
      - The (I) is a leaf, so no more recursive calls happen
- __len__'s recursive step always make a recursive call on every subtree, in total there is one __len__ call per item in the tree
  - The structure of the recursive calls exactly follows the structure of the tree

Analysing the Non-Recursive Part of Each Call

- We can't just count the number of recursive calls, since each call might perform other operations as well
- In addition to making recursive calls, there are some constant time operations, and a for loop that adds to an accumulator
- We can count the number of steps performed by a single recursive call, and add those up across all the different recursive calls that are made
- For the recursive step, we'll count the number of steps taken, assuming each recursive call takes constant time
- For Tree.__len__, the total number of steps taken is:
    - 1 step for the assignment statement
    - k steps for the loop, where k is the number of subtrees in the tree (which determines the number of loop iterations)
        - We are counting the loop body as just 1 step
    - 1 step for the return statement
- The above gives a total of $k + 2$ steps for the *non-recursive cost* of the else branch
- To find the total running time, we need to sum up across all recursive calls
- Challenge: k changes for each recursive call
    - i.e. (A) has $k = 3$, (B) has $k = 2$, (E) has $k = 0$
- We can write these costs for *every* recursive call in our example tree



    - The numbers in this tree represent the *total* number of steps taken by our initial call to Tree.__len__ across all the recursive calls that are made
    - The sum of all the subtrees (bold terms) is 9, which is one less than the total number of items
    - The 20 (sum of the non-bold terms) is the constant number of steps (2) multiplied by the number of recursive calls, which is equal to the number of items (10)

Generalize

- Let $n \in \mathbb{Z}^+$ and suppose we have a tree of size n. We know that there will be n recursive calls made.

    - The "constant time" parts will take $2n$ steps across all n recursive calls
    - The total number of steps taken by the for loop across all recursive calls is equal to the sum of all of the numbers of children of each node, which is $n - 1$
- This gives us a total running time of $2n + (n - 1) = 3n - 1$, which is $\Theta(n)$

Looking Back

- The above technique applies to any tree method of the form:

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

  as long as each of the … is filled in with constant-time operations
- If we have an early return, we will need to show the worst-case running time

13.5 Introduction to Binary Search Trees

The Multiset Abstract Data Type

- *Multiset*
    - Data: an unordered collection of values, allowing duplicates
    - Operations: get size, insert a value, check membership in the multiset

Implementing Multiset Using List and Tree

- Suppose we use a list to implement the Multiset ADT, where we simply append new items to the end of the list
    - This implementation would make *searching* for a particular item a $\Theta(n)$ operation, proportional to the size of the collection
- If we used a Tree, we'd get the same behaviour
    - If the item is not in the tree, every item in the tree must be checked
    - Switching from lists to trees would not do better
- In the case of Python lists, if we assume that the list is *sorted*, then we can use the *binary search* algorithm to improve the efficiency of searching to $\Theta(\log n)$
    - However, insertion and deletion into the front of the list takes $\Theta(n)$

Binary Search Trees: Definitions

- To implement the Multiset ADT efficiently, we will combine the branching structure of trees with the idea of binary search to develop a notion of a "sorted tree"
- *Binary Tree* – a tree in which every item has two (possibly empty) subtrees, which are labelled its *left* and *right* subtrees
- An item in a binary tree satisfied the *binary search tree property* when its value is:
    - Greater than or equal to all items in its left subtree, and
    - Less than or equal to all items in its right subtree
- A binary tree is a *binary search tree (BST)* when *every* item in the tree satisfies the binary search tree property
    - The "every" is important!
- We can think of binary search trees as a "sorted tree"
    - Even if the data isn't inserted in sorted order, the BST keeps track of it in a sorted fashion
    - Efficient in doing operations like searching for an item
    - Comparing to sorted Python lists, more efficient at insertion and deletion

Representing a Binary Search Tree in Python

```
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]
```

- Since the left/right ordering matters, we use explicit attributes to refer to the left and right subtrees

- An empty tree has a _root value of None, and its _left and _right attributes are None as well
- An empty tree is the *only* case where any of the attributes can be None
- The _left and _right attributes might refer to *empty* binary search trees, but this is different from them being None
- The initializer and is_empty methods are based on the corresponding methods for the Tree class

```
class BinarySearchTree:
    def __init__(self, root: Optional[Any]) -> None:
        """Initialize a new BST containing only the given root value.

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

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

  - We do not allow client code to pass in left and right subtrees as parameters to the initializer
    - Binary search trees have a strong restriction on where values can be located in the tree
- A separate method is used to insert new values into the tree that will ensure the BST property is always satisfied

Searching a Binary Search Tree

- For BSTs the initial comparison to the root tells us which subtree we need to check
  - Suppose we're searching for the number 111 in the BST. We check the root of the BST, which is 50. Since 111 is greater than 50, we know that if it does appear in the BST, it must appear in the *right* subtree, and that's the only subtree we need to recurse on

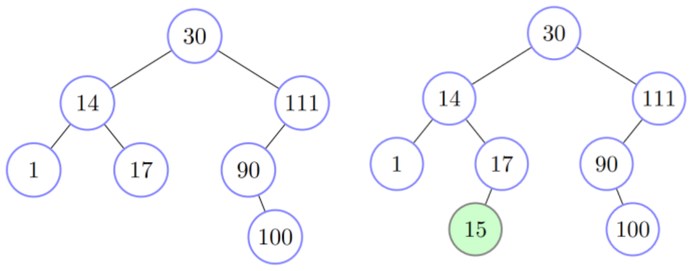
- In the recursive step for <u>BinarySearchTree.__contains__</u>, only one recursive call needs to be made

  ```
  class BinarySearchTree
          def __contains__(self, item: Any) → bool”
                  “””Return whether <item> is in this BST.”””
                  if self.is_empty():
                          return False
                  else:
                          if item == self._root:
                                  return True
                          elif item < self._root:
                                  return self._left.__contains__(item)
                          else:
                                  return self._right.__contains__(item)
  ```

  - We can combine the two levels of nested ifs

13.6 Mutating Binary Search Trees

<u>Insertion</u>

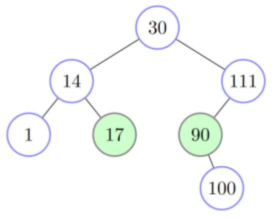
- The simplest approach is to put the new item at the “bottom” of the tree
- We can implement recursively:
  - If the BST is empty, make the new item the root of the tree
  - Otherwise, the item should be inserted into either the left subtree or the right subtree, while maintaining the binary search tree property
- If we want to insert 15 into the below tree, there is only one possible leaf position to put it: to the left of the 17
  - 

<u>Deletion</u>

- Given an item to delete, we take the same approach as __contains__ to search for the item
  - If we find it, it will be at the root of a subtree, where we delete it

```
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()
```

- After deleting the item, we set self._root = None only if the tree consists of *just* the root (with no subtrees)

    - If the BST has at least 1 other item, doing so would violate our representation invariant
- When at least one of the subtrees is empty, but the other one isn't , we can "promote" the other subtree up
- For the case where both subtrees are non-empty, we can fill the "hole" at the root by *replacing* the root item with another value from the tree (and then removing that other value from where it was)
    - For example, if we want to remove the root value 30 from the below tree, the two values we could replace it with are 17 and 90
        - 

    - Our implementation above extracts the maximum value from the left subtree, which requires a helper function

```
class BinarySearchTree:
    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()
```

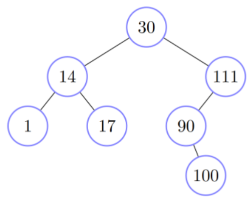
    - The base case here handles two scenarios:
        - self has a left (but no right) child
        - self has *no* children

13.7 The Running Time of Binary Search Tree Operations

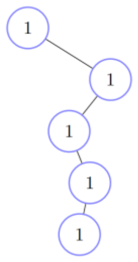
<u>Running-Time Analysis of BinarySearchTree.__contains__</u>

```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
```

- If we assume that each recursive call takes constant time, then the body runs in constant time
- The non-recursive cost is 1 step per recursive call
- BinarySearchTree.__contains__ only recurses on 1 subtree rather than all subtrees
- Suppose we search for the item 99 in the following binary search tree:



  - 5 calls to BinarySEarchTree.__contains__ are made in total
    - The initial call recurses on its right subtree, since 99 > 30
    - The second call recurses on its left subtree, since 99 < 111
    - The third call recurses on its left subtree
    - The fourth call recurses on its left subtree, since 99 < 100
    - The fifth call is on an *empty* tree, and so returns False without making any more recursive calls

  - 

- The number of recursive calls differs depending on the values stored in the tree and the values being searched for
- We will focus on the worst-case running time
- The total number of recursive calls is *at most* the height of the BST plus 1
  - The longest possible "search path" for an item is equal to the height of the BST, plus 1 for recursing into an empty subtree

        - $h+1$ steps, where h is the height of the BST
- The worst-case running time for BinarySearchTree.__contains__ is $\Theta(h)$
    - The same analysis holds for <u>insert</u> and <u>remove</u> as well

<u>Binary Search Tree Height vs. Size</u>
- In general, the height of a BST is much smaller than its size
- Consider a BST with n items, its height can be as large as n (in this case, the BST looks like a list), and can be as small as $\log_2 n$
    - A tree of height h can have size at most $2^h - 1$ items
    - If we want to store n items in a BST, we need $h \geq \log_2(n+1)$ to store all of them
- If we can guarantee that binary search trees always have height roughly $\log_2 n$, then all three BST operations (search, insert, delete) have a worst-case running time of $\Theta(h) = \Theta(\log n)$
- Since sorted lists have $\Theta(n)$ time insertion and deletion at the front of the list, binary search trees are a more efficient implementation of the Multiset ADT

<u>Looking Ahead</u>
- The above relies on the assumption of the height of a binary search tree always being roughly $\log_2 n$
    - The insertion and deletion algorithms we have studied do *not* guarantee this property holds when we mutate a binary search tree
        - Ex. when we insert items into a binary search tree in sorted order

14.1 Introduction to Abstract Syntax Trees

<u>Intro</u>
- The first step that the Python interpreter takes when given a Python file to run is to *parse* file's contents and create a new representation of the program code called an *Abstract Syntax Tree (AST)*

<u>The Expr Class</u>
- An *expression* is a piece of code which is meant to be evaluated, returning the value of that expression
- It is impossible to use one single class to represent all types of expressions, instead, we use different classes to represent each kind of expression and use inheritance to ensure that they all follows the same fundamental interface

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

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

<u>Num: Numeric Literals</u>

- The simplest type of Python expression is a *literal*
    - E.g. <u>3</u>, <u>‘hello’</u>
- Numeric literals include <u>int</u>s and <u>float</u>s

```
class Num(Expr):
    “””A numeric literal.

    Instance Attributes:
        - n: the value of the literal
    “””
    n: Union[int, float]

    def __init__(self, number: Union[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 be 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
```

- Literals are the “base cases”, or “leaves” of an abstract syntax tree

BinOp: Arithmetic Operations

- In Python, an *arithmetic operation* is an expression that consists of 3 parts:
    - Left subexpression
    - Right subexpression
    - 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
```

- The BinOp class is a binary tree
    - Its root value is the operator name, and its left and right subtrees represent the two *operand subexpressions*
- Ex. 3 + 5.5 can be represented in BinOp(Num(3), ‘+’, Num(5.5))
- The left and right attributes of BinOp data type are Exprs (not Nums)
    - This makes this data type recursive, and allows it to represent nested arithmetic operations
- Ex. ((3 + 5.5) * (0.5 + (15.2 * -13.3))) can be represented in

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

- To *evaluate* a binary operation, we first evaluate its left and right operands, and then combine them using the specified arithmetic operator

```
class BinOp(Expr):
    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.

        >>> 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}')
```

<u>Recursion Multiple AST Classes</u>

- <u>BinOp.evaluate</u> actually uses recursion in a subtle way
- Because we are using multiple subclasses, there are multiple <u>evaluate</u> methods, one in each subclass

    - Each time self.left.evaluate and self.right.evaluate are called, they could either refer to BinOp.evaluate or Num.evaluate, depending on the types of self.left and self.right
- Num.evaluate does *not* make any subsequent calls to evaluate, since it just returns the object's n attribute
    - This is the base case of evaluate
        - It is located in a completely different method than BinOp.evaluate
- evaluate is a structural recursion, just one that spans multiple Expr subclasses

14.2 Variables and the Variable Environment

Variables and the Name Class

- Consider the expression x + 5.5
    - x is a *variable*, which is neither a numeric literal nor a nested arithmetic expression

```
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_
```

- We can now represent the expression x + 5.5 by BinOp(Name('x'), '+', Num(5.5))

Evaluating Variables by Dictionary Lookup

- Suppose we type x + 5.5 into the Python console, one of the below will happen:
    - x hasn't been defined yet, and we get a NameError
    - What we get depends on what value x was assigned to earlier in the console
        - This requires a *mapping* between variable names and values
            - The Python interpreter uses dict
- We call such dictionary the *variable environment* and call each key-value pair in the environment a *binding* between a variable and its current value

- We need to modify our <u>Expr.evaluate</u> method header so that it takes an additional argument, <u>env</u>, which contains all of the current variable bindings that can be used when evaluating the expression

```
class Expr:
        def evaluate(self, env: dict[str, Any]) -> Any:
                “””…”””
                raise NotImplementedError
```

    - Also need to add a new parameter <u>env</u> to all subclasses
- We can implement <u>Name.evaluate</u> with the new parameter

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

                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 = BinOpe(expr, ‘+’, Num(5.5))
        >>> binop.evaluate({‘x’: 100})
        105.5
```

- Our environments comes from the assignment statements

14.3 From Expressions to Statements

<u>The Statement Abstract Class</u>

- Expressions are statements, but not all statements are expressions
- When we evaluate an expression, we expect to get a value returned

- When we evaluate a statement, we often do not get back a value, but instead some other effect
- We can create a new abstract class <u>Statement</u>, that is a parent of <u>Expr</u>

```
class Statement:
    “””An abstract class representing a Python statement.

    We think of a Python statement as being a more general piece of code than a
    single expression, and that can have some kind of “effect”.
    “””
    def evaluate(self, env: dict[str, Any]) -> Optional[Any]:
        “””Evaluate this statement with the given environment.

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

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

def Expr(Statement):
    …
```

<u>Assign: An Assignment Statement</u>

- To build up the variable environment, we need to represent assignment statements as a new data type that we’ll call <u>Assign</u>

```
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
```

- We can represent the statement <u>y = x + 5.5</u> by

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

- To assign a variable, we need to mutate <u>env</u>

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

<u>Print: Displaying Text to the User</u>

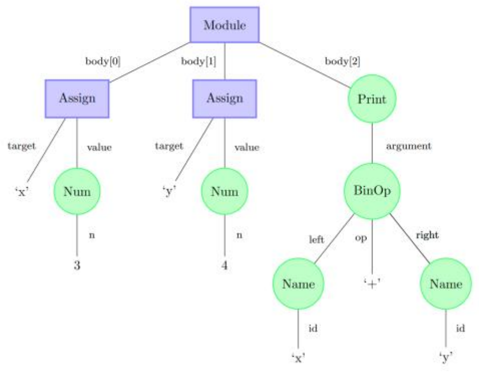
- Represents a call to the function <u>print</u>

```
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))
```

<u>Module: A Sequence of Statements</u>

- We can put statements together
- We will define a new class called <u>Module</u>, which represents a full Python program, consisting of a sequence of statements

```
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
```
- Consider the following Python program
```
        x = 3
        y = 4
        print(x + y)
```
- The above can be represented as
```
        Module([
                Assign('x', Num(3)),
                Assign('y', Num(4)),
                Print(BinOp(Name('x'), '+', Name('y')))
        ])
```
- Module is *not* a subclass of Statement since a Module can't be nested recursively within other Modules
- We can think of a Module as being the *root* of a complete abstract syntax tree



Evaluating Modules

- To evaluate a module, we do two things:
    - 1. Initialize an empty dictionary to represent the environment (starting with no variable bindings)
    - 2. Iterate over each statement of the module body and evaluate it

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

Control Flow Statements

- We can define a restricted form of an if statement that has just two branches – if and else

```
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 if no `else` block)
    """
    test: Expr
    body: list[Statement]
    orelse: list[Statement]
```

- We can represent a for loop over a range of numbers

```
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.
            - body: The statements to execute in the loop body.
    """
    target: str
    start: Expr
    stop: Expr
    body: list[Statement]
```

14.4 Abstract Syntax Trees in Practice

Python’s ast Module

- Python has a built-in module called ast that uses the same approach as this chapter, but covers the entire spectrum of the Python language

Code Analysis

- Abstract syntax trees can be used to analyse a program’s code without running it
  - Known as *static program analysis*
- Examples
  - Check for common errors
    - PyCharm, PythonTA, pylint
  - Identify unused or redundant code
  - Check the types of expressions and definitions
    - mypy
  - Check for common security and efficiency problems

Code Manipulation and Generation

- In PyCharm
  - Autocompletion of variables and attributes as we type
  - “Smart” renaming of functions and variables, that automatically renames all uses of the function or variable being renamed
  - Reorganizing import statements and removing unused imports

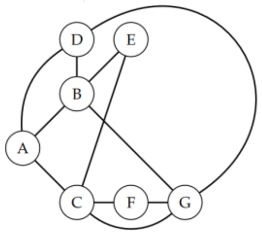
Transpilers

- We can transform an abstract syntax tree in one language to an equivalent abstract syntax tree in another
- Allows us to develop tools to translate code from one programming language to another, or between different versions of the same programming language

- E.g. Different web browsers support different versions of the JavaScript language, and often lag behind the latest JavaScript version
    - Tools like Babel translate between newer versions of JavaScript to older versions, allowing programmers to write code using the latest JavaScript features

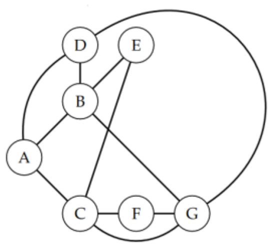
15.1 Introduction to Graphs

Graphs

- $Definition.$ 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 call the the set of *vertices* of the graph
  - E is a set of pairs of objects from V, where each pair $\{v_1, v_2\}$ is a set consisting of 2 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
- The conventional notation to introduce a graph is to write $G = (V, E)$, where G is the graph itself, V is its vertex set, and E is its edge set
- The set of vertices of a graph represents a collection of objects
- The set of edges of a graph represent the relationships between those objects
- Ex. to describe Facebook:
  - Each Facebook user is a *vertex*
  - Each friendship between two Facebook users is an *edge* between the corresponding vertices
- We often draw graphs using:
  - Dots to represent vertices
  - Line segments to represent edges
- Ex. 

  - 7 vertices
  - 11 edges
- Graphs are generalization of trees

    - Rather than enforcing a strict hierarchy on the data, graphs support any vertex being joined by an edge to any other vertex
- $Definition.$ Let $G = (V, E)$, and let $v_1, v_2 \in V$. We say that v_1 and v_2 are *adjacent* if and only if there exists an edge between them, i.e. $\{v_1, v_2\} \in E$
    - Equivalently, we can also say that v_1 and v_2 are *neighbours*
- $Definition.$ Let $G = (V, E)$ and let $v \in V$. We say that the *degree* of v, denoted $d(v)$, is its number of neighbours, or equivalently, how many edges v is part of

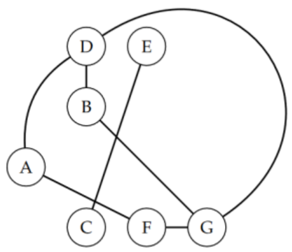


- In the same example
    - A and B are adjacent
    - A and G are not adjacent
    - Degree of vertex A is 3

Paths and Connectedness

- $Definition.$ Let $g = (V, E)$ and let $u, u' \in V$. A *path* between u and u' is a sequence of *distinct* vertices $v_0, v_1, v_2, \dots, v_k \in V$ which satisfy the following properties:
    - $v_0 = u$ and $v_k = u'$ (i.e. the endpoints of the path are u and u')
    - Each consecutive pair of vertices are adjacent (i.e. v_0 and v_1 are adjacent, etc.)
- We allow k to be 0, this path would be just a single vertex v_0
- The *length* of a path is one less than the number of vertices in the sequence
    - i.e. the above sequence would have length k
- The length of the path is the number of *edges* which are used by this sequence
- We say that u and u' are *connected* when there exists a path between u and u'
    - A vertex is always connected to itself
- We say that graph G is *connected* when for all pairs of vertices $u, v \in V$, u and v are connected

Example



- A and B are not adjacent
- A and B are connected (E.g. A, F, G, B)
- The length of the shortest path between vertices B and F is 2 (i.e. B, G, F)

Prove that this example graph is not connected

- $Translation$. Let $G = (V, E)$ refer to this graph. G is not connected means:
    - $\neg(G \text{ is connected})$
    - $\neg(\forall u, v \in V, u \text{ and } v \text{ are connected})$
    - $\exists u, v \in V$, u and v are not connected
    - $\exists u, v \in V$, there is no path between u and v
- $Proof$. Let $G = (V, E)$ be the above graph. Let u and v be the vertices labelled E and B, respectively. We will show that there does not exist a path between u and v.
  Suppose for a contradiction that there exists a path $v_0, v_1, \dots, v_k$ between u and v, where $v_0 = E$. Since v_0 and v_1 must be adjacent, and C is the only vertex adjacent to E, we know that $v_1 = C$. Since we know $v_k = B$, the path cannot be over yet; i.e. $k \geq 2$.
  By the definition of *path*, we know that v_2 must be adjacent to C, and must be distinct from E and C. But the only vertex that's adjacent to C is E, and so v_2 cannot exist, which gives us our contradiction.
  Q.E.D.

15.2 Some Properties of Graphs

<u>The Maximum Number of Edges in a Graph</u>

- **Ex.** Prove that for all graphs $G = (V, E), |E| \leq \frac{|V|(|V|-1)}{2}$
    - $Translation$. Notice that G is not an arbitrary *number*, but an arbitrary *graph*
    - $Discussion$. We cannot assume anything about the structure of the graph. We can use *induction* to prove statements about graphs, but in order to do so, we'll need to introduce a new variable that is a natural number
        - We can introduce a variable representing the *number of vertices* (i.e. $n = |V|$)
        - We will prove $\forall n \in \mathbb{N}, \forall\, G = (V, E), |V| = n \Longrightarrow |E| \leq \frac{n(n-1)}{2}$
    - $Proof$. We will prove this statement by induction on n. Our predicate is $P(n)\colon \forall G = (V, E), |V| = n \Longrightarrow |E| \leq \frac{n(n-1)}{2}$
        - **Base case:** let $n = 0$. We need to prove $P(0)$.
          Let $G = (V, E)$ 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 $|E| \leq \frac{(k+1)k}{2}$.
          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'. 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
          $$|E| = |E_1| + |E_2| \leq k + \frac{k(k-1)}{2} = \frac{(k+1)k}{2}$$
          Q.E.D.

The Transitivity of Connectedness

- If two vertices in a graph are both connected to a third vertex, then they are also connected to each other
- **Ex.** Let $G = (V, E)$ be a graph, and let $u, v, w \in V$. If v is connected to both u and w, then u and w are connected
    - $Translation$. We will use the predicate $Conn(G, u, v)$ to mean that "u and v are connected
        - $\forall G = (V, E), \forall u, v, w \in V, \big(Conn(G, u, v) \wedge Conn(G, v, w)\big) \Rightarrow Conn(G, u, w)$
    - $Discussion$. We have an arbitrary graph and three vertices in that graph. We assume that u and v are connected and that v and w are connected. We need to prove that u and w are also connected. We can create a path between u and w by joining the path between u and v and the one between v and w.
        - The problem is that the paths between u and v and v and w might contain some vertices in common, and paths are not allowe3d to have duplicate vertices
        - We can fix this by finding the first point of intersection between the paths and joining them at that vertex instead

- $Proof$. 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. (By the definition of connectedness, both of these paths must exist)
  Let $S \subseteq V$ be the set for 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 on P_2
  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. By the definition of connectedness, this means that u and w are connected.
  Q.E.D.

A Proof by Contradiction

- **Ex.** Prove that for all graphs $G = (V, E)$, if $|V| \geq 2$, then there exist two vertices in V that have the same degree.
  - $Translation.$ $\forall G = (V, E), |V| \geq 2 \Rightarrow \left(\exists v_1, v_2 \in V, d(v_1) = d(v_2)\right)$
  - $Proof$. 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. Let $n = |V|$.
    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 $\{0, 1, \dots, n - 1\}$ must be the degree of some vertex (note that this set has size n). In particular, there exists a vertex $v_1 \in V$ such that $d(v_1) = 0$, and another 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.
    Q.E.D.

15.3 Representing Graphs in Python

The _Vertex Class

- Represents a single vertex in graph

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

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

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

  - The item instance attribute stores the "data" in each vertex
  - The neighbours attribute stores a set of other _Vertex objects
    - This is where we encode the graph edges
    - Unordered

Enforcing Edge Restrictions

- Two restrictions
  - We cannot have a vertex with an edge to itself
  - All edges are symmetric
- Add two representation invariants

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

The Graph Class

- We define a Graph class that consists of a collection of _Vertex objects

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

    Precondition:
        - 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
```

<u>Checking Adjacency</u>

- Two common questions:

- “Are these two items adjacent?”
- “What items are adjacent to this item?”

```
class Graph
    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
```

15.4 Connectivity and Recursive Graph Traversal

Intro

- Our goal is to implement a generalization of the adjacent method that checks whether two vertices are connected

```
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.
        """
```

<u>Recursively Determining Connectivity</u>

- We can move the trickiest part (recursion) into a helper method

```
class Graph:
    def connected(self, item1: Any, item2: Any) -> bool:
        """..."""
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            return v1.check_connected(item2)
        else:
            return False

class _Vertex:
    def check_connected(self, target_item: Any) -> bool:
        """Return whether this vertex is connected to a vertex corresponding to
        the target_item.
        """
```

- Two vertices are connected when there exists a path between them
- Recursive definition of connectedness: given two vertices v_1 and v_2, they are connected when:
    - $v_1 = v_2$, or
    - there exists a neighbour u of v_1 such that u and v_2 are connected
- This recursion is not *structural* because it doesn't break down the data type into a smaller instance with the same structure
- Implementation of this definition

```
class _Vertex:
    def check_connected(self, target_item: Any) -> bool:
        """..."""
        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
```
- This implementation does not work

RecursionError and the Danger of Infinite Recursion

- Example: we have a graph g with 3 vertices, containing the items 1, 2, 3 (where v_1 and v_2 are connected to each other). In our call to g.connected(1, 3):
    - v_1 makes a recursive call to v_2
    - v_2 makes a recursive call to v_1
    - v_1 makes a recursive call to v_2
    - etc.
- This is an instance of *infinite recursion*
    - The recursive computation does not stop by reaching a base case
    - Each of these function calls adds a new *stack frame* onto the function call stack, which uses up more and more computer memory
    - The Python interpreter enforces a limit on the total number of stack frames that can be on the call stack at any one point in time
        - When the limit is reached, the Python interpreter raises a RecursionError

Fixing _Vertex.check_connected

- If v_1 and v_2 are connected, then we should be able to find a path between a neighbour u and v_2 that *doesn't* use v_1, and then add v_1 to the start of that path
- We can modify the definition: given two vertices v_1 and v_2, they 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
- We should be able to *remove* $v_1$from the graph and still find a path between u and v_2

Adding a Visited Parameter

- We keep track of the items that have been already visited by our algorithm, so that we don't 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]
                        return v1.check_connected(item2, set())
                        # Pass in an empty "visited" set
                else:
```

```
                return False
```

- We need to make two changes to _Vertex.check_connected
    - We add self to visited before making any recursive calls
        - To indicate that the current _Vertex has been visited by our algorithm
    - When looping over self.neighbours, we only make recursive calls into nodes that have not yet been visited

```
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 visited.

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

            return False
```

- With this version, we've eliminated our infinite recursion error

15.5 Cycles and Trees

A "Lower Bound" For Connectivity

- Q: What is the *minimum* number of edges required in a connected graph?
    - i.e. how many edges are *necessary* for a graph to be connected?
- If we have a graph with n vertices which is a path, it will have $n-1$ edges
    - If we remove an edge, the graph will be disconnected
- Any connected graph $G=(V,E)$ must have $|E| \geq |V|-1$

Cycles

- We need to characterize when we can remove an edge from a graph without disconnecting it
- $Definition.$ Let $G = (V, E)$ be a graph. A *cycle* in G is a sequence of vertices $v_0, \dots, v_k$ 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
- A cycle like a path but starts and ends at the same vertex
- The length of a cycle is the number of edges used by the sequence
- Two adjacent vertices are not considered to form a cycle
- If there is a cycle in the graph, it is possible to make a trip which starts and ends at the same vertex, and travels no vertex more than once
- Cycles are a form of *connectedness redundancy* in a graph
    - If one edge is removed, the result is a path
- $Lemma.$ 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
    - $Translation.$ We will use $G - e$ to represent the graph obtained by removing edge e from G
        - $\forall G = (V, E), \forall e \in E, (G \text{ is connected} \land e \text{ is in a cycle of } G) \Rightarrow G - e \text{ is connected}$
    - $Discussion.$ If we start with a connected graph and remove an edge in a cycle, then the resulting graph is still connected. We can make an argument based on the *transitivity of connectedness*
    - $Proof.$ 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 $G' = (V, E \setminus \{e\})$ be a graph formed from G by removing edge e. We want to prove that G' is also connected, i.e. that any two vertices in V are connected in G'.
      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', i.e. there is a path in G' between w_1 and w_2.
      Let P be a path between w_1 and w_2 in G (such a path exists by the definition of connectedness). We divide our proof into two cases: one where P uses the edge e, and another where it does not.
      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 we can divide the path P into three parts: P_1, the part from w_1 to u; the edge $\{u, v\}$; P_2, the part from v to w_2. Since P_1 and P_2 cannot use the edge $\{u, v\}$, they must be paths in G' as well. 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), and so by the *transitivity of connectedness*, w_1 and w_2 are connected in G'.
Q.E.D.

Graphs With No Cycles

- $Definition$. Let $G = (V, E)$ be a graph. We say that G is a *tree* when it is connected and has no cycles.
- This graph-based definition of tree doesn't have a designated root element, and we won't necessarily draw trees top-down
- Trees are "minimally-connected" graphs (i.e. the graphs which have the fewest number of edges possible but are still connected)
- Let G be a connected graph:
  - 1. If G has a cycle, then there exists an edge e in G such that $G - e$ is connected
    - True
  - 2. If G does not have a cycle, then there does not exist an edge e in G such that $G - e$ is connected
    - This is the converse for 1
- $Lemma$. Let G be a graph. If G does not have a cycle, then there does not exist an edge e in G such that $G - e$ is connected.
  - $Translation$.
    - $\forall G = (V, E), G$ does not have a cycle $\Longrightarrow \neg(\exists e \in E, G - e$ is connected$)$
    - Proving there does *not* exist some object satisfying some given conditions is challenging, and so we can prove the contrapositive
    - $\forall G = (V, E), (\exists e \in E, G - e \text{ is connected}) \Longrightarrow G$ has a cycle
  - $Discussion$. If we remove e, we remove one possible path between its endpoints. But since the graph must still be connected after removing e, there must be another path between its endpoints
  - $Proof$. Let $G = (V, E)$ be a graph. Assume that there exists an edge $e \in E$ such that $G - e$ is still connected.
    Let $G' = (V, E \backslash \{e\})$ 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 to it is a cycle in G.
Q.E.D.

<u>Counting Tree Edges</u>

- $Theorem.$ $(Number\ of\ edges\ in\ a\ tree)$ Let $G = (V, E)$ be a tree. Then $|E| = |V| - 1$.
    - $Translation.$ $\forall G = (V, E), G$ is a tree $\Longrightarrow |E| = |V| - 1$.
    - $Discussion.$ We can take a tree, remove a vertex from it, and use induction to show that the resulting tree satisfies this relationship between its numbers of vertices and edges. To do this, we need to pick a vertex that is a *leaf* of the tree (i.e. a vertex that has degree 1) to ensure that we get a tree after the removal. We will prove a lemma first.
- $Lemma.$ Let $G = (V, E)$ be a tree. If $|V| \geq 2$, then G has a vertex with degree 1.
    - $Translation.$ $\forall G = (V, E), (G$ is a tree $\land |V| \geq 2) \Longrightarrow (\exists v \in V, d(v) = 1)$
    - $Discussion.$ A vertex has degree 1 means that it is at the "end" of a tree. Suppose we start at an arbitrary vertex, and traverse edges to try to get as far away as possible. Because there are no cycles, we cannot revisit a vertex. But the path has to end somewhere, so the endpoint must have just 1 neighbour.
    - $Proof.$ 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 1 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, i.e. the path between v and u has maximum possible length (compared to paths between u and any other vertex). We will prove that v has exactly 1 neighbour.
      Let P be the shortest path between v and u. We know that v has *at least* 1 neighbour: the vertex immediately before it on P. v cannot be adjacent to any other vertex on P, as otherwise G would have a cycle. Also, v cannot be adjacent to any other vertex w *not* on P, as otherwise we could extend P to include w, and this would create a longer path.
      And so v has exactly one neighbour (the one on P immediately before v).
      Q.E.D.
- With the above lemma, we can prove the <u>number of edges in a tree theorem</u>
    - We will use induction, removing from the original graph a vertex with just 1 neighbour, so that the number of edges also only changes by 1.
    - Statement we'll prove:

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

- $Proof.$ We will proceed by induction on n, the number of vertices in the tree. Let $P(n)$ be the following predicate (over positive integers):
  $$P(n): \forall G = (V, E), (G \text{ is a tree} \wedge |V| = n) \Longrightarrow |E| = n - 1$$
  We want to prove that $\forall n \in \mathbb{Z}^+, P(n)$.
  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, i.e. for all graphs $G = (V, E)$, if G is a tree and $|V| = k$, then $|E| = k - 1$. We want to prove that $P(k + 1)$ is also true. Unpacking $P(k + 1)$, we get
  $$\forall G = (V, E), (G \text{ is a tree} \wedge |V| = k + 1) \Longrightarrow |E| = k$$
  Let $G = (V, E)$ be a tree, and assume $|V| = k + 1$. We want to prove that $|E| = k$.
  By the previous tree lemma, since $k + 1 \geq 2$, there exists a vertex $v \in V$ that has exactly 1 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.
  Q.E.D.

Putting It All Together

- Since every graph contains at least 1 tree (just keep removing edges in cycles until we cannot remove any more), this constraint on the numbers of edges in a tree translates immediately into a lower bound on the number of edges in any connected graph (in terms of the number of vertices of that graph)
- $Theorem.$ Let $G = (V, E)$ be a graph. If G is connected, then $|E| \geq |V| - 1$

15.6 Computing Spanning Trees

Spanning Trees and the Problem Specification

- $Definition.$ Let $G = (V, E)$ be a *connected* graph. Let $G' = (V, E')$ be another graph with the same vertex set as G, and where $E' \subseteq E$, i.e. its edges are a subset of the edges of G. We say that G' is a *spanning tree* of G when G' is a tree.
  - Every connected graph has at least 1 spanning tree
- Goal: implement a Graph method that computes a spanning tree for the graph
  - We will focus on just returning a *subset of the edges* (instead of a new graph)

```
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 does not create new edges).

        Preconditions:
            - this graph is connected
        “””
```

<u>A Brute Force Approach</u>

- We can start with all of its edges and then try finding a cycle. If we find a cycle, remove one of its edges, and repeat until there are no more cycles
- Inefficient because finding a cycle is complex

<u>A First Step: Printing Out Connected Vertices</u>

- We can print out all of the vertices that <u>self</u> is connected to

```
class _Vertex
    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:
            if u not in visited:
                u.print_all_connected(visited, d + 1)
```

  - By printing anything, we can see the recursive call structure that this method traces

    - The recursive call structure forms a tree that spans all of the vertices that the starting vertex is connected to

A Spanning Tree Algorithm

- Two key changes we'll need to make comparing to the above method
    - 1. Each recursive call will now return a list of edges, so we'll need an accumulator to keep track of them
    - 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:
            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
```

- Use the above method as a helper to implement our original Graph.spanning_tree method
    - We can choose any starting vertex we want because we assume that our graph is connected

```
class Graph
    def spanning_tree(self) -> list[set]:
        """..."""
        # 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())
```

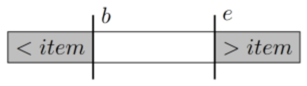
- Different starting vertices can result in different spanning trees
  - We don't care about being able to predict what spanning tree gets returned for now

16.1 Sorted Lists and Binary Search

Binary Search

- If we have a sorted list and want to search for an item, we can take the middle of the list and compare with the item, determine which one is greater, then cut down the list by half and repeat
  - This algorithm is known as *binary search*
  - At every comparison, the range of elements to check is halved

Implementing Binary Search

- We need to keep track of the current search range, which gets smaller every time a new item is checked
  - We can use two variables, b and e to represent the endpoints of this range
  - At any point in time in our algorithm, the current search range is the list slice lst[b:e]
- Our list will be divided into 3 parts
  - lst[0:b] contains only items that are known to be *less than* the item being searched for
  - lst[b:e] is the current search range
  - lst[e:len(lst)] contains only items that are known to be *greater than* the item being searched for
  - 

- At the start of our search, the full list is being searched. The binary search algorithm uses a *while loop* to decrease the size of the range
  - 1. First, we calculate the midpoint m of the current range
  - 2. Then, we compare lst[m] against item
    - If item == lst[m], we can return True right away

        - If item < lst[m], we know that all indexes >= m contains elements larger than item, and so update e to reflect this
        - If item > lst[m], we know that all indexes <= m contain elements less than item, and so update b to reflect this
- The loop should stop when lst[b:e] is empty (i.e. when b >= e)

```
def binary_search(lst: list, item: Any) -> bool:
    """Return whether item is in lst using the binary search algorithm."""
    b = 0
    e = len(lst)

    while b < e:
        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
```

Loop Invariants

- all(lst[i] < item for i in range(0, b))
- all(lst[i] > item for i in range(e, len(lst)))

Running Time

- Two loop variables: b and e that change over time in unpredictable ways
- 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
        - This requires a formal proof
- And so binary_search runs for at most $1 + \log_2 n$ iterations, with each iteration taking constant time
- Worst case running time is $\mathcal{O}(\log n)$

16.2 Selection Sort

The Algorithm Idea

- Given a collection of unsorted elements, we repeatedly extract the smallest element from the collection, building up a sorted list from these elements
- At each step, we *select* the smallest element from the ones remaining
- **Ex.** Suppose we start with the list [3, 7, 2, 5]
    - The smallest element is 2, so that becomes the first element of our sorted list
    - The remaining elements are [3, 7, 5]. The smallest element remaining is 3, and so that is the next element of our sorted list: [2, 3]
    - Etc.

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]

```
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
```

- Works but mutates the input lst
    - Can be fixed by making a copy of lst

In-Place Selection Sort

- A sorting algorithm is *in-place* when it sorts a list by mutating its input list, and without using any additional list objects (e.g. by creating a copy of the list)
- In-place algorithms may still use new computer memory to store primitive values like integers
    - This additional amount of memory is constant with respect to the size of the input list
    - The amount of additional memory used is $\Theta(1)$

- To implement an in-place version of selection sort, we cannot use a new list to accumulate the sorted values
    - We will move elements around in the input list, so that at iteration i in the loop, the first i elements of the list are the sorted part, and the remaining parts are unsorted
- **Ex.** [3, 7, 2, 5]
    - Iteration 0: [|3, 7, 2, 5]
        - Entire list unsorted
        - Find the smallest element in the list (2), and *swap* it with the item at index 0, obtaining the list [2, 7, 3, 5]
    - Iteration 1: [2, |7, 3, 5]
        - [7, 3, 5] unsorted
        - We move the 3 to index 1, obtaining the list [2, 3, 7, 5]
    - [2, 3, |7, 5]
        - Same thing happens
    - [2, 3, 5, |7]
    - [2, 3, 5, 7 |]
    - Sorted before |, unsorted after |
        - Loop variable i right after |

Loop Invariants

- We use the variable i to represent the boundary between the sorted and unsorted parts of the list

```
assert is_sorted(lst[:i])
```

- At iteration i, the first i items must be smaller than all other items in the list

```
assert i == 0 or all(lst[i – 1] < lst[j] for j in range(i, len(lst)))
```

Implementing the Loop

- At iteration i, we need to find the smallest item in the 'unsorted' section of the list, which is lst[i:], and swap it with lst[i]

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

        Note that this is a *mutating* function.
        """
        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
```

Running-Time Analysis for Selection Sort

- We will ignore the assert statements
- Analysis for _min_index: let n be the length of the input list lst
    - The statements outside the loop take constant time. We’ll treat them as 1 step.
    - The loop iterates $n - i - 1$ times, for $j = i + 1, \dots, n - 1$, and the body takes constant time (1 step). So the running time of the loop is $n - i - 1$ steps
    - So the total running time of _min_index is $(n - i - 1) + 1$, which is $\Theta(n - i)$.
- Analysis for selection_sort: let n be the length of the input list lst
    - Inside the body of the loop, there are two statements. The first statement is the call to _min_index, which takes $n - i$ steps, where i is the value of the for loop variable
    - The second statement (swapping the lst values) takes constant time, so we’ll count that as 1 step
    - So the running time of 1 iteration of the for loop is $n - i + 1$, and the for loop iterates once for each i between 0 and $n - 1$, inclusive
    - This gives us a total running time of

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

- Therefore the running time of selection_sort is $\Theta(n^2)$

16.3 Insertion Sort

Insertion Sort Idea

- The in-place version of this algorithm uses a loop, with a loop variable i to keep track of the boundary between the sorted and unsorted parts of the input list lst
- At loop iteration i, the sorted part is lst[:i]
- Insertion sort always takes the next item in the list, lst[i], and *inserts* it into the sorted part by moving it into the correct location to keep this part sorted
- **Ex.** [3, 0, 1, 8, 7]
  - Iteration i = 0: [|3, 0, 1, 8, 7]
    - lst[0] is 3
    - Since every sublist of length 1 is sorted, we do not need to reposition any items
    - After this iteration, lst[:1] is sorted
  - Iteration i = 1: [3, |0, 1, 8, 7]
    - lst[1] is 0
    - Sorted part s [3]
    - We need to swap 0 and 3 to make lst[:2] sorted
  - Iteration i = 2: [0, 3, |1, 8, 7]
    - lst[2] is 1
    - Sorted part is [0, 3]
    - We need to swap 1 and 3 to make lst[:3] sorted
  - Iteration i = 3: [0, 1, 3, |8, 7]
    - lst[3] is 8
    - Sorted part is [0, 1, 3]
    - We don't need to make any swaps (since 8 > 3)
    - The sorted part is lst[:4]
  - Iteration i = 4: [0, 1, 3, 8, |7]
    - lst[4] is 7
    - We need to swap 8 and 7 to make lst[:5] sorted
    - After this iteration is over, the entire list is sorted
  - [0, 1, 3, 7, 8|]
  - Sorted before |, unsorted after |

        - Loop variable i right after |

Loop Structure and Invariant

- Structure similar to selection sort
- Loop invariant:

  ```
  assert is_sorted(lst[:i])
  ```

    - We don't have the second loop invariant saying that lst[:i] contains the i smaller numbers in the list
        - Insertion sort always take the next number in the list, not the smallest number

Implementing the Loop

- We'll use a helper function _insert that performs the insertion part

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

    Preconditons:
        - 0 <= i < len(lst)
        - is_sorted(lst[:i])
    “””
    # Version 1, using an early return
    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
    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
```

- With the _insert function complete, we can simply call it inside our main insertion_sort loop

```
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 for Insertion Sort

- We'll analyse the 'early return' implementation
- *Running-time analysis* (for _insert). Because of the early return, the function can stop early, and so has a spread of running times. We'll need to analyse the worst-case running time
    - Upper bound: let $n \in \mathbb{N}$ and let lst be an arbitrary list of length n. Let $i \in \mathbb{N}$ and assume $i < n$. The loop runs *at most* i times (for $j = i, i - 1, \dots, 1$), and each iteration takes constant time (1 step). The total running time of the loop, and therefore the function, is *at most* i steps. Therefore the worst-case running time of _insert is $\mathcal{O}(i)$.
    - For a matching lower bound, consider an input list lst where lst[i] is greater than all items in lst[:i]. In this case, the expression lst[j – 1] <= lst[j] will always be false, and so the loop will only stop when j == 0, which takes i iterations. So for this input, the running time of _insert is i steps, which is $\Omega(i)$, matching the upper bound.
    - So the worst-case running time of insert is $\Theta(i)$.
- *Running-time analysis* (for insertion_sort). We'll analyse the worst-case.
    - Upper bound: let $n \in \mathbb{N}$ and let lst be an arbitrary list of length n.
        - The loop iterates n times, for $i = 0, 1, \dots, n - 1$. In the loop body, the call to _insert(lst, i) counts as *at most* i steps (since its worst-case running time is $\Theta(i)$). We'll ignore the running time of the assert statement, so the running time of the entire iteration is just i steps.
        - This gives an upper bound on the worst-case running time of $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$, which is $\mathcal{O}(n^2)$.
    - For a matching lower bound, let $n \in \mathbb{N}$ and let lst be the list $[n - 1, n - 2, \dots, 1, 0]$.
        - For all $i \in \mathbb{N}$, this input has lst[i] greater than every element in lst[:i]. This causes each call to _insert to take i steps.

        - This gives a total running time for this input family of $\frac{n(n-1)}{2}$, which is $\Omega(n^2)$, matching the upper bound on the worst-case running time.
    - Therefore, we can conclude that the worst-case running time of <u>insertion_sort</u> is $\Theta(n^2)$

<u>Selection Sort vs. Insertion Sort</u>

- Both are *iterative* sorting algorithms
    - They are implemented using a loop
- Both use the same strategy of building up a sorted sublist within their input list, with each loop iteration increasing the size of the sorted part by 1
- Both have 'two phases':
    - Selecting which remaining item to add to the sorted part
        - Harder for selection sort
    - Adding that item to the sorted part
        - Harder for insertion sort
- Selection sort *always* takes $\Theta(n^2)$ steps, regardless of what input it has been given
- Insertion sort has a spread of running times
    - Faster for lists that are already sorted

16.4 Introduction to Divide-and-Conquer Algorithms

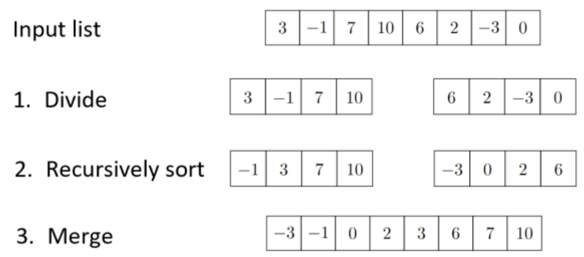
<u>A Divide-and-Conquer Approach</u>

- The *divide-and-conquer* approach to algorithm:
    - 1. Given the problem input, split it up into 2 or more smaller subparts with the same structure
    - 2. Recursively run the algorithm on each subpart separately
    - 3. Combine the results of each recursive call into a single result, solving the original problem
- A *divide-and-conquer* sorting algorithm:
    - 1. Given a list to sort, *split* it up into 2 or more smaller lists
    - 2. Recursively run the sorting algorithm on each smaller list separately
    - 3. *Combine* the sorted results of each recursive call into a single sorted list

16.5 Mergesort

<u>Intro</u>

- The *mergesort* algorithm:
    - 1. Given an input list to sort, divide the input into the left half and right half
    - 2. Recursively sort each half
    - 3. Merge each sorted half together
- The ‘easy’ part is dividing the input into halves
- The ‘hard’ part is merging each sorted half into one final sorted list



-

<u>Implementing Mergesort</u>

- We will use the non-mutating version
- Base case: ‘when can we not divide the list any further?’
    - Occurs when the list has less than 2 elements
- Recursive step: divide the list into two halves, recursively sort each half, and then merge the sorted halves back together
    - Merging the two sorted halves is complicated, and we need a helper

```
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 2 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
                return _merge(left_sorted, right_sorted)
```

<u>The Merge Operation</u>

- We can use the similar idea as selection sort: build up a sorted list one element at a time, by repeatedly removing the next smallest element from lst1 or lst2
- Because both lists are sorted, we can determine what the minimum element is by comparing lst1[0] and lst2[0]

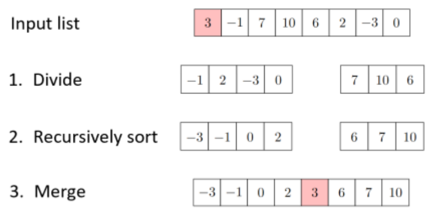
Unmerged items in lst1	**Unmerged items in lst2**	**Comparison**	**Sorted list**
[-1, 3, 7, 10]	[-3, 0, 2, 6]	-1 vs. -3	[-3]
[-1, 3, 7, 10]	[0, 2, 6]	-1 vs. 0	[-3, -1]
[3, 7, 10]	[0, 2, 6]	3 vs. 0	[-3, -1, 0]
[3, 7, 10]	[2, 6]	3 vs. 2	[-3, -1, 0, 2]
[3, 7, 10]	[6]	3 vs. 6	[-3, -1, 0, 2, 3]
[7, 10]	[6]	7 vs. 6	[-3, -1, 0, 2, 3, 6]
[7, 10]	[]	N/A	[-3, -1, 0, 2, 3, 6]

- In each row of the table, the sorted list is built up by one element, until we've exhausted on one of the lists
    - After this happens, we can concatenate the sorted list with the unmerged items from the remaining list, as the latter will all be >= the former
- Our final result is [-3, -1, 0, 2, 3, 6] + [7, 10], yielding the merged sorted list
- Use this idea in our _merge implementation
    - Use index variables to act as the boundary between the merged and unmerged items in lst1 and lst2

```
def _merge(lst1: list, lst2: list) -> list:
    """Return a sorted list with elements in lst1 and lst2.

    Preconditions:
        - is_sorted(lst1)
        - is_sorted(lst2)
    """
    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 lst[: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:]
```

16.6 Quicksort

<u>The Quicksort Implementation</u>

- 1. We pick one element from the input list, which we call the *pivot*. Then, we split up the list into 2 parts: the elements less than or equal to the pivot, and those greater than the pivot
    - This is the *partitioning* step
- 2. We sort each part recursively
- 3. We concatenate the two sorted parts, putting the pivot in between them
- We will chose the *first* item in the list as the pivot
    - 

- The “divide” step is difficult, the “combine” step is easy

```
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)

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

        # Combine the two sorted parts
        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)
```

<u>Mergesort vs. Quicksort</u>

- For mergesort, the "divide" step is simple, since the input list is simply split in half. The "combine" step is the hard part, where we use a _merge helper to merge the two sorted lists together

- For quicksort, the "divide" step is the complex one, where the input list is partitioned into two parts based on comparing the items to a chosen pivot value. But once that's done, the "combine" step is simple: we can just concatenate the results

16.7 Running-Time Analysis for Mergesort and Quicksort

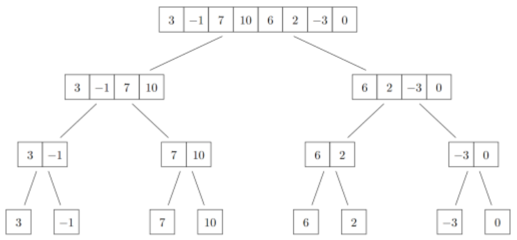
Intro

- Approach for analysing the running time of recursive tree operations:
  - 1. Find a pattern for the tree structure of recursive calls made for our recursive function
  - 2. Analyse the non-recursive running time of the recursive calls
  - 3. 'Fill in the tree' of recursive calls with the non-recursive running time, and then add up all of these numbers to obtain the total running time

Analysing Mergesort

```
def mergesort(lst: list) -> list:
    """..."""
    if len(lst) < 2:
        return lst.copy()  # Use the list.copy method to return a new list object
    else:
        # Divide the list into 2 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
        return _merge(left_sorted, right_sorted)
```

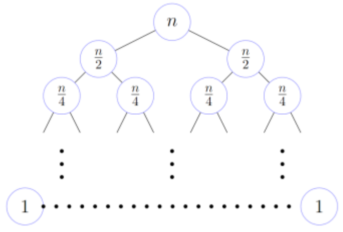
- Suppose we start with a list of length n, where $n > 1$. For simplicity, we assume that n is a power of 2.
- In the recursive step, we divide the list in half, obtaining two lists of length $\frac{n}{2}$, and recurse on each one.
  - Each of those calls divides their list in half, obtaining and recursing on lists of length $\frac{n}{4}$
  - This pattern repeats, with each call to mergesort taking as input a list of length $\frac{n}{2^k}$ (for some $k \in \mathbb{N}$) and recursing on two lists of length $\frac{n}{2^{k+1}}$
  - The base case is reached when mergesort is called on a list of length 1

- Ex. 
    - Binary tree where every non-base-case call to mergesort makes two more recursive calls
    - The input list decreases in size by a factor of 2 at each recursive call
    - There are $\log_2 n + 1$ levels in this tree

Non-Recursive Running Time

- Consider a list lst of length n, where $n \geq 2$. We will assume that n is a power of 2, which allows us to ignore floors/ceilings in our calculation.
- The if condition check (len(lst) < 2) and calculation of mid takes constant time
- The list slicing operations lst[:mid] and lst[mid:] each take time proportional to the length of the slice, which is $\frac{n}{2}$.
- The running time of _merge is $\Theta(n_1 + n_2)$, where n_1 and n_2 are the sizes of its two input lists
- So in the recursive step, left_sorted and right_sorted each have size $\frac{n}{2}$, the running time of _merge(left_sorted, right_sorted) is $\frac{n}{2} + \frac{n}{2} = n$ steps.
- So the total non-recursive running time of mergesort is $1 + \frac{n}{2} + \frac{n}{2} + n = 2n + 1$, when $n \geq 2$

Putting It All Together

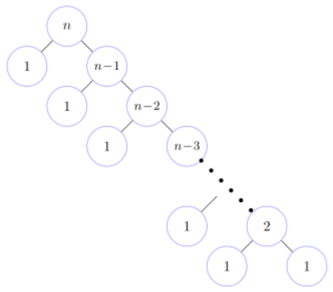
- We will use n instead of $2n + 1$ as the running time, essentially taking the 'simplified Theta bound' of the previous expression
    - The initial mergesort call on a list of length n takes n steps
    - Each of the recursive mergesort calls on a list of length $\frac{n}{2}$ takes $\frac{n}{2}$ steps
    - Then, each of the recursive mergesort calls on a list of length $\frac{n}{4}$ takes $\frac{n}{4}$ steps
    - This pattern continues, until we reach our base case (list of length 1), which takes 1 step.
    - 

- Key observation: 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 node contains the number $\frac{n}{2^d}$
    - When we 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 in total, and we get a total running time of $n \cdot (\log_2 n + 1)$, which is $\Theta(n \log n)$

<u>Quicksort</u>

```
def quicksort(lst: list) -> list:
    """..."""
    if len(lst) < 2::
        return lst.copy()
    else:
        # Partition the list using the first element as the pivot
        pivot = lst[0]
        smaller, bigger = _partition(lst[1:], pivot)

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

        # Combine the two sorted parts
        return smaller_sorted + [pivot] + bigger_sorted
```

- The *non-recursive running time* of quicksort is also $\Theta(n)$, where n is the length of the input list
- Quicksort also always makes 2 recursive calls, on its two partitions. Because of this, the recursive call tree is also a binary tree
- Quicksort does not necessarily split the list into two equal halves
- Suppose at each recursive call we choose the pivot to be the median of the list, so that the two partitions both have size roughly $\frac{n}{2}$
    - In this case, the recursion tree is same as mergesort, and we get a $\Theta(n \log n)$ runtime
- Consider another possibility: if we are always extremely unlucky, and always choose the smallest list element
    - <u>smaller</u> always have no elements, and <u>bigger</u> always have all remaining $n - 1$ elements

- 

    - The size of the bigger partition decreases by 1 at each call
    - There are $n-1$ recursive calls on empty partitions (smaller)
    - The total non-recursive running time is $\left(\sum_{i=1}^{n} i\right)+(n-1)=\frac{n(n+1)}{2}+n-1$
    - The running time is $\Theta(n^2)$
- It is possible to show that the worst-case running time for this implementation of quicksort is $\Theta(n^2)$ and the best-case is $\Theta(n \log n)$
- Choice of pivot is important

Mergesort and Quicksort in Practice

- With asymptotic Theta notation, all $\Theta(n \log n)$ look the same
- An *in-place* implementation of quicksort can be significantly faster than mergesort
    - For most inputs, quicksort has 'smaller constants' than mergesort
- For most inputs quicksort is faster than mergesort, despite having larger worst-case running time
- When we always choose a *random* element to be the pivot in the partitioning step, we can prove that quicksort has an *average-case running time* of $\Theta(n \log n)$, with smaller constant factors than mergesort's $\Theta(n \log n)$
- The actual 'bad' inputs for quicksort are quite rare
- There exists a linear-time algorithm for computing the median of a list of numbers
    - The *median of medians* algorithm
    - However, despite having a better worst-case running time, the extra steps used to calculate the median typically results in a slower running time for more inputs than simply choosing a random pivot
    - Rarely used

17.1 Introduction to Average-Case Running Time

Intro

- The worst-case running time says very little about the 'typical' number in the set, and nothing about the *distribution* of numbers in that set.

- The worst-case running time of quicksort is $\Theta(n^2)$, but it runs faster than mergesort on most inputs
  - The *average-case running time* of quicksort is $\Theta(n \log n)$

Running Time Sets, Worst-Case, and Average-Case

- Let func be a program. For each $n \in \mathbb{N}$, we define the set $\mathcal{I}_{func,n}$ to be the set of all inputs to func of size n
- We define the *worst-case running time* of func as the function

$$WC_{func}(n) = \max\{\text{running time of executing } func(x) \mid x \in \mathcal{I}_{func,n}\}$$

  - $WC_{func}(n)$ is the maximum running time of func on an input of size n
- We define the *average-case running time* of func in a similar way, except now taking the average rather than maximum of the set of running times
- The average-case running time of func is defined as the function $Avg_{func}: \mathbb{N} \to \mathbb{R}^+$, where

$$Avg_{func}(n) = \frac{1}{|\mathcal{I}_{func,n}|} \times \sum_{x \in \mathcal{I}_{func,n}} \text{running time of } func(x)$$

  - When there is only one input per size, or when all inputs of a given size have the same running time, the average-case running time is the same as the worst-case running time, as there is no spread

17.2 Average-Case Running Time of Linear Search

Intro

- The *linear search* algorithm searches for an item in a list by checking each list element one at a time

```
def search(lst: list, x: Any) -> bool:
    “””Return whether x is in lst.”””
    for item in lst:
        if item == x:
            return True
    return False
```

- Worst case is $\Theta(n)$, where n is the length of the list
- We need to precisely define what we mean by “all possible inputs of length n”
  - There could be an infinite number of lists of length n to choose from
  - We cannot take an average of an infinite set of numbers

- In average-case running-time analysis, we choose a particular set of allowable inputs, and then compute the average running time for that set

A First Example

- Let $n \in \mathbb{N}$. We'll choose our input set to be the set of inputs where:
    - lst is always the list [0, 1, 2, …, n – 1]
    - x is an integer between 0 and n – 1, inclusive
- i.e. we'll consider always searching in the same list [0, 1, …, n – 1] and search for one of the elements in the list
    - Use $\mathcal{I}_n$ to denote this set
- *Average-case running time analysis.* for this definition of $\mathcal{I}_n$, we know that $|\mathcal{I}_n| = n$, since there are n different choices for x (and just one choice for lst), From our definition of average-case running time, we have

$$Avg_{search}(n) = \frac{1}{n} \times \sum_{(lst,x)\in\mathcal{I}_n} \text{running time of } search(lst, x)$$

    - To calculate the sum, we need to compute the running time of search(lst, x) for every possible input. Let $x \in \{0, 1, 2, \ldots, n-1\}$. We'll calculate the running time in terms of x.
        - Since lst = [0, 1, 2, …, n – 1], we know that there will be exactly $x + 1$ loop iterations until x is found in the list, at which point the early return will occur and the loop will stop. Each loop iteration takes constant time, for a total of $x + 1$ steps
    - So the running time of search(lst, x) equals $x + 1$, and we can use this to calculate the average-case running time:

$$Avg_{search}(n) = \frac{1}{n} \times \sum_{(lst,x)\in\mathcal{I}_n} \text{running time of } search(lst, x)$$

$$= \frac{1}{n} \times \sum_{x=0}^{n-1} (x+1)$$

$$= \frac{1}{n} \times \sum_{x'=1}^{n} x' \qquad (x' = x + 1)$$

$$= \frac{1}{n} \times \frac{n(n+1)}{2}$$

$$= \frac{n+1}{2}$$

    - And so the average-case running time of search on this set of inputs is $\frac{n+1}{2}$, which is $\Theta(n)$

        - Note: we do not need to compute an upper and lower bound separately, since in this case we have computed an exact average
- For the given set of inputs $\mathcal{I}_n$ for each n, the average-case running time is asymptotically equal to that of the worst-case

A Second Example: Searching in Binary Lists

- We'll keep the same function (search) but choose a different input set to consider. Let $n \in \mathbb{N}$, and let $\mathcal{I}_n$ be the set of inputs (lst, x) where:
    - lst is a list of length n that contains only 0s and 1s
    - x = 0
- We call these lists *binary* lists, analogous to the binary representation of numbers
- lst could be any list of length n that consists of 0s and 1s
- *Average-case running time analysis.* We will divide this running-time analysis into 5 steps:
    - 1. Compute the number of inputs, $|\mathcal{I}_n|$
    - 2. divide the set of inputs into partitions based on the running time of each input
    - 3. compute the size of each partition
    - 4. Using Steps 2 and 3, calculate the sum of the running times for all the inputs in $\mathcal{I}_n$
    - 5. Using Steps 1 and 4, calculate the average-case running time for $\mathcal{I}_n$
    - *Step 1: compute the number of inputs.* Since x is always 0, the size of $\mathcal{I}_n$ is equal to the number of lists of 0s and 1s of length n. This is 2^n, since there are n independent choices for each of the n elements of the list, and each choice has two possible values, 0 and 1.
    - *Step 2: partitioning the inputs by their running time.* To add up the running times for each input, we'll first group them based on their running time.
        - If lst first has a 0 at index i, then the loop will take $i + 1$ iterations, and therefore this many steps. We can divide up the lists based on the index of the first 0 in the list, and treat the list [1, 1, …, 1] as a special case, since it has no 0s.
        - For each $i \in \{0, 1, \dots, n-1\}$, we define $S_{n,i} \subset \mathcal{I}_n$ to be the set of binary lists of length n where their first 0 occurs at index i. For example:
            - $S_{n,0}$ is the set of binary lists lst where lst[0] == 0
                - Every element in $S_{n,0}$ takes 1 step for search
            - $S_{n,1}$ is the set of binary lists lst where lst[0] == 1 and lst[1] == 0
                - Every element in $S_{n,1}$ takes 2 steps for search
            - $S_{n,i-1}$ is the set of binary lists lst where lst[j] == 1 for all natural numbers j that are <i, and lst[i] == 0

        - Every element in $S_{n,i}$ takes $i + 1$ steps for search
    - We'll define a special set $S_{n,n}$ that contains just the list [1, 1, …, 1]
        - This input causes search to take $n + 1$ steps: n steps for the loop, and then 1 step for the return False at the end of the function
- *Step 3: compute the size of each partition.* We will try to establish a pattern:
    - Consider $S_{n,0}$. A list lst in this set must have lst[0] == 0, but its remaining elements could be anything. So there are 2^{n-1} possible lists, since there are 2 choices for each of the remaining $n - 1$ spots. Therefore $|S_{n,0}| = 2^{n-1}$
    - Consider $S_{n,1}$. A list lst in this set has two "fixed" elements, lst[0] == 1 and lst[1] == 0. So there are $n - 2$ remaining spots that could contain either a 0 or 1, and so 2^{n-2} such lists in total. Therefore $|S_{n,1}| = 2^{n-2}$
    - In general, for $i \in \{0, 1, \dots, n-1\}$, lists in $S_{n,i}$ have $i + 1$ fixed spots, and the remaining $n - i - 1$ spots could be either 0 or 1. So $|S_{n,i}| = 2^{n-i-1}$
    - Finally, $|S_{n,n}| = 1$, since we defined that partition to only contain the all-1s list, and there is only one such list.
- *Step 4: calculate the sum of the running times.* To calculate the sum of the running times of all inputs in $\mathcal{I}_n$, we can group the inputs into their partitions:

$$\sum_{(lst,x)\in\mathcal{I}_n} \text{running time of } search(lst, x)$$

$$= \sum_{i=0}^{n} \sum_{lst\in S_{n,i}} \text{running time of } serach(lst, 0) \qquad \text{(note } x = 0\text{)}$$

    - Within a single partition $S_{n,i}$, each list has the same running time, and so the body of the inner summation depends only on i, and is constant for the inner summation. We can use our work in steps 2 and 3 to simplify this summation:

$$\sum_{i=0}^{n} \sum_{lst\in S_{n,i}} \text{running time of } search(lst, 0)$$

$$= \sum_{i=0}^{n} \sum_{lst\in S_{n,i}} (i+1) \qquad \text{(from Step 2)}$$

$$= \sum_{i=0}^{n} |S_{n,i}| \cdot (i+1)$$

$$= \left(\sum_{i=0}^{n-1} |S_{n,i}| \cdot (i+1)\right) + |S_{n,n}| \cdot (n+1) \qquad \text{(splitting off last term)}$$

$$= \left(\sum_{i=0}^{n-1} 2^{n-i-1} \cdot (i+1)\right) + n + 1 \qquad \text{(from Step 3)}$$

  - We'll use the *arithmetic-geometric* summation formula (valid for all $r \in \mathbb{R}$ if $r \neq 1$): $\sum_{i=0}^{m-1} i \cdot r^i = \frac{m \cdot r^m}{r-1} - \frac{r(r^m - 1)}{(r-1)^2}$

$$\left(\sum_{i=0}^{n-1} 2^{n-i-1} \cdot (i+1)\right) + n + 1$$
$$= \left(\sum_{i'=1}^{n} 2^{n-i'} \cdot i'\right) + n + 1 \qquad (i' = i + 1)$$
$$= 2^n \left(\sum_{i'=1}^{n} \left(\frac{1}{2}\right)^{i'} \cdot i'\right) + n + 1$$
$$= 2^n \left(\frac{\frac{n+1}{2^{n+1}}}{-\frac{1}{2}} - \frac{\frac{1}{2}\left(\frac{1}{2^{n+1}} - 1\right)}{\frac{1}{4}}\right) + n + 1 \qquad \text{(Using the formula)}$$
$$= 2^n \left(-\frac{n+1}{2^n} - \frac{1}{2^n} + 2\right) + n + 1$$
$$= -(n+1) - 1 + 2^{n+1} + n + 1$$
$$= 2^{n+1} - 1$$

  - Our total running time for all inputs in $\mathcal{I}_n$ is $2^{n+1} - 1$

- *Step 5: putting everything together.*

$$Avg_{serach}(n) = \frac{1}{|\mathcal{I}_n|} \times \sum_{(lst,x) \in \mathcal{I}_n} \text{running time of } search(lst, x)$$
$$= \frac{1}{2^n} \cdot (2^{n+1} - 1)$$
$$= 2 - \frac{1}{2^n}$$

- Our average-case running time is $2 - \frac{1}{2^n}$ steps, which is $\Theta(1)$

<u>Average-Case Analysis and Choosing Input Sets</u>

- Our two average-case running time analyse for <u>search</u> gave two different results: a $\Theta(n)$ and a $\Theta(1)$
- The average-case running time is always dependent on the set of inputs we choose to analyse
- Neither of the above two analyses is more "correct" than the other, but instead differ in how accurately they reflect the algorithm's actual inputs in practice