Week06_Homework

October 25, 2021

1 Week 6 Problem Set

1.1 Homeworks

HW1. Extend the class `Fraction` to implement the other operators: `- * < <= > >=`.

```
In [18]: def gcd(a, b):
             if b == 0:
                 return a
             else:
                 return gcd(b, a % b)


         class Fraction:
             def __init__(self, num, den):
                 self.num = num
                 self.den = den

             @property
             def num(self):
                 return self._numattr

             @num.setter
             def num(self, value):
                 self._numattr = int(value)

             @property
             def den(self):
                 return self._denattr

             @den.setter
             def den(self, value):
                 # ask about property and how it influences the attribute
                 if value == 0:
                     self._denattr = 1
                 else:
                     self._denattr = int(value)
```

```
def __str__(self):
    return f"{self.num}/{self.den}"

def __add__(self, other):
    num = self.num * other.den + self.den * other.num
    den = self.den * other.den
    return Fraction(num, den).simplify()

def simplify(self):
    common = gcd(self.num, self.den)
    num = self.num // common
    den = self.den // common
    return Fraction(num, den)

def __eq__(self, other):
    left = self.simplify()
    right = other.simplify()
    return left.num == right.num and left.den == right.den

def __sub__(self, other):
    num = self.num * other.den - other.num * self.den
    den = self.den * other.den

    return Fraction(num, den).simplify()

def __mul__(self, other):
    num = self.num * other.num
    den = self.den * other.den
    return Fraction(num, den).simplify()

def __eq__(self, other):
    left = self.simplify()
    right = other.simplify()
    return left.num == right.num and left.den == right.den

def __lt__(self, other):
    if self.num * other.den - other.num * self.den < 0:
        return True
    else:
        return False

def __le__(self, other):
    if self.num * other.den - other.num * self.den <= 0:
        return True
    else:
        return False

def __gt__(self, other):
```

```
            if self.num * other.den - other.num * self.den > 0:
                return True
            else:
                return False

        def __ge__(self, other):
            if self.num * other.den - other.num * self.den >= 0:
                return True
            else:
                return False
```

```
In [19]: f1 = Fraction(3, 4)
         f2 = Fraction(1, 2)
         f3 = f1 - f2
         assert f3 == Fraction(1, 4)
         f4 = f1 * f2
         assert f4 == Fraction(3, 8)
         assert f2 < f1
         assert f2 <= f2
         assert f1 > f3
         assert f3 >= f3
```

```
In [20]: f1 = Fraction(3, 4)
         f2 = Fraction(1, 2)
         f3 = f1 - f2
         assert f3 == Fraction(1, 4)
         f4 = f1 * f2
         assert f4 == Fraction(3, 8)
         assert f2 < f1
         assert f2 <= f2
         assert f1 > f3
         assert f3 >= f3
```

```
In [21]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

```
In [22]: class MixedFraction(Fraction):
             def __init__(self, top, bot, whole=0):
                 num = top + whole * bot
                 self.whole = whole
                 super().__init__(num, bot)

             @property
             def whole(self):
                 return self._whole

             @whole.setter
```

```
    def whole(self, value):
        if isinstance(value,int):
            self._whole = value

    def get_three_numbers(self):
        if self.num > self.den:
            return self.num % self.den, self.den, self.num // self.den
        else:
            return self.num, self.den, self.whole

    def __str__(self):
        top, bot, whole = self.get_three_numbers()
        if whole > 0:
            return "{} {}/{}".format(whole, top, bot)
        else:
            return super().__str__()
```

```
In [23]: mf1 = MixedFraction(5, 3)
         assert mf1.num == 5 and mf1.den == 3
         assert mf1.get_three_numbers() == (2, 3, 1)
         mf2 = MixedFraction(2, 3, 1)
         assert mf2.num == 5 and mf2.den == 3

         result = mf1 + mf2
         assert result.num == 10 and result.den == 3

         result = mf1 * mf2
         assert result.num == 25 and result.den == 9

         mf3 = MixedFraction(1, 2, 1)
         result = mf1 - mf3
         assert result.num == 1 and result.den == 6

         assert str(mf1) == "1 2/3"
```

```
In [24]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

HW2. Write a class called `EvaluateFraction` that evaluates postfix notation implemented using Dequeue data structures only. Postfix notation is a way of writing expressions without using parenthesis. For example, the expression `(1+2)*3` would be written as `1 2 + 3 *`. The class `EvaluateFraction` has the following method: - `input(inp)`: which pushes the input input one at a time. For example, to create a postfix notation `1 2 + 3 *`, we can call this method repetitively, e.g. `e.input('1'); e.input('2'); e.input('+'); e.input('3'); e.input('*')`. Notice that the input is of String data type. - `evaluate()`: which returns the output of the expression. - `get_fraction(inp)`: which takes in an input string and returns a `Fraction` object.

Postfix notation is evaluated using a Stack. Since `Dequeue` can be used for both Stack and Queue, we will implement using `Dequeue`. The input streams from `input()` are stored in a Queue, which we will again implement using Dequeue. If the output of the Queue is a number, the item is pushed into the stack. If it is an operator, we will apply the operator to the two top most item n the stacks and push the result back into the stack.

```
In [25]: class Stack:
             def __init__(self):
                 self.__items = []

             def push(self, item):
                 self.__items.append(item)

             def pop(self):
                 return self.__items.pop(-1) if not self.is_empty else None

             def peek(self):
                 if self.is_empty:
                     return None
                 return self.__items[-1]

             @property
             def is_empty(self):
                 return len(self.__items) == 0

             @property
             def size(self):
                 return len(self.__items)
```

```
In [26]: class Queue:
             """ Queue using 2 stacks instead of python lists"""

             def __init__(self):
                 self.left_stack = Stack()
                 self.right_stack = Stack()

             @property
             def is_empty(self):
                 return self.left_stack.is_empty and self.right_stack.is_empty

             @property
             def size(self):
                 return self.left_stack.size + self.right_stack.size

             def enqueue(self, item):
                 self.right_stack.push(item)

             def dequeue(self):
                 if self.left_stack.is_empty:
                     while not self.right_stack.is_empty:
```

```
                    self.left_stack.push(self.right_stack.pop())
            return self.left_stack.pop()

        def peek(self):
            if self.left_stack.is_empty:
                while not self.right_stack.is_empty:
                    self.left_stack.push(self.right_stack.pop())
            x = self.left_stack.peek()
            return x
```

```
In [27]: class Dequeue(Queue):

             def add_front(self, item):
                 """Add item to the front of the queue"""
                 # push all items from right_stack to left_stack
                 while not self.right_stack.is_empty:
                     self.left_stack.push(self.right_stack.pop())
                 self.left_stack.push(item)
                 # push all items from left_stack to right_stack
                 while not self.left_stack.is_empty:
                     self.right_stack.push(self.left_stack.pop())

             def remove_front(self):
                 """Remove item from the front of the queue"""
                 return self.dequeue()

             def add_rear(self, item):
                 self.enqueue(item)

             def remove_rear(self):
                 return self.right_stack.pop()

             def peek_front(self):
                 return self.peek()

             def peek_rear(self):
                 return self.right_stack.peek()
```

```
In [28]: import operator

         class EvaluateFraction:

             operands = "0123456789"
             ops = {
                 '+': operator.add,
                 '-': operator.sub,
                 '*': operator.mul,
                 '/': operator.truediv,  # use operator.div for Python 2
```

```
            '%': operator.mod,
            '^': operator.xor}

    def __init__(self):
        self.expression = Dequeue()
        self.stack = Dequeue()

    def input(self, item):
        """ Enqueues items"""
        if item in self.ops:
            self.expression.add_rear(item)
        else:
            self.expression.add_rear(self.get_fraction(item))
        print(f"peek front {self.expression.peek_front()}")
        print(f"peek back {self.expression.peek_rear()}")

    def evaluate(self):
        while self.expression.size != 0:
            item = self.expression.remove_front()
            print(item)

            if type(item) == Fraction:
                self.stack.add_front(item)

            elif item in self.ops:
                op1 = self.stack.remove_front()
                op2 = self.stack.remove_front()
                print(op1, op2)
                product = self.ops[item](op2, op1)
                self.stack.add_front(product)

        res = self.stack.remove_front()
        return res

    def get_fraction(self, inp):
        """Takes in inp string and returns Fraction object"""
        num, den = inp.split('/')
        return Fraction(int(num), int(den))
```

```
In [29]: pe = EvaluateFraction()
         pe.input("1/2")
         pe.input("2/3")
         pe.input("+")
         assert pe.evaluate()==Fraction(7, 6)

         pe.input("1/2")
         pe.input("2/3")
         pe.input("+")
```

```
        pe.input("1/6")
        pe.input("-")
        assert pe.evaluate()==Fraction(1, 1)

        pe.input("1/2")
        pe.input("2/3")
        pe.input("+")
        pe.input("1/6")
        pe.input("-")
        pe.input("3/4")
        pe.input("*")
        assert pe.evaluate()==Fraction(3, 4)
```

```
peek front 1/2
peek back None
peek front 1/2
peek back 2/3
peek front 1/2
peek back +
1/2
2/3
+
2/3 1/2
peek front 1/2
peek back None
peek front 1/2
peek back 2/3
peek front 1/2
peek back +
peek front 1/2
peek back 1/6
peek front 1/2
peek back -
1/2
2/3
+
2/3 1/2
1/6
-
1/6 7/6
peek front 1/2
peek back None
peek front 1/2
peek back 2/3
peek front 1/2
peek back +
peek front 1/2
peek back 1/6
peek front 1/2
peek back -
peek front 1/2
peek back 3/4
peek front 1/2
peek back *
1/2
2/3
+
2/3 1/2
1/6
-
1/6 7/6
3/4
*
3/4 1/1
```

```
In [30]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

HW3. Modify HW2 so that it can work with MixedFraction. Write a class called `EvaluateMixedFraction` as a subclass of `EvaluateFraction`. You need to override the following methods: - `get_fraction(inp)`: This function should be able to handle string input for MixedFraction such as `1 1/2` or `3/2`. It should return a `MixedFraction` object. - `evaluate()`: This function should return `MixedFraction` object rather than `Fraction` object.

```
In [31]: import operator

         class EvaluateMixedFraction(EvaluateFraction):
             def get_fraction(self, inp):
                 if len(inp.split(' ')) > 1:
                     whole, frac = inp.split(' ')
                     num, den = frac.split('/')
                 else:
                     whole = 0
                     num, den = inp.split('/')
                 return MixedFraction(int(num), int(den), int(whole))

             def evaluate(self):
                 while self.expression.size != 0:
                     item = self.expression.remove_front()
                     print(item)

                     if type(item) == MixedFraction:
                         self.stack.add_front(item)
```

```
                    elif item in self.ops:
                        op1 = self.stack.remove_front()
                        op2 = self.stack.remove_front()
                        print(op1, op2)
                        product = self.ops[item](op2, op1)
                        self.stack.add_front(product)

                res = self.stack.remove_front()
                return res
```

```
In [32]: pe = EvaluateMixedFraction()
         pe.input("3/2")
         pe.input("1 2/3")
         pe.input("+")
         assert pe.evaluate() == MixedFraction(1, 6, 3)

         pe.input("1/2")
         pe.input("2/3")
         pe.input("+")
         pe.input("1 1/8")
         pe.input("-")
         assert pe.evaluate() == MixedFraction(1, 24)

         pe.input("1 1/2")
         pe.input("2 2/3")
         pe.input("+")
         pe.input("1 1/6")
         pe.input("-")
         pe.input("5/4")
         pe.input("*")
         assert pe.evaluate() == MixedFraction( 3, 4, 3)
```

```
peek front 1 1/2
peek back None
peek front 1 1/2
peek back 1 2/3
peek front 1 1/2
peek back +
1 1/2
1 2/3
+
1 2/3 1 1/2
peek front 1/2
peek back None
peek front 1/2
peek back 2/3
peek front 1/2
peek back +
peek front 1/2
peek back 1 1/8
peek front 1/2
peek back -
1/2
2/3
+
2/3 1/2
1 1/8
-
1 1/8 7/6
peek front 1 1/2
peek back None
peek front 1 1/2
peek back 2 2/3
peek front 1 1/2
peek back +
peek front 1 1/2
peek back 1 1/6
peek front 1 1/2
peek back -
peek front 1 1/2
peek back 1 1/4
peek front 1 1/2
peek back *
1 1/2
2 2/3
+
2 2/3 1 1/2
1 1/6
-
1 1/6 25/6
1 1/4
*
1 1/4 3/1
```

```
In [33]: ###
         ### YOUR CODE HERE
         ###
```

HW4. *Linked List:* We are going to implement Linked List Abstract Data Type. To do so, we will implement two classes: `Node` and `MyLinkedList`. In this part, we will implement the class Node.

The class `Node` has the following attribute and computed property: - `element`: which stores the value of the item in that node. - `next`: which stores the reference to the next `Node` in the list. The setter method should check if the value assigned is of type `Node`.

```
In [34]: class Node:
        def __init__(self, e):
            self.element = e
            self.__next = None

        @property
        def next(self):
            return self.__next

        @next.setter
        def next(self, value):
            if isinstance(value, Node):
                self.__next = value
```

HW5. This is a continuation to implement a Linked List. The class `MyLinkedList` has two different properties: - `head`: which points to the `Node` of the first element. - `tail`: which points to the `Node` of the last element.

It should also have the following methods: - `__init__(items)`: which create the link list object based using the arguments. - `get(index)`: which returns the item at the given `index`. - `add_first(item)`: which adds the `item` as the first element. - `add_last(item)`: which adds the `item` as the last element. - `add_at(index, item)`: which adds the `item` at the position `index`. - `remove_first(item)`: which removes the `item` as the first element. - `remove_last(item)`: which removes the `item` as the last element. - `remove_at(index, item)`: which removes the `item` at the position `index`.

```
In [35]: # copy your solution for MyAbstractList from the Cohort problems

         import collections.abc as c

         class MyAbstractList(c.Iterator):
             size = 0
             _idx = 0

             def __init__(self,list_items):
                 for item in list_items:
                     print(item)
                     self.append(item)

             @property
             def is_empty(self):
                 return self.size == 0

             def append(self, item):
                 self.add_at(self.size, item)

             def remove(self, item):
                 if item not in self:
                     return None
                 self.remove_at(self.index_of(item))
```

```
        def __getitem__(self, index):
            return self.get(index)

        def __setitem__(self, index, value):
            self.set_at(index, value)

        def __delitem__(self, index):
            self.remove_at(index)

        def __len__(self):
            return self.size

        def __iter__(self):
            self._idx = 0
            return self

        def __next__(self):
            if self._idx < self.size:
                n_item = self.get(self._idx)
                self._idx += 1
                return n_item
            else:
                raise StopIteration
```

```
In [36]: class MyLinkedList(MyAbstractList):
             def __init__(self, items):
                 self.head = None
                 self.tail = None
                 super().__init__(items)

             def get(self, index):
                 print(index)

                 counter = 0
                 curr = self.head
                 while counter <= index:
                     if counter == index:
                         print(curr.next.element)

                         return curr.element
                     counter += 1
                     curr = curr.next

             def add_first(self, element):
                 node = Node(element)
                 self.head = node
```

```
        if not self.tail:
            self.tail = node

        self.size += 1

    def add_last(self, element):
        node = Node(element)
        self.size += 1

        if not self.tail:
            self.head = node
            self.tail = node

        else:
            prev_tail = self.tail
            self.tail = node
            prev_tail.next = node

    def add_at(self, index, element):
        if index == 0:
            self.add_first(element)

        elif index >= self.size:
            self.add_last(element)
        else:
            counter = 0
            curr = self.head
            while curr.next is not None and counter <= index:
                if counter == index:
                    node = Node(element)
                    curr_next = curr.next
                    curr.next = node
                    node.next = curr_next
                    self.size += 1
                else:
                    counter += 1
                    curr = curr.next

    def set_at(self, index, element):
        if 0 <= index < self.size:
            current = self.head
            for idx in range(0, index):
                current = current.next
            current.element = element

    def remove_first(self):
        if self.size == 0:
            # if list is empty, return None
            return None
        else:
            # otherwise, do the following:
            # 1. store the head at a temporary variable
            # 2. set the next reference of the current head to be the head
            # 3. reduce size by 1
            # 4. if the new head is now None, it means empty list
            #    -> set the tail to be None also
            # 5. return element of the removed node
            prev_head = self.head
            self.head = prev_head.next
            self.size -= 1
            if not self.head:
                self.tail = None
            return prev_head

    def remove_last(self):
        if self.size == 0:
            # if the list is empty, return None
            return None
        elif self.size == 1:
            prev_tail = self.tail
            self.tail = None
            self.head = None
            self.size -= 1
            return prev_tail
        else:
            curr = self.head
            while curr.next is not None:
                prev = curr
                curr = curr.next
            prev_tail = curr
            self.tail = prev
            prev.next = None
            self.size -= 1
            return prev_tail
            # otherwise, do the following:
            # 1. traverse to the second last node
            # 2. store the tail of the list to a temporary variable
            # 3. set the current node as the tail
            # 4. set the next ref of the tail to be None
            # 5. reduce the size by 1
            # 6. return the element of the removed node in the temp var

    def remove_at(self, index):
        if index < 0 or index >= self.size:
            return None
        elif index == 0:
            return self.remove_first()
        elif index == self.size - 1:
            return self.remove_last()
        else:
            counter = 0
            curr = self.head
            while curr.next is not None and counter <= index:
                if counter == index:
                    prev.next = curr.next
                    curr.next = None
                    self.size -= 1

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

```
In [37]: asean = MyLinkedList(['Singapore', 'Malaysia'])
         assert asean.head.element == 'Singapore'
         assert asean.tail.element == 'Malaysia'

         asean.append('Indonesia')
         assert asean.tail.element == 'Indonesia'
         asean.add_at(0, 'Brunei')
         assert asean.head.element == 'Brunei'
         assert asean.size == 4
         assert len(asean) == 4

         asean[0] = 'Cambodia'
         assert asean[0] == 'Cambodia' and asean[1] == 'Singapore'
         asean[2] = 'Myanmar'
         assert(len(asean)) == 4
         assert [x for x in asean] == ['Cambodia', 'Singapore', 'Myanmar', 'Indonesia']

         del asean[0]
         assert [x for x in asean] == ['Singapore', 'Myanmar', 'Indonesia']

         asean.add_at(2, 'Brunei')
         assert [x for x in asean] == ['Singapore', 'Myanmar', 'Brunei', 'Indonesia']
         del asean[3]
         assert [x for x in asean] == ['Singapore', 'Myanmar', 'Brunei']
         del asean[1]
         assert [x for x in asean] == ['Singapore', 'Brunei']
         del asean[1]
         assert [x for x in asean] == ['Singapore']
         del asean[0]
         assert [x for x in asean] == []
```

```
Singapore
```

```
Malaysia
```

```
        ---------------------------------------------------------------------------

        AssertionError                            Traceback (most recent call last)

        <ipython-input-37-a077f4d07f1d> in <module>
          7 asean.add_at(0, 'Brunei')
          8 assert asean.head.element == 'Brunei'
    ----> 9 assert asean.size == 4
         10 assert len(asean) == 4
         11


        AssertionError:
```

```
In [ ]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

Week06_Cohort

October 25, 2021

1 Week 6 Problem Set

1.1 Cohort Sessions

CS1. Create a class called `Fraction` to represent a simple fraction. The class has two properties: - `num`: which represents a numerator and of the type Integer. - `den`: which represents a denominator and of the type Integer. Denominator should not be a zero. If a zero is assigned, you need to replace it with a `1`.

The class should have the following method: - `__init__(num, den)`: to initialize the numerator and the denominator. You should check if the denominator is zero. If it is you should assign `1` as the denominator instead. - `__str__()`: for the object instance to be convertable to String. You need to return a string in a format of `num/den`

```
In [2]: class Fraction:
            def __init__(self, num, den):
                self.num = num
                self.den = den

            @property
            def num(self):
                return self._numattr

            @num.setter
            def num(self, value):
                self._numattr = int(value)

            @property
            def den(self):
                return self._denattr

            @den.setter
            def den(self, value):
                # ask about property and how it influences the attribute
                if value == 0:
                    self._denattr = 1
                else:
                    self._denattr = int(value)
            def __str__(self):
                return f"{self.num}/{self.den}"
```

```
In [3]: f0 = Fraction(0, 1)
        assert f0.num == 0
        assert f0.den == 1
        assert str(f0) == "0/1"

        f1 = Fraction(1, 2)
        assert f1.num == 1
        assert f1.den == 2
        assert str(f1) == "1/2"

        f1.num = 3
        f1.den = 4
        assert str(f1) == "3/4"
```

```
In [4]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

CS2. Extend the class `Fraction` to support the following operator: `+` and `==`. To do this, you need to overload the following operator: - `__add__(self, other)` - `__eq__(self, other)`

You may want to write a method to simplify a fraction: - `simplify()`: which simplify a fraction to its lowest terms. To simplify a fraction divide both the numerator and the denominator with the greatest common divisor of the the two. This method should return a new `Fraction` object.

```
In [5]: class Fraction:
            def __init__(self, num, den):
                self.num = num
                self.den = den

            @property
            def num(self):
                return self._numattr

            @num.setter
            def num(self, value):
                self._numattr = int(value)

            @property
            def den(self):
                return self._denattr

            @den.setter
            def den(self, value):
                # ask about property and how it influences the attribute
                if value == 0:
                    self._denattr = 1
                else:
                    self._denattr = int(value)

            def __str__(self):
                return f"{self.num}/{self.den}"

            def __add__(self, other):
                num = self.num * other.den + self.den * other.num
                den = self.den * other.den
                return Fraction(num, den).simplify()

            def simplify(self):
                common = gcd(self.num, self.den)
                num = self.num // common
                den = self.den // common
                return Fraction(num, den)

            def __eq__(self, other):
                left = self.simplify()
                right = other.simplify()
                return left.num == right.num and left.den == right.den


        # def gcd(a, b):
        #     """Returns Greatest Common Divisor of a and b"""
        #     while b:
        #         a, b = b, a % b
        #     return a

        def gcd(a, b):
            if b == 0:
                return a
            else:
                return gcd(b, a % b)
```

```
In [6]: f1 = Fraction(1, 2)
        f2 = Fraction(2, 3)
        f3 = f1 + f2

        assert str(f3) == "7/6"

        f4 = Fraction(3, 5)
        f5 = Fraction(1, 3)
        f6 = f4 + f5

        assert str(f6) == "14/15"

        f1 = Fraction(1, 2)
        f2 = Fraction(2, 4)
        assert f1 == f2

        f3 = Fraction(2, 3)
        f4 = Fraction(2, 4)
        assert f3 != f4
```

```
In [7]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

CS3. *Inheritance:* Create a class called `MixedFraction` as a subclass of `Fraction`. A mixed fraction is a fraction that comprises of a whole number, a numerator and a denominator, e.g. `1 2/3` which is the same as `5/3`. The class has the following way of initializing its properties: - `__init__(top, bot, whole)`: which takes in three Integers, the whole number, the numerator, and the denominator, e.g. `whole=1`, `top=2`, `bot=3`. The argument `whole` by default is `0`. You can also specify `top` to be greater than `bot`.

The class only has two properties: - `num`: which is the numerator and can be greater than denominator. - `den`: which is the denominator and must be a non-zero number.

The class should also have the following methods: - `get_three_numbers()`: which is used to calculate the whole number, numerator and the denominator from a given numerator and denominator. The stored properties are `num` and `den` as in `Fraction` class. This function returns three Integers as a tuple, i.e. `(top, bot, whole)`.

The class should also override the `__str__()` method in this manner: - `num/dem` if the numerator is smaller than the denominator. For example, `2/3`. - `whole top/bot` if the numerator is greater than the denominator. For example, `1 2/3`.

```
In [8]: class MixedFraction(Fraction):
            def __init__(self, top, bot, whole=0):
                num = top + whole * bot
                self.whole = whole
                super().__init__(num, bot)

            @property
            def whole(self):
                return self._whole

            @whole.setter
            def whole(self, value):
                if isinstance(value,int):
                    self._whole = value

            def get_three_numbers(self):
                if self.num > self.den:
                    return self.num % self.den, self.den, self.num // self.den
                else:
                    return self.num, self.den, self.whole
```

Week05_Homework

October 25, 2021

1 Week 5 Problem Set

1.1 Homework

HW1. *Dictionary:* Write two functions: 1. `count_degrees(G)`: which sums up the degrees of all vertices in the graph. The degree of a vertex is defined as the number of edges connected to a Vertex. The argument `G` is a dictionary that represents the graph. 2. `count_edges(G)`: which counts the number of edges in the graph. An edge is defined as a connection between two vertices. The argument `G` is a dictionary.

```
In [128]: def count_degrees(G):
              degrees = 0
              for _,edges in G.items():
                  degrees += len(edges)
              return degrees

          def count_edges(G):
              num_edges = 0
              for _, edges in G.items():
                  num_edges += len(edges)
              return num_edges // 2
```

```
In [129]: g1 = {"A": ["B", "E"],
                "B": ["A", "C"],
                "C": ["B", "D", "E"],
                "D": ["C"],
                "E": ["A", "C"]}

          d = count_degrees(g1)
          e = count_edges(g1)

          assert d == 10
          assert e == 5
```

```
In [130]: ###
          ### AUTOGRADER TEST - DO NOT REMOVE
          ###
```

HW2. Create a class called `GraphSearch` which is a subclass of the class `Graph`. This class should override the method `_create_vertex(id)` and instantiate a `VertexSearch` object instead of `Vertex`.

```
In [63]: class Vertex:
             def __init__(self, id=""):
                 self.id = id
                 self.neighbours = {}

             def add_neighbour(self, nbr_vertex, weight=0):
                 self.neighbours[nbr_vertex] = weight

             def get_neighbours(self):
                 #  returns all the Vertices connected to the current Vertex as a list.
                 # The elements of the output list are of Vertex object instances.
                 return list(self.neighbours.keys())

             def get_weight(self, neighbour):
                 return self.neighbours.get(neighbour)

             def __lt__(self, other):
                 # return True if self.id < other.id else False
                 return self.id < other.id

             def __hash__(self):
                 # which calls the hash() function on id and returns it.
                 # This allows the object to be a dictionary key.
                 return hash(self.id)

In [64]: import sys

         class VertexSearch(Vertex):
             def __init__(self, id=""):
                 super().__init__()
                 self.id = id
                 self.colour = "white"
                 self.d = sys.maxsize
                 self.f = sys.maxsize
                 self.parent = None

In [65]: class Graph:
             def __init__(self):
                 self.vertices = {}

             def _create_vertex(self, id):
                 return Vertex(id)

             def add_vertex(self, id):
```

```
            vertex = self._create_vertex(id)
            self.vertices[id] = vertex

        def get_vertex(self, id):
            return self.vertices.get(id)

        def add_edge(self, start_v, end_v, weight=0):
            self.vertices[start_v].add_neighbour(end_v, weight)

        def get_neighbours(self, id):
            vertex = self.vertices.get(id)
            if not vertex:
                return None
            return vertex.get_neighbours()

        def __contains__(self, id):
            return id in self.vertices.keys()

        def __iter__(self):
            for k, v in self.vertices.items():
                yield v

        @property
        def num_vertices(self):
            return len(self.vertices)
```

```
In [66]: class GraphSearch(Graph):
             ##BEGIN SOLUTION
             def _create_vertex(self, id):
                 return VertexSearch(id)
```

```
In [67]: g2 = GraphSearch()
         g2.add_vertex("Z")
         assert(type(g2.vertices["Z"]) == type(VertexSearch()))
```

```
In [68]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

HW3. *Undirected Graph:* **You need to complete CS5 and CS6 before you do this homework.** Paste your answer for `Search2D` and `SearchBFS` classes into the cell below.

Create a subclass of `GraphSearch` class called `UGraphSearch` for undirected graphs. All edges in undirected graphs are bidirectional (i.e. vertex1 <-> vertex2). This means that you need to override the `add_edge()` method.

```
In [69]: class Queue:
             def __init__(self):
                 self.__items = []
```

```
        def enqueue(self, item):
            self.__items.append(item)

        def dequeue(self):
            return self.__items.pop(0) if not self.is_empty else None

        def peek(self):
            return self.__items[0]

        @property
        def is_empty(self):
            return len(self.__items) == 0

        @property
        def size(self):
            return len(self.__items)
```

```
In [70]: # copy Search2D over from CS5

         import sys

         class Search2D:
             def __init__(self, g):
                 self.graph = g

             def clear_vertices(self):
                 for vertex in self.graph:
                     vertex.colour = "white"
                     vertex.d = sys.maxsize
                     vertex.f = sys.maxsize
                     vertex.parent = None

             def __iter__(self):
                 return iter([v for v in self.graph])

             def __len__(self):
                 return len([v for v in self.graph.vertices])
```

```
In [71]: class SearchBFS(Search2D):
             def __init__(self, g):
                 super().__init__(g)
                 # self.graph = g

             def search_from(self, start):
                 queue = Queue()

                 start_vertex = self.graph.get_vertex(start)
```

```
        start_vertex.color = "grey"
        start_vertex.d = 0
        start_vertex.parent = None

        queue.enqueue(start)
        while not queue.is_empty:
            vertex_id = queue.dequeue()

            vertex = self.graph.get_vertex(vertex_id)
            vertext_nbrs = self.graph.get_neighbours(vertex_id)
            for nbr_id in vertext_nbrs:
                nbr = self.graph.get_vertex(nbr_id)
                if nbr.colour == "white":
                    nbr.colour = "grey"
                    nbr.d = vertex.d + 1
                    nbr.parent = vertex
                    queue.enqueue(nbr.id)
            vertex.colour = "black"

    def get_shortest_path(self, start, dest):
        if not self.graph.get_vertex(start) or not self.graph.get_vertex(dest):
            return
        if self.graph.get_vertex(start) == self.graph.get_vertex(dest):
            return [start]
        if self.graph.get_vertex(start).d != 0:
            self.clear_vertices()
            self.search_from(start)

        result = []
        self.get_path(start, dest, result)

        return result

    def get_path(self, start, dest, result):
        start_vert = self.graph.get_vertex(start)
        dest_vert = self.graph.get_vertex(dest)
        if not self.graph.get_vertex(dest).parent and start_vert.id != dest_vert.id:
            result += ["No Path"]
            return

        print(f"start : {start}, dest : {dest}")
        result.insert(0, dest)

        print(f"dest : {dest_vert.id}")

        if start_vert.id == dest_vert.id:
            return True
```

```
                self.get_path(start, self.graph.get_vertex(dest).parent.id, result)
```

In [72]:
```
class UGraphSearch(GraphSearch):
    def add_edge(self, start_v, end_v, weight=0):
        self.vertices[start_v].add_neighbour(end_v, weight)
        self.vertices[end_v].add_neighbour(start_v, weight)
```

In [73]:
```
g2 = UGraphSearch()
assert g2.vertices == {} and g2.num_vertices == 0
g2.add_vertex("L")
g2.add_vertex("I")
g2.add_vertex("S")
g2.add_vertex("P")
assert g2.num_vertices == 4
assert "L" in g2
assert "I" in g2
assert "S" in g2
assert "P" in g2
g2.add_edge("L", "I")
g2.add_edge("I", "S")
g2.add_edge("S", "P")
assert sorted(g2.get_neighbours("L")) == ["I"]
assert sorted(g2.get_neighbours("I")) == ["L", "S"]
assert sorted(g2.get_neighbours("S")) == ["I", "P"]
assert sorted(g2.get_neighbours("P")) == ["S"]
```

In [74]:
```
###
### AUTOGRADER TEST - DO NOT REMOVE
###
```

In [75]:
```
g2 = UGraphSearch()
g2.add_vertex("r")
g2.add_vertex("s")
g2.add_vertex("t")
g2.add_vertex("u")
g2.add_vertex("v")
g2.add_vertex("w")
g2.add_vertex("x")
g2.add_vertex("y")
g2.add_vertex("z")
g2.add_edge("r", "s")
g2.add_edge("r", "v")
g2.add_edge("s", "w")
g2.add_edge("t", "u")
g2.add_edge("t", "x")
g2.add_edge("t", "w")
g2.add_edge("u", "t")
g2.add_edge("u", "x")
g2.add_edge("u", "y")
```

```
        g2.add_edge("v", "r")
        g2.add_edge("w", "s")
        g2.add_edge("w", "t")
        g2.add_edge("w", "x")
        g2.add_edge("x", "w")
        g2.add_edge("x", "t")
        g2.add_edge("x", "u")
        g2.add_edge("x", "y")
        g2.add_edge("y", "u")
        g2.add_edge("y", "x")
        gs = SearchBFS(g2)
        gs.search_from("s")
        assert gs.graph.get_vertex("s").d == 0
        assert gs.graph.get_vertex("t").d == 2
        assert gs.graph.get_vertex("u").d == 3
        assert gs.graph.get_vertex("v").d == 2
        assert gs.graph.get_vertex("w").d == 1
        assert gs.graph.get_vertex("x").d == 2
        assert gs.graph.get_vertex("y").d == 3
        assert gs.graph.get_vertex("r").d == 1
        ans = gs.get_shortest_path("s", "u")
        assert ans == ["s", "w", "t", "u"] or ans == ["s", "w", "x", "u"]
        ans = gs.get_shortest_path("v", "s")
        assert ans == ["v", "r", "s"]
        ans = gs.get_shortest_path("v", "y")
        assert ans == ["v", "r", "s", "w", "x", "y"]
```

```
start : s, dest : u
dest : u
start : s, dest : t
dest : t
start : s, dest : w
dest : w
start : s, dest : s
dest : s
start : v, dest : s
dest : s
start : v, dest : r
dest : r
start : v, dest : v
dest : v
start : v, dest : y
dest : y
start : v, dest : x
dest : x
start : v, dest : w
dest : w
start : v, dest : s
```

```
dest : s
start : v, dest : r
dest : r
start : v, dest : v
dest : v
```

```
In [76]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

HW4. *Depth-First-Search:* Create a class `SearchDFS` as a child class of `Search2D` to implement Depth-First-Search algorithm. You should add one additional attribute: - `time`: which is an attribute to record the discovery time and the finishing time.

The class should also implement the following methods: - `search()`: which modifies the vertices' properties such as `colour`, `d`, and `parent` following Depth-First-Search algorithm. - `dfs_visit(vert)`: which is the recursive method for visiting a vertex in Depth-First-Search algorithm.

```
In [104]: import sys

          class SearchDFS(Search2D):
              def __init__(self, g):
                  super().__init__(g)
                  self.graph = g
                  self.time = 0
                  self.discovery_time = 0
                  self.finishing_time = 0


              def search(self):
                  for vertex in self.graph:
                      vertex.colour = "white"
                      self.parent = None
                      vertex.d = 0
                  self.time = 0
                  for vertex in self.graph:
                      if vertex.colour == "white":
                          self.dfs_visit(vertex.id)

              def dfs_visit(self, vert_id):
                  # which is the recursive method for visiting a vertex in Depth-First-Search
                  if not vert_id or not self.graph.get_vertex(vert_id):
                      return

                  # visit the node
                  vertex = self.graph.get_vertex(vert_id)
                  vertex.colour = "grey"
```

```
                self.time += 1
                vertex.d = self.time

                # visit neighbours
                vertext_nbrs = self.graph.get_neighbours(vert_id)
                for nbr_id in vertext_nbrs:
                    nbr = self.graph.get_vertex(nbr_id)
                    if nbr.colour == "white":
                        nbr.parent = vertex  # why do we want to set parent as vertex
                        nbr.d = vertex.d + 1
                        self.dfs_visit(nbr_id)

                vertex.colour = "black"
                self.time += 1
                vertex.f = self.time
```

```
In [105]: g4 = GraphSearch()
          g4.add_vertex("e")
          g4.add_vertex("m")
          g4.add_vertex("a")
          g4.add_vertex("c")
          g4.add_vertex("s")
          g4.add_edge("e", "m")
          g4.add_edge("m", "a")
          g4.add_edge("a", "c")
          g4.add_edge("c", "s")
          gs = SearchDFS(g4)
          gs.search()
          assert gs.graph.get_vertex("e").parent == None
          print(gs.graph.get_vertex("m").parent)
          print(gs.graph.get_vertex("e"))
          print(gs.graph.get_vertex("m").d)
          print(gs.graph.get_vertex("a").f)
          assert gs.graph.get_vertex("m").parent == gs.graph.get_vertex("e")

          assert gs.graph.get_vertex("m").d == 2 and gs.graph.get_vertex("a").f == 8
          assert gs.graph.get_vertex("c").d == 4 and gs.graph.get_vertex("s").f == 6
```

```
<__main__.VertexSearch object at 0x7f8bf455b510>
<__main__.VertexSearch object at 0x7f8bf455b510>
2
8
```

```
In [106]: g4 = GraphSearch()
          g4.add_vertex("e")
          g4.add_vertex("m")
```

```
        g4.add_vertex("a")
        g4.add_vertex("c")
        g4.add_vertex("s")
        g4.add_edge("e", "m")
        g4.add_edge("m", "a")
        g4.add_edge("a", "c")
        g4.add_edge("c", "s")
        gs = SearchDFS(g4)
        gs.search()
        assert gs.graph.get_vertex("e").parent == None
        assert gs.graph.get_vertex("m").parent == gs.graph.get_vertex("e")

        assert gs.graph.get_vertex("m").d == 2 and gs.graph.get_vertex("a").f == 8
        assert gs.graph.get_vertex("c").d == 4 and gs.graph.get_vertex("s").f == 6
```

```
In [ ]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

HW5. *Topological Sort:* Create a function that takes in a `SearchDFS` object to perform a topological sort: - `topological_sort(g)`: which takes in a `SearchDFS` object and returns a list of `VertexSearch` objects sorted based on their `f` property. This method should call the `search()` method of the `SearchDFS` to first calculate the `f` property of the vertices. Note that you need to copy the `SearchDFS` object of your input argument so as not to mutate the object.

```
In [1]: import sys
        import copy

        def topological_sort(g):
            res = []
            g.search()
            for vertex in g:
                res.append((vertex,vertex.f))
            return [i[0] for i in sorted(res,reverse = True,key= lambda x : x[1])]
```

```
In [126]: import copy
          g = GraphSearch()
          g.add_vertex("3/4 cup milk")
          g.add_vertex("1 egg")
          g.add_vertex("1 tbl oil")
          g.add_vertex("1 cup mix")
          g.add_vertex("heat syrup")
          g.add_vertex("heat griddle")
          g.add_vertex("pour 1/4 cup")
          g.add_vertex("turn when bubbly")
          g.add_vertex("eat")
          g.add_edge("3/4 cup milk", "1 cup mix")
          g.add_edge("1 egg", "1 cup mix")
          g.add_edge("1 tbl oil", "1 cup mix")
```

```
        g.add_edge("1 cup mix", "heat syrup")
        g.add_edge("1 cup mix", "pour 1/4 cup")
        g.add_edge("heat griddle", "pour 1/4 cup")
        g.add_edge("pour 1/4 cup", "turn when bubbly")
        g.add_edge("turn when bubbly", "eat")
        g.add_edge("heat syrup", "eat")
        gs = SearchDFS(g)

        path = topological_sort(gs)
        ans = [v.f for v in copy.copy(path)]
        print(ans)
        assert ans == [18, 16, 14, 12, 11, 10, 9, 6, 5]
        ans = [v.id for v in copy.copy(path)]
        assert ans == ['heat griddle', '1 tbl oil', '1 egg', '3/4 cup milk', '1 cup mix', 'po
```

```
[18, 16, 14, 12, 11, 10, 9, 6, 5]
```

```
In [127]: import copy
          g = GraphSearch()
          g.add_vertex("3/4 cup milk")
          g.add_vertex("1 egg")
          g.add_vertex("1 tbl oil")
          g.add_vertex("1 cup mix")
          g.add_vertex("heat syrup")
          g.add_vertex("heat griddle")
          g.add_vertex("pour 1/4 cup")
          g.add_vertex("turn when bubbly")
          g.add_vertex("eat")
          g.add_edge("3/4 cup milk", "1 cup mix")
          g.add_edge("1 egg", "1 cup mix")
          g.add_edge("1 tbl oil", "1 cup mix")
          g.add_edge("1 cup mix", "heat syrup")
          g.add_edge("1 cup mix", "pour 1/4 cup")
          g.add_edge("heat griddle", "pour 1/4 cup")
          g.add_edge("pour 1/4 cup", "turn when bubbly")
          g.add_edge("turn when bubbly", "eat")
          g.add_edge("heat syrup", "eat")
          gs = SearchDFS(g)

          path = topological_sort(gs)
          ans = [v.f for v in copy.copy(path)]
          assert ans == [18, 16, 14, 12, 11, 10, 9, 6, 5]
          ans = [v.id for v in copy.copy(path)]
          assert ans == ['heat griddle', '1 tbl oil', '1 egg', '3/4 cup milk', '1 cup mix', 'po
```

```
In [ ]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

```
In [ ]:
```

```
In [ ]:
```

Week05_Cohort

October 25, 2021

1 Week 5 Problem Set

1.1 Cohort Sessions

CS1. *Dictionary:* Implement a Graph using a *Dictionary* where the keys are the Vertices in the Graph and the values (in the the key-value pair) correspond to an Array containing the neighbouring Vertices. For example, let's represent the following graph:

```
A -> B
A -> C
B -> C
B -> D
C -> D
D -> C
E -> F
F -> C
```

Create a Dictionary to represent the graph above.

```
In [1]: graph = {
            'A' : ['B','C'],
            'B' : ['C', 'D'],
            'C' : ['D'],
            'D' : ['C'],
            'E' : ['F'],
            'F' : ['C']
        }
```

```
In [2]: print(graph)
```

```
{'A': ['B', 'C'], 'B': ['C', 'D'], 'C': ['D'], 'D': ['C'], 'E': ['F'], 'F': ['C']}
```

Write a function `get_neighbours(graph, vert)` which returns a list of all neighbours of the requested Vertex `vert` in the `graph`. Return `None` if the Vertex is not in the graph.

```
In [3]: def get_neighbours(graph, vert):
            return graph.get(vert)
```

```
In [4]: assert get_neighbours(graph, "B") == ["C", "D"]
        assert get_neighbours(graph, "A") == ["B", "C"]
        assert get_neighbours(graph, "F") == ["C"]
```

```
In [5]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

Write a function `get_source(graph, vert)` which returns a list of all source Vertices pointing to `vert` in the `graph`. For example, Vertex "C" has the following source Vertices: '["A", "B", "D", "F"]. Return an empty list if there are none.

```
In [6]: def get_source(graph, vert):
            source = []
            for node,edges in graph.items():
                if vert in edges:
                    source.append(node)
            return source
```

```
In [7]: assert sorted(get_source(graph, "C")) == ["A", "B", "D", "F"]
        assert sorted(get_source(graph, "D")) == ["B", "C"]
        assert sorted(get_source(graph, "F")) == ["E"]
```

```
In [8]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

CS2. Create a class `Vertex` to represent a vertex in a graph. The class `Vertex` has the following attributes: - `id`: to identify each vertex. This is of String data type. - `neighbours`: which is a Dictionary where the keys are the neighbouring `Vertex` object instances that are connected to the current Vertex and the values are the weights of the edge between the current Vertex and the neighbouring vertices.

The class should also have the following methods:

- `__init__(self, id)`: which is used to initialized the attribute `id`. By default, `id` is set to an empty String . The attribute `neighbours` is always set to an empty dictionary.
- `add_neighbour(self, nbr_vertex, weight)`: which adds a neighbouring Vertex to the current Vertex. The second argument provides the weight of the edge between the current Vertex and the newly added neighbouring Vertex. By default, `weight` is `0`.
- `get_neigbours(self)`: which returns all the Vertices connected to the current Vertex as a list. The elements of the output list are of `Vertex` object instances.
- `get_weight(self, neighbour)`: which returns the weight of the requested neighbour. It should return `None` if the requested neighbour is not found.
- `__eq__(self, other)`: which returns true if the id of the current vertex object is the same as the `other` vertex's id.
- `__lt__(self, other)`: which returns true if the id of the current vertex object is less than the `other` vertex's id.
- `__hash__(self)`: which calls the `hash()` function on `id` and returns it. This allows the object to be a dictionary key.

- `__str__(self)`: This method should return the id of the current vertex and a list of ids of the neighbouring vertices, like `Vertex 2 is connected to: 3, 4, 5.`

```
In [9]: class Vertex:
            def __init__(self, id=""):
                self.id = id
                self.neighbours = {}

            def add_neighbour(self, nbr_vertex, weight=0):
                self.neighbours[nbr_vertex] = weight

            def get_neighbours(self):
                #  returns all the Vertices connected to the current Vertex as a list.
                # The elements of the output list are of Vertex object instances.
                print(self.neighbours.keys())
                return list(self.neighbours.keys())

            def get_weight(self, neighbour):
                return self.neighbours.get(neighbour)

            def __eq__(self, other):
                return self.id == other.id

            def __lt__(self, other):
                # return True if self.id < other.id else False
                return self.id < other.id

            def __hash__(self):
                # which calls the hash() function on id and returns it.
                # This allows the object to be a dictionary key.
                return hash(self.id)

            def __str__(self):
                neighbour_id = [nbr.id for nbr in self.get_neighbours()]
                nbr_string = ", ".join(neighbour_id)

                return f"Vertex {self.id} is connected to: {nbr_string}"

In [10]: v1 = Vertex("1")
         assert v1.id == "1" and len(v1.neighbours) == 0
         v2 = Vertex("2")
         v1.add_neighbour(v2)
         assert v1.get_neighbours()[0].id == "2" and v1.neighbours[v1.get_neighbours()[0]] ==
         v3 = Vertex("3")
         v1.add_neighbour(v3, 3)
         assert v1.get_weight(v3) == 3
         v4 = Vertex("4")
         assert v1.get_weight(v4) == None
```

```
        assert v1 < v2
        assert v1 != v2
        assert str(v1) == "Vertex 1 is connected to: 2, 3"
```

```
dict_keys([<__main__.Vertex object at 0x7fa7340b5d50>])
dict_keys([<__main__.Vertex object at 0x7fa7340b5d50>])
dict_keys([<__main__.Vertex object at 0x7fa7340b5d50>, <__main__.Vertex object at 0x7fa7340b5c
```

```
In [11]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

CS3. Create a class `Graph` to represent a Graph. The class has the following attributes: - `vertices`: which is a *dictionary* of Vertices. The keys are the `id`s of the Vertices and the values are `Vertex` object instances.

The class has the follo - `num_vertices`: which is a *computed* property that returns the number of vertices in the graph.

The class also has the following methods: - `__init__(self)`: which initializes the graph with an empty dictionary. - `_create_vertex(self, id)`: which creates a new `Vertex` object with a given `id`. This method is never called directly and is only used by `add_vertex(id)`. - `add_vertex(self, id)`: which creates a new `Vertex` object, adding it into the dictionary `vertices`. The argument `id` is a String. This method should call `_create_vertex(id)`. - `get_vertex(self, id)`: which returns the `Vertex` object instance of the requested `id`. The method should return `None` if the requested `id` cannot be found. The argument `id` is a String. - `add_edge(start_v, end_v)`: which creates an edge from one Vertex to another Vertex. The arguments are the `id`s of the two vertices and are both Strings. - `get_neighbours(self, id)`: which returns a list of `id`s all the neighbouring vertices (of the specified Vertex `id`). It should return `None` if `id` cannot be found. The argument `id` is a String and the elements of the output list are of `str` data type. - `__contains__(self, id)`: which returns either `True` or `False` depending on whether the graph contains the specified Vertex's `id`. The argument `id` is a String.

```
In [12]: class Graph:
             def __init__(self):
                 self.vertices = {}

             def _create_vertex(self, id):
                 return Vertex(id)

             def add_vertex(self, id):
                 vertex = self._create_vertex(id)
                 self.vertices[id] = vertex

             def get_vertex(self, id):
                 return self.vertices.get(id)

             def add_edge(self, start_v, end_v, weight=0):
                 self.vertices[start_v].add_neighbour(end_v, weight)
```

```
        def get_neighbours(self, id):
            vertex = self.get_vertex(id)
            if not vertex:
                return None
            return vertex.get_neighbours()

        def __contains__(self, id):
            return id in self.vertices.keys()

        def __iter__(self):
            for k, v in self.vertices.items():
                yield v

        @property
        def num_vertices(self):
            return len(self.vertices)
```

```
In [13]: g = Graph()
         assert g.vertices == {} and g.num_vertices == 0
         g.add_vertex("A")
         g.add_vertex("B")
         g.add_vertex("C")
         g.add_vertex("D")
         g.add_vertex("E")
         g.add_vertex("F")
         assert g.num_vertices == 6
         assert "A" in g
         assert "B" in g
         assert "C" in g
         assert "D" in g
         assert "E" in g
         assert "F" in g
         g.add_edge("A", "B")
         g.add_edge("A", "C")
         g.add_edge("B", "C")
         g.add_edge("B", "D")
         g.add_edge("C", "D")
         g.add_edge("D", "C")
         g.add_edge("E", "F")
         g.add_edge("F", "C")
         assert sorted(g.get_neighbours("A")) == ["B", "C"]
         assert sorted(g.get_neighbours("B")) == ["C", "D"]
         assert sorted(g.get_neighbours("C")) == ["D"]
         assert [v.id for v in g] == ["A", "B", "C", "D", "E", "F"]
```

```
dict_keys(['B', 'C'])
dict_keys(['C', 'D'])
dict_keys(['D'])
```

```
In [14]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

CS4. Create a subclass of `Vertex` called `VertexSearch`. This class has the following additional attributes: - `colour`: which is a mark on the vertex during the search algorithm. It is of String data type and should be set to "white" by default. - `d`: which is an Integer denoting the distance from other Vertex to the current Vertex in Breath-First-Search. This is also used to record discovery time in Depth-First-Search. This property should be initialized to `sys.maxsize` at the start. - `f`: which is an Integer denoting the final time in Depth-First-Search. This property should be initialized to `sys.maxsize` at the start. - `parent`: which is a reference to the parent Vertex object. This property should be set to `None` at the start.

```
In [15]: import sys

         class VertexSearch(Vertex):
             def __init__(self, id=""):
                 super().__init__()
                 self.id = id
                 self.colour = "white"
                 self.d = sys.maxsize
                 self.f = sys.maxsize
                 self.parent = None
```

```
In [16]: import sys

         v = VertexSearch()
         assert v.id == ""
         assert v.colour == "white"
         assert v.d == sys.maxsize
         assert v.f == sys.maxsize
         assert v.parent == None
```

```
In [17]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

CS5. You should do this after you completed HW2. Create a class `Search2D` which takes in an object `GraphSearch` for its initialization. The class should have the following methods: - `clear_vertices()`: which sets the properties of all the vertices: - `colour` to "white" - `d` to `sys.maxsize` - `f` to `sys.maxsize` - `parent` to `None`.

```
In [18]: class GraphSearch(Graph):
             def __init__(self):
                 super().__init__()
             ##BEGIN SOLUTION
             def _create_vertex(self, id):
                 return VertexSearch(id)
```

```
In [19]: class Search2D:
             def __init__(self, g):
                 self.graph = g

             def clear_vertices(self):
                 for vertex in self.graph:
                     vertex.colour = "white"
                     vertex.d = sys.maxsize
                     vertex.f = sys.maxsize
                     vertex.parent = None

             def __iter__(self):
                 return iter([v for v in self.graph])

             def __len__(self):
                 return len([v for v in self.graph.vertices])
```

```
In [20]: g4 = GraphSearch()
         g4.add_vertex("A")
         g4.add_vertex("B")
         g4.add_vertex("C")
         g4.add_vertex("D")
         g4.add_vertex("E")
         g4.add_vertex("F")
         g4.add_edge("A", "B")
         g4.add_edge("A", "C")
         g4.add_edge("B", "C")
         g4.add_edge("B", "D")
         g4.add_edge("C", "D")
         g4.add_edge("D", "C")
         g4.add_edge("E", "F")
         g4.add_edge("F", "C")
         gs4 = Search2D(g4)
         gs4.clear_vertices()

         assert len(gs4) == 6
         assert [v.id for v in gs4] == ["A", "B", "C", "D", "E", "F"]
         assert [v.colour for v in gs4] == ["white" for v in range(len(gs4))]
         assert [v.d for v in gs4] == [sys.maxsize for v in range(len(gs4))]
         assert [v.f for v in gs4] == [sys.maxsize for v in range(len(gs4))]
         assert [v.parent for v in gs4] == [None for v in range(len(gs4))]
```

```
In [21]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

CS6. Create a class `SearchBFS` which is a subclass of `Search2D`. This subclass should implement the Breadth First Search algorithm in the following methods:

- `search_from(start)`: which initializes the `d` and `parent` properties of each vertices in the graph from the `start` Vertex following Breadth-First-Search algorithm. Use your previous code that implements `Queue` data structure.
- `get_shortest_path(start, dest)`: which returns a list of vertex ids that forms a shortest path from Vertex `start` to Vertex `dest`. Think how to solve this using recursion. This method should call `search_from()` if the distance at `start` Vertex is not zero. It should return a list containing ["No Path"] if there is no path from the start to the destination vertex. *Hint: you can make use another function for your recursion `get_path(start, dest, result)` where result is an empty list which will be populated in the recursive calls.*

```
In [22]: class Queue:
             def __init__(self):
                 self.__items = []

             def enqueue(self, item):
                 self.__items.append(item)

             def dequeue(self):
                 return self.__items.pop(0) if not self.is_empty else None

             def peek(self):
                 return self.__items[0]

             @property
             def is_empty(self):
                 return len(self.__items) == 0

             @property
             def size(self):
                 return len(self.__items)
```

```
In [1]: class SearchBFS(Search2D):
            def __init__(self, g):
                super().__init__(g)
                # self.graph = g

            def search_from(self, start):
                queue = Queue()

                start_vertex = self.graph.get_vertex(start)
                start_vertex.color = "grey"
                start_vertex.d = 0
                start_vertex.parent = None

                queue.enqueue(self.graph.get_vertex(start))

                while not queue.is_empty:
```

```
            vertex = queue.dequeue()
            vertext_nbrs = self.graph.get_neighbours(vertex_id)

            for nbr_id in vertext_nbrs:
                nbr = self.graph.get_vertex(nbr_id)
                if nbr.colour == "white":
                    nbr.colour = "grey"
                    nbr.d = vertex.d + 1
                    nbr.parent = vertex
                    queue.enqueue(nbr)
            vertex.colour = "black"

    def get_shortest_path(self, start, dest):
        if not self.graph.get_vertex(start) or not self.graph.get_vertex(dest):
            return
        if self.graph.get_vertex(start) == self.graph.get_vertex(dest):
            return [start]
        if self.graph.get_vertex(start).d != 0:
            self.clear_vertices()
            self.search_from(start)

        result = []
        self.get_path(start, dest, result)

        return result

    def get_path(self, start, dest, result):
        start_vert = self.graph.get_vertex(start)
        dest_vert = self.graph.get_vertex(dest)
        if not self.graph.get_vertex(dest).parent and start_vert.id != dest_vert.id:
            result += ["No Path"]
            return

        print(f"start : {start}, dest : {dest}")
        result.insert(0, dest)

        print(f"dest : {dest_vert.id}")

        if start_vert.id == dest_vert.id:
            return True

        self.get_path(start, self.graph.get_vertex(dest).parent.id, result)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)
```

```
<ipython-input-1-450fc2954623> in <module>
----> 1 class SearchBFS(Search2D):
      2     def __init__(self, g):
      3         super().__init__(g)
      4         # self.graph = g
      5


NameError: name 'Search2D' is not defined
```

```
In [2]: g4 = GraphSearch()
        g4.add_vertex("A")
        g4.add_vertex("B")
        g4.add_vertex("C")
        g4.add_vertex("D")
        g4.add_vertex("E")
        g4.add_vertex("F")
        g4.add_edge("A", "B")
        g4.add_edge("A", "C")
        g4.add_edge("B", "C")
        g4.add_edge("B", "D")
        g4.add_edge("C", "D")
        g4.add_edge("D", "C")
        g4.add_edge("E", "F")
        g4.add_edge("F", "C")
        gs4 = SearchBFS(g4)

        gs4.search_from("A")
        assert gs4.graph.get_vertex("A").d == 0
        assert gs4.graph.get_vertex("A").colour == "black"
        assert gs4.graph.get_vertex("A").parent == None
        assert gs4.graph.get_vertex("B").d == 1
        assert gs4.graph.get_vertex("B").colour == "black"
        assert gs4.graph.get_vertex("B").parent == gs4.graph.get_vertex("A")
        assert gs4.graph.get_vertex("C").d == 1
        assert gs4.graph.get_vertex("C").colour == "black"
        assert gs4.graph.get_vertex("C").parent == gs4.graph.get_vertex("A")
        assert gs4.graph.get_vertex("D").d == 2
        assert gs4.graph.get_vertex("D").colour == "black"
        gs4.graph.get_vertex("D").parent
        #assert gs4.graph.get_vertex("D").parent == gs4.graph.get_vertex("B")
        ans = gs4.get_shortest_path("A", "D")
        assert ans == ["A", "B", "D"]
        ans = gs4.get_shortest_path("E", "D")
        assert ans == ["E", "F", "C", "D"]


        ---------------------------------------------------------------------------
```

```
    NameError                                 Traceback (most recent call last)

    <ipython-input-2-7be1ff73c166> in <module>
----> 1 g4 = GraphSearch()
      2 g4.add_vertex("A")
      3 g4.add_vertex("B")
      4 g4.add_vertex("C")
      5 g4.add_vertex("D")


    NameError: name 'GraphSearch' is not defined
```

```
In [25]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

Week04_Homework

October 25, 2021

1 Week 4 Problem Set

1.1 Homeworks

HW1. Implement `Queue` abstract data structure using a Class. This `Queue` only stores items of Integer data type. You can use a `list` as its internal data structure. The class should have the following interface: - `__init__()` to initialize an empty List for the queue to store the items. - `enqueue(item)` which inserts an Integer into the queue. - `dequeue()` which returns and removes the element at the head of the queue. The return value is an optional as it may return `None` if there are no more elements in the queue. - `peek()` which returns the element at the head of the queue.

The class Queue has two computed properties: - `is_empty` which returns either `True` or `False` depending on whether the queue is empty or not. - `size` which returns the number of items in the queue.

```
In [1]: class Queue:
            def __init__(self):
                self.__items = []

            def enqueue(self, item):
                self.__items.append(item)

            def dequeue(self):
                return self.__items.pop(0) if not self.is_empty else None

            def peek(self):
                return self.__items[0]

            @property
            def is_empty(self):
                return len(self.__items) == 0

            @property
            def size(self):
                return len(self.__items)
```

```
In [2]: q1 = Queue()
        q1.enqueue(2)
        assert not q1.is_empty
        assert q1.size == 1
        ans = q1.dequeue()
        assert ans == 2
        assert q1.is_empty
        q1.enqueue(1)
        q1.enqueue(2)
        q1.enqueue(3)
        assert q1.size == 3
        assert q1.peek() == 1
        assert q1.dequeue() == 1
        assert q1.dequeue() == 2
        assert q1.dequeue() == 3
```

```
In [3]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

```
In [4]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

HW2. We are going to create a class that contains both `RobotTurtle` and `Coordinate` class. The name of the class is `TurtleWorld` which is used to simulate when `RobotTurtle` is moving around some two dimensional space. The class has the following methods:

- `add_turtle(name)` which is to add a new `RobotTurtle` into the world with the specified name.
- `remove_turtle(name)` which is to remove the object `RobotTurtle` with the specified name from the world.
- `list_turtles()` which is to list all the turtles in the world using their names in an ascending order.

We give you here the class definition for the `Coordinate` and the `RobotTurtle` from the Notes.

```
In [6]: import math

        class Coordinate:

            def __init__(self, x=0, y=0):
                self.x = x
                self.y = y

            @property
            def distance(self):
                return math.sqrt(self.x * self.x + self.y * self.y)

            def __str__(self):
                return f"({self.x}, {self.y})"
```

```
In [7]: # Class definition
        class RobotTurtle:
            # Attributes:
            def __init__(self, name, speed=1):
                self.name = name
                self.speed = speed
                self._pos = Coordinate(0, 0)

            # property getter
            @property
            def name(self):
                return self._name

            # property setter
            @name.setter
            def name(self, value):
                if isinstance(value, str) and value != "":
                    self._name = value

            # property getter
            @property
            def speed(self):
                return self._speed

            # property setter
            @speed.setter
            def speed(self, value):
                if isinstance(value, int) and value > 0:
                    self._speed = value

            # property getter
            @property
            def pos(self):
                return self._pos

            # Methods:
            def move(self, direction):
                update = {'up' : Coordinate(self.pos.x, self.pos.y + self.speed),
                          'down' : Coordinate(self.pos.x, self.pos.y - self.speed),
                          'left' : Coordinate(self.pos.x - self.speed, self.pos.y),
                          'right' : Coordinate(self.pos.x + self.speed, self.pos.y)}
                self._pos = update[direction]


            def tell_name(self):
                print(f"My name is {self.name}")
```

Now fill in the class definition for `TurtleWorld`. You may want to look into the test cases in the following cell to make sure you define the class properly.

```
In [14]: class TurtleWorld:

             def __init__(self):
                 self.turtles = {}

             def add_turtle(self, name, speed):
                 self.turtles[name] = RobotTurtle(name,speed)


             def remove_turtle(self, name):
                 if self.turtles.get(name):
                     del self.turtles[name]


             def list_turtles(self):
                 return sorted([name for name in self.turtles.keys()])
```

```
In [15]: world = TurtleWorld()
         world.add_turtle('t1', 1)
         assert world.list_turtles() == ['t1']

         world.add_turtle('t2', 2)
         assert world.list_turtles() == ['t1', 't2']

         world.add_turtle('abc', 3)
         assert world.list_turtles() == ['abc', 't1', 't2']

         world.remove_turtle('t2')
         assert world.list_turtles() == ['abc', 't1']

         world.remove_turtle('abc')
         assert world.list_turtles() == ['t1']
```

```
['t1']
['t1', 't2']
['t1', 't2', 'abc']
['t1', 'abc']
['t1']
```

```
In [9]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

HW3. Modify the class `TurtleWorld` to add the following method: - `move_turtle(name, movement)` which is to move the turtle with the specified name with a given input `movement`. The argument `movement` is a string containing letters: `l` for left, `r` for right, `u` for up, and `d` for down.

The movement should be based on the speed. This means that if the turtle has speed of 2 and the `movement` argument is `uulrdd`, the turtle should move up four units, move left two units, move right two units and move down four units.

```
In [24]: class TurtleWorld:
             valid_movements = set('udlr')
             movement_map = {'u': 'up', 'd': 'down', 'l': 'left', 'r': 'right'}

             def __init__(self):
                 self.turtles = {}

             def move_turtle(self, name, movement):
                 if not set(movement).issubset(self.valid_movements) or not self.turtles.get(na
                     return

                 for i in movement:
                     self.turtles[name].move(self.movement_map[i])
                 print(self.turtles[name].pos)

             def add_turtle(self, name, speed):
                 self.turtles[name] = RobotTurtle(name, speed)

             def remove_turtle(self, name):
                 if self.turtles.get(name):
                     del self.turtles[name]

             def list_turtles(self):
                 return sorted([name for name in self.turtles.keys()])

In [25]: world = TurtleWorld()
         world.add_turtle('abc', 1)
         world.move_turtle('abc', 'uu')
         assert str(world.turtles['abc'].pos) == '(0, 2)'

         world.move_turtle('abc', 'rrr')
         assert str(world.turtles['abc'].pos) == '(3, 2)'

         world.move_turtle('abc', 'd')
         assert str(world.turtles['abc'].pos) == '(3, 1)'

         world.move_turtle('abc', 'llll')
         assert str(world.turtles['abc'].pos) == '(-1, 1)'

         world.add_turtle('t1', 2)
         world.move_turtle('t1', 'uulrdd')
         assert str(world.turtles['t1'].pos) == '(0, 0)'

         world.move_turtle('t1', 'urur ur')
         assert str(world.turtles['t1'].pos) == '(6, 6)'
```

```
(0, 2)
(3, 2)
(3, 1)
(-1, 1)
(0, 0)
(6, 6)
```

```
In [26]: ###BEGIN HIDDEN TEST
         world = TurtleWorld()
         world.add_turtle('dabc', 1)
         world.move_turtle('dabc', 'u')
         assert str(world.turtles['dabc'].pos) == '(0, 1)'

         world.move_turtle('dabc', 'rr')
         assert str(world.turtles['dabc'].pos) == '(2, 1)'

         world.move_turtle('dabc', 'dd')
         assert str(world.turtles['dabc'].pos) == '(2, -1)'

         world.move_turtle('dabc', 'llll')
         assert str(world.turtles['dabc'].pos) == '(-2, -1)'

         world.add_turtle('t1', 2)
         world.move_turtle('t1', 'ulrd')
         assert str(world.turtles['t1'].pos) == '(0, 0)'

         world.move_turtle('t1', 'dldldl')
         assert str(world.turtles['t1'].pos) == '(-6, -6)'
         ###BEGIN HIDDEN TEST
```

```
(0, 1)
(2, 1)
(2, -1)
(-2, -1)
(0, 0)
(-6, -6)
```

HW4. Modify the class `TurtleWorld` to include the following method: - `add_movement(turtle, movement)` which adds turtle movement to a queue to be run later. The argument `turtle` is a string containing the turtle's name. The argument `movement` is another string for the movement. For example, value for `turtle` can be something like `'t1'` while the value for the `movement` can be something like `'uullrrdd'`. - `run()` which executes all the movements in the queue.

```
In [35]: class Queue:
             def __init__(self):
                 self.__items = []
```

```
    def enqueue(self, item):
        self.__items.append(item)

    def dequeue(self):
        return self.__items.pop(0) if not self.is_empty else None

    def peek(self):
        return self.__items[0]

    @property
    def is_empty(self):
        return len(self.__items) == 0

    @property
    def size(self):
        return len(self.__items)

class TurtleWorld:
    valid_movements = set('udlr')
    movement_map = {'u': 'up', 'd': 'down', 'l': 'left', 'r': 'right'}

    def __init__(self):
        self.turtles = {}
        self.queue = Queue()

    def add_movement(self, name, movement):
        if not set(movement).issubset(self.valid_movements) or not self.turtles.get(na
            return

        self.queue.enqueue((name, movement))

    def run(self):
        while not self.queue.is_empty:
            name, movement = self.queue.dequeue()
            self.move_turtle(name, movement)

    def move_turtle(self, name, movement):
        if not set(movement).issubset(self.valid_movements) or not self.turtles.get(na
            return

        for i in movement:
            self.turtles[name].move(self.movement_map[i])

    def add_turtle(self, name, speed):
        self.turtles[name] = RobotTurtle(name, speed)

    def remove_turtle(self, name):
```

```
                if self.turtles.get(name):
                    del self.turtles[name]

            def list_turtles(self):
                return sorted([name for name in self.turtles.keys()])
```

```
In [36]: world = TurtleWorld()
         world.add_turtle('t1', 1)
         world.add_turtle('t2', 2)
         world.add_movement('t1', 'ur')
         world.add_movement('t2', 'ur')
         assert str(world.turtles['t1'].pos) == '(0, 0)'
         assert str(world.turtles['t2'].pos) == '(0, 0)'

         world.run()
         assert str(world.turtles['t1'].pos) == '(1, 1)'
         assert str(world.turtles['t2'].pos) == '(2, 2)'

         world.run()
         assert str(world.turtles['t1'].pos) == '(1, 1)'
         assert str(world.turtles['t2'].pos) == '(2, 2)'
```

```
In [37]: ###BEGIN HIDDEN TEST
         world = TurtleWorld()
         world.add_turtle('t1', 1)
         world.add_turtle('t2', 2)
         world.add_movement('t2', 'dldldl')
         world.add_movement('t1', 'dldldl')

         world.run()
         assert str(world.turtles['t1'].pos) == '(-3, -3)'
         assert str(world.turtles['t2'].pos) == '(-6, -6)'
         ###END HIDDEN TEST
```

HW5. Implement a radix sorting machine. A radix sort for base 10 integers is a *mechanical* sorting technique that utilizes a collection of bins: - one main bin - 10 digit-bins

Each bin acts like a *queue* and maintains its values in the order that they arrive. The algorithm works as follows: - it begins by placing each number in the main bin. - Then it considers each value digit by digit. The first value is removed from the main bin and placed in a digit-bin corresponding to the digit being considered. For example, if the ones digit is being considered, 534 will be placed into digit-bin 4 and 667 will placed into digit-bin 7. - Once all the values are placed into their corresponding digit-bins, the values are collected from bin 0 to bin 9 and placed back in the main bin (in that order). - The process continues with the tens digit, the hundreds, and so on. - After the last digit is processed, the main bin will contain the values in ascending order.

Create a class `RadixSort` that takes in a List of Integers during object instantiation. The class should have the following properties: - `items`: is a List of Integers containing the numbers.

It should also have the following methods: - `sort()`: which returns the sorted numbers from `items` as an `list` of Integers. - `max_digit()`: which returns the maximum number of digits of all the numbers in `items`. For example, if the numbers are 101, 3, 1041, this method returns 4 as the result since the maximum digit is four from 1041. - `convert_to_str(items)`: which returns items as a list of Strings (instead of Integers). This function should pad the higher digits with 0 when converting an Integer to a String. For example if the maximum digit is 4, the following items are converted as follows. From `[101, 3, 1041]` to `["0101", "0003", "1041"]`.

Hint: Your implementation should make use of the generic `Queue` class, which you created, for the bins.

```
In [48]: class RadixSort:

             def __init__(self, MyList: list):
                 self.items = MyList
                 self.main_bin = Queue()
                 self.radix_bins = [Queue() for i in range(10)]

             def max_digit(self):
                 # Returns the maximum number of digit in the list
                 return max([len(str(i)) for i in self.items])

             def convert_to_str(self, items):
                 max_digit = self.max_digit()
                 for i in range(len(self.items)):
                     print(i, self.items[i])
                     self.items[i] = str(items[i]).zfill(max_digit)
                 return self.items

             def sort(self):
                 res = []
                 # find the maximum digit
                 max_digit = self.max_digit()
                 # convert all the items to strings and put them in the main bin
                 self.convert_to_str(self.items)
                 for i in self.items:
                     self.main_bin.enqueue(i)

                 for i in range(max_digit - 1, -1, -1):
                     # put the items in the corresponding radix bin as per i
                     while not self.main_bin.is_empty:
                         item = self.main_bin.dequeue()
                         self.radix_bins[int(item[i])].enqueue(item)
                         print(self.radix_bins[i].items)

                     # dequeue the items from the radix bin and put them in the main bin
                     for j in range(len(self.radix_bins)):
                         radix_bin = self.radix_bins[j]
                         while not radix_bin.is_empty:
                             self.main_bin.enqueue(radix_bin.dequeue())
```

```
            # return the sorted list
            while not self.main_bin.is_empty:
                res.append(int(self.main_bin.dequeue()))

            return res


    class Queue:
        def __init__(self):
            self.__items = []

        def enqueue(self, item):
            self.__items.append(item)

        def dequeue(self):
            return self.__items.pop(0) if not self.is_empty else None

        def peek(self):
            return self.__items[0]

        @property
        def items(self):
            return self.__items

        @property
        def is_empty(self):
            return len(self.__items) == 0

        @property
        def size(self):
            return len(self.__items)
```

```
In [49]: list1 = RadixSort([101, 3, 1041])
         assert list1.items == [101,3,1041]
         assert list1.max_digit() == 4
         assert list1.convert_to_str(list1.items) == ["0101", "0003", "1041"]
         ans = list1.sort()
         print(ans)
         assert ans == [3, 101, 1041]
         list2 = RadixSort([23, 1038, 8, 423, 10, 39, 3901])
         assert list2.sort() == [8, 10, 23, 39, 423, 1038, 3901]
```

```
0 101
1 3
2 1041
0 0101
1 0003
2 1041
```

```
[]
['0003']
['0003']
[]
[]
[]
['0101']
['0101']
['0101']
['0003']
['0003']
['0003', '0101']
[3, 101, 1041]
0 23
1 1038
2 8
3 423
4 10
5 39
6 3901
['0023']
['0023']
['0023']
['0023', '0423']
['0023', '0423']
['0023', '0423']
['0023', '0423']
[]
[]
['0023']
['0023', '0423']
['0023', '0423']
['0023', '0423']
['0023', '0423']
[]
[]
[]
[]
[]
[]
[]
['0008']
['0008', '0010']
['0008', '0010', '0023']
['0008', '0010', '0023']
['0008', '0010', '0023', '0039']
['0008', '0010', '0023', '0039', '0423']
['0008', '0010', '0023', '0039', '0423']
```

```
In [18]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

Week04_Cohort

October 25, 2021

1 Week 4 Problem Set

1.1 Cohort Sessions

CS1. We are going to create a simple Car Racing game. First, let's create a class Car with the following properties: - `racer` which stores the name of the driver. This property must be non-empty string. This property should be initialized upon object instantiation. - `speed` which stores the speed of the car. This property can only be non-negative values and must be less than a maximum speed. - `pos` which is an integer specifying the position of the car which can only be non-negative values. - `is_finished` which is a computed property that returns `True` or `False` depending whether the position has reached the finish line.

Each car also has the following attributes: - `max_speed` which specifies the maximum speed the car can have. This attribute should be initialized upon object instantiation. - `finish` which stores the finish distance the car has to go through. Upon initialization, it should be set to -1.

The class has the following methods: - `start(init_speed, finish_distance)` which set the speed property to some initial value. The method also set the finish distance to some value and set the `pos` property to 0. - `race(acceleration)` which takes in an integer value for its acceleratin and modify both the speed and the position of the car.

```
In [1]: class RacingCar:

            def __init__(self, name, max_speed):
                self._racer = name
                self.max_speed = max_speed
                self.finish = -1
                self._speed = 0
                self._pos = 0

            @property
            def racer(self):
                return self._racer

            @racer.setter
            def racer(self, name):
                if isinstance(name, str) and name != "":
                    self._racer = name

            @property
```

```
        def speed(self):
            return self._speed

        @speed.setter
        def speed(self, val):
            if isinstance(val, int) and 0 <= val <= self.max_speed:
                self._speed = val
                self._pos = val

        @property
        def pos(self):
            return self._pos

        @pos.setter
        def pos(self, val):
            if isinstance(val, int) and 0 <= val <= self.max_speed:
                self._pos = val

        @property
        def is_finished(self):
            return self._pos >= self.finish >0

        def start(self, init_speed, finish_dist):
            self._speed = init_speed
            self.finish = finish_dist
            self._pos = 0

        def race(self, acc):
            self._speed += acc
            self._pos += self._speed

        def __str__(self):
            return f"Racing Car {self.racer} at position: {self.pos}, with speed: {self.spe
```

```
In [2]: car = RacingCar("Hamilton", 200)
        assert car.racer == "Hamilton"
        assert car.max_speed == 200
        assert car.finish == -1

        car.racer = "Bottas"
        assert car.racer == "Bottas"
        car.racer = ""
        assert car.racer == "Bottas"
        car.racer = 21
        assert car.racer == "Bottas"

        car.speed = 10
        assert car.speed == 10
```

```
car.speed = 0
assert car.speed == 0
car.speed = -10
assert car.speed == 0
car.speed = car.max_speed
assert car.speed == car.max_speed
car.speed = car.max_speed + 10
assert car.speed == car.max_speed

car.pos = 10
assert car.pos == 10
car.pos = -10
assert car.pos == 10
car.pos = 0
assert car.pos == 0

assert not car.is_finished
car.finish = 20
car.pos = 10
assert not car.is_finished
car.pos = 20
assert car.is_finished
car.pos = 22
assert car.is_finished

car.start(50, 200)
assert car.pos == 0
assert car.speed == 50
assert car.finish == 200

car.race(0)
assert car.speed == 50
assert car.pos == 50
assert not car.is_finished

car.race(10)
assert car.speed == 60
assert car.pos == 110
assert not car.is_finished

car.race(-10)
assert car.speed == 50
assert car.pos == 160
assert not car.is_finished

car.race(0)
assert car.is_finished
```

CS2. Implement a `RacingGame` class that plays car racing using Python `random` module to simulate car's acceleration. The class has the following attribute(s): - `car_list` which is a dictionary containing all the `RacingCar` objects where the keys are the racer's name.

The class has the following properties: - `winners` which list the winners from the first to the last. If there is no winner, it should return `None`.

Upon instantiation, it should initalize the game with some **random seed**. This is to ensure that the behaviour can be predicted.

It has the following methods: - `add_car(name, max_speed)` which creates a new `RacingCar` object and add it into the `car_list`. - `start(finish_distance)` which uses the `random` module to assign different initial speeds (0 to 50) to each of the racing car and set the same finish distance for all cars. - `play(finish)` which contains the main loop of the game that calls the `RacingCar`'s method `race()` until all cars reach the finish line. It takes in an argument for the finish distance.

```
In [3]: import random

        class RacingGame:

            def __init__(self, seed):
                self.car_list = {}
                self._winners = []
                random.seed(seed)

            @property
            def winners(self):
                return None if len(self._winners) == 0 else self._winners

            def add_car(self, name, speed):
                self.car_list[name] = RacingCar(name,speed)

            def start(self, finish):
                for _,car in self.car_list.items():
                    car.speed = random.randint(0,50)
                    car.finish = finish
                    car.pos = 0


            def play(self, finish):
                self.start(finish)
                finished_car = 0
                while True:
                    for racer, car in self.car_list.items():
                        if not car.is_finished:
                            acc = random.randint(-10, 20)
                            car.race(acc)
                            # you can comment out the line below to check the output
                            print(car)
                            if car.is_finished:
                                self._winners.append(racer)
                            finished_car +=1
                    if finished_car == len(self.car_list):
                        break
```

```
In [4]: game = RacingGame(100)
        assert game.car_list == {}
        assert game.winners == None

        game.add_car("Hamilton", 250)
        assert len(game.car_list) == 1
        assert game.car_list["Hamilton"].racer == "Hamilton"

        game.add_car("Vettel", 200)
        assert len(game.car_list) == 2
        assert game.car_list["Vettel"].racer == "Vettel"

        game.start(200)
        assert [ car.pos for car in game.car_list.values()] == [0, 0]
        assert [ car.speed for car in game.car_list.values()] == [9, 29]
        assert [ car.finish for car in game.car_list.values()] == [200, 200]

        game.play(200)
        assert game.winners == ["Vettel", "Hamilton"]

        game = RacingGame(200)
        game.add_car("Hamilton", 250)
        game.add_car("Vettel", 200)
        game.play(200)
        assert game.winners == ["Hamilton", "Vettel"]
```

```
Racing Car Hamilton at position: 24, with speed: 24.
Racing Car Vettel at position: 61, with speed: 61.
Racing Car Hamilton at position: 50, with speed: 26.
Racing Car Vettel at position: 135, with speed: 74.
Racing Car Hamilton at position: 77, with speed: 27.
Racing Car Vettel at position: 212, with speed: 77.
Racing Car Hamilton at position: 110, with speed: 33.
Racing Car Hamilton at position: 158, with speed: 48.
Racing Car Hamilton at position: 199, with speed: 41.
Racing Car Hamilton at position: 247, with speed: 48.
Racing Car Hamilton at position: 15, with speed: 15.
Racing Car Vettel at position: 25, with speed: 25.
Racing Car Hamilton at position: 20, with speed: 5.
Racing Car Vettel at position: 44, with speed: 19.
Racing Car Hamilton at position: 34, with speed: 14.
Racing Car Vettel at position: 80, with speed: 36.
Racing Car Hamilton at position: 63, with speed: 29.
Racing Car Vettel at position: 114, with speed: 34.
Racing Car Hamilton at position: 111, with speed: 48.
Racing Car Vettel at position: 138, with speed: 24.
Racing Car Hamilton at position: 171, with speed: 60.
Racing Car Vettel at position: 166, with speed: 28.
Racing Car Hamilton at position: 243, with speed: 72.
Racing Car Vettel at position: 189, with speed: 23.
Racing Car Vettel at position: 224, with speed: 35.
```

```
In [5]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

CS3. Implement the `Stack` abstract data type using a Class. You can use `list` Python data type as its internal data structure. Name this `list` as `items`.

The class should have the following interface: - `__init__()` to initialize an empty list for the stack to store the items. - `push(item)` which stores an Integer into the top of the stack. - `pop()` which returns and removes the top element of the stack. The return value is optional as it may return `None` if there are no more elements in the stack. - `peek()` which returns the top element of the stack. If the stack is empty, it returns `None`.

The class should have the following properties: - `is_empty` is a computed property which returns either `true` or `false` depending whether the stack is empty or not. - `size` is a computed property which returns the number of items in the stack.

```
In [6]: class Stack:
            def __init__(self):
                self.__items = []

            def push(self, item):
                self.__items.append(item)

            def pop(self):
                return self.__items.pop(-1) if not self.is_empty else None

            def peek(self):
                return self.__items[-1]

            @property
            def is_empty(self):
                return len(self.__items) == 0

            @property
            def size(self):
                return len(self.__items)
```

```
In [7]: s1 = Stack()
        s1.push(2)
        assert not s1.is_empty
        assert s1.pop() == 2
        assert s1.is_empty
        assert s1.pop() == None
        s1.push(1)
        s1.push(2)
        s1.push(3)
        assert not s1.is_empty
        assert s1._Stack__items == [1, 2, 3]
        assert s1.peek() == 3
        assert s1.size == 3
```

```
In [8]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

CS4. Write a class called `EvaluatePostfix` that evaluates postfix notation implemented using Stack data structure. Postfix notation is a way of writing expressions without parenthesis. For example, the expression `(1+2)*3` would be written as `1 2 + 3 *`. The class `EvaluatePostfix` has the following methods: - `input(inp)`: which pushes the input one at a time. For example, to create a postfix notation `1 2 + 3 *`, we can call this method repetitively, e.g. `e.input('1'); e.input('2'); e.input('+'); e.input('3'); e.input('*')`. Notice that the input is of String data type. - `evaluate()`: which returns the output of the expression.

Postfix notation is evaluated using a Stack. The input streams from `input()` are stored in a Queue, which we will implement using Python's List. Note: If you have finished your homework on Queue, you can replace this part with your Queue.

If the output of the Queue is a number, the item is pushed onto the stack. If it is an operator, we will apply the operator to the top two items in the stacks, pushing the result back onto the stack.

```
In [9]: import operator


        class EvaluatePostfix:
            ops = {
                '+' : operator.add,
                '-' : operator.sub,
                '*' : operator.mul,
                '/' : operator.truediv,  # use operator.div for Python 2
                '%' : operator.mod,
                '^' : operator.xor}
            operands = "0123456789"


            def __init__(self):
                self.expression = Queue()
                self.stack = Stack()
```

```
    def input(self, item):
        self.expression.enqueue(item)

    def evaluate(self):
        while self.expression.size != 0:
            item = self.expression.dequeue()

            if item in self.operands:
                self.stack.push(item)

            elif item in self.ops:
                op1 = int(self.stack.pop())
                op2 = int(self.stack.pop())

                product = self.ops[item](op2, op1)
                self.stack.push(product)

        test = self.stack.pop()
#         print(test,type(test))
        return test




class Queue:

    def __init__(self):
        self.__items = []

    def enqueue(self, item):
        self.__items.append(item)

    def dequeue(self):
        return self.__items.pop(0) if not self.is_empty else None

    def peek(self):
        return self.__items[0]

    @property
    def is_empty(self):
        return len(self.__items) == 0

    @property
    def size(self):
        return len(self.__items)
```

```
In [10]: pe = EvaluatePostfix()
```

```
         pe.input("2")
         pe.input("3")
         pe.input("+")
         assert pe.evaluate()== 5

         pe.input("2")
         pe.input("3")
         pe.input("+")
         pe.input("6")
         pe.input("-")
         assert pe.evaluate()== -1
```

```
In [11]: pe = EvaluatePostfix()
         pe.input("2")
         pe.input("3")
         pe.input("+")
         assert pe.evaluate()== 5

         pe.input("2")
         pe.input("3")
         pe.input("+")
         pe.input("6")
         pe.input("-")
         assert pe.evaluate()== -1
```

```
In [12]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

CS5. Implement a Queue abstract data structure using two Stacks instead of Python's list. For this double-stack implementation, the Queue has a *left* Stack and a *right* Stack. The enqueue and dequeue operations work as follows: - enqueue: This operation just pushes the new item into the *right* Stack. - dequeue: This operation is done as follows: - if the *left* Stack is empty: create a new *left* Stack which is the reverse of the items in the *right* Stack. You should then empty the *right* stack. - if the *left* Stack is not empty: pop from the *left* Stack.

```
In [13]: class Stack:
             def __init__(self):
                 self.__items = []

             def push(self, item):
                 self.__items.append(item)

             def pop(self):
                 return self.__items.pop(-1) if not self.is_empty else None

             def peek(self):
                 if len(self.__items)==0:
                     return None
```

```
            return self.__items[-1]

        @property
        def is_empty(self):
            return len(self.__items) == 0

        @property
        def size(self):
            return len(self.__items)
```

In [14]:
```
class Queue:
    def __init__(self):
        self.left_stack = Stack()
        self.right_stack = Stack()

    @property
    def is_empty(self):
        return self.left_stack.is_empty and self.right_stack.is_empty

    @property
    def size(self):
        return self.left_stack.size + self.right_stack.size


    def enqueue(self,item):
        self.right_stack.push(item)

    def dequeue(self):
        if self.left_stack.is_empty:
            while not self.right_stack.is_empty:
                self.left_stack.push(self.right_stack.pop())
        return self.left_stack.pop()

    def peek(self):
        if self.left_stack.is_empty:
            while not self.right_stack.is_empty:
                self.left_stack.push(self.right_stack.pop())
        x = self.left_stack.peek()
        return x
```

In [15]:
```
q1 = Queue()
q1.enqueue(2)
assert not q1.is_empty
assert q1.size == 1
assert q1.dequeue() == 2
assert q1.is_empty
q1.enqueue(1)
```

```
        q1.enqueue(2)
        q1.enqueue(3)
        assert q1.size == 3
        assert q1.peek() == 1
        assert q1.dequeue() == 1
        assert q1.dequeue() == 2
        assert q1.dequeue() == 3
        assert q1.peek() == None
```

```
In [46]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

CS6. You need to complete CS5 before attempting this question. Compute the computational time to do enqueue operation for a list based Queue implementation versus a double-stack based Queue implementation. Which one is faster? Why? There are a few parts you need to fill up. - `enqueue(q, array)`, which is a function to enqueue every element in the array to the Queue q. - `dequeue(q, array)`, which is a function to dequeue every element in the array from the Queue q. *Hint: you don't need the argument `array` but it is put here so that we can make use of the `run_function(f, x, y)`.*

You also need to replace some of the `None` in the code to compute the computational time inside the for-loop.

First you need to paste the Queue implementation using list-based.

```
In [47]: class Queue:
             def __init__(self):
                 self.__items = []

             def enqueue(self, item):
                 self.__items.append(item)

             def dequeue(self):
                 return self.__items.pop(0) if not self.is_empty else None

             def peek(self):
                 return self.__items[0]

             @property
             def is_empty(self):
                 return len(self.__items) == 0

             @property
             def size(self):
                 return len(self.__items)
```

```
In [48]: import time
         import random

         def run_function(f, x, y):
```

```
    start = time.time()
    f(x, y)
    end = time.time()
    return end-start

def enqueue(q, array):
    for item in array:
        q.enqueue(item)

def dequeue(q, array):
    while not q.is_empty:
        q.dequeue()

def gen_random_int(number, seed=100):
    random.seed(seed)
    sorted_list = [i for i in range(number)]
    random.shuffle(sorted_list)

    return sorted_list

def run_function(f, q,a):
    start = time.time()
    f(q,a)
    end = time.time()
    return end-start

time_enqueue_list = []
time_dequeue_list = []

# set the maximum power for 10^power number of inputs
maxpower = 5
for n in range(1, maxpower + 1):
    # create array for 10^1, 10^2, 10^3, etc
    # use seed = 100
    array = gen_random_int(n)
    print(f"random array {array}")

    # create queue
    queue =  Queue()

    # call run_function for enqueue
    result_enqueue = run_function(enqueue,queue,array)

    # call run_function for dequeue
    result_dequeue = run_function(dequeue,queue,array)


    time_enqueue_list.append(result_enqueue)
```

```
            time_dequeue_list.append(result_dequeue)

    print(time_enqueue_list)
    print(time_dequeue_list)
```

```
random array [0]
random array [1, 0]
random array [2, 1, 0]
random array [0, 2, 3, 1]
random array [2, 0, 4, 3, 1]
[1.430511474609375e-06, 1.1920928955078125e-06, 9.5367431640625e-07, 7.152557373046875e-07, 1.4
[3.5762786865234375e-06, 1.9073486328125e-06, 2.384185791015625e-06, 2.6226043701171875e-06, 3
```

Paste the code for the Queue using double Stack implementation.

```
In [49]: class Queue:
             def __init__(self):
                 self.left_stack = Stack()
                 self.right_stack = Stack()

             @property
             def is_empty(self):
                 return self.left_stack.is_empty and self.right_stack.is_empty

             @property
             def size(self):
                 return self.left_stack.size + self.right_stack.size


             def enqueue(self,item):
                 self.right_stack.push(item)

             def dequeue(self):
                 if self.left_stack.is_empty:
                     while not self.right_stack.is_empty:
                         self.left_stack.push(self.right_stack.pop())
                 return self.left_stack.pop()

             def peek(self):
                 if self.left_stack.is_empty:
                     while not self.right_stack.is_empty:
                         self.left_stack.push(self.right_stack.pop())
                 x = self.left_stack.peek()
                 return x
```

```
In [50]: import time
         import random
```

```
def run_function(f, x, y):
    start = time.time()
    f(x, y)
    end = time.time()
    return end-start

def enqueue(q, array):
    for item in array:
        q.enqueue(item)

def dequeue(q, array):
    while not q.is_empty:
        q.dequeue()

def gen_random_int(number, seed=100):
    random.seed(seed)
    sorted_list = [i for i in range(number)]
    random.shuffle(sorted_list)

    return sorted_list

def run_function(f, q,a):
    start = time.time()
    f(q,a)
    end = time.time()
    return end-start

time_enqueue_stack = []
time_dequeue_stack = []

# set the maximum power for 10^power number of inputs
maxpower = 5
for n in range(1, maxpower + 1):
    # create array for 10^1, 10^2, 10^3, etc
    # use seed = 100
    array = gen_random_int(n)
    print(f"random array {array}")

    # create queue
    queue =  Queue()

    # call run_function for enqueue
    result_enqueue = run_function(enqueue,queue,array)

    # call run_function for dequeue
    result_dequeue = run_function(dequeue,queue,array)
```

```
            time_enqueue_stack.append(result_enqueue)
            time_dequeue_stack.append(result_dequeue)

        print(time_enqueue_stack)
        print(time_dequeue_stack)
```

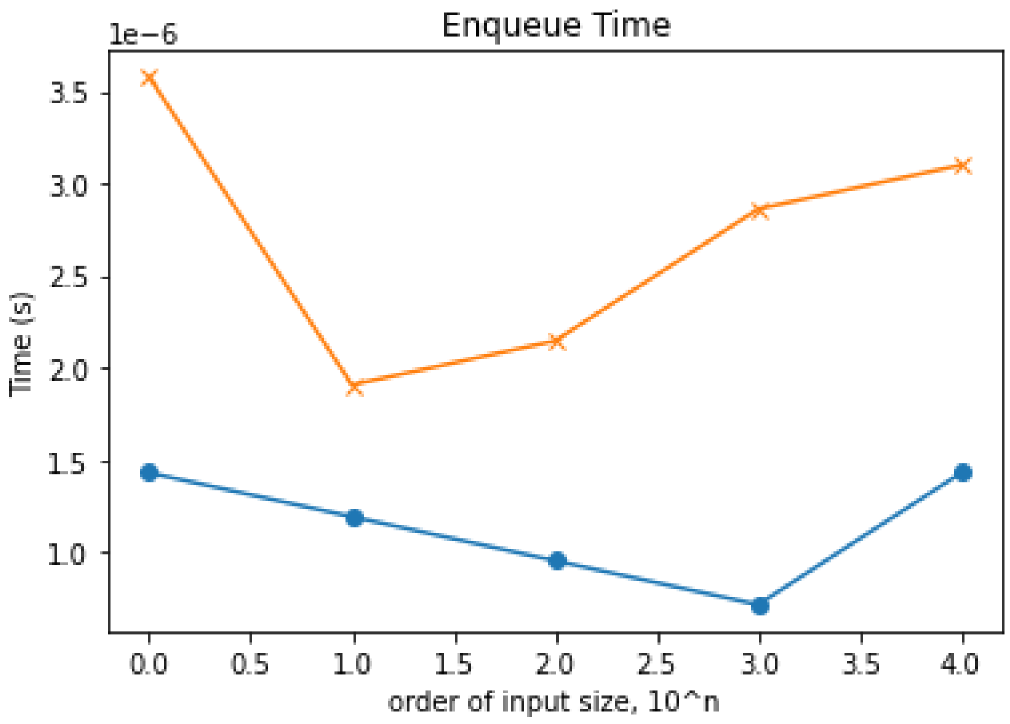
```
random array [0]
random array [1, 0]
random array [2, 1, 0]
random array [0, 2, 3, 1]
random array [2, 0, 4, 3, 1]
[3.5762786865234375e-06, 1.9073486328125e-06, 2.1457672119140625e-06, 2.86102294921875e-06, 3.0
[7.62939453125e-06, 1.9550323486328125e-05, 1.0967254638671875e-05, 1.239776611328125e-05, 1.59
```

```
In [51]: import matplotlib.pyplot as plt
```

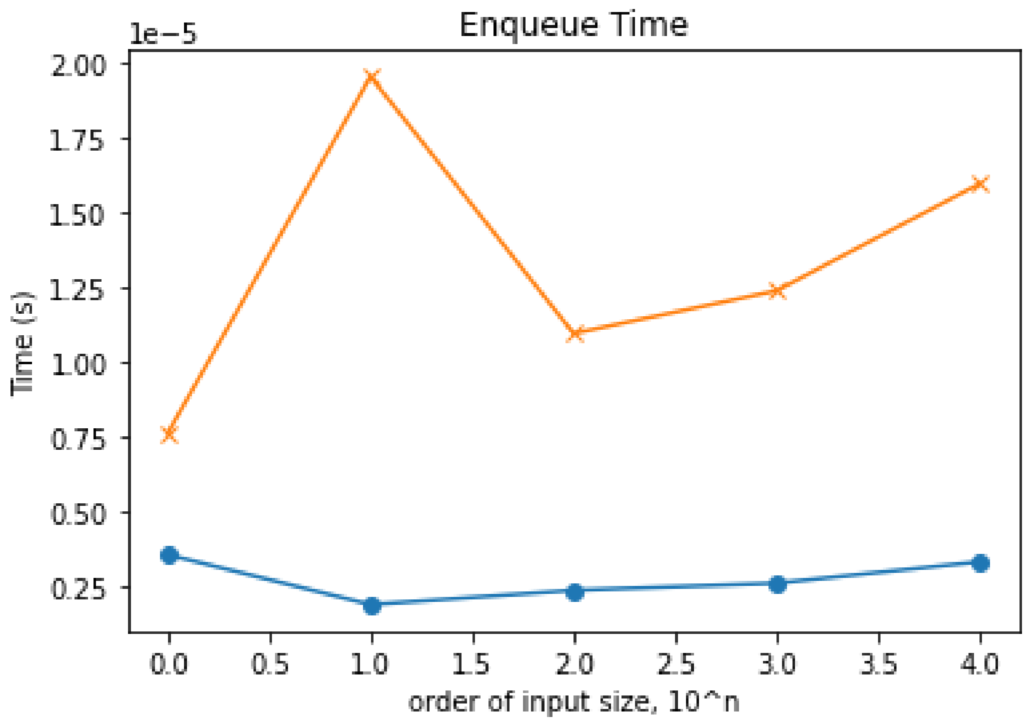
```
Matplotlib is building the font cache; this may take a moment.
```

```
In [52]: plt.plot(time_enqueue_list,'o-')
         plt.plot(time_enqueue_stack,'x-')
         plt.ylabel("Time (s)")
         plt.xlabel("order of input size, 10^n")
         plt.title("Enqueue Time")
```

```
Out[52]: Text(0.5, 1.0, 'Enqueue Time')
```



```
In [53]: plt.plot(time_dequeue_list,'o-')
         plt.plot(time_dequeue_stack,'x-')
         plt.ylabel("Time (s)")
         plt.xlabel("order of input size, 10^n")
         plt.title("dequeue Time")

Out[53]: Text(0.5, 1.0, 'Enqueue Time')
```



```
In [ ]:

In [ ]:
```

Week03_Homework

October 25, 2021

1 Week 3 Problem Set

1.1 Homeworks

HW1. *Fibonaci:* Write a function to find the Fibonacci number given an index. These are the example of a the first few numbers in Fibonacci series.

Indices	0	1	2	3	4	5	6	7	8	9
The series	0	1	1	2	3	5	8	13	21	34

We can write that the Fibonacci number at index i is given by:
$fibo(i) = fibo(i-1) + fibo(i-2)$
User recursion for your implementation.

```
In [9]: def fibonacci(index):
            if index == 0:
                return 0
            elif index == 1 or index == 2:
                return 1
            return fibonacci(index - 1) + fibonacci(index - 2)
```

```
In [10]: assert fibonacci(0) == 0
         assert fibonacci(1) == 1
         assert fibonacci(3) == 2
         assert fibonacci(7) == 13
         assert fibonacci(9) == 34
```

```
In [3]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

HW2. *Max-Heapify:* Recall the algorithm for max-heapify in restoring the *max-heap property* of a binary heap tree. In the previous implementation, we used iteration. Implement the same algorithm using recursion.

```
In [13]: def left_of(i):
             return 2*i + 1
```

```
        def right_of(i):
            return 2*i + 2

        def get_max_child(arr, l, r):
            """Returns the index of the larger child node"""
            if r >= len(arr):
                return l
            elif arr[l] > arr[r]:
                max_child_i = l
            else:
                max_child_i = r
            return max_child_i

        def max_heapify(array, i, heap_size):
            l = left_of(i)
            if l >= heap_size:
                return

            r = right_of(i)
            max_child_i = get_max_child(array, l, r)
            if array[max_child_i] > array[i]:
                array[max_child_i], array[i] = array[i], array[max_child_i]
                max_heapify(array,max_child_i,heap_size)
            return
```

```
In [14]: result = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]
         max_heapify(result, 1, len(result))
         print(result)
         assert result == [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
         result = [4, 1, 10, 14, 16, 9, 3, 2, 8, 7]
         max_heapify(result, 1, len(result))
         print(result)
         assert result == [4, 16, 10, 14, 7, 9, 3, 2, 8, 1]
```

```
[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
[4, 16, 10, 14, 7, 9, 3, 2, 8, 1]
```

```
In [6]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

HW3. *String Permutation:* Write two functions to return an array of all the permutations of a string. For example, if the input is `abc`, the output should be

```
output = ["abc", "acb", "bac", "bca", "cab", "cba"]
```

The first function only has one argument which is the input string, i.e. `permutate(s)`. The second function is the recursive function with two arguments String 1 (`str1`) and and String 2 (str2). The first function calls the second method `permutate("", s)` at the beginning. The second function should use a loop to move a character from `str2` to `str1` and recursively invokes it with a new `str1` and `str2`.

```
In [7]: def permutate(s2):
            res = []

            def helper(s1,s2):
                print(f"str1:{s1},str2:{s2}")
                # base case is when s2 does not have any more string
                if len(s2) == 0:
                    res.append(s1)

                # recursive case
                for i in range(len(s2)):
                    helper(s1+s2[i],s2[:i]+s2[i+1:])

            helper("",s2)

            return res
```

```
In [8]: print(permutate("abc"))
```

```
str1:,str2:abc
str1:a,str2:bc
str1:ab,str2:c
str1:abc,str2:
str1:ac,str2:b
str1:acb,str2:
str1:b,str2:ac
str1:ba,str2:c
str1:bac,str2:
str1:bc,str2:a
str1:bca,str2:
str1:c,str2:ab
str1:ca,str2:b
str1:cab,str2:
str1:cb,str2:a
str1:cba,str2:
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
```

```
In [39]: assert set(permutate("abc"))==set(["abc", "acb", "bac", "bca", "cab", "cba"])
         assert set(permutate("abcd"))==set(["abcd", "abdc", "acbd", "acdb", "adbc", "adcb", "
```

```
abc
acb
bac
bca
```

```
cab
cba
abcd
abdc
acbd
acdb
adbc
adcb
bacd
badc
bcad
bcda
bdac
bdca
cabd
cadb
cbad
cbda
cdab
cdba
dabc
dacb
dbac
dbca
dcab
dcba
```

```
In [9]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

HW4. *GCD:* Implement the recursive implementation of Euclid's algorithm for Greatest Common Divisor (GCD).

Hint: Wikipedia: Greatest Common Divisor

```
In [50]: def gcd(a,b):
             print(a,b)
             if a == b:
                 return a
             elif a == 0 and b != 0:
                 return b
             elif a != 0 and b == 0:
                 return a
             else:
                 if a > b:
                     return gcd(a-b,b)
                 else:
                     return gcd(a,b-a)
```

```
In [ ]:
```

```
In [51]: assert gcd(0,0) == 0
         assert gcd(2,0) == 2
         assert gcd(0,2) == 2
         assert gcd(2,3) == 1
         assert gcd(4, 28) == 4
         assert gcd(1024, 526) == 2
```

```
0 0
2 0
0 2
2 3
2 1
1 1
4 28
4 24
4 20
4 16
4 12
4 8
4 4
1024 526
498 526
498 28
470 28
442 28
414 28
386 28
358 28
330 28
302 28
274 28
246 28
218 28
190 28
162 28
134 28
106 28
78 28
50 28
22 28
22 6
16 6
10 6
4 6
4 2
2 2
```

```
In [12]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

Week03_Cohort

October 25, 2021

1 Week 3 Problem Set

1.1 Cohort Sessions

CS1. You have implemented factorial using iteration in Problem Set 1. Now, implement the factorial problem using a recursion. The function should takes in an Integer input and returns and Integer output which is the factorial of the input. Recall that:
$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$
You should consider the case when n is zero and one as well.

```
In [12]: # recursion
         def factorial_recursion(n):
             if n == 0 or n == 1:
                 return 1
             return n*factorial_recursion(n-1)
```

```
In [14]: assert factorial_recursion(0) == 1
         assert factorial_recursion(1) == 1
         assert factorial_recursion(5) == 120
         assert factorial_recursion(7) == 5040
         assert factorial_recursion(11) == 39916800
```

```
In [6]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

CS2. *Helper Function:* Write a function `palindrome(s)`to check if the string `s` is a Palindrome. To do this, write another function `is_palindrome(s, left, right)` where `left` and `right` are indices from the left and from the right to check the character in `str`. Use recursion instead of iteration in this problem.

```
In [34]: def palindrome(s):
             print(is_palindrome(s,0,len(s)-1))
             return is_palindrome(s,0,len(s)-1)

         def is_palindrome(s, left, right):
             print(s[left],s[right])
             if left >= right:
```

```
            return True
        # recursive case
        elif s[left] != s[right]:
            return False
        else:
            return is_palindrome(s, left+1, right-1)
```

```
In [35]: assert not palindrome("moon")
         assert palindrome("noon")
         assert palindrome("a a")
         assert palindrome("ada")
         assert not palindrome("ad a")
```

```
m n
False
m n
n n
o o
o o
True
n n
o o
o o
a a

True
a a

a a
d d
True
a a
d d
a a
d
False
a a
d
```

```
In [36]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

CS3. *Towers of Hanoi:* Write a function `move_disks(n, from_tower, to_tower, aux_tower)` which returns an array of String for the movement of disks that solves the Towers of Hanoi problem. * The first argument `n` is an Integer input that gives information on the number of disk. * The second argument `from_tower` is a String which is the label of the origin tower. * The third argument `to_tower` is a String which is the label of the destination tower. * The last argument `aux_tower` is a String which is the label of the auxilary tower.

```
In [65]: def move_disks(n, from_tower, to_tower, aux_tower):
             result = []
             def move(n, from_tower, to_tower, aux_tower):
                 if n == 1: # base case where there is only 1 left

                     result.append(print_move(n,from_tower,to_tower))
                 else:
                     # move the n-1 to aux first
                     move(n-1,from_tower,aux_tower,to_tower)
                     result.append(print_move(n,from_tower,to_tower))
                     move(n-1,aux_tower,to_tower,from_tower) # move the rest of the disk from
             move(n, from_tower, to_tower, aux_tower)

             return result

         def print_move(n,from_tower,to_tower):
             return f"Move disk {n} from {from_tower} to {to_tower}."
```

```
In [66]: result = move_disks(3,"A","B","C")
         print(result)
```

```
['Move disk 1 from A to B.', 'Move disk 2 from A to C.', 'Move disk 1 from B to C.', 'Move dis
```

```
In [67]: result = move_disks(3, "A", "B", "C")
         print(result)
         assert result == ["Move disk 1 from A to B.", "Move disk 2 from A to C.", "Move disk 

         result = move_disks(4, "A", "B", "C")
         print(result)
         assert result == ["Move disk 1 from A to C.", "Move disk 2 from A to B.", "Move disk 
```

```
['Move disk 1 from A to B.', 'Move disk 2 from A to C.', 'Move disk 1 from B to C.', 'Move dis
['Move disk 1 from A to C.', 'Move disk 2 from A to B.', 'Move disk 1 from C to B.', 'Move dis
```

```
In [12]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

CS4. *Merge Sort:* Write functions to implement the Merge Sort algorithm. The first function `mergesort(array)` should takes in an array of Integers in `array`. The function should sort the array in place. The second function `merge(array, p, q, r)` should implements the merge procedure. This function takes in an array of Integers in. `array`, the starting index for the left array `p`, the ending index for the left array `q`, and the ending index for the right array `r`. You can use a helper function for your recursion if needed.

```
In [73]: def mergesort(array):
             # split array into 2
             mid = len(array)//2
             mergesort_recursive(array,0,len(array)-1)


         # From pseudo code in the ddw notes
         def mergesort_recursive(arr, p, r):
             """
             p: index of beginning of array
             r : index of the end of array
             """
             # base case array contains only 1 element -> trivally sorted, so all we need to d

             # question : when do we merge the array of index 1?
             if r-p + 1 > 1:
                 q = (p+r)//2  # split the array
                 mergesort_recursive(arr, p, q)  # first half
                 mergesort_recursive(arr, q+1, r)  # second half
                 merge(arr, p, q, r)


         def merge(arr, p: int, q: int, r: int):
             """
             Merge the 2 arrays on left and right which are already sorted
             In place merging

             p : beginning index of the left array. This is also the beginning of the
             q : ending index of the left array. q+1 is the starting index of the right array
             r : ending index of the right array.

             The idea is to start from the beginning of the left and right arrays and compare
             The smaller number will be placed in position pointed by dest
             """
             n_left_arr = q - p + 1  # length of the left arr
             n_right_arr = r-q  # r - (q+1) +1

             print(n_left_arr, n_right_arr)

             left_arr = arr[p:q+1]
             right_arr = arr[(q+1):r+1]

             left = 0
             right = 0
             dest_i = p  # starting with the left of the sequence
             print(f"left arr {left_arr}, right arr {right_arr}")

             while left < n_left_arr and right < n_right_arr:
                 # comparison of left and right pointer and appending the smallest one to inde
                 if left_arr[left] <= right_arr[right]:
                     arr[dest_i] = left_arr[left]
                     left += 1
                 else:
                     arr[dest_i] = right_arr[right]
                     right += 1
                 dest_i += 1

             while left < n_left_arr:
                 arr[dest_i] = left_arr[left]
                 left += 1
                 dest_i += 1

             while right < n_right_arr:
                 arr[dest_i] = right_arr[right]
                 right += 1
                 dest_i += 1
```

```
In [74]: input_array = [5, 2, 4, 7, 1, 3, 2, 6]
         mergesort(input_array)
         assert input_array == [1, 2, 2, 3, 4, 5, 6, 7]
```

```
1 1
left arr [5], right arr [2]
1 1
left arr [4], right arr [7]
2 2
left arr [2, 5], right arr [4, 7]
1 1
left arr [1], right arr [3]
1 1
left arr [2], right arr [6]
2 2
left arr [1, 3], right arr [2, 6]
4 4
left arr [2, 4, 5, 7], right arr [1, 2, 3, 6]
```

```
In [15]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

Week02_Homework

October 25, 2021

1 Week 2 Problem Set

1.1 Homeworks

HW1. *Min-Heap:* Write the following function to implement *min-heap*. A *min-heap* is a binary heap that satisfies the *min-heap property*. This property can be described as:

```
For all nodes except the root:

A[left(i)] >= A[i]
A[right(i)] >= A[i]
```

- `min_child(heap, index)`: which returns the index of the node's smallest child. The node you are referring has index of value `index`
- `min_heapify(array, index, size)`: which moves the node at `index` down the tree so as to satisfy the *min-heap property*. The argument `index` is the index of the node you want to start moving down in the array. The argument `size` is the size of the heap. This size may be the same or less than the number of elements in the array. Hint: You may need the `min_child()` function.
- `build_min_heap(array)`: which build a *min-heap* from an arbitrary array of integers. This function should make use of `min_heapify(array, index)`.

```
In [44]: def left_of(index):
             return 2*index + 1

         def right_of(index):
             return 2*index + 2
```

```
In [6]: def min_child(heap, index):
            size = len(heap)
            left_child = left_of(index)
            right_child = right_of(index)
            if right_child >= size:
                return left_child
            elif heap[left_child] <= heap[right_child]:
                return left_child
            else:
                return right_child
```

```
In [7]: minheap = [1, 2, 4, 3, 9, 7, 8, 10, 14, 16]
        assert min_child(minheap, 0) == 1
        assert min_child(minheap, 2) == 5
        assert min_child(minheap, 3) == 7
        assert min_child(minheap, 1) == 3

In [30]: def min_heapify(array, index, size):
             curr = index
             swapped = True

             while left_of(curr) < size and swapped:
                 swapped = False
                 min_child_i = min_child(array, index)
                 if array[curr] > array[min_child_i]:
                     swapped = True
                     array[curr],array[min_child_i] = array[min_child_i], array[curr]
                 curr = min_child_i

In [31]: array = [1, 3, 4, 2, 9, 7, 8, 10, 14, 16]
         min_heapify(array, 1, len(array))
         assert array == [1, 2, 4, 3, 9, 7, 8, 10, 14, 16]

In [10]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###

In [132]: def build_min_heap(array):
              size = len(array)
              mid = size//2
              for i in range(mid,-1,-1):
          #         print(f"current swap element {i} , array {array}")
                  min_heapify(array, i, size)

          def min_heapify(array, index, size):
              curr = index
              swapped = True

              while left_of(curr) < size and swapped:
                  swapped = False
                  min_child_i = min_child(array, curr)
                  if array[curr] > array[min_child_i]:
                      array[curr],array[min_child_i] = array[min_child_i], array[curr]
                      swapped = True

                  curr = min_child_i

          def min_child(heap, index):
              size = len(heap)
              left_child = left_of(index)
```

```
            right_child = right_of(index)
            if right_child >= size:
                return left_child
            elif heap[left_child] < heap[right_child]:
                return left_child
            else:
                return right_child
```

```
In [100]: arr = [6, 5, 7, 8, 9, 4]
          build_min_heap(arr)
          print(arr)
```

```
current swap element 3 , array [6, 5, 7, 8, 9, 4]
current swap element 2 , array [6, 5, 7, 8, 9, 4]
current swap element 1 , array [6, 5, 4, 8, 9, 7]
current swap element 0 , array [6, 5, 4, 8, 9, 7]
[4, 5, 6, 8, 9, 7]
```

```
In [50]: array = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
         print(array)
         build_min_heap(array)
         print(array)
         assert array == [1, 2, 3, 4, 7, 9, 10, 8, 16, 14]
```

```
[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
current swap element 5
current swap element 4
current swap element 3
current swap element 2
current swap element 1
current swap element 0
[1, 2, 3, 4, 7, 9, 10, 8, 16, 14]
```

HW2. *Heapsort:* Implement heapsort that makes use of *min-heap* instead of *max-heap*. This function returns a *new* array. The strategy is similar to max-heap, but we will use a new array to store the sorted output. Take note of the hints below: - The top of the min-heap is always the smallest. You can take this element and put it into the output array. - To find the next minimum, take the last element of the heap and put it into the first element of the array. Now, the tree is no longer a min-heap. Use `min_heapify()` to restore the min-heap property. This will result in a mean-heap where the first element of the array is the next minimum. You can then take out the top of the min-heap and put it into the output array. - Reduce the heap size as you go. - Return the new output array.

```
In [53]: import random


         def gen_random_int(number, seed):
             random.seed(seed)
```

```
            sorted_list = [i for i in range(number)]
            random.shuffle(sorted_list)

            return sorted_list

In [145]: def heapsort(array):
              result = []
              build_min_heap(array)
              n = len(array)
              while len(array) > 0:
                  result.append(array[0])
                  print(f"before heapify {array}")
                  print("==============================")
                  min_heapify(array,0,len(array))

              return result

          def min_heapify(array, index, size):
              curr = index
              swapped = True

              while left_of(curr) < size and swapped:

                  swapped = False
                  min_child_i = min_child(array, curr)
                  if array[curr] > array[min_child_i]:
                      array[curr],array[min_child_i] = array[min_child_i], array[curr]
                      swapped = True

                  curr = min_child_i

          def min_child(heap, index):
              size = len(heap)
              left_child = left_of(index)
              right_child = right_of(index)
              if right_child >= size:
                  return left_child

              elif heap[left_child] < heap[right_child]:
                  print(f"left child {heap[left_child]},right {heap[right_child]}")
                  return left_child
              else:
                  print(f"left child {heap[left_child]},right {heap[right_child]}")
                  return right_child

In [146]: array = gen_random_int(10,100)
          print(array)
          build_min_heap(array)
```

```
    print(array)
    result = heapsort(array)
    print(result)
```

```
[4, 0, 5, 9, 3, 1, 6, 8, 7, 2]
left child 8,right 7
left child 1,right 6
left child 7,right 2
left child 0,right 1
left child 7,right 2
[0, 2, 1, 7, 3, 5, 6, 8, 9, 4]
left child 8,right 9
left child 5,right 6
left child 7,right 3
left child 2,right 1
before heapify [2, 1, 7, 3, 5, 6, 8, 9, 4]
==============================
left child 1,right 7
left child 3,right 5
before heapify [2, 7, 3, 5, 6, 8, 9, 4]
==============================
left child 7,right 3
before heapify [7, 3, 5, 6, 8, 9, 4]
==============================
left child 3,right 5
left child 6,right 8
before heapify [6, 5, 7, 8, 9, 4]
==============================
left child 5,right 7
left child 8,right 9
before heapify [6, 7, 8, 9, 4]
==============================
left child 7,right 8
before heapify [7, 8, 9, 4]
==============================
left child 8,right 9
before heapify [8, 9, 4]
==============================
left child 9,right 4
before heapify [9, 8]
==============================
before heapify [9]
==============================
before heapify []
==============================
[0, 1, 2, 3, 5, 6, 7, 4, 8, 9]
```

```
In [127]: array = gen_random_int(10, 100)
```

```
         result = heapsort(array)
         assert result == [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
current swap element 5 , array [4, 0, 5, 9, 3, 1, 6, 8, 7, 2]
current swap element 4 , array [4, 0, 5, 9, 3, 1, 6, 8, 7, 2]
before heapify [4, 0, 5, 9, 3, 1, 6, 8, 7, 2]
=============================
current swap element 3 , array [4, 0, 5, 9, 2, 1, 6, 8, 7, 3]
before heapify [4, 0, 5, 9, 2, 1, 6, 8, 7, 3]
=============================
current swap element 2 , array [4, 0, 5, 7, 2, 1, 6, 8, 9, 3]
before heapify [4, 0, 5, 7, 2, 1, 6, 8, 9, 3]
=============================
current swap element 1 , array [4, 0, 1, 7, 2, 5, 6, 8, 9, 3]
before heapify [4, 0, 1, 7, 2, 5, 6, 8, 9, 3]
=============================
current swap element 0 , array [4, 0, 1, 7, 2, 5, 6, 8, 9, 3]
before heapify [4, 0, 1, 7, 2, 5, 6, 8, 9, 3]
=============================
before heapify [0, 4, 1, 7, 2, 5, 6, 8, 9, 3]
=============================
before heapify [0, 2, 1, 7, 4, 5, 6, 8, 9, 3]
=============================
```

```
        ---------------------------------------------------------------------------

        AssertionError                            Traceback (most recent call last)

        <ipython-input-127-567faf0a54f9> in <module>
          1 array = gen_random_int(10, 100)
          2 result = heapsort(array)
    ----> 3 assert result == [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]


        AssertionError:
```

HW3. Compute the computational time for Heap Sort algorithm implemented in Python for various number of inputs. Make use of `gen_random_int(n)` to generate the input array. Use the template below to generate computation time for different number of inputs: 10, 100, 1000, etc.

```
In [117]: import time
          import random

          def run_function(f, x):
              start = time.time()
              f(x)
```

```
        end = time.time()
        return end-start

    def gen_random_int(number, seed):
        random.seed(seed)
        sorted_list = [i for i in range(number)]
        random.shuffle(sorted_list)

        return sorted_list

    time_heapsort = []
    # set the maximum power for 10^power number of inputs
    maxpower = 5

    for n in range(1, maxpower + 1):

        array = gen_random_int(n,100)
        result = run_function(heapsort,array)


        time_heapsort.append(result)

    print(time_heapsort)
```

```
current swap element 0 , array [0]
min_num 0 , array []
current swap element 0 , array []
heaped array []
================================
current swap element 1 , array [1, 0]
current swap element 0 , array [1, 0]
min_num 0 , array [1]
current swap element 0 , array [1]
heaped array [1]
================================
min_num 1 , array []
current swap element 0 , array []
heaped array []
================================
current swap element 1 , array [2, 1, 0]
current swap element 0 , array [2, 1, 0]
min_num 0 , array [1, 2]
current swap element 1 , array [1, 2]
current swap element 0 , array [1, 2]
heaped array [1, 2]
================================
min_num 1 , array [2]
current swap element 0 , array [2]
```

```
heaped array [2]
================================
min_num 2 , array []
current swap element 0 , array []
heaped array []
================================
current swap element 2 , array [0, 2, 3, 1]
current swap element 1 , array [0, 2, 3, 1]
current swap element 0 , array [0, 1, 3, 2]
min_num 0 , array [1, 3, 2]
current swap element 1 , array [1, 3, 2]
current swap element 0 , array [1, 3, 2]
heaped array [1, 3, 2]
================================
min_num 1 , array [3, 2]
current swap element 1 , array [3, 2]
current swap element 0 , array [3, 2]
heaped array [2, 3]
================================
min_num 2 , array [3]
current swap element 0 , array [3]
heaped array [3]
================================
min_num 3 , array []
current swap element 0 , array []
heaped array []
================================
current swap element 2 , array [2, 0, 4, 3, 1]
current swap element 1 , array [2, 0, 4, 3, 1]
current swap element 0 , array [2, 0, 4, 3, 1]
min_num 0 , array [1, 4, 3, 2]
current swap element 2 , array [1, 4, 3, 2]
current swap element 1 , array [1, 4, 3, 2]
current swap element 0 , array [1, 2, 3, 4]
heaped array [1, 2, 3, 4]
================================
min_num 1 , array [2, 3, 4]
current swap element 1 , array [2, 3, 4]
current swap element 0 , array [2, 3, 4]
heaped array [2, 3, 4]
================================
min_num 2 , array [3, 4]
current swap element 1 , array [3, 4]
current swap element 0 , array [3, 4]
heaped array [3, 4]
================================
min_num 3 , array [4]
current swap element 0 , array [4]
```

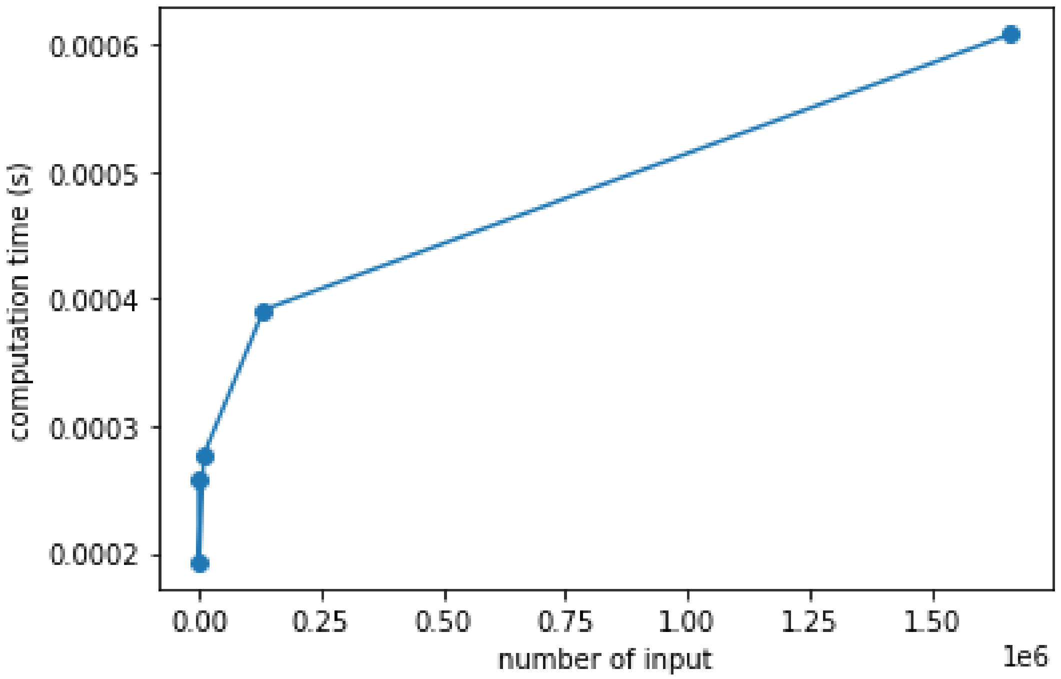
```
heaped array [4]
================================
min_num 4 , array []
current swap element 0 , array []
heaped array []
================================
[0.0002574920654296875, 0.00019311904907226562, 0.00027704238891601562, 0.000391244888305664
```

HW4. Plot the output of HW3 by first calculating a new x-axis computed as $n \log_2(n)$. Use the template below.

Reference: - Numpy Log2 function

```
In [118]: import matplotlib.pyplot as plt
          import numpy as np

          maxpower = 5
          # create an iterable from 1 to maxpowers
          powers = range(1, maxpower + 1)
          # variable n stores the number of items to sort
          n = []

          # Create a list of n for our x axis
          for exp in powers:
              n.append(10**exp)

          # convert to Numpy array
          n = np.array(n)

          # calculate n*log(n) for x axis
          x = n * np.log2(n)
          plt.plot(x, time_heapsort, "-o")
          plt.xlabel("number of input")
          plt.ylabel("computation time (s)")
          plt.show()
```

```
Matplotlib is building the font cache; this may take a moment.
```



0.0006
0.0005
0.0004
0.0003
0.0002
computation time (s)
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1e6
number of input

```
In [ ]:
```

Week02_Cohort

October 25, 2021

1 Week 2 Problem Set

1.1 Cohort Sessions

CS1. *Binary Heap:* Write the following functions:

- `parent_of(index)`: returns the index of node's parent
- `left_of(index)`: returns the index of node's left child
- `right_of(index)`: returns the index of node's right child
- `max_child(array, index, heap_size)`: returns the index of node's largest child. You can assume that the node has at least one child.

Hint: - `index` starts from 0. - You can refer to the pseudocodes in Binary Heap and Heapsort. - When finding the index of the largest child, consider the following cases: 1. when the node only has one child 2. when the node has two children

```
In [1]: def parent_of(index):
            return (index-1)//2

In [2]: assert parent_of(1) == 0
        assert parent_of(2) == 0
        assert parent_of(5) == 2
        assert parent_of(6) == 2

In [3]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###

In [4]: def left_of(index):
            return 2*index + 1

In [5]: assert left_of(0) == 1
        assert left_of(1) == 3
        assert left_of(3) == 7
        assert left_of(6) == 13

In [6]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

```
In [7]: def right_of(index):
            return 2*index + 2

In [8]: assert right_of(0) == 2
        assert right_of(1) == 4
        assert right_of(3) == 8
        assert right_of(5) == 12

In [9]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###

In [10]: def max_child(array, index, heap_size):
             left_child = left_of(index)
             right_child = right_of(index)
             if right_child >= heap_size:
                 return left_child
             elif array[left_child] >= array[right_child]:
                 return left_child
             else:
                 return right_child


         def left_of(index):
             return 2*index + 1

         def right_of(index):
             return 2*index + 2

In [11]: maxheap = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1] #This is not actually a max heap cause of
         assert max_child(maxheap, 0, len(maxheap)) == 1
         assert max_child(maxheap, 2, len(maxheap)) == 5
         assert max_child(maxheap, 3, len(maxheap)) == 8
         assert max_child(maxheap, 1, len(maxheap)) == 3

In [12]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

CS2. *Binary Heap:* Write two functions

- `max_heapify(array, index, size)`: that moves the node down so as to satisfy the heap property. The first argument is the array that contains the heap. The second argument is an integer index where to start the process of heapifying. The third argument is the size of the heap in the array. This argument will be useful in heapsort algorithm where we take out the elements in the array from the heap.

Hint: You should make use of `size` argument to determine the last element of the heap in the array rather than `len(array)`.

- `build_max_heap(heap)`: that builds the max heap from any array. This function should make use of `max_heapify()` in its definition.

Hint: You can refer to the pseudocode in Binary Heap and Heapsort for the above functions.

```
In [13]: def max_heapify(array, i, size):

             curr = i
             swapped = True

             while left_of(curr) < size and swapped:
                 swapped = False
                 max_child_i = max_child(array,curr,size)
                 if array[curr] < array[max_child_i]:
                     swapped = True
                     array[curr],array[max_child_i] = array[max_child_i], array[curr]
                 curr = max_child_i

         def left_of(index):
             return 2*index + 1

         def right_of(index):
             return 2*index + 2

         def max_child(array, index, heap_size):
             left_child = left_of(index)
             right_child = right_of(index)
             if right_child >= heap_size:
                 return left_child
             if array[left_child] >= array[right_child]:
                 return left_child
             else:
                 return right_child
```

```
In [14]: result = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]
         max_heapify(result, 1, len(result))
         print(result)
         assert result == [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
```

```
[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
```

```
In [15]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

```
In [2]: def build_max_heap(array):
            ###
            ### YOUR CODE HERE
            ###
            size = len(array)
            mid = size//2
            for i in range(mid,-1,-1):
                 max_heapify(array, i, size)

        def max_heapify(array, i, size):

            curr = i
            swapped = True

            while left_of(curr) < size and swapped:
                swapped = False
                max_child_i = max_child(array,curr,size)
                if array[curr] < array[max_child_i]:
                    swapped = True
                    array[curr],array[max_child_i] = array[max_child_i], array[curr]
                curr = max_child_i

        def left_of(index):
            return 2*index + 1

        def right_of(index):
            return 2*index + 2

        def max_child(array, index, heap_size):

            left_child = left_of(index)
            print(f"left_child : {left_child}")

            right_child = right_of(index)
            if right_child >= heap_size:
                return left_child
            elif array[left_child] >= array[right_child]:
                return left_child
            else:
                return right_child
```

```
In [3]: array = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]

        build_max_heap(array)
        print(array)
        assert array == [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
```

```
left_child : 9
left_child : 7
left_child : 5
left_child : 3
left_child : 9
left_child : 1
left_child : 3
left_child : 7
[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
```

```
In [4]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

CS3. *Heapsort:* Implement heapsort algorithm following the pseudocode in Binary Heap and Heapsort.

```
In [11]: def heapsort(array):
             build_max_heap(array)
             heap_end_pos = len(array)-1
             while heap_end_pos > 0:
                 print(array)
                 array[0],array[heap_end_pos] = array[heap_end_pos],array[0]
                 heap_end_pos -= 1
                 max_heapify(array,0,heap_end_pos+1)



         def build_max_heap(array):
             ###
             ### YOUR CODE HERE
             ###
             size = len(array)
             mid = size//2
             for i in range(mid,-1,-1):
                  max_heapify(array, i, size)

         def max_heapify(array, i, size):

             curr = i
             swapped = True

             while left_of(curr) < size and swapped:
                 swapped = False
                 max_child_i = max_child(array,curr,size)
                 if array[curr] < array[max_child_i]:
                     swapped = True
                     array[curr],array[max_child_i] = array[max_child_i], array[curr]
                 curr = max_child_i
             return array
         def left_of(index):
             return 2*index + 1

         def right_of(index):
             return 2*index + 2

         def max_child(array, index, heap_size):

             left_child = left_of(index)
             right_child = right_of(index)
             if right_child >= heap_size:
                 return left_child
             elif array[left_child] >= array[right_child]:
                 return left_child
             else:
                 return right_child
```

```
In [12]: array = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
         heapsort(array)
         print(array)
         assert array == [1, 2, 3, 4, 7, 8, 9, 10, 14, 16]
```

```
[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
[14, 8, 10, 4, 7, 9, 3, 2, 1, 16]
[10, 8, 9, 4, 7, 1, 3, 2, 14, 16]
[9, 8, 3, 4, 7, 1, 2, 10, 14, 16]
[8, 7, 3, 4, 2, 1, 9, 10, 14, 16]
[7, 4, 3, 1, 2, 8, 9, 10, 14, 16]
[4, 2, 3, 1, 7, 8, 9, 10, 14, 16]
[3, 2, 1, 4, 7, 8, 9, 10, 14, 16]
[2, 1, 3, 4, 7, 8, 9, 10, 14, 16]
[1, 2, 3, 4, 7, 8, 9, 10, 14, 16]
```

```
In [7]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

CS4. Measure computational time of Python's built-in sort function by filling the template below. Hint: - You will need the function `gen_random_int()` from Week 01 Problem Set. - Use `sorted(list)` function of Python's list See Python's Sorting HOW TO Documentation.

```
In [8]: import time
        import random

        def run_function(f, x):
            start = time.time()
```

```
    f(x)
    end = time.time()
    return end-start

def gen_random_int(number, seed):
    random.seed(seed)
    sorted_list = [i for i in range(number)]
    random.shuffle(sorted_list)

    return sorted_list

# def heapsort(array):
#     n = len(array) - 1
#     build_max_heap(array)
#     while n > 0:
#         array[0],array[n] = array[n],array[0]
#         n -=1
#         array[:n+1] = max_heapify(array[:n+1],0,n+1)

def heapsort(array):
    build_max_heap(array)
    heap_end_pos = len(array)-1
    while heap_end_pos > 0:
        array[0],array[heap_end_pos] = array[heap_end_pos],array[0]
        heap_end_pos -= 1
        max_heapify(array,0,heap_end_pos+1)

time_builtin = []
vic_list = []
# set the maximum power for 10^power number of inputs
maxpower = 1000

for n in range(1, maxpower + 1):
    # create array for 10^1, 10^2, etc
    # use seed 100
    array = gen_random_int(n, 9999999999999999999999999999999999999999999999999999999
    result = run_function(heapsort, array)
    # call run_function with sorted() and array as arguments
    # result = run_function(None, None)

    ###
    ### YOUR CODE HERE
    ###

    time_builtin.append(result)

# for n in range(1, maxpower + 1):
#     # create array for 10^1, 10^2, etc
```

```
        #     # use seed 100
        #     array = gen_random_int(n, 999999999999999999999999999999999999999999999999999999999
        #     test = run_function(heapsort_vic, array)

        #     vic_list.append(test)

        print(sum(time_builtin))
        print(f"vic : {sum(vic_list)}")
```

```
2.479708671569824
vic : 0
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Week01_Homework

October 25, 2021

1 Week 1 Problem Set

1.1 Homework

HW0. Write a function `palindrome(s)`to check if the string `s` is a Palindrome. Use iteration for this solution.

```
In [2]: def palindrome(s):
            return s == s[::-1]
```

```
In [3]: assert not palindrome("moon")
        assert palindrome("noon")
        assert palindrome("a a")
        assert palindrome("ada")
        assert not palindrome("ad a")
```

HW1. Create a function to implement **Bubble Sort Algorithm version 2**. You can check the pseudocode in Bubble Sort and Insertion Sort.

The function should …

- **modify/mutate** an existing array into a sorted one
- **return** the number of comparisons made

Hint: To count the number of comparisons made you can simply increment an integer variable (counter) right before the `if` statement.

```
In [7]: def bubble_sort(arr):
            n = len(arr)
            count = 0
            list_sorted = False
            while list_sorted is False:
                list_sorted = True
                for i in range(1, n):
                    first_number = arr[i-1]
                    second_number = arr[i]
                    count +=1
                    if first_number > second_number:
                        arr[i], arr[i-1] = swap(arr[i-1], arr[i])
                        list_sorted = False
            return count


        def swap(left, right):
            temp = left
            left = right
            right = temp
            return right, left
```

```
In [8]: array = [10,3,8,47,1,0,-39,8,4,7,6,-5]
        count = bubble_sort(array)
        print(array, count)
        assert array == [-39, -5, 0, 1, 3, 4, 6, 7, 8, 8, 10, 47]
        assert count == 121
```

```
[-39, -5, 0, 1, 3, 4, 6, 7, 8, 8, 10, 47] 121
```

```
In [3]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

HW2. The solution in HW1 can be improved!

The n-th pass places the n-th largest element into its final place (i.e. after 1 cycle, the (1st) largest element will be in its correct position).

Hence, we can reduce subsequent iterations by only considering the other (unsorted) `n-1` elements in the array.

Implement the algorithm of **Bubble Sort version 4** under Bubble Sort and Insertion Sort.

```
In [15]: def bubble_sort(arr):
             n = len(arr)
             count = 0
             list_sorted = False
             while not list_sorted:
                 list_sorted = True
                 new_n = 0
                 for i in range(1, n):
                     first_number = arr[i-1]
                     second_number = arr[i]
                     count +=1
                     if first_number > second_number:
                         arr[i], arr[i-1] = swap(arr[i-1], arr[i])
                         list_sorted = False
                         new_n = i  # what is this doing?
                 n = new_n  # what is this doing?
             return count

         def swap(left, right):
             temp = left
             left = right
             right = temp
             return right, left
```

```
In [16]: array = [10,3,8,47,1,0,-39,8,4,7,6,-5]
         count = bubble_sort(array)
         print(array, count)
         assert array == [-39, -5, 0, 1, 3, 4, 6, 7, 8, 8, 10, 47]
         assert count == 66
```

```
[-39, -5, 0, 1, 3, 4, 6, 7, 8, 8, 10, 47] 66
```

```
In [6]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

Week01_Cohort

October 25, 2021

1 Week 1 Problem Set

1.1 Cohort Problems

CS0. Implement factorial problem using iteration. The function should takes in an Integer input and returns and Integer output which is the factorial of the input. Recall that:

$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$

You should consider the case when n is zero and one as well.

```
In [3]: # iteration
        def factorial_iteration(n):
            if n == 0 or n == 1 :
                return 1
            return  n * factorial_iteration(n-1)
```

```
In [4]: assert factorial_iteration(0) == 1
        assert factorial_iteration(1) == 1
        assert factorial_iteration(5) == 120
        assert factorial_iteration(7) == 5040
        assert factorial_iteration(11) == 39916800
```

```
In [3]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

CS1. Create a function that implements **Bubble Sort version 1** (from your Notes) to sort an array of integers. The function should sort the input array in place. Refer to Bubble Sort and Insertion Sort for the pseudocode.

```
In [5]: def bubble_sort(arr):
            count = 0
            n = len(arr)
            for i in range(1, n):
                for j in range(1, n):
                    first_number = arr[j-1]
                    second_number = arr[j]
                    count +=1
                    if first_number > second_number:
```

```
                arr[j] = first_number
                arr[j-1] = second_number
    return arr
```

```
In [6]: array = [10,3,8,47,1,0,-39,8,4,7,6,-5]
        bubble_sort(array)
        print(array)
        assert array == [-39, -5, 0, 1, 3, 4, 6, 7, 8, 8, 10, 47]
```

```
[-39, -5, 0, 1, 3, 4, 6, 7, 8, 8, 10, 47]
```

```
In [7]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

CS2. Modify CS1, so that it returns the number of comparisons that are performed.

Hint: To count the number of comparisons, you can create an integer variable which you *increment* just before the `if` statement.

```
In [10]: def bubble_sort(arr):
             count = 0
             n = len(arr)
             for i in range(1, n):
                 for j in range(1, n):
                     first_number = arr[j-1]
                     second_number = arr[j]
                     count +=1

                     if first_number > second_number:
                         arr[j] = first_number
                         arr[j-1] = second_number
             return count
```

```
In [11]: array = [10,3,8,47,1,0,-39,8,4,7,6,-5]
         count = bubble_sort(array)
         print(array, count)
         assert array == [-39, -5, 0, 1, 3, 4, 6, 7, 8, 8, 10, 47]
         assert count == 121
```

```
[-39, -5, 0, 1, 3, 4, 6, 7, 8, 8, 10, 47] 121
```

```
In [12]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

CS3. Create a function that implements **Insertion Sort** to sort an array of floats. The function should sort the input array in place. Refer to Bubble Sort and Insertion Sort for the pseudocode.

```
In [15]: def insertion_sort(arr):
             n = len(arr)
             for i in range(1, n):
                 curr = i
                 while arr[curr-1] > arr[curr] and curr > 0:
                     arr[curr], arr[curr-1] = swap(arr[curr-1], arr[curr])
                     curr -= 1
             return arr

         def swap(left, right):
             temp = left
             left = right
             right = temp

             return right, left
```

```
In [16]: array = [10.3,3.8,8.4,47.1,1.0,0,-39.8,8.4,4.7,7.6,-6.5,-5.0]
         insertion_sort(array)
         print(array)
         assert array == [-39.8, -6.5, -5.0, 0.0, 1.0, 3.8, 4.7, 7.6, 8.4, 8.4, 10.3, 47.1]
```

```
[-39.8, -6.5, -5.0, 0, 1.0, 3.8, 4.7, 7.6, 8.4, 8.4, 10.3, 47.1]
```

```
In [17]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```

CS4. Write a function `gen_random_int` that generates a list of integers from `0` to `n - 1` with its sequence randomly shuffled. The function should take in an integer `n`, denoting the number of integers to be generated.

Hint: - You can use `random.shuffle(mylist)` to shuffle the elements of a list called `mylist`. - Refer to Python's Random module for more details on how to use `shuffle()`.

```
In [28]: import random

         def gen_random_int(number, seed):
             random.seed(seed)
             sorted_list = [i for i in range(number)]
             random.shuffle(sorted_list)

             return sorted_list
```

```
In [30]: output = gen_random_int(10, 100)
         print(output)
         assert output == [4, 0, 5, 9, 3, 1, 6, 8, 7, 2]
```

```
[4, 0, 5, 9, 3, 1, 6, 8, 7, 2]
```

```
In [12]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###
```