fullName: _____ andrewID: _____ section: ____

15-112 F26 Midterm2 Version B
Time: 80 Minutes

You must write your name on this paper and hand this back in immediately after the assessment. If we do not receive it immediately, you will receive a zero on the assessment. Do not unstaple any pages. All pages must be handed in intact.

Do not use your own scrap paper. You should not need it, but if you must absolutely have scrap paper, raise your hand and we will provide some. Write your andrewID clearly on it and hand it in with your quiz. We will not grade anything on scrap paper.

You may not ask questions during the exam, except for English-language clarification questions. If you are unsure about a problem, take your best guess.

Before and during the exam, you may not view any other notes, prior work, websites or resources, including any form of AI. You may not use calculators, phones, laptops, or any other devices. You may not communicate with anyone else except for current 112 TAs or faculty during the exam. All syllabus policies apply.

You may not discuss this exam with anyone else, even briefly, in any form, until we have released grades. Failure to abide by these rules may result in an academic integrity violation.

Do not use eval() or exec().

For all problems, assume that the required imports (such as 'import math') are present.

You may not leave the exam room during the final 15 minutes of this exam. If you finish early, there are ample bonus problems to keep you gainfully occupied.

Do not open this or look inside (even briefly) before you are ready to begin. Do not spend more than the specified time noted above on this assessment.

Code Tracing [15 pts, 3 pts each]

Indicate what the following code prints. Place your answer (and nothing else) in the box below. If a line of code crashes, just print "crash" (without quotes) and stop the CT at that point.

```
# CT1:

def ct1(L):
    M = [v - 1 for v in L]
    s = set(M) - set(L)
    t = set()
    for v in s:
        for c in str(v):
            t.add(c)
    s.add(int(''.join(sorted(t))))
    return s

print(ct1([23, 25, 24, 38, 24]))
```

```
# CT2:

def ct2(d):
    result = dict()
    for k in d:
        x = d[k]
        if not isinstance(x, int):
            x = len(str(x))
        result[x] = result.get(x, set()) | {k}
    return result

print(ct2({1: 3, 'five': 7, 5: 'six'}))
```

```
# CT3:

def ct3(L, k=1):
    if len(L) == 0:
        return L
    else:
        firstK = L[:k]
        rest = L[k:]
        return ct3(rest, k+1) + [sum(firstK)]

print(ct3([7, 6, 5, 4, 3, 2]))
```



```
# CT4:

def ct4(t, r=1):
    if t.isLeaf():
        return r * t.value
    else:
        result = 0
        for child in t.children:
            result += ct4(child, r * 10)
        return result

t = Tree(1,
         Tree(2,
              Tree(3),
              Tree(5)
              ),
         Tree(7),
         )

print(ct4(t))
```



```
# CT5:

# The class F is used by ct5:
class F:
    def __init__(self, x, y=1):
        self.x = x
        self.y = y

    def __add__(self, v):
        if isinstance(v, int):
            return F(self.x + v, self.y + v)
        else:
            return F(self.x + v.x, self.y + v.y)

    def __repr__(self):
        return f'F{self.x}{self.y}'

def ct5(a):
    b = F(4, 5)
    c = a + b
    d = c + 1
    return [a, b, c, d]

print(ct5(F(2)))
```



True or False [10 pts, 1 pt each]
Indicate True or False for each of the following. Indicate your answer by filling in the dot.

Notes: True means "always True". If a statement is only sometimes True, then it is False.
Assume L and M are both lists with N integers (where N > 0).
Assume D is a dict with N keys (where N > 0).

1. Binary search won't crash on an unsorted list L, but it may return the wrong answer.

○ True

○ False

2. If S is a set and (v in S) is True, then v must be immutable.

○ True

○ False

3. If sorted(L) == sorted(M), then len(set(L)) == len(set(M)).

○ True

○ False

4. In Python, (v in L) uses binary search if L is sorted and linear search if L is unsorted.

○ True

○ False

5. A tree can contain a tree (as a node's value), but a set cannot contain a set.

○ True

○ False

6. Python's built-in sort runs in O(N).

○ True

○ False

7. To enable the < operator to work properly with instances of a class, we can define the __lt__ method in that class.

○ True

○ False

8. If f(L) runs in O(N) and g(L) also runs in O(N), then it is still possible for f(L) to run a million times slower than g(L) for any list L.

○ True

○ False

9. D.get(k, 0) runs in O(1) for any integer value of k.

○ True

○ False

10. If a function f(L) runs in O(1234), then it also runs in O(1).

○ True

○ False

Short Answers [5 pts, 1 pt each]
Write your answers in each box.
Note: All answers in this section must be integers and not expressions.

SA1: When running merge sort over a list of 64 integers (and remember to give your answer as an integer)....

A) How many steps per pass are required (where a step is either a compare or a copy)?

B) How many total passes are required?

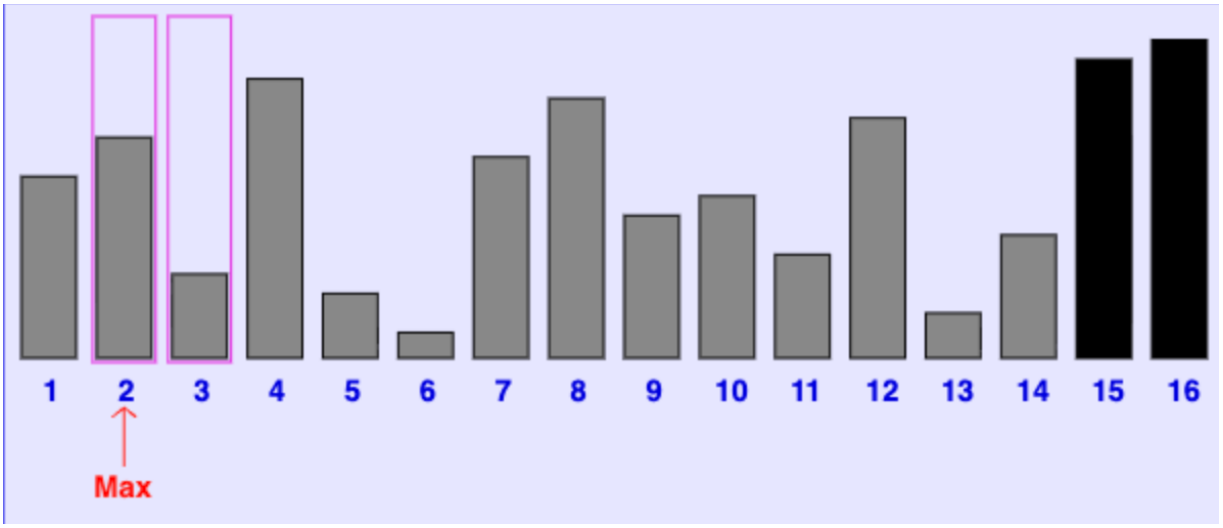
SA2. If selection sort takes 3 seconds to sort 500 values, how long (to the nearest second) would we expect selection sort to take to sort 1500 values on the same computer?

SA3: If merge sort requires 30 passes to sort a list L, how many passes would merge sort require to sort the list M = L * 4?

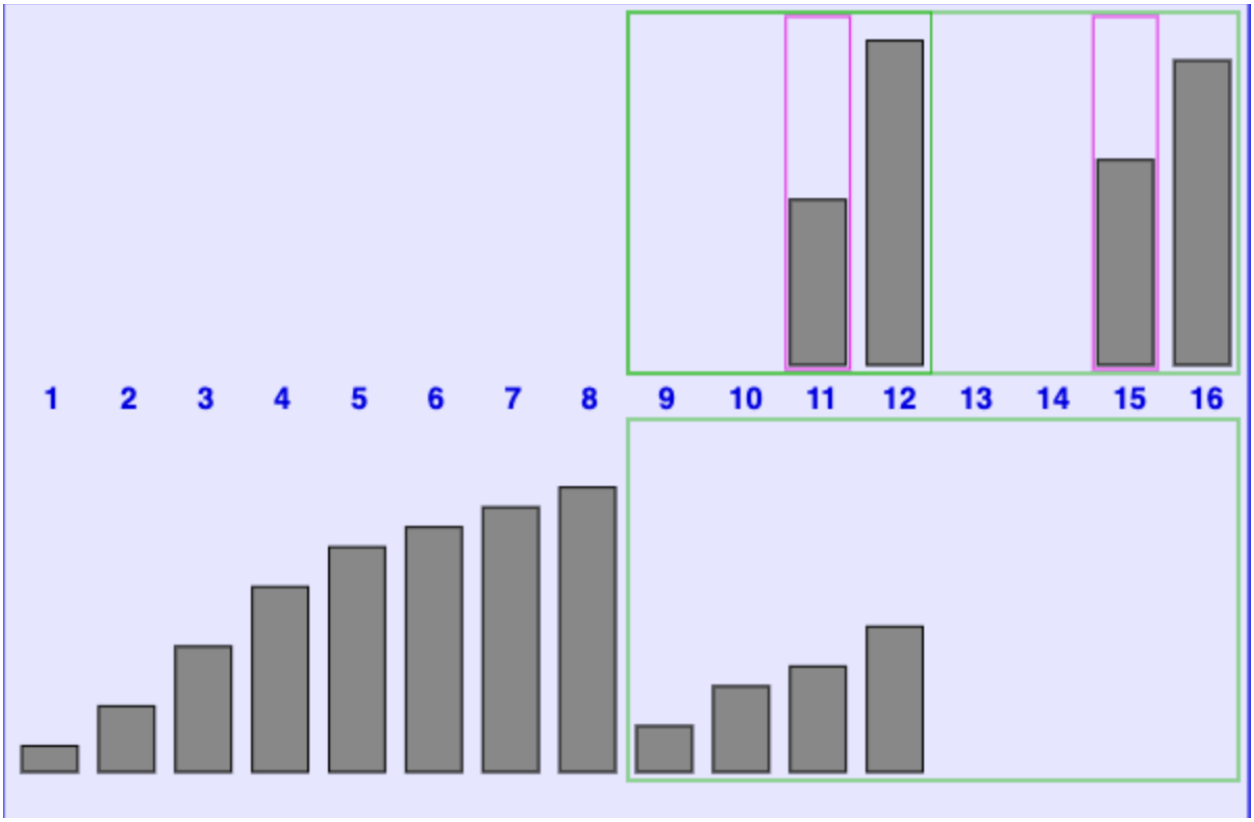
For SA4 and SA5:

- Note that the indexes in xSortLab start at 1 and not 0.

SA4. The following is a snapshot of xSortLab running selection sort. What are the indexes of the next two values to be swapped?



SA5. The following is a snapshot of xSortLab running merge sort. What are the indexes of the next two values to be compared?



Big O [10 pts, 2 pts each]

What is the worst-case Big O runtime of each of the following?
Place your answer (and nothing else) in the box below.

All answers must be simplified (e.g. O(n) instead of O(n+5)).
Assume L is a list with N integers.
Assume N is a positive integer.

 # BigO1:

```
d = dict()
for v in L:
    d[v] = d.get(v, 0) + max(L)
```

 # BigO2:

```
r = 0
for i in range(2, 10):
    j = len(L) // i
    r += sum(L[j : j+10])
```

 # BigO3:

```
M = L * len(L)
for i in range(len(L)):
    print(M.count(i))
```

 # BigO4:

```
M = copy.copy(L)
i = 1
while i < len(L):
    x = M.pop()
    M.insert(0, x + i)
    i *= 2
```



 # BigO5:

```
M = sorted(L)
s = set()
for x in M:
    for y in L:
        s.add(x * y)
```



FR1: Turtle and FixedTurtle classes [20 pts]

Part 1: The Turtle Class

Write the Turtle class so the following test cases work. Do not hardcode the test cases.

```
# A turtle can be at any integer x location, and it can be facing
# either left or right.
t1 = Turtle(2)
assert(str([t1]) == '[Turtle(at x=2, facing right)]')

t1.move(5)
assert(str([t1]) == '[Turtle(at x=7, facing right)]')
t1.turnAround()
assert(str([t1]) == '[Turtle(at x=7, facing left)]')
t1.move(6)
assert(str([t1]) == '[Turtle(at x=1, facing left)]')
t1.turnAround()
assert(str([t1]) == '[Turtle(at x=1, facing right)]')

# two turtles are equal if they are at the same position,
# facing the same direction
t2 = Turtle(1)
assert(str([t2]) == '[Turtle(at x=1, facing right)]')
assert(t1 == t2)
t2.turnAround()
assert(str([t2]) == '[Turtle(at x=1, facing left)]')
assert(t1 != t2) # they are facing different directions
assert(t1 != 'do not crash here')

# spawn creates a new Turtle, facing the same direction, but
# moved by the given amount:
t3 = t2.spawn(1)
assert(str([t3]) == '[Turtle(at x=0, facing left)]')

# you need to keep a count of turtle instances
assert(Turtle.instanceCount == 3)

# turtles can work with sets
s = set()
assert(Turtle(2) not in s)
s.add(Turtle(2))
assert(Turtle(2) in s)
```

Part 2: The FixedTurtle Class

Write the FixedTurtle class so the following test cases work. Do not hardcode the test cases.

The FixedTurtle class uses the Turtle class you just wrote. For this part, we will assume your Turtle class works properly.

Also, your FixedTurtle class must use OOP properly for full credit (for example, do not include any unnecessary methods).

```
# A FixedTurtle cannot move (but it can turnaround).

t4 = FixedTurtle(1)
assert(isinstance(t4, Turtle))
assert(str([t4]) == '[FixedTurtle(at x=1, facing right)]')
assert(t4.move(3) == 'sorry')
assert(str([t4]) == '[FixedTurtle(at x=1, facing right)]')
```

Write your solution on the next page.

FR2: getLock [20 pts]

Notes:

- For this problem you must use backtracking properly to receive credit (even if there are other ways to solve the problem).
- You must use recursion here, but you are also free to use iteration (loops) if it helps.

Background: given a non-empty string s containing only uppercase letters, we will say that a string t is a "lock" of s (a coined term) if:

- t contains every unique letter in s exactly once, and
- no substring of t of length 2 occurs in s

For example, say s = 'ABCDADBA'.

- The unique letters in s are 'ABCD', so any lock t must be some ordering of exactly those characters.
- We see that t = 'ABCD' is NOT a lock because 'AB' is in t and 'AB' is also in s.
- Also, t = 'BDCA' is a lock because:
    - 'BD' is not in s,
    - 'DC' is not in s, and
    - 'CA' is not in s.
- Similarly, t = 'ACBD' is a lock because:
    - 'AC' is not in s,
    - 'CB' is not in s, and
    - 'BD' is not in s.
- In fact, these are the only two legal locks of s.

Thus, all the locks of s are: {'BDCA', 'ACBD'}

Note that it is possible that a string s has no legal locks. For example, if s = 'ABA', neither 'AB' nor 'BA' are legal locks, so it has no legal locks.

With that, write the function getLock(s) that takes a string s and, using backtracking, returns a legal lock of the string s, or None if no legal locks exist. If more than one lock exists, you can return any one of them. Thus, for the string s above:

```
assert(getLock(s) in {'BDCA', 'ACBD'})
```

For some solution approaches, it may be helpful to use a wrapper function here, though that is not required.

FR3: getLeafPaths [20 pts]

Note: for this problem, you may use iteration (for or while loops), but you MUST use recursion in a meaningful way to visit each node in the tree.

Background: we will say that the "path" to a node in a tree is a hyphenated string of all the values from the root to that node. For example, this tree includes comments showing the path to each node in the tree:

```
t0 = Tree('A',              # 'A'
          Tree('B'),        # 'A-B'
          Tree('C',         # 'A-C'
               Tree('D'),   # 'A-C-D'
               Tree('B'))) # 'A-C-B'
```

With that, write the function getLeafPaths(t) that takes a tree t and returns a dict mapping the value of each leaf in t to a set of all the paths to a leaf with that value. Assume all the values are immutable. And, again, your solution must use recursion properly.

For example, here is the same tree from above, but now we only include paths to leaves:

```
t0 = Tree('A',
          Tree('B'),        # 'A-B'
          Tree('C',
               Tree('D'),   # 'A-C-D'
               Tree('B'))) # 'A-C-B'
```

From this we see:

- There are 2 leaves with the value 'B', with these paths:
  - 'A-B'
  - 'A-C-B'
- There is 1 leaf with the value 'D', with this path:
  - 'A-C-D'

And so:

```
assert(getLeafPaths(t0) == { 'B': { 'A-B', 'A-C-B' },
                             'D': { 'A-C-D' },
                           })
```

Here is another example:

```
t1 = Tree(6,
          Tree(4,
               Tree(3),       # '6-4-3'
               Tree(2)),      # '6-4-2'
          Tree(5,
               Tree(3,
                    Tree(2),   # '6-5-3-2'
                    Tree(1))), # '6-5-3-1'
          Tree(3))            # '6-3'

assert(getLeafPaths(t1) == { 1: { '6-5-3-1' },
                             2: { '6-4-2', '6-5-3-2' },
                             3: { '6-4-3', '6-3' }
                           })
```

And one last example:

```
t3 = Tree('Z') # The root is a leaf here, with the path 'Z'

assert(getLeafPaths(t3) == { 'Z' : { 'Z' } })
```

For some solution approaches, it may be helpful to use a wrapper function here, though that is not required.

Write your solution on the next 2 pages.

BonusCT [2 pts, 1 pt each]
These CTs are optional, and intended to be very challenging. They are worth very few points. Indicate what the following code prints. Place your answers (and nothing else) in the boxes below. If a line of code crashes, just print "crash" (without quotes) and stop the CT at that point.

bonusCT1:

```
def bonusCt1(n):
    def f(n): return n if n < 2 else 2*n-1+f(n-1)
    def g(n): return n if n < 2 else 2*g(n-1)
    def h(n): return g(n)/f(n) == round(g(n)/f(n))
    def i(n): return f(n) if h(n) else i(n+1)
    return n + i(n)

print(bonusCt1(3))
```

bonusCT2:

```
def bonusCt2(x):
    def f(g, x): return lambda y: g(x*y)
    def g(n): return 10+n
    return f(f(g, x),x+2)(x+3)

print(bonusCt2(2))
```