fullName:______________________ andrewID:________________ section:___

# 15-112 S25
# Midterm2 version B

Read these instructions carefully before starting:

1. Exam versions are color-coded. You must have a different version (color) of this exam than the students sitting to your left and right.
2. Stop writing and submit the entire exam when instructed by the proctor.
   - **Do not unstaple any pages.**
   - You must submit the entire exam with all pages intact.
3. Do not discuss the exam with anyone else until we announce the grades on Ed. Especially do not discuss with or even near anyone who has not taken the exam, in-person, online, or otherwise.
   - This applies to everyone, including students in either lecture.
4. Do not use your own scrap paper.
   - You should not need scrap paper, there is plenty of room for you on the exam.
   - However, if you absolutely must use scrap paper, raise your hand and we will provide some. Then, you must write your andrew id clearly on the scrap paper, and hand in the scrap paper with your paper exam. We will not grade anything on your scrap paper.
5. You may not ask questions during the exam.
   - The one exception is for English-language clarifications.
   - If you are unsure how to interpret a problem, just take your best guess.
6. Do not use any concepts (including built-in functions) not covered in the notes through week 12 or beyond unit 8 (Object Oriented Programming).
7. Do not hardcode your solutions.
   - We may test your code using additional test cases.
   - Hardcoding will receive zero points.
8. Assume almostEqual(x, y) and rounded(n) are both supplied for you.
   - You must write all other helper functions you wish to use, unless we specify otherwise.
9. Good luck!

**You may use this page to show your work (but we will not grade anything on this page, and will only grade answers in the designated answer boxes)**

**Code Tracing (CT) [10 pts, 2.5 pts each]**
For each CT, indicate what the code prints.
Place your answer (and nothing else) in the box below the code.

**CT1:**

```
#Hint: Draw a box and arrow diagram!
def ct1(L):
    A = [set()]
    for item in L:
        if item in A[0]:
            A.insert(0, {item})
        else:
            A[0].add(item)
    return A

print(ct1([1, 2, 2, 4, 3, 4, 5]))
```

**CT2:**

```
def ct2(d1, d2):
    result = dict()
    for key in list(d1.keys()) + list(d2.keys()):
        x = key*2
        if x in result:
            result[x] *= 2
        else:
            result[x] = d1.get(key, 'k') * d2.get(key, 2)
    return result

print(ct2({1:2, 'b':4}, {1:'a', 2:1}))
```

**CT3:**

```
def ct3(L, d=0):
    if len(L) == 0:
        return [ ]
    else:
        i = len(L)//2
        left = L[:i]
        right = L[i:]
        return [sum(right) * 10**d] + ct3(left, d+1)

print(ct3([1, 2, 3, 4, 5, 6]))
```

**CT4:**

```
def ct4(t):
    if t.isLeaf():
        return t.value
    else:
        return t.value * max([ct4(child) for child in
                              t.children])

t = Tree(2,
         Tree(3,
              Tree(4),
              Tree(5)),
         Tree(6),
         Tree(7,
              Tree(1),
              Tree(2)))

print(ct4(t))
```

**True or False (T/F) [10 pts, 1 pt each]**
For each of the following, bubble in True or False (do not bubble in both!). Fill in the bubble completely.
Assume L and M are both lists with N integers (where N > 0).
Assume S is a set with N integers (where N > 0).

**TF1.** Sets can contain sets.

○True ○False

**TF2.** For a dictionary d and a key k, if (d[k] == d) is True, then d[d[k]] will not crash and will evaluate to d.

○True ○False

**TF3.** When a value v is added to a set s, Python first calls hash(v) and determines which bucket to use in the hashtable representing s.

○True ○False

**TF4.** L.append(L) runs in O(1).

○True ○False

**TF5.** If set(L) == set(M), then set(L + M) == set(M).

○True ○False

**TF6.** S.add(S.pop()) will always crash.

○True ○False

**TF7.** Every non-empty tree contains exactly one root and at least one leaf.

○True ○False

**TF8.** Some recursive functions cannot be rewritten to be non-recursive.

○True ○False

**TF9.** When we define a class, if we do not define the __repr__ method, then Python will crash if we try to print an instance of the class.

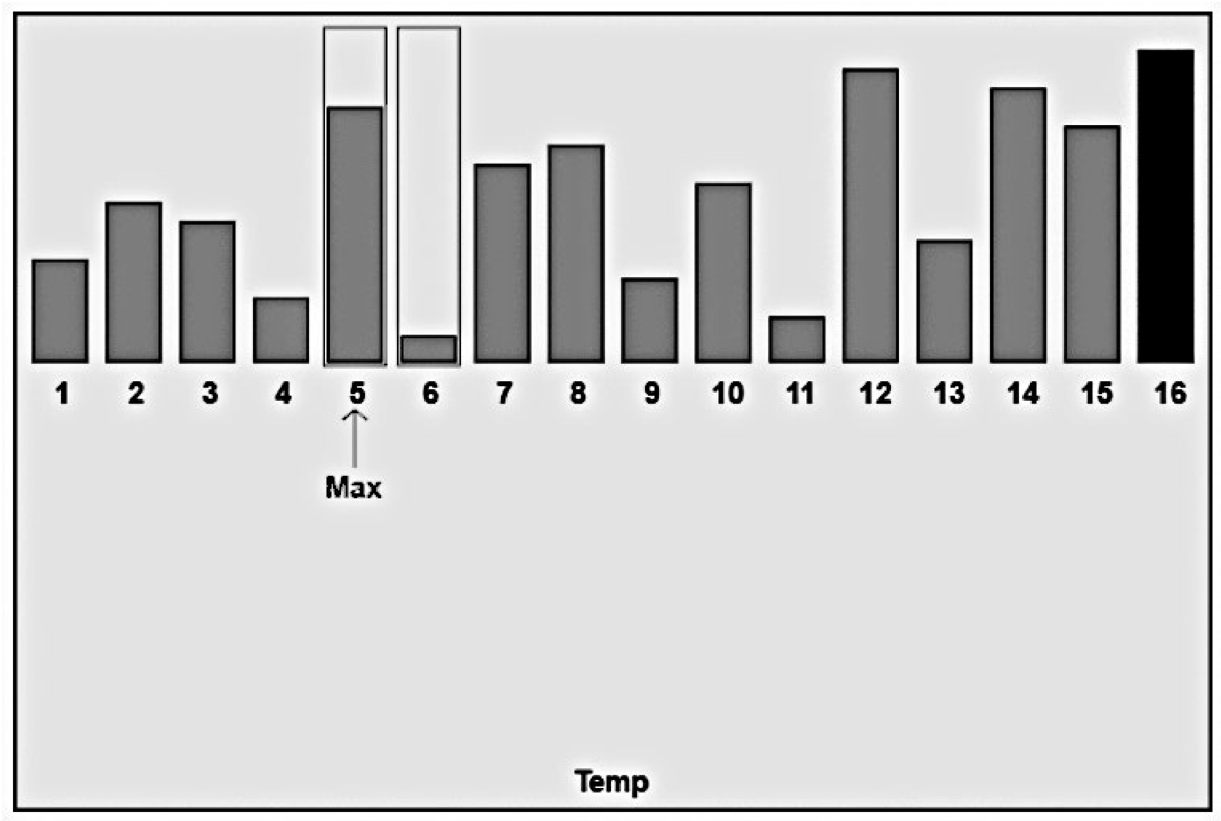
○True ○False

**TF10.** If f(L) runs in O(NlogN) and g(L) runs in O(N**2), where N = len(L), then calling f(L) will run faster than calling g(L) for any list L.

○True ○False

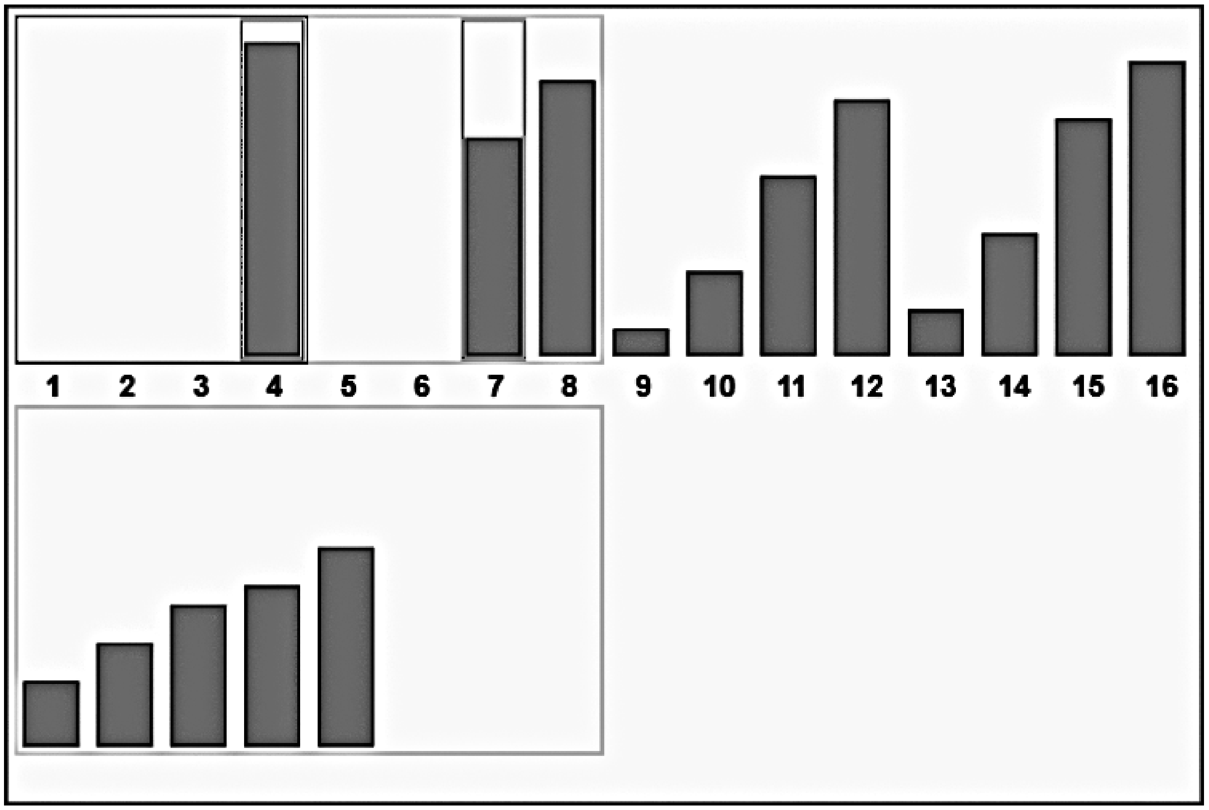
**Short Answer (SA) [5 pts, 1 pts each]**
The following questions focus on searching, sorting, and hashing. Write your answer (and only your answer) in the corresponding box below.

**1.** The following image is from selection sort in xSortLab:



What are the indexes of the next two values to be swapped? (Use the indexes in the image even though they start at 1.)

**2.** The following image is from merge sort in xSortLab:



What is the index of the next value to be copied to the temporary list? (Use the indexes in the image even though they start at 1.)

**3.** Note: For this problem, do not use Big O notation. That is, you should include constants where appropriate in your answers. With that, when running merge sort over a list of N integers, where you may assume that N is a power of 2...

**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?

**4.** If selection sort takes 10 seconds to sort 200 values, how long (to the nearest second) would we expect selection sort to take to sort 600 values on the same computer?

**5.** Note: for the next question:

- you may assume that 2**10 == 1,000, and
- we will accept answers within 2 of the correct answer.

For a sorted list with 2 billion values, about how many comparisons would be required in the worst case for binary search?

**Big O [10pts, 2pts each]**

What is the worst-case Big O runtime of each of the following? All answers must be simplified (e.g. O(n) instead of O(n+5) ).

Assume L is a list with N integers.

Assume S is a set with N integers.

Assume N is a positive integer.

**Big O 1:**

```
x = y = 1
while x < len(L):
    y += 1
    x *= 10
while y > 0:
    y -= 1
    x *= 10
```

**Big O 2:**

```
while len(L) > 1:
    x = L.pop(0)
    x += L.pop(0)
    L.append(x)
```

**Big O 3:**

```
d = dict()
for v in L:
    d[v] = sorted(L + [v])
```

**Big O 4:**

```
M = [ v**2 for v in range(10**6) ]
t = set()
u = set()
for v in M:
    if v in t:
        u.add(v)
    t.add(v)
print(u)
```

**Big O 5:**

```
def f(t):
    if t.isLeaf():
        return 1
    else:
        return f(t.children[0]) + 1

print(f(T))
# Assume that T is a tree with N nodes, where every
# non-leaf has exactly two children. Try drawing a tree
# like this. When N is doubled, roughly how many more
# recursive calls will be made?
```

**Free Response / FR1: interiorNodeCount(tree) [10 pts]**

Background: a node in a tree is an "interior" node if it is not a leaf.

With that, write the function `interiorNodeCount(tree)` that takes a tree and returns the number of interior (non-leaf) nodes in the tree.

For example:

```
    t1 = Tree(1,
              Tree(2),
              Tree(3,
                   Tree(4)))
    assert(interiorNodeCount(t1) == 2) # 1 and 3

    t2 = Tree(1,
              Tree(2),
              Tree(3))
    assert(interiorNodeCount(t2) == 1) # 1

    t3 = Tree(1)
    assert(interiorNodeCount(t3) == 0)
```

**Begin your FR1 answer here or on the next page.**

**Begin or continue your answer to FR1 here:**

**Free Response / FR2: squaresInRange(lo, hi) [10 pts]**

Note: for this problem, you must not use iteration (for or while loops).

Without using iteration, write the function `squaresInRange(lo, hi)` that takes two positive integers lo and hi, where lo <= hi, and returns a list of all the perfect squares in the range [lo, hi] (inclusive on both ends).

For example:

```
    assert(squaresInRange(25, 49) == [25, 36, 49])
    assert(squaresInRange(25, 48) == [25, 36])
    assert(squaresInRange(25, 25) == [25])
    assert(squaresInRange(29, 30) == [ ])
```

**Begin your FR2 answer here or on the next page.**

**Begin or continue your answer to FR2 here:**

**Free Response / FR3: getQuizAverages(scores) [15 pts]**

Background: we can represent student scores on quizzes in a dict like so:
```
    scores = {
        'Ann': { 'quiz1': 90, 'quiz2': 80, 'quiz3': 85 },
        'Ben': { 'quiz1': 70, 'quiz3': 95 },
        'Cam': { 'quiz2': 90 },
        'Del': dict(),
    }
```

In this example, we see that:

- Ann scored 90 on quiz1, 80 on quiz2 and 85 on quiz3
- Ben scored 70 on quiz1, 95 on quiz3, and did not take quiz2
- Cam scored 90 on quiz2, and did not take quiz1 or quiz3
- Del did not take any quiz

With that, write the function `getQuizAverages(scores)` that takes a dict of student scores as above and returns a dict mapping each quiz that appears in the scores to its average **(rounded to the nearest integer).**

For example, with the scores above:

- The quiz1 scores are 90 and 70, which average to 80
- The quiz2 scores are 80 and 90, which average to 85
- The quiz3 scores are 85 and 95, which average to 90

Thus, for the scores above:
```
    assert(getQuizAverages(scores) == { 'quiz1': 80,
                                         'quiz2': 85,
                                         'quiz3': 90})
```

You may assume that there is at least one quiz in the scores dict.

**Begin your FR3 answer on the next page.**

**Begin your answer to F3 here:**

**You may continue your answer to FR3 here:**

**You may continue your answer to FR3 here:**

**Free Response / FR4: backtracking: makeSinglish(L) [15 pts]**

Note: 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 if it helps.

Background: give a list L of uppercase words, we will say that L is "singlish" (a coined term) if each word in L (after the first word) shares exactly one letter with the previous word.

For example:

```
    L1 = ['ABC', 'AEG', 'DEE', 'ZD']
```

This list is singlish because:

- 'ABC' and 'AEG' only share the letter 'A'
- 'AEG' and 'DEE' only share the letter 'E'
- 'DEF' and 'ZD' only share the letter 'D'

As another example:

```
    L2 = ['ABC', 'DEG', 'DEE', 'ZD']
```

This list is not singlish because 'ABC' and 'DEG' share no letters.

Also:

```
    L2 = ['ABC', 'DCA', 'DEE', 'ZD']
```

This list is not singlish because 'ABC' and 'DCA' share two letters ('A' and 'B').

With that, write the function `makeSinglish(L)` that takes a list L of uppercase letters and uses backtracking to return the list M that contains all the same words as L only ordered so that M is singlish. Return None if no such list exists.

Note that there is always more than one correct answer, since if M is correct, then list(reversed(M)) is also correct. You only have to return any one correct answer.

For example:

```
 # This list has two correct answers:
   assert(makeSinglish(['AEG', 'ABC', 'ZD', 'DEE']) in
                       [ ['ABC', 'AEG', 'DEE', 'ZD'],
                         ['ZD', 'DEE', 'AEG', 'ABC'] ])

  # This list has no correct answers:
   assert(makeSinglish(['AEG', 'ABC', 'ZD', 'EF']) == None)
```

Again, you must use backtracking properly to receive full credit.

**Begin your FR4 answer here or on the next page.**

**Begin or continue your answer to FR4 here:**

**You may continue your answer to FR4 here:**

**Free Response / FR5: Line Class [15 pts]**

Write the Line class so the following test code passes.
Do not unstaple this page, even if you really want to.

Do not hardcode any test cases. Your solution must work
in general. However, you do not have to implement any methods that are not
being tested here.

```
line1 = Line(2, 3)
assert(line1.m == 2)
assert(line1.b == 3)
assert(str(line1) == 'y=2x+3')
assert(str([line1]) == '[y=2x+3]')

assert(line1.evalAt(10) == 23) # evaluate y=2x+3 at x=10

assert(line1 == Line(2, 3))
assert(line1 != Line(3, 2))
assert(line1 != 'do not crash here')

s = set()
assert(Line(2, 3) not in s)
s.add(Line(2, 3))
assert(Line(2, 3) in s)

# line1.getPerpendicularLineAt(x) returns a new line that is
# perpendicular to line1 and intersects line1 at x.
# The perpendicular line has a slope of -1/m.
# You can assume that m != 0.
# Thus, line1.getPerpendicularLineAt(2) has a slope of -1/2.
# Since line1 goes through (2, 7), then
#       line2 must also go through (2, 7).
# We use this to find b for line2.
# So line2 is y = -0.5x + 8.

line2 = line1.getPerpendicularLineAt(2)
assert(almostEqual(line2.m, -0.5))
assert(almostEqual(line2.b, 8))
assert(almostEqual(line2.evalAt(2), 7))
```

**Begin your FR5 answer here.**

**You may continue your answer to FR5 here:**

**You may continue your answer to FR5 here:**

**Bonus Code Tracing (BonusCT) [Optional, 1 pt each]**
Bonus problems are not required.
For these CTs, indicate what the code prints.
Place your answer (and nothing else) in the box below the code.

**BonusCT1:**
```
def bonusCt1():
    def f(x, y, z):
        g = (x + y)/2
        d = g**2 - z
        if abs(d) < 10**-8: return round(g)
        else: return f(x, g, z) if d > 0 else f(g, y, z)
    return f(0, 1234, 17)
print(bonusCt1())
```

**BonusCT2:**

```
def bonusCt2(L):
    def g(L):
        for i in range(1, len(L)):
            if abs(L[i]) > abs(L[i-1]):
                L[i], L[i-1] = L[i-1], L[i]
                return True
    return bonusCt2(L) if g(L) else L

print(bonusCt2([2, -5, 5, 4, 3, -3]))
```

**You may use this page to show your work (but we will not grade anything on this page, and will only grade answers in the designated answer boxes)**