fullName:______________________ andrewID:________________ section:___

15-112 S25
Midterm1 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 6 or beyond unit 4 (1D and 2D Lists).
   - Do not use dictionaries, sets, or recursion.
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!

Code Tracing (CT) [20 pts, 5 pts each]
For each CT, indicate what the code prints
Place your answer (and nothing else) in the box below the code.

Note: If floats occur in these CTs, they will have no more than one digit after the decimal point.

CT1:

```
import math

def ct1(x):
    print(int(x)/10, int(x/10))
    x = math.ceil(x) % 5
    y = 10 + 4 * 3 ** 2 - 5
    print(x % y, y % x)
    print(int(f'{x}{y}' * 2) + 1)
    x = y
    y = x
    return x, y
print(ct1(12.5))
```

CT2:

```
def g(n):
    return n**2 + 1 if n % 2 == 0 else 2*n

def f(n):
    return abs(g(n+1) - g(n))

def ct2(n):
    print(f(n), f(n+1))

ct2(5)
```

CT3:

```
def ct3(s):
    L = s.splitlines()
    s = L[1]
    result = ''
    for v in s.split(','):
        if v.isalpha():
            if v.isupper():
                continue
            result += f'X{v}'.lower()
        elif v.isdigit():
            result += chr(ord('A') + int(v[-1]))
        elif v == '':
            break
    return result

print(ct3('''X,bc,1
Cf,A,p2,42,,28
g'''))
```

You may use this page as scratch area for the CTs. However, as usual, you may not unstaple it or any other page, and we will only grade what you write in the answer boxes.

CT4:

```
import copy

def ct4(L):
    M = copy.copy(L)
    N = copy.deepcopy(L)
    M.append([1])
    L[0] += N.pop()
    L = L + [2]
    print(L)
    print(M)
    print(N)

L = [[0], [5]]
ct4(L)
print(L) # do not miss this!
```

You may use this page as scratch area for the CTs. However, as usual, you may not unstaple it or any other page, and we will only grade what you write in the answer boxes.

Free Response / FR1: nthNearlySquarePrime(n) [20 pts]

Note: for this problem, you can call `isPrime(n)` without writing it.

Background: we will say that p is a "nearly-square prime" (a coined term) if p is prime and either `p-1` or `p+1` is a perfect square. (Reminder: A number `s` is a perfect square if `s==k**2` for some integer value of k.) The first few perfect squares are 0, 1, 4, 9, 16, and 25.

With that, write the function `nthNearlySquarePrime(n)` that takes a non-negative int n and returns the nth nearly-square prime. This function should work for reasonably large values of n.

You must not use strings or lists for this problem.

Here are some test cases:

```
    assert(nthNearlySquarePrime(0) == 2)  # 2-1 is 1**2 == 1
    assert(nthNearlySquarePrime(1) == 3)  # 3+1 is 2**2 == 4
    assert(nthNearlySquarePrime(2) == 5)  # 5-1 is 2**2 == 4
    assert(nthNearlySquarePrime(3) == 17) # 17-1 is 4**2 == 16
    assert(nthNearlySquarePrime(4) == 37) # 37-1 is 6**2 == 36
```

Begin your FR1 answer here or on the next page.

Begin or continue your answer to FR1 here:

Free Response / FR2: evalCode(code) [20 pts]

Note: for this problem, you may not call the builtin `exec()` or `eval()` functions.

Background: for this problem, you will write `evalCode(code)` that takes a multiline string of code and returns a list of all the values it prints.
Of course, the code is very restricted. It can only contain:

- blank lines (which are ignored)
- lines of the form: `L[index] = value`
- lines of the form: `L[index] += value`
- lines of the form: `print(L[index])`
- lines can contain comments starting with #. These are ignored.

Also:

- All indexes are single digits (between 0 and 9)
- All values are non-negative integers
- The only operators are = and +=
- Operators always have one space before and one space after them
- The code has no errors of any kind

For example:

```
code = '''
# ignore this comment
L[0] = 3    # sets L[0] to 3
print(L[0]) # prints 3
L[0] += 15  # adds 15 to L[0]
print(L[0]) # prints 18
L[0] = 4    # sets L[0] to 4
print(L[0]) # prints 4
L[7] = 123
print(L[7]) # prints 123
'''
```

Since this code prints 3, then 18, then 4, then 123, `evalCode(code)` returns `[3, 18, 4, 123]` in this case.
Note: Code with no print statements would return an empty list.

With that, write the function `evalCode(code)` that takes a string of code as just described and returns a list of the values that code prints (if any).

Begin your answer to FR2 here:

You may continue your answer to FR2 here:

Free Response / FR3: fixTournament(t) [20 pts]

Background: for this problem, we will consider a tournament played by 6 teams. The teams are named 'A', 'B', 'C', 'D', 'E', and 'F'. The tournament has 5 rounds. In each round, each team plays one game, so there are 3 games per round, each played by two teams. We list the teams in alphabetical order, so 'AF' means that team 'A' played team 'F'.

Here is such a tournament:

```
tournament = [ # round1 round2 round3 round4 round5
               [  'AF',  'CE',  'BC',  'DE',  'BD' ],
               [  'BE',  'DF',  'AD',  'AC',  'CF' ],
               [  'CD',  'AB',  'EF',  'BF',  'AE' ]
             ]
```

For example, in round1, 'A' plays 'F', 'B' plays 'E', and 'C' plays 'D'.

Note that after 5 rounds, every team has played every other team exactly once.

Given this, we now have a broken tournament, where somehow we lost the second team from each pairing in the last round. These are replaced with '?'. So this is a broken tournament:

```
brokenTournament = [ # round1 round2 round3 round4 round5
                     [  'AF',  'CE',  'BC',  'DE',  'B?' ],
                     [  'BE',  'DF',  'AD',  'AC',  'C?' ],
                     [  'CD',  'AB',  'EF',  'BF',  'A?' ]
                   ]
```

We can fix this tournament because we know that every team plays every other team exactly once. So, for example, team B plays 'E' in round1, 'A' in round2, 'C' in round3, 'F' in round4, but never plays 'D', so the first row of round5 must be 'BD'.

With this in mind, write the function `fixTournament(t)` that takes a 3x5 2D list `t` describing a tournament with a broken final round, and non-mutatingly returns the fixed version of that tournament.

Using the example from above:

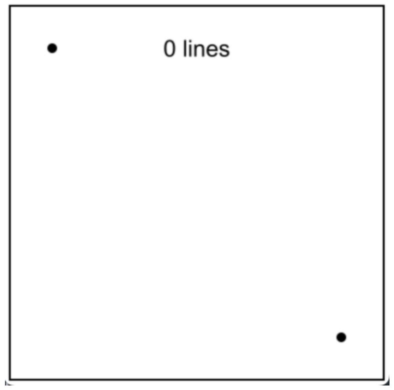
```
  assert(fixTournament(brokenTournament) == tournament)
```

Begin your answer to FR3 here:

You may continue your answer to FR3 here:

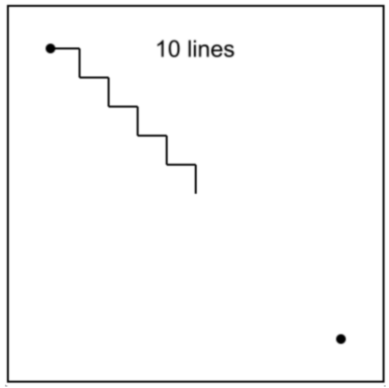
Free Response / FR4: Plus or Minus Lines [20 pts]

Note: for this exercise, you may assume the window is always 400x400.
Write an animation that looks like this at first:



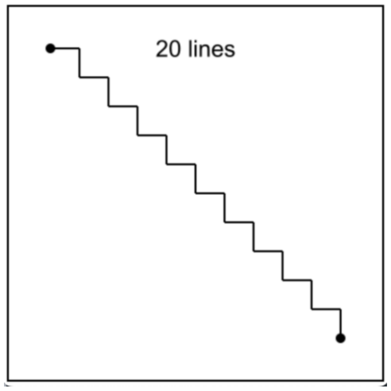
This includes 3 shapes:
1. A dot of radius 5 at (50, 50).
2. A dot of radius 5 at (350, 350)
3. A label – “0 lines” – centered at (200, 50).

Each time the user presses ‘+’ (the plus key), the number of lines increases, forming a staircase from the upper dot to the lower dot.
Here it is after 10 presses of ‘+’:

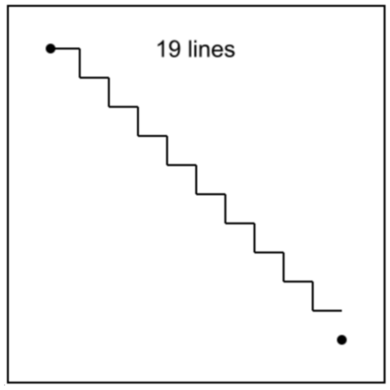


Note:
1. The label now is “10 lines”
2. Of the 10 lines, 5 are horizontal and 5 are vertical, forming a staircase

And here it is after 20 presses of '+':



Also, if the user presses '-' (the minus key), then the number of lines decreases by 1. So here is previous example after the user presses '-' once:



Some final notes:

1. If the user presses '+' when there are 20 lines, ignore it.
2. If the user presses '-' when there are 0 lines, ignore it.
3. If there is only 1 line, the label must be '1 line' and not '1 lines'.
4. All vertical and horizontal lines are of equal length.
5. When all 20 lines exist, 10 are horizontal, and 10 are vertical.

Begin your FR4 answer on the next page.

```
#Begin your answer to FR4 here:

from cmu_graphics import *

def onAppStart(app):










def onKeyPress(app, key):
```

```
#Continue your answer to FR4 here:

def redrawAll(app):
```

```
#You may continue your FR4 answer here
```

```
runApp()
```

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(L):
    L = str(L)
    z = ', ['
    while z in L:
        L = L.replace(z, ', 1, ')
    L = L.replace(']', '0').replace('[', '-').replace(', ', '+')
    return eval(L)

L = [ [[1, 2], [3, 4]], [[5], [6]] ] # a 3d list!
print(bonusCt1(L))
```

BonusCT2:

```
def bonusCt2(z):
    # hint: bin(5) returns '0b101',
    # since 101 in base 2 equals 5 in base 10
    def f(n):
        C = [0] * 20
        for k in range(n):
            s = list(reversed(bin(k)[2:]))
            for i in range(len(s)):
                C[i] += int(s[i])
        return sum(C)
    return f(2**z) - f(z)
print(bonusCt2(4))
```