Name:

E-mail ID:

On my honor, I pledge that I have neither given nor received help on this test.

Signature:

Test rules and information

- Print your name, id, and pledge as requested.
- The only paper you can have in front of you during the exam is the test itself (which includes one scrap piece of paper), and one page of notes.
- The only library allowed to be imported is `Image` for questions 5 and 6. There is no need to make use of `get.py` because of none of your solutions involve getting user input or web resources.
- All submissions must be made during the test. No forgotten submissions will be accepted after the fact.
- This pledged exam is closed textbook. The only device you may access during the test is your own laptop.
- You are not allowed to access class examples or your own past assignments during the test; i.e., the only Python code you may access or view are ones that you develop for this test.
- The only windows that can be open on your computer are PyCharm and a single browser with tabs only open to the class website.
- Code should compile and demonstrate proper programming style; e.g., whitespace, identifier naming, etc.
- *None of the functions you develop should print any output.*
- Each attempted program that does not generate an error when run is worth five points. Each program that passes testing is worth four additional points.
- Each attempted module that does not generate an error when its functions are invoked is worth five points.
- Each attempted function is worth four points except for functions *change*() and *report*(), which are worth six points. *An attempted function that prints output is incorrect.* The expected grading rubric is
    - One point for attempting the function;
    - For functions other than *change*() and *report*() there are three points for getting all test cases correct.
    - For functions *change*() and *report*() there are five points for getting all test cases correct.
- Our testing of the functions will involve different test cases than the ones used for elaborative purposes in the problem descriptions.

1. (9 points) Develop a *program* `q01.py`. The program prints the string *'I pledged to do my honest best on this test.'* and nothing else. Thus, a program run should produce output

```
I pledged to do my honest best on this test.
```

---

2. (9 points) Develop a *program* `q02.py`. The program makes a single input request for four integer values. The values are respectively *a*, *b*, *c*, and *x*. The program prints the value of the equation $a \cdot x^2 + b \cdot x + c$. Below are three sample runs of the program.

```
Enter four integers: 2 3 4 5
69
```

```
Enter four integers: 3 1 4 1
8
```

```
Enter four integers: 2 4 8 16
584
```

---

3. (11 points) Develop module `q03.py`. The module defines a single function *change*().

   - Function *change*() has a single string parameter *s*. The function returns a new string equal to s except the case of all letters is switched (e.g., 'e' becomes 'E' and vice-versa). For example, the following code segment

     ```
     x = 'sAw';      y = q03.change( x );     print( y )
     x = 'toPS';     y = q03.change( x );     print( y )
     x = 'Act 1';    y = q03.change( x );     print( y )
     ```

     should produce output

     ```
     SaW
     TOps
     aCT 1
     ```

   - Program `q03-tester.py` consists of the preceding code segment.

---

4. (13 points) Develop module `q04.py`. The module defines two functions *eval*() and *flip*().

   - Function *eval*() has two parameters *d* and *v*, where *d* is a dict and *v* is of any type. The function returns the list of keys in *d* whose value is *v*. For example, the following code segment

```
d = { 0: 'z', 1 : 'o', 2: 'e', 3: 'o', 4: 'e', 5: 'o' }

ks = q04.eval( d, 'e' );  print( ks )
ks = q04.eval( d, 'i' );  print( ks )
ks = q04.eval( d, 'o' );  print( ks )
```

should produce output

```
[2, 4]
[]
[1, 3, 5]
```

- Function *flip*() has one parameter *d* of type dict. The function returns a new dict *id* whose keys are the values of dict *d*. For every value *v* is in *d.values*(), there is a mapping in dict *id* that maps *v* to all of the keys in *d* that map to it. For example, the following code segment

```
d = { 0: 'z', 1 : 'o', 2: 'e', 3: 'o', 4: 'e', 5: 'o' }

id = q04.flip( d );        print( id )
```

should produce output (the order of your key-value mappings may be different, but the mappings themselves should be the same)

```
{'z': [0], 'o': [1, 3, 5], 'e': [2, 4]}
```

Hint: I expect *eval*() to be helpful in building the return dict *id*.

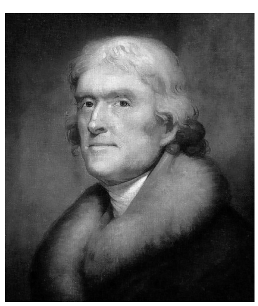
- Program `q04-tester.py` consists of the preceding code segments.

---

5. (9 points) Develop module `q05.py`. The module defines a function *negative*().

- Function *negative*() has one parameter *drawing*, where *drawing* is an *Image*. The function returns a new image of the same dimensions as *drawing*.

  The pixels in the new image are the color negative of the corresponding pixels in *drawing*; that is, if coordinate (*x*, *y*) of *drawing* has RGB value (*r*, *g*, *b*), then the new image at coordinate (*x*, *y*) has RGB color (255 – *r*, 255 – *g*, 255 – *b*). For example, the two below images are color negatives of each other.

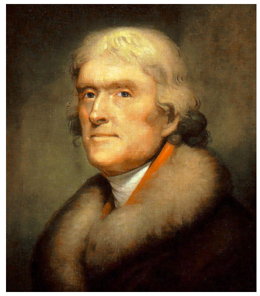
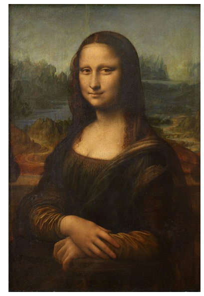


- Program `q05-tester.py` tests function *negative*() and should produce the above image on the right.

---

6. (9 points) Develop module `q06.py`. The module defines a function *average*().

- Function *average*() has one parameter *drawing*, where *drawing* is an *Image*. The function returns a triple (*ar*, *ag*, *ab*), where *ar, ag,* and *ab* are respectively the *integer* averages of the red, green, and blue components of the RGB values for the pixels in *drawing*. Hint: compute the sums of the reds, greens, and blues of the RGB values for all the pixels in *drawing* and divide each sum by the number of pixels in the drawing (i.e., *drawing* width by *drawing* height).

- Program `q06-tester.py` tests function *average*() on color versions of the above two images and should produce output

```
(96, 73, 45)
(75, 61, 42)
```

---

7. (11 points) Develop module `q07.py`. The module defines a function *report*().

- Function *report*() has three parameters, *words*, *misspellings*, and *w*.
  - Parameter *words* is a list of correctly spelled words,
  - Parameter *misspellings* is a dataset of words and how they are sometimes misspelled. Each row in the dataset is a list of strings. The first string in the list is a correctly spelled word, the remaining strings are misspelled variants of that word. For example, one row in the dataset could be the list ['humorous', 'humerous', 'humourous']; i.e., the strings 'humerous' and 'humourous' are misspellings of 'humorous'.
  - Parameter *w* is a string.

  The function *report*() return value is as follows:

  - If *w* is in *words*, the function returns *w*;
  - If instead *w* is an element of one of the rows of *misspellings*, the function returns the first element of that row (i.e., the correct spelling of *w*);
  - Otherwise, the function returns the string '*' + *w* + '*'.

  For example, the following code segment

```
WORDS_URL = 'http://www.cs1112.org/datasets/common_words.txt'
MISSPELLINGS_URL =
'http://www.cs1112.org/datasets/common_misspellings.csv'

common = get.strings_from_url( WORDS_URL )
```

```
corrections = get.csv_sheet_from_url( MISSPELLINGS_URL)

s = 'mispell'; t = q07.report( common, corrections, s );  print( t )
s = 'weird';   t = q07.report( common, corrections, s );  print( t )
s = 'lol';     t = q07.report( common, corrections, s );  print( t )
```

should produce output

```
misspell
weird
*lol*
```

- Program `q07-tester.py` consists of the preceding code segment.

---

8. (25 pts) Develop module `q08.py` to support DNA analyses and manipulations.

   A DNA molecule is a *strand* (sequence) of *nucleotides*. The four types of nucleotides are adenine, cytosine, guanine, and thymine. Nucleotides are normally represented by their first letter in *upper case format*; i.e., A, C, G, and T. A strand is represented as a sequence of A's, C's, G', and T's; e.g., GGAACCATGACATAG. A cell interprets its DNA strand as program for what biological function to carry out. An individual DNA instruction is a three-letter sequence called a *codon*. For DNA strand GGAACCATGACATAG, the codons are GGA, ACC, ATG, ACA, and TAG.

   The module defines functions *length*(), *canonical*(), *slice*(), *splice*(), and *insert*(). The functions all deal with strands of nucleotides.

   - Function *length*() has one string parameter *s*, where *s* is a string of nucleotides. The function returns the number of nucleotides in *s*. For example, the following code segment

     ```
     n = q08.length( 'GGAACCATGACATAG' );   print( n )
     ```

     should produce output

     ```
     15
     ```

   - Function *canonical*() has one string parameter *s*, where *s* is a string of nucleotides. The function returns a new upper-case version of *s*. For example, the following code segment

     ```
     s = q08.canonical( 'acgTgCa' );       print( s )
     ```

     should produce output

     ```
     ACGTGCA
     ```

   - Function *slice*() has three parameters *s*, *a*, and *b*, where *s* is a string of nucleotides, and *a* and *b* are indices into *s*. The function returns a new string of nucleotides that equals the nucleotides of *s* starting with the nucleotide at index *a* and up to but not including the nucleotide at index *b*. For example, the following code segment

```
s = q08.slice( 'GGAACCAT', 2, 5 );          print( s )
```

should produce output

```
AAC
```

- Function *splice*() has two parameters *s* and *t*, where *s* and t are both strings of nucleotides. The function returns a new string of nucleotides that equals the nucleotides of *s* followed by the nucleotides of *t*. For example, the following code segment

```
s = q08.splice( 'CAT', 'ACT' );       print( s )
```

should produce output

```
CATACT
```

- Function *insert*() has three parameters *s*, *t*, and *a*, where *s* and *t* are both strings of nucleotides and *a* is an index. The function returns a new string of nucleotides that equals the nucleotides of *s* starting with its initial nucleotide and up to but not including the nucleotide at index *a*, followed by the nucleotides of *t*, followed the remaining nucleotides of *s*. For example, the following code

```
s = q08.insert( 'CCCGGG', 'ATTA', 3 ); print( s )
```

should produce output

```
CCCATTAGGG
```

- Program `q08-tester.py` consists of the preceding code segments.

---

9. (9 pts) Develop module `q09.py` to further support DNA analysis by defining function *is_legal*().

- Function *is_legal*() has one string parameter *s*. The function returns a Boolean value whether *s* is a string of *upper-case* nucleotides. For example, the following code segment

```
b = q09.is_legal( 'ACTGGTCA' );       print( b )
b = q09.is_legal( 'xyz' );            print( b )
b = q09.is_legal( 'acgtactg' );       print( b )
```

should produce output

```
True
False
False
```

- Program `q09-tester.py` consists of the preceding code segment.

---

10. (9 points) Develop module `q10.py` to further support DNA analysis by defining a function *codons*().

- Function *codons*() has one string parameter *s*, where *s* is a string of nucleotides. The function returns the list of codons (nucleotide triplets) making up *s*. You can assume the length of *s* is a multiple of three. For example, the following code segment

```
c = q10.codons( 'ACGGAACCATGACATAGG' )

print( c )
```

  should produce output

```
['ACG', 'GAA', 'CCA', 'TGA', 'CAT', 'AGG']
```

- Program `q10-tester.py` consists of the preceding code segment.

---

11. (21 points) Develop module `q11.py` to support the playing of the two-person game sticks. The game starts off with three rows of sticks, where the first row has 1 stick, the middle row has 3 sticks, and the last row has 5 sticks. A visual representation would be

```
  |
 |||
|||||
```

A move of the game is the removal of one or two sticks from a single row. *The game is over when there is exactly one stick left altogether in the three rows*.

The module implements four functions *setup*(), *is_over*(), *is_valid*(), and *crossoff*().

- Function *setup*() has no parameters. The function returns a new three-element list of integers whose values are respectively 1, 3, and 5. For example, the following code segment

```
g = q11.setup();

print( g )
```

  should produce output

```
[1, 3, 5]
```

- Function *is_over*() has one parameter *sticks*, where *sticks* is a list of three integers. The function returns a Boolean value whether the sum of the *sticks* elements is one. For example, the following code segment

```
s1 = [ 0, 1, 1 ];     b = q11.is_over( s1 );       print( b )
s2 = [ 0, 0, 3 ];     b = q11.is_over( s2 );       print( b )
s3 = [ 0, 1, 0 ];     b = q11.is_over( s3 );       print( b )
```

  should produce output

```
False
False
True
```

- Function *is_valid*() has three parameters *sticks*, *r*, and *n*, where *sticks* is a list of three integers, and *r* and *n* are integers. The function returns a Boolean value whether the decrementing row *r* of sticks by *n* is a valid move:
  - If *r* is not a valid index into *sticks*, the function returns *False*.
  - If instead *n* is not equal to 1 or 2, the function returns *False*.
  - If instead the r^{th} element of sticks is less than *n*, the function returns *False*.
  - Otherwise, the function returns *True*.

  For example, the following code segment

```
s = [ 0, 2, 1 ] ;     b = q11.is_valid( s, 3, 1 );    print( b )
s = [ 0, 2, 1 ] ;     b = q11.is_valid( s, 0, 1 );    print( b )
s = [ 0, 2, 1 ] ;     b = q11.is_valid( s, 2, 1 );    print( b )
s = [ 0, 2, 1 ] ;     b = q11.is_valid( s, 1, 2 );    print( b )
s = [ 0, 2, 1 ] ;     b = q11.is_valid( s, 1, 1 );    print( b )
s = [ 0, 2, 1 ] ;     b = q11.is_valid( s, 1, 0 );    print( b )
```

  should produce output

```
False
False
True
True
True
False
```

- Function *cross_off*() has three parameters *sticks*, *r*, and *n*, where *sticks* is a list of three integers, and *r* and *n* are integers. The function does not return a value or print any output. If decrementing row *r* of *sticks* by *n* is a legal move, then that decrementing of *sticks* is performed; otherwise, no action is taken. For example, the following code segment

```
s = [ 0, 2, 1 ];    q11.cross_off( s, 3, 1 );     print( s )
s = [ 0, 2, 1 ];    q11.cross_off( s, 0, 1 );     print( s )
s = [ 0, 2, 1 ];    q11.cross_off( s, 2, 1 );     print( s )
s = [ 0, 2, 1 ];    q11.cross_off( s, 1, 2 );     print( s )
s = [ 0, 2, 1 ];    q11.cross_off( s, 1, 1 );     print( s )
s = [ 0, 2, 1 ];    q11.cross_off( s, 1, 0 );     print( s )
```

  should produce output

```
[0, 2, 1]
[0, 2, 1]
[0, 2, 0]
[0, 0, 1]
[0, 1, 1]
[0, 2, 1]
```

- Program `q11-tester.py` consists of the preceding code segments.

Scratch