ECE 606, Fall 2021, Assignment 1
Due: Thursday, September 16, 11:59pm

Submission: submit three things: (i) your solutions for the written problems to crowdmark, (ii) a page with your name and student ID for the **[python3]** problem on crowdmark — you can extract and use the last page of this assignment handout, and, (iii) your **[python3]** file to the appropriate Dropbox on Learn.

1. You have a box that contains the letters VAGABOND. How many different strings of 4 letters can you make using only the letters from the box? Examples of three such strings: 'VAGA,' 'GAAV' and 'OGAB.'

2. Suppose, given an array $A[1,\ldots,n]$ of n integers where n is odd, we want to determine the minimum and maximum of those n integers. Consider the following algorithm.

   (Clarification on notation: we use "$\leftarrow$" for assignment to a variable.)

   ```
   1  min ← A[1], max ← A[1]
   2  i ← 2
   3  while i < n do
   4      if A[i] ≤ A[i + 1] then
   5          if A[i] < min then min ← A[i]
   6          if A[i + 1] > max then max ← A[i + 1]
   7      else
   8          if A[i] > max then max ← A[i]
   9          if A[i + 1] < min then min ← A[i + 1]
   10     i ← i + 2
   11 return min, max
   ```

   (a) As a function of n, how many comparisons does the algorithm perform on input some integer array $A[1,\ldots,n]$?

   (Comparisons are performed in Lines (3)–(6), (8)–(9).)

   (b) In what manner would you edit the above algorithm to account for even values of n in addition to odd values?

3. Apparently, if the Earth is flat, then I am Santa Claus. Prove: I am Santa Claus implies that the Moon is made of cheese, only if the Earth is flat implies that the Moon is made of cheese.

4. **[python3]** Python 3 supports arbitrary precision for integers. In this exercise, we seek to realize algorithms for finite-precision arithmetic with integers in Python 3.

   Two's complement is a way of representing integers when we are afforded finite precision, i.e., a fixed number of digits and non-negative integers, only. I intentionally say "digits," and not, for example, "bits" because we will work with decimal, or base-10, integers for this exercise. Just to push our understanding and thinking process a bit. `:)`

   Given d digits where $d \in \{1,2,\ldots\}$, we are able to represent each of the 10^d integers in the interval $\left[-10^d/2, 10^d/2-1\right]$ as a non-negative integer in the interval $\left[0, 10^d-1\right]$. The

representation is: $i \in \left[-10^d/2, 10^d/2 - 1\right]$ is represented as $i \bmod 10^d$. Recall that $x \bmod y$, for x an integer and y a positive integer, is the unique integer $r \in \{0, 1, \ldots, y-1\}$ such that there exists an integer q with $x = yq + r$. For example, $113 \bmod 12 = 5$ and $-13 \bmod 12 = 11$. Python 3 uses `%` for the modulo operator.

As an example of our encoding, suppose $d = 3$. Then, $10^d = 1000$ and the integers we can represent are $-500, -499, \ldots, 0, 1, \ldots, 499$. The number 226 is represented as $226 \bmod 1000 = 226$, and the number -308 is represented as $-308 \bmod 1000 = 692$.

What you need to do: design and implement the four subroutines in `a1p4.py` that is provided to you on Learn. The subroutines are:

- `encode` — takes as argument d, the number of digits we allow where d is a positive integer, and an integer i in the interval $\left[-10^d/2, 10^d/2 - 1\right]$. The subroutine should return the representation of i in the interval $\left[0, \ldots, 10^d - 1\right]$.
- `decode` — takes as argument d, the number of digits we allow where d is a positive integer, and an integer i in the interval $\left[0, \ldots, 10^d - 1\right]$. The subroutine should return the integer in the interval $\left[-10^d/2, 10^d/2 - 1\right]$ that i represents.
- `add` — takes three arguments, (i) d, a positive integer that is the number of digits we allow in our encoding of an integer, and, (ii) and (iii) i, j both of which are in the interval $\left[0, \ldots, 10^d - 1\right]$ and assumed to encode integers in the interval $\left[-10^d/2, 10^d/2 - 1\right]$. This subroutine should return the sum of i and j in two's complement encoding. For example, `add`$(3, 412, 23)$ should return 435 and `add`$(3, 412, 895)$ should return 307. That is, you should accurately capture overflow that can happen.
- `multiply` — takes three arguments, (i) d, a positive integer that is the number of digits we allow in our encoding, and (ii) and (iii) i, j both of which are in the interval $\left[0, \ldots, 10^d - 1\right]$ and assumed to encode integers in the interval $\left[-10^d/2, 10^d/2 - 1\right]$. This subroutine should return the product $i \times j$ in two's complement encoding. You should assume that the underlying computer natively implements the `add` routine from above only. And each addition can result in an overflow. The algorithm you should implement for multiplication is the following.

  Given two integers, x, y where $y \geq 0$, the product xy is expressed correctly as the following recurrence. ($\lfloor \cdot \rfloor$ is the floor function; in Python 3, one can employ integer division, "//")

$$xy = \begin{cases} 0 & \text{if } y = 0 \\ 2x \times \lfloor y/2 \rfloor & \text{if } y > 0 \text{ and } y \text{ is even} \\ x + 2x \times \lfloor y/2 \rfloor & \text{otherwise} \end{cases}$$

For example, following is the manner in which multiply$(2, 13, 11)$ would proceed.

$$\begin{aligned}
13 \times 11 &= 13 + (13 + 13) \times 5 \\
&= 13 + (13 + 13) + ((13 + 13) + (13 + 13)) \times 2 \\
&= 13 + (13 + 13) + (((13 + 13) + (13 + 13)) + ((13 + 13) + (13 + 13))) \times 1 \\
&= 13 + (13 + 13) + (((13 + 13) + (13 + 13)) + ((13 + 13) + (13 + 13))) \\
&= 13 + (13 + 13) + (((13 + 13) + (13 + 13)) + (26 + 26)) \\
&= 13 + (13 + 13) + (((13 + 13) + (13 + 13)) + 52) \\
&= 13 + (13 + 13) + ((26 + 26) + 52) \\
&= 13 + (13 + 13) + (52 + 52) \\
&= 13 + (13 + 13) + 4 \\
&= 13 + (26 + 4) \\
&= 13 + 30 = 43
\end{aligned}$$

There is overflow when we perform $52 + 52$, i.e., add$(2, 52, 52)$.

What you are provided: (i) a skeleton a1p4.py file which you should populate and submit. You need to meet the constraints mentioned there: we should see no import statements; if you import anything, you get an automatic 0. (ii) a tester file, tester-a1p4.py; this should be runnable directly from a command-line; it provides some test cases. These are not necessarily the test cases we will use for marking.

When we provide a specification for an argument or state an assumption, e.g., "... an integer i in the interval $\big[0, \ldots, 10^d - 1\big]$," this means that we will test only within that specification/assumption. E.g., if $d = 2$, then you can be guaranteed that the only i values we will try will be integers between 0 and 99. ECE 606 is not a course on robust software development. Rather, our focus is algorithms and the logic that underlies them.

What you should submit: your file a1p4.py, exactly with that name, to the appropriate Dropbox on Learn. Do not submit anything else to the Dropbox. Do not change the name of the file or our script will fail and you will get a 0 for the problem.

Page in which you fill in your name and student ID and submit to crowdmark for the **[python3]** problem.

Name:

Student ID: