Written problems:

Note. All problem numbers refer to the **second edition** of the textbook.

1. Throughout this course, we will say that an integer a is an "n-bit (nonnegative) integer" if $0 \leq a < 2^n$ (I will also sometimes use the phrase "exactly n bits long" to mean that $2^{n-1} \leq a < 2^n$, i.e. that a is an n-bit integer but not an $(n-1)$-bit integer).

   The *Data Encryption Standard* (DES) is a private-key encryption algorithm that was a government standard from 1977 to 2002. DES uses 56-bit secret keys. Suppose that Eve attempts a brute-force attack on DES by trying to decrypt an intercepted cipher text with every possible 56-bit key until she finds something that looks like English text. If Eve's system can try 1 billion keys per second, how long would it take her to try all of the keys (and thus be sure to break the encryption)? Feel free to make simplifying assumptions and approximations here, to just get a very rough estimate. The main purpose is to gain some intuition for the scales involved.

   (By 1999, a distributed system was able to break DES encryption in less than 24 hours. DES was replaced in 2002 by a new standard, called AES, which uses keys of at least 128 bits. For "top secret" communication, the government uses AES with 256 bit keys.)

2. Textbook exercise 1.6. (Divisibility facts)

3. Textbook exercise 1.9, parts (a) and (b). Use a calculator/computer for the arithmetic, but show the steps. (Euclidean algorithm for GCD)

4. Textbook exercise 1.11. (Facts relating GCD to Bézout's identity)

5. This problem is meant to allow you to think about how the sizes of an input to a program influence its runtime (and also to practice reading Python code), in a concrete setting. The purpose is for you to try to make some educated guesses for now; you do not need to be correct to receive full points. This problem will also be a chance for you to try out the Gradescope code submission system with some mock submissions that will not count towards your grade.

   (a) Consider the following function. It takes a positive integer n, and returns the number of divisors of n. Read the code and make sure you understand how it works.

   ```
   def numdivs(n):
       count = 0
       for d in range(1,n+1):
           if n%d == 0:
               count += 1
       return count
   ```

   Make a rough guess for how many bits long n can be before this function takes at least 1 second to finish running, and explain your reasoning. You will receive points for this problem as long as your explanation is reasonable.

   (b) On Gradescope, I've created a mock programming problem called "DEMO counting divisors" that will test this function on integers of various sizes (up to 100 bits), with a time limit of 1 second for each test case.

Note. Any "assignment" labeled "DEMO" does not count to your grade in any way; it mereley served as a demonstration.

Use this mock problem to test your answer from part (a), as follows: copy out the code above into a file called `soln1.py`, and submit it to Gradescope. From the autograder output, how many bits could this function in fact handle in 1 second? Compare to your guess from part (a).

(c) Now consider the following alternative implementation. Briefly explain why this also correctly calculates the number of divisors of n.

```
def numdivs(n):
    count = 0
    d = 1
    while d*d < n:
        if n%d == 0:
            count += 2
        d += 1
    if d*d == n:
        count += 1
    return count
```

(d) Predict how many bits long n must be before the code in part (c) cannot finish in less than 1 second, and explain your reasoning. Then copy the code out to a file `soln2.py`, submit it to Gradescope, and see how accurate your guess was. Again, you will receive points for this problem as long as your reasoning makes sense.

Note. You might enjoy trying to implement a more efficient solution to this problem, and testing it with the Gradescope autograder. This is certainly not in any way a required part of the course, but I'd be interested to here about any interesting ideas you have, and how many testcases you're able to solve.

Programming problems:

1. The first part of this week's programming assignment is to work on an excellent Python tutorial that covers almost all of the basics that we will need for the programming in this course. Friday's class (2/11) will be devoted to giving you time to work on the tutorial while I am on hand to help you and answer questions. **You will receive full points for this part as long as you make a good-faith effort to complete the prescribed exercises by the end of next week. Please let me know if you are having difficulty or think you will require more time.** Please let me know if and where you are having trouble. I will be able to monitor your progress from my account, and see which exercises you've completed, so **you do not need to submit anything for this problem.**

   (a) Go to `https://cscircles.cemc.uwaterloo.ca/`, and create a free account. Please use your college (.edu) email address for you account, so that I can match accounts with the class list from acdata.

   (b) After making your account, go to your profile and enter `npflueger` as your "Guru's Username."

   (c) Read the tutorial and complete the exercises from the following sections:

   - Sections 0 through 10, except 2X, 6D, 7A, and 8.
   - Section 13.

- After doing the rest, go back and work through section 6D (on debugging).

The whole tutorial is excellent, of course, so you would benefit from doing the other sections as well. The list above identifies the parts that will be most important for our programming assignments.

If you have a lot of previous experience and complete these sections quickly, you might enjoy trying section 15C, which relates to a toy cryptosystem (the Caesar cipher).

The remaining two problems will be submitted electronically on Gradescope, and automatically graded. They will be due Friday 2/18. Here are some details/suggestions. I will demonstrate all of this in class on Monday 2/14. Give it a try before then, but don't worry if you can't figure some mechanics out (bring your questions on Monday!).

- I recommend installing *Anaconda* on your machine if you are new to programming in general or Python specifically (if you're more experienced and/or prefer a different setup, see the last bullet). It is a much larger system than is really necessary for this course, but it has many nice features. Visit the following link for installation instructions.

  `https://www.anaconda.com/distribution/`

- Every programming homework problem comes with a bank of sample test cases and some starter code. Go to the following link to find the files for each problem. Save the files to a folder on your machine.

  `https://www.dropbox.com/sh/a11zuil8bm8lylb/AABoeXEln6uR4vmnG-O9outza?dl=0`

- Open Anaconda, and launch a Jupyter notebook. Navigate to the folder with the starter files. Click on the `Xsoln.py` (where `X` is the problem name) file to write your code.
- When you want to test your code on the sample cases, open `Xtester.ipynb` from Jupyter. Run the top cell by clicking in it and pressing Shift-Enter. Then click in the cells for the individual test cases, and press Shift-Enter to run them individually.
- When you have tried the sample cases and want to submit your code to run it on the grading test cases, find the problem on Gradescope, and upload the `Xsoln.py` file to have it graded.
- The autograder will run immediately, and report your score on the problem. **You can resubmit as many times as you like,** so you do not need to wait until your code is final to try submitting.
- Please let me know of any bugs or technical difficulties!
- *Alternative setup* (if you know how to use Python a different way): You don't need to install Jupyter and/or Anaconda to write your code and access the sample cases. Instead, you can just read the file `Xsamples.json` to read the samples direclty, and run them however you like. You can also run the script `Xtester.py` to run all sample cases at once, or run the script with specific test case numbers as command line argument to execute specific test cases (e.g. running `python Xtester.py 3 5` from the command line will execute test cases 3 and 5).

2. Write a function `factor(n)` which takes a composite integer n (in fact, your may assume that n is a product of two prime numbers) and returns two proper factors p, q with $p, q > 1$

and $pq = n$. Your function should be efficient enough to finish in less that one second when n is 40 bits long or less. Half of the test cases will consist of integers 20 bits long or less, so you will receive at least half credit if your implementation can factor these in less than a second each. Submit your solution to the Gradescope assignment "PSet 1 factor."

3. Write a function `gcdList(ls)` which takes a list of positive integers, whose sizes may be as large as 1024 bits, and returns the greatest common divisor of the entire list. Half of the test cases will consists of lists of exactly two elements, so you can begin by writing a function to efficiently compute the gcd of two numbers, which will receive at least half of the points. Submit your solution to the Gradescope assignment "PSet 1 gcdList."