Open Computer Test

Solutions for both sections

Code samples are written in Python, but they were written to be as language-agnostic as possible.

A0. An 8-bit signed integer can represent 256 different values, from -128 to 127.

A1. Use a loop, the multiplication operator, and the integer division operator. Sample code that returns the answer:

```
ans = 0
for i in range(1, 101):
     ans += (i*i) // 2019
print(ans)
```

The answer is 127.

A2. Neither x nor y can be greater than 45 in absolute value since $45^2 = 2025$. We can then iterate over x and y using this rough bound and count the number of points that fit the condition. Sample code that returns the answer:

```
count = 0
for x in range(-45, 46):
     for y in range(-45, 46):
          if x*x + y*y < 2019:
               count += 1
print(count)
```

The answer is 6333.

A3. The simplest solution is probably to convert numbers from 1 to 100 into strings and concatenate them together, and then index into the string, remembering that the 0-th character is S_1. Sample code that returns the answer:

```
S = ""
for i in range(1, 101):
     S = S + str(i)
print(int(S[21]) * int(S[38]) * int(S[52]) * int(S[81]) * int(S[94])
* int(S[131]) * int(S[175]))
```

The answer is 2016.

A4. It's straightforward to imagine a naïve solution that recursively calculates the n^{th} Fibonacci number and takes it mod 2019, but this takes exponential time. Two important optimizations that allow the solution to run in reasonable time:

1. Don't use a recursive solution to compute F_n. Instead, only keep the last two numbers computed (F_{n-1}, F_{n-2}) to compute the next one, updating them as needed. A simple way to implement this is to append to an array to store them.
2. Don't store the entire number F_n each time. Instead, only store F_n mod 2019 and do all arithmetic mod 2019, taking advantage of the fact that $\big((a \,\%\, c) + (b \,\%\, c)\big) \,\%\, c = (a + b) \,\%\, c$. This is just restating a fundamental property of modular arithmetic in programming shorthand: that modulo c, the numbers a, b are always equivalent to every number with the same remainder when divided by c. For example, $10000 + 4320$ and $(10000 \; mod \; 10) + (4320 \; mod \; 10)$ are equivalent in modulo 10.

Sample code that implements this solution:

```
# Create array to store intermediate results
# where arr[i] = fibonacci(i) and index 0 is unused
arr = [0, 1, 1]
for i in range(3, 20192020):
      # Calculate fibonacci(i) and append it to the array
      arr.append((arr[i-1] + arr[i-2]) % 2019)
print(arr[20192019])
```

The answer is 848. Using a 2017 Macbook Pro and Python 3, the code above runs in less than 8 seconds.

A5. Remember that since we want to reverse the cipher, this string is $S_1 S_2 \ldots S_n$ and we need to invert the rule given. Rewriting the rule gives $T_n = S_1; T_i = (S_{i+1} - S_i) \; mod \; 26$ for all $1 \leq i \leq n - 1$. Sample code that implements the solution:

```
S = "EXEIXXPHJXAEMEJRMQJFTXFLSLQEYPCKXBPC"
n = len(S)

def char_to_index(c):
      return ord(c) - ord('A')
def index_to_char(i):
      alphabet =
["A","B","C","D","E","F","G","H","I","J","K","L","M","N","O","P","Q","R","S","T
","U","V","W","X","Y","Z"]
      return alphabet[i]

# Iterate backwards from n-2 .. 0
T = S[0]
for i in range(n-2, -1, -1):
      Ti = index_to_char((char_to_index(S[i+1]) - char_to_index(S[i])) % 26)
      T = Ti + T
print(T)
```

The decrypted string reads "THEPASSCODEISFIVETWOEIGHTFOURNINEONE", so the answer is 528491. Fun fact: Those names are two major characters in the 2010 movie Inception, where this number comes from.

B0. Alien Colonies

Calculate this by finding the number of ship-temperature ranges the ship's current temperature is away from the ship-temperature lower bound and adding the same number of planet-temperature ranges to the planet-temperature lower bound.

Sample code:

```
import sys

def ship_temp_to_alien_temp(s_low, s_high, t, p_low, p_high):
    # - The number of temperature ranges t is away from s_low
    #   (meaning e.g. s_high is 1 temp range away,
    #   (s_low + s_high) / 2 is 0.5 temp ranges away)
    # - Can be negative
    num_temp_ranges = (t - s_low) / (s_high - s_low)

    # - Solution for the temp p such that
    #   (p - p_low) / (p_high - p_low) = num_temp_ranges
    # - "p is the same relative distance away from p_low
    #   that t is away from s_low"
    return num_temp_ranges * (p_high - p_low) + p_low

for line in sys.stdin:
    inputs = line.split(" ")

    s_low = float(inputs[0])
    s_high = float(inputs[1])
    t = float(inputs[2])
    p_low = float(inputs[3])
    p_high = float(inputs[4])

    print(ship_temp_to_alien_temp(s_low, s_high, t, p_low, p_high))
```

B1. Math is a four-letter word

The first two conditions are already taken care of. Check the third condition first, and then if that passes, iterate over characters in both words. When letters in the words differ at some position, set a Boolean flag to true. If more letters differ and the flag is set, that means more than one letter differs.

Sample code:

```
import sys

def is_valid_step(S, F):
    if len(S) != len(F):
        return "invalid C"
    if S == F:
        return "invalid D"

    # If we reached this point, then S and F:
    #  1. are the same length
    #  2. differ by at least one letter

    # Set some initial values
    found_different_letter = False
    different_letter_index = -1
    for i in range(len(S)):
        if S[i] != F[i]:

            # We already know we'll enter this check at least once
by (2) above.
            # We want to make sure we enter it exactly once.
            # If the flag is already set at this point, then we
entered it more than once, which means more than one letter differs

            if found_different_letter:
                return "invalid D"
            found_different_letter = True
            different_letter_index = i

    # If we reach this point, the step is valid
    return "valid " + str(different_letter_index)

for line in sys.stdin:
    inputs = line.split(" ")

    S = inputs[0].strip()
    F = inputs[1].strip()

    print(is_valid_step(S,F))
```

B2. Rover Lover

A mathematically sound solution might be to multiply the fraction n/d by 10^{k+2}, then take that value mod 1000 and truncate the fractional part to leave the three digits we want.

The main challenge here is maintaining precision on a computer. On a typical 64-bit machine, this solution works for small k, roughly $k < 15$ or so. As k gets larger, floating-point arithmetic imprecisions start to give incorrect answers. A better solution generates each decimal digit by basically implementing long division, where we multiply the numerator by 10, integer-divide to get the next decimal digit, and keep around the remainder as the next numerator. For example, calculating each digit of $4/7$ manually:

$\frac{4}{7} = \left(\frac{1}{10}\right)\left(\frac{40}{7}\right) = \left(\frac{1}{10}\right)\left(5 + \frac{5}{7}\right)$ -> The first digit of the expansion is $5 = (4 * 10)/7$, setting aside the remainder $(4 * 10)\ \%\ 7 = 5$. The rest of the expansion is just the expansion of $5/7$ but shifted to the right by one decimal place. Now, $\frac{5}{7} = \left(\frac{1}{10}\right)\left(\frac{50}{7}\right) = \left(\frac{1}{10}\right)\left(7 + \frac{1}{7}\right)$ -> The second digit of the expansion is $7 = (5 * 10)/7$, setting aside the remainder $(5 * 10)\ \%\ 7 = 1$.

This makes the final decimal expansion $\frac{4}{7} = \left(\frac{1}{10}\right)\left(5 + \left(\frac{1}{10}\right)\left(7 + \frac{1}{7}\right)\right) = \frac{5}{10} + \frac{7}{100} + \left(\frac{1}{100}\right)\left(\frac{1}{7}\right)$

We can continue indefinitely in this manner.

This prevents any loss of precision. We keep track of the digits we've calculated in an array and return the last three. (We could keep just the last three digits if we want, but it's easier to manage and doesn't affect runtime or correctness if we just keep all of them, since $k < 10^6$.)

Sample code:

```
import sys
import math

def decimal_digit_sequence(n, d, k):
    digits = []

    for i in range(k+2):
        digit_i = (n * 10) // d
        n = (n * 10) % d
        digits.append(digit_i)

    return str(digits[len(digits)-3]) + str(digits[len(digits)-2]) +
str(digits[len(digits)-1])

for line in sys.stdin:
    inputs = line.split(" ")

    n = abs(int(inputs[0]))
    d = abs(int(inputs[1]))
    k = int(inputs[2])

    print(decimal_digit_sequence(n, d, k))
```