SAPO 2019 Round 2 - Solutions

Circular Primes

This problem was similar to the "Right-Truncatable Primes" problem from Round 1. Firstly, we need to define a function which calculates whether a number is prime or not. The constraints are small enough such that this can be done naively, without requiring a sieve. Pseudocode is given below:

```
def is_prime(n):
  i := 2
  while i*i <= n:
    if n mod i = 0:
      return False
    i := i + 1
  return True
```

To calculate whether a number n is a circular prime, we simply check whether n is prime, and then move one of it's digit from the leftmost position to the rightmost position (or vice versa) and then repeat until we arrive at the original number n. This can be done in various ways depending on the language used (e.g. in Python, one can simply do `m = int(str(m)[1:] + str(m)[0])`)

```
def is_circular_prime(n):
  m := n
  do:
    m := int(str(m)[1:] + str(m)[0])
    if m not prime:
      return False
  while (m != n)
  return True
```

Finally, we simply check each positive integer from 1 onwards until we've reached the n-th circular prime, which solves the problem.

Remark: One can check also determine if a number is a circular prime in a purely mathematical way by iterating either of the two following operations:

$$n \quad \to \quad \left\lfloor \frac{n}{10} \right\rfloor + (n \bmod 10) \cdot 10^{\lfloor \log_{10} n \rfloor}$$

or

$$n \quad \to \quad \left\lfloor \frac{n}{10^{\lfloor \log_{10} n \rfloor}} \right\rfloor + (n \bmod 10^{\lfloor \log_{10} n \rfloor}) \cdot 10$$

Roman Numerals

To convert from Arabic numerals to Roman numerals, the easiest would simply be to hardcode the symbols required for each digit in each of the units, tens, hundreds, and thousandths. Then, simply iterate through each of the digits of n and build up a Roman string based on each digit. Given the hardcoded array `symbols`, pseudo-code can be given as:

```
def arabic_to_roman(n):
  roman_num := ""
  for i from 1 to length of n:
    roman_num := roman_num + symbols[i, str(n)[i]]
  return roman_num
```

To convert back from Roman to Arabic, we can do an initial pass through the string, where we identify pairs of adjacent characters `...ab...` where the value of a is **less** than the value of b. We can store this in an integer as $b-a$, remove the two characters and repeat this process until all Roman character remaining are in decreasing value. Simply adding the remaining values yields the correct Arabic value.

Remark: The constraints are small enough such that one needs only to code up a function which convert from Arabic to Roman. To easily convert back, given some Roman string S, simply iterate through all positive integers from 1 onwards until reaching a value n which yields the string S when converting to Roman (this process will terminate assuming S is a valid Roman numeral).

```
def roman_to_arabic(s):
  i := 1
  while (arabic_to_roman(i) != s):
    i := i + 1
  return i
```

Magic Square

Determining whether a given $N \times N$ square is magic or not is simple enough. Simply calculate the sum of values in the first row, then check whether the sum of values in the remaining $N-1$ rows and N columns match the initially calculated value.

To determine whether a given square is **almost magic**, first calculate all row and column sums, and then determine the number of distinct sums obtained (either by sorting, or using a set). If one distinct sum is found, then the square is magic, otherwise if three or more distinct sums are obtained, then the square cannot be almost magic.

If two distinct sums are found, we then simply check if both the row sums and columns sums contains both values obtained, and furthermore that one of the obtained values has **exactly one** occurrence out of all N row sums and **one** occurrence out of all N column sums. If this is the case, the the square is almost magic.

Chess Game

We simply do a breadth-first search over all valid moves, keeping track of both the **position** and **number of moves** that the knight has made. At every position we visit, we then simply check if the knight has either captured the white pawn or forced a stalemate.

Specifically, if the knight is in position (c, r) and has made t moves, and the white pawn started at position (w_c, w_r), then the knight has captured the white pawn if $c = w_c$ and $r = w_r + t$. Otherwise, if $c = w_c$ and $r = w_r + t - 1$, the the knight has forced a stalemate.

All that remains is to keep track of the minimum number of moves required to obtain each of the three possible outcomes, and then output the final outcome accordingly.

Remarks: When doing a BFS, there's no reason to search further than N moves ahead, since if no capture or stalemate is found, then the white pawn will surely win. This results in a time complexity of $\mathcal{O}(N^3)$.