15-112, Week 2, Loops Practice

See how many of these problems you and a partner can solve during today's lab.

For each problem, you write the requested function *and* testcases for it. See `week01/example.py` in the demo code repository for an example of organizing this.

Rules for every problem below: no strings, no lists, and no recursion. You have loops, conditionals, arithmetic, and boolean logic — that's it. In particular, you must take numbers apart with `//` and `%`, not with `str()`.

---

1. nthAdditivePrime(n)

An *additive prime* is a prime number whose digits also sum to a prime. For example, `113` is prime, and `1 + 1 + 3 == 5`, which is also prime, so `113` is an additive prime. Write the function `nthAdditivePrime(n)` that takes a non-negative `int n` and returns the nth additive prime, starting with `n = 0`. You may assume the helper function `isPrime(n)` has already been written and is available to you — do not write it yourself, just call it.

---

2. longestIncreasingRun(n)

Write the function `longestIncreasingRun(n)` that takes a positive `int n` and returns the longest run of consecutive digits in `n` that strictly increase from left to right, as an integer. For example, `longestIncreasingRun(1232)` returns `123`, and `longestIncreasingRun(27648923679)` returns `23679`. If two runs tie for the longest, return the larger of the two, so `longestIncreasingRun(123345)` returns `345`. *Hint: you can build up a run from right to left as you peel digits off with `% 10`, but think carefully about when a run ends and what you need to remember about the best run seen so far.*

---

3. ascendingDigits(n)

Write the function `ascendingDigits(n)` that takes an `int n` and returns the integer made up of only those digits of `n` that ascend as you scan the number from right to left. Start at the rightmost digit and keep it. Then keep each digit that is strictly larger than the last digit you kept, skipping the rest.

For example, with `n = 643512`, the rightmost digit is `2`, so keep it. Moving left, the next digit larger than `2` is `5`, then the next digit larger than `5` is `6`. Everything else is discarded, so the answer is `652`. Negative values keep their sign. Here are some testcases:

```
assert(ascendingDigits(64354512) == 652)
assert(ascendingDigits(4239758345613) == 9863)
assert(ascendingDigits(759) == 9)
assert(ascendingDigits(0) == 0)
assert(ascendingDigits(-26434512) == -652)
```

---

4. countNarrowNumbers(a, b)

We'll say a positive integer is *narrow* if its digit sum is equal to its number of digits (written without leading zeros). So `1` is narrow, since it has one digit and a digit sum of `1`. `12` is not narrow, since it has two digits but a digit sum of `3`. `20` is narrow, since it has two digits and a digit sum of `2`. Write the function `countNarrowNumbers(a, b)` that takes two positive `int` values `a` and `b` and returns how many narrow numbers lie between them, inclusive. Here are some testcases:

```
assert(countNarrowNumbers(1, 20) == 3)     # 1, 11, 20
assert(countNarrowNumbers(1, 50) == 3)     # 1, 11, 20
assert(countNarrowNumbers(1, 300) == 9)    # 1, 11, 20, 102, 111, 120, 201, 210, 300
assert(countNarrowNumbers(100, 150) == 3)  # 102, 111, 120
```

---

5. nthDigitOfSquares(n) — challenge problem

Imagine writing out the perfect squares in order, one after another with no separators, to form an endless stream of digits:

```
1  4  9  16  25  36  49  64  81  100  121  144  169  ...
-> 1 4 9 1 6 2 5 3 6 4 9 6 4 8 1 1 0 0 1 2 1 1 4 4 1 6 9 ...
```

Write the function `nthDigitOfSquares(n)` that takes a non-negative `int` `n` and returns the digit at index `n` of that stream, where index `0` is the leading `1` of `1`. So index `9` is the `4` in `49`, and indices `15`, `16`, and `17` are the three digits of `100`.

Here are some testcases:

```
assert(nthDigitOfSquares(0) == 1)
assert(nthDigitOfSquares(2) == 9)
assert(nthDigitOfSquares(3) == 1)
assert(nthDigitOfSquares(9) == 4)
assert(nthDigitOfSquares(14) == 1)   # last digit of 81
assert(nthDigitOfSquares(15) == 1)   # first digit of 100
assert(nthDigitOfSquares(17) == 0)   # last digit of 100
assert(nthDigitOfSquares(20) == 1)   # last digit of 121
assert(nthDigitOfSquares(1000) == 4)
assert(nthDigitOfSquares(10**6) == 7)
```

5b. Making it fast (super challenge problem)

Your solution above almost certainly slows to a crawl well before `n` reaches `10**10`, because it produces the stream one square at a time. Rewrite it so that it answers essentially instantly even for `n = 10**12`:

```
assert(nthDigitOfSquares(10**12) == 4)
```

To do this you have to count digits in bulk instead of producing them. Group the stream into blocks by digit length: first every 1-digit square, then every 2-digit square, then every 3-digit square, and

so on. If you can compute how many digits an entire block contributes, you can skip whole blocks at a time and only ever examine a single square.

The squares with exactly `d` digits are the squares of the integers `k` with `10**(d-1) <= k**2 < 10**d - 1`, so counting them is an integer-square-root problem — recall problem 5 from last week's lab. Do not reach for `math.sqrt` or `v ** 0.5` here. Floats carry about 15 significant digits, and by the time `n` reaches `10**12` you are working with squares far past that, so a float square root will silently be off by one and quietly corrupt your block counts. Compute integer square roots using integer arithmetic only.