Algorithms and Complexity

Unit 1

David Emanuel Craciunescu

Exercise 3

```
T(n) = T(n/2) + 3n^2 + n
T(n) - T(n/2 = 3n^2 + n

n ≡ 2^n
    T(2^n) - T(2^(n-1)) = 3(2^n)^2 + 2^n

T(2^n) ≡ x^k
    (x^(k-1))(x-1) = 3(2^n)^2 + 2^n

X(h) => x = 1 => A
X(p) => x1 = 2^2, x2 = 2 => B(2^n)^2 + C2^n

X = X(h) + X(p) = A + B(2^n)^2 + C2^n
  = A + B(n^2) + Cn => O(n^2)
```

Exercise 5

Program a function to determine if a number received as a parameter is prime. Analyze the efficiency and complexity of the provided solution(s).

A prime number is a number greater than 1 that cannot be formed by multiplying two smaller natural numbers. As with all problems, there are many ways to implement this.

Naive Solution

We can calculate if a number is prime by simply iterating all the way up to that number and checking every element. This is quite naive. We're not taking into account many optimizations along the way. With this approach we would obtain an obvious complexity of **O(n) temporal and O(1) spatial**.

Implementation

```python
def is_prime_naive(number: int) -> bool:
    is_prime = True

    if number <= 1:
        is_prime = False
    else:
        current = 2
        while is_prime and current < number:
            is_prime = (number % current != 0)
            current += 1

    return is_prime
```

Optimized Solution

We could also take into account that **(1)** even numbers can't be primes, so we can skip them, and (2) if the number 2 divies a given number, that number divided by 2 also divides by number. Thanks to this we can iterate only until the square root of the number we're looking for, since further computation would be completely useless. With this approach we obtain complexities of **O(√n) temporal, and O(1) spatial**.

Implementation

```python
def is_prime_optimized(number: int) -> bool:
    is_prime = True

    if number <= 1:
        is_prime = False
    else:
        current = 3
        while is_prime and current <= math.sqrt(number):
            is_prime = (number % current != 0)
            current += 2

    return is_prime
```

Tests

The methods have been unit-tested against a whole set of numbers, both prime and non-prime. Here are some examples of the tests, followed by the result of their execution.

```python
# Naive
def test_is_prime_naive01(self):
    self.assertTrue(is_prime_naive(797581))

def test_is_prime_naive02(self):
    self.assertTrue(is_prime_naive(1049))

def test_is_prime_naive03(self):
    self.assertTrue(is_prime_naive(23))

# Optimized
def test_is_prime_optimized01(self):
    self.assertTrue(is_prime_optimized(797581))

def test_is_prime_optimized02(self):
    self.assertTrue(is_prime_optimized(1049))

def test_is_prime_optimized03(self):
    self.assertTrue(is_prime_optimized(23))
```

```
┌dave💻ec2 📁 algo 🌿master ❯ python3 -m unittest U1/test/test_e5.py
----------------------------------------------------------------------
Ran 16 tests in 0.095s - OK
```

Exercise 6

Program a function to determine if a number received as a parameter is perfect. Analyze the efficiency and complexity of the provided solution(s).

A perfect number is a number greater is a positive integer such that it is equal to the sum of its divisors. As with all problems, there are many ways to implement this.

Naive Solution

Just like we did with prime numbers, we can iterate all the way up to the number while checking every element. This is quite naive. We're not taking into account many optimizations along the way. With this approach we would obtain an obvious complexity of **O(n) temporal and O(n) spatial**. This is due to the fact that we need to store the divisors temporally along the way.

Implementation

```python
def is_perfect_naive(num: int) -> bool:
    return num == sum([current for current in range(1, num) if num % current == 0])
```

Optimized Solution

We could also take into account that **(1)** divisors present themselves in pairs when calculating, **(2)** the quotient can also count as a divisor when calculating, **(3)** we only really need to go up to the square root of the number, and **(4)** we can store the partial sum instead of occupying memory with a list of divisors. With this approach we obtain complexities of **O(√n) temporal, and O(1) spatial**.

With this we do have a small inconvenience, we sum all the divisors twice (since they come in pairs), but we easily mitigate that effect by iterating to the square root of the number we look for. A complexity of O(2√n), which is O(√n) in the end, still beats O(n).

Implementation

```python
def is_perfect_optimized(num: int) -> bool:
    sum_divisors = 0

    for current in range(1, int(math.sqrt(num))+1):
        # Is divisor of the number.
        if num % current == 0:
            sum_divisors += current

            # Quotient may also be a solution
            if num / current != current:
                sum_divisors += (num // current)

    return num*2 == sum_divisors
```

Tests

The methods have been unit-tested against a whole set of numbers, both prime and non-prime. Here are some examples of the tests, followed by the result of their execution.

```python
# Naive
def test_is_perfect_naive01(self):
    self.assertTrue(is_perfect_naive(6))

def test_is_perfect_naive02(self):
    self.assertTrue(is_perfect_naive(28))

def test_is_perfect_naive03(self):
    self.assertTrue(is_perfect_naive(496))

# Optimized
def test_is_perfect_optimized01(self):
    self.assertTrue(is_perfect_optimized(6))
```

```python
def test_is_perfect_optimized02(self):
    self.assertTrue(is_perfect_optimized(28))

def test_is_perfect_optimized03(self):
    self.assertTrue(is_perfect_optimized(496))
```

```
╭dave💻ec2 📁 algo 🌿master ❯ python3 -m unittest U1/test/test_e6.py
----------------------------------------------------------------------
Ran 16 tests in 0.015s - OK
```

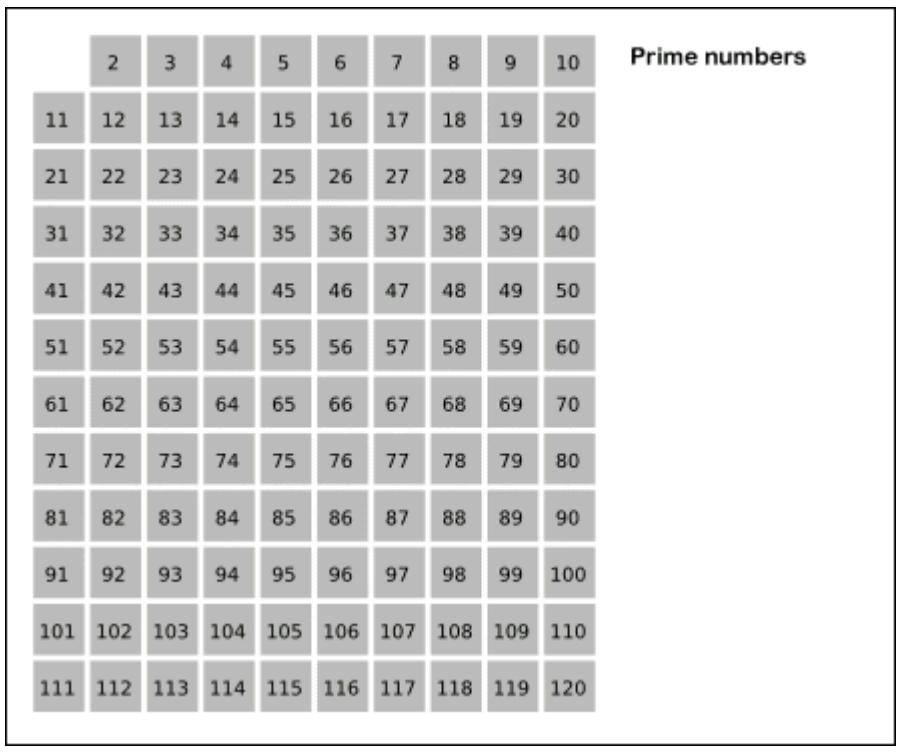
Exercise 7

Write a program that receives a positive number N and obtains the amount of prime numbers and the amount of perfect numbers there are between 1 and N. Analyze the efficiency and complexity of the provided solution.

For this we could reuse the already defined methods in e5 and e6, but that would yield a highly inefficient program. We could solve this problem in two different ways: **(1)** implementing the Sieve of Erastothenes, **(2)** tweaking the implementation of e5 and e6 in order to return amounts and not booleans.

The Sieve of Erastothenes

The Sieve of Erastothenes is an ancient algorithm for finding all prime numbers up to any given limit that works by iteratively calculating primes up to that number and storing in memory the multiples of those primes in order to skip them if found along the way. Thanks to this optimization, the Sieve of Erastothenes is widely regarded as one of the most efficient ways of finding small primes.



We’ve implemented a number of small optimizations along the way that help us reduce its time complexity even further, alghouth they’re at the cost of some more space usage. We’ve decided to use a list of “marked multiples” in order to speed up the process when checking for already calculated numbers. We can also start from the square of the number we’re looking for. With this we obtain a **time complexity of O(n), but also a space complexity of O(n)**, since we have to store our marked primes in memory.

Solution

```python
def amount_primes(num: int) -> int:
    # Generate detected primes.
    primes = [True for _ in range(num+1)]
    primes[0] = primes[1] = False # Clearly not primes
    current = 2

    # Check for unmarked numbers
    while current**2 <= num:
        # If not marked => prime
        if primes[current]:
            # Mark all multiples as non-primes
            for multiple in range(current**2, num+1, current):
                primes[multiple] = False

        current += 1

    # Return amount of true values in list
    return len([prime for prime in primes if prime])
```

Using previous implementation

We can reuse our previous optimized implementation of the algorithm that checks for perfect numbers and loop it to return amounts instead of booleans. Given that our previous complexity was O(√n) and we're iterating up to n, we would obtain a **time complexity of O(n√n), and a space complexity of O(1).**

Solution

```
def amount_perfects(num: int) -> int:
    counter = 0
    for num in range(1, num+1):
        if is_perfect(num):
            counter += 1

    return counter
```

Putting it all together

Now we simply have to put the two algorithms together, since that's what the exercise actually asks for.

```
def amount_primes_perfects(num: int) -> tuple:
    return (amount_primes(num), amount_perfects(num))
```

Tests

The tests are pretty straightforward, they've been reused from previous exercises and adapted to this specific case. Here are some examples followed by evidence of their execution.

```
# Primes
def test_amount_primes01(self):
    self.assertEqual(amount_primes(997), 168)

def test_amount_primes02(self):
    self.assertEqual(amount_primes(461), 89)
```

```
# Perfects
def test_amount_perfects01(self):
    self.assertEqual(amount_perfects(6), 1)

def test_amount_perfects02(self):
    self.assertEqual(amount_perfects(28), 2)
```

```
# Primes and perfects
```

```
def test_amount_perfect_primes01(self):
    self.assertEqual(amount_primes_perfects(997), (168, 3))

def test_amount_perfect_primes02(self):
    self.assertEqual(amount_primes_perfects(461), (89, 2))
```

```
┌dave💻ec2 📁 algo 🌿master ❯ python3 -m unittest U1/test/test_e7.py
----------------------------------------------------------------------
Ran 10 tests in 0.039s - OK
```

Exercise 8

Program a recursive procedure to obtain the inverse number of any given one. Example: 627 -> 726. Analyze the efficiency and complexity of the provided solution.

The approach for this exercise is simple. We iteratively strip away the number's last digit and add it to the inverse. Now, this method works, and the call stack proves it. I'm yet to find a way of making it work without using global variables, which are a terrible antipattern. I'd love some help on this problem. Thanks in advance.

The time complexity of this approach is clearly **O(n)**, since we must iterate over all numbers once no matter what. The space complexity is **O(n)** as well, since we must store at least the number of iterations as calls on the stack.

Solution

```
def inverse_aux(number, result=0):
    if number > 0:
        reminder = number % 10
        result = (result*10) + reminder
        inverse_aux(number // 10, result)

    return result

def inverse(number):
    return inverse_aux(number)
```

Tests

The tests I've designed in order to check if it works are the following. They clearly don't check out.

```
def test_inverse01(self):
    self.assertEqual(inverse(1234), 4321)

def test_inverse02(self):
    self.assertEqual(inverse(1000), 1)

def test_inverse03(self):
    self.assertEqual(inverse(1), 1)

def test_inverse04(self):
    self.assertEqual(inverse(1230), 321)
```

Exercise 9

Program a recursive function to calculate the following sum: S = 1 + 2 + 3 + 4 + (...) + n-1 + n.

Analyze the efficiency and complexity of the provided solution.

This might be one of the simplest examples of recursion. The time complexity we obtain is **O(n)**, since we're basically performing a for operation recursively, and the space complexity we obtain is **O(n)** as well, since we have store all the calls on the stack.

Solution

```
def summation(num):
    if num <= 1:
        return num
    return num + summation(num - 1)
```

Tests

As straightfoward as they can get.

```
def test_summation01(self):
    self.assertEqual(summation(10), sum(range(10+1)))

def test_summation02(self):
    self.assertEqual(summation(100), sum(range(100+1)))

def test_summation03(self):
    self.assertEqual(summation(300), sum(range(300+1)))
```

```
┌dave💻ec2 📁 algo 🌿master ❯ python3 -m unittest U1/test/test_e9.py
----------------------------------------------------------------------
Ran 3 tests in 0.000s - OK
```