CP2410 Assignment

Name: Sihan Chen

Student ID: 14187662

Question 1

Slow solution

```
def find_the_smallest_evenly_divisible_slow(n: int):
#     upper limit would be n factorial
    factorial = 1
    for i in range(1, n+1):
        factorial *= i
    for i in range(n+1, factorial):
        isResult = True
        for j in range(1, n+1):
            if i % j != 0:
                isResult = False
                break
        if isResult:
            return i
```

```
find_the_smallest_evenly_divisible_slow(10)
```

```
2520
```

Explanation:

First, we calculate and set an upper limit -- n factorial, which gurantees a smaller evenly divisible exist.

Then, we run a loop from **n + 1 to n!**, and for each variable `i` , we will execute a second inner loop using variable `j` from **1 to n**, and calculates the reminder of `i % j`. And if all `j` pass the test, we know `i` is the smallest evenly divisible number.

```
def find_the_smallest_evenly_divisible_slow(n: int):
    factorial = 1                        # O(1)
    for i in range(1, n+1):              # O(n)
        factorial *= i                   # O(1)
    for i in range(n+1, factorial):      # O(n! - n)
        isResult = True                  # O(1)
        for j in range(1, n+1):          # O(n)
            if i % j != 0:               # O(1)
                isResult = False         # O(1)
                break
        if isResult:                     # O(1)
            return i
```

$$\because \text{In terms of time complexity, } (n! - n) \text{ is still considered as } n$$

$$\therefore \text{total time cost is: } O(n + (n! - n) \cdot n) \Rightarrow O(n^2)$$

Fast solution

```
def get_primes(n):
    result = [True] * (n + 1)
    # 0 and 1 are not prime number
    result[0] = result[1] = False
    for i in range(2, int(n**(0.5))+1):
        if result[i]:
    # number i is a prime number, remove i**2+n*i
            for j in range(i**2, n+1, i):
                result[j] = False
    return [i for i in range(2, n+1) if result[i]]


def find_the_smallest_evenly_divisible_fast(n: int):
    primes = get_primes(n)
    print(primes)
    factors = {}

    for prime in primes:
        factors[prime] = 1

    squart_root = int(n**0.5)
    print(squart_root)
    i = 0
    while primes[i] <= squart_root:
```

```
        i += 1
    first_half_for_check = primes[:i]

    # find power for first half
    for num in first_half_for_check:
        power = 0
        while num**power < n:
            power += 1
        factors[num] = power - 1

    # for second half, th
    print(factors)
    result = 1
    for k, v in factors.items():
        result *= (k**v)
    return result
```

```
find_the_smallest_evenly_divisible_fast(10)
```

```
[2, 3, 5, 7]
3
{2: 3, 3: 2, 5: 1, 7: 1}
2520
```

Explanation:

Time Complexity Analysis:

Question 2

Slow solution

```
def find_nth_primes_slow(n: int) -> int:
    """ Return n'th prime number. """
    primes = []
    # start with 2, since 0 and 1 are not prime numbers
    i = 2
    while len(primes) < n:
        isPrime = True
        for j in range(2, int(i**(0.5)+1)):
            if i % j == 0:
                isPrime = False
```

```
        if isPrime:
            primes.append(i)
        i += 1
    # exit while loop when collect enough prime numbers,
    # and the last one would be n'th prime
    print(primes)
    return primes[-1]
```

```
# find sixth prime number, expected value: 13
find_nth_primes_slow(6)
```

```
[2, 3, 5, 7, 11, 13]
13
```

Explanation:

Firstly, we need to know how to check if a number is a prime number.

Let's assume if **n is not a prime**, meaning

$$\exists n = a \cdot b$$

$$\Rightarrow \forall n \text{ ,where } n = a \cdot b \begin{cases} a = b = \sqrt{n} \\ (\exists a < \sqrt{n}) \vee (\exists b < \sqrt{n}) \end{cases}$$

$$\Rightarrow \text{For all non-prime number } n \text{, one of its divisor must be smaller or equal to } \sqrt{n}$$

$$\Rightarrow \text{if } n \% m \neq 0 \text{ for } m \text{ from 2 to } \sqrt{n} \text{ (inclusive), then } n \text{ is a prime number}$$

And to find n'th prime, we will use a variable `i` with value 2 and increment by 1 for each while loop iteration. Then, we check whether `i` meets the critiria for a prime number, and store it inside list `primes` if i is a prime.

The while loop will stop when `len(primes)` reaches `n`. And at that point, the last element in `primes` is the n'th prime.

Time Complexity Analysis:

```
def find_nth_primes_slow(n: int) -> int:
    primes = []
    i = 2
    while len(primes) < n:                          # O(n)
        isPrime = True                              # O(1)
        for j in range(2, int(i**(0.5)+1)):         # O(n**0.5)
            if i % j == 0:                          # O(1)
```

```
                isPrime = False                       # O(1)
        if isPrime:                                   # O(1)
            primes.append(i)                          # O(1)
        i += 1                                        # O(1)
    return primes[-1]
```

$$\therefore \text{Total time cost: } O(n \cdot \sqrt{n} + 6) \Rightarrow O(n^{1.5})$$

Fast solution

```
def find_nth_prime_fast(n):
    result = [True] * (n + 1)
    # 0 and 1 are not prime number
    result[0] = result[1] = False
    for i in range(2, int(n**(0.5))+1):
        if result[i]:
          # number i is a prime number, remove i**2+n*i
          for j in range(i**2, n+1, i):
            result[j] = False
    return [i for i in range(2, n+1) if result[i]]
```

Explanation:

To find n's prime number, first we need to know how to determine if a number is a prime number.

To do that, we will create a list called `result`, with n + 1 slots. Each element is representing the integer number equals to its index value, so in total we have integer from 0 to n(inclusive). Initially, all values are set as **true**. And we will turn the boolean value from true to false if we prove the number at that index is not a prime.

For example, [0, 1, 2, 3, 4, 5, 6, 7, ...]:

$$[\text{false, false, true, true, false, true, false, true, ...}]$$

Since we know integer 0 and 1 are not prime numbers, we set result[0] and result[1] as **false**.

Let's assume if **n is not a prime**, meaning

$$\exists n = a \cdot b$$

$$\Rightarrow \forall n \text{ ,where } n = a \cdot b \begin{cases} a = b = \sqrt{n} \\ (\exists a < \sqrt{n}) \vee (\exists b < \sqrt{n}) \end{cases}$$

$$\Rightarrow \text{For all non-prime number } n\text{, one of its divisor must be smaller or equal to } \sqrt{n}$$

And from 2 to $\sqrt{n}$ (inclusive)

Time Complexity Analysis:

Question 3

Slow solution

```
def find_pythagorean_triplet_slow(n:int) -> tuple:
    """ Return tuple of (a, b, c) where three integers constitue pythagorean triplet.
        Return None if no triplet is found. """
    for c in range(5, n+1):
        for b in range(4, c):
            for a in range(3, b):
                if a**2 + b**2 == c**2 and a + b + c == n:
                    return (a, b, c)
    return None


# example: find Pythagorean triplet for n = 1000
print(f"For n = 1000, the Pythagorean triplet is: {find_pythagorean_triplet_slow(1000)}")
```

```
For n = 1000, the Pythagorean triplet is: (200, 375, 425)
```

Explanation:

Here we start loop for a, b, c from 3, 4, 5 as 3, 4, 5 is the smallest Pythagorean triplet.

By defination, a Pythagorean triplet a, b, c means $a < b < c$ and $a^2 + b^2 = c^2$. So to find such triplet, we will start looping with variable `c` from 5 to n+1, since it is the biggest number.

For each c we run another loop with variable `b` from 4 to c, and do the same for varaible `a` from 3 to b. Then, we calculate if such a, b, c combination fits critiria, and return tuple `(a, b, c)` if they meet standard.

Time Complexity Analysis:

```
def find_pythagorean_triplet_slow(n:int) -> tuple:
    for c in range(5, n+1):                                   # O(n)
        for b in range(4, c):                                 # O(n)
            for a in range(3, b):                             # O(n)
                if a**2 + b**2 == c**2 and a + b + c == n:    # O(2)
                    return (a, b, c)
    return None
```

```
    return None
```

$$\therefore \text{The total time cost is: } O(n^3+2) \Rightarrow O(n^3)$$

Fast solution

```
def find_pythagorean_triplet_fast(n:int) -> tuple:
    for c in range(5, n//2):
        for b in range(4, c):
            a = n - c - b
            if a**2 + b**2 == c**2 and a + b + c == n:
                    return (a, b, c)
    return None


# example: find Pythagorean triplet for n = 1000
print(f"For n = 1000, the Pythagorean triplet is: {find_pythagorean_triplet_fast(1000)}")
```

```
For n = 1000, the Pythagorean triplet is: (375, 200, 425)
```

Explanation:

To optimize the previous function, we can apply the knowledge of trigonometry: if a, b, c are Pythagorean triplet, then `c` must be smaller than `a + b`. And this allow us to narrow down the search range of `c` to [5, n//2).

As for a and b, which one is bigger should not concern us, so we can assign either `a` or `b` for the second loop, and just caculate the other one using substitution, knowing $a+b+c=n$.

Lastly, the condition check remain the same.

Time Complexity Analysis:

```
def find_pythagorean_triplet_fast(n:int) -> tuple:
    for c in range(5, n//2):                                # O(n)
        for b in range(4, c):                               # O(n)
            a = n - c - b                                   # O(1)
            if a**2 + b**2 == c**2 and a + b + c == n:      # O(2)
                    return (a, b, c)
    return None
```

$$\therefore \text{Total time cost: } O(n^2)$$