Exercise 3.

```
1. fun Calculation (x, y, z: integer) return value: integer
  2. var i, j, t: integer
         value = 0
  3. For i = x to y do value = value + i efor
  4. If (value/(x+y)) <= 1 then return z
  5. Else
      6. t = x + ((y - x)/2)
      7. For i = x to y do
          8. For j = (3*x) to (3*y) do
              9. value = value + Minimum(i, j)
          10. efor
      11. efor
      12. value = value + 4 * Calculation(t, y, value)
      13. return value
  14. eif
15. efun
```

Lines 1–2 (Declaration and Initialization):
– The function is defined, and variables are declared with the initial assignment *value = 0*.
– This initialization is an elementary operation with constant time complexity, O(1)
Line 3 (For Loop for Summation):
– The loop iterates from *i* = *x* to *y*, summing each value into *value*.
– The number of iterations is roughly *n = y - x + 1*, which gives a cost of O(n).
Line 4 (Conditional Statement):
– The condition *(value/(x+y)) ≤ 1* is evaluated.
– This evaluation takes constant time, O(1).
– If the condition is true, the function immediately returns *z*, meaning the algorithm stops after the summation loop, resulting in an overall cost of O(n) for this best-case scenario.

Lines 5–15 (Else Block – Recursive Case):

1. **Line 6:**
   – The computation *t = x + ((y - x)/2)* is an arithmetic operation, which is O(1).
2. **Lines 7–11 (Nested Loops):**
   – **Outer Loop (Line 7):** Iterates from *i* = *x* to *y*, approximately O(n) iterations.
   – **Inner Loop (Line 8):** Iterates from *j = 3*x* to *3*y*. The number of iterations remains O(n).
   – **Operation (Line 9):** Within the inner loop, the operation *value = value + Minimum(i, j)* is assumed to be O(1).
   – **Total Cost of Nested Loops:** By applying the **Product Rule,** the cost is O(n) × O(n) = $O(n^2)$.
3. **Line 12 (Recursive Call):**
   – The algorithm updates *value* by adding **4 times** the result of a recursive call: *Calculation(t, y, value).*
   – **Recursive Parameter Reduction:** The parameter *t* is computed as *x* + *((y - x)/2)*, meaning the new interval [t, y] has size:
   $n' = y - t = y - (x + (y - x)/2) = (y - x) - (y - x)/2 = (y - x)/ = 2/n$

   – This yields the recurrence relation for the worst-case scenario:
   $T(n) = 4*T(n/2) + O(n^2)$

4. **Line 13:**
   – Returning the final *value* is an O(1) operation.

By applying the master's theorem: T(n) = K*T(n/b) + O(n^p); k = b^p; **$4 = 2^2$**

T(n)=O(n^2log(n))

Exercise 7

Explanation:

First, we iterate through every number between 1 and N. For each of those numbers, we check if it's prime and/or perfect.
To check if it's prime, we go from 1 to the square root of the current number (*why the sqrt(i)? Because any number higher than sqrt(i) must be paired with a smaller factor, therefore being irrelevant to check)* And check if any of those numbers are primes by making the division and checking its result.
To check if it's a perfect number we, first, start with a value of 1, as 1 is always a divisor, and then, we iterate from 2 to the sqrt(i)

Code:

```
from math import sqrt

# Input and validation
n = int(input("Enter a number: "))  # Get user input
Prime_n = 0  # Counter for prime numbers
Perfect_n = 0  # Counter for perfect numbers

# Ensure the number is positive
while n < 0:
    print("Please enter a positive number")
    n = int(input("Enter a number: "))

# Loop from 1 to N
for i in range(1, n+1):
    # ------ Prime Number Check ------
    is_prime = True  # Assume number is prime
    for num in range(2, int(sqrt(i)) + 1):  # Check divisibility up to sqrt(i)
        if i % num == 0:
            is_prime = False  # Not a prime
            break  # Stop checking further
    if is_prime and i > 1:  # If number is prime, increment count
        Prime_n += 1

    # ------ Perfect Number Check ------
    sum_divisors = 1  # 1 is always a divisor (excluding itself)
    for div in range(2, int(sqrt(i)) + 1):  # Check divisibility up to sqrt(i)
        if i % div == 0:
            sum_divisors += div  # Add divisor
            if div != i // div:  # Check if the paired divisor is not the same as div
                sum_divisors += i // div
    if sum_divisors == i and i != 1:  # Check if sum of divisors equals the number
        Perfect_n += 1

# Output results
print("Number of prime numbers: ", Prime_n)
print("Number of perfect numbers: ", Perfect_n)
```

Test case:

Enter a number: 28
Number of prime numbers: 9
Number of perfect numbers: 2

Complexity:

- We have one general loop that goes through all numbers: O(n)
- Inside of it, in order to solve the prime and perfect number calculations we have two loops, which have a complexity of O(n^½)

Therefore, knowing that all other operations have a complexity of O(1) the code will have a complexity of **O(n^3/2)**

Exercise 9

Explanation:

We initialize the sum to 1, then we check if the current number is 0 in order to exit correctly the function. If not 0, we sub the number to the factorization of the next lower one. At the end we reach 0 and the sum value is returned with the correct value.

Code:

```
n = int(input("Enter a number: "))

def fact_sum(n):
    sum = 1 #initialize sum to 1
    if n==0:
        return sum #if n is 0, return the sum so we don't multiply by 0
    else:
        sum=n*fact_sum(n-1) #recursive call to multiply n by the sum of n-1
    return sum

print(fact_sum(n))
```

Test case:

Enter a number: 6
720

Complexity analysis:

The factorial operation goes: T(n)=n*T(n-1)

n*(n-1)*(n-2)*(n-3)...

The complexity will be: **T(n) = O(n)**