Q3

In order to have a higher precision, we could replace the need to do a lot of addition with very small numbers with needing to do addition with larger numbers and later just divide the output by some coefficient since we know that computers do multiplication and division to a higher degree of accuracy that addition and subtraction.

So another way of computing this sum is by rewriting it as $1 + \frac{1}{4}\left(1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \frac{1}{25} + \cdots\right)$
$+ \frac{1}{9}\left(1 + \frac{1}{9} + \frac{1}{25} + \frac{1}{49} + \frac{1}{81} + \cdots\right) + \frac{1}{25}\left(1 + \frac{1}{25} + \frac{1}{49} + \frac{1}{121} + \frac{1}{169} + \cdots\right) + \cdots$

When we expand the above expression, what we get is $\sum_{i=1}^{\infty} \frac{1}{i^2}$. So the summation remains unchanged but the precision is achieved by computing this sum n a different way. However, there are two man things that must be noted bout the above expression.
First, the denominators of all coefficient of parenthesis are squares of prime numbers.
Second, all terms in a given parenthesis are co-prime relative to the coefficients of all the parenthesis that came before.
Both of these conditions have been set in order to ensure that there are no duplicate terms in the expression.

The first code below is the ordinary summation

```
# finding the machine epsilon
x = 1
i = 0
while x > 0:
    x = x/2
    i += 1
mach_epsilon = 1/2**(i-1)

from math import pi

def s1(mach=mach_epsilon):
    i = 2
    total = 0
    term = 1
    while term > mach:
        total += term
        term = 1/i**2
        i += 1
    return i, pi**2/6, total,  (pi**2/6-total)/((pi**2/6)

print(s1()) # the default parameter of S1 is the
# machine epsilon, however setting the input at 1/100000000**2
# already gives an accuracy of about 9e-9
```

While the code below gives us the calculation of the expression, however, it's quite inefficient

```
import numpy as np
from math import sqrt, pi


def isprime(n):
    """Returns True if n is prime."""
    if n == 2:
        return True
    if n == 3:
        return True
    if n % 2 == 0:
        return False
    if n % 3 == 0:
        return False
```

```
    i = 5
    w = 2

    while i * i <= n:
        if n % i == 0:
            return False

        i += w
        w = 6 - w

    return True

mach_epsilon = np.finfo(float).eps

def no_diviser(x1, x2):
    # Checks if x2 divides x1
    out = x1/x2
    if out-int(out) != 0:
        return True
    return False

def no_con_with_coeffs(used_coeffs, num):
    for coeff in used_coeffs:
        # roudng because 1/num and 1/coeff must be intergers
        if not no_diviser(round(1/num), round(1/coeff)):
            return False
    return True

def next_prime(coeff, prime=isprime):
    """Given 1/p^2 where p is prime
    finds the next prime number after p,
    say q, and returns 1/q^2"""
    i = 1
    denom = round(1/sqrt(coeff))
    next = denom + i
    while not prime(next):
        i += 1
        next = denom + i
    return 1/next**2


def s2(epsilon=mach_epsilon, isprime=isprime, coprime=no_diviser, \
        no_con_with_coeffs=no_con_with_coeffs, next_prime=next_prime):
    """Summs up Sigma_{n=1}^{infty} 1/n^2"""
    used_coeffs = []
    i = 2
    e = 1
    i_terms = 1 # expression starts from 1
    total = 1 # expression starts from 1
    coeff = 1/i**2
    # multiplies the coefficient to the bracket
    # sum and adds to the total
    while coeff >= epsilon:
        e = 1
        i_terms += 1
        term = 1/i**2
        bracket_sum = 1 # starts from 1 then goes to the coeff value

        # Sums the terms in the bracket until saturation
```

```
        while coeff*term >= epsilon:
            if no_con_with_coeffs(used_coeffs, term):
                bracket_sum += term
                i_terms += 1

            term = 1/round((i+e)**2) # rounding because we
                                     # know it must be integer
            e += 1

        total += coeff*bracket_sum
        used_coeffs.append(coeff) # filling the list of used coeffs
        i += 1
        coeff = next_prime(coeff) #gives the next coefficient

    return i_terms, pi**2/6, total, (pi**2/6-total)/(pi**2/6)

print(s2()) # the default is set at the machine epsilon
            # however, it may take a very long amount of
            # time for both algorithms to get to the result
```

For small numbers such as $1/10000000^2$ the precision of the suggested algorithms is even though noticeable ($\frac{s-s1}{\frac{\pi^2}{6}} = \frac{1.0000098 \times 10^{-7}}{\frac{\pi^2}{6}}$ versus $\frac{s-s1}{\frac{\pi^2}{6}} = \frac{9.999998 \times 10^{-8}}{\frac{\pi^2}{6}}$), but doesn't justify the inefficiency. But as the terms approach the machine epsilon, the inaccuracies of the first algorithm will become more glaringly obvious while the accuracy of the second method will not be worsened as much as the accuracy of the first.

It's possible to calculate the numbers of terms in the straight way as we just take the the square root of the reciprocal of machine epsilon and round down to the nearest integer. This works as every term corresponds to the square of 1/integer. And the lowest possible 1/integer is near the machine epsilon (if it isn't the machine epsilon itself).

The number of terms in the second methods is going to be equal to the number of terms in the first example as when we expand the expression for the second, we get the same summation as in the first method. The number of parenthesis is going to be significantly lower than the number of terms in the first method, however, the number of terms in defined by the total number of the terms that are in parenthesis, which is the same as in the first method.