# C++ code kata: Week #2

Hello 👋

If you have attempted or not, but here are the solution 🎁 for you.

To start with, if you are coming from plain C/ C++ 🌐 then you may have written your **First** solution like this 🚀

## Solution 1

```
#include <assert.h>

int countDigit_010(int nNum, int nDigit)
{
    assert(nNum >= 0);
    assert(0<=nDigit<=9);
    int nDigitCount = 0;

    while (nNum > -1)
    {
        int nSqr = nNum * nNum;
        do
        {
            if (nSqr % 10 == nDigit)
                ++nDigitCount;
            nSqr /=10;
        }
        while(nSqr);
        --nNum;
    }
    return nDigitCount;
}
```

### Solution 1: Commentry

We have checked some assertions here to make sure we are within limits of the algorithm.

Next, we have used the **while** loop as being used by any beginning developer. The code is mostly self-explanatory wherein we are picking an individual digit from the number and matching it against target digit.

Certain times to compute a repetitive task some developers prefer **Recursion** over iteration. Although it leads to a solution, certainly it becomes a more expensive solution to go for. Our second solution makes use of **Recursion** to create a solution.Here we go 🚀

## Solution 2

```
#include <assert.h>

int splitCount(int nNum, int nDigit){
    if(nNum)
        return (nNum % 10 == nDigit) + splitCount(nNum / 10, nDigit);
    return (nNum == nDigit);
}

int countDigit_020(int nNum, int nDigit)
{
    assert(nNum >= 0);
    assert(0<=nDigit<=9);
    int nDigitCount = 0;
    while (nNum > -1)
    {
        nDigitCount += splitCount(nNum*nNum, nDigit);
        --nNum;
    }
    return nDigitCount;
}
```

### Solution 2: Commentry

The interesting piece of code here is `splitCount` function that recursively splits the number into digits and returns summation. One clever code is **use of equality operator (==)** that not only check equality but return 1 or 0. **Although I discourage such solutions, its worth to show them**.

The **Standard Library of C++** has many useful utility collections that we can use to create bit quirky solutions. Although these may be less expensive than recursion but consider using them only with a bunch of data. So here's to solution ✈

## Solution 3

```
#include <assert.h>
#include <algorithm>
#include <vector>

int countDigit_030(int nNum, int nDigit)
{
    assert(nNum >= 0);
    assert(0<=nDigit<=9);
    std::vector <int> digits;
    auto matcher = [nDigit](int x)  { return x == nDigit;};
    while(nNum > -1)
    {
        int nSqr = nNum * nNum;
        do
        {
            digits.push_back(nSqr%10);
            nSqr /=10;
        }
        while(nSqr);
        --nNum;
    }
    return std::count_if(digits.begin(), digits.end(), matcher);
}
```

## Solution 3: Commentry

What we have done here is to put **Individual digits of the square of all numbers** into a **vector**. Vector is a better choice here rather than a simple array of integers because we do not know the count of digits that we have to handle.
To count the number of matching digits in this big fat vector we have used Standard C++ algorithm **count_if** that relieves not only looping but the conditional statement also. We have used a **lambda expression** to write the comparison code because it serves the purpose of a function object beautifully here.

Our next solution is basically near to our last solutions but to some extent augments the Solution 3. So here we go 🚀

## Solution 4

```
#include <assert.h>
#include <algorithm>
#include <vector>
#include <string>

int counter(std::string szDigits, char cDigit)
{
    return std::count(szDigits.begin(), szDigits.end(), cDigit);
}

int countDigit_040(int nNum, int nDigit)
{
    assert(nNum >= 0);
    assert(0<=nDigit<=9);

    int nDigitCount = 0;
    std::vector <std::string> digits;
    char szbuf[10] = {};
    while(nNum > -1)
    {
        itoa(nNum * nNum, szbuf, 10);
        digits.push_back(szbuf);
        --nNum;
    }
    for (auto &str:digits)
       nDigitCount += counter(str, (char) nDigit + 48);

    return nDigitCount;
}
```

## Solution 4: Commentry

Instead of comparing digits we took a different approach here and converted the all the numbers to strings using **itoa**. Further we have used **count** algorithm to count the target digit. This solution is not preferred for small numbers but may be helpful in large values.

The test script for the problem can be **found here**