Chapter 25. Sample Programming Exam – Topic #2

In This Chapter

In this chapter, we will take a look at the specifications of a **few practical algorithmic problems** from a sample programming exam, and we will offer solutions. While solving the problems, we will follow the guidelines from the chapter "Problem Solving Methodology", and we are going to illustrate their implementation.

Problem 1: Counting the Uppercase / Lowercase Words in a Text

Write a program that **counts the words in a text** entered from the console. The program must output the **total number of words**, the number of words written in **uppercase** and the number of words written in **lowercase**. If a word appears more than once in the text, each repetition counts as a new occurrence. Every character that is not a letter counts as a word separator.

Sample input:

```
Welcome to your first programming exam! Can you think of a
solution to this problem and write it down? GOOD LUCK!
```

Sample output:

```
Word count: 21
Upper case words: 2
Lower case words: 17
```

Coming Up with an Appropriate Idea for a Solution

Intuitively, it comes to mind, that the problem may be solved by **splitting the text up into separate words and counting** those that meet the specified conditions.

Obviously, this approach is correct, but far too general, and it doesn't lead to a particular method for solving the problem. Let's try to **be more specific**, and see if by doing so, we could implement an algorithm that will lead to a

solution. It might turn out that the implementation is difficult, or that the complexity of the solution is too great for the program to complete its execution even with today's powerful computers. If that is the case, we would have to find another solution to the problem.

Breaking Down the Problem into Subproblems

A useful approach for solving algorithmic problems is to try **breaking them down into smaller problems** that are easier and quicker to solve. Let's try defining the necessary steps for solving this problem.

First of all, we have to **split the text up into separate words**. This, in and of itself, is not a simple task, but it is the first step towards breaking down the problem into smaller, although still complicated, subproblems.

Then we need to **count the words** that concern us. This is the second major problem we have to solve. Let's take a look at both problems separately and try breaking them down even further.

How Do We Split the Text Up into Separate Words?

In order to split the text up into separate words, we need to find a way to identify them first. According to the problem specifications every non-letter character functions as a word separator. Therefore, we must first **identify these separators** and use them to split the text in tokens.

So far, we have formulated two subproblems – **finding the separators** and **partitioning (splitting) the text** in accordance with the characters found. We can implement their solutions right away. This was in fact our goal from the start – breaking down complicated problems into smaller and easier subproblems.

In order to **find the separators**, all we need to do is iterate through all characters and extract those that aren't letters.

Once we have identified the separators, we can implement the text partitioning by invoking the **`Split(…)`** method of the **`String`** class.

How Do We Count the Words?

Let's assume we already have a list of all words from the text. We want to **find the total word count, the number of words in uppercase and the number of words in lowercase**.

To do this, we can go through each and every word from the list and check if it meets either of the necessary conditions. At each step we **increment the total word count**. We check if the current word is in uppercase and, if so, we **increment the number of words in uppercase**. Likewise, we check if the word consists only of lowercase letters and **increment the lowercase word counter**.

Thus, we have defined another **two subproblems** – recognizing uppercase and lowercase words. These appear to be very easy. It might even turn out

that the **string** class provides such functionality. After we check, it turns out this is not the case. Yet we notice that there are methods that allow us to convert a string to an uppercase or a lowercase string. This might be of use.

To check if a word consists only of uppercase letters, all we have to do is compare it to the string resulting after converting the word to uppercase. If the two are equal, then the comparison returns true. Performing the check for lowercase words is done likewise.

Verifying the Idea

It seems our idea is a good one. We've bro**ken down the problem into subproblems** and we know how to solve each of them. Should we continue towards the implementation? Haven't we overlooked something?

Shouldn't we have verified the idea by writing down a few examples on paper? Perhaps we would come across something we have missed. We could start with the example given in the problem statement:

```
Welcome to your first programming exam! Can you think of a
solution to this problem and write it down? GOOD LUCK!
```

The separators would be: **spaces**, **?** and **!**. The words that have come up are the following: **Welcome**, **to**, **your**, **first**, **programming**, **exam**, **Can**, **you**, **think**, **of**, **a**, **solution**, **to**, **this**, **problem**, **and**, **write**, **it**, **down**, **GOOD**, **LUCK**.

Counting the words we acquire the correct result. It seems the idea is adequate, and **it works**. Now we can proceed towards implementing it. We will do this step by step and at each step we will implement one subproblem.

Let's Consider the Data Structures

The problem is simple and doesn't need complex data structures.

We can use the **char** data type for storing each separator. During the process of finding the separator characters we add each of them to a list. We can use either **char[]** or **List<char>**. In this case, we will choose the latter.

As for the words in the text, we can use an array of strings **string[]** or **List<string>**.

Let's Consider the Efficiency

Are there any **performance requirements**? How long can the text be?

Since the text will be entered from the console, it's unlikely to be very long. No one is going to type 1MB of text into the console. We can assume that the solution's **performance is not critical**.

Let's Write Down the Solution

It's very good practice to **write the solution down on a piece of paper** before typing it on the computer. This helps uncover drawbacks in our idea or implementation beforehand. In addition, implementing the solution will be considerably quicker, because of the outlines we can provide and because we would then have a better grasp of both the problem and the solution.

Step 1: Finding the Separators in the Text

We will define a method that **extracts all non-letter characters** from the text and return them as an array of characters. Then we will use that array for splitting the text up into separate words. We will use **List<char>** to keep the separators we find when passing through the text:

```
private static char[] ExtractSeparators(string text)
{
   List<char> separators = new List<char>();
   foreach (char character in text)
   {
      // If the character is not a letter,
      // then by definition it is a separator
      if (!char.IsLetter(character))
      {
         separators.Add(character);
      }
   }
   return separators.ToArray();
}
```

We use a loop to iterate through all of the characters in the text. We check if the current character is a letter by invoking the **IsLetter()** method of the primitive data type **char**. If it's not, we add the character to the separators. Finally, our method returns an array containing the separators.

Testing the ExtractSeparators(…) Method

Before we go any further, it's advisable to **test if extracting the separators is working correctly**. For this purpose, we will write two additional methods. The first of these is **TestExtractSeparators()** which will test the execution of **ExtractSeparators(…)** and the second – **GetTestData()** – will return different texts, which will allow us to test our solution:

```
private static void TestExtractSeparators()
{
   List<string> testData = GetTestData();
   foreach (string testCase in testData)
```

```
  {
    Console.WriteLine(
      "Test Case:{0}{1}", Environment.NewLine, testCase);
    Console.WriteLine("Result:");
    foreach (char separator in ExtractSeparators(testCase))
    {
      Console.Write("{0} ", separator);
    }
    Console.WriteLine();
  }
}

private static List<string> GetTestData()
{
  List<string> testData = new List<string>();
  testData.Add("This is wonderful!!! All separators like " +
    "these ,.(? and these /* are recognized. It works.");
  testData.Add("SingleWord");
  testData.Add(string.Empty);
  testData.Add(">?!>?#@?");
  return testData;
}

static void Main()
{
  TestExtractSeparators();
}
```

We start the program and **check if the separators have been correctly identified**. The first test's result is as follows:

```
Test Case:
This is wonderful!!! All separators like these ,.(? and these /*
are recognized.
 It works.
Result:
    ! ! !           , . ( ?       / *     .     .
Test Case:
SingleWord
Result:

Test Case:

Result:
```

```
Test Case:
>?!>?#@?
Result:
> ? ! > ? # @ ?
```

We might think of the above output as **partially correct**. In fact it does extract correctly the separators between the words but most of them are **duplicated several times**. We need all the separators without duplications, right?

Correcting the ExtractSeparators(…) Method

To correct the method for extracting the separators between the words in the text, we can use a different data structure to keep them. We know that **sets** keep elements without duplications. So we could use **`HashSet<char>`** instead of **`List<char>`** to hold the separator characters we find in the text:

```
private static char[] ExtractSeparators(string text)
{
   HashSet<char> separators = new HashSet<char>();
   foreach (char character in text)
   {
      // If the character is not a letter,
      // then by definition it is a separator
      if (!char.IsLetter(character))
      {
         separators.Add(character);
      }
   }
   return separators.ToArray();
}
```

The code is almost the same, but we use a set instead of list to avoid duplicated separators. We might need to include the **`System.Linq`** namespace in the start of the program to use the **`ToArray()`** extension method for converting a hash set to an array.

Testing Again after the Fix

We test the above method with the same testing code and we find **it now works correctly**. The separators are extracted correctly with no duplicates:

```
Test Case:
This is wonderful!!! All separators like these ,.(? and these
/* are recognized.
 It works.
```

```
Result:
  ! , . ( ? / *
Test Case:
SingleWord
Result:

Test Case:

Result:

Test Case:
>?!>?#@?
Result:
> ? ! # @
```

We test also with some **borderline cases** – text consisting of a single word without separators; text consisting of separators only; an empty string. We've already included such tests in our **GetTestData()** method. It seems that the method works fine and we can proceed to the next step.

Step 2: Splitting Up the Text in Separate Words

We will use **string**'s **Split(…)** method with the specified separators for splitting up the text by the separators and extracting the words from it. This is how our method looks like:

```
private static string[] ExtractWords(string text)
{
  char[] separators = ExtractSeparators(text);
  string[] words = text.Split(separators,
    StringSplitOptions.RemoveEmptyEntries);
  return words;
}
```

Testing the Word Extracting Method

Before we carry on to the next step, we have to see if the method works correctly. To do this, we will reuse the **GetTestData()** for the input test data and we will test the new **ExtractWords(…)** method:

```
private static void TestExtractWords()
{
  List<string> testData = GetTestData();
  foreach (string testCase in testData)
  {
```

```
    Console.WriteLine("\nTest Case: {0}", testCase);
    string[] words = ExtractWords(testCase);
    Console.WriteLine("Result: {0}", string.Join(" ", words));
  }
}

static void Main()
{
  TestExtractWords();
}
```

The result from the above test looks **correct**:

```
Test Case: This is wonderful!!! All separators like these ,.(?
and these /* are
recognized. It works.
Result: This is wonderful All separators like these and these
are recognized It
works

Test Case: SingleWord
Result: SingleWord

Test Case:
Result:

Test Case: >?!>?#@?
Result:
```

We check the results from the other test cases. We verify that they are **correct** and that our algorithm is accurate (till this stop).

Step 3: Determining Whether a Word Is in Uppercase or Lowercase

We already have an idea how to implement the **uppercase / lowercase checks**, and we can write the corresponding methods directly:

```
private static bool IsUpperCase(string word)
{
  bool result = word.Equals(word.ToUpper());
  return result;
}

private static bool IsLowerCase(string word)
```

```
{
   bool result = word.Equals(word.ToLower());
   return result;
}
```

We **test the above methods** by passing words in uppercase, lowercase and mixed case. The results are **correct**.

Step 4: Counting the Words

Now we can proceed to **solving the problem itself** – counting the words. All we have to do is iterate through the list of words and depending on the word's type to increment the corresponding counters. Then we print the result:

```
private static void CountWords(string[] words)
{
   int allUpperCaseWordsCount = 0;
   int allLowerCaseWordsCount = 0;
   foreach (string word in words)
   {
      if (IsUpperCase(word))
      {
         allUpperCaseWordsCount++;
      }
      else if (IsLowerCase(word))
      {
         allLowerCaseWordsCount++;
      }
   }

   Console.WriteLine("Total words count: {0}", words.Length);
   Console.WriteLine("Upper case words count: {0}",
      allUpperCaseWordsCount);
   Console.WriteLine("Lower case words count: {0}",
      allLowerCaseWordsCount);
}
```

Testing the Word Counting Method

Let's **check if we count the words correctly**. We will write another test method using the data from the **`GetTestData()`** method and the previously written and tested **`ExtractWords(…)`** method:

```
private static void TestCountWords()
{
```

```
    List<string> testData = GetTestData();
    foreach (string testCase in testData)
    {
        Console.WriteLine("Test Case: {0}", testCase);
        Console.WriteLine("Result: ");
        CountWords(ExtractWords(testCase));
        Console.WriteLine();
    }
}

static void Main()
{
    TestCountWords();
}
```

Executing the application, we obtain the **correct result**:

```
Test Case: This is wonderful!!! All separators like these ,.(? and these /* are recognized. It works.
Result:
Total words count: 13
Upper case words count: 0
Lower case words count: 10

Test Case: SingleWord
Result:
Total words count: 1
Upper case words count: 0
Lower case words count: 0

Test Case:
Result:
Total words count: 0
Upper case words count: 0
Lower case words count: 0

Test Case: >?!>?#@?
Result:
Total words count: 0
Upper case words count: 0
Lower case words count: 0
```

The above **results are correct** (the **typical case** and a few **borderline cases**). We perform few other **borderline tests**, e.g. when the list contains words in uppercase or lowercase only, or when the list is empty. All of them work correctly.

Note that it is a good idea to use **unit testing** instead of these semi-automated tests. Recall how we **write unit tests in Visual Studio** (in the chapter "High-Quality Code") and try to convert our test methods to unit tests for the Visual Studio Team Test (VSTT) framework.

Step 5: Console Input

All that's left to implement is **the final step** – allowing the user to input text:

```
private static string ReadText()
{
   Console.WriteLine("Enter text:");
   return Console.ReadLine();
}
```

Note that as a rule unless the input comes from a text file or is very short (e.g. just one number or few characters) **it should be read as a final step**. Otherwise we will need to enter the input data each time when we start the program and this will waste a lot of time and can lead to errors.

Step 6: Putting All Together

Now after **all subproblems have been solved**, we can proceed to the complete solution to the problem. We need to add a **`Main(…)`** method, which will **combine together the different parts of the solution**:

```
static void Main()
{
   string text = ReadText();
   string[] words = ExtractWords(text);
   CountWords(words);
}
```

Testing the Solution

While implementing the solution, we **wrote test methods for every method**, integrating them with each other gradually. For the moment, we are certain they interact correctly; there's nothing we have overlooked and there is no method that does unnecessary work or that returns incorrect results.

If we would like to **test the solution with more data**, we would only need to add it to the **`GetTestData(…)`** method. If we want, we may even rewrite the **`GetTestData(…)`** method so that it reads the test data from an external source, e.g. from a text file.

Here's how the final solution looks like at the end:

WordsCounter.cs

```
using System;
using System.Collections.Generic;
using System.Linq;

public class WordsCounter
{
   static void Main()
   {
      string text = ReadText();
      string[] words = ExtractWords(text);
      CountWords(words);
   }

   private static string ReadText()
   {
      Console.WriteLine("Enter text:");
      return Console.ReadLine();
   }

   private static char[] ExtractSeparators(string text)
   {
      HashSet<char> separators = new HashSet<char>();
      foreach (char character in text)
      {
         // If the character is not a letter,
         // then by definition it is a separator
         if (!char.IsLetter(character))
         {
            separators.Add(character);
         }
      }
      return separators.ToArray();
   }

   private static string[] ExtractWords(string text)
   {
      char[] separators = ExtractSeparators(text);
      string[] words = text.Split(separators,
         StringSplitOptions.RemoveEmptyEntries);
      return words;
   }

   private static bool IsUpperCase(string word)
   {
```

```
      bool result = word.Equals(word.ToUpper());
      return result;
   }

   private static bool IsLowerCase(string word)
   {
      bool result = word.Equals(word.ToLower());
      return result;
   }

   private static void CountWords(string[] words)
   {
      int allUpperCaseWordsCount = 0;
      int allLowerCaseWordsCount = 0;

      foreach (string word in words)
      {
         if (IsUpperCase(word))
         {
            allUpperCaseWordsCount++;
         }
         else if (IsLowerCase(word))
         {
            allLowerCaseWordsCount++;
         }
      }

      Console.WriteLine("Total words count: {0}", words.Length);
      Console.WriteLine("Upper case words count: {0}",
         allUpperCaseWordsCount);
      Console.WriteLine("Lower case words count: {0}",
         allLowerCaseWordsCount);
   }
}
```

We removed the testing methods from our code to simplify it. The best practice is instead of removing the tests to **create a separate testing project** and put all the tests in a **testing class**. This is best achieved though the Visual Studio's unit testing framework, as it was shown in the chapter "High-Quality Code".

A Word on Performance

Since there are **no explicit performance requirements**, we will only make a suggestion for dealing with the situation when the algorithm turns out to be slow. Splitting the text with separators assumes that the **entire text will be**

loaded into memory. The list of words, after partitioning the text, will also be written to memory. Therefore, if the input text is large, the program will also **consume a large amount of memory**. For example, if the input text is 200MB long, then the program will consume at least 800MB of memory, because each word is stored as 2 bytes for every character (.NET uses UTF-16 character encoding for the strings in memory).

If we want to **avoid high memory consumption** then the words must not be stored in memory all at once. We can come up with **another algorithm**: **scanning the text char by char** and storing the letters into a buffer (such as **`StringBuilder`**). If at a certain moment a separator is encountered, then the buffer contains the most recent word. We can analyze its casing and then empty the buffer. We can repeat this until the end of the file is reached. This appears to be **more efficient**, doesn't it?

A **more efficient lower / upper case checker** would be to iterate through all letters using a loop and to examine them char by char. That way we can skip a lower / upper case conversion, which allocates extra memory for every word. After the word has been processed, the memory will be freed, which would eventually lead to extra CPU utilization (for the .NET garbage collector).

Obviously, the latter solution is **more efficient**. The question is if we should scrap the original solution and write a completely different one. It all **depends on the performance requirements**. The problem description doesn't hint at an input text measuring in the hundreds of megabytes. Therefore the current solution, although not optimal, is still correct and will suffice. We suggest the **reader to implement the proposed fast solution** and to **compare how faster it is**, e.g. by processing an input of 100 MB.

Problem 2: A Matrix of Prime Numbers

Write a program that reads a positive integer N from the standard input and **prints the first N^2 prime numbers as a square matrix of size N x N**. The matrix must be filled with numbers starting from the first row and ending at the last one. Each row must be filled with prime numbers from left to right.

Note: A **prime number** is a number that has no divisors other than 1 and itself. The number 1 is not a prime number.

Sample input:

2	3	4

Sample output:

2 3 5 7	2 3 5 7 11 13 17 19 23	2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53

Coming Up with an Appropriate Idea for a Solution

We can solve the problem by printing the rows and columns of the resulting matrix using **two nested loops**. For each of its elements we will extract and print the corresponding **prime number**.

Breaking Down the Problem into Subproblems

We must solve at least **two subproblems** – **finding each successive prime number** and **printing the prime numbers into a matrix**. We can print the matrix right away, but the process of finding each successive prime number will require additional thinking. Perhaps the most intuitive way to accomplish this is to start testing the primality of each number starting from the last prime number that we found. When a new prime is encountered, it is returned as a result. Thus, a new subproblem has come up – checking whether or not a number is a prime.

Verifying the Idea

Our idea for a solution leads directly to the required result. We **write down a couple of examples on a piece of paper** and make sure that it works.

Consider the Data Structures

The problem makes use of one **data structure** only – a **matrix**. It's only natural to use a two-dimensional array (matrix).

Consider the Efficiency

Displaying at the console large matrices (for example of size 1000 x 1000) cannot be properly handled. This means our solution should work for reasonably large matrices, e.g. on the order of N ≤ 200. We don't need to consider cases where the matrix is too large. When N = 200, our algorithm will find **the first 40,000 prime numbers** and should not run slowly.

Now we are ready for the **implementation of the algorithm** we invented.

Step 1: Check to Find If a Number Is a Prime

To test a number for primality, we can define a method called **`IsPrime(…)`**. The test will verify that dividing the number by any of its predecessors always yields a division remainder. To be more precise, it is sufficient to **check the integers between 2 and the square root of the number**. This holds true, because if the number **`p`** has a divisor **`x`**, then **`p = x.y`**, and at least one or both of the numbers **`x`** and **`y`** will be less than or equal to the square root of **`p`**. What follows is an implementation of the method:

```
private static bool IsPrime(int number)
{
```

```
    int maxDivider = (int)Math.Sqrt(number);
    for (int divider = 2; divider <= maxDivider; divider++)
    {
        if (number % divider == 0)
        {
            return false;
        }
    }
    return true;
}
```

The **algorithm complexity** of the above example is **O(sqrt(number))**, because the amount of checks that will be made is not greater than the square root of the number. This complexity will suffice for the problem at hand, but is it possible to optimize this method even further?

Come to think about it, every second number is even and all even numbers are divisible by 2. In that case, if the number we are testing is odd, the above method will needlessly check all odd numbers from 2 to the square root of the number. How can we omit these unnecessary checks? We could find out if the number is even at the very beginning of the method. If it's not, it will be processed in a modified version of the main loop that skips even numbers. Using this new approach, we have achieved the same computational complexity of **O(sqrt(number))**, but with a better constant 1/2.

This example illustrates how to optimize a bit the existing method:

```
private static bool IsPrime(int number)
{
    if (number == 2)
    {
        return true;
    }
    if (number % 2 == 0)
    {
        return false;
    }

    int maxDivider = (int)Math.Sqrt(number);
    for (int divider = 3; divider <= maxDivider; divider += 2)
    {
        if (number % divider == 0)
        {
            return false;
        }
    }
```

```
    return true;
}
```

As we can see, there was a minimal amount of changes compared to the non-optimized version.

Testing the Prime Checking Method

We can make sure both methods work correctly by consecutively passing to them different numbers, some of which will be primes, and verifying the results.

Always test a method to make sure it works before optimizing it.

The reason to test your methods is that after optimization, the code usually gets longer, more difficult to read and therefore more difficult to debug, if it is incorrect.

Be careful when optimizing a piece of code. Do not go to extremes by making unnecessary optimizations that will make your program marginally faster at the expense of readability and maintenance.

To check the prime checking method we could write a piece of code like this:

```
static void Main()
{
    Console.WriteLine(IsPrime(2));
    Console.WriteLine(IsPrime(3));
    Console.WriteLine(IsPrime(4));
    Console.WriteLine(IsPrime(5));
    Console.WriteLine(IsPrime(121));
}
```

It runs as expected and the produced results are correct:

```
True
True
False
True
False
```

Step 2: Finding the Next Prime Number

In order to find the **next prime number**, we can define a method that takes an integer as a parameter and returns the first prime number equal or larger

than it. To check if the number is prime, we will use the method from the previous step. Below is an implementation of the method:

```
private static int FindNextPrime(int startNumber)
{
   int number = startNumber;
   while (!IsPrime(number))
   {
      number++;
   }
   return number;
}
```

Testing the Next Prime Number Finder

Once again we have to test the method by passing a few numbers and verifying that the result is correct:

```
static void Main()
{
   Console.WriteLine(FindNextPrime(2));
   Console.WriteLine(FindNextPrime(3));
   Console.WriteLine(FindNextPrime(4));
   Console.WriteLine(FindNextPrime(5));
   Console.WriteLine(FindNextPrime(121));
}
```

The result is correct, as expected:

```
2
3
5
5
127
```

Step 3: Printing the Matrix

Now that we have defined the previous methods, we are all set to **print the entire matrix of prime numbers:**

```
private static void PrintMatrixOfPrimes(int dimension)
{
   int lastPrime = 1;
   for (int row = 0; row < dimension; row++)
   {
      for (int col = 0; col < dimension; col++)
```

```
    {
      int nextPrime = FindNextPrime(lastPrime + 1);
      Console.Write("{0,4}", nextPrime);
      lastPrime = nextPrime;
    }
    Console.WriteLine();
  }
}
```

We will test this method as part of testing the entire program.

Step 4: Console Input

All that's left is to add functionality allowing us to **read N from the console**.

```
static void Main()
{
  int n = ReadInput();
  PrintMatrixOfPrimes(n);
}

private static int ReadInput()
{
  Console.Write("N = ");
  string input = Console.ReadLine();
  int n = int.Parse(input);
  return n;
}
```

Testing the Entire Solution

After we have completed all other steps, we can proceed with **testing the entire solution**. To do this, we could look up the first 25 prime numbers (at a sheet of paper) and test the program's output for values of N between 1 and 5. We should include special **border cases** like N=0 and N=1. We know that at border cases the likelihood of making a mistake is significantly higher.

In our case, we can confine ourselves with the examples from the problem description, provided that the methods have been thoroughly tested at each step. This is the output of the program for N = 1, 2, 3 and 4 respectively:

```
2          2 3              2 3 5              2 3 5 7
           5 7              7 11 13            11 13 17 19
                            17 19 23           23 29 31 37
                                               41 43 47 53
```

The result is correct and after few more tests we get convinced that we have **solved correctly the problem** "Matrix of Prime Numbers".

We can make sure the solution works relatively fast even for larger values of N. For example, there is no perceived lag for N = 200.

A Word on Performance

We should point out, that our solution **does not find prime numbers in the most efficient way**. Despite of this drawback, due to the solution's clarity and the reasonably small size of the matrix, we can utilize this algorithm without performance issues.

Improved Performance: Sieve of Eratosthenes

If we have to **improve the performance**, we can find the first N^2 prime numbers using the Sieve of Eratosthenes. That way we will not need to check if every number is prime until we find N^2 prime numbers. You might ask yourself **how large Eratosthenes's Sieve** will be needed if we want to find the first N^2 prime numbers. You might use the following approximation (without any mathematical proof):

```
long sieveSize =
   (long)Math.Truncate(2.4 * n * n * Math.Log(n, Math.E)) + 2;
```

If the Eratosthenes's Sieve is at least **sieveSize** elements large, it will be enough to produce the first N^2 prime numbers and not too much above them. You could check this manually or **you might invite a better formula** using complex mathematical calculations (see http://en.wikipedia.org/wiki/Prime-counting_function). For example if N=10, the estimated **sieveSize** will be 554 and it will find the first 101 prime numbers (we need $10^2 = 100$ prime numbers to fill the matrix, so these 101 prime numbers are enough). If N=1,000, the **sieveSize** will be 16,578,614 and it will find the first 1,065,855 prime numbers. For N=5,000 the **sieveSize** will be 511,031,593 and it will find the first 26,905,486 prime numbers. For significantly bigger sizes Eratosthenes's Sieve will not fit in the memory. You might **try to implement this algorithm** and **check how faster it is**. When comparing the speed, you may redirect the output to a file to save some time which is spent in printing the matrix.

Problem 3: Evaluate an Arithmetic Expression

Write a program that **evaluates a simple arithmetic expression** consisting of unsigned integers and the arithmetic operations "**+**" and "**-**". There will be no blank spaces between the integers.

The expression will have the following format:

```
<number><operation>…<number>
```

Sample input:

```
1+2-7+2-1+28+2+3-37+22
```

Sample output:

```
15
```

Coming Up with an Appropriate Idea for a Solution

To solve this problem, we can take advantage of the strict expression format, which guarantees we have a sequence of a number, operation, another number and so on.

That way, we can first **extract all numbers** from the expression, and then we can **extract all operators** and finally evaluate the result by **combining the numbers with the operators**.

Verifying the Idea

Sure enough, if we test this approach with a few expressions **using pen and paper**, we acquire a correct result. Initially, the result is equal to the first number and at each step we either add or subtract the next number depending on the current operator.

Data Structures and Efficiency

The problem is too simple for us to use complex **data structures**. The numbers and characters can be stored in **arrays**. Performance issues are out of the question, because the characters and numbers are processed exactly once, i.e. the complexity of the algorithm is **linear**. Even with millions of integers and operators, the algorithm is expected to work fast.

Breaking Down the Problem into Subproblems

Now that we have made sure the idea works, we can move on to breaking down the problem into subproblems. The first subproblem we will have to solve is **extracting the numbers** from the expression. The second – **extracting the operators**. Finally, we will **evaluate the entire expression** using the extracted numbers and operators.

Step 1: Extracting the Numbers

In order to extract the numbers, we need to **split the expression using the operators (+ and -) as separators**. This is easily done using the **`Split(…)`** method of the **`string`** class. Afterwards, we have to convert the resulting array of strings to an array of integers:

```
private static int[] ExtractNumbers(string expression)
{
   string[] splitResult = expression.Split('+', '-');
   int[] resultNumbers = new int[splitResult.Length];
   for (int i = 0; i < splitResult.Length; i++)
   {
      resultNumbers[i] = int.Parse(splitResult[i]);
   }
   return resultNumbers;
}
```

We use the **Parse(…)** method of the **Int32** class to convert strings to integers. It takes a string as a parameter and returns the integer value that the string represents.

Why do we **use an array to store the numbers**? Can't we use a linked list or a dynamic array? Of course we can, but in our case we only need to store the integers and iterate through them when evaluating the result. Therefore, an array is sufficient and is the simplest collection that will work.

Testing the Extraction of Numbers

Before we move on to the next step we should **check if the numbers are extracted correctly**. We may use the following example:

```
static void Main()
{
   int[] numbers = ExtractNumbers("1+2-7+2-1+28");
   foreach (int x in numbers)
   {
      Console.Write("{0} ", x);
   }
}
```

The result is **exactly as it should be**:

```
1 2 7 2 1 28
```

We examine the border case when the expression consists only of one number and no operators, and we make sure it is handled properly.

Step 2: Extracting the Operators

We can **extract the operators** by iterating through each consecutive character from the string and check if it is one of the specified operators:

```
private static char[] ExtractOperators(string expression)
```

```
{
   string operatorCharacters = "+-";
   List<char> operators = new List<char>();
   foreach (char c in expression)
   {
      if(operatorCharacters.Contains(c))
      {
         operators.Add(c);
      }
   }
   return operators.ToArray();
}
```

Testing the Extraction of Operators

Here's how we **test** whether or not the method works correctly:

```
static void Main()
{
   char[] operators = ExtractOperators("1+2-7+2-1+28+3+1");
   foreach (char oper in operators)
   {
      Console.Write("{0} ", oper);
   }
}
```

The output after the program's execution is **correct**:

```
+ - + - + + +
```

We create a **test for the border case** when the expression consists of only one number and no operators. Just as expected, we get an empty array and the output of the above testing program is and empty string.

Step 3: Evaluating the Expression

When **evaluating the expression**, we can make use of the fact that the **numbers' count is always greater than the operators' count by one**. Using a single loop we can evaluate the expression, provided we have the lists of numbers and operators:

```
private static int CalculateExpression(int[] numbers,
   char[] operators)
{
   int result = numbers[0];
   for (int i = 1; i < numbers.Length; i++)
```

```
    {
        char operation = operators[i - 1];
        int nextNumber = numbers[i];
        if (operation == '+')
        {
            result += nextNumber;
        }
        else if (operation == '-')
        {
            result -= nextNumber;
        }
    }
    return result;
}
```

Test the Evaluation of Expression

We test the method's execution:

```
static void Main()
{
    // Expression: 1 + 2 - 3 + 4
    int[] numbers = new int[] { 1, 2, 3, 4 };
    char[] operators = new char[] { '+', '-', '+' };
    int result = CalculateExpression(numbers, operators);
    // Expected result is 4
    Console.WriteLine(result);
}
```

The result seems to be **correct**:

```
4
```

We perform few other tests (e.g. a 1, 1+2, 1-1) and it still works correctly.

Step 4: Console Input

We have to provide the user with the means to **enter an expression**:

```
private static string ReadExpression()
{
    Console.Write("Enter expression: ");
    string expression = Console.ReadLine();
    return expression;
}
```

This could be **left untested**, because it is too simple and it will be tested later when the entire program is finished and gets tested.

Step 5: Putting All Together

All that's left is to **integrate all methods** so that they can function together.

```
static void Main()
{
   string expression = ReadExpression();

   int[] numbers = ExtractNumbers(expression);
   char[] operators = ExtractOperators(expression);

   int result = CalculateExpression(numbers, operators);
   Console.WriteLine("{0} = {1}", expression, result);
}
```

Testing the Solution

We can use the **example from the problem description** when testing the solution. We get a correct result:

```
Enter expression: 1+2-7+2-1+28+2+3-37+22
1+2-7+2-1+28+2+3-37+22 = 15
```

In order to make sure the solution runs correctly, we must create a **few more tests** with different parameters, which will include the case when the expression consists of **one number only**.

We can even test with an **empty string**. It's not clear if such input is valid, but we can take it into account just to be on the safe side. It's also ambiguous what would happen, if "**2 + 3**" (with spaces) is entered, instead of "**2+3**". It's a good idea to cope with these situations.

One test case we neglected occurs when a number doesn't fit in an **int** variable. What do you think would happen, if the user enters the expression "**11111111111111111111111111111+22222222222222222222222222222**"?

Small Corrections and Repeated Testing

The program will raise an exception (of type **System.FormatException** or **System.OverflowException** in all likelihood) when the expression is invalid. It's sufficient to catch the exceptions, and when they are thrown, to report that an invalid expression has been entered. Below is the solution's full implementation after this correction:

SimpleExpressionEvaluator.cs

```
using System;
using System.Collections.Generic;
using System.Linq;

public class SimpleExpressionEvaluator
{
   private static int[] ExtractNumbers(string expression)
   {
      string[] splitResult = expression.Split('+', '-');
      int[] resultNumbers = new int[splitResult.Length];
      for (int i = 0; i < splitResult.Length; i++)
      {
         resultNumbers[i] = int.Parse(splitResult[i]);
      }
      return resultNumbers;
   }

   private static char[] ExtractOperators(string expression)
   {
      string operationsCharacters = "+-";
      List<char> operators = new List<char>();
      foreach (char c in expression)
      {
         if (operationsCharacters.Contains(c))
         {
            operators.Add(c);
         }
      }
      return operators.ToArray();
   }

   private static int CalculateExpression(
      int[] numbers, char[] operators)
   {
      int result = numbers[0];
      for (int i = 1; i < numbers.Length; i++)
      {
         char operation = operators[i - 1];
         int nextNumber = numbers[i];
         if (operation == '+')
         {
            result += nextNumber;
```

```
        }
        else if (operation == '-')
        {
          result -= nextNumber;
        }
      }
      return result;
    }

    private static string ReadExpression()
    {
      Console.WriteLine("Enter expression:");
      string expression = Console.ReadLine();
      return expression;
    }

    static void Main()
    {
      try
      {
        string expression = ReadExpression();

        int[] numbers = ExtractNumbers(expression);
        char[] operators = ExtractOperators(expression);

        int result = CalculateExpression(numbers, operators);
        Console.WriteLine("{0} = {1}", expression, result);
      }
      catch (Exception ex)
      {
        Console.WriteLine("Invalid expression!");
      }
    }
}
```

To ensure everything works correctly after the fix we need to **test the above code again** with: single number, two numbers, typical expression (like the sample input from the problem description), expression with spaces (e.g. "1 + 2 -3"), expression with large numbers, invalid expression (e.g. -1).

Performance Test

Finally we could test with very long expression (**performance test**), e.g. sum of 1,000,000 ones. We could generate a test of 1,000,000 numbers with the following sample code:

```
static void Main()
{
   StringBuilder expression = new StringBuilder();
   expression.Append("0");
   for (int i = 0; i < 1000000; i++)
   {
      expression.Append("+");
      expression.Append("1");
   }
   string expr = expression.ToString();
   int[] numbers = ExtractNumbers(expr);
   char[] operators = ExtractOperators(expr);
   int result = CalculateExpression(numbers, operators);
   Console.WriteLine(result);
}
```

The running time seems acceptable and the result is correct.

But what will happen if we **sum 1,000,000 times the value of 5,000,000**? We will get an **integer overflow**. We might fix this by using **long** for the sum instead of **int**:

```
private static long CalculateExpression(
   int[] numbers, char[] operators)
{
   long result = numbers[0];
   for (int i = 1; i < numbers.Length; i++)
   {
      char operation = operators[i - 1];
      int nextNumber = numbers[i];
      if (operation == '+')
      {
         result += nextNumber;
      }
      else if (operation == '-')
      {
         result -= nextNumber;
      }
   }
   return result;
}
```

After this small fix we sum 1,000,000 times the value of 5,000,000 and we **get the correct result**: 5,000,000,000,000. The problem is solved.

Exercises

1. Solve the problem "**Counting the Number of Words in a Text**" by using a buffer (**`StringBuilder`**) to read the text. Was there a change in the complexity of your algorithm?
2. Implement a more efficient solution to the problem "A Matrix of Prime Numbers" by using the "**Sieve of Eratosthenes**": http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes.
3. Add support for **integer multiplication and division** to the solution of "Arithmetic Expression". Bear in mind that these operations have priority over addition and multiplication!
4. Add support for **floating point numbers** in the solution of the "Arithmetic Expression" problem.
5. * Add **parentheses support** to the solution of "Arithmetic Expression". Try to process correctly the **unary minus** (e.g. in the expression -2 + 3).
6. * Write a program that validates an arithmetic expression. For example, "**`2*(2.25+5.25)-17/3`**" is a valid expression, but "**`*232*-25+(33+a`**" is not.

Solutions and Guidelines

1. Your program can read from the input file **character by character**. If the current character is a **letter**, append it to the buffer, and if it is a **separator**, analyze the buffer (since it holds the current word) and clear it. When the end of the input file is reached and the file does not end with a separator, analyze the last word in the buffer. **Test the solution**!
2. First, estimate the count of prime **numbers you will need**. Then consider what the upper limit of the iterations for the **Sieve of Eratosthenes** should be, so that there are enough numbers to fill the matrix. You can come up with a **formula** by experimenting or you may use the formula from the section "Improved Performance: Sieve of Eratosthenes".
3. Taking into account that in math multiplication and division has a **priority** over addition and subtraction you can **calculate all multiplications and divisions first**, **replace** them with their result and **then all the additions and subtractions**. For example, let's the expression is "2*5-8/2+11". You may first calculate all multiplications and divisions and replace them with the results of their execution: "2*5-8/2+11" → "10-4+11". Then you may use the algorithm from the "Evaluate an Arithmetic Expression" section. Did you consider division by zero? **Test your code**. Think of special cases.
4. Floating point arithmetic can be implemented by allowing the use of the character "**`.`**" and replacing **`int`** with **`double`** or **`decimal`**. **Test your code**!
5. You can do the following: locate the first **closing parenthesis** and match it with its **corresponding opening parenthesis**. What remains between them is an arithmetic expression that can be evaluated with the same algorithm recursively. You can substitute the expression with its value and

repeat the process until there aren't any more parentheses. Eventually, you will end up with an expression without parentheses.

For example, if the expression "**2*((3+5)*(4-7*2))**" is entered, you will substitute "**(3+5)**" with **8**, and "**(4-7*2)**" with **-10**. Finally, you will replace "**(8*-10)**" with **-80** and calculate **2*-80**, thus getting the result **-160**. You will have to consider the arithmetic operations with negative numbers, i.e. adding **negative numbers support** when parsing the numbers.

There is one more algorithm. It utilizes a stack and converts the expression to **reverse Polish notation (RPN)**. Look up the terms "**postfix notation**" and "**shunting yard algorithm**" on the Internet.

To handle correctly the **unary minus**, you may consider two situations. The first is a leading unary minus (e.g. -3 + 5). The second is a minus after another operator or after a bracket, e.g. "3 * -2 + 4". The minus can be applied to a number of to an expression in brackets. In both cases you may **insert "0-" and put the number or expression on the right in brackets**. Examples:

- "-3 + 5" → "(0-3) + 5"
- "3 * -2 + 5" → "3 * (0-2) + 5"
- "-(3+2)" → "(0-(3+2))"
- "-(-1) * 3 - -1" → "(0-((0-1))) * 3 – (0-1)"

6. If you evaluate the expression using **reverse polish notation**, you can **expand your algorithm to check the expression for validity**. Follow these rules: when you expect a digit, but the next token is not a digit, then the expression is invalid; when an arithmetic operation is expected, but the next token is not a valid operator, then the expression is invalid; when the parentheses do not match, the stack will either underflow or remain non-empty at the end. Don't forget the special cases "**-1**", "**-(2+4)**", etc. **Test thoroughly your code**! There are many special cases to consider.