Implement the following problems in the attached Visual Studio solution (if you do not have Visual Studio 2010, you can download Visual C# 2010 Express here: <http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-csharp-express>). The method signatures are already defined in DeveloperTest\Implementation.cs; you only need to implement them. There are a number of simple tests included to assist you with beginning the implementation – they can be run by debugging the DeveloperTestRunner project (you should just be able to press *F5*). Be aware, **the tests included do not cover all cases that we will test**. Feel free to add tests of your own to cover any additional cases that you identify.

# Find the Nth Prime Number

Write a method that will print out the nth prime number (a prime number is a natural number that is greater than 1 and can be divided evenly only by 1 and itself). For example if 5 was passed to the method it would skip the first 4 prime numbers (2,3,5,7) and print out 11.

Signature of method you will implement:

void FindNthPrimeNumber(int nthPrimeNumber)

Some sample output:

FindNthPrimeNumber(5); //output should be 11

FindNthPrimeNumber(10); //output should be 29

FindNthPrimeNumber(100); //output should be 541

# Overlapping Appointments

Implement a function that will return a Boolean indicating whether or not two appointments are overlapping. Assume that you can use normal comparison operators on the appointment start and end DateTimes (as you can in .NET). The parameters appointment1Start and appointment1End correspond to the start and end times, respectively, of the first appointment and appointment2Start and appointment2End correspond to the start and end of the second appointment. You may assume that each appointment time has already been validated to ensure that the start of one appointment does not occur after the end of that same appointment. You may not assume that one of the appointments starts or ends before the other starts or ends (the first appointment doesn’t necessarily occur before the second appointment). If one appointment ends at the same time that the other starts, consider them to not be overlapping (since typically you wouldn’t consider an appointment that ends at 5 p.m. to be conflicting with an appointment that starts at 5 p.m.).

The function signature is:

bool OccursDuring(DateTime appointment1Start, DateTime appointment1End,

DateTime appointment2Start, DateTime appointment2End)

Some example calling code:

// July 12, 2011, 2:30 PM

var time1 = new DateTime(2011, 7, 12, 14, 30, 0);

// July 12, 2011, 3:00 PM

var time2 = new DateTime(2011, 7, 12, 15, 0, 0);

// July 12, 2011, 3:30 PM

var time3 = new DateTime(2011, 7, 12, 15, 30, 0);

// July 12, 2011, 4:00 PM

var time4 = new DateTime(2011, 7, 12, 16, 0, 0);

// false

Console.WriteLine(OccursDuring(

time1, time2, time2, time4));

// true

Console.WriteLine(OccursDuring(

time1, time3, time2, time4));

// true

Console.WriteLine(OccursDuring(

time2, time4, time1, time3));

// true

Console.WriteLine(OccursDuring(

time2, time3, time1, time4));

// false

Console.WriteLine(OccursDuring(

time2, time4, time1, time2));

// true

Console.WriteLine(OccursDuring(

time1, time4, time2, time3));

# Find a Sequence

Implement a function that takes two arrays of strings as input. The function should search the first array, which we’ll call the haystack, for the sequence contained in the second array, which we’ll call the needle. If the sequence in the needle is found within the haystack, then the function should return the index of the position of the sequence in the haystack. If the sequence in the needle is not found, then the function should return -1. For example, if the haystack were {“one”, “two”, “three”, “four”} and the needle were {“two”, “three”}, then the function should return 1, since the sequence in the needle {“two”, “three”} is positioned at the second position of the haystack (assume that the arrays are 0-indexed).

The function signature is:

int FindSequence(string[] haystack, string[] needle)

Some example calling code:

// 1

Console.WriteLine(

FindSequence(

new string[] {"apple", "banana", "pear"},

new string[] {"banana", "pear"}));

// -1

Console.WriteLine(

FindSequence(

new string[] {"apple", "banana", "pear"},

new string[] {"four", "five"}));

// -1

Console.WriteLine(

FindSequence(

new string[] {"apple", "banana", "pear"},

new string[] {"pear", "four"}));

// -1

Console.WriteLine(

FindSequence(

new string[] {"apple", "banana", "pear"},

new string[] {"apple", "pear"}));

// 0

Console.WriteLine(

FindSequence(

new string[] {"apple", "banana", "pear"},

new string[] {"apple"}));

// 3

Console.WriteLine(

FindSequence(

new string[] {"eleven", "twelve", "thirteen", "thirteen", "fourteen", "fifteen", "sixteen" },

new string[] {"thirteen", "fourteen", "fifteen"}));

// -1

Console.WriteLine(

FindSequence(

new string[] {},

new string[] {}));

// -1

Console.WriteLine(

FindSequence(

new string[] { "apple", "banana", "pear" },

new string[] {}));

# Build a Tree

Write a method that will build a tree from an input collection and return the root node of the tree. You can assume that the collection being passed to your method would always be sorted so that no element would reference a parent id that had not previously existed in the collection (this means the input representing the root will be passed in first). If the input objects parent Id is 0 or null, that means it is the root.

The signature of the method you implement should be as follows:

public TreeNode BuildTree(IEnumerable<Input> inputs, TreeNode parentNode = null)

The types referenced by *BuildTree* are explained below (do not worry about their implementation details):

class Input

{

    public int Id

    // Represents a link to a parent. If Parent Id is 0 this is the root

    public int ParentId

}

class TreeNode

{

    // The Parent node

    public TreeNode Parent

    // A collection of the child nodes. Items can be added to the

    // underlying collection via AddChild

    public ReadOnlyCollection<TreeNode> Children

    // Id of the node

    public int Id

    // Constructor that takes in an id

    public TreeNode(int id)

    // Method used to add child nodes

    public void AddChild(TreeNode child)

    // Depth first traversal of tree

    public void Traverse(Action<TreeNode> visit)

}

The following can be used as a test case:

Input[] input = new Input[]

{

        new Input (1),

        new Input (3,1),

        new Input (19,3),

        new Input (22,1),

        new Input (4,1),

        new Input (5,22),

};

TreeNode rootNode = BuildTree(input);

rootNode.Traverse(x => Console.WriteLine(child.Id));

Which would output:

1  
3  
19  
22  
5  
4

The logical view of the tree would be:

         1

      /  |  \

     3   22  4

     |   |

     19  5