Mock4

1.

The function `Evens` accepts an array of integers as its formal parameter type and returns the number of even values in that array. For the following questions you may wish to make use of `LINQ`.

i. Write a recursive version of `Evens`.
ii. Write a version of `Evens` that uses a higher-order method.
iii. Write a version that uses the equivalent of a filter higher-order method and a regular method.
iv. Write a version that uses the equivalent of a map higher-order method and a regular method.

4. **[12 marks]** Write a method which uses reflection to find out (and output) which methods are defined for a class. You should include the formal parameters and return types in your output, together with any accessibility modifiers.

8. **[12 marks]** Using the following definition of a `BigInt`, write a `BigInt` method, `Factor`, that returns a `BigInt` that is its smallest factor greater than 1. For example, if `Factor` is applied to a `BigInt` representing `12`, it returns a `BigInt` representing `2`.

- The method `Factor` can assume that its `BigInt` is not in the error state.
- Your code does not have to be efficient.

Please note: you do not (and should not) complete the definitions in `BigInt`.

```
public class BigInt
{
  private int[] digits; // stores BigInt as an array of Int
  private bool error;   // if overflow or underflow occurred, then true else false

  public BigInt(int x=0) // This class deals only with integers >= 0
  {
        digits = new int[x];
  }

  public BigInt(BigInt x) : this(x.digits.Length)
  {
                // ...
  }

  public BigInt(int[] arr) : this(arr.Length)
  {
    // ...
  }

  // Return a String representation of a BigInteger
        public string OutputBigInteger()
        {
                // ...
  }

  // Add x to this, put return new value
        public BigInt AddBigIntegers(BigInt x)
        {
                // ...
  }

  // Subtract x from this, return new value
        public BigInt SubtractBigIntegers(BigInt x)
        {
                // ...
  }

        // Multiply this by x, return new value
        public BigInt MultiplyBigIntegers(BigInt x)
        {
                // ...
  }

  // Divide this by x, return new value
        public BigInt DivideBigIntegers(BigInt x)
        {
                // ...
  }

  // return the remainder of this when divided by val (this is unchanged)
        public BigInt ModulusBigIntegers(BigInt x)
        {
                // ...
  }
  // true when this == x
        public bool IsEqualTo(BigInt x)
        {
                // ...
  }

  // true when this < x
        public bool IsLessThan(BigInt x)
        {
                // ...
  }
}
```

9. **[16 marks]** In this question you will implement a recursive function which evaluates Fibonacci numbers. You will then improve its performance by caching its results using *memoisation*.

   i. Write a recursive method `Fib` which, given an integer n computes and returns the $n-th$ Fibonacci number.

   ```
   Func<int,int> Fib(int n)
   ```

   The first two Fibonacci numbers are one. Every other Fibonacci number is the sum of the previous two Fibonacci numbers. Use recursion in your implementation.

   ii. Our Fibonacci method works, but it has a problem — it is very slow for larger numbers! If we take a look at the recursive calls, we can notice that some Fibonacci numbers are computed more than once. In fact, the number of recursive calls is exponential in the argument n, and this is due to the fact that we call the method `Fib` multiple times with the same arguments. If we could somehow remember that we are already evaluated a certain tuple, we could use the solution we have computed earlier and avoid computing it twice.

   Implement the method `Memo` which takes a function `f` and returns its memoised version. The memoised function retains internally a mutable `Dictionary` that maps function arguments to return values. Each time a memoised function is applied to some argument it uses the `Dictionary` to check if it was previously applied to that argument. If it was, it returns the return value associated with that argument. Otherwise, it uses the original function `f` to compute the return value and returns it. For example, the following snippet:

   ```
   var mf = Memo(fib);
   mf(10);
   mf(10);
   ```

   will not recompute all the Fibonacci numbers up to 10 the second time `mf` is called.