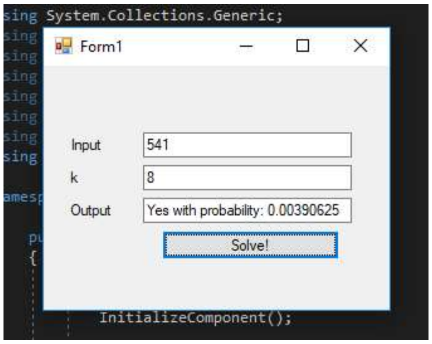
Lab 1: Fermat's Algorithm

Backlink: [@P-CS 312]

Tuesday, September 19, 2017 3:46 AM

Determines if number is prime or not by running k tests.

1. **Screenshot:**



2. **Code:**

```
namespace P1_Fermat
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();

         this.ActiveControl = m_tbInput;
         m_tbInput.Focus();
      }

      private void On_SolveClick(object sender, EventArgs e)
      {
         // Grab the input and the k-value and call pass to Primality Test
         Primality(Convert.ToInt32(m_tbInput.Text), Convert.ToInt32(m_tbK.Text));
      }

      private void Primality(int _num, int _numTests)
      {
         // Preventes entering an infinite loop below when generating numbers to test if _num is prime.
         if (_numTests > _num / 2)
         {
            MessageBox.Show("Please select a smaller k-value");
            return;
         }
         // Used to generate random test cases.
         Random rand = new Random((int)DateTime.Now.Ticks);
         // List of length k stores the integers used to test primality through modular exponentiation.
         List<int> baseNums = new List<int>();
         int baseNum;

         // Initial value that the answer is correctly identified as prime.
         double probability = 1.0;
         bool prime = true;

         // Ensures that the input number is tested k-times by unique values that are less than half the input value
         for (int i = 0; i < _numTests; i++)
         {
```

A: O(k)

```
            // test number is a random number between 1-n/2
            baseNum = rand.Next(1, _num / 2);
            // if the number has already been selected, then
            while (baseNums.Contains(baseNum))
            {
               baseNum = rand.Next(1, _num / 2);
            }
            baseNums.Add(baseNum);

            // Uses modularExponentiation to test if the gcd == 1. gcd == 1, then _num is prime. But if for only one test the gcd != 1, then it is not prime.
            if (modEx(baseNum, _num-1, _num) != 1)
            {
               prime = false;
               break;
            }
            // With each test, the probability that we are incorrect decreases by a factor of 2.
            probability /= 2;
         }

         // If all the tests completed with a gcd == 1, then the number is prime. Output the result.
         if (prime)
         {
```

```
            m_tbOutput.Text = "Yes with probability: " + (1.0 - probability);
        }
        else // The number is not prime.
        {
            m_tbOutput.Text = "No";
        }
    }

    /**
     * @param _baseNum: the value of the base number to be raised to _exp
     * @param _exp: the value of the exponenet
     *  @param _mod: the modular denominator
     */
    private int modEx(int _baseNum, int _exp, int _mod)
    {
        // Anything raised to the 0th power == 1
        if (_exp == 0) return 1;
        // Otherwise, we still need to bitshift. So call recursively and divide by two (bitshift one position).
        int z = modEx(_baseNum, _exp / 2, _mod);
        // If _exp is even, then we didn't lose the remainder when making the recursive call, and not adjustment is necessary.
        if (_exp % 2 == 0)
        {
            return (z * z) % _mod;
        }
        // Else if _exp is odd, then we need to adjust for the remainder lost when making the recursive call.
        // We do this by multiplying _baseNum to the square of the value z returned by the recursive call.
        else
        {
            return (_baseNum * z * z) % _mod;
        }

    }

    private void On_WindowKeyDown(object sender, KeyEventArgs e)
    {
        int num, k;
        if (e.KeyCode == Keys.Enter && int.TryParse(m_tbInput.Text, out num) && int.TryParse(m_tbK.Text, out k))
        {
            Primality(num, k);
        }
    }
  }
}
```

B: $O(n^3)$

3. **Time and Space Complexity:** (Subsections of code are labeled with red letters corresponding to the letters A & B below. Point C is the conclusion for the entire algorithm):
   - **A.** For loop that calls modex() k times = O(k) <= O(n/2), excluding the time for the modex() call. That will be calculated below.
   - **B.** Recursive Modex() method has runtime of $O(n^3)$, as there areat most n recursive calls, where n is the number of bits, and in each recursive call, n-bit numbers are multiplied. Multiplication is $O(n^2)$. So multiplication, performed n times, is $O(n^3)$.
   - **C.** The modex() method call is nested inside the loop described in point A. This means that it is called k times, or n/2 times. So, the total Big-oh for my implementation of the Primality test is $O(n/2 * n^3) = O(n^4)$

4. **Probability Algorithm:** With each successive test, the probability that we are correct increases at least by a factor of two. So, in calculating the probability that the number is prime, I first calculate the probability that I am wrongly identifying a prime number by starting with a probability of 1.0, then dividing by 2 with each test. When all the tests are completed, I subtract the final probability that I am wrong from 1.0, which results in the probability that I correctly identified the number as prime.