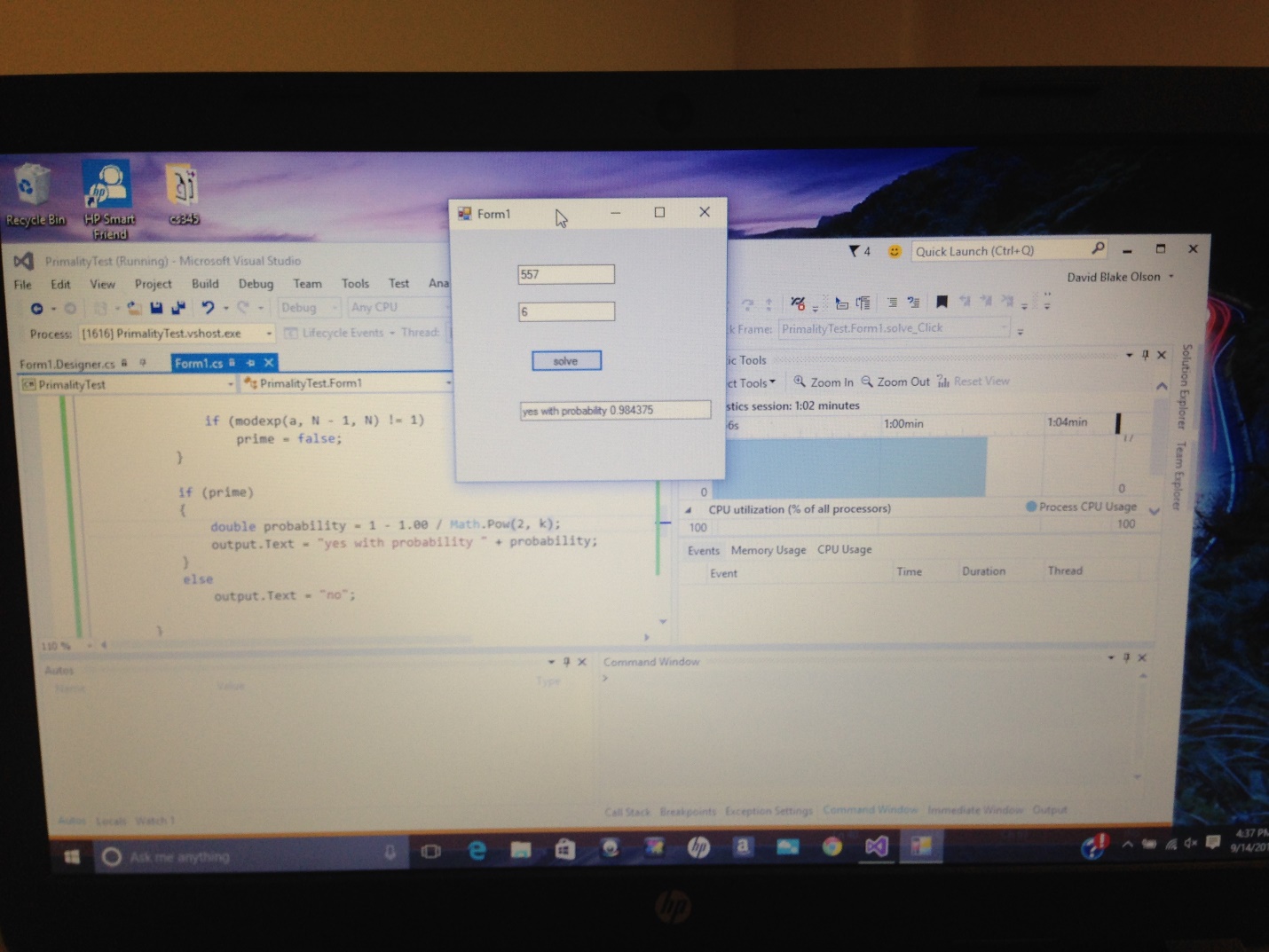
David Olson: Project 1

Screenshot:



Code:

        /\*

            on the button click, run my main primality function.  inputs are

            the number of random values to choose (k) and the number to test

            primality on (input/N)

        \*/

private void solve\_Click(object sender, EventArgs e)

       {

           fermatPrimality(Convert.ToInt32(k.Text), Convert.ToInt32(input.Text));

       }

       //complexity is k(modexp complexity) so O(n^3)

       private void fermatPrimality(int k, int N)

       {

           bool prime = true;

           Random random = new Random();

/\*

            loop through k times

            choose a new random number between 2 and N-1 each time

            compute a^(N-1) mod N each time and if that is ever not 1, N is not prime

\*/

           for(int i = 0; i < k; i++)

           {

               int a = random.Next(2, N - 1);

               if (modexp(a, N - 1, N) != 1)

                   prime = false;

           }

           if (prime)

           {

               //caluculate the sureness probability and output that along with yes

               double probability = 1 - 1.00 / Math.Pow(2, k);

               output.Text = "yes with probability " + probability;

           }

           else

               output.Text = "no";

       }

/\*

        function used to do a^(N-1) mod N from above (x is a, y is N-1)

        this is essentially straight from the book

        function complexity is O(n^2)

             ( the program will run this n times, so overall O(n^3))

\*/

       private int modexp(int x, int y, int N)

       {

           if (y == 0)

               return 1;

           int z = modexp(x, y / 2, N);

           //O(n^2)

           if ((y%2) == 0)

               return (z \* z) % N;

           else

               return (x \* z \* z) % N;

       }

Time/Space Complexity:

As was documented in my code, on an n-bit input, the modular exponentiation function is a O(n^2) function, because multiplication is involved. Since y is halved in every call, this function is called n times. The program itself, or in other words time spent in the fermat function’s call to modexp, is O(n^3). The number chosen as k is just a constant, which doesn’t affect the big-O class. Space complexity is the same as time complexity in this algorithm.

Probability Equation:

The equation I used was 1 - 1.00 / Math.Pow(2, k). The two zeros in 1.00 and the Math.pow are simply c# techniques to get this equation to work, but more simply the equation is 1-1/(k^2). This is the same as raising ½ to the kth power and subtracting it from one. Since there is a 50/50 chance fermat’s theorem is right on each try, the chance that we are wrong halves every time k is increased by one. (1/2, 1/4, 1/8, 1/16 …)