Ashkan Hosseini
CS315
10/08/2012

Bonus 1

I ran the algorithm with 20 randomly generated 16-bit numbers and checked if they were prime. The first time I ran the program none of the 20 numbers were prime. The second time I ran the program, the result was one prime number: 33809. Finally, I ran the program a third time and found four prime numbers: 47093, 49019, 33311, and 35099. In all tests I did not find an incorrect answer.

For the next step, I generated prime numbers of 64, 128, 256, and 512 binary digits. To do this, I randomly generated odd numbers of the desired bit length and used the first primality algorithm discussed in the class notes to find out if one of them is prime.

The results are shown below. The first line tells you the bit length of the number, the second line is the prime number generated (displayed in decimal), the fourth line is the number of numbers generated before finding a prime number, and the final line displays the amount of time in milliseconds it took to generate the prime number:

64-bit
13900564210866473377
Counter: 31
Time: 4884

128-bit
251693902162752412509642054293274144049
Counter: 9
Time: 7787

256-bit
85224276778237716592573621372326173157459671457013168432332996344599963273439
Counter: 39
Time: 246051

512-bit
11980713035465130034606405009814339708684066672833792335804082856999067254835920917601707641362085226585147391588650062251774815891699597368246453795262531
Counter: 122
Time: 5259551

In theory (from class notes) there will be *n* iterations (non-prime numbers) before we hit a prime number. However when looking at the results they don't match very well with the theory. For instance the *64-bit* prime was generated only after *31* non-prime numbers were generated. In fact this is true for each n-bit prime that was generated.

There are two main reasons why this happens. The first is that these numbers are purely randomly generated and we only ran one test per *n*-bit number. If there were more tests, then the results may have matched with the theory better. The second, and more important, reason this is happening is because for every randomly generated number, we make sure that number is odd. This

reduces *n* iterations before a prime to roughly $\frac{n}{2}$ iterations before a prime. Now the test for 64-bit makes more sense, since 31 is roughly $\frac{64}{2}$. Of course the other tests are still off by a little, but this is again because these numbers are randomly generated.

As for the complexity of the algorithm, we first look at the complexity of our primality algorithm, which is roughly $O(n^3)$ based on class notes. Now considering that the probability that we hit a prime is about n, we can say that the worst case running time increases $O(n^4)$. Unfortunately, similar to comparing the results to the theoretical for iteration count, it is not very feasible to find a match for the results to the theoretical running time because the numbers are randomly generated and because we only ran one test per bit count. We can however take the ratio of the time and counter, find the quad root of the result, and then take the ratio of the preceding bit count. By doing this we essentially get rid of the problem of randomness and find that the complexity is indeed $O(n^4)$.