>>> Solution for HW #3 for Capacity Planning (Fall 2001) <<<

```
Being able to compute prime numbers fast is important for cryptographic applications.
The programs prime1.c and prime2.c demonstrate a memory/CPU trade-off.  You should
instrument the two programs for CPU time and memory consumption.  For CPU time you can
use the methods of timeit.c.  For memory consumption, you really only care about the
malloc() size since it will clearly dominate memory consumption for interesting values of
N.

Do the following:

1) Plot CPU versus N and memory versus N for a single test machine (e.g., for your home
   PC).
```

Figure 1 shows a plot of CPU time versus N for prime1.c and prime2.c. All runs were done on a Pentium3, 800-Mhz, 128 Mbyte, Windows2000 PC. A trend line (2nd order polynomial fit) has been added to each series. The R-squared value shows a very good fit for the polynomial. Figure 2 shows memory usage for prime1.c and prime2.c.

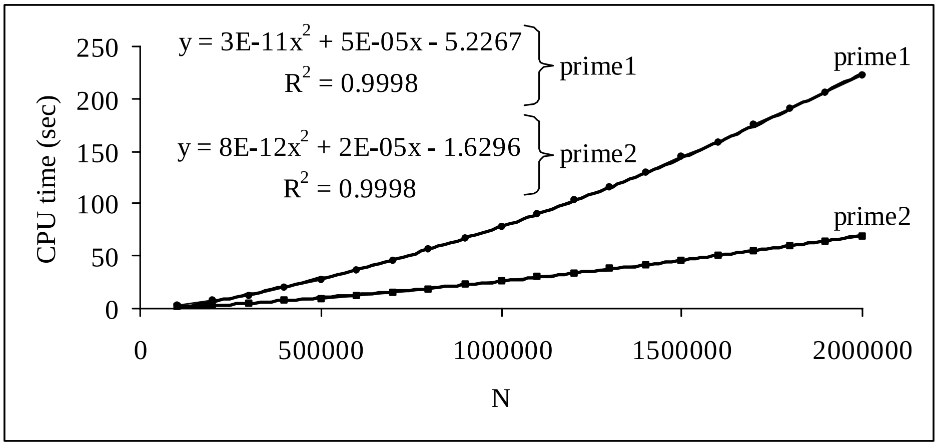

Figure 1 – CPU time versus N for prime1.c and prime2.c

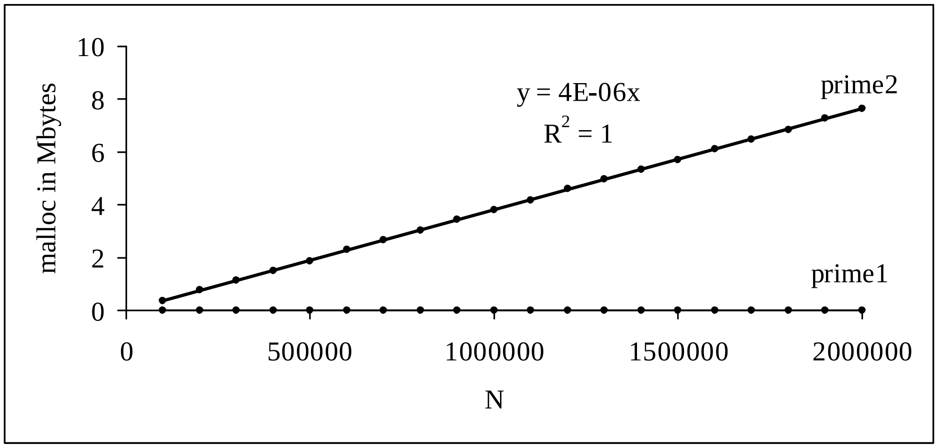

Figure 2 – Malloc space versus N for prime1.c and prime2.c

```
2) Come-up with a heuristic to estimate the CPU time for prime1.c and prime2.c as a
   function of N for this single test machine.  Pick a few values of N and show how
   close your heuristic is.
```

The graphs in Figure 1 show the 2nd order polynomial formulas that best match the plotted results (found using Excel's trend line). To test these formulas, runs for N = 2, 4, and 6 million were made. Table 1 shows the actual and predicted times. The heuristics based on the polynomial formula appear to be amazingly good (within 4%) for N = 2 million, but then increasingly overestimate the run time for larger N. The goodness of these predicitions is an interesting statement on the distribution of prime numbers in the positive integers. That the predicitions overestimate as N increases to "very large" values could be a result the density of prime number for larger values. <u>It would seem to indicate the the density of prime numbers is higher for large value (i.e., less values to test in order to find the Nth prime).</u> Testing this hypothesis would be straightforward by studying densities of prime numbers.

N	**prime1.c (actual)**	**prime1.c (predict)**	**prime2.c (actual)**	**prime2.c (predict)**
2 million	223.1 sec	214.8 sec	69.2 sec	70.4 sec
4 million	637.5	674.7	185.0	206.4
6 million	1177.2	1374.8	330.5	406.4

```
3) On what kind of machines might prime2.c to perform worse than prime1.c?
```

The program prime2.c consumes more memory than prime1.c. Thus, for machines with limited memory that employ disk swapping for virtual memory allocation, prime1.c may be faster (than prime2.c). For finding the Nth prime number, prime2.c allocates 4xN bytes of memory. For a N = 10 million, about 40 Megabytes of memory would need to be allocated. This is more memory than can be allocated on most 64 Megabyte system without swapping taking place.

```
4) The programs have a serious bug in them.  See if you can identify it.  A "serious bug"
   is one where a program can give an incorrect result.
```

If the Nth prime is larger than 2^32 (i.e., 32-bit integer value), the program will wrap-around to zero and give a very incorrect result (i.e., report a prime number much smaller in value than the actual Nth prime number).

```
5) Can this problem (i.e., that of finding the Nth prime number) be parallelized for
   better performance?  If yes, explain how and explain why it will be faster.
```

Computing the Nth prime number is a serial task. However, prime numbers are invariant (i.e., they are always the same) and can be precomputed. Thus, one could envision precomputing prime numbers for a very large N and then storing partitions (e.g., 1, … M, M+1, … 2*M, … N where M evenly divides N) in multiple machines. Then to "compute" the Nth prime number, the value would be looked-up in the appropriate machine. The basis for this solution is that storage is very inexpensive (e.g. a $200 40 gigabyte harddrive could store 10 billion (!) 32-bit values).

```
HINT: One way to come-up with a heuristic is to use curve-fitting.  Excel has a trendline
      feature the can help with this.  A better way to come-up with a heuristic is to
      have some understanding of the underlying problem (in this case, distribution of
      prime numbers).

EXTRA CREDIT (10 pts): Write a prime3.c that is even faster (by at least 10%) than
                       prime2.c.
```

I don't have a solution for this. It is possible that a Sieve of Erastosthenes approach may be faster. I look forward to seeing student solutions.