

CSci 426/526 — Simulation — Fall 2004
Examples

To help you get a sense of my expectations, enclosed are (partial) solutions to three typical exercises. These three exercises illustrate the range of “solution types” you will be expected to produce.

- **exercise 1.2.2:** This is a multi-part exercise that requires a modify-an-existing-program, do-some-computing, and explain-what-you-did solution. Note the explicit reference to relevant material in the text. As an alternative to the in-line format used to discuss how program `ssq1` was modified, you could provide a printed copy of the (complete) modified program with a highlighting marker used to clearly identify the modifications.
- **exercise 1.2.4:** This is a prove/demonstrate/show mathematical exercise that requires (of course) a mathematical solution.
- **exercise 2.1.1:** This is a multi-part exercise that requires a little bit of math and a build-from-scratch program with corresponding numerical results. Note that a printed copy of the program is included as part of the solution. Note also the explicit reference to relevant material in the text.

You will note that all the solutions are typeset. I happen to use \LaTeX for technical typesetting; some prefer to use plain \TeX . You can use either, or neither, or you can use any other document preparation system you are familiar with — provided it can do a decent job with technical typesetting. If you prefer, you can write your solutions (neatly) by hand. I care much more about the content than the style.

Exercise 1.2.2

(a) Modify program `ssq1` to output the additional statistics \bar{l} , \bar{q} and \bar{x} . (b) Similar to example 1.2.8, use this program to compute a table of \bar{l} , \bar{q} and \bar{x} for traffic intensities of 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, and 1.2. (c) Comment on how \bar{l} , \bar{q} and \bar{x} depend on the traffic intensity. (d) Relative to examples 1.2.8 and 1.2.9, if it is decided that \bar{q} greater than 5.0 is not acceptable, to *d.dd* precision what systematic increase in service times would be acceptable?

(a) The modification is based on the equations in theorem 1.2.1 or, equivalently, Little's equations (see page 1.2.9). In particular, relative to these equations, when program `ssq1` terminates there will be the following correspondence between mathematical quantities and program variables

c_n	↔	departure
$\sum_{i=1}^n w_i$	↔	sum.wait
$\sum_{i=1}^n d_i$	↔	sum.delay
$\sum_{i=1}^n s_i$	↔	sum.service.

Because

$$\bar{l} = \left(\frac{n}{c_n} \right) \bar{w} = \frac{1}{c_n} \sum_{i=1}^n w_i$$

with analogous equations for \bar{q} and \bar{x} , it follows that the only modification that must be made to program `ssq1` to output \bar{l} , \bar{q} and \bar{x} is the addition of the following three `printf` statements

```
printf("average nbr in the node .... = ", sum.wait / departure);
printf("average nbr in the queue ... = ", sum.delay / departure);
printf("utilization ..... = ", sum.service / departure);
```

(b) Using the data in the file `ssq1.dat` produces a traffic intensity of $\bar{s}/\bar{r} \simeq 0.721$. To change the traffic intensity to a specific value, say ρ , we can multiply each service time by $\rho/0.721$. At the expense of needing to re-compile each time ρ is changed, the easiest way to accomplish this modification to program `ssq1` is to add the line

- `#define FACTOR (0.7 / 0.721)`
to the beginning of the program (for $\rho = 0.7$), and modify the function `GetService` by changing the `return` to
- `return(FACTOR * s);`

With these changes to program `ssq1`, the following table of results is produced

traffic intensity	\bar{x}	\bar{q}	\bar{l}
0.60	0.60	0.96	1.56
0.70	0.70	1.68	2.38
0.80	0.80	2.99	3.79
0.90	0.90	7.49	8.39
1.00	0.99	25.85	26.84
1.10	1.00	69.44	70.44
1.20	1.00	105.86	106.86

(c) Etc.

(d) Etc.

Exercise 1.2.4

Complete the proof of theorem 1.2.1.

The solution to this exercise is based on the proof/derivation of the “wait” equation on page 1.2.9, modified to handle the “delay” and “service” equations.

To derive the delay equation, for each job $i = 1, 2, \dots, n$ define a *delay indicator function* $\theta_i(t)$ that is 1 when the i^{th} job is in the queue and is 0 otherwise

$$\theta_i(t) = \begin{cases} 1 & a_i < t < b_i \\ 0 & \text{otherwise.} \end{cases}$$

Then the number in the queue at time t is

$$q(t) = \sum_{i=1}^n \theta_i(t) \quad 0 < t < c_n$$

and so

$$\int_0^{c_n} q(t) dt = \int_0^{c_n} \sum_{i=1}^n \theta_i(t) dt = \sum_{i=1}^n \int_0^{c_n} \theta_i(t) dt = \sum_{i=1}^n (b_i - a_i) = \sum_{i=1}^n d_i.$$

The service equation can be derived in a similar way. Or, as an alternative, note that $l(t) = q(t) + x(t)$ for $0 < t < c_n$ and $w_i = d_i + s_i$ for $i = 1, 2, \dots, n$ so that we can use the wait and delay equations to derive the service equation, as illustrated

$$\begin{aligned} \int_0^{c_n} x(t) dt &= \int_0^{c_n} (l(t) - q(t)) dt \\ &= \int_0^{c_n} l(t) dt - \int_0^{c_n} q(t) dt \\ &= \sum_{i=1}^n w_i - \sum_{i=1}^n d_i \\ &= \sum_{i=1}^n (w_i - d_i) \\ &= \sum_{i=1}^n s_i. \end{aligned}$$

Exercise 2.1.1

For the tiny Lehmer generator defined by $g(x) = ax \bmod 127$, find all the full-period multipliers. (a) How many are there? (b) What is the smallest one? (a) Because the prime factorization of $m - 1 = 126$ is $2 \cdot 3^2 \cdot 7$ and because

$$\frac{(2-1)(3-1)(7-1)}{2 \cdot 3 \cdot 7} (2 \cdot 3^2 \cdot 7) = 36$$

from theorem 2.1.3 we know that there are 36 full-period multipliers.

(b) Provided the program parameter A is a full-period multiplier, the enclosed program, based on algorithms 2.1.1 and 2.1.2, generates and prints (“finds”) all of the 36 full-period multipliers. (If A is not a full-period multiplier the program will produce no output.) By inspection of the output we see that 3 is the smallest full-period multiplier.

```
/*
 * exercise 2.1.1
 *
 * the output is ...
 *
 *   3 116 109  92  86  12  83 112  55 114  48  78
 *   67  93 106  65  58  14 118  46  43   6  56  91
 *   57  45  39  97 110 101  53  96  29   7  23  85
 *
 */

#include <stdio.h>

#define M 127          /* for this modulus ...          */
#define A 3           /* 3 is a full-period multiplier */

long g(long x)
{
    return ((A * x) % M);
}

long gcd(long a, long b) /* from appendix B, use a > 0 and b > 0 */
{
    long r = a % b;
    while (r > 0) {
        a = b;
        b = r;
        r = a % b;
    }
}
```

```

};
return (b);
}

int main(void)
{
    long p = 1;
    long x = A;
    int ok;

    while (x != 1) { /* confirm that A is a full-period multiplier */
        p++;
        x = g(x);
    }
    if (p == M - 1)
        ok = 1;
    else
        ok = 0;

    if (ok) { /* use algorithm 2.1.2 */
        long i = 1;
        long j = 0; /* counts full-period multipliers */
        x = A;
        while (x != 1) {
            if (gcd(i, M - 1) == 1) {
                printf("%4ld", x);
                j++;
                if (j % 12 == 0)
                    printf("\n");
            }
            i++;
            x = g(x);
        }
    }

    return (0);
}

```