Programming Design, Spring 2015
Homework 8

Instructor: Ling-Chieh Kung
Department of Information Management
National Taiwan University

To submit your work, please upload a PDF file for Problem 1 and two CPP files for Problems 2 and 3 to PDOGS at http://pdogs.ntu.im/judge/. Each student must submit her/his individual work. No hard copy. No late submission. The due time of this homework is ***8:00am, May 4, 2014***. Please answer in either English or Chinese.

Before you start, please read Chapter 9 of the textbook.[1] The TA who will prepare the solution for this homework is ***Tammy Chang***.

Problem 1

(20 points; 5 points each) Consider the following structure

```
struct LotteryMachine
{
  int tokenCount;
  int* tokens;
  int outcomeCount;
  int* outcomes;
  void initialize();
  void initialize(int n, int t[]);
  bool draw(int m);
  void release();
};
```

which defines a lottery machine. In the machine, there are `tokenCount` tokens. These tokens' values are stored in a dynamic array, whose beginning address is stored in `tokens`. For example, for the case that we draw some tokens out of 42 tokens 1, 2, ..., and 42, we have `tokenCount` equals 42 and `tokens` pointing to a dynamic array storing 1, 2, ..., and 42. There are also outcomes of the last draw, whose length is `outcomeCount` and whose beginning address is `outcomes`.

(a) Implement the `initialize()` function to initialize `tokenCount` and `outcomeCount` to 0 and `tokens` and `outcomes` to `NULL`. Also implement the `release()` function to release the dynamic arrays pointed by `tokens` and `outcomes`.

(b) Implement the `initialize(int n, int t[])` function. Initialize `tokenCount` to `n`, `outcomeCount` to 0, `tokens` to `t`, and `outcomes` to `NULL`. "Initializing `tokens` to `t`" is not to let the two pointers contain the same address; your program should dynamically allocate a space and then copy values in `t` to the newly allocated space. You may assume that `n` is always the length of `t`.

(c) Implement the `draw(int m)` function. If `m` is nonpositive or greater than `tokenCount`, return false and do nothing else; otherwise, randomly draw `m` nonrepeating tokens out of the pool of tokens in `tokens`. Use `rand()`, `srand()`, and `time(0)` to generate random numbers. Include appropriate libraries. Store `m` in `outcomeCount` and the drawn tokens in `outcomes` (in any order you like). You will be graded based on the correctness and efficiency of drawing `m` nonrepeating tokens as well as memory management.

(d) Modify the structure into a class. Set appropriate visibility levels to members. Make constructors and a destructor if they help.

[1] The textbook is *C++ How to Program: Late Objects Version* by Deitel and Deitel, seventh edition.

Problem 2

(50 points) Recall that we have defined a structure

```
struct Randomizer
{
  int a;
  int b;
  int c;
  int cur;
  int rand();
};
int Randomizer::rand()
{
  cur = (a * cur + b) % c;
  return cur;
}
```

for creating random number generators. A randomizer that applies the $r_{i+1} = (ar_i + b) \mod c$ formula is called a *congruent* randomizer. While the formula is generally accepted, the implementation with a structure obviously has some drawbacks. In particular, it is hard to ensure that the values of those attributes are reasonable (e.g., we did not prevent `a` to become negative. Moreover, we do not prevent one from modifying the attributes while making random numbers. In this problem, we refine the structure into a class to remove these drawbacks.

The new class (still named as `Randomizer`) should have the following properties:

1. The values of `a`, `b`, and `c` should be given when an object is created. One may give arguments through a constructor to assign values to these three member variables. The user should either given three arguments or no argument. In the later case, `a`, `b`, and `c` should be set to 1001, 1999, and 32767, respectively. In any case, the value of `cur` should be initialized to 0.
2. After an object is created, no one can read or modify `a`, `b`, and `c` outside the class.
3. There should be a random seed setter `bool srand(int s);`. If `s` is nonnegative, the function assigns `s` to `cur` and returns true; otherwise it does not modify `cur` and returns false. No one can read `cur` outside the class.

You need to implement these into your class.

Input/output formats

There are 10 input files. In each file, there is a line of five nonnegative integers a, b, c, s, and n. Two consecutive values are separated by a white space. Here, a, b, and c are used to construct a randomizer as they form the random number generating formula. s is the random seed. By reading this line, your program should construct a randomizer with arguments a, b, c, and s and then output the first n integers generated by the randomizer in order. Two consecutive values should be separated by a white space. For example, for a line containing

```
10 4 9 0 5
```

as the input data, the output should be

```
4 8 3 7 2
```

in a single line. You may assume that $a > 0$, $b > 0$, $c > 0$, $1 \leq n \leq 100$, and all input numbers are no greater than 10000.

What should be in your source file

You are required to implement a class `Randomizer` by revising the structure provided above to provide the above mentioned properties. This is the first time for you to implement a class. That is why we assign a not-so-complicated task to you and expect you to focus more on appreciating the benefit of building a class. Most grades for this problem will be determined by the content of your program, not the output.

Your .cpp source file should contain C++ codes that will both read testing data and complete the above task. For this problem, you are NOT allowed to use techniques not covered in lectures. You should write relevant comments for your codes.

Grading criteria

20 points for this program will be based on the correctness of your output. PDOGS will compile your program, feed testing data into your program, and check the correctness of your outputs. For each set of input data, if your program outputs correctly without violating the space limit, you get 2 points.

30 points for this program will be based on how you write your program, including the logic and format. Please try to write a robust, efficient, and easy-to-read program.

Problem 3

(30 points) A congruent randomizer generate "pseudorandom" numbers according to the formula $r_{i+1} = (ar_i + b) \mod c$. It is not very hard to see that whether these pseudorandom numbers look random depends on the values of a, b, and c. As an example, if $a = 1$ and $b = 1$, then for a given c, the sequence of pseudorandom numbers will just be increasing until we reach $c - 1$. As another example, if $a = 10$, $b = 5$, and $c = 30$, then only 5, 15, and 25 are possible values that may be generated. Before we use a randomizer in practice, we really need to test the randomness of the randomizer.

In this problem, we introduce a randomness test for a sequence of numbers. This method, the *run test*, will be formally introduced to you in the course *Statistics II* in the IM department (and most equivalent introductory statistics courses). The idea is the following:

- Given a sequence of values $(x_1, x_2, ..., x_n)$, first we calculate their median M. Roughly speaking, M is a value such that half of x_is are above M and half are below M. More precisely, M is defined as
$$M = \begin{cases} x_{(\frac{n+1}{2})} & \text{if } n \text{ is odd} \\ \dfrac{x_{(\frac{n}{2})} + x_{(\frac{n}{2}+1)}}{2} & \text{if } n \text{ is even} \end{cases},$$
where $x_{(i)}$ is the ith large value in $(x_1, x_2, ..., x_n)$.
- We then generate a sequence of values $(y_1, y_2, ..., y_n)$ such that
$$y_i = \begin{cases} 1 & \text{if } x_i > M \\ 0 & \text{if } x_i = M \\ -1 & \text{if } x_i < M \end{cases}$$
for all $i = 1, ..., n$.
- y_is allow us to count the number of "runs" in this sequence, where a run is a consecutive sequence of numbers with no strict sign change. More precisely, let $(z_1, z_2, ..., z_m)$ be a sequence of 1 and -1 generated by removing 0s in $(y_1, y_2, ..., y_n)$ (therefore, we have $m \leq n$) and $(w_2, w_3, ..., w_m)$ be such that
$$w_i = \begin{cases} 1 & \text{if } z_{i-1} \neq z_i \\ 0 & \text{otherwise} \end{cases}$$
for all $i = 2, ..., m$. Then the number of runs is $\sum_{i=1}^{m} w_i + 1$.

As an example, consider a sequence $x = (1, 3, 2, 10, 9, 5, 6, 7, 8, 9)$. For this sequence, the median is $\frac{6+7}{2} = 6.5$. Then we have y, z, and w as recorded in the following table:

i	1	2	3	4	5	6	7	8	9	10
x_i	1	3	2	10	9	5	6	7	8	9
y_i	-1	-1	-1	1	1	-1	-1	1	1	1
z_i	-1	-1	-1	1	1	-1	-1	1	1	1
w_i	–	0	0	1	0	1	0	1	0	0

Therefore, the number of runs of x is 4. As another example, for the sequence $x = (1, 8, 4, 4, 6, 6, 2, 8, 9, 9)$, we have $M = \frac{6+6}{2} = 6$ and

i	1	2	3	4	5	6	7	8	9	10
x_i	1	8	4	4	6	6	2	8	9	9
y_i	-1	1	-1	-1	0	0	-1	1	1	1
z_i	-1	1	-1	-1	-1	1	1	1	–	–
w_i	–	1	1	0	0	1	0	0	–	–

The number of runs is also 4.

Given a sequence x with n numbers, obviously the number of runs is between 1 and n. If we have a very small number of runs, we tend to believe that the sequence is "not so random," as we may expect a lot of small/large numbers once we see a small/large number. Similarly, if we have a very large number of runs, we also tend to believe that the sequence is not random. To apply this principle to test a random number generator, we need to define what are "too small" and "too large." Statisticians have studied this problem and provided scientific ways to obtain a range (n_1, n_2) such that when the number of runs is outside (n_1, n_2), we conclude (with some level of confidence) that the random number generator does not pass the run test and is not a valid "random" number generator. For $n = 10000$, for example, a common range is $(n_1, n_2) = (12, 990)$. The way to calculate n_1 and n_2 will be taught in Statistics II.[2]

In this problem, you will be given a few sets of a, b, and c. You will then use the run test to test whether the congruent random number generators defined by those given arguments pass the run test.

Input/output formats

There are 15 input files. In each file, there are two lines of input data. In the first line, three numbers a, b, and c are given following the same restriction as in Problem 2. In the second line, there are $m + 1$ nonnegative integers m, s_1, s_2, ..., and s_m. Here $m \in \{1, 2, ..., 10\}$ is the number of random seeds and s_1 to s_m are the m random seeds. In each line, two consecutive numbers are separated by a white space. You may assume that $m > 0$ and these random seeds are all smaller than c.

Given an input file, you should use a, b, and c to define a random number generator. Then you should use the m random seeds to generate m sequences of random numbers. For each seed, please generate exactly 1000 random numbers (and, as always, the seed is NOT the first random number; it is used to generate the first random number!). Then please count the number of runs for each sequence. Your program should output the m numbers of runs, separated by white spaces.

As an example, for a file containing

```
69 16 3719
2 11 0
```

as the input data, the output should be

[2]Note that run test is just testing a random number generator in one aspect. Therefore, we do not conclude that a random number generator is valid when it passes the run test; we only conclude that it is invalid when it fails to pass.

```
517 499
```

in a single line.

What should be in your source file

Your .cpp source file should contain C++ codes that will both read testing data and complete the above task. For this problem, you may use whatever technique you like. You should write relevant comments for your codes.

Grading criteria

30 points for this program will be based on the correctness of your output. PDOGS will compile your program, feed testing data into your program, and check the correctness of your outputs. For each set of input data, if your program outputs correctly without violating the space limit, you get 2 points.