CS-102 Design Principles: Project 3

Due date: Monday, February 13 at 11:00pm

This project will show you how computers generate numbers that appear to be random, but are, in fact, what we call *pseudo-random*. A sequence of pseudo-random numbers is generated by an algorithm that takes a *seed* number as its input; all sequences that begin with the same seed will have the same entries throughout. We will use these pseudo-random sequences to generate what appear to be poems written in Japanese (but are, in fact, nonsensical, as anyone who speaks Japanese will immediately see).

Please start by accepting the assignment at:

https://classroom.github.com/a/rXNjuBGz

which gives you a GitHub repository for your project.

How to submit:

- ☐ Add/commit/push your final code to your GitHub repository; this is the code that will be graded.
- ☐ On Gradescope, click on the assignment for this project, and submit your GitHub repository. (Select *master* as your branch.)

Pseudo-random numbers

Random numbers have many uses in computer science, perhaps most importantly for running simulations of real-world events (such as the spread of a virus). To produce true random numbers, a computer can sample from the frequencies of ambient noise (and other such environmental data), but often we can sufficiently meet our need for random numbers using mathematics.

The best that mathematics can do, however, is to create what we call *pseudo-random* numbers. Clever algorithms can produce sequences of pseudo-random numbers that do a fairly good job at passing statistical tests for randomness. This project introduces you to one such algorithm: the *linear congruential generator*. This pseudo-random number generator (abbreviated PNG) is just one among many.

The input to this PNG is a *seed* integer. The PNG uses the seed, along with three constants `A, C,` and `MOD` and a bit of math, to return a pseudo-random number. That bit of math is included in the starter code for this project, and it looks like this as a function:

```
void generate_prnd(int &seed) {
    seed = (A * seed + C) % MOD;
}
```

The starter code uses the following settings for the constants:

```
const int A = 125;
const int C = 11;
const int MOD = 1024;
```

So, for example, if we send the seed `100` to the function `generate_prnd`, it will return `223`, because:

```
(125 * 100 + 11) % 1024 = 12511 % 1024 = 223
```

We overwrite the seed with this new value every time that we call the function. Thus, the next time that our program requires a pseudo-random number, we will send `223` to the function, and it will return `238` — and so on.

This program illustrates what we have learned thus far, and you should run this code before asking us any questions:

```
#include <iostream>

const int A = 125;
const int C = 11;
const int MOD = 1024;

void generate_prnd(int &seed) {
    seed = (A * seed + C) % MOD;
}

int main() {
    int seed = 100;
    for (int i = 0; i < 15; i++) {
        std::cout << seed << " ";
        generate_prnd(seed);
    }
    std::cout << std::endl;
    return 0;
}
```

The output of this program is a sequence of fifteen pseudo-random numbers that are generated by the seed `100`, using our settings for `A, C,` and `MOD`. Note how `223` and `238` are the first two in the sequence, just like we calculated above:

```
100 223 238 65 968 179 882 693 620 711 822 361 80 795 58
```

These numbers appear to be randomly chosen. Perfect!

If we all use the same seed and settings for the constants, every one of us will generate the same sequence of pseudo-random numbers. Thus, our own output will appear to be random, but it will perfectly match everyone else's output. Our simulations come out the same every time if we use a particular seed, which is useful to researchers (scientists and statisticians) who use computers to simulate real-world phenomena.

We will be using pseudo-random numbers to generate something non-useful but fun: poems that appear to be written in the Japanese language.

Hiragana Background

A poem consisting of seventeen syllables, arranged in three lines with five, seven, and five syllables respectively, is a *haiku*. The origins of these poems stretch back to seventeenth century Japan. Here is a haiku by Yosa Buson, a Japanese poet of the 1700s, in Japanese characters, then as hiragana syllables, and finally in English:

春の海　ひねもすのたり　のたりかな

Haruno-umi / Hinemosu-notari / Notarikana

Spring ocean / Swaying gently / All day long

If you count syllables in the English version, you will find that the syllable count is non-standard. However, if you check the syllables in the Japanese version, the syllable count is correct. Hiragana act a bit like an alphabet for the Japanese language; most are either a single vowel or a consonant followed by a vowel, so it is not difficult for English speakers to count syllables.

By contrast, it is more difficult to count syllables in words written in English. For example, the words *wherewithal* and *iota* have the same number of syllables. For this reason, we will use hiragana to write poems rather than use English.

You can view hiragana here:

`https://freejapaneselessons.com/printable-hiragana-chart.pdf`

You may notice that the first character in Yosa Buson's haiku is not in the chart; this is because *haru* is a combination character that means *spring*. So, things are a bit more complicated than we have briefly summarized here.

The starter code includes a subsection of the hiragana chart, and you should not change it:

```
string HIRAGANA[9][5] = {{ "a",   "i",   "u",  "e",  "o"},
                         {"ka",  "ki",  "ku", "ke", "ko"},
                         {"sa", "shi",  "su", "se", "so"},
                         {"ta", "chi", "tsu", "te", "to"},
                         {"na",  "ni",  "nu", "ne", "no"},
                         {"ha",  "hi",  "hu", "he", "ho"},
                         {"ma",  "mi",  "mu", "me", "mo"},
                         {"ra",  "ri",  "ru", "re", "ro"},
                         {"ya",  "wa",  "yu",  "n", "yo"}
                        };
```

Program Input

Your goal is to write a program that accepts some data from standard input, then outputs a poem (in nonsense Japanese) that is based on that data. All of our input data will be in this form:

```
43
3 5 7 5
1 0
```

The lone number in the top row is the seed.

The first number in the second row indicates the number of lines that the poem should have. The rest of the numbers in the second row indicate the number of syllables that each respective line in the poem should have. For this example, our output should be a three-line poem where the first line has 5 syllables, the second has 7 syllables, and the third has 5 syllables — in other words, a haiku!

The third row will always have a pair of numbers, each one being 1 or 0. You can think of these as *on* and *off* settings. The first number will signal if the first character in each line of the poem should be capitalized, and the second will signal if the poem should be written vertically. More on these settings later; for now, just note that the third row in the example is signalling that we *will* capitalize, and that we will *not* write the poem vertically.

One more input example:

```
1009
4 10 10 10 10
0 1
```

This input should prompt our program to use the seed `1009` and output a four-line poem, each line having ten syllables, with no capitalization, and written vertically.

Here we define the bounds on each input. All inputs will be positive integers, and *between* is inclusive of the bounds:

seed: between 1 and 1023
number of lines: between 1 and 15
number of syllables in a line: between 1 and 40

Program Output

The output of your program should simply be a poem that obeys the restrictions in the input, and uses randomness to create spaces that will form (nonsense) words. For example, the poem might look like:

```
Nuka kinuno
Uke shite imoku
Eokame chi
```

This poem has the `5 7 5` syllable structure of a haiku.

In order to force everyone's output to be the same for a given input, we must take care to follow the same procedure for generating a poem. Here is the procedure that you must follow:

Start each line of the poem by using the PNG to choose a number of syllables that is between 1 and 5 (inclusive); the upper limit of 5 is constant no matter what the input is or how many syllables are left to fill in a line. If this number ever exceeds the remaining syllables in this line, then the final word in this line should use up those remaining syllables. Create a word that has that many syllables: for each syllable, use the PNG to generate a random row index, and then use the PNG again to generate a random column index; use the syllable at that location in the hiragana chart.

Let's look at an example. Say that our seed is `100`, and that the input dictates that the first line of our poem must have five syllables, like the first line of a haiku. We ask the PNG for a random number, and it returns `223` (as it must, see above). Our program needs to convert this number to a number from the range 1 to 5, so for these two tasks we use:

```
generate_prnd(seed);
int syl_in_word = seed % 5 + 1;
```

Because `223 % 5 + 1` equals `4`, we know that our first word in the first line of our poem must have four syllables.

We choose the first syllable from the hiragana chart by requesting two more numbers from our PNG (using the new seed `223` to generate the first of the two numbers); as we saw above, `223` will generate `238`. There are nine rows in the hiragana chart; `238 % 9` equals `4` so we use the row with index 4.

Now we use the PNG with seed `238` to generate the number `65`. There are five columns in the chart; `65 % 5` equals `0` so we use the column with index 0. The syllable at row index 4 and column index 0 is *na*, so that is the first syllable of our four-syllable word.

We continue to create syllables in this way until we have chosen four syllables and thus completed the first word: *nahoewa*.

Because the input data specified that the first line of the poem must have five syllables, we have one syllable to go. Despite the fact that we *know* that we only want a one syllable word, we nevertheless request a new random number from the PNG. At this point, our seed is `711`, and the PNG returns `822`. Because `822 % 5 + 1` equals `3`, we plan that our next word in line one must have three syllables; however, we compare this to the number of syllables that we know we still need (that is, one syllable) and therefore decide to create a single syllable word.

We do the same for each line of the poem. For the input

```
100
3 5 7 5
1 0
```

the full poem is:

```
Nahoewa ka
Ne nnuno suyumu
Kooya sehe
```

If you have followed the instructions perfectly, your output should be exactly this, even though we have used a (pseudo) random process to generate it.

Capitalization and Vertical Writing

The third line of the input file has two integers.

If the first is `0` then the first word of each line should not be capitalized, and if the first is `1` then the first word of each line should be capitalized.

If the second is `0` then the output poems should read top to bottom and left to right, the way that English is typically written. All of the output examples above are in this format.

However, if the second is `1` then the output poems should read right to left by columns, starting in the upper right and reading each column downwards, which is how Japanese is sometimes written. So in this mode, the first letter of the poem should be in the upper right of the output, with the rest of the first line reading down that column. The first letter of the second line of the poem should be to the *left* of the first letter of the poem, with the rest of the second line reading down that column. This continues until the end of the poem; the final line of the poem will be in the far left column of the output.

For example, the input

```
55
3 5 7 5
0 0
```

generates the haiku

```
roriha kite
sa u na yahaura
te shikukeo
```

would look like the following, written in this vertical style. Note that the periods represent spaces, and that your actual output would print a space in every spot where there is a period:

```
tsr
eao
..r
sui
h.h
ina
ka.
u.k
kyi
eat
ohe
.a.
.u.
.r.
.a.
```

You will need to use spacing to deal with spaces between words and also with unequal line lengths, but there should be no extra spaces used otherwise. In the example just above, there *are* four spaces in the rightmost column below the final *e*.

There should be no spaces at the right-hand ends of any lines, whether the poem is displayed vertically or not. You will, however, need to place spaces strategically in order to display a poem vertically.

You may use the `iostream` and `string` libraries (and any of their methods), but no other libraries.

Details

The tests will be organized as follows:

- ☐ 8 tests with settings `0 0` (no capitals, no vertical)
- ☐ 2 tests with settings `1 0` (capitals, no vertical)
- ☐ 2 tests with settings `0 1` (no capitals, vertical)

There will be no tests with settings `1 1`. The structure of the testing might inspire you to develop your code to meet those challenges one at a time.

Grading

Your grade will come from:

75%	correctness + passing the tests
25%	style

Projects passing zero tests will receive at most 70% of the project grade (assuming perfect style). Passing tests increases the grade proportionally up from 70%; however, this assumes that your code isn't written to circumvent testing (for example, by "hard coding" answers to the tests), which is a violation of the honor code.

The elements of style that we will assess on this project are:

- ☐ well-designed helper functions: each function fits on screen (so, no more than approximately 20 lines) and each function does one job
- ☐ no magic numbers
- ☐ proper indentation
- ☐ correct placement of square brackets, curly braces, and parentheses
- ☐ correct spacing around operators
- ☐ reasonable variable names in proper naming style: using `lower_case` naming in local scope, and `UPPER_CASE` naming for global constants; be sure to avoid single-character variable names.
- ☐ all file and function header comments are present, in the required format, and filled out correctly