Programming Assignment: Trains

1. Train simulation

Write a program that simulates trains at a switching station.

At the beginning of the simulation, there are *n* trains. Each starts with *c* cars. Each car has a fixed size *s* that limits how much it can carry. The sizes of the cars in each train are 1 ... *c*, from the front to the back of the train. Trains are loaded with more valuable materials toward the back of the train, so the *value* of a train is the sum of *p***s*, where *p* is the position of each car (starting at 1) and *s* is its size. So the starting *value* of each train is 1*1 + 2*2 + ... + *c***c* = *c*(*c*+1)(2*c*+1)/6.

At each step, the simulation tosses an *n*-sided die, which indicates which train is the donor of this step. The simulation then tosses tosses a 2-sided die, indicating whether to remove a car from the front (if the die shows 1) or the back of the donor train (if the die shows 2). Ths simulation then tosses an *n*-sided die, indicating which train accepts the donated car. It is possible for the donor to get its own car back. The accepting train puts the donated car at its front. When a car moves to a new position, it retains its original size, but the values of the train it left and the train it joins do change, because cars are now in a different position.

The simulation ends when any train becomes empty or *t* steps, whichever comes first.

2. What to do

Write a program called `trains` that simulates the switching station. This program must take *n*, *c*, and *t* as command-line parameters. The program must fail gently if any of these parameters is ridiculous, such as *n* = 0 or *c* = -5. The program must read a stream of integers from standard input as its source of randomness; this feature allows us to test the program in a deterministic way. **After each step**, the program should print (to standard output) which train donates a car (front or back) to which recipient and the value of each train after the step. **When the simulation finishes** (either because a train has become empty or the number of steps has reached its limit), the program prints which train has the largest value (if there is a tie, select the earliest train in the collection).

The program must use a doubly-linked list to implement the trains You **must not** use a library routine (such as C++ `std::list` or Java `java.util.LinkedList`) for linked lists; you must write your own routines. This restriction is frustrating, but the point of the exercise is to learn how these lists work. You may use array, **struct**, and **class** elements, but not C++ `vector`*s* or Java `List`*s*. \fBDo not use code you find on the Internet; you will get 0 points and be charged with academic misconduct if we detect any such code.\fP

Try to make the program modular, with separate routines for the main loop, for reading the next random number, for evaluating the value of a train, and for donating and accepting a car.

3. Useful tools

You have access to some useful tools. First, there is a sample `Makefile` at `http://www.cs.uky.edu/~raphael/courses/CS315/prog1/Makefile`. It has a *run* target that compiles the program (either `trains.c` or `trains.cpp`) and runs it. Feel free to modify `Makefile`.

A second tool is `randGen.pl`, a program to generate an appropriate stream of pseudo-random numbers in the range 0 .. 2^{31}–1. The `Makefile` automatically gets a copy of this tool for you; it is stored at `http://www.cs.uky.edu/~raphael/courses/CS315/utils/randGen.pl`. This program, written in Perl, takes an optional integer parameter to seed the random-number generator if you want reproducable results. Here is how

you can invoke it with *trains*, seeding the random-number generator to `82` and setting `n=4`, `c=20`, and `t=100`:

```
randGen.pl 82 | trains 4 20 100
```

To convert a large pseudo-random integer x to one in the range 1..y, use this formula:
$(x \% y) + 1$. Here, `%` is the modulus operator.

Warning: If you run `randGen.pl` by itself, it generates an unbounded list of numbers. You should always pipe its output into another program, such as `trains` or `less`.

You can also get a working program that satisfies the specifications at `http://www.cs.uky.edu/~raphael/courses/CS315/prog1/workingTrains`. The `Makefile` mentioned above automatically gets a copy of this file for you if you make *runWorking*.

4. What to hand in

Submit via Canvas a Zip file containing (1) a copy of your program, (2) `Makefile`, (3) a `README` file with external documentation, and (4) a file `output.txt` created via

```
% randGen.pl 49 | trains 5 6 1000 > output.txt
```

The `Makefile` mentioned above has a *zipAll* target that creates a package ready to submit.

5. Extra credit ideas

Make sure your program is running correctly before attempting any extra credit. You get no extra credit for programs submitted late. If you do extra credit, mention what you have done in your `README` file.

1. Modify the rules of the simulation. For instance, trains can maintain order sorted by size, or they can always donate the largest-size car. If you modify the rules, you must also turn in a version of the program using the standard rules.
2. Compute and print running statistics, such as the average train value or the entropy of the train values.
3. You may invent other extra-credit ideas, but be sure to document them.