In this lab you will simulate scheduling in order to see how the time required depends on the scheduling algorithm and the request patterns. The lab is due 2 March 2017.

A process is characterized by just four non-negative integers A, B, C, and M. A is the arrival time of the process and C is the total CPU time needed. A process execution consists of computation alternating with I/O. I refer to these as CPU bursts and I/O bursts. We make the simplifying assumption that, for each process, the CPU burst times are uniformly distributed random integers (UDRIs) in the interval $(0,B]$. To obtain a UDRI t in some interval $(0,U]$ use the function randomOS(U) described below. If t, the value returned by randomOS(), is larger than the total CPU time remaining, set t to the remaining time.

The I/O burst time for a process is its preceding CPU burst time multiplied by M.

You are to read a file describing n processes (i.e., n quadruples of numbers) and then simulate the n processes until they all terminate. The way to do this is to keep track of the state of each process and advance time making any state transitions needed. At the end of the run you first print an identification of the run including the scheduling algorithm used, any parameters (e.g. the quantum for RR), and the number of processes simulated. You then print for each process

- A, B, C, and M.
- Finishing time.
- Turnaround time (i.e., finishing time - A).
- I/O time (i.e., time in Blocked state).
- Waiting time (i.e., time in Ready state).

Then print the following summary data.

- Finishing time (i.e., when all the processes have finished).
- CPU Utilization (i.e., percentage of time some job is running).
- I/O Utilization (i.e., percentage of time some job is blocked).
- Throughput, expressed in processes completed per hundred time units.
- Average turnaround time.
- Average waiting time.

You must simulate each of the following scheduling algorithms, assuming, for simplicity, that a context switch takes zero time. You need only do calculations every time unit (e.g., you may assume nothing exciting happens at time 2.5).

- FCFS.
- RR with quantum 2.
- Uniprogrammed. Just one process active. When it is blocked, the system waits.
- SJF (This is *not* preemptive, but is *not* uniprogrammed, i.e., we do switch on I/O bursts). Recall that SJF is shortest job first, not shortest burst first. So the time you use to determine priority is the total time remaining (i.e., the input value C minus the number of cycles this process has run).

For each scheduling algorithm there are several runs with different process mixes. A mix is a value of n followed by n A, B, C, M quadruples. Here are the first two input sets. The comments are *not* part of the input.

```
1  0 1 5 1           about as easy as possible
2  0 1 5 1  0 1 5 1  should alternate with FCFS
```

All the input sets, with their corresponding outputs are on the web.

The simple function randomOS(U), which you are to write, reads a random non-negative integer X from a file named random-numbers (in the current directory) and returns the value $1 + (X \bmod U)$. I will supply a file with a large number of random non-negative integers. The purpose of standardizing the random numbers is so that all correct programs will produce the same answers.

Breaking ties

There are two places where the above specification is not deterministic and different choices can lead to different answers. To standardize the answers, you must do the following.

1. A running process can have up to three events occur during the same cycle.
   i. It can terminate (remaining CPU time goes to zero).
   ii. It can block (remaining CPU burst time goes to zero).
   iii. It can be preempted (e.g., the RR quantum goes to zero).

They must be considered in the above order. For example if all three occur at one cycle, the process terminates.

2. Many jobs can have the same ‘‘priority’’. For example, in RR several jobs can become ready at the same cycle and thus have the same priority. You must decide in what order they will subsequently be run. These ties are broken by favoring the process with the earliest arrival time *A*. If the arrival times are the same for two processes with the same priority, then favor the process that is listed earliest in the input. We break ties to standardize answers. We use this particular rule for two reason. First, it does break all ties. Second, it is very simple to implement. It is not especially wonderful and is not used in practice. I refer to this method as the ‘‘lab 2 tie-breaking rule’’ and often use it on exams for the same two reasons.

Running your program

Your program must read its input from a file, whose name is given as a command line argument. The preceding sentence is a *requirement*. The format of the input is shown above (but the ‘‘comments’’ are not necessary); sample inputs are on the web. Be sure you can process the sample inputs. Do *not* assume you can change the input by adding or removing whitespace or commas, etc.

Your program must send its output to the screen (printf() in C; System.out in java).

In addition your program must accept an optional ‘‘--verbose’’ flag, which if present precedes the file name. When --verbose is given, your program is to produce detailed output that you will find useful in debugging (indeed, you will thank me for requiring this option). See the sample outputs on the web for the format of debugging output. So the two possible invocations of your program are

```
<program-name> <input-filename>
<program-name> --verbose <input-filename>
```

My program also supports an even more verbose mode (show-random) that prints the random number chosen each time. This is useful, but your program is **not** required to support it.