This lab was put together by Professor Allan Gottlieb

In this lab you will simulate demand paging and see how the number of page faults depends on page size, program size, replacement algorithm, and job mix (job mix is defined below and includes locality and multiprogramming level).

The idea is to have a driver generate memory references and then have a demand paging simulator (called pager below) decide if each reference causes a page fault. Assume all memory references are for entities of one fixed size, i.e., model a word oriented machine, containing M words. Although in a real OS memory is needed for page tables, OS code, etc., you should assume all M words are available for page frames.

The program is invoked with 6 command line arguments, 5 positive integers and one string

- M, the machine size in words.
- P, the page size in words.
- S, the size of each process, i.e., the references are to virtual addresses 0..S-1.
- J, the "job mix", which determines A, B, and C, as described below.
- N, the number of references for each process.
- R, the replacement algorithm, LIFO (**NOT** FIFO), RANDOM, or LRU.

The driver reads all input, simulates N memory references per program, and produces all output. The driver uses round robin scheduling with quantum q=3 (i.e., 3 references for process 1, then 3 for process 2, etc.).

The driver models locality by ensuring that a fraction A of the references are to the address one higher than the current (representing a sequential memory reference), a fraction B are to a nearby lower address (representing a backward branch), a fraction C are to a nearby higher address (representing a jump around a "then" or "else" block), and the remaining fraction (1-A-B-C) are to random addresses. Specifically, if the current word referenced by a process is w, then the next reference by this process is to the word with address

- w+1 mod S with probability A
- w-5 mod S with probability B
- w+4 mod S with probability C
- a random value in 0..S-1 *each* with probability (1-A-B-C)/S

Since there are S possible references in case 4 each with probability (1-A-B-C)/S, the total probability of case 4 is 1-A-B-C, and the total probability for all four cases is A+B+C+(1-A-B-C) =1 as required.

There are four possible sets of processes (i.e., values for J)

J=1: One process with A=1 and B=C=0, the simplest (fully sequential) case.

J=2: Four processes, each with A=1 and B=C=0.

J=3: Four processes, each with A=B=C=0 (fully random references).

J=4: Four Processes. The first process has A=.75, B=.25 and C=0;
the second process has A=.75, B=0, and C=.25;
the third process has A=.75, B=.125 and C=.125;
and the fourth process has A=.5, B=.125 and C=.125.

The pager routine processes each reference and determines if a fault occurs, in which case it makes this page resident. If there are no free frames for this faulting page, a resident page is evicted using replacement algorithm R. The algorithms are **global** (i.e., the victim can be any frame not just ones used by the faulting process). Because the lab only simulates demand paging and does not simulate the running of actual processes, I believe you will find it easiest to just implement a frame table (see next paragraph) and not page tables. My program is written that way. (This is advice not a requirement.)

As we know **each** process has an associated page table, which contains in its i^{th} entry the number of the frame containing this process's i^{th} page (or an indication that the page is not resident). The frame table (there is only one for the entire system) contains the reverse mapping: The i^{th} entry specifies the page contained in the i^{th} frame (or an indication that the frame is empty). Specifically the i^{th} entry contains the pair (P, p) if page p of process P is contained in frame i.

The system begins with all frames empty, i.e. no pages loaded. So the first reference for each process will definitely be a page fault. If a run has D processes (J=1 has D=1, the others have D=4), then process k ($1 \le k \le D$) begins by referencing word 111*k mod S.

Your program echos the input values read and produces the following output. For each process, print the number of page faults and the average residency time. The latter is defined as the time (measured in memory references) that the page was evicted minus the time it was loaded. So at eviction calculate the current page's residency time and add it to a running sum. (Pages never evicted do not contribute to this sum.) The average is this sum divided by the number of **evictions**. Finally, print the total number of faults and the overall average residency time (the total of the running sums divided by the total number of evictions).

Use the same file of random numbers as in lab2.

Good luck!

Notes:

1. Despite what some books may say, the % operator in C, C++, and Java is the remainder function **not** the mod function. For most (perhaps all) C/C++/Java compilers, (-2)%9 is -2; whereas (-2) mod 9 = 7. So to calculate (w-5) mod S above, write (w-5+S)%S.
2. The big issue in this lab is the REplacement of pages. But the placement question does arise early in the run when there are multiple free frames. It is important that we all choose the same free frame so that you can get the benefit of my answers and debugging output and so that everyone will be referring to the same situation. I choose the **highest** numbered free frame; you must do so as well.
3. Since random numbers are involved, we must choose the random numbers in the same order. Here is a non-obvious example. In the beginning of your program you set the referenced word for each job to be 111*k as described in the lab. Now you want to simulate q (quantum) references for each job. I suggest and used code like the following.

   ```
   for (int ref=0; ref<q; ref++) {
       simulate this reference for this process
       calculate the next reference for this process
     }
   ```

   One effect is that after simulating the q^{th} reference you will calculate the first reference for the next quantum. Hence, you may be reading the random number file before you switch to the next process. Specifically, at the beginning of the run you have the first reference given to you for process 1, namely 111*1=111 mod S. Now you simulate q references (the first to address 111 mod S) and you calculate the next q addresses. These calculations use one or two random numbers for each reference (two if a random reference occurs). So you read the random number file once or twice for the last reference (q+1), even though you will be context switching before simulating this reference. Although you do not have to use my code above, you do need to use the random numbers the same way I do.
4. When calculating the next word to reference, you have four cases with probability A, B, C, and 1-A-B-C. Read a random number from the file and divide it by RAND_MAX+1 = 2147483648 (RAND_MAX is the largest value returned by the random number generator I used to produce the file; it happens to equal Integer.MAX_VALUE). This gives a quotient y satisfying 0≤y<1. If the random number was called r (an integer) the statement you want in Java is (note the 1d)

   double y = r / (Integer.MAX_VALUE + 1d)
   The C/C++ equivalent is (note the 1.0)
   double y = r / (MAXINT + 1.0)

   If y<A, do case 1 (it will occur with probability A),
   else if y<A+B, do case 2, (it will occur with probability B),
   else if y<A+B+C, do case 3 (it will occur with probability C).
   else /* y>=A+B+C */, do case 4 (it will occur with probability 1-A-B-C.)

Input and outputs

I fancied up my program to produce debugging information optionally with an extra last argument. You are **NOT** being asked to do that.

Note the column for input. It shows the line used to run the program to get the normal output. For example, if you are writing in java and your program is called Paging, you would execute input 1 by typing

```
java Paging 10 10 20 1 10 lru 0
```

If you are writing in C or C++ the line would be same without the initial java. To get the corresponding debugging or "show random" output, the final 0 is changed. But remember you do **NOT** have to support debugging or "show random" output. Of course, you might find it a wonderful debugging aid.

The outputs for corresponding inputs are on NYU Classes.

Number	**Input**
1	10 10 20 1 10 lru 0
2	10 10 10 1 100 lru 0
3	10 10 10 2 10 lru 0
4	20 10 10 2 10 lru 0
5	20 10 10 2 10 random 0
6	20 10 10 2 10 lifo 0
7	20 10 10 3 10 lru 0
8	20 10 10 3 10 lifo 0
9	20 10 10 4 10 lru 0
10	20 10 10 4 10 random 0
11	90 10 40 4 100 lru 0
12	40 10 90 1 100 lru 0
13	40 10 90 1 100 lifo 0
14	800 40 400 4 5000 lru 0
15	10 5 30 4 3 random 0
16	1000 40 400 4 5000 lifo 0