UCF "Practice" Local Contest — Aug 25, 2012

The Clock Algorithm

filename: `pagealloc`

In operating systems design, there is a technique called *virtual memory*, which enables the computer to run a program whose required memory (or logical memory) is larger than the available physical memory of the computer. The memory (for computer and program) are divided into pages, each consisting of a block of fixed size. When a program requests to access a page that is not yet loaded into memory, a *page fault* occurs. *Thus, a page fault will occur the first time each page is accessed.* In addition, if the program needs to access more pages than the available memory can provide, one of the pages that were loaded previously will have to be swapped out (written onto disk) in order to provide free memory space for the requested page. The algorithm that decides which page to swap out is called the page replacement strategy. One of the widely-known replacement algorithm is called the least-recently-used (LRU) strategy.

The Problem:

In this problem, we'll explore a variation of LRU known as the clock algorithm. This algorithm works as follows. Initially, all n page cells in memory are free; you can think of them as an array with indices $1 \ldots n$. Each element of the array contains the amount of memory for one page, and a "*flag*". The clock algorithm also keeps a "*hand pointer*", which initially points to cell 1.

When the program needs to access a page, first it checks to see if the page is currently loaded in memory; if so, it simply accesses that page (no page fault) and sets the page's flag to *new*.

If the page it needs is not currently loaded in memory, then it loads the page in the next available free cell (cell with the lowest index) if there is one available and sets the page's flag to *new*. If there is no free cell, then it checks the cell pointed by *hand pointer*. If the page pointed by hand is marked as *old*, then it will be swapped out for the new request (i.e., the page is loaded and marked as *new*) and the pointer advances one position. If the page pointed by hand is marked as *new* (i.e., the page is not *old*), the algorithm then marks that cell/page as *old* and the pointer advances one position. Advancing the pointer simply means the pointer will point to the next memory cell (with 1 higher index), and wraps back to the first cell if advancing from the n^{th} cell, similar to the hands of a clock. Eventually, some page pointed by the hand will be seen as *old* and is swapped out, giving a free cell for the requested page.

Note that when a page is referenced or loaded, its flag is set to *new*.

This strategy gives each cell a second chance, being set to *old* before getting swapped out. If a page is referenced while the cell/page is *old*, it will be marked as *new* again, indicating that it is recently used. Thus a page will only be swapped out if the hand pointer sees it twice (first time marks it as *old*, and second time swaps it out) without the page ever being used during that period.

Your task is to implement the clock algorithm.

The Input:

There will be multiple test cases. The first line of each test case contains two integers, n ($1 \le n \le 50$), the number of page cells in available memory, and r ($1 \le r \le 50$), the number of page requests made by the program. The next line contains r integers r_i ($1 \le r_i \le 50$), separated by spaces. These are pages that are accessed by the program (the pages are listed in the order of being accessed by the program).

The last test case will be followed by a line which contains "0 0", i.e., end-of-data is indicated by $n = 0$ and $r = 0$.

The Output:

At the beginning of each test case, output "`Program` p", where p is the test case number (starting from 1). For each request, output: "`Page` r_i `loaded into cell` c." if the request r_i is not in memory, and is loaded into cell c. If the request r_i is already in memory, output: "`Access page` r_i `in cell` c." where c is the cell that page r_i is located in memory. At the end of each test case, output: "`There are a total of` k `page faults.`" where k is the total number of page faults (loads) experienced during the execution of the program.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Notes:

- The size of each page is not important, as the input is already given in units of pages.
- The logical memory requirement for a program is not specified, but you may assume that all programs have a size of 50 pages (as bounded by page request, r_i).

Sample Input:

```
3 12
3 2 1 5 3 2 4 3 2 1 5 4
5 16
1 3 5 7 9 9 7 5 1 8 3 5 2 3 12 18
8 1
1
0 0
```

(Sample Output on the next page)

9

Sample Output:

```
Program 1
Page 3 loaded into cell 1.
Page 2 loaded into cell 2.
Page 1 loaded into cell 3.
Page 5 loaded into cell 1.
Page 3 loaded into cell 2.
Page 2 loaded into cell 3.
Page 4 loaded into cell 1.
Access page 3 in cell 2.
Access page 2 in cell 3.
Page 1 loaded into cell 2.
Page 5 loaded into cell 3.
Access page 4 in cell 1.
There are a total of 9 page faults.

Program 2
Page 1 loaded into cell 1.
Page 3 loaded into cell 2.
Page 5 loaded into cell 3.
Page 7 loaded into cell 4.
Page 9 loaded into cell 5.
Access page 9 in cell 5.
Access page 7 in cell 4.
Access page 5 in cell 3.
Access page 1 in cell 1.
Page 8 loaded into cell 1.
Access page 3 in cell 2.
Access page 5 in cell 3.
Page 2 loaded into cell 4.
Access page 3 in cell 2.
Page 12 loaded into cell 5.
Page 18 loaded into cell 3.
There are a total of 9 page faults.

Program 3
Page 1 loaded into cell 1.
There are a total of 1 page faults.
```