The goal of this lab is to do resource allocation using both an optimistic resource manager and the banker's algorithm of Dijkstra. The optimistic resource manager is simple: Satisfy a request if possible, if not make the task wait; when a release occurs, try to satisfy pending requests in a FIFO manner.

Your program is to read all the input for a run, then perform the simulation, and then produce output. Thirteen required runs, including expected output, are available on the web. We will test your program on additional runs as well.

The input begins with two values T, the number of tasks, and R, the number of resource types, followed by R additional values, the number of units present of each resource type. (If you set ''arbitrary limits'' on say T or R, you must, document this in your readme, check that the input satisfies the limits, print an error if it does not, and set the limits high enough so that the required inputs all pass.) Then come multiple inputs, each representing the next activity of a specific task. The possible activities are initiate, request, compute, release, and terminate. Time is measured in fixed units called cycles and, for simplicity, no fractional cycles are used. The manager can process one activity (initiate, request, or release) for each task in one cycle. However, the terminate activity does **NOT** require a cycle.

The initiate activity (which much precede all others for that task) is written

```
initiate   task-number resource-type   initial-claim
```

(The optimistic manager ignores the claim.) If there are R resource types, there are R initiate activities for each task.

The request and release activities are written

```
request    task-number resource-type   number-requested
release    task-number resource-type   number-released
```

The compute activity is written

compute task-number number-of-cycles

This activity means that for the next several cycles the process is computing and will make no requests or releases. It retains its current resources during the computation.

Finally the terminate operation, which does **NOT** require a cycle is written

terminate task-number

Commenting Your Program

You must include enough high-level comments in your program so that a reader (e.g., the grader) who is expert in the programming language you use and knowledgeable about resource management can understand the basic operation of your program. For example, you should make clear when you are checking for safety, when you are checking for deadlock, when you are releasing a previously blocked task, etc. You must also supply comments for your major data structures.

Input

As in lab 1 (linker) we use free form input; so a single activity may span several lines and (parts of) several activities may be on one line. Unlike lab 1, lab 3 permits you to read all the data before processing. The data for each task occurs in order, but the tasks themselves may be interspersed. Input set 1, on the web and reprinted below, has all of task one followed by all of task two. Input set 8, on the web, represents the same run but the lines for tasks one and two are interleaved.

Output

At the end of the run, print, for each task, the time taken, the waiting time, and the percentage of time spent waiting. Also print the total time for all tasks, the total waiting time, and the overall percentage of time spent waiting.

Error Checks

When implementing banker's algorithm, if a task's initial claim exceeds the resources present or if, during execution, a task's requests exceed its claims, print a message, abort the task, and release all its resources. This does not apply to the optimistic allocator since it does not have the concept of initial claim.

Deadlock

Deadlock cannot occur for banker's algorithm, but can for the optimistic resource manager. If deadlock is detected, print a message and abort the lowest numbered deadlocked task after releasing all its resources. If deadlock remains, print another message and abort the next lowest numbered deadlocked task, etc.

We learned sophisticated algorithms for detecting deadlock. You are **NOT** expected to implemented one of these. Instead you should use the trivial algorithm that detects a deadlock when all non-terminated tasks have outstanding requests that the manager cannot satisfy. Note that, if a deadlock actually occurs during cycle n, you may not detect it until much later since there may be non-deadlocked processes running. If you detect the deadlock at cycle k, you abort the task(s) at cycle k and hence its/their resources become available at cycle k+1. This trivial deadlock detection algorithm is not used in practice.

Important Note

Items returned at time n are not available until time n+1, For example, if task 1 returns 10 units during cycle 6-7 and task 3 requests 10 units during that same cycle, the 10 units returned by task 1 are **not** available for the optimistic manager to give out and, when the banker makes its safety check, these 10 units are **not** considered available. They become available at time 7 for use in cycle 7-8.

The first data set

The input for the first run is.

```
2 1 4
initiate  1 1 4
request   1 1 1
release   1 1 1
terminate 1
initiate  2 1 4
request   2 1 1
release   2 1 1
terminate 2
```

The first line asserts that this run has 2 tasks and 1 resource type with 4 units.

The next line indicates that the run begins (at cycle 0-1, the cycle starting at 0 and ending at 1) with task 1 claiming (all) 4 units of resource 1. Further down on line 6 we see that task 2 also claims 4 units of resource 1 during cycle 0-1.

From lines 3 and 7 we learn that each task requests a unit during cycle 1-2 and returns that unit during the next cycle after the request is granted. For the optimistic manager, the request is granted at 2 (the end of cycle 1-2) and the resource is returned during 2-3.

Input 1 does not use the compute activity. For the optimistic manager each task terminates at time 3.