2006 South Central USA Regional Programming Contest

Core Wars

Introduction:

Core Wars is a game in which two opposing warrior programs attempt to destroy each other in the memory of a virtual machine. They do this by overwriting each other's instructions, and the first program to execute an illegal instruction is declared the loser. Each program is written in an assembly-like language called **Redcode**, and the virtual machine which executes the two programs is known as the **Memory Array Redcode Simulator** (MARS). Your goal is to write a MARS that will read in two Redcode programs, simulate them, and print out which program was the winner.

MARS simulates a somewhat unusual environment compared to other virtual machines and processor architectures. The following list describes these differences in detail:

1. MARS simulates a machine with 8000 memory locations and each location stores exactly one Redcode instruction. In fact, a memory location can only store instructions and cannot directly store any data. However, each instruction includes two numeric operands, and these in turn can be manipulated by other instructions for data storage. This also makes self-modifying code possible.
2. The memory locations are arranged as a continuous array with the first location having address 0 and the last having 7999. All address calculations are performed using modulo 8000 arithmetic. Put in another way, memory addresses wrap around so that addresses 8000, 8001, and 8002 refer to same memory locations respectively as addresses 0, 1, and 2. This also works for negative numbers. For example, -7481, -15481, 519, and 8519 all refer to the same memory location.
3. All arithmetic and comparison operations are performed modulo 8000. Additions must normalize their final result to be in the range of 0 to 7999 (inclusive) before writing that result into memory. This also implies that -124 is considered to be *greater* than 511 since after normalization, -124 becomes 7876, and 7876 is greater than 511.
4. The simulator maintains two separate instruction pointers (IPs) that store the address of the next instruction to be executed by the warrior programs. After loading both programs into memory, these IPs are initialized to the first instruction of each program. As the programs run, the IP is incremented by one (modulo 8000) after each instruction is executed. If a jump/skip instruction is executed, then the IP is instead loaded with the destination address of the jump/skip and execution continues from this new address.
5. The simulator "time slices" between warriors by executing one instruction at a time, and alternating between programs after each instruction. For example, if the two programs were loaded at addresses 2492 and 6140, the first six instructions would be executed in this order (assuming no jump/skip instruction were executed): 2492, 6140, 2493, 6141, 2494, 6142.

Every instruction in MARS consists of an opcode, written as a three letter mnemonic, and two operands called the A and B fields. Each operand is a number in the range 0-7999 (inclusive) and each can use one of

three addressing modes: immediate, direct, and indirect. These modes are explained in more detail below:

- Immediate operands are written with a "#" sign in front, as in `#1234`. An immediate operand specifies a literal value for the instruction to operate on. For example, the first operand (i.e. the A field) of an `ADD` instruction (which performs integer addition) can be an immediate. In that case, the literal value specified by the first operand provides one of the numbers being added.
- Direct operands identify the memory locations which an instruction is to access. They are written with a "$" sign in front, as in `$1234`. One example would be `ADD #5 $3`. A direct operand is actually an offset relative to the current IP address. For example, if the `ADD #5 $3` instruction were stored in memory location 4357, it would actually be adding together a literal number five with a second value stored in the B field of location 4360 (4357 + 3). However, if that same instruction were stored at location 132 it would be adding five to a value in the B field of location 135 (132 + 3).
- Indirect operands are analogous to how pointers in some programming languages work. Indirect operands are written with a "@" sign in front of the number, as in `ADD #5 @3`. Like before, the indirect operand is an offset relative to the current IP address, and therefore identifies a particular memory location. However, the value stored in the B field of this memory location is then used as an offset relative to *that* location to identify a *second* location. It is the B field of this second location which will actually be operated on by the instruction itself. For example, if location 4357 contained `ADD @1 @3`, location 4358 contained 11 in its B field, and location 4360 contained 7996 in its B field, then this instruction would actually be adding the values stored in locations 4369 (4358 + 11) and 4356 (4360 + 7996 modulo 8000).

The list below explains what each instruction does based on its opcode. Although not all instructions use both of their operands, these must still be specified since other instructions might use these operands for data storage. **Some instructions update the B field of another instruction; this only affects the numerical value of the field, but does *not* change its addressing mode.**

`DAT` This instruction has two purposes. First, it can be used as a generic placeholder for arbitrary data. Second, attempting to execute this instruction terminates the simulation and the program which tried to execute it loses the match. This is the only way that a program can terminate, therefore each warrior attempts to overwrite the other one's program with `DAT` instructions. Both A and B operands must be immediate.

`MOV` If the A operand is immediate, the value of this operand is copied into the B field of the instruction specified by `MOV`'s B operand. If neither operand is immediate, the entire instruction (including all field values and addressing modes) at location A is copied to location B. The B operand cannot be immediate.

`ADD` If the A operand is immediate, its value is added to the value of the B field of the instruction specified by `ADD`'s B operand, and the final result is stored into the B field of that same instruction. If neither operand is immediate, then they both specify the locations of two instructions in memory. In this case, the A and B fields of one instruction are respectively added to the A and B fields of the second instruction, and both results are respectively written to the A and B fields of the instruction specified by the `ADD`'s B operand. The B operand cannot be immediate.

`JMP` Jump to the address specified by the A operand. In other words, the instruction pointer is loaded with a new address (instead of being incremented), and the next instruction executed after the `JMP` will be from the memory location specified by A. The A operand cannot be immediate. The B

operand must be immediate, but is not used by this instruction.

`JMZ` If the B field of the instruction specified by `JMZ`'s B operand is zero, then jump to the address specified by the A operand. Neither the A nor B operand can be immediate.

`SLT` If A is an immediate operand, its value is compared with the value in the B field of the instruction specified by `SLT`'s B operand. If A is not immediate, the B fields of the two instructions specified by the operands are compared instead. If the first value (i.e the one specified by A) is less than the second value, then the next instruction is skipped. The B operand cannot be immediate.

`CMP` The entire contents of memory locations specified by A and B are checked for equality. If the two locations are equal, then the next instruction is skipped. Memory locations are considered equal to another if they both have the same opcodes **and** they have the same values and addressing modes in their respective operand fields. The A or B operands cannot be immediate.

Input:

The input begins with a line containing a single integer *n* indicating the number of independant simulations to run. For each simulation the input will contain a pair of programs, designated as warrior number one and warrior number two. Each warrior program is specified using the following format:

One line with integer *m* (1 <= *m* <= 8000) indicating the number of instructions to load for this warrior. A second line containing an integer *a* (0 <= *a* <= 7999) gives the address at which to start loading the warrior's code. These two lines are then followed by *m* additional lines containing the warrior's instructions, with one instruction per line. If the warrior is loaded at the end of memory, the address will wrap around and the instructions continue loading from the beginning of memory.

The address ranges occupied by the two programs will not overlap. **All other memory locations which were not loaded with warrior code must be initialized to `DAT #0 #0`.** Execution always begins with warrior number one (i.e. the warrior read in first from the input file).

Output:

Each simulation continues running until either warrior executes a `DAT` instruction or until a total of 32000 instructions (counting both warriors) are executed. If one warrior program executes a `DAT`, the other is declared the winner; display "`Program #x is the winner.`", where *x* is either `1` or `2` and represents the number of the winning warrior. If neither program executes a `DAT` after the maximum instruction count is reached, then the programs are tied; display "`Programs are tied.`"

Sample Input:

```
2
3
185
ADD #4 $2
JMP $-1 #0
DAT #0 #-3
5
100
JMP $2 #0
```

```
DAT #0 #-1
ADD #5 $-1
MOV $-2 @-2
JMP $-2 #0
1
5524
MOV $0 $1
5
539
JMP $2 #0
DAT #0 #-1
ADD #5 $-1
MOV $-2 @-2
JMP $-2 #0
```

Sample Output:

```
Program #2 is the winner.
Programs are tied.
```