I. Arctic Polar Explorer

The latest expedition to the North Pole has encountered a problem. The autonomous Arctic Polar Explorer robot (APE) needs to reach a higher cliff, but it was never designed for that task. The scientists at mission control needed to improvise and they came up with an ingenious solution: the robot can climb up the cliff by building its own staircase.

The APE's mission takes place in the Arctic Circle, where the landscape is littered with rocks, icebergs and the occasional lost penguin. The APE has collected a bunch of rocks of different sizes, and its goal is now to move the rocks in such a way that they become ordered by increasing size. The size of a rock is proportional to its weight. The rocks lie in a straight line, with one rock per meter. The APE is currently standing next to the first rock, while the cliff is just after the last rock. The robot can move left and right, parallel to the line of rocks. A possible situation is illustrated in figure 3.

The only way of manipulating the outside world is through two arms with grippers that can be used to pick up rocks. Unfortunately, there is no way of measuring the weight of rocks. After another lengthy brainstorm session the scientists at mission control have determined that the weight of the rocks in the grippers can be compared by measuring the tilt of the robot. If the robot tilts to the left then the left gripper contains a heavier rock, and if it tilts to the right then the right gripper contains a heavier rock.

Finally the APE has a very limited amount of memory, making writing programs for it rather difficult. All software for the robot is written in a specialized language, APECODE.

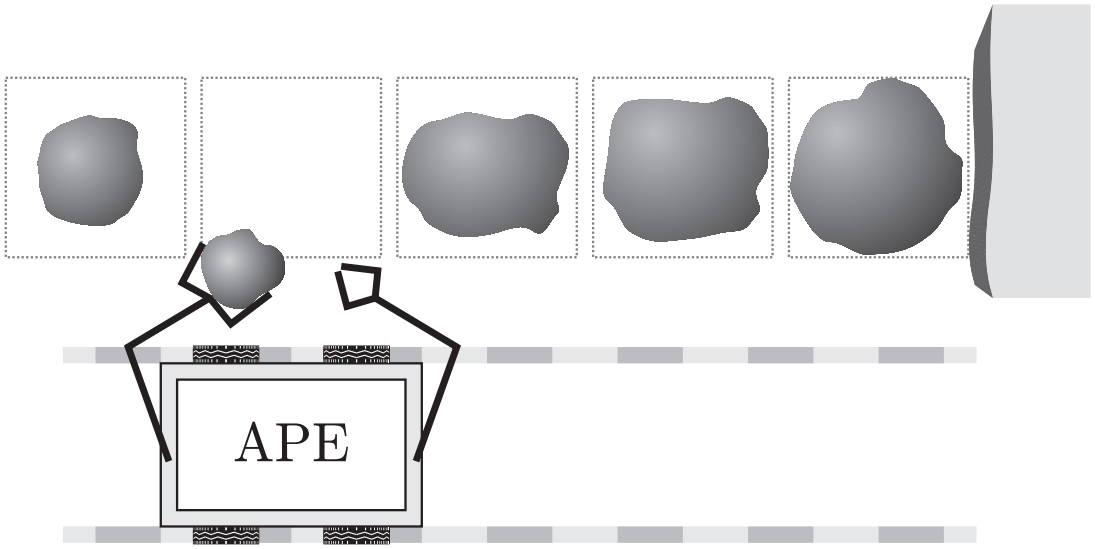


Figure 3: A sketch of a possible situation for the Arctic Polar Explorer. Here it holds a rock in the left gripper, while the right gripper is empty. There are five rocks, and the cliff is to the right.

The Language

Robots programmed in APECODE operate as a stack/state machine. The robot sequentially executes the statements in the current state, and when it reaches the end of a state it starts again from the beginning. APECODE has the following statements:

- With the "`call`" statement the robot jumps into a new state.
- With the "`return`" statement the robot returns to the calling state. (To the point just after the corresponding "`call`" statement.)
- The "`then`" statement executes the following body if and only if the last "`call`" returned "`true`", otherwise the "`else`" branch is executed (if there is any).

Programs must be given as a ⟨PROGRAM⟩ from the following grammar:

$$
\begin{aligned}
\langle \text{COMMENT} \rangle &::= \text{“//”}\ \langle \text{CHARACTER} \rangle^*\ \langle \text{NEWLINE} \rangle \\
&\mid \text{“/*”}\ \langle \text{CHARACTER} \rangle^*\ \text{“*/”} \\
\langle \text{LETTER} \rangle &::= \text{“_”} \mid \text{“a”} \mid \cdots \mid \text{“z”} \mid \text{“A”} \mid \cdots \mid \text{“Z”} \\
\langle \text{LETTER_OR_DIGIT} \rangle &::= \langle \text{LETTER} \rangle \mid \text{“0”} \mid \cdots \mid \text{“9”} \\
\langle \text{NAME} \rangle &::= \langle \text{LETTER} \rangle \langle \text{LETTER_OR_DIGIT} \rangle^* \\
\langle \text{PROGRAM} \rangle &::= \langle \text{STATE} \rangle^* \\
\langle \text{STATE} \rangle &::= \text{“state”}\ \langle \text{NAME} \rangle\ \langle \text{STMTS} \rangle \\
\langle \text{STMTS} \rangle &::= \text{“\{”}\ \langle \text{STMT} \rangle^*\ \text{“\}”} \\
\langle \text{STMT} \rangle &::= \text{“call”}\ \langle \text{NAME} \rangle\ \text{“;”} \\
&\mid \text{“return”}\ \langle \text{BOOL} \rangle\ \text{“;”} \\
&\mid \text{“then”}\ \langle \text{STMTS} \rangle \\
&\mid \text{“then”}\ \langle \text{STMTS} \rangle\ \text{“else”}\ \langle \text{STMTS} \rangle \\
\langle \text{BOOL} \rangle &::= \text{“true”} \mid \text{“false”}
\end{aligned}
$$

Comments function as in C: ⟨CHARACTER⟩ is any character, while ⟨NEWLINE⟩ is the end of a line, comments do not nest. Just as in C, whitespace is ignored and names are case sensitive.

The Library

There is a built-in library of states for manipulating the robot. After completing the action, control is returned to the calling state.

- The states `move_left` and `move_right` move the robot one meter left or right respectively.
- The states `pick_up_left` and `pick_up_right` pick up a rock in one of the robot's grippers. It is a fatal error if that gripper already holds something.
- The states `put_down_left` and `put_down_right` place the contents of one of the robot's grippers on the ground. It is a fatal error if the ground around the robot is not clear of rocks.
- The states `if_empty_left` and `if_empty_right` return `true` if and only if the left/right gripper does not hold anything.

- The states `if_tilt_left` and `if_tilt_right` return `true` if and only if the robot is currently tilted to the left/right.
- The state `remember` remembers the return value of the last call, the state `recall` returns the last remembered value.
- The state `trace` can be used for debugging, it outputs a schematic representation of the current situation.

The Compiler

An APECODE compiler is provided for you. This command line program can be invoked with the command

`apecc <program_name>.ape`

This produces an executable named `<program_name>` that *simulates* the Arctic Polar Explorer's behavior.

The Simulator

The simulator produced by the compiler reads from the standard input and writes to the standard output.

On the first line of the simulator's input is a positive integer, the number of test cases. Then for each test case:

- The simulator reads a line containing a single positive integer $n < 5000$, the number of rocks, followed by a line with n positive integers, the weights of each rock in kilograms.
- The robot is placed next to the first rock with both grippers empty.
- The program starts in the state `main` and is run until that state returns.
- The simulator outputs a line containing the weights of the rocks after executing the APECODE program. If a certain place contains no rock, then the character '`-`' is printed.

Example

Input	Output
1 9 7 1 6 3 4 9 2 5 8	1 2 3 4 5 6 7 8 9

Note: You must write a program in APECODE, other programming languages do not work on the Arctic Polar Explorer.