Ants on a plane!

In this étude you are going to be simulating (some of) the behaviour of creatures related to Langton's ant. Imagine an (in principle) infinite plane of square tiles, with a lonely ant initially standing at $(0, 0)$. The ant will take a certain sequence of steps of unit length in one of the four compass directions determined by various rules (*i.e.*, its "DNA"). These rules specify the direction of the next step based on the previous step, and the state of the ant's current position—they may also specify a change in that state.

For instance, Langton's original Ant lives on a plane with two possible states (black and white) and with the rules "right on white, left on black, flip the colour".

The problem you have to solve here is, given an ant's DNA, determine its location after a certain number of steps (starting on a plane whose points are all of some given state).

Task

In this task, input will come from `stdin` and will consist of a sequence of "scenarios" separated from one another by empty lines. (An empty line contains no characters, not even spaces or tabs.) There may also be some lines beginning with the hash character, #. These are comments and should be ignored entirely. (A comment does *not* count as an empty line.) The output for each scenario, specified below, should be written to `stdout`.

An individual scenario consists of the DNA of an ant, followed by a positive integer up to 10^9 (the number of steps you are to follow the ant for). For Langton's ant, the DNA is represented as follows:

```
w ESWN bbbb
b WNES wwww
```

Each line of DNA consists of three parts separated by single spaces:

- A single character representing a state.
- A sequence of four compass directions representing the direction of the next step from a point in this state, after arriving with a North, East, South, and West step respectively (so if a Langton's ant arrives

at a white point from the south, *i.e.*, using a North step, its next step is East—a right turn).

- A sequence of four characters representing the state of the point after the ant leaves (again based on the incoming direction).

At each step, the ant:

- determines the state of the tile it is standing on;
- finds the rule in its DNA for that state;
- uses its direction to select the new tile state;
- uses its direction to determine its new direction; and
- moves one step in that new direction.

Initially, all points of the plane are presumed to be in the state coded by the first line of the DNA (*e.g.*, `w` above), and the ant is facing North on the point $(0, 0)$ (*i.e.*, its first step is to be determined as if it had arrived at $(0, 0)$ from the south). The "mathematical" coordinate system is used—a North step increases the second (or y) coordinate by 1, and an East step increases the first (or x) coordinate by 1.

It is *not* to be assumed that there will always be just two states, or that they are coded by letters. State names can be *any* character other than CR, LF or "#". For instance this is a perfectly valid DNA representation:

```
. EEEE xx x
x SEWN  .9x
  WEEE 9999
9 NWES .x 9
```

You *may* assume that the DNA is always complete (*i.e.*, you will not be given next states in the third part that are not represented in the list).

The output for a scenario is to echo the input (except comments), followed by a line consisting of a # character, followed by a space, then the x-coordinate of the ant's final position, another space, and the y-coordinate of the ant's final position. The output of each distinct scenario should be separated from the next by a single empty line. An empty line contains *no* characters. The last line of the output *must not* be followed by an empty line unless that line was echoed from the input.

(Pair 2)

Notes

What is the relevance of this to Computer Science?

First, Langton's Ant is closely related to Cellular Automata, which have applications from modelling the growth of organisms to explorations of the possibility that physics might work a similar way at the deepest levels. They are a useful tool for modelling other things, such as forest fires and epidemics.

Second, there are several cellular automata, including Langton's Ant, that have been shown to be computation universal.

Third, Langton's Ant in particular is an extremely simple set of rules with behaviour that nobody can predict by looking at them. Nor, looking at the behaviour of the ant as it bumbles around at the beginning, would you ever expect that it suddenly switches into highly regular "highway-building" behaviour; yet the two visibly different kinds of activity are governed by the *same* rules. It's a lesson that complexity can be encoded in the *environment* (data structures) rather than an organism's neural system (executable code).

Fourth, that gives us an idea of the difficulty of debugging arbitrary programs. If we have such difficulty prdicting the behaviour of a model *this* simple, what hope do we have of predicting what some strange piece of code will do? If we didn't already have a working implementation of Langton's Ant, how would you debug yours? There are partial answers to these questions, but they are worth pondering.

Fifth, the central loop of this étude is essentially the same as simulating a Deterministic Finite State Automation as used in lexical analysis or a Turing Machine—an ant basically is a Turing machine with a 2D "tape"—of importance in theoretical Computer Science.

More concretely, this is an easy warm-up exercise. An adequate program can be written in 23 lines of AWK. You have to parse some very simple line-oriented input, taking great care to ensure that you have understood the specification. For example, a direction must be one of the four capital letters NEWS (not a lower case letter), but nobody said that a state name could not be a space. You have to produce exactly the output required, no more and no less.

You do not need any *complicated* data structures, but you do need to think *carefully* about data structure design. If an ant moves for 200,000,000 steps, how big can x and y get?

A surprising number of students have trouble figuring out how to decide which pair of characters to use from a rule, and think it has something to do with searching for the current direction as a character in a string. To see that this cannot be the case, consider an ant that travels North to a tile in state "x".

Rule	Problem
"`x EEEE yyyy`"	there is no N
"`x EEEN yyyy`"	there is nothing after N
"`x NENW yyyy`"	"after N" is ambiguous
"`x NENE abcd`"	which new state to use?

So that interpretation cannot possibly work. Hint: the directions were listed in the order N E S W for a reason.

To "echo the input" means to copy it *exactly*. For example, if a number line is 007 (which is legal), it is simply *wrong* to print it as 7; the output must be 007. Storing lines in memory and regenerating them later is arguably not echoing either; in conventional usage "echoing" is done immediately after reading and perhaps classifying, but before any processing.

Testing

You want to find your mistakes before anyone else does. That means you want to test your code. It often helps to think about how you will test your code before you write it. So how can you test this program?

One good way to test a program is to feed it inputs where you already know what the outputs should be. Here is one "ant":

```
x NNNN xxxx
```

If the number of steps is s, what should the output be?

Here is another "ant":

```
x ESWN xxxx
```

If the number of steps is $s = 4m + k$ with $0 \leq k < 4$, what should the output be?

What tests of your own can you devise?

Would you be willing to share tests with other people?