Problem A
Adventurer Dabi
Time Limit: 3 Seconds

Dabi, the galaxy-famous adventurer, has entered a long-forgotten underground city said to hide an ancient treasure. Within its pitch-black tunnels, she must find a key and reach the treasure chest before the city collapses.

The entire city can be represented as an $h \times w$ grid ($3 \leq h, w \leq 16$). Each cell of this grid is one of the following types:

- **Empty cell**, which is a traversable passage.
- **Wall cell**, which is a solid block that cannot be entered.
- **Teleport cell**, which is an ancient device that instantly connects two distant places.

The following facts are known about this underground city:

- Every cell on the grid's boundary (i.e., topmost/bottommost row, or leftmost/rightmost column) is a wall.
- All wall cells are connected through four directions: north (up), south (down), east (right), and west (left)—that is, they form a single 4-directional connected component.
- All empty cells are connected, forming a single 4-directional connected component surrounded by walls.
- For every teleport cell, all **eight** neighboring cells are empty cells.
- Every teleport cell belongs to exactly one teleport pair. If Dabi steps into a teleport cell from any direction, she is immediately transported to its paired cell and then moves **one additional step** in the same direction she entered the teleport cell. This process never triggers another teleportation.
- Each teleport pair is labeled with one of the uppercase letters `A`, `B`, `C`, `D`, `E`, and `F`. Therefore, there are at most $6 \times 2 = 12$ teleport cells in total.
- There is exactly one **key** and one **treasure chest**, each placed on distinct empty cells, but neither is at Dabi's initial position.

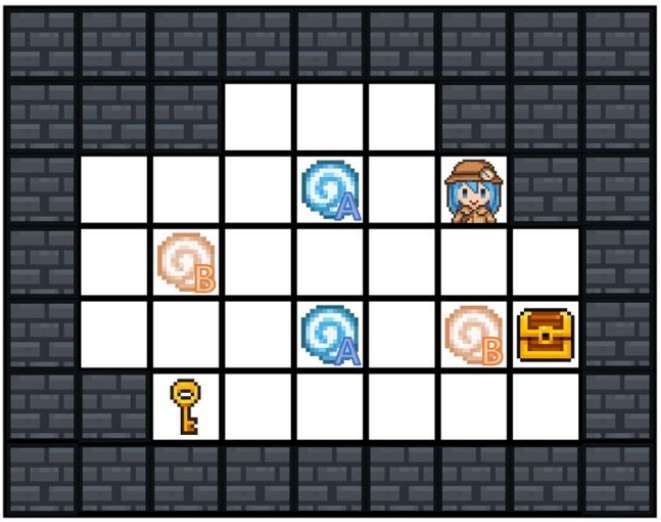


Figure 1. An example configuration of the underground city ($h = 7;\ w = 9$) with two teleport pairs `A` and `B`.

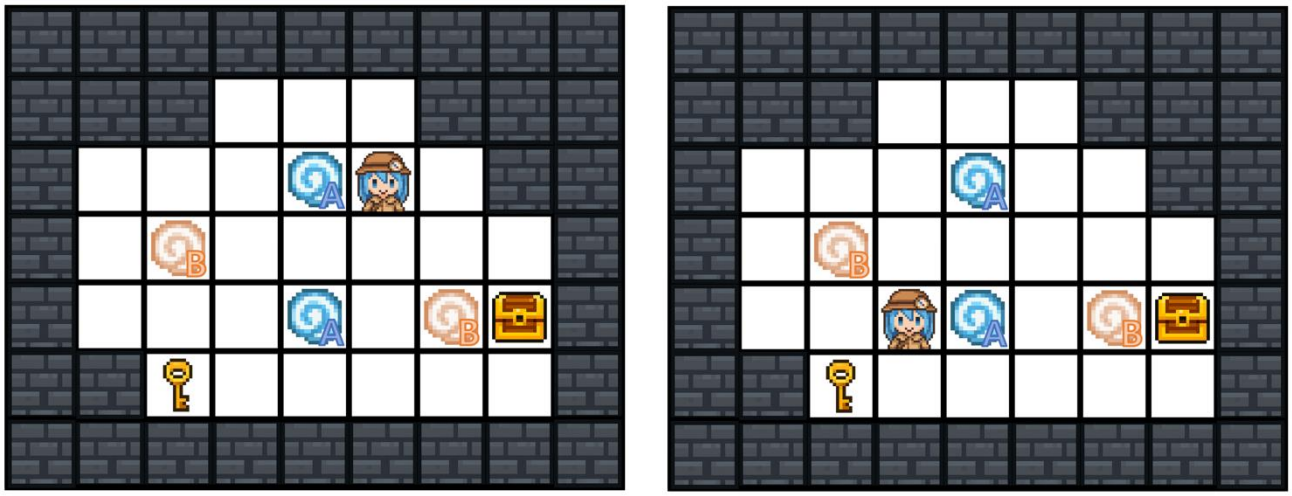
Dabi can perform the following two actions:

- Move to one of the **four** adjacent cells that is **not** a wall. Teleportation and the subsequent extra step are considered part of 'entering a teleport cell' and do not count as separate actions.
- Pick up the key if she is on the cell with the key.

Dabi always stands on an empty cell before and after each action.

Figure 1 illustrates an example of the underground city where $h = 7$ and $w = 9$. Teleport pair `A` connects the two cells marked with the letter `A`, and teleport pair `B` connects the two cells marked with the letter `B`. Note that Figure 1 corresponds to the city described in the attached `sample.in` file.

As shown in Figure 2, the teleportation process works as follows. In Figure 2(a), Dabi is standing on the cell immediately east of teleport `A`. When she moves west, she steps into the teleport cell marked `A`, is instantly transported to its paired cell, and then moves one additional step to the west, resulting in the configuration shown in Figure 2(b).



(a) **(b)**

Figure 2. Illustration of the teleportation process. **(a)** Dabi stands on the cell immediately east of teleport `A`. **(b)** After moving west, she steps into teleport cell `A`, is transported to its paired cell, and then moves one additional step to the west.

When Dabi picks up the key, the city begins to collapse. From that moment, she must reach the treasure cell in the **minimum** possible number of actions. Any longer route causes the attempt to fail.

Dabi carries a reliable compass and can always tell north, south, east, and west. By feeling the walls around her, she can sense for each of the **four** directions whether the adjacent cell is a **wall** or not. However, she does not initially know her coordinates or the overall layout of the city. At any moment, she can also sense whether her current cell contains the **key** or the **treasure**. She may stand on the key cell **without** immediately picking it up.

Your task is to guide Dabi through the city to obtain the key and then reach the treasure chest, obeying all movement rules. Write a program to accomplish this by interacting with the interactor through a sequence of commands.

Interaction

This is an **interactive** problem. Your submitted program will interact with an **interactor** inside the grading server, which reads input from and writes output to your program. Your program must control Dabi by printing commands and reading the interactor's replies. After every line of output, you must immediately **flush** the output.

Initial Input

At the beginning of the interaction, the interactor provides information about Dabi's current cell. The information is given as a single line containing six characters without spaces:

- Each of the six characters is either '`0`' or '`1`'.
- For the first four characters, '`1`' means that the adjacent cell in that direction is a wall, and '`0`' means that the adjacent cell in that direction is **not** a wall. The order of directions is north, south, east, and west.
- The fifth character is '`1`' if the key is present on Dabi's current cell and has not yet been picked up; otherwise, it is '`0`'.
- The sixth character is '`1`' if the treasure chest is on Dabi's current cell; otherwise, it is '`0`'.
- Since the key and the treasure chest are on different cells, the fifth and sixth characters are never both '`1`'.

For example, "`100010`" means that there is a wall to the north, the other three directions are open, the unpicked key is on the current cell, and there is no treasure.

Commands

Your program may repeatedly output exactly one of the following commands until the interactor outputs "`correct`" or terminates the interaction.

1. "`N`", "`S`", "`E`", or "`W`"
    - Move one step in the chosen direction—"`N`" (north), "`S`" (south), "`E`" (east), or "`W`" (west). The target cell must **not** be a wall. If it is a teleport cell, Dabi will be teleported according to the rules described above.
    - Attempting to move into a wall causes the interactor to output "`wrong`".
2. "`K`"
    - Pick up the key on the current cell. After this action, the treasure chest opens, and from that moment Dabi must reach the treasure cell in the **minimum** possible number of actions.
    - If there is no unpicked key on the current cell, the interactor outputs "`wrong`".

Each command must be printed on its own line and immediately flushed. The total number of commands issued **must not exceed** 230,611.

Interactor Responses

After each action, the interactor responds as follows:

- The interactor outputs "`correct`" if Dabi has already picked up the key and has just reached the treasure cell using the minimum possible number of actions.
- The interactor immediately terminates the interaction if any rule is violated—for example, a malformed command, an invalid move, attempting to pick up a nonexistent or already collected key, exceeding 230,611 actions in total, or reaching the treasure through a route that does not use the minimum possible number of actions after the key has been collected.
- Otherwise, the interactor outputs a six-character string describing Dabi's new current cell, in the same format as the initial input.

Receiving "`correct`" means that your program has successfully completed the mission and should terminate gracefully. **Do not forget to flush the output** after printing each command.

The city layout is fixed throughout the interaction; the interactor is **not adaptive**.

The time and memory used by the interactor are also included in the calculation of your program's execution time and memory usage. You can assume that the maximum time used by the interactor is 1 second and the maximum amount of memory is 64 MiB.

The following shows a sample interaction when the city layout and Dabi's initial position are the same as in Figure 1.

Read (Interactor's Response)	Sample Interaction 1	Write (Your Command)
`101000`		
	`W`	
`000000`		
	`W`	
`000000`		
	`S`	
`010000`		
	`W`	
`010110`		
	`K`	
`010100`		
	`N`	
`000000`		
	`W`	
`010100`		
	`N`	
`000100`		
	`E`	
`correct`		

To flush, you need to do the following right after writing a command and a newline:

- `fflush(stdout)` in C;
- `std::cout << std::flush` in C++;
- `System.out.flush()` in Java or Kotlin;
- `sys.stdout.flush()` in Python.

A testing tool is provided to help you develop your solution, so it is not mandatory to use it. If you want to use it, you can download the attachment `testing_tool.py` from the *DOMjudge Problemset* page. You can run the testing tool to see how your program interacts for a specific city layout. **This testing tool can be used regardless of the language of your program as follows.** The same explanation is also included in the comments of `testing_tool.py`.

Usage: `python3 testing_tool.py -f <inputfile> <program>`

Use the `-f` parameter to specify the input file, e.g. `input.in`.

Format of the input file: The first line contains two integers h and w. The next h lines each contain a string of length w, describing the map of the city. Each character represents a cell type as follows:

- '`#`': A wall cell.
- '`.`': An empty cell.
- '`k`': The empty cell containing the key.
- '`t`': The empty cell containing the treasure chest.
- '`v`': The empty cell where Dabi initially stands.
- '`A`', '`B`', '`C`', '`D`', '`E`', '`F`': Teleport cells, where each letter represents one teleport pair.

Example: The city shown in Figure 1 is represented as follows.

```
7 9
#########
###...###
#...A.v##
#.B.....#
#...A.Bt#
##k.....#
#########
```

The tool is provided as-is, and you should feel free to make whatever alterations or augmentations you like to it. **Note that it is not guaranteed that a program that passes the testing tool will be accepted**. For example, the testing tool **does not verify** whether Dabi reaches the treasure chest using the minimum possible number of actions after picking up the key; instead, it only reports how many actions were taken.

If you have a C++ solution stored in a file called "`sol.cpp`", you must first compile using "`g++ sol.cpp -o sol`" and then invoke the testing tool with:

```
python3 testing_tool.py -f input.in ./sol
```

If you have a Python solution that you would run using "`pypy3 solution.py`", you can invoke the testing tool with:

```
python3 testing_tool.py -f input.in pypy3 solution.py
```

If you have a Java solution that you would run using "`java MyClass`", you can invoke the testing tool with:

```
python3 testing_tool.py -f input.in java MyClass
```

If you have a Kotlin solution that you would run using "`kotlin SolutionKt`", you can invoke the testing tool with:

```
python3 testing_tool.py -f input.in kotlin SolutionKt
```