Sweepmine.TM

Slam! The clang of that useless flap of slightly-rusted metal hanging off the bottom of the whiteboard, intended as a marker holder, but in reality simply condemned to languish in its role as a cymbal off Temu whenever the occasional unsuspecting victim's elbow makes contact, wakes you up.

It is the first Alpha evening lecture of December Camp. Having slept at 10:31 pm last night (oh no, one minute past curfew), you were too tired to stay awake. Unfortunately, it means you have to head over to labs with what minimal knowledge permeated your overly thick skull during said lecture.

What was that lecture even about? You can't remember, but you overhear your fellow Alphas talking about something. Sounds like "sweep... sweepmine"? A wave of nostalgia washes over the same thick skull we were talking about previously, reminding you about an old game you once played where you swept mines or something like that. Fortunately, this makes you lose interest in the lab problems, and motivates you to stay up until 2:04 am the next day crafting a bot that will make you the best *Sweepmine* player the world has ever seen.

—

Sweepmine is a game played on a R by C grid of cells. The rules are simple - so simple in fact that they can be described in poetry:

In *Sweepmine*, a game of lore,
Each cell hides what lies in store.
A mine may lurk, or else it may show,
The number of mines around it: left, right, up and below...
(Or any pair of the aforementioned cardinal directions, leading to eight neighbour tiles ergo)

Your move can toggle flags or see,
What numbers lie beneath, to be.
The board begins a hidden land,
Your first move safe, by game's command.

Reveal a mine, and you shall lose,
A careful step, it's best to choose.
But find a safe square, numbers tell,
How many mines around it dwell.

If zero's shown, then have no fear,
Adjacent cells will soon be clear.
This spreads, revealing more and more,
Unveiling non-zero safe tiles not seen before.

Clear all safe squares, your victory's bright,
No need for flags, you've won the fight.
In *Sweepmine*'s game, with cautious cheer,
Unveil the grid, make the fog disappear.

—

Anyways, since only dumb kids play the original version, you have always played *Sweepmine.TM* (this obviously stands for "**T**hough, I forgot to mention that each square has a *charge level.* **M**oreover, the game is played on a grid homeomorphic to the 2-torus").

Practically, as all organisms with any semblance of semi-intelligence would know, if a square (i, j) with charge c_{ij} has a mine, it would be recognised by adjacent safe cells as having c_{ij} mines. Note that this sometimes means that a cell shows 0 when there are mines adjacent to it. Because of this, when $\exists c_{ij} \neq 1$, the game **does not** perform automatic clearing of cells adjacent to zeroes.

Additionally, since we play on the fundamental domain of the 2-torus, cell (i, j) is adjacent to all cells $((i - 1 \pm 1) \bmod R + 1, (j - 1 \pm 1) \bmod C + 1)$.

—

The night is still young (it's currently 1:56 am). Time is not of the essence... but your win percentage is. Can you **maximise the average number of wins** over 1000 *Sweepmine.TM* games?

Implementation Details

In this task, **do not read from standard input, do not write to standard output**. Do not interact with any files. Do not implement a main function. Instead, begin your program by including the header file `sweepmine.h` (`#include "sweepmine.h"`) and interact with it as described below.

Functions

In each test case, your solution will play multiple games. Your implementation will be called **once per game**. You must implement `next_move`, which will be called **multiple times per game**:

```
pair<bool, pair<int, int>> next_move(int R, int C, int N,
   vector<vector<int>> grid, vector<vector<int>> charge, bool T);
```

- `R`, `C`, are the rows and columns of the sweepmine grid respectively.
- `N` is the number of mines on the grid.
- `grid` is a 1-indexed vector of vectors, storing information about the grid. In particular:
  - If `grid[x][y] = 100`, the cell at the x'th row and the y'th column has not yet been uncovered.
  - If `grid[x][y] = 101`, the cell at the x'th row and the y'th column is flagged.
  - Else, the cell at the x'th row and the y'th column has `grid[x][y]` mines adjacent to it.
- `charge` is a 1-indexed vector of vectors, such that `charge[i][j]` is c_{ij}.
- `T` is 0/FALSE if the game is played on a standard grid, and 1/TRUE otherwise (i.e. torus).
- This function must return `{t, {x, y}}`:
  - `t` is 0/FALSE to uncover, 1/TRUE to toggle a flag. Note that flags are not marked, and are for program use only - that is, **your program is not killed if it returns an incorrect flag**. Furthermore, **you do not have to flag all mines to win**.
  - `{x, y}` is the cell location (row `x`, column `y`).

Rows are numbered from 1 to R, columns are numbered from 1 to C.

If you return a square not within the boundary of the grid, runtime error, or crash the judging program somehow, your program will be judged as incorrect, and you will score 0% for the test case.

Subtasks and Constraints

For all subtasks:

- $1 \leq R \leq 16$, $1 \leq C \leq 30$
- $1 \leq N < R \times C$
- $-10 \leq c_{ij} \leq 10$, $c_{ij} \neq 0$
- Each subtask consists of one test case.
- Each test case consists of 1000 games.

Note that the only board sizes your program will be tested on are:

- *Beginner*: $R = 9$, $C = 9$, $N = 10$ ($\rho \approx 12.34\%$)
- *Intermediate*: $R = 16$, $C = 16$, $N = 40$ ($\rho \approx 15.63\%$)
- *Expert*: $R = 16$, $C = 30$, $N = 99$ ($\rho \approx 20.63\%$)

Additional constraints for each subtask are listed below:

Subtask	Points	Additional constraints
1	10	Beginner; $c_{ij} = 1$ for all i, j; $T = 0$
2	20	Intermediate; $c_{ij} = 1$ for all i, j; $T = 0$
3	30	Expert; $c_{ij} = 1$ for all i, j; $T = 0$
4	10	$c_{ij} = 1$ for all i, j
5	10	$T = 0$
6	20	No additional constraints.

Scoring

If your solution violates any of the conditions in the Implementation Details section, your score will be 0. Otherwise, let P be your percentage of wins over 1000 games. Let Q be the percentage wins of the judging system. The score function, $S(x)$ is calculated as follows:

$$S(x) = \frac{(k_1 - 1)\, x^{k_2}}{k_1} + \frac{x}{k_1}, \text{ s.t. } (k_1, k_2) = \begin{cases} (7.0, 4.4) & \text{Sub. } 1 \\ (1.5, 9.0) & \text{Sub. } 2 \\ (1.3, 0.1) & \text{Sub. } 3 \\ (4.4, 7.7) & \text{Sub. } 4 \\ (1.2, 1.4) & \text{Sub. } 5 \\ (1.2, 1.4) & \text{Sub. } 6 \end{cases}$$

Your score is determined as $S(\frac{P}{Q})$. See below a table with reference scores.

Subtask	Q	P (for 25% score)	P (for 50% score)
1	90.0%	60.8%	74.8%
2	78.0%	29.3%	56.4%
3	37.0%	3.33%	14.0%
4	74.0%	55.5%	65.4%
5	96.0%	25.7%	49.7%
6	96.0%	25.7%	49.7%

Experimentation

In order to experiment and run your program, download the provided files: `grader.cpp` and `sweepmine.h` (Surely you don't need a stub file).

You don't need a compile command either it's really not that hard.

The provided grader reads in from standard input, and outputs to standard output. As such, **writing to standard output in your implementation WILL break the system**. It reads as follows:

- Line 1: R C N T O D
- If $D = 1$:
  - Line x from 2 to $R + 1$: grid[x][1], grid[x][2], ..., grid[x][C]
  - Line x from $R + 2$ to $2R + 1$: c_{x1}, c_{x2}, ..., c_{xC}
- Else, the grader will generate its own grid and charge values. If $O = 0$, all charge values will be 1.

When $D = 1$, debug mode is turned on. In this mode, you give the grader a "grader grid"[1] and a board of charges, which will be the starting board. Note that while automatic board generation will guarantee the first cell revealed is not a mine, by providing a grid, this is no longer guaranteed. Furthermore, when $D = 1$, all board states that your program sees throughout the game are printed to the game logs.

The grader will play **one** game with these inputs (i.e. R by C grid, N mines, T being whether its a

[1]Explained later in **Reading Game Data**.

torus, etc.).

The grader will also write to an output file named `sweepmine.out`. This will contain game data. By a similar token, **do not write to this output file**. The grader will either print `"You win."` or `"You lose."` to standard output, depending on the result of the game.

Randomisation

The grader has a random device, `rng`, which is seeded off time. Feel free to edit this for debugging purposes.

Sample Grader Input

```
3 3 1 0 0 1
1 1 1
1 101 1
1 1 1
1 1 1
1 1 1
1 1 1
```

Benchmarking (Outdated)

In order to run multiple games and simulate the actual grading system, use the script `benchmarker.py` provided. This script depends on the Python libraries `subprocess` and `time`.

The benchmarking script reads from the grader's standard output. It reads from its own standard input. To use it, first ensure that you have ran the compilation command:

```
g++ grader.cpp sweepmine.cpp -o sweepmine
```

Then, run with:

```
python3 benchmarker.py
```

The script takes input on one line of the form $R\ C\ N\ T\ O\ X$, where X is the number of games to play, and O determines the charge board generation as per described in **Experimentation**. It will play X games, printing the result of each one, before printing the percentage of wins out of those X games.

Sample Benchmarker Input (Outdated)

```
9 9 10 0 0 1000
```

Reading Game Data

If you decide to use debug mode, it may be helpful to be able to understand the file `sweepmine.out`, and the way the grader operates.

The grader holds an internal board, which will henceforth be known as the **"grader grid"**. The grader grid is formatted slightly differently to the grid your program takes in as input, notably:

- A value of 101 denotes a mine.
- All other values denote the cell's value, as per usual. 100 does not appear.

The file `sweepmine.out` begins with a description of the game. The first line contains $R\ C\ N\ T\ O\ D$, as described in **Experimentation**. The next R lines denote the grader grid. The next R lines after that denote the charge grid.

If $D = 1$ when the grader was run, the output file will contain a move $t\ x\ y$, followed by the grid **your**

program sees after said move. Else, the file only has the list of moves t x y.

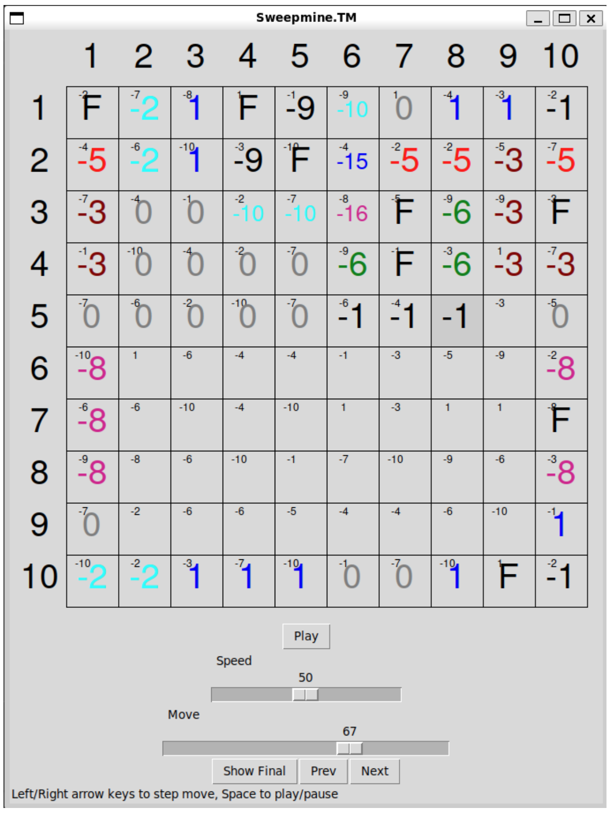
The final line is either one of `DIE`, `WIN`, or nothing. The first two are self explanatory. The last one means the grader stopped (most likely due to a runtime error) before returning a verdict.

Visualiser

For ease of viewing, a visualiser (`visualiser.py`) has been provided. This script depends on the Python library `tkinter`. Ensure `sweepmine.out` is non-empty, before running the following command:

```
python3 visualiser.py
```

A window should open as below:



Sweepmine.TM

	1	2	3	4	5	6	7	8	9	10
1	-2 F	-7 -2	-8 1	1 F	-1 -9	-9 -10	1 0	-4 1	-3 1	-2 -1
2	-4 -5	-6 -2	-10 1	-3 -9	-10 F	-4 -15	-2 -5	-2 -5	-5 -3	-7 -5
3	-7 -3	-4 0	-1 0	-2 -10	-7 -10	-8 -16	-5 F	-9 -6	-9 -3	-2 F
4	-1 -3	-10 0	-4 0	-2 0	-7 0	-9 -6	-1 F	-3 -6	1 -3	-7 -3
5	-7 0	-6 0	-2 0	-10 0	-7 0	-6 -1	-4 -1	-1	-3	-5 0
6	-10 -8	1	-6	-4	-4	-1	-3	-5	-9	-2 -8
7	-6 -8	-6	-10	-4	-10	1	-3	1	1	-8 F
8	-9 -8	-8	-6	-10	-1	-7	-10	-9	-6	-3 -8
9	-7 0	-2	-6	-6	-5	-4	-4	-6	-10	-1 1
10	-10 -2	-2 -2	-3 1	-7 1	-10 1	-1 0	-7 0	-10 1	1 F	-2 -1

Left/Right arrow keys to step move, Space to play/pause

There are a number of key points to the visualiser:

- To change the size, edit `board_size` (at the top of the script).
- The big numbers in each cell show the value of revealed squares. The little numbers show the charge value of each square. Blank squares are unrevealed tiles. Flag symbols denote flagged tiles.
- The numbers along the left and top index the rows and columns respectively.
- Incorrectly placed flags (`F`) will be marked **red** (different to standard **black** flag colour).
- The speed slider changes the speed of playback, which can be toggled with either the spacebar or with the play button. (Speed also capped by move refresh speed, can't be bothered to optimise)
- You can step backwards and forwards with the previous and next buttons, or the left and right arrow keys respectively. You can also drag the slider to view a specific move.
- The `Show Final` button toggles the full board.
- The visualiser shows up to the last move **before** program exit. If the last move reveals a mine, the square will be marked by a mine (`B`) symbol. If the last move does not reveal all safe squares, the program did not run until completion.