CS100 Homework 4 (Spring, 2022)

Deadline: 2022-04-02 23:59

Late submission will open for 24 hours after the deadline, with -50% point deduction.

Problem 1. Landlord

"Landlord" is one of the most popular card games played in China. The landlord's aim is to be the first to play out all his cards in valid combinations, and the peasants win if any one of them manages to play all their cards before the landlord.

This game uses a 54-card pack including two jokers, red and black. The cards rank from high to low:

red joker, black joker, 2, A, K, Q, J, 10, 9, 8, 7, 6, 5, 4, 3

Suits are irrelevant in this game.

We have made some simplifications to this game. All valid combinations are described in this table 2. Note that the two jokers are **NOT** regarded as a pair, and a quad with two jokers is not allowed.

The cards you played each time must be a valid combination. What is the minimum number of times you need to play all the cards?

Input Description

You will receive two integers T and n in the first line, separated by a space. T stands for the number of rounds. n stands for the number of cards for each round.

Each of the following T lines contains n integers, separated by a space. Each integer stands for a card, as shown in table 1. Jokers will appear at most twice. Each card except the jokers will appear at most 4 times. It is guaranteed that $n \leq 17$ and $T \leq 10$.

Table 1: Cards

Card	A	2-10	J	Q	K	Joker
Number	1	2-10	11	12	13	14

Output Description

You need to output T lines. Each line has an integer indicating the minimum number of times to play all the cards.

Sample input 1

```
1 8
3 3 3 3 4 5 6 7
```

Sample output 1

```
2
```

Sample input 2

```
1 17
1 1 4 4 4 4 5 6 7 8 9 10 10 10 11 12 13
```

Sample output 2

```
2
```

Hints:

- You can use *recursion* to solve this problem. Once you find a valid combination, you can move on to the rest of the cards.
- You can use an array to store all the cards by ascending rank.

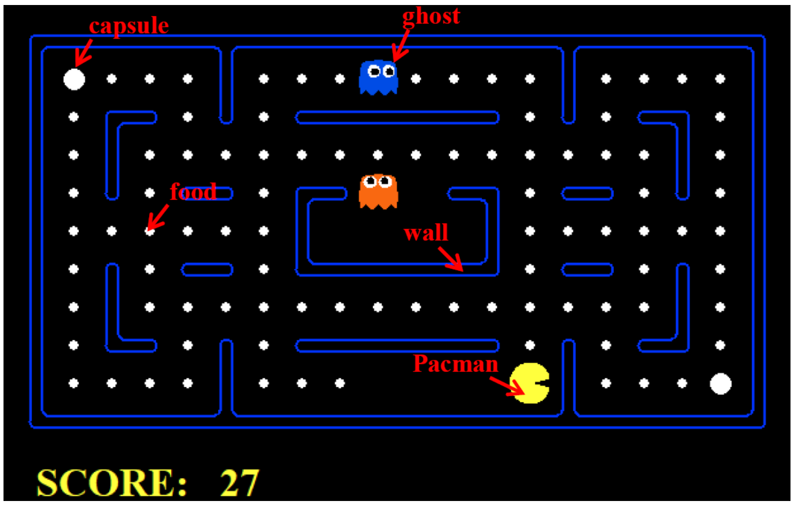
Table 2: Valid combinations

Solo	Any single card	5
Pair	2 cards of the same rank	6-6
Trio	3 cards of the same rank	7-7-7
Trio with an attached card	A trio with any single card	6-6-6-8
Trio with an attached pair	A trio with any pair	9-9-9-K-K
Sequence	At least 5 cards of consecutive rank. Twos and jokers cannot be used	7-8-9-10-J
Sequence of pairs	At least 3 pairs of consecutive ranks. Twos and jokers cannot be used	10-10-J-J-Q-Q-K-K
Sequence of trios	At least 2 trios of consecutive ranks. Twos and jokers cannot be used	4-4-4-5-5-5
Quad with 2 attached cards	A quad with 2 cards of different rank	6-6-6-6-8-9
Quad with 2 attached pairs	A quad with 2 pairs of different rank	6-6-6-6-8-8-9-9
Rocket	A pair of jokers	Black Joker-Red Joker
Bomb	4 cards of the same rank	2-2-2-2

Problem 2. Pac-Man

Pac-Man is a classic video game released in 1980. Believe it or not, you have already learned everything needed to make this game on your own!

This is a screenshot from a Pac-Man game. Several components of the game, shown below, are Pacman, walls, foods, energy capsules, and ghosts.



Our version of "Pac-Man" does not feature such graphics. Disappointingly, it is composed of all ASCII characters. Pacman is represented by a similar-looking `'C'`, walls by `'#'`, foods by `'.'`, capsules by `'o'`, and ghosts by `'@'`. (Boundaries are printed by the provided templates, so you do not need to consider.)

```
 ------------------
|....#........#....|
|.##.#@######.#.##.|
|.#. ..........@.#.|
|.#.##.##..##.##.#.|
|..o..o#....#o..o..|
|.#.##.######.##.#.|
|.#..............#.|
|.##.#.######.#.##.|
|....#...    C#....|
\------------------/

Score: 27
There are 100 foods remaining!
Pacman wants food! (control by w/a/s/d/i, confirm by Enter)
```

Along with the code templates, we have also provided compiled executables for Windows, MacOS and Linux. We recommend you to run the executable and play the game to get a basic idea of what you should do. Also, when you have any doubt on whatever part you write, if your behavior matches our example program, it is (almost) sure to say you are safe.

- On MacOS or Linux, in order to run the provided executable, you may need to run the command `"chmod +x Pacman_MacOS"` or `"chmod +x Pacman_Linux"` in your terminal first. Raise a question on Piazza if you have any problem.

Unlike in the real-time video game Pac-Man, we control our Pacman frame by frame. The game will pause every frame and wait for your input. You can type `"w/a/s/d"` into the game to move Pacman, or "i" (idle) for it to stand still. Confirm your input by "Enter", and the game will show you the next frame.

Part A. Make a Game From Scratch

The game operates by a `struct game` structure. We have already specified some components of it:

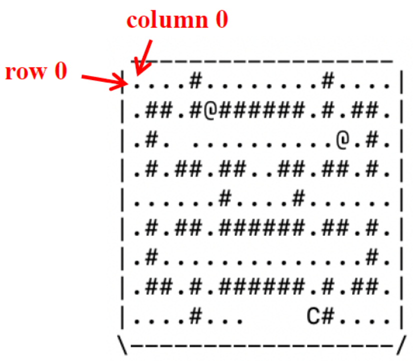
```
typedef struct game {
  char** grid;      // a 2-dimensional array of characters to display the game;
  int rows;         // number of rows of the grid;
  int columns;      // number of columns of the grid;
  int foodCount;    // number of remaining food in the game;
  int score;        // current score;
  GameState state; // the state of the game, one of losing, onGoing, or winning.
} Game;
```

Feel free to add more components to this structure if you would like to.

- Although it may appear that there is only one "global" game, or that the components of the game are "global" to the Pacman (or ghosts), it is **NOT** a wise choice to declare global variables for them. If you choose to, for example, write Pacman (or ghosts in part B) in a separate structure, and want Pacman (or ghosts) to know about the game, a good option is to store a `Game*`, a Game pointer, as a member of that structure. You will learn more about such conventions when you practice Object-Oriented Programming (OOP) in C++.

Apart from the structure, there are many functions you need to write for this game to operate. When implementing provided function prototypes in the templates, you should follow the instructions below, or see the comments. You **MUST NOT** modify the function names, or add/remove parameters. You can also add more functions if you like. We will not check any function that are not provided.

A game of given rows and columns is created by calling the function `Game* NewGame(int rows, int columns)`. You should dynamically allocate space for a `Game` pointer, and initialize all member variables of your `Game` structure. (For example, `foodCount and score should be initialized to 0`.) The member `grid` should be created by dynamically allocating a 2-dimensional array of given size. Boundary is not included in either rows or columns, and the cell at top-left corner is at row 0 and column 0.



```
 ------------------
|....#........#....|
|.##.#@######.#.##.|
|.#. ..........@.#.|
|.#.##.##..##.##.#.|
|......#....#......|
|.#.##.######.##.#.|
|.#..............#.|
|.##.#.######.#.##.|
|....#...    C#....|
\------------------/
```

When the game ends, the function `void EndGame(Game* game)` is automatically called. In this function, you should `free` any memory you dynamically allocated, such as `grid`. After that, you should `free` the parameter `game`, as it is also dynamically created.

Walls, foods and Pacman are added to the game by functions `AddWall`, `AddFood`, and `AddPacman`. All of these game components can only be added to an empty cell. Pacman, in particular, cannot be added to the game if there is already a Pacman. After you add any item, you should modify the `grid` in your `Game` structure to make sure it displays correctly.

Finally, you can write the function `void MovePacman(Game* game, Direction direction)` to control your Pacman. `Direction` is an `enum` of {`up, down, left, right, idle`}. The rule to move your Pacman is as follows:

- On `idle`, Pacman will stay still.
- If Pacman would move to an empty cell, Pacman will do so successfully;
- If Pacman would move to a food cell, Pacman will move to it and eat the food. Your score will increase by `FOOD_SCORE = 10`. If Pacman eats the last food, you win the game. You should mark the `state` of this game as `winning`.
- If Pacman would bump into a wall or a boundary, Pacman will stay still.

In any of the cases above, your score should decrease by 1, for one turn you have played.

How to play this game:

You can create your custom game in your `main()` function by calling `NewGame`. After that, you can add walls and foods to any specific location. Don't forget to add a Pacman to the game.

When your game is prepared, you can call the provided `PlayGame` function. When you win or lose, `PlayGame` will terminate by calling `EndGame`.

If your game runs... Congratulations! You now have a "complete" Pacman game, where you can move, eat food, and win! You can already submit it to OJ for Part A, but if you wonder why your game is a little different and boring, let's go to Part B.

Submission Guidelines:

When you submit your code, your `main` function will be replaced by one on OJ. and the functions you implemented will be directly called with parameters. Therefore, you **MUST NOT** modify the function names, add or remove parameters, or put your code in `main` function. Otherwise, you will **NOT** receive any score.

Part B. Ghosts!

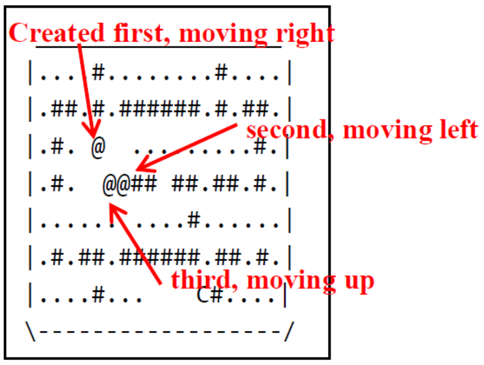
Your game is missing a part of the greatest fun - the ghosts. In this part, you will add ghosts and energy capsules to your game so that it will become more playable.

We do not force any restrictions on how you should store your data for ghosts and capsules. Do you think you need to write a structure, especially for ghosts? If so, what do you need to store in it? Your design can be in any way you like (but still try not to use global variables), as long as it meets the requirements below:

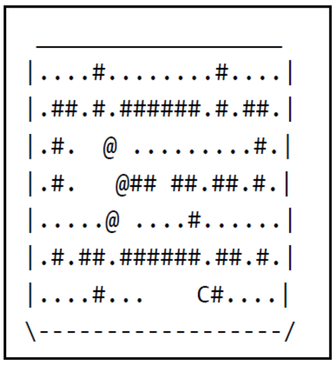
Requirements for ghosts:

1. There are at most `MAX_GHOSTS = 30` ghosts.
2. Ghosts are added to the game by the function `bool AddGhost(Game* game, int r, int c, Direction direction)`.
   - This function is slightly different, as ghosts can be added on a cell with food or a capsule. Ghosts cover foods and capsules in display, so their cells (originally `'.'` or `'o'`) will be displayed in `'@'`. However, those food or capsules must still exist, and should be displayed again when ghosts leave their cells.
   - Direction defines how a ghost moves. Ghosts move either in a horizontal line or a vertical line. The parameter `direction` in this function is the ghost's initial direction.
3. Ghosts are moved by the function `void MoveGhosts(Game* game)`.
   - This function will move all ghosts in the game by one step to their own directions.
   - Ghosts should be moved in the order they were added.
   - if a ghost would move onto a cell with food or a capsule, it will cover the food or capsule in display, so that cell (originally `'.'` or `'o'`) will be displayed in `'@'`. However, that food or capsule must still exist, and should be displayed again when this ghost leaves that cell.
   - If a ghost would bump into a wall, another ghost, or a boundary, its direction will reverse, and it will try to move in the new direction immediately this turn. If then it would bump into another wall/ghost/boundary, it will stop and won't move for this turn, with its direction reversed.

To better explain the case where a ghost bumps into another ghost, see this situation:



```
 __________________
|....#........#....|
|.##.#.######.#.##.|
|.#. @  .........#.|
|.#.  @@## ##.##.#.|
|...... ....#......|
|.#.##.######.##.#.|
|....#...    C#....|
\------------------/
```



```
 __________________
|....#........#....|
|.##.#.######.#.##.|
|.#.  @ .........#.|
|.#.   @## ##.##.#.|
|.....@ ....#......|
|.#.##.######.##.#.|
|....#...    C#....|
\------------------/
```

Now it is possible to lose the game. By rules, Pacman always moves first. If Pacman directly bumps into a ghost, Pacman will move to that cell, and get killed. You should mark the game state as `losing`. If a food or a capsule is below that ghost, Pacman cannot eat it. If Pacman moves to a cell that a ghost also attempts to move to, Pacman will perform a successful move, and the ghost then moves onto Pacman's cell. You will also lose the game.

Requirements for capsules:

1. Capsules are large food that give Pacman superpower. Therefore, capsules counts to the number of foods in the game. Pacman must eat all food and capsules to win.
2. Capsules are added by the function `bool AddCapsule(Game* game, int r, int c)`. Like food, a capsule cannot be added to a cell with a wall, a Pacman, or a ghost. However, a capsule can be added to a cell with a food, resulting in that food being **upgraded** to a capsule.
3. When Pacman eats a capsule, your score will increase by `CAPSULE_SCORE = 50`, and Pacman will gain superpower for its next `CAPSULE_DURATION = 10` moves. Its superpower is that:

- All ghosts will be scared, and their display change from '`@`' to '`X`'. When Pacman's superpower expires, they change back to their cute evil faces '`@`'.
- Scared ghosts are slowed down by 50%, which is showed by that they cannot move every other turn. (We cannot move ghosts by half a cell, after all!) They will be able to move on the same turn when Pacman eats a capsule, but cannot move the next turn. This goes on until Pacman's superpower expires.
- When with superpower, Pacman can eat ghosts! When Pacman moves onto a grid with a scared ghost, it eats the ghost, earning a score of `GHOST_SCORE = 200`. If there is a food or a capsule below that ghost, Pacman eats it as well. That ghost will not respawn (Different from the original Pac-Man game) and can be removed from the game. The same goes for the case when a scared ghost bumps into Pacman. (Different again. In the original game, scared ghosts always run away from Pac-Man.)

4. Pacman's superpower activates immediately when it eats a capsule, and counts down right after Pacman's turn, starting from its next turn. For example, if Pacman and a scared ghost attempt to move onto a same grid on Pacman's 10^{th} turn of superpower, Pacman will move first, but its superpower will immediately expire, and that ghost, not scared anymore, will kill Pacman. (No...!) In other words, Pacman's superpower ends after 10 turns at **the same moment** of eating a capsule.
5. If Pacman eats another capsule while it has superpower, the duration of superpower will be refreshed to 10 turns, rather than stack. In this case, it is possible that a scared ghost has already moved on the turn (the new 9^{th} turn) right before the turn when Pacman's superpower expires (10^{th}). That ghost can still move on its next turn (10^{th}), because it will not be a scared ghost then.

Submission Guidelines:

Finally, you can add ghosts and capsules to your game in your `main` function, and enjoy the finished game of Pacman. The submission is the same as how you did for part A. Good luck and have fun!