Important message on plagiarism

The single most important point for you to realize before the beginning of your studies at ShanghaiTech is the meaning of “plagiarism”:

Plagiarism is the practice of taking someone else's work or ideas and passing them off as one's own. It is the misrepresentation of the work of another as your own. It is academic theft; a serious infraction of a University honor code, and the latter is your responsibility to uphold. Instances of plagiarism or any other cheating will be reported to the university leadership, and will have serious consequences. Avoiding any form of plagiarism is in your own interest. If you plagiarize and it is unveiled at a later stage only, it will not only reflect badly on the university, but also on your image/career opportunities.

Plagiarism is academic misconduct, and we take it very serious at ShanghaiTech. In the past we have had lots of problems related to plagiarism especially with newly arriving students, so it is important to get this right upfront:

You may…

• … discuss with your peers about course material.
• … discuss generally about the programming language, some features, or abstract lines of code. As long as it is not directly related to any homework, but formulated in a general, abstract way, such discussion is acceptable.
• … share test cases with each other.
• … help each other with setting up the development environment etc.

You may not …

• … read, possess, copy or submit the solution code of anyone else (including people outside this course or university)!
• … receive direct help from someone else (i.e. a direct communication of some lines of code, no matter if it is visual, verbal, or written)!
• … give direct help to someone else. Helping one of your peers by letting him read your code or communicating even just part of the solution in written or in verbal form will have equal consequences.
• … gain access to another one’s account, no matter if with or without permission.
• … give your account access to another student. It is your responsibility to keep your account safe, always log out, and choose a safe password. Do not just share access to your computer with other students without prior lock--out and disabling of automatic login functionality. Do not just leave your computer on without a lock even if it is just for the sake of a 5--minute break.
• … work in teams. You may meet to discuss generally about the material, but any work on the homework is to be done individually and in privacy. Remember, you may not allow anyone to even just read your source code.

With the Internet, "paste", and "share" are easy operations. Don’t think that it is easy to hide and that we will not find you, we have just as easy to use, fully automatic and intelligent tools that will identify any potential cases of plagiarism. And do not think that being the original author will make any difference. Sharing an original solution with others is just as unethical as using someone else’s work.

CS100 Homework 4 (Spring, 2021)

In this homework, you are required to do advanced practices with all you learned of C programming language, focusing mainly on recursions and structures. Have fun and enjoy coding. Practice, practice and practice.

Submission deadline:

2021-04-14 23:59:59

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

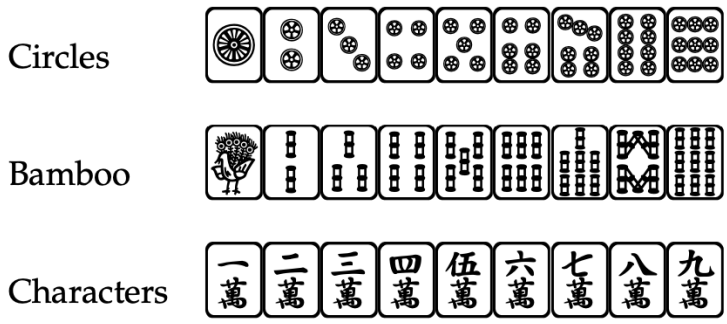
Special Note:

We have provided framework for all problems. You should start with the framework, and finish any function left for you. When submitting, do not submit the main function. Your submission should include the implementation of all required functions. A simple and safe way is to just submit all your codes, except the main function.

Problem 1. Does Gezi Wang win? (25+10 points)

Mahjong is a traditional game. There are 34 basic tiles in classical mahjong, which is presented below (1.1 and 1.2). A full set of mahjong tiles contains **four identical tiles** of each of these.

1.1 The three suits



In particular, the one of bamboo is often decorated with bird, the design of which often varies between mahjong sets). For convenience, we use “1p, 2p, 3p, 4p, 5p, 6p, 7p, 8p, 9p” to represent circles (饼/筒), “1s, 2s, 3s, 4s, 5s, 6s, 7s, 8s, 9s” to represent bamboos (条/索) and “1m, 2m, 3m, 4m, 5m, 6m, 7m, 8m, 9m” to represent characters (万). For an instance, “7m” represents the tile “七萬”.

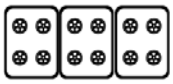
1.2 The honours

In addition to the suit tiles, there are **seven** different honours: **four winds and three dragons**. The

winds are shown in the order east-south-west-north (東-南-西-北), and the dragons are shown in the order red-white-green (中-白-發). The shorthand for honours are: “1z”, “2z”, “3z”, “4z” for “東”, “南”, “西”, 北“”, and “5z”, “6z”, “7z” for “中”, “白”, “發” (which is a little different from some popular mahjong games, like *maj-soul* or *tenhou*).

With **four of each of above tiles**, a mahjong set consists of **136 tiles**.

There are no fixed rules for mahjong. In this problem, we just need to consider a simplified rule of classical mahjong. Under this rule, mahjong is played by 4 players. Each player has 13 random initial tiles in his/her hand. A player’s turn contains two phases: first, the player will draw a tile from the remaining tiles (the wall, 牌山), and judge if he/she wins (和牌). If he/she wins, the game ends. If not, the player will take the second phase of his/her turn, which is to drop a tile from his/her hand (the player can decide which card to drop). The four players will take their turn one by one, until one player wins.
The **14 tiles** in a player’s hand (the status after drawing a tile from the wall) is called the **Mahjong Hand**. We say a player wins if and only if his/her mahjong hand is composed of **exactly four sets and a pair**. A set is either a Chow (吃; 顺子), or a Pung (碰; 刻子). A Chow is **three consecutive tiles in same suit**. Chows **cannot** be made by dragons or winds. A Pung is composed of **three identical tiles**. A pair contains **two identical tiles**. An example of Chow and Pung is shown below.

Chow  Pung 

We don’t need to consider some special rules like thirteen orphans or seven pairs in this problem.

For example, if some player’s hand is: [1s, 1s, 1s, 2m, 3m, 4m, 5m, 6m, 7m, 1z, 1z, 1z, 8s, 8s], then he/she wins the game because these 14 tiles can be decomposed as four sets: [1s, 1s, 1s] (a Pung); [2m, 3m, 4m] (a Chow); [5m, 6m, 7m] (a Chow); [1z, 1z, 1z] (a Pung), and a pair: [8s, 8s].

One day, Gezi Wang is playing mahjong with little Gugu and other friends, but he does not familiar with these mahjong rules. You need to write a function to help him to judge whether he wins the game after he draws a card from the wall. The function declaration is given below. (We will call your function directly to judge your program on OJ, please don’t modify this declaration).

```
int CheckWin(char* mahjongHand[]);
```

It’s guaranteed that the length of `mahjongHand` is 14, and each string in `mahjongHand` is a legal tile (either “1s” - “9s”, “1p” - “9p”, “1m” - “9m” or “1z” - “7z”), and each identical card will appear at most 4 times. But it’s not guaranteed that the tiles are in order. Your function needs to return **1** if he wins the game, and return **0** otherwise. You can add other helper functions if you need.

Input description:
There are no specified inputs. You can costume your `main()` function as you like to test your function. For example, you may invoke your function using the following example.

```
char* hand[14] = {"1s", "1s", "1s", "2s", "3s", "4s", "5s", "6s", \
```

```
                    "7s", "8s", "9s", "9s", "9s", "9s"};
    int result = CheckWin(hand);
```

In this case, we will get `result = 1`, respectively.

Output description:

There are no specified outputs, just write the "`CheckWin`" function.

Sample 1:

```
    char* hand[14] = {"1s", "1s", "1s", "2p", "2p", "2p", "5s", "6s", \
                      "7s", "6z", "6z", "9s", "6z", "9s"};
```

Expected return value of `CheckWin(hand)`: 1.

Sample 2:

```
    char* hand[14] = {"1s", "1s", "1s", "2z", "3z", "4z", "5s", "6s", \
                      "7s", "6z", "6z", "6z", "9s", "9s"};
```

Expect return value of `CheckWin(hand)`: 0, for Chows cannot be made up with "2z", "3z", "4z".

Bonus: Count waiting tiles!

Gezi Wang wants to know drawing which card(s) will he win the game before his turn. You need to write another function "`CountWaitingTiles`" to help him. In this function, you should count the number of **different** titles that leads him to win (听牌). The declaration of this function is:

```
int CountWaitingTiles(char* currentTiles[])
```

Notice that there are only 13 tiles in Gezi Wang's hand before his turn. Consequently, the length of parameter "`currentTiles`" is guaranteed to be 13, and all tiles are valid. You need to return the number of different tiles that leads him to win. Also, we will test your function directly, so you can test it in `main()` as you want, but the function declaration cannot be modified. If you have no idea on this problem, just return -1 to make the compiler happy.

For example, if the following figure shows Gezi Wang's hand (["3p", "3p", "3p", "7p", "8p", "9p", "2s", "2s", "4s", "5s", "7s", "8s", "9s"]), then there are 2 different tiles "3s" and "6s" will lead him to win.

Because if he draws a "3s" in his turn, his hand is composed of 4 sets: "3p, 3p, 3p"; "7p, 8p, 9p", "3s, 4s, 5s", "7s, 8s, 9s" and a pair "2s, 2s", and "6s" is similar (except one set is "4s, 5s, 6s"). In this case, your function should return 2 for there are 2 different tiles he is waiting for.

In particular, if the tile that he is waiting for is used up (already 4 such tiles in his hand), then you should ignore this tile when counting the waiting tiles. If no tile will lead him to win, return 0.

Sample1:

```
char* tiles[13] = {"1s", "1s", "1s", "2p", "2p", "2p", "5s", "6s", \
                   "7s", "6m", "6m", "6m", "7m"};
```

Expect return value of `CountWaitingTiles(tiles)`: 3, because “5m”, “7m”, “8m” will lead him to win.

Sample2:

```
char* tiles[13] = {"1s", "1s", "1s", "2p", "3p", "4p", "4p", "4p", \
                   "4p", "5p", "6p", "1z", "1z"};
```

Expect return value of `CountWaitingTiles(tiles)`: 3. Although “1p”, “4p”, “7p” and “1z” will lead him to win, but there are already four “4p” in his hand.

Sample3:

```
char* tiles[13] = {"1s", "1s", "1s", "2s", "3s", "4s", "5s", "6s", \
                   "7s", "8s", "9s", "9s", "9s"};
```

Expect return value of `CountWaitingTiles(tiles)`: 9, because either “1s”, “2s”, “3s”, “4s”, “5s”, “6s”, “7s”, “8s” or “9s” will lead him to win.

Problem 2. Constructing a Linked List (25 points)

In this problem, you are required to implement a singly linked-list. Our singly linked lists have at least these members: a pointer "`head`" pointing to the first node, and a pointer "`tail`" pointing to the last node, and a "`size`" indicating the number of nodes. In addition, the "`next`" pointer of the last node should be set to NULL, indicating the end of the list. For interior node, it should have a pointer "`next`" pointing to its next node in list, which should also be pointed to by its previous node using its "`next`" pointer.

The basic structure of our list:

```
An empty list looks like this
        (head)--->(NULL)   (tail)--->(NULL)                size = 0
A list with two elements in it looks like this:
                    +-------+    +-------+
         (head)--->|   1   |--->|   2   |--->(NULL)       size = 2
                    +-------+    +-------+
                                   ↑ (tail)
```

Note: You can see this statement in the template:

```
typedef /*YOUR STRUCTURE*/ LinkedList;
```

Please write your structure at the position of this comment, or, before this statement and put the name of your structure at the position of this comment. You can refer to the slides for detailed information on forms when writing a structure. The head must be named as "head", the tail must be named as "tail", and the size must be named as "size".

Note: The `head` and `tail` in this structure are pointers to nodes. You need to initialize the head and tail to NULL when the list is created and before any node is added.

For each node in list, it should contain:
(1) An int, named "`value`". (2) A pointer to next node, named "`next`"

More detailed, you need to implement these functions in this part:

- `void list_init(LinkedList* l);`
  In this function, you should initialize your list, the size should be initialized to 0, the head and tail should be NULL.
  It does not have a return value.

- `void list_insert(LinkedList* l, int val);`

In this function, you should create a new node which value equals to "`val`", and insert this node into your list **immediately after the last node (which tail points to).** Also, you need to update tail to the new position.
It does not have a return value.
Note: Do not forget to update size of the list.

- `void list_clear(LinkedList* l);`

  In this function, you should clear the whole linked list (i.e., remove all nodes and data in them), and **free all memory you allocated.**
  It does not have a return value.
  Note: Do not forget to update size of the list.

- `Node* list_remove(LinkedList* l, Node* target);`

  In this function, the additional parameter is a pointer to a node (if well-behaved, this node should be in the list). First, you should traverse the list and check if the target is in the list. If it is not found, return NULL. If it is found, you should remove it from list and re-link **involved pointers(*)**, and make sure you don't forget to **free the memory you allocated**! If you have successfully removed a node, this function should **return the (originally) next node of target node**.

  Note: Do not forget to update size of the list.
  (*) When you delete a node, there are many questions that you should ask yourself. Take the following one as a hint:
  Is it a node in the middle of the linked list, or maybe in a special position? Is there any pointer that the special position may affect?

Problem Description:

You will need to implement a structure of a LinkedList. To do this, you may also need a structure of a Node.

You also need to implement 4 LinkedList operations as shown. Please pay attention to parameter lists and return types of these functions.

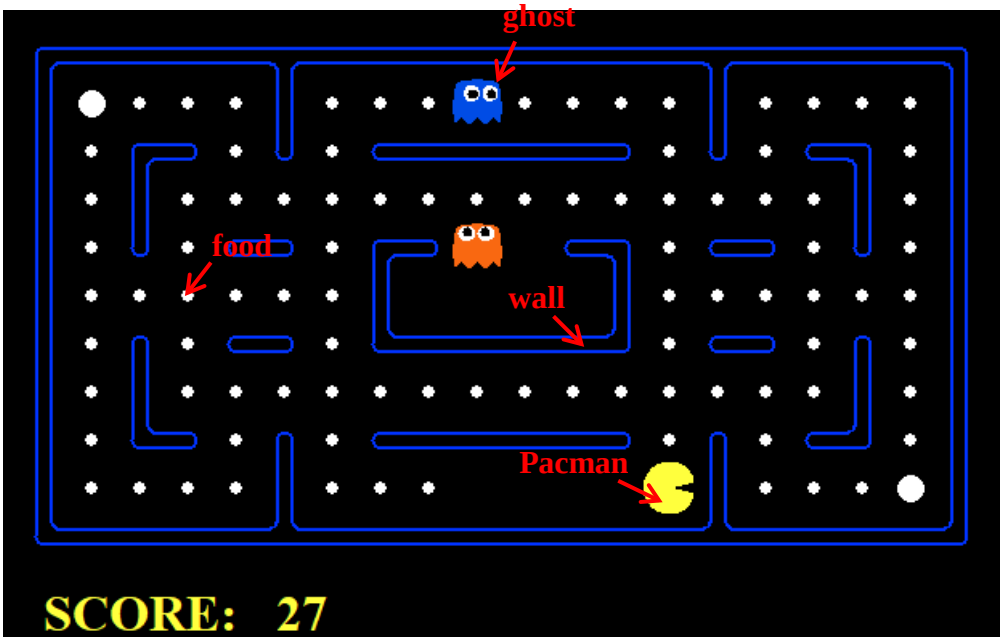
The type name of your linked list structure and node structure should be type-defined to be `LinkedList` and `Node`, respectively.

You should not change the name of any function listed above. Otherwise compile errors may happen.

Problem 3. Pacman - Part 1 (30 points)

Pacman is a classic video game. Believe it or not, you have already learned everything needed to make this game on your own!

This is a screenshot of a real Pacman game. Several components of the game, shown below, are Pacman, foods, walls, and ghosts. [1]



Our version does not feature such graphics. Disappointingly, it is composed of all ASCII characters. Pacman is represented by a similar-looking `'C'`, foods by `'.'`, walls by `'#'`, and ghosts by `'@'`. (Boundaries are different because that's part of framework, and you do not need to consider.)

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

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

[1] To those who are curious: Those big foods are called "capsules", but we do not have them in our game.

Along with the code frameworks, we have provided you a compiled executable for Windows, and also one for MacOS and one for Linux. (On MacOS or Linux, you may need to run the command `"chmod +x FILENAME"` in your terminal. Raise a question on Piazza if you cannot run it.) 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's (almost) sure to say you're safe.

Unlike the real Pacman, we control our Pacman frame by frame. After each move, the game will pause 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.

How to make this game from scratch:

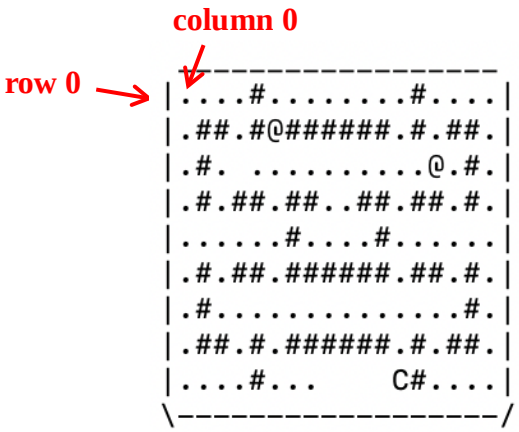
The game operates by a `struct game` structure. We have already specified some components of it. They are:

- `char** grid`, a 2-dimensional array of characters to show the game;
- `int columns, int rows`, size of the grid;
- `int foodCount`, the number of remaining foods on the board;
- `int score`;
- `GameState state`, to mark whether the game is `losing`, `onGoing`, or `winning`.

We left for you anything more you would like to include in this structure.

Apart from the structure, there are many functions you should write for this game to operate. You can also add more functions if you feel like so. We won't check any function that are not provided in the framework.

A game is created by calling the function `Game* NewGame(int rows, int columns)`. You should implement this function so that it creates a new game with given rows and columns. Boundary is not included in either rows or columns, and the left-upper corner is at row 0 and column 0. You should dynamically allocate space for a Game pointer, dynamically allocate a 2-dimensional char grid, and initialize any other value in your Game structure. (For example, `foodCount` and `score` should be initialized to 0)



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

Walls, foods and Pacman are added to the game by functions `AddWall`, `AddFood`, and `AddPacman`. All of these 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 move to it normally;

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 in the game, 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've 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 your Pacman to the game.

When your game is prepared, you can call the `PlayGame` function we provided. `PlayGame` will terminate when you win or lose. You should call your `EndGame` on the next line to clean up.

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 1, but if you wonder why your game is a little different and boring, go to Part 2.

How to submit:

We will not use your main function, but rather call the functions you wrote. Feel free to use your main function to create your own games, but do not submit it to OJ.

Problem 4. Pacman - Part 2 (10 points)

Your game is missing the most fun part – the ghosts. In this part, you should add ghosts to your game.

We do not force any restrictions on how you should store your data for ghosts, nor do we provide you any hint. Do you think you need to write a structure? If so, what do you need to store in it? You can write in any way, as long as it meets the requirement below:

There are at most `MAX_GHOSTS = 30` ghosts.

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. Ghosts cover foods in display, so that cell will be a `'@'`. However, that food must still exist, and will be displayed again when this ghost leaves that cell.

Direction defines how a ghost moves. Ghosts move either in a horizontal line or a vertical line. The `direction` in this function defines how the ghost moves initially.

Ghosts are moved by the function `void MoveGhosts(Game* game)`.

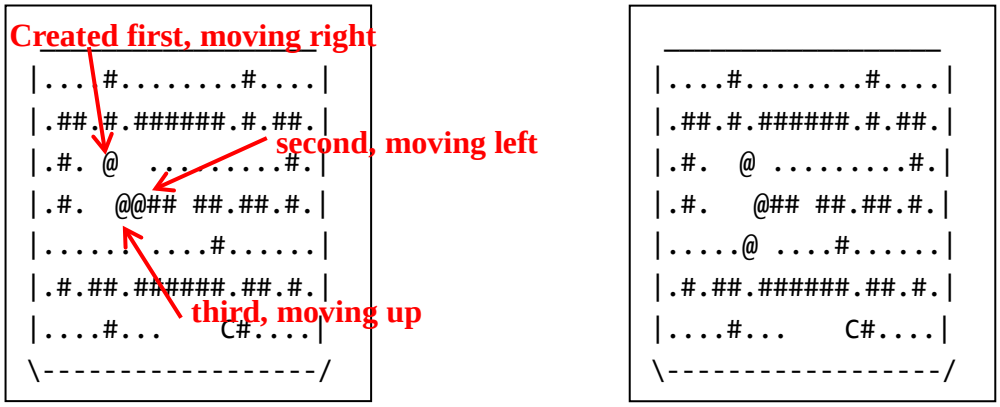
This function will move all ghosts in the game by one step to their own directions.

Ghosts should be moved in the order they were added.

If a ghost would 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 it bumps into another wall/ghost/boundary, it would stop and won't move for this turn.

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



Now it's possible to lose the game. By rules, Pacman always moves first. If Pacman directly bumps into a ghost, Pacman successfully move to the cell, and get killed. If that ghost is on a food, Pacman cannot eat that food. After Pacman moves, if a ghost then moves to Pacman's cell, you will also lose the game.

Finally, you can add ghosts to your game in your main function, and enjoy the finished game of Pacman. The submission is similar to how you did for part 1. Good luck!