I've solved every problem. Where I have struggled are

1. hash collision(i.e. using LRU mechanism, building my own table) too often when I want to speed it up
2. speed up methods, tried a lot but most of them don't work well
3. running too slow makes me waiting for lots of time

that I don't know how to optimze the solutions of Problem 6. Some sample result in problem 6 are worse than the result in problem 4.

problem 1

1. Prerequisites(they also will be used in the following questions):

set the directions, the score from every action, the location of overlapping between the ghost and food.

```
ALL_DIRECTIONS = {"E": (0, 1), "N": (-1, 0), "S": (1, 0), "W": (0,
-1)}
EAT_FOOD_SCORE = 10
PACMAN_EATEN_SCORE = -500
PACMAN_WIN_SCORE = 500
PACMAN_MOVING_SCORE = -1
OVERLAP = dict()
```

2. parse and generate result(they also will be used in the following questions):

generate the result based on the actions, subjects(who is taking action in one round), and the layout of game-board.

```
def generate_result(actions: list[str], subjects: list[str],
                        layouts: list[list[list[str]]], scores:
list[int], winner, seed) -> str:
```

parse the problem from parse.py

```
def parse_problem(problem: str) -> (str, list[list[str]])
```

3. core function and method

randomly choose a direction to help the ghost and pacman move

```
def determine_direction_and_random_choose(layout: list[list[str]],
row: int, col: int) -> str:
        """
        :param layout: the layout
        :param row: the subject's position
        :param col: the subject's position
        :return: the eligible directions

        """
```

pacman choose a direction to move and get the score, or nothing, or being eaten. the following situations are needed to take care:

1. the next moving place is " "
2. the next moving place is "W" but there isn't food
3. the next moving place is "W" but there is food
4. the next moving place is "." but after eating that there has another food.
5. the next moving place is "." but after eating that there has no more food.

```
def move_pacman(layout: list[list[str]], x: str, y: str) -> (bool,
int, int, list[list[str]]):
        """
        :param layout: the layout of game-board.
        :param x: the row position of pacman
        :param y: the column position of pacman
```

```
        :return: (
            bool: the game is finished or not;
            int: new_x;
            int: new_y;
            layout: list[list[str]]
        )

        find a direction to move pacman.

        """
```

Move the ghost to somewhere. One more thing needs to take care is that the overlap between food and ghost.

```
    def move_ghost(layout:list[list[str]], x:int, y:int):
        """
        :param layout: the layout of game-board.
        :param x: the row position of pacman
        :param y: the column position of pacman
        :return: (
                bool: the game is finished or not;
                int: new_x;
                int: new_y;
                layout: list[list[str]]
            )
        find a direction to move pacman.
        """
```

problem 2

NO.	NUM TRAILS	TIME(SECONDS)	WINNING RATE
1	100	0.14379024505615234	100%
2	100	0.005718708038330078	100%
3	100	0.015811920166015625	89%
4	100	1.291214942932129	100%
5	100	101.73687624931335	76.0%

Core function

```
@functools.lru_cache(maxsize=2048)
def get_distance(x1, y1, x2, y2) -> int:
        """
  calculate the manhattan distance from point 1 to point 2
        """
    return abs(x1 - x2) + abs(y1 - y2)
```

use manhattan distance to evaluate the distance between two objects. The reasons are as follows:

1. Manhattan distance records how many steps these two objects could   if both of them move wisely.
2. easy to calculate

Since it will call this function lots of time, LRU is all my need.

```
@functools.lru_cache(maxsize=1024)
def evaluate_func(p_r: int, p_c: int, w_r: int, w_c: int, food_r:
int, food_c: int) -> int:
```

```
        """
        :param p_r: pacman row index
        :param p_c: pacman column index
        :param w_r: ghost row index
        :param w_c: ghost column index
        :param food_r: food row index
        :param food_c: food column index
        :return: the score

        score consists of a penalty and the distance from food.
        if the distance between ghost and pacman is so close, there
will be a penalty. otherwise, pacman always finds
        the closest path to the nearest food
        """
    food_distance = get_distance(p_r, p_c, food_r, food_c)
    ghost_distance = get_distance(p_r, p_c, w_r, w_c)
    penalty = 0
    if ghost_distance == 2:
        penalty -= 100
    elif ghost_distance == 1:
        penalty -= 1000
    elif ghost_distance == 0:
        penalty -= 10000
    return -food_distance + penalty
```

the evaluate_func function only cares about the nearest food. If the distance between ghost and pacman, the function will get a penalty.

```
def determine_direction_and_wisely_choose_for_pacman(layout:
list[list[str]], row: int, col: int) -> str:
```

it will use the evaluate_func return values as the score and choose the max score. If there are more than one max value, choose one randomly based on their frequency. **NOT CHOOSE THE FIRST ONE**. If some results come up often, it means that they have a high frequencies than other, and it will converge more quickly.

Problem 3

This problem only has to care for the situation which is when two ghosts meet, then the moving ghost just stays in the same place as before.

problem 4

NO.	NUM TRAILS	TIME(SECONDS)	WINNING RATE
1	100	0.19561219215393066	43.0%
2	100	0.0070858001708984375	66.0%
3	100	0.010499238967895508	25.0%
4	100	3.111083745956421	98.0%
5	100	0.026218891143798828	26.0%
6	100	0.045494794845581055	36.0%
7	100	0.03174185752868652	29.0%
8	100	107.43443512916565	66.0%
9	100	101.733882188797	56.0%

Core function

```python
def evaluate_func(p_r, p_c, w_rs, w_cs, food_r, food_c):
    """
    :param p_r: pacman row index
    :param p_c: pacman column index
    :param w_rs: all of ghosts row indice
    :param w_cs: all of ghosts column indice
    :param food_r: food row index
    :param food_c: food column index
    :return: a score

     score consists of a penalty and the distance from food.
    if the distance between ghost and pacman is so close, there will
be a penalty. otherwise, pacman always finds
    the closest path to the nearest food
    """
    penalty = 0

    food_distance = get_distance(p_r, p_c, food_r, food_c)
    ghosts_distances = []
    for w_r, w_c in zip(w_rs, w_cs):
        ghosts_distances.append(get_distance(p_r, p_c, w_r, w_c))
    ghosts_distances = [min(ghosts_distances)]
    for ghost_distance in ghosts_distances:
        if ghost_distance == 2:
            penalty -= 100
        elif ghost_distance == 1:
            penalty -= 1000
        elif ghost_distance == 0:
            penalty -= 10000
        # if ghost_distance < 2:
        #     penalty-=1000
    return penalty - food_distance

@functools.lru_cache(maxsize=2 ** 11)
```

```python
def get_distance(x1, y1, x2, y2):
    return abs(x1 - x2) + abs(y1 - y2)
```

these two functions are the same in **problem 4**. The strategy is find the nearest food and makes sure that the nearest ghost has 2 and more step away from the pacman. If not, the score will have a penalty. What's more, using the distance from pacman to the nearest ghost represents the distance between pacmand the ghost group.

problem 5

ATTENTION: k in here means the same moves for everyone. For example: if k==2, which means that pacman moves 2 steps and the ghost moves 2 steps.

NO.	K	NUM TRAILS	TIME(SECONDS)	WINNING RATE
1	1	100	0.03359508514404297	100%
2	1	100	0.006026029586791992	100%
3	4	100	0.04183387756347656	99%
4	3	100	0.07683038711547852	95%
5	6	100	2.8372292518615723	98%
6	1	100	0.07768011093139648	100%
7	1	100	1.1187121868133545	100%
8	2	100	162.4913489818573	93.0

Core funtion

```python
# it is used for caching the expectimax function parameter
expectimax_cache = {}
```

the reason why I didn't use @functools.lru_cache is that I use factory function/closure to implement my thoughts and is hard to get the some data outside the function.

```
def expectimax(level, p_r, p_c, w_r, w_c, food_r, food_c) -> int:
    """
    :param level: the current height of expectimax
    :param p_r: the pacman's row index
    :param p_c: the pacman's column index
    :param w_r: the ghost's row index
    :param w_c: the pacman's column index
    :param food_r: the food's row index
    :param food_c: the food's column index
    :return: a score
    """
    cache_key = (level, p_r, p_c, w_r, w_c, food_r, food_c)
    if cache_key in expectimax_cache:
        return expectimax_cache[cache_key]
    # the ghost meets pacman
    if get_distance(p_r, p_c, w_r, w_c) == 0:
        utility = -2000
        expectimax_cache[cache_key] = utility
        return utility
    # get the expectimax leaves
    if level == 0:
        utility = -get_distance(p_r, p_c, food_r, food_c)
        expectimax_cache[cache_key] = utility
        return utility
    if level % 2 == 1:  # Maximizer Node (Pacman's turn)
        max_utility = -float('inf')
        for direction, (delta_x, delta_y) in ALL_DIRECTIONS.items():
            new_x, new_y = p_r + delta_x, p_c + delta_y
            if is_valid_move(new_x, new_y):
                penalty = 0
                distance_p_w = get_distance(new_x, new_y, w_r, w_c)
```

```
                if distance_p_w == 2:
                    penalty = -20
                elif distance_p_w == 1:
                    penalty = -200
                elif distance_p_w == 0:
                    penalty = -2000
                utility = expectimax(level - 1, new_x, new_y, w_r,
w_c, food_r, food_c) + penalty
                max_utility = max(max_utility, utility)
        expectimax_cache[cache_key] = max_utility
        return max_utility
    else:  # Chance Node (Ghost's turn)
        total_utility = 0
        num_moves = 0
        for direction, (delta_x, delta_y) in ALL_DIRECTIONS.items():
            new_x, new_y = w_r + delta_x, w_c + delta_y
            if is_valid_move(new_x, new_y):
                utility = expectimax(level - 1, p_r, p_c, new_x,
new_y, food_r, food_c)
                total_utility += utility
                num_moves += 1
        utility = total_utility / num_moves if num_moves > 0 else 0
        expectimax_cache[cache_key] = utility
        return utility
```

The implementation of expectimax is based on DFS. The chance node is take the average utilities and the maximizer layer also considers the penalty of closing to the ghost.

problem 6

ATTENTION: k in here means the same moves for everyone. For example: if k==2, which means that pacman moves 2 steps and every ghost moves 2 steps.

NO.	K	NUM TRAILS	TIME(SECONDS)	WINNING RATE
1	1	100	17.177525997161865	51.0%
2	1	100	0.012510061264038086	66.0%
3	1	100	0.019810914993286133	25.0%
4	1	10	44.432141065597534	100%
5	1	100	0.15531325340270996	26%
6	1	100	0.906822919845581	41%
7	1	100	0.25983381271362305	28%
8	1	10	183.2962248325348	50%
9	1	10	391.8677968978882	50.0%

Core function

It uses the same cache mechanism. the different thing is that expectimax is not cached. I try to do that but I found lots of hash collisions.

In the chance layer, new utility will consider the average of old utilities. In leaves node, the utility also will consider the average of distance from all ghosts.

```
def expectimax(level, p_r, p_c, w_rs, w_cs, food_r, food_c):
    cache_key = (level, p_r, p_c, tuple(w_rs), tuple(w_cs), food_r,
food_c)
    if cache_key in expectimax_cache:
        return expectimax_cache[cache_key]
    # Calculate average distance from all ghosts
```

```
    avg_ghost_distance = sum([get_distance(p_r, p_c, w_r, w_c) for
w_r, w_c in zip(w_rs, w_cs)]) / len(w_rs)

    # If any ghost meets Pacman
    if any([get_distance(p_r, p_c, w_r, w_c) == 0 for w_r, w_c in
zip(w_rs, w_cs)]):
        utility = -2000
        expectimax_cache[cache_key] = utility - avg_ghost_distance
        return utility

    if level == 0:
        utility = -get_distance(p_r, p_c, food_r, food_c)
        expectimax_cache[cache_key] = utility
        return utility

    # pacman's move
    if level % (number_of_ghost + 1) == 0:
        max_utility = -float('inf')
        for direction, (delta_x, delta_y) in ALL_DIRECTIONS.items():
            new_x, new_y = p_r + delta_x, p_c + delta_y
            if is_valid_move(new_x, new_y):
                min_ = min([get_distance(new_x, new_y, w_r, w_c) for
w_r, w_c in zip(w_rs, w_cs)])
                penalty = 0
                if min_ <= 2:
                    penalty -= 20
                # if min_ == 2:
                #     penalty = -10
                # elif min_ == 1:
                #     penalty = -100
                # elif min_ == 0:
                #     penalty = -2000

                utility = expectimax(level - 1, new_x, new_y,
w_rs[:], w_cs[:], food_r, food_c) + penalty
                max_utility = max(max_utility, utility)
```

```
        expectimax_cache[cache_key] = max_utility
        return max_utility

    else:  # ghosts' moves
        ghost_idx = (level % (number_of_ghost + 1)) - 1
        w_r, w_c = w_rs[ghost_idx], w_cs[ghost_idx]

        total_utility = 0
        num_moves = 0
        for direction, (delta_x, delta_y) in ALL_DIRECTIONS.items():
            new_x, new_y = w_r + delta_x, w_c + delta_y
            if is_valid_move(new_x, new_y):
                new_w_rs = w_rs[:ghost_idx] + [new_x] +
w_rs[ghost_idx + 1:]
                new_w_cs = w_cs[:ghost_idx] + [new_y] +
w_cs[ghost_idx + 1:]
                utility = expectimax(level - 1, p_r, p_c, new_w_rs,
new_w_cs, food_r, food_c)
                total_utility += utility
                num_moves += 1

        utility = total_utility / num_moves if num_moves > 0 else 0
        expectimax_cache[cache_key] = utility
        return utility
```

p1: 3h

p2: 1h

p3: 2h

p4: 1h

p5:3h

p6:3h

after finishing, time for optimize the code: 6h

writing report: 1h

in total: 20h