Input Validation

Introduction

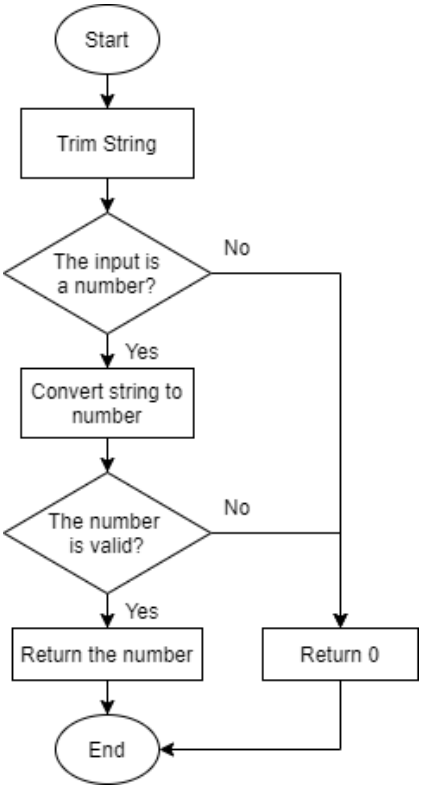
In the game, there are only two types of inputs required, which are the number of players and location of the mark placed. To avoid runtime errors caused by inputting non-numbers, the input is first read as a string using `fgets`. Then, the spaces in the front and end of the string are trimmed. Afterwards, `isdigit` is applied to check whether the processed string is a number. If yes, the string will be converted to number. For input number of players, the number is checked for whether it lies between 1 and 2. For input location of mark, the number is checked for whether it is within the range 1-9, and whether the grid is occupied. If the input cannot pass the above validation check, users will be required to input again until a valid input is obtained.

Implementation Details

Code segment for inputting string using a do-while loop that prompts user to input until a valid input is obtained. *(input number of players used as an example)*

```
do{
    fgets(input, 300, stdin);
    numOfHumanPlayers = input_number_of_players(input);
    if (numOfHumanPlayers != 0) break;
    printf("Invalid input! Please enter 1 or 2\n");
} while (1);
```

Flow chart and code segments for the validation function (*input mark location used as an example*)




```
// trim string
int length = strlen(c);
char s[300];
int st = 0, ed = length - 1;    //start and end location
while (c[st] == ' ') ++st;
while (!isalnum(c[ed])) --ed;
for (int i = st; i <= ed; ++i) s[i-st] = c[i];
// check if the string is number
for (int i = 0; i < strlen(s); ++i){
    if (!isdigit(s[i])) return 0;
    // 0 is a flag to indicate that the input is invalid
}
// convert string to integer
int res = 0;
for (int i = 0; i < strlen(s); ++i){
    res *= 10;
    res += (s[i] - '0');
}
// range check (1-9)
if (res < 1 || res > 9) return 0;
// check if the grid is empty
if (gameBoard[2-(res-1)/3][(res-1)%3] != 0) return 0;
else return res;    //validation check passed
```

Number validation code for input number of players (*line 284*)

```
// fixed value check
if (res == 1 || res == 2) return res;
else return 0;
```

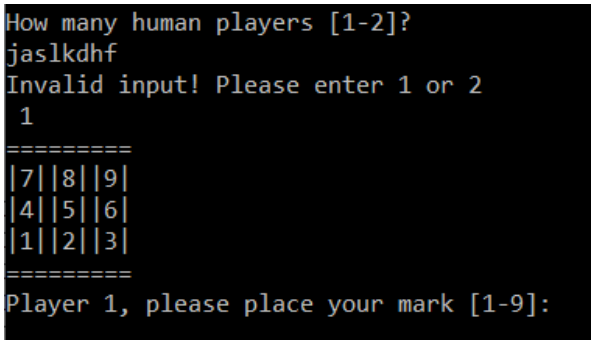
Relevant line numbers:

Input number of players: input (*line 305-314*), validation function `input_number_of_players` (*line 264-286*)

Input mark location: input (*line 231-239*), validation function `input_mark` (*line 202-226*)

Sample Runs

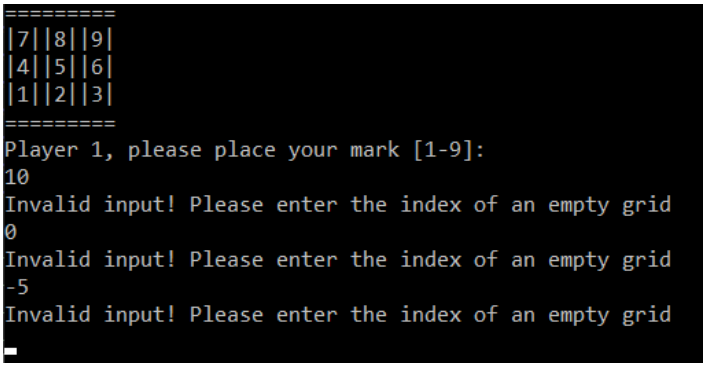
<table>
<tr><th>Input number of human players</th><th>Input mark location</th></tr>
<tr><td>



Input a meaningless string → asked to re-enter

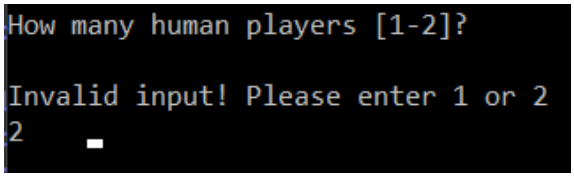
Input 1 with leading spaces → proceed to next step

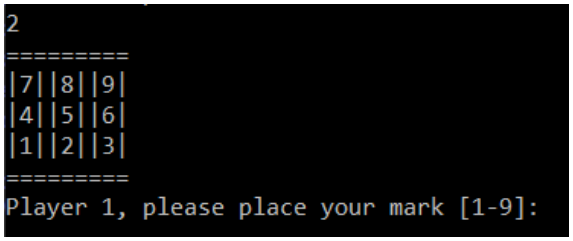
</td><td>



Input numbers out of range → asked to re-enter

</td></tr>
<tr><td>

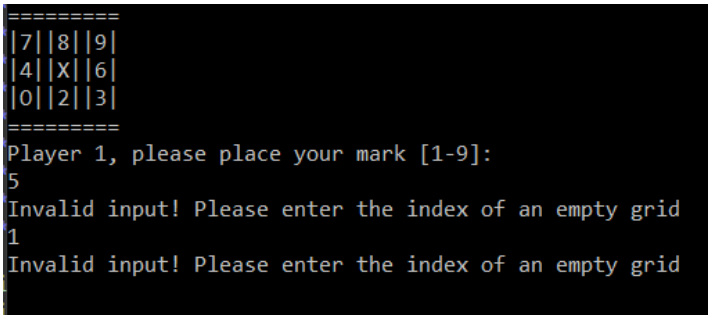




Input a blank line → asked to re-enter

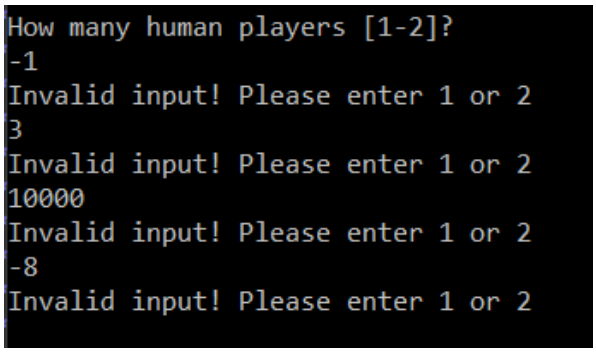
Input 2 with trailing spaces → proceed to next step

</td><td>

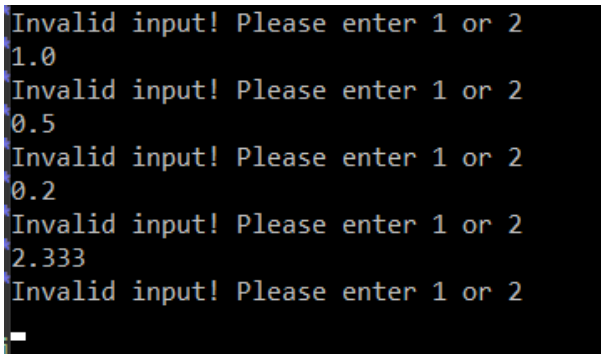


Input occupied grids → asked to re-enter

</td></tr>
<tr><td>

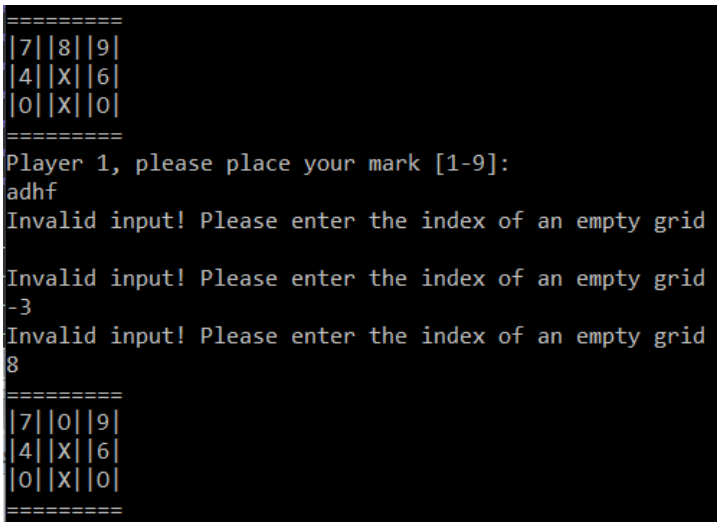


Input integers that are out of range → asked to re-enter



Input real numbers → asked to re-enter

</td><td>



Input random strings and blank line → asked to re-enter

Input a valid number 8 → pass the check and correctly places the mark at 8th grid

</td></tr>
</table>

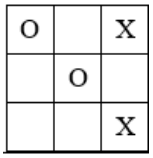
Module	Techniques Employed and Justifications
Input String	**`do-while` loop**: to prompt user to input until a valid input is obtained **`break`**: end the loop when a valid input is obtained **`fgets`**: to read the input as a line of string avoids runtime error that halts the program when users accidentally input non-digit character when `scanf("%d")` is used. `scanf("%s")` is not used as it does not read characters after the first blank space, so correct input with trailing spaces cannot be properly read.
Trim String	**`strlen`**: obtain length of string, learnt in string lecture **`while (c[st] == ' ') ++st`***(line 207)*: application of loop to find the first occurrence of non-space character in string `c` **`isalnum, isdigit`**: function in standard library to check if a character is alphanumeric and digit respectively **`return 0`**: make use of return value in a function as a flag to indicate result (failed the check)
Convert String to Integer	**`res += s[i] - '0'`**: use arithmetic calculations and ASCII character '0' to obtain the integral value of a digit from its character representation
Validate Integer	**`if (res < 1 || res > 9)`**: use if statements to perform range and fixed value check

Improved Computer AI Strategies

Introduction

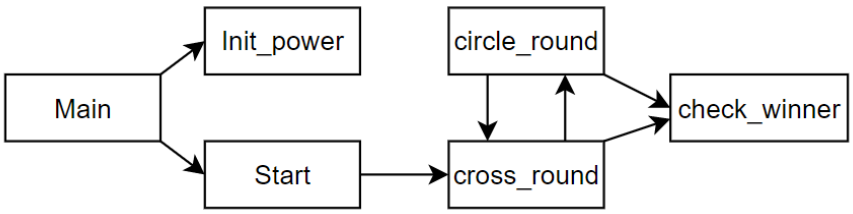
The feature uses recursion to run a depth-first search on all possible game state before the game starts. As it is known that tic-tac-toe has guaranteed non-losing method, the aim of the search would be finding such results, and record that for each possible game state, where should the next mark be placed to achieve such results. To allow easier implementation, I encoded the game state as a base-3 number instead of the 3x3 array. Hence, after the depth-first search, a 4*(AI moves at step 2,4,6,8)* x 20000*(3^9=19683)* array named `state` is generated. After game starts, the move stored in `state[step][game_state]` is the optimal move to be adopted by the AI. As the branches leading to losing are all cut in the depth-first search, it is guaranteed that the AI would not lose in any game.

Implementation Details

Major Variable, Struct and Data Structure Used
`int state[4][20000]`*(in main program)* / `s[4][20000]` *(in other functions)* `state[i][j]` stores the location *(0-8, ordered from left to right then up to down)* to place the next mark at computer's `(i+1)`th step with game state `j` *(as a base 10 representation of the state encoded in base-3)*.
`int p3[10]` + base-3 encoding game state (`int board`) `p3[i] = 3^i`, useful for encoding and decoding the base-3 game state. In the encoded game state, 0 represents empty, 1 represents circle, 2 represents cross. To add place a mark at xth grid, simply add p3[x] * [1*(circle)* or 2*(cross)*] to the game state. To remove a mark, the opposite can be done. For example, the grid at right is 102010002_3 = `p3[0]+p3[2]*2+p3[4]+p3[8]*2` = 13222_{10} In such case, the variable `board` will store the base-10 representation of the state, which is 13222. 
`struct result{ double win, lose, draw; };` *(line 34-36)* It is the data type returned by the two major recursive function (`circle_round` and `cross_round`). It stores the probability of winning and drawing under the current game state if all steps onwards done by AI is optimal. Hence, it helps to prioritize which step to choose under a game state. Note that the value of `lose` is always set to 1 if there is a possibility of losing (under the assumption that there exist non-losing strategy, any branch leading to possibly losing should be avoided).

Relationship between functions

(arrow from A to B means that function A calls function B)



Relevant Functions and Details

`void init_power(int p3[10])` *(line 171-177)*

Initialize array `p3` with `p3[x]` storing x^{th} power of 3. It facilitates the game state encoding and decoding process in other functions.

```
tmp = 1
for i from 0 to 9 do
      p3[i] = tmp
      tmp = tmp * 3
```

`void start(int s[4][20000], int p3[10])` *(line 164-169)*

Start of depth-first search. It generates all possible game states after first step and pass it to the recursive function `cross_round`.

```
for i from 0 to 8 do
      cross_round(s, 2, p3[i], p3)
```

`int check_winner(int board, int p3[10])` *(line 63-77)*

Convert the game state from base-3 encoded form to 3x3 gameboard array. Then use `hasWinner` function written in part 1 to check if the game has ended.

```
for i from 8 downto 0 do
      gameBoard[i/3][i%3] = board / p3[i]
      board = board % p3[i]
return hasWinner(gameBoard)
```

`struct result cross_round(int s[4][20000], int step, int board, int p3[10])`*(line 134-161)*

One of the major recursive functions in the depth-first search. At first, it would check if the game has ended. If yes, it means that the previous player (aka human player) won. Hence, a losing flag would be directly returned. Then, it would simulate all possible steps of computer player under the current game state, and recursively pass it to `circle_round(next step)` to compute the results to see how optimal the step is. Afterwards, it would select the best step and record it in `state` array. Finally, it returns the result if the best step is chosen.

```
// state array, xth step (2/4/6/8), encoded board state, p3 array
struct result cross_round(int s[4][20000], int step, int board, int p3[10]){
    // check if the game is ended at current state
    // if yes, it means that the previous player won, which is human player
    if (check_winner(board, p3)){
        struct result r = {0, 1, 0};    // result indicates losing
        return r;
    }
    // best stores the best result obtained out of all possible steps
    struct result tmp, best = {-1, 0, -1};
    int best_step;  // the mark location of the optimal step
    for (int i = 0; i < 9; ++i){ // exhaust all possible mark location
        if (board % p3[i+1] / p3[i] == 0){  //grid i is empty
            // search the next step to obtain the result of applying the mark
            tmp = circle_round(s, step + 1, board + p3[i] * 2, p3);
            // if this step is the best step until now, update best and best_step
            // (the best step means that losing is avoided, and the chance of winning is the highest)
            if (tmp.lose == 0 && tmp.win > best.win){
                best_step = i;
                best = tmp;
            }
        }
    }
    /* update state array: under this step and board status,
       placing the mark at best_step yields the best result */
    s[step/2-1][board] = best_step;
    //return the result when the optimal step is chosen
    return best;
}
```

`struct result circle_round(int s[4][20000],int step,int board,int p3[10])` *(line 80-131)*

Another major recursive functions in the depth-first search. At first, it would check if the game has ended. If yes, it means that the previous player (aka computer) won. Hence, a winning result would be directly returned.

```
// state array, xth step (3/5/7/9), encoded board state, p3 array
struct result circle_round(int s[4][20000], int step, int board, int p3[10]){
    if (check_winner(board, p3)){   //won after previous step, stop searching
        struct result r = {1, 0, 0};    // return "win result" to show that the previous step is optimal
        return r;
    }
```

It would simulate all possible steps of circle and recursively pass it to `cross_round(next step)` to compute the results if such step is chosen. Afterwards, it would calculate the probability of winning and draw of this game state based on the average of results of all possible steps.

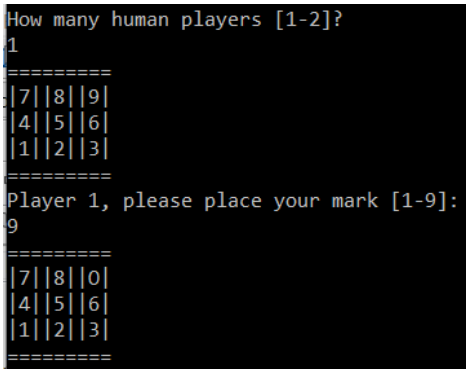
```
struct result tmp, r = {0, 0, 0};
for (int i = 0; i < 9; ++i){
    if (board % p3[i+1] / p3[i] == 0){  //grid i is empty
        tmp = cross_round(s, step + 1, board + p3[i], p3);
        /*
        as explained previously, any possibility of losing can be eliminated
        hence, if the probability of losing > 0, the step should never be chosen
        */
        if (tmp.lose > 0){
            tmp.win = 0;
            tmp.lose = 1;   // continue to return "losing result" to avoid this branch
            tmp.draw = 0;
            return tmp;
        }
        // else, compute the probability of win and draw respectively
        // (assuming an equal chance of choosing any empty grid)
        r.win += tmp.win / (9 - step);
        r.draw += tmp.draw / (9 - step);
    }
}
// return the results under the inputted game status
return r;
```

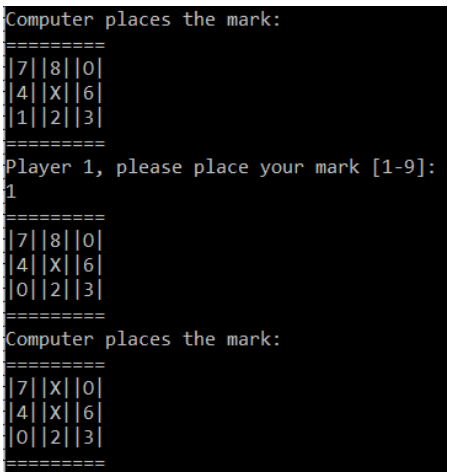
However, if the current step is step 9 (last possible step), it would simply simulate all steps and check if the result is lose or draw (win is not possible as last mover is human player). Note that it will not call other functions as it is the termination step of the recursion.

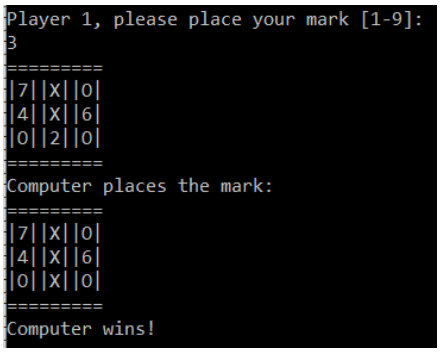
```
if (step == 9){ // terminal condition of the recursion
    struct result tmp = {0, 0, 1};  // preset draw state
    for (int i = 0; i < 9; ++i){
        if (board % p3[i+1] / p3[i] == 0){  //grid i is empty
            /*
                player 1 wins if the mark is placed here, this implies that the previous move
                is not optimal since it led to a possibility of losing
                under the assumption that there always exist a non-losing method, this
                branch should not be arrived, so "lose result" is returned
                in such case, the previous step would not be chosen
            */
            if (check_winner(board + p3[i], p3)){
                tmp.win = 0;
                tmp.lose = 1;
                tmp.draw = 0;
                return tmp;
            }
            break;
        }
    }
    return tmp; //since player 1 is the last mover, the game must draw if AI did not lose
}
```

Sample Run

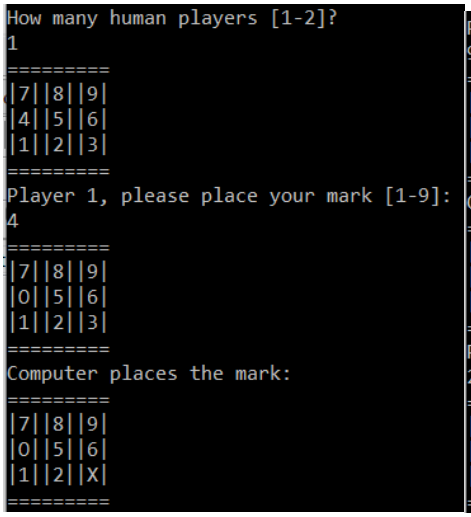
Simulate player who plays randomly

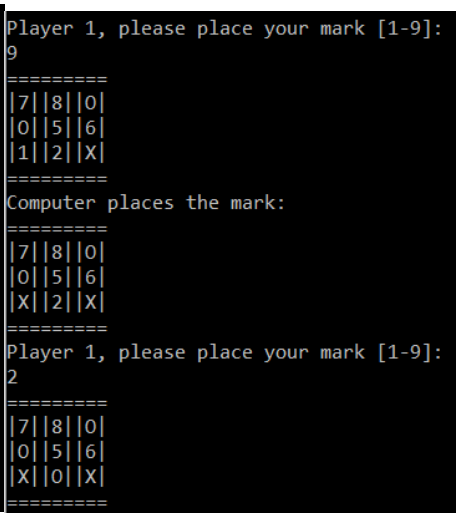


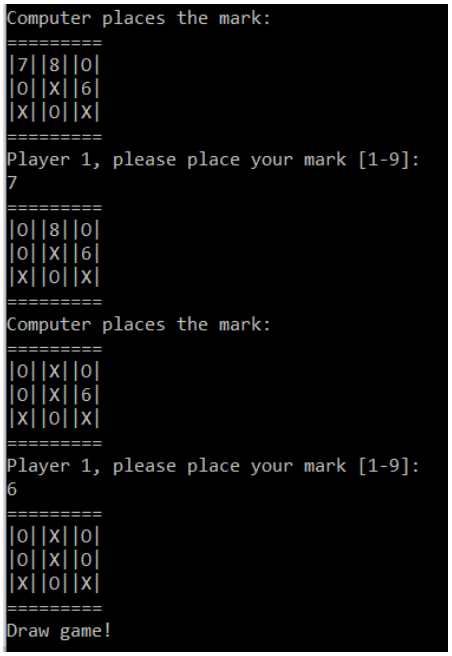




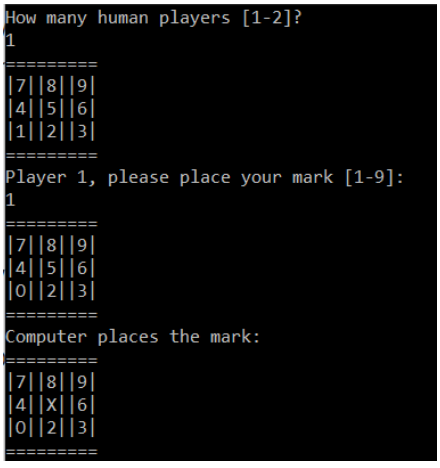
Simulate normal player who tends to select a “good” step, but not necessarily an optimal one

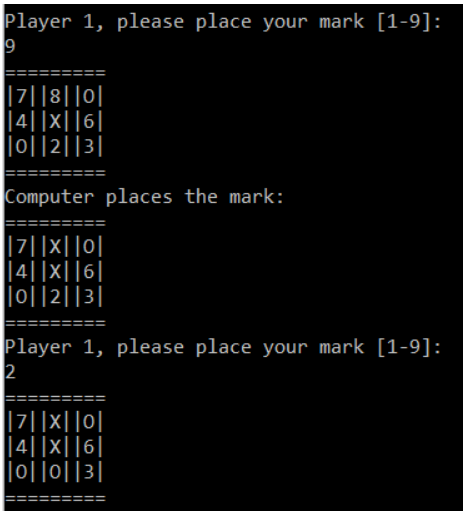


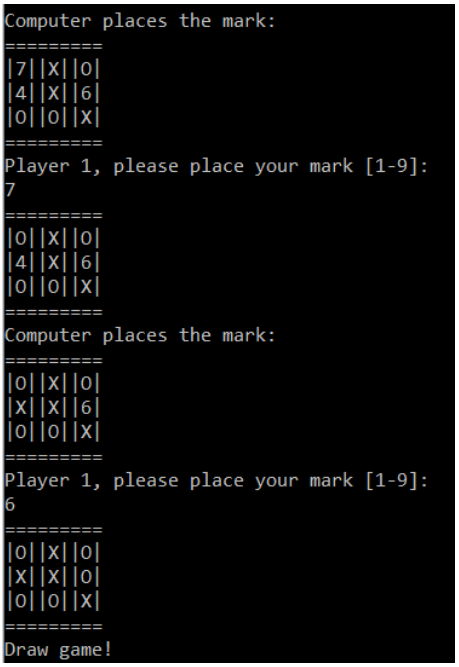




Simulate player who plays optimally







Techniques Employed
Depth-first Search with Recursion Since the total number of possible game states of tic-tac-toe is small (3^9 < 20000), it is possible to run a depth-first search that possibly exhausts all game states. Hence, a recursive function is implemented to do so. The recursion starts from step 1 of the game, then step 2, 3, …, until step 9. Afterwards, results are computed and returned in a bottom-up sequence from step 9 back to step 1. Depth-first search makes it possible to generate and select the best game state and step possible. Whereas recursion allows an elegant implementation of this searching process.
Pruning (剪枝) Pruning is used to avoid redundant computation and speed up the searching process. At any step, when it is found out that one of the outcome is losing, this step's result is immediately returned as "lose" and remaining searches in this step will stop. This can avoid searching in branches that are abandoned and therefore become irrelevant.
Dynamic Programming The recursion determines where to place the mark based on the results computed by subsequent searches. For example, results of step 7 is based on step 8, while that of step 8 is based on step 9. Such approach of computing results based on previous results is the core idea of dyanmic programming.
Base-3 Encoding Method This method is derived from the commonly used base-2 encoding method that represents different states using a 01 sequences. Although base-3 encoding is computationally slower than base-2 without bitwise functions (& | ^) available, it still allows a more convenient representation of game state compared to 3x3 array.
Struct `struct result{ double win, lose, draw; };` Struct is used to represent the results of each step in the depth-first search.
Lookup Table A lookup table for the powers of 3 (`int p3[10]`) is created at the start of the program. It allows a `O(1)` retrival of xth power of 3, which is faster than the retrival of `O(log x)` if the `pow(3,x)` function is used. Since the values are retrived for many times in the recursion, the lookup table helps reduce redundant computation and shorten program run time.