IMPLEMENTATION:

The highest-priority and the lowest-priority rules seemed obvious to me right away. The highest-priority are these:

1. If I can win on this move, do it.

2. If the other player can win on the next move, block that winning square.

Here are the lowest-priority rules, used only if there is nothing suggested more strongly by the board position:

n-2. Take the center square if it's free.

n-1. Take a corner square if one is free.

n. Take whatever is available.

The highest priority rules are the ones dealing with the most urgent situations: either I or my opponent can win on the next move. The lowest priority ones deal with the least urgent situations, in which there is nothing special about the moves already made to guide me.

What was harder was to find the rules in between.I knew that the goal of my own tic-tac-toe strategy was to set up a fork, a board position in which I have two winning moves,

so my opponent can only block one of them.

Here is an example:

x	o	
	x	
x		o

X can win by playing in square 3 or square 4.It’s O’s turn,

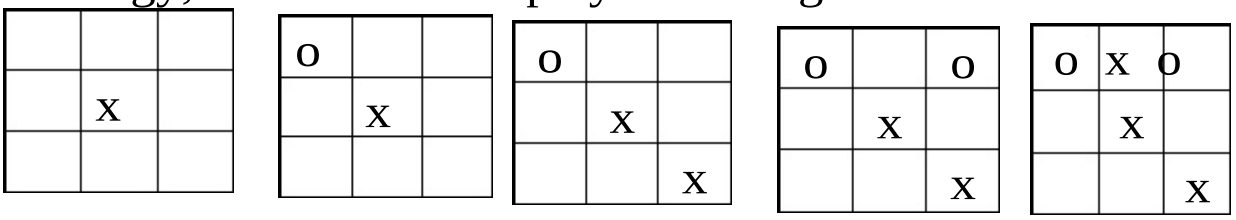
but poor O can only block one of those squares at a time. Whichever O picks, X will then win by picking the other one. Given this concept of forking, I decided to use it as the next highest priority rule:

3.If I can make a move that will set up a fork for myself, do it.That was the end of the easy part. My first attempt at writing the program used only these six rules.

Unfortunately, it lost in many different situations. I needed to add something, but I had trouble finding a good rule to add. My first idea was that rule 4 should be the defensive equivalent of rule 3, just as rule 2 is the defensive equivalent of rule 1:

4a. If, on the next move, my opponent can set up a fork,block that possibility by moving into the square that is common to his two winning combinations.

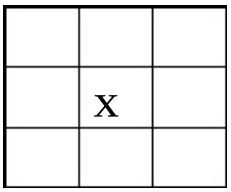
In other words, apply the same search technique to the opponent's position that I applied to my own.This strategy works well in many cases, but not all. For example, here is a sequence of moves under this strategy,with the human player moving first:



In the fourth grid, the computer (playing O) has discovered that X can set up a fork by moving in square 6, between the winning combinations 456and 369. The computer moves to block this fork.

Unfortunately,X can also set up a fork by moving in squares 3, 7, or 8. The computer's move in square 6 has blocked one combination of the square-3 fork,but X can still set up the other two. In the fifth grid, X has moved in square 8. This sets up the winning combinations 258 and 789.The computer can only block one of these, and X will win on the next move.

Since X has so many forks available, does this mean that the game was already hopeless before O moved in square 6? No. Here is something O could have done:



o		

o		
	x	
		x

o		
	x	
o		x

o		
x	x	
o		x

o		
x	x	o
o		x

In this sequence, the computer's second move is in square 7. This move also blocks afork, but it wasn't chosen for that reason. Instead, it was chosen*to force X's next move* . In the fifth grid, X has had to move in square 4, to prevent an immediate win by O. The advantage of this situation for O is that square 4 was not one of the ones with which X could set up a fork. O's next move, in the sixth grid, is also forced. But by then the board is too crowded for either player to force a win; the game ends in a tie, as usual.

This analysis suggests a different choice for an intermediate-level strategy rule, taking the offensive:

4b.If I can make a move that will set up a winning combination for myself, do it. Compared to my earlier try,this rule has the benefit of simplicity. It's much easier

for the program to look for a single winning combination than for a fork, which is two such combinations with a common square.

Unfortunately, this simple rule isn't quite good enough. In the example just above,the computer found the winning combination in which it already had square 1,and the other two were free. But why should it choose to move in square 7 rather than square 4? If the program did choose square 4, then X's move would still be forced, into square 7.

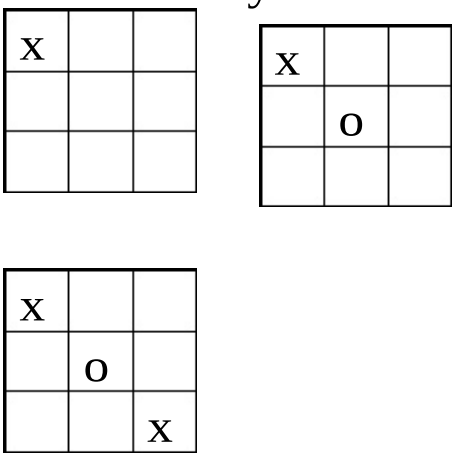
We would then have forced X into creating a fork, which would defeat the program on the next move.

It seems that there is no choice but to combine the ideas from rules 4a and 4b:

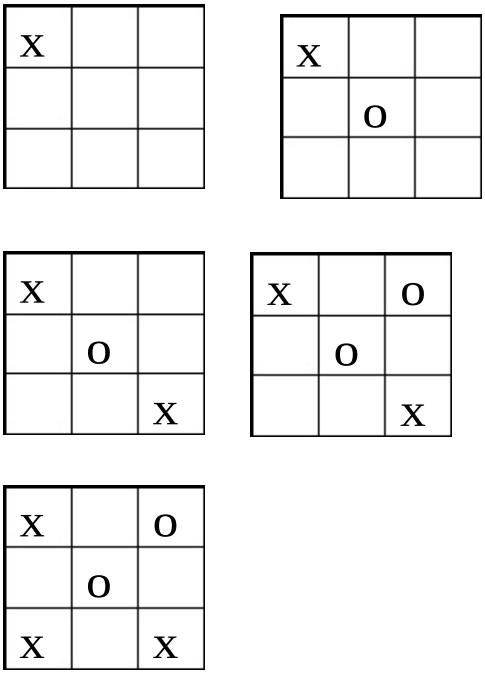
4.If I can make a move that will set up a winning combination for myself, do it. But ensure that this move does not force the opponent into establishing a fork. What this means is that we are looking for a winning combination in which the computer already owns one square and the other two are empty. Having found such a combination, we can move in either of its empty squares. Whichever we choose, the opponent will be forced to choose the other one on the next move. If one of the two empty squares would create a fork for the opponent, then the computer must choose that square and leave the other for the opponent.

What if both of the empty squares in the combination we find would make forks for the opponent? In that case,

we’ve chosen a bad winning combination. It turns out that there is only one situation in which this can happen:



Again, the computer is playing O. After the third grid, it is looking for a possible winning combination for itself. There are three possibilities: 258,357 and 456. So far we have not given the computer any reason to prefer one over another. But here is what happens if the program happens to choose 357:



By this choice, the computer has forced its opponent into a fork that will win the game for the opponent. If the computer chooses either of the other two possible winning combinations, the game ends in a tie. (All moves after this choice turn out to be forced.)

This particular game sequence was very troublesome for me because it goes against most of the rules I had chosen earlier. For one thing, the correct choice for the program is any edge square, while the corner squares must be avoided. This is the opposite of the usual priority. Another point is that this situation contradicts rule 4a (prevent forks for the other player) even more sharply than the example we considered earlier. In that example, rule 4a wasn't enough guidance to ensure a correct choice, but the correct choice was at least with the rule. That is, just blocking a fork isn't enough, but threatening a win and blocking a fork is better than just threatening a win alone. This is the meaning of rule 4. But in this new situation, the corner square (the move we have to avoid) block a fork, while the edge square (the correct move) block a fork!

When I discovered this anomalous case, I was ready to give up on the idea of beautiful, general rules. I almost decided to build into the program a special check for this precise board configuration. That would have been pretty ugly, I think. But a shift in viewpoint makes this case easier to understand: What the program must do is force

the other player's move, and force it in a way that helps the computer win. If one possible winning combination doesn't allow us to meet these conditions, the program should try another combination. My mistake was to think either about forcing alone (rule 4b) or about the opponent's forks alone (rule 4a).

As it turns out, the boar d situation we've been considering is the only one in which a possible winning combination could include two possible forks for the opponent. What's more, in this board situation, it's a diagonal combination that gets us in trouble, while a horizontal or vertical combination is always okay. Therefore, I was able to implement rule 4 in a way that only considers one possible winning combination by setting up the program's data structures so that diagonal combinations are the last to be chosen. This trick makes the program's design less than obvious from reading the actual program, but it does save the program some effort.

PROGRAM FOR TIC TAC TOE GAME :

```
#include <stdio.h>
#include <stdlib.h>
char matrix[3][3];
char check(void);
void init_matrix(void);
void get_player_move(void);
void get_computer_move(void);
void disp_matrix(void);
int main(void)
{
char done;
printf("This is the game of Tic Tac Toe.\n");
printf("You will be playing against the computer.\n");
done = ' ';
init_matrix();
do {
disp_matrix();
get_player_move();
done = check();
```

```
if(done!= ' ') break;
get_computer_move();
done = check();
}
while(done== ' ');
if(done=='X')
printf("You won!\n");
else printf("I won!!!!\n");
disp_matrix();
return 0;
}
void init_matrix(void)
{
int i, j;
for(i=0; i<3; i++)
for(j=0; j<3; j++) matrix[i][j] = ' ';
}
void get_player_move(void)
{
int x, y;
printf("Enter X,Y coordinates for your move: ");
scanf("%d%*c%d", &x, &y);
x--; y--;
if(matrix[x][y]!= ' '){
printf("Invalid move, try again.\n");
get_player_move();
}
else matrix[x][y] = 'X';
```

```
}
void get_computer_move(void)
{
int i, j;
for(i=0; i<3; i++){
for(j=0; j<3; j++)
if(matrix[i][j]==' ') break;
if(matrix[i][j]==' ') break;
}
if(i*j==9)  {
printf("draw\n");
exit(0);
}
else
matrix[i][j] = 'O';
}
void disp_matrix(void)
{
int t;
for(t=0; t<3; t++) {
printf(" %c | %c | %c ",matrix[t][0],
matrix[t][1], matrix [t][2]);
if(t!=2) printf("\n---|---|---\n");
}
printf("\n");
}
char check(void)
{
```

```
int i;
for(i=0; i<3; i++)
if(matrix[i][0]==matrix[i][1] &&
matrix[i][0]==matrix[i][2]) return matrix[i][0];
for(i=0; i<3; i++) /* check columns */
if(matrix[0][i]==matrix[1][i] &&
matrix[0][i]==matrix[2][i]) return matrix[0][i];
if(matrix[0][0]==matrix[1][1] &&
matrix[1][1]==matrix[2][2])
return matrix[0][0];
if(matrix[0][2]==matrix[1][1] &&
matrix[1][1]==matrix[2][0])
return matrix[0][2];
getch();
return ' ';}
```