Our game went through three major phases. These are outlined below:

**FIRST PHASE**:

The main objective was to win the game by exhausting all the possible moves using **backtracking**. We used a **deterministic** algorithm which was implemented using **iteration** rather than recursion because it was much more **efficient in terms of both memory and computations.** In fact, it was the **fastest program** – the first one able to completely exhaust all the moves to find a winning move right at the start of the game. We won all the games in the first assignment except the finals.

**Data structure for representing the coins and finding the valid moves:**

** To indicate the presence of coins we used a Boolean array of size 21.

** To find the validity of the moves we used the following numbering technique

```
                11
              12   22
            13   23   33
          14   24   34   44
        15   25   35   45   55
      16   26   36   46   56   66
```

The difference between any two consecutive coins that can be picked is one among
1, -1, 10, -10, 11, -11
To find whether a given move is valid we have to sort the coins, find the differences according to the numbering scheme and check whether all the differences are mutually equal and one of the above.

To arrive at the numbering scheme, each number is represented as a two digit number with the first digit signifying the row number (starting from 1) and the second digit, the column number (again starting from 1).

** We also figured that rather than regenerating the entire set of possible moves at each stage, it should be generated once when the program starts. And, to test the moves, we use a temporary array to store the move being considered. Thus we have a **highly efficient program in terms of memory and computations.**

**SECOND PHASE:**

In this phase, we changed our program from an **iterative one to a recursive one.** Using pure recursion alone, it is not possible to exhaust all the moves. In order to have a **deterministic algorithm** that could exhaust all the moves right from the beginning of the game, we employed the following techniques:

** **Optimal substructure:** the game has an optimal substructure property so we used memoization.

** **Memoization** (we solve sub-problems only once and store the results) The results are stored in an array – each entry of which could either be representing a winning position, a losing position or a position yet to be evaluated.

** **Hashing** to store the results and access them (the fastest way to do it). Our hash function used an array of length 2^21.
Each position was represented by a binary number using a Boolean array of length 21. The Boolean value depended on whether the corresponding coin had been picked up or not.
The index was arrived at by simply adding those powers of two which had unpicked coins. This is **very efficient in terms of computational requirements.**

** **Learning algorithm:** the program learns from each game, giving faster outputs each time. The whole array is stored in a file so as to enable the program to use previously evaluated moves. After once exhausting the set in a game, our program took only about 15-20 seconds to output a winning move.

** **Heuristics:** we chose to pick a single coin when we don't have a strategy for a sure win. This seemed to be a good heuristic for maximizing the score.

** Playing first, the winning move 4, 7 was made.

**THIRD PHASE:**

In the last phase, we decided to change our strategy from a completely deterministic one to one that could change itself based on the point scheme. We have tried to draw on the idea of ANNs.

**Motive:**

In each of the assignments a different scoring system was implemented and more importantly, the number points for winning varied drastically. If the number of points for winning the game was large (as was the case in the first two assignments), then a purely recursive game would succeed. On the other hand if the number of points for winning a game was less (as was the case in the third assignment), then a good strategy would probably be a greedy choice.
These were easy decisions to make. But on a later date, the points scheme could be different and might even vary dynamically (from game to game).
We would then have to come up with a good set of heuristics. But heuristics generally means a lot of research and an insight into the domain knowledge.
To overcome this effort we designed a game that can design the heuristics by itself by playing many games. The detailed approach is given below:

Our program seeks to find the optimal value of parameter N, where N represents the number of coins at which the program should start trying to find a wining move.
If the number of coins is greater than N, the program makes a greedy choice.
In order to train the program, we first play 16 trial games – N is given values 0, 10, 11, 12, 13, 14, 15, 36. For each game played, we store N, the points gained by our program, the opponent's program and the number of games played with the same value of N.
After two sets of trials (i.e. the first round), our program picks that value of N for which our points (weighted average) was maximum. Thus, after playing a sufficiently large number of games, it should be able to figure out the best stage to switch its approach from greedy to one aimed at winning. Of course, if too many games are played with a particular point scheme and then the scheme is changed, the learning will be slow because we are taking the weighted average of the scores – you cannot teach an old dog new tricks!
This is in keeping with the natural learning curve - when the program is still young, it learns fast and stabilizes.

**Group members:**

**#01010119 Rohan Sasanka**
**#01010121 Sameer Narang**
**#01010134 Vipin Puri**