Strategy File for Group No. 9

Members:

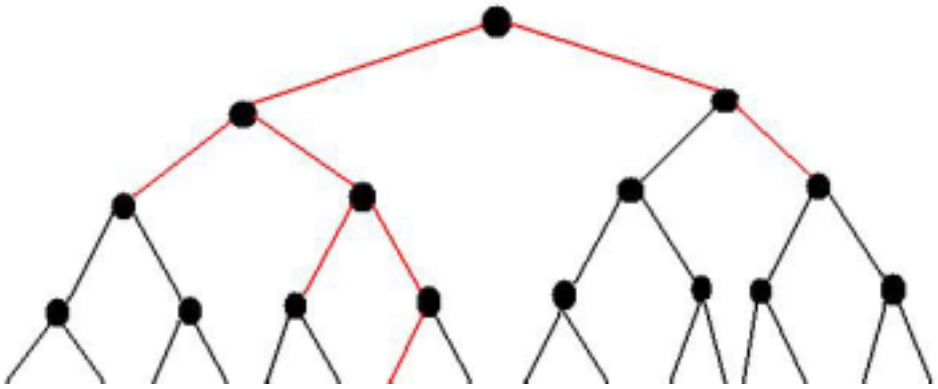
Tarang Vaish	01010131
Vikash Kumar	01010132
Vinod Rao	01010133

Program 1

Being the first program a lot of thinking went into, what the strategy should be? And What kind of processing has to be done by the program?
A recursive solution was thus thought of which was very immature but formed the basis of this deterministic approach for a Sure Shot Win.

The initial Algorithm was a **Recursive algorithm** which could look forward by picking Coins again and again until a point came (**coins left <=4**) when the program knows for sure that it is going to win or not. Thus the recursion followed **Backtracking** where it kept on playing moves until a move comes when the opponent can Never Win(that is whatever move the opponent plays I can still win).

Backtracking



Algorithm

1. Start
2. Play default moves of picking 1-6 coins that are available until coins left are less than a fixed value(around 8-10 coins)
3. Now recursively try to find if a Winning move is possible, otherwise play a random move. Keep on doing this until No coins are left
4. Stop

The moves list was generated each time by using those moves out of the total possible whose coins are present, also checking for valid moves was done by having groups of 2,3,4,5,6 coins separately and looking linearly through the arrays if the coins are present or not.

Running Time (Maximum 2minutes, minimum 1minute for 16-17 coins)

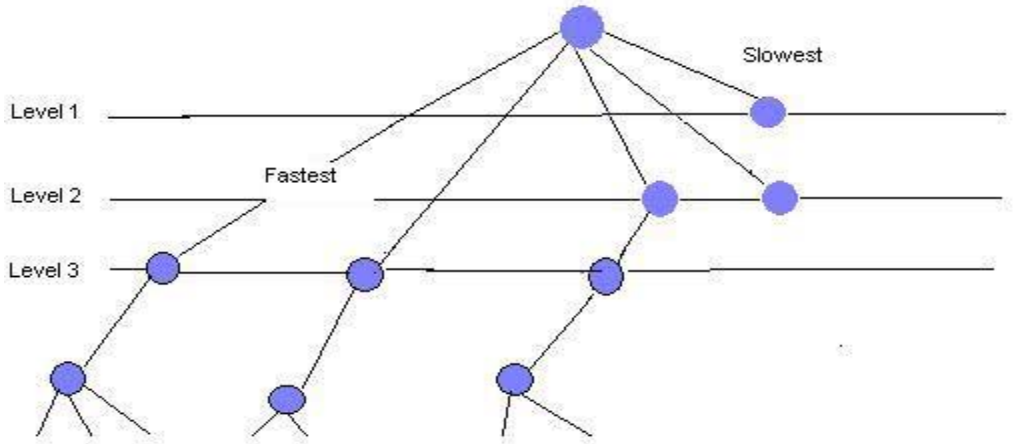
The recursion for N coins would have taken N! time and then the searching and updating moves used to take Constant time. Hence running time approached O(N!).

Program 2

This program had lot of tweaks and improvement in usage of data structures etc.
This time the moves were structured so that when they were generated they came in Increasing order of coins, and then the moves played in recursion started Backwards.

That is order the order in which moves were played was:

MV1 1, 3,6,10
2: 1, 3, 6
3: 3, 6, 10
4: 1, 3
5: 3, 6
…….

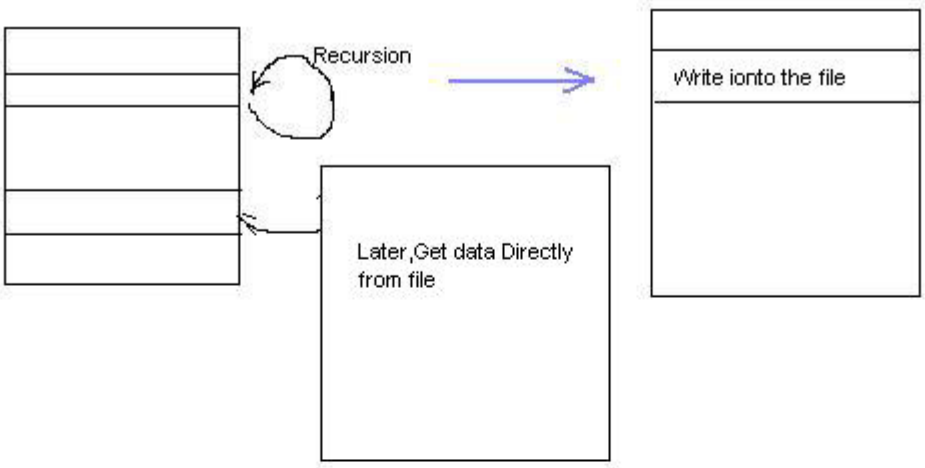


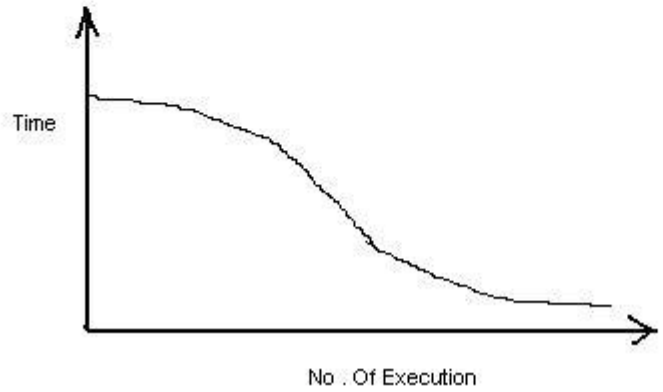
The advantage of this algorithm was that the "MORE likely" and FASTER moves were played before others. This way lot of time was saved and the program could start from thinking from more coins.

The checking of geometric shapes like for Pair of Coins now had Hashing in it.
All the coins near each coins were listed in an Adjacency list and then it could be accessed in O(1) time.

That is:
Coin 1: 2, 3
Coin 2: 1,3,4,5
Coins 3: 1,2,5,6
…..

Memoization





When coins were more than 10 a result of WIN or LOSS was stored into a file
As a structure having the fields as

Array containg 21 places to store 1 if coin (I) is present or 0 if not present.
WIN or loss result.

The searching for this board in the file was done using the **6 WAY symmetry of** the **TRIANGLE**, hence when a board was to be searched, it was rotated left, right, mirrored to get 6 arrays. Now searching was done in the file.

This allowed reducing the SIZE of the LOG file, and also COMPUTATION time for the boards symmetric to the stored ones to be read directly from the file.

For **Default Move** instead of playing Randomly or choosing the first move,
Move with more coins that is those of 3-4 coins were preferred as they had more **points**,
This was the reason for coming first in that game where others were almost equally matched.

Running Time

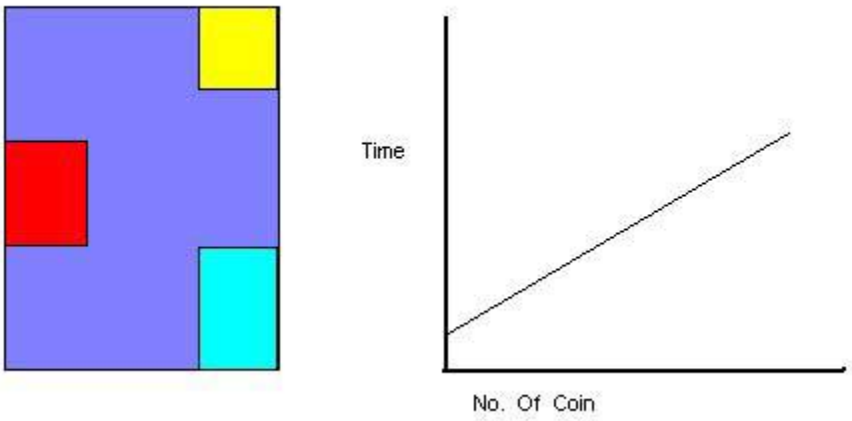
Maximum (2minutes, minimum 5 seconds)
The improvement in minimum time came because of playing moves in decreasing order. It was found that generally 3-6 coins picked up at coins>15 was a more probable winning move.

Time due to recursion would cost N!, but the linear factors like searching, updating moves made it faster, Memoization then was able to bring down a considerable time when many games were played, thus changing the running time to almost LINEAR as more and more games are played.

Program 3

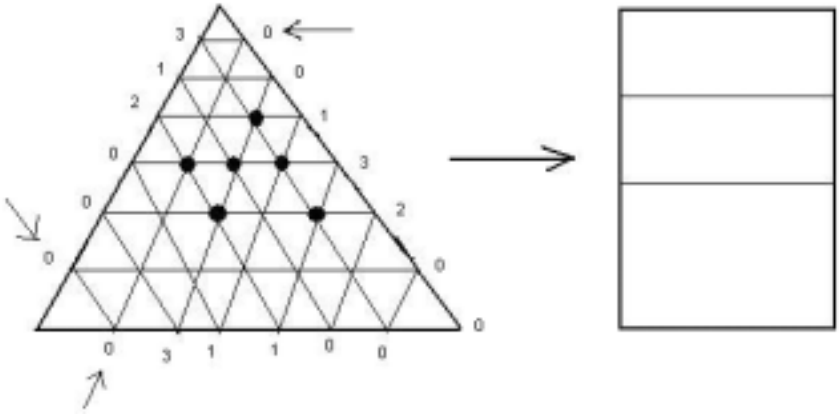
Again many tweaks were suggested on the previous algorithm, lot of redundant code was removed, and a new way for memoizing the board was deviced.

This method involved use of FLOOD FILL algorithm which could divide the board into small independent TREES, which could then be coded onto the file.
Flood Fill Alogorithm



These algorithms worked by COLORING the first uncolored coin found , and then COLOR all coins which are connected to this one using **Breadth First Search**. Then this colored tree is stored in some array and then again the board is searched for more trees.

The coding was a as follows,



For each tree the number of coins present in the tree in each horizontal line parallel to the base of the triangle was found, then the board was rotated once to get a new set of values, and again to get a third.

Thus the data structure contained an array which could store for each tree 6*3 places, where coins in each line were present.

The COST of memoization increased but now number of such memoizations greatly reduced as the **same set of trees placed anywhere and anyway rotated could be matched!.**

Running Time (Maximum 60 seconds minimum 1-2 second)

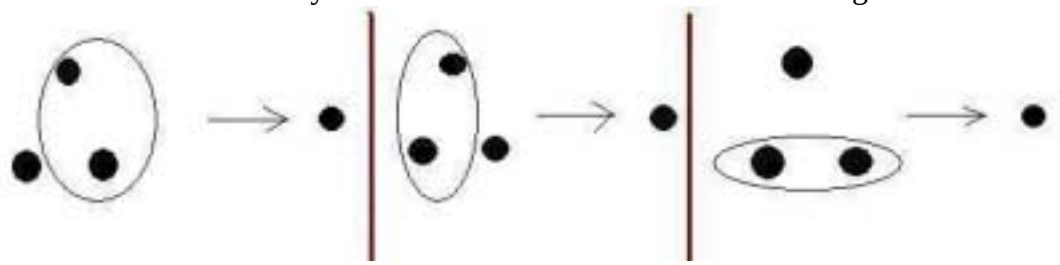
The running time was less than previous time in the long run as once a log file starts filling up, LOT of board configurations were getting actually coded.

Program 4

This game had a very **greedy** oriented approach for achieving more points, instead of looking at one step greedy the entire path to no coins was searched to find a path which has more points.
This game along with the algorithms of previous code had a lot of new algorithms and memorization techniques

The program had 2 levels of **ISOMORPHISM**

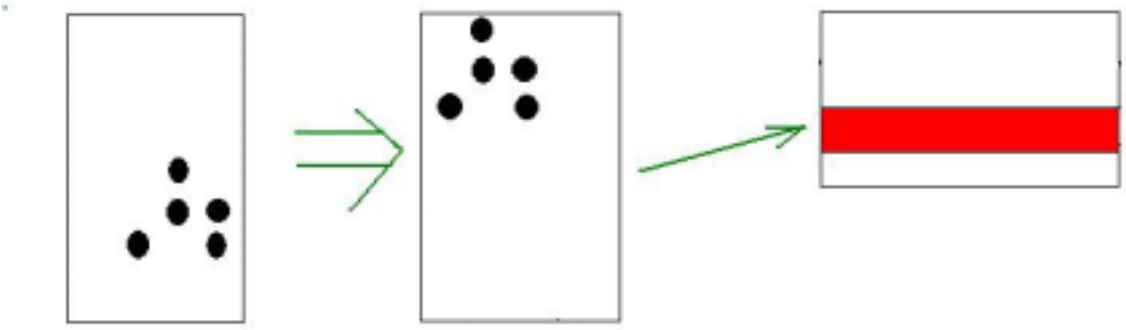
Firstly Trees of coins were formed using the FLOOD FILL ALGO. Then trees with 4 or less coins were searched for symmetric moves like one shown in the diagram:



It was seen that for coins less than 15 that lot of trees with 2-5 coins are present,
The initial strategy of the game was to Enhance this portioning by moving DIAGONAL moves, this increased portioning.
Now when the above technique is used the number of moves generally reduced by 5-10%
Now when we analyze a recursive algorithm where each step is reduced by 5-10%, the recursive time got reduced by 20-30%.

Second level of **ISOMORPHISM**

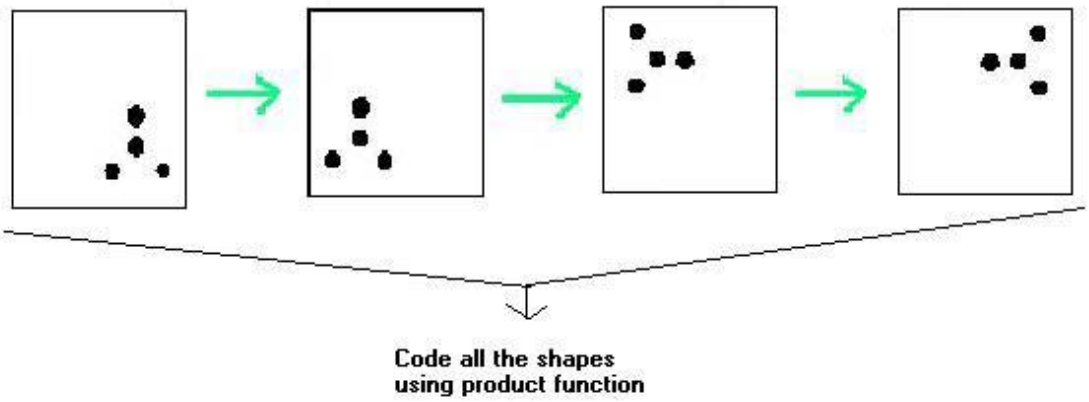
This method was able to code **ANY number** of coin trees on the board to a unique integer value.



The tree is first MOVED to the left top corner of the board and then a **PRODUCT OF ALL POSITIONS** where the coins are present, and then stored into the file.

The file contained a small structure having Number Of trees, Elements in each tree, Product of Each tree and WIN pts for that board.

For Comparing two boards trees with SAME number of coins is checked if they conform to the same product or not, along with this the **CODE of mirrored, left rotated, again rotated tree** is calculated and also compared, now if any of these products match to that of the file, the two trees are isomorphic.



The advantage being that now Two trees placed Anywhere and rotated in any direction will conform to the same product.
For eg: if we have 5 coins with 2,3 isolated trees then when we code the structures into products, We have memorized a result for all 5 coin boards having these 2,3 isolated trees places anywhere on the board.
This process can save around **90-95% of the memoization storage and search time.**

With the help of the above two algorithms Dynamic Programming was achieved.

Running Time (Maximum 120 seconds(first time), Minimum 1-2 seconds(for later times)

The O(N!) time was overcome my reducing the 'N' factor at each iteration by isomorphism.

Program 5

This program was attempted to improve upon the previous game.
When default initial moves were played the program was given DUMMY GAMES
In which it play virtually a second game taking around 30seconds, and then a default GREEDY move was played. The purpose of the above process was to increase memoization even before the recursion of 17 coins could start

The game plan was changed from point system to Winning move.
The moves in the beginning were updated dynamically until 17 coins arrived and then isomorphic moves were found for further steps.
An Alarm system was used to generate an interrupt when time taken was near 40 seconds This was done as memoization has a bottleneck of large time consumption when it is executed for the first time.On repeated execution the running time comes down and gives good moves.

Running Time (Maximium – 40 seconds, minimum <1 second)