CS 461  
Professor Brian Hare  
Program 1 – Coins Game  
Due 5 September 2017  
Scott Peery

For this report, we will ignore the exception handling throughout the code to include the exceptions in the AI strategy. None of the exception handling in the AI strategy affects the strategy of the algorithm, rather it was a quick fix to dealing with the index when all moves had been exhausted in the game.

I chose not to store the moves that the AI determined where the best (memorization) as I was not able to implement it correctly when I tried, and I did not want to use a search tree as that would take more time to build each round and search. Instead I told the AI to look forward from the current position of the game board (coins left) to find the moves that would allow it the highest score from the current state. This works reasonably well for longer coins board playing against a human that is trying to recalculate the remain moves as we are prone to mistakes.

In a large sample of coins, n, the AI does a very good job of making it a close game until it gets to the last 4 to 6 coins. Then it seems to lack the foresight to limit the players coin values. I think, given more time on the issue, I might have come up with a solution to this problem. It has escaped my resolve for the last several days on coming up with the solution.

The code for the strategy I implemented is as follows (I believe it is straight forward with my comments):

*# function to determine winning moves for AI***def** ai\_strategy(coin\_list, start, end):  
 *# import the global variable for valid move* **global** valid\_move  
 *# check that the end index is not less than the start index  
 # if it is, end game* **if** start > end:  
 **return** 0  
 *# define variables looking from start of list and end of list* front\_strategy = str(coin\_list[start])  
 back\_strategy = str(coin\_list[end])  
 *# for strategy taking coin from start of list* **try**:  
 *# look one move forward from left side of list* **if** coin\_list[start+1] < coin\_list[end]:  
 result\_front\_strategy = front\_strategy + str(ai\_strategy(coin\_list, start+1, end-1))  
 **else**:  
 result\_front\_strategy = front\_strategy + str(ai\_strategy(coin\_list, start+2, end))  
 *# handle the index error that occurs at end of game for last coin in list* **except** IndexError:  
 valid\_move = **"front"  
 return** valid\_move  
 *# for strategy taking coin from back of list* **try**:  
 *# look one move forward from the right side of the list* **if** coin\_list[start] < coin\_list[end-1]:  
 result\_back\_strategy = back\_strategy + str(ai\_strategy(coin\_list, start, end-2))  
 **else**:  
 result\_back\_strategy = back\_strategy + str(ai\_strategy(coin\_list, start+1, end-1))  
 *# handle the index error that occurs at end of game for last coin in list* **except** IndexError:  
 valid\_move = **"front"  
 return** valid\_move  
 *# compare each strategy and save the move that yields best result* **if** result\_front\_strategy > result\_back\_strategy:  
 game\_state[(start, end)] = result\_front\_strategy  
 valid\_move = **"back"  
 return** valid\_move  
 **else**:  
 game\_state[(start, end)] = result\_back\_strategy  
 valid\_move = **"front"  
 return** valid\_move

It is comparing the values of the coins after a move. For example, if the list is 10, 5, 20, 5, it is looking to see what choses the player can make with what is left, and is choosing the coin that will leave the higher coin for itself. The only problem with this setup is, since there are an even number of coins, I found it hard to get the AI to win in these type scenarios. The only time the AI did better, was with a longer list of coins.

The efficiency is of O()