Erik O. Kriz
Algorithms and Complexity
Full code can be found at bottom of report

"Chopsticks"

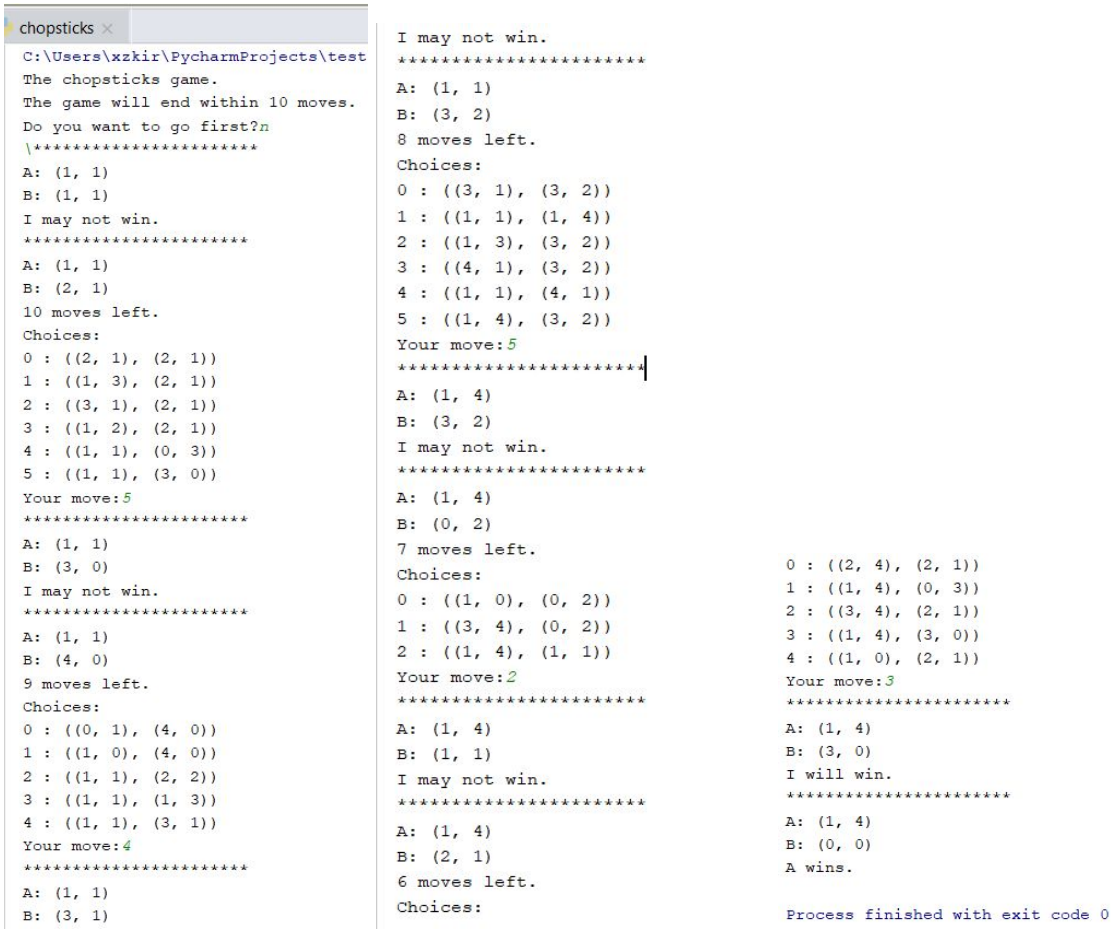
Description: The section of code I completed for this assignment is the best_move_dp() function. The point of this function is to create a game tree for 'chopsticks' using dynamic programming. So, first it creates the final dictionary that will be what we return, then it handles all of the base cases for the tree. In this case, our base cases/ leaves of the tree are when the depth is at its max. Since there are no more turns to take, we can assign a value of -1, 0, or 1 to each game state depending on if player B won, if there was a tie, or if player A won respectively. From there, I use dynamic programming to fill in the rest of the tree. In order to reduce total running time, I create a set that acts as a holder for all of the game states that I have checked and that exist in the dictionary. The following for loop is telling the code to work its way form one above the maximum depth, all the way to depth 0. What it is doing in each of those steps is filling out the game tree for each of those depths. In order to do this, I check every possible game state at that depth, there are 625 of them. First, I check the keys of the dictionary, using the set, to see if I already have computed this information. Then, at each state where I haven't, I check to see if it is a base case where there is a tie or someone has won, if it is, it gets a similar assignment to the original base cases. For those cases that do not match a base case, I use the next_moves() function to get a list of all possible game states after one move. Using this list I can determine what is the optimal move to make at the current state. I figure out the optimal move using a minmax type of function, this means that if the depth corresponds to player A, which is an even turn, then you will want to take the turn with the maximum value. If the depth is odd, then it would be player B's turn, which means you would want to take the move with the minimum possible value as that is the one that will detriment A the most. When I check to see the values for each move, I can assume that every single move that is at a lower depth has already been added to the dictionary. This is because I had to work my way from the bottom up, and if I am at depth x, then I have already worked depth x-1. Once I have discovered the optimal move, by taking the minimum or maximum of all possible moves depending, I insert that move into the dictionary for that specific game state. Once this has iterated through every depth, I have a completed game tree, and the game can continue.

Trade Offs: The most major trade off of my code was figuring out how to make the set work within the function. As I was writing the code, I realized that it could be sped up by adding each dictionary key that I create into a set, and check that set instead of the keys when checking to see if I have computed that key before.

Extensions: One extension that would really help how well this could play the game is by having much more descriptive values for each of the possible moves. For example, if you were to look at each possible move for a given game state, instead of just have -1s, 0s, and 1s, there could be a way to make it so the value of each of those moves is equal to the total number of -1 states that move leads to summed with the total number of +1 states that that move leads to, then picking the minimum or maximum of that sum. This would make it so you are not just picking the first, 1 or -1 in the sequence ou come across. You would instead be picking the move with gives you the highest probability of winning, since there are more winning states created by that move. This would probably not take that much to implement, just some tuning for the definition of a dictionary entry

Test Cases: For My test cases, I decided that the best way to find if the code worked was to play many games, both going first and second, to see if there was some sort of error. In the end, I chose to have 4 representative cases for my test cases.

The first test case that I tried was to copy exactly the path that the assignment made, to see if mine matched.



- The result was that it matched perfectly.

Need 2 more where I go first, and one more where the bot goes first.

- Here is one instance where I decide to go first

```
C:\Users\xzkir\PycharmProjects\test\venv\Scripts\python.exe "C:/Users/xzkir/Desktop/School/CSE 3500/Programming 3/chopsticks.py"
The chopsticks game.
The game will end within 10 moves.
Do you want to go first?y
**********
A: (1, 1)
B: (1, 1)
10 moves left.
Choices:
0 : ((1, 1), (2, 1))
1 : ((0, 2), (1, 1))
2 : ((1, 1), (1, 2))
3 : ((2, 0), (1, 1))
Your move:0
**********
A: (1, 1)
B: (2, 1)
I may not win.
**********
A: (2, 1)
B: (2, 1)
9 moves left.
Choices:
0 : ((2, 1), (2, 2))
1 : ((2, 1), (3, 1))
2 : ((2, 1), (4, 1))
3 : ((3, 0), (2, 1))
4 : ((2, 1), (2, 3))
5 : ((0, 3), (2, 1))
Your move:4
**********
A: (2, 1)
B: (2, 3)
I will win.
**********
A: (0, 1)
B: (2, 3)
8 moves left.
Choices:
0 : ((0, 1), (3, 3))
1 : ((0, 1), (2, 4))
Your move:0
```

```
**********
A: (0, 1)
B: (3, 3)
I will win.
**********
A: (0, 1)
B: (2, 4)
7 moves left.
Choices:
0 : ((0, 1), (2, 0))
1 : ((0, 1), (3, 4))
Your move:0
**********
A: (0, 1)
B: (2, 0)
I will win.
**********
A: (0, 1)
B: (1, 1)
6 moves left.
Choices:
0 : ((0, 1), (1, 2))
1 : ((0, 1), (2, 1))
Your move:0
**********
A: (0, 1)
B: (1, 2)
I will win.
**********
A: (0, 1)
B: (3, 0)
5 moves left.
Choices:
0 : ((0, 1), (4, 0))
Your move:0
**********
A: (0, 1)
B: (4, 0)
I will win.
**********
A: (0, 0)
B: (4, 0)
B wins.
```

- Here is another case where the program goes first. Here I also check to make sure that wrong answers are not a problem for the code

```
C:\Users\xzkir\PycharmProjects\test\venv\Scripts\python.exe "C:/Users/xzkir/Desktop/School/CSE 3500/Programming 3/chopsticks.py"
The chopsticks game.
The game will end within 10 moves.
Do you want to go first?n
**********
A: (1, 1)
B: (1, 1)
I may not win.
**********
A: (1, 1)
B: (2, 1)
10 moves left.
Choices:
0 : ((2, 1), (2, 1))
1 : ((1, 3), (2, 1))
2 : ((3, 1), (2, 1))
3 : ((1, 2), (2, 1))
4 : ((1, 1), (0, 3))
5 : ((1, 1), (3, 0))
Your move:10
Your move:12
Your move:20
Your move:4
**********
A: (1, 1)
B: (0, 3)
I may not win.
**********
A: (1, 1)
B: (0, 4)
9 moves left.
Choices:
0 : ((1, 1), (2, 2))
1 : ((1, 1), (1, 3))
2 : ((1, 0), (0, 4))
3 : ((1, 1), (3, 1))
4 : ((0, 1), (0, 4))
Your move:4
**********
A: (0, 1)
B: (0, 4)
I will win.
**********
A: (0, 1)
B: (0, 0)
A wins.
```

- Here, I have one more case where I go first

```
C:\Users\xzkir\PycharmProjects\test\venv\Scripts\python.exe "C:/Users/xzkir/Desktop/School/CSE 3500/Programming 3/chopsticks.py"
The chopsticks game.
The game will end within 10 moves.
Do you want to go first?y
**********
A: (1, 1)
B: (1, 1)
10 moves left.
Choices:
0 : ((1, 1), (2, 1))
1 : ((0, 2), (1, 1))
2 : ((1, 1), (1, 2))
3 : ((2, 0), (1, 1))
Your move:2
**********
A: (1, 1)
B: (1, 2)
I may not win.
**********
A: (1, 2)
B: (1, 2)
9 moves left.
Choices:
0 : ((1, 2), (1, 3))
1 : ((1, 2), (3, 2))
2 : ((1, 2), (1, 4))
3 : ((1, 2), (2, 2))
4 : ((0, 3), (1, 2))
5 : ((3, 0), (1, 2))
Your move:3
**********
A: (1, 2)
B: (2, 2)
I may not win.
**********
A: (3, 2)
B: (2, 2)
8 moves left.
Choices:
0 : ((3, 2), (2, 4))
1 : ((3, 2), (0, 2))
2 : ((3, 2), (4, 2))
3 : ((1, 4), (2, 2))
4 : ((4, 1), (2, 2))
5 : ((3, 2), (2, 0))
```

```
Your move:
Your move:3
**********
A: (1, 4)
B: (2, 2)
I will win.
**********
A: (1, 0)
B: (2, 2)
7 moves left.
Choices:
0 : ((1, 0), (2, 3))
1 : ((1, 0), (3, 2))
Your move:1
**********
A: (1, 0)
B: (3, 2)
I will win.
**********
A: (3, 0)
B: (3, 2)
6 moves left.
Choices:
0 : ((2, 1), (3, 2))
1 : ((1, 2), (3, 2))
2 : ((3, 0), (3, 0))
3 : ((3, 0), (0, 2))
Your move:2
**********
A: (3, 0)
B: (3, 0)
I will win.
**********
A: (0, 0)
B: (3, 0)
B wins.

Process finished with exit code 0
```

Full Python code below:

```
# sum a and b together
# if the sum is at least 5
# return 0
# otherwise, return a + b
def overflow_sum(a, b):
    if a > 5 or b > 5:
        print("input error.")
        return
    if a < 0 or b < 0:
        print("input error.")
        return
    if a + b >= 5:
        return 0
    else:
        return a + b

# this function returns a list of all the possible next moves
# the format is the following
# (the state of player A, the state of player B, level)
# the state of a player is a tuple of two elements
# number of fingers on the left hand, and the number of fingers on the right hand

def next_moves(v):
    L = []
    # if the game is already over, return the empty list
    if v[0] == (0, 0) or v[1] == (0, 0): return L
    # the following depends on whose turn it is
    l0, r0 = v[0][0], v[0][1]
    l1, r1 = v[1][0], v[1][1]
    h = v[2]
    #print(l0, r0, l1, r1, h)
    if h % 2 == 0:
        s = set()
        # to make sure that we do not add zero with other numbers
        if l0 > 0 and l1 > 0:
            item = ((l0, r0), (overflow_sum(l0, l1), r1), h + 1)
            s.add(item)
        if l0 > 0 and r1 > 0:
            item = ((l0, r0), (l1, overflow_sum(l0, r1)), h + 1)
            s.add(item)
        if r0 > 0 and l1 > 0:
            item = ((l0, r0), (overflow_sum(r0, l1), r1), h + 1)
            s.add(item)
        if r0 > 0 and r1 > 0:
            item = ((l0, r0), (l1, overflow_sum(r0, r1)), h + 1)
            s.add(item)

        #move fingers from the left hand to the right hand
        for i in range(1, l0 + 1):
            l = l0 - i
            # to make sure we do not overflow
            if r0 + i < 5:
                r = overflow_sum(r0, i)
                #print(l, r, l0, r0)
                if (r, l) != (l0, r0):
                    item = ((l, r), (l1, r1), h + 1)
                    s.add(item)
        #move fingers from the right hand to the left hand
        for i in range(1, r0 + 1):
            l = overflow_sum(l0, i)
            r = r0 - i
            #print(l, r, l0, r0)
            if l0 + i < 5:
                if (r, l) != (l0, r0):
                    item = ((l, r), (l1, r1), h + 1)
                    s.add(item)
```

```
    else:
        s = set()
        if l1 > 0 and l0 > 0:
            item = ((overflow_sum(l0, l1), r0), (l1, r1), h + 1)
            s.add(item)
        if l1 > 0 and r0 > 0:
            item = ((l0, overflow_sum(r0,l1)), (l1, r1), h + 1)
            s.add(item)
        if r1 > 0 and l0 > 0:
            item = ((overflow_sum(l0, r1), r0), (l1, r1), h + 1)
            s.add(item)
        if r1 > 0 and r0 > 0:
            item = ((l0, overflow_sum(r0, r1)), (l1, r1), h + 1)
            s.add(item)

        for i in range(1, l1 + 1):
            l = l1 - i
            r = overflow_sum(r1, i)
            if r1 + i < 5:
                if (r, l) != (l1, r1):
                    item = ((l0, r0), (l, r), h + 1)
                    s.add(item)

        for i in range(1, r1 + 1):
            l = overflow_sum(l1, i)
            r = r1 - i
            if l1 + i < 5:
                if (r, l) != (l1, r1):
                    item = ((l0, r0), (l, r), h + 1)
                    s.add(item)
    for item in s:
        L.append(item)

    return L



#building the dictionary bottom up
# level = depth is the bottom
# level = 0 is the top
#this function should return a dictionary
#the key of the dictionary is state of player A, the state of player B and the level
# For example, a key looks like ((i, j), (k, l), level), where
#(i, j) is the state of player A, (k, l) is the state of the player B
#the value of the dictionary is the predetermined game result if both players make the right moves,
# and the best next move if the level is even, or the worst move if the level is odd
# note the best and the worst is measured in terms of player A
# For example, if the level is odd, and we choose a move that leads to the smallest value.
# But that is the best move for player B.

# the game result is with respective to player A
# -1 means the first player will lose
# 0 means it will be a tie
# 1 means the first player will win


def best_move_dp(depth):
    assert(depth % 2 == 0)
    d = {}
    # base cases
    for i in range(0, 5):
        for j in range(0, 5):
            for k in range(0, 5):
                for l in range(0, 5):
                    if i == 0 and j==0 and k == 0 and l == 0:
                        d[((i, j), (k, l), depth)] = (0, [])
```

```
                elif i==0 and j==0:
                    d[((i, j), (k, l), depth)] = (-1, [])
                elif k==0 and l==0:
                    d[((i, j), (k, l), depth)] = (1, [])
                else:
                    d[((i, j), (k, l), depth)] = (0, [])
    #using a set to make check if a move is in the dictionary easier
    MyKeys = set()
    for h in range(depth -1, -1, -1):
        #this is skewed towards whoever playing first, that means that if you are going first, you want to take
the move
        #with the highest value, and if you go second, you want to take the move with the lowest value

        for i in range(0, 5):
            for j in range(0, 5):
                for k in range(0, 5):
                    for l in range(0, 5):
                        #this means that this state is not in the dictionary yet
                         if str(((i, j), (k, l), h)) not in MyKeys:
                            #no matter what the dictionary will have this key by the end of this loop
                             MyKeys.add(str(((i, j), (k, l), h)))
                            #case where someone has won/ a tie
                             if i == 0 and j == 0 and k == 0 and l == 0:
                                d[((i, j), (k, l), h)] = (0, [])
                                continue
                             elif i == 0 and j == 0:
                                d[((i, j), (k, l), h)] = (-1, [])
                                continue
                             elif k == 0 and l == 0:
                                d[((i, j), (k, l), h)] = (1, [])
                                continue

                             #Checks all next moves and how many next moves there are
                             NMoves = next_moves(((i, j), (k, l), h))
                            length = len(NMoves)
                            #since you have to build this tree from the bottom up, every move in a lower depth is
already in the dictionary
                             if length > 0:
                                if h%2 == 0:
                                    #this is player A, meaning you want the move with the highest value
                                     #start at -2 as there's nothing lower than that, so worst case we have
max(-1)
                                     nextV = -2
                                    nextM = []
                                    #checking for the max
                                     for x in range(length):
                                        if d[NMoves[x]][0] > nextV:
                                            #set new max
                                              nextV = d[NMoves[x]][0]
                                            nextM = NMoves[x]
                                    d[((i, j), (k, l), h)] = (nextV, nextM)

                                #this means you take the move with the maximum value
                                 else:
                                #this means that h%2 is 1, this means that this is player B
                                 #and you have to take the move with the minimum possible value
                                     #worst case minimum will still be smaller than this 2
                                     nextV = 2
                                    nextM = []
                                    #for all possible moves
                                     for y in range(length):
                                        if d[NMoves[y]][0] < nextV:
                                            #set new min
                                              nextV = d[NMoves[y]][0]
                                            nextM = NMoves[y]
                                    d[((i, j), (k, l), h)] = (nextV, nextM)
```

```
                            #last base case, a final else just in case
                            else:
                                d[((i, j), (k, l), depth)] = (0, [])
                                continue
    return d


# display the game state
def display(w):
    print("**********************")
    print("A:", w[0])
    print("B:", w[1])


def chopsticks_game(moves, index):
    depth = 2*moves
    d = best_move_dp(depth)
    w = ((1, 1), (1, 1), 0)
    display(w)
    moves_left = moves
    turn = 0
    while w[0] != (0, 0 ) and w[1] != (0, 0) and w[2] < depth:
        # computer will use d[w] to decide its move
        if (turn + 1) % 2 == index:
            result, v = d[w]
            if (result == 1 and index == 1) or (result == -1 and index == 0): print("I will win.")
            else: print("I may not win.")
            w = v
            moves_left -= 1
        else:
            all_next_moves = next_moves(w)
            num = len(all_next_moves)
            print(moves_left + index, "moves left.")
            print("Choices:")
            for i in range(len(all_next_moves)):
                print(i, ':', all_next_moves[i][:2])
            while True:
                s = input("Your move:")
                while not s.isnumeric():
                    s = input("Your move:")
                choice = int(s)
                if choice >= 0 and choice < num:
                    break
            v = list(w)
            v = all_next_moves[choice]
            v = tuple(v)
            w = v
        display(w)
        turn = turn +1
    if w[0] != (0,0) and w[1] != (0,0) and w[2] == depth:
        print("It is a tie.")
    elif w[0] == (0, 0):
        print("B wins.")
    else:
        print("A wins.")

moves = 10
print("The chopsticks game.")
print("The game will end within", moves, "moves.")
answer = input("Do you want to go first?")
if answer in ["Y", "y", "Yes", "YES", "yes"]:
    chopsticks_game(moves, 0)
else:
    chopsticks_game(moves, 1)
```