IB subject: Mathematics

Extended Essay

Random Variables and Uncertainties:

RQ: To what extent can a computer using the Kelly Criterion and conditional probability be used to increase the win percentage and overall profits of the casino game blackjack?

Word Count = 3991

Table of contents:

Introduction…… 2

Betting…… 3

Calculations for decision making …….. 8

Strategy ……. 15

Results and analysis …… 17

Ethical considerations …….. 22

Conclusion ……. 22

Bibliography …… 24

Entire program/code used ……. 24

Introduction:

I have always had a keen interest in probability and statistics throughout my life. This research uses that interest to find more knowledge about the casino game blackjack. More specifically, this research question will shed light on the dangers and mathematical problems of gambling. Simply, understanding that gambling is just another part of mathematics, where the odds are almost never in the players favor. The rules and parameters, that differ from the basic rules ("Blackjack"), that I will be using in this research are as follows: dealer hits until soft 17, if the player and dealer tie, play is pushed, however the player may adjust their bet. Splitting may only happen once, and doubling can be done on either with either 2 or 3 cards in the player's hand. There will be no buying insurance, and DAS (doubling after split) is not allowed. The shoe will consist of 6 decks shuffled together, and reshuffled when there are less than 75 cards remaining. There is no surrender, or refusal to play a hand, and the minimum bet that must be placed is 10. The goal of this research is to compare the win percentages, profits, and general outcomes between a computer algorithm I have created against a card counting human player.

To solve this problem, a series of simulations of blackjack for each algorithm is run. Trying to increase the profits and win percentages of the player during the simulation, using probability to break down each problem into a mathematical best solution, forming a strategy and statistics to analyze and decrypt the information presented from the simulations. By storing useful information from the simulations into an external excel file, it is easier to understand and process the data. To my knowledge, the extent of computer programs to increase the win percentage of the casino game blackjack has not been researched, since it cannot be

implemented into real casinos. My program started off as a simple blackjack simulation that only had the decisions to hit and stand (Hermon). Adding onto the program and tweaking it, I knew that I could eventually build the program found at the end of this paper.

Research question: To what extent can a computer using the Kelly Criterion and conditional probability be used to increase the win percentage and overall profits of the casino game blackjack? The Kelly Criterion is a formula which helps define the size of the bet that should be placed before each hand, which correlates to the increase in profits. On the other hand, conditional probability is mainly used to find the decision that has the greatest rate of success, increasing the win percentage. Finally, some strategy will be implemented, based on the different values and payouts of the game, an example being the double down option, which can double the potential payout.

Betting:

$$f = p - \frac{1-p}{b}$$

f = proportion of money to wager
p = the chance of winning the bet
b = the proportion of bet returned

, Equation 1

To determine the amount of money to bet on any given gamble, the gambling formula is used. This formula is a Kelly bet or Kelly criterion (“Kelly”), which is an equation (Equation 1) that can find the proportion of money to wager, f, in percentages, in accordance with the player's

bankroll. Meant for bets that are either won or lost, which is the case for blackjack. The fraction of money to bet is found using two inputs: the proportion of bet returned (b) and the chance of winning the bet (p).

Solving for the former (b) is slightly more complicated than just 1 since blackjack has different payouts for different hands. Most of the time, the blackjack payout is 1:1, since either the bet is doubled and returned for 100% return, or the bet is lost. However, when getting a natural blackjack (an ace in addition to any card valued at 10) the payout is 3:2, meaning the bet is not only doubled, but an extra 50% of the bet is gained. This is the reason why blackjack as a casino game can give the player an edge. Equation 2 below, finds the value of b, using two variables: the probability that the player receives a natural blackjack (P(x)), and the probability that the dealer receives a blackjack (P(y)), given that the player has also received a natural blackjack (P(y|x). When finding the probability of the player getting a 150% payout, the chance the player receiving a blackjack (P(x)) is multiplied by the chance the dealer does not get a blackjack (1-P(y|x)). For a payout of 100% (1), the probability of the player getting 150% payout is subtracted from 1 (1-P(x) · (1-P(y|x))).

$$b=1\cdot\left(1-P(x)\cdot\left(1-P(y|x)\right)\right)+1.5\cdot P(x)\cdot\left(1-P(y|x)\right)$$ Equation 2

$$=1-0.5\cdot P(x)\cdot\left(P(y|x)-1\right)$$ Equation 2 (factorized)

Code extract 1 calculates the maximum value of b (b_max), which then gives the maximum fraction of money that should be placed on any given hand (f_max = money · b_max). This calculates the best-case scenario: length of the deck is 75 cards, number of tens is 51 cards, and

number of aces is 24 cards. Resulting in a b_max of 1.1240330741700606. It does this by calculating the value of b for every possible value of the deck length (d), number of tens (tens) and number of aces (elevens).

```
b_max = 0
for q in range(75,313):
    d = q
    for w in range(97):
        tens = w
        if tens > d:
            tens = d
        for e in range(25):
          elevens = e
          if elevens + tens > d:
              elevens = d - tens
          b_d = ((elevens-1)/(d-2))*((tens-1)/(d-3))+((tens-1)/(d-2))*((elevens-1)/(d-3))
          b_2 = (elevens/d)*(tens/(d-1))+(tens/d)*(elevens/(d-1))
          #b = 1 * (1 - (b_2 * (1 - b_d))) + 1.5 * (b_2 * (1 - b_d))
          b = 1-0.5*b_2*(b_d-1)
          if b > b_max:
              b_max = b
              best_d = d
              best_elevens = elevens
              best_tens = tens
print(b, b_max, b_2, b_d)
print(best_d, best_tens, best_elevens)
```

Code extract 1

Solving for the win percentage (p), the mean wins throughout all the simulations is calculated to find the overall percentage chance of winning. Different win percentages will be displayed in the form Px, x being the number of simulations that formed the win percentage. An example being P150k = 0.498742, which is the average from the first 150 thousand simulations. This means that the win percentage used throughout different simulations may differ, giving different results.

Code extract 2 applies the gambling formula to the program, including its variables. To find the probability of hitting a natural blackjack (b_2), the program multiplies the probability of drawing an ace (deck.count(11)/len(deck), marked as P(11)) with the probability of drawing a ten, given that an ace has been drawn from the deck (deck.count(10)/(len(deck)-1), marked as

P(10|11). That when added to the opposite scenario of drawing a card valued at 10 first, gives all the scenarios in which the player receives a natural blackjack and therefore the probability of drawing a natural blackjack (P(NB)) in equation 3. To find the probability that the dealer has a natural blackjack given the player has also drawn a natural blackjack (p_d_bj), equation 4 considers the conditional probabilities of drawing an ace, given that one has already been drawn in addition to a ten ((deck.count(11)-1)/(len(deck)-2), marked as P(11|NB)), and drawing a ten, given both the players natural blackjack and the first ace ((deck.count(10)-1)/(len(deck)-3), marked as P(10|NB,11)).

$$P(NB)=P(11)\cdot P(10|11)+P(10)\cdot P(11|10)$$ Equation 3

$P(NB)$ = natrual blackjack

$P(10)$ = drawing a ten

$P(11)$ = drawing an ace

$P(DNB)$ = dealer getting a natural blackjack

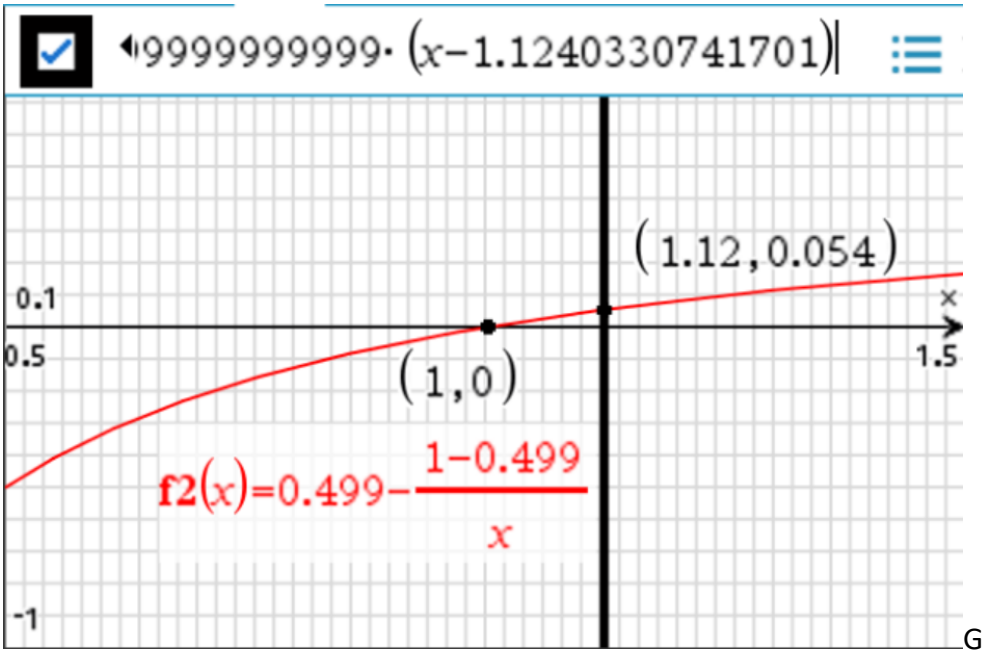
$$P(DNB)=P(11|NB)\cdot P(10|NB,11)+P(10|NB)\cdot P(11|NB,10)$$ Equation 4

```
b_2 = (deck.count(11)/len(deck))*(deck.count(10)/(len(deck)-1))+(deck.count(10)/len(deck))*(deck.count(11)/(len(deck)-1))
p_d_bj = ((deck.count(11)-1)/(len(deck)-2))*((deck.count(10)-1)/(len(deck)-3))+((deck.count(10)-1)/(len(deck)-2))*((deck.count(11)-1)/(len(deck)-3))
#b = 1*(1-(b_2*(1-p_d_bj))) + 1.5*(b_2*(1-p_d_bj)) this and the bottom one are the same (factorised)
b = 1-0.5*b_2*(p_d_bj-1)
if b > float(new_b):
    new_b = str(b)
p = 0.498742
f = p - (1-p)/b
amount = f*money
if amount < 10:
    return 10
max_bet = 0.053 * money
```

Code extract 2

Graph 1 shows the variance of betting sizes, for P100k, of which the variable x is the proportion of bet returned (b) and y is the percentage of the bankroll to place. The red function is the gambling formula, and the point (1.12,0.054) represents the highest value of b possible, b_max,

and shown above the graph. This means that the maximum percentage of the player's bankroll that should be bet on any given hand is 5.4%, according to the gambling formula.



Graph 1

In addition to the gambling formula, my program utilizes the card counting system proposed by Peter Griffin in his book "The Theory of Blackjack" (1979). Table 1 was formed based on his prior research (Griffin, p.44). For every visible card that is played, the values given by table 1 are added or subtracted to the total, which is then divided by the length of the deck remaining. Code extract 3 illustrates that information in the algorithm, by adding the corresponding values together and dividing by the length of the deck (len(deck_still)). Code extract 4 uses the value given from code extract 3 (true_count) to determine the best bet to wager. If the true_count is less than 1, then the minimum bet is placed. Unlike if the true_count was between 1 and 3, in which case the rounded value from the gambling formula (rounded) would be returned. Otherwise, it rounds the true_count (round(true_count)), subtracts this value by 1 and multiplies by the rounded proportion of money to bet (rounded). However, if this exceeds the

maximum bet that should be placed (max_bet or f_max · money), then the maximum bet is placed instead.

2	3	4	5	6	7	8	9	10	11
38	44	55	69	46	28	0	-18	-51	-61

Table 1

```
(deck_gone.count(2)*38+deck_gone.count(3)*44+deck_gone.count(4)*55+deck_gone.coun
t(5)*69+deck_gone.count(6)*46+deck_gone.count(7)*28-deck_gone.count(11)*61-
deck_gone.count(10)*51-deck_gone.count(9)*18)/len(deck_still)
```

Code extract 3

```
if true_count < 1:
    return 10
elif true_count < 3:
    if rounded <= max_bet:
        return rounded
    else:
        return round(max_bet)
else:
    if (round(true_count)-1)*rounded > max_bet:
        return round(max_bet)
    return (round(true_count)-1)*rounded
```

Code extract 4

Calculations for decision making:

An important percentage to have in any game is the opponent's chances to win or lose. Firstly, the chance that the dealer goes bust should be calculated. Simultaneously, the probability that the dealer reaches 17,18,19, 20 or 21 can be found. Before looking at the code constructed, it is important to understand its basis. Diagram 1 shows a simplified tree diagram of the dealer's actions and results: Hit (while under a total of 17), Stay (when the total ranges from 17-21) and finally Bust (when the total goes above 21). Each path has its own probability of occurring and

then at the end of each branch either returns a 1 or 0. Each edge of a path is multiplied together to find the percentage value of that path. Adding all the paths will give you the total percentage of that occurring. Flor example, in Diagram, the probability that the dealer goes bust can be found by multiplying each path that ends in a one ($1/3 \cdot 1/2 + 1/3 = 0.5$), resulting in a 50% chance the dealer will go over 21.

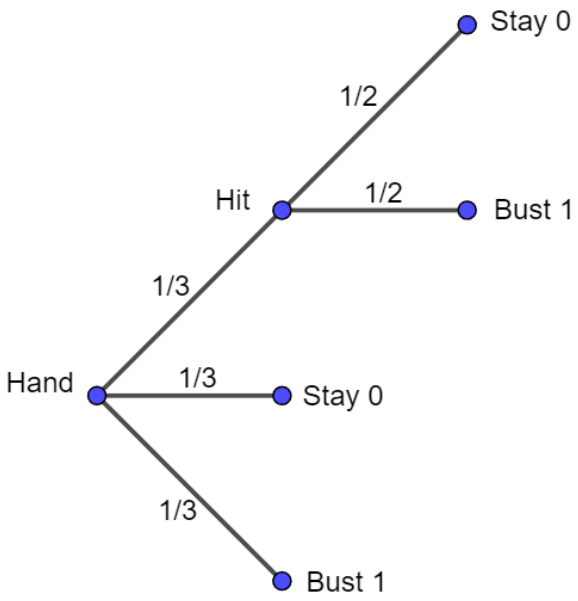


Diagram 1

Code extract 5 first creates all the possible hands that the dealer could have with his faceup card (starting_hands), which is the dealers face up card (a, in equation 5) added to every possible card in the deck (h) and then matches each hand with the probability of it occurring (matching_probability = deck.count(choice_2)/len(deck)). After this it calls upon the function (P in equation 6) searching for different outcomes (17,18,19, etc.). An example being dealer_p_18, which finds all the paths of the tree which results in the dealer having a total of 18. The function (D in equation 5) needs an input of the hand (a+h, sum of the two starting cards that the dealer could have), matching_deck (b-h, the deck which has removed the card added to the

hand), starting_minus_tens (c, for the amount of aces which are valued at 11, meaning it could be made into a 1), and [0,1,0,0,0,0] (d, a list which defines the target total for the dealers hand). Once this function returns a sum of the results (from P_dealer/P) it is then multiplied by the matching_probability (h∩b) and is added to the total sum (dealer_p_18).

```
starting_hands = []
choices_2 = list(set(deck))
starting_minus_tens = []
matching_probability = []
for choice_2 in choices_2:
    matching_probability.append(deck.count(choice_2)/len(deck))
    if choice_2 == 11 and dealerhand[0] == 11:
        choice_2 = 1
        starting_minus_tens.append(1)
    elif choice_2 == 11 or dealerhand[0] == 11:
        starting_minus_tens.append(1)
    else:
        starting_minus_tens.append(0)
    starting_hands.append(dealerhand[0]+choice_2)
dealer_p_bust = 0;dealer_p_17 = 0;dealer_p_18 = 0;dealer_p_19 = 0;dealer_p_20 = 0;dealer_p_21 = 0
for hand in starting_hands:
    matching_deck = deck.copy()
    matching_deck.remove(choices_2[starting_hands.index(hand)])
    dealer_p_17+=P_dealer(hand,matching_deck,starting_minus_tens[starting_hands.index(hand)],[1,0,0,0,0,0]) * matching_probability[starting_hands.i
    dealer_p_18+=P_dealer(hand,matching_deck,starting_minus_tens[starting_hands.index(hand)],[0,1,0,0,0,0]) * matching_probability[starting_hands.i
    dealer_p_19+=P_dealer(hand,matching_deck,starting_minus_tens[starting_hands.index(hand)],[0,0,1,0,0,0]) * matching_probability[starting_hands.i
    dealer_p_20+=P_dealer(hand,matching_deck,starting_minus_tens[starting_hands.index(hand)],[0,0,0,1,0,0]) * matching_probability[starting_hands.i
    dealer_p_21+=P_dealer(hand,matching_deck,starting_minus_tens[starting_hands.index(hand)],[0,0,0,0,1,0]) * matching_probability[starting_hands.i
    dealer_p_bust += P_dealer(hand,matching_deck,starting_minus_tens[starting_hands.index(hand)],[0,0,0,0,0,1]) * matching_probability[starting_han
```

Code extract 5

$$D(a,b,c,d) = \sum_{h=2}^{11} \left(P(h+a,b-h,c,d \mid h+a \leq 16 \text{ or } h+a-10\cdot c \leq 16)\cdot h \cap b\right) + \sum_{h=2}^{11} \left(1\cdot h \cap b \mid 17 \leq h+a \leq 21 \text{ or } 17 \leq h+a-10\cdot c \leq 21\right)$$

Equation 5

$D(a,b,c,d)$ = starting hands

a = dealers face up card

b = deck

c = aces of value 11

d = goal

h = card drawn from deck

$h \cap b$ = chance of drawing card h

Code extract 6 is the function that is called upon by code extract 5, shown in equation 6 as function P: n = w = hand, placementdeck = x = matching_deck, minusten = y = starting_minus_tens, and goal = z = the goal as a list. If the total of the hand is less than 17 then the program will create a new path for each card it can get from the deck, otherwise it checks the total and returns the respective goal (1 or 0). The list [0,0,0,0,0,0] corresponds to getting [17,18,19,20,21, or over 21]. Which means the list [0,0,0,0,0,1] returns the result of all paths that end in the dealer's total being over 21. With the combination of these two extracts the probability of the dealer getting 17,18,19,20,21, and busting can be found. Equation 6 does this slightly differently, since it has an additional line of addition, for the card value 11 or an ace. It does this separately, because the value y must also increase by one with the path adding an ace (y+1).

```
def P_dealer(n,placementdeck,minusten,goal):
    if n < 17:
        p_sum = 0
        choices = list(set(placementdeck))
        randomlist = []
        randomlist.append(minusten)
        for choice in choices:
            new_deck = placementdeck.copy()
            new_deck.remove(choice)
            new_minusten = randomlist[0]
            total = n + choice
            if choice == 11:
                    new_minusten += 1
            if total > 21 and new_minusten > 0:
                new_minusten -= 1
                total -= 10
            p_sum += P_dealer(total,new_deck,new_minusten,goal)*(new_deck.count(choice)+1)/(len(new_deck)+1)
        return p_sum
    elif n == 17:
        return goal[0]
    elif n == 18:
        return goal[1]
    elif n == 19:
        return goal[2]
    elif n == 20:
        return goal[3]
    elif n == 21:
        return goal[4]
    else:
        return goal[5]
```

Code extract 6

$$P(w,x,y,z) = \sum_{i=2}^{i=10} \left(P(w+i,x-i,y,z|i+w\leq 16 \text{ or } i+w-10\cdot y\leq 16)\cdot i\cap x\right) + \sum_{i=2}^{11} \left(1\cdot i\cap x|17\leq i+w\leq 21 \text{ or } 17\leq i+w-10\cdot y\leq 21\right) + P(w+11,x-11,y+1,z|w+11\leq 16 \text{ or } w+11-10\cdot y\leq 16)\cdot 11\cap x$$ Equation 6

$P(w,x,y,z)$ = sum of the paths from the possible starting hands

w = hand

x = matching deck

y = aces of value 11

z = goal

i = card drawn from deck

$i\cap x$ = chance of drawing card i

The function P_player_double in code extract 7 finds the probabilities of each new hand after hitting once/doubling down (under 17, 18, 19, 20, 21, bust). It adds up the results of each path that that leads to the specific outcome and then inputs that value into the list (p_of_player). Comparing these results to those gained from the dealer's chances, the probability of winning if the player doubles down can be found. The code extract 8 is an equation which finds this probability (win_pecentage_double), by adding each possible outcome in which the players score is higher than the dealers score.

```
def P_player_double(hand_of_player,placement_deck):
    p_of_player = [0,0,0,0,0,0,0]
    choices_3 = list(set(placement_deck))
    player_minustens = hand_of_player.count(11)
    for choice in choices_3:
        newer_minustens = int(str(player_minustens))
        total_2 = sum(hand_of_player) + choice
        if choice == 11:
            newer_minustens += 1
        while total_2 > 21 and newer_minustens > 0:
            total_2 -= 10
            newer_minustens -= 1
        if total_2 <= 16:
            p_of_player[0] += placement_deck.count(choice)/len(placement_deck)
        elif total_2 == 17:
            p_of_player[1] += placement_deck.count(choice)/len(placement_deck)
        elif total_2 == 18:
            p_of_player[2] += placement_deck.count(choice)/len(placement_deck)
        elif total_2 == 19:
            p_of_player[3] += placement_deck.count(choice)/len(placement_deck)
        elif total_2 == 20:
            p_of_player[4] += placement_deck.count(choice)/len(placement_deck)
        elif total_2 == 21:
            p_of_player[5] += placement_deck.count(choice)/len(placement_deck)
        elif total_2 > 21:
            p_of_player[6] += placement_deck.count(choice)/len(placement_deck)
    return p_of_player
```

Code extract 7

```
win_percentage_double =
hit_p[0]*dealer_p_bust+hit_p[1]*(dealer_p_bust)+hit_p[2]*(dealer_p_bust+dealer_p_
17)+hit_p[3]*(dealer_p_bust+dealer_p_18+dealer_p_17)+hit_p[4]*(dealer_p_bust+deal
er_p_17+dealer_p_18+dealer_p_19)+hit_p[5]*(dealer_p_bust+dealer_p_17+dealer_p_18+
dealer_p_19+dealer_p_20)
```

Code extract 8

To find the win percentage of standing with any given hand (win_percentage_stand), there are two values to consider: the chance you beat the dealer, and the chance you tie with the dealer (expressed in the code as a list [the former, and the latter]. To find the probability of winning without taking any more cards, the total of the players hand is compared to the probability of the dealer having a worse outcome. For example, if the total of the players hand is seventeen, then the former part of the list is comprised of the dealer going bust, since that is the only outcome of the dealer that the player can beat with a total of seventeen. The latter part of the list by taking the chance the dealer also receives a total of seventeen.

```
if total(player_hand) < 17 or total(player_hand) > 21:
    win_percentage_stand = [dealer_p_bust,0]
elif total(player_hand) == 17:
    win_percentage_stand = [dealer_p_bust, dealer_p_17]
elif total(player_hand) == 18:
    win_percentage_stand = [dealer_p_bust+dealer_p_17, dealer_p_18]
elif total(player_hand) == 19:
    win_percentage_stand = [dealer_p_bust+dealer_p_17+dealer_p_18, dealer_p_19]
elif total(player_hand) == 20:
    win_percentage_stand = [dealer_p_bust+dealer_p_17+dealer_p_18+dealer_p_19, dealer_p_20]
elif total(player_hand) == 21:
    win_percentage_stand = [dealer_p_bust+dealer_p_17+dealer_p_18+dealer_p_19+dealer_p_20, dealer_p_21]
```

Code extract 9

The function in code extract 10 below, finds the general probability of winning when hitting. When the function is called it requires three variables of information: the player's hand (ph), the dealer's hand (dh) and the deck (placementdeck). Essentially, the function is yet another tree diagram, such as the one seen in code extract 6. However, the main difference is when to take another card further down the path. The function stand_and_double() which is called inside of this function finds the probability of winning if the player stands or doubles. Which is then used as a decision to decide the next course of action. If it is more beneficial to double down, then the function creates a new path for each card it can draw. Otherwise, if standing is the best choice, it decides that if the player ever got to that position, it would take the odds given from standing. If there is no risk in taking another card, then it will always hit. The function will add the results from each path to determine the strength of hitting as a percentage. This method does not yield the precise probability of winning if the player hits, since it uses the doubling down percentage to make its decision each time there is a variance.

```
def P_hitting(ph,dh,placementdeck):
    if total(ph) > 21:
        return 0
    starting_list = stand_and_double(ph,dh)
    if starting_list[4] == 0 and total(ph) <17 or starting_list[2] + starting_list[3] > starting_list[0] + starting_list[1]:
        p_sum = 0
        choices = list(set(placementdeck))
        for choice in choices:
            new_deck = placementdeck.copy()
            new_deck.remove(choice)
            total_1 = []
            for i in range(len(ph)):
                total_1.append(ph[i])
            total_1.append(choice)
            p_sum += P_hitting(total_1,dh,new_deck)*(new_deck.count(choice)+1)/(len(new_deck)+1)
        return p_sum
    else:
        return starting_list[0]+starting_list[1]
```

Code extract 10

Strategy:

Strategy in blackjack is simply comparing the probabilities of different actions and finding the one that yields the greatest rate of success. Using the calculation in the previous chapter, we are given percentage values for: hitting, doubling down, and splitting, including the probability each decision will end with a draw/push. Though, noting that in most cases splitting is not an option, and after splitting a hand, the player cannot double down.

Firstly, the decision of whether to double in code extract 11. It is not as simple as checking whether the probability of winning when doubling down is highest since the reward is two times greater for the player. The first rule of the program's strategy is to never double down, when the probability of a win is below 50% (stand_vs_double[2]>0.5). Logically, if the player has more than a 50% chance of winning, they should double their bet. Secondly, it checks whether 10% plus the win percentage of standing is greater than or equal to the win percentage when doubling down (stand_vs_double[2]>=stand_vs_double[0]-0.1). For example, if the probability of winning when doubling down is 55% and the probability of winning when standing is 80%, it makes much more sense to take the secure choice and stand.

```
if stand_vs_double[2] > 0.5 and stand_vs_double[2] >= stand_vs_double[0] - 0.1 and len(playerhand) <= 3 and len(player_hand_2)<1:
    choice = "d"
```

Code extract 11

Secondly, the decision of whether to stand or hit is considered, if doubling down is not a sure enough bet. In most cases (code extract 12) the simple decision can be made when looking directly at the chance of winning when hitting (win_percentage_hit) against the chance of winning when standing (stand_vs_double[0], win percentage, + stand_vs_double[1], draw percentage). If win_percentage_hit is higher, then the choice is to take another card, otherwise

stand. However, there is one exception to this rule. When there is no risk in taking another card and the player does not have an ace, doubling down is not an option and neither is splitting, the best course of action is receiving a so-called “free card” (since there is no risk in hitting). Code extract 13 takes this into account by hitting if the players total is less than 12, meaning no card taken from the deck can possibly result in the total of the hand going over 21, since an ace can be valued at 1.

```
if win_percentage_hit > stand_vs_double[0]+stand_vs_double[1]:
    choice = "h"
else:
    choice = "st"
```
Code extract 12

```
elif total(playerhand) < 12:
    choice = "h"
```
Code extract 13

When determining whether to split a hand, the hitting percentage function is also used. Comparing the results of the function when the player has a total of 8 versus a total of 4 (such as if the player’s hand was comprised of two cards valued at 4). Then finding the decision that yields the greatest rate of success.

The human card counting algorithm is based on the Hi-Lo system (“Card”) and basic strategy (“Learn”). The latter is used to determine the decision, for each possible set of hands and dealers face-up cards. The former is another card counting system, like the Griffin Ultimate, but the Hi-Lo system is the most used and considered the easiest card counting system, which determines the bet.

Results and analysis:

The win percentage of my algorithm after 160 thousand simulation is 49,9485%

The win percentage of the card counting algorithm after 5 million simulations is 49,7281%

Win percentages

P12k = 0,493487, P16k = 0,498581, P35k = 0,498744, P50k = 0,497812, P55k = 0,497856,

P76k = 0,498376, P83k = 0,499824, P90k = 0,498715, P100k = 0,99333, P150k = 0,498742,

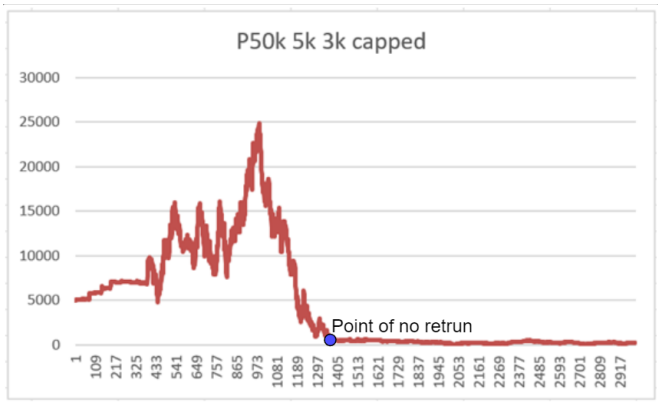
P160k = 0,499485

P values

After a quick glance of the different win percentages, it does not stand out to be an exceptionally large difference at all. However, just a 0,22% over thousands and thousands of simulations gives a strong advantage to using my program. This result is much less than my expected increase of the win percentage, which might indicate some slight errors in either the code or my calculations. In this specific case, looking at the win percentages, blackjack as a game seems to be unbeatable.

Combining the calculator and the betting system, both for my algorithm and the general card counting human, yield interesting results. Some of which, helped me improve the structure and design of the experiment. An example of this would include Graph 2, which shows a sharp increase in the players bankroll, then a sudden drop (to the point of no return), after which the bankroll remains near zero. This is because once the bankroll goes too low, there is no ability to bet large amount, meaning that the minimum bet of ten is always placed. And as determined earlier, if the bet stays the same, the game is losing. This made me change it to a system where after a certain amount of profit or loss, it will either buy in for more (money < 0.5 ·

flaot(initial_bank)) or deposit some money out (money>1.5 · float(initial_bank)). Initial_bank being the bankroll the player started with, as shown in code extract 20.



P50k = my algorithm with a win percentage after 50k simulations

5k = starting bankroll

3k = simulations

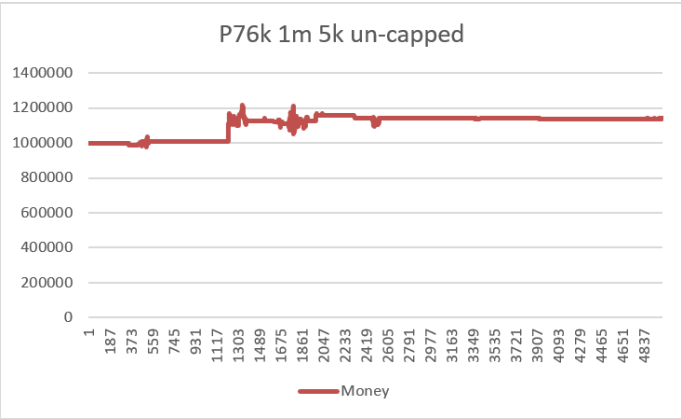
capped = max_bet active

Graph 2

```
if money < 0.5*float(initial_bank) or money > 1.5*float(initial_bank):
    money = The_Great_Reset(money)
```

Code extract 20

Graphs 3 and 4 below show how when the true count is low, both systems can maintain a general bankroll. Graph 4 is deceiving as it looks as though the bankroll is jumping up and down, but with closer inspection, you will find that the maximum point is 112365 and minimum point is 988530. Being only a 1,38% variance from the initial bankroll after playing 500 thousand hands. This proves that when the system at a high bankroll is always betting the same low value, its bankroll stays the same.



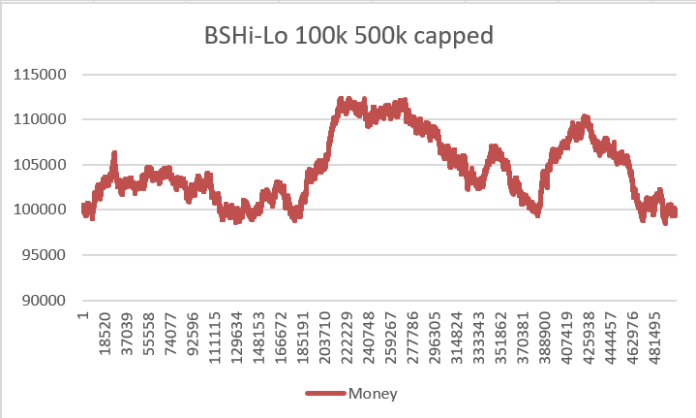
P76k = my algorithm with a win percentage after 76k simulations

1m = starting bankroll

5k = simulations

un-capped = no max_bet

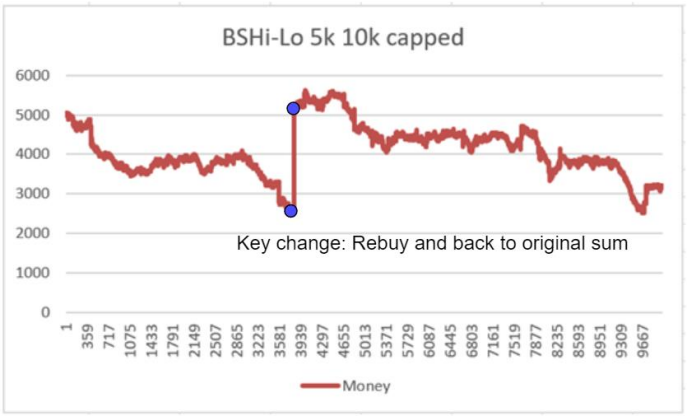
Graph 3



BSHi-Lo = Basic strategy and Hi-Lo system
100k = starting bankroll
500k = simulations
capped = max_bet active

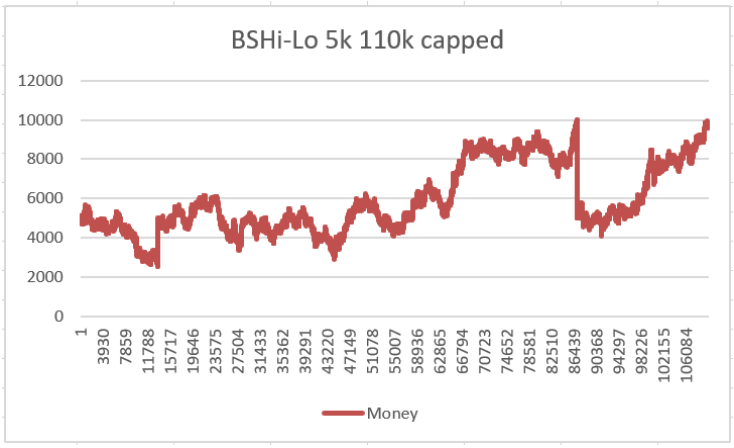
Graph 4

These systems are not immune to losing money, since in the end the probabilities are so close that there is bound to be some situations where they fail to profit. In graph 2, the failure to sustain a profit is seen in my system. It fails to maintain an upwards trend, which means that in a sense the program had failed. Similarly, in graph 5 the human program had a downwards trend. This time however, a key change took place, when it reached a bankroll of 2500, it bought in for more and then continued down its trajectory. Losing 4315 seems like quite significant, since it is 86,3% of the original bankroll. However, when considering this has over 10 thousand hands played. The loss per hand was only 0,4315 or 4,3% of the minimum bet.



BSHi-Lo = Basic strategy and Hi-Lo system
5k = starting bankroll
10k = simulations
un-capped = no max_bet

Graph 5



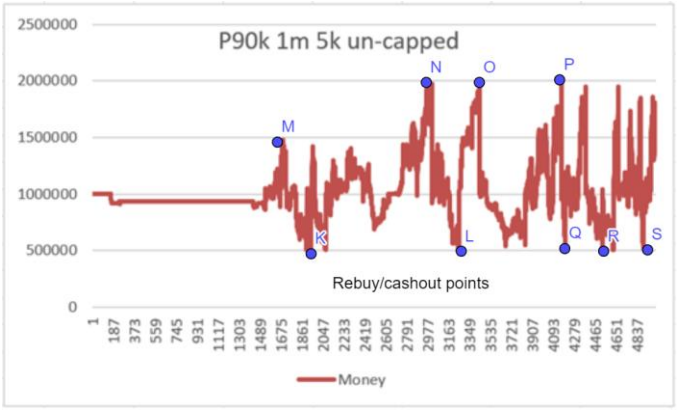
BSHi-Lo = Basic strategy and Hi-Lo system

5k = starting bankroll

110k = simulations

capped = max_bet active

Graph 6



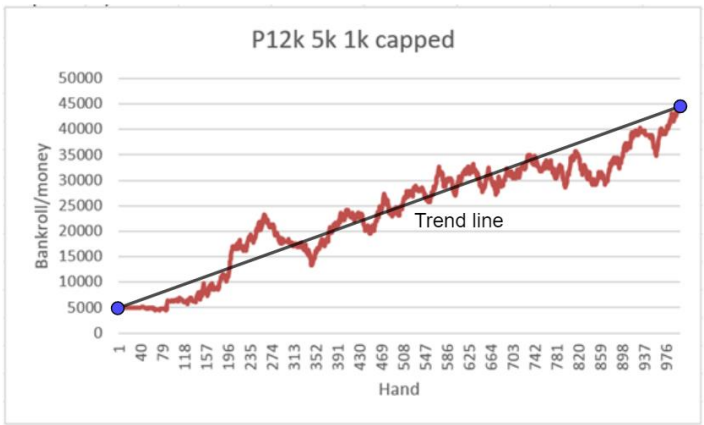
P90k = my algorithm with a win percentage after 90k simulations

1m = starting bankroll

5k = simulations

un-capped = no max_bet

Graph 7



P12k = my algorithm with a win percentage after 12k simulations

5k = starting bankroll

1k = simulations

capped = max_bet active

Graph 8

Graphs 6,7 and 8 are examples of the program succeeding in its task, which was to produce a higher bankroll. Table 3 compares useful information between each graph, but first the differences between each graph must be explained. Graph 6 represents the basic strategy in

combination with the high-low card counting system. Graph 7 represents my algorithm without an external betting cap. Finally, graph represents a casino cap of 500 for each bet.

Graph	6	7	8
Profit	7060	2196794	39588,5
Initial bankroll	5000	1000000	5000
Percentage profit (%)	141,2	219,6794	791,77
Hands played	110000	5000	1000
Profit per hand	0,064182	439,3588	39,5885
Win percentage	0,50009	0,50963	0,5299

Table 3

Graph 6 managed to make the money with the lowest win percentage. All being greater than their respective averages. To earn great amounts of money, luck had to be on the player's side. However, many simulations with a lower win percentage than the average were able to make a profit. Though in most cases the profit was minimal or insignificant. There were some of the best performers from their respective groups. Which is why graph 6 reflects well the strength on the human algorithm, being typically a slow and long increase in profits, however consistent.

Moving to graph 7, its main strength lies in its raw profit and profit per hand, since it was significantly higher than the other two. On the other hand, it began with an initial bankroll 200 times larger than the other two. Which is why when looking at performance graph 8 did the best. It increased its initial bankroll by the highest percentage 792% with the least number of hands played at 1000. To put it into perspective, it gained 3,6 times more of its initial bankroll than graph 7 and 5 times faster. The win percentage was abnormally high, however when playing such a little number of hands the win percentage is more likely to vary. Put against the

card counting human in graph 6, it gained 5,6 times more of its initial bankroll and 110 times faster.

Ethical considerations:

The ethical considerations of this research include: the unlawful use of this information to cheat at the game of blackjack and endorsing gambling. This information is purely hypothetical and should not be by any means used in the real-world, both online and in casinos or with real money. Gambling is a serious problem and can lead to addiction that should be taken seriously. Though hypothetical results may sound alluring in both human and computer play, in the end it comes down to chance. I am highly against gambling in the real-world and would advise others to keep away. This research is not to be misinterpreted. It is purely hypothetical, with no real risk or reward, meant only to use mathematical concepts to change results of an unfair game. The algorithm is not to be used in any game that is played with real money.

Conclusion:

My algorithm managed to outperform the card counting human in both win percentage and profits. Moreover, the combination of each win percentage with their respective card counting systems is where the key difference in abilities was seen. Its main advantage being the speed at which profits could be earned. However, it is important to notice that both systems, not equal, manage to consistently turn a profit. It is made clear here why casinos ban the use of external

help, in the form of a computer, because using mathematics is strong enough to turn the odds against the house. Meaning they would lose money.

Even in a hypothetical sense, where I am unable to see the simulations, I found myself drawn to the idea of gaining money so easily. It has never been clearer to me why some people in our society are eaten up by gambling and its potential rewards. This however is a problem that I believe should be addressed, together we should fight against these urgers to wager money, since it is mathematically not in our favor. In the end, these equations and calculations are incredibly difficult for anyone to do in their head, so is this hypothetical algorithm harmless? Can this information be used incorrectly to instead increase the problems related to gambling?

Bibliography:

- "Learn Blackjack Strategy", Blackjack Apprenticeship, Blackjack Strategy Charts - How to Play Perfect Blackjack (blackjackapprenticeship.com). Accessed 23.8.2023.
- Hermon, Michael. "python_blackjack.py", GitHub Gist, python blackjack · GitHub. Accessed 24.7.2023.
- Griffin, Peter. The Theory of Blackjack. Huntington Press, 1979.
- "Blackjack Rules", Online Gambling, 30.7.2021, Blackjack Rules - How to Play Blackjack (Beginners Guide) (onlinegambling.com)
- "Kelly Criterion", Wikipedia, Kelly criterion - Wikipedia. Accessed 23.8.2023.
- "Card Counting Systems", Betandbeat, Card Counting Systems – Hi-Lo, Hi-Opt, Zen, Halves & More (betandbeat.com). Accessed 12.9.2023.

Entire program/code used:

My calculator and decision-making algorithm:

```
import random
import sys
import xlsxwriter
sys.setrecursionlimit(1000000)
workbook = xlsxwriter.Workbook("P150k 500k 5k Griffin 5.3%.xlsx")
worksheet = workbook.add_worksheet("Sheet 1")
worksheet.write(0,0,"Hand")
worksheet.write(0,1,"Money")
worksheet.write(0,2,"Bet")
worksheet.write(0,4,"Lost or won")
```

```
worksheet.write(0,3,"Wins")


def set_bet(deck_still,deck_gone):
    global true_count;global new_b
    b_2 = (deck.count(11)/len(deck))*(deck.count(10)/(len(deck)-
1))+(deck.count(10)/len(deck))*(deck.count(11)/(len(deck)-1))
    p_d_bj = ((deck.count(11)-1)/(len(deck)-2))*((deck.count(10)-1)/(len(deck)-
3))+((deck.count(10)-1)/(len(deck)-2))*((deck.count(11)-1)/(len(deck)-3))
    #b = 1*(1-(b_2*(1-p_d_bj))) + 1.5*(b_2*(1-p_d_bj)) this and the bottom one
are the same (factorised)
    b = 1-0.5*b_2*(p_d_bj-1)
    p = 0.498742
    f = p - (1-p)/b
    amount = f*money
    if amount < 10:
        return 10
    max_bet = 0.053 * money
    rounded = round(amount)
    true_count =
(deck_gone.count(2)*37+deck_gone.count(3)*45+deck_gone.count(4)*52+deck_gone.coun
t(5)*70+deck_gone.count(6)*46+deck_gone.count(7)*27-deck_gone.count(11)*60-
deck_gone.count(10)*50-deck_gone.count(9)*17)/len(deck_still)
    if true_count < 1:
        return 10
    elif true_count < 3:
        if rounded <= max_bet:
```

```
            return rounded
        else:
            return round(max_bet)
    else:
        if (round(true_count)-1)*rounded > max_bet:
            return round(max_bet)
        return (round(true_count)-1)*rounded


decks_number = 6
deck = [2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10, 11]*(int(decks_number)*4)
card_reset_amount = 75
hands_to_play = int(input("How many hands should the AI play: "))
money = int(input("How much money does the player start with: "))
initial_bank = str(money)
random.shuffle(deck)
wins = 0
losses = 0
hands_played = 0
counting_deck = []
profits = 0


def reset():
    global deck
    deck = [2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10, 11]*(int(decks_number)*4)
    random.shuffle(deck)
```

```
def deal(deck):
    hand = []
    for i in range(2):
        card = deck.pop()
        counting_deck.append(card)
        hand.append(card)
    return hand


def play_again():
    global hands_to_play;global hands_played;global diddle
    hands_played += 1
    hands_to_play -= 1
    if len(deck) > card_reset_amount:
        again = "y"
    else:
        reset()
        again = "y"
    if again == "y" and hands_to_play != 0:
        print(bet,money)
        global diddle
        diddle = 1
    else:
        print(wins,losses,wins/(losses+wins),hands_played,money-
float(initial_bank))
        diddle = 0
```

```
def total(hand):
    total = 0
    ace_minus_count = hand.count(11)
    for card in hand:
        total += card
    while total > 21 and ace_minus_count > 0:
        total -= 10
        ace_minus_count -= 1
    return total


def hit(hand):
    card = deck.pop()
    counting_deck.append(card)
    hand.append(card)
    return hand


def P_dealer(n,placementdeck,minusten,goal):
    if n < 17:
        p_sum = 0
        choices = list(set(placementdeck))
        randomlist = []
        randomlist.append(minusten)
        for choice in choices:
            new_deck = placementdeck.copy()
            new_deck.remove(choice)
            new_minusten = randomlist[0]
```

```
            total = n + choice
            if choice == 11:
                    new_minusten += 1
            if total > 21 and new_minusten > 0:
                new_minusten -= 1
                total -= 10
            p_sum +=
P_dealer(total,new_deck,new_minusten,goal)*(new_deck.count(choice)+1)/(len(new_de
ck)+1)
        return p_sum
    elif n == 17:
        return goal[0]
    elif n == 18:
        return goal[1]
    elif n == 19:
        return goal[2]
    elif n == 20:
        return goal[3]
    elif n == 21:
        return goal[4]
    else:
        return goal[5]

def P_player_double(hand_of_player,placement_deck):
    p_of_player = [0,0,0,0,0,0,0]
    choices_3 = list(set(placement_deck))
```

```
    player_minustens = hand_of_player.count(11)
    for choice in choices_3:
        newer_minustens = int(str(player_minustens))
        total_2 = sum(hand_of_player) + choice
        if choice == 11:
                newer_minustens += 1
        while total_2 > 21 and newer_minustens > 0:
            total_2 -= 10
            newer_minustens -= 1
        if total_2 <= 16:
            p_of_player[0] += placement_deck.count(choice)/len(placement_deck)
        elif total_2 == 17:
            p_of_player[1] += placement_deck.count(choice)/len(placement_deck)
        elif total_2 == 18:
            p_of_player[2] += placement_deck.count(choice)/len(placement_deck)
        elif total_2 == 19:
            p_of_player[3] += placement_deck.count(choice)/len(placement_deck)
        elif total_2 == 20:
            p_of_player[4] += placement_deck.count(choice)/len(placement_deck)
        elif total_2 == 21:
            p_of_player[5] += placement_deck.count(choice)/len(placement_deck)
        elif total_2 > 21:
            p_of_player[6] += placement_deck.count(choice)/len(placement_deck)
    return p_of_player


def blackjack(dealer_hand, player_hand):
```

```
    global wins;global losses;global money;global bet
    if total(player_hand) == 21 == total(dealer_hand):
        dealer_hand = []
        player_hand = []
        global quit;quit=True
    elif total(player_hand) == 21:
        money += bet*1.5
        wins += 1.5
        quit=True
    elif total(dealer_hand) == 21:
        money -= bet
        losses += 1
        quit=True


def score(player_hand):
        print("Scoring...")
        global wins;global losses;global money;global bet;global dealer_hand
        if total(player_hand) > 21:
            money -= bet
            losses += 1
        elif total(player_hand)<22:
            while total(dealer_hand)<17:
                hit(dealer_hand)
            if total(dealer_hand) > 21:
                money += bet
                wins += 1
```

```
            elif total(player_hand) == 21 and total(dealer_hand)==21:
                print("push")
            elif total(player_hand) == 21:
                money += bet
                wins += 1
            elif total(dealer_hand) == 21:
                money -= bet
                losses += 1
            elif total(player_hand) > 21:
                money -= bet
                losses += 1
            elif total(dealer_hand) > 21:
                money += bet
                wins += 1
            elif total(player_hand) < total(dealer_hand):
                money -= bet
                losses += 1
            elif total(player_hand) > total(dealer_hand):
                money += bet
                wins += 1
        print(wins, losses)


def game():
    global wins;global losses;global player_hand;global quit;global quit_2;global
player_hand_2;global bet;global stop;global dealer_hand;global did_win_2
    player_hand_2 = []
```

```
    quit=False
    quit_2=False
    stop = False
    bet=set_bet(deck,counting_deck)
    dealer_hand = deal(deck)
    player_hand = deal(deck)
    print ("The dealer is showing a " + str(dealer_hand[0]))
    print ("You have a " + str(player_hand) + " for a total of " +
str(total(player_hand)))
    blackjack(dealer_hand, player_hand)
    while not quit:
        decision(player_hand, dealer_hand)
    while quit_2:
        hit(player_hand_2)
        quit_2 = False
        while not stop:
            decision(player_hand_2, dealer_hand)


def stand_and_double(player_hand,dealerhand):
        starting_hands = []
        choices_2 = list(set(deck))
        starting_minus_tens = []
        matching_probability = []
        for choice_2 in choices_2:
            matching_probability.append(deck.count(choice_2)/len(deck))
            if choice_2 == 11 and dealerhand[0] == 11:
```

```
                choice_2 = 1
                starting_minus_tens.append(1)
            elif choice_2 == 11 or dealerhand[0] == 11:
                starting_minus_tens.append(1)
            else:
                starting_minus_tens.append(0)
            starting_hands.append(dealerhand[0]+choice_2)
        dealer_p_bust = 0;dealer_p_17 = 0;dealer_p_18 = 0;dealer_p_19 =
0;dealer_p_20 = 0;dealer_p_21 = 0
        for hand in starting_hands:
            matching_deck = deck.copy()
            matching_deck.remove(choices_2[starting_hands.index(hand)])
            dealer_p_17+=P_dealer(hand,matching_deck,starting_minus_tens[starting
_hands.index(hand)],[1,0,0,0,0,0]) *
matching_probability[starting_hands.index(hand)]
            dealer_p_18+=P_dealer(hand,matching_deck,starting_minus_tens[starting
_hands.index(hand)],[0,1,0,0,0,0]) *
matching_probability[starting_hands.index(hand)]
            dealer_p_19+=P_dealer(hand,matching_deck,starting_minus_tens[starting
_hands.index(hand)],[0,0,1,0,0,0]) *
matching_probability[starting_hands.index(hand)]
            dealer_p_20+=P_dealer(hand,matching_deck,starting_minus_tens[starting
_hands.index(hand)],[0,0,0,1,0,0]) *
matching_probability[starting_hands.index(hand)]
```

```
            dealer_p_21+=P_dealer(hand,matching_deck,starting_minus_tens[starting
_hands.index(hand)],[0,0,0,0,1,0]) *
matching_probability[starting_hands.index(hand)]
            dealer_p_bust +=
P_dealer(hand,matching_deck,starting_minus_tens[starting_hands.index(hand)],[0,0,
0,0,0,1]) * matching_probability[starting_hands.index(hand)]
        if total(player_hand) < 17 or total(player_hand) > 21:
            win_percentage_stand = [dealer_p_bust,0]
        elif total(player_hand) == 17:
            win_percentage_stand = [dealer_p_bust, dealer_p_17]
        elif total(player_hand) == 18:
            win_percentage_stand = [dealer_p_bust+dealer_p_17, dealer_p_18]
        elif total(player_hand) == 19:
            win_percentage_stand = [dealer_p_bust+dealer_p_17+dealer_p_18,
dealer_p_19]
        elif total(player_hand) == 20:
            win_percentage_stand =
[dealer_p_bust+dealer_p_17+dealer_p_18+dealer_p_19, dealer_p_20]
        elif total(player_hand) == 21:
            win_percentage_stand =
[dealer_p_bust+dealer_p_17+dealer_p_18+dealer_p_19+dealer_p_20, dealer_p_21]
        hit_p = P_player_double(player_hand,deck)
        win_percentage_double =
hit_p[0]*dealer_p_bust+hit_p[1]*(dealer_p_bust)+hit_p[2]*(dealer_p_bust+dealer_p_
17)+hit_p[3]*(dealer_p_bust+dealer_p_18+dealer_p_17)+hit_p[4]*(dealer_p_bust+deal
```

```
er_p_17+dealer_p_18+dealer_p_19)+hit_p[5]*(dealer_p_bust+dealer_p_17+dealer_p_18+
dealer_p_19+dealer_p_20)
        draw_percentage_hit =
hit_p[1]*(dealer_p_17)+hit_p[2]*(dealer_p_18)+hit_p[3]*(dealer_p_19)+hit_p[4]*(de
aler_p_20)+hit_p[5]*(dealer_p_21)
        return
[win_percentage_stand[0],win_percentage_stand[1],win_percentage_double,draw_perce
ntage_hit,hit_p[6]]


def P_hitting(ph,dh,placementdeck):
    if total(ph) > 21:
        return 0
    starting_list = stand_and_double(ph,dh)
    if starting_list[4] == 0 and total(ph) <17 or starting_list[2] +
starting_list[3] > starting_list[0] + starting_list[1]:
        p_sum = 0
        choices = list(set(placementdeck))
        for choice in choices:
            new_deck = placementdeck.copy()
            new_deck.remove(choice)
            total_1 = []
            for i in range(len(ph)):
                total_1.append(ph[i])
            total_1.append(choice)
            p_sum +=
P_hitting(total_1,dh,new_deck)*(new_deck.count(choice)+1)/(len(new_deck)+1)
```

```
        return p_sum
    else:
        return starting_list[0]+starting_list[1]


def decision(playerhand, dealerhand):
        global money;global bet;global quit_2;global stop
        deck.append(dealerhand[1])
        last_resort = 0
        stand_vs_double = stand_and_double(playerhand,dealerhand)
        if playerhand[0] == playerhand[1] and len(playerhand) == 2 and
len(player_hand_2) == 0:
            if playerhand[0] == 11 or playerhand[0] == 8:
                choice = "sp"
            elif playerhand[0] == 5 or playerhand[0] == 4:
                last_resort = 1
            elif playerhand[0] == 10:
                choice = "st"
            elif dealerhand[0]<7:
                choice = "sp"
            elif dealerhand[0]>7 and playerhand[0]<8:
                last_resort = 1
            elif dealerhand[0]==7:
                if playerhand[0]<4 or playerhand[0]==7:
                    choice = "sp"
                else:
                    last_resort = 1
```

```
            else:
                w_p_h = 0;w_p_h_2 = 0
                s_h = [];m_p = []
                s_h_2 = []
                c_2s = list(set(deck))
                for c_2 in c_2s:
                    n_h = []
                    n_h_2 = []
                    m_p.append(deck.count(c_2)/len(deck))
                    n_h.append(playerhand[0]);n_h.append(playerhand[1]);n_h.appen
d(c_2)
                    n_h_2.append(playerhand[0]);n_h_2.append(c_2)
                    s_h.append(n_h)
                    s_h_2.append(n_h_2)
                for h in s_h:
                    m_d = deck.copy()
                    m_d.remove(c_2s[s_h.index(h)])
                    w_p_h += P_hitting(h,dealerhand,m_d) * m_p[s_h.index(h)]
                for h_2 in s_h_2:
                    m_d = deck.copy()
                    m_d.remove(c_2s[s_h_2.index(h_2)])
                    w_p_h_2 += P_hitting(h_2,dealerhand,m_d) * m_p[s_h.index(h)]
                if w_p_h_2 >= 0.5 and w_p_h_2 >=
stand_vs_double[0]+stand_vs_double[1] - 0.1 or w_p_h_2 > w_p_h and w_p_h_2 >
stand_vs_double[0]+stand_vs_double[1]:
                    choice = "sp"
```

```
                elif stand_vs_double[2] >= 0.5 and stand_vs_double[2] >=
stand_vs_double[0] - 0.1 and len(player_hand_2)<1 and len(playerhand)<=3:
                    choice = "d"
                elif w_p_h >= stand_vs_double[0]+stand_vs_double[1] or
stand_vs_double[4] == 0:
                    choice = "h"
                else:
                    choice = "st"
                print(w_p_h,w_p_h_2)
        elif playerhand.count(11) > 0 and sum(playerhand) -
(10*(playerhand.count(11)-1)) < 22: #indicates a soft hand
            if stand_vs_double[2] > 0.5 and stand_vs_double[2] >=
stand_vs_double[0] - 0.1 and len(playerhand) <= 3 and len(player_hand_2)<1:
                choice = "d"
            elif stand_vs_double[2] + stand_vs_double[3] >
stand_vs_double[0]+stand_vs_double[1] or total(playerhand) < 17:
                choice = "h"
            elif stand_vs_double[0]+stand_vs_double[1] > 0.5:
                choice="st"
            else:
                last_resort=1
        else:
            last_resort = 1
        if last_resort==1:
            if stand_vs_double[2] >= 0.5 and stand_vs_double[2] >=
stand_vs_double[0] - 0.1 and len(player_hand_2)<1 and len(playerhand)<=3:
```

```
                choice = "d"
            elif playerhand[0]==11==playerhand[1] and len(playerhand)==2:
                choice = "h"
            elif stand_vs_double[0]+stand_vs_double[1] >= 0.5:
                choice = "st"
            elif total(playerhand) < 12:
                choice = "h"
            elif total(playerhand) > 15:
                if stand_vs_double[0]+stand_vs_double[1] >
stand_vs_double[2]+stand_vs_double[3]:
                    choice = "st"
                else:
                    choice = "h"
            else:
                win_percentage_hit = 0
                starting_hands = [];matching_probability = []
                choices_2 = list(set(deck))
                for choice_2 in choices_2:
                    new_hand = []
                    matching_probability.append(deck.count(choice_2)/len(deck))
                    for i in range (len(playerhand)):
                        new_hand.append(playerhand[i])
                    new_hand.append(choice_2)
                    starting_hands.append(new_hand)
                for hand in starting_hands:
                    matching_deck = deck.copy()
```

```
                    matching_deck.remove(choices_2[starting_hands.index(hand)])
                    win_percentage_hit +=
P_hitting(hand,dealerhand,matching_deck) *
matching_probability[starting_hands.index(hand)]
                if win_percentage_hit > stand_vs_double[0]+stand_vs_double[1]:
                    choice = "h"
                else:
                    choice = "st"
        deck.pop()
        global wins;global losses
        if choice == "h":
            global quit
            if len(player_hand_2) > 1:
                hit(player_hand_2)
                if total(player_hand_2) < 22:
                    print(player_hand_2)
                elif total(player_hand_2) >= 22:
                    score(player_hand)
                    score(player_hand_2)
                    stop = True
            elif len(player_hand_2)==1:
                hit(player_hand)
                if total(player_hand) > 21:
                    quit = True
            else:
                hit(player_hand)
```

```
                print(player_hand)
                if total(player_hand)>21:
                    losses += 1
                    money -= bet
                    quit=True
        elif choice == "st":
            if len(player_hand_2)==1:
                print("next hand")
            elif len(player_hand_2) >= 2:
                quit=True
                stop = True
                score(player_hand)
                score(player_hand_2)
            else:
                score(player_hand)
            quit=True
        elif choice == "sp" and len(playerhand)==2 and
playerhand[0]==playerhand[1]:
            player_hand_2.append(player_hand.pop())
            hit(player_hand)
            quit_2 = True
        elif choice == "d" and len(playerhand) <= 3 and len(player_hand_2)<1:
                hit(playerhand)
                print("Hand total: " + str(total(playerhand)))
                score(player_hand);score(player_hand)
                quit=True
```

```
        elif choice == "q":
            print(wins,losses)
            quit=True;quit_2=True
            exit()


def The_Great_Reset(bankroll):
    global profits
    profits += bankroll;profits -= float(initial_bank)
    return float(initial_bank)


def change_in_money(new_bank,og):
    if new_bank + bet == og:
        return [0,1]
    elif new_bank == og:
        return [0,0]
    elif new_bank - bet == og:
        return [1,0]
    elif new_bank - 1.5*bet == og:
        return [1.5,0]
    elif new_bank + 2*bet == og:
        return [0,2]
    elif new_bank - 2*bet == og:
        return [2,0]
    elif new_bank > og:
        return [1,0]
    elif new_bank < og:
```

```
        return [0,1]


diddle = 1;hand = 0;bet = 0;true_count = 0;original = str(money)


if __name__ == "__main__":
    while diddle == 1:
       result = change_in_money(money, float(original))
       worksheet.write(hand+1 ,0,hand)
       worksheet.write(hand+1 ,1,money)
       worksheet.write(hand+1 ,2,bet)
       worksheet.write(hand+1 ,3,wins)
       worksheet.write(hand+1, 4,str(result))
       hand += 1
       if money < 0.5*float(initial_bank) or money > 1.5*float(initial_bank):
           money = The_Great_Reset(money)
       original = str(money)
       game()
       play_again()
    worksheet.write(2,6,profits)
    worksheet.write(2,8,losses)
    worksheet.write(2,7,wins)
    workbook.close()
```

The algorithm meant to imitate a human player:

```
decks_number = 6
```

```
deck = [2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10, 11]*(int(decks_number)*4)
card_reset_amount = 75
hands_to_play = int(input("How many hands should the AI play: "))
money = int(input("How much money does the player start with: "))
initial_bank = str(money)
random.shuffle(deck)
wins = 0
losses = 0
hands_played = 0
counting_deck = []
profits = 0


def reset():
    global deck
    deck = [2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10, 11]*(int(decks_number)*4)
    random.shuffle(deck)


def deal(deck):
    hand = []
    for i in range(2):
        card = deck.pop()
        counting_deck.append(card)
        hand.append(card)
    return hand


def play_again():
```

```
    global hands_to_play;global hands_played;global diddle
    hands_played += 1
    hands_to_play -= 1
    if len(deck) > card_reset_amount:
        again = "y"
    else:
        reset()
        again = "y"
    if again == "y" and hands_to_play != 0:
        print(bet,money)
        global player_hand;global player_hand_2; global diddle
        player_hand.clear()
        player_hand_2.clear()
        diddle = 1
    else:
        print(wins,losses,wins/(losses+wins),hands_played,money-
float(initial_bank),profits)
        diddle = 0


def total(hand):
    total = 0
    ace_minus_count = hand.count(11)
    for card in hand:
        total += card
    while total > 21 and ace_minus_count > 0:
        total -= 10
```

```
        ace_minus_count -= 1
    return total


def hit(hand):
    card = deck.pop()
    counting_deck.append(card)
    hand.append(card)
    return hand


def blackjack(dealer_hand, player_hand):
    global wins;global losses;global money;global bet
    if total(player_hand) == 21 == total(dealer_hand):
        print(wins, losses)
        dealer_hand = []
        player_hand = []
        game()
    elif total(player_hand) == 21:
        money += bet*1.5
        wins += 1.5
        print(wins, losses)
        global quit;quit=True
    elif total(dealer_hand) == 21:
        money -= bet
        losses += 1
        print(wins, losses)
        quit=True
```

```
def score(player_hand):
        global wins;global losses;global money;global bet;global dealer_hand
        if total(player_hand) > 21:
            money -= bet
            losses += 1
        if total(player_hand)<22:
            while total(dealer_hand)<17:
                hit(dealer_hand)
            if total(dealer_hand) > 21:
                money += bet
                wins += 1
            elif total(player_hand) == 21  and total(dealer_hand)==21:
                print("push")
            elif total(player_hand) == 21:
                money += bet
                wins += 1
            elif total(dealer_hand) == 21:
                money -= bet
                losses += 1
            elif total(player_hand) > 21:
                money -= bet
                losses += 1
            elif total(dealer_hand) > 21:
                money += bet
                wins += 1
```

```
            elif total(player_hand) < total(dealer_hand):
                money -= bet
                losses += 1
            elif total(player_hand) > total(dealer_hand):
                money += bet
                wins += 1
        print(wins, losses)


def game():
    global wins;global losses;global player_hand;global quit;global quit_2;global
player_hand_2;global bet;global stop;global dealer_hand
    player_hand_2 = []
    quit=False
    quit_2=False
    stop = False
    bet=set_bet(deck,counting_deck)
    dealer_hand = deal(deck)
    player_hand = deal(deck)
    print ("The dealer is showing a " + str(dealer_hand[0]))
    print ("You have a " + str(player_hand) + " for a total of " +
str(total(player_hand)))
    blackjack(dealer_hand, player_hand)
    while not quit:
        decision(player_hand, dealer_hand)
    while quit_2:
        hit(player_hand_2)
```

```
        quit_2 = False
        while not stop:
            decision(player_hand_2, dealer_hand)


hard_totals = [
    ["st","st","st","st","st","st","st","st","st","st"],
    ["st","st","st","st","st","h","h","h","h","h"],
    ["st","st","st","st","st","h","h","h","h","h"],
    ["st","st","st","st","st","h","h","h","h","h"],
    ["st","st","st","st","st","h","h","h","h","h"],
    ["h","h","st","st","st","h","h","h","h","h"],
    ["d","d","d","d","d","d","d","d","d","d"],
    ["d","d","d","d","d","d","d","d","h","h"],
    ["h","d","d","d","d","h","h","h","h","h"],
    ["h","h","h","h","h","h","h","h","h","h"],
]
soft_totals =[
    ["st","st","st","st","st","st","st","st","st","st"],
    ["st","st","st","st","d","st","st","st","st","st"],
    ["d","d","d","d","d","st","st","h","h","h"],
    ["h","d","d","d","d","h","h","h","h","h"],
    ["h","h","d","d","d","h","h","h","h","h"],
    ["h","h","d","d","d","h","h","h","h","h"],
    ["h","h","h","d","d","h","h","h","h","h"],
    ["h","h","h","d","d","h","h","h","h","h"],
]
```

```
pair_splitting = [
    ["sp","sp","sp","sp","sp","sp","sp","sp","sp","sp"],
    ["st","st","st","st","st","st","st","st","st","st"],
    ["sp","sp","sp","sp","sp","sp","st","sp","st","st"],
    ["sp","sp","sp","sp","sp","sp","sp","sp","sp","sp"],
    ["sp","sp","sp","sp","sp","sp","h","h","h","h"],
    ["sp","sp","sp","sp","sp","h","h","h","h","h"],
    ["d","d","d","d","d","d","d","d","h","h"],
    ["h","h","h","h","h","h","h","h","h","h"],
    ["h","h","sp","sp","sp","sp","h","h","h","h"],
    ["h","h","sp","sp","sp","sp","h","h","h","h"],
]
dictionary_dealer = {2:0,3:1,4:2,5:3,6:4,7:5,8:6,9:7,10:8,11:9,}
dictionary_player_hard =
{21:0,20:0,19:0,18:0,17:0,16:1,15:2,14:3,13:4,12:5,11:6,10:7,9:8,8:9,7:9,6:9,5:9,
4:9,3:9,}
dictionary_player_soft = {21:0,20:0,19:1,18:2,17:3,16:4,15:5,14:6,13:7,12:7,}
dictionary_player_pair = {12:0,20:1,18:2,16:3,14:4,12:5,10:6,8:7,6:8,4:9,}


def decision(playerhand, dealerhand):
        global money;global bet;global stop;global quit_2
        if playerhand[0] == playerhand [1] and len(playerhand) == 2 and
len(player_hand_2) == 0:
            choice =
pair_splitting[dictionary_player_pair[total(playerhand)]][dictionary_dealer[deale
rhand[0]]]
```

```
        elif playerhand.count(11) > 0 and sum(playerhand) -
(10*(playerhand.count(11)-1)) < 22:
            choice =
soft_totals[dictionary_player_soft[total(playerhand)]][dictionary_dealer[dealerha
nd[0]]]
        else:
            choice =
hard_totals[dictionary_player_hard[total(playerhand)]][dictionary_dealer[dealerha
nd[0]]]
        if choice == "d" and len(playerhand) > 3 or choice == "d" and
len(player_hand_2)>0:
            choice = "h"
        global wins;global losses
        if choice == "h":
            global quit
            if len(player_hand_2) > 1:
                hit(player_hand_2)
                if total(player_hand_2) < 22:
                    print(player_hand_2)
                elif total(player_hand_2) >= 22:
                    score(player_hand)
                    score(player_hand_2)
                    stop = True
            elif len(player_hand_2)==1:
                hit(player_hand)
                if total(player_hand) > 21:
```

```
                    quit = True
            else:
                hit(player_hand)
                print(player_hand)
                if total(player_hand)>21:
                    losses += 1
                    money -= bet
                    quit=True
        elif choice == "st":
            if len(player_hand_2)==1:
                print("next hand")
            elif len(player_hand_2) >= 2:
                quit=True
                stop = True
                score(player_hand)
                score(player_hand_2)
            else:
                score(player_hand)
            quit=True
        elif choice == "sp" and len(playerhand)==2 and
playerhand[0]==playerhand[1]:
            player_hand_2.append(player_hand.pop())
            hit(player_hand)
            quit_2 = True
        elif choice == "d" and len(playerhand) <= 3 and len(player_hand_2)<1:
                hit(playerhand)
```

```
                print("Hand total: " + str(total(playerhand)))
                if total(player_hand)>21:
                    money -= bet*2
                    losses += 2
                else:
                    score(player_hand);score(player_hand)
                quit=True
        elif choice == "q":
            print(wins,losses)
            quit=True;quit_2=True
            exit()

def The_Great_Reset(bankroll):
    global profits
    profits += bankroll;profits -= float(initial_bank)
    return float(initial_bank)

diddle = 1;hand = 0;bet = 0;true_count = 0
if __name__ == "__main__":
    while diddle == 1:
       worksheet.write(hand+1 ,0,hand)
       worksheet.write(hand+1 ,1,money)
       worksheet.write(hand+1 ,2,bet)
       worksheet.write(hand+1 ,3,wins)
       worksheet.write(hand+1 ,4,losses)
       worksheet.write(hand+1 ,5,true_count)
```

```
        hand += 1
        game()
        play_again()
        if money < 0.5*float(initial_bank) or money > 2*float(initial_bank):
            money = The_Great_Reset(money)
    worksheet.write(2,7,profits)
    workbook.close()
```

The algorithm of which the maximum value of b can be found:

```
b_max = 0
for q in range(75,313):
    d = q
    for w in range(97):
        tens = w
        if tens > d:
            tens = d
        for e in range(25):
          elevens = e
          if elevens + tens > d:
              elevens = d - tens
          b_d = ((elevens-1)/(d-2))*((tens-1)/(d-3))+((tens-1)/(d-2))*((elevens-
1)/(d-3))
          b_2 = (elevens/d)*(tens/(d-1))+(tens/d)*(elevens/(d-1))
          #b = 1 * (1 - (b_2 * (1 - b_d))) + 1.5 * (b_2 * (1 - b_d))
          b = 1-0.5*b_2*(b_d-1)
```

```
            if b > b_max:
                b_max = b
                best_d = d
                best_elevens = elevens
                best_tens = tens
print(b, b_max, b_2, b_d)
print(best_d, best_tens, best_elevens)
```