Monte Carlo Comparison of Strategies for Blackjack

Author
Atul Vaidyanathan

Advisor
Peter Winkler

Undergraduate Thesis
Department of Mathematics
Dartmouth College

May 23, 2014

Abstract

Blackjack, one of gambling's most prominent card games, has been mathematically analyzed to derive its Basic Strategy, the optimal strategy for players, given a particular set of rules. Although this process has been proven to be effective, it begs the question as to whether one can obtain an optimal strategy from experience. Can a gambler with no experience and no prior knowledge of Blackjack strategy eventually arrive at a strategy that produces results similar to those of Basic Strategy? Can a gambler learn this strategy by experiencing wins and losses through repeated simulations? With the use of the Monte Carlo Method, this paper provides a comparison between an inexperienced gambler (Simple Strategy), an experienced gambler (Basic Strategy), a "cheating" gambler (Counting Cards), and a learning gambler (Monte Carlo Strategy) to ultimately conclude that this learning strategy can produce results close to Basic Strategy.

Introduction

Upon first glance, Blackjack is a card game with simple rules. A player tries to get a score as close to 21 as possible without overshooting it, while the dealer plays no strategy and keeps adding cards until reaching a score of at least 17. As long as neither scores above 21, the higher score wins. But behind this game is a series of strategies that have been developed over time to ensure maximum success to card players. Often, these tactics are unknown to beginners and fully appreciated by professionals. They have been combined together to formulate the most established method of playing Blackjack, known as *Basic Strategy*[1]. Although some components of Basic Strategy have been derived from mathematical reasoning, a majority of its development can be attributed to the simple method of trial and error. So the question is this: Can a gambler with no experience and no knowledge of Blackjack learn Basic Strategy over time? In other words, can a simulation replicate a beginner gaining experience from playing Blackjack and ultimately arrive at results somewhat resembling Basic Strategy?

To determine whether this is feasible, this paper will first explore the game of Blackjack. Despite the clear-cut stance that this paper has taken towards reaching the goal of Basic Strategy, there are caveats to the rules of Blackjack that need to be explained before reaching the strategies themselves. This section will cover rules, strategy, other approaches to Blackjack, and the derivation of Basic Strategy.

This will lead into the methodology used to achieve the desired learning algorithm in this paper, known as the Monte Carlo Method. The Monte Carlo Method makes use of repeated simulations to obtain a distribution of results from which one can derive the appropriate decision

[1] Peter A. Griffin. *The Theory of Blackjack: The Compleat Card Counter's Guide to the Casino Game of 21*, Sixth Edition Indexed. (Las Vegas, NV: Huntington, 1999), p. 12.

for a gambler to make[2]. By running enough simulations, the results will be similar to a gambler experiencing wins and losses over time, and learning from his successes and mistakes. This paper will go on to explain the Monte Carlo Method in more detail, along with its application within the game of Blackjack.

After explaining the methodology and objective, this paper will explore the simulations themselves. The simulations have been created to replicate a novice gambler (Simple Strategy), an experienced gambler (Basic Strategy), a gambler who can perfectly count cards (Counting Cards), and a gambler who learns from experience (Monte Carlo Method). The expectation is that by using the Monte Carlo Method, the gambler who learns from experience will have a win probability that approaches but does not exceed that of the gambler who plays Basic Strategy.

Perhaps the most interesting viewpoint of Blackjack is from a game theory perspective. Rather than playing against other players on the table, Blackjack is structured such that players play against the house. The house plays no variant strategy, only adding cards until it reaches a score of 17 or above. If the house overshoots 21, the player wins. As a result, the primary strategy of interest is that of the player. The scope of this paper will be limited to only one player against the dealer. Although the introduction of multiple players does not affect gamblers in the long run, this paper will stick to one player for the sake of simplicity[3].

Introduction to Blackjack

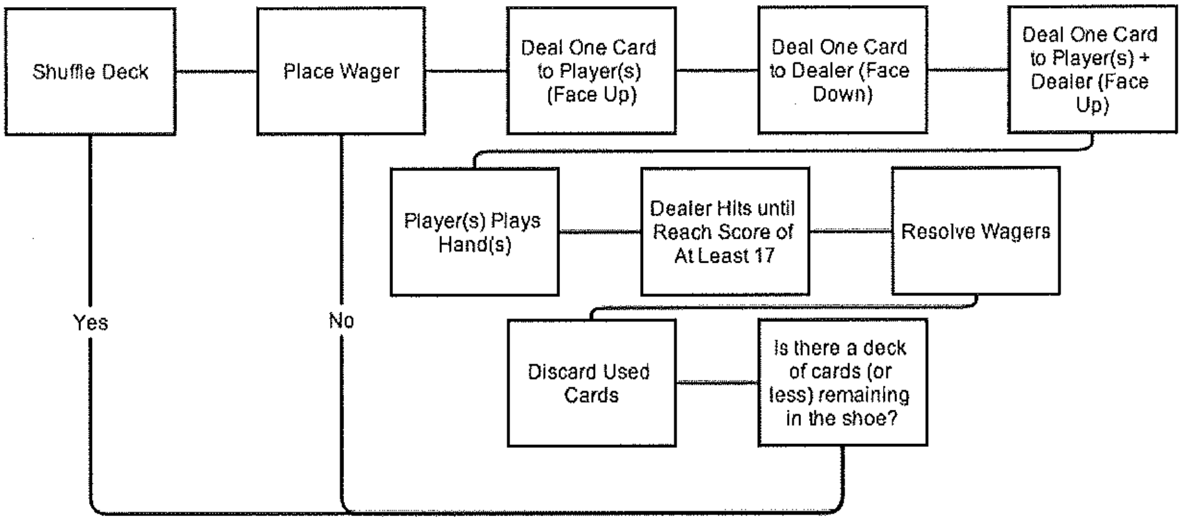
The game of Blackjack was not created by a single individual. The clearest evidence suggests that Blackjack came from French casinos in 1700, although there are records of a

[2] Nicholas Metropolis. "The Beginning of the Monte Carlo Method." *Los Alamos Science* Special Issue (1987): 127

[3] Michael Shackleford. "Blackjack - FAQ." *The Wizard of Odds* (accessed May 21, 2014); available from http://wizardofodds.com/ask-the-wizard/blackjack/.

similar gambling game dating as far back as Ancient Rome[4]. In France, the game was called *Vingt-et-un*, or literally translated "Twenty-One", and cycled through different variations of the game until it was brought to the United States[5]. The game was renamed Blackjack after one of the variations of the game (involving the Ace of Spades and either the Jack of Clubs or Spades), although the rules of that variation were discarded[6].

The structure of the game of Blackjack is as follows:

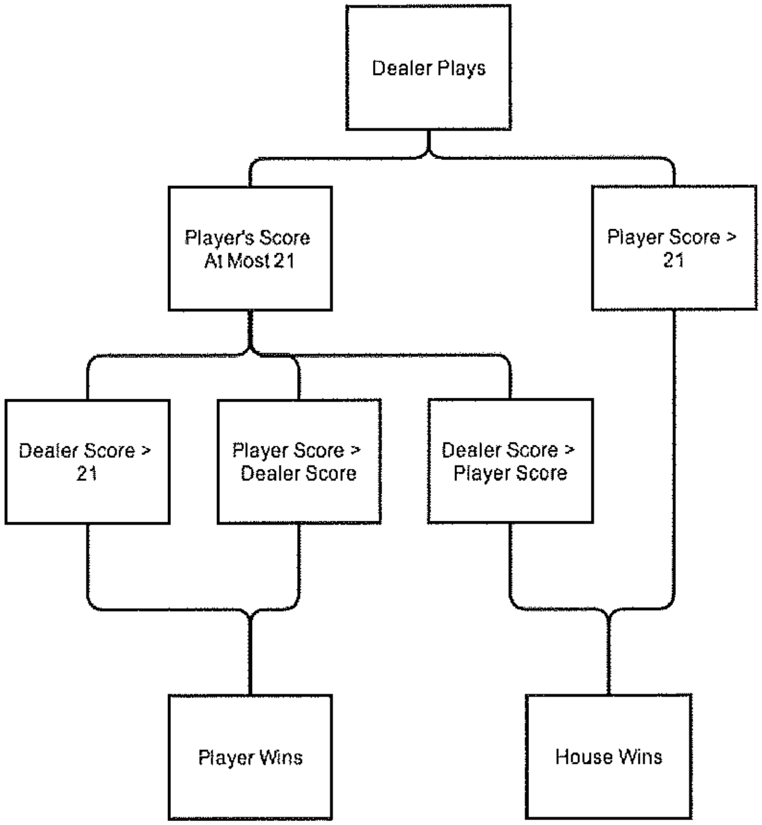

Shuffle Deck
Place Wager
Deal One Card to Player(s) (Face Up)
Deal One Card to Dealer (Face Down)
Deal One Card to Player(s) + Dealer (Face Up)
Player(s) Plays Hand(s)
Dealer Hits until Reach Score of At Least 17
Resolve Wagers
Discard Used Cards
Is there a deck of cards (or less) remaining in the shoe?
Yes
No

The goal of Blackjack is simple. Players bet on each hand, gambling that they will beat the dealer. Players want to score as close to 21 as possible without overshooting it. A large deck, of usually 6 decks[7], is shuffled after which players place wagers on the hands that they expect to play. Cards are then dealt, where the player's cards are both visible, while only one of the dealer's cards can be seen. The player plays his hand, after which the dealer adds cards from the shoe until he reaches a score of at least 17. Then the wagers are resolved and the cards from the

[4] Simon Wintle and Adam Wintle. "History of Blackjack." *The World of Playing Cards* (accessed May 21, 2014); available from http://www.wopc.co.uk/history/blackjack/blackjack.html.
[5] Wintle and Wintle
[6] Wintle and Wintle
[7] Casinos use deck sizes ranging from 4 to 8 decks, but 6 decks is the typical size used.

hand are discarded. If there is about a deck left in the shoe[8], the cards are reshuffled (along with the discarded stack of cards), after which wagers may be placed.

Below is a chart to properly describe how wagers are played out:


Dealer Plays
Player's Score At Most 21
Player Score > 21
Dealer Score > 21
Player Score > Dealer Score
Dealer Score > Player Score
Player Wins
House Wins

As visible above, once the dealer reaches at least 17, the player and dealer compare cards.

The house wins if either:

→ The player goes above 21.

→ The dealer has a score higher than the player.

The player wins if he maintains a score of at most 21 **and** either:

→ The dealer goes above 21.

→ The player has a score higher than the dealer.

[8] The "shoe" is a gaming device that holds the large stack of cards from which cards are drawn.

If the scores are the same, the hand is a "Push" and the player keeps his original wager. Cards are assigned a value equivalent to their numbers, with the face cards (King, Queen, and Jack) having a value of 10 as well. An Ace has the unique position of having values of 11 or 1, whichever gives the highest score without overshooting 21.

A player has the option to play four different moves during his turn:

1) Hit - A player can "hit", or add another card from the shoe to his hand if he thinks that that adding another card will keep him below 21 and give him a higher chance of winning. The player's turn is not over after a hit, unless the hit results in a "bust"[9].

2) Stay - A player can "stay" if he is satisfied with his hand. This means that the player does not want to add anymore cards from the deck and believes that the addition of another card will result in him busting. The player's turn is over after he stays.

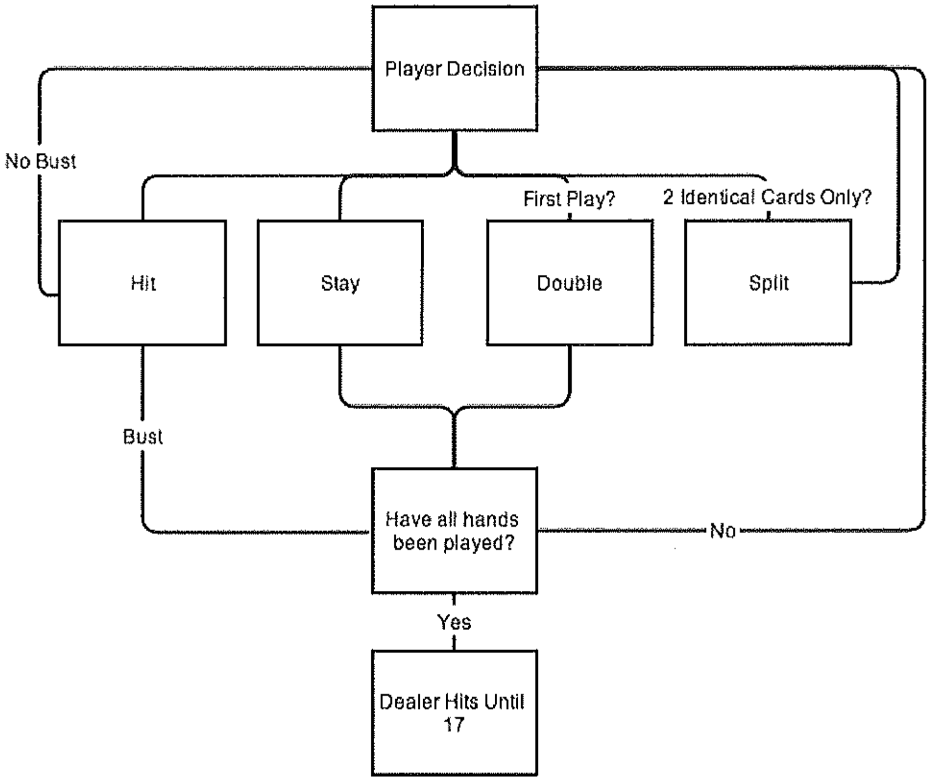
3) Split - A player may "split" if there are two identical cards only in the hand. Splitting a hand will result in the cards being placed in two separate hands, with an additional card from the shoe added to each hand. The amount of the wager on the initial hand is added to the second hand, and each hand is treated separately as a brand new hand. The player's turn is not over after a split.

4) Double - A player may "double down"[10] if he thinks he is likely to win with one additional card and thus would like to double the wager. The player doubles the amount of the initial wager, adds exactly one more card, and then stays. The player may double down only if it is his first turn with the hand, and his turn is over afterwards.

Below is a chart to properly describe player decisions:

[9] A player "busts" if he has a score above 21.

[10] When a player seeks to "double", the proper terminology is "the player wants to double down" or "the player doubles down".



This is the standard gameplay of Blackjack. In addition to these base rules, these simulations will be incorporating some other rules. First, the dealer has to stay if he reaches a "Soft 17". Namely, if the dealer has a score of 17 or 7 (because an Ace is involved), he must stay. In comparison, some casinos enforce a "Hard 17", where if the dealer has a score of 17 or 7, it is considered a 7 and the dealer must hit again[11].

In addition, the simulations will assume that a blackjack pays 3 to 2 odds. In most casinos, if a player gets a blackjack (achieves a score of 21 on the two cards he is dealt without adding any extras - the player must have an Ace and either a face card or a 10), he wins 1.5 times the amount he bets[12]. However, this is conditional on the house not having a blackjack. If they both have blackjacks, the hand could be a Push and the player won't make any money. Because

[11] Kenneth Smith. "Casino Blackjack: Rules of the Game." *BlackjackInfo.com* (accessed May 21, 2014); available from http://www.blackjackinfo.com/blackjack-rules.php.

[12] Smith

of this, dealers often offer players “Even Odds”. A player will take even odds if he thinks that the dealer might have a blackjack (based on the one card he can see). The result of this is that the player just wins the amount he bets without gaining the extra 50% because of his blackjack. In these simulations, it will be assumed that a player takes the even odds whenever he is offered it.

Furthermore, the simulations will not allow players to take insurance. If a dealer’s upcard is an Ace, a player might have the option to insure against the dealer having a Blackjack. The player puts in a side wager equal to half of the player’s original bet. In case the dealer has a Blackjack, the player does not lose any money, since insurance pays twice of the side bet, which cancels out the loss of the initial wager[13].

Finally, the simulations will not allow non-alike face cards to be split. In blackjack, the player has the ability to split two of a kind into separate hands. Then, the player adds one card to each of those hands, adds the same money on his previous hand to the new hand, and treats both hands as completely new (and separate) hands. However, some casinos play a rule that if you have two dissimilar face cards or a 10 and a face card, you can split the cards since they have the same value. This rule grants an advantage to players, especially those who are counting cards[14]. This simulation will not be incorporating this caveat, and thus only allows identical cards to be split.

Now that the rules have been covered, the next important component of Blackjack is Basic Strategy. Contrary to popular belief, there exist a number of Basic Strategies when playing Blackjack. A Basic Strategy (BS) is a list of decisions that are intended for a player to make when playing a hand of Blackjack. It must provide a list of optimal decisions for every possible

[13] Smith

[14] See section titled "Counting Cards".

hand that a player has against every possible upcard for the dealer. For any given set of Blackjack rules, there is only one correct BS[15].

How does one go about discovering a Basic Strategy given the set of rules provided? There exist two main methods. The first is called the *Combinatorial Approach*. The Combinatorial Approach is based on calculating statistical probabilities and expectations for different decisions[16]. Depending on whether the player chooses to hit, stand, split, or double down, the combinatorial approach should provide different expected values. The correct decision is the one that yields the highest expectation. For example, if the player were to get a hand of (5,5), the combinatorial method should provide the probabilities of adding any possible card and the respective expected value of hitting, splitting the cards (and the individual probabilities for each hand), the expected value of staying, and the expected value of doubling down. Often the calculation of these individual hands could be irrelevant because of the statistical unlikelihood of the hands, but a strong program should take all these hands into account[17].

Below is the basic outline of how a Combinatorial Approach would proceed[18]:

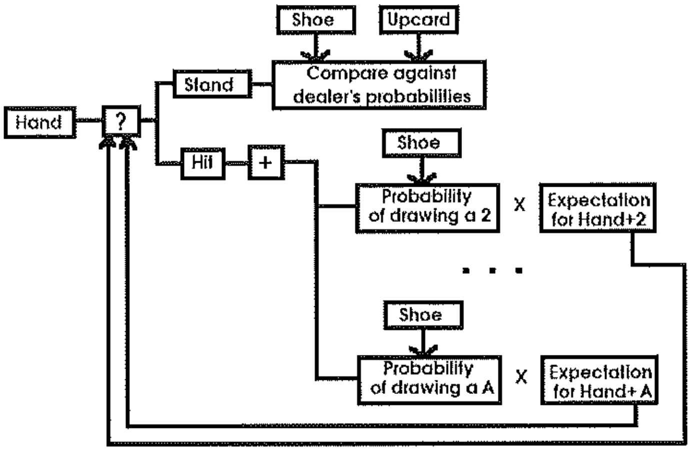
[15] Hoyte Blackjack Labs. "The Derivation of a Basic Strategy." *Hoyte Blackjack Labs: The Derivation of a Basic Strategy* (accessed May 21, 2014); available from http://www.hcsw.org/hbjl/deriv.html.

[16] Hoyte Blackjack Labs

[17] Hoyte Blackjack Labs

[18] Image courtesy of Hoyte Blackjack Labs.

How Basic Strategy Is Determined - Combinatorics


Shoe
Upcard
Stand
Compare against dealer's probabilities
Hand
?
Hit
+
Shoe
Probability of drawing a 2
x
Expectation for Hand+2
. . .
Shoe
Probability of drawing a A
x
Expectation for Hand+A

HBJL (C) 2004 Hoyte Blackjack Labs

In each case, the probability of adding each available card in the deck is calculated. This, in turn, derives the expected value of each remaining card to be drawn, as does the overall expected value of hitting. Then the expected value of the dealer's hand (and additional probabilities) is calculated and set to the expected value of staying. After this, the option of hitting versus staying is compared and whichever provides a higher expected value is the selected decision. Ultimately these probabilities are extended over numerous simulations and different hands, which result in a Basic Strategy for a player, given the specific set of house rules.

This method is mathematically intensive and accurate but ultimately requires high computability to provide accurate outputs. This is one of two main approaches that can be used to create a Basic Strategy. The other method is called the *Simulation Approach*. It is a methodology similar to the Simulation approach that will serve as an integral component of this paper.

Introduction to Monte Carlo

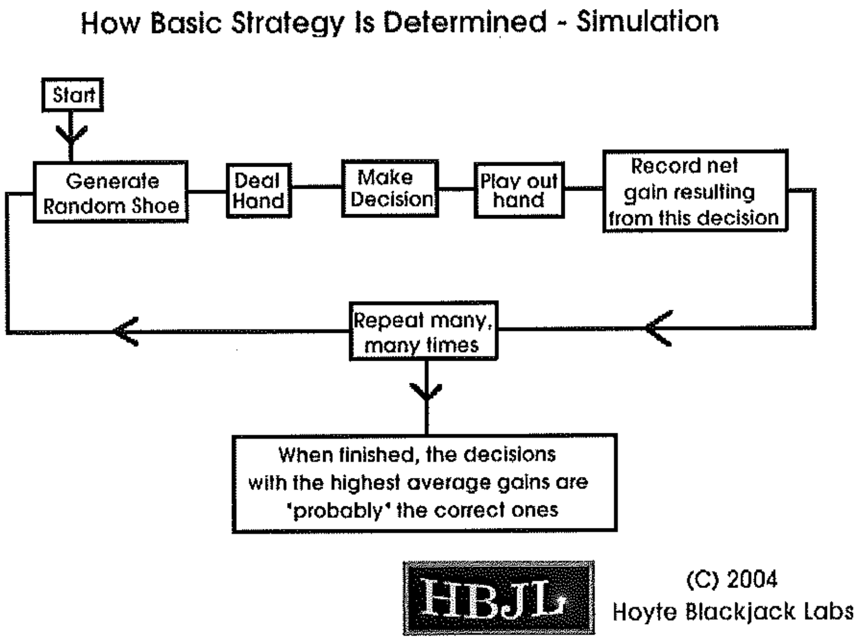
The Monte Carlo Method is aptly named after the city in Monaco, given the prevalence of casinos in the area[19]. Dating back to 1944, the method refers to a broad group of experiments that are dependent on repeated statistical sampling to provide approximate solutions[20]. The name comes from the resemblance to playing and recording results in a real casino. Through repeated results in different simulated scenarios, one is able to determine an ideal decision when presented with a particular situation.

There are three main uses of the Monte Carlo Method: Integration, Optimization, and Inverse. The MC *integration* method refers to numerically computing a definite integral using random numbers. Although deterministic integration is useful when there are few dimensions, the Monte Carlo integration method is useful when the integration has many variables (multidimensional)[21]. The MC *optimization* method involves minimizing or maximizing the optimal paths or solutions based on available data[22]. One runs a series of simulated paths and eventually selects the path of decisions that provides the optimal solution. The MC *inverse* method refers to defining a probability distribution that combines prior information and new information through the use of some definable data[23]. Because of the broad class of algorithms that exist within these Monte Carlo methods, the applications are endless, stretching from biology to artificial intelligence, and from engineering to finance and business.

[19] Metropolis, p. 127.
[20] Metropolis, p. 128.
[21] Helmut Katzgraber. "Introduction to Monte Carlo Methods." Diss. Texas A&M U, 2011, p. 3.
[22] Gilles Guillot. *Monte Carlo Optimization Methods*. Technical University of Denmark (accessed May 22, 2014); available from http://www2.imm.dtu.dk/courses/02443/slides2014/optim_HO.pdf.
[23] Klaus Mosegaard and Malcolm Sambridge. "Monte Carlo Analysis of Inverse Problems." Inverse Problems 18.3 (2002): R29-54.

In the case of Blackjack, the Monte Carlo methods do not apply independently of one another. It could be considered an optimization problem, but the issue is that the game tree is highly extensive and volatile[24]. Attempting to narrow down a solution would require extensive computing power. It could be considered a probability distribution problem, but there is no collection of pre-existing models to add the probability distribution to, since the simulations are generated from a clean slate. As a result, the Blackjack problem is best considered a hybrid between an optimization problem and an inverse problem. The simulation will try to optimize the player's payoffs, for a set of available decisions, by amending the probability matrix with incoming data from completed hands. This will thus create an optimal Basic Strategy matrix.

So will the Monte Carlo simulations be similar to how typical simulations in blackjack are run? This is how simulations are typically used to determine a Basic Strategy[25]:


How Basic Strategy Is Determined - Simulation
Start
Generate Random Shoe
Deal Hand
Make Decision
Play out hand
Record net gain resulting from this decision
Repeat many, many times
When finished, the decisions with the highest average gains are 'probably' the correct ones
HBJL

(C) 2004
Hoyte Blackjack Labs

[24] A game tree refers to a list of decisions available in a non-cooperative game. It specifies the moves available to each player, the players' knowledge during each move, and the players' payoffs after completing their decisions.

[25] Image courtesy of Hoyte Blackjack Labs.

Instead of calculating statistical probabilities like the Combinatorial Approach, this methodology involves more randomness. It takes into account the fact that no two hands are alike because of the randomness of the shoe, as well as the dealer's hand, and thus calculates likelihood rather than expected values. After repeating the simulations numerous times, the decision that yields the highest chance of victory is "probably" the correct one.

The incorporated methodology in this paper varies from the basic simulation approach. Rather than generating a random decision and recording the results, the decisions made by the player will depend on a probability matrix created from past decisions. By amending this matrix in real-time rather than analyzing randomized results, the simulation program will be quicker and more accurate in reaching Basic Strategy. It is this updating of the probability matrix that replicates the inverse method. At the same time, the programs will be running repeated simulations to reach an optimal decision matrix, similar to the Optimization Method. Furthermore, the simulations will incorporate elements of the regular simulation, thus incorporating a weighted randomness into the methodology. The details of this Monte Carlo methodology, along with the other types of simulations, will be explored in the *Simulations* section.

Simulations

As described earlier, there are four different simulations. The simulations will replicate a beginner gambler, an experienced gambler playing Basic Strategy, a gambler who can perfectly count cards, and finally a gambler who can learn from experience over time. The convenient thing about these different simulations is that they are all organized in a very similar manner. As a result, the paper will first cover concepts and functionality that remains similar throughout all of the simulations.

The best way to fully explain these simulations is to walk through them step by step. First, each simulation will generate a random shoe of cards to play blackjack with. Most casinos play with 6 decks (or 24 suits) so the deck will generate the list of cards and then use a shuffle function to incorporate randomness. The result is a randomly shuffled deck. Every time the player or dealer wants to add a card, the first element of the shoe is popped off the deck and added to the player. Every time the deck has a certain threshold of cards remaining (about 1 deck left), it is reshuffled, to simulate protocol in a real casino.

There exists a list of the strategies used by a player during each hand. For example, if a player chooses to hit twice and then stay given a certain set of cards, then the list of strategies will say "Hit", "Hit", and "Stay". Ultimately it is the last strategy used by the dealer that is recorded as part of the results, but each hand's total strategy is temporarily recorded. If a player were to split his cards into two separate hands, each hand is treated individually and the strategies are then split into two. After the round is over, *i.e.* the dealer has played his hand, the last strategy used by the player is recorded and the rest of the previous decisions are discarded.

The results of every hand after every decision is recorded in the following format: (Player score, Dealer Score, Net Gain by Player, Strategy Employed). If a player busts on a hand, the player's score prior to the additional card is recorded, the net gain is -1 (based on the standard wager of 1 dollar), and the strategy employed is hit. As later employed by the MC strategy, this methodology is very convenient in terms of updating a probability matrix.

Each program will incorporate a score-calculating function, a deal function, a hit function, a split function, and a stay function. In the case of a split, additional cards are added to each hand, and the hands are dealt with sequentially. After all simulations have been run, the program calculates out the total number of hands played (splits count as 2 hands), the net gain, the percentage of hands won, and the net gain percentage. It is by this format that all simulations have been created to ensure regularity.

Simple Strategy

The first simulation is designed to replicate a novice Blackjack player, similar to the beginner state of the Monte Carlo simulations. This player will need to have tendencies similar to that of an inexperienced gambler. For starters, beginner gamblers have a tendency to play with the cards they have, rather than playing against the dealer[26]. For example, players will have a tendency to stay when they have 16, even though they see that the dealer is showing an Ace (in this case, the proper strategy is to hit). Or players will split when they have two 10s and they see that the dealer has a 6, with the hopes of winning two separate hands (in this case, the proper strategy is to stay). It becomes clear that beginner blackjack players play with significant randomness attached to their strategy. As a result, the strategy incorporated into the simulation includes one of randomness.

The full code for the Simple Strategy simulation can be found in Appendix 1. The simulation follows the pattern explained in the "Simulation" section, and displays its uniqueness in the function *reg_sim*. All blackjack players know to hit on a score of 11 or lower. It is almost always the case the players choose to stay on a score of 18 or higher. The simulations reflect this strategy for beginner players as well. However, from the range of 12 to 17, blackjack players do not always have a right strategy. While beginners do play with randomness, it is definitely more likely for a beginner to hit on a 12, than to hit on a 17. As a result, the decisions for beginner players are weighed accordingly. The best way to attribute some weighted randomness is by

[26] HitOrSplit.com. "Top Ten Most Common Blackjack Mistakes That Cost You Money." HitOrSplit.com (accessed May, 21 2014); available from http://www.hitorsplit.com/articles/Top_Ten_Most_Common_Blackjack_Mistakes.html.

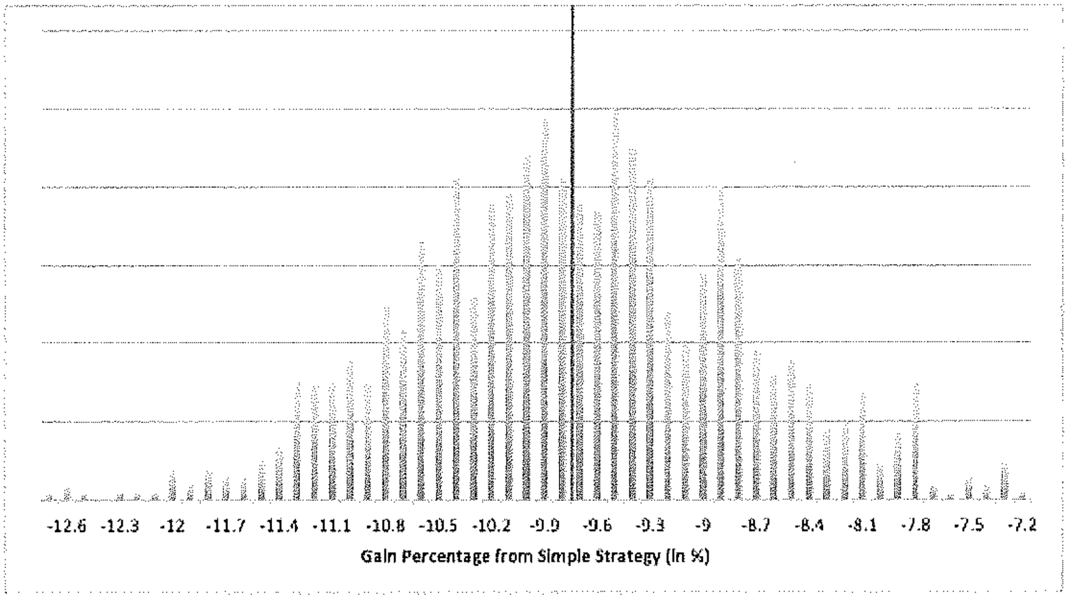
creating a list of possible plays, with more likely plays added multiple times to the list, and then randomly selecting one. This is the methodology pursued in the Simple Strategy case.

To start, the player checks if the hand can be split. If the hand can be split, this option is added to the list of possible plays. Then the player checks if the score is above 17 or less than 12. If the player's score qualifies as either, the list of strategies adds the respective move and randomly select from the list (either one or two options). This methodology is intended to replicate the idea that a beginner player might split even if he has a clear-cut win in hand, with the hopes of winning multiple hands.

If the player's score sits in the range of 12 to 17, then the player scraps the list of player decisions and starts over. If the player's score is in this range and the player can split, then the option of splitting is added 3 times to the list of plays. Then, the simulations subtract the player's score from 18 and add that number of "hit" options to the list. They does the same for the option to "stay", except it subtracts 11 from the player's score and add that number of "stay" options to the list. As long as the player's score sits between 12 and 17, there are 10 options in the list itself. Through this method, the simulation creates a 30% chance of splitting the hand, but also creates a weighted distribution for "hits" and "stays" for the player. This is intended to replicate the skewed randomness that novice blackjack players tend to display.

If the player's score is in the range of 12 to 17 and the player cannot split, then the only options are to hit or split, which are weighed accordingly. The breakout is as follows: A player with a score of 12 hits about 90% of the time, a score of 13 hits about 70% of the time, a score of 14 hits about 60% of the time, a score of 15 hits about 40% of the time, a score of 16 hits about 30% of the time, and a score of 17 hits about 10% of the time. Although these numbers may not provide an exact replication, it allows one to conceptualize how randomly novice blackjack players are capable of playing.

After this, the simulation creates randomly generated hands and runs them against the Simple Strategy. After running a large number of simulations (in this case 10,000), the program adds up the net gains and losses to calculate the gain percentage. Then, in order to provide an accurate distribution of what the gain percentage can look like, the program runs 1,000 more sets of these 10,000 simulations, giving a series of gain percentages. This data creates a frequency distribution curve for the simulations, which can be found below.



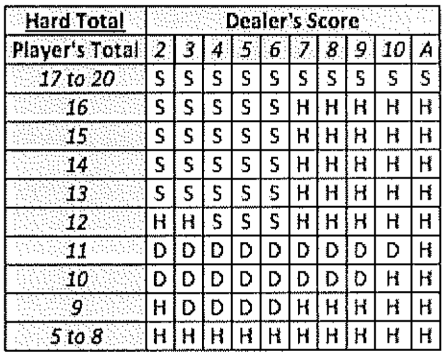
Ultimately, the results give a distribution with the mean at around -9.73% (blue line) and a standard deviation of .94. If this were a perfect Gaussian distribution[27], the average gain percentage for a novice gambler playing a "Simple Strategy" would sit between -11.6% and - 7.6% about 95% of the time. Although this is not a perfect distribution, it is clear that this is somewhat reflected by the graph above.

[27] A Gaussian distribution is also informally known as a bell curve.

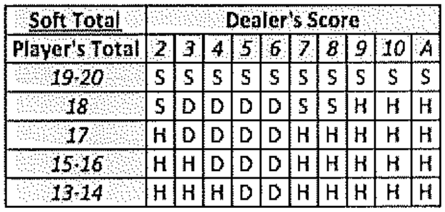
This replication of a novice gambler makes it apparent that this strategy is not sustainable. With an average loss of 10%, the expectation is that the Monte Carlo Method's gain percentage will significantly exceed that of the Simple Strategy.

Basic Strategy

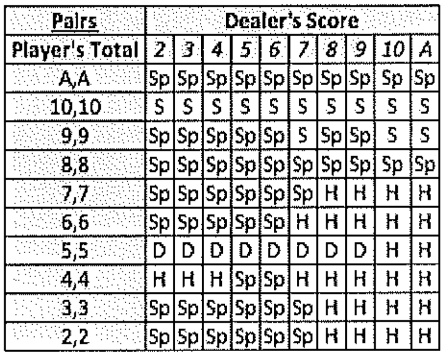
This simulation is designed to replicate the strategy of an experienced gambler familiar with Basic Strategy. As discussed earlier, each set of Blackjack house rules implies that for every situation faced, there is an ideal Basic Strategy. For example, some casinos play regular odds for Blackjack, a "Hard 17" requirement for the dealer, and allow the splitting of non-alike face cards. As discussed earlier, the casinos in these simulations provide a 3:2 odds for a Blackjack (and non-dealer Blackjack), a "Soft 17" requirement, and do not allow the splitting of non-alike face cards. For these house rules, this is the Basic Strategy:



Hard Total	Dealer's Score									
Player's Total	2	3	4	5	6	7	8	9	10	A
17 to 20	S	S	S	S	S	S	S	S	S	S
16	S	S	S	S	S	H	H	H	H	H
15	S	S	S	S	S	H	H	H	H	H
14	S	S	S	S	S	H	H	H	H	H
13	S	S	S	S	S	H	H	H	H	H
12	H	H	S	S	S	H	H	H	H	H
11	D	D	D	D	D	D	D	D	D	H
10	D	D	D	D	D	D	D	D	H	H
9	H	D	D	D	D	H	H	H	H	H
5 to 8	H	H	H	H	H	H	H	H	H	H



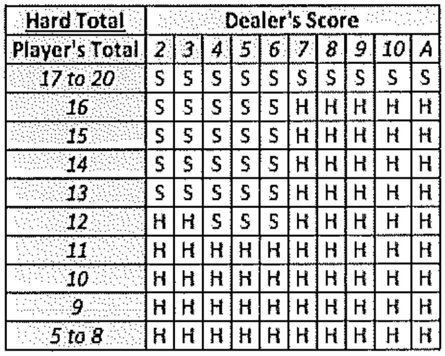
Soft Total	Dealer's Score									
Player's Total	2	3	4	5	6	7	8	9	10	A
19-20	S	S	S	S	S	S	S	S	S	S
18	S	D	D	D	D	S	S	H	H	H
17	H	D	D	D	D	H	H	H	H	H
15-16	H	H	D	D	D	H	H	H	H	H
13-14	H	H	H	D	D	H	H	H	H	H



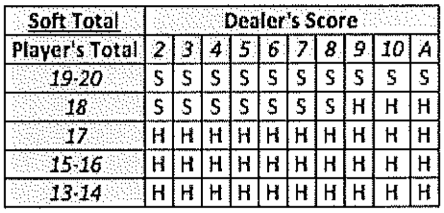
Pairs	Dealer's Score									
Player's Total	2	3	4	5	6	7	8	9	10	A
A,A	Sp	Sp	Sp	Sp	Sp	Sp	Sp	Sp	Sp	Sp
10,10	S	S	S	S	S	S	S	S	S	S
9,9	Sp	Sp	Sp	Sp	Sp	S	Sp	Sp	S	S
8,8	Sp	Sp	Sp	Sp	Sp	Sp	Sp	Sp	Sp	Sp
7,7	Sp	Sp	Sp	Sp	Sp	Sp	H	H	H	H
6,6	Sp	Sp	Sp	Sp	Sp	H	H	H	H	H
5,5	D	D	D	D	D	D	D	D	H	H
4,4	H	H	H	Sp	Sp	H	H	H	H	H
3,3	Sp	Sp	Sp	Sp	Sp	Sp	H	H	H	H
2,2	Sp	Sp	Sp	Sp	Sp	Sp	H	H	H	H

As presented above, the strategy for this Basic Strategy implies that the player has the option to hit (H), split (Sp), stay (S), or double down (D). For example, if a player's hand has a hard total of 14, he should stay if the dealer's card is below 7 and hit otherwise. Or in case a player has an 11, he should double his wager unless the dealer is showing an Ace, in which case his optimal strategy is to just hit. The hope is to replicate this veteran player's strategy.

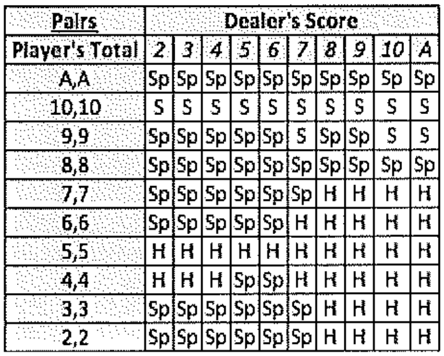
In some casinos, players are not given the option to double down. In the case of the Monte Carlo Method (run later), it will be clear that it is very hard to accurately randomize when a player should double down versus just hitting. This particular Monte Carlo simulation, therefore, does not allow players to double down. Despite the prominence of the "double down" rule, Basic Strategy needs to be modified to this new rule into account. Therefore, the variant Blackjack Strategy will be as below:



Hard Total	Dealer's Score									
Player's Total	2	3	4	5	6	7	8	9	10	A
17 to 20	S	S	S	S	S	S	S	S	S	S
16	S	S	S	S	S	H	H	H	H	H
15	S	S	S	S	S	H	H	H	H	H
14	S	S	S	S	S	H	H	H	H	H
13	S	S	S	S	S	H	H	H	H	H
12	H	H	S	S	S	H	H	H	H	H
11	H	H	H	H	H	H	H	H	H	H
10	H	H	H	H	H	H	H	H	H	H
9	H	H	H	H	H	H	H	H	H	H
5 to 8	H	H	H	H	H	H	H	H	H	H



Soft Total	Dealer's Score									
Player's Total	2	3	4	5	6	7	8	9	10	A
19-20	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	H	H	H
17	H	H	H	H	H	H	H	H	H	H
15-16	H	H	H	H	H	H	H	H	H	H
13-14	H	H	H	H	H	H	H	H	H	H



Pairs	Dealer's Score									
Player's Total	2	3	4	5	6	7	8	9	10	A
A,A	Sp	Sp	Sp	Sp	Sp	Sp	Sp	Sp	Sp	Sp
10,10	S	S	S	S	S	S	S	S	S	S
9,9	Sp	Sp	Sp	Sp	Sp	S	Sp	Sp	S	S
8,8	Sp	Sp	Sp	Sp	Sp	Sp	Sp	Sp	Sp	Sp
7,7	Sp	Sp	Sp	Sp	Sp	Sp	H	H	H	H
6,6	Sp	Sp	Sp	Sp	Sp	H	H	H	H	H
5,5	H	H	H	H	H	H	H	H	H	H
4,4	H	H	H	Sp	Sp	H	H	H	H	H
3,3	Sp	Sp	Sp	Sp	Sp	Sp	H	H	H	H
2,2	Sp	Sp	Sp	Sp	Sp	Sp	H	H	H	H

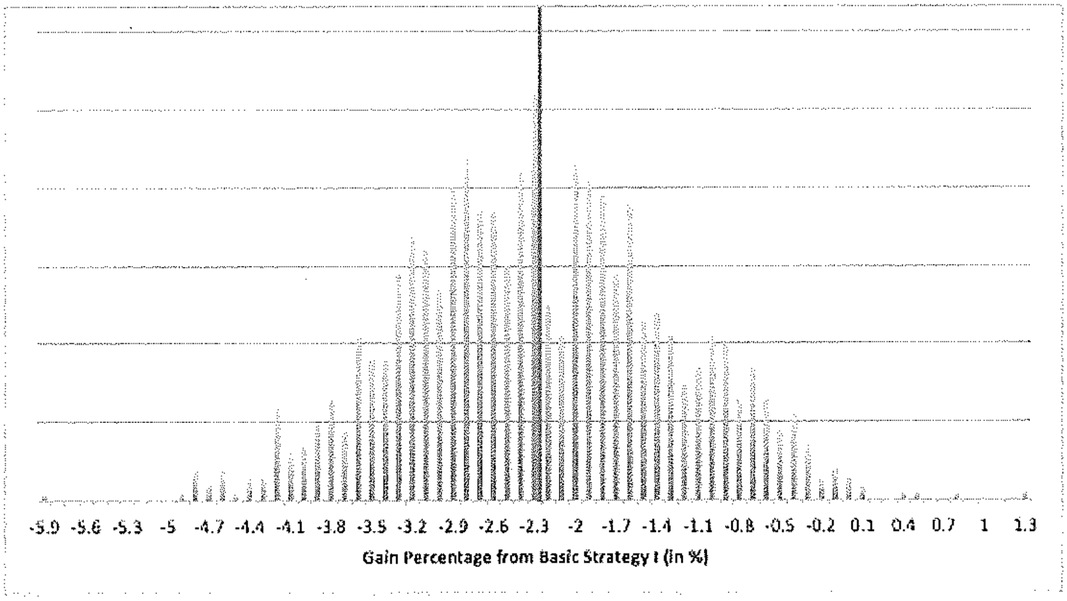
Because of the similarity between the two strategies, these strategies will be labeled separately, as Basic Strategy I (no doubling) and Basic Strategy II. Both strategies are built in a nearly identical manner (with the only difference being the option to double). The gain percentage with the option to doubling down is about 1.6% higher than the option without[28]. As a result, we would expect Basic Strategy II to have a higher gain percentage than Basic Strategy I. The simulations for both strategies can be found in the Appendix. Rather than attempt to list the

[28] Griffin, p. 19.

results together, it would be better to list the results of these simulations separately and explain them.

Basic Strategy I

With the Basic Strategy I simulation, the program runs a methodology similar to the Simple Strategy simulations. It generates 10,000 simulations per gain percentage calculated, and then runs that 1,000 times to produce a distribution curve. The results of the repeated simulation can be found below.

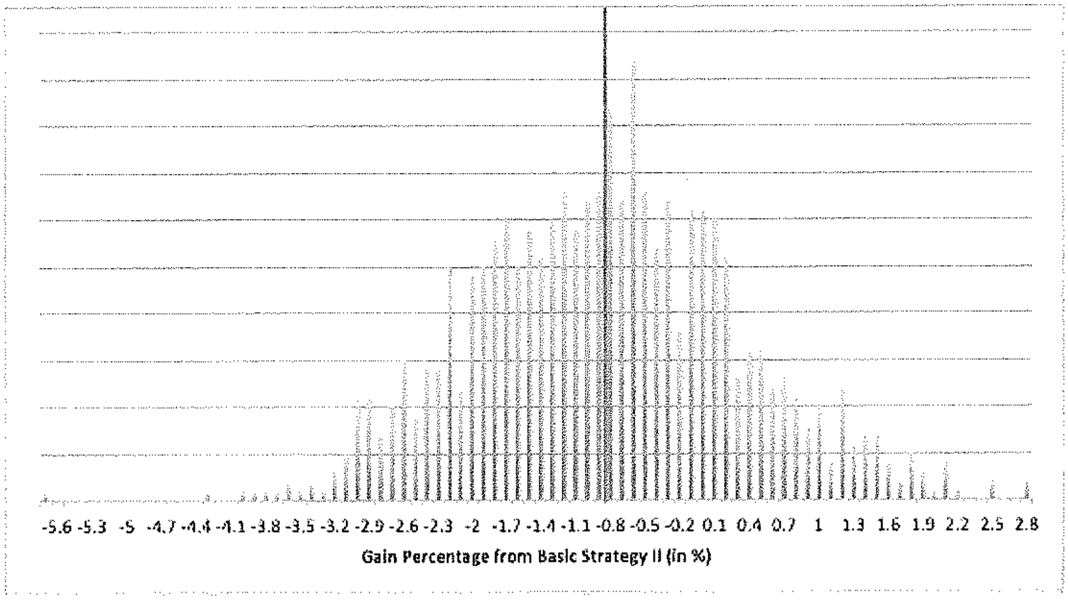


The results give a distribution with the mean gain percentage at around -2.28% (blue line) and a standard deviation of 1. If this were a perfect Gaussian distribution, the average gain percentage for a gambler playing Basic Strategy I would sit between -4.28% and -0.28% about 95% of the time. Similar to the Simple Strategy simulations, the data somewhat reflects these expectations.

The gain percentage of the Monte Carlo Strategy should exceed the gain percentage of Simple Strategy as presented earlier, and is expected to approach but not exceed the gain percentage of the Basic Strategy I.

Basic Strategy II

With an identical methodology to that used in the Basic Strategy I simulation, the results of the Basic Strategy II simulation are below:



The results give a distribution with the mean at around -0.84% (blue line) and a standard deviation of 1.17. A perfect Gaussian distribution would entail the average gain percentage for a gambler playing Basic Strategy II to sit between -3.18% and 1.50% about 95% of the time.

As mentioned above, it is expected that the Monte Carlo Strategy would somewhat approach this strategy, but it is not likely to exceed the gain percentage of the Basic Strategy I. As a result, the gain percentage of Basic Strategy II exceeded the gain percentage of Basic Strategy I by about 1.4% (although expected to be 1.6%), but is expected to remain under the gain percentage of card counting.

Counting Cards

Edward Thorp, an American mathematician, first developed the concept of counting cards[29]. Using computers, he began to examine whether keeping track of the cards could be advantageous to a player. Eventually, he made the discovery that with some mental mathematical dexterity, it was possible to gain an advantage over the dealer[30]. The key component behind counting cards is the wager. If a player is likely to be advantageous in a certain scenario, then he should bet more money. Thorp used a methodology called the Ten-Count (later renamed the Thorp Ten-Count), where a player would select 16 and 36, multiply each by the number of decks being played with, and count backwards by dividing by the number of tens left in the deck to determine the advantage for the player[31]. He published a book titled "Beat the Dealer", starting the craze to beat the odds of Blackjack.

Eventually, other people took to the idea of keeping track of cards in different ways such as the KO Strategy, the Zen Count, the Omega II, etc. The MIT Blackjack team (operating from 1979 until the early 2000s) was famous for its group counting strategy, where players would count together to coordinate high-wagering gamblers[32]. Counting cards, although not cheating, is seriously frowned upon in current casinos (because it allows players an advantage over casinos), and those who get caught often get banned from the casino. However, it provides an interesting mathematical comparison to Basic Strategy and the Monte Carlo Method.

The methodology employed in this paper is known as the Hi-Lo strategy. The Hi-Lo strategy is the most basic card counting strategy, and is contingent on players counting high and

[29] Ofton, Loudon. "The History of Blackjack." *Blackjack Apprenticeship (*accessed May 21, 2104); available from http://www.blackjackapprenticeship.com/resources/history-of-blackjack/.
[30] Ofton
[31] Ofton
[32] Griffin, p. 60.

low cards as they land on the table. By keeping a count of the cards being dealt during a hand, a player can adjust his bets accordingly to increase the expected payoff for each hand. By extending this logic to the series of hands a player faces, he can increase the expectation for the duration he is at the table[33].

The strategy itself is straightforward. For each card dealt, if the value of the card is between 2 and 6, then the count for the card is +1. If the card is a 7, 8, or 9, the count for the card is +0. Otherwise, for cards 10 through A, the count is -1. The player then adds the count for all the cards to form the *running count.* The running count starts at 0 and is maintained between rounds (but is reset if the shoe is reshuffled)[34]. A positive count means that a higher number of small cards have been played, and that the deck possesses higher cards than lower cards.

The running count is used 99% of the time, but is not the final value to gauge a player's wager. The player needs to convert the running count into the *true count.* The true count is determined by dividing the running count by the estimated number of decks left to be played[35]. The true count thus gives the relative proportion of high value cards[36] within the deck. Used in conjunction with Basic Strategy, a higher true count entails that a player should be wagering more.

In the case of this simulation, a true count above +2 will give a wager that is 2 times the standard wager before the next hand is played. Similarly, a true count above +4 will give 3 times the original wager, a true count above +6 will give 4 times the wager, and a true count above +8

[33] Ariell Zimran, Anna Klis, Alejandra Fuster, and Christopher Rivelli. *The Game of Blackjack and Analysis of Counting Cards.* University of Texas (accessed May 21, 2014); available from http://www.annaklis.com/uploads/6/4/7/2/6472295/zimran_klis_fuster_rivelli.gametheory.blackjack.pdf.

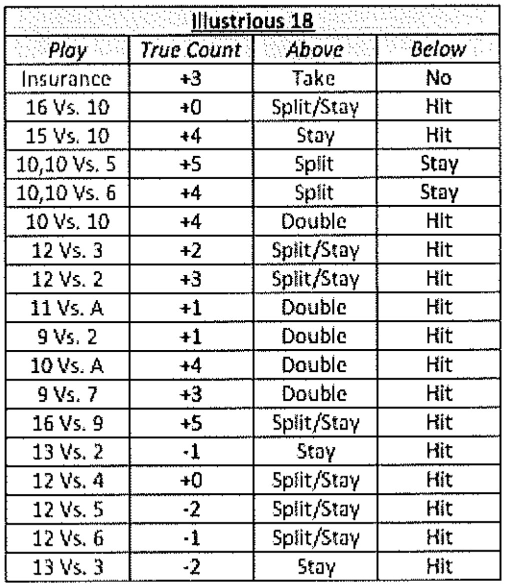
[34] Zimran et al.

[35] Shackleford

[36] High value cards are 10, J, Q, K, and A.

will give 5 times the wager. The simulation then allows the player to play the randomized hand in accordance with Basic Strategy (in this case, Basic Strategy II).

However, the true count is not only used in the wager. In some cases, the true count might affect how an individual should play his hand. Called the Illustrious 18, these are a series of scenarios during which a player should play against Basic Strategy's recommendation. The Illustrious 18 is presented below[37]:

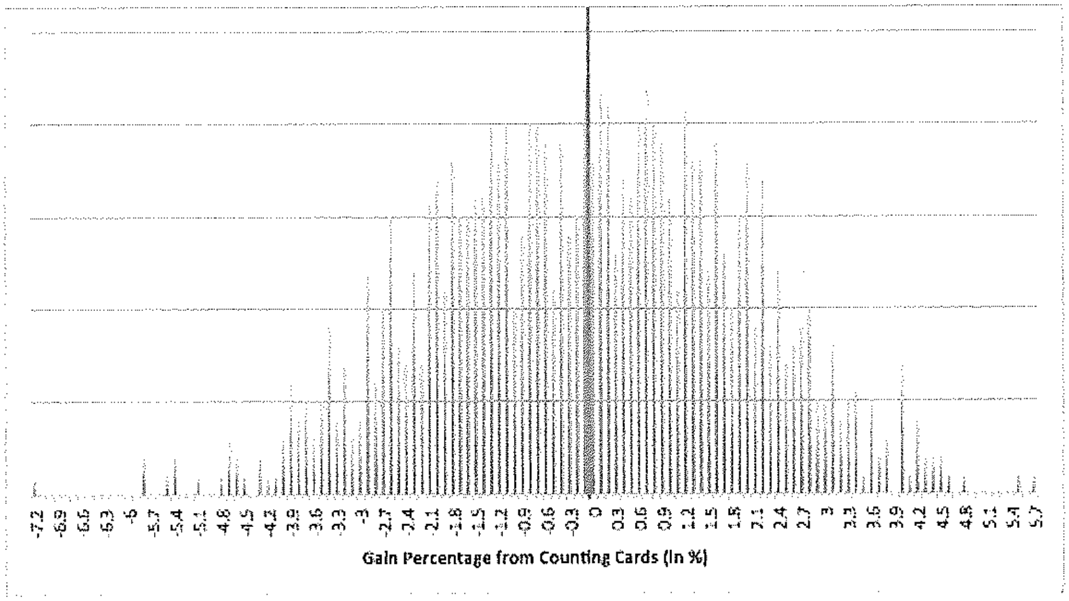


Illustrious 18			
Play	*True Count*	*Above*	*Below*
Insurance	+3	Take	No
16 Vs. 10	+0	Split/Stay	Hit
15 Vs. 10	+4	Stay	Hit
10,10 Vs. 5	+5	Split	Stay
10,10 Vs. 6	+4	Split	Stay
10 Vs. 10	+4	Double	Hit
12 Vs. 3	+2	Split/Stay	Hit
12 Vs. 2	+3	Split/Stay	Hit
11 Vs. A	+1	Double	Hit
9 Vs. 2	+1	Double	Hit
10 Vs. A	+4	Double	Hit
9 Vs. 7	+3	Double	Hit
16 Vs. 9	+5	Split/Stay	Hit
13 Vs. 2	-1	Stay	Hit
12 Vs. 4	+0	Split/Stay	Hit
12 Vs. 5	-2	Split/Stay	Hit
12 Vs. 6	-1	Split/Stay	Hit
13 Vs. 3	-2	Stay	Hit

Since the rules for these simulations do not allow players to take Insurance, it is not an option. Other than that, some of these plays are counter-intuitive, and against Basic Strategy. For example, Basic Strategy recommends that players never split 10s. However, if the true count is above +4 and the dealer is showing a 6, then it is in the player's best interest to split the 10s.

Using the true count for wagers, the Illustrious 18, and Basic Strategy II, the simulations generated a series of gain percentages, the results of which are displayed below:

[37] Shackleford



The results of these simulations give a distribution with the mean at around -0.07% (blue line) and a standard deviation of 1.99. A perfect bell curve would entail the average gain percentage for a gambler who is accurately counting cards to sit between -4.05% and 3.91% about 95% of the time. Unlike the previous simulations where the standard deviations were narrower, the wide range (and thicker bell curve) provides evidence that the gain percentage from counting cards is significantly less predictable when compared to the other strategies.

One would not expect the Monte Carlo Strategy to approach this strategy, since this is the most optimal strategy in Blackjack for this set of rules. However, it is interesting to note that although the mean gain percentage through this simulation is practically 0%, alternative literature suggests that counting cards can often render a higher advantage[38]. The most obvious reason for this variation is the difference in house rules used by this simulation compared to other simulations.

[38] Griffin, p. 19.

Now that all comparative simulations have been covered (namely Simple Strategy, Basic Strategy I and II, and Counting Cards), the final simulation remaining utilizes the Monte Carlo Method.

Monte Carlo Method

As discussed earlier, the Monte Carlo Method employed in this simulation utilizes a combination of the Optimization and Inverse methods. The repeated simulations attempt to optimize a clean-slated probability distribution. By amending the distribution in real time, rather than waiting for the end of every round, the strategy matrix will be improved quickly and more accurately compared to simulation approach that is typically employed.

So how does this simulation work? As discussed earlier, the option to double down will not be included for the player. This leaves three options for the player, namely to split, stay, or hit given a particular hand. However, accounting for the option to split is unique because it is not always available for a given scenario (it requires both cards to be identical). Hence, the ideal strategy for splitting will be taken from Basic Strategy I. This leaves two options for the player, the option to hit and the option to stay. These two options will be weighed against each other to determine which is optimal given a particular scenario.

The simulation starts by creating two 18 (4-21 for player) by 10 (2-11 for dealer) matrices. The first matrix will serve as the total times a player faces a scenario (referred to as the total matrix), while the second matrix will serve as the total times a player successfully hits within a scenario (referred to the hit matrix). The future decisions that the player makes will be based on the ratio between the two matrices.

Because matrices start with index 0, the player's score minus 4 and the dealer's score minus 2 will give the correct cell location. The matrix will operate as follows.

Given the player's score (prior to a hit) is *m* and the dealer's score is *n*:

→ If the player hits but does not bust, then the (m-4, n-2) coordinates for both the total and hit matrices will increase by 1.

→ If the player hits and busts, then the (m-4, n-2) coordinate for the total matrix will increase by 1.

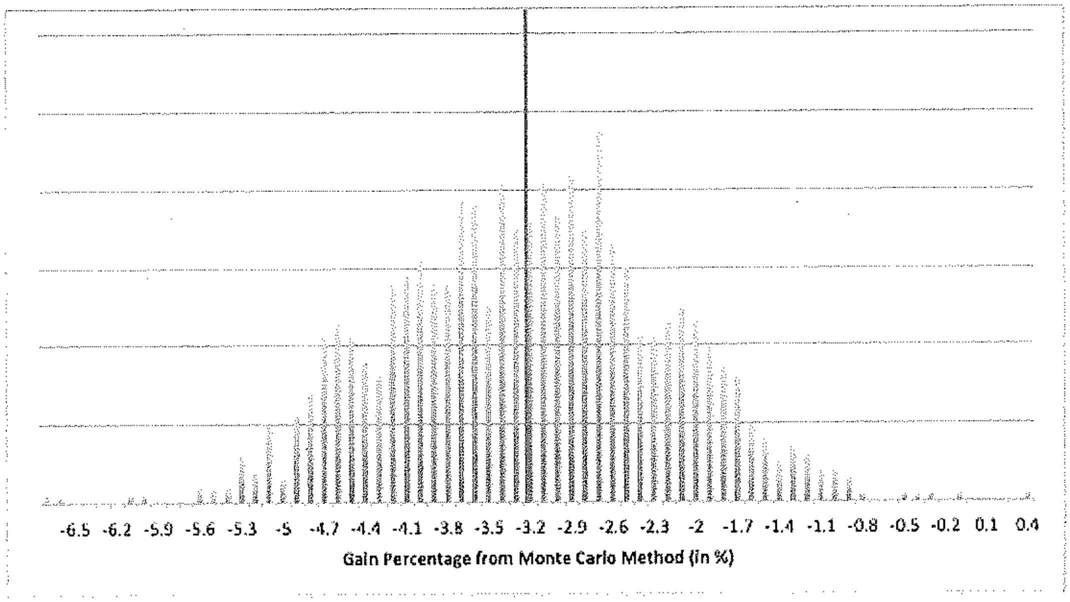
→ If the player stays and wins, then the (m-4, n-2) coordinate for the total matrix will increase by 1.

→ If the player stays and loses, then the (m-4, n-2) coordinates for both the total and hit matrices will increase by 1.

Doing this ensures that a positive outcome from "staying" during a particular scenario increases the future likelihood of staying and decreases the future likelihood of hitting. By extending the logic to include positive and negative outcomes for both hitting and staying, this creates a balanced ratio where the outcome of a hand is able to directly influence future scenarios.

The simulation starts by testing whether it is ideal (and if possible) to split the hand. If the hand should be split, it is, with each hand treated individually. Then, if the player's score is below 12, the player chooses to hit. If the player's score is above 17, the player stays. If neither of these scenarios fit, then the simulation determines whether the situation has been played before, *i.e.* whether the player and dealer have both had their particular scores. If the situation has not been played, then the simulation randomly chooses between staying and hitting. If the situation has been played, the simulation takes the ratio between the (m-4, n-2) cells of the hit and total matrices, giving a decimal between 0 and 1 (labeled *r*). The simulation then generates a random decimal, x. If $x < r$, then the simulation allows the player to hit. Otherwise, the player stays. The simulations continue, continually updating the matrix.

By running a significant number of simulations, this continually updates the matrix until it reaches an optimal equilibrium. To run a methodology similar to the previous examples, it runs 10,000 simulations per set, and run 1,000 sets to create a frequency distribution. The results of the distribution are below.

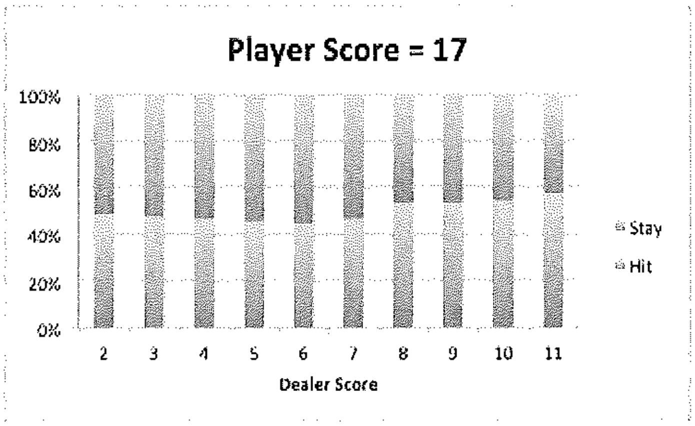


The results of these simulations give a distribution with the mean at around -3.22% (blue line) and a standard deviation of 1. A perfect Gaussian distribution curve would expect the average gain percentage for a gambler who is accurately counting cards to sit between -5.22% and -1.22% about 95% of the time.

As visible from the previous examples, the mean gain percentage from the Monte Carlo Method sits well above the Simple Strategy, and approaches the mean gain percentage from the Basic Strategy I simulation. However, because the frequency distributions between the two simulations overlap, it is difficult to make an accurate comparison between the means. The comparisons between the Monte Carlo and the Basic Strategy I simulation can be found in the Results section.

Results

The comparison between the Monte Carlo Method and Basic Strategy can be done in two ways: through the difference in strategy and the overall outcomes. One of the outputs of the Monte Carlo method is the ratio matrix, which provides the more optimal outcome between hitting and staying given a particular hand. For example, in case the player's score is a 17:



As visible from the graph, the player is better off staying if the dealer is showing below an 8 and better off hitting otherwise. This strategy is different from Basic Strategy, because Basic Strategy suggests that a player always stays with a 17 if they have a hard total, and always hits if they are playing with a soft total. One of the main reasons for this difference is that under Basic Strategy, there is a different strategy between hard and soft totals. The Monte Carlo simulations do not differentiate between soft and hard totals. Inclusion of this differentiation would be ideal, but the incorporation of two additional matrices would result in a doubling of runtime (at least), as well as a definite decrease in accuracy for the matrices.

The exact charts for each player's score can be found in the Appendix. When comparing the Monte Carlo strategy to Basic Strategy I, the preferred strategies are those that are more than 50% likely. As a result, below is the strategy layout for the Monte Carlo Strategy:

Hard Total	**Dealer's Score**									
Player's Total	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*A*
20 to 21	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	**H**	**H**
18	S	S	S	S	S	S	S	**H**	**H**	**H**
17	S	S	S	S	S	S	**H**	**H**	**H**	**H**
16	**H**	**H**	**H**	**H**	**H**	H	H	H	H	H
15	**H**	**H**	**H**	**H**	**H**	H	H	H	H	H
14	**H**	**H**	**H**	**H**	**H**	H	H	H	H	H
13	**H**	**H**	**H**	**H**	**H**	H	H	H	H	H
12	H	H	**H**	**H**	**H**	H	H	H	H	H
11	H	H	H	H	H	H	H	H	H	H
10	H	H	H	H	H	H	H	H	H	H
9	H	H	H	H	H	H	H	H	H	H
5 to 8	H	H	H	H	H	H	H	H	H	H

Soft Total	**Dealer's Score**									
Player's Total	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*A*
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	**H**	**H**
18	S	S	S	S	S	S	S	H	H	H
17	**S**	**S**	**S**	**S**	**S**	**S**	H	H	H	H
15-16	H	H	H	H	H	H	H	H	H	H
13-14	H	H	H	H	H	H	H	H	H	H

The actions in bold are those that are different from Basic Strategy. In comparing the strategies, a majority remains the same. Almost all modifications to the strategy in one table remain unchanged in the other. For example, in the case of 17, the optimal strategy is to always hit with a soft total and never hit with a hard total. In the case of the Monte Carlo Method, it is optimal to hit if the dealer is showing above an 8, and to stay otherwise. It is this merging of strategies that ultimately results in the variation. The only exception to this is the case of a player having a 19. In this case, Basic Strategy recommends that a player always stay while the Monte Carlo Method suggests a player should hit if the dealer is showing a 10 or an Ace.

Now that the strategies have been compared, it is time to turn towards the overall outcomes. The overall outcomes have been discussed above in the individual sections, but it would be ideal to make a comparison between the two. With only the data available, both the true means and the true standard deviations are unknown. As a result, the way to understand the compare the two means is by the formula[39]:

$$t = \frac{Signal}{Noise} = \frac{\overline{x}_1 - \overline{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

The $\bar{x}_1$ and $\bar{x}_2$ values represent the means of the data set, s_1 and s_2 represent the standard deviations, and n_1 and n_2 refers to the sample sizes (1,000 in all samples). The formula gives the output *t*, the t-statistic, which can be used to calculate the statistical significance. The statistical significant allows one to distinguish the means of two data sets. The calculations of the t-statistics can be found below:

Dataset 1	Dataset 2	$\bar{x}_1$	$\bar{x}_2$	s_1	s_2	n_1	n_2	$\lvert t \rvert$
Simple Strategy	Monte Carlo	-9.73	-3.22	0.94	1	1000	1000	149.63
Simple Strategy	Basic Strategy I	-9.73	-2.28	0.94	1	1000	1000	171.3
Simple Strategy	Basic Strategy II	-9.73	-0.84	0.94	1.17	1000	1000	186.9
Simple Strategy	Counting Cards	-9.73	-0.07	0.94	1.99	1000	1000	138.49
Monte Carlo	Basic Strategy I	-3.22	-2.28	1	1	1000	1000	20.86
Monte Carlo	Basic Strategy II	-3.22	-0.84	1	1.17	1000	1000	48.66
Monte Carlo	Counting Cards	-3.22	-0.07	1	1.99	1000	1000	44.55
Basic Strategy I	Basic Strategy II	-2.28	-0.84	1	1.17	1000	1000	29.52
Basic Strategy I	Counting Cards	-2.28	-0.07	1	1.99	1000	1000	31.33
Basic Strategy II	Counting Cards	-0.84	-0.07	1.17	1.99	1000	1000	10.55

In the case of simulations where the number of observations exceeds 120, the absolute value of the t-statistic needs to exceed about 2.58 to be considered statistically significant at the 1% level. Every test performed above is statistically significant, meaning that the results can be interpreted as follows: the difference between the means of any two strategies is statistically significant at the 1% level.

[39] Daniel Katzman, Jessica Moreno, Jason Noelanders, and Mark Winston-Galant. Comparisons of Two Means. University of Michigan (accessed on May 21, 2014); available from https://controls.engin.umich.edu/wiki/index.php/Comparisons_of_two_means.

The t-values are highest for comparisons to the Simple Strategy, which makes sense because its mean was significantly further away from the rest of the simulations. The difference between the Monte Carlo Method and Basic Strategy I is statistically significant, but even more significant is the difference between the two Basic Strategies themselves.

Conclusion

By the end of this paper, it becomes evident that the usage of the Monte Carlo Method does provide a close replication to a gambler learning to play blackjack over time. Because of the methodology that mandates a choice between hitting and staying, this simulation was able to get close to Basic Strategy. The Monte Carlo Method's mean gain percentage of -3.22% approaches but does not reach the Basic Strategy I's -2.28% mean gain percentage.

Although the simulations were able to accurately live up to this paper's goals, there are a few caveats to this paper's changes that could have improved the experimentation, and thus increased the gain percentage. As mentioned earlier, a significant improvement to the project would be to treat hard and soft totals separately. It would guarantee more accuracy in the simulation and would likely result in a higher gain percentage. The issue with this is that it would not only double the runtime of each simulation, but it would also increase the number of simulations necessary to reach an equilibrium state. With the tradeoff between accuracy and efficiency, these simulations were chosen to have a reasonable balance.

Another improvement to this simulation could have been the additional options to split and double. It would allow the direct comparison of the Monte Carlo method to Basic Strategy II, without the need of a non-doubling version of Basic Strategy. Each additional option provides its own set of problems. First, it could be difficult to determine if splitting a hand results in a positive outcome. If both hands win or lose, then it is clear whether the strategy should be played again. If one hand wins while the other hand loses, then it becomes unclear whether splitting was the optimal strategy. In the case of doubling, it is a two-fold strategy, with a combination of hitting and then staying. In this case, it would be much easier to determine whether hitting is ideal given a particular scenario, and if staying is ideal given the following scenario.

Ultimately the creation of these simulations provides a realistic understanding of how Blackjack players learn optimal strategy over time. The usage of the Monte Carlo Method serves the role of a learning algorithm that approaches Basic Strategy. With the gains from Blackjack for casinos decreasing over time, it would be interesting to study how long Blackjack remains a prominent gambling game before casinos decide to further modify the rules to increase their advantage.

Bibliography

Griffin, Peter A. *The Theory of Blackjack: The Compleat Card Counter's Guide to the Casino Game of 21.* Las Vegas, NV: Huntington, 1999.

Guillot, Gilles. *Monte Carlo Optimization Methods.* Technical University of Denmark. Accessed May 22, 2014. Available from http://www2.imm.dtu.dk/courses/02443/slides2014/optim_HO.pdf.

HitOrSplit.com. "Top Ten Most Common Blackjack Mistakes That Cost You Money." *HitOrSplit.com.* Accessed May, 21 2014. Available from http://www.hitorsplit.com/articles/Top_Ten_Most_Common_Blackjack_Mistakes.html.

Hoyte Blackjack Labs. "The Derivation of a Basic Strategy." *Hoyte Blackjack Labs: The Derivation of a Basic Strategy.* Accessed May 21, 2014. Available from http://www.hcsw.org/hbjl/deriv.html.

Katzgraber, Helmut. "Introduction to Monte Carlo Methods." Diss. Texas A&M U, 2011.

Katzman, Daniel, Jessica Moreno, Jason Noelanders, and Mark Winston-Galant. Comparisons of Two Means. University of Michigan. Accessed on May 21, 2014. Available from https://controls.engin.umich.edu/wiki/index.php/Comparisons_of_two_means.

Metropolis, Nicholas. "The Beginning of the Monte Carlo Method." *Los Alamos Science* Special Issue (1987): 125-30.

Metropolis, Nicholas, and Stanislaw Ulam. "The Monte Carlo Method." *Journal of the American Statistical Association* 44.247 (1949): 335-341.

Mosegaard, Klaus, and Malcolm Sambridge. "Monte Carlo Analysis of Inverse Problems." Inverse Problems 18.3 (2002): R29-54.

Ofton, Loudon. "The History of Blackjack." *Blackjack Apprenticeship*. Accessed May 21, 2104. Available from http://www.blackjackapprenticeship.com/resources/history-of-blackjack/.

Press, William H., Teukolsky, Saul A., Vetterling, William T., and Flannery, B.P., *Numerical Recipes: The Art of Scientific Computing*, Third Edition (New York: Cambridge University Press, 2007).

Shackleford, Michael. "Blackjack - FAQ." *The Wizard of Odds*. Accessed May 21, 2014. Available from http://wizardofodds.com/ask-the-wizard/blackjack/.

Smith, Kenneth. "Casino Blackjack: Rules of the Game." *BlackjackInfo.com*. Accessed May 21, 2014. Available from http://www.blackjackinfo.com/blackjack-rules.php.

Wintle, Simon, and Adam Wintle. "History of Blackjack." *The World of Playing Cards*. Accessed May 21, 2014. Available from http://www.wopc.co.uk/history/blackjack/blackjack.html.

Zimran, Ariell, Anna Klis, Alejandra Fuster, and Christopher Rivelli. *The Game of Blackjack and Analysis of Counting Cards*. University of Texas. Accessed May 21, 2014. Available from http://www.annaklis.com/uploads/6/4/7/2/6472295/zimran_klis_fuster_rivelli.gametheory.blackjack.pdf.

Appendix

```
# Appendix I
# simple_strategy.py
# This file serves as the file by which we run Simple Strategy simulations.

import random

global s
global k
global player2
global strategy
global results
global tot_hands
global total_deck


def new_deck():
    del total_deck[:]
    one_deck=[2,3,4,5,6,7,8,9,10,"J","Q","K","A"]
    q=0
    while q < 24:                           # Append 24 suits or 6 decks
        for i in one_deck:
            total_deck.append(i)
        q +=1
    random.shuffle(total_deck)

def init_hit(hand):
    card1 = total_deck.pop(0)

    #Adds cards to hand
    hand.append(card1)

def game():
    global s
    global k
    global player2
    global strategy
    global bust
    global tot_hands

    k = False                       # Stay counter
    s = False                       # Split counter
    bust = False                    # Bust counter
    firstbust = False
    secondbust = False

    player=[]
    player2=[]
    dealer=[]
    strategy=[]

    init_hit(player)                # Deals players 1st card
    init_hit(player)                # Deals players 2nd card
    init_hit(dealer)               # Deals computers 1st card
    init_hit(dealer)               # Deals computers 2nd card

    while k == False and bust == False:            # Run non-split option until "Stay"
    counter
        reg_sim(player)
        if k == False and bust == False:            # If mid-play and no stay/bust
            run1 = [score(player[:-1]),score([dealer[0]]),0,strategy[-1]]
            results.append(run1)

    if k == False and bust == True:                 # IF first hand busts
        run1 = [score(player[:-1]),score([dealer[0]]),-1,strategy[-1]]
        results.append(run1)
        firstbust = True

    tot_hands += 1
    k = False                                           # Reset "Stay" counter
    bust = False                                        # Reset "Bust" counter

    if s == True:
        tot_hands += 1

    while s == True and k == False and bust == False:          # Run non-split option for
    second if split happened until "Stay" counter
        reg_sim(player2)
        if k == False and bust == False:
            run1 = [score(player2[:-1]),score([dealer[0]]),0,strategy[-1]]
            results.append(run1)

    if s == True and k == False and bust == True:
        run1 = [score(player2[:-1]),score([dealer[0]]),-1,strategy[-1]]
        results.append(run1)
        secondbust = True
    run_dealer(dealer)                                # Run dealer

    if firstbust == False:
        points = score_check(player,dealer)         # Calculate Win, Loss, or Push
        run1 = [score(player[:-1]),score([dealer[0]]),points,strategy[-1]]
        results.append(run1)

    if secondbust == False and s == True:
        points2 = score_check(player2,dealer)
        run2 = [score(player2[:-1]),score([dealer[0]]),points2,strategy[-1]]
        results.append(run2)


def hit(hand):
    strategy.append('Hit')
    card = total_deck.pop(0)

    # Adds card to hand
    hand.append(card)
```

```
def score(hand):
    global bust

    total = 0
    a=0
    for cards in hand:
        if cards == "J" or cards == "Q" or cards == "K":
            total+= 10
        elif cards == "A":
            total+= 11
            a+=1
        else:
            total += cards

    while a>0 and total > 21:          # If ace(s), subtracts 10 for each ace until below 21
        total -= 10
        a -= 1

    if total > 21:
        bust = True

    return total

def split(hand):
    global s
    global player2
    strategy.append('Split')
    s = True                       # Indicates that a split has occured
    hand.pop(1)                 # Pops off second card
    player2.append(hand[0])     # Assigns second card to second hand

    hit(hand)
    hit(player2)

def stay(hand):
    global k
    strategy.append('Stay')
    k = True

def reg_sim(hand):
    plays = []
    q = False                          # Split hasn't occured
    yet
    if (hand[0]==hand[1]) and s == False:         # Run split option is possible
        plays.append(split(hand))
        q = True                       # Split option is possible
    if score(hand) > 17:
        plays.append(stay(hand))
    elif score(hand) < 12:
        plays.append(hit(hand))
    else:
        if q == True:
            plays = [split(hand)] * 3  + [hit(hand)] * (18-score(hand)) + [stay(hand)] * (
            score(hand) - 11)
        else:
            if score(hand) == 12:
                plays = [hit(hand)] * 9 + [stay(hand)] * 1
            elif score(hand) == 13:
                plays = [hit(hand)] * 7 + [stay(hand)] * 3
            elif score(hand) == 14:
                plays = [hit(hand)] * 6 + [stay(hand)] * 4
            elif score(hand) == 15:
                plays = [hit(hand)] * 4 + [stay(hand)] * 6
            elif score(hand) == 16:
                plays = [hit(hand)] * 3 + [stay(hand)] * 7
            elif score(hand) == 17:
                plays = [hit(hand)] * 1 + [stay(hand)] * 9
    random.sample(plays,1)



def run_dealer(comp):
    while score(comp) < 17:
        init_hit(comp)

def score_check(hand,comp):
    if score(hand) > 21:
        return -1
    elif score(hand) == 21 and len(hand) == 2:
        if score(hand) == score(comp) and len(comp) == 2:
            return 1
        else:
            return 1.5
    elif score(hand) > score(comp):
        return 1
    elif score(comp) > 21:
        return 1
    elif score(hand) == score(comp):
        return 0
    else:
        return -1
sim = 0
totals=[]
n = int(raw_input("How many simulations?: "))
while sim < 1000:
    tot_hands = 0
    results = []

    total_deck = []

    i = 0
    wins = 0
    wins1 = 0
    losses = 0
    losses1 = 0
    ties = 0
    while i < n:
        if len(total_deck)<53:
            new_deck()
        game()
        i += 1

    m = 0
    while m < len(results):
        if results[m][2] < 0:
            losses += (-1*results[m][2])
            losses1 += 1
        elif results[m][2] > 0:
            wins += results[m][2]
            wins1 += 1
        else:
            ties += 1
        m += 1

    net_gain = wins-losses
    net_gain1 = wins1-losses1

    gain = round(100*float((net_gain))/float((tot_hands)),1)

    totals.append(gain)
    sim += 1

print "Total Hands: ", tot_hands
print "Total Wins: ", wins
print "Total Losses: ", losses
print "Total Ties / Mid-Play (Discounted): ", ties
print "Net Gain: ", net_gain
print "Winning Hands Percentage: ", round(100*float((net_gain1))/float((wins1+losses1)),2),
"%"
print "Total Gain Percentage: ", round(100*float((net_gain))/float((tot_hands)),2),"%"

print totals
```

```
# Appendix II
# basic_strategy_i.py
# This file serves as the file by which we run Basic Strategy I simulations.

import random

global results
global total_deck
global tot_hands


def new_deck():
    del total_deck[:]
    one_suit=[2,3,4,5,6,7,8,9,10,"J","Q","K","A"]
    q=0
    while q < 24:                           # Append 24 suits or 6 decks
        for i in one_suit:
            total_deck.append(i)
        q +=1
    random.shuffle(total_deck)

def init_hit(hand):
    card = [total_deck.pop(0)]

    #Adds cards to hand
    hand.append(card[0])

def no_record_hit(hand):                   # For dealer's second card (unseen)
    card = [total_deck.pop(0)]

    #Adds cards to hand
    hand.append(card[0])


def wager():
    return 1

def game():
    global s
    global k
    global player2
    global strategy
    global bust
    global tot_hands

    k = False                       # Stay counter
    s = False                       # Split counter
    bust = False                # Bust counter
    firstbust = False           # First hand has been busted
    secondbust = False          # Second hand has been busted

    player=[]
    player2=[]
    dealer=[]
    strategy=[]

    init_hit(player)                # hits players 1st card
    init_hit(player)                # hits players 2nd card
    init_hit(dealer)               # hits computers 1st card
    no_record_hit(dealer)          # hits computers 2nd card (Doesn't affect running count)

    while k == False and bust == False:                            # Run non-split option
    until "Stay" counter
        counting_strategy(player,dealer)
        if k == False and bust == False:            # If mid-play and no stay/bust
            run1 = [score(player[:-1]),score([dealer[0]]),0,strategy[-1]]
            results.append(run1)

    if k == False and bust == True:                 # IF first hand busts
        run1 = [score(player[:-1]),score([dealer[0]]),-1*wager(),strategy[-1]]
        results.append(run1)
        firstbust = True

    k = False                                           # Reset "Stay" counter
    bust = False                                        # Reset "Bust" counter
    tot_hands += 1

    if s == True:
        tot_hands += 1

    while s == True and k == False and bust == False:          # Run non-split option for
    second if split happened until "Stay" counter
        counting_strategy(player2,dealer)
        if k == False and bust == False:
            run1 = [score(player2[:-1]),score([dealer[0]]),0,strategy[-1]]
            results.append(run1)

    if s == True and k == False and bust == True:
        run1 = [score(player2[:-1]),score([dealer[0]]),-1*wager(),strategy[-1]]
        results.append(run1)
        secondbust = True
    run_dealer(dealer)                                # Run dealer

    if firstbust == False:
        points = score_check(player,dealer)         # Calculate Win, Loss, or Push
        run1 = [score(player[:-1]),score([dealer[0]]),points*wager(),strategy[-1]]
        results.append(run1)

    if secondbust == False and s == True:
        points2 = score_check(player2,dealer)
        run2 = [score(player2[:-1]),score([dealer[0]]),points2*wager(),strategy[-1]]
        results.append(run2)
```

```
def hit(hand):
    strategy.append('Hit')
    init_hit(hand)

def score(hand):
    global bust
    global hardtotal

    total = 0
    hardtotal = True
    a=0
    for cards in hand:
        if cards == "J" or cards == "Q" or cards == "K":
            total+= 10
        elif cards == "A":
            total+= 11
            a+=1
            hardtotal = False
        else:
            total += cards

    while a>0 and total > 21:          # If ace(s), subtracts 10 for each ace until below 21
        total -= 10
        a -= 1
    if a == 0:
        hardtotal = True

    if total > 21:
        bust = False

    return total

def split(hand):
    global s
    strategy.append('Split')
    s = True                       # Indicates that a split has occured
    hand.pop(1)                 # Pops off second card
    player2.append(hand[0])     # Assigns second card to second hand

    hit(hand)
    hit(player2)

def stay(hand):
    global k
    strategy.append('Stay')
    k = True

def split_strategy(play1,hit1):
    if play1[0]==play1[1] and s == False:
        if play1[0]=="A":
            return 1
        elif play1[0]==8:
            return 1
        elif (play1[0]==2 or play1[0]==3 or play1[0]==7) and score([hit1[0]]) < 8:
            return 1
        elif play1[0]==6 and score([hit1[0]]) < 7:
            return 1
        elif play1[0]==9 and score([hit1[0]]) < 10 and score([hit1[0]]) != 7:
            return 1
        elif play1[0]==4 and score([hit1[0]]) < 7 and score([hit1[0]]) > 4:
            return 1
    else:
        return 0

def counting_strategy(p1,d1):
    if split_strategy(p1,d1) == 1:                      # Basic Strategy (Splits)
            split(p1)
    elif hardtotal == False:                            # Basic Strategy (Soft Totals)
        if score(p1) < 18:
            hit(p1)
        elif score(p1) == 18 and score([d1[0]]) > 8:
            hit(p1)
        else:
            stay(p1)
    elif hardtotal == True:                                 # Basic Strategy (Hard Totals)
        if score(p1) < 12:
            hit(p1)
        elif score(p1) < 17 and score([d1[0]]) > 6:
            hit(p1)
        elif score(p1) == 12 and score([d1[0]]) < 4:
            hit(p1)
        else:
            stay(p1)

def run_dealer(comp):
    while score(comp) < 17:
        init_hit(comp)

def score_check(hand,comp):
    if score(hand) > 21:
        return -1
    elif score(hand) == 21 and len(hand) == 2:
        if score(hand) == score(comp) and len(comp) == 2:
            return 1
        else:
            return 1.5
    elif score(hand) > score(comp):
        return 1
    elif score(comp) > 21:
        return 1
    elif score(hand) == score(comp):
        return 0
    else:
        return -1
```

```
sim = 0
totals=[]
n = int(raw_input("How many simulations?: "))
while sim < 1000:
    tot_hands = 0

    results = []
    total_deck = []

    i = 0
    wins = 0
    wins1 = 0
    losses = 0
    losses1 = 0
    ties = 0
    while i < n:
        if len(total_deck)<53:
            new_deck()
        game()
        i += 1

    m = 0
    while m < len(results):
        if results[m][2] < 0:
            losses += (-1*results[m][2])
            losses1 += 1
        elif results[m][2] > 0:
            wins += results[m][2]
            wins1 += 1
        else:
            ties += 1
        m += 1

    net_gain = wins-losses
    net_gain1 = wins1-losses1

    gain = round(100*float((net_gain))/float((tot_hands)),1)

    totals.append(gain)
    sim += 1

print "Total Hands: ", tot_hands
print "Total Wins: ", wins
print "Total Losses: ", losses
print "Total Ties / Mid-Play (Discounted): ", ties
print "Net Gain: ", net_gain
print "Winning Hands Percentage: ", round(100*float((net_gain1))/float((wins1+losses1)),2),
"%"
print "Total Gain Percentage: ", round(100*float((net_gain))/float((tot_hands)),2),"%"

print totals
```

```
# Appendix III
# basic_strategy_ii.py
# This file serves as the file by which we run Basic Strategy II simulations.

import random

global results
global total_deck
global tot_hands


def new_deck():
    del total_deck[:]
    one_suit=[2,3,4,5,6,7,8,9,10,"J","Q","K","A"]
    q=0
    while q < 24:                           # Append 24 suits or 6 decks
        for i in one_suit:
            total_deck.append(i)
        q +=1
    random.shuffle(total_deck)

def init_hit(hand):
    card = [total_deck.pop(0)]

    #Adds cards to hand
    hand.append(card[0])

def no_record_hit(hand):                   # For dealer's second card (unseen)
    card = [total_deck.pop(0)]

    #Adds cards to hand
    hand.append(card[0])


def wager():
    return 1

def game():
    global s
    global k
    global player2
    global strategy
    global bust
    global tot_hands
    global double

    double = 1                      # Doubling weight
    k = False                       # Stay counter
    s = False                       # Split counter
    bust = False                # Bust counter
    firstbust = False           # First hand has been busted
    secondbust = False          # Second hand has been busted
    player=[]
    player2=[]
    dealer=[]
    strategy=[]

    init_hit(player)                # hits players 1st card
    init_hit(player)                # hits players 2nd card
    init_hit(dealer)               # hits computers 1st card
    no_record_hit(dealer)          # hits computers 2nd card (Doesn't affect running count)

    while k == False and bust == False:                            # Run non-split option
    until "Stay" counter
        counting_strategy(player,dealer)
        if k == False and bust == False:            # If mid-play and no stay/bust
            run1 = [score(player[:-1]),score([dealer[0]]),0,strategy[-1]]
            results.append(run1)

    if k == False and bust == True:                 # IF first hand busts
        run1 = [score(player[:-1]),score([dealer[0]]),-1*double*wager(),strategy[-1]]
        results.append(run1)
        firstbust = True

    k = False                                           # Reset "Stay" counter
    bust = False                                        # Reset "Bust" counter
    tot_hands += 1

    if s == True:
        tot_hands += 1

    while s == True and k == False and bust == False:          # Run non-split option for
    second if split happened until "Stay" counter
        counting_strategy(player2,dealer)
        if k == False and bust == False:
            run1 = [score(player2[:-1]),score([dealer[0]]),0,strategy[-1]]
            results.append(run1)

    if s == True and k == False and bust == True:
        run1 = [score(player2[:-1]),score([dealer[0]]),-1*double*wager(),strategy[-1]]
        results.append(run1)
        secondbust = True
    run_dealer(dealer)                                # Run dealer

    if firstbust == False:
        points = score_check(player,dealer)         # Calculate Win, Loss, or Push
        run1 = [score(player[:-1]),score([dealer[0]]),points*double*wager(),strategy[-1]]
        results.append(run1)

    if secondbust == False and s == True:
        points2 = score_check(player2,dealer)
        run2 = [score(player2[:-1]),score([dealer[0]]),points2*double*wager(),strategy[-1]]
        results.append(run2)
```

```
def hit(hand):
    strategy.append('Hit')
    init_hit(hand)

def score(hand):
    global bust
    global hardtotal

    total = 0
    hardtotal = True
    a=0
    for cards in hand:
        if cards == "J" or cards == "Q" or cards == "K":
            total+= 10
        elif cards == "A":
            total+= 11
            a+=1
            hardtotal = False
        else:
            total += cards

    while a>0 and total > 21:          # If ace(s), subtracts 10 for each ace until below 21
        total -= 10
        a -= 1
    if a == 0:
        hardtotal = True

    if total > 21:
        bust = False

    return total

def split(hand):
    global s
    strategy.append('Split')
    s = True                        # Indicates that a split has occured
    hand.pop(1)                  # Pops off second card
    player2.append(hand[0])      # Assigns second card to second hand

    hit(hand)
    hit(player2)

def stay(hand):
    global k
    strategy.append('Stay')
    k = True

def split_strategy(play1,hit1):
    if play1[0]==play1[1] and s == False:
        if play1[0]=="A":
            return 1
        elif play1[0]==8:
            return 1
        elif (play1[0]==2 or play1[0]==3 or play1[0]==7) and score([hit1[0]]) < 8:
            return 1
        elif play1[0]==6 and score([hit1[0]]) < 7:
            return 1
        elif play1[0]==9 and score([hit1[0]]) < 10 and score([hit1[0]]) != 7:
            return 1
        elif play1[0]==4 and score([hit1[0]]) < 7 and score([hit1[0]]) > 4:
            return 1
    else:
        return 0

def counting_strategy(p1,d1):
    global double
    if split_strategy(p1,d1) == 1:                      # Basic Strategy (Splits)
            split(p1)
    elif hardtotal == False:                            # Basic Strategy (Soft Totals)
        if score(p1) == 13 and score([d1[0]]) > 4 and score([d1[0]]) < 7:
            double *= 2
            hit(p1)
            stay(p1)
        elif score(p1) == 14 and score([d1[0]]) > 4 and score([d1[0]]) < 7:
            double *= 2
            hit(p1)
            stay(p1)
        elif score(p1) == 15 and score([d1[0]]) > 3 and score([d1[0]]) < 7:
            double *= 2
            hit(p1)
            stay(p1)
        elif score(p1) == 16 and score([d1[0]]) > 3 and score([d1[0]]) < 7:
            double *= 2
            hit(p1)
            stay(p1)
        elif score(p1) == 17 and score([d1[0]]) > 2 and score([d1[0]]) < 7:
            double *= 2
            hit(p1)
            stay(p1)
        elif score(p1) == 18 and score([d1[0]]) > 2 and score([d1[0]]) < 7:
            double *= 2
            hit(p1)
            stay(p1)
        elif score(p1) < 18:
            hit(p1)
        elif score(p1) == 18 and score([d1[0]]) > 8:
            hit(p1)
        else:
            stay(p1)
    elif hardtotal == True:                                 # Basic Strategy (Hard Totals)
        if score(p1) == 9 and score([d1[0]]) > 2 and score([d1[0]]) < 7:
            double *= 2
            hit(p1)
            stay(p1)
        elif score(p1) == 10 and score([d1[0]]) < 10:
            double *= 2
            hit(p1)
            stay(p1)
        elif score(p1) == 11 and score([d1[0]]) < 11:
            double *= 2
            hit(p1)
            stay(p1)
        elif score(p1) < 12:
            hit(p1)
        elif score(p1) < 17 and score([d1[0]]) > 6:
            hit(p1)
        elif score(p1) == 12 and score([d1[0]]) < 4:
            hit(p1)
        else:
            stay(p1)

def run_dealer(comp):
    while score(comp) < 17:
        init_hit(comp)

def score_check(hand,comp):
    if score(hand) > 21:
        return -1
    elif score(hand) == 21 and len(hand) == 2:
        if score(hand) == score(comp) and len(comp) == 2:
            return 1
        else:
            return 1.5
    elif score(hand) > score(comp):
        return 1
    elif score(comp) > 21:
        return 1
    elif score(hand) == score(comp):
        return 0
    else:
        return -1

sim = 0
totals=[]
n = int(raw_input("How many simulations?: "))
while sim < 1000:
    tot_hands = 0

    results = []
    total_deck = []

    i = 0
    wins = 0
    wins1 = 0
    losses = 0
    losses1 = 0
    ties = 0
    while i < n:
        if len(total_deck)<53:
            new_deck()
        game()
        i += 1

    m = 0
    while m < len(results):
        if results[m][2] < 0:
            losses += (-1*results[m][2])
            losses1 += 1
        elif results[m][2] > 0:
            wins += results[m][2]
            wins1 += 1
        else:
            ties += 1
        m += 1

    net_gain = wins-losses
    net_gain1 = wins1-losses1
    gain = round(100*float((net_gain))/float((tot_hands)),1)

    totals.append(gain)
    sim += 1

print "Total Hands: ", tot_hands
print "Total Wins: ", wins
print "Total Losses: ", losses
print "Total Ties / Mid-Play (Discounted): ", ties
print "Net Gain: ", net_gain
print "Winning Hands Percentage: ", round(100*float((net_gain1))/float((wins1+losses1)),2),
"%"
print "Total Gain Percentage: ", round(100*float((net_gain))/float((tot_hands)),2),"%"

print totals
```

```
# Appendix IV
# counting_cards.py
# This file serves as the file by which we run Counting Cards simulations.

import random

global results
global total_deck
global true_count
global tot_hands

def new_deck():
    global running_count

    del total_deck[:]
    one_suit=[2,3,4,5,6,7,8,9,10,"J","Q","K","A"]
    q=0
    while q < 24:                           # Append 24 suits or 6 decks
        for i in one_suit:
            total_deck.append(i)
        q +=1
    random.shuffle(total_deck)

    running_count = 0                       # Resetting the running count after a shuffle

def get_run_count(card0):
    global true_count
    global running_count

    if score(card0) < 7:                        # generate running count
        running_count += 1
    elif score(card0) < 10:
        running_count += 0
    else:
        running_count -= 1

    true_count = running_count/int(len(total_deck)/52)        # Generates an integer-valued
    true count

def init_hit(hand):
    card = [total_deck.pop(0)]
    get_run_count(card)

    #Adds cards to hand
    hand.append(card[0])

def no_record_hit(hand):                   # For dealer's second card (unseen)
    card = [total_deck.pop(0)]

    #Adds cards to hand
    hand.append(card[0])
```

```
def wager():
    if prehand_count >= 8:
        val = 5
    elif prehand_count >= 6:
        val = 4
    elif prehand_count >= 4:
        val = 3
    elif prehand_count >= 2:
        val = 2
    else:
        val = 1

    return val

def game():
    global s
    global k
    global player2
    global strategy
    global bust
    global prehand_count
    global tot_hands
    global double

    double = 1
    k = False                       # Stay counter
    s = False                       # Split counter
    bust = False                # Bust counter
    firstbust = False           # First hand has been busted
    secondbust = False          # Second hand has been busted

    player=[]
    player2=[]
    dealer=[]
    strategy=[]

    init_hit(player)                # hits players 1st card
    init_hit(player)                # hits players 2nd card
    init_hit(dealer)               # hits computers 1st card
    no_record_hit(dealer)          # hits computers 2nd card (Doesn't affect running count)

    prehand_count = true_count
    while k == False and bust == False:                            # Run non-split option
    until "Stay" counter
        counting_strategy(player,dealer)
        if k == False and bust == False:            # If mid-play and no stay/bust
            run1 = [score(player[:-1]),score([dealer[0]]),0,strategy[-1]]
            results.append(run1)

    if k == False and bust == True:                 # IF first hand busts
        run1 = [score(player[:-1]),score([dealer[0]]),-1*double*wager(),strategy[-1]]
        results.append(run1)
        firstbust = True

    k = False                                           # Reset "Stay" counter
    bust = False                                        # Reset "Bust" counter
    tot_hands += 1

    if s == True:
        tot_hands += 1

    while s == True and k == False and bust == False:         # Run non-split option for
    second if split happened until "Stay" counter
        counting_strategy(player2,dealer)
        if k == False and bust == False:
            run1 = [score(player2[:-1]),score([dealer[0]]),0,strategy[-1]]
            results.append(run1)

    if s == True and k == False and bust == True:
        run1 = [score(player2[:-1]),score([dealer[0]]),-1*double*wager(),strategy[-1]]
        results.append(run1)
        secondbust = True
    run_dealer(dealer)                                # Run dealer

    if firstbust == False:
        points = score_check(player,dealer)         # Calculate Win, Loss, or Push
        run1 = [score(player[:-1]),score([dealer[0]]),points*double*wager(),strategy[-1]]
        results.append(run1)

    if secondbust == False and s == True:
        points2 = score_check(player2,dealer)
        run2 = [score(player2[:-1]),score([dealer[0]]),points2*double*wager(),strategy[-1]]
        results.append(run2)


def hit(hand):
    strategy.append('Hit')
    init_hit(hand)

def score(hand):
    global bust
    global hardtotal

    total = 0
    hardtotal = True
    a=0
    for cards in hand:
        if cards == "J" or cards == "Q" or cards == "K":
            total+= 10
        elif cards == "A":
            total+= 11
            a+=1
            hardtotal = False
        else:
            total += cards

    while a>0 and total > 21:          # If ace(s), subtracts 10 for each ace until below 21
        total -= 10
        a -= 1
    if a == 0:
        hardtotal = True

    if total > 21:
        bust = False

    return total

def split(hand):
    global s
    strategy.append('Split')
    s = True                       # Indicates that a split has occured
    hand.pop(1)                 # Pops off second card
    player2.append(hand[0])     # Assigns second card to second hand

    hit(hand)
    hit(player2)

def stay(hand):
    global k
    strategy.append('Stay')
    k = True

def split_strategy(play1,hit1):
    if play1[0]==play1[1] and s == False:
        if play1[0]=="A":
            return 1
        elif play1[0]==8:
            return 1
        elif (play1[0]==2 or play1[0]==3 or play1[0]==7) and score([hit1[0]]) < 8:
            return 1
        elif play1[0]==6 and score([hit1[0]]) < 7:
            return 1
        elif play1[0]==9 and score([hit1[0]]) < 10 and score([hit1[0]]) != 7:
            return 1
        elif play1[0]==4 and score([hit1[0]]) < 7 and score([hit1[0]]) > 4:
            return 1
    else:
        return 0

def counting_strategy(p1,d1):
    global double
    if score(p1) == 16 and score([d1[0]]) == 10:        # 16 vs. 10 (True Count 0)
        if true_count > 0:
            if split_strategy(p1,d1) == 1:
                split(p1)
            else:
                stay(p1)
        else:
            hit(p1)
    elif score(p1) == 15 and score([d1[0]]) == 10:      # 15 vs. 10 (True Count 4)
        if true_count >= 4:
            if split_strategy(p1,d1) == 1:
                split(p1)
            else:
                stay(p1)
        else:
            hit(p1)
    elif p1[0]==p1[1] and score([p1[0]]) == 10 and score([p1[1]]) == 10 and score([d1[0]])
    == 5:         # 10 and 10 vs. 5
        if true_count >= 5:
            split(p1)
        else:
            stay(p1)
    elif p1[0]==p1[1] and score([p1[0]]) == 10 and score([p1[1]]) == 10 and score([d1[0]])
    == 6:         # 10 and 10 vs. 6
        if true_count >= 4:
            split(p1)
        else:
            stay(p1)
    elif score(p1) == 10 and score([d1[0]]) == 10:      # 10 vs. 10 (True Count 4)
        if true_count >= 4:
            double *= 2
            hit(p1)
            stay(p1)
        else:
            hit(p1)
    elif score(p1) == 12 and score([d1[0]]) == 3:       # 12 vs. 3 (True Count 2)
        if true_count >= 2:
            if split_strategy(p1,d1) == 1:
                split(p1)
            else:
                stay(p1)
        else:
            hit(p1)
    elif score(p1) == 12 and score([d1[0]]) == 2:       # 12 vs. 2 (True Count 3)
        if true_count >= 3:
            if split_strategy(p1,d1) == 1:
                split(p1)
            else:
                stay(p1)
        else:
            hit(p1)
    elif score(p1) == 11 and score([d1[0]]) == 11:      # 11 vs. A (True Count 1)
        if true_count >= 1:
            double *= 2
            hit(p1)
            stay(p1)
        else:
            hit(p1)
    elif score(p1) == 9 and score([d1[0]]) == 2:        # 9 vs. 2 (True Count 1)
        if true_count >= 1:
            double *= 2
            hit(p1)
            stay(p1)
        else:
            hit(p1)
    elif score(p1) == 10 and score([d1[0]]) == 11:      # 10 vs. A (True Count 4)
        if true_count >= 4:
            double *= 2
            hit(p1)
            stay(p1)
        else:
            hit(p1)
    elif score(p1) == 9 and score([d1[0]]) == 7:        # 9 vs. 7 (True Count 3)
        if true_count >= 3:
            double *= 2
            hit(p1)
            stay(p1)
        else:
            hit(p1)
    elif score(p1) == 16 and score([d1[0]]) == 9:       # 16 vs. 9 (True Count 5)
        if true_count >= 5:
            if split_strategy(p1,d1) == 1:
                split(p1)
            else:
                stay(p1)
        else:
            hit(p1)
    elif score(p1) == 13 and score([d1[0]]) == 2:       # 13 vs. 2 (True Count -1)
        if true_count >= -1:
            if split_strategy(p1,d1) == 1:
                split(p1)
            else:
                stay(p1)
        else:
            hit(p1)
    elif score(p1) == 12 and score([d1[0]]) == 4:       # 12 vs. 4 (Running Count 0)
        if true_count >= 0:
            if split_strategy(p1,d1) == 1:
                split(p1)
            else:
                stay(p1)
        else:
            hit(p1)
    elif score(p1) == 12 and score([d1[0]]) == 5:       # 12 vs. 5 (True Count -2)
        if true_count >= -1:
            if split_strategy(p1,d1) == 1:
                split(p1)
            else:
                stay(p1)
        else:
            hit(p1)
    elif score(p1) == 12 and score([d1[0]]) == 6:      # 12 vs. 6 (True Count -1)
        if true_count >= -1:
            if split_strategy(p1,d1) == 1:
                split(p1)
            else:
                stay(p1)
        else:
            hit(p1)
    elif score(p1) == 13 and score([d1[0]]) == 3:      # 13 vs. 3 (True Count -2)
        if true_count >= -2:
            if split_strategy(p1,d1) == 1:
                split(p1)
            else:
                stay(p1)
        else:
            hit(p1)
    elif split_strategy(p1,d1) == 1:                   # Basic Strategy (Splits)
            split(p1)
    elif hardtotal == False:                           # Basic Strategy (Soft Totals)
        if score(p1) == 13 and score([d1[0]]) > 4 and score([d1[0]]) < 7:
            double *= 2
            hit(p1)
            stay(p1)
        elif score(p1) == 14 and score([d1[0]]) > 4 and score([d1[0]]) < 7:
            double *= 2
            hit(p1)
            stay(p1)
        elif score(p1) == 15 and score([d1[0]]) > 3 and score([d1[0]]) < 7:
            double *= 2
            hit(p1)
            stay(p1)
        elif score(p1) == 16 and score([d1[0]]) > 3 and score([d1[0]]) < 7:
            double *= 2
            hit(p1)
            stay(p1)
        elif score(p1) == 17 and score([d1[0]]) > 2 and score([d1[0]]) < 7:
            double *= 2
            hit(p1)
            stay(p1)
        elif score(p1) == 18 and score([d1[0]]) > 2 and score([d1[0]]) < 7:
            double *= 2
            hit(p1)
            stay(p1)
        elif score(p1) < 18:
            hit(p1)
        elif score(p1) == 18 and score([d1[0]]) > 8:
            hit(p1)
        else:
            stay(p1)
    elif hardtotal == True:                                # Basic Strategy (Hard Totals)
```

```
        if score(p1) == 9 and score([d1[0]]) > 2 and score([d1[0]]) < 7:
            double *= 2
            hit(p1)
            stay(p1)
        elif score(p1) == 10 and score([d1[0]]) < 10:
            double *= 2
            hit(p1)
            stay(p1)
        elif score(p1) == 11 and score([d1[0]]) < 11:
            double *= 2
            hit(p1)
            stay(p1)
        elif score(p1) < 12:
            hit(p1)
        elif score(p1) < 17 and score([d1[0]]) > 6:
            hit(p1)
        elif score(p1) == 12 and score([d1[0]]) < 4:
            hit(p1)
        else:
            stay(p1)

def run_dealer(comp):
    while score(comp) < 17:
        init_hit(comp)

def score_check(hand,comp):
    if score(hand) > 21:
        return -1
    elif score(hand) == 21 and len(hand) == 2:
        if score(hand) == score(comp) and len(comp) == 2:
            return 1
        else:
            return 1.5
    elif score(hand) > score(comp):
        return 1
    elif score(comp) > 21:
        return 1
    elif score(hand) == score(comp):
        return 0
    else:
        return -1

sim = 0
totals=[]
n = int(raw_input("How many simulations?: "))
while sim < 1000:
    tot_hands = 0

    results = []
    total_deck = []
    true_count = 0
    running_count = 0
    i = 0
    wins = 0
    wins1 = 0
    losses = 0
    losses1 = 0
    ties = 0
    while i < n:
        if len(total_deck)<75:
            new_deck()
        game()
        i += 1

    m = 0
    while m < len(results):
        if results[m][2] < 0:
            losses += (-1*results[m][2])
            losses1 += 1
        elif results[m][2] > 0:
            wins += results[m][2]
            wins1 += 1
        else:
            ties += 1
        m += 1

    net_gain = wins-losses
    net_gain1 = wins1-losses1
    gain = round(100*float((net_gain))/float((tot_hands)),1)

    totals.append(gain)
    sim += 1

print "Total Hands: ", tot_hands
print "Total Wins: ", wins
print "Total Losses: ", losses
print "Total Ties / Mid-Play (Discounted): ", ties
print "Net Gain: ", net_gain
print "Winning Hands Percentage: ", round(100*float((net_gain1))/float((wins1+losses1)),2),
"%"
print "Total Gain Percentage: ", round(100*float((net_gain))/float((tot_hands)),2),"%"

print totals
```

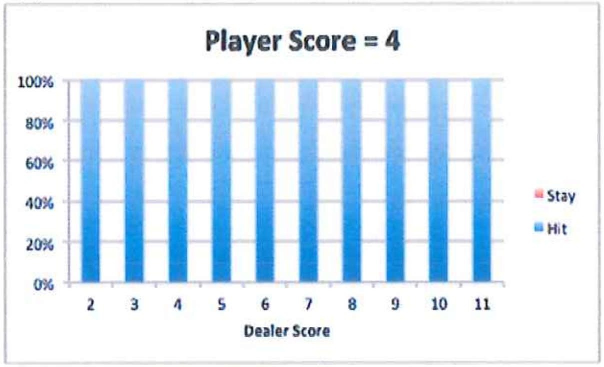
```
# Appendix V
# monte_carlo.py
# This file serves as the file by which we run Monte Carlo Method simulations.

import random

global s
global k
global h
global player2
global strategy
global results
global total_deck
global hits_used
global total_used
global tot_hands


def new_deck():
    del total_deck[:]
    one_suit=[2,3,4,5,6,7,8,9,10,"J","Q","K","A"]
    q = 0
    while q < 24:                          # Append 24 suits or 6 decks
        for i in one_suit:
            total_deck.append(i)
        q +=1
    random.shuffle(total_deck)

def init_hit(hand):
    card = [total_deck.pop(0)]

    #Adds cards to hand
    hand.append(card[0])

def game():
    global s
    global k
    global h
    global player2
    global strategy
    global bust
    global total_used
    global hits_used
    global tot_hands

    k = False                      # Stay counter
    s = False                      # Split counter
    h = False                      # Hit counter
    bust = False                   # Bust counter
    firstbust = False
    secondbust = False
```

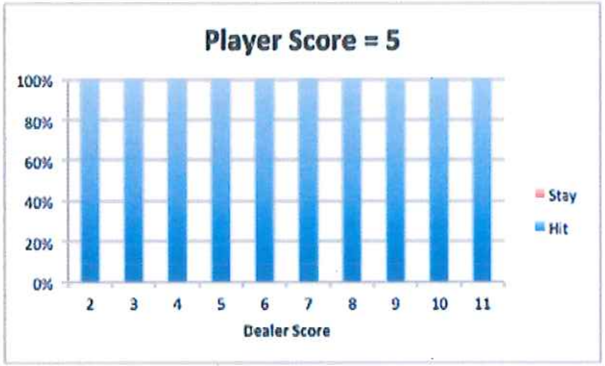
```
points = 0
player = []
player2 = []
dealer = []
strategy = []
init_hit(player)                # hits players 1st card
init_hit(player)                # hits players 2nd card
init_hit(dealer)               # hits computers 1st card
init_hit(dealer)               # hits computers 2nd card

while k == False and bust == False:             # Run non-split option until "Stay"
counter
    pick_play(player,dealer)
    bust_check(player)
    if k == False and bust == False and h == True:             # If mid-play and no
    stay/bust
        run1 = [score(player[:-1]),score([dealer[0]]),0,strategy[-1]]
        results.append(run1)
        change_matrices(player,dealer,1,1)
        h = False

if k == False and bust == True and h == True:                  # IF first hand busts
    run1 = [score(player[:-1]),score([dealer[0]]),-1,strategy[-1]]
    results.append(run1)
    change_matrices(player,dealer,1,0)
    firstbust = True

h = False                                            # Reset "hit" counter
k = False                                            # Reset "Stay" counter
bust = False                                         # Reset "Bust" counter
tot_hands += 1

if s == 1:
    tot_hands += 1

while s == True and k == False and bust == False:           # Run non-split option for
second if split happened until "Stay" counter
    pick_play(player2,dealer)
    bust_check(player2)
    if s == True and k == False and bust == False and h == True:
        run1 = [score(player2[:-1]),score([dealer[0]]),0,strategy[-1]]
        results.append(run1)
        change_matrices(player2,dealer,1,1)
        h = False

if s == True and k == False and bust == True and h == True:
    run1 = [score(player2[:-1]),score([dealer[0]]),-1,strategy[-1]]
    results.append(run1)
    change_matrices(player2,dealer,1,0)
    secondbust = True

run_dealer(dealer)                                 # Run dealer
```

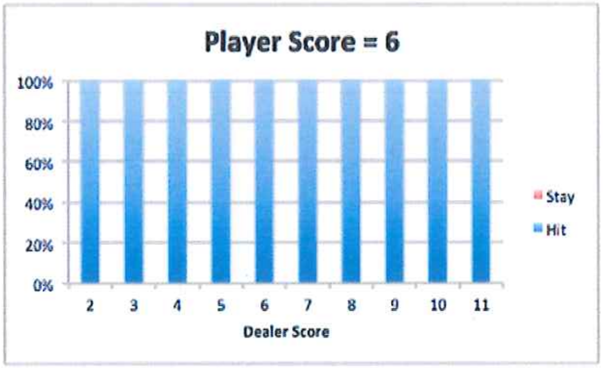
```
    if firstbust == False:
        points = score_check(player,dealer)         # Calculate Win, Loss, or Push
        run1 = [score(player[:-1]),score([dealer[0]]),points,strategy[-1]]
        results.append(run1)
        if points == 1:
            change_stay_matrices(player,dealer,1,0)
        elif points == -1:
            change_stay_matrices(player,dealer,1,1)

    if secondbust == False and s == True:
        points2 = score_check(player2,dealer)
        run2 = [score(player2[:-1]),score([dealer[0]]),points2,strategy[-1]]
        results.append(run2)
        if points2 == 1:
            change_stay_matrices(player2,dealer,1,0)
        elif points2 == -1:
            change_stay_matrices(player2,dealer,1,1)

def hit(hand):
    global h
    strategy.append('Hit')
    h = True
    init_hit(hand)

def change_stay_matrices(play1, deal1, p1, p2):
    global total_used
    global hits_used

    row = score(play1)-4
    col = score([deal1[0]])-2

    total_used[row][col] += p1
    hits_used[row][col] += p2

def change_matrices(play1, deal1, p1, p2):
    global total_used
    global hits_used

    row = score(play1[:-1])-4
    col = score([deal1[0]])-2

    total_used[row][col] += p1
    hits_used[row][col] += p2

def score(hand):
    global bust

    total = 0
    a=0
    for cards in hand:
        if cards == "J" or cards == "Q" or cards == "K":
            total+= 10
        elif cards == "A":
            total+= 11
            a+=1
        else:
            total += cards

    while a>0 and total > 21:          # If ace(s), subtracts 10 for each ace until below 21
        total -= 10
        a -= 1

    if total > 21:
        bust = True

    return total

def split(hand):
    global s
    strategy.append('Split')
    s = True                       # Indicates that a split has occurred
    hand.pop(1)
    player2.append(hand[0])        # Assigns second card to second hand

    hit(hand)
    hit(player2)

def stay(hand):
    global k
    strategy.append('Stay')
    k = True

def bust_check(hand):
    global bust
    global k

    if score(hand) > 21:
        bust = True
        k = False

def split_strategy(play1,hit1):
    if play1[0]==play1[1] and s == False:
        if play1[0]=="A":
            return 1
        elif play1[0]==8:
            return 1
        elif (play1[0]==2 or play1[0]==3 or play1[0]==7) and score([hit1[0]]) < 8:
            return 1
        elif play1[0]==6 and score([hit1[0]]) < 7:
            return 1
        elif play1[0]==9 and score([hit1[0]]) < 10 and score([hit1[0]]) != 7:
            return 1
        elif play1[0]==4 and score([hit1[0]]) < 7 and score([hit1[0]]) > 4:
            return 1
    else:
        return 0

def pick_play(p1,d1):
    global total_used
    global hits_used
    global bust

    if split_strategy(p1,d1) == 1:
        split(p1)

    tot = total_used[score(p1)-4][score([d1[0]])-2] # gets total value of cell in matrix

    if score(p1) > 17:
        stay(p1)
    elif score(p1) < 12:
        hit(p1)
    else:
        if tot < 1:
            random.choice([stay(p1),hit(p1)])
        else:
            dnd = hits_used[score(p1)-4][score([d1[0]])-2]
            r = float(dnd)/float(tot)
            x = random.random()
            if x < r:
                hit(p1)
            else:
                stay(p1)

def run_dealer(comp):
    while score(comp) < 17:
        init_hit(comp)

def score_check(hand,comp):
    if score(hand) > 21:
        return -1
    elif score(hand) == 21:
        if score(hand) == score(comp):
            return 1
        else:
            return 1.5
    elif score(comp) > 21:
        return 1
    elif score(hand) > score(comp):
        return 1
    elif score(hand) == score(comp):
        return 0
    else:
        return -1

sim = 0
```

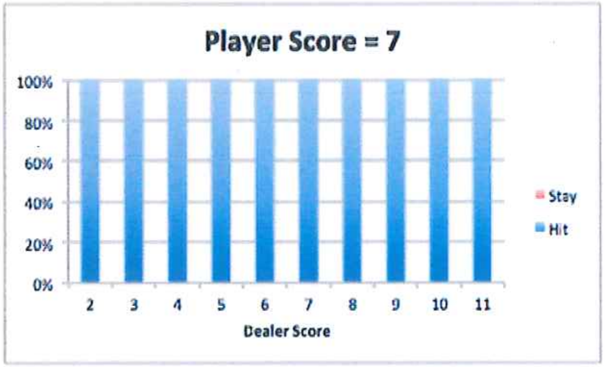
```
totals=[]
n = int(raw_input("How many simulations?: "))
while sim < 1000:
    tot_hands = 0

    results = []
    total_deck = []
    hits_used = [[]]
    total_used = [[]]
    ratio_used=[[]]

    hits_used = [[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,
    0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,
    0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,
    0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,
    0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0]]
    total_used = [[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0
    ,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0
    ,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0
    ,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0
    ,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0]]
    ratio_used = [[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0
    ,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0
    ,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0
    ,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0
    ,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0]]

    i = 0
    wins = 0
    wins1 = 0
    losses = 0
    losses1 = 0
    ties = 0
    while i < n:
        if len(total_deck)<53:
            new_deck()
        game()
        i += 1

    m = 0
    while m < len(results):
        if results[m][2] < 0:
            losses += (-1*results[m][2])
            losses1 += 1
        elif results[m][2] > 0:
            wins += results[m][2]
            wins1 += 1
        else:
            ties += 1
        m += 1

    net_gain = wins-losses
    net_gain1 = wins1-losses1

    z = 0
    while z < 18:
        x = 0
        while x < 10:
            if float(total_used[z][x]) > 0:
                ratio_used[z][x] = round(float(hits_used[z][x])/float(total_used[z][x]),2)
            else:
                ratio_used[z][x] = 0
            x += 1
        z += 1

    gain = round(100*float((net_gain))/float((tot_hands)),1)

    totals.append(gain)
    sim += 1


print "Total Hands: ", tot_hands
print "Total Wins: ", wins
print "Total Losses: ", losses
print "Total Ties / Mid-Play (Discounted): ", ties
print "Net Gain: ", net_gain
print "Winning Hands Percentage: ", round(100*float((net_gain1))/float((wins1+losses1)),2),
"%"
print "Total Gain Percentage: ", round(100*float((net_gain))/float((tot_hands)),2),"%"

print totals
print ratio_used
```

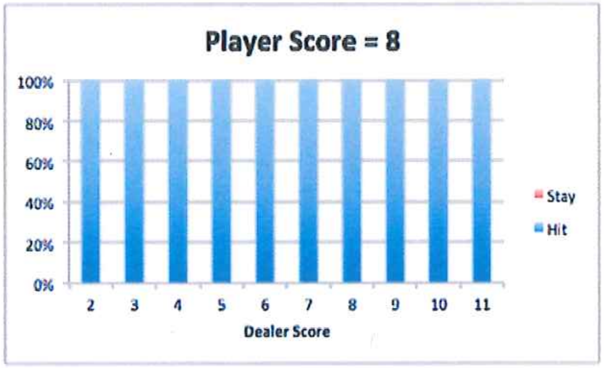
Appendix VI:

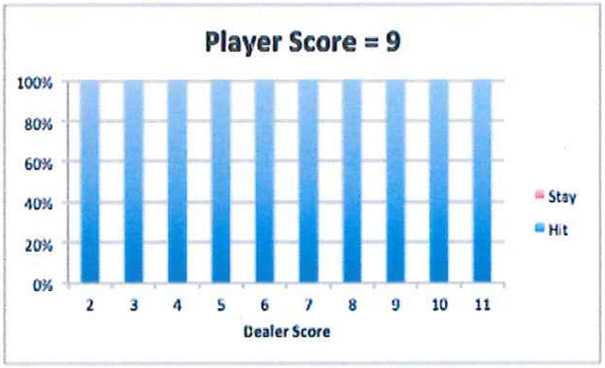

Player Score = 4
100%
80%
60%
40%
20%
0%
2
3
4
5
6
7
8
9
10
11
Dealer Score
Stay
Hit

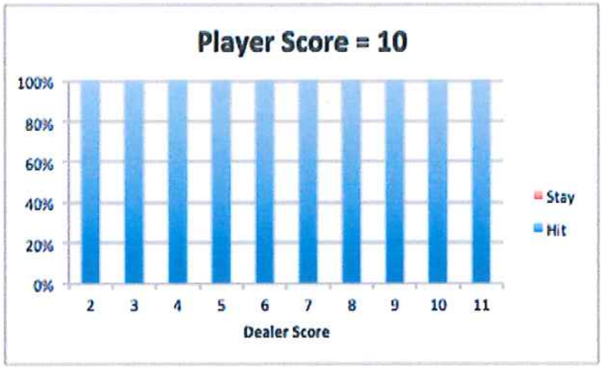

Player Score = 5
100%
80%
60%
40%
20%
0%
2
3
4
5
6
7
8
9
10
11
Dealer Score
Stay
Hit

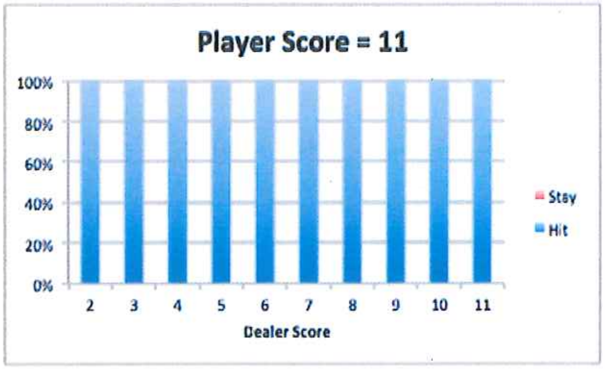

Player Score = 6
100%
80%
60%
40%
20%
0%
2
3
4
5
6
7
8
9
10
11
Dealer Score
Stay
Hit

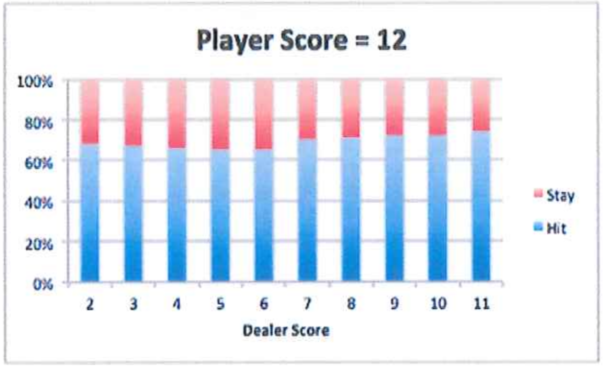

Player Score = 7
100%
80%
60%
40%
20%
0%
2
3
4
5
6
7
8
9
10
11
Dealer Score
Stay
Hit

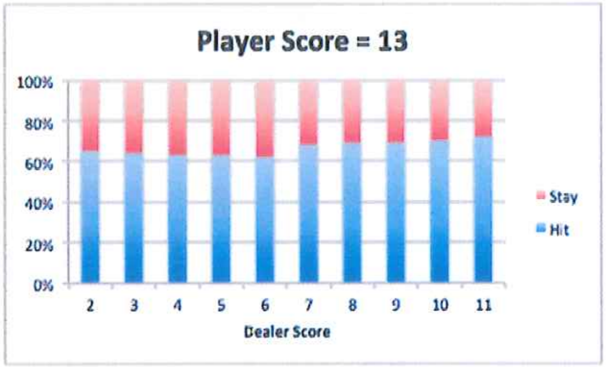

Player Score = 8
100%
80%
60%
40%
20%
0%
2
3
4
5
6
7
8
9
10
11
Dealer Score
Stay
Hit

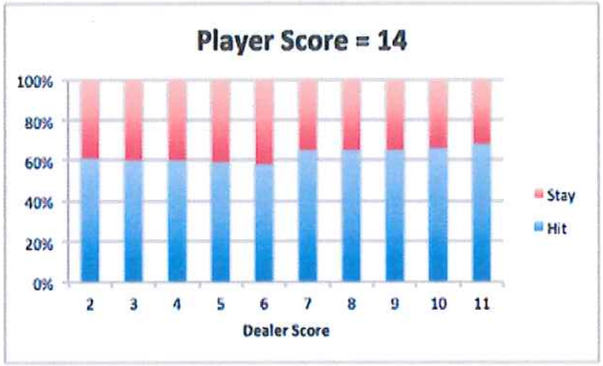

Player Score = 9
100%
80%
60%
40%
20%
0%
2
3
4
5
6
7
8
9
10
11
Dealer Score
Stay
Hit

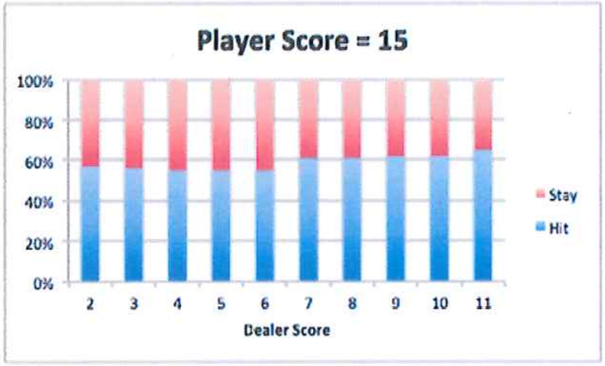

Player Score = 10
100%
80%
60%
40%
20%
0%
2
3
4
5
6
7
8
9
10
11
Dealer Score
Stay
Hit

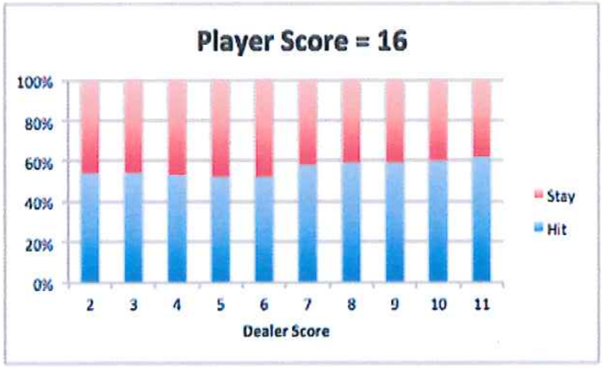

Player Score = 11
100%
80%
60%
40%
20%
0%
2
3
4
5
6
7
8
9
10
11
Dealer Score
Stay
Hit

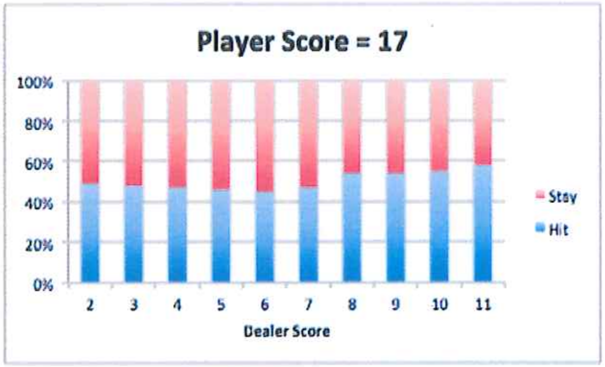

Player Score = 12
100%
80%
60%
40%
20%
0%
2 3 4 5 6 7 8 9 10 11
Dealer Score
Stay
Hit

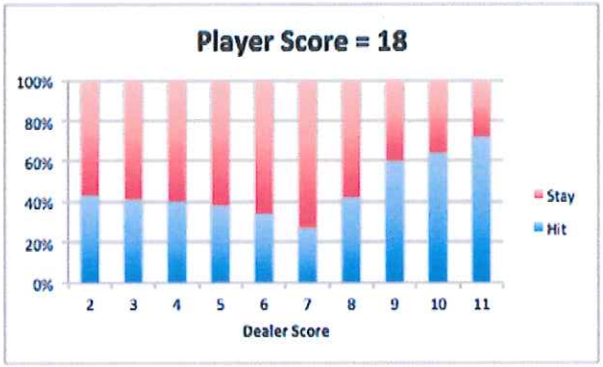

Player Score = 13
100%
80%
60%
40%
20%
0%
2 3 4 5 6 7 8 9 10 11
Dealer Score
Stay
Hit

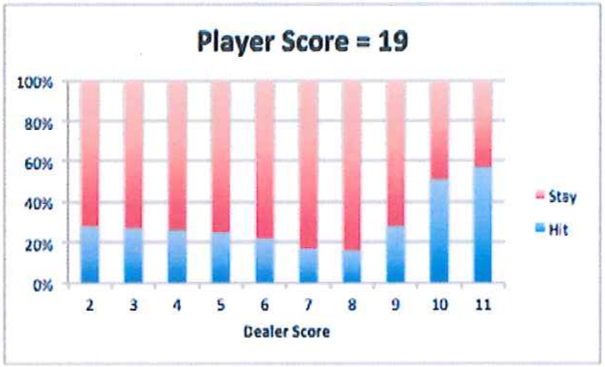

Player Score = 14
100%
80%
60%
40%
20%
0%
2 3 4 5 6 7 8 9 10 11
Dealer Score
Stay
Hit

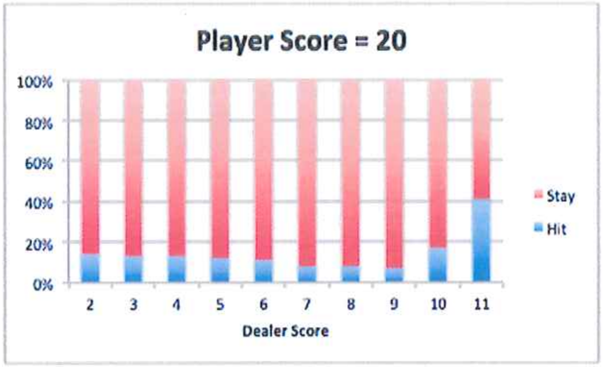

Player Score = 15
100%
80%
60%
40%
20%
0%
2 3 4 5 6 7 8 9 10 11
Dealer Score
Stay
Hit

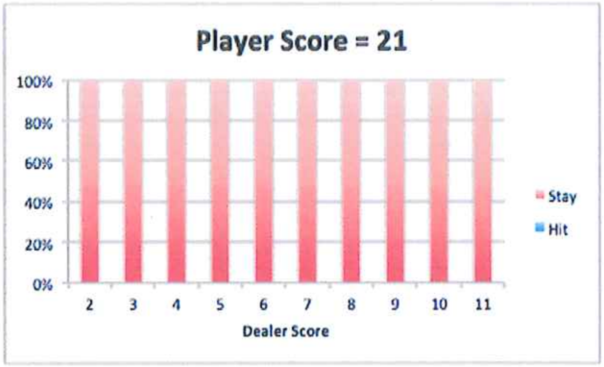

Player Score = 16
100%
80%
60%
40%
20%
0%
2 3 4 5 6 7 8 9 10 11
Dealer Score
Stay
Hit


Player Score = 17
100%
80%
60%
40%
20%
0%
2 3 4 5 6 7 8 9 10 11
Dealer Score
Stay
Hit


Player Score = 18
100%
80%
60%
40%
20%
0%
2 3 4 5 6 7 8 9 10 11
Dealer Score
Stay
Hit


Player Score = 19
100%
80%
60%
40%
20%
0%
2 3 4 5 6 7 8 9 10 11
Dealer Score
Stay
Hit


Player Score = 20
100%
80%
60%
40%
20%
0%
2
3
4
5
6
7
8
9
10
11
Dealer Score
Stay
Hit


Player Score = 21
100%
80%
60%
40%
20%
0%
2
3
4
5
6
7
8
9
10
11
Dealer Score
Stay
Hit