Racko!

Carrie Franks, Amanda Geddes,
Chris Carter, and Ruby Garza

Our group project is the modeling of the card game Racko. The members in the group are: Carrie Franks, Amanda Geddes, Chris Carter, and Ruby Garza. We each wrote a code of the program that resulted in the model of the card game. Also, there will be an explanation of how the program was coded, how it was simplified, how it could be improved, and the result of the program. There is another section on how probability relates to Racko and the rules of the game.

Over 50 years ago, Milton Bradley produced a card game. This card game is called Racko and is played by two to four people. The object of the game is to arrange ten cards in increasing sequential order by inter changing one of your cards with a facedown or discard pile card. Each person is given ten cards from one to sixty and

those ten cards have to stay in the same order that they were given. With each turn a card is chosen from either the facedown deck or the discard pile to swap with one card in their hand, except for the first round, the first person has to choose from the facedown deck in order to start the discard pile. Once the numbers in a hand are in sequential order the game is over. An example of a sequential hand is [2, 13, 19, 22, 26, 35, 41, 44, 58, 59], and an example of a non-sequential hand is [3, 13, 9, 34, 56, 23, 27, 48, 2, 10]. The person, who is able to get their ten cards in sequential order first, wins. There are many ways to play Racko and the program Racko is only one way to play Racko.

Simplifications:

In order to make this game easy to code we had to make a few simplifications. In the traditional game of Racko, in addition to winning you include point values and play for several hands until a player reaches 500 points. Excluding this feature would not only allow the game to be finished in a shorter amount of time it would also decrease the amount of overhead needed to manage the game. Additionally, we had to settle for a less graphical representation of the hands. We used Python to code and run the project in order to use user input while the code was running, but the graphics display was coded in the notebook with an understanding of sage Graphics. When we tried to figure out how to convert it to python only code, we ran into problems and things we didn't understand and decided to just give a demo in our presentation of what the display would look like, but using sage.

Model:

In order to code Racko we had to juggle around objects in order to keep track of all the game playing elements. To start with we realized that the game would require a large degree of user input and that this input would need to handle all sorts to cases. This includes garbage input such as typos or incorrectly formatted requests.

For example:

```
def findplayercount():
    user = raw_input('How many people wish to play? ')
    if user == "4" or user == "3" or user == "2":
        num_players = int(user)
    else:
        print "Invalid input; Must be a number {4, 3, 2}"
        print ""
        num_players = findplayercount()
    return num_players
```

The findplayercount() function above illustrates how typical error handling would occur. In this case only the valid counts for number of players would be accepted. In the case of invalid input, such as a user entering in “15” or “four” would instead print the allowed inputs and request the information again by calling itself.

In order to store the player’s hands, the draw pile, and the discard pile, we used lists which would make for easy drawing and discarding (pop and append) as well as fast index searching which was used to determine where cards were stored in the hand.

The creation of hands was actually quite simple. Shuffling the deck before a game required just a few lines of code:

```
def shuffledeck():
    a = range(1, 61)
    random.shuffle(a)
    return a
```

Python’s built in randomizer, which is actually pseudo-random, worked perfectly for our needs.

Finally, to bring the whole thing together we created a generic main() function which would call all the other functions and sort of act like the game-mediator. As you can see below it was also responsible for tracking the winner and ensuring to stop the game when it was appropriate too do so.

```
def main():
    print "Hello. So I see you would like to play some Racko!"
    user = raw_input('Do you need to review the rules? (yes/no)
')
    if user == 'yes' or user == 'Yes' or user == 'Y' or user ==
'y':
        print_rules()
        print ""
        print ""
    print ""
    print "Alright, let's begin!"
    facedown_deck = shuffledeck()
    faceup_deck = []
    num_players = findplayercount()
    player_names = find_player_names(num_players)
    temp = create_hands(num_players, facedown_deck)
    created_hands = temp[0]
    facedown_deck = temp[1]
    has_won = False
    count = 0
    while not has_won:
        print ""
        print ""
        print "----------------------------------------------"
        print ""
        print ""
        hand = created_hands[count%num_players]
        name = player_names[count%num_players]
```

```
            temp = take_turn(name,hand,faceup_deck,facedown_deck)
            facedown_deck = temp[0]
            faceup_deck = temp[1]
            hand = temp[2]
            has_won = check_if_won(hand)
            if has_won:
                print "We have a winner!"
                print str(player_names[count%num_players]) + " wins!"
                break
            count += 1
```

Since python objects are generally local, things like the two decks, and the hands would be passed into most functions and returned back in the form of a list in the event that they were changed.

Results:

Since the game very much relies on visual clues to analyze your cards we needed to create a convenient way to look at your hand. The default list view seemed to work out pretty well while being fast and convenient. Finally our text version may not be pretty but it is certainly functional. The following is what a turn would look like:

```
Player1:
[12, 36, 56, 7, 17, 24, 44, 34, 48, 8]
Top card on discard pile is: 2

(Input choices: facedown, faceup)
Would you like to draw from facedown pile or take the faceup
card? facedown
You drew: 58
Which card do you wish to discard? (Enter number only) 8
```

The red text is user input. In the event of garbage input the game will attempt to re-ask the question. Finally when the game will check each hand and declare a winner when a player has successfully arranged all their cards into increasing order.

Underlying Mathematics:

Racko doesn't exactly require a wealth of mathematical prowess to understand and play. Actually, if your about six years old and can count, you can probably play Racko… and win. That isn't to say, however, that there isn't anything to the game mathematically. Like most card games, we can take a look at combinatorics and probability to understand a little more of the intrinsic qualities of the game. Let's start with something easy… how about the probability of getting a hand that has already "won", that is to say, is already in strictly increasing order.

P(winning on the deal) = (number of ways to get a strictly inc. hand)
____
(total number of ways to get a hand).

Def: Binomial Coefficient (a counting definition): The number of ways to choose a subset of k elements from a set containing n elements, stated "n choose k" with formula: n!/(k!(n-k)!)

So on being dealt 10 cards from 60, there are "60 choose 10" ways to get a hand = 60!/(10!50!) , and because there are no double cards (only one of each number 1-60 in the deck), given 10 random cards, there is only one sequence that is strictly increasing. Therefore, the number of ways to get a strictly inc. hand is 1* 60! /(10!50!)

The total number of ways to get a hand is 60! /(10!50!) *10! (Where the 10! Comes in because there are 10! Permutations you could have been dealt with any 10 random cards) and therefore…

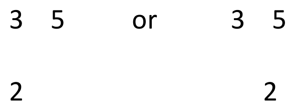
P(winning on the first deal) = 1*60!/(10!50!)/(60!/(10!50!)*10!) = 1/10! = 1/3628800 which is tiny! (So unless you're carrying a four-leaf clover, holding a rabbit's foot and have a horseshoe dangling for your belt, you're probably out of luck on this one)

Now that we know that we are probably not winning the first deal, lets look at what we are given, and analyze it a bit. We have ten cards whose orders are set, we cannot permute them, but we can exchange any individual card with a card outside of our hand (an unknown random card from the deck). Now if you're really unlucky, you can have a hand with a strictly dec. sequence (also with a probability of 1/3628800) or most likely, a combination of both inc. and dec. sequences. Something that we might want to know is the length of the longest increasing subsequence in our hand and also, an example of one with that length. There can be two or more inc. subsequences with the same length, so lets call it *a* longest subsequence. Finding the longest increasing subsequence is a topic studied and analyzed in probability and combinatorics and I will discuss the work of David Aldous and Peri Diaconis in an article for The American Mathematical Society entitled "Longest Increasing Subsequence: From Patience Sorting to The Baik-Deift-Johansson theorem. The article goes into the relationship between "Young Tableaus," a simple card game they entitled "Patience Sorting" and some more advanced mathematics and while it turns out to be quite interesting, we can get the gist of the conclusion according to our problem with just the "Patience Sorting" game.

The game is as follows: Given a random permutation of n cards (say in a shuffled deck), you must sort the deck according to a rule:

- A new card can be placed below a stack of cards if it is lesser in value than the last card in the stack or it can be placed to the right to form a new pile.

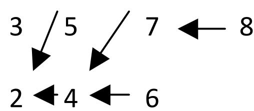
Example1: Given the permutation: [3, 5, 2], we place the three down, the five is not less than three, so we make it a new pile, and the two is less than three and five so we put it below either:



The *magic* happens if we implement this sorting into a "Greedy" algorithm. In the algorithm, the next card always goes in the leftmost place possible (so in the example above, only the left configuration works). Doing this creates a bijection between the number of piles created in the sorting and the length of the longest subsequence!! (The proof is pretty simple, assume there are L piles and an increasing subsequence of length L+1, a contradiction follows closely for you can never put a card of increased value below a card already placed, so L must be the greatest length of an inc subsequence… then follow the method outlined right after this to prove there exists a sequence of at least length L, and you're done!)

Now, to find a longest sequence: If we put a pointer on each number that goes into a pile NOT in the most left pile, that points to the bottommost card in the pile adjacent to the left, we can start from the upper right card and follow a path through the list that is strictly increasing (when read from left to right).

Example2: Given [3, 5, 2, 4, 7, 8, 6]



There are four piles, so the longest subsequence (increasing) has length four. If we start from the top right (8) and follow the path, we get [8, 7, 4, 2] as our "backwards-increasing" subsequence. It's easy to see that this isn't the only such inc. subsequence, for instance we can

use [8, 6, 4, 3]… it just guarantee's us at least one of maximum length. Also, although in this example the top cards in every row form an increasing subsequence, that is usually not the case.

Improvements:

If we were to develop our game even further, we would like to include an implementation of this algorithm so that a player could call a "cheat" function that returned a list of their greatest increasing subsequence in the index' of their original hands, and zero's in all the other places. It's simple to create the sorting of the hand but there was some difficulty creating the pointers in order to retrieve the actual subsequence. There could be a function that just returns the length of the longest list, to let a player know if they were on the right track. I imagined the cheat function would be like a "lifeline" in "Who Wants to be a Millionaire" and we could have another lifeline or lifelines that each player could play exactly once.

Another idea for a "lifeline" was a function that returned the probability of finding a desirable card in the facedown_deck, in order to help the player make a decision. The "dumbed-down" version of this function would be fairly easy if you just accept the longest subsequence that the function above returned as the subsequence the player wanted to continue with. In that case, we can just take the facedown_deck and calculate the number of cards that would numerically fit into the "holes" in the players' deck that they needed to exchange in order to complete the sequence. In reality thought, the player may find that a given card in the facedown_deck is desirable, but with a differently chosen subsequence. In this case, this lifeline would be more helpful if you only had one or two cards left out of order.

There are more improvements that could be made to our program, like most programs out there in the world, especially games (which explains why they never stopped with Atari!). The first we could make would be to add in a scoring system that keeps record of the game.

This way you could play several hands and find who wins in the end or you could simply play to a particular point value. One way we might do this is to create a function that calculates scores at the end of the game. Then we would have to pass the calculated scores as a parameter to the main function or somehow recall them within main() somehow. The general way to score points is by calling "RACK-O!". This is done when you have your cards in order from lowest to highest. By doing this you win 75 points. We would probably just add the calling of "RACK-O!" in our check_if_won function and then adding the 75 points. It would be more difficult to create a way for a player to enter it, when the program is designed to be played on one computer. On top of this players get extra points for cards in consecutive order and every other player gets 5 points for each card they have in order. You can also use custom scoring rules, which we could try and institute, but this would probably be very tricky.

Another improvement is to institute computer players. This could be very lengthy or very difficult. We could possibly do this in a simple, but cumbersome manner. That would be to try and code for every situation the computer could come upon. The problem is this would significantly lower the speed of the program as it would have to run through all the other situations. Instead, we would need to see if we could find a set of rules or patterns for the computer to use. The problem is this may turn into a very predictable and boring artificial intelligence. If we could find a way to effectively program a fairly intelligent AI, then we could also program difficulty levels. These levels could be easy, medium or hard or whatever is desired. To distinguish the difficulty levels we could program the lower ones to make a certain number of mistakes on average and hard to make little or no mistakes.

One more improvement would be to institute some networking code, so players could play from separate computers. This would enable people to play each other from their respective homes. To do this in a general sense we would probably have to make another client

program that is similar to our original code. The difference would be that one player would have to act as the host using our main program and the others would use the client program to "plug" into the host. We also would have to do some extra coding to include networking over the internet.

Finally, we could add a GUI interface. In the interface we could create the hand for the person to see as well as the discard pile. We could allow people to select an avatar or import their own avatar. We could also institute one before the actual game screen to select how many players will be involved in the game with interactive buttons on the screen as well as a button to see the rules of the game. We could also combine this with the computer player idea and allow for selection of how many computer players you want to play against. While doing this we could also add in the ability to fill empty seats with computer players if you are playing with less than four human players.

Overall, the group worked well together, each contributing their ideas and work for the finished product. With some more advanced coding, we could continue work on the game to make it look "cooler" while adding some more interactive "fun" attributes like "lifelines." We are happy with the way the game works and turned out, and it was interesting (and informative) being able to see a project go from just our ideas and thoughts to something we could interact with and play with on the computer screen.

References (not exactly in research-quality format):

The instructions to the game were found in this website: http://www.hasbro.com/shop/details.cfm?guid=8F0C7AA5-6D40-1014-8BF0-9EFBF894F9D4&product_id=9632&src=endeca and from http://en.wikipedia.org/wiki/Racko

An example of patience-sort and some explanation of the Aldous-Diaconis article:
http://wordaligned.org/articles/patience-sort
In 1999 The Bulletin of the American Mathematical Society published a paper by David Aldous and Persi Diaconis entitled: “Longest Increasing Subsequences: From Patience Sorting to the Baik-Deift-Johansson Theorem”.

http://www.ams.org/journals/bull/1999-36-04/S0273-0979-99-00796-X/S0273-0979-99-00796-X.pdf

```
#RACKO
import random

def main():
    print "Hello. So I see you would like to play some Racko!"
    user = raw_input('Do you need to review the rules? (yes/no) ')
    if user == 'yes' or user == 'Yes' or user == 'Y' or user == 'y':
        print_rules()
        print ""
        print ""
    print ""
    print "Alright, let's begin!"
    facedown_deck = shuffledeck()
    faceup_deck = []
    num_players = findplayercount()
    player_names = find_player_names(num_players)
    temp = create_hands(num_players, facedown_deck)
    created_hands = temp[0]
    facedown_deck = temp[1]
    has_won = False
    count = 0
    while not has_won:
        print ""
        print ""
        print "----------------------------------------------"
        print ""
        print ""
        hand = created_hands[count%num_players]
        name = player_names[count%num_players]
        temp = take_turn(name,hand,faceup_deck,facedown_deck)
        facedown_deck = temp[0]
        faceup_deck = temp[1]
        hand = temp[2]
```

```
        has_won = check_if_won(hand)
        if has_won:
            print ""
            print "We have a winner!"
            print str(player_names[count%num_players]) + " wins!"
            break
        count += 1

def find_player_names(num_players):
    i = 1
    player_names = []
    while i <= num_players:
        print "Please enter name for Player " + str(i),
        user = raw_input(':: ')
        player_names.append(user)
        i += 1
    return player_names


def findplayercount():
    user = raw_input('How many people wish to play? ')
    if user == "4" or user == "3" or user == "2":
        num_players = int(user)
    else:
        print "Invalid input; Must be a number {4, 3, 2}"
        print ""
        num_players = findplayercount()
    return num_players

def draw_facedown(facedown_deck):
    ls = facedown_deck[1:]
    return [facedown_deck[0], ls]

def draw_faceup(faceup_deck):
    ls = faceup_deck[1:]
    return [faceup_deck[0], ls]

def take_turn(name,hand,faceup_deck,facedown_deck):
    print str(name) + ": "
    print str(hand) ###
    if faceup_deck == []:
        print "You are the first to go. Drawing from pile..."
        a = draw_facedown(facedown_deck)
        card = a[0]
        facedown_deck = a[1]
        print "You drew: " + str(card)
        x = True
        while x:
            x = False
            user = raw_input('Which card do you wish to discard? (Enter
number only) ')
            if hand.count(int(user)) == 1:
                index = hand.index(int(user))
                hand.insert(index,card)
                card = hand.pop(index+1)
                faceup_deck.insert(0,card)
            elif int(user) == card:
```

```
                faceup_deck.insert(0,card)
            else:
                x = True
                print "Card not found. Please pick again" # repeat
command
                print ""
    else:
        if facedown_deck == []:
            temp = faceup_deck.pop(0)
            facedown_deck = faceup_deck.reverse()
            faceup_deck = temp
        print "Top card on discard pile is: " + str(faceup_deck[0])
        print ""
        while True:
            print "(Input choices: facedown, faceup)"
            user = raw_input('Would you like to draw from facedown pile
or take the faceup card? ')
            if user == "facedown" or user == "faceup":
                break
            print "Invalid input. Please enter choice again."
            print ""
        if user == "facedown":
            a = draw_facedown(facedown_deck)
            card = a[0]
            facedown_deck = a[1]
            print "You drew: " + str(card)
            x = True
            while x:
                x = False
                user = raw_input('Which card do you wish to discard?
(Enter number only) ')
                if hand.count(int(user)) == 1:
                    index = hand.index(int(user))
                    hand.insert(index,card)
                    card = hand.pop(index+1)
                    faceup_deck.insert(0,card)
                elif int(user) == card:
                    faceup_deck.insert(0,card)
                else:
                    x = True
                    print "Card not found. Please pick again" # repeat
command
                    print ""
        if user == "faceup":
            a = draw_faceup(faceup_deck)
            card = a[0]
            faceup_deck = a[1]
            print "You took: " + str(card) + " from the discard pile."
            x = True
            while x:
                x = False
                user = raw_input('Which card do you wish to discard?
(Enter number only) ')
                if hand.count(int(user)) == 1:
                    index = hand.index(int(user))
                    hand.insert(index,card)
                    card = hand.pop(index+1)
```

```
                    faceup_deck.insert(0,card)
                elif int(user) == card:
                    faceup_deck.insert(0,card)
                else:
                    x = True
                    print "Card not found. Please pick again" # repeat
command
                    print ""

    return [facedown_deck,faceup_deck,hand]


def create_hands(num_players, facedown_deck):
    i = 0
    created_hands = []
    while i < num_players:
        hand = []
        j = 0
        while j < 10:
            temp = draw_facedown(facedown_deck)
            hand.append(temp[0])
            facedown_deck = temp[1]
            j += 1
        created_hands.append(hand)
        i += 1
    return [created_hands,facedown_deck]

def check_if_won(ls):
    hand = True
    for i in range(1,10):
        if(ls[i-1] > ls[i]):
            hand = False
            break
    return hand

def shuffledeck():
    a = range(1, 61)
    random.shuffle(a)
    return a

def print_rules():
    print "Number of Players: 2 to 4"
    print ""
    print "Objective: Align your hand (10 cards) into ascending order."
    print "First to do so wins the game."
    print ""
    print "How to play: It's a fairly straightforward game. At the
beginning"
    print "of your turn you either draw from the discard pile or draw"
    print "from the deck (face-down) then you pick a card from your
hand (or"
    print "the one you drew) to discard and your new card takes its
place."
    print "You are not allowed to change the order of the cards in your
hand."
    print "The cards are labeled from 1 to 60."
    print ""
```

```
print "Typical Play Time: 15 minutes"
```