Project 2: The Curious Card Conundrum

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

1 Change Log

With a common lab assignment we do not typically have time to change the lab in response to student questions, concerns, and common misunderstandings. However, as this project lasts two and a half weeks, it is relatively common for small updated, extra hints, and minor typos to be added. This page will list any such modifications. **I recommend checking occasionally for updates on canvas and listening for updates announced in lecture**

- Version 1.0 Initial Version – grading details are still subject to change a bit.

1.1 FAQS

1. How closely your Tournament outputs should match ours. It should be close (+- 0.1 at the most, probably closer) but not exact.
   - If your Tournament values seem *Backwards* there may be many issues, but I would check by seeing if you're reporting the right winner and counting correctly.
   - If your scores are off after that there is little *direct* debugging we can do. (There is little direct relationship between what numbers you get in tournament and what errors you have in your code. Asking us to identify a problem from only those numbers is asking for the impossible)
     - Check both your AI and your GAME. Minor changes in how the game is played (AI 1 always goes first, or hands are redrawn between rounds) can lead to quite different outcomes.
     - Carefully re-read the Game description – look for every detail and check if your code implements that detail as written.
     - Add print statements to your code. Lots of them. Use these print statements to *watch a game get played* Make sure the behavior of your game matches your expectations
     - Split your code into smaller functions – test and review each function in isolation.

2 Essential Information

2.1 Pacing and Planning

This is a 3-week long assignment. As such you should expect it to be longer, and more complicated, than past assignments you have undertaken. Similar to last project – there will be several moments in this project where you need to make some key design decisions. As before – **plan time for understanding and designing code for this problem**.

2.2 Deadline

The deadline for this assignment is Friday Nov 10th at 6:00PM. Late work is not generally accepted on projects after 72 hours – but beyond this grace period late work will not be accepted without an exception.

2.3 Files

This project uses a fairly typical object-oriented design. This means that you will have many required files, but each file will not necessarily be super-big. All these files must be in the student package. (located in a directory student inside of src and have package student; at the top of the file)

- `Card.java`
- `Deck.java`
- `Hand.java`
- `CardPile.java`
- `UnoWarMatch.java`
- `AI.java`
- `SmallestCardAI.java`
- `BiggestCardAI.java`
- `Tournament.java`

Additional testing files may be provided over the course of the project, but ultimately you will be responsible for testing, and debugging, the code on your own.

2.4 Other important restrictions

Individual project – Unlike labs, where partner work is allowed, this project is an individual assignment. This means that you are expected to solve this problem independently relying only on course resources (zybook, lecture, office hours) for assistance. Inappropriate online resources, or collaboration at any level with another student will result in **a grade of 0 on this assignment**, even if appropriate attribution is given. Inappropriate online resources, or collaboration at any level with another student without attribution will be treated as an **incident of academic dishonesty**.

To be very clear, you can ask other students questions only about this document itself ("What is Daniel asking for on page 3?"). Questions such as "how would you approach function X", "How are you representing data in part 2?", or even "I'm stuck on part B can you give me a pointer" are considered inappropriate collaboration even if no specific code is exchanged. Coming up with general approaches for data representation, and finding active ways to become unstuck are all parts

of the programming process, and therefore part of the work of this assignment that we are asking you to do independently. Likewise, you are free to google search for any information you want *about the java programming language and included java classes* but it is not reasonable to look for information about the problem in this project (I.E. "how to program card games" would be an inappropriate google search)

Allowed java classes There are A LOT of java classes, which solve many interesting problems. You are not allowed to use any of the pre-built java classes except those we've explicitly discussed in class such as Scanner, Random, and String. If you are unsure if a java class would be allowed here you should ask. Using unapproved java classes can lead to failing this assignment.

Specific examples of BANNED java classes:

- Any subclass of java.util.List (java's list classes)
- Any subclass of java.util.Map (java's dictionary classes)
- Any subclass of java.util.Set (java's set classes)
- java.util.Collections

Part of our goal here is practicing data storage using arrays. Every data storage task can be done by-hand without these advanced data structures.

3 Introduction

Over spring break, my friend and I were playing a card game. The game was kind of a strange mix of Uno and War, played with a standard deck of playing cards. Each player has a hand of 5 cards. The game starts by dealing a card from the deck onto a pile in the middle. Players then take turns playing a card from their hand. The card they play must have the same suit, as the card in the middle, or the same or higher rank. Play goes back and forth until one of the players is unable to play a card. At that point the other player gets one point. The first player to 10 points wins.

As far as things go this is a rather silly game. The game feels pretty "random" as you play it, but there is *some* strategy to it. In fact my roommate was winning pretty regularly against my approach. Therefore, I came up with a cunning plan: I would program a version of this card game in java and test several simple AIs against each other. By seeing what types of AIs perform the best I would learn what strategies work the best.

As a software project, this will have two core parts:

- Representing Card games (in general) and the UnoWar card game (in particular). This part of the project will focus on standard java objects and arrays.
- Representing different approaches to playing this game, and writing code to automatically compare strategies. This part of the project will focus on inheritance and polymorphism.

You can think of this as two "layers" to the code. First you'll need a "layer" simply to allow java programs talking about / thinking about the card game. Then you will write MORE core using this first "layer" of code, which will automatically compare stratgies for our game.

3.1 Learning Goals

This assignment is designed with a few learning goals in mind:

- Implement several java objects of varying complexity.
- Make a few object representation decisions based on a general description of an object
- Practice using arrays for task-specific data structures (this will help prepare you for general-purpose data structures later!)
- See a specific fully-worked-out example where polymorphism and inheritance allow much more general and generic code.
- See how AI driven simulations can provide insight into competitive game playing.

4 Theory: The UnoWar card game

The purpose of this section is to teach the rules of the UnoWar Card game *very detailed* understanding of this game will be required to correctly implement it's rules into Java. I recommend reading this section carefully, and if you have cards, trying to play a game or two on your own to make sure you understand the rules. I also recommend **taking notes on this section**.

The card game you will be implementing is a rough mix of Uno and War. The game is played with a standard deck of cards. Each player has a 5 card hand, and draws from the same deck of cards. After a player plays a card they always draw a replacement (so hands always have 5 cards) and if the deck is ever empty – it is immediately re-shuffled (so the players can always draw another card). (Obviously, in real-life play when reshuffling the deck it will only have cards that are not

in hands. However, this is quite hard to program well with the tools we have so-far. Therefore we will instead simulate the game as if each time the deck is empty we open and shuffle a new 52 card deck of cards.)

The game of Unowar is structured into rounds. Each round players play until someone loses the round – the other player (that rounds winner) gets one point. The winner then has to go first the next round (first round is normally given to "player 1" – chosen before the game begins) Rounds are played back-to-back, so players keep their hands between rounds, and making sure you save good cards for later rounds is a deliberate part of the strategy of the game. The game is over when one player has 10 points (wins 10 total rounds).

A single round of Unowar starts by dealing one card from the deck and placing it in the "card pile" between the two players. Then players take turns playing a card into the card pile from their hand. Cards can only be played if they meet one or both of the following conditions:

- same suit as the card in the middle (suit is the symbol on the card: clubs, spades, diamonds, or hearts)
- equal or higher rank than the card in the middle (rank is the number of the card: aces, twos, and threes are low, 10, jacks, queens, and kings are high, and so-forth.)

If a player plays a card that does not meet one or both requirements then that player looses the round. If a player does not have any cards that they can play, they also lose the round. Players keep going this way until the current player can't play – meaning they lose. In theory a single round can last quite a long time with the right cards. Regardless of who wins/loses, the card pile is discarded at the end of the round (As a new card pile will be started in the next round with a new card from the deck).

So a game of uno war would look like this:

1. Shuffle the deck of cards
2. Deal two 5-card hands two player 1 and player 2
3. Player 1 plays first in the first round
4. while no player has won 10 rounds play one round:
   (a) deal a card to start the card pile in the middle
   (b) Starting with player 1 in the first round, or whoever won the last round in all other rounds, players take turns picking who goes first.
   (c) if a player has no legal cards, that player loses the round, and the other player wins. The other player will have to go first next round.
   (d) Discard the card pile in the middle.
5. Whoever won 10 rounds is the winner of the game!

Take a moment to review this outline You will ultimately need to program this logic. Take a moment to make sure you understand *every little detail* of this – as there are a lot of important details that effect the natrue of the game. For example, if you make player 1 go first every time the game becomes much less fair than if the player who won last round goes first. Once you think you have the details, you might want to take a moment to think out how to divide this logic into useful functions. For example, you could have a function to handle one turn, or one round. As you do this, think about how you will track details like who goes first in which round, or whose turn is next. This thinking will come in useful later – so take notes now while the design of the game is fresh in your mind!

4.1 UnoWar strategy

While there are theoretically infinite possible strategies, we will only look at 3 simple strategies. You are of course encouraged to explore more – see if you can find the BEST (or worst) possible strategy!

- The basic AI will be a "baseline" – this means we don't expect it to perform well, but we think it will be useful to compare against. This basic AI will play a random **valid** card each turn. (Strictly, we will ask you to program the AI to play the first valid card in their hand – this is effectively random sincew we don't sort hands anyway) Since this AI has no real "strategy" – any AI that does better than it can be said to have a good strategy, and any AI that does worse can be said to have bad strategy (worse than not trying!)
- The second AI will always play the smallest-ranked valid card in their hand. This strategy is designed to give the player the most options in the future. As a downside – however, this conservative strategy also leaves the most options open to the opponent, and tends to lead to longer games.
- The third AI will always play the highest-ranked valid card in their hand. This strategy is designed to force decisive victories by limiting their opponents choices. This tends to lead to fast games – Playing your strongest card can backfire if the opponent has stronger cards than you!

5 Requirements

The design for this program is very typical of an Object-Oriented design. This is to say, it has many classes, most of which are small and serve one well defined purpose:

- `Card` - represents one playing card
- `Deck` - represents a deck of playing cards
- `Hand` - represents a hand of playing cards
- `CardPile` - represents a pile of playing cards
- `UnoWarMatch` - represents a match-up of two AIs at UnoWar (one match-up may be settled by thousands of actual games, this object handles both single games, and multiple game series)
- `AI` - a Java class. This serves both as a Random AI, and the parent class for all other AIs.
- `SmallestCardAI` - an AI that plays the lowest-rank valid card in its hand.
- `BiggestCardAI` - an AI that plays the highest-rank valid card in its hand.
- `Tournament` - a driver class with a main method that reports the win-rate for every possible pair of AIs.

While this may sound like many classes to write, most of them are relatively simple on their own. By splitting the behavior up like this you can also focus on single classes at a time. You could, for example, try to write one (and only one) class a day and be done in a little over a week.

5.1 Card

The `Card` class represents a single playing card. There are MANY ways to represent a playing card in any modern programming language. We are going to represent these with two integers: `rank` (the number on the card) and `suit`. For rank we will use 1 to represent "Ace", 2 to represent

"Two", and so-forth until we hit 11 (Jack) 12 (Queen) and 13 (King). For suit we will use 1 to represent Spades, 2 to represent Hearts, 3 to represent Clubs, and 4 to represent Diamonds.

Your card object should be immutable, meaning that once constructed there should be no way to change it. Your Card class can have any other variables or methods you want, but it must have all of the following methods. These methods must have these exact names, and parameter types (although you can name the parameters differently)

- `public Card(int rank, int suit)`
  Constructor - the first `int` should indicate the rank of the card (1 = Ace, 2 = Two, ..., 11 = Jack, 12 = Queen, 13 = King) The second `int` indicates the suit 1 = Spades, 2 = Hearts, 3 = Clubs, 4 = Diamonds. This constructor should validate it's inputs – if an invalid suit or rank is given it should print an error message (See the tester code for the exact expected message) and then set private variables to be the Ace of Spades.
- `public int getRankNum()`
  This method should return the number representation of the cards rank.
- `public String getRankName()`
  This method should return the string naming the cards rank. (I.E. "Ace", "Four", "Ten", "Queen" etc.) All rank names should be capitalized
- `public String getSuitName()`
  This method should return the string naming the cards suit ("Spades", "Hearts", "Clubs", or "Diamonds")
- `public String toString()`
  Your Card should override the default toString method to one that prints a human readable name for the card. Once written this will help you when using print to debug since it provides a human readable description of the card. Examples of the type of output we are looking for can be found in the test files.
- `public boolean equals(Object obj)`
  You card class should override the default equals method. A Card should only be equal to other instances of the Card class, and then only other cards that have the same rank and suit. **note** This is not a typo – the parameter should be Object, not Card here. We will discuss this function, why it is the way it is, and how to program it, in lecture.

5.2 Deck

The `Deck` class represents a deck of cards. It must use an array of type cards (length 52) to represent the cards and their current order. You will also need at least one additional variable to track which cards have been dealt, and which have not.

NOTE/WARNING I'm giving you extra design freedom for how to make the private internal variable management of this class work. You are required to store cards in an array, but beyond that we are not telling you how to track which cards are used and which are not. As such, you should expect some many of the function descriptions are written *abstractly* representing the external view of the object, not it's internal view. For example, the *draw* method will say it "removes a card from the deck" – since you are storing your deck in an array you likely *will not* actually remove the card from the array, but instead would update variables so that this card cannot be drawn again. You should seek an efficient way to do this, and think carefully about the $O(1)$ efficiency requirements below. As a hint – one integer, used well, can solve this problem.

We are going to implement Deck so that if you draw from an empty deck it automatically reshuffles. In the real world reshuffling a deck of cards would only replace cards not otherwise in use. To make life simpler, however, we will reshuffle a new deck of 52 cards. You can think of this like opening a new deck of cards, rather than trying to collect all the discarded cards. While this may lead to certain cards being duplicated in our game, it shouldn't substantially effect the quality of our simulation, so it should be fine.

Your `Deck` can have any other variables or methods you want, but it must have all of the following methods. These methods must have these exact names, and parameter types (although you can name the parameters differently)

- `public Deck()`
  Constructor - creates a new deck. Makes an array containing 52 different Cards. You must use one or more loops: you will receive no points if you just write 52 assignment statements. The order of Card's within the array does not matter. The last line of your constructor should call your shuffle method so that all decks are shuffled by default.
- `public void shuffle()`
  Shuffle the deck of Card's that is represented by the array you made in the constructor. The easiest way is the Durstenfeld-Fisher-Yates[1] algorithm, named after its inventors. The algorithm exchanges randomly chosen pairs of array elements, and works in $O(n)$ time for an array of size n. You must use the following pseudocode for this algorithm.
  Do the following steps for the integer values of i starting from the length of the array minus one, and ending with 1.
  1. Let j be a random integer between 0 and i, inclusive.
  2. Exchange the array elements at indexes i and j.

  (To generate random numbers, please use Java's Random object, and in particular the Random class method `public int nextInt(int bound)` which generates a number between 0 and *bound* (not including *bound*)).
- `public Card draw()`
  Draw and return the next card. Whatever card is drawn should not be drawn again until the deck is shuffled again. This should decrease the number of cards remaining. Note you do not want to pick a card at random here – the `Card` array should be in a random order, so you should come up with a way to return the "next" card that has not been drawn. This method must work in $O(1)$ time (unless it needs to reshuffle). If the deck is empty it should shuffle the cards and then deal the new top of the deck.
- `public int cardsRemaining()`
  Returns the number of cards remaining before the next reshuffle. This should return 52 after constructor or a call to shuffle, and decrease with calls to draw down to 0.
- `public boolean isEmpty()`
  Returns whether or not the deck is empty. If this returns true, the next call to draw will trigger a reshuffle.

[1]For more information on this algorithm I recommend the Wikipedia page it is both informative, and currently free of actual Java source code (looking up source code for this algorithm would be cheating, so be careful what you search)

5.3 Hand

The `Hand` class represents a hand full of cards. To make this class easier to use, we will make the hand class automatically draw cards from a deck as cards are removed from the hand. While we will only use one hand size for this specific program, the `Hand` class should be designed to work with any sized hand. Designing it this way will let this class be reused in the future.

The `Hand` class must represent the hand of cards using an array of cards. You will need other private variables than just the array – we will give no hint for this.

- `public Hand(Deck deck, int size)`
  Constructor. This should create an array to store cards of the given size, and then draw it full of cards using the supplied deck.
- `public int getSize()`
  Get the size of the hand.
- `public Card get(int i)`
  Get the card at the given index in this hand. If the index is 0 it would be the first, card, 1 should give the second card, etc. If an index is given that is out of bounds this method should print an error (see the tester code for the exact string) and then return the first card.
- `public boolean remove(Card card)`
  This method should remove a given card from the hand. If the card is found in the hand it should be removed, and a replacement card should be drawn from the deck (specifically the instance of `Deck` that was passed as a parameter to the constructor). In this case the method should return true. If the card is not found in the hand it should return false. You should design this method to work if the input card is null (you can, however, safely assume that the cards in the hand are not null)

5.4 CardPile

The `CardPile` class represents the pile of cards that players play onto, and is where several of the rules of the game are implemented. We *could* use an array or some other data structure to represent the pile of cards that have been played. However, since we only ever need to reference the top card – this will not be necessary, and you can accomplish this with a single Card variable (plus an int variable to track the size)

The methods the card pile should implement are described below:

- `public CardPile(Card topCard)`
  Constructor. This should create a new card pile with the given card as the initial top card.
- `public boolean canPlay(Card card)`
  This method should check if the input card is legal to play on the current stack. As a reminder of the rules for this: a card can be played if it has a higher rank than the current top card, has the same rank as the current top card, or if it has the same suit as the top card.
- `public void play(Card card)`
  Adds another card to the card pile, making this the new top card. If the input card is not legal to play on the top of the card pile, this method should print an error message (see the tester code for this error message) and make no other change.
- `public int getNumCards()`
  Gets the number of cards in the `CardPile`.

- `public Card getTopCard()`
  Gets the current top card for this `CardPile`

5.5 AI class

As a reminder, the AI class has two major goals in our design. First, it will play the role of "parent class" to our other two AIs. This will let us use polymorphism in the Game code and easily work with any AI subclass. Secondly, this class will also serve as a baseline "no strategy" AI strategy by choosing cards essentially at random.

The AI class should have two methods:

- `public Card getPlay(Hand hand, CardPile cardPile)`
  This method takes two parameters, the hand, full of cards the AI is allowed to play, and the cardPile that the AI is playing on. An AI (here we're talking about any-subclass) should pick a card from the hand and return it to mark it as the card the AI intends to play. The AI can return null to indicate that they have no card that can be played on this card pile. The AI is not responsible for removing the card from the hand – this will be managed by another class – just choosing which card to play. The AI should not return a card that is not in the input hand.
  - for this specific class (the AI class, not it's subtypes) – this function should pick the first card in the hand that is valid processing from left-to-right. Since we don't sort the cards in the hand – this is effectively random.
  - Other AI subclasses will work in other ways.
- `public String toString()`
  This method should return the name of the AI. For the AI class itself, that's "Random Card AI"

5.6 SmallestCardAI

The SmallestCardAI class should be a subclass of AI and should override both methods:

- `public Card getPlay(Hand hand, CardPile cardPile)`
  This AI class should return the smallest-rank valid card in the hand. (Ties can be broken arbitrarily - we do not require any specific policy)
- `public String toString()`
  This method should return the name of the AI. For the SmallestCardAI that's "Smallest Card AI"

5.7 BiggestCardAI

The BiggestCardAI class should be a subclass of AI and should override both methods:

- `public Card getPlay(Hand hand, CardPile cardPile)`
  This AI class should return the biggest-rank valid card in the hand. (Ties can be broken arbitrarily - we do not require any specific policy)
- `public String toString()`
  This method should return the name of the AI. For the BiggestCardAI that's "Biggest Card AI"

5.8 UnoWarMatch

The `UnoWarMatch` class contains the majority of the code involving playing a game of UnoWar between two strategies that are implemented as instances of the `AI` class. This class sits at the intersection of the two major "parts" of this project: playing UnoWar, and comparing AIs. As such it has two core functions: one plays a single game of UnoWar – reporting who won, The other plays MANY games of unoWar, reporting the how often player1 Won as a double. Clearly these two functions are closely related (the "play many times" function just needs to call the "play once" function in a loop and count...)

Note, the only instance variables for this class should be two AI objects. While there are other attributes such as a deck of cards, hands, or win/loss rates that you will be tempted to make instance variables, all of these are needed only for one method, and can therefore be local to that method. Making these variables function-local (instead of instance-variables) will save you a lot of debugging and headache.

While we only mandate the following three methods for UnoWarMatch, we recommend that you make several other private methods. The process of playing a single game of UnoWar is complicated enough that decomposing it into methods will be useful.

- `public UnoWarMatch(AI ai1, AI ai2)`
  Constructor. This takes the two AIs that this `UnoWarMatch` class is intended to compare.
- `public boolean playGame()`
  Play a single game of Uno War. This should play the UnoWar game as described earlier in the write-up until one of the AIs has won 10 rounds. The return value should be true if ai1 wins, and false if ai2 wins.
  As a brief reminder: The basic flow of the game is done in rounds, with each round being worth one point. Each round the AIs take turns playing a card from their hand into the card pile. The round ends when one AI has no valid card in their hand, at which point the other AI gets a point, and plays first in the next round.
  Before you start building this function I recommend re-reading the full-detail description of the UnoWar game rules. Most people work on this class towards the end of the project, and it would be easy to have forgotten some of the required details and design of the UnoWar game. Even *quite small* changes to the correct function of the game can lead to *quite large* differences in ultimate outcomes to this project.
  Some details on this method:
  - This method should start by constructing a new deck, and hand objects – these are not reused game-to-game, but should stay the same for the entire process of one game.
  - Remember that the CardPile should be created new each round of the game.
  - This method should make sure cards are removed from the AIs hands once they are chosen for play. We do this here instead of in the AI class so that we can't write a cheating AI.
  - Remember, the AI can chose to play null if no card is valid – make sure your code can handle "null" here.
  - AI 1 should play first in the first round.
  - In all other rounds, the AI who won last round should go first in the next round.
- `public double winRate(int nTrials)`
  This method should have the AIs play each other `nTrials` times, and report the percent of

times AI 1 beat AI2 as a double. The return value should be between 0 and 1 inclusive, where 1 means "AI 1 always won", and 0 mean "AI 2 always won" Since the winner of this game is partially determined by chance (the best AI can't win if it only gets bad cards) the game may need to be repeated thousands of times to get a precise estimate of the chance of one AI beating another. But, if you repeat thousands of times, you should get relatively reliable numbers.

5.9 Tournament

This class stores the actual runable program for this project, meaning it only needs to have a main method. The main method should print the win rate of every pair of AIs. The output of my program is given below: Your code does not need to output this exactly (in fact, I would be surprised if it did, as estimated winRate is somewhat random) but it should contain all this information. Make sure you clearly label which winRate goes with which pair of AIs, and test AIs in both AI1 and AI2 position (as the game may be effected by who goes first in the first round!) When calling the winRate method to compute win rates, I recommend using nTrails around 1000 or higher – less than that can be heavily subject to random chance. That said, while testing if your code works, you are free to test with smaller nTrials for speed.

```
Random Card AI vs. Random Card AI winRate: 0.499
Random Card AI vs. Smallest Card AI winRate: 0.002
Random Card AI vs. Biggest Card AI winRate: 0.842
Smallest Card AI vs. Random Card AI winRate: 0.998
Smallest Card AI vs. Smallest Card AI winRate: 0.499
Smallest Card AI vs. Biggest Card AI winRate: 0.999
Biggest Card AI vs. Random Card AI winRate: 0.156
Biggest Card AI vs. Smallest Card AI winRate: 0.0
Biggest Card AI vs. Biggest Card AI winRate: 0.491
```

6 Testing and incremental development

This project is quite typical of an object-oriented program design: There are A LOT of things to program, many of which are individually quite small. The idea here is to spread the complexity of the program over more code – that way no one function is individually super-complicated. This leads to longer development time, but often faster testing and debugging, since each function and class can be debugged individually.

To make this process faster, we've provided testing code for several of the simpler classes (those with limited random behavior) Since some of these tests rely on the behavior of other classes (you can't test Deck without a working Card class, for example) there is an intended order to these tests. The intended order of completing these tests is as follows:

1. `CardTest.java`
2. `DeckTest.java`
3. `HandTest.java`
4. `CardPileTest.java`
5. `AITest.java`

Note, these tests only cover some of the simpler classes. Once you pass all these test you will still need to manually evaluate your UnoWarMatch class for correctness. The easiest way to do this is to add print statements so that you can "watch" the games get played, and confirm that the rules are being followed.

To help you plan your progress on this assignment, I've developed a rough development schedule:

- Week 1 goals: read the PDF, play (by hand) a game of UnoWarm, write the Card, Deck, and Hand classes.
- Week 2, write the AI class, CardPile, and UnoWarMatch.
- Week 3, Finish and debug UnoWarMatch, write the other two AIs, test and debug carefully before submission.

7 Submission

For this project you should submit the following files:

- `AI.java`
- `BiggestCardAI.java`
- `Card.java`
- `CardPile.java`
- `Deck.java`
- `Hand.java`
- `SmallestCardAI.java`
- `Tournament.java`
- `UnoWarMatch.java`

These files must be submitted on gradescope before the deadline (found at the beginning of this document) There is no plan to accept late work on this assignment without an extension.

8 Grading

The autograder will be based on the test code, and will be worth 18 points. **This autograder is not even remotely comprehensive.** Much of the code in this project is not well suited to testing, and making it *more testable* would alo involve telling you how to structure the code (which is something you need to be deciding for yourself!) **Make sure you are manually reviewing your code and the assignment PDF so you don't lose the 82 points based on direct code review.**

The remaining points will probably be split roughly as follows:

- 10 points for code style
- 12 points for the Card class
- 12 points for the Deck class
- 6 points for the Hand class
- 6 points for CardPile class
- 6 points for the AI class,
- 6 points for the SmallestCardAI
- 6 points for the BiggestCardAI
- 16 points for UnoWarMatch
- 2 points for the Tournament class