1. **Provide written response that does all of the following :**
   - **Describes the overall purpose of the program**
     The main goal of this blackjack iOS app is to give people a digital way to play blackjack. The app tries to make playing blackjack feel like it would in a real casino, but also lets people play on their mobile devices for added convenience.

   - **Describes what functionality of the program is demonstrated in the video**
     The functionality demonstrated in this blackjack iOS app is that the user plays as the player, while the dealer is controlled by the CPU. The app restricts the player's options to two basic moves, "hit" or "stand", thus simplifying the game to its core mechanics. The program implements simple logic of blackjack : dealing cards, executing hits, betting, determining winners, and calculating scores.

   - **Describes the input and output of the program demonstrated in the video**
     The user interacts with the app by making decisions, such as choosing to "hit" or "stand", and indicating their desired action through the app's user interface. The app then displays the cards dealt to the player and the dealer on the screen. It also provides messages to the user, such as "Dealer lost, you won!", to inform the user of the outcome of each hand, and keeps track of the scores for each player, displaying the current scores on the screen.

2. Capture and past two program code segments you developed during the administration of this task that contain a list (or other collection type) being used to manage complexity in your program

   - **The first program code segment must show how data has been stored in the list**

```
@State var playerCards: [Card] = []
```

   - **The second program code segment must show the data in the same list being used, such as creating new data from the existing data or accessing multiple elements in the list, as part of fulfilling the program's purpose.**

```
func hitCard() {
    if playerBust == false {
        playerCards.append(deck[deck.count - 1])
        playerTotal += playerCards[playerCards.count - 1].rank
        deck.removeLast()
        turnNumber += 1
        playerHit = true
        if playerTotal > 21 {
            playerBust = true
            playerWins = false
        }
    }
}
```

Then, provide a written response that does all three of the following

- **Identifies the name of the list being used in this response**
  The name of the list being used in this response is called ‘playerCards’

- **Describes what the data contained in the list represents in your program.**
  The data contained in the list will store the cards that the player has been dealt. Eg. Hearts of 5, etc

- **Explains how the selected list manages complexity in your program code by explaining why your program code could not be written, or how it would be written differently, if you did not use the list.**
  1. The use of the list helps manage complexity by allowing the program to store multiple pieces of related data in a single and organized structure. Without using a list, the program would need to use separate variables to store each card that the player receives. This would lead to complex and inefficient code, as the number of variables would increase with each card dealt to the player. It would also make it more difficult to access and manipulate the data, as the information about the player's cards would be scattered across multiple variables. In addition, when using the append() method as shown in the code segment, the program is able to add new elements to the end of the list in a simple and efficient manner, which would be much more complex if separate variables were used, and the removeLast() method allows the program to remove elements from the end of the list, which would also be much more complex if separate variables were used.

- **3. Capture and paste two program code segments you developed during the administration of this task that contain a student-developed procedure that implements an algorithm used in your program and a call to that procedure.**
- **i. The first program code segment must be a student-developed procedure that:**
  - **a. Defines the procedure's name and return type (if necessary)**
  - **b. Contains and uses one or more parameters that have an effect on the functionality of the procedure**
  - **c. Implements an algorithm that includes sequencing, selection and iteration**

```
struct Card {
    let rank: Int
    let suit: String
}

func fisherYatesShuffle(deck: inout [Card]) {
    setFisherYatesShuffle = true
    for i in (0..<deck.count-1).reversed() {
        let randomIndex = Int.random(in: 0...i)
        if i != randomIndex {
            deck.swapAt(randomIndex, i)
        } else {
```

```
            continue
        }
    }
}
```

- **The second program code segment must show where your student-developed procedure is being called in your program.**

```
func createDeck() -> [Card] {
    for rank in ranks.keys {
        for suit in suits.keys {
            let card = Card(rank: rank, suit: suit)
            deck.append(card)
        }
    }

    if setFisherYatesShuffle == true {
        fisherYatesShuffle(deck: &deck)
    } else if setKnuthShuffle == true {
        knuthShuffle(deck: &deck)
    }

    return deck
}
```

- **Describes in general what the identified procedure does and how it contributes to the overall functionality of the program.**
  The identified procedure, "fisherYatesShuffle", is a shuffling algorithm used to rearrange the elements in a deck of cards in a random order. This procedure contributes to the overall functionality of the program by allowing the program to shuffle the deck of cards before starting a game. Since the cards are going to be shuffled fairly and randomly with this procedure, the procedure adds an element of randomness to the game and makes it more challenging and entertaining for the player, making it more difficult for the player to predict the order of the cards and providing a more enjoyable gaming experience

- **Explains in detailed steps how the algorithm implemented in the identified procedure works. Your explanation must be detailed enough for someone else to recreate it.**
  1. Create a parameter that takes in a deck of cards
  2. Initialize a variable setFisherYatesShuffle to true.

3. Loop through the deck of cards in reverse order, starting from the last index deck.count-1 and ending at index 0.
4. For each iteration of the loop, generate a random index within the current range of the loop using Int.random(in: 0...i) - so in our cards example, from 0 -> 53.
5. If the current index i is not equal to the randomly generated index randomIndex, swap the element at the current index i with the element at the randomly generated index randomIndex. Otherwise, continue step 4 until the current index i is not equal to the randomly generated index randomIndex.
6. Repeat steps 3-4 for each iteration of the loop until all elements in the deck have been processed.
7. The deck will now be shuffled randomly.

- **Describes two calls to the procedure identified in written response 3c. Each call must pass a different argument(s) that causes a different segment of code in the algorithm to execute.**
  First Call:

```
func createDeck() -> [Card] {
    for rank in ranks.keys {
        for suit in suits.keys {
            let card = Card(rank: rank, suit: suit)
            deck.append(card)
        }
    }

    if setFisherYatesShuffle == true {
        fisherYatesShuffle(deck: &deck)
    } else if setKnuthShuffle == true {
        knuthShuffle(deck: &deck)
    }

    return deck
}
```

Conditions tested by the first call

```
setFisherYatesShuffle = true
    for i in (0..<deck.count-1).reversed() {
        let randomIndex = Int.random(in: 0...i)
        if i != randomIndex {
            deck.swapAt(randomIndex, i)
        } else {
            continue
        }
    }
```

Result of the first call:
The result of calling fisherYatesShuffle(deck: &deck) is that the input array deck is shuffled in place, meaning that the elements of the array are rearranged randomly. There is no output for the function as it does not return anything.

Second call :

```
var array = [1, 2, 3, 4, 5]
fisherYatesShuffle(deck: &array)
```

Condition(s) tested by the second call:

```
setFisherYatesShuffle = true
    for i in (0..<deck.count-1).reversed() {
        let randomIndex = Int.random(in: 0...i)
        if i != randomIndex {
            deck.swapAt(randomIndex, i)
        } else {
            continue
        }
    }
```

Result of the second call :
This call will shuffle the input array [1, 2, 3, 4, 5] using the Fisher-Yates shuffle algorithm. The output of the program will be a randomly shuffled array, which could be, eg. [2, 5, 1, 3, 4].