Additional Questions:

Questions:
1) How might you adapt your code to check for sequences of 3, 4, or more cards from the previous shuffle?

In the current implementation, the way to check if the current shuffled deck contains a sequence of two cards similar to the any two sequences in the previous shuffle is by keeping track of the previous shuffle values using a Map. The map contains a mapping between a card and it's next in line based on the shuffle order. So the way we know that two shuffles are unique is that

```
for each Card in the reshuffle,
        check if the next Card in the reshuffle for the current Card is the same as the next card in the
shuffle of the previous iteration.
```

The map helps in providing O(1) access to each element and hence it's next element in the shuffle sequence.

In the case where a sequence of 3,4 or more cards need to be checked, the map would contain the same key:value pair but the implementation of the comparison would not stop at at just one comparison.

The implementation of the comparison would traverse the map N times to follow the key:value pair.

example:

previous sequence 1 4 6 3

map:

1 : 4
4 : 6
6 : 3

current sequence

1 4 6 8

N = 3 so a matching 3 card sequence needs to trigger a re-shuffle

The main method will be implemented as follows:

```
        bool Deck::CheckBetweenShuffles(){

                //Only the first time will the shuffle map be empty
                if(shuffleMap.empty())
                        return false;

                // The loop will run for all the cards however
                // we increment i by j cards at a time since
                // we check the equality of the cards j cards at
                // a time with a maximum of j = N-1

                for(int i = 0; i < cards.size()-1; i +=j){
                        //Here N = number of cards in the sequence
                        int j = 0;
                        for(; j < N-1; ++j){
                            if(shuffleMap[cards[i+j].GetCardValue()] !=
            cards[i+j+1].GetCardValue()){
                                break;
                            }
                        }
                        //This means that the sequence was reached.
                        if(j == N-1)
                            return false;
                }
                return true;
        }
```

The runtime remains O(n) since if we don't find a match, we start the iteration of search from where the current Card is done being processed.

2) How might you adapt your code to check for combinations of N cards, rather than sequences?

In this case, Two sets of cards will cause a failure in the shuffling loop if set1 has the same cards as set2 regardless of the order of those cards.

example:
set1: hearts5 diamonds4 spadesAce spades6

set2: spadesAce diamonds4 hearts5 spades6

These are the same set of cards in a different order and according to the condition specified, this will cause a failure.

To solve this condition:

In the main for loop to check for sequences of a pair of cards,

1. Get the next N sequence of cards from the current shuffle and the previous shuffle
2. Sort the two sequences of cards.
3. Compare the two vectors of cards.
4. If they are not equal, then the combination is different and the loop will exit to reshuffle the cards.

```
        vector<int> currentShuffle, previousShuffle;

        for(int i = 0; i < cards.size()-1; i = i+j){
            for(int j = 0; j < N; j+=2){

            currentShuffle.push_back(cards[i+j]); // add the current card
            currentShuffle.push_back(cards[i+j+1]); //add the next card

            previousShuffle.push_back(cards[i+j]);
            prevousShuffle.push_back(shuffleMap[cards[i+j]]) // add the next card in the
prevous shuffle
        }

        sort(currentShuffle.begin(), currentShuffle.end());
        sort(previousShuffle.begin(), previousShuffle.end())

        //The vector compare checks if the elements are the same
        if((currentShuffle == previousShuffle) == false)
            return false;

    }
```

3) How might you adapt your code to check for card sequences that were present in any previous shuffle? (i.e. comparing against all previous shuffles, not just the most recent one)

This can be solved by
1. Defining a constant MaxPreviousShuffles. This will define how many of the past shuffle data will be stored.
2. Store all the previous transitions for each card (to an extent of MaxPreviousShuffles) in the shuffleMap in a
   vector of Cards.

This will gives us a history of the previous transitions between each card. So, the history will let us check if any other past transition for the current card was the same as the transition for the current shuffle.

Algorithm:
1. For each card in the current shuffle, get the vector of past transitions for this card
   from the shuffleMap.
2. Iterate through the past transitions for this current card and check if any of the past
   transitions match the current transition.

Example:

Current Scenario:

shuffleMap:

1 : 2
2 : 4
4 : 6
3 : 7

New Scenario:

shuffleMap:

1 : [2, 3, 3, 2 ..]
2 : [3, 4, 6, 7 ..]
4 : [5, 8, 9, 10 ..]

current shuffle : 1, 8, 2, 4

The main loop to check would be:

```
    for(int i = 0; i < cards.size()-1; ++i){
        // this will give us access to the vector of past transitions
        vector<int>& pastCards = shuffleMap[cards[i]];

        // Check if the next card in the current shuffle
        // was found anywhere in the past shuffles for this
        // card.
        for(int card : pastCards){
            if(card == cards[i+1])
                return false;
        }
    }
```

The runtime for this is : O(n * MaxPreviousShuffles) since we check all the past transitions for a particular card in the current shuffle.

4) If necessary, how might you change your code to accommodate more than 4 suits, or more than 13 different card values.

The current design already takes that into account. The user can enter the number of suits and cards per suit and
the Deck class will create a card deck with those number of cards.
Currently, the number of suits and the cards per suit are set to 4 and 13.

5) Would you do things any differently if you were optimizing purely for speed? How about if you were trying to minimize memory usage?

Memory:
The current algorithm performs with the best case of O(numberOfCards) in terms of memory. This is because we store the pervious shuffle values in a map. The map contains all the cards and their corresponding next card in a map.

Optimization:
1. *Remove the usage of a map by storing the previous shuffle values as a string and perform Longest Substring matching from the current shuffle.*

It's possible to store the previous card shuffle in a string instead and reduce the memory consumption and the hash function's processing requirement we currently have with the map.
The algorithm to find the sequence of two consecutive cards would be implemented by using a Longest Substring match algorithm on the current and the previous shuffles.

PseudoCode:

```
previousShuffle = StringOfAppendedCardValuesFromPreviousShuffle;
currentShuffle = StringOfAppendedCardValuesFromCurrentShuffle;

if(LongestSubstringMatch(currentShuffle, previousShuffle) > 2)
        then Reshuffle();
else
        return SUCCESS;
```

This has a runtime performance of O(n) in terms of memory to store the previous shuffle in a string and a performance of O(n*n) in terms of speed to perform the longest substring match.

2. *Remove the need for a map to store the previous shuffle and store the mapping between cards in the Card class. This will make the Card class more modular and also remove the memory needed to store the previous shuffle values in the map.*

The next card data that is stored in the map could be stored in the Card class itself. This would remove the need to have an extra map data structure in the Deck class.

```
class Card{
        uint64_t value;
        uint64_t nextCardValue;
};
```

Speed:
The current algorithm performs with a worst case of O(numberOfCards) in terms of processing speed. This is because
we have to iterate over all the cards in the current shuffle to find if the next card in the current shuffle is the same as the next card in the previous shuffle (from the map).

Since we need to access all the cards in the current shuffle, the most optimized algorithm would need a run time of O(n).