Function Name: `scrabbleScore`

Inputs:

1. (*double*) A 1xN vector of your word scores
2. (*double*) A 1xM vector of your opponent's word scores

Outputs:

1. *(char*) A string describing the outcome of the game

Banned Functions:

`sum(), cumsum(), mean()`

Function Description:

You and your friend just completed an intense game of scrabble. The scores were neck and neck, but now that the game has ended, you have to determine a winner. The worst part of any scrabble game is adding up the scores in the end, but with your newfound coding knowledge, you can use MATLAB to sum up the scores for you!

Given two vectors of scores for each word played by you and your opponent throughout a scrabble game, write a function called `scrabbleScore()` that totals up your and your opponent's score and determines which player won. If your final score is greater, output:

```
'I am the Scrabble champion!'
```

If however, your opponent's score was greater, output:

```
'Beginner's luck...'
```

If the scores were equal, output:

```
'I challenge you to a rematch!'
```

Function Name: `war`

Inputs:

1. (*double*) A 1x6 vector representing player 1’s starting hand
2. (*double*) A 1x6 vector representing player 2’s starting hand

Outputs:

1. *(char*) A statement saying who won and the total number of rounds in the game

Background:

Have you played the card game war and wanted to know who would win before you even start? You realize you can, with the help of MATLAB! With this function, you will take the card hand of each player and determine who will win the game of war. To play war, each player puts down the top card in their stack of cards. The player with the higher card wins both the cards and puts them at the bottom of their stack. This process is repeated until one person ends up with all the cards, and they are the winner.

Function Description:

For your function, you will be playing war with cards 2 through king (no aces), which will be represented by the numbers 2 through 13. Each of the two players will start off with 6 cards in their hand represented by the 2 input vectors. They will play starting with the first number in their vector and the winner will gain both the cards. The larger number will beat the smaller number EXCEPT for the 2 card which will beat the king (card 13) and only the king. The two cards that were won will go to the end of the winner’s hand, with the winning card first followed by the other person’s card.

The game ends when one player has all the cards. You should output the following string:

```
'Player <winner> defeated player <loser> in <num> rounds.'
```

<Winner> and <loser> correspond to the person’s number (1 or 2), and <num> corresponds to the total number of rounds.

Example:

Input vectors:

```
hand1 = [11 8 4 9 5 13]
hand2 = [12 6 7 10 3 2]
```

After 1 round:

```
hand1 = [8 4 9 5 13]
hand2 = [6 7 10 3 2 12 11]
```

After 6 rounds:

```
hand1 = [8 6 5 3]
hand2 = [12 11 7 4 10 9 2 13]
```

Continued...

After 10 rounds:

```
hand1 = []
hand2 = [10 9 2 13 12 8 11 6 7 5 4 3]
```

Output:

```
'Player 2 defeated player 1 in 10 rounds.'
```

Notes:

- The deck only has one of each card so you don't need to worry about ties.

Hints:

- The function `isempty()` will be useful!

Function Name: `blackJack`

Inputs:

1. (*double*) A 1x2 vector containing the card values in your hand
2. (*double*) A 1x5 vector containing the next five cards in the deck
3. (*double*) The total value of the opponent's hand

Outputs:

1. *(logical*) A logical stating whether you will win the round

Function Description:

As the director of an online Blackjack tournament company, not only do you get to play as "The House", but you also know the sequence of the cards. You will be given a 1x2 vector containing your initial two card values. The values of cards are as given: the number designation (`2-10`) on number cards are their value, face cards are worth `10` points, and an Ace is worth ONLY `11` points.

After determining the value of your hand, determine whether you have enough to beat your opponent's value (higher total value). If you do not, grab the first value in the deck vector and add it to your total value. The deck vector is a type double vector of numbers representing either a number card (`2-10`), or face value cards (`10`) and Ace (`11`).

As per the rules of Blackjack, if your total point value is more than `21`, you are considered "busted" and you lose the round. If your total is greater than the opponent's and is still 21 points or lower, you win. If your new total value is still not higher than your opponents, you need to add the next value in the deck vector to your total value.

Example:

```
>> in1 = [3 10];
>> in2 = [4 10 7 11 2];
>> in3 = 20;
>> out = blackJack(in1,in2,in3)
   out = false
```

Notes:

- You will always be dealt two cards (two values) in your "hand" vector (input 1).
- If your total value matches your opponent's value, you win.
- The opponent will always have a value less than 22.

Function Name: `goFish`

Inputs:

1. (*double*) A 1xN vector of the cards in player one's hand
2. (*double*) A 1xN vector of the cards in player two's hand
3. (*double*) A 1xM vector of the cards in the deck

Outputs:

1. *(char*) A string stating which player won
2. *(double)* A 1x2 vector of the final score

Function Description:

You and your friend are super bored playing Go Fish at a normal speed so you decide to spice your game life up with some Go Fish, MATLAB style! You will be given a vector of the cards in each player's hand -don't worry, jacks will be given to you as an 11, queens a `12`, and kings a `13`-. As in normal Go Fish, the game will end once one player runs out of cards. Player one will always go first.

Each player will "ask" the other for the first card in their hand, the first element in the hand vector. If the other player has one of the same type of card in their hand, remove the number from both hands and add a point to the proper player's score. If the other player doesn't have the same type of card, first take the first card from the deck and append it to the end of the hand, then take the card at the front of that player's hand and move it to the end of the hand (this makes sure they don't ask for the same card twice in a row). Repeat this process until one player doesn't have any cards left.

Before you play the game, ensure that the players' hands have no repeat cards. If they do have repeats, remove each pair of cards (if you have three of the same card in one hand, remove the first two instances of the card). Make sure that every time you draw from the deck, you check to see if it pairs with any other card in the hand. Anytime a player makes a pair, add a point to their score. The winner is the player with the most points at the end of the game, not the one who clears their hand first.

If player one wins, output the string:	`'Player 1 won!'`
If player two wins, output the string:	`'Player 2 won!'`
If it is a tie, output the string:	`'It's a tie!'`

Example:

```
        >> [winner,score] = goFish([1,2],[2,3],[1,4,4,3])
        score = [1,1]
        winner = 'It's a tie!'
%Player one asks player two for a 1. Player one draws a card from the deck
%and gains a point because it matched a card in player one's hand. player two
%asks player one for a 2 and gains a point. The game then ends
```

Function Name: `wordSearch`

Inputs:

1. (*char*) The unsolved puzzle, an MxN character array
2. *(char)* A PxQ array, with a different word on each line

Outputs:

1. *(char*) The MxN solved puzzle
2. *(double)* A Px2 array of the row and column indices of the first letter of each word

Function Description:

Fed up with only being able to search for one word at a time with your `quibbler` function, you decide to write a new function to solve word searches completely! You will be given an MxN character array of lowercase letters containing the wordsearch, and a PxQ character array containing a different word to search for on each line. Since you cannot have a jagged array, some of the the words will be followed by spaces.

Find where each word occurs in the array, then replace all its letters in the array with `'#'` characters to indicate where it is. You must also output a Px2 array of the row and column indices of each word's starting letter, with the row indices in the left column and the column indices in the right column.

Notes:

- Words may appear vertically or horizontally. They may be backwards or forwards.
- Words in the word list may be uppercase or lowercase.
- Words may overlap in the array.

Extra Credit

Function Name: `wordSearchHard`

Inputs:

1. (*char*) The unsolved puzzle, an MxN character array
2. *(char)* A list of comma separated words to be searched for in the array

Outputs:

1. *(char*) The PxQ solved puzzle

Background:

Anxious about the return of The Dark Lord, Harry Potter has been riding the muggle train system to keep his mind of things. However, that doesn't stop him from reading the Daily Prophet and his favorite tabloid, The Quibbler. With so much on his mind though, he accidentally leaves a copy of The Quibbler inside a coffee shop, only to be found by the muggle working for MATLAB Illustrated, the hip new magazine entering the market. Fascinated by the moving pictures and the magical word search inside, the muggle comes up with the ingenious idea to develop a new, much harder word search for the magazine.

Function Description:

After waking up late and making your coffee Sunday morning, you pick up your copy of MATLAB Illustrated from the porch only to find that there's a new type of word search in the entertainment section. It says that, given a character array of mixed upper and lowercase letters and a list of comma separated words to search for, find where each of the words exist in the array and replace each of its letters with `'#'` characters.

However, this is no simple word search. Words can appear horizontally, vertically, forward or backward, AND words can wrap around the edges of the character array!

There is also no guarantee that all of the words in the list will be found in the array. For every word that is not found, add 1 border layer around the array. The border should consist of `'@'` characters.

Example:

```
charArr =           list = 'dog,WIN,run,pun,fun'
       'AbudoG
        inrrpw
        Lynhan'
```

Continued...

```
solved =
      '@@@@@@@@@@
       @@@@@@@@@@
       @@Ab####@@
       @@###rp#@@
       @@Ly#han@@
       @@@@@@@@@@
       @@@@@@@@@@'
```

Notes:

- Words in the array the list can appear lowercase, uppercase or a combination of both.
- You must conserve the case of any characters in the array that aren't replaced by a `'#'`.
- Words in the list may appear multiple times in the array. You must "cross out" each of the occurrences of the word.

Hints:

- Think about when you should add the border to the array.

Extra Credit

Function Name: `blackJackHard`

Inputs:

1. (*double*) A 1x4 vector containing the percent risk values for the 4 players
2. (*double*) A 1xM vector containing the values of upcoming cards in the deck
3. (*double*) A 1xN vector containing the values of the cards in the discard pile

Outputs:

1. *(double)* A 1x5 vector of the final hand values of the 4 players and the dealer
2. *(double*) A 1xP vector of the numbers of the players that beat the dealer

Background:

Your online Blackjack company has been a huge success since you automated it in MATLAB, and it is starting to attract more and more professional blackjack players. You have received some complaints that your blackJack function does simulate the actual game well enough, so you decide you need to update your MATLAB function a bit.

Function Description:

Each game of blackjack will now have 4 separate people at the table playing against the dealer. The second and third inputs to the function are of the same format as the second input of the `blackJack` function, where cards of value 2-10 will be represented by that number, and Ace, King, Queen, and Jack will be represented by the ascii values 65, 75, 81, and 74, respectively. In this function Ace should be treated as having the value 11 in all cases EXCEPT when the value of a player's hand or the dealer's hand containing an Ace exceeds 21, and when calculating the chance of safe pick (see below). Then, the Ace should be treated as having the value 1.

The second input variable represents the next several cards in the deck. To start the game of blackjack, you must deal out cards to all the players. The dealer would first deal a card face down to Player 1, Player 2, Player 3, Player 4, and then to himself. The second round of cards is distributed in the same order, but face up, so that everyone at the table can see.

From here, play will begin with Player 1, who needs to decide whether to hit and receive another card, or stand, after which they cannot take any further action until the end of the round. To determine whether a player will hit or stand, they will perform "card counting" to determine the chance that they will remain safely at or below 21 after taking another card. This can be expressed as:

$$Chance\ of\ safe\ hit = \frac{\#\ of\ safe\ unknown\ cards}{total\ \#\ of\ unknown\ cards}$$

Continued...

To find which cards are unknown, you will have to use the cards that each player knows could not be in the deck, and “subtract” those from a full deck of cards. Input 3, which a vector representing cards in the discard pile from previous rounds of blackjack, is known to all players. In the formula, safe cards would be defined as cards that will not cause the player to bust. For example, if a player currently has an 8 and a 7 in their hand, any card of value 6 or below would be considered safe. NOTE: for this part of the problem, Aces should be treated as having the value 1, since a player with a current value of 15 in their hand could hit and receive an Ace resulting in a score of 16 rather than 26, so they would not bust. Once you have found the chance of safe hit, which should be a decimal between 0 and 1, you will need to use the first input to the function. Each of the four elements of this vector represent the “risk factor” of players 1 through 4, respectively. A player will only hit if they have calculated their chance of safe hit is greater than or equal to their risk factor. Otherwise, they will stand.

This behavior applies to players 1, 2, 3, and 4, but the dealer plays by dealer rules. For this function, we will define dealer rules as follows:

- Dealer must hit if their current hand value is less than 17.
- Dealer must stand if their current hand value is 17 or higher, unless…
- If the dealer has a soft 17 (a current hand value of 17, but one of their cards is an Ace), the dealer will hit.

Play proceeds from Player 1 to Player 2 to Player 3 to Player 4 to the Dealer, and then back around. The round ends when all of the 5 participants have either decided to stand, or have busted. Now we can calculate the final scores. The winning players will have a hand value greater than the value of the dealer’s hand, but not exceeding 21. A tie with the dealer does not count as a win. You will output a vector of the final hand values of players 1-4 and the dealer, in that order, and a second vector containing the numbers of the players that beat the dealer.

Notes:

- It is not guaranteed that inputs 2 and 3 will have a combined length of 52 (they may not represent the full deck of cards). The cards not given would be in the deck, unplayed, but not included in input 2.
- The length of the deck in input 2 is guaranteed to contain enough of the cards remaining in the deck to complete one round of blackjack.
- If a player or the dealer hits and receives more cards than their original 2, the new cards are face up so the whole table can see.

Hints:

- Make sure to be careful when determining which of the 52 cards each player knows cannot be in the deck.
- Think about how you can use helper functions to generalize certain parts of your function so your code will be shorter.