Function Name: `minMax`

Inputs:

1. *(double)* An array of doubles

Outputs:

1. *(double)* Minimum value in the array
2. *(double)* Maximum value in the array

Banned Functions:

`min(), max(), sort()`

Function Background:

You're serious gamer; a board gamer! To prove how good you are at board games to yourself and to the world, you place all of your scores for many different games into an array, but quickly realize that your board gaming nemesis stole the `min()` and `max()` functions from MATLAB! Using your new iteration skills, you write them yourself in order to use them to prove that your maximum score is amazing (and even your minimum score isn't really so bad).

Function Description:

Write a function that will find the maximum value and the minimum value of a given array.

Notes:

- You only need to iterate through the array once.
- The array will have at least one element in it.

Function Name: `primeTime`

Inputs:

1. (double) a number representing the upper limit of primes to output

Outputs:

1. (double) a 1 x N vector containing all prime numbers less than or equal to the cap number

Banned Functions:

`primes, isprime, nextPrime, prevPrime, ithprime,` or any other built-in prime functions

Background:

You look at the clock and see that it's prime time for television, but you decide to take it old school and play some board games instead! Before you can decide what to play, you wonder if now would really be the prime time for playing board games, which inspires you to use MATLAB to quickly calculate all of the primes up to a certain number.

Function Description:

Write a function that takes in a number, n, and outputs a vector of prime numbers up to n. Remember a prime number has only two factors: 1 and itself.

Example:

```
out = primeTime(7)
>> out =  [2 3 5 7]
```

Notes:

- All inputs will be 2 or greater
- 1 is not a prime number

Hints:

- The `mod()` function will be helpful to check for divisibility.

Function Name: `playChess`

Inputs:

1. *(char)* an Nx3 matrix representing the positions of pieces on the board

Outputs:

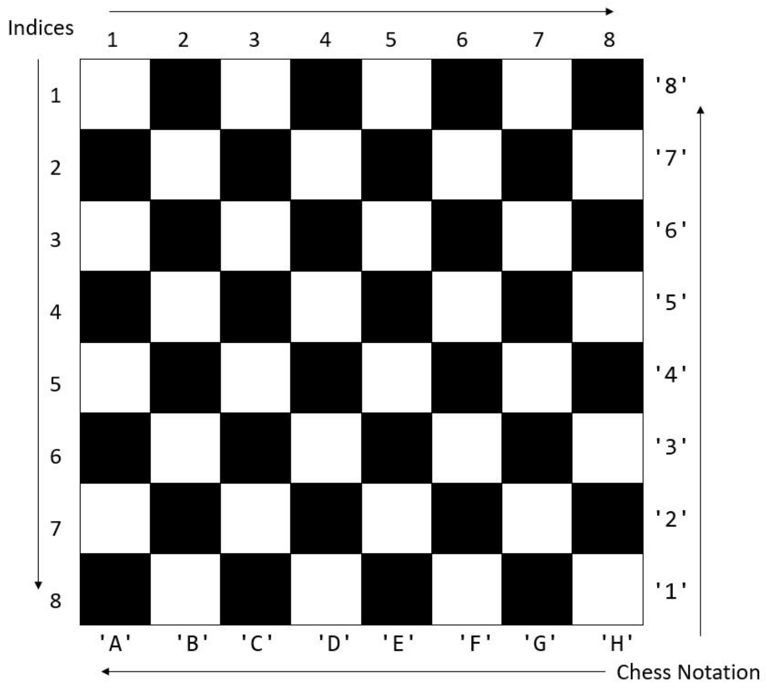
1. *(char)* an 8x8 matrix representing the filled in board

Background:

While walking past the CS 1371 TA Office, you notice a chessboard and get inspired to learn about chess! The first step in learning about chess is understanding where the pieces can be placed. To get a better feel for it, you decide to turn to your trusty friend MATLAB!

Function Description:

Given a character array of variable height where each row is the position of a piece, fill in a chess board. Each row will have the format '<piece><letter><number>' (e.g. 'KE1'). Put the letter representing the piece (the first column) into the position denoted by the second and third column. The positions are given by the graphic below. (e.g. 'E1' => position [8, 5]).



Every position that is not given a character should be given the value of a space.

Example:

```
input = ['KE1';'kE8';'QD1';'qD8'];
board = playChess(input)
board => '   qk   '
         '        '
         '        '
         '        '
         '        '
         '        '
         '        '
         '   QK   '
```

Notes:

- Most of the test cases are randomly generated and thus may not be valid chess positions.

Function Name: `blackJack`

Inputs:

1. (*double*) A 1x2 vector describing your hand
2. (*double*) A 1x2 vector describing the dealer's hand
3. (*double*) A vector with the next cards to be drawn from the deck

Outputs:

1. *(char*) A vector describing the game

Background:

Now that you've sharpened your MATLAB skills, you're ready for the next step of your plan to get rich! This time your blackJack function will be complex enough to play out an entire hand of blackjack.

In blackjack, the goal is to achieve a total score higher than the dealer without going over a score of 21. You will begin with two cards in your hand. You can choose to hit (take another card) or stand (keep the hand you have) until you are satisfied with your hand or until you reach a score of 21 or more. In this version of blackjack, **aces always count as 11**, 2-10 count as their face value, and face cards also count as 10.

Function Description:

You are given a vector describing your hand. Assume that you can see the first card in the input describing the dealer's hand. You will decide to hit or stand based on the following rules:

1. Always hit if you have 11 or fewer points in your hand.
2. Always stand if you have 17 or more points in your hand.
3. If you have 12-16, stand if the dealer's first card is 2-6, hit otherwise.

Every time you take another card, you should add that card to the vector representing your hand, delete the card from the deck, recalculate your score, and decide whether or not to hit again. After you've completed your turn, the dealer will draw cards until they have a score of 17 or higher. Finally, output one of the following character vectors describing the result of the game:

```
'I won!'
'The dealer wins.'
'We both busted.'
```

Example:

```
myHand = [2 6]
dealerHand = [4 8]
deck = [6 4 4 8 7 2]
```

```
out = blackJack(myHand, dealerHand, deck)
```

At the start of the hand, you have 8 points, so you accept a card. Now you have `[2 6 6]`. You have 14 and the dealer is showing a 4, so you stand, hoping that the dealer will bust. The deck is now `[4 4 8 7 2]`.

The dealer has a score of 12 at the start of the hand. The dealer hits and attains a score of 16. The deck is now `[4 8 7 2]`. The dealer hits again, and attains a score of 20.

```
out => 'The dealer wins.'
```

Notes:

- The dealer wins in a tie.
- All cards will be represented in the inputs by their point value (2-11).
- There will always be enough cards in the deck to play out the hand.

Function Name: `forgotMathHw`

Inputs:

1. (*double*) An N x (N + 1) array of doubles

Outputs:

1. *(double*) The input array transformed into reduced row echelon form

Banned Functions:

`rref()`

Background:

Oh no! You forgot to finish your math homework and now you have to take a break from the board games to finish it! Unfortunately chess will have to wait.

Function Description:

This function takes in an N x (N+1) array and transforms it into reduced row echelon form. This means that there is a NxN diagonal array of ones, followed by a column of values on the right side, like so:

```
1.0000         0         0    1.5250
     0    1.0000         0    0.6500
     0         0    1.0000    0.0250
```

Use the following steps to transform an array into reduced row echelon form, using the following array as an example:

```
8     1     6    13
3     5     7     8
4     9     2    12
```

Step 1: For each row *i*:

Step 1.a: Divide the entire row *i* by the first nonzero entry in the row. By dividing row 1 by 8, we obtain the following:

```
1.0000    0.1250    0.7500     1.6250
3.0000    5.0000    7.0000     8.0000
4.0000    9.0000    2.0000    12.0000
```

(*i* = 1)

Step 1.b: For each row *j*, *j* > *i* (all rows below row *i*):

Subtract row *i* multiplied by first nonzero entry in row *j* from row *j*. This is to obtain a zero in the *i*th index of the *j*th row. After subtracting three times the first row from the second row, we obtain:

```
1.0000    0.1250    0.7500     1.6250
     0    4.6250    4.7500     3.1250
4.0000    9.0000    2.0000    12.0000
```

(*i* =1, *j* = 2)

Repeat **Step 1.b** for the third row. After subtracting four times the first row from the third row, we obtain:

```
    1.0000    0.1250    0.7500    1.6250
         0    4.6250    4.7500    3.1250
         0    8.5000   -1.0000    5.5000
```

(i = 1, j = 3)

After completing **Step 1.b** for every row j, there should be all 0's in the ith column below row i.

Repeat **Step 1.a** and **Step 1.b** for the remaining rows.

After dividing the second row by 4.625, we obtain:

```
    1.0000    0.1250    0.7500    1.6250
         0    1.0000    1.0270    0.6757
         0    8.5000   -1.0000    5.5000
```

(i = 2)

After subtracting 8.5 times the second row from the third row, we obtain:

```
    1.0000    0.1250    0.7500    1.6250
         0    1.0000    1.0270    0.6757
         0         0   -9.7297   -0.2432
```

(i = 2, j = 3)

After dividing the last row by -9.7297, we obtain

```
    1.0000    0.1250    0.7500    1.6250
         0    1.0000    1.0270    0.6757
         0         0    1.0000    0.0250
```

(i = 3)

Since this row is the last row, **Step 1.b** is not repeated again.

Step 2: For each row k, starting with the *last* row and decreasing for k > 1:

Step 2.a: For each row m, such that m < k (all rows above row k in the array):
Subtract row k multiplied by the kth index of row m from the mth row.
After subtracting 1.027 times the third row from the second row, we obtain:

```
    1.0000    0.1250    0.7500    1.6250
         0    1.0000         0    0.6500
         0         0    1.0000    0.0250
```

(k = 3, m = 2)

Repeat **Step 2.a** for each row m.
After subtracting 0.75 times the third row from the first row, we obtain:

```
    1.0000    0.1250         0    1.6063
         0    1.0000         0    0.6500
         0         0    1.0000    0.0250
```

(*k* = 3, *m* = 1)

Repeat **Step 2** for the each row *k*, up until row 1.

After subtracting 0.125 times the second row from the first row, we obtain:

```
    1.0000         0         0    1.5250
         0    1.0000         0    0.6500
         0         0    1.0000    0.0250
```

(*k* = 2, *m* = 1)

The process is now complete!

Notes:

- Please round your output array to 5 decimal places.
- It is guaranteed that this procedure will result in an answer in the correct format (reduced row echelon form) for any test case we give you.
- Try to think about this problem using a real matrix, like the one in the example.

Extra Credit

Function Name: `moveSnek`

Inputs:

1. *(double)* An MxN array representing the current game board
2. *(char)* A 1xN character vector representing the sequence of moves

Outputs:

1. *(double* OR *char)* Either the updated game board or an endgame string

Background:

You are bored one Friday evening and decide that the best way to entertain yourself is to play a game of snake! However, you cannot find any good apps so you decide to make your own version using MATLAB!

Function Description:

If you have never played snake before, you control a snake that is moving around a screen and is trying to increase its length by eating cookies that appear on screen. The game ends when you either run into the edge of the board or run into yourself. Here is a link to an online snake game you can use to familiarize yourself with the game.

For our version of the game, the board will be composed of three types of elements: empty space (`0`), the snake (`>0`), and a cookie byte (`-1`), the snake's food. Below is a visualization of a potential game board.

```
board = [0  0  0  0  0  0  0;
         0  1  2  3  4  0  0;
         0  0  0  0  5  0  0;
         0  0  0  0  6  0  0;
         0  0  0  0  7  8  9;
         0  0  0  0  0  0 -1;
         0  0  0  0  0  0  0]
```

The snake will always be represented as consecutive integers starting at 1. The head of the snake is always the largest value on the game board.

The movement of the snake will be described by the 4 cardinal directions (North, South, East, West) represented by the characters `'N'`, `'S'`, `'E'`, or `'W'`.

For this problem, the sequence of moves will be a string of **unknown length** representing how to move the snake. For example, the string `'NNWSSS'` means move the snake North two steps, West one step, then South three steps.

The snake should always "follow itself", meaning that the head will move in the direction specified and the nth segment of the body will move into the place of the (n+1)th segment. In the case that the snake hits a cookie, the length of the snake should increase by one. To do this, the head value should be incremented by one and placed on top of the cookie and the rest of the snake should remain the same. For example, given:

Continued...

```
currBoard = [0, 0, 0, 0;...         dirSequence = 'SE'
             1, 2, 0, 0;...
             0, 3, 0, 0;...
             0,-1, 0, 0]
```

After the first character is read from the sequence, the snake would move South, running over the cookie and producing the following board (assuming a new cookie was spawned at `(1, 3)`):

```
iteration1Board = [0, 0,-1, 0;...
                   1, 2, 0, 0;...
                   0, 3, 0, 0;...
                   0, 4, 0, 0]
```

Then, after the next character is read from the sequence, the snake will move East, producing a final result of:

```
finalBoard = [0, 0,-1, 0;...
              0, 1, 0, 0;...
              0, 2, 0, 0;...
              0, 3, 4, 0]
```

When you go to move the snake, you should use the provided `checkCollision()` function to see if the next move is valid. You should call this function every move to see what will happen. It will return a string that will tell you if you are doing a normal move, if you run into a cookie, or if you will have a game over. Please type `help checkCollision` in your command window to see how this function works.

To generate new cookies on the board, you should use the provided `spawnCookie()` function. This function should be called after each time you move the snake. This function will do one of two things. If there is already a cookie on the board, or if there is no empty space on the board, it will return an unchanged board. If there is not a cookie on the board, it will return a new board with a cookie in a random empty space on the board. You should type `help spawnCookie` for more information on how to use the function.

In the case that the snake hits itself or goes off the edge of the board (array), you should output `'Game Over!'` instead of the final state of the board. Otherwise, you should output the final state of the board after the sequence of directions is executed.

Since this is a lot of information, here is the general idea for what you should do: (1) use checkCollision, (2) if it's not a valid move, then game over, (3) check if it hits a cookie, (4) If it does, extend the head, (5) spawn a cookie, (6) move the snake

Notes:

- DO NOT submit the `spawnCookie()` or the `checkCollision()` functions.
- The board can be of any size, and the snake can be of any length
- You are guaranteed to never fill up the board

Hints:

- Try to come up with your own moves when testing your code
- Think carefully about which type of loop is going to be the most effective
- Think about the different steps you need to take and in what order: Use checkCollision, end the game if the next move is invalid, add a new head if you run into a cookie, or move the snake
- Is there a way to guard your code to not process any more moves if the game has ended using concepts we have already learned?
- Can you make any assumptions about finding the next head of the snake to make your steps easier?
- And then how can math be used to make manipulating the snake easier?