Function Name: `primeFinder`

Inputs:

1. *(double)* an integer greater than zero

Outputs:

1. *(logical)* a logical telling if the input is prime

Banned Functions:

`isprime(), primes(), factor()`

Function Description:

Prime numbers are whole numbers that are divisible by only 1 and themselves. Write a function that takes in a positive integer and determines if it is prime, returning `true` if it is prime and `false` if it is not.

Notes:

- 1 is not a prime number.
- The input is guaranteed to be a positive integer (1,2,3…).

Hints:

- Notice you only need to test for divisibility of the input up to its square root.

Function Name: `secretMessage`

Inputs:

1. *(char)* A string containing a secret message

Outputs:

1. *(char)* The secret message

Function Description:

Lately you've grown concerned about the privacy of your messages, so you've decided to hide secret messages in your texts by placing them inside parentheses.

For example, given a string that says:

```
str = '(I) (lo)ve MATLAB, it''s the coole(st) thing ever(!)'
```

The secret message is:

```
secretMessage = 'Ilost!'
```

The secret message is made by concatenating together all the characters within each set of parentheses in the string. Given a string, produce the secret message.

Notes:

- Parentheses will always appear in pairs, for example (...)...(...) and never (...(...)...).
- There may be any number of characters between an open and close parentheses.
- Every open parenthesis will have a corresponding closing parenthesis.

Function Name: `moveSnake`

Inputs:

1. *(double)* The current game board
2. *(char)* A sequence of moves

Outputs:

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

Function Description:

Now that you know iteration, you can build upon last week's `checkCollision()` problem and actually move your snake around the board! This function will take in the current state of the game as an array. The board will be formatted in exactly the same way as it was for `checkCollision()`.

As a refresher, the snake is an increasing sequence of positive integers with the largest number on the board being the head of the snake. A "cookie" is represented by a -1 and empty space is represented as zeros. The boundaries of the array are the boundaries of the game.

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 n^{th} 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

```
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, 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]
```

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. If at any time the snake fills the entire board, you should output the string `'You Win!'`. In all other cases, you should output the final state of the board after the sequence of directions is executed.

Notes:

- DO NOT submit the `spawnCookie()` function.
- Feel free to use your `checkCollision()` code, just be sure to either copy it as a helper function into your `moveSnake.m` file, or submit your `checkCollision.m` file with this homework.

Function Name: `lostAbroad`

Inputs:

1. *(char)* An MxN character array map
2. *(double)* A 2xP array describing how to move through the map

Outputs:

1. *(char)* A string indicating the outcome of your adventure

Function Description:

While on a study abroad trip, you decide to go on an adventure in the city but get lost! That's when you realize you have no phone, you don't understand the language, and you are terrible at directions! Luckily, you find a character array map of the area (ha). Below is the legend for symbols on the map:

`'#'`: forbidden areas; if you come across one, you are lost forever

`'*'`: meeting point where you'll reunite with your friends

All other characters: roads on the map

The second input describes which direction you should go by providing a sequence of row/column indices to traverse. These indices will be provided as a 2xP array where the first row is the rows of the map and the second row is the columns of the map. For example, the array:

```
[3, 3, 6;...
 4, 7, 7]
```

describes starting at location (3, 4) in the map, then moving to location (3, 7) and finally ending at location (6, 7). The travel direction is always linear (in the same row or column). You will output the characters you "pick up" on your journey, excluding the last `'*'` at your meeting point.

For example, given the following inputs:

```
map = ['TLAB*';...
       'A##CS';...
       'MQW##']
```

```
directions = [3, 1, 1;...
              1, 1, 5]
```

your output would be `'MATLAB'`. You do not repeat the character at the turning locations.

If you come across a forbidden area `'#'` at **any** point during your journey (including your final location), your output should be

```
'No record of this person exists. -Government'
```

If you did not come across a forbidden area, but your final location does not **end** with the meeting point `'*'`, you have recorded the wrong directions, and your output should be

```
'I am lostAbroad and can't find my friends.'
```

Notes:

- The second input is guaranteed to be at least 2x2 (a start and end point).
- The meeting point `'*'` will only appear once in the array, but it must be at the end of the string to be considered a valid escape route.
- Each location is guaranteed to be within the same row or same column as the previous location.

EXTRA CREDIT

Function Name: `whenIsGood`

Inputs:

1. *(double)* An Mx24 array representing a schedule
2. *(char)* An NxP array containing what meetings to be scheduled

Outputs:

1. *(double)* An Mx24 array representing the updated schedule
2. *(char)* A String describing how successful the scheduling operation was

Function Description:

On a scale of 1 to America, how free are you tonight?

In this problem, write a function that attempts to fit meetings into multiple people's schedules. The first input of the function will be an Mx24 array, in which each row represents the schedule of a different person. Each column represents 30 minute intervals between 8am and 8pm. During the times when someone is busy, there will be an `Inf` in the corresponding indices. Conversely, during the times when the participant is free, there will be a `0` in the corresponding indices. See the following array for an example:

```
schedule =

   Inf   Inf   Inf   Inf     0     0     0     0     0   Inf   Inf   Inf
     0     0     0     0     0     0     0   Inf   Inf   Inf   Inf     0
```

In this array, the times 8:00am - 2:00pm are shown (remaining times truncated for illustration purposes). Participant one is preoccupied from 8:00am to 10:00am, and 12:30pm to 2:00pm. Similarly, participant two is preoccupied from 11:30am to 1:30pm. They are free at all the other times in the part of the array shown.

The second input to the function is an NxP char array in which each row represents a separate meeting and will contain 2 pieces of information: the required participants and the meeting duration, formatted as follows:

`'[1 2 3 4], 120'`

The above string would indicate that participants 1, 2, 3, and 4 need to be present at this event and that the event will last for 120 minutes. The participant list will always be in square brackets, and there will always be a comma between the participant list and the duration. There are no guarantees about whitespace in the string.

Your output to the function will be the same array as the first input, EXCEPT that during times when meetings have now been scheduled, the `0` should be changed to the corresponding

meeting number. The meeting number is the row number of that meeting as it appears in the second input.

Clearly, a unique solution may not exist (e.g. if all participants are free the entire day), so you should always schedule longer meetings before the shorter meetings, and always schedule meetings as early in the day as possible. If multiple meetings span the same duration, the one that is listed first (has a lower meeting number) should be scheduled first.

Your second output should be a string that is one of the following:

- If all the meetings can be scheduled, output:
  `'All meetings scheduled!'`
- If any meetings cannot fit into the schedule, output:
  `'X meeting(s) could not be scheduled!'`
  where X is the number of meetings that could not be scheduled. If x is 1, the word meeting should be singular (meeting). If it is greater than 1, it should be plural (meetings).

Notes:

- A test case generator, `schedulegen()`, has been provided to generate additional test cases for this problem. Please type `help schedulegen` in the command window to see its functionality.
- The number of minutes of an event will never exceed 720.
- All values found in all participant lists will be valid indexes in the input array.

Hints:

- This problem involves quite a few steps. Break them down into manageable pieces and organize your thoughts before you start coding!
- `str2num('[3 4 5]')` ⇒ `[3 4 5]`
- See what happens when you use `strfind()` on vectors.