Function Name: `EVAtime`

Inputs:

1. (*char*) A string of names of astronauts or space programs
2. (*double*) A vector of EVA times

Outputs:

1. (*char*) The astronaut or space program with the most EVA time
2. (*double*) The EVA time

Function Description:

In the world of spaceflight, an EVA (extravehicular activity) is any activity done by an astronaut outside of a spacecraft beyond Earth's atmosphere. Every country that has gone into space has kept careful records of how many hours of EVA time each astronaut has had, as well as the EVA time each space program taken.

Given a string of comma-separated names of either astronauts or space programs, and a vector of corresponding hours of EVA time, write a function that outputs the name of the astronaut or space program with the longest EVA time and the number of hours of EVA time they had (rounded to two decimal places).

For example, if the inputs were `'Mahalakshmi,Edward,Adam,Rachel'` and `[8.48 8.36 112.00 8.48]`, the function should output `'Adam'` and `112`.

There will never be a repeated name in the list.

Notes:

- You may NOT use `strfind()`, `find()`, `strsplit()`, or `regexp()` for this problem.
- Round the 2nd output to 2 decimal places.

Function Name: changeCalc

Inputs:

1. *(double)* An amount of money to translate into change

Outputs:

1. *(char)* A formatted string of the coins that sum to the amount

Function Description:

One of the reasons the values of US currency (and most world currencies) are the way they are is so that the minimum number of coins that sum to a particular value can be calculated using a greedy algorithm. You can read more about greedy algorithms here but the basics for calculating change are as follows:

- Select the coin that does not exceed the total
- Add this coin to the solution
- Subtract the value of the coin from the total
- Repeat until the total is zero

Write a function that will calculate the minimum number of coins that sum to a given amount. The amount will always be non-negative valid amount of money ($12.523 is an example of an invalid amount of money). Format your answer as:

```
'Quarters: # | Dimes: # | Nickels: # | Pennies: #'
```

Where each `'#'` is replaced with the respective number of coins.

Notes:

- Make sure you check your string formatting against the solution function using `isequal()`

Function Name: `ugaMath`

Inputs:

1. *(char)* A string representation of a math function
2. *(double)* A value at which to evaluate the function

Outputs:

1. *(double)* The value of the function at the given value

Function Description:

Given a math function and a value, write a MATLAB function that evaluates the math function at that value. Except the function must use the math skills taught at u[sic]ga. That is, the function completely ignores order of operations and parentheses, and it evaluates all operators strictly left-to-right. Inputs are guaranteed to only have `+`, `-`, `*`, `/`, and `^` as operators in the function. The left-hand side of the input function will always follow the form:

```
functionName(varName) =
```

There will always be one and only one operator between each number and/or variable in the input function. There will not be any leading operators. There will also be a single space in between each number and operator. There will be no parentheses or extraneous characters on the right-hand side of the input function. The value that you evaluate the function at will always be greater than or equal to zero.

Notes:

- The variable name can be multiple characters long.
- Round the output to 2 decimal places.

Hints:

- `strrep()` may be a useful function.

Function Name: `layerBricks`

Inputs:

1. *(double)* The desired height of the brick pyramid
2. *(char)* The desired color of the bricks

Outputs:

1. *(char)* A character array displaying the constructed brick pyramid

Function Description:

This function will take in a brick color as a string and a double indicating a pyramid's height, and create a character array of that pyramid. In order to build your pyramid, you must first create a brick, which is done by placing square brackets around the string of the brick color. So, if the second input is `'blue'` then your brick should be `'[blue]'`. There will be no spaces between your bricks. Using these bricks, stack a pyramid of the indicated height, with the top layer containing one brick in the center, and each subsequent row containing one plus the above layer's number of bricks (each row is centered). For example:

```
pyramid = layerBricks(3, 'blue')
pyramid =>           [blue]
                  [blue][blue]
               [blue][blue][blue]
```

In this case, the function creates a single 4x24 character array, where the "non-brick" areas are filled with spaces.. Remember that a string is just a vector of characters!

Notes:

- You are guaranteed the given brick color will have an even number of letters.
- Make sure to compare your function's output to the solution code's output!

Hints:

- Think about how the height of the pyramid relates to how many bricks need to be laid at the base, and thus the overall number of array columns needed.

*See what happens if you enter the name of a famous brick stacking game from the 80's as the color!

Function Name: `cutTheRope`

Inputs:

1. *(double)* A 1xN vector of integers representing rope turns

Outputs:

1. *(double)* The location that cuts the rope into the most pieces
2. *(double)* The maximum number of pieces

Function Description

If you loved the game Cut the Rope, then try to contain your excitement; apart from the name, this problem has nothing to do with that game.

Imagine you have a rope that zig-zags along a number line with the locations of the corners defined by the integers in the input to the function. You must calculate the location along the number line where you can make a vertical cut to break the rope into the maximum number of pieces. The rope will always turn at integer values (there would never be a turn at 3.45, for example). Additionally, you are only allowed to cut the rope halfway between integers (at 1.5, 2.5, -4.5, etc).

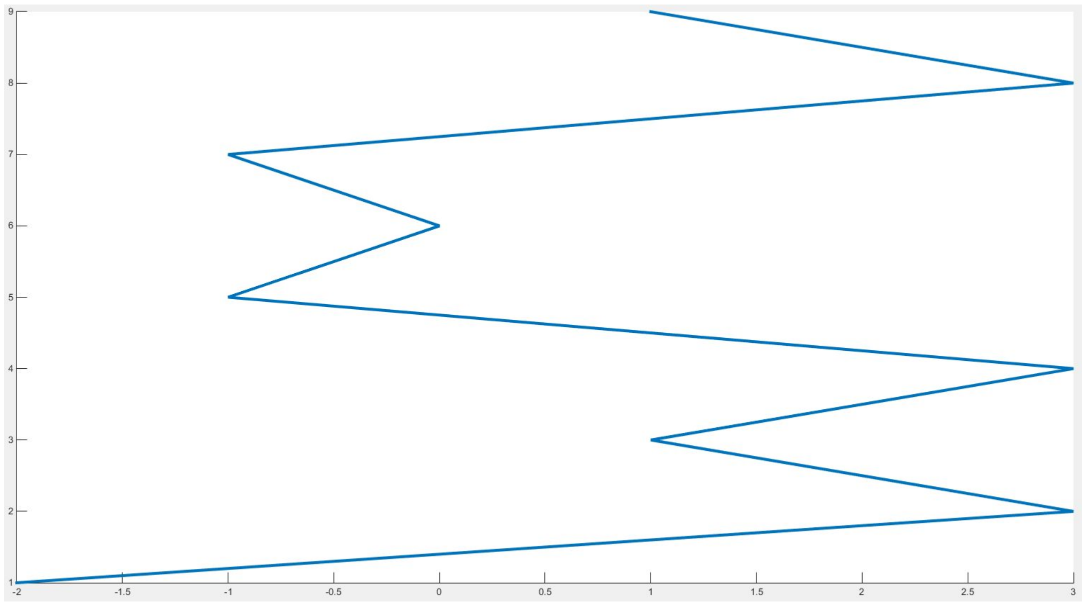
It may be the case that any cut on a particular range (or multiple disjoint ranges) will produce the same number of pieces. In this case, you should output the **smallest** cut location that will produce the maximum number of pieces. For example, if cutting the rope anywhere between -2 and 5 or between 7 and 8 would produce 10 pieces, and this is the maximum number of pieces, then you should output -1.5 as the cut location because this is the smallest number on these ranges that lies halfway between 2 integers.

(example on next page)

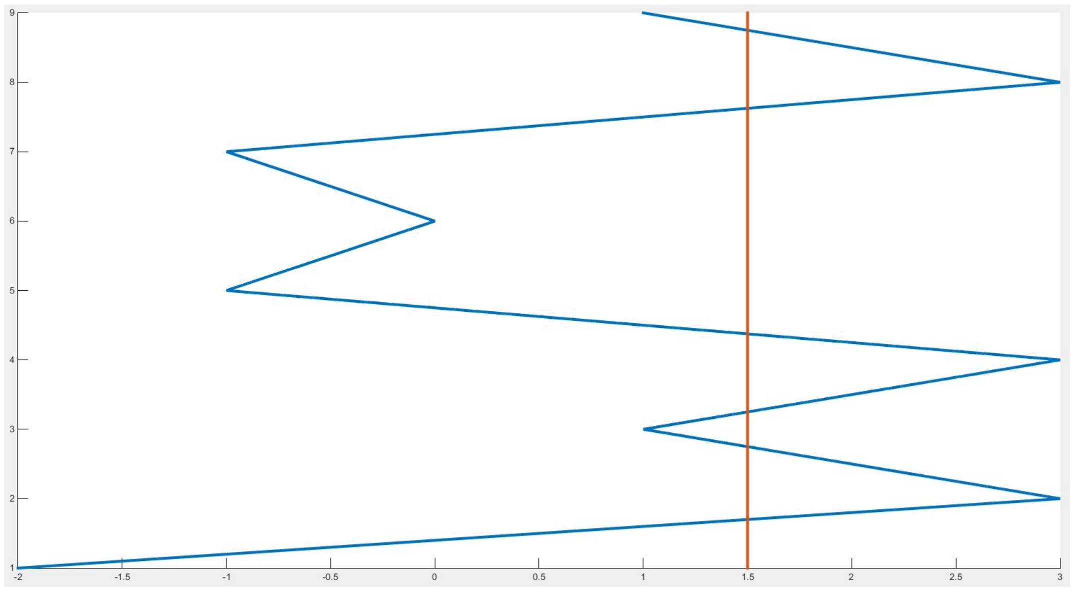
Consider the following example:

An input of: `[-2, 3, 1, 3, -1, 0, -1, 3, 1]`

would represent a rope that zig-zags in the following way:



Any cut on the range (1, 3) will produce the maximum number of pieces (7), but because we must cut the rope at the smallest location on this range that is halfway between integers, the cut location would be 1.5. Therefore the first output for this case (cut location) would be 1.5 and the second output (maximum number of pieces) would be 7, as visualized by the next plot:



To help you visualize any test case, we have provided a function called `cutTheRope_plot()` that will produce plots like the one seen here given any vector of turn locations.

Notes:

- The values of the input vector are guaranteed to be integers and will change directions on each successive element. For example, the vector `[3, 5, 8]` is not a valid input.
- The input vector will have length greater than or equal to 2.
- If you are interested in the general case of this problem (where the rope is not restricted to turning at integer locations and you can cut anywhere), check out range trees.