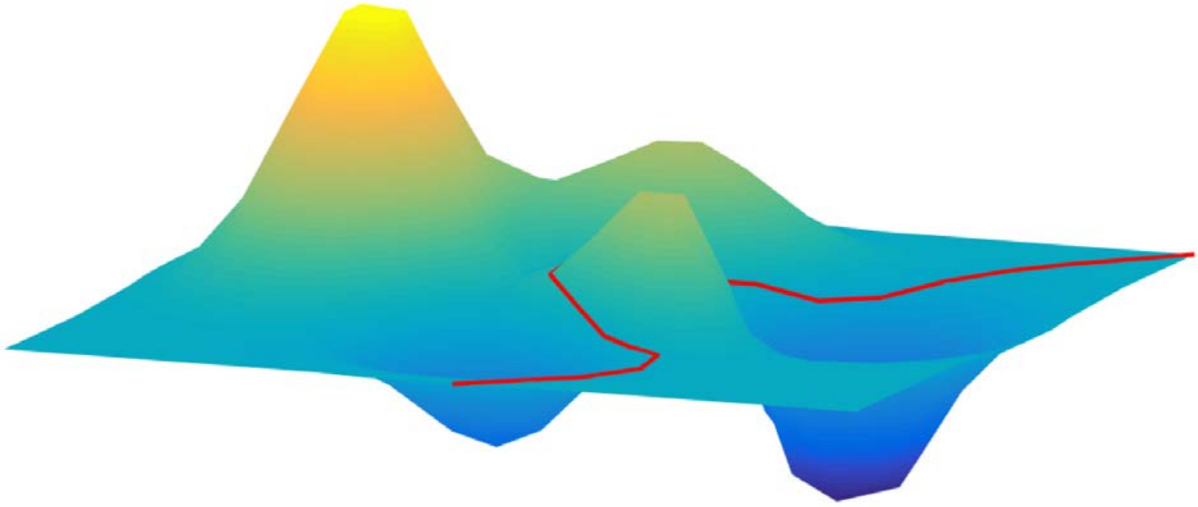
Greedy walks in pursuit of laziness

Project due date: You will need to upload your code before

11.59pm on Sunday the 17th of September.



Introduction

This project requires you to write code that will allow you to find the best path from one side of a map to the other, where the map is represented as a 2D array containing elevation data. For our purposes the best path is deemed to be the one where a person moving from the west side of the map to the east (or vice versa) would experience the smallest total change in elevation as they move along that path (where any increase or decrease in elevation is interpreted as a positive change).

It is expected that you will create your own elevation arrays to test your code (make sure you work out the answers by hand so you know what you should get). Also before the submission date you will be supplied with some test data, test scripts and some functions which can be used to visualise your results (so that you can produce images for your path like the one shown above). Do NOT rely solely on the supplied test scripts, part of doing the project is to extensively test your own code on a range of different values.

This project is designed to take the average student around **15 hours** to code up (excluding the final challenge task) but some people may find it takes upwards of 60 hours (and others will finish it in just a few hours). You will not know ahead of time how long it will take you, so please don't leave it to the last minute to make a start!

How to tackle the project

Do not be daunted by the length of this document. It is long because a lot of explanation is given as to exactly what each function needs to do.

The best way to tackle a big programming task is to break it down into small manageable chunks, which has been done for you in the form of eight functions to write. Each function has a detailed description of what it needs to do. Have a read through the entire document and then pick a function to start on. Remember you don't have to start with `Reverse`, although it is a fairly straight forward function to write, so it might be a good warm up. `FindSmallestElevationChange` and `FindPathElevationsAndCost` are other good ones to start with. Write as many of the functions as you can.

Several functions require careful thought (remember those 5 steps of problem solving!). If you are having trouble understanding how a function should work, remember to work through the problem by hand.

Note that I don't expect most people to get through all eight functions. Some of them are relatively easy (e.g. `Reverse`, `FindSmallestElevationChange` and `FindPathElevationsAndCost`) while others are a bit tricky (e.g. `BestGreedyPath`). Note that `BestPath` is designed to challenge even the most talented students. You can still get a pretty good mark even if you don't complete all e functions.

An "A" grade student should be able to nut out everything except perhaps `BestPath` (which is **extremely** challenging). B and C grade students might not get a fully working solution. Even if you only get half the functions working, you can still get over 50% for the project, as long as the code you submit is well written (since you get marks for using good style).

An overview of finding the best greedy path.

Before we delve into some more detail, let's take a quick high level outline of how a greedy walk works.

A greedy algorithm is one where we try and find the best solution to a larger, global problem by making a sequence of best choice for smaller, local problems. Greedy algorithms do not always find the best possible solution but they can often find a pretty good solution in a reasonable number of steps. Finding the very best solution to a problem may require an unreasonable number of steps (due to searching through a large number of possible outcomes), or a more sophisticated approach.

For more on greedy algorithms see: https://en.wikipedia.org/wiki/Greedy_algorithm

Our larger global problem is finding the best path (i.e. the one with the lowest total elevation change) as we move from one side of our map to the other. We can attempt to do this by moving across the map a column at a time, choosing the best position to move to from the small number of adjacent elements in our 2D array that are in our direction of motion (we choose the one with the smallest elevation change).

Here is an example of a greedy walk starting from row 3, column 1 of our array (on the western side of the map) and heading towards the eastern side of the map. Note that at each step along the path we must move to an adjacent easterly array element (where an element is adjacent if it shares an edge or corner).

North

West

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

East

South

Let's see how this path was found, assuming the above information is stored in an array called E.

Step 1	We are at array element E(3,1) which contains the value 7. To the east there are three adjacent elements to choose from. Calculate the elevation changes for each possible choice.	elev. chge 4 3 7 9 2 8 1	We choose to move to the element containing 8, as this results in the minimum elevation change of 1. Hence our path now includes E(4,2)
Step 2	We are at array element E(4,2) To the east there are only two adjacent elements to choose from since we are at the bottom of the map.	elev. chge 5 3 8 4 4	We choose to move to the element containing 5, as this results in the minimum change of 3. Hence our path now includes E(3,3)
Step 3	We are at array element E(3,3) To the east there are three adjacent elements to choose from. Calculate the elevation changes for each possible choice.	elev. chge 4 1 5 6 1 3 2	We have a tie for the minimum change of 1. In the instance of a tie we will agree to always choose the northern most element. Hence our path now includes the element containing the value 4, E(2,4)
Step 4	We are at array element E(2,4) To the east there are three adjacent elements to choose from. Calculate the elevation changes for each possible choice.	elev. chge 2 2 4 1 3 9 5	We choose to move to the element containing 2, as this results in the minimum change of 2. Hence our path now includes E(1,5)
Step 5	We are at array element E(1,5) To the east there are only two adjacent elements to choose from since we are at the top of the map.	elev. chge 2 5 3 4 2	We choose to move to the element containing 4, as this results in the minimum change of 2. Hence our path now includes E(2,6)

We have determined a path from the western side to the eastern side but is it the best one?

To figure this out, we need to calculate the cost of our path and compare it against other possible paths.

The cost of the path will just be the sum of the elevation changes, for our path we have a cost of

|8-7|+|5-8|+|4-5|+|2-4|+|4-2| = 1+3+1+2+2 = 9

Note that we are taking the absolute value of the differences to ensure we always treat an elevation change as a positive value. We can compare this total cost against the total costs of paths starting at other points on the western edge (i.e. other points starting in column 1. Here are the other three paths, with their costs

3	6	**3**	7	**2**	5
1	**4**	2	**4**	1	**4**
7	9	5	6	9	2
10	8	4	3	10	5

Starting at E(1,1) heading east
Cost = 1+1+1+2+2 = 7

3	6	**3**	7	**2**	5
1	**4**	2	**4**	1	**4**
7	9	5	6	9	2
10	8	4	3	10	5

Starting at E(2,1) heading east
Cost = 3+1+1+2+2 = 9

3	6	3	7	**2**	5
1	4	2	**4**	1	**4**
7	**9**	**5**	6	9	2
10	8	4	3	10	5

Starting at E(4,1) heading east
Cost = 1+4+1+2+2 = 10

Hence the best path from heading east will be the one starting at position E(1,1) with a cost of 7.

Of course we could just as easily have started from the eastern edge and headed west. It may be that one of these paths would be better, so let us check them.

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

Starting at E(1,6) heading west
Cost = 3+2+1+1+1 = 8

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

Starting at E(2,6) heading west
Cost = 2+2+1+1+1 = 7

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

Starting at E(3,6) heading west
Cost = 1+3+1+1+1 = 7

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

Starting at E(4,6) heading west
Cost = 4+3+1+1+1=10

So far it looks like the best greedy walk possible after considering starting on the western edge or the eastern edge will be one involving a total cost of 7.

Are there any other greedy paths we could consider? YES!! We don't have to start on a western or eastern edge to create a greedy path. For example we can start at any element in the array, heading west on one side and east on the other. If we started at element E(4,4) and headed in both directions we would get the path below.

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

Element E(4,4) contains the value 3. We can head east from this point which results in adding E(3,5) and E(4,6) to the path. We can also head west which adds E(4,3), E(4,2) and E(3,1) to the path.
Next calculate the cost (which can be done easily by retracing the path just found, starting at E(3,1) with the value 7)
Cost = 1+4+1+6+4 = 16

Starting at E(4,4) certainly didn't find us a lower cost path (in fact it is the worst we have seen so far!) Note however to be sure we have found the very best greedy path we shouldn't just use the elements on the western edge or the eastern edge, we should find the greedy paths originating from EVERY element in the array otherwise we could easily miss out on finding the best one.

Consider the following array

1	2	3	4	3	2	1
1	2	3	4	3	2	1
1	2	4	5	4	2	1
1	3	3	3	3	3	1

For the above example checking just the paths from the western edge and the eastern edge will not result in the best solution (see if you can figure out why).

If we check the greedy paths that originate from EVERY single array element, we can be sure we have found the best greedy path, given our decision to always choose the northern most element in a tie, however this isn't always guaranteed to be the best possible path for the map. By always choosing the northern most element in a tie we are of course excluding some possible paths. Also greedy algorithms have an inherent limitation. Because they only deal with local information and don't look ahead, they can't take advantage of information just beyond what is locally available. Because of that it is possible to create maps where a greedy walk will not find the best path, even if we start one from every single array element. Some more thought is required if we want to be sure we have the best possible path.

Overview of Functions to Write

There are eight in total:

- Reverse
- FindSmallestElevationChange
- GreedyPick
- GreedyWalk
- FindPathElevationsAndCost
- BestGreedyPathHeadingEast
- BestGreedyPath
- BestPath

The pages following have detailed specifications of what each function should do.

The Reverse function

`Reverse` simply returns a reversed version of the 1D array passed to it (i.e. the output is the array passed in with the order of the elements reversed). It serves as a good warm up and you may find it helpful to call your `Reverse` function in later tasks, in the event that you need to reverse an array

Note the capital R in the name. Case matters in Matlab and you should take care that your function names EXACTLY match those in the projection specification.

`Reverse` takes **a single** input, a 1D array to reverse.
It returns a single output, a 1D array with the elements in reverse order to those of the array that was passed in as an input.

Worked example

Suppose our array contains the values [5 3 2 4 1]

The reversed array will contain the values [1 4 2 3 5]

Example calls

Here are some examples of calls to `Reverse`

```
>> r = Reverse([1 2 3])

r =

     3     2     1

>> r = Reverse([5 3 2 4 1])

r =

     1     4     2     3     5
```

The FindSmallestElevationChange function

FindSmallestElevationChange finds the position in an array of elevations of the element or elements that correspond to the smallest change in elevation, given the current elevation.

It takes **two** inputs in the following order:

1) A number representing the current elevation
2) A 1D array of new elevations to choose from

It returns a single output, a 1D array containing a list of indices that identify the array element(s) that correspond to the smallest change in elevation (the change will be measured as a positive number regardless of whether the elevation change is an increase or decrease).

The output array will have at least one element but may have multiple elements if there are multiple elements in the new elevations that produce a tie for the smallest elevation

Worked example

Suppose our current elevation is 7 and our new possible elevations are [4 9 8].

The change in elevation for each of these elements is 3, 2 and 1 respectively. Hence we should choose element 3 of our new possible elevations.

Example calls

Here are some examples of calls to FindSmallestElevationChange

```
>> smallestPosition = FindSmallestElevationChange(7,[4,9,8])

smallestPosition =

     3

>> smallestPosition = FindSmallestElevationChange(7,[3,7,8])

smallestPostion =

     2

>> smallestPosition = FindSmallestElevationChange(3,[2,7,4,5])

smallestPosition =

     1     3

>> smallestPosition = FindSmallestElevationChange(5,[10,4])

smallest =

     2
```

The GreedyPick function

The `GreedyPick` function chooses which array element to go to next, based on which adjacent element result in the minimum change in elevation. In the event of a tie it will pick the northern-most element (i.e. the one with the lowest row number). You will likely find it helpful to call your `FindSmallestElevationChange` function when writing `GreedyPick`.

Note that an adjacent element is defined as one to the east (or west depending on which direction you are moving) which shares an edge or corner with the current element.

In the example below, the adjacent elements to the east of the central value of E(3,3)= 5 are those containing the values 7, 4 and 6 (in green). The adjacent elements to the west of the central value of 5 are those containing the values 8, 9, 5 (highlighted in blue).

3	6	3	7	2
3	8	3	7	2
1	9	5	4	1
7	5	1	6	9
10	8	4	3	10

The `GreedyPick` element takes **three** inputs in the following order:

1) a 2 element 1D array representing the current position, where the first element is the row number of the current position and the second element is the column number
2) an integer representing the direction to head in, with a value of +1 for heading east and -1 for heading west
3) the elevation data stored in a 2D m by n matrix (where m is the number of rows and n the number of columns).

It returns a single output, a 2 element 1D array representing the new position that has been picked, where the first element is the row number of the new position and the second element is the column number.

Worked examples

For the example above, if we were at array element E(3,3)=5 and heading in the easterly direction (represented by positive one) our choice would be the element containing 4, i.e. E(3,4), as this results in the minimum elevation change of 1. This means we would return an array with the values [3,4] representing the position in the third row, fourth column. Note that the element in position E(4,4) also produced an elevation change of 1 but in the event of a tie `GreedyPick` should always pick the northern-most element.

If we were at array element E(3,3) and heading in the westerly direction (represented by negative one) our choice would be the element containing 5, i.e. E(4,2) as this results in the minimum elevation change of 0, so we would return an array with the values [4,2].

Note that when the current position is in the top or bottom row `GreedyPick` should take care not to pick values outside the boundaries of the array.

Example calls

Here is an example of some calls to `GreedyPick` assuming that E is the array introduced on page 3:

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

```
>> E=[3 6 3 7 2 5;1 4 2 4 1 4; 7 9 5 6 9 2; 10 8 4 3 10 5]

>> pick = GreedyPick([3,1],1,E)  % pick should be equal to  [4,2]

>> pick = GreedyPick([4,2],1,E)  % pick should be equal to  [3,3]

>> pick = GreedyPick([3,3],1,E)  % pick should be equal to  [2,4]

>> pick = GreedyPick([2,4],1,E)  % pick should be equal to  [1,5]

>> pick = GreedyPick([1,5],1,E)  % pick should be equal to  [2,6]
```

Here are some more calls but this time heading in a westerly direction

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

```
>> pick = GreedyPick([3,6],-1,E)  % pick should be equal to  [2,5]

>> pick = GreedyPick([2,5],-1,E)  % pick should be equal to  [2,4]

>> pick = GreedyPick([2,4],-1,E)  % pick should be equal to  [1,3]

>> pick = GreedyPick([1,3],-1,E)  % pick should be equal to  [2,2]

>> pick = GreedyPick([2,2],-1,E)  % pick should be equal to  [1,1]
```

The GreedyWalk function

`GreedyWalk` finds a greedy path from a specified start position heading in a specified direction (either heading west or east) and continue walking until the eastern or western edge of the array is reached. At each step along the path the next element in the path is picked using a greedy choice (i.e. we choose the adjacent element that results in the minimum elevation change, with the northern-most element chosen in an event of a tie). You will likely find it helpful to call your `GreedyPick` function when writing `GreedyWalk`.

`GreedyWalk` takes three inputs in the following order

1) a 2 element 1D array representing the starting position for the walk, with the first element representing the row number and the second the column number
2) an integer representing the direction to head in, with a value of +1 for heading east and -1 for heading west
3) the elevation data stored in a 2D m x n matrix

It returns two outputs in the following order

1) a 1D array representing containing the row indices of the path
2) a 1D array representing the corresponding column indices of the path

Worked example

Consider the example path from page 3

North

West

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

East

South

				E(1,5)	
			E(2,4)		E(2,6)
E(3,1)		E(3,3)			
	E(4,2)				

Here the walk starts with the element containing the value 7 in position E(3,1)

The path then proceeds with the elements E(4,2), E(3,3), E(2,4), E(1,5), E(2,6)

We can represent this path using an array containing row indices for the path another array containing the column indices for the path:

Row Indices = [3 4 3 2 1 2]

Column indices = [1 2 3 4 5 6]

Example calls

Here is an example of a call to `GreedyWalk` assuming that E is the array introduced on page 3:

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

```
>> E=[3 6 3 7 2 5;1 4 2 4 1 4; 7 9 5 6 9 2; 10 8 4 3 10 5]

>> [pathRows,pathCols] = GreedyWalk([3,1],1,E)

pathRows =

     3     4     3     2     1     2

pathCols =

     1     2     3     4     5     6
```

Here is another call but this time heading in a westerly direction (note particularly the order of the values in the row path and the corresponding column path, the column indices are decreasing as we are moving towards the west).

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

```
>> [pathRows,pathCols] = GreedyWalk([3,6],-1,E)

pathRows =

     3     2     2     1     2     1

pathCols =

     6     5     4     3     2     1
```

The FindPathElevationsAndCost function

`FindPathElevationsAndCost` takes a specified path and finds the elevations encountered while walking along the path and also calculates the total cost of traversing that path. The total cost is found by summing all the elevation changes where an elevation change is treated as a positive value regardless of whether the elevation is a decrease or increase.

`FindPathElevationsAndCost` takes **three** inputs in the following order:

1) a 1D array representing containing the row indices of the path
2) a 1D array representing the corresponding column indices of the path
3) the elevation data stored in a 2D m x n matrix

It returns two outputs in the following order:

1) a 1D array containing the elevations for the corresponding row and column indices for the path
2) the total cost of traversing the path

Worked example

Recall the example introduced on page 3:

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

The highlighted elements show the path starting at E(3,1) and heading east.
The row indices for the path are [3 4 3 2 1 2]
The column indices for the path [1 2 3 4 5 6]
The corresponding elevation values are [7 8 5 4 2 4]
The total cost would be |8-7|+|5-8|+|4-5|+|2-4|+|4-2|
Which is equal to 1+3+1+2+2=9

If we were heading in a westerly direction starting at E(3,6) we would have the path

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

The row indices for the path are [3 2 2 1 2 1]
The column indices for the path [6 5 4 3 2 1] (note the order!)
The corresponding elevation values are [2 1 4 3 4 3]
The total cost would be |1-2|+|4-1|+|3-4|+|4-3|+|3-4|
Which is equal to 1+3+1+1+1=7

Example calls

Here is an example of a call to `FindPathElevationsAndCost` assuming that E is the array introduced on page 3:

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

```
>> E=[3 6 3 7 2 5;1 4 2 4 1 4; 7 9 5 6 9 2; 10 8 4 3 10 5]

>> [elev,cost]=FindPathElevationsAndCost([3 4 3 2 1 2], [1 2 3 4 5 6],E)

elev =

     7     8     5     4     2     4

cost =

     9
```

Here is another call but this time heading in a westerly direction (note particularly the order of the columns, the column indices are decreasing as we are moving towards the west).

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

```
>> [elev,cost]=FindPathElevationsAndCost([3 2 2 1 2 1], [6 5 4 3 2 1],E)

elev =

     2     1     4     3     4     3

cost =

     7
```

The BestGreedyPathHeadingEast function

`BestGreedyPathHeadingEast` will find the best greedy path starting from the westerly edge (i.e. using starting points in column 1). In the event of one or more paths tying for the lowest cost, the path that starts with the **lowest** row number will be returned. You will likely find it useful to call your `FindPathElevationsAndCost and GreedyWalk` functions when writing `BestGreedyPathHeadingEast.`

`BestGreedyPathHeadingEast` takes **a single** input, the elevation data stored in a 2D m x n matrix

It returns **three** outputs in the following order:

1) a 1D array representing containing the row indices of the path
2) a 1D array representing the corresponding column indices of the path
3) a 1D array containing the elevations for the corresponding row and column indices for the path

Worked example

With the elevation data below from page 3 we have four possible paths from the western edge, heading east:

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

Starting at E(1,1), cost 7

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

Starting at E(2,1) ,cost 9

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

Starting at E(3,1), cost 9

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

Starting at E(4,1), cost 10

Example call

Here is an example of a call to `BestGreedyPathHeadingEast` assuming that E is the array introduced on page 3, where the best path heading east is

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

```
>> E=[3 6 3 7 2 5;1 4 2 4 1 4; 7 9 5 6 9 2; 10 8 4 3 10 5]

>> [pathRow,pathCol,pathElev]= BestGreedyPathHeadingEast(E)

pathRow =

     1     2     1     2     1     2


pathCol =

     1     2     3     4     5     6


pathElev =

     3     4     3     4     2     4
```

The BestGreedyPath function

`BestGreedyPath` will find the best greedy path by calculating greedy paths from **every** single element in the elevation array, iterating through the elements as you would read a book (starting in row 1, col 1, moving across the columns and the proceeding to the second row first column once the entire row 1 has been checked).

For points not on a western or eastern edge you will need to find a greedy path to the west starting at that point and a greedy path to the east starting at that point. This path data can then be used to assemble a greedy path that goes through that particular point.

In the event of one or more paths tying for the lowest cost, the first path that was discovered to have that cost will be returned.

`BestGreedyPath` takes **a single** input, the elevation data stored in a 2D m x n matrix

It returns **three** outputs in the following order:

1) a 1D array representing containing the row indices of the path
2) a 1D array representing the corresponding column indices of the path
3) a 1D array containing the elevations for the corresponding row and column indices for the path

Worked example

With the elevation data below from page 3 we have multiple paths having the lowest cost of 7. The very first path found if we start searching in row 1, col 1 will be that shown below.

3	6	3	7	2	5
1	4	2	4	1	4
7	9	5	6	9	2
10	8	4	3	10	5

Example call

Here is an example of a call to `BestGreedyPath` assuming that E is the array introduced on page 3,

```
>> E=[3 6 3 7 2 5;1 4 2 4 1 4; 7 9 5 6 9 2; 10 8 4 3 10 5]

>> [pathRow,pathCol,pathElev] = BestGreedyPath(E)

pathRow =

     1     2     1     2     1     2


pathCol =

     1     2     3     4     5     6


pathElev =

     3     4     3     4     2     4
```

The BestPath function (extremely challenging!)

BestPath will find the **best** possible path across the map. For our purposes the best path is deemed to be the one where a person moving from the west side of the map to the east (or vice versa) would experience the smallest total change in elevation as they move along that path (where any increase or decrease in elevation is interpreted as a positive change).

How you implement BestPath is completely up to you. You can come up with your own method but also feel free to research useful algorithms. If you find an existing algorithm that you decide to implement in Matlab be sure to credit the source of the algorithm in your comments.

BestPath takes **a single** input, the elevation data stored in a 2D m x n matrix

It returns **three** outputs in the following order:

1) a 1D array representing containing the row indices of the best path
2) a 1D array representing the corresponding column indices of the best path
3) a 1D array containing the elevations for the corresponding row and column indices for the path

If there is a tie for the path with the lowest cost, you mary return **any** valid lowest cost path.

Example call

BestPath will be called in a similar way to BestGreedyPath, as follows

```
>> [pathRow,pathCol,pathElev] = BestPath(E)
```

How the project is marked

A mark schedule and some test scripts will be published prior to the due date, outlining exactly how marks will be allocated for each part of the project and giving you the opportunity to test your code. You will receive marks both for correctness (does your code work?), style (is it well written?) and execution time (how fast does it run?).

Each of the **eight** required function will be marked independently for correctness, so even if you can't get everything to work, please do submit the functions you have written. Some functions may be harder to write than others, so if you are having difficulty writing a function you might like to try working on a different function for a while and then come back to the harder one later.

Note it is still possible to get marks for good style, even if your code does not work at all! Style includes elements such as using sensible variable names, having header comments, commenting the rest of your code, avoiding code repetition and using correct indentation.

Each function can be written in less than 20 lines of code (not including comments) and some functions can be written using just a few lines but do not stress if your code is longer. It is perfectly fine if your project solution runs to several hundred lines of code, as long as your code works and uses good programming style.

How the project is submitted

Submission is done by uploading your code to the Aropa website. More information will be provided on how to submit the project to Aropa in a Canvas email announcement. If you want to be out of Auckland over the break and still want to work on the project that will be perfectly fine as long as you have access to a computer with Matlab installed and can access the internet to upload your final submission.

Submission Checklist

Here is a list of the **eight** functions specified in this document. Remember to include all eight (or as many of them as you managed to write) in your project submission. The filenames should be EXACTLY the same as shown in this list (including case). They should also work EXACTLY as described in this document (**pay close attention to the prescribed inputs and outputs**). In addition if your functions call any other "helper" functions that you may have written, remember to include them too.

- `Reverse`
- `FindSmallestElevationChange`
- `GreedyPick`
- `GreedyWalk`
- `FindPathElevationsAndCost`
- `BestGreedyPathHeadingEast`
- `BestGreedyPath`
- `BestPath`

Any questions?

If you have any questions regarding the project please first check through this document. If it does not answer your question then feel free to ask the question on the class Piazza forum. Remember that you should **NOT** be publicly posting any of your project code on Piazza, so if your question requires that you include some of your code, make sure that it is posted as a **PRIVATE** piazza query.

Have fun!