CS1132 Spring 2015 Assignment 1b Due March 4th

> *Adhere to the Code of Academic Integrity.* You may discuss background issues and general strategies with others and seek help from course staff, but the implementations that you submit must be your own. In particular, you may discuss general ideas with others but you may not work out the detailed solutions with others. It is never OK for you to see or hear another student's code and it is never OK to copy code from published/Internet sources. If you feel that you cannot complete the assignment on you own, seek help from the course staff.

When submitting your assignment, follow the instructions summarized in Section 4 of this document.

Do not use the `break` or `continue` statement in any homework or test in CS1132.

1 Sorting and Searching Fun!

Sorting a collection of objects has become indispensable for today's computer programs. From arranging a list of products by price on Amazon.com to ordering flight itineraries by the duration of the flight, sorting is everywhere. One main benefit of sorting is the increased ease with *searching* given a sorted collection.
In this exercise you will write the code to sort a 1-dimensional array in *non-descending* order according to the given algorithm. You will also write code for searching in a 1-dimensional array and then experimentally evaluate the performance of the sorting and searching algorithms as the size of the array changes.

Do not use built-in functions `sort`, `find`, `max`, `or min`.

1.1 Insertion Sort

Insertion sort is a simple sorting algorithm; a comparison sort in which the sorted array is "built" one entry at a time. In every step of insertion sort, an element is taken from the unsorted segment of the array and is "inserted" into the correct position in the already-sorted segment of the array. This is done until no elements remain in the unsorted segment.

Given the description above, you can see that insertion sort begins by dividing the array into a sorted segment and an unsorted segment. What is the trivial sorted segment? A subarray of length 1! So we begin by treating the first element as the sorted segment and the remaining elements as the unsorted segment. In each iteration, the first element in the *un*sorted segment is inserted into the sorted segment. The result is that after k iterations the first $k+1$ entries of the array are sorted. An example of insertion sort with a length 5 array is shown below. `A` represents the unsorted segment of the array and `B` represents the sorted segment of the array. The 'X' says that there is no element in that position in the sorted or unsorted segment.

Original vector:		8	6	0	9	2
Starting point:	A	X	6	0	9	2
	B	8	X	X	X	X
After 1 iteration:	A	X	X	0	9	2
	B	6	8	X	X	X
After 2 iterations:	A	X	X	X	9	2
	B	0	6	8	X	X
After 3 iterations:	A	X	X	X	X	2
	B	0	6	8	9	X
After 4 iterations:	A	X	X	X	X	X
	B	0	2	6	8	9

How is the insertion in each iteration done? By comparing *adjacent* values and swapping as necessary. For example, in iteration 4 above, 2 needs to be inserted into the sorted subvector [0 6 8 9]. First compare 2 with 9, then swap, so the intermediate result is [0 6 8 2 9]. Then compare 2 with 8 and swap, obtaining the intermediate result [0 6 2 8 9]. Then compare 2 with 6 and swap again to get the intermediate result [0 2 6 8 9]. Finally compare 2 with 0, which shows that 2 is already at the right place so no swapping is done and therefore the 4th iteration ends. In the 4th iteration for the example above, 4 comparisons with adjacent values were done. The number of comparisons with adjacent values done in iterations 1, 2, and 3 for the example above are 1, 2, and 1, respectively.

The sorting is done *in-place*, i.e., use only the original array memory space (the parameter)—do not create additional arrays in your code.

Implement the insertion sort algorithm as described above in the following function:

```
function A = insertionSort(A)
% Sort vector A in non-descending order using Insertion Sort algorithm
% Pre:   A is the vector of numbers to be sorted, the length of A is > 0
% Post:  A is the sorted array
```

2 Random Walk

A random walk is a trajectory that consists of taking successive random steps. For example, it can be the trajectory of a particle inside liquid or gas, or the fluctuating stock price[1]. In this assignment, you will write the code to simulate a random walk on a 2D grid (represented by a 2D matrix in MATLAB). You will do this by first implementing a function to simulate a single step, then a function to simulate the random walk from a starting point until reaching the given goal. Finally, you will write a fragment of code to experimentally evaluate the statistics of the random walk as the size of the grid increases, and plot the results.

2.1 Single Random Step

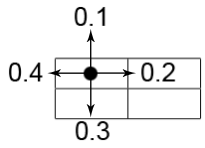
A random walk step is defined by a set of probabilities of different directions of movement. In a 2D grid, the possible movements are moving upward, rightward, downward, and leftward. (In this assignment, we define "up" to be the direction of rows with smaller indices, and "left" to be the direction of columns with smaller indices. Obviously, "down" and "right" are the opposite directions of "up" and "left") The probabilities of the same direction at different places of the grid can be different; this means we need to use a 2D matrix to store the possibilities of that direction at all places on the grid. We have four possible directions, so we need

[1] http://en.wikipedia.org/wiki/Random_walk

four 2D matrices to store the possibilities: `U, R, D, L` (stand for "up", "right", "down", and "left"). For example, consider the following inputs for a 2-by-2 grid:

0.1	0.4	0.2	0.3	0.3	0.15	0.4	0.15
0.25	0.35	0.1	0.35	0.4	0.1	0.25	0.2
U		R		D		L	

then the probabilities of the random steps at `(1,1)` are



Notice the probabilities at one grid location sum to one. In this assignment, you can assume that the corresponding elements of `U, R, D, L` always sum to one. If the randomly chosen direction corresponds to a movement to the outside of the grid (e.g., moving upward or leftward at `(1,1)`), then the movement is ignored (i.e., we stay at the same place for this step).

Implement the following function to do the random step described above:

```
function new_pos = randomWalk(U, R, D, L, pos)
% Do a single step of random walk from the given probabilities and
% position. Notice that the sum of corresponding elements of U,R,D,L is
% always equal to 1. When pos is at the boundary of the grid, ignore the
% movement that would go outside of the grid (i.e., stay in the same position
% if the randomly selected movement exceeds the boundary)
% U: a 2D matrix containing the probabilities of going upward (row index
%    minus one) at each position in the matrix. The size of this matrix
%    defines the size of the grid.
% R: similar to U: the probabilities of going rightward (col index plus
%    one)
% D: similar to U: the probabilities of going downward (row index plus one)
% L: similar to U: the probabilities of going leftward (col index minus
%    one)
% pos: a 1x2 vector storing the current position. The first element is row
%      index, and the second element is column index
% new_pos: a 1x2 vector of the new position
```

As an example, if we call this function by `randomWalk(U,R,D,L,[1 1])` where `U, R, D, L` is given by the example above, then it should return `(1,2)` with probability 0.2, `(2,1)` with 0.3, and `(1,1)` with 0.5.

2.2 Random Walk to Reach Goal

Now, given the `randomWalk` we implemented in the previous question, we can simulate a random walk on a 2D grid. We start the random walk at `(1,1)`, and keep walking until it finally reaches the given goal on the grid.

Implement the following function to do a random walk and return the number of steps and the trajectory. The number of steps includes the starting point `(1,1)` (e.g., if the goal is at `(1,1)`, then number of steps is one), and the trajectory is stored in two row-vectors for the row and column indices. Make effective use of function `randomWalk`.

```
function [n, r, c] = randomWalkToGoal(U, R, D, L, goal_pos)
% Start a random walk from (1,1) until the goal at goal_pos is reached.
% Return the path taken and its length.
% U, R, D, L: the 2D matrices of probabilities of going up, right, down, and left.
% goal_pos: the 1x2 vector of the goal position. The first element is row
%           index, and the second element is column index
% n: the length of the path (including (1,1))
% r: a 1 x n vector of the row indices of the path
% c: a 1 x n vector of the col indices of the path
```

2.3 Random Walk Statistics

Now we are interested in the statistics of the random walk when the size of the grid changes. To do this, you will write a single script to do the following things:

- Run `randomWalkToGoal` on the following grid sizes: `1x1, 5x5, 10x10, 12x12, 15x15, 17x17, 20x20`.
- The probabilities of going all 4 directions at any place of the grid are the same.
- The goal is at the lower right of the grid (i.e., `(m,m)` where `m` is the grid size).
- For each grid size, run the random walk 100 times, and compute the average number of steps required to reach the goal.
- For grid size = 20, compute the average (over 100 random walks) number of visits for each column and each row.
- Plot two figures and set its title and X-,Y-axis labels properly:
  - Figure 1. Show how the number of steps required increases as the grid size increases. X-axis represents the size of the grid, and Y-axis represents the average number of steps. Connect the points with a line.
  - Figure 2 Show the number of visits to each row and column. X-axis represents the row or column indices, and Y-axis represents the average number of visits. (Note that we have the same number of rows and columns so both curves can be plotted on the same figure.) Connect the points belonging to the same method with a line and use colors to differentiate between row and column statistics. Use `legend` to specify which line represents which statistics.

If the trajectory moves inside the same row or column (or simply stay at the same point) for a step, that row or column is considered visited again. For example, the trajectory `(2,2)` $\rightarrow$ `(2,3)` represents two visits at row 2, one visit at column 2 and one at column 3.

Use arrays and loops to test different grid sizes. *DO NOT* duplicate your code 7 times.

Save your script as `runtime.m`.

3 Self-check list

The following is a list of the minimum *necessary* criteria that your assignment must meet in order to be considered *satisfactory*. Failure to satisfy any of these conditions will result in an immediate request to resubmit your assignment. Save yourself and the graders time and effort by going over it before submitting your assignment for the first time.
Note that, although all of these are necessary, meeting all of them might still not be *sufficient* to consider your submission satisfactory. We cannot list everything that could be possibly wrong with any particular assignment!

Δ Comment your code! If any of your functions is not properly commented, regarding function purpose and input/output arguments, you will be asked to resubmit.

- Δ Suppress all unnecessary output by placing semicolons (;) appropriately. At the same time, make sure that all output that your program intentionally produces is formatted in a user-friendly way.
- Δ Make sure your functions names are *exactly* the ones we have specified, *including* case.
- Δ Check that the number and order of input and output arguments for each of the functions matches exactly the specifications we have given.
- Δ Test each one of your functions independently, whenever possible, or write short scripts to test them.
- Δ Check that your scripts do not crash (i.e., end unexpectedly with an error message) or run into infinite loops. Check this by running each script several times in a row. Before each test run, you should type the commands `clear all; close all;` to delete all variables in the workspace and close all figure windows.

4 Submission instructions

1. Upload files `insertionSort.m` `randomWalk.m`, `randomWalkToGoal.m` and `runtime.m` to CMS in the submission area corresponding to Assignment 1b in CMS.
2. Please do not make another submission until you have received and read the grader's comments.
3. Wait for the grader's comments and be patient.
4. Read the grader's comments carefully and think for a while.
5. If you are asked to resubmit, fix all the problems and go back to Step 1! Otherwise you are done with this assignment. Well done!