1 The Monty Hall Dillemma

You may have heard about the Monty Hall problem. It takes its name from the host of the American television show Let's Make a Deal. The problem has been formulated as follows:

"Suppose you’re on a game show and you’re given the choice of three doors. Behind one door is a car; behind the others, goats. The car and the goats were placed randomly behind the doors before the show. The rules of the game show are as follows: After you have chosen a door, the door remains closed for the time being. The game show host, Monty Hall, who knows what is behind the doors, now has to open one of the two remaining doors, and the door he opens must have a goat behind it. If both remaining doors have goats behind them, he chooses one randomly. After Monty Hall opens a door with a goat, he will ask you to decide whether you want to stay with your first choice or to switch to the last remaining door. Imagine that you chose Door 1 and the host opens Door 3, which has a goat. He then asks you “Do you want to switch to Door Number 2?"

Here is a short visual explanation of the problem.

1.1 Programming the game

Your first task is to program an interactive version of the game, where the code/computer is the host and the user is the player. Your code should ask the user for a first guess (a number from 1 to 3), open a door with a goat for the user, ask the user if s/he wants to switch and then communicate the outcome, i.e., car or goat. Ask the user if s/he wants to play again. Save your script in montyHallInteractive.m. Below is a possible scenario for this interaction:

```matlab
>> montyHallInteractive
Hello and welcome to the game!
Please choose one of the three doors: 3
You have chosen door number 3.
Door number 1 has a goat behind it!
Would you like to switch from door 3 to door 2?(y/n) y
Sorry, you lost. Door number 3 has the car :|
Would you like to play again?(y/n) n
Good bye!
```

You may be asking yourself: should the person switch or not? Do the odds of winning change after one door has been opened?

There are two ways of thinking about this:

- There is a 1/3 chance that the player will pick the prize door, and a 2/3 chance that s/he will miss the prize. Not switching means 1/3 is the probability to get the prize. However, if s/he missed (and this event has a 2/3 probability), then the prize is behind one of the remaining two doors. Furthermore, of these two, the host will open the empty one, leaving the prize door closed. Therefore, if the player misses and then switches, s/he is certain to get the prize. Summing up, if s/he does not switch the chance of winning is 1/3 whereas if s/he does switch the chance of winning is 2/3. Hard to believe, right?

---

After the host opens one door, two remain closed with equal probabilities of having the prize behind them. Therefore, regardless of whether the player switches or not, s/he has a 50-50 chance (i.e., probabilities 1/2) to hit or miss the prize door.

Which explanation is the right one? You will answer this question experimentally.

1.2 Simulating the game

Your must implement a function `simulateGame` that simulates the game:

```matlab
function i = simulateGame
    % i: the function returns 1 if the player wins by switching, and 0 if the players wins by not switching.
    Your code must simulate the game. One way of doing it is to randomly pick which door has the car and randomly pick which door is chosen by the player. If the player picked the right door in the first place than return 0, otherwise return 1.

1.3 Estimating probabilities

Using the previous function, simulate the games n times by implementing a function `runGames(n)`:

```matlab
function runGames(n)
    % n: the number of times to simulate the game.
    Your code must call `simulateGame`. Using `simulateGame`'s return value, compute the probability of winning by 1) always switching and 2) never switching. After you've simulate n games, display the probabilities in a visually pleasing way using bar graphs: you'll find useful Matlab's built-in function `bar`. See page 138 of the Gilat test book.

To make sure your code is doing the right thing, run it with a high n, e.g., 10000. Hint: watch the explanation for the probability distribution given in the movie [21].

2 Sorting 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. In this exercise we examine three ways in which sorting can be performed. All three are based on comparing adjacent entries in an array and swapping them if the values are not ordered (for the purpose of this exercise let's assume we want to order the array in ascending order). Do not use built-in function `sort`.

2.1 Bubble Sort

Bubble sort is a very simple sorting algorithm: it works by repeatedly going through the array to be sorted, comparing two adjacent items at a time and swapping them if they are in the wrong order. Specifically, in a pass through the array item 1 and 2 are compared and possibly swapped, then items 2 and 3 are compared and possibly swapped, and so forth until the end of the array is reached. The algorithm passes through the array as long as swaps continue to be needed, that is, the array is not yet completely sorted. The algorithm gets its name from the way smaller elements “bubble” to the beginning of the array. You must implement this algorithm in the function:

```matlab
function sortedArray = bubbleSort(A)
    % A: the vector of numbers to be sorted
% sortedArray: the sorted array

2.2 Insertion Sort

Insertion sort is another elementary sorting algorithm. Let the sequence to be sorted be \( a_1, a_2, \ldots, a_n \). The algorithm starts at the beginning of the sequence and for each element \( a_i \) it inserts it in the right place in the sequence preceding it. This is done by repeatedly swapping it with its neighbor to the left as long as the neighbor is larger than the current element. Thus, after \( a_i \) has found its place, the sequence \( a_1, \ldots, a_i \) will be sorted, while \( a_{i+1}, \ldots, a_n \) remains to be sorted. In the next step, the same procedure is performed for \( a_{i+1} \); and so on until the last element is inserted in its place and the array is sorted.

You must implement the sorting in place (i.e., do not use an auxiliary array). Save your code in the function:

```matlab
function sortedArray = insertionSort(A)
    % A: the vector of numbers to be sorted
    % sortedArray: the sorted array
```

2.3 Shell Sort

Shell sort is a variation of the insertion sort; it is named after its inventor: Donald Shell. It is based on the observation that insertion sort can be inefficient because it moves values only one position at a time. Shell sort speeds up insertion sort by comparing elements separated by a gap of several positions. This lets an element take “leaps” toward its expected position. Unlike insertion sort which only performs one pass through the data, multiple passes are taken with smaller and smaller gap sizes. In your implementation of this algorithm start with a gap size of \( \text{trunc}(\text{length}(A)/3) \). After each pass through the array, decrease the gap by half (round down if the result is not an integer), until the gap becomes 1. In other words, on the last pass through the array, Shell sort is a plain insertion sort, but by then, the array of data is guaranteed to be almost sorted.

Here is an example. The array to be sorted has 14 entries:

For the first pass through the array the gap is \( \text{trunc}(14/3) \), which is 4. You can think of this as grouping elements separated by 4 positions into small groups, on which insertion sort is performed. In the following, the same color signifies elements grouped together (note: you need not actually split the array into smaller arrays; the sorting should happen in place, i.e., in the same array):

Elements having the same color are sorted following the same procedure as in insertion sort. Here’s the result (note that same color entries are sorted; also, notice how large elements, such as 94, made big leaps in the array with only a few swaps):

In the second iteration, the new gap is obtained by dividing the previous gap by 2, i.e., \( \text{trunc}(4/2) \) which is 2. Again, same color signifies subgroups of elements to be sorted:

After the sorting, the array looks like this (note again, same color elements are sorted):
In the last iteration, the gap becomes 1 (i.e., just like a regular insertion sort), so all the elements belong to the same group:

2 7 9 25 13 30 18 36 21 47 53 60 71 94

And a final sorting is performed, resulting in a sorted array:

2 7 9 13 18 21 25 30 36 47 53 60 71 94

You must implement this algorithm in the function:

```matlab
function sortedArray = shellSort(A)
% A: the vector of numbers to be sorted
% sortedArray: the sorted array
```

2.4 Comparing Runtimes

To get an idea of the relative speed of these sorting methods, you will run each of them and compare their run times. To do so, you will write a single script that generates random arrays and runs each of the three methods on the generated arrays.

You should try at least three different array sizes: $10^2$, $10^3$, $10^4$, ... For each array size generate 10 arrays. Make sure that each generated array is sorted by each of the three methods! Compute the average running time for each array size and for each method. Display the results in a plot, where the x-axis represents the size of the array and the y-axis represents the time. Connect the points belonging to the same method with a line and use colors to differentiate between methods.

To record the time it takes to run a method use `tic` and `toc`. Together they provide the functionality of a stopwatch: `tic` starts the timer and `toc` stops it and returns the elapsed time. To exemplify their use, here's a code snippet measuring the time needed to run `bubbleSort` on some array `A`:

```matlab
tic;
bubbleSort(A);
elapsedTime = toc;
```

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!

- 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.
- 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.
Make sure your functions names are exactly the ones we have specified, including case.

Check that the number and order of input and output arguments for each of the functions matches exactly the specifications we have given. In particular, the functions required in this project are:

1. Function `montyHallInteractive`.
2. Function `simulateGame`, which returns 1 if the person is better off switching, 0 otherwise.
3. Function `runGames(n)`, which takes an input the number of simulations to be run and does not return anything.
4. Function `bubbleSort(A)`, which takes an array A and returns the sorted array.
5. Function `insertionSort(A)`, which takes an array A and returns the sorted array.
6. Function `shellSort(A)`, which takes an array A and returns the sorted array.
7. Script `runtime` calls the three sorting methods above.

Test each one of your functions independently, whenever possible, or write short scripts to test them.

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 `montyHallInteractive.m`, `simulateGame.m`, `runGames.m`, `bubbleSort.m`, `insertionSort.m`, `shellSort.m` and `runtime.m` to CMS in the submission area corresponding to Assignment 1 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!