CS1132 Fall 2008 Assignment 1

Adhere to the Code of Academic Integrity. You may discuss background issues and general solution strategies with others and seek help from course staff, but the homework you submit must be the work of just you. When submitting you assignment, be careful to follow the instructions summarized in Section 4 of this document.

1 The Key Game

In this contest `n` people have the chance to win a car. There are `n` available keys, one will start the car, while the others are fake. The players take turns, each selecting a key and trying it in the car. When a player chooses the correct key, he or she wins the car and the contest is over.
You may be asking yourself: does the order of the players matter? In other words, does each player have an equal chance of winning? Isn't it unfair that some players might not even get to choose? We will answer these questions *experimentally* in this exercise.

1.1 Understanding Probabilities

One way to understand the game is to consider two types of risk each player faces:

- Opportunity risk: the risk of not getting the chance to choose a key.
- Chance risk: the risk of choosing a wrong key.

These risks balance each other. Early players have high chance risk and low opportunity risk. Later players have high opportunity risk and low chance risk because some keys have been eliminated before they choose. For example, the first player has a 100% chance of getting to choose, but only one chance in `n` of choosing the correct key. The second player has `n-1` chances in `n` of getting to choose, but then has one chance in `n-1` of choosing the correct key. The last player has only one chance in `n` of getting to choose, but then has a 100% chance of choosing the right key. (There will be only one key left at that point.) Overall, the i^{th} player has `(n-i+1)/n` chance of playing and `1/(n-i+1)` chance of picking the right key. By multiplying these two probabilities we get the chance of winning for each player, which is `1/n`, i.e. the winning probability is independent of `i`, which means everyone gets the same chance.

1.2 Simulating one Game

Your first task will be to implement a function `keyGame(n)` that will simulate the game:

```
function i = keyGame(n)
% n:  the number of players in the game.
% i:  the rank of the winning player.  For example if 3 is returned,
% it means that the players 1 and 2 picked fake keys and player 3
% picked the right key.
```

Your code must simulate the game: players `1` to `n` take turns picking a key. The simulation stops when a player has picked the right key. You must pay particular attention that your program maintains the same probabilities of picking the right key as a real game would. Remember that for the first player the probability of picking the right key is 1 in n; for the second player it is 1 in n-1; while for the last player it is 1 in 1, that is if the last player actually gets to choose a key, he is guaranteed to win (since the last key would be the right key).

1.3 Estimating Probabilities

In this part, you will compute the probability of winning, as well as the probability of getting the chance to pick a key and the probability of picking the right key. You will implement the function `simulateKeyGames(games, n)`:

```
simulateKeyGames(games, n)
% games:  the number of games to be played
% n:  the number of players in the game
% It computes for each player three distinct
% probabilities and displays them in a table.
```

When simulating a game you must call the function `keyGame(n)`. For each player you must keep track of the number of times the player had the opportunity to choose and also the number of times the player chose the right key. Use these numbers to approximate the three probabilities mentioned above for each player. Display them neatly in the form of a table. For example for 5 players, one might get a table like this one:

	P1	P2	P3	P4	P5
Opportunity	1.000	0.801	0.599	0.401	0.199
Chance	0.201	0.250	0.332	0.548	1.000
Win	0.201	0.201	0.199	0.220	0.199

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).

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 pass through the array is repeated until no swaps occur, which means the array is 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:

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

The positions of the elements in bubble sort is quite important to the number of passes that will be needed to sort the array. Large elements at the beginning of the array do not raise problems, as they are quickly swapped. However, small elements towards the end move to the beginning extremely slowly. This has led to these types of elements being named ***rabbits*** and ***turtles***, respectively.

2.2 Cocktail Shaker Sort

To solve the problem with the turtles in bubble sort a variation of bubble sort was developed. The algorithm differs from bubble sort in that it sorts in both directions: it alternates between rightward passes (starting at the beginning of the array) and leftward passes (starting at the end of the array).
The first rightward pass will shift the largest element to its correct place at the end (just like in bubble sort), and the following leftward pass will shift the smallest element to its correct place at the beginning. The second set of right and left passes will shift the second largest and second smallest elements to their correct places, and so on. After $2i$ (i in each direction) passes, the first i and the last i elements in the list are in their correct positions, and do not need to be checked. By shortening the part of the list that is sorted each time, the number of operations can be halved. You must implement this version of the algorithm in the

function:

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

2.3 Gnome Sort

A variation of bubble sort that only takes one pass through the array is called Gnome Sort. Gnome Sort has been proposed by Dick Grune, a computer science professor at Amsterdam's Vrije Universiteit: "It is based on the technique used by the standard Dutch Garden Gnome to sort a line of flower pots. Basically, he looks at the flower pot next to him and the previous one; if they are in the right order he steps one pot forward, otherwise he swaps them and steps one pot backwards. Boundary conditions: if there is no previous pot, he steps forwards; if there is no pot next to him, he is done [sorting]."
You must implement this algorithm in the function:

```
function sortedArray = gnomeSort(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`:

```
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!

- 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.

∆ 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 `keyGame(n)`, which takes an integer value `n` and returns one output value: a number between `1` and `n`.
2. Function `simulatekeyGames(games, n)`, which takes two numbers as input and does not return anything.
3. Function `bubbleSort(A)`, which takes an array `A` and returns the sorted array.
4. Function `cocktailShakerSort(A)`, which takes an array `A` and returns the sorted array.
5. Function `gnomeSort(A)`, which takes an array `A` and returns the sorted array.
6. 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 `keyGame.m`, `simulateKeyGames.m`, `bubbleSort.m`, `cocktailShakerSort.m`, `gnomeSort.m` and `runtime.m` to CMS in the submission area corresponding to Assignment 1 in CMS.
2. Please don't 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!