# CS1132 Spring 2015 Assignment 1a Due Feb 11th

*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 3 of this document.

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

# 1 Shuffling Cards

In many games, it is necessary to shuffle a deck of cards throughly so as to guarantee randomness—hence excitement—in the game. In this problem you will study the efficiency of a method used to produce a perfect shuffling of a regular deck of 52 cards.
To keep things simple, we will represent the cards as numbers from 1 to 52, i.e., we don't care about their real values and suits. A shuffled deck will be represented by a vector such as $[c_1, c_2, ..., c_{52}]$ in which each card, i.e., each number from 1 to 52, appears once and there are no repetitions. Such a vector is also known as a *permutation* of the set `[1:52]`.

## 1.1 Generating a random number

Using the `help` command, learn about the `rand, ceil, fix`, and `floor` functions. After reading their brief descriptions, implement a random integer generator using those functions only. Do NOT use **randi**. Implement the following function:

```
function result = myRandInt(startInt,endInt)
% Returns an integer from startInt to endInt, inclusive, with equal probability.
% If startInt > endInt, use startInt as the upper bound and endInt as the lower
% bound.
```

## 1.2 Generating a random permutation

Your next task is to write a function `genRandomPerm(n)` that generates a permutation of `(1:n)`. Later, we will call this function with `n = 52` only, but it is better to have a parameter `n` than to base the function on a fixed value (52) so that the function can be used in a variety of contexts. The function must use the following strategy (do not use built-in function `randperm`):

1. Start with an empty vector `perm`.
2. Do the following until `perm` contains exactly `n` elements: Generate a random number between 1 and `n` and check whether it is in vector `perm`. If it is not, *append* it to `perm`. If it is, *discard* it. Repeat. Take advantage of the function `myRandInt`.

This algorithm is simple but has an important shortcoming that we will investigate. As vector `perm` grows larger, it becomes more likely that a newly generated random number is already in `perm` and hence has to be discarded. Thus the rate at which `perm` grows decreases over time. The result is that the total count of random numbers generated will, in general, be much greater than `n`. Can you guess how much greater? We will study this value with another function later.

Function `genRandomPerm(n)` has *two* return (output) arguments: the permutation generated, i.e., vector `perm`, and the count of how many random numbers between 1 and $n$ have been generated.

**Example:**

Assume we call `genRandomPerm(6)` and that the sequence of generated random numbers is 6, 3, 6, 5, 4, 3, 4, 2, 5, 3, 2, 1. Then, the algorithm will work as follows:
6 (append), 3 (append), 6 (discard), 5 (append), 4(append), 3(discard), 4(discard), 2 (append), 5 (discard), 3 (discard), 2 (discard), 1 (append and stop).
The produced permutation is `[6 3 5 4 2 1]` and is returned as the first output argument. The value 12, which is how many numbers were generated in total, is returned as the second output argument.

## 1.3 Is it a permutation?

Your next task is to test the function you have just written. For this, you will write a function `isPermutation(v)` that determines whether or not vector `v` is a permutation of the numbers from 1 to `length(v)`. This function should return 1 if `v` is such a permutation and 0 otherwise.
You may implement this function in anyway you like. Here is one possible approach:

- Sort `v` in ascending order and store the result as `w`. (You can easily sort a vector in MATLAB by using the function `sort`[1] ).
- Check that `w(i)` equals `i` for `i=1:length(v)`. If any of these checks fails, then you conclude that $v$ isn't a legitimate permutation.

Be sure to verify the correctness of function `isPermutation(v)` by means of a few test cases. For instance, `isPermutation([6 3 5 4 1 2])` should return 1, as the input vector is a permutation of `(1:6)` whereas `isPermutation([6 3 5 4 1 1 2])` should return `0` as `1` is repeated.
After testing `isPermutation` thoroughly, you can use it to test the output of your function `genRandomPerm`.

## 1.4 Investigating `genRandomPerm`

As we pointed out initially, we are interested in investigating the *efficiency* of function `genRandomPerm(n)` as measured by the count of how many random numbers need to be generated to make a complete permutation. Recall that this count is returned as the second output argument by `genRandomPerm(n)`. From here on, we will refer to this quantity simply as `count`.
We want to estimate the following statistics for `n=52`:

- What is the *average* value of `count`?
- What is the *most likely* value of `count`?
- How likely is it that `count` is bigger than 400 and smaller than 600?

Your task is to write a script `countStudy` to answer these questions. Your script should do the following:

1. The script should call `genRandomPerm(52)` a large number of times to build a cumulative sum of the `count` values returned. This allows you to calculate the average value of `count`. The number of trials, `M`, should be in the order of 10,000 or 100,000 for finding the average, but during program development, you may want to use a smaller `M`, e.g., 100.
2. The script should keep a vector `frequencies` of length 1000. Component `frequencies(c)` stores the number of times that `count` has the value `c`. Since we are generating permutations of 52 numbers, we expect components `frequencies(1:51)` to have the value zero. We can use a vector length of 1000 because we expect a `count` greater than 1000 to be extremely unlikely (since we are generating permutations of 52 numbers). However, if a `count` greater than 1000 does occur, simply discard that value (that trial).

[1] Type `help sort` to learn how to use the sort function

3. The script should draw a histogram of the `count` values. This is simple because you already have the data in `frequencies` and also because of MATLAB's built-in function `bar` for drawing a bar graph. The command `bar(frequencies)` will create and show a graph with 1000 bars (since `frequencies` is of length 1000). Be sure to add a meaningful title and axes labels to the plot to make it informative using built-in functions `title`, `xlabel`, and `ylabel`.

4. The script should output the following numerical results, accompanied by user friendly messages in English explaining what these results represent:

   - The average value of `count`
   - The most likely value of `count`, i.e., the value of `count` that occurred most frequently. This is sometimes called the *mode*. You can write your own code to find the *mode* or use built-in function `max`.[2]
   - The fractions of experiments with `count` $\geq$ `400` and `count` $\leq$ `600`.

## 2 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!

- $\Delta$ 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.
- $\Delta$ 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.
- $\Delta$ Make sure your functions names are *exactly* the ones we have specified, *including* case.
- $\Delta$ Check that the number and order of input and output arguments for each of the functions matches exactly the specifications we have given.
- $\Delta$ Test each one of your functions independently, whenever possible, or write short scripts to test them.
- $\Delta$ 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.

## 3 Submission instructions

1. Upload files `myRandInt`, `genRandomPerm.m`, `isPermutation.m` and `countStudy.m` to CMS in the submission area corresponding to Assignment 1a 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 need to resubmit, fix all the problems and go back to Step 1! Otherwise you are done with this assignment. Well done!

[2] You can learn about this function by typing `help max` in MATLAB's Command Window.