# SYSC 2100, Winter 2017
# Assignment 1: Recursion (Counting Things)
# Due: January 27, 2017

Wireless spectrum is a scarce resource, so a lot of researchers are interested in using it efficiently. For example, WiFi (IEEE 802.11) operates in either the 2.4 GHz or 5 GHz ISM band, with only a limited and finite amount of spectrum, typically 11 MHz for a single channel. As all nodes in a WiFi network access the shared channel at the same time, using a random access protocol, having more nodes increases the chances of collisions and wastes the spectrum. In a coffee shop or a lab, for example, that number could be high, say 50 nodes, all competing for access to the same 11 MHz channel. Would it be more efficient to divide the 11 MHz into 11 channels, each 1 MHz wide, and assign these 50 nodes randomly to one of these 11 channels? On the one hand, as the sub-channels are smaller, a node cannot transmit as fast (but different nodes using different sub-channels could transmit in parallel). And with only 4 or 5 nodes (on average) contending for access to such a sub-channel, the waste due to collisions would be reduced. Of course, if sub-channel assignment is truly random, sometimes some of the sub-channels will not be used (no node selects to use this channel), wasting spectrum.

To evaluate this strategy, we need to determine how many different ways there are to assign `n` nodes to `k` channels. And as the performance depends very much on how many nodes are assigned to a specific sub-channel, we need to know this as well. On the other hand, it is not important to know the identities of the nodes assigned to each sub-channel. Take a simple example of assigning 3 nodes to 2 channels. We will identify the channels by numbers, starting with 0, and nodes by characters, starting with A. Assuming that each node randomly and with equal probability picks one of the two channels, the following configurations are possible:

1. All three nodes (A, B, and C) select channel 0, and channel 1 is not used at all
   ```
   Channel 0: nodes A, B, C    Channel 1: empty
   ```
2. Two nodes select channel 0 and the remaining third node uses channel 1. There are three different possibilities for the identify of this third node, so there are three distinct ways in which this configuration can be achieved:
   ```
   Channel 0: nodes A, B       Channel 1: node C
   Channel 0: nodes A, C       Channel 1: node B
   Channel 0: nodes B, C       Channel 1: node A
   ```
3. One node selects channel 0 and the remaining two nodes select channel 1. Again, there are three different possibilities for the identity of this first node, so a total of three distinct ways exist to achieve this configuration:
   ```
   Channel 0: node A           Channel 1: nodes B, C
   Channel 0: node B           Channel 1: nodes A, C
   Channel 0: node C           Channel 1: nodes A, B
   ```
4. No node selects channel 0 and all three nodes select channel 1.
   ```
   Channel 0: empty      Channel 1: nodes A, B, C
   ```

In total, with `n` nodes being assigned randomly to `k` channels, $k^n$ different assignments exist (for the above example, $2^3 = 8$). However, for performance evaluation purposes, it does not matter which specific node for example is the sole node occupying Channel 1. So we need to know all possible configuration with distinct number of nodes across the channels, and keep track of how often such a configuration exists (which is a much smaller number than $k^n$). Write a recursive program, called **`ConfigurationCounting`** that does this job. It should produce output such as this for the example with
3 nodes being randomly assigned to 2 channels used above:

```
3 Nodes, 2 Channels:
1 set(s) with occupancies: 0 3
3 set(s) with occupancies: 1 2
3 set(s) with occupancies: 2 1
1 set(s) with occupancies: 3 0
Total number of assignments: 8
```

Another sample output, this time for 3 nodes being assigned to 3 channels (which has a total of $3^3 = 27$ possible assignments of nodes to channels), would be:

```
3 Nodes, 3 Channels:
1 set(s) with occupancies: 0 0 3
```

```
3 set(s) with occupancies: 0 1 2
3 set(s) with occupancies: 0 2 1
1 set(s) with occupancies: 0 3 0
3 set(s) with occupancies: 1 0 2
6 set(s) with occupancies: 1 1 1
3 set(s) with occupancies: 1 2 0
3 set(s) with occupancies: 2 0 1
3 set(s) with occupancies: 2 1 0
1 set(s) with occupancies: 3 0 0
Total number of assignments: 27
```

Your solution does not have to generate the configurations in exactly this order. The sum displayed in the last line should not be derived by simply evaluating $k^n$. Rather, the recursive function should return how many different assignments exist. To keep the programming task simple, you may choose to define the number of nodes and channels as constants in your program. Admittedly though it would be nicer of you could prompt the user to input these values. In developing the recursive solution, think through the steps we discussed in class:

1. How can you define the problem in terms of a smaller problem of the same type?
2. How does each recursive call diminish the size of the problem?
3. What instance of the problem can serve as the base case?
4. As the problem size diminishes, will you reach this base case?

Once you completed your program, test your solution with a range of different values for the number of nodes and the number of channels, including cases where the number of nodes exceeds the number of channels and some where the number of channels exceeds the number of nodes. To keep the running time and program output reasonably short, assume that neither value will exceed ten (otherwise, if, for example you were to assign 50 nodes to 11 channels, you would deal with $11^{50}$ assignments, resulting in many different configurations).

**Submission Requirements**: Submit your assignment (**the source files and bytecode**) using ***cuLearn***. Your program should compile and run as is in the default lab environment, and the code should be well documented. Submit all files without using any archive or compression as separate files. The main program should be called **ConfigurationCounting.java**, if you need to define additional classes etc., you are free to name them according to your own needs. But the TA(s) should be able to run your application by entering **`java ConfigurationCounting`** on a command-line.

Marks will be based on:

| | 25% | 50% | 75% | 100% |
|---|---|---|---|---|
| Program (12 marks) | Compiles | Runs | Correct, non-recursive solution to problem | Correct, recursive solution to the problem |
| Following good coding style (4 marks) | Hard to decipher | Can follow flow of control | Easy to read, meaningful variable and function names | Recursive solution to the problem. |
| Comments (2 marks) | Occasional | Lots, but extraneous | Few, but meaningful | Sufficient and high-quality comments, both in-line and method declaration |
| Completeness of your submission (2 marks) | Something in CULearn | Lots of effort to extract and run | Some effort to extract and run | Adhering to the submission requirements |

The due date is based on the time of the ***cuLearn*** server and will be strictly enforced. If you are concerned about missing the deadline, here is a tip: multiple submissions are allowed. So you can always submit a (partial) solution early, and resubmit an improved solution later. This way, you will reduce the risk of running late, for whatever reason (slow computers/networks, unsynchronized clocks, failure of the Internet connection at home, etc.).

In ***cuLearn***, you can manage the submission until the deadline, taking it back, deleting/adding files, etc, and resubmitting it. The system also provides online feedback whether you submitted something for an assignment. It may take a while to learn the submission process, so I would encourage you to experiment with it early and contact the TA(s) in case you have problems, as only assignments properly and timely submitted using ***cuLearn*** will be marked and will earn you assignment credits.