# Claremont McKenna College
# Computer Science

## CS 62 Handout 3: PS 3 September 23, 2015

This problem set is due **11:55pm on Thursday, October 1**.

**Problems 1 and 2:** Write all of your answers for Problems 1 and 2 in a *plain text file* called `p12.txt` or a Word file `p12.doc`. Organize your file for readability.

**Problems 3 and 4:** For Problems 3 and 4 be sure to follow the specifications exactly.

**Be sure to include** a comment at the top of each file submitted that gives your name and email address.

Submit the following files to the **Assignments** area on **Sakai**. Multiple submissions are allowed within the time limit. Please do **not** submit `.class` files or Eclipse project files.

```
p12.txt or p12.doc
Partition.java
SquareMatrix.java
any other file that we will need to grade your work.
```

## Problem 1

Using the Model of Computation that we described in lecture, do the following.

1. For each of the functions `f01` through `f10` in `Loops.java` given in the *given* folder, how many steps would it take to compute? Answer it by giving the running time complexity in $\Theta$ notation. Justification is not required.

2. For the functions `f02`, `f03`, `f05`, and `f08`, run each function by doubling and tripling the input size and see if you are getting the kind of running time that you expect to get. Notice that the number of times that the body of the inner loop is executed is also the final value of *sum*, which is returned by the function. This is the *empirical* way of measuring efficiency. For this part, you only need to say "Yes, I did it." or "No, I didn't do it.".

## Problem 2

For functions `f01, f02, f03, f04,` and `f08` (that is `f08`, not `f05`) in Problem 1, apply the formal *counting technique* that we studied in class. The result should be the running time complexity expressed in $\Theta$ notation. This is a way of verifying your answer to question 1 in Problem 1. That is, this is the *analytical* way of measuring efficiency.

## Problem 3

Given a positive integer $n$, an integer $p$, and an array $M$ consisting of $n$ integers $M[0], ..., M[n-1]$, reorder the elements in $M$ such that $M[0], ..., M[r-1]$ are all less then $p$, $M[r], ..., M[s-1]$ are all equal to $p$, and $M[s], ...M[n-1]$ are all greater than $p$, for some positive $r$ and $s$ with $0 \leq r \leq s \leq n$.

1. Describe an algorithm that solves this problem for all possible inputs $n, p$, and $M$, and whose worst-case running time is $O(n)$. Your algorithm is not allowed to use any other compound data structure (such as an additional array) than the array $M$ given to it as input. However, you are allowed to use additional variables of primitive data type, e.g., an `int` variable. Add the description of your algorithm as comments at the top of `Partition.java`.

   Among the integers less than $p$, they can be in any order. Similarly among the integers greater than $p$, they can be in any order.

2. Also, implement the algorithm in Java. Create a class named `Partition` in a file named `Partition.java`. In the file, implement the function with the following signature (Note that $n$ is not included in the parameter list since it would be redundant in Java):

```
public static void partition(int p, int[] M) {
    // your code goes here
}
```

3. Also test your implementation of `partition` by calling it with the following input data. So, you will have to create a `main` that calls `partition` once for each of the following data sets:

   - Once with `p = 5` and `M` containing `5, 6, 3, 5, 8, 7, 6, 5, 9, 2` in that order.
   - Another with `p = 5` and `M` containing `7, 6, 8, 9, 2, 3, 1, 5, 5, 5` in that order.
   - Yet one more time with `p = 5` and `M` containing `2, 3, 1, 7, 9, 8` in that order.

4. Now, prove that your implemented algorithm is indeed $O(n)$ using the *counting technique*. You can add the proof as comments at the bottom of `Partition.java` or as a separate file if you want to use something like Word.

**Hints:** There is an algorithm that meets the requirements that traverses the array *only once*. If your algorithm has to traverse the array two times, would it still be considered $\Theta(n)$? What if it has to traverse it three times? These questions in this Hints section do not need to be answered in the solution that you hand in, but it would help you formulate your answer for this problem.

## Problem 4

Suppose that each row of an $n$ by $n$ array `A` consists of 1's and 0's such that, in any row of `A`, all the 1's come before any 0's in that row.

Implement a method named `findRow` that returns the index of the row in `A` with the most number of 1's in linear time, i.e., in $\Theta(n)$. The first index of the input array is the row index and the second index is the column index, i.e., `A[row][column]`.

The method signature is given below:

```
// Pre:  A is a square matrix, i.e., two dimensional square array.
// Post: the row index for the row having the most number of 1's has
//       been returned.
public static int findRow (int[][] A) {
    // your code goes here
}
```

Put this method in a class called `SquareMatrix` in a file named `SquareMatrix.java`. Add a `main` to test your implementation of `findRow`. The `main` would look like something like the following:

```
public static void main(String[] args){
    int[][] A = {{1,0,0,0,0},
                 {1,1,1,0,0},
                 {1,1,0,0,0},
                 {1,1,1,1,0},
                 {0,0,0,0,0}};
    System.out.println("The row found is " + findRow(A));
}
```

Make up at least one more square array in addition to the one given above in testing your implementation and call `findRow` as many times as the number of arrays that you include in `main`.

In addition, explain why your algorithm is $\Theta(n)$. You do not need to use the formal counting technique. Explain in English in sufficient detail so that the reader (your grader) will be convinced that your explanation is sufficient and clear. Add your explanation as comments at the top of the file `SquareMatrix.java`.