1: Verify that one can have a good non-adaptive scheme for $n = 11$ through 13 coins by producing a three-row matrix without a good coin having $n$ columns, obeying the conditions that rows have sum 0, no two columns are the same or negative of each other, for each of these values of $n$.

Hint: you can without great effort produce 4 groups of 3 columns each, so that each group of 3 sums to 0; and also the all 0 column. These will handle the cases of 12 and 13 coins. To get the 11 case you may take the 12 solution, add a missing column and omit two columns that add up to it.

2: Write a program, or build a spreadsheet, for finding one bad coin out of 13 coins (without a good coin) using three weighings. You can do this with a spreadsheet, in pseudocode, in a programming language, or within Maple, Mathematica or MATLAB.

The input should be a list of the weights of the coins, and the output should be the number of the coin that is bad.

For Maple, Mathematica, or MATLAB, it may be useful to realize that the sign of the dot product of the list of weights of coins and the $k$th row will tell you how much more the left pan weighs than the right pan on the $k$ weighing.

For a spreadsheet, the `sum`, `if`, `and`, or `or` functions. may be useful.

3: Consider the following not terribly efficient sorting algorithm.

(a) place the $n$ numbers to be sorted in a $\sqrt{n} \times \sqrt{n}$ array.

(b) sort each of the columns in increasing order

(c) sort each of the rows in increasing order

At this point both the rows and columns should be in increasing order.

(d) Repeat (d1) and (d2) until the array is empty.

(d1) The top left position is the smallest of the objects in the array. Remove it and place it in a list.

(d2) Now slide either the element directly below the empty square or the element directly to its right into its position, so that all the columns and rows are still in increasing order. There is now a new empty square. Fill it using the same sliding process, and repeat until all empty squares are together in the lower right of the array.

3.1: What is (in big-O notation; that is, up to a multiplicative constant) the asymptotic running time of the algorithm. Does it matter how you perform the sorting in steps (b) and (c)?
3.2: Show that after step (c), both the rows and columns are in increasing order, as claimed.

Hint: Consider columns \( j \) and \( j + 1 \). Show that the \( k \)th smallest element in column \( j \) is smaller than the \( k \)th smallest element in column \( j + 1 \).

3.3: Does this sorting algorithm resemble any of the four that I described in class? Explain any similarities.

3.4: The sorting algorithm as described above seems difficult to implement “in place.” That is, only using a small amount of extra memory in addition to the original \( n \) memory cells taken by the items to be sorted. Show how you can modify the algorithm described above to implement it “in place.”

Hint: Modify step (d1) so that instead of leaving the top left square empty in step (d1), you stick the rightmost element of the lowest non-empty row in it. How does step (d2) need to be modified?