In [11]:
```
from __future__ import division
```

Exercise1

In lecture we saw the function `make_first_coeff_nonzero(eqs)` that took 3 equations in 3 variables and ensured that the first coefficient of the first equation is nonzero.

Write a general version of this function that would work for *every number of equations*.

That is, `make_first_coeff_nonzero_general(eqs)` will take a list of lists of numbers, and change its ordering so that the first number of the first list is nonzero. (You don't have to worry about the case that all lists have their first number zero.)

Here are examples of outputs:

In [8]:
```
L = [ [1,2,3],[4,5,6],[7,8,9]]
make_first_coeff_nonzero_general(L)
L
```

Out[8]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In [9]:
```
L = [ [0,2,3],[4,5,6],[7,8,9]]
make_first_coeff_nonzero_general(L)
L
```

Out[9]:
```
[[4, 5, 6], [0, 2, 3], [7, 8, 9]]
```

In [10]:
```
L = [ [0,1,2,3,4] , [0,5,6,7,8], [0,9,10,11,12],[1,2,4,6,8]]
make_first_coeff_nonzero_general(L)
L
```

Out[10]:
```
[[1, 2, 4, 6, 8], [0, 5, 6, 7, 8], [0, 9, 10, 11, 12], [0, 1, 2, 3, 4]]
```

Exercise 2:

Write the function `solve7(eqs)` that solves 7 equations in 7 variables (that is, the equations are in variables $x_0, x_1, x_2, \ldots, x_6$ and for $i = 0, 1, \ldots, 6$ the i^{th} equation has the form $eqs[i][0]x_0 + eqs[i][1]x_1 + \cdots + eqs[i][6]x_6 + eqs[i][7] = 0$.)

This may seem that it would be very complicated, but you can use the provided function `solve6(eqs)` that solves 6 equations in 6 variables. You can also use `make_first_coeff_nonzero_general` as well as the functions `multiply_equation` and `add_equations` that actually already work for equations of any number of variables.

In [16]:
```
def multiply_equation(eq,num):
    """Multiply all coefficients of equation eq by number num.
       Return result"""
    res = []
    for x in eq:
        res += [x*num]
    return res
```

In [17]:
```
def add_equations(eq1,eq2):
    """Add eq1 and eq2. Return result"""
    res = []
    for i in range(len(eq1)):
        res.append(eq1[i]+eq2[i])
    return res
```

In [67]:
```
# use the following function but don't copy its code!
# you don't need to read or understand the code also
#
# this function takes eqs : a list of 6 lists, where each list has 7 numbers c
orresponding
# to an equation of the form a_0x_0+a_1x_1+...+a_5x_5+a_6 = 0
# the function returns a solution of these equations: a list of 6 numbers corr
esponding to the
# values of the variables x_0,...,x_6
import numpy as np
def solve6(eqs):
    A = np.ndarray([6,6])
    for i in range(6):
        for j in range(6):
            A[i,j] = eqs[i][j]
    b = np.ndarray([6,1])
    for i in range(6):
        b[i,0] = -eqs[i][6]
    C = np.linalg.inv(A)
    sol = np.dot(C,b)
    return [round(sol[i,0],3) for i in range(6)]
```

In [110]:
```
solve7( [ [1, 1, 1, 0, 1, -1, -21],
          [-1, -1, -1, 0, 1, 0, 6],
          [-1, 0, 0, 1, 0, 0, 10],
          [0, 1, -1, 1, 0, 0, 11],
          [0, -1, -1, -1, -1, 0, 2],
          [-1, -1, -1, -1, 0, -1, -10]
        ])
```

Out[110]:
```
[0.0, 4.0, 5.0, -10.0, 3.0, -9.0]
```

In [94]:
```
solve7( [ [1, -1, 0, -1, 1, -1, 23],
          [1, 0, -1, -1, 0, 0, 10],
          [-1, 0, 0, -1, 1, 0, -5],
          [0, 1, 1, 0, -1, 0, -14],
          [0, 0, -1, 0, 0, 0, 3],
          [1, 0, 0, 0, 0, -1, 15]
        ])
```

Out[94]:
```
[-10.0, 3.0, 3.0, -3.0, -8.0, 5.0]
```

In [121]:
```
solve7( [ [0, -1, 1, 1, 0, 1, 7],
          [-1, 0, 1, 0, 1, -1, -8],
          [0, 1, -1, 1, -1, 1, 15],
          [0, 0, -1, 1, -1, -1, 7],
          [0, -1, 1, 0, 0, 1, 4],
          [1, 1, 0, 0, 1, 1, -7]
        ])
```

Out[121]:
```
[2.0, -2.0, -3.0, -3.0, 10.0, -3.0]
```

Exercise 3:

Write a function `solve(eqs)` that can solve n equations in n variables for every number n. In particular it should be the case that if you have 3 equations in 3 variables then it would hold that `solve(eqs)==solve3(eqs)`, if you have 4 equations in 4 variables then `solve(eqs)=solve4(eqs)` and if you have 7 equations in 7 variables then `solve(eqs)==solve7(eqs)`

Hint: Use *recursion*: when given as input n equations in n variables `solve` should work on these equations to reduce the task to solving $n-1$ equations on $n-1$ variables, at which point it can call *itself* to do so.

Here are some input examples:

In [123]:
```
solve( [ [-1, 1, -1, 1, 0, -2],
         [-1, 0, 1, -1, 1, -1],
         [1, -1, 0, 0, -1, -4],
         [-1, 1, 0, -1, 0, 5],
         [-1, -1, 0, -1, -1, -10]
       ])
```

Out[123]:
```
[-7.0, -4.0, 9.0, 8.0, -7.0]
```

In [126]:
```
solve( [ [-1, -1, -1, -1, -1, 8],
         [1, 1, 1, 1, -1, 12],
         [0, 1, 1, -1, 0, -4],
         [-1, -1, 0, 1, -1, 18],
         [1, -1, 1, 0, 1, -36]
      ])
```

Out[126]:
```
[10.0, -10.0, 6.0, -8.0, 10.0]
```

In [128]:
```
solve( [ [1, 1, 0, -1, 1, -1, -11],
         [1, 1, -1, -1, -1, 1, -19],
         [0, -1, -1, -1, -1, 1, -10],
         [0, 0, 1, -1, 0, 0, 6],
         [0, 1, 0, -1, 0, 1, -4],
         [-1, 1, 1, 1, 1, -1, 13]
      ])
```

Out[128]:
```
[3.0, 3.0, -10.0, -4.0, -2.0, -3.0]
```

In [132]:
```
solve( [ [1, -1, 0, -1, 1, 1, -1, -1, 9],
         [-1, 0, -1, 0, 1, -1, 1, 0, -15],
         [1, -1, 1, -1, -1, -1, 0, -1, 11],
         [-1, 1, 0, -1, 1, 0, 1, 1, 11],
         [1, 1, -1, -1, -1, -1, 0, 0, -5],
         [-1, 0, 0, 1, 1, -1, 0, 1, -14],
         [1, 1, 1, 1, -1, 1, -1, 0, 9],
         [1, 0, -1, 1, 0, -1, -1, -1, -26]
      ])
```

Out[132]:
```
[-3.0, -1.0, -10.0, 7.0, 0.0, -6.0, -4.0, -2.0]
```

In [138]:
```
solve( [ [0, -1, -1, -16],
         [0, -1, 0, -6],
         [-1, 0, 0, 0]
      ])
```

Out[138]:
```
[0.0, -6.0, -10.0]
```