Yesterday and this morning we saw how we can solve linear equations in 3,4 variables, and some of you have even seen how to do it in general.

We will now go over this more carefully.

### Helper functions

```python
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

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
```

**Solving linear equations - general recipe**

**Input:** List of $n$ equations $eqs = [eqs[0], \ldots, eqs[n-1]]$ in $n$ variables. For every $i = 0..n-1$, $eqs[i]$ is a list of $n+1$ numbers.

**Output:** List of $n$ numbers $[x_0, \ldots, x_{n-1}]$ that are a solution to the equations. That is, for every $i = 0..n-1$,

$$eqs[i][0]x_0+\ldots+eqs[i][n-1]x_{n-1}+eqs[i][n] = 0$$

**Assumption:** We already know how to solve $n-1$ equations in $n-1$ variables.

**Operation:**

- Ensure that the first coefficient of the first equation is nonzero
- Divide the first equation by its first coefficient to make it one.
- For every $i = 1..n-1$, add to the $i^{th}$ equation a copy of the first equation multiplied by $-eqs[i][0]$ so now the first coefficients of equations $1, \ldots, n-1$ is zero.
- Run a solver for the last $n-1$ equations and last $n-1$ variables.
- Use solution to get solution for first variable.
def solve100(eqs):
    n = len(eqs)
    make_first_coeff_nonzero_general(eqs)  # make 1st coef of 1st equation non zero
    eqs[0] = multiply_equation(eqs[0], 1/eqs[0][0])
    # make 1st coef of 1st equation equal 1

    for i in range(1, n-1):
        eqs[i] = add_equations(eqs[i], multiply_equation(eqs[0], -eqs[i][0]))  # zero out first coefficient in eqs 1,2
        # make 1st coef of 2nd .. n-th equation equal zero

    rest_equations = []
    for i in range(1, n):
        rest_equations.append(eqs[i][1:n+1])

    solutions = solve99(rest_equations)
    # solve remainder of equations for remainder of variables

    x = -eqs[0][n]
    for i in range(1, n):
        x -= eqs[0][i] * solutions[i-1]
        # solve 1st variable using solution for 2nd and 3rd variable

    return [x] + solutions
```python
In [23]: def solve99(eqs):
    n = len(eqs)
    make_first_coeff_nonzero_general(eqs)  # make 1st coef of 1st equation non zero
    eqs[0] = multiply_equation(eqs[0],1/eqs[0][0])
    # make 1st coef of 1st equation equal 1
    for i in range(1,n-1):
        eqs[i] = add_equations(eqs[i],multiply_equation(eqs[0],-eqs[i][0]))  # zero out first coefficient in eqs 1,2
        # make 1st coef of 2nd .. n-th equation equal zero
    rest_equations = []
    for i in range(n):
        rest_equations.append(eqs[i][1:n+1])
    solutions = solve98(rest_equations)
    # solve remainder of equations for remainder of variables
    x = -eqs[0][n]
    for i in range(1,n):
        x -= eqs[0][i]*solutions[i-1]
        # solve 1st variable using solution for 2nd and 3rd variable
    return [x] + solutions

So, we could write functions solve1,solve2,...,solve10000
```
But we can see that they are very similar. The solution for $\text{solvem}$ uses $\text{solvem} - 1$ This suggests that we should use recursion

```python
In [22]: def solve(eqs):
    n = len(eqs)
    make_first_coeff_nonzero_general(eqs)  # make 1st coef of 1st equation non zero
    eqs[0] = multiply_equation(eqs[0], 1/eqs[0][0])
    # make 1st coef of 1st equation equal 1
    for i in range(1,n-1):
        eqs[i] = add_equations(eqs[i], multiply_equation(eqs[0], -eqs[i][0]))  # zero out first coefficient in eqs 1,2
        # make 1st coef of 2nd .. n-th equation equal zero
    rest_equations = []
    for i in range(1,n):
        rest_equations.append(eqs[i][1:n+1])
    solutions = solve(rest_equations)
    # solve remainder of equations for remainder of variables
    x = - eqs[0][n]
    for i in range(1,n):
        x -= eqs[0][i]*solutions[i-1]
        # solve 1st variable using solution for 2nd and 3rd variable
    return [x] + solutions

Let's see if it works:
```
Since it didn't work let's try to see where the problem was: let's print the length of equations so we understand where in the recursion it fails.

```python
In [20]: def solve(eqs):
    n = len(eqs)
    print("Solving ", n, " equations in ", n, "variables"
    make_first_coeff_nonzero_general(eqs) # make 1st coef of 1st equation nonzero
    eqs[0] = multiply_equation(eqs[0],1/eqs[0][0])
    # make 1st coef of 1st equation equal 1

    for i in range(1,n-1):
        eqs[i] = add_equations(eqs[i],multiply_equation(eqs[0],-eqs[i][0])) # zero out first coefficient in eqs 1,2
        # make 1st coef of 2nd .. n-th equation equal zero

    rest_equations = []
    for i in range(1,n):
        rest_equations.append(eqs[i][1:n+1])

    solutions = solve(rest_equations)
    # solve remainder of equations for remainder of variables

    x = - eqs[0][n]
    for i in range(1,n):
        x -= eqs[0][i]*solutions[i-1]
        # solve 1st variable using solution for 2nd and 3rd variable

    return [x] + solutions
```
We tried to solve zero equations in zero variables! No wonder we ran into trouble.

The problem is that we need to always have a base for the recursion. Just like we need a base for proofs by inductions in mathematics.

Here is an updated version:
In [35]:
def solve(eqs):
    n = len(eqs)
    print "Solving ", n, " equations in ", n, "variables"
    if n==1:
        return [ -eqs[0][1]/eqs[0][0] ]
    make_first_coeff_nonzero_general(eqs)  # make 1st coef of 1st equation non zero
    eqs[0] = multiply_equation(eqs[0],1/eqs[0][0])
    # make 1st coef of 1st equation equal 1

    for i in range(1,n):
        eqs[i] = add_equations(eqs[i],multiply_equation(eqs[0],-eqs[i][0]))  # zero out first coefficient in eqs 1,2
        # make 1st coef of 2nd .. n-th equation equal zero

    rest_equations = []
    for i in range(1,n):
        rest_equations.append(eqs[i][1:n+1])

    solutions = solve(rest_equations)
    # solve remainder of equations for remainder of variables

    x = - eqs[0][n]
    for i in range(1,n):
        x -= eqs[0][i]*solutions[i-1]
        # solve 1st variable using solution for 2nd and 3rd variable

    return [x] + solutions

In [36]: solve([[1,2,3],[4,5,6]])

Solving  2 equations in  2 variables
Solving  1 equations in  1 variables
Solving  0 equations in  0 variables

Out[36]: [1.0, -2.0]

Let's check if it can solve 25 equations in 25 variables.
In [42]:
    n = 25
    solutions = []
    for j in range(n):
        solutions.append(j)

In [43]:
solutions

Out[43]:
[0,
  1,
  2,
  3,
  4,
  5,
  6,
  7,
  8,
  9,
 10,
 11,
 12,
 13,
 14,
 15,
 16,
 17,
 18,
 19,
 20,
 21,
 22,
 23,
 24]

In [44]:
    import random
    equations = []
    for i in range(n):
        constant_term = 0
        eq = []
        for j in range(n):
            x = random.randint(-100, +100)
            eq.append(x)
            constant_term -= x*solutions[j]
        eq.append(constant_term)
        equations.append(eq)
In [49]: my_solutions = solve(equations)

Solving 25 equations in 25 variables
Solving 24 equations in 24 variables
Solving 23 equations in 23 variables
Solving 22 equations in 22 variables
Solving 21 equations in 21 variables
Solving 20 equations in 20 variables
Solving 19 equations in 19 variables
Solving 18 equations in 18 variables
Solving 17 equations in 17 variables
Solving 16 equations in 16 variables
Solving 15 equations in 15 variables
Solving 14 equations in 14 variables
Solving 13 equations in 13 variables
Solving 12 equations in 12 variables
Solving 11 equations in 11 variables
Solving 10 equations in 10 variables
Solving 9 equations in 9 variables
Solving 8 equations in 8 variables
Solving 7 equations in 7 variables
Solving 6 equations in 6 variables
Solving 5 equations in 5 variables
Solving 4 equations in 4 variables
Solving 3 equations in 3 variables
Solving 2 equations in 2 variables
Solving 1 equations in 1 variables
Solving 0 equations in 0 variables

In [50]: my_solutions

Out[50]: [-4.0456527017340704e-13,
0.999999999998579,
1.999999999998543,
3.000000000005302,
4.000000000001315,
4.99999999999893,
6.00000000000206,
6.99999999999762,
8.00000000000284,
8.99999999999783,
9.99999999999716,
11.00000000000341,
12.00000000000338,
12.99999999999893,
14.00000000000146,
14.99999999999332,
15.9999999999972,
16.9999999999982,
17.9999999999974,
19.00000000000005,
19.9999999999986,
20.99999999999563,
22.00000000000005,
22.99999999999698,
23.9999999999945]
In [51]:
round_solutions = []
for x in my_solutions:
    round_solutions.append(round(x,3))
round_solutions

Out[51]: [-0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0, 20.0, 21.0, 22.0, 23.0, 24.0]
In [53]: [round(x,3) for x in my_solutions ]

Out[53]: [-0.0,  
1.0,  
2.0,  
3.0,  
4.0,  
5.0,  
6.0,  
7.0,  
8.0,  
9.0,  
10.0,  
11.0,  
12.0,  
13.0,  
14.0,  
15.0,  
16.0,  
17.0,  
18.0,  
19.0,  
20.0,  
21.0,  
22.0,  
23.0,  
24.0]

**Sorting**

We have now obtained a function `solve` that solves general linear equations. This is still not enough however. Eventually, we want to be able to achieve a function that reads the equations and outputs the solution, like the following:

In [66]: `solve_eqs()`

Number of variables / equations?3
Enter equation number 1:  5x - y + z = 0
Enter equation number 2:  2y - 3y = 4 + z
Enter equation number 3:  10z - 4x +3y = 20
Solving 3 equations in 3 variables
Solving 2 equations in 2 variables
Solving 1 equations in 1 variables
Solving 0 equations in 0 variables
x = -2.13953488372
y = -7.3488372093
z = 3.3488372093

Note that the equations now are given in arbitrary order, so we will need to sort them to make them into the standard format of $ax + by + cy + d = 0$ so we can extract the coefficients $[a, b, c, d]$ for our solve function.
So, we will now talk about sorting lists. That is, coming up with a function `sort_list`.

```
In [77]: sort_list([9,8,7,6,5])
Out[77]: [5, 6, 7, 8, 9]
```

```
In [78]: sort_list([100,3,4,8,7])
Out[78]: [3, 4, 7, 8, 100]
```

```
In [79]: sort_list([3,1,4,1,5,9,2])
Out[79]: [1, 1, 2, 3, 4, 5, 9]
```

As usual, we will start by writing `sort2`:

```
In [7]: def sort2(L):
    if L[0]>L[1]:
        L[0],L[1] = L[1],L[0]
    return L
```

```
In [8]: sort2([1,2])
Out[8]: [1, 2]
```

```
In [9]: sort2([2,1])
Out[9]: [1, 2]
```

And then `sort3`:

```
In [17]: def sort3(L):
    if L[0]>L[1]:
        L[0],L[1] = L[1],L[0]
    if L[0]>L[2]:
        L[0],L[1] = L[1],L[0]
    return [L[0]] + sort2(L[1:3])
```

```
In [18]: sort3([9,5,8])
Out[18]: [5, 8, 9]
```

**Theorem:** For every three numbers $x_0, x_1, x_2$, $\text{sort3}([x_0, x_1, x_2])$ returns a list $[x_i, x_j, x_k]$ such that $x_i \leq x_j \leq x_k$ and $i, j, k$ are distinct numbers in $\{0, 1, 2\}$. 

Proof: Suppose we run $\text{sort3}([x_0, x_1, x_2])$. Let's split into cases:

**Case 1:** $x_0 \leq \min\{x_1, x_2\}$. Then both if's don't execute, and we output $[x_0] + \text{sort2}( [x_1, x_2] )$. Since $x_0$ is the smallest element then this output will be sorted.

**Case 2:** $x_0 > x_1$ but $x_1 \leq x_2$. Then the first if executes and after it is done, $L[0] = x_1$. Because $x_1 \leq x_2$, the second if does not execute, and we output $[x_1] + \text{sort2}( [x_0, x_2] )$. Since $x_1$ is the smallest element then this output will be sorted.

**Case 3:** $x_0 > x_1$ and $x_1 > x_2$. Then the first if executes, and after it $L[0] = x_1$ and then the second if executes and after it, $L[0] = x_2$. We output $[x_2] + \text{sort2}( [x_1, x_0] )$ which will be sorted since $x_2$ is the smallest element.

**Curious fact:**

```
In [19]: sort3(['cat','apple','dog'])
Out[19]: ['apple', 'cat', 'dog']
```

```
In [21]: 'apple' < 'cat'
Out[21]: True
```

```
In [22]: 'car' > 'cat'
Out[22]: False
```

**Lab Work**

**Exercise 1**

Write the function $\text{sort4}(L)$ that takes a list of 4 elements and sorts it. The last line of the function must be return $[L[0]] + \text{sort3}(L[1:4])$.

```
In [30]: sort4 = sort_list
```

```
In [31]: def sort4(L):
    
    #
    # your code goes below
    #
    return [L[0]] + sort3(L[1:4])
```
Here are some output examples:

In [27]: `sort4([7,8,1,2])`
Out[27]: [1, 2, 7, 8]

In [28]: `sort4([1,9,2,3])`
Out[28]: [1, 2, 3, 9]

In [29]: `sort4(['Mickey','Donald','Goofy','Minney'])`
Out[29]: ['Donald', 'Goofy', 'Mickey', 'Minney']

Exercise 2

Suppose that you are given the function `sort9` that sorts a list of 9 elements. Write a function `sort10(L)` that sorts a list `L` of 10 elements. The last line of the function must be `return [L[0]]+sort9(L[1:4])`

In [32]: 
   # you can use this function as a "black box" but there's no need to read it or understand its code
   def sort9(L):
       return sorted(L[0:9])

In [33]: def sort10(L):
    #
    # your code goes below
    #
    return [L[0]] + sort9(L[1:10])

In [37]: sort10([0, 1, 6, 10, 9, 3, 3, 9, 9, 5])
Out[37]: [0, 1, 3, 3, 5, 6, 9, 9, 9, 10]

In [40]: sort10([15, 16, 19, 13, 5, 1, 7, 19, 12, 4])
Out[40]: [15, 1, 4, 5, 7, 12, 13, 16, 19, 19]

In [42]: sort10([5, 9, 13, 8, 15, 17, 20, 9, 10, 8])
Out[42]: [5, 8, 8, 9, 9, 10, 13, 15, 17, 20]

Exercise 3

Use recursion to write the general `sort_list(L)` function that works for lists of any length. Again, the last line of your code must be a recursive call to `sort_list` of the form `return [L[0]]+sort_list(L[1:len(L)])`
In [43]: def sort_list(L):
    #
    # your code goes below
    #
    return [L[0]] + sort_list(L[1:len(L)])


The array below contains the names of all the students that were registered to the course. Compute an array that contains these students in alphabetical order by first name. Use the function you wrote to sort it by first name.


**Exercise 4**

Sort the array above in *reverse alphabetical order* by first name (so that L[0] will be the name that is last in alphabetical order and L[80] will be the name that is first)

**Exercise 5 (bonus)**

Sort the array in alphabetical order by *last name*.