Sample solutions

Bookmark this page

part1 (Score: 9.0 / 9.0)

1. Test cell (Score: 5.0 / 5.0)
2. Test cell (Score: 1.0 / 1.0)
3. Test cell (Score: 3.0 / 3.0)

Important note! Before you turn in this lab notebook, make sure everything runs as expected:

- First, **restart the kernel** -- in the menubar, select Kernel→Restart.
- Then **run all cells** -- in the menubar, select Cell→Run All.

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE."

Part 1: Intro to Numpy/Scipy

Numpy (http://www.numpy.org/) is a Python module that provides fast primitives for multidimensional arrays. It's well-suited to implementing numerical linear algebra algorithms, and for those can be much faster than Python's native list and dictionary types when you only need to store and operate on numerical data.

Some of the material from this lesson is copied from the following, and more comprehensive, tutorial: link (http://www.scipy-lectures.org/intro/numpy/index.html)

Quick demo. The recommended importing idiom is:

In [1]:
```
import numpy as np
```

Creating a simple numpy array

In [2]:
```
a = np.array([1,2,3,4])
print(a)
```

```
[1 2 3 4]
```

Why bother with Numpy? A motivating example

We already have lists and dictionary types, which are pretty easy to use and very flexible. So why bother with this special type?

Exercise 0 (ungraded). One reason to consider Numpy is that it "can be much faster," as noted above. But how much faster is that? Run the experiment below to see.

In [3]:
```
n = 1000000
```

In [4]:
```
L = range(n)
%timeit [i**2 for i in L]
```

```
1 loop, best of 3: 502 ms per loop
```

In [5]:
```
np.arange(10) # Moral equivalent to `range`
```

Out[5]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [6]:
```
A = np.arange(n)
%timeit A**2
```

```
100 loops, best of 3: 3.53 ms per loop
```

Creating multidimensional arrays

Beyond simple arrays, Numpy supports multidimensional arrays. To do more than one dimension, call `numpy.array()` but nest each new dimension within a list. It's easiest to see by example.

In [7]:
```
# Create a two-dimensional array of size 3 rows x 4 columns:
B = np.array([[0, 1, 2, 3],
              [4, 5, 6, 7],
              [8, 9, 10, 11]])

print(B)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

In [8]:
```
print(B.ndim) # What does this do?
print(B.shape) # What does this do?
print(len (B)) # What does this do?
```

```
2
(3, 4)
3
```

In [9]:
```
C1 = [[0, 1, 2, 3],
      [4, 5, 6, 7],
      [8, 9, 10, 11]]

C2 = [[12, 13, 14, 15],
      [16, 17, 18, 19],
      [20, 21, 22, 23]]

C = np.array([C1, C2])

print(C)
print(C.ndim)
print(C.shape)
print(len (C))
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
3
(2, 3, 4)
2
```

There are routines for creating various kinds of structured matrices as well, which are similar to those found in MATLAB (http://www.mathworks.com/products/matlab/) and Octave (https://www.gnu.org/software/octave/).

In [10]:
```
print(np.zeros((3, 4)))
```

```
[[ 0.  0.  0.  0.]
```

```
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```

In [11]:
```
print(np.ones((3, 4)))
```

```
[[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
```

In [12]:
```
print(np.eye(3))
```

```
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

In [13]:
```
print(np.diag([1, 2, 3]))
```

```
[[1 0 0]
 [0 2 0]
 [0 0 3]]
```

You can also create empty (uninitialized) arrays. What does the following produce?

In [14]:
```
A = np.empty((3, 4)) # An "empty" 3 x 4 matrix
print(A)
```

```
[[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
```

Exercise 1 (ungraded). The following code creates an identity matrix in two different ways, which are found to be equal according to the assertion. But in fact there is a subtle difference between the `I` and `I_u` matrices created below; can you spot it?

In [15]:
```
n = 3
I = np.eye(n)

print("==> I = eye(n):")
print(I)

u = [1] * n
I_u = np.diag(u)

print("\n==> u:\n", u)
print("==> I_u = diag (u):\n", I_u)

assert np.all(I_u == I)
```

```
==> I = eye(n):
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]

==> u:
 [1, 1, 1]
==> I_u = diag (u):
 [[1 0 0]
 [0 1 0]
 [0 0 1]]
```

Answer. Give this some thought before you read the answer that follows!

The difference is in the element types. The `eye()` function returns an identity matrix and uses a floating-point type as the element type. By contrast, `diag()`, which expects a list of initializer values upon input, derives the element type from that input. In this case, `u` contains values that will be stored as integers; therefore, `diag()` constructs its output assuming integer elements.

Try running `print(I_u.dtype)` and `print(I.dtype)` to confirm that these element types differ.

Indexing and slicing

Indexing and slicing

The usual 0-based slicing and indexing notation you know and love from lists is also supported for Numpy arrays. In the multidimensional case, including their natural multidimensional analogues with index ranges separated by commas.

In [16]:
```
# Recall: C
print (C)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

What part of C will the following slice extract? Run the code to find out.

In [17]:
```
print (C[0, 2, :])
```

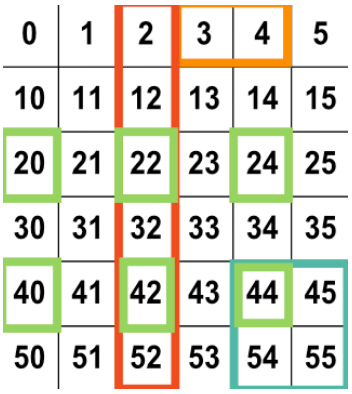
```
[ 8  9 10 11]
```

What will the following slice return? Run the code to find out.

In [18]:
```
print (C[1, 0, ::-1])
```

```
[15 14 13 12]
```

Exercise 2 (5 points). Consider the following 6×6 matrix, which has 4 different subsets highlighted.



For each subset illustrated above, write an indexing or slicing expression that extracts the subset. Store the result of each slice into `Z_green`, `Z_red`, `Z_orange`, and `Z_cyan`.

In [19]:
Student's answer (Top)

```
Z= np.array([[0,1,2,3,4,5],[10,11,12,13,14,15],[20,21,22,23,24,25],[30,31,32,33,34,35],
[40,41,42,43,44,45],[50,51,52,53,54,55]])

# Construct `Z_green`, `Z_red`, `Z_orange`, and `Z_cyan`:
Z_orange = Z[0, 3:5]
Z_red = Z[:, 2]
Z_green = Z[2::2, ::2]
Z_cyan = Z[4:, 4:]
```

In [20]:
Grade cell: `check_Z` Score: 5.0 / 5.0 (Top)

```
# Test cell: `check_Z`

print("==> Z:\n", Z)
assert (Z == np.array([np.arange(0, 6),
                       np.arange(10, 16),
```

```
                        np.arange(20, 26),
                        np.arange(30, 36),
                        np.arange(40, 46),
                        np.arange(50, 56)])).all()

print("\n==> Orange slice:\n", Z_orange)
assert (Z_orange == np.array ([3, 4])).all()

print("\n==> Red slice:\n", Z_red)
assert (Z_red == np.array ([2, 12, 22, 32, 42, 52])).all()

print("\n==> Cyan slice:\n", Z_cyan)
assert (Z_cyan == np.array ([[44, 45], [54, 55]])).all()

print("\n==> Green slice:\n", Z_green)
assert (Z_green == np.array ([[20, 22, 24], [40, 42, 44]])).all()

print("\n(Passed!)")
```

```
==> Z:
 [[ 0  1  2  3  4  5]
 [10 11 12 13 14 15]
 [20 21 22 23 24 25]
 [30 31 32 33 34 35]
 [40 41 42 43 44 45]
 [50 51 52 53 54 55]]

==> Orange slice:
 [3 4]

==> Red slice:
 [ 2 12 22 32 42 52]

==> Cyan slice:
 [[44 45]
 [54 55]]

==> Green slice:
 [[20 22 24]
 [40 42 44]]

(Passed!)
```

Slices are views

To help save memory, when you slice a Numpy array, you are actually creating a *view* into that array. That means modifications through the view will modify the original array.

In [21]:
```
print("==> Recall C: %s" % str(C.shape))
print(C)
```

```
==> Recall C: (2, 3, 4)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

In [22]:
```
C_view = C[1, 0::2, 1::2] # Question: What does this produce?
print ("==> C_view: %s" % str (C_view.shape))
print (C_view)
```

```
==> C_view: (2, 2)
[[13 15]
 [21 23]]
```

In [23]:
```
C_view[:, :] = -C_view[::-1, ::-1] # Question: What does this do?
print (C_view)
```

```
[[-23 -21]
 [-15 -13]]
```

In [24]:
```
print (C)
```

```
[[[  0   1   2   3]
  [  4   5   6   7]
  [  8   9  10  11]]

 [[ 12 -23  14 -21]
  [ 16  17  18  19]
  [ 20 -15  22 -13]]]
```

You can force a copy using the `.copy()` method:

In [25]:
```
C_copy = C[1, 0::2, 1::2].copy ()
C_copy[:, :] = -C_copy[::-1, ::-1]

print ("==> C_view:")
print (C_view)

print ("\n==> C_copy:")
print (C_copy)
```

```
==> C_view:
[[-23 -21]
 [-15 -13]]

==> C_copy:
[[13 15]
 [21 23]]
```

And to check whether two Numpy array variables point to the same object, you can use the `numpy.may_share_memory()` function:

In [26]:
```
print ("C and C_view share memory: %s" % np.may_share_memory (C, C_view))
print ("C and C_copy share memory: %s" % np.may_share_memory (C, C_copy))
```

```
C and C_view share memory: True
C and C_copy share memory: False
```

Indirect addressing

Two other common ways to index a Numpy array are to use a boolean mask or to use a set of integer indices.

In [27]:
```
np.random.seed(3)
x = np.random.randint(0, 20, 15) # 15 random ints in [0, 20)
print(x)
```

```
[10  3  8  0 19 10 11  9 10  6  0 12  7 14 17]
```

In [28]:
```
# Pull out an arbitrary subset of elements
inds = np.array([3, 7, 8, 12])
print(x[inds])
```

```
[ 0  9 10  7]
```

Before looking at how to use a boolean mask for indexing, let's create one.

Exercise 3 (1 point). Given the input array, `x[:]`, above, create an array, `mask_mult_3[:]` such that `mask_mult_3[i]` is true only if `x[i]` is a positive multiple of 3.

In [29]:

Student's answer (Top)

```
mask_mult_3 = (x > 0) & (x % 3 == 0)
```

In [30]:

Grade cell: `mask_mult_3_test` Score: 1.0 / 1.0 (Top)

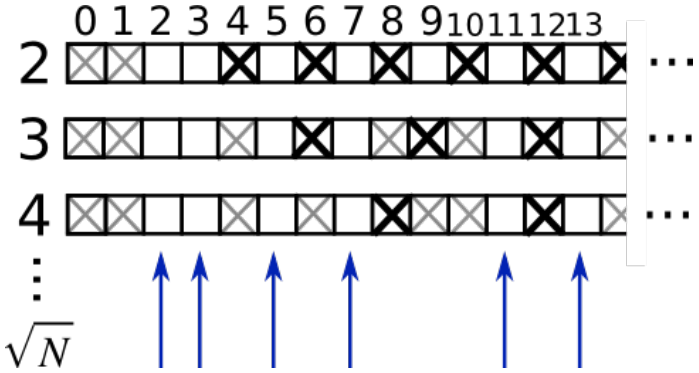
```
# Test cell: `mask_mult_3_test`

print ("x:", x)
print ("mask_mult_3:", mask_mult_3)
print ("==> x[mask_mult_3]:", x[mask_mult_3])

inv_mask_mult_3 = np.invert (mask_mult_3)
assert ((x[mask_mult_3] % 3) == np.zeros (sum (mask_mult_3))).all ()
assert (((x[inv_mask_mult_3] % 3) != np.zeros (sum (inv_mask_mult_3))) | (x[inv_mask_mu
lt_3] == 0)).all ()
```

```
x: [10  3  8  0 19 10 11  9 10  6  0 12  7 14 17]
mask_mult_3:
[False  True False False False False False  True False  True False  True
 False False False]
==> x[mask_mult_3]: [ 3  9  6 12]
```

Exercise 4 (3 points). Complete the prime number sieve algorithm, which is illustrated below.



That is, given a positive integer n, the algorithm iterates from $i \in \{2, 3, 4, \ldots, \lfloor \sqrt{n} \rfloor\}$, repeatedly "crossing out" values that are strict multiples of i. "Crossing out" means maintaining an array of, say, booleans, and setting values that are multiples of i to `False`.

In [31]:

Student's answer (Top)

```
from math import sqrt

def sieve(n):
    """
    Returns the prime number 'sieve' shown above.

    That is, this function returns an array `X[0:n+1]`
    such that `X[i]` is true if and only if `i` is prime.
    """
    is_prime = np.empty(n+1, dtype=bool) # the "sieve"

    # Initial values
    is_prime[0:2] = False # {0, 1} are _not_ considered prime
    is_prime[2:] = True # All other values might be prime

    # Implement the sieving loop
    for k in range(2, int(sqrt(n))):
        is_prime[(2*k):(n+1):k] = False

    return is_prime

# Prints your primes
print("==> Primes through 20:\n", np.nonzero(sieve(20))[0])
```

```
==> Primes through 20:
 [ 2  3  5  7 11 13 17 19]
```

In [32]:

Grade cell: `prime_sieve_test` Score: 3.0 / 3.0 (Top)

```
# Test cell: `prime_sieve_test`

is_prime = sieve(20)
assert len (is_prime) == 21
assert (is_prime == np.array([False, False, True, True, False, True, False, True, False
, False, False, True, False, True, False, False, False, True, False, True, False])).all
()
```

---

part2 (Score: 6.0 / 6.0)

1. Test cell (Score: 2.0 / 2.0)
2. Test cell (Score: 1.0 / 1.0)
3. Test cell (Score: 3.0 / 3.0)

Important note! Before you turn in this lab notebook, make sure everything runs as expected:

- First, **restart the kernel** -- in the menubar, select Kernel→Restart.
- Then **run all cells** -- in the menubar, select Cell→Run All.

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE."

Part 2: Dense matrix storage

This part of the lab is a brief introduction to efficient storage of matrices.

Exercise 0 (ungraded). Import Numpy!

In [1]:

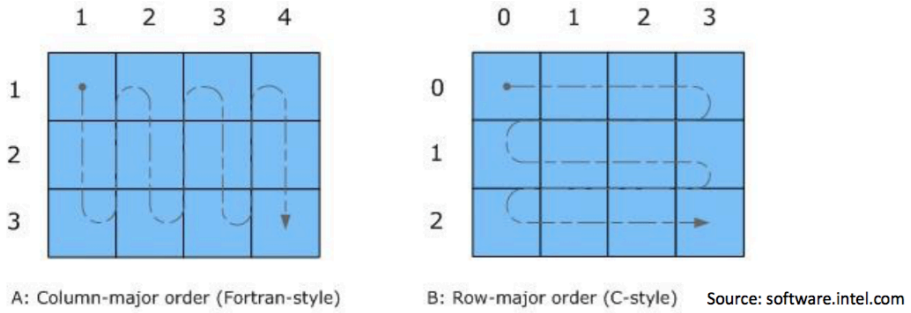
```
import numpy as np
```

Dense matrix storage: Column-major versus row-major layouts

For linear algebra, we will be especially interested in 2-D arrays, which we will use to store matrices. For this common case, there is a subtle performance issue related to how matrices are stored in memory.

By way of background, physical storage---whether it be memory or disk---is basically one big array. And because of how physical storage is implemented, it turns out that it is much faster to access consecutive elements in memory than, say, to jump around randomly.

A matrix is a two-dimensional object. Thus, when it is stored in memory, it must be mapped in some way to the one-dimensional physical array. There are many possible mappings, but the two most common conventions are known as the *column-major* and *row-major* layouts:



A: Column-major order (Fortran-style) B: Row-major order (C-style) Source: software.intel.com

Exercise 1 (2 points). Let A be an $m \times n$ matrix stored in column-major format. Let B be an $m \times n$ matrix stored in row-major format.

Based on the preceding discussion, recall that these objects will be mapped to 1-D arrays of length mn, behind the scenes. Let's call the 1-D array representations $\hat{A}$ and $\hat{B}$. Thus, the (i, j) element of a, a_{ij}, will map to some element $\hat{a_u}$ of $\hat{A}$; similarly, b_{ij} will map to some element $\hat{b_v}$ of $\hat{B}$.

Determine formulae to compute the 1-D index values, u and v, in terms of $\{i, j, m, n\}$. Assume that all indices are 0-based, i.e., $0 \le i \le m-1$, $0 \le j \le n-1$, and $0 \le u, v \le mn-1$.

In [2]:

Student's answer (Top)

```
def linearize_colmajor(i, j, m, n): # calculate `u`
    """
    Returns the linear index for the `(i, j)` entry of
    an `m`-by-`n` matrix stored in column-major order.
    """
    return i + j*m
```

In [3]:

Student's answer (Top)

```
def linearize_rowmajor(i, j, m, n): # calculate `v`
    """
    Returns the linear index for the `(i, j)` entry of
    an `m`-by-`n` matrix stored in row-major order.
    """
    return i*n + j
```

In [4]:

Grade cell: `calc_uv_test` Score: 2.0 / 2.0 (Top)

```
# Test cell: `calc_uv_test`

# Quick check (not exhaustive):
assert linearize_colmajor(7, 4, 10, 20) == 47
assert linearize_rowmajor(7, 4, 10, 20) == 144

assert linearize_colmajor(10, 8, 86, 26) == 698
assert linearize_rowmajor(10, 8, 86, 26) == 268

assert linearize_colmajor(8, 34, 17, 40) == 586
assert linearize_rowmajor(8, 34, 17, 40) == 354

assert linearize_colmajor(32, 48, 37, 55) == 1808
assert linearize_rowmajor(32, 48, 37, 55) == 1808

assert linearize_colmajor(24, 33, 57, 87) == 1905
assert linearize_rowmajor(24, 33, 57, 87) == 2121

assert linearize_colmajor(10, 3, 19, 74) == 67
assert linearize_rowmajor(10, 3, 19, 74) == 743

print ("(Passed.)")
```

```
(Passed.)
```

Requesting a layout in Numpy

In Numpy, you can ask for either layout. The default in Numpy is row-major.

Historically numerical linear algebra libraries were developed assuming column-major layout. This layout happens to be the default when you declare a 2-D array in the Fortran programming language. By contrast, in the C and C++ programming languages, the default convention for a 2-D array is row-major layout. So the Numpy default is the C/C++ convention.

In your programs, you can request either order of Numpy using the `order` parameter. For linear algebra operations (common), we

recommend using the column-major convention.

In either case, here is how you would create column- and row-major matrices.

In [5]:
```
n = 5000
A_colmaj = np.ones((n, n), order='F') # column-major (Fortran convention)
A_rowmaj = np.ones((n, n), order='C') # row-major (C/C++ convention)
```

Exercise 2 (1 point). Given a matrix A, write a function that scales each column, $A(:,j)$ by j. Then compare the speed of applying that function to matrices in row and column major order.

In [6]:
Student's answer (Top)

```
def scale_colwise(A):
    """Given a Numpy matrix `A`, visits each column `A[:, j]`
    and scales it by `j`."""
    assert type(A) is np.ndarray

    n_cols = A.shape[1] # number of columns
    for j in range(n_cols):
        A[:, j] *= j
    return A
```

In [7]:
Grade cell: `scale_colwise_test` Score: 1.0 / 1.0 (Top)

```
# Test (timing) cell: `scale_colwise_test`

# Measure time to scale a row-major input column-wise
%timeit scale_colwise(A_rowmaj)

# Measure time to scale a column-major input column-wise
%timeit scale_colwise(A_colmaj)
```

```
1 loop, best of 3: 606 ms per loop
The slowest run took 11.02 times longer than the fastest. This could mean that an interme
diate result is being cached.
1 loop, best of 3: 31.5 ms per loop
```

Python vs. Numpy example: Matrix-vector multiply

Look at the definition of matrix-vector multiplication from Da Kuang's linear algebra notes (https://www.dropbox.com/s/f410k9fgd7iesdv/kuang-linalg-notes.pdf?dl=0). Let's benchmark a matrix-vector multiply in native Python, and compare that to doing the same operation in Numpy.

First, some setup. (What does this code do?)

In [8]:
```
# Dimensions; you might shrink this value for debugging
n = 2500
```

In [9]:
```
# Generate random values, for use in populating the matrix and vector
from random import gauss

# Native Python, using lists
A_py = [gauss(0, 1) for i in range(n*n)] # Assume: Column-major
x_py = [gauss(0, 1) for i in range(n)]
```

In [10]:
```
# Convert values into Numpy arrays in column-major order
A_np = np.reshape(A_py, (n, n), order='F')
x_np = np.reshape(x_py, (n, 1), order='F')
```

In [11]:
```
# Here is how you do a "matvec" in Numpy:
%timeit A_np.dot(x_np)
```

```
100 loops, best of 3: 3.06 ms per loop
```

Exercise 3 (3 points). Implement a matrix-vector product that operates on native Python lists. Assume the 1-D **column-major** storage of the matrix.

In [12]: Student's answer (Top)

```
def matvec_py(m, n, A, x):
    """
    Native Python-based matrix-vector multiply, using lists.
    The dimensions of the matrix A are m-by-n, and x is a
    vector of length n.
    """
    assert type(A) is list and all([type(aij) is float for aij in A])
    assert type(x) is list
    assert len(x) >= n
    assert len(A) >= (m*n)

    y = [0.] * m
    for j in range(n):
        for i in range(m):
            y[i] += A[i + j*m] * x[j]
    return y
```

In [13]: Grade cell: matvec_py_test Score: 3.0 / 3.0 (Top)

```
# Test cell: `matvec_py_test`

# Estimate a bound on the difference between these two
EPS = np.finfo (float).eps # "machine epsilon"
CONST = 10.0 # Some constant for the error bound
dy_max = CONST * n * EPS

print ("""==> Error bound estimate:
         C*n*eps
         == %g*%g*%g
         == %g
""" % (CONST, n, EPS, dy_max))

# Run the Numpy version and your code
y_np = A_np.dot (x_np)
y_py = matvec_py (n, n, A_py, x_py)

# Compute the difference between these
dy = y_np - np.reshape (y_py, (n, 1), order='F')
dy_norm = np.linalg.norm (dy, ord=np.inf)

# Summarize the results
from IPython.display import display, Math

comparison = "\leq" if dy_norm <= dy_max else "\gt"
display (Math (
        r'||y_{\textrm{np}} - y_{\textrm{py}}||_{\infty}'
        r' = \textrm{%g} %s \textrm{%g}\ (\textrm{estimated bound})'
        % (dy_norm, comparison, dy_max)
    ))

if n <= 4: # Debug: Print all data for small inputs
    print ("@A_np:\n", A_np)
    print ("@x_np:\n", x_np)
    print ("@y_np:\n", y_np)
    print ("@A_py:\n", A_py)
    print ("@x_py:\n", x_np)
    print ("@y_py:\n", y_py)
    print ("@dy:\n", dy)

# Trigger an error on likely failure
assert dy_norm <= dy_max
print("\n(Passed!)")
```

```
==> Error bound estimate:
        C*n*eps
        == 10*2500*2.22045e-16
        == 5.55112e-12
```

$||y_{\text{np}} - y_{\text{py}}||_\infty = 1.13687\text{e-}13 \leq 5.55112\text{e-}12$ (estimated bound)

```
(Passed!)
```

In [14]:
```
%timeit matvec_py (n, n, A_py, x_py)
```

```
1 loop, best of 3: 2.95 s per loop
```

Fin! If you've reached this point and everything executed without error, you can submit this part and move on to the next one.

---

part3 (Score: 24.0 / 24.0)

1. Test cell (Score: 0.0 / 0.0)
2. Test cell (Score: 0.0 / 0.0)
3. Test cell (Score: 3.0 / 3.0)
4. Test cell (Score: 3.0 / 3.0)
5. Test cell (Score: 3.0 / 3.0)
6. Test cell (Score: 3.0 / 3.0)
7. Test cell (Score: 3.0 / 3.0)
8. Test cell (Score: 3.0 / 3.0)
9. Test cell (Score: 3.0 / 3.0)
10. Test cell (Score: 3.0 / 3.0)

Important note! Before you turn in this lab notebook, make sure everything runs as expected:

- First, **restart the kernel** -- in the menubar, select Kernel→Restart.
- Then **run all cells** -- in the menubar, select Cell→Run All.

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE."

Part 3: Sparse matrix storage

This part is about sparse matrix storage in Numpy/Scipy. Start by running the following code cell to get some of the key modules you'll need.

In [1]:
```
import numpy as np
import pandas as pd
from random import sample # Used to generate a random sample
from IPython.display import display
```

Sample data

For this part, you'll need to download the dataset below. It's a list of pairs of strings. The strings, it turns out, correspond to anonymized Yelp! user IDs; a pair (a, b) exists if user a is friends on Yelp! with user b.

Exercise 0 (ungraded). Verify that you can obtain the dataset and take a peek by running the two code cells that follow.

In [2]: Grade cell: download Score: 0.0 / 0.0 (Top)

In [ ]: Grade cell: download Score: 0.0 / 0.0 (Top)

```
import requests
import os
import hashlib
import io

def is_vocareum():
    return os.path.exists('.voc')

file = 'UserEdges-1M.csv'
if is_vocareum():
    url = 'https://cse6040.gatech.edu/datasets/{}'.format(file)
    local_filename = '../resource/lib/publicdata/{}'.format(file)
else:
    local_filename = file
    url = 'https://github.com/cse6040/labs-fa17/raw/master/datasets/{}'.format(file)

if os.path.exists(local_filename):
    print("[{}]\n==> '{}' is already available.".format(url, local_filename))
else:
    print("[{}] Downloading...".format(url))
    r = requests.get(url)
    with open(local_filename, 'w', encoding=r.encoding) as f:
        f.write(r.text)

checksum = '4668034bbcd2fa120915ea2d15eafa8d'
with io.open(local_filename, 'r', encoding='utf-8', errors='replace') as f:
    body = f.read()
    body_checksum = hashlib.md5(body.encode('utf-8')).hexdigest()
    assert body_checksum == checksum, \
            "Downloaded file '{}' has incorrect checksum: '{}' instead of '{}'".format(
local_filename,

body_checksum,

checksum)
    print("==> Checksum test passes: {}".format(checksum))

print("==> '{}' is ready!\n".format(local_filename))
```

```
[https://github.com/cse6040/labs-fa17/raw/master/datasets/UserEdges-1M.csv]
==> 'UserEdges-1M.csv' is already available.

==> Checksum test passes: 4668034bbcd2fa120915ea2d15eafa8d
==> 'UserEdges-1M.csv' is ready!
```

In [3]: Grade cell: peek Score: 0.0 / 0.0 (Top)

```
# Peek at the data:
edges_raw = pd.read_csv(local_filename)
display(edges_raw.head ())
print("...\n`edges_raw` has {} entries.".format(len(edges_raw)))
```

	Source	Target
0	18kPq7GPye-YQ3LyKyAZPw	rpOyqD_893cqmDAtJLbdog
1	18kPq7GPye-YQ3LyKyAZPw	4U9kSBLuBDU391x6bxU-YA
2	18kPq7GPye-YQ3LyKyAZPw	fHtTaujcyKvXglE33Z5yIw
3	18kPq7GPye-YQ3LyKyAZPw	8J4IIYcqBlFch8T90N923A
4	18kPq7GPye-YQ3LyKyAZPw	wy6l_zUo7SN0qrvNRWgySw

```
...
`edges_raw` has 1000000 entries.
```

Evidently, this dataframe has one million entries.

Exercise 1 (ungraded). Explain what the following code cell does.

In [4]:
```
edges_raw_trans = pd.DataFrame({'Source': edges_raw['Target'],
                                'Target': edges_raw['Source']})
edges_raw_symm = pd.concat([edges_raw, edges_raw_trans])
edges = edges_raw_symm.drop_duplicates()

V_names = set(edges['Source'])
V_names.update(set(edges['Target']))

num_edges = len(edges)
num_verts = len(V_names)
print("==> |V| == {}, |E| == {}".format(num_verts, num_edges))
```

```
==> |V| == 107456, |E| == 882640
```

Answer. Give this question some thought before peeking at our suggested answer, which follows.

Recall that the input dataframe, `edges_raw`, has a row (a, b) if a and b are friends. But here is what is unclear at the outset: if (a, b) is an entry in this table, is (b, a) also an entry? The code in the above cell effectively figures that out, by computing a dataframe, `edges`, that contains both (a, b) and (b, a), with no additional duplicates, i.e., no copies of (a, b).

It also uses sets to construct a set, `V_names`, that consists of all the names. Evidently, the dataset consists of 107,456 unique names and 441,320 unique pairs, or 882,640 pairs when you "symmetrize" to ensure that both (a, b) and (b, a) appear.

Graphs

One way a computer scientist thinks of this collection of pairs is as a *graph*: https://en.wikipedia.org/wiki/Graph_(discrete_mathematics) (https://en.wikipedia.org/wiki/Graph_(discrete_mathematics))

The names or user IDs are *nodes* or *vertices* of this graph; the pairs are *edges*, or arrows that connect vertices. That's why the final output objects are named `V_names` (for vertex names) and `edges` (for the vertex-to-vertex relationships). The process or calculation to ensure that both (a, b) and (b, a) are contained in `edges` is sometimes referred to as *symmetrizing* the graph: it ensures that if an edge $a \rightarrow b$ exists, then so does $b \rightarrow a$. If that's true for all edges, then the graph is *undirected*. The Wikipedia page linked to above explains these terms with some examples and helpful pictures, so take a moment to review that material before moving on.

We'll also refer to this collection of vertices and edges as the *connectivity graph*.

Sparse matrix storage: Baseline methods

Let's start by reminding ourselves how our previous method for storing sparse matrices, based on nested default dictionaries, works and performs.

In [5]:
```
def sparse_matrix(base_type=float):
    """Returns a sparse matrix using nested default dictionaries."""
    from collections import defaultdict
    return defaultdict(lambda: defaultdict (base_type))

def dense_vector(init, base_type=float):
    """
    Returns a dense vector, either of a given length
    and initialized to 0 values or using a given list
    of initial values.
    """
    # Case 1: `init` is a list of initial values for the vector entries
    if type(init) is list:
        initial_values = init
        return [base_type(x) for x in initial_values]

    # Else, case 2: `init` is a vector length.
    assert type(init) is int
    return [base_type(0)] * init
```

Exercise 2 (3 points). Implement a function to compute $y \leftarrow Ax$. Assume that the keys of the sparse matrix data structure are integers in the

interval $[0, s)$ where s is the number of rows or columns as appropriate.

In [6]:

Student's answer (Top)

```
def spmv(A, x, num_rows=None):
    if num_rows is None:
        num_rows = max(A.keys()) + 1
    y = dense_vector(num_rows)

    # Recall: y = A*x is, conceptually,
    # for all i, y[i] == sum over all j of (A[i, j] * x[j])

    for i, row_i in A.items():
        s = 0.
        for j, a_ij in row_i.items():
            s += a_ij * x[j]
        y[i] = s

    return y
```

In [7]:

Grade cell: `spmv_baseline_test` Score: 3.0 / 3.0 (Top)

```
# Test cell: `spmv_baseline_test`

#   / 0.   -2.5   1.2 \   / 1. \   / -1.4 \
#   | 0.1   1.    0.  | = | 2. | = |  2.1 |
#   \ 6.   -1.    0.  /   \ 3. /   \  4.0 /

A = sparse_matrix ()
A[0][1] = -2.5
A[0][2] = 1.2
A[1][0] = 0.1
A[1][1] = 1.
A[2][0] = 6.
A[2][1] = -1.

x = dense_vector ([1, 2, 3])
y0 = dense_vector ([-1.4, 2.1, 4.0])


# Try your code:
y = spmv(A, x)

max_abs_residual = max([abs(a-b) for a, b in zip(y, y0)])

print ("==> A:", A)
print ("==> x:", x)
print ("==> True solution, y0:", y0)
print ("==> Your solution, y:", y)
print ("==> Residual (infinity norm):", max_abs_residual)
assert max_abs_residual <= 1e-14

print ("\n(Passed.)")
```

```
==> A: defaultdict(<function sparse_matrix.<locals>.<lambda> at 0x10761fbf8>, {0: default
dict(<class 'float'>, {1: -2.5, 2: 1.2}), 1: defaultdict(<class 'float'>, {0: 0.1, 1: 1.0
}), 2: defaultdict(<class 'float'>, {0: 6.0, 1: -1.0})})
==> x: [1.0, 2.0, 3.0]
==> True solution, y0: [-1.4, 2.1, 4.0]
==> Your solution, y: [-1.4000000000000004, 2.1, 4.0]
==> Residual (infinity norm): 4.440892098500626e-16

(Passed.)
```

Next, let's convert the `edges` input into a sparse matrix representing its connectivity graph. To do so, we'll first want to map names to integers.

In [8]:

```
id2name = {} # id2name[id] == name
name2id = {} # name2id[name] == id
```

```
name2id = {} # name2id[name] == id

for k, v in enumerate (V_names):
    # for debugging
    if k <= 5: print ("Name %s -> Vertex id %d" % (v, k))
    if k == 6: print ("...")

    id2name[k] = v
    name2id[v] = k
```

```
Name CLS4cITfvt8ex0ckFrl7Eg -> Vertex id 0
Name tQcsgxvBnL6FWgiIuCSbSg -> Vertex id 1
Name i-XxQeVD0wH0_chARIMQXw -> Vertex id 2
Name HKnZ2i_fLupIiT05QDAaYw -> Vertex id 3
Name quT3zHVSdqxS9DF5MKNTLA -> Vertex id 4
Name _4vcsFR5Uy1RdgvRILg23Q -> Vertex id 5
...
```

Exercise 3 (3 points). Given `id2name` and `name2id` as computed above, convert `edges` into a sparse matrix, `G`, where there is an entry `G[s][t] == 1.0` wherever an edge `(s, t)` exists.

In [9]:

Student's answer (Top)

```
G = sparse_matrix()

for i in range(len(edges)): # edges is the table above
    s = edges['Source'].iloc[i]
    t = edges['Target'].iloc[i]
    s_id = name2id[s]
    t_id = name2id[t]
    G[s_id][t_id] = 1.0
```

In [10]:

Grade cell: `edges2spmat1_test` Score: 3.0 / 3.0 (Top)

```
# Test cell: `edges2spmat1_test`

G_rows_nnz = [len(row_i) for row_i in G.values()]
print ("G has {} vertices and {} edges.".format(len(G.keys()), sum(G_rows_nnz)))

assert len(G.keys()) == num_verts
assert sum(G_rows_nnz) == num_edges

# Check a random sample
for k in sample(range(num_edges), 1000):
    i = name2id[edges['Source'].iloc[k]]
    j = name2id[edges['Target'].iloc[k]]
    assert i in G
    assert j in G[i]
    assert G[i][j] == 1.0

print ("\n(Passed.)")
```

```
G has 107456 vertices and 882640 edges.

(Passed.)
```

Exercise 4 (3 points). In the above, we asked you to construct `G` using integer keys. However, since we are, after all, using default dictionaries, we could also use the vertex *names* as keys. Construct a new sparse matrix, `H`, which uses the vertex names as keys instead of integers.

In [11]:

Student's answer (Top)

```
H = sparse_matrix()

for i in range(len(edges)): # edges is the table above
    s = edges['Source'].iloc[i]
    t = edges['Target'].iloc[i]
```

```
    H[s][t] = 1.0
```

In [12]:

Grade cell: `create_H_test` Score: 3.0 / 3.0 (Top)

```
# Test cell: `create_H_test`

H_rows_nnz = [len(h) for h in H.values()]
print("`H` has {} vertices and {} edges.".format(len(H.keys()), sum(H_rows_nnz)))

assert len(H.keys()) == num_verts
assert sum(H_rows_nnz) == num_edges

# Check a random sample
for i in sample(G.keys(), 100):
    i_name = id2name[i]
    assert i_name in H
    assert len(G[i]) == len(H[i_name])

print ("\n(Passed.)")
```

```
`H` has 107456 vertices and 882640 edges.

(Passed.)
```

Exercise 5 (3 points). Implement a sparse matrix-vector multiply for matrices with named keys. In this case, it will be convenient to have vectors that also have named keys; assume we use dictionaries to hold these vectors as suggested in the code skeleton, below.

In [13]:

Student's answer (Top)

```
def vector_keyed(keys=None, values=0, base_type=float):
    if keys is not None:
        if type(values) is not list:
            values = [base_type(values)] * len(keys)
        else:
            values = [base_type(v) for v in values]
        x = dict(zip(keys, values))
    else:
        x = {}
    return x

def spmv_keyed(A, x):
    """Performs a aparse matrix-vector multiply for keyed matrices and vectors."""
    assert type(x) is dict

    y = vector_keyed(keys=x.keys(), values=0.0)
    for i, A_i in A.items():
        for j, a_ij in A_i.items():
            y[i] += a_ij * x[j]
    return y
```

In [14]:

Grade cell: `spmv_keyed_test` Score: 3.0 / 3.0 (Top)

```
# Test cell: `spmv_keyed_test`

#   'row':  / 0.   -2.5   1.2 \   / 1. \   / -1.4 \
#  'your':  | 0.1   1.    0.  | = | 2. | = |  2.1 |
#  'boat':  \ 6.   -1.    0.  /   \ 3. /   \  4.0 /

KEYS = ['row', 'your', 'boat']

A_keyed = sparse_matrix ()
A_keyed['row']['your'] = -2.5
A_keyed['row']['boat'] = 1.2
A_keyed['your']['row'] = 0.1
A_keyed['your']['your'] = 1.
A_keyed['boat']['row'] = 6.
A_keyed['boat']['your'] = -1.
```

```
x_keyed = vector_keyed (KEYS, [1, 2, 3])
y0_keyed = vector_keyed (KEYS, [-1.4, 2.1, 4.0])


# Try your code:
y_keyed = spmv_keyed (A_keyed, x_keyed)

# Measure the residual:
residuals = [(y_keyed[k] - y0_keyed[k]) for k in KEYS]
max_abs_residual = max ([abs (r) for r in residuals])

print ("==> A_keyed:", A_keyed)
print ("==> x_keyed:", x_keyed)
print ("==> True solution, y0_keyed:", y0_keyed)
print ("==> Your solution:", y_keyed)
print ("==> Residual (infinity norm):", max_abs_residual)
assert max_abs_residual <= 1e-14

print ("\n(Passed.)")
```

```
==> A_keyed: defaultdict(<function sparse_matrix.<locals>.<lambda> at 0x10759e730>, {'row
': defaultdict(<class 'float'>, {'your': -2.5, 'boat': 1.2}), 'your': defaultdict(<class
'float'>, {'row': 0.1, 'your': 1.0}), 'boat': defaultdict(<class 'float'>, {'row': 6.0, '
your': -1.0})})
==> x_keyed: {'row': 1.0, 'your': 2.0, 'boat': 3.0}
==> True solution, y0_keyed: {'row': -1.4, 'your': 2.1, 'boat': 4.0}
==> Your solution: {'row': -1.4000000000000004, 'your': 2.1, 'boat': 4.0}
==> Residual (infinity norm): 4.440892098500626e-16

(Passed.)
```

Let's benchmark `spmv()` against `spmv_keyed()` on the full data set. Do they perform differently?

In [15]:
```
x = dense_vector ([1.] * num_verts)
%timeit spmv (G, x)

x_keyed = vector_keyed (keys=[v for v in V_names], values=1.)
%timeit spmv_keyed (H, x_keyed)
```

```
1 loop, best of 3: 422 ms per loop
```

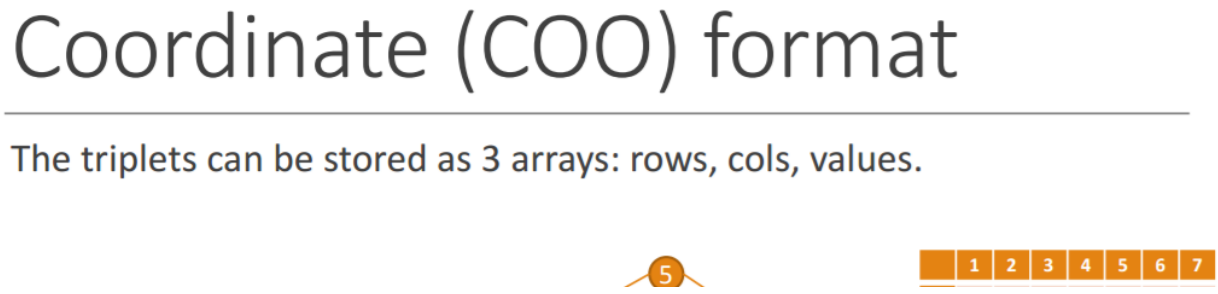
```
1 loop, best of 3: 729 ms per loop
```

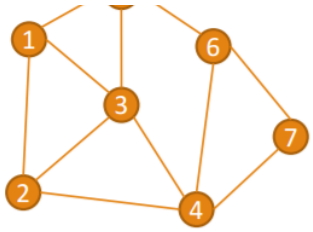
Alternative formats:

Take a look at the following slides: link (https://www.dropbox.com/s/4fwq21dy60g4w4u/cse6040-matrix-storage-notes.pdf?dl=0). These slides cover the basics of two list-based sparse matrix formats known as *coordinate format* (COO) and *compressed sparse row* (CSR). We will also discuss them briefly below.

Coordinate Format (COO)

In this format we store three lists, one each for rows, columns and the elements of the matrix. Look at the below picture to understand how these lists are formed.





1		1	1		1		
2	1		1	1			
3	1	1		1	1		
4		1	1			1	1
5	1		1			1	
6				1	1		1
7				1		1	

rows = [0, 0, 0, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6]

cols = [1, 2, 4, 0, 2, 3, 0, 1, 3, 4, 1, 2, 5, 6, 0, 2, 5, 3, 4, 6, 3, 5]

values = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Note: 0-based arrays

Exercise 6 (3 points). Convert the `edges[:]` data into a coordinate (COO) data structure in native Python using three lists, `coo_rows[:]`, `coo_cols[:]`, and `coo_vals[:]`, to store the row indices, column indices, and matrix values, respectively. Use integer indices and set all values to 1.

In [16]:

Student's answer (Top)

```
coo_rows = [name2id[s] for s in edges['Source']]
coo_cols = [name2id[t] for t in edges['Target']]
coo_vals = [1.0]*len(edges)
```

In [17]:

Grade cell: `create_coo_test` Score: 3.0 / 3.0 (Top)

```
# Test cell: `create_coo_test`

assert len (coo_rows) == num_edges
assert len (coo_cols) == num_edges
assert len (coo_vals) == num_edges
assert all ([v == 1. for v in coo_vals])

# Randomly check a bunch of values
coo_zip = zip (coo_rows, coo_cols, coo_vals)
for i, j, a_ij in sample (list (coo_zip), 1000):
    assert (i in G) and j in G[i]

print ("\n(Passed.)")
```

```
(Passed.)
```

Exercise 7 (3 points). Implement a sparse matrix-vector multiply routine for COO implementation.

In [18]:

Student's answer (Top)

```
def spmv_coo(R, C, V, x, num_rows=None):
    """
    Returns y = A*x, where A has 'm' rows and is stored in
    COO format by the array triples, (R, C, V).
    """
    assert type(x) is list
    assert type(R) is list
    assert type(C) is list
    assert type(V) is list
    assert len(R) == len(C) == len(V)
```

```
    if num_rows is None:
        num_rows = max(R) + 1

    y = dense_vector(num_rows)
    for i, j, a_ij in zip(R, C, V):
        y[i] += a_ij * x[j]
    return y
```

In [19]:

Grade cell: `spmv_coo_test` Score: 3.0 / 3.0 (Top)

```
# Test cell: `spmv_coo_test`

#   / 0.   -2.5   1.2 \   / 1. \   / -1.4 \
#   | 0.1   1.    0.  | = | 2. | = |  2.1 |
#   \ 6.   -1.    0.  /   \ 3. /   \  4.0 /

A_coo_rows = [0, 0, 1, 1, 2, 2]
A_coo_cols = [1, 2, 0, 1, 0, 1]
A_coo_vals = [-2.5, 1.2, 0.1, 1., 6., -1.]

x = dense_vector([1, 2, 3])
y0 = dense_vector([-1.4, 2.1, 4.0])

# Try your code:
y_coo = spmv_coo(A_coo_rows, A_coo_cols, A_coo_vals, x)

max_abs_residual = max ([abs(a-b) for a, b in zip(y_coo, y0)])

print("==> A_coo:", list(zip(A_coo_rows, A_coo_cols, A_coo_vals)))
print("==> x:", x)
print("==> True solution, y0:", y0)
print("==> Your solution:", y_coo)
print("==> Residual (infinity norm):", max_abs_residual)
assert max_abs_residual <= 1e-15

print("\n(Passed.)")
```

```
==> A_coo: [(0, 1, -2.5), (0, 2, 1.2), (1, 0, 0.1), (1, 1, 1.0), (2, 0, 6.0), (2, 1, -1.0
)]
==> x: [1.0, 2.0, 3.0]
==> True solution, y0: [-1.4, 2.1, 4.0]
==> Your solution: [-1.4000000000000004, 2.1, 4.0]
==> Residual (infinity norm): 4.440892098500626e-16

(Passed.)
```

Let's see how fast this is...

In [20]:

```
x = dense_vector([1.] * num_verts)
%timeit spmv_coo(coo_rows, coo_cols, coo_vals, x)
```

```
1 loop, best of 3: 545 ms per loop
```

Compressed Sparse Row Format

This is similar to the COO format excpet that it is much more compact and takes up less storage. Look at the picture below to understand more about this representation

cols = [1, 2, 4, 0, 2, 3, 0, 1, 3, 4, 1, 2, 3, 0, 0, 2, 3, 3, 4, 0, 3, 3]

values = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

The COO format needs 3nnz elements to store the matrix. Can we do better?

When the nonzeros are stored row by row, we can compress the above storage:

rowptr = [0, 3, 6, 10, 14, 17, 20, 22] Row pointer

colind = [1, 2, 4, 0, 2, 3, 0, 1, 3, 4, 1, 2, 5, 6, 0, 2, 5, 3, 4, 6, 3, 5] Column index

values = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] Values

This CSR format needs 2nnz+n elements to store the matrix.

Exercise 8 (3 points). Now create a CSR data structure, again using native Python lists. Name your output CSR lists `csr_ptrs`, `csr_inds`, and `csr_vals`.

It's easiest to start with the COO representation. We've given you some starter code.

In [21]: Student's answer (Top)

```
from operator import itemgetter
C = sorted(zip(coo_rows, coo_cols, coo_vals), key=itemgetter(0))
nnz = len(C)
assert nnz >= 1

assert (C[-1][0] + 1) == num_verts  # Why?

csr_inds = [j for _, j, _ in C]
csr_vals = [a_ij for _, _, a_ij in C]

# Your task: Compute `csr_ptrs`
### BEGIN SOLUTION
C_rows = [i for i, _, _ in C] # sorted row indices
csr_ptrs = [0] * (num_verts + 1)
cur_row = -1 # current row (initially an invalid index, -1)
for k, row in enumerate(C_rows):
    if row != cur_row: # Found a new row index
        csr_ptrs[row] = k # Record location
        cur_row = row

# Some rows might have been empty. The
# next step fills those in.
from itertools import accumulate
csr_ptrs = list(accumulate(csr_ptrs, max))

# Lastly, the boundary element:
csr_ptrs[-1] = nnz
### END SOLUTION
```

In [22]: Grade cell: `create_csr_test` Score: 3.0 / 3.0 (Top)

```
# Test cell: `create_csr_test`

assert type(csr_ptrs) is list, "`csr_ptrs` is not a list."
assert type(csr_inds) is list, "`csr_inds` is not a list."
assert type(csr_vals) is list, "`csr_vals` is not a list."

assert len(csr_ptrs) == (num_verts + 1), "`csr_ptrs` has {} values instead of {}".forma
t(len(csr_ptrs), num_verts)
```

```
assert len(csr_inds) == num_edges, "`csr_inds` has {} values instead of {}".format(len(
csr_inds), num_edges)
assert len(csr_vals) == num_edges, "`csr_vals` has {} values instead of {}".format(len(
csr_vals), num_edges)
assert csr_ptrs[num_verts] == num_edges, "`csr_ptrs[{}]` == {} instead of {}".format(nu
m_verts, csr_ptrs[num_verts], num_edges)

# Check some random entries
for i in sample(range(num_verts), 10000):
    assert i in G
    a, b = csr_ptrs[i], csr_ptrs[i+1]
    msg_prefix = "Row {} should have these nonzeros: {}".format(i, G[i])
    assert (b-a) == len(G[i]), "{}, which is {} nonzeros; instead, it has just {}.".for
mat(msg_prefix, len(G[i]), b-a)
    assert all([(j in G[i]) for j in csr_inds[a:b]]), "{}. However, it may have missing
or incorrect column indices: csr_inds[{}:{}] == {}".format(msg_prefix, a, b, csr_inds[a
:b])
    assert all([(j in csr_inds[a:b] for j in G[i].keys())]), "{}. However, it may have
missing or incorrect column indices: csr_inds[{}:{}] == {}".format(msg_prefix, a, b, cs
r_inds[a:b])

print ("\n(Passed.)")
```

```
(Passed.)
```

Exercise 9 (3 points). Now implement a CSR-based sparse matrix-vector multiply.

In [23]:

Student's answer (Top)

```
def spmv_csr(ptr, ind, val, x, num_rows=None):
    assert type(ptr) == list
    assert type(ind) == list
    assert type(val) == list
    assert type(x) == list
    if num_rows is None: num_rows = len(ptr) - 1
    assert len(ptr) >= (num_rows+1)  # Why?
    assert len(ind) >= ptr[num_rows]  # Why?
    assert len(val) >= ptr[num_rows]  # Why?

    y = dense_vector(num_rows)
    for i in range(num_rows):
        for k in range(ptr[i], ptr[i+1]):
            y[i] += val[k] * x[ind[k]]
    return y
```

In [24]:

Grade cell: `spmv_csr_test` Score: 3.0 / 3.0 (Top)

```
# Test cell: `spmv_csr_test`

#   / 0.   -2.5   1.2 \   / 1. \   / -1.4 \
#   | 0.1   1.    0.  | = | 2. | = |  2.1 |
#   \ 6.   -1.    0.  /   \ 3. /   \  4.0 /

A_csr_ptrs = [ 0,        2,       4,       6]
A_csr_cols = [ 1,   2,   0,   1,  0,   1]
A_csr_vals = [-2.5, 1.2, 0.1, 1., 6., -1.]

x = dense_vector([1, 2, 3])
y0 = dense_vector([-1.4, 2.1, 4.0])

# Try your code:
y_csr = spmv_csr(A_csr_ptrs, A_csr_cols, A_csr_vals, x)

max_abs_residual = max([abs(a-b) for a, b in zip(y_csr, y0)])

print ("==> A_csr_ptrs:", A_csr_ptrs)
print ("==> A_csr_{cols, vals}:", list(zip(A_csr_cols, A_csr_vals)))
print ("==> x:", x)
print ("==> True solution, y0:", y0)
print ("==> Your solution:", y_csr)
```

```
print ("==> Residual (infinity norm):", max_abs_residual)
assert max_abs_residual <= 1e-14

print ("\n(Passed.)")
```

```
==> A_csr_ptrs: [0, 2, 4, 6]
==> A_csr_{cols, vals}: [(1, -2.5), (2, 1.2), (0, 0.1), (1, 1.0), (0, 6.0), (1, -1.0)]
==> x: [1.0, 2.0, 3.0]
==> True solution, y0: [-1.4, 2.1, 4.0]
==> Your solution: [-1.4000000000000004, 2.1, 4.0]
==> Residual (infinity norm): 4.440892098500626e-16

(Passed.)
```

In [25]:
```
x = dense_vector([1.] * num_verts)
%timeit spmv_csr(csr_ptrs, csr_inds, csr_vals, x)
```

```
1 loop, best of 3: 346 ms per loop
```

Using Scipy's implementations

What you should have noticed is that the list-based COO and CSR formats do not really lead to sparse matrix-vector multiply implementations that are much faster than the dictionary-based methods. Let's instead try Scipy's native COO and CSR implementations.

In [26]:
```
import numpy as np
import scipy.sparse as sp

A_coo_sp = sp.coo_matrix((coo_vals, (coo_rows, coo_cols)))
A_csr_sp = A_coo_sp.tocsr() # Alternatively: sp.csr_matrix((val, ind, ptr))
x_sp = np.ones(num_verts)
```