# Introduction to Programming Homework 7

## Due Friday Nov 11

You will turn in your homework **via GitHub**! **Please do Exercise 1 before starting on the other problems.** After Exercise 1, Exercise 2 has the most detailed `git` instructions, so if you choose to do Exercise 3 first, be sure to read the `git` instructions in Exercise 2.

# Exercise 1 (Git and GitHub)

Ask for my help if your had difficulties wit `git` and GitHub.

# Exercise 2 (Conditionally Convergent Series)

Go back to your favorite text editor and create a file called `ccs.py`

- **a.** Recall that if $\{a_i\}_{i\in\mathbb{N}} \in \mathbb{R}$ and

$$\sum_{i=0}^{\infty} a_i \text{ converges but } \sum_{i=0}^{\infty} |a_i| \text{ diverges}$$

then for every $\beta \in \mathbb{R}$ we can find a bijective map $\sigma_\beta : \mathbb{N} \to \mathbb{N}$ such that

$$\beta = \sum_{i=0}^{\infty} a_{\sigma_\beta(i)}$$

Given `beta` and a **generator function** `cond_conv_seq()` write a generator function `rearrange(cond_conv_seq,beta)` that yields $a_{\sigma_\beta(i)}$ in order.

- **b.** Write a function called `test_rearrange(cond_conv_seq, error)` that picks a random float `beta` and returns `True` if after some finite number of steps $N$, $\left|\beta - \sum_{i=0}^{N} a_{\sigma_\beta(i)}\right| < error$ using your generator function from part **a.** Run a few tests using

```
def cond_conv_seq() :
  """ Generates (-1)^(n+1)/n. """
  n = 1.
  sign = 0
  while True :
      yield (-1)**sign/n
      n += 1.
      sign = 1 - sign
```

In [1]:

```
import random

def rearrange(cond_conv_seq,beta) :
```

```
    """ Given a generator cond_conv_seq that yield

    a sequence of conditionally convergent numbers,
    this generator yields a rearearrangement such
    that the rearrangement converges to beta. """
    pos = []
    neg = []
    seq = cond_conv_seq()
    total = 0.
    for a in seq :
        if a > 0. :
            pos.append(a)
        else :
            neg.append(a)
        if total < beta :
            if len(pos) > 0 :
                b = pos.pop(0)
            else :
                continue
        else :
            if len(neg) > 0 :
                b = neg.pop(0)
            else :
                continue
        yield b
        total += b

# a maximum for the abs of the
# random beta to be chosen in the
# test below
beta_abs_max = 5.
max_seq_steps = 10**10

def test_rearrange(cond_conv_seq, error) :
    """ Return True if in some reasonable finite number of
    steps, a rearearrangement of cond_conv_seq begins to
    converge to a randomly chosen float from the interval
    (- beta_abs_max, beta_abs_max). """
    # we don't declare beta_abs_max and max_seq_steps
    # as globals, becase we don't plan to modiy them
    # and if someone updates this code, they can make
    # local versions named the same thing
    beta = random.uniform(-beta_abs_max, beta_abs_max)
    beta_seq = rearrange(cond_conv_seq,beta)
    total = 0.
    count = 0
    print(beta)
    while abs(beta - total) > error :
        if count > max_seq_steps :
            return False
        total += next(beta_seq)
        count += 1
    return True

def alt_harmonic_seq() :
    """ Generates (-1)^(n+1)/n. """
    n = 1.
    sign = 0
```

```
    while True :
        yield (-1)**sign/n
        n += 1.
        sign = 1 - sign
```

In [3]:

```
test_rearrange(alt_harmonic_seq, 0.01)
```

```
-3.0823519164950044
```

Out[3]:

```
True
```

# Exercise 3 (Random Text Files)

Go back to your favorite text editor and create a file called `random_text.py`

- **a.** Recall that a partition of a positive number $n$ is a tuple of numbers $(a_1, \ldots, a_k)$ such that $a_1 \geq a_2 \geq a_k \geq 1$ and $a_1 + \cdots a_k = n$. Write a function `partitions(n)` which returns the list of all partitions of `n` (i.e. returns a list of tuples). If you can, make your function a generator.
- **b.** Write a function called `generate_random_text(file_name)` which will write random text to a file called `file_name` using the following algorithm :
  - pick a random number `num_lines` from $\{20, \ldots 100\}$
  - for each line pick a random number `num_letters` from $\{10, 50\}$
  - pick a random partition `p` of `num_letters` using part **a**
  - for each `i = 0 ... len(p) - 1` generate a random word `w[i]` of length `p[i]`
    - you can use the module `string` the data `string.ascii_letters` for sampling the letters for `w[i]`
    - make sure you choose randomly **with replacement**
    - to create a string out of a list of characters, use `''.join(char_list)`
  - write a the words `w[i]` on a line of the file `file_name` **separated by a space**
  - make sure you have written `num_lines` lines to the file using this algorithm
- **c.** Write a function called `count_capitals(file_name)` which will read a file at the path `file_name` and return a dictionary mapping line numbers (starting with line 1) to the number of capital letters in each line. You can use `str.isupper` to test if a character is uppercase or not.

For example, if `file_name` contains

```
o Agy fId ZGIBwAGM bnSxLFLCcHZpjab
M TiRmMn weyfZVKT wNftXrUrjuLmECV
```

then your code should return

```
{ 1 : 16, 2 : 15 }
```

In [4]:

```
import random
import string
```

```
def partition_gen(n) :
    """ Generates the partitions of a positive
    integer n in reverse lexographic order. """
    assert isinstance(n, int) and n > 0
    part = [n]
    yield tuple(part)
    # last non-unit index
    nu_idx = 0
    while part[0] != 1 :
        # if the previous partition ends with
        # ..., v, 1,1,1,..,1) we replace this tail
        # with ..., v-1, v-1, v-1,..., v-1, r) where
        # r < v-1 and the totals are preserved.
        val = part[nu_idx] - 1
        if val == 1 :
            part[nu_idx] -= 1
            part.append(1)
            nu_idx -= 1
        else :
            total = val + len(part) - nu_idx
            reps, rest = divmod(total, val)
            part[nu_idx:] = reps*[val]
            if rest > 0 :
                part.append(rest)
            nu_idx = len(part) - 1 - (rest == 1)
        yield tuple(part)

def partitions(n) :
    """ Returns the list of partitions of a positive
    integer n in reverse lexographic order. """
    return [ p for p in partition_gen(n) ]

def random_from_reservoir(item_iter) :
    """ Produces a random item from an non-empty iterator
    item_iter without pior knowldege of the number of items.
    This is called reservoir sampling. """
    selected = None
    count = 0
    for item in item_iter :
        if random.random() * count < 1 :
            selected = item
        count += 1
    return selected

def generate_random_text(file_name) :
    """ Writes random words and lines to file_name.
    Attention : overwrites the file if it already exits. """
    num_lines = random.randint(20,100)
    with open(file_name, 'w') as fp :
        for _ in range(num_lines) :
            num_letters = random.randint(10,50)
            part_gen = partition_gen(num_letters)
            random_part = random_from_reservoir(part_gen)
            words = []
            for word_len in random_part :
                chars = [ random.choice(string.ascii_letters)
                          for _ in range(word_len) ]
```

```
                words.append(''.join(chars))

            fp.write(' '.join(words) + '\n')

def count_capitals(file_name) :
    """ Returns a dictionary mapping line numbers to
    number of uppercase letter per line in file_name. """
    upper_count = {}
    line_count = 0
    with open(file_name,'r') as fp :
        for line in fp :
            line_count += 1
            num_upper = sum(map(str.isupper, line))
            upper_count[line_count] = num_upper
    return upper_count
```