# Midterm review

CS5 Green

# Midterm Thursday 11/4

- Logistics
  - Given in class (Shan 2460)
  - 75 min (9:35-10:50)
  - 8 ½ x 11 sheet of handwritten notes permitted
- More on the test
  - 4 problems based on material from class or homework
  - Paper and pencil coding problems
  - Docstrings: will be provided for main functions. You should write short one for any helper functions you add
- How to study

# Major topics

- Main data types (numbers, strings, lists, Booleans, tuples, dictionaries)
- Functions
- Conditionals (if, elif, else)
- Loops (for, while)
- Recursion (including use it or lose it)
- Hmmm
- Phylogenetic trees

# Today: four problems

- Research with Prof Walker (squareWalk)
- Construct
- Finding polymorphic sites
- Isomorphic trees

# Research with Prof Walker

Professor Rand M. Walker has a large empty square office. He starts at the center of his office (aka, "the origin"). The four office walls are `dist` steps to the north, south, east, and west of his starting position, that is at `x = dist, x = -dist, y = dist, and y = -dist.`

While thinking hard about a research problem, Prof. Walker walks around randomly: At each step, he goes either north, south, east, or west one step. If he ever hits a wall (that is, his x-coordinate or y-coordinate becomes either `dist` or `-dist`), he spills his coffee, and the random walk comes to an end.

Write a function called `squareWalk(dist)` that takes a positive integer `dist` as input and simulates Prof. Walker's random walk. Here is what the program should do:

•

The walker starts at x = 0 and y = 0.

•At each step, the program changes just one of x or y by +1 or -1.

•*After* each step, the program should print the current coordinates of the walker.

•When the walker hits a wall, the program should print a message: `Hit a wall in 42 steps.` Of course, the number 42 will vary because of the distance to the walls and the random moves.

You may assume that the line…

```
import random
```

...is already in your file.  In your program, you can use the syntax...

```
random.choice(L)
```

… which returns a random element from the list L.  For example…

`random.choice(["a", "b"])` will return an “a” or a “b” at random.

Some examples of the program in action:
>>> squareWalk(2) ← The walls *in this example* are at x = 2, x = -2, y = 2, y = -2
At location 0 1
At location -1 1
At location -1 2
Hit boundary in 3 steps

Write the function squareWalk below. Typical implementations will be between 10 and 15 lines of code.

```
def squareWalk(dist):
    """Simulates a random walk until we hit a wall at
    x = dist, x = -dist, y = dist, or y = -dist."""
```

```
def squareWalk(dist):
    """Simulates a random walk until we hit a wall at
    x = dist, x = -dist, y = dist, or y = -dist."""

    counter = 0
    x = 0
    y = 0
    while (-dist < x) and (x < dist) and (-dist < y) and (y < dist):
        direction = random.choice(["horizontal", "vertical"])
        move = random.choice([-1, 1])
        if direction == "horizontal":
            x += move
        else:
            y += move
        counter += 1
        print("At location", x, y)
    print("Hit boundary in", counter, "steps")
```

# Construct

A colleague of yours at 42 And Me has devised a crafty scheme for inserting foreign DNA into a bacterium. This method requires her to combine fragments of DNA to a certain exact length. The fragments to combine come from two separate groups. They can only be combined by taking alternately from one group and then from the other. It is possible to use a fragment more than once.

Consider the following example case:

```
aL = [80,120]      # group w/ fragments of length 80, 120
bL = [300,400,700] # group w/ fragments of length 300, 400, 700
```

We want to know, can we combine these fragments into a piece of DNA of length exactly 500, starting with a fragment from `aL`, and alternating so that we never take two fragments in a row from the same group.

The function `construct(target,L1,L2)` returns `True` or `False` to answer this. Below are some example calls:

```
>>> construct(500,aL,bL)
True
```

Here we get a True because the sum of 80 (from aL) plus 300 (from bL) plus 120 (from aL) equals 500.

If, however, we put bL first and aL second we get a False:

```
>>> construct(500,bL,aL)
False
```

There is no way to produce a combined fragment 500 bp long using a fragment from bL first (and alternating thereafter).

Another example:

```
>>> construct(680,bL,aL)
True
```

Here we get True because the combination 300 + 80 + 300 (again) is 680.

And one last example:

```
>>> construct(1180,bL,aL)
True
```

Here we get True because 400 + 80 + 700 is 1180. Note that we are allowed to “skip” fragments; here we did not use the first fragment of bL.

Write the function `construct` below. It will be a **Use-It-Or-Lose-It type recursive function**. Typical implementations will be between 10 and 15 lines of code.

```
def construct(target,L1,L2):
    """Returns True if there is a combination of alternating
    elements from L1 and L2 whose total length adds up to
    target."""
```

```
def construct(target,L1,L2):
    """Returns True if there is a combination of alternating
    elements from L1 and L2 whose total length adds up to
    target."""

    if target == 0:
        # finished combining
        return True
    elif L1 == []:
        # nothing in L1 left so cannot combine further
        return False
    elif L1[0] > target:
        # cannot use first fragment of L1, try next L1 fragment
        return construct(target, L1[1:], L2)
    else:
        # either use first fragment of L1 or not,
        # then alternate to use L2
        useIt = construct(target - L1[0], L2, L1) # rev order
        loseIt = construct(target, L1[1:], L2)
        return useIt or loseIt
```

# Finding polymorphic sites

After your successes at various labs and companies, you've joined forces with the lab of Professor Polly Morfique, a famous geneticist at **S.P.A.M**: the **S**hmorbodian **P**olytechnic **A**cade**M**y . Here's the problem that you're investigating...

Consider a particular gene in a population of organisms. That gene is likely to have many sites where polymorphisms are found, that is, sites in the gene where different characters are found in the population. (A character could be a nucleotide or an amino acid, depending on whether we are looking at DNA or protein sequences.)

Your job is to write a function called `fracPolySites(alleleList)` that
takes as input a list of strings and returns the ratio of sites that are polymorphic (not identical).
Each string in the input list is an allele sampled from a population. You can assume that all of these alleles are the same length).

Consider an example

```
individual 1 AAA
individual 2 TAA
individual 3 AAA

>>> fracPolySites(["AAA", "TAA", "AAA"])
0.3333333333333333
```

Here only the first of the three sites contains a polymorphism, so we get back ⅓.

And another example.

```
individual 1 CAA
individual 2 AAG
individual 3 ATA

>>> fracPolySites(["CAA", "AAG", "ATA"])
1.0
```

In this case, all three sites contain at least one polymorphism, so we get back 1.0.

You may wish to write a helper function to help your `fracPolySites` function or you can write it all in one function. Either way is fine and should result in about 10-15 lines of code total.

```
def fracPolySites(alleleList}:
    '''returns the fraction of polymorphic sites in a list
    of alleles all of which have the same length.'''
```

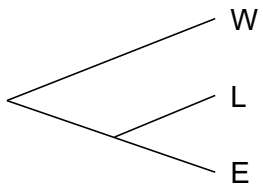
```
def allSame(alleleList, index):
    ''' Returns True if all of the alleles in the alleleList are the same
        at the site at the given index.'''
    character = alleleList[0][index]
    for allele in alleleList:
        if allele[index] != character: return False
    return True

def fracPolySites(alleleList):
    ''' Takes as input a list of alleles all of the same length and returns
        the fraction of sites where polymorphisms are found.'''
    alleleLength = len(alleleList[0])  # all alleles have same length
    numPolySites = 0 # count the number of polymorphic sites
    for index in range(alleleLength):
        if not allSame(alleleList, index):
            numPolySites += 1
    return numPolySites / alleleLength
```

# Isomorphic trees

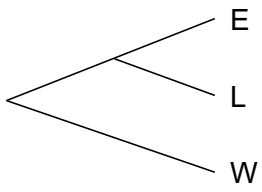
T1

```
("Anc",
   ("W", (), ()),
   ("Anc",
      ("L", (), ()),
      ("E", (), ())
   )
)
```



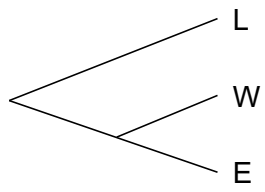

T2

```
("Anc",
   ("Anc",
      ("E", (), ()),
      ("L", (), ())
   ),
   ("W", (), ())
)
```




T3

```
("Anc",
   ("L", (), ()),
   ("Anc",
      ("W", (), ()),
      ("E", (), ())
   )
)
```




Two trees are **isomorphic** if they imply the same underlying phylogenetic relationships

- T1 and T2 are isomorphic
- T3 is not isomorphic either with T1 or T2

```
>>> isomorphic(('A',(),()),('A',(),()))
True
>>> isomorphic(('A',(),()),T1)
False
>>> isomorphic(('A',(),()),('B',(),()))
False
```

```
>>> isomorphic(T1,T2)
True
>>> isomorphic(T1,T3)
False
>>> isomorphic(T2,T3)
False
```

```
def isomorphic(tree1, tree2):
    """Returns boolean indicating if tree1 and tree2 are isomorphic."""
```

```
def isomorphic(tree1, tree2):
    """Returns boolean indicating if tree1 and tree2 are isomorphic."""
    if tree1 == tree2: return True
    elif tree1[1] == () or tree2[1] == (): return False
    else:
        option1 = isomorphic(tree1[1],tree2[1]) and \
                  isomorphic(tree1[2],tree2[2])
        option2 = isomorphic(tree1[1],tree2[2]) and \
                  isomorphic(tree1[2],tree2[1])
        return option1 or option2
```