Matching DNA Sequences

Today we will look at the code structure of the DNA sequence matching problem in PSet 4.

A DNA sample is represented as a sequence of characters such as A, G, C, T in upper case which represent the nucleotides. Given two DNA sequences, we want to see how similar they are. So for a subsequence of length k in one DNA sample, we want to find whether it occurs in the other DNA sample, and if it does, we also want to see whether it occurs at the similar position. We also want to do this for all the possible subsequences of length k.

```
1   # The arguments are, in order: 1) Your getExactSubmatches
2   # function, 2) the filename to which the image should be written,
3   # 3) a tuple giving the width and height of the image, 4) the
4   # filename of sequence A, 5) the filename of sequence B, 6) k, the
5   # subsequence size, and 7) m, the sampling interval for sequence
6   # A.
7   compareSequences(getExactSubmatches, sys.argv[3], (500,500), sys.argv[1],
8                    sys.argv[2], 8, 100)
```

```
1 def compareSequences(getExactSubmatches, imgfile, imgsize, afile, bfile, k, m):
2   a = kfasta.FastaSequence(afile)
3   b = kfasta.FastaSequence(bfile)
4   matches = getExactSubmatches(a, b, k, m)
5   buildComparisonImage(imgfile, imgsize[0], imgsize[1],
6                        kfasta.getSequenceLength(afile),
7                        kfasta.getSequenceLength(bfile), matches)
```

The main function calls `compareSequences`. In `compareSequences`, it creates two `FastaSequences` `a` and `b` which are the inputs to `getExactSubmatches` that you need to implement.

Python Iterator

A `FastaSequence` is an iterator that returns the nucleotide sequence stored in the given FASTA file. In Python, an iterator object is required to support the following two methods, which together form the *iterator protocol*.

- `iterator.__iter__()`: Returns the iterator object itself. This allows iterators to be used with the `for` and `in` statements.
- `iterator.next()`: Returns the next item. If there are no further items, raise the `StopIteration` exception.

Let's see how these two methods are implemented in `FastaSequence`. Because the file is very big, it only reads one line of the file a time and stores it in `self.buf`. This is more memory efficient than reading the entire file into the memory. Calling `next()` returns and removes the first element in `self.buf`, and when `self.buf` is empty, another line is read. It raises an `StopIteration` exception when the end of the file is reached.

```
# An iterator that returns the nucleotide sequence stored in the given FASTA file.
class FastaSequence:
  def __init__(self, filename):
    self.f = open(filename, 'r')
    self.buf = ''
    self.info = self.f.readline()
    self.pos = 0
  def __iter__(self):
    return self
  def next(self):
    while '' == self.buf:
      self.buf = self.f.readline()
      if '' == self.buf:
        self.f.close()
        raise StopIteration
      self.buf = self.buf.strip()
    nextchar = self.buf[0]
    self.buf = self.buf[1:]
    self.pos += 1
    return nextchar
```

Python Generator

The first method you need to implement is `subsequenceHashes(seq, k)` which returns all k-length subsequences and their hashes given a sequence of nucleotides. Note the input `seq` is a `FastaSequence` obtained by reading the input file.

The DNA sequences are tens of millions of nucleotides long. So there are also tens of millios of subsequences. But every time, we are checking one subsequence to see whether it occurs in the other sequence. So we do not need to build a list of all the subsequences in the memory, and that's why it is more memory efficient to implement the function as a generator.

A generator is a function which returns an iterator. It looks like a normal function except that it contains `yield` statements for producing a series a values usable in a for-loop or that can be retrieved one at a time with the `next()` function. Using a `yield` expression in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

The execution starts when one of the generators methods is called, e.g. `next()`. At that time, the execution proceeds to the first yield expression, where it is suspended again, returning the value of expression_list to generators caller. Each yield temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks-up where it left-off (in contrast to functions which start fresh on every invocation).

Let's look at a simple example of a generator that yields the first n non-negative integers.

```
def firstn(n):
  """ A generator that yields items instead of returning a list."""
  num = 0
  while num < n:
```

```
5       yield num
6       num += 1
7
8   print sum(firstn(1000000)) # sum takes an iterable or iterator as an argument.
```

Note that `firstn()` is equivalent to the built-in `xrange()` function, and in Python 3 `range()` is a generator. In Python 2.x, `range()` is non-generator. So `sum(range(1000000))` is more expensive in terms of memory cost because it builds a 1,000,000 element list in memory to find its sum. This is a waste, considering that we use these 1,000,000 elements just to compute the sum.

Rolling Hash

You can use the `RollingHash` class provided. Note that it combines `append()` and `skip()` into one `slide()` method.