Problem 2: DNA Sequence Analysis

This problem is about strings and regular expressions. It has four (4) exercises, numbered 0-3. They are worth a total of ten (10) points.

In [ ]:
```
import re # You'll need this module
```

DNA Sequence Analysis

Your friend is a biologist who is studying a particular DNA sequence. The sequence is a string built from an alphabet of four possible letters, `A`, `G`, `C`, and `T`. Biologists refer to each of these letters a *base*.

Here is an example of a DNA fragment as a string of bases.

In [ ]:
```
dna_seq = 'ATGGCAATAACCCCCCGTTTCTACTTCTAGAGGAGAAAAGTATTGACATGAGCGCTCCCGGCACAAGGGCCA
print("=== Sequence (Number of bases: {}) ===\n\n{}".format(len(dna_seq), dna_seq))
```

```
=== Sequence (Number of bases: 2012) ===

ATGGCAATAACCCCCCGTTTCTACTTCTAGAGGAGAAAAGTATTGACATGAGCGCTCCCGGCACAAGGGCCAAAGAAGTCTCCA
ATTTCTTATTTCCGAATGACATGCGTCTCCTTGCGGGTAAATCACCGACCGCAATTCATAGAAGCCTGGGGGAACAGATAGGTC
TAATTAGCTTAAGAGAGTAAATCCTGGGATCATTCAGTAGTAACCATAAACTTACGCTGGGGCTTCTTCGGCGGATTTTTACAG
TTACCAACCAGGAGATTTGAAGTAAATCAGTTGAGGATTTAGCCGCGCTATCCGGTAATCTCCAAATTAAAACATACCGTTCCA
TGAAGGCTAGAATTACTTACCGGCCTTTTCCATGCCTGCGCTATACCCCCCCACTCTCCCGCTTATCCGTCCGAGCGGAGGCAG
TGCGATCCTCCGTTAAGATATTCTTACGTGTGACGTAGCTATGTATTTTGCAGAGCTGGCGAACGCGTTGAACACTTCACAGAT
GGTAGGGATTCGGGTAAAGGGCGTATAATTGGGGACTAACATAGGCGTAGACTACGATGGCGCCAACTCAATCGCAGCTCGAGC
GCCCTGAATAACGTACTCATCTCAACTCATTCTCGGCAATCTACCGAGCGACTCGATTATCAACGGCTGTCTAGCAGTTCTAAT
CTTTTGCCAGCATCGTAATAGCCTCCAAGAGATTGATGATAGCTATCGGCACAGAACTGAGACGGCGCCGATGGATAGCGGACT
TTCGGTCAACCACAATTCCCCACGGGACAGGTCCTGCGGTGCGCATCACTCTGAATGTACAAGCAACCCAAGTGGGCCGAGCCT
GGACTCAGCTGGTTCCTGCGTGAGCTCGAGACTCGGGATGACAGCTCTTTAAACATAGAGCGGGGGCGTCGAACGGTCGAGAAA
GTCATAGTACCTCGGGTACCAACTTACTCAGGTTATTGCTTGAAGCTGTACTATTTTAGGGGGGGAGCGCTGAAGGTCTCTTCT
TCTCATGACTGAACTCGCGAGGGTCGTGAAGTCGGTTCCTTCAATGGTTAAAAAACAAAGGCTTACTGTGCGCAGAGGAACGCC
CATCTAGCGGCTGGCGTCTTGAATGCTCGGTCCCCTTTGTCATTCCGGATTAATCCATTTCCCTCATTCACGAGCTTGCGAAGT
CTACATTGGTATATGAATGCGACCTAGAAGAGGGCGCTTAAAATTGGCAGTGGTTGATGCTCTAAACTCCATTTGGTTTACTCG
TGCATCACCGCGATAGGCTGACAAAGGTTTAACATTGAATAGCAAGGCACTTCCGGTCTCAATGAACGGCCGGGAAAGGTACGC
GCGCGGTATGGGAGGATCAAGGGGCCAATAGAGAGGCTCCTCTCTCACTCGCTAGGAGGCAAATGTAAAACAATGGTTACTGCA
TCGATACATAAAACATGTCCATCGGTTGCCCAAAGTGTTAAGTGTCTATCACCCCTAGGGCCGTTTCCCGCATATAAACGCCAG
GTTGTATCCGCATTTGATGCTACCGTGGATGAGTCTGCGTCGAGCGCGCCGCACGAATGTTGCAATGTATTGCATGAGTAGGGT
TGACTAAGAGCCGTTAGATGCGTCGCTGTACTAATAGTTGTCGACAGACCGTCGAGATTAGAAAATGGTACCAGCATTTTCGGA
GGTTCTCTAACTAGTATGGATTGCGGTGTCTTCACTGTGCTGCGGCTACCCATCGCCTGAAATCCAGCTGGTGTCAAGCCATCC
CCTCTCCGGGACGCCGCATGTAGTGAAACATATACGTTGCACGGGTTCACCGCGGTCCGTTCTGAGTCGACCAAGGACACAATC
GAGCTCCGATCCGTACCCTCGACAAACTTGTACCCGACCCCCGGAGCTTGCCAGCTCCTCGGGTATCATGGAGCCTGTGGTTCA
TCGCGTCCGATATCAAACTTCGTCATGATAAAGTCCCCCCCTCGGGAGTACCAGAGAAGATGACTACTGAGTTGTGCGAT
```

In this problem, you will help your friend analyze this sequence.

Exercise 0 (2 point). Complete the function, `count_bases(s)`. It takes as input a DNA sequence as a string, `s`. It should compute the number of occurrences of each base (i.e.,

`'A'`, `'C'`, `'G'`, and `'T'`) in `s`. It should then return these counts in a dictionary whose keys are the bases.

In [ ]:
```
# make a frequency dictionary
def count_bases(s):
    assert type(s) is str
    assert all([b in ['A', 'C', 'G', 'T'] for b in s])
    base_dict = {}
    for base in s:
        base_dict[base] = base_dict.get(base, 0) + 1
    return base_dict

count_bases(dna_seq)
```

Out[ ]:
```
{'A': 501, 'T': 496, 'G': 508, 'C': 507}
```

In [ ]:
```
# Test cell: `exercise_0_test`

base_counts = count_bases(dna_seq)
print("Your result:", base_counts)

assert type(base_counts) is dict, "`base_counts` is of type `{}`, not `dict`.".form
assert len(base_counts) <= 4, "There can be at most 4 bases."
for b, c in [('A', 501), ('C', 507), ('G', 508), ('T', 496)]:
    assert base_counts[b] == c, "Base '{}' has a count of {} when it should be {}."

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

```
Your result: {'A': 501, 'T': 496, 'G': 508, 'C': 507}

(Passed!)
```

Enzyme "scissors." Your friend is interested in what will happen to the sequence if she uses certain "restriction enzymes" to cut it. The enzymes work by scanning the DNA sequence from left to right for a particular pattern. It then cuts the DNA wherever it finds a match.

A biologist's notation. Your friend does not know about regular expressions. Instead, she uses a special notation that other biologists use to describe base patterns. These are "extra letters" that have a special meaning.

For example, the special letter `N` denotes any base, i.e., any single occurrence of an `A`, `C`, `G`, or `T`. Therefore, when a biologist writes, `ANT`, that means `AAT`, `ACT`, `AGT`, or `ATT`.

Here is the complete set of special letters:

- `R`: Either `G` or `A`
- `Y`: Either `T` or `C`
- `K`: Either `G` or `T`
- `M`: Either `A` or `C`
- `S`: Either `G` or `C`
- `W`: Either `A` or `T`

- `B` : Anything but `A` (i.e., `G` , `T` , or `C`)
- `D` : Anything but `C`
- `H` : Anything but `G`
- `V` : Anything but `T`
- `N` : Anything, i.e., `A` , `C` , `G` , or `T`

Exercise 1 (4 points). Given a string in the biologist's notation, complete the function `bio_to_regex(pattern_bio)` so that it returns an equivalent pattern in Python's regular expression language.

If your function is correct, then the following code would also work:

```
assert re.search(bio_to_regex('ANT'), 'AGATTA') is not None
```

That's because `ANT` matches `ATT` , which is contained in `AGATTA` .

In [ ]:
```
import string
import re

def bio_to_regex(pattern_bio):
    iupac_codes = {
        'R': '[GA]',
        'Y': '[TC]',
        'K': '[GT]',
        'M': '[AC]',
        'S': '[GC]',
        'W': '[AT]',
        'B': '[GTC]',  # Not A
        'D': '[GAT]',  # Not C
        'H': '[ACT]',  # Not G
        'V': '[ACG]',  # Not T
        'N': '[ACGT]',  # Anything
    }
    regex_pattern = ''
    for char in pattern_bio:
        if char in iupac_codes:
            regex_pattern += iupac_codes[char]  # Use the regex pattern for the IUP
        else:
            regex_pattern += char  # For standard nucleotides, just add them to the
    return regex_pattern

# Usage example
pattern = 'ANT'
regex_pattern = bio_to_regex(pattern)
print(regex_pattern)  # Should print: A[ACGT]T
```

```
A[ACGT]T
```

In [ ]:
```
re.search(bio_to_regex('ANT'), 'AGATTA')
```

Out[ ]:
```
<re.Match object; span=(2, 5), match='ATT'>
```

In [ ]:
```
# Test cell: `exercise_1_test_0`
```

```
assert re.search(bio_to_regex('ANT'), 'AGATTA') is not None
assert set(re.findall(bio_to_regex('ANTAAT'), dna_seq)) == {'ATTAAT', 'ACTAAT'}
assert set(re.findall(bio_to_regex('GCRWTG'), dna_seq)) == {'GCGTTG', 'GCAATG'}
assert len(re.findall(bio_to_regex('CDCHA'), dna_seq)) == 18

print("\n(Passed first group of tests!)")
```

```
(Passed first group of tests!)
```

In [ ]:
```
# Test cell: `exercise_1_test_1`
if False:
    for c in {'Y', 'K', 'M', 'S', 'B', 'D', 'V'}:
        from random import sample
        x = ''.join([sample('ACGT', 1)[0] for _ in range(2)])
        y = ''.join([sample('ACGT', 1)[0] for _ in range(2)])
        pattern = '{}{}{}'.format(x, c, y)
        ans = set(re.findall(bio_to_regex(pattern), dna_seq))
        print("assert set(re.findall(bio_to_regex('{}'), dna_seq)) == {}".format(pa

assert set(re.findall(bio_to_regex('GABAT'), dna_seq)) == {'GACAT', 'GAGAT', 'GATAT
assert set(re.findall(bio_to_regex('GAVCA'), dna_seq)) == {'GACCA', 'GAACA'}
assert set(re.findall(bio_to_regex('TGYGG'), dna_seq)) == {'TGTGG', 'TGCGG'}
assert set(re.findall(bio_to_regex('GCKAA'), dna_seq)) == {'GCGAA'}
assert set(re.findall(bio_to_regex('ATSCA'), dna_seq)) == {'ATCCA'}
assert set(re.findall(bio_to_regex('GCMTT'), dna_seq)) == {'GCCTT', 'GCATT'}
assert set(re.findall(bio_to_regex('AGDCC'), dna_seq)) == {'AGTCC', 'AGACC'}

print("\n(Passed second set of tests!)")
```

```
(Passed second set of tests!)
```

Restriction sites. When an enzyme cuts the string, it does it in a certain location with respect to the target pattern. This information is encoded as a *restriction site*.

The way a biologist specifies the restriction site is with a special notation that embeds the cut in the pattern. For example, there is one enzyme that has a restriction site of the form, `ANT|AAT`, where the vertical bar, `'|'`, shows where the enzyme will split the sequence. So, if the input DNA sequence were

```
GCATAGTAATGTATTAATGGC
```

then there would two matches:

```
GCATAGTAATGTATTAATGGC
    ^^^^^^  ^^^^^^
    match!  match!
```

Furthermore, there would be two cuts, since this enzyme splits its pattern in the middle (between `ANT` and `AAT`):

```
GCATAGT|AATGTATT|AATGGC
    ^^^ ^^^  ^^^ ^^^
```

That would result in three fragments: `GCATAGT`, `AATGTATT`, and `AATGGC`.

Exercise 3 (5 points). Complete the function, `sim_cuts(site_pattern, s)`, below. The first argument, `site_pattern`, is the biologist's restriction site pattern, e.g., `ANT|AAT`, where there may be an embedded cut. The second argument, `s`, is the DNA sequence to cut. The function should return the fragments in the sequence order.

For the preceding example,

```
sim_cuts('ANT|AAT', 'GCATAGTAATGTATTAATGGC') == ['GCATAGT', 'AATGTATT',
'AATGGC']
```

> **Note.** There are *two* test cells, below. Both must pass for full credit, but if only one passes, you'll at least get some partial credit.

In [ ]:
```
def sim_cuts(site_pattern, s):
    split_pattern = site_pattern.split('|')
    pattern_bio="".join(split_pattern)
    m_iter = re.finditer(bio_to_regex(pattern_bio), s)
    off_set = len(split_pattern[0])
    inds = [0] + [m.span()[0] + off_set for m in m_iter] + [len(s)]
    m_list = []
    for start, end in zip(inds[:-1], inds[1:]):
        m_list.append(''.join(s[start:end]))
    return m_list


# Test the function with the provided example
result = sim_cuts('ANT|AAT', 'GCATAGTAATGTATTAATGGC')
expected_result = ['GCATAGT', 'AATGTATT', 'AATGGC']
print("Result:", result)
print("Expected:", expected_result)
print("Correct:", result == expected_result)
```

```
Result: ['GCATAGT', 'AATGTATT', 'AATGGC']
Expected: ['GCATAGT', 'AATGTATT', 'AATGGC']
Correct: True
```

In [ ]:
```
# Test cell: `exercise_3_test_0`

def check_sim_cuts(bio_pattern, s, true_cuts):
    print("\nChecking: '{}'...".format(bio_pattern))
    your_cuts = sim_cuts(bio_pattern, s)
    print("   Your result ({} fragments): {}".format(len(your_cuts), your_cuts))
    print("   True result ({}): {}".format(len(true_cuts), true_cuts))
    assert your_cuts == true_cuts, "Did not match!"
    print("   ==> Matched!")

# Check a simple case:
check_sim_cuts('ANT|AAT', 'GCATAGTAATGTATTAATGGC', ['GCATAGT', 'AATGTATT', 'AATGGC'

print("\n(Passed first test of Exercise 3; two more to go in the next cell.)")
```

```
Checking: 'ANT|AAT'...
   Your result (3 fragments): ['GCATAGT', 'AATGTATT', 'AATGGC']
   True result (3): ['GCATAGT', 'AATGTATT', 'AATGGC']
   ==> Matched!

(Passed first test of Exercise 3; two more to go in the next cell.)
```

In [ ]:
```
# Test cell: `exercise_2_test_1`

check_sim_cuts('ANT|AAT', dna_seq, ['ATGGCAATAACCCCCCGTTTCTACTTCTAGAGGAGAAAAGTATTGA
 'AATCCATTTCCCTCATTCACGAGCTTGCGAAGTCTACATTGGTATATGAATGCGACCTAGAAGAGGGCGCTTAAAATTGGC
 'AATAGTTGTCGACAGACCGTCGAGATTAGAAAATGGTACCAGCATTTTCGGAGGTTCTCTAACTAGTATGGATTGCGGTGT
check_sim_cuts('GCRW|TG', dna_seq, ['ATGGCAATAACCCCCCGTTTCTACTTCTAGAGGAGAAAAGTATTGA
 'TGAACACTTCACAGATGGTAGGGATTCGGGTAAAGGGCGTATAATTGGGGACTAACATAGGCGTAGACTACGATGGCGCCA
 'TGTATTGCATGAGTAGGGTTGACTAAGAGCCGTTAGATGCGTCGCTGTACTAATAGTTGTCGACAGACCGTCGAGATTAGA

print("\n(Passed second tests of Exercise 3!)")
```

```
Checking: 'ANT|AAT'...
   Your result (3 fragments): ['ATGGCAATAACCCCCCGTTTCTACTTCTAGAGGAGAAAAGTATTGACATGAG
CGCTCCCGGCACAAGGGCCAAAGAAGTCTCCAATTTCTTATTTCCGAATGACATGCGTCTCCTTGCGGGTAAATCACCGACCGC
AATTCATAGAAGCCTGGGGGAACAGATAGGTCTAATTAGCTTAAGAGAGTAAATCCTGGGATCATTCAGTAGTAACCATAAACT
TACGCTGGGGCTTCTTCGGCGGATTTTTACAGTTACCAACCAGGAGATTTGAAGTAAATCAGTTGAGGATTTAGCCGCGCTATC
CGGTAATCTCCAAATTAAAACATACCGTTCCATGAAGGCTAGAATTACTTACCGGCCTTTTCCATGCCTGCGCTATACCCCCCC
ACTCTCCCGCTTATCCGTCCGAGCGGAGGCAGTGCGATCCTCCGTTAAGATATTCTTACGTGTGACGTAGCTATGTATTTTGCA
GAGCTGGCGAACGCGTTGAACACTTCACAGATGGTAGGGATTCGGGTAAAGGGCGTATAATTGGGGACTAACATAGGCGTAGAC
TACGATGGCGCCAACTCAATCGCAGCTCGAGCGCCCTGAATAACGTACTCATCTCAACTCATTCTCGGCAATCTACCGAGCGAC
TCGATTATCAACGGCTGTCTAGCAGTTCTAATCTTTTGCCAGCATCGTAATAGCCTCCAAGAGATTGATGATAGCTATCGGCAC
AGAACTGAGACGGCGCCGATGGATAGCGGACTTTCGGTCAACCACAATTCCCCACGGGACAGGTCCTGCGGTGCGCATCACTCT
GAATGTACAAGCAACCCAAGTGGGCCGAGCCTGGACTCAGCTGGTTCCTGCGTGAGCTCGAGACTCGGGATGACAGCTCTTTAA
ACATAGAGCGGGGGCGTCGAACGGTCGAGAAAGTCATAGTACCTCGGGTACCAACTTACTCAGGTTATTGCTTGAAGCTGTACT
ATTTTAGGGGGGGAGCGCTGAAGGTCTCTTCTTCTCATGACTGAACTCGCGAGGGTCGTGAAGTCGGTTCCTTCAATGGTTAAA
AAACAAAGGCTTACTGTGCGCAGAGGAACGCCCATCTAGCGGCTGGCGTCTTGAATGCTCGGTCCCCTTTGTCATTCCGGAT
T', 'AATCCATTTCCCTCATTCACGAGCTTGCGAAGTCTACATTGGTATATGAATGCGACCTAGAAGAGGGCGCTTAAAATTG
GCAGTGGTTGATGCTCTAAACTCCATTTGGTTTACTCGTGCATCACCGCGATAGGCTGACAAAGGTTTAACATTGAATAGCAAG
GCACTTCCGGTCTCAATGAACGGCCGGGAAAGGTACGCGCGCGGTATGGGAGGATCAAGGGGCCAATAGAGAGGCTCCTCTCTC
ACTCGCTAGGAGGCAAATGTAAAACAATGGTTACTGCATCGATACATAAAACATGTCCATCGGTTGCCCAAAGTGTTAAGTGTC
TATCACCCCTAGGGCCGTTTCCCGCATATAAACGCCAGGTTGTATCCGCATTTGATGCTACCGTGGATGAGTCTGCGTCGAGCG
CGCCGCACGAATGTTGCAATGTATTGCATGAGTAGGGTTGACTAAGAGCCGTTAGATGCGTCGCTGTACT', 'AATAGTTGTC
GACAGACCGTCGAGATTAGAAAATGGTACCAGCATTTTCGGAGGTTCTCTAACTAGTATGGATTGCGGTGTCTTCACTGTGCTG
CGGCTACCCATCGCCTGAAATCCAGCTGGTGTCAAGCCATCCCCTCTCCGGGACGCCGCATGTAGTGAAACATATACGTTGCAC
GGGTTCACCGCGGTCCGTTCTGAGTCGACCAAGGACACAATCGAGCTCCGATCCGTACCCTCGACAAACTTGTACCCGACCCCC
GGAGCTTGCCAGCTCCTCGGGTATCATGGAGCCTGTGGTTCATCGCGTCCGATATCAAACTTCGTCATGATAAAGTCCCCCCCT
CGGGAGTACCAGAGAAGATGACTACTGAGTTGTGCGAT']
   True result (3): ['ATGGCAATAACCCCCCGTTTCTACTTCTAGAGGAGAAAAGTATTGACATGAGCGCTCCCGGC
ACAAGGGCCAAAGAAGTCTCCAATTTCTTATTTCCGAATGACATGCGTCTCCTTGCGGGTAAATCACCGACCGCAATTCATAGA
AGCCTGGGGGAACAGATAGGTCTAATTAGCTTAAGAGAGTAAATCCTGGGATCATTCAGTAGTAACCATAAACTTACGCTGGGG
CTTCTTCGGCGGATTTTTACAGTTACCAACCAGGAGATTTGAAGTAAATCAGTTGAGGATTTAGCCGCGCTATCCGGTAATCTC
CAAATTAAAACATACCGTTCCATGAAGGCTAGAATTACTTACCGGCCTTTTCCATGCCTGCGCTATACCCCCCCACTCTCCCGC
TTATCCGTCCGAGCGGAGGCAGTGCGATCCTCCGTTAAGATATTCTTACGTGTGACGTAGCTATGTATTTTGCAGAGCTGGCGA
ACGCGTTGAACACTTCACAGATGGTAGGGATTCGGGTAAAGGGCGTATAATTGGGGACTAACATAGGCGTAGACTACGATGGCG
CCAACTCAATCGCAGCTCGAGCGCCCTGAATAACGTACTCATCTCAACTCATTCTCGGCAATCTACCGAGCGACTCGATTATCA
ACGGCTGTCTAGCAGTTCTAATCTTTTGCCAGCATCGTAATAGCCTCCAAGAGATTGATGATAGCTATCGGCACAGAACTGAGA
CGGCGCCGATGGATAGCGGACTTTCGGTCAACCACAATTCCCCACGGGACAGGTCCTGCGGTGCGCATCACTCTGAATGTACAA
GCAACCCAAGTGGGCCGAGCCTGGACTCAGCTGGTTCCTGCGTGAGCTCGAGACTCGGGATGACAGCTCTTTAAACATAGAGCG
GGGGCGTCGAACGGTCGAGAAAGTCATAGTACCTCGGGTACCAACTTACTCAGGTTATTGCTTGAAGCTGTACTATTTTAGGGG
GGGAGCGCTGAAGGTCTCTTCTTCTCATGACTGAACTCGCGAGGGTCGTGAAGTCGGTTCCTTCAATGGTTAAAAAACAAAGGC
TTACTGTGCGCAGAGGAACGCCCATCTAGCGGCTGGCGTCTTGAATGCTCGGTCCCCTTTGTCATTCCGGATT', 'AATCCAT
TTCCCTCATTCACGAGCTTGCGAAGTCTACATTGGTATATGAATGCGACCTAGAAGAGGGCGCTTAAAATTGGCAGTGGTTGAT
GCTCTAAACTCCATTTGGTTTACTCGTGCATCACCGCGATAGGCTGACAAAGGTTTAACATTGAATAGCAAGGCACTTCCGGTC
TCAATGAACGGCCGGGAAAGGTACGCGCGCGGTATGGGAGGATCAAGGGGCCAATAGAGAGGCTCCTCTCTCACTCGCTAGGAG
GCAAATGTAAAACAATGGTTACTGCATCGATACATAAAACATGTCCATCGGTTGCCCAAAGTGTTAAGTGTCTATCACCCCTAG
GGCCGTTTCCCGCATATAAACGCCAGGTTGTATCCGCATTTGATGCTACCGTGGATGAGTCTGCGTCGAGCGCGCCGCACGAAT
GTTGCAATGTATTGCATGAGTAGGGTTGACTAAGAGCCGTTAGATGCGTCGCTGTACT', 'AATAGTTGTCGACAGACCGTCG
AGATTAGAAAATGGTACCAGCATTTTCGGAGGTTCTCTAACTAGTATGGATTGCGGTGTCTTCACTGTGCTGCGGCTACCCATC
GCCTGAAATCCAGCTGGTGTCAAGCCATCCCCTCTCCGGGACGCCGCATGTAGTGAAACATATACGTTGCACGGGTTCACCGCG
GTCCGTTCTGAGTCGACCAAGGACACAATCGAGCTCCGATCCGTACCCTCGACAAACTTGTACCCGACCCCCGGAGCTTGCCAG
CTCCTCGGGTATCATGGAGCCTGTGGTTCATCGCGTCCGATATCAAACTTCGTCATGATAAAGTCCCCCCCTCGGGAGTACCAG
AGAAGATGACTACTGAGTTGTGCGAT']
   ==> Matched!

Checking: 'GCRW|TG'...
   Your result (3 fragments): ['ATGGCAATAACCCCCCGTTTCTACTTCTAGAGGAGAAAAGTATTGACATGAG
CGCTCCCGGCACAAGGGCCAAAGAAGTCTCCAATTTCTTATTTCCGAATGACATGCGTCTCCTTGCGGGTAAATCACCGACCGC
```

```
AATTCATAGAAGCCTGGGGGAACAGATAGGTCTAATTAGCTTAAGAGAGTAAATCCTGGGATCATTCAGTAGTAACCATAAACT
TACGCTGGGGCTTCTTCGGCGGATTTTTACAGTTACCAACCAGGAGATTTGAAGTAAATCAGTTGAGGATTTAGCCGCGCTATC
CGGTAATCTCCAAATTAAAACATACCGTTCCATGAAGGCTAGAATTACTTACCGGCCTTTTCCATGCCTGCGCTATACCCCCCC
ACTCTCCCGCTTATCCGTCCGAGCGGAGGCAGTGCGATCCTCCGTTAAGATATTCTTACGTGTGACGTAGCTATGTATTTTGCA
GAGCTGGCGAACGCGT', 'TGAACACTTCACAGATGGTAGGGATTCGGGTAAAGGGCGTATAATTGGGGACTAACATAGGCGT
AGACTACGATGGCGCCAACTCAATCGCAGCTCGAGCGCCCTGAATAACGTACTCATCTCAACTCATTCTCGGCAATCTACCGAG
CGACTCGATTATCAACGGCTGTCTAGCAGTTCTAATCTTTTGCCAGCATCGTAATAGCCTCCAAGAGATTGATGATAGCTATCG
GCACAGAACTGAGACGGCGCCGATGGATAGCGGACTTTCGGTCAACCACAATTCCCCACGGGACAGGTCCTGCGGTGCGCATCA
CTCTGAATGTACAAGCAACCCAAGTGGGCCGAGCCTGGACTCAGCTGGTTCCTGCGTGAGCTCGAGACTCGGGATGACAGCTCT
TTAAACATAGAGCGGGGGCGTCGAACGGTCGAGAAAGTCATAGTACCTCGGGTACCAACTTACTCAGGTTATTGCTTGAAGCTG
TACTATTTTAGGGGGGGAGCGCTGAAGGTCTCTTCTTCTCATGACTGAACTCGCGAGGGTCGTGAAGTCGGTTCCTTCAATGGT
TAAAAAACAAAGGCTTACTGTGCGCAGAGGAACGCCCATCTAGCGGCTGGCGTCTTGAATGCTCGGTCCCCTTTGTCATTCCGG
ATTAATCCATTTCCCTCATTCACGAGCTTGCGAAGTCTACATTGGTATATGAATGCGACCTAGAAGAGGGCGCTTAAAATTGGC
AGTGGTTGATGCTCTAAACTCCATTTGGTTTACTCGTGCATCACCGCGATAGGCTGACAAAGGTTTAACATTGAATAGCAAGGC
ACTTCCGGTCTCAATGAACGGCCGGGAAAGGTACGCGCGCGGTATGGGAGGATCAAGGGGCCAATAGAGAGGCTCCTCTCTCAC
TCGCTAGGAGGCAAATGTAAAACAATGGTTACTGCATCGATACATAAAACATGTCCATCGGTTGCCCAAAGTGTTAAGTGTCTA
TCACCCCTAGGGCCGTTTCCCGCATATAAACGCCAGGTTGTATCCGCATTTGATGCTACCGTGGATGAGTCTGCGTCGAGCGCG
CCGCACGAATGTTGCAA', 'TGTATTGCATGAGTAGGGTTGACTAAGAGCCGTTAGATGCGTCGCTGTACTAATAGTTGTCGA
CAGACCGTCGAGATTAGAAAATGGTACCAGCATTTTCGGAGGTTCTCTAACTAGTATGGATTGCGGTGTCTTCACTGTGCTGCG
GCTACCCATCGCCTGAAATCCAGCTGGTGTCAAGCCATCCCCTCTCCGGGACGCCGCATGTAGTGAAACATATACGTTGCACGG
GTTCACCGCGGTCCGTTCTGAGTCGACCAAGGACACAATCGAGCTCCGATCCGTACCCTCGACAAACTTGTACCCGACCCCCGG
AGCTTGCCAGCTCCTCGGGTATCATGGAGCCTGTGGTTCATCGCGTCCGATATCAAACTTCGTCATGATAAAGTCCCCCCCTCG
GGAGTACCAGAGAAGATGACTACTGAGTTGTGCGAT']
   True result (3): ['ATGGCAATAACCCCCCGTTTCTACTTCTAGAGGAGAAAAGTATTGACATGAGCGCTCCCGGC
ACAAGGGCCAAAGAAGTCTCCAATTTCTTATTTCCGAATGACATGCGTCTCCTTGCGGGTAAATCACCGACCGCAATTCATAGA
AGCCTGGGGGAACAGATAGGTCTAATTAGCTTAAGAGAGTAAATCCTGGGATCATTCAGTAGTAACCATAAACTTACGCTGGGG
CTTCTTCGGCGGATTTTTACAGTTACCAACCAGGAGATTTGAAGTAAATCAGTTGAGGATTTAGCCGCGCTATCCGGTAATCTC
CAAATTAAAACATACCGTTCCATGAAGGCTAGAATTACTTACCGGCCTTTTCCATGCCTGCGCTATACCCCCCCACTCTCCCGC
TTATCCGTCCGAGCGGAGGCAGTGCGATCCTCCGTTAAGATATTCTTACGTGTGACGTAGCTATGTATTTTGCAGAGCTGGCGA
ACGCGT', 'TGAACACTTCACAGATGGTAGGGATTCGGGTAAAGGGCGTATAATTGGGGACTAACATAGGCGTAGACTACGAT
GGCGCCAACTCAATCGCAGCTCGAGCGCCCTGAATAACGTACTCATCTCAACTCATTCTCGGCAATCTACCGAGCGACTCGATT
ATCAACGGCTGTCTAGCAGTTCTAATCTTTTGCCAGCATCGTAATAGCCTCCAAGAGATTGATGATAGCTATCGGCACAGAACT
GAGACGGCGCCGATGGATAGCGGACTTTCGGTCAACCACAATTCCCCACGGGACAGGTCCTGCGGTGCGCATCACTCTGAATGT
ACAAGCAACCCAAGTGGGCCGAGCCTGGACTCAGCTGGTTCCTGCGTGAGCTCGAGACTCGGGATGACAGCTCTTTAAACATAG
AGCGGGGGCGTCGAACGGTCGAGAAAGTCATAGTACCTCGGGTACCAACTTACTCAGGTTATTGCTTGAAGCTGTACTATTTTA
GGGGGGGAGCGCTGAAGGTCTCTTCTTCTCATGACTGAACTCGCGAGGGTCGTGAAGTCGGTTCCTTCAATGGTTAAAAAACAA
AGGCTTACTGTGCGCAGAGGAACGCCCATCTAGCGGCTGGCGTCTTGAATGCTCGGTCCCCTTTGTCATTCCGGATTAATCCAT
TTCCCTCATTCACGAGCTTGCGAAGTCTACATTGGTATATGAATGCGACCTAGAAGAGGGCGCTTAAAATTGGCAGTGGTTGAT
GCTCTAAACTCCATTTGGTTTACTCGTGCATCACCGCGATAGGCTGACAAAGGTTTAACATTGAATAGCAAGGCACTTCCGGTC
TCAATGAACGGCCGGGAAAGGTACGCGCGCGGTATGGGAGGATCAAGGGGCCAATAGAGAGGCTCCTCTCTCACTCGCTAGGAG
GCAAATGTAAAACAATGGTTACTGCATCGATACATAAAACATGTCCATCGGTTGCCCAAAGTGTTAAGTGTCTATCACCCCTAG
GGCCGTTTCCCGCATATAAACGCCAGGTTGTATCCGCATTTGATGCTACCGTGGATGAGTCTGCGTCGAGCGCGCCGCACGAAT
GTTGCAA', 'TGTATTGCATGAGTAGGGTTGACTAAGAGCCGTTAGATGCGTCGCTGTACTAATAGTTGTCGACAGACCGTCG
AGATTAGAAAATGGTACCAGCATTTTCGGAGGTTCTCTAACTAGTATGGATTGCGGTGTCTTCACTGTGCTGCGGCTACCCATC
GCCTGAAATCCAGCTGGTGTCAAGCCATCCCCTCTCCGGGACGCCGCATGTAGTGAAACATATACGTTGCACGGGTTCACCGCG
GTCCGTTCTGAGTCGACCAAGGACACAATCGAGCTCCGATCCGTACCCTCGACAAACTTGTACCCGACCCCCGGAGCTTGCCAG
CTCCTCGGGTATCATGGAGCCTGTGGTTCATCGCGTCCGATATCAAACTTCGTCATGATAAAGTCCCCCCCTCGGGAGTACCAG
AGAAGATGACTACTGAGTTGTGCGAT']
   ==> Matched!

(Passed second tests of Exercise 3!)
```

Fin! If you've reached this point and all tests above pass, your biologist friend thanks you and you are ready to submit your solution to this problem. Don't forget to save you work prior to submitting.

Portions of this problem were inspired by a fun book called Python for Biologists.