Joshua Burkhart
BMI 650
11/18/2015

Midterm 2015

1: Determine length of shortest subsequence that occurs exactly once in a genomic sequence of length N. Provide assumptions.

```
s = 'ABABABABABABABWWWWWWWWWRRRRRRRRXXXXXXXX'

def short_u_sub_strs(s):
    """
    This function assumes |s| is finite and fits in RAM.

    Runtime is bound by the triangle number of |s|,
    approx. O((n^2)/2), though an average case execution
    would run in O(((n/2)^2)/2) = O((n^2)/8).
    :param s: input string
    :return: list of shortest unique substrings
    """
    u_sub_strs = []
    for l in range(1,len(s)+1):
        found_u_sub_strs = False
        sub_strs = {}
        for i in range((len(s)-l)+1):
            sub_str = str(s[i:i+l])
            if sub_str in sub_strs:
                sub_strs[sub_str] += 1
            else:
                sub_strs[sub_str] = 1
        for sub_str in sub_strs:
            if sub_strs[sub_str] == 1:
                u_sub_strs.append(sub_str)
                found_u_sub_strs = True
        if found_u_sub_strs:
            break
    return u_sub_strs

shortlist = short_u_sub_strs(s)

print(shortlist)
print(len(shortlist[0]) if len(shortlist) > 0 else 0)
```

```
['WR', 'RX', 'BW']
2
```

2: Regarding HG38 and CD8B

a. What chromosome is this gene on?

The below link shows it's on chromosome 2.

https://genome.ucsc.edu/cgi-bin/hgc?hgsid=454559201_hi3ZA8G9DUTVIRE16jb1ZW8vB2jf&c=chr2&o=86815336&t=86861924&g=refGene&i=NM_172213

b. What is the start and end coordinate of this gene?

Transcript links (bottom) show positions:

chr2:86815337-86861924
chr2:86815337-86861924
chr2:86815337-86861924
chr2:86815337-86861924
chr2:86841566-86861924

The widest range of these coordinates is 86815337-86861924. As all transcripts are shown to be on the negative strand, we'll report the start coordinate as 86861924 and the end as 86815337.

https://genome.ucsc.edu/cgi-bin/hgc?hgsid=454559201_hi3ZA8G9DUTVIRE16jb1ZW8vB2jf&c=chr2&o=86815336&t=86861924&g=refGene&i=NM_172213
https://genome.ucsc.edu/cgi-bin/hgc?hgsid=454559201_hi3ZA8G9DUTVIRE16jb1ZW8vB2jf&c=chr2&o=86815336&t=86861924&g=refGene&i=NM_172102
https://genome.ucsc.edu/cgi-bin/hgc?hgsid=454559201_hi3ZA8G9DUTVIRE16jb1ZW8vB2jf&c=chr2&o=86815336&t=86861924&g=refGene&i=NM_172101
https://genome.ucsc.edu/cgi-bin/hgc?hgsid=454559201_hi3ZA8G9DUTVIRE16jb1ZW8vB2jf&c=chr2&o=86815336&t=86861924&g=refGene&i=NM_001178100
https://genome.ucsc.edu/cgi-bin/hgc?hgsid=454559201_hi3ZA8G9DUTVIRE16jb1ZW8vB2jf&c=chr2&o=86841565&t=86861924&g=refGene&i=NM_004931

c. What strand of the DNA contains this gene?

Negative (see above links)

d. How many transcripts are in this gene as reported by the RefSeq Consortium?

5 (see above links)

e. How many distinct exons are reported in the RefSeq transcripts?

Visualy inspecting the 5 transcripts using the UCSC genome browser (link at bottom) indicates the first has 6 exons, the second has 5 exons (all shared with the first), the third has 7 exons (1 distinct and 6 shared with the first), the fourth has 4 exons (all shared with the first), and the fifth has 6 exons (1 distinct and 5 shared with the first). Together, the transcripts have 8 distinct exons.

https://genome.ucsc.edu/cgi-bin/hgTracks?db=hg38&position=chr2%3A86812162-86863658&hgsid=454559201_hi3ZA8G9DUTVIRE16jb1ZW8vB2jf

f. How many distinct introns are reported in the RefSeq transcripts?

Again, visually inspecting the 5 transcripts using the UCSC genome browser (see above) shows the first has 5 introns, the second has 4 introns (1 distinct and 3 shared with the first), the third has 6 introns (2 distinct and 4 shared with the first), the fourth has 3 introns (1 distinct and 2 shared with the first), and the fifth has 5 introns (1 distinct and 4 shared with the first). Together, the transcripts have 10 distinct introns.

g. List the genomic coordinates of each intron (start-end) from above which contains a canonical splice site.

Visually inspecting the transcript alignments (links at bottom) shows 9 introns are cononical, begining with GT and ending with AG. The coordinates are as follows:

86861822 - 86858417 [gt-ag]
86858056 - 86853087 [gt-ag]
86852996 - 86846774 [gt-ag]
86846683 - 86844959 [gt-ag]
86844821 - 86815719 [gt-ag]
86852996 - 86844959 [gt-ag]
86844821 - 86822375 [gt-ag]
86852996 - 86815719 [gt-ag]
86844821 - 86842320 [gt-ag]

https://genome.ucsc.edu/cgi-bin/hgc?hgsid=454559201_hi3ZA8G9DUTVIRE16jb1ZW8vB2jf&g=htcCdnaAli&i=NM_172213&c=chr2&l=86800257&r=86903247&o=86815336&aliTable=refSeqAli&table=refGene
https://genome.ucsc.edu/cgi-bin/hgc?hgsid=454559201_hi3ZA8G9DUTVIRE16jb1ZW8vB2jf&g=htcCdnaAli&i=NM_172102&c=chr2&l=86800257&r=86903247&o=86815336&aliTable=refSeqAli&table=refGene
https://genome.ucsc.edu/cgi-bin/hgc?hgsid=454559201_hi3ZA8G9DUTVIRE16jb1ZW8vB2jf&g=htcCdnaAli&i=NM_172101&c=chr2&l=86800257&r=86903247&o=86815336&aliTable=refSeqAli&table=refGene
https://genome.ucsc.edu/cgi-bin/hgc?hgsid=454559201_hi3ZA8G9DUTVIRE16jb1ZW8vB2jf&g=htcCdnaAli&i=NM_001178100&c=chr2&l=86800257&r=86903247&o=86815336&aliTable=refSeqAli&table=refGene
https://genome.ucsc.edu/cgi-bin/hgc?hgsid=454559201_hi3ZA8G9DUTVIRE16jb1ZW8vB2jf&g=htcCdnaAli&i=NM_004931&c=chr2&l=86800257&r=86903247&o=86841565&aliTable=refSeqAli&table=refGene

h. List the genomic coordinates of each intron (start-end) from above which contains a non-canonical splice site.

86822320 - 86844959 [AT...AGgt-agGC...AA]

(see links above)

1) What is the official gene symbol of the nearest gene to CD8B?

Visually inspection using the UCSC genome browser (link below) shows several genes are nearby.

Assuming the UCSC genome browser uses official gene symbols, ANAPC1P1 is the nearest in bp, though it's on the opposite (positive) strand.

https://genome.ucsc.edu/cgi-bin/hgTracks?db=hg38&position=chr2%3A86753188-86926991&hgsid=454559201_hi3ZA8G9DUTVIRE16jb1ZW8vB2jf

2) What strand of the DNA contains this nearest neighboring gene?

Positive (see above)

3) How would the location of these genes be described in relation to one another?

The first exons of CD8B and ANAPC1P1 are derived from the same area of the genome, though on different strands. Both CD8B and ANAPC1P1 extend in opposite directions on their respective strands.

3: Describe minimum information needed to unambiguously define the location of a gene. Provide example.

[Genome Assembly, Start Coordinate, Stop Coordinate]

Example:

Genome Assembly: GRCh38/hg38
Start Coordinate: chr2:86861924
Stop Coordinate: chr2:86815337

We can infer strand by testing if the start coordinate is in front of or behind the stop coordinate.

4: Write program to randomly generate two sequences which will result in...

a. 1) a better local alignment as compared to global alignment (using blosum62 scoring matrix), both 10 amino acids long.

```
from random import randint

SIGMA = 'ARNDCQEGHILKMFPSTWYVBZX'

def r_char():
    """
    Generates a random character
    :return: a random character from SIGMA
    """
    return SIGMA[randint(0,len(SIGMA)-1)]


def random_sequence(l):
    """
```

```
    Generates a random sequence
    :param l: length of random sequence to build
    :return: random sequence of length l
    """
    r_seq = ''
    for i in range(l):
        r_seq += r_char()
    return r_seq

def local_align_sequence(r_seq):
    """
    Matches middle third of the characters in r_seq
    :param r_seq: any random sequence
    :return: a sequence that will align locally to r_seq
    """
    l_seq = ''
    t3 = len(r_seq)
    t1 = int(len(r_seq)/3)
    t2 = 2*t1
    for i in range(t1):
        l_seq += r_char().lower()
    for i in range(t1,t2):
        l_seq += r_seq[i]
    for i in range(t2,t3):
        l_seq += r_char().lower()
    return l_seq

r = random_sequence(10)
l = local_align_sequence(r)

print('r: ' + r)
print('l: ' + l)
```

```
r: BMPTSQCVNS
l: vvtTSQalxc
```

a. 2) Discuss criteria and any limitations in generating your sequences.

A pseudorandom number generator (PRNG) is used to select all characters for an initial 'random' sequence. The PRNG is reused to select the first and last thirds of a second sequence of equal length, while the middle third is selected by simply using the characters from the first.

a. 3) Align the two sequences using dynamic programming using the blosum62 scoring matrix. Show the DP matrix and the resultant alignment.

```
import math
from random import randint

SIGMA = 'ARNDCQEGHILKMFPSTWYVBZX'
ILLEGAL_CHARS = ' -\n'
GAP_PENALTY = -4 # taken from blosum62 matrix
```

```
BLOSUM62 = {frozenset({'R', 'P'}): -2.0, frozenset({'S', 'G'}): 0.0, frozenset({'T', 'K'}): -1.0,
            frozenset({'N', 'G'}): 0.0, frozenset({'Z', 'G'}): -2.0, frozenset({'P', 'B'}): -2.0,
            frozenset({'T', 'W'}): -2.0, frozenset({'N', 'D'}): 1.0, frozenset({'*', 'N'}): -4.0, frozen
            frozenset({'Q', 'A'}): -1.0, frozenset({'N', 'K'}): 0.0, frozenset({'*', 'M'}): -4.0,
            frozenset({'N', 'A'}): -2.0, frozenset({'Q', 'P'}): -1.0, frozenset({'X', 'A'}): 0.0,
            frozenset({'Z', 'K'}): 1.0, frozenset({'*', 'K'}): -4.0, frozenset({'H', 'D'}): -1.0,
            frozenset({'C', 'D'}): -3.0, frozenset({'B', 'X'}): -1.0, frozenset({'Q', 'R'}): 1.0,
            frozenset({'W', 'P'}): -4.0, frozenset({'S', 'Y'}): -2.0, frozenset({'*', 'W'}): -4.0,
            frozenset({'W', 'D'}): -4.0, frozenset({'K', 'M'}): -1.0, frozenset({'I', 'V'}): 3.0,
            frozenset({'E', 'K'}): 1.0, frozenset({'C', 'Y'}): -2.0, frozenset({'R', 'D'}): -2.0,
            frozenset({'Z', 'B'}): 1.0, frozenset({'P', 'G'}): -2.0, frozenset({'Q', 'C'}): -3.0,
            frozenset({'R', 'S'}): -1.0, frozenset({'B', 'L'}): -4.0, frozenset({'R', 'Z'}): 0.0,
            frozenset({'I', 'F'}): 0.0, frozenset({'S', 'A'}): 1.0, frozenset({'I', 'B'}): -3.0,
            frozenset({'F', 'Y'}): 3.0, frozenset({'D', 'A'}): -2.0, frozenset({'E', 'C'}): -4.0,
            frozenset({'H', 'E'}): 0.0, frozenset({'I', 'C'}): -1.0, frozenset({'H', 'I'}): -3.0,
            frozenset({'Q', 'T'}): -1.0, frozenset({'V', 'D'}): -3.0, frozenset({'K', 'D'}): -1.0,
            frozenset({'Q', 'V'}): -2.0, frozenset({'C', 'B'}): -3.0, frozenset({'*', 'V'}): -4.0,
            frozenset({'X', 'Y'}): -1.0, frozenset({'F', 'V'}): -1.0, frozenset({'I', 'M'}): 1.0,
            frozenset({'P', 'M'}): -2.0, frozenset({'E', 'W'}): -3.0, frozenset({'W', 'A'}): -3.0,
            frozenset({'E', 'L'}): -3.0, frozenset({'T', 'Z'}): -1.0, frozenset({'*', 'E'}): -4.0,
            frozenset({'I', 'N'}): -3.0, frozenset({'E', 'Z'}): 4.0, frozenset({'H', 'A'}): -2.0,
            frozenset({'B', 'K'}): 0.0, frozenset({'M'}): 5.0, frozenset({'T', 'Y'}): -2.0, frozenset({
            frozenset({'K'}): 5.0, frozenset({'W', 'N'}): -4.0, frozenset({'I', 'X'}): -1.0, frozenset(
            frozenset({'N', 'L'}): -3.0, frozenset({'W', 'C'}): -2.0, frozenset({'H', 'W'}): -2.0,
            frozenset({'F', 'M'}): 0.0, frozenset({'S', 'D'}): 0.0, frozenset({'*', 'D'}): -4.0,
            frozenset({'P', 'X'}): -2.0, frozenset({'T', 'L'}): -1.0, frozenset({'L', 'M'}): 2.0,
            frozenset({'D', 'M'}): -3.0, frozenset({'*', 'T'}): -4.0, frozenset({'P', 'Y'}): -3.0,
            frozenset({'T', 'G'}): -2.0, frozenset({'R', 'Y'}): -2.0, frozenset({'I', 'P'}): -3.0,
            frozenset({'*', 'S'}): -4.0, frozenset({'S', 'W'}): -3.0, frozenset({'F', 'G'}): -3.0,
            frozenset({'I', 'T'}): -1.0, frozenset({'T', 'E'}): -1.0, frozenset({'G'}): 6.0, frozenset(
            frozenset({'W', 'Y'}): 2.0, frozenset({'V', 'M'}): 1.0, frozenset({'N', 'C'}): -3.0,
            frozenset({'V', 'L'}): 1.0, frozenset({'Q', 'Z'}): 3.0, frozenset({'R', 'G'}): -2.0,
            frozenset({'F', 'X'}): -1.0, frozenset({'W', 'B'}): -4.0, frozenset({'H', 'B'}): 0.0,
            frozenset({'V', 'A'}): 0.0, frozenset({'Z', 'Y'}): -2.0, frozenset({'W', 'L'}): -2.0,
            frozenset({'K', 'A'}): -1.0, frozenset({'T', 'A'}): 0.0, frozenset({'Y', 'D'}): -3.0,
            frozenset({'R', 'C'}): -3.0, frozenset({'X', 'M'}): -1.0, frozenset({'Q', 'L'}): -2.0,
            frozenset({'Q', 'D'}): 0.0, frozenset({'I', 'W'}): -3.0, frozenset({'E', 'D'}): 2.0,
            frozenset({'R', 'T'}): -1.0, frozenset({'Y', 'G'}): -3.0, frozenset({'S', 'E'}): 0.0,
            frozenset({'F', 'P'}): -4.0, frozenset({'S'}): 4.0, frozenset({'W'}): 11.0, frozenset({'C',
            frozenset({'T', 'B'}): -1.0, frozenset({'R', 'F'}): -3.0, frozenset({'H', 'K'}): -1.0,
            frozenset({'H', 'M'}): -2.0, frozenset({'C', 'M'}): -1.0, frozenset({'S', 'M'}): -1.0,
            frozenset({'Q', 'I'}): -3.0, frozenset({'R'}): 5.0, frozenset({'F', 'C'}): -2.0,
            frozenset({'H', 'F'}): -1.0, frozenset({'V', 'G'}): -3.0, frozenset({'Z', 'L'}): -3.0,
            frozenset({'W', 'V'}): -3.0, frozenset({'*', 'Z'}): -4.0, frozenset({'H', 'C'}): -3.0,
            frozenset({'N', 'V'}): -3.0, frozenset({'L', 'D'}): -4.0, frozenset({'R', 'V'}): -3.0,
            frozenset({'Q', 'Y'}): -1.0, frozenset({'S', 'L'}): -2.0, frozenset({'N', 'M'}): -2.0,
            frozenset({'Z', 'M'}): -1.0, frozenset({'F', 'L'}): 0.0, frozenset({'H', 'G'}): -2.0,
            frozenset({'C', 'G'}): -3.0, frozenset({'F', 'Z'}): -3.0, frozenset({'V', 'B'}): -3.0,
            frozenset({'H', 'L'}): -3.0, frozenset({'E'}): 5.0, frozenset({'*', 'C'}): -4.0,
            frozenset({'H', '*'}): -4.0, frozenset({'F', 'B'}): -3.0, frozenset({'E', 'A'}): -1.0,
            frozenset({'R', 'A'}): -1.0, frozenset({'H', 'R'}): 0.0, frozenset({'E', 'N'}): 0.0,
            frozenset({'T', 'V'}): 0.0, frozenset({'*', 'B'}): -4.0, frozenset({'G', 'A'}): 0.0,
            frozenset({'H', 'S'}): -1.0, frozenset({'Q', 'F'}): -3.0, frozenset({'N', 'P'}): -2.0,
```

```
            frozenset({'D'}): 6.0, frozenset({'Z', 'X'}): -1.0, frozenset({'F', 'W'}): 1.0, frozenset({
            frozenset({'X', 'D'}): -1.0, frozenset({'Z', 'V'}): -2.0, frozenset({'Q', '*'}): -4.0,
            frozenset({'Q', 'S'}): 0.0, frozenset({'L', 'K'}): -2.0, frozenset({'W', 'Z'}): -3.0,
            frozenset({'*', 'Y'}): -4.0, frozenset({'*', 'L'}): -4.0, frozenset({'Q'}): 5.0,
            frozenset({'L', 'A'}): -1.0, frozenset({'E', 'V'}): -2.0, frozenset({'I', 'Z'}): -3.0,
            frozenset({'I', 'K'}): -3.0, frozenset({'E', 'Y'}): -2.0, frozenset({'C', 'K'}): -3.0,
            frozenset({'I', 'Y'}): -1.0, frozenset({'*', 'G'}): -4.0, frozenset({'E', 'X'}): -1.0,
            frozenset({'S', 'X'}): 0.0, frozenset({'Q', 'X'}): -1.0, frozenset({'P', 'A'}): -1.0,
            frozenset({'F', 'K'}): -3.0, frozenset({'S', 'Z'}): 0.0, frozenset({'Z'}): 4.0, frozenset({
            frozenset({'Q', 'H'}): 0.0, frozenset({'*', 'F'}): -4.0, frozenset({'F', 'D'}): -3.0,
            frozenset({'T', 'D'}): -1.0, frozenset({'S', 'K'}): 0.0, frozenset({'I', '*'}): -4.0,
            frozenset({'S', 'T'}): 1.0, frozenset({'C', 'L'}): -1.0, frozenset({'E', 'P'}): -1.0,
            frozenset({'X'}): -1.0, frozenset({'R', 'X'}): -1.0, frozenset({'B', 'Y'}): -3.0,
            frozenset({'R', 'L'}): -2.0, frozenset({'E', 'M'}): -2.0, frozenset({'*', 'R'}): -4.0,
            frozenset({'H', 'V'}): -3.0, frozenset({'Q', 'W'}): -2.0, frozenset({'E', 'F'}): -3.0,
            frozenset({'Z', 'D'}): 1.0, frozenset({'P', 'V'}): -2.0, frozenset({'V', 'K'}): -2.0, froze
            frozenset({'R', 'W'}): -3.0, frozenset({'S', 'P'}): -1.0, frozenset({'Z', 'A'}): -1.0,
            frozenset({'P', 'D'}): -1.0, frozenset({'*', 'X'}): -4.0, frozenset({'V', 'Y'}): -1.0,
            frozenset({'X', 'L'}): -1.0, frozenset({'I', 'G'}): -4.0, frozenset({'E', 'G'}): -2.0,
            frozenset({'L', 'Y'}): -1.0, frozenset({'P', 'K'}): -1.0, frozenset({'R', 'M'}): -1.0,
            frozenset({'R', 'K'}): 2.0, frozenset({'W', 'M'}): -1.0, frozenset({'C'}): 9.0, frozenset({
            frozenset({'I', 'S'}): -2.0, frozenset({'S', 'B'}): 0.0, frozenset({'X', 'G'}): -1.0,
            frozenset({'V', 'X'}): -1.0, frozenset({'L', 'G'}): -4.0, frozenset({'T', 'C'}): -1.0,
            frozenset({'T'}): 5.0, frozenset({'Q', 'E'}): 2.0, frozenset({'B'}): 4.0, frozenset({'P'}):
            frozenset({'W', 'K'}): -3.0, frozenset({'I'}): 4.0, frozenset({'*'}): 1.0, frozenset({'R',
            frozenset({'T', 'F'}): -2.0, frozenset({'T', 'M'}): -1.0, frozenset({'Z', 'N'}): 0.0,
            frozenset({'I', 'E'}): -3.0, frozenset({'T', 'N'}): 0.0, frozenset({'S', 'F'}): -2.0,
            frozenset({'E', 'B'}): 1.0, frozenset({'Y', 'A'}): -2.0, frozenset({'W', 'G'}): -2.0,
            frozenset({'I', 'R'}): -3.0, frozenset({'R', 'E'}): 0.0, frozenset({'Y'}): 7.0, frozenset({
            frozenset({'T', 'X'}): 0.0, frozenset({'G', 'M'}): -3.0, frozenset({'K', 'G'}): -2.0,
            frozenset({'S', 'C'}): -1.0, frozenset({'Z', 'C'}): -3.0, frozenset({'H', 'Z'}): 0.0,
            frozenset({'Q', 'B'}): 0.0, frozenset({'C', 'V'}): -1.0, frozenset({'V'}): 4.0, frozenset({
            frozenset({'H', 'Y'}): 2.0, frozenset({'W', 'X'}): -2.0, frozenset({'Q', 'N'}): 0.0,
            frozenset({'R', 'N'}): 0.0, frozenset({'B', 'D'}): 4.0, frozenset({'H', 'X'}): -1.0,
            frozenset({'C', 'X'}): -2.0, frozenset({'*', 'A'}): -4.0, frozenset({'N', 'B'}): 3.0,
            frozenset({'B', 'G'}): -1.0, frozenset({'*', 'P'}): -4.0, frozenset({'K', 'Y'}): -2.0,
            frozenset({'Y', 'M'}): -1.0, frozenset({'P', 'L'}): -3.0, frozenset({'Q', 'K'}): 1.0,
            frozenset({'S', 'V'}): -2.0, frozenset({'F'}): 6.0, frozenset({'M', 'A'}): -1.0,
            frozenset({'F', 'A'}): -2.0, frozenset({'N', 'X'}): -1.0, frozenset({'T', 'P'}): -1.0,
            frozenset({'S', 'N'}): 1.0, frozenset({'Q', 'G'}): -2.0, frozenset({'B', 'A'}): -2.0,
            frozenset({'H', 'P'}): -2.0, frozenset({'C', 'P'}): -3.0, frozenset({'X', 'K'}): -1.0,
            frozenset({'Q', 'M'}): 0.0, frozenset({'Z', 'P'}): -1.0, frozenset({'B', 'M'}): -3.0,
            frozenset({'I', 'L'}): 2.0, frozenset({'H', 'T'}): -2.0}


class MultipleAlignment:
    """
    represents a simple multiple alignment
    """

    def __init__(self):
        self.sequence_matrix = []
```

```
    def add_sequence(self, sequence):
        """
        adds sequence characters to matrix, appending
        to existing values such that aligned characters
        will end up occupying the same index
        ex:
        'ZWAB'
        'B-ZX'
        will end up as
        ['ZB','W-','AZ','BX']
        :param sequence:
        """
        for idx, char in enumerate(sequence):
            if idx < len(self.sequence_matrix):
                self.sequence_matrix[idx] += char
            else:
                self.sequence_matrix.append(char)

    def sum_aligned_char_scores(self, scoring_function, *sf_args):
        """
        addition commutes. aligned characters of matrix
        are summed, simulating the summation of whichever
        scoring function
        :param scoring_function:
        :param sf_args:
        :return:
        """
        score_sum = 0.0
        for aligned_chars in self.sequence_matrix:
            if all(char not in ILLEGAL_CHARS for char in aligned_chars):
                score_sum += scoring_function(aligned_chars, sf_args)
        return score_sum

    def minimize_aligned_char_scores(self, scoring_function, *sf_args):
        """
        returns the minimum value returned by the
        scoring function for aligned characters
        :param scoring_function:
        :param sf_args:
        :return:
        """
        score_list = []
        for aligned_chars in self.sequence_matrix:
            score_list.append(scoring_function(aligned_chars, sf_args))
        return min(score_list)

def sum_of_pairs(aligned_chars, param_tuple):
    """Sum-of-Pairs (SP-score)
    Compeau, P, Pevzner, P,c Active Learning Publishers,
    Bioinformatics Algorithms: An Active Learning Approach,
    2nd Ed., Vol. I, p. 290
    :param aligned_chars: the characters in the alignment
    :param param_tuple: a tuple with a scoring matrix wrapper
```

```
    """
    matrix_wrapper = param_tuple[0]
    sp = 0.0
    for i in range(0, len(aligned_chars)):
        for j in range(i + 1, len(aligned_chars)):
            sp += matrix_wrapper(aligned_chars[i], aligned_chars[j])
    return sp + 0  # trick from http://goo.gl/8u8kQw

def h(char_i, char_j):
    """
    simply wraps dictionary
    """
    return BLOSUM62[frozenset([char_i, char_j])]

def set_globs(s1,s2):
    global _s1
    global _s2
    global _dp
    _s1 = '*' + s1
    _s2 = '*' + s2

    # create DP matrix with S1 + 1 columns and S2 + 1 rows
    _dp = [[0 for x in range(len(_s1) + 1)] for x in range(len(_s2) + 1)]

def fill_l_dp_matrix():
    global _dp
    for i in range(len(_s2)):
        for j in range(len(_s1)):
            if i > 0 and j > 0:
                _dp[i][j] = max(_dp[i - 1][j - 1] + h(_s2[i], _s1[j]),
                                _dp[i - 1][j] + GAP_PENALTY,
                                _dp[i][j - 1] + GAP_PENALTY,
                                0) # allow for local alignment
            elif i > 0:
                _dp[i][j] = _dp[i - 1][j] + GAP_PENALTY
            elif j > 0:
                _dp[i][j] = _dp[i][j - 1] + GAP_PENALTY
            else:
                _dp[i][j] = 0

def fill_dp_matrix():
    global _dp
    for i in range(len(_s2)):
        for j in range(len(_s1)):
            if i > 0 and j > 0:
                _dp[i][j] = max(_dp[i - 1][j - 1] + h(_s2[i], _s1[j]),
                                _dp[i - 1][j] + GAP_PENALTY,
                                _dp[i][j - 1] + GAP_PENALTY)
            elif i > 0:
                _dp[i][j] = _dp[i - 1][j] + GAP_PENALTY
            elif j > 0:
                _dp[i][j] = _dp[i][j - 1] + GAP_PENALTY
            else:
```

```
                _dp[i][j] = 0

def print_dp_matrix():
    for c in range(len(_s1)):
        print(' {0}'.format(_s1[c]).center(4, ' '), end='')
    print()
    for i in range(len(_s2)):
        print('{0}'.format(_s2[i]), end='')
        for j in range(len(_s1)):
            print('{0}'.format(_dp[i][j]).center(4, ' '), end='')
        print()

def lbacktrace():
    s1_alignment = ''
    s2_alignment = ''
    i = len(_s2) - 1
    j = len(_s1) - 1
    while i > 0 or j > 0:
        if i > 0 and j > 0:
            diag = _dp[i - 1][j - 1]
            up = _dp[i - 1][j]
            left = _dp[i][j - 1]
            if _dp[i][j] == 0:
                if len(s2_alignment.strip()) > 0: #done with local alignment, abort
                    i = 0
                else:
                    s1_alignment = _s1[j] + s1_alignment
                    s2_alignment = ' ' + s2_alignment
                    i -= 1
                    j -= 1
            elif (_dp[i][j] - diag) == h(_s2[i],_s1[j]):
                s1_alignment = _s1[j] + s1_alignment
                s2_alignment = _s2[i] + s2_alignment
                i -= 1
                j -= 1
            elif (_dp[i][j] - up) == GAP_PENALTY:
                s1_alignment = '*' + s1_alignment
                s2_alignment = _s2[i] + s2_alignment
                i -= 1
            elif (_dp[i][j] - left) == GAP_PENALTY:
                s1_alignment = _s1[j] + s1_alignment
                s2_alignment = '*' + s2_alignment
                j -= 1
        elif i > 0:  # up
            s1_alignment = '*' + s1_alignment
            s2_alignment = _s2[i] + s2_alignment
            i -= 1
        elif j > 0:  # left
            s1_alignment = _s1[j] + s1_alignment
            s2_alignment = ' ' + s2_alignment
            j -= 1
    return s1_alignment, s2_alignment
```

```
def backtrace():
    s1_alignment = ''
    s2_alignment = ''
    i = len(_s2) - 1
    j = len(_s1) - 1
    while i > 0 or j > 0:
        if i > 0 and j > 0:
            diag = _dp[i - 1][j - 1]
            up = _dp[i - 1][j]
            left = _dp[i][j - 1]
            if (_dp[i][j] - up) == GAP_PENALTY:
                s1_alignment = '*' + s1_alignment
                s2_alignment = _s2[i] + s2_alignment
                i -= 1
            elif (_dp[i][j] - diag) == h(_s2[i],_s1[j]):
                s1_alignment = _s1[j] + s1_alignment
                s2_alignment = _s2[i] + s2_alignment
                i -= 1
                j -= 1
            elif (_dp[i][j] - left) == GAP_PENALTY:
                s1_alignment = _s1[j] + s1_alignment
                s2_alignment = '*' + s2_alignment
                j -= 1
        elif i > 0:  # up
            s1_alignment = '*' + s1_alignment
            s2_alignment = _s2[i] + s2_alignment
            i -= 1
        elif j > 0:  # left
            s1_alignment = _s1[j] + s1_alignment
            s2_alignment = '*' + s2_alignment
            j -= 1
    return s1_alignment, s2_alignment

def lalign(s1,s2):
    set_globs(s1.upper(),s2.upper())
    print('Local DP matrix:')
    fill_l_dp_matrix()
    print_dp_matrix()
    return lbacktrace()

def galign(s1,s2):
    set_globs(s1.upper(),s2.upper())
    print('Global DP matrix:')
    fill_dp_matrix()
    print_dp_matrix()
    return backtrace()

r = 'BMPTSQCVNS'
l = 'vvtTSQalxc'

lalign1,lalign2 = lalign(r,l)
print(lalign1+'\n'+lalign2)
ma = MultipleAlignment()
```

```
ma.add_sequence(lalign1.upper())
ma.add_sequence(lalign2.upper())
print('Local Sum of Pairs = {0}'.format(
        ma.sum_aligned_char_scores(sum_of_pairs, h)))

galign1,galign2 = galign(r,l)
print(galign1+'\n'+galign2)
gma = MultipleAlignment()
gma.add_sequence(galign1.upper())
gma.add_sequence(galign2.upper())
print('Global Sum of Pairs = {0}'.format(
        gma.sum_aligned_char_scores(sum_of_pairs, h)))
```

```
Local DP matrix:
  *   B   M   P   T   S   Q   C   V   N   S
* 0   -4  -8 -12 -16 -20 -24 -28 -32 -36 -40
V -4  0   0   0   0   0   0   0   0   0   0
V -8  0  1.0  0  0.0  0   0   0  4.0 0.0  0
T-12  0   0  0.0 5.0 1.0  0   0  0.0 4.0 1.0
T-16  0   0   0  5.0 6.0 2.0  0  0.0 0.0 5.0
S-20  0   0   0  1.0 9.0 6.0 2.0  0  1.0 4.0
Q-24  0  0.0  0   0  5.0 14.010.06.0 2.0 1.0
A-28  0   0   0  0.0 1.0 10.014.010.06.0 3.0
L-32  0  2.0  0   0   0  6.0 10.015.011.07.0
X-36  0   0  0.0 0.0 0.0 2.0 6.0 11.014.011.0
C-40  0   0   0   0   0   0  11.07.0 10.013.0
BMPTSQCVNS
   TSQALXC
Local Sum of Pairs = 13.0
Global DP matrix:
  *   B   M   P   T   S   Q   C   V   N   S
* 0   -4  -8 -12 -16 -20 -24 -28 -32 -36 -40
V -4 -3.0-3.0-7.0-11.0-15.0-19.0-23.0-24.0-28.0-32.0
V -8 -7.0-2.0-5.0-7.0-11.0-15.0-19.0-19.0-23.0-27.0
T-12 -9.0-6.0-3.00.0 -4.0-8.0-12.0-16.0-19.0-22.0
T-16 -13.0-10.0-7.02.0 1.0 -3.0-7.0-11.0-15.0-18.0
S-20 -16.0-14.0-11.0-2.06.0 2.0 -2.0-6.0-10.0-11.0
Q-24 -20.0-16.0-15.0-6.02.0 11.07.0 3.0 -1.0-5.0
A-28 -24.0-20.0-17.0-10.0-2.07.0 11.07.0 3.0 0.0
L-32 -28.0-22.0-21.0-14.0-6.03.0 7.0 12.08.0 4.0
X-36 -32.0-26.0-24.0-18.0-10.0-1.03.0 8.0 11.08.0
C-40 -36.0-30.0-28.0-22.0-14.0-5.08.0 4.0 7.0 10.0
BMPTSQCVNS
VVTTSQALXC
Global Sum of Pairs = 10.0
```

a. 4) Discuss the alignment

The local alignment is able to outscore the global alignment as it can 'skip' aligning characters that would put the total score below 0. This is reflected in the sparseness of the DP matrix for the local alignment.

b. 1) a better global alignment as compared to a local alignment (using blosum62 scoring matrix), both 10 amino acids long.

```
from random import randint

SIGMA = 'ARNDCQEGHILKMFPSTWYVBZX'

def r_char():
    """
    Generates a random character
    :return: a random character from SIGMA
    """
    return SIGMA[randint(0,len(SIGMA)-1)]


def random_sequence(l):
    """
    Generates a random sequence
    :param l: length of random sequence to build
    :return: random sequence of length l
    """
    r_seq = ''
    for i in range(l):
        r_seq += r_char()
    return r_seq

def global_align_sequence(r_seq):
    """
    Matches every other character in r_seq
    :param r_seq: any random sequence
    :return: a sequence that will align globally to r_seq
    """
    g_seq = ''
    for i in range(len(r_seq)):
        if i%2 == 0:
            g_seq += r_seq[i]
        else:
            g_seq += r_char().lower()
    return g_seq

r = random_sequence(10)
g = global_align_sequence(r)

print('r: ' + r)
print('g: ' + g)
```

```
r: INHBRADLCD
g: IhHpRwDiCp
```

b. 2) Discuss criteria and any limitations in generating your sequences.

A pseudorandom number generator (PRNG) is used to select all characters for an initial 'random' sequence. The PRNG is reused to select every odd-index character of a second sequence of equal

length, while the even-index characters are selected by simply using the characters from the first.

b. 3) Align the two sequences using dynamic programming using the blosum62 scoring matrix. Show the DP matrix and the resultant alignment.

```
import math
from random import randint

SIGMA = 'ARNDCQEGHILKMFPSTWYVBZX'
ILLEGAL_CHARS = ' -\n'
GAP_PENALTY = -4 # taken from blosum62 matrix
BLOSUM62 = {frozenset({'R', 'P'}): -2.0, frozenset({'S', 'G'}): 0.0, frozenset({'T', 'K'}): -1.0,
            frozenset({'N', 'G'}): 0.0, frozenset({'Z', 'G'}): -2.0, frozenset({'P', 'B'}): -2.0,
            frozenset({'T', 'W'}): -2.0, frozenset({'N', 'D'}): 1.0, frozenset({'*', 'N'}): -4.0, froze
            frozenset({'Q', 'A'}): -1.0, frozenset({'N', 'K'}): 0.0, frozenset({'*', 'M'}): -4.0,
            frozenset({'N', 'A'}): -2.0, frozenset({'Q', 'P'}): -1.0, frozenset({'X', 'A'}): 0.0,
            frozenset({'Z', 'K'}): 1.0, frozenset({'*', 'K'}): -4.0, frozenset({'H', 'D'}): -1.0,
            frozenset({'C', 'D'}): -3.0, frozenset({'B', 'X'}): -1.0, frozenset({'Q', 'R'}): 1.0,
            frozenset({'W', 'P'}): -4.0, frozenset({'S', 'Y'}): -2.0, frozenset({'*', 'W'}): -4.0,
            frozenset({'W', 'D'}): -4.0, frozenset({'K', 'M'}): -1.0, frozenset({'I', 'V'}): 3.0,
            frozenset({'E', 'K'}): 1.0, frozenset({'C', 'Y'}): -2.0, frozenset({'R', 'D'}): -2.0,
            frozenset({'Z', 'B'}): 1.0, frozenset({'P', 'G'}): -2.0, frozenset({'Q', 'C'}): -3.0,
            frozenset({'R', 'S'}): -1.0, frozenset({'B', 'L'}): -4.0, frozenset({'R', 'Z'}): 0.0,
            frozenset({'I', 'F'}): 0.0, frozenset({'S', 'A'}): 1.0, frozenset({'I', 'B'}): -3.0,
            frozenset({'F', 'Y'}): 3.0, frozenset({'D', 'A'}): -2.0, frozenset({'E', 'C'}): -4.0,
            frozenset({'H', 'E'}): 0.0, frozenset({'I', 'C'}): -1.0, frozenset({'H', 'I'}): -3.0,
            frozenset({'Q', 'T'}): -1.0, frozenset({'V', 'D'}): -3.0, frozenset({'K', 'D'}): -1.0,
            frozenset({'Q', 'V'}): -2.0, frozenset({'C', 'B'}): -3.0, frozenset({'*', 'V'}): -4.0,
            frozenset({'X', 'Y'}): -1.0, frozenset({'F', 'V'}): -1.0, frozenset({'I', 'M'}): 1.0,
            frozenset({'P', 'M'}): -2.0, frozenset({'E', 'W'}): -3.0, frozenset({'W', 'A'}): -3.0,
            frozenset({'E', 'L'}): -3.0, frozenset({'T', 'Z'}): -1.0, frozenset({'*', 'E'}): -4.0,
            frozenset({'I', 'N'}): -3.0, frozenset({'E', 'Z'}): 4.0, frozenset({'H', 'A'}): -2.0,
            frozenset({'B', 'K'}): 0.0, frozenset({'M'}): 5.0, frozenset({'T', 'Y'}): -2.0, frozenset({
            frozenset({'K'}): 5.0, frozenset({'W', 'N'}): -4.0, frozenset({'I', 'X'}): -1.0, frozenset(
            frozenset({'N', 'L'}): -3.0, frozenset({'W', 'C'}): -2.0, frozenset({'H', 'W'}): -2.0,
            frozenset({'F', 'M'}): 0.0, frozenset({'S', 'D'}): 0.0, frozenset({'*', 'D'}): -4.0,
            frozenset({'P', 'X'}): -2.0, frozenset({'T', 'L'}): -1.0, frozenset({'L', 'M'}): 2.0,
            frozenset({'D', 'M'}): -3.0, frozenset({'*', 'T'}): -4.0, frozenset({'P', 'Y'}): -3.0,
            frozenset({'T', 'G'}): -2.0, frozenset({'R', 'Y'}): -2.0, frozenset({'I', 'P'}): -3.0,
            frozenset({'*', 'S'}): -4.0, frozenset({'S', 'W'}): -3.0, frozenset({'F', 'G'}): -3.0,
            frozenset({'I', 'T'}): -1.0, frozenset({'T', 'E'}): -1.0, frozenset({'G'}): 6.0, frozenset(
            frozenset({'W', 'Y'}): 2.0, frozenset({'V', 'M'}): 1.0, frozenset({'N', 'C'}): -3.0,
            frozenset({'V', 'L'}): 1.0, frozenset({'Q', 'Z'}): 3.0, frozenset({'R', 'G'}): -2.0,
            frozenset({'F', 'X'}): -1.0, frozenset({'W', 'B'}): -4.0, frozenset({'H', 'B'}): 0.0,
            frozenset({'V', 'A'}): 0.0, frozenset({'Z', 'Y'}): -2.0, frozenset({'W', 'L'}): -2.0,
            frozenset({'K', 'A'}): -1.0, frozenset({'T', 'A'}): 0.0, frozenset({'Y', 'D'}): -3.0,
            frozenset({'R', 'C'}): -3.0, frozenset({'X', 'M'}): -1.0, frozenset({'Q', 'L'}): -2.0,
            frozenset({'Q', 'D'}): 0.0, frozenset({'I', 'W'}): -3.0, frozenset({'E', 'D'}): 2.0,
            frozenset({'R', 'T'}): -1.0, frozenset({'Y', 'G'}): -3.0, frozenset({'S', 'E'}): 0.0,
            frozenset({'F', 'P'}): -4.0, frozenset({'S'}): 4.0, frozenset({'W'}): 11.0, frozenset({'C',
            frozenset({'T', 'B'}): -1.0, frozenset({'R', 'F'}): -3.0, frozenset({'H', 'K'}): -1.0,
            frozenset({'H', 'M'}): -2.0, frozenset({'C', 'M'}): -1.0, frozenset({'S', 'M'}): -1.0,
```

```
        frozenset({'Q', 'I'}): -3.0, frozenset({'R'}): 5.0, frozenset({'F', 'C'}): -2.0,
        frozenset({'H', 'F'}): -1.0, frozenset({'V', 'G'}): -3.0, frozenset({'Z', 'L'}): -3.0,
        frozenset({'W', 'V'}): -3.0, frozenset({'*', 'Z'}): -4.0, frozenset({'H', 'C'}): -3.0,
        frozenset({'N', 'V'}): -3.0, frozenset({'L', 'D'}): -4.0, frozenset({'R', 'V'}): -3.0,
        frozenset({'Q', 'Y'}): -1.0, frozenset({'S', 'L'}): -2.0, frozenset({'N', 'M'}): -2.0,
        frozenset({'Z', 'M'}): -1.0, frozenset({'F', 'L'}): 0.0, frozenset({'H', 'G'}): -2.0,
        frozenset({'C', 'G'}): -3.0, frozenset({'F', 'Z'}): -3.0, frozenset({'V', 'B'}): -3.0,
        frozenset({'H', 'L'}): -3.0, frozenset({'E'}): 5.0, frozenset({'*', 'C'}): -4.0,
        frozenset({'H', '*'}): -4.0, frozenset({'F', 'B'}): -3.0, frozenset({'E', 'A'}): -1.0,
        frozenset({'R', 'A'}): -1.0, frozenset({'H', 'R'}): 0.0, frozenset({'E', 'N'}): 0.0,
        frozenset({'T', 'V'}): 0.0, frozenset({'*', 'B'}): -4.0, frozenset({'G', 'A'}): 0.0,
        frozenset({'H', 'S'}): -1.0, frozenset({'Q', 'F'}): -3.0, frozenset({'N', 'P'}): -2.0,
        frozenset({'D'}): 6.0, frozenset({'Z', 'X'}): -1.0, frozenset({'F', 'W'}): 1.0, frozenset({
        frozenset({'X', 'D'}): -1.0, frozenset({'Z', 'V'}): -2.0, frozenset({'Q', '*'}): -4.0,
        frozenset({'Q', 'S'}): 0.0, frozenset({'L', 'K'}): -2.0, frozenset({'W', 'Z'}): -3.0,
        frozenset({'*', 'Y'}): -4.0, frozenset({'*', 'L'}): -4.0, frozenset({'Q'}): 5.0,
        frozenset({'L', 'A'}): -1.0, frozenset({'E', 'V'}): -2.0, frozenset({'I', 'Z'}): -3.0,
        frozenset({'I', 'K'}): -3.0, frozenset({'E', 'Y'}): -2.0, frozenset({'C', 'K'}): -3.0,
        frozenset({'I', 'Y'}): -1.0, frozenset({'*', 'G'}): -4.0, frozenset({'E', 'X'}): -1.0,
        frozenset({'S', 'X'}): 0.0, frozenset({'Q', 'X'}): -1.0, frozenset({'P', 'A'}): -1.0,
        frozenset({'F', 'K'}): -3.0, frozenset({'S', 'Z'}): 0.0, frozenset({'Z'}): 4.0, frozenset({
        frozenset({'Q', 'H'}): 0.0, frozenset({'*', 'F'}): -4.0, frozenset({'F', 'D'}): -3.0,
        frozenset({'T', 'D'}): -1.0, frozenset({'S', 'K'}): 0.0, frozenset({'I', '*'}): -4.0,
        frozenset({'S', 'T'}): 1.0, frozenset({'C', 'L'}): -1.0, frozenset({'E', 'P'}): -1.0,
        frozenset({'X'}): -1.0, frozenset({'R', 'X'}): -1.0, frozenset({'B', 'Y'}): -3.0,
        frozenset({'R', 'L'}): -2.0, frozenset({'E', 'M'}): -2.0, frozenset({'*', 'R'}): -4.0,
        frozenset({'H', 'V'}): -3.0, frozenset({'Q', 'W'}): -2.0, frozenset({'E', 'F'}): -3.0,
        frozenset({'Z', 'D'}): 1.0, frozenset({'P', 'V'}): -2.0, frozenset({'V', 'K'}): -2.0, froze
        frozenset({'R', 'W'}): -3.0, frozenset({'S', 'P'}): -1.0, frozenset({'Z', 'A'}): -1.0,
        frozenset({'P', 'D'}): -1.0, frozenset({'*', 'X'}): -4.0, frozenset({'V', 'Y'}): -1.0,
        frozenset({'X', 'L'}): -1.0, frozenset({'I', 'G'}): -4.0, frozenset({'E', 'G'}): -2.0,
        frozenset({'L', 'Y'}): -1.0, frozenset({'P', 'K'}): -1.0, frozenset({'R', 'M'}): -1.0,
        frozenset({'R', 'K'}): 2.0, frozenset({'W', 'M'}): -1.0, frozenset({'C'}): 9.0, frozenset({
        frozenset({'I', 'S'}): -2.0, frozenset({'S', 'B'}): 0.0, frozenset({'X', 'G'}): -1.0,
        frozenset({'V', 'X'}): -1.0, frozenset({'L', 'G'}): -4.0, frozenset({'T', 'C'}): -1.0,
        frozenset({'T'}): 5.0, frozenset({'Q', 'E'}): 2.0, frozenset({'B'}): 4.0, frozenset({'P'}):
        frozenset({'W', 'K'}): -3.0, frozenset({'I'}): 4.0, frozenset({'*'}): 1.0, frozenset({'R',
        frozenset({'T', 'F'}): -2.0, frozenset({'T', 'M'}): -1.0, frozenset({'Z', 'N'}): 0.0,
        frozenset({'I', 'E'}): -3.0, frozenset({'T', 'N'}): 0.0, frozenset({'S', 'F'}): -2.0,
        frozenset({'E', 'B'}): 1.0, frozenset({'Y', 'A'}): -2.0, frozenset({'W', 'G'}): -2.0,
        frozenset({'I', 'R'}): -3.0, frozenset({'R', 'E'}): 0.0, frozenset({'Y'}): 7.0, frozenset({
        frozenset({'T', 'X'}): 0.0, frozenset({'G', 'M'}): -3.0, frozenset({'K', 'G'}): -2.0,
        frozenset({'S', 'C'}): -1.0, frozenset({'Z', 'C'}): -3.0, frozenset({'H', 'Z'}): 0.0,
        frozenset({'Q', 'B'}): 0.0, frozenset({'C', 'V'}): -1.0, frozenset({'V'}): 4.0, frozenset({
        frozenset({'H', 'Y'}): 2.0, frozenset({'W', 'X'}): -2.0, frozenset({'Q', 'N'}): 0.0,
        frozenset({'R', 'N'}): 0.0, frozenset({'B', 'D'}): 4.0, frozenset({'H', 'X'}): -1.0,
        frozenset({'C', 'X'}): -2.0, frozenset({'*', 'A'}): -4.0, frozenset({'N', 'B'}): 3.0,
        frozenset({'B', 'G'}): -1.0, frozenset({'*', 'P'}): -4.0, frozenset({'K', 'Y'}): -2.0,
        frozenset({'Y', 'M'}): -1.0, frozenset({'P', 'L'}): -3.0, frozenset({'Q', 'K'}): 1.0,
        frozenset({'S', 'V'}): -2.0, frozenset({'F'}): 6.0, frozenset({'M', 'A'}): -1.0,
        frozenset({'F', 'A'}): -2.0, frozenset({'N', 'X'}): -1.0, frozenset({'T', 'P'}): -1.0,
        frozenset({'S', 'N'}): 1.0, frozenset({'Q', 'G'}): -2.0, frozenset({'B', 'A'}): -2.0,
        frozenset({'H', 'P'}): -2.0, frozenset({'C', 'P'}): -3.0, frozenset({'X', 'K'}): -1.0,
```

```
            frozenset({'Q', 'M'}): 0.0, frozenset({'Z', 'P'}): -1.0, frozenset({'B', 'M'}): -3.0,
            frozenset({'I', 'L'}): 2.0, frozenset({'H', 'T'}): -2.0}


class MultipleAlignment:
    """
    represents a simple multiple alignment
    """

    def __init__(self):
        self.sequence_matrix = []

    def add_sequence(self, sequence):
        """
        adds sequence characters to matrix, appending
        to existing values such that aligned characters
        will end up occupying the same index
        ex:
        'ZWAB'
        'B-ZX'
        will end up as
        ['ZB','W-','AZ','BX']
        :param sequence:
        """
        for idx, char in enumerate(sequence):
            if idx < len(self.sequence_matrix):
                self.sequence_matrix[idx] += char
            else:
                self.sequence_matrix.append(char)

    def sum_aligned_char_scores(self, scoring_function, *sf_args):
        """
        addition commutes. aligned characters of matrix
        are summed, simulating the summation of whichever
        scoring function
        :param scoring_function:
        :param sf_args:
        :return:
        """
        score_sum = 0.0
        for aligned_chars in self.sequence_matrix:
            if all(char not in ILLEGAL_CHARS for char in aligned_chars):
                score_sum += scoring_function(aligned_chars, sf_args)
        return score_sum

    def minimize_aligned_char_scores(self, scoring_function, *sf_args):
        """
        returns the minimum value returned by the
        scoring function for aligned characters
        :param scoring_function:
        :param sf_args:
        :return:
        """
```

```
        score_list = []
        for aligned_chars in self.sequence_matrix:
            score_list.append(scoring_function(aligned_chars, sf_args))
        return min(score_list)

def sum_of_pairs(aligned_chars, param_tuple):
    """Sum-of-Pairs (SP-score)
    Compeau, P, Pevzner, P,c Active Learning Publishers,
    Bioinformatics Algorithms: An Active Learning Approach,
    2nd Ed., Vol. I, p. 290
    :param aligned_chars: the characters in the alignment
    :param param_tuple: a tuple with a scoring matrix wrapper
    """
    matrix_wrapper = param_tuple[0]
    sp = 0.0
    for i in range(0, len(aligned_chars)):
        for j in range(i + 1, len(aligned_chars)):
            sp += matrix_wrapper(aligned_chars[i], aligned_chars[j])
    return sp + 0  # trick from http://goo.gl/8u8kQw

def h(char_i, char_j):
    """
    simply wraps dictionary
    """
    return BLOSUM62[frozenset([char_i, char_j])]

def set_globs(s1,s2):
    global _s1
    global _s2
    global _dp
    _s1 = '*' + s1
    _s2 = '*' + s2

    # create DP matrix with S1 + 1 columns and S2 + 1 rows
    _dp = [[0 for x in range(len(_s1) + 1)] for x in range(len(_s2) + 1)]

def fill_l_dp_matrix():
    global _dp
    for i in range(len(_s2)):
        for j in range(len(_s1)):
            if i > 0 and j > 0:
                _dp[i][j] = max(_dp[i - 1][j - 1] + h(_s2[i], _s1[j]),
                                _dp[i - 1][j] + GAP_PENALTY,
                                _dp[i][j - 1] + GAP_PENALTY,
                                0) # allow for local alignment
            elif i > 0:
                _dp[i][j] = _dp[i - 1][j] + GAP_PENALTY
            elif j > 0:
                _dp[i][j] = _dp[i][j - 1] + GAP_PENALTY
            else:
                _dp[i][j] = 0

def fill_dp_matrix():
```

```
    global _dp
    for i in range(len(_s2)):
        for j in range(len(_s1)):
            if i > 0 and j > 0:
                _dp[i][j] = max(_dp[i - 1][j - 1] + h(_s2[i], _s1[j]),
                                _dp[i - 1][j] + GAP_PENALTY,
                                _dp[i][j - 1] + GAP_PENALTY)
            elif i > 0:
                _dp[i][j] = _dp[i - 1][j] + GAP_PENALTY
            elif j > 0:
                _dp[i][j] = _dp[i][j - 1] + GAP_PENALTY
            else:
                _dp[i][j] = 0

def print_dp_matrix():
    for c in range(len(_s1)):
        print(' {0}'.format(_s1[c]).center(4, ' '), end='')
    print()
    for i in range(len(_s2)):
        print('{0}'.format(_s2[i]), end='')
        for j in range(len(_s1)):
            print('{0}'.format(_dp[i][j]).center(4, ' '), end='')
        print()

def lbacktrace():
    s1_alignment = ''
    s2_alignment = ''
    i = len(_s2) - 1
    j = len(_s1) - 1
    while i > 0 or j > 0:
        if i > 0 and j > 0:
            diag = _dp[i - 1][j - 1]
            up = _dp[i - 1][j]
            left = _dp[i][j - 1]
            if _dp[i][j] == 0:
                if len(s2_alignment.strip()) > 0: #done with local alignment, abort
                    i = 0
                else:
                    s1_alignment = _s1[j] + s1_alignment
                    s2_alignment = ' ' + s2_alignment
                    i -= 1
                    j -= 1
            elif (_dp[i][j] - diag) == h(_s2[i],_s1[j]):
                s1_alignment = _s1[j] + s1_alignment
                s2_alignment = _s2[i] + s2_alignment
                i -= 1
                j -= 1
            elif (_dp[i][j] - up) == GAP_PENALTY:
                s1_alignment = '*' + s1_alignment
                s2_alignment = _s2[i] + s2_alignment
                i -= 1
            elif (_dp[i][j] - left) == GAP_PENALTY:
                s1_alignment = _s1[j] + s1_alignment
```

```
                s2_alignment = '*' + s2_alignment
                j -= 1
        elif i > 0:  # up
            s1_alignment = '*' + s1_alignment
            s2_alignment = _s2[i] + s2_alignment
            i -= 1
        elif j > 0:  # left
            s1_alignment = _s1[j] + s1_alignment
            s2_alignment = ' ' + s2_alignment
            j -= 1
    return s1_alignment, s2_alignment

def backtrace():
    s1_alignment = ''
    s2_alignment = ''
    i = len(_s2) - 1
    j = len(_s1) - 1
    while i > 0 or j > 0:
        if i > 0 and j > 0:
            diag = _dp[i - 1][j - 1]
            up = _dp[i - 1][j]
            left = _dp[i][j - 1]
            if (_dp[i][j] - up) == GAP_PENALTY:
                s1_alignment = '*' + s1_alignment
                s2_alignment = _s2[i] + s2_alignment
                i -= 1
            elif (_dp[i][j] - diag) == h(_s2[i],_s1[j]):
                s1_alignment = _s1[j] + s1_alignment
                s2_alignment = _s2[i] + s2_alignment
                i -= 1
                j -= 1
            elif (_dp[i][j] - left) == GAP_PENALTY:
                s1_alignment = _s1[j] + s1_alignment
                s2_alignment = '*' + s2_alignment
                j -= 1
        elif i > 0:  # up
            s1_alignment = '*' + s1_alignment
            s2_alignment = _s2[i] + s2_alignment
            i -= 1
        elif j > 0:  # left
            s1_alignment = _s1[j] + s1_alignment
            s2_alignment = '*' + s2_alignment
            j -= 1
    return s1_alignment, s2_alignment

def lalign(s1,s2):
    set_globs(s1.upper(),s2.upper())
    print('Local DP matrix:')
    fill_l_dp_matrix()
    print_dp_matrix()
    return lbacktrace()

def galign(s1,s2):
```

```
    set_globs(s1.upper(),s2.upper())
    print('Global DP matrix:')
    fill_dp_matrix()
    print_dp_matrix()
    return backtrace()

r = 'INHBRADLCD'
l = 'IhHpRwDiCp'

lalign1,lalign2 = lalign(r,l)
print(lalign1+'\n'+lalign2)
ma = MultipleAlignment()
ma.add_sequence(lalign1.upper())
ma.add_sequence(lalign2.upper())
print('Local Sum of Pairs = {0}'.format(
        ma.sum_aligned_char_scores(sum_of_pairs, h)))

galign1,galign2 = galign(r,l)
print(galign1+'\n'+galign2)
gma = MultipleAlignment()
gma.add_sequence(galign1.upper())
gma.add_sequence(galign2.upper())
print('Global Sum of Pairs = {0}'.format(
        gma.sum_aligned_char_scores(sum_of_pairs, h)))
```

```
Local DP matrix:
  *   I   N   H   B   R   A   D   L   C   D
* 0   -4  -8 -12 -16 -20 -24 -28 -32 -36 -40
I -4 4.0 0.0  0   0   0   0   0   0   0   0
H -8 0.0 5.0 8.0 4.0 0.0  0   0   0   0   0
H-12  0  1.0 13.09.0 5.0 1.0  0   0   0   0
P-16  0   0  9.0 11.07.0 4.0 0.0  0   0   0
R-20  0  0.0 5.0 8.0 16.012.08.0 4.0 0.0  0
W-24  0   0  1.0 4.0 12.013.09.0 6.0 2.0  0
D-28  0  1.0  0  5.0 8.0 10.019.015.011.08.0
I-32  0   0   0  1.0 4.0 7.0 15.021.017.013.0
C-36  0   0   0   0  0.0 4.0 11.017.030.026.0
P-40  0   0   0   0   0  0.0 7.0 13.026.029.0
INHBRADLCD
IHHPRWDICP
Local Sum of Pairs = 29.0
Global DP matrix:
  *   I   N   H   B   R   A   D   L   C   D
* 0   -4  -8 -12 -16 -20 -24 -28 -32 -36 -40
I -4 4.0 0.0 -4.0-8.0-12.0-16.0-20.0-24.0-28.0-32.0
H -8 0.0 5.0 8.0 4.0 0.0 -4.0-8.0-12.0-16.0-20.0
H-12 -4.01.0 13.09.0 5.0 1.0 -3.0-7.0-11.0-15.0
P-16 -8.0-3.09.0 11.07.0 4.0 0.0 -4.0-8.0-12.0
R-20 -12.0-7.05.0 8.0 16.012.08.0 4.0 0.0 -4.0
W-24 -16.0-11.01.0 4.0 12.013.09.0 6.0 2.0 -2.0
D-28 -20.0-15.0-3.05.0 8.0 10.019.015.011.08.0
I-32 -24.0-19.0-7.01.0 4.0 7.0 15.021.017.013.0
C-36 -28.0-23.0-11.0-3.00.0 4.0 11.017.030.026.0
```

```
P-40 -32.0-27.0-15.0-7.0-4.00.0 7.0 13.026.029.0
INHBRADLCD
IHHPRWDICP
Global Sum of Pairs = 29.0
```

b. 4) Discuss the alignment.

> This sequence, by being 'pinned' by every other matching characters has no advantage in aligning locally. The Sum of Pairs scores are equal for both local and global alignments. This is a result of the worst-case local alignment score having an upper bound of the global alignment score.

5: Describe a scenario where a researcher would be interested in investigating both the local and global alignment of two sequences.

> A researcher may be interested in comparing the evolutionary history of part of a chromosome recovered from an ancient hominid, C, in some modern human genome, M. A global alignment of C->M would show which area, Mc, of M resembled C in its entirety. A local alignment of parts of C, Cp, onto M, could reveal gene duplication or chromosome structure events if Cp was found to align well outside of Mc.

6: Determine if two DNA strings are circular rotations of each other. For example TTGATC is a circular rotation of ATCTTG. You must state all assumptions.

```python
# This code assumes D1 and D2 can match exactly (no
# indels, different lengths, mismatches, etc).

D1 = 'TTGATC'
D2 = 'ATCTTG'


def circular_suffix_array(dna_string):
    """
    No need to add a terminal character as circles don't
    end.
    :param dna_string: a dna string
    :return: a circular suffix array of dna_string
    """
    csa = []
    [csa.append(dna_string[substring:] +
                dna_string[:substring]) for substring in range(len(dna_string))]
    return sorted(csa)

def bwt(dna_string):
    """
    Just ripping the last characters off the circular
    suffix array.
    :param dna_string: a dna string
    :return: a BWT string representation
    """
    csa = circular_suffix_array(dna_string)
```

```
    return ''.join([suffix[-1:] for suffix in csa])

def rotations_of_each_other(d1,d2):
    """
    Checking for matching BWT's
    :param d1: a dna string to compare
    :param d2: a dna string to compare
    :return: True if d1,d2 are rotations, else False
    """
    return bwt(d1) == bwt(d2)

print(rotations_of_each_other(D1,D2))
```

```
True
```

7: Given a suffix tree of S, count the number of distinct substrings of S. This should run in O(n).

```
#representing the suffix tree for AATATT as a list of nested tuples
S = [
    ('T',
     (('','ATT$'),
      ('','T$'),
      ('','$'))
    ),
    ('A',
     (('','ATATT$'),
     ('T',
      (('','ATT$'),
      ('','T$'))))
    ),
    ('','$')]

L = 0 # label
D = 1 # data

def terminal(node):
    """
    # is this a terminal node?
    :param node: a node in a suffix tree
    :return: True if node is terminal, else False
    """
    return node[L] == ''

def distinct_substr_counter(st):
    """
    Classic BFS w inner node & terminal node length counting.
    By summing the lengths of the inner nodes of our tree and
    our terminal (leaf) nodes, we are able to count the distinct
    substrings of our suffix tree. This leverages the non-redundant
    structure of the tree and is able to achieve a runtime of O(n).
```

```
    """
    count = 0
    for node in st: # bound by 2 x |S| (Bioinformatics Algorithms Volume 2, p. 131)
        if terminal(node):
            count += len(node[D]) - 1
        else:
            count += len(node[L])
            """
            I'm unsure how Python handles this but it can be performed in
            constant time in a language that allows direct memory access (C/C++).
            We just need to append a reference to this node to the list we're
            iterating over.
            """
            [st.append(x) for x in node[D]]
    return count

print(distinct_substr_counter(S))
```

```
16
```