Problem Set 3 answers

Due Tuesday, January 31, at the **beginning** of class. Assignments turned in more than 10 minutes after the beginning of class will be penalized. The Python problems may take you a long time - do not procrastinate. Data files to use in your programs are separately linked on web page: chr21.txt and blastn_OUT.txt.

1. (10 points) De novo genome assembly typically starts with a huge number of relatively short nucleotide sequences. Using methods we have talked about in class, propose a sensible approach to determining which reads have sequence overlaps. A few sentences will suffice – you don't have to be very technical. (you can assume no sequencing errors and no long repeat sequences in the source genome)

Conceptually, *de novo* genome assembly involves joining overlapping reads iteratively until it's no longer possible (because the entire genome/chromosome has been assembled, or upon encountering a repeat larger than any of the short sequences). This is a similar (albeit more involved) problem to BLAST, and the heuristic approach in BLAST is helpful here as well – using seed sequences to prioritize the search space. We can also use a greedy algorithm, like in tree algorithms like UPGMA, to further speed up the process. We can find the best overlapping pair of sequences, merge those sequences, and repeat.

Python tips:

- Be sure to run your program - even experienced programmers make minor errors
- Be sure your program runs on cases other than the example shown
- Build your program step by step, printing out lots of intermediate results (or parts of them - some of these problems work on very large files). At the end just remove or comment out the intermediate printing steps.
- Always keep in your head what type of object to which each variable refers. If you have trouble with this, then put the type into the variable name (e.g. listOfStrings or myIntAsString)
- Stretch yourself by solving a more complex version of one or two of the problems or write a program to solve some problem you've faced in your own research (not required).
- When you get to the end, pause and reflect on how you can now do fairly sophisticated things with large data sets.

2. (15 points) Write a program `mutants.py` that takes a DNA sequence and output file name as command-line arguments and creates a file of that name containing all possible mutant sequences produced by a single base substitution from the query sequence. Each line should contain one mutant sequence (in any order), and the query sequence should not be in the output file.

```
>python mutants.py ACTGAC mutants.txt
>cat mutants.txt
CCTGAC
TCTGAC
GCTGAC
AATGAC
ATTGAC
AGTGAC
ACAGAC
ACCGAC
ACGGAC
ACTAAC
ACTCAC
ACTTAC
ACTGCC
ACTGTC
ACTGGC
ACTGAA
ACTGAT
ACTGAG
```

This problem only requires for loops, if statements, and string/list operations – dictionaries are not particularly useful. Many of you had a lot of nearly repeated code, considering each nucleotide separately (if base == "A", …) . While this works ok for DNA, it would be much less ok for proteins! Remember that if you have a lot of similar code, it means you can probably use a loop (combined with if statements) instead.

```
import sys
seq = sys.argv[1]
outf = open(sys.argv[2],'w')
nts = ('A', 'C', 'T', 'G')
# loop through positions in seq
for i in range(len(seq)):
   # loop through all possible nucleotides
   for nt in nts:
       newseq = list(seq) # make list, which is mutable
       if (seq[i] != nt): # exclude  the original sequence
           newseq[i] = nt # make mutation
           outf.write(''.join(newseq) + '\n') # print mutant
outf.close()
```

3. (20 points) Next-generation DNA sequencers can process several samples in parallel, each identified by a "barcode" sequence, which are then bioinformatically split into separate data files. Write a program `filter_by_barcode.py` that takes three command-line arguments – a query barcode sequence and two files containing the same number of DNA sequences, where the first (e.g. `reads.txt`) contains sequencing reads

of interest and the second (e.g. `barcodes.txt`) contains corresponding barcode sequences – and prints the sequences from the reads file that have corresponding barcode sequences that match the query. All sequencing reads and barcodes should each be the same length. Make sure to only load one sequencing read-barcode pair at a time, to ensure that the program can deal with extremely large files.

```
>cat reads.txt
AACACCAGTATCATCT
CATTAGATCGGATCTA
GAAGTCTACCCCTATC
TTAGGCCCTCTACGGT
>cat barcodes.txt
ACTGGT
CTAGAC
AGGTTT
TACCTG
>python filter_by_barcode.py AGGTTT reads.txt barcodes.txt
GAAGTCTACCCCTATC
```

To be able to deal with large files, it's best to deal with one line (or pair of lines) at a time. To read the two files simultaneously, we can simply loop through one file and read one line of the second file inside that loop. Alternatively, you could make a list of the line numbers of barcodes matching the query, and extract those lines from the reads file … at the cost of storing that list of line numbers in memory.

```
import sys
query = sys.argv[1]
reads = open(sys.argv[2])
barcodes = open(sys.argv[3])
# get lines of reads file one at a time
for read in reads:
   # for each read, get one line from the barcode file
   barcode = barcodes.readline().strip()
   # if the barcode matches the query, print the read
   if barcode == query:
          print read.strip()
```

4. (20 points). DNA sequencers can produce errors, even while sequencing barcodes. However, if only a small number of barcodes are expected, barcode sequencing reads with few errors can still be matched to the correct sample. For example, if you have two samples with expected barcodes CTAGAC and AGGTTT, you can infer that a barcode sequence AGGTAT should be matched to the AGGTTT sample since it is 1 substitution (single base difference) away, compared to 5 substitutions away from CTAGAC. Modify your program from the previous problem to include sequencing reads with barcodes that have up to 1 substitution from the query barcode.

```
>python filter_by_barcode2.py AGGTAT reads.txt barcodes.txt
```

```
GAAGTCTACCCCTATC
```

Some of you tried to reuse code from problem 2, comparing the barcode to all sequences 1 substitution away from the query. However, it's much simpler to compare the query to the barcode one letter at a time, using a loop.

```
import sys
query = sys.argv[1]
reads = open(sys.argv[2])
barcodes = open(sys.argv[3])
for read in reads:
   barcode = barcodes.readline().strip()
   diff = 0 # number of base differences
   for i in range(len(query)): # assumes query is at least
as long as barcode
       if barcode[i] != query[i]: # compare 1 pair of bases
           diff += 1
   if diff <= 1: # only print read if within 1 substitution
          print read.strip()
```

5. (25 points) Write a program `find_seq.py` that finds all the positions of exact matches on human chromosome 21 (chr21.txt) for a DNA sequence given as a command-line argument and prints each position and the total number of matches. Make sure that the search doesn't depend on the case of the query or the sequence in the file. TIPS: You don't need to use lists to solve the problem. Remember that string1.find(string2, start) returns the first position where string2 appears in string1 after start position in string1, and that if string2 is not found it returns -1 (a common way of indicating not found or failed).

```
>python find_seq.py GATTGATGATA
1725839
5312484
7185252
8417800
8639981
8946117
11518008
11582415
11814084
14410790
16307228
19025838
22553983
13 matches
```

```
import sys
f = open("chr21.txt")
```

```
# load sequence as upper-case string
seq = ""
for line in f:
   seq += line.strip().upper()
f.close()

count = 0 # number of matches
p = seq.find(sys.argv[1]) # look for first match
while p != -1: # until there are no more matches
   print p + 1 # print 1-indexed position
   count += 1 # increment match count
   p = seq.find(sys.argv[1],p+1) # look for next match

print "%d matches" % count
```

6. (20 points) The file blastn_OUT.txt contains the text output from a blastn search (slightly edited for clarity). Look at the file and notice that it gives a series of alignments, each preceded by three lines that describe general values for the alignment (Score = etc.). Write a program `blastn_parse.py` that reads a blastn text output file and lists the alignment score and E-value (labeled Expect in the output) for each alignment, one per line.

```
>python blastn_parse.py blastn_OUT.txt
Score 1742 bits, E-value 0.0
Score 48.1 bits, E-value 7e-004
Score 44.1 bits, E-value 0.011
Score 44.1 bits, E-value 0.011
Score 42.1 bits, E-value 0.045
Score 40.1 bits, E-value 0.18
Score 40.1 bits, E-value 0.18
Score 40.1 bits, E-value 0.18
Score 40.1 bits, E-value 0.18
Score 40.1 bits, E-value 0.18
Score 38.2 bits, E-value 0.70
Score 38.2 bits, E-value 0.70
Score 38.2 bits, E-value 0.70
Score 38.2 bits, E-value 0.70
Score 38.2 bits, E-value 0.70
Score 38.2 bits, E-value 0.70
Score 38.2 bits, E-value 0.70
Score 38.2 bits, E-value 0.70
Score 38.2 bits, E-value 0.70
Score 38.2 bits, E-value 0.70
Score 38.2 bits, E-value 0.70
Score 38.2 bits, E-value 0.70
```

```
import sys
f = open(sys.argv[1])
for line in f:
   line = line.strip()
   if line.startswith("Score"):
       fields = line.split()
       print fields[0], fields[2], fields[3] + ", E-value",
fields[7]
f.close()
```

Challenge problem 1. Assuming that you have obtained the values of mu and lambda for the blast run in problem 6, write a program that makes the same output but adds an entry for the pair-alignment P-value. Give mu and lambda as command-line arguments.

```
>python blastn_compute_parse.py blastn_OUT.txt 25.0 0.79
Score 42.1 bits, E-value 0.045, P-value foo
etc.
```

```
import sys
import math

# arguments
f = open(sys.argv[1]) # blastn output file
mu = float(sys.argv[2]) # mu
lam = float(sys.argv[3]) # lambda

for line in f:
   line = line.strip()
   if line.startswith("Score"):
       fields = line.split()
       score = fields[2]
       print "Score %s bits, E-value %s, P-value %.2e" %
(score, fields[7], 1-math.exp(-math.exp(-
lam*(float(fields[2]) - mu))))
f.close()
```

Challenge problem 2. Write a program that fills an M x N 2-dimensional list with random integer values between -100 and 100 and writes them to a file. The values M and N should be command line arguments. TIP: the `random` module contains functions related to random numbers - use the python docs or google to figure out how to use it. Confirm that it produces a different matrix every time you run it.

```
>python rand_mat.py 3 3 result.txt
>cat result.txt
-3  12  81
```

```
-27  -5  77
19  -44  34
```

```
import sys
import random

# arguments
m = int(sys.argv[1]) # rows in array
n = int(sys.argv[2]) # columns in array
outf = open(sys.argv[3],'w') # output file

# generate M x N random array and store in 2D list
array = []
for i in range(m):
   row = []
   for j in range(n):
       row.append(random.randint(-100,100))
   array.append(row)

# print array
for i in range(len(array)):
   # convert row from ints to strings
   strrow = []
   for j in range(len(array[i])):
       strrow.append(str(array[i][j]))
   # print row
   outf.write("\t".join(strrow) + "\n")
outf.close()
```