Problem Set 1 Answers, GS559, Winter 2016

Note - for the programming assignments throughout my part of the course, there are many variations that will work - as long as your code works it is fine. Later on in the course we may start grading based on having compact efficient code.

1) Alanine has the smallest variance in scores (this is variance in the usual statistical sense). This suggests that during evolution, alanine tends to be relatively readily changed to and from other amino acids, so that these changes should be neither rewarded nor penalized very heavily (relative to other amino acids). This make sense biochemically as well, since alanine has perhaps the most "generic" side chain – a single methyl group. Tryptophan has the largest variance in scores for exactly the opposite reasons – it tends to be the most conserved amino acid and biochemically has a very large and chemically unique side chain.

2) Blosum62, linear gap -4 score sum 36

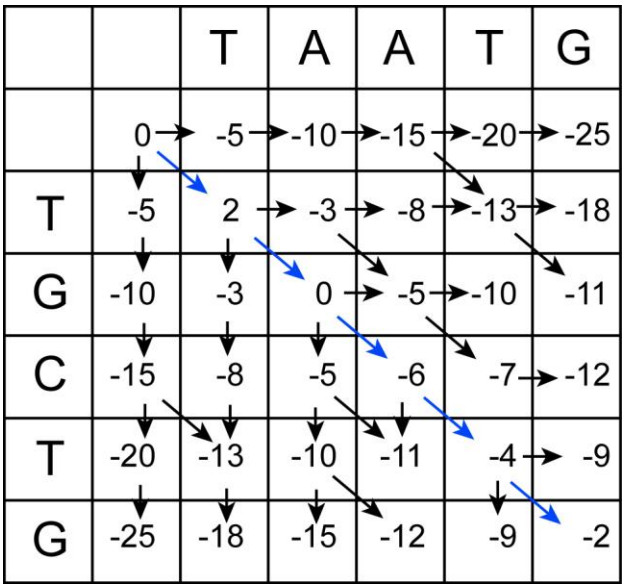
```
R L I N L M P - - - - K V L A R E Y K N Y
Q F F P L M P P A P Y K I L A K D F E N Y
+1 +0 +0 -2 +4 +5 +7 -4 -4 -4 -4 +5 +3 +4 +4 +2 +2 +3 +1 +6 +7
```

Blosum40, gap open -9, gap extend -1 score sum 66

```
R L I N L M P - - - - K V L A R E Y K N Y
Q F F P L M P P A P Y K I L A K D F E N Y
+1 +0 -1 -4 +6 +9 +12 -9 -1 -1 -1 +8 +4 +6 +7 +3 +2 +4 +1 +9 +11
```

You see what a pain in the neck this is to do by hand!?

3) Okay, so this is an even bigger pain in the neck to do by hand.



The optimal global alignment has score -2. The traceback arrows used for the alignment are shown in blue and the best alignment is gap free:

TGCTG
TAATG

Programming tips:
- try out syntax in the python interpreter
- build the program step by step, printing the output at each step so you are sure it is working correctly
- use sensible variable names

Some of the long lines below are wrapped - in the python program these have to be on a single line or on multiple lines each ending with

a single '\' character (indicates that the next line should be read as part of the previous line). There are many details that can be changed – e.g. many of you assigned variable names before printing, which is fine.

4) get-three-args1.py :

```
import sys
print sys.argv[1].upper()
print sys.argv[2].upper()
print sys.argv[3].upper()
```

5) get-three-args2.py :

```
import sys
print sys.argv[1].upper() + sys.argv[2].upper() + sys.argv[3].upper()
# note that if you put commas between the strings instead of
concatenating you get spaces in the output
```

6) get-subsequence.py :

```
import sys
seq = sys.argv[1]
start = int(sys.argv[2]) - 1  # the -1 accounts for the fact that
python
# starts at index 0
# of course you could do the -1 on the last line instead
end = int(sys.argv[3])
print seq[start:end]
```

7) count-substrings.py :

```
import sys
target = sys.argv[1]
query = sys.argv[2]
n = target.count(query)
print "The sequence " + query + " appears in the sequence " + target
+ " " + str(n) + " times."
```

8) challenge questions (these are a lot easier with methods we hadn't covered yet, which my answers use, but you can solve these using only the methods we had covered)

General tip - if you want to work with a sequence that won't be changed, you are probably best off representing it as a string object (which is immutable). If you want to work with a mutable sequence, you generally want to convert it to a list of characters (a list of one-character strings).

reverse.py :

```
import sys
slist = list(sys.argv[1])  # get the string argument as a list so
that we can use reverse()
slist.reverse()            # reverse the order of elements in the
list
s = ''.join(slist)         # convert back to a string with no
separator
print s
```

reverse-complement1.py :

```
import sys
slist = list(sys.argv[1].upper())
slist.reverse()
for i in range(len(slist)):
   if (slist[i] == 'A'):
        slist[i] = 'T'
   elif (slist[i] == 'C'):
        slist[i] = 'G'
   elif (slist[i] == 'G'):
        slist[i] = 'C'
   elif (slist[i] == 'T'):
        slist[i] = 'A'
s = ''.join(slist)
print s
```

An alternative approach to above is to use strings directly and convert all the 'A' to 'T', then all the 'C' to 'G' etc., but in this case you need to

use what you might call placeholders or dummy values: e.g. first change all 'A' to '1', all 'C' to '2', etc., then convert all '1' to 'T', all '2' to 'G', etc.

reverse-complement2.py is the same except don't convert to uppercase and add 4 additional conversions to the loop ('a' to 't' etc.)

reverse-complement3.py is the same as reverse-complement2.py except you add at end of the loop:

```
   else:
        print "Input error: character " + slist[i] + " not a valid
nucleotide"
```

...the output is a little cleaner if you include a "flag" that tells the program not to print the normal output if it encountered an invalid character. Flags are often handy. Here is the final version:

```
import sys
slist = list(sys.argv[1])
slist.reverse()
all_valid = True                # my flag for all valid characters
for i in range(len(slist)):
   if (slist[i] == 'A'):
        slist[i] = 'T'
   elif (slist[i] == 'C'):
        slist[i] = 'G'
   elif (slist[i] == 'G'):
        slist[i] = 'C'
   elif (slist[i] == 'T'):
        slist[i] = 'A'
   elif (slist[i] == 'a'):
        slist[i] = 't'
   elif (slist[i] == 'c'):
        slist[i] = 'g'
   elif (slist[i] == 'g'):
        slist[i] = 'c'
   elif (slist[i] == 't'):
        slist[i] = 'a'
   else:                    # if the loop reaches this point there is
an invalid character
        print "Input error: character " + slist[i] + " not a valid
```

```
nucleotide"
        all_valid = False    # change the flag to False
        break                # there is no need to continue the loop
if (all_valid):
   s = ''.join(slist)
   print s
```

You could make this even more useful to the user if you also printed the positions of all the invalid characters.

By the way, we'll learn a much more compact way to code this when we get to dictionaries.