**Problem Set #2 Answers**

The first three questions pertain to the following DP matrix:

| | | | A | | C | | D | | E | | F |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | → | -2 | → | -4 | → | -6 | → | -8 | → | -10 |
| | ↓ | ↘ | | ↘ | | ↘ | | ↘ | | | |
| G | -2 | | 7 | → | 5 | | 12 | → | 10 | → | 8 |
| | ↓ | | ↓ | ↘ | | ↘ | | | | | |
| H | -4 | | 5 | | 9 | | 14 | → | 12 | → | 10 |
| | ↓ | | ↓ | | ↓ | ↘ | | ↘ | | ↘ | |
| I | -6 | | 3 | | 7 | | 20 | → | 18 | | 21 |
| | ↓ | | ↓ | ↘ | | | ↓ | | ↓ | ↘ | |
| K | -8 | | 1 | | 12 | | 18 | → | 16 | | 24 |

1. (5 points) Was this DP matrix generated by the Smith-Waterman or Needleman-Wunsch algorithm? How do you know?

Needleman-Wunsch, because Smith-Waterman never has negative scores.

2. (5 points) For this DP matrix, is the gap penalty linear or affine? Give the value(s).

Linear, which you can easily see from the uniform steps of 2 in the top row or first column. Value is -2.

3. (10 points) Draw an empty amino acid substitution matrix, and fill in as many values as you can, based on the above DP matrix.

Some of you interpreted this as applying to the "standard" amino acid matrix, which is fine. Below I just give the characters that are actually part of the sequences. We can tell the score for every pair that aligns (all the diagonals), and to get the score we look for the difference between the score before and after the diagonal.

| | A | C | D | E | F | G | H | I | K |
|---|---|---|---|---|---|---|---|---|---|
| A | | | | | | 7 | | | |
| C | | | | | | 7 | 2 | | 9 |
| D | | | | | | 16 | 9 | 11 | |
| E | | | | | | 16 | | 4 | |
| F | | | | | | | | 9 | 6 |
| G | 7 | 7 | 16 | 16 | | | | | |
| H | | 2 | 9 | | | | | | |
| I | | | 11 | 4 | 9 | | | | |
| K | | 9 | | | 6 | | | | |

4. (10 points) Explain how blast speeds up finding sequences related to a query.

The query sequence is broken into small words (usually 3 residues long for proteins), which act as seeds for searches. The target dataset is pre-indexed for all positions that have a high-scoring match to each possible word. Each high scoring match is then extended using local alignment, continuing until the alignment score drops below a threshold. This allows the search to be limited to sequences near high scoring regions, thereby speeding up the search.

5. (5 points) Qualitatively describe how decreasing or increasing the blast word length would change the speed and false-negative rate for blast searches (for simplicity, assume alignments start only from exact word matches).

Decreasing the word length makes the search slower because more alignments need to be tested and the alignment is the slow part of blast. However, it will make the search more sensitive (fewer false negatives) because among those extra attempted alignments might be correct ones that would be missed with longer word length (the target fails to have any long identical matches with the query, even though overall it has substantial similarity). Remember that the word indexing is done once BEFORE any searching, so the time to make the index is not relevant (it is pretty fast anyway).

Some of you got this exactly reversed, though nearly everyone realized there was a tradeoff. If you think about the extreme cases, it might help clarify the logic. If the word size is 1 hardly anything will ever be missed (anything with even one aligned residue identity will be found), but there will be little or no speed up because alignments will be tried starting any place there is even a single match. If the search word is very long, very few alignments will be attempted but many real alignments will be missed because they don't happen to include a long stretch of identical residues.

Remember that the alignments themselves are not influenced by the word matching – they are done by standard local alignment dynamic programming.

6. (10 points) Write a program copy-file.py that copies a given file. For example, if you have a file called `seq1.txt` that contains one line ("GATCCAT"), then you could create a copy of this file called `seq2.txt` as follows:

```
import sys
infile = open(sys.argv[1], "r")
outfile = open(sys.argv[2], "w")
outfile.write(infile.read())
infile.close()
outfile.close()
```

Alternatively, if you don't worry about closing the files (they get closed automatically when this very simple program exits):

```
import sys
open(sys.argv[2], "w").write(open(sys.argv[1], "r").read())
```

This is very compact, but it is a lot harder to comprehend the code.

7. (10 points) Write a program reverse-lines.py that reads in the contents of a file, and prints out the lines in reverse order.

```
import sys
infile = open(sys.argv[1], "r")
lineList = infile.readlines()
lineList.reverse() # works in place, elements are now in reverse order
print "".join(lineList)
infile.close()
```

Alternatively, using a loop:

```
import sys
infile = open(sys.argv[1], "r")
lineList = infile.readlines()
index = len(lineList) - 1
for foo in range(len(lineList)): # I used foo to indicate that I do not use
the variable
      print lineList[index].strip()
      index -= 1
```

or a little nicer using range decrement:

```
import sys
infile = open(sys.argv[1], "r")
lineList = infile.readlines()
for index in range(len(lineList), 0, -1):
      print lineList[index-1].strip()
```

8. (10 points) Write a program split-number.py that reads a real number from the command line and prints its integer part on one line, followed by its decimal part (i.e., the digits after the decimal point) on a second line. For the decimal part, print no more than 6 digits after the decimal, but do not print trailing zeroes.

```
import sys
val = sys.argv[1]
print "%d" % float(val)
index = val.find(".")
val = val[index + 1: index + 7]
print val.rstrip("0") # lstrip and rstrip are like strip but only one end
```

A nice way to do this with number formatting alone, which comes close to the exact answer (though the second output line is 0.#### rather than just ####):

```
import sys
val = float(sys.argv[1])
remainder = val - int(val) # this leaves just the part right of the decimal
print "%d" % val
print "%.6g" % remainder
```

9. (10 points) Write a program format-number.py that takes as input two arguments: a number and a format, where the format is either integer, real or scientific. Print the given number in the requested format, and print an error if an invalid format string is provided.

```
import sys
val = float(sys.argv[1])
format = sys.argv[2]
if (format == "integer"):
      print "%d" % val
elif (format == "real"):
      print "%f" % val
elif (format == "scientific"):
      print "%e" % val
else:
      print "Invalid format:", format
```

10. (10 points) Write a program merge-lines.py that reads a file and prints the contents of the file all on one line (no white space, no newlines).

NOTE - since I didn't specify, some of you solved this using a specific file hard-coded in the program, which is fine. This solution takes the file name from the command-line. I didn't detract for not eliminating tabs (but they are also common white-space). Many of you only eliminated newlines at the end of lines, which isn't enough - you need to eliminate internal spaces as well.

```
import sys
f = open(sys.argv[1])
str = f.read()
f.close()
str = str.replace("\n","").replace("\t","").replace(" ","")
print str
```

11. (15 points) Write a program `read_matrix.py` that opens a file like `matrix.txt`, stores the entries in a 2-dimensional list and prints the matrix out on the screen. Don't just read the lines and spit them back - you must store the values in a 2-dimensional list of numbers first (as if you were going to use them to score an alignment). The format of the file is tab-delimited text, with one integer value in each data field and one row of the matrix on each line. Don't worry about getting the output to look pretty, just have it be readable. Make sure your program works on ANY file with the format of matrix.txt regardless of how many rows and columns there are.

```
import sys
openFile = open(sys.argv[1])
multiplier = int(sys.argv[2])
matrix = [] # initialize list to store rows of input file, each a list
for line in openFile:
      row = line.strip().split('\t') # list of entries in row
      matrix.append(row)
openFile.close()
# print the matrix
for row in matrix:
      print '\t'.join(row)
```

How would you change your program to print each matrix value multiplied by a command-line specified integer? (you don't need to write the program, just indicate the changes)

```
import sys
openFile = open(sys.argv[1])
multiplier = int(sys.argv[2])
matrix = [] # initialize list to store rows of input file, each a list
for line in openFile:
      row = []
      for value in line.strip().split('\t'):
            row.append(str(int(value) * multiplier))
      matrix.append(row)
openFile.close()
for row in matrix:
      print '\t'.join(row)
```

12. Challenge problem (solution to first part). Read a sequence from a file (where the sequence may be on one or many lines) and randomize (shuffle) the order of residues in the sequence.

```
import sys
import random
seq = open(sys.argv[1], "r").read()
seq = seq.replace("\n", "")
residueList = list(seq) # explode the string into a list of characters
random.shuffle(residueList) # shuffle the order of the list
print "".join(residueList)
```

Note - I found the shuffle function just by typing "python shuffle" into Google. I saw that it acts on lists, so I converted the sequence into a list of characters first.

13. Challenge problem (solution to first part). Read a file of tab-delimited text and print the Nth and Mth fields from each line, where N and M are command-line specified.

```
import sys
openFile = open(sys.argv[1], "r")
n = int(sys.argv[2]) - 1
m = int(sys.argv[3]) - 1
for line in openFile:
      # strip new line and split each line on tabs
      fieldList = line.strip().split("\t")
      # test that both Nth and Mth fields exist
      if len(fieldList) > n and len(fieldList) > m:
            print fieldList[n] + "\t" + fieldList[m]
openFile.close()
```