# Comp 590-087/790-087: BioAlgorithms -- Fall 2011

# Problem Set #1

## Issued: 9/4/2011 Due: In class 9/22/2010

---

**Homework Information**: Some of the problems are probably too long to attempt the night before the due date, so plan accordingly. No late homework will accepted. However, your lowest homework will be dropped. Feel free to work with others, but the work you hand in should be your own.
**Note:** 590-087 students are not required to answer the starred (*) parts of questions.

---

**Question 1.** Given a 3 billion base-pair genome, what is the expected length of a unqiue sub-sequence (one that appears nowhere else in the genome)? What are the assumptions of your answer, if any?
How many times would one expect to see a given 10-base subsequence appear in the same genome, assuming all bases are independent and equally likely?

Assuming a genome of length N where nucleotides are equally likely and their co-occurrences independent, the expected number of appearances of any k-mer would be $N/4^k$. Our objective is to make the expectation this expectation less than 1. Thus, $3.0 \times 10^9/4^k < 1.0$. Solving for k suggests a length of at least 16. Using the same equation, one would expect a given 10-mer to appear $3.0 \times 10^9/4^{10} \approx 2861$ times.

---

**Question 2.** Based on the condons shown in table 3.1 of the book and on page 22 of Lecture 1, which of the three codon positions is the least sensitve to mutation (i.e. substituting an alternative base at that position is least likely to change the amino acid)?

*Mutations can be broken into two classes, transitions, which exchange bases of a similar shape and size (A ↔ G, C ↔ T), and transversions, which exchange bases of dissimilar sizes (A ↔ C, A ↔ T, C ↔ G, G ↔ T). Eventhough there are twice as many transversions as transitions, they are far less likely. How many transitions do not result in a change of the amino acid coded (consider all three positions)?

I wrote the following 35 lines of code to answer this question:

```
Codon = {'ttt':'F', 'tct':'S', 'tat':'Y', 'tgt':'C',
         'ttc':'F', 'tcc':'S', 'tac':'Y', 'tgc':'C',
         'tta':'L', 'tca':'S', 'taa':'-', 'tga':'-',
         'ttg':'L', 'tcg':'S', 'tag':'-', 'tgg':'W',
         'ctt':'L', 'cct':'P', 'cat':'H', 'cgt':'R',
         'ctc':'L', 'ccc':'P', 'cac':'H', 'cgc':'R',
         'cta':'L', 'cca':'P', 'caa':'Q', 'cga':'R',
         'ctg':'L', 'ccg':'P', 'cag':'Q', 'cgg':'R',
         'att':'I', 'act':'T', 'aat':'N', 'agt':'S',
         'atc':'I', 'acc':'T', 'aac':'N', 'agc':'S',
         'ata':'I', 'aca':'T', 'aaa':'K', 'aga':'R',
         'atg':'M', 'acg':'T', 'aag':'K', 'agg':'R',
         'gtt':'V', 'gct':'A', 'gat':'D', 'ggt':'G',
         'gtc':'V', 'gcc':'A', 'gac':'D', 'ggc':'G',
         'gta':'V', 'gca':'A', 'gaa':'E', 'gga':'G',
         'gtg':'V', 'gcg':'A', 'gag':'E', 'ggg':'G'}

mutation = {'a': 'cgt', 'c': 'agt', 'g': 'act',  't': 'acg'}
```

```
transition = {'a': 'g', 'c': 't', 'g': 'a',  't': 'c'}

def mutationCount(pos, type):
    total = 0
    count = 0
    for code, amino in Codon.iteritems():
        amino = Codon[code]
        nucleotide = code[pos]
        for mutant in type[nucleotide]:
            total += 1
            newCode = ''.join([mutant if (i == pos) else code[i] for i in xrange(3)])
            if (amino != Codon[newCode]):
                count += 1
    return count, total

if __name__ == "__main__":
    for pos in xrange(3):
        print "%d: %s" % (pos+1, str(mutationCount(pos, mutation)))
    for pos in xrange(3):
        print "%d: %s" % (pos+1, str(mutationCount(pos, transition)))
```

There are 184, 190, and 64 mutations in positions 1, 2, and 3 respectively that result in a change of amino acid, thus making position 3 the least sensitive. Of the 3*64=192 possible transitions 66 do not result in a change of amino acid (192 – 126).

---

**Question 3.** Reconsider the incorrect "Change Problem" algorithm from Lecture 3. In how many of the 99 possible inputs for M does it given an incorrect answer for the denominations c = (25, 20, 10, 5, 1)? Suppose that you are designing coinage for a new nation, but you are limited to minting only 3 denominations. How do you choose them such that the average number of coins is minimized for all values of change from 1 to 99? Will the greedy algorithm work with your proposed coinage?

Based on the dynamic programming solution to the Change Problem covered in lecture one can extend the incorrect solutions for (40 (0,2,0,0,0), 41(0,2,0,0,1), 42 (0,2,0,0,2), 43 (0,2,0,0,3), 44 (0,2,0,0,4)) to include (65 (1,2,0,0,0), 66 (1,2,0,0,1), 67 (1,2,0,0,2), 68 (1,2,0,0,3), 69 (1,2,0,0,4)). Likewise, they can be extended to (90 (2,2,0,0,0), 91(2,2,0,0,1), 92 (2,2,0,0,2), 93 (2,2,0,0,3), 94 (2,2,0,0,4)), giving 15 total incorrect solutions.

The optimal coinage for a 3 coin-system is 19, 12, and 1, which averages 5.20 coins to make all change values between 1 and 99. The greedy change algorithm does not give the optimal change for this system (i.e. 24 (0,2,0) optimally and (1,0,5) using the greedy approach).

---

**Question 4.** Construct a 21-element partial digest that maximizes the number of calls to Place(), and thus achieves the worse case performance.

*Explain how breakpoints should be spaced on a line segment in order maximize the number of calls to Place().

The goal of this exercise is to construct a 7 break-point example that achieves the maximum number of calls to Place. One can instrument the code to count calls to Place as follows:

```
PlaceCalls = 0

def Place(L, X):
    global PlaceCalls
    PlaceCalls += 1
```

```
    if (len(L) == 0):
#        print sorted(X)
        return
    y = max(L)
    dyX = delta(y, X)
    if (dyX.subset(L)):
        X.append(y); map(L.remove, dyX.items)
        Place(L, X)
        X.remove(y); map(L.append, dyX.items)
    w = max(X) - y          # width - y
    dwX = delta(w, X)
    if (dwX.subset(L)):
        X.append(w); map(L.remove, dwX.items)
        Place(L, X)
        X.remove(w); map(L.append, dwX.items)
    return
```

Note the following: Prior to the first call to Place, the digest list, L, is reduced by 1 (removal of max). The 1st call removes 2 more elements before recursion, the 2nd removes 3, and so on until none are left. Thus, the depth of the recursion is limited to 6 calls to remove all 21 elements. Since each level of recursion can, at most, call Place twice. Thus, the upper bound, not to be exceeded number of calls to Place is $2^6$ -1 = 63.

One approach to this problem is to enumerate all possibilities on a segment of a given length. I wrote a simple code fragment to enumerate all $\binom{n}{5}$ for breakpoint choices for lengths from 6 to 64, and came up with [0, 1, 3, 6, 7, 8, 12] which made 37 calls to Place(), a little less than half. No answer turned in exceeded my number. To contrast, the most common answer of [0,1,2,3,4,5,6] makes 21 calls.

---

**Question 5.** Given a long genome sequence *G*, and a shorter pattern string *S*, sketch out an algorithm (i.e. write pseudo code) for finding the first occurence of S in G (if any). What is the complexity of your algorithm? Describe the input that gives the worst-case performance for your algorithm.
*Estimate the average performace of your algorithm.

The following pseudo code implements the algorithm described:

```
def FindFirst(S,G):
    for i in xrange(len(G)-len(S)+1):
        for j in xrange(len(S)):
            if S[j] != G[i+j]:
                break
        else:
            return i
    return -1
```

Notice how the for-else construct of Python simplifies the code. The else clause is executed when the for-loop runs to completion (i.e. without a break). The outer loop executes at most (len(*G*) – len(S)) times, while the inner loop executes at most len(S) times, giving a complexity of *O*(len(S) (len(*G*) – len(S)) = *O*(len(S)len(*G*)). The worse case occurs when the target string either does not appear, or it only appears as the suffix of *G*. The average performance is the smaller of $4^{\text{len(S)}}$ or len(*G*)/2.

There is a more efficient algorithm for finding the first occurrence of substring within a string called the Knuth-Morris-Pratt algorithm, which has a worse case and average performance of *O*(len(*G*)+len(S)).

**Programming Problem.** Modify the code for PartialDigest given in class so that it outputs only exactly one instance of each correct output. The approach of storing previous results and comparing a potential new output to them is frowned upon. I suggest instrumenting the code to figure out why results are revisited and reorganizing the code to avoid reexamination of decisions that lead to the same answer.
Turn in a listing of the source code for your algorithm, as well as a print out of its output on the input:
1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,5,5,5,6,6,7,7,8,8,8,9,9,10,10,11,11,12,12,12,13,14,15
also provide a print out of your code's output for the 21-element partial digest you came up with for in Question 4.

The trick here is to maintain the list of breakpoints in sorted order, and verify that no sequence is revisited.

```
def insort(u, L):
    for i, v in enumerate(L):
        if v >= u:
            break
    L[i:i] = [u]
    return

def Place(L, X, considered = set()):
    order = tuple(X)
    if order in considered:
        return
    considered.add(order)
    if (len(L) == 0):
        print X
        return
    y = max(L)
    dyX = delta(y, X)
    if (dyX.subset(L)):
        insort(y, X); map(L.remove, dyX.items)
        Place(L, X, considered)
        X.remove(y); map(L.append, dyX.items)
    w = max(X) - y          # width - y
    dwX = delta(w, X)
    if (dwX.subset(L)):
        insort(w, X); map(L.remove, dwX.items)
        Place(L, X, considered)
        X.remove(w); map(L.append, dwX.items)
    return
```