**University of Virginia Computer Science**
**CS216: Program and Data Representation, Spring 2006**
*http://www.cs.virginia.edu/cs216/ps/ps1/*
18 January 2006

# Problem Set 1
# Sequence Alignment

Out: 18 January
Due: Monday, 23 January (11am)

**What to Turn In**

On Monday, 23 January, bring to class a stapled turn in containing your answers to questions 1-10 (and optionally question 11). You and your partner should turn in a single assignment reflecting your best combined efforts with both of your names on it. Your answers should be printed in order and formatted clearly. Include all the code you wrote, but do not include large amounts of code we provided in your turn in.

## Collaboration Policy - Read Carefully

For this assignment, you should do the first two parts (questions 1-6) on your own, and then meet with your assigned partner. You will receive an email shortly after you submit your registration survey that will identify your partner for this assignment.

When you meet with your partner, you should first discuss your answers to the first two parts to arrive at a consensus best answer for each question. The consensus answer is the only answer you will turn in. Then, you should work as a team on the final part (questions 7-10). When you are working as a team, both partners should be actively involved all the time and you should take turns driving (who is typing at the keyboard). Both teammates can (optionally) provide their own answers to question 11.

You may consult any outside resources including books, papers, web sites and people, you wish for information on Python programming. You should not, however, conduct web searches or look at reference material on Sequence Alignment or related problems. The point of this assignment is to get you thinking on your own and with your partner about good solutions to this problem; it would defeat that purpose if you spent your time looking up well known good solutions instead of thinking for yourself.

You are **strongly encouraged** to take advantage of the staffed lab hours (which will be posted on the CS216 web site).

## Purpose

- Gain experience with order notation
- Get familiar with the Python programming language and tools
- Learn to experimentally analyze the asymptotic efficiency of a data structure
- Understand and analyze a brute-force genome alignment algorithm

**Reading:** Read Chapters 1-3 of the textbook.
**Download:** ps1.zip (this file contains the `Timer.py` and `Align.py` code)

## Orders of Growth

Section 1.3 of the textbook introduces the order notation we will use frequently in CS216 to analyze the running time (and sometime space use) of algorithms. These questions check your understanding of this notation.

These questions ask you to prove properties of the factorial function, $n!$, defined by this recurrance relation:

factorial (0) = 1
factorial ($n$) = $n$ * factorial($n$ - 1) if $n \geq 1$.

In your answers, you may not use Stirling's approximation (or any other tight bound you know for factorial), but instead should construct a proof using simple properties of the recurrance relation.

**1.** Prove that $n! \in O(n^n)$

**2.** Prove that $n! \in \Omega(2^n)$

**3.** Prove that $n^\alpha \in o(c^n)$ for any $\alpha > 0$ and $c > 1$. (This is Theorem 3 on page 22 of the textbook.)

## Analyzing Asymptotic Efficiency

For these questions, your goal is to analyze the asmptotic complexity of the Python list operations. Unlike most algorithm analysis we will do, however, in this case you will not be considering the design and implementation of the algorithms, but analyzing them experimentally by measuring how long different sequences of operations take. An experimental analysis can never establish with certainty the asymptotic complexity of any algorithm, unless it is possible to try it on all but a finite number of possible inputs. Since the number of possible inputs to the list operations is infinite, it is possible that the experimental results will be incorrect. However, by carefully designing your experiments, you should be able to get a tight bound on the time complexity with a high degree of confidence.

It is up to you to determine the experiments to do. We have provided a Python class `Timer.py` that you may find useful. It provides methods `start` and `stop` for starting and stoping a timer, and `elapsed` that returns the elapsed time between the start and stop calls (in seconds).

For example, this code measures the time it takes to perform 100000 list append operations:

```
import Timer

timer = Timer.Timer ()
alist = []
timer.start ()

for i in range (100000)
   alist.append (i)

timer.stop ()
print "Time: %2.6f" % (timer.elapsed ())
```

**4.** Determine the asymptotic running time of the list `append` operation as a function of the input list size. Express your answer using order notation, as precisely as possible. Your answer should include an explanation of the experiments you did, as well as the Python code and the results of your experiments.

**5.** Determine the asymptotic complexity of the list `insert` operation. The time of an `insert` operation may be affected by two properties of the input: the size of the list, *n* and the location index where the new value is inserted, *l*. Express your answer using order notation, as precisely as possible. Your answer should include an explanation of the experiments you did, as well as the Python code and the results of your experiments.

**6.** Briefly speculate on how Python implements lists and the list operations. Can you determine if Python lists are implemented using a continuous representation or a linked representation? To answer this well, you may need to experiment with other list operations such as access (`lst[i]`) and selection (`lst[1:]` or `lst[a:b]`).

## Sequence Alignment

The sequence alignment problem takes as input two or more sequences, and produces as output an arrangement of those sequences that highlights their similarities and differences. This is done by introducing gaps (denoted using dashes) in the sequences so the similar segments line up.

Sequence alignment is an important component of many genome analyses. For example, it is used to determine which sequences are likely to have come from a common ancestor and to construct phylogenetic trees that explain the most likely evolutionary relationships among a group of genes, species, or organisms. As genomes evolve, they mutate. Sequences can be altered by substitution (one base is replaced by another base), insertions (some new bases are inserted in the sequence), or deletions (some bases are deleted from the sequence).

To identify a good sequence alignment, we need a way of measuring how well an arrangement is aligned. This is done using a goodness metric, that takes as input the two aligned sequences and calculates a score that indicates how well aligned they are. The goodness metric we will use is:

$$score\,(u, v) = c \quad \text{if } u = v$$
$$score\,(u, v) = -g \quad \text{if } u \text{ or } v \text{ is a gap}$$

where *u* and *v* are the bases at a given location on each sequence.

The values of *c* and *g* are constants choosen to reflect the relative likelihood of a point mutation and insertion or deletion. For these examples, we will use values of *c* = 10 and *g* = 2. Note that the values selected will affect which alignment is best.

**Example.** Consider the catish sequences:

*U* = `catcatggaa`
*V* = `catagcatgg`

To compute the goodness score, we sum the scores at each position and subtract the gap penalties:

```
catcatggaa
|||
catagcatgg
```

goodness = 3 * *c* = 30

Note that this is not the best possible alignment of the sequences. We can do better by inserting gaps:

```
cat--catggaa
|||  |||||
catagcatgg--
```

goodness = 8 * *c* - 4 * *g* = 72

## Finding the Best Alignment

A brute force technique for finding the best alignment is to try all possible alignments, and keep the one with the best goodness score.

We can do this by recursively trying all three possibilities at each position:

1. No gap
2. Insert a gap for the first sequence
3. Insert a gap for the second sequence

Note that inserting a gap in both sequences makes no sense, since we could always improve the goodness score by removing that double gap.

Hence, we can find the best allignment of sequences *U* and *V* using this Python code (found in `Align.py`:

```
def bestAlignment (U, V, c, g):
    print "Best Alignment: ", U, V, c, g
    if len(U) == 0 or len(V) == 0:
        while len(U) < len(V): U = U + GAP
        while len(V) < len(U): V = V + GAP
        return U, V
    else:
        # try with no gap
        (U0, V0) = bestAlignment (U[1:], V[1:], c, g)
        scoreNoGap = goodnessScore (U0, V0, c, g)
        if U[0] == V[0]: scoreNoGap += c

        # try inserting a gap in U (no match for V[0])
        (U1, V1) = bestAlignment (U, V[1:], c, g)
        scoreGapU = goodnessScore (U1, V1, c, g) - g

        # try inserting a gap in V (no match for U[0])
        (U2, V2) = bestAlignment (U[1:], V, c, g)
        scoreGapV = goodnessScore (U2, V2, c, g) - g

        if scoreNoGap >= scoreGapU and scoreNoGap >= scoreGapV:
            return U[0] + U0, V[0] + V0
        elif scoreGapU >= scoreGapV:
            return GAP + U1, V[0] + V1
        else:
            return U[0] + U2, GAP + V2
```

The procedure `bestAlignment` in `Align.py` implements this algorithm.

**7.** Implement the missing `goodnessScore` procedure. Your procedure should calculate the goodness score of two genomes with gaps inserted.

**8.** Determine analytically the asymptotic time of the `bestAlignment` procedure. Include a convincing argument supporting your answer and be sure to clearly define all variables you use in your answer.

**9.** A typical gene is a few thousand bases long. Predict how long it would take your `bestAlignment` procedure to align two 1000-base pair human and mouse genes.

**10.** Suggest approaches for improving the performance of `bestAlignment` and predict how they would affect your answers to 8 and 9. (You do not need to implement them, although we will be especially impressed if you do.)

**11.** (optional, ungraded)
**a.** How much time did you spend on this problem set?
**b.** How much of that time seemed worthwhile?
**c.** Do you have any other comments on improving this problem set or the structure for future problem sets in CS216?