CS107 Handout 36
Spring 2008 May 28, 2008

Assignment 8: Tinkering With Python

There are several alignment methods for measuring the similarity of two DNA sequences (which for the purposes of this problem can be thought of as strings over a four-letter alphabet: **A**, **C**, **G**, and **T**). One such method to align two sequences **x** and **y** consists of inserting spaces at arbitrary locations (including at either end) so that the resulting sequences **x'** and **y'** have the same length but do not have a space in the same position. Then you can assign a score to each position. Position **j** is scored as follows:

- +1 if **x'[j]** and **y'[j]** are the same and neither is a space,
- –1 if **x'[j]** and **y'[j]** are different and neither is a space,
- –2 if either **x'[j]** or **y'[j]** is a space.

The score for a particular alignment is just the sum of the scores over all positions. For example, given the sequences **GATCGGCAT** and **CAATGTGAATC**, one such alignment (though not necessarily the best one) is:

```
positive scores for matches

+   11 1 1 11
  G ATCG GCAT
  CAAT GTGAATC
- 12  2 2 1  2

 negative scores for misses
```

The positive scores are listed above the alignment, and negative scores are listed below. This particular alignment has a total score of –4.

Due: Thursday, June 5th at 11:59 p.m.

Within the **assn-8-align** folder you'll find a Python program that generates two short but otherwise random DNA strings and posts the score of an optimal alignment (and I say **an** optimal alignment as opposed to **the** optimal alignment, because two alignments can have the same score.)

Here's a transcript of what happens when you invoke the **align.py** script:

```
jerry% ./align.py
Generate random DNA strands? yes
Aligning these two strands: GAACATCTGC
                            CCTCCACAGC
Optimal alignment score is -2
Generate random DNA strands? yes
Aligning these two strands: GTGGCGAG
                            ACCGTGGA
```

```
Optimal alignment score is -4
Generate random DNA strands? yes
Aligning these two strands: ATCCTCACAG
                            GCGCGATG
Optimal alignment score is -6
Generate random DNA strands? yes
Aligning these two strands: TGTAAGAGAT
                            CTAATCGA
Optimal alignment score is -2
Generate random DNA strands? no
jerry%
```

Your job is to update the implementation to generate not only the optimal score, but to generate the alignment itself. Here's a sample run of the script after all the necessary changes have been made:

```
jerry% python solution.pyc
Generate random DNA strands? yes
Aligning these two strands:

   TGTACCCGCCGATCCCCGACTAAAAACTCTGGGTATTGGGGTGTACTTCCACCAA
   CGACTCACAACATTCGGATAGAGAAAGCCGTTAGACAGGGCTTAGTGAGACATT

Optimal alignment score is -10

  +  1 11 1 1   11 1  11 11 1 111  1 1  1 1  111 1 11 1   11
    TGTACCCGCCG ATCCCCGACTA A AAACTC TGGGTATTGGGGTGTACTTCCACCAA
    CG ACTCACAACATTCG GA TAGAGAAAGCCGTTAG ACAGGGCT TAGTGAGACATT
  - 1 2  1 1 112  1 12  2  2 2   11 2 11 2 11   1 2  1 111  111

Generate random DNA strands? yes
Aligning these two strands:

   TGTCGAGAATATCTTTCCTGTGGCTCAGACGCAGCGGTCCCCGTAGTCAA
   AGCCGGGGTAATGCGCACGGGAGCCTGCATTTACAATAGCCAGGTGCCCATTTTCAAC

Optimal alignment score is -6

  +  1 11 1   11 1   1  1 1 11 11   11   111  111 111 1  1111
    TGTCGAGAATAT CTTTCCTGTGGCT CAG  ACGC AGC  GGTCCCCGTAGTCAA
    AGCCGGGGTAATGCGCACGGGAGCCTGCATTTACAATAGCCAGGTGCCCATTTTCAAC
  - 1 1  1 111  2 111 11 1 1  2  122  112   22   1   1 11    2

Generate random DNA strands? yes
Aligning these two strands:

   TCCATATATTGGACGATATACCAGATTCGCCACCTTGTCAGTGGTTGCCCGTAGGG
   TTTCCTGCCCAAGAGTTTCCGACTTGTCCAGGGGTTGCCCGCCGTTGCGGCGCGGG

Optimal alignment score is -7

  + 1  1 1      11 1 1 1 11 11 1 1 111    111 1 1  11111  11  111
    TC CATATATTGGACGATATACCAGATTCG CCACC  TTGTCAGTGGTTGCC CGTAGGG
    TTTCCTGCCCAAGA GTT T CC GACTTGTCCAGGGGTTGCCCGCCGTTGCGGCGC GGG
  -  12 1 111111  2 1 2 2  2  1 1 2   1122   1 1 11     12  12

Generate random DNA strands? no
jerry%
```

My recommendation is that you divide the entire task into two subtasks:

- Update the code you're given to synthesize the alignment alongside the score, resting assured that the DNA strings are no longer than 12 characters in length. By limiting the lengths of the two strings, you can assume that the recursion will complete in a matter of seconds as opposed to a matter of years. My solution packages the two strands of the alignment and the alignment score into a three-key dictionary, where the keys were **`"strand1"`**, **`"strand2"`**, and **`"score"`**.
- Once you're properly generating alignments for short DNA strands, you'll want optimize your implementation to run much more quickly. That way, you can align longer DNA strands and expect the program to finish. I expect you to be able to run the alignment of two strands up to length 60 and have it return within 10 seconds.

Memoization

The problem with the program I give you (besides the fact that it only generates the alignment score but not the alignment itself) is that it's not **scalable**—that is, it works acceptably well enough when aligning two strings of length 12, but force it to deal with DNA strands of length 40, 500 or (like real ones) 3,000,000,000 and you'll sit in front of the computer for hours, years, or all of time, respectively.

The problem lies not in the recursive formulation itself, but rather the implementation. Each call to the supplied **`findOptimalAlignment`** function makes either one or three recursive calls, depending on whether or not the leading letters in each strand match. For two arbitrary DNA strands of the same length, you expect the ternary recursive scenario to present itself around 75% of the time, which makes for a slow algorithm for anything but laughably short DNA strands.

One technique often used to speed up intensely recursive algorithms is caching—or more elaborating known in computer theory circles as **memoization**. Memoization (yes, it's spelled correctly—there's no r) isn't always useful, but it most certainly is in situations where an overwhelming majority of the recursive calls are repeats—ones that have been computed earlier during computation.

For instance, consider an attempt to align the two woefully incompatible strands **`"AATT"`** and **`"GGCC"`**. A call to **`findOptimalAlignment("AATT","GGCC")`** will call **`findOptimalAlignment`** on three separate suffix pairs:

```
findOptimalAlignment("AATT","GGCC") calls
   findOptimalAlignment("ATT","GCC"),
   findOptimalAlignment("AATT","GCC"),
   findOptimalAlignment(“ATT","GGCC")
```

Each of those three recursive calls will make three more calls:

repeated
recursive calls.

```
findOptimalAlignment("ATT","GCC") calls
   findOptimalAlignment("TT","CC"),
   findOptimalAlignment("ATT","CC"),
   findOptimalAlignment("TT","GCC")

findOptimalAlignment("AATT","GCC") calls
   findOptimalAlignment("ATT","CC"),
   findOptimalAlignment("AATT","CC"),
   findOptimalAlignment("ATT","GCC")

findOptimalAlignment(“ATT","GGCC") calls
   findOptimalAlignment("TT","GCC"),
   findOptimalAlignment("ATT","GCC"),
   findOptimalAlignment("TT","GGCC")
```

Notice that we're already seeing some repeated calls. It's not functionally incorrect to make the same call a second and third time, but if you know the answer will be the same, then you'd be smart to cache (or **memoize**) the result of the first call, consult the cache the second and third times and return the previously computed result instead of re-computing it from scratch. This reduces an exponential algorithm to a polynomial one. Huge. Huge. Savings.

Update the functionally correct but lethargic implementation and introduce memoization to speed it up. You can declare a single Python dictionary—initially empty—in the **`main`** function and pass it through the recursion as an extra argument. This is the single, shared cache where you look to see if you've solved the same problem before.