BIOL591: Introduction to Bioinformatics (2003)

Companion to:

Program to superimpose one sequence on a known structure

I. Summary of recent events

In our last episode, we decided that the way to make UDP-glucose dehydrogenase (UDPGD) from *Mesorhizobium loti* soluble when overexpressed in *E. coli* was directed mutagenesis. And the way to figure out where to put those mutations was to note the location on the three dimensional structure of the protein of mutations that had already proved partially successful in increasing solubility. We already have in hand a program, ThreadProtein.pl, that superimposes the amino acid sequence of the enzyme from *M. loti* onto the known three-dimensional structure of the enzyme from *Streptococcus pyogenes.* Our task now is to modify that program so that is also superimposes the positions of the effective mutations.

We had divided the modification of the program into three tasks:

1. Read in the list of mutations
   We pretty much achieved this in class last Wednesday.
2. Incorporate the sites of mutation into the summary string ($comparison_summary) that is used to hold the positions of matches, replacements, insertions, and deletions with regard to the two UDPGD sequences.
3. Use the summary string to build a new PDB file that puts in a separate chain amino acids that were mutated.

Unfortunately, we didn't come very close today (Monday) to finishing the task and (most importantly) seeing the results of the program through Protein Explorer. I'd like to push to completion so that we can spend more time on Wednesday on matters related to Markov models. To that end, I'll start on task #2 and work together with you.

II. Incorporate mutation sites into the summary string

The mutation positions we read in are given in terms of the mutated *M. loti* protein. This is reasonable of course. The mutations were determined by sequencing the mutated gene, and so the positions were reported with respect to that gene. Unfortunately, we need to know the coordinates of the mutations in terms of the ***aligned*** UDPGD sequences. The differences between these two coordinate systems are illustrated in Fig. 1. In this example, the 5th amino acid of the second sequence corresponds to the 8th position of the alignment. The difference in coordinate systems invites confusion.

```
     ....:....|...
Sp: MRTVSSAGW-CPL
Ml: MLS---AGWAC-L
aa: 123---45678-9
```

Fig 1: Alignment of beginnings of protein sequences. Mythical sequences from two organisms are shown, with a ruler on top to show alignment coordinate and numbering below to show the position of each *Ml* amino acid.

The desired subroutine, one that will insert the positions of the mutations into $comparison_summary, will be much simpler to write if we didn't have to worry about different coordinate systems. For that reason, I suggested that you first write a subroutine that translates amino acid position to coordinate in the alignment. Once that subroutine was tested, you wouldn't have to worry about the problem again.

II.A. Subroutine to translate amino acid position to alignment coordinate

I suggest that whenever you write a subroutine from scratch, you go through the following steps:

1. Write (as comments) a title and some brief description lines. You can supplement these later
2. Write the first and last line of the subroutine (i.e. sub xxx { and }) in one breath, to avoid the possibility of introducing mismatched brackets
3. Write the line assigning arguments to named variables
4. Write the line returning the result of the subroutine (if any)

If you do this, you should end up with something like this:

```
##### TRANSLATE_ML_COORDINATES (aa position)
#     Translates coordinate from Mesorhizobium loti UDPGDH
#        into coordinate system of $comparison_summary
#     Takes as argument position of amino acid in M. loti
#        sequence
#     Returns coordinate of same amino acid in alignment
#        coordinate system
  sub Translate_ml_coordinates {
    my ($aa_position) = @_;
    [substance of subroutine goes here]
    return $ coordinate;
  }
```

Now we need a plan. Usually that means quiet meditation, going deep into yourself to discover how you as a human being accomplish a task. Then you translate that insight into a computer language. So, how would you translate amino acid position into a coordinate of the alignment? Refer to Fig. 1. If you were given an amino acid position and could see ***only*** the line labeled *Ml* then what procedure could you use to guarantee that you could produce the appropriate coordinate? Suppose you wanted to find the coordinate of the 5th amino acid. You would find the 5th amino acid by counting from the beginning of the sequence, but taking care to count only amino acids, not gaps. When you got to the 5th amino acid (which happens to be G), you could ask what is the coordinate of that amino acid in the alignment (answer: 8). Or, if you were thinking ahead, you could be counting the coordinate position at the same time you were counting amino acids.

This sounds like a loop, something like:

1. Consider each coordinate one-by-one, from the first to the (potentially) last
2. If the character is not a gap, count it as an amino acid
3. If the amino acid count has reached the position you're after, leave the loop
4. Otherwise go back to step 1

Comments:

1. Consider each coordinate one-by-one, from the first to the (potentially) last
   - *The loop might be defined by something like foreach $ (0 .. last coordinate) { }*

2. If the character is not a gap, count it as an amino acid
   - *Where do you get those characters? Refer back to the variable list at the beginning of the program. Which variable contains the aligned M. loti UDPGDH sequence (including gaps)? If you're not sure (and who is ever sure about documentation?), then print out the candidate variables to see what they look like.*
   - *How do you determine if a character is a gap? First you have to access that character. You know the variable that has the characters. You know the coordinate of the character in question. You know you only want one character. This sounds like the perfect time for the function substr(string, coordinate, length).*
   - *How do you word the if statement? Here are some examples:*

     ```
     if (string1 eq string2) {   If the two strings are equal then…
     if (string1 ne string2) {   If the two strings are not equal then…
     ```

   - *How do you keep track of the number of amino acids counted? You'll need to define a variable and initialize it to zero. Of course, you'll do this outside the loop (otherwise it will continuously be set to zero – no counting).*
3. If the amino acid count has reached the position you're after, leave the loop
   - *How do you leave a loop? If you chose to use a while loop, then you can set as a condition that the amino acid count is less than the position you're after. If you're using a foreach loop instead, then there's a Perl statement you want to know about. Consider the following loop:*

     ```
     foreach $i (1 .. 10) {
       print $i, " ";
       if ($i > 5) {last}
       };
     ```

     (`last` causes the immediate departure from the loop)

You still have a few details to take care of (declaring variables, initializing them), but the loop should do most of the work of the subroutine. If you are not able to get a working subroutine, please contact me and we'll find a solution.

II.A. Subroutine to translate amino acid position to alignment coordinate

What will this subroutine do? What will it need? What will it produce? There are a few right answers to these questions, but I've decided that it will take as an argument the summary string ($comparison_summary) and the list of mutations (whatever you chose to call it) and it will return a new summary string, with mutations added. Now I can compose the frame of the subroutine:

```
##### ADD_MUTATIONS_TO_SUMMARY_STRING (summary string, mutation list)
#      Puts symbol for mutation at appropriate positions of
#         summary string
#      Takes as arguments:
#         pre-made summary string (made by Analyze_alignment)
#         mutation list (made by XXX) (whatever you called it)
#      Returns revised summary string, with mutations added
  sub Add_mutations_to_summary_string {
    my ($comparison, @mutation_list) = @_;
    [substance of subroutine goes here]
    return $comparison;
}
```

The summary string the subroutine is given will be a string exactly as long as enzyme sequence, with gaps, and it uses the alignment coordinates. The list of mutations will be a list of amino acid positions. The most straightforward thing to do is to go through the list of mutations one by one and insert a symbol into the summary string at the appropriate position.

That sounds like a loop, one that takes the elements of the list one at a time. A foreach loop does that.

The loop will give you an amino acid position. How can you use it to change a specific coordinate of the summary string? First of all, you need to translate amino acid position to alignment coordinate. You know how to do that.

Once you've done that, you need to access the string at that specific position and put there a symbol representing mutation. substr was made for that:

substr($comparison, coordinate derived from aa position, 1) = "symbol of some sort";

Once you've completed the subroutine, don't forget that you have to call it in the Main Program (with the appropriate arguments).

III. Build a new PDB file with mutation information in a new chain

It might sound complicated to build a PDB file, unless you have past experience with this kind of file, which is not likely. Fortunately, you don't have to know much about the task except that ThreadProtein.pl is able to do it. Of course, you'd also like to know which ***part*** of ThreadProtein does it, but that's not so difficult, in fact the subroutine Create_new_structure_file stands out as a candidate.

There's a lot one ***could*** try to understand in that subroutine, most becomes unnecessary as soon as you realize that what you're trying to do (put all mutations in a separate chain) is almost exactly the same thing the subroutine already does with regards to replacements. How does the subroutine make sure that the coordinates are correct? How does it deal with extra amino acids? Who cares? Just tie your fate to some code that's proven to work and does what is analogous to what you want to do, and you should be all right.

Here's the code:

```
} elsif ($residue_status eq "R") {      # Replacements go in chain B
   $new_residue = substr($threaded_seq, $residue, 1);
   $line = Update_residue_name($line, $new_residue);
   $line = Update_chain_ID($line, "B");
   push @{$atom_lines{"B"}}, $line;
   $previous_line = $line;     # Save $line for insertions after end of seq
   $line = <OLD_STRUCTURE>;             # Read next line
```

Of course you're going to have to use a different symbol, not "R", to pick out the mutations within the summary string. And of course you will need to use a different chain, not "B", to hold the mutated amino acids. But otherwise, you can pretty much copy this code intact and add on to the list of elsifs that tell the subroutine what to do.

That's it! That's really all that's necessary to change ThreadProtein.pl into a mutation-processing machine

IV. Analysis of structure

Now comes the fun part. Bring the new PDB file into ProteinExplorer and gaze at the structure for a while. Identify what color is what type of amino acid. Change the colors if you want your favorite amino acids to stick out more. (Once your in Quick Views, you can select specific chains and change their colors). Turn the structure around and try to find its best side. What kind of generalities can you come up with?