1 Python

Below is a very short introduction to some Python syntax, specifically some things you may need for the homework assignment. Much more information is available at \texttt{http://www.python.org}.

Python features include:

1. Object oriented programming
2. Exception handling
3. Automatic garbage collecting
4. A large standard library
5. An interpreter
6. Runs on several operating systems (MacOS, Windows, Unix even OS/2)
7. And much more...

Some things to keep in mind when using Python:

1. You can test code out in the interpreter
2. White space matters! Indentation is used to delineate blocks
3. You do not need to have semicolons but you cannot break up lines unless you use a backslash:

\begin{verbatim}
  a = 4 + 5 + 2 + \\
  4 + 5
\end{verbatim}

4. \texttt{not}, \texttt{and} and \texttt{or} are used in place of \texttt{!}, \texttt{&&}, \texttt{||}, respectively

5. Control structures must have a colon:

\begin{verbatim}
  if not 5 > x:
    x = 5
  while x > 5:
    x = x - 1
  # prints numbers from 0 to 9
  for x in range(10):
    print x
\end{verbatim}

6. A zero list (array) of size \texttt{x} can be made with the following command: \texttt{[0]*x}

7. Strings, which are basically lists that cannot be changed, can be indexed like in C (but with ranges) or concatenated like in perl:

\begin{verbatim}
  x = "a"*3     # z = "aaa"
  x = x + "bcd" # z = "aaabcd"
  x = x[3:5]    # z = "bc"
\end{verbatim}

8. Functions have the following syntax:
1. `def foo(a, b):
   return a + b + 5`

9. Tuples are list lists but they cannot be changed once they have been created but they can contain different datatypes.

   `t = ((1,4,2),324)`

10. Dictionaries allow for a mapping from an immutable (unchangeable) value to some value.

   `a = {'a': 4, 'b': "three", 3:4, (3,7): "three"}
   print a[(3,7)] # "three"

2 Dynamic Programming

Dynamic programming is basically recursion without repetition. To solve a problem using dynamic programming you must do the following:

1. Formulate the solution recursively
2. Solve the recursion in a bottom up manner

2.1 Global Alignment

With global alignment dynamic programming is implemented by realizing that there are three possibilities for the last column of the optimal alignment of `< a₁, a₂, . . . , aₙ >` with `< b₁, b₂, . . . , bₘ >`:

1. `aₙ` is aligned with `bₘ`. In this case, we have the cost of aligning `< a₁, a₂, . . . , aₙ₋₁ >` with `< b₁, b₂, . . . , bₘ₋₁ >` plus the cost of matching `aₙ` with `bₘ`:
   \[ A(< a₁, a₂, . . . , aₙ₋₁ >, < b₁, b₂, . . . , bₘ₋₁ >) + s(aₙ, bₘ) \]

2. `aₙ` is aligned with a gap. Now we must align `< a₁, a₂, . . . , aₙ₋₁ >` with `< b₁, b₂, . . . , bₘ >` and pay for a gap:
   \[ A(< a₁, a₂, . . . , aₙ₋₁ >, < b₁, b₂, . . . , bₘ >) − d \]

3. `bₘ` is aligned with a gap. This is symmetric to the previous case:
   \[ A(< a₁, a₂, . . . , aₙ >, < b₁, b₂, . . . , bₘ₋₁ >) − d \]

Because we are looking for the *optimal* alignment (or the highest scoring one), we just take the maximum over the three possibilities. If we look at the recursion tree, we see that the same subproblem is solved several times. We take advantage of this by solving all \(nm\) subproblems from the bottom up and storing the data in a table.

2.1.1 Number of alignments

To figure out the total number of possible alignments, we can just look at the number of paths through the table because there is a one-to-one correspondence between an alignment and a path through the table. Let \(d\) be the number of diagonals we take in the table. Because we must go up and to the left exactly \(m + n\) times, and a diagonal goes both up and left, we are left with:

\[
N(m, n) = \sum_{d=0}^{\min(m,n)} \binom{m+n-d}{d} \binom{m+n-2d}{m-d}
\]

because we have to pick when to take the diagonals and then when to go up in the matrix.
2.2 Optimal Triangulation of a Convex Polygon

Dynamic programming is also used to solve many other problems. To get practice with it we will look at one from computational geometry.

Here, we will define the optimal triangulation of a convex polygon (a polygon where all the corners point outward) to be the breaking up of the polygon into triangles along its diagonals such that the sum of all the perimeters of the triangles is minimized. Optimal triangulations have several applications in fields such as computer graphics.

For convenience, we number the corners of the polygon from 1 to n such that i and i + 1 share an edge and n and 1 share an edge. Let \( P(i, j, k) \) be the perimeter of the triangles where \( i, j, k \) are corners of the polygon. The primary insight for this problem is that each edge of the polygon is part of exactly one triangle and this triangle breaks the larger polygon into subregions. Thus, it follows that the score of the optimal triangulation is given by:

\[
S(i, j) = \begin{cases} 
0 & \text{if } j = i + 1 \\
\min_{j < k < j}(P(i, j, k) + S(i, k) + S(k, j)) & \text{otherwise}
\end{cases}
\]

We can implement this recursion using dynamic programming. Because our recursive calls are always made on smaller ranges, we first compute the smaller ranges and then move up (hence the bottom up approach of dynamic programming). We can backtrack this matrix to find the actual optimal triangulation (rather than just the score).

Once possible ordering for how to fill in the table for \( n = 6 \) is given below:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
<td>10</td>
<td>13</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>7</td>
<td>11</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>8</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notice, while we only need to fill a \( O(n^2) \) matrix, at each step we must perform a \( \max \) which can take \( n \) time. Thus, our running time is \( O(n^3) \).

2.3 Optimal Matrix Multiplication

The standard way to multiply \( x \times y \) and \( y \times z \) matrices takes \( O(xyz) \) arithmetical operations. Thus, a natural problem is the best order to to perform (i.e. parenthesize) \( X_1 \times X_2 \times \cdots \times X_n \) where \( X_i \) is a \( h_{i-1} \times h_i \) matrix.

We can readily see that there must be some pair of outermost parathesis in the optimal multiplication order (this corresponds to the last two multiplications). We search for this, leading to the recursion:

\[
S(i, j) = \begin{cases} 
0 & \text{if } i = j \\
\min_{i < k < j}(h_{i-1}h_kh_j + S(i, k) + S(k + 1, j)) & \text{otherwise}
\end{cases}
\]

This matrix is filled in similarly as to the previous problem and as the same running and space requirements.

3 Substitution Matrices

Typically, substitution matrices are developed by manually aligning very similar sequences and then extrapolating to obtain the probability of certain amino acids being replaced by other ones or being maintained.

If we had a prior to indicate what percentage of pairs of sequences are related, it may have been more appropriate to use a Bayesian approach. Because we do not, we typically use a log odds ratio. We consider two models: the unrelated and related. For the unrelated model, we assume that the nucleotides in the two sequences are independent of each other. Thus, \( P(a, b|U) = P(a|U)P(b|U) \). For the related model we have \( P(a, b|R) \). Both these values are derived from frequency counts. If for a pair of sequences \( A \) and \( B \), \( P(A, B|R) \gg P(A, B|U) \), then we have a high confidence that \( A \) and \( B \) are related.
The log ratio of the single nucleotide probabilities gives us

\[ s(a, b) = \log \frac{P(a, b|R)}{P(a|U)P(b|U)}. \]

Because we assume that the substitutions are independent, the score of an alignment of \( A \) and \( B \) leads to:

\[
S(A, B) = \sum s(A_i, B_i) = \sum \log \frac{P(A_i, B_i|R)}{P(A_i|U)P(B_i|U)}
\]

\[
= \log \prod \frac{P(A_i, B_i|R)}{P(A_i|U)P(B_i|U)}
\]

\[
= \log \frac{P(A, B|R)}{P(A|U)P(B|U)}
\]

which should be large when two sequences are related and small when they are unrelated.

Note: this does not account for the score of gaps. The gap penalty is often derived in a heuristic manner.

The value along the diagonal of a substitution matrix measures to what extent the alignment of an amino acid with itself increases our confidence that two sequences are related. Thus, we expect variance amongst these values because certain amino acids are very common and finding them aligned to does not significantly increase how sure we are that two sequences are related.

4 Miscellaneous

1. Questions on homework?

2. Good times for office hours for me and professors?