Dynamic Programming for Bioinformatics...

Longest Common Subsequence

Subsequence. Given a string $S = s_1 s_2 \ldots s_n$, a *subsequence* of S is any string $P = p_1 \ldots p_k$, such that:

1. For all $1 \leq i \leq k$, $p_i = s_j$ for some $j > 0$;
2. If p_i is s_j and p_{i+1} is s_l, then $l > j$.

Informally, a subsequence P of string S can be obtained by removing zero or more characters from S and preserving the order of the characters not removed.

Example. Let $S = ATCATTCGC$. Then, ATC, AAT, $ATATG$ and $CCCG$ are all subsequences of S, while AAA, $ATTA$ and CCT are not.

Longest Common Subsequence (LCS) Problem. The Longest Common Subsequence (LCS) problem is specified as follows: given two strings S and T, find the longest string P which is a substring for both S and T.

Brute-Force Solution.

A **naïve algorithm** for solving LCS is:

1. Enumerate all possible subsequences of S.
2. For each subsequence of S check if it is also a subsequence of T.
3. Keep track of the longest common subsequence found and its length.

Analysis. A string $S = s_1 \ldots s_n$ has 2^n possible subsequences (each subsequence is essentially a choice of which characters are **in** and which characters are **out**). Some of these subsequences are not unique, but in a brute-force algorithm, there is no way to know that ahead of time. Checking if a string $T = t_1 \ldots t_m$ contains a subsequence $P = p_1 \ldots p_k$ can be done in $O(m + k) = O(m)$ (if $k > m$, the answer is an automatic "no") time. Thus, the overall complexity of the brute-force algorithm is $O(m2^n)$.

Characterization of a Longest Common Subsequence

To help us develop an efficient algorithm for LCS, we need to be able to understand what a longest common subsequence of two sequences looks like. The following theorem provides the key idea for an efficient algorithm:

Theorem. Let $S = s_1 \dots s_n$ and $T = t_1 \dots t_m$ be two strings and let $P = p_1 \dots p_k$ be their longest common subsequence. Then:

1. **if** $s_n = t_m$, **then** $p_1 \dots p_{k-1}$ is the longest common subsequence of $s_1 \dots s_{n-1}$ and $t_1 \dots t_{m-1}$;
2. **if** $s_n \neq t_m$ **and** $p_k \neq s_n$, **then** P is the longest common subsequence of $s_1 \dots s_{n-1}$ and T.
3. **if** $s_n \neq t_m$ **and** $p_k \neq t_m$, **then** P is the longest common subsequence of S and $t_1 \dots t_{m-1}$.

Given $S = s_1 \dots s_n$ and $T = t_1 \dots t_m$, let $c[i,j]$ (for $1 \leq i \leq n$ and $1 \leq j \leq m$) represent the length of the maximal longest subsequence of $s_1 \dots s_i$ and $t_1 \dots t_j$. For the sake of consistency we set $c[0,0] = 0$.

The theorem suggests the following approach to determining the length of the LCS of S and T:

- Build the matrix $c[i,j]$ from $c[0,0]$ all the way to $c[n,m]$. $c[n,m]$ will contain the length of the LCS of S and T.
- Make sure that the construction of the matrix allows for a fast determination of the actual LCS.

Building the matrix $c[i,j]$. Using the theorem above, we can derive the following about $c[i,j]$:

- if $s_i = t_j$ then $c[i,j] = c[i-1,j-1]+1$.

  If the two last characters of the substrings agree, then the LCS extends to include this character.

- if $s_i \neq t_j$ then $c[i,j] = \max(c[i,j-1], c[i-1,j])$.

  Essentially, if the last characters of the substring differ, then the LCS of $s_1 \dots s_i$ and $t_1 \dots t_j$ is also the LCS of one of the two strings and the other string without the last character.

We represent this formally as the following recurrence relation:

$$c[i,j] = \begin{cases} 0 & \text{if } i = j = 0; \\ c[i-1,j-1]+1 & \text{if } i,j > 0; s_i = t_j; \\ \max(c[i,j-1], c[j,i-1]) & \text{if } i,j > 0; s_i \neq t_j \end{cases}$$

Essentially, $c[i,j]$ can be determined if we know the values in the following cells: $c[i-1,j-1]$, $c[i,j-1]$ and $c[i-1,j]$. We can set $c[0,j] = 0$ and $c[i,0] = 0$ for all $1 \leq j \leq m$ and $1 \leq i \leq n$. This makes it possible to compute $c[1,1]$, which, in turn, makes it possible to compute $c[1,2]$ and $c[2,1]$, and so on.

"Remembering" the LCS. On each step (i,j) of computation of $c[i,j]$, we can determine which of the three cells $c[i-1,j-1]$ (diagonally above and to the left), $c[i,j-1]$ (to the left) or $c[i-1,j]$ (above) is the one whose value is used in computing $c[i,j]$.

We create a table $u[i,j]$. In cell $s[i,j]$ we store the "pointer" to the cell from which $c[i,j]$ was constructed. We use symbols $\leftarrow$, $\uparrow$ and $\nwarrow$ to denote the following cases:

$u[i,j]$ Symbol	s_i vs. t_j	Table condition
$\nwarrow$	$s_i = t_j$	N/A
$\uparrow$	$s_i \neq t_j$	$c[i-1,j] \geq c[i,j-1]$
$\leftarrow$	$s_i \neq t_j$	$c[i,j-1] > c[i-1,j]$

Proposition. There is a path from $s[n,m]$ to $s[0,0]$. The LCS of $S = s_1 \ldots s_n$ and $T = t_1 \ldots t_m$, given a constructed matrix $u[i,j]$ can be found by combining all s_i characters for all locations $[i,j]$, where $u[i,j] = \nwarrow$.

Dynamic Programming Algorithm for LCS

To find the LCS of two strings, we need to construct the two matrices: $c[i,j]$ and $s[i,j]$. The following **iterative** version of the algorithm can do it.

```
Algorithm LCS(S = s_1 ... s_n, T = t_1 ... t_m)
begin
  declare c[0..n, 0..m];
  declare u[0..n, 0..m];
  for i = 0 to n do
    c[i, 0] := 0;
  end for
  for j = 1 to m do
    c[0, j] := 0;
  end for
  for i = 1 to n do
    for j = 1 to m do
      if  s_i = t_j  then
        c[i, j] := c[i − 1, j − 1] + 1;
        u[i, j] :=↖;
      else
        if  c[i − 1, j] ≥ c[i, j − 1]  then
          c[i, j] := c[i − 1, j];
          u[i, j] :=↑;
        else
          c[i, j] := c[i, j − 1];
          u[i, j] :=←;
        end if
      end if
    end for
  end for
  LCSLength:= c[n, m];
  LCS:= LCSRecover(S, T, u[]);
  return (LCS, LCSLength);
end
```

The algorithm LCSRecover takes as input two strings, S and T[1] and the matrix $u[i,j]$ that encodes how $c[i,j]$ was filled, and returns back the LCS of S and T. The algorithm works as follows (in the algorithm below, + on string values is a concatenation operation).

```
Algorithm LCSREcover(S = s_1 ... s_n, T = t_1 ... t_n, u[0..n, 0..m])
begin
  P := "";
  i := n;
  j := m;
  P := s_i + P;
  while i > 0 and j > 0 do
   if u[i,j] = ↖ then
    P := s_i + P;
    i := i − 1;
    j := j − 1;
   else
    if u[i,j] = ← then
     j := j − 1;
    else          // u[i,j] = ↑
     i := i − 1;
    end if
   end if
  end while
  return P;
end
```

Analysis. Algorithm LCS contains a double nested loop that iterates $n \cdot m$ times. Each loop iteration completes in $O(1)$.

On each step of the main loop of the algorithm LCSRecover either i or j gets decreased (and on some steps, both i and j are decreased). This means that the main loop of LCSRecover runs no more than $m+n$ times, and the algorithm itself has $O(m+n)$ runtime complexity.

As a result, algorithm LCS has $O(nm) + O(n+m) = O(nm)$ runtime complexity.

Edit Distance

Edit Distance. Given two strings $S = s_1 \ldots s_n$ and $T = t_1 \ldots t_m$, the **edit distance** between S and T is defined as the *smallest number of atomic edit operations* necessary to transform S into T. The *atomic edit operations* are

- Character insertion. An insertion of a single character from the alphabet into any position in the string.
- Character deletion. A removal of any character from the string.
- Character replacement. A replacement of any character in the string with another character from the alphabet.

[1]It actually needs only one string, since it returns the common sequence.

Example. Given a word `"cat"`, the following words have an edit distance of 1 from it:

- `"at"`, obtained from `"cat"` by deleting its first character:

```
cat
X||
_at
```

- `"cast"`, obtained from `"cat"` by inserting a character `"s"` into the third position of the string:

```
ca_t
||X|
cast
```

- `"vat"`, obtained from `"cat"` by replacing the first character with `"v"`:

```
cat
X||
vat
```

Computing the Edit Distance. We want to develop a dynamic programming algorithm for computing the edit distance. In preparation for this, we will consider using a data structure similar to the one we used when solving the LCS problem.

Let $c[i, j]$ be the edit distance between the prefixes $S_i = s_1 \dots s_i$ and $T_j = t_1 \dots t_j$ of the strings S and T. Our algorithm will construct the table $c[i, j]$. When completed, $c[n, m]$ will contain the edit distance between S and T.

The construction of $c[i, j]$ is guided by the following observations:

- $c[0, 0] = 0$. For the sake of consistency, S_0 and T_0 are empty strings. The edit distance between two empty strings is 0.
- $c[0, j] = j$ for all $1 \leq j \leq m$. The edit distance between an empty string and any non-empty string of length j is j: the string can be constructed via j consecutive insertions.
- $c[i, 0] = i$: see above (the empty string is constructed from $s_1 \dots s_i$ via i consecutive deletions).
- If $s_i = t_j$, then $c[i, j] = c[i-1, j-1]$. If the last characters of the two prefixes coincide, then the edit distance between them is the same as the edit distance between the prefixes without the last characters.
- If $s_i \neq t_j$, then an atomic edit is needed to match the last characters of the strings S_i and T_j. We must select one of the three possible atomic edits (insertion, deletion, or replacement). When selecting which one to use, we basically are reducing computing the edit distance between S_i and T_j to:

1. computing the edit distance between S_{i-1} and T_{j-1} if replacement is chosen.
2. computing the edit distance between S_{i-1} and T_j if deletion is chosen.
3. computing the edit distance between S_i and T_{j-1} if insertion is chosen.

These insights can be properly encoded as follows:

$$c[i,j] = \begin{cases} 0 & \text{if } i = j = 0 \\ i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ c[i-1, j-1] & \text{if } i, j \geq 1 \text{ and } s_i = t_j \\ \min(c[i-1,j-1], c[i-1,j], c[i,j-1]) + 1 & \text{if } i, j \geq 1 \text{ and } s_i \neq t_j \end{cases}$$

Algorithm for Edit Distance Computation

Using the formula derived above, we can write the following algorithm for computing the table $c[i,j]$. The algorithm returns $c[n,m]$, which contains the edit distance between the input strings S and T.

```
Algorithm EditDistance(S = s_1 ... s_n, T = t_1 ... t_m)
begin
  declare c[0..n, 0..m];
  for i = 0 to n do
    c[i,0] := 0;
  end for
  for j = 1 to m do
    c[0,j] := 0;
  end for
  for i = 1 to n do
    for j = 1 to m do
      if  s_i = t_j  then
        c[i,j] := c[i-1, j-1];
      else
        c[i,j] := min(c[i-1,j], c[i,j-1], c[i-1,j-1]) + 1;
      end if
    end for
  end for
  return c[n,m];
end
```

Analysis. The double nested loop executes $n \cdot m$ times. Each iteration runs in $O(1)$. Therefore, the algorithmic complexity of the EditDistance algorithm is $O(nm)$.