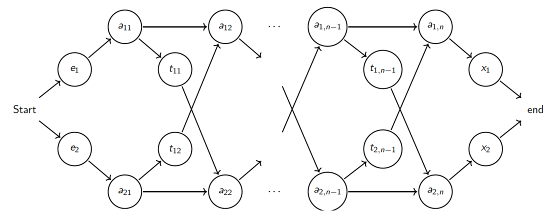


Dynamic Programming 2
Lecture 24

COSC 242 – Algorithms and Data Structures

Today’s outline

1. Longest common subsequence
2. Brute force LCS algorithm
3. Memoised and iterative implementations
4. Edit Distance

Today’s outline

1. Longest common subsequence
2. Brute force LCS algorithm
3. Memoised and iterative implementations
4. Edit Distance

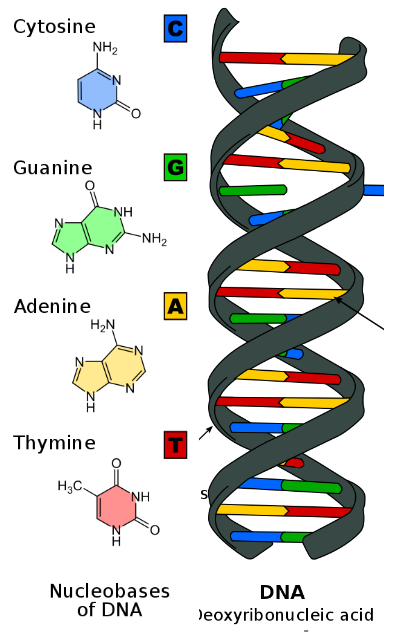
String similarity applications

Quantifying the similarity of strings has many applications, particularly in the realm Biological Sciences.

DNA consists of a string of molecules called nucleobases: Adenine, Guanine, Cytosine, and Thymine. These bases are typically represented by their initial letter.

We can therefore express a strand of DNA as a string over the finite set: $\{A, G, C, T\}$.

One reason to compare two strands of DNA is to determine how "similar" the strands are.



Review: Strings

A **String** over the finite set *S*, is a sequence of elements from *S*.

For example, there are 4 binary strings of length 2:

00, 01, 10, 11

A **substring** *s′* of a string *s* is an ordered sequence of consecutive elements of *s*.

A string of length *k* can be called a **k-string**. A **k-substring** of a string is a substring of length *k*.

<u>Example</u>: 00 is a 2-substring of 0100101 (beginning at position 3). But 11 is not a substring.

String similarity

Lets consider two strings:

- $S1$ = ACCGGTCGAGTGCGCGG
- $S2$ = GTCGTTCGGAATGCC

In this example, neither $S1$ nor $S2$ are a substring of the other. Yet they are clearly similar to one another.

Can we come up with some notion of how close these two strings are when the order of the letters is important?

Notions of similarity

- the longest substring common to both strings
- the number of changes to convert one string into another (this is called edit distance)
- the longest string, S_3, such that the characters in S_3 occur in both S_1 and S_2 in the same order, but not necessarily consecutively. This measure is called the **longest common subsequence** (LCS).

A subsequence of a given sequence is just the given sequence with zero or more elements left out. This is different to a substring, which requires consecutive common characters.

Class challenge

What is the LCS of the two strings:

- S_1 = ACCGGTCGAGTGCGCGG
- S_2 = GTCGTTCGGAATGCC

S_3 = ?

Scale of the problem

The class challenge probably took you a while.

Lets consider for a moment the scale of the problem facing biologists.

The human genome is thought to have over 46,000 genes[1]. Each gene varies from a few hundred DNA to over 2 million[2]. Overall, there are thought to be over 3 billion DNA base pairs[3].

Producing a computationally efficient method for substring similarity is an important problem, with profound impacts on human science.

Defining a subsequence

Given two sequences: $X = \langle x_1, x_2, \ldots, x_m \rangle$, and $Z = \langle z_1, z_2, \ldots, z_k \rangle$.

We say that *Z* is a **subsequence** of *X*, if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of *X*, such that for all *j* = 1, 2, …, *k*, we have $x_{i_j} = z_j$.

For example, $X = \langle A, G, C, G, T, A, G \rangle$, then $Z = \langle G, C, T, G \rangle$ is a subsequence of *X*, with the index sequence: $\langle 2, 3, 5, 7 \rangle$.

$$X = \left\langle \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ A, & G, & C, & G, & T, & A, & G \end{array} \right\rangle$$

Defining a common subsequence

Given two sequences X and Y, then Z is a **common subsequence** of X and Y, if Z is a subsequence of both X and Y.

Example: $X = \langle A, G, C, G, T, A, G \rangle$ and $Y = \langle G, T, C, A, G, A \rangle$.

The subsequence $\langle G, C, A \rangle$ is a common subsequence of X and Y. At length 3, it is not the longest common subsequence.

The subsequence $\langle G, C, G, A \rangle$, which is common to both X and Y, is a **Longest Common Subsequence** (LCS), at length 4. The subsequence $\langle G, T, A, G \rangle$ is also an LCS. $X = \langle A, \boldsymbol{G}, C, G, \boldsymbol{T}, \boldsymbol{A}, \boldsymbol{G} \rangle$ and $Y = \langle \boldsymbol{G}, \boldsymbol{T}, C, \boldsymbol{A}, \boldsymbol{G}, A \rangle$

There is no LCS of length 5 or more. Otherwise these would be, by definition, the LCS.

Longest common subsequence problem

In the **Longest Common Subsequence problem**, we want to find the longest common subsequence of two sequences, X and Y.

A brute force approach is to enumerate the set of all subsequences of X, and check if they are common to Y. Each subsequence of X has the subset of indices: $\{1,2,\ldots,m\}$ of X.

However as X has 2^m subsequences, this requires exponential time. Just like the 0-1 knapsack and Assembly line scheduling, also 2^n, we have a binary choice for each element: include or exclude.

As each subsequence then takes $\mathbf{\Theta}(n)$ times to check, this brute force approach takes: $\mathbf{\Theta}(n2^m)$.

Optimal substructure

As we will show on our next slide, the LCS problem exhibits optimal substructure. The theorem relies on pairs of prefixes that refer to the two input sequences.

Given a sequence $X = \langle x_1, x_2, \dots, x_m \rangle$, the *i*th **prefix** of X, with *i* = 1,2,…,*m*, corresponds to $X_i = \langle x_1, x_2, \dots, x_i \rangle$.

For example, $X = \langle A, G, C, G, T, A, G \rangle$, then $X_4 = \langle A, G, C, G \rangle$, while X_0 is the empty sequence. $X_7 = \langle A, G, C, G, T, A, G \rangle$ where *i* = *m* = 7.

Theorem of Optimal substructure in an LCS

Let $X = \langle x_1, x_2, \dots, x_m \rangle$, and $Y = \langle y_1, y_2, \dots, y_n \rangle$ be two strings.

Let $Z = \langle z_1, z_2, \dots, z_k \rangle$ be any **<u>LCS</u>** of X and Y.

Prefixes

Then the following holds:

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then:

   2.1 $z_k \neq x_m$ implies Z is an LCS of X_{m-1} and Y; or then

   2.2 $z_k \neq y_n$ implies Z is an LCS of X and Y_{n-1}.

Theorem (part 1)

If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}

Proof (through contradiction)

Suppose that the last characters z_k of Z and x_m of X were unequal: $z_k \neq x_m$. We could then append $x_m = y_n$ to the end of Z, such that $Z' = \langle z_1, \dots, z_k, x_m \rangle$. This would create a subsequence of length *k* + 1, contradicting our statement that Z is the *longest* common subsequence. Thus, $z_k = x_m = y_n$.

Next, assume there exists a common subsequence W of X_{m-1} and Y_{n-1} that's longer than Z_{k-1}, i.e. whose length ≥ *k*. Appending $x_m = y_n$ to the end of W, to make W'. This subsequence is a common subsequence of X and Y, where W' length ≥ *k + 1*, contradicting our statement that Z is the *longest* common subsequence. Thus, Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

Theorem (parts 2.1 and 2.2)

$z_k \neq x_m$ implies Z is an LCS of X_{m-1} and Y

$z_k \neq y_n$ implies Z is an LCS of X and Y_{n-1}

Proof (through contradiction)

2.1) If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. Suppose there exists a common subsequence W of X_{m-1} and Y whose length > *k*. Then W is a common subsequence of X and Y, contradicting Z being the *longest* common subsequence. Thus, Z is an LCS of X_{m-1} and Y.

2.2) The symmetric argument applies to part 2.2.

■

Today’s outline

1. Longest common subsequence
2. Brute force LCS algorithm
3. Memoised and iterative implementations
4. Edit Distance

Longest common subsequence problem

Our theorem gives us a very clear optimal substructure problem from which we can write a naive recursive solution (brute force).

Let $l[i,j]$ be the length of an LCS of the sequences X_i and Y_j. If either prefix is zero, $i = 0 \; or \; j = 0$, then the LCS has length 0. Then:

$$l[i,j] \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0\,, \\ l[i-1,j-1]+1, & \text{If } i,j > 0 \text{ and } x_i = y_j\,, \\ \max(l[i,j-1], l[i-1,j]) & \text{If } i,j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

1) Empty strings

2) Match char, or

3) Max of:
   a) X & Y-1
   b) X-1 and Y

Longest common subsequence problem

Therefore, when $x_i = y_j$, we should solve the subproblem of finding an LCS of X_{i-1} and Y_{j-1}. That is, our new length $l[i, j]$ is just our previous length $l[i-1, j-1]$, plus 1.

Otherwise, we should solve the two subproblems of finding an LCS of X_i and Y_{j-1} and X_{i-1} and Y_j. Whichever is largest, $l[i, j-1]$ or $l[i-1, j]$.

Naïve algorithm

```
function RecursiveLCS(X, Y)                            // String X, String Y
    if X.length == 0 or Y.length == 0
        return ""                                      // X or Y are empty strings (1)
    if X[m] == Y[n]
        return RecursiveLCS(X[1:m-1], Y[1:n-1]) + X[m] // Chars match (2)
    lcs1 = RecursiveLCS(X, Y[1:n-1])                   // Check LCS on X and Y-1 (3a)
    lcs2 = RecursiveLCS(X[1:m-1], Y)                   // Check LCS on X-1 and Y (3b)
    if lcs1.length > lcs2.length
        return lcs1
    else
        return lcs2
end procedure
```

where m = X.length and n = Y.length

Naïve algorithm

Base case: terminate when we go past the start of our string. Then we work forwards, assembling our solution.

```
function RecursiveLCS(X, Y)                                // String X, String Y
1:   if X.length == 0 or Y.length == 0
2:       return ""                                         // X or Y are empty strings (1)
3:   if X[m] == Y[n]
4:       return RecursiveLCS(X[1:m-1], Y[1:n-1]) + X[m]   // Chars match (2)
5:   lcs1 = RecursiveLCS(X, Y[1:n-1])                      // Check LCS on X and Y-1 (3a)
6:   lcs2 = RecursiveLCS(X[1:m-1], Y)                      // Check LCS on X-1 and Y (3b)
7:   if lcs1.length > lcs2.length
8:       return lcs1
9:   else
10:      return lcs2
end procedure
```

We start at the end of the string, and work our way backwards

where m = X.length and n = Y.length

Today’s outline

1. Longest common subsequence
2. Brute force LCS algorithm
3. Memoised and iterative implementations
4. Edit Distance

Memoised version

```
    initialise global memo[m, n] as 2D array, all cells set to -1
function MemoLCSRecurse(X, Y)                            // String X, String Y
    if m == 0 or n == 0
        return ""                                        // X or Y are empty strings (1)
    if memo[m, n] != -1                                  // Previously computed?
        return memo[m, n]                                // Yes, return cache
    if X[m] == Y[n]
        memo[m, n] = MemoLCSRecurse(X[1:m-1], Y[1:n-1] + X[m] // Chars match (2)
    else
        lcs1 = MemoLCSRecurse (X, Y[1:n-1])              // Check LCS on X and Y-1 (3a)
        lcs2 = RecursiveLCS(X[1:m-1], Y)                 // Check LCS on X-1 and Y (3b)
        if lcs1.length > lcs2.length
            memo[m, n] = lcs1
        else
            memo[m, n] = lcs2
    return memo[m, n]
end procedure
```

Iterative version

The memoized recursive version is fairly simple and efficient, but the strings do not need to be very long before we will blow the system stack.

If we're comparing DNA for example, then we could expect strings millions or billions of characters long. Even with an iterative version, billions of characters will cause a problem. Why?

Nevertheless, an iterative version will work for much larger problems than a recursive one.

Iterative version

In previous iterative dynamic programming solutions, we've been able to mirror the recursive version and also work backwards. But in this case, it makes more sense to run the algorithm forwards.

We could have done the other algorithms forwards too, or this one backwards, but it's better to choose the direction that's most natural based on the problem at hand.

The disadvantage of the forward algorithm is that all elements of the array will be filled, whereas the recursive backwards algorithm only fills those elements that are needed.

Iterative algorithm

```
function IteractiveLCS(X, Y)                         // String X, String Y
    memo = array of strings of size m+1 × n+1
    for i = 1 to m
        for j = 1 to n
            if X[i] == Y[j]
                memo[i+1, j+1] = memo[i,j] + X[i]  // Chars match (2)
            else
                if memo[i+1, j].length > memo[i, j+1].length
                    memo[i+1,j+1] = memo[i+1,j]   // LCS on X+1 and Y (3a)
                else
                    memo[i+1,j+1] = memo[i,j+1]   // LCS on X and Y+1 (3b)
end procedure
```

We can do better than this in terms of space efficiency. There is no need to keep the whole array in memory at one time - we just need to reference the current row and the previous row. That means instead of space with O(*mn*), we can use only O(*2n*).

Today’s outline

1. Longest common subsequence
2. Brute force LCS algorithm
3. Memoised and iterative implementations
4. Edit Distance

Edit Distance

The UNIX utility *diff* uses edit distance to find differences between two files (or strings).

This utility, or others like it, are used in all sorts of places, but most notably in version control systems such as *git* or *subversion*.

These version control systems are used to track all changes in a document or a group of documents and are especially useful for tracking changes in source code.

Edit distance can also be used to fix spelling errors.

Edit Distance

Edit distance is very similar to LCS. In LCS, we effectively only allow deletion from one string or the other in each step.

In edit distance, we try to transform one string into the other by allowing operations: insert, delete, or substitute; just as if we were editing the source string.

We aren't going to worry about the resulting alignment in the following, but it can be reconstructed from the edit distance array.

Remember, edit distance refers to the number of changes required to transform one string, X, into another, Y.

Edit distance recurrence

The recurrence for the edit distance is (transform X to Y):

$$d[i,j] \begin{cases} d[i-1,j-1], & \text{if } X[i] = Y[j] \\ min \begin{cases} d[i-1,j]+1, & \text{delete } X[i] \\ d[i,j-1]+1, & \text{delete } Y[j] \\ d[i-1,j-1]+1, & \text{substitute } Y[j]\ for\ X[i] \end{cases} & \text{if } X[i] \neq Y[j] \end{cases}$$

Where
$d[0,j] = j$: insert all characters up to $Y[j]$
and
$d[i,0] = i$: delete all characters up to $X[i]$

1) Match char, carry over previous distance, or
2) Min of
   a) Shortened X
   b) Shortened Y
   c) Copying Y char into X char

Iterative algorithm

```
function EditDistance(X, Y)                                  // String X, String Y
    if m == 0
        return n      // String X is empty, edit distance is Y.length
    if n == 0
        return m      // String Y is empty, edit distance is X.length
    if X[m] == Y[n]
        return EditDistance(X[1:m-1], Y[1:n-1])     // Chars match (1)
    del_xi = 1 + EditDistance(X[1:m-1], Y)           // Shortened X (2a)
    ins_yj = 1 + EditDistance(X, Y[1:n-1])           // Shortened y (2b)
    sub_yj4xi = 1 + EditDistance(X[1:m-1], Y[1:n-1]) // Sub y for x (2c)
    return min(del_xi, ins_yj, sub_yj4xi)
end procedure
```

Iterative algorithm

If either string is empty, the edit distance must be the remaining length of the other string (all substitutions).

```
function EditDistance(X, Y)                                 // String X, String Y
    if m == 0
        return n      // String X is empty, edit distance is Y.length
    if n == 0
        return m      // String Y is empty, edit distance is X.length
    if X[m] == Y[n]
        return EditDistance(X[1:m-1], Y[1:n-1])      // Chars match (1)
    del_xi = 1 + EditDistance(X[1:m-1], Y)           // Shortened X (2a)
    ins_yj = 1 + EditDistance(X, Y[1:n-1])           // Shortened y (2b)
    sub_yj4xi = 1 + EditDistance(X[1:m-1], Y[1:n-1]) // Sub y for x (2c)
    return min(del_xi, ins_yj, sub_yj4xi)
end procedure
```

Work our way backwards from end of string

Suggested reading

Longest common subsequence is discussed in section 15.4 of the textbook.

The edit distance problem is discussed at the end of chapter 5 in the Problems section (15.3).

The essential elements of dynamic programming are discussed in Section 15.3.

Solutions

Class challenge

What is the LCS of the two strings:

- S_1 = ACCGGTCGAGTGCGCGG
- S_2 = GTCGTTCGGAATGCC

S_3 = GGTCGGTGCC

The length of S_3 is 10. While there are many shortest common subsequences (e.g., ACC), S_3 is the *longest* common subsequence.

Image attributions