CSE 417 Algorithms and Complexity
Autumn 2023
Lecture 22
Longest Common Subsequence
Announcements

• Lecture plans
  – Monday: Longest Common Subsequence
  – Wednesday: Shortest Paths
  – Friday: No Class
  – After Thanksgiving: Network Flow + NP Completeness

• Homework plans
  – HW 8, Due Wednesday, November 29
  – HW 9, Due Friday, December 8
Last week, subset sum

- Given integers \{w_1, \ldots, w_n\} and an integer K
- Find a subset that is as large as possible that does not exceed K
- \text{Opt}[j, K] the largest subset of \{w_1, \ldots, w_j\} that sums to at most K
- \text{Opt}[j, K] = \max(\text{Opt}[j-1, K], \text{Opt}[j-1, K-w_j] + w_j)

\begin{align*}
\text{for } j &= 1 \text{ to } n \\
&\quad \text{for } k = 1 \text{ to } W \\
&\quad \text{Opt}[j, k] = \max(\text{Opt}[j-1, k], \text{Opt}[j-1, k-w_j] + w_j)
\end{align*}
Two dimensional dynamic programming

Subset sum and knapsack

\[ \text{Opt}[j, K] = \max(\text{Opt}[j - 1, K], \text{Opt}[j - 1, K - w_j] + w_j) \]

\[ \text{Opt}[j, K] = \max(\text{Opt}[j - 1, K], \text{Opt}[j - 1, K - w_j] + v_j) \]
Reducing dimensions

- Computing values in the array only requires the previous row
  - Easy to reduce this to just tracking two rows
  - And sometimes can be implemented in a single row
- Space savings is significant in practice
- Reconstructing values is harder
Longest Common Subsequence

• $C=c_1 \ldots c_g$ is a subsequence of $A=a_1 \ldots a_m$ if $C$ can be obtained by removing elements from $A$ (but retaining order)

• $LCS(A, B)$: A maximum length sequence that is a subsequence of both $A$ and $B$

occurranec attacggct

occurrence tacgacca
Determine the LCS of the following strings

BARTHOLEMEWSIMPSON

KRUSTYTHECLOWN
String Alignment Problem

- Align sequences with gaps
  
  CAT TGA  AT
  
  CAGAT AGGA

- Charge $\delta_x$ if character $x$ is unmatched
- Charge $\gamma_{xy}$ if character $x$ is matched to character $y$

Note: the problem is often expressed as a minimization problem, with $\gamma_{xx} = 0$ and $\delta_x > 0$. 
Recursive Version

LCS(a_1a_2...a_m, b_1b_2...b_n) {
    if (a_m == b_n)
        return LCS(a_1a_2...a_{m-1}, b_1b_2...b_{n-1}) + 1;
    else
        return max(LCS(a_1a_2...a_{m-1}, b_1b_2...b_n),
                    LCS(a_1a_2...a_m, b_1b_2...b_{n-1}));
}
LCS Optimization

• $A = a_1a_2...a_m$
• $B = b_1b_2...b_n$

• $\text{Opt}[j, k]$ is the length of $\text{LCS}(a_1a_2...a_j, b_1b_2...b_k)$
Optimization recurrence

If $a_j = b_k$,  $\text{Opt}[ j,k ] = 1 + \text{Opt}[ j-1, k-1 ]$

If $a_j \neq b_k$,  $\text{Opt}[ j,k] = \max(\text{Opt}[ j-1,k], \text{Opt}[ j,k-1])$
Give the Optimization Recurrence for the String Alignment Problem

- Charge $\delta_x$ if character $x$ is unmatched
- Charge $\gamma_{xy}$ if character $x$ is matched to character $y$

$$\text{Opt}[j, k] =$$

Let $a_j = x$ and $b_k = y$
Express as minimization
String edit with Typo Distance

- Find closest dictionary word to typed word
- $\text{Dist}('a', 's') = 1$
- $\text{Dist}('a', 'u') = 6$
- Capture the likelihood of mistyping characters
Dynamic Programming Computation
Code to compute Opt[ n, m]

for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        if (A[ i ] == B[ j ])
            Opt[ i,j ] = Opt[ i-1, j-1 ] + 1;
        else if (Opt[ i-1, j ] >= Opt[ i, j-1 ])
            Opt[ i, j ] := Opt[ i-1, j ];
        else
            Opt[ i, j ] := Opt[ i, j-1 ];
Storing the path information

\[ A[1..m], \ B[1..n] \]

for \( i := 1 \) to \( m \) \hspace{1cm} \text{Opt}[i, 0] := 0;
for \( j := 1 \) to \( n \) \hspace{1cm} \text{Opt}[0,j] := 0;
\text{Opt}[0,0] := 0;
for \( i := 1 \) to \( m \)
  for \( j := 1 \) to \( n \)
    if \( A[i] = B[j] \) \hspace{1cm} \{ \text{Opt}[i,j] := 1 + \text{Opt}[i-1,j-1]; \ \text{Best}[i,j] := \text{Diag}; \}
    else if \( \text{Opt}[i-1,j] \geq \text{Opt}[i,j-1] \)
      \{ \text{Opt}[i,j] := \text{Opt}[i-1,j], \ \text{Best}[i,j] := \text{Left}; \}
    else \hspace{1cm} \{ \text{Opt}[i,j] := \text{Opt}[i,j-1], \ \text{Best}[i,j] := \text{Down}; \}
Reconstructing Path from Distances
How good is this algorithm?

• Is it feasible to compute the LCS of two strings of length 300,000 on a standard desktop PC? Why or why not.
public int ComputeLCS() {
    int n = str1.Length;
    int m = str2.Length;

    int[,] opt = new int[n + 1, m + 1];
    for (int i = 0; i <= n; i++)
        opt[i, 0] = 0;
    for (int j = 0; j <= m; j++)
        opt[0, j] = 0;

    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            if (str1[i - 1] == str2[j - 1])
                opt[i, j] = opt[i - 1, j - 1] + 1;
            else if (opt[i - 1, j] >= opt[i, j - 1])
                opt[i, j] = opt[i - 1, j];
            else
                opt[i, j] = opt[i, j - 1];

    return opt[n, m];
}
N = 17000

Runtime should be about 5 seconds*

* Personal PC, 10 years old
public int SpaceEfficientLCS() {
    int n = str1.Length;
    int m = str2.Length;
    int[] prevRow = new int[m + 1];
    int[] currRow = new int[m + 1];

    for (int j = 0; j <= m; j++)
        prevRow[j] = 0;

    for (int i = 1; i <= n; i++) {
        currRow[0] = 0;
        for (int j = 1; j <= m; j++) {
            if (str1[i - 1] == str2[j - 1])
                currRow[j] = prevRow[j - 1] + 1;
            else if (prevRow[j] >= currRow[j - 1])
                currRow[j] = prevRow[j];
            else
                currRow[j] = currRow[j - 1];
        }
        for (int j = 1; j <= m; j++)
            prevRow[j] = currRow[j];
    }

    return currRow[m];
}
N = 300000

N: 10000 Base 2 Length: 8096  Gamma: 0.8096  Runtime:00:00:01.86
N: 20000 Base 2 Length: 16231  Gamma: 0.81155  Runtime:00:00:07.45
N: 30000 Base 2 Length: 24317  Gamma: 0.8105667  Runtime:00:00:16.82
N: 40000 Base 2 Length: 32510  Gamma: 0.81275  Runtime:00:00:29.84
N: 50000 Base 2 Length: 40563  Gamma: 0.81126  Runtime:00:00:46.78
N: 60000 Base 2 Length: 48700  Gamma: 0.8116667  Runtime:00:01:08.06
N: 70000 Base 2 Length: 56824  Gamma: 0.8117715  Runtime:00:01:33.36

N: 300000 Base 2 Length: 243605  Gamma: 0.8120167  Runtime:00:28:07.32
Observations about the Algorithm

• The computation can be done in $O(m+n)$ space if we only need one column of the Opt values or Best Values

• The computation requires $O(nm)$ space if we store all of the string information
Computing LCS in $O(nm)$ time and $O(n+m)$ space

- Divide and conquer algorithm
- Recomputing values used to save space

- Section 6.7 of the text, but we will not have time to cover in detail (so you are not responsible for section 6.7)
Divide and Conquer Algorithm

• Where does the best path cross the middle column?

• For a fixed i, and for each j, compute the LCS that has $a_i$ matched with $b_j$
Algorithm Analysis

- $T(m, n) = T(m/2, j) + T(m/2, n-j) + cnm$
- Solution: $T(m, n) \leq 2cnm$