CS 341: ALGORITHMS
Lecture 11: dynamic programming III
Readings: see website
Trevor Brown
https://student.cs.uwaterloo.ca/~cs341
trevor.brown@uwaterloo.ca

PROBLEM: LONGEST COMMON SUBSEQUENCE (LCS)

EXAMPLES
- X=aaaaa Y=bbbbb Z=LCS(X,Y)=Ø
- X=abcde Y=bcd Z=LCS(X,Y)=bcd
- X=abcde Y=labef Z=LCS(X,Y)=abe

POSSIBLE GREEDY SOLUTIONS?
Alg: for each 𝑥𝑖∈𝑋, try to choose a matching 𝑦𝑗∈𝑌 that is to the right of all previously chosen 𝑦𝑗 values
- X=azbracadabra Y=bracadabraz Z=az

POSSIBLE GREEDY SOLUTIONS?
Alg: for each 𝑥𝑖∈𝑋, try to choose a matching 𝑦𝑗∈𝑌 that is to the right of all previously chosen 𝑦𝑗 values
- X=azbracadabra Y=bracadabraz
- X=azbracadabra Y=bracadabraz [no 𝑦𝑗 after z]
- X=azbracadabra Y=bracadabraz [no 𝑦𝑗 after z]

DEFINING SUBPROBLEMS
- Full problem: return |LCS(𝑋, 𝑌)| (i.e., length of LCS)
- Reduce size by taking prefixes of 𝑋 or 𝑌
- Let 𝑋𝑖=(𝑥1,...,𝑥𝑖) and 𝑌𝑖=(𝑦1,...,𝑦𝑖)
  \[
  \begin{array}{cccccccc}
  x_1 & x_2 & \cdots & x_i \\
  y_1 & y_2 & \cdots & y_i \\
  \end{array}
  \]
  Note 𝑋=𝑋_𝑛 and 𝑌=𝑌_𝑛
- Subproblem: return |LCS(𝑋_𝑖, 𝑌_𝑖)|
- Idea for recurrence: remove the last letter of 𝑋 or 𝑌
**Building Solutions from Subproblems**

**Example #1 to Build Intuition**

- This cannot be the final $e$ in $Z$
- $x_m$ is a subsequence of $x_i$
- $x_i$ is a must be in $X_{m+1}$
- $x_m$ is not a must be in $Y_{m+1}$
- This cannot be the final $e$ in $Z$

**Example #2**

- Or maybe this is
- But this certainly is not :)
### PSEUDO CODE

Algorithm: LCS($X = (x_1, \ldots, x_m), Y = (y_1, \ldots, y_n)$)

1. for $i$ from 0 to $m$
   - $c[i, 0] = 0$
2. for $j$ from 0 to $n$
   - $c[0, j] = 0$
3. for $i$ from 1 to $m$
   - for $j$ from 1 to $n$
     - if $x_i = y_j$
       - $c[i, j] = c[i-1, j-1] + 1$
     - else
       - $c[i, j] = \max(c[i-1, j], c[i, j-1])$
4. return $c[m, n]$

### COMPUTING THE LCS

**Not Just Its Length**

To make it easy to find the actual LCS (not just its length), we keep track of which particular entry was used to calculate $c[i, j]$. We store the direction to that entry in an array $s[i, j]$.

**Example**

Consider which table entry was used to calculate $c(4, 2)$.

**Case:** $c(4, 2) = c(3, 1) + 1$
- We store "U" in $s[4, 2]$ to indicate decreasing $i$. (To get $c(1, 0)$ before $c(1, 1)$.)
- Recall in this case, $x_4 = y_2$, so we include $x_4$ in the LCS.

**Case:** $c(4, 2) = c(4, 1)$
- This is a no-op for both $i$ and $j$. We store "D" in $s[4, 2]$ to indicate decreasing $j$. (To get $c(4, 0)$ before $c(4, 1)$.)

**Case:** $c(4, 2) = c(3, 2)$
- We store "L" in $s[4, 2]$ to indicate decrementing $j$. (To get $c(1, 2)$ before $c(1, 1)$.)

**How to obtain LCS=gvet from this table?**

---

### SAVING THE DIRECTION TO THE PREDECESSOR SUBPROBLEM π

Suppose $X = \text{gdevtega}$ and $Y = \text{gycekst}$.

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>0</td>
</tr>
<tr>
<td>$j$</td>
<td>0</td>
</tr>
</tbody>
</table>

Suppose $X = \text{gdevtega}$ and $Y = \text{gycekst}$.

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>0</td>
</tr>
<tr>
<td>$j$</td>
<td>0</td>
</tr>
</tbody>
</table>

---

### COMPARISON OF SPACE/TIME

- **Space:** $O(mn)$
- **Time:** $O(mn)$

---

### EQ

$$c(i, j) = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
max(c(i-1, j-1), c(i-1, j), c(i, j-1)) & \text{otherwise}
\end{cases}$$
FOLLOWING PREDECESSORS TO COMPUTE THE LCS

```python
1. Find_LCS(c[0...m], m[0...n], X[0...n])
2. lcs = new string
3. i = n
4. j = n
5. while i > 0 and j > 0
6.   if m[i][j] == "I"
7.     lcs.append(X[i])
8.     i--
9.   else if m[i][j] == "J"
10.    j--
11.   else // m[i][j] == "I"
12.    i--
13. return reverse(lcs)
```

Complexities of this trace-back algo:
Space? Time?
(word RAM model)
space: $O(n+m)$ words
time: $O(n+m)$

PROBLEM: MINIMUM LENGTH TRIANGULATION

Input: $n$ points $q_1, q_2, \ldots, q_n$ in 2D space that form a convex $n$-gon $P$
- Assume points are sorted clockwise around the center of $P$
Find: a triangulation of $P$ such that the sum of the perimeters of the $n-2$ triangles is minimized
Output: the sum of the perimeters of the triangles in $P$

HOW HARD IS THIS PROBLEM?

- How many triangulations are there?
- Number of triangulations of a convex $n$-gon = the $(n-2)$th Catalan number
- This is $C_{n-2} = \frac{1}{n-1} \binom{2n-4}{n-2}$
- It can be shown that $C_{n-2} = \Theta(4^n/(n-2)^{3/2})$

PROBLEM DECOMPOSITION

The edge $q_k q_l$ is in a triangle with a third vertex $q_o$, where $k \in \{2, \ldots, n-1\}$.
For a given $k$, we have:
the triangle $q_k q_o q_l$.  (1)

PROBLEM DECOMPOSITION

The edge $q_k q_l$ is in a triangle with a third vertex $q_o$, where $k \in \{2, \ldots, n-1\}$.
For a given $k$, we have:
the triangle $q_k q_o q_l$.  (1)
the polygon with vertices $q_1, \ldots, q_n$.  (2)
PROBLEM DECOMPOSITION
The edge \( q_0 q_i \) is in a triangle with a third vertex \( q_k \), where \( k \in \{2, \ldots, n-1\} \).

For a given \( k \), we have:
1. the triangle \( q_0 q_i q_k \)
2. the polygon with vertices \( q_1, \ldots, q_k \)
3. the polygon with vertices \( q_k, \ldots, q_i \)

The optimal solution will consist of optimal solutions to the two subproblems in (2) and (3), along with the triangle in (1).

PROBLEM DECOMPOSITION
The edge \( q_0 q_i \) is in a triangle with a third vertex \( q_k \), where \( k \in \{2, \ldots, n-1\} \).

For a given \( k \), we have:
1. the triangle \( q_0 q_i q_k \)
2. the polygon with vertices \( q_1, \ldots, q_i \)
3. the polygon with vertices \( q_i, \ldots, q_k \)

The optimal solution will consist of optimal solutions to the two subproblems in (2) and (3), along with the triangle in (1).

RECURRENCE RELATION
Let \( S(i, j) \) be optimal solution to the subproblem consisting of the polygon with vertices \( q_i \ldots q_j \)

Let \( \Delta_{ijk} \) denote \( \text{perimeter}(q_i q_j q_k) \)

If a given triangle \( q_i q_j q_k \) is in the optimal solution, then

\[
S(i, j) = S(i, k) + \Delta_{ijk} + S(k, j)
\]

But we don't know the optimal \( k \)

Minimize over all \( k \) strictly between \( i \) and \( j \)

\[
S(i, j) = \begin{cases} 
\min \{ S(i, k) + \Delta_{ijk} + S(k, j) \} & \text{if } j \geq i + 2 \\
0 & \text{otherwise}
\end{cases}
\]

FILLING IN THE TABLE
- Table \( S[1..n, 1..n] \) of solutions to \( S(i, j) \) for all \( i, j \in \{1..n\} \)

Dependencies:
\[ S[i, j] \text{ and } S[k, l] \]
\[ \text{for } k = \{i+1, \ldots, j-1\} \]

We depend on larger \( j \)
And some \( i \) but smaller \( j \)
What's a correct fill order?
For \( i = n, 1 \), For \( j = 1, n \)
**Runtime**

**Word RAM Model**

- Number of subproblems: \( n^2 \)
- Time to solve subproblem: \( S(i, j): O(j - i) \subseteq O(n) \)
- So total runtime is in \( O(n^3) \)
- More effort needed to show \( \Omega(n^3) \), since so many subproblems are base cases, which take \( \Theta(1) \) steps
- Incidentally, this is polynomial time (in the input size)
- But basic runtime analysis does not require such an argument

\[
S(i, j) = \min \left( \frac{f(k)}{S(k, j)} + S(k, j) \right), \quad \forall j \geq i + 2
\]

**Memoization: An Alternative to DP**

Recall that the goal of dynamic programming is to eliminate solving subproblems more than once.

Memoization is another way to accomplish the same goal. Memoization is a recursive algorithm based on some recurrence relation as would be used by a dynamic programming algorithm.

The idea is to remember which subproblems have been solved; if the same subproblem is encountered more than once during the recursion, the solution will be looked up in a table rather than being re-calculated.

This is easy to do if initialize a table of all possible subproblems having the value undefined in every entry.

Whenever a subproblem is solved, the table entry is updated.

**Example: Using Memoization to Compute Fibonacci Numbers Efficiently**

```plaintext
main
for i ← 2 to n
    do M[i] ← -1
return (RecFib(n))
```

**Procedure RecFib(n)**

- If \( M[n] \) is already computed, don't recurse!
- If \( n = 0 \) then \( f ← 0 \)
- Else if \( n = 1 \) then \( f ← 1 \)
- Else if \( M[n] \neq -1 \) then \( f ← M[n] \)
- \( f ← f + f[i-2] \)
- \( M[n] ← f \)
- return \( f \)

**Visualization of Memoization**

- If \( M[n] \) is already computed, don't recurse!
- If \( n = 0 \) then \( f ← 0 \)
- Else if \( n = 1 \) then \( f ← 1 \)
- Else \( f ← f[i-2] \)
- \( M[n] ← f \)
- return \( f \)

**Model Comparison**

- **Word RAM**
  - Each variable (or array entry, etc.) is a word
  - All words have the same size
  - \( O(\log W) \) bits where \( W \) is size of words in the input
  - Runtime is the number of operations on words (where each word operation takes \( O(1) \) time)
  - Space complexity is the number of words used
- **Bit Complexity**
  - Each variable \( x \) is a bit string
  - Variables can have different numbers of bits
  - \( \log x \) is encoded in \( O(\log x) \) bits
  - Runtime is the number of operations on bits (where each bit operation takes \( O(1) \) time)
  - Space complexity is the number of bits used
CALCULATIONS USING INPUT SIZE

Clarification: you can compute space/time complexity without calculating the input size
- Input size calculations are typically only needed if we ask you to show an algorithm runs in polytime
- "runs in polytime" means the runtime is at most a polynomial in the number of bits in the input
- Lots of this in tractability / NP completeness

Just trying to expose you to these ideas ahead of time...

SO... IS DP LCS A POLYTIME ALGORITHM?

- Is \( nm \) polynomial in the input size (\# of bits in the input)?
  - Word RAM model
    - Input contains \( \Theta(n + m) \) words
    - Word RAM model says each word stores \( \Theta(\log w) \) bits
      where \( w = \# \) words in the input
      - So in this case, \( \Theta(\log(n + m)) \) bits per word
      - \( S \in \Theta((n + m)\log(n + m)) \)
      - Want a term that looks like \( nm \)

SO... IS DP LCS A POLYTIME ALGORITHM?

- Is \( nm \) polynomial in the input size (\# of bits in the input)?
  - Calculation using words would be simpler...
    - Words in the input \( W \in \Theta(n + m) \)
    - \( W^2 \in \Theta((n + m)^2) = \Theta(n^2 + nm + m^2) \)
    - So \( \Theta(nm) \subseteq \Theta(W^2) \)
      - I.e., polynomial in the \# of words in the input

  - How does this help us?
    - \# words \( W \) in the input \( \leq \# \) bits \( S \) in the input
      - So \( \Theta(nm) \subseteq \Theta(W^2) \) implies \( \Theta(nm) \subseteq \Theta(S^2) \)

So, for showing polytime, you can calculate the input size using words (and justify like this)