CS 341: ALGORITHMS
Lecture 9: dynamic programming III
Readings: see website
Trevor Brown
https://student.cs.uwaterloo.ca/~cs341
trevor.brown@uwaterloo.ca

PROBLEM: MINIMUM LENGTH TRIANGULATION

Input: \( n \) points \( q_1, \ldots, q_n \) 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 \( (n-2) \)nd Catalan number
This is \( C_{n-2} = \frac{1}{n-1} \binom{2n-4}{n-2} \)
It can be shown that \( C_{n-2} \in \Theta \left( \frac{4^n}{\sqrt{n}} \right) \)

PROBLEM DECOMPOSITION
The edge \( q_k q_1 \) is in a triangle with a third vertex \( q_j \), where \( k \in [2, \ldots, n-1] \).
For a given \( k \), we have:
the triangle \( q_k q_1 q_j \). (1)

PROBLEM DECOMPOSITION
The edge \( q_k q_1 \) is in a triangle with a third vertex \( q_j \), where \( k \in [2, \ldots, n-1] \).
For a given \( k \), we have:
the triangle \( q_k q_1 q_j \). (1)
the polygon with vertices \( q_1, \ldots, q_k \). (2)
**Problem Decomposition**

The edge $q_1q_3$ is in a triangle with a third vertex $q_k$, where $k \in \{2, \ldots, n-1\}$.

For a given $k$, we have:
- the triangle $q_1q_3q_k$ (1)
- the polygon with vertices $q_1, \ldots, q_k$ (2)
- the polygon with vertices $q_k, \ldots, q_n$ (3)

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 the optimal solution to the subproblem consisting of the polygon with vertices $q_i \ldots q_j$.
- Let $\Delta_{ijk}$ denote $\text{perimeter}(i,j,k)$.

If a given triangle $q_iq_jq_k$ 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:
- $S[i,k]$ and $S[k,j]$ for $k = i+1, \ldots, j-1$
- $S[k,j]$ for $k = j+1$
- $S[i,j]$ for $i = 1, \ldots, j$

We depend on larger $i$ and smaller $j$. What is the correct fill order?
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 \(O(n^3)\)
- Some 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

\[ S(i,j) = \begin{cases} 
S(k,k) + \Delta_{jk} + S(k,j), & \text{if } |j-i| \geq 1 + \delta \\
0, & \text{otherwise}
\end{cases} \]

PROBLEM: LONGEST COMMON SUBSEQUENCE (LCS)

Let's first solve for the length of the LCS

EXAMPLES

- \(X=aaaaa\) \(Y=bbbb\) \(Z=LCS(X,Y)=\emptyset\)
- \(X=abcde\) \(Y=bc\) \(Z=LCS(X,Y)=bc\)
- \(X=abcde\) \(Y=labef\) \(Z=LCS(X,Y)=abe\)

POSSIBLE GREEDY SOLUTIONS?

Alg: for each \(x_i \in X\), try to choose a matching \(y_j \in Y\) that is to the right of all previously chosen \(y_j\) values

- \(X=abcd\) \(Y=bc\)
- \(X=abcd\) \(Y=ab\)
- \(X=abcd\) \(Y=labef\) [no suitable \(y_j\) found]
- \(X=abcd\) \(Y=labef\) [no suitable \(y_j\) found]
- \(X=abcd\) \(y=labef\)
- \(Z=abe\) Optimal?

POSSIBLE GREEDY SOLUTIONS?

Alg: for each \(x_i \in X\), try to choose a matching \(y_j \in Y\) that is to the right of all previously chosen \(y_j\) values

- \(X=bracadabra\) \(Y=bracadabra\)
- \(X=bracadabra\) \(Y=bracadabra\) [no \(y_j\) after \(z\)]
- \(X=bracadabra\) \(Y=bracadabra\) [no \(y_j\) after \(z\)]

DEFINING SUBPROBLEMS

- Full problem: \(LCS(X,Y)\) [i.e., length of LCS]
- Reduce size by taking prefixes of \(X\) or \(Y\)
- Let \(X_1 = (x_1, \ldots, x_k)\) and \(Y_1 = (y_1, \ldots, y_l)\)

Note \(X = X_m\) and \(Y = Y_m\)
- Subproblem: \(LCS(X_1, Y_1)\)
- Shrinking the problem: remove the last letter of \(X\) or \(Y\)
BUILDING SOLUTIONS FROM SUBPROBLEMS
EXAMPLE #1 TO BUILD INTUITION

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
<td>3</td>
</tr>
</tbody>
</table>

This cannot be the final a in Z

Neither of these is a must be in Y_{m-1}

So, Z = LCS(Y_{m-1}, Y_{n-1})

Consider optimal solution Z = LCS(Y_{m-1}, Y_{n-1})

Since x_{m-1} \notin Z we know Z = LCS(x_{m-1}, Y_{n-1})

In particular, this means Z = LCS(Y_{m-1}, Y_{n-1})

BUILDING SOLUTIONS FROM SUBPROBLEMS
EXAMPLE #2

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
<td>3</td>
</tr>
</tbody>
</table>

There might be the final a in Z

Since Z is a subsequence of Y, x_{j} a must be in Y_{m-1}

Z_{m-1} = \max \{Z_{m-1}, Z_{m-1}, Z_{m-1} + x_{j}\}

Since y_{k} \notin Z we know Z = LCS(Y_{m-1}, Y_{n-1})

CASES 2 & 3: trivial

If one string is empty, so r \neq x_{c} Z_{1} = 0

Try all cases and maximize

Can simplify!

SUMMARIZING CASES

- \( x_{k} \) matches neither \( x_{m} \) nor \( y_{n} \) \( (x_{m} \neq y_{n}) \) \( Z = LCS(x_{m-1}, Y_{n-1}) \)
- \( x_{k} \) matches \( x_{m} \) but not \( y_{n} \) \( (x_{m} \neq y_{n}) \) \( Z = LCS(Y_{m-1}, Y_{n-1}) \)
- \( x_{k} \) matches \( y_{n} \) but not \( x_{m} \) \( (x_{m} \neq y_{n}) \) \( Z = LCS(x_{m-1}, Y_{n-1}) \)
- \( x_{k} \) matches both \( (x_{m} = y_{n}) \) \( Z = LCS(x_{m-1}, Y_{n-1}) + x_{j} \)

- \( \ldots \) but we don't know \( z_{i} \)
- Try all cases and maximize
- Careful: last case is only valid if \( x_{m} = y_{n} \)
- Also note \( x_{m} = y_{n} \) only holds in the last case
- Cases 2 & 3: trivial
- Case 1: if \( x_{m} \neq y_{n} \) then we can improve \( Z \) (contra)

DERIVING A RECURRENCE

Recall \( Z = LCS(x_{m}, Y_{n}) \)

- \( x_{k} \) matches neither \( x_{m} \) nor \( y_{n} \) \( (x_{m} \neq y_{n}) \) \( Z = LCS(x_{m-1}, Y_{n-1}) \)
- \( x_{k} \) matches \( x_{m} \) but not \( y_{n} \) \( (x_{m} \neq y_{n}) \) \( Z = LCS(Y_{m-1}, Y_{n-1}) \)
- \( x_{k} \) matches \( y_{n} \) but not \( x_{m} \) \( (x_{m} \neq y_{n}) \) \( Z = LCS(x_{m-1}, Y_{n-1}) \)
- \( x_{k} \) matches both \( (x_{m} = y_{n}) \) \( Z = LCS(x_{m-1}, Y_{n-1}) + x_{j} \)

Let \( c(i,j) = |LCS(x_{i}, y_{j})| \)

- Brainstorming sensible base cases
  - if \( i = 0 \) one string is empty, so \( c(0,j) = 0 \) [similarly for \( j = 0 \)]
  - General cases
    - \( c(i,j) = c(i-1,j-1) + 1 \) if \( x_{m} = y_{n} \)
    - \( c(i,j) = \max(c(i-1,j-1), c(i-1,j), c(-1,j)) \) if \( x_{m} \neq y_{n} \)

RECURRENCES

Combining expressions

\[
c(i,j) = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
(i-1,j-1) + 1 & \text{if } i, j \geq 1 \text{ and } x_{i} = y_{j} \\
\max(c(i,j-1), c(i-1,j), c(i-1,j)) & \text{if } i, j \geq 1 \text{ and } x_{i} \neq y_{j}
\end{cases}
\]

Can simplify!

Observe \( c(i-1,j-1) \leq c(i-1,j) \) (former is a subproblem of the latter)

\[
c(i,j) = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
(i-1,j-1) + 1 & \text{if } i, j \geq 1 \text{ and } x_{i} = y_{j} \\
\max(c(i,j-1), c(i-1,j)) & \text{if } i, j \geq 1 \text{ and } x_{i} \neq y_{j}
\end{cases}
\]
Suppose $X = gdvegta$ and $Y = gveckat$

<table>
<thead>
<tr>
<th>$X$</th>
<th>$i = 0$</th>
<th>$i = 1$</th>
<th>$i = 2$</th>
<th>$i = 3$</th>
<th>$i = 4$</th>
<th>$i = 5$</th>
<th>$i = 6$</th>
<th>$i = 7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$j = 0$</td>
<td>g</td>
<td>d</td>
<td>v</td>
<td>e</td>
<td>t</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$j = 1$</td>
<td>g</td>
<td>d</td>
<td>v</td>
<td>e</td>
<td>t</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$j = 2$</td>
<td>c</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$j = 3$</td>
<td>e</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$j = 4$</td>
<td>k</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$j = 5$</td>
<td>a</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$j = 6$</td>
<td>t</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PSEUDOCODE**

Algorithm: LCS($X = (x_1 \ldots x_n), Y = (y_1 \ldots y_m)$)

for $i = 1$ to $m$
for $j = 1$ to $n$

c[i][j] = 0
if $x_i = y_j$ then
    c[i][j] = c[i-1][j-1] + 1
else
    c[i][j] = max(c[i-1][j], c[i][j-1])
return c[m][n];

**COMPUTING THE LCS**

To make it easy to find the actual LCS (not just its length),

Consider which table entry was used to calculate c[i][j]

Case 1: c[i][j] = c[i-1][j-1] + 1

Case 2: c[i][j] = c[i-1][j] + 1

Case 3: c[i][j] = c[i][j-1] + 1

We store the direction to that entry in an array $x[i][j]$ to indicate positioning:

Case 1: $x[i][j] = c[i-1][j-1] + 1$

Case 2: $x[i][j] = c[i-1][j] + 1$

Case 3: $x[i][j] = c[i][j-1] + 1$

Recall in the case, $x_i = y_j$, so we include $x_i$ in the LCS

**SAVING THE DIRECTION TO THE PREDECESSOR SUBPROBLEM $\pi$**

Case: $c[i][j] = c[i-1][j-1] + 1$

We store "I" in $x[i][j]$ to indicate positioning both $i$ and $j$.

In our example table we just draw an arrow to the entry.

Case: $c[i][j] = c[i-1][j] + 1$

We store "I" in $x[i][j]$ to indicate positioning both $i$ and $j$. This is not in the LCS.

Case: $c[i][j] = c[i][j-1] + 1$

We store "I" in $x[i][j]$ to indicate positioning both $i$ and $j".

**Example**

Suppose $X = gdvegta$ and $Y = gveckat$.

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

Done

seq-gvet

seq-gvet

seq-gvet

seq-gvet

seq-gvet
FOLLOWING PREDECESSORS TO COMPUTE THE LCS

1. FindLCS(c[0...n][0...m], m[0...n][0...m], X[0...n])
2. lcs = new string
3. i = n
4. j = m
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] == "*"
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)

UNLIKELY TO GET THIS FAR
So this is likely just an exercise for you…

COIN CHANGING

There is a denomination with unit value!

In 0-1 knapsack, we only considered two subproblems in our recurrence:
taking on item, or not.

Here we can do more than use a coin denomination or not!

There are: some sensible base cases[s]?

General case:
What are the different ways we could use coin denomination d_{ij}?  
What subproblems / solutions should we use?

Find recurrence relation

Also N[i,0] = 0 for all i.

COIN CHANGING

Let N[i, t] denote the optimal solution to the subproblem consisting of
the first i coin denominations d_{1}, …, d_{i} and target sum t.

Exploring: some sensible base case[s]?

General case:
What are the different ways we could use coin denomination d_{ij}?
What subproblems / solutions should we use?

Final recurrence relation

Let N[i, t] denote the optimal solution to the subproblem consisting of
the first i coin denominations d_{1}, …, d_{i} and target sum t.

General case:
What are the different ways we could use coin denomination d_{ij}?
What subproblems / solutions should we use?

Final recurrence relation

Problem 5.2

Coin Changing

Instance: A list of coin denominations, d_{1}, d_{2}, …, d_{n}, and a
positive integer T, which is called the target sum.

Find: A tuple of non-negative integers, say A = [a_{1}, …, a_{n}], such
that \( \sum_{i=1}^{n} a_{i} \cdot d_{i} = T \) and such that N = \( \sum_{i=1}^{n} a_{i} \cdot d_{i} \), is minimized.

What subproblems should be considered?
What value should we fill in?

In 0-1 knapsack, we only considered two subproblems in our recurrence:
taking on item, or not.
Let \( N[i, t] \) denote the optimal solution to the subproblem consisting of the first \( i \) coin denominations \( d_1, \ldots, d_i \) and target sum \( t \).

Since \( d_1 = 1 \), we immediately have \( N[1, t] = t \) for all \( t \).

For \( i \geq 2 \), the number of coins of denomination \( d_i \) is an integer \( j \) where \( 0 \leq j \leq \lfloor t/d_i \rfloor \).

If we use \( j \) coins of denomination \( d_i \), then the target sum is reduced to \( t - jd_i \), which we must achieve using the first \( i - 1 \) coin denominations.

Thus we have the following recurrence relation:

\[
N[i, t] = \begin{cases} 
\min \{ j + N[i - 1, t - jd_i] : 0 \leq j \leq \lfloor t/d_i \rfloor \} & \text{if } i \geq 2 \\
0 & \text{if } i = 1 \text{ or } t = 0
\end{cases}
\]

**FILLING THE ARRAY**

\[ N[1 \ldots n, 0 \ldots T] = \begin{cases} 
\min \{ j + N[i - 1, t - jd_i] : 0 \leq j \leq \lfloor t/d_i \rfloor \} & \text{if } i \geq 2 \\
0 & \text{if } i = 1 \text{ or } t = 0
\end{cases} \]

No data dependencies on any other array cells.

Consider cell \( N[i, t] \)

\[ N[i, t] = \min \{ j + N[j - 1, t - jd_i] : 0 \leq j \leq \lfloor t/d_i \rfloor \} \quad \text{if } i \geq 2 \\
0 \quad \text{if } i = 1 \text{ or } t = 0
\]

We only look at the previous row!

It is sufficient to fill:

row \( i = 1 \) (base case), then

for \( i = 2 \ldots n \),

for \( t = 0 \ldots T \),

**OUTPUTTING OPTIMAL SET OF COINS**

Recall \( J[i, t] = \# \text{ of coins of type } d_i \text{ used in } N[i, t] \)

We start at \( J[n, T] = \# \text{ of coins of type } d_n \text{ used in the optimal solution} \)

Exercise for later:

compute the correct output without using \( J[i, t] \)

\( \text{i.e., using only } N, d, T \)
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 same 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

```python
def RecFib(n):
    if M[n] is already computed, don't recurse!
    M[n] = f
    if n = 0 then f ← 0
    else if n = 1 then f ← 1
    else if M[n] ≠ -1 then f ← M[n]
    else f ← RecFib(n - 2) + RecFib(n - 1)
    return f.
```

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

VISUALIZING MEMOIZATION

If M[n] is already computed, don't recurse!

```
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 M[n] ≠ -1 then f ← M[n]
else f ← RecFib(n - 2) + RecFib(n - 1)
return f.
```

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