CSE 101, Winter 2018

Design and Analysis of Algorithms

Lecture 11: Dynamic Programming, Part 2

Class URL: http://vlsicad.ucsd.edu/courses/cse101-w18/

Goal: continue with DP (Knapsack, All-Pairs SPs, …)
Problem: Given \(x[1..m]\) and \(y[1..n]\), find LCS

\[x: \quad A \quad B \quad C \quad B \quad D \quad A \quad B\]

\[y: \quad B \quad D \quad C \quad A \quad B \quad A\]

\[\Rightarrow \quad B \quad C \quad B \quad A\]

Brute-force algorithm:
- for every subsequence of \(x\), check if it is in \(y\)
- \(O(n2^m)\) time
  - \(2^m\) subsequences of \(x\) (each element is either in or out of the subsequence)
  - \(O(n)\) for scanning \(y\) with \(x\)-subsequence \((m \leq n)\)

What is a subproblem of LCS\((x,y)\)?
Small Examples, Intuition

Let’s use $\varepsilon$ to denote the empty string (zero chars)

- $\text{LCS} (A, \varepsilon) = \varepsilon$
- $\text{LCS} (A, B) = \varepsilon$
- $\text{LCS} (A, BD) = \varepsilon$
- $\text{LCS} (AB, B) = B$
- $\text{LCS} (AB, BD) = B = \text{LCS} (AB, B)$
- $\text{LCS} (ABC, BDC) = \text{LCS} (AB, BD) + \text{“C”} = BC$
Recurrent Formula for LCS

- Let $c[i,j] = \text{length of LCS of } X[i]=x[1..i], Y[j]=y[1..j]$
- Then $c[m,n] = \text{length of LCS of } x \text{ and } y$
- Claim:
  - if $x[i] = y[j]$ then $c[i,j] = c[i-1,j-1] + 1$
  - otherwise $c[i,j] = \max(c[i,j-1], c[i-1,j])$

Proof: $x[i] = y[j] \Rightarrow \text{LCS([X[i], Y[j]])} = \text{LCS(X[i-1], Y[j-1])} + x[i]$

**IN OTHER WORDS**

- 0) $c(i,j) \geq c(i-1, j-1)$
- 1) $c(i,j) = c(i-1, j-1) + 1$ if $x_i = y_j$
- 2) $c(i,j) \geq c(i-1, j)$ when $x_i$ is not in LCS
- 3) $c(i,j) \geq c(i, j-1)$ when $y_j$ is not in LCS

**Boundary Conditions:**

$\begin{cases} 
  c[i,0] = 0 & \forall i \\
  c[0,j] = 0 & \forall j 
\end{cases}$
DP for LCS: Pseudocode

For i = 0..m:  \( T[i,0] = 0 \)  // initialize left col
For j = 0..n:  \( T[0,j] = 0 \)  // initialize top row
For j = 1..n:  // column
    For i = 1..m:  // row
        if \( x[i] = y[j] \) then \( T[i,j] = 1 + T[i-1,j-1] \)
        else \( T[i,j] = \max(T[i-1,j], T[i,j-1]) \)
Return \( T[m,n] \)

Other DP solutions for string problems (e.g., “minimum edit distance”) will look similar
DP Table: Efficient Ordering of Subproblems

• After computing solution of a subproblem, store in *table*
• Time = \(O(mn)\)
• When computing \(c[i,j]\) we need \(O(1)\) time if we have:
  - \(x[i], y[j]\)
  - \(c[i,j-1]\)
  - \(c[i-1,j]\)
  - \(c[i-1,j-1]\)
### DP Table for LCS

<table>
<thead>
<tr>
<th></th>
<th>y</th>
<th>B</th>
<th>D</th>
<th>C</th>
<th>A</th>
<th>A</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### DP Table for LCS

<table>
<thead>
<tr>
<th></th>
<th>y</th>
<th>B</th>
<th>D</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The table shows the dynamic programming approach to find the Longest Common Subsequence (LCS) of two strings. The table is filled with the maximum lengths of subsequences up to each character of the input strings. The final answer can be obtained by following the path from the bottom right to the top left of the table, which is marked with red circles. In this case, the LCS is `AB` with a length of 2.
**DP Table for LCS**

<table>
<thead>
<tr>
<th></th>
<th>y</th>
<th>B</th>
<th>D</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Before Going On …

Recurrence = ?  Subproblem = ?

“Table” = ?

Dimension = ?  Order = ?
DP: Knapsack (Section 6.4)

- **KNAPSACK:** You are going camping. Each item type \(i\) that can go into the knapsack has a weight \(w_i\) and a value \(v_i\). You can carry up to a given weight limit \(b\). What should go into the knapsack so as to **maximize** the total value?
  - N.B.: This is a type of “integer program” that has just one constraint

- Notation: Define \(F_k(y) = \max_{0 \leq k \leq n} \left( \sum_{j=1}^{k} w_j x_j \right)\) with \(\sum_{j=1}^{k} v_j x_j \) \(\leq y \leq b\)

- \(F_k(y)\) is the maximum value possible using only the first \(k\) item types, when the weight limit is \(y\).
DP: Knapsack

• B.C.’s:  
  1. $F_0(y) = 0 \ \forall y$ \quad when 0 item types are allowed
  2. $F_k(0) = 0 \ \forall k$ \quad when weight limit is 0
  3. $F_1(y) = \lfloor \frac{y}{w_1}\rfloor v_1$ \quad first row is easy

DP recurrence:

$$F_k(y) = \max\{ F_{k-1}(y), F_k(y-w_k)+v_k \}$$

• Build a DP table
DP: Knapsack

• Example:  \( k = 4, \ b = 10 \)

  \( y = \# \text{pounds limit, } k = \# \text{item types allowed} \)

  \( v_1 = 1, w_1 = 2; \quad v_2 = 3, w_2 = 3; \)
  \( v_3 = 5, w_3 = 4; \quad v_4 = 9, w_4 = 7 \)

Recurrence:  \( F_k(y) = \max\{F_{k-1}(y), F_k(y - w_k) + v_k\} \)

\[
\begin{array}{cccccccccc}
  \text{Y} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
  \text{K} & \hline
  1 & 0 & 1 & 1 & 2 & 2 & 3 & 3 & 4 & 4 & 5 \\
  2 & 0 & 1 & 3 & 3 & 4 & 6 & 6 & 7 & 9 & 9 \\
  3 & 0 & 1 & 3 & 5 & 5 & 6 & 8 & 10 & 10 & 11 \\
  4 & 0 & 1 & 3 & 5 & 5 & 6 & 9 & 10 & 10 & 12 \\
\end{array}
\]

Note: Have left out the 0th row and the 0th column!
DP: Knapsack

- Note: $12 = \max(11, 9 + F_4(3)) = \max(11, 9 + 3) = 12$
- What is missing here? (Like in SP, we know the SP’s cost, but we don’t know SP itself…)
- So, we need another table:
  \[ i(k, j) = \text{maximum index such that item type } j \text{ is used in } F_k(y), \text{ i.e., } i(k, y) = j \rightarrow x_j \geq 1 \text{ and } x_q = 0 \quad \forall q > j \]
- B.C.’s: \[ i(1, y) = 0 \text{ if } F_1(y) = 0 \]
  \[ i(1, y) = 1 \text{ if } F_1(y) \neq 0 \]
- General:
  \[ i(k, y) = \begin{cases} 
    i(k-1, y) & \text{if } F_{k-1}(y) > F_k(y - w_k) + v_k \\
    k & \text{if } F_{k-1}(y) \leq F_k(y - w_k) + v_k
  \end{cases} \]
DP: Knapsack

- Trace Back: if $i(k,y) = q$, use item $q$ once, check $i(k,y-w(q))$.
- Example:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

- E.g. $F_4(10) = 12$. $i(4,10) = 4 \Rightarrow 4^{th}$ item is used once
DP: Knapsack

\[ i(4, 10 - w_4) = i(4,3) = 2 \rightarrow 2^{nd} \text{ item used once} \]
\[ i(4, 3 - w_2) = i(4,0) = 0 \rightarrow \text{done} \]

- Notice that \( i(4,8)=3 \)
  \[ \rightarrow \text{don’t use the most valuable item type} \]

- Many flavors of Knapsack Problem
- Example: Use as few stamps of denominations \( d_1, d_2, \ldots, d_n \) as possible to make up exactly \( C \) cents of postage
- Various change-making exercises in textbook
DP Examples

• Chapter 6 sections
  – Longest Increasing Subsequence
  – Minimum Edit Distance\(^\text{(Week 6 discussion)}\)
  – Knapsack\(^{\text{(Lecture)}}\)
  – Matrix Chain Product\(^{\text{(Lecture)}}\)
  – Shortest Paths (Floyd-Warshall)\(^{\text{(Lecture)}}\)
  – Independent Set in Trees

• Other
  – Rental Car / Canoe Problem
  – Longest Common Subsequence\(^{\text{(PA3 #3)}}\)
  – Optimum Polygon Triangulation
  – Transitive Closure
  – Change-Making\(^{\text{(PA3 #4)}}\)
  – Maximum Consecutive Subsequence
  – Optimal Parenthesization\(^{\text{(Week 6 discussion)}}\)
All-Pairs Shortest Paths

- **Input:** Directed graph $G = (V,E)$, weight $E \rightarrow \mathbb{R}$
- **Goal:** Create $n \times n$ matrix of SP distances $\delta(u,v)$
- Running Bellman-Ford once from each vertex
  \[ O(|V| \times |V| \times |E|) = O(|V|^4) \text{ for dense graphs} \]
All-Pairs Shortest Paths

• **Input:** Directed graph $G = (V,E)$, weight $E \rightarrow \mathbb{R}$

• **Goal:** Create $n \times n$ matrix of SP distances $\delta(u,v)$

• Running Bellman-Ford once from each vertex
  \[ O(|V| \times |V| \times |E|) = O(|V|^4) \] for dense graphs

• Assume **Input** = adjacency matrix
  
  – $n \times n$ matrix $W = (w_{ij})$ of edge weights
  
  – assume $w_{ii} = 0 \quad \forall i$,

  SP to self has no edges, as long as there are no negative cycles
Simple APSP Dynamic Programming

• “Relaxation” or “Induction” idea: \( d_{ij}^{(m)} = \) weight of SP from \( i \) to \( j \) with \( \leq m \) edges (\( \rightarrow \) relax \( m \) from 0 to \( n-1 \))

Base Case: \( d_{ij}^{(0)} = 0 \) if \( i = j \) and \( d_{ij}^{(0)} = \infty \) if \( i \neq j \)

Recurrence: \( d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + w_{kj} \} \)

• Runtime = \( O(n^4) \)
  \( n-1 \) passes, each computing \( n^2 \) \( d_{ij} \)'s in \( O(n) \) time
DP for APSP: Floyd-Warshall Algorithm

- Also DP, but faster $\Rightarrow O(n^3)$
- $c_{ij}^{(m)} =$ weight of SP from $i$ to $j$ with intermediate vertices in the set $\{1, 2, \ldots, m\} \Rightarrow \delta(i, j) = c_{ij}^{(n)}$
- DP: compute $c_{ij}^{(n)}$ in terms of smaller $c_{ij}^{(n-1)}$
- Base Case:
- Recurrence:

![Diagram](image.png)
DP for APSP: Floyd-Warshall Algorithm

- Also DP, but faster $\rightarrow O(n^3)$
- $c_{ij}^{(m)} = \text{weight of SP from } i \text{ to } j \text{ with intermediate vertices in the set } \{1, 2, ..., m\} \Rightarrow \delta(i, j) = c_{ij}^{(n)}$
- DP: compute $c_{ij}^{(n)}$ in terms of smaller $c_{ij}^{(n-1)}$
- **Base Case:** $c_{ij}^{(0)} = w_{ij}$
- **Recurrence:** $c_{ij}^{(m)} = \min \{c_{ij}^{(m-1)}, c_{im}^{(m-1)} + c_{mj}^{(m-1)} \}$
Floyd-Warshall Algorithm

for \( m=1..n \) do
  for \( i=1..n \) do
    for \( j = 1..n \) do
      \( c_{ij}^{(m)} = \min \{ c_{ij}^{(m-1)}, c_{im}^{(m-1)} + c_{mj}^{(m-1)} \} \)

• Runtime \( O(n^3) \)
Floyd-Warshall Algorithm

- Difference from previous Simple APSP DP: we do not check *all* possible intermediate vertices.
- \[
\text{for } m=1..n \text{ do for } i=1..n \text{ do for } j = 1..n \text{ do }
\]
  \[
c_{ij}^{(m)} = \min \{ c_{ij}^{(m-1)} , c_{im}^{(m-1)} + c_{mj}^{(m-1)} \}
\]
- Runtime \(O(n^3)\)
- Application: Transitive Closure \(G^*\) of graph \(G\):
  - \((i,j) \in G^* \text{ iff } \exists \text{ path from } i \text{ to } j \text{ in } G\)
  - Adjacency matrix, elements on \(\{0,1\}\)
  - Floyd-Warshall with \(\text{“ min ” } \rightarrow \text{“ OR” , “+” } \rightarrow \text{“ AND ”}\)
  - Runtime \(O(n^3)\)
  - Useful in many problems
Matrix-chain multiplication problem: Give a chain of \( n \) matrices \( \langle A_1, A_2, \ldots, A_n \rangle \) to be multiplied, how to get the product \( A_1 A_2 \ldots A_n \) with minimum number of scalar multiplications.

Associativity of matrix multiplication \( \Rightarrow \) many possible orderings to calculate a given matrix chain product:

- Only one way to multiply \( A_1 \times A_2 \)
- Best way for triple: Cost \( (A_1, A_2) + \text{Cost}((A_1 A_2) \times A_3) \) 
  or Cost \( (A_2, A_3) + \text{Cost}(A_1 (A_2 \times A_3)) \).
Example with 4 matrices: \( d = (13, 5, 89, 3, 34) \)

Best order of multiplication: 2856 multiplies

Worst order of multiplication: 54201 multiplies

→ Meaningful problem!
• Problem instance \(=\) chain of matrices

• What is a subproblem?

• What is the corresponding subsolution?
DP: Matrix Chain Product

• How do we build bottom-up?
  1) From last example:
    • Best way for triple: $\text{Cost } (A_1, A_2) + \text{Cost}((A_1 A_2) \times A_3)$
      or $\text{Cost } (A_2, A_3) + \text{Cost}(A_1 (A_2 \times A_3))$.
    • Save the best solutions for contiguous groups of $A_i$.
  2) Cost of $(i \times j)(j \times k)$ is $ijk$
    E.g.,
    
    $\begin{array}{c}
    3 & 3 \\
    \hline
    5 & 10 \\
    \hline
    3 & 10
    \end{array}
    = \begin{array}{c} 3 \\
    10
    \end{array}
    $
    
    Each of $3 \times 10$ entries requires 5 multiplies (+ 4 adds)
**DP: Matrix Chain Product**

- Cost of *final* multiplication?
  \[ A_1 \cdot A_2 \cdot A_3 \cdot \ldots \cdot A_{k-1} \cdot A_k \cdot \ldots \cdot A_n. \]

\[
\begin{align*}
&d_1 \times d_k \\
&d_k \times d_{n+1}
\end{align*}
\]

- Find the best splitting point \( k \) over all possibilities
- Each subproblem has been already solved optimally – we just need to look up in a table
DP: Matrix Chain Product

• FORMULATION:
  – Table entries $a_{ij}$, $1 \leq i \leq j \leq n$, where $a_{ij} =$ optimal solution (i.e., minimum # of multiplications) for $A_i \cdot A_{i+1} \cdot \ldots \cdot A_{j-1} \cdot A_j$.
  – DP will fill up the table of $a_{ij}$ values.
  – Matrix dimensions are given by vector of $d_i$ values, $1 \leq i \leq n+1$, i.e., matrix $A_i$ has dimensions $d_i \times d_{i+1}$.
DP: Matrix Chain Product

• **Build Table:**

  Diagonal $S$ contains all $a_{ij}$ with $j - i = S$.

  - $S = 0$: $a_{ij} = 0, \ i = 1, 2, \ldots, n$
  - $S = 1$: $a_{i, i+1} = d_i d_{i+1} d_{i+2}, \ i = 1, 2, \ldots, n-1$
  - $1 < S < n$: $a_{i, i+s} = \min_{i \leq k \leq i+s} (a_{i, k} + a_{k+1, i+s} + d_i d_k d_{i+s})$

• **Example:** 4 matrices, $d = (13, 5, 89, 3, 34)$

  - $S = 1$: $a_{12} = 5785$ ($= 13 \times 5 \times 89$)
  - $a_{23} = 1335$ ($= 5 \times 89 \times 3$)
  - $a_{34} = 9078$ ($= 89 \times 3 \times 34$)
DP: Matrix Chain Product

\( S = 2: \ a_{13} = \min(a_{11} + a_{23} + 13 \cdot 5 \cdot 3, \ a_{12} + a_{33} + 13 \cdot 89 \cdot 3) = 1530 \)
\( a_{24} = \min(a_{22} + a_{34} + 5 \cdot 89 \cdot 34, \ a_{23} + a_{44} + 5 \cdot 3 \cdot 34) = 1845 \)

\( S = 3: \ a_{14} = \min(\{k=1\} a_{11} + a_{24} + 13 \cdot 5 \cdot 34, \ \{k=2\} a_{12} + a_{34} + 13 \cdot 89 \cdot 34, \ \{k=3\} a_{13} + a_{44} + 13 \cdot 3 \cdot 34) = 2856 \)

(Note: max cost is 54201 multiplies!)

- **Complexity:** For \( S>0 \), choose among \( S \) choices for each of \( n-S \) elements in diagonal, so runtime is \( \Theta(n^3) \).

  Justification: \( \sum_{i=1}^{n} i(n-i) = \sum_{i=1}^{n} (ni - i^2) = n(n(n+1)/2) - (n(n+1)(2n+1)/6) = \Theta(n^3) \)
DP: Optimal Polygon Triangulation

• Polygon has *sides* and *vertices*
• Polygon is *simple* = not self-intersecting.
• Polygon P is *convex* if any line segment with ends in P lies entirely in P
• *Triangulation* of P is partition of P with chords into triangles.
• **Problem:** Given a convex polygon and weight function defined on triangles (e.g. the perimeter). Find triangulation of minimum weight (of minimum total length).
• # of triangles: Always have n-2 triangles with n-3 chords
DP: Optimal Polygon Triangulation

• What is a subproblem?

• How do we combine solutions to smaller subproblems into the solution to a given (larger) subproblem?
**DP: Optimal Polygon Triangulation**

- Optimal sub-triangulation of optimal triangulation

- Recurrent formula:
  
  \[
  t[i, j] = \min_{i \leq k \leq j-1} \{ t[i, k] + t[k + 1, j] + w(\Delta v[i - 1]v[k]v[j]) \}
  \]

- Runtime = \(O(n^3)\), space = \(O(n^2)\)
Maximum Consecutive Subsequence

- **Problem:** Given a sequence $x_1, x_2, \ldots, x_n$ of real numbers (not necessarily positive), find a subsequence $x_i, x_{i+1}, \ldots, x_j$ (of consecutive elements) such that the sum of the numbers in it is maximum over all subsequences of consecutive elements.

Example 1: -3, 5, 2, 9, -8, 40, -1

Example 2: -3, 5, 2, 9, -18, 40, -1
Maximum Consecutive Subsequence

• What are subproblems?

• What do I need to know “inductively”?
Complexities

- How many previously-calculated subsolutions do you need to look at when calculating a new subsolution?

  - Fibonacci ($F_i$):
  - Bellman-Ford SSSP ($d_{j}^{(k)}$):
  - Knapsack ($k,y$):
  - Longest Common Subsequence ($i,j$):
  - String Reconstruction ($i$):
  - Matrix Chain Product ($i,i+s$):
  - Naïve APSP (grow #edges) ($d_{ij}^{(m)}$):
  - Floyd-Warshall APSP (grow set of allowed v’s) ($d_{ij}^{(m)}$):
Reconstructing Solutions

• Knapsack: auxiliary array of “highest-index food type used”
  • Trace Back: if \( i(k,y) = q \), use item \( q \) once, check \( i(k,y-w(q)) \).
  • Example:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

  • E.g. \( F_4(10) = 12 \). \( i(4,10) = 4 \rightarrow 4^{th} \) item is used once

• Shortest Paths: auxiliary array of “prev” information

• Parenthesizing (MCP, Polygon Triangulation, …): auxiliary array of “split point” information
More (from discussion, Lecture 10 slides)
DP: Minimum Edit Distance

- Minimum edit distance (Section 6.3)
  - Given two strings $x[1..m]$ and $y[1..n]$, what is the minimum number of edit operations (insert, delete, replace) needed to transform one string into the other?
  - Example: $PINE \rightarrow TREE$: $PINE$-$TINE$-$TRNE$-$TREE$
  - Example: $BLACK \rightarrow CAT$: $BLACK$-$LACK$-$LAC$-$CAC$-$CAT$

- Charles L. Dodgson (= Lewis Carroll) invented parlor game of Doublets (shortest path in graph of words)
  - $PINE \rightarrow TREE$, $BLACK \rightarrow WHITE$, etc.
  - See Knuth, *The Stanford GraphBase*

- **DP Recurrence**
  \[
  E[i,j] = \min\{ 1 + E[i-1,j], 1 + E[i,j-1], \text{diff}(i,j) + E[i-1,j-1] \}
  \]
  B.C.s: $E[i,0] = i$ for $i = 0..m$; $E[0,j] = j$ for $j = 0..n$

  (E.g., it takes $j$ edit operations to transform the first zero characters of $X$ into the first $j$ characters of $Y$)
DP: Minimum Edit Distance

- $E[i,j] = \min\{1 + E[i-1,j], 1 + E[i,j-1], \text{diff}(i,j) + E[i-1,j-1]\}$
  - $1 + E[i-1,j]$ : edit distance of first $i-1$ characters of $X$ to first $j$ characters of $Y$, then add the last character of $X$
  - $1 + E[i,j-1]$ : edit distance of first $j-1$ characters of $Y$ to first $i$ characters of $X$, then add the last character of $Y$
  - $\text{diff}(i,j) + E[i-1,j-1]$ : edit distance of first $i-1$ characters of $X$ to first $j-1$ characters of $Y$, then add the edit distance between the last character of $X$ and the last character of $Y$

- B.C.s: $E[i,0] = i$ for $i = 0..m$; $E[0,j] = j$ for $j = 0..n$
## DP Table for Minimum Edit Distance

<table>
<thead>
<tr>
<th>y</th>
<th>T</th>
<th>R</th>
<th>E</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>P</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>N</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

- Red numbers show base cases
- Red arrows show which term is minimum in the recurrence
  
  - E.g., $E[2,3] = 1 + E[2,2]$
• **Situation:** You need to drive along a highway using rental cars. There are \( n \) rental car agencies 1, 2, \ldots, \( n \) along the highway. At any of the agencies, you can rent a car that can be returned at any other agency down the road. You cannot backtrack, i.e., you can drive only in one direction along the highway. So, a car rented at agency \( i \) can be returned only at some agency \( j > i \). For each pair of agencies \( i,j \) with \( j > i \), the cost of the \( i \)-to-\( j \) car rental is known.

• **Problem:** What is the minimum-cost sequence of car rentals that gets you from 1 to \( n \) ?
Even More …
Longest Common Substring (DPV 6.8)

6.8. Given two strings $x = x_1 x_2 \cdots x_n$ and $y = y_1 y_2 \cdots y_m$, we wish to find the length of their longest common substring, that is, the largest $k$ for which there are indices $i$ and $j$ with $x_i x_{i+1} \cdots x_{i+k-1} = y_j y_{j+1} \cdots y_{j+k-1}$. Show how to do this in time $O(mn)$.

- Subproblem definition

- Goal

- Recurrence
Longest Palindromic Subsequence (DPV 6.7)

6.7. A subsequence is *palindromic* if it is the same whether read left to right or right to left. For instance, the sequence


has many palindromic subsequences, including \( A, C, G, C, A \) and \( A, A, A, A \) (on the other hand, the subsequence \( A, C, T \) is not palindromic). Devise an algorithm that takes a sequence \( x[1 \ldots n] \) and returns the (length of the) longest palindromic subsequence. Its running time should be \( O(n^2) \).

- **Subproblem definition**
  - **Goal**
- **Recurrence**
Min-Edit “Palindromization”

• Given a string, find an efficient algorithm that computes the minimum number of letters that must be inserted to make the string into a palindrome.
  
  loyal → loayaol  (2 insertions)
  aaaabbb → bbbaaaabbb  (3 insertions)

• Subproblem
  
  – Goal

• Recurrence
All-Pairs Shortest Paths

What are the two “successive approximations” that we have seen for shortest paths?

(Dijkstra)

(Bellman-Ford)
Transitive Closure $G^*$ of Directed Graph $G$

- $(i,j) \in G^*$ iff $\exists$ path from $i$ to $j$ in $G$
- Input = adjacency matrix, elements $\in \{0,1\}$
Transitive Closure $G^*$ of Directed Graph $G$

- $(i,j) \in G^*$ iff $\exists$ path from $i$ to $j$ in $G$
- Input = adjacency matrix, elements $\in \{0,1\}$
Transitive Closure $G^*$ of Directed Graph $G$

- $(i,j) \in G^*$ iff $\exists$ path from $i$ to $j$ in $G$
- Input = adjacency matrix, elements $\in \{0,1\}$

for $m=1..n$ do
  for $i=1..n$ do
    for $j = 1..n$ do
      $c_{ij}^{(m)} = \text{OR} \{c_{ij}^{(m-1)}, [c_{im}^{(m-1)} \text{ AND } c_{mj}^{(m-1)}] \}$