String Reconstruction  (DPV Exercise 6.4)

• Given corrupted document with no punctuation, can you reconstruct it as a sequence of valid words (with aid of a dictionary) ?
  – E.g., “tomatomiciced” = tom | atom | iced (Yes)
    “catchaserat” = cat | chase | rat (Yes)
    “afarfarbettething” = ? (No)
String Reconstruction (DPV Exercise 6.4)

- Given corrupted document with no punctuation, can you reconstruct it as a sequence of valid words (with aid of a dictionary) ?
  - E.g., “tomatomiced” = tom | atom | iced
    “catchaserat” = cat | chase | rat
    “afarfarbettething” = ?

- Input:
  - corrupted document $x[1..n]$ (array of characters)
  - dictionary function $\text{dict}(w)$ (returns true if $w$ is a valid word)

- Brute-force?
- Greed?
Dynamic Programming Basic Approach

• Formulate problem recursively using subproblems
  – How do we obtain our desired solution from the solutions of subproblems?
    “Principle of Optimality” should hold

• Unlike Divide and Conquer, subproblems may overlap
  – As long as there aren’t too many subproblems, we’re okay

• Solve subproblems in an efficient order
  – “Bottom-up” (not recursively)
Formulate Recursively Using Subproblems

- Car Rentals on cross-country trip

Start

- SD
- SF
- PHX
- AUS
- ATL
- MEM
- NYC
- BOS

End

Prefixes (O(n))

Intervals (O(n^2))
Formulate Recursively Using Subproblems

- **Knapsack / Coin-Changing**
  - Pack knapsack (limit = B) with greatest value
  - Make change (amount = B) with fewest coins

\[ \frac{V_i}{W_i} \quad \text{for rice, corn, beans, ...} \]

\[ \text{with } 1\text{¢, 5¢, 10¢, 25¢, 50¢, ...} \]

\[ \$0.97 \]
Formulate Recursively Using Subproblems

- **Knapsack / Coin-Changing**
  - Pack knapsack (limit = B) with greatest value
  - Make change (amount = B) with fewest coins

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>.....</th>
<th>10</th>
<th>.....</th>
<th>.....</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,5,10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Note: relax = “un-constrain” item types, +12¢, denominations +25¢, +50¢, 100¢*
Formulate Recursively Using Subproblems

- Single-source shortest paths (Bellman-Ford)

SP cost from $v_0$

$\leq k-1$ edges

$\leq k$ edges

get to vertex $v_j$ from source using at most $k$ edges

$\leq n-1$ edges

$\ell^{(k)}_j = \min \left( \ldots, \min \ldots \right)$
Formulate Recursively Using Subproblems

- All-Pairs Shortest Paths

\[ \text{relaxation: which vertices are allowed on the } i-j \text{ path?} \]

\[ \text{dist}(i,j,k) = \min (\text{dist}(i,j,k-1) + \text{dist}(k,j,k-1)) \]

\[ k = \text{index of allowed vertices} \]
Formulate Recursively Using Subproblems

- **GridSum (PA3)** Given $n \times n$ matrix of integers, what is sum of entries in a specified submatrix?

  PA3 asks for $O(n^2)$ preprocessing (DP) and $O(1)$ query processing

- If you know all of the sums of rectangles with one corner at $(0,0)$, does that help?
Formulate Recursively Using Subproblems

• GridSum (PA3)

-5 10 12 1 -18
55 -80 3 -11 41
11 4 -15 0 99
-82 12 5 33 -45

• How efficiently can you find the sums of all rectangles that have one corner at (0,0) ? (DP!)
What’s a Good Subproblem Definition for String Reconstruction?

- Subproblem:

  \[ \text{first } k \text{ characters ("prefix")} \]
What’s a Good Subproblem Definition for String Reconstruction?

- Subproblem:

  \[ T(k) = \begin{cases} 
  \text{true} & \text{if } x[1..k] \text{ is a valid sequence of words} \\
  \text{false} & \text{otherwise} 
  \end{cases} \]

- \( T(0) = \text{true} \)

- We want \( T(n) \)
Express the Solution Recursively…

\[ T(k) = \text{true} \text{ iff there is some } 1 \leq j \leq k \text{ such that} \]
  \- \ T(j-1) \text{ is true}
  \- \ X[j..k] \text{ is a valid word}

• Define \( T(0) = \text{true} \)

• Given \( T(0), T(1), \ldots, T(k-1) \), how do we obtain \( T(k) \)?

… But Don’t Solve Recursively!

Recall Pascal’s Triangle, Fibonacci: \textit{tabulate} to avoid redundant calculations!
Solve Subproblems in an Efficient Order

- In what order should we compute the T(j)’s?
  - T(0), T(1), T(2), ...

- **Algorithm**
  
  ```
  T(0) = true
  For k = 1 to n
    T(k) = false
    for j = 1 to k
      if T(j-1) && dict(x[j..k])
        T(k) = true
        D(k) = j
  ```

- Running time: \( O(n^2) \)
- To reconstruct the document, use auxiliary array D[1..n]
  - If T(k) = true then D(k) = starting position of the word that ends at x[k]
    → reconstruct text by following backpointers
DP is a Promising Approach if You See:

• **Optimal substructure:** A problem exhibits optimal substructure if an optimal solution to the problem contains *within it* optimal solutions to subproblems. Whenever a problem exhibits optimal substructure, it is a good clue that DP might apply. (An alternative: greed.) (In Shortest Paths discussion, we called this the "Principle of Optimality")

• **Overlapping subproblems:** A recursive algorithm for the problem would end up solving the same subproblems over and over, rather than always generating new subproblems.
DQ vs. DP

- In both cases, start by formulating the problem recursively, in terms of subproblems

**DQ**
- Problem of size n can be decomposed into a few subproblems that are significantly smaller (e.g., n/2, 3n/4, etc.)
- Size of subproblems decreases geometrically
- Use a recursive algorithm

**DP**
- Problem of size n can be expressed in terms of subproblems that are not much smaller (e.g., n-1, n-2, etc.)
  - Recursive algorithm takes exponential time
  - But only polynomially many subproblems in total
- Avoid recursion, and solve subproblems one by one, saving answers in a table
DP Examples

- Chapter 6 sections
  - Longest Increasing Subsequence
  - Minimum Edit Distance (Week 6 discussion)
  - Knapsack (Lecture, PA3)
  - Matrix Chain Product (Lecture)
  - Shortest Paths (Floyd-Warshall) (Lecture)
  - Independent Set in Trees

- Other
  - Rental Car / Canoe Problem
  - Longest Common Subsequence (PA3)
  - Optimum Polygon Triangulation
  - Transitive Closure
  - Change-Making
  - Maximum Consecutive Subsequence
  - Optimal Parenthesization (Week 6 discussion)
DP: Longest Common Subsequence (PA3)
(Treatment from Cormen et al., Chapter 15)

• Problem: Given $x[1..m]$ and $y[1..n]$, find LCS

  $x$: A B C B D A B

  $y$: B D C A B A

  ⇒ B C B 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($x,y$)?
Recurrent Formula for LCS

- Let $c[i, j] =$ length of LCS of $X[i]=x[1..i], Y[j]=y[1..j]$
- Then $c[m, n] =$ length of LCS of $x$ and $y$
- Claim:
  - If $x[i] = y[j]$,
    $$c[i, j] = c[i-1, j-1] + 1$$
  - Otherwise,
    $$c[i, j] = \max(c[i, j-1], c[i-1, j])$$
- Proof: $x[i] = y[j] \implies \text{LCS}([X[i], Y[j]]) = \text{LCS}(X[i-1], Y[j-1]) + x[i]$
Relevant Properties That Motivate DP

• Any part of the optimal answer is also optimal
  – A subsequence of LCS(X,Y) is the LCS for some subsequences of X and Y

• Subproblems overlap
  – LCS(X[m],Y[n-1]) and LCS(X[m-1],Y[n]) have common subproblem LCS(X[m-1],Y[n-1])
  – There are polynomially few subproblems == O(mn) in total for the LCS problem
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: 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>x</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DP Table for LCS

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>B</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>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The table represents the dynamic programming approach for finding the longest common subsequence (LCS) of two given sequences.
<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>B</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>
<tr>
<td></td>
<td>y</td>
<td>B</td>
<td>D</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<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>1</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>2</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>
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 $T[m,n]$
**DP: Knapsack (Section 6.4, PA3)**

- **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 $F_k(y) = \max_{0 \leq k \leq n} \left( \sum_{j=1}^{k} w_j x_j, \sum_{j=1}^{k} v_j x_j \right)$ (with $0 \leq k \leq n$, $0 \leq y \leq b$)
  - $F_k(y)$ is the maximum value possible using only the first $k$ item types, when the weight limit is $y$. 

- \[ \sum_{j=1}^{k} w_j x_j \] total weight of items chosen
- \[ \sum_{j=1}^{k} v_j x_j \] total value of items chosen
DP: Knapsack

- B.C.’s:
  1. $F_0(y) = 0 \ \forall y$ when 0 item types are allowed
  2. $F_k(0) = 0 \ \forall k$ when weight limit is 0
  3. $F_1(y) = \lfloor y/w_1 \rfloor v_1$ 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:** \( k = 4, \ b = 10 \)
  
  \( y = \) #pounds limit, \( k = \) #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:** \( F_k(y) = \max\{F_{k-1}(y), F_k(y - w_k) + v_k\} \)

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

*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:
  
  $i(k, j) =$ maximum index such that item type $j$ is used in $F_k(y)$, i.e., $i(k, y) = j \rightarrow x_j \geq 1$ and $x_q = 0 \ \forall q > j$
- B.C.’s: $i(1, y) = 0$ if $F_1(y) = 0$
  
  $i(1, y) = 1$ 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>Y</td>
<td>k</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</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>3</td>
<td>4</td>
<td>3</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 \( i(4,8) = 3 \)
  \( \rightarrow \) 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 (= PA3)
- Various change-making exercises in textbook
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
  
  $O(|V| \times |V|^2|E|) = O(|V|^4)$ for dense graphs
- Adjacency-matrix representation of graph:
  - $n \times n$ 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
Similarity of APSP to Matrix Multiplication

• Similar: $C = A \cdot B$, with $A, B$ being two $n \times n$ matrices

Computing all $c_{ij} = \sum_k a_{ik} \cdot b_{kj}$ using $O(n^3)$ operations

• Replacing: 
  
  \[ + \] → \[ \text{\textbf{min}} \] 
  
  \[ \cdot \] → \[ + \]

  – gives $c_{ij} = \min_k \{a_{ik} + b_{kj}\}$
  
  – $D^{(m)} = D^{(m-1)} \times W$ (going from $\leq m-1$ edge paths to $\leq m$ edge paths)

  – Identity matrix is $D^{(0)}$

  – Cannot use Strassen’s because no “ - ” subtraction operator

• Time is still $O(n \cdot n^3) = O(n^4)$

• Speedup: Repeated squaring: $W^{2n} = W^n \times W^n$
  
  – Addition chains, just as we saw with Fibonacci in Lecture 1
  
  – Compute $W, W^2, W^4, \ldots, W^{2^k}$, $k = \log n \rightarrow O(n^3 \log n)$
DP APSP: Floyd-Warshall Algorithm

- Also DP, but faster (by another log n factor → O(n^3))
- \( c_{ij}^{(m)} \) = weight of SP from i to j 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

• Difference from previous: we do not check all possible intermediate vertices.
• 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 \( O(n^3) \)
• Example in Transitive Closure \( G^* \) of graph \( G \):
  – \( (i,j) \in G^* \) iff \( \exists \) path from \( i \) to \( j \) in \( G \)
  – Adjacency matrix, elements on \{0,1\}
  – Floyd-Warshall with “min” \( \rightarrow \) “OR”, “+” \( \rightarrow \) “AND”
  – Runtime \( O(n^3) \)
  – Useful in many problems
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>K</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>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

• Shortest Paths: auxiliary array of “prev” information

• Parenthesizing (MCP, Polygon Triangulation, ...): auxiliary array of “split point” information
**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: $\text{PINE} \rightarrow \text{TREE}$: $\text{PINE-TINE-TRNE-TREE}$
    - Example: $\text{BLACK} \rightarrow \text{CAT}$: $\text{BLACK-LACK-LAC-CAC-CAT}$

- Charles L. Dodgson (= Lewis Carroll) invented parlor game of Doublets (shortest path in graph of words)
  - $\text{PINE} \rightarrow \text{TREE}$, $\text{BLACK} \rightarrow \text{WHITE}$, 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></th>
<th>T</th>
<th>R</th>
<th>E</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>x</strong></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><strong>P</strong></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><strong>I</strong></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><strong>N</strong></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td><strong>E</strong></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., $E[2,3] = 1 + E[2,2]$