DEFINITION
Dynamic Programming is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems
Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS
Main idea:
• set up a recurrence relating a solution for a larger instance to solutions of some smaller instances
• solve smaller instances once
• record solutions in a table
• extract solution to the initial instance from that table

EXAMPLE 1: FIBONACCI NUMBERS
• Recall definition of Fibonacci numbers:
  – F(n) = F(n-1) + F(n-2)
  – F(0) = 0
  – F(1) = 1
• Computing the n\textsuperscript{th} Fibonacci number recursively (top-down):

Let C(n) = \# of additions to calculate F(n)
What is the efficiency of Fib?

Dynamic programming: build array Results from the bottom up

<table>
<thead>
<tr>
<th>Results</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>8</th>
<th>13</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</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>
</tr>
</tbody>
</table>
EXAMPLE 2: COIN-ROW PROBLEM

Puzzle:

- There is a row of n coins whose values are positive integers \( c_1, c_2, \ldots, c_n \), which are not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two adjacent coins in the row can be picked up.
- E.g.: 5, 1, 2, 10, 6, 2. What is the best selection?

Recurrence Relation

- Let \( F(n) \) be the maximum amount that can be picked up from the row of \( n \) coins. To derive a recurrence for \( F(n) \), we partition all the allowed coin selections into two groups:
  - those without last coin – the max amount is ?
  - those with last coin – the max amount is ?

- Thus we have the following recurrence
  - \( F(n) = \max\{c_n + F(n-2), \ F(n-1)\} \) for \( n > 1 \),
  - \( F(0) = 0, \ F(1)=c_1 \)

<table>
<thead>
<tr>
<th>Index i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coin c_i</td>
<td>--</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>F[i]</td>
<td>0</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Solution[i]</td>
<td>∅</td>
<td>{1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Algorithm:

\[
\begin{align*}
F[0] &= 0 \\
Solution[0] &= \{} \\
F[1] &= c_1 \\
Solution[1] &= \{1\} \\
\text{for i from 2 to n do} & \\
& \quad \text{addcoin} = c_i + F[i-2] \\
& \quad \text{if addcoin > F[i-1]} \\
& \quad \quad F[i] = \text{addcoin} \\
& \quad \quad Solution[i] = Solution[i-2] \cup \{i\} \\
& \quad \text{else} \\
& \quad \quad F[i] = F[i-1] \\
& \quad \quad Solution[i] = Solution[i-1]
\end{align*}
\]
EXAMPLE 3: COIN-COLLECTING PROBLEM

Problem

- Several coins are placed in cells of an $n \times m$ board.
- A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell.
- On each step, the robot can move either one cell to the right or one cell down from its current location. It tries moving horizontally first.

Approach

- Let $F(i,j)$ be the largest number of coins the robot can collect and bring to cell $(i,j)$ in the $i$th row and $j$th column.
- Let $P(i,j)$ be the last step that the robot took to collect the maximum coins to bring to cell $(i,j)$ in the $i$th row and $j$th column.

<table>
<thead>
<tr>
<th>F</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recurrence

Encode the distribution of coins on the board into a table $C$, where

\[
C(i,j) = \begin{cases} 
1 & \text{if there is a coin in cell } (i,j) \\
0 & \text{otherwise}
\end{cases}
\]

If the coins are coming from the left, $F(i,j) = F(i,j-1) + C(i,j)$
If the coins are coming from above, $F(i,j) = F(i-1,j) + C(i,j)$

Therefore

\[
F(i, j) = \max \{ F(i-1, j), F(i, j-1) \} + C(i,j)
\]

$F(0, j) = 0, F(i, 0) = 0$
for $1 \leq i \leq n, 1 \leq j \leq m$

Algorithm

//Fill first row: robot always comes from cell on the left
$F(1,1) = C(1,1) \quad P(1,1) = \text{nothing}$
for $j$ from 2 to $m$ do
    $F(1,j) = F(1,j-1) + C(1,j)$
    $P(1,j) = (1,j-1)$

//Fill other rows
for $i$ from 2 to $n$ do
    // In first column robot always comes from cell above
    $F(i,1) = F(i-1,1) + C(i,1)$
    $P(i,1) = (i-1,1)$
for $j$ from 2 to $m$ do
    if $F(i-1,j) > F(i,j-1)$
        $F(i,j) = F(i-1,j) + C(i,j)$
        $P(i,j) = (i-1,j)$
    else
        $F(i,j) = F(i,j-1) + C(i,j)$
        $P(i,j) = (i,j-1)$
EXAMPLE 4: KNAPSACK PROBLEM

Problem

- Knapsack of capacity \( W \)
- \( n \) items of weights \( w_1, w_2, \ldots w_n \), and values \( v_1, v_2, \ldots v_n \)
- All weights are positive integers.
- Find most valuable subset of items which fit into knapsack

Notation

- Let \( V(i,w) \) = value of the most valuable combination of the first \( i \) items that fit into knapsack of capacity \( w \)
- Let \( C(i,w) \) = contents of the knapsack that results in \( V(i,w) \)
- We want to find \( V(n,W) \) and \( C(n,W) \)

Recursive Analysis

To determine \( V(i,w) \), need to decide whether the \( i^{th} \) item, which weighs \( w_i \), should be inserted.

There are 3 cases:

- If \( w_i > w \) then it *cannot* be inserted: \( V(i,w) = V(i-1,w) \), and \( C(i,w) = C(i-1,w) \)
- Otherwise, decide whether it is worth adding it: compare
  - the best way of adding it: \( V(i-1,w-w_i) + v_i \)
  - with the value if it is not added \( V(i-1,w) \)

and pick the best

- If it is added, then \( C(i,w) = C(i-1,w-w_i) \cup \{w_i\} \)
- Otherwise, \( C(i,w) = C(i-1,w) \)

Recurrence

\[
V(i,j) = \begin{cases} 
V(i - 1,j) & \text{if } j - w_i < 0 \\
\max[V(i - 1,j), V(i - 1, w - w_i) + v_i] & \text{if } j - w_i \geq 0
\end{cases}
\]

Set initial conditions as:

\( V(i,0) = 0 \quad V(0,j) = 0 \), for \( i,j \geq 0 \)
Solution

\[
\begin{array}{ccccccc}
& 0 & \ldots & j-w_i & \ldots & j & \ldots & W \\
0 & 0 & \ldots & 0 & \ldots & 0 & 0 & \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \\
i-1 & 0 & V(i-1,j-w_i) & & V(i-1,j) & \\
i & 0 & & & V(i,j) & \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \\
n & 0 & & & & & \text{goal} & \\
\end{array}
\]

Memory Functions Method

- Comparison of top-down with bottom-up:
  - Bottom-up does not repeat work
  - Top-down guides calculation of only necessary sub-cases

- Memory Functions method:
  - Algorithm is top-down and recursive
  - When calculating a sub-case for the first time, save it in table
  - At each function call, check table to see if value calculated already

Memory Functions Algorithm

// Assume answers will be stored in array MFK[n,W]
// initialized to -1 except for row 0 and column 0 initialized with 0s

Function MFKnapsack(i, j)

if MFK[i, j] \geq 0 return MFK[i, j]
if w_i > j MFK[i, j] = MFKnapsack(i-1, j)
else MFK[i, j] = \max(MFKnapsack(i-1, j), MFKnapsack(i-1, j-w_i) + v_i)
return MFK[i, j]

Example:
Knapsack of capacity W=5, n=4.

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight (w_i)</th>
<th>Value (v_i)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>(w_i)</td>
<td>(v_i)</td>
<td>0</td>
<td>$0$</td>
<td>$0$</td>
<td>$0$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>$12$</td>
<td>1</td>
<td>$0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>$10$</td>
<td>2</td>
<td>$0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>$20$</td>
<td>3</td>
<td>$0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>$15$</td>
<td>4</td>
<td>$0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Calculate MFKnapsack(4,5)
EXAMPLE 5: WARSHALL’S ALGORITHM

Definitions: Transitive Closure

• Graph Definition: The **transitive closure** of a directed graph with n vertices is the graph which shows, for every vertex, all the vertices it can eventually reach.

• Adjacency Matrix Definition: The **transitive closure** of a directed graph with n vertices is the n \times n boolean matrix T in which element T(i,j) is 1 if there exists a non-trivial path (i.e. directed path of positive length) from the i^{th} vertex to the j^{th} vertex, and 0 otherwise.

Example

Brute Force Construction

• Algorithm:
  − For each vertex i in the graph:
    ▪ Perform a DFS or BFS traversal of the graph starting at vertex i
    ▪ For each vertex j encountered during that traversal, set T(i,j) = 1

• Efficiency: n vertices \times whole graph traversal starting at each vertex
  ▪ \( \in O(n^3) \) if adjacency matrices are used
  ▪ \( \in O(n \times (n+\text{edges})) \) if adjacency lists are used
Warshall’s Algorithm
- Construct transitive closure $T$ as the last matrix in the sequence of $n$-by-$n$ matrices $R^{(0)}$, … , $R^{(k)}$, … , $R^{(n)}$ where
- $R^{(k)}[i,j] = 1$ iff there is nontrivial path from $i$ to $j$ with only first $k$ vertices allowed as intermediate
- Note that $R^{(0)} = A$ (adjacency matrix), $R^{(n)} = T$ (transitive closure)

Example

![Diagram of a digraph with matrices $R^{(0)}$, $R^{(1)}$, $R^{(2)}$, $R^{(3)}$, and $R^{(4)}$. Each matrix reflects the existence of paths with increasing intermediate vertices, boxed row and column are used for getting the next matrix.]

$R^{(0)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$

$R^{(1)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$

$R^{(2)} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$

$R^{(3)} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$

$R^{(4)} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$

1’s reflect the existence of paths with no intermediate vertices (e.g., $R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

1’s reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex $a$ (note a new path from $d$ to $b$); boxed row and column are used for getting $R^{(2)}$.

1’s reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., $a$ and $b$ (note two new paths); boxed row and column are used for getting $R^{(3)}$.

1’s reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., $a$, $b$, and $c$ (no new paths); boxed row and column are used for getting $R^{(4)}$.

1’s reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., $a$, $b$, $c$, and $d$ (note five new paths).

FIGURE 8.13 Application of Warshall’s algorithm to the digraph shown. New 1’s are in bold.
Mechanism

\[ R^{(k-1)} = \begin{bmatrix} 1 & \boxed{k} \\ i & 0 \rightarrow 1 \end{bmatrix} \quad \implies \quad R^{(k)} = \begin{bmatrix} j & k \\ i & 1 & 1 \end{bmatrix} \]

**Figure 8.12** Rule for changing zeros in Warshall’s algorithm.

Recurrence

On the k-th iteration, the algorithm determines for every pair of vertices (i, j) if a path exists from i to j with just vertices 1,...,k allowed as intermediate.

\[
R^{(k)}[i,j] = \begin{cases} 
R^{(k-1)}[i,j] & \text{if a path using just 1,...,k-1 exists from } i \text{ to } j \text{ using just 1,...,k-1} \\
\text{or} \\
R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j] & \text{if paths } i \rightarrow k \text{ and } k \rightarrow j \text{ using just 1,...,k-1 exist from } i \text{ to } j \text{ using just 1,...,k-1} 
\end{cases}
\]

Matrix Generation

Recurrence implies the following rules for generating \( R^{(k)} \) from \( R^{(k-1)} \):

- **Rule 1** If an element in row i and column j is 1 in \( R^{(k-1)} \), it remains 1 in \( R^{(k)} \).
- **Rule 2** If an element in row i and column j is 0 in \( R^{(k-1)} \), it must be changed to 1 in \( R^{(k)} \) if and only if the element in its row i and column k and the element in its column j and row k are both 1’s in \( R^{(k-1)} \).

Pseudocode

**ALGORITHM Warshall(A[1..n, 1..n])**

//Implements Warshall’s algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
\[ R^{(0)} \leftarrow A \]
\[ \text{for } k \leftarrow 1 \text{ to } n \text{ do} \]
\[ \quad \text{for } i \leftarrow 1 \text{ to } n \text{ do} \]
\[ \quad \quad \text{for } j \leftarrow 1 \text{ to } n \text{ do} \]
\[ \quad \quad \quad R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j] \text{ or } (R^{(k-1)}[i, k] \text{ and } R^{(k-1)}[k, j]) \]
\[ \text{return } R^{(n)} \]

Analysis

- Time efficiency: \( \Theta(n^3) \)
- Space efficiency: Matrices can be written over their predecessors
EXAMPLE 6: FLOYD'S ALGORITHM

Problem
In a weighted (di)graph, find shortest paths between every pair of vertices

Example

\[
\begin{array}{c|cccc}
 & a & b & c & d \\
\hline
a &  &  &  &  \\
b &  &  &  &  \\
c &  &  &  &  \\
d &  &  &  &  \\
\end{array}
\]

Representation

Floyd's Algorithm
- Same idea as Marshall’s algorithm:
- construct solution through series of matrices \(D^{(0)}, \ldots, D^{(n)}\) using increasing subsets of the vertices allowed as intermediate
- At the \(k\)-th iteration, the algorithm determines shortest paths between every pair of vertices \(i, j\) that use only vertices among 1,\ldots,\(k\) as intermediate
- \(D^{(k)}[i,j] = \min \{D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}\)
**Example**

![Diagram of a digraph with vertices a, b, c, and d connected by edges with weights 2, 6, and 7.]

$$D^{(0)} = \begin{bmatrix} a & b & c & d \\ a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{bmatrix}$$

Lengths of the shortest paths with no intermediate vertices (\(D^{(0)}\) is simply the weight matrix).

$$D^{(1)} = \begin{bmatrix} a & b & c & d \\ a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & 5 & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \end{bmatrix}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just \(a\) (note two new shortest paths from \(b\) to \(c\) and from \(d\) to \(c\)).

$$D^{(2)} = \begin{bmatrix} a & b & c & d \\ a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & 5 & \infty \\ c & 9 & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \end{bmatrix}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., \(a\) and \(b\) (note a new shortest path from \(c\) to \(a\)).

$$D^{(3)} = \begin{bmatrix} a & b & c & d \\ a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 9 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{bmatrix}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., \(a\), \(b\), and \(c\) (note four new shortest paths from \(a\) to \(b\), from \(a\) to \(d\), from \(b\) to \(d\), and from \(d\) to \(b\)).

$$D^{(4)} = \begin{bmatrix} a & b & c & d \\ a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 7 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{bmatrix}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., \(a\), \(b\), \(c\), and \(d\) (note a new shortest path from \(c\) to \(a\)).

**FIGURE 8.16** Application of Floyd’s algorithm to the digraph shown. Updated elements are shown in bold.
Pseudocode

**ALGORITHM**  \textit{Floyd}(W[1..n, 1..n])

// Implements Floyd’s algorithm for the all-pairs shortest-paths problem  
// Input: The weight matrix \(W\) of a graph with no negative-length cycle  
// Output: The distance matrix of the shortest paths’ lengths  
\(D \leftarrow W \) // is not necessary if \(W\) can be overwritten

\textbf{for} \(k \leftarrow 1\) \textbf{to} \(n\) \textbf{do}

\textbf{for} \(i \leftarrow 1\) \textbf{to} \(n\) \textbf{do}

\textbf{for} \(j \leftarrow 1\) \textbf{to} \(n\) \textbf{do}

\(D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}\)

\textbf{return} \(D\)

Analysis

- Time efficiency: \(\Theta(n^3)\)
- Space efficiency: Matrices can be written over their predecessors

Actual path

![Graph diagram]