Chapter 8

Dynamic Programming
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.

- “Programming” here means “planning”.

- Main idea:
  - set up a recurrence relating a solution to 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: Fibonacci numbers

• Recall definition of Fibonacci numbers:

\[ F(n) = F(n-1) + F(n-2) \]
\[ F(0) = 0 \]
\[ F(1) = 1 \]

• Computing the \( n \)th Fibonacci number recursively (top-down):

\[ F(n) \]
\[ F(n-1) \quad + \quad F(n-2) \]
\[ F(n-2) \quad + \quad F(n-3) \quad F(n-3) \quad + \quad F(n-4) \]
\[ \ldots \]
Example: Fibonacci numbers (cont.)

Computing the \( n \)th Fibonacci number using bottom-up iteration and recording results:

\[
\begin{align*}
F(0) &= 0 \\
F(1) &= 1 \\
F(2) &= F(1) + F(0) = 1
\end{align*}
\]

\[
\ldots
\]

\[
F(n-2) = \\
F(n-1) = \\
F(n) = F(n-1) + F(n-2)
\]

Efficiency:
- time
- space
Examples of DP algorithms

- Computing a binomial coefficient
- Warshall’s algorithm for transitive closure
- Floyd’s algorithm for all-pairs shortest paths
- Constructing an optimal binary search tree
- Some instances of difficult discrete optimization problems:
  - traveling salesman
  - knapsack
Computing a binomial coefficient by DP

Binomial coefficients are coefficients of the binomial formula:

\[(a + b)^n = C(n,0)a^n b^0 + \ldots + C(n,k)a^{n-k}b^k + \ldots + C(n,n)a^0 b^n\]

Recurrence: \[C(n,k) = C(n-1,k) + C(n-1,k-1) \text{ for } n > k > 0\]

\[C(n,0) = 1, \quad C(n,n) = 1 \text{ for } n \geq 0\]

Value of \(C(n,k)\) can be computed by filling a table:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>\ldots</th>
<th>k-1</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n-1</td>
<td>C(n-1,k-1)</td>
<td>C(n-1,k)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>C(n,k)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Computing $C(n, k)$: pseudocode and analysis

ALGORITHM Binomial($n, k$)

// Computes $C(n, k)$ by the dynamic programming algorithm
// Input: A pair of nonnegative integers $n \geq k \geq 0$
// Output: The value of $C(n, k)$
for $i \leftarrow 0$ to $n$ do
    for $j \leftarrow 0$ to min($i, k$) do
        if $j = 0$ or $j = i$
            $C[i, j] \leftarrow 1$
        else $C[i, j] \leftarrow C[i - 1, j - 1] + C[i - 1, j]$
    return $C[n, k]$

Time efficiency: $\Theta(nk)$

Space efficiency: $\Theta(nk)$
Knapsack Problem by DP

Given \( n \) items of

- integer weights: \( w_1 \ w_2 \ldots \ w_n \)
- values: \( v_1 \ v_2 \ldots \ v_n \)

a knapsack of integer capacity \( W \)

find most valuable subset of the items that fit into the knapsack

Consider instance defined by first \( i \) items and capacity \( j \) \((j \leq W)\).

Let \( V[i,j] \) be optimal value of such instance. Then

\[
\max \{V[i-1,j], \ v_i + V[i-1,j-w_i]\} \quad \text{if } j - w_i \geq 0
\]

\[
V[i,j] = \begin{cases} 
V[i-1,j] & \text{if } j - w_i < 0 
\end{cases}
\]

Initial conditions: \( V[0,j] = 0 \) and \( V[i,0] = 0 \)
Knapsack Problem by DP (example)

Example: Knapsack of capacity $W = 5$

<table>
<thead>
<tr>
<th>item</th>
<th>weight</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>$12</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>$10</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>$20</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>$15</td>
</tr>
</tbody>
</table>

Capacity $j$

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
0 & & & & & \\
\end{array}
\]

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

?
Warshall's Algorithm: Transitive Closure

- Computes the transitive closure of a relation
- Alternatively: existence of all nontrivial paths in a digraph
- Example of transitive closure:

```
  0  0  1  0
  1  0  0  1
  0  0  0  0
  0  1  0  0
```

```
  0  0  1  0
  1  1  1  1
  0  0  0  0
  1  1  1  1
```
Warshall’s Algorithm

Constructs transitive closure $T$ as the last matrix in the sequence of $n$-by-$n$ matrices $R^{(0)}$, \ldots, $R^{(k)}$, \ldots, $R^{(n)}$ 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).
Warshall’s Algorithm (recurrence)

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

$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path using just } 1, \ldots, k-1) \\ or & \\ R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j] & \text{(path from } i \text{ to } k \\ and from } k \text{ to } i \text{ using just } 1, \ldots, k-1) \end{cases}$$
Recurrence relating elements $R^{(k)}$ to elements of $R^{(k-1)}$ is:

$$R^{(k)}[i,j] = R^{(k-1)}[i,j] \text{ or } (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j])$$

It 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 has to 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 $R^{(k-1)}$. 
Warshall's Algorithm (example)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>R(0) =</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>R(1) =</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

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

\[
R^{(4)} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}
\]
Warshall’s Algorithm (pseudocode and analysis)

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)} ← A

for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            R^{(k)}[i,j] ← R^{(k-1)}[i,j] or (R^{(k-1)}[i,k] and R^{(k-1)}[k,j])

return R^{(n)}

Time efficiency: Θ(n^3)

Space efficiency: Matrices can be written over their predecessors
Floyd’s Algorithm: All pairs shortest paths

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

Same idea: construct solution through series of matrices $D^{(0)}$, $\ldots$, $D^{(n)}$ using increasing subsets of the vertices allowed as intermediate

Example:
Floyd’s Algorithm (matrix generation)

On 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.

$$D^{(k)}[i,j] = \min \{D^{(k-1)}[i,j], \ D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$$
Floyd’s Algorithm (example)

D^{(0)} =
\begin{bmatrix}
0 & \infty & 3 & \infty \\
2 & 0 & \infty & \infty \\
\infty & 7 & 0 & 1 \\
6 & \infty & \infty & 0
\end{bmatrix}

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

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

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

D^{(4)} =
\begin{bmatrix}
0 & 10 & 3 & 4 \\
2 & 0 & 5 & 6 \\
7 & 7 & 0 & 1 \\
6 & 16 & 9 & 0
\end{bmatrix}
Floyd’s Algorithm (pseudocode and analysis)

```
ALGORITHM 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 ← W // is not necessary if W can be overwritten
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
    return D
```

**Time efficiency:** $\Theta(n^3)$

**Space efficiency:** Matrices can be written over their predecessors

**Note:** Shortest paths themselves can be found, too
Problem: Given $n$ keys $a_1 < ... < a_n$ and probabilities $p_1 \leq ... \leq p_n$ searching for them, find a BST with a minimum average number of comparisons in successful search.

Since total number of BSTs with $n$ nodes is given by $C(2n,n)/(n+1)$, which grows exponentially, brute force is hopeless.

Example: What is an optimal BST for keys $A$, $B$, $C$, and $D$ with search probabilities 0.1, 0.2, 0.4, and 0.3, respectively?
DP for Optimal BST Problem

Let $C[i,j]$ be minimum average number of comparisons made in $T[i,j]$, optimal BST for keys $a_i < ... < a_j$, where $1 \leq i \leq j \leq n$. Consider optimal BST among all BSTs with some $a_k$ ($i \leq k \leq j$) as their root; $T[i,j]$ is the best among them.

$$C[i,j] = \min_{i \leq k \leq j} \{ p_k \cdot 1 + \sum_{s = i}^{k-1} p_s \text{ (level } a_s \text{ in } T[i,k-1] + 1) + \sum_{s = k+1}^{j} p_s \text{ (level } a_s \text{ in } T[k+1,j] + 1) \}$$
DP for Optimal BST Problem (cont.)

After simplifications, we obtain the recurrence for $C[i,j]$: 

$$C[i,j] = \min_{i \leq k \leq j} \{C[i,k-1] + C[k+1,j]\} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \leq i \leq j \leq n$$

$$C[i,i] = p_i \quad \text{for } 1 \leq i \leq j \leq n$$
The tables below are filled diagonal by diagonal: the left one is filled using the recurrence

\[ C[i,j] = \min_{i \leq k \leq j} \{C[i,k-1] + C[k+1,j]\} + \sum_{s=i}^{j} p_s, \quad C[i,i] = p_i; \]

the right one, for trees' roots, records \( k \)'s values giving the minima.
ALGORITHM OptimalBST(P[1..n])

// Finds an optimal binary search tree by dynamic programming
// Input: An array P[1..n] of search probabilities for a sorted list of n keys
// Output: Average number of comparisons in successful searches in the
// optimal BST and table R of subtrees’ roots in the optimal BST

for i ← 1 to n do
    C[i, i − 1] ← 0
    C[i, i] ← P[i]
    R[i, i] ← i
C[n + 1, n] ← 0

for d ← 1 to n − 1 do // diagonal count
    for i ← 1 to n − d do
        j ← i + d
        minval ← ∞
        for k ← i to j do
            if C[i, k − 1] + C[k + 1, j] < minval
                minval ← C[i, k − 1] + C[k + 1, j]; kmin ← k
            R[i, j] ← kmin
        sum ← P[i]; for s ← i + 1 to j do sum ← sum + P[s]
        C[i, j] ← minval + sum

return C[1, n], R
Analysis DP for Optimal BST Problem

Time efficiency: \( \Theta(n^3) \) but can be reduced to \( \Theta(n^2) \) by taking advantage of monotonicity of entries in the root table, i.e., \( R[i,j] \) is always in the range between \( R[i,j-1] \) and \( R[i+1,j] \)

Space efficiency: \( \Theta(n^2) \)

Method can be expended to include unsuccessful searches