Dynamic Programming (DP)

- Richard Bellman (1920 - 1984)
- Technique for solving optimization problems:
  - Scheduling
  - Bioinformatics (Smith-Waterman algorithm)
  - Packaging and inventory management
- Problem = set of interdependent subproblems
- Solves subproblems and uses the results to solve larger subproblems until the entire problem is solved.
- Similar to divide and conquer except that there may be interrelations across subproblems.
- Solution to a subproblem is a function of solutions to one or more subproblems at the preceding levels.
Example: Shortest-path problem

- **Problem**: Find the shortest path between a pair of vertices in an acyclic graph
- $G$ contains $n$ nodes $\{0, 1, \ldots, n - 1\}$ and has an edge $(i, j)$ only if $i < j$.
- Find the least-cost path between nodes 0 and $n - 1$
- $f(x)$ - cost of the least-cost path from 0 to $x$
- DP formulation:

$$f(x) = \begin{cases} 
0 & x = 0 \\
\min_{0 \leq j \leq x} \{ f(j) + c(j, x) \} & 1 \leq x \leq n - 1
\end{cases}$$
Example: Shortest-path problem

Find $f(4)$

$$f(4) = \min\{f(3) + c(3, 4), f(2) + c(2, 4)\}$$
Composition function

Cost of a solution composed of subproblems $x_1, x_2, \ldots, x_l$

$$r = g(f(x_1), f(x_2), \ldots, f(x_l))$$

- $g$ - composition function (depends on the problem)
- DP formulation if the optimal solution is determined by composing optimal solutions to subproblems and selecting min or max.
Example: Composition of subproblem solutions

\[
\begin{align*}
  f(x_1) & \quad r_1 = g(f(x_1), f(x_3)) \\
  f(x_2) & \\
  f(x_3) & \quad r_2 = g(f(x_4), f(x_5)) \\
  f(x_4) & \\
  f(x_5) & \\
  f(x_6) & \quad r_3 = g(f(x_2), f(x_6), f(x_7)) \\
  f(x_7) & \\
  f(x_8) & = \min\{r_1, r_2, r_3\}
\end{align*}
\]

Composition of solutions into a term

Minimization of terms
Functional equation

- Functional equation (optimization equation)
  - Recursive equation that represents the solution
  - Left side: an unknown quantity
  - Right side: a min or max expression.

- Monadic DP formulation: functional equation contains a single recursive term

- Polyadic DP formulation: functional equation contains multiple recursive terms
Dependencies between subproblems can be represented by a graph. If the graph is acyclic then the nodes can be organized into levels such that subproblems at a particular level depend only on subproblems at previous levels.

**Serial DP formulation:** subproblems at all levels depend only on the results at the immediately preceding levels.

**Nonserial DP formulation:** dependencies on other than immediately preceding levels.

Four categories of DP formulations (not an exhaustive classification!):
- Serial monadic
- Serial polyadic
- Nonserial monadic
- Nonserial polyadic

Difficult to develop generic parallel algorithms!
Problem: Find the shortest path from \( S \) to \( R \).
$v_i^l$ - $i$-th node at level $l$

$c_{i,j}^l$ - cost of the edge connecting $v_i^l$ to $v_{j+1}^l$

$C_i^l$ - cost of reaching $R$ from any node $v_i^l$

$C^l = [C_0^l, C_1^l, \ldots, C_{n-1}^l]^T$ - cost of reaching $R$ from level $l$ nodes

**Shortest-path problem**: compute $C^0$

The graph has only one starting node, thus $C^0 = [C_0^0]$

Any path from $v_i^l$ to $R$ includes a node $v_{j+1}^l$, $(0 \leq j \leq n - 1)$
The cost of the shortest path between $v_i^l$ and $R$:

$$C_i^l = \min\{(c_{i,j}^l + C_j^{l+1}) | j \text{ is a node at level } l + 1\}$$

At the last level: $C_j^{r-1} = c_{j,R}^{r-1}$

**Monadic**: only one recursive term in its right hand side

**Serial**: require solutions to subproblems at the immediate preceding level
The cost of reaching $R$ from any node at level $l$ ($0 \leq l \leq r - 1$):

$$C_0^l = \min\{ (c_{0,0}^l + C_0^{l+1}), (c_{0,1}^l + C_1^{l+1}), \ldots, (c_{0,n-1}^l + C_{n-1}^{l+1}) \}$$

$$C_1^l = \min\{ (c_{1,0}^l + C_0^{l+1}), (c_{1,1}^l + C_1^{l+1}), \ldots, (c_{1,n-1}^l + C_{n-1}^{l+1}) \}$$

$$\vdots$$

$$C_{n-1}^l = \min\{ (c_{n-1,0}^l + C_0^{l+1}), (c_{n-1,1}^l + C_1^{l+1}), \ldots, (c_{n-1,n-1}^l + C_{n-1}^{l+1}) \}$$
× is the matrix-vector operation where + is replaced by min and · is replaced by +.

\[ C^l = M_{l,l+1} \times C^{l+1} \]

\[ M_{l,l+1} = \begin{bmatrix}
    c^l_{0,0} & c^l_{0,1} & \cdots & c^l_{0,n-1} \\
    c^l_{1,0} & c^l_{1,1} & \cdots & c^l_{1,n-1} \\
    \vdots & \vdots & \ddots & \vdots \\
    c^l_{n-1,0} & c^l_{n-1,1} & \cdots & c^l_{n-1,n-1}
\end{bmatrix} \]
Sequential algorithm: a sequence of matrix-vector multiplications

1. Compute $C^{r-1}$
2. Compute $C^{r-k-1}$ for $k = 1, 2, \ldots, r - 2$
3. Compute $C^0$

Running time: $\Theta(rn^2)$

Parallel algorithm:

- Use the parallel matrix-vector multiplication algorithm
- Matrix vector multiplication on $\Theta(n)$ processors takes $\Theta(n)$

Parallel running time: $\Theta(rn)$

In case of sparse graphs use sparse matrix-vector multiplication algorithms
0/1 Knapsack Problem:

- Given a knapsack of capacity $c$ and a set of $n$ objects
- Object $i$ has weight $w_i$ and profit $p_i$
- Find a set of objects to put in the knapsack such that:

$$\sum_{i=1}^{n} w_i v_i \leq c$$

and

$$\sum_{i=1}^{n} p_i v_i$$

is maximized.

- $v_i = 1$ if object $i$ is in the knapsack, and $v_i = 0$ otherwise.
Serial Monadic DP: The 0/1 Knapsack Problem

- $F[i, x]$ - maximum profit for a knapsack of capacity $x$ using only objects $\{1, 2, \ldots, i\}$.
- $F[c, n]$ is the solution.
- DP formulation:

$$ F[i, x] = \begin{cases} 0 & x \geq 0, i = 0 \\ -\infty & x \leq 0, i = 0 \\ \max\{F[i - 1, x], (F[i - 1, x - w_i] + p_i)\} & 1 \leq x \leq n \end{cases} $$

- Two choices:
  - Object not included, capacity and profit do not change
  - Object included, capacity becomes $x - w_i$, profit increases by $p_i$
Serial Monadic DP: The 0/1 Knapsack Problem

**Sequential algorithm:** construct the table in row-major order.
**Running time:** $\Theta(nc)$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1 - $w_i$</th>
<th>(j)</th>
<th>(c - 1)</th>
<th>(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(P_0)</td>
<td>(P_{j - w_i - 1})</td>
<td>(P_{j - 1})</td>
<td>(P_{c - 2})</td>
<td>(P_{c - 1})</td>
</tr>
</tbody>
</table>

**Table \(F\)**
CREW PRAM algorithm:

- $c$ processors: $P_0, \ldots, P_c$.
- $P_{r-1}$ computes the $r$-th column of matrix $F$.
- Computing $F[j, r]$ requires constant time.
- Parallel runtime: $\Theta(n)$.
- Cost: $\Theta(nc) \implies$ cost optimal.
Distributed memory machine:

- $c$ processors: $P_0, \ldots, P_c$.
- $F$ distributed column-wise.
- Computing $F[j, r]$ requires $F[j - 1, r - w_j]$ from another processor.
- Need to perform a $w_j$-circular shift, takes time $(t_s + t_w) \log c$.
- Parallel runtime: $\Theta(n \log c)$.
- Cost: $\Theta(nc \log c) \implies$ not-cost optimal.
Distributed memory machine with more columns per processor:

- $p$ processors each computing $\frac{c}{p}$ elements of the table in each iteration.
- Need to perform a circular shift; total message size of $\frac{c}{p}$
- Total time for each iteration: $2t_s + t_w \frac{c}{p} + t_c \frac{c}{p}$
- Parallel runtime: $\Theta(\frac{nc}{p})$.
- Cost: $\Theta(nc) \Rightarrow$ cost optimal.
LCS Problem:

- Given two sequences
  \[ A = \langle a_1, a_2, \ldots, a_n \rangle \]
  \[ B = \langle b_1, b_2, \ldots, b_m \rangle \]
- Find the longest sequence that is a subsequence of both \( A \) and \( B \).
- Example:
  \[ A = \langle c, a, d, b, r, z \rangle \]
  \[ B = \langle a, s, b, z \rangle \]
  \[ LCS = \langle a, b, z \rangle \]
Nonserial Monadic DP: LCS Problem

- \( F[i, j] \) - length of the longest common subsequence of the first \( i \) elements of \( A \) and the first \( j \) elements of \( B \).
- **Objective:** determine \( F[n, m] \).
- **DP formulation:**

\[
F[i, j] = \begin{cases} 
0 & i = 0 \text{ or } j = 0 \\
F[i - 1, j - 1] + 1 & i, j \geq 0 \text{ and } x_i = y_j \\
\max\{F[i, j - 1], F[i - 1, j]\} & i, j \geq 0 \text{ and } x_i \neq y_j
\end{cases}
\]

- **Sequential Algorithm:** compute \( F \) table in row-major order.
- **Running time:** \( \Theta(nm) \)
Nonserial: each node depends on two subproblems at the preceding level and one problem two levels earlier.

Monadic: equation has a single recursive term
### Computing LCS of two amino-acid sequences

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>E</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>W</th>
<th>G</th>
<th>H</th>
<th>E</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>0</td>
<td>0</td>
<td>0</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>W</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>H</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>A</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>4</td>
</tr>
<tr>
<td>E</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>4</td>
</tr>
</tbody>
</table>


**LCS:** A W H E E
CREW PRAM algorithm:

- For simplicity assume $m = n$
- CREW PRAM with $n$ processors
- $P_i$ computes the $i$-th column of matrix $F$.
- Entries computed in a diagonal sweep.
- Computing $F[i,j]$ requires constant time.
- Parallel runtime: $\Theta(n)$.
- Cost: $\Theta(n^2) \implies$ cost optimal.
Linear array algorithm:

- For simplicity assume $m = n$
- Linear array with $n$ processors
- $P_i$ computes the $i$-th column of matrix $F$.
- Entries computed in a diagonal sweep.
- Computing $F[i, j]$ requires constant time.
- Parallel runtime: $T_p = (2n - 1)(t_s + t_w + t_c)$.
- Efficiency: $E = \frac{n^2 t_c}{n(2n-1)(t_s+t_w+t_c)}$.
- Maximum efficiency: $E_{max} = \frac{1}{2-1/n}$, bounded above by 0.5
LCS: Linear array parallel formulation

\[ \begin{array}{cccccc}
0 & 1 & 2 & \cdots & m \\
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & & & & \\
2 & 0 & & & & \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
n & 0 & & & & \\
\end{array} \]

\[ P_0 & P_1 & \cdots & P_{n-1} \]
Serial Polyadic DP: Floyd’s Algorithm

- All-pairs shortest paths algorithm
- \( d_{i,j}^k \) - minimum cost of a path from \( i \) to \( j \) using only nodes \( v_0, v_1, \ldots, v_{k-1} \).
- DP formulation:

\[
d_{i,j}^k = \begin{cases} 
  c_{i,j} & k = 0 \\
  \min\{d_{i,j}^{k-1}, (d_{i,k}^{k-1} + d_{k,j}^{k-1})\} & 0 \leq k \leq n - 1
\end{cases}
\]

- Polyadic formulation: two recursive terms in the equation.
- Sequential algorithm running time: \( \Theta(n^3) \).
CREW PRAM algorithm:

- CREW PRAM with $n^2$ processors
- $P_{i,j}$ computes $d_{i,j}^k$, for $k = 1, 2, \ldots, n$.
- Computing $d_{i,j}^k$ requires constant time.
- Parallel runtime: $\Theta(n)$.
- Cost: $\Theta(n^3) \Rightarrow$ cost optimal.
Multplying \( n \) matrices, \( A_1, A_2, \ldots, A_n \)

\( A_i \) has \( r_{i-1} \) rows and \( r_i \) columns.

**Problem:** Determine a parenthesization that minimizes the number of operations.

**Example:** \( A_1: 10 \times 20; A_2: 20 \times 30; A_3: 30 \times 40; \)

\((A_1 \times A_2) \times A_3:\)

\( A_1 \times A_2 \) requires \( 10 \times 20 \times 30 \) operations

The result matrix is \( 10 \times 30 \).

Multiplying by \( A_3 \) requires additional \( 10 \times 30 \times 40 \) operations

Total number of operations: 18000

\( A_1 \times (A_2 \times A_3): \)

requires \( 20 \times 30 \times 40 + 10 \times 20 \times 40 = 32000 \) operations.
Nonserial Polyadic DP: Optimal Matrix-Parenthesization

- $C[i, j]$ - optimal cost of multiplying matrices $A_i, \ldots, A_j$
- Product of two smaller chains: $A_i, A_{i+1}, \ldots, A_k$ and $A_{k+1}, A_{k+2}, \ldots, A_j$
- First chain $\Rightarrow r_{i-1} \times r_k$ matrix;
  second chain $\Rightarrow r_k \times r_j$ matrix
- Cost of multiplying these two: $r_{i-1}r_kr_j$
- DP formulation:

$$C[i, j] = \begin{cases} 0 & j = i, 0 \leq i \leq n \\ \min_{i \leq k \leq j} \{C[i, k] + C[k + 1, j] + r_{i-1}r_kr_j\} & 1 \leq i \leq j \leq n - 1 \end{cases}$$

- Problem: Find $C[1, n]$
Nonserial Polyadic DP: Optimal Matrix-Parenthesization

C[1,1]  C[1,2]  C[1,3]  C[1,4]


C[3,3]  C[3,4]

C[4,4]
Sequential algorithm:

- Fills the table diagonally.
- Computing $C[i, j]$ requires evaluating $j - i$ terms and selecting their minimum.
- Each entry in diagonal $l$ can be computed in time $lt_c$.
- The algorithm computes $n - 1$ chains of length 2, taking time $(n - 1)t_c$
  $n - 2$ chains of length 3, taking time $(n - 2)2t_c$, and so on.
- $T_S = \sum_{i=1}^{n-1} (n - i)it_c \approx (n^3/6)t_c$
- $T_S = \Theta(n^3)$
Parallel formulation:

- Logical ring of \( n \) processors
- Step \( l \): each processor computes a single element belonging to the \( l \)-th diagonal.
- After computing its value a processor sends it to all others by all-to-all broadcast.
- Total time to compute the entries in diagonal \( l \):
  \[ t_{lC} + t_s \log n + t_w (n - 1) \]
- \( T_P = \sum_{l=1}^{n-1} (t_{lC} + t_s \log n + t_w (n - 1)) \)
  \[ = \frac{(n-1)n}{2} t_c + t_s (n - 1) \log n + t_w (n - 1)^2 \]
- \( T_P = \Theta(n^2) \)
- Cost: \( \Theta(n^3) \) \( \Rightarrow \) cost optimal.
Parallel formulation (using $p$ processors):

- Logical ring of $p$ processors; each processor storing $n/p$ nodes.
- After computing its corresponding $C[i,j]$ values a processor sends them to all others by all-to-all broadcast.
- Time required for all-to-all broadcast of $n/p$ words:
  $$t_s \log p + t_w n(p - 1)/p \approx t_s \log p + t_w n$$
- $$T_P = \sum_{l=1}^{n-1} (\frac{lt_c n}{p} + t_s \log p + t_w n)$$
  $$= \frac{(n-1)n^2}{2p} t_c + t_s (n - 1) \log p + t_w n(n - 1)$$
- $$T_P = \Theta(n^3/p) + \Theta(n^2)$$