Dynamic Programming
Dijkstra’s Algorithm Revisited

- Think of a graph as a kind of race track.
  - Runners are waiting to be tagged at each vertex.
  - Edge weights are running times.

- At time 0, runners take off from vertex $s$. 
Dijkstra’s Algorithm Revisited, cont’d

- At time 4, the runner reaches vertex $y$ and tags.
  - Runners take off from vertex $y$.
- One of the runners from vertex $y$ reaches vertex $t$ before the runner from vertex $s$.
  - The runner from vertex $s$ loses and leaves the race.
Dijkstra’s Algorithm Revisited, cont’d

- The runner from vertex $y$ reaches vertex $z$.
- The runner from vertex $t$ reaches vertex $x$ first.
- Now we have the shortest (fastest) path from vertex $s$ to each of the other vertices.
Greedy Algorithms

- Proceed in stages.
- At each stage, choose a **local optimum**.
  - Attempt to do what is best based on current information.
  - “Take what you can get now.”
- Hope this process leads to the **global optimum**.
  - Doesn’t always work.
Example Greedy Algorithms

- Dijkstra’s algorithm for shortest weighted path.
- Prim’s algorithm for minimum spanning tree.
- Kruskal’s algorithm for minimum spanning tree.
Divide and Conquer Algorithms

- **Divide** a problem into at least two smaller problems.
  - The subproblems can (but not necessarily) be solved recursively.

- **Conquer** by forming the solution to the original problem from the solutions to the smaller problems.
Example Divide and Conquer Algorithms

- Solution to the *Towers of Hanoi* puzzle.
- Mergesort
- Quicksort
Multiplying Two Large Integers

- We want to multiply two large \( N \)-digit integers \( X \) and \( Y \).

- Done the usual way, it’s \( \Theta(N^2) \).
Multiplying Two Large Integers

- Let $X = 61,438,521$ and $Y = 94,736,407$.

- **Divide and conquer:**
  Break each number into two parts.
  - $X_L = 6,143$ and $X_R = 8,521$
  - $Y_L = 9,473$ and $Y_R = 6,407$

- Therefore:
  - $X = X_L10^4 + X_R$
  - $Y = Y_L10^4 + Y_R$
  - $XY = X_LY_L10^8 + (X_LY_R + X_RY_L)10^4 + X_RY_R$
Multiplying Two Large Integers

\[ XY = X_L Y_L 10^8 + (X_L Y_R + X_R Y_L)10^4 + X_R Y_R \]

- Replace the multiplier of 10^4 with:
  \[ X_L Y_R + X_R Y_L = (X_L - X_R)(Y_R - Y_L) + X_L Y_L + X_R Y_R \]

- Since we’re already computing \(X_L Y_L\) and \(X_R Y_R\), we’ve reduced the number of multiplications by one.

- With these algebraic manipulations, computing \(XY\) is \(O(N^{\log 3}) = O(N^{1.59})\).
# Multiplying Two Large Integers

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
<th>Computational Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_L$</td>
<td>6,143</td>
<td>Given</td>
</tr>
<tr>
<td>$X_R$</td>
<td>8,521</td>
<td>Given</td>
</tr>
<tr>
<td>$Y_L$</td>
<td>9,473</td>
<td>Given</td>
</tr>
<tr>
<td>$Y_R$</td>
<td>6,407</td>
<td>Given</td>
</tr>
<tr>
<td>$D_1 = X_L - X_R$</td>
<td>−2,378</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>$D_2 = Y_R - Y_L$</td>
<td>−3,066</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>$X_L Y_L$</td>
<td>58,192,639</td>
<td>$T(N/2)$</td>
</tr>
<tr>
<td>$X_R Y_R$</td>
<td>54,594,047</td>
<td>$T(N/2)$</td>
</tr>
<tr>
<td>$D_1 D_2$</td>
<td>7,290,948</td>
<td>$T(N/2)$</td>
</tr>
<tr>
<td>$D_3 = D_1 D_2 + X_L Y_L + X_R Y_R$</td>
<td>120,077,634</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>$X_R Y_R$</td>
<td>54,594,047</td>
<td>Computed above</td>
</tr>
<tr>
<td>$D_3 10^4$</td>
<td>1,200,776,340,000</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>$X_L Y_L 10^8$</td>
<td>5,819,263,900,000,000</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>$X_L Y_L 10^8 + D_3 10^4 + X_R Y_R$</td>
<td>5,820,464,730,934,047</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>

**Figure 10.37** The divide-and-conquer algorithm in action
Dynamic Programming Algorithms

- Break a problem into smaller subproblems.
  - But we don’t know exactly which subproblems to solve.

- We solve them all and store the results in a table.
  - Use a table instead of recursion.

- Use the stored results to solve larger problems.

Richard Bellman coined the term dynamic programming in the 1950s when *programming* meant *planning*. *Dynamic programming* involved optimally planning multistage processes.
Example Dynamic Programming Algorithm

- Compute the **fibonacci series**.
- The table consists of the last two computed numbers.
- It’s extremely inefficient to use recursion to compute the series.
The Knapsack Problem

- A thief burglarizing a safe finds that it contains \( N \) types of items of various weights and values.

<table>
<thead>
<tr>
<th>Item ( i )</th>
<th>Weight ( w )</th>
<th>Value ( v )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>$30</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>$14</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>$16</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>$9</td>
</tr>
</tbody>
</table>

- The thief has a knapsack that can hold only \( W \) pounds.
  - The thief can take multiple items of each type.

The Knapsack Problem, *cont’d*

- What is the *optimum* (most valuable) haul that the thief can carry away in the knapsack?

- The solution will use *dynamic programming*.
  - In other words, it will use a table.

- What are the *subproblems*?
  - Consider *smaller knapsacks* that have a smaller capacities.
  - Solve for each knapsack size.
Let $K(w) = \text{maximum value for a knapsack with capacity weight } w$

Suppose the optimum solution to $K(w)$ includes item $I$

Then removing this item from the knapsack leaves an optimal solution for a smaller knapsack, $K(w - w_i)$

In other words, $K(w) = K(w - w_i) + v_i$ for some $i$.

We don’t know which $i$, so let’s try them all:

$$K(w) = \max_{i: w_i \leq w} \{K(w - w_i) + v_i\}$$
int K[] = new int[W+1];
K[0] = 0;

for (int ww = 1; ww <= W; ww++) {
    K[ww] = K[ww-1];
    int max = 0;
    int item = 0;
    for (int i = 1; i <= N; i++) {
        if (w[i] <= ww) {
            int value = K[ww - w[i]] + v[i];
            if (max < value) {
                max = value;
                item = i;
            }
        }
    }
    K[ww] = max;
}

// Table

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>$30</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>$14</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>$16</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>$9</td>
</tr>
</tbody>
</table>

// Computed Table

<table>
<thead>
<tr>
<th>w</th>
<th>item weight</th>
<th>value</th>
<th>K[w]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>6</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>6</td>
<td>30</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>6</td>
<td>30</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>6</td>
<td>48</td>
</tr>
</tbody>
</table>
The Knapsack Problem, *cont’d*

- Which items are in the knapsack?

<table>
<thead>
<tr>
<th>w</th>
<th>item</th>
<th>weight</th>
<th>value</th>
<th>K[w]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>9</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
<td>14</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>6</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>14</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>6</td>
<td>30</td>
<td>39</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>6</td>
<td>30</td>
<td>44</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>6</td>
<td>30</td>
<td>48</td>
</tr>
</tbody>
</table>

### Table: Value Calculations

<table>
<thead>
<tr>
<th>w</th>
<th>item</th>
<th>value</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>30</td>
<td>48 - 30 = 18</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
<td>9</td>
<td>18 - 9 = 9</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>9</td>
<td>9 - 9 = 0</td>
</tr>
</tbody>
</table>
Ordering Matrix Multiplications

- Suppose we want to multiply together four matrices:

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>50 x 20</td>
</tr>
<tr>
<td>B</td>
<td>20 x 1</td>
</tr>
<tr>
<td>C</td>
<td>1 x 10</td>
</tr>
<tr>
<td>D</td>
<td>10 x 100</td>
</tr>
</tbody>
</table>

- Matrix multiplication is associative.

What order should we multiply them to minimize the cost?

Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes approximately $mnp$ multiplications.

Ordering Matrix Multiplications, cont’d

Figure 6.6 \( A \times B \times C \times D = (A \times (B \times C)) \times D. \)

(a) \[
\begin{align*}
A & : 50 \times 20 \\
B & : 20 \times 1 \\
C & : 1 \times 10 \\
D & : 10 \times 100
\end{align*}
\]

(b) \[
\begin{align*}
A & : 50 \times 20 \\
B \times C & : 20 \times 10 \\
D & : 10 \times 100
\end{align*}
\]

(c) \[
\begin{align*}
A \times (B \times C) & : 50 \times 10 \\
D & : 10 \times 100
\end{align*}
\]

(d) \[
\begin{align*}
(A \times (B \times C)) \times D & : 50 \times 100
\end{align*}
\]

Ordering Matrix Multiplications, cont’d

- Use a **dynamic programming** algorithm to find the optimal (least cost) multiplication order.

- Represent each parenthesization as a binary expression tree:

```
Figure 6.7 (a) \((A \times B) \times C) \times D\); (b) \(A \times ((B \times C) \times D)\); (c) \((A \times (B \times C)) \times D\).
```

![Diagram of binary expression trees](https://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf)
For a tree to be optimal, its subtrees must also be optimal.

What are the subproblems?

Consider the multiplications represented by the subtrees.

Ordering Matrix Multiplications, cont’d

- Let $C(i, j) = \text{minimum cost of multiplying } A_i \times A_{i+1} \times ... \times A_j$
- The size of each subproblem is $|j - i|$ multiplications.
  - The smallest problem is $i = j$, so $C(i, i) = 0$.
  - For $j > i$, consider the optimal subtree for $C(i, j)$.
  - The uppermost branch splits the product into two parts, $A_i \times ... \times A_k$ and $A_{k+1} \times ... \times A_j$ for some $k$ between $i$ and $j$.
  - The cost of the subtree is the cost of the two partial products, plus the cost of combining them:

$$C(i, j) = \min_{i \leq k < j} \{C(i, k) + C(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j\}$$
for i = 1 to n {
    C[i,i] = 0;
}

// s is the subproblem size
for s = 1 to n-1 {
    for i = 1 to n-s {
        j = i+s;
        for k = i to j-1 {
            cost = C[i,k] + C[k+1,j] + m[i-1]*m[k]*m[j];
            if (cost < C[i,j]) {
                C[i,j] = cost;
                lastChange[i,j] = k;
            }
        }
    }
}

Record costs in the two-dimensional array C[][]

Compute \( C(i, j) = \min_{i \leq k < j} \{ C(i, k) + C(k+1, j) + m_{i-1} \cdot m_k \cdot m_j \} \)

Find the optimal value of \( k \)

Record changes in array lastChange[][]
Recover the optimal parenthesization from the `lastChange[][][]` array:

```java
void order(i, j)
{
    if (i == j) System.out.print(name[i]);
    else {
        System.out.print ("(");
        order(i, lastChange[i,j]-1);
        System.out.print ("*");
        order(lastChange[i,j], j);
        System.out.print (")");
    }
}
```
Common Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation. But there are a few standard choices that seem to arise repeatedly in dynamic programming.

i. The input is $x_1, x_2, \ldots, x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.

\[
\begin{array}{cccccccc}
  x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} \\
\end{array}
\]

The number of subproblems is therefore linear.

ii. The input is $x_1, \ldots, x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$.

\[
\begin{array}{cccccccc}
  x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} \\
  y_1 & y_2 & y_3 & y_4 & y_5 & y_6 & y_7 & y_8 \\
\end{array}
\]

The number of subproblems is $O(mn)$.
iii. The input is $x_1, \ldots, x_n$ and a subproblem is $x_i, x_{i+1}, \ldots, x_j$.

\[
\begin{array}{ccccccc}
  & & & & x_3 & x_4 & x_5 & x_6 \\
 x_1 & x_2 & & & x_7 & x_8 & x_9 & x_{10}
\end{array}
\]

The number of subproblems is $O(n^2)$.

iv. The input is a rooted tree. A subproblem is a rooted subtree.