Introduction to Algorithms

Chapter 15: Dynamic Programming

Dynamic Programming

- Well known algorithm design techniques:
  - Brute-Force (iterative) algorithms
  - Divide-and-conquer algorithms

- Another strategy for designing algorithms is dynamic programming.
  - Used when problem breaks down into recurring small subproblems

Dynamic programming is typically applied to optimization problems.
  - In such problem there can be many solutions. Each solution has a value, and we wish to find a solution with the optimal value.

Divide-and-conquer

- Divide-and-conquer method for algorithm design:
  - Divide: If the input size is too large to deal with in a straightforward manner, divide the problem into two or more disjoint subproblems
  - Conquer: conquer recursively to solve the subproblems
  - Combine: Take the solutions to the subproblems and "merge" these solutions into a solution for the original problem

Divide-and-conquer - Example

- For example, MergeSort

  MergeSort(A, p, q)
  if p < q then
  q ← (p+q)/2
  MergeSort(A, p, q)
  MergeSort(A, q+1, r)
  Merge(A, p, q, r)

- The subproblems are independent, all different.
Dynamic programming

- Dynamic programming is a way of improving on inefficient divide-and-conquer algorithms.
- By "inefficient", we mean that the same recursive call is made over and over.
- If same subproblem is solved several times, we can use table to store result of a subproblem the first time it is computed and thus never have to recompute it again.
- Dynamic programming is applicable when the subproblems are dependent, that is, when subproblems share subsubproblems.
- "Programming" refers to a tabular method

Difference between DP and Divide-and-Conquer

- Using Divide-and-Conquer to solve these problems is inefficient because the same common subproblems have to be solved many times.
- DP will solve each of them once and their answers are stored in a table for future use.

Elements of Dynamic Programming (DP)

DP is used to solve problems with the following characteristics:

- Simple subproblems
  - We should be able to break the original problem to smaller subproblems that have the same structure
- Optimal substructure of the problems
  - The optimal solution to the problem contains within optimal solutions to its subproblems.
- Overlapping subproblems
  - there exist some places where we solve the same subproblem more than once.

Steps to Designing a Dynamic Programming Algorithm

1. Characterize optimal substructure
2. Recursively define the value of an optimal solution
3. Compute the value bottom up
4. (if needed) Construct an optimal solution
Fibonacci Numbers

- $F_n = F_{n-1} + F_{n-2}$, $n \geq 2$
- $F_0 = 0$, $F_1 = 1$
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
- Straightforward recursive procedure is slow!
- Let’s draw the recursion tree

Fibonacci Numbers

We can calculate $F_n$ in linear time by remembering solutions to the solved subproblems – dynamic programming

Compute solution in a bottom-up fashion

In this case, only two values need to be remembered at any time

Ex1: Assembly-line scheduling

- Cars factory with two assembly lines.
  - Each line has the same number “n” of stations. Numbered $j = 1, 2, ..., n$.
  - We denote the $j$th station on line $i$ (where $i$ is 1 or 2) by $S_{i,j}$.
  - The $j$th station on line 1 ($S_{1,j}$) performs the same function as the $j$th station on line 2 ($S_{2,j}$).
  - The time required at each station varies, even between stations at the same position on the two different lines, as each assembly line has different technology.
  - time required at station $S_{i,j}$ is $(a_{i,j})$.
  - There is also an entry time ($e_i$) for the chassis to enter assembly line $i$ and an exit time ($x_i$) for the completed auto to exit assembly line $i$. 
**Ex1: Assembly-line scheduling**

- After going through station $S_{ij}$, can either
  - q stay on same line
    - next station is $S_{i,j+1}$
    - no transfer cost, or
  - q transfer to other line
    - next station is $S_{3,i-j+1}$
    - transfer cost from $S_{ij}$ to $S_{3,i-j+1}$ is $t_{ij}$ ($j = 1, \ldots, n-1$)
    - No $t_{i,n}$ because the assembly line is complete after $S_{i,n}$

**Problem Definition**

- **Problem**: Given all these costs, what stations should be chosen from line 1 and from line 2 for minimizing the total time for car assembly.

- “Brute force” is to try all possibilities.
  - requires to examine $\Omega(2^n)$ possibilities
  - Trying all $2^n$ subsets is infeasible when $n$ is large.
  - Simple example: 2 stations $\rightarrow (2^n)$ possibilities $= 4$

**Step 1: Optimal Solution Structure**

- **optimal substructure**: choosing the best path to $S_{ij}$.

  - The structure of the fastest way through the factory (from the starting point)

  - The fastest possible way to get through $S_{i,1}$ ($i = 1, 2$)
    - Only one way: from entry starting point to $S_{i,1}$
    - take time is entry time ($e_i$)
**Step 1: Optimal Solution Structure**

- The fastest possible way to get through $S_{i,j}$ ($i = 1, 2$) ($j = 2, 3, ..., n$). Two choices:
  - Stay in the same line: $S_{i,j-1} \rightarrow S_{i,j}$
  - Time is $T_{i,j-1} + a_{i,j}$
    - If the fastest way through $S_{i,j}$ is through $S_{i,j-1}$, it must have taken a fastest way through $S_{i,j-1}$
  - Transfer to other line: $S_{3,i,j-1} \rightarrow S_{i,j}$
  - Time is $T_{3,i,j-1} + t_{3,i,j-1} + a_{i,j}$
    - Same as above

**Example**

- $S_{1,1} \rightarrow 2 + 7 = 9$
- $S_{2,1} \rightarrow 4 + 8 = 12$

**Step 1: Optimal Solution Structure**

- An optimal solution to a problem
  - finding the fastest way to get through $S_{ij}$
  - contains within it an optimal solution to sub-problems
    - finding the fastest way to get through either $S_{i,j-1}$ or $S_{3,i,j-1}$
  - Fastest way from starting point to $S_{ij}$ is either:
    - The fastest way from starting point to $S_{i,j-1}$ and then directly from $S_{i,j-1}$ to $S_{ij}$
    - Or
    - The fastest way from starting point to $S_{3,i,j-1}$ then a transfer from line 3 to line $i$ and finally to $S_{ij}$

→ **Optimal Substructure.**
Step 2: Recursive Solution

- Define the value of an optimal solution recursively in terms of the optimal solution to sub-problems
- Sub-problem here
  - finding the fastest way through station \( j \) on both lines \((i=1,2)\)
  - Let \( f_{i,j} \) be the fastest possible time to go from starting point through \( S_{i,j} \)
  - The fastest time to go all the way through the factory: \( f^* \)
    \[
    f^* = \min(f_{1,n} + x_1, f_{2,n} + x_2)
    \]
- \( x_1 \) and \( x_2 \) are the exit times from lines 1 and 2, respectively

---

**Example**

<table>
<thead>
<tr>
<th>( j )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f )</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>( e_1 )</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>( t_i )</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( x_i )</td>
<td>8</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

---

**Diagram**
Step 2: Recursive Solution

To help us keep track of how to construct an optimal solution, let us define
- $l_{ij}$: line # whose station $j-1$ is used in a fastest way through $S_{ij}$ (i = 1, 2, and j = 2, 3, ..., n)
- we avoid defining $l_{i1}$ because no station precedes station 1 on either lines.

We also define
- $l^*$: the line whose station n is used in a fastest way through the entire factory.

Using the values of $l^*$ and $l_{ij}$ shown in Figure (b) in next slide, we would trace a fastest way through the factory shown in part (a) as follows:
- Starting with $l^* = 1$, use station $S_{1,6}$
- $l_{16} = 2$, use station $S_{2,5}$
- $l_{25} = 2$, use station $S_{2,4}$
- $l_{24} = 1$, use station $S_{1,3}$
- $l_{13} = 2$, use station $S_{2,2}$
- $l_{22} = 1$, use station $S_{1,1}$

The fastest total time comes from choosing stations:
- Line 1: 1, 3, & 6
- Line 2: 2, 4, & 5

Min finish time $f^* = 38$.

---

**Example**

<table>
<thead>
<tr>
<th>j</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$</td>
<td>9</td>
<td>18</td>
<td>20</td>
<td>24</td>
<td>32</td>
<td>35</td>
</tr>
<tr>
<td>$f_2$</td>
<td>12</td>
<td>16</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>37</td>
</tr>
</tbody>
</table>

---

**Step 2: Recursive Solution**

Using the values of $l^*$ and $l_{ij}$ shown in Figure (b) in next slide, we would trace a fastest way through the factory shown in part (a) as follows:
- Starting with $l^* = 1$, use station $S_{1,6}$
- $l_{16} = 2$, use station $S_{2,5}$
- $l_{25} = 2$, use station $S_{2,4}$
- $l_{24} = 1$, use station $S_{1,3}$
- $l_{13} = 2$, use station $S_{2,2}$
- $l_{22} = 1$, use station $S_{1,1}$

The fastest total time comes from choosing stations:
- Line 1: 1, 3, & 6
- Line 2: 2, 4, & 5

Min finish time $f^* = 38$.

---

To help us keep track of how to construct an optimal solution, let us define
- $l_{ij}$: line # whose station $j-1$ is used in a fastest way through $S_{ij}$ (i = 1, 2, and j = 2, 3, ..., n)
- we avoid defining $l_{i1}$ because no station precedes station 1 on either lines.

We also define
- $l^*$: the line whose station n is used in a fastest way through the entire factory.
Step 3: Optimal Solution Value

- You can write a divide-and-conquer recursive algorithm to compute $f^*$.

- Inefficient implementation

```java
public int f(int i, int j) {
    if (j == 1) return e[i][1]+a[i][1];
    else {
        int temp1 = f(i, j-1);
        int temp2 = f(3-i, j-1)+t[3-i][j-1];
        return min(temp1, temp2)+a[i][j];
    }
}
```

Step 3: Optimal Solution Value

Sketch of proof:
- Let $T(j)$ be the time taken to execute $f(i, j)$
- $T(1) = c$ no further recursive calls
- $T(j) = c + 2 T(j-1)$
  - $f(i, j)$ calls both $f(i, j-1)$ and $f(3-i, j-1)$
- Expand the expression until $T(1)$ ...

- So top down isn’t a good way to compute $f[i][j]$.

Use substitution method to solve it $\mathcal{O}(2^n) \rightarrow$ exponential
Step 3: Optimal Solution Value

- We can do much better if we compute the $f[j]$ values in different order from the recursive way.

- **Observation:**
  - $f[j]$ depends only on $f[j-1]$ and $f_2[j-1]$ for $j \geq 2$

- So compute $f[j]$ in order of increasing $j$ and store $f[j]$ in a table
  - $\Theta(n)$

---

```java
public int fastestWay() {
    f1[1] = e1 + a1[1];
    f2[1] = e2 + a2[1];
    for (int j=2; j<=n; j++) {
        f1[j] = min(f1[j-1], f2[j-1] + t2[j-1])
            + a1[j-1];
        f2[j] = min(f2[j-1], f1[j-1] + t1[j-1])
            + a2[j-1];
    }
    return min(f1[n]+x1; f2+x2);
}
```

---

Step 4: Optimal Solution

**Constructing the fastest way through the**

```
PRINT-STATIONS (l, n)
1 i ← l
2 print “line” i “, station” n
3 for j ← n downto 2
4 do i ← l[i]
5 print “line” i “, station” j – 1
```

- line 1, station 6
- line 2, station 5
- line 2, station 4
- line 1, station 3
- line 2, station 2
- line 1, station 1
Multiplying Matrices

Two matrices, $A$ with $(p \times q)$ matrix and $B$ with $(q \times r)$ matrix, can be multiplied to get $C$ with dimensions $p \times r$, using scalar multiplications:

$$
\begin{pmatrix}
    a_{11} & a_{12} \\
    a_{21} & a_{22} \\
    a_{31} & a_{32}
\end{pmatrix}
\begin{pmatrix}
    b_{11} & b_{12} & b_{13} \\
    b_{21} & b_{22} & b_{23} \\
    \vdots & \vdots & \vdots \\
    b_{m1} & b_{m2} & b_{m3}
\end{pmatrix} =
\begin{pmatrix}
    c_{11} & c_{12} & c_{13} \\
    c_{21} & c_{22} & c_{23} \\
    \vdots & \vdots & \vdots \\
    c_{m1} & c_{m2} & c_{m3}
\end{pmatrix}
\quad c_{i,j} = \sum_{l=1}^{m} a_{i,l} \cdot b_{l,j}
$$

Ex2: Matrix-chain multiplication

We are given a sequence (chain) $<A_1, A_2, ..., A_n>$ of $n$ matrices to be multiplied, and we wish to compute the product $A_1 \cdot A_2 \cdots A_n$.

Matrix multiplication is associative, and so all parenthesizations yield the same product:

$$(A_1 (A_2 (A_3 A_4))),
(A_1 ((A_2 A_3) A_4)),
((A_1 A_2) (A_3 A_4)),
((A_1 (A_2 A_3)) A_4),
(((A_1 A_2) A_3) A_4).$$

Matrix-chain multiplication – cont.

We can multiply two matrices $A$ and $B$ only if they are compatible:

- the number of columns of $A$ must equal the number of rows of $B$.

If $A$ is a $p \times q$ matrix and $B$ is a $q \times r$ matrix, the resulting matrix $C$ is a $p \times r$ matrix.

The time to compute $C$ is dominated by the number of scalar multiplications, which is $pqr$. 

Matrix-Multiply($A,B$):

```
if columns[A] != rows[B] then
    error "incompatible dimensions"
else for i = 1 to rows[A] do
    for j = 1 to columns[B] do
        for k = 1 to columns[A] do
return C
```
Matrix-chain multiplication — cont.

- Ex: consider the problem of a chain \(<A_1, A_2, A_3>\) of three matrices, with the dimensions of \(10 \times 100, 100 \times 5, \) and \(5 \times 50\), respectively.
- If we multiply according to \(((A_1A_2)A_3)\), we perform \(10 \cdot 100 \cdot 5 = 5000\) scalar multiplications to compute the \(10 \times 5\) matrix product \(A_1A_2\), plus another \(10 \cdot 5 \cdot 50 = 2500\) scalar multiplications to multiply this matrix by \(A_3\) for a total of \(7500\) scalar multiplications.
- If instead, we multiply as \((A_1(A_2A_3))\), we perform \(100 \cdot 5 \cdot 50 = 25,000\) scalar multiplications to compute the \(100 \times 50\) matrix product \(A_2A_3\), plus another \(10 \cdot 100 \cdot 50 = 50,000\) scalar multiplications to multiply \(A_1\) by this matrix, for a total of \(75,000\) scalar multiplications.
- Thus, computing the product according to the first parenthesization is 10 times faster.

Matrix Chain Multiplication (MCM) Problem

- Input:
  - Matrices \(A_1, A_2, \ldots, A_n\), each \(A_i\) of size \(p_i-1 \times p_i\).
- Output:
  - Fully parenthesised product \(A_1 \times A_2 \times \ldots \times A_n\) that minimizes the number of scalar multiplications.
- Note:
  - In MCM problem, we are not actually multiplying matrices.
  - Our goal is only to determine an order for multiplying matrices that has the lowest cost.

Matrix-chain multiplication — cont.

- Matrix multiplication is associative
  - \((AB)C = A(BC)\)
- The parenthesization matters
- Consider \(A \times B \times C \times D\), where
  - \(A\) is \(30 \times 1\), \(B\) is \(1 \times 40\), \(C\) is \(40 \times 10\), \(D\) is \(10 \times 25\)
- Costs:
  - \(((AB)C)D\) = 1200 + 12000 + 7500 = 20700
  - \((AB)(CD)\) = 1200 + 10000 + 30000 = 41200
  - \(A((BC)D)\) = 400 + 250 + 750 = 1400
- Matrix Chain Multiplication (MCM) Problem

- Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications.
- So, exhaustively checking all possible parenthesizations does not yield an efficient algorithm.
Counting the Number of Parenthesizations

Denote the number of alternative parenthesizations of a sequence of \((n)\) matrices by \(P(n)\), then a fully parenthesized matrix product is given by:

\[
P(n) = \begin{cases} 
1 & \text{if } n = 1 \\
\sum_{k=1}^{n-1} P(k) P(n-k) & \text{if } n \geq 2
\end{cases}
\]

If \(n = 4\), then the number of alternative parenthesis is 5.

\[
P(4) = P(1)P(3) + P(2)P(2) + P(3)P(1) = 5
\]

Step 1: Optimal Sub-structure

- Let \(A_{i,j}(i \leq j)\) denote the result of \(A_i \times A_{i+1} \times \ldots \times A_j\).
- \(A_{i,j}\) can be obtained by splitting it into \(A_{i,k}\) and \(A_{k+1,j}\) and then multiplying the sub-products, \(i \leq k < j\).
- The cost of this parenthesization is:
  - cost of computing the matrix \(A_{i,k}\):
  - cost of computing the matrix \(A_{k+1,j}\):
  - cost of multiplying them together \((A_{i,k} \times A_{k+1,j}) = P_{i,k}P_{k,j}\):

Step 1: Optimal Sub-structure

- Consider \(A_{1,4} = A_1 \times A_2 \times A_3 \times A_4\):
  - \(A_{1,1}.A_{2,4}\) \(k = 1\)
  - \(A_{1,2}.A_{3,4}\) \(k = 2\)
  - \(A_{1,3}.A_{4,4}\) \(k = 3\)
  - There are \(j-i\) possible splits (i.e., \(k = i, \ldots, j-1\)).
  - In the above example, # of splits are 3 = 4-1.
Step 1: Optimal Sub-structure

- The optimal substructure of this problem is as follows:
  - Suppose that an optimal parenthesization of $A_iA_{i+1}...A_j$ splits the product between $A_i$ and $A_{k-1}$.
  - Within the optimal parenthesization of $A_{i,k}$, the parenthesization of $A_i..k$ must be optimal.
  - The parenthesization of $A_{k-1,j}$ must be optimal.
- The same thing holds for the sub-chain $A_{k-1}A_{k+2}...A_j$.

Step 1: Optimal Sub-structure

- The minimal cost $A_{i,j} = A_i A_{i+1} ... A_j$ is optimal.
- Suppose $A_i A_{i+1} (A_j A_k)$ and $(A_i A_j) A_k$ must be optimal for $A_i A_{i+1} A_j A_k A_l$.
- Otherwise, if $(A_i A_j A_k)$ is optimal for $A_i A_j A_k A_l A_m$.

Step 2: Recursive Solution

- For MCM, sub-problems are:
  - problems of determining the min-cost of a parenthesization of $A_i A_{i+1}...A_j$ for $1 <= i <= j <= n$.
- Let $m[i,j] = \text{min } \# \text{ of scalar multiplications}$ needed to compute the matrix $A_i...A_j$.
  - For the whole problem, we need to find $m[1,n]$.
- Since $A_i...A_j$ can be obtained by breaking it into $A_i...A_k A_{k+1}...A_j$, we have (each $A_i$ of size $p_i x p_i$):
  - $m[i,j] = m[i,k] + m[k+1,j] + p_i p_k p_j$.
- Thus:
  - $m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i < k < j} \{m[i,k] + m[k+1,j] + p_i p_k p_j\} & \text{if } i < j \end{cases}$
- The $m[i,j]$ values give the cost of optimal solutions to sub-problems.
- Let $s[i,j]$ be the value $k$ where the optimal split occurs.
  - $m[i,j] = m[i,k] + m[k+1,j] + p_i p_k p_j$.

Step 2: Recursive Solution
Step 3: Computing the Optimal Costs

- A direct recursive implementation is exponential
  - there is a lot of duplicated work (why?)
  - each sub-problem may be computed many times in different branches of its recursion tree

![Recursion Tree](image)

Overlapping Sub-problems

The following code assumes that matrix $A_i$ has dimensions $p_{i-1} \times p_i$ for $i=1,2,\ldots,n$

```python
Matrix-Chain-Order(P, P_1, \ldots, P_n)
1 for i \leftarrow 1 \text{ to } n do
2 \quad m[i,i] \leftarrow 0 // no multiplication for 1 matrix
3 for d \leftarrow 2 \text{ to } n do // d: length of sub-chain
4 \quad for i \leftarrow 1 \text{ to } n-d+1 do // i: start of sub-chain
5 \quad \quad j \leftarrow i+d-1 // j: end of sub-chain
6 \quad m[i,j] \leftarrow \infty
7 \quad for k \leftarrow i \text{ to } j-1 do
8 \quad \quad q \leftarrow m[i,k] + m[k+1,j] + P_{i-1}P_kP_j
9 \quad if q < m[i,j] then
10 \quad \quad m[i,j] \leftarrow q
11 \quad s[i,j] \leftarrow k
12 return m, s
```

Elements of Dynamic Programming

- DP is used to solve problems with the following characteristics:
  - Optimal substructure (Principle of Optimality)
    - optimal solution to the problem is constructed from the
      optimal solutions to subproblems
    - within the optimal parenthesization of $A_{i,r}$
      - the parenthesization of $A_{i,k}$ must be optimal
      - the parenthesization of $A_{k+1,r}$ must be optimal
  - Overlapping subproblems
    - there exist some places where we solve the same subproblem
      more than once
    - In MCM, $A_{i,j}$ is common to the sub-problems $A_{i,j}$ and $A_{j+1}$
    - Effort wasted in solving common sub-problems repeatedly

![Optimal Substructure](image)

Step 3: Computing the Optimal Costs

- $m[i, j] = 0$
- \begin{verbatim}
Matrix-Chain-Order((P_0, P_1, ..., P_n))
for i ← 1 to n do
    m[i,i] ← 0 // no multiplication for 1 matrix
for i = 1 to n do
    for j = i to n do // d: length of sub-chain
        m[i,j] ← ∞
        for k ← i to j-1 do // start of sub-chain
            q ← m[i,k] + m[k+1,j] + P_{i-1}P_kP_j
            if q < m[i,j] then
                m[i,j] ← q
                s[i,j] ← k
\end{verbatim}

- $m[i, i+1]$ for $i = 1, 2, ..., n-1$ for chains of length $d = 2$
- $m[i, i+2]$ for $i = 1, 2, ..., n-2$ for chains of length $d = 3$
- ...

Step 3: Computing the Optimal Costs

- Example:
- Solve the following MCM instance ($n = 6$):
  - Matrix  
    - $A_1$: 30 × 35
    - $A_2$: 35 × 15
    - $A_3$: 15 × 5
    - $A_4$: 5 × 10
    - $A_5$: 10 × 20
    - $A_6$: 20 × 25
  - $p = (30, 35, 15, 5, 10, 20, 25)$

- At each step, the $m[i, j]$ cost computed at lines 9-12 depends only on table entries $m[i, k]$ and $m[k+1, j]$ already computed.
- After execution:
  - $m[1, n]$ contains the value of the optimal solution and
  - $s$ contains optimal subdivisions (choices of $k$) of any sub-problem into two sub-sub-problems

- Running time
  - It is easy to see that it is $O(n^3)$
    - $O(n^3)$ entries, $O(n)$ time to compute each $→ O(n^3)$ total time
  - It turns out, it is also $Ω(n^3)$

- From exponential time to polynomial
Step 3: Computing the Optimal Costs

- The previous table shows the structure of the problem $m[1,6]$ and all the subproblems, $m[1,2]$, $m[1,3]$, etc.

- The problem $m[1,6]$ is at the top of the table with the cost of 15125.

- As an example, the subproblem $m[2,5]$ is highlighted in the darkly shaded square.

- Notice that this subproblem is made up of the further subproblems to the left and down.

Step 4: Constructing an Optimal Solution

- Although **Matrix-Chain-Order** determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it *does not* directly show how to multiply the matrices.

- Using table $s$, we can construct an optimal solution.

Step 3: Computing the Optimal Costs

- The minimal cost of $m[2,5]$ is found by exhaustively searching all values of $k$.
  - $m[2,5] = \min \{ \begin{align*}
  m[2,2]+m[3,5]+p_3p_5 &= 0+2500+35 \times 15 \times 20 = 13000, \\
  m[2,3]+m[4,5]+p_3p_5 &= 2625+1000+35 \times 5 \times 20 = 7125, \\
  m[2,4]+m[5,5]+p_3p_5 &= 4375+0+35 \times 10 \times 20 = 11375
  \end{align*}\}
  - $= 7125$

$$
\begin{align*}
  m[3,6] &= \min \left\{ m[3,4] \cdot m[5,5] + 15 \times 10 \times 20 \\
  &\quad= 750 + 0 + 3000 = 3750, \\
  m[3,3] \cdot m[4,5] + 15 \times 5 \times 20 \\
  &\quad= 0 + 1000 + 1500 = 2500
\right\}
\end{align*}
$$

Step 4: Constructing an Optimal Solution

- The previous table $s$ table which is structured just like the $m$ table.

- Each entry in the table is indexed by the $[i, j]$ pair.

- The $s$ table contains the calculated $k$ value for each entry in the $m$ table.

- The $s$ table can be used to formulate the optimal parenthesization of the matrix chain.

- This completes the structure of the problem.
Step 4: Constructing an Optimal Solution

- Each $s[i, j]$ entry contains the calculated $k$ value such that the optimal parenthesization of $A_{i,j} = A_{i:k}A_{k:j}$ splits the product between to $A_{i,k}$ and $A_{k,j}$.

- A simple recursive algorithm `Print-Optimal-Parents(s, i, j)` can be used to reconstruct an optimal parenthesization.

---

Example: Matrix Chain Multiplication

- What if we want to multiply a string of more than two matrices together?

- Matrix multiplication is associative, that is $A(BC) = (AB)C$, so the result is the same no matter in what order we multiply the matrices, but it can make a big difference in the amount of work that is done.

```
Print-Optimal-Parentheses(s, i, j)
1  if i == j
2    then print "A"i
3  else print "("
4    PrintOptimalParentheses(s, i, s[i, j])
5  PrintOptimalParentheses(s, s[i, j]+1, j)
6  Print ")"
```

- The optimal split of $A_{i,n}$ is stored in $s[i,n]
  - $A_{i,[1,n]}$ $A_{[i,n]+1,n}$

---

Example: Matrix Chain Multiplication

```
((A1A2)A3)A4 30*1*40 + 30*40*10 + 30*10*25 = 20,700
A1(A2(A3A4)) 40*10*25 + 1*40*25 + 30*1*25 = 11,750
(A1A2)(A3A4) 30*1*40 + 40*10*25 + 30*40*25 = 41,200
A1((A2A3)A4) 1*40*10 + 1*10*25 + 30*1*25 = 1,400
```
First, the costs of multiplying chains of length 1 are initialized to 0.

Then the cost of multiplying chains of length 2 are calculated (only one way to multiply chains of length two).

Then the best way to multiply chains of length 3 are calculated.
Then the best way to Multiply the entire chain is calculated

\[
\begin{array}{cccc|c}
\text{A}_1 & \times & \text{A}_2 & \times & \text{A}_3 & \text{A}_4 \\
30 \times 1 & 1 \times 40 & 40 \times 10 & 10 \times 25 & \text{cost} = \\
\hline
0 & 1200 & 700 & 1400 & [1] \\
0 & 400 & 650 & [2] \\
0 & 10000 & [3] \\
\hline
\end{array}
\]

\[
\begin{align*}
(\text{A}_1 \times (\text{A}_2 \times \text{A}_3 \times \text{A}_4)) & = 0 + 650 + 30 \times 1 \times 25 = 1400 \\
(\text{A}_1 \times \text{A}_2 \times \text{A}_3) \times \text{A}_4 & = 700 + 0 + 30 \times 10 \times 25 = 10,450 \\
(\text{A}_1 \times \text{A}_2) \times (\text{A}_3 \times \text{A}_4) & = 1200 + 10,000 + 30 \times 40 \times 25 = 41,200 \\
\end{align*}
\]

Dynamic Programming Solution

\[
\text{PRINT-OPTIMAL-PARENS}(\text{split}, i, j) \\
\text{if}(i == j) \quad \text{// base case -- chain of length 1, just print A_i} \\
\text{print} \quad "A"_i \\
\text{else} \\
\text{print} \quad "(" \\
\text{PRINT-OPTIMAL-PARENS}(\text{split}, i, \text{split}[i,j]) \\
\text{PRINT-OPTIMAL-PARENS}(\text{split}, \text{split}[i,j]+1, j) \\
\text{print} \quad ")"
\]

\[
\begin{array}{cccc}
\text{split} = \\
\hline
\hline
1 & 1 & 1 & 1 \\
2 & 3 & 3 & 3 \\
\hline
\end{array}
\]

The array split is used to reconstruct the optimal ordering

\[
(\text{A}_1 ((\text{A}_2 \text{A}_3) \text{A}_4))
\]
Longest Common Subsequence

- In biological applications, we often want to compare the DNA of two (or more) different organisms.

- A strand of DNA consists of a string of molecules called bases, where the possible bases are adenine, quanine, cytosine, and thymine (A, C, G, T).

Similarity can be defined in different ways:

- Two DNA strands are similar if one is a substring of the other.

- Two strands are similar if the number of changes needed to turn one into the other is small.

- There is a third strand $S_3$ in which the bases in $S_3$ appear in each of $S_1$ and $S_2$; these bases must appear in the same order, but not necessarily consecutively. The longer the strand $S_3$ we can find, the more similar $S_1$ and $S_2$ are.

Example of Longest Common Subsequence

LCS for $S_1$ and $S_2$:

$S_1=\text{ACC\text{GG}TC\text{GA}GT\text{CG}C\text{G}A\text{G}C\text{CG}G\text{CCG}A\text{A}}$

$S_2=\text{GT\text{CG}G\text{TA}G\text{AA}T\text{G}C\text{CG}T\text{T}G\text{C}T\text{G}C\text{T}A\text{A}A}$

$S_3=\text{GT\text{CG}G\text{TA}G\text{AA}T\text{G}C\text{CG}T\text{T}G\text{C}T\text{G}C\text{T}A\text{A}A}$

Example

$X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

$\langle B, C, B, A \rangle$ and $\langle B, D, A, B \rangle$ are longest common subsequences of $X$ and $Y$ (length = 4)

$\langle B, C, A \rangle$, however is not a LCS of $X$ and $Y$
LCS Brute-force solution

- Enumerate all subsequences of $X$ and check each subsequence to see if it is also a subsequence of $Y$, keeping track of the longest subsequence found.
- Each subsequence of $X$ corresponds to a subset of the indices $\{1, 2, \ldots, m\}$ of $X$.
- There are $2^m$ subsequences of $X$.
- Exponential Time!

A Recursive Solution

Case 1: $x_i = y_j$

e.g.: $X = \langle A, B, D, E \rangle$
$Y = \langle Z, B, E \rangle$

$$c[i, j] = c[i - 1, j - 1] + 1$$

- Append $x_i = y_j$ to the LCS of $X_{i-1}$ and $Y_{j-1}$
- Must find a LCS of $X_{i-1}$ and $Y_{j-1}$ => optimal solution to a problem includes optimal solutions to subproblems

Case 2: $x_i \neq y_j$

e.g.: $X = \langle A, B, D, G \rangle$
$Y = \langle Z, B, D \rangle$

$$c[i, j] = \max\{ c[i - 1, j], c[i, j - 1]\}$$

- Must solve two problems
  - find a LCS of $X_{i-1}$ and $Y_j$: $X_{i-1} = \langle A, B, D \rangle$ and $Y_j = \langle Z, B, D \rangle$
  - find a LCS of $X_i$ and $Y_{j-1}$: $X_i = \langle A, B, D, G \rangle$ and $Y_{j-1} = \langle Z, B \rangle$
- Optimal solution to a problem includes optimal solutions to subproblems

Computing the Length of the LCS

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } x_i \neq y_j \end{cases}$$
**LCS-Length(X, Y) // dynamic programming solution**

\[ m = \text{X.length()} \]
\[ n = \text{Y.length()} \]

- For \( i = 0 \) to \( m \) do \( c[i,0] = 0 \)
- For \( j = 0 \) to \( n \) do \( c[0,j] = 0 \)

\[ O(nm) \]

- For \( i = 1 \) to \( m \) do // row
  - For \( j = 1 \) to \( n \) do // column
    - If \( x_i = y_j \)
      - \( b[i,j] = \) "\"
    - Else if \( c[i-1, j] \geq c[i, j-1] \)
      - \( b[i,j] = \) "^" 
    - Else \( c[i,j] = c[i, j-1] \)
      - \( b[i,j] = \) "<"

\[ O(nm) \]
To construct the LCS, start in the bottom right-hand corner and follow the arrows. A \( \Rightarrow \) indicates a matching character.
Constructing an LCS

Print-LCS(b,X,i,j)
if i = 0 or j = 0 then
    return
if b[i,j] = “K” then
    Print-LCS(b, X, i-1, j-1)
    print x_i
else if b[i,j] = “^” then
    Print-LCS(b, X, i-1, j)
else Print-LCS(b, X, i, j-1)

Knapsack problem

- There are two versions of the problem:
  (1) “0-1 knapsack problem” and
  (2) “Fractional knapsack problem”

(1) Items are indivisible; you either take an item or not. Solved with dynamic programming.
(2) Items are divisible: you can take any fraction of an item. Solved with a greedy algorithm.

0-1 Knapsack problem

- Thief has a knapsack with maximum capacity $W$, and a set $S$ consisting of $n$ items.
- Each item $i$ has some weight $w_i$ and benefit value $v_i$ (all $w_i$, $v_i$ and $W$ are integer values).
- Problem: How to pack the knapsack to achieve maximum total value of packed items?
- Goal:
  - find $x_i$ such that for all $x_i \in \{0, 1\}, i = 1, 2, .., n$
  - $\sum w_ix_i \leq W$ and
  - $\sum x_iv_i$ is maximum

0-1 Knapsack - Greedy Strategy

**E.g. 1:**

<table>
<thead>
<tr>
<th>Item</th>
<th>Benefit</th>
<th>Weight</th>
<th>$v_i$/pound</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Knapsack holds a maximum of 22 pounds.
Fill it to get the maximum benefit.

**E.g. 2:**

<table>
<thead>
<tr>
<th>Item</th>
<th>Benefit</th>
<th>Weight</th>
<th>$v_i$/pound</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>10</td>
<td>4</td>
</tr>
</tbody>
</table>

Knapsack holds a maximum of 22 pounds.
Fill it to get the maximum benefit.
0-1 Knapsack problem: brute-force approach

- Let’s first solve this problem with a straightforward algorithm.
- Since there are $n$ items, there are $2^n$ possible combinations of items.
- We go through all combinations and find the one with the most total value and with total weight less or equal to $W$.
- Running time will be $O(2^n)$.

0-1 Knapsack - Dynamic Programming

- $P(i, w)$ – the maximum profit that can be obtained from items 1 to $i$, if the knapsack has size $w$.
  - Case 1: thief takes item $i$.
    $P(i, w) = v_i + P(i - 1, w - w_i)$.
  - Case 2: thief does not take item $i$.
    $P(i, w) = P(i - 1, w)$.

Recursive Formula

$$P(i, w) = \begin{cases} P[i-1, w] & \text{if } w_j > w \\ \max \{v_i + P[i-1, w-w_k], P[i-1, w] \} & \text{else} \end{cases}$$

- The best subset that has the total weight $w$, either contains item $i$ or not.
- First case: $w_i > w$. Item $i$ can’t be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable.
- Second case: $w_i \leq w$. Then the item $i$ can be in the solution, and we choose the case with greater value.

0-1 Knapsack - Dynamic Programming

The table illustrates the dynamic programming solution, with the final profit $P(i, w)$ calculated for each item and weight combination.
Example:

\[ P(i, w) = \max \{v_i + P(i - 1, w - w_i), P(i - 1, w) \} \]

<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>
</tbody>
</table>

\[ P(1, 1) = P(1, 2) = \max(12 + 0, 0) = 12 \]
\[ P(0, 1) = 0 \]

Optimal Substructure

- Consider the most valuable load that weighs at most \( W \) pounds
- If we remove item \( j \) from this load
  - The remaining load must be the most valuable load weighing at most \( W - w_j \) that can be taken from the remaining \( n - 1 \) items

Reconstructing the Optimal Solution

- Start at \( P(n, W) \)
- When you go left-up \( \Rightarrow \) item \( i \) has been taken
- When you go straight up \( \Rightarrow \) item \( i \) has not been taken

0-1 Knapsack Algorithm

for \( w = 0 \) to \( W \)

\[ P[0,w] = 0 \]

for \( i = 0 \) to \( n \)

\[ P[i,0] = 0 \]

for \( w = 0 \) to \( W \)

if \( w_i \leq w \) // item \( i \) can be part of the solution

if \( v_i + P[i-1,w-w_i] > P[i-1,w] \)

\[ P[i,w] = v_i + P[i-1,w-w_i] \]

else

\[ P[i,w] = P[i-1,w] \] // \( w_i > w \]
## Overlapping Subproblems

\[
P(i, w) = \max \{v_i + P(i - 1, w-w_i), P(i - 1, w) \}
\]

\[
\begin{array}{cccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & & & & & & & & & \\
i-1 & 0 & & & & & & & & \\
i & 0 & & & & & & & & \\
n & 0 & & & & & & & & \\
\end{array}
\]

*E.g.*: all the subproblems shown in grey may depend on \(P(i-1, w)\)

## Example

Let's run our algorithm on the following data:

- \(n = 4\) (# of elements)
- \(W = 5\) (max weight)
- Elements (weight, benefit): (2,3), (3,4), (4,5), (5,6)

## Conclusion

- Dynamic programming is a useful technique of solving certain kind of problems

- When the solution can be recursively described in terms of partial solutions, we can store these partial solutions and re-use them as necessary