Algorithmic Paradigms

Divide and Conquer
Idea: Divide problem instance into smaller sub-instances of the same problem, solve these recursively, and then put solutions together to a solution of the given instance.
Examples: Mergesort, Quicksort, Strassen’s algorithm, FFT.

Greedy Algorithms
Idea: Find solution by always making the choice that looks optimal at the moment — don’t look ahead, never go back.
Examples: Prim’s algorithm, Kruskal’s algorithm.

Dynamic Programming
Idea: Turn recursion upside down.
Example: Floyd-Warshall algorithm for the all pairs shortest path problem.

Dynamic Programming - A Toy Example
Fibonacci Numbers
\[ F_0 = 0, \]
\[ F_1 = 1, \]
\[ F_n = F_{n-1} + F_{n-2} \quad (\text{for } n \geq 2). \]

A recursive algorithm

Algorithm REC-FIB(n)
1. if \( n = 0 \) then
2. \quad return 0
3. else if \( n = 1 \) then
4. \quad return 1
5. else
6. \quad return \( \text{REC-FIB}(n-1) + \text{REC-FIB}(n-2) \)

Ridiculously slow: exponentially many repeated computations of \( \text{REC-FIB}(j) \) for small values of \( j \).

Fibonacci Example (cont’d)
Why is the recursive solution so slow?
Running time \( T(n) \) satisfies
\[ T(n) = T(n - 1) + T(n - 2) + \Theta(1) \geq F_n \sim (1.6)^n. \]

BOARD: We show \( F_n \geq \frac{1}{2}(3/2)^n \) for \( n \geq 8 \).
Fibonacci Example (cont’d)

Dynamic Programming Approach

Algorithm DYN-FIB(n)
1. \( F[0] = 0 \)
2. \( F[1] = 1 \)
3. for \( i \leftarrow 2 \) to \( n \) do
4. \( F[i] \leftarrow F[i - 1] + F[i - 2] \)
5. return \( F[n] \)

Build “from the bottom up”

Running Time
\( \Theta(n) \)

Very fast in practice - just need an array (of linear size) to store the \( F(i) \) values.

Example

Compute
\[
A \cdot B \cdot C \cdot D
\]
\[
30 \times 1 \quad 1 \times 40 \quad 40 \times 10 \quad 10 \times 25
\]

Multiplication order \( (A \cdot B) \cdot (C \cdot D) \) requires
\[
30 \cdot 1 \cdot 40 + 40 \cdot 10 \cdot 25 + 30 \cdot 40 \cdot 25 = 41,200
\]

Multiplication order \( A \cdot ((B \cdot C) \cdot D) \) requires
\[
1 \cdot 40 \cdot 10 + 1 \cdot 10 \cdot 25 + 30 \cdot 1 \cdot 25 = 1,400
\]

Multiplying Sequences of Matrices

Recall

Multiplying a \((p \times q)\) matrix with a \((q \times r)\) matrix (in the standard way) requires \( pqr \) multiplications.

We want to compute products of the form
\[
A_1 \cdot A_2 \cdots A_n.
\]

How do we set the parentheses?

The Matrix Chain Multiplication Problem

Input:
Sequence of matrices \( A_1, \ldots, A_n \), where \( A_i \) is a \( p_{i-1} \times p_i \)-matrix

Output:
Optimal number of multiplications needed to compute \( A_1 \cdot A_2 \cdots A_n \), and an optimal parenthesisation to realise this

Running time of algorithms will be measured in terms of \( n \).
Solution “Attempts”

Approach 1: Exhaustive search (CORRECT but SLOW).
Try all possible parenthesisations and compare them. Correct, but extremely slow; running time is $\Omega(3^n)$. UGLY PROOF

Approach 2: Greedy algorithm (INCORRECT).
Always do the cheapest multiplication first. Does not work correctly — sometimes, it returns a parenthesisation that is not optimal:

Example: Consider $A_1 \cdot A_2 \cdot A_3$

Solution proposed by greedy algorithm: $A_1 \cdot (A_2 \cdot A_3)$ with $100 \cdot 2 \cdot 2 + 3 \cdot 100 \cdot 2 = 1000$ multiplications.

Optimal solution: $(A_1 \cdot A_2) \cdot A_3$ with $3 \cdot 100 \cdot 2 + 3 \cdot 2 \cdot 2 = 612$ multiplications.

Solution “Attempts” (cont’d)

Approach 3: Alternative greedy algorithm (INCORRECT).
Set outermost parentheses such that cheapest multiplication is done last.

Doesn’t work correctly either (Exercise!).

Approach 4: Recursive (Divide and Conquer) - (SLOW - see over).

Divide:

$$(A_1 \cdots A_k) \cdot (A_{k+1} \cdots A_n)$$

For all $1 \leq i \leq j \leq n$, let

$$m[i,j] = \text{least number of multiplications needed to compute } A_i \cdots A_j$$

Then

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{1 \leq k < j} (m[i,k] + m[k+1,j] + p_{i-1}p_kp_j) & \text{if } i < j. \end{cases}$$

Dynamic Programming Solution

As before:

$$m[i,j] = \text{least number of multiplications needed to compute } A_i \cdots A_j$$

Moreover,

$$s[i,j] = \text{(the smallest) } k \text{ such that } i \leq k < j \text{ and}$$

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j.$$ 

$s[i,j]$ can be used to reconstruct the optimal parenthesisation.

Idea

Compute the $m[i,j]$ and $s[i,j]$ in a bottom-up fashion.

TURN RECURSION UPSIDE DOWN :-)
**Algorithm** Matrix-Chain-Order(p)

1. \( n \leftarrow p.length - 1 \)
2. for \( i \leftarrow 1 \) to \( n \) do
3. \( m[i,i] \leftarrow 0 \)
4. for \( \ell \leftarrow 2 \) to \( n \) do
5. for \( i \leftarrow 1 \) to \( n - \ell + 1 \) do
6. \( j \leftarrow i + \ell - 1 \)
7. \( m[i,j] \leftarrow \infty \)
8. for \( k \leftarrow i \) to \( j - 1 \) do
9. \( q \leftarrow m[i,k] + m[k+1,j] + p_{i-1}p_{k}p_{j} \)
10. if \( q < m[i,j] \) then
11. \( m[i,j] \leftarrow q \)
12. \( s[i,j] \leftarrow k \)
13. return \( s \)

**Running Time:** \( \Theta(n^3) \)

**Example**

<table>
<thead>
<tr>
<th></th>
<th>( A_1 )</th>
<th>( A_2 )</th>
<th>( A_3 )</th>
<th>( A_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>30 \times 1</td>
<td>1 \times 40</td>
<td>40 \times 10</td>
<td>10 \times 25</td>
</tr>
</tbody>
</table>

**Solution** for \( m \) and \( s \)

<table>
<thead>
<tr>
<th>( m )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1200</td>
<td>700</td>
<td>1400</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>400</td>
<td>650</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>10000</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

**Optimal Parenthesisation**

\( A_1 \cdot ((A_2 \cdot A_3) \cdot A_4) \)

**Multiplying the Matrices**

**Algorithm** Matrix-Chain-Multiply(A,p)

1. \( n \leftarrow A.length \)
2. \( s \leftarrow \text{Matrix-Chain-Order}(p) \)
3. return \( \text{Rec-Mult}(A,s,1,n) \)

**Algorithm** Rec-Mult(A,s,i,j)

1. if \( i < j \) then
2. \( C \leftarrow \text{Rec-Mult}(A,s,i,s[i,j]) \)
3. \( D \leftarrow \text{Rec-Mult}(A,s,s[i,j]+1,j) \)
4. return \( (C) \cdot (D) \)
5. else
6. return \( A_i \)

**Problems**

1. Review the Edit-Distance Algorithm and try to understand why it is a dynamic programming algorithm.
2. Exercise 15.2-1 of [CLRS].