Advanced Design and Analysis Techniques

Part 1

15.1 and 15.2
Note for the students:

- These slides are meant for the lecturers to conduct lectures only. It is **NOT** suitable to be used as a study material.
- Students are expected to study by reading the textbook for this course:
Techniques -1

• This part covers three important techniques for the design and analysis of efficient algorithms:
  – dynamic programming (Chapter 15),
  – greedy algorithms (Chapter 16), and
  – amortized analysis (Chapter 17).
Techniques - 2

• Earlier parts have presented other widely applicable techniques, such as
  – divide-and-conquer,
  – randomization, and
  – the solution of recurrences.
Dynamic programming

• Dynamic programming typically applies to optimization problems in which a set of choices must be made in order to arrive at an optimal solution.

• Dynamic programming is effective when a given subproblem may arise from more than one partial set of choices; the key technique is to store the solution to each such subproblem in case it should reappear.
Greedy algorithms

• Like dynamic-programming algorithms, greedy algorithms typically apply to optimization problems in which a set of choices must be made in order to arrive at an optimal solution. The idea of a greedy algorithm is to make each choice in a locally optimal manner.
Dynamic programming

- **Dynamic programming**, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.

- **Divide and conquer algorithms** partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.
Dynamic programming -2

- **Dynamic programming** is applicable when the subproblems are not independent, that is, when subproblems share subsubproblems.

- A dynamic-programming algorithm solves every subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subsubproblem is encountered.
Dynamic programming -2

• Dynamic programming is typically applied to *optimization problems*. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution, since there may be several solutions that achieve the optimal value.
The development of a dynamic-programming algorithm

• The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

  1. Characterize the structure of an optimal solution.

  2. Recursively define the value of an optimal solution.

  3. Compute the value of an optimal solution in a bottom-up fashion.

  4. Construct an optimal solution from computed information.
Assembly-line scheduling
Step 1: The structure of the fastest way through the factory

![Diagram of the factory structure]

(a) Graphical representation of the factory's stations and assembly lines.

(b) Tables showing the distances and optimal values:

<table>
<thead>
<tr>
<th></th>
<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>$f_1[j]$</td>
<td></td>
<td>9</td>
<td>18</td>
<td>20</td>
<td>24</td>
<td>32</td>
<td>35</td>
</tr>
<tr>
<td>$f_2[j]$</td>
<td></td>
<td>12</td>
<td>16</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>37</td>
</tr>
</tbody>
</table>

$f^* = 38$

<table>
<thead>
<tr>
<th></th>
<th>$j$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_1[j]$</td>
<td></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$I_2[j]$</td>
<td></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

$r^* = 1$
Step 2: A recursive solution

\[ f_1[1] = e_1 + a_{1,1} , \quad (15.2) \]
\[ f_2[1] = e_2 + a_{2,1} . \quad (15.3) \]

\[ f_1[j] = \min(f_1[j - 1] + a_{1,j}, f_2[j - 1] + t_{2,j-1} + a_{1,j}) \quad (15.4) \]

for \( j = 2, 3, \ldots, n \). Symmetrically, we have

\[ f_2[j] = \min(f_2[j - 1] + a_{2,j}, f_1[j - 1] + t_{1,j-1} + a_{2,j}) \quad (15.5) \]
Step 3: Computing the fastest times

FASTEST-WAY \((a, t, e, x, n)\)

1. \(f_1[1] \leftarrow e_1 + a_{1,1}\)
2. \(f_2[1] \leftarrow e_2 + a_{2,1}\)
3. for \(j \leftarrow 2\) to \(n\)
4. do if \(f_1[j - 1] + a_{1,j} \leq f_2[j - 1] + t_{2,j-1} + a_{1,j}\)
5. then \(f_1[j] \leftarrow f_1[j - 1] + a_{1,j}\)
6. \(l_1[j] \leftarrow 1\)
7. else \(f_1[j] \leftarrow f_2[j - 1] + t_{2,j-1} + a_{1,j}\)
8. \(l_1[j] \leftarrow 2\)
9. if \(f_2[j - 1] + a_{2,j} \leq f_1[j - 1] + t_{1,j-1} + a_{2,j}\)
10. then \(f_2[j] \leftarrow f_2[j - 1] + a_{2,j}\)
11. \(l_2[j] \leftarrow 2\)
12. else \(f_2[j] \leftarrow f_1[j - 1] + t_{1,j-1} + a_{2,j}\)
13. \(l_2[j] \leftarrow 1\)
14. if \(f_1[n] + x_1 \leq f_2[n] + x_2\)
15. then \(f^* = f_1[n] + x_1\)
16. \(l^* = 1\)
17. else \(f^* = f_2[n] + x_2\)
18. \(l^* = 2\)
Step 4: Constructing the fastest way through the factory

**PRINT-STATIONS**$(l, n)$

1. $i \leftarrow l^*$
2. print “line ” $i$ “, station ” $n$
3. for $j \leftarrow n$ downto 2
   4. do $i \leftarrow l[i][j]$
   5. print “line ” $i$ “, station ” $j - 1$

In the example of Figure 15.2, PRINT-STATIONS would produce the output

line 1, station 6
line 2, station 5
line 2, station 4
line 1, station 3
line 2, station 2
line 1, station 1
Matrix-chain multiplication

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.
Counting the number of parenthesizations

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

(15.11)
• Step 1: The structure of an optimal parenthesization
• Step 2: A recursive solution
• Step 3: Computing the optimal costs
Step 3: Computing the optimal costs

MATRIX-CHAIN-ORDER (p)
1 n ← length[p] − 1
2 for i ← 1 to n
3 do m[i, i] ← 0
4 for l ← 2 to n    ▷ l is the chain length.
5 do for i ← 1 to n − l + 1
6 do j ← i + l − 1
7 m[i, j] ← ∞
8 for k ← i to j − 1
9 do q ← m[i, k] + m[k + 1, j] + p_{i−1}p_kp_j
10 if q < m[i, j]
11 then m[i, j] ← q
12 s[i, j] ← k
13 return m and s
Figure 15.3  The \( m \) and \( s \) tables computed by \textsc{Matrix-Chain-Order} for \( n = 6 \) and the following matrix dimensions:

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_1 )</td>
<td>( 30 \times 35 )</td>
</tr>
<tr>
<td>( A_2 )</td>
<td>( 35 \times 15 )</td>
</tr>
<tr>
<td>( A_3 )</td>
<td>( 15 \times 5 )</td>
</tr>
<tr>
<td>( A_4 )</td>
<td>( 5 \times 10 )</td>
</tr>
<tr>
<td>( A_5 )</td>
<td>( 10 \times 20 )</td>
</tr>
<tr>
<td>( A_6 )</td>
<td>( 20 \times 25 )</td>
</tr>
</tbody>
</table>

The tables are rotated so that the main diagonal runs horizontally. Only the main diagonal and upper triangle are used in the \( m \) table, and only the upper triangle is used in the \( s \) table. The minimum number of scalar multiplications to multiply the 6 matrices is \( m[1, 6] = 15,125 \). Of the darker entries, the pairs that have the same shading are taken together in line 9 when computing

\[
m[2, 5] = \min \left\{ m[2, 2] + m[3, 5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000, \right.
\]
\[
m[2, 3] + m[4, 5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \)
\[
m[2, 4] + m[5, 5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375
\]

= 7125.
Step 4: Constructing an optimal solution

```
PRINT-OPTIMAL-PARENS(s, i, j)
1   if i = j
2     then print “A”;
3   else print “(";
4     PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5     PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6     print “)"
```