Design and Analysis of Algorithms

CSE 5311
Lecture 12  Dynamic Programming

Junzhou Huang, Ph.D.
Department of Computer Science and Engineering
Optimization Problems

• In which a set of choices must be made in order to arrive at an optimal (min/max) solution, subject to some constraints. (There may be several solutions to achieve an optimal value.)

• Two common techniques:
  – Dynamic Programming (global)
  – Greedy Algorithms (local)
Dynamic Programming (DP)

• Like divide-and-conquer, solve problem by combining the solutions to sub-problems.

• Differences between divide-and-conquer and DP:
  – Independent sub-problems, solve sub-problems independently and recursively, (so same sub(sub)problems solved repeatedly)
  – DP is applicable when the sub-problems are not independent, i.e. when sub-problems share sub-sub-problems. It solves every sub-sub-problem just once and save the results in a table to avoid duplicated computation.
Application domain of DP

• Optimization problem
  – Find a solution with optimal (maximum or minimum) value.
  – *An* optimal solution, not *the* optimal solution, since may more than one optimal solution, any one is OK.

• Typical steps
  – Characterize the structure of an optimal solution.
  – Recursively define the value of an optimal solution.
  – Compute the value of an optimal solution in a bottom-up fashion.
  – Compute an optimal solution from computed/stored information.
Elements of DP Algorithms

- **Sub-structure**: decompose problem into smaller sub-problems. Express the solution of the original problem in terms of solutions for smaller problems.

- **Table-structure**: Store the answers to the sub-problem in a table, because sub-problem solutions may be used many times.

- **Bottom-up computation**: combine solutions on smaller sub-problems to solve larger sub-problems, and eventually arrive at a solution to the complete problem.
Applicability to Optimization Problems

• **Optimal sub-structure (principle of optimality):** for the global problem to be solved optimally, each sub-problem should be solved optimally. This is often violated due to sub-problem overlaps. Often by being “less optimal” on one problem, we may make a big savings on another sub-problem.

• **Small number of sub-problems:** Many NP-hard problems can be formulated as DP problems, but these formulations are not efficient, because the number of sub-problems is exponentially large. Ideally, the number of sub-problems should be at most a polynomial number.
Optimized Chain Operations

• Determine the optimal sequence for performing a series of operations. (the general class of the problem is important in compiler design for code optimization & in databases for query optimization)

• For example: given a series of matrices: $A_1 \ldots A_n$, we can “parenthesize” this expression however we like, since matrix multiplication is associative (but not commutative).

• Multiply a $p \times q$ matrix $A$ times a $q \times r$ matrix $B$, the result will be a $p \times r$ matrix $C$. (# of columns of $A$ must be equal to # of rows of $B$.)
Matrix Chain-Products

• Dynamic Programming is a general algorithm design paradigm.
  – Rather than give the general structure, let us first give a motivating example:
  – Matrix Chain-Products

• Review: Matrix Multiplication.
  – \( C = A \times B \)
  – \( A \) is \( d \times e \) and \( B \) is \( e \times f \)
  – \( O(d \cdot e \cdot f) \) time

\[
C[i, j] = \sum_{k=0}^{e-1} A[i, k] \times B[k, j]
\]
Matrix Chain-Products

• **Matrix Chain-Product:**
  - Compute $A = A_0 * A_1 * \ldots * A_{n-1}$
  - $A_i$ is $d_i \times d_{i+1}$
  - Problem: How to parenthesize?

• Example
  - B is $3 \times 100$
  - C is $100 \times 5$
  - D is $5 \times 5$
  - $(B*C)*D$ takes $1500 + 75 = 1575$ ops
  - $B*(C*D)$ takes $1500 + 2500 = 4000$ ops
Enumeration Approach

- **Matrix Chain-Product Algorithm.**:
  - Try all possible ways to parenthesize $A = A_0 * A_1 * \ldots * A_{n-1}$
  - Calculate number of ops for each one
  - Pick the one that is best

- **Running time**:
  - The number of parenthesizations is equal to the number of binary trees with $n$ nodes
  - This is exponential!
  - It is called the Catalan number, and it is almost $4^n$.
  - This is a terrible algorithm!
Greedy Approach

• Idea #1: repeatedly select the product that uses the fewest operations.
• Counter-example:
  – A is $101 \times 11$
  – B is $11 \times 9$
  – C is $9 \times 100$
  – D is $100 \times 99$
  – Greedy idea #1 gives $A \times ((B \times C) \times D)$, which takes $109989 + 9900 + 108900 = 228789$ ops
  – $(A \times B) \times (C \times D)$ takes $9999 + 89991 + 89100 = 189090$ ops
• The greedy approach is not giving us the optimal value.
“Recursive” Approach

• Define subproblems:
  – Find the best parenthesization of \(A_i \cdot A_{i+1} \cdots \cdot A_j\).
  – Let \(N_{ij}\) denote the number of operations done by this subproblem.
  – The optimal solution for the whole problem is \(N_{0,n-1}\).

• Subproblem optimality: The optimal solution can be defined in terms of optimal subproblems
  – There has to be a final multiplication (root of the expression tree) for the optimal solution.
  – Say, the final multiplication is at index \(i\): \((A_0 \cdot \ldots \cdot A_i)(A_{i+1} \cdot \ldots \cdot A_{n-1})\).
  – Then the optimal solution \(N_{0,n-1}\) is the sum of two optimal subproblems, \(N_{0,i}\) and \(N_{i+1,n-1}\) plus the time for the last multiplication.
Characterizing Equation

• The global optimal has to be defined in terms of optimal subproblems, depending on where the final multiplication is at.

• Let us consider all possible places for that final multiplication:
  – Recall that $A_i$ is a $d_i \times d_{i+1}$ dimensional matrix.
  – So, a characterizing equation for $N_{i,j}$ is the following:

$$N_{i,j} = \min_{i \leq k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$

• Note that subproblems are not independent—the subproblems overlap.
Subproblem Overlap

Algorithm `RecursiveMatrixChain(S, i, j)`:  
  **Input:** sequence `S` of `n` matrices to be multiplied  
  **Output:** number of operations in an optimal parenthesization of `S`  
  if `i=j`  
    then return 0  
  for `k ← i` to `j` do  
    \[ N_{i,j} \leftarrow \min\{N_{i,j}, \text{RecursiveMatrixChain}(S, i ,k)+\text{RecursiveMatrixChain}(S, k+1,j)+d_i d_{k+1} d_{j+1}\} \]  
  return `N_{i,j}`
Subproblem Overlap
This divide-and-conquer recursive algorithm solves the overlapping problems over and over.

In contrast, DP solves the same (overlapping) subproblems only once (at the first time), then store the result in a table, when the same subproblem is encountered later, just look up the table to get the result.

The computations in green color are replaced by table look up in \textsc{memoized-matrix-chain}(p,1,4).

The divide-and-conquer is better for the problem which generates brand-new problems at each step of recursion.
Dynamic Programming Algorithm

- Since subproblems overlap, we don’t use recursion.
- Instead, we construct optimal subproblems “bottom-up.”
- $N_{i,i}$’s are easy, so start with them.
- Then do problems of “length” 2, 3, … subproblems, and so on.
- Running time: $O(n^3)$

**Algorithm matrixChain(S):**

**Input:** sequence $S$ of $n$ matrices to be multiplied

**Output:** number of operations in an optimal parenthesization of $S$

```
for i ← 1 to n − 1 do
    N_{i,i} ← 0
for b ← 1 to n − 1 do
    \{ $b = j - i$ is the length of the problem \}
    for i ← 0 to n - b - 1 do
        j ← i + b
        N_{i,j} ← +\infty
        for k ← i to j - 1 do
            N_{i,j} ← \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}
return N_{0,n-1}
```
Dynamic Programming Algorithm Visualization

- The bottom-up construction fills in the $N$ array by diagonals.
- $N_{i,j}$ gets values from previous entries in $i$-th row and $j$-th column.
- Filling in each entry in the $N$ table takes $O(n)$ time.
- Total run time: $O(n^3)$.
- Getting actual parenthesization can be done by remembering “k” for each $N$ entry.

$$N_{i,j} = \min_{i\leq k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$
## Dynamic Programming Algorithm Visualization

- $A_0$: 30 X 35; $A_1$: 35 X 15; $A_2$: 15 X 5; $A_3$: 5 X 10; $A_4$: 10 X 20; $A_5$: 20 X 25

### Formula

$$N_{i,j} = \min_{i \leq k < j} \{N_{i,k} + N_{k+1,j} + d_id_{k+1}d_{j+1}\}$$

### Calculations

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15,750</td>
<td>7,875</td>
<td>9,375</td>
<td>11,875</td>
<td>15,125</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2,625</td>
<td>4,375</td>
<td>7,125</td>
<td>10,500</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>750</td>
<td>2,500</td>
<td>5,375</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1,000</td>
<td>3,500</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5,000</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Calculations for $N_{1,4}$**

- $N_{1,1} + N_{2,4} + d_1d_2d_5 = 0 + 2500 + 35 * 15 * 20 = 13000,$
- $N_{1,2} + N_{3,4} + d_1d_3d_5 = 2625 + 1000 + 35 * 5 * 20 = 7125,$
- $N_{1,3} + N_{4,4} + d_1d_4d_5 = 4375 + 0 + 35 * 10 * 20 = 11375$

$N_{1,4} = \min\{13000, 7125, 11375\} = 7125$
Dynamic Programming Algorithm Visualization

\[(A_0 \times (A_1 \times A_2)) \times ((A_3 \times A_4) \times A_5)\]
Assembly-Line Scheduling

- Two parallel assembly lines in a factory, lines 1 and 2
- Each line has \( n \) stations \( S_{i,1} \ldots S_{i,n} \)
- For each \( j \), \( S_{1,j} \) does the same thing as \( S_{2,j} \), but it may take a different amount of assembly time \( a_{i,j} \)
- Transferring away from line \( i \) after stage \( j \) costs \( t_{i,j} \)
- Also entry time \( e_i \) and exit time \( x_i \) at beginning and end
Assembly Line Scheduling (ALS)

Figure 15.1 A manufacturing problem to find the fastest way through a factory. There are two assembly lines, each with $n$ stations; the $j$th station on line $i$ is denoted $S_{i,j}$ and the assembly time at that station is $a_{i,j}$. An automobile chassis enters the factory, and goes onto line $i$ (where $i = 1$ or 2), taking $e_i$ time. After going through the $j$th station on a line, the chassis goes on to the $(j + 1)$st station on either line. There is no transfer cost if it stays on the same line, but it takes time $t_{i,j}$ to transfer to the other line after station $S_{i,j}$. After exiting the $n$th station on a line, it takes $x_i$ time for the completed auto to exit the factory. The problem is to determine which stations to choose from line 1 and which to choose from line 2 in order to minimize the total time through the factory for one auto.
Concrete Instance of ALS

Figure 15.2  (a) An instance of the assembly-line problem with costs $e_i$, $a_{i,j}$, $t_i$, $t_j$, and $x_i$ indicated. The heavily shaded path indicates the fastest way through the factory. (b) The values of $f_1[j]$, $f^*$, $l_1[j]$, and $l^*$ for the instance in part (a).
**Brute Force Solution**

- List all possible sequences,
- For each sequence of \( n \) stations, compute the passing time. 
  (the computation takes \( \Theta(n) \) time.)
- Record the sequence with smaller passing time.
- However, there are total \( 2^n \) possible sequences.
ALS --DP steps: Step 1

- Step 1: find the structure of the fastest way through factory
  - Consider the fastest way from starting point through station $S_{1,j}$ (same for $S_{2,j}$)
    - $j=1$, only one possibility
    - $j=2,3,\ldots,n$, two possibilities: from $S_{1,j-1}$ or $S_{2,j-1}$
      - from $S_{1,j-1}$, additional time $a_{1,j}$
      - from $S_{2,j-1}$, additional time $t_{2,j-1} + a_{1,j}$
    - suppose the fastest way through $S_{1,j}$ is through $S_{1,j-1}$, then the chassis must have taken a fastest way from starting point through $S_{1,j-1}$. Why???
    - Similarly for $S_{2,j-1}$.
DP step 1: Find Optimal Structure

- An optimal solution to a problem contains within it an optimal solution to subproblems.
- The fastest way through station $S_{i,j}$ contains within it the fastest way through station $S_{1,j-1}$ or $S_{2,j-1}$.
- Thus can construct an optimal solution to a problem from the optimal solutions to subproblems.
ALS --DP steps: Step 2

• Step 2: A recursive solution
• Let \( f_i[j] \) \((i=1,2\) and \(j=1,2,\ldots, n)\) denote the fastest possible time to get a chassis from starting point through \( S_{i,j} \).
• Let \( f^* \) denote the fastest time for a chassis all the way through the factory. Then
• \( f^* = \min(f_1[n] + x_1, f_2[n] + x_2) \)
• \( f_1[1]=e_1+a_{1,1} \), fastest time to get through \( S_{1,1} \)
• \( f_1[j]=\min(f_1[j-1]+a_{1,j}, f_2[j-1]+ t_{2,j-1} + a_{1,j}) \)
• Similarly to \( f_2[j] \).
ALS --DP steps: Step 2

- Recursive solution:
  - $f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$
  - $f_1[j] = e_1 + a_{1,1}$ if $j = 1$
  - $f_1[j] = \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j})$ if $j > 1$
  - $f_2[j] = e_2 + a_{2,1}$ if $j = 1$
  - $f_2[j] = \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j})$ if $j > 1$

- $f_i[j]$ ($i = 1, 2; j = 1, 2, \ldots, n$) records optimal values to the subproblems.

- To keep track of the fastest way, introduce $l_i[j]$ to record the line number (1 or 2), whose station $j-1$ is used in a fastest way through $S_{ij}$.

- Introduce $l^*$ to be the line whose station $n$ is used in a fastest way through the factory.
ALS --DP steps: Step 3

- Step 3: Computing the fastest time
  - One option: a recursive algorithm.
    - Let $r_i(j)$ be the number of references made to $f_i[j]$
      - $r_1(n) = r_2(n) = 1$
      - $r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1)$
      - $r_i(j) = 2^{n-j}$.
      - So $f_1[1]$ is referred to $2^{n-1}$ times.
      - Total references to all $f_i[j]$ is $\Theta(2^n)$.
    - Thus, the running time is exponential.
  - Non-recursive algorithm.
ALS FAST-WAY Algorithm

FASTEST-WAY \((a, t, e, x, n)\)
\[
\begin{align*}
1 & \quad f_1[1] \leftarrow e_1 + a_{1,1} \\
2 & \quad f_2[1] \leftarrow e_2 + a_{2,1} \\
3 & \quad \textbf{for } j \leftarrow 2 \textbf{ to } n \\
4 & \quad \quad \textbf{do if } f_1[j - 1] + a_{1,j} \leq f_2[j - 1] + t_{2,j-1} + a_{1,j} \\
5 & \quad \quad \quad \textbf{then } f_1[j] \leftarrow f_1[j - 1] + a_{1,j} \\
6 & \quad \quad \quad \quad l_1[j] \leftarrow 1 \\
7 & \quad \quad \quad \textbf{else } f_1[j] \leftarrow f_2[j - 1] + t_{2,j-1} + a_{1,j} \\
8 & \quad \quad \quad \quad l_1[j] \leftarrow 2 \\
9 & \quad \quad \textbf{if } f_2[j - 1] + a_{2,j} \leq f_1[j - 1] + t_{1,j-1} + a_{2,j} \\
10 & \quad \quad \quad \textbf{then } f_2[j] \leftarrow f_2[j - 1] + a_{2,j} \\
11 & \quad \quad \quad \quad l_2[j] \leftarrow 2 \\
12 & \quad \quad \quad \textbf{else } f_2[j] \leftarrow f_1[j - 1] + t_{1,j-1} + a_{2,j} \\
13 & \quad \quad \quad \quad l_2[j] \leftarrow 1 \\
14 & \quad \textbf{if } f_1[n] + x_1 \leq f_2[n] + x_2 \\
15 & \quad \quad \textbf{then } f^* = f_1[n] + x_1 \\
16 & \quad \quad \quad l^* = 1 \\
17 & \quad \quad \textbf{else } f^* = f_2[n] + x_2 \\
18 & \quad \quad \quad l^* = 2
\end{align*}
\]

Running time: \(O(n)\).
ALS --DP steps: Step 4

- **Step 4:** Construct the fastest way through the factory

```plaintext
PRINT-STATIONS (l, n)
1   i ← l*
2   print "line " i " station " n
3   for j ← n downto 2
4       do i ← l_i[j]
5           print "line " i " station " j - 1
```
Optimal Substructure Varies in Two Ways

• How many subproblems
  – In assembly-line schedule, one subproblem
  – In matrix-chain multiplication: two subproblems

• How many choices
  – In assembly-line schedule, two choices
  – In matrix-chain multiplication: $j-i$ choices

• DP solve the problem in bottom-up manner.
Running Time for DP Programs

• #overall subproblems × #choices.
  – In assembly-line scheduling, $O(n) \times O(1) = O(n)$.
  – In matrix-chain multiplication, $O(n^2) \times O(n) = O(n^3)$

• The cost = costs of solving subproblems + cost of making choice.
  – In assembly-line scheduling, choice cost is
    $a_{ij}$ if stay in the same line, $t_{i';j-1} + a_{ij}$ ($i' \neq i$) otherwise.
  – In matrix-chain multiplication, choice cost is $p_{i-1}p_k p_j$. 