Dynamic Programming
Outline and Reading

- Matrix Chain-Product (§5.3.1)
- The General Technique (§5.3.2)
- 0-1 Knapsack Problem (§5.3.3)
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 $e \times f$

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

- $O(d^f)$ time
Matrix Chain-Products

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

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

Matrix Chain-Product Alg.:

- 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 $n$ leaf nodes (Why?)
  - This is exponential! It is called the Catalan number, and it is almost $4^n$.
- This is a terrible algorithm!
A Greedy Approach

Idea #1: repeatedly select the product that uses (up) the most operations.

Counter-example:

- A is 10 \times 5
- B is 5 \times 10
- C is 10 \times 5
- D is 5 \times 10

Greedy idea #1 gives \((A \times B) \times (C \times D)\), which takes
\(500 + 1000 + 500 = 2000\) ops

\(A \times ((B \times C) \times D)\) takes \(500 + 250 + 250 = 1000\) ops
Another Greedy Approach

Idea #2: 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 #2 gives $A * ((B * C) * D))$, which takes $109989 + 9900 + 108900 = 228789$ ops
- $(A * B) * (C * D)$ takes $9999 + 89991 + 89100 = 189090$ ops

The greedy approach is not giving us the optimal value.
A “Recursive” Approach

Define **subproblems**:
- Find the best parenthesization of $A_i A_{i+1} \cdots A_j$.
- Let $N_{i,j}$ 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 multiply is at index $i$: $(A_0 \cdots A_i)(A_{i+1} \cdots A_{n-1})$.
- 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 multiply.
- If the global optimum did not have these optimal subproblems, we could define an even better “optimal” solution.
A Characterizing Equation

- The global optimal has to be defined in terms of optimal subproblems, depending on the position of the final multiply.
- Let us consider all possible places for that final multiply:
  - Assume 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.
A 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 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 \( S \)

```plaintext
for i ← 1 to n-1 do
  \( N_{i,i} \) ← 0
for b ← 1 to n-1 do
  for i ← 0 to n-b-1 do
    j ← i+b
    \( N_{i,j} \) ← +infinity
    for k ← i to j-1 do
      \( N_{i,j} \) ← min{\( N_{i,j} \), \( N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \)}
```
The bottom-up construction fills in the \(N\) array by diagonals.

\(N_{i,j}\) gets values from pervious 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.

\[
N_{i,j} = \min_{i \leq k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}
\]
Determining the Run Time

It’s not easy to see that filling in each square takes $O(n)$ time. So, let’s look at this another way

$$n + 2(n - 1) + 4(n - 2) + \sum_{i=0}^{n-1} 2i(n-i) + 2(n - 1)(n - (n - 1))$$

$$= \sum_{i=0}^{n-1} 2i(n-i) = \sum_{i=0}^{n-1} 2in - \sum_{i=0}^{n-1} 2i^2 =$$

$$= 2n \frac{(n-1)n}{2} - 2 \frac{(n-1)n(2(n-1)+1)}{6}$$

$$= n(n-1)\left(n - \frac{2n-1}{6}\right) = n(n-1)\left(\frac{4n+1}{6}\right)$$
The General Dynamic Programming Technique

Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:

- **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as \( j, k, l, m, \) and so on.
- **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
- **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).