Dynamic Programming

Matrix Chain-Products

**Matrix Chain-Product:**
- Compute $A = A_0 A_1 \cdots A_{n-1}$
- $A_i$ 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

Outline and Reading

- Matrix Chain-Product (§5.3.1)
- The General Technique (§5.3.2)
- 0-1 Knapsack Problem (§5.3.3)

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 $(A^B)^*(C^D)$, which takes $500+1000+500 = 2000$ ops
- $A^*(B^C)^D$ takes $500+250+250 = 1000$ ops

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.
  - $C = A^B$
  - $A$ is $d \times e$ and $B$ is $e \times f$
  - $O(d \times e \times f)$ time

$$C[i, j] = \sum_{k=i}^{j} a[i, k] * b[k, j]$$

Enumeration Approach

**Matrix Chain-Product Alg.:**
- Try all possible ways to parenthesize $A = A_0 A_1 \cdots A_{n-1}$
- 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!
Another Greedy Approach

Idea #2: repeatedly select the product that uses the fewest operations.

Counter-example:

A is 101 × 11
B is 11 × 9
C is 9 × 100
D is 100 × 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.

"Recursive" Approach

Define subproblems:
- Find the best parenthesization of $A_1 \times A_2 \times \ldots \times A_n$
- Let $N_{i,j}$ denote the number of operations done by this subproblem.

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 $i$: $(A_1 \times \ldots \times A_i) \times (A_{i+1} \times \ldots \times A_n)$.
- Then the optimal solution $N_{i,j}$ is the sum of two optimal subproblems, $N_{i,k}$ and $N_{k+1,j}$ 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.

Characterizing Equation

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

Let us consider all possible places for that final multiply:
- 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.

Dynamic Programming Algorithm

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

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).
The 0/1 Knapsack Problem
- Given: A set S of n items, with each item i having:
  - \( w_i \) - a positive weight
  - \( b_i \) - a positive benefit
- Goal: Choose items with maximum total benefit but with weight at most W.
- If we are not allowed to take fractional amounts, then this is the 0/1 knapsack problem.
- In this case, we let \( T \) denote the set of items we take.
- Objective: maximize \( \sum_{i \in T} b_i \)
- Constraint: \( \sum_{i \in T} w_i \leq W \)

A 0/1 Knapsack Algorithm, Second Attempt
- \( S_k \): Set of items numbered 1 to k.
- Define \( B[k, w] \) to be the best selection from \( S_k \) with weight at most w.
- Good news: this does have subproblem optimality.
- \( B[k, w] = \begin{cases} \text{max} \{ B[k-1, w], B[k-1, w - w_k] + b_k \} & \text{if } w_k \leq w \\ B[k-1, w] & \text{else} \end{cases} \)
- I.e., the best subset of \( S_k \) with weight at most w is either:
  - the best subset of \( S_{k-1} \) with weight at most w or
  - the best subset of \( S_{k-1} \) with weight at most \( w - w_k \) plus item k.

Example
- Given: A set S of n items, with each item i having:
  - \( b_i \) - a positive "benefit"
  - \( w_i \) - a positive "weight"
- Goal: Choose items with maximum total benefit but with weight at most W.

0/1 Knapsack Algorithm
- \( B[k, w] = \begin{cases} \text{max} \{ B[k-1, w], B[k-1, w - w_k] + b_k \} & \text{if } w_k \leq w \\ B[k-1, w] & \text{else} \end{cases} \)
- Recall the definition of \( B[k, w] \)
- Since \( B[k, w] \) is defined in terms of \( B[k-1, w] \), we can use two arrays of instead of a matrix.
- Running time: \( O(nW) \).
- Not a polynomial-time algorithm since W may be large.
- This is a pseudo-polynomial time algorithm.

A 0/1 Knapsack Algorithm, First Attempt
- \( S_k \): Set of items numbered 1 to k.
- Define \( B[k] \) = best selection from \( S_k \).
- Problem: does not have subproblem optimality:
  - Consider set \( S = \{(3,2),(5,4),(8,5),(4,3),(10,9)\} \) of (benefit, weight) pairs and total weight \( W = 20 \)
  - Best for \( S_1 \):
  - Best for \( S_2 \):