Dynamic Programming

Outline and Reading

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

Matrix Chain Product

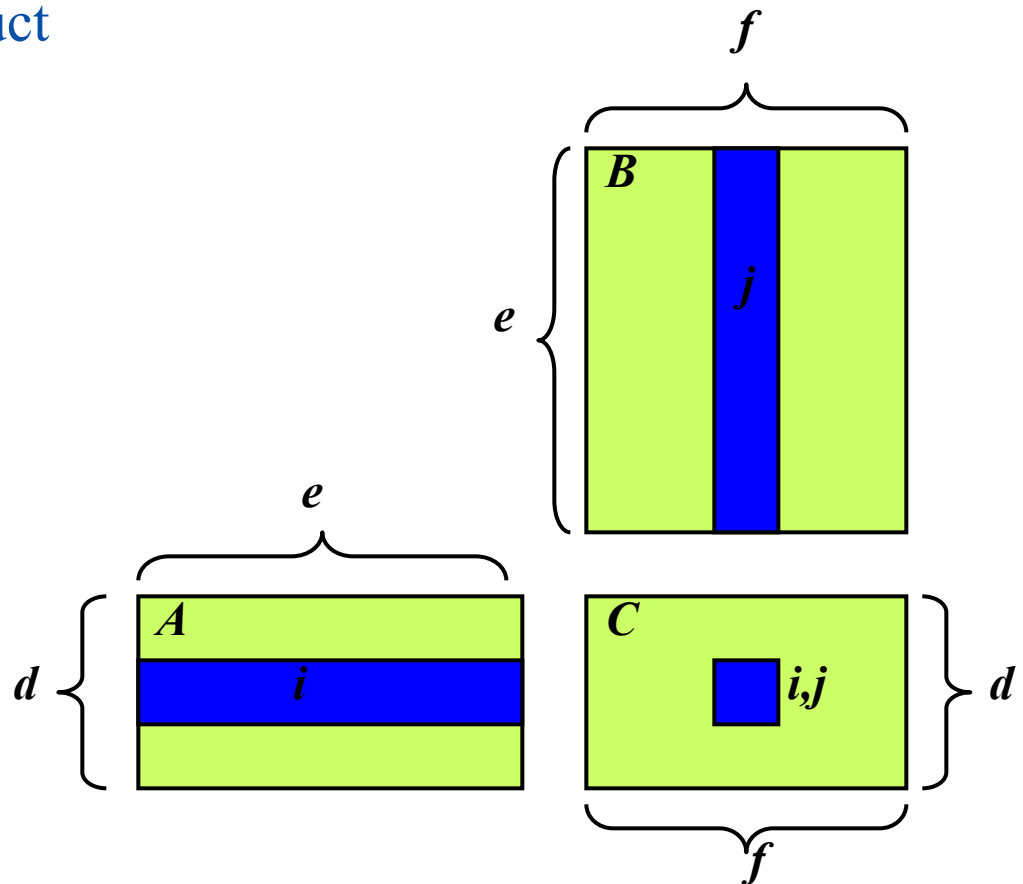Dynamic Programming is a general algorithm design paradigm.

- Rather than give the general structure, we first give a motivating example: Matrix Chain-Product

Review: Matrix Multiplication

- $C = A*B$

- A is $d \times e$ and B is $e \times f$

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

- $O(d \cdot e \cdot f)$ time

Matrix Chain Product

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 in such a way that **minimizes** the total number of scalar multiplications?

Example:

- B is 3×100
- C is 100×5
- D is 5×5
- (B*C)*D takes $1500 + 75 = 1575$ ops
- B*(C*D) takes $1500 + 2500 = 4000$ ops

One Approach: Brute Force

- Try all possible ways to parenthesize $A = A_0 * A_1 * \ldots * A_{n-1}$
- Calculate number of operations 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 Approach: Greedy (v1)

Idea: Repeatedly select the product that uses (up) the most operations.

Counter-example:

- A is 10×5
- B is 5×10
- C is 10×5
- D is 5×10

This greedy approach gives (A*B)*(C*D)

- takes 500+1000+500 = 2000 ops

A better solution: A*((B*C)*D)

- takes 500+250+250 = 1000 ops

Another Approach: Greedy (v2)

Idea: 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

This greedy approach gives A*((B*C)*D))
- takes 109989+9900+108900=228789 ops

A better solution is (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 $A_i*A_{i+1}*\ldots*A_j$.
- 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: $(A_0*\ldots*A_i)*(A_{i+1}*\ldots*A_{n-1})$.
- 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.

Characterizing Equation

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

- 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:

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

- Note that subproblems are not independent – meaning subproblems overlap.

Dynamic Programming Algorithm Visualization

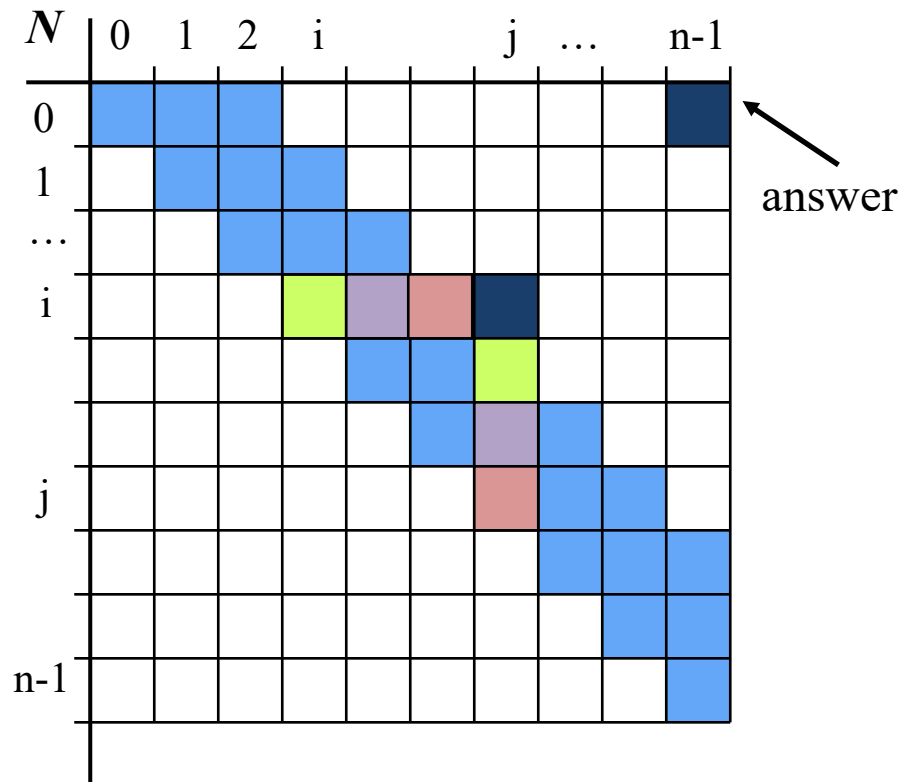The bottom-up construction fills in the N array by diagonals

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

$N_{i,j}$ 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 in a separate table

answer

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 $i \leftarrow 0$ **to** $n-1$ **do**

    $N_{i,i} \leftarrow 0$

  for *length* $\leftarrow 1$ **to** $n-1$ **do**

    { *length* $= j - i$ is the length of the chain }

    for $i \leftarrow 0$ **to** $n - 1 - length$ **do**

      $j \leftarrow i + length$

      $N_{i,j} \leftarrow +\infty$

      for $k \leftarrow i$ **to** $j - 1$ **do**

        $N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i\, d_{k+1}\, d_{j+1}\}$

        record k that produces minimum $N_{i,j}$

  return $N_{0,n-1}$

General
Dynamic Programming Technique

Applies to an optimization 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 overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

- **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems

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.

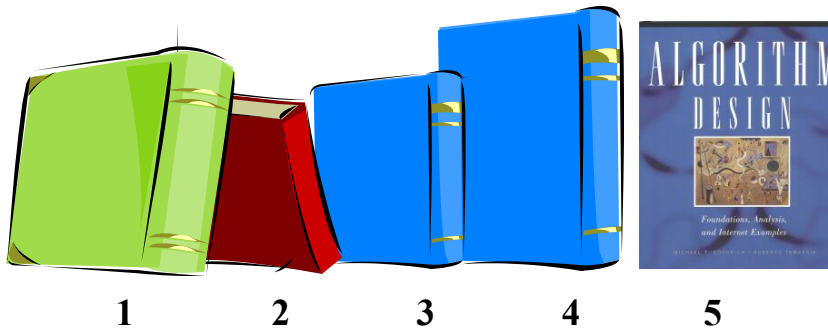- In this case, we let *T* denote the set of items we take
- Objective: maximize

$$\sum_{i \in T} b_i$$

- Constraint:

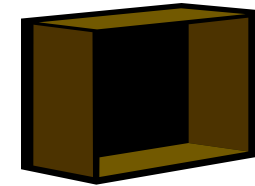$$\sum_{i \in T} w_i \le W$$

Example

- <u>Given</u>: A set S of n items, with each item i having
  - b_i - a positive "benefit"
  - w_i - a positive "weight"
- <u>Goal</u>: Choose items with maximum total benefit but with weight at most W.

Items:

	1	**2**	**3**	**4**	**5**
Weight:	4 in	2 in	2 in	6 in	2 in
Benefit:	$20	$3	$6	$25	$80

"knapsack"

box of width 9 in

Solution:
- item 5 ($80, 2 in)
- item 3 ($6, 2in)
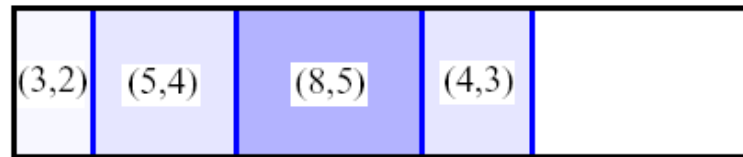- item 1 ($20, 4in)

A 0/1 Knapsack Algorithm: First Attempt
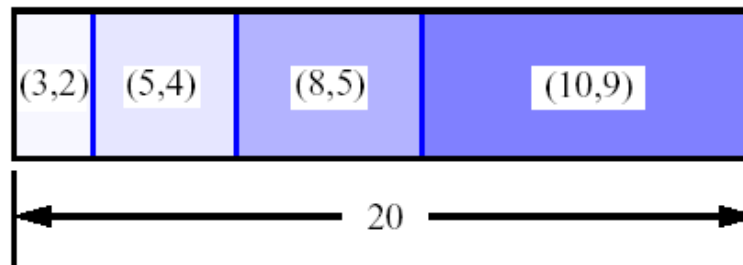
S_k: Set of items numbered 1 to k.

- <u>Idea</u>: 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_4:

(3,2)	(5,4)	(8,5)	(4,3)	

Best for S_5:

(3,2)	(5,4)	(8,5)	(10,9)

← 20 →

A 0/1 Knapsack Algorithm: Second Attempt

S_k: Set of items numbered 1 to k.

- <u>Idea</u>: Define B[k,w] to be the best selection from S_k with weight at most w
- Good news: this does have subproblem optimality.

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\, B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

That is, 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

0/1 Knapsack Algorithm

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\, B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- Recall the definition of B[k,w]

- Since B[k,w] is defined in terms of B[k–1,*], 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

Algorithm *01Knapsack(S, W)*:
> **Input:** set *S* of *n* items with benefit b_i and weight w_i; maximum weight *W*
>
> **Output:** benefit of best subset of *S* with weight at most *W*
>
> let *A* and *B* be arrays of length *W* + 1
>
> **for** *w* ← 0 **to** *W* **do**
>> *B[w]* ← 0
>
> **for** *k* ← 1 **to** *n* **do**
>> copy array *B* into array *A*
>>
>> **for** *w* ← w_k **to** *W* **do**
>>> **if** $A[w-w_k] + b_k > A[w]$ **then**
>>>> $B[w] ← A[w-w_k] + b_k$
>
> **return** *B[W]*