CS 421: Design and Analysis of Algorithms

Chapter 15: Dynamic Programming

Dr. Gaby Dagher
Department of Computer Science
Boise State University

March 8, 2022

[Part of the lecture notes is based on materials by David Luebke]

Content of this Chapter

- ❑ Introduction to dynamic programming
- ❑ Rod cutting
- ❑ Matrix-chain multiplication
- ❑ Elements of dynamic programming
- ❑ 0-1 Knapsack Problem

Content of this Chapter

- **Introduction to dynamic programming**
- Rod cutting
- Matrix-chain multiplication
- Elements of dynamic programming
- 0-1 Knapsack Problem

Dynamic Programming

- An algorithm design technique (similar to divide-and-conquer)
- Divide and conquer
  - Partition the problem into *independent* subproblems.
  - Solve the subproblems recursively.
  - Combine the solutions to solve the original problem.
- Observations about Merge Sort:
  - One decision: split the input in half
  - The same decision is repeated
  - Subproblems do not share subsubproblems.

Dynamic Programming

- *Dynamic programming* is applicable when subproblems are **not** independent: *Subproblems share subsubproblems*.

- Used for **optimization problems**
  - A set of choices must be made to get an optimal solution.
  - Find a solution with the optimal value (minimum or maximum).
  - There may be many solutions that lead to an optimal value.
  - Our goal: **find *an* optimal solution.**

Steps of Dynamic Programming Algorithm

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. (not always required)

Content of this Chapter

- Introduction to dynamic programming
- **Rod cutting**
- Matrix-chain multiplication
- Elements of dynamic programming
- 0-1 Knapsack Problem

The Rod Cutting Problem

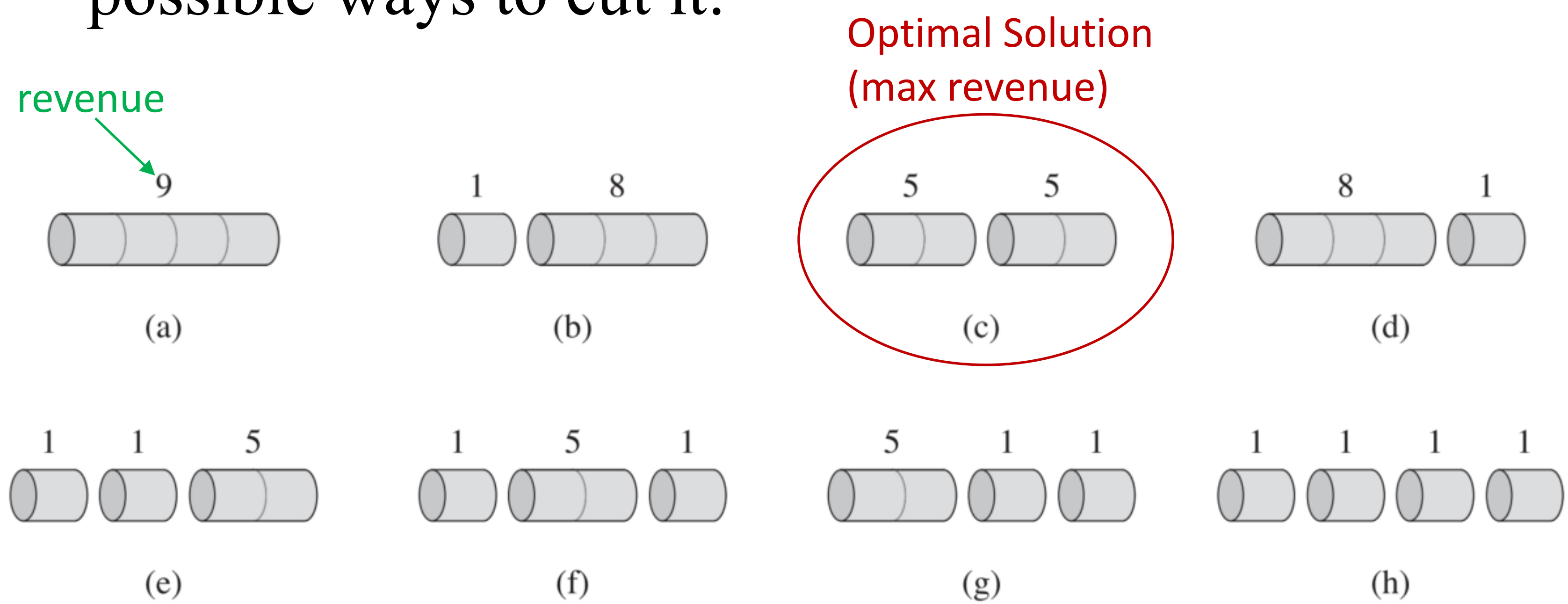
Given a rod of length n inches and a table of prices, determine the maximum revenue obtainable by cutting up the rod and selling the pieces.

Example of a price table:

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Possible Cuts

- Possible ways to cut a rod of length n is 2^{n-1}.
- For example, given a rod of length 4, there are $2^{4-1} = 8$ possible ways to cut it:



Optimal Revenue

- If an *optimal solution* cuts the rod into k pieces, for some $1 \le k \le n$,

- Then an optimal decomposition of the rod into pieces of lengths $i_1, i_2, \ldots, i_k$ provides *maximum corresponding revenue*:

$$\underset{\text{Max Revenue}}{r_n} = \underbrace{p_{i_1} + p_{i_2} + \cdots + p_{i_k}}_{\text{Optimal Decomposition}}$$

Example – Rod Cutting

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Max Revenue

Length n

Optimal Solution

$r_1 = 1$ from solution $1 = 1$ (no cuts), $k = 1$

$r_2 = 5$ from solution $2 = 2$ (no cuts), $k = 1$

$r_3 = 8$ from solution $3 = 3$ (no cuts), $k = 1$

$r_4 = 10$ from solution $4 = 2 + 2$, $k = 2$

$r_5 = 13$ from solution $5 = 2 + 3$, $k = 2$

$r_6 = 17$ from solution $6 = 6$ (no cuts), $k = 1$

$r_7 = 18$ from solution $7 = 1 + 6$ or $7 = 2 + 2 + 3$, $k = 2$ $k = 3$

$r_8 = 22$ from solution $8 = 2 + 6$, $k = 2$

$r_9 = 25$ from solution $9 = 3 + 6$, $k = 2$

$r_{10} = 30$ from solution $10 = 10$ (no cuts). $k = 1$

Optimal Revenue

- The optimal (maximum) solution must start the cutting at position i where $1 \leq i \leq n$.
- Since we don't know ahead of time which value of i optimizes revenue, we have to consider all possible values for i and pick the one that maximizes revenue.
- The optimal revenue r_n for a rod of length n is:

$$r_n = \max\left(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\right) .$$

Optimal Substructure

- *Optimal substructure* property:
  - **Optimal solutions to a problem incorporate optimal solutions to related subproblems, which <u>we may solve independently</u>.**
- The rod-cutting problem exhibits *optimal substructure* property:
  - The overall optimal solution incorporates optimal solutions to the two related subproblems, maximizing revenue from each of those two pieces.

Optimal Revenue

- **<u>Recall</u>**: The optimal revenue r_n for a rod of length n based on the ***first cut*** is:

$$r_n = \max\left(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\right) \qquad \textbf{(1)}$$

- **Optimal revenue based on the *<u>leftmost cut</u>*:**
  Let i be the first cut from left (*leftmost cut*), then the above equation can be simplified as follows:

Max revenue of remaining piece of length *n-i*

$$r_n = \max_{1 \le i \le n}\left(p_i + r_{n-i}\right) \qquad \textbf{(2)}$$

Price of first left piece of length *i*

Optimal Revenue

We can re-write the recurrences in (1) and (2) as follows:

- **Optimal revenue based on the *first cut*:**

$$r[n] = \begin{cases} p_1 & \text{if } n=1 \\ \max\left(\max\limits_{1 \le i \le n-1} (r[i] + r[n-i]),\ p_n\right) & \text{if } n>1 \end{cases}$$

- **Optimal revenue based on the *leftmost cut*:**

$$r[n] = \begin{cases} p_1 & \text{if } n=1 \\ \max\limits_{1 \le i \le n} (p_i + r[n-i]) & \text{if } n>1 \end{cases}$$

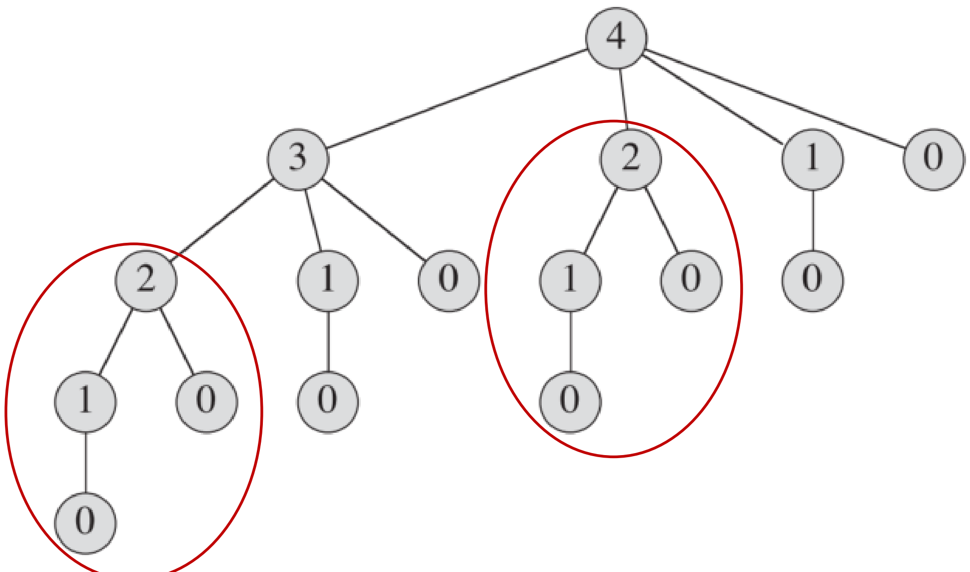
Recursive Top-Down Rod Cutting Algorithm

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

$T(n) = \Theta(2^n)$

Why top-down is inefficient?

The recursion tree (recursive calls) for a rod of length: $n = 4$



Problem: Recursive calls to solve similar sub-problems.

Dynamic-Programming Approaches

- Approach#1: Top-down with ***memoization***
  - We write the procedure recursively in a natural manner, but modified to *save the result of each subproblem* (e.g. in an array or a table).

- Approach#2: Bottom-up
  - We sort the subproblems by size and *solve them in size order, smallest first*.
  - When solving a subproblem, we have already solved (and saved) all of the smaller subproblems its solution depends upon.

Dynamic-Programming Approaches

- Both approaches usually yield algorithms with the same asymptotic running time.

- The *bottom-up* approach often has much better constant factors, since it has less overhead for procedure calls.

Approach#1: Top-down with memoization

```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

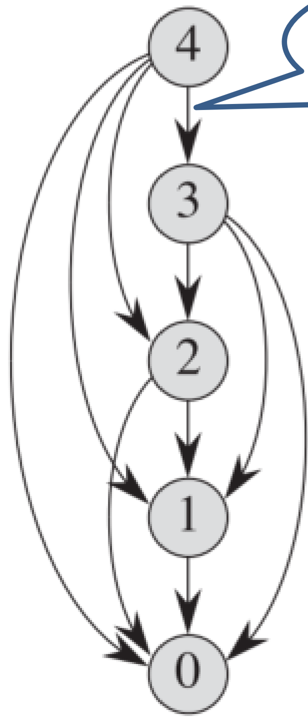
Approach#2: Bottom-up approach

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

$T(n) = \Theta(n^2)$

Subproblem Graphs

The subproblem graph (collapsed tree) for a rod of length: $n = 4$



This edge indicates that the solution for sub-problem with input *n=3* is required to solve sub-problem with input *n=4*

Extended bottom-up method

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        if q < p[i] + r[j - i]
            q = p[i] + r[j - i]
            s[j] = i
    r[j] = q
return r and s
```

Extended bottom-up method

```
PRINT-CUT-ROD-SOLUTION(p, n)
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]
    n = n - s[n]
```

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

- PRINT-CUT-ROD-SOLUTION (p, 10) → 10
- PRINT-CUT-ROD-SOLUTION (p, 7) → 1,6
- PRINT-CUT-ROD-SOLUTION (p, 5) → 2,3

Content of this Chapter

- ❑ Introduction to dynamic programming
- ❑ Rod cutting
- ➢ **Matrix-chain multiplication**
- ❑ Elements of dynamic programming
- ❑ 0-1 Knapsack Problem

Matrix-chain Multiplication

- Suppose we have a sequence or chain $A_1, A_2, \ldots, A_n$ of n matrices (*not necessarily square matrices*) to be multiplied.
  - That is, we want to compute the product: $\mathbf{A_1} \times \mathbf{A_2} \times \ldots \times \mathbf{A_n}$ in the most efficient way.

Matrix-chain Multiplication

- To compute the number of scalar multiplications necessary, we must know:
  1. Algorithm to multiply two matrices
  2. Matrix dimensions

Algorithm to Multiply 2 Matrices

Input: Matrices $A_{p\times q}$ and $B_{q\times r}$ (with dimensions $p\times q$ and $q\times r$)
Result: Matrix $C_{p\times r}$ resulting from the product $A\cdot B$

MATRIX-MULTIPLY$(A_{p\times q}, B_{q\times r})$

1. **for** $i \leftarrow 1$ **to** p
2.   **for** $j \leftarrow 1$ **to** r
3. $C[i, j] \leftarrow 0$
4.   **for** $k \leftarrow 1$ **to** q
5. $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
6. **return** C

Scalar multiplication in line 5 dominates time to compute C_{pxr}
Number of scalar multiplications = $p.q.r$

Matrix-chain Multiplication

- **Observation**: There are several possible ways (parenthesizations) to compute the product.
- Example: consider the chain A_1, A_2, A_3, A_4 of 4 matrices. Let us try to compute the product $A_1A_2A_3A_4$
- There are 5 possible ways:

1. $(A_1(A_2(A_3A_4)))$
2. $(A_1((A_2A_3)A_4))$
3. $((A_1A_2)(A_3A_4))$
4. $((A_1(A_2A_3))A_4)$
5. $(((A_1A_2)A_3)A_4)$

Q: Does it matter?
A: Yes.

Matrix-chain Multiplication

Why Parenthesizations Matter?

- Example: Consider three matrices $A_{10\times 100}$, $B_{100\times 5}$, and $C_{5\times 50}$
- There are 2 ways to parenthesize:
  - $((AB)C) = D_{10\times 5} \cdot C_{5\times 50}$
    - $AB \Rightarrow 10\cdot 100\cdot 5 = 5{,}000$ scalar multiplications
    - $DC \Rightarrow 10\cdot 5\cdot 50 = 2{,}500$ scalar multiplications

    Total: 7,500
  - $(A(BC)) = A_{10\times 100} \cdot E_{100\times 50}$
    - $BC \Rightarrow 100\cdot 5\cdot 50=25{,}000$ scalar multiplications
    - $AE \Rightarrow 10\cdot 100\cdot 50 =50{,}000$ scalar multiplications

    Total: 75,000

Matrix-chain Multiplication Problem

- Given a chain $A_1, A_2, \ldots, A_n$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$:
  - Parenthesize the product $A_1A_2 \ldots A_n$ such that the total number of scalar multiplications is ***minimized***.

Matrix-chain Multiplication

- **Observation#1**: Any parenthesization splits the chain of matrices into two sub-chains, each of which can be parenthesized separately.
- Example: All possible parenthesizations for $A_1A_2A_3A_4$ are:

1. $(A_1(A_2(A_3A_4))) \Rightarrow (A_1)((A_2)(A_3A_4))$
2. $(A_1((A_2A_3)A_4)) \Rightarrow (A_1)((A_2A_3)(A_4))$
3. $((A_1A_2)(A_3A_4)) \Rightarrow (A_1A_2)(A_3A_4)$
4. $((A_1(A_2A_3))A_4) \Rightarrow ((A_1)(A_2A_3))(A_4)$
5. $(((A_1A_2)A_3)A_4) \Rightarrow ((A_1A_2)(A_3))(A_4)$

Matrix-chain Multiplication

- **Observation#2**: Unlike the rod-cutting problem, the cost of subproblems with the same size differs from one subproblem to another.

- Example: Consider $A_{10\times100}$, $B_{100\times5}$, and $C_{5\times50}$
  - (AB) and (BC) are two subproblems of size 2.
  - AB $\Rightarrow$ $10\cdot100\cdot5 = 5{,}000$ scalar multiplications
  - BC $\Rightarrow$ $100\cdot5\cdot50=25{,}000$ scalar multiplications

First Attempt: Brute-force Approach

- Let *P(n)* be the *number of alternative parenthesizations* of a sequence of n matrices.

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

- Brute force method of exhaustive search takes time exponential in n: $\Omega(2^n)$

Second Attempt: Dynamic Programming

Recall Dynamic Programming Approach:

Step1: Characterize the structure of an optimal solution.

Step2: Recursively define the value of an optimal solution.

Step3: Compute the value of an optimal solution.

Step4: Construct an optimal solution from computed information.

Step1: The structure of an optimal solution

- **Notation**: Let us use the notation $A_{i..j}$ for the matrix that results from the product $A_i\ A_{i+1}\ \ldots\ A_j$
- An optimal parenthesization of the product $A_1A_2\ldots A_n$ splits the product between A_k and A_{k+1} for some integer k where $1 \le k < n$
- First compute matrices $A_{1..k}$ and $A_{k+1..n}$; then multiply them to get the final matrix $A_{1..n}$

Step1: The structure of an optimal solution

- **Key observation**: parenthesizations of the subchains $A_1A_2\ldots A_k$ and $A_{k+1}A_{k+2}\ldots A_n$ must also be optimal if the parenthesization of the chain $A_1A_2\ldots A_n$ is optimal.
- That is, *the optimal solution to the problem contains within it the optimal solution to subproblems.*
- Why? Use a "cut-and-paste" argument

Optimal Substructure Property

- Let S be a parenthesization of the product $A_1A_2\ldots A_n$ that splits the product between A_k and A_{k+1} for some integer k where1 $\leq k < n$: $A_{1..k} \cdot A_{k+1..n}$
- If S is an optimal parenthesization (total number of scalar multiplications is minimized) of the product $A_1A_2\ldots A_n$, then the parenthesizations S_1 and S_2 of $A_{1..k}$ and $A_{k+1..n}$, respectively, must also be optimal.
- Why? "Cut-and-Paste" Argument:
  - If we could find a parenthesization S'_1 of $A_{1..k}$ that requires less scalar multiplications than S_1, then we could construct a solution S' consisting of S'_1 and S_2 that would require less scalar multiplications than S ➔ contradicting the optimality of S.
  - The same *cut-and-paste* argument applies to $A_{k+1..n}$ where we reach a contradiction if there exists S'_2 of $A_{k+1..n}$ that requires less scalar multiplications than S_2 .

Step2: A recursive solution

- Let $m[\ i, j\ ]$ be the minimum number of scalar multiplications necessary to compute $A_{i..j}$
- Minimum cost to compute $A_{1..n}$ is $m[1, n]$
- Suppose the optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1} for some integer k where $i \le k < j$, *then:*

Step2: A recursive solution

- $A_{i..j} = (A_i A_{i+1} \ldots A_k) \cdot (A_{k+1} A_{k+2} \ldots A_j) = A_{i..k} \cdot A_{k+1..j}$
- Cost of computing $A_{i..j}$ = cost of computing $A_{i..k}$ + cost of computing $A_{k+1..j}$ + cost of multiplying $A_{i..k}$ and $A_{k+1..j}$
- Cost of multiplying $A_{i..k}$ and $A_{k+1..j}$ is $p_{i-1} p_k p_j$
- $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$ *where:* $i \le k < j$.
- $m[i, i] = 0$ for $i = 1, 2, \ldots, n$

Step2: A recursive solution

- But… optimal parenthesization occurs at one value of *k* among all possible $i \le k < j$
- Check all these and select the best one

$$m[i,j] = \begin{cases} 0 & \text{if } i=j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i<j \end{cases}$$

Step2: A recursive solution

- To keep track of how to construct an optimal solution, we use a table *s*
- $s[i, j]$ = value of k at which $A_i A_{i+1} \ldots A_j$ is split for optimal parenthesization

Step3: Computing the optimal costs

- Algorithm:
  - First computes costs for chains of length l=1
  - Then for chains of length l=2,3, … and so on
  - Computes the optimal cost bottom-up

Algorithm to Compute Optimal Cost

Input: Array $p[0\ldots n]$ containing matrix dimensions and n

Result: Minimum-cost table m and split table s

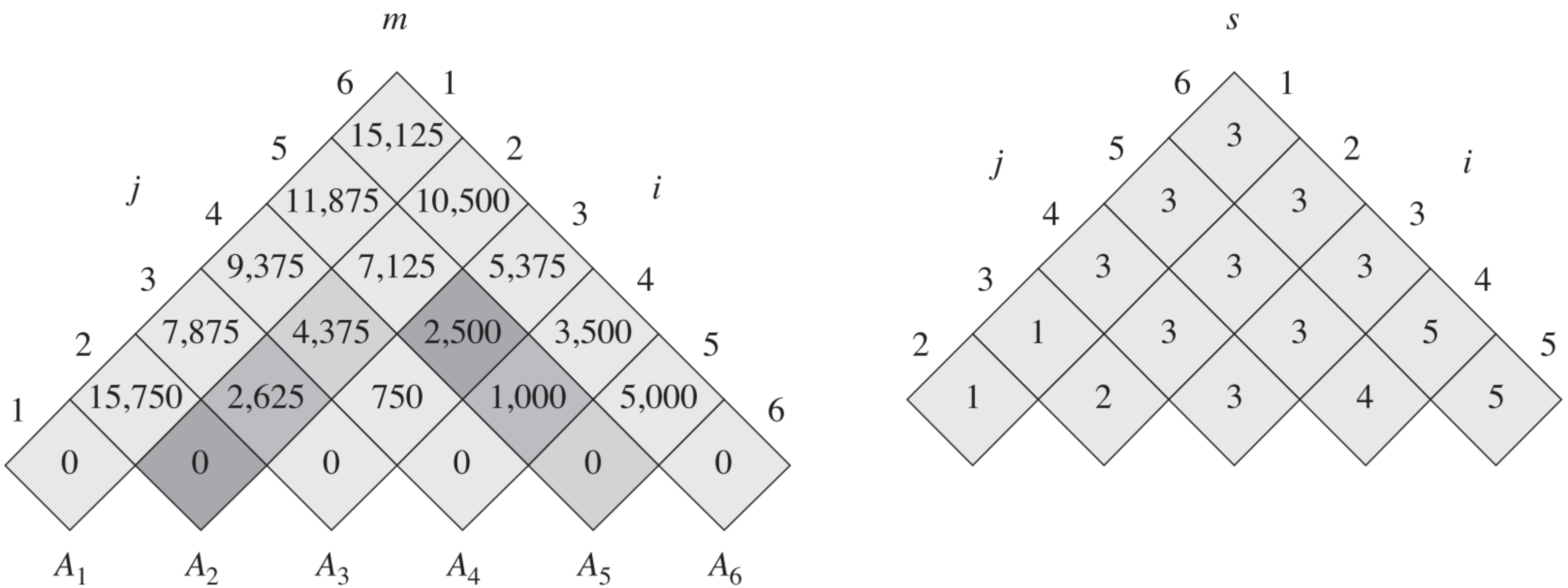
MATRIX-CHAIN-ORDER($p[\ ]$, n)

```
for i ← 1 to n
    m[i, i] ← 0
for l ← 2 to n
    for i ← 1 to n-l+1
            j ← i+l-1
            m[i, j] ← ∞
            for k ← i to j-1
                    q ← m[i, k] + m[k+1, j] + p[i-1] p[k] p[j]
                    if q < m[i, j]
                            m[i, j] ← q
                            s[i, j] ← k
return m and s
```

Takes $O(n^3)$ time

Requires $O(n^2)$ space

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25



$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000 , & \text{k = 2} \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 , & \text{k = 3} \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 & \text{k = 4} \end{cases}$$

$$= 7125 .$$

Step4: Constructing an optimal solution

- Our algorithm computes the minimum-cost table *m* and the split table *s*
- The optimal solution can be constructed from the split table *s*
  - Each entry $s[i, j]=k$ shows where to split the product $A_i A_{i+1} \ldots A_j$ for the minimum cost

Step4: Constructing an optimal solution

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
    print "A"_i
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

Continue with example

- Show how to multiply this matrix chain optimally.
- Optimal parenthesization $((A_1(A_2A_3))((A_4 A_5)A_6))$
- Minimum cost 15,125

Matrix	Dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Content of this Chapter

- Introduction to dynamic programming
- Rod cutting
- Matrix-chain multiplication
- **Elements of dynamic programming**
- 0-1 Knapsack Problem

Elements of dynamic programming

There are *two properties* an optimization problem must have in order for dynamic programming to apply:

- Optimal Substructure
- Overlapping Subproblems

Optimal Substructure

- A problem exhibits *optimal substructure* if an optimal solution to the problem contains within it optimal solutions to subproblems.

- In dynamic programming, we build an optimal solution to the problem from optimal solutions to subproblems.

Optimal Substructure

- *Optimal substructure* varies across problem domains in two ways:

  1. How many subproblems an optimal solution to the original problem uses, and

  2. How many choices we have in determining which subproblem(s) to use in an optimal solution.

Optimal substructure

- In the rod-cutting problem $\Theta(n^2)$:
  - $\Theta(n)$ subproblems overall, and
  - At most n choices to examine for each
- In the matrix-chain multiplication problem $\Theta(n^3)$:
  - $\Theta(n^2)$ subproblems overall, and
  - At most $n-1$ choices to examine for each

DP: Bottom-up approach

- Dynamic programming often uses optimal substructure in a bottom-up fashion:
  - First, find optimal solutions to subproblems.
  - Second, find an optimal solution to the problem.
    - Make a choice among subproblems as to which we will use in solving the problem.
- The cost of the problem solution is usually the subproblem costs *plus a cost that is directly attributable to the choice*.

DP: Bottom-up approach (continue)

For example, in the matrix-chain multiplication problem, to determine the *cost that is directly attributable to the choice*:

- First, we determined an optimal parenthesis of subchain of $A_i\, A_{i+1} \ldots A_j$.
- Second, we chose the matrix A_k at which to split the product.
- Then the cost attributable to the choice itself is:

$$p[i\text{-}1]\; p[k]\; p[j]$$

Optimal Substructure

- **Question:** Do all problems exhibit *optimal substructure property*?

- **Answer**: No!

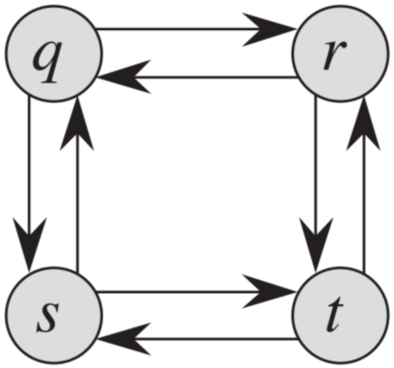
Unweighted *shortest* simple path

- **Problem:** Find a simple path from u to v consisting of the *least* possible edges.
- Does the unweighted shortest-path problem exhibit optimal substructure?
- Yes! Use a "cut-and-paste" argument.

Unweighted *longest* simple path

- **Problem:** Find a simple path from u to v consisting of the most possible edges.

- Does the unweighted longest-path problem exhibit optimal substructure? No!



Counter Example:

The longest path from s to r is s→q→r (or s→t→r), which is of length **2**.

However, the longest path from s to q is of length **3**: s→t→r→q. Similarly, the longest path from q to r is of length **3**: q→s→t→r !!

Overlapping Subproblems

- When a recursive algorithm revisits the *same problem* repeatedly, we say that the optimization problem has *overlapping subproblems*.

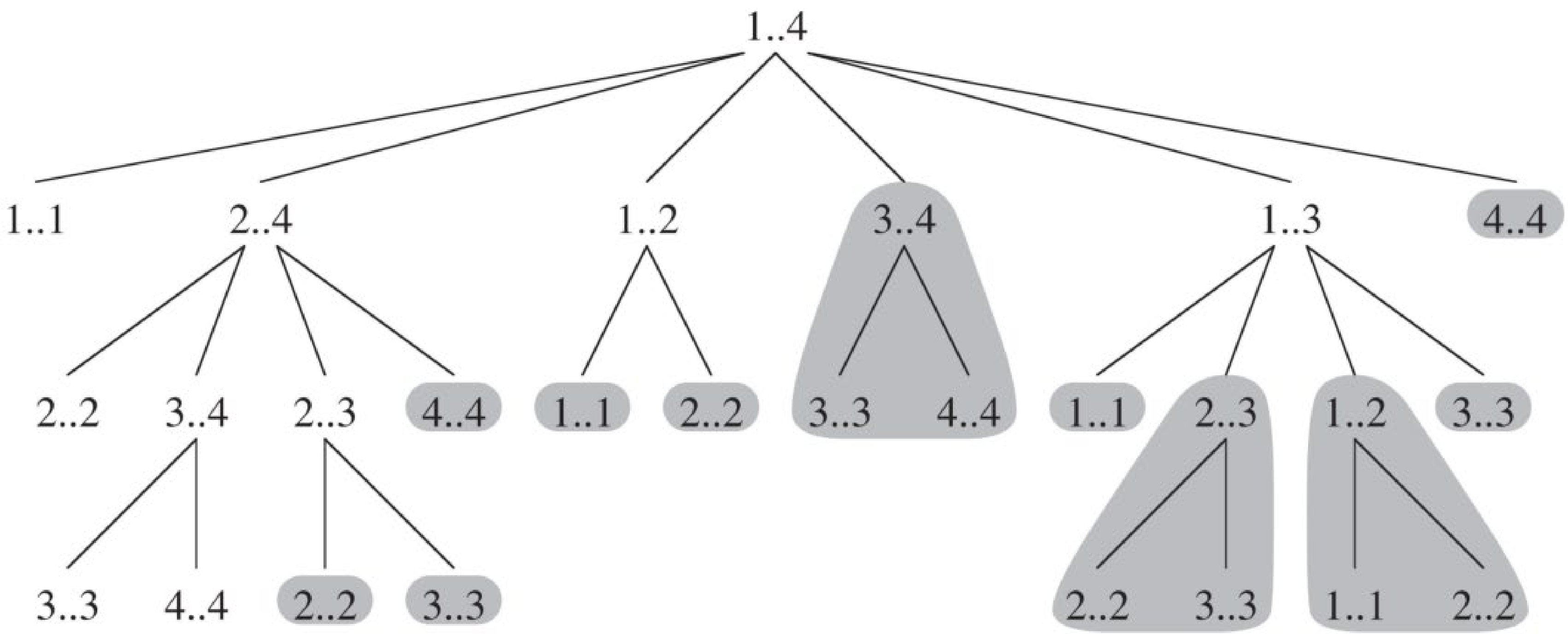
- A problem for which a divide-and-conquer approach is suitable usually generates brand-new problems at each step of the recursion.

- In contrast, DP algorithms take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup.

Recursive Approach (inefficient)

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
if i == j
    return 0
m[i, j] = ∞
for k = i to j − 1
    q = RECURSIVE-MATRIX-CHAIN(p, i, k)
        + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
        + p_{i−1} p_k p_j
    if q < m[i, j]
        m[i, j] = q
return m[i, j]
```

Recursion Tree for RECURSIVE-MATRIX-CHAINU



Content of this Chapter

- Introduction to dynamic programming
- Rod cutting
- Matrix-chain multiplication
- Elements of dynamic programming
- **0-1 Knapsack Problem**

0-1 Knapsack Problem

- We have a knapsack that has a maximum capacity (weight) W.
- We have several items $I_1, \ldots, I_n$.
- Each item I_j has a value v_j and weight w_j.
- We want to place certain items in the knapsack such that:
  1. The knapsack weight capacity is not exceeded, and
  2. The total value of items in the knapsack is maximal.

0-1 because we cannot take a fractional amount of an item or take an item more than once.

Example

Item	Weight	Value
I_1	10 lb	\$60
I_2	20 lb	\$100
I_3	30 lb	\$120

Knapsack Capacity = 50 lb

0-1 Knapsack Problem

Observation:

Let S_i be the optimal subset of elements from $\{I_1, I_2, \ldots, I_i\}$. The optimal subset from the elements $\{I_1, I_2, \ldots, I_{i+1}\}$ may not correspond to the optimal subset of elements from $\{I_1, I_2, \ldots, I_i\}$ in any regular pattern.

0-1 Knapsack Problem

Item	Weight	Value
I_1	3	10
I_2	8	4
I_3	9	9
I_4	8	11

- **The maximum weight the knapsack can hold is W = 20.**

- The best set of items from $\{I_1, I_2, I_3\}$ is $\{I_1, I_2, I_3\}$.
- BUT the best set of items from $\{I_1, I_2, I_3, I_4\}$ is $\{I_1, I_3, I_4\}$.

0-1 Knapsack Problem

Optimal Substructure:

If we remove item j from the optimal load (most valuable load that weighs at most W pounds), then the remaining load must be the most valuable load weighing at most $W\text{-}w_j$ that we can take from the $n\text{-}1$ original items excluding item j.

0-1 Knapsack Problem

Item	Weight	Value
I_1	3	10
I_2	8	4
I_3	9	9
I_4	8	11

- **The maximum weight the knapsack can hold is W = 20.**
- For W=20: the best set of items from $\{I_1, I_2, I_3, I_4\}$ is $\{I_1, I_3, I_4\}$.
- However, if I_4 is removed, then the best set of items from $\{I_1, I_2, I_3\}$ for W'=20-8=12 and is $\{I_1, I_3, I_4\}\backslash\{I_4\} = \{I_1, I_3\}$.

Dynamic *0-1* Knapsack

- Let $c[i, w]$ denote the maximum value (size of an optimal solution) for the item set $S_i = \{I_1, I_2, \ldots, I_i\}$, and weight w.
- Then:

$$c[i, w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0\,, \\ c[i-1, w] & \text{if } w_i > w\,, \\ \max(\underbrace{v_i + c[i-1, w-w_i]}_{\text{including } I_i}, \underbrace{c[i-1, w]}_{\text{without } I_i}) & \text{if } i > 0 \text{ and } w \geq w_i\,. \end{cases}$$

Dynamic *0-1* Knapsack Algorithm

DYNAMIC-0-1-KNAPSACK(v, w, n, W)

let $c[0\,..\,n, 0\,..\,W]$ be a new two-dimensional array (table)
for $w = 0$ **to** W
    $c[0, w] = 0$
for $i = 1$ **to** n
    $c[i, 0] = 0$
    for $w = 1$ **to** W
        if $w_i \le w$
            if $v_i + c[i-1, w-w_i] > c[i-1, w]$
                $c[i, w] = v_i + c[i-1, w-w_i]$
            else $c[i, w] = c[i-1, w]$
        else $c[i, w] = c[i-1, w]$

Complexity: $\Theta(n.W) = \underbrace{\Theta(n.W)}_{\text{fill table } c} + \underbrace{\Theta(n)}_{\text{trace solution}}$

Dynamic *0-1* Knapsack Algorithm

- Example:
  - n = 4 (# of elements)
  - W = 5 (max weight)
  - Elements (weight, value):
    (2,3), (3,4), (4,5), (5,6)

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 1}$

$w - w_i = -1$

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 2}$

$w\text{-}w_i = 0$

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 3}$

$w\text{-}w_i = 1$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0					
3	0					
4	0					

$i = 1$
$v_i = 3$
$w_i = 2$
$\mathbf{w = 4}$
$w\text{-}w_i = 2$

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 5}$

$w\text{-}w_i = 3$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 1}$
$w - w_i = -2$

Knapsack 0-1 Example

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 2}$
$w\text{-}w_i = -1$

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		
3	0					
4	0					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 3}$

$w\text{-}w_i = 0$

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0					
4	0					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 4}$

$w\text{-}w_i = 1$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 5}$
$w - w_i = 2$

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4		
4	0					

$i = 3$

$v_i = 5$

$w_i = 4$

$\mathbf{w = 1..3}$

$w\text{-}w_i = -3..-1$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

$i = 3$
$v_i = 5$
$w_i = 4$
$\mathbf{w = 4}$
$w - w_i = 0$

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

$\boxed{3: (4,5)}$

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

$i = 3$

$v_i = 5$

$w_i = 4$

$\mathbf{w = 5}$

$w - w_i = 1$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	

$i = 4$
$v_i = 6$
$w_i = 5$
$\mathbf{w = 1..4}$
$w - w_i = -4..-1$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 4$
$v_i = 6$
$w_i = 5$
$\mathbf{w = 5}$
$w-w_i = 0$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Knapsack 0-1 Example

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	**7**

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

Finding the Items

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 4$
$k = 5$
$v_i = 6$
$w_i = 5$
$\mathbf{c[i,k] = 7}$
$c[i-1,k] = 7$

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

Finding the Items

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 3$

$k = 5$

$v_i = 5$

$w_i = 4$

$c[i,k] = 7$

$c[i-1,k] = 7$

Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

Item 2

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 2$
$k = 5$
$v_i = 4$
$w_i = 3$
$c[i,k] = 7$
$c[i-1,k] = 3$
$k - w_i = 2$

Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

Item 2
Item 1

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 1$
$k = 2$
$v_i = 3$
$w_i = 2$
$\mathbf{c[i,k] = 3}$
$c[i-1,k] = 0$
$k - w_i = 0$

Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

Item 2
Item 1

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 1$
$k = 2$
$v_i = 3$
$w_i = 2$
$\mathbf{c[i,k] = 3}$
$c[i-1,k] = 0$
$k - w_i = 0$