Design and Analysis of Algorithms
Part IV: Dynamic Programming
Lecture 8: 0-1 Knapsack and Rod Cutting Problems

Yongxin Tong (童咏昕)

School of CSE, Beihang University

yxtong@buaa.edu.cn

Outline

- Introduction to Part IV

- 0-1 Knapsack Problem
  - Problem Definition
  - A Bruteforce Algorithm
  - A Dynamic Programming Algorithm
  - Analysis of DP Algorithm

- Rod Cutting Problem
  - Problem Definition
  - A Bruteforce Algorithm
  - A Dynamic Programming Algorithm

Outline

- Introduction to Part IV

- 0-1 Knapsack Problem
  - Problem Definition
  - A Bruteforce Algorithm
  - A Dynamic Programming Algorithm
  - Analysis of DP Algorithm

- Rod Cutting Problem
  - Problem Definition
  - A Bruteforce Algorithm
  - A Dynamic Programming Algorithm

Introduction

- Dynamic Programming (DP) is similar to Divide and Conquer (D&C)

Introduction

- Dynamic Programming (DP) is similar to Divide and Conquer (D&C)
  - They both partition a problem into smaller subproblems

Introduction

- Dynamic Programming (DP) is similar to Divide and Conquer (D&C)
  - They both partition a problem into smaller subproblems
- DP is preferable when the subproblems overlap, i.e., they share common subproblems

Introduction

- Dynamic Programming (DP) is similar to Divide and Conquer (D&C)
  - They both partition a problem into smaller subproblems
- DP is preferable when the subproblems overlap, i.e., they share common subproblems
- Often DP is used for optimization problems

Introduction

- Dynamic Programming (DP) is similar to Divide and Conquer (D&C)
  - They both partition a problem into smaller subproblems
- DP is preferable when the subproblems overlap, i.e., they share common subproblems
- Often DP is used for optimization problems
  - Problems that have many solutions, and we want to find the best one

Introduction

- Dynamic Programming (DP) is similar to Divide and Conquer (D&C)
  - They both partition a problem into smaller subproblems
- DP is preferable when the subproblems overlap, i.e., they share common subproblems
- Often DP is used for optimization problems
  - Problems that have many solutions, and we want to find the best one
- Main idea of DP

Introduction

- Dynamic Programming (DP) is similar to Divide and Conquer (D&C)
  - They both partition a problem into smaller subproblems
- DP is preferable when the subproblems overlap, i.e., they share common subproblems
- Often DP is used for optimization problems
  - Problems that have many solutions, and we want to find the best one
- Main idea of DP
  - Analyze the structure of an optimal solution

Introduction

- Dynamic Programming (DP) is similar to Divide and Conquer (D&C)
  - They both partition a problem into smaller subproblems
- DP is preferable when the subproblems overlap, i.e., they share common subproblems
- Often DP is used for optimization problems
  - Problems that have many solutions, and we want to find the best one
- Main idea of DP
  - Analyze the structure of an optimal solution
  - Recursively define the value of an optimal solution

Introduction

- Dynamic Programming (DP) is similar to Divide and Conquer (D&C)
  - They both partition a problem into smaller subproblems
- DP is preferable when the subproblems overlap, i.e., they share common subproblems
- Often DP is used for optimization problems
  - Problems that have many solutions, and we want to find the best one
- Main idea of DP
  - Analyze the structure of an optimal solution
  - Recursively define the value of an optimal solution
  - Compute the value of an optimal solution (usually bottom-up)

Introduction to Part IV

- In Part IV, we will illustrate Dynamic Programming (DP) using several examples:
  - 0-1 Knapsack (0–1背包)
  - Rod-Cutting (钢条切割)
  - Chain Matrix Multiplication (矩阵链乘法)
  - Longest Common Subsequences (最长公共子序列)
  - All-Pairs Shortest Paths (所有结点对的最短路径)

Outline

- Introduction to Part IV

- 0-1 Knapsack Problem
  - Problem Definition
  - A Bruteforce Algorithm
  - A Dynamic Programming Algorithm
  - Analysis of DP Algorithm

- Rod Cutting Problem
  - Problem Definition
  - A Bruteforce Algorithm
  - A Dynamic Programming Algorithm

0-1 Knapsack Problem (Informal Description)

We have n items.

0-1 Knapsack Problem (Informal Description)

We have n items.

- v_i denotes the value of the i-th item
- w_i denotes the weight of the i-th item

0-1 Knapsack Problem (Informal Description)

We have n items.

- v_i denotes the value of the i-th item
- w_i denotes the weight of the i-th item
- You are given a knapsack capable of holding total weight W.

0-1 Knapsack Problem (Informal Description)

We have n items.

- v_i denotes the value of the i-th item
- w_i denotes the weight of the i-th item
- You are given a knapsack capable of holding total weight W.

Goal: Use the knapsack to carry items, such that the total value is maximum.

0-1 Knapsack Problem (Informal Description)

We have n items.

- v_i denotes the value of the i-th item
- w_i denotes the weight of the i-th item
- You are given a knapsack capable of holding total weight W.

Goal: Use the knapsack to carry items, such that the total value is maximum.

- We want to find a subset of items to carry such that
  - The total weight is at most W .

0-1 Knapsack Problem (Informal Description)

We have n items.

- v_i denotes the value of the i-th item
- w_i denotes the weight of the i-th item
- You are given a knapsack capable of holding total weight W.

Goal: Use the knapsack to carry items, such that the total value is maximum.

- We want to find a subset of items to carry such that
  - The total weight is at most W .
  - The total value of the items is as large as possible.

0-1 Knapsack Problem (Informal Description)

We have n items.

- v_i denotes the value of the i-th item
- w_i denotes the weight of the i-th item
- You are given a knapsack capable of holding total weight W.

Goal: Use the knapsack to carry items, such that the total value is maximum.

- We want to find a subset of items to carry such that
  - The total weight is at most W .
  - The total value of the items is as large as possible.

We cannot take parts of items, we take the whole item or nothing.

0-1 Knapsack Problem (Informal Description)

We have n items.

- v_i denotes the value of the i-th item
- w_i denotes the weight of the i-th item
- You are given a knapsack capable of holding total weight W.

Goal: Use the knapsack to carry items, such that the total value is maximum.

- We want to find a subset of items to carry such that
  - The total weight is at most W .
  - The total value of the items is as large as possible.

We cannot take parts of items, we take the whole item or nothing.(This is why it is called 0-1.)

0-1 Knapsack Problem (Informal Description)

We have n items.

- v_i denotes the value of the i-th item
- w_i denotes the weight of the i-th item
- You are given a knapsack capable of holding total weight W.

Goal: Use the knapsack to carry items, such that the total value is maximum.

- We want to find a subset of items to carry such that
  - The total weight is at most W .
  - The total value of the items is as large as possible.

We cannot take parts of items, we take the whole item or nothing.(This is why it is called 0-1.)

Question

How should we select the items?

0-1 Knapsack Problem (Formal Description)

Definition

Given $W > 0$,

0-1 Knapsack Problem (Formal Description)

Definition

Given $W > 0$, and two n-tuples of positive numbers
$\langle v_1, v_2, \ldots, v_n \rangle$ and $\langle w_1, w_2 \ldots, w_n \rangle$,

0-1 Knapsack Problem (Formal Description)

Definition

Given $W > 0$, and two n-tuples of positive numbers $\langle v_1, v_2, \ldots, v_n \rangle$ and $\langle w_1, w_2 \ldots, w_n \rangle$, we wish to determine the subset $T \subseteq \{1, 2, \ldots, n\}$ (of items to carry)

0-1 Knapsack Problem (Formal Description)

Definition

Given $W > 0$, and two n-tuples of positive numbers $\langle v_1, v_2, \ldots, v_n \rangle$ and $\langle w_1, w_2 \ldots, w_n \rangle$, we wish to determine the subset $T \subseteq \{1, 2, \ldots, n\}$ (of items to carry) that

maximizes

0-1 Knapsack Problem (Formal Description)

Definition

Given $W > 0$, and two n-tuples of positive numbers $\langle v_1, v_2, \ldots, v_n \rangle$ and $\langle w_1, w_2 \ldots, w_n \rangle$, we wish to determine the subset $T \subseteq \{1, 2, \ldots, n\}$ (of items to carry) that

$$\text{maximizes} \quad \sum_{i \in T} v_i$$

0-1 Knapsack Problem (Formal Description)

Definition

Given $W > 0$, and two n-tuples of positive numbers $\langle v_1, v_2, \ldots, v_n \rangle$ and $\langle w_1, w_2 \ldots, w_n \rangle$, we wish to determine the subset $T \subseteq \{1, 2, \ldots, n\}$ (of items to carry) that

$$\text{maximizes} \quad \sum_{i \in T} v_i$$

subject to

0-1 Knapsack Problem (Formal Description)

Definition

Given $W > 0$, and two n-tuples of positive numbers $\langle v_1, v_2, \ldots, v_n \rangle$ and $\langle w_1, w_2 \ldots, w_n \rangle$, we wish to determine the subset $T \subseteq \{1, 2, \ldots, n\}$ (of items to carry) that

$$\text{maximizes} \quad \sum_{i \in T} v_i$$

$$\text{subject to} \quad \sum_{i \in T} w_i$$

0-1 Knapsack Problem (Formal Description)

Definition

Given $W > 0$, and two n-tuples of positive numbers $\langle v_1, v_2, \ldots, v_n \rangle$ and $\langle w_1, w_2 \ldots, w_n \rangle$, we wish to determine the subset $T \subseteq \{1, 2, \ldots, n\}$ (of items to carry) that

$$\text{maximizes} \quad \sum_{i \in T} v_i$$

$$\text{subject to} \quad \sum_{i \in T} w_i \leq W.$$

0-1 Knapsack Problem (Formal Description)

Definition

Given $W > 0$, and two n-tuples of positive numbers $\langle v_1, v_2, \ldots, v_n \rangle$ and $\langle w_1, w_2 \ldots, w_n \rangle$, we wish to determine the subset $T \subseteq \{1, 2, \ldots, n\}$ (of items to carry) that

$$\text{maximizes} \quad \sum_{i \in T} v_i$$

$$\text{subject to} \quad \sum_{i \in T} w_i \leq W.$$

Remark: This is an optimization problem.

0-1 Knapsack Problem (Formal Description)

Definition

Given $W > 0$, and two n-tuples of positive numbers $\langle v_1, v_2, \ldots, v_n \rangle$ and $\langle w_1, w_2 \ldots, w_n \rangle$, we wish to determine the subset $T \subseteq \{1, 2, \ldots, n\}$ (of items to carry) that

$$\text{maximizes} \quad \sum_{i \in T} v_i$$

$$\text{subject to} \quad \sum_{i \in T} w_i \leq W.$$

Remark: This is an optimization problem. The brute force solution is to try all 2^n possible subsets T.

Outline

- Introduction to Part IV
- 0-1 Knapsack Problem
  - Problem Definition
  - A Bruteforce Algorithm
  - A Dynamic Programming Algorithm
  - Analysis of DP Algorithm
- Rod Cutting Problem
  - Problem Definition
  - A Bruteforce Algorithm
  - A Dynamic Programming Algorithm

Simple Recursion

Let V [i, w] denote the maximum (combined) value of any subset of items {1,2,…,i} of (combined) weight at most w.

Simple Recursion

Let V [i, w] denote the maximum (combined) value of any subset of items {1,2,…,i} of (combined) weight at most w.

We can either leave the i-th item or take it. If we take it, we gain value v_i, but only $w - w_i$ space remains. Therefore:

Simple Recursion

Let V [i, w] denote the maximum (combined) value of any subset of items {1,2,…,i} of (combined) weight at most w.

We can either leave the i-th item or take it. If we take it, we gain value v_i, but only w − w_i space remains. Therefore:

$$V[i,w] = \max(V[i-1,w], v_i + V[i-1, w-w_i])$$

Simple Recursion

KnapsackSR(i,w)

Input: Candidate item set $\{1, 2, ..., \boldsymbol{i}\}$, allowed maximum weight of items $\boldsymbol{w}$.

Output: Maximum value of any subset of items $\{1, 2, ..., i\}$ of weight at most w.

Simple Recursion

KnapsackSR(i,w)

Input: Candidate item set $\{1, 2, ..., \boldsymbol{i}\}$, allowed maximum weight of items $\boldsymbol{w}$.

Output: Maximum value of any subset of items $\{1, 2, ..., i\}$ of weight at most w.

if $W < 0$ **then**
| **return** $-\infty$;
end

Simple Recursion

KnapsackSR(i,w)

Input: Candidate item set $\{1, 2, ..., \boldsymbol{i}\}$, allowed maximum weight of items $\boldsymbol{w}$.

Output: Maximum value of any subset of items $\{1, 2, ..., i\}$ of weight at most w.

if $W < 0$ **then**
  return $-\infty$;
end
if i *is equal to* 0 **then**
  return 0;
end

Simple Recursion

KnapsackSR(i,w)

Input: Candidate item set $\{1, 2, ..., \boldsymbol{i}\}$, allowed maximum weight of items $\boldsymbol{w}$.

Output: Maximum value of any subset of items $\{1, 2, ..., i\}$ of weight at most w.

if $W < 0$ **then**
    return $-\infty$;
end
if i *is equal to* 0 **then**
    return 0;
end
$V_1 \leftarrow$KnapsackSR$(i-1, w)$;

Simple Recursion

KnapsackSR(i,w)

Input: Candidate item set $\{1, 2, ..., \boldsymbol{i}\}$, allowed maximum weight of items $\boldsymbol{w}$.

Output: Maximum value of any subset of items $\{1, 2, ..., i\}$ of weight at most w.

if $W < 0$ **then**
| **return** $-\infty$;
end
if i *is equal to* 0 **then**
| **return** 0;
end
$V_1 \leftarrow$KnapsackSR$(i - 1, w)$;
$V_2 \leftarrow$KnapsackSR$(i - 1, w - w_i)$;

Simple Recursion

KnapsackSR(i,w)

Input: Candidate item set $\{1, 2, ..., \boldsymbol{i}\}$, allowed maximum weight of items $\boldsymbol{w}$.
Output: Maximum value of any subset of items $\{1, 2, ..., i\}$ of weight at most w.

if $W < 0$ **then**
    return $-\infty$;
end
if i *is equal to* 0 **then**
    return 0;
end
$V_1 \leftarrow$KnapsackSR$(i - 1, w)$;
$V_2 \leftarrow$KnapsackSR$(i - 1, w - w_i)$;
return $max\{V_1, v_i + V_2\}$;

Simple Recursion: How bad can it be?

- Consider the simple case w[i] = 1 for all i.

Simple Recursion: How bad can it be?

- Consider the simple case w[i] = 1 for all i.
- Function calls:

Simple Recursion: How bad can it be?

- Consider the simple case w[i] = 1 for all i.
- Function calls:
  - Entry Level: V (n, W)

Simple Recursion: How bad can it be?

- Consider the simple case w[i] = 1 for all i.
- Function calls:
  - Entry Level: V (n, W)
  - Level 1:V(n−1, W), V(n−1, W−1)

Simple Recursion: How bad can it be?

- Consider the simple case w[i] = 1 for all i.
- Function calls:
  - Entry Level: V (n, W)
  - Level 1:V(n−1, W), V(n−1, W−1)
  - Level 2:V(n−2, W), V(n−2, W−1); V(n−2, W−1),V(n−2, W−2)

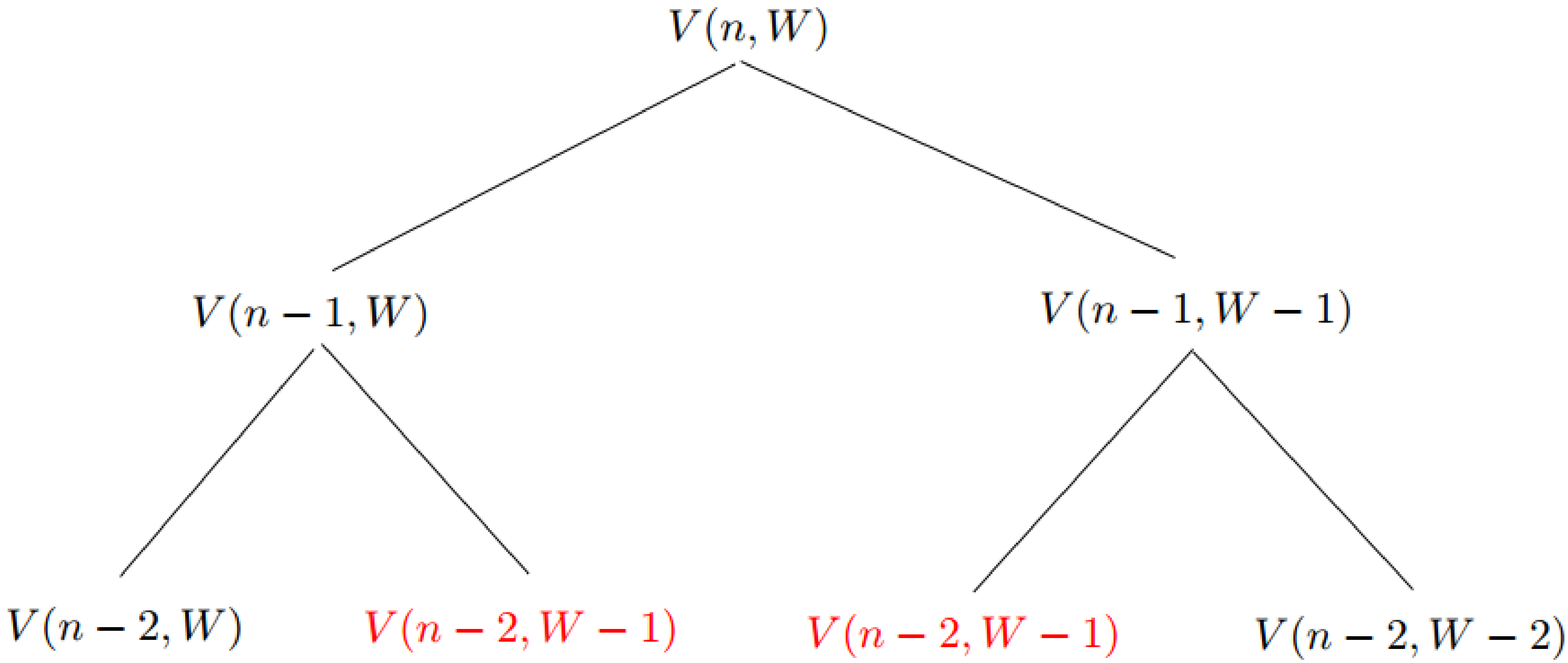
Simple Recursion: How bad can it be?

- Consider the simple case w[i] = 1 for all i.
- Function calls:
  - Entry Level: V (n, W)
  - Level 1:V(n−1, W), V(n−1, W−1)
  - Level 2:V(n−2, W), V(n−2, W−1); V(n−2, W−1),V(n−2, W−2)
  - ...
- Running time: At least $\Omega(2^W)$!

Simple Recursion: How bad can it be?

- Consider the simple case w[i] = 1 for all i.
- Function calls:
  - Entry Level: V (n, W)
  - Level 1:V(n−1, W), V(n−1, W−1)
  - Level 2:V(n−2, W), V(n−2, W−1); V(n−2, W−1),V(n−2, W−2)
  - ...
- Running time: At least $\Omega(2^W)$!
- Why? We invoke the same function call too many times!

Simple Recursion: How bad can it be?



Outline

- Introduction to Part IV

- 0-1 Knapsack Problem
  - Problem Definition
  - A Bruteforce Algorithm
  - A Dynamic Programming Algorithm
  - Analysis of DP Algorithm

- Rod Cutting Problem
  - Problem Definition
  - A Bruteforce Algorithm
  - A Dynamic Programming Algorithm

Developing a DP Algorithm for Knapsack

Step 1: Space of Subproblems (State)

Developing a DP Algorithm for Knapsack

Step 1: Space of Subproblems (State)

- For 1 ≤ i ≤ n, and 0 ≤ w ≤ W , define

$$\mathrm{V}[i,w] = \max\{\sum_{j\in T} v_j : T \subseteq \{1,2,\ldots,i\}, \sum_{j\in T} w_j \leq w\}$$

Developing a DP Algorithm for Knapsack

Step 1: Space of Subproblems (State)

- For $1 \le i \le n$, and $0 \le w \le W$, define

$$\mathrm{V}[i,w] = \max\{\sum_{j \in T} v_j : T \subseteq \{1,2,\ldots,i\}, \sum_{j \in T} w_j \le w\}$$

The maximum (combined) value of any subset of items {1,2,…,i} of (combined) weight at most w.

Developing a DP Algorithm for Knapsack

Step 1: Space of Subproblems (State)

- For 1 ≤ i ≤ n, and 0 ≤ w ≤ W , define

$$\mathrm{V}[i, w] = \max\{\sum_{j \in T} v_j : T \subseteq \{1,2, \dots, i\}, \sum_{j \in T} w_j \le w\}$$

  The maximum (combined) value of any subset of items {1,2,…,i} of (combined) weight at most w.

- Note that our problem is:

Developing a DP Algorithm for Knapsack

Step 1: Space of Subproblems (State)

- For 1 ≤ i ≤ n, and 0 ≤ w ≤ W , define

$$\mathrm{V}[i,w] = \max\{\sum_{j\in T} v_j : T \subseteq \{1,2,\ldots,i\}, \sum_{j\in T} w_j \leq w\}$$

  The maximum (combined) value of any subset of items {1,2,…,i} of (combined) weight at most w.

- Note that our problem is: find V[n, W]

Developing a DP Algorithm for Knapsack

Step 1: Space of Subproblems (State)

- For 1 ≤ i ≤ n, and 0 ≤ w ≤ W , define

$$V[i,w] = \max\{\sum_{j\in T} v_j : T \subseteq \{1,2,\ldots,i\}, \sum_{j\in T} w_j \le w\}$$

  The maximum (combined) value of any subset of items {1,2,…,i} of (combined) weight at most w.

- Note that our problem is: find V[n, W]

Terms:

- T is a solution for [i, w] if T ⊆ {1,2,…,i} and $\sum_{j\in T} w_j \le w$

Developing a DP Algorithm for Knapsack

Step 1: Space of Subproblems (State)

- For 1 ≤ i ≤ n, and 0 ≤ w ≤ W , define

$$V[i,w] = \max\{\sum_{j\in T} v_j : T \subseteq \{1,2,\dots,i\}, \sum_{j\in T} w_j \le w\}$$

  The maximum (combined) value of any subset of items {1,2,…,i} of (combined) weight at most w.

- Note that our problem is: find V[n, W]

Terms:

- T is a solution for [i, w] if T ⊆ {1,2,…,i} and $\sum_{j\in T} w_j \le w$
- T is an optimal solution for [i, w] if

Developing a DP Algorithm for Knapsack

Step 1: Space of Subproblems (State)

- For 1 ≤ i ≤ n, and 0 ≤ w ≤ W , define

$$V[i,w] = \max\{\sum_{j \in T} v_j : T \subseteq \{1,2,\ldots,i\}, \sum_{j \in T} w_j \leq w\}$$

The maximum (combined) value of any subset of items {1,2,…,i} of (combined) weight at most w.

- Note that our problem is: find V[n, W]

Terms:

- T is a solution for [i, w] if T ⊆ {1,2,…,i} and $\sum_{j \in T} w_j \leq w$
- T is an optimal solution for [i, w] if T is a solution and

Developing a DP Algorithm for Knapsack

Step 1: Space of Subproblems (State)

- For 1 ≤ i ≤ n, and 0 ≤ w ≤ W , define

$$V[i,w] = \max\{\sum_{j\in T} v_j : T \subseteq \{1,2,\dots,i\}, \sum_{j\in T} w_j \le w\}$$

The maximum (combined) value of any subset of items {1,2,…,i} of (combined) weight at most w.

- Note that our problem is: find V[n, W]

Terms:

- T is a solution for [i, w] if T ⊆ {1,2,…,i} and $\sum_{j\in T} w_j \le w$
- T is an optimal solution for [i, w] if T is a solution and $\sum_{j\in T} v_j = V[i,w]$.

Developing a DP Algorithm for Knapsack

Step 2: Relating a problem to its subproblems

Developing a DP Algorithm for Knapsack

Step 2: Relating a problem to its subproblems

Recursion:

$$V[i, w] =$$

Developing a DP Algorithm for Knapsack

Step 2: Relating a problem to its subproblems

Recursion:

$$V[i, w] = \qquad v_i + V[i-1, w-w_i]$$

Developing a DP Algorithm for Knapsack

Step 2: Relating a problem to its subproblems

Recursion:

$$V[i, w] = \qquad V[i-1, w], v_i + V[i-1, w-w_i]$$

Developing a DP Algorithm for Knapsack

Step 2: Relating a problem to its subproblems

Recursion:

$$V[i,w] = \max(V[i-1,w], v_i + V[i-1,w-w_i])$$

Developing a DP Algorithm for Knapsack

Step 2: Relating a problem to its subproblems

Recursion:

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w-w_i])$$

for 1 ≤ i ≤ n, 0 ≤ w ≤ W.

Developing a DP Algorithm for Knapsack

Step 2: Relating a problem to its subproblems

Recursion:

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w-w_i])$$

for $1 \leq i \leq n$, $0 \leq w \leq W$.

- Optimal structure: optimal solution for problem contains optimal solutions to subproblems. (Indication that DP might apply)

Developing a DP Algorithm for Knapsack

Step 2: Relating a problem to its subproblems

Recursion:

$$V[i,w] = \max(V[i-1,w], v_i + V[i-1, w-w_i])$$

for 1 ≤ i ≤ n, 0 ≤ w ≤ W.

- Optimal structure: optimal solution for problem contains optimal solutions to subproblems. (Indication that DP might apply)
- The recursion is why we chose the problem space {V[i, w] : 1 ≤ i ≤ n; 0 ≤ w ≤ W} at the first place.

Developing a DP Algorithm for Knapsack

Step 2: Relating a problem to its subproblems

Recursion:

$$V[i,w] = \max(V[i-1,w], v_i + V[i-1,w-w_i])$$

for $1 \le i \le n$, $0 \le w \le W$.

- Optimal structure: optimal solution for problem contains optimal solutions to subproblems. (Indication that DP might apply)
- The recursion is why we chose the problem space $\{V[i, w] : 1 \le i \le n; 0 \le w \le W\}$ at the first place.

Boundary cases:

Developing a DP Algorithm for Knapsack

Step 2: Relating a problem to its subproblems

Recursion:

$$V[i,w] = \max(V[i-1,w], v_i + V[i-1, w-w_i])$$

for $1 \le i \le n$, $0 \le w \le W$.

- Optimal structure: optimal solution for problem contains optimal solutions to subproblems. (Indication that DP might apply)
- The recursion is why we chose the problem space $\{V[i, w] : 1 \le i \le n; 0 \le w \le W\}$ at the first place.

Boundary cases:

Set

$V[0, w] = 0$ for $0 \le w \le W$, no item

Developing a DP Algorithm for Knapsack

Step 2: Relating a problem to its subproblems

Recursion:

$$V[i,w] = \max(V[i-1,w], v_i + V[i-1,w-w_i])$$

for 1 ≤ i ≤ n, 0 ≤ w ≤ W.

- Optimal structure: optimal solution for problem contains optimal solutions to subproblems. (Indication that DP might apply)
- The recursion is why we chose the problem space {V[i, w] : 1 ≤ i ≤ n; 0 ≤ w ≤ W} at the first place.

Boundary cases:

Set

V [0, w] = 0 for 0 ≤ w ≤ W , no item

V [i, w] = −∞ for w < 0, illegal

Developing a DP Algorithm for Knapsack

Step 3: Bottom-up computation of V [i, w]

Recurrence:

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w-w_i])$$

Developing a DP Algorithm for Knapsack

Step 3: Bottom-up computation of V [i, w]

Recurrence:

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w-w_i])$$

We compute and save V[i, w]($0 \le i \le n$, $0 \le w \le W$) in such an order that:

Developing a DP Algorithm for Knapsack

Step 3: Bottom-up computation of V [i, w]

Recurrence:

$$V[i,w] = \max(V[i-1,w], v_i + V[i-1,w-w_i])$$

We compute and save V[i, w]($0 \le i \le n$, $0 \le w \le W$) in such an order that: When it is time to compute V [i, w], the values of V[i – 1, w] and V [i – 1, w – w_i] are available.

Developing a DP Algorithm for Knapsack

Step 3: Bottom-up computation of V [i, w]

Recurrence:

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w-w_i])$$

We compute and save V[i, w](0 ≤ i ≤ n, 0 ≤ w ≤ W) in such an order that: When it is time to compute V [i, w], the values of V[i – 1, w] and V [i – 1, w – w_i] are available.

So, we fill the following table row by row and left to right.

V[i,w]	w=0	1	2	3	···	···	W	
i= 0	0	0	0	0	...	...	0	bottom
1								
2								
⋮								
n								

An Example of the Bottom-up Computation

i	**1**	**2**	**3**	**4**
v_i	10	40	30	50
w_i	5	4	6	3

W=10

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$											

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0

An Example of the Bottom-up Computation

i	**1**	**2**	**3**	**4**
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$i = 0$	0	0	0	0	0	0	0	0	0	0	0

Intialization

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1											

w[i]>w

$keep$	0	1	2	3	4	5	6	7	8	9	10
$i=1$											

An Example of the Bottom-up Computation

$\boldsymbol{i}$	**1**	**2**	**3**	**4**
$\boldsymbol{v_i}$	10	40	30	50
$\boldsymbol{w_i}$	5	4	6	3

W=10

$\boldsymbol{V[i,w]}$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$\boldsymbol{i=0}$	0	0	0	0	0	0	0	0	0	0	0
$\mathbf{1}$	0	0	0	0	0						

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5		6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0			0	0	0
1	0				0						

$V[i]$

$V[i-1, w-w[i]]$

$V[i-1, w]$

$V[i] + V[i-1, w-w[i]] > V[i-1, w]$

$$V[i,w] = \max(V[i-1,w], v_i + V[i-1, w-w_i])$$

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10					

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10				

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10			

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10		

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0							

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40						

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40					

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40				

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40		

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40					

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40					

$V[i] + V[i-1, j-w[i]] < \boldsymbol{V[i-1, j]}$

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40				

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40			

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40		

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0								

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50							

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50						

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50					

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50				

An Example of the Bottom-up Computation

i	**1**	**2**	**3**	**4**
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i, w]$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90			

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90		

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

An Example of the Bottom-up Computation

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

Max value = 90

The Dynamic Programming Algorithm

Knapsack(v,w,n,W)

Input: $\boldsymbol{v}$ and $\boldsymbol{w}$ are values and weights of $\boldsymbol{n}$ items, $\boldsymbol{W}$ is the allowed maximum weight of items.

Output: Maximum value of any subset of items $\{1, 2, ..., n\}$ of weight at most W.

The Dynamic Programming Algorithm

Knapsack(v,w,n,W)

Input: $\boldsymbol{v}$ and $\boldsymbol{w}$ are values and weights of $\boldsymbol{n}$ items, $\boldsymbol{W}$ is the allowed maximum weight of items.

Output: Maximum value of any subset of items $\{1, 2, ..., n\}$ of weight at most W.

Let $V[0..n, 0..W]$ be a new 2-dimension array;

The Dynamic Programming Algorithm

Knapsack(v,w,n,W)

Input: $\boldsymbol{v}$ and $\boldsymbol{w}$ are values and weights of $\boldsymbol{n}$ items, $\boldsymbol{W}$ is the allowed maximum weight of items.

Output: Maximum value of any subset of items $\{1, 2, ..., n\}$ of weight at most W.

```
Let V[0..n, 0..W] be a new 2-dimension array;
for w = 0 to W do
    V[0, w] = 0
end
for i = 1 to n do
    for w = 0 to W do
        if w[i] ≤ w then
            V[i, w] = max{V[i − 1, w], v[i] + V[i − 1, w − w[i]]}
        else
            V[i, w] = V[i − 1, w]
        end
    end
end
return V[n, W]
```

Constructing the Optimal Solution

- The algorithm for computing V [i, w] described in the previous slide does not record which subset of items gives the optimal solution.

Constructing the Optimal Solution

- The algorithm for computing V [i, w] described in the previous slide does not record which subset of items gives the optimal solution.
- To compute the actual subset, we can

Constructing the Optimal Solution

- The algorithm for computing V [i, w] described in the previous slide does not record which subset of items gives the optimal solution.
- To compute the actual subset, we can add an auxiliary boolean array keep[i, w]

Constructing the Optimal Solution

- The algorithm for computing V [i, w] described in the previous slide does not record which subset of items gives the optimal solution.
- To compute the actual subset, we can add an auxiliary boolean array keep[i, w] which is
    - 1 if we choose item i in V [i, w] and

Constructing the Optimal Solution

- The algorithm for computing V [i, w] described in the previous slide does not record which subset of items gives the optimal solution.
- To compute the actual subset, we can add an auxiliary boolean array keep[i, w] which is
  - 1 if we choose item i in V [i, w] and
  - 0 otherwise.

Constructing the Optimal Solution

- The algorithm for computing V [i, w] described in the previous slide does not record which subset of items gives the optimal solution.
- To compute the actual subset, we can add an auxiliary boolean array keep[i, w] which is
    - 1 if we choose item i in V [i, w] and
    - 0 otherwise.

Question

How do we use all the values $keep[i, w]$ to determine the subset T of items having the maximum value?

Constructing the Optimal Solution

If keep[n, W] is 1, then $n \in T$.

Constructing the Optimal Solution…

If keep[n, W] is 1, then $n \in T$.

We can now repeat this argument for keep[$n-1$, $W-w_n$].

Constructing the Optimal Solution

If keep[n, W] is 1, then $n \in T$.

We can now repeat this argument for keep[n−1, W−w_n].

If keep[n, W] is 0, then $n \notin T$.

Constructing the Optimal Solution

If keep[n, W] is 1, then $n \in T$.

We can now repeat this argument for keep[n−1, W−w_n].

If keep[n, W] is 0, then $n \notin T$.

We repeat the argument for keep[n−1, W].

Constructing the Optimal Solution

If keep[n, W] is 1, then $n \in T$.

We can now repeat this argument for keep[n−1, W−w_n].

If keep[n, W] is 0, then $n \notin T$.

We repeat the argument for keep[n−1, W].

- Therefore, the following partial program will output the elements of T:

Constructing the Optimal Solution

If keep[n, W] is 1, then $n \in T$.

We can now repeat this argument for keep[n−1, $W-w_n$].

If keep[n, W] is 0, then $n \notin T$.

We repeat the argument for keep[n−1, W].

- Therefore, the following partial program will output the elements of T:

getResult(W,V)

Input: Allowed maximum weight $\boldsymbol{W}$, intermediate array from Knapsack $\boldsymbol{V}$

Output: Maximum value of any subset of items $\{1, 2, ..., n\}$ of weight at most W.

Constructing the Optimal Solution

If keep[n, W] is 1, then n $\in$ T.

We can now repeat this argument for keep[n−1, W−w_n].

If keep[n, W] is 0, then n $\notin$ T.

We repeat the argument for keep[n−1, W].

- Therefore, the following partial program will output the elements of T:

getResult(W,V)

Input: Allowed maximum weight $\boldsymbol{W}$, intermediate array from Knapsack $\boldsymbol{V}$

Output: Maximum value of any subset of items $\{1, 2, ..., n\}$ of weight at most W.

$K \leftarrow W$;

Constructing the Optimal Solution

If keep[n, W] is 1, then $n \in T$.

We can now repeat this argument for keep[n−1, W−w_n].

If keep[n, W] is 0, then $n \notin T$.

We repeat the argument for keep[n−1, W].

- Therefore, the following partial program will output the elements of T:

getResult(W,V)

Input: Allowed maximum weight $\boldsymbol{W}$, intermediate array from Knapsack $\boldsymbol{V}$

Output: Maximum value of any subset of items $\{1, 2, ..., n\}$ of weight at most W.

$K \leftarrow W$;

for $i \leftarrow n$ *to* 1 **do**

  if $keep[i, K]$ *is equal to* 1 **then**

Constructing the Optimal Solution

If keep[n, W] is 1, then $n \in T$.

We can now repeat this argument for keep[n−1, $W-w_n$].

If keep[n, W] is 0, then $n \notin T$.

We repeat the argument for keep[n−1, W].

- Therefore, the following partial program will output the elements of T:

getResult(W,V)

Input: Allowed maximum weight $\boldsymbol{W}$, intermediate array from Knapsack $\boldsymbol{V}$

Output: Maximum value of any subset of items $\{1, 2, ..., n\}$ of weight at most W.

$K \leftarrow W$;

for $i \leftarrow n$ *to* 1 **do**

  if $keep[i, K]$ *is equal to* 1 **then**

    Output i;

    $K \leftarrow$

Constructing the Optimal Solution

If keep[n, W] is 1, then $n \in T$.

We can now repeat this argument for keep[n−1, $W-w_n$].

If keep[n, W] is 0, then $n \notin T$.

We repeat the argument for keep[n−1, W].

- Therefore, the following partial program will output the elements of T:

getResult(W,V)

Input: Allowed maximum weight $\boldsymbol{W}$, intermediate array from Knapsack $\boldsymbol{V}$

Output: Maximum value of any subset of items $\{1, 2, ..., n\}$ of weight at most W.

```
K ← W;
for i ← n to 1 do
    if keep[i, K] is equal to 1 then
        Output i;
        K ← K − w[i];
    end
end
```

The Complete Algorithm for the Knapsack Problem

Let $V[0..n, 0..W]$ and $keep[0..n, 0..W]$ be two new 2-dimension arrays;

The Complete Algorithm for the Knapsack Problem

```
Let V[0..n, 0..W] and keep[0..n, 0..W] be two new 2-dimension arrays;
for w = 0 to W do V[0, w] = 0;
for i = 1 to n do
    for w = 0 to W do
        if (w[i] ≤ w) and (v[i] + V[i − 1, w − w[i]] > V[i − 1, w]) then
            V[i, w] = v[i] + V[i − 1, w − w[i]];
            keep[i, w] = 1;
        else
            V[i,w]= V[i-1,w];
            keep[i, w] = 0;
        end
    end
end
K = W;
for i = n downto 1 do
    if keep[i, K] == 1 then
        output i;
        K= K-w[i];
    end
end
return V[n, W]
```

Example of Optimal Solution Construction

i	**1**	**2**	**3**	**4**
v_i	10	40	30	50
w_i	5	4	6	3

W=10

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10

$keep$	0	1	2	3	4	5	6	7	8	9	10

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$											

$keep$	0	1	2	3	4	5	6	7	8	9	10

Example of Optimal Solution Construction

i	**1**	**2**	**3**	**4**
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$i=0$	0	0	0	0	0	0	0	0	0	0	0

$keep$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0

Intialization

$keep$	0	1	2	3	4	5	6	7	8	9	10

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1											

$keep$	0	1	2	3	4	5	6	7	8	9	10
1											

Example of Optimal Solution Construction

i	**1**	**2**	**3**	**4**
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1											

w[i]>j

$keep$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$\mathbf{i = 1}$											

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0						

$keep$	0	1	2	3	4	5	6	7	8	9	10
$i = 1$	0	0	0	0	0						

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5		6	3

W=10

$V[i]$

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0				0						

$V\left[i-1, j-w[i]\right]$

$\boldsymbol{V[i-1, j]}$

$V[i] + V\left[i-1, j-w[i]\right] > \boldsymbol{V[i-1, j]}$

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i = 1}$	0	0	0	0	0						

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10					

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i = 1}$	0	0	0	0	0	1					

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10				

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i = 1}$	0	0	0	0	0	1	1				

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10			

$keep$	0	1	2	3	4	5	6	7	8	9	10
$i = 1$	0	0	0	0	0	1	1	1			

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10		

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathrm{i} = 1$	0	0	0	0	0	1	1	1	1		

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i = 1}$	0	0	0	0	0	1	1	1	1	1	

Example of Optimal Solution Construction

i	**1**	**2**	**3**	**4**
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10

$keep$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$\mathbf{i=1}$	0	0	0	0	0	1	1	1	1	1	1

Example of Optimal Solution Construction

i	**1**	**2**	**3**	**4**
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10

$keep$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$\mathbf{i=1}$	0	0	0	0	0	1	1	1	1	1	1

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0							

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i = 1}$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0							

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40						

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i} = 1$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1						

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40					

$keep$	0	1	2	3	4	5	6	7	8	9	10
i = 1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1					

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40				

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i = 1}$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1				

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40			

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i = 1}$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1			

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40		

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i = 1}$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1		

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i}=1$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i = 1}$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3											

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i} = 1$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3											

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40					

$keep$	0	1	2	3	4	5	6	7	8	9	10
$i = 1$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0					

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40					

$V[i] + V[i-1, j-w[i]] < V[i-1, j]$

$keep$	0	1	2	3	4	5	6	7	8	9	10
$i = 1$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0					

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40				

$keep$	0	1	2	3	4	5	6	7	8	9	10
i = 1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0				

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40			

$keep$	0	1	2	3	4	5	6	7	8	9	10
i = 1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0			

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40		

$keep$	0	1	2	3	4	5	6	7	8	9	10
$i = 1$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0		

Example of Optimal Solution Construction

i	**1**	**2**	**3**	**4**
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	

$keep$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$\mathbf{i = 1}$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	$\boxed{30}$	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	$\boxed{40}$	40	40	40	40	50	$\boxed{50}$
3	0	0	0	0	40	40	40	40	40	50	70

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i=1}$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	1

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4											

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathrm{i}=1$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	1
4											

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0								

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i}=1$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0								

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50							

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i} = 1$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	1							

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50						

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i = 1}$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	1	1						

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50					

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i = 1}$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	1	1	1					

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50				

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i=1}$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	1	1	1	1				

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90			

$keep$	0	1	2	3	4	5	6	7	8	9	10
$i=1$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	1	1	1	1	1			

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90		

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i} = 1$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	1	1	1	1	1	1		

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i = 1}$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	1	1	1	1	1	1	1	

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i=1}$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	1	1	1	1	1	1	1	1

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

Max value = 90

$keep$	0	1	2	3	4	5	6	7			
$\mathbf{i = 1}$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	1	1	1	1	1	1	1	1

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i = 1}$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	1	1	1	1	1	1	1	1

W=10 Item set = {}

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i} = 1$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	1	1	1	1	1	1	1	$\boxed{1}$

W=7 Item set = {4,}

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i = 1}$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	1	1	1	1	1	1	1	1

W=7 Item set = {4,}

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i = 1}$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	1	1	1	1	1	1	1	1

W=7 Item set = {4,}

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i = 1}$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	1	1	1	1	1	1	1	1

W=3 Item set = {4,2}

Example of Optimal Solution Construction

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

W=10

$V[i,j]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

$keep$	0	1	2	3	4	5	6	7	8	9	10
$\mathbf{i} = 1$	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	1	1	1	1	1	1	1	1

W=3 Item set = {4,2}

Outline

- Introduction to Part IV

- 0-1 Knapsack Problem
  - Problem Definition
  - A Bruteforce Algorithm
  - A Dynamic Programming Algorithm
  - Analysis of DP Algorithm

- Rod Cutting Problem
  - Problem Definition
  - A Bruteforce Algorithm
  - A Dynamic Programming Algorithm

Dynamic Programming vs. Simple Recursion

- Dynamic programming saves us from having to recompute
- previously calculated subsolutions!
- The DP algorithm runs in O(   ) time.

Dynamic Programming vs. Simple Recursion

- Dynamic programming saves us from having to recompute
- Previously calculated subsolutions!
- The DP algorithm runs in O(nW) time.

Dynamic Programming vs. Simple Recursion

- Dynamic programming saves us from having to recompute
- Previously calculated subsolutions!
- The DP algorithm runs in O(nW) time.
- The simple recursion algorithm runs in $\Omega(2^n)$ time.

Summary of DP

How to develop a dynamic programming?

Summary of DP

How to develop a dynamic programming? Four steps

- Structure: Analyze structure of an optimal solution, and thereby choose a space of subproblems (states).

Summary of DP

How to develop a dynamic programming? Four steps

- Structure: Analyze structure of an optimal solution, and thereby choose a space of subproblems (states).
- Recursion: Establish relationship between the optimal value the problem and those of some subproblems

Summary of DP

How to develop a dynamic programming? Four steps

- Structure: Analyze structure of an optimal solution, and thereby choose a space of subproblems (states).
- Recursion: Establish relationship between the optimal value the problem and those of some subproblems
- Bottom-up computation:
  - Compute the optimal values of the smallest subproblems first, save them in the table,

Summary of DP

How to develop a dynamic programming? Four steps

- Structure: Analyze structure of an optimal solution, and thereby choose a space of subproblems (states).
- Recursion: Establish relationship between the optimal value the problem and those of some subproblems
- Bottom-up computation:
  - Compute the optimal values of the smallest subproblems first, save them in the table,
  - Then compute optimal values of larger subproblems, and so on, until the optimal value of the original problem is computed.

Summary of DP

How to develop a dynamic programming? Four steps

- Structure: Analyze structure of an optimal solution, and thereby choose a space of subproblems (states).
- Recursion: Establish relationship between the optimal value the problem and those of some subproblems
- Bottom-up computation:
  - Compute the optimal values of the smallest subproblems first, save them in the table,
  - Then compute optimal values of larger subproblems, and so on, until the optimal value of the original problem is computed.
- Construction of optimal solution: Assemble optimal solution by tracing the computation at the previous step.

Summary of DP

How to develop a dynamic programming? Four steps

- Structure: Analyze structure of an optimal solution, and thereby choose a space of subproblems (states).
- Recursion: Establish relationship between the optimal value the problem and those of some subproblems
- Bottom-up computation:
  - Compute the optimal values of the smallest subproblems first, save them in the table,
  - Then compute optimal values of larger subproblems, and so on, until the optimal value of the original problem is computed.
- Construction of optimal solution: Assemble optimal solution by tracing the computation at the previous step.

Notes:

- Steps 1 and 2 are related.

Summary of DP

How to develop a dynamic programming? Four steps

- Structure: Analyze structure of an optimal solution, and thereby choose a space of subproblems (states).
- Recursion: Establish relationship between the optimal value the problem and those of some subproblems
- Bottom-up computation:
  - Compute the optimal values of the smallest subproblems first, save them in the table,
  - Then compute optimal values of larger subproblems, and so on, until the optimal value of the original problem is computed.
- Construction of optimal solution: Assemble optimal solution by tracing the computation at the previous step.

Notes:

- Steps 1 and 2 are related.
- Step 4 is not always necessary: we sometimes need only the optimal value.

Summary of DP

Dynamic programming (DP) vs Divide-and-Conquer (DC)

Summary of DP

Dynamic programming (DP) vs Divide-and-Conquer (DC)

- Commonalities:
  - Partition the problem into particular subproblems.

Summary of DP

Dynamic programming (DP) vs Divide-and-Conquer (DC)

- Commonalities:
  - Partition the problem into particular subproblems.
  - Solve the subproblems.

Summary of DP

Dynamic programming (DP) vs Divide-and-Conquer (DC)

- Commonalities:
  - Partition the problem into particular subproblems.
  - Solve the subproblems.
  - Combine the solutions to solve the original one.

Summary of DP

Dynamic programming (DP) vs Divide-and-Conquer (DC)

- Commonalities:
  - Partition the problem into particular subproblems.
  - Solve the subproblems.
  - Combine the solutions to solve the original one.
- Differences:
  - DC:
    - Efficient when the subproblems are independent.

Summary of DP

Dynamic programming (DP) vs Divide-and-Conquer (DC)

- Commonalities:
  - Partition the problem into particular subproblems.
  - Solve the subproblems.
  - Combine the solutions to solve the original one.
- Differences:
  - DC:
    - Efficient when the subproblems are independent.
    - Not efficient when subproblems share subsubproblems.

Summary of DP

Dynamic programming (DP) vs Divide-and-Conquer (DC)

- Commonalities:
  - Partition the problem into particular subproblems.
  - Solve the subproblems.
  - Combine the solutions to solve the original one.
- Differences:
  - DC:
    - Efficient when the subproblems are independent.
    - Not efficient when subproblems share subsubproblems.
    - Some subproblems might be solved many times.

Summary of DP

Dynamic programming (DP) vs Divide-and-Conquer (DC)

- Commonalities:
  - Partition the problem into particular subproblems.
  - Solve the subproblems.
  - Combine the solutions to solve the original one.
- Differences:
  - DC:
    - Efficient when the subproblems are independent.
    - Not efficient when subproblems share subsubproblems.
    - Some subproblems might be solved many times.
  - DP:
    - Suitable when subproblems share subsubproblems.

Summary of DP

Dynamic programming (DP) vs Divide-and-Conquer (DC)

- Commonalities:
  - Partition the problem into particular subproblems.
  - Solve the subproblems.
  - Combine the solutions to solve the original one.
- Differences:
  - DC:
    - Efficient when the subproblems are independent.
    - Not efficient when subproblems share subsubproblems.
    - Some subproblems might be solved many times.
  - DP:
    - Suitable when subproblems share subsubproblems.
    - Do each subproblem only once.

Summary of DP

Dynamic programming (DP) vs Divide-and-Conquer (DC)

- Commonalities:
  - Partition the problem into particular subproblems.
  - Solve the subproblems.
  - Combine the solutions to solve the original one.
- Differences:
  - DC:
    - Efficient when the subproblems are independent.
    - Not efficient when subproblems share subsubproblems.
    - Some subproblems might be solved many times.
  - DP:
    - Suitable when subproblems share subsubproblems.
    - Do each subproblem only once.
    - The result is stored in a table in case it is needed elsewhere.
    - DP trades space for

Summary of DP

Dynamic programming (DP) vs Divide-and-Conquer (DC)

- Commonalities:
  - Partition the problem into particular subproblems.
  - Solve the subproblems.
  - Combine the solutions to solve the original one.
- Differences:
  - DC:
    - Efficient when the subproblems are independent.
    - Not efficient when subproblems share subsubproblems.
    - Some subproblems might be solved many times.
  - DP:
    - Suitable when subproblems share subsubproblems.
    - Do each subproblem only once.
    - The result is stored in a table in case it is needed elsewhere.
    - DP trades space for time.

Outline

- Introduction to Part IV
- 0-1 Knapsack Problem
  - Problem Definition
  - A Bruteforce Algorithm
  - A Dynamic Programming Algorithm
  - Analysis of DP Algorithm
- Rod Cutting Problem
  - Problem Definition
  - A Bruteforce Algorithm
  - A Dynamic Programming Algorithm

Rod Cutting

- Input: We are given a rod of length n and a table of prices p_i for i = 1,…, n, where p_i is the price of a rod of length i

Rod Cutting

- Input: We are given a rod of length n and a table of prices p_i for i = 1,…, n, where p_i is the price of a rod of length i
- Goal: to determine the maximum revenue r_n, obtainable by cutting up the rod and selling the pieces

Rod Cutting

- Input: We are given a rod of length n and a table of prices p_i for i = 1,…, n, where p_i is the price of a rod of length i
- Goal: to determine the maximum revenue r_n, obtainable by cutting up the rod and selling the pieces
- Example: Consider n = 4 and p_1=1, p_2=5, p_3=8, p_4=9

Rod Cutting

- Input: We are given a rod of length n and a table of prices p_i for i = 1,…, n, where p_i is the price of a rod of length i
- Goal: to determine the maximum revenue r_n, obtainable by cutting up the rod and selling the pieces
- Example: Consider n = 4 and p_1=1, p_2=5, p_3=8, p_4=9
  - If we do not cut the rod, we can earn p_4 = 9

Rod Cutting

- Input: We are given a rod of length n and a table of prices p_i for i = 1,…, n, where p_i is the price of a rod of length i
- Goal: to determine the maximum revenue r_n, obtainable by cutting up the rod and selling the pieces
- Example: Consider n = 4 and p_1=1, p_2=5, p_3=8, p_4=9
  - If we do not cut the rod, we can earn $p_4 = 9$
  - If we cut it into 4 pieces of length 1 each, we can earn $4 \cdot p_1 = 4$

Rod Cutting

- Input: We are given a rod of length n and a table of prices p_i for i = 1,…, n, where p_i is the price of a rod of length i
- Goal: to determine the maximum revenue r_n, obtainable by cutting up the rod and selling the pieces
- Example: Consider n = 4 and p_1=1, p_2=5, p_3=8, p_4=9
  - If we do not cut the rod, we can earn $p_4 = 9$
  - If we cut it into 4 pieces of length 1 each, we can earn $4 \cdot p_1 = 4$
  - If we cut it into 2 pieces of length 2 each, we can earn $2 \cdot p_2 = 10$

Rod Cutting

- Input: We are given a rod of length n and a table of prices p_i for i = 1,…, n, where p_i is the price of a rod of length i
- Goal: to determine the maximum revenue r_n, obtainable by cutting up the rod and selling the pieces
- Example: Consider n = 4 and p_1=1, p_2=5, p_3=8, p_4=9
  - If we do not cut the rod, we can earn $p_4 = 9$
  - If we cut it into 4 pieces of length 1 each, we can earn $4 \cdot p_1 = 4$
  - If we cut it into 2 pieces of length 2 each, we can earn $2 \cdot p_2 = 10$
  - There are more options, but the maximum revenue is 10

Rod Cutting

- Input: We are given a rod of length n and a table of prices p_i for i = 1,…, n, where p_i is the price of a rod of length i
- Goal: to determine the maximum revenue r_n, obtainable by cutting up the rod and selling the pieces
- Example: Consider n = 4 and p_1=1, p_2=5, p_3=8, p_4=9
  - If we do not cut the rod, we can earn $p_4 = 9$
  - If we cut it into 4 pieces of length 1 each, we can earn $4 \cdot p_1 = 4$
  - If we cut it into 2 pieces of length 2 each, we can earn $2 \cdot p_2 = 10$
  - There are more options, but the maximum revenue is 10
- In general, we can cut the rod of length n in 2^{n-1} different ways, since we have an independent option of cutting, or not cutting, at distance i ($1 \le i \le n - 1$) from the left end

Outline

- Introduction to Part IV
- 0-1 Knapsack Problem
  - Problem Definition
  - A Bruteforce Algorithm
  - A Dynamic Programming Algorithm
  - Analysis of DP Algorithm
- Rod Cutting Problem
  - Problem Definition
  - A Bruteforce Algorithm
  - A Dynamic Programming Algorithm

Optimal Solution

- We can define the maximum revenue r_n in terms of optimal revenues for shorter rods

$$\mathrm{r}_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$

Optimal Solution

- We can define the maximum revenue r_n in terms of optimal revenues for shorter rods

$$\mathrm{r}_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$

  - p_n if we do not cut at all

Optimal Solution

- We can define the maximum revenue r_n in terms of optimal revenues for shorter rods

$$\mathrm{r}_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$

  - p_n if we do not cut at all
  - $r_1 + r_{n-1}$ if we take the sum of optimal revenues for 1 and n−1

Optimal Solution

- We can define the maximum revenue r_n in terms of optimal revenues for shorter rods

$$\mathrm{r}_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

  - p_n if we do not cut at all
  - $r_1 + r_{n-1}$ if we take the sum of optimal revenues for 1 and n−1
  - $r_2 + r_{n-2}$ if we take the sum of optimal revenues for 2 and n−2
  - . . .

Optimal Solution

- We can define the maximum revenue r_n in terms of optimal revenues for shorter rods

$$\mathrm{r}_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

  - p_n if we do not cut at all
  - $r_1 + r_{n-1}$ if we take the sum of optimal revenues for 1 and n−1
  - $r_2 + r_{n-2}$ if we take the sum of optimal revenues for 2 and n−2
  - . . .

- Simpler definition

$$\mathrm{r}_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

Optimal Solution

- We can define the maximum revenue r_n in terms of optimal revenues for shorter rods

$$\mathrm{r}_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$

  - p_n if we do not cut at all
  - $r_1 + r_{n-1}$ if we take the sum of optimal revenues for 1 and n−1
  - $r_2 + r_{n-2}$ if we take the sum of optimal revenues for 2 and n−2
  - . . .

- Simpler definition

$$\mathrm{r}_n = \max_{1 \le i \le n}(p_i + r_{n-i})$$

  - We cut a piece of length i, and a remainder of length n− i

Optimal Solution

- We can define the maximum revenue r_n in terms of optimal revenues for shorter rods

$$\mathrm{r}_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

  - p_n if we do not cut at all
  - $r_1 + r_{n-1}$ if we take the sum of optimal revenues for 1 and n−1
  - $r_2 + r_{n-2}$ if we take the sum of optimal revenues for 2 and n−2
  - . . .

- Simpler definition

$$\mathrm{r}_n = \max_{1 \le i \le n}(p_i + r_{n-i})$$

  - We cut a piece of length i, and a remainder of length n− i
  - Only the remainder, and not the first piece, may be further divided

Recursive Top-down Implementation

Cut-Rod(p,n)

Input: Price list $\boldsymbol{p}$, a rod of length $\boldsymbol{n}$.
Output: Maximum revenue $\boldsymbol{q}$.

```
if n is equal to 0 then
|   return 0;
end
q ← −∞; for i ← 1 to n do
|   q ← max(q, p[i]+Cut-Rod(p, n − i));
end
return q;
```

Recursive Top-down Implementation

Cut-Rod(p,n)

Input: Price list $\boldsymbol{p}$, a rod of length $\boldsymbol{n}$.
Output: Maximum revenue $\boldsymbol{q}$.

```
if n is equal to 0 then
|   return 0;
end
q ← −∞; for i ← 1 to n do
|   q ←max(q, p[i]+Cut-Rod(p, n − i));
end
return q;
```

Cost

Recursive Top-down Implementation

Cut-Rod(p,n)

Input: Price list $\boldsymbol{p}$, a rod of length $\boldsymbol{n}$.
Output: Maximum revenue $\boldsymbol{q}$.
if n *is equal to* 0 **then**
  return 0;
end
$q \leftarrow -\infty$; **for** $i \leftarrow 1$ *to* n **do**
  $q \leftarrow \max(q, p[i] + \text{Cut-Rod}(p, n - i))$;
end
return q;

Cost

- T(n): the total number of calls made to Cut-Rod when called with rod length n

Recursive Top-down Implementation

Cut-Rod(p,n)

```
Input: Price list p, a rod of length n.
Output: Maximum revenue q.
if n is equal to 0 then
 |  return 0;
end
q ← −∞; for i ← 1 to n do
 |  q ←max(q, p[i]+Cut-Rod(p, n − i));
end
return q;
```

Cost

- T(n): the total number of calls made to Cut-Rod when called with rod length n

$$T(n) = \begin{cases} 1 + \sum_{0 \leq j \leq n-1} T(j), & if\ n > 0 \\ 1, & if\ n = 0 \end{cases}$$

Recursive Top-down Implementation

Cut-Rod(p,n)

```
Input: Price list p, a rod of length n.
Output: Maximum revenue q.
if n is equal to 0 then
 |  return 0;
end
q ← −∞; for i ← 1 to n do
 |  q ←max(q, p[i]+Cut-Rod(p, n − i));
end
return q;
```

Cost

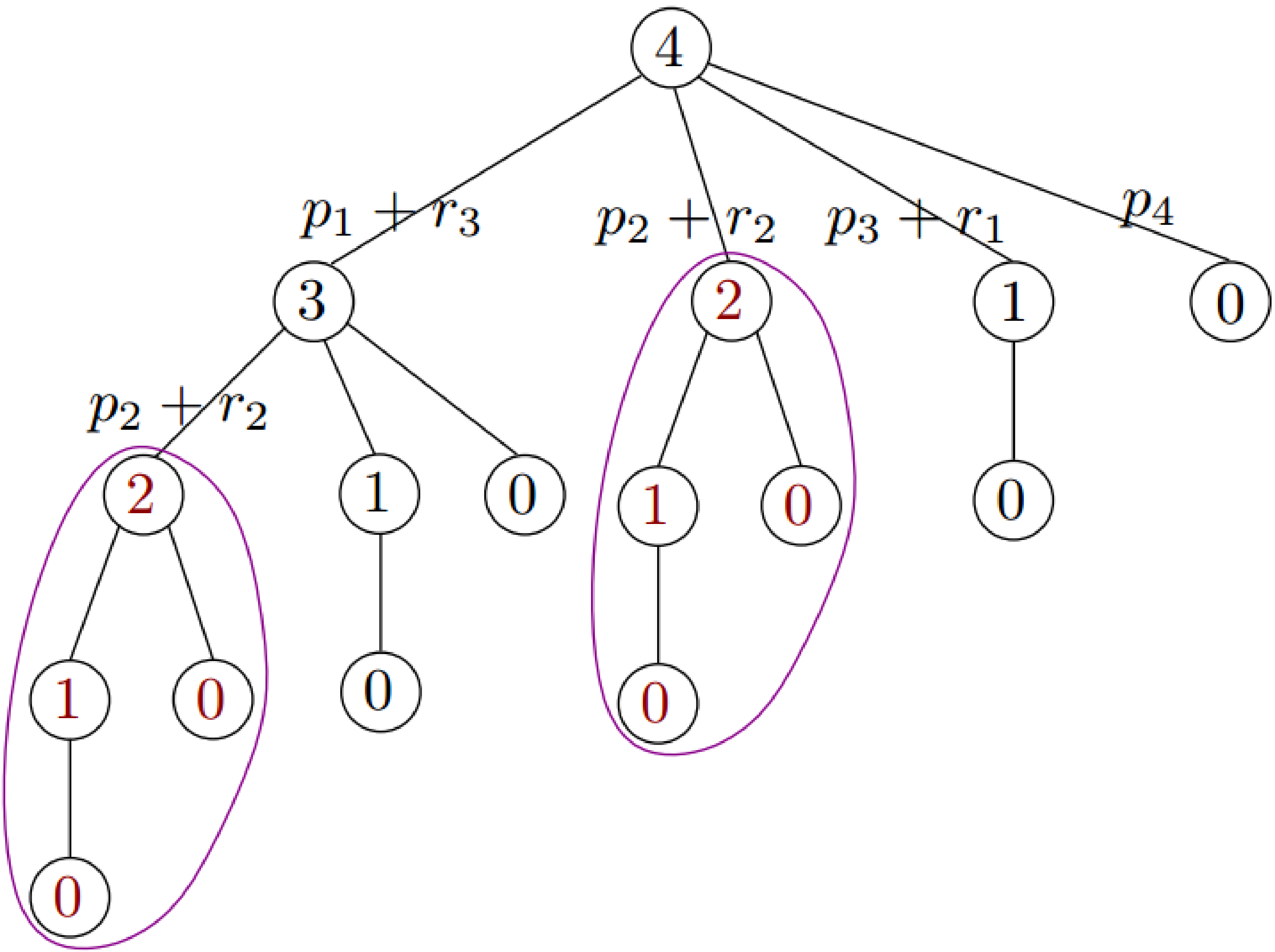
- T(n): the total number of calls made to Cut-Rod when called with rod length n

$$T(n) = \begin{cases} 1 + \sum_{0 \le j \le n-1} T(j), & if\ n > 0 \\ 1, & if\ n = 0 \end{cases}$$

- By induction, we have T(n)=2^n

Explanation of Exponential Cost

- We solve the same subproblem many times



Outline

- Introduction to Part IV
- 0-1 Knapsack Problem
  - Problem Definition
  - A Bruteforce Algorithm
  - A Dynamic Programming Algorithm
  - Analysis of DP Algorithm
- Rod Cutting Problem
  - Problem Definition
  - A Bruteforce Algorithm
  - A Dynamic Programming Algorithm

Concept of DP

- When you solve a subproblem, store the solution

Concept of DP

- When you solve a subproblem, store the solution
    - Next time you find the same subproblem, lookup the solution, instead of solving it again

Concept of DP

- When you solve a subproblem, store the solution
  - Next time you find the same subproblem, lookup the solution, instead of solving it again
  - Use space to save time

Concept of DP

- When you solve a subproblem, store the solution
  - Next time you find the same subproblem, lookup the solution, instead of solving it again
  - Use space to save time
- Two main methodologies: top-down and bottom-up

Concept of DP

- When you solve a subproblem, store the solution
  - Next time you find the same subproblem, lookup the solution, instead of solving it again
  - Use space to save time
- Two main methodologies: top-down and bottom-up
  - Corresponding algorithms have the same asymptotic cost, but bottom-up is usually faster in practice

Concept of DP

- When you solve a subproblem, store the solution
  - Next time you find the same subproblem, lookup the solution, instead of solving it again
  - Use space to save time
- Two main methodologies: top-down and bottom-up
  - Corresponding algorithms have the same asymptotic cost, but bottom-up is usually faster in practice
- Main idea of bottom-up DP

Concept of DP

- When you solve a subproblem, store the solution
  - Next time you find the same subproblem, lookup the solution, instead of solving it again
  - Use space to save time
- Two main methodologies: top-down and bottom-up
  - Corresponding algorithms have the same asymptotic cost, but bottom-up is usually faster in practice
- Main idea of bottom-up DP
  - We sort the subproblems in size and solve the smallest subproblem first

DP Bottom-up Implementation

Bottom-Up-Cut-Rod(p,n)

```
Input: Price list p, a rod of length n.
Output: Maximum revenue q.
Let r[0..n] be a new array;// Array stores the computed optimal values.
r[0] ← 0;
for j ← 0 to n do
    // Consider problems in increasing order of size.
    q ← −∞;
    for i ← 1 to j do
        // To solve a problem of size j, we need to consider all
        decompositions into i and j − i.
        q ←max(q, p[i] + r[j − i]);
    end
    r[j] ← q;
end
return r[n];
```

DP Bottom-up Implementation

Bottom-Up-Cut-Rod(p,n)

Input: Price list $\boldsymbol{p}$, a rod of length $\boldsymbol{n}$.
Output: Maximum revenue $\boldsymbol{q}$.

```
Let r[0..n] be a new array; // Array stores the computed optimal values.
r[0] ← 0;
for j ← 0 to n do
    // Consider problems in increasing order of size.
    q ← −∞;
    for i ← 1 to j do
        // To solve a problem of size j, we need to consider all
        decompositions into i and j − i.
        q ← max(q, p[i] + r[j − i]);
    end
    r[j] ← q;
end
return r[n];
```

- Cost: $O(n^2)$

DP Bottom-up Implementation

Bottom-Up-Cut-Rod(p,n)

```
Input: Price list p, a rod of length n.
Output: Maximum revenue q.
Let r[0..n] be a new array;// Array stores the computed optimal values.
r[0] ← 0;
for j ← 0 to n do
  | // Consider problems in increasing order of size.
  | q ← −∞;
  | for i ← 1 to j do
  |   | // To solve a problem of size j, we need to consider all
  |   | decompositions into i and j − i.
  |   | q ←max(q, p[i] + r[j − i]);
  | end
  | r[j] ← q;
end
return r[n];
```

- Cost: $O(n^2)$
  - The outer loop computes r[1], r[2],…, r[n] in this order

DP Bottom-up Implementation

Bottom-Up-Cut-Rod(p,n)

```
Input: Price list p, a rod of length n.
Output: Maximum revenue q.
Let r[0..n] be a new array;// Array stores the computed optimal values.
r[0] ← 0;
for j ← 0 to n do
    // Consider problems in increasing order of size.
    q ← −∞;
    for i ← 1 to j do
        // To solve a problem of size j, we need to consider all
        decompositions into i and j − i.
        q ←max(q, p[i] + r[j − i]);
    end
    r[j] ← q;
end
return r[n];
```

- Cost: $O(n^2)$
  - The outer loop computes r[1], r[2],…, r[n] in this order
  - To compute r[j], the inner loop uses all values r[0], r[1],…, r[j − 1] (i.e., r[j – i] for $1 \leq i \leq j$)

Extended Implementation

Extended-Bottom-Up-Cut-Rod(p,n)

```
Input: Price list p, a rod of length n.
Output: Maximum revenue q and sizes of pieces.
Let r[0..n] and s[0..n] be two new arrays;
r[0] ← 0;
for j ← 0 to n do
    q ← −∞;
    for i ← 1 to j do
        // Solve problem of size j.
        if q < p[i] + r[j − i] then
            q ← p[i] + r[j − i];
            s[j] ← i;// Store the size of the first piece.
        end
    end
    r[j] ← q;
end
while n > 0 do
    Output s[n];
    n ← n − s[n];
end
```

Example of Optimal Solution Construction

n=10

$\boldsymbol{k}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$\boldsymbol{p}[\boldsymbol{k}]$	1	5	8	9	10	17	17	20	24	26

$\boldsymbol{k}$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$r[k]$											
$s[k]$											

Example of Optimal Solution Construction

n=10

$\boldsymbol{k}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$\boldsymbol{p}[\boldsymbol{k}]$	1	5	8	9	10	17	17	20	24	26

$\boldsymbol{k}$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$r[k]$	0										
$s[k]$											

Intialization

Example of Optimal Solution Construction

n=10

k	1	2		4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

$p[i]$

j = 1

i	1
	1

$p[i] + r[j - i]$

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0										
$s[k]$											

$r[j - i]$

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 1

i	1
	1

$max\{p[i] + r[j - i]\}$

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0										
$s[k]$											

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 1

i	1
	1

$max\{p[i] + r[j - i]\}$

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1									
$s[k]$		1									

Example of Optimal Solution Construction

n=10

$\boldsymbol{k}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$\boldsymbol{p[k]}$	1	5	8	9	10	17	17	20	24	26

j = 2

$\boldsymbol{i}$	**1**	**2**
	2	5

$\boldsymbol{k}$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$r[k]$	0	1									
$s[k]$		1									

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 2

i	1	2
	2	5

$\boldsymbol{max\{p[i] + r[j-i]\}}$

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1									
$s[k]$		1									

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 2

i	1	2
	2	5

$max\{p[i] + r[j - i]\}$

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5								
$s[k]$		1	2								

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 3

i	1	2	3
	6	6	8

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5								
$s[k]$		1	2								

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 3

i	1	2	3
	6	6	8

$\boldsymbol{max\{p[i] + r[j-i]\}}$

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5								
$s[k]$		1	2								

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 3

i	1	2	3
	6	6	8

$max\{p[i] + r[j - i]\}$

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5	8							
$s[k]$		1	2	3							

Example of Optimal Solution Construction

n=10

$\boldsymbol{k}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$\boldsymbol{p[k]}$	1	5	8	9	10	17	17	20	24	26

j = 4

$\boldsymbol{i}$	**1**	**2**	**3**	**4**
	9	10	9	9

$\boldsymbol{k}$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$r[k]$	0	1	5	8							
$s[k]$		1	2	3							

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 4

i	1	2	3	4
	9	10	9	

$$max\{p[i] + r[j - i]\}$$

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5	8							
$s[k]$		1	2	3							

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 4

i	1	2	3	4
	9	10	9	

$$max\{p[i] + r[j-i]\}$$

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5	8	10						
$s[k]$		1	2	3	2						

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 5

i	1	2	3	4	5
	11	13	13	10	10

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5	8	10						
$s[k]$		1	2	3	2						

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 5

i	1	2	3	4	5
	11	13	13		

$$max\{p[i] + r[j - i]\}$$

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5	8	10						
$s[k]$		1	2	3	2						

Example of Optimal Solution Construction

n=10

$\boldsymbol{k}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$\boldsymbol{p[k]}$	1	5	8	9	10	17	17	20	24	26

j = 5

$\boldsymbol{i}$	**1**	**2**	**3**	**4**	**5**
	11	13	[illegible]		

$\boldsymbol{max\{p[i] + r[j-i]\}}$

$\boldsymbol{k}$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$r[k]$	0	1	5	8	10	13					
$s[k]$		1	2	3	2	2					

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 6

i	1	2	3	4	5	6
	14	15	16	14	11	17

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5	8	10	13					
$s[k]$		1	2	3	2	2					

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 6

i	1	2	3	4	5	6
	14	15	16	14	11	$\boxed{17}$

$max\{p[i] + r[j - i]\}$

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5	8	10	13					
$s[k]$		1	2	3	2	2					

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 6

i	1	2	3	4	5	6
	14	15	16	14	11	17

$max\{p[i] + r[j - i]\}$

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5	8	10	13	17				
$s[k]$		1	2	3	2	2	6				

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 7

i	1	2	3	4	5	6	7
	18	18	18	17	15	18	17

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5	8	10	13	17				
$s[k]$		1	2	3	2	2	6				

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 7

i	1	2	3	4	5	6	7
	18	18	[illegible]	[illegible]	[illegible]	[illegible]	17

$$\boldsymbol{max\{p[i] + r[j-i]\}}$$

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5	8	10	13	17				
$s[k]$		1	2	3	2	2	6				

Example of Optimal Solution Construction

n=10

$\boldsymbol{k}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$\boldsymbol{p[k]}$	1	5	8	9	10	17	17	20	24	26

j = 7

$\boldsymbol{i}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**
	$\boxed{18}$	18	[illegible]	[illegible]	[illegible]	[illegible]	17

$$\boldsymbol{max\{p[i] + r[j-i]\}}$$

$\boldsymbol{k}$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$r[k]$	0	1	5	8	10	13	17	18			
$s[k]$		1	2	3	2	2	6	1			

Example of Optimal Solution Construction

n=10

$\boldsymbol{k}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$\boldsymbol{p[k]}$	1	5	8	9	10	17	17	20	24	26

j = 8

$\boldsymbol{i}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	19	22	21	19	18	22	18	20

$\boldsymbol{k}$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$r[k]$	0	1	5	8	10	13	17	18			
$s[k]$		1	2	3	2	2	6	1			

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 8

i	1	2	3	4	5	6	7	8
	19	22	21					20

$max\{p[i] + r[j-i]\}$

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5	8	10	13	17	18			
$s[k]$		1	2	3	2	2	6	1			

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 8

i	1	2	3	4	5	6	7	8
	19	$\boxed{22}$	21					20

$max\{p[i] + r[j-i]\}$

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

Example of Optimal Solution Construction

n=10

$\boldsymbol{k}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$\boldsymbol{p[k]}$	1	5	8	9	10	17	17	20	24	26

j = 9

$\boldsymbol{i}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	23	23	25	22	20	25	22	21	24

$\boldsymbol{k}$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$r[k]$	0	1	5	8	10	13	17	18	22		
$s[k]$		1	2	3	2	2	6	1	2		

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 9

i	1	2	$\boxed{3}$	4	5	6	7	8	9
	23	23	$\boxed{25}$	22	[illegible]	[illegible]	[illegible]	[illegible]	24

$max\{p[i] + r[j - i]\}$

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

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 9

i	1	2	$\boxed{3}$	4	5	6	7	8	9
	23	23	$\boxed{25}$	22	[illegible]	[illegible]	[illegible]	[illegible]	24

$max\{p[i] + r[j-i]\}$

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

Example of Optimal Solution Construction

n=10

$\boldsymbol{k}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$\boldsymbol{p[k]}$	1	5	8	9	10	17	17	20	24	26

j = 10

$\boldsymbol{i}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
	26	27	26	26	23	27	25	25	25	26

$\boldsymbol{k}$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$r[k]$	0	1	5	8	10	13	17	18	22	25	
$s[k]$		1	2	3	2	2	6	1	2	3	

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 10

i	1	2	3	4	5	6	7	8	9	10
	26	27	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	25	25	26

$max\{p[i] + r[j-i]\}$

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

Example of Optimal Solution Construction

n=10

$\boldsymbol{k}$	1	2	3	4	5	6	7	8	9	10
$\boldsymbol{p[k]}$	1	5	8	9	10	17	17	20	24	26

j = 10

$\boldsymbol{i}$	1	2	3	4	5	6	7	8	9	10
	26	27	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	25	25	26

$\boldsymbol{max\{p[i] + r[j-i]\}}$

$\boldsymbol{k}$	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5	8	10	13	17	18	22	25	27
$s[k]$		1	2	3	2	2	6	1	2	3	2

Example of Optimal Solution Construction

n=10

$\boldsymbol{k}$	1	2	3	4	5	6	7	8	9	10
$\boldsymbol{p}[\boldsymbol{k}]$	1	5	8	9	10	17	17	20	24	26

j = 10

$\boldsymbol{i}$	1	2	3	4	5	6	7	8	9	10
	26	27	26	26	23	27	25	25	25	26

$\boldsymbol{k}$	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5	8	10	13	17	18	22	25	27
$s[k]$		1	2	3	2	2	6	1	2	3	2

Max revenue = r[10] = 27

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 10

i	1	2	3	4	5	6	7	8	9	10
	26	27	26	26	23	27	25	25	25	26

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5	8	10	13	17	18	22	25	27
$s[k]$		1	2	3	2	2	6	1	2	3	2

Max revenue = r[10] = 27

Pieces of rods = { 2,

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 10

i	1	2	3	4	5	6	7	8	9	10
	26	27	26	26	23	27	25	25	25	26

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5	8	10	13	17	18	22	25	27
$s[k]$		1	2	3	2	2	6	1	2	3	2

Max revenue = r[10] = 27

Pieces of rods = { 2, 2, 6

Example of Optimal Solution Construction

n=10

k	1	2	3	4	5	6	7	8	9	10
$p[k]$	1	5	8	9	10	17	17	20	24	26

j = 10

i	1	2	3	4	5	6	7	8	9	10
	26	27	26	26	23	27	25	25	25	26

k	0	1	2	3	4	5	6	7	8	9	10
$r[k]$	0	1	5	8	10	13	17	18	22	25	27
$s[k]$		1	2	3	2	2	6	1	2	3	2

Max revenue = r[10] = 27

Pieces of rods = { 2, 2, 6}


dank u
ju faleminderit
Tack
谢谢
Tak
mulţumesc
Asante
kiitos
Gracias
Salamat!
Terima kasih
Aliquam
Merci
Dankie
Obrigado
ありがとう
köszönöm
grazie
Aliquam
Go raibh maith agat
děkuji
Thank you
gam