

# Dynamic Programming

Hengfeng Wei

hfwei@nju.edu.cn

June 11 – June 19, 2017



# Dynamic Programming

- 1 Overview
- 2 1D DP
- 3 2D DP
- 4 3D DP
- 5 DP on Graphs
- 6 The Knapsack Problem
- 7 Summary

# What is DP?

DP  $\approx$  “brute force”

DP  $\approx$  “smart scheduling of subproblems”

DP  $\approx$  “shortest/longest paths in some DAG”

# What is DP?

DP  $\approx$  “smarter brute force”

DP  $\approx$  “smart scheduling of subproblems”

DP  $\approx$  “shortest/longest paths in some DAG”

# What is not DP?

Programming = Planning

# What is not DP?

Programming = Planning

Programming  $\neq$  Coding

(Richard Bellman, 1940s)

# Steps for applying DP

1. Define subproblems
  - ▶ # of subproblems
2. Set the goal
3. Define the recurrence
  - ▶ larger subproblem  $\leftarrow$  # smaller subproblems
  - ▶ init. conditions
4. Write pseudo-code: fill “table” in topo. order
5. Analyze Time/Space complexity
6. Extract the optimal solution

# Common subproblems in DP

1D subproblems:

**Input:**  $x_1, x_2, \dots, x_n$  (array, sequence, string)

**Subproblems:**  $x_1, x_2, \dots, x_i$  (prefix/suffix)

**#:**  $\Theta(n)$



# Common subproblems in DP

1D subproblems:

**Input:**  $x_1, x_2, \dots, x_n$  (array, sequence, string)

**Subproblems:**  $x_1, x_2, \dots, x_i$  (prefix/suffix)

**#:**  $\Theta(n)$

**Examples:** Fib, Maximum-sum subarray, Longest increasing subsequence, Highway restaurants, Text justification

# Common subproblems in DP

2D subproblems:

1. Input:  $x_1, x_2, \dots, x_m; \quad y_1, y_2, \dots, y_n$

Subproblems:  $x_1, x_2, \dots, x_i; \quad y_1, y_2, \dots, y_j$

#:  $\Theta(mn)$

# Common subproblems in DP

2D subproblems:

1. Input:  $x_1, x_2, \dots, x_m; \quad y_1, y_2, \dots, y_n$

Subproblems:  $x_1, x_2, \dots, x_i; \quad y_1, y_2, \dots, y_j$

#:  $\Theta(mn)$

Examples: Edit distance, Longest common subsequence

# Common subproblems in DP

2D subproblems:

1. Input:  $x_1, x_2, \dots, x_m; \quad y_1, y_2, \dots, y_n$

Subproblems:  $x_1, x_2, \dots, x_i; \quad y_1, y_2, \dots, y_j$

#:  $\Theta(mn)$

Examples: Edit distance, Longest common subsequence

2. Input:  $x_1, x_2, \dots, x_n$

Subproblems:  $x_i, \dots, x_j$

#:  $\Theta(n^2)$

# Common subproblems in DP

2D subproblems:

1. Input:  $x_1, x_2, \dots, x_m; \quad y_1, y_2, \dots, y_n$

Subproblems:  $x_1, x_2, \dots, x_i; \quad y_1, y_2, \dots, y_j$

#:  $\Theta(mn)$

Examples: Edit distance, Longest common subsequence

2. Input:  $x_1, x_2, \dots, x_n$

Subproblems:  $x_i, \dots, x_j$

#:  $\Theta(n^2)$

Examples: Matrix chain multiplication, Optimal BST

# Common subproblems in DP

3D subproblems:

- ▶ Floyd-Warshall algorithm

$$d(i, j, k) = \min\{d(i, j, k - 1), d(i, k, k - 1) + d(k, j, k - 1)\}$$

# Common subproblems in DP

3D subproblems:

- ▶ Floyd-Warshall algorithm

$$d(i, j, k) = \min\{d(i, j, k-1), d(i, k, k-1) + d(k, j, k-1)\}$$

DP on graphs:

1. On rooted tree

**Subproblems:** rooted subtrees

2. On DAG

**Subproblems:** nodes after/before in the topo. order

# Common subproblems in DP

3D subproblems:

- ▶ Floyd-Warshall algorithm

$$d(i, j, k) = \min\{d(i, j, k-1), d(i, k, k-1) + d(k, j, k-1)\}$$

DP on graphs:

1. On rooted tree

**Subproblems:** rooted subtrees

2. On DAG

**Subproblems:** nodes after/before in the topo. order

Knapsack problem:

- ▶ Subset sum problem, change-making problem



# Common subproblems in DP

And Others . . .

# Recurrences in DP

Make choices by asking yourself the right question:

1. Binary choice
  - ▶ whether ...
2. Multi-way choices
  - ▶ where to ...
  - ▶ which one ...

# Dynamic Programming

- 1 Overview
- 2 1D DP**
- 3 2D DP
- 4 3D DP
- 5 DP on Graphs
- 6 The Knapsack Problem
- 7 Summary

$$f(S(n)) = 1$$

$$f(S(n)) = 1 \text{ (Problem 7.2)}$$

$$f(n) = \begin{cases} n - 1 & \text{if } n \in \mathbb{Z}^+ \\ n/2 & \text{if } n \% 2 = 0 \\ n/3 & \text{if } n \% 3 = 0 \end{cases}$$

$S(n)$  : minimum number of steps taking  $n$  to 1.

$$f(S(n)) = 1$$

$$f(S(n)) = 1 \text{ (Problem 7.2)}$$

$$f(n) = \begin{cases} n - 1 & \text{if } n \in \mathbb{Z}^+ \\ n/2 & \text{if } n \% 2 = 0 \\ n/3 & \text{if } n \% 3 = 0 \end{cases}$$

$S(n)$  : minimum number of steps taking  $n$  to 1.

$S(i)$  : minimum number of steps taking  $i$  to 1

$$f(S(n)) = 1$$

$$f(S(n)) = 1 \text{ (Problem 7.2)}$$

$$f(n) = \begin{cases} n - 1 & \text{if } n \in \mathbb{Z}^+ \\ n/2 & \text{if } n \% 2 = 0 \\ n/3 & \text{if } n \% 3 = 0 \end{cases}$$

$S(n)$  : minimum number of steps taking  $n$  to 1.

$S(i)$  : minimum number of steps taking  $i$  to 1

$$S(i) = 1 + \min\{S(i - 1), S(i/2)(\text{if } i \% 2 = 0), S(i/3)(\text{if } i \% 3 = 0)\}$$

$$S(1) = 0$$

$$f(S(n)) = 1$$

Collatz ( $3n + 1$ ) conjecture:

$$f(n) = \begin{cases} n/2 & \text{if } n \% 2 = 0 \\ 3n + 1 & \text{if } n \% 2 = 1 \end{cases}$$

$$f^*(n) = 1?$$

$$f(S(n)) = 1$$

Collatz ( $3n + 1$ ) conjecture:

$$f(n) = \begin{cases} n/2 & \text{if } n \% 2 = 0 \\ 3n + 1 & \text{if } n \% 2 = 1 \end{cases}$$

$$f^*(n) = 1?$$

*“Mathematics may not be ready for such problems.”*

— Paul Erdős



# Longest increasing subsequence

## Longest increasing subsequence (Problem 7.3)

- ▶ Given an integer array  $A[1 \dots n]$
- ▶ To find (the length of) a longest increasing subsequence.

$$5, 2, 8, 6, 3, 6, 9, 7 \implies 2, 3, 6, 9$$

# Longest increasing subsequence

Subproblem:  $L(i)$  : the length of the LIS of  $A[1 \dots i]$

Goal:  $L(n)$

# Longest increasing subsequence

Subproblem:  $L(i)$  : the length of the LIS of  $A[1 \dots i]$

Goal:  $L(n)$

Make choice: whether  $A[i] \in LIS[1 \dots i]$ ?

Recurrence:

$$L(i) = \max\{L(i-1), 1 + \max_{j < i \wedge A[j] \leq A[i]} L(j)\}$$

# Longest increasing subsequence

Subproblem:  $L(i)$  : the length of the LIS of  $A[1 \dots i]$

Goal:  $L(n)$

Make choice: whether  $A[i] \in LIS[1 \dots i]$ ?

Recurrence:

$$L(i) = \max\{L(i-1), 1 + \max_{j < i \wedge A[j] \leq A[i]} L(j)\}$$

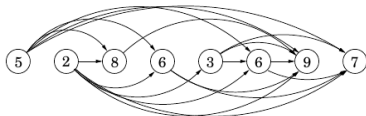
Init:

$$L(0) = 0$$

Time:

$$O(n^2) = \Theta(n) \cdot O(n)$$

# Longest increasing subsequence



Longest path distance in the DAG!

# Maximum-sum subarray

## Maximum-sum subarray (Google Interview)

- ▶ Array  $A[1 \cdots n]$ ,  $a_i \geq 0$
- ▶ To find (the sum of) a maximum-sum subarray of  $A$

$$A[-2, 1, -3, 4, -1, 2, 1, -5, 4] \implies [4, -1, 2, 1]$$

# Maximum-sum subarray

## Maximum-sum subarray (Google Interview)

- ▶ Array  $A[1 \cdots n]$ ,  $a_i \geq 0$
- ▶ To find (the sum of) a maximum-sum subarray of  $A$

$$A[-2, 1, -3, 4, -1, 2, 1, -5, 4] \implies [4, -1, 2, 1]$$

**Subproblem:**  $MSS[i]$ : sum of an MS[i] of  $A[1 \cdots i]$

**Goal:**  $mss = MSS[n]$

# Maximum-sum subarray

## Maximum-sum subarray (Google Interview)

- ▶ Array  $A[1 \cdots n]$ ,  $a_i \geq 0$
- ▶ To find (the sum of) a maximum-sum subarray of  $A$

$$A[-2, 1, -3, 4, -1, 2, 1, -5, 4] \implies [4, -1, 2, 1]$$

**Subproblem:**  $MSS[i]$ : sum of an MS[i] of  $A[1 \cdots i]$

**Goal:**  $mss = MSS[n]$

**Make choice:** Is  $a_i \in MS[i]$ ?

**Recurrence:**

$$MSS[i] = \max\{MSS[i-1], ???\}$$



# Maximum-sum subarray

Subproblem:  $MSS[i]$ : sum of an MS[i] *ending with*  $a_i$

Goal:  $mss = \max_{1 \leq i \leq n} MSS[i]$

# Maximum-sum subarray

Subproblem:  $MSS[i]$ : sum of an MS[i] *ending with*  $a_i$

Goal:  $mss = \max_{1 \leq i \leq n} MSS[i]$

Make choice: Where does the MS[i] start?

Recurrence:

$$MSS[i] = \max\{MSS[i-1] + a_i, a_i\} \text{ (Proof!)}$$

# Maximum-sum subarray

Subproblem:  $MSS[i]$ : sum of an MS[i] *ending with*  $a_i$

Goal:  $mss = \max_{1 \leq i \leq n} MSS[i]$

Make choice: Where does the MS[i] start?

Recurrence:

$$MSS[i] = \max\{MSS[i-1] + a_i, a_i\} \text{ (Proof!)}$$

Init:

$$MSS[0] = 0$$

# Maximum-sum subarray

Subproblem:  $MSS[i]$ : sum of an MS[i] *ending with*  $a_i$

Goal:  $mss = \max_{1 \leq i \leq n} MSS[i]$

Make choice: Where does the MS[i] start?

Recurrence:

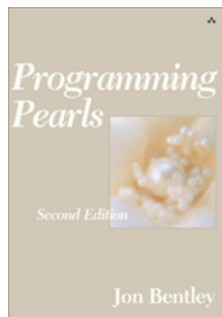
$$MSS[i] = \max\{MSS[i-1] + a_i, a_i\} \text{ (Proof!)}$$

Init:

$$MSS[0] = 0$$

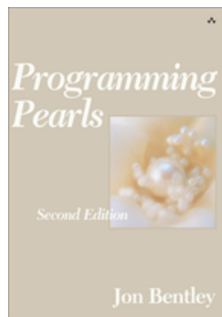
Time:  $\Theta(n)$

# Maximum-sum subarray



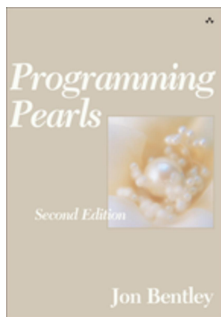
Ulf Grenander  $O(n^3) \implies O(n^2)$

# Maximum-sum subarray



Ulf Grenander  $O(n^3) \implies O(n^2)$   
Michael Shamos  $O(n \log n)$ , onenight

# Maximum-sum subarray

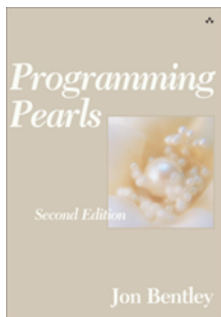


Ulf Grenander  $O(n^3) \implies O(n^2)$

Michael Shamos  $O(n \log n)$ , onenight

Jon Bentley Conjecture:  $\Omega(n \log n)$

# Maximum-sum subarray



Ulf Grenander  $O(n^3) \implies O(n^2)$

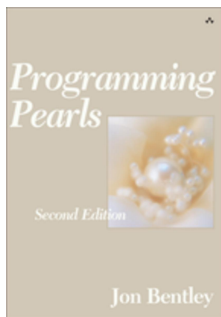
Michael Shamos  $O(n \log n)$ , onenight

Jon Bentley Conjecture:  $\Omega(n \log n)$

Michael Shamos Carnegie Mellon seminar



# Maximum-sum subarray



Ulf Grenander  $O(n^3) \implies O(n^2)$

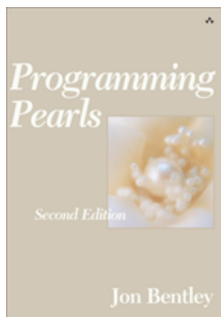
Michael Shamos  $O(n \log n)$ , onenight

Jon Bentley Conjecture:  $\Omega(n \log n)$

Michael Shamos Carnegie Mellon seminar

Jay Kadane  $O(n)$ ,

# Maximum-sum subarray



Ulf Grenander  $O(n^3) \implies O(n^2)$

Michael Shamos  $O(n \log n)$ , onenight

Jon Bentley Conjecture:  $\Omega(n \log n)$

Michael Shamos Carnegie Mellon seminar

Jay Kadane  $O(n)$ ,  $\leq 1$  minute

# Maximum-product subarray

## Maximum-product subarray (Problem 7.4)

- ▶ Array  $A[1 \dots n]$
- ▶ Find maximum-product subarray of  $A$

(1)  $a_i \in \mathbb{N}$

(2)  $a_i \in \mathbb{Z}$

(3)  $a_i \in \mathbb{R}$

# Maximum-product subarray

## Maximum-product subarray (Problem 7.4)

- ▶ Array  $A[1 \dots n]$
- ▶ Find maximum-product subarray of  $A$

(1)  $a_i \in \mathbb{N}$

(2)  $a_i \in \mathbb{Z}$

(3)  $a_i \in \mathbb{R}$

sum vs. product

# Maximum-product subarray

Subproblem:  $\text{MaxP}[i], \text{MinP}[i]$

		$\frac{1}{2}$	4	-2	5	$-\frac{1}{5}$	8
$\text{MaxP}[i]$	1	$\frac{1}{2}$	4	-2	5	8	64
$\text{MinP}[i]$	1	$\frac{1}{2}$	2	-8	-40	-1	-8

$$\text{MaxP}[i] = \max\{\text{MaxP}[i-1] \cdot a_i, \text{MinP}[i-1] \cdot a_i, a_i\}$$

$$\text{MinP}[i] = \min\{\text{MaxP}[i-1] \cdot a_i, \text{MinP}[i-1] \cdot a_i, a_i\}$$

# Reconstructing string

## Reconstructing string (Problem 7.9)

- ▶ String  $S[1 \cdots n]$
- ▶ Dict for *lookup*:

$$\text{dict}(w) = \begin{cases} \text{true} & \text{if } w \text{ is a valid word} \\ \text{false} & \text{o.w.} \end{cases}$$

- ▶ Is  $S[1 \cdots n]$  valid (reconstructed as a sequence of valid words)?

# Reconstructing string

Subproblem:  $V[i]$ : Is  $S[1 \dots i]$  valid?

Goal:  $V[n]$

# Reconstructing string

Subproblem:  $V[i]$ : Is  $S[1 \dots i]$  valid?

Goal:  $V[n]$

Make choice: Where does the last word start?

Recurrence:

$$V[i] = \bigvee_{j=1 \dots i} (V[j-1] \wedge \text{dict}(S[j \dots i]))$$



# Reconstructing string

Subproblem:  $V[i]$ : Is  $S[1 \dots i]$  valid?

Goal:  $V[n]$

Make choice: Where does the last word start?

Recurrence:

$$V[i] = \bigvee_{j=1 \dots i} (V[j-1] \wedge \text{dict}(S[j \dots i]))$$

Init:

$$V[0] = \text{true}$$

Time:

$$O(n^2) = \Theta(n) \cdot O(n)$$

# Hotel along a trip

## Hotel along a trip (Problem 7.15)

- ▶ Hotel sequence (distance):  $a_0 = 0, a_1, \dots, a_n$
- ▶  $a_0 \rightsquigarrow a_n$
- ▶ Stop at only hotels
- ▶ Cost:  $(200 - x)^2$
- ▶ To minimize overall cost

# Hotel along a trip

Subproblem:  $C[i]$ : minimum cost when the destination is  $a_i$

Goal:  $C[n]$

# Hotel along a trip

Subproblem:  $C[i]$ : minimum cost when the destination is  $a_i$

Goal:  $C[n]$

Make choice: What is the last but one hotel  $a_j$  to stop?

Recurrence:

$$C[i] = \min_{0 \leq j < i} \{C[j] + (200 - (a_i - a_j))^2\}$$

# Hotel along a trip

Subproblem:  $C[i]$ : minimum cost when the destination is  $a_i$

Goal:  $C[n]$

Make choice: What is the last but one hotel  $a_j$  to stop?

Recurrence:

$$C[i] = \min_{0 \leq j < i} \{C[j] + (200 - (a_i - a_j))^2\}$$

Init:

$$C[0] = 0$$

Time:

$$O(n^2) = \Theta(n) \cdot O(n)$$

# Highway restaurants

## Highway restaurants (Problem 7.16)

- ▶ Locations:  $L[1 \dots n]$
- ▶ Profits:  $P[1 \dots n]$
- ▶ Any two hotels should be  $\geq k$  miles apart
- ▶ To maximize the total profit

Subproblem:  $T[i]$ : max profit achievable using only  $L[1 \dots i]$

Goal:  $T[n]$

# Highway restaurants

## Highway restaurants (Problem 7.16)

- ▶ Locations:  $L[1 \dots n]$
- ▶ Profits:  $P[1 \dots n]$
- ▶ Any two hotels should be  $\geq k$  miles apart
- ▶ To maximize the total profit

Subproblem:  $T[i]$ : max profit achievable using only  $L[1 \dots i]$

Goal:  $T[n]$

Make choice: Whether to open a restaurant at  $L_i$ ?

Recurrence:

$$T[i] = \max\{T[i-1], P_i + T[\text{prev}(i)]\}$$

$$\text{prev}(i) = \max\{j \mid j < i \wedge L_i - L_j \geq k\}$$

# Highway restaurants

## Highway restaurants (Problem 7.16)

- ▶ Locations:  $L[1 \dots n]$
- ▶ Profits:  $P[1 \dots n]$
- ▶ Any two hotels should be  $\geq k$  miles apart
- ▶ To maximize the total profit

Subproblem:  $T[i]$ : max profit achievable using only  $L[1 \dots i]$

Goal:  $T[n]$

Make choice: Whether to open a restaurant at  $L_i$ ?

Recurrence:

$$T[i] = \max\{T[i-1], P_i + T[\text{prev}(i)]\}$$

$$\text{prev}(i) = \max\{j \mid j < i \wedge L_i - L_j \geq k\}$$

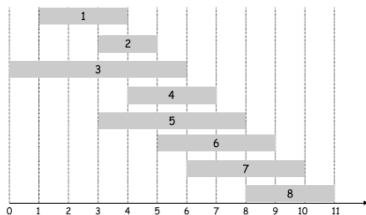
Init:  $T[0] = 0$



# Weighted interval/class scheduling

## Weighted interval/class scheduling (Problem 7.14)

- ▶ Classes:  $\mathcal{C} = \{c_1, c_2, \dots, c_n\}$   $c_i \triangleq \langle g_i, s_i, f_i \rangle$
- ▶ Choosing non-conflicting classes to maximize your grades



sort  $\mathcal{C}$  by finishing time.

# Weighted interval/class scheduling

Greedy algorithms fail:

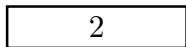
2

1

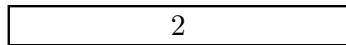
By finishing time.

# Weighted interval/class scheduling

Greedy algorithms fail:



By finishing time.



By weights.

# Weighted interval/class scheduling

Subproblem:  $G[i]$ : the maximal grades obtained from  $\{c_1, c_2, \dots, c_i\}$

Goal:  $G[n]$

# Weighted interval/class scheduling

**Subproblem:**  $G[i]$ : the maximal grades obtained from  $\{c_1, c_2, \dots, c_i\}$

**Goal:**  $G[n]$

**Make choice:** Choose  $c_i$  or not?

**Recurrence:**

$$G[i] = \max\{G[i-1], G[\text{prev}(i)] + g_i\}$$

$$\text{prev}(i) = \max\{j \mid j < i \wedge c_i \cap c_j = \emptyset\}$$

# Weighted interval/class scheduling

**Subproblem:**  $G[i]$ : the maximal grades obtained from  $\{c_1, c_2, \dots, c_i\}$

**Goal:**  $G[n]$

**Make choice:** Choose  $c_i$  or not?

**Recurrence:**

$$G[i] = \max\{G[i-1], G[\text{prev}(i)] + g_i\}$$

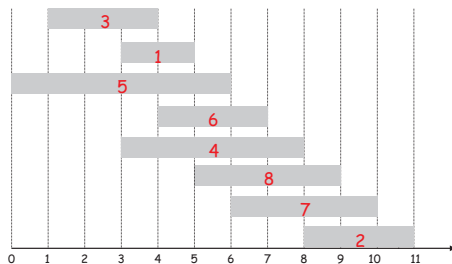
$$\text{prev}(i) = \max\{j \mid j < i \wedge c_i \cap c_j = \emptyset\}$$

**Init:**  $G[0] = 0$

**Time:**  $O(n \log n) + T(p(i)) + O(n) \cdot O(1)$

# Weighted interval/class scheduling

Why is ordering necessary?

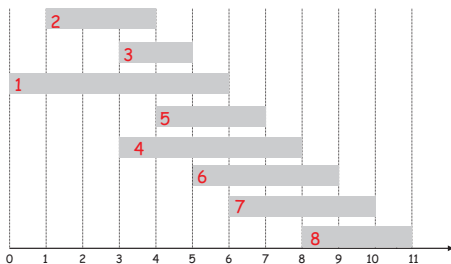


$$G[7] = \max\{G[6], G[\{1, 3, 5\}] + g_7\}$$

subproblems changed: all  $O(2^n)$  subsets

# Weighted interval/class scheduling

What about sorting by starting time?



$$G[6] = \max\{G[5], G[\{2, 3\}] + g_6\}$$

subproblems changed: all  $O(2^n)$  subsets



# Dynamic Programming

- 1 Overview
- 2 1D DP
- 3 2D DP**
- 4 3D DP
- 5 DP on Graphs
- 6 The Knapsack Problem
- 7 Summary

# Longest common subsequence

LCS: longest common subsequence (Problem 7.5)

$$X = X_1 \cdots X_m \quad Y = Y_1 \cdots Y_n$$

(1) Find (the length of) an LCS of  $X$  and  $Y$

$$X = \langle A, B, C, B, D, A, B \rangle$$

$$Y = \langle B, D, C, A, B, A \rangle$$

$$Z = \langle B, C, B, A \rangle$$

# Longest common subsequence

**Subproblem:**  $L[i, j]$ : the length of an LCS of  $X[1 \cdots i]$  and  $Y[1 \cdots j]$

**Goal:**  $L[m, n]$

# Longest common subsequence

**Subproblem:**  $L[i, j]$ : the length of an LCS of  $X[1 \dots i]$  and  $Y[1 \dots j]$

**Goal:**  $L[m, n]$

**Make choice:** Is  $X_i = Y_j$ ?

**Recurrence:** (Proof!)

$$L[i, j] = \begin{cases} L[i-1, j-1] + 1 & \text{if } X_i = Y_j \\ \max\{L[i-1, j], L[i, j-1]\} & \text{if } X_i \neq Y_j \end{cases}$$

# Longest common subsequence

**Subproblem:**  $L[i, j]$ : the length of an LCS of  $X[1 \cdots i]$  and  $Y[1 \cdots j]$

**Goal:**  $L[m, n]$

**Make choice:** Is  $X_i = Y_j$ ?

**Recurrence:** (Proof!)

$$L[i, j] = \begin{cases} L[i-1, j-1] + 1 & \text{if } X_i = Y_j \\ \max\{L[i-1, j], L[i, j-1]\} & \text{if } X_i \neq Y_j \end{cases}$$

**Init:**

$$L[0, j] = 0, \quad 0 \leq j \leq n$$

$$L[i, 0] = 0, \quad 0 \leq i \leq m$$

**Time:**  $\Theta(mn)$

# Longest common subsequence

## Longest common subsequence (Problem 7.5)

$$X = X_1 \cdots X_m \quad Y = Y_1 \cdots Y_n$$

- (2) Allowing repetition of  $X$
- (3) Allowing repetition  $\leq k$  of  $X$

# Longest common subsequence

## Longest common subsequence (Problem 7.5)

$$X = X_1 \cdots X_m \quad Y = Y_1 \cdots Y_n$$

- (2) Allowing repetition of  $X$
- (3) Allowing repetition  $\leq k$  of  $X$

$$L[i, j] = \begin{cases} L[\textcolor{red}{i}, j - 1] + 1 & \text{if } X_i = Y_j \\ \max\{L[i - 1, j], L[i, j - 1]\} & \text{if } X_i \neq Y_j \end{cases}$$

# Longest common subsequence

## Longest common subsequence (Problem 7.5)

$$X = X_1 \cdots X_m \quad Y = Y_1 \cdots Y_n$$

- (2) Allowing repetition of  $X$
- (3) Allowing repetition  $\leq k$  of  $X$

$$L[i, j] = \begin{cases} L[\textcolor{red}{i}, j - 1] + 1 & \text{if } X_i = Y_j \\ \max\{L[i - 1, j], L[i, j - 1]\} & \text{if } X_i \neq Y_j \end{cases}$$

$$X \implies X^{(k)} \triangleq X_1^{(k)} \cdots X_m^{(k)}$$



# Longest common substring

What about longest common *substring*?

# Shortest common supersequence

## Shortest common supersequence (Problem 7.6)

$$X = X_1 \cdots X_m \quad Y = Y_1 \cdots Y_n$$

- Find (the length of) a shortest common subsequence of  $X$  and  $Y$

# Shortest common supersequence

**Subproblem:**  $L[i, j]$ : the length of an SCS of  $X[1 \dots i]$  and  $Y[1 \dots j]$

**Goal:**  $L[m, n]$

# Shortest common supersequence

**Subproblem:**  $L[i, j]$ : the length of an SCS of  $X[1 \dots i]$  and  $Y[1 \dots j]$

**Goal:**  $L[m, n]$

**Make choice:** Is  $X_i = Y_j$ ?

**Recurrence:**

$$L[i, j] = \begin{cases} L[i-1, j-1] + 1 & \text{if } X_i = Y_j \\ \max\{L[i-1, j] + 1, L[i, j-1] + 1\} & \text{if } X_i \neq Y_j \end{cases}$$

# Shortest common supersequence

**Subproblem:**  $L[i, j]$ : the length of an SCS of  $X[1 \dots i]$  and  $Y[1 \dots j]$

**Goal:**  $L[m, n]$

**Make choice:** Is  $X_i = Y_j$ ?

**Recurrence:**

$$L[i, j] = \begin{cases} L[i-1, j-1] + 1 & \text{if } X_i = Y_j \\ \max\{L[i-1, j] + 1, L[i, j-1] + 1\} & \text{if } X_i \neq Y_j \end{cases}$$

**Init:**

$$L[0, j] = j, \quad 0 \leq j \leq n$$

$$L[i, 0] = i, \quad 0 \leq i \leq m$$

# Shortest common supersequence

**Subproblem:**  $L[i, j]$ : the length of an SCS of  $X[1 \dots i]$  and  $Y[1 \dots j]$

**Goal:**  $L[m, n]$

**Make choice:** Is  $X_i = Y_j$ ?

**Recurrence:**

$$L[i, j] = \begin{cases} L[i-1, j-1] + 1 & \text{if } X_i = Y_j \\ \max\{L[i-1, j] + 1, L[i, j-1] + 1\} & \text{if } X_i \neq Y_j \end{cases}$$

**Init:**

$$L[0, j] = j, \quad 0 \leq j \leq n$$

$$L[i, 0] = i, \quad 0 \leq i \leq m$$

**Remark**

$$\max(m, n) \leq L(m, n) \leq m + n$$

# Variants of LCS

## Variants of LCS (Problem 7.7)

# Longest contiguous substring both forward and backward

## Longest contiguous substring both forward and backward (Problem 7.8)

- ▶ String  $T[1 \cdots n]$
- ▶ Find a longest contiguous substring (LCS) both forward and backward

dynamicprogrammingmanytimes

- ▶ try subproblem  $L[i]$ : the length of an LCS in  $T[1 \cdots i]$
- ▶ try subproblem  $L[i, j]$ : the length of an LCS in  $T[i \cdots j]$



# Longest contiguous substring both forward and backward

**Subproblem:**  $L[i, j]$ : the length of an LCS starting with  $T_i$  and ending with  $T_j$

**Goal:**  $\max_{1 \leq i \leq j \leq n} L[i, j]$

# Longest contiguous substring both forward and backward

**Subproblem:**  $L[i, j]$ : the length of an LCS **starting with  $T_i$  and ending with  $T_j$**

**Goal:**  $\max_{1 \leq i \leq j \leq n} L[i, j]$

**Make choice:** Is  $T_i = T_j$ ?

**Recurrence:**

$$L[i, j] = \begin{cases} 0 & \text{if } T_i \neq T_j \\ L[i + 1, j - 1] + 1 & \text{if } T_i = T_j \end{cases}$$

# Longest contiguous substring both forward and backward

**Subproblem:**  $L[i, j]$ : the length of an LCS **starting with  $T_i$  and ending with  $T_j$**

**Goal:**  $\max_{1 \leq i \leq j \leq n} L[i, j]$

**Make choice:** Is  $T_i = T_j$ ?

**Recurrence:**

$$L[i, j] = \begin{cases} 0 & \text{if } T_i \neq T_j \\ L[i + 1, j - 1] + 1 & \text{if } T_i = T_j \end{cases}$$

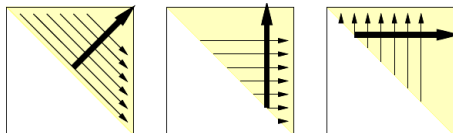
**Init:**

$$L[i, i] = 0, 0 \leq i \leq n$$

$$L[i, i + 1] = \begin{cases} 1 & \text{if } T_i = T_{i+1} \\ 0 & \text{if } T_i \neq T_{i+1} \end{cases}$$

# Longest contiguous substring both forward and backward

Code: three ways of filling the table



```

for d = 2 to n-1
  for i = 1 to n-d
    j = i + d
    ...
return max_{1 <= i <= j <= n} L[i,j]

```

# Longest palindrome subsequence

## Longest palindrome subsequence (Problem 7.10)

(1) Find (the length of) a longest palindrome subsequence of  $S[1 \cdots n]$

Subproblem:  $L[i, j]$ : the length of an LSP of  $S[i \cdots j]$

Goal:  $L[1, n]$

# Longest palindrome subsequence

## Longest palindrome subsequence (Problem 7.10)

(1) Find (the length of) a longest palindrome subsequence of  $S[1 \cdots n]$

Subproblem:  $L[i, j]$ : the length of an LSP of  $S[i \cdots j]$

Goal:  $L[1, n]$

Make choice: Is  $S[i] = S[j]$ ?

Recurrence:

$$L[i, j] = \begin{cases} L[i + 1, j - 1] + 2 & \text{if } S[i] = S[j] \\ \max L[i + 1, j], L[i, j - 1] & \text{if } S[i] \neq S[j] \end{cases}$$

# Longest palindrome subsequence

## Longest palindrome subsequence (Problem 7.10)

(1) Find (the length of) a longest palindrome subsequence of  $S[1 \cdots n]$

Subproblem:  $L[i, j]$ : the length of an LSP of  $S[i \cdots j]$

Goal:  $L[1, n]$

Make choice: Is  $S[i] = S[j]$ ?

Recurrence:

$$L[i, j] = \begin{cases} L[i + 1, j - 1] + 2 & \text{if } S[i] = S[j] \\ \max L[i + 1, j], L[i, j - 1] & \text{if } S[i] \neq S[j] \end{cases}$$

Init:

$$L[i, i] = 1, \forall 1 \leq i \leq n$$

# Palindrome splitting

## Palindrome splitting (Problem 7.10)

(2) Split a string  $S[1 \dots n]$  into minimum number of palindromes (# cuts)

**Subproblem:**  $C[i, j]$ : minimum number of cuts for string  $S[i \dots j]$

**Goal:**  $C[1, n] + 1$



# Palindrome splitting

## Palindrome splitting (Problem 7.10)

(2) Split a string  $S[1 \dots n]$  into minimum number of palindromes (# cuts)

Subproblem:  $C[i, j]$ : minimum number of cuts for string  $S[i \dots j]$

Goal:  $C[1, n] + 1$

Make choice: Where is the first cut?

Recurrence:

$$C[i, j] = \begin{cases} 0 & \text{if } S[i \dots j] \text{ is a palindrome} \\ \min_{i+1 \leq k \leq j-1} C[i, k] + 1 + C[k+1, j] & \text{o.w.} \end{cases}$$

# Palindrome splitting

## Palindrome splitting (Problem 7.10)

(2) Split a string  $S[1 \dots n]$  into minimum number of palindromes (# cuts)

**Subproblem:**  $C[i, j]$ : minimum number of cuts for string  $S[i \dots j]$

**Goal:**  $C[1, n] + 1$

**Make choice:** Where is the first cut?

**Recurrence:**

$$C[i, j] = \begin{cases} 0 & \text{if } S[i \dots j] \text{ is a palindrome} \\ \min_{i+1 \leq k \leq j-1} C[i, k] + 1 + C[k+1, j] & \text{o.w.} \end{cases}$$

**Init:**  $C[i, i] = 0$

**Time:**  $O(n^3)$

# Palindrome splitting

## Palindrome splitting (Problem 7.10)

(2) Split a string  $S[1 \dots n]$  into minimum number of palindromes

Subproblem:  $P[i]$ : minimum number of palindromes for  $S[1 \dots i]$

Goal:  $P[n]$

# Palindrome splitting

## Palindrome splitting (Problem 7.10)

(2) Split a string  $S[1 \dots n]$  into minimum number of palindromes

Subproblem:  $P[i]$ : minimum number of palindromes for  $S[1 \dots i]$

Goal:  $P[n]$

Make choice: Where does the last palindrome start from?

Recurrence:

$$P[i] = \min_{\substack{1 \leq k \leq i \\ S[k \dots i] \text{ is a palindrome}}} P[k-1] + 1$$

# Palindrome splitting

## Palindrome splitting (Problem 7.10)

(2) Split a string  $S[1 \dots n]$  into minimum number of palindromes

Subproblem:  $P[i]$ : minimum number of palindromes for  $S[1 \dots i]$

Goal:  $P[n]$

Make choice: Where does the last palindrome start from?

Recurrence:

$$P[i] = \min_{\substack{1 \leq k \leq i \\ S[k \dots i] \text{ is a palindrome}}} P[k-1] + 1$$

Init:  $P[0] = 1$

Time:  $O(n^2)$

# String splitting

## String splitting (Problem 7.11)

- ▶ Split a string  $S$  into many pieces
- ▶ Cost  $|S| = n \implies n$
- ▶ Given locations of  $m$  cuts:  $C_0, C_1, \dots, C_m, C_{m+1}$
- ▶ Find the minimum cost of splitting  $S$  into  $m + 1$  pieces  $S_0 \cdots S_m$

# String splitting

**Subproblem:**  $C[i, j]$ : the minimum cost of splitting substring  $S_i \cdots S_{j-1}$  using cuts  $C_{i+1} \cdots C_{j-1}$

**Goal:**  $C[0, m + 1]$

# String splitting

**Subproblem:**  $C[i, j]$ : the minimum cost of splitting substring  $S_i \cdots S_{j-1}$  using cuts  $C_{i+1} \cdots C_{j-1}$

**Goal:**  $C[0, m + 1]$

**Make choice:** What is the first cut in  $C_{i+1} \cdots C_{j-1}$ ?

**Recurrence:**

$$C[i, j] = \min_{i < k < j} (C[i, k] + C[k, j] + l(S_i \cdots S_{j-1}))$$



# String splitting

**Subproblem:**  $C[i, j]$ : the minimum cost of splitting substring  $S_i \cdots S_{j-1}$  using cuts  $C_{i+1} \cdots C_{j-1}$

**Goal:**  $C[0, m + 1]$

**Make choice:** What is the first cut in  $C_{i+1} \cdots C_{j-1}$ ?

**Recurrence:**

$$C[i, j] = \min_{i < k < j} (C[i, k] + C[k, j] + l(S_i \cdots S_{j-1}))$$

**Init:**  $C[i, i + 1] = 0$

Subproblem:

Subproblem:

Goal:

Make choice:

Subproblem:

Goal:

Make choice:

Recurrence:

Init:

Time:

# Dynamic Programming

- 1 Overview
- 2 1D DP
- 3 2D DP
- 4 3D DP**
- 5 DP on Graphs
- 6 The Knapsack Problem
- 7 Summary

# 3-D DP

## Floyd-Warshall algorithm

(1) DP for Floyd-Warshall algorithm for APSP on **directed** graphs

**Subproblem:**  $\text{dist}[i, j, k]$ : the length of the shortest path from  $i$  to  $j$  via only nodes in  $v_1 \cdots v_k$

**Goal:**  $\text{dist}[i, j, n], \forall i, j$

# 3-D DP

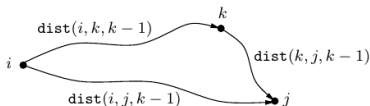
## Floyd-Warshall algorithm

(1) DP for Floyd-Warshall algorithm for APSP on **directed** graphs

Make choice: Is  $v_k$  on the ShortestPath $[i, j, k]$ ?

Recurrence:

$$\text{dist}[i, j, k] = \min\{\text{dist}[i, j, k-1], \text{dist}[i, k, k-1] + \text{dist}[k, j, k-1]\}$$



# 3-D DP

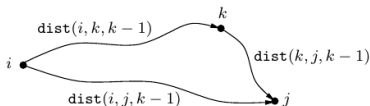
## Floyd-Warshall algorithm

(1) DP for Floyd-Warshall algorithm for APSP on **directed** graphs

Make choice: Is  $v_k$  on the ShortestPath $[i, j, k]$ ?

Recurrence:

$$\text{dist}[i, j, k] = \min\{\text{dist}[i, j, k-1], \text{dist}[i, k, k-1] + \text{dist}[k, j, k-1]\}$$



Init:

$$\text{dist}[i, j, 0] = \begin{cases} 0 & i = j \\ w(i, j) & (i, j) \in E \\ \infty & \text{o.w.} \end{cases}$$



## 3-D DP

Floyd-Warshall algorithm (Problem 6.25)

(2) Routing table for Floyd-Warshall algorithm

```
for all  $k \leftarrow 1 \dots n$  do  
  for all  $i \leftarrow 1 \dots n$  do  
    for all  $j \leftarrow 1 \dots n$  do  
      if  $\text{dist}[i, j] > \text{dist}[i, k] + \text{dist}[k, j]$  then  
         $\text{dist}[i, j] \leftarrow \text{dist}[i, k] + \text{dist}[k, j]$ 
```

# 3-D DP

Floyd-Warshall algorithm (Problem 6.25)

(2) Routing table for Floyd-Warshall algorithm

```

for all  $k \leftarrow 1 \dots n$  do
  for all  $i \leftarrow 1 \dots n$  do
    for all  $j \leftarrow 1 \dots n$  do
      if  $\text{dist}[i, j] > \text{dist}[i, k] + \text{dist}[k, j]$  then
         $\text{dist}[i, j] \leftarrow \text{dist}[i, k] + \text{dist}[k, j]$ 
         $\text{Go}[i, j] \leftarrow \text{Go}[i, k]$ 
  
```

# 3-D DP

## Floyd-Warshall algorithm (Problem 6.25)

### (2) Routing table

```

for all  $i \leftarrow 1 \dots n$  do
    for all  $j \leftarrow 1 \dots n$  do
         $\text{dist}[i, j] \leftarrow \infty$ 
         $\text{Go}[i, j] \leftarrow \text{Nil}$ 
for all  $(i, j) \in E$  do
     $\text{dist}[i, j] \leftarrow w(i, j)$ 
     $\text{Go}[i, j] \leftarrow j$ 
for all  $i \leftarrow 1 \dots n$  do
     $\text{dist}[i, i] \leftarrow 0$ 
     $\text{Go}[i, i] \leftarrow \text{Nil}$ 
  
```

# 3-D DP

## Floyd-Warshall algorithm (Problem 6.25)

### (2) Routing table

```

for all  $i \leftarrow 1 \dots n$  do
    for all  $j \leftarrow 1 \dots n$  do
         $\text{dist}[i, j] \leftarrow \infty$ 
         $\text{Go}[i, j] \leftarrow \text{Nil}$ 
for all  $(i, j) \in E$  do
     $\text{dist}[i, j] \leftarrow w(i, j)$ 
     $\text{Go}[i, j] \leftarrow j$ 
for all  $i \leftarrow 1 \dots n$  do
     $\text{dist}[i, i] \leftarrow 0$ 
     $\text{Go}[i, i] \leftarrow \text{Nil}$ 
  
```

```

procedure  $\text{PATH}(i, j)$ 
    if  $\text{Go}[i, j] = \text{Nil}$  then
        Output "No Path."
  
```

```

    Output " $i$ "
    while  $i \neq j$  do
         $i \leftarrow \text{Go}[i, j]$ 
    Output " $i$ "
  
```

## 3-D DP

Floyd-Warshall algorithm (Problem 6.29)

(3) Find minimum-weighted cycle of directed graph ( $w(e) > 0$ )

$$\text{dist}[i, i] \leftarrow 0 \implies \text{dist}[i, i] \leftarrow \infty$$

## 3-D DP

Floyd-Warshall algorithm (Problem 6.29)

(3) Find minimum-weighted cycle of directed graph ( $w(e) > 0$ )

$$\text{dist}[i, i] \leftarrow 0 \implies \text{dist}[i, i] \leftarrow \infty$$

$$\exists i : \text{dist}[i, i] < 0$$

$$\forall i : \text{dist}[i, i] \geq 0$$

# Shortest paths on undirected graphs

## Finding shortest paths in undirected graphs with possibly negative edge weights



2



2

The book "[Algorithms](#)" by Robert Sedgewick and Kevin Wayne hinted that (*see the quote below*) there are efficient algorithms for finding shortest paths in undirected graphs with possibly negative edge weights (**not** by treating an undirected edge as two directed one which means that a single negative edge implies a negative cycle). However, no references are given in the book. Are you aware of any such algorithms?

Q. How can we find shortest paths in undirected (edge-weighted) graphs?

A. For positive edge weights, Dijkstra's algorithm does the job. We just build an `EdgeWeightedDigraph` corresponding to the given `EdgeWeightedGraph` (by adding two directed edges corresponding to each undirected edge, one in each direction) and then run Dijkstra's algorithm. ***If edge weights can be negative (emphasis added)***, efficient algorithms are available, but they are more complicated than the Bellman-Ford algorithm.

algorithms

graph-theory

shortest-path

weighted-graphs

reference-question

share cite edit close delete flag

edited Jun 9 at 14:15

asked Jun 9 at 13:58



hengxin

6,056 11 38

<https://cs.stackexchange.com/q/76578/4911>

# Dynamic Programming

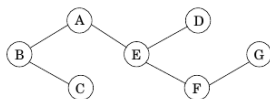
- 1 Overview
- 2 1D DP
- 3 2D DP
- 4 3D DP
- 5 DP on Graphs**
- 6 The Knapsack Problem
- 7 Summary



# Minimum vertex cover on trees

## Minimum vertex cover on trees [Problem: 2.2.18]

- ▶ Undirected tree  $T = (V, E)$ ; **No designated root!**
- ▶ Compute (the size of) a minimum vertex cover of  $T$



# Minimum vertex cover on trees

Rooted  $T$  at any node  $r$ .

# Minimum vertex cover on trees

Rooted  $T$  at any node  $r$ .

**Subproblem:**  $I(u)$ : the size of an MVC of subtree  $T_u$  rooted at  $u$

**Goal:**  $I(r)$

# Minimum vertex cover on trees

Rooted  $T$  at any node  $r$ .

Subproblem:  $I(u)$ : the size of an MVC of subtree  $T_u$  rooted at  $u$

Goal:  $I(r)$

Make choice: Is  $u$  in MVC $[u]$ ?

Recurrence:

$$I(u) = \min\{\# \text{ children of } u + \sum_{v: \text{ grandchildren of } u} I(v), \\ 1 + \sum_{v: \text{ children of } u} I(v)\}$$

# Minimum vertex cover on trees

Rooted  $T$  at any node  $r$ .

Subproblem:  $I(u)$ : the size of an MVC of subtree  $T_u$  rooted at  $u$

Goal:  $I(r)$

Make choice: Is  $u$  in MVC $[u]$ ?

Recurrence:

$$I(u) = \min\{\# \text{ children of } u + \sum_{v: \text{ grandchildren of } u} I(v), \\ 1 + \sum_{v: \text{ children of } u} I(v)\}$$

Init:  $I(u) = 0$ , if  $u$  is a leaf

# Minimum vertex cover on trees

DFS on  $T$  from root  $r$ :

when  $u$  is “finished”:

**if**  $u$  is a leaf **then**

$$I(u) \leftarrow 0$$

**else**

$$I(u) \leftarrow \dots$$

# Minimum vertex cover on trees

DFS on  $T$  from root  $r$ :

when  $u$  is “finished”:

**if**  $u$  is a leaf **then**

$$I(u) \leftarrow 0$$

**else**

$$I(u) \leftarrow \dots$$

Greedy algorithm:

Theorem

*There is an MVC which contains no leaves.*

# DP on DAG

## Longest path in DAG (Problem 7.17)

- ▶ Direction:  $\downarrow$  OR  $\rightarrow$
- ▶ Score:  $\geq < 0$



# DP on DAG

## Longest path in DAG (Problem 7.17)

- ▶ Direction:  $\downarrow$  OR  $\rightarrow$
- ▶ Score:  $\geq < 0$

1. digraph  $G$
2. node weight  $\rightarrow$  edge weight
3. adding an extra sink  $s$
4.  $G \rightarrow G^T$

# DP on DAG

## Longest path in DAG (Problem 7.17)

- ▶ Direction:  $\downarrow$  OR  $\rightarrow$
- ▶ Score:  $\geq < 0$

1. digraph  $G$
2. node weight  $\rightarrow$  edge weight
3. adding an extra sink  $s$
4.  $G \rightarrow G^T$

Compute a longest path from  $s$  in DAG

# DP on DAG

**Subproblem:**  $\text{dist}[v]$ : longest distance from  $s$  to  $v$

**Goal:**  $\text{dist}[v], \forall v \in V$

# DP on DAG

Subproblem:  $\text{dist}[v]$ : longest distance from  $s$  to  $v$

Goal:  $\text{dist}[v], \forall v \in V$

Make choice:

Recurrence:

$$\text{dist}[v] = \max_{u \rightarrow v} (\text{dist}[u] + w(u \rightarrow v))$$

# DP on DAG

Subproblem:  $\text{dist}[v]$ : longest distance from  $s$  to  $v$

Goal:  $\text{dist}[v], \forall v \in V$

Make choice:

Recurrence:

$$\text{dist}[v] = \max_{u \rightarrow v} (\text{dist}[u] + w(u \rightarrow v))$$

Init:  $\text{dist}[s] = 0$

# DP on DAG

Subproblem:  $\text{dist}[v]$ : longest distance from  $s$  to  $v$

Goal:  $\text{dist}[v], \forall v \in V$

Make choice:

Recurrence:

$$\text{dist}[v] = \max_{u \rightarrow v} (\text{dist}[u] + w(u \rightarrow v))$$

Init:  $\text{dist}[s] = 0$

Compute  $\text{dist}[v]$  in topo. order

# Bitonic tour

Bitonic tour (Problem 7.18)

# Bitonic tour



# Dynamic Programming

- 1 Overview
- 2 1D DP
- 3 2D DP
- 4 3D DP
- 5 DP on Graphs
- 6 The Knapsack Problem**
- 7 Summary

# The change-making problem

## The change-making problem (Problem 7.12)

- ▶ Coins values:  $x_1 \dots x_n$
- ▶ Amount:  $v$
- ▶ Is it possible to make change for  $v$ ?

# The change-making problem

The change-making problem (Problem 7.12(2), Problem 7.1 (Subset sum))  
(2) Without repetition (0/1)

# The change-making problem

The change-making problem (Problem 7.12(2), Problem 7.1 (Subset sum))  
(2) Without repetition (0/1)

**Subproblem:**  $C[i, w]$ : Possible to make change for  $w$  using only  $x_1 \dots x_n$ ?

**Goal:**  $C[n, v]$

# The change-making problem

The change-making problem (Problem 7.12(2), Problem 7.1 (Subset sum))  
 (2) Without repetition (0/1)

Subproblem:  $C[i, w]$ : Possible to make change for  $w$  using only  $x_1 \dots x_n$ ?

Goal:  $C[n, v]$

Make choice: Using  $x_i$  or not?

Recurrence:

$$C[i, w] = C[i - 1, w] \vee (C[i - 1, w - x_i] \wedge w \geq x_i)$$

# The change-making problem

The change-making problem (Problem 7.12(2), Problem 7.1 (Subset sum))  
 (2) Without repetition (0/1)

Subproblem:  $C[i, w]$ : Possible to make change for  $w$  using only  $x_1 \dots x_n$ ?

Goal:  $C[n, v]$

Make choice: Using  $x_i$  or not?

Recurrence:

$$C[i, w] = C[i - 1, w] \vee (C[i - 1, w - x_i] \wedge w \geq x_i)$$

Init:

$$C[i, 0] = \text{true}$$

$$C[0, w] = \text{false, if } w > 0$$

$$C[0, 0] = \text{true}$$

Time:  $O(nv)$

# The change-making problem

The change-making problem (Problem 7.12(1))

(1) Unbounded repetition ( $\infty$ )

# The change-making problem

The change-making problem (Problem 7.12(1))

(1) Unbounded repetition ( $\infty$ )

**Subproblem:**  $C[i, w]$ : Possible to make change for  $w$  using only  $x_1 \dots x_n$ ?

**Goal:**  $C[n, v]$



# The change-making problem

## The change-making problem (Problem 7.12(1))

### (1) Unbounded repetition ( $\infty$ )

**Subproblem:**  $C[i, w]$ : Possible to make change for  $w$  using only  $x_1 \dots x_n$ ?

**Goal:**  $C[n, v]$

**Make choice:** Using  $x_i$  or not?

**Recurrence:**

$$C[i, w] = C[i - 1, w] \vee (C[\textcolor{red}{i}, w - x_i] \wedge w \geq x_i)$$

# The change-making problem

## The change-making problem (Problem 7.12(1))

### (1) Unbounded repetition ( $\infty$ )

**Subproblem:**  $C[i, w]$ : Possible to make change for  $w$  using only  $x_1 \dots x_n$ ?

**Goal:**  $C[n, v]$

**Make choice:** Using  $x_i$  or not?

**Recurrence:**

$$C[i, w] = C[i - 1, w] \vee (C[i, w - x_i] \wedge w \geq x_i)$$

**Init:**

$$C[i, 0] = \text{true}$$

$$C[0, w] = \text{false, if } w > 0$$

$$C[0, 0] = \text{true}$$

**Time:**  $O(nv)$

# The change-making problem

The change-making problem (Problem 7.12(1))

(1) Unbounded repetition ( $\infty$ )

# The change-making problem

The change-making problem (Problem 7.12(1))

(1) Unbounded repetition ( $\infty$ )

Subproblem:  $C[w]$ : Possible to make change for  $w$ ?

Goal:  $C[v]$

# The change-making problem

The change-making problem (Problem 7.12(1))

(1) Unbounded repetition ( $\infty$ )

Subproblem:  $C[w]$ : Possible to make change for  $w$ ?

Goal:  $C[v]$

Make choice: Suppose  $x_i$  is used.

Recurrence:

$$C[w] = \bigvee_{i: x_i \leq w} C[w - x_i]$$

# The change-making problem

The change-making problem (Problem 7.12(1))

(1) Unbounded repetition ( $\infty$ )

Subproblem:  $C[w]$ : Possible to make change for  $w$ ?

Goal:  $C[v]$

Make choice: Suppose  $x_i$  is used.

Recurrence:

$$C[w] = \bigvee_{i: x_i \leq w} C[w - x_i]$$

Time:  $O(nv)$

# The change-making problem

The change-making problem (Problem 7.12(1))

(1) Unbounded repetition ( $\infty$ )

Subproblem:  $C[w]$ : Possible to make change for  $w$ ?

Goal:  $C[v]$

Make choice: Suppose  $x_i$  is used.

Recurrence:

$$C[w] = \bigvee_{i: x_i \leq w} C[w - x_i]$$

Time:  $O(nv)$

Q:  $C[i, w]$  vs.  $C[w]$

# The change-making problem

The change-making problem (Problem 7.12(3))

(3) Unbounded repetition with  $\leq k$  coins



# The change-making problem

The change-making problem (Problem 7.12(3))

(3) Unbounded repetition with  $\leq k$  coins

**Subproblem:**  $C[i, w, l]$ : Possible to make change for  $w$  with  $\leq l$  coins of  $x_1 \dots x_i$ ?

**Goal:**  $C[n, v, k]$

# The change-making problem

## The change-making problem (Problem 7.12(3))

(3) Unbounded repetition with  $\leq k$  coins

**Subproblem:**  $C[i, w, l]$ : Possible to make change for  $w$  with  $\leq l$  coins of  $x_1 \dots x_i$ ?

**Goal:**  $C[n, v, k]$

**Make choice:** Using  $x_i$  or not?

**Recurrence:**

$$C[i, w, l] = C[i - 1, w, l] \vee (C[\textcolor{red}{i}, w - x_i, \textcolor{red}{l} - 1] \wedge w \geq x_i)$$

# The change-making problem

## The change-making problem (Problem 7.12(3))

(3) Unbounded repetition with  $\leq k$  coins

**Subproblem:**  $C[i, w, l]$ : Possible to make change for  $w$  with  $\leq l$  coins of  $x_1 \dots x_i$ ?

**Goal:**  $C[n, v, k]$

**Make choice:** Using  $x_i$  or not?

**Recurrence:**

$$C[i, w, l] = C[i - 1, w, l] \vee (C[\textcolor{red}{i}, w - x_i, \textcolor{red}{l} - 1] \wedge w \geq x_i)$$

**Init:**

$$C[0, 0, l] = \text{true}, \quad C[0, w, l] = \text{false}, \text{ if } w > 0$$

$$C[i, 0, l] = \text{true}, \quad C[i, w, 0] = \text{false}, \text{ if } w > 0$$

# Dynamic Programming

- 1 Overview
- 2 1D DP
- 3 2D DP
- 4 3D DP
- 5 DP on Graphs
- 6 The Knapsack Problem
- 7 Summary**

# More DPs . . .

## Algorithms that use dynamic programming [\[ edit \]](#) [edit source](#) ]



This section **does not cite any sources**. Please help [improve this section](#) by adding citations to reliable sources. Unsourced material may be challenged [how and when to remove this template message](#)

- Recurrent solutions to [lattice models](#) for protein-DNA binding
- [Backward induction](#) as a solution method for finite-horizon [discrete-time](#) dynamic optimization problems
- Method of undetermined coefficients can be used to solve the [Bellman equation](#) in infinite-horizon, discrete-time, [discounted](#), time-invariant dynamic optimization problems
- Many string algorithms including [longest common subsequence](#), [longest increasing subsequence](#), [longest common substring](#), [Levenshtein distance](#) (edit distance)
- Many algorithmic problems on [graphs](#) can be solved efficiently for graphs of bounded [treewidth](#) or bounded [clique-width](#) by using dynamic programming on a [tree decomposition](#) of the graph.
- The Cocke-Younger-Kasami (CYK) algorithm which determines whether and how a given string can be generated by a given context-free grammar
- Knuth's word wrapping algorithm that minimizes raggedness when word wrapping text
- The use of [transposition tables](#) and [refutation tables](#) in computer chess
- The Viterbi algorithm (used for hidden Markov models)
- The Earley algorithm (a type of [chart parser](#))
- The Needleman-Wunsch algorithm and other algorithms used in [bioinformatics](#), including [sequence alignment](#), [structural alignment](#), [RNA structure prediction](#)
- Floyd's all-pairs shortest path algorithm
- Optimizing the order for chain matrix multiplication
- Pseudo-polynomial time algorithms for the [subset sum](#), [knapsack](#) and [partition](#) problems
- The dynamic time warping algorithm for computing the global distance between two time series
- The Selinger (a.k.a. [System R](#)) algorithm for relational database query optimization
- De Boor algorithm for evaluating B-spline curves
- Duckworth-Lewis method for resolving the problem when games of cricket are interrupted
- The value iteration method for solving [Markov decision processes](#)
- Some graphic image edge following selection methods such as the "magnet" selection tool in [Photoshop](#)
- Some methods for solving [interval scheduling](#) problems
- Some methods for solving the [travelling salesman problem](#), either exactly (in [exponential time](#)) or approximately (e.g. via the [bitonic tour](#))
- Recursive least squares method
- Beat tracking in [music information retrieval](#)
- Adaptive-critic training strategy for [artificial neural networks](#)
- Stereo algorithms for solving the [correspondence problem](#) used in stereo vision
- [Seam carving](#) (content-aware image resizing)
- The [Bellman-Ford algorithm](#) for finding the shortest distance in a graph
- Some approximate solution methods for the [linear search problem](#)
- Kadane's algorithm for the [maximum subarray problem](#)

