



2025 Fall CSE5025

Combinatorial Optimization

组合优化

Instructor: 刘晟材

Lecture 4: Dynamic Programming for Combinatorial Optimization

Agenda for Today's Lecture



In this lecture, we will focus on

- General idea of Dynamic Programming (DP)
- Different Types of DP
- Pseudo-polynomial time DP and Exponential-time DP

Learning Objectives for this lecture

- Master the basic idea of DP
- Know different types of DP
- Master how to design DP

What is Dynamic Programming (DP)?

DP is an **algorithm design technique** for solving complex problems by breaking them down into **simpler, overlapping** subproblems.

DP is not a specific algorithm, but a **programming paradigm** (范式), like “Divide & Conquer”.

The name DP is coined by Richard Bellman in the 1950s:

- “Dynamic” was chosen to sound impressive; “Programming” meant “planning” (规划) or “tabulation” (like in “Linear Programming”), not coding.

DP is essentially “**careful brute-force**” or “**recursion with memorization**”.

The Two Hallmarks (标志) of DP

A problem can be solved using DP if it exhibits two key properties.

Optimal Substructure

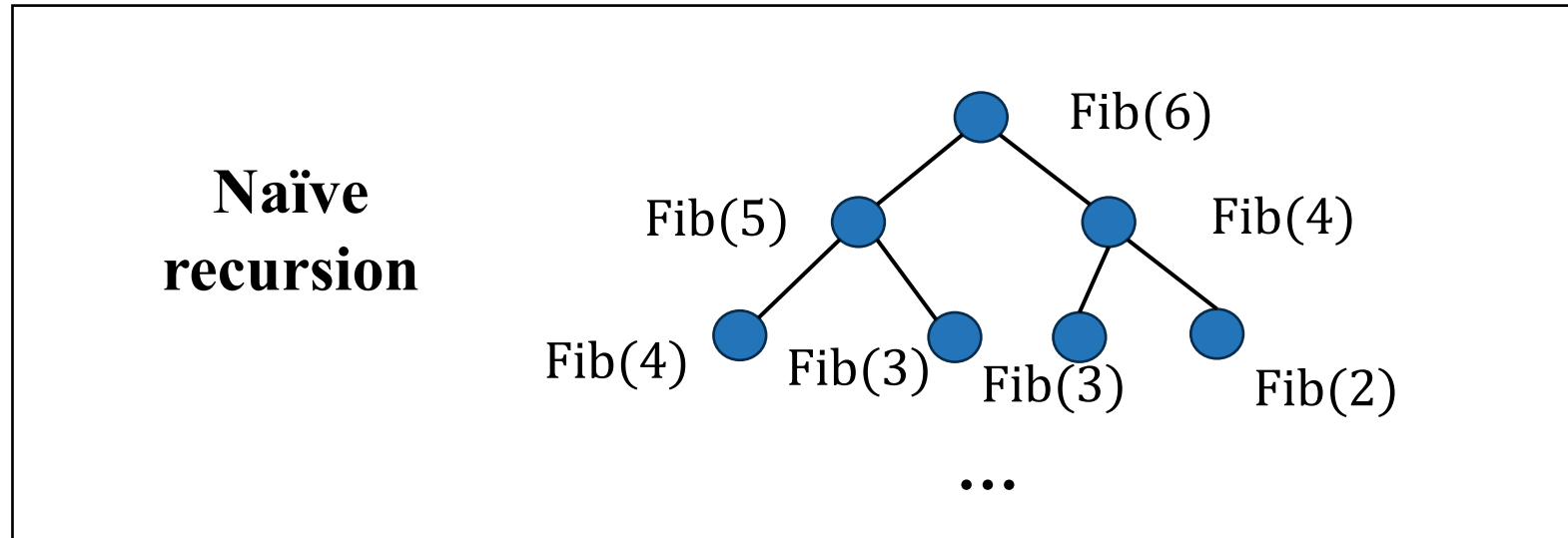
- The optimal solution to the overall problem can be constructed from the optimal solutions to its subproblems.
- Example: $Fib(n) = Fib(n - 1) + Fib(n - 2)$

Overlapping Subproblems

- A naive recursive solution would solve the same subproblems over and over again.
- DP solves each subproblem exactly once and stores its result in a table (memorization) for future use.

DP vs. Naïve Recursion

Consider computing the n -th Fibonacci number recursively:



Naïve Recursion: Exponential time $O(2^n)$.

DP (Memorization and Tabulation): Build from the bottom up: $\text{Fib}(0), \text{Fib}(1), \text{Fib}(2), \dots \text{Fib}(n)$. Each $\text{Fib}(i)$ is computed only once.

DP Runtime: Linear time $O(n)$.

DP vs. Divide & Conquer (DC)



Paradigm	Subproblems	Subproblem Overlap	Strategy
DC	Independent	No	Top-down. Solves independent parts, then combines. (e.g., Quick Sort and Merge Sort)
DP	Dependent	Yes	Bottom-up. Solves all necessary subproblems and stores the value to find the best.

Formulating a DP algorithm



To create a DP algorithm, we must define three things:

The **State**:

- What is the subproblem we are solving?
- This is the “lookup key” in our table. e.g., $DP(i)$, $DP(i, j)$, $DP(S, i)$.
- **This is the hardest part of DP!**

The **Recurrence**:

- How can we solve a larger problem (state) using the solutions to smaller subproblems (states)?
- This is the “optimal substructure” property in action.

The **Base cases**:

- What are the smallest subproblems we can solve directly? e.g., $DP(0)$, $DP(i, i)$.

DP for The 0/1 KP



Given

- A set of n items, $N = \{1, 2, \dots, n\}$.
- Each item i has a weight $w_i > 0$ and a value $v_i > 0$.
- A knapsack with a total weight capacity W

Goal

- Select a subset of items to maximize the total value $\sum v_i x_i$
- Subject to $\sum w_i x_i \leq W$ and $x_i \in \{0, 1\}$

DP Attempt: Greedy-Style

Let's try to build a DP based on items $1, \dots, n$.

Attempted State: $DP(i)$ = Optimal value using items only from $\{1, \dots, i\}$.

Recurrence? $DP(i) = ?$

- If the optimal solution does not take item i : $DP(i) = DP(i - 1)$.
- If the optimal solution takes item i : $DP(i) = v_i + ???$

Problem: We don't know what $???$ is. To know if we can add item i , we need to know the remaining capacity. $DP(i - 1)$ only tells us the value, not the weight of the optimal solution for using items from $\{1, \dots, i - 1\}$.

Conclusion: The state $DP(i)$ is **insufficient**. It fails to capture all the information needed to build the solution.

DP based on Weight (1)



Let's add the missing information (weight) to our state.

State: $DP(i, Q)$ = The optimal value to such a KP problem instance: using items only from $\{1, \dots, i\}$ with a total weight of at most Q .

Let $P(i, Q)$ be the problem instance corresponding to $DP(i, Q)$.

DP based on Weight (2)

Recurrence: When considering $DP(i, Q)$ and item i (with v_i, w_i)

- **Case 1: The optimal solution to $P(i, Q)$ does not take item i .** Then the optimal value is the same as the value of the optimal solution to $P(i - 1, Q)$, i.e., using items only from $\{1, \dots, i - 1\}$ with capacity Q .
- In case 1: $DP(i, Q) = DP(i - 1, Q)$
- **Proof:** In case 1,
 - 1) Because the optimal solution (not taking i) to $P(i, Q)$ is also a valid solution to $P(i - 1, Q)$, then $DP(i, Q) \leq DP(i - 1, Q)$.
 - 2) Because the optimal solution to $P(i - 1, Q)$ is also a valid solution to $P(i, Q)$ (by definition), then $DP(i, Q) \geq DP(i - 1, Q)$.
 - 3) We must have $DP(i, Q) = DP(i - 1, Q)$

DP based on Weight (3)

Recurrence: When considering $DP(i, Q)$ and item i (with v_i, w_i)

- **Case 2: The optimal solution to $P(i, Q)$ takes item i .** This is only possible when $w_i \leq Q$. Then the optimal value is v_i plus the value of the optimal solution to $P(i - 1, Q - w_i)$, i.e., using items only from $\{1, \dots, i - 1\}$ with capacity $Q - w_i$.
- In case 2: $DP(i, Q) = v_i + DP(i - 1, Q - w_i)$
- **Proof:** In case 2,
 - 1) Let S_1 be the optimal solution to $P(i - 1, Q - w_i)$. Then $S_1 \cup \{i\}$ is a valid solution to $P(i, Q)$. Let S_2 be the optimal solution to $P(i, Q)$ and thus S_2 contains item i . And $S_2 \setminus \{i\}$ is a valid solution to $P(i - 1, Q - w_i)$.
 - 3) Then we have $\text{Value}(S_2 \setminus \{i\}) \leq \text{Value}(S_1)$ and $\text{Value}(S_1 \cup \{i\}) \leq \text{Value}(S_2)$
 - 4) Then we have $\text{Value}(S_1 \cup \{i\}) = \text{Value}(S_1) + v_i \leq \text{Value}(S_2) \leq \text{Value}(S_1) + v_i$.
 - 5) Hence, $\text{Value}(S_2) = \text{Value}(S_1) + v_i$

DP based on Weight (4)



Recurrence: when considering $DP(i, Q)$ and item i (with v_i, w_i)

- $$DP(i, Q) = \max \begin{cases} DP(i-1, Q) \\ v_i + DP(i-1, Q - w_i), \text{ if } Q \geq w_i \end{cases}$$

Base cases: $DP(0, Q) = 0$ for all $Q \geq 0$.

DP based on Weight: Pseudocode



Algorithm Knapsack_DP_by_Weight(v, w, n, W):

// 1. Create a table of size $(n+1) \times (W+1)$

DP = create_array($n+1, W+1, 0$)

// 2. Fill the table

for i from 1 to n :

 for Q from 0 to W :

 // Option 1: Don't take item i

$val_1 = DP[i-1][Q]$

 // Option 2: Take item i

$val_2 = -\text{infinity}$

 if $w[i] \leq Q$:

$val_2 = v[i] + DP[i-1][Q - w[i]]$

$DP[i][Q] = \max(val_1, val_2)$

// 3. Final answer

return $DP[n][W]$

Pseudo-Polynomial
runtime: $O(nW)$

Finding the Solution: Traceback

Our DP table stores the value of the optimal solution to each subproblem.

They don't store the solution itself.

We find the solution by “walking backwards” through the DP table. This is called **traceback** or **reconstruction**.

The General Idea:

- Start at the final state.
- Look at the recurrence relation that built this state. Ask: “Which choice must we have made to get this optimal value?” (e.g., Did we take item n or not?)
- Move to the state that corresponds to that choice
- Repeat this process until we reach a base case.

Traceback for DP[n][W]



Algorithmic Steps:

Start with $i = n$ and $Q = W$. Initialize $\text{SolutionSet} = \{\}$.

While $i > 0$:

Let $\text{val_A} = \text{DP}[i-1][Q]$ (Value if we *did not* take item i).

If $\text{DP}[i][Q] == \text{val_A}$:

// This means we *could* have gotten the optimal value $\text{DP}[i][Q]$ *without* taking item i . So, we assume we *did not* take it. Move to the previous state:

$i = i - 1$;

Else:

// This means $\text{DP}[i][Q]$ *must* have come from $v_i + \text{DP}[i-1][Q - w_i]$. The only way to get this value was by *taking* item i .

// Add item i to SolutionSet . And Move to the state we came from:

$\text{SolutionSet} = \{i\} \cup \text{SolutionSet}$

$i = i - 1$

$Q = Q - w_i$.

Return SolutionSet

Runtime: $O(n)$

DP based on Value (1)



State: Let's swap the roles of value and weight. $DP(i, L)$ = The **minimum weight** required to achieve a total value of **exactly** L , using items only from $\{1, \dots, i\}$.

If L is impossible to achieve, then $DP(i, L) = \infty$

Let $P(i, L)$ be the problem instance corresponding to $DP(i, L)$.

Recurrence: When considering $DP(i, L)$ and item i (with v_i, w_i)

- **Case 1: The optimal solution to $P(i, L)$ does not take item i .** Then the optimal (minimum) weight is the same as the weight of the optimal solution to $P(i - 1, L)$, i.e., using items only from $\{1, \dots, i - 1\}$ to achieve L .
- In case 1: $DP(i, L) = DP(i - 1, L)$
- **Proof:** In case 1,
 - 1) Because the optimal solution (not taking i) to $P(i, L)$ is also a valid solution to $P(i - 1, L)$, then $DP(i, L) \geq DP(i - 1, L)$.
 - 2) Because the optimal solution to $P(i - 1, L)$ is also a valid solution to $P(i, L)$ (by definition), then $DP(i, L) \leq DP(i - 1, L)$.
 - 3) We must have $DP(i, L) = DP(i - 1, L)$

DP based on Value (3)

Recurrence: When considering $DP(i, L)$ and item i (with v_i, w_i)

- **Case 2: The optimal solution to $P(i, L)$ does take item i .** This is only possible when $v_i \leq L$. Then the optimal weight is w_i plus the weight of the optimal solution to $P(i - 1, L - v_i)$, i.e., using items only from $\{1, \dots, i - 1\}$ to achieve $L - v_i$.
- In case 2: $DP(i, L) = w_i + DP(i - 1, L - v_i)$
- **Proof:** In case 2,
 - 1) Let S_1 be the optimal solution to $P(i - 1, L - v_i)$. Then $S_1 \cup \{i\}$ is a valid solution to $P(i, L)$. Let S_2 be the optimal solution to $P(i, L)$ and thus S_2 contains item i . And $S_2 \setminus \{i\}$ is a valid solution to $P(i - 1, L - v_i)$.
 - 2) How to proceed?

DP based on Value (4)



Recurrence: when considering $DP(i, L)$ and item i (with v_i, w_i)

- $$DP(i, L) = \min \begin{cases} DP(i-1, L) \\ v_i + DP(i-1, L - v_i), \text{ if } L \geq v_i \end{cases}$$

Base cases: $DP(0, Q) = 0, DP(0, L) = \infty$ for $L > 0$.

DP based on Value: Pseudocode



Algorithm Knapsack_DP_by_Value(v, w, n, W):

// 1. Get max possible value L_max

L_max = sum of all v_i

// 2. Create table of size (n+1) x (L_max+1)

DP = create_array(n+1, L_max+1, infinity)

DP[0][0] = 0 // Base Case

// 3. Fill the table

for i from 1 to n:

 for L from 0 to L_max:

 weight_1 = DP[i-1][L] // Case 1: Don't take item i

 weight_2 = infinity // Case v2: Take item i

 if v[i] <= L:

 weight_2 = w[i] + DP[i-1][L - v[i]]

 DP[i][L] = min(weight_1, weight_2)

// 4. Find the final answer: the largest value s.t. its weight <= W

max_value = 0

for L from 0 to L_max:

 if DP[n][L] <= W:

 max_value = L

return max_value

Pseudopolynomial
runtime: $O(n^2 v_{\max})$

Traceback for $DP[n][max_value]$



Algorithmic Steps:

Start with $i = n$ and $L = max_value$. Initialize $SolutionSet = \{\}$.

While $i > 0$:

Let $weight_A = DP[i-1][L]$ (Weight if we *did not* take item i).

If $DP[i][L] == weight_A$:

// This means we do not take item i . Move to the previous state:

$i = i - 1$;

Else:

// This means $DP[i][L]$ *must* have come from $w_i + DP[i-1][L - v_i]$. The only way to get this value was by *taking* item i .

// Add item i to $SolutionSet$. And Move to the state we came from:

$SolutionSet = \{i\} \cup SolutionSet$

$i = i - 1$

$L = L - v_i$.

Return $SolutionSet$

Runtime: $O(n)$

DP for the Partitioning Problem

The Partitioning Problem is another classic weakly NP-hard problem.

Given

- A set of n items, $A = \{a_1, \dots, a_n\}$.
- Let $T = \sum_{a_i \in A} a_i$ be the total sum.

Goal

- Can we partition A into two disjoint subsets S_1 and S_2 such that the sum of elements in S_1 equals to the sum of elements in S_2 ?
- This is only possible if T is even.
- This is equivalent to: Does a subset $S \subseteq A$ exist such that $\sum_{a_i \in S} a_i = T/2$

Partition DP based on Target Sum (1)



State: $DP(i, Q)$ = Boolean (True/False).

$DP(i, Q)$ indicates whether we can achieve a sum of exactly Q using items only from $\{a_1, \dots, a_i\}$?

Partition DP based on Target Sum (2)



Recurrence: When considering $DP(i, Q)$ and item a_i :

- We can form a sum Q if:
 - Case 1: We could already form Q without a_i
 - $DP(i - 1, Q) = \text{True}$
 - Case 2: We can form $Q - a_i$ using the previous items. Only possible when $Q \geq a_i$
 - $DP(i - 1, Q - a_i) = \text{True}$
- $DP(i, L) = DP(i - 1, Q) \parallel DP(i - 1, Q - a_i)$ (if $Q \geq v_i$)

Base cases: $DP(i, 0) = \text{True}$ for $i > 0$. $DP(0, Q) = \text{False}$ for $Q > 0$

Partition DP based on Target Sum: Pseudocode

Algorithm Partition_DP(A, n):

$T = \text{sum}(A)$

 if T is odd return False

$W = T / 2$

 // 1. Initialize DP table: $DP[i][Q] = \text{True}$ if sum Q can be made from $\{a_1..a_i\}$

$DP = \text{create_array}(n+1, W+1, \text{False})$

 for i from 0 to n:

$DP[i][0] = \text{True}$ // Base case: 0 sum is always possible

 // 2. Fill the table (DP recurrence)

 for i from 1 to n:

 for Q from 1 to W:

 if $Q \geq A[i]$:

$DP[i][Q] = DP[i-1][Q - A[i]] \parallel DP[i-1][Q]$ //Take item A[i] OR Don't take item A[i]

 else:

$DP[i][Q] = DP[i-1][Q]$ // Can't take item A[i] (it's too big)

 // 3. Final answer

 return $DP[n][W]$

Partition DP: Analysis

The DP table has $(n + 1) \times (W + 1)$ entries, where $W = T/2$.

DP table Size = $O(nW) = O(nT)$.

Work per State:

- Each entry $DP(i, Q)$ is a Boolean OR of two previous entries.
- Work = $O(1)$.

Total Runtime: $O(nT)$

Conclusion:

- This is pseudo-polynomial (polynomial in n and the value T).
- The Partition problem is weakly NP-hard.

DP for Strongly NP-Hard Problems

For strongly NP-hard problems, we cannot (unless $P=NP$) find a pseudo-polynomial time algorithm.

The runtime of any exact algorithm will be exponential in n in the worst case (assuming input numbers are “small”).

Question: Can DP still help? Answer: Yes!

DP can provide an exact algorithm that is **much** better than naive brute-force.

- Brute-Force: $O(n!)$
- DP: $O(n^2 2^n)$
- This pattern is often called “DP on Subsets”.

DP for TSP (strongly NP-hard)

TSP: Given a complete weighted undirected graph with n vertices, and a cost function $c(u, v)$ that gives the the weight (cost) of each edge (u, v) , **find a Hamiltonian cycle (tour) of least weight (cost).**

An equivalent definition:

given:

- A set of n cities, $V = \{1, \dots, n\}$.
- A cost matrix C_{ij} for traveling from city i to j .

Goal:

- Find a tour (visiting every city exactly once and returns to the starting city) that minimizes the total cost.

The Held-Karp Algorithm



DP over subsets: this is the key insight for many graph problems.

State: We need to know two things:

- Which cities have we visited? (A subset $S \subseteq V$)
- Where are we now? (The last city $j \in S$)
- $DP(S, j)$ = The minimum cost of a path starting at city 1, visiting **all** cities in the set S (and only cities in S) exactly once, and ending at city j .

We will assume the tour starts at city 1. So, $1 \in S$ for all states.

Held-Karp: Recurrence

Recurrence: We want to compute $DP(S, j)$.

- The path must have arrived at j from some other city $i \in S \setminus \{j\}$.
- The path to i must have visited all cities $\in S \setminus \{j\}$. The cost of this path is, by definition, $DP(S \setminus \{j\}, i)$.
- We add the cost of the last edge: $+ C_{ij}$.
- We want the *minimum over* all possible “penultimate” (倒数第二) cities i .
- $DP(S, j) = \min_{i \in S \setminus \{j\}} DP((S \setminus \{j\}, i) + C_{ij}$

Held-Karp: Base case

Base cases: the smallest subproblems are paths of length 1 (i.e., $|S| = 2$, only two cities).

- State: $DP(\{1, j\}, j)$ for $j \neq 1$.
- Meaning: The cost of a path from 1, visiting only $\{1, j\}$, and ending at j .
- This is just the direct edge cost: $DP(\{1, j\}, j) = C_{1j}$

Held-Karp: Finding the Optimal Tour Cost

After we compute the DP table, $DP[V, j]$ now contains the cost of a path visiting all cities and ending at j .

To complete the tour, we must add the cost of the edge returning to city 1.

$$\text{Optimal_Cost} = \min_{j \in V \setminus \{1\}} DP(V, j) + C_{j1}$$

Why is this the optimal cost?

Held-Karp Algorithm : Pseudocode

Algorithm Held_Karp_TSP(C, n):

// 1. Initialize DP table

DP = create_table(2^n , n, infinity)

// 2. Base Case

for j from 2 to n:

 DP[{1, j}][j] = C[1][j]

// 3. Fill table by increasing subset size

for s from 3 to n:

 for S in (subsets of V containing 1 of size s):

 for j in S where $j \neq 1$:

 min_cost = infinity

 for i in S where $i \neq j$:

 cost = DP[S \ {j}][i] + C[i][j]

 if cost < min_cost:

 min_cost = cost

 DP[S][j] = min_cost

// 4. Calculate final answer

min_tour = infinity

V = {1, ..., n} // The full set

for j from 2 to n:

 tour = DP[V][j] + C[j][1]

 if tour < min_tour:

 min_tour = tour

return min_tour

Held-Karp Algorithm: Analysis

DP Table Size $O(n \cdot 2^n)$

- Number of subsets $S \subseteq V: 2^n$, Number of choices for $j \in V: n$

Work per State $O(n)$:

- To compute $DP(S, j)$, we iterate over all $i \in S$

Total Runtime: $O(n^2 \cdot 2^n)$

Memory $O(n \cdot 2^n)$:

- This is often also the bottleneck. For $n = 30$, $n \cdot 2^n \approx 3 \times 10^{10}$, requiring $3 \times 10^{10} \times 4 \text{ Bytes} > 120 \text{GB}$ of RAM.

Conclusion:

- This is exponential-time DP algorithm
- The $O(n^2 \cdot 2^n)$ runtime is vastly better than the $O(n!)$ brute-force

TSP Solution Reconstruction/Traceback (1)

- The simple DP table only stores the cost, not *which* i gave the minimum.
- We must maintain a second table: `Path[S][j]`.
- **Modification to DP Algorithm:**

```
for s from 3 to n:
  for S in (subsets...):
    for j in S where j != 1:
      min_cost = infinity
      best_i = null
      for i in S \ {j}:
        cost = DP[ S\{j} ][i] + C[i][j]
        if cost < min_cost:
          min_cost = cost
          best_i = i
      DP[S][j] = min_cost
      Path[S][j] = best_i  // <-- STORE THE PREDECESSOR
```

TSP Solution Reconstruction/Traceback (2)

- 1) Find the last city $j = \operatorname{argmin}_{k \in V \setminus \{1\}} DP(V, k) + C_{k1}$
- 2) Initialize $\text{Tour} = [j]$ and Current set $V = \{1, \dots, n\}$

3)

Loop $n-1$ times:

- $i = \text{Path}[\text{current_set}][j]$ (Get the predecessor from the table).
- Append i to the Tour list.
- $\text{current_set} = \text{current_set} \setminus \{j\}$ (Update the set).
- $j = i$ (Update the current city).

Tour is now $[j^*, i^*, \dots, 1]$.

Return $\text{reverse}(\text{Tour})$. (e.g., $[1, \dots, i^*, j^*]$).

Runtime: $O(n)$

DP for the Set Covering Problem with Cost (1)



DP for a Covering Problem

Given:

- A "universe" of m elements, $U = \{1, \dots, m\}$.
- A collection of n sets, $\mathcal{S} = \{S_1, \dots, S_n\}$, where each $S_j \subseteq U$.
- Each set S_j has a cost c_j .

Goal:

- Find a sub-collection of sets $\mathcal{C} \subseteq \mathcal{S}$ that covers all elements in U (i.e., $\cup_{S \in \mathcal{C}} S = U$)
- ... such that the total cost $\sum_{S \in \mathcal{C}} c_j$ is minimized.

This problem is also **strongly NP-hard**.

DP for the Set Covering Problem with Cost (2)

Like TSP, the state must capture “what we've accomplished so far”. Here, it’s “which elements have we covered”.

State: $DP(i, M)$ = The **minimum cost** to cover exactly the set of elements represented by the bitmask M (a subset of the m elements), using sets only from $\{S_1, S_2, \dots, S_i\}$

DP for the Set Covering Problem with Cost (2)

Recurrence: Consider $DP(i, M)$ and set S_i with cost c_i

- **Case 1:** Don't use set S_i : We must have already covered M using only the previous $i - 1$ sets. $DP(i, M) = DP(i - 1, M)$
- **Case 2:** Use set S_i . We pay the cost c_i . This set S_i will cover the elements in $M \cap S_i$. That is to say, we must have already covered the remaining elements, $M \setminus S_i$, using the previous $i - 1$ sets. $DP(i, M) = c_i + DP(i - 1, M \setminus S_i)$
- $DP(i, M) = \min\{DP(i - 1, M), c_i + DP(i - 1, M \setminus S_i)\}$

DP for the Set Covering Problem with Cost (3)



Base cases:

- $DP(0, \emptyset) = 0$ (Cost to cover nothing with no sets is 0)
- $DP(0, M) = \infty$ for all $M \neq \emptyset$

DP for the Set Covering Problem with Cost: Analysis



DP Table Size $O(n \cdot 2^m)$

Work per State $O(1)$:

Total Runtime: $O(n \cdot 2^m)$

Conclusion:

- This is exponential-time DP algorithm
- The $O(n \cdot 2^m)$ runtime vs. the $O(m \cdot n \cdot 2^n)$ brute-force

We have seen

- DP on a prefix: $DP(i, \dots)$ (Knapsack, Partition)
- DP on a subset: $DP(S, \dots)$ (TSP, Covering)

A third common pattern is DP on **an interval or range**.

- State: $DP(i, j)$ = Optimal solution for the subproblem defined on items $i, i + 1, \dots, j$.
- Recurrence: Usually involves splitting the interval at some point k : $DP(i, j) = \min_{\{i < k < j\}} (DP(i, k) + DP(k + 1, j) + \dots)$

This is **not** a pattern typically to get exact solutions for NP-Hard problems

- Number of states: $O(n^2)$, work per state (looping over k): $O(n)$
- Typical Total runtime: $O(n^3)$, this is “too fast” for NP-hard problems
- Examples: Matrix Chain Multiplication, Optimal Binary Search Tree (OBST)

DP: A Multi-Faceted Tool



DP is a General Technique: It provides exact solutions by systematically exploring a DP table (or called a state-space graph, or a directed acyclic graph).

A simple traceback algorithm can always reconstruct the solution from the value table.

DP & Complexity: The state structure determines the complexity.

Summary

Problem	State Definition Trick	DP Type
0/1 KP	“Prefix + Constraint”: The state must include a dimension to track the limited resource (weight).	Pseudo-polynomial
	“Swap Objective & Constraint”: Move the objective (value) into the state and optimize the constraint (weight).	Pseudo-polynomial
Partition	“Prefix + Target Value”: Similar to Knapsack (by Weight), but the optimization is a boolean feasibility.	Pseudo-polynomial
TSP	“Subset + Endpoint”: The state must encode which elements have been visited (subset S) and the current city (j).	Exponential
Set Cover	“Prefix + Subset”: The state encodes the elements covered (subset M), using a prefix of the available items (sets).	Exponential

Assignment 2 (Part 2)

Q3. Design a DP algorithm for the maximum independent set problem (MIS)

The Problem: Maximum Independent Set (MIS)

- **Definition:**
 - Given an undirected graph $G = (V, E)$, an **Independent Set** is a subset of vertices $S \subseteq V$ such that no two vertices in S are connected by an edge (i.e., for any $u, v \in S$, the edge $(u, v) \notin E$).
 - The **Maximum Independent Set** problem asks to find an independent set S with the maximum possible size, $|S|$.
- **Your Task:** You must design a Dynamic Programming algorithm that finds the **size** of the maximum independent set for a general graph G with n vertices. Your algorithm is expected to run in exponential time.

具体要求请看下一页 (Please refer to the next page for detailed requirements)

Assignment 2 (Part 2) – cont' d



Required Components: You must provide a complete solution that includes the following five parts:

1. **DP State Definition:**

- Clearly define your DP state (e.g., $DP(S)$).

2. **DP Recurrence Relation:**

- Provide the mathematical formula and briefly explain its logic.

3. **Base Cases:**

- Define the starting values for your recurrence.

4. **Pseudocode:**

- Write high-level pseudocode for the algorithm. You must specify the order in which the DP table should be filled.

5. **Complexity Analysis:**

- Analyze the **Time Complexity** and **Space Complexity** of your algorithm, justifying your answer by counting the states and work-per-state.

Project 组队与答辩安排



组队信息

组队人数：

两人一组，共35组

组队时间：

11月5日-11月19日

组队方式：

组好队填写在线表格

<https://docs.qq.com/sheet/DUHp6UWxzakNQaE9P>



答辩要求

答辩形式：

基于Slides做Presentation，
小组每个人讲自己负责
的部分

答辩时长：

共7分钟汇报+3分钟问答

答辩顺序：

随机，在分组确定后公
布顺序



重要日期

Project正式要求细则：

11月12日（下周）发布

答辩时间：

12月10日/17日/24日，
其中24日前1-2节课用于答
辩