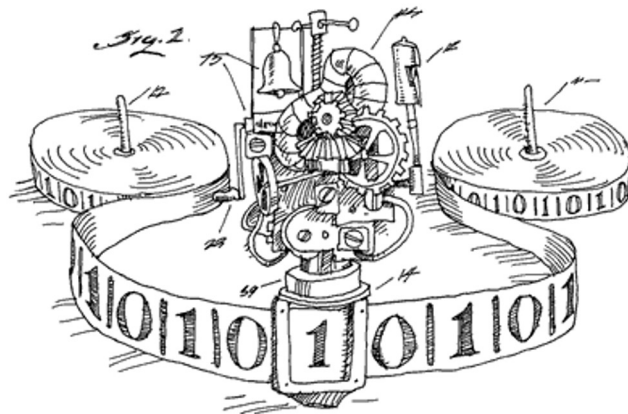


EECS 376: Foundations of Computer Science

Lecture 05 - Dynamic Programming 2



Dynamic Programming Review

- **Step 1:** Write a **recursive formulation** of the solution.
Bound the number of **distinct subproblems** that ever appear in your formulation
- **Step 2:** Create a table representing distinct subproblems.
Fill in the table from the **bottom-up.**
- **Runtime:** (#subproblem \times time per subproblem)

Longest Common Subsequence

Motivation: DNA Comparison



- Your DNA is a (*long*) string over {A, T, C, G}.
- “Humans and chimps are 98.9% similar.”
 - *X*: ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
 - *Y*: GTCGTTCGGAATGCCGTTGCTCTGTAA
- The length of the longest common subsequence between two genomes is a measure of similarity.

Longest Common Subsequence

- Given strings $X[1..m]$ and $Y[1..n]$
- **Goal:** find the length of a **longest common subsequence** of X and Y
 - A **subsequence** of X is a string obtainable from X by deleting chars (may not be consecutive)
 - A **common subsequence** of X and Y is a subsequence of both X and Y
- **Example:**
 - “CT” is a common subsequence of “CGATG” and “CATGT”.
 - **Q:** What’s the longest?
- **Q:** What’s a brute force solution?
 - Each character of X and Y is either deleted or not:
Runtime: $O(2^{m+n})$

Recurrence for *LCS*: First, define the function

- **Def:** $LCS(i, j)$ = length of a LCS of $X[1..i]$ and $Y[1..j]$.
 - $i = 0$ means X is the empty string
 - $j = 0$ means Y is the empty string
- **Goal:** $LCS(m, n)$ (compute the length of LCS, will compute LCS itself soon)
- **Example:** Suppose $X = \text{"ATGCC"}$ and $Y = \text{"TAGC"}$.
 - **Q:** What's $LCS(1,0)$? 0
 - **Q:** What's $LCS(5,3)$? 2 TG or AG
 - **Q:** What's $LCS(4,4)$? 3 TGC or AGC
 - **Q:** What's $LCS(5,4)$? 3 TGC or AGC

Recurrence for LCS

- **Def:** $LCS(i, j)$ = length of a LCS of $X[1..i]$ and $Y[1..j]$.
 - $i = 0$ means X is the empty string
 - $j = 0$ means Y is the empty string
- **Goal:** return $LCS(m, n)$
- What is a recursion for $LCS(i, j)$?

$$LCS(i, j) = \begin{cases} ? & \text{if } i = 0 \text{ or } j = 0 \\ \text{Pause and Think...} & \begin{array}{l} \text{if } X[i] = Y[j] \\ \text{if } X[i] \neq Y[j] \end{array} \end{cases}$$

Def: $LCS(i, j) =$
length of a LCS of $X[1..i]$ and $Y[1..j]$.

Recurrence for LCS

(所有 optimal sol 中的 char 数都一样, 因而可以放心删)

那么有某个 optimal sol 中包含它

- **Case 1:** $X[i] = Y[j]$ (ends with the same character)

因而把它 delete 掉 (即 $i, j--$)

- **Example:** $X[1..i] = \text{"CTGCA"}$ and $Y[1..j] = \text{"TCGA"}$

length ++

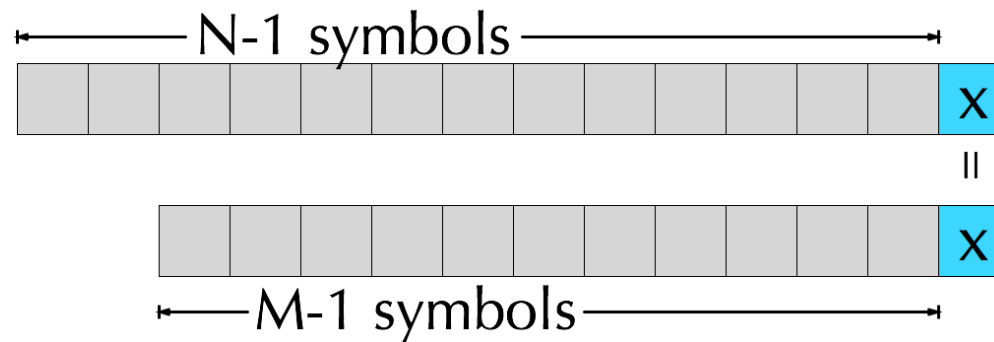
- **Claim.** There exists an optimal solution OPT that matches $X[i]$ and $Y[j]$.

然后看下一个

- **Proof by contradiction.**

- Suppose for contradiction: all optimal sol OPT do not match $X[i]=Y[j]$ (Say, $X[i] = \text{"A"}$).
- OPT + "A" is an LCS too. But OPT + "A" is longer than OPT.
- So, OPT is not optimal, contradiction.

- $LCS(i, j) = 1 + LCS(i - 1, j - 1)$



因为这是最后一个字符，如果两个都在那么

它们一定是一样的

Def: $LCS(i, j)$ =
length of a LCS of $X[1..i]$ and $Y[1..j]$.

↑ Recurrence for LCS
那么 $X[i]$ 和 $Y[j]$ 中至少有一个不在

$$LCS(i, j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ 1 + LCS(i - 1, j - 1) & X[i] = Y[j] \\ ? & X[i] \neq Y[j] \end{cases}$$

optimal sol 中

- **Case 2:** $X[i] \neq Y[j]$ (end with different characters)
 - **Example:** $X[1..i] = \text{"GTCA"}$ and $Y[1..j] = \text{"GTC"}$

- So, either $X[i]$ or $Y[j]$ is not part of LCS
- **Q:** How do we know which one?
 - Try both! And take the better one

选取 better one : 即到底哪个在 optimal sol 中 (可以都不在)。即: 去掉 $X[i]$ 大

- $LCS(i, j) = \max\{LCS(i - 1, j), LCS(i, j - 1)\}$ 还是去掉 $Y[j]$ 大

Recurrence for *LCS*

- **Def:** $LCS(i, j)$ = length of a LCS of $X[1..i]$ and $Y[1..j]$.
 - $i = 0$ means X is the empty string
 - $j = 0$ means Y is the empty string

- **Goal:** return $LCS(m, n)$

- We have:
$$LCS(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + LCS(i - 1, j - 1) & \text{if } X[i] = Y[j] \\ \max \begin{cases} LCS(i - 1, j) \\ LCS(i, j - 1) \end{cases} & \text{if } X[i] \neq Y[j] \end{cases}$$

- **Q:** How many subproblems does this recurrence generate?
- **Q:** how much time does it take per subproblem?

$(|X||Y|)$

mn

$O(1)$

Example

Table of subproblems:
2-dimensional!

Task: compute LCS of

- “APOCRYPHAL”
- “POLYPEPTIDE”

Col: prefix of “APOCRYPHAL”

Row: prefix of
“POLYPEPTIDE”

LCS between
empty strings is 0

| | | A | P | O | C | R | Y | P | H | A | L |
|---|---|---|---|---|---|---|----|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| P | 0 | | | | | | | | | | |
| O | 0 | | | | | | | | | | |
| L | 0 | | | | | | | | | | |
| Y | 0 | | | | | | | | | | |
| P | 0 | | | | | | | | | | |
| E | 0 | | | | | | | | | | |
| P | 0 | | | | | | | | | | |
| T | 0 | | | | | | | | | | |
| I | 0 | | | | | | | | | | |
| D | 0 | | | | | | | | | | |
| E | 0 | | | | | | 19 | | | | |

5/14/24

Table Filling

$$LCS(i, j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ 1 + LCS(i - 1, j - 1) & X[i] = Y[j] \\ \max \{LCS(i - 1, j), LCS(i, j - 1)\} & X[i] \neq Y[j] \end{cases}$$

- $LCS(X, Y)$

table = **2D-array**, indexed from 0 to $|X|$ and 0 to $|Y|$

table $[i, j]$ stores $LCS(X[1 \dots i], Y[1 \dots j])$

For $i = 0, \dots, |X|$: table $[i, 0] = 0$ Base Case

For $j = 0, \dots, |Y|$: table $[0, j] = 0$

For i, j in which order?

...

Return ??

| | | A | P | O | C | R | Y | P | H | A | L |
|---|---|----|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| P | 0 | | | | | | | | | | |
| O | 0 | | | | | | | | | | |
| L | 0 | | | | | | | | | | |
| Y | 0 | | | | | | | | | | |
| P | 0 | | | | | | | | | | |
| E | 0 | | | | | | | | | | |
| P | 0 | | | | | | | | | | |
| T | 0 | | | | | | | | | | |
| I | 0 | | | | | | | | | | |
| D | 0 | | | | | | | | | | |
| E | 0 | 20 | | | | | | | | | |

Table Filling

$$LCS(i, j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ 1 + LCS(i - 1, j - 1) & X[i] = Y[j] \\ \max\{LCS(i - 1, j), LCS(i, j - 1)\} & X[i] \neq Y[j] \end{cases}$$

$O(|X||Y|)$ operations total

- $LCS(X, Y)$

table = **2D-array**, indexed from 0 to $|X|$ and 0 to $|Y|$
table $[i, j]$ stores $LCS(X[1 \dots i], Y[1 \dots i])$

For $i = 0, \dots, |X|$: table $[i, 0] = 0$ Base Case

For $j = 0, \dots, |Y|$: table $[0, j] = 0$

For $i = 1, \dots, |X|$:

$O(|X||Y|)$ rounds of loop

For $j = 1, \dots, |Y|$:

if $X[i] = Y[j]$ **then**

 table $[i][j] \leftarrow 1 + \text{table}[i - 1][j - 1]$

else

 table $[i][j] \leftarrow \max\{\text{table}[i - 1][j], \text{table}[i][j - 1]\}$

Return table $[|X|, |Y|]$

$O(1)$ operations/loop

Longest Common Subsequence, via Backtracking

Input: strings X and Y

Output: a longest common subsequence of the strings

function LCS($X[1 \dots |X|]$, $Y[1 \dots |Y|]$)

table = **2D-array**, indexed from 0 to $|X|$ and 0 to $|Y|$

table[i, j] stores LCS($X[1 \dots i]$, $Y[1 \dots j]$)

$s = \varepsilon$

◁ the empty string

$i = |X|$, $j = |Y|$

while $i > 0$ and $j > 0$ **do**

if $X[i] = Y[j]$ **then**

$s = X[i] + s$

$i = i - 1$, $j = j - 1$

else if table[i][$j - 1$] > table[$i - 1$][j] **then**

$j = j - 1$

else

$i = i - 1$

return s

Example

Task: compute LCS of

- "APOCRYPHAL"
- "POLYPEPTIDE"

| | | A | P | O | C | R | Y | P | H | A | L |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| P | 0 | 0 | 1 | 1 | 1 | 1 | 1 | ? | | | |
| O | 0 | | | | | | | | | | |
| L | 0 | | | | | | | | | | |
| Y | 0 | | | | | | | | | | |
| P | 0 | | | | | | | | | | |
| E | 0 | | | | | | | | | | |
| P | 0 | | | | | | | | | | |
| T | 0 | | | | | | | | | | |
| I | 0 | | | | | | | | | | |
| D | 0 | | | | | | | | | | |
| E | 0 | | | | | | | | | | |

$= 1 + 0$
 $\max(1, 0)$

Example

Task: compute LCS of

- "APOCRYPHAL"
- "POLYPEPTIDE"

| | | A | P | O | C | R | Y | P | H | A | L |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| P | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| O | 0 | 0 | | | | | | | | | |
| L | 0 | | | | | | | | | | |
| Y | 0 | | | | | | | | | | |
| P | 0 | | | | | | | | | | |
| E | 0 | | | | | | | | | | |
| P | 0 | | | | | | | | | | |
| T | 0 | | | | | | | | | | |
| I | 0 | | | | | | | | | | |
| D | 0 | | | | | | | | | | |
| E | 0 | | | | | | | | | | |

$? = 1 + 0$

$= 1 + 0$

$= \max(1, 0)$

5/14/24

考虑了X中的P和
每个Y中的P分别match时的情况

27

Example

Task: compute LCS of

- "APOCRYPHAL"
- "POLYPEPTIDE"

| | | A | P | O | C | R | Y | P | H | A | L |
|---|---|---|---|-----|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| P | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| O | 0 | 0 | 1 | 2 | | | | | | | |
| L | 0 | | | 1+1 | | | | | | | |
| Y | 0 | | | | | | | | | | |
| P | 0 | | | | | | | | | | |
| E | 0 | | | | | | | | | | |
| P | 0 | | | | | | | | | | |
| T | 0 | | | | | | | | | | |
| I | 0 | | | | | | | | | | |
| D | 0 | | | | | | | | | | |
| E | 0 | | | | | | | | | | |

Example

Task: compute LCS of

- “APOCRYPHAL”
- “POLYPEPTIDE”

Q: What is the length of LCS?

A: 4

Q: What is the LCS itself?

A: **POYP**

Exercise: Think how to edit the program so that it can return the LCS itself.

| | | A | P | O | C | R | Y | P | H | A | L |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| P | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| O | 0 | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| L | 0 | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 |
| Y | 0 | 0 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 |
| P | 0 | 0 | 1 | 2 | 2 | 2 | 3 | 4 | 4 | 4 | 4 |
| E | 0 | 0 | 1 | 2 | 2 | 2 | 3 | 4 | 4 | 4 | 4 |
| P | 0 | 0 | 1 | 2 | 2 | 2 | 3 | 4 | 4 | 4 | 4 |
| T | 0 | 0 | 1 | 2 | 2 | 2 | 3 | 4 | 4 | 4 | 4 |
| I | 0 | 0 | 1 | 2 | 2 | 2 | 3 | 4 | 4 | 4 | 4 |
| D | 0 | 0 | 1 | 2 | 2 | 2 | 3 | 4 | 4 | 4 | 4 |
| E | 0 | 0 | 1 | 2 | 2 | 2 | 3 | 4 | 4 | 4 | 4 |

Time and Space Complexity

- How efficient is this algorithm? Computing a single table entry requires a constant number of operations. Since there are $(N + 1) \cdot (M + 1)$ entries, constructing the table takes $O(NM)$ time and requires $O(NM)$ space.
- Backtracking also does a constant number of operations per entry on the path, and the path length is at most $N + M + 1$, so backtracking takes $O(N + M)$ time.
- Thus, the total complexity of this algorithm is $O(NM)$ in both time and space.

0-1 KNAPSACK

0-1 Knapsack Problem

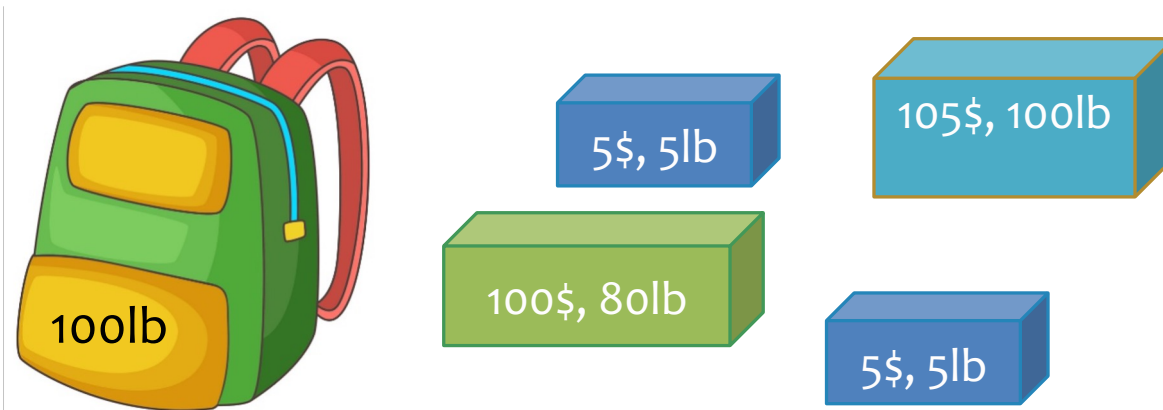
Input:

- n items t_1, \dots, t_n : each with **value** v_i and **weight** w_i .
- W : the total capacity of the 0-1 knapsack.
- All v_i, w_i, W are integers.

$$t_i = \{ \overset{\text{value}}{\uparrow} v_i, w_i \downarrow \text{weight} \}$$

Output: A set of items $S \subseteq \{1, \dots, n\}$ such that

- (not too heavy) $\sum_{i \in S} w_i \leq W$
- (max value) $\sum_{i \in S} v_i$ as large as possible



$Knapsack(\{t_1, \dots, t_n\}, W)$

Recursive Strategy:
Take t_n ?

Reject: $val_{rej} = Knapsack(\{t_1, \dots, t_{n-1}\}, W)$

Accept: $val_{acc} = v_n + Knapsack(\{t_1, \dots, t_{n-1}\}, W - w_n)$

(reject: 不拿 t_i , 不损失 weight, 取 $i-1$ 个物品时的最优解.)

- Do not know whether optimal solution actually has t_n
 - Again, try both and take the better one.

- The recurrence:**

$$Knapsack(\{t_1, \dots, t_n\}, W) = \max \begin{cases} Knapsack(\{t_1, \dots, t_{n-1}\}, W) \\ v_n + Knapsack(\{t_1, \dots, t_{n-1}\}, W - w_n) \end{cases}$$

- The base cases:**

- $Knapsack(\emptyset, W') = 0$ for all W'
- $Knapsack(\{t_1, \dots, t_i\}, 0) = 0$ for all i

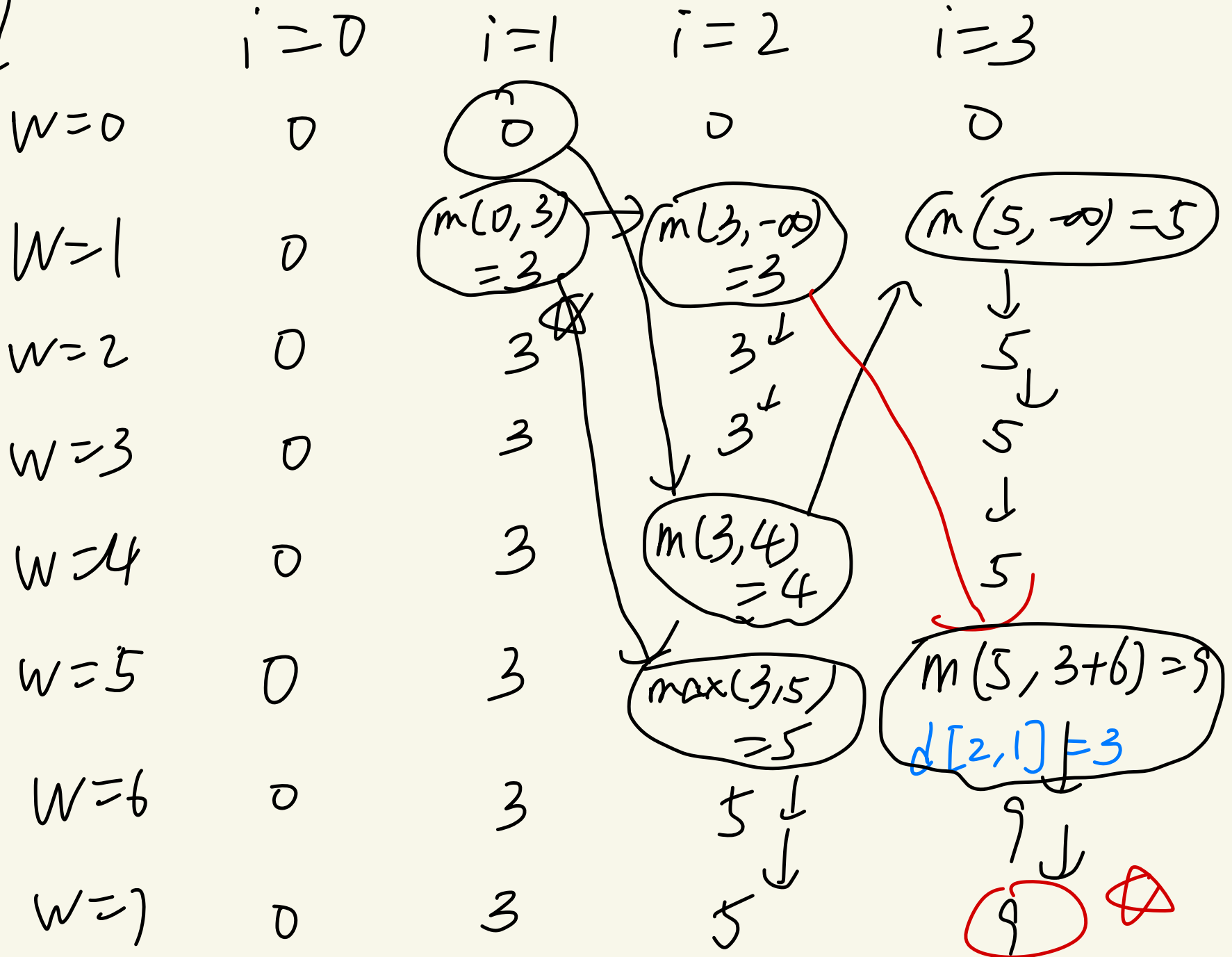
Valid only when $W \geq w_n$

- What is the size of the table?
- What is time spent per cell?

Accept: 拿 t_i , 但是要牺牲一点 weight, 因而是 fill 满 $W - w_i$ 的最优解 + 重为 w_i 的 t_i 的 v_i

$t_1: V_1=3, W_1=1$; $t_2: V_2=5, W_2=4$; $t_3: V_3=6, W_3=4$

$W=7$



Knapsack Solution

- *Knapsack*($\{t_1, \dots, t_n\}, W$)

$O(nW)$ ops

table = **2D-array**, indexed from 0 to n and 0 to W

table[i, W'] stores subproblem Knapsack ($\{t_1, \dots, t_i\}, W'$)

For $j = 0, \dots, n$:

– table[$j, 0$] = 0 Base Case 1: no weight to carry \rightarrow no value

For $k = 0, \dots, W$:

– table[$0, k$] = 0 Base Case 2: no items to choose from \rightarrow no value

For $j = 1, \dots, n$:

$O(nW)$ loops

For $k = 1, \dots, W$:

reject = table[$j - 1, k$]

accept = ($v_j + \text{table}[j - 1, k - w_j]$) if $k \geq w_j$ else $-\infty$

table[j, k] = max{reject, accept}

Return table[n, W]

$O(1)$ operations/loop

Is this polynomial in input size?

- What is the “input size” in term of n and W ?
- Is there $\text{poly}(n, \log W)$ -time algorithm?
 - **No**, unless $P = NP$.
 - You will learn about “ P vs. NP ” after the midterm.

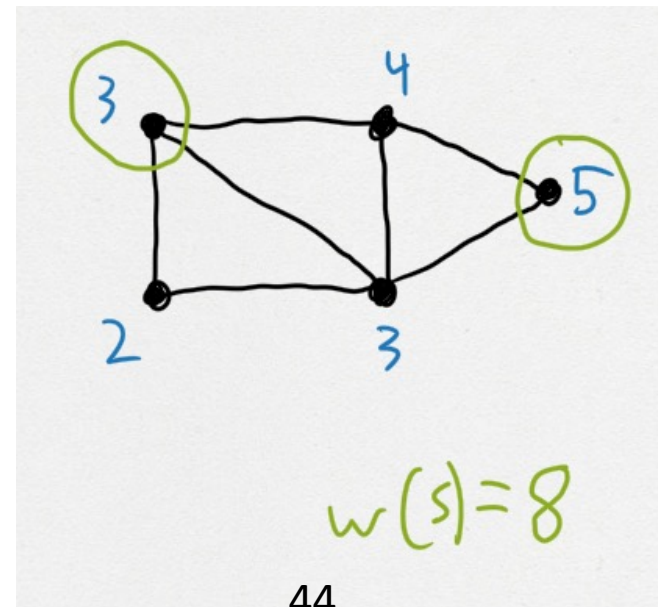
Wrap Up

- We have seen more examples of dynamic programming
- **Next:**
 - Using graphs (instead of 1D/2D tables) in dynamic programming algo.
 - For: shortest path problems.

BONUS:
MAXIMUM WEIGHT INDEPENDENT SET
OF TREES

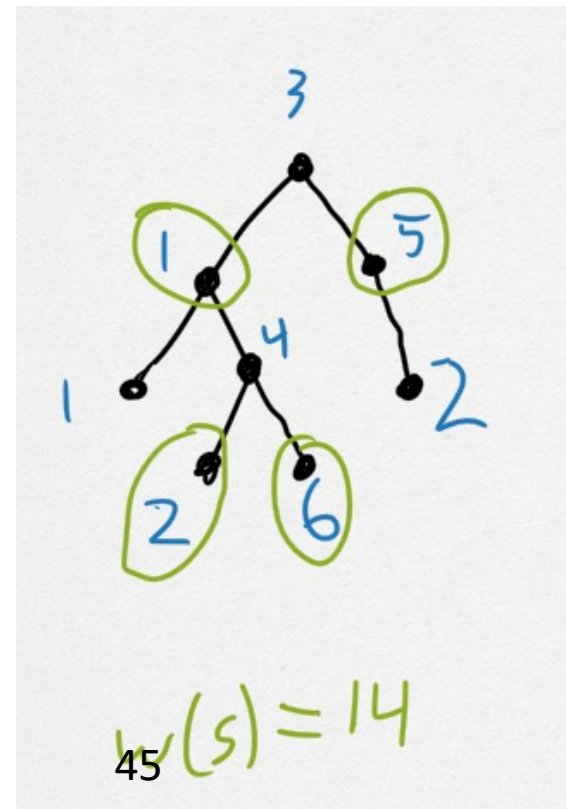
Max Weight Independent Set

- **Input:** Graph G with a weight w_v on each **node** v
- A set of nodes S is **independent** if no edges between nodes in S .
- **Output:** **Independent set** S maximizing $\sum_{s \in S} w_s$
- NP-Hard ☹️



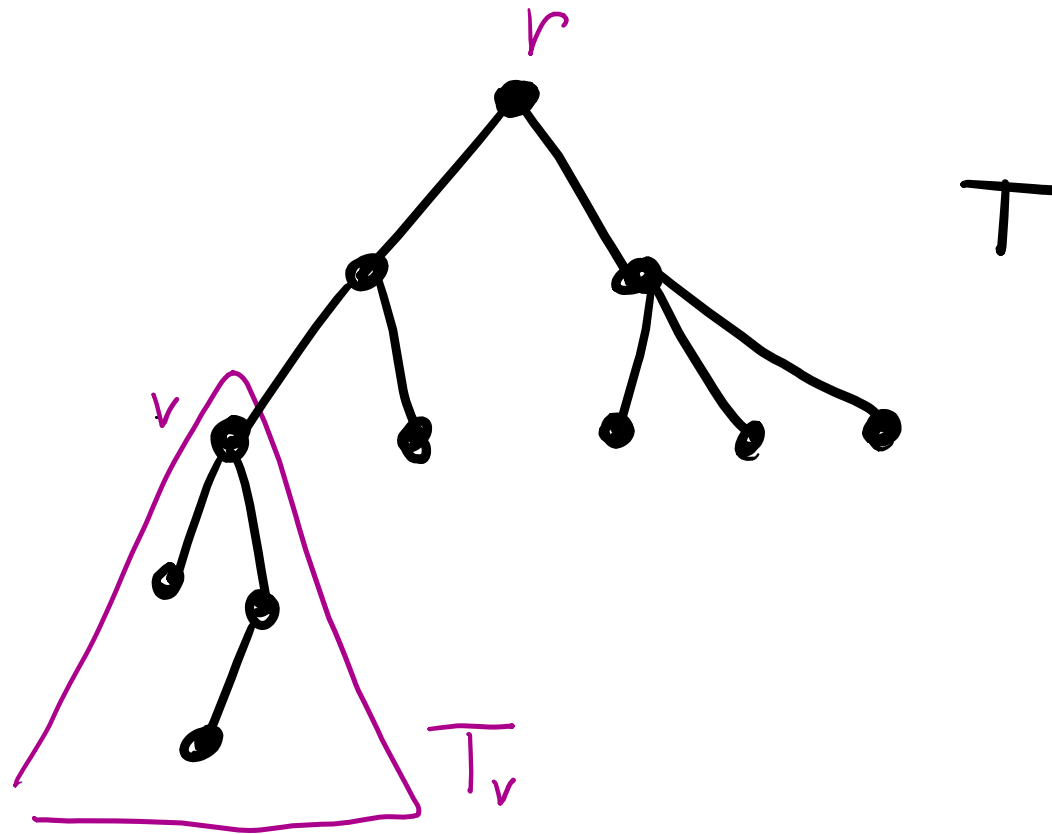
Max Weight Independent Set **On Trees**

- **Input:** Tree T with a weight w_v on each **node** v
- A set of nodes S is **independent** if no edges between nodes in S .
- **Output:** Independent set S maximizing $\sum_{s \in S} w_s$
- Admits linear-time algo!



Coming up with the recurrence

- $\text{MWIS}(v)$: max weight independent set of subtree T_v rooted at v
- **Goal:** $\text{MWIS}(r)$ where r is the root of T

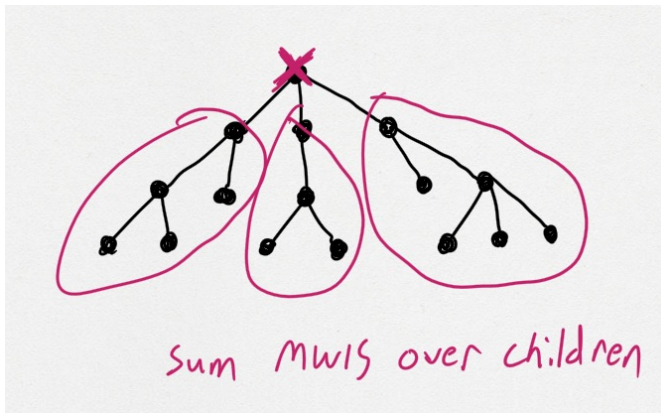


Coming up with the recurrence

- $MWIS(v)$: max independent set of subtree T_v rooted at v
- **Goal:** $MWIS(r)$ where r is the root of T

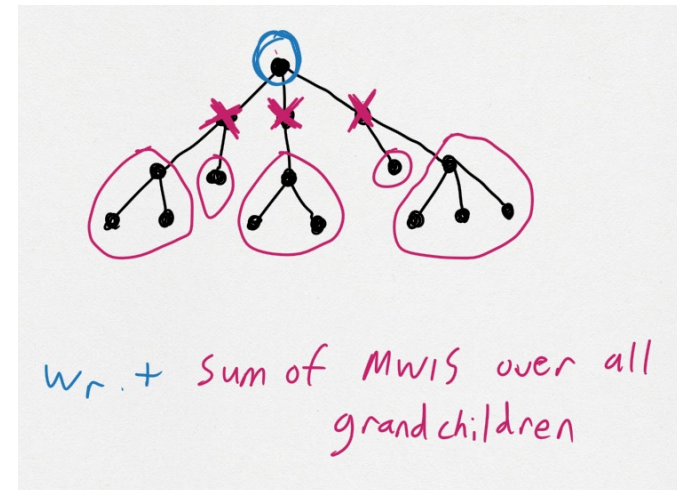
Recursive Strategy:
Consider the root node r .
Include r in S ?

Reject root r



$$MWIS(r) = \sum_{v \text{ child of } r} MWIS(v)$$

Accept root r



$$MWIS(r) = w_r + \sum_{v \text{ grandchild of } r} MWIS(v)$$

Max Weight Independent Set **On Trees**

- $MWIS(T)$ assume T nonempty
Computes max possible total weight of $MWIS$
(Could easily tweak algorithm to also output that independent set)

r = root node of T

table = array, indexed by tree nodes memo[v] stores $MWIS$ of subtree rooted at v

For each node v in T , **in which order???**

....

Return table[r]

Max Weight Independent Set On Trees

- MWIS(T) assumes T is nonempty
Computes max possible total weight of MWIS
(Could easily tweak algorithm to also output that independent set)

r = root node of T

table = array, indexed by tree nodes memo[v] stores *MWIS* of subtree rooted at v

For each node v in T, in ascending tree order process all descendants before v itself

reject $\leftarrow \sum_{c \text{ child of } v} \text{table}[c]$

don't take v_i

accept $\leftarrow w_v + \sum_{g \text{ grandchild of } v} \text{table}[g]$

take v_i , reject its children

$\text{table}[v] \leftarrow \max\{\text{reject}, \text{accept}\}$

Return table[r]

n loops

Ops/loop depends on # of
children/grandchildren

Tree of branching factor **b** has $O(nb^2)$ ops total

Max Weight Independent Set **On Trees**

- **Today:**
 - $O(nb^2)$ time
- **Challenge:**
 - $O(n)$ time
 - **Hint: Same algorithm. Smarter analysis.**
Show that $\sum_{v \in T} |\text{grandchildren}(v)| = O(n)$

Upshot:

Dynamic Programming on Trees

Many hard problems
usually become easy on trees
via dynamic programming

Longest Palindromic Substring

(Another classic coding problem)

- Given a string $X[1..n]$
- **Goal:** Find the length of the longest palindrome in X .
 - A **palindrome** is a string that's equal to its reverse
- **Example:** “aba”, “aca”, and “ada” are the longest palindromes in $X = \text{“abacada”}$ (**Note:** “aaca” doesn't count!)
- **Q:** What's a brute force algorithm?
 - Try every substring $X[i..j]$

Recurrence for PAL

- Given a string $X[1..n]$
- Let $PAL(i, j)$ be a Boolean (T/F) value for whether $X[i..j]$ is a palindrome

Sometimes the problem we solve might seem quite “different”

$$PAL(i, j) = \begin{cases} X[i] = X[j] & j \leq i + 1 \\ X[i] = X[j] \text{ and } PAL(i + 1, j - 1) & j > i + 1 \end{cases}$$

Q: Given this recurrence, how do we find the length of a longest palindrome?