香港科技大学(广州)
THE HONG KONG
UNIVERSITY OF SCIENCE AND
TECHNOLOGY (GUANGZHOU)

Design and Analysis of Algorithms

Jing Tang | DSAA 2043 Fall 2024

Dynamic Programming

Last time

Dynamic Programming!

- Not coding in an action movie.

Last time

- Dynamic programming is an algorithm design paradigm.
- Basic idea:
  - Identify **optimal sub-structure**
    - Optimum to the big problem is built out of optima of small sub-problems
  - Take advantage of **overlapping sub-problems**
    - Only solve each sub-problem once, then use it again and again
  - Keep track of the solutions to sub-problems in a table as you build to the final solution.

Today

- Examples of dynamic programming:

    1. Longest common subsequence
    2. Independent sets in trees
    3. Balanced partition problem

The goal of this lecture

- For you to get really bored of dynamic programming

Longest Common Subsequence (LCS)

- A subsequence of a sequence/string *S* is obtained by deleting zero or more symbols from *S*.
- For example, the following are some subsequences of “president”: pred, sdn, predent. In other words, the letters of a subsequence of S appear in order in *S*, but they are not required to be consecutive.
- The longest common subsequence problem is to find a maximum length common subsequence between two sequences.

Longest Common Subsequence

- How similar are these two species?

DNA:
AGCCCTAAGGGCTACCTAGCTT

DNA:
GACAGCCTACAAGCGTTAGCTTG

- Pretty similar, their DNA has a long common subsequence:

AGCCTAAGCTTAGCTT

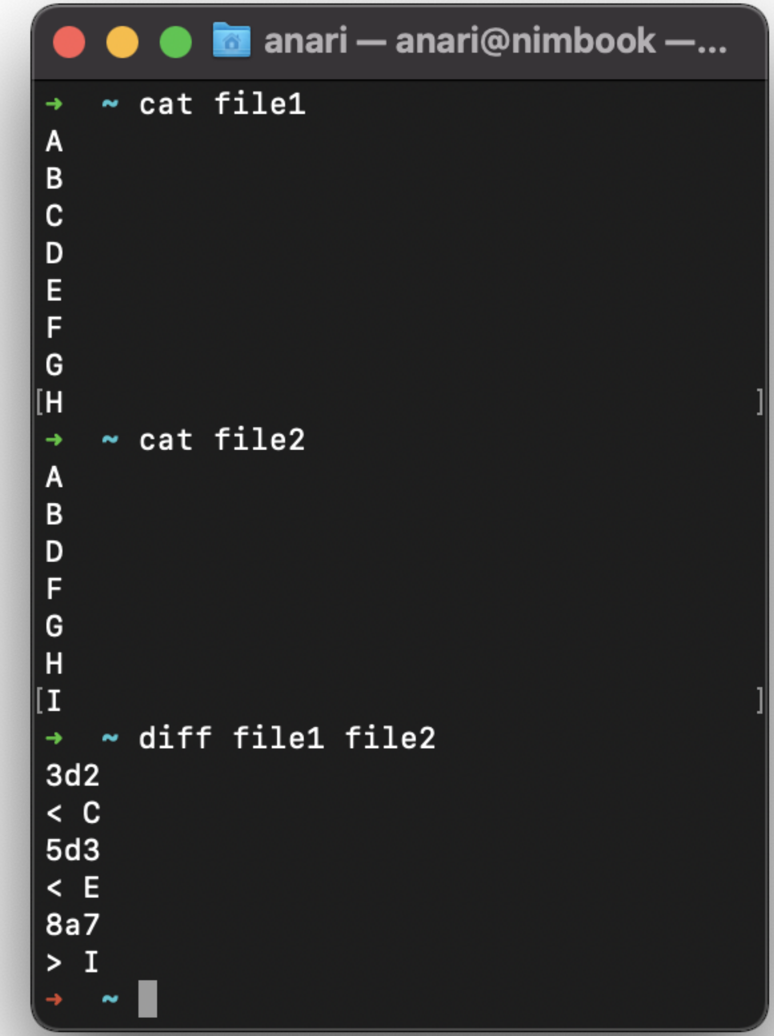
Longest Common Subsequence

- Subsequence:
  - BDFH is a **subsequence** of ABCDEFGH
- If X and Y are sequences, a **common subsequence** is a sequence which is a subsequence of both.
  - BDFH is a **common subsequence** of ABCDEFGH and of ABDFGHI
- A **longest common subsequence**…
  - …is a common subsequence that is longest.
  - The **longest common subsequence** of ABCDEFGH and ABDFGHI is ABDFGH.

We sometimes want to find these

- Applications in bioinformatics

- The unix command diff
- Merging in version control
  - svn, git, etc…



Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
- **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- **Step 5:** If needed, code this up like a reasonable person.

Step 1: Optimal substructure

Prefixes:

X							
	A	C	G	G	T		

Y							
	A	C	G	C	T	T	A

Notation: denote this prefix **ACGC** by Y_4

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i,j] = length_of_LCS(X_i, Y_j)

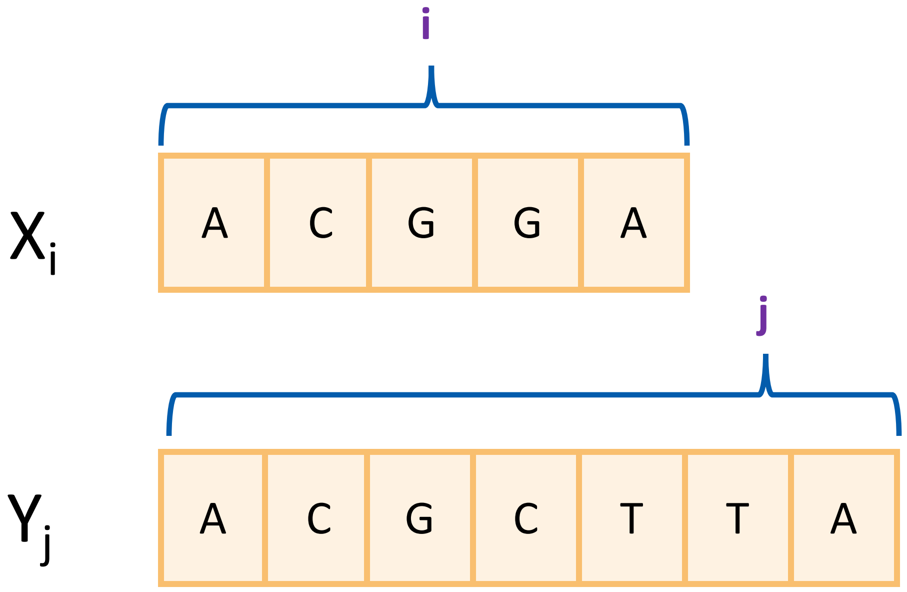
Examples: C[2,3] = 2
C[4,4] = 3

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
- **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- **Step 5:** If needed, code this up like a reasonable person.

Goal

- Write C[i,j] in terms of the solutions to smaller sub-problems



$C[i,j] = \text{length\_of\_LCS}(X_i, Y_j)$

Two cases

Case 1: X[i] = Y[j]

X_i: $\overbrace{\boxed{A}\boxed{C}\boxed{G}\boxed{G}\boxed{A}}^{i}$

Y_j: $\overbrace{\boxed{A}\boxed{C}\boxed{G}\boxed{C}\boxed{T}\boxed{T}\boxed{A}}^{j}$

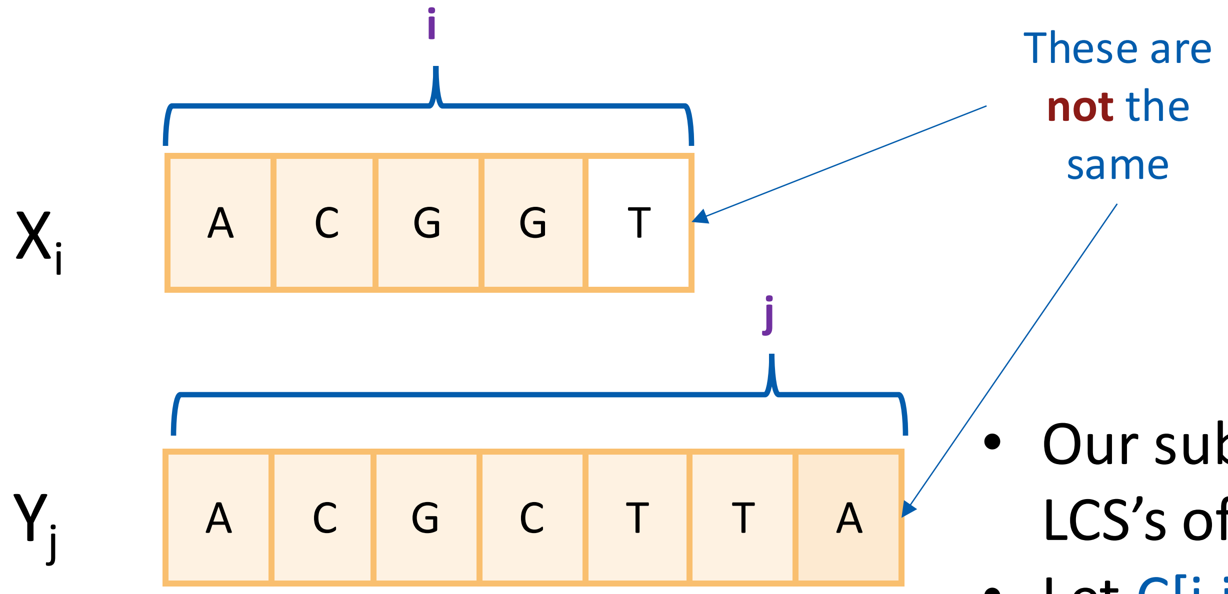
These are the same

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i,j] = length_of_LCS(X_i, Y_j)

- Then C[i,j] = 1 + C[i-1,j-1].
  - because LCS(X_i,Y_j) = LCS(X_{i-1},Y_{j-1}) followed by $\boxed{A}$

Two cases

Case 2: X[i] != Y[j]



- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i,j] = length_of_LCS(X_i, Y_j)

- Then C[i,j] = max{ C[i-1,j], C[i,j-1] }.
  - either LCS(X_i,Y_j) = LCS(X_{i-1},Y_j) and $\boxed{\text{T}}$ is not involved,
  - or LCS(X_i,Y_j) = LCS(X_i,Y_{j-1}) and $\boxed{\text{A}}$ is not involved,
  - (maybe both are not involved, that's covered by the "or").

Recursive formulation of the optimal solution

X_0 | |

Y_j | A | C | G | C | T | T | A |

Case 0

- $$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{\, C[i, j-1], C[i-1, j] \,\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

Case 1

X_i | A | C | G | G | A |

Y_j | A | C | G | C | T | T | A |

Case 2

X_i | A | C | G | G | T |

Y_j | A | C | G | C | T | T | A |

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
- **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- **Step 5:** If needed, code this up like a reasonable person.

LCS DP

- **LCS**(X, Y):
  - C[i,0] = C[0,j] = 0 for all i = 0,…,m, j=0,…n.
  - **For** i = 1,…,m and j = 1,…,n:
    - **If** X[i] = Y[j]:
      - C[i,j] = C[i-1,j-1] + 1
    - **Else:**
      - C[i,j] = max{ C[i,j-1], C[i-1,j] }
  - Return C[m,n]

Running time: O(nm)

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{ C[i, j-1], C[i-1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Example

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{ C[i,j-1], C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Y

		A	C	T	G
	0	0	0	0	0
A	0				
C	0				
G	0				
G	0				
A	0				

X

X	A	C	G	G	A
Y	A	C	T	G	

Example

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{\, C[i, j-1], C[i-1, j] \,\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

X \ Y		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

X	A	C	G	G	A
Y	A	C	T	G	

So the LCS of X and Y has length 3.

Recipe for applying Dynamic Programming

- Step 1: Identify optimal substructure.
- Step 2: Find a recursive formulation for the length of the longest common subsequence.
- Step 3: Use dynamic programming to find the length of the longest common subsequence.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- Step 5: If needed, code this up like a reasonable person.

Example

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{\, C[i, j-1], C[i-1, j] \,\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

X: | A | C | G | G | A |

Y: | A | C | T | G |

X \ Y		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

Example

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{ C[i, j-1], C[i-1, j] \} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Y

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

X

X	A	C	G	G	A
Y	A	C	T	G	

- Once we've filled this in, we can work backwards.

Example

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{\, C[i, j-1], C[i-1, j] \,\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Y

			A	C	T	G
		0	0	0	0	0
X	A	0	1	1	1	1
	C	0	1	2	2	2
	G	0	1	2	2	3
	G	0	1	2	2	3
	A	0	1	2	2	3

That 3 must have come from the 3 above it.

X	A	C	G	G	A

Y	A	C	T	G

- Once we've filled this in, we can work backwards.

Example

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{\, C[i,j-1], C[i-1,j] \,\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

		Y: A	C	T	G
X		0	0	0	0	0
	A	0	1	1	1	1
	C	0	1	2	2	2
	G	0	1	2	2	3
	G	0	1	2	2	3
	A	0	1	2	2	3

This 3 came from that 2 – we found a match!

X: | A | C | G | G | A |

Y: | A | C | T | G |

- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

G

Example

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{ C[i, j-1], C[i-1, j] \} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Y

A	C	T	G

X

X					
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

That 2 may as well have come from this other 2.

X

A	C	G	G	A

Y

A	C	T	G

- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

G

Example

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{ C[i, j-1], C[i-1, j] \} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Y

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

X

X	A	C	G	G	A

Y	A	C	T	G

- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

G

Example

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{ C[i, j-1], C[i-1, j] \} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Y

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

X

X	A	C	G	G	A
Y	A	C	T	G	

- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

C	G

Example

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{ C[i, j-1], C[i-1, j] \} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Y

		A	C	T	G
X		0	0	0	0
	A	0	1	1	1	1
	C	0	1	2	2	2
	G	0	1	2	2	3
	G	0	1	2	2	3
	A	0	1	2	2	3

X: A C G G A

Y: A C T G

- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

A C G

This is the LCS!

Finding an LCS

- Good exercise to write out pseudocode for what we just saw!
  - Or you can find it in lecture notes.
- Takes time O(mn) to fill the table
- Takes time O(n + m) on top of that to recover the LCS
  - We walk up and left in an n-by-m array
  - We can only do that for n + m steps.
- Altogether, we can find LCS(X,Y) in time O(mn).

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
- **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- **Step 5:** If needed, code this up like a reasonable person.

Example

Input X: springtime

Input Y: printing

Max Size: 10 - +

Char Variant: 4 - +

Sequences: Random()

printi

i : 10 j : 8	Xi : e Yj : g	Step: Xi='e' not equal to Yi='g' b[10, 8]='↑' and c[10, 8]=c[9, 8]=6 See line number 13 and 14

	j	0	1	2	3	4	5	6	7	8
i		y	p	r	i	n	t	i	n	g
0	x	0	0	0	0	0	0	0	0	0
1	s	0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0	↑ 0
2	p	0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
3	r	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2
4	i	0	↑ 1	↑ 2	↖ 3	← 3	← 3	↖ 3	← 3	← 3
5	n	0	↑ 1	↑ 2	↑ 3	↖ 4	← 4	← 4	↖ 4	← 4
6	g	0	↑ 1	↑ 2	↑ 3	↑ 4	↑ 4	↑ 4	↑ 4	↖ 5
7	t	0	↑ 1	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	↑ 5
8	i	0	↑ 1	↑ 2	↖ 3	↑ 4	↑ 5	↖ 6	← 6	← 6
9	m	0	↑ 1	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6
10	e	0	↑ 1	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6

LCS-LENGTH(X, Y) Execute LCS Length

```
1  m = length[X]
2  n = length[Y]
3  for i = 1 to m
4       do c[i, 0] = 0
5  for j = 1 to n
6       do c[0, j] = 0
7  for i = 1 to m
8       do for j = 1 to n
9            do if Xi == Yj
10                then c[i, j] = c[i-1, j-1] + 1
11                     b[i, j] = ARROW_CORNER
12                else if c[i-1, j] >= c[i, j-1]
13                     then c[i, j]=c[i-1, j]
14                          b[i, j] = ARROW_UP
15                     else c[i, j]=c[i, j-1]
16                          b[i, j] = ARROW_LEFT
17 return c and b
```

PRINT-LCS(b, X, i, j) Execute PRINT LCS

```
1  if i=0 or j=0
2       then return
3  if b[i, j] == ARROW_CORNER
4       then PRINT-LCS(b, X, i-1, j-1)
5            print Xi
6  elseif b[i, j] == ARROW_UP
7       then PRINT-LCS(b, X, i-1, j)
```

http://lcs-demo.sourceforge.net

Our approach actually isn't so bad

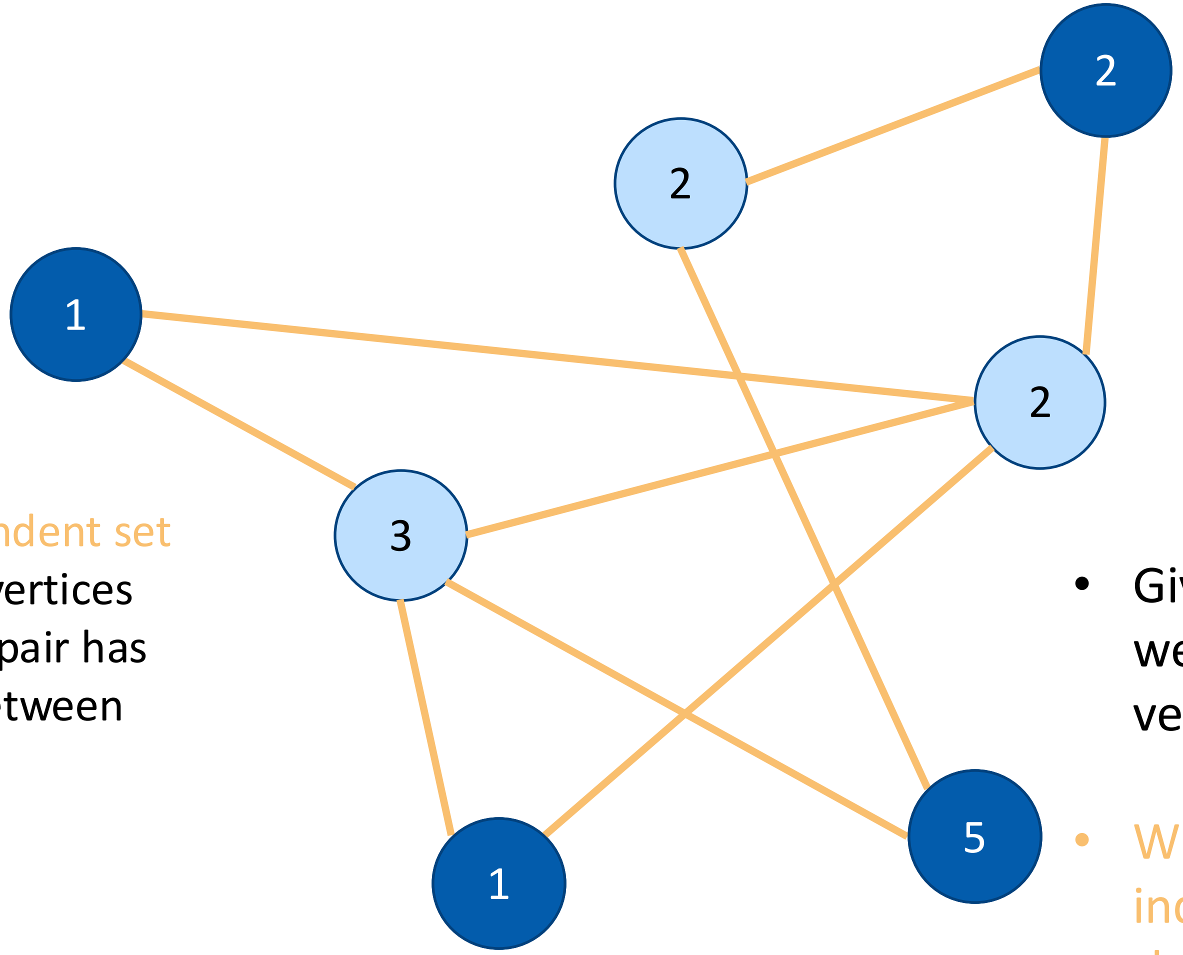
- If we are only interested in the length of the LCS we can do a bit better on space:
  - Since we go across the table one-row-at-a-time, we can only keep two rows if we want.
- If we want to recover the LCS, we need to keep the whole table.

- Can we do better than O(mn) time?
  - A bit better.
    - By a log factor or so.
  - Try to design it (as your lab work)!

What have we learned?

- We can find LCS(X,Y) in time O(nm)
  - if |Y|=n, |X|=m
- We went through the steps of coming up with a dynamic programming algorithm.
  - We kept a 2-dimensional table, breaking down the problem by decrementing the length of X and Y.

Independent Set

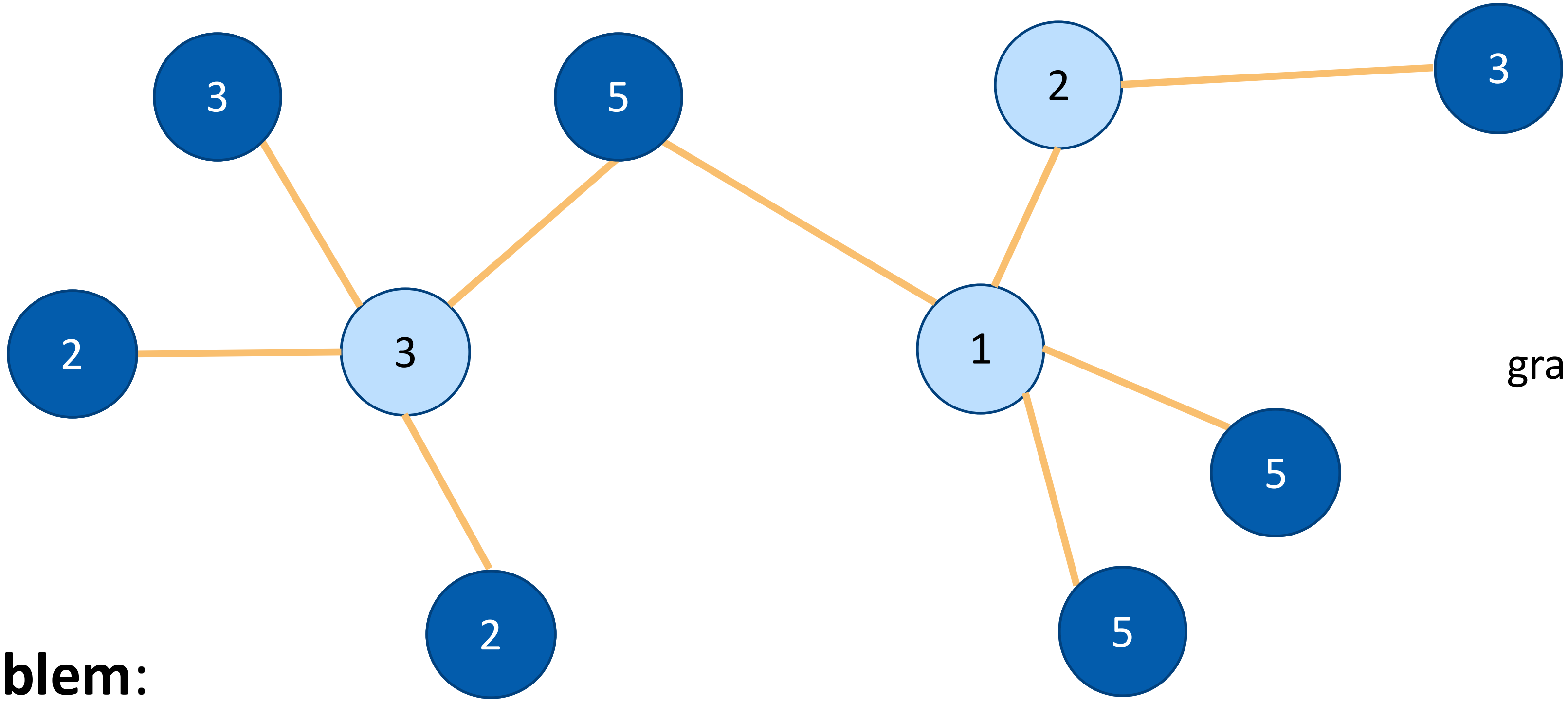


An independent set is a set of vertices so that no pair has an edge between them.

- Given a graph with weights on the vertices…
- What is the independent set with the largest weight?

Independent Set

- Actually, this problem is NP-complete.
  So, we are unlikely to find an efficient algorithm.
- But if we also assume that the graph is a tree…



A tree is a connected graph with no cycles.

Problem:

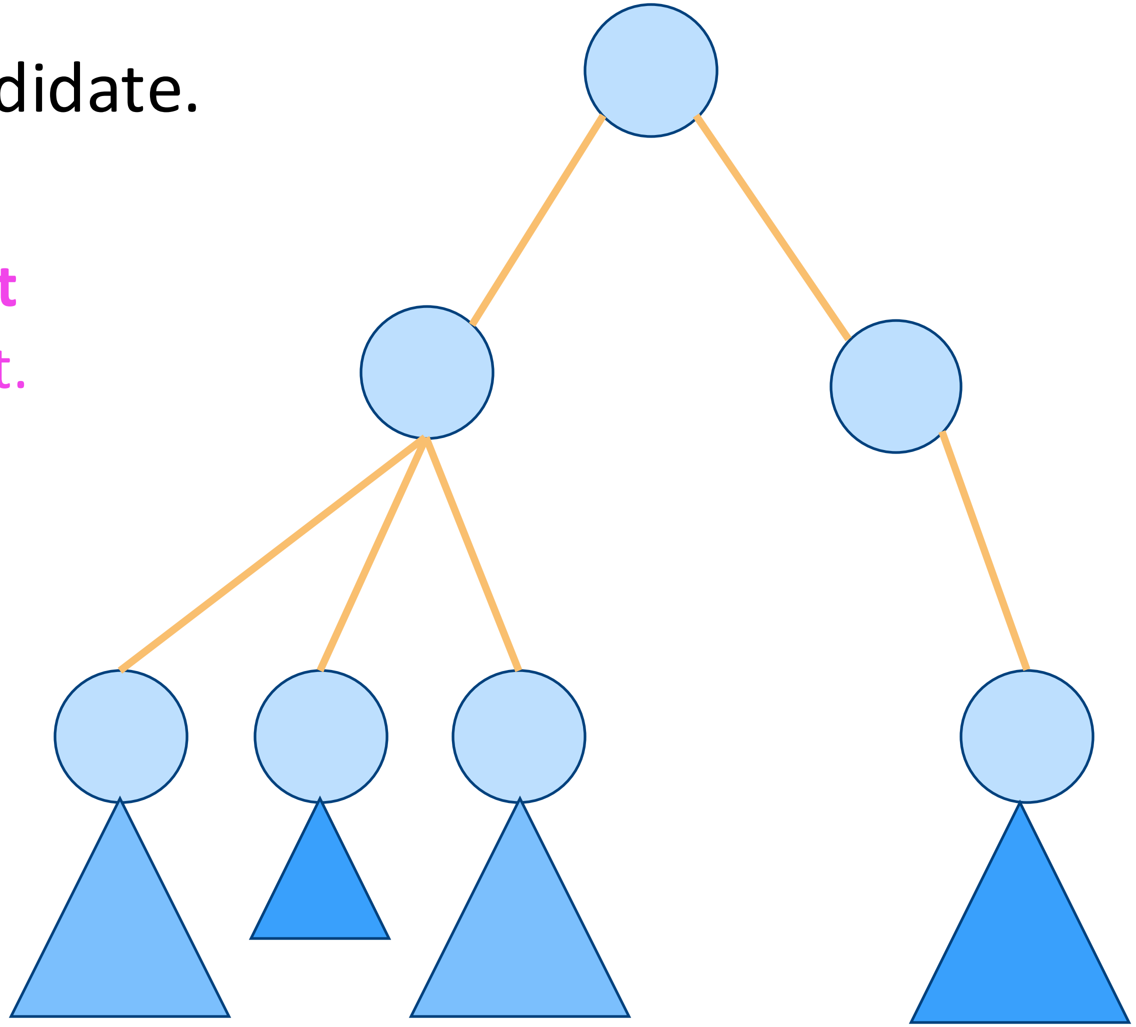
find a maximal independent set in a tree (with vertex weights).

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution
- **Step 3:** Use dynamic programming to find the value of the optimal solution
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

Optimal substructure

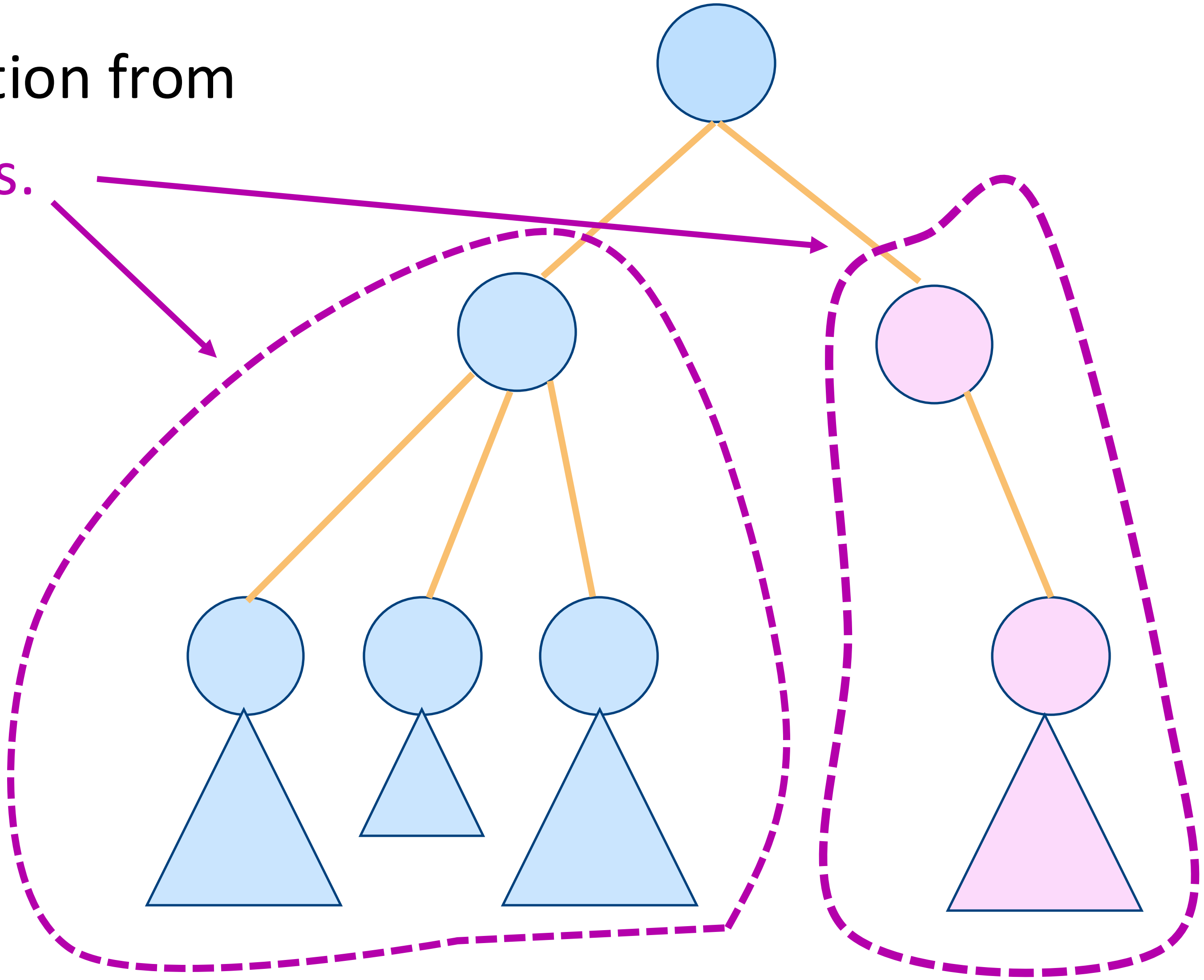
- **Subtrees** are a natural candidate.
- There are **two cases**:
  1. The root of this tree is **not** in a maximal independent set.
  2. Or it is



Case 1: the root is not in a maximal independent set

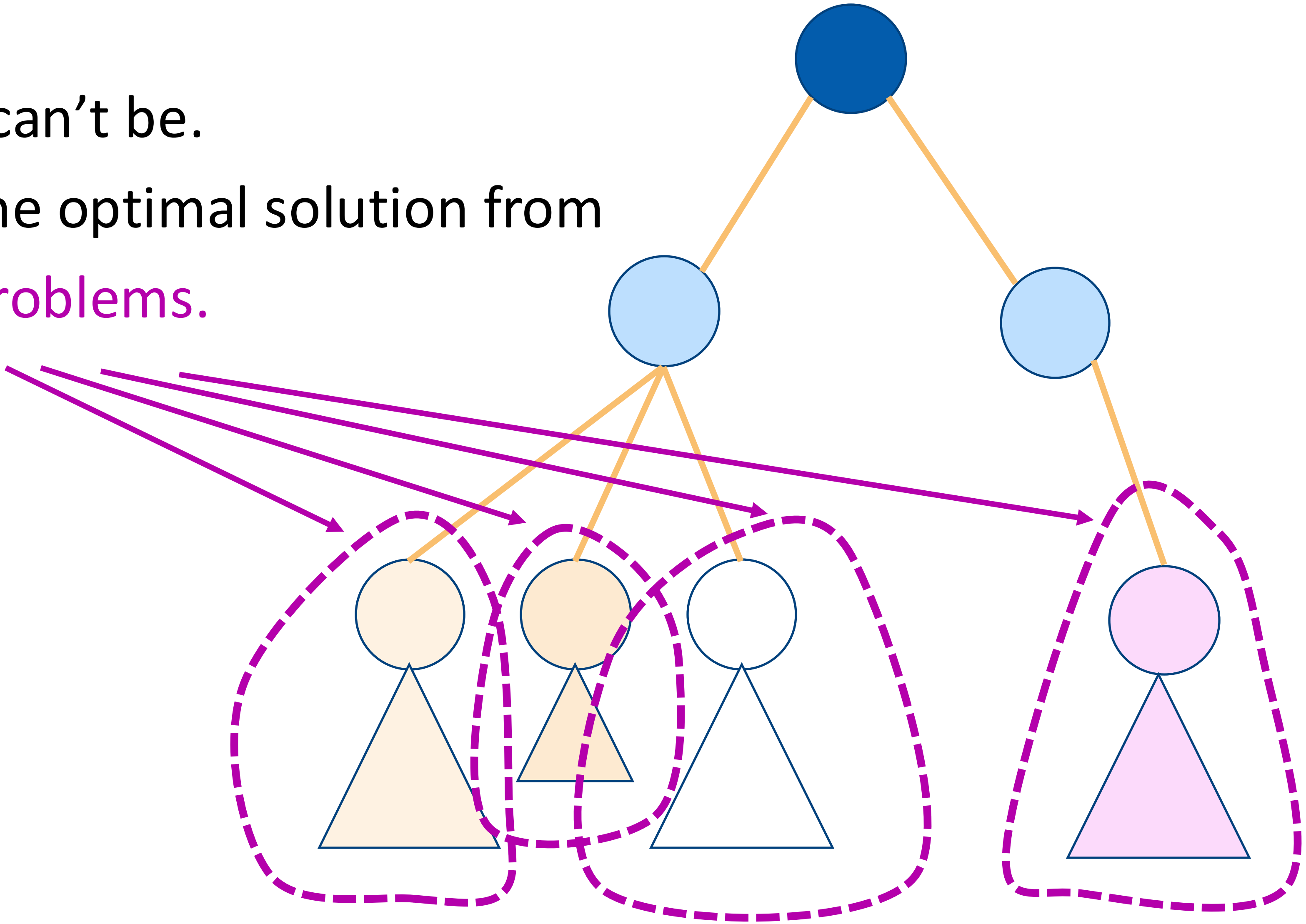
- Use the optimal solution from

these smaller problems.



Case 2 : the root is in an maximal independent set

- Then its children can't be.
- Below that, use the optimal solution from these smaller subproblems.

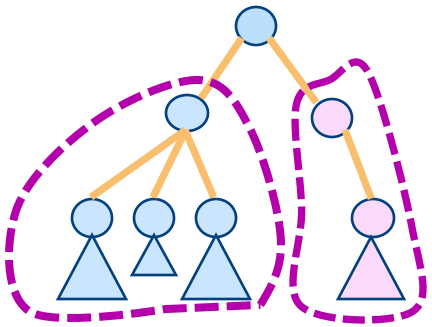


Recipe for applying Dynamic Programming

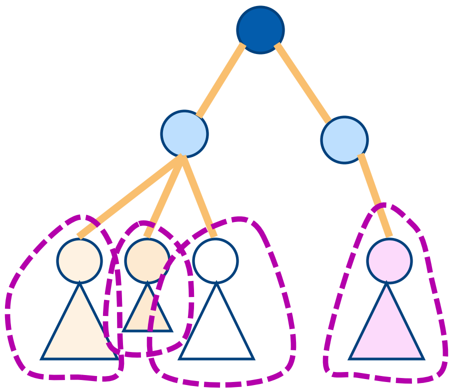
- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

Recursive formulation: try 1

- Let A[u] be the weight of a maximal independent set in the tree rooted at u.



- $A[u] = \max \begin{cases} \sum_{v \in u.\text{children}} A[v] \\ \text{weight}(u) + \sum_{v \in u.\text{grandchildren}} A[v] \end{cases}$

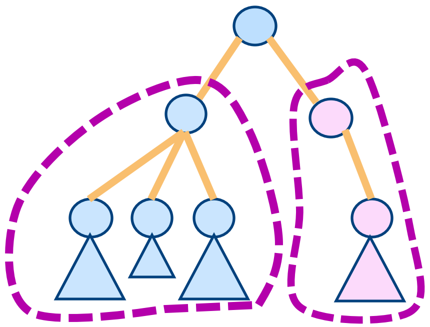


When we implement this, how do we keep track of this term?

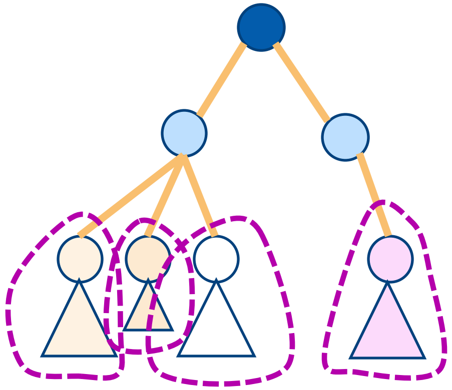
Recursive formulation: try 2

Keep two arrays!

- Let A[u] be the weight of a maximal independent set in the tree rooted at u.
- Let B[u] = $\sum_{v \in u.\text{children}} A[v]$



- $A[u] = \max \begin{cases} \sum_{v \in u.\text{children}} A[v] \\ \text{weight}(u) + \sum_{v \in u.\text{children}} B[v] \end{cases}$



Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

Dynamic Programming

香港科技大学(广州) THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- MIS_subtree(u):
  - **if** u is a leaf:
    - A[u] = weight(u)
    - B[u] = 0
  - **else**:
    - **for** v in u.children:
      - MIS_subtree(v)
    - $A[u] = \max\{ \sum_{v \in u.\text{children}} A[v], \text{weight}(u) + \sum_{v \in u.\text{children}} B[v] \}$
    - $B[u] = \sum_{v \in u.\text{children}} A[v]$

Initialize global arrays A, B that we will use in all of the recursive calls.

- MIS(T):
  - MIS_subtree(T.root)
  - **return** A[T.root]

Running time?

- We visit each vertex once, and for every vertex we do O(1) work:
  - Make a recursive call
  - Participate in summations of parent node
- Running time is O(|V|)

Recipe for applying Dynamic Programming

- Step 1: Identify optimal substructure.
- Step 2: Find a recursive formulation for the value of the optimal solution.
- Step 3: Use dynamic programming to find the value of the optimal solution.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- Step 5: If needed, code this up like a reasonable person.

You do this one!

What have we learned?

- We can find maximal independent sets in trees in time $O(|V|)$ using dynamic programming!

- For this example, it was natural to implement our DP algorithm in a top-down way.

Balanced Partition (BP) Problem

- We are given n integers $I = \{k_1, k_2, \ldots, k_n\}$, s.t. $0 \le k_i \le K$.
- We like to **partition** them into two sets S_1 and S_2 s.t. the difference d of the total sizes of the two sets is as small as possible

$$\min_{S_1, S_2} d \;\text{ s.t. } d = \left| \sum_{i \in S_1} k_i - \sum_{j \in S_2} k_j \right|.$$

$k_1 = 1, k_2 = 3, k_3 = 4, k_4 = 6, k_5 = 7$

1,3,6

4,7

$|S_1| = 10 \quad |S_2| = 11$

$d = |10 - 11| = 1$

Solution

max item size

$$|S_1| = \left\lfloor \frac{M}{2} \right\rfloor - m$$

- Let $M = \sum_i k_i \leq nK$.
- $m = 0$: The best we can hope for is $|S_1| = \left\lfloor \frac{M}{2} \right\rfloor - 0$ and $S_2 = M - S_1$.

Solution

max item size

$|S_1| = \left\lfloor \frac{M}{2} \right\rfloor - m$

- Let $M = \sum_i k_i \leq nK$.
- $m = 0$: The best we can hope for is $|S_1| = \left\lfloor \frac{M}{2} \right\rfloor - 0$ and $S_2 = M - S_1$.
- $m = 1$: If this is not possible, the next best is $|S_1| = \left\lfloor \frac{M}{2} \right\rfloor - 1$ and $S_2 = M - S_1$.

Solution

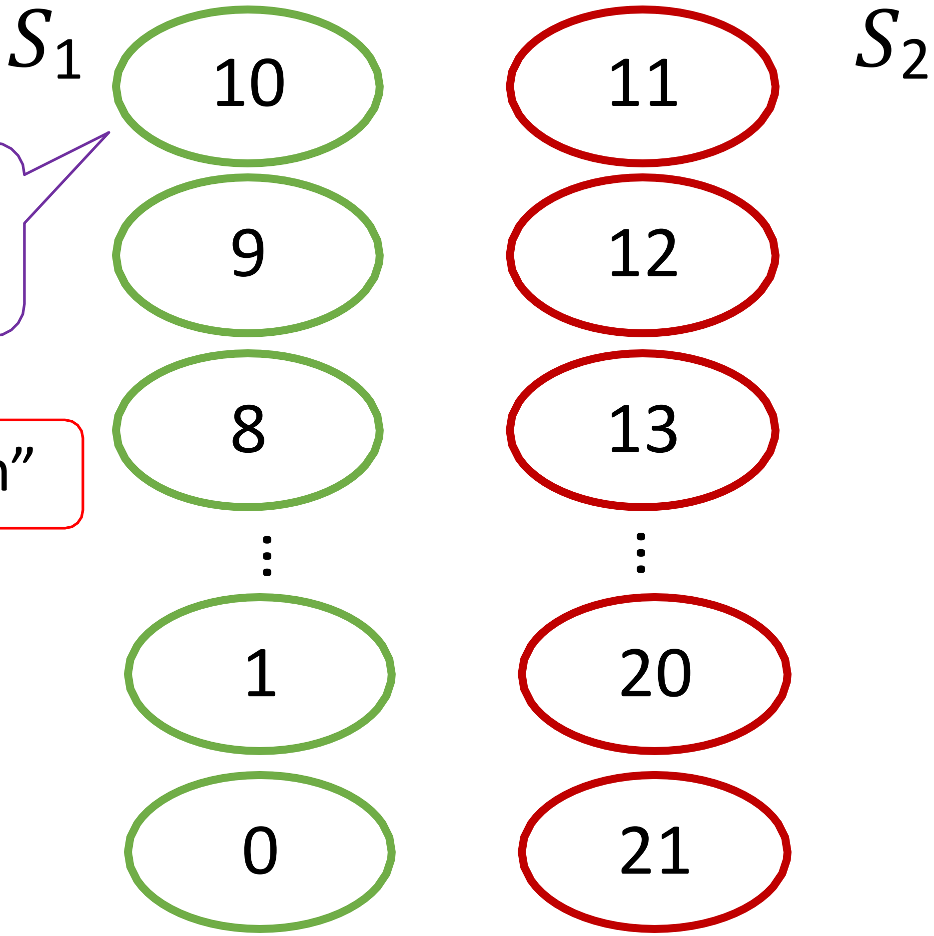
max item size

$|S_1| = \left\lfloor \frac{M}{2} \right\rfloor - m$

- Let $M = \sum_i k_i \leq nK$.
- $m = 0$: The best we can hope for is $|S_1| = \left\lfloor \frac{M}{2} \right\rfloor - 0$ and $S_2 = M - S_1$.
- $m = 1$: If this is not possible, the next best is $|S_1| = \left\lfloor \frac{M}{2} \right\rfloor - 1$ and $S_2 = M - S_1$.
- $m = 2$: If this is not possible, the next best is $|S_1| = \left\lfloor \frac{M}{2} \right\rfloor - 2$ and $S_2 = M - S_1$.
- $m = 3$: If this is not possible, the next best is $|S_1| = \left\lfloor \frac{M}{2} \right\rfloor - 3$ and $S_2 = M - S_1$.
- … try up to $m = \left\lfloor \frac{M}{2} \right\rfloor$. This is always possible since we have $S_1 = \emptyset, S_2 = I$.
- So, lets check the best we can achieve starting from $m = 0$.

Example

Possible allocations



Can I fill exactly a knapsack of size 10?

The "subsetum problem"

Given:

$k_1 = 1, k_2 = 3, k_3 = 4,$
$k_4 = 6, k_5 = 7$

$M = 21, \quad \left\lfloor \frac{M}{2} \right\rfloor = 10$

Reduction to the Subsetum Problem (SP)

- We reduced *BP* to the problem *SP*:

- $SP[n, D]$: We are given n integers $I = \{k_1, \ldots, k_n\}$, $s.t.\ 0 \le k_i \le K$, and an integer $D \le nK$. Is there a subset S of them such that $\sum_{i \in S} k_i = D$? (True/False).

Reduction to the Subsetum Problem (SP)

- Solution of *BP*:
- Solve *BP* by finding the smallest value of $m = 0, 1, \ldots, \left\lfloor \frac{M}{2} \right\rfloor$ for which $SP\left[n, \left\lfloor \frac{M}{2} \right\rfloor - m\right] = True.$
- Do we need to solve *SP* repeatedly (again and again form scratch) to solve *BP*?
- Can we reuse the solution of subproblems?

Solving SP

- Write the DP equations for *SP.*

- Very similar to knapsack problem.

- Can you guess them?

Solving SP

- Recursion for $SP[n, D]$:

given items $1..j$, is j used to fill X exactly?

$SP[j, X] = \max\{SP[j-1, X], SP[j-1, X-k_j]\}$, $0 \le j \le n, X \le D$,

$SP[j, 0] = 1, j = 0, \dots, n$, $SP[0, X > 0] = 0$, $SP[k, X < 0] = 0$.

- Solution: $SP[n, D]$.

- Topological sort: $j = 0,1,2 \dots, n$, $X = 0,1, \dots, D$.

- Complexity: ?? -> same as Knapsack = $O(nD)$.

Solving SP

- Recursion for $SP[n, D]$:

given items $1..j$, is j used to fill X exactly?

$SP[j, X] = \max\{SP[j-1, X], SP[j-1, X-k_j]\}$, $0 \le j \le n, X \le D$,

$SP[j, 0] = 1, j = 0, \ldots, n$, $SP[0, X > 0] = 0$, $SP[k, X < 0] = 0$.

Solution for *BP*:

M: sum of item sizes

- Solve $SP[n, \lfloor M/2 \rfloor]$, fill in table of sub-problems.
- Find largest $X = \lfloor M/2 \rfloor, \lfloor M/2 \rfloor - 1, \ldots,$ s.t. $SP[1..n, X] = 1$.

Exercise

- Solve *BP* for item sizes 1,2,3,4.

$M = 10, \lfloor M/2 \rfloor = 5$

$SP[j, X] = \max\{SP[j-1, X], SP[j-1, X-k_j]\}, 0 \leq j \leq n, X \leq D,$

$SP[j, 0] = 1, j = 0, \ldots, n, SP[0, X > 0] = 0, SP[k, X < 0] = 0.$

	X=0	1	2	3	4	5
j=0						
1						
2						
3						
4						

Exercise

- Solve *BP* for item sizes 1,2,3,4.

$M = 10$, $\lfloor M/2 \rfloor = 5$

$SP[j,X] = \max\{SP[j-1,X], SP[j-1,X-k_j]\}, 0 \le j \le n, X \le D,$
$SP[j,0] = 1, j = 0,\ldots,n, SP[0,X>0] = 0, SP[k,X<0] = 0.$

	X=0	1	2	3	4	5
j=0	1	0	0	0	0	0
1	1	1	0	0	0	0
2	1	1	1	1	0	0
3	1	1	1	1	1	1
4	1	1	1	1	1	1

Conclusions

- DP is a technique for solving complex optimization problems computationally.
- Key idea is to decompose a problem into a calculation involving the independent solution of similar type problems defined on reduced size systems (recurrence).
- The reduction of the complexity is due to memoization: solving each subproblem only once and remembering the results.

The End