CS2040S
Data Structures and Algorithms

Dynamic Programming…

Semester Roadmap

Where are we?

- Searching
- Sorting
- Lists
- Trees
- Hash Tables
- Graphs
- **Dynamic Programming**

You are here

Roadmap

Today and Tuesday: Dynamic Programming

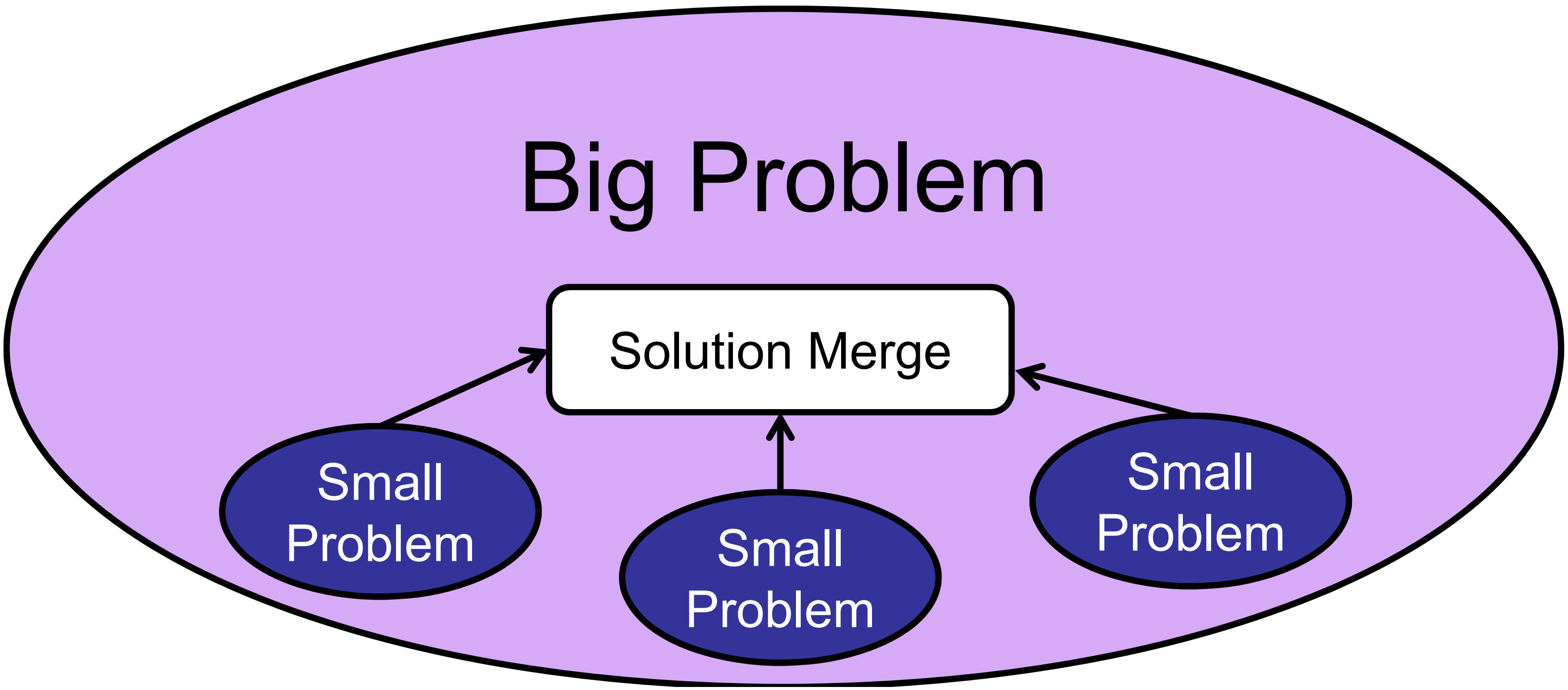
- Basics of DP
- Example: Longest Increasing Subsequence
- Example: Bounded Prize Collecting
- Example: Vertex Cover on a Tree
- Example: All-Pairs Shortest Paths

Dynamic Programming Basics

Dynamic Programming Basics

Optimal sub-structure:

- Optimal solution can be constructed from optimal solutions to smaller sub-problems.



Which of these problems exhibit optimal sub-structure? (Choose all that apply.)

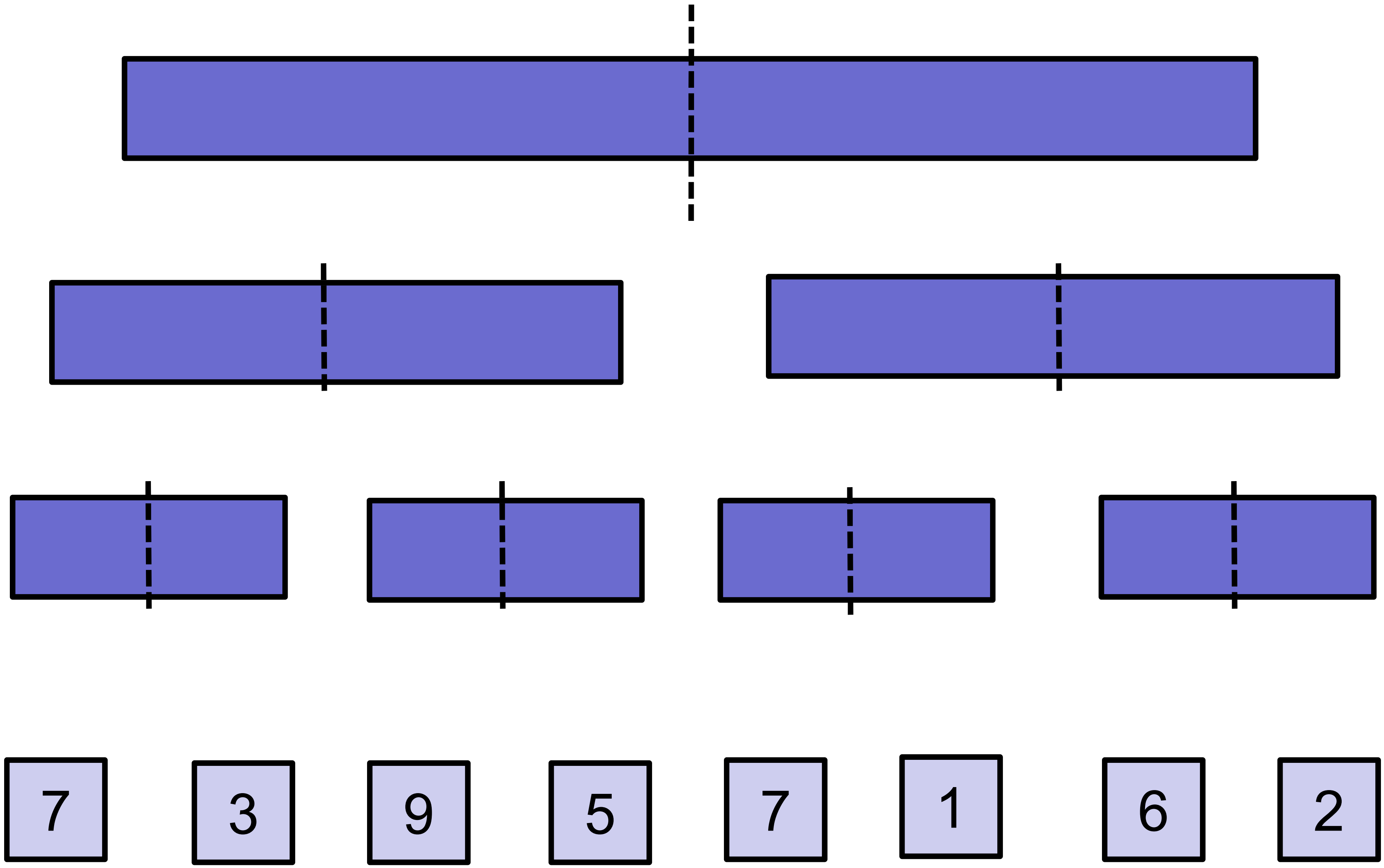
1. Sorting
2. Reversing a string
3. Merging two arrays
4. Shortest paths
5. Minimum spanning tree

Optimal Sub-structure

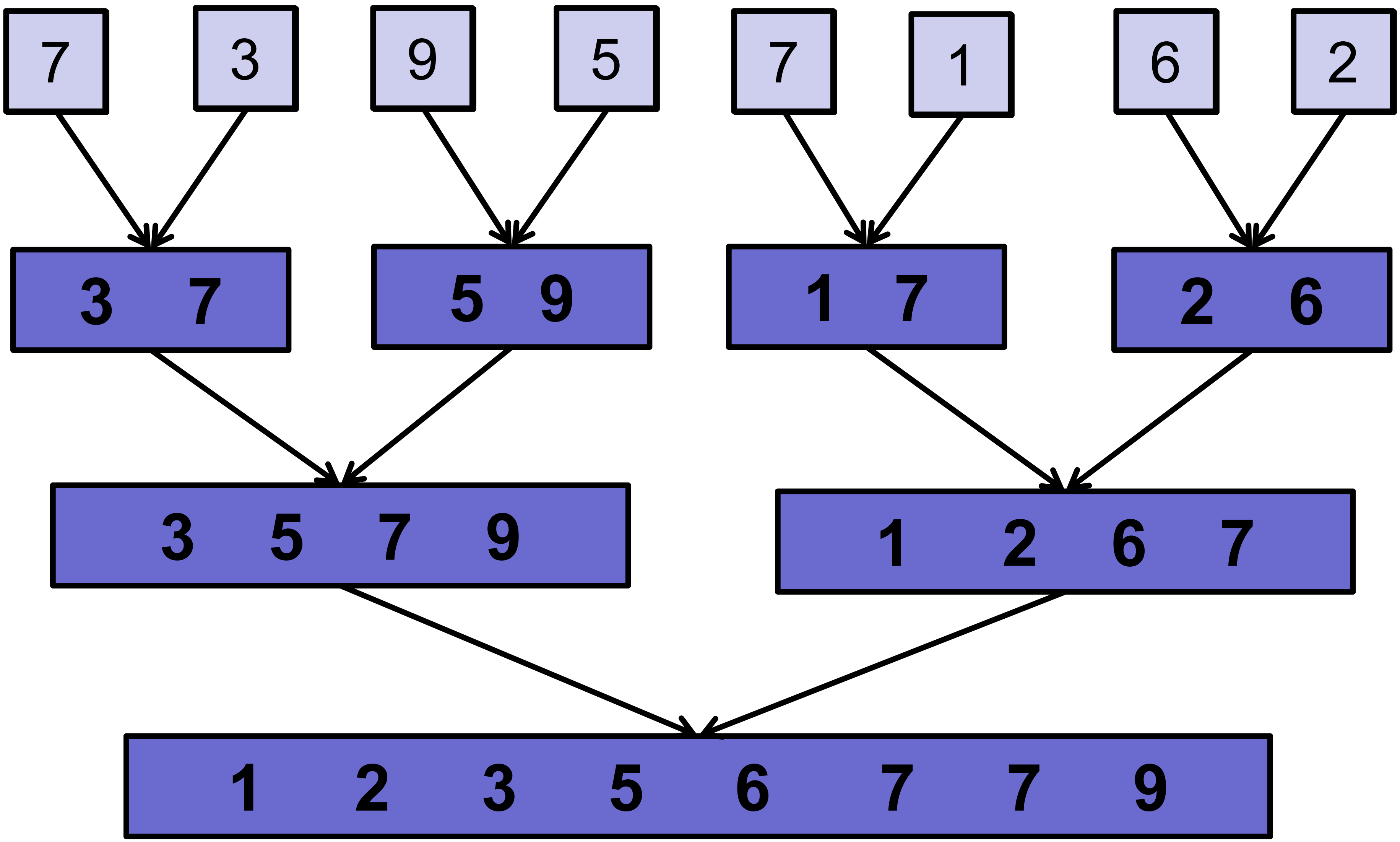
Property of (nearly) every problem we study:

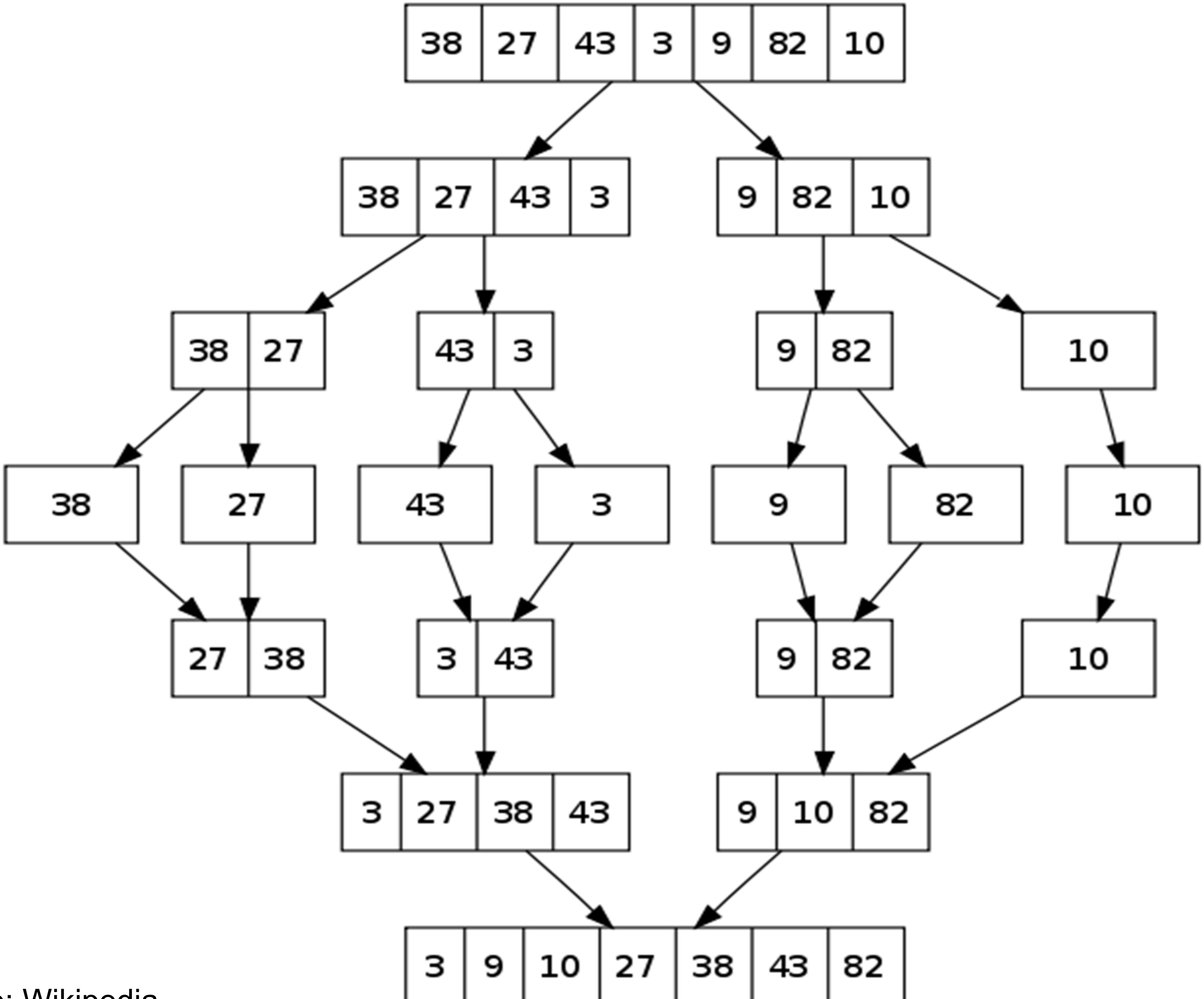
- Greedy algorithms
  - Dijkstra’s Algorithm
  - Minimum Spanning Tree algorithms
- Divide-and-conquer algorithms
  - MergeSort
  - Fast Fourier Transform

Divide-and-Conquer



Merging





Source: Wikipedia

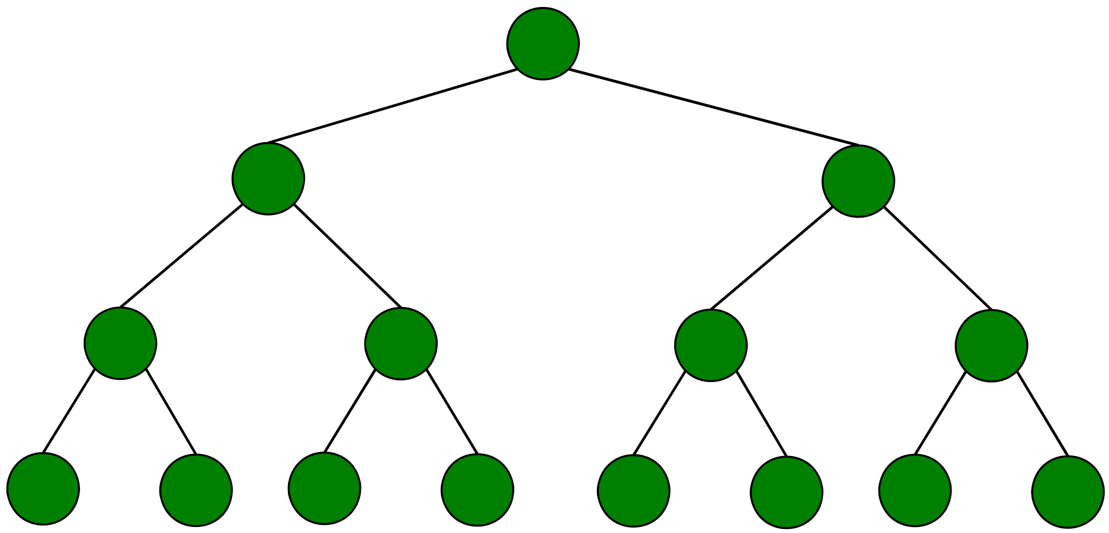
Optimal Sub-structure

Property of (nearly) every problem we study:

- Greedy algorithms
  - Dijkstra's Algorithm
  - Minimum Spanning Tree algorithms
- Divide-and-conquer algorithms
  - MergeSort
  - Fast Fourier Transform

Dynamic Programming

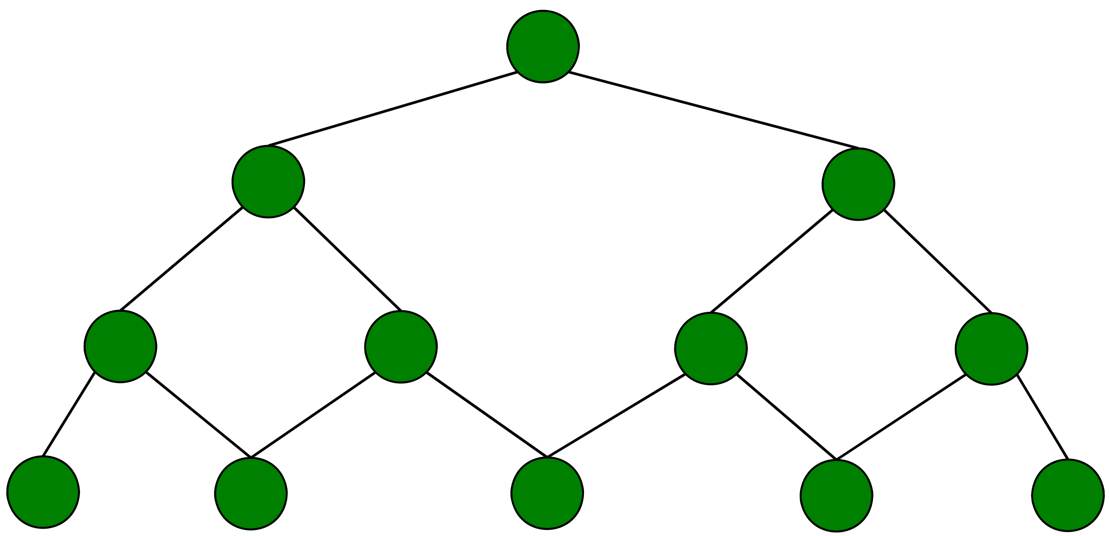
Optimal substructure:



Dynamic Programming

Overlapping sub-problems:

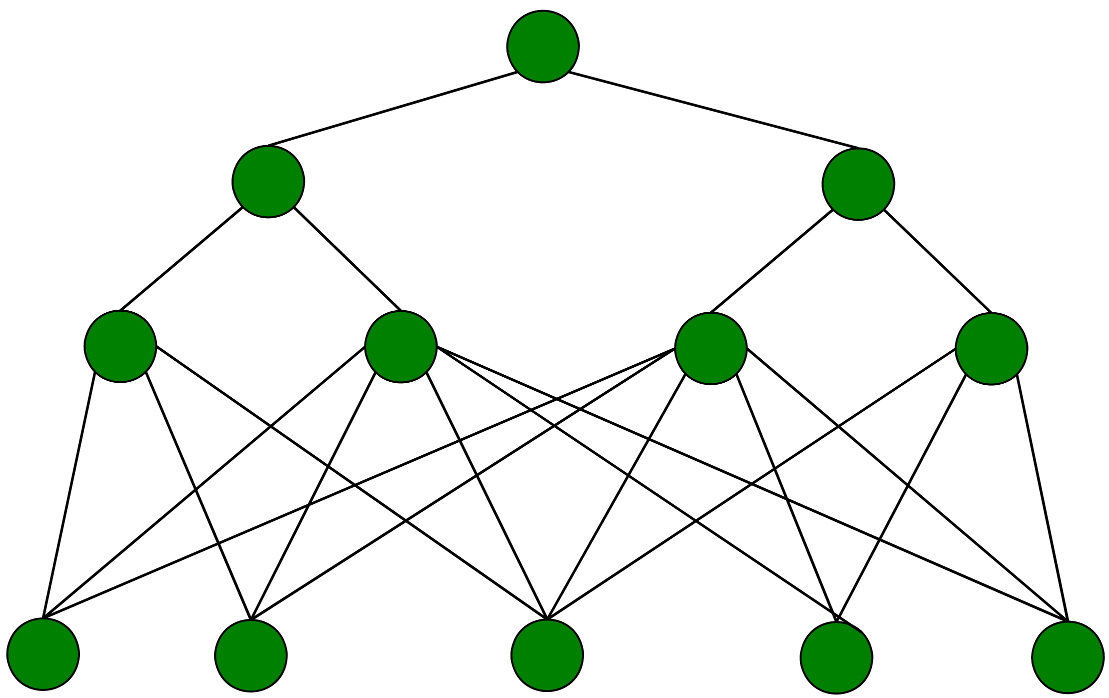
- The same smaller problem is used to solve multiple different bigger problems.



Dynamic Programming

Overlapping sub-problems:

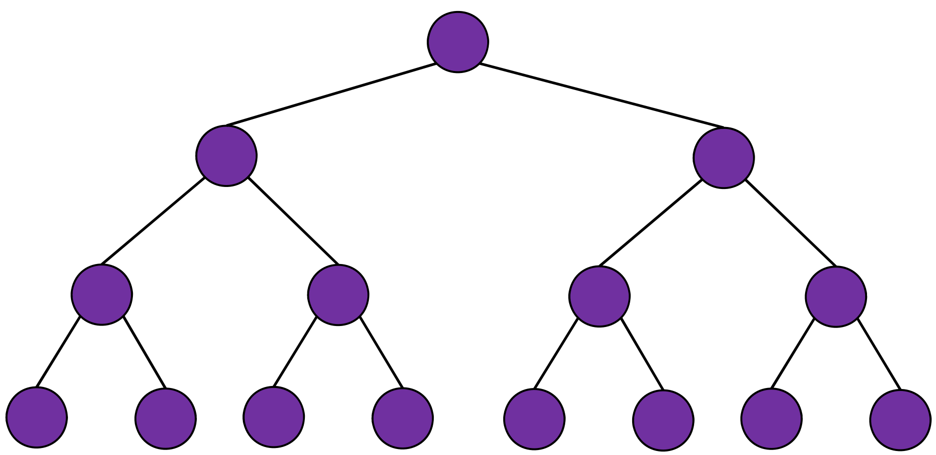
– The same smaller problem is used to solve multiple different bigger problems.



Dynamic Programming

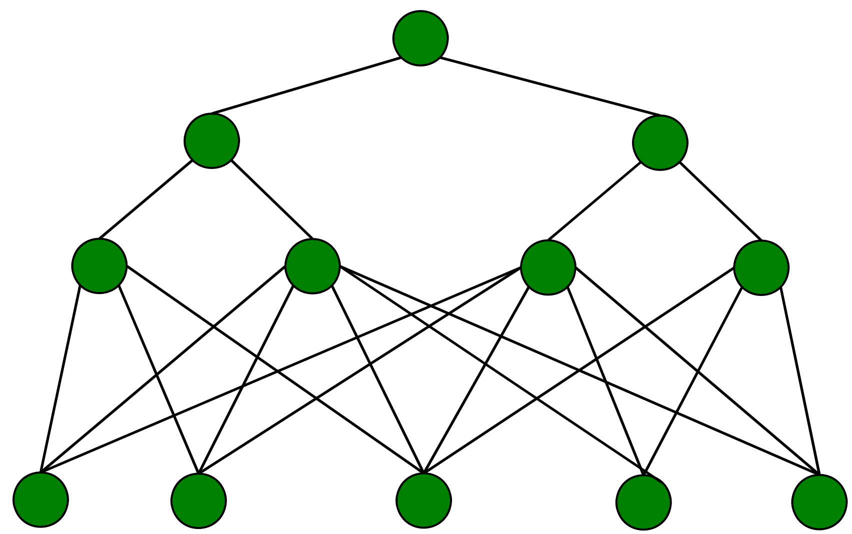
Contrast: Both have optimal substructure

No overlapping subproblems



Divide-and-Conquer

Overlapping subproblems



Dynamic Programming

Dynamic Programming

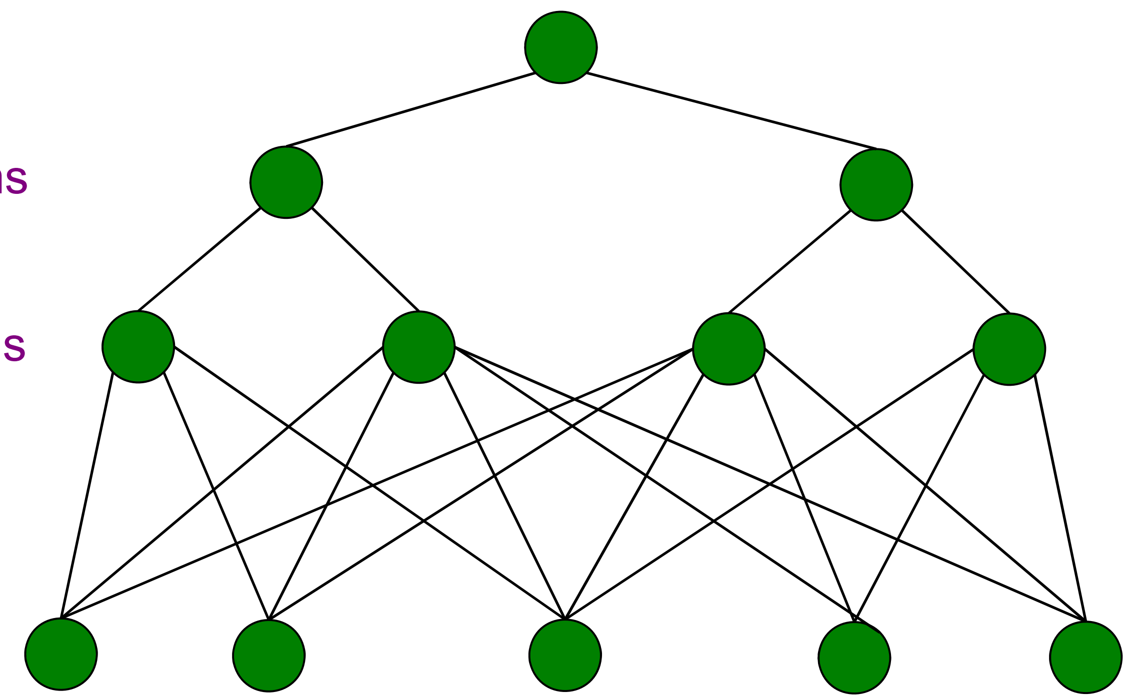
Basic strategy: *(bottom up dynamic programming)*

Step 4: solve root problem

Step 3: combine smaller problems

Step 2: combine smaller problems

Step 1: solve smallest problems

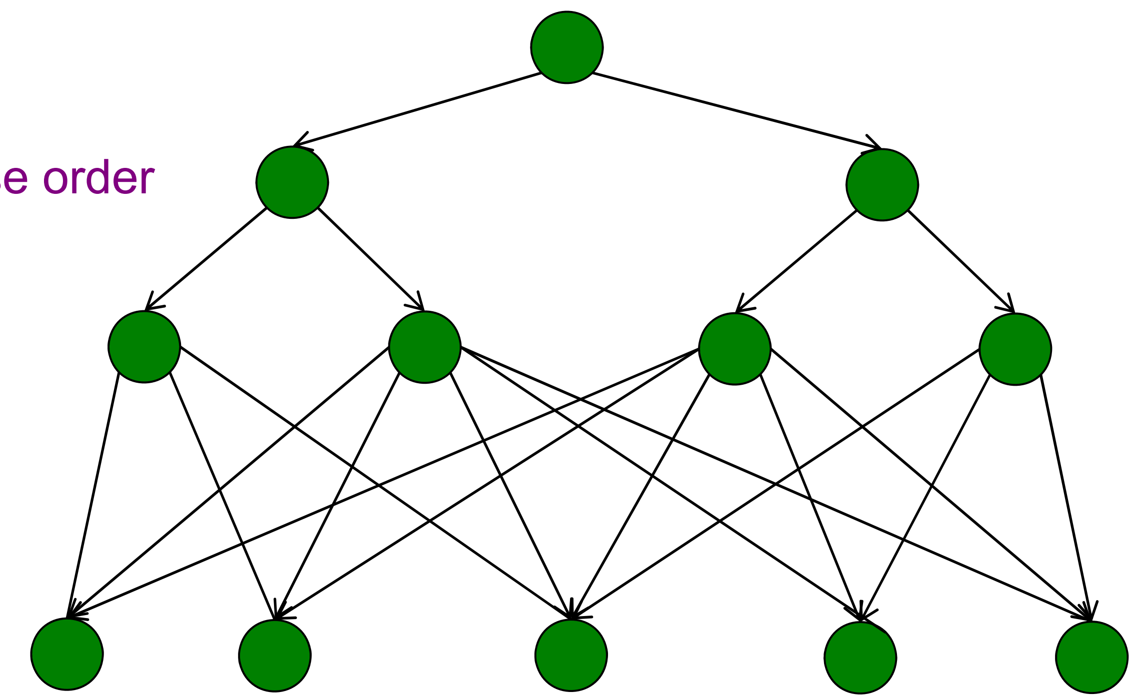


Dynamic Programming

Basic strategy: *(DAG + topological sort)*

Step 1: Topologically sort DAG

Step 2: Solve problems in reverse order



Dynamic Programming

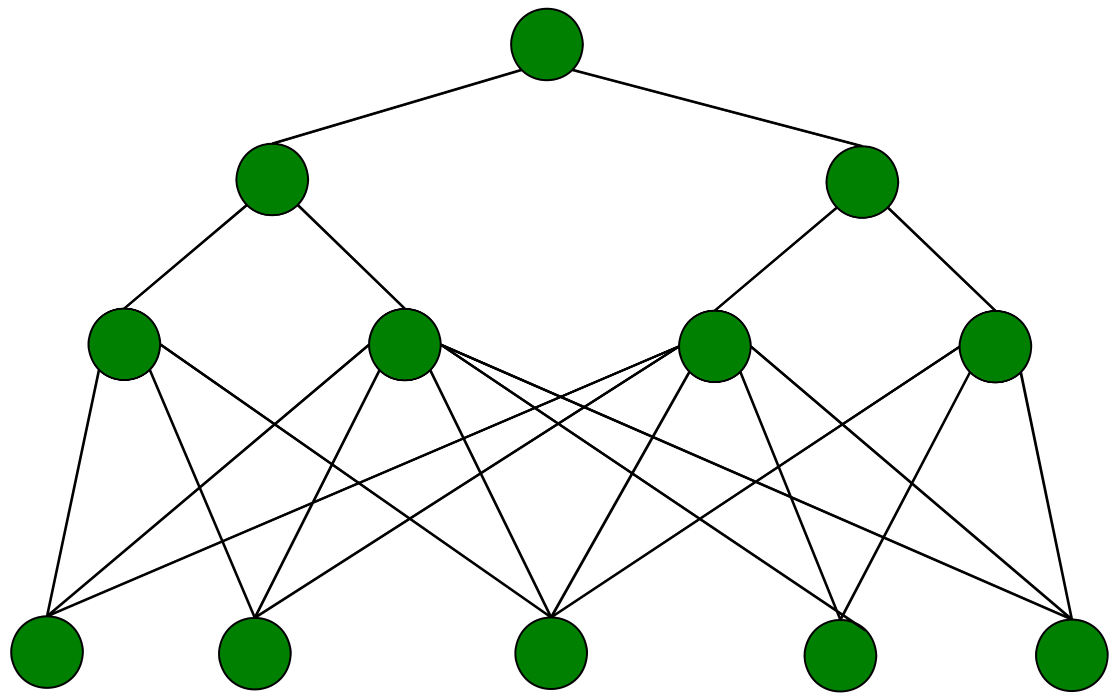
Basic strategy: *(top down dynamic programming)*

Step 1: Start at root and recurse.

Step 2: Recurse.

Step 3: Recurse.

Step 4: Solve and memoize.
Only compute each
solution once.



Dynamic Programming

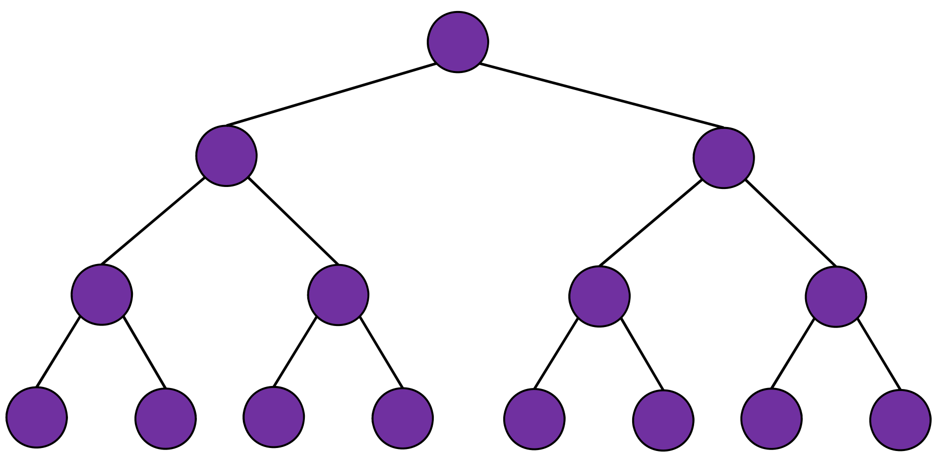
Table view:

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
1	17	22	14	19	8	4	9	12	15	7	5	9	13	14	18	4
2	15	12	13	13	7											
3																
4																
5																
6																
7																
8																
9																
10																
11																

Dynamic Programming

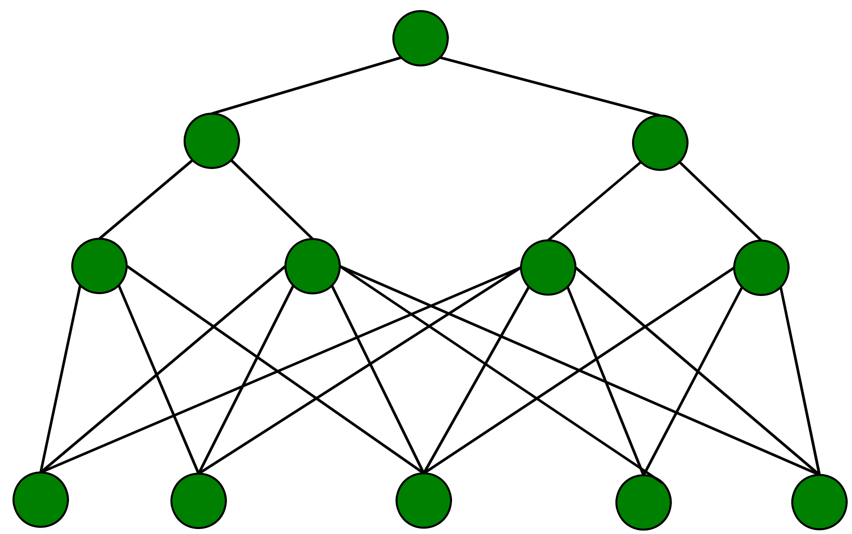
Contrast: Both have optimal substructure

No overlapping subproblems



Divide-and-Conquer

Overlapping subproblems



Dynamic Programming

Roadmap

Today: Dynamic Programming

- Basics of DP
- Example: Longest Increasing Subsequence
- Example: Bounded Prize Collecting
- Example: Vertex Cover on a Tree
- Example: All-Pairs Shortest Paths

CS2040S
Data Structures and Algorithms

Dynamic Programming…
(Three Examples)

Roadmap

Today: Dynamic Programming

- Basics of DP
- **Example: Longest Increasing Subsequence**
- **Example: Bounded Prize Collecting**
- **Example: Vertex Cover on a Tree**
- Example: All-Pairs Shortest Paths

Longest Increasing Subsequence

Input: Sequence of integers

- Example: {8, 3, 6, 4, 5, 7, 7}

Output: Increasing subsequence

- Example: {8, 3, 6, 4, 5, 7, 7}

Goal: Output sequence of maximum length

- Example: {8, 3, 6, 4, 5, 7, 7}

Longest Increasing Subsequence

Input: Sequence of integers

- Example: {8, 3, 6, 4, 5, 7, 7}

Output: <u>Length</u> of increasing subsequence

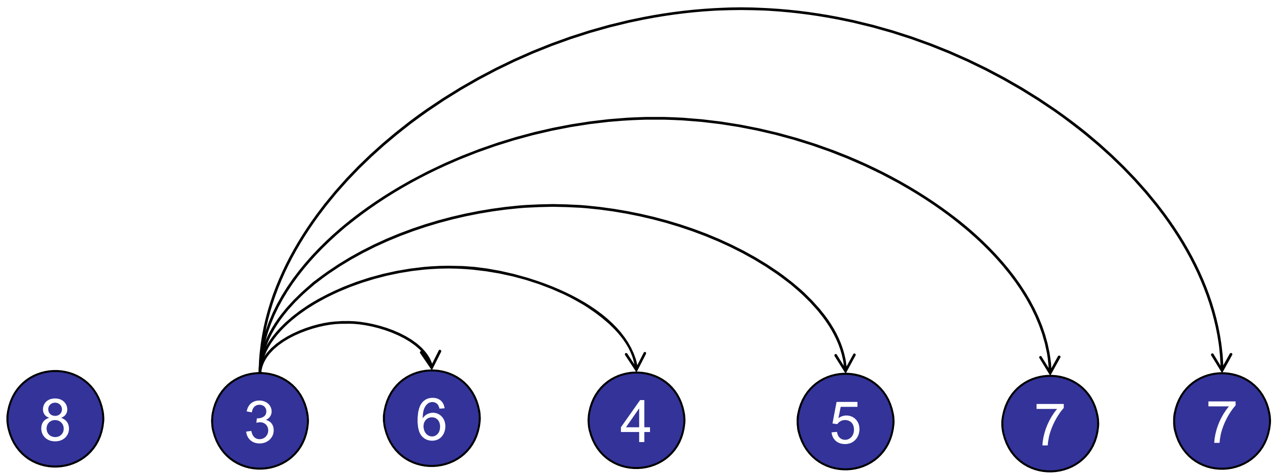
- Example: 3 → {8, 3, 6, 4, 5, 7, 7}

Goal: Output maximum length

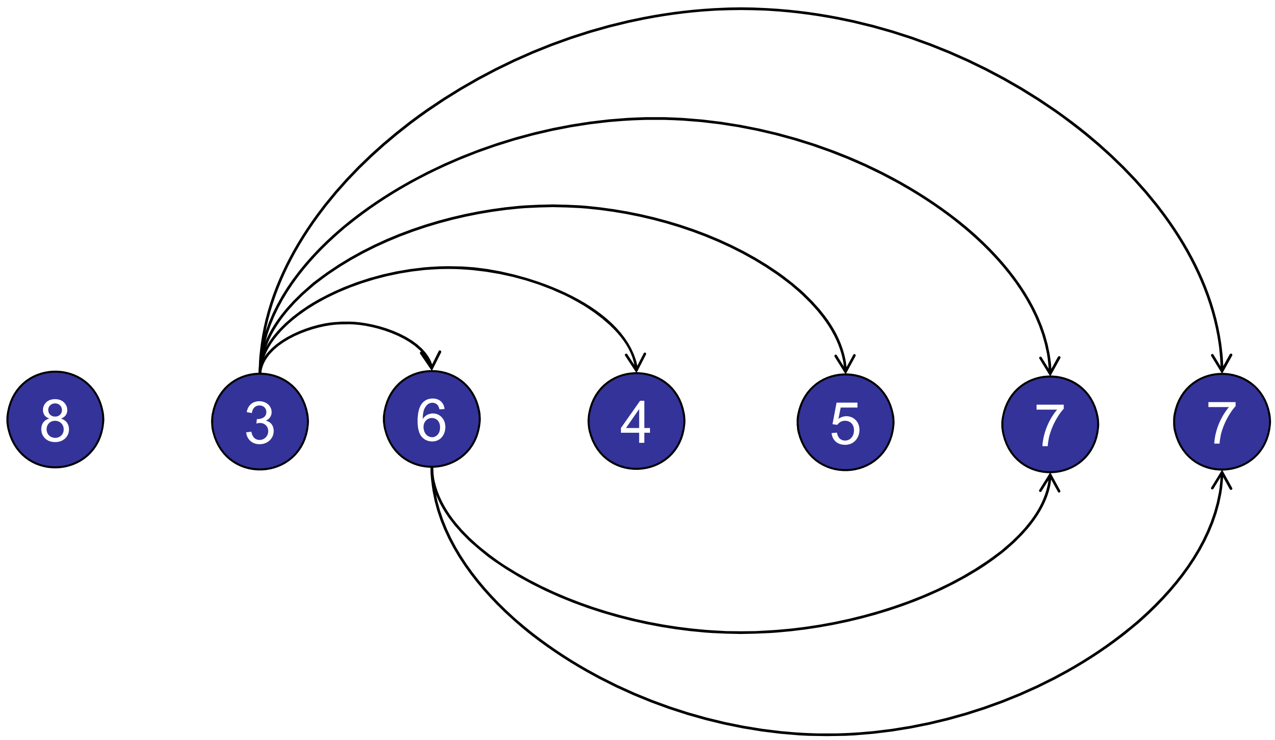
- Example: 4 →{8, 3, 6, 4, 5, 7, 7}

DAG Solution

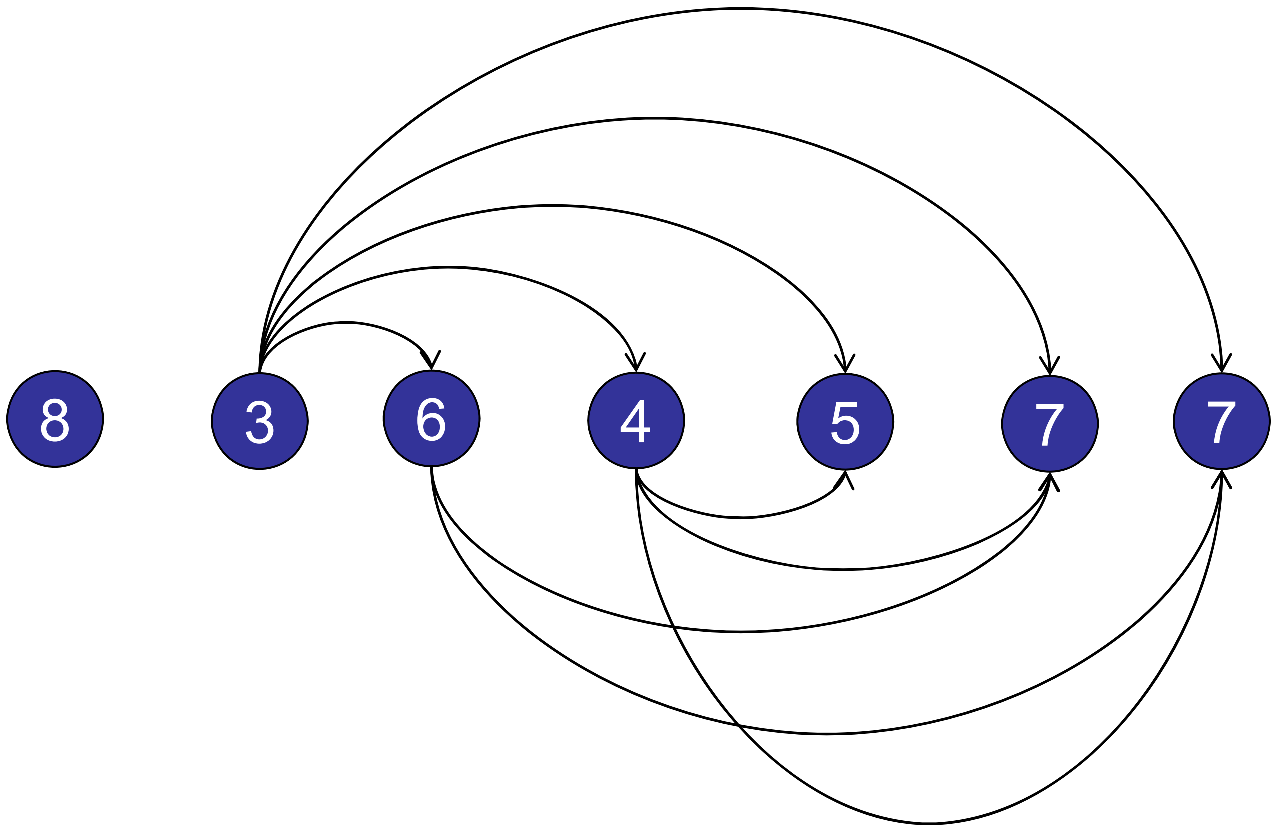
DAG Solution



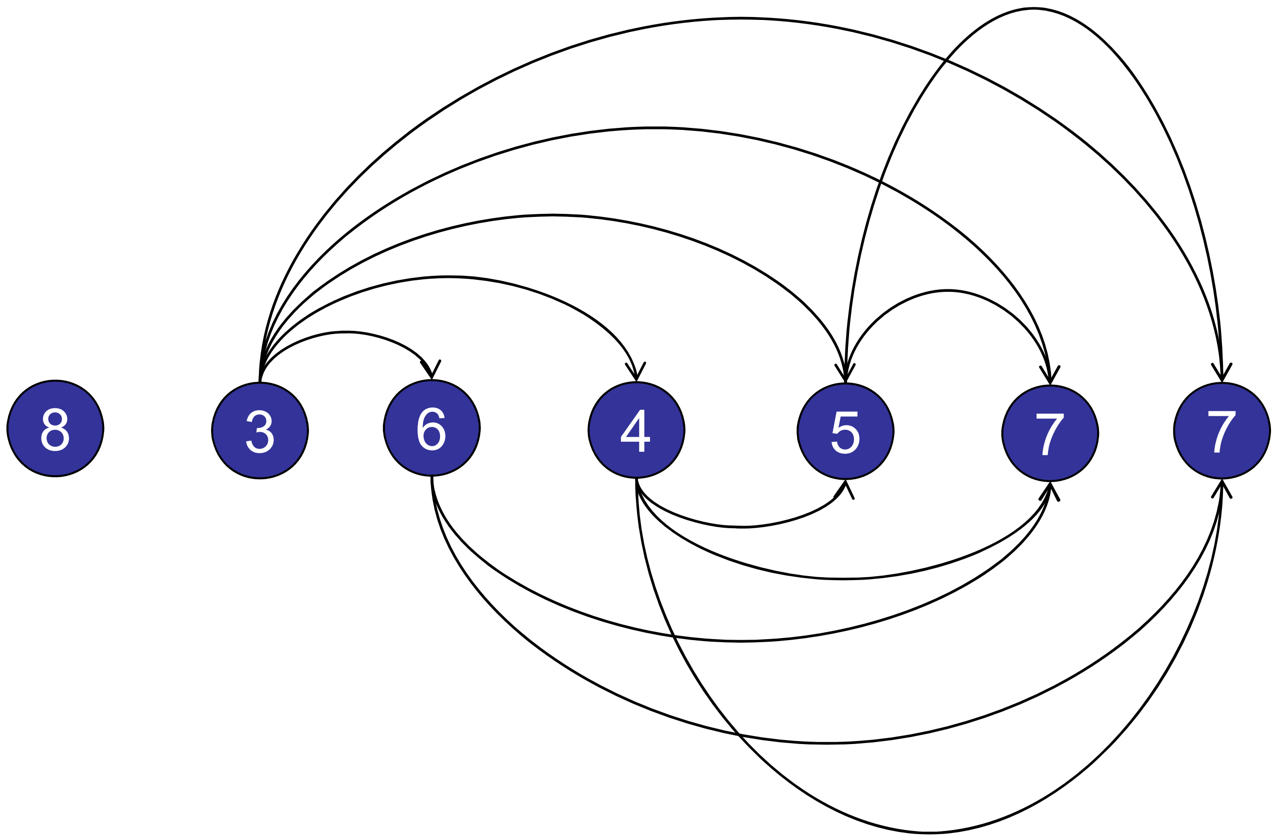
DAG Solution



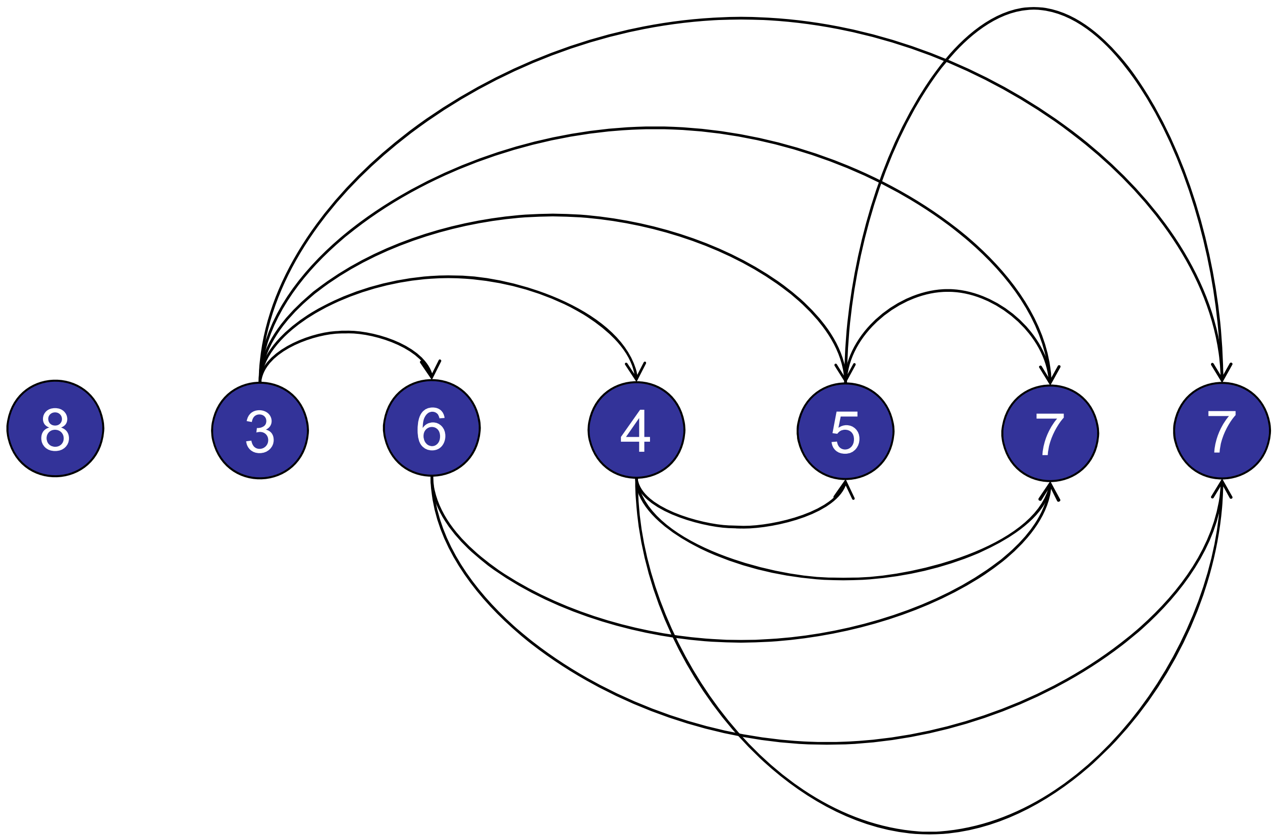
DAG Solution



DAG Solution

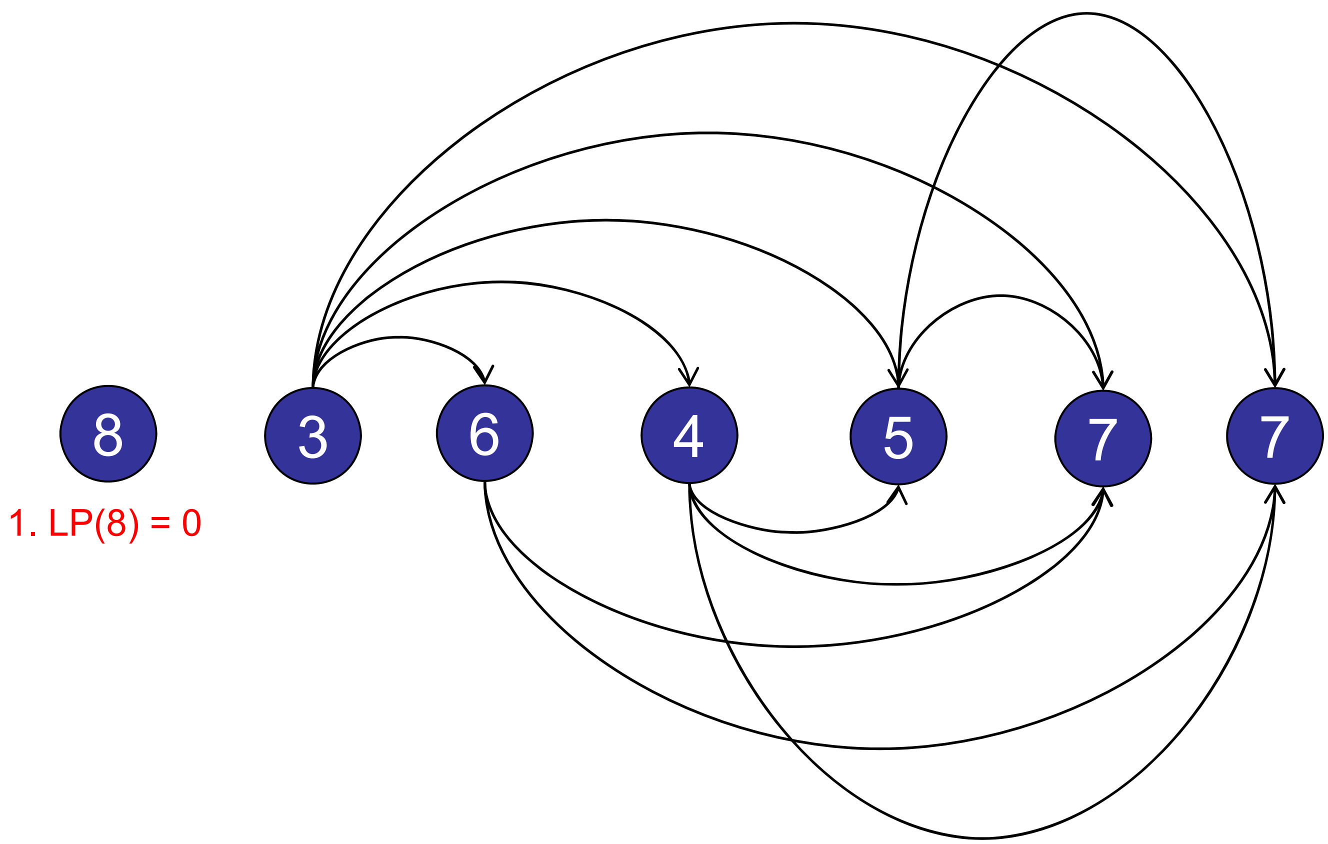


DAG Solution



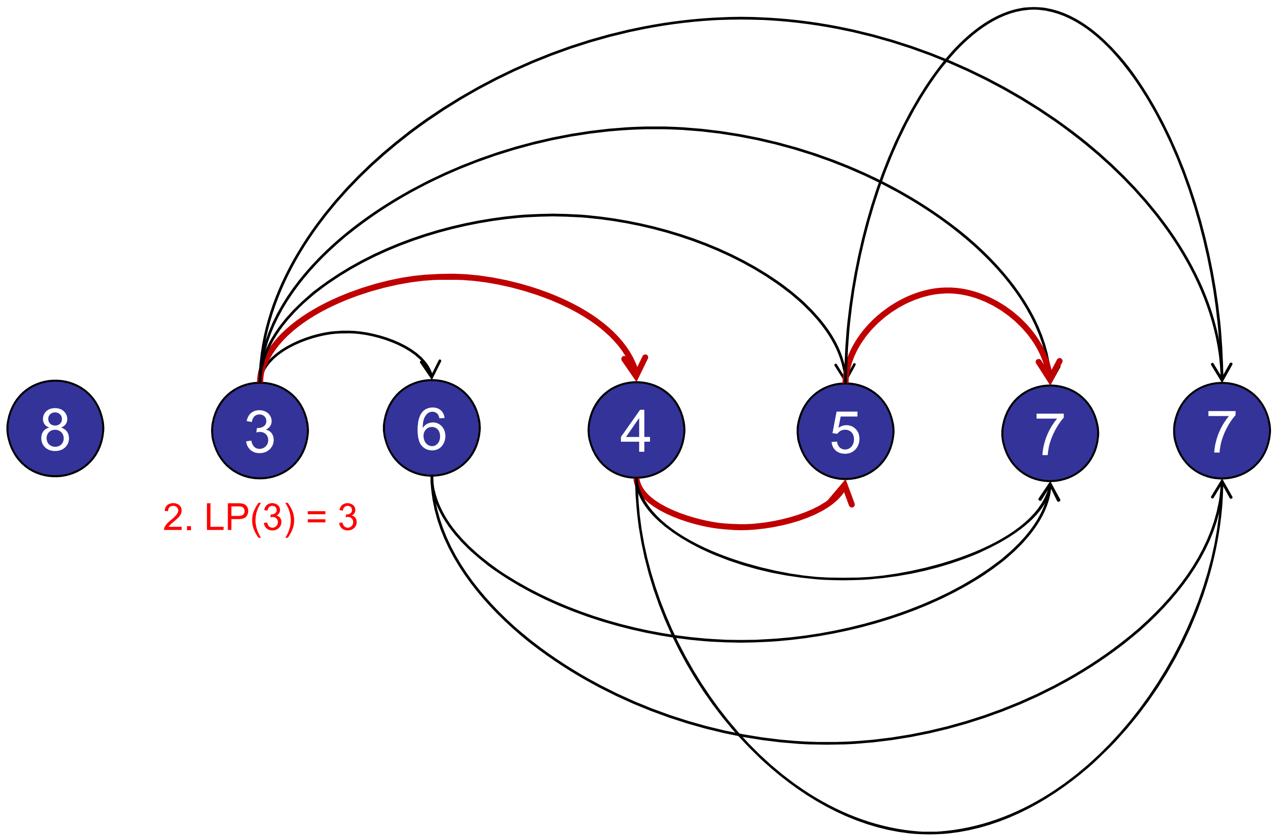
Step 1: Topological sort. (Oops, nothing to do.)

DAG Solution



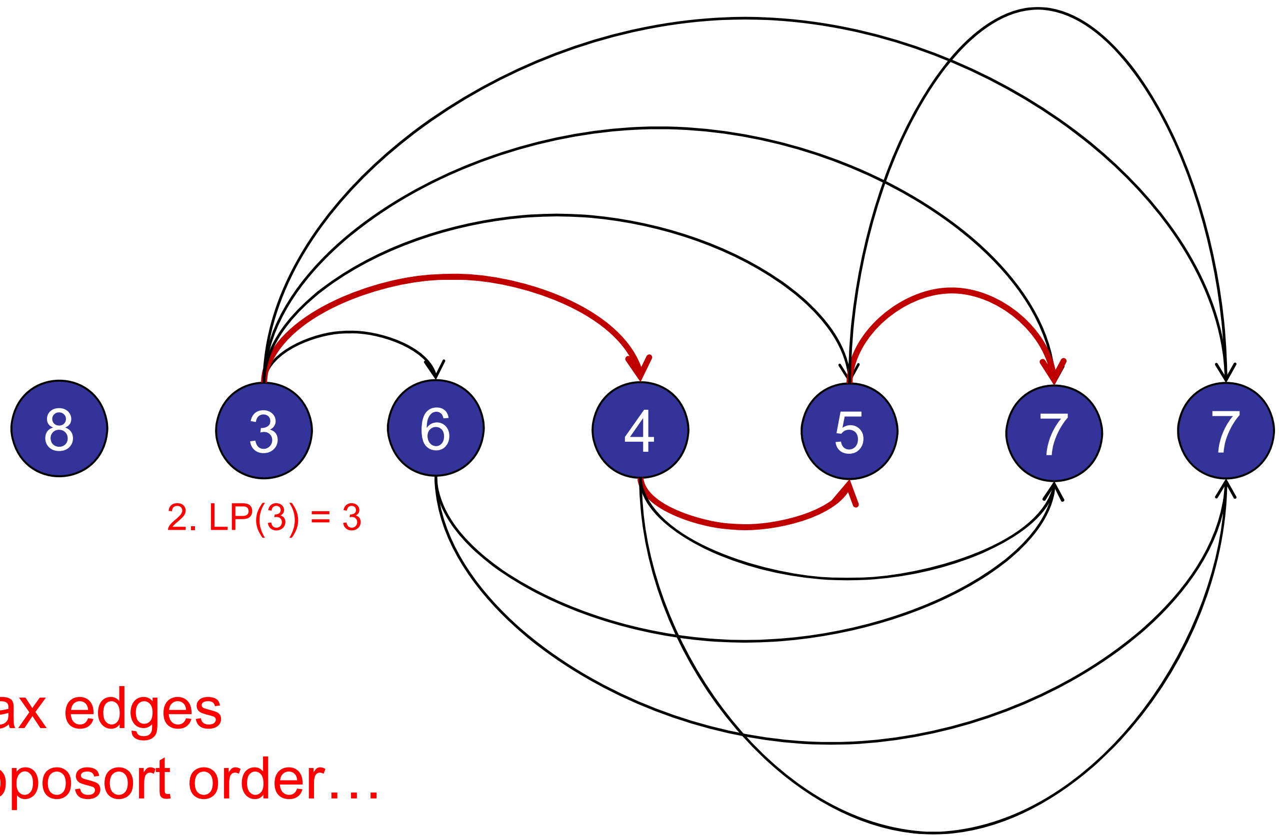
Step 2: Calculate longest paths.

DAG Solution



Step 2: Calculate longest paths.

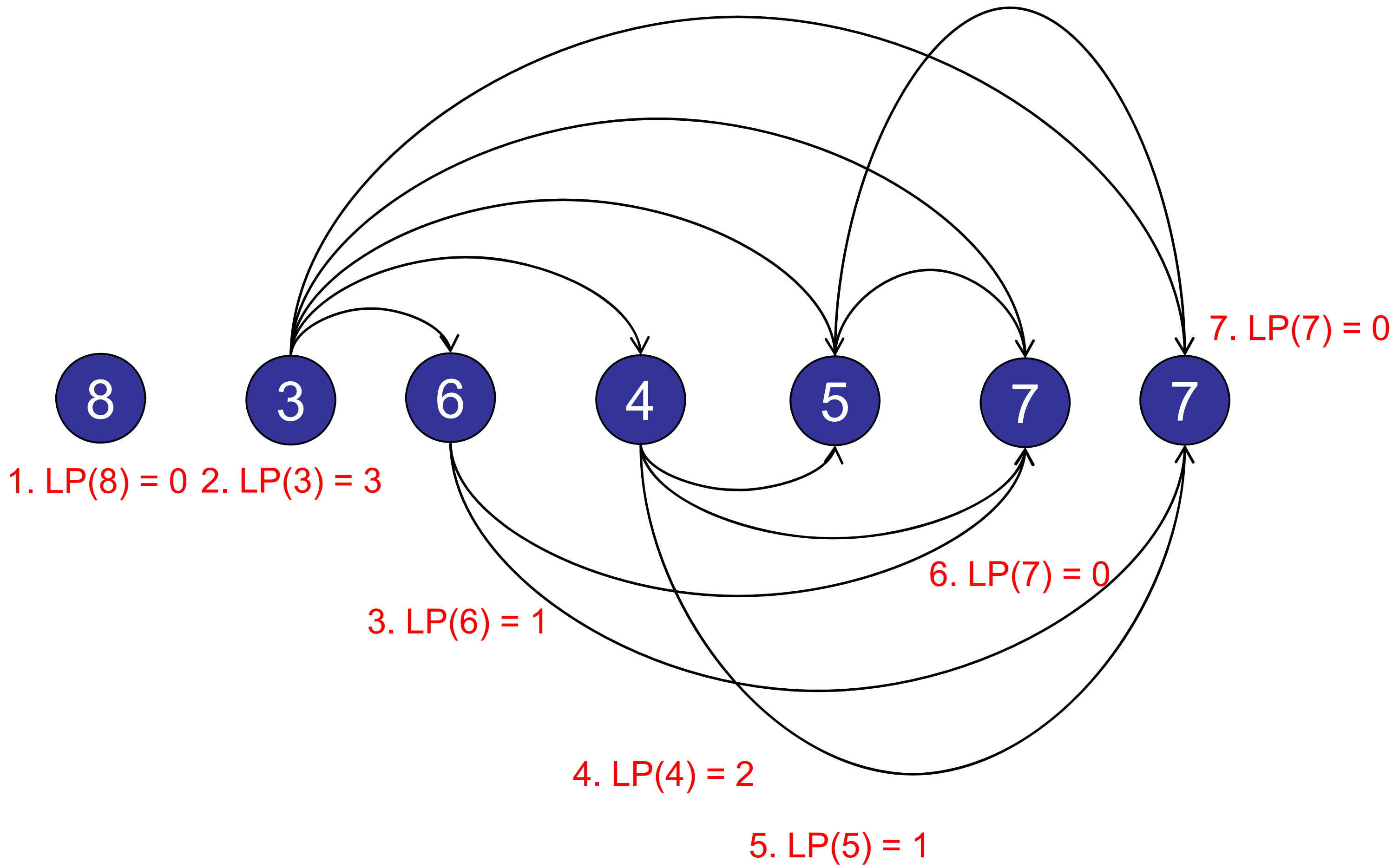
DAG Solution



Relax edges
in toposort order…

Step 2: Calculate longest paths.

DAG Solution

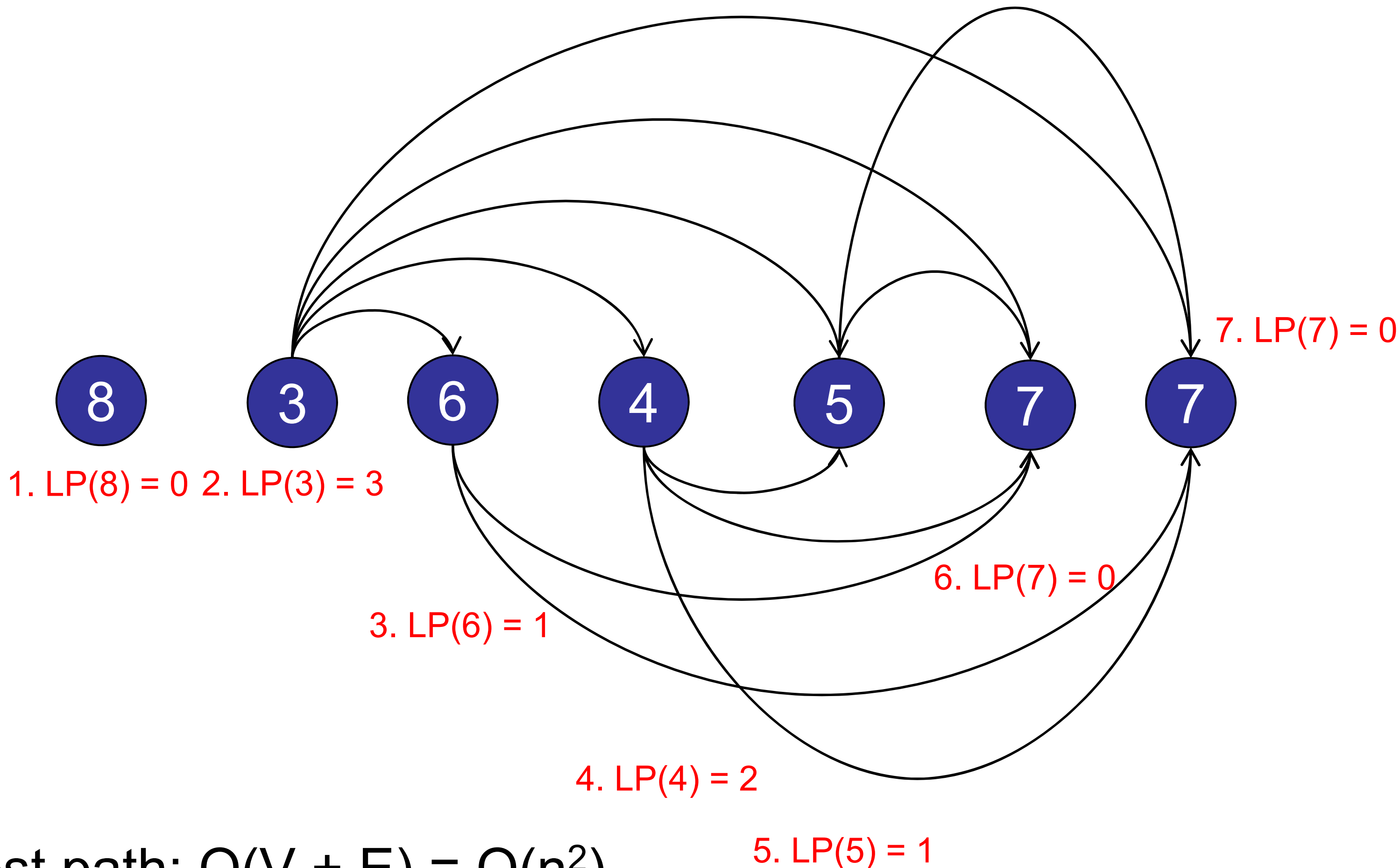


Step 2: Calculate longest paths. LIS = max(LP)+1

What is the running time of the DAG alg for a sequence of n numbers?

1. $O(n)$
2. $O(n \log n)$
3. $O(n^2)$
4. $O(n^2 \log n)$
5. $O(n^3)$
6. None of the above.

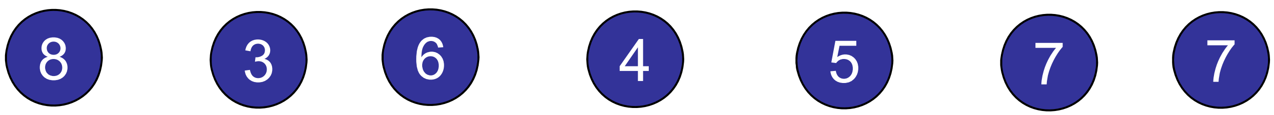
DAG Solution



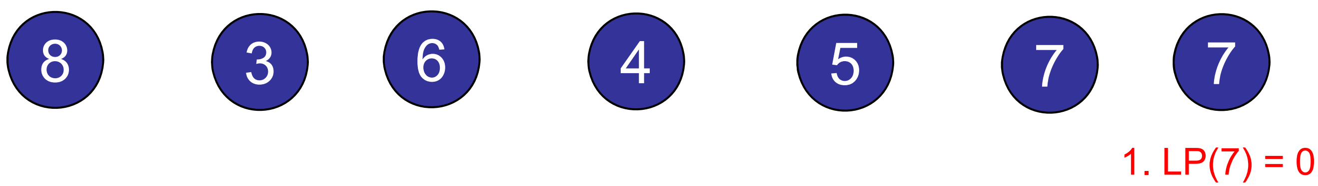
Longest path: $O(V + E) = O(n^2)$

Run longest path n times = $O(n^3)$

Overlapping Subproblems

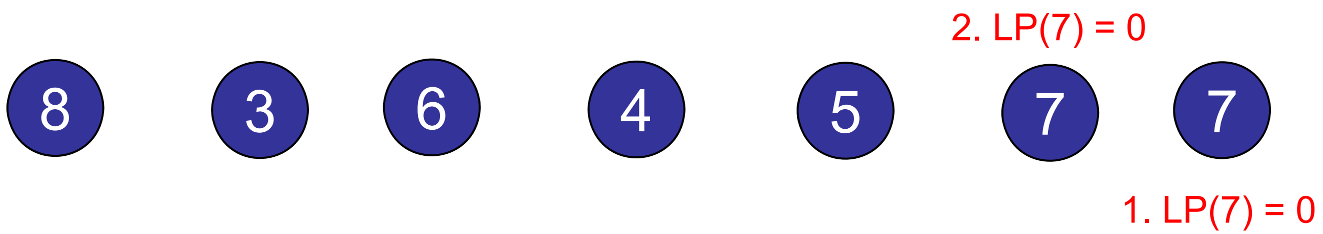


Overlapping Subproblems



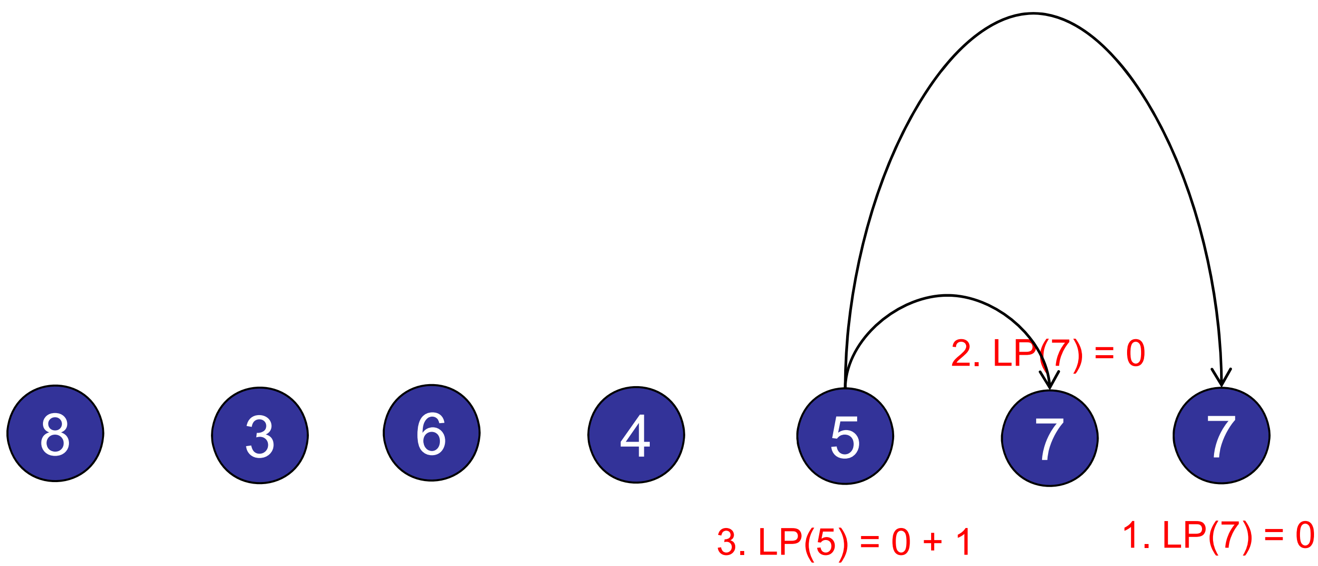
Start with the smallest sub-problem: LP(7)

Overlapping Subproblems



Start with the smallest sub-problem: LP(7)

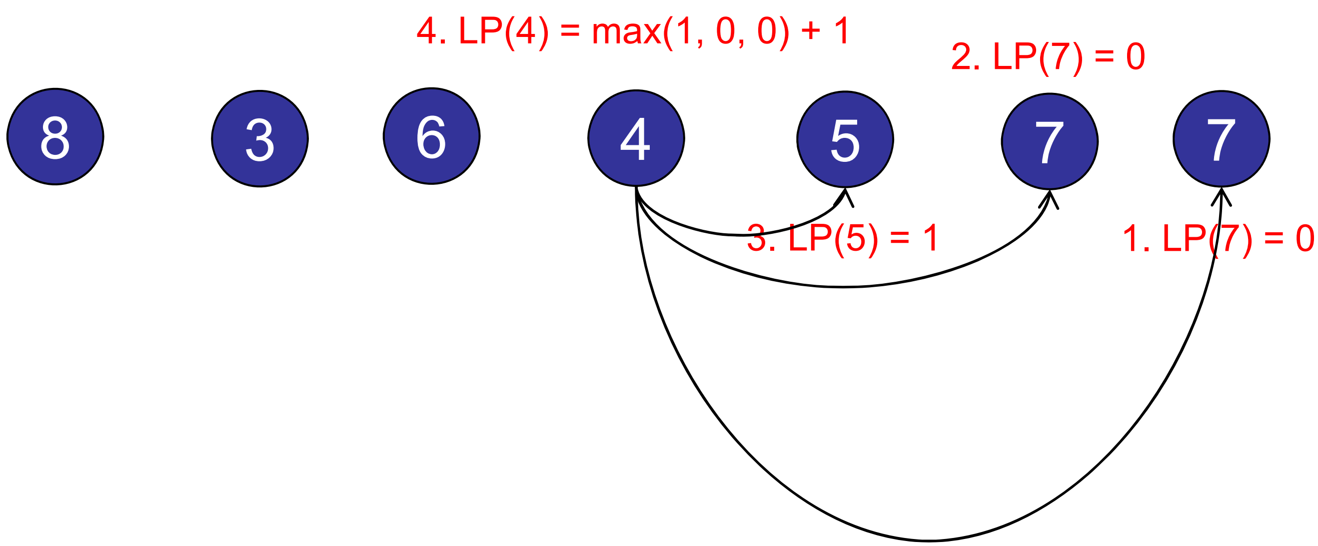
Overlapping Subproblems



Calculate LP(5):

- Examine each outgoing edge.
- Find the maximum.
- Add 1.

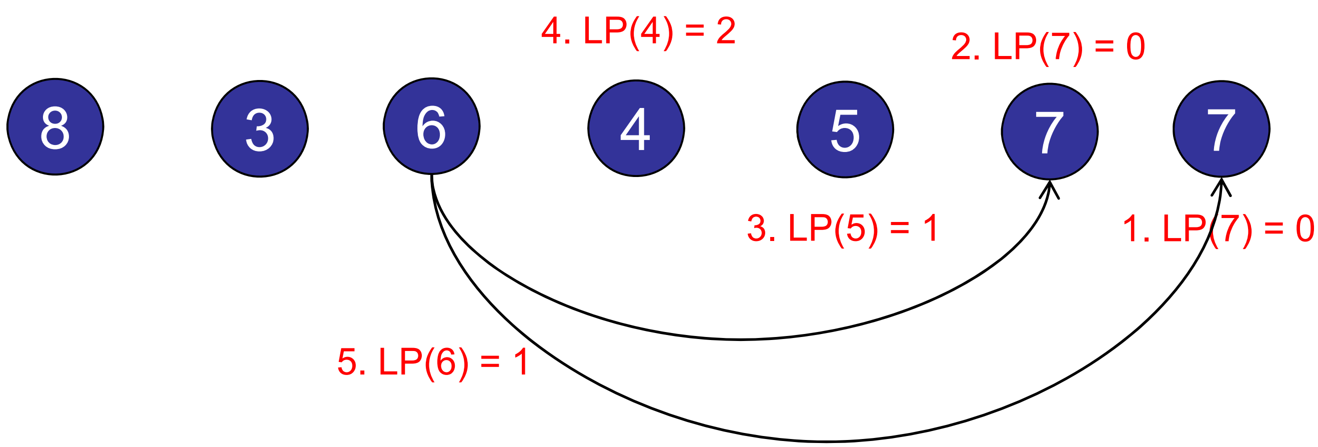
Overlapping Subproblems



Calculate LP(4):

- Examine each outgoing edge.
- Find the maximum.
- Add 1.

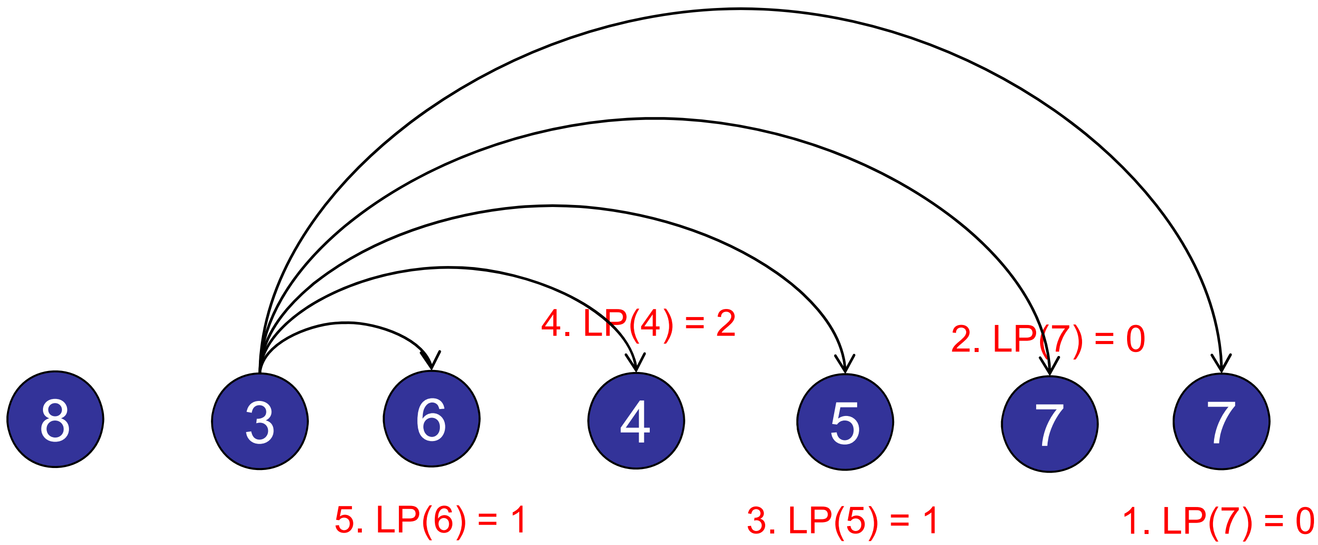
Overlapping Subproblems



Calculate LP(6):

- Examine each outgoing edge.
- Find the maximum.
- Add 1.

Overlapping Subproblems



6. LP(3) = max(1, 2, 1, 0, 0) + 1 = 3

Calculate LP(3):

- Examine each outgoing edge.
- Find the maximum.
- Add 1.

Longest Increasing Subsequence

Input:

- Array A[1..n]

Define sub-problems:

- S[i] = LIS(A[i..n]) starting at A[i]

Example: {8, 3, 6, 4, 5, 7, 7}

- S[5] = 2 → {8, 3, 6, 4, 5, 7, 7}
- S[2] = 4 → {8, 3, 6, 4, 5, 7, 7}

Dynamic Programming

Table view:

Node	Longest path that starts at node X
7	0
7	0
5	…
4	
6	
3	
8	

Longest Increasing Subsequence

Input:

- Array A[1..n]

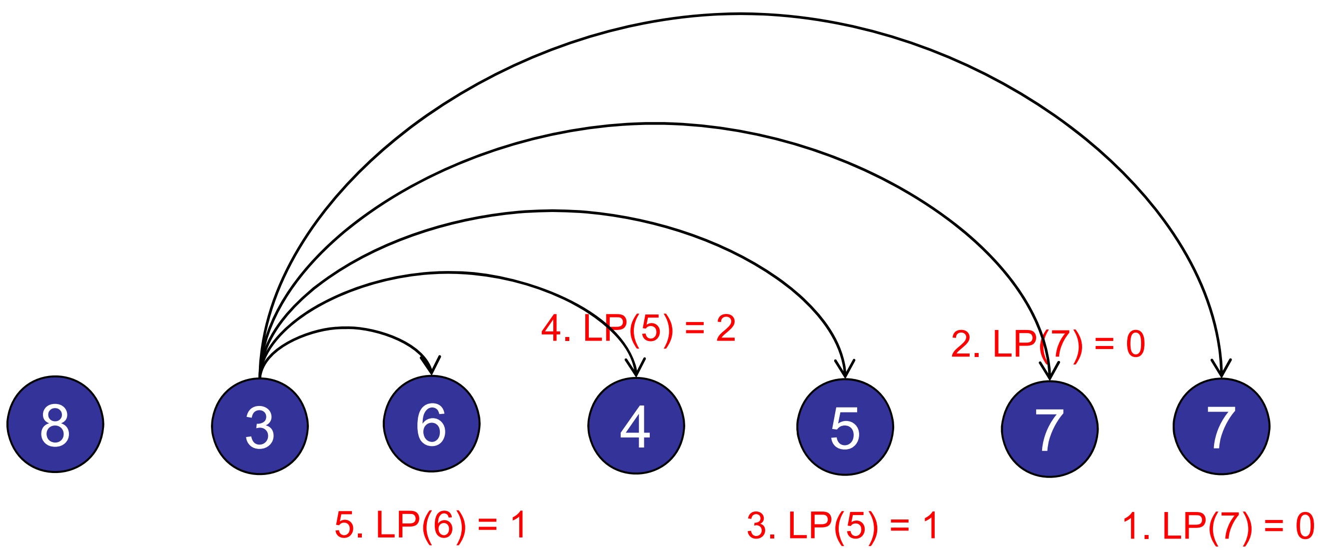
Define sub-problems:

- S[i] = LIS(A[i..n]) starting at A[i]

Solve using sub-problems:

- S[n] = 0
- $S[i] = (\max_{(i,j) \in E} S[j]) + 1$

Overlapping Subproblems



6. LP(2) = max(1, 2, 1, 0, 0) + 1 = 3

Calculate LP(2):

- Examine each outgoing edge.
- Find the maximum.
- Add 1.

Longest Increasing Subsequence

LIS(V): // Assume graph is already topo-sorted

```
int[] S = new int[V.length];  // Create memo array
for (i=0; i<V.length; i++) S[i] = 0; // Initialize array to zero
S[n-1] = 1; // Base case: node V[n-1]
for (int v = A.length-2; v>=0; v--) {
    int max = 0; // Find maximum S for any outgoing edge
    for (Node w : v.nbrList()) { // Examine each outgoing edge
            if (S[w] > max) max = S[w]; // Check S[w], which we already
                                        // calculated earlier.
    }
    S[v] = max + 1; // Calculate S[v] from max of outgoing edges.
}
```

Longest Increasing Subsequence

Input:

- Array A[1..n]

Alternate definition:

- S[i] = LIS(A[1..i]) **ending** at A[i]

Example: {8, 3, 6, 4, 5, 7, 7}

- S[4] = 2 → {8, 3, 6, 4, 5, 7, 7}
- S[5] = 3 → {8, 3, 6, 4, 5, 7, 7}

Longest Increasing Subsequence

Input:

- Array A[1..n]

Alternate definition:

- S[i] = LIS(A[1..i]) **ending** at A[i]

Solve using sub-problems:

- S[1] = 0
- $S[i] = (\max_{(j < i,\ A[j] < A[i])} S[j]) + 1$

Longest Increasing Subsequence

LIS(A):

```
int[] S = new int[A.length];  // Create memo array
for (i=0; i<A.length; i++) S[i] = 0; // Initialize array to zero
S[0] = 1; // Base case: length 1
for (int i = 0; i<A.length; i++) {
    int max = 0; // Find maximum S for any preceding node
    for (int j=0; j<i; j++) { // Examine each preceding element in the sequence
            if (A[j] < A[i]) // If A[i] is bigger than A[j]
                    if (S[j] > max)
                            max = S[j]; // If S[j] is longer sequence
    }
    S[i] = max + 1; // Calculate S[i] from max of preceding elements.
}
```

What is the running time of the LP-LIS alg for a sequence of n numbers?

1. $O(n)$
2. $O(n \log n)$
3. $O(n^2)$
4. $O(n^2 \log n)$
5. $O(n^3)$
6. None of the above.

Longest Increasing Subsequence

Summary:

- Greedy subproblems: S[i] = LIS(A[1..i])
  - n subproblems
  - Subproblem i takes takes times O(i)
- Total time: $O(n^2)$

Challenge of the Day:

How do you solve LIS in time O(n log n)?

Hint: use binary search to solve subproblem faster.

Roadmap

Today: Dynamic Programming

- DP Basics
- Longest Increasing Subsequence
- Prize Collecting
- Vertex Cover on a Tree
- All-Pairs-Shortest-Paths