CS 170
Efficient Algorithms and Intractable Problems

Lecture 12
Dynamic Programming I

Nika Haghtalab and John Wright

EECS, UC Berkeley

Announcements

1. Midterm 1 is done. Yay!

→ We will aim to grade the exams by early next week.

2. Discussion 6 has been out.

→ This and the next lectures will build on it!

→ GO TO SECTIONS!

3. Homework 6 is released.

→ It's a short, so you can rest a bit after studying so hard for Midterm 1.

→ Due next Wednesday (not the usual Monday deadline).

→ No changes to the TA Office Hours and Homework Parties!

→ Our TAs: "Please please go to homework parties (Friday, Monday) and early OH".
OH capacity on Tues-Wed is more limited.

Today

Revisit some discussion material, this time in the context of dynamic programming!

→ Fibonacci numbers. (Disc 6)
→ Bellman-Ford. (Disc 5).

What is dynamic programming (DP) exactly?

This and the next 2 lectures: Many examples of how to design DP algorithms.

→ Shortest path in DAGs
→ All-pair shortest path
→ …

How (not) to compute Fibonacci Numbers

In 61A, you learned to compute Fibonacci number using this code.

Discussion 6 material.

```
def fibo(n):
    if n <= 1:
        return n
    return fibo(n-1) + fibo(n-2)
```

How fast/slow is this?

→ In discussion 6, you'll show that this algorithm runs in time

$$T(n) = T(n-1) + T(n-2),$$

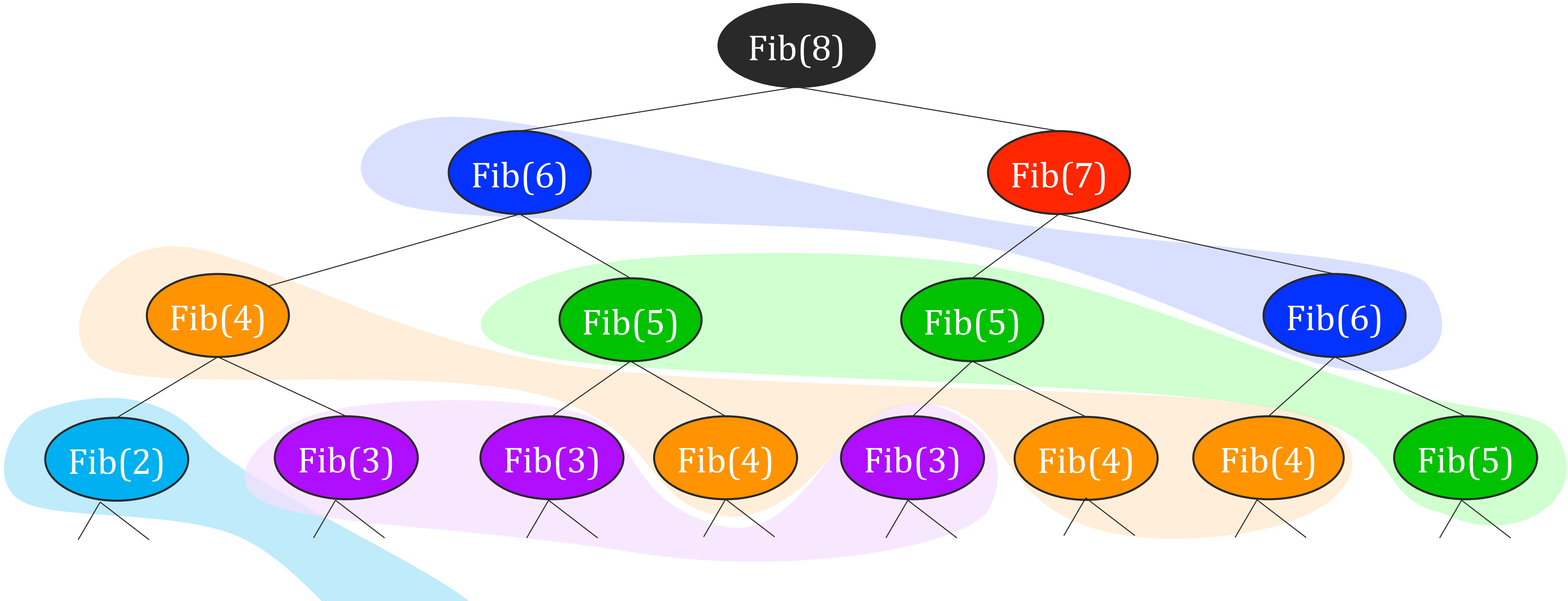
which you will show means that $T(n) \geq 2^{n/2}$.

This is way too big!

What went wrong?

The recursion tree repeats a lot of the subproblems.

→ For every node, it recomputes the problem from scratch.



How to fix this?

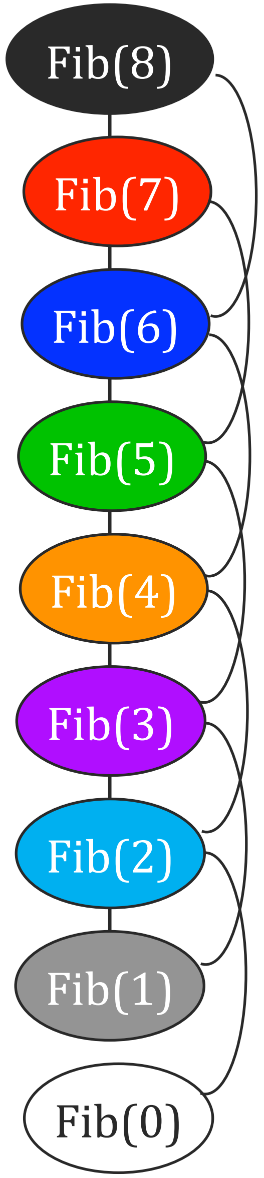
Remember the computations we did elsewhere.

This is called **memo-ization!**

→ keep an array of Fibonacci values **memo**. Whenever a value is computed, store it in there.

```
memo = [0, 1, None, None, …, None ]
def Fib-memo-TopDown(n):
    if memo[n] != None return memo[n]
    else
        memo[n] = Fib-memo-TopDown(n-1)
                + Fib-memo-TopDown(n-2)
    return memo[n]
```

The number of recursive calls to `Fib-memo-TopDown` is $O(n)$.
→we only recurse when the corresponding memo is not yet stored.



Elements of dynamic programming?

1. **Subproblems** (aka "**optimal substructure**"):
→ The fact that large problems break up into sub-problems.
→ So, optimal solution of some big problem (or its computation) can be expressed in terms of the optimal solutions to smaller sub-problems.

E.g., In Fibonacci

$$Fib(i+1) = Fib(i) + Fib(i-1)$$

So far, this seems just like the Divide and Conquer paradigm!

Elements of dynamic programming?

2. **Overlapping subproblems:**
→ A lot of the subproblems overlap. This means that we can save resources by solving a subproblem once and storing its value, and then use that subproblem many times over.

E.g., In Fibonacci $Fib(i+1)$ and $Fib(i+2)$ both directly use $Fib(i)$. Also $Fib(i+3), Fib(i+4), \ldots$. All use $Fib(i)$ indirectly. So, we **memo-ize** $Fib(i)$.

In Dynamic Programming:
We keep a memo (table of solutions) to the smaller problems and use these solutions to solve bigger problems.

This looks new!

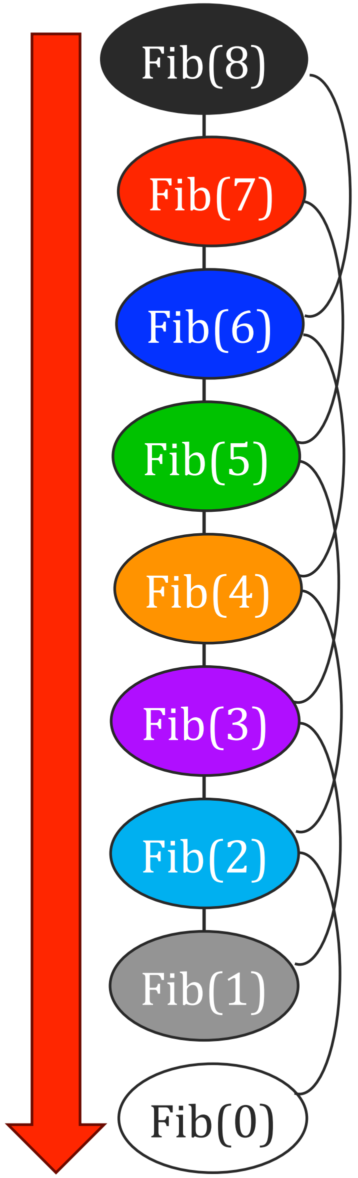
Two ways to do DP:

1. **Top-Down:** We saw this in `Fib-memo-TopDown`.

→Start from the biggest problem and recurse to smaller problems.

→ Looks just like recursion/divide and conquer, with one exception:
  Memo-ization: keeping track of what smaller problems we have solved already

```
memo = [0, 1, None, None, …, None ]
def Fib-memo-TopDown(n):
    if memo[n] != None return memo[n]
    else
        memo[n] = Fib-memo-TopDown(n-1)
                  + Fib-memo-TopDown(n-2)
    return memo[n]
```



Two ways to do DP:

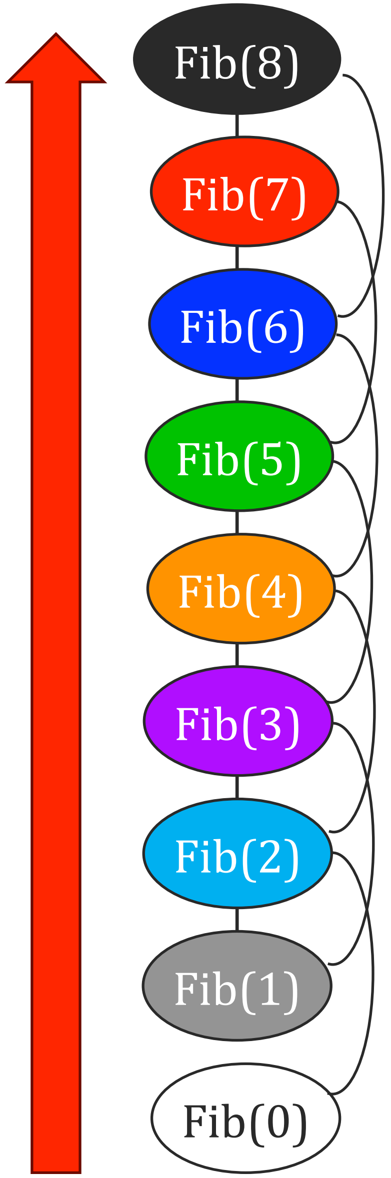
2. Bottom-Up:

→ Start from the smallest problems first and then bigger problems,

→Still memo-ize: Fill in a table of values from small to largest problems.

→Doesn't usually have a recursive call.

```
def Fib-memo-BottomUp(n):
    memo= [0, 1, None, None, …, None ]
    for i = 2, …, n:
        memo[i] = memo[i-1] + memo[i-2]
    return memo[n]
```



Order of Computation and DAGs

There is an implicit DAG in dynamic programming!

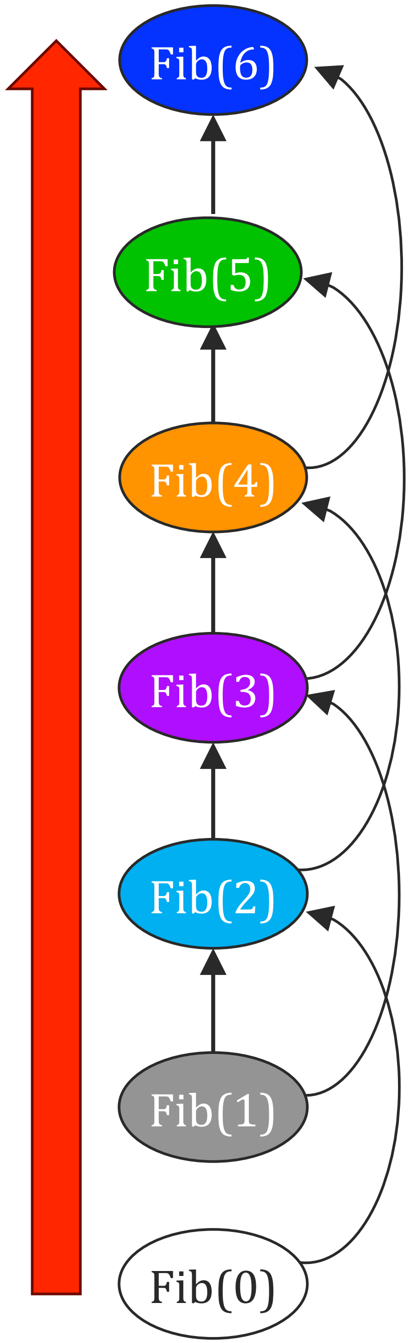
Let's view each subproblem as a node in a graph.

→There is an implicit directed edge (i, j) if the solution to subproblem j directly depends on/uses the solution to subproblem i.

Implicitly, consider a topological sort on this DAG.

→BottomUp: Solve problems in the order of the topological sort!

In Top-Down: We start recursing at the top, but the **memo-ization table is still filled according to the topological sort!**



Recap What's Dynamic Programming?

It's a paradigm in algorithm design.

- Uses **subproblems/optimal substructure**
- Uses **overlapping subproblems**
- Can be implemented **bottom-up** or **top-down.**

Where does the name come from?

Richard Bellman made up the name in 1950s when he was working at RAND corporation --- a think tank funded mostly by the US government and Air Force at the time. Here is what Bellman said of the name:

> "It's impossible to use the word, dynamic, in the pejorative sense…I thought dynamic programming was a good name. It was something not even a Congressman could object to."

Bellman … of Bellman-Ford?!

What's next in this lecture:

Revisiting Shortest Path problems and the Bellman-Ford algorithm.

→These are also examples of dynamic programming algorithms!

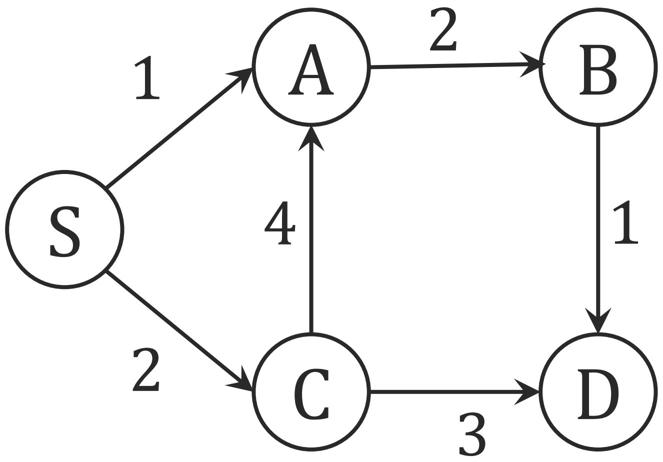
Shortest Paths on DAGs

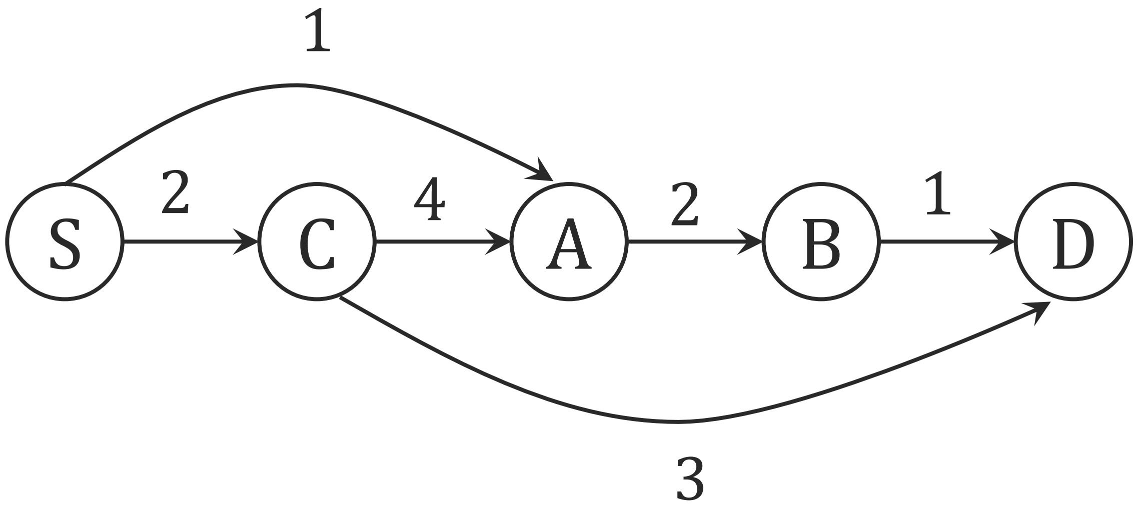
Input: A **DAG** $G = (V, E)$, "source" $S \in V$, edge costs $\ell(u, v)$, positive or negative.
Output: For all $u \in V$, $dist(u) =$ cost of shortest path from s to u.

HW4 Q3 material.

We could run Bellman-Ford of runtime $O(nm)$ but we want to do much better.
→ We will aim for $O(n + m)$.

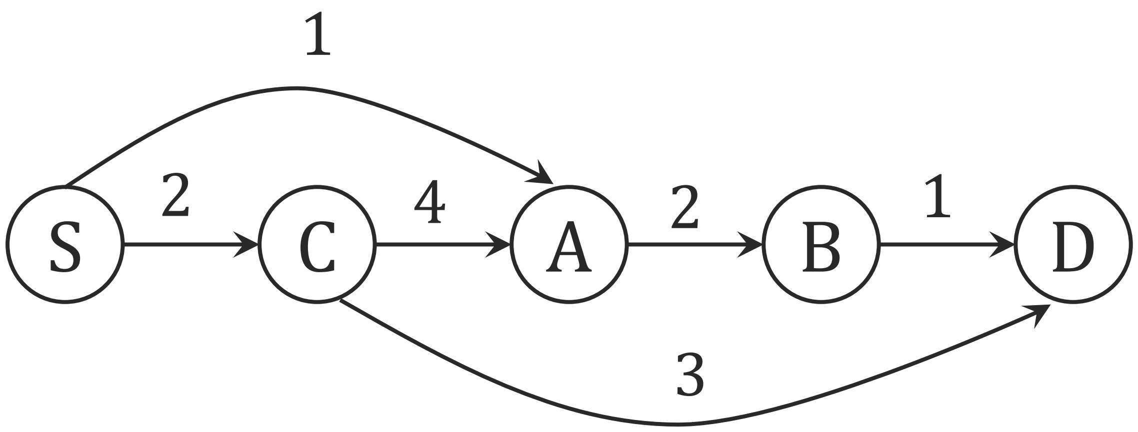
Recall, we can always do topological sort on a DAG in $O(n + m)$.





Shortest Paths on DAGs: Subproblems

<u>Input:</u> A DAG $G = (V, E)$, "source" $S \in V$, edge costs $\ell(u, v)$, positive or negative.
<u>Output:</u> For all $u \in V$, $dist(u) =$ cost of shortest path from s to u.



What are the subproblems?

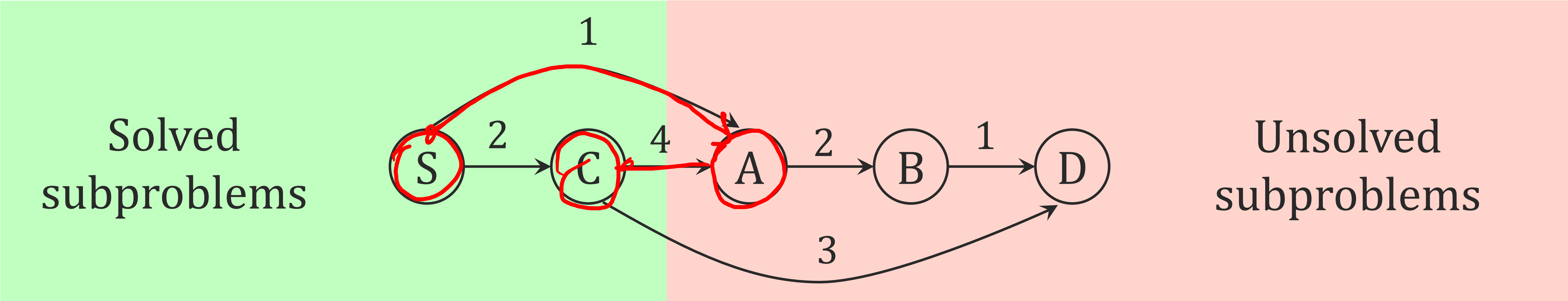
→ One subproblem per node, $dist(u)$ for all $u \in V$.

→ **A natural order to them:** smaller subproblem for nodes that appear earlier in the topological sort.

The Dynamic Programming's implicit DAG is the same as this DAG!

Shortest Paths on DAGs: Recurrence

<u>Input:</u> A DAG $G = (V, E)$, "source" $S \in V$, edge costs $\ell(u, v)$, positive or negative.
<u>Output:</u> For all $u \in V$, $dist(u)$ = cost of shortest path from s to u.



What does the recurrence relationship look like?

→ If we have already computed $dist[S] = 0$ and $dist[C] = 2$

$$dist[A] = \min\left(dist(C) + \overset{\ell(C,A)}{4}, \; dist(S) + \overset{\ell(S,A)}{1} \right) = 1$$

→ Generally, $dist[u] \leftarrow \min_{(v,u)\in E} \{dist[v] + \ell(v, u)\}$

Shortest Paths on DAGs: Algorithm

Input: A DAG $G = (V, E)$, "source" $S \in V$, edge costs $\ell(u, v)$, positive or negative.
Output: For all $u \in V$, $dist(u) =$ cost of shortest path from s to u.

Runtime:

- Topological Sort: $O(m + n)$.
- Number of subproblems: $O(n)$.
- For each vertex $u \in V$, the update step considers all of its incoming edges.
  → So, $O(\text{indeg}(u))$ for node u
  → So, overall $O(m)$ for updates

Total time: $O(m + n)$.

SSSP-DAG($G = (V, E), s$)

  array $dist$ of length n

  $dist = 0$ and $dist[u] = \infty$ for all other $u \in V$.

  For $u \in V$ in topological sort order

    $dist[u] \leftarrow \min_{(v,u) \in E} \{dist[v] + \ell(v, u)\}$

  return $dist$

Dynamic Programming Recipe

- **Step 1:** Identify the subproblems (optimal substructure)

- **Step 2:** Find a recursive formulation for the subproblems

- **Step 3:** Design the dynamic programming algorithm

→ Fill in a table, starting with the smallest sub-problems and building up.

More Shortest Paths:
Reliable Shortest Paths and Bellman-Fod

Shortest Path Revisited

We saw in lecture 8, that Bellman-Ford solves the Single-Source Shortest Path Problem (SSSP) with negative weights (no negative cycle).

Bellman-Ford:
$k = n - 1$

Bellman-Ford(G, s)

for i=1, , ..., $n - 1$

for $(u, v) \in E$,

$dist[v] \leftarrow \min\{dist[v], dist[u] + \ell(u, v)\}$

DP in disguise!

We saw a related version of SSSP problem in Discussion 5:

2 Bellman Ford Properties

e) Given a value of $k < n$, describe a $O(kn + km)$ algorithm to find the cost of the shortest path from s to every vertex v that uses at most k edges.

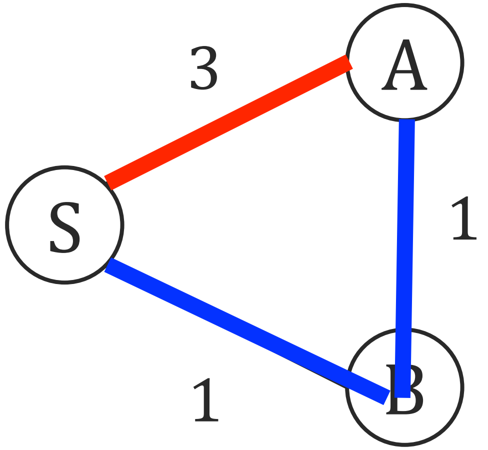
Discussion 5 material.

"Reliable" Shortest Path

Cost can be negative, but no negative cycles

Input: Graph $G = (V, E)$, "source" $S \in V$, edge costs $\ell(u, v)$ for $(u, v) \in E$, parameter k
Output: For all $u \in V$, $dist_k(u)$ = cost of shortest path from s to u, that uses $\leq k$ edges.

Shortest S-A path



Shortest S-A path for $k = 1$.

Discussion 5 material.

Dynamic Programming Recipe

- **Step 1:** Identify the subproblems (optimal substructure)

- **Step 2:** Find a recursive formulation for the subproblems

- **Step 3:** Design the dynamic programming algorithm

→ Fill in a table, starting with the smallest sub-problems and building up.

Sub-Problems

Input: Graph $G = (V, E)$, "source" $S \in V$, edge costs $\ell(u, v)$ for $(u, v) \in E$, parameter k
Output: For all $u \in V$, $dist_k(u)$ = cost of shortest path from s to u, that uses $\leq k$ edges.

What are the subproblems?

- $dist_i(u)$ for all $u \in V$. all $i = 0, \dots, k$
- Every subproblem tracks the cost of the shortest s-u path using $\leq i$ edges.

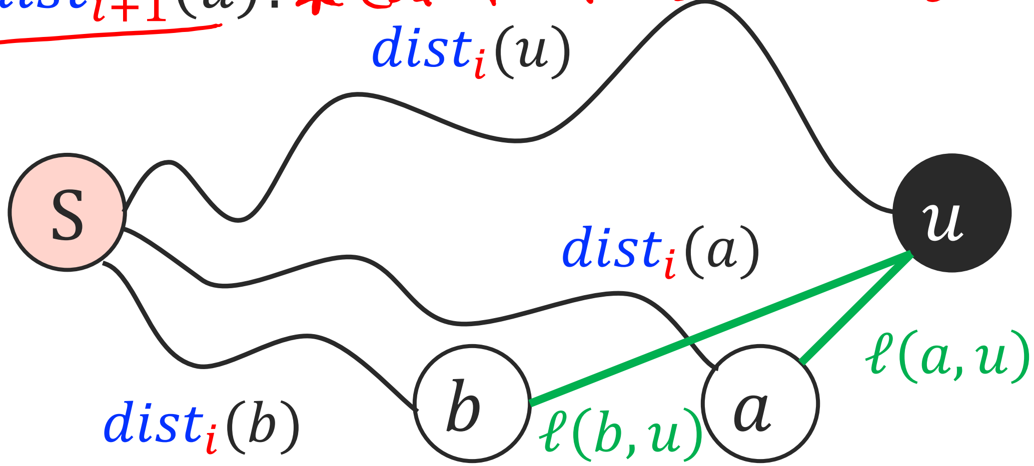
Discussion 5 material.

The Recurrence Relation

Input: Graph $G = (V, E)$, "source" $S \in V$, edge costs $\ell(u, v)$ for $(u, v) \in E$, parameter k
Output: For all $u \in V$, $dist_k(u)$ = cost of shortest path from s to u, that uses $\leq k$ edges.

What is the recurrence relation?

- Say, we have compute $dist_1(u), dist_2(u), \ldots, dist_i(u)$ for all $u \in V$.
- How do we compute $dist_{i+1}(u)$? * cost of SP s-u using $\leq$ i+1 edges.



Case 1: The shortest path uses $< i + 1$ edges

Case 2: The shortest path uses $= i + 1$ edges

case 1

$dist_{i+1}(u) = \min\{ \qquad \}$

$\leq dist_i(u)$

Discussion 5 material.

Design the Algorithm

Input: Graph $G = (V, E)$, "source" $S \in V$, edge costs $\ell(u, v)$ for $(u, v) \in E$, parameter k
Output: For all $u \in V$, $dist_k(u)$ = cost of shortest path from s to u, that uses $\leq k$ edges.

Given recurrence relation, how to memo-ize?

$$dist_{i+1}(u) = \min\{dist_i(u), \underbrace{\min_{(v,u)\in E}\{dist_i(v) + \ell(v, u)\}}\}$$

	s	a	b	$\cdots$	u
$dist_0$	0	∞	∞		∞
$\vdots$					
$dist_i$					
$dist_{i+1}$					
$\vdots$					
$dist_{n-1}$					

$\boxed{\text{Bellman-Ford: } k = n - 1}$

DP DAG:
Arrows where $(v, u) \in E$

Runtime of this algorithm

Input: Graph $G = (V, E)$, "source" $S \in V$, edge costs $\ell(u, v)$ for $(u, v) \in E$, parameter k
Output: For all $u \in V$, $dist_k(s, u) =$ cost of shortest path from s to u, that uses $\leq k$ edges.

Computation for each table row:
→ Goes through every edge
→Total computation: $O(km)$.

Number of subproblems to track in the table?
→$O(kn)$.

Overall runtime: $O(kn + km)$.

Reliable-SSSP($G = (V, E), s, k$)

arrays $dist_0$, $dist_1$,..., $dist_k$ of length n

$dist_0[s] = 0$ and $dist_0[u] = \infty$ for all other $u \in V$.

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

For $u \in V$:

$$dist_i[u] \leftarrow \min\{dist_{i-1}[u],\ \min_{(v,u)\in E}\{dist_{i-1}[v] + \ell(v, u)\}\ \}$$

All-Pair Shortest Path Problem

All-Pair Shortest Path (APSP)

We want to know the shortest distance between any pair of nodes in a graph.

→ Not just from a special single source.

→ Another example of DP!

Input: Graph $G = (V, E)$, edge costs $\ell(u, v)$ for $(u, v) \in E$ (not necessarily positive)
Output: For all $u, v \in V$, $dist(u, v)$ = cost of shortest path from u to v

Naïve algorithm:
→ Run Bellman-Ford starting from every $s \in V$ as a source.
→ Bellman-Ford runs in time $O(mn)$
  → Total runtime for APSP would be $O(n^2 m)$. Could be as large as $O(n^4)$ for dense graphs. We are aiming for $O(n^3)$.

Dynamic Programming Recipe

- **Step 1:** Identify the subproblems (optimal substructure)

- **Step 2:** Find a recursive formulation for the subproblems

- **Step 3:** Design the dynamic programming algorithm

→ Fill in a table, starting with the smallest sub-problems and building up.

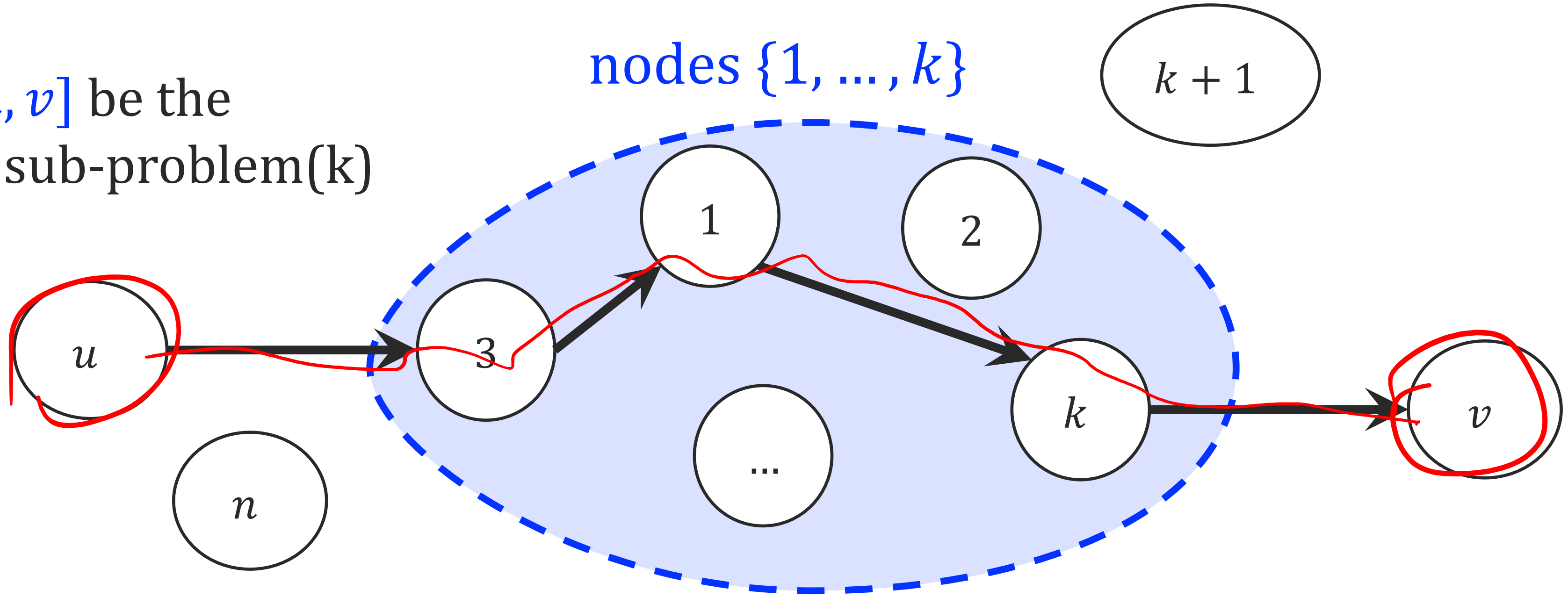
Identify the subproblems (optimal substructure)

Sub-problem(k): For all pairs $u, v \in V$, find the shortest u-v path whose internal vertices only use nodes $\{1, \dots, k\}$.

→ Sub-problem (n) is the APSP we want to solve.

→This may look unintuitive, but let's see why it's helpful!

Let $dist_k[u, v]$ be the solution to sub-problem(k)



This is an overview picture, not all edges are shown.

Dynamic Programming Recipe

- **Step 1:** Identify the subproblems (optimal substructure)

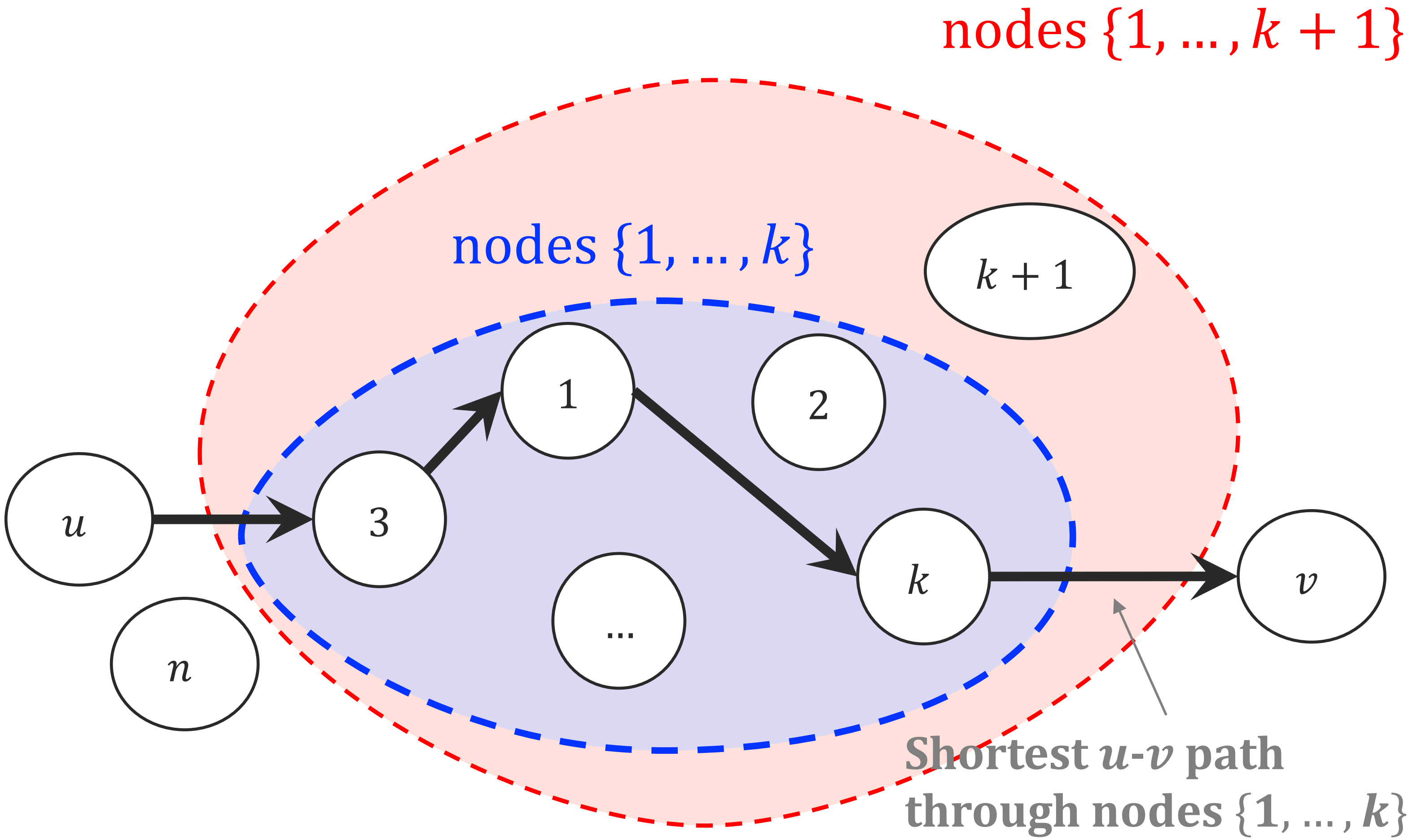
- **Step 2:** Find a recursive formulation for the subproblems

- **Step 3:** Design the dynamic programming algorithm

→ Fill in a table, starting with the smallest sub-problems and building up.

Recursive Formulation

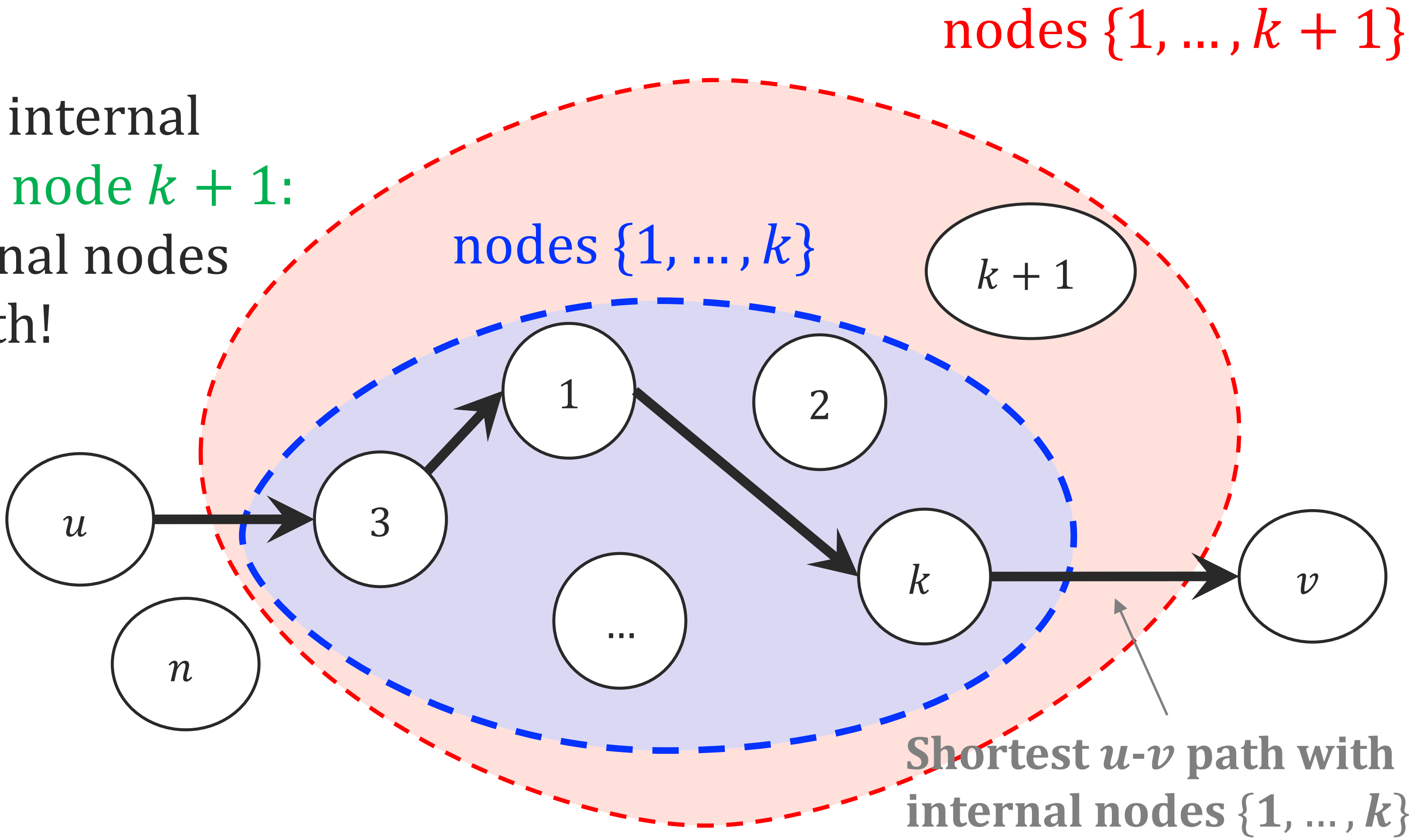
How do I solve **sub-problem(k+1)** knowing all the solutions $dist_k(u, v)$ to **Sub-problem(k)?**



Recursive Formulation

How do I solve **sub-problem(k+1)** knowing all the solutions $dist_k(u, v)$ to **Sub-problem(k)?**

Case 1: Shortest u-v path with internal nodes $\{1, \dots, k+1\}$ doesn't use node $k+1$:
→ Shortest u-v path with internal nodes $\{1, \dots, k\}$ is still the shortest path!
$dist_{k+1}(u, v) = dist_k(u, v)$!



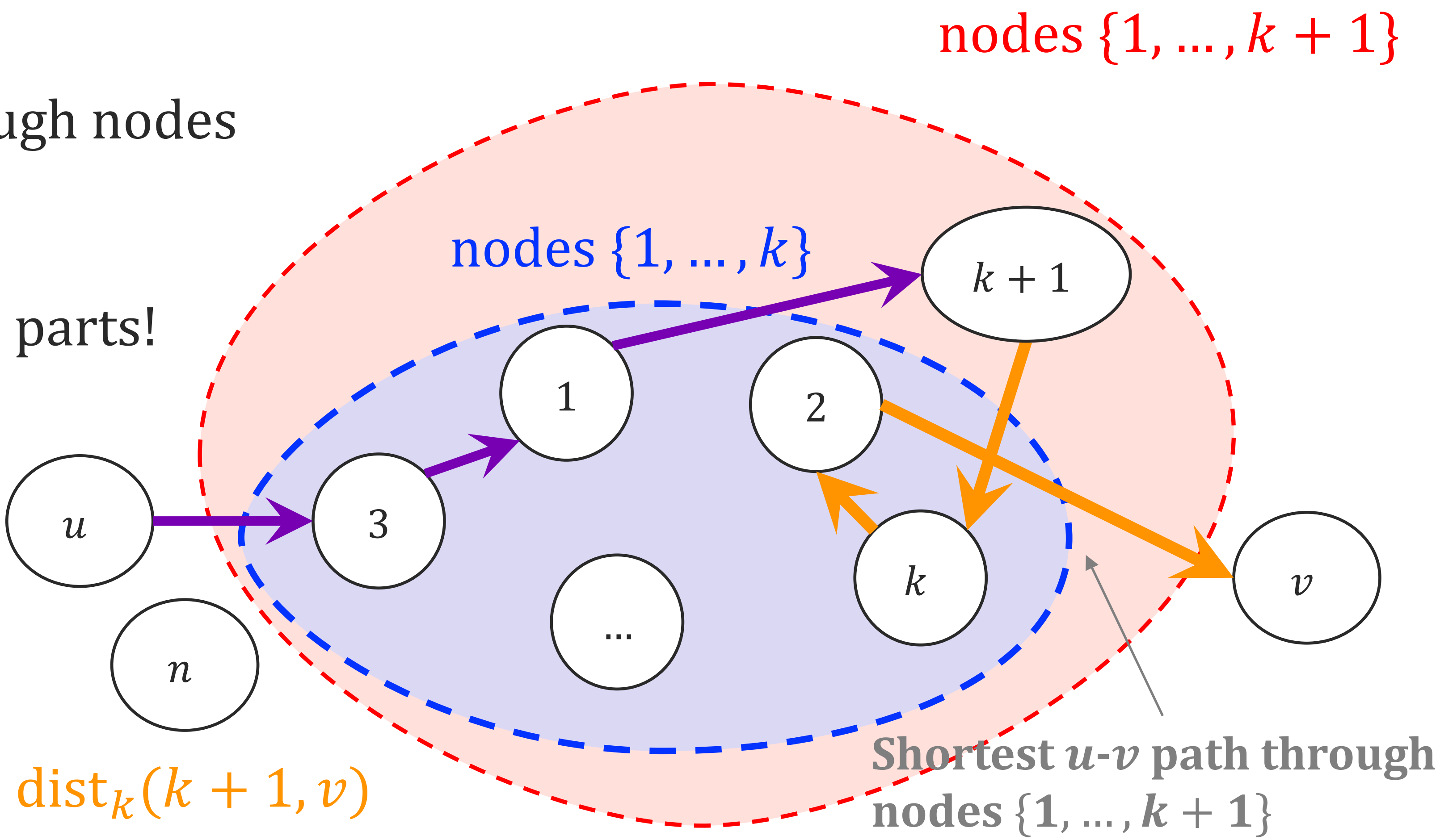
Recursive Formulation

How do I solve **sub-problem(k+1)** knowing all the solutions $dist_k(u, v)$ to **Sub-problem(k)?**

Case 2: Shortest u-v path through nodes $\{1, \dots, k+1\}$ uses node $k+1$:

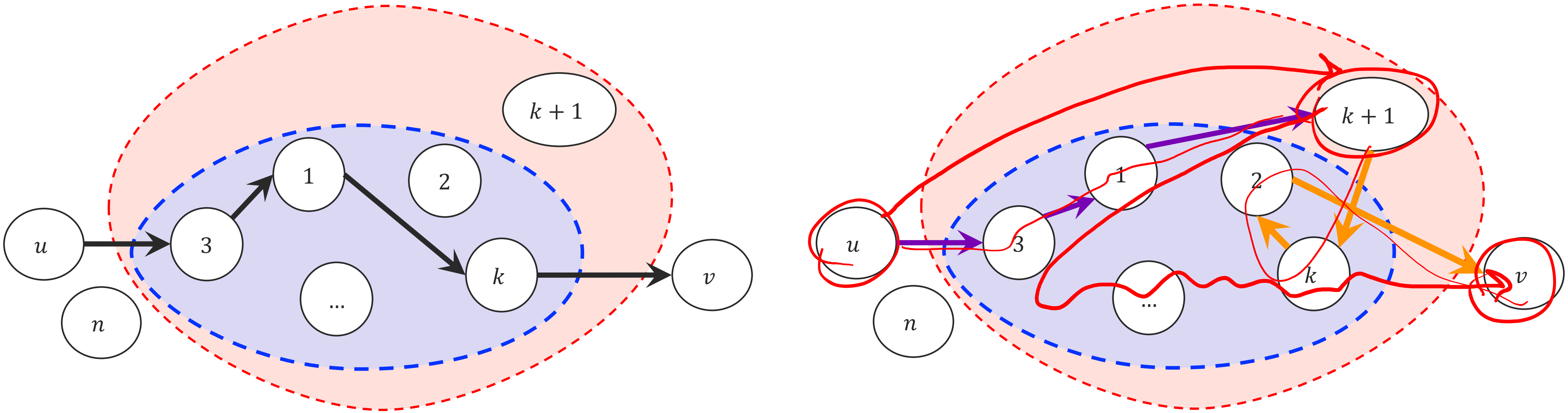
This path can be broken to two parts!

- Shortest u-$(k+1)$ path
- Shortest $(k+1)$-v path
- Both using internal nodes $\{1, \dots, k\}$ only.



$$dist_{k+1}(u, v) = dist_k(u, k+1) + \mathrm{dist}_k(k+1, v)$$

Putting the two cases together



Case 1: Shortest u-v path with internal nodes $\{1, \ldots, k+1\}$ doesn't use node $k+1$:

Case 2: Shortest u-v path with internal nodes $\{1, \ldots, k+1\}$ uses node $k+1$:

$dist_k(u, k+1) + dist_k(k+1, v)$

The recursive solution for All-Pair Shortest Path

$$dist_{k+1}(u,v) = \min\{dist_k(u,v)\,,\, dist_k(u, k+1)\ + \mathrm{dist}_k(k+1, v)\}$$

Dynamic Programming Recipe

- **Step 1:** Identify the subproblems (optimal substructure)

- **Step 2:** Find a recursive formulation for the subproblems

- **Step 3:** Design the dynamic programming algorithm

→ Fill in a table, starting with the smallest sub-problems and building up.

The Floyd-Washall Algorithm for APSP

Input: Graph $G = (V, E)$, edge costs $\ell(u, v)$ for $(u, v) \in E$ (not necessarily positive)
Output: For all $u, v \in V$, $dist(u, v) =$ cost of shortest path from u to v

Each update is just $O(1)$.

The loop over k and u, v repeats $O(n^3)$ times.

Overall, $O(n^3)$ runtime.

Floyd-Warshall ($G = (V, E)$)

$n \times n$ matrices $dist_0, dist_1, \ldots, dist_n$ initialized to ∞

For $(u, v) \in E$, $dist_0[u, v] \leftarrow \ell(u, v)$

// $dist_0$ paths have no internal nodes.

$$\left.\begin{array}{l}\textbf{For } k = 1, \ldots, n: \quad n \\ \quad \textbf{For } u, v \in V: \quad n^2\end{array}\right\}$$

$dist_k[u, v] \leftarrow \min\{dist_{k-1}[u, v],$ $dist_{k-1}[u, k] + dist_{k-1}[k, v]\}$ $O(1)$

Wrap up

We saw a recipe for dynamic programming:
Step 1: Identify the subproblems
Step 2: Figure out the recursive structure
Step 3: Design the DP algorithm by solving smallest to largest problem and memo-izing!

We saw some examples:

- Fibonacci
- Shortest Path on DAGs
- Bellman-Ford again
- Floyd-Warshall.

Next time

- More examples of DP