CS 170
Efficient Algorithms and Intractable Problems

Lecture 11
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.
→ While waiting, Pls don't ask questions about the exam until then.
→ We will have TA-student 1-1 chats in the next couple week: discuss micterm performance, career advice, etc.

2. Mid-semester Feedback form will be released with midterm grades

→ Extra HW drop opportunity if you fill it out!

Today

Finish up greedy!

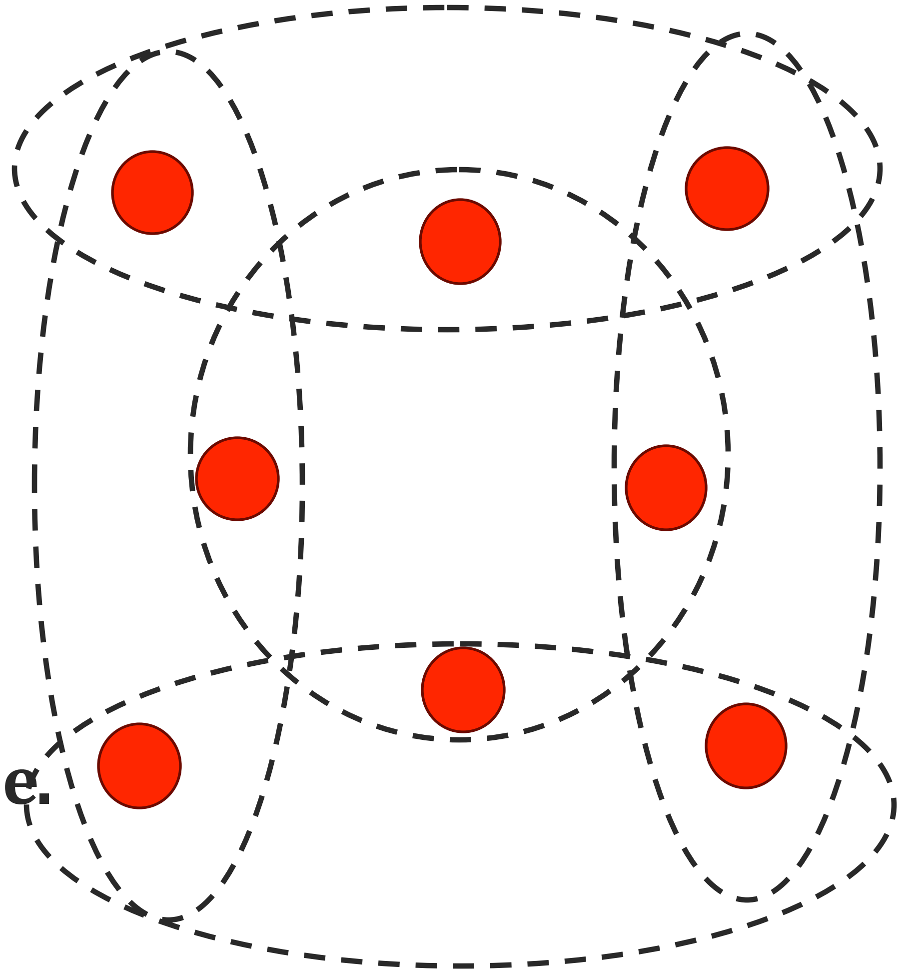
Start a new topic:
→ Dynamic Programming!

Recap: The Set Cover Problem

Input:

→Universe of n elements $U = \{1, \ldots, n\}$, and

→Subsets $S_1, S_2, \ldots, S_m \subseteq U$, s.t., $\bigcup_{i=1}^{m} S_i = U$



Output:

A collection of subsets covering U of **minimal size.**

i.e., $J \subseteq \{1,2, \ldots, m\}$ s.t., $\bigcup_{i \in J} S_i = U$

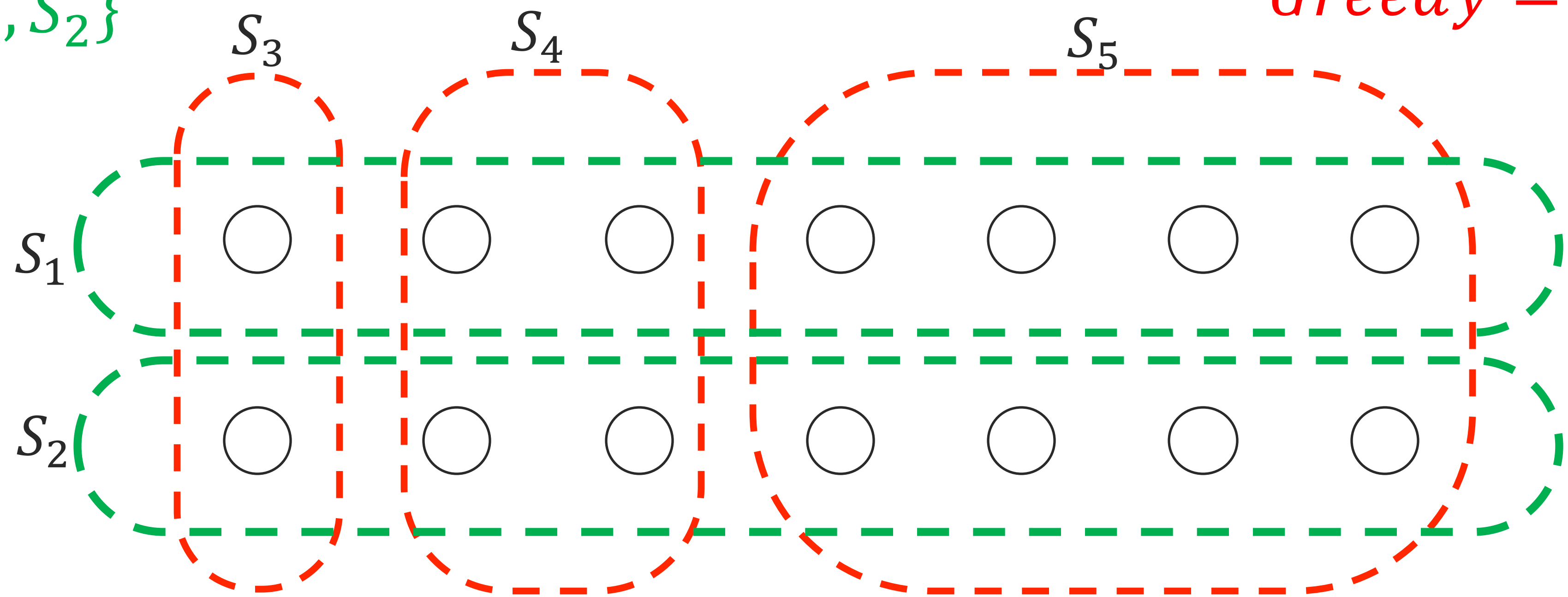
Greedy is Not Optimal

A suggested greedy algorithm:

Repeat until all elements of U are covered: Pick the set with the largest number of uncovered elements.

$OPT = \{S_1, S_2\}$

$Greedy = \{S_3, S_4, S_5\}$



Greedy is Approximately Optimal

Claim: For any instance of the Set Cover problem. If the optimal solution uses k sets, the Greedy algorithm uses at most $k \ln(n)$ sets.

Proof: Let n_t be the number of elements not covered after t step of the Greedy algorithm. (E.g., $n_0 = n$).

Our goal: Show that for $t = k \ln(n)$, $n_t < 1$.
→ If we achieve this goal; then we have $n_t = 0$. i.e., all elements of the set are covered by Greedy after $k \ln(n)$ rounds.

Greedy is Approximately Optimal

Let n_t be the number of elements not covered after t step of the Greedy algorithm.

Our goal: Show that for $t = k \ln(n)$, $n_t < 1$.

Subclaim 1: $n_1 \leq n_0 - \frac{n_0}{k}$

Greedy is Approximately Optimal

Let n_t be the number of elements not covered after t step of the Greedy algorithm.

Our goal: Show that for $t = k\ln(n)$, $n_t < 1$.

Subclaim 2: For any t, $n_{t+1} \leq n_t(1 - 1/k)$

Very similar proof as before.

Greedy is Approximately Optimal

Let n_t be the number of elements not covered after t step of the Greedy algorithm.

Our goal: Show that for $t = k\ln(n)$, $n_t < 1$.

Repeatedly applying subclaim 2, we have that for any t

$$n_t \le n_{t-1}\left(1 - \frac{1}{k}\right)$$

Final subclaim: $n\left(1-\frac{1}{k}\right)^{k\ln(n)} < 1$.

Proof: We use a mathematical fact that for any $x \neq 0$, $1 - x < e^{-x}$.

Approximation Factor

We showed that Greedy does not find the optimal set cover.

We also showed that Greedy outputs $\leq k \ln(n)$ sets, where $k = \text{OPT}$ is the number of sets used in the optimal solution.

"Greedy has an **approximation factor** of $\ln(n)$ for Set Cover"

Formally, approximation factor of an algorithm (for minimizing cost) is

$$\frac{\text{Cost(Alg(input x))}}{\text{Cost(optimal solution for input x)}}$$

What is the best polynomial time approximation algorithm for Set Cover?
Greedy! Meaning, **approximation factor** $< \ln(n)$ is not achievable in polynomial time.

Show at home: Greedy's approximation factor is no better than $\ln(n)$.
→ Generalize our first "bad" example showing Greedy is not optimal.

Done with Greedy!!!

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

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

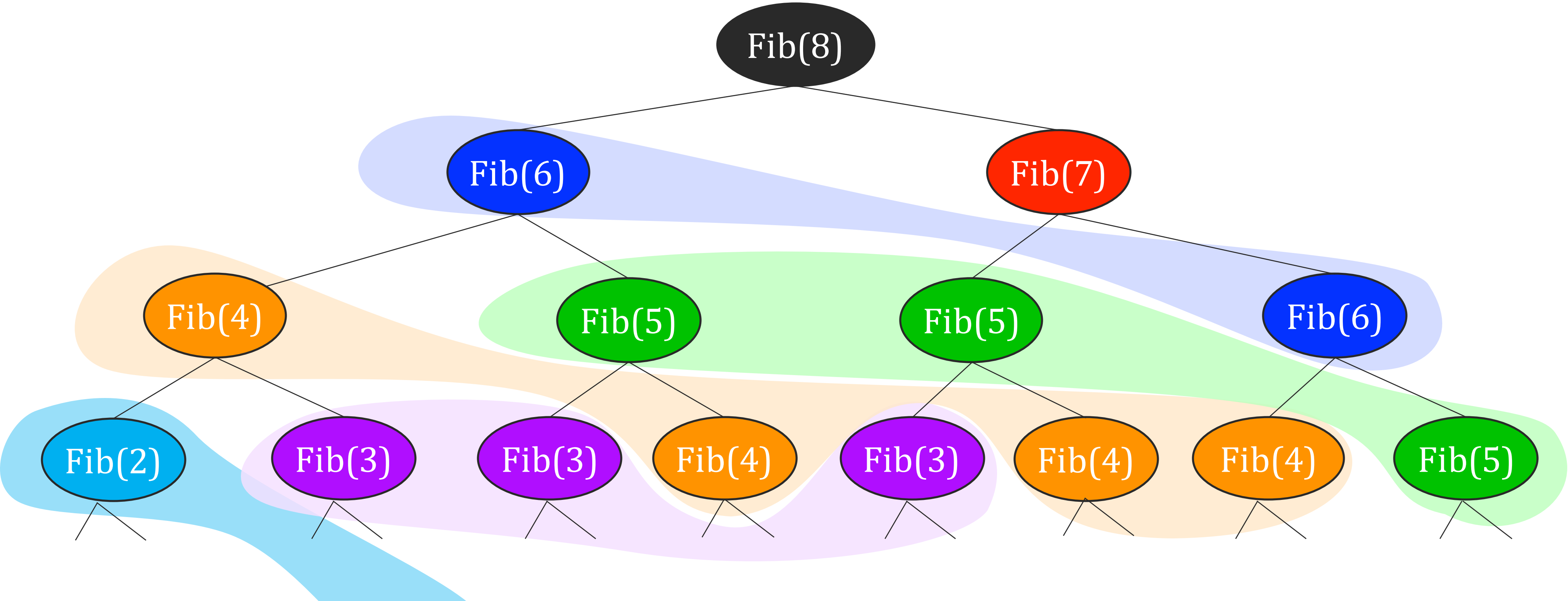
which you will show means that $T(n) \geq 2^{n/2}$.

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?

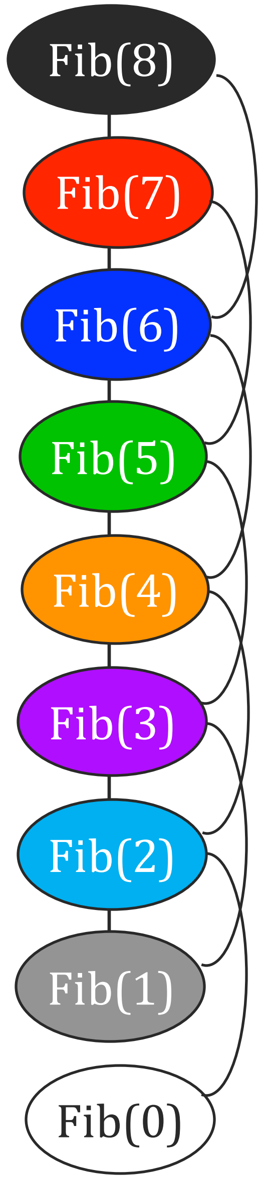
Remember the computations we did elsewhere.

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 $Fib(i+3), Fib(i+4), \dots$. 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!

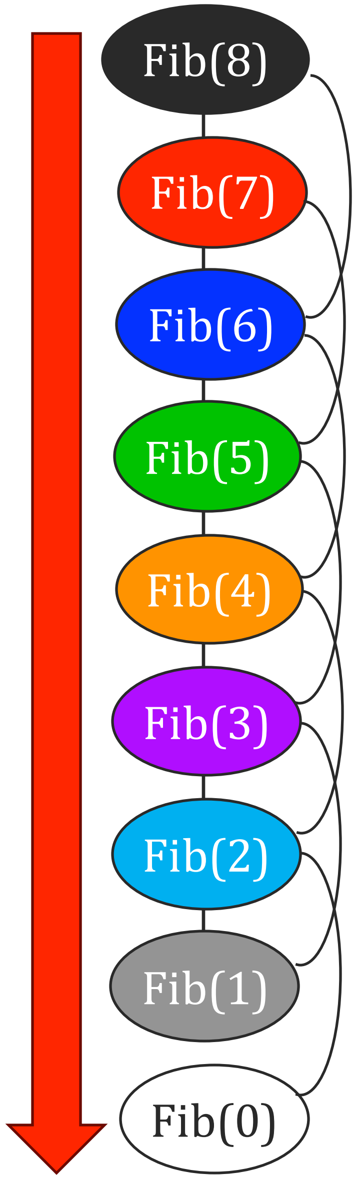
Two ways to do DP:

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:

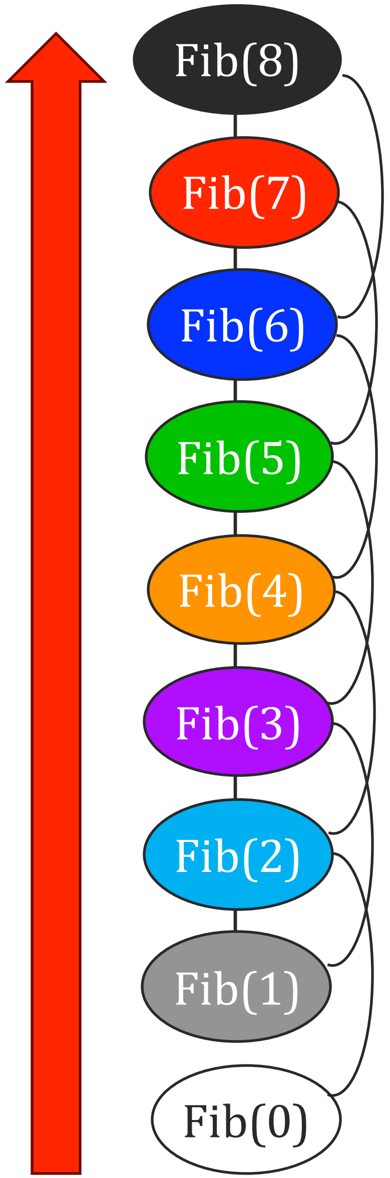
2. Bottom-Up:

→ 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!

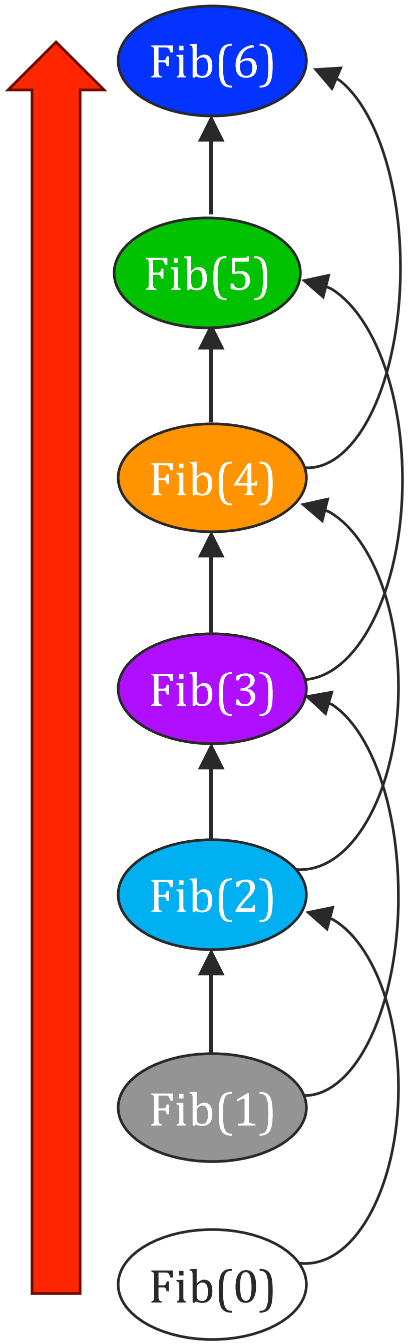
Let's view each subproblem as a node in a graph.

→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."

Revisiting Shortest Path problems

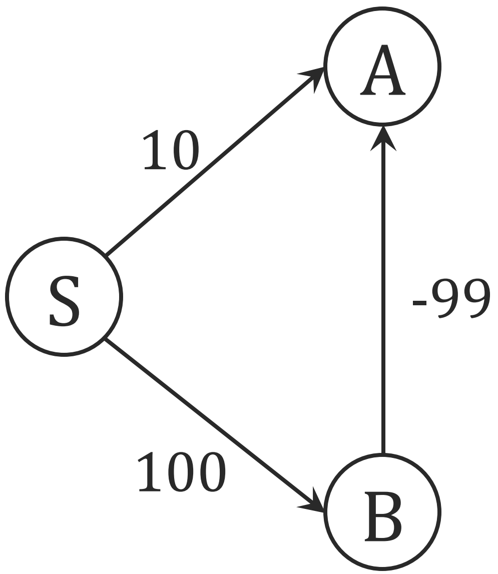
This time with negative weights!

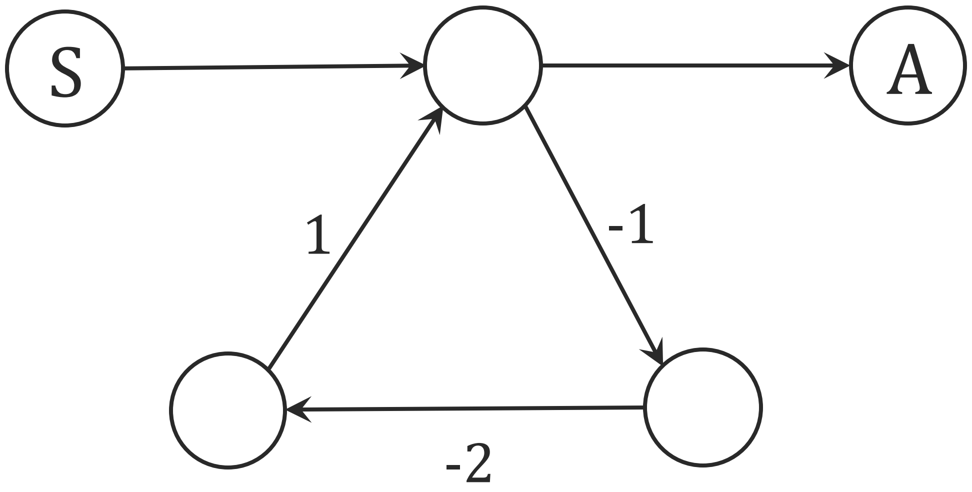
Negative Weights on Shortest Paths

We saw Dijkstra for computing Single Source Shortest Paths on directed graphs with positive edge lengths.

Sometimes there are **negative weights** on graphs:

- Instead of total cost, recording cost saved/spent





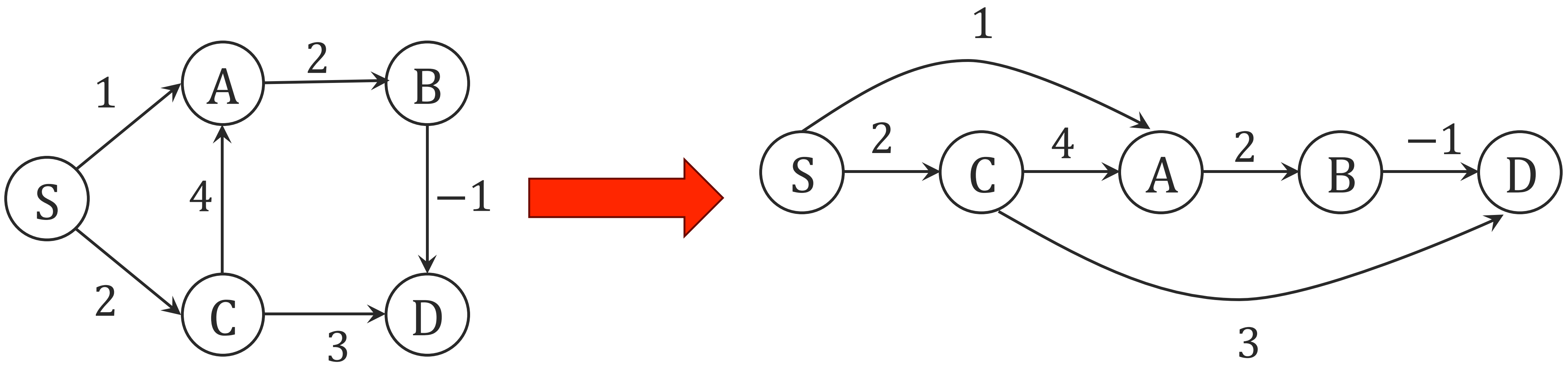
Shortest path is well-defined if no cycle has negative length.

Shortest Paths on DAGs

<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.

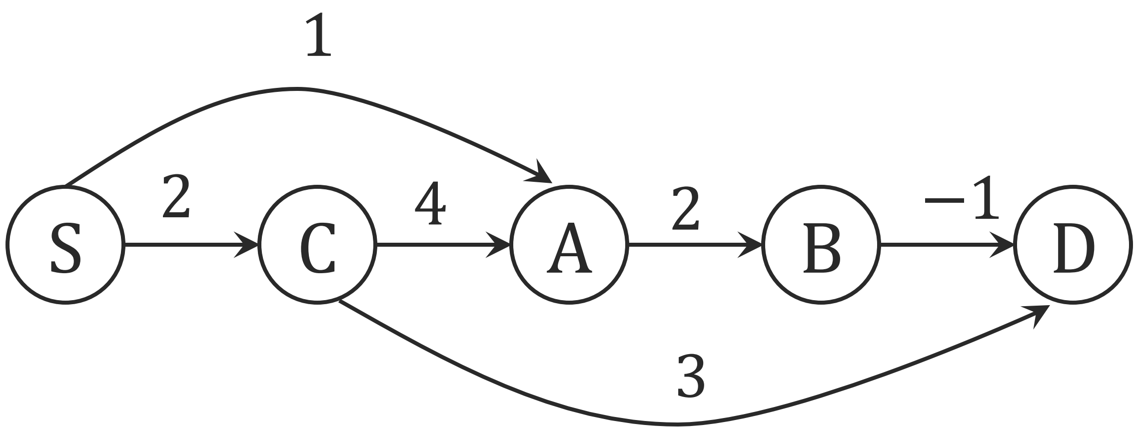
We want to aim for a $O(n + m)$ algorithm.
→ Even with just positive weight, Dijkstra works but it's $O\big((n + m)\log(n)\big)$.

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



Shortest Paths on DAGs: Subproblems

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.

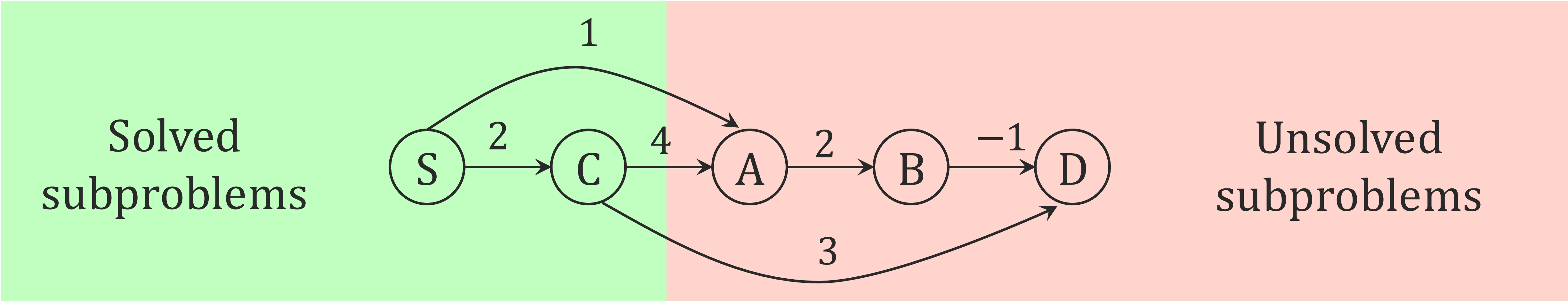


What are the subproblems?
→ One subproblem per node, $dist(u)$ for all $u \in V$.
→ **A natural order to them:** smaller subproblem for nodes that appear earlier in the topological sort.

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

Shortest Paths on DAGs: Recurrence

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.



Discuss

Write the recurrence relation for $dist[u]$:

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**.
  → $O(\text{indeg}(u))$ 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

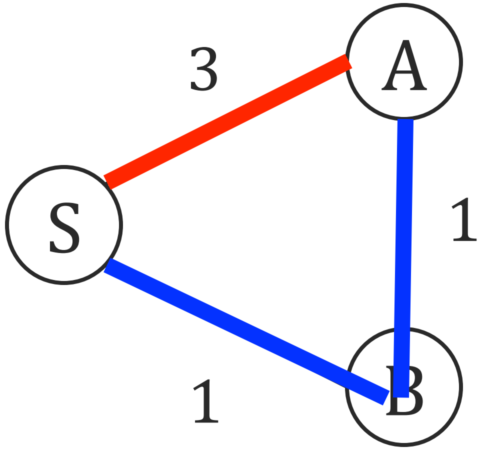
“Reliable” Shortest Path

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 $k = 1$.

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 $u \in V$.
- Every subproblem tracks the **cost of the shortest s-u path using** $\leq i$ edges.

The Recurrence Relation

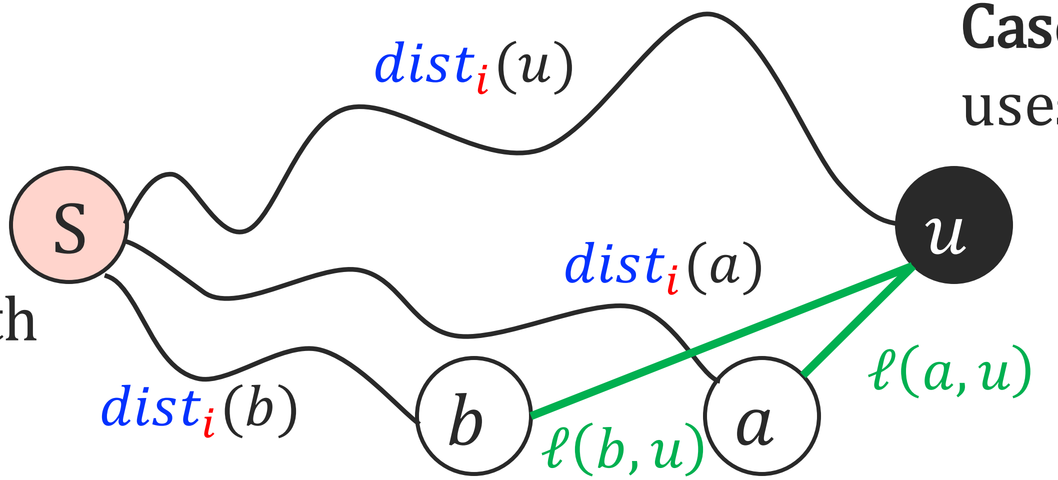
Input: Graph $G = (V, E)$, "source" $S \in V$, edge costs $\ell(u, v)$ for $(u, v) \in E$, parameter k
Output: For all $u \in V$, $dist_k(u)$ = cost of shortest path from s to u, that uses $\leq k$ edges.

Discuss

Say, we have compute $dist_1(u), dist_2(u), \ldots, dist_i(u)$ for all $u \in V$.

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

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



What is the recurrence relation for

$$dist_{i+1}(u) =$$

Design the Algorithm

<u>Input:</u> Graph $G = (V, E)$, "source" $S \in V$, edge costs $\ell(u, v)$ for $(u, v) \in E$, parameter k

<u>Output:</u> 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?

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

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

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 $u \in V$, $dist_k(u)$ = 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?
→$O(kn)$, but could reduce to $O(n)$

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

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

arrays $dist_0, dist_1, \ldots, dist_k$ of length n

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

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

For $u \in V$:

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

Bellman-Ford Algorithm

Shortest Path with Negative Weights

- Input: A $G = (V, E)$, "source" $S \in V$, edge costs $\ell(u, v) \in \mathbb{R}$. No negative cycles.
- Output: For all $u \in V$, $dist(u)$ = cost of shortest path from s to u.

This is the same problem statement as "reliable" Shortest Path when the number of edges (k) on the path can be as large as you want!
→ If there are no negative cycles, the shortest path from S to any node should use at most $n - 1$ edges.

Just run reliable shortest path with $k = n - 1$

This is called the Bellman-Ford algorithm.
Runtime of $O(nm)$.

Bellman-Ford Algorithm

Discussion 6 material.

Summary of shortest path algs.

- Breadth First Search

→ Not for weighted graphs.

→ $O(n+m)$

- Dijkstra

→ Positive edge weights.

→ $O(m + n\log(n))$

- Bellman-Ford

→ Positive or negative edge weights, as long as no negative cycles.

→ $O(nm)$

Bellman-Ford1$(G = (V, E), s)$

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

For $i = 1, \dots, n-1$:

For $u \in V$:

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

Bellman-Ford2$(G = (V, E), s)$

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

For $i = 1, \dots, n-1$:

Same as Dijkstra's "Update" Function

For $(v,u) \in E$:

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

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

Next time

- More examples of DP