CS 170
Efficient Algorithms and Intractable Problems

Lecture 12
Dynamic Programming II

Nika Haghtalab and John Wright

EECS, UC Berkeley

Announcements

Midterm 1 grade are out!
→ Midterm regrade requests are open on Monday

Midsemester feedback form is open
→ Fill out by Monday and you'll get a free homework drop.

Discussions and OH resume as usual this week
→ As usual, my OH is after class today. Meet outside of the lecture hall.

Recap of the Last Lecture

Dynamic Programming!

The recipe!

Step 1. Identify subproblems (aka optimal substructure)
Step 2. Find a recursive formulation for the subproblems
Step 3. Design the Dynamic Programming Algorithm
→ Memo-ize computation starting from smallest subproblems and building up.

We saw a lot of examples already
→ Fibonacci
→ Shortest Paths with negative edge-weights (in DAGs, Reliable Shortest path, Bellman-Ford)

This lecture

See more dynamic programming examples!
- → Shortest Path between all pairs
- → Longest increasing subsequence
- → Edit distance
- → And even more next lecture

Best way to learn dynamic programming is by doing a lot of examples!

By doing more examples today, we will also develop intuition about how to choose subproblems (Recipe's step 1).

Recap: 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.

If there are no negative cycles, the shortest path from S to any node should use at most $n - 1$ edges.
→ 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!

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.

→ $O(nm)$

The same implementation as "reliable" shortest path from last lecture

```
Bellman-Ford1($G=(V,E),s$)
$dist[s]=0$ and $dist[u]=\infty$ for all other $u\in V$.
  For $i=1,\ldots,n-1$:
    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 $i=1,\ldots,n-1$:
    For $(v,u)\in E$:
      $dist[u]\leftarrow\min\{dist[u],dist[v]+\ell(v,u)\}$
```

Same as Dijkstra's "Update" Function

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 $O(n^2m)$. 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.

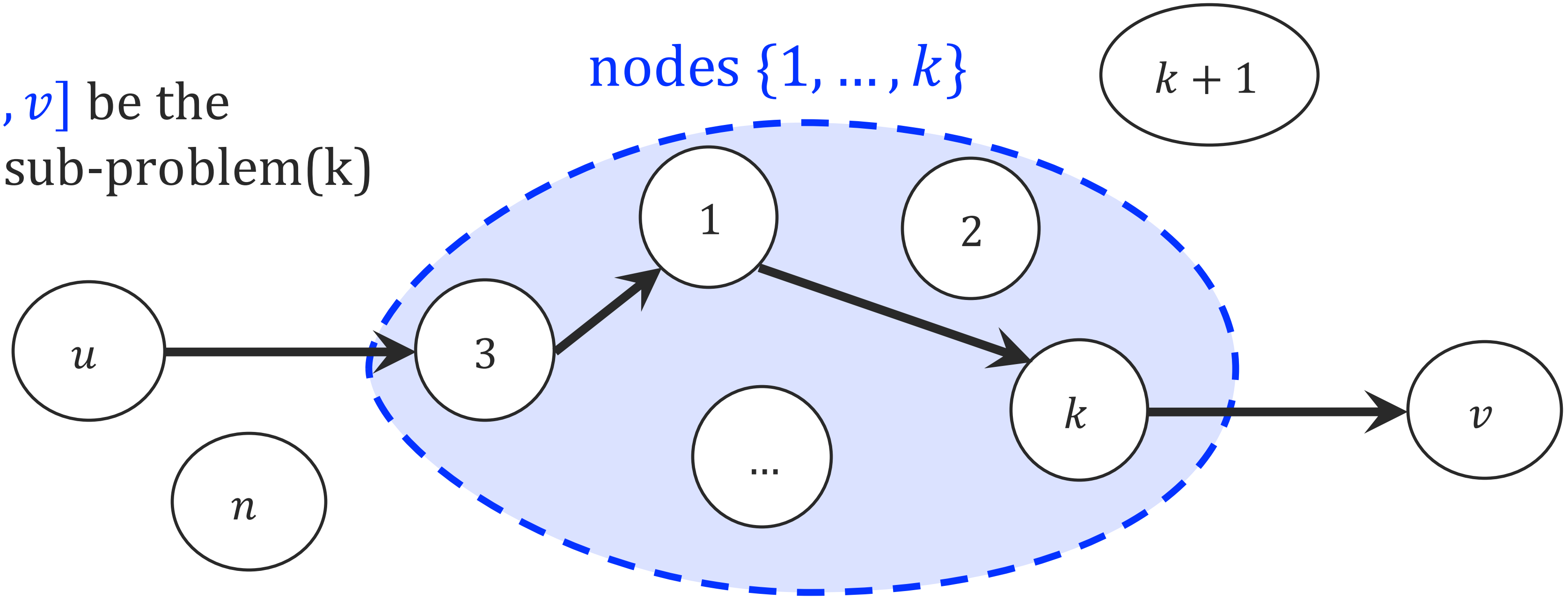
Identify the subproblems (optimal substructure)

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

→ 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)

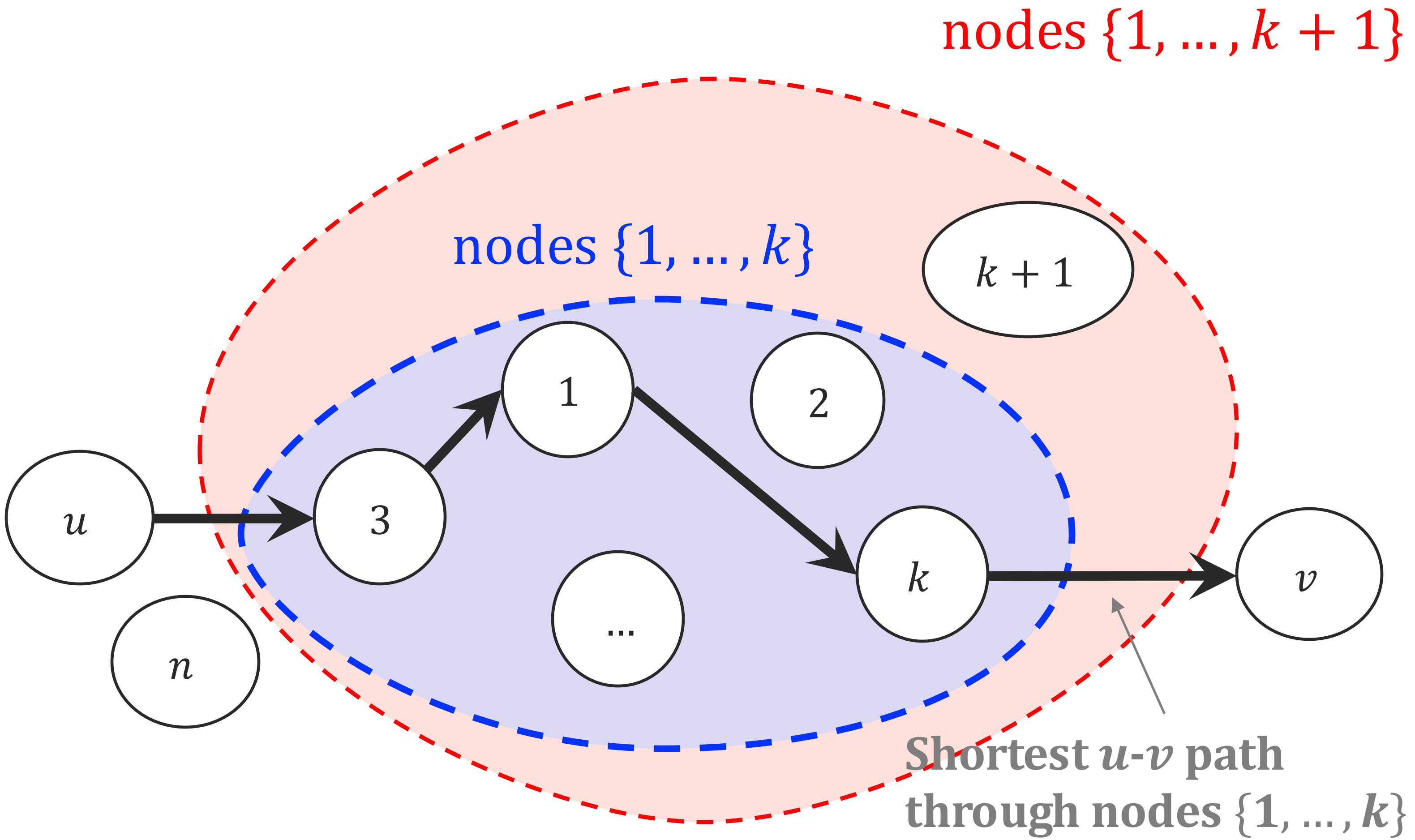
- **Step 2:** Find a recursive formulation for the subproblems

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

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

Recursive Formulation

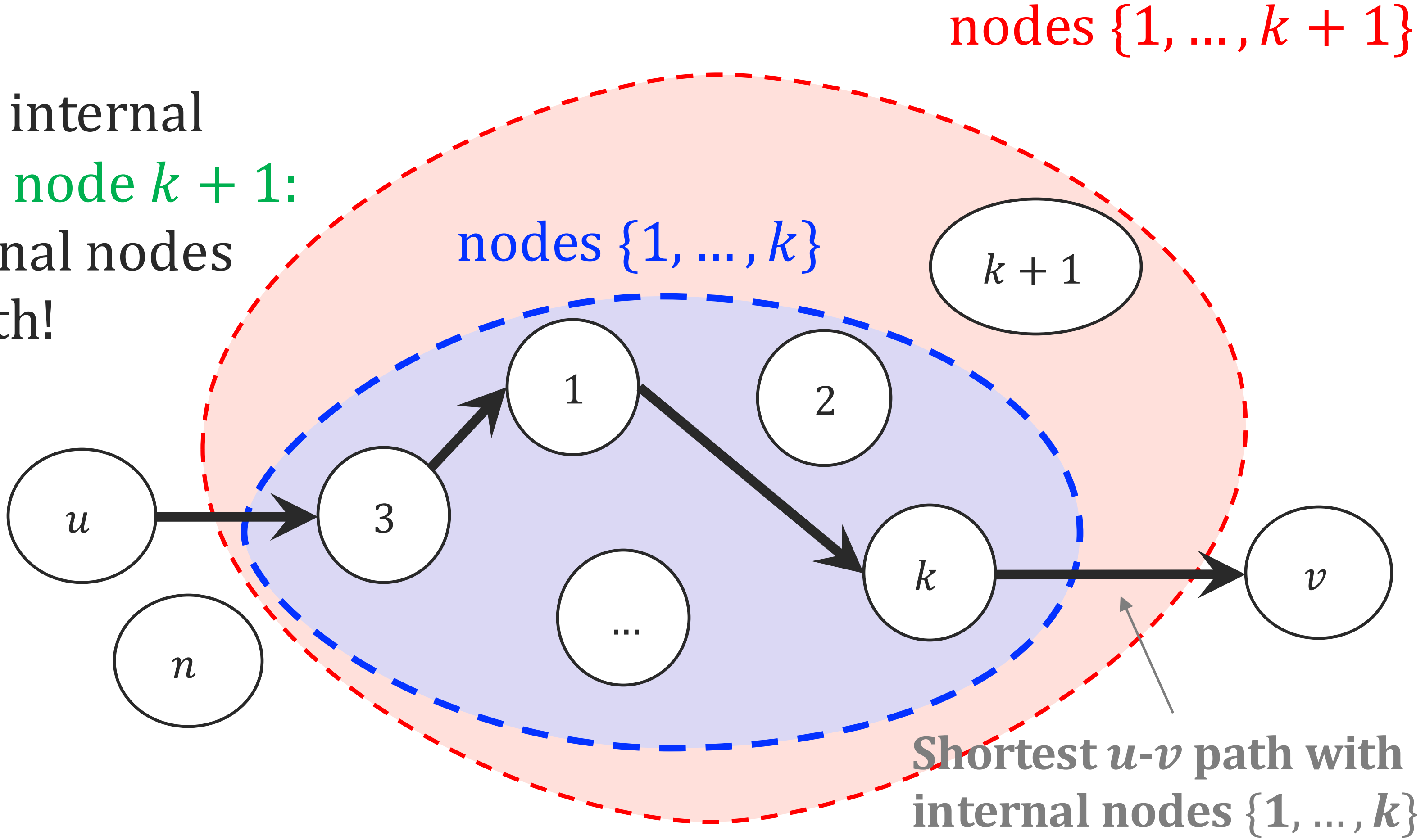
How do I solve **sub-problem(k+1)** knowing all the solutions $dist_k(u, v)$ to **Sub-problem(k)?**



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



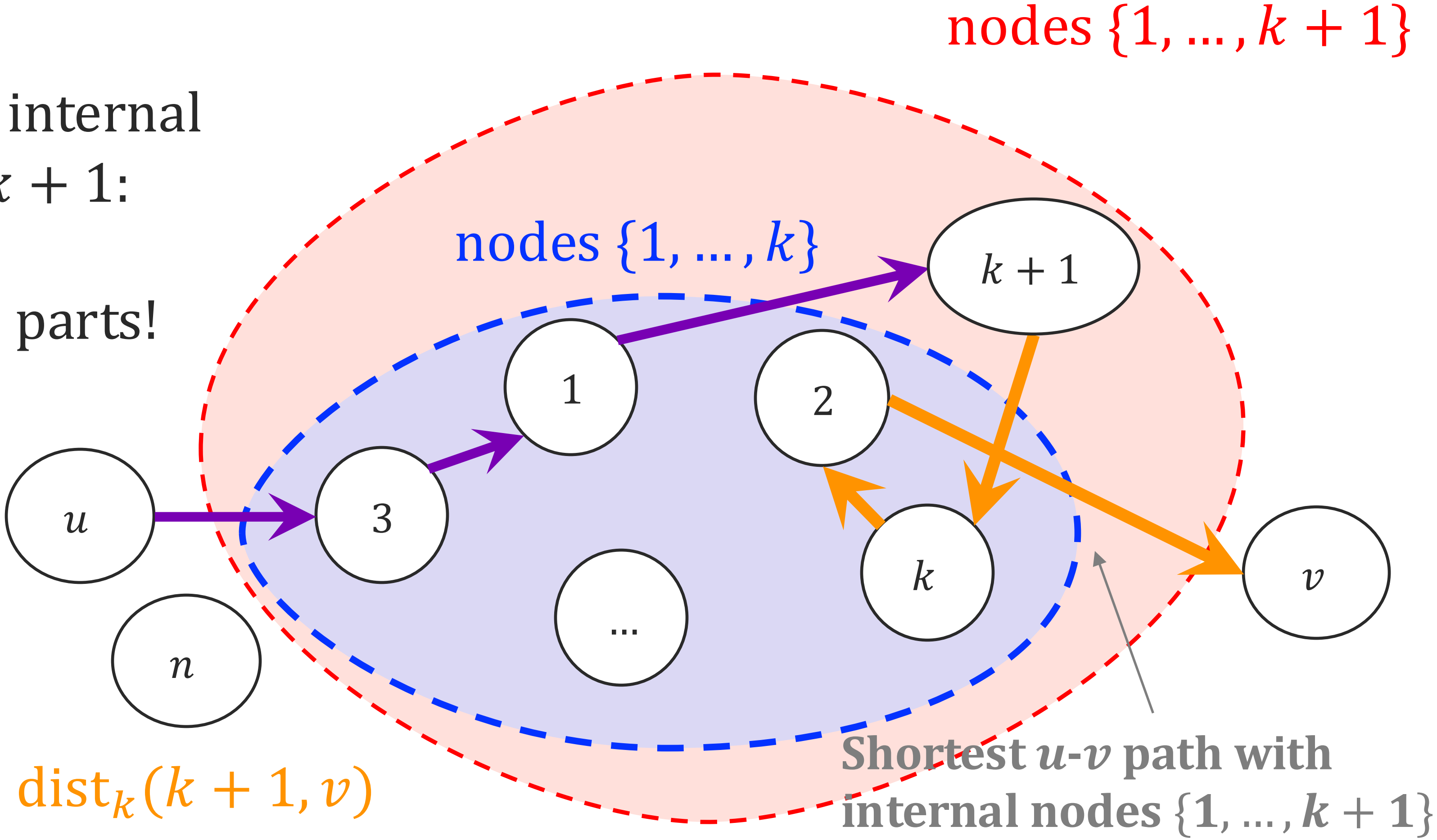
Recursive Formulation

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 with internal nodes $\{1, \dots, k+1\}$ uses node $k+1$:

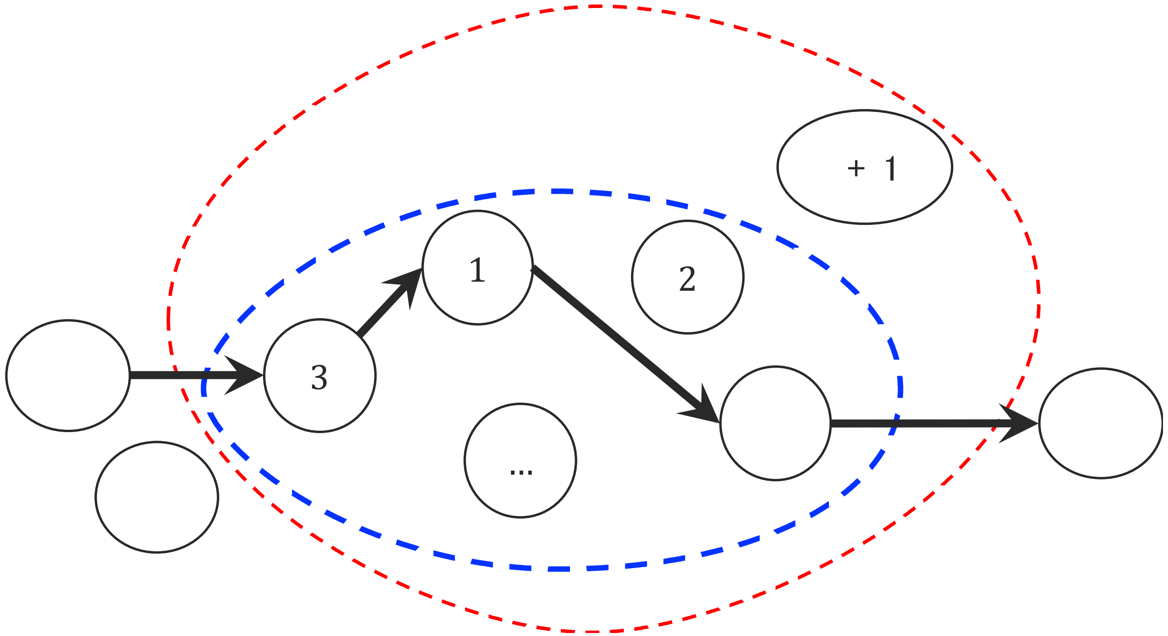
This path can be broken to two parts!

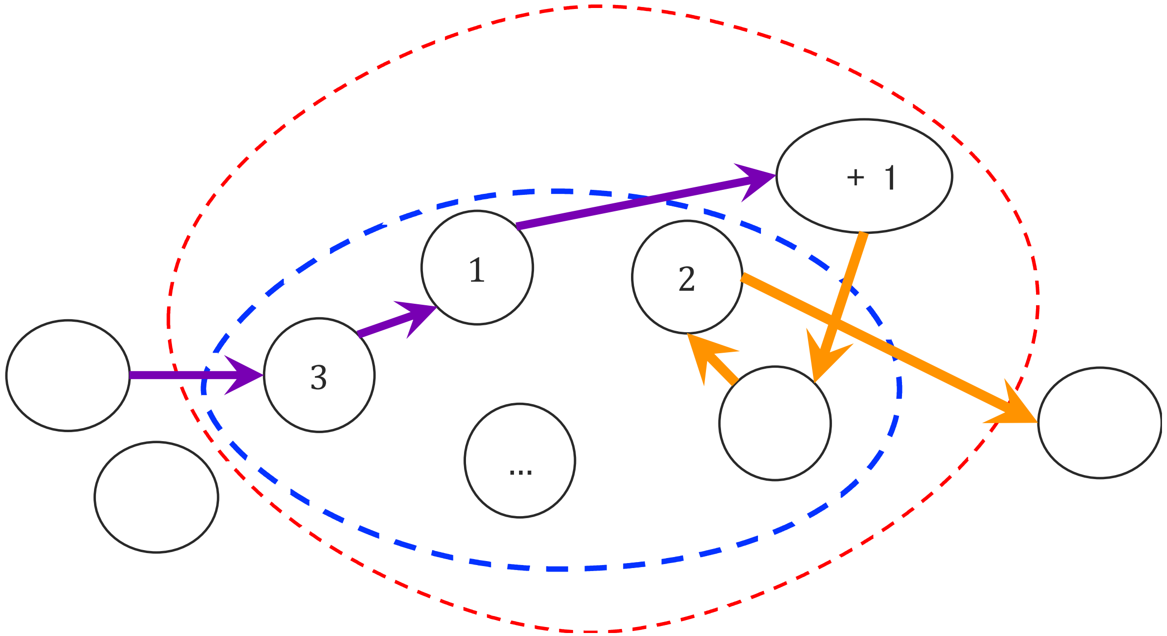
- Shortest u-$(k+1)$ path
- Shortest $(k+1)$-v path
- Both using internal nodes $\{1, \dots, k\}$ only.



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

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 interna nodes $\{1, \ldots, k+1\}$ uses node $k+1$:

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 $\infty$
    For $(u, v) \in E$, $dist_0[u, v] \leftarrow \ell(u, v)$
                    // $dist_0$ paths have no internal nodes.
    For $k = 1, \ldots, n$:
      For $u, v \in V$:
          $dist_k[u, v] \leftarrow \min\{dist_{k-1}[u, v]$ ,
                                   $dist_{k-1}[u, k] + dist_{k-1}[k, v]\}$
```

Longest Increasing Subsequences

Longest Increasing Subsequences (LIS)

To be consistent with the book, we aren't using 0-indexing for the input.

- Input: An array of n integers $a = [a_1, \dots, a_n]$
- Output: The length of the longest increasing subsequence of the input.

"Subsequences" can be non-contiguous by definition.

$$a = 5 \quad 2 \quad 8 \quad 6 \quad 3 \quad 6 \quad 9 \quad 7$$

Longest Increasing Subsequence

Why care about this problem?

- An important algorithmic preprocessing step.
- Useful for understanding random processes.

→Shuffle cards and play the game of Solitaire (aka Patience Sorting), how many piles you need?

→Computations over random graphs, networks, social media.

The recipe!

Step 1. Identify subproblems (aka optimal substructure)
Step 2. Find a recursive formulation for the subproblems
Step 3. Design the Dynamic Programming Algorithm
→ Memo-ize computation starting from smallest subproblems and building up.

Step 1: Subproblems of LIS

- Input: An array of n integers $a = [a_1, \dots, a_n]$
- Output: The length of the longest increasing subsequence (LIS) of the input.

Discuss

Which of these two subproblems is more appropriate for designing a dynamic programming algorithm?

1. $L[j] = len. of\ LIS\ in\ array\ [a_1, \dots, a_j]$, for $j = 1, \dots, n$
2. $L[j] = len. of\ LIS\ in\ array\ [a_1, \dots, a_j]$ that ends in a_j, for $j = 1, \dots, n$

What makes for good subproblems?

- Not too many of them (the more subproblems the slower the DP algorithm)
- Must have enough information in it to compute subproblems recursively (needed for step 2).

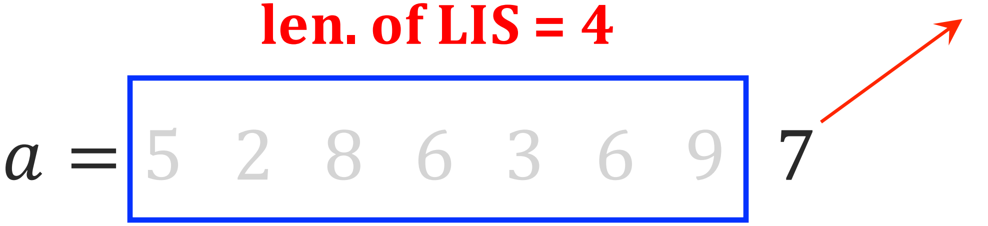
Step 1: Subproblems of LIS

- Input: An array of n integers $a = [a_1, \dots, a_n]$
- Output: The length of the longest increasing subsequence (LIS) of the input.

Subproblems: $L[j] = len.\,of\ LIS\ in\ array\ [a_1, \dots, a_j]$ that ends in a_j, for $j = 1, \dots, n$

→ Because, if we don't keep track of the last (largest) element of the LIS we don't know whether we can add a new element to the subsequence, recursively.

→ Think of the subproblem's stored info as the only thing you observe about smaller instances!


len. of LIS = 4

$a = \boxed{5\ \ 2\ \ 8\ \ 6\ \ 3\ \ 6\ \ 9}\ \ 7$

Knowing only Len of LIS, we wouldn't know if we can add 7

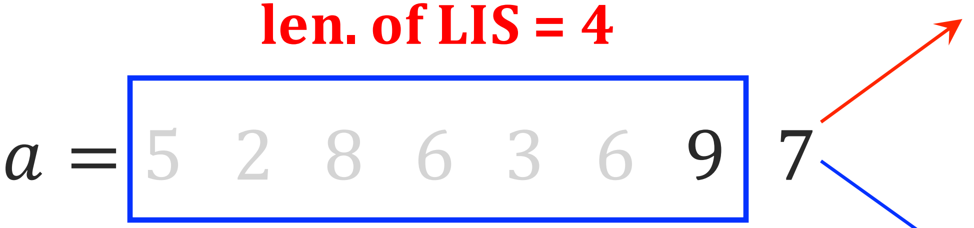
Step 1: Subproblems of LIS

- Input: An array of n integers $a = [a_1, \dots, a_n]$
- Output: The length of the longest increasing subsequence (LIS) of the input.

Subproblems: $L[j] = len.of\ LIS\ in\ array\ [a_1, \dots, a_j]$ that ends in a_j, for $j = 1, \dots, n$

→ Because, if we don't keep track of the last (largest) element of the LIS we don't know whether we can add a new element to the subsequence, recursively.

→ Think of the subproblem's stored info as the only thing you observe about smaller instances!



Knowing only Len of LIS, we wouldn't know if we can add 7

We know for sure, we can't add 7

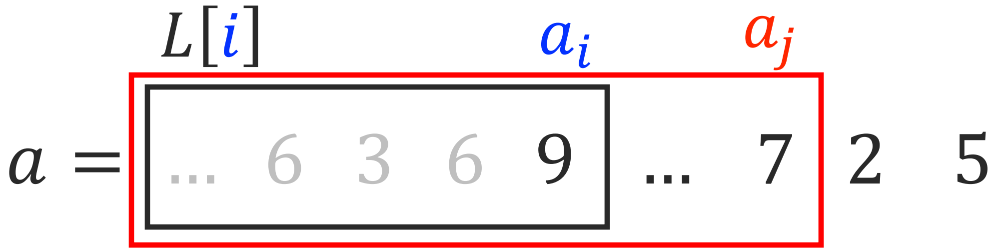
Step 2: Recurrence of LIS subproblems

- <u>Input</u>: An array of n integers $a = [a_1, \dots, a_n]$
- <u>Output</u>: The length of the longest increasing subsequence (LIS) of the input.

Step 1: $L[j] = len.\, of\ LIS\ in\ array\ [a_1, \dots, a_j]$ that ends in a_j

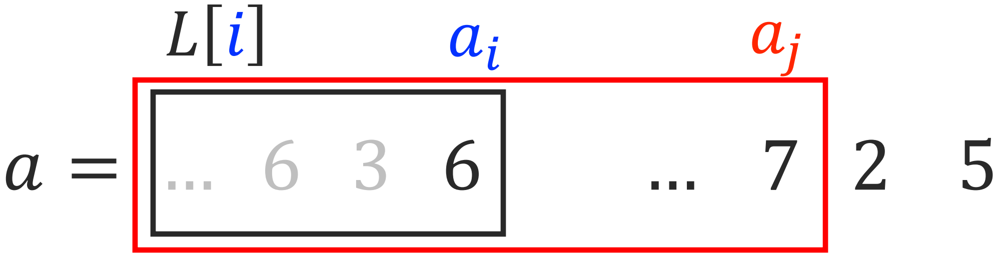
Step 2: Compute the recurrence: $L[j]$ in terms of $L[i]$ for $i < j$.

Case 1: $a[j] \leq a[i]$



Can't add $a[j]$ to lengthen $L[i]$

Case 2: $a[j] > a[i]$



$L[j] = L[i] + 1$

Step 2: Recurrence of LIS subproblems

To be consistent with the book, we aren't using 0-indexing for the input.

- Input: An array of n integers $a = [a_1, \dots, a_n]$
- Output: The length of the longest increasing subsequence (LIS) of the input.

Step 1: $L[j] = len.\,of\ LIS\ in\ array\ [a_1, \dots, a_j]$ that ends in a_j

Discuss

Step 2: Compute the recurrence: $L[j]$ in terms of $L[i]$ for $i < j$.

Step 3: Design the DP Algorithm

- Input: An array of n integers $a = [a_1, \ldots, a_n]$
- Output: The length of the longest increasing subsequence (LIS) of the input.

To be consistent with the book, we aren't using 0-indexing for the input.

Subproblems: $L[j] = len. of\ LIS\ in\ array$ $[a_1, \ldots, a_j]$ that ends in a_j

Runtime:
$O(n)$ subproblems

For each subproblem, we look at at most n smaller subproblems.
→ $O(n)$ time per subproblem.

Total: $O(n^2)$ runtime.

```
LIS($a_1, \ldots a_n$)
    array $L$ of length $n$
    for $j = 1, \ldots, n$
        If exists $i < j$, s.t., $a_i < a_j$
            $L[j] \leftarrow 1 + \max_{i<j}\{ L[i] \mid a_i < a_j \}$
        Else $L[j] \leftarrow 1$
    return $max_i\ L[i]$
```

Edit Distance

Computing the Edit Distance

Input: Two strings $S[1 \ldots m]$ and $T[1 \ \ldots \ n]$

Output: Compute the smallest number of edits to turn S into T.

Edits allowed:

1. Insert a character into S
2. Delete character from S
3. Change one character to another character.

Example:
What's the edit distance between
$S =$ "$SNOWY$" and $T =$ "$SUNNY$"?

S N O W Y

Add U

S U N O W Y

Change O to N

S U N N W Y

Delete W

S U N N Y

Applications of Edit Distance

- Auto correct!
- Word suggestions in search engines
- DNA analysis of similarities.

Edit Distance and Cost of Alignment

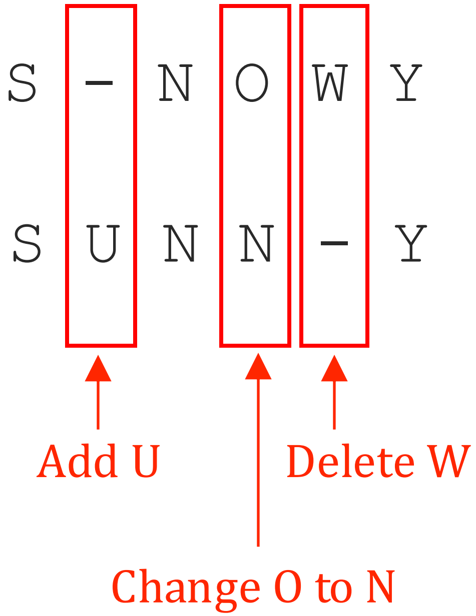
Input: Two strings $S[1 \dots m]$ and $T[1 \ \dots \ n]$

Output: Compute the smallest number of edits to turn S into T.

Edit Distance is the minimal cost of alignment between two strings.

→ An alignment: line up two words. Cost of an alignment = # columns that don't match

An alignment with cost 3



S – N O W Y

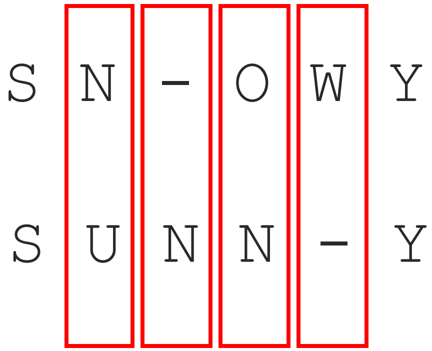
S U N N – Y

Add U

Delete W

Change O to N

An alignment with cost 4



S N – O W Y

S U N N – Y

Step 1: Subproblems of Edit Distance

Input: Two strings $S[1 \ldots m]$ and $T[1 \ \ldots \ n]$

Output: Compute the smallest number of edits to turn S into T.

What makes for good subproblems?

- Not too many of them (the more subproblems the slower the DP algorithm)
- Must have enough information in it to compute subproblems recursively (needed for step 2).

Subproblems: for all $0 \leq i \leq m$ and $0 \leq j \leq n$

$$E(i, j) = EditDist(S[1 \ldots i], T[1 \ldots j])$$

Cost of optimal alignment between $S[1 \ldots i], T[1 \ldots j]$

Step 2: Recurrence Relation of Edit Distance

Input: Two strings $S[1 \ldots m]$ and $T[1 \ldots n]$

Output: Compute the smallest number of edits to turn S into T.

Step 1: $E(i, j) = EditDist(S[1 \ldots i], T[1 \ldots j])$, for all $0 \leq i \leq m$ and $0 \leq j \leq n$

Discuss

There are three ways of aligning $S[1 \ldots i]$ and $T[1 \ldots j]$. What are their costs recursively?

Case 1

$S[1 \cdots i-1]$	$S[i]$
$T[1 \cdots j]$	$-$

Case 2

$S[1 \cdots i]$	$-$
$T[1 \cdots j-1]$	$T[j]$

Case 3

$S[1 \cdots i-1]$	$S[i]$
$T[1 \cdots j-1]$	$T[j]$

$E(i, j) =$

Step 2: Recurrence Relation of Edit Distance

Input: Two strings $S[1 \ldots m]$ and $T[1 \ \ldots \ n]$

Output: Compute the smallest number of edits to turn S into T.

Step 1: $E(i, j) = EditDist(S[1 \ldots i], T[1 \ldots j])$, for all $0 \le i \le m$ and $0 \le j \le n$

Step 2: The recurrence relation

$$E(i, j) = \min \{E(i-1, j) + 1, E(i, j-1) + 1, E(i-1, j-1) + 1(S[i] \neq T[j])\}$$

Base case: $E(i, 0) = i$ and $E(0, j) = j$

Step 3: Design the Algorithm

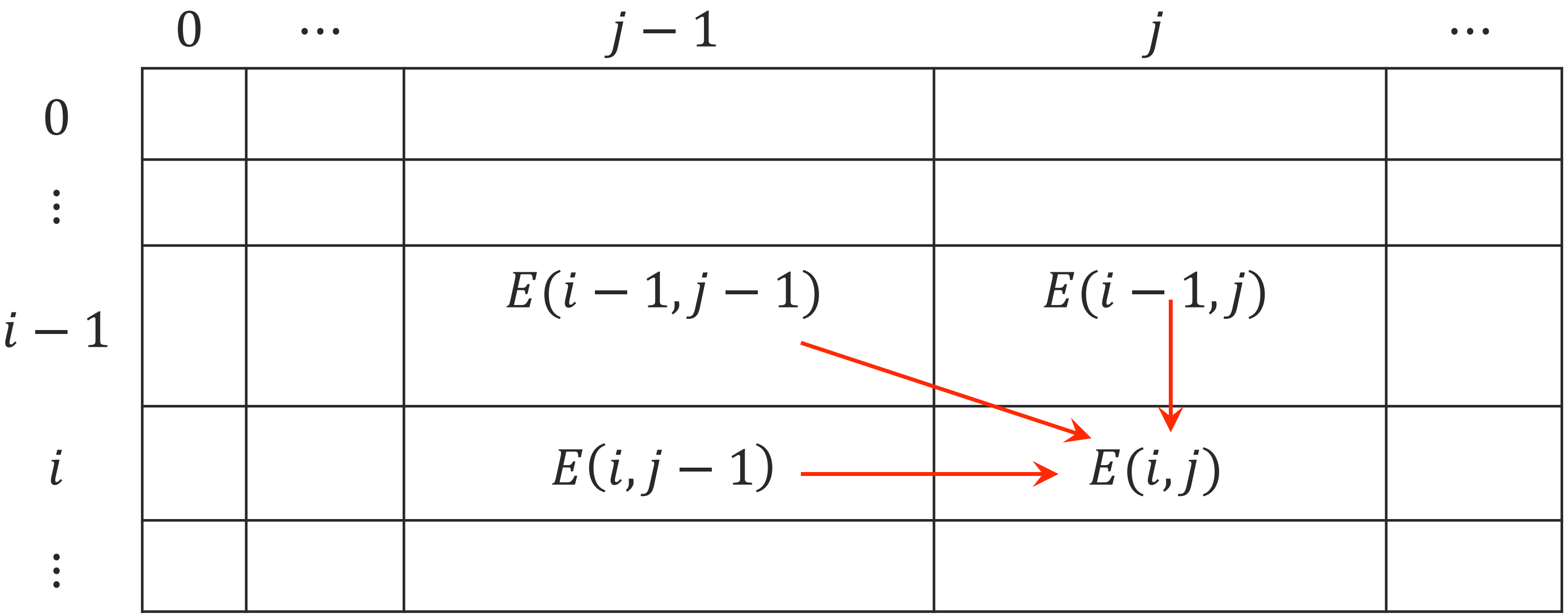
Input: Two strings $S[1 \dots m]$ and $T[1 \ \dots \ n]$

Output: Compute the smallest number of edits to turn S into T.

How do we memo-ize the subproblems in this recurrence relation?

$$E(i, j) = \min\{E(i-1, j) + 1, E(i, j-1) + 1, E(i-1, j-1) + 1(S[i] \neq T[j])\}$$

Base case: $E(i, 0) = i$ and $E(0, j) = j$



Step 3: Design the Algorithm

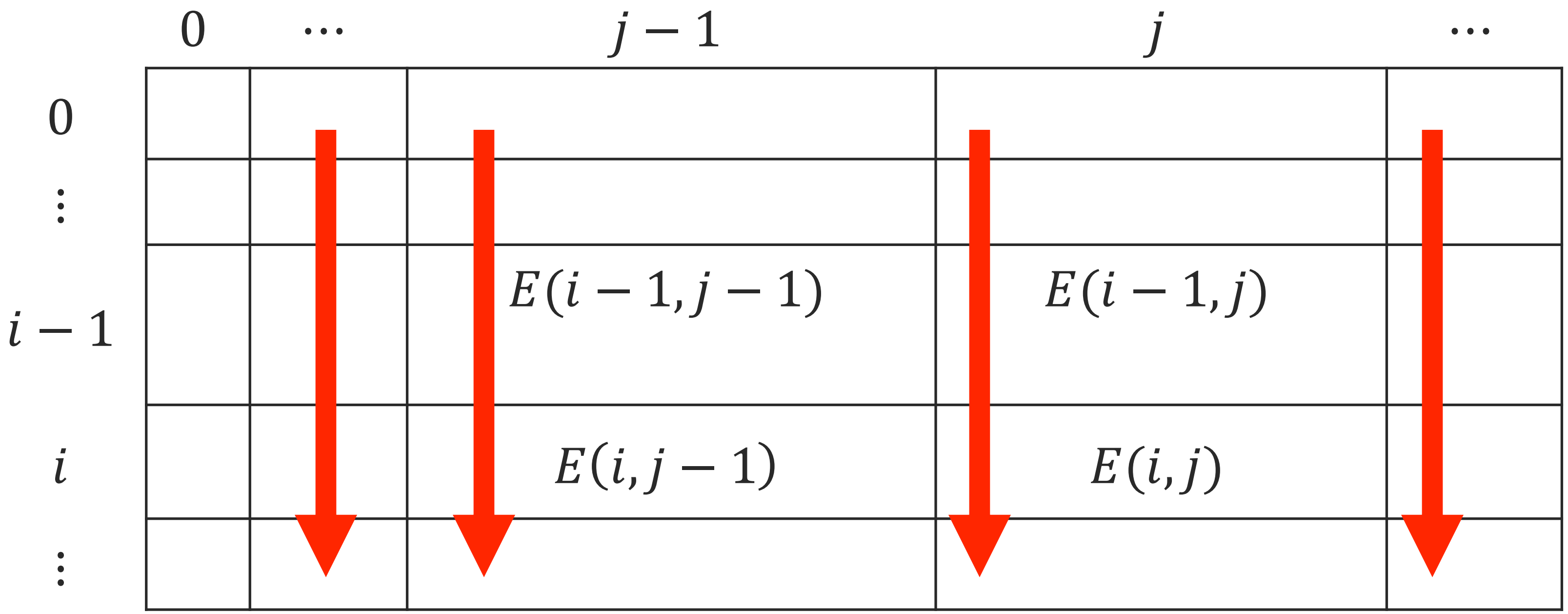
Input: Two strings $S[1 \dots m]$ and $T[1 \dots n]$

Output: Compute the smallest number of edits to turn S into T.

How do we memo-ize the subproblems in this recurrence relation?

$$E(i,j) = \min\{E(i-1,j)+1, E(i,j-1)+1, E(i-1,j-1)+1(S[i] \neq T[j])\}$$

Base case: $E(i,0) = i$ and $E(0,j) = j$



Step 3: Design the Algorithm

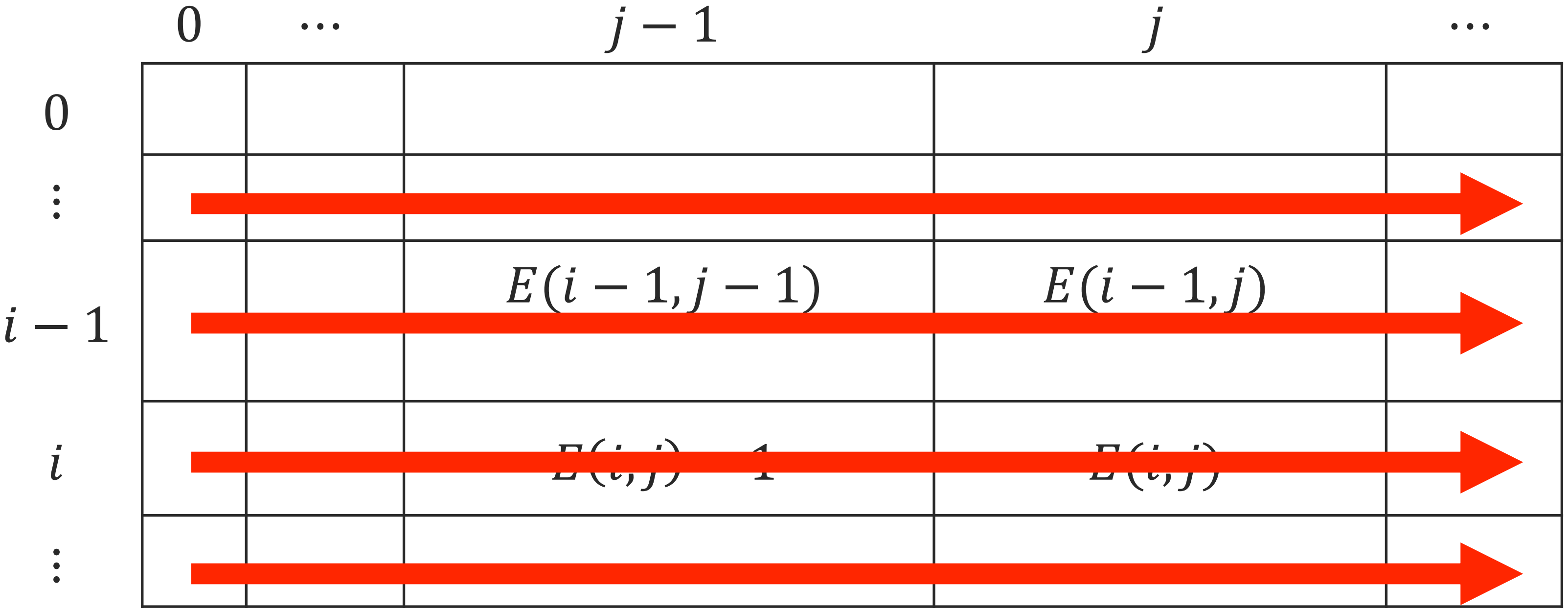
Input: Two strings $S[1 \dots m]$ and $T[1 \dots n]$

Output: Compute the smallest number of edits to turn S into T.

How do we memo-ize the subproblems in this recurrence relation?

$$E(i,j) = \min\{E(i-1,j)+1, E(i,j-1)+1, E(i-1,j-1)+1(S[i] \neq T[j])\}$$

Base case: $E(i,0) = i$ and $E(0,j) = j$



Step 3: Design the Algorithm

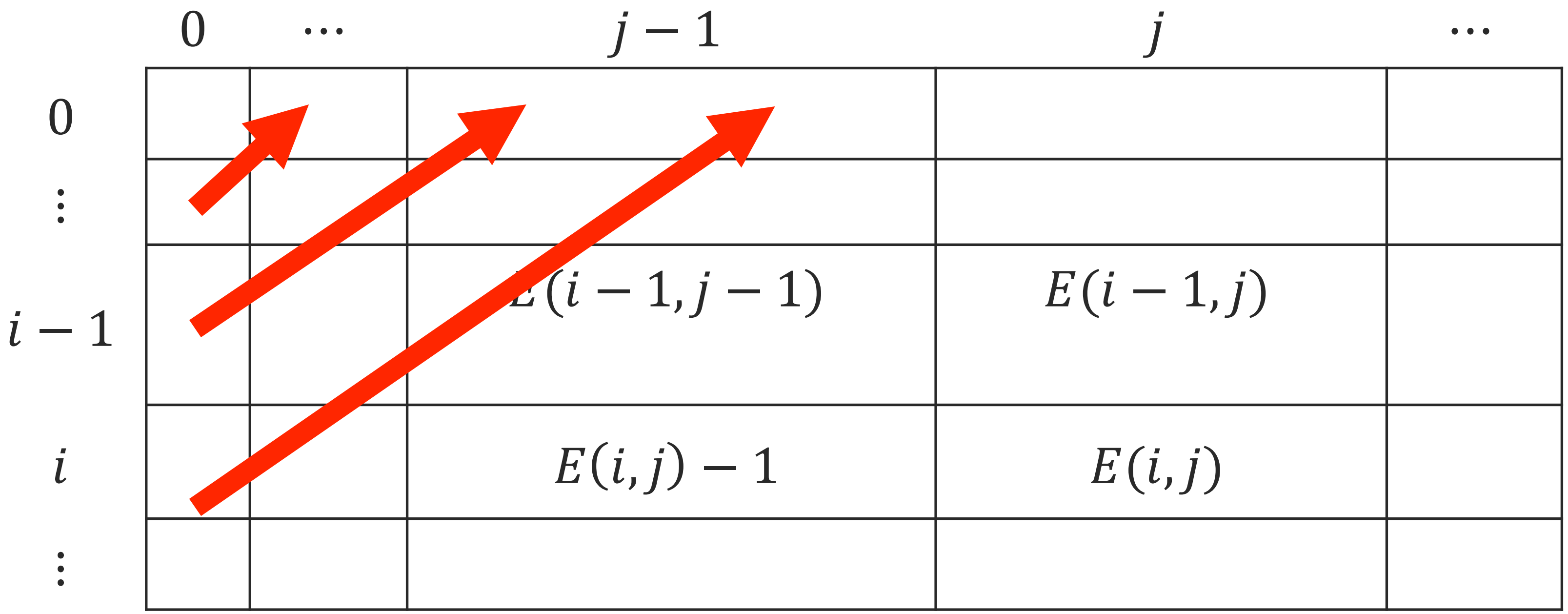
Input: Two strings $S[1 \ldots m]$ and $T[1 \ \ldots \ n]$

Output: Compute the smallest number of edits to turn S into T.

How do we memo-ize the subproblems in this recurrence relation?

$$E(i,j) = \min\{E(i-1,j)+1, E(i,j-1)+1, E(i-1,j-1)+1(S[i] \neq T[j])\}$$

Base case: $E(i,0) = i$ and $E(0,j) = j$



	0	$\cdots$	$j-1$	j	$\cdots$
0					
$\vdots$					
$i-1$			$E(i-1,j-1)$	$E(i-1,j)$	
i			$E(i,j)-1$	$E(i,j)$	
$\vdots$					

Runtime of this algorithm

Input: Two strings $S[1 \dots m]$ and $T[1 \ \dots \ n]$

Output: Compute the smallest number of edits to turn S into T.

$O(mn)$ number of subproblems.

For each subproblem, we take minimum of 3 values.
→ Work per subproblem $O(1)$

Total runtime: $O(mn)$.

```
Edit-Distance($S[1 \dots m], T[1 \cdots n]$)
    $(m+1) \times (n+1)$ array $E$
    **For** $i = 0, 1, \dots, m$, $E[i, 0] = i$
    **For** $j = 0, 1, \dots, n$, $E[0, j] = j$
    **For** $i = 1, \dots, m$
        **For** $j = 1, \dots, n$
            $\mathrm{E}(i, j) \leftarrow \min \left\{ \begin{array}{c} E(i-1, j) + 1, \\ E(i, j-1) + 1, \\ E(i-1, j-1) + 1(S[i] \neq T[j]) \end{array} \right\}$
    **return** $\mathrm{E}(m, n)$
```

Wrap up

More examples of dynamic programming.

- Longest increasing subsequence
- Edit distance
- Knapsack (with repetition)

→ Also got more experience on how to choose subproblems.

Next time: More examples of DP

Knapsack and other graph problems