What is the running time of the following algorithm:

Consider computing \( f(x, y) \) by recursive function + memoization.

\[
f(x, y) = \sum_{i=1}^{\min(x,y)} x \times f(x + y - i, i - 1),
\]

\[
f(0, y) = y \quad f(x, 0) = x.
\]

The resulting algorithm when computing \( f(n, n) \) would take:

(a) \( O(n^2) \)
(b) \( O(n^3) \)
(c) \( O(2^n) \)
(d) \( O(n^n) \)
(e) The function is ill defined - it can not be computed.
Pre-lecture brain teaser

What is the running time of the following algorithm:
Consider computing $f(x, y)$ by recursive function + memoization.

$$
\begin{align*}
f(x, y) &= \sum_{i=1}^{\min(x,y)} x \cdot f(x+y-i, i-1), \\
f(0, y) &= y \\
f(x, 0) &= x.
\end{align*}
$$

The resulting algorithm when computing $f(n, n)$ would take:

(a) $O(n^2)$  
(b) $O(n^3)$  
(c) $O(2^n)$  
(d) $O(n^n)$  
(e) The function is ill defined - it can not be computed.
Recipe for Dynamic Programming

1. Develop a recursive backtracking style algorithm $A$ for given problem.
2. Identify structure of subproblems generated by $A$ on an instance $I$ of size $n$
   2.1 Estimate number of different subproblems generated as a function of $n$. Is it polynomial or exponential in $n$?
   2.2 If the number of problems is “small” (polynomial) then they typically have some “clean” structure.
3. Rewrite subproblems in a compact fashion.
4. Rewrite recursive algorithm in terms of notation for subproblems.
5. Convert to iterative algorithm by bottom up evaluation in an appropriate order.
6. Optimize further with data structures and/or additional ideas.
Dynamic programming was a technique “invented” by Richard Bellman. From his autobiography:

I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is, Where did the name, dynamic programming, come from? The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. I’m not using the term lightly; I’m using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word “programming”. I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying I thought, lets kill two birds with one stone. Lets take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it’s impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It’s impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.
Edit Distance and Sequence Alignment
Spell Checking Problem

Given a string “exponen” that is not in the dictionary, how should a spell checker suggest a nearby string?
Spell Checking Problem

Given a string “exponen” that is not in the dictionary, how should a spell checker suggest a nearby string?

What does nearness mean?

**Question:** Given two strings $x_1x_2 \ldots x_n$ and $y_1y_2 \ldots y_m$ what is a distance between them?
Given a string “exponen” that is not in the dictionary, how should a spell checker suggest a nearby string?

What does nearness mean?

**Question:** Given two strings $x_1x_2\ldots x_n$ and $y_1y_2\ldots y_m$ what is a distance between them?

**Edit Distance:** minimum number of “edits” to transform $x$ into $y$. 
Definition

Edit distance between two words X and Y is the number of letter insertions, letter deletions and letter substitutions required to obtain Y from X.

Example

The edit distance between FOOD and MONEY is at least 4:

FOOD → MOOD → MONOD → MONED → MONEY
Alignment
Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

F O O D
M O N E Y
Alignment
Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

\[
\begin{array}{cccccc}
F & O & O & D \\
M & O & N & E & Y
\end{array}
\]

Formally, an alignment is a set \( M \) of pairs \((i, j)\) such that each index appears at most once, and there is no “crossing”: \( i < i' \) and \( i \) is matched to \( j \) implies \( i' \) is matched to \( j' > j \). In the above example, this is \( M = \{(1, 1), (2, 2), (3, 3), (4, 5)\} \).
Alignment
Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

\[
\begin{align*}
\text{FOOD} \\
\text{MONEY}
\end{align*}
\]

Formally, an alignment is a set \( M \) of pairs \((i, j)\) such that each index appears at most once, and there is no “crossing”: \( i < i' \) and \( i \) is matched to \( j \) implies \( i' \) is matched to \( j' > j \). In the above example, this is \( M = \{(1, 1), (2, 2), (3, 3), (4, 5)\} \). Cost of an alignment is the number of mismatched columns plus number of unmatched indices in both strings.
Problem
Given two words, find the edit distance between them, i.e., an alignment of smallest cost.
Applications

- Spell-checkers and Dictionaries
- Unix `diff`
- DNA sequence alignment ... but, we need a new metric
Sequence alignment problem - Similarity Metric

**Definition**
For two strings $X$ and $Y$, the cost of alignment $M$ is

- **[Gap penalty]** For each gap in the alignment, we incur a cost $\delta$.
- **[Mismatch cost]** For each pair $p$ and $q$ that have been matched in $M$, we incur cost $\alpha_{pq}$; typically $\alpha_{pp} = 0$. 

Edit distance is a special case when $\delta = \alpha_{pq} = 1$. 

Definition
For two strings $X$ and $Y$, the cost of alignment $M$ is

- **[Gap penalty]** For each gap in the alignment, we incur a cost $\delta$.
- **[Mismatch cost]** For each pair $p$ and $q$ that have been matched in $M$, we incur cost $\alpha_{pq}$; typically $\alpha_{pp} = 0$.

Edit distance is a special case when $\delta = \alpha_{pq} = 1$. 
Edit distance as alignment
Example

| o | c | u | r | r | a | n | c | e |
| o | c | c | u | r | r | e | n | c | e |

Cost = $\delta + \alpha_{ae}$

Alternative:

| o | c | u | r | r | a | n | c | e |
| o | c | c | u | r | r | e | n | c | e |

Cost = $3\delta$

Or a really stupid solution (delete string, insert other string):

| o | c | u | r | r | a | n | c | e |
| o | c | c | u | r | r | e | n | c | e |

Cost = $19\delta$. 
What is the minimum edit distance for the following two strings, if insertion/deletion/change of a single character cost 1 unit?

374
473

(a) 1
(b) 2
(c) 3
(d) 4
(e) 5
What is the minimum edit distance for the following two strings, if insertion/deletion/change of a single character cost 1 unit?

(a) 1
(b) 2
(c) 3
(d) 4
(e) 5
What is the minimum edit distance for the following two strings, if insertion/deletion/change of a single character cost 1 unit?

1. 1
2. 2
3. 3
4. 4
5. 5
Sequence Alignment

**Input** Given two words $X$ and $Y$, and gap penalty $\delta$ and mismatch costs $\alpha_{pq}$

**Goal** Find alignment of minimum cost
Edit distance: The algorithm
Let $X = \alpha x$ and $Y = \beta y$

$\alpha, \beta$: strings.

$x$ and $y$ single characters.

Think about optimal edit distance between $X$ and $Y$ as alignment, and consider last column of alignment of the two strings:

- $\alpha$  |  $x$
  - $\beta$  |  $y$

- $\alpha$  |  $x$
  - $\beta y$

- $\alpha x$  |  
  - $\beta$  |  $y$

Prefixes must have optimal alignment!
Let $X = x_1x_2 \cdots x_m$ and $Y = y_1y_2 \cdots y_n$. If $(m, n)$ are not matched then either the $m^{th}$ position of $X$ remains unmatched or the $n^{th}$ position of $Y$ remains unmatched.

- **Case** $x_m$ and $y_n$ are matched.
  - Pay mismatch cost $\alpha_{x_my_n}$ plus cost of aligning strings $x_1 \cdots x_{m-1}$ and $y_1 \cdots y_{n-1}$

- **Case** $x_m$ is unmatched.
  - Pay gap penalty plus cost of aligning $x_1 \cdots x_{m-1}$ and $y_1 \cdots y_n$

- **Case** $y_n$ is unmatched.
  - Pay gap penalty plus cost of aligning $x_1 \cdots x_m$ and $y_1 \cdots y_{n-1}$
Subproblems and Recurrence

| $x_1 \ldots x_{i-1}$ | $x_i$  | or | $x_1 \ldots x_{i-1}$ | $x$   | or | $x_1 \ldots x_{i-1}x_i$ | $y_1 \ldots y_{j-1}$ | $y_j$ |
|----------------------|-------|    |----------------------|-------|    |----------------------|----------------------|-------|
| $y_1 \ldots y_{j-1}$ | $y_j$ |    | $y_1 \ldots y_{j-1}$ | $y_j$ |    |

Optimal Costs
Let $\text{Opt}(i, j)$ be optimal cost of aligning $x_1 \cdots x_i$ and $y_1 \cdots y_j$. Then

$$
\text{Opt}(i, j) = \min \begin{cases} 
\alpha_{x_i y_j} + \text{Opt}(i - 1, j - 1), \\
\delta + \text{Opt}(i - 1, j), \\
\delta + \text{Opt}(i, j - 1)
\end{cases}
$$
Subproblems and Recurrence

<table>
<thead>
<tr>
<th>$x_1 \ldots x_{i-1}$</th>
<th>$x_i$</th>
<th>or</th>
<th>$x_1 \ldots x_{i-1}$</th>
<th>$x$</th>
<th>or</th>
<th>$x_1 \ldots x_{i-1}x_i$</th>
<th>$x_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_1 \ldots y_{j-1}$</td>
<td>$y_j$</td>
<td></td>
<td>$y_1 \ldots y_{j-1}$</td>
<td>$y_j$</td>
<td></td>
<td>$y_1 \ldots y_{j-1}y_j$</td>
<td>$y_j$</td>
</tr>
</tbody>
</table>

Optimal Costs
Let $\text{Opt}(i, j)$ be optimal cost of aligning $x_1 \ldots x_i$ and $y_1 \ldots y_j$. Then

$$\text{Opt}(i, j) = \min \begin{cases} 
\alpha_{x_iy_j} + \text{Opt}(i - 1, j - 1), \\
\delta + \text{Opt}(i - 1, j), \\
\delta + \text{Opt}(i, j - 1) 
\end{cases}$$

Base Cases: $\text{Opt}(i, 0) = \delta \cdot i$ and $\text{Opt}(0, j) = \delta \cdot j$
Recursive Algorithm

Assume $X$ is stored in array $A[1..m]$ and $Y$ is stored in $B[1..n]$.
Array $COST$ stores cost of matching two chars. Thus $COST[a, b]$ give the cost of matching character $a$ to character $b$.

\[
EDIST(A[1..m], B[1..n])
\]

- If ($m = 0$) return $n \delta$
- If ($n = 0$) return $m \delta$
- $m_1 = \delta + EDIST(A[1..(m - 1)], B[1..n])$
- $m_2 = \delta + EDIST(A[1..m], B[1..(n - 1)])$
- $m_3 = COST[A[m], B[n]] + EDIST(A[1..(m - 1)], B[1..(n - 1)])$

return min($m_1, m_2, m_3$)
**Example: DEED and DREAD**

### Table

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>D</th>
<th>R</th>
<th>E</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Optimal Path Calculation

$$\text{Opt}(i, j) = \min \left\{ \alpha_{x_iy_j} + \text{Opt}(i - 1, j - 1), \right.$$

$$\delta + \text{Opt}(i - 1, j),$$

$$\delta + \text{Opt}(i, j - 1) \right\}$$

### Base Cases:

- $\text{Opt}(i, 0) = \delta \cdot i$
- $\text{Opt}(0, j) = \delta \cdot j$
Example: DEED and DREAD

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>D</th>
<th>R</th>
<th>E</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{Opt}(i, j) =
\begin{cases}
\alpha x_i y_j + \text{Opt}(i-1, j-1), \\
\min \{ \delta + \text{Opt}(i-1, j), \delta + \text{Opt}(i, j-1) \}
\end{cases}
\]

**Base Cases:**
- $\text{Opt}(i, 0) = \delta \cdot i$
- $\text{Opt}(0, j) = \delta \cdot j$
**Example: DEED and DREAD**

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>$D$</th>
<th>$R$</th>
<th>$E$</th>
<th>$A$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>$D$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>$E$</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$D$</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$$
\text{Opt}(i, j) = \min \begin{cases} 
\alpha_{x_iy_j} + \text{Opt}(i-1, j-1), \\
\delta + \text{Opt}(i-1, j), \\
\delta + \text{Opt}(i, j-1)
\end{cases}
$$

**Base Cases:**
- $\text{Opt}(i, 0) = \delta \cdot i$
- $\text{Opt}(0, j) = \delta \cdot j$
Example: DEED and DREAD

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>D</th>
<th>R</th>
<th>E</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$$\text{Opt}(i, j) = \min \begin{cases} 
\alpha_{x_iy_j} + \text{Opt}(i - 1, j - 1), \\
\delta + \text{Opt}(i - 1, j), \\
\delta + \text{Opt}(i, j - 1) 
\end{cases}$$

Base Cases:
- $\text{Opt}(i, 0) = \delta \cdot i$
- $\text{Opt}(0, j) = \delta \cdot j$
Example: DEED and DREAD

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>D</th>
<th>R</th>
<th>E</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Opt\((i, j) = \min \left\{ \begin{array}{c}
\alpha_{x_iy_j} + \text{Opt}(i - 1, j - 1), \\
\delta + \text{Opt}(i - 1, j), \\
\delta + \text{Opt}(i, j - 1)
\end{array} \right\}

Base Cases:
- Opt\((i, 0) = \delta \cdot i
- Opt\((0, j) = \delta \cdot j
Example: DEED and DREAD

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>D</th>
<th>R</th>
<th>E</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

D | R | E | A | D
D | E | E | D | D
### Example: DEED and DREAD

<table>
<thead>
<tr>
<th></th>
<th>ɛ</th>
<th>D</th>
<th>R</th>
<th>E</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>ɛ</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
### Example: DEED and DREAD

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>D</th>
<th>R</th>
<th>E</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

- **DEED**: D E E D
- **DREAD**: D R E A D

![Graph](Image)

20
Example: DEED and DREAD

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>D</th>
<th>R</th>
<th>E</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>$D$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>$E$</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$E$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$D$</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th>R</th>
<th>E</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>E</td>
<td>E</td>
<td>A</td>
<td>D</td>
</tr>
</tbody>
</table>
Dynamic programming algorithm for edit-distance
The cost of aligning a character against another character
\( \Sigma: \) Alphabet

We are given a cost function (in a table):

\[
\forall b, c \in \Sigma \quad \text{COST}[b][c] = \text{cost of aligning } b \text{ with } c.
\]

\[
\forall b \in \Sigma \quad \text{COST}[b][b] = 0
\]

\( \delta \): price of deletion of insertion of a single character
Dynamic program for edit distance

\[
EDIST(A[1..m], B[1..n])
\]

\[
\text{int } M[0..m][0..n]
\]

\[
\text{for } i = 1 \text{ to } m \text{ do } M[i, 0] = i\delta
\]

\[
\text{for } j = 1 \text{ to } n \text{ do } M[0, j] = j\delta
\]

\[
\text{for } i = 1 \text{ to } m \text{ do }
\]

\[
\text{for } j = 1 \text{ to } n \text{ do }
\]

\[
M[i][j] = \min \left\{ \begin{array}{l}
\text{COST}[A[i]][B[j]] + M[i - 1][j - 1], \\
\delta + M[i - 1][j], \\
\delta + M[i][j - 1]
\end{array} \right\}
\]

Analysis

• Running time is

• Space used is
Dynamic program for edit distance

\[ EDIST(A[1..m], B[1..n]) \]

\[
\begin{align*}
    \text{int} & \quad M[0..m][0..n] \\
    \text{for } i = 1 \text{ to } m \text{ do } M[i, 0] &= i \delta \\
    \text{for } j = 1 \text{ to } n \text{ do } M[0, j] &= j \delta \\
    \text{for } i = 1 \text{ to } m \text{ do } \\
        \text{for } j = 1 \text{ to } n \text{ do } & \\
        M[i][j] = \min\{ \\
            \underbrace{COST[A[i]][B[j]] + M[i - 1][j - 1]}_{\text{cost of insert}}, \\
            \delta + M[i - 1][j], \\
            \underbrace{\delta + M[i][j - 1]}_{\text{cost of delete}} \}
\end{align*}
\]

Analysis

- Running time is
- Space used is
Reducing space for edit distance
Matrix and DAG of computation of edit distance

*Figure 1:* Iterative algorithm in previous slide computes values in row order.
Optimizing Space

• Recall

\[ M(i, j) = \min \left\{ \begin{array}{l}
\alpha_{x_i,y_j} + M(i-1, j-1), \\
\delta + M(i-1, j), \\
\delta + M(i, j-1)
\end{array} \right. \]

• Entries in \( j^{th} \) column only depend on \( (j - 1)^{st} \) column and earlier entries in \( j^{th} \) column

• Only store the current column and the previous column reusing space; \( N(i, 0) \) stores \( M(i, j-1) \) and \( N(i, 1) \) stores \( M(i, j) \)
Example: DEED vs. DREAD filled by column

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>D</th>
<th>R</th>
<th>E</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Example: DEED vs. DREAD filled by column

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>$D$</th>
<th>$R$</th>
<th>$E$</th>
<th>$A$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>$D$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$D$</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: DEED vs. DREAD filled by column

<table>
<thead>
<tr>
<th></th>
<th>e</th>
<th>D</th>
<th>R</th>
<th>E</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: DEED vs. DREAD filled by column

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>$D$</th>
<th>$R$</th>
<th>$E$</th>
<th>$A$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>$D$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$D$</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: DEED vs. DREAD filled by column

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>D</th>
<th>R</th>
<th>E</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Example: DEED vs. DREAD filled by column

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>D</th>
<th>R</th>
<th>E</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Example: DEED vs. DREAD filled by column

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>D</th>
<th>R</th>
<th>E</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Figure 2: $M(i, j)$ only depends on previous column values. Keep only two columns and compute in column order.
Space Efficient Algorithm

for all $i$ do $N[i, 0] = i\delta$

for $j = 1$ to $n$ do

$N[0, 1] = j\delta$ (* corresponds to $M(0, j)$ *)

for $i = 1$ to $m$ do

$N[i, 1] = \min\left\{ \alpha_{x_i y_j} + N[i - 1, 0], \delta + N[i - 1, 1], \delta + N[i, 0] \right\}$

for $i = 1$ to $m$ do

Copy $N[i, 0] = N[i, 1]$

Analysis

Running time is $O(mn)$ and space used is $O(2m) = O(m)$
Analyzing Space Efficiency

• From the $m \times n$ matrix $M$ we can construct the actual alignment (exercise)

• Matrix $N$ computes cost of optimal alignment but no way to construct the actual alignment

Longest Common Subsequence Problem
Definition
LCS between two strings X and Y is the length of longest common subsequence between X and Y.

ABAZDC
BACBAD

ABAZDC
BACBAD
Definition

LCS between two strings $X$ and $Y$ is the length of longest common subsequence between $X$ and $Y$.

\[
\begin{align*}
\text{ABAZDC} & \quad \text{ABAZDC} \\
\text{BACBAD} & \quad \text{BACBAD}
\end{align*}
\]

Example

LCS between ABAZDC and BACBAD is 4 via ABAD
LCS Problem

Definition

LCS between two strings X and Y is the length of longest common subsequence between X and Y.

\[
\begin{align*}
&\text{ABAZDC} & \text{ABAZDC} \\
&\text{BACBAD} & \text{BACBAD}
\end{align*}
\]

Example

LCS between ABAZDC and BACBAD is 4 via ABAD

Derive a dynamic programming algorithm for the problem.
How do we plan out the recursion?

Start off with $A_{1 \ldots m}$ and $B_{1 \ldots n}$ and reason the following:

1. Assuming $A_m \neq B_n$:
   - The one or neither of the end characters are in the LCS.
   - Therefore becomes:
     $$\max(\text{LCS}(A_{1 \ldots m-1}, B_{1 \ldots n}), \text{LCS}(A_{1 \ldots m}, B_{1 \ldots n-1}))$$

2. Assuming $A_m = B_n$:
   - Either $A_m$ and $B_n$ are both in the LCS. Therefore:
     $$\text{LCS}(A_{1 \ldots m}, B_{1 \ldots n}) = 1 + \text{LCS}(A_{1 \ldots m-1}, B_{1 \ldots n-1})$$
   - Or $A_m$ and $B_n$ is not in the LCS. Therefore the LCS is either:
     $$\min(\text{LCS}(A_{1 \ldots m}, B_{1 \ldots n}), \text{LCS}(A_{1 \ldots m-1}, B_{1 \ldots n-1}))$$

Base Case:
How do we plan out the recursion?

Start off with $A[1...m]$ and $B[1...n]$ and reason the following:

• Assuming $A[m] \neq B[n]$
  • The one or neither of the end characters are in the LCS. Therefore becomes:
    $$\max(\text{LCS}(A[1...m-1], B[1...n]), \text{LCS}(A[1...m], B[1...n-1]))$$

• Assuming $A[m] = B[n]$
  • Either $A[m]$ and $B[n]$ are both in the LCS. Therefore:
    $$\text{LCS}(A[1...m], B[1...n]) = 1 + \text{LCS}(A[1...m-1], B[1...n-1])$$
  • Or $A[m]$ and $B[n]$ is not in the LCS. Therefore the LCS is either:
    $$\text{LCS}(A[1...m-1], B[1...n]), \text{LCS}(A[1...m], B[1...n-1])$$

• Base Case:
How do we plan out the recursion?

Start off with $A[1...m]$ and $B[1...n]$ and reason the following:

- Assuming $A[m] \neq B[n]$
  - The one or neither of the end characters are in the LCS. Therefore becomes:
    \[
    \max (LCS(A[1...m-1], B[1...n]), LCS(A[1...m], B[1...n-1]))
    \]

- Or $A[m]$ and $B[n]$ is not in the LCS. Therefore the LCS is either:
  \[
  LCS(A[1...m-1], B[1...n]) \quad \text{or} \quad LCS(A[1...m], B[1...n-1])
  \]

- Base Case:

How do we plan out the recursion?

Start off with $A[1...m]$ and $B[1...n]$ and reason the following:

- Assuming $A[m] \neq B[n]$
  - The one or neither of the end characters are in the LCS. Therefore becomes:
    $$\max (LCS(A[1...m - 1], B[1...n]), LCS(A[1...m], B[1...n - 1]))$$

- Assuming $A[m] = B[n]$
How do we plan out the recursion?

Start off with $A[1...m]$ and $B[1...n]$ and reason the following:

- Assuming $A[m] \neq B[n]$
  - The one or neither of the end characters are in the LCS. Therefore becomes:
    \[
    \max (LCS(A[1...m-1], B[1...n]), LCS(A[1...m], B[1...n-1]))
    \]

- Assuming $A[m] = B[n]$
  - Either $A[m]$ and $B[n]$ are both in the LCS. Therefore:
    \[
    LCS(A[1...m], B[1...n]) = 1 + LCS(A[1...m-1], B[1...n-1])
    \]
How do we plan out the recursion?

Start off with $A[1...m]$ and $B[1...n]$ and reason the following:

- Assuming $A[m] \neq B[n]$
  - The one or neither of the end characters are in the LCS. Therefore becomes:
    \[
    \max (LCS(A[1...m - 1], B[1...n]), LCS(A[1...m], B[1...n - 1]))
    \]

- Assuming $A[m] = B[n]$
  - Either $A[m]$ and $B[n]$ are both in the LCS. Therefore:
    \[
    LCS(A[1...m], B[1...n]) = 1 + LCS(A[1...m - 1], B[1...n - 1])
    \]
  - Or $A[m]$ and $B[n]$ is not in the LCS. Therefore the LCS is either:
    \[
    LCS(A[1...m - 1], B[1...n]) \\
    LCS(A[1...m], B[1...n - 1])
    \]
How do we plan out the recursion?

Start off with \(A[1...m]\) and \(B[1...n]\) and reason the following:

- Assuming \(A[m] \neq B[n]\)
  - The one or neither of the end characters are in the LCS. Therefore becomes:
    \[
    \max (LCS(A[1...m - 1], B[1...n]), LCS(A[1...m], B[1...n - 1]))
    \]

- Assuming \(A[m] = B[n]\)
  - Either \(A[m]\) and \(B[n]\) are both in the LCS. Therefore:
    \[
    LCS(A[1...m], B[1...n]) = 1 + LCS(A[1...m - 1], B[1...n - 1])
    \]
  - Or \(A[m]\) and \(B[n]\) is not in the LCS. Therefore the LCS is either:
    \[
    LCS(A[1...m - 1], B[1...n])
    \]
    \[
    LCS(A[1...m], B[1...n - 1])
    \]

- Base Case:
LCS recursive definition

\[ A[1..n], B[1..m]: \text{Input strings.} \]

\[
LCS(i, j) = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
\max \begin{pmatrix} 
LCS(i - 1, j), & A[i] \neq B[j] \\
LCS(i, j - 1), & A[i] = B[j] 
\end{pmatrix} & \text{otherwise}
\end{cases}
\]
LCS recursive definition

$A[1..n], B[1..m]$: Input strings.

$$LCS(i, j) = \begin{cases} 
0 & i = 0 \text{ or } j = 0 \\
\max \left( \begin{array}{c}
LCS(i - 1, j), \\
LCS(i, j - 1), \\
LCS(i - 1, j), \\
1 + LCS(i - 1, j - 1)
\end{array} \right) & A[i] \neq B[j] \\
\max \left( \begin{array}{c}
LCS(i - 1, j), \\
LCS(i, j - 1), \\
LCS(i - 1, j), \\
1 + LCS(i - 1, j - 1)
\end{array} \right) & A[i] = B[j]
\end{cases}$$

Running time:
Space:
Longest common subsequence is just edit distance for the two sequences...

A, B: input sequences, Σ: “alphabet” all the different values in A and B

\[ ∀b, c ∈ Σ : b ≠ c \quad \text{COST}[b][c] = +∞. \]
\[ ∀b ∈ Σ \quad \text{COST}[b][b] = 1 \]

1: price of deletion of insertion of a single character
Longest common subsequence is just edit distance for the two sequences...

$A, B$: input sequences, $\Sigma$: “alphabet” all the different values in $A$ and $B$

\[
\begin{align*}
\forall b, c \in \Sigma : b \neq c & \quad \text{COST}[b][c] = +\infty. \\
\forall b \in \Sigma & \quad \text{COST}[b][b] = 1
\end{align*}
\]

1: price of deletion of insertion of a single character

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>R</th>
<th>E</th>
<th>A</th>
<th>D</th>
<th>D</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>D</th>
<th>ED</th>
<th>LCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum ED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>Min LCS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sub-opt ED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Sub-opt LCS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Min ED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Max LCS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Longest common subsequence is just edit distance for the two sequences...

\( A, B: \) input sequences, \( \Sigma: \) “alphabet” all the different values in \( A \) and \( B \)

\[
\forall b, c \in \Sigma : b \neq c \quad \text{COST}[b][c] = +\infty.
\]

\[
\forall b \in \Sigma \quad \text{COST}[b][b] = 1
\]

1: price of deletion of insertion of a single character

Length of longest common sub-sequence =
Is in $L^k$?
A variation

**Input** A string $w \in \Sigma^*$ and access to a language $L \subseteq \Sigma^*$ via function `IsStringinL(string x)` that decides whether $x$ is in $L$, and non-negative integer $k$

**Goal** Decide if $w \in L^k$ using `IsStringinL(string x)` as a black box sub-routine

**Example**
Suppose $L$ is *English* and we have a procedure to check whether a string/word is in the *English* dictionary.

- Is the string “isthisandanenglishsentence” in *English*\(^5\)?
- Is the string “isthisandanenglishsentence” in *English*\(^4\)?
- Is “asinineat” in *English*\(^2\)?
- Is “asinineat” in *English*\(^4\)?
- Is “zibzzzad” in *English*\(^1\)?
Recursive Solution

When is $w \in L^k$?
Recursive Solution

When is $w \in L^k$?

$$\text{IsStringinLk}(A[1\ldots i], k) = \begin{cases} 
  w \in L^k \text{ iff } w = \epsilon & k = 0 \\
  w \in L^k \text{ iff } w \in L & k = 1 \\
  w \in L^k \text{ iff } u \in L^{k-1}, |w = uv| & k > 1 
\end{cases}$$

**IsStringinLk**(A[1...i], k):

1. if $k = 0$ and $i = 0$ then return YES
2. if $k = 0$ then return NO  // i > 0
3. if $k = 1$ then
   - return IsStringinL(A[1...i])

   for $\ell = 1...i - 1$ do
   - if IsStringinLk(A[1...\ell], k - 1) and IsStringinL(A[\ell + 1...i]) then return YES

4. return NO
IsStringinLk(A[1...i], k):
  if k = 0 and i = 0 then return YES
  if k = 0 then return NO  // i > 0
  if k = 1 then
    return IsStringinL(A[1...i])

  for ℓ = 1...i − 1 do
    if IsStringinLk(A[1...ℓ], k − 1) and IsStringinL(A[ℓ + 1...i]) then
      return YES

  return NO

• How many distinct sub-problems are generated by IsStringinLk(A[1..n], k)?
Analysis

**IsStringinLk**($A[1\ldots i], k$):

1. if $k = 0$ and $i = 0$ then return YES
2. if $k = 0$ then return NO  // $i > 0$
3. if $k = 1$ then
   - return **IsStringinL**($A[1\ldots i]$)

   for $\ell = 1\ldots i - 1$ do
     - if **IsStringinLk**($A[1\ldots \ell], k - 1$) and **IsStringinL**($A[\ell + 1\ldots i]$) then
       - return YES

   return NO

- How many distinct sub-problems are generated by **IsStringinLk**($A[1..n], k$)? $O(nk)$
IsStringinLk(A[1...i], k):
  if $k = 0$ and $i = 0$ then return YES
  if $k = 0$ then return NO  // $i > 0$
  if $k = 1$ then
    return IsStringinL(A[1...i])
  for $\ell = 1...i - 1$ do
    if IsStringinLk(A[1...\ell], k - 1) and IsStringinL(A[\ell + 1...i]) then
      return YES
  return NO

- How many distinct sub-problems are generated by IsStringinLk(A[1..n], k)? $O(nk)$
- How much space?
IsStringinLk(A[1\ldots i], k):
  if \( k = 0 \) and \( i = 0 \) then return YES
  if \( k = 0 \) then return NO  // \( i > 0 \)
  if \( k = 1 \) then
    return IsStringinL(A[1\ldots i])
  
  for \( \ell = 1 \ldots i - 1 \) do
    if IsStringinLk(A[1\ldots \ell], k - 1) and IsStringinL(A[\ell + 1\ldots i]) then return YES
  
  return NO

• How many distinct sub-problems are generated by IsStringinLk(A[1..n], k)? \( O(nk) \)
• How much space? \( O(nk) \)
Analysis

```python
IsStringinLk(A[1...i], k):
    if k = 0 and i = 0 then return YES
    if k = 0 then return NO  // i > 0
    if k = 1 then
        return IsStringinL(A[1...i])
    for ℓ = 1...i - 1 do
        if IsStringinLk(A[1...ℓ], k - 1) and IsStringinL(A[ℓ + 1...i]) then
            return YES
    return NO
```

• How many distinct sub-problems are generated by
  \texttt{IsStringinLk}(A[1..n], k)? $O(nk)$
• How much space? $O(nk)$
• Running time if we use memoization?
IsStringinLk(A[1...i], k):
  if \( k = 0 \) and \( i = 0 \) then return YES
  if \( k = 0 \) then return NO  // \( i > 0 \)
  if \( k = 1 \) then
    return IsStringinL(A[1...i])

  for \( \ell = 1...i - 1 \) do
    if IsStringinLk(A[1...\ell], k - 1) and IsStringinL(A[\ell + 1...i]) then
      return YES

  return NO

- How many distinct sub-problems are generated by IsStringinLk\((A[1..n], k)\)? \(O(nk)\)
- How much space? \(O(nk)\)
- Running time if we use memoization? \(O(n^2k)\)