More Dynamic Programming

Lecture 14
Tuesday, October 13, 2020
14.1
Review of dynamic programming and some new problems
What is the running time of the following?

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

$$f(x, y) = \sum_{i=1}^{x+y-1} x \cdot f(x + y - i, i - 1),$$
$$f(0, y) = y \quad f(x, 0) = x.$$

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

- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(n^3)$

The function is ill defined - it can not be computed.
Recipe for Dynamic Programming

1. Develop a recursive backtracking style algorithm $\mathcal{A}$ for given problem.
2. Identify structure of subproblems generated by $\mathcal{A}$ on an instance $I$ of size $n$.
   1. Estimate number of different subproblems generated as a function of $n$. Is it polynomial or exponential in $n$?
   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.
14.1.1
Is in $L^k$?
A variation

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

Goal Decide if \( w \in L^k \) using \( \text{IsStringinL}(\text{string } x) \) as a black box sub-routine

Example

Suppose \( L \) is \textit{English} and we have a procedure to check whether a string/word is in the \textit{English} dictionary.

- Is the string “isthisanenglishsentence” in \textit{English}^5?
- Is the string “isthisanenglishsentence” in \textit{English}^4?
- Is “asinineat” in \textit{English}^2?
- Is “asinineat” in \textit{English}^4?
- Is “zibzzzad” in \textit{English}^1?
Recursive Solution

When is $w \in L^k$?

$k = 0$: $w \in L^k$ iff $w = \epsilon$

$k = 1$: $w \in L^k$ iff $w \in L$

$k > 1$: $w \in L^k$ if $w = uv$ with $u \in L^{k-1}$ and $v \in L$

Assume $w$ is stored in array $A[1..n]$

```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
```
Recursive Solution

When is \( w \in L^k \)?

\( k = 0 \): \( w \in L^k \) iff \( w = \epsilon \)

\( k = 1 \): \( w \in L^k \) iff \( w \in L \)

\( k > 1 \): \( w \in L^k \) if \( w = uv \) with \( u \in L^{k-1} \) and \( v \in L \)

Assume \( w \) is stored in array \( A[1..n] \)

```plaintext
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
```
Recursive Solution

When is $w \in L^k$?

$k = 0$: $w \in L^k$ iff $w = \epsilon$

$k = 1$: $w \in L^k$ iff $w \in L$

$k > 1$: $w \in L^k$ if $w = uv$ with $u \in L^{k-1}$ and $v \in L$

Assume $w$ is stored in array $A[1..n]$

```plaintext
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
```
Analysis

\textbf{IsStringinLk}(A[1 \ldots i], k):

- if $k = 0$ and $i = 0$ then return \textbf{YES}
- if $k = 0$ then return \textbf{NO} \hspace{1em} // $i > 0$
- if $k = 1$ then
  \hspace{1em} return \textbf{IsStringinL}(A[1 \ldots i])

for $\ell = 1 \ldots i - 1$ do

- if \textbf{IsStringinLk}(A[1 \ldots \ell], k - 1) and \textbf{IsStringinL}(A[\ell + 1 \ldots i]) then
  \hspace{1em} return \textbf{YES}

return \textbf{NO}

- How many distinct sub-problems are generated by \textbf{IsStringinLk}(A[1..n], k)? $O(nk)$
- How much space? $O(nk)$
- Running time if we use memoization? $O(n^2 k)$
IsStringinLk(\text{A}[1 \ldots i], k):
    \begin{itemize}
    \item if \( k = 0 \) and \( i = 0 \) then return \textsc{YES}
    \item if \( k = 0 \) then return \textsc{NO}  // i > 0
    \item if \( k = 1 \) then
        return IsStringinL(\text{A}[1 \ldots i])
    \end{itemize}

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

    \textbf{return} \ \text{NO}

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

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)? $O(nk)$
- How much space? $O(nk)$
- Running time if we use memoization? $O(n^2k)$
Analysis

\textbf{IsStringinLk}(A[1 \ldots i], k):
\begin{itemize}
  \item if \( k = 0 \) and \( i = 0 \) then return YES
  \item if \( k = 0 \) then return NO \quad // i > 0
  \item if \( k = 1 \) then
    \begin{itemize}
      \item return \textbf{IsStringinL}(A[1 \ldots i])
    \end{itemize}
  \end{itemize}
\begin{itemize}
  \item for \( \ell = 1 \ldots i - 1 \) do
    \begin{itemize}
      \item if \textbf{IsStringinLk}(A[1 \ldots \ell], k - 1) and \textbf{IsStringinL}(A[\ell + 1 \ldots i]) then
        \begin{itemize}
          \item return YES
        \end{itemize}
    \end{itemize}
\end{itemize}
\begin{itemize}
  \item return NO
\end{itemize}

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

```plaintext
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)`? \( O(nk) \)
- How much space? \( O(nk) \)
- Running time if we use memoization? \( O(n^2k) \)
Analysis

\[ \text{IsStringinLk}(A[1\ldots i], k) : \]
\[
\begin{align*}
&\text{if } k = 0 \text{ and } i = 0 \text{ then return YES} \\
&\text{if } k = 0 \text{ then return NO } // i > 0 \\
&\text{if } k = 1 \text{ then} \\
&\quad \text{return IsStringinL}(A[1\ldots i]) \\
&\quad \text{for } \ell = 1\ldots i - 1 \text{ do} \\
&\quad \quad \text{if IsStringinLk}(A[1\ldots \ell], k - 1) \text{ and IsStringinL}(A[\ell + 1\ldots i]) \text{ then} \\
&\quad \quad \quad \text{return YES} \\
&\quad \text{return NO}
\end{align*}
\]

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

**Question:** What if we want to check if $w \in L^i$ for some $0 \leq i \leq k$? That is, is $w \in \bigcup_{i=0}^{k} L^i$?
Exercise

Definition

A string is a palindrome if \( w = w^R \).
Examples: \( i, \text{RACECAR, MALAYALAM, DOOFFOOD} \)

Problem: Given a string \( w \) find the longest subsequence of \( w \) that is a palindrome.

Example

\( \text{MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM} \) has
\( \text{MHYMROMYHM} \) as a palindromic subsequence
Exercise

Definition
A string is a palindrome if \( w = w^R \).
Examples: \( I \), \( RACECAR \), \( MALAYALAM \), \( DOOFFOOD \)

Problem: Given a string \( w \) find the longest subsequence of \( w \) that is a palindrome.

Example
\( MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM \) has
\( MHYMROMYHM \) as a palindromic subsequence
Exercise

Assume \( w \) is stored in an array \( A[1..n] \)

\( \text{LPS}(A[1..n]) \): length of longest palindromic subsequence of \( A \).

Recursive expression/code?
THE END

... (for now)
14.2
Edit Distance and Sequence Alignment
14.2.1

Problem definition and background
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?

Edit Distance: minimum number of “edits” to transform $x$ into $y$. 
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 \( x_1 x_2 \ldots x_n \) and \( y_1 y_2 \ldots y_m \) what is a distance between them?

**Edit Distance:** minimum number of “edits” to transform \( x \) into \( y \).
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?

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

**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 most 4:

FOOD $\rightarrow$ MOOD $\rightarrow$ MONOD $\rightarrow$ MONED $\rightarrow$ MONEY
Edit Distance: Alternate View

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.

```
FOO
 M O N E Y
```

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.
Edit Distance: Alternate View

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.
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.

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

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 \( M = \{(1, 1), (2, 2), (3, 3), (4, 5)\} \). Cost of an alignment is the number of mismatched columns plus number of unmatched indices in both strings.
Edit Distance Problem

Problem

Given two words, find the edit distance between them, i.e., an alignment of smallest cost.
Applications

1. Spell-checkers and Dictionaries
2. Unix diff
3. DNA sequence alignment ... but, we need a new metric
Similarity Metric

Definition

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

1. [Gap penalty] For each gap in the alignment, we incur a cost $\delta$.

2. [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 special case when $\delta = \alpha_{pq} = 1$. 
For two strings $X$ and $Y$, the cost of alignment $M$ is

1. **[Gap penalty]** For each gap in the alignment, we incur a cost $\delta$.
2. **[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 special case when $\delta = \alpha_{pq} = 1$. 
THE END

...

(for now)
14.2.2
Edit distance as alignment
An Example

Example

Example

\[ \text{o c c u r r a n c e} \text{Cost} = \delta + \alpha_{ae} \]

Alternative:

\[ \text{o c c u r r e n c e} \text{Cost} = 3\delta \]

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

\[ \text{o c c u r r a n c e} \text{o c c u r r e n c e} \text{Cost} = 19\delta. \]
What is the edit distance between...

What is the minimum edit distance for the following two strings, if insertion/deletion/change of a single character cost 1 unit?

374
473

1
2
3
4
5
What is the edit distance between...

What is the minimum edit distance for the following two strings, if insertion/deletion/change of a single character cost 1 unit?

373
473

1
2
3
4
5
What is the edit distance between...

What is the minimum edit distance for the following two strings, if insertion/deletion/change of a single character cost 1 unit?

37
473

1
2
3
4
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
Sequence Alignment in Practice

1. Typically the DNA sequences that are aligned are about $10^5$ letters long!
2. So about $10^{10}$ operations and $10^{10}$ bytes needed
3. The killer is the 10GB storage
4. Can we reduce space requirements?
THE END

... (for now)
14.2.3
Edit distance: The algorithm
Edit distance

Basic observation

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:

\[
\begin{array}{c|c}
\alpha & x \\
\hline
\beta & y \\
\end{array}
\quad \text{or} \quad
\begin{array}{c|c}
\alpha & x \\
\hline
\beta & y \\
\end{array}
\quad \text{or} \quad
\begin{array}{c|c}
\alpha x & \\
\hline
\beta & y \\
\end{array}
\]

Observation

Prefixes must have optimal alignment!
Problem Structure

Observation

Let $X = x_1 x_2 \cdots x_m$ and $Y = y_1 y_2 \cdots y_n$. 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.

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

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

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

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

Optimal Costs

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

$$
\text{Opt}(i, j) = \min \left\{ \alpha_{x_i y_j} + \text{Opt}(i-1, j-1), \delta + \text{Opt}(i-1, j), \delta + \text{Opt}(i, j-1) \right\}
$$

Base Cases: $\text{Opt}(i, 0) = \delta \cdot i$ and $\text{Opt}(0, j) = \delta \cdot j$
### 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>$y_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_1 \ldots y_{j-1}$</td>
<td>$y_j$</td>
<td></td>
<td>$y_1 \ldots y_{j-1}y_j$</td>
<td></td>
<td></td>
<td>$y_1 \ldots y_{j-1}$</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

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

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 $b$.

```plaintext
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>
<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>ε</td>
<td></td>
<td>D</td>
<td>R</td>
<td>E</td>
<td>A</td>
<td>D</td>
</tr>
<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 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>
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>ε</td>
<td>ε</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Example: DEED and DREAD**

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

![Graph showing the connections between DEED and DREAD]
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></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$E$</td>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$E$</td>
<td></td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$D$</td>
<td></td>
<td>3</td>
<td>3</td>
<td>3</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>

[Diagram showing possible connections between elements $D$, $R$, $E$, $A$, and $D$]
Example: DEED and DREAD

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

D | R | E | A | D
---|---|---|---|---
D | E | E | A | D
THE END

... (for now)
14.2.4
Dynamic programming algorithm for edit-distance
As part of the input...

The cost of aligning a character against another character

Σ: Alphabet

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

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

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

δ: price of deletion of insertion of a single character
Memoizing the Recursive Algorithm (Explicit Memoization)

Input: Two strings
\[ A[1 \ldots m] \]
\[ B[1 \ldots n] \]

```
EditDistance(A, B)
    int M[0..m][0..n]
    ∀i, j M[i][j] ← ∞
    return edEMI(m, n)

edEMI(i, j) // A[1 \ldots i], B[1 \ldots j]
    if M[i][j] < ∞
        return M[i][j] // stored value
    if i = 0 or j = 0
        M[i][j] = (i + j)δ
        return M[i][j]
    m_1 = δ + edEMI(i - 1, j)
    m_2 = δ + edEMI(i, j - 1)
    m_3 = COST[A[i]][B[j]]
        + edEMI(i - 1, j - 1)
    M[i][j] = min(m_1, m_2, m_3)
    return M[i][j]
```
Dynamic program for edit distance
Removing Recursion to obtain Iterative Algorithm

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

\[
\begin{align*}
\text{int } & \ M[0..m][0..n] \\
\text{for } i = 1 \text{ to } m \text{ do } & \ M[i, 0] = i \delta \\
\text{for } j = 1 \text{ to } n \text{ do } & \ M[0, j] = j \delta \\
\text{for } i = 1 \text{ to } m \text{ do } & \ \\
\text{for } j = 1 \text{ to } n \text{ do } & \ M[i][j] = \min \left\{ \begin{array}{ll}
\text{COST}[A[i]][B[j]] + M[i - 1][j - 1], \\
\delta + M[i - 1][j], \\
\delta + M[i][j - 1]
\end{array} \right. \\
\end{align*}
\]

Analysis

Running time is \( O(mn) \).
Dynamic program for edit distance
Removing Recursion to obtain Iterative Algorithm

\[
EDIST(A[1..m], B[1..n])
\]
\[
\text{int} \quad M[0..m][0..n]
\]
\[
\text{for } i = 1 \text{ to } m \text{ do } M[i, 0] = i\delta
\]
\[
\text{for } j = 1 \text{ to } n \text{ do } M[0, j] = j\delta
\]
\[
\text{for } i = 1 \text{ to } m \text{ do }
\]
\[
\text{for } j = 1 \text{ to } n \text{ do }
\]
\[
M[i][j] = \min \left\{ \begin{array}{l}
\text{COST}[A[i]][B[j]] + M[i - 1][j - 1], \\
\delta + M[i - 1][j], \\
\delta + M[i][j - 1]
\end{array} \right. 
\]

Analysis

Running time is \(O(mn)\).
Dynamic program for edit distance
Removing Recursion to obtain Iterative Algorithm

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

\[
\text{int } M[0..m][0..n] \\
\text{for } i = 1 \text{ to } m \text{ do } M[i, 0] = i\delta \\
\text{for } j = 1 \text{ to } n \text{ do } M[0, j] = j\delta \\
\text{for } i = 1 \text{ to } m \text{ do} \\
\quad \text{for } j = 1 \text{ to } n \text{ do} \\
\quad \quad M[i][j] = \min \left\{ \begin{array}{c}
\text{COST}[A[i]][B[j]] + M[i - 1][j - 1], \\
\delta + M[i - 1][j], \\
\delta + M[i][j - 1]
\end{array} \right. 
\]

Analysis

1. Running time is \( O(mn) \).
2. Space used is \( O(mn) \).
THE END

... (for now)
14.2.5
Reducing space for edit distance
Matrix and **DAG** of computation of edit distance

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

1. Recall

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

2. Entries in \( j \)th column only depend on \( (j - 1) \)st column and earlier entries in \( j \)th column.

3. 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. BREAD filled by column

<table>
<thead>
<tr>
<th></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>
</tr>
<tr>
<td>ε</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>
</tr>
<tr>
<td>E</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>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: DEED vs. BREAD 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></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. BREAD 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></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. BREAD 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. BREAD filled by column**

$$
\begin{array}{cccccc}
\varepsilon & D & R & E & A & D \\
\hline
\varepsilon & 0 & 1 & 2 & 3 & 4 & 5 \\
D & 1 & 0 & 1 & 2 \\
E & 2 & 1 & 1 & 1 \\
E & 3 & 2 & 2 & 1 \\
D & 3 & 3 & 3 & 2 \\
\end{array}
$$
Example: DEED vs. BREAD 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>3</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></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. BREAD 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>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>
Computing in column order to save space

Figure: $M(i, j)$ 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
    \[
    N[i, 1] = \min \begin{cases} 
        \alpha_{x_iy_j} + N[i - 1, 0] \\
        \delta + N[i - 1, 1] \\
        \delta + N[i, 0]
    \end{cases}
    \]
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

1. From the $m \times n$ matrix $M$ we can construct the actual alignment (exercise)
2. Matrix $N$ computes cost of optimal alignment but no way to construct the actual alignment
THE END
...
(for now)
14.2.6

Longest Common Subsequence Problem
LCS Problem

Definition
LCS between two strings \( X \) and \( Y \) is the length of longest common subsequence between \( X \) and \( Y \).

Example
LCS between ABAZDC and BACBAD is 4 via ABAD

Derive a dynamic programming algorithm for the problem.
LCS Problem

Definition

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

Example

LCS between ABAZDC and BACBAD is 4 via ABAD

Derive a dynamic programming algorithm for the problem.
LCS Problem

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

Example
LCS between ABAZDC and BACBAD is 4 via ABAD

Derive a dynamic programming algorithm for the problem.
LCS recursive definition

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

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

Similar to edit distance... *O(nm)* time algorithm *O(m)* space.
LCS recursive definition

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

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

Similar to edit distance... \( O(nm) \) time algorithm \( O(m) \) space.
Longest common subsequence is just edit distance for the two sequences...

$A$, $B$: input sequences

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

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

$\forall b \in \Sigma \quad COST[b][b] = 1$

1: price of deletion of insertion of a single character

Length of longest common subsequence $= m + n - ed(A, B)$
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 subsequence = \( m + n - \text{ed}(A, B) \)
THE END

...(for now)
14.3
Maximum Weighted Independent Set in Trees
Maximum Weight Independent Set Problem

**Input** Graph $G = (V, E)$ and weights $w(v) \geq 0$ for each $v \in V$

**Goal** Find maximum weight independent set in $G$

![Graph with weights](image)

Maximum weight independent set in above graph: $\{B, D\}$
Maximum Weight Independent Set Problem

Input  Graph $G = (V, E)$ and weights $w(v) \geq 0$ for each $v \in V$

Goal  Find maximum weight independent set in $G$

Maximum weight independent set in above graph: $\{B, D\}$
Maximum Weight Independent Set in a Tree

Input: Tree \( T = (V, E) \) and weights \( w(v) \geq 0 \) for each \( v \in V \)

Goal: Find maximum weight independent set in \( T \)

Maximum weight independent set in above tree: ??
Towards a Recursive Solution

For an arbitrary graph $G$:

1. Number vertices as $v_1, v_2, \ldots, v_n$

2. Find recursively optimum solutions without $v_n$ (recurse on $G - v_n$) and with $v_n$ (recurse on $G - v_n - N(v_n)$ & include $v_n$).

3. Saw that if graph $G$ is arbitrary there was no good ordering that resulted in a small number of subproblems.

What about a tree? Natural candidate for $v_n$ is root $r$ of $T$?
Towards a Recursive Solution

For an arbitrary graph $G$:

1. Number vertices as $v_1, v_2, \ldots, v_n$

2. Find recursively optimum solutions without $v_n$ (recurse on $G - v_n$) and with $v_n$ (recurse on $G - v_n - N(v_n)$ & include $v_n$).

3. Saw that if graph $G$ is arbitrary there was no good ordering that resulted in a small number of subproblems.

What about a tree? Natural candidate for $v_n$ is root $r$ of $T$?
Towards a Recursive Solution

For an arbitrary graph $G$:

1. Number vertices as $v_1, v_2, \ldots, v_n$
2. Find recursively optimum solutions without $v_n$ (recurse on $G - v_n$) and with $v_n$ (recurse on $G - v_n - N(v_n)$ & include $v_n$).

3. Saw that if graph $G$ is arbitrary there was no good ordering that resulted in a small number of subproblems.

What about a tree? Natural candidate for $v_n$ is root $r$ of $T$?
Towards a Recursive Solution

Natural candidate for $v_n$ is root $r$ of $T$? Let $O$ be an optimum solution to the whole problem.

Case $r \notin O$ : Then $O$ contains an optimum solution for each subtree of $T$ hanging at a child of $r$.

Case $r \in O$ : None of the children of $r$ can be in $O$. $O - \{r\}$ contains an optimum solution for each subtree of $T$ hanging at a grandchild of $r$.

Subproblems? Subtrees of $T$ rooted at nodes in $T$.

How many of them? $O(n)$
Towards a Recursive Solution

Natural candidate for $v_n$ is root $r$ of $T$? Let $O$ be an optimum solution to the whole problem.

Case $r \notin O$ : Then $O$ contains an optimum solution for each subtree of $T$ hanging at a child of $r$.

Case $r \in O$ : None of the children of $r$ can be in $O$. $O - \{r\}$ contains an optimum solution for each subtree of $T$ hanging at a grandchild of $r$.

Subproblems? Subtrees of $T$ rooted at nodes in $T$.

How many of them? $O(n)$
Towards a Recursive Solution

Natural candidate for $v_n$ is root $r$ of $T$? Let $O$ be an optimum solution to the whole problem.

Case $r \not\in O$: Then $O$ contains an optimum solution for each subtree of $T$ hanging at a child of $r$.

Case $r \in O$: None of the children of $r$ can be in $O$. $O - \{r\}$ contains an optimum solution for each subtree of $T$ hanging at a grandchild of $r$.

Subproblems? Subtrees of $T$ rooted at nodes in $T$.

How many of them? $O(n)$
Towards a Recursive Solution

Natural candidate for $v_n$ is root $r$ of $T$? Let $O$ be an optimum solution to the whole problem.

Case $r \notin O$ : Then $O$ contains an optimum solution for each subtree of $T$ hanging at a child of $r$.

Case $r \in O$ : None of the children of $r$ can be in $O$. $O - \{r\}$ contains an optimum solution for each subtree of $T$ hanging at a grandchild of $r$.

Subproblems? Subtrees of $T$ rooted at nodes in $T$.

How many of them? $O(n)$
Towards a Recursive Solution

Natural candidate for $v_n$ is root $r$ of $T$? Let $O$ be an optimum solution to the whole problem.

Case $r \notin O$ : Then $O$ contains an optimum solution for each subtree of $T$ hanging at a child of $r$.

Case $r \in O$ : None of the children of $r$ can be in $O$. $O - \{r\}$ contains an optimum solution for each subtree of $T$ hanging at a grandchild of $r$.

Subproblems? Subtrees of $T$ rooted at nodes in $T$.

How many of them? $O(n)$
A Recursive Solution

$T(u)$: subtree of $T$ hanging at node $u$

$OPT(u)$: max weighted independent set value in $T(u)$

$$OPT(u) = \max \left\{ \sum_{v \text{ child of } u} OPT(v),
\sum_{v \text{ grandchild of } u} OPT(v),
\nu(u) + \sum_{v \text{ grandchild of } u} OPT(v) \right\}$$
A Recursive Solution

$T(u)$: subtree of $T$ hanging at node $u$

$OPT(u)$: max weighted independent set value in $T(u)$

$$OPT(u) = \max \left\{ \sum_{\text{child of } u} OPT(v), \quad w(u) + \sum_{\text{grandchild of } u} OPT(v) \right\}$$
Iterative Algorithm

1. Compute $OPT(u)$ bottom up. To evaluate $OPT(u)$ need to have computed values of all children and grandchildren of $u$

2. What is an ordering of nodes of a tree $T$ to achieve above? Post-order traversal of a tree.
Iterative Algorithm

1. Compute $OPT(u)$ bottom up. To evaluate $OPT(u)$ need to have computed values of all children and grandchildren of $u$.

2. What is an ordering of nodes of a tree $T$ to achieve above? Post-order traversal of a tree.
Iterative Algorithm

**MIS-Tree** ($T$):

Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of $T$

for $i = 1$ to $n$ do

$$M[v_i] = \max \left( \sum_{j \text{ child of } v_i} M[v_j], w(v_i) + \sum_{j \text{ grandchild of } v_i} M[v_j] \right)$$

return $M[v_n]$ (* Note: $v_n$ is the root of $T$ *)

Space: $O(n)$ to store the value at each node of $T$

Running time:

- Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are $n$ evaluations.
- Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.
Iterative Algorithm

**MIS-Tree** ($T$):

Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of $T$

for $i = 1$ to $n$ do

$$M[v_i] = \max \left( \sum_{v_j \text{ child of } v_i} M[v_j], w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \right)$$

return $M[v_n]$ (* Note: $v_n$ is the root of $T$ *)

Space: $O(n)$ to store the value at each node of $T$

Running time:

1. Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are $n$ evaluations.
2. Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.
Iterative Algorithm

**MIS-Tree**(*T*):

Let \( v_1, v_2, \ldots, v_n \) be a post-order traversal of nodes of \( T \)

for \( i = 1 \) to \( n \) do

\[
M[v_i] = \max \left( \sum_{\text{child of } v_i} M[v_j], \quad w(v_i) + \sum_{\text{grandchild of } v_i} M[v_j] \right)
\]

return \( M[v_n] \) (* Note: \( v_n \) is the root of \( T \) *)

**Space:** \( O(n) \) to store the value at each node of \( T \)

**Running time:**

- Naive bound: \( O(n^2) \) since each \( M[v_i] \) evaluation may take \( O(n) \) time and there are \( n \) evaluations.

- Better bound: \( O(n) \). A value \( M[v_j] \) is accessed only by its parent and grand parent.
Iterative Algorithm

**MIS-Tree** ($T$):

Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of $T$

**for** $i = 1$ **to** $n$ **do**

$$M[v_i] = \max \left( \sum_{v_j \text{ child of } v_i} M[v_j], w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \right)$$

**return** $M[v_n]$ (*Note: $v_n$ is the root of $T$*)

**Space:** \(O(n)\) to store the value at each node of $T$

**Running time:**

1. Naive bound: \(O(n^2)\) since each $M[v_i]$ evaluation may take \(O(n)\) time and there are $n$ evaluations.

2. Better bound: \(O(n)\). A value $M[v_j]$ is accessed only by its parent and grandparent.
Iterative Algorithm

**MIS-Tree**$(T)$:

Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of $T$

for $i = 1$ to $n$ do

\[
M[v_i] = \max \left( \sum_{\text{child of } v_i} M[v_j], w(v_i) + \sum_{\text{grandchild of } v_i} M[v_j] \right)
\]

return $M[v_n]$ (* Note: $v_n$ is the root of $T$ *)

**Space:** $O(n)$ to store the value at each node of $T$

**Running time:**

1. Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are $n$ evaluations.

2. Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.
Example

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```

```
Example

```
THE END
...
(for now)
14.4
Dynamic programming and DAGs
Takeaway Points

1. Dynamic programming is based on finding a recursive way to solve the problem. Need a recursion that generates a small number of subproblems.

2. Given a recursive algorithm there is a natural DAG associated with the subproblems that are generated for given instance; this is the dependency graph. An iterative algorithm simply evaluates the subproblems in some topological sort of this DAG.

3. The space required to evaluate the answer can be reduced in some cases by a careful examination of that dependency DAG of the subproblems and keeping only a subset of the DAG at any time.