More Dynamic Programming

Lecture 14
March 9, 2017
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:

- (A) $O(n)$
- (B) $O(n \log n)$
- (C) $O(n^2)$
- (D) $O(n^3)$
- (E) 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.
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 $\text{English}$ and we have a procedure to check whether a string/word is in the $\text{English}$ dictionary.

- Is the string “isthisenganlishsentence” in $\text{English}^5$?
- Is the string “isthisenganlishsentence” in $\text{English}^4$?
- Is “asinineat” in $\text{English}^2$?
- Is “asinineat” in $\text{English}^4$?
- Is “zibzzzzad” in $\text{English}^1$?
Recursive Solution

When is \( w \in L^k \)?
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 $u \in L$ and $v \in L^{k-1}$
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 $u \in L$ and $v \in L^{k-1}$

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

```python
IsStringinLk(A[1..n], k):
    If (k = 0)
        If (n = 0) Output YES
        Else Output NO
    If (k = 1)
        Output IsStringinL(A[1..n])
    Else
        For (i = 1 to n - 1) do
            If (IsStringinL(A[1..i]) and IsStringinLk(A[i+1..n], k - 1))
                Output YES
        Output NO
```
IsStringinLk\((A[1..n], k)\):

If \((k = 0)\)
  If \((n = 0)\) Output YES
  Else Output NO

If \((k = 1)\)
  Output IsStringinL\((A[1..n])\)

Else
  For \((i = 1\) to \(n - 1\)) do
    If \((\text{IsStringinL}(A[1..i]) \text{ and IsStringinLk}(A[i + 1..n], k - 1))\)
      Output YES

Output NO

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

\textbf{IsStringinLk}(A[1..n], k):
\begin{itemize}
  \item If \((k = 0)\)
    \begin{itemize}
      \item If \((n = 0)\) Output YES
      \item Else Output NO
    \end{itemize}
  \item If \((k = 1)\)
    \begin{itemize}
      \item Output \textbf{IsStringinL}(A[1..n])
    \end{itemize}
  \item Else
    \begin{itemize}
      \item For \((i = 1 \text{ to } n - 1)\) do
        \begin{itemize}
          \item If \((\text{IsStringinL}(A[1..i]) \text{ and IsStringinLk}(A[i + 1..n], k - 1))\)
            \begin{itemize}
              \item Output YES
            \end{itemize}
        \end{itemize}
    \end{itemize}
\end{itemize}
Output NO

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

IsStringinLk(A[1..n], k):
    If (k = 0)
        If (n = 0) Output YES
        Else Output NO
    If (k = 1)
        Output IsStringinL(A[1..n])
    Else
        For (i = 1 to n - 1) do
            If (IsStringinL(A[1..i]) and IsStringinLk(A[i + 1..n], k - 1))
                Output YES
        Output NO

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

Chandra Chekuri (UIUC)  CS374  6  Spring 2017  6 / 42
Analysis

\textbf{IsStringinLk}(A[1..n], k):

- If \((k = 0)\)
  - If \((n = 0)\) Output YES
  - Else Output NO
- If \((k = 1)\)
  - Output \textbf{IsStringinL}(A[1..n])
- Else
  - For \((i = 1\) to \(n - 1\)\) do
    - If \((\text{IsStringinL}(A[1..i])\) and \(\text{IsStringinLk}(A[i + 1..n], k - 1)\))
      - Output YES

Output NO

- How many distinct sub-problems are generated by \(\text{IsStringinLk}(A[1..n], k)\)? \(O(nk)\)
- How much space? \(O(nk)\) pause
- Running time?
IsStringinLk\((A[1..n], k)\):

If \((k = 0)\)

If \((n = 0)\) Output YES
Else Output NO

If \((k = 1)\)

Output IsStringinL\((A[1..n])\)

Else

For \((i = 1\) to \(n - 1\)) do

If \((\text{IsStringinL}(A[1..i]) \text{ and } \text{IsStringinLk}(A[i + 1..n], k - 1))\)

Output YES

Output NO

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

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

Examples: $I$, *RACECAR*, *MALAYALAM*, *DOOFFOOD*
**Exercise**

**Definition**
A string is a palindrome if $w = w^R$.
Examples: $I$, $RACECAR$, $MALAYALAM$, $DOFFFOOD$

**Problem:** Given a string $w$ find the longest subsequence of $w$ that is a palindrome.

**Example**

$MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM$ has $MHYMROHYM$ as a palindromic subsequence
Exercise

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

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

Recursive expression/code?
Part I

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?

What does nearness mean?

Question: Given two strings $x_1 \ldots x_n$ and $y_1 \ldots y_m$, 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:

$FOOD \rightarrow MOOD \rightarrow MONOD \rightarrow MONED \rightarrow MONEY$
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.

```
FOOD
MONEY

FOOD
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.

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

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 \( M = \{(1, 1), (2, 2), (3, 3), (4, 5)\} \).
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.

```
FOOD
MONEY
```

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
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 $\alpha_{pp} = 0$.
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 a special case when $\delta = \alpha_{pq} = 1$. 
**Example**

\[
\begin{array}{c}
\text{o} \\
\text{occurrence} \\
\text{o} \\
\text{occurrence}
\end{array}
\quad \text{Cost} = \delta + \alpha_{ae}
\]

Alternative:

\[
\begin{array}{c}
\text{o} \\
\text{occurrence} \\
\text{o} \\
\text{occurrence}
\end{array}
\quad \text{Cost} = 3\delta
\]

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

\[
\begin{array}{c}
\text{ocurrence} \\
\text{ocurrence}
\end{array}
\quad \text{Cost} = 19\delta.
\]
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?

(A) 1
(B) 2
(C) 3
(D) 4
(E) 5
Input  Given two words $X$ and $Y$, and gap penalty $\delta$ and mismatch costs $\alpha_{pq}$

Goal  Find alignment of minimum cost
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:

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

**Observation**

*Prefixes must have optimal alignment!*
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 $y_1 \cdots y_{n-1}$
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

Optimal Costs

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

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

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
Memoizing the Recursive Algorithm

\[
\text{int } M[0..m][0..n] \\
\text{Initialize all entries of } M[i][j] \text{ to } \infty \\
\text{return } \text{EDIST}(A[1..m], B[1..n])
\]

\[
\text{EDIST}(A[1..m], B[1..n]) \\
\text{If } (M[i][j] < \infty) \text{ return } M[i][j] \text{ (* return stored value *)}
\]

\[
\text{If } (m = 0) \\
M[i][j] = n\delta \\
\text{ElseIf } (n = 0) \\
M[i][j] = m\delta \\
\text{Else} \\
m_1 = \delta + \text{EDIST}(A[1..(m - 1)], B[1..n]) \\
m_2 = \delta + \text{EDIST}(A[1..m], B[1..(n - 1)]) \\
m_3 = \text{COST}[A[m], B[n]] + \text{EDIST}(A[1..(m - 1)], B[1..(n - 1)]) \\
M[i][j] = \min(m_1, m_2, m_3) \\
\text{return } M[i][j]
\]
EDIST(A[1..m], B[1..n])

\[
\begin{align*}
\text{int } M[0..m][0..n] \\
\text{for } i = 1 \text{ to } m \text{ do } M[i, 0] = i\delta \\
\text{for } j = 1 \text{ to } n \text{ do } M[0, j] = j\delta \\
\text{for } i = 1 \text{ to } m \text{ do } \\
\quad \text{for } j = 1 \text{ to } n \text{ do } \\
\quad \quad M[i][j] = \min\left\{ \alpha_{x_i,y_j} + M[i - 1][j - 1], \\
\delta + M[i - 1][j], \\
\delta + M[i][j - 1] \right\}
\end{align*}
\]

Analysis

Running time is \(O(mn)\).
Removing Recursion to obtain Iterative Algorithm

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

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

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

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

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

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

\[
M[i][j] = \min\left\{\begin{array}{l}
\alpha_{x_iy_j} + M[i - 1][j - 1], \\
\delta + M[i - 1][j], \\
\delta + M[i][j - 1]
\end{array}\right\}
\]

Analysis

Running time is \(O(mn)\).
Removing Recursion to obtain Iterative Algorithm

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

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

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

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

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

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

\[ M[i][j] = \min \left\{ \alpha_{x_i y_j} + M[i-1][j-1], \delta + M[i-1][j], \delta + M[i][j-1] \right\} \]

Analysis

1. Running time is \( O(mn) \).
2. Space used is \( O(mn) \).
Figure: Iterative algorithm in previous slide computes values in row order.
Example

DEED and DREAD
Typically the DNA sequences that are aligned are about $10^5$ letters long!

So about $10^{10}$ operations and $10^{10}$ bytes needed

The killer is the 10GB storage

Can we reduce space requirements?
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 \( M(i, j) \)
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.

Part II

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
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
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.
Part III

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 $G$
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.
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?
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 \( r \not\in O \): 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 \( r \).
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 $r$. 
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 $T$. 
Towards a Recursive Solution

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

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

**Case** $r \in \mathcal{O}$ : None of the children of $r$ can be in $\mathcal{O}$. $\mathcal{O} - \{r\}$ 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?
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? \( O(n) \)
Example

```
Example

r
a  b
c  d  e  f  g
h  i  j
10
5 8
4 4 9
2 7 8
11
3
```

Chandra Chekuri  (UIUC)  
CS374  
Spring 2017
A Recursive Solution

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

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

$$OPT(u) =$$
A Recursive Solution

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

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

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

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

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

\textbf{MIS-Tree}(T):

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

\begin{verbatim}
for \( i = 1 \) to \( n \) do

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

\end{verbatim}

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

Space:
Iterative Algorithm

\textbf{MIS-Tree}(T):

\begin{align*}
\text{Let } v_1, v_2, \ldots, v_n \text{ be a post-order traversal of nodes of } T \\
\text{for } i = 1 \text{ to } n \text{ 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) \\
\text{return } M[v_n] \quad (* \text{ Note: } v_n \text{ is the root of } T *)
\end{align*}

Space: \(O(n)\) to store the value at each node of \(T\)
Running time:
Iterative Algorithm

\[ \text{MIS-Tree}(T): \]

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

\( \text{for } i = 1 \text{ to } n \text{ do} \)

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

\( \text{return } M[v_n] \) (* 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.
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.
Example
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.