

# More Dynamic Programming

## Lecture 14

Tuesday, March 5, 2019

LaTeXed: March 5, 2019 16:39

# What is the running time of the following?

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

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

*f(x, y)*  
*x < 2n*  
*y < 2n*

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.
- i=1 → n f(2n-1, 0)*  
*i=2 → n f(2n-2, 1)*

# 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 **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 “isthisanenglishsentence” in *English*<sup>5</sup>?
- Is the string “isthisanenglishsentence” in *English*<sup>4</sup>?
- Is “asinineat” in *English*<sup>2</sup>?
- Is “asinineat” in *English*<sup>4</sup>?
- Is “zibzzzad” in *English*<sup>1</sup>?

# 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  $A[1..n]$

**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

# 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 **IsStringinL**( $A[i + 1..n]$ ,  $k - 1$ ))

            Output YES

Output NO

- How many distinct sub-problems are generated by **IsStringinL**( $A[1..n]$ ,  $k$ )?



# 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 **IsStringinL**( $A[i + 1..n]$ ,  $k - 1$ ))

            Output YES

Output NO

- How many distinct sub-problems are generated by **IsStringinL**( $A[1..n]$ ,  $k$ )?  $O(nk)$

# 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 **IsStringinL**( $A[i + 1..n]$ ,  $k - 1$ ))

            Output YES

Output NO

- How many distinct sub-problems are generated by **IsStringinL**( $A[1..n]$ ,  $k$ )?  $O(nk)$
- How much space?

# 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 **IsStringinL**( $A[i + 1..n]$ ,  $k - 1$ ))

            Output YES

Output NO

- How many distinct sub-problems are generated by **IsStringinL**( $A[1..n]$ ,  $k$ )?  $O(nk)$
- How much space?  $O(nk)$
- Running time?

# 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 **IsStringinL**( $A[i + 1..n]$ ,  $k - 1$ ))

            Output YES

Output NO

- How many distinct sub-problems are generated by **IsStringinL**( $A[1..n]$ ,  $k$ )?  $O(nk)$
- How much space?  $O(nk)$
- 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  $w \in \cup_{i=0}^k L^i$ ?

# Exercise

## Definition

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*, *DOOFFOOD*

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

## Example

*MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM* has  
*MHYMRORMYHM* 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?

# 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  $x_1x_2 \dots x_n$  and  $y_1y_2 \dots y_m$  what is a *distance* between them?

# 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  $x_1x_2 \dots x_n$  and  $y_1y_2 \dots y_m$  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 → MOOD → MONOD → MONED → MONEY

*replace*      *insert*      *replace*      *replace*

# 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.

A handwritten alignment diagram showing the words "FOO" and "MON EY" aligned vertically. The word "FOO" is on the top line, and "MON EY" is on the bottom line. A blue arrow labeled "replac" points from the 'F' in "FOO" to the 'M' in "MON EY". Another blue arrow labeled "insert" points from the 'E' in "MON EY" to the 'D' in "FOO".

F	O	O		D
M	O	N	E	Y

# 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.

F O O     D  
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  $M = \{(1, 1), (2, 2), (3, 3), (4, 5)\}$ .

# 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.

F	O	O		D
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 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  $\alpha_{pp} = 0$ .

# 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  $\delta = \alpha_{pq} = 1$ .

# An Example

## Example

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

*Handwritten annotations:* "insert" with an arrow pointing to the second column (c) in the second row; "replace" with an arrow pointing to the sixth column (a) in the first row.

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

Alternative:

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

*Handwritten annotations:* "insert" with an arrow pointing to the second column (c) in the second row; "insert" with an arrow pointing to the sixth column (a) in the first row; "delete" with an arrow pointing to the sixth column (a) in the first row.

$$\text{Cost} = \underline{3\delta}$$

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		

$$\text{Cost} = \underline{19\delta}.$$

# 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

- (A) 1
- (B) 2
- (C) 3
- (D) 4
- (E) 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

- (A) 1
- (B) 2
- (C) 3
- (D) 4
- (E) 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

- (A) 1
- (B) 2
- (C) 3
- (D) 4
- (E) 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

## 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:

$\alpha$	$x$
$\beta$	$y$

or

$\alpha$	$x$
$\beta y$	

or

$\alpha x$	
$\beta$	$y$

## Observation

*Prefixes must have optimal alignment!*

# Problem Structure

## Observation

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.

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

## Optimal Costs

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

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

# Subproblems and Recurrence

## Optimal Costs

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

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

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  $m\delta$

$m_1 = \delta + EDIST(A[1..(m - 1)], B[1..n])$

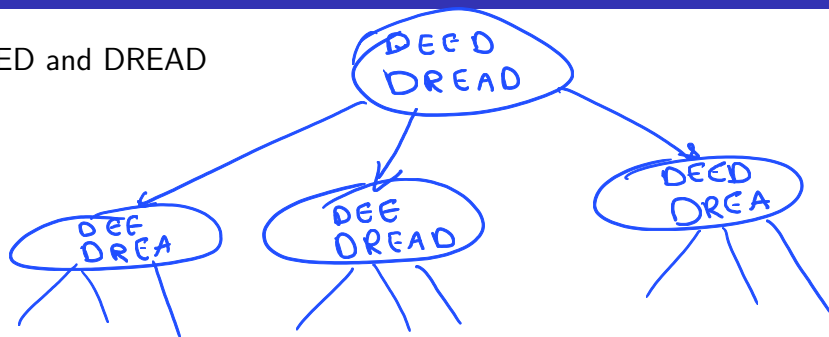
$m_2 = \delta + EDIST(A[1..m], B[1..(n - 1)])$

$m_3 = COST[A[m], B[n]] + EDIST(A[1..(m - 1)], B[1..(n - 1)])$

return  $\min(m_1, m_2, m_3)$

# Example

DEED and DREAD



# Memoizing the Recursive Algorithm

```
int M[0..m][0..n]
```

```
Initialize all entries of  $M[i][j]$  to  $\infty$   
return  $EDIST(A[1..m], B[1..n])$ 
```

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

```
If ( $M[i][j] < \infty$ ) return  $M[i][j]$  (* return stored value *)
```

```
If ( $m = 0$ )
```

```
     $M[i][j] = n\delta$ 
```

```
ElseIf ( $n = 0$ )
```

```
     $M[i][j] = m\delta$ 
```

```
Else
```

```
     $m_1 = \delta + EDIST(A[1..(m-1)], B[1..n])$ 
```

```
     $m_2 = \delta + EDIST(A[1..m], B[1..(n-1)])$ 
```

```
     $m_3 = COST[A[m], B[n]] + EDIST(A[1..(m-1)], B[1..(n-1)])$ 
```

```
     $\rightarrow M[i][j] = \min(m_1, m_2, m_3)$ 
```

```
return  $M[i][j]$ 
```



# Removing Recursion to obtain Iterative Algorithm

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

*int*  $M[0..m][0..n]$

for  $i = 1$  to  $m$  do  $M[i, 0] = i\delta$

for  $j = 1$  to  $n$  do  $M[0, j] = j\delta$

for  $i = 1$  to  $m$  do

for  $j = 1$  to  $n$  do

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

# Removing Recursion to obtain Iterative Algorithm

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

*int*  $M[0..m][0..n]$

for  $i = 1$  to  $m$  do  $M[i, 0] = i\delta$

for  $j = 1$  to  $n$  do  $M[0, j] = j\delta$

for  $i = 1$  to  $m$  do

for  $j = 1$  to  $n$  do

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

## Analysis

# Removing Recursion to obtain Iterative Algorithm

```
EDIST(A[1..m], B[1..n])  
  int M[0..m][0..n]  
  for i = 1 to m do M[i, 0] = iδ  
  for j = 1 to n do M[0, j] = jδ  
  
  for i = 1 to m do  
    for j = 1 to n do  
      
$$M[i][j] = \min \begin{cases} \alpha_{x_i y_j} + M[i-1][j-1], \\ \delta + M[i-1][j], \\ \delta + M[i][j-1] \end{cases}$$

```

## Analysis

- 1 Running time is  $O(mn)$ .
- 2 Space used is  $O(mn)$ .

# Matrix and DAG of Computation

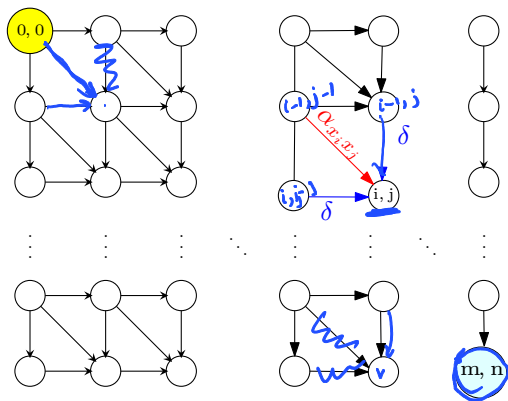


Figure: Iterative algorithm in previous slide computes values in row order.

# Example

DEED and DREAD

# 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?

# 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  $M(i, j)$

# Computing in column order to save space

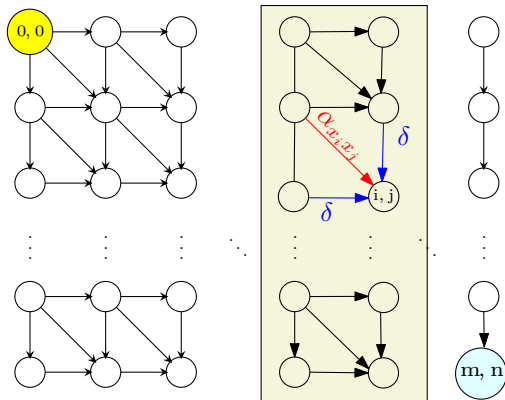


Figure:  $M(i, j)$  only depends on previous column values. Keep only two columns and compute in column order.



# Space Efficient Algorithm

```
for all  $i$  do  $N[i, 0] = i\delta$ 
for  $j = 1$  to  $n$  do
   $N[0, j] = j\delta$  (* corresponds to  $M(0, j)$  *)
  for  $i = 1$  to  $m$  do
    
$$N[i, j] = \min \begin{cases} \alpha_{x_i y_j} + N[i - 1, 0] \\ \delta + N[i - 1, j] \\ \delta + N[i, 0] \end{cases}$$

  for  $i = 1$  to  $m$  do
    Copy  $N[i, 0] = N[i, j]$ 
```

## 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
- ③ Space efficient computation of alignment? More complicated algorithm — see notes and Kleinberg-Tardos book.

## 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.

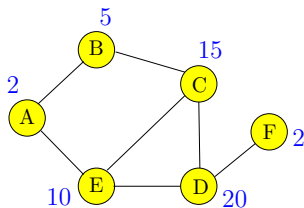
## Part III

# 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  $G$

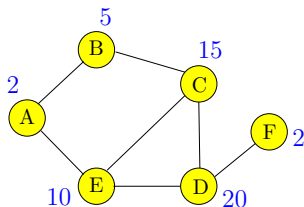




# Maximum Weight Independent Set Problem

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

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

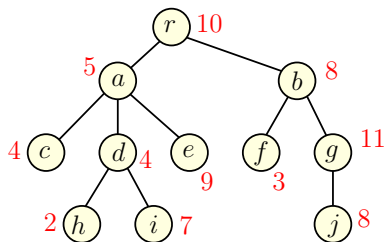


Maximum weight independent set in above graph:  $\{B, D\}$

# 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  $v_1, v_2, \dots, v_n$
- 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  $v_1, v_2, \dots, v_n$
- 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  $v_1, v_2, \dots, v_n$
- 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  $\mathcal{O}$  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  $r$ .

# 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  $r$ .

# 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  $T$ .



# 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  $\mathcal{O}$  be an optimum solution to the whole problem.

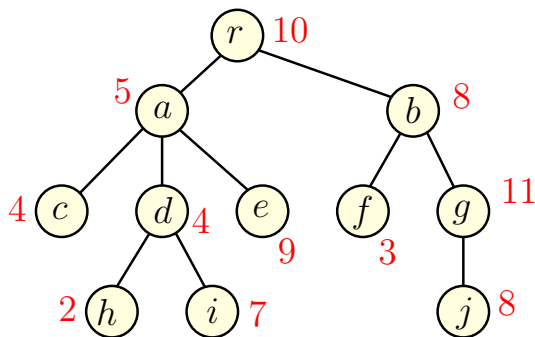
Case  $r \notin \mathcal{O}$  : Then  $\mathcal{O}$  contains an optimum solution for each subtree of  $T$  hanging at a child of  $r$ .

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?  $O(n)$

# Example



# 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  $T(u)$

$$OPT(u) = \max \left\{ \begin{array}{l} \sum_{v \text{ child of } u} OPT(v), \\ w(u) + \sum_{v \text{ grandchild of } u} OPT(v) \end{array} \right.$$

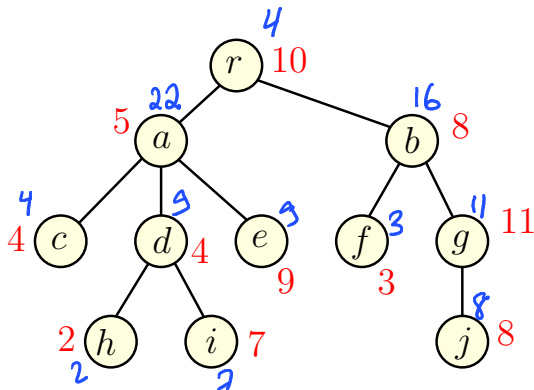
# Iterative Algorithm

- 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.

# Example





# Iterative Algorithm

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

Let  $v_1, v_2, \dots, v_n$  be a post-order traversal of nodes of  $T$   
**for**  $i = 1$  to  $n$  **do**

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

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

# Iterative Algorithm

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

Let  $v_1, v_2, \dots, v_n$  be a post-order traversal of nodes of  $T$   
**for**  $i = 1$  **to**  $n$  **do**

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

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

Space:

# Iterative Algorithm

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

Let  $v_1, v_2, \dots, v_n$  be a post-order traversal of nodes of  $T$   
**for**  $i = 1$  to  $n$  **do**

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

**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:**

# Iterative Algorithm

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

Let  $v_1, v_2, \dots, v_n$  be a post-order traversal of nodes of  $T$   
**for**  $i = 1$  **to**  $n$  **do**

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

**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.

# Iterative Algorithm

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

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

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

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

**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.

# 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.