

Dynamic Programming

Jared Saia

University of New Mexico

Dynamic Programming

- Intro
- Edit Distance and String Alignment
- Matrix-Chain Multiplication
- Tree Independent Set

DP Intro

“Those who cannot remember the past are doomed to repeat it.” - George Santayana, The Life of Reason, Book I: Introduction and Reason in Common Sense (1905)

What is Dynamic Programming?

- Dynamic programming applies when a problem has *optimal substructure* and *overlapping subproblems*.
- Each distinct subproblem is solved once, using either top-down *memoization* or bottom-up *tabulation*.
- Basic trick: *Do not solve the same subproblem more than once!*

Fibonacci Example

Let $F_0 = F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. Consider the following procedure for computing F_n :

Fib(n)

if $n < 2$ **then return** 1

return $\text{Fib}(n - 1) + \text{Fib}(n - 2)$

Analysis

- Q: What is the runtime of Fib?
- A: Except for recursive calls, the entire algorithm takes a constant number of steps. If $T(n)$ is the runtime of the algorithm on input n , then we can say that:
$$T(0) = T(1) = 1, T(n) = T(n-2) + T(n-1) + 1$$
- It is easy to show by induction that $T(n) = 2F_n - 1$. This is very bad!

Aside

- Q: How can we solve $T(n)$ exactly?
- A: We solved this recurrence using annihilators in the last lecture to get $T(n) = c_1\phi^n + c_2\hat{\phi}^n + c_31^n$ where $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi} = \frac{1-\sqrt{5}}{2}$.

Aside II

- If we solve for constants, we get that:

$$T(0) = 1 = c_1 + c_2 + c_3$$

$$T(1) = 1 = c_1\phi + c_2\hat{\phi} + c_3$$

$$T(2) = 3 = c_1\phi^2 + c_2\hat{\phi}^2 + c_3$$

Solving this system of linear equations (using Gaussian elimination) gives:

$$c_1 = 1 + \frac{1}{\sqrt{5}}, \quad c_2 = 1 - \frac{1}{\sqrt{5}}, \quad c_3 = -1,$$

Aside III

- So our final solution is

$$T(n) = \left(1 + \frac{1}{\sqrt{5}}\right) \phi^n + \left(1 - \frac{1}{\sqrt{5}}\right) \hat{\phi}^n - 1 = \Theta(\phi^n).$$

The Problem

- The reason Fib is so slow is that it computes the same Fibonacci numbers over and over
- For $n \geq 3$, $\text{Fib}(n - 2)$ is evaluated twice, and lower subproblems are repeated even more often
- We can greatly speed up the algorithm by writing down the results of the recursive calls and looking them up if needed

Memoized Fibonacci

Assume $M[0 \dots n]$ is initially Undefined.

Memoized-Fib(n)

if $n < 2$ **then return** 1

if $M[n] = \text{Undefined}$ **then**

$M[n] \leftarrow \text{Memoized-Fib}(n - 1) + \text{Memoized-Fib}(n - 2)$

return $M[n]$

Bottom-Up Fibonacci

Bottom-Up-Fib(n)

if $n < 2$ **then return** 1

$F[0] \leftarrow 1; F[1] \leftarrow 1$

for $i \leftarrow 2$ **to** n **do**

$F[i] \leftarrow F[i - 1] + F[i - 2]$

return $F[n]$

Analysis

- Each distinct subproblem $\text{Fib}(x)$, for $0 \leq x \leq n$, is computed exactly once.
- Computing a new table entry takes constant time.
- Thus both algorithms run in $\Theta(n)$ time, a *huge* savings.
- The bottom-up version uses $\Theta(n)$ space as written, or $\Theta(1)$ space if we retain only the two most recent values.

Take Away

Dynamic programming differs from divide and conquer in the following way:

- “Divide and Conquer” divides problem into independent subproblems, solves the subproblems recursively and then combines solutions to solve original problem
- Dynamic programming is useful when subproblems overlap
- For these kinds of problems, divide and conquer does more work than necessary
- Dynamic programming solves each distinct subproblem once and saves the answer in a table for future reference

The Pattern

- **Formulate the problem recursively.** Write down a formula for the whole problem as a simple combination of answers to smaller subproblems
- **Build solutions to your recurrence from the bottom up.** Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution by considering the intermediate subproblems in the correct order.

Note: Dynamic programs store the results of intermediate subproblems. This is frequently *but not always* done with some type of table.

Part I: Edit Distance

- The *edit distance* between two words is the minimum number of letter insertions, letter deletions, and letter substitutions required to transform one word into another. For example, the edit distance between FOOD and MONEY is at most four:

FOOD $\rightarrow$ MOOD $\rightarrow$ MON_^D $\rightarrow$ MONED $\rightarrow$ MONEY

String Alignment

Better way to display this process:

- Place the words one above the other in a table
- Put a gap in the first word for every insertion and a gap in the second word for every deletion
- Columns with two different characters correspond to substitutions
- Then the number of editing steps is just the number of columns that do not contain the same character twice

Example

- String Alignment for “FOOD” and “MONEY”:

F	O	O		D
M	O	N	E	Y

- It is not too hard to see that we cannot do better than four for the edit distance between “FOOD” and “MONEY”

Example II

- Unfortunately, it can be more difficult to compute the edit distance exactly. Example:

A	L	G	O	R		I		T	H	M
A	L		T	R	U	I	S	T	I	C

Key Observation

- If we remove the last column in an optimal alignment, the remaining alignment must also be optimal
- Easy to prove by contradiction: Assume there is some better subalignment of all but the last column. Then we can just paste the last column onto this better subalignment to get a better overall alignment.
- Note: The last column can be either: 1) a blank on top aligned with a character on bottom, 2) a character on top aligned with a blank on bottom or 3) a character on top aligned with a character on bottom

DP Solution

- To develop a DP algorithm for this problem, we first need to find a recursive definition
- Assume string A has length m and string B has length n
- Let $E(i, j)$ be the edit distance between the first i characters of A and the first j characters of B
- Then what we want to find is $E(m, n)$

Recursive Definition

- Say we want to compute $E(i, j)$ for some i and j
- Further say that the “Recursion Fairy” can tell us the solution to $E(i', j')$, for all $i' \leq i$, $j' \leq j$, *except* for $i' = i$ and $j' = j$
- Q: Can we compute $E(i, j)$ efficiently with help from our fairy friend?

Recursive Definition

There are three possible cases for the last column:

- **Blank on Bottom:** $E(i, j) = 1 + E(i - 1, j)$
- **Blank on Top:** $E(i, j) = 1 + E(i, j - 1)$
- **No Blank:** If $a_i = b_j$, $E(i, j) = E(i - 1, j - 1)$, else $E(i, j) = E(i - 1, j - 1) + 1$

Summary

Let $I(A[i] \neq B[j]) = 1$ if $A[i]$ and $B[j]$ are different, and 0 if they are the same. Then:

$$E(i, j) = \min \left\{ \begin{array}{l} E(i-1, j) + 1, \\ E(i, j-1) + 1, \\ E(i-1, j-1) + I(A[i] \neq B[j]) \end{array} \right\}$$

Base Case(s)

It is not too hard to see that:

- $E(0, j) = j$ for all j , since the j characters of B must be aligned with blanks
- Similarly, $E(i, 0) = i$ for all i

Naive Recursive Algorithm

- We now have enough information to create a recursive algorithm
- The runtime of this recursive algorithm would be given by the following recurrence:

$$T(m, 0) = T(0, n) = O(1)$$

$$T(m, n) = T(m, n - 1) + T(m - 1, n) + T(m - 1, n - 1) + O(1)$$

- In the worst case, $T(n, n) = 2^{\Omega(n)}$: the naive recursive algorithm takes exponential time.

Better Idea

- We can build an $(m + 1) \times (n + 1)$ table containing all values of $E(i, j)$
- We start by filling in the base cases for this table: the entries in the 0-th row and 0-th column
- To fill in any other entry, we need the values directly above, to the left, and diagonally above-left.
- Thus we can fill in the table in the standard way: left to right and top-to-bottom to ensure that the entries we need to fill in each cell are always available

Example Table

- Bold numbers indicate places where characters in the strings are equal
- Arrows represent predecessors that define each entry: horizontal arrow is deletion, vertical is insertion and diagonal is substitution.
- Bold diagonal arrows are “free” substitutions of a letter for itself
- Any path of arrows from the top-left to the bottom-right corner gives an optimal alignment (there are three paths in this example table, so there are three optimal edit sequences).

	A L G O R I T H M									
	0	→ 1	→ 2	→ 3	→ 4	→ 5	→ 6	→ 7	→ 8	→ 9
A	↓ 1	↘ 0	→ 1	→ 2	→ 3	→ 4	→ 5	→ 6	→ 7	→ 8
L	↓ 2	↓ 1	↘ 0	→ 1	→ 2	→ 3	→ 4	→ 5	→ 6	→ 7
T	↓ 3	↓ 2	↓ 1	↘ 1	→ 2	→ 3	→ 4	↘ 4	→ 5	→ 6
R	↓ 4	↓ 3	↓ 2	↓ 2	↘ 2	↘ 2	→ 3	→ 4	→ 5	→ 6
U	↓ 5	↓ 4	↓ 3	↘ 3	↘ 3	↘ 3	↘ 3	→ 4	→ 5	→ 6
I	↓ 6	↓ 5	↓ 4	↘ 4	↘ 4	↘ 4	↘ 4	↘ 3	→ 4	→ 5
S	↓ 7	↓ 6	↓ 5	↘ 5	↘ 5	↘ 5	↘ 5	↓ 4	↘ 4	↘ 5
T	↓ 8	↓ 7	↓ 6	↘ 6	↘ 6	↘ 6	↘ 6	↓ 5	↘ 4	→ 5
I	↓ 9	↓ 8	↓ 7	↘ 7	↘ 7	↘ 7	↘ 7	↓ 6	↓ 5	↘ 5
C	↓ 10	↓ 9	↓ 8	↘ 8	↘ 8	↘ 8	↘ 8	↓ 7	↓ 6	↘ 6

Pseudocode

Edit-Distance($A[1 \dots m], B[1 \dots n]$)

$E[0, 0] \leftarrow 0$

for $i \leftarrow 1$ **to** m **do** $E[i, 0] \leftarrow i$

for $j \leftarrow 1$ **to** n **do** $E[0, j] \leftarrow j$

for $i \leftarrow 1$ **to** m **do**

for $j \leftarrow 1$ **to** n **do**

$E[i, j] \leftarrow \min\{E[i - 1, j] + 1, E[i, j - 1] + 1,$
 $E[i - 1, j - 1] + I(A[i] \neq B[j])\}$

return $E[m, n]$

Analysis

- Let m be the length of the first string and n the length of the second string
- Then there are $\Theta(nm)$ entries in the table, and it takes $\Theta(1)$ time to fill each entry
- This implies that the runtime of the algorithm is $\Theta(nm)$
- Q: Can you find a faster algorithm?

Reconstructing optimal alignment

- In this code, we do not keep information needed to reconstruct an optimal alignment
- However, it is a simple matter to keep around another array which stores, for each cell, a pointer to the cell that was used to achieve the current cell's minimum edit distance
- To reconstruct a solution, we then need only follow these pointers from the bottom-right corner up to the top-left corner

In Class Exercise

- Create a string alignment table for the two strings “abba” and “bab”. Put “abba” at the top of the table and “bab” on the left side
- Q_i : ($i = 1, 2, \dots, 5$) What is the i -th row of your table?
- Q6: What is the minimum edit distance and how many alignments achieve it?

Take Away

- To solve the string alignment problem, we did the following:
1) formulated the problem recursively 2) built a solution to the recurrence from the bottom up
- Next we will see how a similar technique can be used to solve the matrix-chain multiplication problem.

Part II: Matrix Chains

Problem:

- We are given a sequence of n matrices, $A_1, A_2, \dots, A_n$, where for $i = 1, 2, \dots, n$, matrix A_i has dimension p_{i-1} by p_i
- We want to compute the product $A_1 A_2 \cdots A_n$ as quickly as possible.
- In particular, we want to fully *parenthesize* the expression above so there are no ambiguities about how the matrices are multiplied
- A product of matrices is *fully parenthesized* if it is either a single matrix, or the product of two fully parenthesized matrix products, surrounded by parentheses

_____ Parenthesizing Matrices _____

- There are many ways to parenthesize the matrices
- Each way gives the same output because matrix multiplication is associative
- However, the way we parenthesize will affect the *time* to compute the output
- Our goal: Find a parenthesization that requires the minimum number of scalar multiplications

Example

$$A = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{pmatrix}_{5 \times 1} \quad B = (b_1 \ b_2 \ b_3 \ b_4 \ b_5)_{1 \times 5} \quad C = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{pmatrix}_{5 \times 1}$$

- Multiplying the last two matrices first produces a 1×1 matrix:
 $A(BC)$ costs $1 \cdot 5 \cdot 1 + 5 \cdot 1 \cdot 1 = 10$.
- Multiplying the first two matrices first produces a 5×5 matrix:
 $(AB)C$ costs $5 \cdot 1 \cdot 5 + 5 \cdot 5 \cdot 1 = 50$.

A Problem

Problem: There can be many ways to parenthesize. For example:

- $(A_1(A_2(A_3A_4)))$
- $(A_1((A_2A_3)A_4))$
- $((A_1A_2)(A_3A_4))$
- $((A_1(A_2A_3))A_4)$
- $((((A_1A_2)A_3)A_4)$

A Problem

- Let $P(n)$ be the number of ways to parenthesize n matrices. Then $P(1) = 1$
- For $n \geq 2$, we know that a fully parenthesized product is the product of two fully parenthesized products, and the split can occur anywhere from $k = 1$ to $k = n - 1$.
- Hence for $n \geq 2$:

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$$

- You can show that the solution to this recurrence is $\Omega(2^n)$

Apply the Pattern

Q: Can we develop a dynamic-programming solution to this problem?

- We will reuse the pattern from slide 13: define the subproblems, derive a recurrence, and evaluate the subproblems in a valid bottom-up order.

Key Observation

- Let $A_{i..j}$ (for $i \leq j$) be the matrix that results from evaluating the product $A_i A_{i+1} \cdots A_j$
- Imagine we are computing $A_{i..j}$
- The last multiplication we do must look like this:

$$A_{i..j} = (A_{i..k}) * (A_{k+1..j})$$

for some k between i and $j - 1$

- Then total cost to compute $A_{i..j}$ is:

cost to compute $A_{i..k}$ +
cost to compute $A_{k+1..j}$ +
cost to multiply $A_{i..k}$ and $A_{k+1..j}$

Recursive Formulation

- For $1 \leq x \leq y \leq n$, let $m(x, y)$ be the minimum cost of computing $A_{x..y}$
- Then for any k between i and $j - 1$,
$$m(i, j) \leq \text{optimal cost to compute } A_{i..k} + \text{optimal cost to compute } A_{k+1..j} + \text{cost to multiply } A_{i..k} \text{ and } A_{k+1..j}$$
- In other words:

$$m(i, j) \leq m(i, k) + m(k + 1, j) + \text{cost to multiply } A_{i..k} \text{ and } A_{k+1..j}$$

Cost to Multiply

- $A_{i..k}$ is a p_{i-1} by p_k matrix
- $A_{k+1..j}$ is a p_k by p_j matrix
- Thus multiplying $A_{i..k}$ and $A_{k+1..j}$ takes $p_{i-1}p_kp_j$ operations
- Hence we have:

$$m(i, j) \leq m(i, k) + m(k+1, j) + p_{i-1}p_kp_j$$

Recursive Formulation

- We have shown that $m(i, j) \leq m(i, k) + m(k + 1, j) + p_{i-1}p_kp_j$ for any $k = i, i + 1, \dots, j - 1$
- Further note that the optimal parenthesization must use some value of $k = i, i + 1, \dots, j - 1$. So we need only pick the best
- Thus we have:

$$\begin{aligned} m(i, j) &= 0 \text{ if } i = j \\ m(i, j) &= \min_{i \leq k < j} \{m(i, k) + m(k + 1, j) + p_{i-1}p_kp_j\} \end{aligned}$$

The Recursive Algorithm

- We now have enough information to write a recursive function to solve the problem
- The recursive solution will have runtime given by the following recurrence:
- $T(1) = 1,$
- $T(n) = 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1)$
- Unfortunately, the solution to this recurrence is $\Omega(2^n)$

DP Algorithm

- Note that we must solve one subproblem for each choice of i and j satisfying $1 \leq i \leq j \leq n$
- This is only $\binom{n}{2} + n = \Theta(n^2)$ subproblems
- The recursive algorithm encounters each subproblem many times in the branches of the recursion tree.
- However, we can just compute these subproblems from the bottom up, storing the results in a table (this is the DP solution)

Pseudocode

Matrix-Chain-Order($p[0 \dots n]$)

for $i \leftarrow 1$ **to** n **do** $m[i, i] \leftarrow 0$

for $\ell \leftarrow 2$ **to** n **do**

for $i \leftarrow 1$ **to** $n - \ell + 1$ **do**

$j \leftarrow i + \ell - 1$; $m[i, j] \leftarrow \infty$

for $k \leftarrow i$ **to** $j - 1$ **do**

$q \leftarrow m[i, k] + m[k + 1, j] + p[i - 1]p[k]p[j]$

if $q < m[i, j]$ **then**

$m[i, j] \leftarrow q$; $s[i, j] \leftarrow k$

return $(m[1, n], s)$

Pseudocode

- This code computes both the optimal cost and a parenthesization that achieves that cost
- It uses $m[i, j]$ to store the optimal cost and $s[i, j]$ to store a split k that achieves that cost.
- To recover a parenthesization, recursively print A_i when $i = j$; otherwise print the solutions for $(i, s[i, j])$ and $(s[i, j] + 1, j)$ inside parentheses.

Analysis

- This code has three nested loops, each of which takes on at most $n - 1$ values, and the inner loop takes $O(1)$ time.
- Thus the runtime is $\Theta(n^3)$
- The algorithm also requires $\Theta(n^2)$ space

Example

- Consider the sequence of three matrices, A_1, A_2, A_3 whose dimensions are given by the sequence 3, 1, 2, 1 (i.e. $p_0 = 3, p_1 = 1, p_2 = 2, p_3 = 1$)
- Let's construct the tables giving the optimal parenthesization
- The (i, j) entry of the first table will give the optimal cost for computing $A_{i..j}$, the (i, j) entry of the second table will give a k value which achieves this optimal cost

Computations

- $m(1, 1) = m(2, 2) = m(3, 3) = 0$
- $m(1, 2) = p_0 p_1 p_2 = 6$
- $m(2, 3) = p_1 p_2 p_3 = 2$

Computations

$$\begin{aligned} m(1, 3) &= \min \left\{ \begin{array}{l} m(1, 1) + m(2, 3) + p_0 p_1 p_3, \\ m(1, 2) + m(3, 3) + p_0 p_2 p_3 \end{array} \right\} \\ &= \min \left\{ \begin{array}{l} 0 + 2 + 3, \\ 6 + 0 + 6 \end{array} \right\} \\ &= 5 \end{aligned}$$

Example, m array

	1	2	3
1	0	6	5
2	-	0	2
3	-	-	0

Example, s array

	1	2	3
1	-	1	1
2	-	-	2
3	-	-	-

Example

- Thus an optimal parenthesization is $(A_1(A_2A_3))$
- The cost of this is 5

Example II

- Consider the sequence of four matrices, A_1, A_2, A_3, A_4 whose dimensions are given by the sequence 3, 1, 2, 1, 2 (i.e. $p_0 = 3, p_1 = 1, p_2 = 2, p_3 = 1, p_4 = 2$)
- Let's construct the tables giving the optimal parenthesization
- The (i, j) entry of the first table will give the optimal cost for computing $A_{i..j}$, the (i, j) entry of the second table will give a k value which achieves this optimal cost

Example II, m array

	1	2	3	4
1	0	6	5	10
2	-	0	2	4
3	-	-	0	4
4	-	-	-	0

Example II, s array

	1	2	3	4
1	-	1	1	1
2	-	-	2	3
3	-	-	-	3
4	-	-	-	-

Example Computation

$$\begin{aligned} m(1, 4) &= \min \left\{ \begin{array}{l} m(1, 1) + m(2, 4) + p_0 p_1 p_4, \\ m(1, 2) + m(3, 4) + p_0 p_2 p_4, \\ m(1, 3) + m(4, 4) + p_0 p_3 p_4 \end{array} \right\} \\ &= \min \left\{ \begin{array}{l} 0 + 4 + 6, \\ 6 + 4 + 12, \\ 5 + 0 + 6 \end{array} \right\} \\ &= 10 \end{aligned}$$

This minimum is achieved when $k = 1$

Example II

- Thus an optimal parenthesization is $(A_1((A_2A_3)A_4))$
- The cost of this is 10

In-Class Exercise

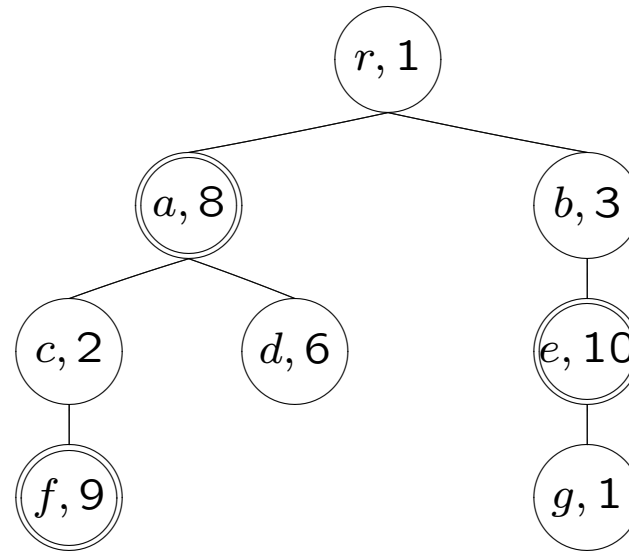
- Consider the sequence of three matrices, A_1, A_2, A_3 whose dimensions are given by the sequence 1, 2, 1, 2 (i.e. $p_0 = 1$, $p_1 = 2$, $p_2 = 1$, $p_3 = 2$)
- Q1: What are the m array and s array for these inputs?
- Q2: What is the optimal parenthesization?

Part III: Tree Independent Set

Problem:

- We are given a rooted tree $T = (V, E)$ and a nonnegative weight $w(v)$ for every vertex $v \in V$
- An *independent set* is a set $S \subseteq V$ containing no pair of adjacent vertices
- The weight of S is $w(S) = \sum_{v \in S} w(v)$
- Goal: Find an independent set of maximum total weight

Example



The double circles mark the maximum-weight independent set $\{a, e, f\}$, with weight 27. It includes only some vertices from each non-root level.

Brute Force

- We could enumerate every subset $S \subseteq V$
- For each subset, check whether it is independent and, if so, compute its total weight
- Q: How many subsets are there?
- A: There are $2^{|V|}$ subsets, so this approach takes exponential time

Apply the Pattern

Root the tree at r . For each vertex v , let T_v be the subtree rooted at v .

- If we include v , we must exclude every child of v
- If we exclude v , then for each child we may independently choose whether to include or exclude that child
- Thus the right subproblems are determined both by a subtree and by whether its root is included

Recursive Formulation

For each vertex v , define two quantities:

- $I(v)$: maximum weight of an independent set in T_v that *includes* v
- $O(v)$: maximum weight of an independent set in T_v that *omits* v
- If r is the root, the maximum weight independent set is:

$$\max\{I(r), O(r)\}.$$

Recursive Formulation

Let $C(v)$ denote the children of v . Then

$$I(v) = w(v) + \sum_{u \in C(v)} O(u)$$

and

$$O(v) = \sum_{u \in C(v)} \max\{I(u), O(u)\}.$$

For a leaf v , these formulas give $I(v) = w(v)$ and $O(v) = 0$.

Why the Recurrence Is Correct

We use induction on the size of T_v .

- If an independent set includes v , it must exclude each child u ; the best possible contribution from T_u is therefore $O(u)$
- If it omits v , the solution in each child subtree may either include or omit the child, whichever is better
- Different child subtrees have no edges between them, so their optimal solutions can be combined independently
- Therefore the two formulas give exactly the best solutions in the two possible states

The Recursive Algorithm

- We now have enough information to write mutually recursive functions for $I(v)$ and $O(v)$
- A direct implementation can compute the same state many times: for example, $O(u)$ may be requested while computing both $I(v)$ and $O(v)$
- On a path, this repeated work leads to a Fibonacci-like exponential recurrence
- But there are only $2|V|$ distinct states

DP Algorithm

- Process the vertices in *postorder*, so every child is processed before its parent
- Store $I(v)$ and $O(v)$ after computing them
- When processing v , all the values required by the recurrence are already available
- This is a bottom-up dynamic program on the tree

Pseudocode

Tree-MWIS(T, r, w)

for each v **in a postorder traversal of** T **do**

$I[v] \leftarrow w(v); O[v] \leftarrow 0$

for each child u **of** v **do**

$I[v] \leftarrow I[v] + O[u]$

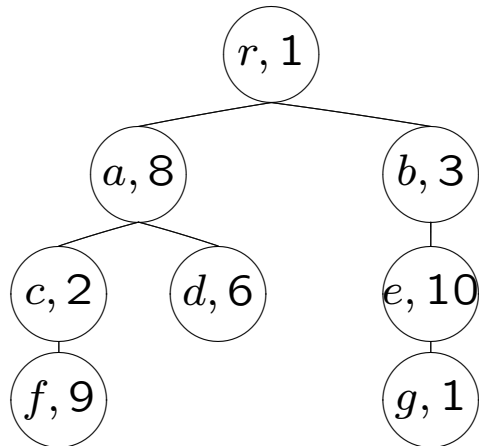
$O[v] \leftarrow O[v] + \max\{I[u], O[u]\}$

return $\max\{I[r], O[r]\}$

Analysis

- There are two states for each vertex, so the tables use $\Theta(|V|)$ space
- Each vertex is processed once
- The inner loop examines each parent–child edge once
- A tree with $|V|$ vertices has $|V| - 1$ edges
- Therefore the algorithm runs in $\Theta(|V|)$ time

Example Computation



v	f	d	g	c	e	a	b	r
$w(v)$	9	6	1	2	10	8	3	1
$I(v)$	9	6	1	2	10	17	4	26
$O(v)$	0	0	0	9	1	15	10	27

Maximum weight: $\max\{I(r), O(r)\} = 27$.

Reconstructing an Optimal Set

Start at the root and work downward.

- If a vertex is included, omit all its children
- If a vertex is omitted, choose the better of $I(u)$ and $O(u)$ for each child u
- Continue recursively until reaching the leaves
- Ties may be broken arbitrarily; different choices can produce different optimal independent sets
- For our example this yields $\{a, e, f\}$

Take Away

- We have seen four dynamic-programming algorithms
- In each case, we did the following 1) found a recurrence for the solution 2) built solutions to the recurrence from the bottom up
- DP tables need not be rectangular: for the tree problem, the subproblems followed the structure of the input tree
- You should be prepared to do this on your own now!