COMP 251

Algorithms & Data Structures (Winter 2021)

Algorithm Paradigms – Dynamic Programming 2

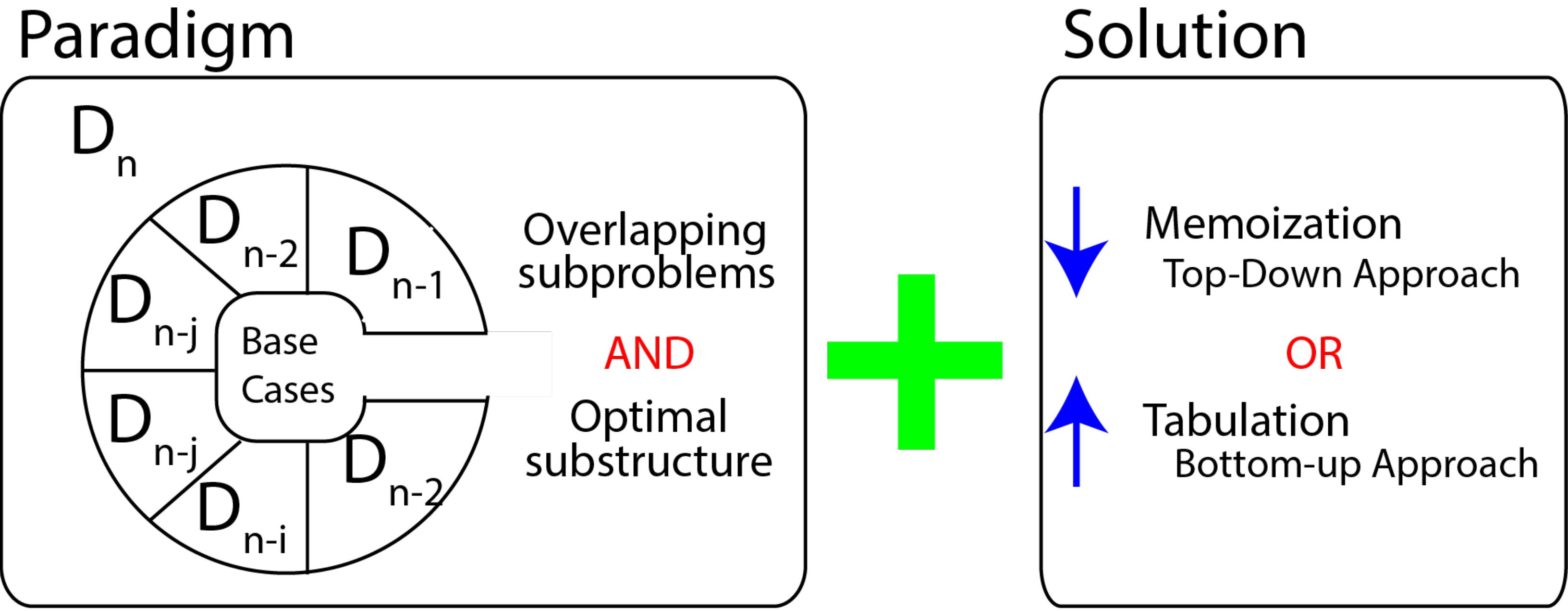
School of Computer Science
McGill University

Slides of (Comp321 ,2021), Langer (2014), Kleinberg & Tardos, 2005 & Cormen et al., 2009, Jaehyun Park' slides CS 97SI, Top-coder tutorials, T-414-AFLV Course, Programming Challenges books.

Outline

- Complete Search
- Divide and Conquer.
- Dynamic Programming.
  - Introduction.
  - Examples.
- Greedy.

Dynamic Programming– Take home picture



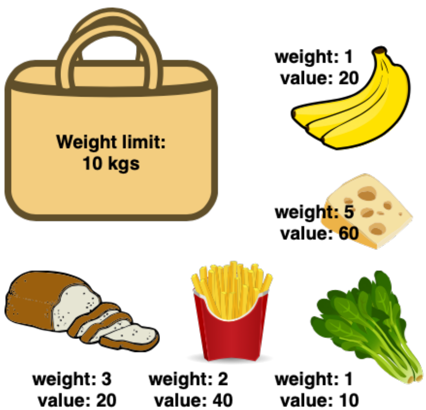
Dynamic Programming– 2D - Knapsack

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ and has value $v_i > 0$.
- Knapsack has capacity of W.
- Goal: fill knapsack so as to maximize total value.

Ex. { 1, 2, 5 } has value 35.
Ex. { 3, 4 } has value 40.
Ex. { 3, 5 } has value 46 (but exceeds weight limit).

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance
(weight limit W = 11)



Taken form baeldung.com

Dynamic Programming– 2D - Knapsack

Step 1: Identify the sub-problems (in words).

Step 1.1: Identify the possible sub-problems.

Let OPT(i) be the maximum total value of items 1 to i (i.e., value of the optimal solution to the problem including activities 1 to i).

I just copy the same definition used for the weighted interval scheduling

- Let OPT(i) be the maximum total weight of compatible activities 1 to i (i.e., value of the optimal solution to the problem including activities 1 to i).

Dynamic Programming– 2D - Knapsack

Step 2: Find the recurrence.

Step 2.1: What decision do I make at every step?.

Case 1: OPT does not select (activity) item *i*

- Must include optimal solution on other (activities) items *{1, 2, …, i-1}*.

Case 2: OPT selects (activity) item i

- (activity) Add weight w_i -- (item) Add weight w_i and value v_i
- (activity) Cannot use incompatible activities – (item) ??
- (activity) Must include optimal solution on remaining compatible activities {1, 2, … , p(j) }. -- (item) ??
  - Selecting item i does not immediately imply that we will have to reject other items
  - Without knowing what other items were selected before i, we do not even know if we have enough room for i.

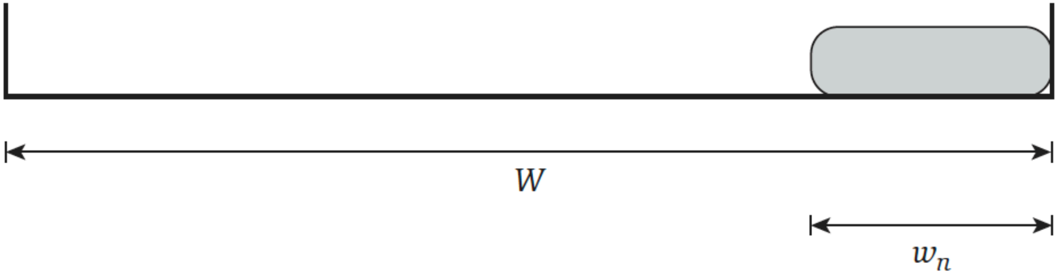
Dynamic Programming– 2D - Knapsack

Step 2: Find the recurrence.

Case 2: OPT selects (activity) item i

- Selecting item i does not immediately imply that we will have to reject other items
- Without knowing what other items were selected before i, we do not even know if we have enough room for i.

Conclusion: We need more subproblems!!!!!



After item *n* is included in the solution, a weight of w_n is used up and there is $W - w_n$ available weight left

Dynamic Programming– 2D - Knapsack

Step 1: Identify the sub-problems (in words).

Step 1.1: Identify the possible sub-problems.

Let OPT(i, w) be the maximum profit subset of items 1 to i with weight limit w.

Dynamic Programming– 2D - Knapsack

Step 2: Find the recurrence.

Case 1: OPT does not select item *i*

- OPT selects best of {1, 2, …, i-1} using weight limit w.

Optimal substructure property

Case 2: OPT selects item *i*
- New weight limit = $w - w_i$
- OPT selects best of {1, 2, … , i-1 } using this new weight limit.

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{ OPT(i-1,w), \quad v_i + OPT(i-1,w-w_i) \} & \text{otherwise} \end{cases}$$

Dynamic Programming– 2D - Knapsack

KNAPSACK $(n, W, w_1, \ldots, w_n, v_1, \ldots, v_n)$

FOR $w = 0$ TO W

  $M[0, w] \leftarrow 0.$

FOR $i = 1$ TO n

  FOR $w = 1$ TO W

  IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w].$

  ELSE $M[i, w] \leftarrow \max \{ M[i-1, w],\ v_i + M[i-1, w - w_i] \}.$

RETURN $M[n, W].$

Dynamic Programming– 2D - Knapsack

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Max weight W = 11

Dynamic Programming– 2D - Knapsack

w

i

M	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0											
{1,2}	0											
{1,2,3}	0											
{1,2,3,4}	0											
{1,2,3,4,5}	0											

Dynamic Programming– 2D - Knapsack

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W = 11

FOR $i = 1$ TO n

  FOR $w = 1$ TO W

  IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w]$.

  ELSE $M[i, w] \leftarrow \max\ \{ M[i-1, w],\ v_i + M[i-1, w-w_i] \}$

M	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1,2}	0											
{1,2,3}	0											
{1,2,3,4}	0											
{1,2,3,4,5}	0											

Dynamic Programming– 2D - Knapsack

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W = 11

FOR $i = 1$ TO n

FOR $w = 1$ TO W

IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w]$.

ELSE $M[i, w] \leftarrow \max \{ M[i-1, w], v_i + M[i-1, w - w_i] \}$

M	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1,2}	0	1										
{1,2,3}	0											
{1,2,3,4}	0											
{1,2,3,4,5}	0											

Dynamic Programming– 2D - Knapsack

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W = 11

FOR $i = 1$ TO n

  FOR $w = 1$ TO W

  IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w]$.

  ELSE $M[i, w] \leftarrow \max \{ M[i-1, w],\ v_i + M[i-1, w - w_i] \}$

M	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
		1	6									
{1,2,3}	0											
{1,2,3,4}	0											
{1,2,3,4,5}	0											

V_2+M(i-1,w-w_2)

M(i-1,w)

Dynamic Programming– 2D - Knapsack

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W = 11

FOR $i = 1$ TO n

  FOR $w = 1$ TO W

  IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w]$.

  ELSE $M[i, w] \leftarrow \max \{ M[i-1, w],\ v_i + M[i-1, w-w_i] \}$

M	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1,2}			6	7								
{1,2,3}	0											
{1,2,3,4}	0											
{1,2,3,4,5}	0											

$V_2+M(i-1,w-w_2)$

$M(i-1,w)$

Dynamic Programming– 2D - Knapsack

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W = 11

FOR $i = 1$ TO n

FOR $w = 1$ TO W

IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w]$.

ELSE $M[i, w] \leftarrow \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \}$

M	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1,2}	0	1	6	7	7	7	7	7	7	7	7	7
{1,2,3}	0											
{1,2,3,4}	0											
{1,2,3,4,5}	0											

Dynamic Programming– 2D - Knapsack

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W = 11

FOR $i = 1$ TO n

FOR $w = 1$ TO W

IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w]$.

ELSE $M[i, w] \leftarrow \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \}$

M	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1,2}	0	1	6	7	7	7	7	7	7	7	7	7
{1,2,3}	0	1	6	7	7	18	19	24	25	25	25	25
{1,2,3,4}	0											
{1,2,3,4,5}	0											

Dynamic Programming– 2D - Knapsack

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W = 11

FOR $i = 1$ TO n

  FOR $w = 1$ TO W

  IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w]$.

  ELSE $M[i, w] \leftarrow \max \{ M[i-1, w], v_i + M[i-1, w - w_i] \}$

M	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1,2}	0	1	6	7	7	7	7	7	7	7	7	7
{1,2,3}	0	1	6	7	7	18	19	24	25	25	25	25
{1,2,3,4}	0	1	6	7	7	18	22	24	28	29	29	40
{1,2,3,4,5}	0											

Dynamic Programming– 2D - Knapsack

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W = 11

FOR $i = 1$ TO n

  FOR $w = 1$ TO W

  IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w]$.

  ELSE $M[i, w] \leftarrow \max \{ M[i-1, w],\ v_i + M[i-1, w - w_i] \}$

M	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0						0	0	0	0	0
{1}	0	1										
{1,2}	0	1	6	7	7	7	7					
{1,2,3}	0	1	6	7	7	18	19	24	25	25	25	25
{1,2,3,4}	0	1	6	7	7	18	22	24	28	29	29	40
{1,2,3,4,5}	0	1	6	7	7	18	22	28	29	34	35	40

Item 3 in solution

Item 4 in solution

Dynamic Programming– 2D - Knapsack

Theorem. There exists an algorithm to solve the knapsack problem with n items and maximum weight W in $\Theta(n\ W)$ time and $\Theta(n\ W)$ space.

weights are integers between 1 and W

Pf.

- Takes $O(1)$ time per table entry.
- There are $\Theta(n\ W)$ table entries. "pseudo-polynomial"
- After computing optimal values, can trace back to find solution: take item i in $OPT(i, w)$ iff $M[i, w] < M[i-1, w]$. ▪

Dynamic Programming– 2D

Problem: given two strings x and y, find the longest common subsequence (LCS) and print its length.

Example:

- x : A**BC**BD**AB**
- y : **B**D**CAB**C
- "BCAB" is the longest subsequence found in both sequences, so the answer is 4.

Dynamic Programming– 2D

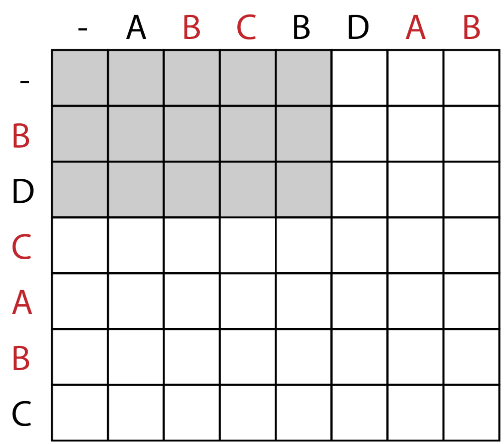
Step 1: Identify the sub-problems (in words).

Step 1.1: Identify the original problem.

Let C_{nm} be the length of the LCS of $x_{1..n}$ and $y_{1..m}$

Step 1.2: Identify the possible sub-problems.

Let C_{ij} be the length of the LCS of $x_{1..i}$ and $y_{1..j}$



Dynamic Programming– 2D

Step 2: Find the recurrence.

Step 2.1: What decision do I make at every step?.

Two options. To contribute to the LCS length or not.

- If $x_i = y_j$, they both contribute to the LCS => match
- If $x_i \neq y_j$, either x_i or y_j does not contribute to the LCS, so one can be dropped

Dynamic Programming– 2D

Step 2: Find the recurrence.

Step 2.1: What decision do I make at every step?.

Two options. To contribute to the LCS length or not.

- If $x_i = y_j$, they both contribute to the LCS => match
- If x_i != y_j , either x_i or y_j does not contribute to the LCS, so one can be dropped

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Optimal substructures

Dynamic Programming– 2D

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

- If $z_k \neq x_m$, then we could append $x_m = y_n$ to *Z* to obtain a common subsequence of *X* and *Y* of length $k + 1$, contradicting the supposition that *Z* is a LCS of *X* and *Y*.
- The prefix Z_{k-1} is a common subsequence of X_{m-1} and Y_{n-1} with length *k-1*. We wish to show that it is an LCS.
  - Suppose for the purpose of contradiction that there exists a common subsequence *W* of X_{m-1} and Y_{n-1} with length greater than *k-1.* Then, appending $x_m = y_n$ to produce *W* produces a common subsequence of *X* and *Y* whose length is greater than *k*, which is a contradiction.

Dynamic Programming– 2D

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.

- If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. If there were a common subsequence W of X_{m-1} and Y with length greater than k, then W would also be a common subsequence of X_m and Y, contradicting the assumption that Z is an LCS of X and Y.

Dynamic Programming– 2D

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

- Overlapping
  - To find an LCS of X and Y, we may need to find the LCSs of X and Y_{n-1} and of X_{m-1} and Y. But each of these subproblems has the subsubproblem of finding an LCS of X_{m-1} and Y_{n-1}.

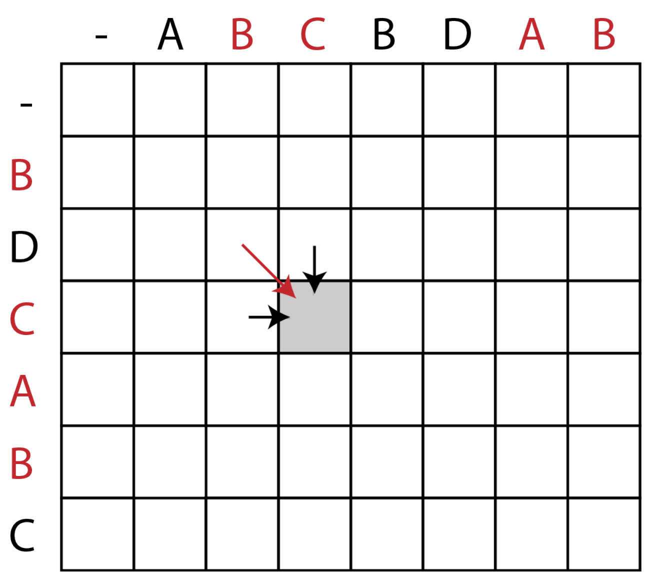
Dynamic Programming– 2D

Step 2: Find the recurrence.

Step 2.1: What decision do I make at every step?.

Two options. To contribute to the LCS length or not.

- If $x_i = y_j$, they both contribute to the LCS => match
- If $x_i \neq y_j$, either x_i or y_j does not contribute to the LCS, so one can be dropped



Dynamic Programming– 2D

Step 2: Find the recurrence.

- If $x_i = y_j$, they both contribute to the LCS => match
  - $C_{ij} = C_{i-1,j-1} + 1$
- Otherwise, either x_i or y_j does not contribute to the LCS, so one can be dropped
  - $C_{ij} = \max\{C_{i-1,j}, C_{i,j-1}\}$

Step 3: Recognize and solve the base cases.

- $C_{i0} = C_{0j} = 0$.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

Dynamic Programming– 2D

Step 4: Implement a solving methodology.

```
for(i=0;i<=n;i++) c[i][0]=0;
for(j=0;j<=m;j++) c[0][j]=0;
for(i=1;i<=n;i++){
   for(j=1;j<=m;j++){
       if(x[i]==y[j])
           c[i][j]=c[i-1][j-1]+1;
        else
           c[i][j]=max(c[i-1][j],c[i][j-1])
   }
}
```

Dynamic Programming– 2D

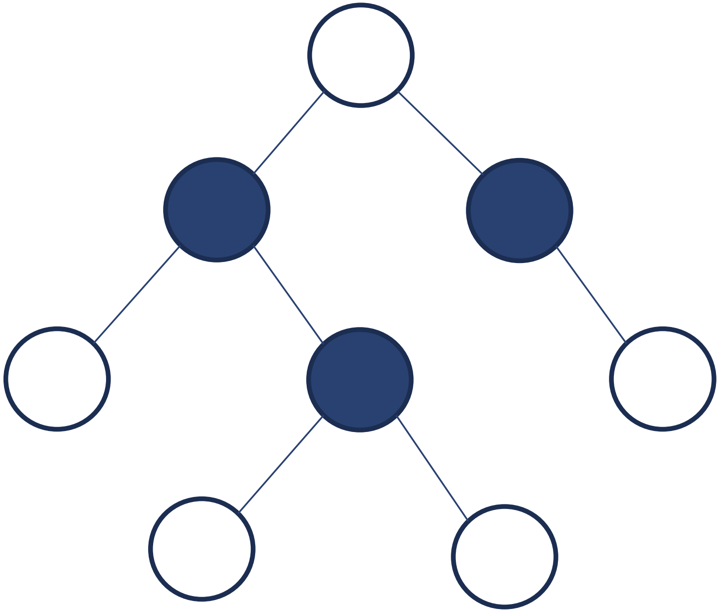
Step 4: Implement a solving methodology.

	-	A	B	C	B	D	A	B
-	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	0	1	2	2	2	3	3
B	0	0	1	2	2	2	3	4
C	0	0	1	2	2	2	3	4

Dynamic Programming– trees

Problem: given a tree, find the size of the **L**argest **I**ndependent **S**et (LIS). A set of nodes is an independent set if there are no edges between the nodes.

Example:



The largest independent set (LIS) is in white and size of the LIS is 5.

Dynamic Programming– trees

Step 1: Identify the sub-problems (in words).

Step 1.1: Identify the original problem.

MIS(r) denote the size of the largest independent set in the tree with root at r.

Step 1.2: Identify the possible sub-problems.

MIS(v) denote the size of the largest independent set in the subtree rooted at v.

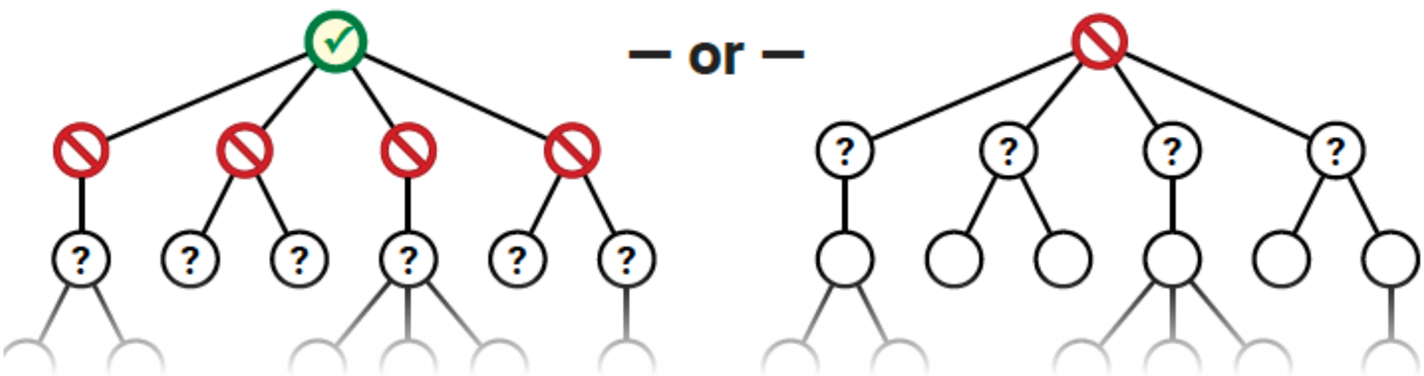
Dynamic Programming– 2D

Step 2: Find the recurrence.

Step 2.1: What decision do I make at every step?.

Two options.

- Include the node.
  - A set that includes *v* necessarily excludes all of v's children.
- Do not include the current node (root).
  - Any independent set is the union of independent sets in the subtrees rooted at the children of *v*.



Dynamic Programming– 2D

Step 2: Find the recurrence.

Step 2.1: What decision do I make at every step?.

Two options.

- Include the node.
  - A set that includes *v* necessarily excludes all of v's children.
- Do not include the current node (root).
  - Any independent set is the union of independent sets in the subtrees rooted at the children of *v*.

$$MIS(v) = \max\left\{ \sum_{w \downarrow v} MIS(w),\ 1 + \sum_{w \downarrow v} \sum_{x \downarrow w} MIS(x) \right\}$$

notation $w \downarrow v$ means "w is a child of v"

children w of v

grandchildren x of v

Dynamic Programming– 2D

$$MIS(v) = \max\left\{ \sum_{w\downarrow v} MIS(w),\ 1 + \sum_{w\downarrow v}\sum_{x\downarrow w} MIS(x) \right\}$$

Step 4: Implement a solving methodology.

- What data structure should we use to memoize this recurrence?
  - Array, 2D array, tree?
- What's a good order to consider the subproblems?
  - The subproblems associated with any node *v* depends on the subproblems associated with the children and grandchildren of *v*.
    - We can visit the nodes in any order we like, provided that every vertex is visited before its parent.
      - Pre-order? In-Order? Post-Order?

Dynamic Programming– 2D

Step 4: Implement a solving methodology.

- What data structure should we use to memoize this recurrence?
  - The most natural choice is the tree itself! Specifically, for each vertex v, we store the result of *MIS(v)* in a field *v.MIS*
- What's a good order to consider the subproblems?
  - The subproblems associated with any node *v* depends on the subproblems associated with the children and grandchildren of *v*.
    - We can visit the nodes in any order we like, provided that every vertex is visited before its parent.
      - Post-Order traversal.

Dynamic programming is *not* about filling in tables.
It's about smart recursion!

Dynamic Programming– 2D

Step 4: Implement a solving methodology.

- We can derive an even simpler algorithm by defining two separate functions over the nodes of the tree.
  - Let MISyes(v) denote the size of the largest independent set of the subtree rooted at v that includes v.
  - Let MISno(v) denote the size of the largest independent set of the subtree rooted at v that excludes v.

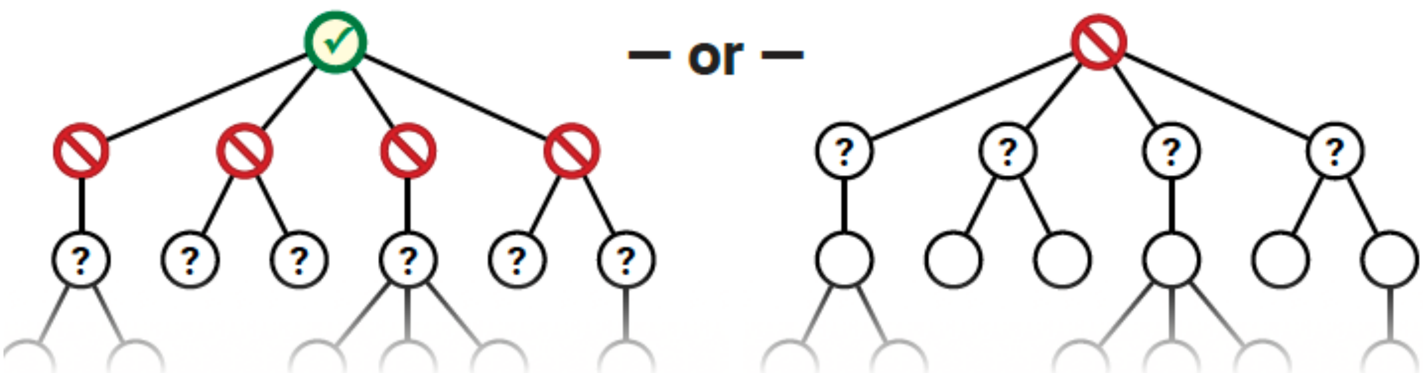
$$MISyes(v) = 1 + \sum_{w \downarrow v} MISno(w)$$

$$MISno(v) = \sum_{w \downarrow v} \max \{MISyes(w), MISno(w)\}$$

Dynamic Programming– 2D

$$MISyes(v) = 1 + \sum_{w \downarrow v} MISno(w)$$

$$MISno(v) = \sum_{w \downarrow v} \max\{MISyes(w), MISno(w)\}$$



Dynamic Programming– 2D

$$MISyes(v) = 1 + \sum_{w \downarrow v} MISno(w)$$

$$MISno(v) = \sum_{w \downarrow v} \max\{MISyes(w), MISno(w)\}$$

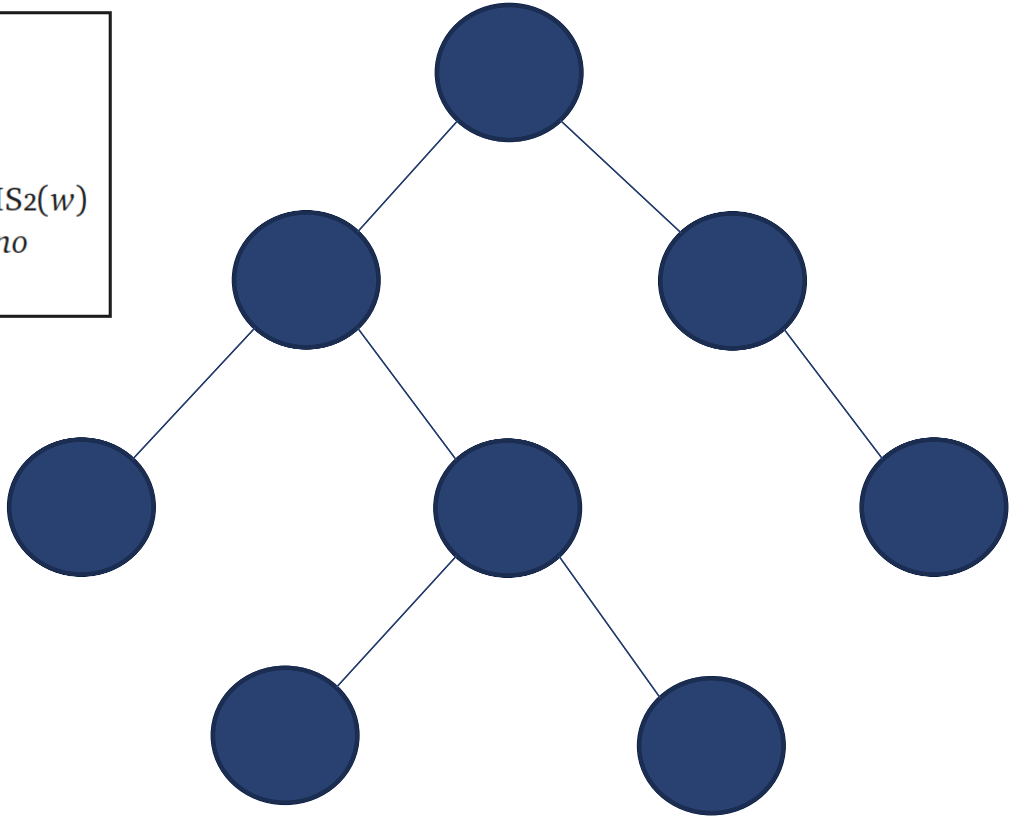
<u>$\text{TreeMIS2}(v)$:</u>

$v.MISno \leftarrow 0$
$v.MISyes \leftarrow 1$
for each child w of v
    $v.MISno \leftarrow v.MISno + \text{TreeMIS2}(w)$
    $v.MISyes \leftarrow v.MISyes + w.MISno$
return $\max\{v.MISyes, v.MISno\}$

Dynamic Programming– 2D

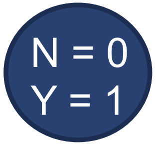
<u>TreeMIS2(v):</u>

$v.MISno \leftarrow 0$
$v.MISyes \leftarrow 1$
for each child w of v
    $v.MISno \leftarrow v.MISno + \text{TreeMIS2}(w)$
    $v.MISyes \leftarrow v.MISyes + w.MISno$
return $\max\{v.MISyes, v.MISno\}$



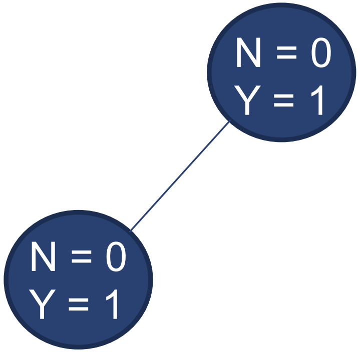
Dynamic Programming– 2D

```
TreeMIS2(v):
    v.MISno ← 0
    v.MISyes ← 1
    for each child w of v
        v.MISno ← v.MISno + TreeMIS2(w)
        v.MISyes ← v.MISyes + w.MISno
    return max{v.MISyes, v.MISno}
```



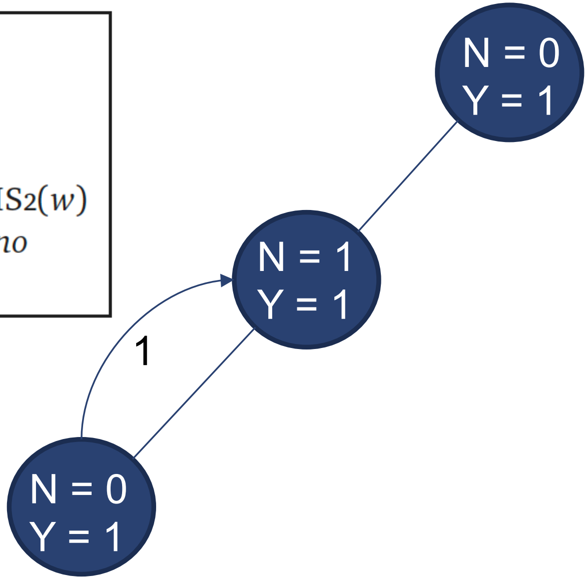
Dynamic Programming– 2D

```
TreeMIS2(v):
    v.MISno ← 0
    v.MISyes ← 1
    for each child w of v
        v.MISno ← v.MISno + TreeMIS2(w)
        v.MISyes ← v.MISyes + w.MISno
    return max{v.MISyes, v.MISno}
```



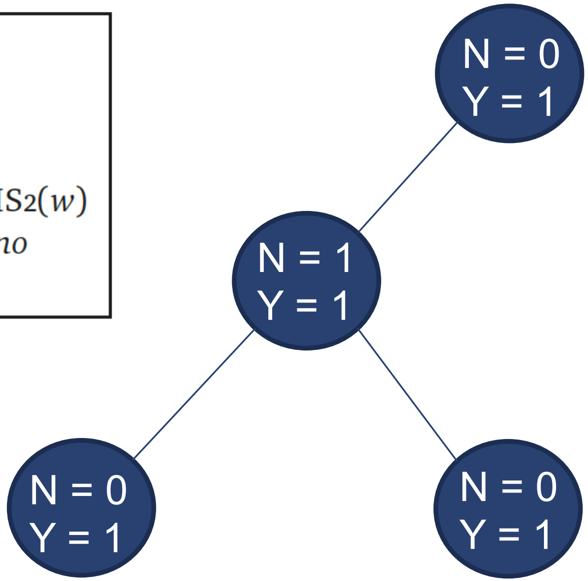
Dynamic Programming– 2D

<u>TreeMIS2(v):</u>
$v.MISno \leftarrow 0$
$v.MISyes \leftarrow 1$
for each child w of v
  $v.MISno \leftarrow v.MISno + \text{TreeMIS2}(w)$
  $v.MISyes \leftarrow v.MISyes + w.MISno$
return $\max\{v.MISyes, v.MISno\}$



Dynamic Programming– 2D

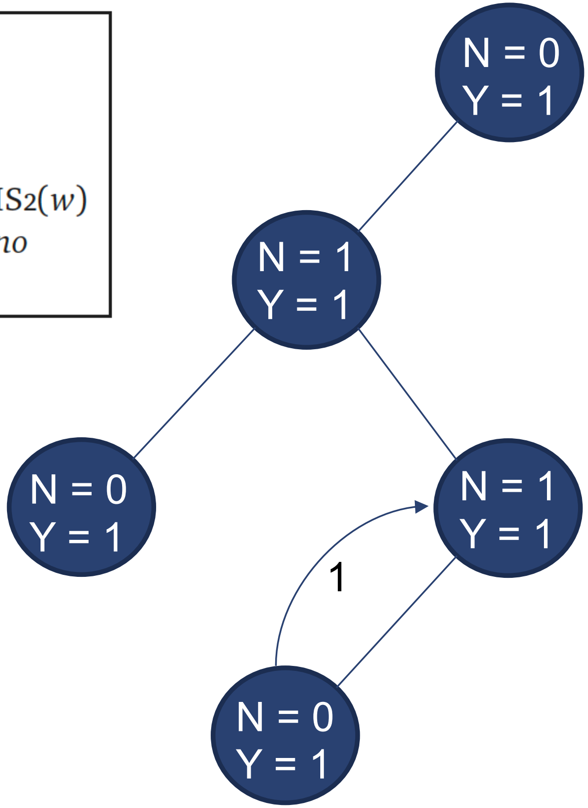
```
TREEMIS2(v):
  v.MISno ← 0
  v.MISyes ← 1
  for each child w of v
      v.MISno ← v.MISno + TREEMIS2(w)
      v.MISyes ← v.MISyes + w.MISno
  return max{v.MISyes, v.MISno}
```



Dynamic Programming– 2D

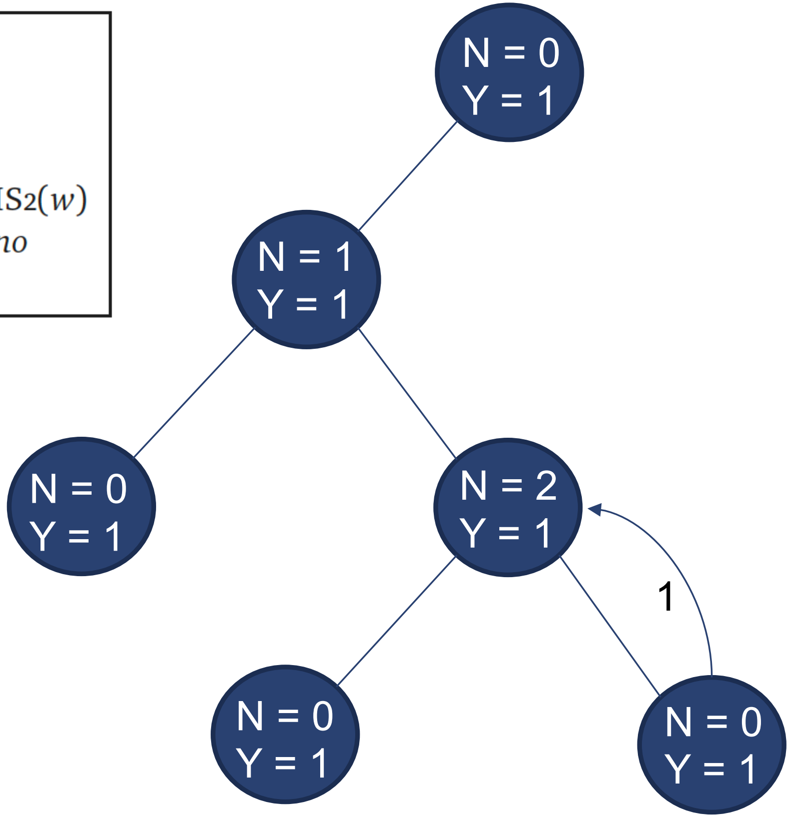
<u>TREEMIS2(v):</u>

$v.MISno \leftarrow 0$
$v.MISyes \leftarrow 1$
for each child w of v
  $v.MISno \leftarrow v.MISno + \text{TREEMIS2}(w)$
  $v.MISyes \leftarrow v.MISyes + w.MISno$
return $\max\{v.MISyes, v.MISno\}$



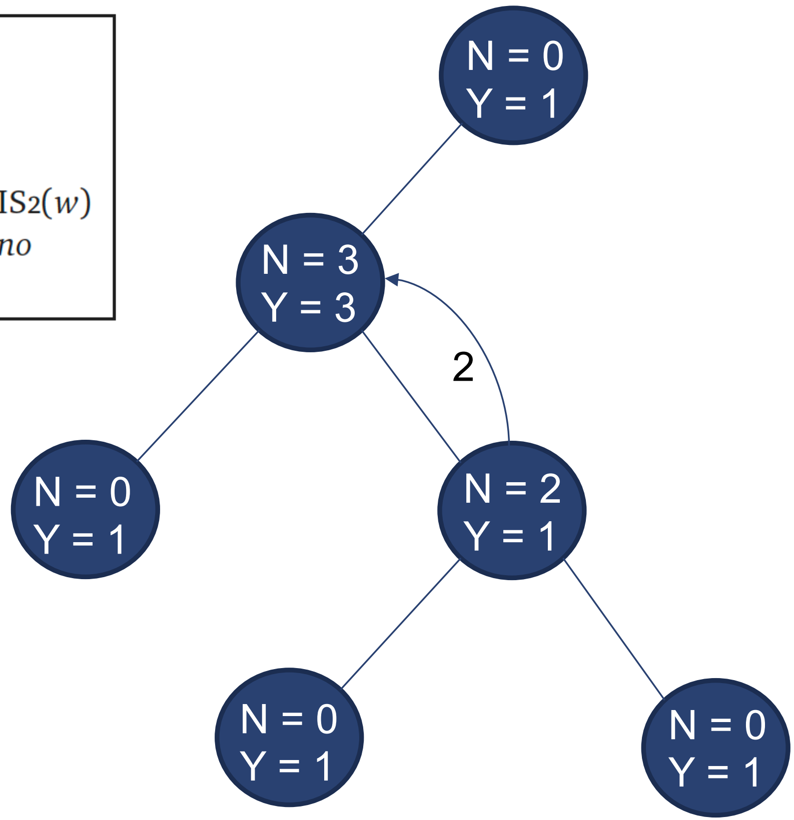
Dynamic Programming– 2D

$\underline{\text{TREEMIS2}(v)\text{:}}$
  $v.MISno \leftarrow 0$
  $v.MISyes \leftarrow 1$
  for each child w of v
    $v.MISno \leftarrow v.MISno + \text{TREEMIS2}(w)$
    $v.MISyes \leftarrow v.MISyes + w.MISno$
  return $\max\{v.MISyes, v.MISno\}$



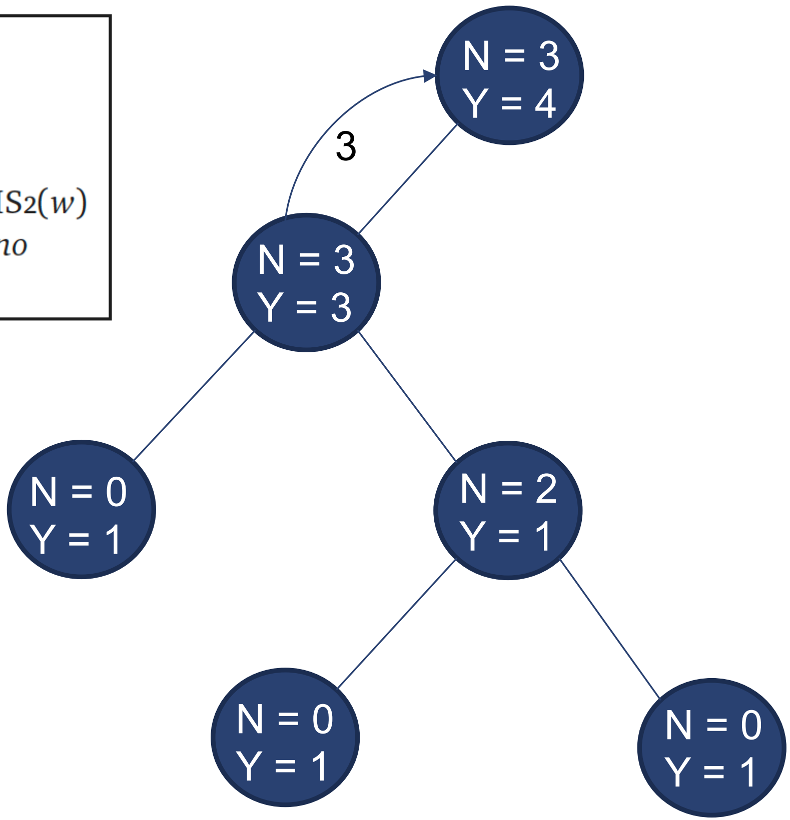
Dynamic Programming– 2D

```
TREEMIS2(v):
    v.MISno ← 0
    v.MISyes ← 1
    for each child w of v
        v.MISno ← v.MISno + TREEMIS2(w)
        v.MISyes ← v.MISyes + w.MISno
    return max{v.MISyes, v.MISno}
```



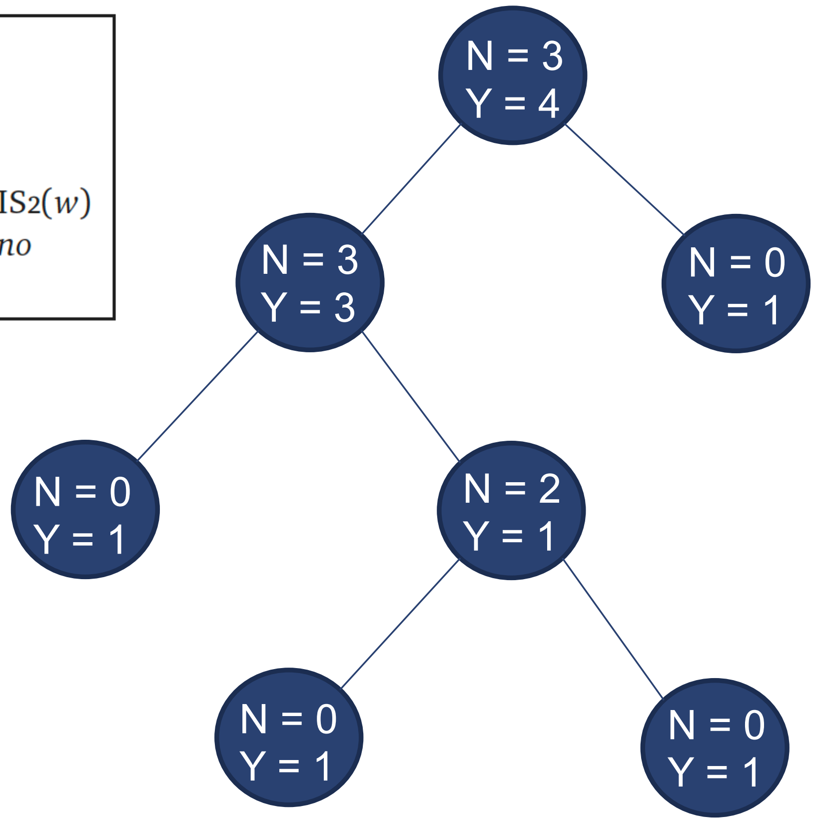
Dynamic Programming– 2D

```
TREEMIS2(v):
  v.MISno ← 0
  v.MISyes ← 1
  for each child w of v
      v.MISno ← v.MISno + TREEMIS2(w)
      v.MISyes ← v.MISyes + w.MISno
  return max{v.MISyes, v.MISno}
```



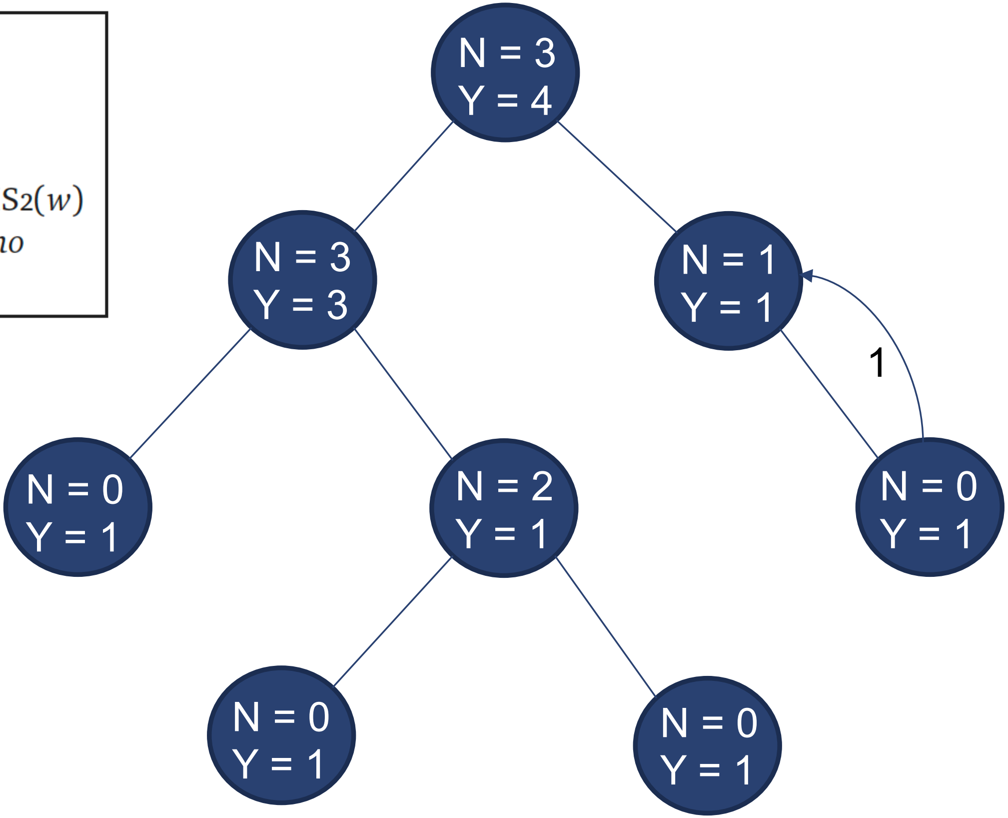
Dynamic Programming– 2D

<u>TreeMIS2(v):</u>
$v.MISno \leftarrow 0$
$v.MISyes \leftarrow 1$
for each child w of v
  $v.MISno \leftarrow v.MISno + \text{TreeMIS2}(w)$
  $v.MISyes \leftarrow v.MISyes + w.MISno$
return $\max\{v.MISyes, v.MISno\}$



Dynamic Programming– 2D

```
TreeMIS2(v):
    v.MISno ← 0
    v.MISyes ← 1
    for each child w of v
        v.MISno ← v.MISno + TreeMIS2(w)
        v.MISyes ← v.MISyes + w.MISno
    return max{v.MISyes, v.MISno}
```



Dynamic Programming– 2D

$\underline{\text{TreeMIS2}(v):}$
$\quad v.MISno \leftarrow 0$
$\quad v.MISyes \leftarrow 1$
$\quad$for each child w of v
$\qquad v.MISno \leftarrow v.MISno + \text{TreeMIS2}(w)$
$\qquad v.MISyes \leftarrow v.MISyes + w.MISno$
$\quad$return $\max\{v.MISyes, v.MISno\}$