9. Dynamic Programming

Goals

- Learn elements of dynamic programming.
- Understand when to apply dynamic programming.
- Understand how to apply dynamic programming through several problems

Background

- Recursive solution
  - There is a subproblem (i.e., identical problem with a smaller size) inside a problem.
- pros
  - Conceptually simple way of solving a problem
- cons
  - There may be an excessive number of recursive calls.

Pros and Cons of Recursion

- Good cases
  - Mergesort, Quicksort
  - Computing a factorial
  - Depth-first search of a graph
  - …
- Bad cases
  - Computing Fibonacci numbers
  - Matrix-chain multiplication
  - …

Fibonacci Numbers

- $f(n) = f(n\text{-}1) + f(n\text{-}2)$
  $f(1) = f(2) = 1$

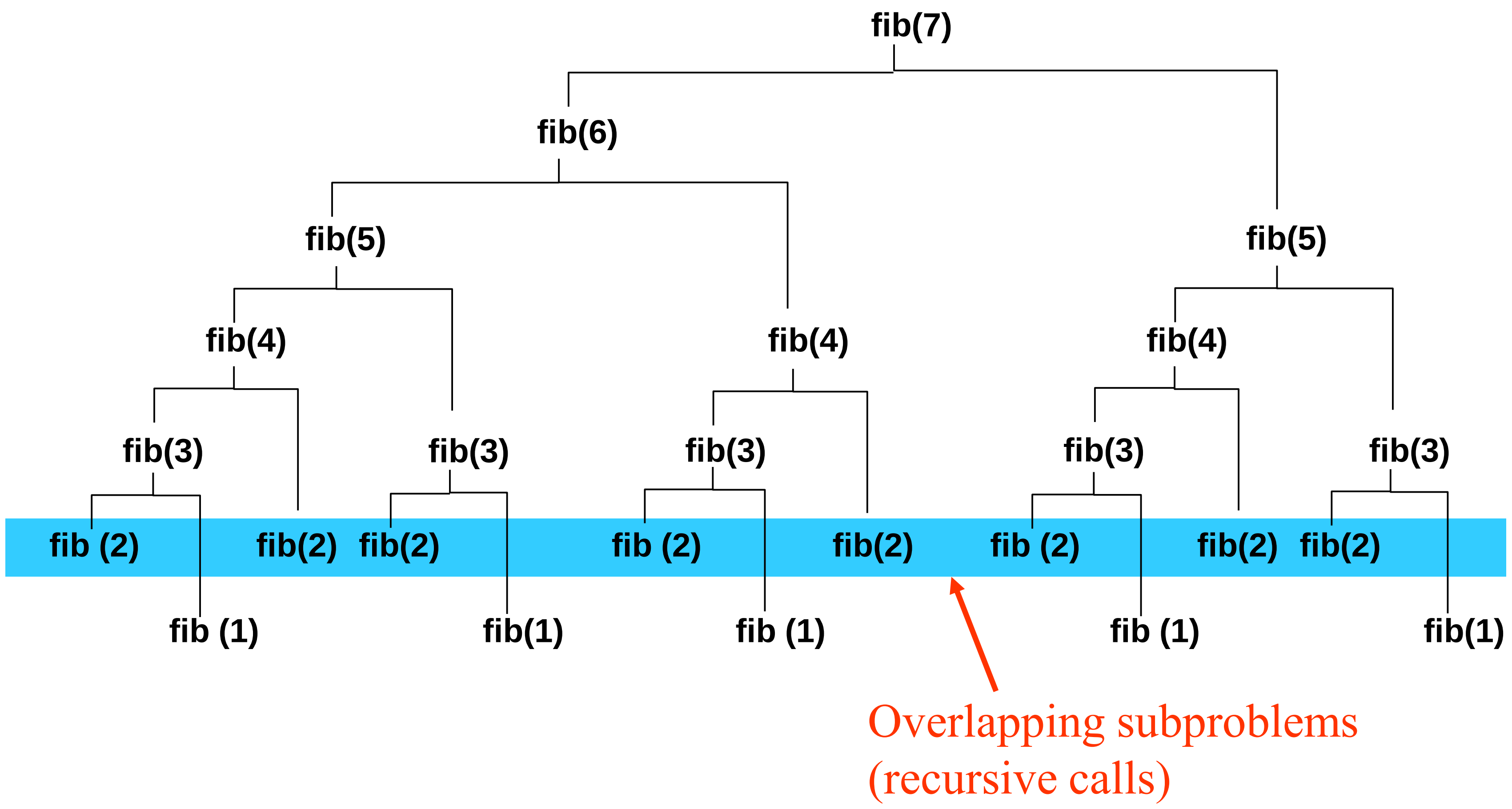
- Simple problem, but it shows the bad case of recursion (which is a motivation for dynamic programming).

Fibonacci Numbers

```
fib(n)
{
    if (n = 1 or n = 2)
        then return 1;
        else return (fib(n-1) +fib(n-2));
}
```

✓ There is an excessive number of overlapping recursive calls

Recursion Tree for Fibonacci Numbers



Dynamic Programming for Fibonacci Numbers

```
fibonacci(n)
{
    f[1] ← f[2] ← 1;
    for i ← 3 to n
        f[i] ← f[i-1] +f[i-2];
    return f[n];
}
```

✓ O(n) time

Elements of Dynamic Programming

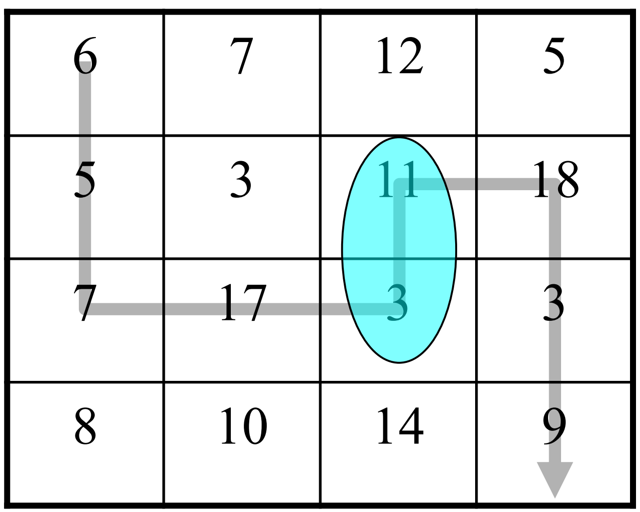
- Optimal substructure
  - An optimal solution to the problem contains within it optimal solutions to subproblems.
- Overlapping subproblems (recursive calls)
  - A recursive algorithm for the problem solves the same subproblems over and over.

➡ Dynamic programming is the solution!

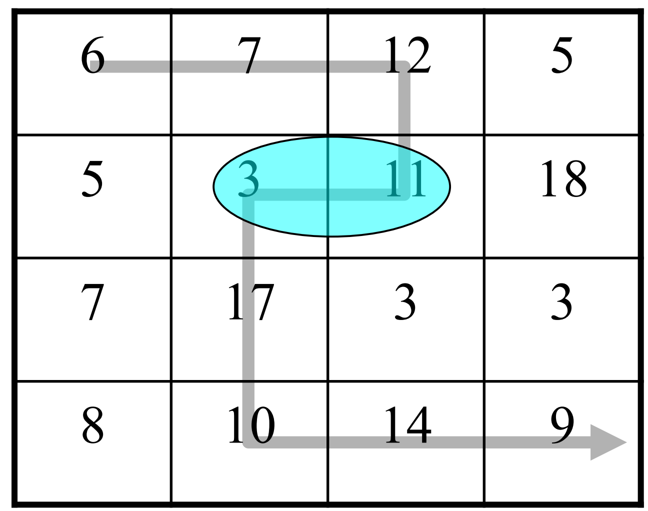
Problem 1: Matrix Path Problem

- Given an $n \times n$ matrix of positive numbers, we want to move from the upper-left corner to the lower-right corner.
- Constraints
  - Move only to the right or below
  - (Moving to the left, above, or diagonally is not allowed)
- Goal: maximize the sum of the numbers in the visited entries

Move



Not allowed



Not allowed

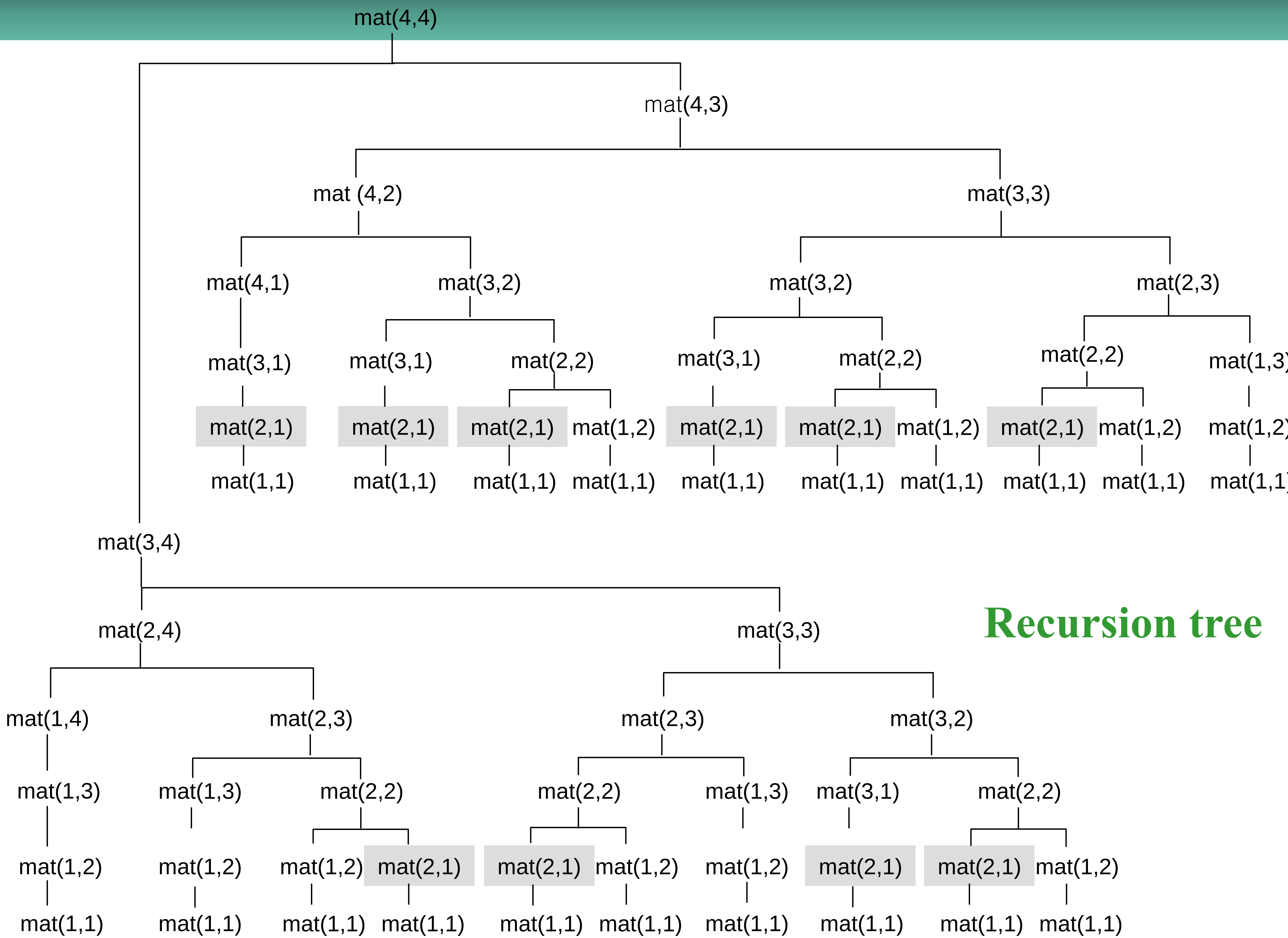
Move

6	7	12	5
5	3	11	18
7	17	3	3
8	10	14	9

6	7	12	5
5	3	11	18
7	17	3	3
8	10	14	9

Recursive Algorithm

```
matrixPath(i, j)
▷ returns maximum value from (1,1) to (i, j)
{
    if (i = 0 or j = 0) then return 0;
    else return (m_ij + (max(matrixPath(i-1, j), matrixPath(i, j-1))));
}
```



Recursion tree

DP recurrence

c[i, j]: maximum value from (1,1) to (i, j)

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ m_{ij} + \max(c[i-1,\ j],\ c[i,\ j-1]) & \text{otherwise.} \end{cases}$$

DP algorithm

```
matrixPath(n)
▷ returns maximum value from (1,1) to (n, n)
{
    for i ← 0 to n
        c[i, 0] ← 0;
    for j ← 1 to n
        c[0, j] ← 0;
    for i ← 1 to n
        for j ← 1 to n
            c[i, j] ← m_ij + max(c[i-1, j], c[i, j-1]);
    return c[n, n];
}
```

Problem 2: Placing Pebbles

- In each entry of a $3 \times N$ table, a positive or negative number is written. We want to place pebbles on the entries.
- Constraints
  - Pebbles cannot be placed in two (vertically or horizontally) adjacent entries.
  - In each column, at least one pebble should be placed.
- Goal: maximize the sum of numbers in the entries where pebbles are placed

Table

6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

Allowed

6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

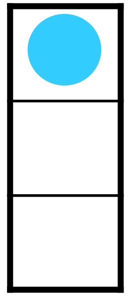
Not allowed

6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

Violation!

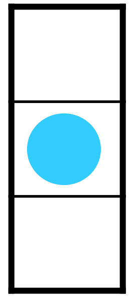
Patterns

Pattern 1:



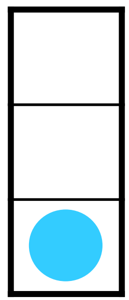
6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

Pattern 2:



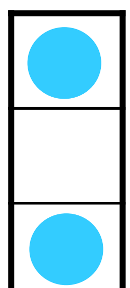
6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

Pattern 3:



6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

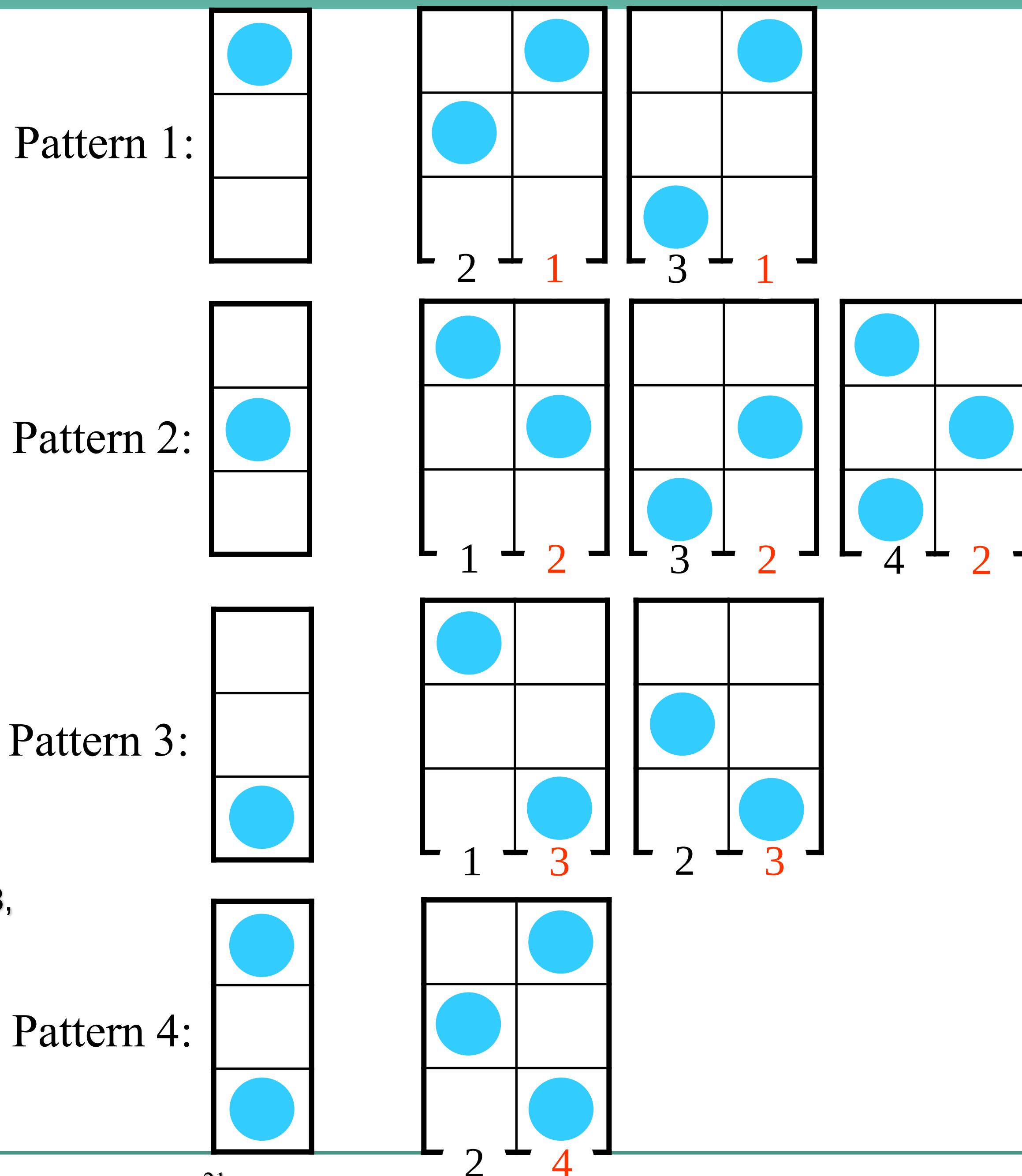
Pattern 4:



6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

There are 4 possible patterns for each column

Compatible Patterns



Pattern 1 is compatible with patterns 2, 3,
pattern 2 with patterns 1, 3, 4,
pattern 3 with patterns 1, 2, and
pattern 4 with pattern 2.

Recursive Algorithm

```
pebble(i, p)
▷ returns maximum value up to column i when column i has pattern p
▷ w[i, p]: sum of numbers at column i when column i has pattern p
{
    if (i = 1)
        then return w[1, p] ;
        else {
            max ← -∞ ;
            for q ← 1 to 4 {
                if (pattern q compatible with p)
                then {
                    tmp ← pebble(i—1, q) ;
                    if (tmp > max) then max ← tmp ;
                }
            }
            return (max + w[i, p]) ;
        }
}
```

pebbleSum(n)

▷ find maximum value up to column n

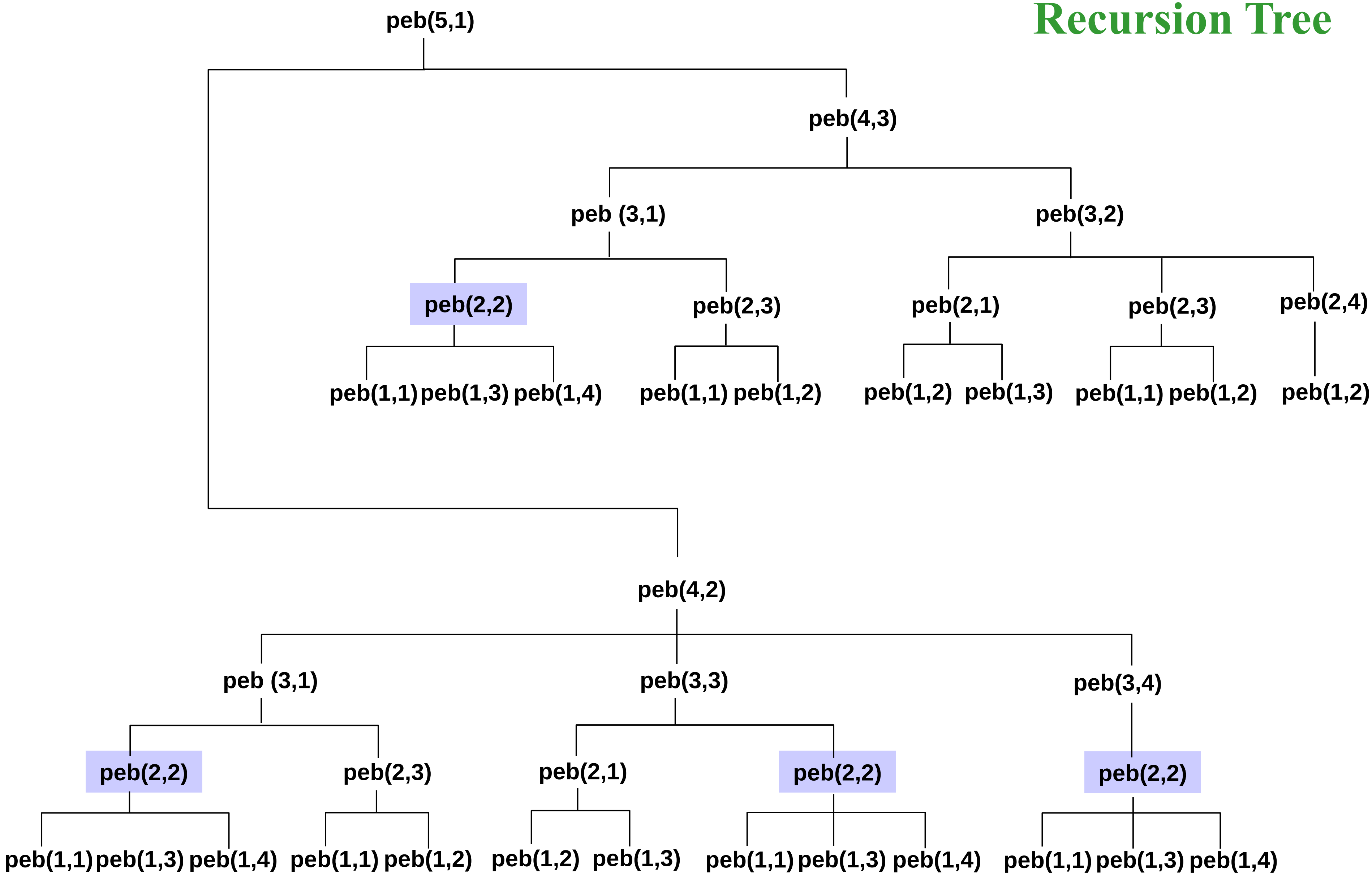
{

$$\textbf{return} \max_{p=1,2,3,4} \{ \text{pebble}(n, p) \} ;$$

}

✓ maximum of pebble(i, 1), …, pebble(i, 4) is the answer

Recursion Tree



Dynamic Programming

- Elements of dynamic programming
  - Optimal substructure
    - An optimal solution to the problem contains within it optimal solutions to subproblems.
    - peb[i, .] contains peb[i-1, .]
  - Overlapping subproblems
    - A recursive algorithm for the problem solves the same subproblems over and over.

Dynamic Programming

w[i, p]: sum of numbers at column i when column i has pattern p
peb[i, p]: maximum value up to column i when column i has pattern p

Recurrence for peb[i, p]:

$$\text{peb}[i, p] = \begin{cases} \text{w}[1, p] & \text{if } i = 1 \\ \max_{q \text{ compatible with } p} \{\text{peb}[i-1, q]\} + \text{w}[i, p] & \text{if } i > 1 \end{cases}$$

Finally, maximum of peb[n, 1] to peb[n, 4] is the answer

Dynamic Programming

pebble (n)
{
    for $p \leftarrow 1$ **to** 4
        $\text{peb}[1, p] \leftarrow \text{w}[1, p]$
    for $i \leftarrow 2$ **to** n
        for $p \leftarrow 1$ **to** 4
            $\text{peb}[i, p] \leftarrow \max_{q \text{ compatible with } p} \{\text{peb}[i\text{-}1, q]\} + \text{w}[i, p]$
    return $\max_{p=1,2,3,4} \{ \text{peb}[n, p] \}$
}

✓ Time Complexity : $\Theta(n)$

Problem 3: Matrix-Chain Multiplication

- Matrices A, B, C
  - (AB)C = A(BC)
- Dimensions: A:10 x 100, B:100 x 5, C:5 x 50
  - (AB)C: 10x100x5 + 10x5x50 = 7,500 scalar mults
  - A(BC): 100x5x50 + 10x100x50 = 75,000 scalar mults
- Optimal way of multiplying $A_1, A_2, A_3, \ldots, A_n$?
  - Way of parenthesizing $A_1, A_2, A_3, \ldots, A_n$ to minimize scalar multiplications

Optimal Substructure

- Last matrix multiplication
  - n-1 possibilities
    - $A_1(A_2 \ldots A_n)$
    - $(A_1A_2)(A_3 \ldots A_n)$
    - $(A_1A_2A_3)(A_4 \ldots A_n)$
    - ...
    - $(A_1 \ldots A_{n-2})(A_{n-1}A_n)$
    - $(A_1 \ldots A_{n-1})A_n$
  - Which one is the best?

Recurrence

- ✓ Dimensions of A_k: $p_{k-1} \times p_k$
- ✓ m[i, j]: minimum number of scalar multiplications to compute $A_i \ldots A_j$

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k \le j-1} \{ \underbrace{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j}_{(A_i \ldots A_k)\,(A_{k+1} \ldots A_j)} \} & \text{if } i < j \end{cases}$$

Recursive Algorithm

```
rMatrixChain(i, j)
▷ returns min number of scalar mults to compute A_i…A_j
{
    if (i = j) then return 0;    ▷ when there is only one matrix
    min ← ∞;
    for k ← i to j-1 {
        q ← rMatrixChain(i, k) + rMatrixChain(k+1, j) + p_{i-1}p_k p_j;
        if (q < min) then min ← q;
    }
    return min;
}
```

✓ excessive number of recursive calls!

Dynamic Programming

```
matrixChain(i, j)
{
    for i ← 1 to n
        m[i, i] ← 0;    ▷ when there is only one matrix
    for r ← 1 to n-1    ▷ r = j - i
        for i ← 1 to n-r {
            j ← i+r;
            m[i, j] ← ∞;
            for k ← i to j-1 {
                q ← min{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j};
                if (q < m[i, j]) then m[i, j] ← q;
            }
        }
    return m[1, n];
}
```

✓ Time complexity: $\Theta(n^3)$

Parenthesizing $A_1, A_2, A_3, \ldots, A_n$

```
matrixChain(i, j)
{
        for i ← 1 to n
                m[i, i] ← 0;    ▷ when there is only one matrix
        for r ← 1 to n-1        ▷ r = j - i
                for i ← 1 to n-r {
                        j ← i+r;
                        m[i, j] ← ∞;
                        for k ← i to j-1 {
                                q ← min{m[i, k] + m[k+1, j] + p_{i-1}p_k p_j};
                                if (q < m[i, j]) then {m[i, j] ← q; s[i, j] ← k;}
                        }
                }
        return m[1, n];
}
```

s[i, j]: index k to get minimum value m[i, j]

Problem 4: Longest Common Subsequence

- Similarity between two strings
- Subsequence
  - <bcdb> is a subsequence of string <abcbdab>
- Common subsequence
  - <bca> is a common subsequence of <abcbdab> and <bdcaba>
- Longest common subsequence (LCS)
  - <bcba> is a longest common subsequence of <abcbdab> and <bdcaba>.
  - <bdab> is also an LCS of <abcbdab> and <bdcaba>.
  - lcs(X, Y) denotes the length of an LCS of X and Y

Optimal Substructure

- For two strings $X_m = \langle x_1 x_2 \ldots x_m \rangle$ and $Y_n = \langle y_1 y_2 \ldots y_n \rangle$
  - If $x_m = y_n$

    $lcs(X_m, Y_n) = lcs(X_{m-1}, Y_{n-1}) + 1$
  - If $x_m \neq y_n$

    $lcs(X_m, Y_n) = \max(lcs(X_m, Y_{n-1}),\ lcs(X_{m-1}, Y_n))$

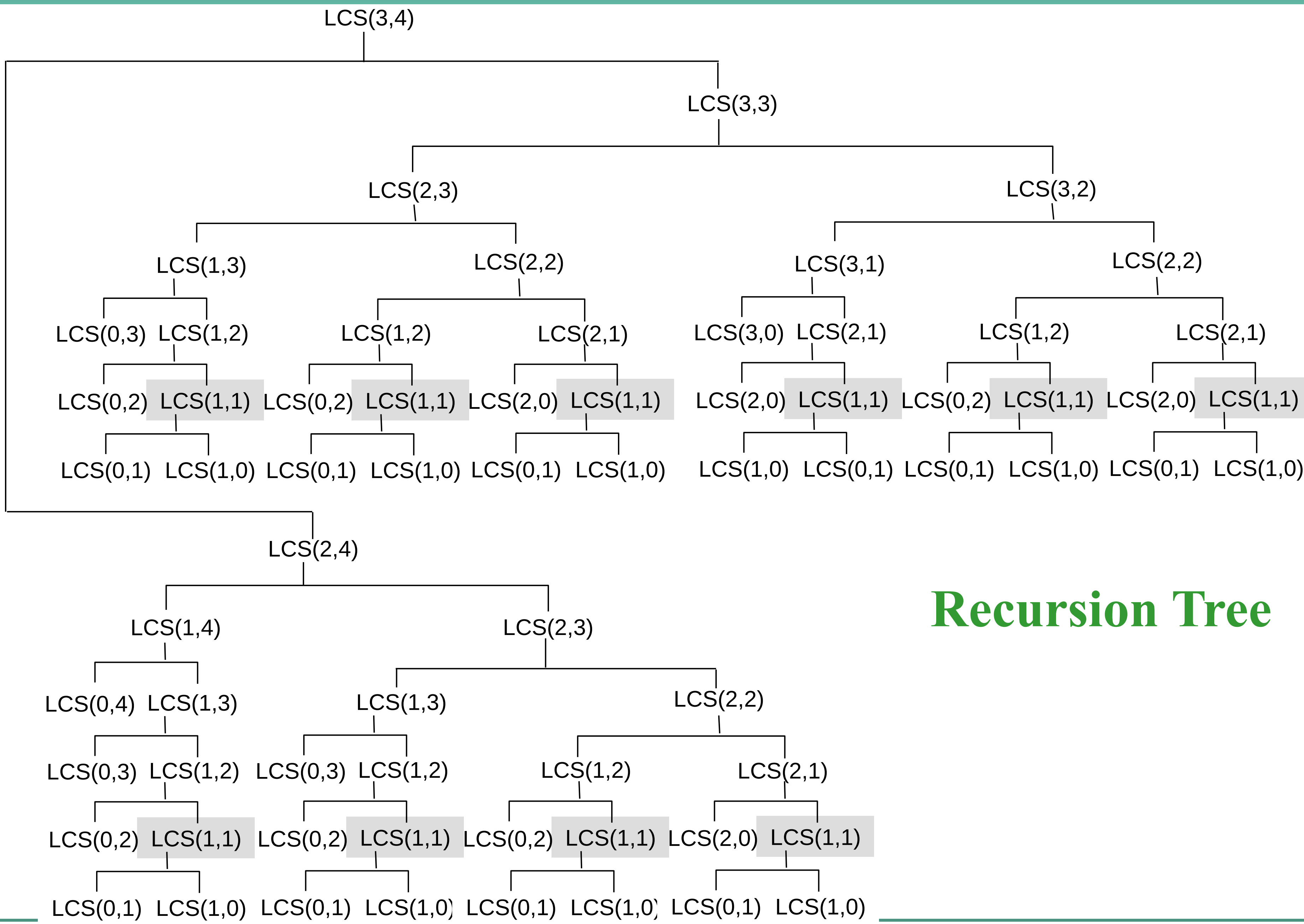
- $$C[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i\text{-}1, j\text{-}1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{C[i\text{-}1, j], C[i, j\text{-}1]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

✓ $C[i, j]$: $lcs(X_i, Y_j)$, where $X_i = \langle x_1 x_2 \ldots x_i \rangle$ and $Y_j = \langle y_1 y_2 \ldots y_j \rangle$

Recursive Algorithm

LCS(m, n)
▷ returns lcs(X_m, Y_n)
{
    if (m = 0 **or** n = 0) **then return** 0;
    else if ($x_m = y_n$) **then return** LCS(m-1, n-1) + 1;
    else return max(LCS(m-1, n), LCS(m, n-1));
}

✓ excessive number of recursive calls!



Recursion Tree

Dynamic Programming

```
LCS(m, n)
▷ computes an LCS of X_m and Y_n
{
    for i ← 0 to m
        C[i, 0] ← 0;
    for j ← 0 to n
        C[0, j] ← 0;
    for i ← 1 to m
        for j ← 1 to n
            if (x_i = y_j) then C[i, j] ← C[i-1, j-1] + 1; B[i, j] ← 1
            elseif (C[i-1, j] ≥ C[i, j-1]) then C[i, j] ← C[i-1, j] ; B[i, j] ← 2
            else C[i, j] ← C[i, j-1]; B[i, j] ← 4
    return C, B;
}
```

✓ Time complexity: $\Theta(mn)$

C and B tables

		b	d	c	a	b	a
	0	0	0	0	0	0	0
a	0	0	0	0	1	1	1
b	0	1	1	1	1	2	2
c	0	1	1	2	2	2	2
b	0	1	1	2	2	3	3
d	0	1	2	2	2	3	3
a	0	1	2	2	3	3	4
b	0	1	2	2	3	4	4

2	2	2	1	4	1
1	4	4	2	1	4
2	2	1	4	2	2
1	2	2	2	1	4
2	1	6	2	2	2
2	2	2	1	2	1
1	2	2	2	1	6

C table

LCS

bcba, bdab, bcab

B table

Thank you