Dynamic Programming (DP)

Background

- Recursive methods
  - Similar subproblems inside the original problem
  - Not always good
    - Too many redundant calls

Cons and Pros of Recursion

- Recursion is good for these:
  - Sort algorithms such as Quick Sort, Merge Sort, etc.
  - Factorial (Well, actually NO in pratcice!)
  - DFS over graphs
  - …
  - (In REAL programming practice, recursion is mostly avoided!)
- But bad for these:
  - Fibonacci numbers
  - Optimal order of matrices multiplication
  - …

Introduction: Fibonacci

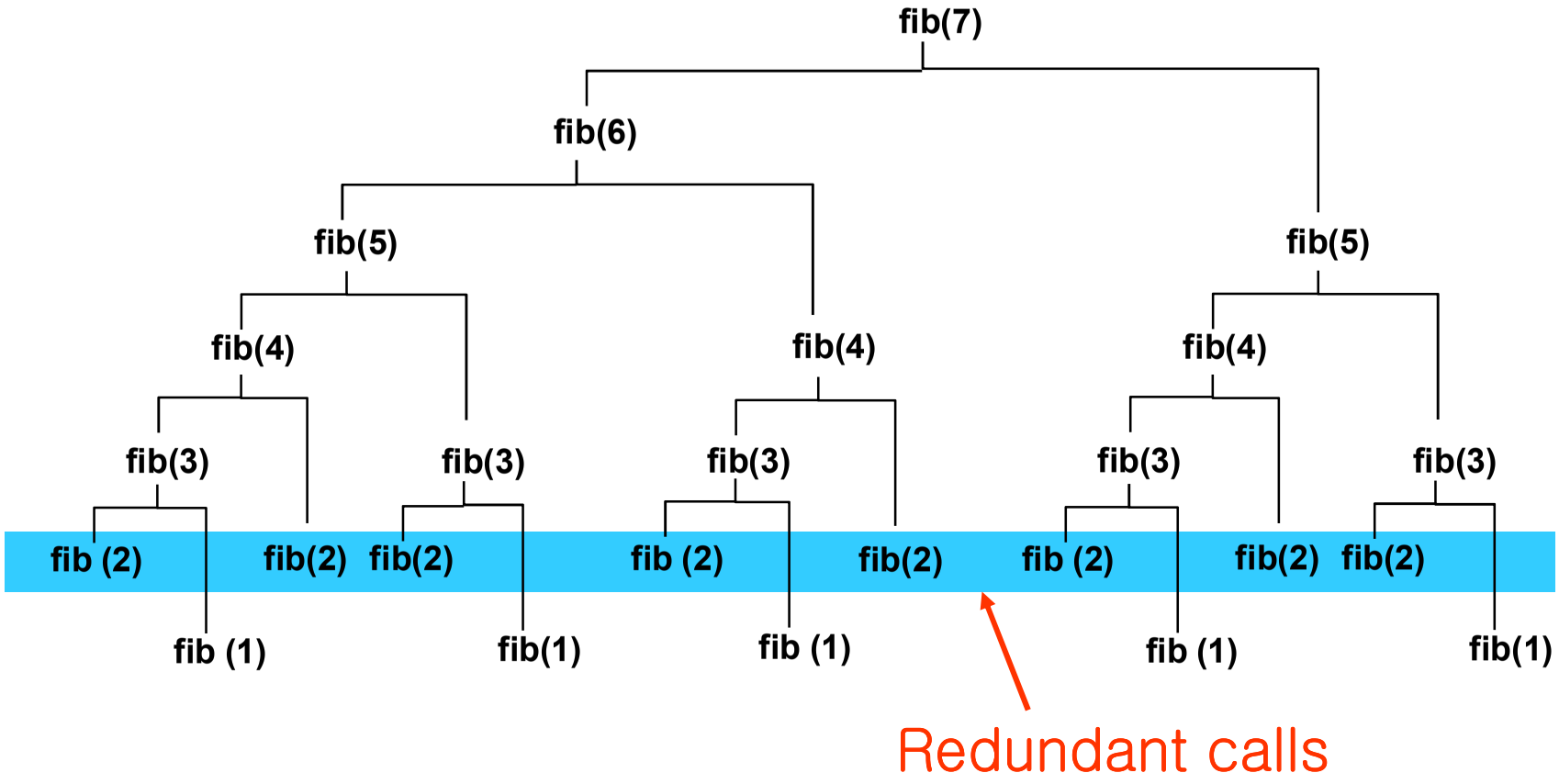
- $f_n = f_{n-1} + f_{n-2}$
- Very simple problem
  - But contains good motivations and implementation details of Dynamic programming
- Fibonacci is a fast growing function!

Recursive Algorithm for Fibonacci

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

✓ Many redundant recursive calls

Call Tree of Fibonacci



DP Algorithm for Fibonacci

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

✓ $\Theta(n)$ time complexity

We need DP when…

- Optimal substructure
  - Optimal solution has optimal sub-solution
- Overlapping recursive calls
  - In recursive solution, too many redundant recursive calls for the same subproblem

➡ DP is a solution!

DP's property

- DP is implicitly related with DAG data structure
  - page 170 of DPV
- Comparison with Recursion
  - First, it gets its intermediate values using table lookup instead of recursive calls.
  - Second, it updates the parent field of each step, which will enable us to reconstruct the solution later.
  - Third, it is instrumented using a more general function instead of just returning the intermediate value. This will enable us to apply the routine to a wider class of problems.

Example 1:Pebbling a checkerboard

- In each cell of a 3×N table, there is an integer number, either positive or negative
- You put a pebble on a cell under these condition:
  - At each column, there should be at least one pebble
  - There should be no pebbles horizontally or vertically adjacent to a pebble
- Goal: Place pebbles so that the sum of numbers in the cells filled with pebbles is maximum

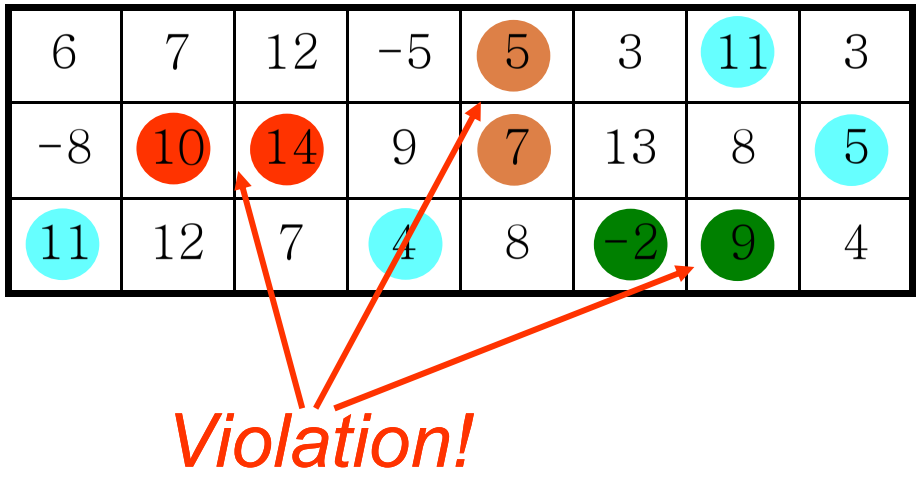
Example

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

Legal

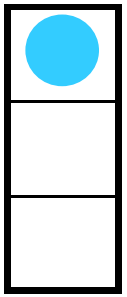
6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

Illegal



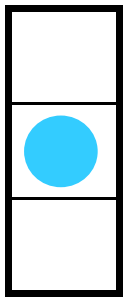
Possible patterns

Pattern 1:



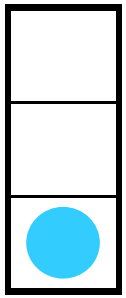
6	7	12	**-5**	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

Pattern 2:



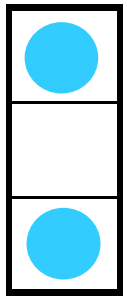
6	7	12	-5	5	3	11	3
-8	10	14	**9**	7	13	8	5
11	12	7	4	8	-2	9	4

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

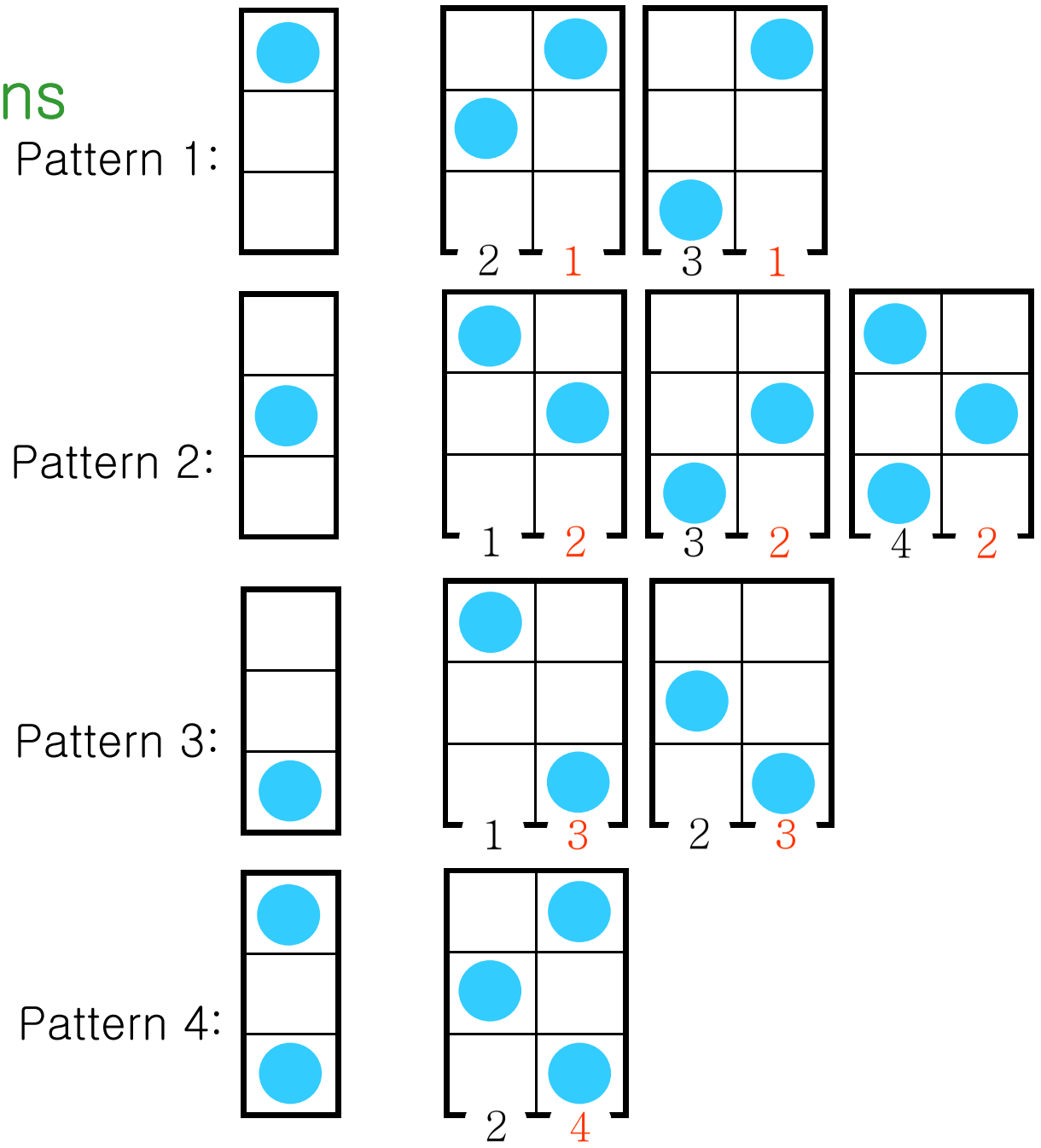
Pattern 4:



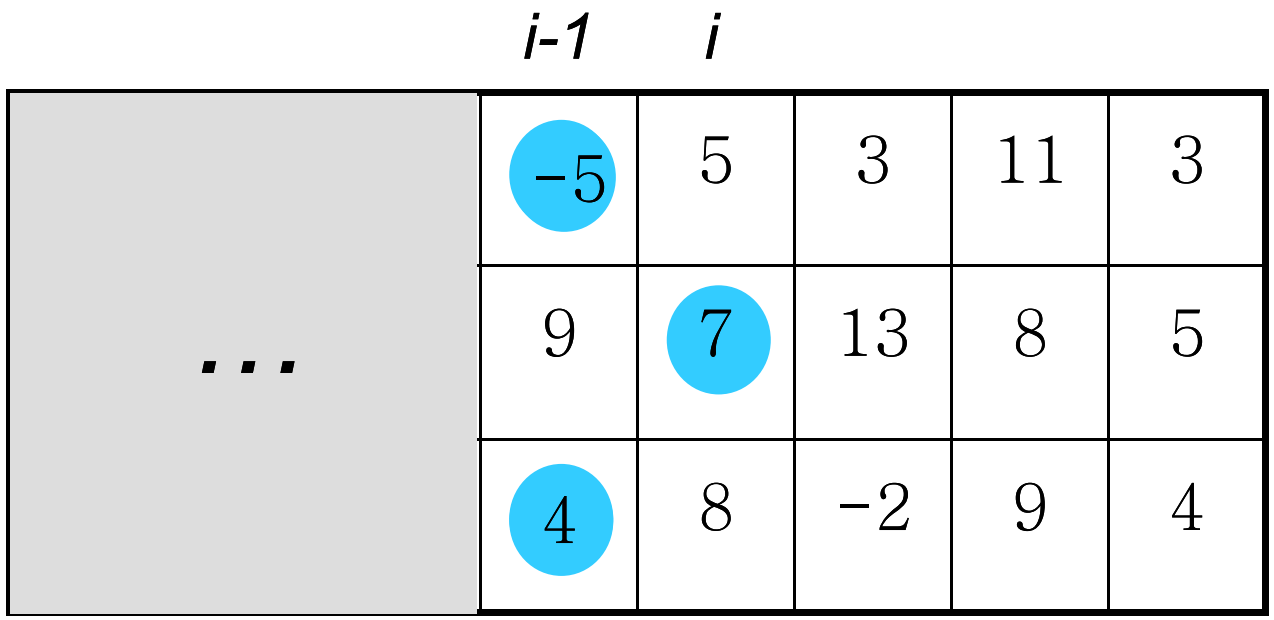
6	7	12	**-5**	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	**4**	8	-2	9	4

There are only four patterns to fill one column.

Compatible Patterns



Pattern 1 is compatible with 2, and 3
Pattern 2 is compatible with 1, 3, and 4
Pattern 3 is compatible with 1, and 2
Pattern 4 is compatible with 2



Consider the relation between column i and column i-1.

Recursive Version

```
pebble(i, p)
▷ Calculate the max sum upto the column i when pattern p is on column i.
▷ w[i, p] : sum of pebbled integers when pattern p is on column i: p ∈ {1, 2, 3, 4}
{
    if (i = 1)
        then return w[1, p] ;
        else {
            max ← −∞ ;
            for q ← 1 to 4 {
                if (pattern q is compatible with pattern p)
                then {
                    tmp ← pebble(i−1, q) ;
                    if (tmp > max) then max ← tmp ;
                }
            }
            return (w[i, p] + max) ;
        }
}
```

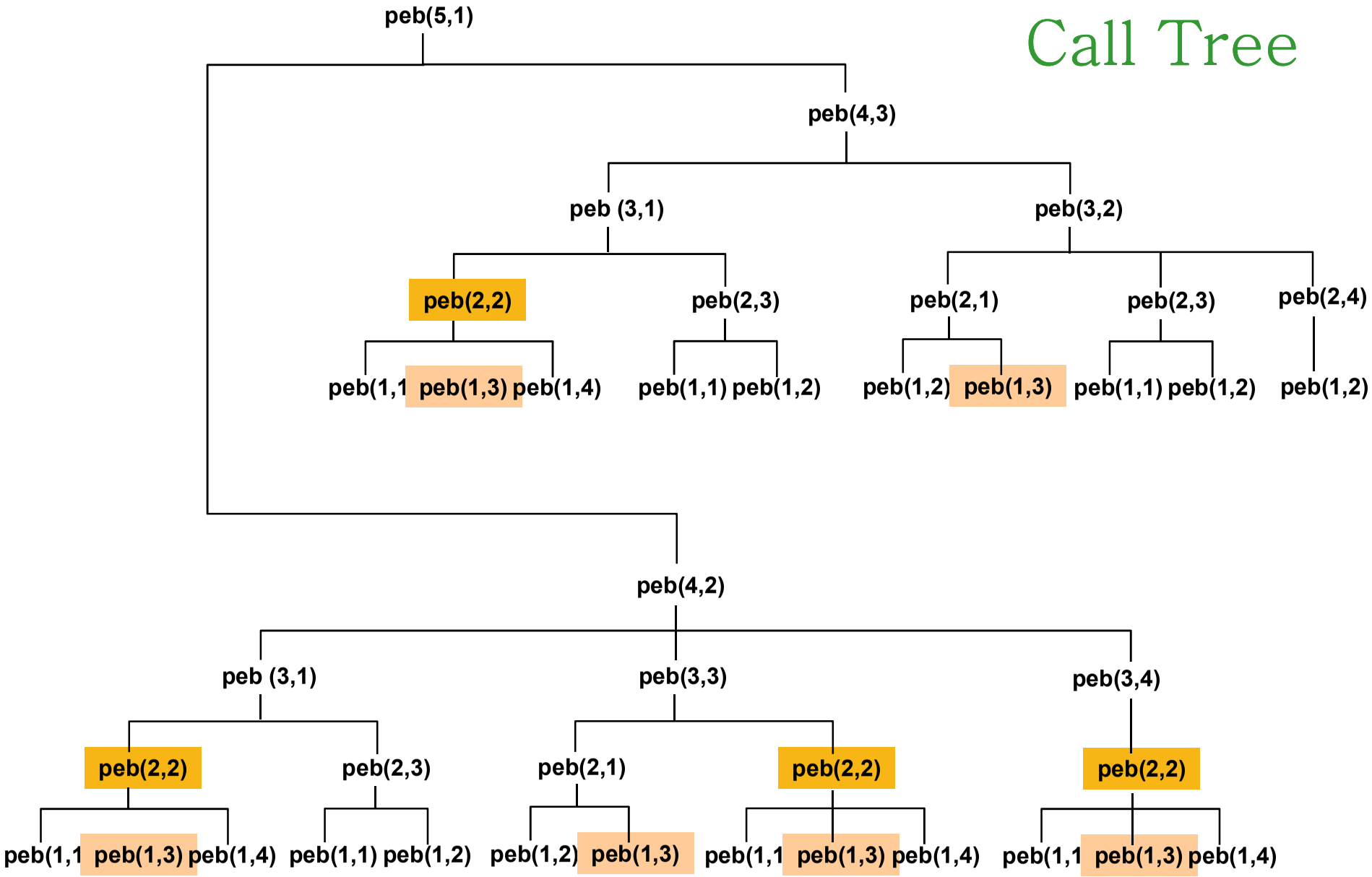
pebbleSum(*n*)

▷ Calculate the max sum upto column *n*

```
{
        return  max { pebble(n, p) } ;
                p =1,2,3,4
}
```

✓ The final answer is the max among pebble(*i*, *1*), …, pebble(*i*, *4*)

Call Tree


peb(5,1)
peb(4,3)
peb (3,1)
peb(3,2)
peb(2,2)
peb(2,3)
peb(2,1)
peb(2,3)
peb(2,4)
peb(1,1
peb(1,3)
peb(1,4)
peb(1,1)
peb(1,2)
peb(1,2)
peb(1,3)
peb(1,1)
peb(1,2)
peb(1,2)
peb(4,2)
peb (3,1)
peb(3,3)
peb(3,4)
peb(2,2)
peb(2,3)
peb(2,1)
peb(2,2)
peb(2,2)
peb(1,1
peb(1,3)
peb(1,4)
peb(1,1)
peb(1,2)
peb(1,2)
peb(1,3)
peb(1,1
peb(1,3)
peb(1,4)
peb(1,1
peb(1,3)
peb(1,4)

Applying DP

- DP's properties
  - Optimal substructure
    - pebble(*i*, .) has pebble(*i*-1, .)
    - So, big problem's optimal solution contains smaller problem's optimal solutions
  - Overlapping recursive calls
    - Recursive version has too many redundant calls

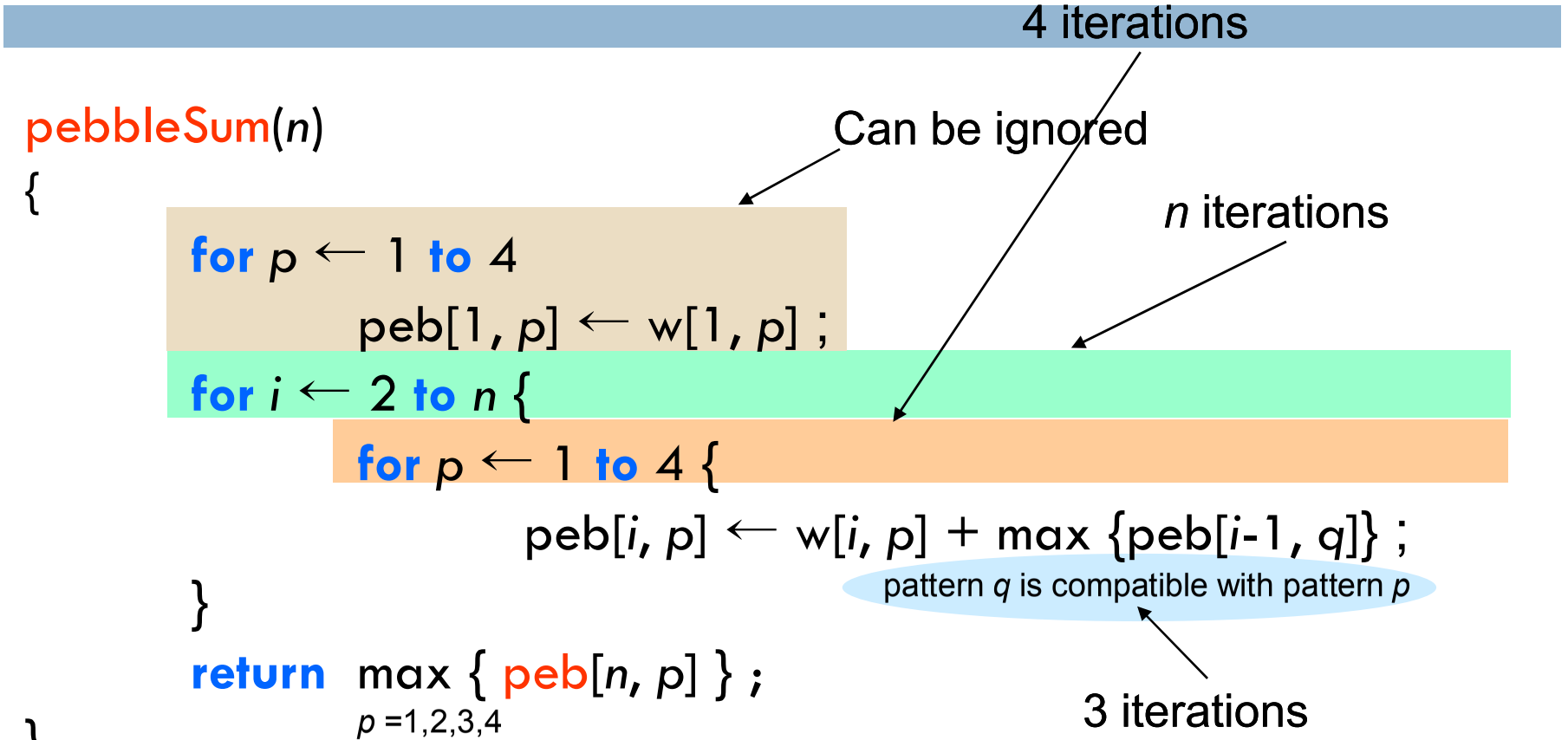
DP

```
pebbleSum(n)
{
        for p ← 1 to 4
                peb[1, p] ← w[1, p] ;
        for i ← 2 to n {
                for p ← 1 to 4 {
                        peb[i, p] ← w[i, p] + max {peb[i-1, q]} ;
        }
        return  max { peb[n, p] } ;
                p =1,2,3,4
}
```

For each pattern q which is compatible with pattern p

✓Complexity : $O(n)$

Complexity

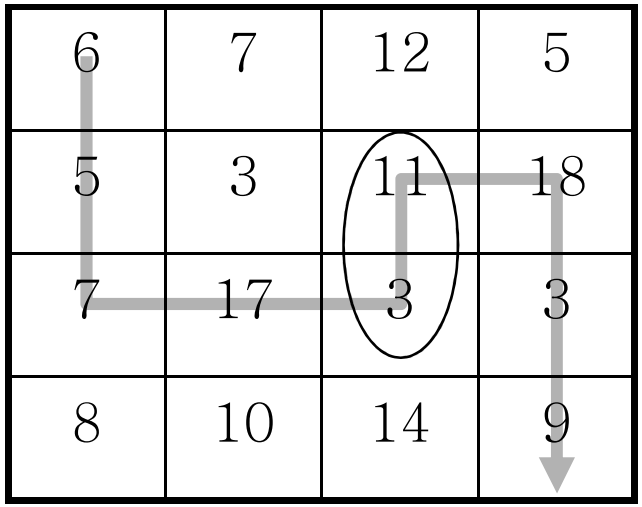


✓Complexity : $O(n)$ n * 4 * 3 = $O(n)$

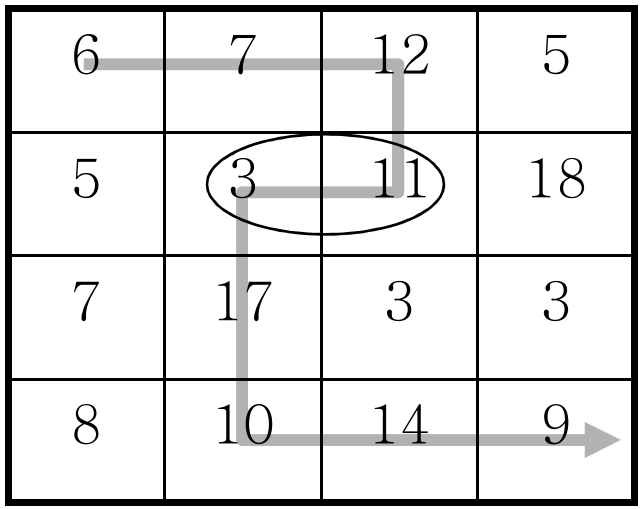
Example 2: Path in the array

- Given $n \times n$ matrix whose elements are integers, the algorithm moves from the top-left corner to the bottom-right corner of the matrix.
- Move with the following constraints:
  - Move to the right direction or the downward direction
  - Cannot move to the left, upward, or diagonal directions
- Goal: Move from the top-left corner to the bottom-right corner so that the sum of the numbers along the visited cells is the maximum

Illegal moves



Going up



Going left

Valid moves

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

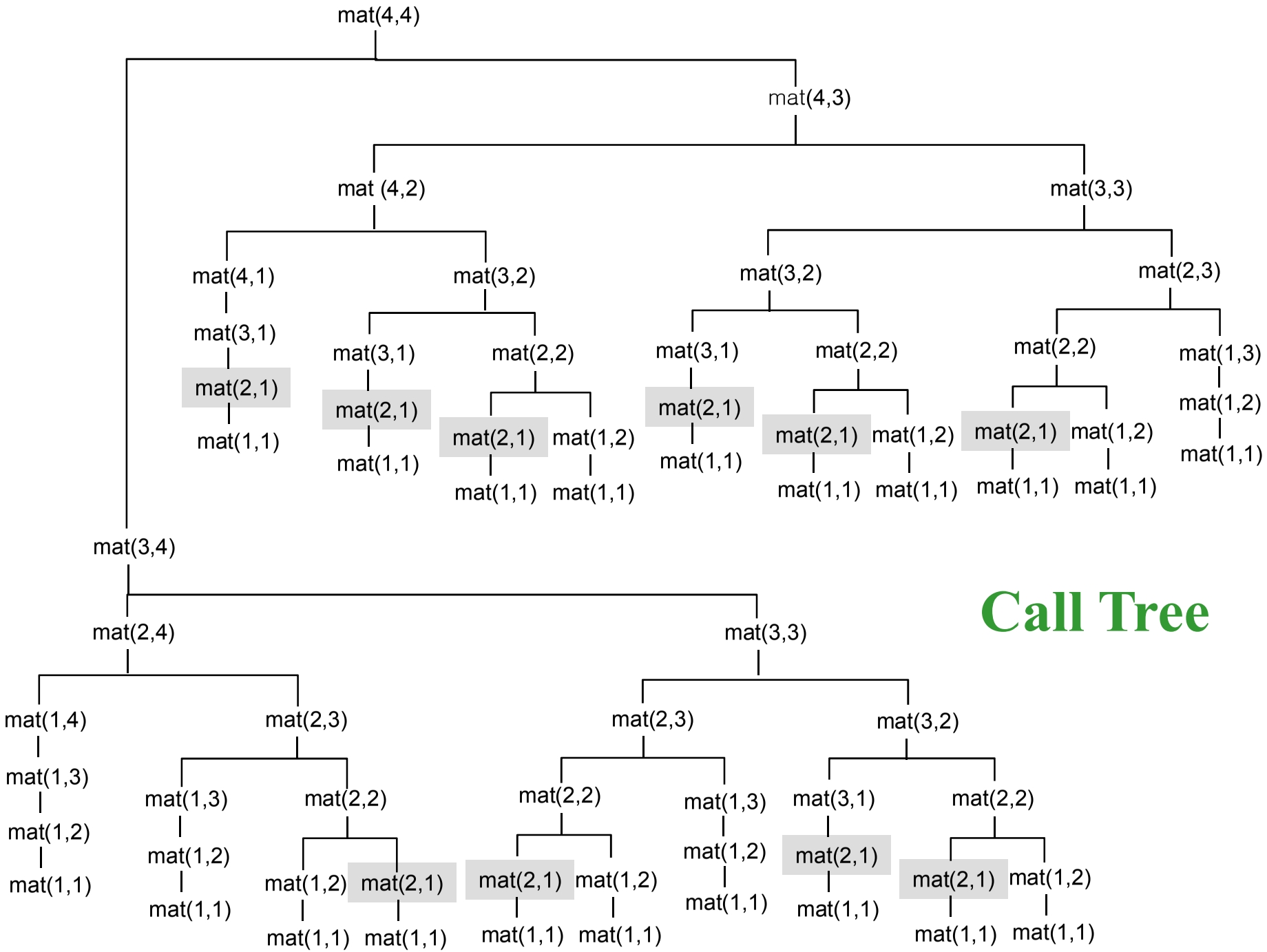
matrixPath(i, j)

▷ The max sum at the cell (i, j)

{

```
if (i = 1 and j = 1) then return m_ij;
else if (i = 1) then return (matrixPath(1, j-1) + m_ij);
else if (j = 1) then return (matrixPath(i-1, 1) + m_ij);
else return ((max(matrixPath(i-1, j), matrixPath(i, j-1)) + m_ij);
```

}


mat(4,4)
mat(4,3)
mat (4,2)
mat(3,3)
mat(4,1)
mat(3,2)
mat(3,2)
mat(2,3)
mat(3,1)
mat(3,1)
mat(2,2)
mat(3,1)
mat(2,2)
mat(2,2)
mat(1,3)
mat(2,1)
mat(2,1)
mat(2,1)
mat(2,1)
mat(1,2)
mat(2,1)
mat(1,2)
mat(1,2)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(3,4)
mat(2,4)
mat(3,3)
mat(1,4)
mat(2,3)
mat(2,3)
mat(3,2)
mat(1,3)
mat(1,3)
mat(2,2)
mat(2,2)
mat(1,3)
mat(3,1)
mat(2,2)
mat(1,2)
mat(1,2)
mat(1,2)
mat(2,1)
mat(2,1)
mat(1,2)
mat(2,1)
mat(1,2)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
Call Tree

DP

matrixPath(i, j)

▷ The max sum at the cell (i, j)

{

c[1,1] ← m_{11} ;

for i ← 2 **to** n

c[i,1] ← m_{i1} + c[i-1,1];

for j ← 2 **to** n

c[1, j] ← m_{1j} + c[1, j-1];

for i ← 2 **to** n

for j ← 2 **to** n

c[i, j] ← m_{ij} + max(c[i-1, j], c[i, j-1]);

return c[n, n];

}

Example 3: Matrix-Chain Multiplication

- Matrices A, B, C
  - (AB)C = A(BC)
- Ex: A:10 × 100, B:100 × 5, C:5 × 50
  - (AB)C: 7500 times multiplications
  - A(BC): 75000 times multiplications
- What is the best order to multiply A_1, A_2, A_3, …, A_n?

Recursive Relation

- At the time of the last matrix multiplication
  - n-1 possibilities
    - $(A_1 \ldots A_{n-1})A_n$
    - $(A_1 \ldots A_{n-2})\ (A_{n-1}A_n)$
    - . . .
    - $(A_1A_2)(A_3 \ldots A_n)$
    - $A_1(A_2 \ldots A_n)$
  - Which one is most interesting?

- m[i, j]: The cost of multiplying $A_i, A_{i+1}, \ldots, A_j$
- A_k's dimension: $p_{k-1}p_k$

$$m[i,j] = \begin{cases} 0 & , i=j \\ \min\limits_{i \le k \le j-1} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & , i<j \end{cases}$$

Recursive Algorithm

```
rMatrixChain(i, j)
▷ Calculate the minimum cost to multiply matrices
{
    if (i = j) then return 0;    ▷ 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;
}
```

✓ Too many redundant recursive calls!

DP

```
matrixChain(i, j)
{
        for i ← 1 to n
                m[i, i] ← 0;            ▷ 0 when there is only one matrix
        for r ← 1 to n-1    ▷ Size of the problem is r+1
                for i ← 1 to n-r {
                        j ← i+r;
                        m[i, j] ← min {m[i, k] + m[k+1, j] + p_{i-1} p_k p_j};
                                  i ≤ k ≤ j-1
                }
        return m[1, n];
}
```

✓ Complexity: $\Theta(n^3)$

Example 4:
Longest Common Subsequence(LCS)

- Find the longest common subsequence between two strings
- Example of subsequence
  - <bcdb> is a subsequence of <abcbdab>
- Example of Common subsequence
  - <bca> is a common subsequence of strings <abcbdab> and <bdcaba>
- Longest common subsequence(LCS)
  - The longest among common subsequences
  - Ex: <bcba> is an LCS of string <abcbdab> and <bdcaba>

Optimal Substructure

- 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 $x_m = y_n$, the length of LCS of X_m and Y_n is 1+ LCS of X_{m-1} and Y_{n-1}
  - if $x_m \neq y_n$, the length of LCS of X_m and Y_n is max of { LCS of X_m and Y_{n-1}, LCS of X_{m-1} and Y_n }

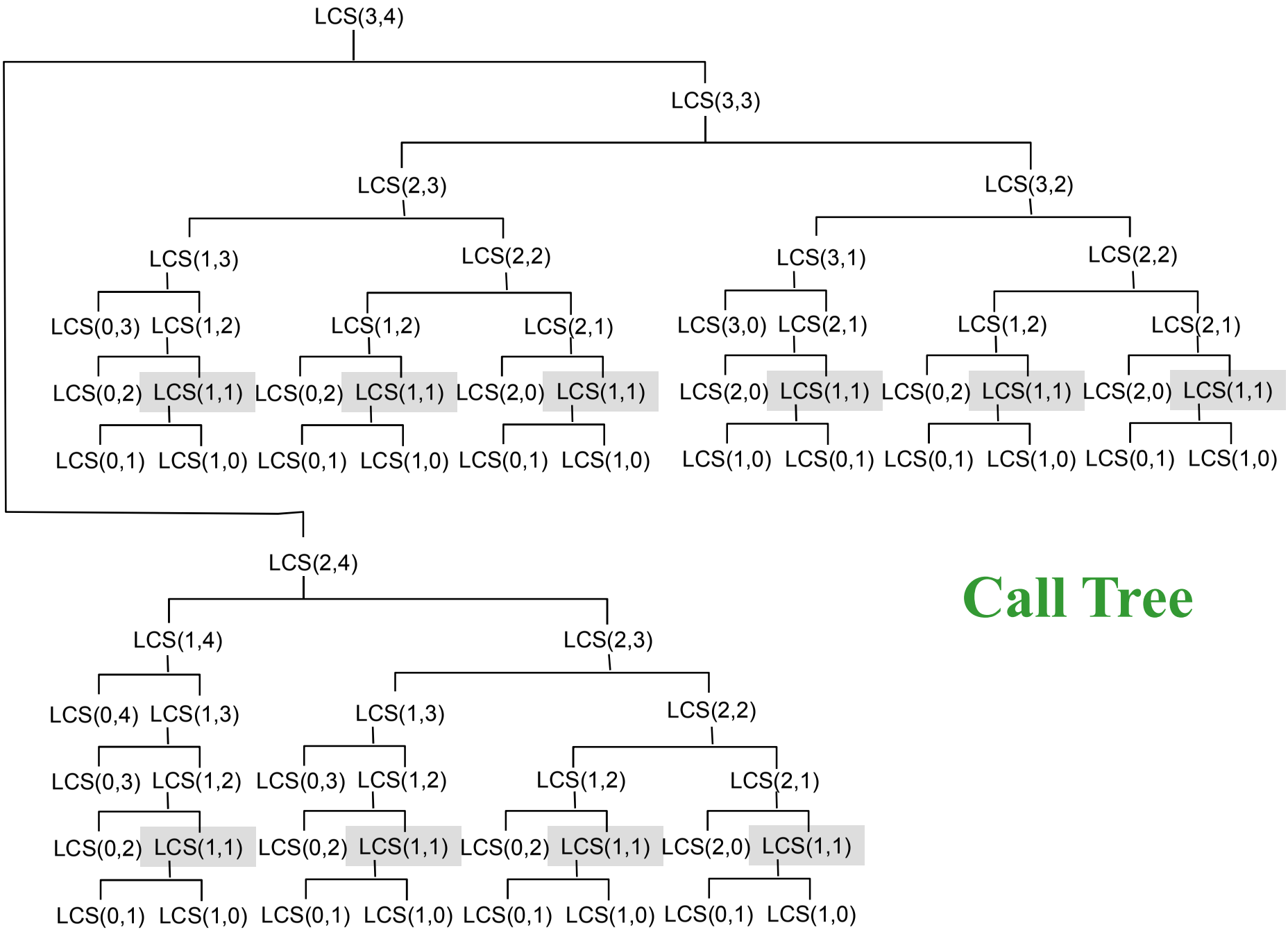
- $$c_{ij} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c_{i-1,\,j-1} + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{c_{i-1,\,j},\ c_{i,\,j-1}\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

✓ c_{ij} : The length of LCS of two strings $X_i = \langle x_1 x_2 \cdots x_i \rangle$ and $Y_j = \langle y_1 y_2 \cdots y_j \rangle$

Recursive Algorithm

```
LCS(m, n)
▷ Calculate the length of LCS of two strings X_m and 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));
}
```

✓ Many redundant recursive calls!


LCS(3,4)
LCS(3,3)
LCS(2,3)
LCS(3,2)
LCS(1,3)
LCS(2,2)
LCS(3,1)
LCS(2,2)
LCS(0,3) LCS(1,2)
LCS(1,2) LCS(2,1)
LCS(3,0) LCS(2,1)
LCS(1,2) LCS(2,1)
LCS(0,2) LCS(1,1)
LCS(0,2) LCS(1,1)
LCS(2,0) LCS(1,1)
LCS(2,0) LCS(1,1)
LCS(0,2) LCS(1,1)
LCS(2,0) LCS(1,1)
LCS(0,1) LCS(1,0)
LCS(0,1) LCS(1,0)
LCS(0,1) LCS(1,0)
LCS(1,0) LCS(0,1)
LCS(0,1) LCS(1,0)
LCS(0,1) LCS(1,0)
LCS(2,4)
LCS(1,4)
LCS(2,3)
LCS(0,4) LCS(1,3)
LCS(1,3)
LCS(2,2)
LCS(0,3) LCS(1,2)
LCS(0,3) LCS(1,2)
LCS(1,2)
LCS(2,1)
LCS(0,2) LCS(1,1)
LCS(0,2) LCS(1,1)
LCS(0,2) LCS(1,1)
LCS(2,0) LCS(1,1)
LCS(0,1) LCS(1,0)
LCS(0,1) LCS(1,0)
LCS(0,1) LCS(1,0)
LCS(0,1) LCS(1,0)

Call Tree

DP

```
LCS(m, n)
▷ Calculate the length of LCS of two strings 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_m = y_n) then C[i, j] ← C[i-1, j-1] + 1;
            else C[i, j] ← max(C[i-1, j], C[i, j-1]);
    return C[m, n];
}
```

✓ Complexity: $\Theta(mn)$

Maximum Monotone Subsequence

- A numerical sequence is monotonically increasing if the i^{th} element is at least as big as the $(i-1)^{st}$ element.
- The maximum monotone subsequence problem seeks to delete the fewest number of elements from an input string S to leave a monotonically increasing subsequence.
- Thus a longest increasing subsequence of "243517698" is "23568."

Reduction of Maximum Monotone Subsequence to LCS

- In fact, this is just a longest common subsequence problem, where the second string is the elements of S sorted in increasing order.
- Any common sequence of these two must
  - (a) represent characters in proper order in S, and
  - (b) use only characters with increasing position in the collating sequence, so the longest one does the job.

Example 5: Shortest Path

- Weighted digraph $G=(V, E)$
  - $w_{i,j}$: The edge length from vertex i to vertex j
    - No Edge : ∞
- Goal
  - Calculate all the shortest path lengths from the starting vertex *s* to the other vertices

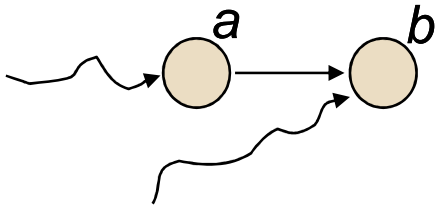
- d_t^k : Shortest path length from vertex s to vertex t through at most k edges
- Goal: d_t^{n-1}
- For $i \neq s$,
  - $d_t^0 = \infty$
  - $d_t^1 = w_{s,t}$

Recursive Relation

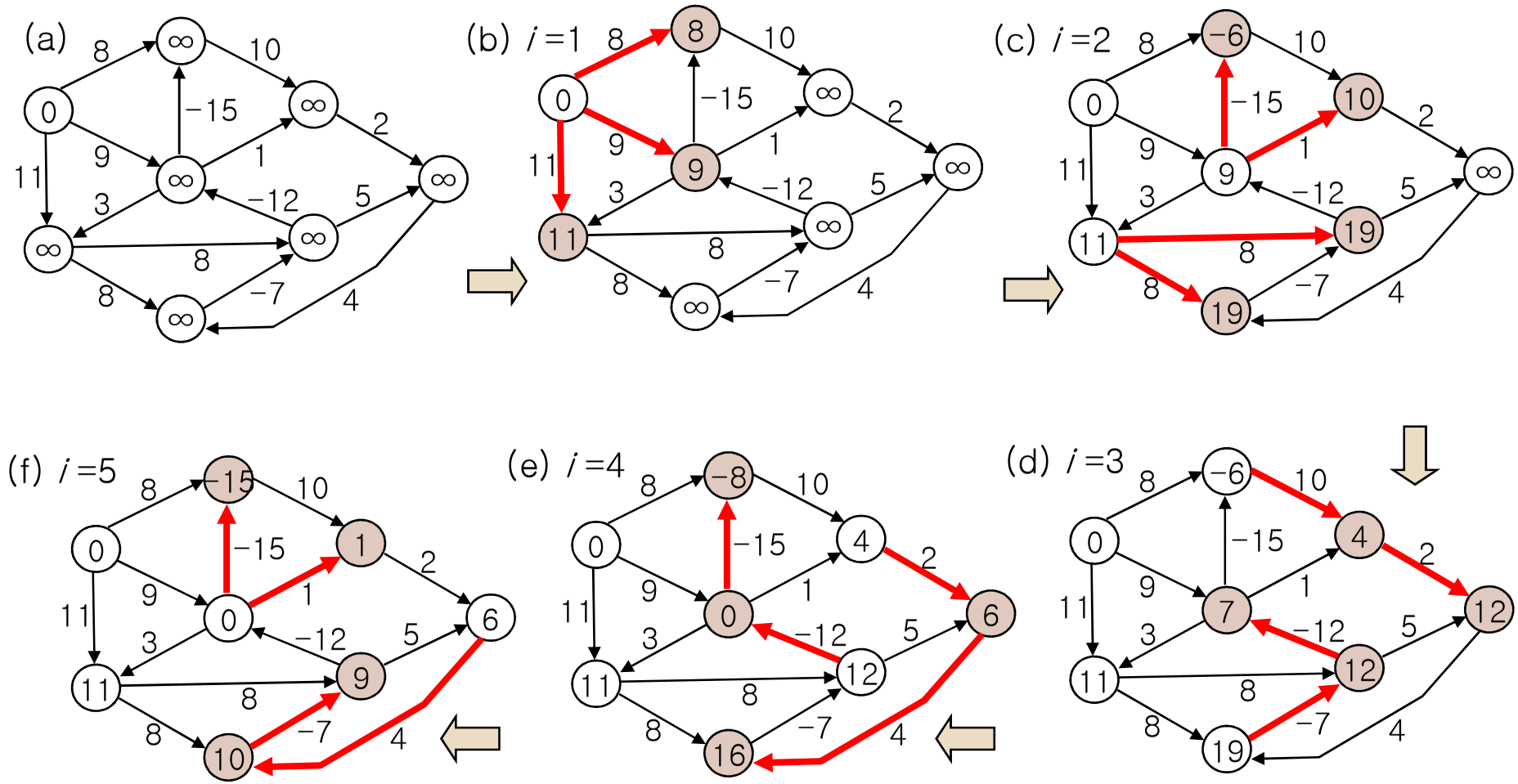
$$\begin{cases} d_t^{\,k} = \min_{\text{for all edges } (r,\,t)} \{d_r^{\,k-1} + w_{r,\,t}\} \\ d_s^{\,0} = 0; \\ d_t^{\,0} = \infty; \end{cases}$$

DP

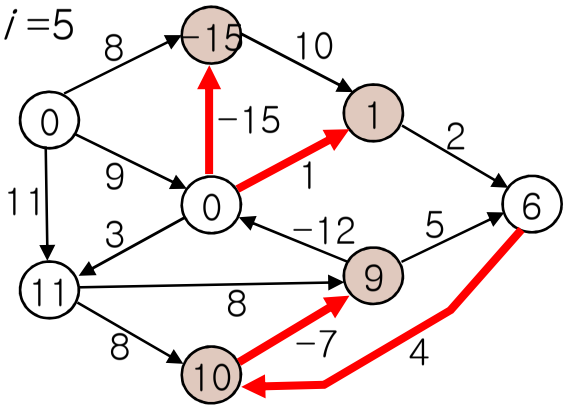
```
Bellman-Ford(G, s)
{
        d_s ← 0;
        for all vertices i ≠ s
                d_i ← ∞;
        for k ← 1 to n-1 {
                for all edges (a, b) {
                        if (d_a + w_a,b < d_b )  then d_b ← d_a + w_a,b ;
                }
        }
}
```



✓ Think of “propagation”!

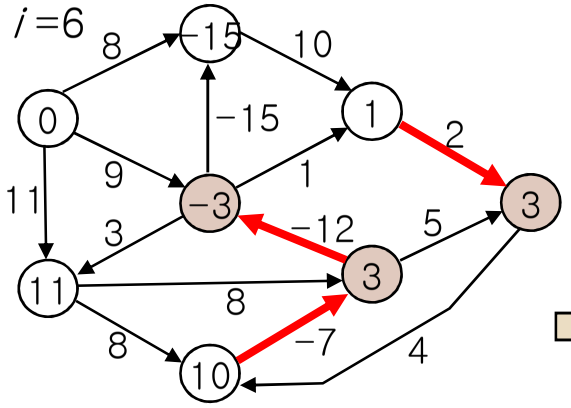

(a)
(b) i =1
(c) i =2
(d) i =3
(e) i =4
(f) i =5

(f) $i=5$

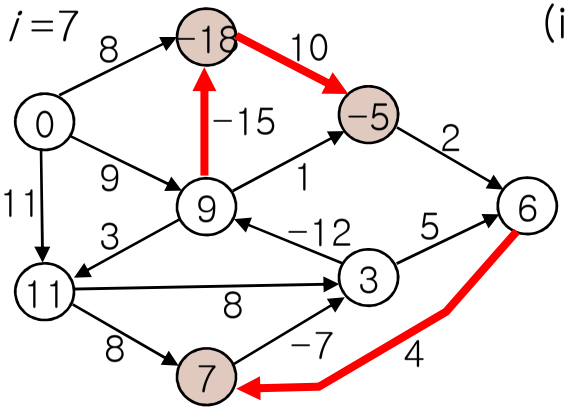


We discussed this
in the graph section

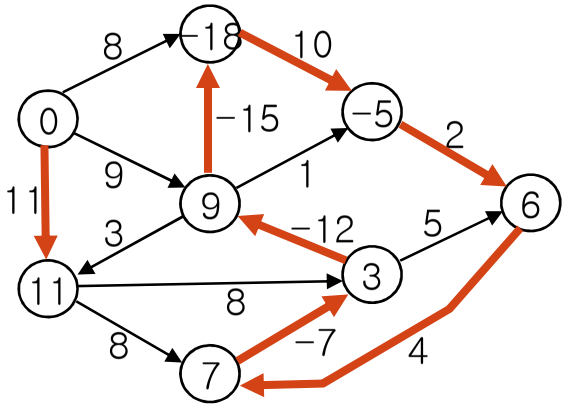
(g) $i=6$



(h) $i=7$



(i)



Example: Interwoven Strings

- Suppose you are given three strings of characters: X, Y , and Z, where $|X| = n$, $|Y| = m$, and $|Z| = n + m$. Z is said to be a shuffle of X and Y iff Z can be formed by interleaving the characters from X and Y in a way that maintains the left-to-right ordering of the characters from each string.

1. Show that cchocohilaptes is a shuffle of chocolate and chips, but chocochilatspe is not.
2. The strings abac and bbc occur interwoven in cabbabccdw.
3. Give an efficient dynamic-programming algorithm that determines whether Z is a shuffle of X and Y . (Hint: The values of the dynamic programming matrix you construct should be Boolean, not numeric.)