Dynamic Programming (DP)

Dae-Ki Kang

Background

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

Cons and Pros of Recursion

- Theory guys consider recursion to be good for these:
  - Sort algorithms such as Quick Sort, Merge Sort, etc.
  - Factorial (Well, actually NO in pratcice!)
  - DFS over graphs
  - …

  (REAL programmers avoid any recursion!)

- But bad for these:
  - Fibonacci numbers
  - Optimal order of matrices multiplication
  - …

Introduction: Fibonacci

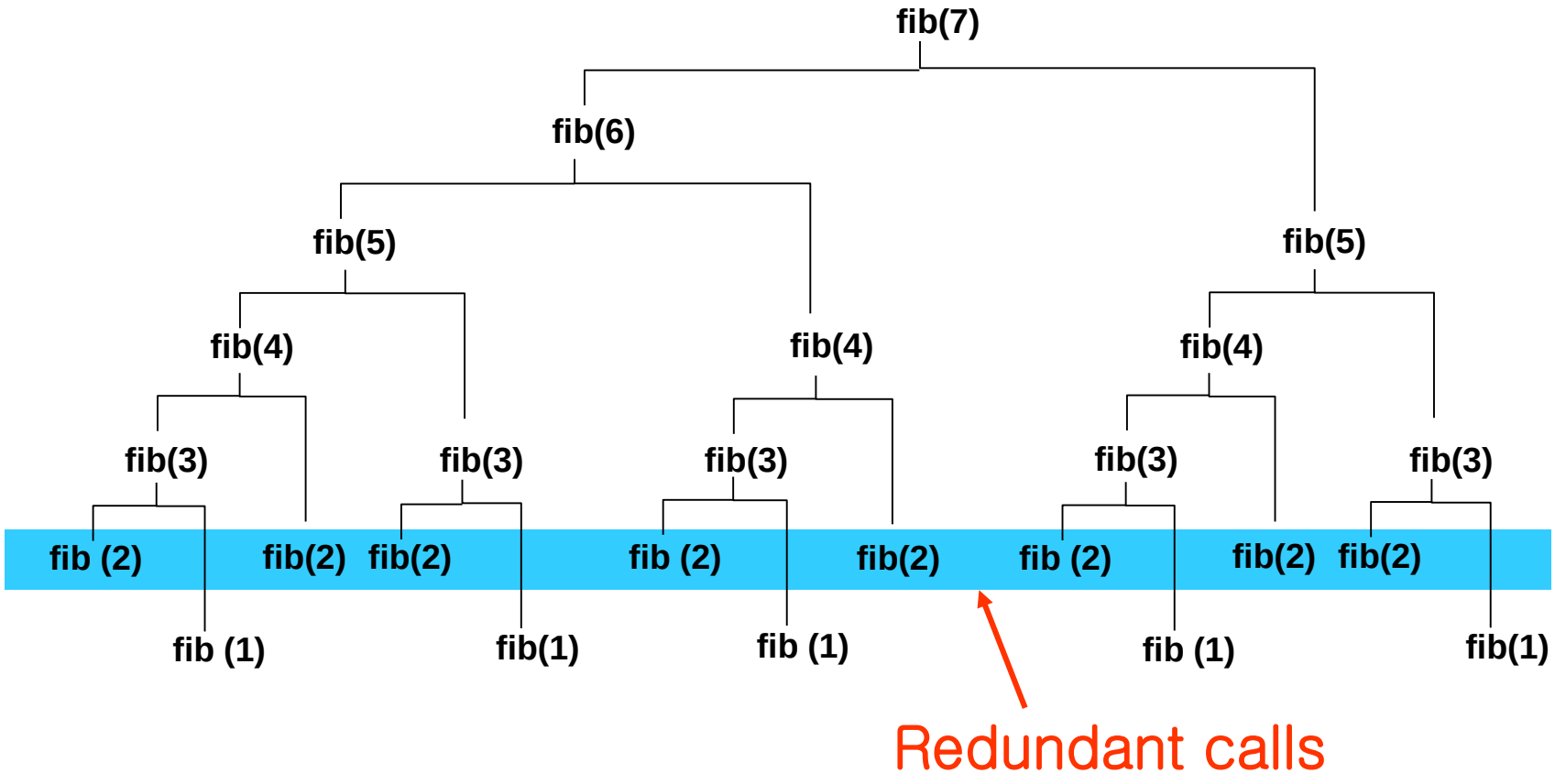
- $f_n = f_{n-1} + f_{n-2}$
- Very simple problem
  - But contains good motivations and implementation details of Dynamic programming
- Fibonacci is a fast growing function!

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

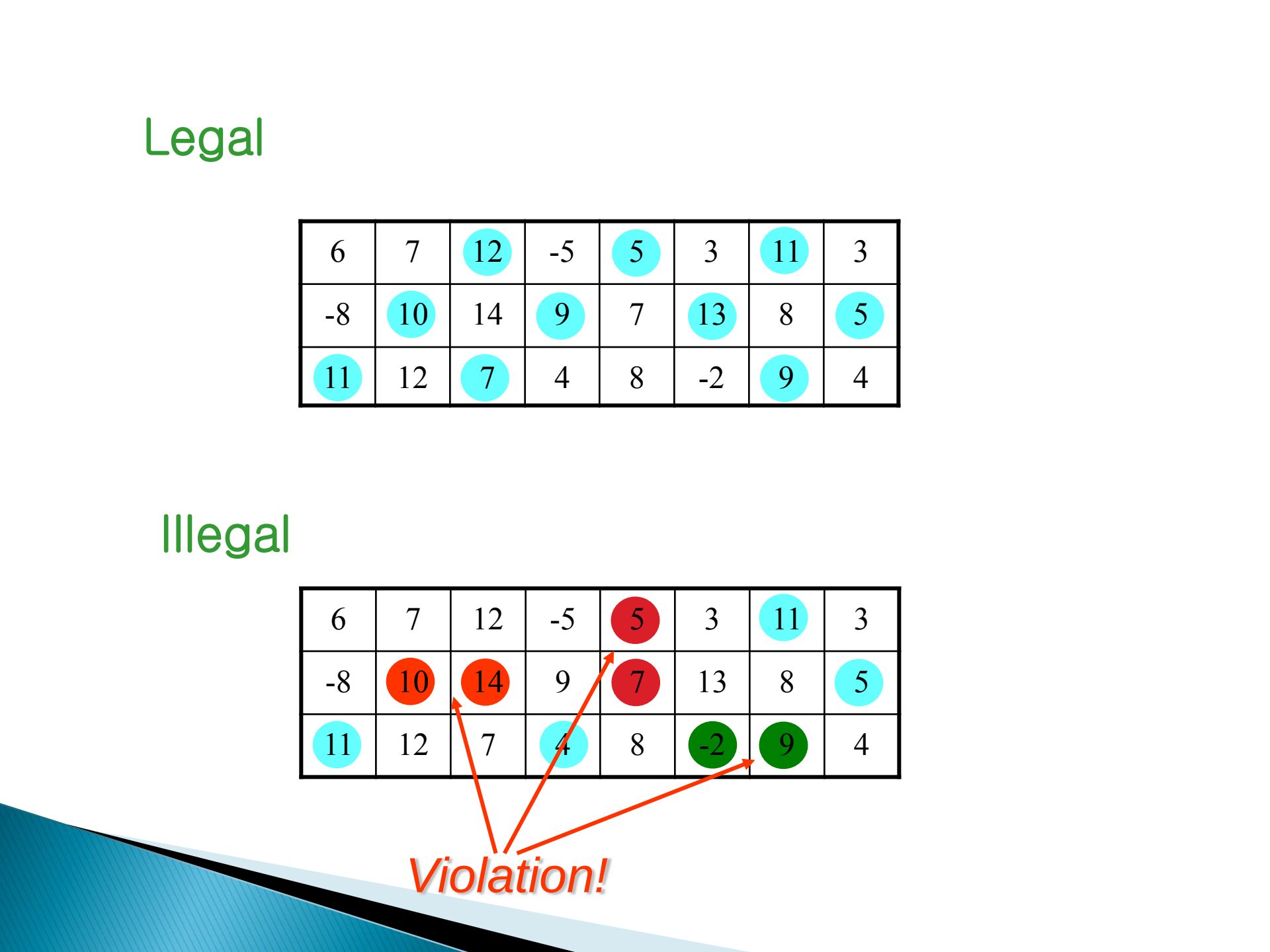
6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

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

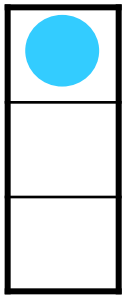
6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4



Possible patterns

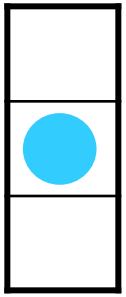
There are only four patterns to fill one column.

Pattern 1:



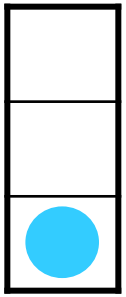
6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

Pattern 2:



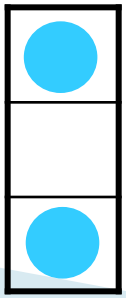
6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

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:

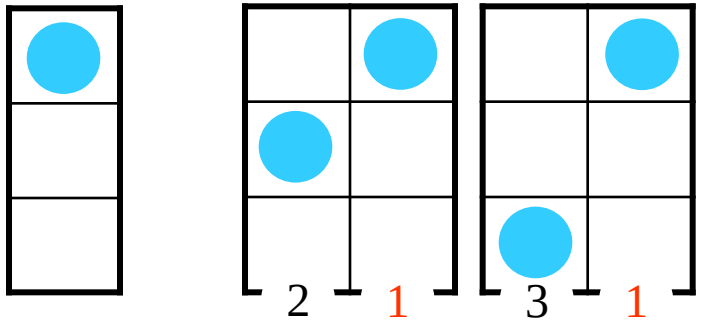


6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

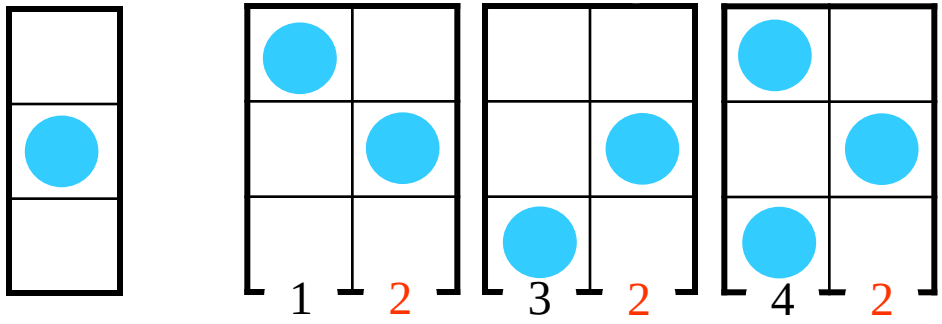
Compatible Patterns

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

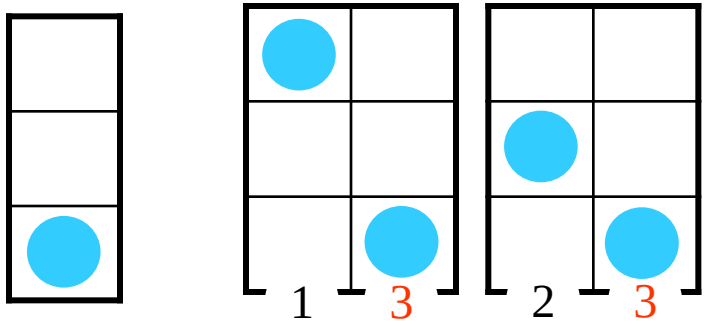
Pattern 1:



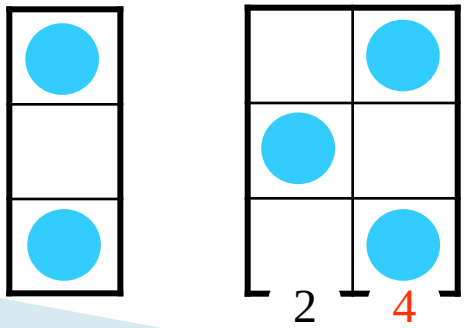
Pattern 2:



Pattern 3:



Pattern 4:



	i-1	*i*			
. . .	-5	5	3	11	3
	9	7	13	8	5
	4	8	-2	9	4

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

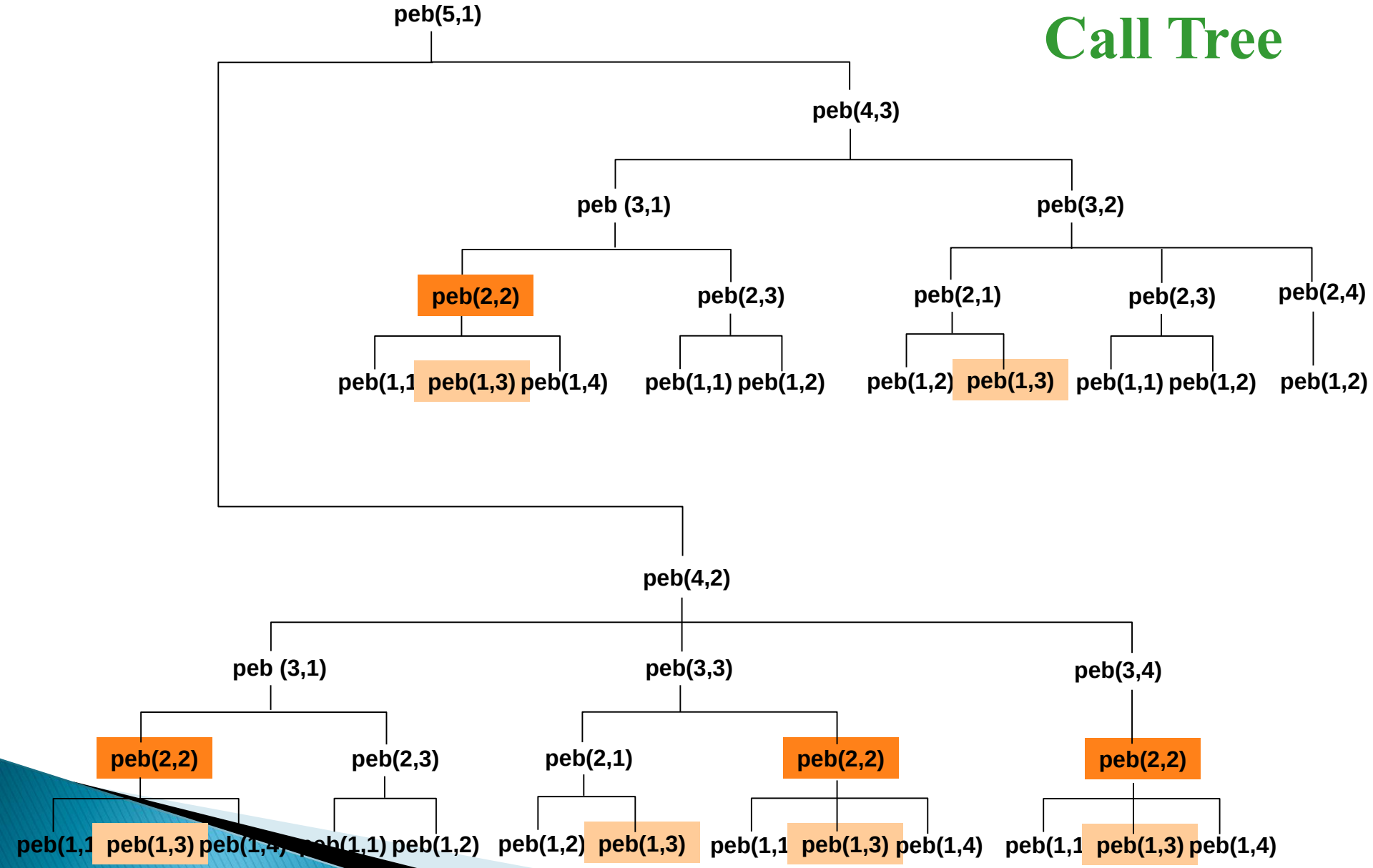
Recursive Version

```
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(1,1)
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

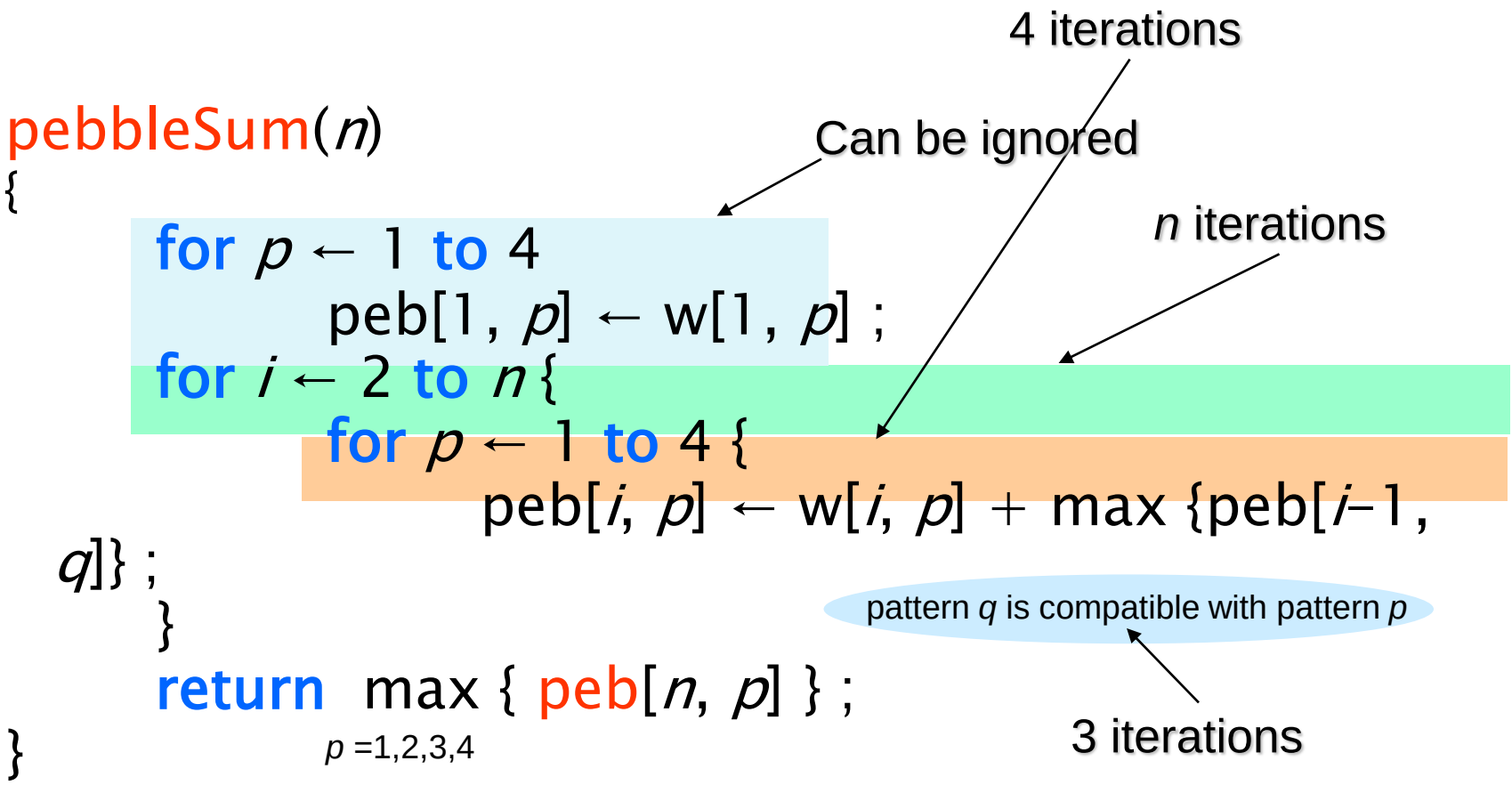
```
pebbleSum(n)
{
        for p ← 1 to 4
                peb[1, p] ← w[1, p] ;
        for i ← 2 to n {
                for p ← 1 to 4 {
                        peb[i, p] ← w[i, p] + max {peb[i–1, q]} ;
        }
        return  max { peb[n, p] } ;
}
```

For each pattern q which is compatible with pattern p

p =1,2,3,4

✓Complexity : $O(n)$

Complexity



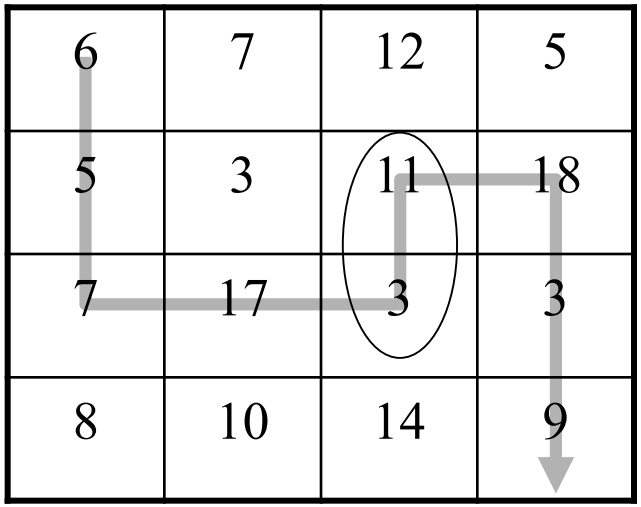
✓Complexity : $O(n)$

$n * 4 * 3 = O(n)$

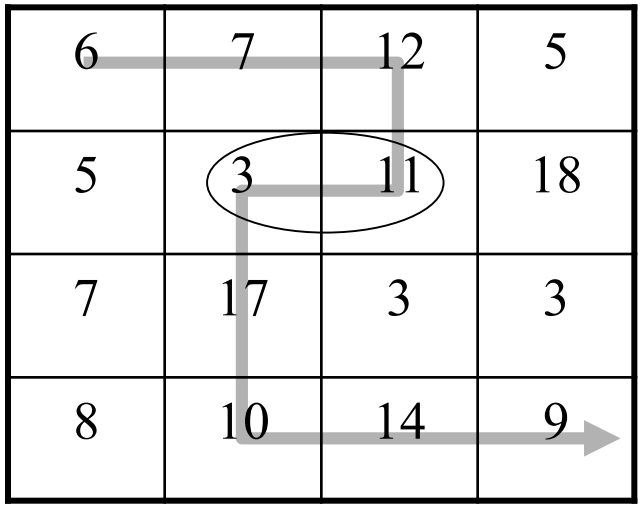
Example 2: Path in the array

- 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



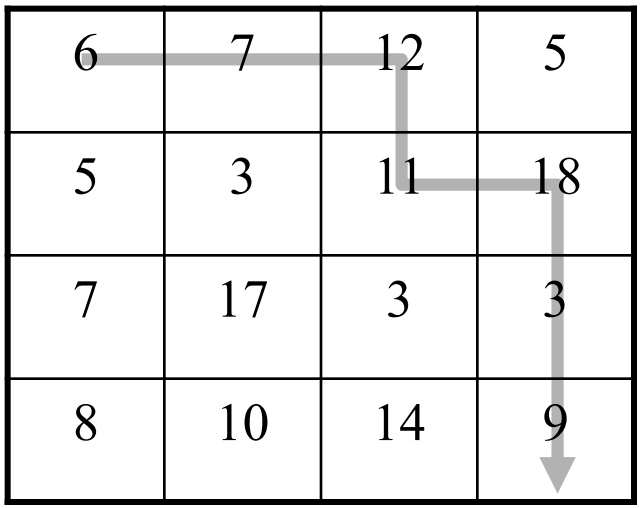
Going up



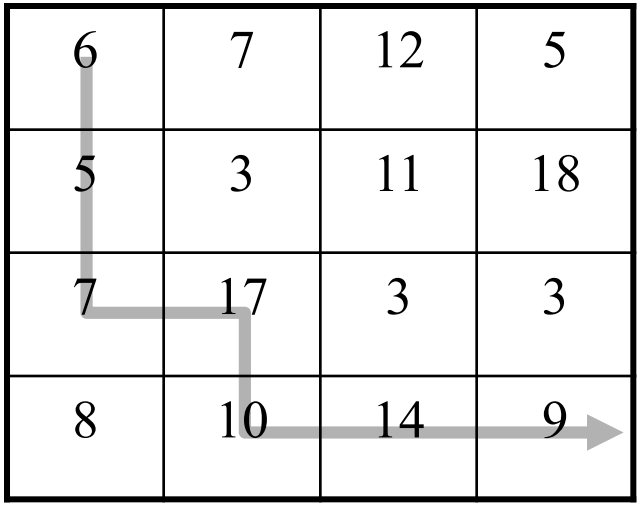
Going left

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)

{

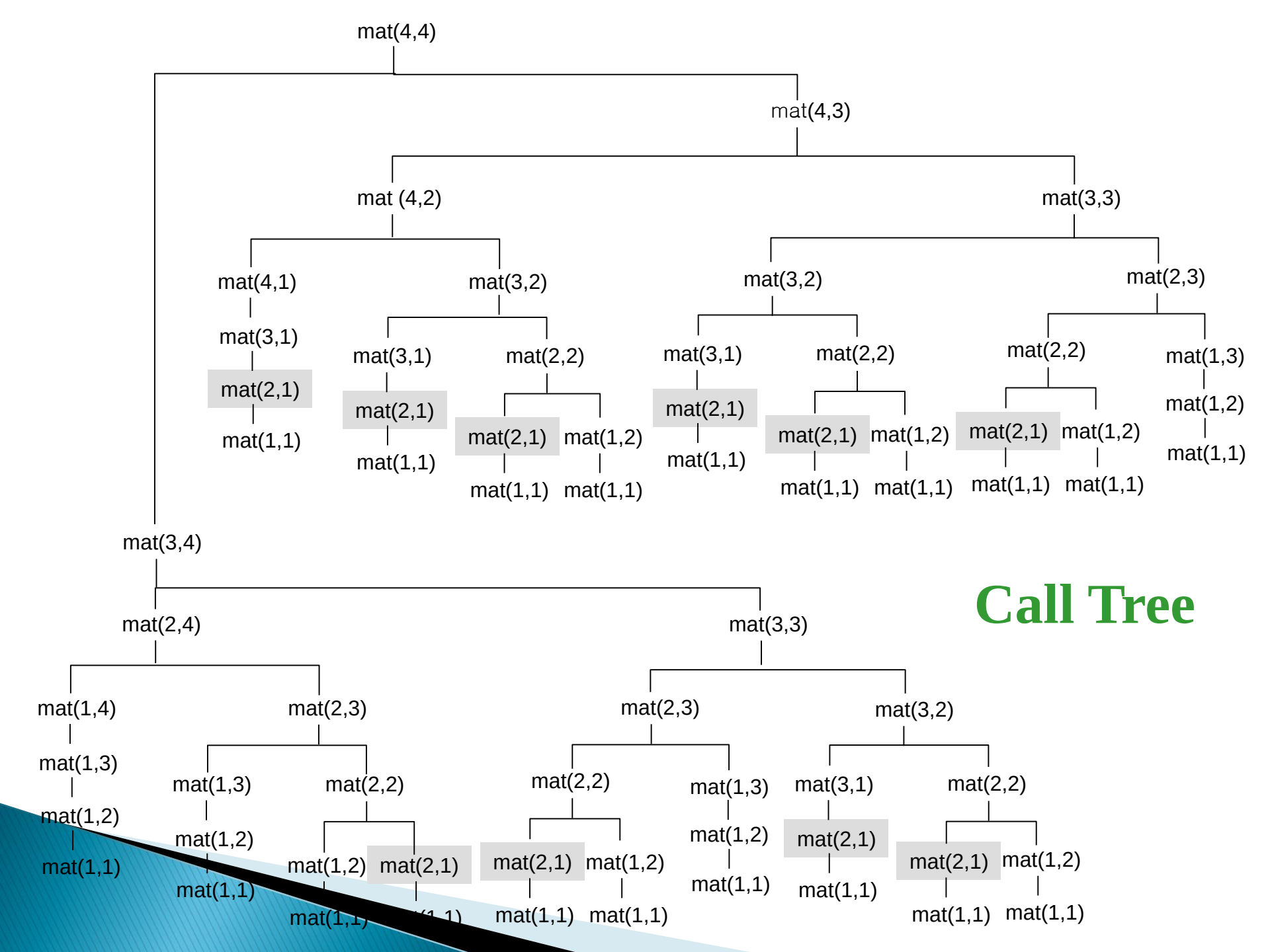
if ($i = 1$ and $j = 1$) then return m_{ij};

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

DP

matrixPath(*i*, *j*)

▷ The max sum at the cell (*i*, *j*)

```
{
        c[1,1] ← m11 ;
        for i ← 2 to n
                c[i,1] ←  mi1 + c[i–1,1];
        for j ← 2 to n
                c[1, j] ← m1j + c[1, j–1];
        for i ← 2 to n
                for j ← 2 to n
                        c[i, j] ← mij + max(c[i–1, j], c[i, j–1]);
        return c[n, n];
}
```

Example 3: 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 $X_m = <x_1 x_2 \ldots x_m>$ and $Y_n = <y_1 y_2 \ldots y_n>$
  - 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 }

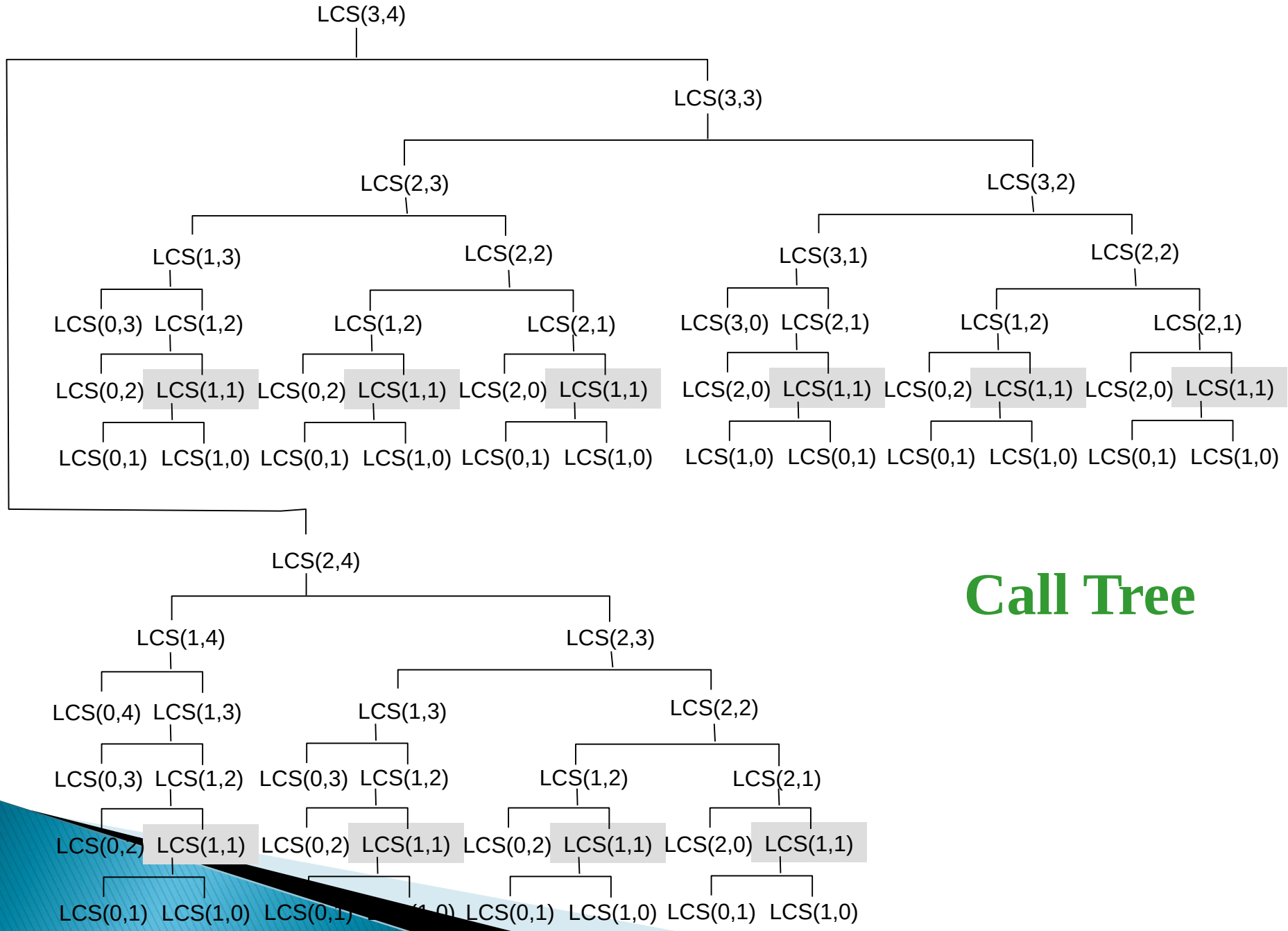
- $$c_{ij} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c_{i-1,j-1} + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{c_{i-1,j}, c_{i,j-1}\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

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

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(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 4 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.

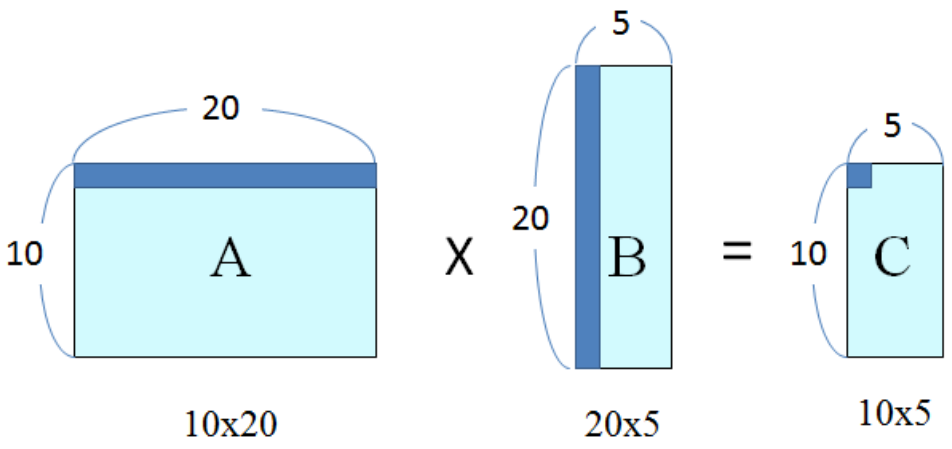
1. Show that cchocohilaptes is a shuffle of chocolate and chips, but chocochilatspe is not.
2. The strings abac and bbc occur interwoven in cabbabccdw.
3. Give an efficient dynamic-programming algorithm that determines whether Z is a shuffle of X and Y . (Hint: The values of the dynamic programming matrix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

Example 5: Matrix-Chain Multiplication

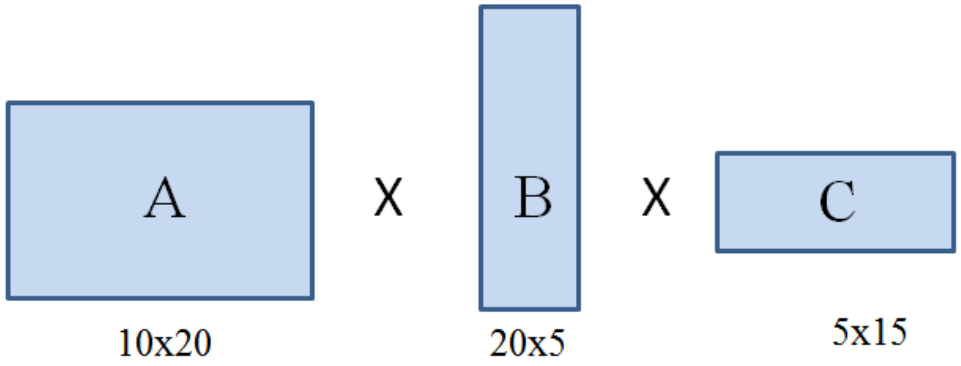
- Matrices A, B, C
  - (AB)C = A(BC)
- Ex: A:10 x 100, B:100 x 5, C:5 x 50
  - (AB)C: 7500 times multiplications
  - A(BC): 75000 times multiplications
- What is the best order to multiply A_1, A_2, A_3, …, A_n?

Detailed Explanation

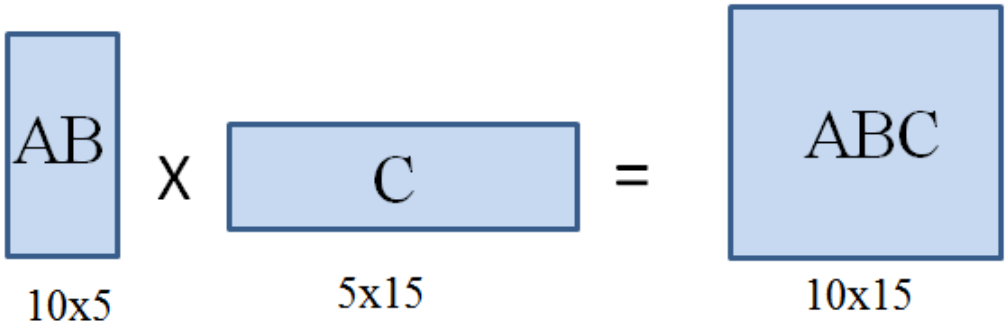
- Finding the minimum # of multiplications for
- chained matrix multiplications.
- To multiply A: 10x20 and B: 20x5, the total # of multiplications is 10x20x5 = 1,000. And the dimension of C=(AB) is 10x5.



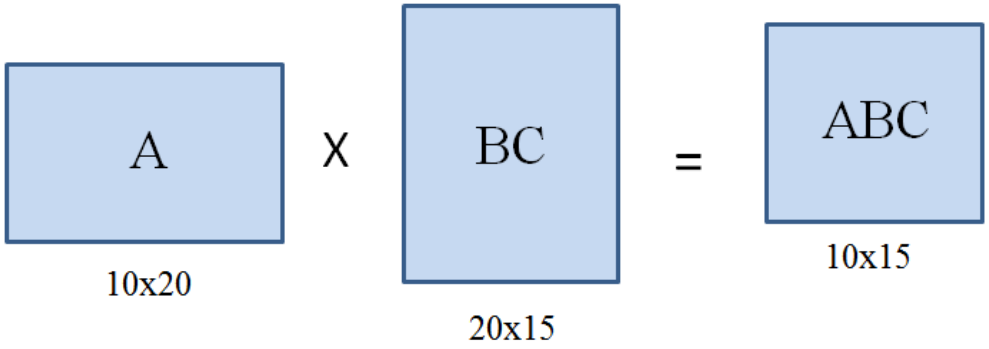
- For 1 element in C, 20 elements in a row of A and 20 element in a column of B are multiplied each other, thus 20 multiplications.
- To multiply 3 matrices, we can use associative law.
- AxBxC = (AxB)xC = Ax(BxC)
- Suppose A: 10x20, B: 20x5, C: 5x15



- For AxB and then C:
- 10x20x5 = 1,000 multiplications for AxB
- Since AxB is 10x5, (AB)xC needs 10x5x15 = 750 multiplications.
- Total of 1,000+750 = 1,750 multiplications.



- If we do BxC first, 20x5x15 = 1,500 multiplications and the result is 20x15.
- To calculate Ax(BC), 10x20x15 = 3,000 multiplications.
- Total is 1,500+3,000 = 4,500.



- Note that for the same result, the difference of multiplications is 4,500–1,700 = 2,800.
- Since multiplication is an expensive operation, we need to find an optimal order of multiplications.

- Size of subproblems # of subproblems

1	A B C D E	5
2	AxB BxC CxD DxE	4
3	AxBxC BxCxD CxDxE	3
4	AxBxCxD BxCxDxE	2
5	A x B x C x D x E	1

Recursive Relation

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

- m[i, j]: The cost of multiplying A_i, A_{i+1}, …, 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;
}
```

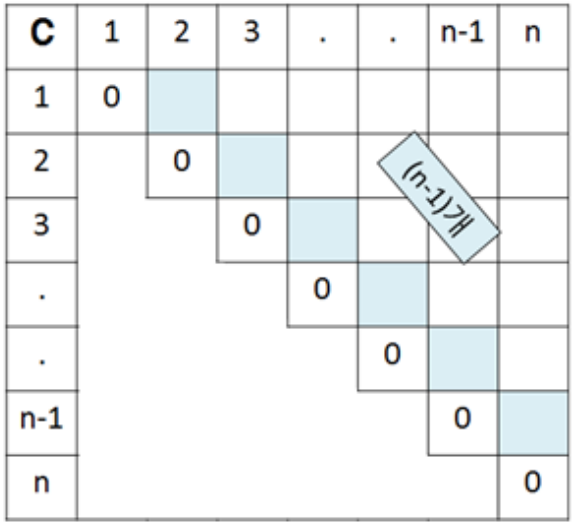
✓ Too many redundant recursive calls!

DP

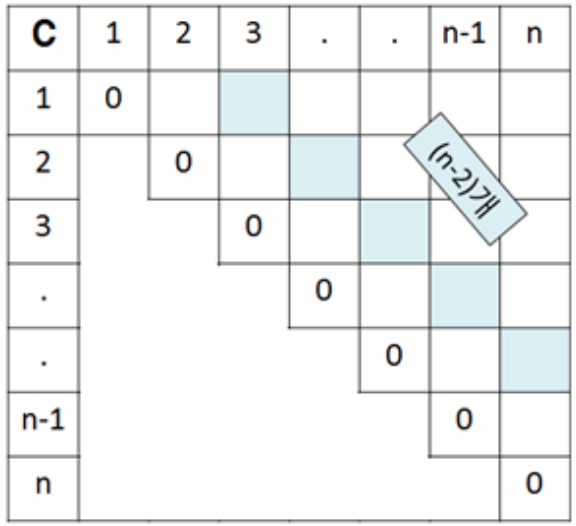
```
matrixChain(i, j)
{
        for i ← 1 to n
                C[i, i] ← 0;          ▷ 0 when there is only one matrix
        for L← 1 to n–1               ▷ Size of the problem is L+1
                for i ← 1 to n–L {
                        j ← i+L;
                        C[i, j] ← min {C[i, k]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

✓ Complexity: $\Theta(n^3)$

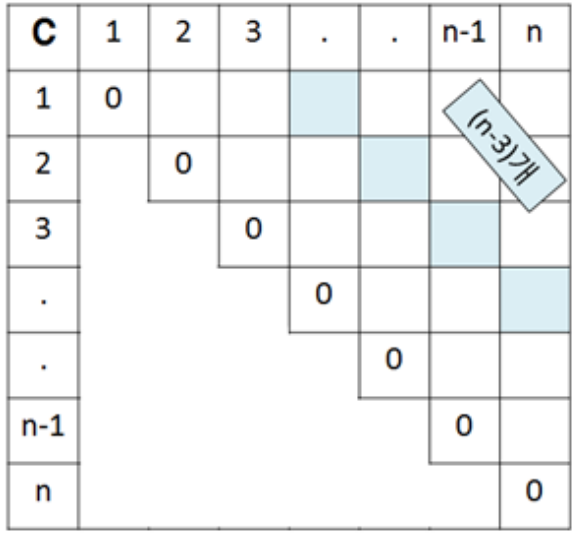
L = 1



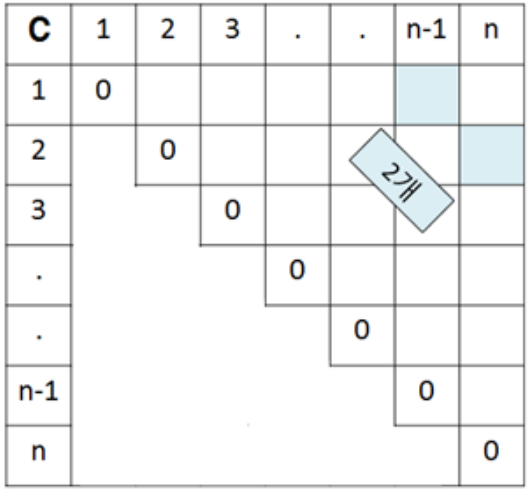
L = 2



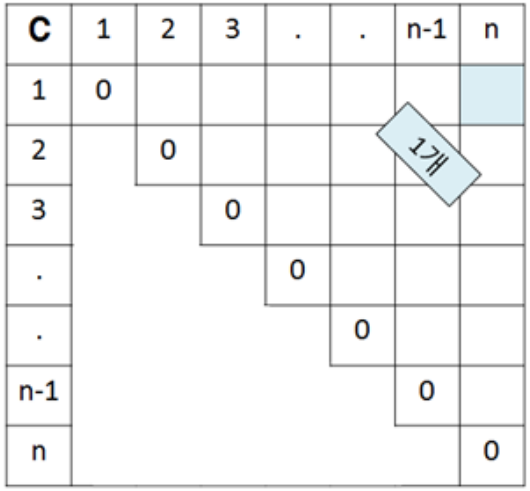
L = 3

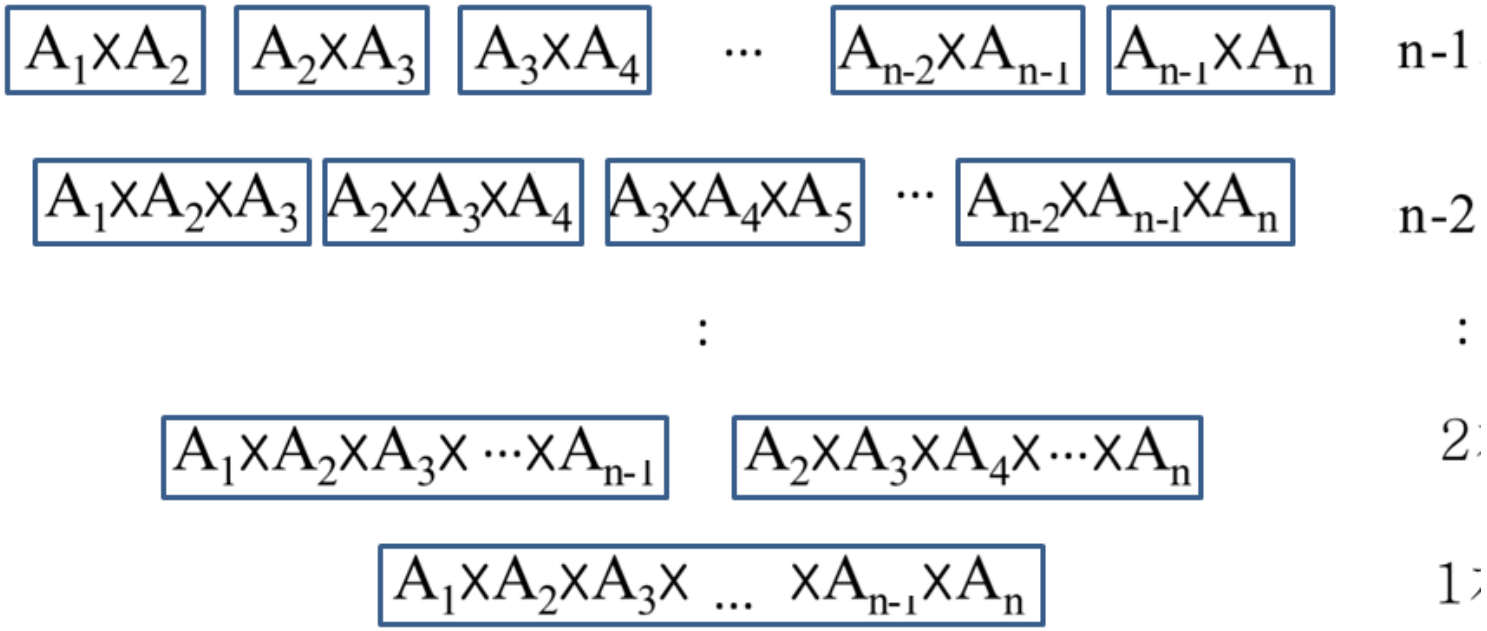


L = n-1

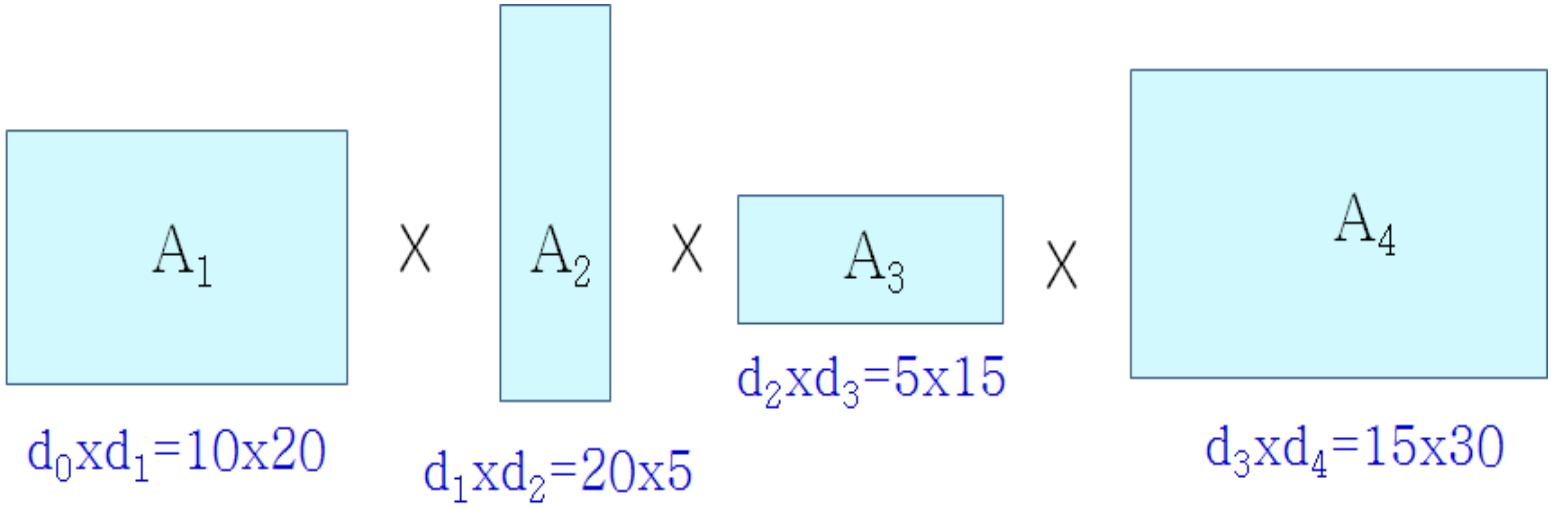


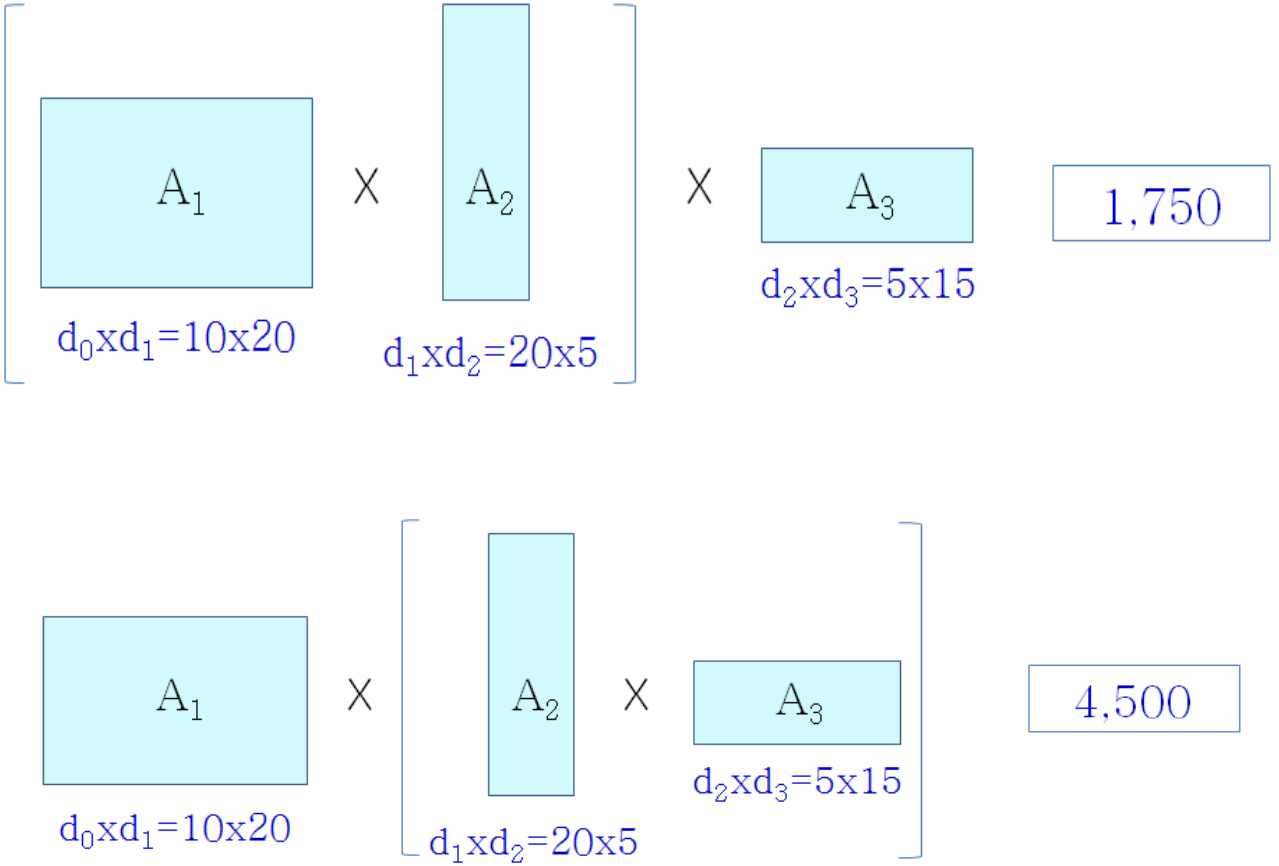
L = n

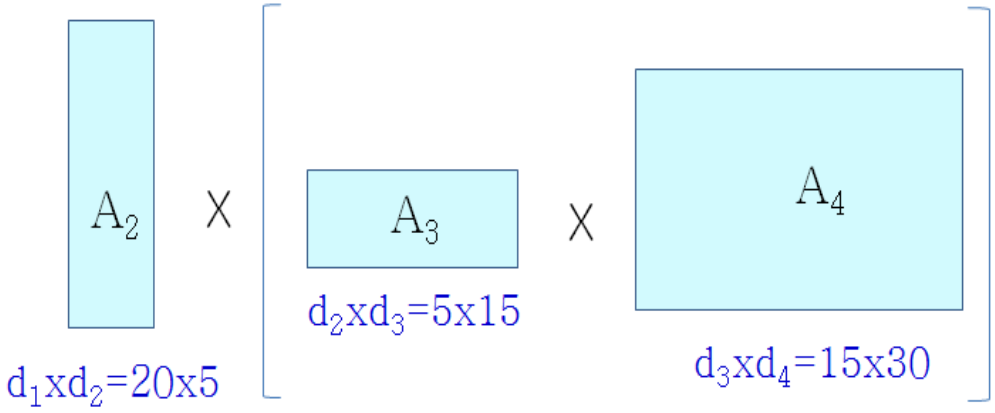



$A_1 \times A_2$
$A_2 \times A_3$
$A_3 \times A_4$
$\cdots$
$A_{n-2} \times A_{n-1}$
$A_{n-1} \times A_n$
n-1
$A_1 \times A_2 \times A_3$
$A_2 \times A_3 \times A_4$
$A_3 \times A_4 \times A_5$
$\cdots$
$A_{n-2} \times A_{n-1} \times A_n$
n-2
:
:
$A_1 \times A_2 \times A_3 \times \cdots \times A_{n-1}$
$A_2 \times A_3 \times A_4 \times \cdots \times A_n$
2
$A_1 \times A_2 \times A_3 \times \ldots \times A_{n-1} \times A_n$
1

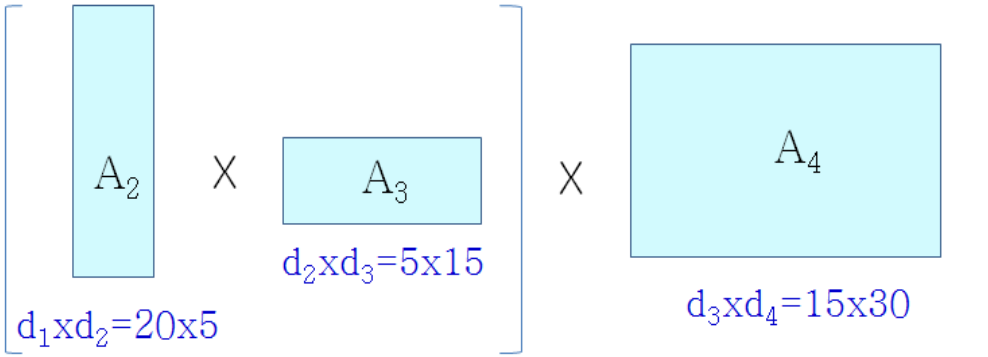
- A_1 10x20
- A_2 20x5
- A_3 5x15
- A_4 15x30



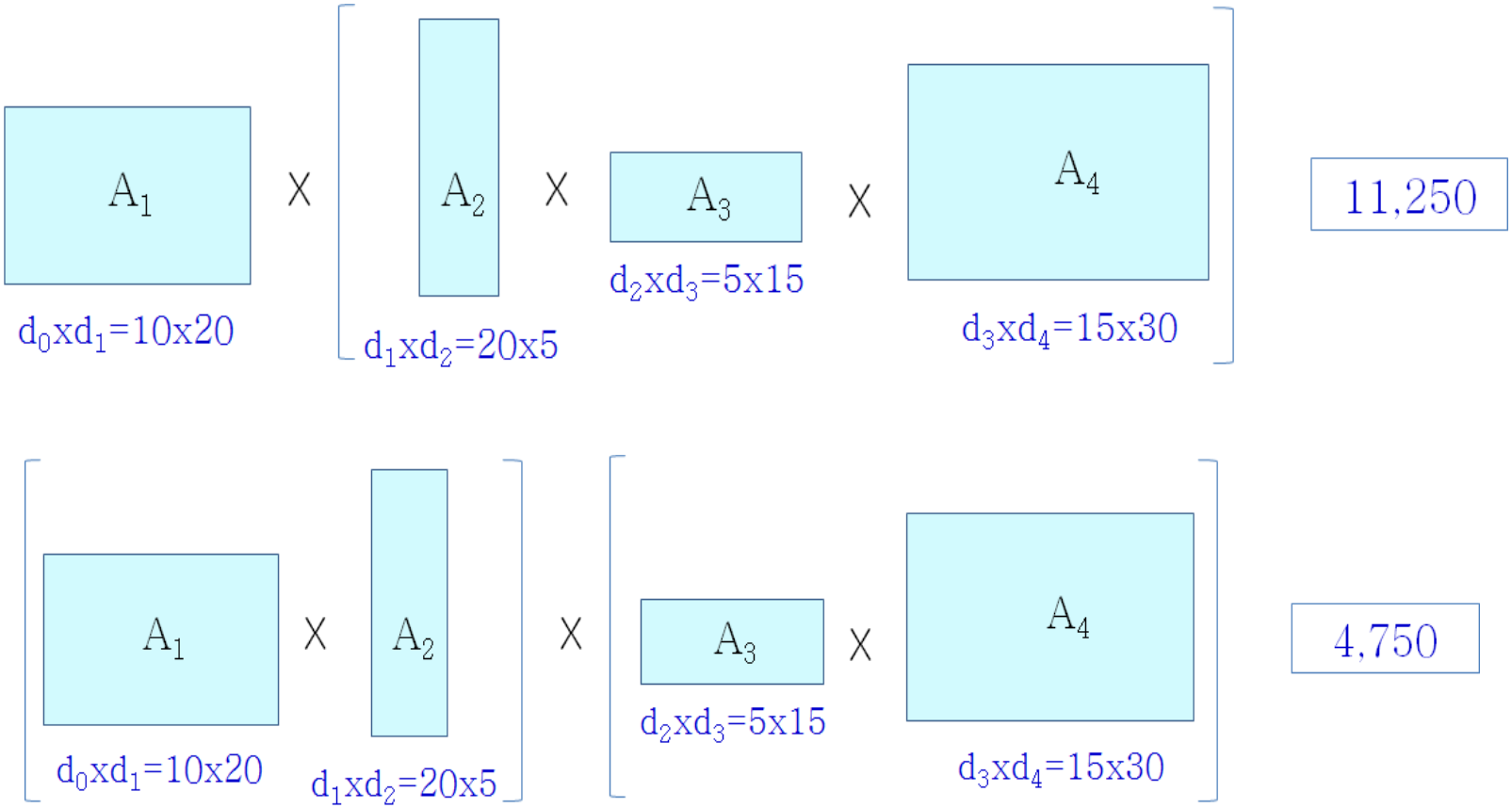

A1
X
A2
X
A3
1,750
d0xd1=10x20
d1xd2=20x5
d2xd3=5x15
A1
X
A2
X
A3
4,500
d0xd1=10x20
d1xd2=20x5
d2xd3=5x15

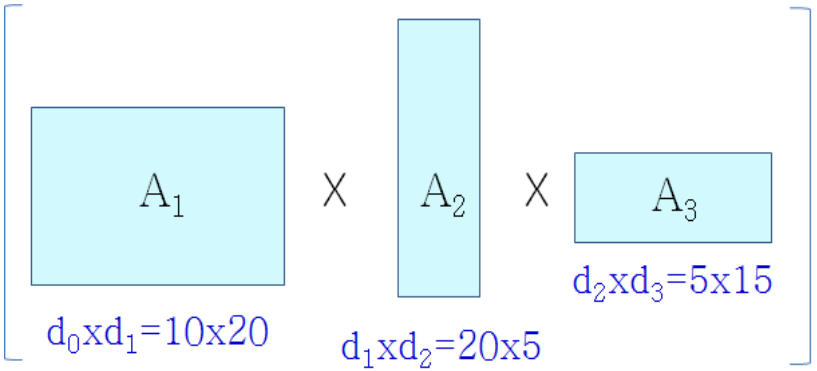

A2
X
A3
X
A4
d1xd2=20x5
d2xd3=5x15
d3xd4=15x30

5,250


A2
X
A3
X
A4
d1xd2=20x5
d2xd3=5x15
d3xd4=15x30

10,500


A1
X
A2
X
A3
X
A4
d0xd1=10x20
d1xd2=20x5
d2xd3=5x15
d3xd4=15x30
11,250
A1
X
A2
X
A3
X
A4
d0xd1=10x20
d1xd2=20x5
d2xd3=5x15
d3xd4=15x30
4,750



6,250

C	1	2	3	4
1	0	1,000	1,750	4,750
2		0	1,500	5,250
3			0	2,250
4				0

Example

- Show how to multiply this matrix chain optimally

Matrix	Dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

- Solution on the board
  - Minimum cost 15,125
  - Optimal parenthesization $((A_1(A_2A_3))((A_4\,A_5)A_6))$

Matrix-Chain multiplication (cont.)

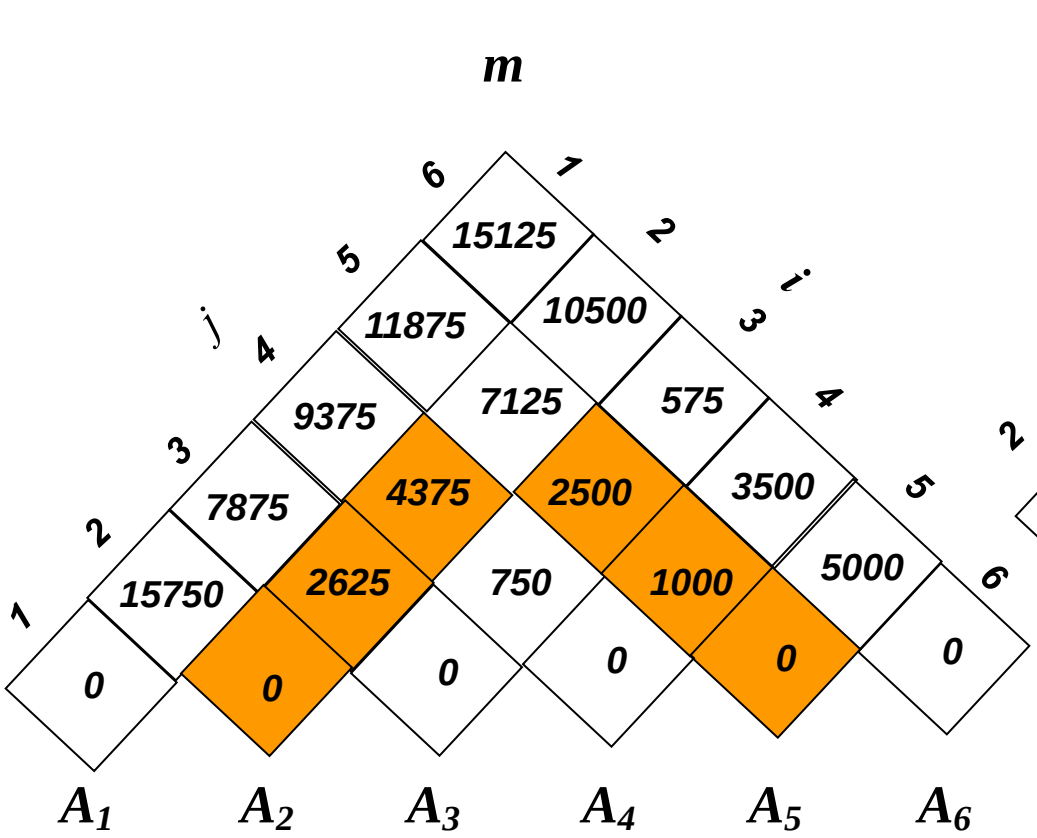
An example:

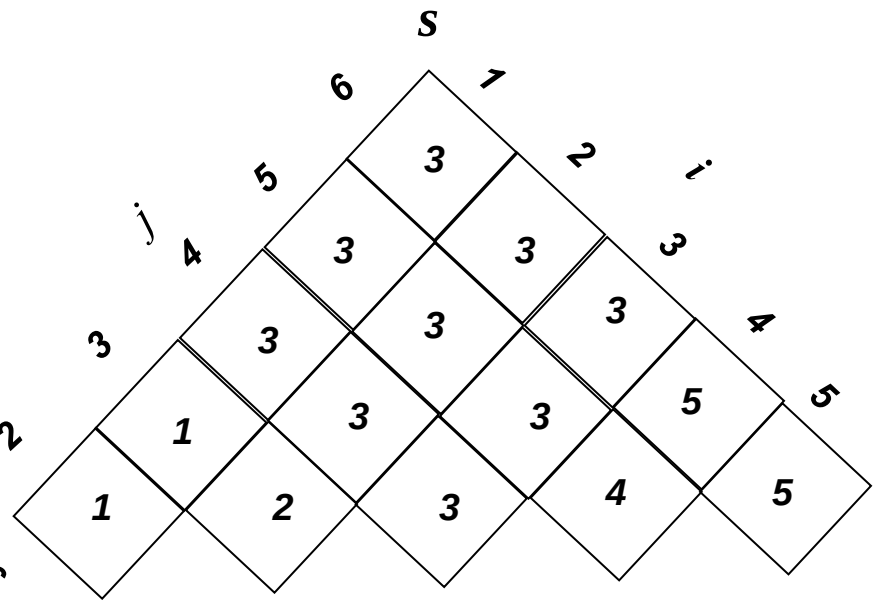
matrix	dimension
A_1	*30* x 35
A_2	*35* x 15
A_3	*15* x 5
A_4	*5* x 10
A_5	*10* x 20
A_6	*20* x 25

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 100 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2,4] + m[5,5] + p_1 p4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$

$$= (7125)$$

Matrix-Chain multiplication (cont.)





Time Complexity

- For 2 D nxn matrix C, # of elements is n^2, and we need to calculate $n^2/2$ elements.
- For one element, C[i,j], k-loop is iterated at most (n-1) times.
- $O(n^2)xO(n) = O(n^3)$

Example 6: Edit Distance

- Basic edit operations (insert, delete and substitute) are needed to convert a string S to the other string T.
- Edit distance is the minimum # of edit operations to convert S to T.

- For example, we edit ‘strong’ into ‘stone’.

s	t	_	r	o	n	g
		insert	delete	delete		substitute
s	t	o	_	_	n	e

- We don’t change the first ‘s’ and ‘t’.
- We insert ‘o’.
- We delete ‘r’ and ‘o’.
- We don’t change ‘n’.
- We replace ‘g’ by ‘e’.
- Total of 4 operations are used.

- Another way is as follows:

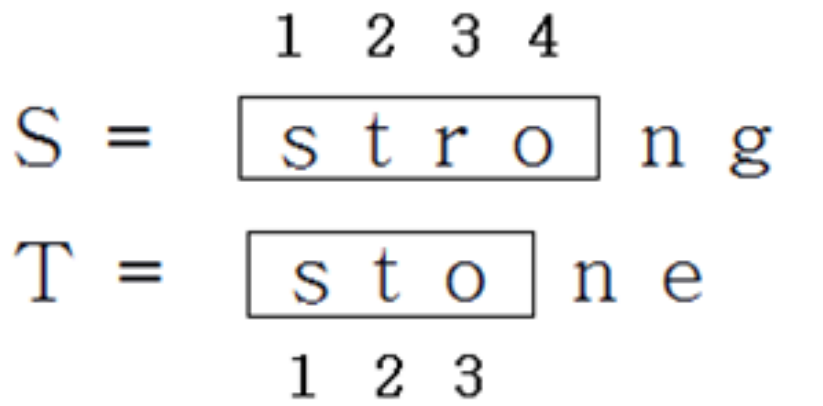
  s t r o n g

  delete substitute

  s t _ o n e

- We don't change the first 's' and 't'.
- We delete 'r'.
- We don't change 'o'.
- We replace 'g' by 'e'.
- Total of 2 operations are used.
-
- The # of edit operations needed can vary depending on the order of operations.

- How to represent subproblems of DP for edit distance problem?
- If we already know the minimum edit distance between the prefixes ‘stro’ and ‘sto’, we calculate the edit distance between the remaining suffixes ‘ng’ and ‘ne’ to obtain the whole edit distance.



- Let m be the length of S
- Let n be the length of T
- Each character of S and T to be S_i and t_j, where i = 1, 2, ···, m and j = 1, 2, ···, n.

  $S = s_1\ s_2\ s_3\ s_4 \cdots s_m$

  $T = t_1\ t_2\ t_3\ t_4 \cdots t_n$

- Definition of subproblems: E[i,j] to be a minimum # of edit operations (i.e. edit distance) to convert the first i characters of S to the first j characters of T.

- For example, for the strings ‘strong’ and ‘stone’, the edit distance from ‘stro’ to ‘sto is E[4,3].
- The solution of the problem is E[6,5].
- We calculate the edit distances of the following subproblems.

	1	2	3	4	5	6
S	s	t	r	o	n	g
T	s	t	o	n	e	

- $s_1 \rightarrow t_1$ [‘s’ to ‘s’]: E[1,1]=0. $s_1 = t_1 =$ ‘s’.
- $s_1 \rightarrow t_1 t_2$ [‘s’ to ‘st’]: E[1,2]=1. $s_1 = t_1 =$ ‘s’ and 1 operation to insert ‘t’.
- $s_1 s_2 \rightarrow t_1$ [‘st’ to ‘s’]: E[2,1]=1. $s_1 = t_1 =$ ‘s’, and 1 operation to delete ‘t’.
- $s_1 s_2 \rightarrow t_1 t_2$ [‘st’ to ‘st’]: E[2,2]=0. $s_1 = t_1 =$ ‘s’ and $s_2 = t_2 =$ ‘t’. We already calculated E[1,1]=0, and $s_2 = t_2 =$ ‘t’, therefore E[1,1]+0=0.

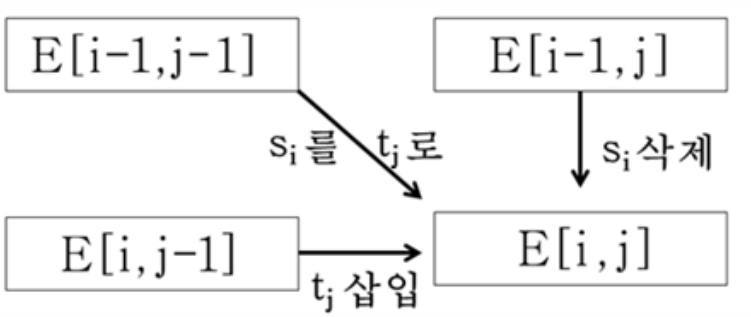
- Subproblem $s_1s_2s_3s_4 \rightarrow t_1t_2t_3$ [‘stro’ to ‘sto’]: how to calculate E[4,3]?

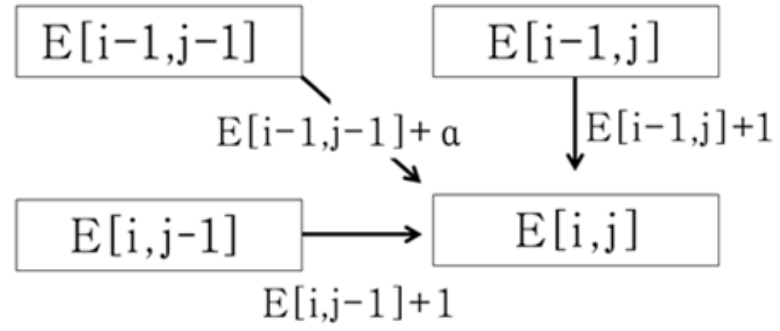
- $s_1s_2s_3s_4 \rightarrow t_1t_2$ [‘stro’ to ‘st’]: if we know E[4,2], we insert t_3=‘o’. So, # of edit distances E[4,2]+1.
- $s_1s_2s_3 \rightarrow t_1t_2t_3$ [‘str’ to ‘sto’]: if we know E[3,3], we delete s_4=‘o’. So, # of edit distances E[3,3]+1.
- $s_1s_2s_3 \rightarrow t_1t_2$ [‘str’ to ‘st’]: if we know E[3,2], we need to calculate the operations to edit s_4=‘o’ to t_3=‘o’. We don’t need to edit because the two characters are the same. Therefore the edit distance for this is E[3,2]+0.

		E	T	ε	s	t	o
	S	i \ j	0	1	2	3
	ε	0	0	1	2	3
	s	1	1	0	1	2
	t	2	2	1	0	1
	r	3	3	2	1	1
	o	4	4	3	2	1

- The edit distance of E[4,3] can be obtained from the three subproblems E[4,2], E[3,3], E[3,2]. Since E[4,2]=2, E[3,3]=1, E[3,2]=1, the edit distance of E[4,3] is 1 which is the minimum among (2+1), (1+1), and (1+0).

- If E[i-1,j], E[i,j-1], E[i-1,j-1] are calculated, we can get E[i,j]. Thus the implicit order of subproblems in eidt distance is as follows:





- Left of E[i,j]: E[i,j−1] is for $s_1 s_2 \cdots s_i$ and $t_1 t_2 \cdots t_{j-1}$, and we insert t_j then (E[i,j−1]+1) is for $s_1 s_2 \cdots s_i$ to $t_1 t_2 \cdots t_j$.
- Up of E[i,j]: E[i−1,j] is for $s_1 s_2 \cdots s_{i-1}$ and $t_1 t_2 \cdots t_j$, and we delete s_i then, (E[i−1,j]+1) is for $s_1 s_2 \cdots s_i$ to $t_1 t_2 \cdots t_j$.
- Upper left of E[i,j]: the # of operations is (E[i−1,j−1]+α), where $\alpha=0$ if $s_i=t_j$, otherwise $\alpha=1$. If s_i is t_j, we don't need edit operations, and if s_i is not t_j, we need replacement operation.

- So, from the previous three cases, we choose the minimum value as E[i,j].

$$E[i,j] = \min\{E[i,j-1]+1,\ E[i-1,j]+1,\ E[i-1,j-1]+\alpha\}$$

where, $\alpha=1$ if $s_i \neq t_j$, else $\alpha=0$

- For the above equation, we initialize E[0,0], E[1,0], E[2,0], ···, E[m,0] and E[0,1], E[0,2], ···, E[0,n] as follows:

	T	ε	t_1	t_2	t_3	..	t_n
S		0	1	2	3	..	n
ε	0	0	1	2	3	..	n
s_1	1	1					
s_2	2	2					
s_3	3	3					
.	.	.					
.	.	.					
s_m	m	m					

- 0^{th} row of E is initialized as 0, 1, 2, ⋯, n. Each value is the number of insert operations to construct T's prefix from null string (S=ϵ).
- E[0,0]=0, this is before making the first character of T, so no operations needed.
- E[0,1]=1, we construct T's first character by inserting 't_1'.
- E[0,2]=2, we construct T's first 2 characters by inserting 't_1t_2'.

  …
- E[0,n]=5, we construct T by insrerting '$t_1t_2t_3\cdots t_n$'.

- 0^{th} column of E is initialized to 0, 1, 2, ⋯, m. This means the # of delete operations for converting S into T(=ϵ).
- E[0,0]=0, no operation needed because it is before deleting the 1^{st} character of S.
- E[1,0]=1, we delete 's_1', the 1^{st} character of S to make T=ϵ.
- E[2,0] = 2, we delete the 1^{st} two characters 's_1s_2' of S to make T=ϵ.

  …
- E[m,0]=m, , we delete all the m characters of S to make T=ϵ.

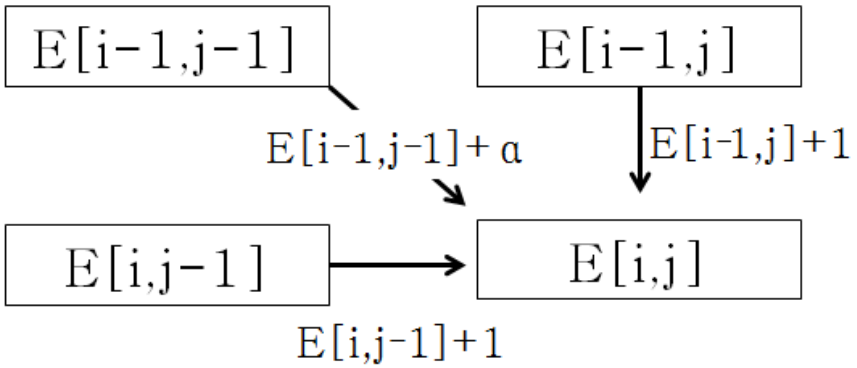
Algorithm

EditDistance

Input: strings S and T, the lengths of S, T are m, n.
Output: edit distance from S to T, E[m,n]

1. for i=0 to m E[i,0]=i // initialization of 0th column
2. for j=0 to n E[0,j]=j // initialization of 0th row
3. for i=1 to m
4. for j=1 to n
5. E[i,j] = min{E[i,j-1]+1, E[i-1,j]+1, E[i-1, j-1]+α}
6. return E[m,n]

- Line 1~2: initialize 0^{th} row and 0^{th} column of the array.
- Line 3~5: we fill the 1^{st} row, the 2^{nd} row, etc. The value of E[i,j] is the minimum of (left element+1), (upper element+1), and (upper left element+α).



- The following matrix has edit distances of EditDistance algorithm to change ‘strong’ into ‘stone’.

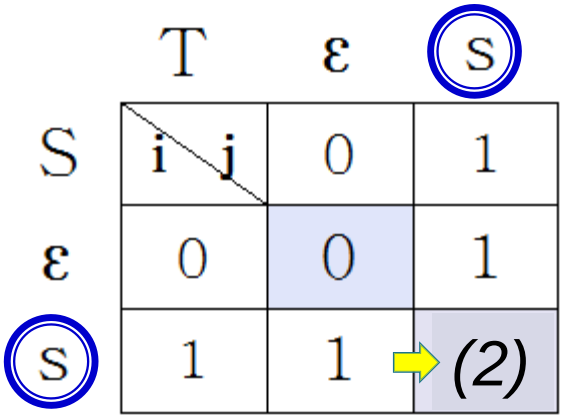
		E	T	ε	s	t	o	n	e
		S		0	1	2	3	4	5
	0	ε	0	0	1	2	3	4	5
	1	s	1	1	**0**	1	2	3	4
i	2	t	2	2	1	**0**	1	2	3
	3	r	3	3	2	**1**	1	2	3
	4	o	4	4	3	2	**1**	2	3
	5	n	5	5	4	3	2	**1**	2
	6	g	6	6	5	4	3	2	**2**

j

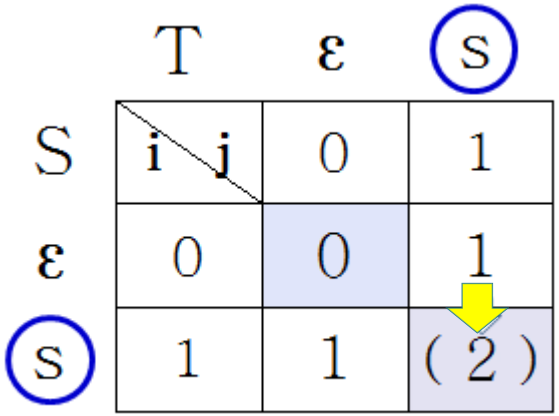
- We explain the blue entries.

		T	ε	s	t	o	n	e
	S		0	1	2	3	4	5
ε	0	0	1	2	3	4	5
s	1	1	**0**	1	2	3	4
t	2	2	1	**0**	1	2	3
r	3	3	2	**1**	1	2	3
o	4	4	3	2	**1**	2	3
n	5	5	4	3	2	**1**	2
g	6	6	5	4	3	2	**2**

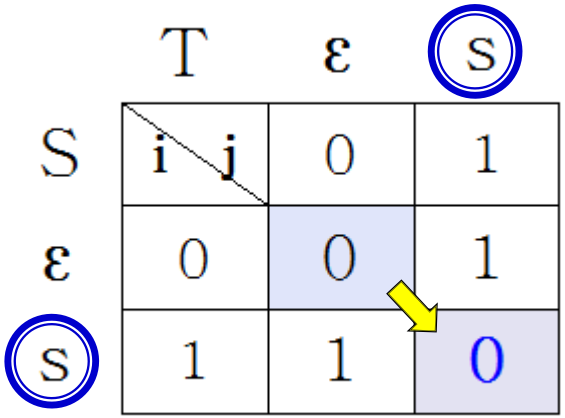
- E[1,1] = min{E[1,0]+1, E[0,1]+1, E[0,0]+α} = min{(1+1), (1+1), (0+0)} = 0
  - 'E[1,0]+1=2' means we delete the 1st character of S (E[1,0]=1) and insert('+1') the first character of T (t_1='s'). This means from the state T=ϵ, we insert the first character of T, 's'.



- ‘E[0,1]+1=2’: we insert the 1st character of T, s_1(E[0,1]=1), and delete(‘+1’) the 1st character of S(s_1=‘s’). This means the 1st character of T(‘s’) is already inserted and we remove the 1st character of S.



- E[0,0]+α = 0+0 = 0: α=0 because the 1st character of S is the same as the 1st character of T. They are the same, so no operation is needed.



- E[1,1] is the minimum of the three cases, 0.

- E[2,2] = min{E[2,1]+1, E[1,2]+1, E[1,1]+α} = min{(1+1), (1+1), (0+0)} = 0. The 1st character of T('s') is already made and since the 2nd character of S is the same as the 2nd character of T, no operations are need for 'st'.

		T	ε	s	t
S	i \ j	0	1	2
ε	0	0	1	2
s	1	1	0	1
t	2	2	1	0

- E[3,2] = min{E[3,1]+1, E[2,3]+1, E[2,2]+α} = min{2+1, 0+1, 1+1} = 1. The 1st two characters of T('st') is made and we remove the 3rd character 'r' of S.

		T	ε	s	t
	S	i \ j	0	1	2
	ε	0	0	1	2
	s	1	1	0	1
	t	2	2	1	0
r		3	3	2	1

- E[4,3] = min{E[4,2]+1, E[3,3]+1, E[3,2]+α} = min{(2+1), (1+1), (1+0)} = 1. The 1st two characters ('st') of T are made (with one delete) and since the 4th character of S is the same as the 3rd character of T, no operations are needed for 'sto'.

		T	ε	s	t	o
S	i \ j	0	1	2	3
ε	0	0	1	2	3
s	1	1	0	1	2
t	2	2	1	0	1
r	3	3	2	1	1
o	4	4	3	2	1

- E[5,4] = min{E[5,3]+1, E[4,4]+1, E[4,3]+α} = min{(2+1), (1+1), (1+0)} = 1. ‘sto’ is made and since 5th of S = 4th of T, no additional operations for ‘ston’.

		T	ε	s	t	o	n
	S	i \ j	0	1	2	3	4
	ε	0	0	1	2	3	4
	s	1	1	0	1	2	3
	t	2	2	1	0	1	2
	r	3	3	2	1	1	2
	o	4	4	3	2	1	2
	n	5	5	4	3	2	1

- E[6,5] = min{E[6,4]+1, E[5,5]+1, E[5,4]+α} = min{(2+1), (2+1), (1+1)} = 2.

	T	ε	s	t	o	n	e
S	i \ j	0	1	2	3	4	5
ε	0	0	1	2	3	4	5
s	1	1	0	1	2	3	4
t	2	2	1	0	1	2	3
r	3	3	2	1	1	2	3
o	4	4	3	2	1	2	3
n	5	5	4	3	2	1	2
g	6	6	5	4	3	2	2

Time complexity

- O(mn) where m and n is the lengths of the strings
- # of all subproblems are the # of entries of the matrix E, mxn.
- For each entry, we check at most three entries (up, left, upper left) and get the minimum, which takes O(1) time.

Applications

- If the edit distance between two strings are small, we consider the strings are similar. In bioinformatics, edit distance can be used to compare two DNA sequences or protein sequences to check how similar the two genomes are.
- For example, we can compare cancer-related proteins in the patient's protein sequences to examine cancers early.
- We can analyze the characteristics of cancer cells.
- We can find genes that have advantageous traits.
- Spell Checker, Correction System in Optical Character Recognition, Natural Language Translation

Example 7: Coin Change Problem

- To find the minimum # of coins for the given change
- Greedy algorithm could work but not always.
- DP algorithm can produce the optimal solution for all cases.

- We need to find subproblems.
- Let the coins be d_1, d_2, ···, d_k, $d_1 > d_2 > \cdots > d_k = 1$.
- Change is n.
- In Korea, k=5, and $d_1 = 500$, $d_2 = 100$, $d_3 = 50$, $d_4 = 10$, $d_5 = 1$.
- In Knapsack DP problem, we increase the capacity of knapsack by 1kg.

- Similarly, we increment 1 won here.
- Change is the capacity of knapsack, and coin is the stuffs.
- We preserve the subsolutions in 1-d array, C.
- Change=1 won, The minimum # of coins for 1 won = C[1]
- Change=2 won, The minimum # of coins for 2 won = C[2]

  …
- The minimum # of coins for j won = C[j]

  …
- The minimum # of coins for n won = C[n]

- Check the implicit order of subproblems.
- What subproblems are needed for C[j]?
- For j won, the possible coins to add are (d_1=500, d_2=100, d_3=50, d_4=10, d_5=1).
- If 500 won coin is needed, we add one 500 won coin to the solution of (j-500) won C[j-500] = C[j-d_1]. That is C[j-500]+1.
- If 100 won coin is needed, we add one 100 won coin to the solution of (j-100) won C[j-100] = C[j-d_2]. That is C[j-100]+1.

- If 50 won coin is needed, we add one 50 won coin to the solution of (j-50) won C[j-50] = C[j-d_3]. That is C[j-50]+1.
- If 10 won coin is needed, we add one 10 won coin to the solution of (j-10) won C[j-10] = C[j-d_4]. That is C[j-10]+1.
- If 1 won coin is needed, we add one 1 won coin to the solution of (j-1) won C[j-1] = C[j-d_5]. That is C[j-1]+1.
- We choose C[j] to be the minimum of the above 5 cases as follows.

$$C[j] = \min_{1 \le i \le k}\{C[j-d_i] + 1\}, \text{ if } j \ge d_i$$

Algorithm

DPCoinChange

Input: Change n won, k coins and their values, $d_1 > d_2 > \cdots > d_k = 1$

Output: C[n]

```
1. for i = 1 to n C[i]=∞
2. C[0]=0
3. for j = 1 to n {
4.       for i = 1 to k {
5.            if (d_i ≤ j) and (C[j-d_i]+1<C[j])
6.                 C[j]=C[j-d_i]+1
       }
   }
7. return C[n]
```

- Line 1: initialize C ∞
- Line 2: C[0]=0.
- Line 3~6: in the for-loop, we increment j from 1 to n.
- Line 4~6: we set $C[j]=\min_{1 \le i \le k}\{C[j-d_i] + 1\}$.
- Note that $d_i \le j$ to be considered, otherwise always C[j]=0 for $d_i > j$.

- Example: $d_1=16$, $d_2=10$, $d_3=5$, $d_4=1$, $n=20$.

- Line 1~2: initialize C.

j	0	1	2	3	4	5	6	7	8	9	10	…	16	17	18	19	20
C	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	…	∞	∞	∞	∞	∞

- When j=1..4, only 1 won ($d_4=1$) can be considered, C[j]=C[j-1]+1 (adding 1 won from the previous solution).
- If i=4 (1 won coin), if condition ($1 \le j$) of line 5 is true and ($C[j-1]+1 < \infty$) is also true, so C[j] is as follows:

- Line 1~2: initialize C.

j	0	1	2	3	4	5	6	7	8	9	10	…	16	17	18	19	20
C	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	…	∞	∞	∞	∞	∞

- When j=1..4, only 1 won (d_4=1) can be considered, C[j]=C[j-1]+1 (adding 1 won from the previous solution).
- If i=4 (1 won coin), if condition ($1 \leq j$) of line 5 is true and ($C[j-1]+1 < \infty$) is also true, so C[j] is as follows:

- C[1] = C[j−1]+1 = C[1−1]+1 = C[0]+1 = 0+1 = 1

j	0	1
	0	∞

⇨

j	0	1
	0	1

- C[2] = C[j−1]+1 = C[2−1]+1 = C[1]+1 = 1+1 = 2

j	1	2
	1	∞

⇨

j	1	2
	1	2

- C[3] = C[j−1]+1 = C[3−1]+1 = C[2]+1 = 2+1 = 3

j	2	3
	2	∞

⇨

j	2	3
	2	3

- C[4] = C[j−1]+1 = C[4−1]+1 = C[3]+1 = 3+1 = 4

j	3	4
	3	∞

⇨

j	3	4
	3	4

- j=5,
  - i=3 (5 won coin): line 5's if-condition (5≤5) is true and, (C[5−5]+1 < C[5]) = (C[0]+1 <∞) = (0+1 <∞) is true, so 'C[j] = C[j−d_i]+1'. C[5] = C[5−5]+1 = C[0]+1 = 0+1 = 1. Therefore, C[5]=1.

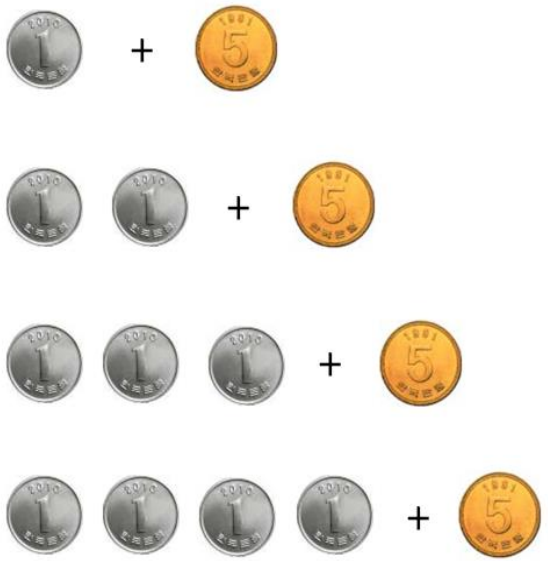
j	0	1	2	3	4	5
	0	1	2	3	4	∞

⇨

j	0	1	2	3	4	5
	0	1	2	3	4	1

  - i=4 (1 won coin): line 5's if-condition ($d_5 \leq 5$) is true, but (C[j−d_i]+1 < C[j]) = (C[5−1] +1 < C[4]) = (C[4] +1 < C[5]) = (4+1 < 1) = (5 < 1) is false, so C[5](=1) is unchanged.

- j=6, 7, 8, 9, and i=3 (5 won coin)
- C[6]=C[j-5]+1=C[6-5]+1=C[1]+1=1+1 = 2
- C[7]=C[j-5]+1=C[7-5]+1=C[2]+1=2+1 = 3
- C[8]=C[j-5]+1=C[8-5]+1=C[3]+1=3+1 = 4
- C[9]=C[j-5]+1=C[9-5]+1=C[4]+1=4+1 = 5
- When i=4 (1 won coin), line 5's if-condition $(C[j-d_i]+1 < C[j]) = (C[j-1]+1 < C[j])$ becomes $(1+1) < 2$, $(2+1) < 3$, $(3+1) < 4$, $(4+1) < 5$ for each j and they are all false, therefore C[j] is unchanged.



j	0	1	2	3	4	5	6	7	8	9
C	0	1	2	3	4	1	∞	∞	∞	∞
	0	1	2	3	4	1	**2**	∞	∞	∞
	0	1	2	3	4	1	2	**3**	∞	∞
	0	1	2	3	4	1	2	3	**4**	∞
	0	1	2	3	4	1	2	3	4	**5**

- j=10,
  - i=2 (10 won coin)
  - line 5: if-condition ($d_i \leq j$)=(10≤10) is true, (C[j-d_i]+1<C[j]) = (C[10-10]+1<C[10]) = (C[0]+1< C[10]) = (0+1<∞) is true, so 'C[j]=C[j-d_i]+1'. C[10] = C[10-10]+1 = C[0]+1 = 0+1 = 1. C[10]=1.

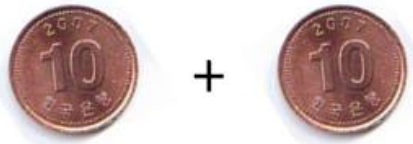
  - i=3 (5 won coin), line 5's if-condition ($d_i \leq j$) = (5<10) is true, but (C[10-5]+1<C[10]) = (C[5]+1<C[10]) = (1+1<1) is false, so C[10] is unchanged.

- i=4 (1 won coin) line 5's if-condition ($d_i \leq j$) = (1<10) is true, but (C[j-d_i]+1<C[j]) = (C[10-1]+1<C[10]) = (C[9]+1< C[10]) = (5+1<1) = (6<1) is false, so C[10](=1) is unchanged.

j	0	1	2	3	4	5	6	7	8	9	10
C	0	1	2	3	4	1	2	3	4	5	**1**

- j=20
  - i=1 (16 won coin): C[20] = C[j-16]+1 = C[4]+1 = 4 +1 = 5

  - i=2 (10 won coin): line 5's if-condition C[j-10]+1 = C[10]+1 = 1+1 = 2, which is smaller than C[20]=5. So if-condition is true and C[20]=2.



  - i=3 (5 won coin): line 5's if-condition (C[j-d_i]+1<C[j]) = (C[j-5]+1<C[j]) = (C[20-5]+1<C[20]) = (C[15]+1<C[20]) = (3<2) is false, C[20] not changed.

- i=4 (1 won coin): line 5's if-condition (C[20-1]+1<2) = (5<2) is false, so C[20] unchanged.

j	0	1	2	3	4	5	6	7	8	9	10
C	0	1	2	3	4	1	2	3	4	5	1

11	12	13	14	15	16	17	18	19	20
2	3	4	5	2	1	2	3	4	2

- Change 20 won:
  - With DP, final solution C[20]=2.
  - Greedy algorithm returns five from one 16 won coin and four 1 won coins.

- Greedy

DP

Example 8: Single Source Shortest Path Problem

- 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

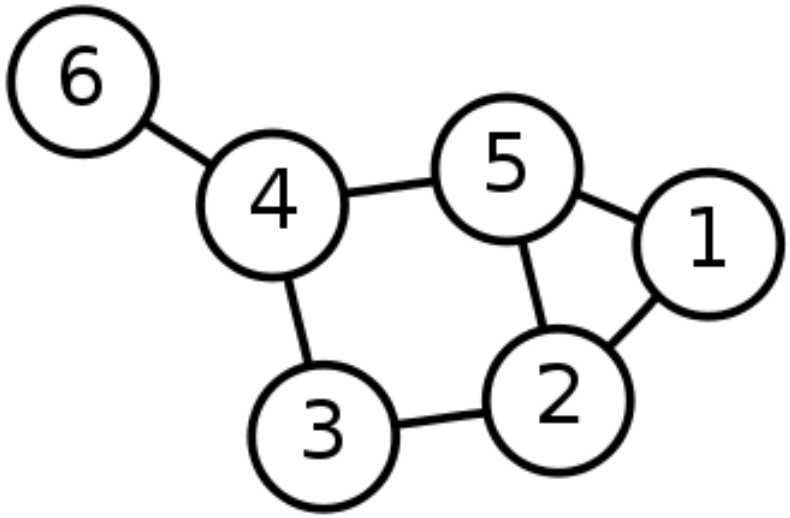
- d_t^k : Shortest path length from vertex s to vertex t through at most k edges
- Goal: d_t^{n-1}
- For $i \neq s$,
  - $d_t^0 = \infty$
  - $d_t^1 = w_{s,t}$

Recursive Relation

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

Single-Source Shortest Path Problem

Single-Source Shortest Path Problem - The problem of finding shortest paths from a source vertex *v* to all other vertices in the graph.



Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Approach: Greedy

Input: Weighted graph G={E,V} and source vertex $v \in V$, such that all edge weights are nonnegative

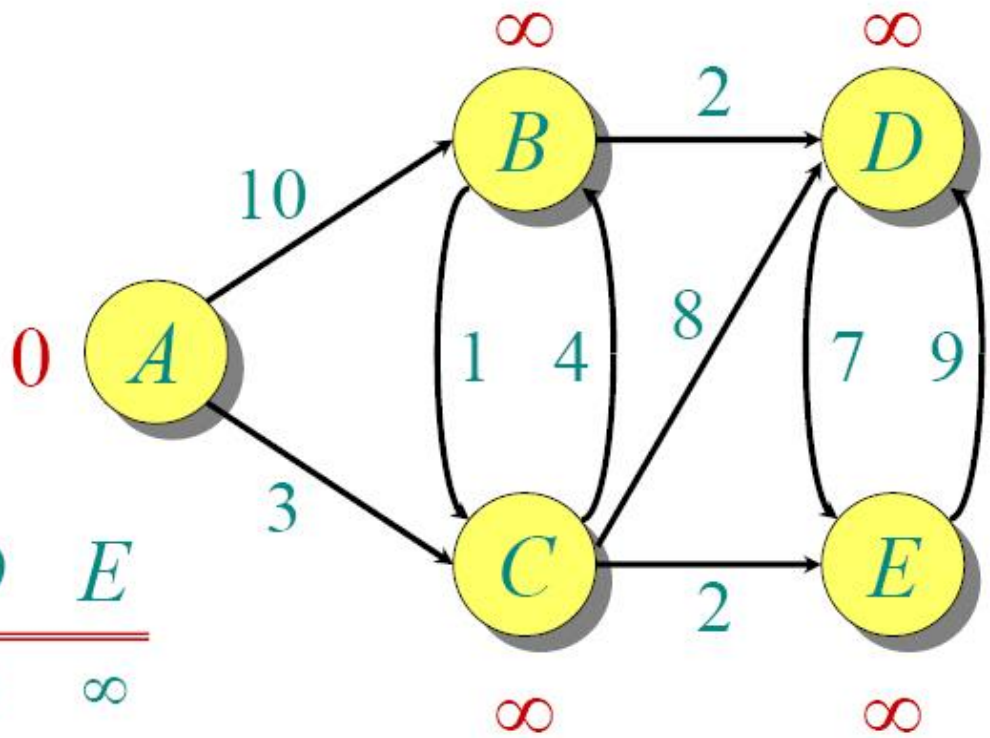
Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

Dijkstra's algorithm - Pseudocode

```
dist[s] ←o                          (distance to source vertex is zero)
for all v ∈ V–{s}
      do dist[v] ←∞                 (set all other distances to infinity)
S←∅                                 (S, the set of visited vertices is initially empty)
Q←V                                 (Q, the queue initially contains all
vertices)
while Q ≠∅                          (while the queue is not empty)
do  u ← mindistance(Q,dist)         (select the element of Q with the min. distance)
    S←S ∪{u}                        (add u to list of visited vertices)
     for all v ∈ neighbors[u]
          do if  dist[v] > dist[u] + w(u, v)          (if new shortest path found)
                then     d[v] ←d[u] + w(u, v)        (set new value of shortest path)
                     (if desired, add traceback code)
return dist
```

Dijkstra Animated Example

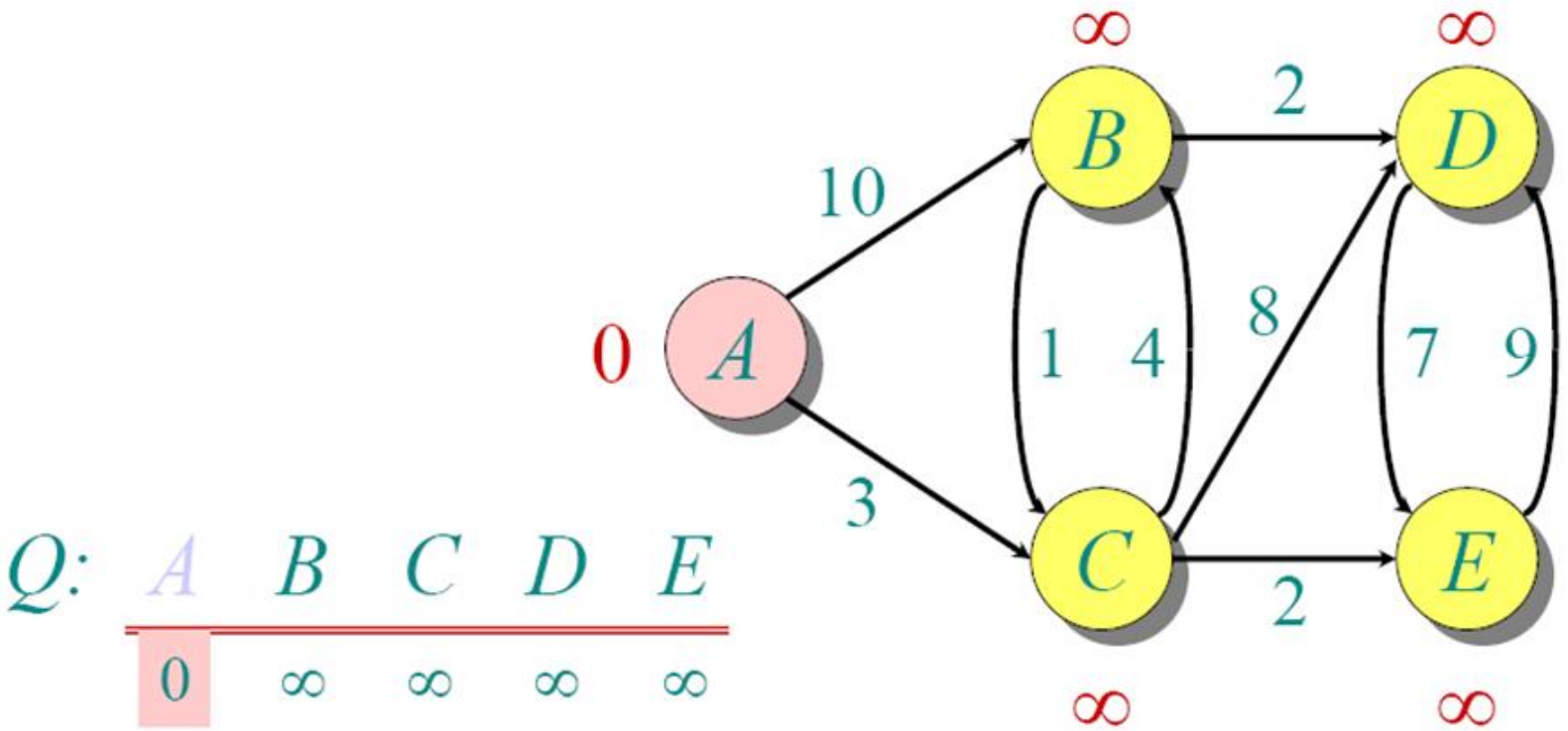
Initialize:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

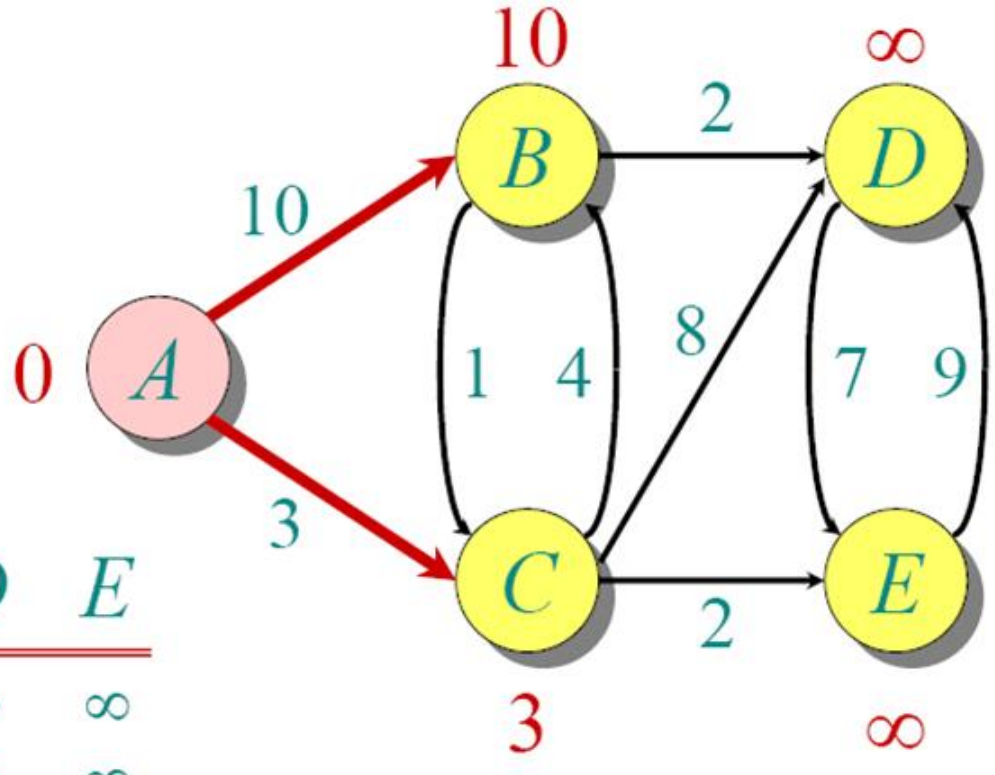
S: {}

Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

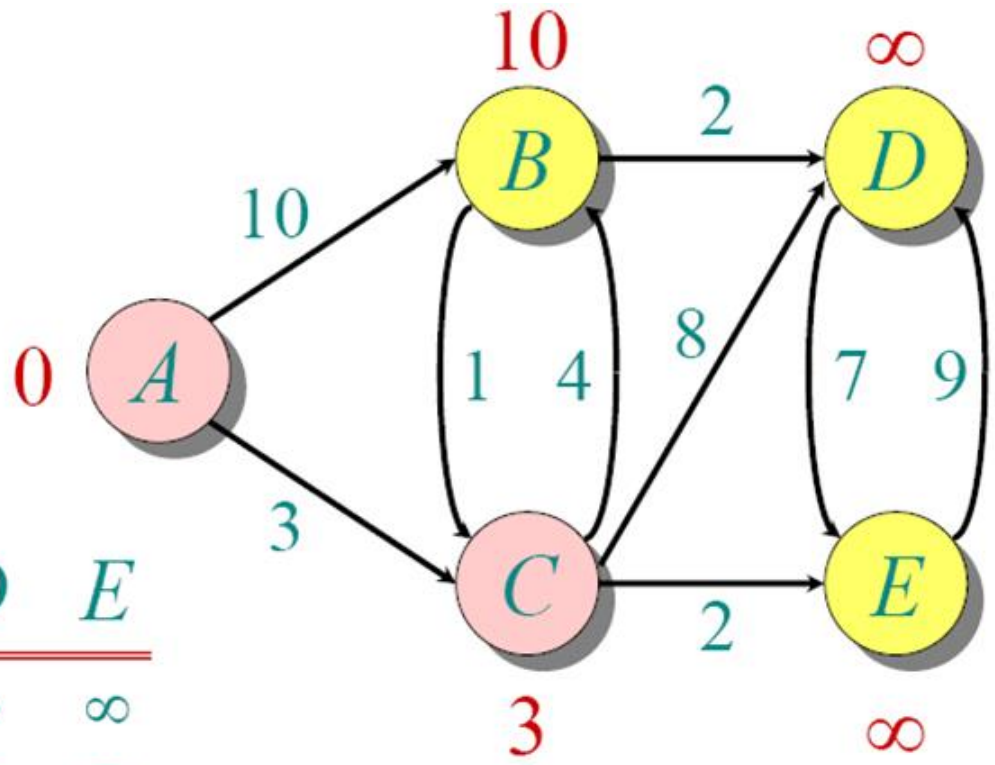
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

$S: \{ A \}$

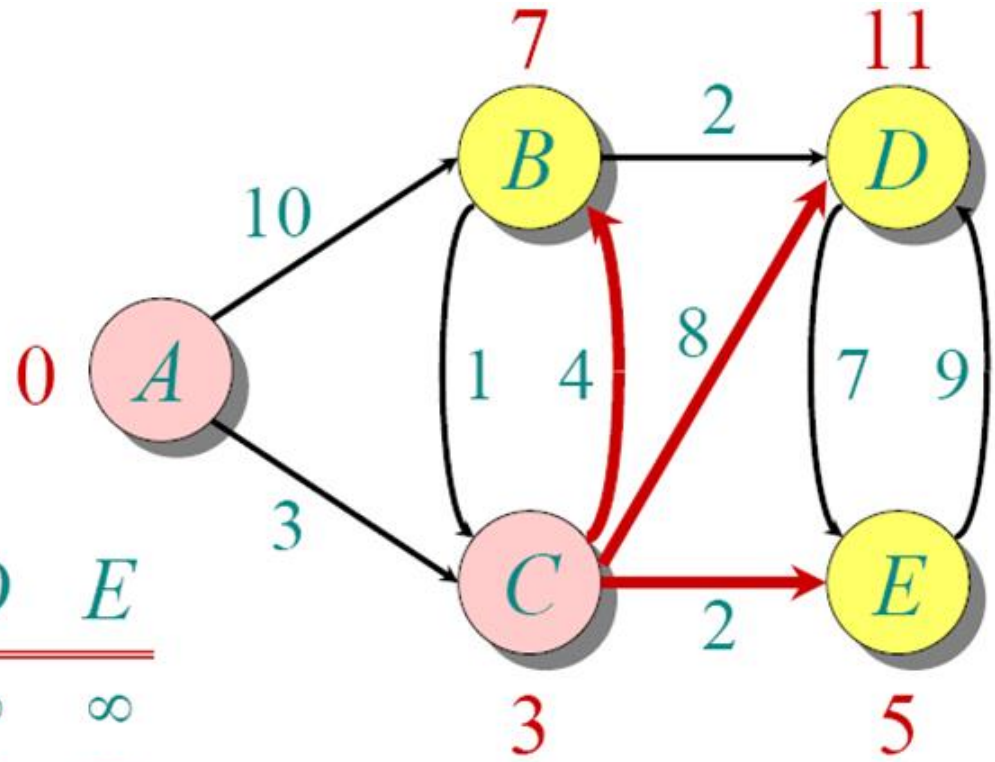
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

$S: \{ A, C \}$

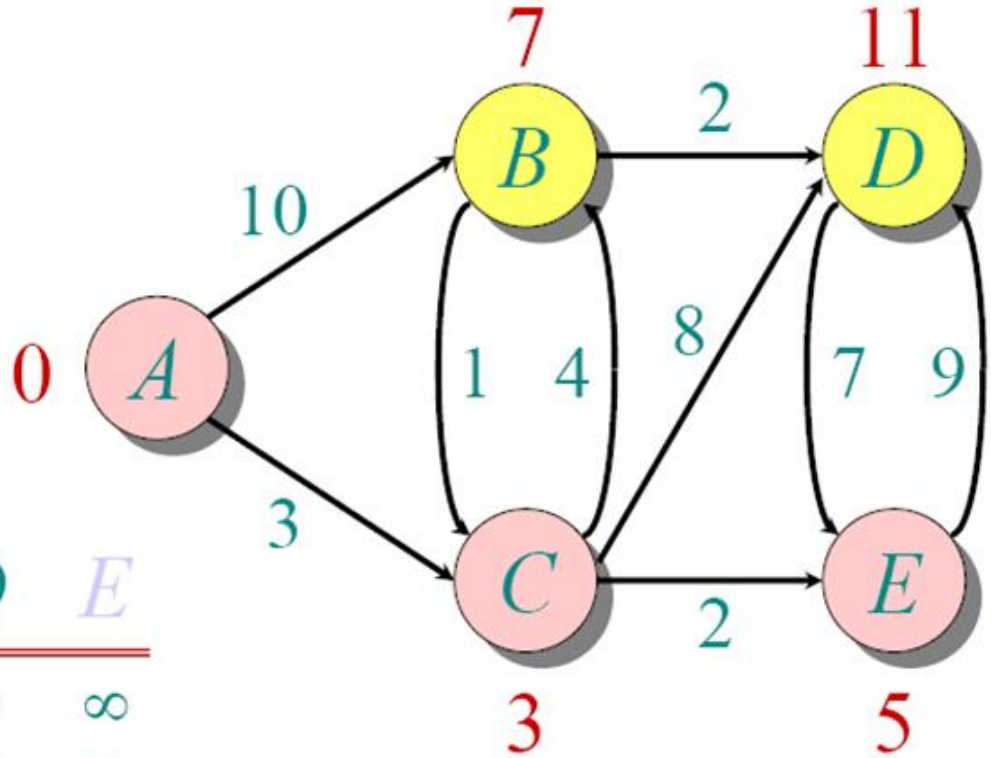
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

$S: \{ A, C \}$

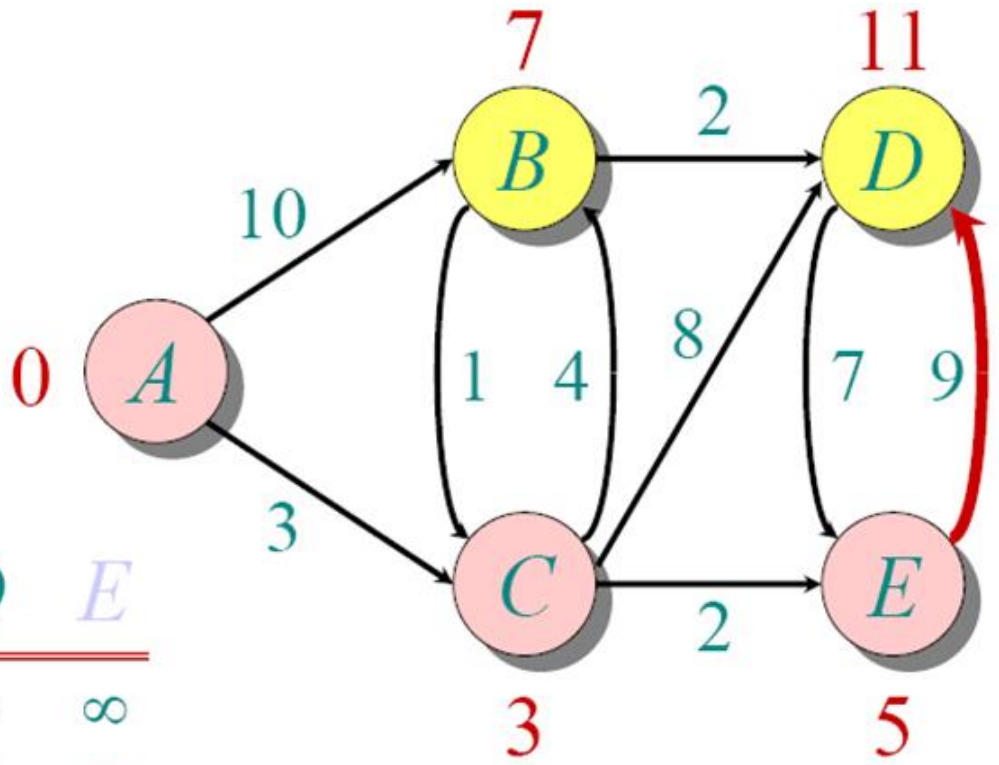
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

$S: \{ A, C, E \}$

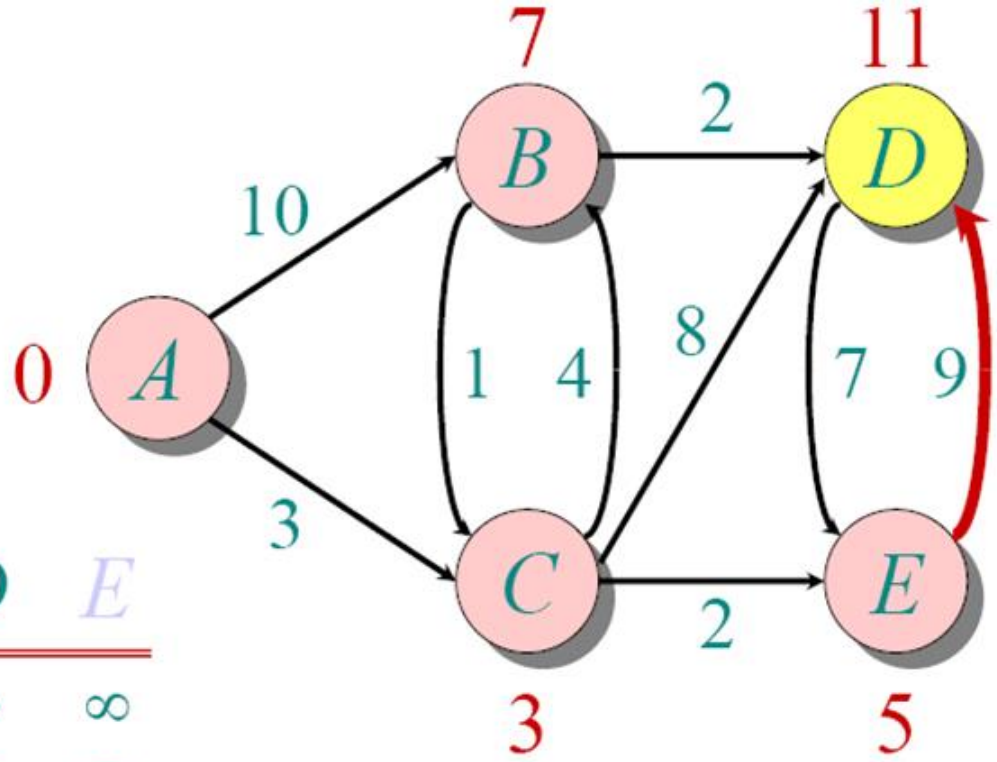
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

$S: \{ A, C, E \}$

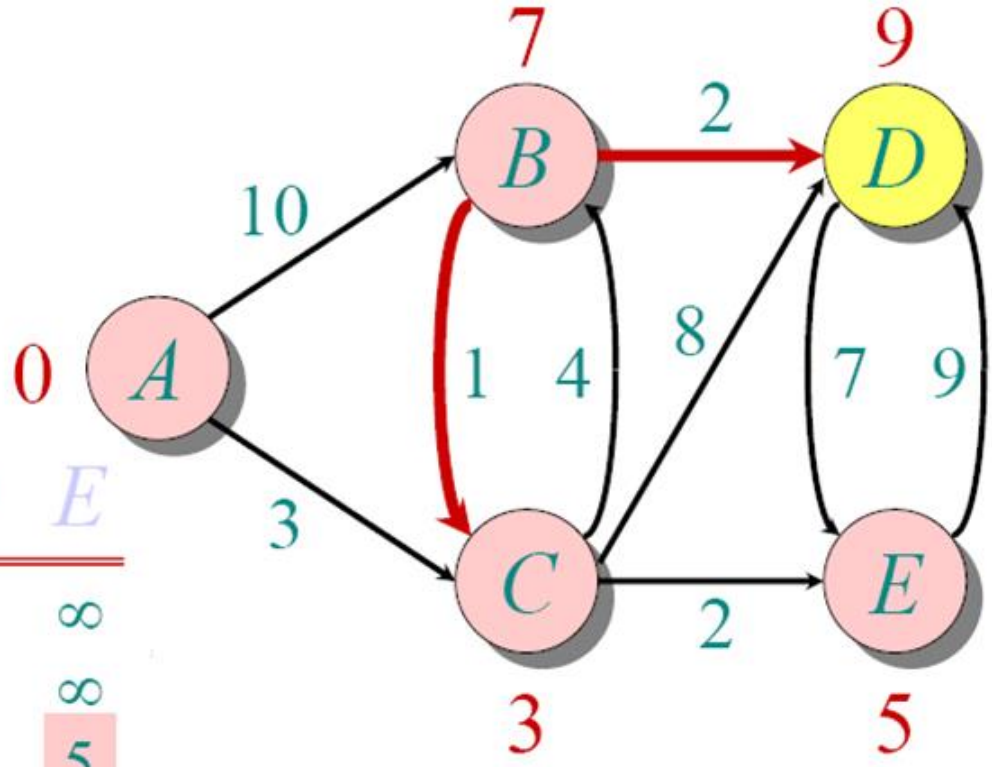
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

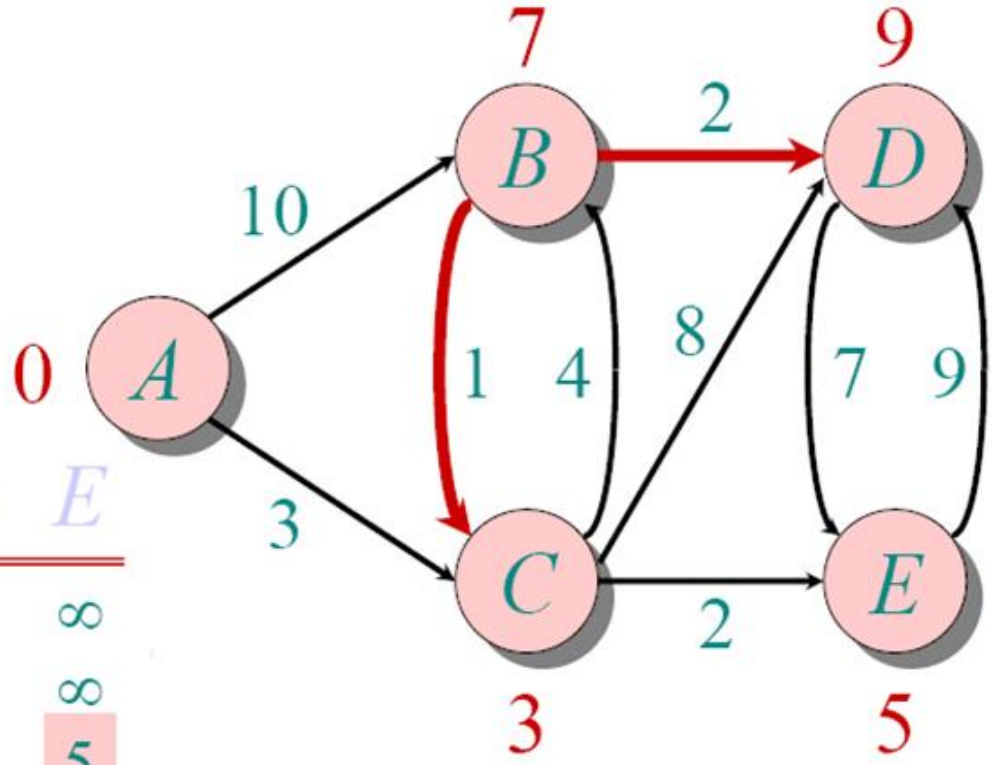
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B }

Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B, D }

IMPLEMENTATIONS AND RUNNING TIMES

The simplest implementation is to store vertices in an array or linked list. This will produce a running time of

$O(|V|^2 + |E|)$

For sparse graphs, or graphs with very few edges and many nodes, it can be implemented more efficiently storing the graph in an adjacency list using a binary heap or priority queue. This will produce a running time of

$O((|E|+|V|) \log |V|)$

Dijkstra's Algorithm - Why It Works

- As with all greedy algorithms, we need to make sure that it is a correct algorithm (e.g., it *always* returns the right solution if it is given correct input).

- A formal proof would take longer than this presentation, but we can understand how the argument works intuitively.

- If you can't sleep unless you see a proof, see the second reference or ask us where you can find it.

Dijkstra's Algorithm - Why It Works

- To understand how it works, we'll go over the previous example again. However, we need two mathematical results first:

- **Lemma 1**: Triangle inequality
  If $\delta(u,v)$ is the shortest path length between u and v,
  $\delta(u,v) \leq \delta(u,x) + \delta(x,v)$
- **Lemma 2**:
  The subpath of any shortest path is itself a shortest path.

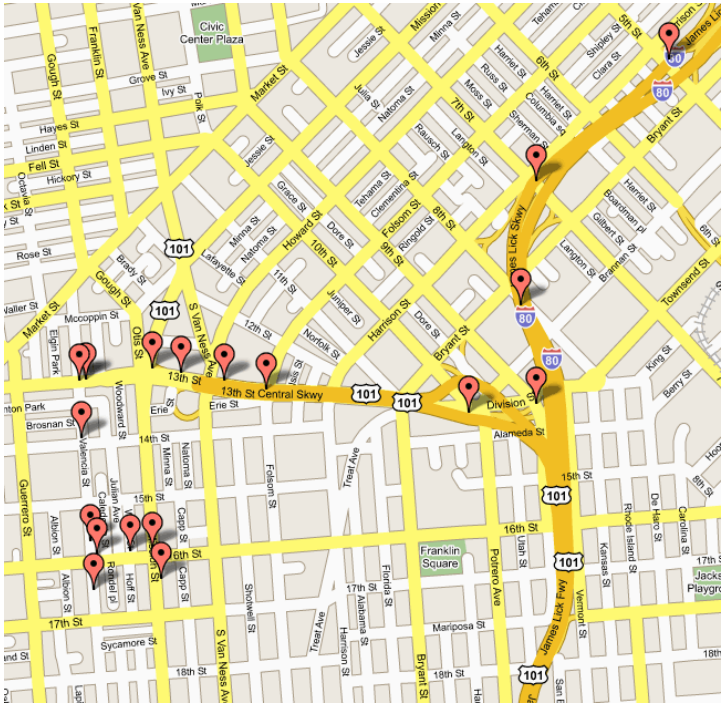
- The key is to understand why we can claim that anytime we put a new vertex in S, we can say that we already know the shortest path to it.
- Now, back to the example…

Dijkstra's Algorithm - Why use it?

- As mentioned, Dijkstra's algorithm calculates the shortest path to every vertex.
- However, it is about as computationally expensive to calculate the shortest path from vertex *u* to every vertex using Dijkstra's as it is to calculate the shortest path to some particular vertex *v*.
- Therefore, anytime we want to know the optimal path to some other vertex from a determined origin, we can use Dijkstra's algorithm.

Applications of Dijkstra's Algorithm

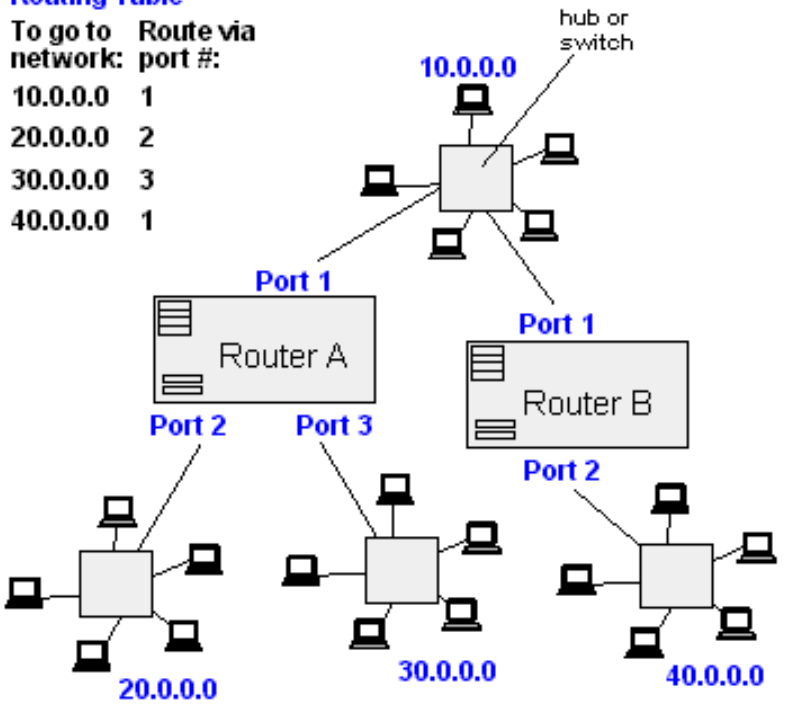
- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- Routing Systems



From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.

Router A
Routing Table

To go to network:	Route via port #:
10.0.0.0	1
20.0.0.0	2
30.0.0.0	3
40.0.0.0	1



All Pairs Shortest Paths

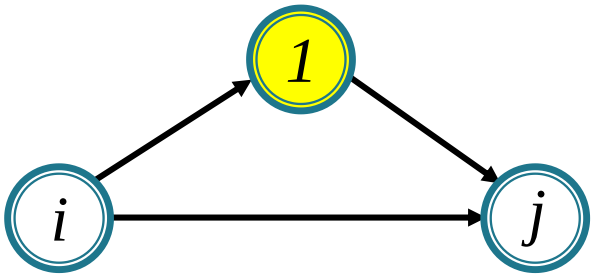
- To find shortest paths for each nodes.

	서울 Seoul	인천 Incheon	수원 Suwon	대전 Daejeon	전주 Jeonju	광주 Gwangju	대구 Daegu	울산 Ulsan	부산 Busan
서울 Seoul		40.2	41.3	154	232.1	320.4	297	407.5	432
인천 Incheon			54.5	174	253.3	351.6	317.6	447	453
수원 Suwon				132.6	189.4	299.6	268.1	356	390.7
대전 Daejeon					96.9	185,2	148.7	259.1	283.4
전주 Jeonju						105.9	219.7	331.1	322.9
광주 Gwangju							219.3	329.9	268
대구 Daegu								111.1	135.5
울산 Ulsan									52.9
부산 Busan									

- How about running Dijkstra for each nodes?
- Time complexity using array is $(n-1)xO(n^2) = O(n^3)$.
- Warshall proposed DP that finds transitive closure of all pairs.
- Floyd proposed All-Pairs Shortest Paths algorithm using Warshall's finding.
- Floyd-Warshall algorithm finds All-Pairs Shortest Paths.

- Time complexity of Floyd-Warshall algorithm is $O(n^3)$ which is the same as (n-1)xDijkstra.
- Floyd-Warshall is much simpler and more efficient than Dijkstra.

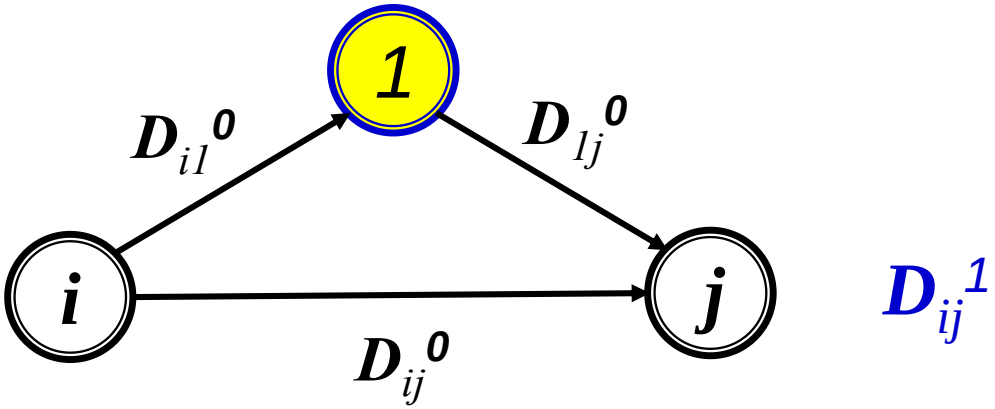
- We find subproblems first.
- When there are 3 points, to find the shortest path from i to j, we choose shorter distance between i–j and i–1–j.



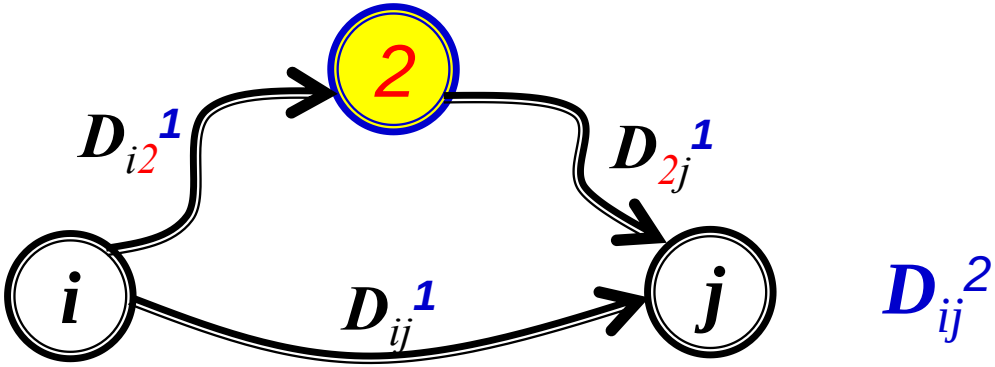
- Another important idea : transitive closure
  - From point 1 → points 1,2 → points1, 2, 3…
  - Add points one by one.
  - Finally, consider all points 1~n and calculate the shortest paths for all pair.
- Subproblem: points – 1, 2, 3, …, n

  D_{ij}^{k} = The shortest distance from i to j, when we consider points {1, 2, …, k} as traversable points.
- Note that D_{ij}^{k} can be the shortest path without passing any points from 1 to k.
- If k≠i, k≠j, k=0, since there is no point 0 in the graph, it means we don't pass any other points. D_{ij}^{0} is the weight of edge (i,j).

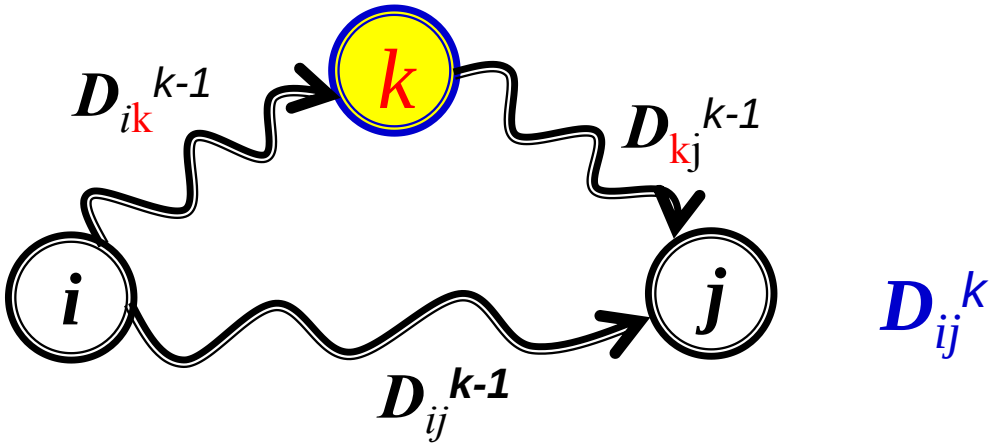
- D_{ij}^{1} is the shortest path from i to j using the point 1.
- So, D_{ij}^{1} for all pairs is the smallest subproblems. i≠1, j≠1.



- Then we calculate D_{ij}^{2}, the shortest path from i to j using the points 1 and 2. The distance passing the point 2 is $D_{i2}^{1} + D_{2j}^{1}$.
- So the next smallest problem is to calculate D_{ij}^{2}. i≠2, j≠2.



- D_{ij}^{2}, the shortest path from i to j using the points 1, 2, …, k. The distance passing the poitn k is $D_{ik}^{k-1} + D_{kj}^{k-1}$, i≠k, j≠k.
- We iterate this until k is from 1 to n.



All-Pairs Shortest Paths

AllPairsShortest

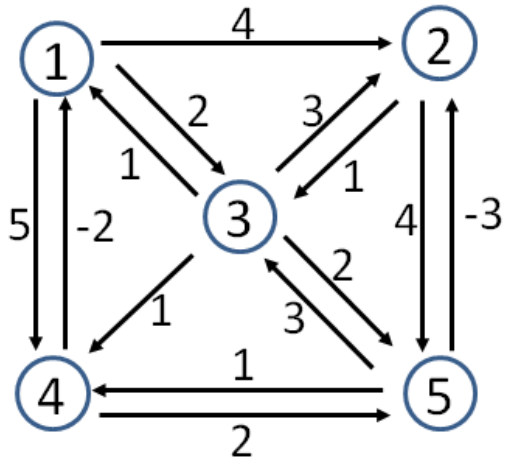
Input: 2D matrix D, D[i,j]=weight of (i,j), If there is no edge (i,j), D[i,j]=∞, For all i, D[i,i]=0.

Output: Distance of all-pairs shortest paths in D

```
1. for k = 1 to n
2.     for i = 1 to n (i≠k)
3.         for j = 1 to n (j≠k, j≠i)
4.             D[i,j] = min{D[i,k]+D[k,j], D[i,j]}
```

- Note that there should be no negative cycle in this algorithm.
- Because if there are negative cycles, then each iteration will diminish the distance.

How AllPairsShortest algorithm works?

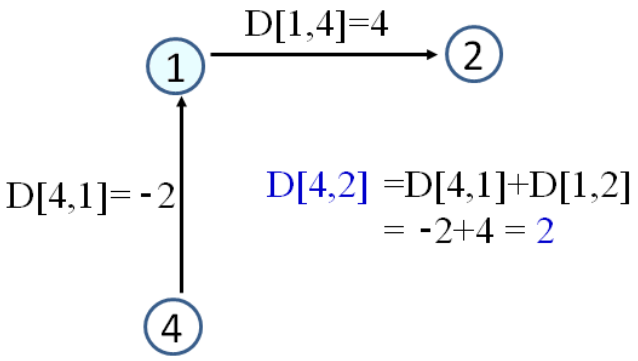


D	1	2	3	4	5
1	0	4	2	5	∞
2	∞	0	1	∞	4
3	1	3	0	1	2
4	-2	∞	∞	0	2
5	∞	-3	3	1	0

- We see how D's elements change from k=1 to k=5.

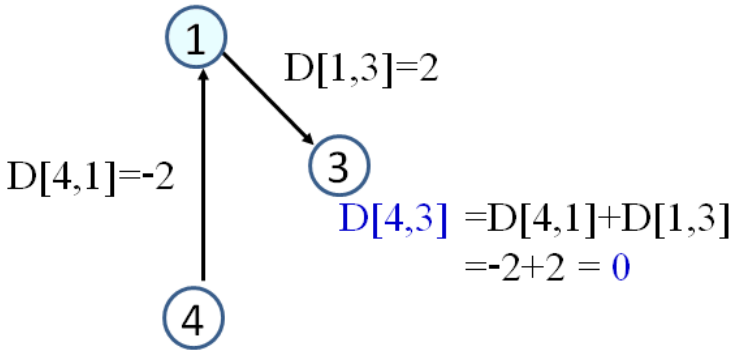
- k=1:
  - D[2,3] = min{D[2,3], D[2,1]+D[1,3]} = min{1, ∞+2} = 1
  - D[2,4] = min{D[2,4], D[2,1]+D[1,4]} = min{∞, ∞+5} = ∞
  - D[2,5] = min{D[2,5], D[2,1]+D[1,5]} = min{4, ∞+∞} = 4
  - D[3,2] = min{D[3,2], D[3,1]+D[1,2]} = min{3, 1+4} = 3
  - D[3,4] = min{D[3,4], D[3,1]+D[1,4]} = min{1, 1+5} = 1
  - D[3,5] = min{D[3,5], D[3,1]+D[1,5]} = min{2, 1+∞} = 2
  - D[4,2] = min{D[4,2], D[4,1]+D[1,2]} = min{∞, −2+4} = 2

// updated



- D[4,3] = min{D[4,3], D[4,1]+D[1,3]} = min{∞, -2+2} = 0

// updated



- D[4,5] = min{D[4,5], D[4,1]+D[1,5]} = min{2, -2+∞} = 2
- D[5,2] = min{D[5,2], D[5,1]+D[1,2]} = min{-3, ∞+4} = -3
- D[5,3] = min{D[5,3], D[5,1]+D[1,3]} = min{3, ∞+2} = 3
- D[5,4] = min{D[5,4], D[5,1]+D[1,4]} = min{1, ∞+5} = 1

- k=1: D[4,2] updated to 2, D[4,3] updated to 0. Others unchanged.

D	1	2	3	4	5
1	0	4	2	5	∞
2	∞	0	1	∞	4
3	1	3	0	1	2
4	-2	**2**	**0**	0	2
5	∞	-3	3	1	0

- k=2:
  - D[1,5] : 1 → 2 → 5, distance 8
  - D[5,3]: 5 → 2 → 3, distance −2

D	1	2	3	4	5
1	0	4	2	5	**8**
2	∞	0	1	∞	4
3	1	3	0	1	2
4	-2	2	0	0	2
5	∞	-3	**-2**	1	0

- k=3: 7 cells updated

D	1	2	3	4	5
1	0	4	2	**3**	**4**
2	**2**	0	1	**2**	**3**
3	1	3	0	1	2
4	-2	2	0	0	2
5	**-1**	-3	-2	**-1**	0

- k=4: 3 elements updated.

D	1	2	3	4	5
1	0	4	2	3	4
2	**0**	0	1	2	3
3	**-1**	3	0	1	2
4	-2	2	0	0	2
5	**-3**	-3	-2	-1	0

- k=5: 3 elements updated.

D	1	2	3	4	5
1	0	**1**	2	3	4
2	0	0	1	2	3
3	-1	**-1**	0	1	2
4	-2	**-1**	0	0	2
5	-3	-3	-2	-1	0

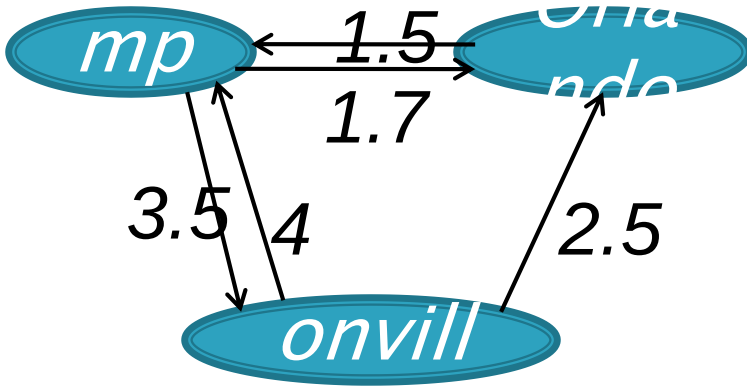
Time Complexity

- For each k, all (i,j) pairs are considered, so, nxnxn = n^3 calculations and each calculation takes O(1).
- Time complexity of AllPairsShortest is $O(n^3)$.

Floyd-Warshall Algorithm

- A weighted, directed graph is a collection vertices connected by weighted edges (where the weight is some real number).
  - One of the most common examples of a graph in the real world is a road map.
    - Each location is a vertex and each road connecting locations is an edge.
    - We can think of the distance traveled on a road from one location to another as the weight of that edge.

	Tampa	Orlando	Jaxville
Tampa	0	1.7	3.5
Orlando	1.5	0	∞
Jax	4	2.5	0

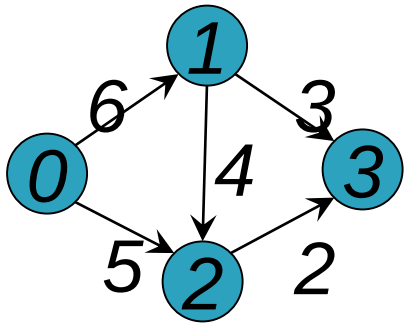


Storing a Weighted, Directed Graph

- Adjacency Matrix:
  - Let **D** be an edge-weighted graph in adjacency-matrix form
  - **D(*i*,*j*)** is the weight of edge (i, j), or ∞ if there is no such edge.
- Update matrix **D**, with the shortest path *through immediate vertices.*

D =

	0	1	2	3
0	0	6	5	∞
1	∞	0	4	3
2	∞	∞	0	2
3	∞	∞	∞	0



Floyd-Warshall Algorithm

- Given a weighted graph, we want to know the shortest path from one vertex in the graph to another.
  - The Floyd-Warshall algorithm determines the shortest path between all pairs of vertices in a graph.
  - What is the difference between Floyd-Warshall and Dijkstra's??

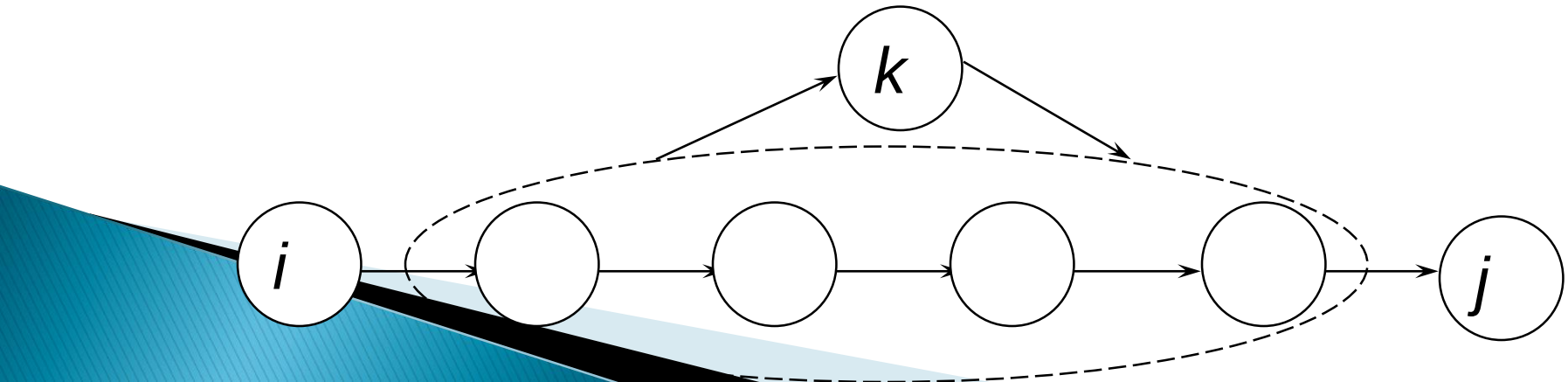
Floyd-Warshall Algorithm

- If V is the number of vertices, Dijkstra's runs in $\Theta(V^2)$
  - We could just call Dijkstra |V| times, passing a different source vertex each time.
  - $\Theta(V \times V^2) = \Theta(V^3)$
  - (Which is the same runtime as the Floyd-Warshall Algorithm)

- *BUT*, Dijkstra's doesn't work with negative-weight edges.

Floyd Warshall Algorithm

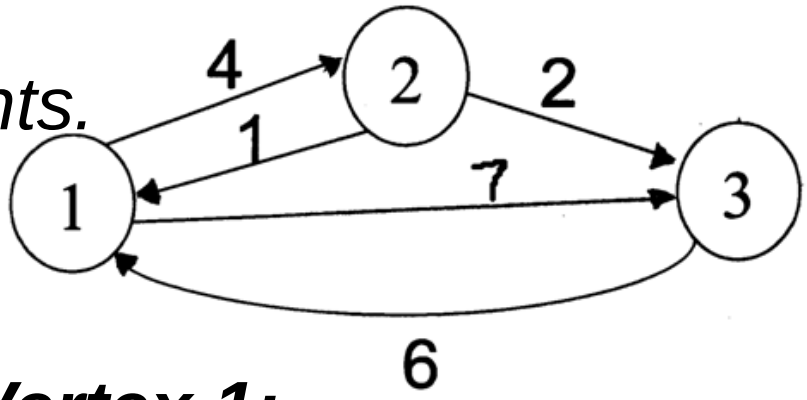
- Let's go over the premise of how Floyd-Warshall algorithm works…
  - Let the vertices in a graph be numbered from 1 … n.
  - Consider the subset {1,2,…, k} of these n vertices.
  - Imagine finding the shortest path from vertex i to vertex j that uses vertices in the set {1,2,…,k} only.
  - There are two situations:
    1) k is an intermediate vertex on the shortest path.
    2) k is not an intermediate vertex on the shortest path.



Floyd Warshall Algorithm – Example

$$D^{(0)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & \infty & 0 \end{bmatrix}$$

Original weights.



$$D^{(1)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 1:

D(3,2) = D(3,1) + D(1,2)

$$D^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 2:

D(1,3) = D(1,2) + D(2,3)

$$D^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 3:

Nothing changes.

Floyd Warshall Algorithm

- Looking at this example, we can come up with the following algorithm:
  - Let D store the matrix with the initial graph edge information initially, and update D with the calculated shortest paths.

```
For k=1 to n {
  For i=1 to n {
      For j=1 to n
        D[i,j] = min(D[i,j],D[i,k]+D[k,j])
  }
}
```

  - The final D matrix will store all the shortest paths.

Floyd Warshall – Path Reconstruction

- The path matrix will store the last vertex visited on the path from i to j.
  - So path[i][j] = k means that in the shortest path from vertex i to vertex j, the LAST vertex on that path before you get to vertex j is k.

- Based on this definition, we must initialize the path matrix as follows:
  - path[i][j] = i if i!=j and there exists an edge from i to j
  - = NIL otherwise

- The reasoning is as follows:
  - If you want to reconstruct the path at this point of the algorithm when you aren't allowed to visit intermediate vertices, the previous vertex visited MUST be the source vertex i.
  - NIL is used to indicate the absence of a path.

Floyd Warshall – Path Reconstruction

- Before you run Floyd's, you initialize your distance matrix ***D*** and path matrix ***P*** to indicate the use of no immediate vertices.
  - (Thus, you are only allowed to traverse direct paths between vertices.)

- Then, at each step of Floyd's, you essentially find out whether or not using vertex k will *improve* an estimate between the distances between vertex i and vertex j.

- If it ***does improve*** the estimate here's what you need to record:

1) record the new shortest path weight between i and j
2) record the fact that the shortest path between i and j goes through k

Floyd Warshall – Path Reconstruction

- If it ***does improve*** the estimate here's what you need to record:
  1) record the new shortest path weight between i and j
     - **We don't need to change our path and we do not update the path matrix**
  2) record the fact that the shortest path between i and j goes through k
     - **We want to store the last vertex from the shortest path from vertex k to vertex j. *This will NOT necessarily be k, but rather, it will be path[k][j].***

This gives us the following update to our algorithm:

```
if (D[i][k]+D[k][j] < D[i][j]) { // Update is necessary to use k as intermediate vertex
    D[i][j] = D[i][k]+D[k][j];
    path[i][j] = path[k][j];
}
```

Path Reconstruction

- Now, the once this path matrix is computed, we have all the information necessary to reconstruct the path.
  - Consider the following path matrix (indexed from 1 to 5 instead of 0 to 4):

NIL	**3**	**4**	**5**	**1**
4	**NIL**	**4**	**2**	**1**
4	**3**	**NIL**	**2**	**1**
4	**3**	**4**	**NIL**	**1**
4	**3**	**4**	**5**	**NIL**

- Reconstruct the path from vertex1 to vertex 2:
  - First look at path[1][2] = 3. This signifies that on the path from 1 to 2, 3 is the last vertex visited before 2.
    - Thus, the path is now, 1…3->2.
  - Now, look at path[1][3], this stores a 4. Thus, we find the last vertex visited on the path from 1 to 3 is 4.
  - So, our path now looks like 1…4->3->2. So, we must now look at path[1][4]. This stores a 5,
  - thus, we know our path is 1…5->4->3->2. When we finally look at path[1][5], we find 1,
  - which means our path really is 1->5->4->3->2.

Transitive Closure

- Computing a transitive closure of a graph gives you complete information about which vertices are connected to which other vertices.

- Input:
  - Un-weighted graph G: $W[i][j] = 1$, if $(i,j) \in E$, $W[i][j] = 0$ otherwise.
- Output:
  - $T[i][j] = 1$, if there is a path from i to j in G, $T[i][j] = 0$ otherwise.
- Algorithm:
  - Just run Floyd-Warshall with weights 1, and make $T[i][j] = 1$, whenever $D[i][j] < \infty$.
  - More efficient: use only Boolean operators

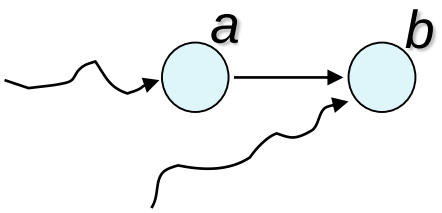
Transitive Closure

```
Transitive-Closure(W[1..n][1..n])
01 T ← W      // T(0)
02 for k ←1 to n do // compute T(k)
03     for i ←1 to n do
04         for j ←1 to n do
05             T[i][j] ← T[i][j] ∨ (T[i][k] ∧ T[k][j])
06 return T
```

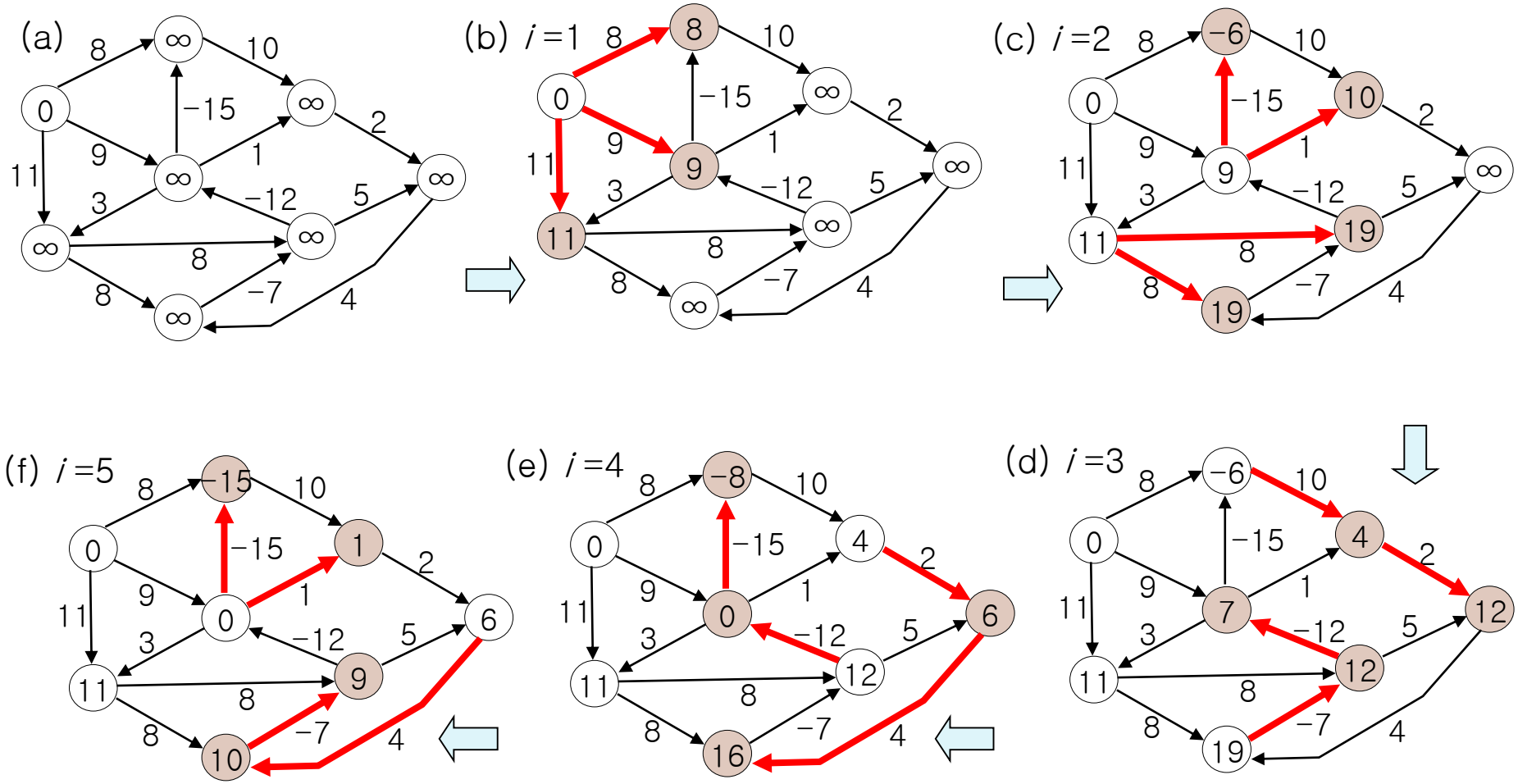
- This is the SAME as the other Floyd-Warshall Algorithm, except for when we find a non-infinity estimate, we simply add an edge to the transitive closure graph.
- Every round we build off the previous paths reached.
  - After iterating through all vertices being intermediate vertices, we have tried to connect all pairs of vertices i and j through all intermediate vertices k.

Bellman Ford Algorithm

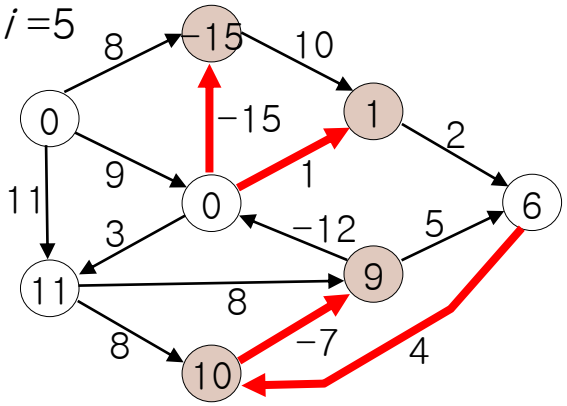
Bellman-Ford(*G, s*)
{
    $d_s \leftarrow 0$;
    for all vertices $i \neq s$
        $d_i \leftarrow \infty$;
    for $k \leftarrow 1$ **to** n-1 {
        for all edges (a, b) {
            if ($d_a + w_{a,b} < d_b$) **then** $d_b \leftarrow d_a + w_{a,b}$;
        }
    }
}



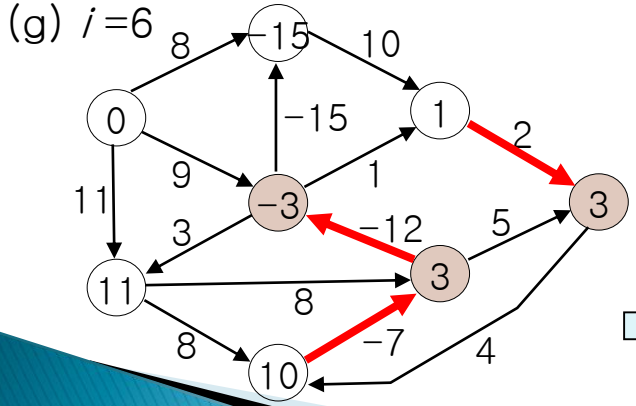
✓ Think of “propagation”!


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

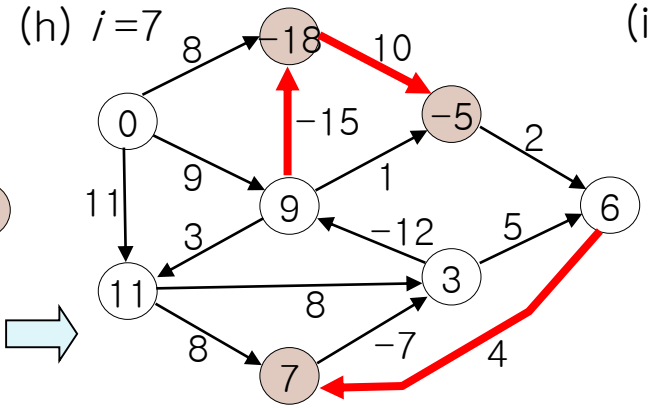
(f) $i=5$


−15
8
10
0
1
−15
2
9
1
11
0
6
3
−12
5
11
9
8
8
−7
4
10

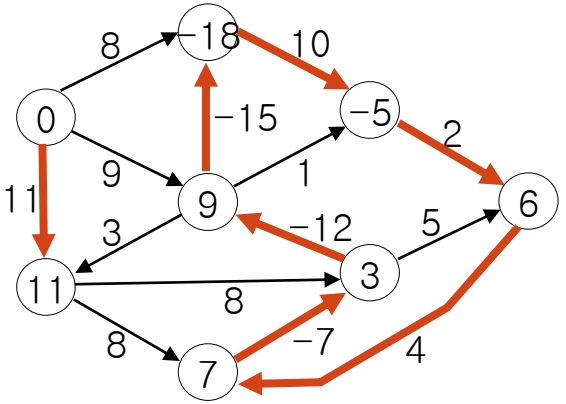
(g) $i=6$


8
−15
10
0
1
−15
2
9
1
11
−3
3
3
−12
5
11
3
8
8
−7
4
10

(h) $i=7$


8
−18
10
0
−15
−5
2
9
1
11
9
6
3
−12
5
11
3
8
8
−7
4
7

(i)


8
−18
10
0
−15
−5
2
9
1
11
9
6
3
−12
5
11
3
8
8
−7
4
7

Applications

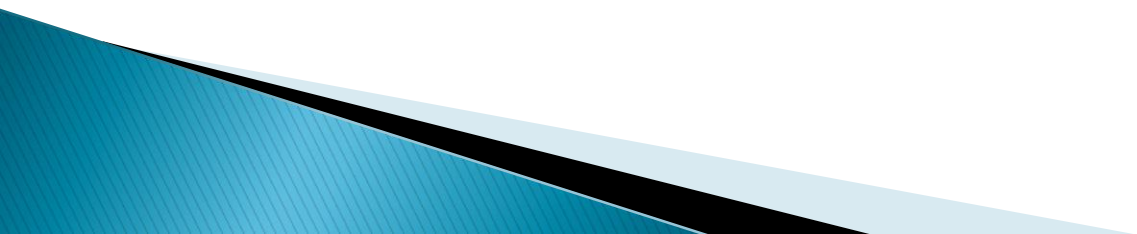
- Map service of Mapquest and Google
- Car navigator service
- Network analysis of GIS
- Communication network and mobile communication
- Game
- Operations Research
- Robotics
- Traffic analysis
- VLSI Design

Example 9: Knapsack Problem

- Knapsack problem tries to maximize the summed values of all items in a knapsack
  - n items,
  - w_i weights of items i
  - v_i values of items i
  - C capacity of knapsack

  - Maximize the value of items in the knapsack
  - Summed weights of items in the knapsack < C
  - The number of each item is 1.

- Can be solved by DP for limited input.
- Items, weights, values, and capacity
- We use items and weights for subproblem definition.
- Starting from empty knapsack, we decide whether to put an item or not from the sum of values in the knapsack.
- We also need to examine the capacity overflow.



- Subproblem can be defined as follows:
- K[i,w] = for item 1~i, max value of knapsack with weight w , where i = 1, 2, …, n, and w = 1, 2, 3, …, C.
- Final solution is K[n,C]
- We increment the weight from 1 to C.
- If C is too big, the running time is too slow.
- So the following algorithm is efficient for limited inputs.

Algorithm

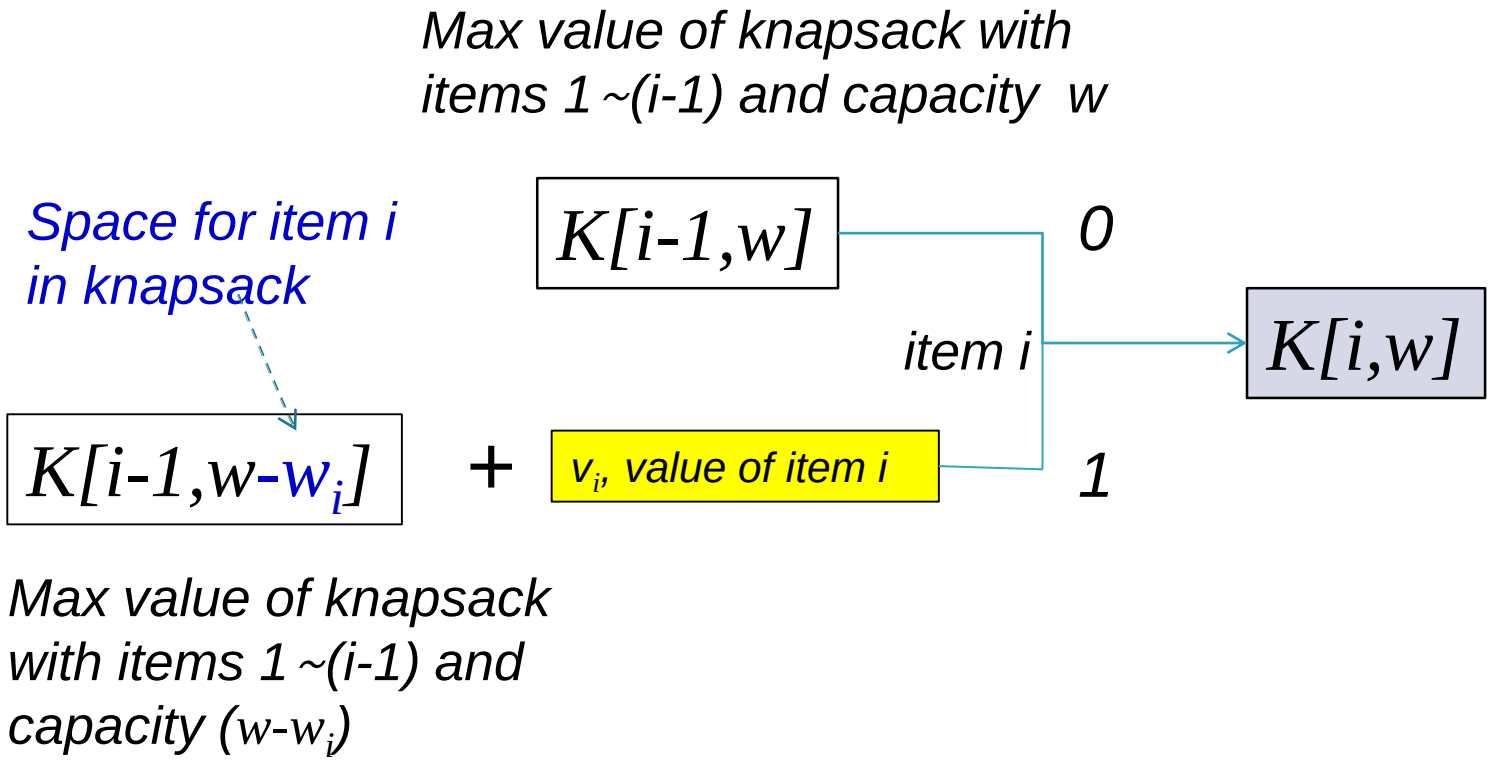
Knapsack

Input: capacity C, n items, weight of item i=w_i, value of item i=v_i,
where, i = 1, 2, ···, n

Output: K[n,C]

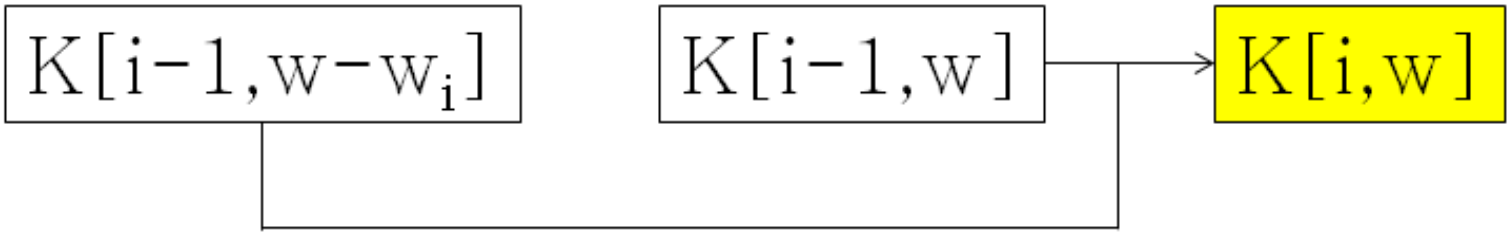
```
1. for i = 0 to n   K[i,0]=0      // C=0
2. for w = 0 to C  K[0,w]=0 // no items inside
3. for i = 1 to n {
4.      for w = 1 to C { // weights
5.              if ( w_i > w ) // if weight of i is bigger than w
6.                      K[i,w] = K[i-1,w]
7.              else // whether item i is inside or not
8.                      K[i,w] = max{K[i-1,w], K[i-1,w-w_i]+v_i}
        }
    }
9. return K[n,C]
```

- Line 3~8: Consider item from 1 to n and capacity from 1 to C.
- Line 5~6: if the weight of current item i is bigger than the capacity of knapsack, we cannot store the item i. Therefore the max value K[i,w] is K[i-1,w].
- Line 7~8: if we can store i inside, but we cannot add the item i because it will make the weight of knapsack ($w+w_i$), bigger than current capacity w.


Max value of knapsack with items 1 ~(i-1) and capacity w
Space for item i in knapsack
K[i-1,w]
0
item i
K[i,w]
K[i-1,w-w_i]
+
v_i, value of item i
1
Max value of knapsack with items 1 ~(i-1) and capacity (w-w_i)

- To store item i, we need to consider two cases as the previous slide.
- Item i not inside: K[i,w] = K[i-1,w].
- Item i inside: We subtract the weight of i, w_i from current weight w. And we add v_i the value of item i to K[i-1,w-w_i], the maximum value when we consider (i-1) items with w-w_i weight, which will become K[i,w].
- Line 8: we choose the maximum of the two above cases.

- Implicit order of knapsack problem.
  - Two subsolutions $K[i-1,w-w_i]$ and $K[i-1,w]$ should be calculated for $K[i,w]$.



Process of Knapsack Algorithm

- C=10kg, weight and value of each item shown below.

Item	1	2	3	4
Weight	5	4	6	3
Value	10	40	30	50

- Line 1~2: we initialize 0^{th} row and 0^{th} column with 0.

C=10

Knapsack Capacity → w=			0	1	2	3	4	5	6	7	8	9	10
Weight	Value	Item	0	0	0	0	0	0	0	0	0	0	0
5	10	1	0										
4	40	2	0										
6	30	3	0										
3	50	4	0										

- Line 3: item # changes from 1~4.
- Line 4: the capacity of knapsack w increase by 1kg until it becomes 10kg.

- i=1: we consider item 1 only.
- w=1: knapsack capacity is 1kg. Since $w_1 > w$ (1>5), K[1,1] = K[i-1,w] = K[1-1,1] = K[0,1] = 0.

- w=2, 3, 4: w_1>w, so K[2,1]=0, K[1,3]=0, K[1,4]=0. Obviously, we cannot store 5kg item inside 4kg knapsack.

- w=5: w_1=w, so we can store item 1.

$$K[1,5] = \max\{K[i-1,w], K[i-1,w-w_i]+v_i\}$$
$$= \max\{K[1-1,5], K[1-1,5-5]+10\}$$
$$= \max\{K[0,5], K[0,0]+10\}$$
$$= \max\{0, 0+10\}$$
$$= \max\{0, 10\} = 10.$$

- w=6, 7, 8, 9, 10: as w=5, we can store item 1. K[1,6] = K[1,7] = K[1,8] = K[1,9] = K[1,10] = 10.

- Results until now.

C=10

Knapsack Capacity → w=			0	1	2	3	4	5	6	7	8	9	10
Weight	Value	Item	0	0	0	0	0	0	0	0	0	0	0
5	10	i = 1	0	**0**	**0**	**0**	**0**	**10**	**10**	**10**	**10**	**10**	**10**
4	40	2	0										
6	30	3	0										
3	50	4	0										

- i=2: we use subsolutions for i=1.
- w=1, 2, 3: $w_2 > w$, (w_2=4kg) , so K[2,1]=0, K[2,2]=0, K[2,3]=0.

- w=4: we store item 2.

$$K[2,4] = \max\{K[i-1,w], K[i-1,w-w_i]+v_i\}$$
$$= \max\{K[2-1,4], K[2-1,4-4]+40\}$$
$$= \max\{K[1,4], K[1,0]+40\}$$
$$= \max\{0, 0+40\}$$
$$= \max\{0, 40\} = 40$$

- w=5: we can store item 2 (4kg). However, we compare the case item 1 is stored and the case item 2 is stored, and choose the item with bigger value.

$$K[2,5] = \max\{K[i-1,w], K[i-1,w-w_i]+v_i\}$$
$$= \max\{K[2-1,5], K[2-1,5-4]+40\}$$
$$= \max\{K[1,5], K[1,1]+40\}$$
$$= \max\{10, 0+40\}$$
$$= \max\{10, 40\} = 40$$

We remove item 1 and store item 2.
Resulting value is 40.

- w=6, 7, 8: it is more valuable to remove item 1 and store item 2. K[2,6] = K[2,7] = K[2,8] = 40.

- w=9: w_2<w, we can store item 2 into knapsack.

$$K[2,9] = \max\{K[i-1,w], K[i-1,w-w_i]+v_i\}$$
$$= \max\{K[2-1,9], K[2-1,9-4]+40\}$$
$$= \max\{K[1,9], K[1,5]+40\}$$
$$= \max\{10, 10+40\}$$
$$= \max\{10, 50\} = 50.$$

- We can store item 1 and 2 into the knapsack and the resulting value is 50.

- w=10, w_2<w, so as in w=9, K[2,10]=50= value of 1 and 2.

- Value of 1 and 2, Capacity=1..C

C=10

Knapsack Capacity → W=			0	1	2	3	4	5	6	7	8	9	10
Weight	Value	Item	0	0	0	0	0	0	0	0	0	0	0
5	10	1	0	0	0	0	0	10	10	10	10	10	10
4	40	i = 2	0	**0**	**0**	**0**	**40**	**40**	**40**	**40**	**40**	**50**	**50**
6	30	3	0										
3	50	4	0										

- i=3 and i=4

C

Knapsack Capacity → W=			0	1	2	3	4	5	6	7	8	9	10
Item	Value	Weight	0	0	0	0	0	0	0	0	0	0	0
1	10	5	0	0	0	0	0	10	10	10	10	10	10
2	40	4	0	0	0	0	40	40	40	40	40	50	50
3	30	6	0	0	0	0	40	40	40	40	40	50	70
4	50	3	0	0	0	50	50	50	50	90	90	90	90

- Final solution K[4,10], Final value is 90, the sum of value 2 and value 4.

Time complexity

- Time complexity of one problem is O(1)
  - line 5: one weight comparison
  - line 6: one read of subproblem
  - line 8: two reads of solutions
- # of subproblems is nxC, the # of elements in K. C is the capacity of knapsack.
- Time complexity of Knapsack O(1)xnxC = O(nC)

Applications

- Cutting stock problem that minimize stock's loss when a stock is cut apart.
- Financial portfolio and asset investments
- Password generation (Merkle–Hellman Knapsack Cryptosystem)