Algorithm

Ch 4: Dynamic Programming

SEUNGSOO LEE

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INCHEON NATIONAL UNIVERSITY

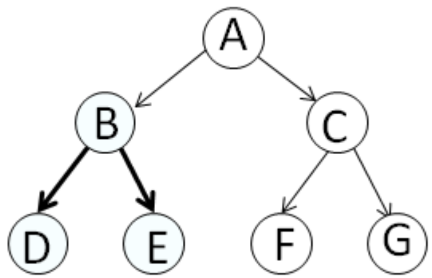
Idea

- Dynamic Programming
- 'Dynamic Paths' to solve <u>sub</u>problems (maybe)
- Difference with 'divide and conquer'
  - Conceptually, no difference

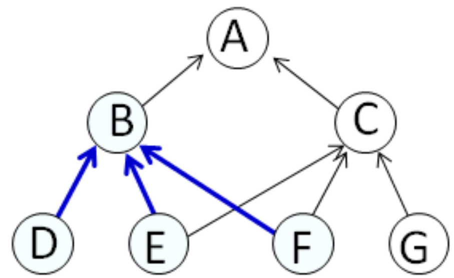
- Distinguished Property
  - Memorization of results/states of subproblems
  - Do not solve the same subproblems again

Idea

- ‘Divide and Conquer’ vs ‘Dynamic Programming’



분할 정복 알고리즘



동적 계획 알고리즘

- Merging paths are more complex
- Repeated subproblem solving

Idea

- **Dependency on previous results** should be considered when we get the solution to larger problem

- Often complex, unpredictable
- > called as an **implicit order**

Searching All Pairs of Shortest Paths

5.1 Searching All Pairs of Shortest Paths

- Goal
  - Searching the shortest paths of all starting and ending point pairs

	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									

5.1 Searching All Pairs of Shortest Paths

- How to solve this problem?

5.1 Searching All Pairs of Shortest Paths

- Dijkstra Algorithm
- Input of Dijkstra (G, starting point)
- For n starting point, run Dijkstra algorithm **repeatedly**
- Time complexity: $(n-1) \times O(n^2) = O(n^3)$, $n = |V|$

5.1 Searching All Pairs of Shortest Paths

- Can we reduce the time complexity?
- Maybe
  - In the n **repetition** of running Dijkstra
    - The weights and some shortest paths are evaluated **many times**
  - Warshall developed dynamic programming for the another problem of searching “transitive closure”
  - Floyd applied it to find the shortest paths for all pairs
  - > Floyd-Warshall Algorithm

5.1 Searching All Pairs of Shortest Paths

- For all starting nodes

- Sequentially calculate the shortest path from the starting node

- Repeat this process until we find the shortest path for all nodes

- **Re-use** the results of subproblems
- > time complexity reduction

5.1 Searching All Pairs of Shortest Paths

- What information should be memorized?

5.1 Searching All Pairs of Shortest Paths

- What information should be memorized?

- The memorized result for the final problem should be the final solution

- Correctness of the larger subsolutions should be **guaranteed**

5.1 Searching All Pairs of Shortest Paths

- What information should be memorized?

- The shortest paths between all paths that we observed so far

- If we search remaining paths and shorter path is found

  -> update the weight sum of the shortest path and **record** the path

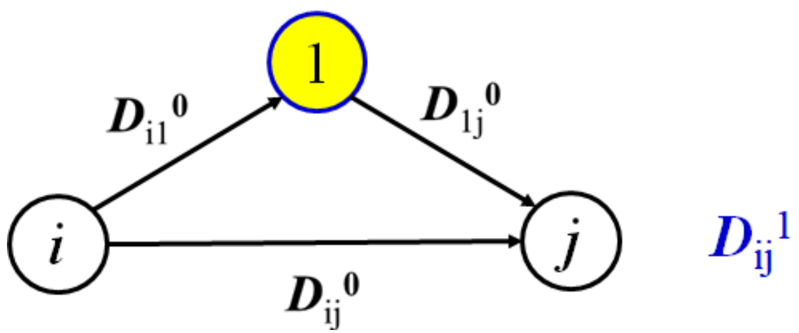
  -> If we search all paths, **the memorized path** is the shortest path

5.1 Searching All Pairs of Shortest Paths

- OK, let's memorize the shortest path that we found at each iteration.
- But what paths we will search **at each iteration**?
  (searching the correct implicit order)
- This part **should be carefully designed** to guarantee the correctness of the final results.
- Here, we will search shortest paths to pass 1 to N-2 nodes
  to reach the ending point.

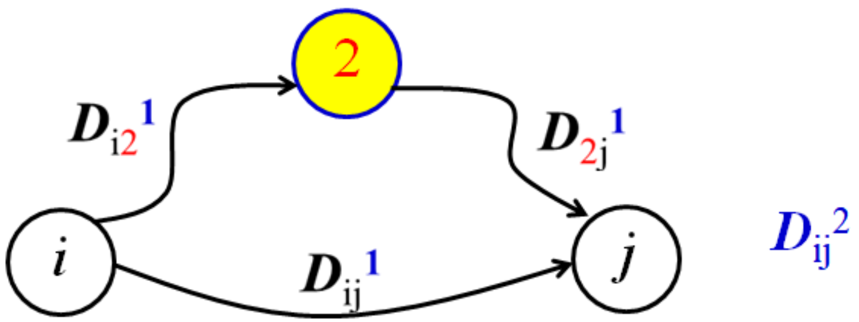
5.1 Searching All Pairs of Shortest Paths

- If we have a path from i to j passing another node
- If we found the shortest path for (i,1)
  -> we can use it to calculate the path for (i,j)
  -> min (s_path(i,1) + weight(1,j), weight (i,j))
  -> i ≠ 1, j ≠ 1



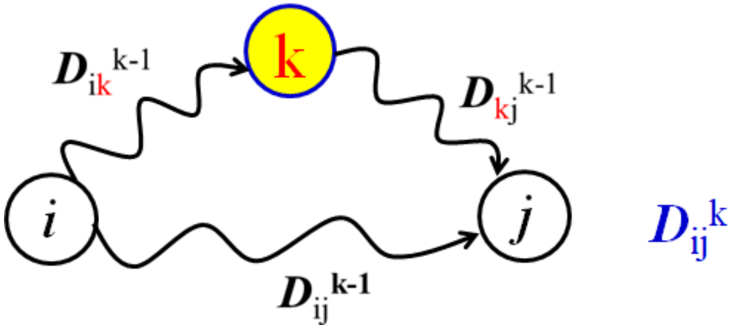
5.1 Searching All Pairs of Shortest Paths

- -> min (s_path(i,2) + weight(2,j), weight (i,j))
  -> i ≠ 2, j ≠ 2



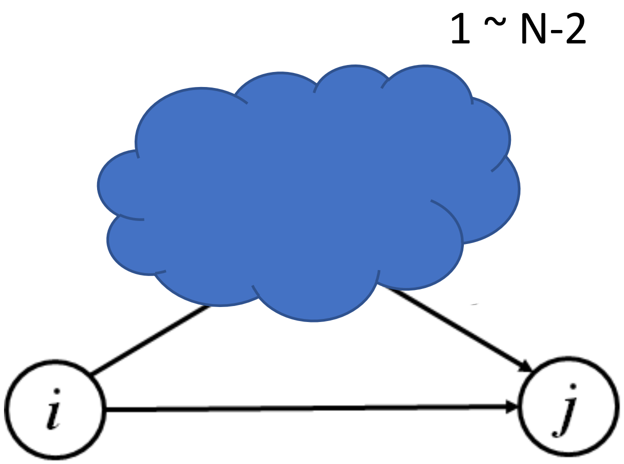
5.1 Searching All Pairs of Shortest Paths

- -> min (s_path(i,k) + weight(k,j), weight (i,j))
  -> i ≠ k, j ≠ k



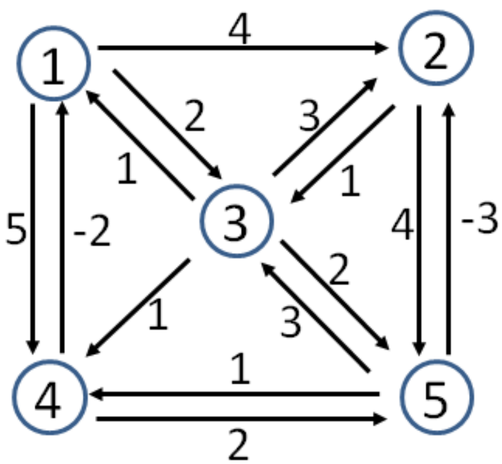
5.1 Searching All Pairs of Shortest Paths

- We need to check all paths to pass more than 1 node



5.1 Searching All Pairs of Shortest Paths

- Initial Setting



step 0

end

start

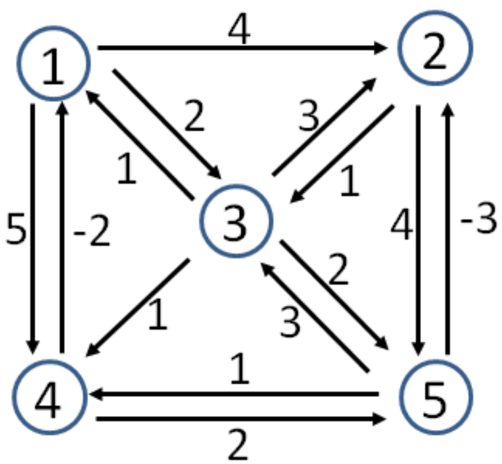
D	1	2	3	4	5
1	0	4	2	5	∞
2	∞	0	1	∞	4
3	1	3	0	1	2
4	−2	∞	∞	0	2
5	∞	−3	3	1	0

- D[i,j] = the minimum weight sum from node i to node j using **0** intermediate node

5.1 Searching All Pairs of Shortest Paths

- Then update D[i,j] = min(D[i,j], D[i,k] + D[k,j])
  for all k, k != i and k != j

step 1



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

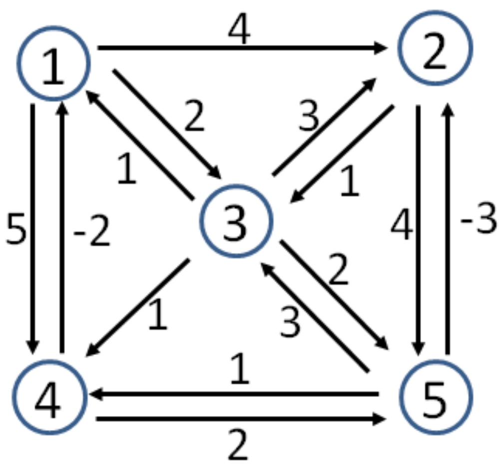
- D[i,j] = the minimum weight sum from node i to node j
  using **1** intermediate node

5.1 Searching All Pairs of Shortest Paths

- How many steps we need for searching the shortest path? N-2 steps

step s

- Does this algorithm guarantee the correct solution?



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

- D[i,j] = the minimum weight sum from node i to node j using **s** intermediate node(s)

5.1 Searching All Pairs of Shortest Paths

- Time Complexity

- Outer iteration: s, 0 ~ N-2
- Inner iteration: for all i,j,k, 0 ~ N-1

- O(N^4)

- Can we reduce more?

5.1 Searching All Pairs of Shortest Paths

- We repeatedly check the score of shortest paths

- When s = k
  - D[i,k] + D[k,j] may have various length from s+1 ~ s+s
  - s+2 length path may be re-evaluted at s=k+1 step
  - Unnecessary evaluation

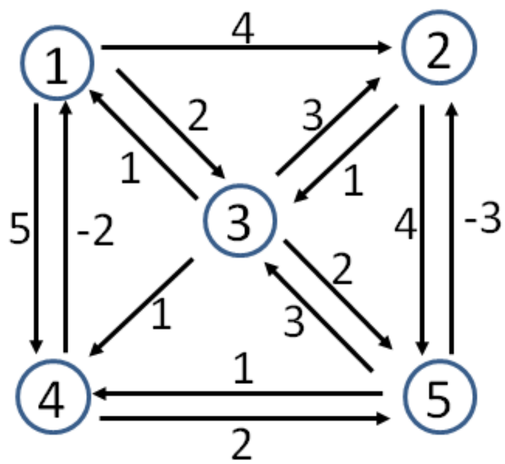
- How to remove this redundancy?

5.1 Searching All Pairs of Shortest Paths

- Update D for the same intermediate node k

- Underlying idea
  - We do not know which paths will pass this node
  - But if they use the node, they will pass this node and its adjacent two nodes
  - Therefore, finding smaller score between **D(i,j) and D(i,k) +D(k,j) always guarantees** shorter paths from i to j with any length and conditions
  - All paths have maximum N nodes
  - If we repeat this for all nodes, then we can get the shortest path

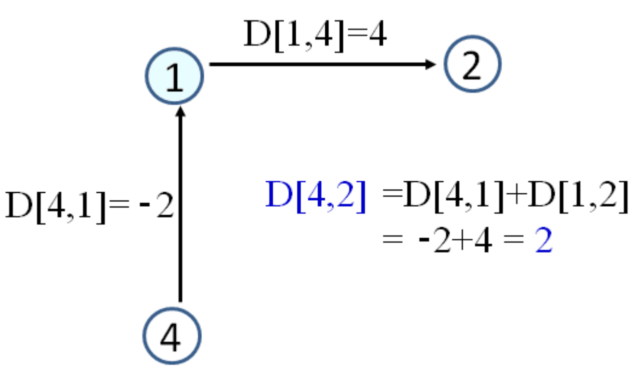
5.1 Searching All Pairs of Shortest Paths

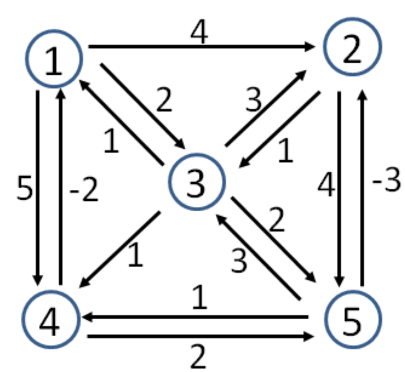


D	1	2	3	4	5
1	0	4	2	5	∞
2	∞	0	1	∞	4
3	1	3	0	1	2
4	−2	∞	∞	0	2
5	∞	−3	3	1	0

5.1 Searching All Pairs of Shortest Paths

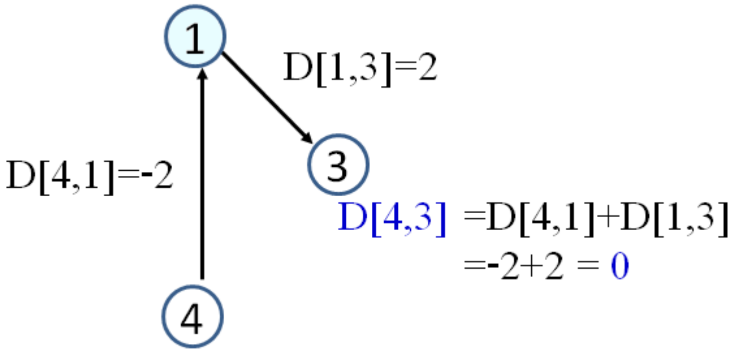
- K=1, D[i,j] = min(D[i,j], D[i,k]+D[k,j])
  - D[2,3] = min{D[2,3], D[2,1]+D[1,3]} = min{1, ∞+2} = 1
  - D[2,4] = min{D[2,4], D[2,1]+D[1,4]} = min{∞, ∞+5} = ∞
  - D[2,5] = min{D[2,5], D[2,1]+D[1,5]} = min{4, ∞+∞} = 4
  - D[3,2] = min{D[3,2], D[3,1]+D[1,2]} = min{3, 1+4} = 3
  - D[3,4] = min{D[3,4], D[3,1]+D[1,4]} = 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

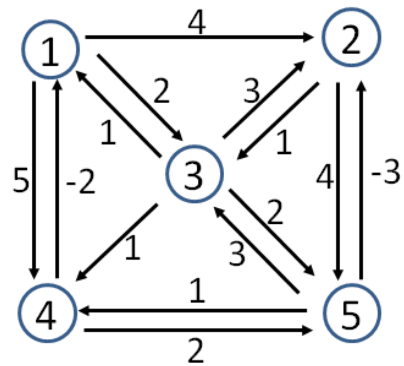




5.1 Searching All Pairs of Shortest Paths

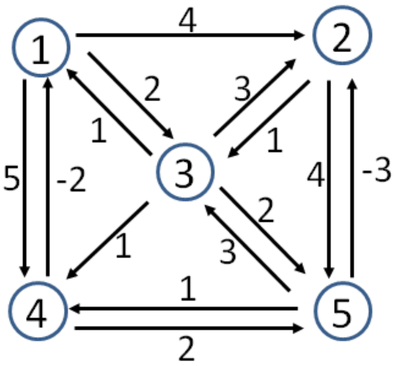
- D[4,3] = min{D[4,3], D[4,1]+D[1,3]} = min{∞, -2+2} = 0





- 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

5.1 Searching All Pairs of Shortest Paths

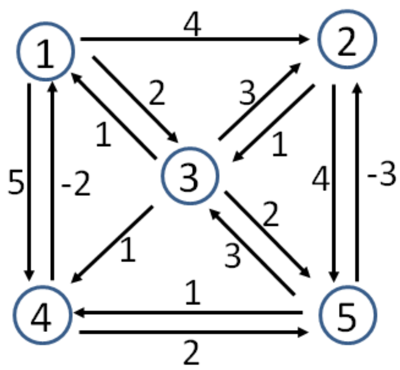


- K=1
  update entries: D[4,2] -> 2, D[4,3] -> 0

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

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

5.1 Searching All Pairs of Shortest Paths

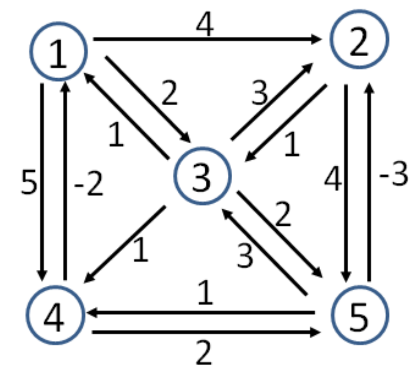


- K=2
  - D[1,5]: 8 (path: 1 -> 2 -> 5)
  - D[5,3]: -2 (path: 5 -> 2 -> 3)

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

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

5.1 Searching All Pairs of Shortest Paths



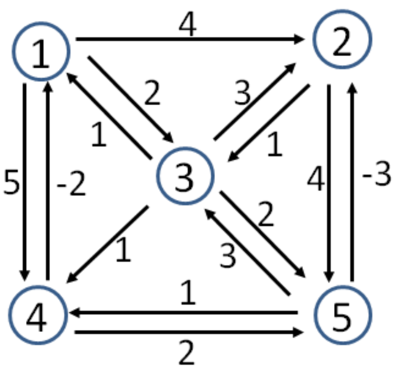
- K=3

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

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

5.1 Searching All Pairs of Shortest Paths

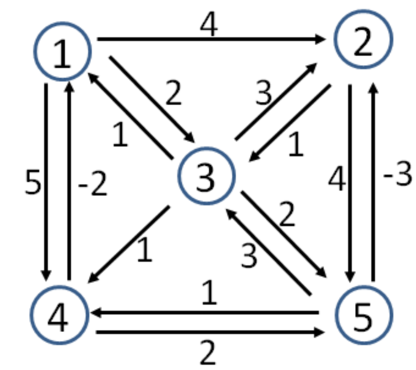
- K=4



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

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

5.1 Searching All Pairs of Shortest Paths



- K=5

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

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

5.1 Searching All Pairs of Shortest Paths

- Time Complexity

- Outer loop: K increases from 0 to N-1
- Inner loop: all (i,j) pairs are re-evaluated with constant comparison

- (N-1) x N^2

- O(N^3) -> Floyd-Warshall Algorithm

5.1 Searching All Pairs of Shortest Paths

- What information should be memorized?
- What if all results of subproblems are connected and we should evaluate them together anyway?
- We need starting subproblems correctly solved
- In this case, the shortest path to adjacent nodes

5.1 Searching All Pairs of Shortest Paths

- Write pseudo code

5.1 Searching All Pairs of Shortest Paths

AllPairsShortest(D)

- Input: 2D-array D, D[i,j]=weight of edge (i,j),
  D[i,j]= ∞ if node i and j are disconnected,
  D[i,i]=0
- Output: a 2D-array to store weight sum values for all pairs

```
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] )
```

5.1 Searching All Pairs of Shortest Paths

- Repeating Dijkstra algorithm: O(n^3)
- Floyd-Warshall algorithm: O(n^3)

- Why we use the Floyd-Warshall algorithm?
  - Easy to implement
  - A practical issue of software development

5.1 Searching All Pairs of Shortest Paths

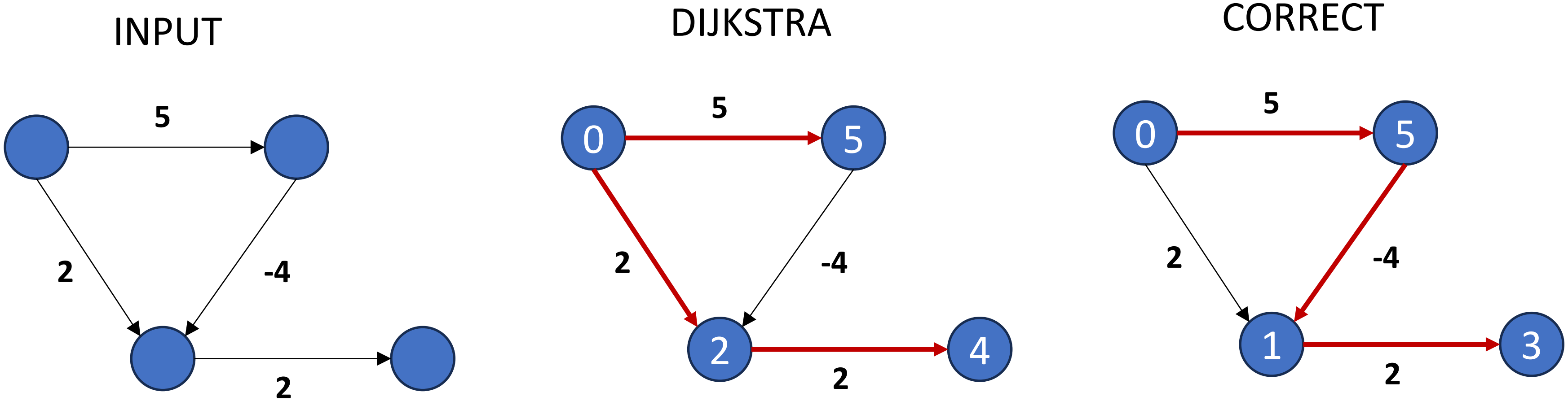
- Additional property

- Negative **cycle**
  - a cycle whose weight sum is negative
  - We can not find the shortest path of such graphs
  - Because the shortest path is **infinitely** visiting the nodes of the cycle
  - If we restrict a path to visit a node **at most one time**?

5.1 Searching All Pairs of Shortest Paths

- **Bellman-Ford Algorithm**
- Shortest path algorithm to allow negative weights
- Starting point is given

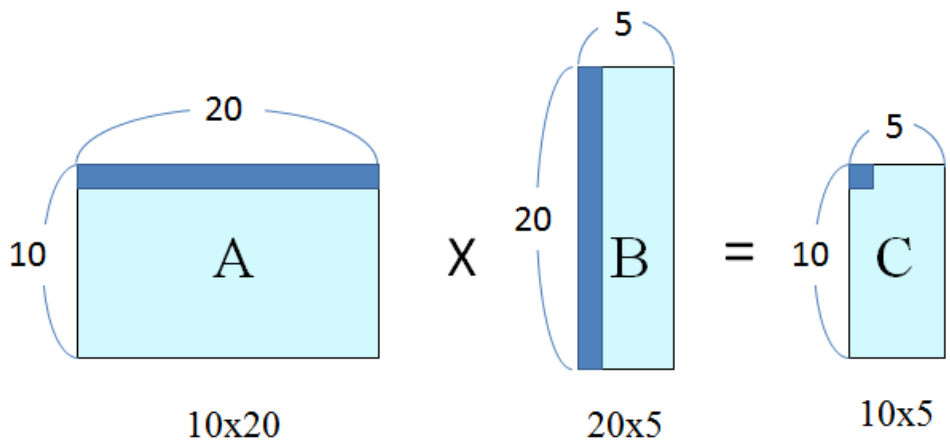
NEGATIVE EDGE WEIGHTS



Chained Matrix Multiplications

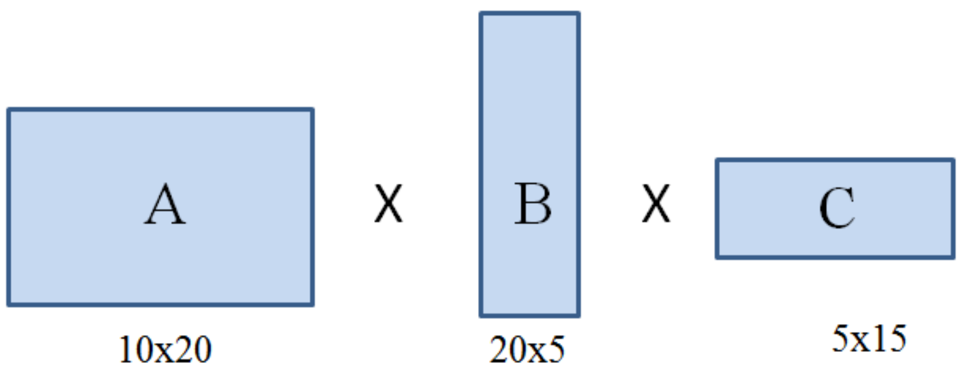
5.2 Chained Matrix Multiplications

- Goal
  - Find the fastest order of changed matrix multiplications
- Matrix A: 10x20, Matrix B: 20x5
  - total element multiplication 10x20x5 = 1,000
- Matrix C: 10x5
- To obtain the element of C,
  20 multiplication are required (one row of A x one column of B)



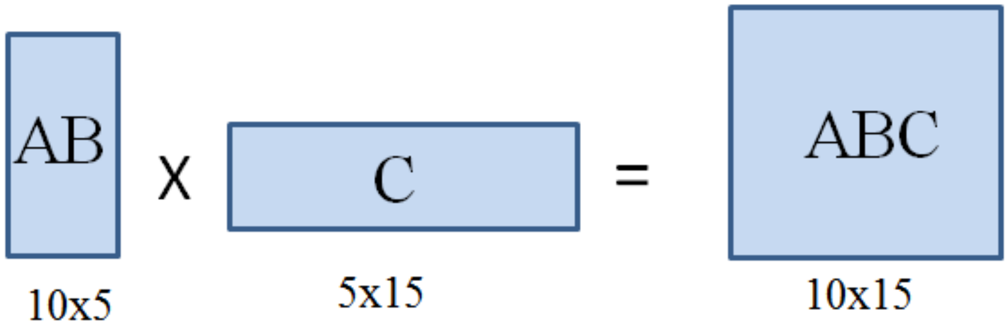
5.2 Chained Matrix Multiplications

- Three matrix multiplication case
  - AxBxC = (AxB)xC = Ax(BxC)
- A: 10x20
- B: 20x5
- C: 5x15



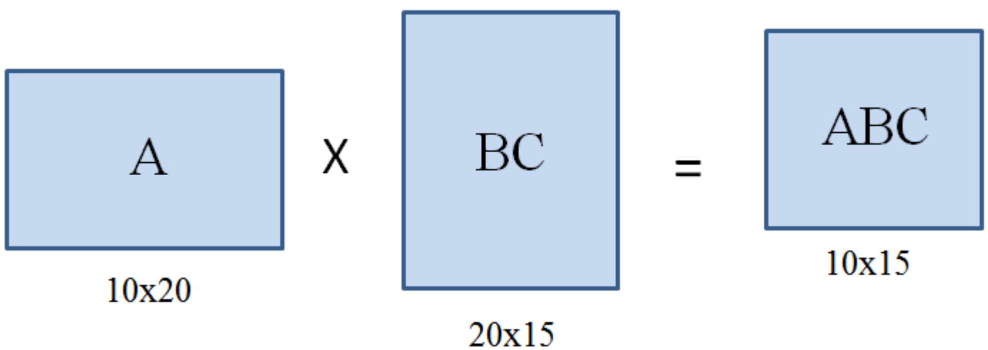
5.2 Chained Matrix Multiplications

- The first case
  - (AxB)xC
- Element multiplication of AxB:
  - 10x20x5 = 1,000
  - Shape of the result matrix: 10x5
- Element multiplication of the result matrix x C:
  - 10x5x15 = 750
- Total 1,000 + 750 = 1,750



5.2 Chained Matrix Multiplications

- The second case
  - Ax(BxC)
- Element multiplication of BxC:
  - 20x5x15 = 1,500
  - Shape of the result matrix: 20x15
- Element multiplication of the result matrix x C:
  - 10x20x15 = 3000
- Total 1,500 + 3,000 = 4,500



5.2 Chained Matrix Multiplications

- Huge difference of calculation speed
  - 2800 more multiplication in the seconed case

- Problem: finding **the best order** of applying the binary operator (matrix multiplication)

- How to solve this problem?

5.2 Chained Matrix Multiplications

- Given N matrices
- We can assign the multiplication order
- Total N-1 indices

- Permutation search
  - (N-1) factorial
  - For each permutation, we need O(N) multiplicaitons and additions
  - O(N(N-1)!)

5.2 Chained Matrix Multiplications

- How to efficiently evaluate the best order?
- **Re-use** the results of already solved subproblems

- Given a permutation, for example, **143**25 for 6 matrices
- Assume the multiplication freq. is checked for the permutation
- How to evaluate the multiplication frequency of **143**52?

5.2 Chained Matrix Multiplications

- We can reuse the results of frequency evaluation for (143)

- How to generalize this re-using mechanism?
- What subproblems can be used to get the final result?
- AxBxCxDxE
- (AxB)x(CxDxE)
- Ax(Bx(CxDxE)) … Some components are **duplicated**

5.2 Chained Matrix Multiplications

- Let’s start to find the **minimum frequency** for small chains

- Length 1 chain: no multiplication
- Length 2 chain: only one order
- Length 3 chain: two orders
- Length 4 chain: four orders
- Length 5 chain: eight orders
- Length 6 chain: 2^(k-1) orders

5.2 Chained Matrix Multiplications

- Store the order to multiplicate small chains with minimum frequency

- Any longer chain using the small chain **can use the result**
  - No dependency to other multiplication if the chain multiply the small chain first

- However, larger problems may use different smaller chains
  -> Check all possible combinations

5.2 Chained Matrix Multiplications

부분 문제 크기						부분 문제 개수
1	A	B	C	D	E	5개
2	AxB	BxC	CxD	DxE		4개

- Evaluate the minimum frequency of multiplication for **the shortest chain first**
- Derive the minimum frequency for longer chain

5.2 Chained Matrix Multiplications

(AXB)XC AX(BXC)

3	$\boxed{A \times B \times C}$	$B \times C \times D$	$C \times D \times E$	3개
4	$A \times B \times C \times D$	$B \times C \times D \times E$		2개
5	$A \times B \times C \times D \times E$			1개

- Repeat the valuation for all length
- What is the implicit order?
- What do we need to evaluate and memorize to get the results of subproblems?

5.2 Chained Matrix Multiplications

- To determine implicit order
  - How to derive the solution of the larger problem from those of smaller problems?
  - all subsolutions have **minimum frequency**
  - Mixing them
  - We use only one multiplication to derive the final solution
  - We can use any result of subproblems smaller than the current subproblem to solve

5.2 Chained Matrix Multiplications

$(A_i) \times (A_{i+1} \times A_{i+2} \times \cdots \times A_j)$ k=i 일때

One
More
multiplication

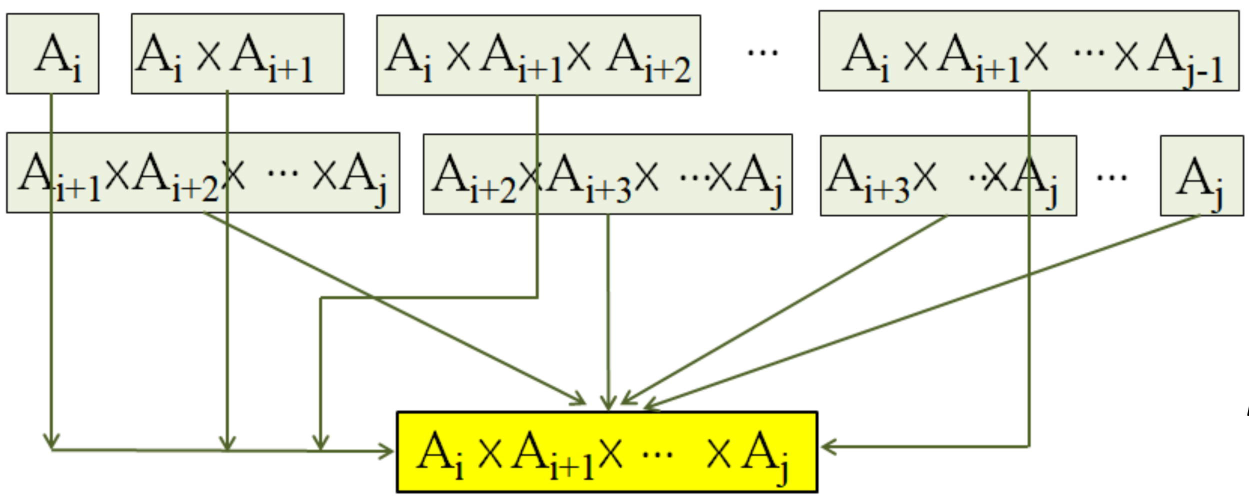
$(A_i \times A_{i+1}) \times (A_{i+2} \times A_{i+3} \times \cdots \times A_j)$ k=i +1 일때

$(A_i \times A_{i+1} \times A_{i+2}) \times (A_{i+3} \times \cdots \times A_j)$ k=i +2 일때

: :

Any other
orders to derive
the result
of this chain?

$(A_i \times A_{i+1} \times \cdots \times A_{j-1}) \times (A_j)$ k=j-1 일때



Find the minimum
frequency

implicit order

5.2 Chained Matrix Multiplications

- To find the minimum freq. for length L chain
  - Total L–1 cases to check

- All possible L-1 length chains should be checked first

- To find the minimum freq. for length L-1 chain
  > All possible L-2 length chains should be evaluated

- **We need to memorize all matrix chains for all lengths**

5.2 Chained Matrix Multiplications

- We only need to check subproblems shorter than our current subproblem
- Implicit order is determined
- Now we can represent them as a data structure

$\boxed{A_1 \text{X } A_2}$ $\boxed{A_2 \text{X } A_3}$ $\boxed{A_3 \text{X } A_4}$ $\boxed{A_{n-2} \text{X } A_{n-1}}$ $\boxed{A_{n-1} \text{X } A_n}$ n-1 개

$\boxed{A_1 \text{X } A_2 \text{X } A_3}$ $\boxed{A_2 \text{X } A_3 \text{X } A_4}$ $\boxed{A_3 \text{X } A_4 \text{ X } A_5}$ $\boxed{A_{n-2} \text{X } A_{n-1} \text{ X } A_n}$ n-2 개

$\boxed{A_1 \text{X } A_2 \text{X } A_3 \text{ X ... X } A_{n-1}}$ $\boxed{A_2 \text{X } A_3 \text{X } A_4 \text{ X ... X } A_n}$ 2 개

$\boxed{A_1 \text{X } A_2 \text{X } A_3 \text{ X ... X } A_{n-1} \text{X } A_n}$ 1 개

5.2 Chained Matrix Multiplications

- Matrix index start from 1
- M1 x M2
  - Needs M1 and M2 results
- M1 x M2 x M3
  - M1 x M2 and M3, M1 and M2 x M3
- M1 x M2 x .. x Mj?
  - min(f(1,2,j), f(1,3,j), f(1,4,j) … f(1,j-1,j))
  - f: (x,y,z) -> N
    x is the **starting** point, y is the **splitting** point, z is the **final** point
  - Point: a matrix index

5.2 Chained Matrix Multiplications

- The results that we need to **store**: O(N^2)
- > N x N matrix
- Freq. of multiplication of i to j chain: f(i, j)

- f(i,j) = min(**f(i,i+1,j)**, f(i,i+2,j), f(i,i+3,j) … f(i,j-1,j))

- **(i,i+1), (i+1,j)**
- (i,i+2) (i+2,j)
- (i,i+3) (i+3,j)
- …

5.2 Chained Matrix Multiplications

- We need to get the results of entries on the diagonal line whose i and j index is less than or equal to the target entry (according to the implicit order)

(1,2) (2,n)

L = n

C	1	2	3	.	.	n-1	n
1	0						
2		0					
3			0				
.				0			
.					0		
n-1						0	
n							0

1개

(1,3) (3,n)

L = n

C	1	2	3	.	.	n-1	n
1	0						
2		0					
3			0				
.				0			
.					0		
n-1						0	
n							0

1개

(1,n-1) (n-1,n)

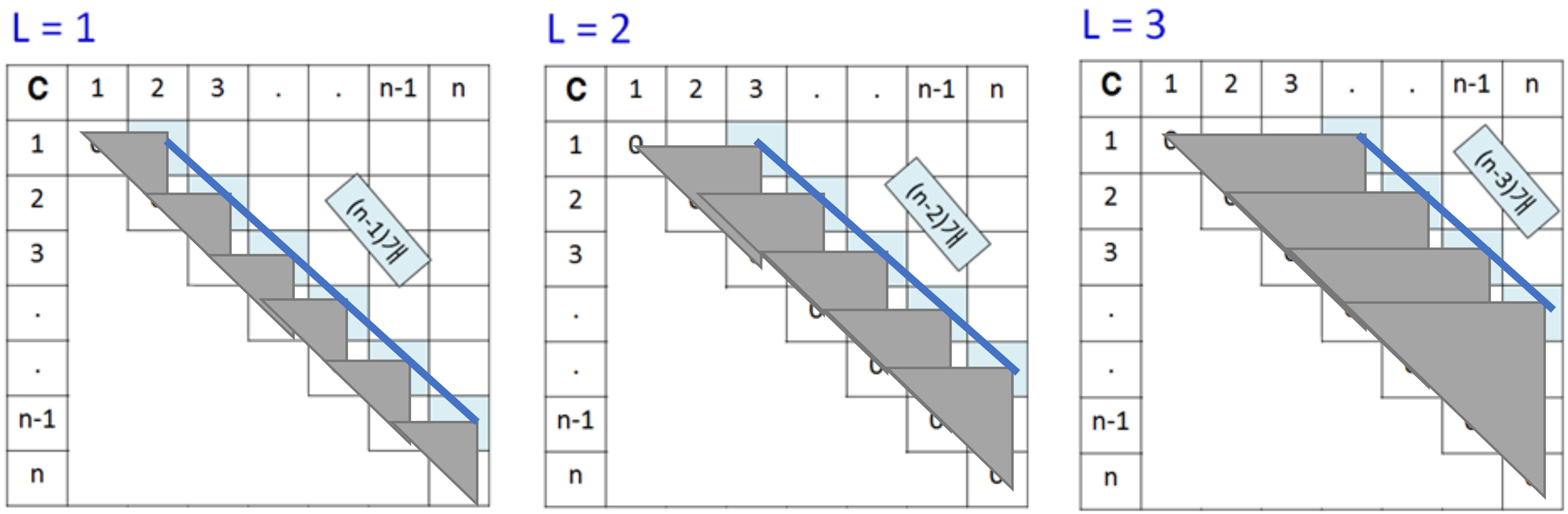
L = n

C	1	2	3	.	.	n-1	n
1	0						
2		0					
3			0				
.				0			
.					0		
n-1						0	
n							0

1개

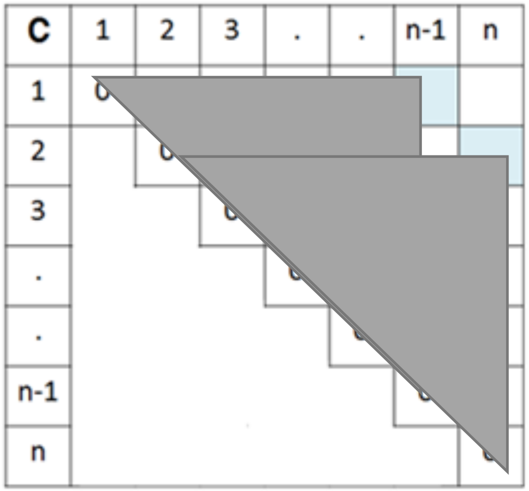
5.2 Chained Matrix Multiplications

- Not to break the implicit order
  - Start evaluation of the safe entries first

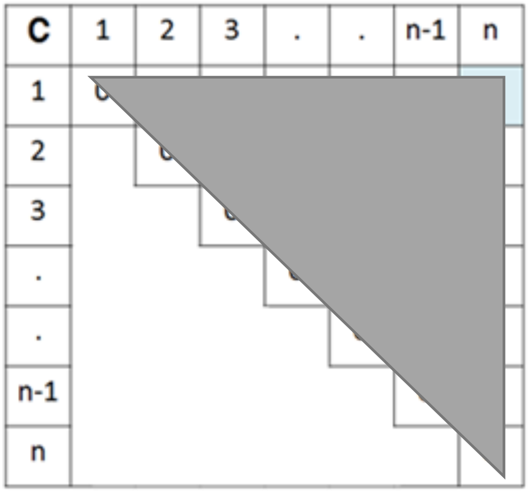


5.2 Chained Matrix Multiplications

L = n-2



L = n-1



5.2 Chained Matrix Multiplications

- Matrix shape change
  - Result matrix shape: [d_i x d_j] x [d_j x d_k] = d_i x d_k
  - Multiplication: d_i x d_j x d_k

$$(A_i) \times (A_{i+1} \times A_{i+2} \times \cdots \times A_j) \quad k=i \text{ 일때}$$

d_{i-1} x d_i d_i x d_j

$$(A_i \times A_{i+1}) \times (A_{i+2} \times A_{i+3} \times \cdots \times A_j) \quad k=i+1 \text{ 일때}$$

d_{i-1} x d_{i+1} d_{i+1} x d_j

$$(A_i \times A_{i+1} \times A_{i+2}) \times (A_{i+3} \times \cdots \times A_j) \quad k=i+2 \text{ 일때}$$

d_{i-1} x d_{i+2} d_{i+2} x d_j

:

$$(A_i \times A_{i+1} \times \cdots \times A_{j-1}) \times (A_j) \quad k=j-1 \text{ 일때}$$

d_{i-1} x d_{j-1} d_{j-1} x d_j

5.2 Chained Matrix Multiplications

- 1. Initialize diagonal entries to 0

- 2. Assign **infinite** number to the entries for subproblems to solve at next iteration

- 3. Find the minimum freq. of the subproblems by comparing all results in their implicit orders

5.2 Chained Matrix Multiplications

- 1. Initialize diagonal entries to 0

- 2. Assign **infinite** number to the entries for subproblems to solve at next iteration
  - > iteration counter L = 1 ~ N-1
  - > initialize all entries of L > 0 to infinite number (because we will use **min()** function)

- 3. Find the minimum freq. of the subproblems by comparing all results in their implicit orders

5.2 Chained Matrix Multiplications

- 1. Initialize diagonal entries to 0
- 2. Assign **infinite** number to the entries for subproblems to solve at next iteration
- 3. Find the minimum freq. of the subproblems by comparing all results in their implicit orders
  - > given L, evaluate f(i,i+L) for i in [1,N-L]
  - > given i, f(i,j) = **min**(f(i,i+1,j), f(i,i+2,j), f(i,i+3,j),… f(i,k,j)) = $d_i\ d_k\ d_j$ + C[i,k] + C[k,j] (k in [i,j])

5.2 Chained Matrix Multiplications

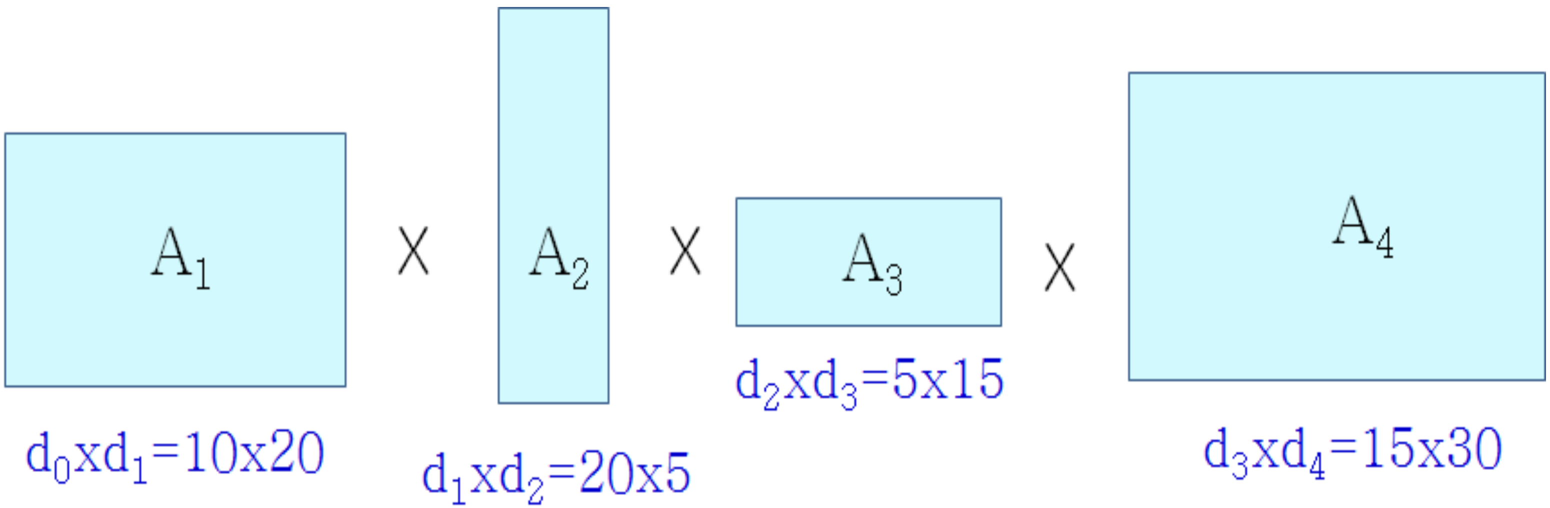
MatrixChain

- Input: $A_1 x A_2 x \cdots x A_n$, A_1 shape is $d_0 x d_1$, A_2 shape is $d_1 x d_2$, $\cdots$, A_n shape is $d_{n-1} x d_n$
- Output: minimum freq. of element multiplication for the matrix chain multiplication

```
for i = 1 to n
        C[i, i] = 0
for L = 1 to n-1 {
      for i = 1 to n-L {
            j = i + L
            C[i, j] = ∞
            for k = i to j-1 {
                    temp = C[i, k] + C[k+1, j] + d_{i-1}d_kd_j
                    if (temp < C[i, j])
                          C[i, j] = temp
            }
      }
}
return C[1,n]
```

5.2 Chained Matrix Multiplications

- A_1: 10x20, A_2: 20x5, A_3: 5x15, A_4: 15x30
- What is the final state of the frequency matrix C ?



L=1일 때

- C[1, 2] = $d_0 d_1 d_2$ = 10x20x5 = 1,000
- C[2, 3] = 20x5x15 = 1,500
- C[3, 4] = 5x15x30 = 2,250

i =1

i =2

i =3

L = 1

C	1	2	3	4
1	0			
2		0		
3			0	
4				0

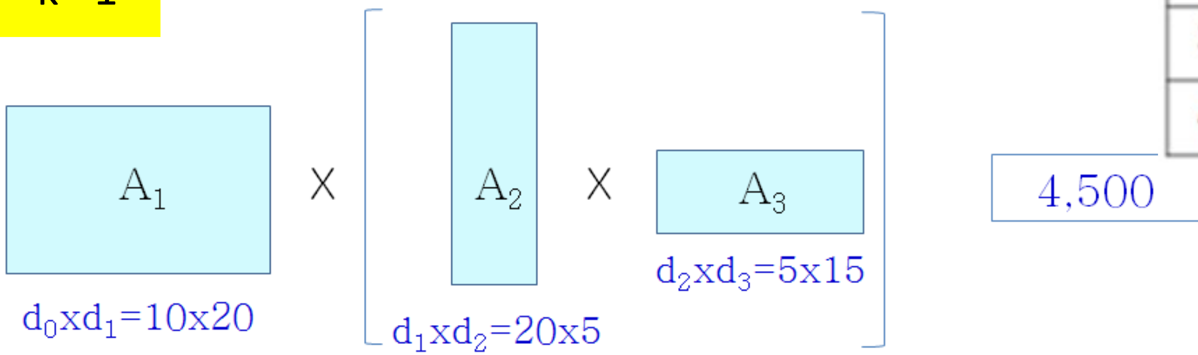
L=2, i=1일 때

- $A_1 x A_2 x A_3$을 계산한다. C[1, 3] =1,750

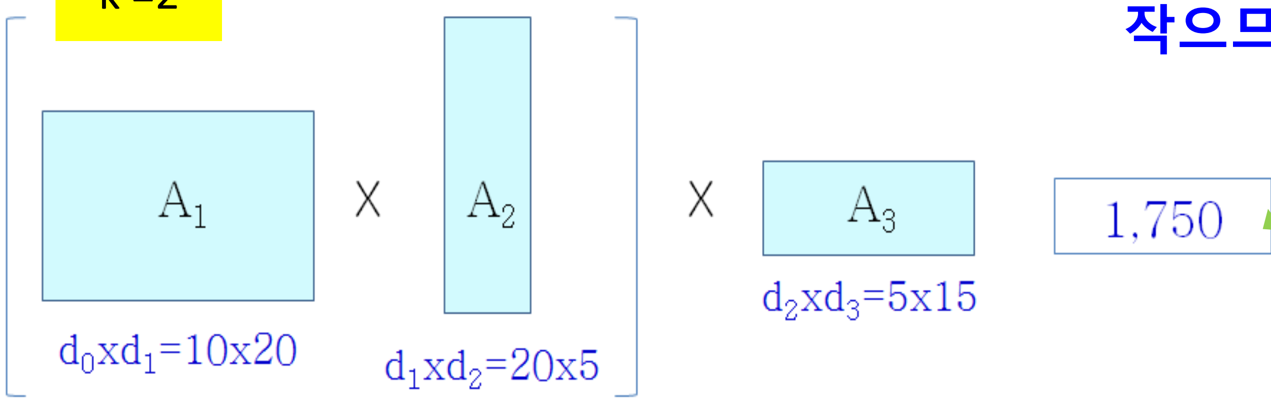
L = 2

C	1	2	3	4
1	0			
2		0		
3			0	
4				0

k =1



k =2



1,750이 4,500보다 작으므로

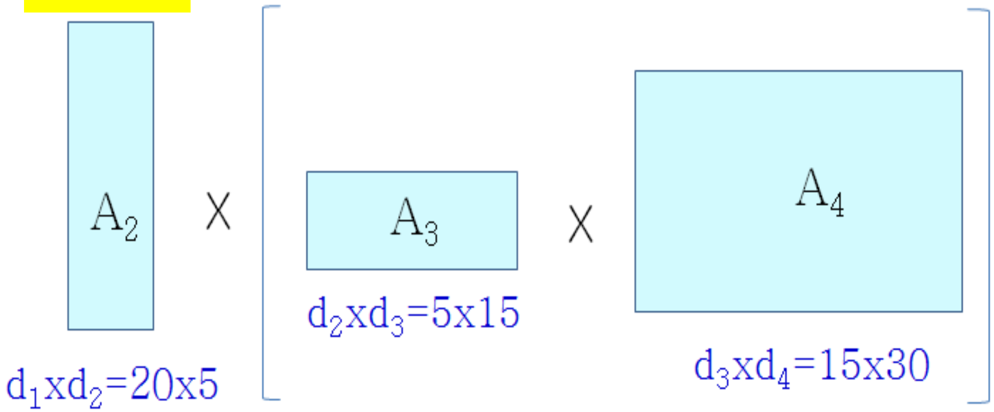
L=2, i=2일 때

- $A_2 x A_3 x A_4$를 계산한다. C[2, 4] =5,250

L = 2

C	1	2	3	4
1	0			
2		0		
3			0	
4				0

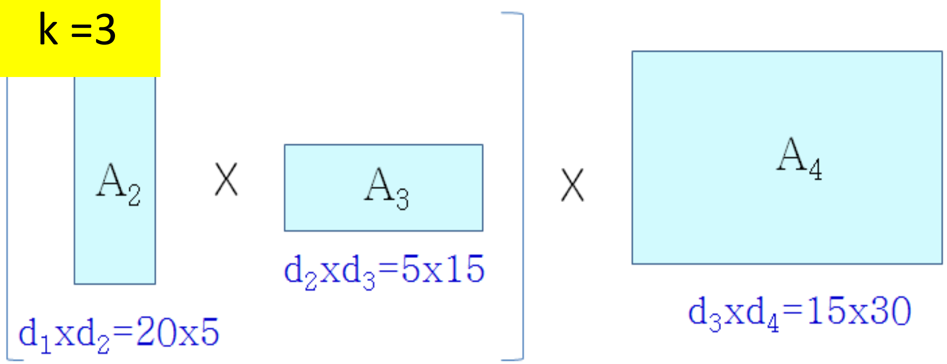
k =2



5,250

5,250이 10,500보다 작으므로

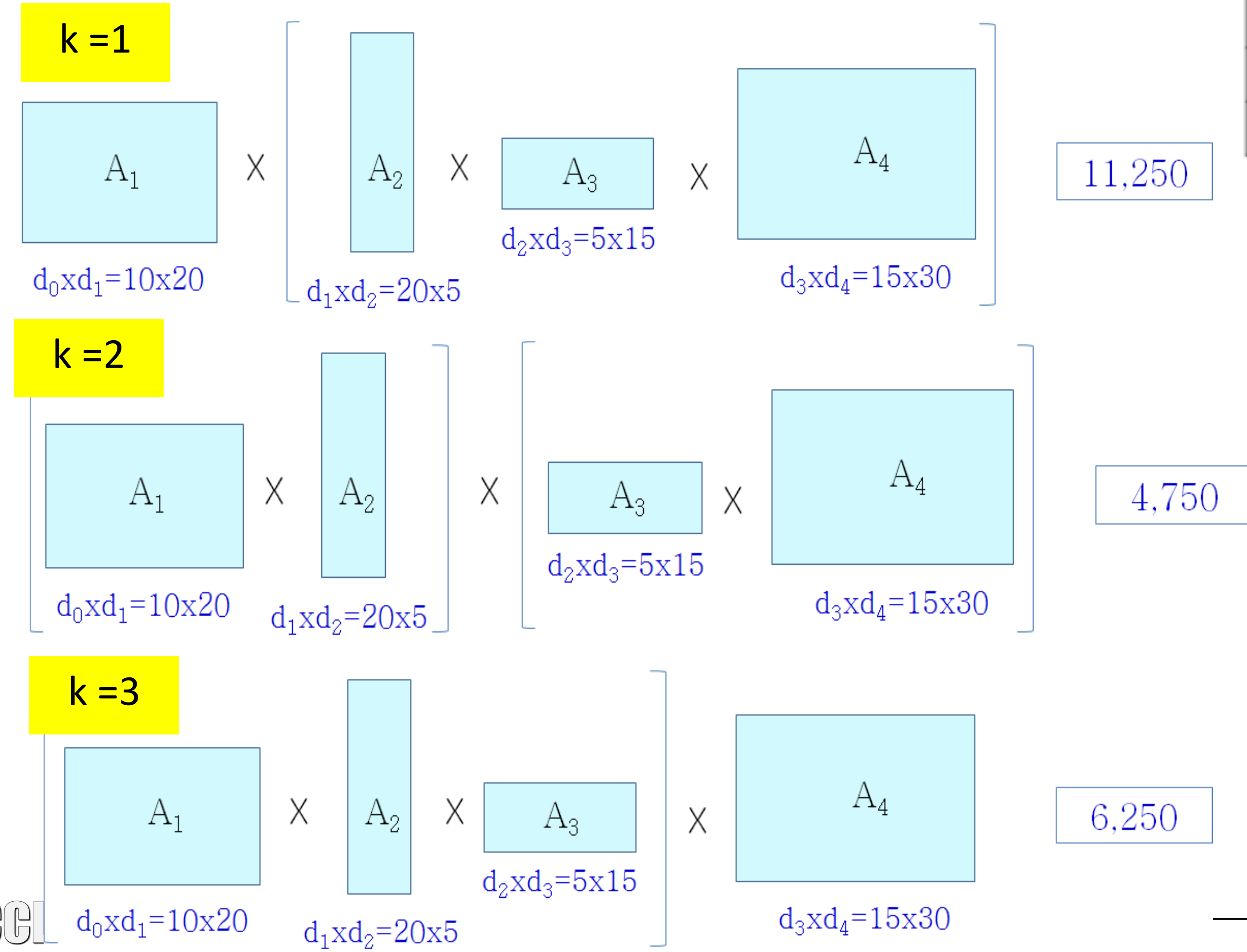
k =3



10,500

L=3일 때

- $A_1xA_2xA_3xA_4$를 계산한다. C[1, 4] =4,750



L = 3

C	1	2	3	4
1	0			
2		0		
3			0	
4				0

가장 작으므로

5.2 Chained Matrix Multiplications

- (Derive it together)

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

5.2 Chained Matrix Multiplications

- Time Complexity

5.2 Chained Matrix Multiplications

MatrixChain

- Input: $A_1 x A_2 x \cdots x A_n$, A_1 shape is $d_0 x d_1$, A_2 shape is $d_1 x d_2$, ⋯, A_n shape is $d_{n-1} x d_n$
- Output: minimum freq. of element multiplication for the matrix chain multiplication

```
1. for i = 1 to n
2.         C[i, i] = 0
3.  for L = 1 to n-1 {                                      O(n)
4.       for i = 1 to n-L {                                 O(n)
5.             j = i + L
6.            C[i, j] = ∞
7.            for k = i to j-1 {                            O(n)
8.                    temp = C[i, k] + C[k+1, j] + d_{i-1}d_{k}d_{j}      O(1)
9.                    if (temp < C[i, j])
10.                            C[i, j] = temp
                     }
               }
         }                                                  total O(n^3)
11. return C[1,n]
```

Edit Distance

5.3 Edit Distance

- Conditions
  - Two input stringgs are given

- Goal
  - Evaluate “Edit Distance”

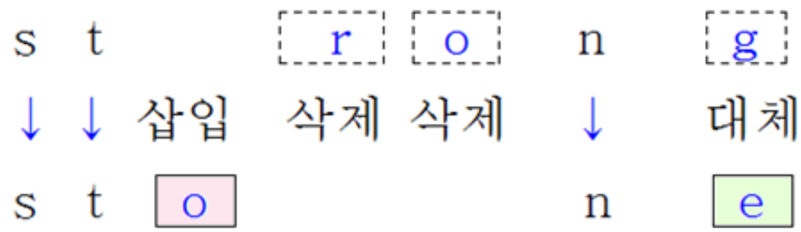
- Edit Distance
  - The minimal number of edition to make two strings equal
  - Edition: insertion, deletion, substitution

5.3 Edit Distance

- Distance:
- Metrics should satisfy the following conditions
  - Positive value
  - Symmetry (d(a,b) = d(b,a))
  - Triangle inequality (d(a,b)+d(b,c) > d(a,c))

5.3 Edit Distance

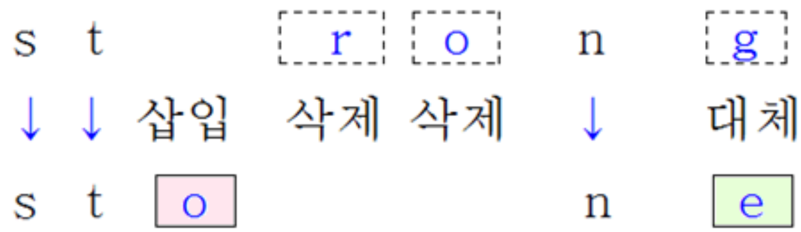
- String 1: ‘strong’
- String 2: ‘stone’



- Use ‘s’ and ‘t’ without edition
- Insert ‘o’
- Delete ‘r’ and ‘o’
- Use ‘n’ without edition
- Substitute ‘g’ to e

5.3 Edit Distance

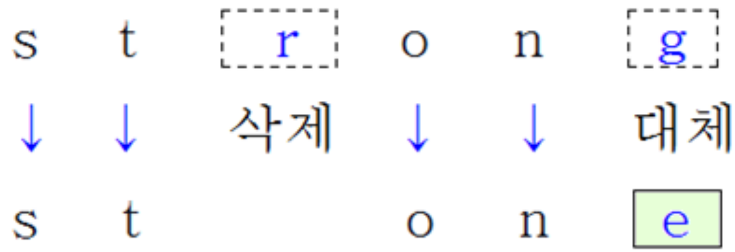
- String 1: ‘strong’
- String 2: ‘stone’



- 1 insertion, 2 deletion, 1 substitution
- 4 editions
- Is it minimum?

5.3 Edit Distance

- Use ‘s’ and ‘t’
- Delete ‘r’
- Use ‘o’ and ‘n’
- Substitute ‘g’ to ‘e’



- 1 deletion, 1 substitution -> 2 editions

5.3 Edit Distance

- How to find the minimum number of editions?
  - For arbitrarily given two strings

- Worst case ?

5.3 Edit Distance

- String 1: N character
- String 2: M character

- Delete N and insert M
- Or delete M and insert N characters
- **Upper bound** of the number of editions: N + M

- Substitute all N characters to M, if N > M
- **Upper bound** of the number of editions: N

- How to reduce the editions?

5.3 Edit Distance

- String 1: N character
- String 2: M character

- If we can find matching substrings?
  - Use them **without any edition**

- If we found matching substrings, required editions: deleting all unmatched characters

- How to find the matching substrings?

5.3 Edit Distance

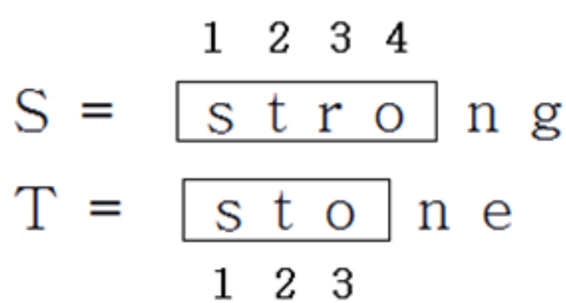
- String 1: N character
- String 2: M character

- Two empty strings are always equal
- If there is a **substring** to use in the final solution, we would get it after some editions

- Do not know the paths
- Let's check every path
- Definitely some paths will be checked **repeatedly**
  > dynamic programming

5.3 Edit Distance

- Matching ‘strong’ and ‘stone’
- Assume that we know edit distance between ‘stro’ and ‘sto’ **prefix**
- Edit distance of strong and stone is
  sum (E(stro, sto) + E(ng,ne))
- Idea? we can **recursively** find the edit distance from the starting position



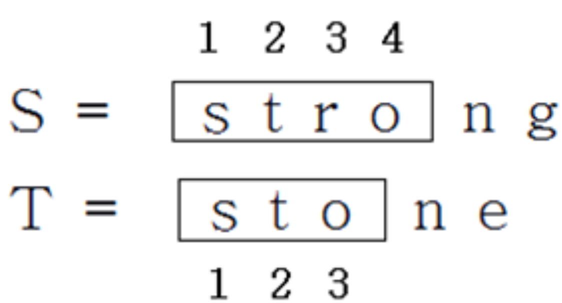
5.3 Edit Distance

- Is this property always correct?
  - Edit distance of strong and stone is
    sum (E(stro, sto) + E(ng,ne))

$$\begin{array}{rl} & \;1\;\;2\;\;3\;\;4 \\ S = & \boxed{s\ t\ r\ o}\ n\ g \\ T = & \boxed{s\ t\ o}\ n\ e \\ & \;1\;\;2\;\;3 \end{array}$$

5.3 Edit Distance

- Yes, but we can not generalize the property
- sum (E(stro, st) + E(ng,one)) = E(strong, stone)?
  - May not be correct
- We need to search all combinations



5.3 Edit Distance

- Given string S and T with length m and n,
  s_i: character at ith position of S
  t_j: character at jth position of T
  i is in [1, m],   j is in [1,n]

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

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

- E[i,j]: edit distance between prefix i characters of S
  and prefix j characters of T

5.3 Edit Distance

- ‘strong’ and ‘stone’
- Subproblem: ‘stro’ and ‘sto’ -> E[4,3]
- E[6,5] can be derived from E[4,3]
- Example of algorithm running

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

5.3 Edit Distance

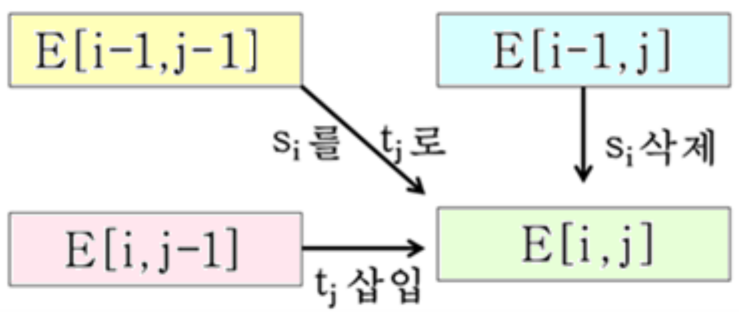
- $s_1 \rightarrow t_1$ ['s' -> 's']: $E[1, 1] = 0$
  - $s_1 = t_1 =$'s'
- $s_1 \rightarrow t_1 t_2$ ['s' -> 'st']: $E[1, 2] = 1$
  - $s_1 = t_1 =$'s',
  - 't' insertion
- $s_1 s_2 \rightarrow t_1$ ['st' -> 's']: $E[2, 1] = 1$
  - $s_1 = t_1 =$'s',
  - 't' deletion
- $s_1 s_2 \rightarrow t_1 t_2$ ['st' -> 'st']: $E[2, 2] = 0$
  - $s_1 = t_1 =$'s'
  - $s_2 = t_2 =$ 't'
  - $E[2, 2] = E[1,1] + 0 = 0$

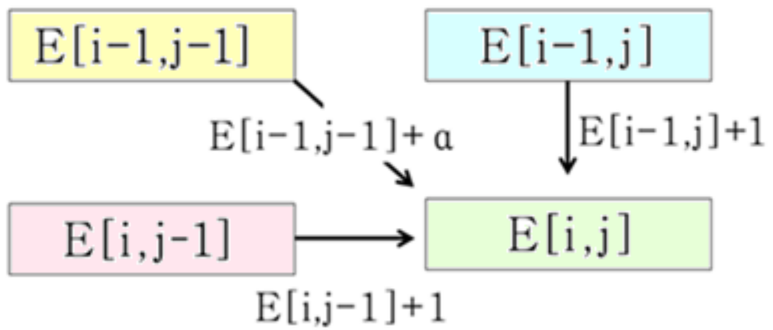
5.3 Edit Distance

- $s_1s_2s_3s_4 \rightarrow t_1t_2t_3$ [‘stro’ -> ‘sto’]: E[4,3]?
- $s_1s_2s_3s_4 \rightarrow t_1t_2$ [‘stro’ -> ‘st’]: E[4,2],
  - Insert t_3 = ‘o’, E[4,2]+1
- $s_1s_2s_3 \rightarrow t_1t_2t_3$ [‘str’ -> ‘sto’]: E[3,3],
  - delete s_4=‘o’, E[3,3]+1
- $s_1s_2s_3 \rightarrow t_1t_2$ [‘str’ -> ‘st’]: E[3,2],
  - s_4 =‘o’ -> t_3=‘o’ , no need to change ‘o’, E[3,2] + 0

5.3 Edit Distance

- E[4,3] = min(E[4,2] +1, E[3,3]+1, E[3,2])
- E[4,3] = E[3,2] = 1
- Generalized version: E[i-1,j], E[i,j-1], E[i-1,j-1]





5.3 Edit Distance

- What we will do:

- We will derive E[i,j] from all possible paths from i=0 and j=0 conditions
- Starting condition
  - E[0,0]: edit distance = 0
- For given i and j
  - E[i,j] = min(E[i-1,j] + a, E[i,j-1] +b, E[i-1,j-1] + c)
  - a,b,c,: required **editions** to obtain E[i,j] from the substring pairs

5.3 Edit Distance

- Is this correct?
- If S_i and T_j is equal
  - E[i-1, j-1] is the minimal edit distance
- Otherwise?
  - E[i-1,j] or E[i,j-1] + one insertion(or deletion)
  - E[i-1,j-1] + one substitution

5.3 Edit Distance

T

E		ε	s	t	o
	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

S

5.3 Edit Distance

- E[i,j-1]
  - $s_1\ s_2 \cdots s_i$ and $t_1\ t_2 \cdots t_{j-1}$
  - Insert t_j
  - E[i,j-1]+1 to build $s_1\ s_2 \cdots s_i$ and $t_1\ t_2 \cdots t_j$
- E[i-1,j]
  - $s_1\ s_2 \cdots s_{i-1}$ and $t_1\ t_2 \cdots t_j$
  - delete s_i
  - E[i-1,j]+1
- E[i-1,j-1]
  - $s_i = t_i$, E[i-1,j-1] +0
  - Otherwise, E[i-1,j-1] +1 (substitution)

5.3 Edit Distance

$$E[i,j] = \min\{E[i,j-1]+1,\ E[i-1,i]+1,\ E[i-1,j-1]+\alpha\}$$
$$,\ \alpha=1 \text{ if } s_i \neq t_i \text{ else } \alpha=0$$

- Initialize E[i,j] from (0,0)

T

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

5.3 Edit Distance

Editdistance(S,T)

- Input: string S and T, |S| = m, |T|= n
- Output: Edit distance of S and T, E[m,n]

```
1. for i=0 to m    E[i, 0] = i
2. for j=0 to n    E[0, j] = j
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]
```

5.3 Edit Distance

- The result of running the algorithm for ‘strong’ and ‘stone’

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**

5.3 Edit Distance

- 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
  - (s,) -> (s,s)

	T	ε	s
S	i \ j	0	1
ε	0	0	1
s	1	1	(2)

5.3 Edit Distance

- ‘E[0,1]+1=2’
- (,s) -> (s,s)

	T	ε	s
S	i \ j	0	1
ε	0	0	1
s	1	1	(2)

5.3 Edit Distance

- E[0,0]+ α = 0+0 = 0 (s,s)
- (,) -> (s,s)
- s=s

	T	ε	s
S	i \ j	0	1
ε	0	0	1
s	1	1	0

- min(2,2,0) = 0

5.3 Edit Distance

- $E[2, 2] = \min\{E[2, 1]+1, E[1, 2]+1, \underline{E[1, 1]+\alpha}\}$

  $= \min\{(1+1), (1+1), \underline{(0+0)}\}$

  $= 0$

- (s,s) -> (st,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

5.3 Edit Distance

- E[3, 2] = min{E[3, 1]+1, E[2, 3]+1, E[2, 2]+α}

  = min{2+1, 0+1, 1+1}

  = 1

- (st,st) -> (str,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
r	3	3	2	1

5.3 Edit Distance

- E[4, 3] = min{E[4, 2]+1, E[3, 3]+1, E[3, 2]+α}

  = min{(2+1), (1+1), (1+0)}

  = 1

- (st,st) -> (sto,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

5.3 Edit Distance

- E[5, 4] = min{E[5, 3]+1, E[4, 4]+1, <u>E[4, 3]+α</u>}

  = min{(2+1), (1+1), <u>(1+0)</u>}

  = 1

- (sto,sto) -> (ston,ston)

S \ T	ε	s	t	o	n
	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

5.3 Edit Distance

- 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

5.3 Edit Distance

- Time Complexity
  - O(mn), where m and n are the length of two given strings
  - 2 comparison for each element
  - Derive the frequency for all entries

5.3 Edit Distance

- Application
  - Any problem using symbolic data
  - Clustering
  - Bioinformatics: DNA similarity measure
  - Spell Checker
  - Optical Character Recognition
  - **Natural Language Translation**
  - …

Knapsack Problem

5.4 Knapsack Problem (not fractional)

- Goal
  - Put all products to a knapsack with some **constraints** on product attribtues
  - Example
    - Weight constraint for a knapsack: C
    - Attribute for product i:
      weight w_i, price v_i
    - **Additional constraint:**
      the quantity of each product is 1

5.4 Knapsack Problem (not fractional)

- Strong Dependency between Decisions
- Example
  - Selected A, B, C, D, E → over-weighted
  - Which product should be **excluded**?
  - All products are responsible for the over-weighting
- Possible Solution
  - Omit a product with the least value/weight rate
  - Omit a product with the heaviest product
  - We do not know which one will be better until we check **all possible cases** by omitting one product

5.4 Knapsack Problem (not fractional)

- Dynamic Programming for Knapsack Problem
- Check points
  - How to divide this problem?
  - How to design recursive derivation of solutions for larger subproblems from smaller subproblems?
  - What is **the implicit order** to check all possibility?
  - How can we be sure that the derived results are correct and do not need to be changed later?

5.4 Knapsack Problem (not fractional)

- **How to divdie this problem?**
- Final results: the price of products in the knapsack given weight limit

- Split by the weight sum of products
- Split by the price sum of products
- Split by the product ID what we select at the last step
- ....

- Any other approach?

5.4 Knapsack Problem (not fractional)

- **How to derive results of larger subproblems?**
  Weight limit case
- Assume that we found the best product combination with a given weight limit w
- How to derive the best solution for weight limit w+1?
- What if there is a product with weight w+1 and
  the largest price-weight rate?
- We should remove all product and put the new product
- We can not derive the next subproblem results by solving smaller problems

5.4 Knapsack Problem (not fractional)

- **How to derive results of larger subproblems?**
  More fine splitting
- We need to **distinguish** which products are selected
- Then, we can separately memorize the case of putting the product with the best rate and w+1
- Compared to the previous approach,
  - Memorize N times more results

5.4 Knapsack Problem (not fractional)

- **How to derive results of larger subproblems?**
  More fine splitting – Can we derived?
- p_i: the best price of combinations using a product i,
  given weight limit w
- Increase the weight limit w → w + 1
- Is there any case in that we can not find the best combinations from the p_is?
- We can not be sure that the best results of p_is with w limit guarantee derivation of the best results for p_is with w+1 limit

5.4 Knapsack Problem (not fractional)

- **How to derive results of larger subproblems?**
  More clear relation of deriving the results of larger subproblems
- Assume that there is the best combination for p_i with w+1 limit
- We can exclude one product from the combination
- Then, the remaining combinations should have **the best price** with the limit of their weight sum
- If not, we can find a better combination than the assumed best combination

5.4 Knapsack Problem (not fractional)

- **How to derive results of larger subproblems?**
  - A problem decomposition
- We want to derive V[w+1] from V_i for all i
- $f(i,w) = V[w+1-w_i] + v_i$
- $V[w+1] = \max(f(i,w) \text{ for all } i)$
- Is this correct?
- (the case introducing a new best product is correctly detected)

5.4 Knapsack Problem (not fractional)

- **How to derive results of larger subproblems?**
  - A problem decomposition: limit
- Assume $V[w+1] = V[w-w_i] + v_i$ for a specific i
- $V[w+1]$ uses v_i
- $V[w-w_i]$ should not use v_i
- However, the best case for $V[w-w_i]$ may use v_i

5.4 Knapsack Problem (not fractional)

- **How to derive results of larger subproblems?**
  - A problem decomposition: negative example
- $V_1 = 10$, $W_1 = 10$
- $V_2 = 4$, $W_2 = 8$
- W + 1 = 20
- V[W+1] = V[20] = max(V[20**-10**]**+10**, V[20**-8**]**+4**)
  = max(V[10]+10, V[12]+4)
- V[10] = 10, V[12] = 10
- But, can we actually generate V[W+1]?
  V[10] uses product 1
  V[20] uses product 1 ***again*** to obtain the best price

5.4 Knapsack Problem (not fractional)

- **Implicit Order**
- We do not know which combination is the best with the given limit w+1
- Best result with (w+1 – w_i) limit + the price of v_i
  - w_i is the weight of a product i
- We do not know which product should be omitted
  > evaluate the results for all i
- We are not sure about the correct path, but we know the correct path will be in one of the previously evaluated entries

5.4 Knapsack Problem (not fractional)

- **Example**
- Given condition

물건	1	2	3	4
무게 (kg)	5	4	6	3
가치(만원)	10	40	30	50

5.4 Knapsack Problem (not fractional)

- **Initialization**
- 0 weight limit – 0 price
- No product – 0 price

C=10

배낭 용량 → w=			0	1	2	3	4	5	6	7	8	9	10
무게	가치	물건	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										

5.4 Knapsack Problem (not fractional)

- **Meaning of each row**
- Using only the row id product? -> we miss some combinations
- Using all products? -> some combinations are **duplicated** and **ambiguous** which one to select

C=10

배낭 용량 → w=			0	1	2	3	4	5	6	7	8	9	10
무게	가치	물건	0	0	0	0	0	0	0	0	0	0	0
5	10	1	0	0	0	0	0	10					
4	40	2	0	0	0	0	40	40					
6	30	3	0	0	0	0	40	40					
3	50	4	0	0	0	50	50	50					

5.4 Knapsack Problem (not fractional)

- **Too many combinations to check**
- We need clear combinations omitting a product for derivation of a next entry
- -> Too many possible combinations

C=10

배낭 용량 → w=			0	1	2	3	4	5	6	7	8	9	10
무게	가치	물건	0	0	0	0	0	0	0	0	0	0	0
5	10	1	0	0	0	0	0	10					
4	40	2	0	0	0	0	40	40					
6	30	3	0	0	0	0	40	40					
3	50	4	0	0	0	50	50	50					

5.4 Knapsack Problem (not fractional)

- **Compact implicit order**
- Meaning of each entry at row i :
  the best solutions using product 0 ~ i
- We need to check only **the entries of upper row**

C=10

배낭 용량 → w=			0	1	2	3	4	5	6	7	8	9	10
무게	가치	물건	0	0	0	0	0	0	0	0	0	0	0
5	10	1	0	0	0	0	0	10					
4	40	2	0	0	0	0	40	40					
6	30	3	$\boxed{0}$	0	0	0	40	40					
3	50	4	0	0	0	$\boxed{50}$	50	50					

5.4 Knapsack Problem (not fractional)

- **A Doubt**
- When we evaluate the best price for an entry at the third row
- 4th product is missed from the best price evaluation of an entry at the 3rd row

C=10

		배낭 용량 → w=	0	1	2	3	4	5	6	7	8	9	10
무게	가치	물건	0	0	0	0	0	0	0	0	0	0	0
5	10	1	0	0	0	0	0	10					
4	40	2	0	0	0	0	40	40					
6	30	3	0	0	0	0	$\boxed{40}$	40					
3	50	4	$\boxed{0}$	0	0	50	50	50					

5.4 Knapsack Problem (not fractional)

- **Correctness**
- What we want: best price when we **use all products**
- Can we sure that we saw all possible paths to derive each entry from the restricted cases?

C=10

배낭 용량 → W=			0	1	2	3	4	5	6	7	8	9	10
무게	가치	물건	0	0	0	0	0	0	0	0	0	0	0
5	10	1	0	0	0	0	0	10					
4	40	2	0	0	0	0	40	40					
6	30	3	0	0	0	0	40	40					
3	50	4	0	0	0	50	50	50					$\boxed{}$

5.4 Knapsack Problem (not fractional)

- **Correctness**
- p(i, j): the price at (i,j) entry, i is product id, j is the weight limit
- p(k, c)
  $= p\,(k\text{-}1, c - w_k) + v_k$ if the best solution use product k
  $= p\,(k\text{-}1, c)$ otherwise

5.4 Knapsack Problem (not fractional)

- **Correctness – proof by contradiction**
- Assumption:
  - the results derived from p(k,c) function is not the best.
  - All p(k-1, c-m) for all product m are the best prices.
- > there exists a larger price p' *than* p(k,c)
- If it uses product k, $p(k-1, c - w_k) < p' - v_k$
  $p(k-1, c - w_k)$ is not the best -> contradiction
- Otherwise, p(k-1,c) < p' -> contradiction
- If the condition is given, then we can assure that (k,c) entry always guarantees the best solution

5.4 Knapsack Problem (not fractional)

- **Correctness – in propositional logic**
- P: all p(i,j) where i <= k and j <= c are the best price
  except i=k and j=c case
  Q: p(k-1, c-m) through the implicit order is the best price
  !Q: there exists the best price greater than the result of Q

  - Assume that P is true (P)
  - Our argument: Q is true (Q)
  - Assume Q is false (!Q)
  - Then, we can prove that !Q -> !P (by the deriving a combination to provide better price for p(i,j))
  - P ^ !P -> contradiction

5.4 Knapsack Problem (not fractional)

- **Implicit Order**
- Fill up all entries by evaluating the best price when we allow only one product

C=10

배낭 용량→ W=			0	1	2	3	4	5	6	7	8	9	10
무게	가치	물건	$\boxed{0}$	0	0	0	0	$\boxed{0}$	0	0	0	0	0
5	10	i =1	0	**0**	**0**	**0**	**0**	$\boxed{\mathbf{10}}$	**10**	**10**	**10**	**10**	**10**
4	40	2	0										
6	30	3	0										
3	50	4	0										

5.4 Knapsack Problem (not fractional)

- **Implicit Order**
- Fill up all entries by evaluating the best price when we allow only one product

C=10

배낭 용량 → W=			0	1	2	3	4	5	6	7	8	9	10
무게	가치	물건	0	0	0	0	0	0	0	0	0	0	0
5	10	1	$\boxed{0}$	0	0	0	$\boxed{0}$	10	10	10	10	10	10
4	40	i = 2	0	**0**	**0**	**0**	$\boxed{\mathbf{40}}$	**40**	**40**	**40**	**40**	**50**	**50**
6	30	3	0										
3	50	4	0										

5.4 Knapsack Problem (not fractional)

- **Implicit Order**
- Fill up all entries by evaluating the best price when we allow only one product

C=10

배낭 용량 → w=			0	1	2	3	4	5	6	7	8	9	10
무게	가치	물건	0	0	0	0	0	0	0	0	0	0	0
5	10	1	0	$\boxed{0}$	0	0	0	$\boxed{10}$	10	10	10	10	10
4	40	i = 2	0	**0**	**0**	**0**	**40**	$\boxed{\mathbf{40}}$	**40**	**40**	**40**	**50**	**50**
6	30	3	0										
3	50	4	0										

5.4 Knapsack Problem (not fractional)

- **Implicit Order**
- Fill up all entries by evaluating the best price when we allow only one product

C=10

배낭 용량 → w=			0	1	2	3	4	5	6	7	8	9	10
무게	가치	물건	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										

5.4 Knapsack Problem (not fractional)

- **Implicit Order**
- Fill up all entries by evaluating the best price when we allow only one product

C

배낭 용량 → w=			0	1	2	3	4	5	6	7	8	9	10
물건	가치	무게	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	$\boxed{90}$

5.4 Knapsack Problem (not fractional)

Maximum price when we allow to use product 1 ~ (i-1) with weight limit w

Assume product i is used in the maximum price case

K[i-1,w]

0

All paths to derive the best price for (I,w)

K[i,w]

$K[i-1,w-w_i]$ + The price of product i, v_i

1

Maximum price when we allow to use product 1 ~ (i-1) with weight limit (w-w_i)

5.4 Knapsack Problem (not fractional)

- Pseudo code and time complexity

5.4 Knapsack Problem (not fractional)

Knapsack(C, n)

Input: weight limit C, n products, weight and price w_i and v_i for ith product, , i = 1, 2, ⋯, n

Output: K[n,C]

1. for i = 0 to n K[i,0]=0 // knapsack weight limit = 0 (initial case)
2. for w = 0 to C K[0,w]=0 // No product is allowed to use initially
3. for i = 1 to n {
4.   for w = 1 to C { // w eight limit
5.   if (w_i > w) // additional restriction, if w_i is larger than weight, we can skip
6.   K[i, w] = K[i-1, w] // maximization
7.   else // find the maximum price between all possible candidates
8.   **K[i, w] = max{K[i-1, w], K[i-1, w-w_i]+v_i}**

    }

  }

9. return K[n, C]

Coin Change

5.5 Coin Change

- 대부분의 경우 그리디 알고리즘으로 해결되나, 해결 못하는 경우도 있다.
- DP 알고리즘은 모든 동전 거스름돈 문제에 대하여 항상 최적해를 찾는다.

5.5 Coin Change

- 부분문제
- 문제에 주어진 요소들
    - 동전의 종류, d_1, d_2, ⋯, d_k, 단, $d_1 > d_2 > \cdots > d_k = 1$
    - 거스름돈 n원

배낭 문제의 DP 알고리즘에서 배낭의 용량을 1kg씩 증가시켜가며 문제를 해결

- 1원씩 증가시켜가며 문제를 해결하자.
- 거스름돈을 배낭의 용량과 같이 생각하자.

5.5 Coin Change

- 부분문제
- 1차원 배열 C
  - C[1] = 1원을 거슬러 받을 때 사용되는 최소의 동전 수
  - C[2] = 2원을 거슬러 받을 때 사용되는 최소의 동전 수

    ⋮
  - C[n] = n원을 거슬러 받을 때 사용되는 최소의 동전 수

5.5 Coin Change

- C[j]를 계산하는데 어떤 부분 문제가 필요할까?
- 500원 동전이 필요하면 (j-500)원의 해, 즉, C[j-500]에다가 500원 동전 1개 추가
- 100원 동전이 필요하면 (j-100)원의 해, 즉, C[j-100]에다가 100원 동전 1개 추가
- 50원 동전이 필요하면 (j-50)원의 해, 즉, C[j-50]에다가 50원 동전 1개 추가
- 10원 동전이 필요하면 (j-10)원의 해, 즉, C[j-10]에다가 10원 동전 1개 추가
- 1원 동전이 필요하면 (j-1)원의 해, 즉, C[j-1]에다가 1원 동전 1개 추가

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

5.5 Coin Change

DPCoinChange(n, K)

Input: 거스름돈 n원, k개의 동전의 액면, $d_1 > d_2 > \cdots > d_k = 1$

Output: C[n]

```
for i = 1 to n C[i]=∞
C[0]=0
for j = 1 to n  // j는 1원부터 증가하는 (임시) 거스름돈 액수
    for i = 1 to k
        if (d_i ≤ j) and (C[j-d_i]+1<C[j])
            C[j] = C[j-d_i] + 1
return C[n]
```

5.5 Coin Change

- d_1=16, d_2=10, d_3=5, d_4=1이고, 거스름돈 n=20일 때

수행 과정

- Line 1~2: 배열 C 초기화

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

거스름돈 1원~4원까지

- 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

거스름돈 5원

1개

j	0	1	2	3	4	5
	0	1	2	3	4	∞

5개

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

거스름돈 6, 7, 8, 9원

2개

3개

4개

5개

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

거스름돈 10원

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

1개

거스름돈 20원

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

2개

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

수행 결과

- 거스름돈 20원에 대한 최종해 = C[20]=2

- 그리디 알고리즘은 20원에 대해 16원 동전을 먼저 '욕심 내어' 취하고, 4원이 남게 되어, 1원 4개를 취하여, 모두 5개의 동전이 해라고 답한다.

그리디 알고리즘의 해

동적 계획 알고리즘의 해

시간 복잡도

- O(nk)
- 거스름돈 j가 1원~n원까지 변하며, 각각의 j에 대해서 최대 모든 동전 (d_1, d_2, ⋯, d_k)을 (즉, k개를) 1번씩 고려하기 때문

Summary

- 동적 계획(Dynamic Programming) 알고리즘은 최적화 문제를 해결하는 알고리즘으로서 입력 크기가 작은 부분 문제들을 모두 해결한 후에 그 해들을 이용하여 보다 큰 크기의 부분 문제들을 해결하여, 주어진 입력의 문제를 해결하는 알고리즘

- DP 알고리즘에는 부분 문제들 사이에 함축적 순서가 존재한다.

Summary

- 모든 쌍 최단 경로(All Pairs Shortest Paths) 문제를 위한 Floyd-Warshall 알고리즘은 $O(n^3)$ 시간에 해를 찾는다.
  - 경유 가능한 점들을 점 1로부터 시작하여, 점 1과 2, 그 다음엔 점 1, 2, 3으로 하나씩 추가하여, 마지막에는 점 1에서 점 n까지의 모든 점을 경유 가능한 점들로 고려하면서, 모든 쌍의 최단 경로의 거리를 계산한다.

- 연속 행렬 곱셈(Chained Matrix Multiplications) 문제를 위한 $O(n^3)$ DP 알고리즘은 이웃하는 행렬들끼리 곱하는 모든 부분 문제들을 해결하여 최적해를 찾는다.

Summary

- 편집 거리(Edit Distance) 문제를 위한 DP 알고리즘은 E[i, j]를 3개의 부분 문제 E[i, j-1], E[i-1, j], E[i-1, j-1]만을 참조하여 계산한다. 시간 복잡도는 O(mn)이다. 단, m과 n은 두 스트링의 길이이다.

- 배낭(Knapsack) 문제를 위한 DP 알고리즘은 부분 문제 K[i, w]를 물건 1~i까지만 고려하고, (임시) 배낭의 용량이 w일 때의 최대 가치로 정의하여, i를 1 ~ 물건 수인 n까지, w를 1 ~ 배낭 용량 C까지 변화시키며 해를 찾는다. 시간 복잡도는 O(nC)이다.

Summary

- 동전 거스름돈(Coin Change)문제는 1원씩 증가시켜 문제를 해결한다. 시간 복잡도는 O(nk)이다. 단, n은 거스름돈 액수이고, k는 동전 종류의 수이다.

- DP 알고리즘은 부분 문제들 사이의 관계를 빠짐없이 고려하여 문제를 해결한다.

- DP 알고리즘은 최적 부분 구조 (optimal substructure) 또는 최적성 원칙 (principle of optimality) 특성을 가지고 있다.
  - 문제의 최적해 속에 부분 문제의 최적해가 포함되어 있다.
  - 그리디 알고리즘도 같은 속성을 가진다.