11주차 3차시 동적 계획 알고리즘의 이해 Ⅲ

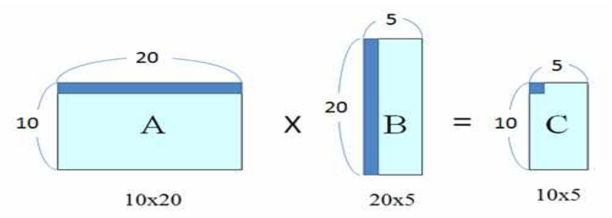
【학습목표】

1. 연속 행렬 곱셈 알고리즘을 이해할 수 있다.
2. 동전 거스름돈 알고리즘을 이해할 수 있다.

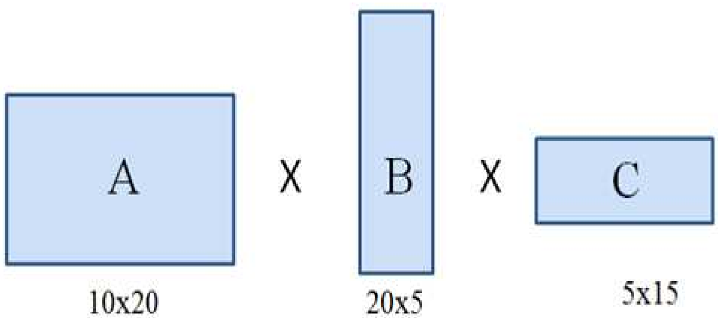
학습내용1 : 연속 행렬 곱셈

1. 연속 행렬 곱셈 (Chained Matrix Multiplications)

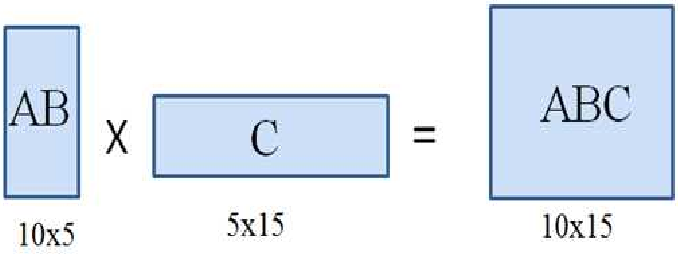
연속 행렬 곱셈 (Chained Matrix Multiplications) 문제는 연속된 행렬들의 곱셈에 필요한 원소간의 최소 곱셈 횟수를 찾는 문제이다.
10x20 행렬 A와 20x5 행렬 B를 곱하는데 원소간의 곱셈 횟수는 10x20x5 = 1,000이다. 그리고 두 행렬을 곱한 결과 행렬 C는 10x5이다.



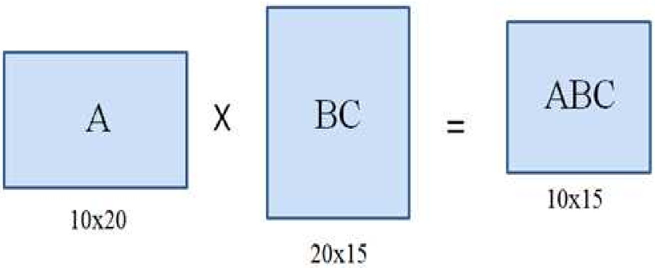
- 행렬 C의 1개의 원소를 위해서 행렬 A의 1개의 행에 있는 20개 원소와 행렬 B의 1개의 열에 있는 20개의 원소를 각각 곱한 값을 더해야 하므로 20회의 곱셈이 필요하다.
- 3개의 행렬을 곱해야 하는 경우
- 연속된 행렬의 곱셈에는 결합 법칙이 허용된다.
- 즉, AxBxC = (AxB)xC = Ax(BxC)이다.
- 다음과 같이 행렬 A가 10x20, 행렬 B가 20x5, 행렬 C가 5x15라고 하자.



• 먼저 AxB를 계산한 후에 그 결과 행렬과 행렬 C를 곱하기 위한 원소간의 곱셈 횟수를 세어 보면, AxB를 계산하는데 10x20x5 = 1,000번의 곱셈이 필요하고, 그 결과 행렬의 크기가 10x5이므로, 이에 행렬 C를 곱하는데 10x5x15 = 750번의 곱셈이 필요하다.
• 총 1,000+750 = 1,750회의 원소의 곱셈이 필요하다



• 이번엔 BxC를 먼저 계산한 후에 행렬 A를 그 결과 행렬과 곱하면, BxC를 계산하는데 20x5x15 = 1,500번의 곱셈이 필요하고, 그 결과 20x15 행렬이 만들어지므로, 이를 행렬 A와 곱하는데 10x20x15 = 3,000번의 곱셈이 필요하다.
• 따라서 총 1,500+3,000 = 4,500회의 곱셈이 필요하다.



• 동일한 결과를 얻음에도 불구하고 원소간의 곱셈 횟수가 4,500-1,700 = 2,800이나 차이 난다.
• 따라서 연속된 행렬을 곱하는데 필요한 원소간의 곱셈 횟수를 최소화시키기 위해서는 적절한 행렬의 곱셈 순서를 찾아야 한다.
• 주어진 행렬의 순서를 지켜서 이웃하는 행렬끼리 반드시 곱해야 하기 때문
• 예를 들어, AxBxCxDxE를 계산하려는데, B를 건너뛰어서 AxC를 수행한다든지, B와 C를 건너뛰어서 AxD를 먼저 수행할 수 없다.
• 따라서 다음과 같은 부분문제들이 만들어진다.

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

• 맨 윗줄의 가장 작은 부분문제들은 입력으로 주어진 각각의 행렬 그 자체이고, 크기가 2인 부분문제는 2개의 이웃하는 행렬의 곱셈으로 이루어진 4개이다.
• 여기서 눈여겨보아야 할 것은 부분 문제들이 겹쳐져 있다는 것이다.
• 만일 AxB, CxD와 같이 부분문제가 서로 안 겹치면 AxBxCxD를 계산할 때 BxC에 대한 해가 없으므로 새로이 계산해야 한다.
• 이러한 경우를 대비하여 이웃하여 서로 겹치는 부분 문제들의 해도 미리 구하여 놓는다.

부분 문제 크기		부분 문제 개수
3	AxBxC BxCxD CxDxE	3개
4	AxBxCxD BxCxDxE	2개
5	A x B x C x D x E	1개

- 그 다음은 크기가 3인 부분문제가 3개이고, 이들 역시 서로 이웃하는 부분문제들끼리 겹치어 있음을 확인할 수 있다.
- 다음 줄에는 크기가 4인 부분문제가 2개이고, 마지막에는 1개의 문제로서 입력으로 주어진 문제이다.

* 알고리즘

MatrixChain

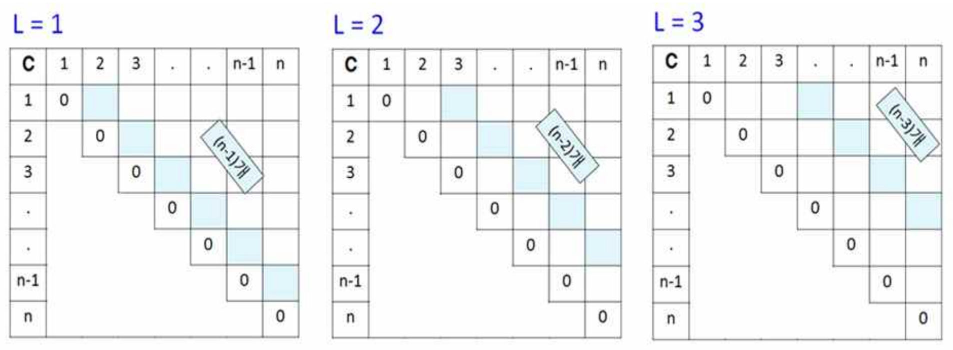
입력: 연속된 행렬 $A_1xA_2x \cdots xA_n$, 단, A_1은 d_0xd_1, A_2는 d_1xd_2, …, A_n은 d_{n-1} x d_n이다.

출력: 입력의 행렬 곱셈에 필요한 원소의 최소 곱셈 횟수

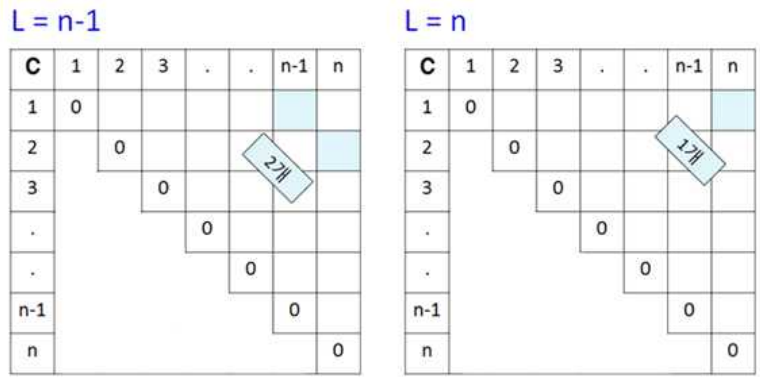
```
1. for i = 1 to n
2.         C[i,i] = 0
3. for L = 1 to n-1 { // L은 부분 문제의 크기를 조절하는 인덱스이다.
4.       for i = 1 to n-L {
5.              j = i + L
6.             C[i,j] = ∞
7.             for k = i to j-1 {
8.                    temp = C[i,k] + C[k+1,j] + di-1dkdj
9.                    if (temp < C[i,j])
10.                          C[i,j] = temp
                         }
                   }
     }
11. return C[1,n]
```

- Line 1~2: 배열의 대각선 원소들, 즉, C[1,1], C[2,2], …, C[n,n]을 0으로 각각 초기화시킨다.
- 그 의미는 행렬 A_1xA_1, A_2xA_2, …, A_nxA_n을 각각 계산하는데 필요한 원소간의 곱셈 횟수가 0이란 뜻이다.
- 즉, 실제로는 아무런 계산도 필요 없다는 뜻이다. 이렇게 초기화하는 이유는 C[i,i]가 가장 작은 부분 문제의 해이기 때문이다.

- Line 3의 for-루프의 L은 1~(n-1)까지 변하는데, L은 부분문제의 크기를 2~n까지 조절하기 위한 변수이다. 즉, 이를 위해 line 4의 for-루프의 i가 1~(n-L)까지 변한다.
- L=1일 때, i는 1~(n-1)까지 변하므로, 크기가 2인 부분문제의 수가 (n-1)개이다.
- L=2일 때, i는 1~(n-2)까지 변하므로, 크기가 3인 부분문제의 수가 (n-2)개이다.
- L=3일 때, i는 1~(n-3)까지 변하므로, 크기가 4인 부분문제의 수가 (n-3)개이다.



- L=n-2일 때, i는 1~n-(n-2)=2까지 변하므로, 크기가 (n-1)인 부분 문제의 수는 2개이다.
- L=n-1일 때, i는 1~n-(n-1)=1까지 변하므로, 크기가 n인 부분 문제의 수는 1개이다.



Line 5에서는 j=i+L인데, 이는 행렬 A_i x…x A_j에 대한 원소간의 최소 곱셈 횟수, 즉, C[i,j]를 계산하기 위한 것이다.
따라서
L=1일 때,
i=1: j=1+1=2 (A1xA2를 계산하기 위하여),
i=2: j=2+1=3 (A2xA3을 계산하기 위하여),
i=3: j=3+1=4 (A3xA4를 계산하기 위하여),
…
i=n-L=n-1: j=(n-1)+1=n (A_{n-1}xA_n을 계산하기 위하여)
크기 2인 부분문제의 수: (n-1)개

- L=2일 때,

i=1: j=1+2=3 ($A_1xA_2xA_3$을 계산하기 위하여),
i=2: j=2+2=4 ($A_2xA_3xA_4$를 계산하기 위하여),
i=3: j=3+2=5 ($A_3xA_4xA_5$를 계산하기 위하여),
…
i=n-L=n-2: j=(n-2)+2=n ($A_{n-2}xA_{n-1}xA_n$을 계산하기 위해)
크기 3인 부분문제의 수: (n-2)개

- L=3일 때, $A_1xA_2xA_3xA_4$, $A_2xA_3xA_4xA_5$, …,

$A_{n-3}xA_{n-2}xA_{n-1}xA_n$을 계산한다.
크기가 4인 부분문제의 수가 (n-3)개
…

- L=n-2일 때, 2개의 부분문제 $A_1xA_2x...xA_{n-1}$, $A_2xA_3x...xA_n$을 계산한다.
- L=n-1일 때, i=1이면 j=1+(n-1)=n이고, 주어진 문제 $A_1xA_2x...xA_n$을 계산한다

$\boxed{A_1xA_2}$ $\boxed{A_2xA_3}$ $\boxed{A_3xA_4}$ … $\boxed{A_{n-2}xA_{n-1}}$ $\boxed{A_{n-1}xA_n}$ n-1개

$\boxed{A_1xA_2xA_3}$ $\boxed{A_2xA_3xA_4}$ $\boxed{A_3xA_4xA_5}$ … $\boxed{A_{n-2}xA_{n-1}xA_n}$ n-2개

: :

$\boxed{A_1xA_2xA_3x\cdots xA_{n-1}}$ $\boxed{A_2xA_3xA_4x\cdots xA_n}$ 2개

$\boxed{A_1xA_2xA_3x \ldots xA_{n-1}xA_n}$ 1개

- Line 6에서는 최소 곱셈 횟수를 찾기 위해 C[i,j]=∞로 초기화 시킨다.
- Line 7~10의 for-루프는 k가 i ”(j-1)까지 변하면서 어떤 부분 문제를 먼저 계산하면 곱셈 횟수가 최소인지를 찾아서 최종적으로 C[i,j]에 그 값을 저장한다.

• 즉, k가 $A_i x A_{i+1} x \cdots x A_j$를 2개의 부분문제로 나누어 어떤 경우에 곱셈 횟수가 최소인지를 찾는데, 여기서 부분문제간의 함축적 순서가 존재함을 알 수 있다.

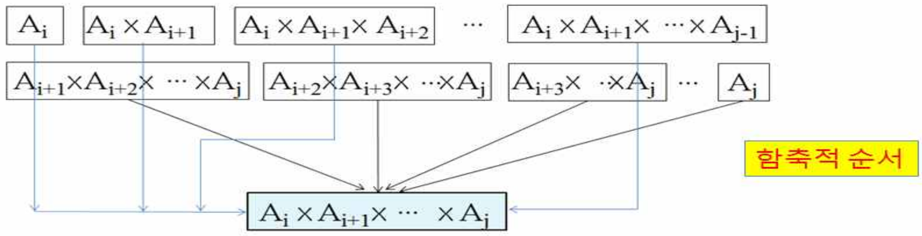
$(A_i) \times (A_{i+1} \times A_{i+2} \times \cdots \times A_j)$ k=i 일때

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

: :

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



• Line 8: 2개의 부분문제로 나뉜 각각의 경우에 대한 곱셈 횟수를 계산한다.
☞ 첫 번째 부분문제의 해는 C[i,k]에 있고,
☞ 두 번째 부분문제의 해는 C[k+1,j]에 있으며,
☞ 2개의 해를 합하고, 이에 $d_{i-1}d_kd_j$를 더한다.
☞ 여기서 $d_{i-1}d_kd_j$를 더하는 이유는 두 부분문제들이 각각 $d_{i-1}xd_k$ 행렬과d_kxd_j 행렬이고,
☞ 두 행렬을 곱하는데 필요한 원소간의 곱셈 횟수가 $d_{i-1}d_kd_j$이기 때문이다.

• 다음은 k값의 변화에 따른 2개의 부분문제에 해당 하는 행렬을 각각 보여주고 있다.

$\underset{d_{i-1}xd_i}{(A_i)} \times \underset{d_ixd_j}{(A_{i+1} \times A_{i+2} \times \cdots \times A_j)}$ k=i 일때

$\underset{d_{i-1}xd_{i+1}}{(A_i \times A_{i+1})} \times \underset{d_{i+1}xd_j}{(A_{i+2} \times A_{i+3} \times \cdots \times A_j)}$ k=i +1 일때

$\underset{d_{i-1}xd_{i+2}}{(A_i \times A_{i+1} \times A_{i+2})} \times \underset{d_{i+2}xd_j}{(A_{i+3} \times \cdots \times A_j)}$ k=i +2 일때

: :

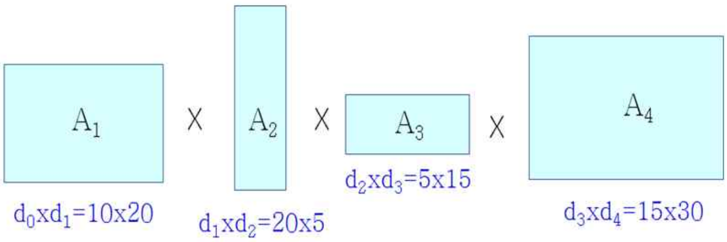
$\underset{d_{i-1}xd_{j-1}}{(A_i \times A_{i+1} \times \cdots \times A_{j-1})} \times \underset{d_{j-1}xd_j}{(A_j)}$ k=j-1 일때

• Line 9~10: line 8에서 계산된 곱셈 횟수가 바로 직전까지 계산되어 있는 C[i,j]보다 작으면 그 값으로 C[i,j]를 갱신하며, k=(j-1)일 때까지 수행되면 최종적으로 가장 작은 값이 C[i,j]에 저장된다.
• Line 11: 주어진 문제의 해가 있는 C[1,n]을 리턴

* Matrixchain 알고리즘의 수행 과정

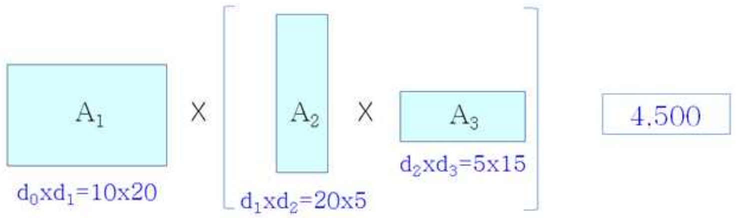
MatrixChain 알고리즘의 수행 과정

- A_1이 10x20, A_2가 20x5, A_3이 5x15, A_4가 15x30이다.

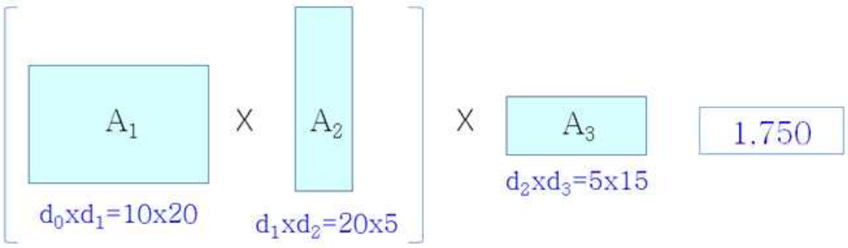


- Line 1~2에서 C[1,1]=C[2,2]=C[3,3]=C[4,4]= 0으로 초기화
- Line 3에서 L이 1~(4-1)=3까지 변하고, 각각의 L값에 대하여, i가 변화하며 C[i,j]를 계산한다.
  - L=1일 때, i는 1~(n-L)=4-1=3까지 변한다.
    - i=1이면 j=1+1=2이므로, $A_1 x A_2$를 위해 line 6에서 C[1,2]=∞로 초기화하고, line 7의 k는 1~(j-1)=2-1=1까지 변하므로 사실 k=1일 때 1번만 수행된다. Line 8에서 temp = C[1,1] + C[2,2] + $d_0 d_1 d_2$ = 0+0+(10x20x5) = 1,000이 되고, line 9에서 현재 C[1,2]=∞가 temp보다 크므로, C[1,2]=1,000이 된다.
    - i=2이면 j=2+1=3이므로, $A_2 x A_3$을 위해 line 6에서 C[2,3]=∞로 초기화하고, line 7의 k는 2 ~(j-1)=3-1=2까지 변하므로 k=2일 때 역시 1번만 수행된다. Line 8에서 temp = C[2,2] + C[3,3] + $d_1 d_2 d_3$ = 0+0+(20x5x15) = 1,500이 되고, line 9에서 현재 C[2,3]=∞가 temp보다 크므로, C[2,3]=1,500이 된다.
    - i=3이면 $A_3 x A_4$에 대해 C[3,4] = 2,250이 된다.

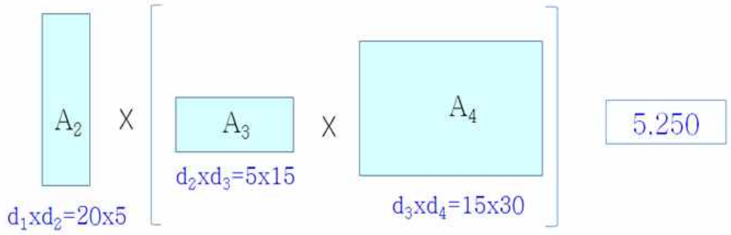
- L=2일 때 i는 1 ~(n-L)=4-2=2까지 변한다.
  - i=1이면 j=1+2=3이므로, $A_1 x A_2 x A_3$을 계산하기 위해 line 6에서 C[1,3]=∞로 초기화하고, line 7의 k는 1 ~(j-1)=3-1=2까지 변하므로, k=1과 k=2일 때 2번 수행된다.
    - k=1일 때, line 8에서 temp = C[i,k] + C[k+1,j] + $d_{i-1}d_kd_j$ = C[1,1]+C[2,3]+$d_0d_1d_3$ = 0+1,500+(10x20x15) = 4,500이 되고, line 9에서 현재 C[1,3]=∞이고 temp보다 크므로, C[1,3]=4,500이 된다.



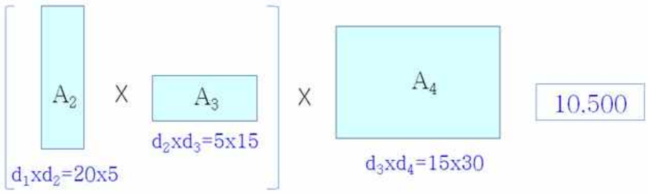
- k=2일 때, line 8에서 temp = C[i,k] + C[k+1,j] + $d_{i-1}d_kd_j$ = C[1,2]+C[3,3]+$d_0d_2d_3$ = 1,000+0+(10x5x15) = 1,750이 되고, line 9에서 현재 C[1,3] = 4,500인데 temp보다 크므로, C[1,3]=1,750으로 갱신된다.



- i=2이면 j=2+2=4이므로, $A_2 x A_3 x A_4$를 계산하기 위해 line 6에서 C[2,4]=∞로 초기화하고, line 7의 k는 2 ~(j-1)=3까지 변하므로, k=2와 k=3일 때 2번 수행된다.
  - k=2일 때, line 8에서 temp = C[i,k] + C[k+1,j] + $d_{i-1}d_kd_j$ = C[2,2]+C[3,4]+$d_1d_2d_4$ = 0+2,250+(20x5x30) = 5,250이 되고, line 9에서 현재 C[2,4]=∞가 temp보다 크므로, C[2,4] = 5,250이 된다.

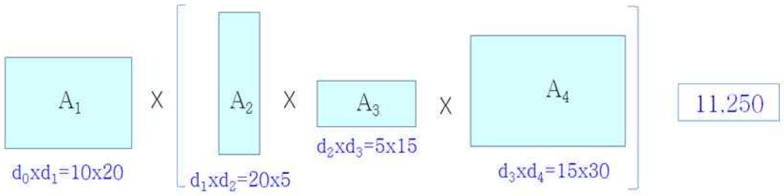


- k=3일 때, line 8에서 temp = C[i,k] + C[k+1,j] + $d_{i-1}d_kd_j$ = C[2,3]+C[4,4]+$d_1d_3d_4$ = 1,500+0+(20x15x30) = 10,500이 된다.

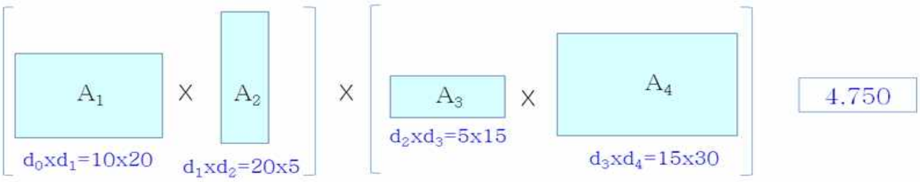


그러나 line 9에서 현재 C[2,4]=5,250이 temp보다 작으므로, 그대로 C[2,4]=5,250이다.

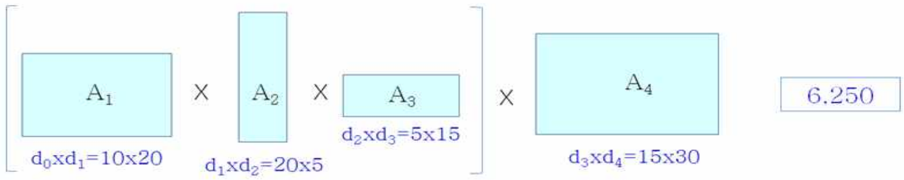
- L=3일 때 i는 1 ~ (n-L)=4-3=1까지 이므로 i=1일 때 만 수행된다.
  - i=1이면 j=1+3=4이므로, $A_1xA_2xA_3xA_4$를 계산하기 위해 line 6에서 C[1,4]=∞로 초기화하고, line 7의 k는 1~(j-1) =4-1=3까지 변하므로, k=1, k=2, k=3일 때 각각 수행된다.
    - k=1일 때, line 8에서 temp = C[i,k] + C[k+1,j] + $d_{i-1}d_kd_j$ = C[1,1]+C[2,4]+$d_0d_1d_4$ = 0+5,250+(10x20x30) = 11,250이 되고, line 9에서 현재 C[1,4]=∞가 temp보다 크므로, C[1,4]=11,250이 된다.



- k=2일 때, line 8에서 temp = C[i,k] + C[k+1,j] + $d_{i-1}d_kd_j$ = C[1,2]+C[3,4]+$d_0d_2d_4$ = 1,000+2,250+(10x5x30) = 4,750이 되고, line 9에서 현재 C[1,4]=11,250이 temp보다 크므로, C[1,4] = 4,750으로 갱신된다.



- k=3일 때, line 8에서 temp = C[i,k] + C[k+1,j] + $d_{i-1}d_kd_j$ = C[1,3]+C[4,4]+$d_0d_3d_4$ = 1,750+0+(10x15x30) = 6,250이 된다.



- 그러나 line 9에서 현재 C[1,4] = 4,750이 temp보다 작으므로, 그대로 C[1,4]=4,750이다.

- 따라서 최종해는 4,750번이다. 먼저 A_1xA_2를 계산하고, 그 다음엔 A_3xA_4를 계산하여, 각각의 결과를 곱하는 것이 가장 효율적이다. 다음은 알고리즘이 수행된 후의 배열 C이다.

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

2. 시간복잡도

- MatrixChain 알고리즘의 시간복잡도는 2차원 배열 C만 보더라도 쉽게 알 수 있다.
- 배열 C가 nxn이고, 원소의 수는 n^2인데, 거의 1/2 정도의 원소들의 값을 계산해야 한다.
- 그런데 하나의 원소, 즉, C[i,j]를 계산하기 위해서는 k-루프가 최대 (n-1)번 수행되어야 한다.
- 따라서 MatrixChain 알고리즘의 시간복잡도는 $O(n^2)xO(n) = O(n^3)$이다.

학습내용2 : 동전 거스름돈

1. 동전 거스름돈의 개요

- 잔돈을 동전으로 거슬러 받아야 할 때, 누구나 적은 수의 동전으로 거스름돈을 받고 싶어 한다.
- 대부분의 경우 그리디 알고리즘으로 해결되나, 해결 못하는 경우도 있다.
- 동적 계획 알고리즘은 모든 동전 거스름돈 문제에 대하여 항상 최적해를 찾는다.

- 동적 계획 알고리즘을 고안하기 위해서는 부분 문제를 찾아내야 한다.
- 동전 거스름돈 문제에 주어진 문제 요소들을 생각해보자. 정해진 동전의 종류, d_1, d_2, …, d_k가 있고, 거스름돈 n원이 있다. 단, $d_1 > d_2 > \cdots > d_k = 1$ 이라고 하자.
- 예를 들어, 우리나라의 동전 종류는 5개로서, d_1 = 500, d_2 =100, d_3 = 50, d_4 = 10, d_5 = 1이다.
- 그런데 배낭 문제의 동적 계획 알고리즘을 살펴보면, 배낭의 용량을 1kg씩 증가시켜 문제를 해결한다.

- 여기서 힌트를 얻어서, 동전 거스름돈 문제도 1원씩 증가시켜 문제를 해결한다.
- 즉, 거스름돈을 배낭의 용량으로 생각하고, 동전을 물건으로 생각
- 부분 문제들의 해를 아래와 같이 1차원 배열 C에 저장하자.
- 1원을 거슬러 받을 때 사용되는 최소의 동전 수 C[1]
- 2원을 거슬러 받을 때 사용되는 최소의 동전 수 C[2]

…
- j원을 거슬러 받을 때 사용되는 최소의 동전 수 C[j]

…
- n원을 거슬러 받을 때 사용되는 최소의 동전 수 C[n]

- 부분문제들 사이의 '함축적인 순서', 즉, 한 부분문제의 해를 구하는데 어떤 부분 문제의 해가 필요한지를 살펴보자.
- 구체적으로 C[j]를 구하는데 어떤 부분문제가 필요할까
- j원을 거슬러 받을 때 최소의 동전 수를 다음의 동전들 (d_1=500, d_2=100, d_3=50, d_4=10, d_5=1)로 생각해 보자.
- 500원짜리 동전이 거스름돈 j원에 필요하면 (j-500)원의 해, 즉, C[j-500] = C[j-d_1]에다가 500원짜리 동전 1개를 추가한다.
- 100원짜리 동전이 거스름돈 j원에 필요하면 (j-100)원의 해, 즉, C[j-100] = C[j-d_2]에다가 100원짜리 동전 1개를 추가한다.

- 50원짜리 동전이 거스름돈 j원에 필요하면 (j-50)원의 해, 즉, C[j-50] = C[j-d_3]에다가 50원짜리 동전 1개를 추가한다.
- 10원짜리 동전이 거스름돈 j원에 필요하면 (j-10)원의 해, 즉, C[j-10] = C[j-d_4]에다가 10원짜리 동전 1개를 추가한다.
- 1원짜리 동전이 거스름돈 j원에 필요하면 (j-1)원의 해, 즉, C[j-1] = C[j-d_5]에다가 1원짜리 동전 1개를 추가한다.
- 위의 5가지 중에서 당연히 가장 작은 값을 C[j]로 정해야 한다. 따라서 C[j]는 아래와 같이 정의된다.

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

위의 식에서는 i가 1~k까지 각각 변하면서, 즉, d_1, d_2, d_3, …, d_k 각각에 대하여 해당 동전을 거스름돈에 포함시킬 경우의 동전 수를 고려하여 최소값을 C[j]로 정한다.

* 알고리즘

알고리즘

DPCoinChange
입력: 거스름돈 n원, k개의 동전의 액면, $d_1 > d_2 > \cdots > d_k = 1$
출력: C[n]

```
1. for i = 1 to n C[i]=∞
2. C[0]=0
3. for j = 1 to n { // j는 1원부터 증가하는 (임시) 거스름돈 액수
                     이고, j=n이면 입력에 주어진 거스름돈이 된다.
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: 배열 C의 각 원소를 ∞로 초기화 한다. 이는 문제에서 거슬러 받는 최소 동전 수를 구하기 때문이다.
- Line 2: C[0]=0으로 초기화한다. 이는 line 5에서 C[j-d_i]의 인덱스인 j에서 d_i를 뺀 값이 0이 되는 경우, 즉, C[0]이 되는 경우를 위해서이다.
- Line 3~6의 for-루프에서는 (임시) 거스름돈 액수 j를 1원부터 1원씩 증가시키며, line 4~6에서 $\min_{1 \le i \le k}$ {C[j-d_i] + 1}을 C[j]로 정한다.
- 이를 위해 line 4~6의 for-루프에서는 가장 큰 액면의 동전부터 1원짜리 동전까지 차례로 동전을 고려해보고, 그 중에서 가장 적은 동전 수를 C[j]로 결정한다. 단, 거스름돈 액수인 j원보다 크지 않은 동전에 대해서만 고려한다.

- 다음은 d_1=16, d_2=10, d_3=5, d_4=1이고, 거스름돈 n=20일 때 DPCoinChange 알고리즘의 수행되는 과정이다.

- 다음의 표에서 대각선 원소가 C[j]이고, 파란 음영으로 표시된 원소들이 C[j]를 계산하는데 필요한 부분문제들이다.

- Line 1~2에서 배열 C를 아래와 같이 초기화시킨다.

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

- 거스름돈 j원은 1원~4원까지는 1원짜리 동전 (d_4=1) 밖에 고려할 동전이 없으므로, 각 j에서 1을 뺀, 즉, (j-1)의 해인 (C[j-1]+1)이 C[j]가 된다.
- 따라서 i=4 (1원짜리 동전)일 때의 line 5의 if-조건인 (1≤j)가 '참'이고, (C[j-1]+1<∞)도 '참'이 되어 각각 아래와 같이 C[j]가 결정된다.
- C[1] = C[j-1]+1 = C[1-1]+1 = C[0]+1 = 0+1 = 1

j	0	1
	0	∞

⇨

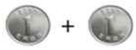
j	0	1
	0	1

- 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=5이면 임시 거스름돈이 5원일 때,
  - i=3 (5원짜리 동전)에 대해서, line 5의 if-조건인 (5≤5)가 '참'이고, (C[5-5]+1 < C[5]) = (C[0]+1 <∞) = (0+1 <∞)이므로 '참'이 되어 'C[j] = C[j-d_i]+1'가 수행된다. 따라서 C[5] = C[5-5]+1 = C[0]+1 = 0+1 = 1이 된다. 즉, C[5]=1이다.

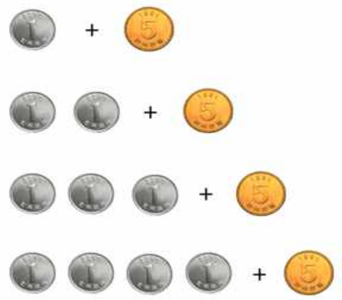
j	0	1	2	3	4	5
	0	1	2	3	4	∞

⇨

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

  - i=4 (1원짜리 동전)일 때는 line 5의 if-조건인 (d_5 ≤ 5)는 '참'이나 (C[j-d_i]+1 < C[j]) = (C[5-1] +1 < C[4]) = (C[4] +1 < C[5]) = (4+1 < 1) = (5 < 1)가 '거짓'이 되어 C[5]는 변하지 않고 그대로 1을 유지한다.
  - 즉, 1원짜리 동전으로 거스름돈을 주려 하면 오히려 동전 수가 늘어나기 때문이다.

- j=6, 7, 8, 9이고, i=3 (5원짜리 동전)일 때, 각각 아래와 같이 수행된다.
- 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
- 단, i=4 (1원짜리 동전)일 때에는 line 5의 if-조건의 (C[j-d_i]+1 < C[j])= (C[j-1]+1 < C[j])이 각각의 j에 대해서 (1+1) < 2, (2+1) < 3, (3+1) < 4, (4+1) < 5로서 '거짓'이 되어 C[j]는 변경되지 않는다. 사실은 i=3일 때와 동일하므로 각각 갱신 안 된다.



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=10이면 거스름돈이 10원이면,
  - i=2 (10원짜리 동전)일 때, line 5의 if-조건인 (d_i≤j)=(10≤10)은 '참'이고, (C[j-d_i]+1<C[j]) = (C[10-10]+1<C[10]) = (C[0]+1< C[10]) = (0+1<∞)이 '참'이 되어 'C[j]=C[j-d_i]+1'이 수행된다. 따라서 C[10] = C[10-10]+1 = C[0]+1 = 0+1 = 1이다. 즉, C[10]=1이다.

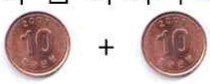
  - i=3 (5원짜리 동전)일 때, line 5의 if-조건인 (d_i≤j) = (5<10)는 '참'이나, (C[10-5]+1<C[10]) = (C[5]+1<C[10]) = (1+1<1)이 '거짓'이 되어서 C[10]은 변하지 않는다. 즉, 5원짜리 2개보다는 10원짜리 1개 낫다.

- i=4 (1원짜리 동전)일 때는 line 5의 if-조건인 (d_i≤j) = (1<10)는 '참'이나, (C[j-d_i]+1<C[j]) = (C[10-1]+1<C[10]) = (C[9]+1< C[10]) = (5+1<1) = (6<1)이 '거짓'이므로 C[10]이 변하지 않고 그대로 1을 유지한다.

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=20일 때,
  - i=1 (16원짜리 동전)일 때, C[20] = C[j-16]+1 = C[4]+1 = 4 +1 = 5

  - i=2 (10원짜리 동전)일 때, line 5의 if-조건에서 C[j-10]+1 = C[10]+1 = 1+1 = 2이므로 현재 C[20]의 값인 5보다 작다. 따라서 if-조건이 '참'이 되어 C[20]=2가 된다.



  - i=3 (5원짜리 동전)일 때에는 line 5의 if-조건의 (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)이 '거짓'이 되어 C[20]이 변경되지 않는다.

  - i=4 (1원짜리 동전)일 때에도 line 5의 if-조건이 (C[20-1]+1<2) = (5<2)이 '거짓'이므로 C[20]이 변경되지 않는다

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

- 따라서 거스름돈 20원에 대한 최종해는 C[20]=2개의 동전이다. 4.1절의 그리디 알고리즘은 20원에 대해 16원짜리 동전을 먼저 '욕심내어' 취하고, 4원이 남게 되어, 1원짜리 4개를 취하여, 모두 5개의 동전이 해라고 답한다.

그리디 알고리즘의 해

동적 계획 알고리즘의 해

2. 시간복잡도

- DPCoinChange 알고리즘의 시간복잡도는 O(nk)인데 이는 거스름돈 j가 1원n원까지 변하며, 각각의 j에 대해서 최악의 경우 모든 동전 (d_1, d_2, …, d_k)을 (즉, k개를) 1번씩 고려하기 때문이다.

【학습정리】

1. 연속 행렬 곱셈
연속 행렬 곱셈 (Chained Matrix Multiplications) 문제는 연속된 행렬들의 곱셈에 필요한 원소간의 최소 곱셈 횟수를 찾는 문제이다.
10x20 행렬 A와 20x5 행렬 B를 곱하는데 원소간의 곱셈 횟수는 10x20x5 = 1,000이다. 그리고 두 행렬을 곱한 결과 행렬 C는 10x5이다.

2. 동전 거스름돈
- 잔돈을 동전으로 거슬러 받아야 할 때, 누구나 적은 수의 동전으로 거스름돈을 받고 싶어 한다.
- 대부분의 경우 그리디 알고리즘으로 해결되나, 해결 못하는 경우도 있다.
- 동적 계획 알고리즘은 모든 동전 거스름돈 문제에 대하여 항상 최적해를 찾는다.

- 동적 계획 알고리즘을 고안하기 위해서는 부분 문제를 찾아내야 한다.
- 동전 거스름돈 문제에 주어진 문제 요소들을 생각해보자. 정해진 동전의 종류, d_1, d_2, …, d_k가 있고, 거스름돈 n원이 있다. 단, $d_1 > d_2 > \cdots > d_k=1$ 이라고 하자.
- 예를 들어, 우리나라의 동전 종류는 5개로서, d_1 = 500, d_2 =100, d_3 = 50, d_4 = 10, d_5 = 1이다.
- 그런데 배낭 문제의 동적 계획 알고리즘을 살펴보면, 배낭의 용량을 1kg씩 증가시켜 문제를 해결한다.

- 여기서 힌트를 얻어서, 동전 거스름돈 문제도 1원씩 증가시켜 문제를 해결한다.
- 즉, 거스름돈을 배낭의 용량으로 생각하고, 동전을 물건으로 생각
- 부분 문제들의 해를 아래와 같이 1차원 배열 C에 저장하자.
- 1원을 거슬러 받을 때 사용되는 최소의 동전 수 C[1]

- 2원을 거슬러 받을 때 사용되는 최소의 동전 수 C[2]

  …

- j원을 거슬러 받을 때 사용되는 최소의 동전 수 C[j]

  …

- n원을 거슬러 받을 때 사용되는 최소의 동전 수 C[n]

- 부분문제들 사이의 '함축적인 순서', 즉, 한 부분문제의 해를 구하는데 어떤 부분 문제의 해가 필요한지를 살펴보자.
- 구체적으로 C[j]를 구하는데 어떤 부분문제가 필요할까
- j원을 거슬러 받을 때 최소의 동전 수를 다음의 동전들 (d_1=500, d_2=100, d_3=50, d_4=10, d_5=1)로 생각해 보자.
- 500원짜리 동전이 거스름돈 j원에 필요하면 (j-500)원의 해, 즉, C[j-500] = C[j-d_1]에다가 500원짜리 동전 1개를 추가한다.
- 100원짜리 동전이 거스름돈 j원에 필요하면 (j-100)원의 해, 즉, C[j-100] = C[j-d_2]에다가 100원짜리 동전 1개를 추가한다.

- 50원짜리 동전이 거스름돈 j원에 필요하면 (j-50)원의 해, 즉, C[j-50] = C[j-d_3]에다가 50원짜리 동전 1개를 추가한다.
- 10원짜리 동전이 거스름돈 j원에 필요하면 (j-10)원의 해, 즉, C[j-10] = C[j-d_4]에다가 10원짜리 동전 1개를 추가한다.
- 1원짜리 동전이 거스름돈 j원에 필요하면 (j-1)원의 해, 즉, C[j-1] = C[j-d_5]에다가 1원짜리 동전 1개를 추가한다.
- 위의 5가지 중에서 당연히 가장 작은 값을 C[j]로 정해야 한다. 따라서 C[j]는 아래와 같이 정의된다.

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

위의 식에서는 i가 1~k까지 각각 변하면서, 즉, d_1, d_2, d_3, …, d_k 각각에 대하여 해당 동전을 거스름돈에 포함시킬 경우의 동전 수를 고려하여 최소값을 C[j]로 정한다.