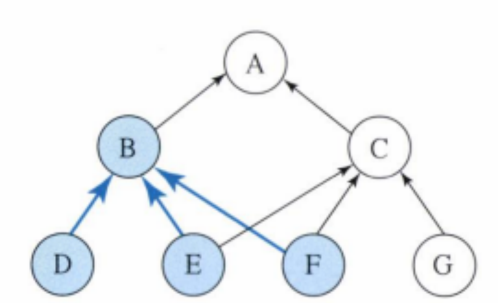
🍏

ch.05 동적 계획 알고리즘

동적 계획 알고리즘

그리디 알고리즘과 같이 최적화 문제를 해결하는 알고리즘

→ 입력 크기가 작은 부분 문제들을 모두 해결한 후, 이 해들을 이용하여 큰 크기의 부분문제 해결



동적 계획 알고리즘

+) 문제마다 사용하는 sub routine이 다름
+) 부분문제들을 중복해서 사용

→ 의존적 관계
↳ 잘 보이지 않음
⇒ 함축적 순서

- 적용 요건
    - 최적 부분 구조(optimal substructure)
        - 큰 문제의 최적 솔루션에 작은 문제 최적 솔루션 포함
    - 재귀호출 시 중복(overlapping recursive calls)
        - 재귀적 해법으로 풀면 같은 문제에 대한 재귀호출이 심하게 중복

5.1 모든 쌍 최단 경로(All pairs shortest paths)

각 쌍의 점 사이의 최단 경로를 찾는 문제

→ 각 점을 시작점으로 정하여 다익스트라 알고리즘 수행

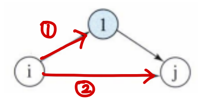
→ time complexity : (n-1) * O(n^2) = O(n^3) (n은 점의 수)

⇓

- 플로이드 알고리즘

- 모든 쌍 최단 경로를 찾는 동적 계획 알고리즘
- time complexity : O(n^3)
- but 다익스트라 사용보다 더 효율적임

1. 가장 작은 부분 문제 생각하기 → 3개의 점이 있는 경우
    a. i→j까지의 최단 경로 : i에서 j로 직접 or 1을 경유하는 것 중 짧은 것



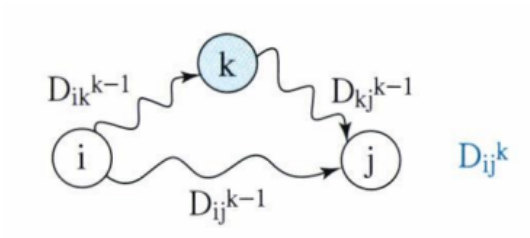
↳ 뭐가 더 짧음?

2. 1과 같이 경유 가능한 점들을 하나씩 추가하여 n개까지의 경유 가능한 점 고려

D^k_{ij} = 점 {1, 2, ⋯, k}만을 경유 가능한 점들로 고려하여, 점 i로부터 점 j까지의 모든 경로 중에서 가장 짧은 경로의 거리

- 모든 점들은 반드시 경유하는 경로는 아님.
- k=0 → Dij : 입력으로 주어지는 간선 (i,j)의 가중치 (경유 X , 직빵으로 감)

3. 모든 쌍의 최단 경로 계산 가능
    a. 둘 중에 짧은 경로가 Dijk



4. 모든 점을 경유 가능한 점들로 고려된 모든 쌍 i와 j의 최단 경로의 거리를 찾는 방식

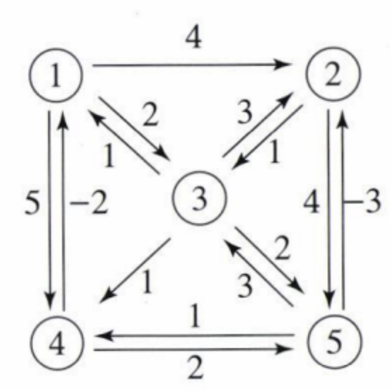
↳ 출발, 경유, 시작이 같은 경우 X

```
AllPairsShortest()

입력 : 2차원 배열 D, 단 D[i,j]=간선 (i,j)의 가중치
       만일 간선 (i,j)이 존재하지 않으면 D[i,j] = ∞
       모든 i에 대하여 D[i,i] = 0
//입력 그래프에는 가중치 합이 음수가 되는 사이클이 없어야 함 -> 있으면 거리가 감소
출력 : 모든 쌍의 최단 경로의 거리를 저장한 2차원 배열 D

for k=1 to n
  for i=1 to n //(i != k)
    for j=1 to n //(j!=k , j!=i)
    D[i,j] = min{D[i,k] + D[k,j]. D[i.j]} //최단 거리 갱신
```

경유 ← for k / 출발 ← for i / 시작 ← for j

↳ 미리 계산되어 있어야 할 부분문제

example



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

① k=1

i=2

j=3

D[2,3] = min{D[2,1]+D[1,3], D[2,3]} = 1

∞ 2 1

j=4

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

∞ 5 ∞

⋮

⑤ k=5

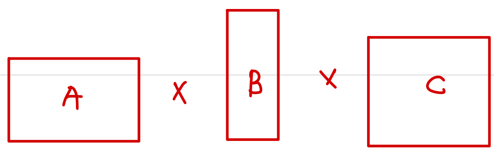
time complexity

i,j,k에 대하여 for문 3번 → O(n^3)

각 계산 → O(1)

⇒ O(n^3)

5.2 연속 행렬 곱셈



⇒ 순서 어떻게?

연속된 행렬들의 곱셈에 필요한 원소 간의 최소 곱셈 횟수를 찾음 → 최적화

⇒ 동일한 행렬들의 곱셈이더라도 순서에 따라 곱셈 횟수 ↑

1. 결합 법칙 성립
   - A * B * C = (A*B) *C* = A* (B*C)
2. 부분문제 → 순서를 지켜서 이웃하는 행렬의 곱

↳ col, row 맞춰야하기 때문!

부분문제 크기						부분문제 개수
1	A	B	C	D	E	5개
2	A×B	B×C	C×D	D×E		4개

3. 부분문제들이 겹쳐져 있음 → 겹친 것의 해도 구하기

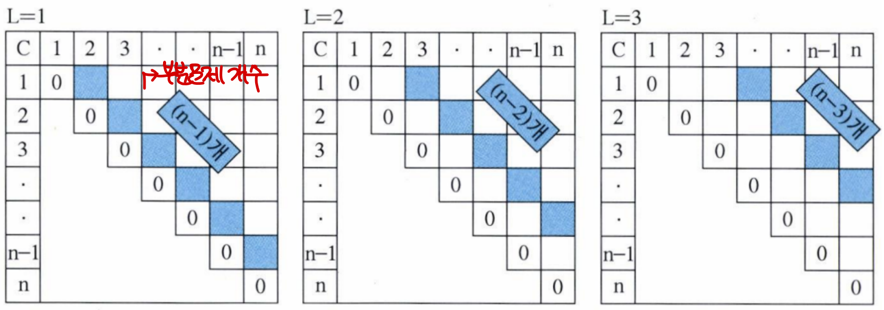
부분문제 크기				부분문제 개수
3	A×B×C	B×C×D	C×D×E	3개
4	A×B×C×D	B×C×D×E		2개
5	A×B×C×D×E			1개

→이전 부분문제들의 해로 현재 부분문제 해 구함

4. 최소 곱셈 횟수 찾기

```
MatrixChain()
//입력 : 연속된 행렬 A(1) * A(2) * ... * A(n)
//       A(1) = d(0)*d(1), ... A(n) = d(n-1)*d(n)
//출력 : 입력의 행렬 곱셈에 필요한 원소 간의 최소 곱셈 횟수

for i = 1 to n
  C[i, i] = 0; → 자기자신것을 곱한다 ⇒ 곱셈 횟수 0 !

for L = 1 to n-1 //L = 부분문제의 크기 -1
  for i = 1 to n-L{ //i = 부분문제가 시작하는 행렬

    // 부분문제의 크기가 L+1일 경우 n-L개의 부분문제 존재

    j = i+L //j=부분문제가 끝나는 행렬
    C[i,j] = ∞
    for k = i to j-1{ //k = 부분문제를 2개로 나누는 자리
      temp = C[i,k] + C[k+1, j] + d(i-1)d(k)d(j)
      //    첫번째         두번째          마지막 연산 곱셈 횟수
      if(temp < C[i,j]) //더 작은 것 선택    ↳ ex. ① A×B  ② C×D → ①×②의 횟수
        C[i,j] = temp
      }
    }
  }
return C[1,n]
```

부분문제크기

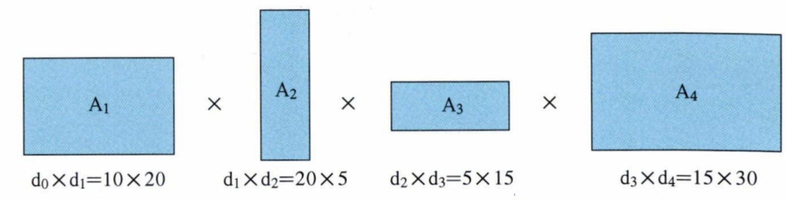


부분문제   부분문제 개수

$A_1 \times A_2$ $A_2 \times A_3$ $A_3 \times A_4$ ⋯ $A_{n-2} \times A_{n-1}$ $A_{n-1} \times A_n$	n − 1개
$A_1 \times A_2 \times A_3$ $A_2 \times A_3 \times A_4$ $A_3 \times A_4 \times A_5$ ⋯ $A_{n-2} \times A_{n-1} \times A_n$	n − 2개
⋮	⋮
$A_1 \times A_2 \times A_3 \times \cdots \times A_{n-1}$ $A_2 \times A_3 \times A_4 \times \cdots \times A_n$	2개
$A_1 \times A_2 \times A_3 \times \cdots \times A_{n-1} \times A_n$	1개

example

n=4



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

L=1
i= 1/3
j= 2
k=1/1
$C[1,1]+C[2,2]+d_0d_1d_2 = 10\times20\times5 = 1000$

i=2
j=3
k=2
$C[2,2]+C[3,3]+d_1d_2d_3 = 20\times5\times15 = 1500$

i=3
j=4
k=3
$C[3,3]+C[4,4]+d_2d_3d_4 = 5\times15\times30 = 2250$

⋮

all pairs minumum VS chained matrix multiplication

- all pairs
    - 직전 부분문제만 현재 부분문제 해 구할 때 필요
        - 메모리 효용이 더 높음
- chained matrix
    - 모든 부분문제를 저장해놓고 계속 업데이트 필요
        - 메모리 bad
- **vs** divide conquer
    - 자기와 근접인 부분문제와만 관련 있음
    - 다이나믹 프로그래밍은 관련 x

time complexity

- 배열 크기 : n*n
- k-loop : (n-1)번

⇒ O(n^2) * O(n) = O(n^3)

+) 재귀적 호출 버전 → 엄청난 중복 호출 발생 ⇒ O(n^2)

```
rMatrixChain(i, j)
//행렬곱 Ai…Aj를 구하는 최소 비용 구하기
{ if (i = j) then return 0; //행렬이 하나뿐인 경우의 비용은 0
  min ← ∞;
  for k ← i to j-1 {
    q ← rMatrixChain(i, k) + rMatrixChain(k+1, j) + pi-1pkpj;
    if (q < min) then min ← q;
  }
  return min;
}
```

5.4 배낭 문제(Knapsack)

n개의 물건 → 각 i라고 했을 때 무게 wi와 가치 vi

배낭 용량 C

⇒ 배낭에 담을 수 있는 물건의 최대 가치를 찾는 문제

⇒ 0-1 배낭 문제 : 담기지 않은 경우 → 0, 담긴 경우 → 1

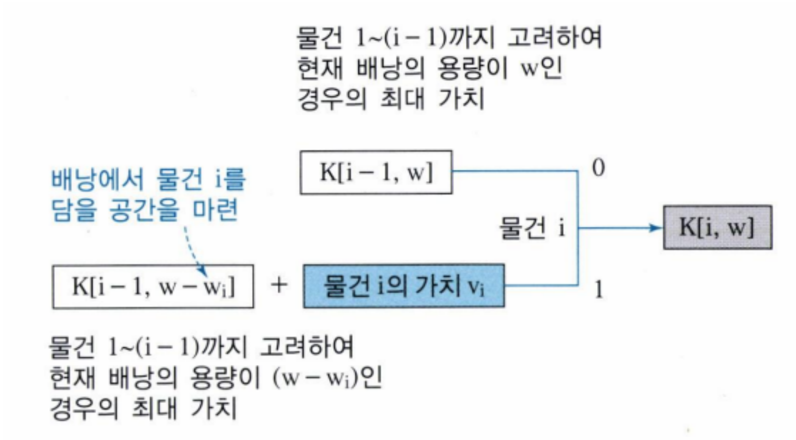
1. 부분 문제 정의 → 물건, 물건의 무게
    a. 물건의 가치의 합에 근거하여 담을지 말지 결정
    b. 배낭 용량의 초과 여부 검사

2. 0부터 용량 C까지 1씩 증가하여 가치 커지는지 판단
    a. C가 너무 크면 알고리즘 효용 떨어짐 → 제한적인 크기만 수행 가능

3. K[i, w] = 물건 1~i까지만 고려하고, (임시) 배낭 용량이 w일 때의 최대 가치
    a. (단 i = 1, 2, … n / w = 1, 2, 3 … C)
    b. 2가지 경우 중 더 큰 값 선택 → 가치가 더 큰 값 선택
        i. 물건 i를 배낭에 담지 않는 경우 → K[i,w] = K[i-1, w]
        ii. 물건 i를 배낭에 담는 경우 → K[i-1, w-wi] + vi

⇒ 최적해 : K[n,C]

```
Knapsack()

입력 : 배낭의 용량 C, n개의 물건과 각 물건 i의 무게 wi, 가치 vi
출력 : K[n, C]

//초기화
for i=0 to n //배낭의 용량 = 0
  K[i,0] = o  → 0일 때는 담을 수 없기에 최대가치가 0
for w=0 to C //물건 0 -> 어떤 물건도 배낭에 담기 위해 고려 x
  K[0,w] = 0  → 담을 물건이 없기에 가치가 0

for i=1 to n{
  for w=1 to C{ //w는 배낭의 임시 용량 -> w=C가 되어 배낭의 용량
    if(wi > w) //물건 i의 무게가 임시 배낭의 용량 초과하면
      K[i,w] = K[i-1, w]
    else //물건 i를 배낭에 담지 않을 경우와 담을 경우를 고려
      K[i,w] = max{K[i-1, w], K[i-1, w-wi] + vi}  → 지금 담을거
                                  원래 있던거 제외
  }
}

return K[n, C]
```



example

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

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	2	0				40	40	. .	-	- . .	50	50
6	30	3	0				40	40	40	40	40	50	70
3	50	4	0			50	50	50	50	90	90	90	90

$i=1$

$w=1$

$w_1 > 1 \rightarrow K[1,1] = K[0,1]$

$w=2$

$w_1 > 2 \rightarrow K[1,2] = K[0,2]$

물건 w_1

⋮

$w=5$

$w_1 = 5 \rightarrow K[1,5] = \max\{K[0,5], K[0,0]+10\} = 10$

물건 w_1

⋮

$w=10$

$i=2 \quad w=1$

⋮

$w=3$

$w=4 \rightarrow K[2,4] = \max\{K[1,4], K[1,0]+40\} = 40$

물건: $w_1 + w_2$

⋮

$w=9 \rightarrow K[2,9] = \max\{K[1,9], K[1,5]+40\} = 50$

time complexity

- 부분문제 : O(1) 시간
- 부분문제 개수 : n * C

⇒ O(1) * n * C = O(nC)

5.5 동전 거스름 문제

그리디 알고리즘으로 해결 못 했던 것 → 동적계획으로 해결

⇒ knapsack 알고리즘 참고

→ 거스름돈 : 배낭 용량 / 동전 : 물건 → 1원씩 증가하면서 바꿀 수 있다면 더 높은 단위로!

1. 동전 : d1, d2 … dk (d1 > d2 > … > dk = 1)
2. 거스름돈 : n원
3. C[1], C[2], … C[n] : 거슬러 받을 때 사용되는 최소의 동전 수
4. C[j]를 구하는데 필요한 부분문제 → 계산하려는 가격 − 필요한 동전 단위
    a. d1 동전이 필요하면 C[j-d1]에다가 +1
    b. d2 동전이 필요하면 C[j-d2]에다가 +1
    c. … dk 동전이 필요하면 C[j-dk]에다가 +1

    ⇒ 이 중 가장 작은 값이 C[j] (j보다 큰 동전은 고려하지 x)

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

```
DPCoinChange()

입력 : 거스름돈 n원, k개의 동전의 액면, d1 > d2 > ... > dk =1
출력 : C[n]

for i=1 to n
  C[i] = ∞
C[0] = 0

for j=1 to n{ //j=1원부터 증가하는 (임시) 거스름돈 액수, j=n이면 입력에 주어진 거스름돈
  for i=1 to k{ //액면이 가장 높은 동전부터 1원 동전까지
    if(di <= j) and C[j-di]+1<C[j])
      C[j] = C[j-di] + 1
    }
  }
return C[n]
}
```

↳가장 큰 액수부터 check

example

d1=16, d2=10, d3=5, d4=1

n=20

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

j=1
  i=1 → x
  i=2 → x
  i=3 → x
  i=4 : d4=1 ≤ 1 and C[0]+1 < C[1]) ⇒ C[1]=1

j=2

i=1

i=2

i=3

i=4 : $d_4 \le 2$ and C[1]+1 < C[2]) ⇒ C[2]=2

j=3

:

i=4 : C[3]=3

j=4

:

i=4 : C[4]=4

j=5 :

i=3 : $d_3 \le 5$ and C[0]+1 < C[5]) ⇒ C[5]=1

i=4 : $d_4 \le 5$ C[4]+1 C[5] > X

j=6

:

i=3 : $d_3 \le 6$ and C[1]+1 < C[6]) ⇒ C[6]=2

⋮

j=20

i=1 : $d_1 \le 20$ and C[4]+1 < C[20] = 5

i=2 : $d_2 \le 20$ and C[10]+1 < C[20] = 2

$\boxed{\therefore 2}$

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
C	0	1	2	3	4	1	2	3	4	5	1	2	3	4	5	2	1

j	17	18	19	20
C	2	3	4	2

time complexity

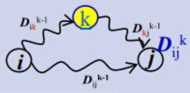
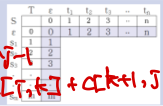
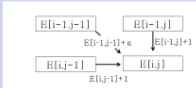
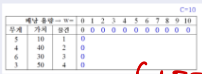
j = 1 ~ n까지 변하며 k를 1번씩 각각 고려

⇒ O(nk)

Dynamic Programming (Summary)

$k=1$ ~ n
$i=1$ ~ n
$j=1$ ~ n

$D[i,j] = \min_{k \neq i \neq j} \{D[i,k]+D[k,j],\ D[i,j]\}$

	Sub-problem	Combining (Implicit order)	Complexity
All pairs shortest path	(table below)		O(n^3)
Chained matrix multi.		$(A_i) \times (A_{i+1} \times A_{i+2} \times \cdots \times A_j)$ k=i 일때 $(A_i \times A_{i+1}) \times (A_{i+2} \times A_{i+3} \times \cdots \times A_j)$ k=i+1일때 $(A_i \times A_{i+1} \times A_{i+2}) \times (A_{i+3} \times \cdots \times A_j)$ k=i+2일때 : $(A_i \times A_{i+1} \times \cdots \times A_{j-1}) \times (A_j)$ k=j-1일때	O(n^3)
~~Edit distance~~			O(mn)
0-1 Knapsack			O(nC)
Coin change	(table below)		O(nk)

All pairs shortest path, sub-problem:

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

Chained matrix multi.:
$L=1$ ~ $n-1$
$i=1$ ~ $n-L$
$j=i+L$
$k=i$ ~ $j-1$
$temp = C[i,k] + C[k+1,j] + d_{i-1}d_k d_j$

0-1 Knapsack:
$K[i,w] = \max_{1 \le i \le n} \{K[i-1,w],\ K[i-1, w-w_i]+v_i\}$ $(w \le C)$

Coin change, sub-problem:

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

Coin change, combining:

	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

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

*연습문제 정리