2025 겨울방학 알고리즘 스터디

동적 계획법(Top-down)

컴퓨터 공학과 20230546 서보경

목차

동적 계획법을 푸는 방법 - 복기

동적 계획법(DP)의 기본 개념

➔ 큰 문제를 작은 부분 문제로 나누고, 그 결과를 저장하여 중복계산을 방지하는 **최적화** 기법.

	탑다운(Top-down)	바텀업(Bottom-up)
방식	재귀 사용	반복문 사용
메모이제이션	필수 (이미 계산된 값 저장)	있음 (테이블 갱신)
호출 방식	**필요**할 때만 계산	작은 문제부터 차례대로 계산
재귀 오버헤드	O(N) 이상의 호출 스택 부담	O(1) (스택 부담 X)
코드 구조	논리적으로 **문제를 나누어** 해결	배열을 기반으로 모든 부분 문제를 순차적으로 계산
유리한 문제 유형	**특정 경우만 계산**하는 경우	모든 부분 문제를 다 풀어야 하는 경우
예시	행렬 곱셈 순서 / LCS / 트리 DP	계단 오르기 문제 / 배낭 문제 / LIS

이번장에서는 탑다운 기반의 동적계획법을 배운다.
탑다운은 점화식의 개념이 상당히 옅다.

최장 공통 수열(LCS)

LCS(Longest Common Subsequence, 최장 공통 부분 수열)문제는 두 수열이 주어졌을 때, 모두의 부분 수열이 되는 수열 중 가장 긴 것을 찾는 문제이다.

예를 들어, ACAYKP와 CAPCAK의 LCS는 ACAK가 된다.

입력

첫째 줄과 둘째 줄에 두 문자열이 주어진다. 문자열은 알파벳 대문자로만 이루어져 있으며, 최대 1000글자로 이루어져 있다.

출력

첫째 줄에 입력으로 주어진 두 문자열의 LCS의 길이를 출력한다.

예제 입력 1 복사	예제 출력 1 복사
ACAYKP CAPCAK	4

정말 많이 알려진 문제인 LCS 문제이다.

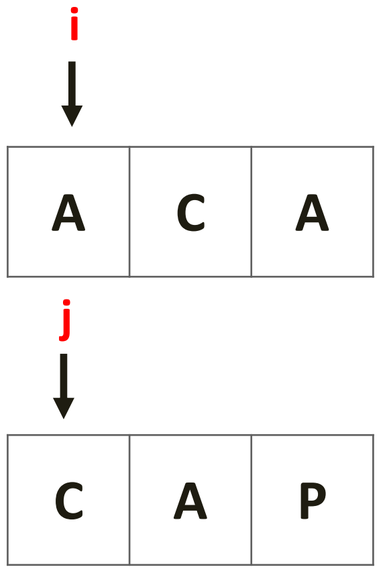
LCS의 핵심은 두 문자열에서 순서는 유지하면서 삭제만 가능한 방식으로 만들 수 있는 가장 긴 공통 부분 수열을 찾는 것이다.

이 문제를 바텀업으로도 당연히 풀 수 있다!

그러나 탑다운 방식이 더 쉽다고 볼 수 있는데,재귀적으로 문제를 분할하면서 필요한 부분만 계산할 수 있기 때문이다.

또한, 불필요한 상태를 건너뛸 수 있어 바텀업보다 직관적이다.

최장 공통 수열(LCS)



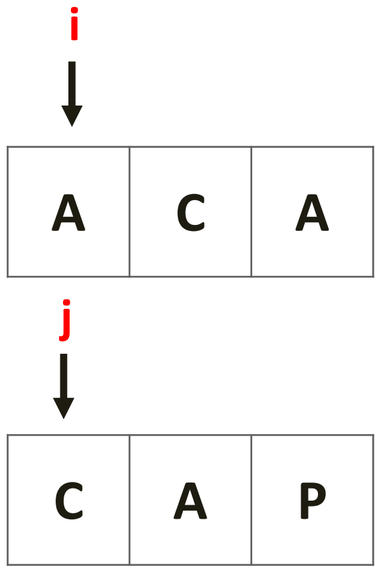
j \ i	0	1	2
0	-1	-1	-1
1	-1	-1	-1
2	-1	-1	-1

탑다운 dp를 쓰기전에 전제 조건이 있다.

프로그래머가 재귀를 쓸 줄 알아야 한다는 것이다. 재귀에 대한 기본적인 지식이 없으면 탑다운 방식은 오히려 독이 될 수 있다.

먼저, 두 문자열의 인덱스를 지정할 포인터를 지정한다.
또한 dp 테이블에서 방문 여부를 확인해야 하기때문에 초기값은 -1로 초기화 한다.

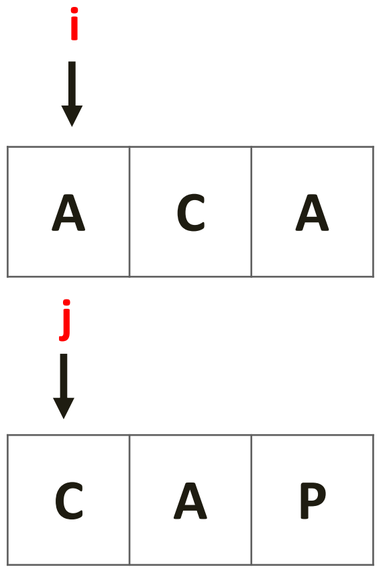
최장 공통 수열(LCS)



j \ i	0	1	2
0	0	-1	-1
1	-1	-1	-1
2	-1	-1	-1

먼저 현재 인덱스를 보자. i와 j가 0이다.
dp[0][0] (인덱스들이 차원이다.) 의 값은 현재 -1이다. 고로 **갱신 되지 않았음**을 의미한다.
먼저 현재 차원의 값에 0을 대입한다.(-1은 방문 여부를 확인하는 더미값)
우리가 여기서 취할 수 있는 방법은 세가지 중 하나이다.
1) 두 인덱스가 가리키는 문자가 같다면 i와 j를 동시에 늘려 추가적인 상태공간 탐색 하기
2) i를 늘려서 다른 상태공간 탐색 하기
3) j를 늘려서 다른 상태공간 탐색 하기

최장 공통 수열(LCS)



j \ i	0	1	2
0	0	-1	-1
1	-1	-1	-1
2	-1	-1	-1

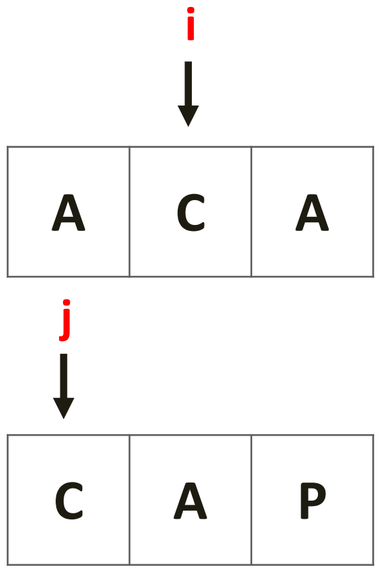
i : 0 / j : 0 / value : 0

아쉽게도, 현재 i의 문자(A)와 j의 문자 (C)가 다르다. 고로 1번 방법은 쓸 수 없다.

2번과 3번 방법을 택해야 하는데 먼저 2번 방법을 선택해 보겠다.

i 포인터를 늘려 다음 상태 공간을 찾아보자. 재귀 함수의 특성상, 상태 공간의 정보가 그대로 스택에 쌓이는데 이를 적극적으로 활용해야 한다.

최장 공통 수열(LCS)



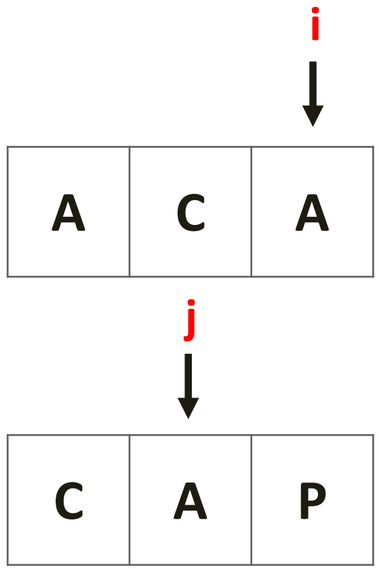
j \ i	0	1	2
0	0	0	-1
1	-1	-1	-1
2	-1	-1	-1

i : 1 / j : 0 / value : 1
i : 0 / j : 0 / value : 0

현재 인덱스를 보자. i는 1, j는 0 이다.
dp[1][0]은 -1이다. 고로 갱신되지 않았으므로 추가적 탐색을 해도 된다.
먼저 현재 차원의 값에 0을 대입한다.
현재 i의 문자(C)와 j의 문자(C)가 같다. 고로 1번 방법을 쓸 수 있다. (2,3번 방법은 추후에 수행)

이 경우에는 공통적으로 겹치는 문자가 있기때문에 value에 1을 대입하고 i와 j 포인터를 동시에 늘린다.

최장 공통 수열(LCS)



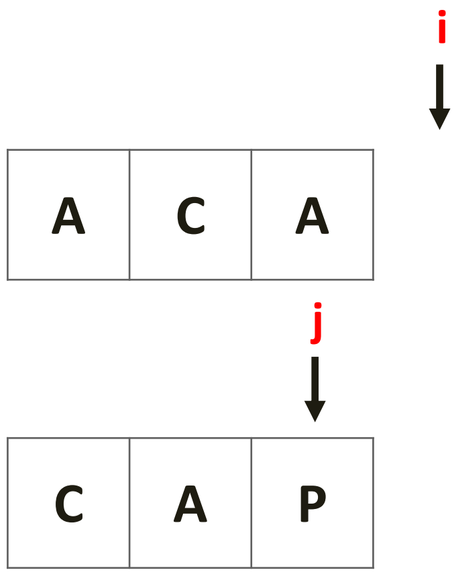
j \ i	0	1	2
0	0	0	-1
1	-1	-1	0
2	-1	-1	-1

i : 2 / j : 1 / value : 1
i : 1 / j : 0 / value : 1
i : 0 / j : 0 / value : 0

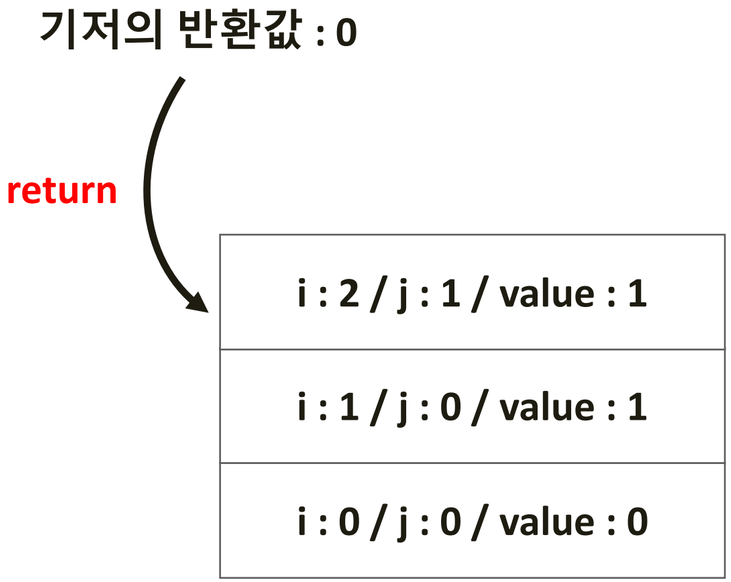
현재 인덱스를 보자. i는 2, j는 1 이다.
dp[2][1]은 -1이다. 고로 갱신되지 않았으므로 추가적 탐색을 해도 된다.
먼저 현재 차원의 값에 0을 대입한다.
현재 i의 문자(A)와 j의 문자(A)가 같다. 고로 1번 방법을 쓸 수 있다.

이 경우에는 공통적으로 겹치는 문자가 있기때문에 value에 1을 대입하고 i와 j 포인터를 동시에 늘린다.

최장 공통 수열(LCS)



j \ i	0	1	2
0	-0	0	-1
1	-1	-1	0
2	-1	-1	-1

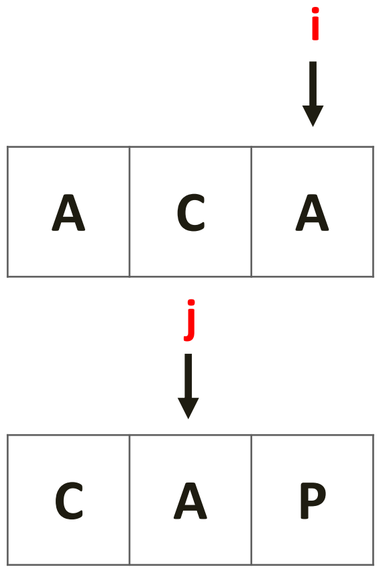


현재 인덱스를 보자. i는 3, j는 2 이다.
j는 문자열 크기 범위안에 있어서 괜찮지만, i는 이미 구간을 넘어 가버렸다.

우리가 구하고자 하는건 '두 문자열의 최장 공통 수열'이기 때문에 한 문자열이 이미 끝났다면(범위를 넘어 갔다면) 더 이상 존재 하지 않는다고 판단한다. (기저 조건이다)

고로, 겹치지 않는다는 표시인 0을 return 하여 이전 상태 공간으로 값을 전달한다.

최장 공통 수열(LCS)



j \ i	0	1	2
0	0	0	-1
1	-1	-1	1
2	-1	-1	-1

현재 사용 된 정보

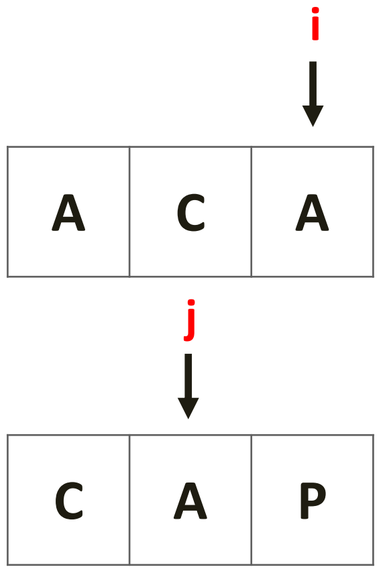
~~i : 2 / j : 1 / value : 1~~
i : 1 / j : 0 / value : 1
i : 0 / j : 0 / value : 0

다시 i 가 2, j가 1인 차원에 돌아왔다. 아까 스택에 올린 정보를 이제 쓸 수 있게 되었다!
(현재 상태에서 끝까지 도달했으므로). 또한 다음 차원(dfs(3,2))의 반환 값은 0이다.

고로 dp[2][1] = max(dp[2][1], value + dfs(3,2))에 의해 dp[2][1]가 1로 갱신 된다.
그 이후, 스택에서 그 상태를 지운다. (사실 재귀함수에 의해 자연적으로 지워 지긴 한다.)

아직 2번과 3번 과정을 하지 않았다.

최장 공통 수열(LCS)



j \ i	0	1	2
0	0	0	-1
1	-1	-1	1
2	-1	-1	-1

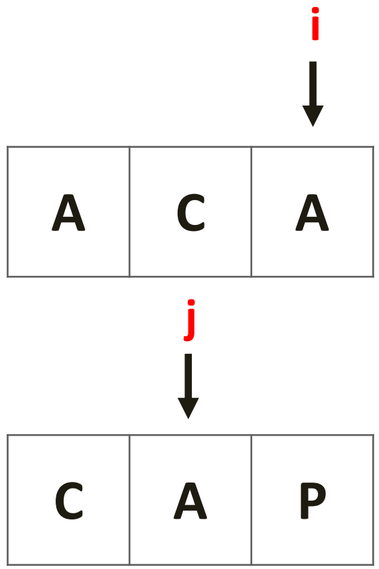
i : 1 / j : 0 / value : 1
i : 0 / j : 0 / value : 0

2번 과정은 솔직히 안 봐도 알 것이다. i를 늘리면 크기를 넘어가게 되고, 그러면 결국 기저조건에 의해 0이 반환되게 된다. 종합적으로 value + dfs(3,1)의 값은 0이 될 것이다.(value가 0이기 때문)

i와 j를 늘린다는 건 현재 문자가 같음과 다름을 따지지 않고 상태 공간을 넓히는 행동이다.

즉, 모든 상태 공간에 대해 한 포인터만 늘려서 value가 0인 행동을 취할 수 있다.
이때 만약 두 포인터가 가리키는 문자가 같다면 value가 1인 특별한 행동을 추가로 취할 수 있다.

최장 공통 수열(LCS)



j \ i	0	1	2
0	0	0	-1
1	-1	-1	1
2	-1	-1	-1

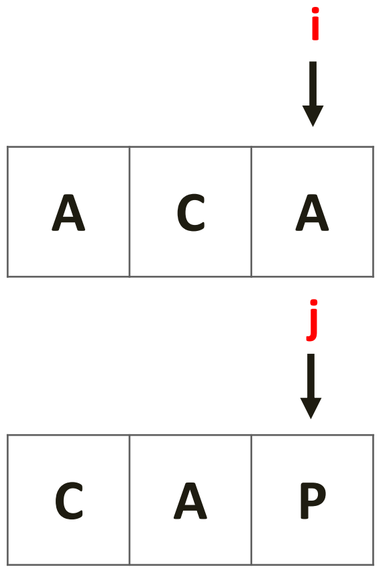
i : 2 / j : 1 / value : 0
i : 1 / j : 0 / value : 1
i : 0 / j : 0 / value : 0

이제 2번 과정이 value가 0인 행동을 하였고, 다음 상태 공간의(기저) 반환 값이 0이기 때문에 0이 된다는 걸 깨달았을 것이다.

그러면 이제 3번 행동을 한번 취해보자.

3번 행동은 value가 0인 행동이다.
j 포인터를 하나 늘리고 현재 상태공간을 스택에 쌓은 후에 다음 상태로 넘어간다.

최장 공통 수열(LCS)



j \ i	0	1	2
0	0	0	-1
1	-1	-1	1
2	-1	-1	0

i : 2 / j : 2 / value : 0
i : 2 / j : 1 / value : 0
i : 1 / j : 0 / value : 1
i : 0 / j : 0 / value : 0

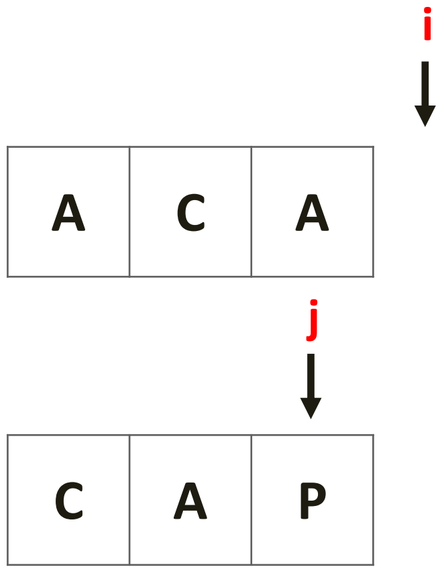
현재 인덱스를 보자. i는 2, j는 2 이다.
dp[2][2]가 -1이므로 갱신 되지 않은 상태이다.

아까 말했듯이 1번 행동을 취하기 위해선 두 포인터가 가리키는 문자가 같아야 한다.

그러나 두 포인터는 서로 다른(A != P)문자를 가리키기 때문에 2번과 3번 행동밖에 취하지 못한다.

먼저 2번 행동을 먼저 취해보자. i포인터를 먼저 늘리고 현재 상태 공간을 스택에 저장한다.

최장 공통 수열(LCS)



j \ i	0	1	2
0	0	0	-1
1	-1	-1	1
2	-1	-1	0

기저의 반환값 : 0

return

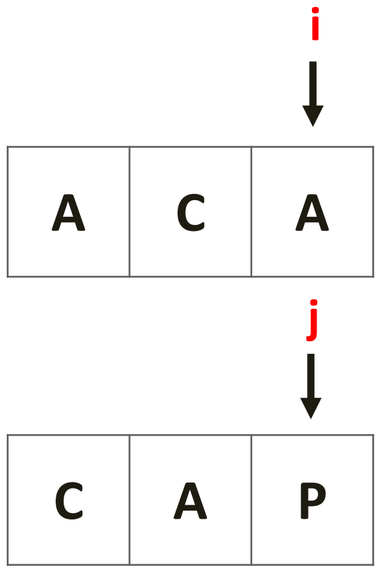
i : 2 / j : 2 / value : 0
i : 2 / j : 1 / value : 0
i : 1 / j : 0 / value : 1
i : 0 / j : 0 / value : 0

현재 인덱스를 보자. i는 3, j는 2 이다.

j는 문자열 크기 범위안에 있어서 괜찮지만, i는 이미 구간을 넘어 가버렸다.

고로, 겹치지 않는다는 표시인 0을 return 하여 이전 상태 공간으로 값을 전달한다.

최장 공통 수열(LCS)



j \ i	0	1	2
0	0	0	-1
1	-1	-1	1
2	-1	-1	0

현재 사용 된 정보

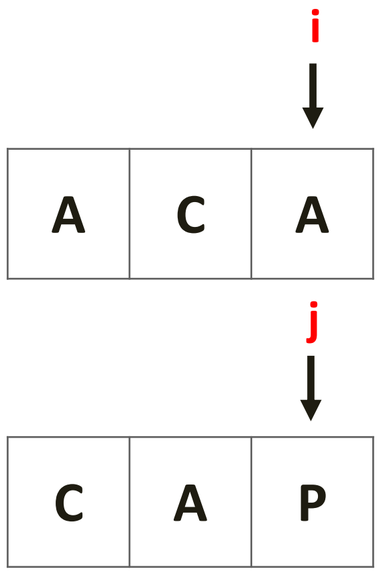
~~i : 2 / j : 2 / value : 0~~
i : 2 / j : 1 / value : 0
i : 1 / j : 0 / value : 1
i : 0 / j : 0 / value : 0

다시 i 가 2, j가 2인 차원에 돌아왔다. 아까 스택에 올린 정보를 이제 쓸 수 있게 되었다!
또한 다음 차원(dfs(3,2))의 반환 값은 0이다.

고로 dp[2][2] = max(dp[2][2], value + dfs(3,2))에 의해 dp[2][2]가 0으로 유지된다.
그 이후, 스택에서 그 상태를 지운다.

3번 행동을 하지 않았지만 똑같은 결과이기 때문에 시뮬레이션을 돌리지 않겠다.

최장 공통 수열(LCS)



j \ i	0	1	2
0	0	0	-1
1	-1	-1	1
2	-1	-1	0

현재 상태 공간의 반환값 : 0

return

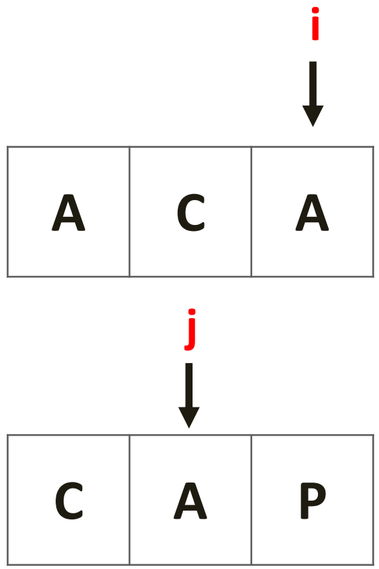
i : 2 / j : 1 / value : 0
i : 1 / j : 0 / value : 1
i : 0 / j : 0 / value : 0

2번 3번 행동을 모두 취했다고 가정 했을때, 현재 dp[2][2]의 값은 0이다.

이는 i가 2, j가 2일 때 가능한 모든 행동을 취했을 때 최대로 얻을 수 있는 값이 0을 의미한다.

현재 상태공간에서 할 수 있는 모든 행동이 끝났으므로 현재 상태 공간의 값을 return 한다.
(즉, i : 2 / j : 1일 때 3번 행동에서 얻을 수 있는 최댓값을 반환해 주기)

최장 공통 수열(LCS)



j \ i	0	1	2
0	0	0	-1
1	-1	-1	1
2	-1	-1	0

현재 사용 된 정보

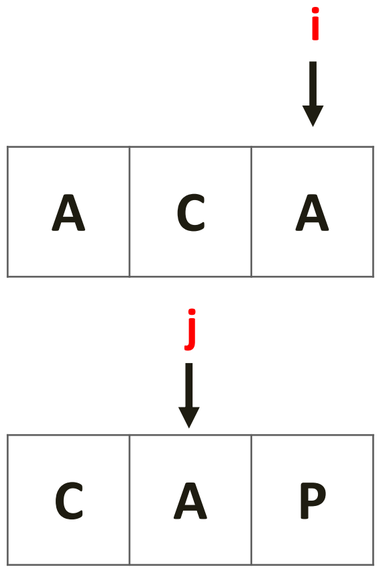
~~i : 2 / j : 1 / value : 0~~
i : 1 / j : 0 / value : 1
i : 0 / j : 0 / value : 0

다시 i 가 2, j가 1인 차원에 돌아왔다. 아까 스택에 올린 정보를 이제 쓸 수 있게 되었다!
또한 다음 차원(dfs(2,2))의 반환 값은 0이다.

고로 dp[2][1] = max(dp[2][1], value + dfs(2,2))에 의해 dp[2][2]는 그대로 0 이다.

그 이후, 스택에서 그 상태를 지운다.

최장 공통 수열(LCS)



j \ i	0	1	2
0	0	0	-1
1	-1	-1	1
2	-1	-1	0

현재 상태 공간의 반환값 : 1

return

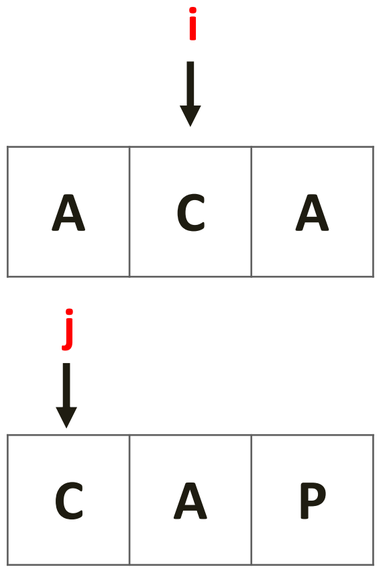
i : 1 / j : 0 / value : 1
i : 0 / j : 0 / value : 0

i가 2고 j가 1인 상태 공간에서 할 수 있는 행동은 전부 다 했다.

고로 이전 상태 공간으로 값을 반환하는 작업을 해야 한다.

현재 dp[2][1]의 값은 1이다. 이전 상태 공간에 1을 반환하자.

최장 공통 수열(LCS)



j \ i	0	1	2
0	0	2	-1
1	-1	-1	1
2	-1	-1	0

현재 사용 된 정보

~~i : 1 / j : 0 / value : 1~~

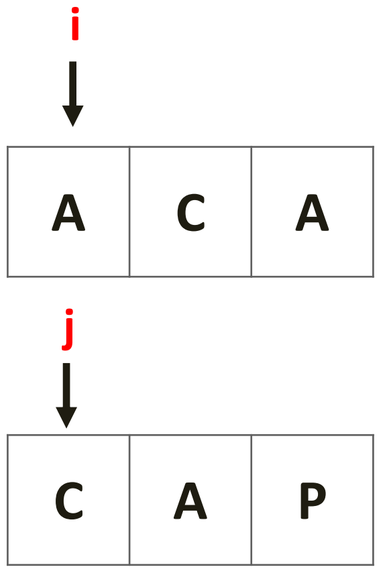
i : 0 / j : 0 / value : 0

다시 i 가 1, j가 0인 차원에 돌아왔다. 아까 스택에 올린 정보를 이제 쓸 수 있게 되었다!
또한 다음 차원(dfs(2,1))의 반환 값은 1이다.

고로 dp[1][0] = max(dp[1][0], value + dfs(2,1))에 의해 dp[1][0]가 2로 갱신 된다.
그 이후, 스택에서 그 상태를 지운다.

이후로는 지금까지 했던 행동과 똑같은 기조로 반복하면 된다.

최장 공통 수열(LCS)



j \ i	0	1	2
0	2	2	1
1	1	1	1
2	0	0	0

모든 차원을 다루기엔 방위가 너무 방대하기 때문에 일정 부분 스킵한 점은 양해 바란다.

논리 자체는 크게 이해하는데 어려움이 없을 것이다.

하지만 '이걸 굳이 재귀로 짜야해?'라고 생각이 들 수 도 있는데 재귀나 반복문을 쓰는건 프로그래머 본인의 선택이다.
하지만 필자는 코드가 간결하며 상태 전이가 직관적인 탑다운 방식을 선호하는 편이다.
결국엔 두가지 방법 다 쓸 줄 아는게 베스트이다!

행렬 곱셈 순서

크기가 N×M인 행렬 A와 M×K인 B를 곱할 때 필요한 곱셈 연산의 수는 총 N×M×K번이다. 행렬 N개를 곱하는데 필요한 곱셈 연산의 수는 행렬을 곱하는 순서에 따라 달라지게 된다.

예를 들어, A의 크기가 5×3이고, B의 크기가 3×2, C의 크기가 2×6인 경우에 행렬의 곱 ABC를 구하는 경우를 생각해보자.

- AB를 먼저 곱하고 C를 곱하는 경우 (AB)C에 필요한 곱셈 연산의 수는 5×3×2 + 5×2×6 = 30 + 60 = 90번이다.
- BC를 먼저 곱하고 A를 곱하는 경우 A(BC)에 필요한 곱셈 연산의 수는 3×2×6 + 5×3×6 = 36 + 90 = 126번이다.

같은 곱셈이지만, 곱셈을 하는 순서에 따라서 곱셈 연산의 수가 달라진다.

행렬 N개의 크기가 주어졌을 때, 모든 행렬을 곱하는데 필요한 곱셈 연산 횟수의 최솟값을 구하는 프로그램을 작성하시오. 입력으로 주어진 행렬의 순서를 바꾸면 안 된다.

입력

첫째 줄에 행렬의 개수 N($1 \le N \le 500$)이 주어진다.

둘째 줄부터 N개 줄에는 행렬의 크기 r과 c가 주어진다. ($1 \le r, c \le 500$)

항상 순서대로 곱셈을 할 수 있는 크기만 입력으로 주어진다.

출력

첫째 줄에 입력으로 주어진 행렬을 곱하는데 필요한 곱셈 연산의 최솟값을 출력한다. 정답은 2^{31}-1 보다 작거나 같은 자연수이다. 또한, 최악의 순서로 연산해도 연산 횟수가 2^{31}-1보다 작거나 같다.

3학년 1학기 알고리즘 시간에도 다루게 될 행렬 곱셈 순서 문제이다.

안타깝게도 이 문제는 바텀업으로 풀기 상당히 복잡하다.

**어떤 분할점을 정해서 행렬 곱셈을 두 부분으로 나눈 뒤,
각 부분의 최적 값을 구하고, 이를 합쳐 전체 최소 연산 횟수를 계산해야 한다.**

하지만 탑다운 방식으로 접근하면, 필요한 부분만 계산하며 재귀적으로 문제를 분할할 수 있어 훨씬 직관적이다.

행렬 곱셈 순서

30/35	35/15	15/5	5/10

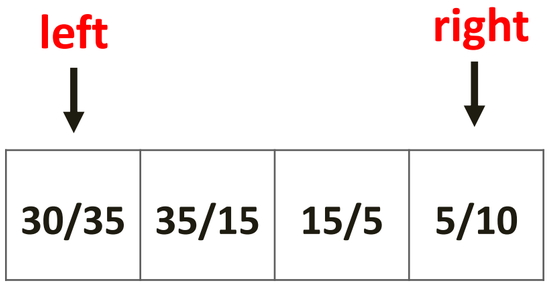
left \ right	1	2	3	4
1	-1	-1	-1	-1
2	-1	-1	-1	-1
3	-1	-1	-1	-1
4	-1	-1	-1	-1

행렬 곱셈 순서 문제는 대표적으로 탑다운으로 쉽게 풀리는 문제이다.
각 행렬의 행과 열이 주어지는데 항상 순서대로 곱셈을 할 수 있는 크기가 주어지기 때문에 아무렇게나 곱해도 하나로 합칠 수 있다.

여기서 비용을 줄이기 위해서는 합치는 순서가 매우 중요한데, 어느 분할점을 기준으로 왼쪽 오른쪽으로 나누어서 비용을 구하면 된다.
이때 비용을 구하는 식은 다음과 같다.

cost = matrix[left].row * matrix[mid].col * matrix[right].col (행렬 곱셈에 의해)

행렬 곱셈 순서



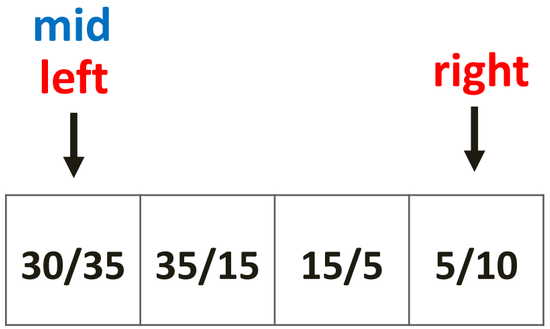
left \ right	1	2	3	4
1	-1	-1	-1	INF
2	-1	-1	-1	-1
3	-1	-1	-1	-1
4	-1	-1	-1	-1

당연히 우리는 모든 행렬을 하나로 합치는 연산을 해야하므로 left와 right의 초기값은
각각 1과 4이다.(1-based로 하겠다)
먼저 현재 차원의 값에 INF을 대입한다.(lcs는 최대값, 행렬 곱셈 순서는 최소값을 찾아야 하므로)

딱 보면 어느 분할점에서 계산을 해야 최적이 되는지 직관적으로 보이지 않는다.

고로, 반복문을 돌려 구간을 반씩 나눠 줄 것이다. (left ~ mid / mid + 1 ~ right)

행렬 곱셈 순서



left \ right	1	2	3	4
1	-1	-1	-1	INF
2	-1	-1	-1	-1
3	-1	-1	-1	-1
4	-1	-1	-1	-1

mid : 1

현재 분할점은 1이다.

즉 구간이 (1 ~ 1) / (2 ~ 4)로 나누어 지게 된다.

먼저, 1 ~ 1 구간부터 탐색해 보도록 하자.

행렬 곱셈 순서

right
left

30/35	35/15	15/5	5/10

left \ right	1	2	3	4
1	0	-1	-1	INF
2	-1	-1	-1	-1
3	-1	-1	-1	-1
4	-1	-1	-1	-1

현재 상태 공간의 반환값 : 0

return

left : 1 / right : 4

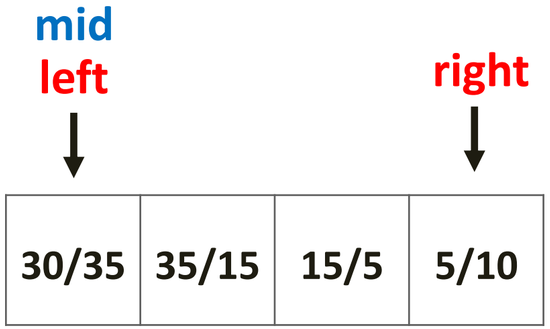
현재 left와 right는 서로 같은 포인터를 가진다. 즉 동일한 값을 가리킨다.

행렬을 곱하기 위해선 적어도 2개의 행렬이 필요한데 현재 값은 1개의 값만을 가진다.

이는 나눌 수 없는 최소 단위까지 나눠졌다는 뜻이기 때문에 아무런 연산이 발생하지 않는다.

고로 (1~1)에 0을 대입한후 결과값(0)을 리턴 한다.

행렬 곱셈 순서



left \ right	1	2	3	4
1	0	-1	-1	INF
2	-1	-1	-1	-1
3	-1	-1	-1	-1
4	-1	-1	-1	-1

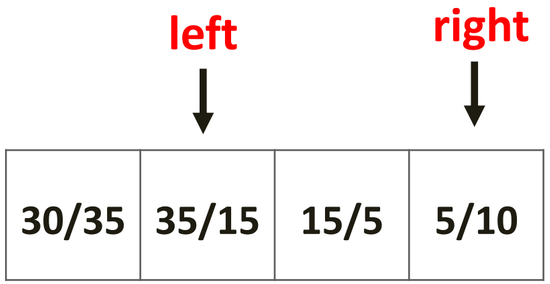
mid : 1

다시 1 ~ 4 구간에 돌아왔다. 아까 언급했듯이 분할점은 1이였다.

즉 왼쪽 분할에서 얻을 수 있는 값은 0이였다.

다음으로 오른쪽 구간인 (2 ~ 4) 구간을 탐색해 보도록 하자.

행렬 곱셈 순서



right / left	1	2	3	4
1	0	-1	-1	INF
2	-1	-1	-1	INF
3	-1	-1	-1	-1
4	-1	-1	-1	-1

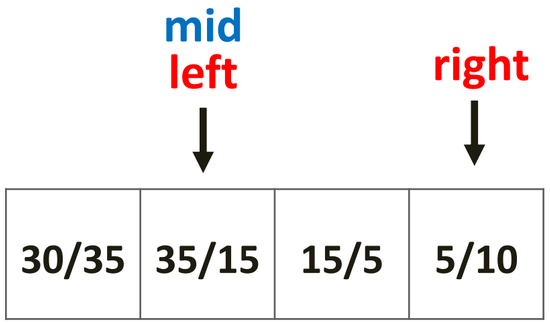
left : 1 / right : 4

현재 left는 2이며, right는 4이다.
먼저 현재 차원이 방문이 되지 않았으므로 값에 INF을 대입한다.

행렬들의 개수가 3개 이상이므로 바로 곱할 수가 없으며 분할을 해서 최적해를 찾아야한다.

새로운 상태 공간에서도 분할점을 찾기 위해서 반복문을 돌려준다.

행렬 곱셈 순서



left \ right	1	2	3	4
1	0	-1	-1	INF
2	-1	-1	-1	INF
3	-1	-1	-1	-1
4	-1	-1	-1	-1

left : 1 / right : 4

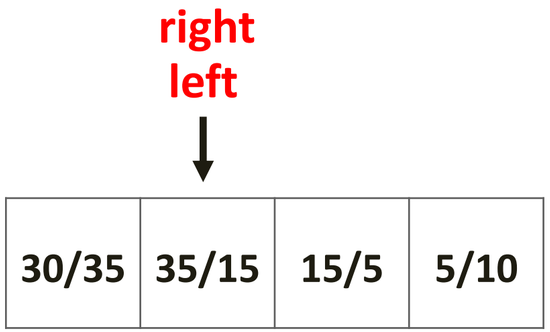
mid : 2

현재 분할점은 2이다.

즉 구간이 (2~2) / (3~4)로 나누어 지게 된다.

먼저 (2 ~ 2) 구간부터 탐색해보자.

행렬 곱셈 순서



left \ right	1	2	3	4
1	0	-1	-1	INF
2	-1	0	-1	INF
3	-1	-1	-1	-1
4	-1	-1	-1	-1

현재 상태 공간의 반환값 : 0

return

left : 2 / right : 4
left : 1 / right : 4

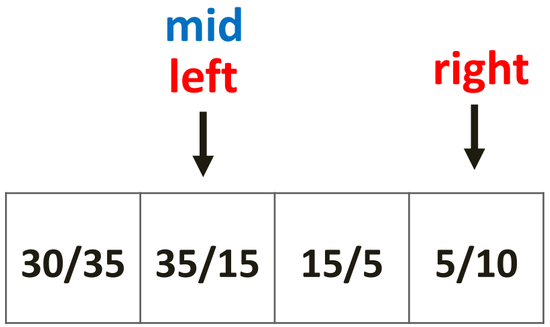
현재 left와 right는 서로 같은 포인터를 가진다. 즉 동일한 값을 가리킨다.

행렬을 곱하기 위해선 적어도 2개의 행렬이 필요한데 현재 값은 1개의 값만을 가진다.

이는 나눌 수 없는 최소 단위까지 나눠졌다는 뜻이기 때문에 아무런 연산이 발생하지 않는다.

고로 (2~2)에 0을 대입한후 결과값(0)을 리턴 한다.

행렬 곱셈 순서



right / left	1	2	3	4
1	0	-1	-1	INF
2	-1	0	-1	INF
3	-1	-1	-1	-1
4	-1	-1	-1	-1

left : 1 / right : 4

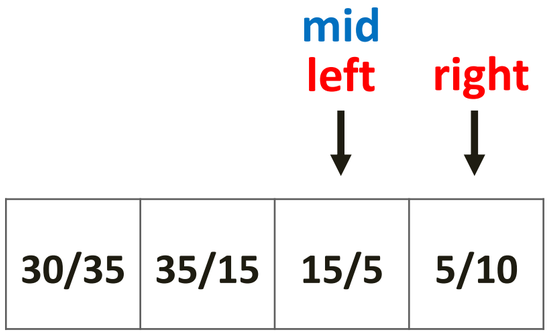
mid : 2

다시 2 ~ 4 구간에 돌아왔다. 아까 언급했듯이 분할점은 2이였다.

즉 왼쪽 분할에서 얻을 수 있는 값은 0이였다.

다음으로 오른쪽 구간인 (3 ~ 4) 구간을 탐색해 보도록 하자.

행렬 곱셈 순서



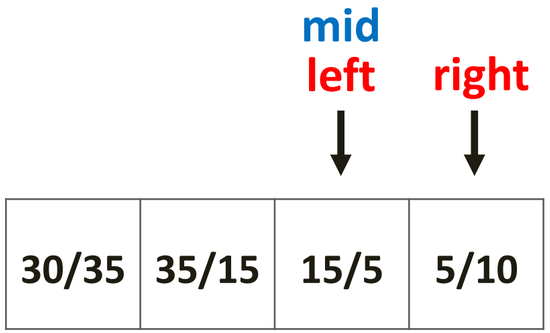
left \ right	1	2	3	4
1	0	-1	-1	INF
2	-1	0	-1	INF
3	-1	-1	-1	INF
4	-1	-1	-1	-1

left : 2 / right : 4
left : 1 / right : 4

현재 left는 3이며, right는 4이다.
먼저 현재 차원이 방문이 되지 않았으므로 값에 INF을 대입한다.
행렬이 딱 2개이다! 문제에서 언급했듯이 크기가 행렬A(N*M) x 행렬B(M*K)를 곱하는 연산 횟수는 N*M*K라고 하였다.

행렬이 2개인데 분할을 스킵 할 수 있는 이유는 분할점이 단 하나밖에 나오지 않기 때문이다.
(분할점의 범위는 left ~ right - 1)

행렬 곱셈 순서



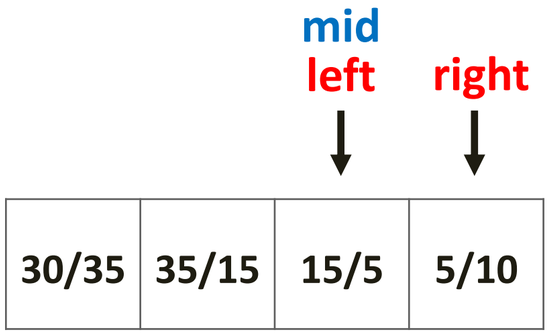
left \ right	1	2	3	4
1	0	-1	-1	INF
2	-1	0	-1	INF
3	-1	-1	0	INF
4	-1	-1	-1	0

left : 2 / right : 4
left : 1 / right : 4

즉,
(1)저 두 행렬을 그대로 곱하여 750 (15 * 5 * 10)이라는 값
(2)left의 행(15), mid의 열(left라고 가정, 5), right의 열(15)을 곱한 값
(1)번 값과 (2)번 값이 동일하게 되는데, 분할 정복을 위해 분할점을 잡는 당위성을 확보할 수 있다는 걸 보여주고 싶었다.

즉 행렬이 단 2개 있다고 하더라도 (2)번식을 사용해서 계산 할 수 있게 되므로 다른 경우에 대해서도 저 계산식을 사용할 수 있게 되었다.

행렬 곱셈 순서



left \ right	1	2	3	4
1	0	-1	-1	INF
2	-1	0	-1	INF
3	-1	-1	0	750
4	-1	-1	-1	0

현재 상태 공간의 반환값 : 750

return

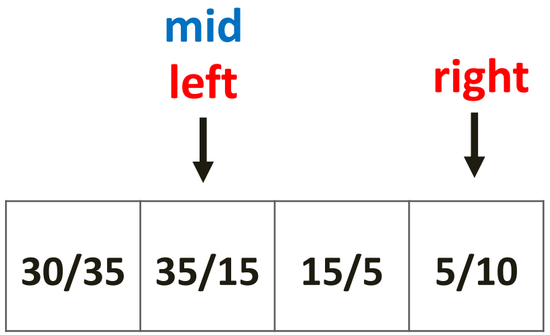
left : 2 / right : 4
left : 1 / right : 4

결론적으로 left가 3, right가 4일 때 얻을 수 있는 최소 값은 750이다.

이는 기존 dp[3][4]의 값인 INF보다 작다.

고로 (3~4)에 750을 저장한 후, 결과값을 return 한다.

행렬 곱셈 순서



left \ right	1	2	3	4
1	0	-1	-1	INF
2	-1	0	-1	6000
3	-1	-1	0	750
4	-1	-1	-1	0

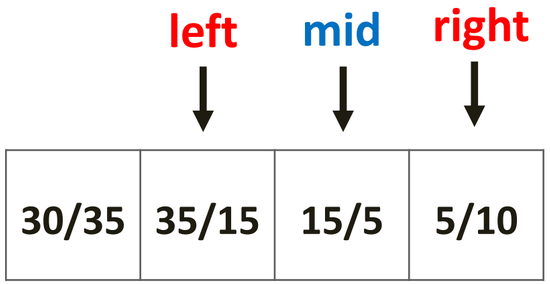
left : 1 / right : 4

mid : 2

다시 2 ~ 4 구간에 돌아왔다.
왼쪽에서 얻을 수 있는 값(dfs(2,2))인 0, 오른쪽에서 얻을 수 있는 값(dfs(3,4))인 750

(2 ~ 4) 구간에서 분할점을 2로 잡았을 때의 결과값은 아래와 같다.
dp[2][4] = min(dp[2][4], dfs(2,2) + dfs(3,4) + matrix[left].row * matrix[mid].col * matrix[right].col)
min(INF , 750 + 35 * 15 * 10)에 의해 6000으로 갱신된다.

행렬 곱셈 순서



right / left	1	2	3	4
1	0	-1	-1	INF
2	-1	0	-1	6000
3	-1	-1	0	750
4	-1	-1	-1	0

left : 1 / right : 4

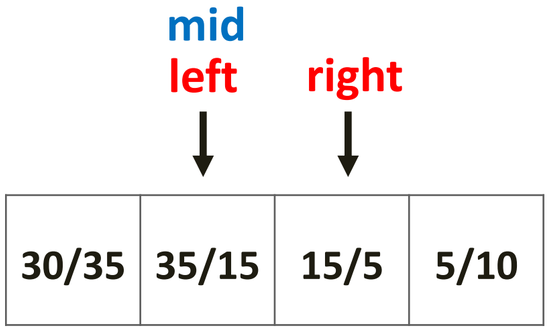
mid : 3

이번 분할점은 3번이다.

즉 구간이 (2~3) / (4~4)로 나누어 지게 된다.

먼저 (2 ~ 3) 구간부터 탐색해보자.

행렬 곱셈 순서



left \ right	1	2	3	4
1	0	-1	-1	INF
2	-1	0	2625	6000
3	-1	-1	0	750
4	-1	-1	-1	0

현재 상태 공간의 반환값 : 2625

return

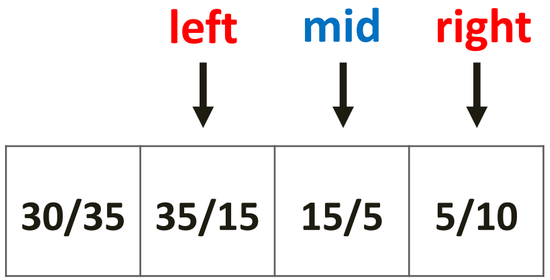
left : 2 / right : 4
left : 1 / right : 4

현재 left는 2이며, right는 3이다.
먼저 현재 차원이 방문이 되지 않았으므로 값에 INF을 대입한다.

행렬이 딱 2개이다!

아까 상황에도 보았듯이 바로 계산을 해버리면 된다. (실제 코드에선 분할되게 계산 할것)
즉 dp[2][3] = min(dp[2][3], dfs(2,2) + dfs(3,3) + 35 * 15 * 5)에 의해 2625로 갱신된다.
현재 값을 반환해준다.

행렬 곱셈 순서



right / left	1	2	3	4
1	0	-1	-1	INF
2	-1	0	2625	4375
3	-1	-1	0	750
4	-1	-1	-1	0

left : 1 / right : 4

mid : 3

다시 left 2이며 right 4인 구간에 돌아왔다. 오른쪽 구간의 계산은 스킵 하겠다.(행렬 단 한개)

계산은 아래와 같이 이루어진다.

dp[2][4] = min(dp[2][4], dfs(2,3) + dfs(4,4) + 35 * 5 * 10) -> min(6000, 2625 + 1750)
위 식에 의해 4375라는 더 작은 값으로 갱신된다.

행렬 곱셈 순서

left right

30/35	35/15	15/5	5/10

left \ right	1	2	3	4
1	0	-1	-1	INF
2	-1	0	2625	4375
3	-1	-1	0	750
4	-1	-1	-1	0

현재 상태 공간의 반환값 : 4375

return

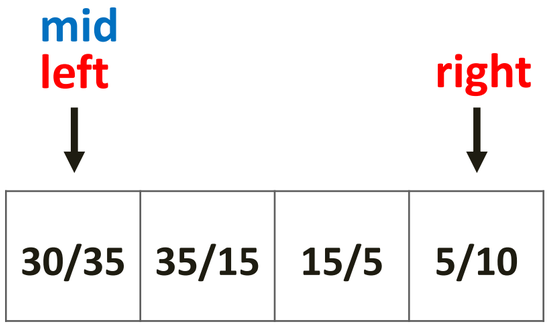
left : 1 / right : 4

이로써 left가 2이고 right가 4일 때의 모든 분할점을 살펴보았다.

최소값은 4375였다.

이제 이 값을 반환해준다.

행렬 곱셈 순서



right / left	1	2	3	4
1	0	-1	-1	14875
2	-1	0	2625	4375
3	-1	-1	0	750
4	-1	-1	-1	0

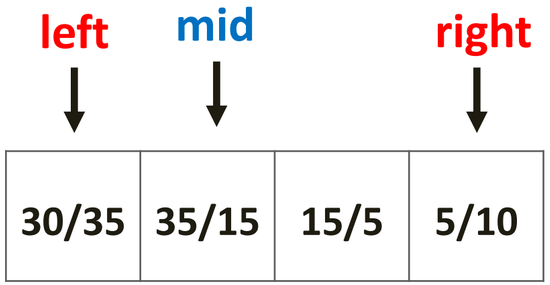
mid : 1

다시 1 ~ 4 구간에 돌아왔다.

계산은 아래와 같이 이루어 진다.

dp[1][4] = min(dp[1][4], dfs(1,1) + dfs(2,4) + 30 * 35 * 10) -> min(INF, 4375 + 10500)
위 식에 의해 14875라는 더 작은 값으로 갱신된다.

행렬 곱셈 순서



left \ right	1	2	3	4
1	0	-1	-1	14875
2	-1	0	2625	4375
3	-1	-1	0	750
4	-1	-1	-1	0

mid : 2

다음 분할 점을 보자. 현재 분할점은 2이다.

즉 구간이 (1~2) / (3~4)로 나누어 지게 된다.

먼저 (1 ~ 2) 구간부터 탐색해보자.

행렬 곱셈 순서

left right

30/35	35/15	15/5	5/10

left \ right	1	2	3	4
1	0	15750	-1	14875
2	-1	0	2625	4375
3	-1	-1	0	750
4	-1	-1	-1	0

현재 상태 공간의 반환값 : 15750

return

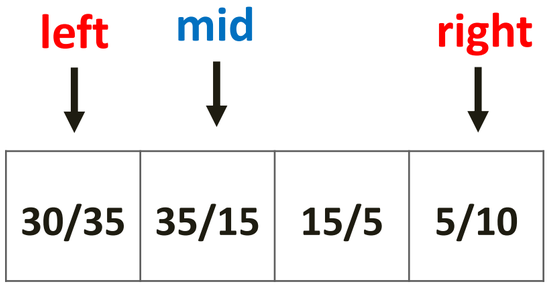
left : 1 / right : 4

현재 left는 1이며, right는 2이다.

먼저 현재 차원이 방문이 되지 않았으므로 값에 INF을 대입한다.

현재 구간엔 행렬이 단 2개 있다.
고로 min(INF, 30 * 35 * 15)에 의해 15750값이 선택된다.
이 값을 return 하자.

행렬 곱셈 순서



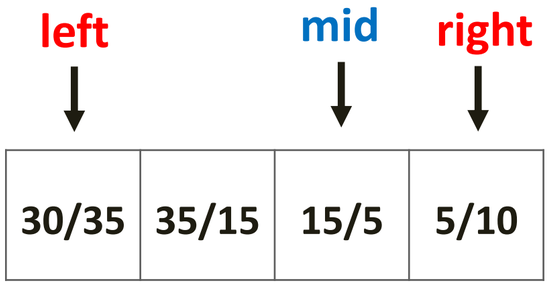
left \ right	1	2	3	4
1	0	15750	-1	14875
2	-1	0	2625	4375
3	-1	-1	0	750
4	-1	-1	-1	0

mid : 2

다시 1 ~ 4 구간에 돌아왔다.
왼쪽 구간인 (1~2)에선 15750을 얻을 수 있었다. 또한 오른쪽 구간인(3~4)는 이미 계산 되어있다. (750)
계산은 아래와 같이 이루어 진다.

dp[1][4] = min(dp[1][4], + dfs(1,2) + dfs(3,4) + 30 * 15 * 10) -> min(14875, 15750 + 750 + 4500)
위 식에 의해 더 작은 값인 14875가 선택된다.

행렬 곱셈 순서



left \ right	1	2	3	4
1	0	15750	-1	14875
2	-1	0	2625	4375
3	-1	-1	0	750
4	-1	-1	-1	0

mid : 3

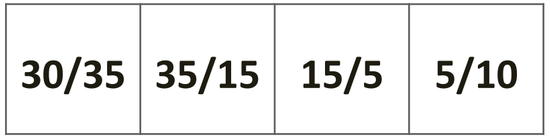
다음 분할 점을 보자. 현재 분할점은 3이다.

즉 구간이 (1~3) / (4~4)로 나누어 지게 된다.

먼저 (1 ~ 3) 구간부터 탐색해보자.

행렬 곱셈 순서

left right



30/35	35/15	15/5	5/10

left \ right	1	2	3	4
1	0	15750	INF	14875
2	-1	0	2625	4375
3	-1	-1	0	750
4	-1	-1	-1	0

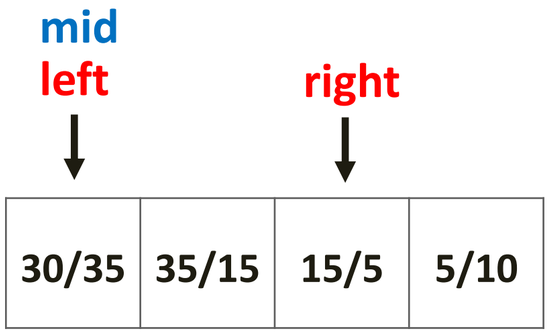
left : 1 / right : 4

현재 left는 1이며, right는 3이다.

먼저 현재 차원이 방문이 되지 않았으므로 값에 INF을 대입한다.

구간에 행렬이 3개 이상 있으므로 분할을 해야한다.

행렬 곱셈 순서



left \ right	1	2	3	4
1	0	15750	7875	14875
2	-1	0	2625	4375
3	-1	-1	0	750
4	-1	-1	-1	0

left : 1 / right : 4

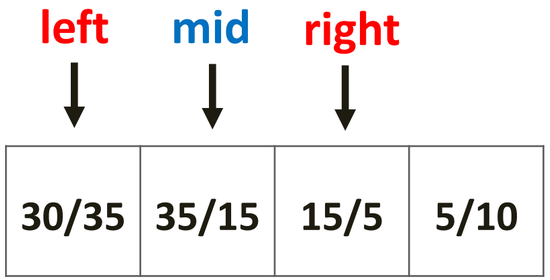
mid : 1

현재 분할점은 1이다.

구간이 (1~1) / (2~3)으로 나누어지게 된다.
이미 두 값이 구해져 있으므로 계산은 아래와 같이 이루어 진다.

dp[1][3] = min(dp[1][3], dfs(1,1) + dfs(2,3) + 30 * 35 * 5) -> min(INF, 2625 + 5250)
위 식에 의해 더 작은 값인 7875가 선택되게 된다.

행렬 곱셈 순서



left \ right	1	2	3	4
1	0	15750	7875	14875
2	-1	0	2625	4375
3	-1	-1	0	750
4	-1	-1	-1	0

left : 1 / right : 4

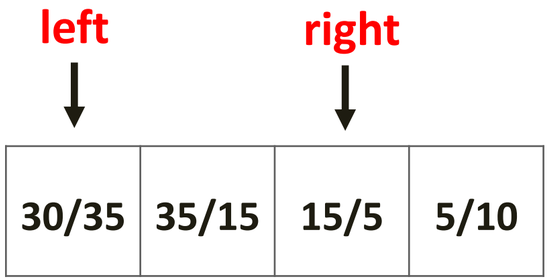
mid : 2

다음 분할점을 보자. 현재 분할점은 2이다.

구간이 (1~2) / (3~3)으로 나누어지게 된다.
이미 두 값이 구해져 있으므로 계산은 아래와 같이 이루어 진다.

dp[1][3] = min(dp[1][3], dfs(1,2) + dfs(3,3) + 30 * 15 * 5) -> min(7875, 15750 + 2250)
위 식에 의해 더 작은 값인 7875가 선택되게 된다.

행렬 곱셈 순서



left \ right	1	2	3	4
1	0	15750	7875	14875
2	-1	0	2625	4375
3	-1	-1	0	750
4	-1	-1	-1	0

현재 상태 공간의 반환값 : 7875

return

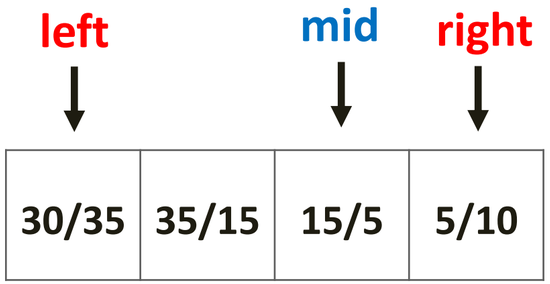
left : 1 / right : 4

left 가 1이고 right가 3일 때의 모든 분할점을 살펴 보았다.

최종적으로 얻을 수 있는 최소값은 7875였다.

이 값을 반환하자.

행렬 곱셈 순서



left \ right	1	2	3	4
1	0	15750	7875	9375
2	-1	0	2625	4375
3	-1	-1	0	750
4	-1	-1	-1	0

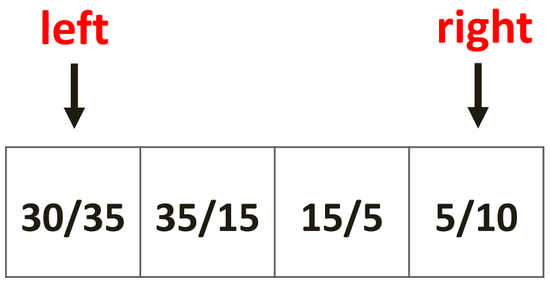
mid : 3

다시 구간 1 ~ 4에 돌아왔다.
오른쪽 구간인 (4,4)는 계산하지 않는다.(0)

계산은 아래와 같이 이루어 진다.

dp[1][4] = min(dp[1][4], dfs(1,3) + dfs(4,4) + 30 * 5 * 10) -> min(14875, 7875 + 1500)
위 식에 의해 더 작은 값인 9375가 선택되게 된다.

행렬 곱셈 순서



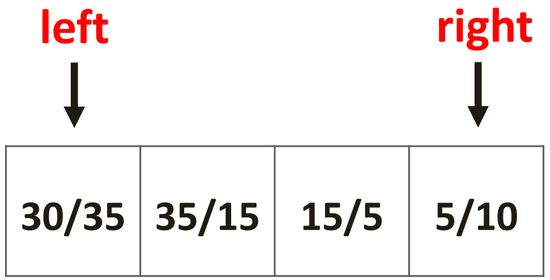
left \ right	1	2	3	4
1	0	15750	7875	9375
2	-1	0	2625	4375
3	-1	-1	0	750
4	-1	-1	-1	0

이제 구간 1 ~ 4에서 볼 수 있는 모든 분할점을 살펴 보았다.

left가 1, right가 4 일 때의 최소값은 9375였다.

이 값을 리턴 하게 되면 main으로 돌아가게 되며 자연스레 우리가 구하려는 최소값이 반환되게 된다.

행렬 곱셈 순서



right / left	1	2	3	4
1	0	15750	7875	9375
2	-1	0	2625	4375
3	-1	-1	0	750
4	-1	-1	-1	0

탑다운 dp 방식은 분할 문제에서 빛을 발한다.

직접 코드를 짤 때 신기한(?)점이 있는데 함수의 매개변수 인자수가 dp테이블의 차원 크기와 아예 동일하다는 점이다.

탑다운을 잘 사용하기 위해선 기저 조건을 잘 정해야 하며, dp테이블을 값을 저장하는 용도인 동시에 방문 배열 용도로 사용해 주어야한다.