Chapter 04.
동적 계획법 #1

Clip 01 | [2748] 피보나치수 2
동적 계획법이란?
점화식을 잘 유도하는 방법

Clip 02 | [11727] 2*n 타일링 2
기준 점과 방법의 수

Clip 03 | [2156] 포도주 시식
중복을 허용하지 않는 순열 생성

Clip 04 | [11047] 동전 0
최소 동전의 개수를 찾는 그리디

Clip 05 | [2294] 동전 2
최소 동전의 개수를 찾는 DP

Clip 06 | [2293] 동전 1
가치의 경우의 수를 찾는 DP

Clip 07 | [2624] 동전 바꿔주기
개수 제한이 있는 동전 교환

Clip 08 | [12865] 평범한 배낭
배낭의 최대 가치를 위한 DP

Ch04. 동적 계획법 #1

1. [2748] 피보나치 수 2

동적 계획법?

정의

복잡한 문제를 작은 하위 문제로 나눠 해결하고,

그 결과를 조합하여 최종 해답을 얻는 알고리즘 설계 기법

작은 부분 문제들의 해를 미리 구해서 중복 계산을 방지

동적 계획법?

필수 조건

- 최적 부분 구조 (Optimal Substructure)
  - 큰 문제의 최적 해가 작은 문제의 최적 해를 포함

- 중복 부분 문제 (Overlapping Subproblems)
  - 같은 부분 문제가 여러 번 반복되어야 함

동적 계획법?

최적 부분 구조

- 큰 문제의 최적 해가 작은 문제의 최적 해를 포함

---

ex) 피보나치 수
n번째 피보나치 수를 f(n) 이라고 표현하면

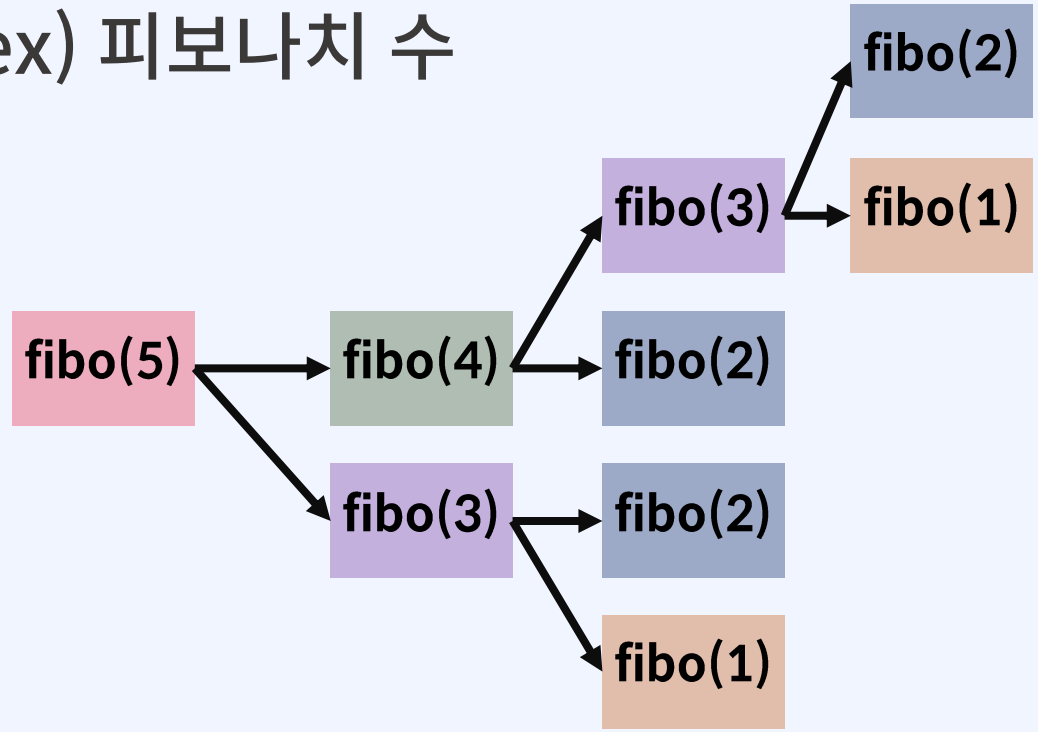
- f(n) = f(n-1) + f(n-2)
- f(n-1) = f(n-2) + f(n-3)
- f(n-2) = f(n-3) + f(n-4)

동적 계획법?

중복 부분 문제 (Overlapping Subproblems)

- 같은 부분 문제가 여러 번 반복되어야 함

ex) 피보나치 수



- fibo(5): 1회
- fibo(4): 1회
- fibo(3): 2회
- fibo(2): 3회
- fibo(1): 2회

동적 계획법?

중복 부분 문제 (Overlapping Subproblems)

- 같은 부분 문제가 여러 번 반복되어야 함

부분 문제가 여러 번 반복되지 않는다면?

반복되는 연산을 줄이는 목적에 부합하지 않음

재사용하는 않는 문제의 정답을 모두 메모리에 저장하면
사용하지 않는 메모리만 낭비됨

동적 계획법?

접근 방법

하향식 접근 (Memoization)

- 계산한 결과를 메모리에 저장하고 재사용
- 재귀식 접근
- 상대적으로 입력이 적은 경우에 사용

상향식 접근 (Tabulation)

- 하위 문제의 결과를 표에 저장
- 입력이 많은 경우에 적합
- 하위 문제가 한번에 구해지지 않는 경우에 사용

BOJ2748: 피보나치 수2

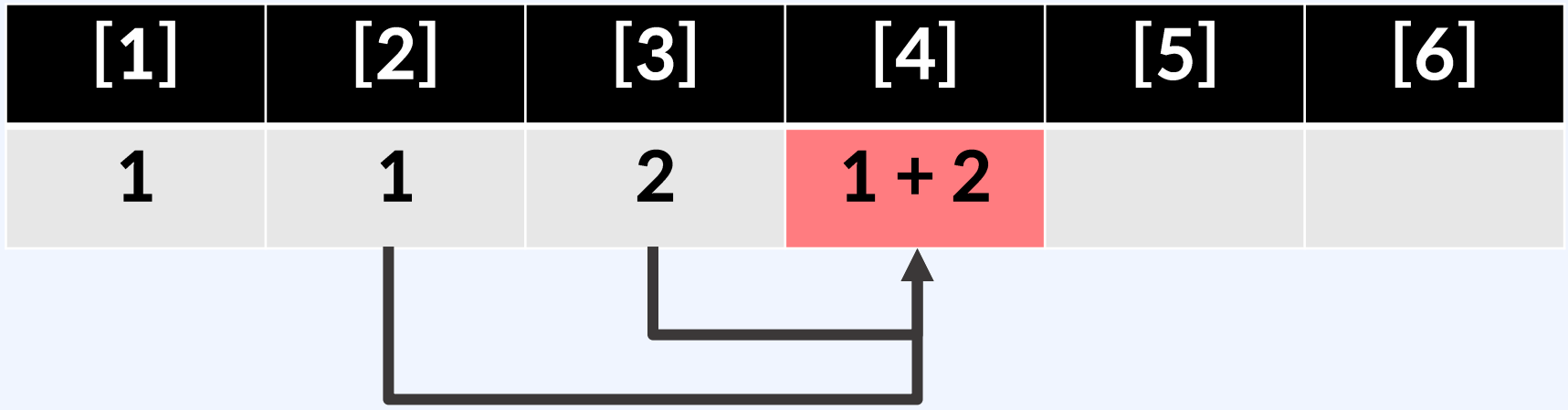
1. 하향식 접근

```
static long[] memo = new long[91];
static long fibo(int n) {
    if (n == 1 || n == 2) return 1;
    if(memo[n] != 0) return memo[n];
    memo[n] = fibo(n - 1) + fibo(n - 2);
    return memo[n];
}
```

n번째 피보나치 수를 구하기 위해 하향(n-1, n-2)접근 한다

BOJ2748: 피보나치 수2

2. 상향식 접근 (1)



{i-1}, {i-2} 번째 피보나치 수를 구하고, {i}번째에 상향 반영한다

BOJ2748: 피보나치 수2

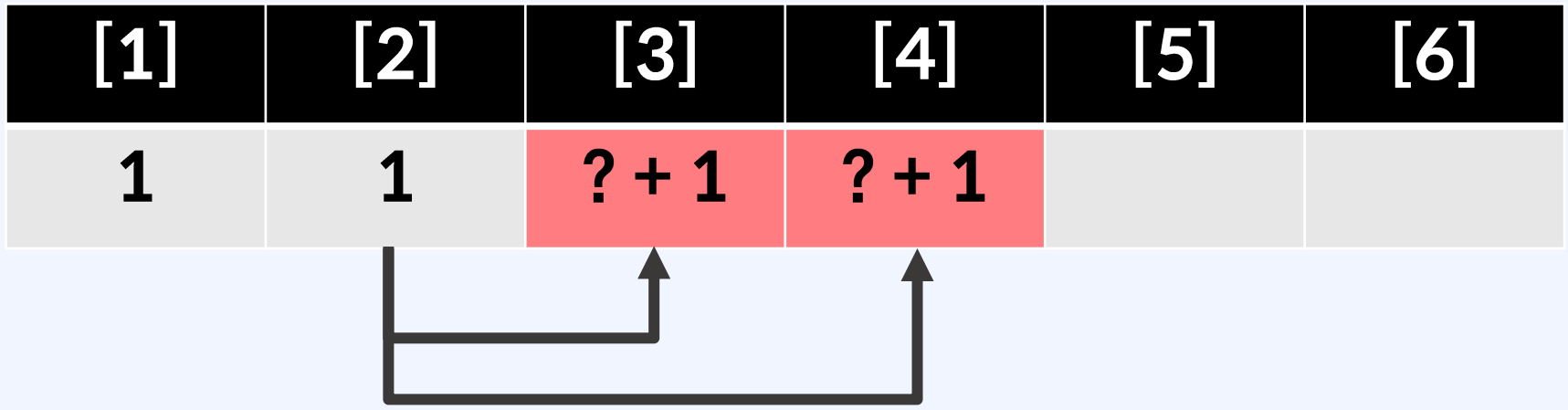
2. 상향식 접근 (1)

```
static long[] dp = new long[91];
static long fibo2(int n) {
    dp[1] = dp[2] = 1;
    for(int i = 3; i <= n; i++)
        dp[i] = dp[i - 1] + dp[i -2];
    return dp[n];
}
```

{i-1}, {i-2} 번째 피보나치 수를 구하고, {i}번째에 상향 반영한다

BOJ2748: 피보나치 수2

2. 상향식 접근 (2)



{i} 번째 피보나치 수를 {i + 1}, {i + 2} 번째에 누적 합으로 반영한다

BOJ2748: 피보나치 수2

2. 상향식 접근 (2)

```
static long[] dp = new long[93];
static long fibo3(int n) {
    dp[1] = 1;
    for (int i = 1; i <= n; i++) {
        dp[i + 1] += dp[i];
        dp[i + 2] += dp[i];
    }
    return dp[n];
}
```

{i-1}, {i-2} 번째 피보나치 수를 구하고, {i}번째에 상향 반영한다

BOJ2748: 피보나치 수2

동적 계획법 잘 짜는 법?

1. 문제의 요구사항을 점화식의 결과가 되도록 유도한다

2. 문제를 작은 부분문제로 쪼갠다
   - 단, {i} 번째 문제는 {i – k} 번째 부분문제를 활용할 수 있도록 쪼갠다

3. 동적 배열의 차원 수를 줄일 수 있는지 한번 더 검토한다

Ch04. 동적 계획법 #1

2. [11727] 2*N 타일링

BOJ11727: 2*N 타일링 2

문제 요약

- 2 * N 직사각형을 채우는 방법의 수 (1 <= n <= 1,000)
  - 방법의 수를 10,007로 나눈 나머지를 출력

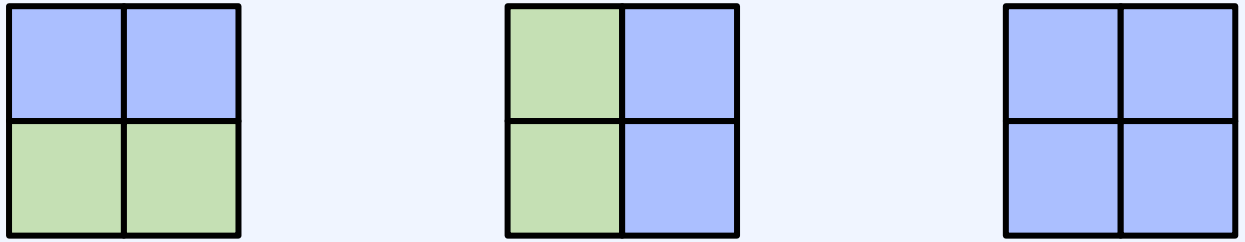
- 타일
  - 1 * 2
  - 2 * 1
  - 2 * 2

BOJ11727: 2*N 타일링 2

문제 요약

입력 데이터
2

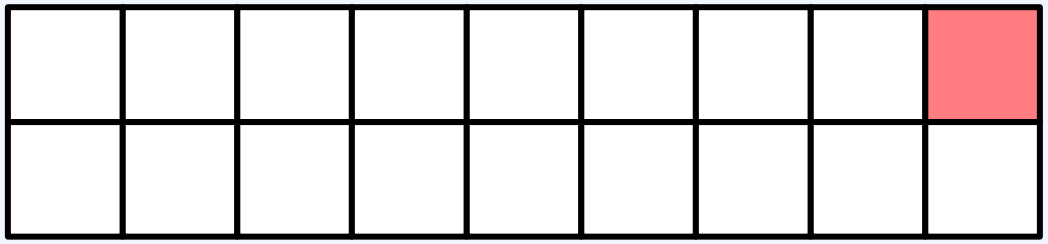
출력 데이터
3



BOJ11727: 2*N 타일링 2

문제 분석

- 문제의 요구사항: 방법의 수
- d[n] = 2 * n 영역을 채우는 방법의 수

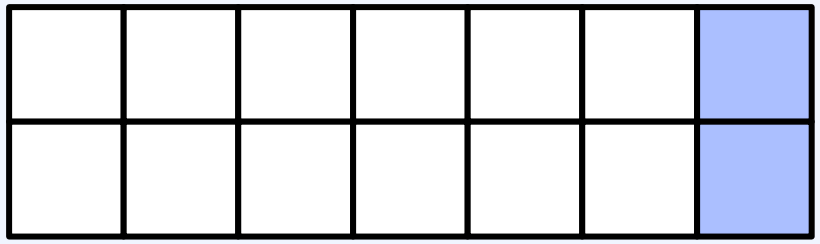


기준점 하나를 잡고 생각해보자
가장 오른쪽 위의 영역은 무슨 종류의 타일로 채워질까?

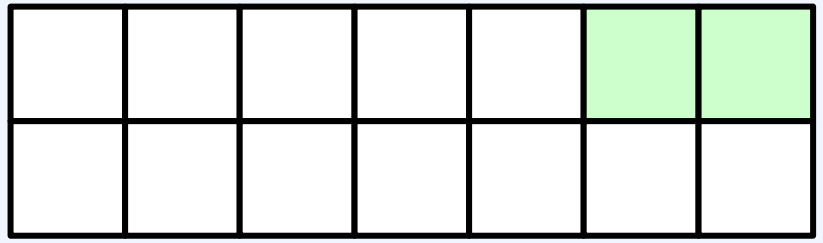
BOJ11727: 2*N 타일링 2

문제 분석

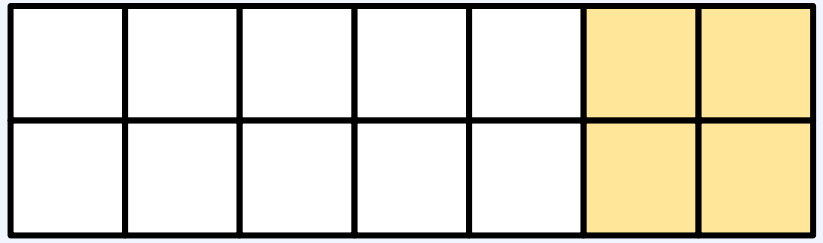
Type1. (2 * 1)



Type2. (1 * 2)



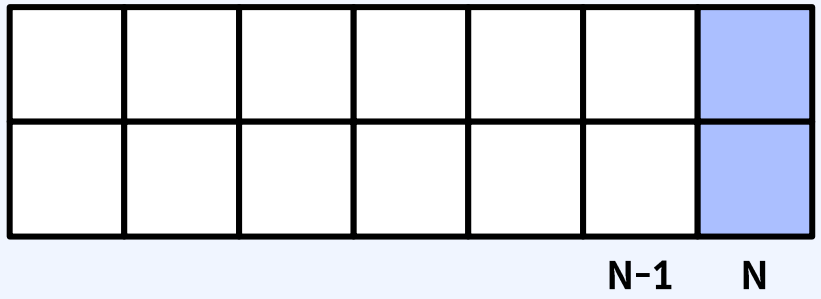
Type3. (2 * 2)



BOJ11727: 2*N 타일링 2

문제 분석

Type1. (2 * 1)



d[n] = 2 * n 영역을 채우는 방법의 수

우측 상단을 (2 * 1)로 채운다면?

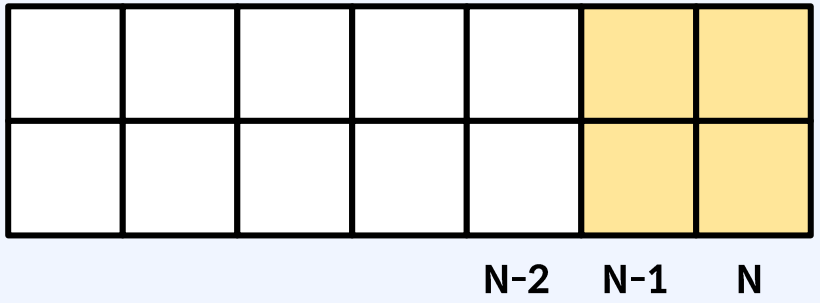
[1, N-1] 구간이 모두 비어있다

2 * (n-1) 영역을 채우는 방법의 수를 구하면 된다
→ d[n-1]

BOJ11727: 2*N 타일링 2

문제 분석

Type3. (2 * 2)



d[n] = 2 * n 영역을 채우는 방법의 수

우측 상단을 (2 * 2)로 채운다면?

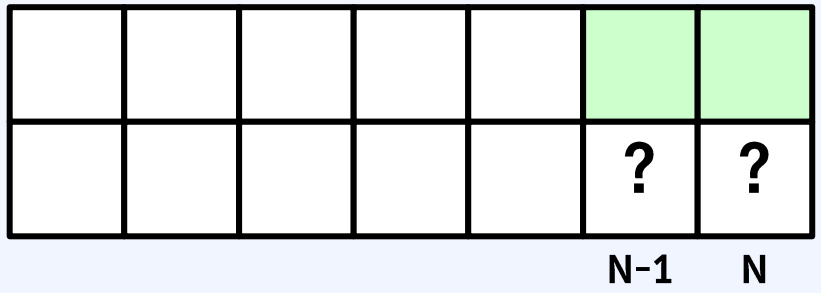
[1, N-2] 구간이 모두 비어있다

2 * (n-2) 영역을 채우는 방법의 수를 구하면 된다
→ d[n-2]

BOJ11727: 2*N 타일링 2

문제 분석

Type2. (1 * 2)



우측 상단을 (1 * 2)로
채운다면?

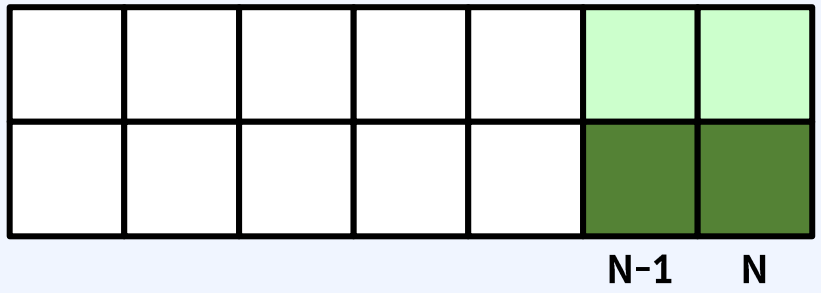
[1, N-2] 구간이 모두 비어있다
(2, N-1), (2, N) 가 비어있다

어떻게 계산을 해야할까?

BOJ11727: 2*N 타일링 2

문제 분석

Type2. (1 * 2)



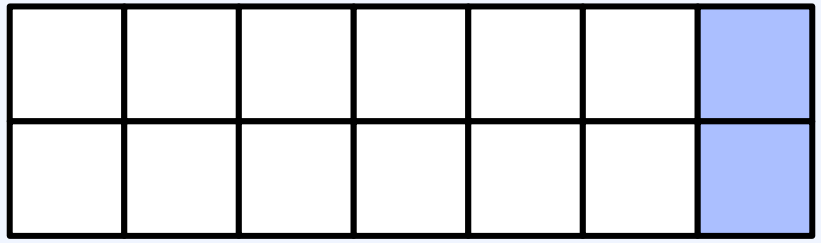
(2, N-1), (2, N) 을 채우는
방법의 수?

저 공간에 들어갈 수 있는 타일은
(1 * 2) 타일 하나밖에 없다
→ 경우의 수가 1개 뿐이다
→ d[n-2]

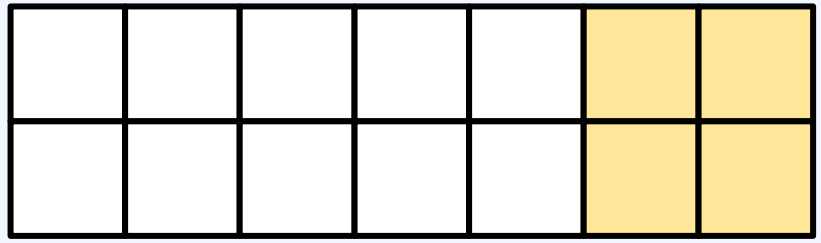
BOJ11727: 2*N 타일링 2

문제 분석

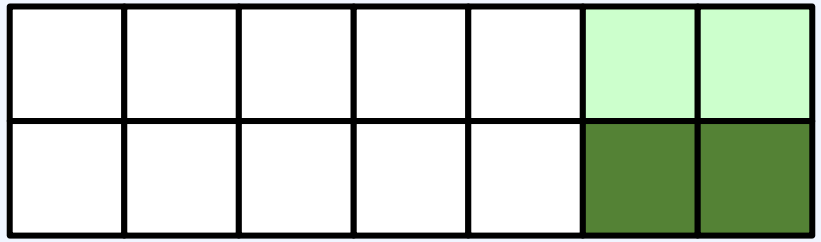
Type1. (2 * 1)



Type3. (2 * 2)



Type2. (1 * 2)



d[n] 을 채우는 방법의 수

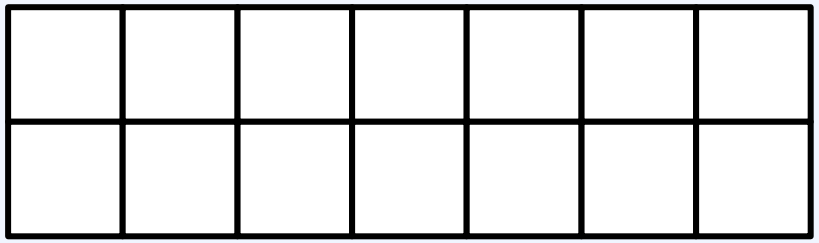
Type1 + Type2 + Type3
d[n-1] + d[n-2] + d[n-2]

BOJ11727: 2*N 타일링 2

구현

d[n] 을 채우는 방법의 수

Type1 + Type2 + Type3
d[n-1] + d[n-2] + d[n-2]



d[1] = 1
d[2] = 3

BOJ11727: 2*N 타일링 2

구현

```
d[1] = 1;
d[2] = 3;
for (int i = 3; i <= n; i++) {
    d[i] = (d[i - 1] + d[i - 2] + d[i - 2]) % 10007;
}
```

Type1 + Type2 + Type3
d[n-1] + d[n-2] + d[n-2]

Ch04. 동적 계획법 #1

3. [2156] 포도주 시식

BOJ2156: 포도주 시식

문제 요약

- N개의 잔에 포도주가 담겨있음 (포도주 양: <= 1000)
- 잔의 개수 (1 <= n <= 10,000)
- 규칙
  - 잔을 고르면 모두 마시고 원래위치에 두어야 한다
  - 연속으로 놓여있는 3잔을 마실 수 없다
- 최대한 합이 커지도록 선택

BOJ2156: 포도주 시식

문제 요약

입력 데이터
6 6 10 13 9 8 1

출력 데이터
33

BOJ2156: 포도주 시식

문제 분석

- 문제 요구사항: 최대 포도주의 양
  - d[n]? = n번째 잔까지 선택한 최대 포도주의 양
  - 조건에 따라서 점화식의 차원이 더 늘어날 수 있다

BOJ2156: 포도주 시식

문제 분석

- 연속해서 3잔의 포도주를 선택할 수 없다
  - {N}번을 선택하지 않은 경우
  - {N-1}을 선택하지 않고 {N}번을 선택한 경우
  - {N-1}을 선택하고, {N}번을 선택한 경우
  - {N-2}, {N-1}, {N}을 모두 선택할 수는 없다

BOJ2156: 포도주 시식

문제 분석

- [case 0] {N}번을 선택하지 않은 경우
  - d[0][n]
- [case 1] {N-1}을 선택하지 않고 {N}번을 선택한 경우
  - d[1][n]
- [case 2] {N-1}을 선택하고, {N}번을 선택한 경우
  - d[2][n]

BOJ2156: 포도주 시식

문제 분석

[case 0] {N}번을 선택하지 않은 경우
- d[0][n]

N-1을 선택한 경우의 수 중에 가장 큰 값을 고른다
- max(d[0][n-1], d[1][n-1], d[2][n-1])

BOJ2156: 포도주 시식

문제 분석

[case 1] {N-1}을 선택하지 않고 {N}번을 선택한 경우

- d[1][n]

N-1을 선택하지 않았으므로: d[0][n-1]
N을 선택했으므로: wine[n]

→ d[0][n-1] + wine[n]

BOJ2156: 포도주 시식

문제 분석

[case 2] {N-1}을 선택하고, {N}번을 선택한 경우

- d[2][n]

N-1을 선택 했으므로: d[1][n-1]
(d[2][n-1] 이 아닌 이유는?)
N을 선택했으므로: wine[n]

→ d[1][n-1] + wine[n]

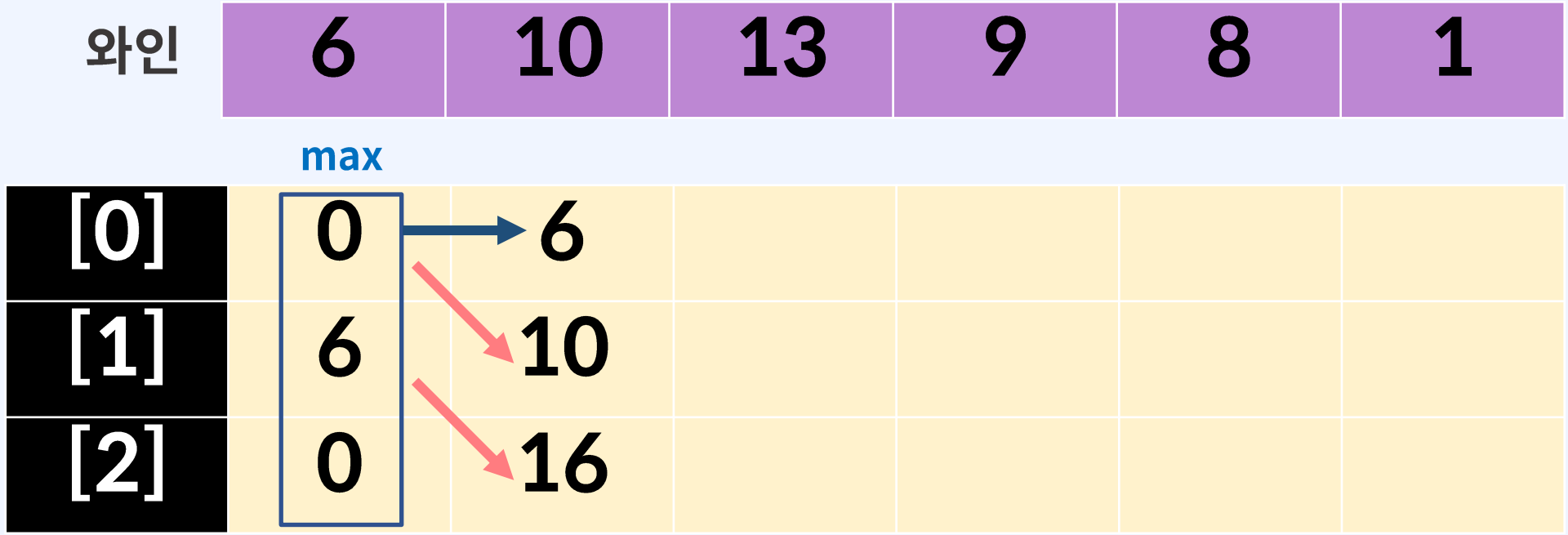
BOJ2156: 포도주 시식

문제 분석

- [case 0] {N}번을 선택하지 않은 경우
  - d[0][n] = max(d[0][n-1], d[1][n-1], d[2][n-1])
- [case 1] {N-1}을 선택하지 않고 {N}번을 선택한 경우
  - d[1][n] = d[0][n-1] + wine[n]
- [case 2] {N-1}을 선택하고, {N}번을 선택한 경우
  - d[2][n] = d[1][n-1] + wine[n]

BOJ2156: 포도주 시식

구현



BOJ2156: 포도주 시식

구현

와인	6	10	13	9	8	1
		max				
[0]	0	6	16			
[1]	6	10	19			
[2]	0	16	23			

BOJ2156: 포도주 시식

구현

와인	6	10	13	9	8	1
			max			
[0]	0	6	16	23		
[1]	6	10	19	25		
[2]	0	16	23	28		

BOJ2156: 포도주 시식

구현

와인	6	10	13	9	8	1
				max		
[0]	0	6	16	23	28	
[1]	6	10	19	25	31	
[2]	0	16	23	28	33	

BOJ2156: 포도주 시식

구현

와인	6	10	13	9	8	1
					max	
[0]	0	6	16	23	28 →	33
[1]	6	10	19	25	31 ↘	29
[2]	0	16	23	28	33 ↘	32

BOJ2156: 포도주 시식

구현

와인	6	10	13	9	8	1
						max
[0]	0	6	16	23	28	33
[1]	6	10	19	25	31	29
[2]	0	16	23	28	33	32

최종 정답은 N번째 잔을 고르지 않을 수도 있으므로
d[*][n] 중에 가장 큰 값을 출력한다

BOJ2156: 포도주 시식

구현

```
d[0][1] = 0;
d[1][1] = wine[1];
d[2][1] = wine[1];
for(int i = 2; i <= n; i++) {
    d[0][i] = Math.max(d[0][i - 1],
Math.max(d[1][i - 1], d[2][i - 1]));
    d[1][i] = d[0][i - 1] + wine[i];
    d[2][i] = d[1][i - 1] + wine[i];
}
```

Ch04. 동적 계획법 #1

4. [11047] 동전 0

BOJ11047: 동전 0

문제 요약

- N종류의 동전이 주어짐 (1<= N <= 10)
- 동전을 조합해서 K가치를 만들어야 함 (1 <= K <= 1억)
- 동전의 최소 개수를 구하기
- 단, 동전은 배수관계로 주어짐

BOJ11047: 동전 0

문제 요약

입력 데이터
10 4200 1 5 10 50 100 500 1000 5000 10000 50000

출력 데이터
6

1000 * 4
100 * 2

BOJ11047: 동전 0

문제 분석

- 단, 동전은 배수관계로 주어짐

- 위 조건에 의해 그리디로 풀이가 가능함
  - 이후의 동전 1, 2 문제에서 동적계획법으로 접근

- 왜 그리디가 가능할까?

BOJ11047: 동전 0

문제 분석

- 5원
- 10원
- 20원
- 40원

- {i}번째 동전과 {i+1}번째 동전이 배수관계라면?
- 무조건 큰 액수의 동전을 먼저 사용하는 것이 좋다
- {i}번째 동전으로 만들 수 있는 액수는 {i-1}, {i-2}번째 동전으로도 만들 수 있다

BOJ11047: 동전 0

구현

- 동전은 큰 액수부터 선택하여 사용
- 잔액은 동전의 액수로 나눈 몫만큼 사용
  - 나머지 값보다 작거나 같은 액수의 동전을 골라서 위 과정을 반복

BOJ11047: 동전 0

구현

```
int count = 0;
for (int i = n - 1; i >= 0; i--) {
    if (k >= coin[i]) {
        count += k / coin[i];
        k %= coin[i];
    }
}
```

- 입력이 오름차순 정렬되어 있으므로 역순으로 선택
- 동전이 잔액을 초과하면 안되므로 몫만큼 사용
- 남은 금액은 작거나 같은 동전을 찾아 위의 과정을 반복

Ch04. 동적 계획법 #1

5. [2294] 동전 2

BOJ2294: 동전 2

문제 요약

- N종류의 동전이 주어짐 (1<= N <= 100)
- 동전을 조합해서 K가치를 만들어야 함 (1 <= K <= 10,000)
- 동전의 최소 개수를 구하기 (불가능하면 -1 출력)

BOJ2294: 동전 2

문제 분석

- 문제 요구사항 : k 원을 만드는 동전의 최소 개수
- d[k]: 동전의 최소 개수
  - 점화 관계를 어떻게 작성해야 할까?

BOJ2294: 동전 2

문제 분석

- d[k]: 동전의 최소 개수
  - d[1..k-1] 까지의 결과를 알고 있다고 가정

- 방법1:
  - 이미 만든 금액을 {i}로 취급
  - dp[i + coin[j]] = min(dp[i + coin[j]], dp[i] + 1)
- 방법2:
  - 만들려는 금액을 {i}로 취급
  - dp[i] = min(dp[i], dp[i - coin[j]] + 1)

BOJ2294: 동전 2

문제 분석

방법1

- 이미 만든 금액을 {i}로 취급
- 만들 수 있는 금액에 동전의 액수를 더해서 새로운 경우 생성
- 이미 만든 적이 있는 방법과 비교해서 최솟값 선택
- dp[i + coin[j]] = min(dp[i + coin[j]], dp[i] + 1)

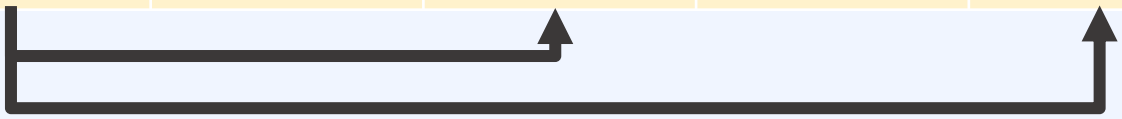
BOJ2294: 동전 2

문제 분석

- 방법1:
  - dp[i + coin[j]] = min(dp[i + coin[j]], dp[i] + 1)

- ex) 5원을 만드는 최소 개수가 2개를 알고 있을 때
  3원과 10원 동전을 들고 있는 경우

	…	5	…	$8_{(5+3)}$	…	$15_{(5+10)}$
dp[]		2		2 + 1		2 + 1



BOJ2294: 동전 2

문제 분석

방법2

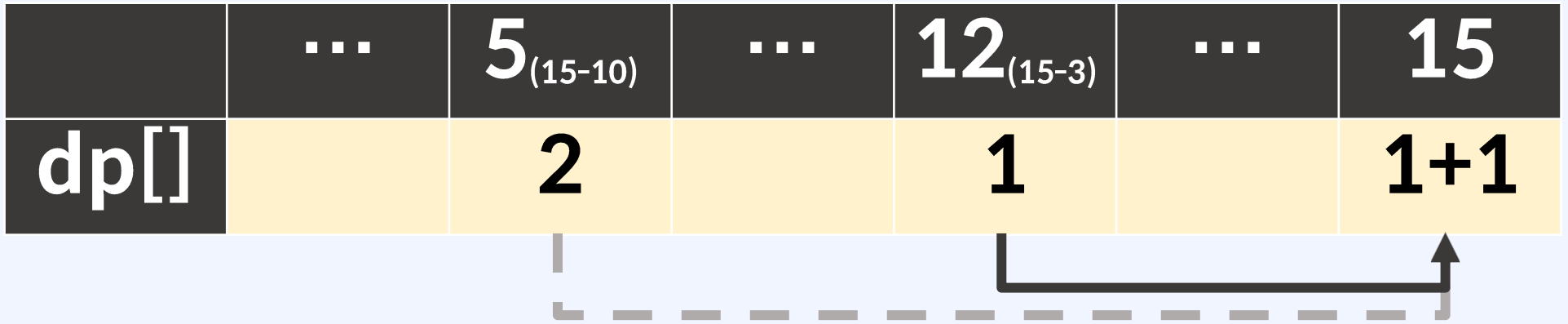
- 만들려는 금액을 {i}로 취급
- {i}에서 동전의 액수를 뺀 금액에서 경우의 수를 가져오기
- 이미 만든 적이 있는 방법과 비교해서 최솟값 선택
- dp[i] = min(dp[i], dp[i - coin[j]] + 1)

BOJ2294: 동전 2

문제 분석

- 방법2
  - dp[i] = min(dp[i], dp[i - coin[j]] + 1);

- ex) 15원을 만들려고 할때
  3원과 10원 동전을 들고 있는 경우



BOJ2294: 동전 2

구현

```
for (int i = 1; i <= k; i++) {
    dp[i] = 100001;
}

for (int i = 0; i < n; i++) {
    coin[i] = sc.nextInt();
    if (coin[i] <= k)
        dp[coin[i]] = 1;
}
```

초기화

- dp배열에 최대 액수보다 큰 값을 넣는다

- 가지고 있는 동전에는 경우의 수에 1을 넣는다

BOJ2294: 동전 2

구현

방법 1

```
for (int i = 1; i <= k; i++) {
    for (int j = 0; j < n; j++) {
        if (i + coin[j] <= k)
            dp[i + coin[j]] = Math.min(dp[i + coin[j]], dp[i] + 1);
    }
}
```

방법 2

```
for (int i = 1; i <= k; i++) {
    for (int j = 0; j < n; j++) {
        if (i - coin[j] >= 0)
            dp[i] = Math.min(dp[i], dp[i - coin[j]] + 1);
    }
}
```

Ch04. 동적 계획법 #1

6. [2293] 동전 1

BOJ2293: 동전 1

문제 요약

- N종류의 동전이 주어짐 (1<= N <= 100)
- 동전을 조합해서 K가치를 만들어야 함 (1 <= K <= 10,000)
- K가치를 만드는 경우의 수를 구하기

BOJ2293: 동전 1

문제 분석

- 동전 1과 유사한 문제
- 문제 요구사항: k원을 만드는 경우의 수
  - d[k]: 경우의 수
- 동일하게 두가지 방법으로 접근 가능하다

BOJ2293: 동전 1

문제 분석

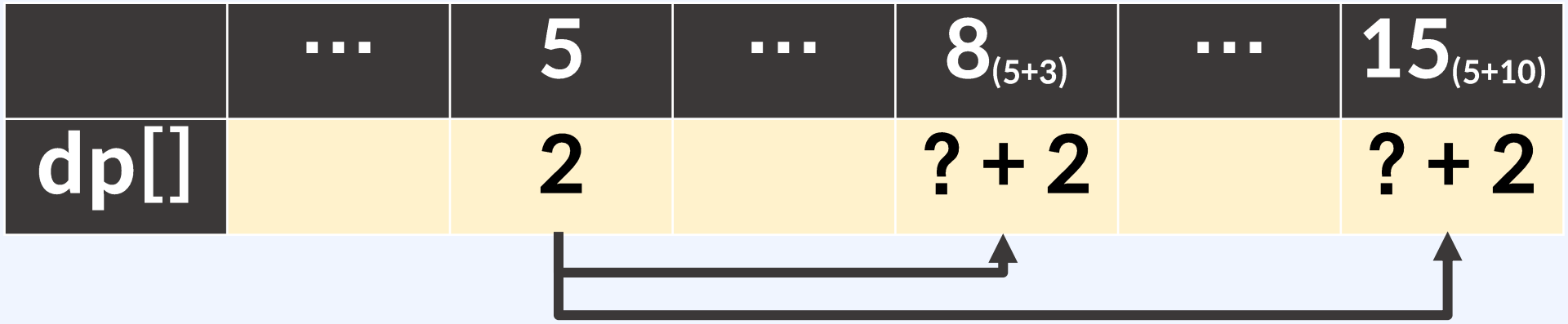
- d[k]: 경우의 수
  - d[1..k-1] 까지의 결과를 알고 있다고 가정
- 방법1:
  - 이미 구한 경우의 수를 {j}로 취급
  - dp[j + coin[i]] += dp[j]
- 방법2:
  - 구하려는 경우의 수를 {j}로 취급
  - dp[j] += dp[j – coin[i]]

BOJ2293: 동전 1

문제 분석

- 방법1
  - dp[j + coin[i]] += dp[j]

- ex) 5원을 만드는 경우의 수가 있을 때
  3원과 10원 동전을 들고 있는 경우

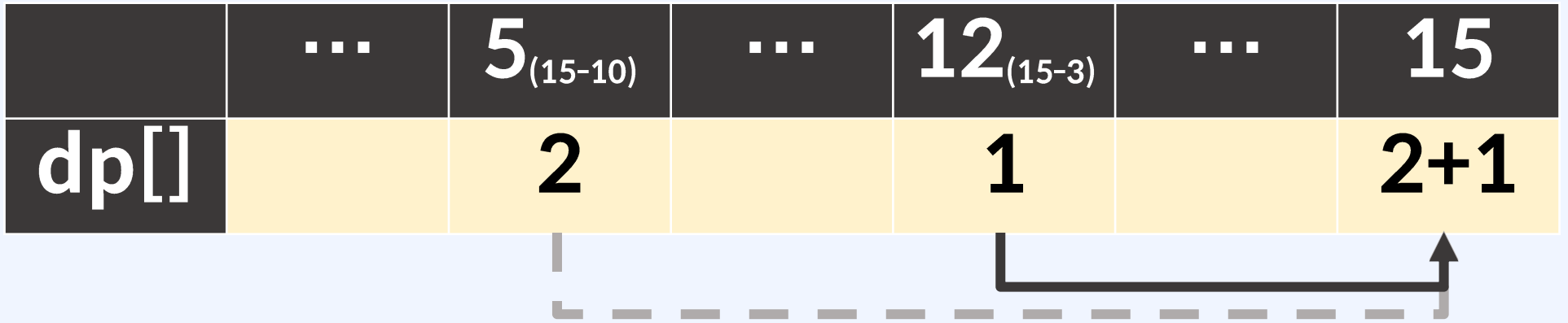


BOJ2293: 동전 1

문제 분석

- 방법2
  - dp[j] += dp[j – coin[i]]

- ex) 15원을 만드는 경우의 수를 찾으려는 경우
  3원과 10원 동전을 들고 있는 경우



BOJ2293: 동전 1

구현

방법 1

```
for(int i = 0; i < n; i++) {
    for(int j = 0; j + coin[i] <= k; j++) {
        dp[j + coin[i]] += dp[j];
    }
}
```

방법 2

```
for(int i = 0; i < n; i++) {
    for(int j = coin[i]; j <= k; j++) {
        dp[j] += dp[j - coin[i]];
    }
}
```

Ch04. 동적 계획법 #1

7. [2624] 동전 바꿔주기

BOJ2624: 동전 바꿔주기

문제 요약

- T원의 지폐를 동전으로 바꿔줘야 함
  - 1 <= T <= 10,000
- 동전마다 개수 제한이 있음 (n: 동전 개수, k: 동전 가지 수)
  - (1 <= n <= 1,000) (1 <= k <= 100)
- 교환 가능한 경우의 수

BOJ2624: 동전 바꿔주기

문제 분석

- 직전에 클립에서 풀이한 '동전1' 과 유사한 문제
- 동전 사용가능 개수에 제한이 있음
- 문제 요구 사항: 동전 교환 방법의 가지 수
  - d[t] = t원을 교환하는 방법 수
- 동적배열의 차원을 늘려 정보를 추가해보자

BOJ2624: 동전 바꿔주기

문제 분석

- 문제 요구 사항: 동전 교환 방법의 가지 수
  - d[?][t] = t원을 교환하는 방법 수

- 동전은 k개가 주어진다.
  - {i}번째 동전을 고려할 때, {i-1}번째 동전까지 사용한 경우에서 값을 가져와 생각해볼 수 있다

- d[k][t] = k번째 동전까지 사용했을 때 t원을 교환하는 방법의 수

BOJ2624: 동전 바꿔주기

문제 분석

- d[k][t] = k번째 동전까지 사용했을 때 t원을 교환하는 방법의 수

---

ex)

- 3번째 동전을 사용했을 때, {1} {2} {5} {10} 원을 만들 수 있었고
- 4번째 동전의 가치가 v원이라면?
  - {1 + v}, {2 + v}, {5 + v}, {10 + v} 원을 만들 수 있다
  - 경우의 수는 3번째 동전에서 계산한 횟수를 그대로 사용한다

d[4][1 + v] += d[3][1]
d[4][2 + v] += d[3][2]
d[4][5 + v] += d[3][5]
d[4][10 + v] += d[3][10]

BOJ2624: 동전 바꿔주기

문제 분석

- d[k][t] = k번째 동전까지 사용했을 때 t원을 교환하는 방법의 수

- 단, 동전의 개수에 제한이 있으므로, 반복문을 통해 제한한다

```
for (int cnt = 0; cnt <= coinCount; cnt++) {
    int nextValue = value + coinPrice * cnt;
    // 점화식 작성
}
```

BOJ2624: 동전 바꿔주기

구현

```
for (int i = 1; i <= k; i++) {
    int coinPrice = sc.nextInt();
    int coinCount = sc.nextInt();
    for (int value = 0; value <= t; value++) {
        for (int cnt = 0; cnt <= coinCount; cnt++) {
            int nextValue = value + coinPrice * cnt;
            if (nextValue > t) break;
            dp[i][nextValue] += dp[i - 1][value];
        }
    }
}
```

- 매 동전을 입력 받으면서
- {i-1}번째 동전을 이용해서 만든 가치를 경우의 수를 {i}번째 동전에 반영해 준다
- 단 만들려는 가치가 최대가치 t 를 넘지 않도록 제한한다

BOJ2624: 동전 바꿔주기

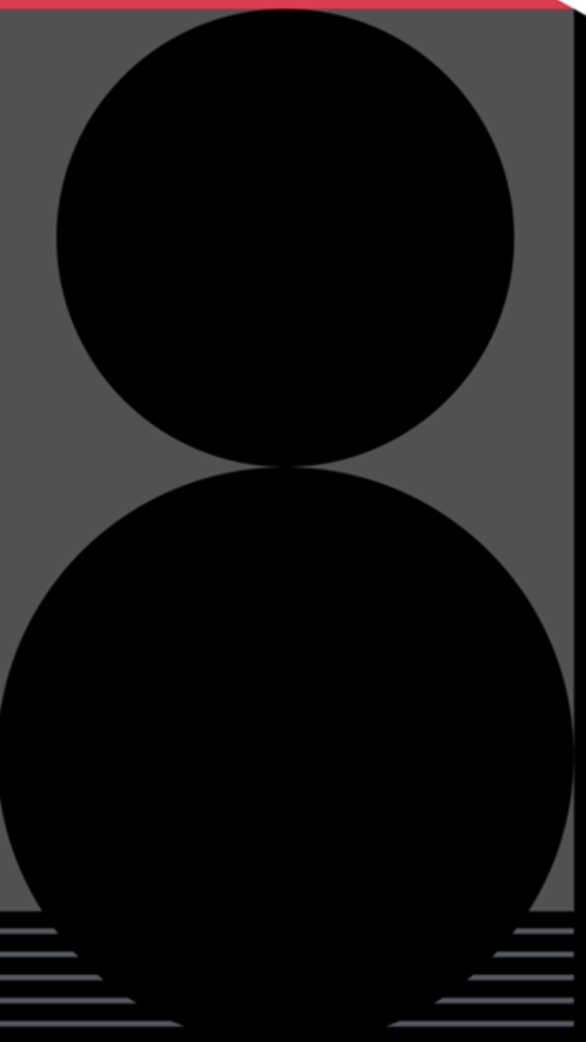
최적화

```
int nextValue = value + coinPrice * cnt;
dp[i][nextValue] += dp[i - 1][value];
```

- 동적 배열의 차원 수를 줄일 수 있을까?
  - {i}번째 동전을 사용한 가치는 오직 {i-1} 번째 동전만 참조한다
  - value 변수는 동전의 종류와 상관 없이 0부터 t까지 모든 범위를 갱신한다

- 따라서 특정 가치에 대한 경우의 수를, 동전단위로 제한하지 않고
  한번에 계산해도 동일한 결과를 얻을 수 있다

```
for (int j = t; j >= 0; j--) {
    for (int cnt = 1; cnt <= coinCount; cnt++) {
        int nextValue = j + coinPrice * cnt;
        if (nextValue > t) break;
        d[nextValue] += d[j];
    }
}
```



Ch04. 동적 계획법 #1

8. [12865] 평범한 배낭

BOJ12865: 평범한 배낭

문제 요약

- 배낭에 최대한 가치 있는 물건을 넣기
  - 물건의 수 1 <= N <= 100

- 각 물건은 {무게 W} {가치 V} 상태를 가짐
  - (1 <= W <= 100,000) (0 <= V <= 1,000)

- 배낭에는 최대 K 무게까지 담을 수 있음
  - 1 <= K <= 100,000

BOJ12865: 평범한 배낭

문제 분석

- 물건을 분할해서 담을 수 없다
  (가치 / 무게) 순으로 배낭에 담으면, 빈 공간이 생길 수 있다

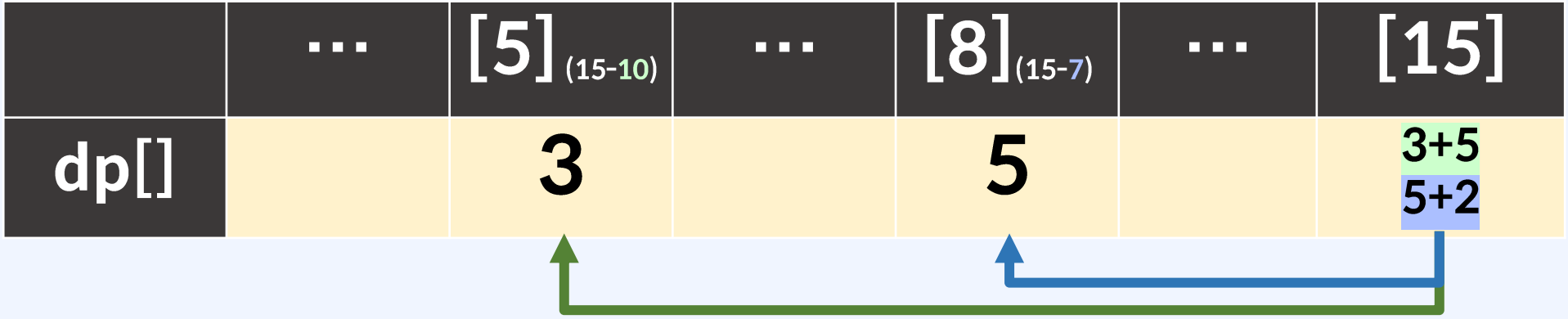
- 무게가 넘지 않으면서
  최대한의 가치를 가지도록 생각해야 한다

- 문제 요구 사항: 배낭에 담긴 물건들의 최대 가치
  - d[k] = 무게를 k까지 담았을 때 최대 가치

BOJ12865: 평범한 배낭

문제 분석

- 만들려는 무게를 기준으로
- 새로 들어오는 물건의 무게를 배열의 인덱스에 더하고
- 가치를 배열의 값에 더해 최대 값 갱신이 되는지 체크한다



BOJ12865: 평범한 배낭

구현

```
int dp[] = new int[k + 1];
for(int i = 0; i < n; i++) {
    int w = sc.nextInt();
    int v = sc.nextInt();
    for(int j = k; j >= w; j--) {
        dp[j] = Math.max(dp[j], dp[j - w] + v);
    }
}
```

현재 입력받은 물건의
무게: w, 가치: v 일 때

{j} 무게를 만들기 위해 w만큼
공간이 비어있는 상태의
최대 가치에 v를 더한다