Dynamic Programming Basic - 1

DP 기초 - 1

競技プログラミングの鉄則

KPSC Algorithm Study 24/12/19 Thu.

by Haru_101

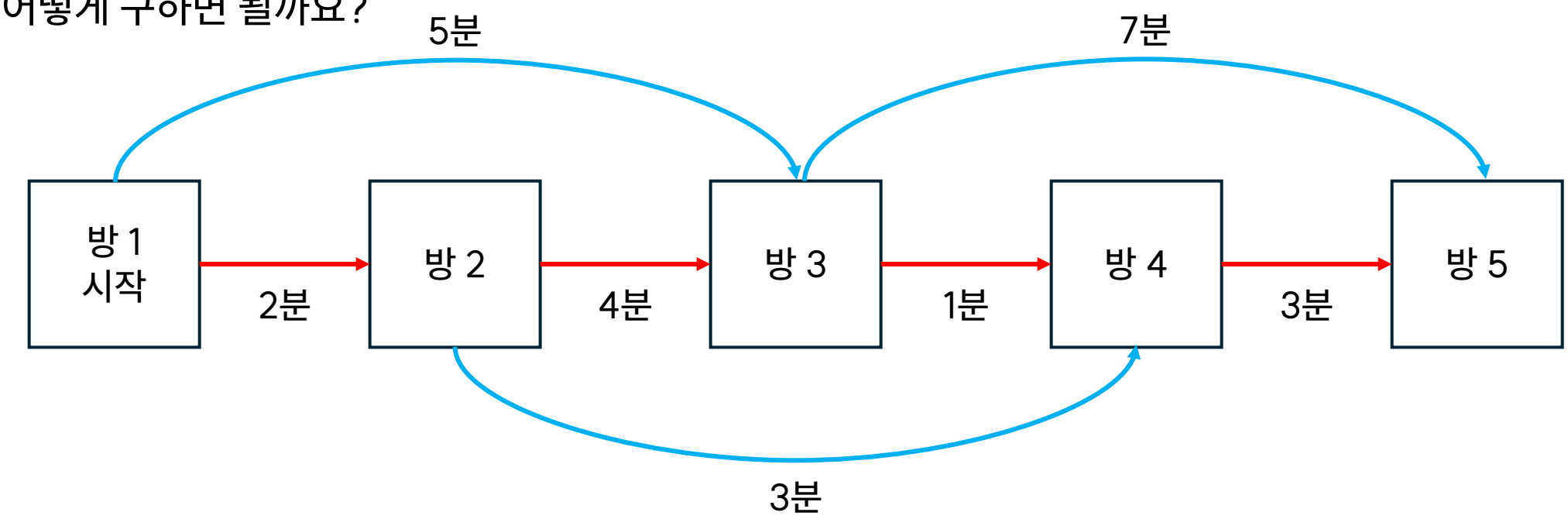
Dynamic Programming

- 다이나믹 프로그래밍(DP)는 작은 문제의 결과를 이용해 문제를 해결하는 방법을 얘기합니다.
- 분할정복과는 살짝 다릅니다.
- 다음 예를 보면서 DP가 어떤 식으로 작동하는지 한번 살펴봅시다.

Dynamic Programming

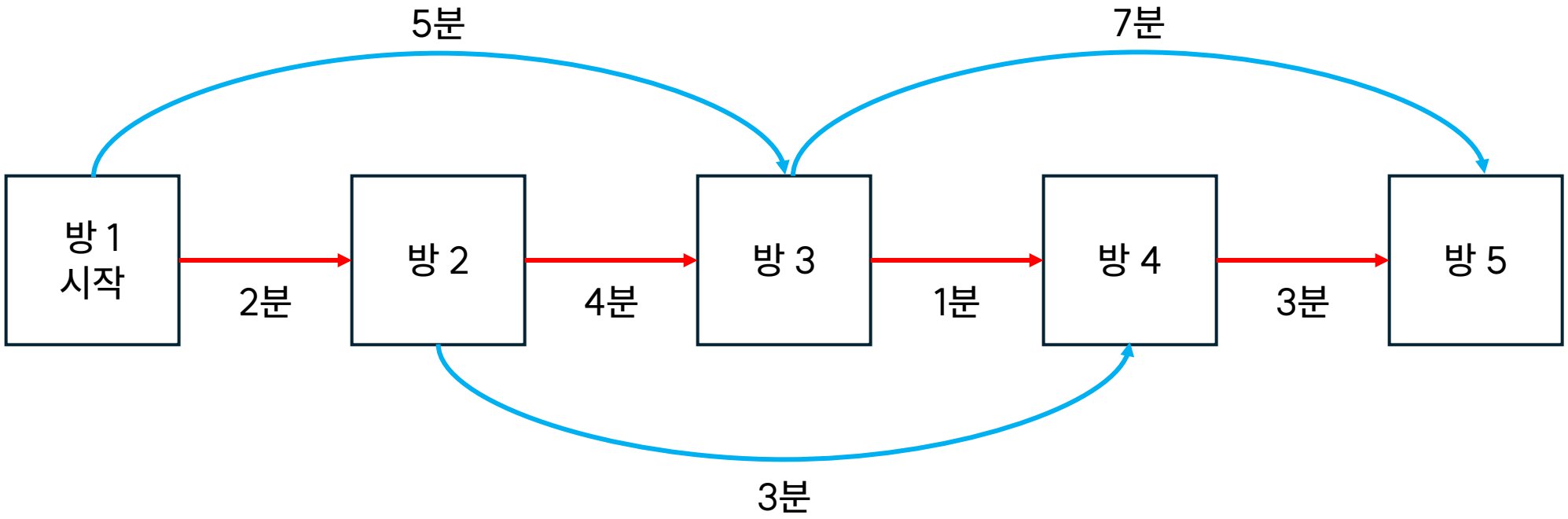
- 방 1에서 시작해서 방 5까지 가능한 여러 경로 중에서 최단 시간을 구하고 싶습니다.

- 어떻게 구하면 될까요?



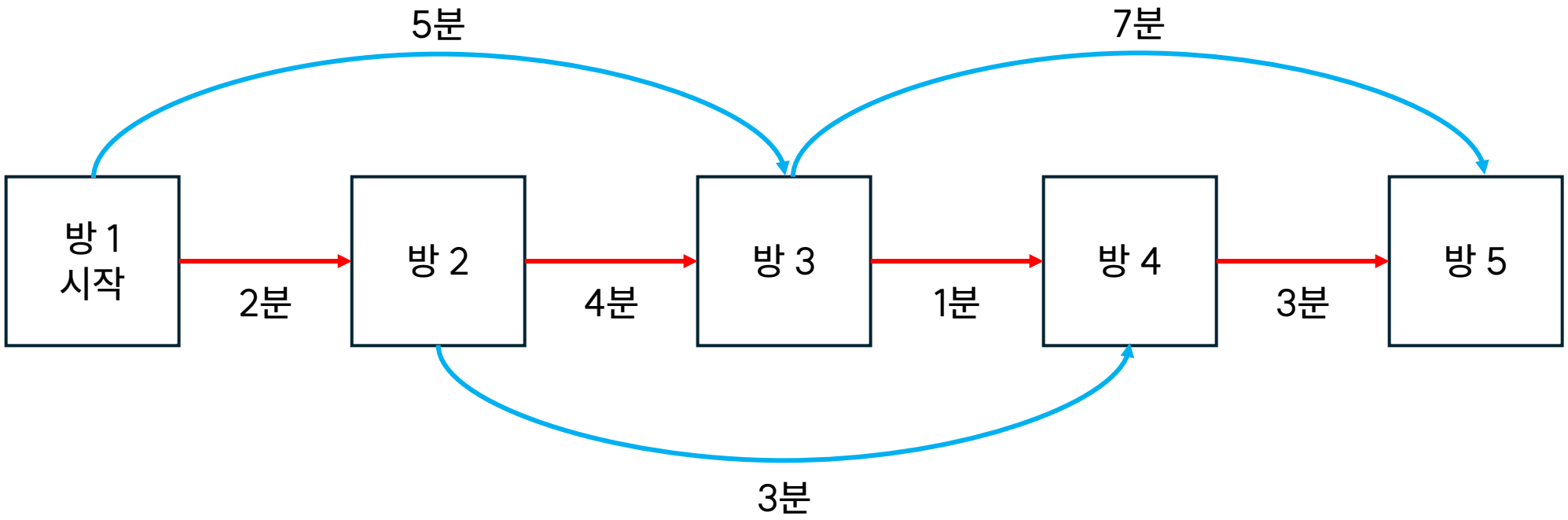
Dynamic Programming

- 가능한 경로를 나열해보면, 1->3->5, 1->2->3->4->5, 1->2->4->5, ...과 같은 경로가 가능합니다.
- 이 경로를 모두 나열하는 것은 쉽지 않아보입니다. (깊이 우선 탐색으로 나열할 수 있겠으나 일단 생략합니다.)



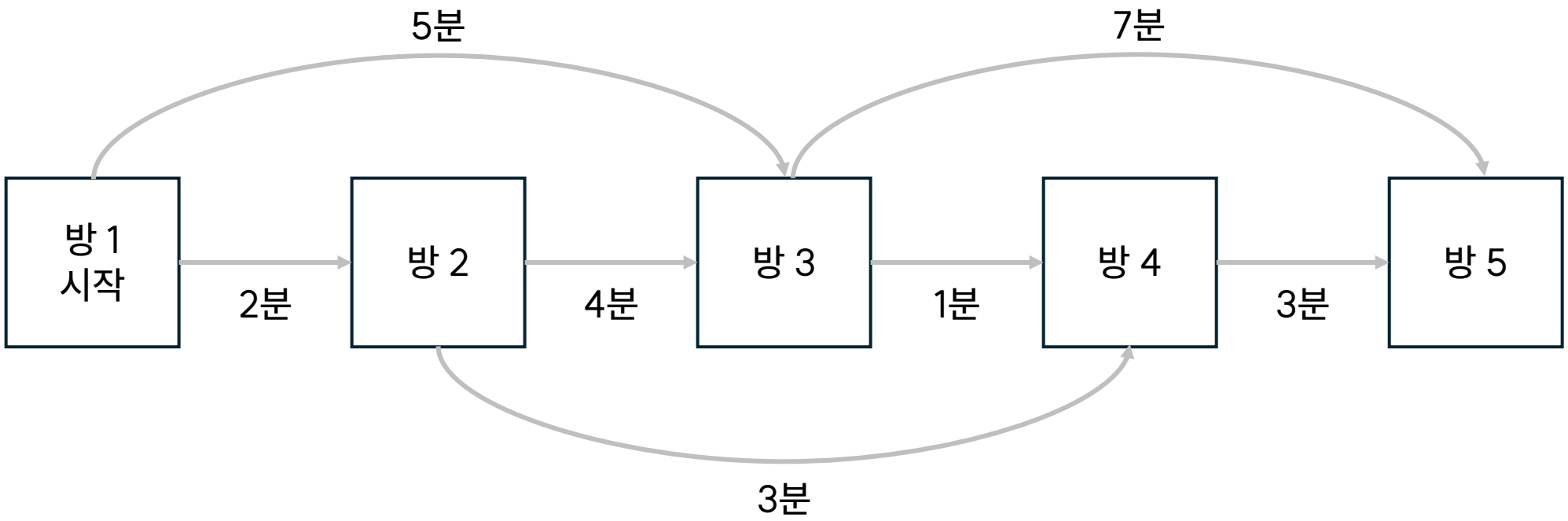
Dynamic Programming

- 여기서 사용할 수 있는 기법이 DP입니다.
- DP[i]를 방 i까지 가는 데에 걸리는 최단 시간이라고 정의합시다.



Dynamic Programming

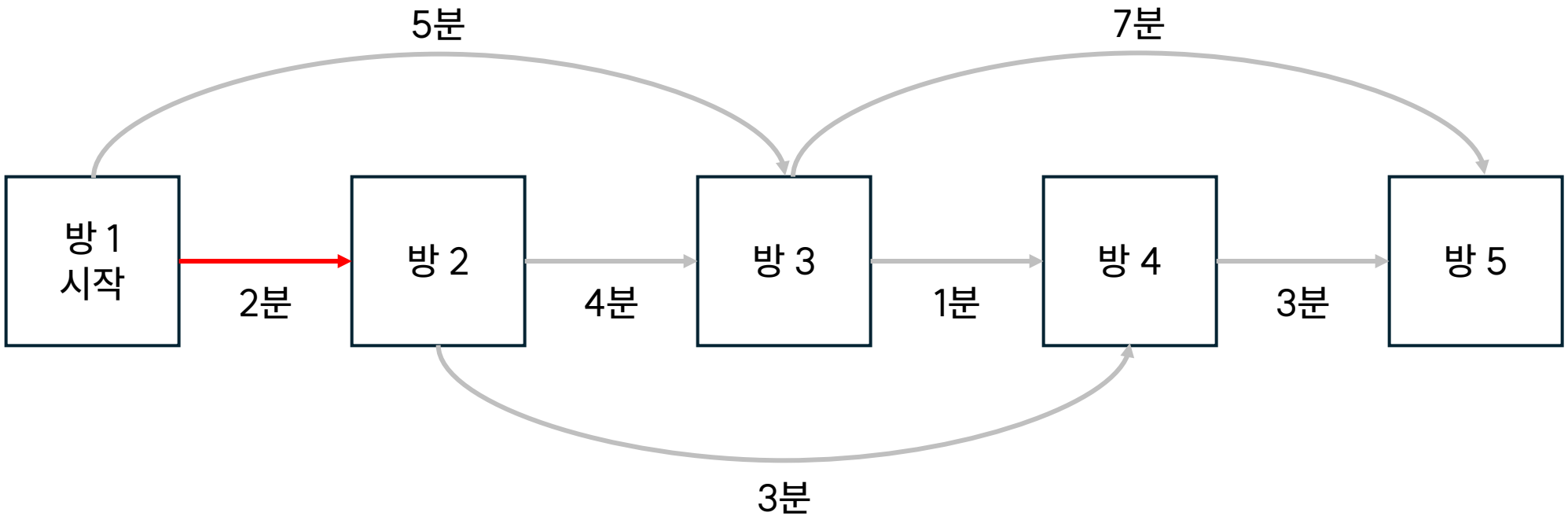
- 먼저 방 1에서 출발을 하니 DP[1]은 0이 됩니다. (출발지까지의 최단시간은 0)



DP	0	INF	INF	INF	INF

Dynamic Programming

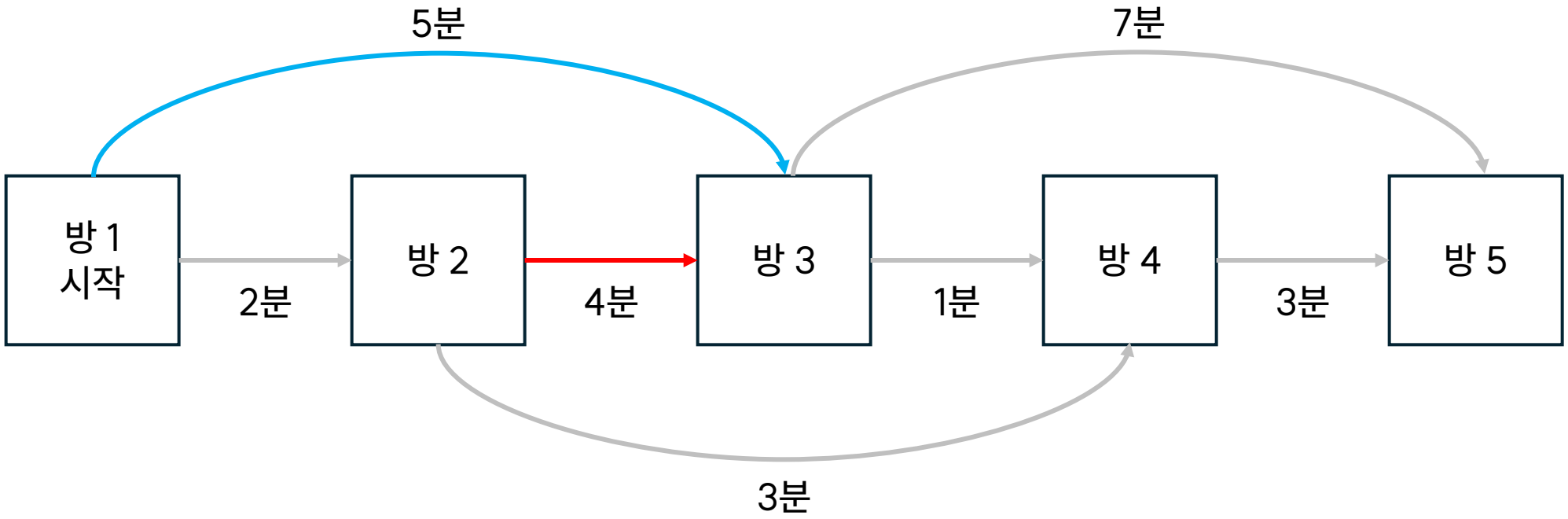
- 그 다음, 방 2의 관점에서 보았을 때, 자신에게 들어오는 화살표는 1->2 (2분) 단 하나입니다.
- 따라서 DP[2] = min(DP[2], DP[1]+2) = 2가 됩니다.



DP	0	2	INF	INF	INF

Dynamic Programming

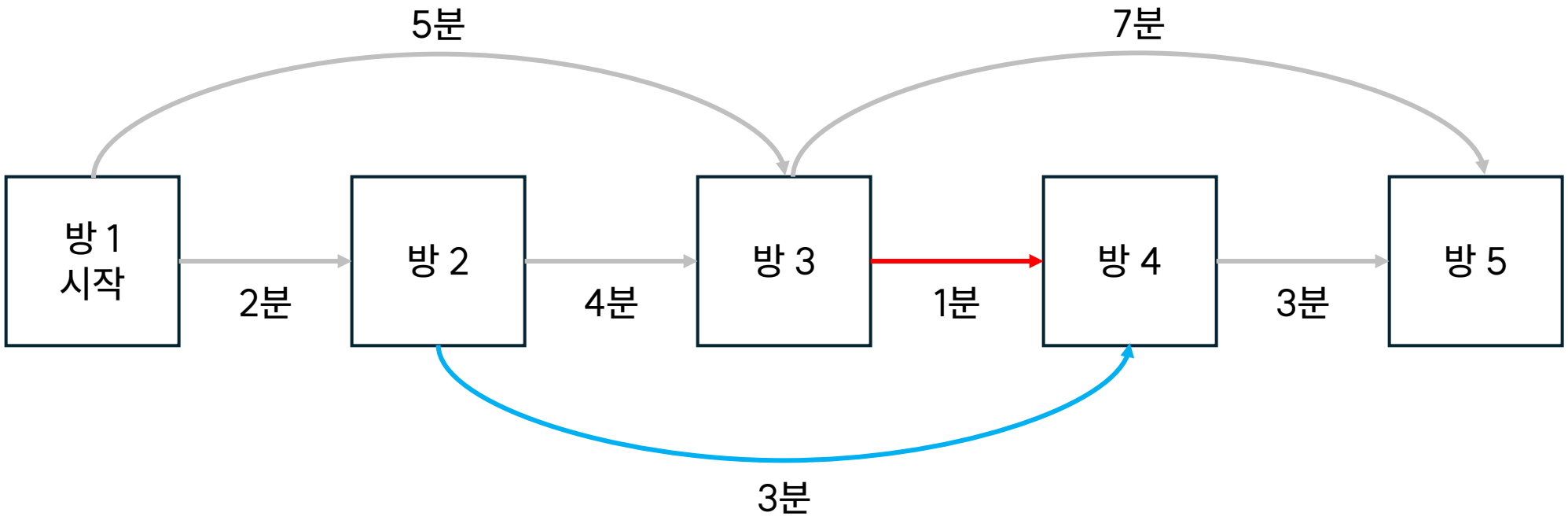
- 그 다음, 방 3의 관점에서 보았을 때, 자신에게 들어오는 화살표는 2->3 (4분), 1->3 (5분) 총 2개입니다.
- 따라서 DP[3] = min(DP[3], DP[2]+4, DP[1]+5) = min(INF, 6, 5) = 5가 됩니다.



DP	0	2	5	INF	INF

Dynamic Programming

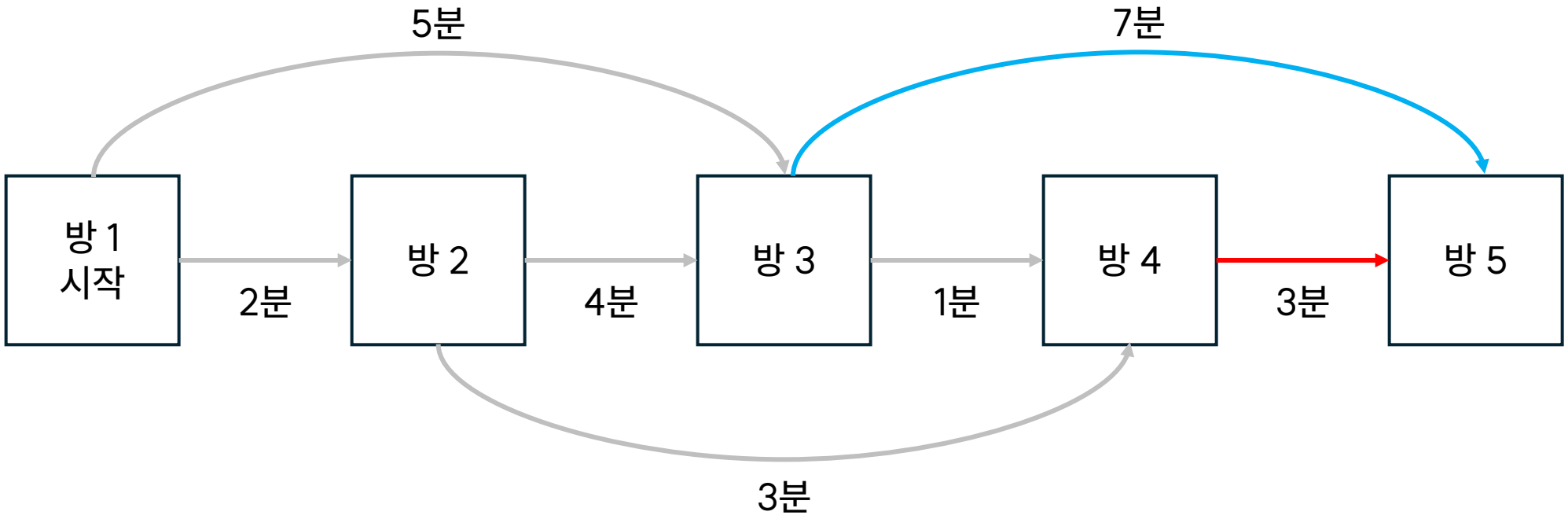
- 그 다음, 방 4의 관점에서 보았을 때, 자신에게 들어오는 화살표는 3->4 (1분), 2->4 (3분) 총 2개입니다.
- 따라서 DP[4] = min(DP[4], DP[3]+1, DP[2]+3) = min(INF, 6, 5) = 5가 됩니다.



DP	0	2	5	5	INF

Dynamic Programming

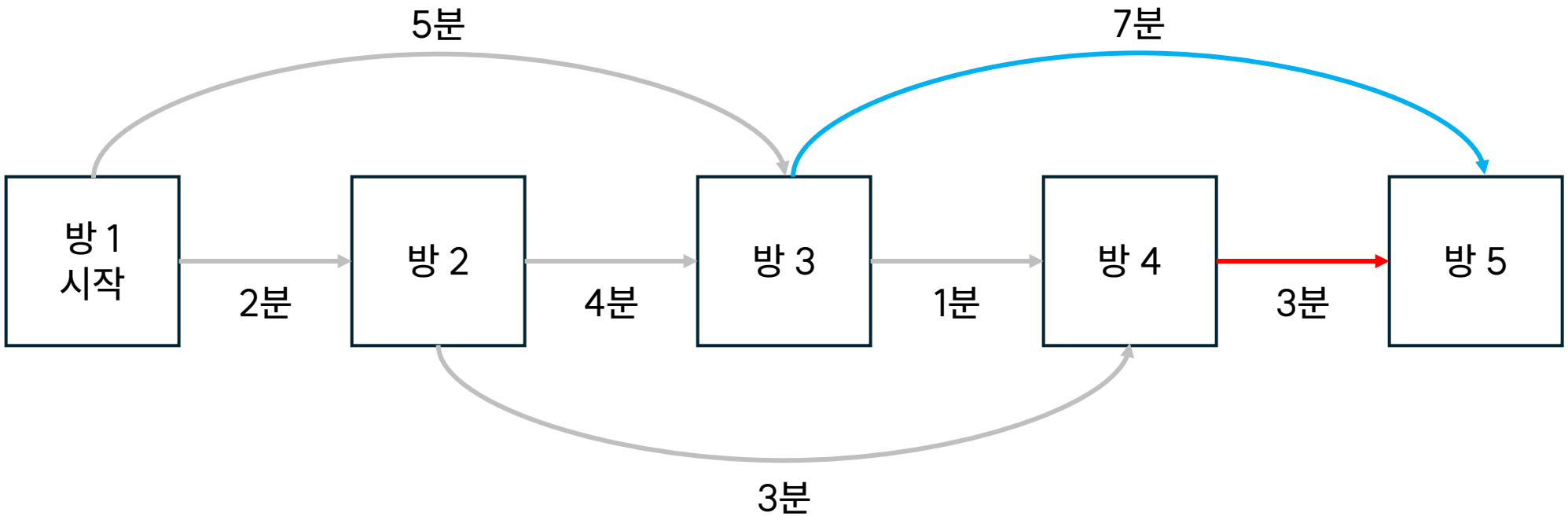
- 그 다음, 방 5의 관점에서 보았을 때, 자신에게 들어오는 화살표는 4->5 (3분), 3->5 (7분) 총 2개입니다.
- 따라서 DP[5] = min(DP[5], DP[4]+3, DP[3]+7) = min(INF, 8, 12) = 8이 됩니다.



DP	0	2	5	5	8

Dynamic Programming

- 우리가 찾고자 한 값이 방 5까지의 최단 시간이므로 DP[5]를 출력해주면 됩니다.
  (DP를 그렇게 정의했으므로)



DP	0	2	5	5	8

Dynamic Programming

- 이제 대충 어찌저찌 하는지는 감이 오실거 같으니 다른 문제들을 풀면서 DP의 세계에 입문해봅시다.

Dynamic Programming

- 아래 문제를 풀어봅시다.

問題文

あるダンジョンには N 個の部屋があり、1 から N までの番号が付けられています。このダンジョンは**一方通行**であり、通路を介して 1 つ先または 2 つ先の部屋に移動することができます。各通路における移動時間は以下の通りです。

- 部屋 $i-1$ から部屋 i に向かう通路を通るのに A_i 分 $(2 \leq i \leq N)$ かかる。
- 部屋 $i-2$ から部屋 i に向かう通路を通るのに B_i 分 $(3 \leq i \leq N)$ かかる。

太郎君が部屋 1 から部屋 N に移動するのに、最短何分かかりますか。答えを求めるプログラムを作成してください。

$$
\begin{array}{llll}
N & & & \\
A_2 & A_3 & A_4 & \cdots \quad A_N \\
B_3 & B_4 & \cdots & B_N
\end{array}
$$

制約

- $3 \leq N \leq 100\,000$
- $1 \leq A_i \leq 100\ (2 \leq i \leq N)$
- $1 \leq B_i \leq 100\ (3 \leq i \leq N)$
- 入力される値はすべて整数である。

- 어떤 던전에는 N개의 방이 있고, 방은 1부터 N까지의 번호가 붙여져 있습니다. 이 던전은 일방통행이며, 통로를 통해 다음 방 혹은 다다음 방으로 이동할 수 있습니다.
  - A_i : 방 $i-1$에서 방 i로 이동하는 데에 걸리는 시간
  - B_i : 방 $i-2$에서 방 i로 이동하는 데에 걸리는 시간
- 이때 방 1에서 방 N까지 이동하는 데에 걸리는 시간을 출력하세요.

Dynamic Programming

- 이 문제는 앞에 설명한 문제입니다.
- 따라서 이를 바로 코드로 짜면 다음과 같습니다.

```
int dp[100005];
int main() {
    fastio();
    int N;
    cin >> N;
    for(int i=2; i<=N; i++) {
        cin >> A[i];
    }
    for(int i=3; i<=N; i++) {
        cin >> B[i];
    }
    for(int i=2; i<=N; i++) {
        if(i==2) {
            dp[i] = A[i];
        } else {
            dp[i] = min(dp[i-1] + A[i], dp[i-2] + B[i]);
        }
    }
    cout << dp[N];
}
```

Dynamic Programming

- 아래 문제를 풀어봅시다.

問題文

あるダンジョンには N 個の部屋があり、1 から N までの番号が付けられています。このダンジョンは**一方通行**であり、通路を介して 1 つ先または 2 つ先の部屋に移動することができます。各通路における移動時間は以下の通りです。

- 部屋 $i-1$ から部屋 i に向かう通路を通るのに A_i 分 $(2 \leq i \leq N)$ かかる。
- 部屋 $i-2$ から部屋 i に向かう通路を通るのに B_i 分 $(3 \leq i \leq N)$ かかる。

太郎君が部屋 1 から部屋 N へ**最短時間で移動する方法**を 1 つ出力するプログラムを作成してください。

制約

- $3 \leq N \leq 100\,000$
- $1 \leq A_i \leq 100\,(2 \leq i \leq N)$
- $1 \leq B_i \leq 100\,(3 \leq i \leq N)$
- 入力される値はすべて整数である。

$$N$$
$$A_2 \quad A_3 \quad A_4 \quad \cdots \quad A_N$$
$$B_3 \quad B_4 \quad \cdots \quad B_N$$

$$K$$
$$P_1 \quad P_2 \quad \cdots \quad P_K$$

- 어떤 던전에는 N개의 방이 있고, 방은 1부터 N까지의 번호가 붙여져 있습니다. 이 던전은 일방통행이며, 통로를 통해 다음 방 혹은 다다음 방으로 이동할 수 있습니다.
  - A_i : 방 $i-1$에서 방 i로 이동하는 데에 걸리는 시간
  - B_i : 방 $i-2$에서 방 i로 이동하는 데에 걸리는 시간
- 이때 방 1에서 방 N까지 최단 시간으로 이동하는 경로에 대해 거쳐야 하는 방의 개수 K와 거쳐야 하는 방을 오름차순으로 출력하세요. 단, $P_1 = 1, P_K = N$

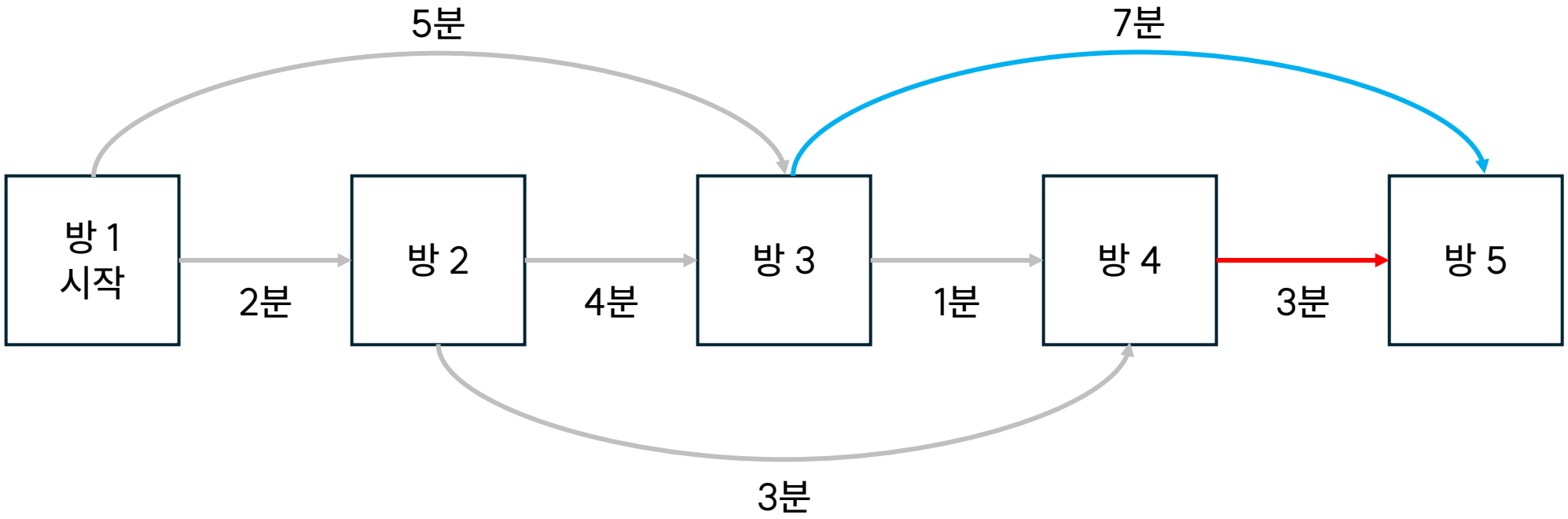
Dynamic Programming

- 방금까지는 DP로 걸리는 시간만 구했는데, 이제는 경로를 출력해야합니다.

- 어떤 식으로 접근할 수 있을까요?

Dynamic Programming

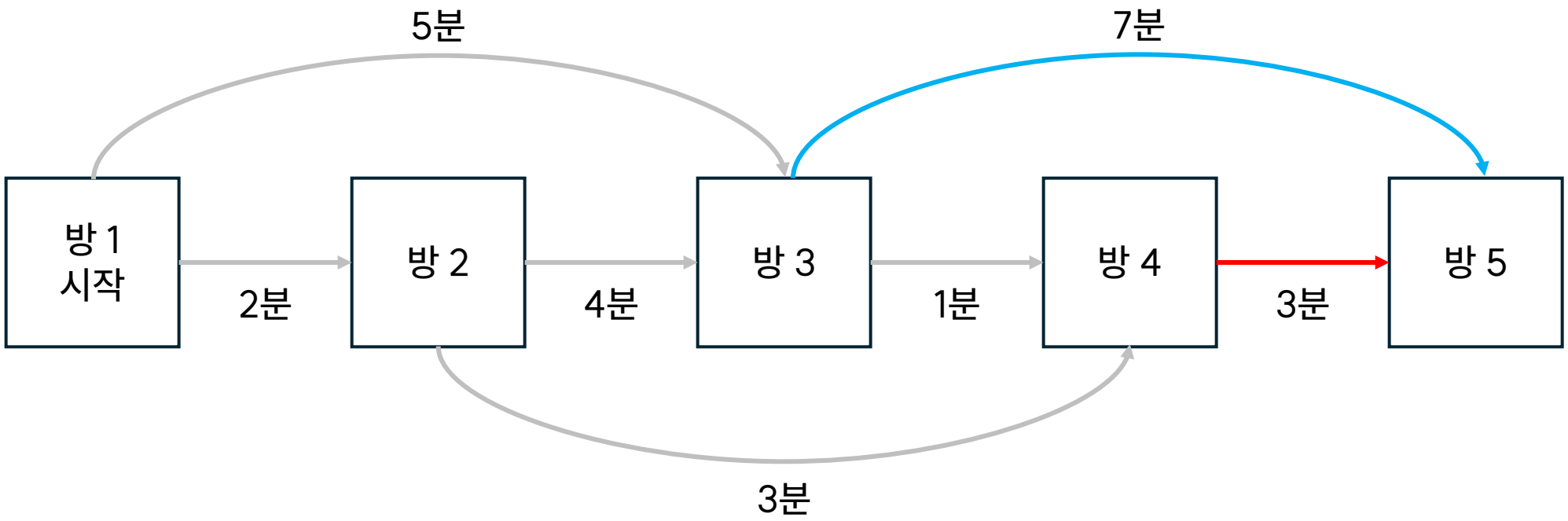
- 이 그림에서 방 5로 가기 위해서는 3->5 (7분), 4->5 (3분) 중 하나의 경로를 거쳐야 하는데, 만약 3->5가 최적 경로였다면, DP[3]은 1이 되어야 합니다.
- 하지만, DP[3]은 5이므로, 3->5는 최적 경로가 아닙니다.



DP	0	2	5	5	8

Dynamic Programming

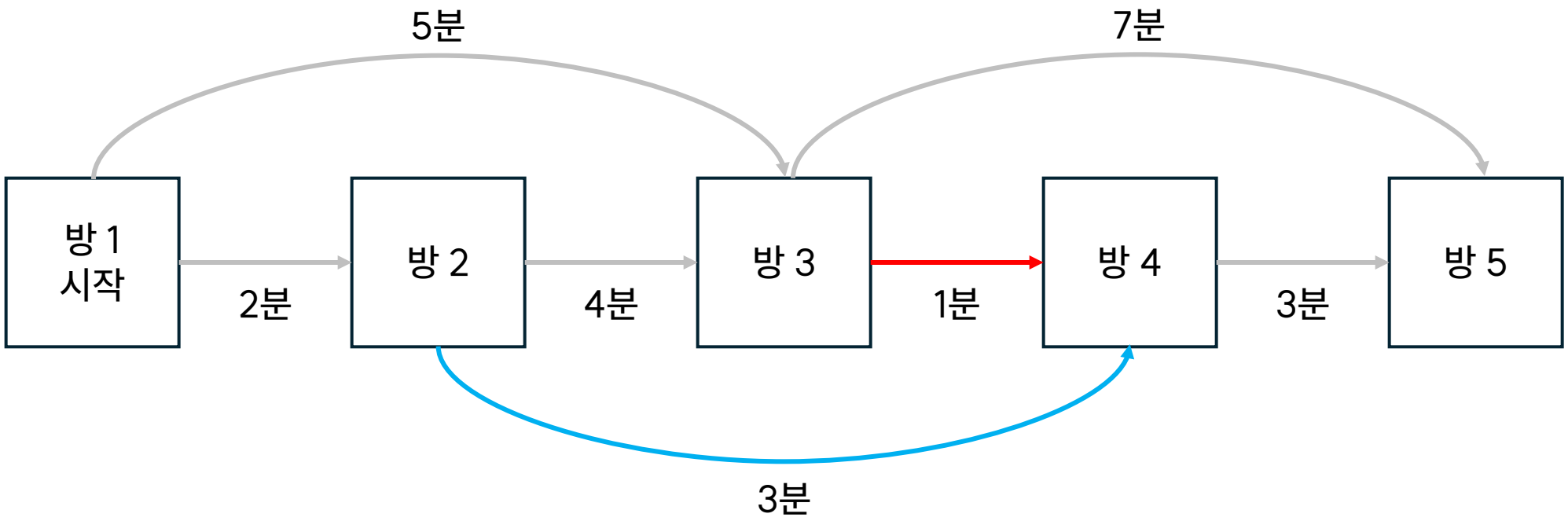
- 따라서 4->5 (3분) 경로가 최적 경로라고 판단할 수 있게 됩니다. (DP[5] = DP[4]+3)



DP	0	2	5	5	8

Dynamic Programming

- 이제 $i = 4$에서도 이 과정을 반복해서 최종적으로 $i = 1$까지 도달해서 거쳐온 경로를 뒤집으면 정답이 됩니다.



DP	0	2	5	5	8

Dynamic Programming

- 코드로 짜면 다음과 같습니다.

```cpp
int A[100005];
int B[100005];
int dp[100005];
int main() {
    fastio();
    int N;
    cin >> N;
    for(int i=2; i<=N; i++) {
        cin >> A[i];
    }
    for(int i=3; i<=N; i++) {
        cin >> B[i];
    }
    dp[2] = A[2];
    for(int i=3; i<=N; i++) {
        dp[i] = min(dp[i-1] + A[i], dp[i-2] + B[i]);
    }
    int idx = N;
    vector<int> ans;
    ans.push_back(N);
    while(idx >= 2) {
        if(idx == 2) {
            break;
        } else {
            if(dp[idx-1] + A[idx] >= dp[idx-2] + B[idx]) {
                ans.push_back(idx-2);
                idx = idx-2;
            } else {
                ans.push_back(idx-1);
                idx = idx-1;
            }
        }
    }
    if(idx != 1) ans.push_back(1);
    cout << ans.size() << '\n';
    for(int i=ans.size()-1; i>=0; i--) {
        cout << ans[i] << ' ';
    }
}
```

Dynamic Programming

- 아래 문제를 풀어봅시다.

問題文

N 枚のカードが一列に並べられており、左から i 番目のカード（以下、カード i とする）には整数 A_i が書かれています。

カードの中からいくつかを選んで、書かれた整数の合計が S となるようにする方法は存在しますか。

制約

- $1 \leq N \leq 60$
- $1 \leq S \leq 10000$
- $1 \leq A_i \leq 10000$
- 入力はすべて整数

$$N \quad S$$
$$A_1 \quad A_2 \quad \cdots \quad A_N$$

- N장의 카드가 일렬로 나열되어 있고, 왼쪽부터 카드 1, ..., 카드 N이라고 할 때, 카드 i에는 A_i가 써져 있습니다.
- 이 카드들 중에서 몇 장을 뽑아서 뽑은 카드들에 써진 수들의 합이 S가 되는 경우가 존재하는지 출력하는 프로그램을 작성하세요.

Dynamic Programming

- 단순히 생각해보면 각 카드를 뽑냐 마냐 두 가지 경우를 각 카드마다 다 시도해보면 되지 않을까요?
- 하지만, 이 방법엔 한계가 있습니다.
- 카드 N장에 대해 뽑냐, 마냐 2가지 경우를 다 시도해본다면 경우의 수는 2^N개가 나옵니다.
- $N = 60$이면 2^{60}번을 시도해야 하기 때문에 시간초과가 당연히 나게 됩니다.

Dynamic Programming

- 이를 DP로 접근하는 방법을 소개하겠습니다.

- $S = 7, A_i = [2, 2, 3]$이라고 합시다.

Dynamic Programming

- DP[i][j]를 카드 i까지 사용해서 합계가 j가 될 수 있는지 여부라고 합시다.

- 먼저 카드 0장으로 0부터 S를 만들 수 있는지 판단해봅시다. 0장으로는 0밖에 불가능합니다.
- 따라서 DP[0][0]=True입니다.

j

DP	0	1	2	3	4	5	6	7	8
0	T								
1									
2									
3									

i

Dynamic Programming

- 그 다음, 카드 1을 사용했을 때 나올 수 있는 합들을 계산해봅시다.
- 사용한 경우에는 모든 j에 대해 DP[i][j]가 True일려면 DP[i-1][j-A_i]가 True여야 합니다.
- 사용하지 않는 경우엔, DP[i][j]가 True이려면 DP[i-1][j]가 True여야 합니다.

j

DP	0	1	2	3	4	5	6	7	8
0	T								
1	T		T						
2									
3									

i

Dynamic Programming

- 그 다음, 카드 2를 사용했을 때 나올 수 있는 합들을 계산해봅시다.
- 사용한 경우에는 모든 j에 대해 DP[i][j]가 True일려면 DP[i-1][j-A_i]가 True여야 합니다.
- 사용하지 않는 경우엔, DP[i][j]가 True이려면 DP[i-1][j]가 True여야 합니다.

j

DP	0	1	2	3	4	5	6	7	8
0	T								
1	T		T						
2	T		T						
3									

i

Dynamic Programming

- 그 다음, 카드 2를 사용했을 때 나올 수 있는 합들을 계산해봅시다.
- 사용한 경우에는 모든 j에 대해 DP[i][j]가 True일려면 DP[i-1][j-A_i]가 True여야 합니다.
- 사용하지 않는 경우엔, DP[i][j]가 True이려면 DP[i-1][j]가 True여야 합니다.

j

DP	0	1	2	3	4	5	6	7	8
0	T								
1	T		T						
2	T		T		T				
3									

i

Dynamic Programming

- 그 다음, 카드 3을 사용했을 때 나올 수 있는 합들을 계산해봅시다.
- 사용한 경우에는 모든 j에 대해 DP[i][j]가 True일려면 DP[i-1][j-A_i]가 True여야 합니다.
- 사용하지 않는 경우엔, DP[i][j]가 True이려면 DP[i-1][j]가 True여야 합니다.

j

DP	0	1	2	3	4	5	6	7	8
0	T								
1	T		T						
2	T		T		T				
3	T			T					

i

Dynamic Programming

- 그 다음, 카드 3을 사용했을 때 나올 수 있는 합들을 계산해봅시다.
- 사용한 경우에는 모든 j에 대해 DP[i][j]가 True일려면 DP[i-1][j-A_i]가 True여야 합니다.
- 사용하지 않는 경우엔, DP[i][j]가 True이려면 DP[i-1][j]가 True여야 합니다.

j

DP	0	1	2	3	4	5	6	7	8
0	T								
1	T		T						
2	T		T		T				
3	T		T	T		T			

i

Dynamic Programming

- 그 다음, 카드 3을 사용했을 때 나올 수 있는 합들을 계산해봅시다.
- 사용한 경우에는 모든 j에 대해 DP[i][j]가 True일려면 DP[i-1][j-A_i]가 True여야 합니다.
- 사용하지 않는 경우엔, DP[i][j]가 True이려면 DP[i-1][j]가 True여야 합니다.

j

DP	0	1	2	3	4	5	6	7	8
0	T								
1	T		T						
2	T		T		T				
3	T		T	T	T	T		T	

i

Dynamic Programming

- 이제 DP[N][S]가 True이면 Yes, 아니면 No를 출력하면 됩니다.
- 시간복잡도는 $O(NS)$입니다.

j

DP	0	1	2	3	4	5	6	7	8
0	T								
1	T		T						
2	T		T		T				
3	T		T	T	T	T		T	

i

Dynamic Programming

- 코드로 짜면 다음과 같습니다.

```cpp
bool dp[65][10005];
int arr[65];
int main() {
    fastio();
    int N, S;
    cin >> N >> S;
    for(int i=1; i<=N; i++) {
        cin >> arr[i];
    }
    dp[0][0] = true;
    for(int i=1; i<=N; i++) {
        for(int j=0; j<=S; j++) {
            dp[i][j] = dp[i-1][j];
            if(j - arr[i] >= 0) {
                dp[i][j] |= dp[i-1][j-arr[i]];
            }
        }
    }
    if(dp[N][S]) cout << "Yes";
    else cout << "No";
}
```

Dynamic Programming

- 아래 문제를 풀어봅시다.

問題文

宝箱には N 個の品物が入っており、それぞれ 1 から N までの番号が付けられています。
品物 i の重さは w_i であり、価値は v_i です。

太郎君は、いくつかの品物を選んで持ち帰りたいと考えています。
しかし、彼のナップザックには容量制限があるので、重さの合計が W 以下になるようにする必要があります。
価値の合計としてあり得る最大の値はいくつですか。

制約

- $1 \leq N \leq 100$
- $1 \leq W \leq 100000$
- $1 \leq w_i \leq W$
- $1 \leq v_i \leq 10^9$
- 入力はすべて整数

$$
\begin{array}{ll}
N & W \\
w_1 & v_1 \\
w_2 & v_2 \\
\vdots & \\
w_N & v_N
\end{array}
$$

- 보물상자에 N개의 물건이 있고, 각 물건은 1부터 N까지 번호가 붙어 있습니다.
- 물건 i의 무게는 w_i이고 가치는 v_i이며, 물건 몇 개를 선택해서 가져갈 수 있습니다.
- 물건을 담는 가방이 버틸 수 있는 무게는 W일때 (최대 W), 가치의 합의 최댓값을 구하는 프로그램을 작성하세요.

Dynamic Programming

- 이전에 N개의 카드에서 몇 장을 골라서 뽑았을 때 써져있는 정수들의 합이 S가 되는지 판단하는 문제가 있었습니다.

- 이를 조금 변형해서 생각을 해봅시다.

Dynamic Programming

- DP[i][j]를 물건 i까지 넣을 수 있고, 가방 무게가 최대 j일때, 가치의 최대값이라고 합시다.
- 먼저 물품 0개로는 가치의 합이 0이므로 DP[0][0] = 0입니다.
- 또한 가방의 최대 무게가 늘어나도 물품 0개로 만들 수 있는 가치의 합은 0이므로 $i = 0$일땐 모두 0입니다.

j

i DP	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1								
2								
3								
4								

Dynamic Programming

- 이제 1번째 물품($w_1 = 3, v_1 = 13$)을 넣거나 넣지 않는 경우를 계산해보겠습니다.
- 가방에 넣지 않는 경우에는 DP[i][j] = DP[i-1][j]와 같습니다. (넣지 않으면 가치합이 증가하지 않음)

j

i DP	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2								
3								
4								

Dynamic Programming

- 이제 물품 1을 가방에 넣는 경우를 보겠습니다.
- 넣는 경우에는 가방에 최소한 w_1만큼의 여유 공간이 필요합니다.
- 따라서 j $\geq w_1$인 j에 대해 DP[i][j] = max(DP[i][j], DP[i−1][j−w_1] + v_1)가 됩니다.

j

i	DP	0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
	1	0	0	0	13	13	13	13	13
	2								
	3								
	4								

Dynamic Programming

- 이제 2번째 물품($w_2 = 3, v_2 = 17$)을 넣거나 넣지 않는 경우를 계산해보겠습니다.
- 가방에 넣지 않는 경우에는 DP[i][j] = DP[i-1][j]와 같습니다. (넣지 않으면 가치합이 증가하지 않음)

j

i DP	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	13	13	13	13	13
2	0	0	0	13	13	13	13	13
3								
4								

Dynamic Programming

- 이제 물품 2를 가방에 넣는 경우를 보겠습니다.
- 넣는 경우에는 가방에 최소한 w_2만큼의 여유 공간이 필요합니다.
- 따라서 j $\geq w_2$인 j에 대해 DP[i][j] = max(DP[i][j], DP[i−1][j−w_2] + v_2)가 됩니다.

j

i DP	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	13	13	13	13	13
2	0	0	0	17	17	17	30	30
3								
4								

Dynamic Programming

- 이제 3번째 물품($w_3 = 5, v_2 = 29$)을 넣거나 넣지 않는 경우를 계산해보겠습니다.
- 가방에 넣지 않는 경우에는 DP[i][j] = DP[i-1][j]와 같습니다. (넣지 않으면 가치합이 증가하지 않음)

j

i DP	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	13	13	13	13	13
2	0	0	0	17	17	17	30	30
3	0	0	0	17	17	17	30	30
4								

Dynamic Programming

- 이제 물품 3을 가방에 넣는 경우를 보겠습니다.
- 넣는 경우에는 가방에 최소한 w_3만큼의 여유 공간이 필요합니다.
- 따라서 j $\geq w_3$인 j에 대해 DP[i][j] = max(DP[i][j], DP[i−1][j−w_3] + v_3)가 됩니다.

j

i DP	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	13	13	13	13	13
2	0	0	0	17	17	17	30	30
3	0	0	0	17	17	29	30	30
4								

Dynamic Programming

- 이제 4번째 물품($w_4 = 1, v_2 = 10$)을 넣거나 넣지 않는 경우를 계산해보겠습니다.
- 가방에 넣지 않는 경우에는 DP[i][j] = DP[i-1][j]와 같습니다. (넣지 않으면 가치합이 증가하지 않음)

j

i DP	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	13	13	13	13	13
2	0	0	0	17	17	17	30	30
3	0	0	0	17	17	29	30	30
4	0	0	0	17	17	29	30	30

Dynamic Programming

- 이제 물품 4를 가방에 넣는 경우를 보겠습니다.
- 넣는 경우에는 가방에 최소한 w_4만큼의 여유 공간이 필요합니다.
- 따라서 j $\geq w_4$인 j에 대해 DP[i][j] = max(DP[i][j], DP[i−1][j−w_4] + v_4)가 됩니다.

j

i DP	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	13	13	13	13	13
2	0	0	0	17	17	17	30	30
3	0	0	0	17	17	29	30	30
4	0	10	10	17	27	29	39	40

Dynamic Programming

- 모든 DP 테이블을 채웠으므로, 이제 정답을 출력해주면 됩니다.
- 정답은 가능한 i, j에 대해서 DP[i][j]의 최댓값을 출력해주면 됩니다.
- 정답은 40입니다.

j

i	DP	0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
	1	0	0	0	13	13	13	13	13
	2	0	0	0	17	17	17	30	30
	3	0	0	0	17	17	29	30	30
	4	0	10	10	17	27	29	39	40

Dynamic Programming

- 코드로 짜면 다음과 같습니다.

```cpp
pair<ll, ll> arr[105];
ll dp[105][100005];
int main() {
    fastio();
    int N; ll W;
    cin >> N >> W;
    dp[0][0] = 0;
    for(int i=1; i<=N; i++) {
        cin >> arr[i].first >> arr[i].second;
    }
    ll ans = 0;
    for(int i=1; i<=N; i++) {
        for(int j=0; j<=W; j++) {
            dp[i][j] = max(dp[i-1][j], dp[i][j]);
            if(j - arr[i].first >= 0) {
                dp[i][j] = max(dp[i-1][j - arr[i].first] + arr[i].second, dp[i][j]);
            }
            ans = max(dp[i][j], ans);
        }
    }
    cout << ans;
}
```

Dynamic Programming Basic - 1

끝.

다음주에는 Dynamic Programming Basic – 2로 찾아뵙겠습니다.