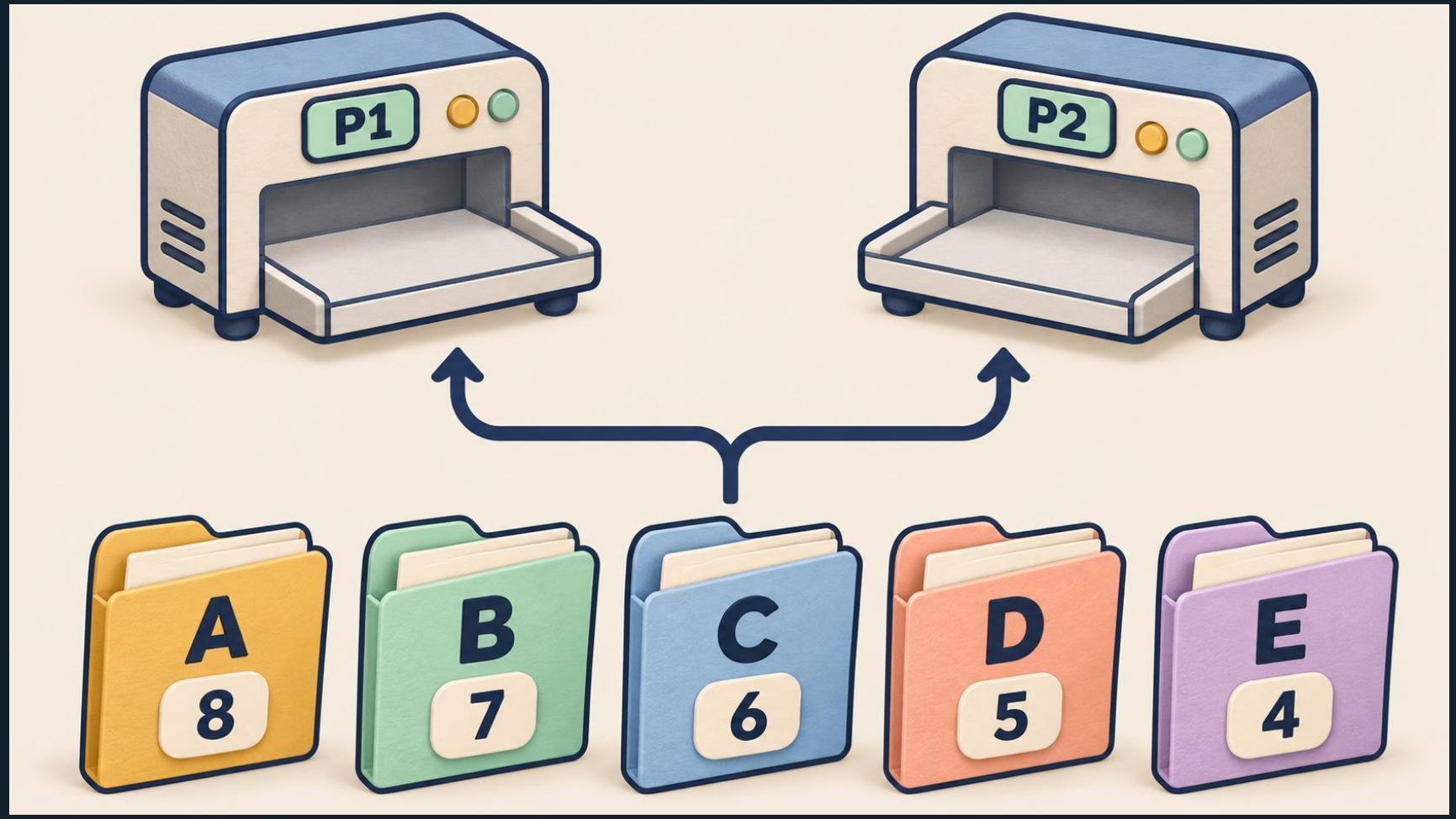
근사 알고리즘



같은 속도의 프린터 두 대입니다.

파일 다섯 개를 한 번씩 출력합니다.

모두 가장 빨리 끝나게 나눠 보세요.

각 파일의 숫자는 출력 시간입니다. A부터 E까지 8, 7, 6, 5, 4분입니다.

9-1 같은 일을 두 기계에 나눠 맡깁니다

목표는 모든 작업을 끝내는 시각을 가능한 한 작게 만드는 것입니다.

주어진 작업 8, 7, 6, 5, 4분짜리 작업 다섯 개입니다.

같은 조건 기계 두 대의 속도가 같고, 시간 0에 함께 시작합니다.

배정 조건 각 작업을 한 기계에 한 번만 맡겨 끝까지 처리합니다.

작업 진행 각 기계는 맡은 작업을 하나씩 연속해서 처리합니다.

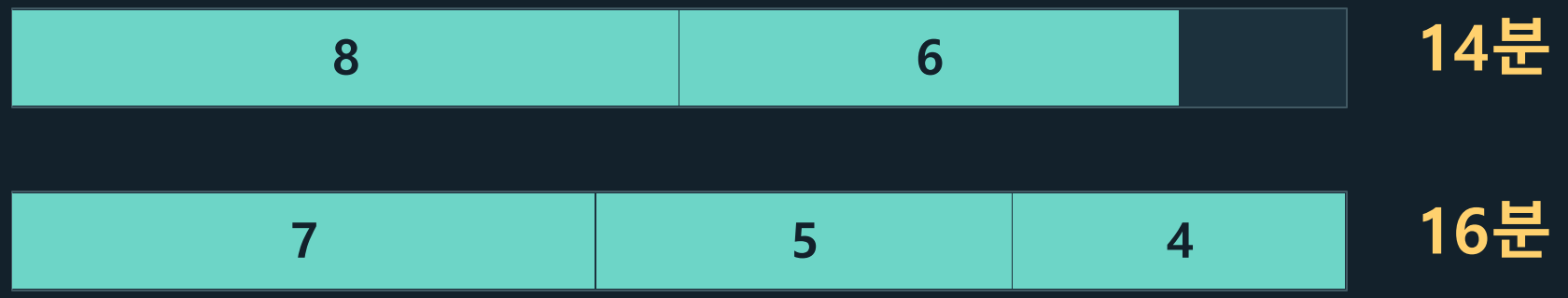
한 기계에 맡긴 시간의 합이 그 기계의 완료 시각입니다.

9-1 14분과 16분이면 모두 끝나는 시각은 16분입니다

왼쪽은 8+6=14분, 오른쪽은 7+5+4=16분입니다.

작업 시간 = [8, 7, 6, 5, 4]

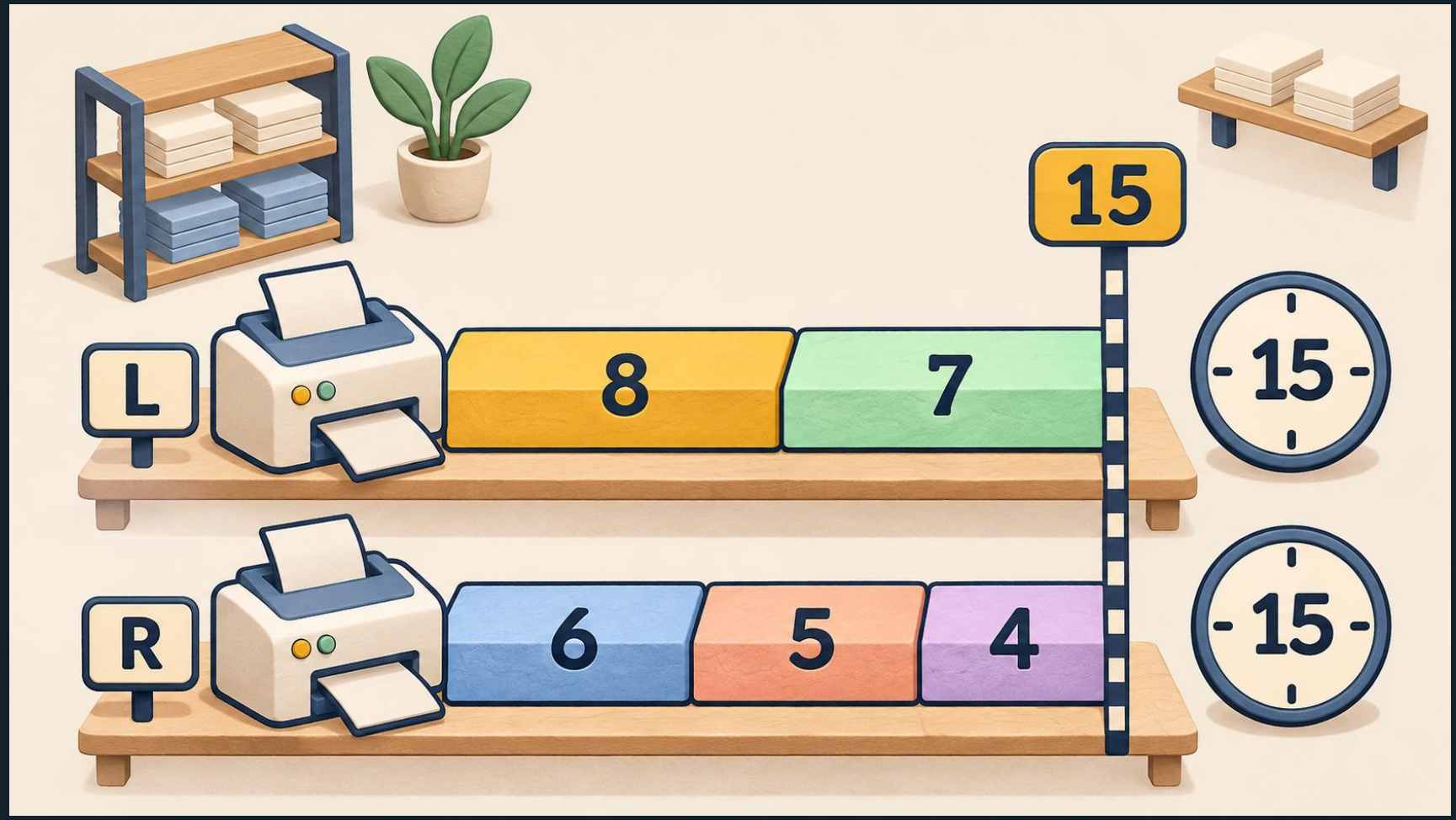
모든 작업의 배정 완료



왼쪽 14분　　오른쪽 16분

왼쪽이 끝난 14분에도 오른쪽 일이 남아 있습니다. 16분에 모두 끝납니다.

9-1 두 프린터가 함께 15분에 끝납니다



위쪽 L은 8+7=15분입니다.

아래 R은 6+5+4=15분입니다.

두 기계가 15분에 함께 끝납니다.

왼쪽 8+7=15분, 오른쪽 6+5+4=15분. 전체 완료 시각은 15분입니다.

9-1 15분보다 빠르게 끝낼 수는 없습니다

필요한 작업량은 30분어치입니다.

왼쪽 기계 — 14분 동안 14분어치

오른쪽 기계 — 14분 동안 14분어치

처리할 수 있는 양 28분어치

필요한 30분어치에서 2분어치가 남습니다.

두 기계가 15분 일하면 30분어치를 처리합니다. 실제 15분 배정도 찾았습니다.

9-1 작업이 많으면 배정할 조합도 많아집니다

각 작업마다 왼쪽과 오른쪽 중 하나를 고릅니다.

작업 수	두 기계를 구분하여 세는 배정 수
5개	2^5 = 32개
20개	2^{20} = 1,048,576개
40개	2^{40} = 약 1조 1천억 개

큰 입력에서도 빨리 쓸 수 있는 배정을 만들고 싶습니다.

9-1 덜 맡은 기계에 다음 작업을 보냅니다

배정 계획을 만들 때 지금까지 맡긴 시간의 합을 비교합니다.

읽는 순서 입력 목록의 앞에서부터 작업을 하나씩 읽습니다.

기계 선택 맡긴 시간의 합이 작은 쪽에 이번 작업을 더합니다.

두 합이 같으면 왼쪽 기계에 맡깁니다.

앞에서 정한 배정은 유지하고, 다음 작업을 어디에 맡길지 정합니다.

9-1 코드에 쓰는 이름을 먼저 정합니다

jobs에는 작업 시간 목록, left와 right에는 누적 시간을 둡니다.

jobs	[8,7,6,5,4]라는 작업 시간 목록입니다.
left / right	왼쪽과 오른쪽 기계에 지금까지 맡긴 시간의 합입니다.
duration	지금 읽은 작업 하나의 시간입니다. 8, 7, 6, 5, 4로 바뀝니다.

처음에는 left=0, right=0입니다. 아직 어느 기계에도 작업을 맡기지 않았습니다.

9-1 함수가 받을 입력과 반환할 숫자

two_machine_time(jobs)

입력	0~1000개 작업이며 각 시간은 0~1,000,000인 정수입니다.
정해진 규칙	작은 합에 추가하고, 같으면 왼쪽을 고릅니다.
반환값	모두 배정한 뒤 left와 right 중 큰 값을 반환합니다.

이 함수에는 작업 목록을 줍니다. 최적값을 미리 구해서 넣을 필요가 없습니다.

9-1 비교하고 더하는 기호를 읽습니다

코드에서 사용할 세 표현입니다.

표현	하는 일
left <= right	왼쪽 합이 오른쪽 합보다 작거나 같은지 확인
left += duration	원래 left에 이번 시간을 더해 다시 저장
max(left, right)	두 값 중 큰 값을 선택

left += 5에서 left가 8이었다면 새 값은 13입니다.

9-1 Python 전체 답안

```python
def two_machine_time(jobs):
    left = 0
    right = 0
    for duration in jobs:
        if left <= right:
            left += duration
        else:
            right += duration
    return max(left, right)
```

두 합을 0으로 준비

작업 시간을 하나씩 읽음

작거나 같은 쪽에 더함

마지막에 큰 합 반환

다음 장부터 같은 입력을 넣어 각 줄의 동작을 확인합니다.

9-1 두 합은 시작할 때 한 번 0으로 만듭니다

```
01  def two_machine_time(jobs):
02      left = 0
03      right = 0
04      for duration in jobs:
```

변수	처음 값
jobs	[8,7,6,5,4]
left	0
right	0
첫 duration	8

2~3행은 반복문 밖입니다.

앞서 맡긴 합을 다음 반복에도 사용합니다.

for duration in jobs는 8, 7, 6, 5, 4를 차례로 읽습니다.

9-1 첫 작업 8분을 배정합니다

left=0, right=0. 0 <= 0은 참입니다.

`jobs = [8, 7, 6, 5, 4]`

이번 작업: 8분

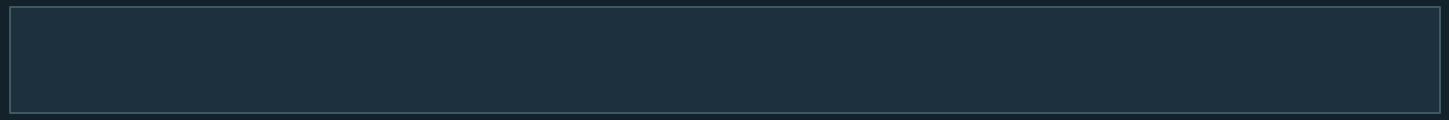
배정 전: 왼쪽 0, 오른쪽 0

왼쪽 · left



8분

오른쪽 · right



0분

`left = 8` **`right = 0`**

`left += duration` `0 + 8 = 8`

이번 작업 뒤 left=8, right=0입니다.

9-1 둘째 작업 7분을 배정합니다

left=8, right=0. 8 <= 0은 거짓입니다.

`jobs = [8, 7, 6, 5, 4]`

이번 작업: 7분　　배정 전: 왼쪽 8, 오른쪽 0

왼쪽 · left



8분

오른쪽 · right



7분

`left = 8`　　`right = 7`

`right += duration`　　`0 + 7 = 7`

이번 작업 뒤 left=8, right=7입니다.

9-1 셋째 작업 6분을 배정합니다

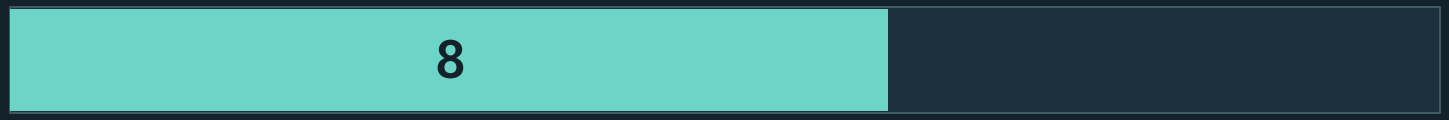
left=8, right=7. 8 <= 7은 거짓입니다.

```
jobs = [8, 7, 6, 5, 4]
```

이번 작업: 6분

배정 전: 왼쪽 8, 오른쪽 7

왼쪽 · left



8분

오른쪽 · right



13분

left = 8 **right = 13**

`right += duration` `7 + 6 = 13`

이번 작업 뒤 left=8, right=13입니다.

9-1 넷째 작업 5분을 배정합니다

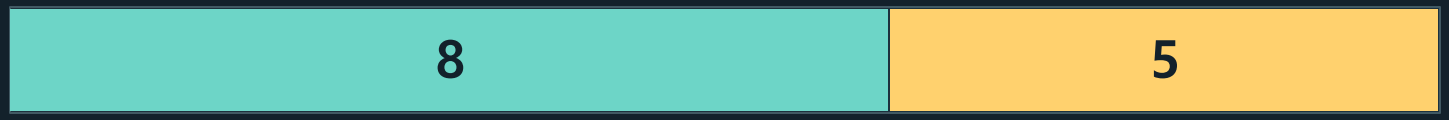
left=8, right=13. 8 <= 13은 참입니다.

`jobs = [8, 7, 6, 5, 4]`

이번 작업: 5분

배정 전: 왼쪽 8, 오른쪽 13

왼쪽 · left



13분

오른쪽 · right



13분

`left = 13` **`right = 13`**

`left += duration` `8 + 5 = 13`

이번 작업 뒤 left=13, right=13입니다.

9-1 다섯째 작업 4분을 배정합니다

left=13, right=13. 13 <= 13은 참입니다.

`jobs = [8, 7, 6, 5, 4]`

이번 작업: 4분

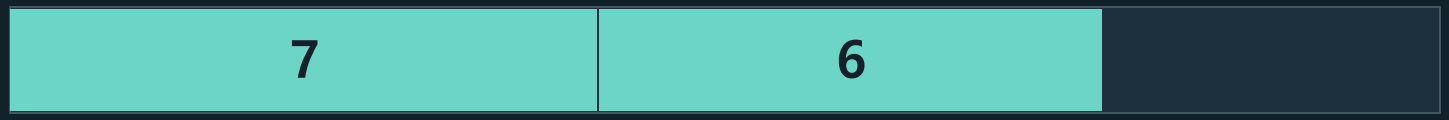
배정 전: 왼쪽 13, 오른쪽 13

왼쪽 · left



17분

오른쪽 · right



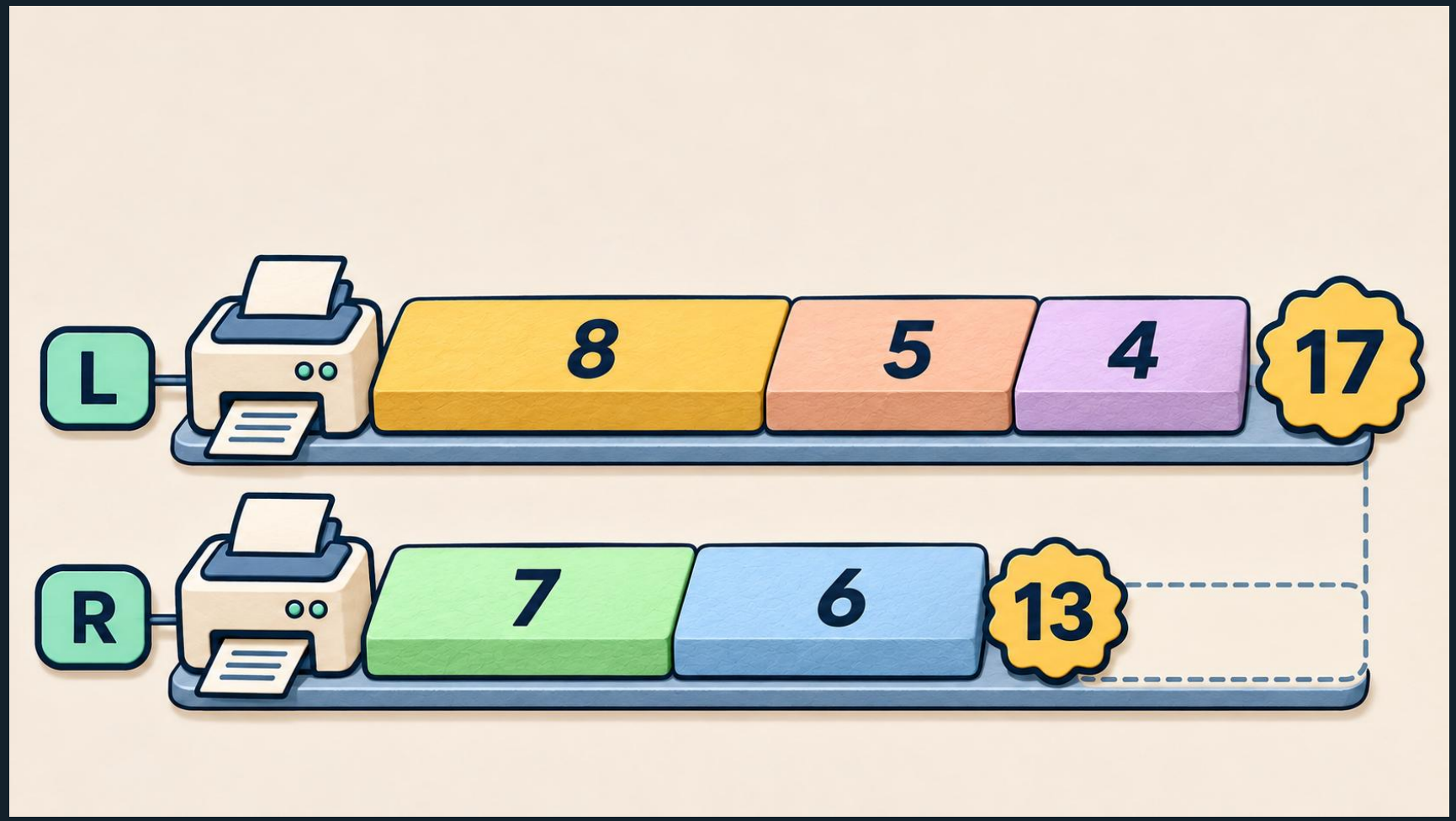
13분

`left = 17` **`right = 13`**

`left += duration` `13 + 4 = 17`

이번 작업 뒤 left=17, right=13입니다.

9-1 빨리 만든 배정은 17분에 끝납니다



L에는 8, 5, 4를 맡겼습니다.

R에는 7, 6을 맡겼습니다.

R이 끝나도 L을 기다려야 합니다.

left=17, right=13이므로 max(17,13)=17입니다.

9-1 return은 모든 작업을 읽은 뒤 실행합니다

```
09          return max(left, right)
```

반복 종료 뒤	값
left	17
right	13
max(left, right)	17

9행은 for와 같은 들여쓰기입니다.

max가 고른 17을 함수 결과로 보냅니다.

return을 반복문 안에 넣으면 첫 작업 뒤 함수가 끝나므로 위치를 확인합니다.

9-1 C로도 같은 배정을 만듭니다

```c
int two_machine_time(const int jobs[], int n) {
    int left = 0;
    int right = 0;
    for (int i = 0; i < n; ++i) {
        int duration = jobs[i];
        if (left <= right) {
            left += duration;
        } else {
            right += duration;
        }
    }
    if (left >= right) return left;
    return right;
}
```

jobs는 시간 배열

n은 배열의 작업 개수

i는 배열의 위치 번호

큰 합을 정수로 반환

Python에서 본 17분 결과를 C로도 계산합니다.

9-1 C에서 int와 입력 배열을 읽습니다

```
01  int two_machine_time(const int jobs[], int n) {
02      int left = 0;
03      int right = 0;
```

코드	뜻
int two_machine_time(...)	정수 결과를 반환하는 함수
const int jobs[]	작업 시간을 담은 입력 배열
int n	입력 배열의 작업 개수
int left=0, int right=0	정수로 된 두 누적 합을 준비

const는 입력 배열을 보존합니다.

int는 정수를 저장합니다.

각 선언 끝의 세미콜론은 그 문장의 끝입니다.

9-1 C의 위치 번호 i와 작업 시간을 구분합니다

```
04        for (int i = 0; i < n; ++i) {
05            int duration = jobs[i];
```

표현	이 예제의 동작
int i = 0	첫 위치인 0부터 시작
i < n	n=5이므로 위치 0~4를 읽음
++i	한 작업 뒤 i를 1 증가
jobs[i]	8, 7, 6, 5, 4라는 시간

i=0일 때 jobs[0]=8입니다.

i=1일 때 jobs[1]=7입니다.

i는 작업의 위치이고, duration은 해당 작업에 걸리는 시간입니다.

9-1 C의 if와 else도 한쪽에만 더합니다

```
06          if (left <= right) {
07              left += duration;
08          } else {
09              right += duration;
10          }
```

배정 직전	실행
0 <= 0은 참	left에 8을 더함
8 <= 0은 거짓	right에 7을 더함
13 <= 13은 참	left에 4를 더함

참이면 if의 중괄호 안을 실행합니다.

거짓이면 else의 중괄호 안을 실행합니다.

한 작업을 한 기계에만 맡기므로 두 덧셈 중 하나를 실행합니다.

9-1 C는 반복을 마친 뒤 큰 합을 반환합니다

```
11        }
12        if (left >= right) return left;
13        return right;
14    }
```

줄	하는 일
11행의 }	작업을 하나씩 읽는 반복 범위를 끝냄
12행	left >= right이면 left를 반환
13행	그 외에는 right를 반환

17 >= 13은 참입니다.

12행에서 17을 반환하고 끝납니다.

최종 합이 3과 6이면 12행을 지나 13행에서 6을 반환합니다.

9-1 이제 결과 17분의 품질을 따져 봅니다

빠른 규칙이 만든 배정은 유효하지만, 최적 배정보다 늦을 수 있습니다.

직접 찾은 최적값	[8,7]과 [6,5,4]로 나누면 15분입니다.
규칙의 결과	[8,5,4]와 [7,6]으로 나누어 17분입니다.
증명할 것	어떤 입력에서도 이 규칙의 결과는 최적 시간의 2배 이내입니다.

2배는 결과가 넘지 않는 한도입니다. 이번 비율은 17/15입니다.

9-1 어떤 배정에도 필요한 시간을 하한이라 부릅니다

최적값을 직접 몰라도, 그보다 작거나 같은 기준은 구할 수 있습니다.

전체 작업량 30분어치를 두 기계가 처리하므로 최소 15분은 필요합니다.

작업 하나의 길이 4분짜리 E를 한 기계가 끝까지 처리하므로 최소 4분은 필요합니다.

하한의 의미 가장 좋은 배정이라도 이 시간보다 짧게 끝낼 수 없습니다.

하한은 최적 시간의 아래쪽 기준입니다. 최적과 같을 수도, 더 작을 수도 있습니다.

9-1 가장 늦게 끝나는 작업은 E입니다

전체 완료 시각 17분을 정한 작업은 왼쪽의 마지막 E입니다.



17분 = E가 시작되기 전 13분 + E를 처리하는 4분

9-1 E를 맡기기 직전에는 두 합이 13입니다

아직 E를 추가하기 전입니다. 앞서 맡긴 작업의 합은 13과 13입니다.

`jobs = [8, 7, 6, 5, 4]`

이번 작업: 4분　　배정 전: 왼쪽 13, 오른쪽 13



left = 13　　**right = 13**

`min(13, 13) = 13`　　`(13 + 13) / 2 = 13`

규칙이 고른 작은 합은 두 합의 평균보다 클 수 없습니다.

9-1 앞부분 13분은 최적 시간 이내입니다

E를 맡기기 전의 작업량은 26분어치, 모든 작업량은 30분어치입니다.

E를 맡기기 전

26 ÷ 2 = 13분

기계 수는 두 대입니다.

E까지 포함한 전체

30 ÷ 2 = 15분

기계 수는 두 대입니다.

작은 합 ≤ 당시 평균 ≤ 전체 평균 ≤ 최적 시간

13분 ≤ 26/2 = 13분 ≤ 30/2 = 15분 ≤ 최적 시간

9-1 E 자체의 4분도 최적 시간 이내입니다

최적 배정도 E를 어느 한 기계에서 끝까지 처리해야 합니다.

왼쪽에 맡겨도 **E를 처리하는 4분**

오른쪽에 맡겨도 **E를 처리하는 4분**

어느 배정도 이 작업을 완료해야 합니다.

E의 길이 4분 ≤ 최적 시간

9-1 두 부분을 더하면 최적의 2배 이내입니다

E 앞의 시간과 E 자체의 시간을 각각 최적 시간과 비교했습니다.

시작 전 시간 13분 — 최적 시간 이하

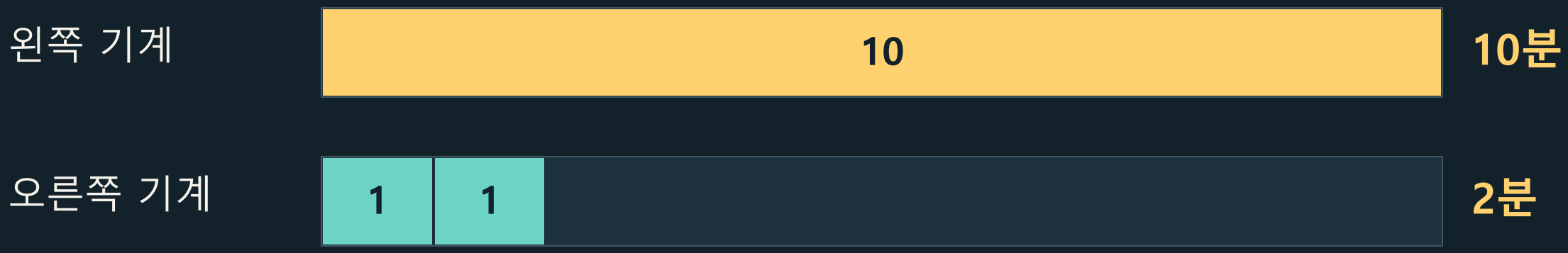
그 작업의 길이 4분 — 최적 시간 이하

$$13 + 4 = 17\text{분} \le 2 \times \text{최적 시간}$$

전체 완료 시각 17분 ≤ 최적 시간 + 최적 시간 = 최적 시간의 2배

9-1 입력의 맨 끝 작업이 가장 늦게 끝나지는 않을 수 있습니다

[10,1,1]은 왼쪽 10분, 오른쪽 2분입니다. 가장 늦게 끝나는 것은 10분 작업입니다.



시작 전 시간은 0분입니다.



증명에서는 완료 시각이 가장 늦은 작업 하나를 골라 앞부분과 자기 길이를 나눕니다.

9-1 어떤 입력에서도 같은 비교가 성립합니다

완료 시각이 가장 늦은 작업을 골라 시작 전 시간과 자기 길이로 나눕니다.

시작 전 시간 규칙이 고른 작은 합 ≤ 당시 평균 ≤ 전체 작업량/2 ≤ 최적 시간

그 작업의 길이 최적 배정에서도 그 작업을 끝내야 하므로 최적 시간 이하입니다.

두 부분의 합 우리 완료 시각은 두 부분의 합이므로 최적 시간의 2배 이하입니다.

이미 읽은 작업량은 전체 작업량 이하입니다. 나중 작업이 있어도 같은 관계가 성립합니다.

9-1 다른 입력의 반환값도 직접 확인합니다

입력 순서를 따라 두 합을 계산하고 큰 값을 반환합니다.

jobs	최종 left, right	반환값
[3,2,4]	3, 6	6
[4,3,2]	4, 5	5
[0,0]	0, 0	0
[]	0, 0	0

작업 n개를 한 번 읽으므로 시간 O(n), 누적 합만 저장하므로 추가 공간 O(1)입니다.

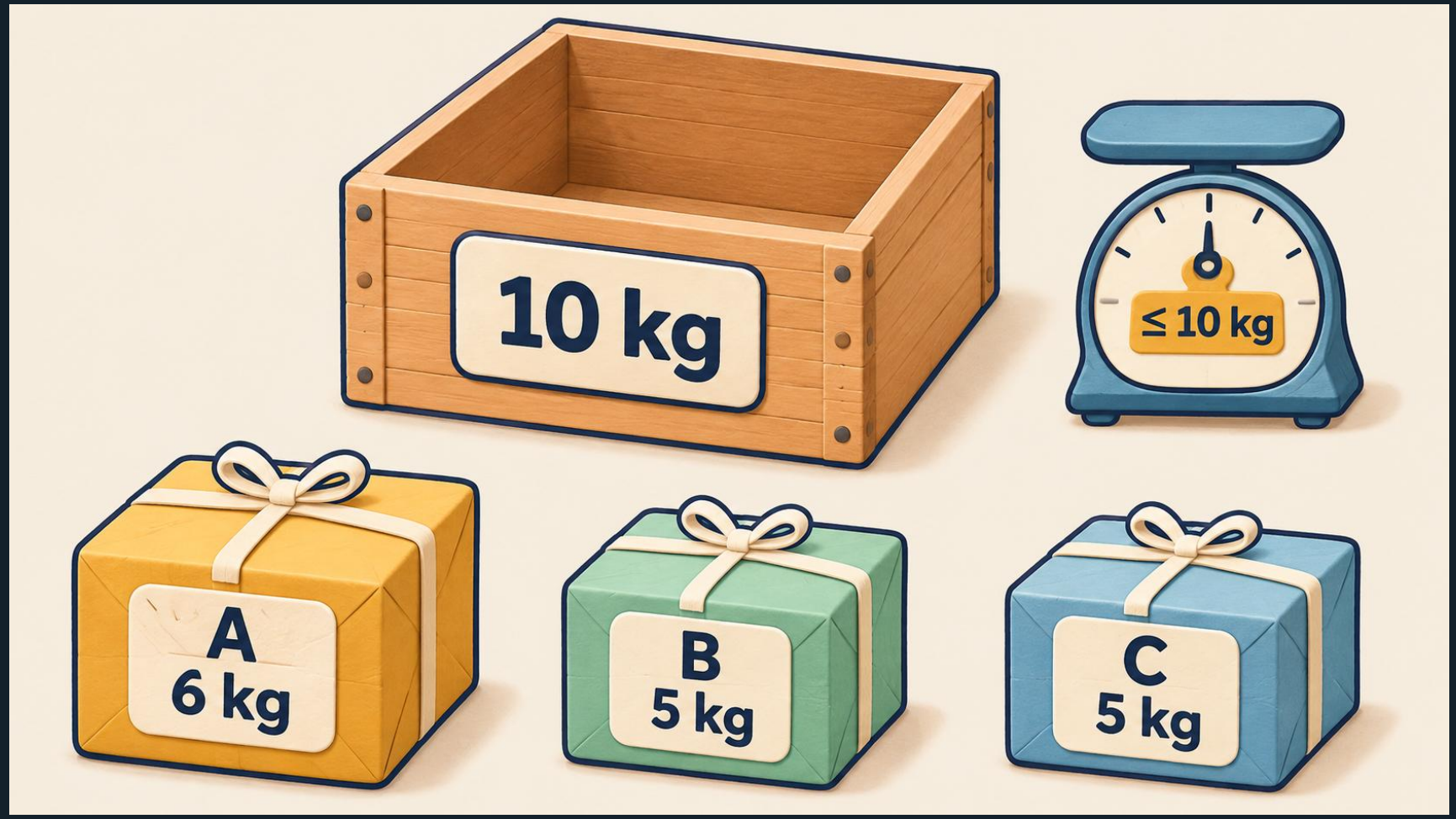
9-2 이번에는 상자에 담은 무게를 크게 만듭니다

상자 하나에 물건을 골라 담습니다. 제한 이하에서 무게 합을 최대화합니다.

주어진 것	물건별 무게 목록과, 상자가 버틸 수 있는 최대 무게입니다.
고르는 것	각 물건을 담을지 정합니다. 한 물건은 한 번만 담습니다.
지킬 조건	물건을 쪼개지 않고, 담은 무게의 합이 제한을 넘지 않아야 합니다.
구할 것	조건을 지키면서 가장 무겁게 담는 선택입니다.

물건의 가격이나 크기는 주어지지 않습니다. 무게만 비교합니다.

9-2 세 물건 중 무엇을 담을까요?



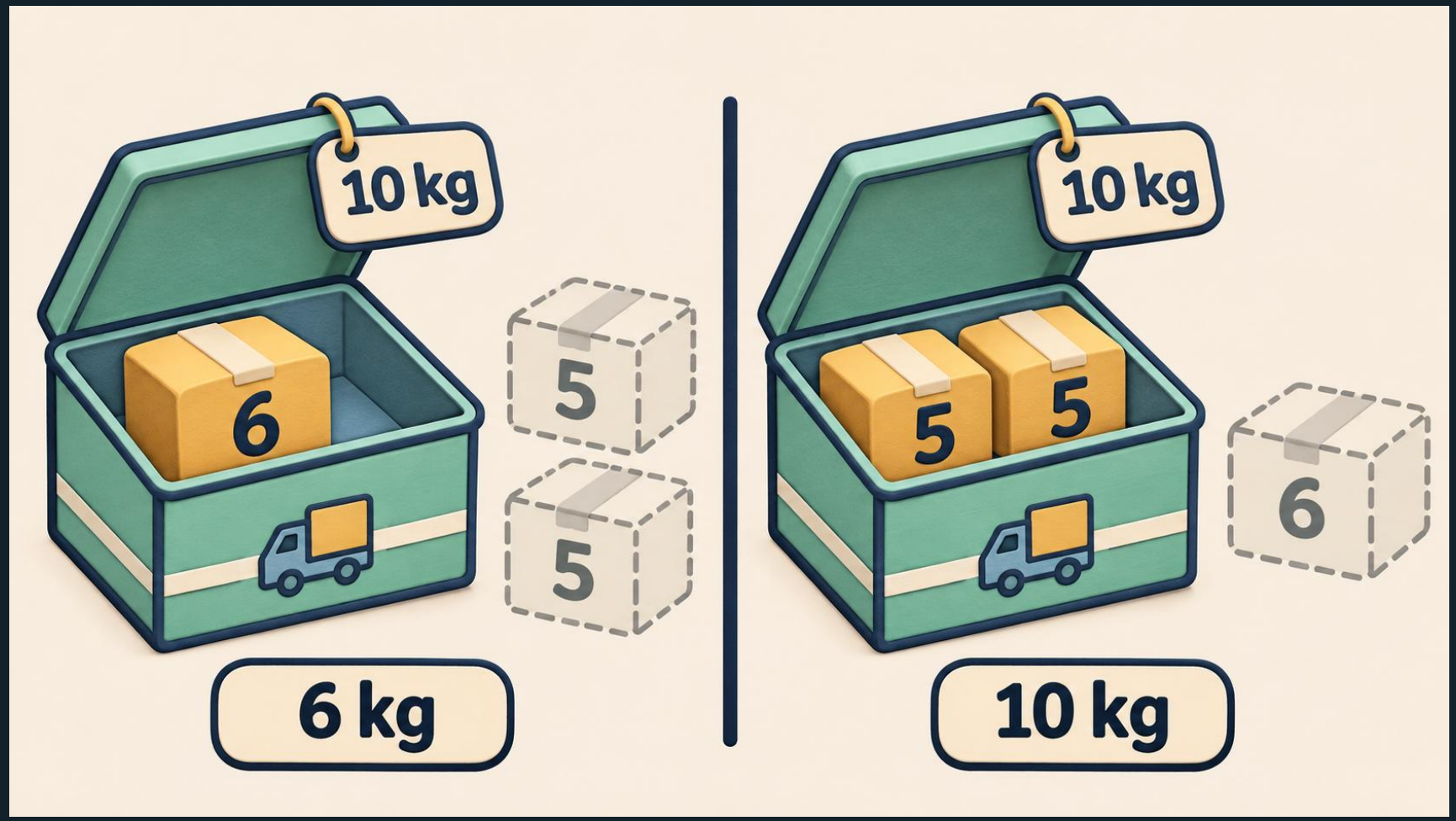
상자의 무게 제한은 10kg입니다.

A는 6kg, B와 C는 각각 5kg입니다.

제한 안에서 가장 무겁게 담아 보세요.

A는 6kg, B와 C는 각각 5kg입니다. 합이 10kg 이하가 되도록 고릅니다.

9-2 가장 무거운 물건을 빼면 더 많이 담깁니다



A 하나는 6kg입니다.

B와 C를 함께 담으면
10kg입니다.

제한을 꽉 채워
최적값은 10kg입니다.

같은 물건들로 만든 두 선택을 비교합니다. 상자의 제한은 모두 10kg입니다.

9-2 물건이 많아지면 담을 조합도 많아집니다

물건마다 담거나 빼는 두 선택이 있습니다.

물건 수	모든 부분집합의 수
3개	2^3 = 8개
20개	2^{20} = 1,048,576개
40개	2^{40} = 약 1조 1천억 개

목록을 한 번 읽으며 실제로 사용할 수 있는 후보를 만들겠습니다.

9-2 첫 후보는 들어가는 물건을 순서대로 담습니다

담은 합은 0에서 시작합니다. 다음 물건을 더해도 제한 이하면 담습니다.

들어가면	현재 합에 이번 물건의 무게를 더합니다.
들어가지 않으면	현재 합을 유지하고 다음 물건을 확인합니다.
앞서 담은 물건	계속 담아 둡니다. 뒤에서 꺼내거나 순서를 바꾸지 않습니다.

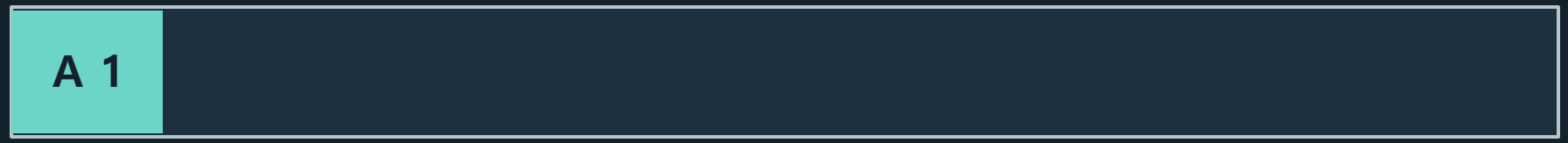
[6,5,5], 제한 10이면 6만 담아 첫 후보는 6kg입니다.

9-2 첫 후보 하나만 쓰면 결과가 작을 수 있습니다

다른 입력 [1,10]을 제한 10kg 상자에 넣어 봅니다.

A	B
1kg	10kg

무게 제한 10kg

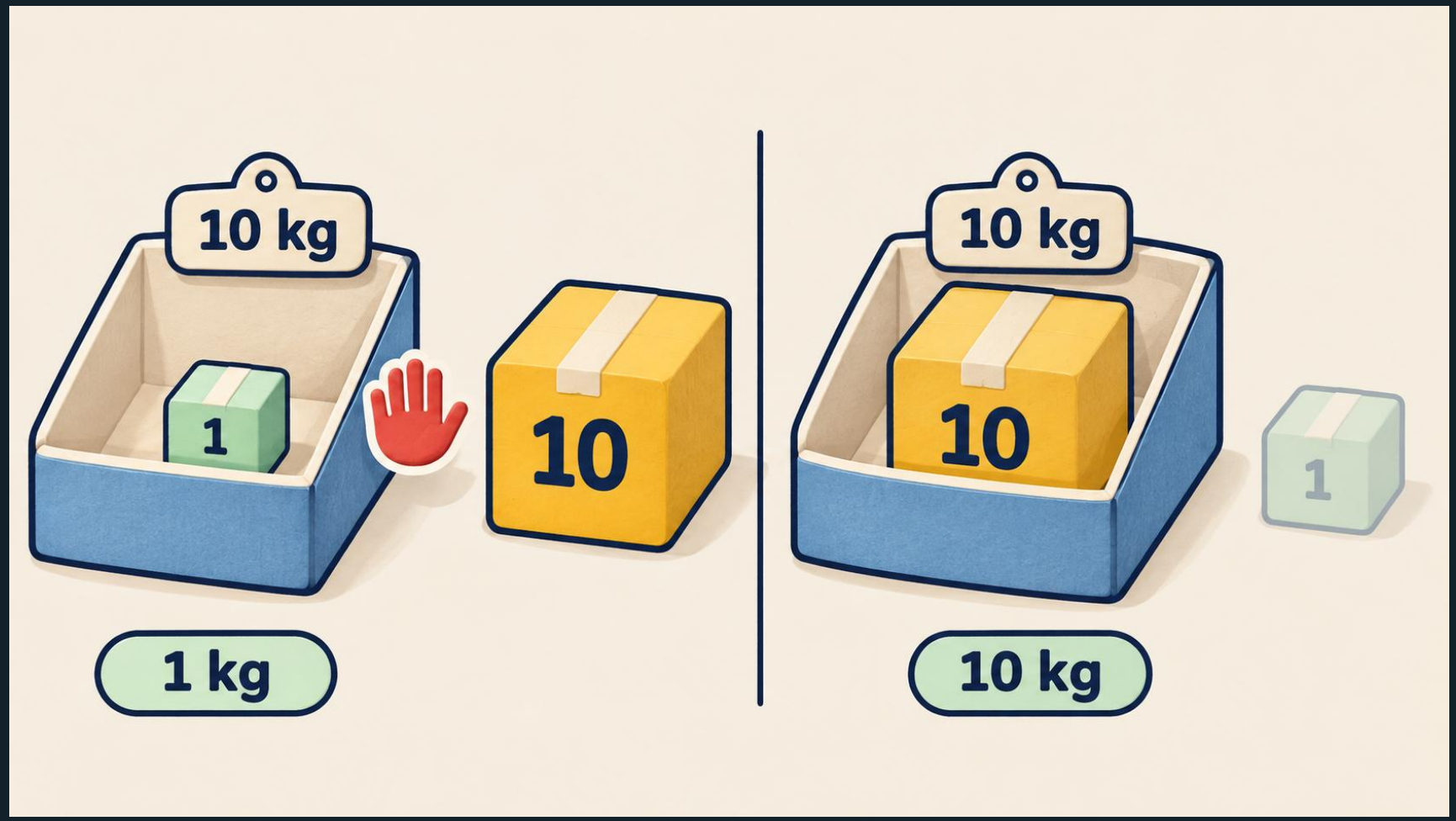


담은 무게 1kg / 빈자리 9kg

1을 먼저 담으면 1+10=11이라 10을 추가하지 못합니다.

첫 후보는 1kg입니다. 10kg 물건 하나를 담는 선택도 가능합니다.

9-2 한 물건만 담는 계획도 함께 비교합니다



첫 후보는 1kg만 담았습니다.

두 번째 후보는 10kg 하나입니다.

더 큰 무게 10kg를 선택합니다.

한 상자에 사용할 두 대안입니다. 마지막에는 무게가 큰 대안을 고릅니다.

9-2 total과 single은 서로 다른 후보입니다

total은 여러 물건을 순서대로 담은 합, single은 물건 하나의 무게입니다.

total 들어가는 대로 담아 만든 한 계획의 총무게입니다.

single 혼자 상자에 들어가는 물건 중 가장 무거운 값입니다.

마지막 선택 두 후보 중 더 큰 무게 하나를 반환합니다.

[1,10], 제한 10이면 total=1, single=10이고 반환값은 10입니다.

9-2 함수의 입력과 반환값을 정합니다

approx_weight(weights, capacity)

weights	0~1000개 물건의 목록이며, 각 무게는 1~1,000,000입니다.
capacity	상자의 무게 제한입니다. 1~1,000,000의 정수입니다.
반환값	정해진 두 후보를 만들고, 더 큰 무게 합을 반환합니다.

상자보다 무거운 물건은 혼자도 들어가지 않으므로 두 후보에서 제외합니다.

9-2 한 물건을 읽을 때 확인하는 순서

순서대로 검사해야 두 후보가 모두 올바르게 계산됩니다.

단계	확인과 처리
1	이번 물건이 제한보다 크면 다음 물건으로 이동
2	혼자 들어가면 single보다 무거운지 확인해 갱신
3	total에 더해도 제한 이하면 total에 추가

total에 추가하지 못해도 혼자 들어가는 물건이면 single의 후보가 됩니다.

9-2 Python 전체 답안

```python
def approx_weight(weights, capacity):
    total = 0
    single = 0
    for weight in weights:
        if weight > capacity:
            continue
        if weight > single:
            single = weight
        if total + weight <= capacity:
            total += weight
    return max(total, single)
```

total과 single을 0으로 준비

무거운 물건은 제외

두 후보를 각자 갱신

큰 후보의 무게를 반환

아래에서 무게 하나를 읽는 각 줄을 실제 숫자와 연결합니다.

9-2 두 후보는 처음에 한 번 0으로 만듭니다

```
01  def approx_weight(weights, capacity):
02      total = 0
03      single = 0
04      for weight in weights:
```

변수	[6,5,5] 예제에서의 의미
weights	전체 물건 목록 [6,5,5]
capacity	무게 제한 10
total / single	둘 다 0으로 시작
weight	지금 읽는 6, 다음 5, 마지막 5

weight는 이번 물건 하나입니다.

weights는 전체 목록입니다.

같은 무게 5가 두 번 나오면 서로 다른 두 물건을 차례로 읽습니다.

9-2 혼자도 못 들어가는 물건은 continue로 건너뜁니다

```
05            if weight > capacity:
06                continue
```

이번 무게 / 제한	조건과 결과
12 / 10	12 > 10은 참, 다음 물건으로 이동
10 / 10	10 > 10은 거짓, 아래 검사로 진행
6 / 10	6 > 10은 거짓, 아래 검사로 진행

continue는 이번 반복을 끝냅니다.

함수 전체는 다음 물건을 계속 처리합니다.

혼자 들어가는 10kg는 제한 10kg와 같으므로 사용할 수 있습니다.

9-2 single에는 더 큰 단일 무게를 남깁니다

```
07            if weight > single:
08                single = weight
```

기존 single / 이번 weight	새 single
0 / 6	6
6 / 5	6 유지
1 / 10	10

비교하는 대상은 혼자 들어가는 물건입니다.

더 클 때만 값을 교체합니다.

single = weight는 이전 후보의 무게를 이번의 더 큰 무게로 바꾸는 대입입니다.

9-2 total에는 합이 제한 이하일 때만 더합니다

```
09            if total + weight <= capacity:
10                total += weight
```

현재 total / 이번 weight	판단
0 / 6	0+6 <= 10이므로 total=6
6 / 5	6+5 > 10이므로 6 유지
6 / 4	6+4 <= 10이므로 total=10

<=는 제한과 같은 경우도 허용합니다.

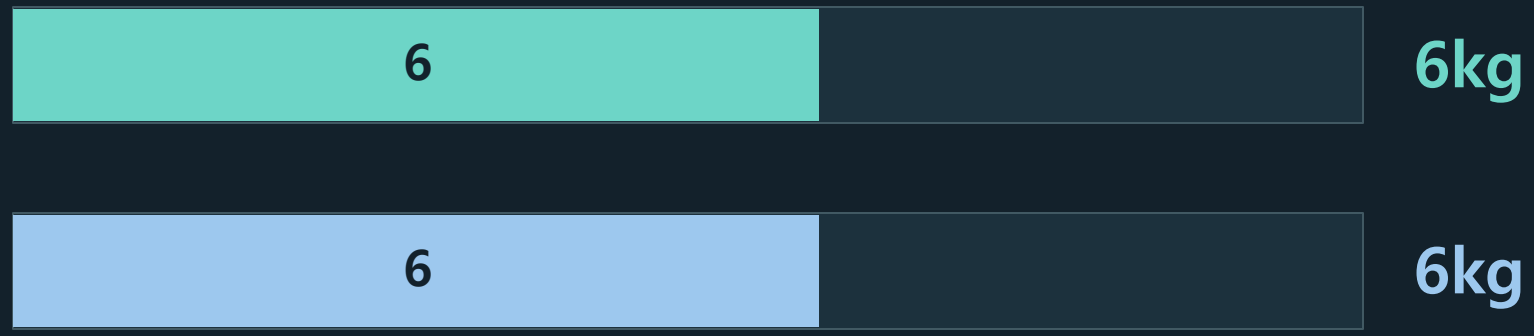
+=는 지금 합에 이번 무게를 더합니다.

담지 못한 물건이 있어도 다음 물건의 무게로 다시 검사합니다.

9-2 첫 6kg를 읽으면 두 후보 모두 6입니다

```
weights=[6,5,5], capacity=10
```

1번째 6kg	2번째 5kg	3번째 5kg



6 > 10은 거짓, 혼자 들어갑니다.

6 > 0이므로 single=6입니다.

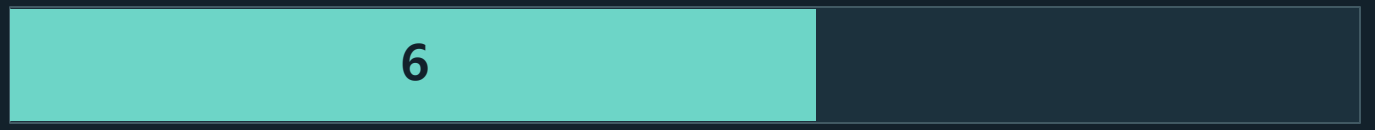
0+6 <= 10이므로 total=6입니다.

9-2 다음 5kg는 total에 추가하지 못합니다

```
weights=[6,5,5], capacity=10
```

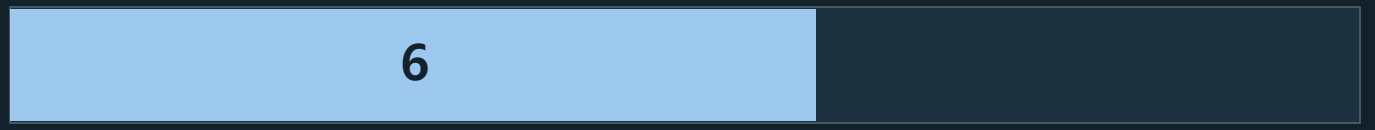
1번째 6kg	**2번째 5kg**	3번째 5kg

첫 후보 total



6kg

단일 후보 single



6kg

5 > 10은 거짓, 혼자 들어갑니다.

5 > 6은 거짓, single=6 유지입니다.

6+5=11이므로 total=6 유지입니다.

9-2 마지막 5kg도 못 담아 반환값은 6입니다

```
weights=[6,5,5], capacity=10
```

1번째 6kg	2번째 5kg	**3번째 5kg**

첫 후보 total — 6 — **6kg**

단일 후보 single — 6 — **6kg**

마지막 5도 single보다 작습니다.

6+5=11이라 total에 못 더합니다.

max(6,6)=6을 반환합니다.

9-2 [1,10]에서는 첫 후보와 단일 후보가 달라집니다

`weights=[1,10], capacity=10`

1번째 1kg | **2번째 10kg**

첫 후보 total — 1 — **1kg**

단일 후보 single — 10 — **10kg**

10은 혼자 들어가므로 검사합니다.
single은 1에서 10으로 바뀝니다.
1+10=11이라 total은 1입니다.

9-2 모든 물건을 읽고 큰 후보 하나를 반환합니다

```
11        return max(total, single)
```

입력과 제한	total	single	반환값
[6,5,5], 10	6	6	6
[1,10], 10	1	10	10
[6,5,4], 10	10	6	10

max는 두 값 중 큰 값을 고릅니다.

return이 그 무게를 돌려줍니다.

11행은 for와 같은 들여쓰기입니다. 모든 물건을 확인한 다음 실행합니다.

9-2 C 전체 답안

```c
int approx_weight(const int weights[], int n, int capacity) {
    int total = 0;
    int single = 0;
    for (int i = 0; i < n; ++i) {
        int weight = weights[i];
        if (weight > capacity) continue;
        if (weight > single) single = weight;
        if (total + weight <= capacity) {
            total += weight;
        }
    }
    if (total >= single) return total;
    return single;
}
```

weights는 무게 배열

n은 물건 개수

capacity는 무게 제한

마지막 두 줄이 max 역할

Python과 같은 두 후보를 C의 정수 변수로 계산합니다.

9-2 C의 입력과 초기값을 읽습니다

```
01  int approx_weight(const int weights[], int n, int capacity) {
02      int total = 0;
03      int single = 0;
```

이름과 표현	뜻
const int weights[]	정수 무게를 담은 입력 배열
int n	물건의 수, 즉 배열 길이
int capacity	상자의 무게 제한
int total=0, single=0	두 후보를 정수 0에서 시작

함수 앞 int는 정수 반환값입니다.

const는 입력 배열을 보존합니다.

각 후보는 제한 이하이고, 비교하는 합도 최대 2,000,000이어서 int로 계산합니다.

9-2 C의 i는 위치, weight는 물건 무게입니다

```
04        for (int i = 0; i < n; ++i) {
05            int weight = weights[i];
```

표현	[6,5,5]에서의 값
n	물건이 3개이므로 3
i	0, 1, 2로 변화
weights[i]	6, 5, 5를 차례로 읽음
++i	이번 물건을 마친 뒤 다음 위치로

i < n인 동안 반복합니다.

첫 위치 번호는 0입니다.

현재 무게 weight를 읽은 다음 세 가지 조건을 순서대로 검사합니다.

9-2 C는 혼자 못 들어가는 물건만 먼저 넘깁니다

```
06          if (weight > capacity) continue;
07          if (weight > single) single = weight;
```

조건	실행할 문장
weight > capacity	continue로 다음 반복
weight > single	single = weight로 더 큰 후보 저장
조건이 거짓일 때	바로 뒤 문장을 실행하지 않고 다음 줄로

두 if는 각각 독립된 검사입니다.

single의 갱신은 total에 넣기 전에 합니다.

한 줄 if에서는 조건 바로 뒤의 문장 하나가 조건부로 실행됩니다.

9-2 C도 합을 확인하고 total을 늘립니다

```
08          if (total + weight <= capacity) {
09              total += weight;
10          }
```

현재 total / weight / 제한	결과
6 / 5 / 10	합 11이 넘으므로 6 유지
6 / 4 / 10	합 10이므로 total=10
1 / 10 / 10	합 11이 넘으므로 1 유지

중괄호 안은 조건이 참일 때 실행합니다.

+=는 기존 합에 더해 저장합니다.

현재 물건을 담지 못해도 for는 다음 위치를 계속 읽습니다.

9-2 C에서 큰 후보를 반환하는 두 줄

```
11        }
12        if (total >= single) return total;
13        return single;
14    }
```

total / single	반환
6 / 6	>=가 참이므로 total 6
1 / 10	>=가 거짓이므로 single 10
10 / 6	>=가 참이므로 total 10

11행은 for의 범위 끝입니다.

return하면 함수 실행이 끝납니다.

같은 값이어도 total >= single이 참이라 같은 숫자를 반환합니다.

9-2 두 후보는 각각 실제로 담을 수 있습니다

반환값이 제한을 지키는지 먼저 확인합니다.

total 후보 합이 제한 이하일 때만 추가했으므로 항상 제한을 지킵니다.

single 후보 혼자 들어가는 물건만 비교했으므로 제한을 지킵니다.

최종 선택 두 후보 중 하나를 고르므로 실제로 사용할 수 있는 무게입니다.

이제 이 유효한 결과가 최적 무게의 절반 이상인지 확인하겠습니다.

9-2 최대화 문제의 보장은 최적 무게의 절반 이상입니다

많이 담을수록 좋으므로, 결과가 얼마나 작아질 수 있는지 확인합니다.

상자의 제한 어떤 선택도 capacity를 넘을 수 없습니다.

최적 무게 제한 이하에서 담을 수 있는 가장 큰 합입니다.

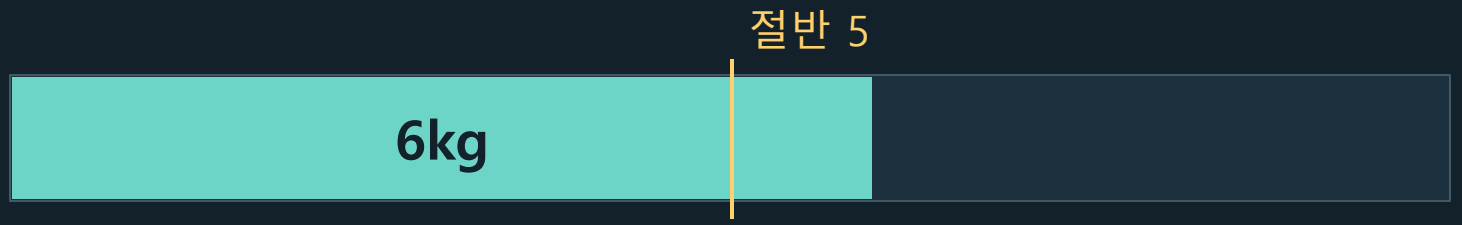
증명할 관계 우리 결과 ≥ 최적 무게/2

9-1은 시간을 작게, 9-2는 무게를 크게 만드는 문제라 비교 방향이 달라집니다.

9-2 total이 상자 절반 이상이면 이미 충분합니다

예제의 total=6은 제한 10의 절반 5 이상입니다.

첫 후보 total



최적 무게 ≤ 제한 10kg

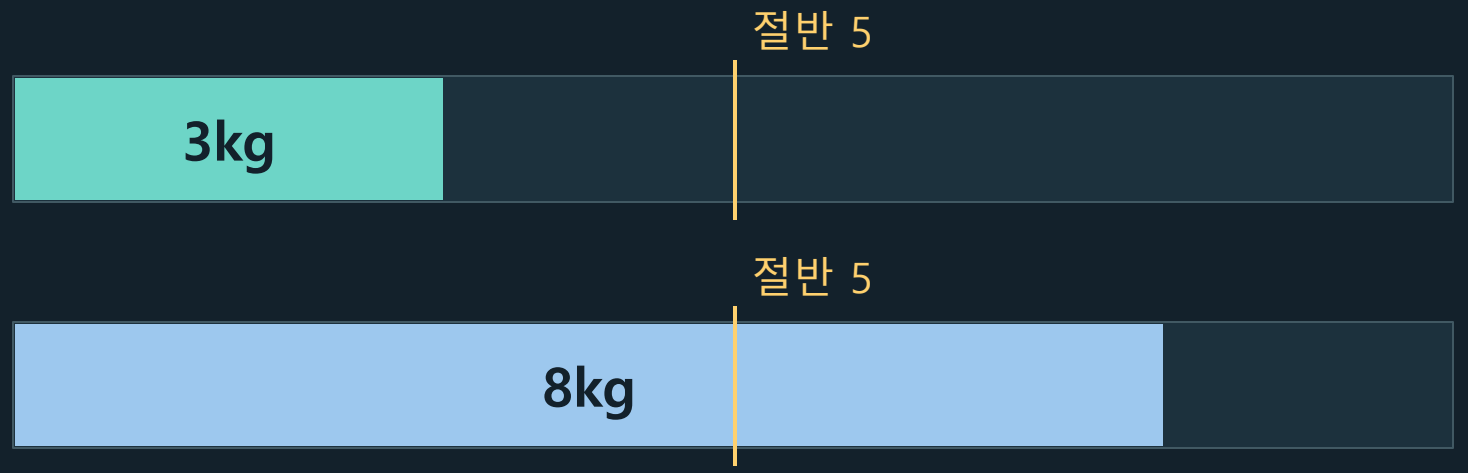
최적 무게는 상자 제한 10 이하입니다.

결과는 total보다 작아지지 않아 최적의 절반 이상입니다.

결과 ≥ total ≥ capacity/2 ≥ 최적 무게/2

9-2 total이 작아도 무거운 single이 보완합니다

제한 10에서 3을 담은 뒤 8은 추가하지 못했습니다.

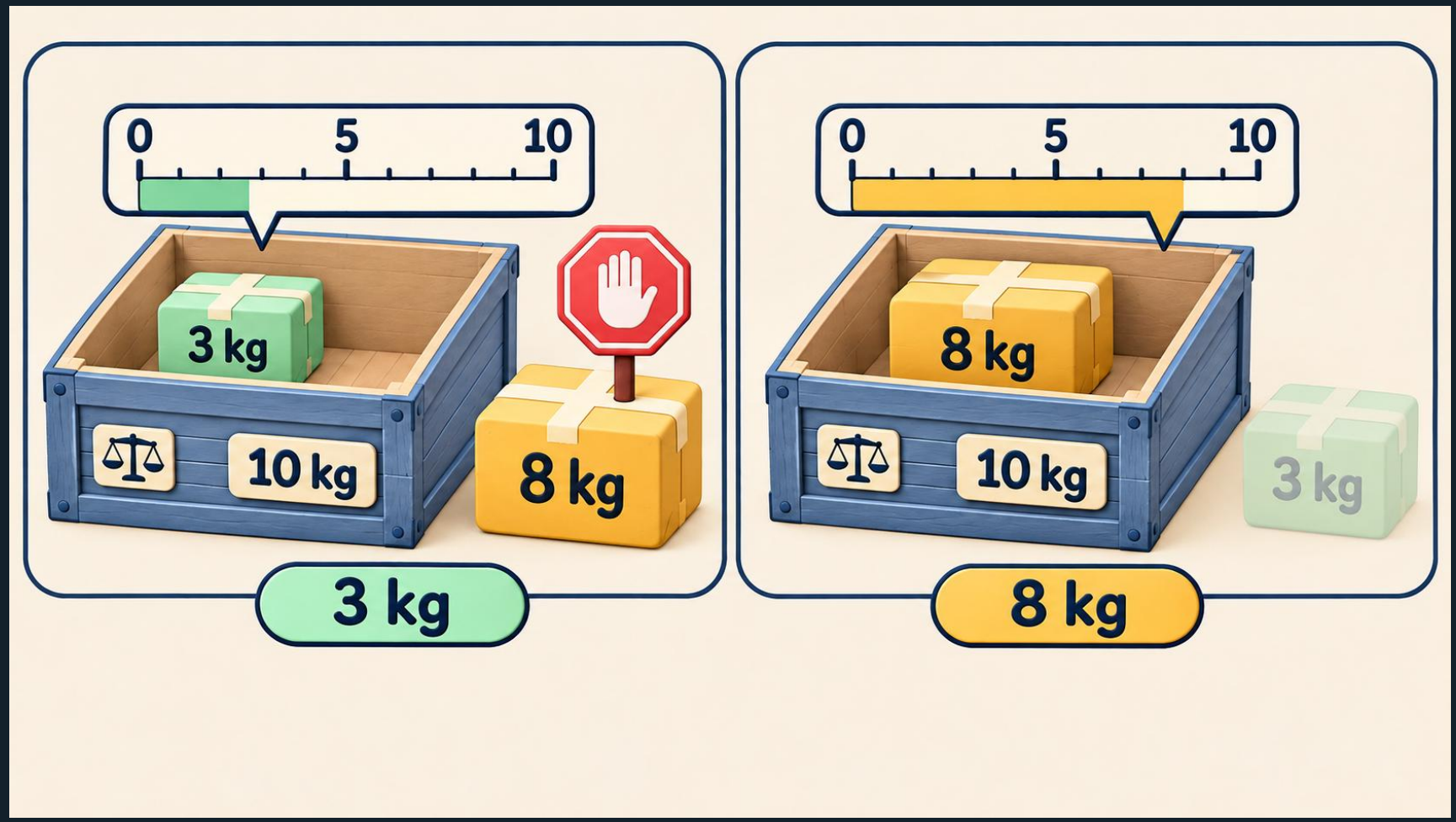


3+8=11이라 total은 3, single은 8입니다.

첫 후보가 작아도 최종 선택은 8kg입니다.

이 물건은 혼자 들어가므로 single 후보로 사용할 수 있습니다.

9-2 3kg에 못 더한 8kg를 혼자 담습니다



3kg를 먼저 담은 계획입니다.

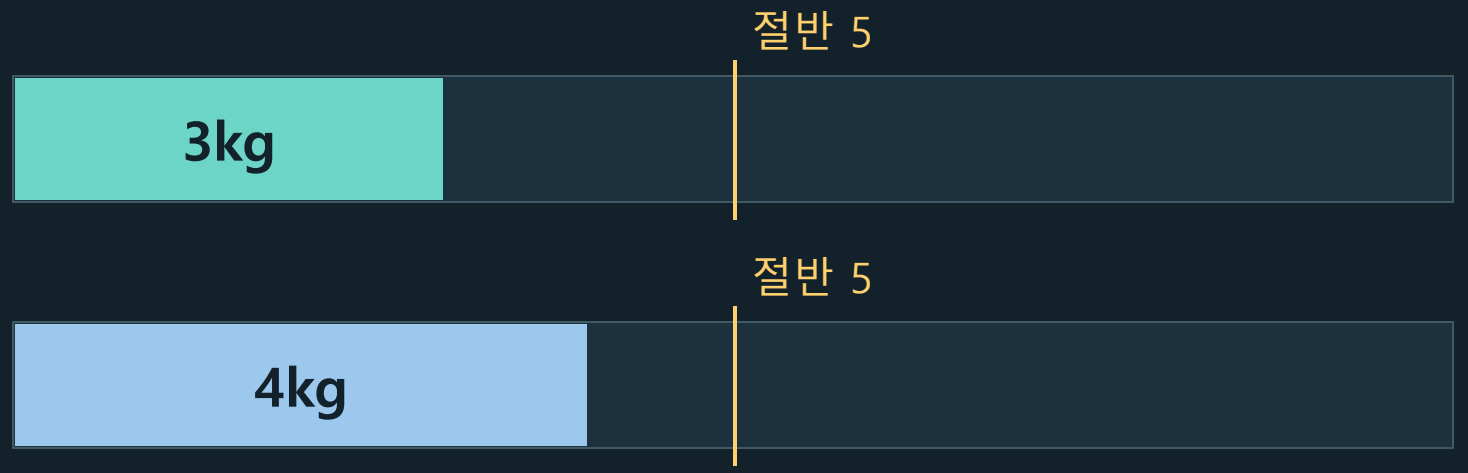
8kg는 더하면 11kg라 못 담습니다.

단일 8kg를 선택하면 절반을 넘습니다.

동일한 물건을 사용해 만든 두 계획입니다. 큰 값 8을 반환합니다.

9-2 거절한 당시 합은 최종 합보다 클 수 없습니다

[3,8,1], 제한 10이면 8을 거절한 당시 합은 3, 최종 합은 4입니다.



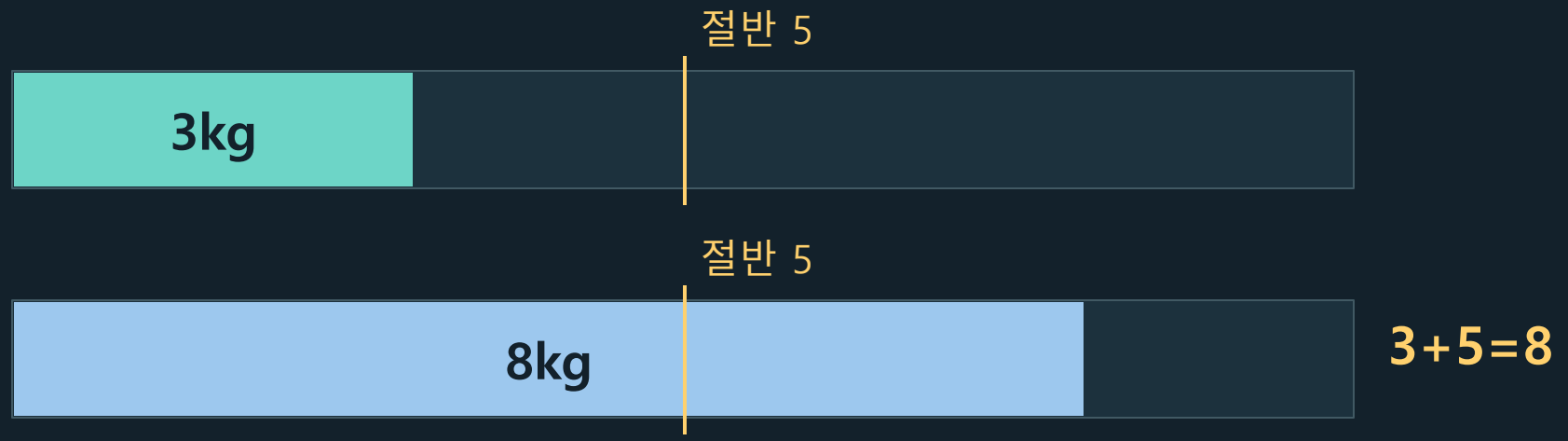
담을 때만 더하므로 total은 줄어들지 않습니다.

최종 합이 절반 미만이면 거절 당시 합도 절반 미만입니다.

거절 당시 합 ≤ 최종 total < capacity/2

9-2 작은 합에 절반 이하 물건은 반드시 들어갑니다

제한의 절반보다 작은 합에, 절반 이하 물건을 더해 봅니다.



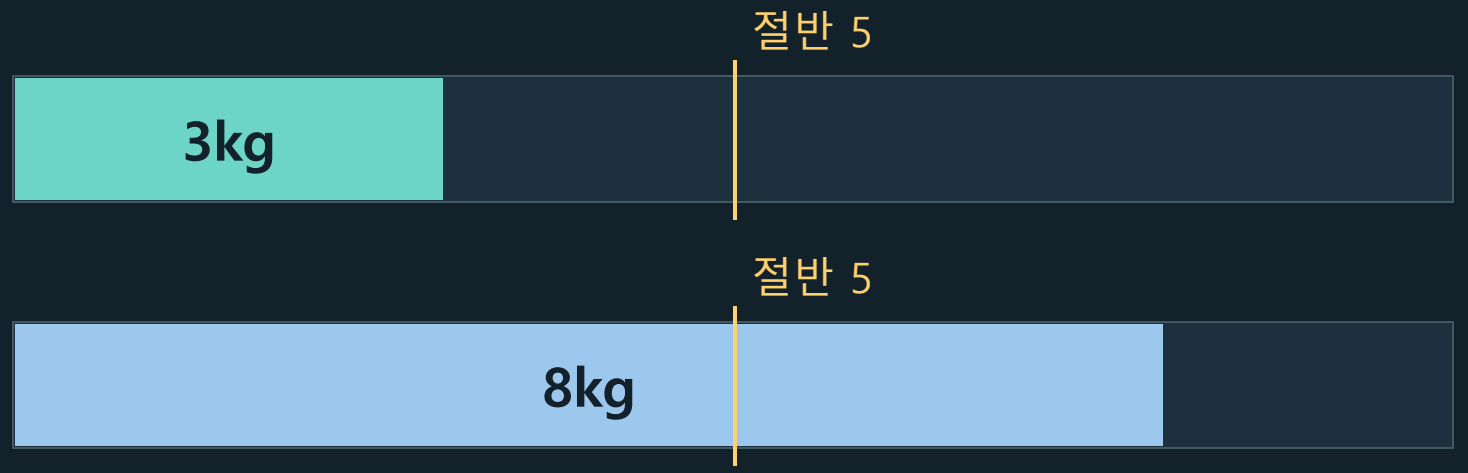
현재 합 3 < 5, 이번 물건 5 ≤ 5이면 합은 8입니다.

두 값을 더해도 제한 10을 넘지 않습니다.

거절 당시 합 < capacity/2이고 weight ≤ capacity/2이면 합 < capacity입니다.

9-2 못 담은 물건이 절반보다 크면 single에 남습니다

혼자 들어가지만 total에 못 넣은 8kg는 single에서 비교합니다.



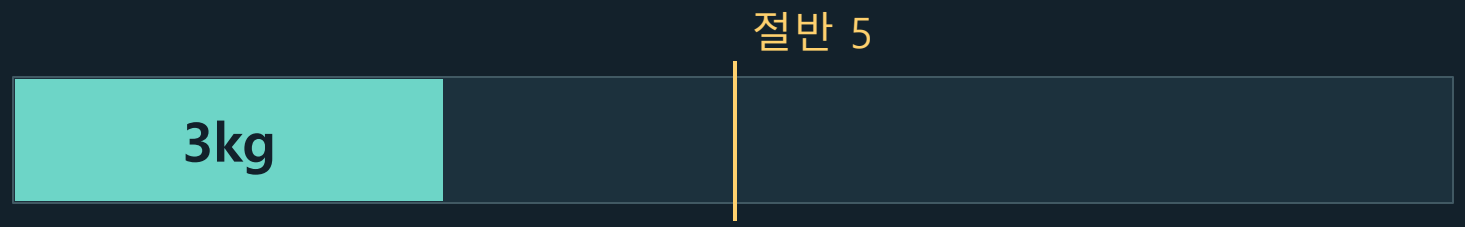
이 물건의 무게가 절반보다 크다는 것을 알았습니다.

single은 그 물건 이상이고, 결과는 single 이상입니다.

결과 ≥ single ≥ 거절한 물건 > capacity/2 ≥ 최적 무게/2

9-2 거절이 없었다면 가능한 물건을 모두 담았습니다

[1,2,15], 제한 10에서는 1과 2를 모두 담고 15는 혼자도 못 담습니다.



15kg는 혼자도 제한 10kg를 넘어서 제외합니다.

혼자 들어가는 물건을 모두 담아 total=3입니다.

어떤 선택도 15를 쓸 수 없어 최적값도 3입니다.

혼자 들어가는 물건을 한 번도 거절하지 않았다면 total 자체가 최적입니다.

9-2 세 경우를 확인하면 절반 이상이 보장됩니다

최종 total을 기준으로 경우를 나누었습니다.

실행 결과	보장을 얻는 이유
total이 제한의 절반 이상	결과 ≥ total ≥ 최적의 절반
total이 작고, 혼자 들어가는 물건을 거절함	그 물건이 절반보다 커서 single이 보완
total이 작고, 그런 거절이 전혀 없음	가능한 물건을 모두 담아 total이 최적

모든 입력에서 결과 ≥ 최적 무게/2이며, 결과는 제한을 넘지 않습니다.

9-2 큰 입력도 같은 검사로 처리합니다

weights=[15,4,7,3,8,2], capacity=12

이번 무게	처리 뒤 total	처리 뒤 single
15	0, 혼자도 못 들어감	0
4	4	4
7	11	7
3	11 유지	7
8	11 유지	8
2	11 유지	8

최종 max(11,8)=11입니다. 직접 고른 7+3+2=12가 이 입력의 최적값입니다.

9-2 최종 답과 경계 입력을 확인합니다

빈 목록과 제한 초과 물건도 같은 코드로 처리합니다.

weights / capacity	반환값	선택 이유
[6,5,5] / 10	6	두 후보 모두 6
[1,10] / 10	10	단일 후보 10
[6,5,4] / 10	10	5를 넘긴 뒤 4 추가
[12,15] / 10	0	혼자도 들어가지 않음
[] / 10	0	두 초기값 그대로

물건 n개를 한 번 읽으므로 시간 O(n), 두 후보를 저장하므로 추가 공간 O(1)입니다.

9-3 모든 선에 닿도록 점을 고릅니다

목표는 모든 선의 조건을 만족하면서, 고른 점의 수를 줄이는 것입니다.

주어진 것 점들과, 두 점 사이를 잇는 선들입니다.

고르는 것 동그라미로 표시한 점을 자유롭게 고릅니다.

꼭 지킬 조건 선 하나마다 양 끝 중 적어도 한 점을 골라야 합니다.

이 조건을 만족하는 점들의 모음을 정점 커버라고 부릅니다.

9-3 선마다 양쪽 끝을 확인합니다

점 번호는 이름입니다. 선의 양 끝 중 적어도 한 점에 색이 있어야 합니다.

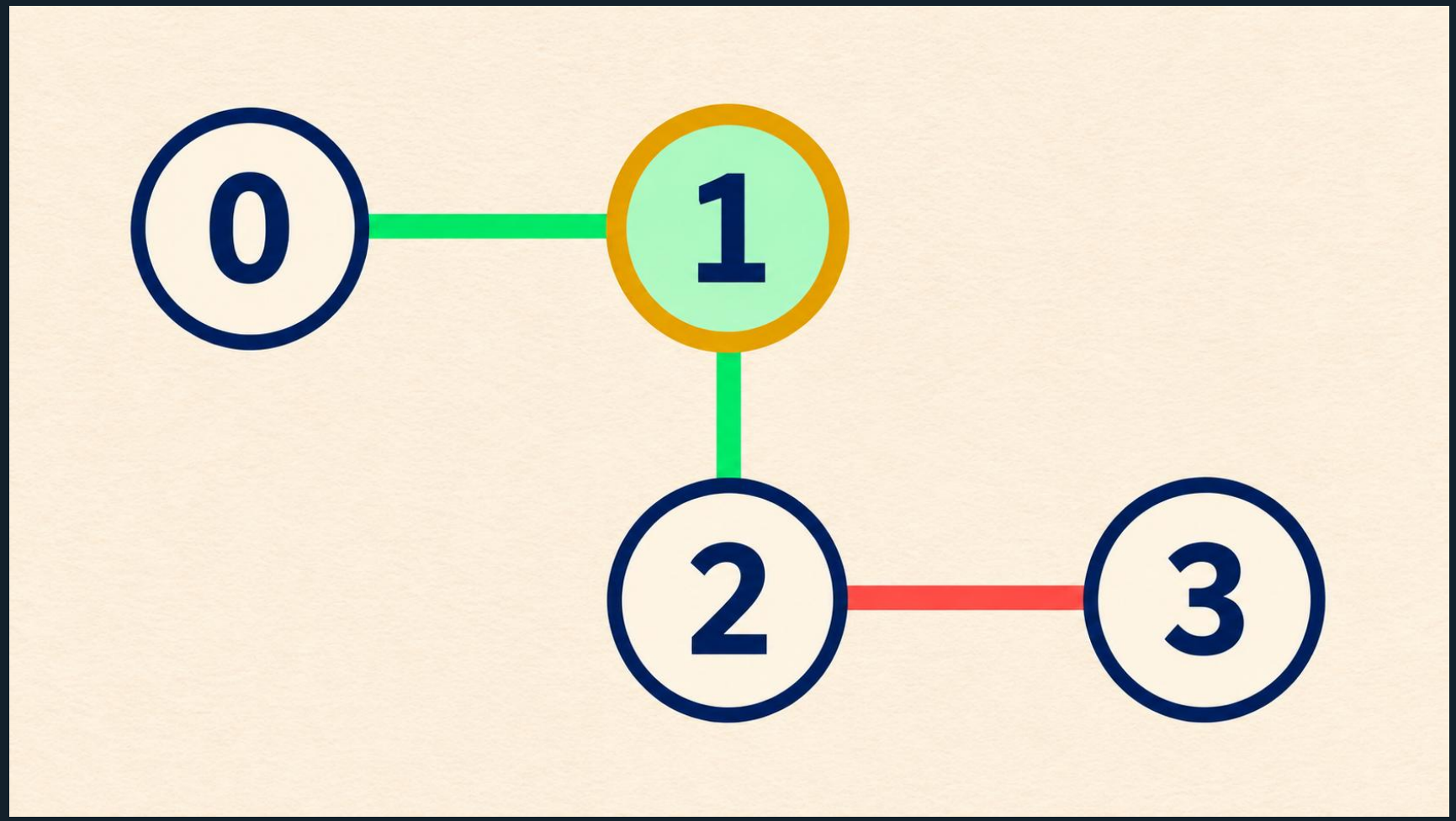


0–1 선은 0번이나 1번을 고르면 됩니다.

1–2 선은 1번이나 2번, 2–3 선은 2번이나 3번입니다.

각 동그라미의 색만 바꿉니다. 선은 처음 주어진 연결 그대로 확인합니다.

9-3 1번 하나로는 마지막 선이 남습니다



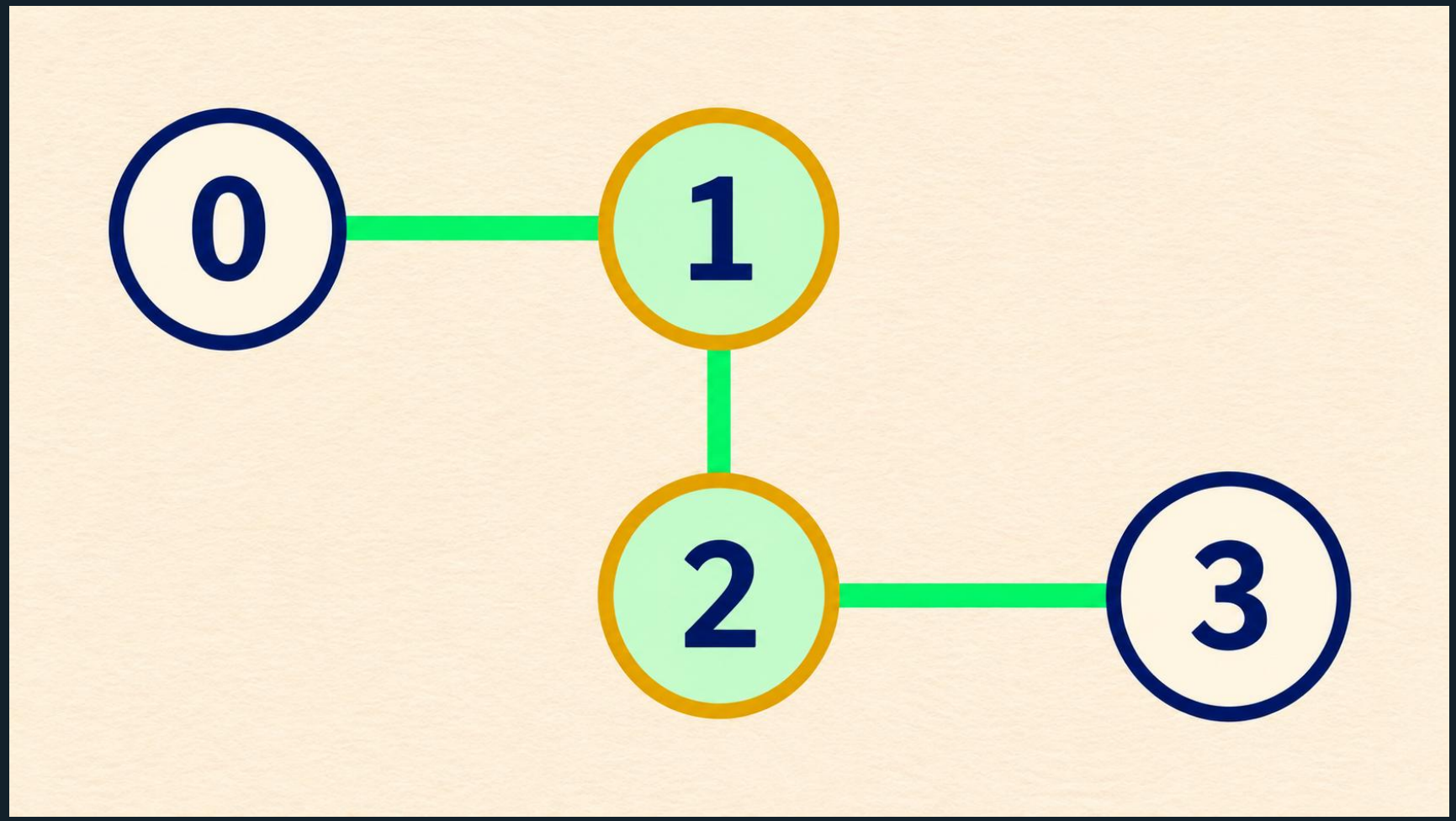
고른 점은 1번 하나입니다.

0–1과 1–2는 초록색입니다.

2–3은 두 끝을 안 골라 빨간색입니다.

1번은 0–1과 1–2에 붙어 있습니다. 2–3을 해결할 점을 더 골라야 합니다.

9-3 2번을 더 고르면 세 선이 모두 해결됩니다



고른 점은 1번과 2번입니다.

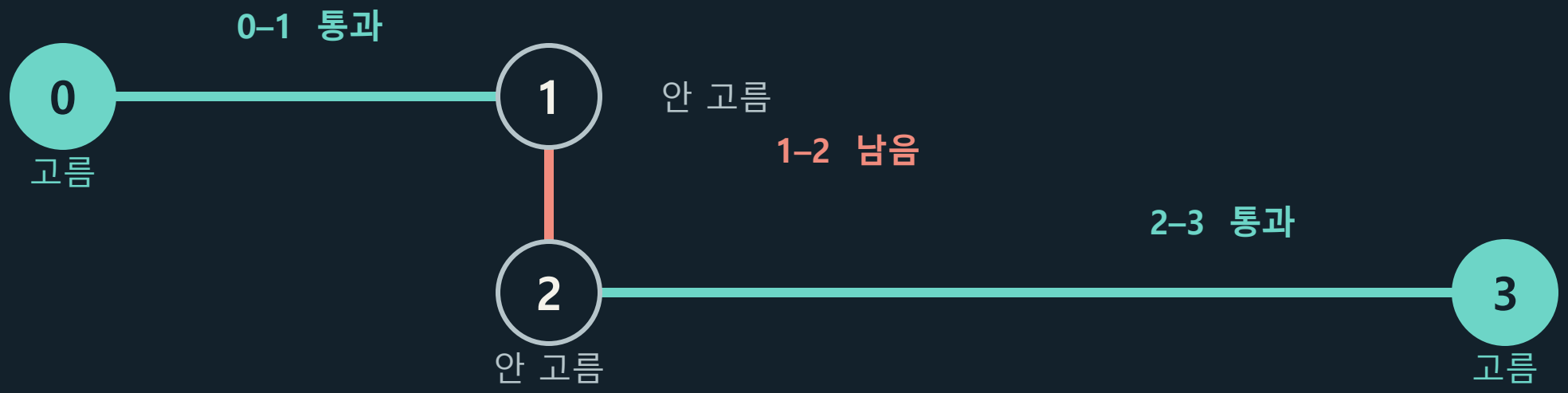
2–3 선에도 고른 2번이 생겼습니다.

세 선 모두 초록색이므로 끝났습니다.

1번과 2번을 고른 상태입니다. 세 선 모두 조건을 만족하고, 고른 점은 2개입니다.

9-3 0번과 3번을 고르면 가운데 선이 남습니다

0번과 3번 동그라미에만 색칠합니다.



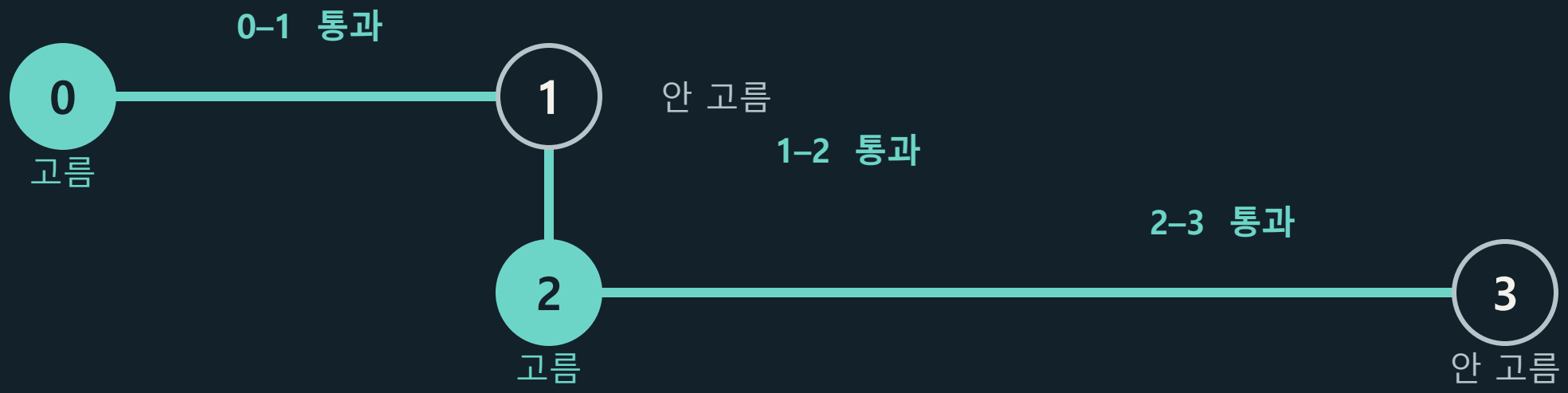
0–1은 0번 덕분에, 2–3은 3번 덕분에 통과합니다.

가운데 1–2의 두 끝은 안 골랐으므로 이 선이 남습니다.

고른 점의 개수와 함께, 모든 선의 조건을 만족했는지도 확인해야 합니다.

9-3 0번과 2번을 골라도 해결됩니다

서로 떨어진 두 점을 골라도 모든 선을 해결할 수 있습니다.



0–1에는 고른 0번이 있습니다.

1–2와 2–3에는 고른 2번이 있어 세 선 모두 통과합니다.

최소 두 점으로 가능한 선택은 {0,2}, {1,2}, {1,3}입니다.

9-3 1번과 3번도 이 그림에서는 가능합니다

처음 그림의 선은 0–1, 1–2, 2–3 세 개입니다.



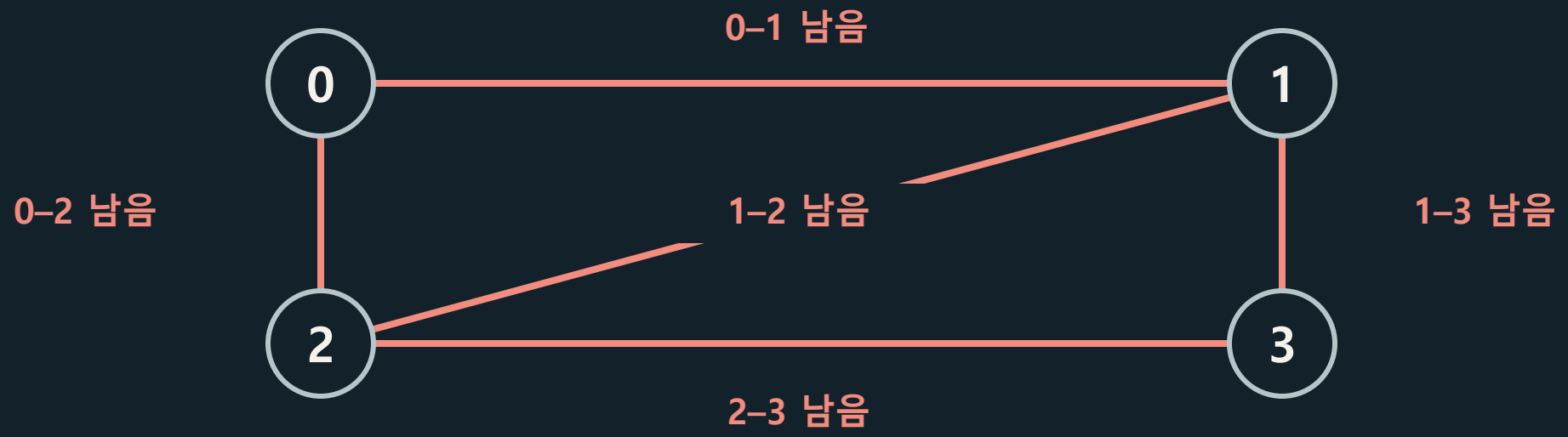
1번이 0–1과 1–2를 해결합니다.

3번이 2–3을 해결하므로 두 점으로 끝납니다.

점을 고르는 결과는 실제로 주어진 선의 연결에 따라 달라집니다.

9-3 이제 선 두 개를 더한 그래프를 사용합니다

앞의 세 선에 0–2와 1–3을 더했습니다. 이제 선은 다섯 개입니다.



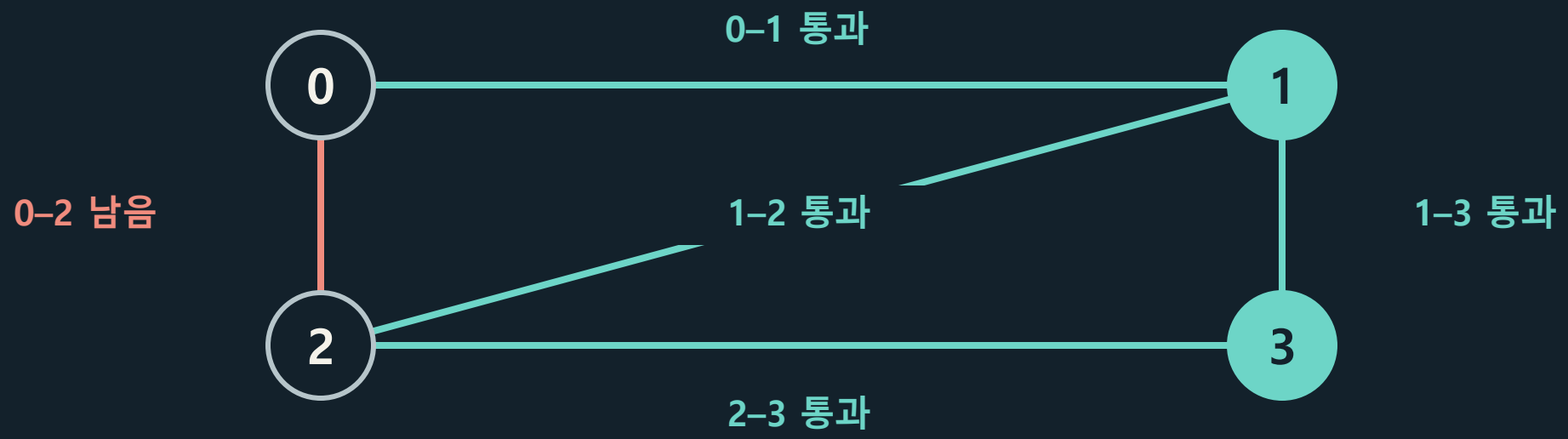
선 목록은 0–1, 1–2, 2–3, 0–2, 1–3입니다.

이후 코드 실행과 2배 설명에는 이 그림을 계속 사용합니다.

배치를 바꾸어 그렸습니다. 선 옆에 적힌 두 점 번호로 연결을 확인합니다.

9-3 1번과 3번을 고르면 0–2가 남습니다

원래 세 선에 0–2와 1–3을 추가했습니다.



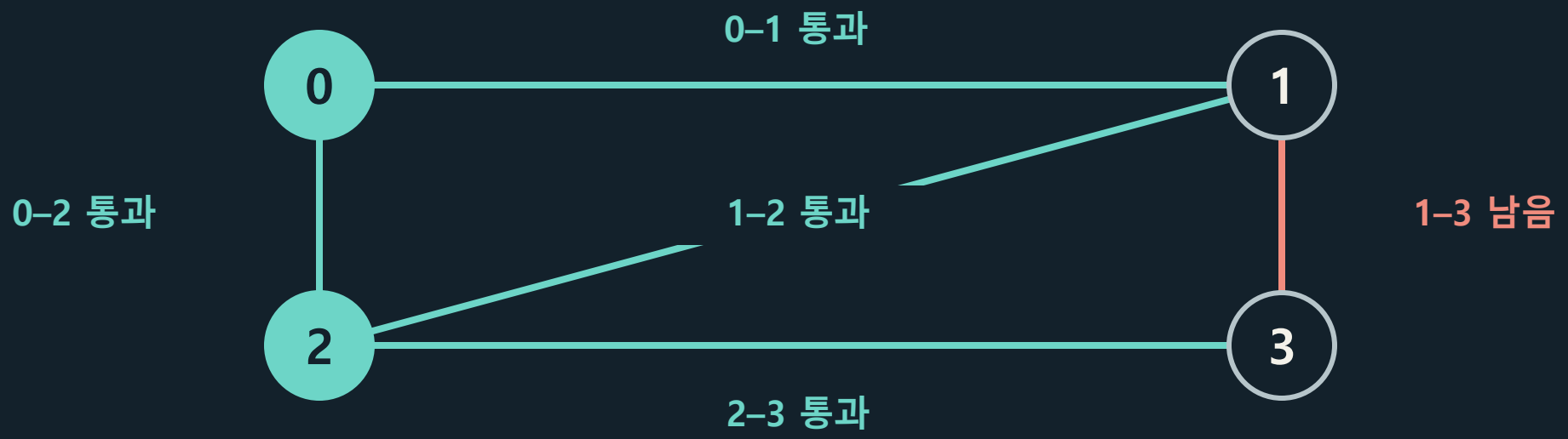
1번과 3번은 0–1, 1–2, 2–3, 1–3을 해결합니다.

추가한 0–2는 양 끝을 모두 안 골라 빨간색으로 남습니다.

홀수 번호만 고르는 선택은 이 그래프의 0–2 선을 해결하지 못합니다.

9-3 0번과 2번을 고르면 1–3이 남습니다

같은 다섯 선에서, 이번에는 0번과 2번만 색칠합니다.



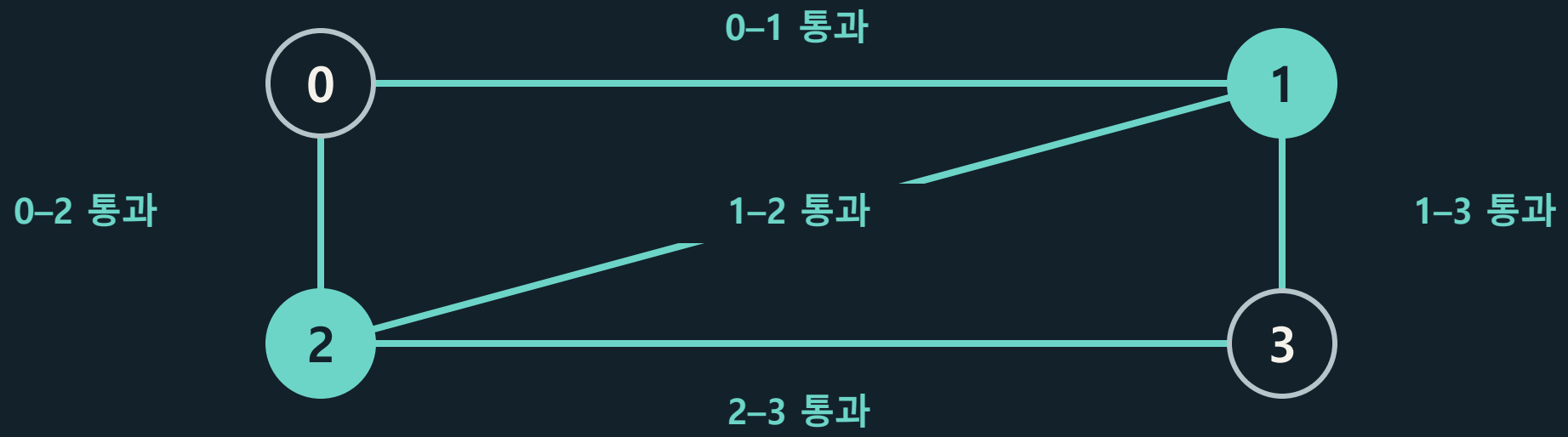
기존 세 선과 추가한 0–2는 조건을 만족합니다.

1–3의 양 끝은 안 골랐으므로 이 선이 남습니다.

짝수 번호만 고르는 선택은 이 그래프의 1–3 선을 해결하지 못합니다.

9-3 1번과 2번이면 다섯 선을 모두 해결합니다

어떤 점 번호인지보다, 어떤 선의 끝에 붙어 있는지를 확인합니다.



1번은 0–1, 1–2, 1–3을 해결합니다.

2번은 1–2, 2–3, 0–2를 해결해 다섯 선 모두 통과합니다.

가능한 선택 하나를 찾았습니다. 다음에는 한 점만으로 가능한지 확인합니다.

9-3 이 그래프의 최적값은 2입니다

0–1에서 최소 1개, 2–3에서 별도의 최소 1개입니다.



0–1에 최소 1개

2–3에 별도로 1개

최적값 = 2

두 노란 선을 해결하려면 적어도 두 점이 필요합니다.

1번과 2번 두 점으로 다섯 선 전체를 해결할 수 있습니다.

노란색은 꼭 확인할 선입니다. 흐린 선도 원래 그래프에 그대로 있습니다.

9-3 큰 그래프에서는 빠른 선택 규칙을 씁니다

점이 많아지면 가능한 점 조합도 빠르게 늘어납니다.

점 4개 각 점을 고르거나 빼면 가능한 조합은 16개입니다.

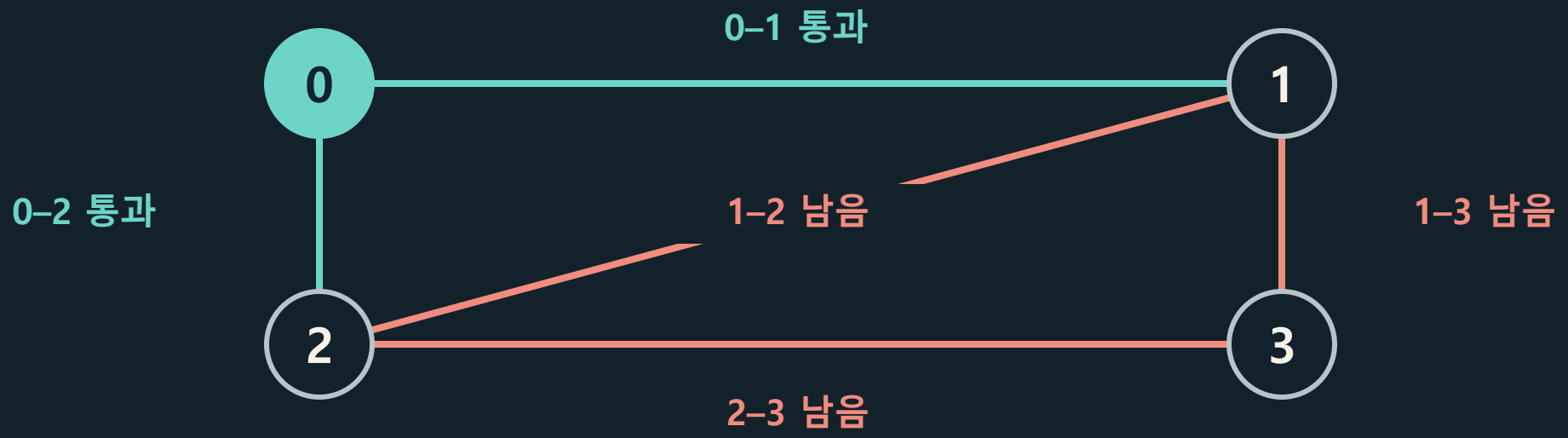
점 30개 가능한 조합은 2^{30}개, 약 10억 개입니다.

이번 방법 선 목록을 한 번 읽으면서 사용할 점을 정합니다.

계산을 빠르게 마치면서, 고른 개수가 최적의 2배 이내임을 보장합니다.

9-3 한 선만 보면 한 끝점으로 충분합니다

0–1 선은 0번 하나만 골라도 해결됩니다.



0–1과 0–2는 고른 0번이 붙어 있어서 해결됩니다.

아직 1–2, 2–3, 1–3은 남습니다.

이번 빠른 규칙은 양쪽이 모두 비어 있으면 두 끝을 함께 고릅니다.

9-3 아직 비어 있는 선을 만나면 두 끝을 고릅니다

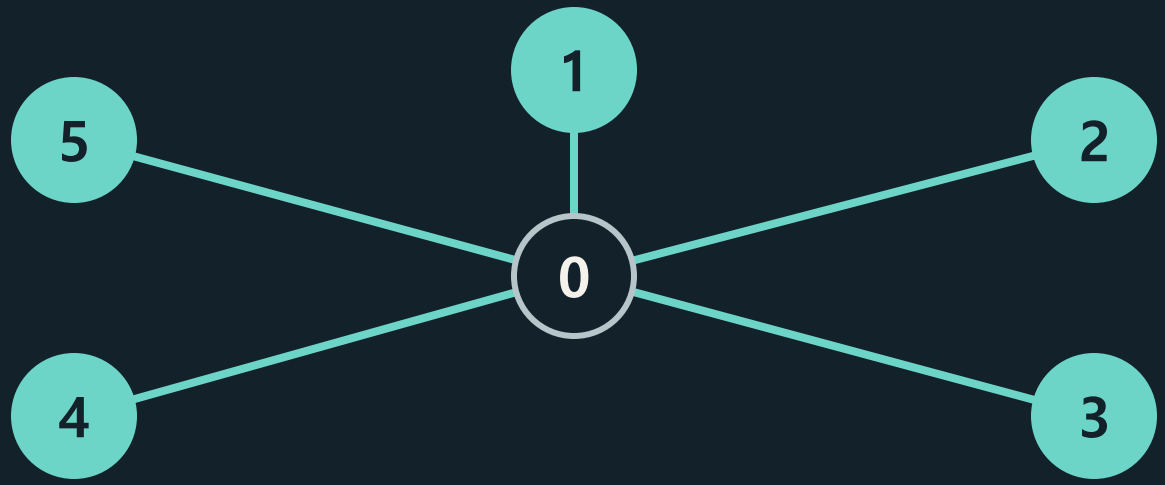
입력된 선을 앞에서부터 읽습니다. 고른 점은 계속 유지합니다.

처음	아무 점도 고르지 않습니다.
양 끝이 비었으면	두 점을 함께 고르고, 개수에 2를 더합니다.
한쪽이라도 골랐으면	그 선은 해결되었으므로 다음 선으로 갑니다.

다음 선에서도 지금까지 고른 점을 그대로 사용합니다.

9-3 한쪽을 계속 잘못 고르면 점을 많이 씁니다

다섯 선 모두 가운데 0번에 붙어 있습니다. 최적값은 1입니다.



고른 점 5개

바깥 점 다섯 개를 고름

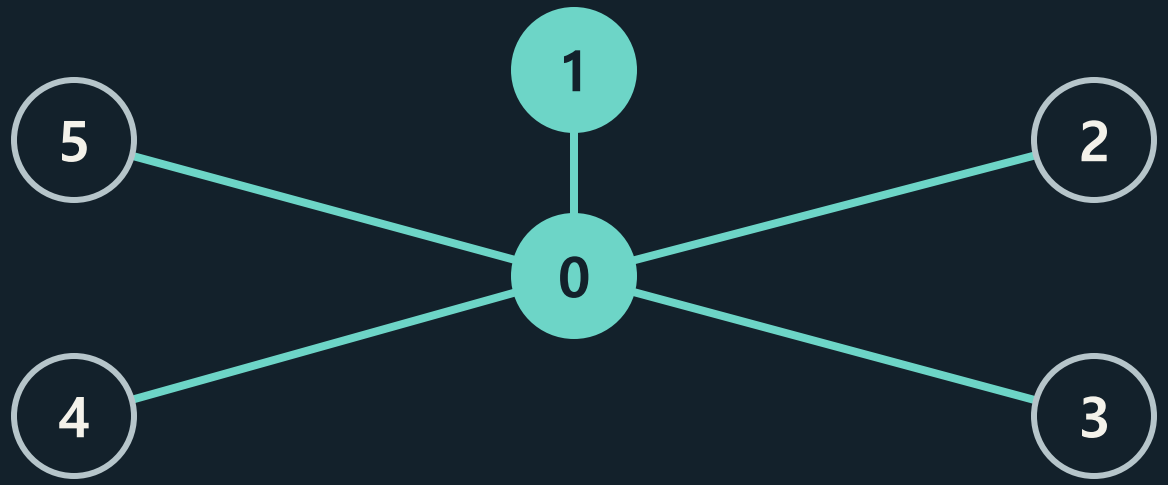
선마다 바깥쪽 끝만 고르면 1·2·3·4·5, 다섯 점을 씁니다.

가운데 0번 하나면 다섯 선을 모두 해결할 수 있습니다.

이 그림에서 바깥 점만 계속 고르면 5개를 사용합니다.

9-3 두 끝을 함께 고르면 이 그림은 두 점으로 끝납니다

처음 읽은 선이 1–0이면, 1번과 0번을 함께 고릅니다.



고른 점 2개

가운데 0번과 바깥 1번

0번이 나머지 모든 선의 끝에도 붙어 있습니다.

이후 선에서는 추가하지 않아 두 점으로 끝납니다.

이 예에서는 최적 1개, 빠른 규칙 2개입니다. 다음 장부터 원래 다섯 선으로 돌아갑니다.

9-3 함수에 주는 입력과 반환할 값

cover_size(n, edges)

n = 4	점이 네 개입니다. 점 번호는 0, 1, 2, 3입니다.
edges	[[0,1],[1,2],[2,3],[0,2],[1,3]]
반환할 값	정한 규칙으로 실제 고른 점의 개수를 반환합니다.
입력 조건	점은 1~100개, 선은 0~1000개입니다. 각 선의 두 점은 서로 다릅니다.

선 한 쌍 [0,1]은 0번과 1번 사이의 선입니다. 쌍의 입력 순서를 지킵니다.

9-3 chosen은 점마다 고른 상태를 적는 목록입니다

False는 안 고름, True는 고름입니다. answer에는 고른 개수를 적습니다.

점 번호	0	1	2	3
처음 chosen	False	False	False	False
0번과 1번을 고른 뒤	True	True	False	False

chosen[1]은 1번 점의 상태입니다. 처음 answer는 0이고, 두 점을 고르면 2입니다.

9-3 u와 v에는 지금 읽는 선의 두 점 번호를 넣습니다

한 쌍의 앞 번호가 u, 뒤 번호가 v입니다.

읽는 순서	읽은 선	u	v
첫째	[0,1]	0	1
둘째	[1,2]	1	2
셋째	[2,3]	2	3
넷째	[0,2]	0	2
다섯째	[1,3]	1	3

u와 v가 다음 선의 번호로 바뀌어도, chosen과 answer의 기록은 유지합니다.

9-3 두 점 모두 안 골랐는지 확인하는 조건

not은 True와 False를 뒤집고, and는 두 조건을 함께 확인합니다.

표현	값을 읽는 방법
chosen[u]	u번 점을 골랐으면 True, 안 골랐으면 False
not chosen[u]	u번 점을 안 골랐으면 True
A and B	A와 B가 모두 True일 때만 True
not chosen[u] and not chosen[v]	u번도 안 골랐고, v번도 안 골랐을 때 True

조건이 True이면 두 점을 추가하고, False이면 다음 선으로 갑니다.

9-3 Python 전체 답안

```python
def cover_size(n, edges):
    chosen = [False] * n
    answer = 0
    for u, v in edges:
        if not chosen[u] and not chosen[v]:
            chosen[u] = True
            chosen[v] = True
            answer += 2
    return answer
```

chosen은 점별 선택 기록

answer는 고른 개수

두 끝이 비어 있으면 두 점 추가

모든 선 뒤에 개수 반환

다음 장부터 입력값을 넣어 코드의 각 줄을 따라갑니다.

9-3 먼저 네 점의 기록과 개수를 준비합니다

```
def cover_size(n, edges):
    chosen = [False] * n
    answer = 0
```

점 번호	0	1	2	3
처음 chosen	False	False	False	False
뜻	안 고름	안 고름	안 고름	안 고름

[False] * n

n=4이므로 네 칸을 만듭니다.

answer=0은 고른 개수입니다.

False는 안 고름, True는 고름입니다.

9-3 for 문이 선 목록을 한 쌍씩 꺼냅니다

```
04        for u, v in edges:
```

반복 순서	읽은 선	u	v
1	[0,1]	0	1
2	[1,2]	1	2
3	[2,3]	2	3
4	[0,2]	0	2
5	[1,3]	1	3

u, v는 점 번호

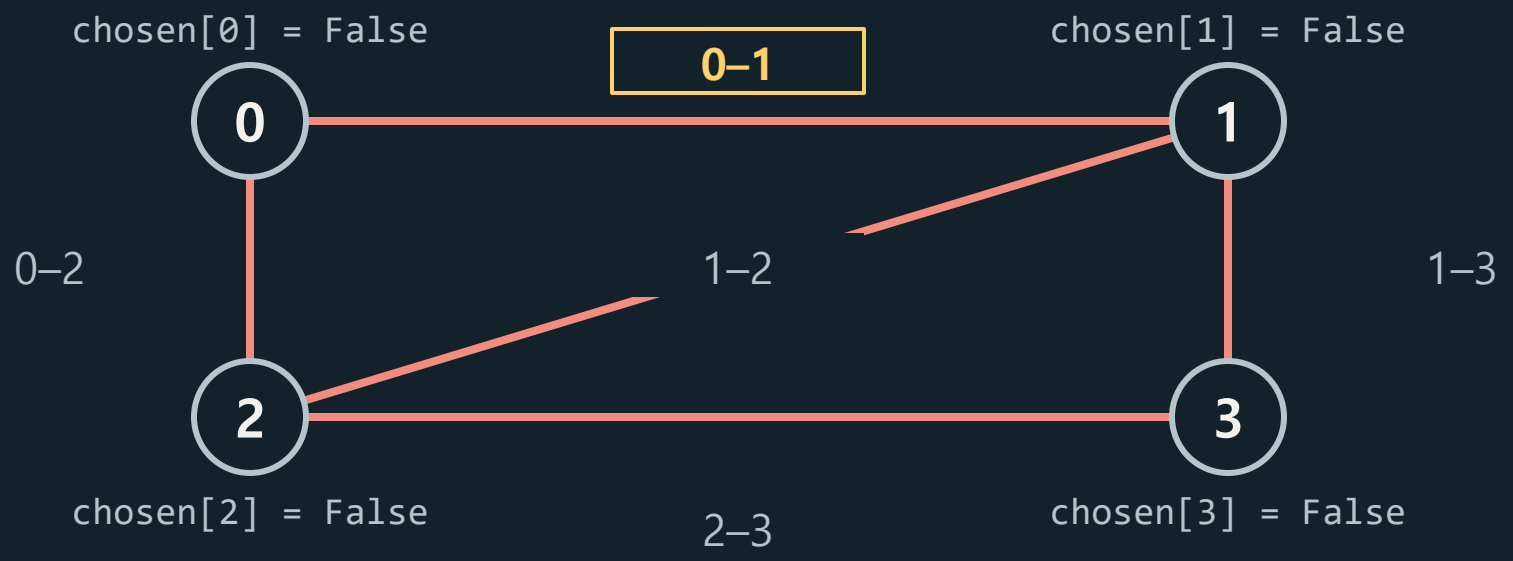
한 쌍의 앞 번호가 u,

뒤 번호가 v입니다.

for 문은 선 목록에서 한 쌍씩 꺼냅니다.

9-3 첫 선 0–1은 양쪽 모두 False입니다

```
if not chosen[0] and not chosen[1]:
```




```
not False and not False

True and True
```

두 점을 추가합니다.

두 점 추가한 선

chosen[0]과 chosen[1]이 모두 False이므로 조건은 True입니다.

이제 0번과 1번을 함께 고르고 answer에 2를 더합니다.

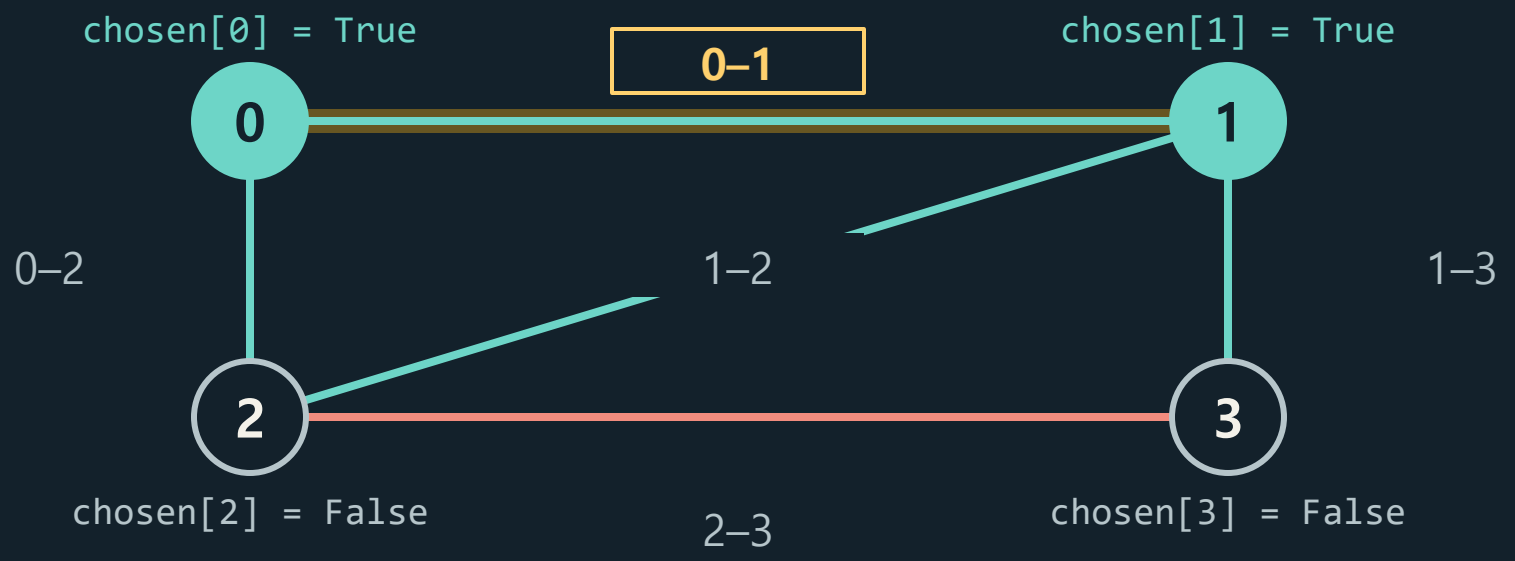
테두리는 지금 확인하는 선, 굵은 밑선은 두 점을 추가했던 선입니다.

9-3 0번과 1번을 고르면 기록이 계속 남습니다

```
6  chosen[0] = True
```

```
7  chosen[1] = True
```

```
8  answer += 2
```



0번과 1번을 선택

answer = 0 + 2 = 2

두 점 추가한 선 1개

두 점 추가한 선
0–1

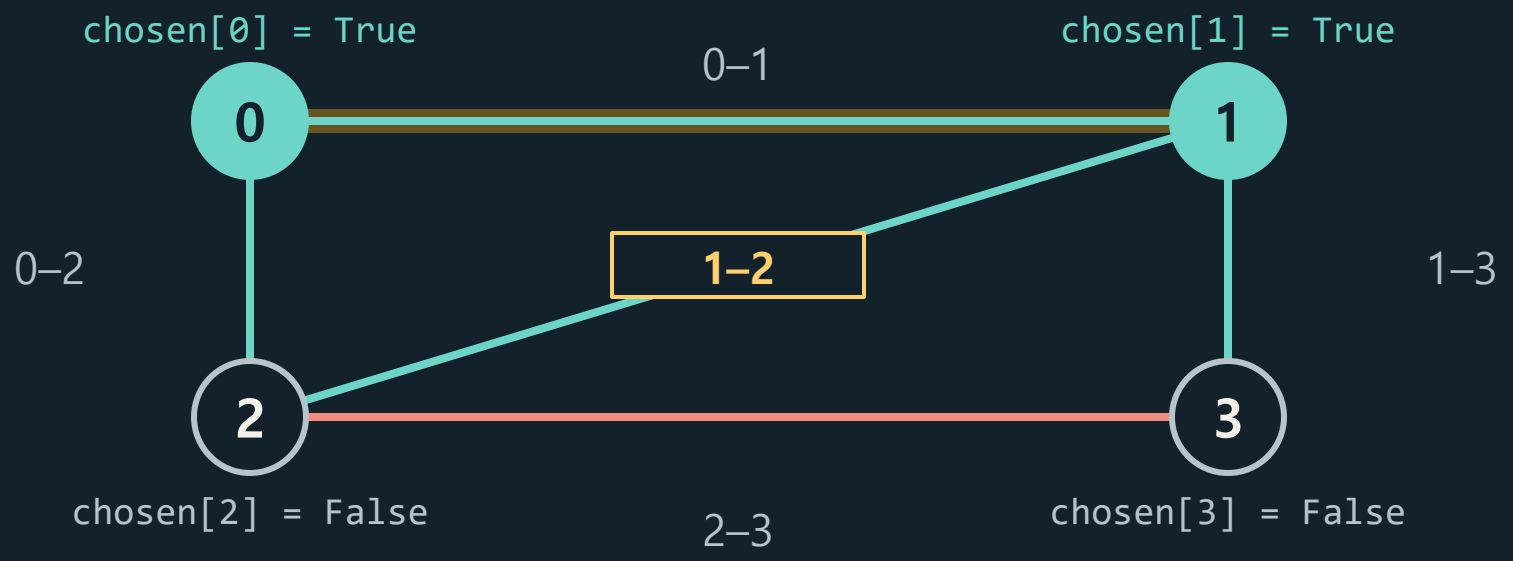
두 칸을 True로 바꾸고, 고른 개수를 0에서 2로 바꿉니다.

다음 선을 읽을 때도 0번과 1번은 True로 남습니다.

테두리는 지금 확인하는 선, 굵은 밑선은 두 점을 추가했던 선입니다.

9-3 다음 1–2는 이미 고른 1번을 공유합니다

```
if not chosen[1] and not chosen[2]:
```




```
not True and not False

False and True
```

추가 없이 answer = 2

두 점 추가한 선
0–1

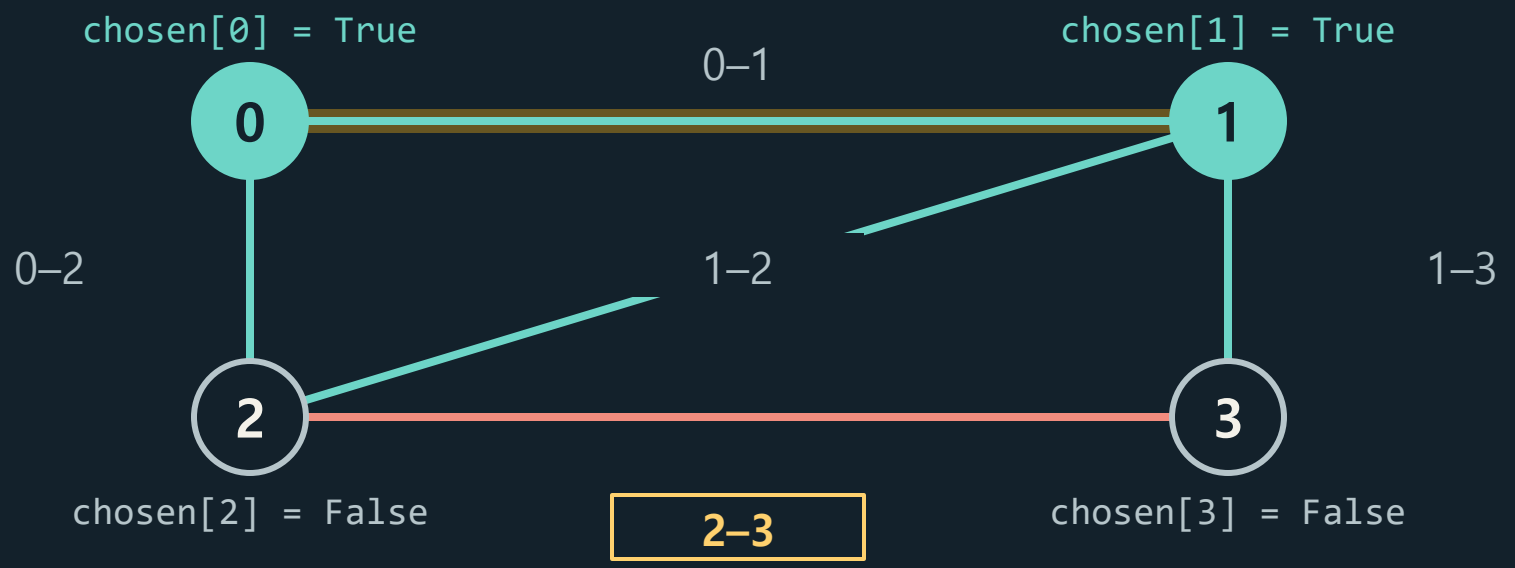
1번이 이미 True이므로 조건은 False입니다.

1–2는 새로 두 점을 고른 선에 들어가지 않습니다.

테두리는 지금 확인하는 선, 굵은 밑선은 두 점을 추가했던 선입니다.

9-3 다음 2–3은 두 점 모두 아직 비어 있습니다

```
if not chosen[2] and not chosen[3]:
```




```
not False and not False

True and True
```

두 점을 추가합니다.

두 점 추가한 선
0–1

2번과 3번은 앞에서 고른 0번·1번과 모두 다른 점입니다.

조건이 True이므로 2번과 3번을 함께 고를 수 있습니다.

테두리는 지금 확인하는 선, 굵은 밑선은 두 점을 추가했던 선입니다.

9-3 2번과 3번을 추가하면 answer는 4입니다

```
6   chosen[2] = True
```

```
7   chosen[3] = True
```

```
8   answer += 2
```



2번과 3번을 선택

answer = 2 + 2 = 4

두 점 추가한 선 2개

두 점 추가한 선
0–1, 2–3

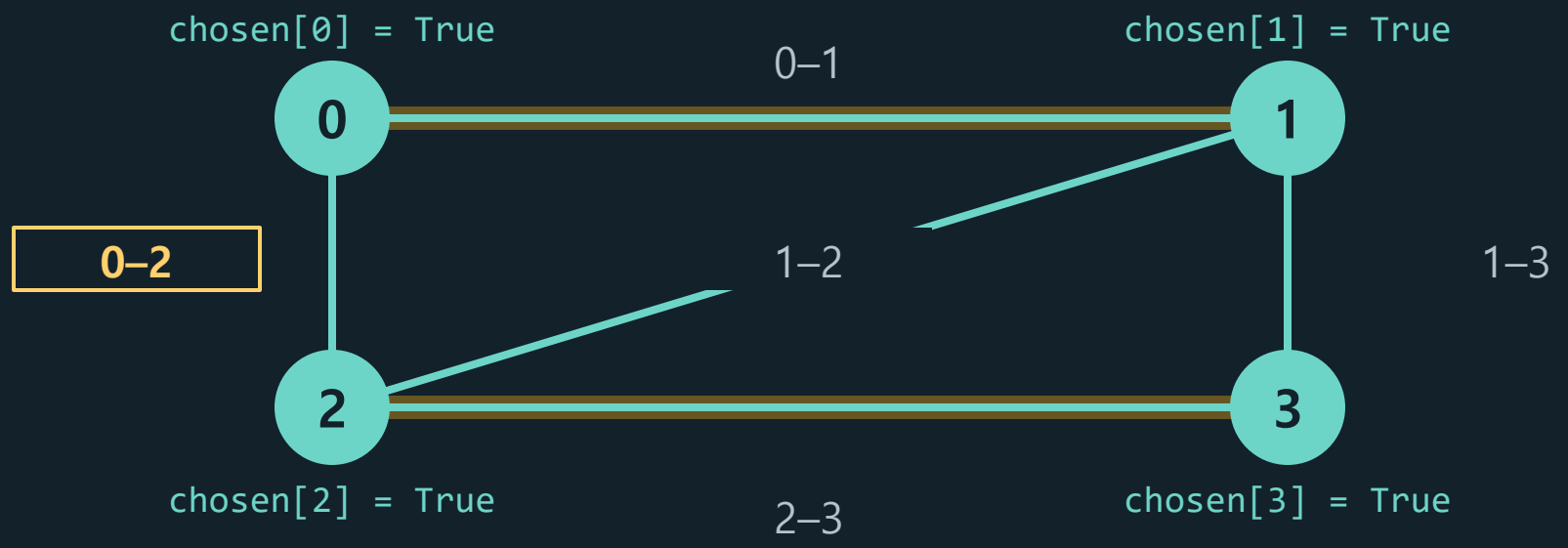
chosen은 [True,True,True,True]가 되었습니다.

두 점을 추가한 선은 0–1과 2–3입니다. 끝점 네 개가 모두 다릅니다.

테두리는 지금 확인하는 선, 굵은 밑선은 두 점을 추가했던 선입니다.

9-3 0–2는 이미 고른 점이 양쪽에 있습니다

```
if not chosen[0] and not chosen[2]:
```




```
not True and not True

False and False
```

추가 없이 answer = 4

두 점 추가한 선
0–1, 2–3

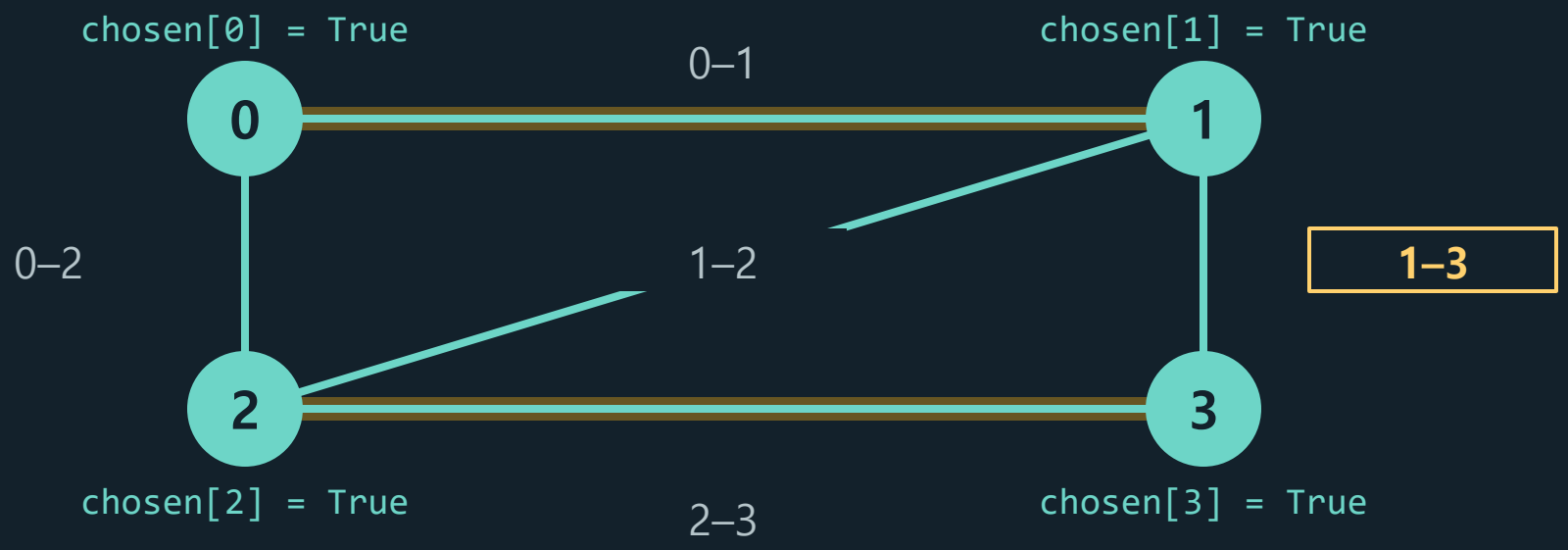
0번과 2번 모두 True이므로 조건은 False입니다.

0–2는 두 점을 새로 추가한 선에 들어가지 않습니다.

테두리는 지금 확인하는 선, 굵은 밑선은 두 점을 추가했던 선입니다.

9-3 마지막 1–3도 추가 없이 넘어갑니다

```
if not chosen[1] and not chosen[3]:
```




```
not True and not True

False and False
```

추가 없이 answer = 4

두 점 추가한 선
0–1, 2–3

1번과 3번도 이미 True라서 추가하지 않습니다.

다섯 선을 모두 읽었고 최종 개수는 4입니다.

테두리는 지금 확인하는 선, 굵은 밑선은 두 점을 추가했던 선입니다.

9-3 모든 선을 읽은 뒤 4를 반환합니다

```
04        for u, v in edges:
05            if not chosen[u] and not chosen[v]:
06                chosen[u] = True
07                chosen[v] = True
08                answer += 2
09        return answer
```

실행 지점	값
처음	answer = 0
0–1 선택 뒤	answer = 2
2–3 선택 뒤	answer = 4
반복이 끝난 뒤	return 4

return은 for와 같은 들여쓰기입니다.

선 다섯 개를 모두 읽은 다음

현재 answer를 돌려줍니다.

반환값은 고른 점의 개수 4입니다.

9-3 같은 선택 규칙을 C로 작성합니다

```c
int cover_size(int n, const int edges[][2], int m) {
    int chosen[100] = {0};
    int answer = 0;
    for (int i = 0; i < m; ++i) {
        int u = edges[i][0];
        int v = edges[i][1];
        if (!chosen[u] && !chosen[v]) {
            chosen[u] = 1;
            chosen[v] = 1;
            answer += 2;
        }
    }
    return answer;
}
```

0은 안 고름, 1은 고름

n은 점 수, m은 선 수

i는 선 목록의 행 번호

최종 반환값도 4

선의 순서와 선택 규칙은 Python과 같습니다.

9-3 C의 입력과 선택 기록을 준비합니다

```
01  int cover_size(int n, const int edges[][2], int m) {
02      int chosen[100] = {0};
03      int answer = 0;
```

코드	뜻
int cover_size(...)	함수가 정수 결과를 반환합니다.
const int edges[][2]	각 행에 두 점 번호가 있는 입력 표
int chosen[100] = {0};	정수 100칸을 모두 0으로 만듭니다.
int answer = 0;	고른 개수를 0으로 시작합니다.

점은 최대 100개입니다.

이 예제는 chosen의 0~3번 칸을 씁니다.

const는 입력 표를 보존하겠다는 표시입니다.

chosen의 0은 안 고름입니다. 선택할 때 해당 칸을 1로 바꿉니다.

9-3 C는 행 번호 i로 선을 하나씩 읽습니다

```
04        for (int i = 0; i < m; ++i) {
05            int u = edges[i][0];
06            int v = edges[i][1];
```

표현	예제에서 하는 일
int i = 0	첫 행 번호 0으로 시작
i < m	m=5이므로 i가 0~4일 때 반복
++i	한 선을 처리한 뒤 i를 1 증가
edges[i][0], edges[i][1]	그 행의 두 점 번호를 u와 v로 읽음

첫 행은 [0,1]입니다.

i=0일 때 u=0, v=1입니다.

다음 행에서는 u=1, v=2입니다.

i는 선의 행 번호이고, u와 v는 그 선에 붙은 점 번호입니다.

9-3 C도 양쪽이 0일 때만 두 점을 추가합니다

```
07          if (!chosen[u] && !chosen[v]) {
08              chosen[u] = 1;
09              chosen[v] = 1;
10              answer += 2;
```

첫 선 [0,1]	둘째 선 [1,2]
!0 && !0	!1 && !0
참 && 참 = 참	거짓 && 참 = 거짓
두 칸을 1로, 개수 +2	아래 세 줄을 건너뜀

!0은 참입니다.

!1은 거짓입니다.

&&는 양쪽이 모두 참이어야 합니다.

조건이 참일 때만 중괄호 안에서 두 기록과 개수를 함께 바꿉니다.

9-3 C도 반복문 밖에서 결과를 반환합니다

```
11              }
12          }
13          return answer;
14      }
```

위치	하는 일
11행 }	if 범위 끝
12행 }	다섯 선을 읽는 for 범위 끝
13행 return answer	최종 4 반환

첫 선 뒤 2

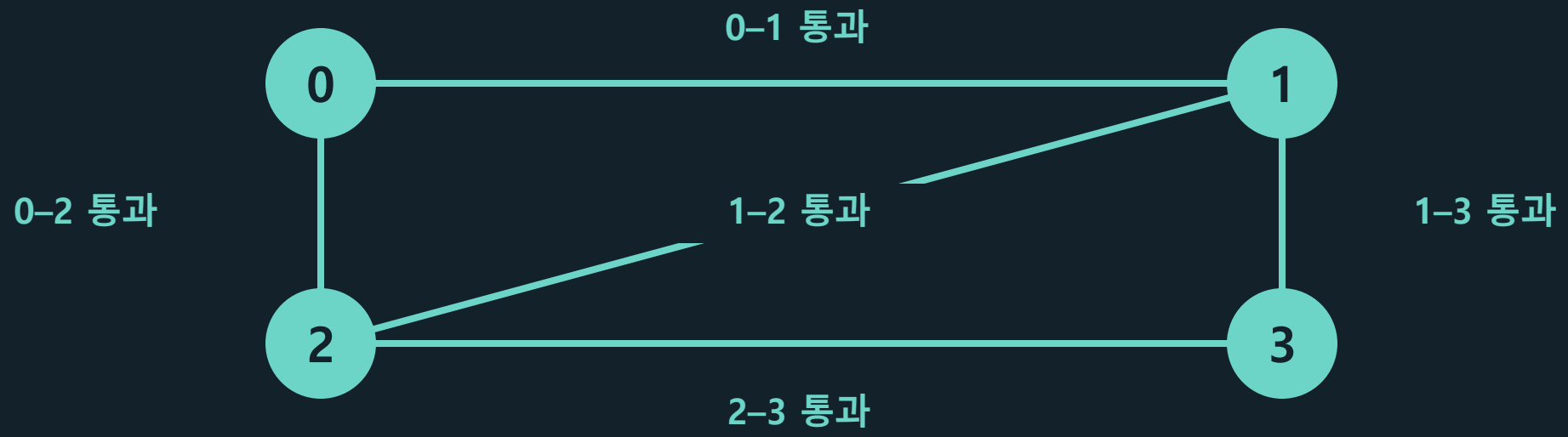
셋째 선 뒤 4

넷째·다섯째 선 뒤에도 4

중괄호가 반복문과 조건문의 범위를 구분합니다.

9-3 반환할 때 모든 선이 해결되는 이유

선 하나를 읽을 때마다 그 선이 해결된 상태로 다음 선으로 갑니다.



이미 고른 끝이 있으면 그 끝이 선을 해결합니다.

양 끝이 비어 있으면 둘 다 골라 그 선을 해결합니다.

한 번 고른 점을 빼지 않으므로 앞에서 해결한 선도 계속 해결된 상태입니다.

9-3 같은 그래프에서 최적은 2, 이번 결과는 4입니다

정한 규칙을 실행한 결과와, 가능한 최소 개수를 비교합니다.

직접 찾은 최적 선택 1번과 2번을 고르면 전체를 해결합니다. 2개입니다.

이번 입력 순서 0–1, 1–2, 2–3, 0–2, 1–3을 차례로 읽었습니다.

코드가 고른 점 0번·1번을 고른 뒤 2번·3번도 골라 4개입니다.

최적값 2에 비해 사용한 점은 4개, 정확히 2배입니다.

9-3 1–2를 먼저 읽으면 반환값은 2입니다

선의 연결은 같고 입력 목록의 순서만 바꾸었습니다.

바꾼 입력 [[1,2],[0,1],[2,3],[0,2],[1,3]]

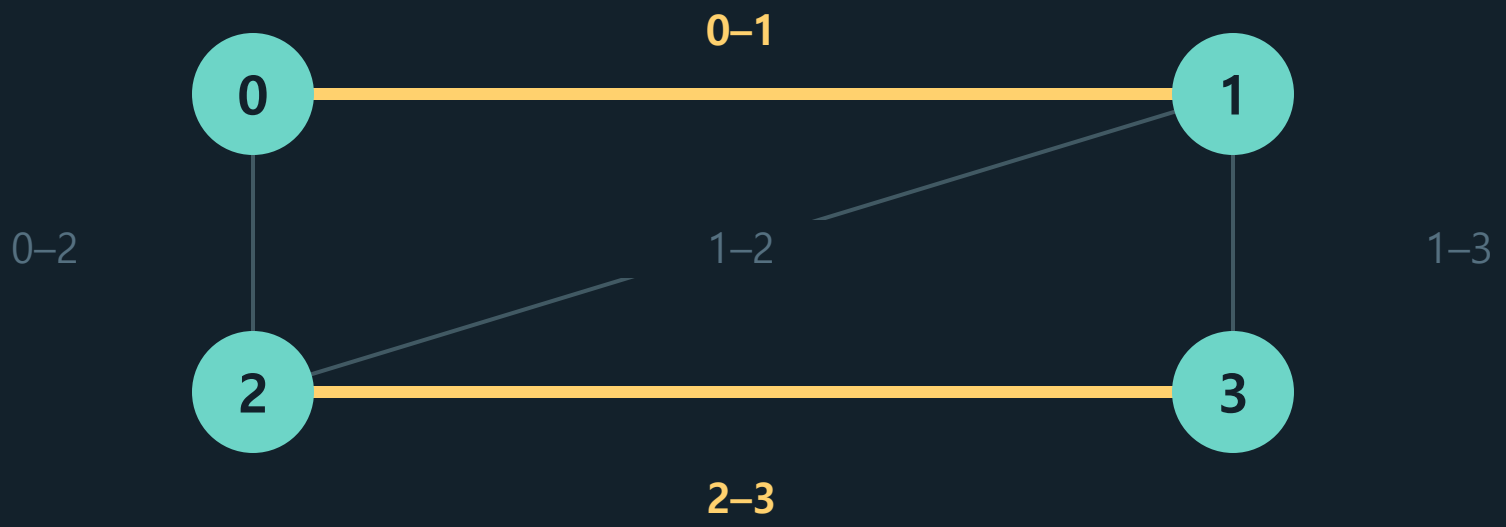
첫 선 1–2 1번과 2번을 고릅니다. 개수는 2입니다.

남은 네 선 모두 1번이나 2번에 붙어 있어 추가하지 않습니다.

이번 실행은 최적 선택과 같습니다. 주어진 선 순서대로 계산합니다.

9-3 두 점을 추가했던 선만 노란색으로 표시합니다

원래 순서에서 두 점을 추가한 선은 0–1과 2–3입니다.



추가한 선 0–1

추가한 선 2–3

코드가 고른 점 4개

노란 선 하나마다 서로 다른 새 점을 2개씩 골랐습니다.

나머지 선에서는 개수를 늘리지 않았습니다.

노란색은 꼭 확인할 선입니다. 흐린 선도 원래 그래프에 그대로 있습니다.

9-3 앞의 끝점을 공유하면 새 두 점을 고를 수 없습니다

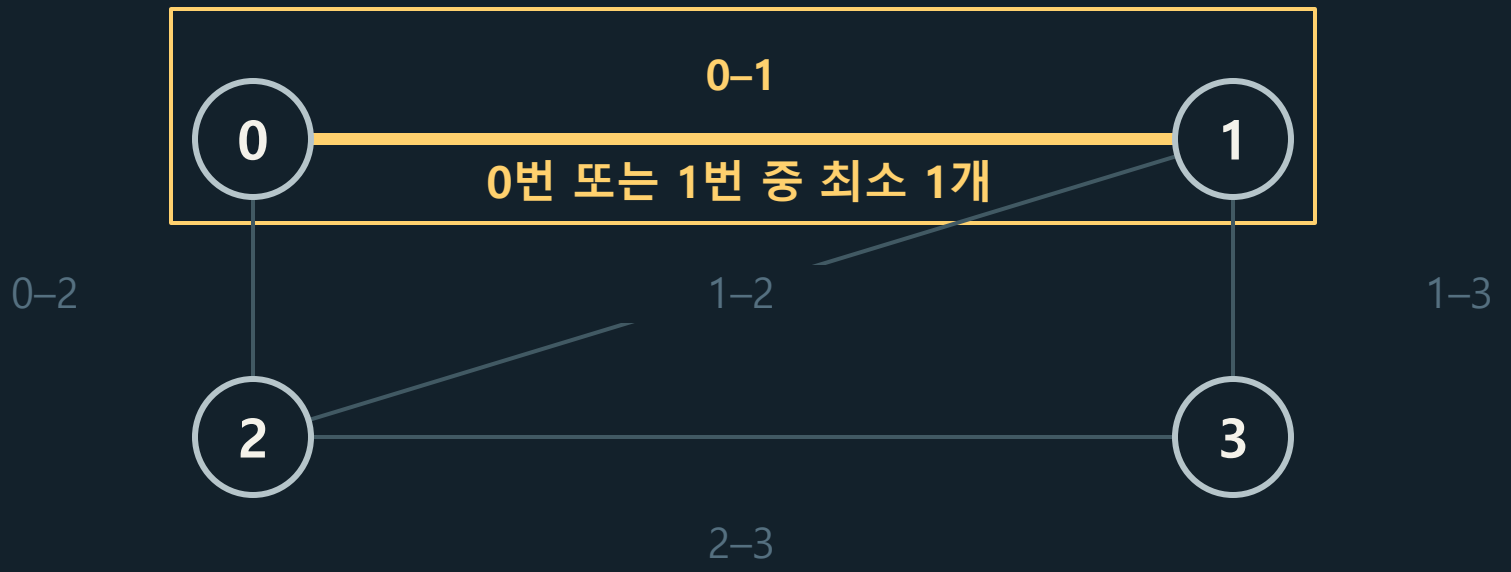
앞서 고른 점은 계속 True입니다. 같은 점을 다시 만나면 조건이 False입니다.

나중 선의 상태	if 조건	새 점 추가
u번을 이미 고름	not True and … → False	없음
v번을 이미 고름	… and not True → False	없음
두 점 모두 안 고름	not False and not False → True	두 점

그래서 두 점을 추가했던 선들끼리는 끝점이 하나도 겹치지 않습니다.

9-3 노란 선 0–1에는 어떤 정답도 한 점이 필요합니다

어떤 정답이든 0번 또는 1번을 적어도 하나 골라야 합니다.



0–1을 해결하려면

0번 또는 1번 필요

최소 1개

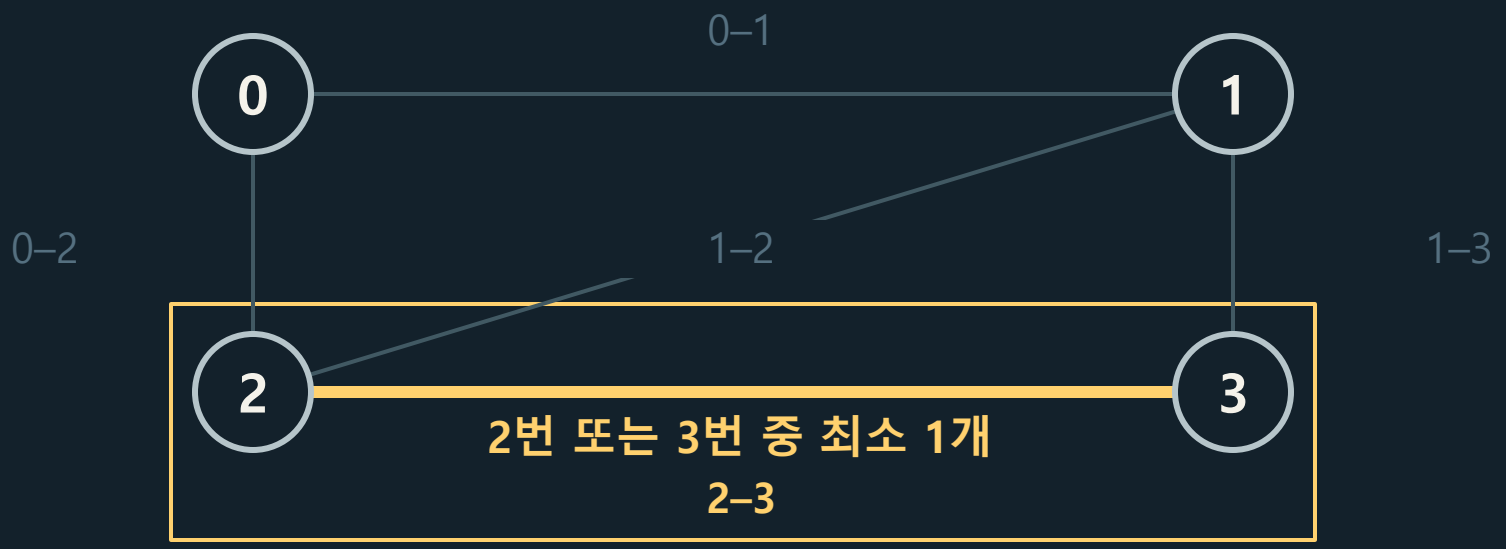
최적 선택이 어느 쪽을 골랐는지는 몰라도 됩니다.

0번과 1번 중 적어도 하나는 반드시 필요합니다.

노란색은 꼭 확인할 선입니다. 흐린 선도 원래 그래프에 그대로 있습니다.

9-3 노란 선 2–3에도 별도로 한 점이 필요합니다

아래쪽 2–3은 2번 또는 3번을 골라야 해결됩니다.



0–1에 최소 1개

2–3에 별도로 1개

합해서 최소 2개

0·1번과 2·3번에는 같은 점이 없습니다.

한 점을 두 노란 선의 끝점으로 함께 사용할 수 없습니다.

노란색은 꼭 확인할 선입니다. 흐린 선도 원래 그래프에 그대로 있습니다.

9-3 최적값을 몰라도 최소 2개라는 기준을 압니다

0–1에서 최소 1개, 2–3에서 별도의 최소 1개입니다.



노란 선 2개

서로 끝점이 다름

최적값은 2 이상

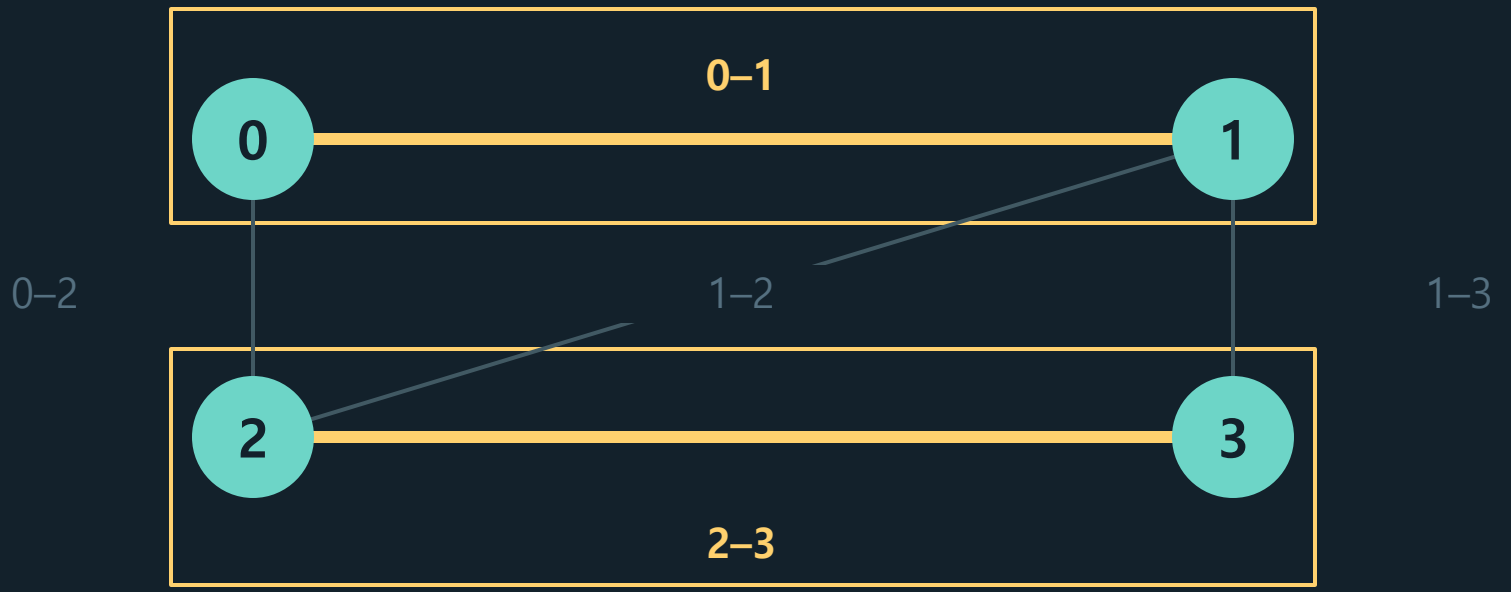
전체 선을 해결하는 정답은 두 노란 선도 모두 해결해야 합니다.

노란 선마다 한 점씩 필요하므로, 최적값은 적어도 2입니다.

노란색은 꼭 확인할 선입니다. 흐린 선도 원래 그래프에 그대로 있습니다.

9-3 코드의 4개가 최적의 2배 이내인 이유

코드가 고른 수는 4개이고, 최적의 수는 적어도 2개입니다.



코드 결과 = 4

최적값 ≥ 2

4 ≤ 2 × 최적값

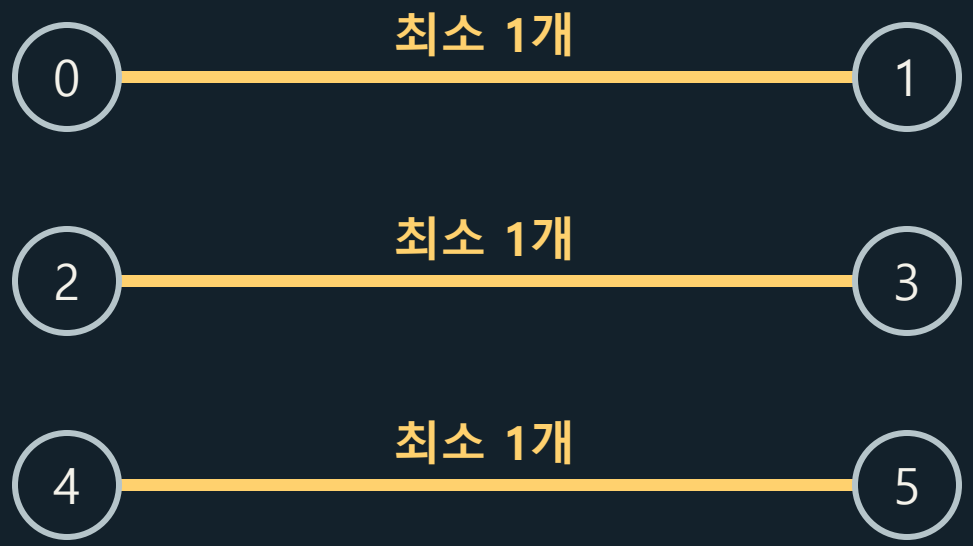
최적값이 2 이상이므로, 그 2배는 4 이상입니다.

코드가 쓴 4개는 최적값의 2배를 넘지 않습니다.

노란색은 꼭 확인할 선입니다. 흐린 선도 원래 그래프에 그대로 있습니다.

9-3 그런 선이 세 개이면 같은 계산을 합니다

큰 그래프에서 두 점을 추가한 선이 세 개였다고 해 보겠습니다.



선마다 최소 1개

최적값 ≥ 3

코드 결과 = 6

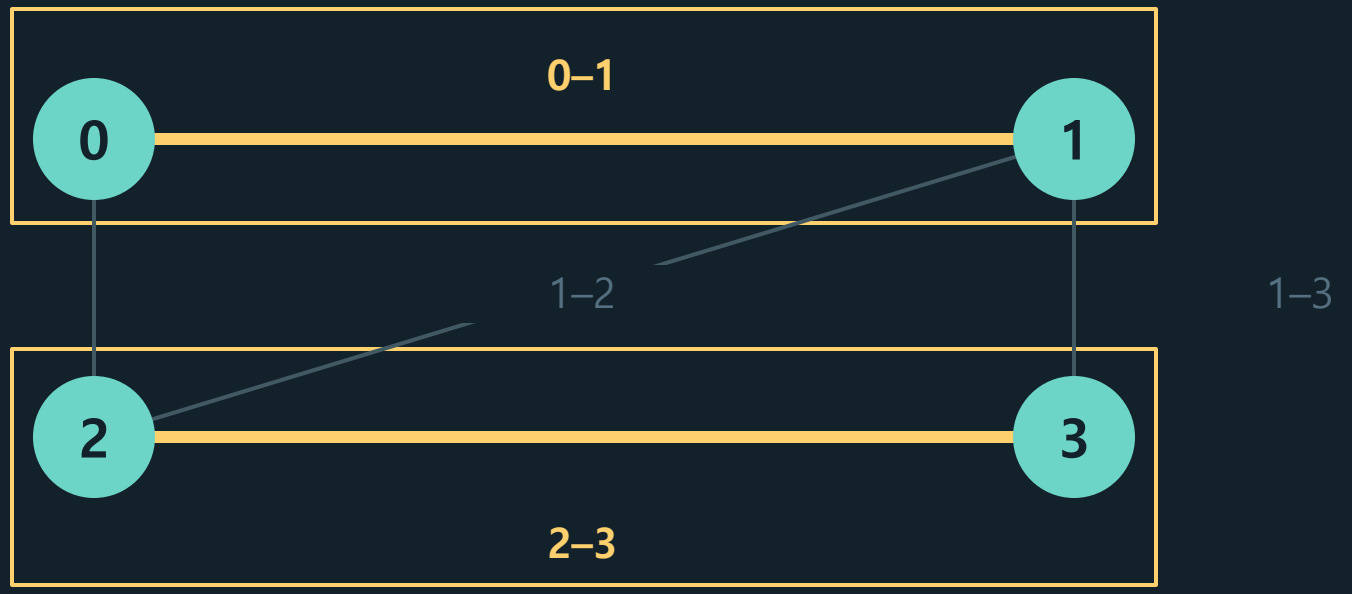
어떤 정답도 서로 다른 점을 적어도 3개 골라야 합니다.

코드는 6개를 골랐으므로 6 ≤ 2 × 최적값입니다.

각 노란 선의 끝점 번호가 서로 다릅니다.

9-3 선이 k개이면 코드 2k개, 최적은 k개 이상입니다

k는 두 점을 새로 추가했던 선의 수입니다. 그림은 k=2인 예입니다.



최적값 ≥ k

코드 결과 = 2k

2k ≤ 2 × 최적값

새로 두 점을 고른 선은 끝점이 겹치지 않아 선마다 한 점이 필요합니다.

우리 코드는 같은 선마다 두 점씩 쓰므로 최적의 2배 이내입니다.

노란색은 꼭 확인할 선입니다. 흐린 선도 원래 그래프에 그대로 있습니다.

9-3 같은 선과 비어 있는 입력도 규칙대로 처리합니다

기록을 유지하기 때문에 같은 점을 중복해서 세지 않습니다.

같은 선 두 번 [[0,1],[0,1]]은 처음에만 두 점을 골라 2를 반환합니다.

연결되지 않은 점 어느 선에도 나오지 않는 점은 고르지 않습니다.

선이 없는 경우 edges=[]이면 반복 없이 처음 값 0을 반환합니다.

선이 없으면 최적값도 0입니다. 결과 0도 2배 이내 조건을 만족합니다.

9-3 계산 시간은 점 수와 선 수에 비례합니다

n은 점 수이고, m은 선 수입니다.

선택 기록 준비 Python에서 n칸을 False로 만듭니다.

선 처리 m개 선을 한 번씩 읽고, 선마다 두 칸만 확인합니다.

사용하는 공간 점마다 상태를 기억할 chosen에 n칸이 필요합니다.

알고리즘의 시간복잡도는 O(n+m), 추가 공간은 O(n)입니다.

9-3 입력 순서와 최종 반환값을 확인합니다

공통 입력 n=4, 두 실행 모두 정해진 선택 규칙을 지킵니다.

입력에서 달라지는 부분	반환값	이유
0–1부터 읽는 다섯 선	4	0–1과 2–3에서 두 점씩
1–2부터 읽는 같은 다섯 선	2	1번과 2번으로 전체 해결
[[0,1],[0,1]]	2	같은 점을 다시 세지 않음
[]	0	고를 점이 필요 없음

모든 선을 해결하고, 고른 점 수는 최적의 2배 이내입니다.

세 문제에서 구현하는 함수와 원래 목표

기말 범위는 9-1, 9-2, 9-3입니다. 9-4는 추가 연습입니다.

문제	원래 최적화 목표	지정한 함수의 예제 답
9-1 두 기계	전체 완료 시간을 최소화	[8,7,6,5,4] → 17
9-2 상자 채우기	제한 안에서 무게 합 최대화	[6,5,5], 10 → 6
9-3 정점 커버	모든 선을 덮는 점 수 최소화	네 점·다섯 선의 예제 → 4

예제의 최적값은 각각 15, 10, 2입니다. 규칙의 반환값과 따로 기억합니다.

9-4 추가 연습 · 모든 구역을 적은 팀으로 맡기기

이 문제는 기말고사 범위에 포함하지 않습니다.

구역과 팀 A는 {0,1}, B는 {0,2}, C는 {1,3} 구역을 맡을 수 있습니다.

원래 목표 0,1,2,3 모든 구역을 맡기면서 팀 수를 최소화합니다.

먼저 해 보기 두 팀만 고른다면 어느 팀을 골라야 할까요?

B와 C를 함께 쓰면 모든 구역을 맡길 수 있습니다.

9-4 아직 맡기지 않은 구역을 가장 많이 덮는 팀

아직 담당 팀이 없는 구역을 가장 많이 맡길 수 있는 팀을 하나씩 고릅니다.

처음 상태	네 구역 모두 아직 맡기지 않았습니다.
한 번의 선택	각 팀이 이번에 새로 맡을 수 있는 구역만 셉니다. 수가 같으면 입력에서 먼저 나온 팀을 고릅니다.
반환할 값	모두 맡기면 선택한 팀 수입니다. 남은 곳을 어느 팀도 맡을 수 없으면 −1입니다.

선택할 때마다 남은 구역을 기준으로 세는 것이 핵심입니다.

9-4 처음에는 세 팀이 모두 두 곳씩 맡습니다

팀	구역 0	구역 1	구역 2	구역 3	새로 맡는 수
A	●	●	–	–	2
B	●	–	●	–	2
C	–	●	–	●	2

선택을 마친 뒤

0번 A	1번 A	2번 남음	3번 남음

0번과 1번을 맡겼습니다. 남은 구역은 {2, 3}입니다.

첫째 선택은 A팀

선택 전에는 0, 1, 2, 3번이 모두 남았습니다.

세 팀이 2곳으로 같아 입력의 첫 팀 A를 고릅니다.

9-4 두 번째에는 B와 C가 한 곳씩 맡을 수 있습니다

팀	구역 0	구역 1	구역 2	구역 3	새로 맡는 수
A	●	●	–	–	0
B	●	–	●	–	1
C	–	●	–	●	1

선택을 마친 뒤

0번 A	1번 A	2번 B	3번 남음

B가 2번을 새로 맡았습니다. 이제 3번만 남았습니다.

둘째 선택은 B팀

B의 0번은 이미 맡겼고, 2번만 새로 맡습니다.

C도 3번 한 곳입니다. 더 앞에 나온 B를 고릅니다.

9-4 마지막 남은 3번을 C에게 맡깁니다

팀	구역 0	구역 1	구역 2	구역 3	새로 맡는 수
A	●	●	–	–	0
B	●	–	●	–	0
C	–	●	–	●	1

선택을 마친 뒤

0번 A	1번 A	2번 B	3번 C

셋째 선택은 C팀

3번에 갈 수 있는 팀은 C뿐입니다.

네 구역에 모두 담당 팀이 생겼으므로 여기서 끝납니다.

A, B, C 세 팀을 골랐으므로 반환값은 3입니다.

9-4 처음 B를 고르면 두 팀으로 끝납니다

입력이 A, B, C일 때

3팀

A가 0·1번을 맡습니다.

B는 2번, C는 3번을 이어서 맡습니다.

입력이 B, C, A일 때

2팀

B가 0·2번을 맡습니다.

남은 1·3번을 C가 한꺼번에 맡습니다.

한 팀이 모두 맡을 수는 없으므로 최적은 2팀. 세 팀을 골랐을 때는 최적의 1.5배입니다.

9-4 팀이 맡을 수 있는 칸에 1을 씁니다

구역 번호	3	2	1	0
A가 맡는 구역	–	–	●	●
1과 0으로 기록	0	0	1	1
각 자리가 뜻하는 수	8	4	2	1

구역 k에 대응하는 자리가 1이면 그 구역을 포함합니다.

9-4 3과 5는 서로 다른 두 구역을 뜻합니다

팀과 담당 구역	3번 · 8	2번 · 4	1번 · 2	0번 · 1	들어오는 수
A {0, 1}	0	0	1	1	2 + 1 = 3
B {0, 2}	0	1	0	1	4 + 1 = 5
C {1, 3}	1	0	1	0	8 + 2 = 10
빈 집합	0	0	0	0	0

정수값은 집합을 기록한 값입니다. 5 안에 들어 있는 구역은 두 곳입니다.

9-4 입력과 반환값 예제

`greedy_set_cover(n, sets)`

n	4. 전체 구역의 번호는 0, 1, 2, 3입니다.
sets	[3, 5, 10]. 각 정수는 한 팀이 맡을 수 있는 구역을 담습니다.
반환값	세 팀을 차례로 선택해 3. 다 맡길 수 없으면 −1입니다.

같은 팀을 [5, 10, 3]으로 받으면 이 규칙은 2를 반환합니다.

n 1~20 · 팀 0~30개 · 각 마스크 $0 \sim 2^n - 1$

9-4 처음에는 남은 구역의 네 칸이 모두 1입니다

계산	오른쪽을 기준으로 본 비트	정수값
1	00001	1
1 << 4	10000	16
(1 << 4) − 1	01111	15

처음 remaining = 15는 구역 0, 1, 2, 3이 전부 남았다는 표시입니다.

9-4 네 칸의 뜻을 그림 아래에 그대로 기록합니다

구역 번호는 왼쪽부터 3, 2, 1, 0입니다.

	구역 3	구역 2	구역 1	구역 0	
아직 팀이 없는 구역	1	1	0	0	= 12
B가 갈 수 있는 구역	0	1	0	1	= 5
두 줄 모두 1인 구역	0	1	0	0	= 4

같은 자리의 두 표시가 모두 1인 칸만 남기는 연산이 &입니다.

9-4 이번에 필요한 수는 '1이 몇 개 있는가'입니다

4와 8은 켜져 있는 위치가 다릅니다. 켜진 칸의 수는 같습니다.

	구역 3	구역 2	구역 1	구역 0		
B의 새 구역 {2}	0	1	0	0	= 4	1곳
C의 새 구역 {3}	1	0	0	0	= 8	1곳

bit_count()는 1인 칸의 개수를 세어 두 팀을 1 대 1로 비교하게 합니다.

9-4 맡길 팀을 고른 뒤 해당 칸을 0으로 바꿉니다

새로 맡긴 2번은 0이 되고, 아직 남은 3번은 1로 둡니다.

	구역 3	구역 2	구역 1	구역 0	
B를 고르기 전	1	1	0	0	= 12
B가 맡길 수 있는 곳	0	1	0	1	= 5
B를 고른 뒤	1	0	0	0	= 8

remaining에는 아직 맡기지 않은 곳만 1로 남깁니다.

9-4 겹치는 칸에만 1을 남겨 계산합니다

네 칸의 의미	3번	2번	1번	0번	정수값
남은 구역 {2, 3}	1	1	0	0	12
B의 구역 {0, 2}	0	1	0	1	5
12 & 5	0	1	0	0	4

결과 마스크는 4. 새로 맡는 구역 수는 1인 칸의 개수, 즉 1입니다.

9-4 선택한 팀의 구역을 남은 목록에서 지웁니다

구역 번호	3번	2번	1번	0번	정수값
선택 전 remaining	1	1	0	0	12
선택한 B의 마스크	0	1	0	1	5
~best의 오른쪽 네 칸	1	**0**	1	0	–
선택 후 remaining	**1**	0	0	0	**8**

선택 전 remaining & ~best를 계산하면 3번만 남습니다.

9-4 남은 구역을 맡을 팀이 없으면 −1입니다

입력	실제로 선택한 마스크	반환값
n = 4, sets = [3, 6, 12]	3 → 12	2
n = 4, sets = [3]	3 다음에 2·3번이 남음	−1
n = 4, sets = []	첫 선택부터 없음	−1
n = 1, sets = [0, 1]	0을 지나쳐 1 선택	1

한 곳이라도 남았는데 모든 팀의 추가 수가 0이면 멈춥니다.

9-4 Python 1~5행 · 이번 선택의 기록을 초기화

```
01  def greedy_set_cover(n, sets):
02      remaining = (1 << n) - 1
03      count = 0
04      while remaining:
05          best, gain = 0, 0
```

변수	시작할 때
remaining	15. 네 구역이 모두 남았습니다.
count	0. 아직 고른 팀이 없습니다.
best, gain	0, 0. 이번 비교의 후보와 점수입니다.

while은 맡길 곳이 남은 동안

remaining = 0이면 반복을 끝냅니다.

반복에 들어갈 때마다 best와 gain을 다시 0으로 둡니다.

count는 이전 선택을 기억하고, best와 gain은 매번 새로 계산합니다.

9-4 Python 6~9행 · 새 구역을 많이 맡는 팀 찾기

```
06            for mask in sets:
07                added = (mask & remaining).bit_count()
08                if added > gain:
09                    best, gain = mask, added
```

A를 고른 직후	added	비교 뒤 best
A: mask = 3	0	0
B: mask = 5	1	5
C: mask = 10	1	5 그대로

남은 곳은 {2, 3}

&로 겹치는 칸을 남깁니다.

bit_count()로 1의 개수를 셉니다.

added가 더 클 때만 후보를 바꿉니다.

2와 3만 남았을 때 B와 C가 1로 같으면 B를 유지합니다.

9-4 Python 10~14행 · 진행이 되면 남은 곳 갱신

```
10            if gain == 0:
11                return -1
12            remaining &= ~best
13            count += 1
14        return count
```

선택을 마칠 때	gain	remaining	count
A를 고른 뒤	2	15 → 12	0 → 1
B를 고른 뒤	1	12 → 8	1 → 2
C를 고른 뒤	1	8 → 0	2 → 3

gain이 0인 경우에는 −1

남은 곳을 어떤 팀도 맡지 못합니다.

gain이 양수이면 고른 팀의 칸을 지우고 count를 한 번 늘립니다.

remaining이 0이 되면 while 밖에서 count를 반환합니다.

9-4 Python 전체 풀이

```python
def greedy_set_cover(n, sets):
    remaining = (1 << n) - 1
    count = 0
    while remaining:
        best, gain = 0, 0
        for mask in sets:
            added = (mask & remaining).bit_count()
            if added > gain:
                best, gain = mask, added
        if gain == 0:
            return -1
        remaining &= ~best
        count += 1
    return count
```

남은 구역을 숫자로 따라가기

15 → 12 → 8 → 0

추가 수가 0이면 즉시 −1

0까지 줄였다면 선택 횟수 3

남은 구역을 한 곳 이상 줄이는 반복만 다음으로 넘어갑니다.

9-4 C에서는 오른쪽 끝 1을 하나씩 읽습니다

지금 x	오른쪽 자리 x & 1	지금까지의 개수	오른쪽으로 한 칸 옮긴 x
101 (5)	1	1	10 (2)
10 (2)	0	1	1 (1)
1 (1)	1	2	0

x가 0이 되면 멈춥니다. 5에는 1이 두 자리 있습니다.

9-4 C 보조 함수 count_bits

```c
static int count_bits(int x) {
    int count = 0;
    while (x) {
        count += x & 1;
        x >>= 1;
    }
    return count;
}
```

count_bits(5)의 결과는 2

첫 반복 x = 5, count = 1

둘째 반복 x = 2, count = 1

셋째 반복 x = 1, count = 2

x & 1은 맨 끝자리, x >>= 1은 다음 자리를 읽기 위한 이동입니다.

9-4 C 1~6행 · 팀 수 m을 받아 배열을 읽습니다

```
01  int greedy_set_cover(int n, const int sets[], int m) {
02      int remaining = (1 << n) - 1;
03      int count = 0;
04      while (remaining) {
05          int best = 0, gain = 0;
06          for (int i = 0; i < m; ++i) {
```

C에서 받은 값	예제에서의 값
n	4개의 구역
sets	{3, 5, 10}
m	3개의 팀
처음 remaining	15

for 문은 m번 실행

i가 0, 1, 2일 때 팀을 하나씩 확인합니다.

n은 구역 수이고 m은 팀 수입니다.

n은 불을 켤 비트 수, m은 확인할 팀의 수입니다.

9-4 C 7~12행 · 겹치는 곳을 센 뒤 후보를 갱신

```
07                int added = count_bits(sets[i] & remaining);
08                if (added > gain) {
09                    best = sets[i];
10                    gain = added;
11                }
12            }
```

2·3번이 남았을 때	교집합 마스크	added
A, sets[i] = 3	0	0
B, sets[i] = 5	4	1
C, sets[i] = 10	8	1

count_bits에 마스크를 건넵니다

겹친 마스크 4와 8은 다른 값입니다.

각각 1이 한 자리라 added는 같습니다.

added > gain일 때만 best를 바꾸므로 B가 남습니다.

9-4 C 13~18행 · 남은 곳을 지우고 횟수를 반환

```
13            if (gain == 0) return -1;
14            remaining &= ~best;
15            ++count;
16        }
17        return count;
18    }
```

상황	실행할 다음 단계
gain = 0	return -1
gain = 1, best = 5	remaining: 12 → 8
remaining = 0	반복을 나와 count 반환

~best로 선택한 위치를 지웁니다

선택이 이뤄졌을 때만 ++count를 실행합니다.

선택한 팀 수는 while 밖에서 돌려줍니다.

보조 함수와 주 함수의 반환값이 다릅니다. 하나는 칸 수, 다른 하나는 팀 수입니다.

9-4 C 전체 풀이 (1/2)

```
01  int greedy_set_cover(int n, const int sets[], int m) {
02      int remaining = (1 << n) - 1;
03      int count = 0;
04      while (remaining) {
05          int best = 0, gain = 0;
06          for (int i = 0; i < m; ++i) {
07              int added = count_bits(sets[i] & remaining);
08              if (added > gain) {
09                  best = sets[i];
10                  gain = added;
11              }
12          }
```

선택 후보를 기록하는 부분

아직 맡기지 않은 곳을 1로 표시

count_bits로 새로 맡는 수를 계산

동점은 입력에서 앞선 팀이 유지

앞 장의 count_bits를 이 함수 앞에 둡니다. 13행부터는 다음 장에서 이어집니다.

9-4 C 전체 풀이 (2/2)

```
13            if (gain == 0) return -1;
14            remaining &= ~best;
15            ++count;
16        }
17        return count;
18    }
```

선택을 확정하는 부분

gain이 0이면 −1

맡긴 곳을 지우고 count 증가

remaining이 0이면 count 반환

n은 최대 20이라 수업의 32비트 int에서 마스크를 다룰 수 있습니다.