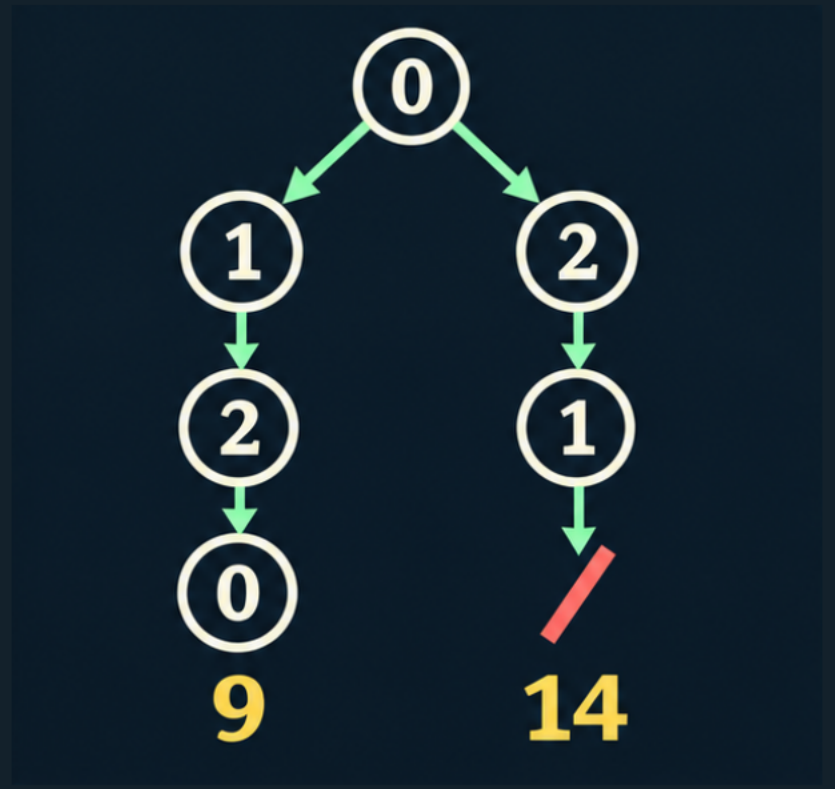
08

분기한정 기법

더 싼 답을 찾고, 비싼 갈래는 일찍 멈추기



8장 기말고사 범위

8-1, 8-2, 8-3에서 출제합니다.

8-1 주어진 순서의 왕복 비용

8-2 남은 비용의 하한

8-3 서로 다른 작업의 최소 배정 비용

8-4 추가 연습이며 기말고사에 출제하지 않습니다.

코드 실행 결과로 평가합니다.

끝까지 계산하기 전에 멈출 수 있습니다



이 선택 아래의 계산을 생략합니다.

이미 찾은 답의 비용은 9

다른 선택은 아무리 싸게 잡아도 16

16보다 더해질 비용이 있어도 9보다 싸질 수 없습니다.

한 번의 선택이 여러 갈래를 만듭니다

분기 = 다음 선택을 하나씩 나누어 보기

사람 0에게 작업 0을 맡기는 경우

사람 0에게 작업 1을 맡기는 경우

사람 0에게 작업 2를 맡기는 경우

각 선택 뒤에는 남은 사람의 선택이 이어집니다.

두 사람에게 서로 다른 작업을 맡겨 봅시다

사람 / 작업	작업 0	작업 1
사람 0	4	1
사람 1	2	3

목표는 두 비용의 합을 가장 작게 하는 것입니다.

숫자는 그 작업을 맡길 때 드는 비용

사람마다 작업 하나를 맡습니다.

작업 하나를 두 사람이 함께 맡을 수 없습니다.

두 배정의 비용을 직접 비교합니다

첫 번째 배정

사람 0에게 작업 0

사람 1에게 작업 1

4 + 3 = 7

두 번째 배정

사람 0에게 작업 1

사람 1에게 작업 0

1 + 2 = 3

두 가지를 비교하면 최소 비용은 3입니다.

이 장에서 직접 작성할 네 함수

같은 비용 표도 질문에 따라 쓰는 방법이 달라집니다

8-1 주어진 순서대로 이동한 비용 더하기

8-2 남은 작업을 아주 싸게 잡은 비용 계산하기

8-3 서로 다른 작업을 맡기는 최소 비용 찾기

8-4 방문 순서까지 골라 최소 왕복 비용 찾기

기말고사 범위는 8-1~8-3입니다. 8-4는 추가 연습입니다.

8-1 주어진 방문 순서의 왕복 비용

정해진 순서대로 도시를 돌았을 때 드는 총비용을 구합니다.

주어진 것 도시 사이의 이동 비용 표와 방문할 도시의 순서가 주어집니다.

이동 순서 0번 도시에서 출발해 주어진 순서대로 모든 도시를 한 번씩 방문합니다.

마지막 이동 마지막 도시에서 출발지인 0번 도시로 돌아오는 비용도 더합니다.

구할 답은 이 왕복 경로에 드는 비용의 합입니다.

8-1 입력과 반환값 예제

`route_cost(dist, route)`

비용 표 dist

[[0, 2, 8], [1, 0, 3], [4, 6, 0]]
dist[i][j]는 도시 i에서 도시 j로 가는 비용입니다.

방문 순서 route

[0, 1, 2]
배열에 적힌 순서대로 방문하며, 맨 앞은 항상 0입니다.

반환값

0번에서 1번까지 2, 1번에서 2번까지 3,
2번에서 0번까지 4가 들어서 9를 반환합니다.

2 + 3 + 4 = 9

도시 1~9개 · 비용 0~100 · 같은 도시로 가는 비용 0 · 도시 1개이면 반환값 0

dist의 행은 출발지, 열은 도착지입니다

출발 / 도착	도시 0	도시 1	도시 2
도시 0	0	2	8
도시 1	1	0	3
도시 2	4	6	0

dist[0][1] = 2

0행에서 1열을 읽습니다.

반대로 dist[1][0]은 1입니다.

반대 방향의 비용도 표에서 따로 읽습니다.

8-1 이동할 때마다 합이 커집니다



세 이동의 비용을 빠짐없이 한 번씩 더합니다.

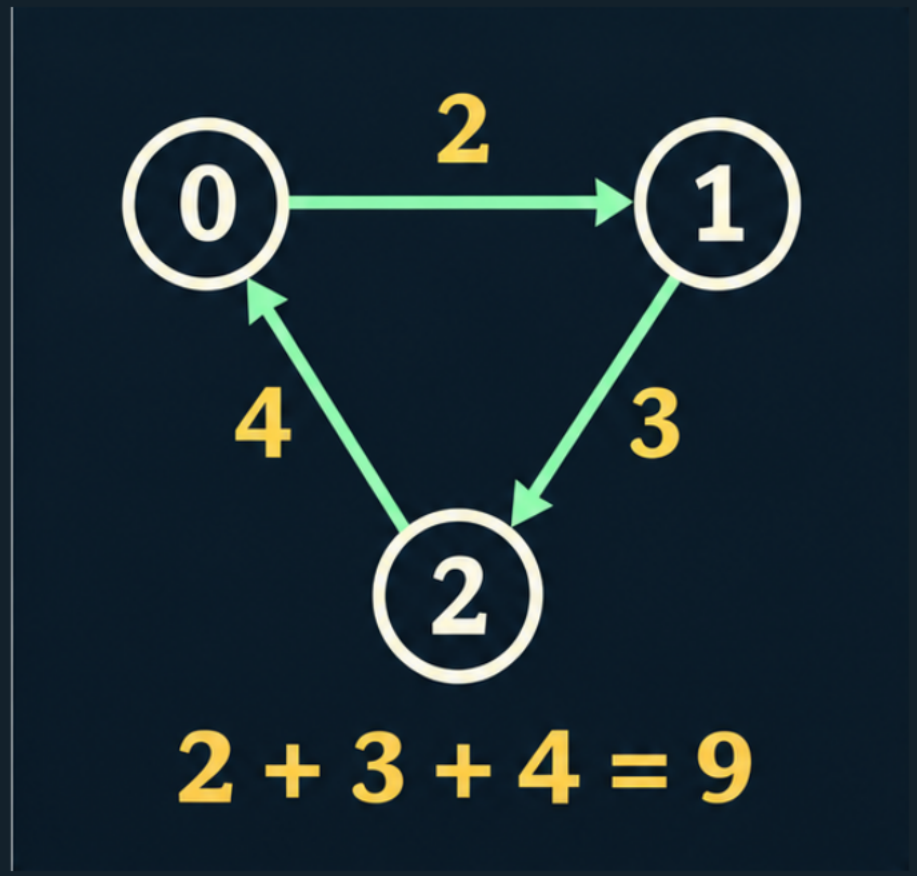
마지막 귀환도 한 번의 이동입니다.

0에서 1까지 누적 2

1에서 2까지 누적 5

2에서 0으로 돌아오면 9

주어진 순서의 세 이동을 더합니다



route = [0, 1, 2]

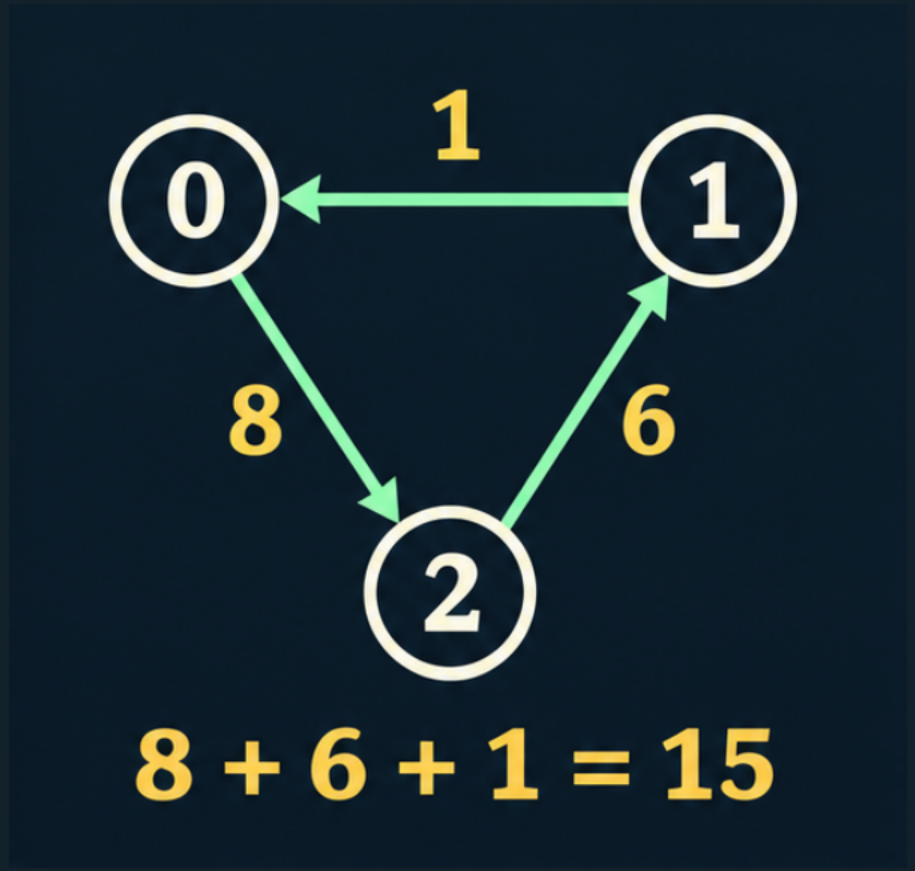
0에서 1로 가는 비용 2

1에서 2로 가는 비용 3

2에서 0으로 돌아오는 비용 4

2 + 3 + 4 = 9

순서가 달라지면 읽는 비용도 달라집니다



route = [0, 2, 1]

이번에는 0에서 2로 갑니다.

그다음 2에서 1, 마지막 1에서 0입니다.

8 + 6 + 1 = 15

배열 위치와 도시 번호를 구분합니다

i	route[i]	다음 도시	더할 비용
0	0	1	2
1	1	2	3
2	2	0	4

위치 2 다음의 위치는 0입니다.

i는 배열에서 읽을 위치입니다

i가 0일 때 도시 0을 읽습니다.

마지막 위치 다음에는 첫 도시로 돌아갑니다.

나머지 연산으로 첫 위치로 돌아옵니다

```
(i + 1) % len(route)
```

route의 길이가 3일 때

i=0이면 (0+1)%3은 1

i=1이면 (1+1)%3은 2

i=2이면 (2+1)%3은 0

%는 나눗셈의 나머지입니다.

한 줄에 들어 있는 배열 읽기를 나누어 봅니다

```
dist[route[i]][route[(i+1)%len(route)]]
```

i=1, route=[0,1,2]

route[1]은 1입니다.

route[2]는 2입니다.

따라서 dist[1][2]의 값 3을 읽습니다.

이번 이동 비용 3을 total에 더합니다.

8-1 Python 1행 · 입력 두 개 받기

```
01  def route_cost(dist, route):
```

변수·상태	예제에서의 값
dist	3×3 비용 표
route	[0, 1, 2]

입력 두 개 받기

route_cost는 주어진 방문 순서의 비용을 계산하는 함수 이름입니다. 괄호 안 dist에는 이동 비용 표, route에는 방문 순서 배열을 받습니다. def 줄은 계산 방법을 정의하고, 실제 계산은 함수가 호출될 때 시작합니다.

입력 두 개 받기

8-1 Python 2행 · 비용 합을 0에서 시작하기

```
02        total = 0
```

변수·상태	예제에서의 값
total	0
처리한 이동	아직 없음

비용 합을 0에서 시작하기

total은 지금까지 이동한 비용의 합입니다. 아직 어떤 이동도 계산하지 않았으므로 0에서 시작합니다. 더하기를 누적하려면 이전 값이 남아 있지 않도록 호출할 때마다 새로 0으로 만듭니다.

비용 합을 0에서 시작하기

8-1 Python 3행 · 방문 순서의 각 칸 읽기

```
03      for i in range(len(route)):
```

변수·상태	예제에서의 값
len(route)	3
i의 순서	0, 1, 2

방문 순서의 각 칸 읽기

len(route)는 방문 순서에 적힌 도시의 수입니다. 길이가 3이면 range는 0, 1, 2를 차례로 만듭니다. i는 도시 번호가 아니라 route에서 읽을 칸 번호입니다. 세 칸을 모두 읽어 세 번의 이동 비용을 더합니다.

방문 순서의 각 칸 읽기

8-1 Python 4행 · 현재 도시와 다음 도시 연결하기

```
04          total += dist[route[i]][route[(i+1)%len(route)]]
```

변수·상태	예제에서의 값
i	2
현재 / 다음 도시	2 / 0
더할 비용	dist[2][0] = 4

현재 도시와 다음 도시 연결하기

이 줄을 안쪽부터 읽겠습니다. route[i]는 지금 도시입니다. i에 1을 더하면 다음 칸이고, 배열 길이로 나눈 나머지를 쓰면 마지막 칸 다음이 0번 칸으로 돌아갑니다. 그 다음 칸의 도시를 route에서 읽고, dist의 출발 행과 도착 열을 찾아 비용을 더합니다. total 더하기 등호는 total에 이번 비용을 더한 값을 다시 total에 저장합니다.

현재 도시와 다음 도시 연결하기

8-1 Python 5행 · 합을 정답으로 반환하기

```
05        return total
```

변수·상태	예제에서의 값
더한 비용	2 + 3 + 4
반환값	9

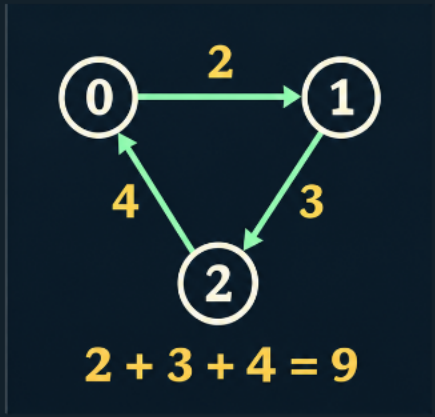
합을 정답으로 반환하기

세 이동을 모두 계산한 뒤 total을 호출한 곳에 돌려줍니다. 출력문으로 화면에 찍는 일과 반환은 다릅니다. 사이트가 받으려는 값은 함수가 반환하는 정수입니다.
예제에서는 2, 3, 4를 더한 9를 반환합니다.

합을 정답으로 반환하기

8-1 Python 전체 풀이

```python
def route_cost(dist, route):
    total = 0
    for i in range(len(route)):
        total += dist[route[i]][route[(i+1)%len(route)]]
    return total
```



route = [0, 1, 2]

2 + 3 + 4 = 9

주어진 경로의 왕복 비용을 정수 하나로 반환한다.

C에서는 배열의 길이 n도 함께 받습니다

dist[][9]의 9는 한 행의 저장 공간입니다

실제 도시 수는 n입니다. n은 1~9입니다.

route의 앞 n개 칸을 읽습니다.

int는 정수, const는 입력을 읽어서 사용한다는 표시입니다.

사이트가 전달한 배열과 n으로 계산합니다.

8-1 C 1행 · 배열과 실제 길이를 함께 받기

```
01  int route_cost(const int dist[][9], int n, const int route[]) {
```

변수·상태	예제에서의 값
n	3
route 앞 세 칸	[0, 1, 2]

배열과 실제 길이를 함께 받기

맨 앞 int는 정수 값을 반환한다는 표시입니다. dist는 이동 비용 표이고 n은 실제 도시 수입니다. C의 이 배열 인자에서는 각 행의 저장 폭 9를 함께 적습니다. route는 방문 순서 배열입니다. const는 입력 배열을 이 함수 안에서 고치지 않고 읽는다는 표시입니다.

배열과 실제 길이를 함께 받기

8-1 C 2~3행 · 합을 준비하고 각 칸 읽기

```
02      int total=0;
03      for (int i=0; i<n; ++i)
```

변수·상태	예제에서의 값
total 시작값	0
i	0, 1, 2

합을 준비하고 각 칸 읽기

정수 변수 total을 만들고 0을 넣습니다. Python의 total과 같은 누적 합입니다. 정수 변수는 초기값을 명시해야 이전 메모리의 내용으로 계산하는 일을 피할 수 있습니다.

합을 준비하고 각 칸 읽기

8-1 C 4행 · 마지막 다음 칸을 0으로 돌리기

```
04          total+=dist[route[i]][route[(i+1)%n]];
```

변수·상태	예제에서의 값
i = 2일 때	(2+1)%3 = 0
마지막 비용	dist[2][0] = 4

마지막 다음 칸을 0으로 돌리기

현재 도시 route[i]와 다음 도시 route[(i+1)%n]을 찾아 그 이동 비용을 total에 더합니다. 퍼센트는 나머지 연산입니다. i가 마지막인 2이면 3을 3으로 나눈 나머지가 0이라 첫 도시로 돌아옵니다. 세미콜론은 이 C 문장을 끝낸다는 표시입니다.

마지막 다음 칸을 0으로 돌리기

8-1 C 5~6행 · 계산한 합을 반환하기

```
05        return total;
06    }
```

변수·상태	예제에서의 값
total	9
함수 결과	9

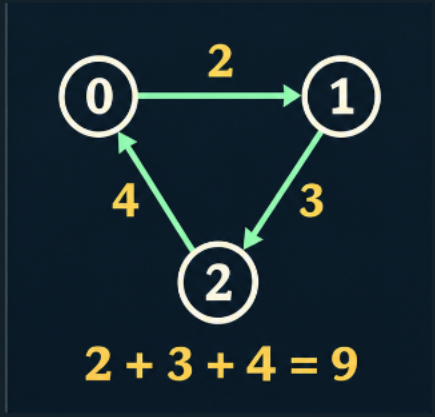
계산한 합을 반환하기

모든 이동을 더한 total을 정수로 반환합니다. 예제에서는 9이며, 사이트는 이 반환값을 예상 값과 비교합니다.

계산한 합을 반환하기

8-1 C 전체 풀이

```
01  int route_cost(const int dist[][9], int n, const int route[]) {
02      int total=0;
03      for (int i=0; i<n; ++i)
04          total+=dist[route[i]][route[(i+1)%n]];
05      return total;
06  }
```



route = [0, 1, 2]

2 + 3 + 4 = 9

주어진 경로의 왕복 비용을 정수 하나로 반환한다.

8-2 남은 비용의 하한 구하기

실제로 필요한 남은 비용보다 커지지 않는 금액을 빠르게 구합니다.

작업 배정 사람마다 작업 하나를 맡습니다. 같은 작업을 두 사람에게 맡길 수 없습니다.

현재 상태 앞의 몇 사람은 배정을 마쳤습니다. 남은 사람들의 비용을 봅니다.

이번에 구할 값 "실제로 어떻게 배정해도 적어도 이만큼은 든다"는 금액을 구합니다.

이 금액을 남은 비용의 하한이라고 부릅니다.

① 비용 표의 한 칸은 무엇을 뜻할까요?

사람 / 작업	작업 0	작업 1	작업 2
사람 0	9	2	7
사람 1	6	4	3
사람 2	5	8	1

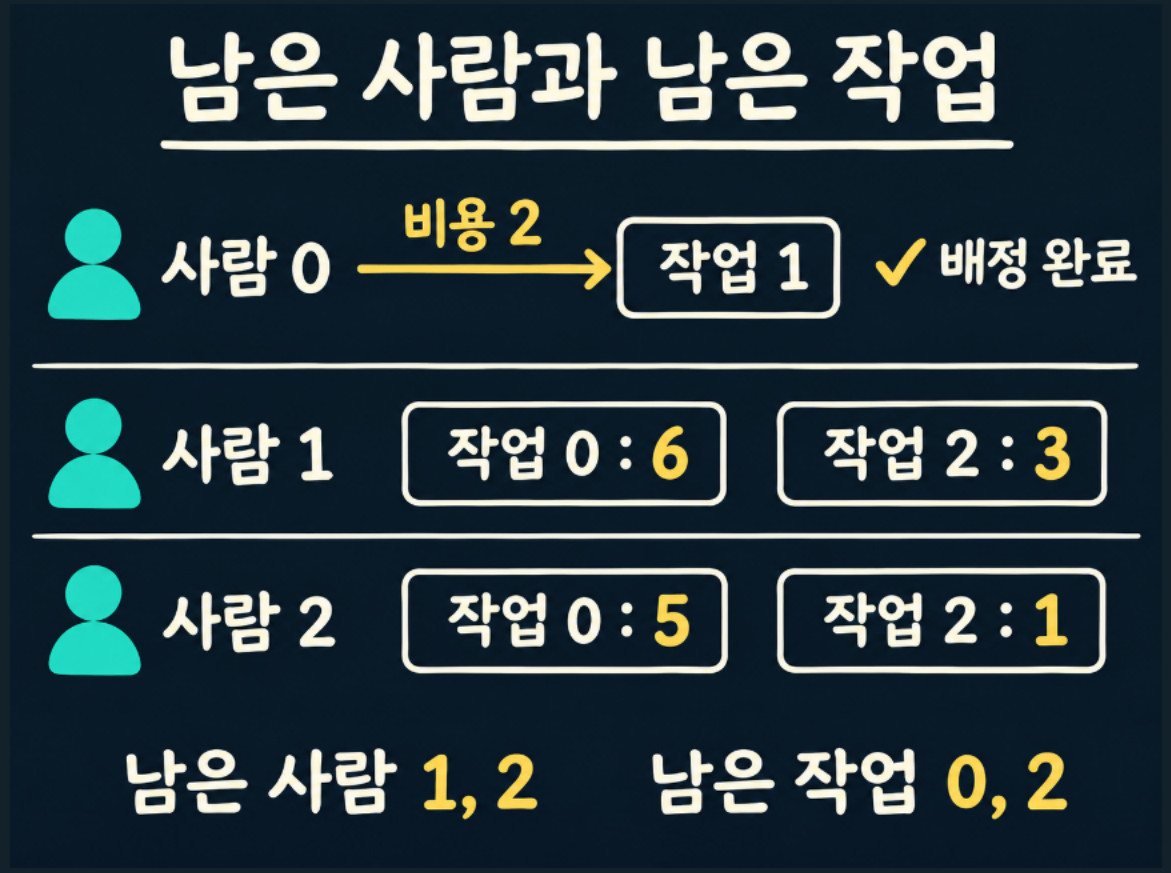
한 행은 한 사람, 한 열은 한 작업입니다.

사람마다 같은 작업에 드는 비용이 다릅니다.

사람 1에게 작업 0을 맡기면 6입니다.

사람 1에게 작업 2를 맡기면 3입니다.

② 사람 0의 배정은 이미 끝났습니다



사람 0에게 작업 1을 맡겼습니다.

남은 사람은 1과 2입니다.

남은 작업은 0과 2입니다.

이미 쓴 비용 2는 따로 기억합니다.

이제 두 사람에게 남은 두 작업을 맡겨야 합니다.

③ 사람 1의 가장 싼 비용은 3입니다

사람 / 작업	작업 0	작업 2
사람 1	6	3
사람 2	5	1

사람 1의 최소 후보 비용 = min(6, 3) = 3

사람 1의 후보 비용은 6과 3입니다.

두 값 중 작은 3을 기억합니다.

사람 1에게는 적어도 3이 듭니다.

④ 사람 2의 가장 싼 비용은 1입니다

사람 / 작업	작업 0	작업 2
사람 1	6	3
사람 2	5	1

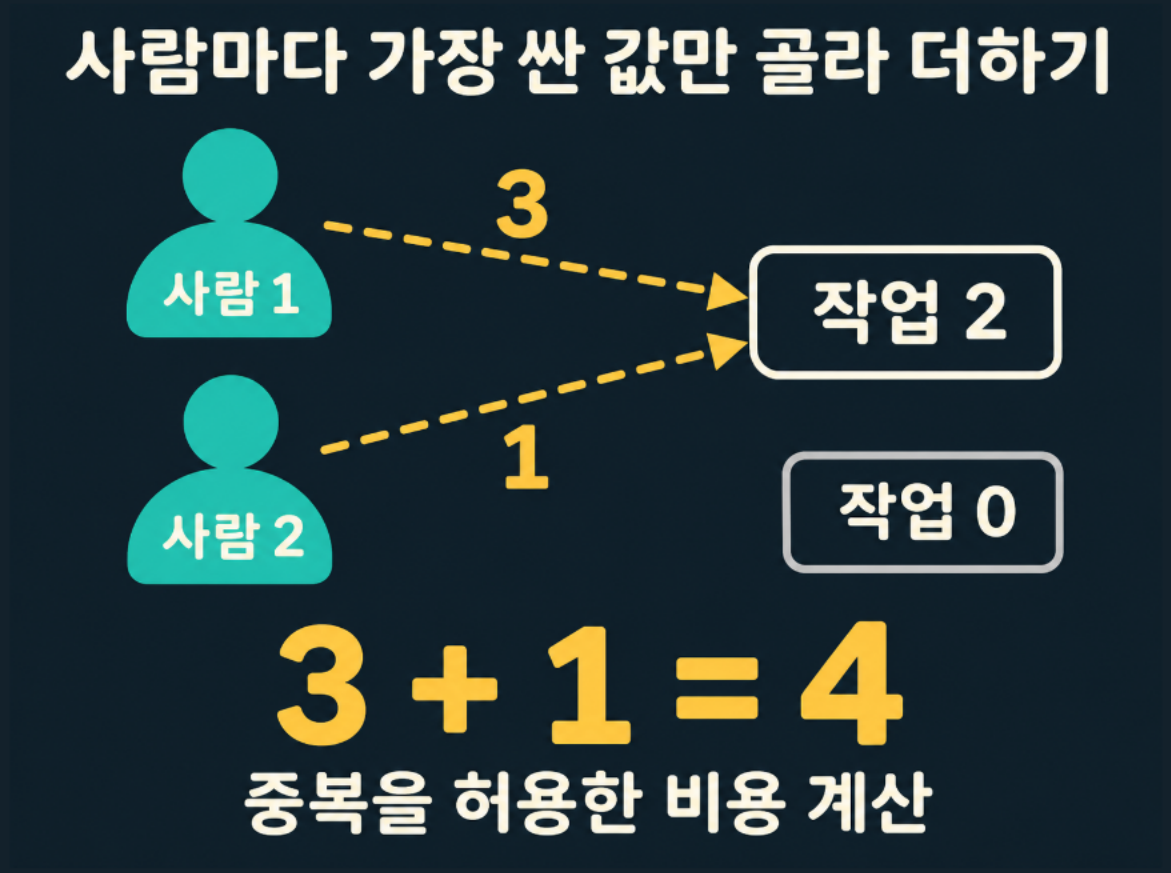
사람 2의 최소 후보 비용 = min(5, 1) = 1

사람 2의 후보 비용은 5와 1입니다.

두 값 중 작은 1을 기억합니다.

사람 2에게는 적어도 1이 듭니다.

⑤ 가장 싼 값만 더하면 4입니다



사람 1은 3, 사람 2는 1입니다.

둘 다 작업 2의 값을 골랐습니다.

각자의 가장 싼 가격만 더했습니다.

남은 비용을 낮게 잡은 값은 3 + 1 = 4입니다.

⑥ 같은 작업을 골라도 하한 계산에 쓸 수 있는 이유

각 사람의 실제 비용은 자기 줄의 최솟값 이상입니다.

사람 1의 실제 비용은 3 이상입니다.

사람 2의 실제 비용은 1 이상입니다.

따라서 두 사람의 실제 비용 합은 4 이상입니다.

작업이 겹쳐 한 사람이 더 비싼 일을 맡으면, 합은 더 커집니다.

⑦ 실제 배정 A의 남은 비용은 7입니다



사람 1에게 작업 0을 맡깁니다.

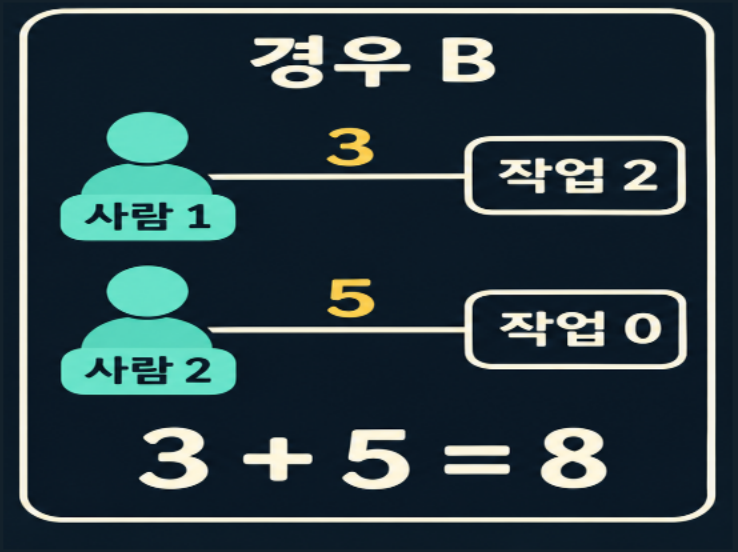
사람 2는 작업 2를 맡습니다.

비용은 각각 6과 1입니다.

서로 다른 작업을 맡았습니다.

실제 남은 비용 A = 6 + 1 = 7

⑧ 실제 배정 B의 남은 비용은 8입니다



이번에는 사람 1에게 작업 2를 맡깁니다.

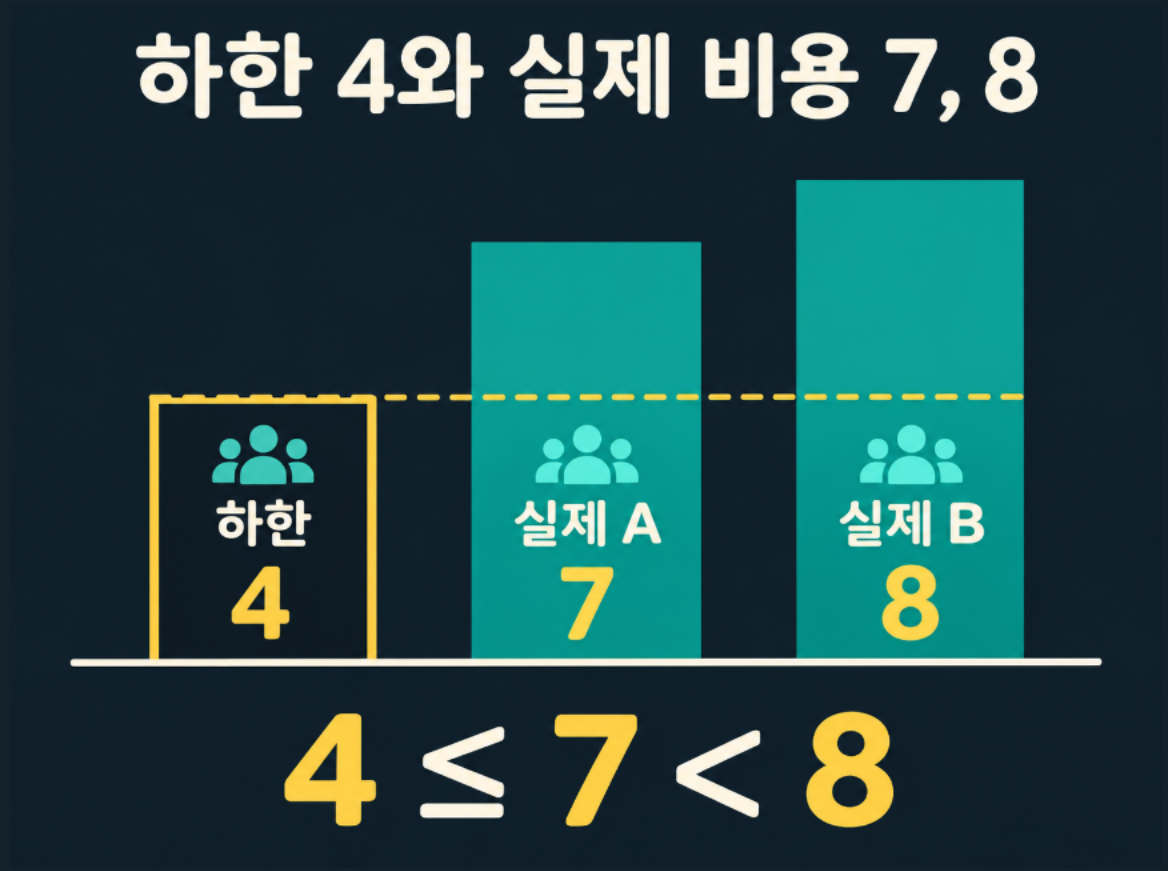
사람 2는 작업 0을 맡습니다.

비용은 각각 3과 5입니다.

이번에도 작업은 겹치지 않습니다.

실제 남은 비용 B = 3 + 5 = 8

⑨ 하한 4는 실제 비용 7과 8의 아래에 있습니다



하한은 실제 비용보다 커지지 않는 값입니다.

낮게 잡은 값은 4입니다.

실제 가능한 값은 7과 8입니다.

실제 남은 최솟값은 7입니다.

4 ≤ 실제 남은 최소 비용 7

⑩ 하한은 빠르게 계산할 수 있습니다

각 사람의 가장 싼 비용만 한 번씩 찾으면 됩니다.

사람별 후보를 한 줄씩 읽습니다.

작업 중복까지 풀어서 모든 배정을 비교할 필요가 줄어듭니다.

낮게 잡은 값만으로도 포기할 갈래를 찾을 수 있습니다.

하한이 실제 최솟값에 가까울수록 더 많은 갈래를 멈출 수 있습니다.

⑪ 8-2 함수의 입력과 반환값

`assignment_bound(cost, used, next_row)`

비용 표 cost

[[9, 2, 7], [6, 4, 3], [5, 8, 1]]
행은 사람 번호, 열은 작업 번호입니다.

현재 배정 상태

used=[0, 1, 0], next_row=1
작업 1은 사용 중이며, 사람 1부터 비용을 계산합니다.

반환값

사람 1은 min(6, 3)=3, 사람 2는 min(5, 1)=1
두 사람이 고른 값의 합 4를 반환합니다.

남은 비용 3 + 1 = 4 · 앞서 쓴 비용 2는 제외

사람·작업 1~8개 · 비용 0~100 · next_row=n이면 남은 사람이 없으므로 0

⑫ 이미 쓴 비용과 남은 비용을 합쳐 봅니다

계산할 값	계산식	결과
완성 비용의 하한	2 + 4	6
실제 배정 A	2 + 7	9
실제 배정 B	2 + 8	10

8-2가 반환하는 값은 남은 하한 4입니다.

사람 0에게 이미 쓴 비용은 2입니다.

전체 하한은 2+4=6입니다.

실제 완성 비용은 9 또는 10입니다.

⑬ 전체 하한 6으로는 이 갈래를 멈출 수 없습니다

이미 찾은 완성 답이 9라고 가정하겠습니다.

현재 갈래의 전체 비용은 적어도 6입니다.

6은 이미 찾은 답 9보다 작습니다.

하한만으로는 더 좋은 답의 가능성을 지울 수 없습니다.

이 갈래는 계속 살펴봅니다.

⑭ 사람 0에게 작업 2를 맡긴 다른 갈래

사람 / 작업	작업 0	작업 1	작업 2
사람 0	9	2	7
사람 1	6	4	3
사람 2	5	8	1

남은 하한은 4 + 5 = 9입니다.

이번 갈래에서는 이미 7을 썼습니다.

남은 작업은 0과 1입니다.

사람 1의 최솟값은 4입니다.

사람 2의 최솟값은 5입니다.

⑮ 하한이 이미 찾은 답보다 비싸면 멈춥니다



전체 하한 16 ≥ 이미 찾은 답 9이므로 이 갈래를 중단합니다.

⑯ used는 이미 맡긴 작업을 표시합니다

작업 번호	used 값	처리
0	0	후보로 확인
1	1	건너뜀
2	0	후보로 확인

원래 예제의 used는 [0, 1, 0]입니다.

작업 1은 이미 사용했습니다.

남은 사람마다 같은 표시를 읽습니다.

하한 계산 중에는 used의 값을 바꾸지 않습니다.

⑰ 첫 번째 사람을 읽는 순서

지금 읽는 칸	처리	low
작업 0의 6	후보 확인	6
작업 1의 4	이미 사용해 건너뜀	6
작업 2의 3	더 작은 값으로 변경	3

한 줄을 읽을 때마다 low를 새로 시작합니다.

r=1은 사람 1의 줄입니다.

low는 현재 줄에서 찾은 최솟값입니다.

줄을 다 읽으면 low는 3입니다.

⑱ 두 번째 사람을 읽고 합을 완성합니다

시점	이번 low	answer
시작	아직 없음	0
사람 1을 다 읽음	3	3
사람 2를 다 읽음	1	4

answer는 사람별 최솟값의 합입니다.

사람 1의 3을 더하면 answer=3입니다.

사람 2의 1을 더하면 answer=4입니다.

low는 한 사람의 값, answer는 처리한 사람들의 합입니다.

8-2 Python 1~3행 · 입력과 합 준비하기

```
01  def assignment_bound(cost, used, next_row):
02      n = len(cost)
03      answer = 0
```

변수·상태	예제에서의 값
n	3
next_row	1
answer	0

입력과 합 준비하기

함수는 비용 표 cost, 작업 사용 표시 used, 다음 사람 번호 next_row를 받습니다. 이 세 값으로 어느 사람부터 어떤 작업을 후보로 볼지 결정합니다.

입력과 합 준비하기

8-2 Python 4행 · 남은 사람만 읽기

```
04        for r in range(next_row, n):
```

변수·상태	예제에서의 값
r	1, 2
사람 0	이미 배정 완료

남은 사람만 읽기

r은 지금 확인하는 사람의 번호입니다. next_row가 1, n이 3이면 r은 1과 2가 됩니다. 배정을 마친 사람 0의 행은 다시 읽지 않습니다.

남은 사람만 읽기

8-2 Python 5행 · 사람마다 최솟값 다시 준비하기

```
05          low = 10**9
```

변수·상태	예제에서의 값
새 사람 시작	low = 10**9
첫 후보 6	low = 6

사람마다 최솟값 다시 준비하기

low는 지금 한 사람의 가장 싼 후보 비용입니다. 십억으로 시작하면 실제 비용 최대 100보다 크므로 처음 만나는 후보가 바로 low가 됩니다. 이 줄이 사람 반복문 안에 있어 다음 사람마다 다시 크게 시작합니다.

사람마다 최솟값 다시 준비하기

8-2 Python 6~7행 · 남은 작업만 후보로 고르기

```
06            for c in range(n):
07                if used[c] == 0:
```

변수·상태	예제에서의 값
used	[0, 1, 0]
확인할 작업	0, 2

남은 작업만 후보로 고르기

c는 작업 번호입니다. 0부터 n-1까지 모든 작업을 살펴봅니다. 이미 사용한 작업인지는 다음 줄에서 따로 확인합니다.

남은 작업만 후보로 고르기

8-2 Python 8행 · 더 싼 가격 기억하기

08

```
low = min(low, cost[r][c])
```

변수·상태	예제에서의 값
사람 1 후보	6, 3
최종 low	3

더 싼 가격 기억하기

min은 두 수 중 작은 값을 돌려줍니다. 이전 low와 이번 사람의 이번 작업 비용을 비교해서 더 작은 값을 low에 저장합니다. 사람 1에서는 후보 6을 읽고, 다음 후보 3을 읽어 최종 low가 3이 됩니다.

더 싼 가격 기억하기

8-2 Python 9행 · 한 사람의 값을 한 번만 더하기

```
09            answer += low
```

변수·상태	예제에서의 값
사람 1 뒤	answer = 3
사람 2 뒤	answer = 4

한 사람의 값을 한 번만 더하기

안쪽 작업 반복문이 끝났으므로 지금 사람의 low를 answer에 한 번 더합니다. 들여쓰기가 안쪽 for보다 한 단계 바깥입니다. 후보를 읽을 때마다 더하면 같은 사람을 여러 번 더하게 되므로 한 행을 다 읽은 뒤에 더합니다.

한 사람의 값을 한 번만 더하기

8-2 Python 10행 · 남은 비용의 하한 반환하기

```
10          return answer
```

변수·상태	예제에서의 값
반환값	4
이미 쓴 비용	합에 포함하지 않음

남은 비용의 하한 반환하기

남은 모든 사람을 처리한 answer를 반환합니다. 예제는 사람 1의 3과 사람 2의 1을 더해 4입니다. 실제 배정 최소 비용 7은 별도 문제에서 구하며, 여기서는 낮게 잡은 값 4가 답입니다.

남은 비용의 하한 반환하기

8-2 Python 전체 풀이

```python
def assignment_bound(cost, used, next_row):
    n = len(cost)
    answer = 0
    for r in range(next_row, n):
        low = 10**9
        for c in range(n):
            if used[c] == 0:
                low = min(low, cost[r][c])
        answer += low
    return answer
```

남은 사람들의 행별 최소 비용을 더한 정수를 반환한다.

⑳ C 코드의 기호를 먼저 읽어 봅니다

두 반복문도 Python과 같은 순서로 움직입니다.

r은 사람 번호, c는 작업 번호입니다.

!used[c]는 사용하지 않은 작업인지 확인합니다.

&&는 양쪽 조건을 모두 만족할 때 참입니다.

이번 비용이 low보다 작을 때 low를 바꿉니다.

8-2 C 1~3행 · 입력과 합 준비하기

```
01  int assignment_bound(const int cost[][8], int n,
02                       const int used[], int next_row) {
03      int answer=0;
```

변수·상태	예제에서의 값
n	3
next_row	1
answer	0

입력과 합 준비하기

정수를 반환하는 함수이며 cost는 비용 표입니다. n은 실제 사람과 작업의 수입니다. 배열에 쓴 8은 한 행의 저장 공간이므로 계산할 때에는 앞 n행과 n열만 읽습니다.

입력과 합 준비하기

8-2 C 4~5행 · 사람마다 low 다시 만들기

```
04        for (int r=next_row; r<n; ++r) {
05            int low=1000000;
```

변수·상태	예제에서의 값
r	1, 2
새 low	1000000

사람마다 low 다시 만들기

r은 next_row에서 시작해 n보다 작은 동안 증가합니다. 예제에서는 사람 1과 2의 행을 처리합니다. 중괄호 안의 여러 문장을 한 사람마다 실행합니다.

사람마다 low 다시 만들기

8-2 C 6~7행 · 두 조건을 모두 확인하기

```
06          for (int c=0; c<n; ++c)
07              if (!used[c] && cost[r][c]<low) low=cost[r][c];
```

변수·상태	예제에서의 값
used[1]	1이라 제외
low 변화	1000000 → 6 → 3

두 조건을 모두 확인하기

c를 0부터 n-1까지 증가시키며 작업을 읽습니다. 이 for에 중괄호가 없으므로 바로 다음 if 문장 하나가 반복할 대상입니다.

두 조건을 모두 확인하기

8-2 C 8~9행 · 줄을 다 읽은 뒤 한 번 더하기

```
08            answer+=low;
09        }
```

변수·상태	예제에서의 값
사람별 low	3, 1
answer	0 → 3 → 4

줄을 다 읽은 뒤 한 번 더하기

작업을 모두 읽었으므로 이번 사람의 최솟값을 answer에 더합니다. 이 문장은 안쪽 for의 반복 대상에 속하지 않아 사람마다 한 번 실행합니다.

줄을 다 읽은 뒤 한 번 더하기

8-2 C 10~11행 · 하한을 정수로 반환하기

```
10          return answer;
11      }
```

변수·상태	예제에서의 값
return answer	4
반환 자료형	int

하한을 정수로 반환하기

모든 사람의 최솟값을 더한 answer를 반환합니다. 예제의 값은 4입니다.

하한을 정수로 반환하기

8-2 C 전체 풀이

```c
int assignment_bound(const int cost[][8], int n,
                     const int used[], int next_row) {
    int answer=0;
    for (int r=next_row; r<n; ++r) {
        int low=1000000;
        for (int c=0; c<n; ++c)
            if (!used[c] && cost[r][c]<low) low=cost[r][c];
        answer+=low;
    }
    return answer;
}
```

남은 사람들의 행별 최소 비용을 더한 정수를 반환한다.

㉒ 남은 사람이 없으면 0을 반환합니다

next_row와 n이 같으면 모든 배정이 끝났습니다.

새로 계산할 사람도, 남은 비용도 없습니다.

사람 반복문을 한 번도 실행하지 않습니다.

answer는 처음 값 0으로 남습니다.

이 함수는 앞으로 남은 비용만 계산합니다.

㉓ 이번 함수가 하는 일을 다시 읽어 봅니다

남은 사람마다, 남은 작업 중 가장 싼 비용을 골라 더합니다.

같은 작업의 값을 여러 행에서 골라도 됩니다.

각 사람의 실제 비용은 그 최솟값보다 작아질 수 없습니다.

그래서 합도 실제 남은 비용보다 커지지 않습니다.

예제의 반환값은 하한 4이고, 실제 남은 최소 비용은 7입니다.

8-3 모든 사람에게 서로 다른 작업 배정

사람마다 작업 하나를 맡겼을 때 가능한 가장 작은 총비용을 구합니다.

주어진 것 사람 n명과 작업 n개가 있습니다. 각 조합의 비용이 표로 주어집니다.

배정 규칙 모든 사람은 작업을 정확히 하나 맡고, 모든 작업은 한 사람에게만 맡깁니다.

비교할 것 규칙을 만족하는 배정들을 비교해, 사람별 비용을 더한 값이 가장 작은 것을 찾습니다.

구할 답은 최소 총비용입니다. 정수 한 개를 반환합니다.

8-3 입력과 반환값 예제

`assignment_min(cost)`

비용 표 cost

[[9, 2, 7], [6, 4, 3], [5, 8, 1]]
cost[r][c]는 사람 r에게 작업 c를 맡길 때 드는 비용입니다.

가장 싼 배정

사람 0은 작업 1, 사람 1은 작업 0,
사람 2는 작업 2를 맡습니다. 서로 다른 작업입니다.

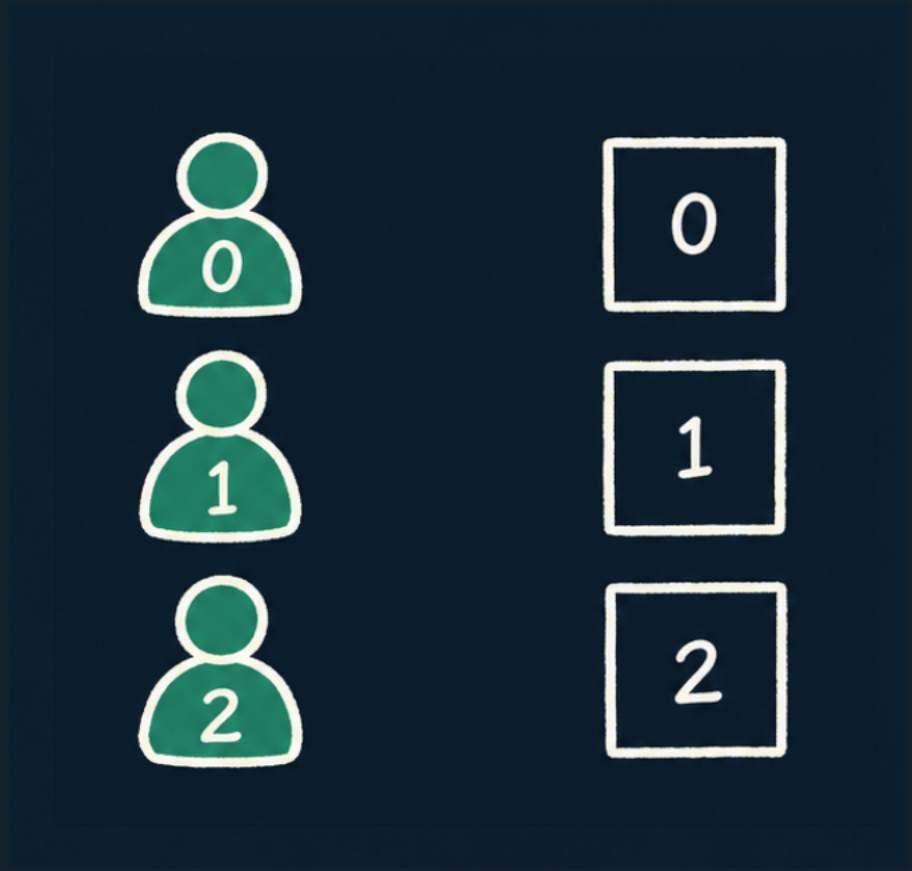
반환값

각 사람에게 드는 비용은 2, 6, 1입니다.
가능한 모든 배정 중 가장 싼 합인 9를 반환합니다.

2 + 6 + 1 = 9

사람·작업 1~8개 · 비용 0~100 · 한 명이면 표의 유일한 비용을 반환

아무도 배정하지 않은 상태에서 시작합니다



다음 사람 row=0, 현재 비용 total=0

used=[False, False, False]

모든 작업을 아직 고를 수 있습니다.

사람 0에게 맡길 작업부터 하나씩 시도합니다.

8-3 다음 사람 번호와 현재 비용



search(1, 2)는 이 상태에서 이어 갑니다.

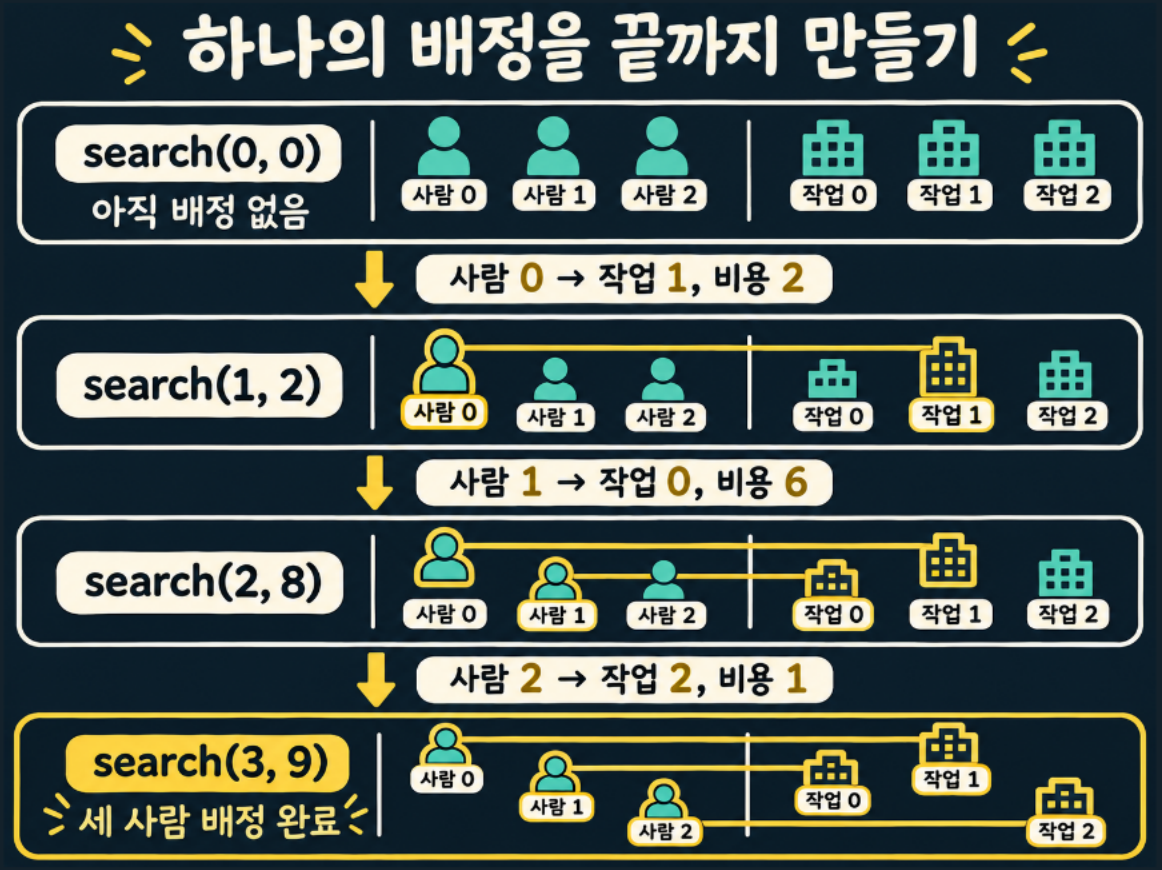
사람 0은 작업 1을 맡았습니다.

남은 사람은 1과 2입니다.

지금까지 비용은 2입니다.

괄호 안의 두 값은 다음 사람과 이미 쓴 비용입니다.

8-3 한 가지 배정을 끝까지 따라가 봅니다



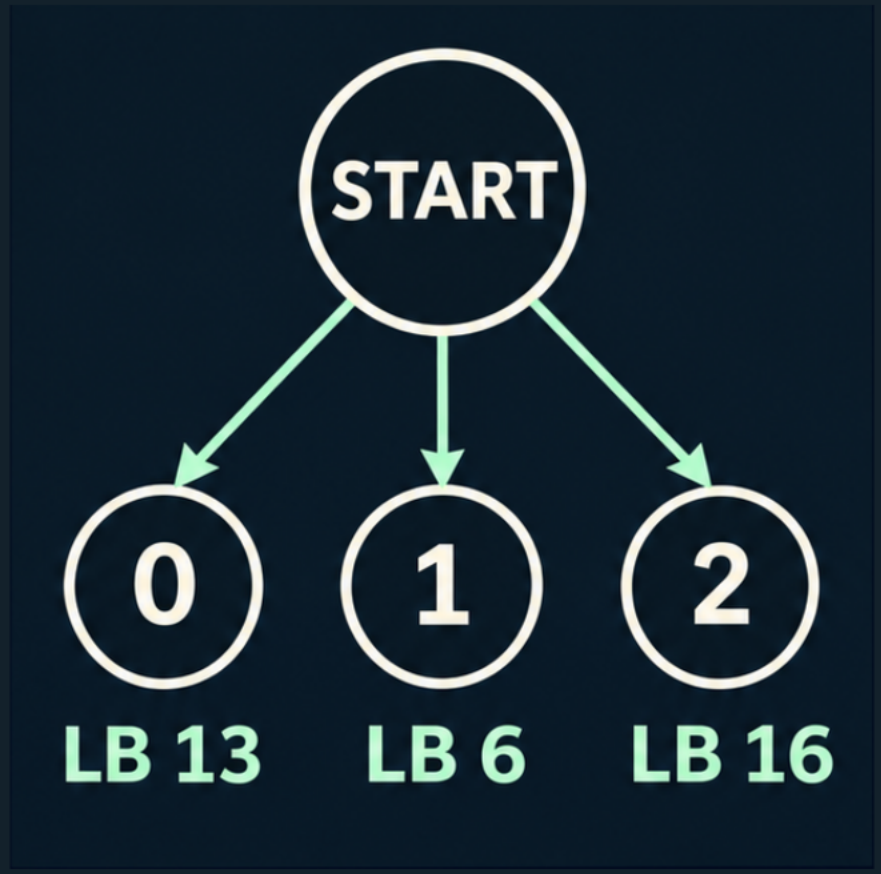
각 호출은 다음 사람 한 명의 작업을 결정합니다.

사람 번호는 0, 1, 2, 완료 상태 3입니다.

비용은 0, 2, 8, 9로 바뀝니다.

마지막 호출에서 세 사람의 배정이 끝납니다.

첫 사람의 선택이 세 갈래로 나뉩니다



원 아래 숫자는 사람 0에게 맡길 작업입니다.

0을 고르면 지금까지의 비용은 9

1을 고르면 비용은 2

2를 고르면 비용은 7

LB는 각 갈래의 완성 비용을 낮게 잡은 값입니다.

우선 한 배정을 끝까지 완성해 봅니다

사람 / 작업	작업 0	작업 1	작업 2
사람 0	9	2	7
사람 1	6	4	3
사람 2	5	8	1

작업을 0, 1, 2 순서로 맡깁니다.

사람 0의 비용 9

사람 1의 비용 4

사람 2의 비용 1

완성 비용 14를 best에 기억합니다.

첫 번째 큰 갈래에서 얻는 비용은 14와 20입니다



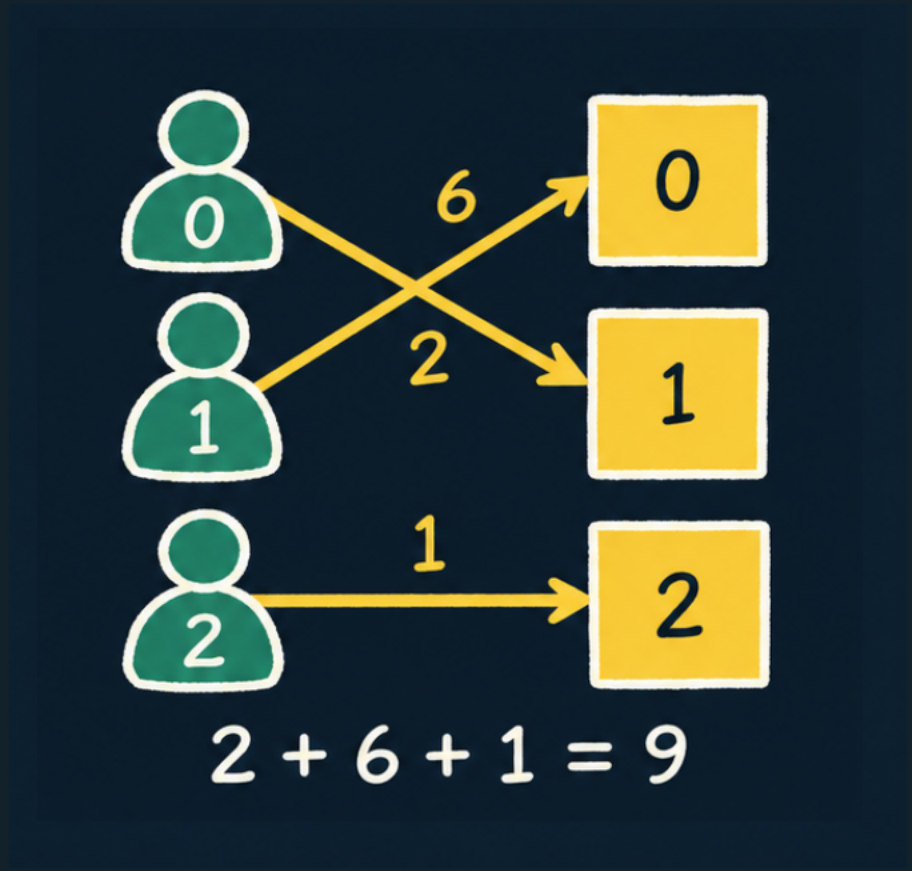
사람 0에게 작업 0을 맡긴 갈래

남은 작업은 1과 2입니다.

두 완성 배정의 비용은 14와 20입니다.

더 작은 14를 기억하고 다른 큰 갈래로 갑니다.

다음 갈래에서 더 싼 답 9를 찾습니다



사람 0은 작업 1을 맡습니다.

사람 1은 작업 0을 맡습니다.

사람 2는 작업 2를 맡습니다.

best를 14에서 9로 갱신합니다.

같은 큰 갈래의 다른 배정은 10입니다



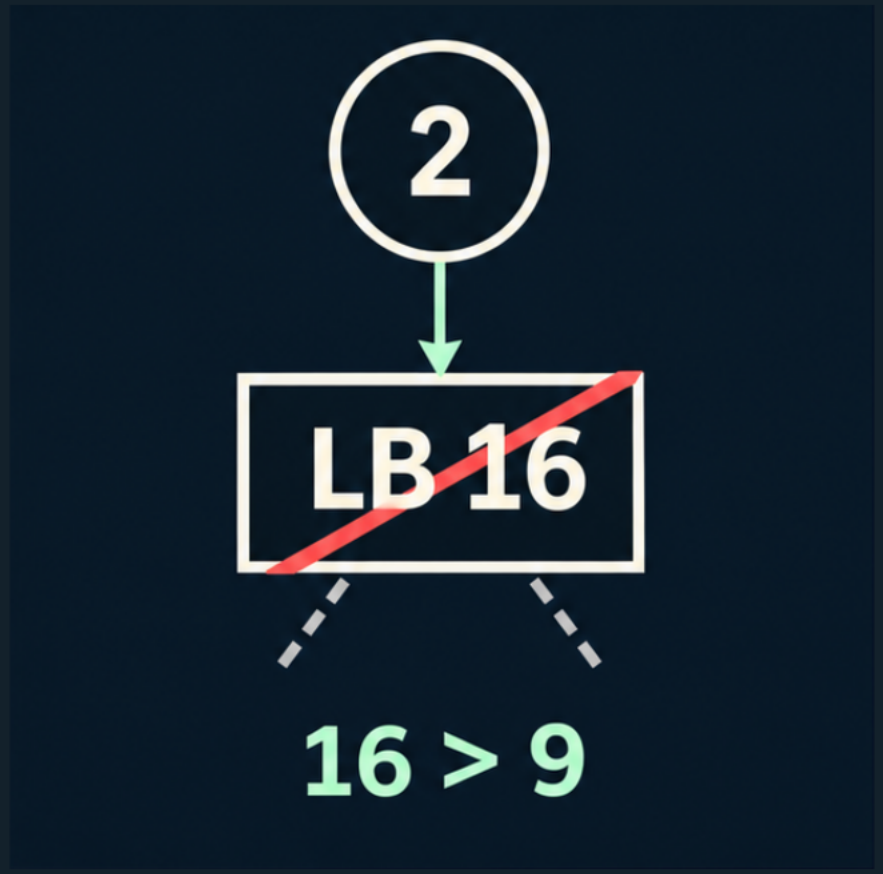
사람 0에게 작업 1을 맡긴 갈래

남은 작업을 0,2로 주면 총비용 9

남은 작업을 2,0으로 주면 총비용 10

best는 더 작은 값 9를 유지합니다.

하한 16인 갈래는 더 계산하지 않아도 됩니다



사람 0에게 작업 2를 맡기면 현재 7

남은 비용 하한은 4+5=9입니다.

완성 비용은 적어도 7+9=16입니다.

하한을 활용하면 이 갈래 전체를 바로 멈춥니다.

기본 코드에서는 현재 비용을 하한으로 씁니다

남은 비용은 모두 0 이상입니다.

현재 비용이 12이면 완성 비용도 12 이상입니다.

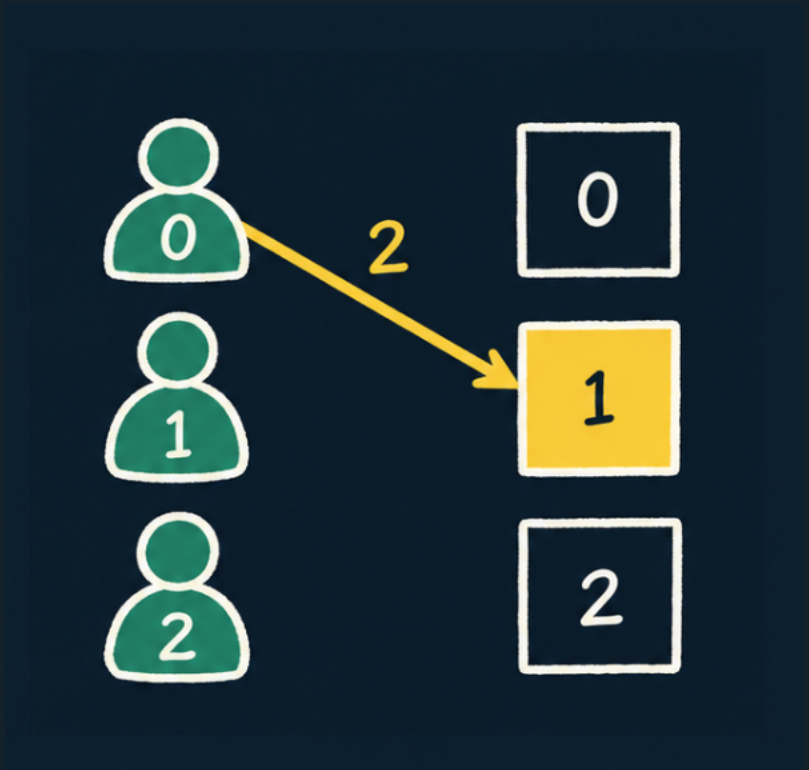
best가 9이면 12 이상인 갈래를 멈춥니다.

현재 비용이 9여도 더 작은 답을 만들 수 없습니다.

판정식은 total >= best입니다.

함수 호출은 현재 상태를 다음 실행에 넘깁니다

```
search(row + 1, total + cost[row][col])
```



row=0, col=1, total=0인 상태

사람 0에게 작업 1을 맡깁니다. 비용은 2입니다.

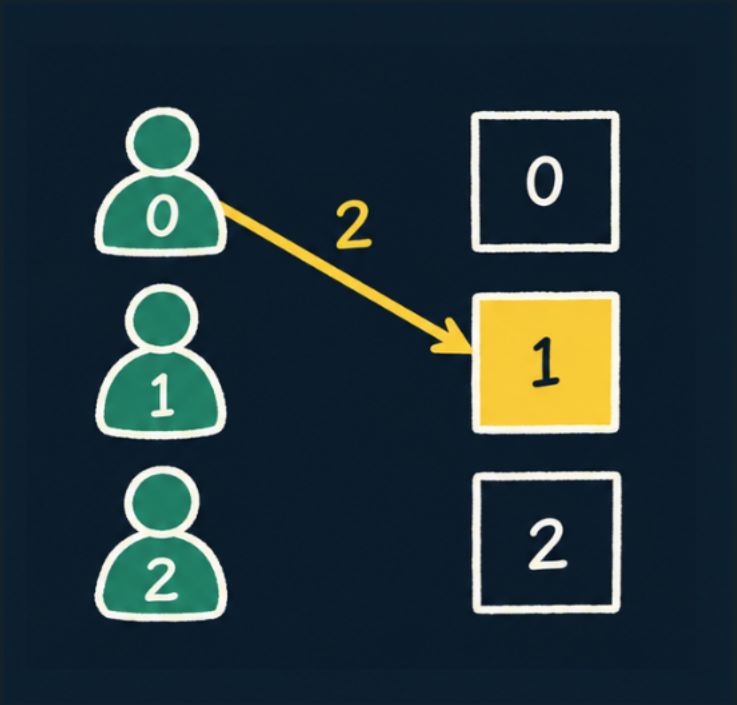
다음 실행은 search(1, 2)입니다.

사람 1부터 고르고, 이미 쓴 비용 2를 기억합니다.

첫 인자는 다음 사람, 둘째 인자는 현재까지의 비용입니다.

다음 호출 전에 고른 작업을 표시합니다

```
used[col] = True
search(row+1, total+cost[row][col])
```



작업 1을 고르면 used[1]을 True로 바꿉니다.

다음 사람은 작업 1을 다시 고를 수 없습니다.

나머지 작업 0과 2만 후보로 남습니다.

모든 호출이 같은 used 배열을 읽습니다.

안쪽 탐색이 끝나면 작업 표시를 되돌립니다

```
used[col] = False
```



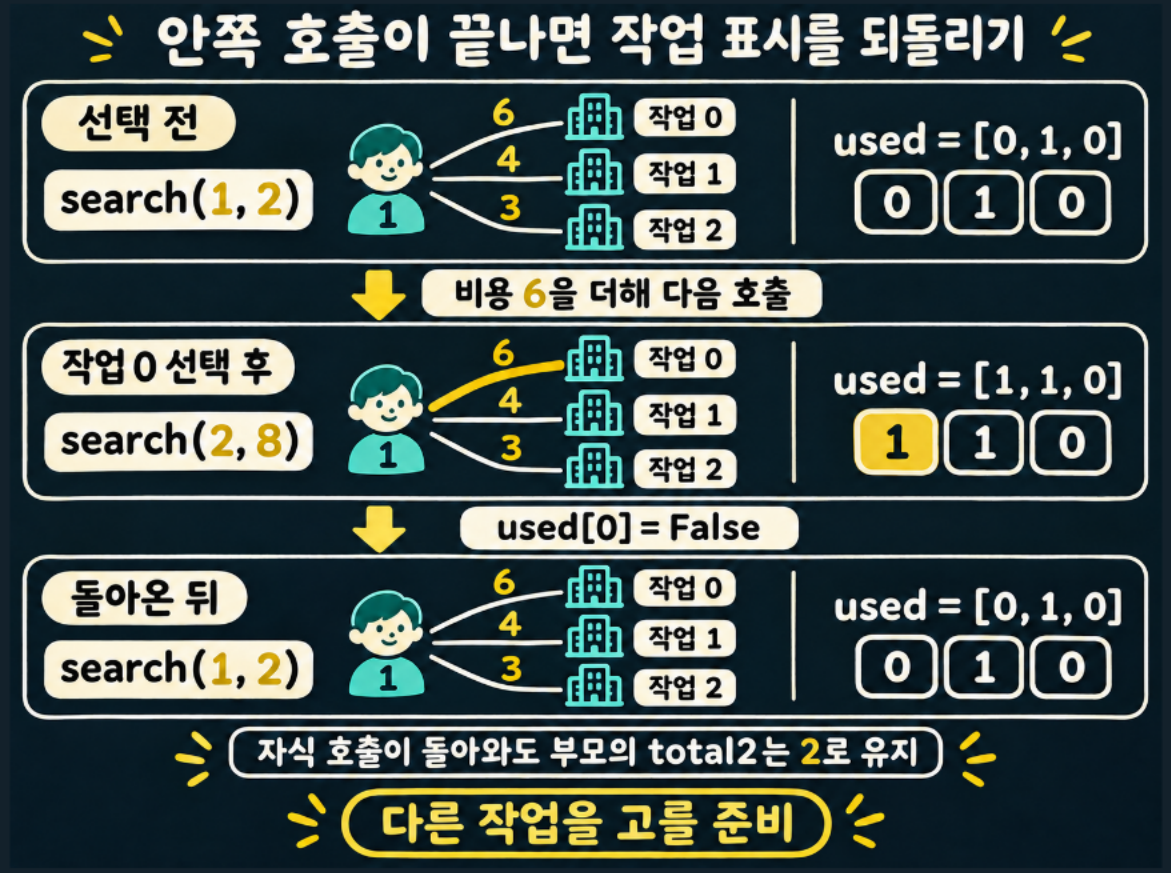
한 선택을 다 확인하고 돌아왔습니다.

이제 같은 사람에게 다른 작업을 맡겨 봅니다.

다른 갈래에서도 작업 1을 사용할 수 있어야 합니다.

True로 표시하고, 호출하고, False로 되돌립니다.

8-3 자식 호출에서 돌아온 뒤의 상태



부모의 total은 계속 2입니다.

자식에게는 2+6인 8을 넘겼습니다.

함께 쓰는 used만 직접 되돌립니다.

그다음 다른 작업을 고릅니다.

숫자 인자와 공유 배열을 나누어 생각합니다.

best는 여러 호출이 공유하는 최소 비용입니다

nonlocal best

best는 바깥 함수에서 만든 변수입니다.

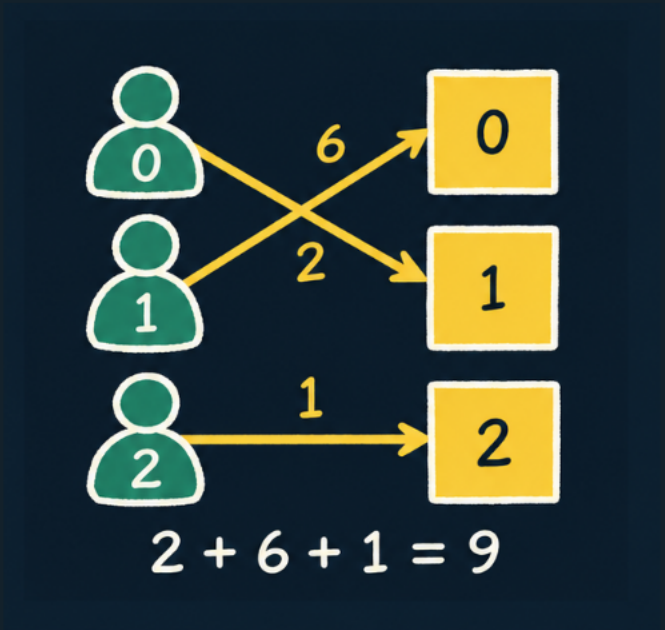
안쪽 search가 완성 배정을 찾으면 best를 바꿉니다.

다른 갈래에서도 갱신된 best를 읽습니다.

nonlocal은 바깥의 best를 갱신하겠다는 표시입니다.

완성 배정을 찾은 순간 기록을 바꿉니다

```
if row == n:
    best = total
    return
```



사람 0,1,2를 배정하면 다음 row는 3입니다.

n도 3이므로 배정이 끝났습니다.

현재 total을 best에 저장하고 돌아갑니다.

먼저 total >= best를 걸렀으므로 더 싼 완성 값입니다.

8-3 Python 1~4행 · 아무도 배정하지 않은 상태 만들기

```
01  def assignment_min(cost):
02      n = len(cost)
03      used = [False]*n
04      best = 10**9
```

변수·상태	예제에서의 값
n	3
used	[False, False, False]
best	1000000000

아무도 배정하지 않은 상태 만들기

assignment_min은 모든 사람에게 서로 다른 작업을 하나씩 맡기는 최소 총비용을 구합니다. cost 표 하나를 입력으로 받습니다. 행은 사람, 열은 작업입니다.

아무도 배정하지 않은 상태 만들기

8-3 Python 5~6행 · 다음 사람과 현재 비용을 받기

```
05      def search(row, total):
06          nonlocal best
```

변수·상태	예제에서의 값
row	1
total	2
뜻	사람 1부터, 이미 비용 2

다음 사람과 현재 비용을 받기

search는 배정을 이어 가는 안쪽 함수입니다. row는 다음 사람 번호이고 total은 앞 사람들에게 이미 쓴 비용입니다. 예를 들어 search(1,2)는 사람 0을 배정한 뒤, 비용 2인 상태에서 사람 1부터 계속하라는 호출입니다.

다음 사람과 현재 비용을 받기

8-3 Python 7행 · 이미 비싸진 갈래 멈추기

```
07            if total >= best: return
```

변수·상태	예제에서의 값
total	13
best	9
비교	13 >= 9라서 중단

이미 비싸진 갈래 멈추기

지금 비용이 이미 best 이상이면 이 호출을 끝냅니다. 앞으로 더할 비용은 모두 0 이상이라 합이 줄어들 수 없습니다. 예를 들어 best가 9인데 total이 13이면 계속 배정해도 9보다 싼 답을 만들 수 없습니다. 뒤의 return은 값 없이 이 search 실행을 끝내고 이전 호출로 돌아갑니다.

이미 비싸진 갈래 멈추기

8-3 Python 8~10행 · 모든 사람을 배정하면 기록하기

```
08            if row == n:
09                best = total
10                return
```

변수·상태	예제에서의 값
row / n	3 / 3
완성 total	9
새 best	9

모든 사람을 배정하면 기록하기

row가 n이면 사람 번호 0부터 n-1까지 모두 처리했습니다. row 3인 사람을 새로 배정하는 것이 아니라, 세 사람의 배정이 끝났다는 신호입니다.

모든 사람을 배정하면 기록하기

8-3 Python 11~12행 · 아직 남은 작업 고르기

```
11            for col in range(n):
12                if not used[col]:
```

변수·상태	예제에서의 값
used	[False, True, False]
고를 수 있는 col	0, 2

아직 남은 작업 고르기

다음 사람에게 맡겨 볼 작업 번호 col을 0부터 n-1까지 차례로 봅니다. 한 호출은 현재 row 사람에게 줄 작업을 하나 고르는 일을 합니다.

아직 남은 작업 고르기

8-3 Python 13행 · 선택한 작업 표시하기

13 `used[col] = True`

변수·상태	예제에서의 값
선택 col	0
used 변화	[0,1,0] → [1,1,0]

선택한 작업 표시하기

이번에 고른 작업을 사용 중으로 표시합니다. 이 표시를 다음 호출도 함께 보기 때문에 뒤 사람은 같은 작업을 다시 고를 수 없습니다.

선택한 작업 표시하기

8-3 Python 14행 · 다음 사람에게 배정 이어 주기

14 `search(row+1, total+cost[row][col])`

변수·상태	예제에서의 값
row / total	1 / 2
cost[1][0]	6
새 호출	search(2, 8)

다음 사람에게 배정 이어 주기

다음 사람 번호 row+1과, 이번 작업 비용을 더한 total을 새 호출에 전달합니다. row=1, total=2, col=0이면 cost[1][0]이 6이라 search(2,8)을 실행합니다. 부모 호출의 total 2를 고쳐 쓰는 대입문은 없으므로 부모 값은 그대로 2입니다.

다음 사람에게 배정 이어 주기

8-3 Python 15행 · 돌아오면 선택 표시 지우기

15 `used[col] = False`

변수·상태	예제에서의 값
돌아온 부모	search(1, 2)
used 복구	[1,1,0] → [0,1,0]

돌아오면 선택 표시 지우기

안쪽 호출이 끝나 돌아왔으므로 방금 골랐던 작업 표시를 지웁니다. 그래야 같은 사람에게 다른 작업을 주는 경우도 시작할 수 있습니다. used는 호출들이 함께 쓰는 배열이어서 직접 복구해야 합니다. 각 호출의 숫자 total은 따로 있으므로 여기서 6을 빼는 줄은 필요하지 않습니다.

돌아오면 선택 표시 지우기

8-3 Python 16행 · 첫 호출로 탐색 시작하기

```
16          search(0, 0)
```

변수·상태	예제에서의 값
첫 row	0
첫 total	0
실행할 함수	search(0, 0)

첫 호출로 탐색 시작하기

안쪽 함수를 정의하기만 해서는 배정이 시작되지 않습니다. 사람 0, 비용 0인 출발 상태로 search를 실제 호출합니다. 이 호출이 돌아오면 필요한 모든 갈래를 처리한 상태입니다.

첫 호출로 탐색 시작하기

8-3 Python 17행 · 가장 싼 완성 비용 반환하기

```
17        return best
```

변수·상태	예제에서의 값
완성 후보	14, 20, 9, 10, 21, 16
최솟값	9

가장 싼 완성 비용 반환하기

search들이 함께 갱신한 best를 반환합니다. 예제에서는 배정 비용 14를 처음 찾은 뒤 더 싼 9를 찾아 best가 9가 됩니다. 사이트가 받는 최종 결과는 이 정수 9입니다.

가장 싼 완성 비용 반환하기

8-3 Python 전체 풀이 (1/2)

```python
def assignment_min(cost):
    n = len(cost)
    used = [False]*n
    best = 10**9
    def search(row, total):
        nonlocal best
        if total >= best: return
        if row == n:
            best = total
            return
```

모든 사람에게 작업을 배정한 총비용의 최솟값을 반환한다. 배정 목록은 반환하지 않는다.

8-3 Python 전체 풀이 (2/2)

```
            for col in range(n):
                if not used[col]:
                    used[col] = True
                    search(row+1, total+cost[row][col])
                    used[col] = False
        search(0, 0)
        return best
```

모든 사람에게 작업을 배정한 총비용의 최솟값을 반환한다. 배정 목록은 반환하지 않는다.

C는 갱신한 best를 반환해서 이어받습니다

best = assign_go(...)

C의 보조 함수는 best를 인자로 받습니다.

더 싼 답을 찾으면 바뀐 best를 반환합니다.

호출한 쪽은 반환값을 자신의 best에 저장합니다.

되돌아오는 값은 지금까지 찾은 최소 비용입니다.

8-3 C 1~2행 · 보조 함수의 입력 읽기

```
01  static int assign_go(const int cost[][8], int n, int row,
02                       int used[], int total, int best) {
```

변수·상태	예제에서의 값
row	다음 사람 번호
total	이미 쓴 비용
best	현재까지의 최솟값

보조 함수의 입력 읽기

assign_go는 배정을 이어 가는 보조 함수입니다. static은 이 소스 파일 안에서 사용하는 함수라는 표시입니다. cost와 n 다음의 row는 다음 사람 번호입니다. 함수 이름은 달라도 Python의 search와 같은 역할입니다.

보조 함수의 입력 읽기

8-3 C 3~4행 · 중단과 완료를 먼저 검사하기

```
03        if (total>=best) return best;
04        if (row==n) return total;
```

변수·상태	예제에서의 값
중단 예	13 >= 9
완료 예	row=3, total=9

중단과 완료를 먼저 검사하기

지금 비용이 best 이상이면 더 싼 답을 만들 수 없어 현재 best를 반환합니다. 부모는 이 값을 받아 기존 최소 비용을 유지합니다.

중단과 완료를 먼저 검사하기

8-3 C 5~6행 · 남은 작업을 선택하고 표시하기

```
05        for (int c=0; c<n; ++c) if (!used[c]) {
06            used[c]=1;
```

변수·상태	예제에서의 값
고른 c	0
used[c]	0에서 1로

남은 작업을 선택하고 표시하기

작업 c를 하나씩 보고 사용하지 않은 작업만 고릅니다. 느낌표 used[c]는 값이 0인 작업을 통과시킵니다. 여는 중괄호 안의 선택, 재귀 호출, 복구가 한 묶음입니다.

남은 작업을 선택하고 표시하기

8-3 C 7행 · 반환된 best를 꼭 저장하기

```
07          best=assign_go(cost,n,row+1,used,total+cost[row][c],best);
```

변수·상태	예제에서의 값
다음 호출	row=2, total=8
돌려받은 best	9

반환된 best를 꼭 저장하기

다음 사람과 새 총비용으로 assign_go를 호출합니다. 반환된 최소 비용을 best에 저장합니다. 이 대입을 빼면 안쪽에서 찾은 더 좋은 답을 현재 호출이 잃어버립니다.

반환된 best를 꼭 저장하기

8-3 C 8~11행 · 선택을 복구하고 결과 돌려주기

```
08            used[c]=0;
09        }
10        return best;
11    }
```

변수·상태	예제에서의 값
used[c] 복구	1에서 0으로
return best	부모에게 최소 비용 전달

선택을 복구하고 결과 돌려주기

안쪽 호출이 돌아왔으므로 작업 c를 다시 0으로 만듭니다. 다음 작업을 선택할 때 이전 선택의 흔적이 남지 않게 합니다.

선택을 복구하고 결과 돌려주기

8-3 C 12~14행 · 첫 상태를 만들어 보조 함수 부르기

```
12  int assignment_min(const int cost[][8], int n) {
13      int used[8]={0}; return assign_go(cost,n,0,used,0,1000000000);
14  }
```

변수·상태	예제에서의 값
used 초기값	앞 n칸 모두 0
시작 상태	row=0, total=0

첫 상태를 만들어 보조 함수 부르기

사이트가 직접 호출하는 assignment_min 함수를 정의합니다. 비용 표와 실제 크기를 받아 보조 함수를 시작할 준비를 합니다.

첫 상태를 만들어 보조 함수 부르기

8-3 C 전체 풀이 (1/2)

```c
static int assign_go(const int cost[][8], int n, int row,
                     int used[], int total, int best) {
    if (total>=best) return best;
    if (row==n) return total;
    for (int c=0; c<n; ++c) if (!used[c]) {
        used[c]=1;
        best=assign_go(cost,n,row+1,used,total+cost[row][c],best);
        used[c]=0;
    }
    return best;
}
```

모든 사람에게 작업을 배정한 총비용의 최솟값을 반환한다. 배정 목록은 반환하지 않는다.

8-3 C 전체 풀이 (2/2)

```
12  int assignment_min(const int cost[][8], int n) {
13      int used[8]={0}; return assign_go(cost,n,0,used,0,1000000000);
14  }
```



사람 0, 현재 비용 0에서 시작

탐색 결과를 반환합니다.

모든 사람에게 작업을 배정한 총비용의 최솟값을 반환한다. 배정 목록은 반환하지 않는다.

더 일찍 멈추려면 8-2의 하한을 연결합니다

현재 비용 + 남은 비용 하한을 비교합니다.

기본 코드의 판정은 total >= best입니다.

8-2를 함께 쓰면 total+remaining >= best입니다.

작업 2의 갈래는 7+9=16에서 바로 멈출 수 있습니다.

두 함수를 연결하는 추가 판정

```
remaining = assignment_bound(cost, used, row)
if total + remaining >= best:
    return
```



search 안에서 현재 상태로 하한을 구합니다.

8-2 함수를 같은 파일 위에 작성합니다.

기본 코드의 첫 비용 판정 다음에 이 검사를 넣습니다.

반환값은 같고, 살펴보는 갈래를 더 줄일 수 있습니다.

비용이 0인 경우도 같은 방식으로 처리합니다

처음부터 best를 0으로 두면 안 됩니다.

모든 비용이 0이면 실제 정답도 0입니다.

처음 best는 모든 정답보다 크게 잡습니다.

첫 완성 답 0을 찾은 뒤 나머지는 멈출 수 있습니다.

같은 최솟값을 만드는 배정 수는 이번에 세지 않습니다.

8-4 최소 왕복 경로 · 기말 제외

도시의 방문 순서를 직접 골라, 왕복 비용을 가장 작게 만듭니다.

주어진 것 도시 사이의 이동 비용 표가 주어집니다. 모든 도시 사이를 이동할 수 있습니다.

방문 규칙 0번 도시에서 출발합니다. 다른 도시들을 한 번씩 방문한 뒤 0번으로 돌아옵니다.

비교할 것 가능한 방문 순서마다 왕복 비용을 계산해, 그중 가장 작은 값을 찾습니다.

구할 답은 최소 왕복 비용입니다. 정수 한 개를 반환합니다.

8-4 입력과 반환값 예제

`tsp_min(dist)`

비용 표 dist

[[0, 2, 8], [1, 0, 3], [4, 6, 0]]
dist[i][j]는 도시 i에서 도시 j로 가는 비용입니다.

가능한 두 순서

0, 1, 2, 0의 비용은 2+3+4=9입니다.
0, 2, 1, 0의 비용은 8+6+1=15입니다.

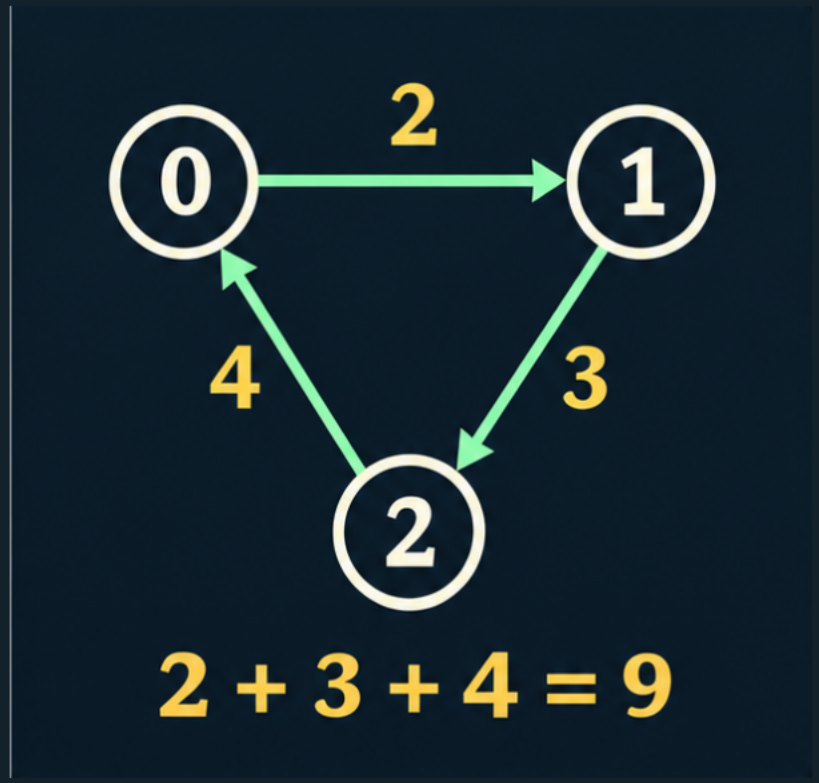
반환값

두 순서 중 더 싼 왕복 비용 9를 반환합니다.
출발지로 돌아오는 마지막 이동까지 포함한 값입니다.

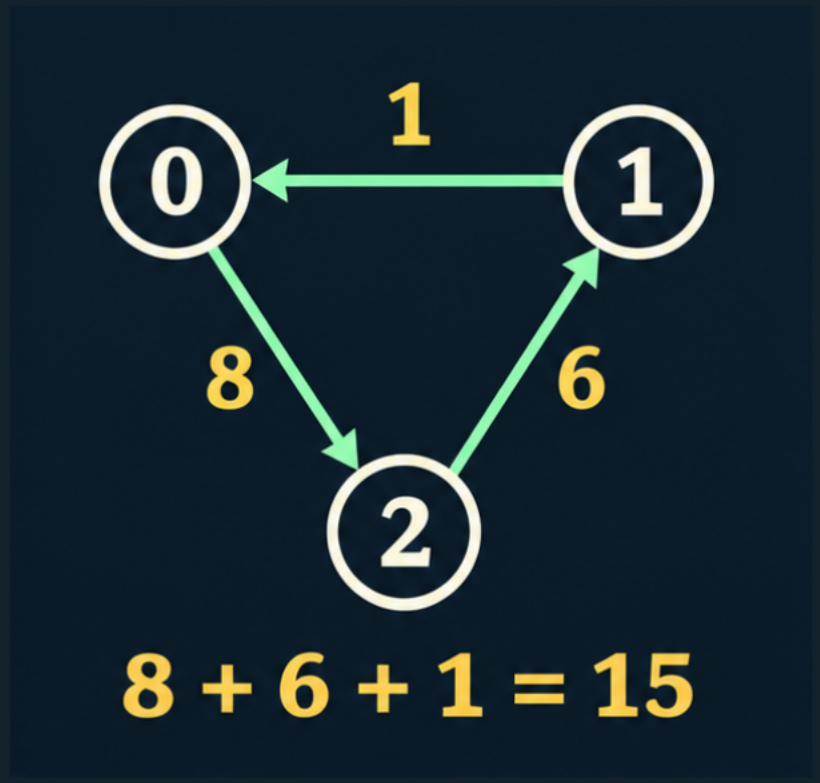
min(9, 15) = 9

도시 1~9개 · 비용 0~100 · 같은 도시로 가는 비용 0 · 도시 1개이면 반환값 0

세 도시의 가능한 왕복을 비교합니다



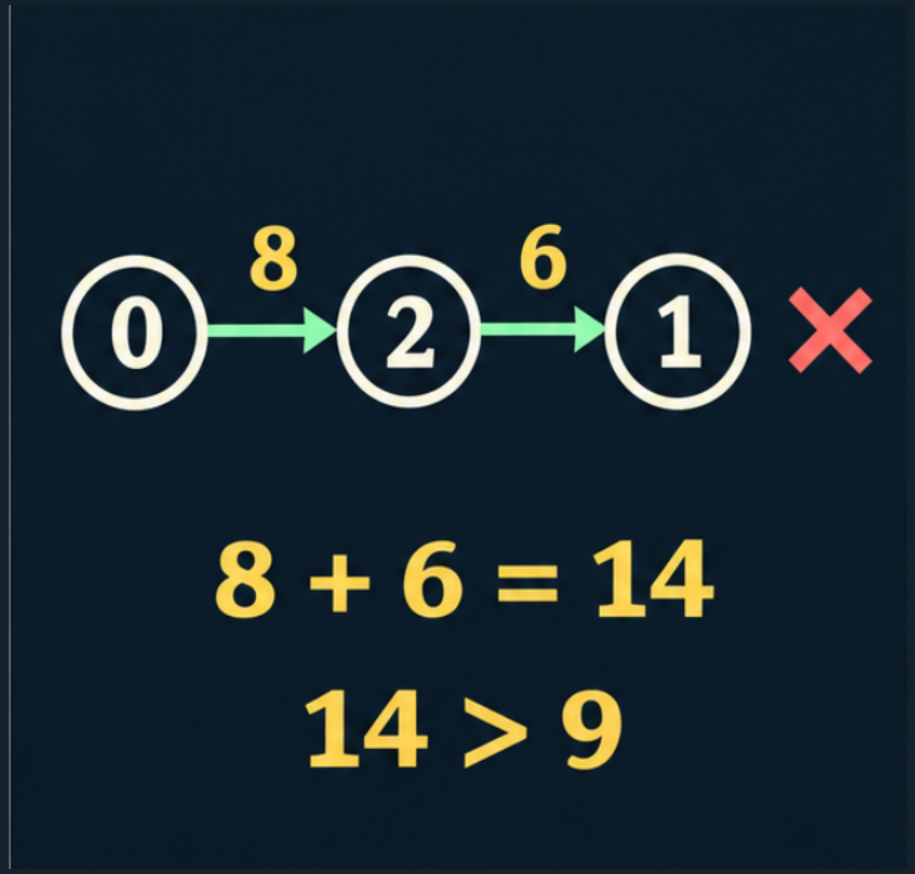
1을 먼저 가면 비용 9



2를 먼저 가면 비용 15

tsp_min(dist)의 반환값은 9입니다.

두 번째 경로는 도중에 멈출 수 있습니다



먼저 완성한 왕복 비용은 9입니다.

0에서 2까지 8을 썼습니다.

2에서 1까지 더하면 현재 비용은 14입니다.

14는 이미 9 이상이므로 귀환 계산 전에 멈춥니다.

도시 0도 처음부터 방문한 도시입니다

used[0] = True

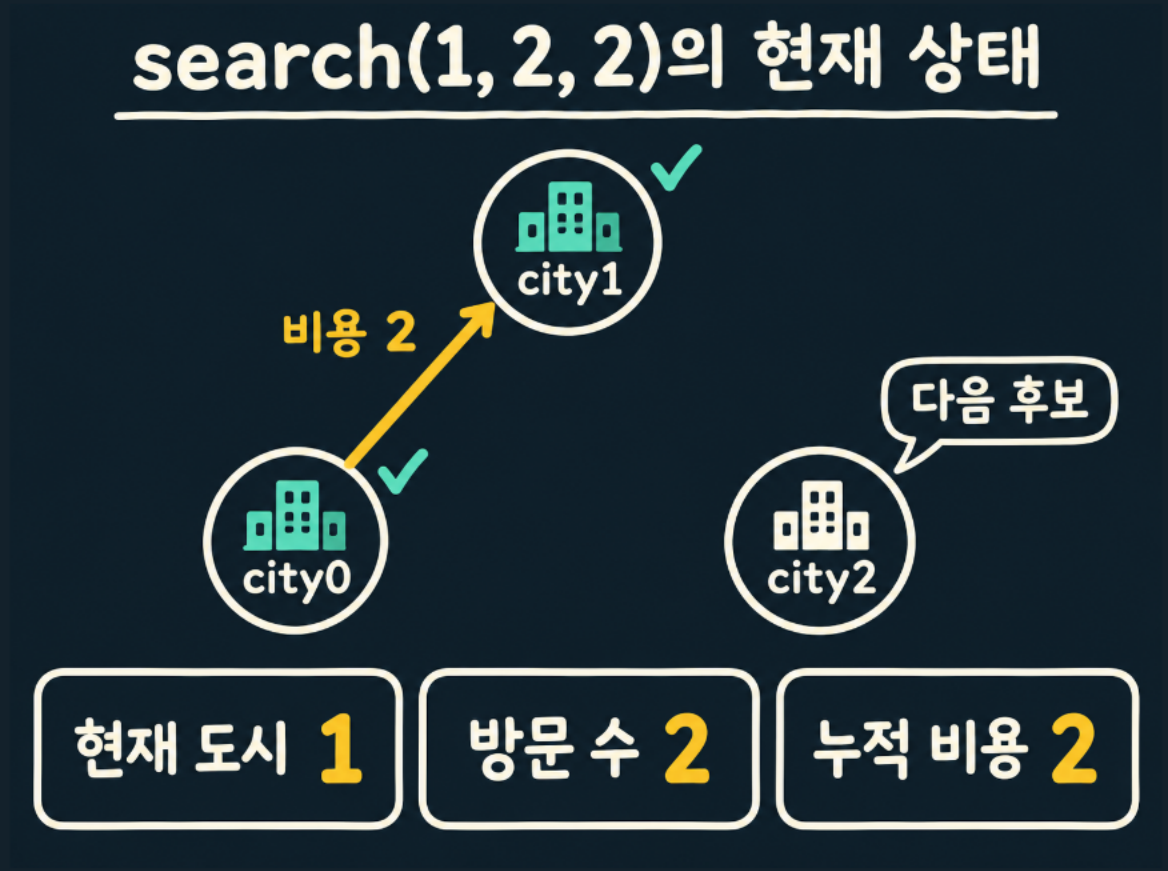
출발할 때 방문 수 count는 1입니다.

현재 도시 last는 0입니다.

지금까지 쓴 total은 0입니다.

첫 호출은 search(0, 1, 0)입니다.

8-4 세 인자를 그림의 상태와 맞춥니다



search(1, 2, 2)를 읽어 봅니다.

현재 도시는 1입니다.

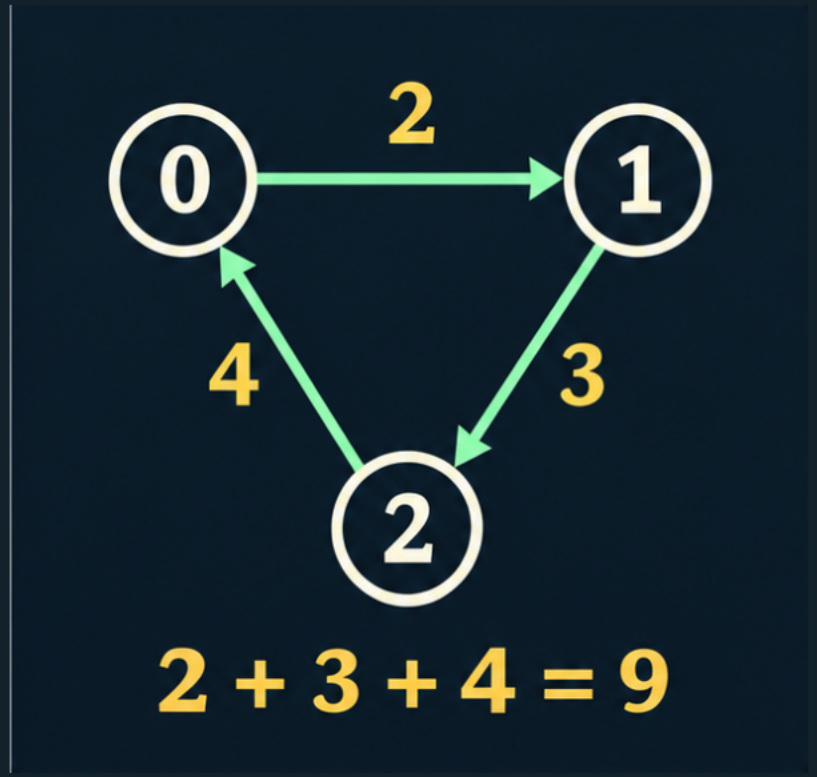
도시 0과 1, 두 곳을 방문했습니다.

지금까지 이동 비용은 2입니다.

다음 후보는 아직 방문하지 않은 도시 2입니다.

다음 도시를 고르면 세 값이 바뀝니다

```
search(nxt, count+1, total+dist[last][nxt])
```



last=0, count=1, total=0에서 nxt=1 선택

다음 현재 도시는 1입니다.

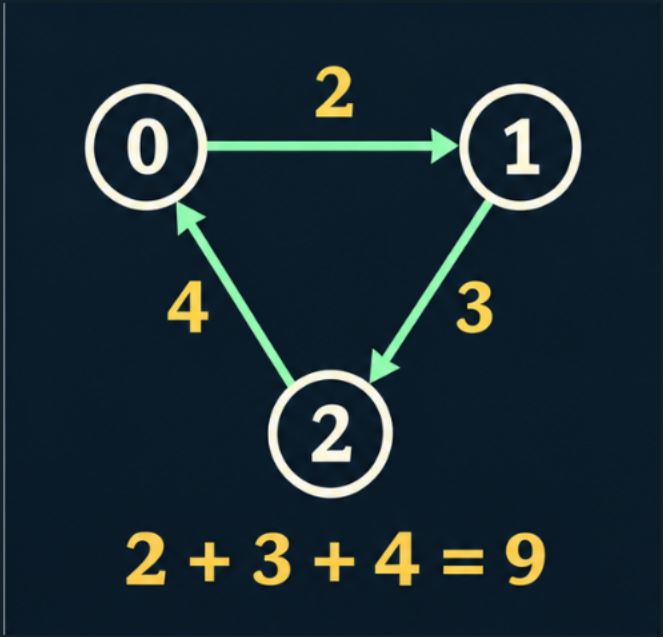
방문 수는 2가 됩니다.

이동 비용 2를 더합니다.

새 호출은 search(1, 2, 2)입니다.

모든 도시를 방문하면 귀환 비용을 더합니다

```
if count == n:
    best = min(best, total + dist[last][0])
    return
```



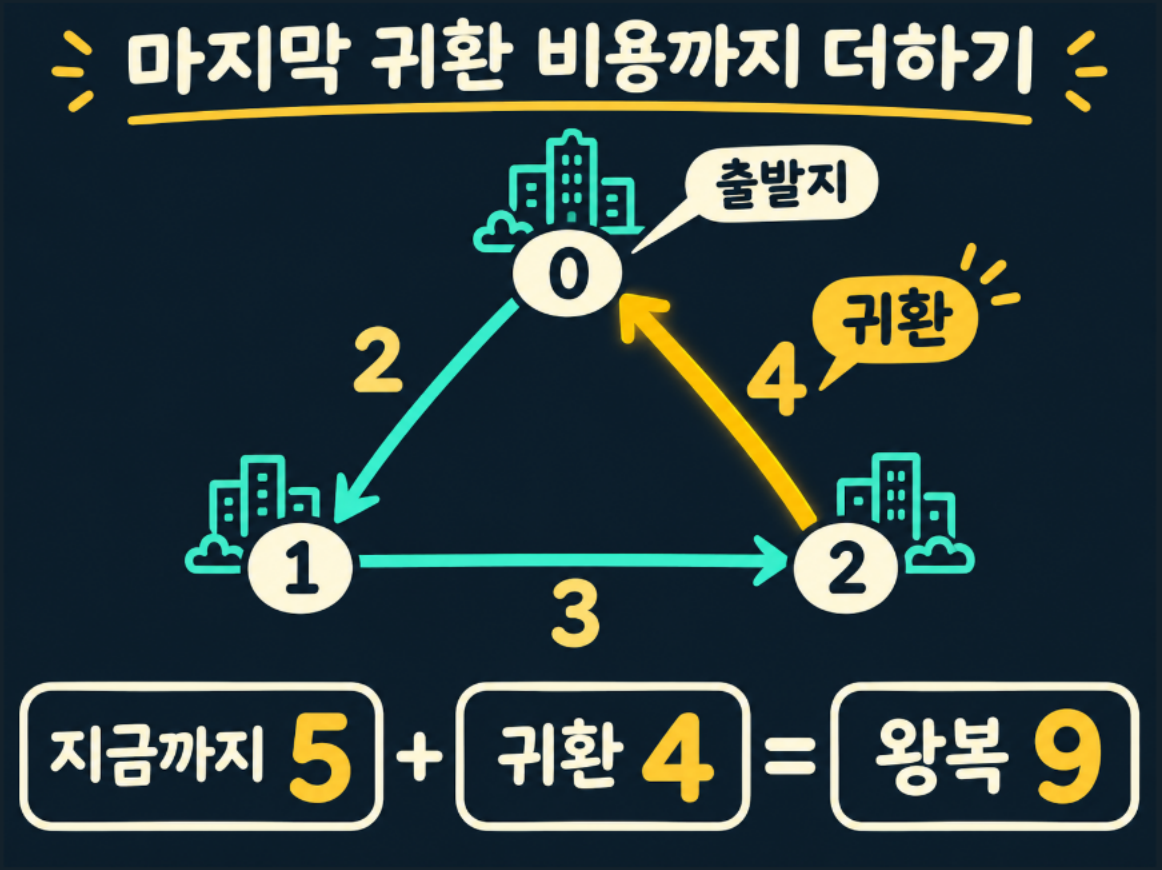
0,1,2를 방문했으면 total=5, last=2입니다.

도시 2에서 0으로 돌아가는 비용은 4입니다.

완성 왕복 비용 5+4=9를 best와 비교합니다.

마지막 귀환 비용을 빠뜨리지 않습니다.

8-4 도시를 다 방문한 뒤 귀환 비용



도시 2까지 왔을 때 비용은 5입니다.

0번에서 1번까지 2

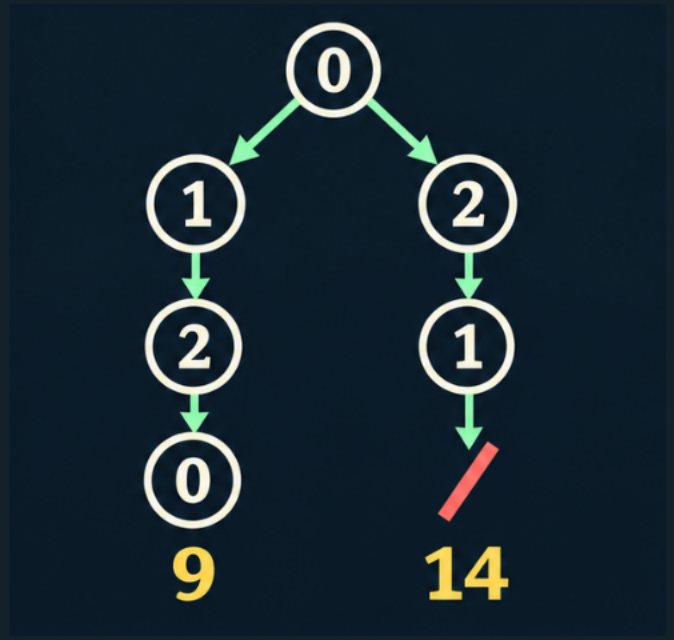
1번에서 2번까지 3

2번에서 0번까지 4

왕복을 완성한 비용은 5 + 4 = 9입니다.

방문 표시를 남기고 들어가서 지우고 나옵니다

```
used[nxt] = True
search(nxt, count+1, total+dist[last][nxt])
used[nxt] = False
```



이번 경로에서 같은 도시를 다시 고르지 않도록 합니다.

안쪽 호출은 방문 표시를 함께 읽습니다.

돌아오면 다른 방문 순서를 시도합니다.

새 순서를 위해 방금 방문한 표시를 복구합니다.

8-4 Python 1~5행 · 출발 도시를 방문한 상태 만들기

```python
def tsp_min(dist):
    n = len(dist)
    used = [False]*n
    used[0] = True
    best = 10**9
```

변수·상태	예제에서의 값
n	3
used	[True, False, False]
best	1000000000

출발 도시를 방문한 상태 만들기

tsp_min은 방문 순서까지 골라 최소 왕복 비용을 구합니다. 입력 dist는 도시 사이의 비용 표입니다. 이 문제는 추가 연습이며 기말 출제 범위에서 제외합니다.

출발 도시를 방문한 상태 만들기

8-4 Python 6~7행 · 도시·방문 수·비용을 받기

```
06        def search(last, count, total):
07            nonlocal best
```

변수·상태	예제에서의 값
last / count / total	1 / 2 / 2
뜻	도시 1, 두 도시 방문, 비용 2

도시·방문 수·비용을 받기

search의 last는 지금 서 있는 도시, count는 지금까지 방문한 도시 수, total은 지금까지 이동한 비용입니다. 세 값의 뜻을 순서대로 고정해서 읽습니다.

도시·방문 수·비용을 받기

8-4 Python 8행 · 이미 비싸진 경로 멈추기

```
08            if total >= best: return
```

변수·상태	예제에서의 값
total / best	14 / 9
결과	이번 경로 중단

이미 비싸진 경로 멈추기

이미 쓴 비용이 best 이상이면 이 경로를 끝냅니다. 이동 비용이 모두 0 이상이라 앞으로 돌아오는 비용까지 더하면 현재 값보다 줄어들 수 없습니다.

이미 비싸진 경로 멈추기

8-4 Python 9~11행 · 모든 도시를 보면 귀환 비용 더하기

```
09            if count == n:
10                best = min(best, total+dist[last][0])
11                return
```

변수·상태	예제에서의 값
last / total	2 / 5
귀환 비용	4
완성 왕복	9

모든 도시를 보면 귀환 비용 더하기

count가 n이면 모든 도시를 방문한 상태입니다. 아직 출발지로 돌아오는 마지막 이동 비용을 더할 차례입니다.

모든 도시를 보면 귀환 비용 더하기

8-4 Python 12~13행 · 아직 안 간 도시를 후보로 고르기

```
12            for nxt in range(1, n):
13                if not used[nxt]:
```

변수·상태	예제에서의 값
후보 nxt	1, 2
도시 0	마지막 귀환에 사용

아직 안 간 도시를 후보로 고르기

다음 도시 후보 nxt는 1부터 n-1까지 봅니다. 출발 도시 0은 마지막 귀환 때만 다시 가므로 중간 방문 후보에서 제외합니다.

아직 안 간 도시를 후보로 고르기

8-4 Python 14행 · 다음 도시를 방문 표시하기

14 `used[nxt] = True`

변수·상태	예제에서의 값
선택 nxt	1
used	[True, True, False]

다음 도시를 방문 표시하기

다음 도시 nxt를 방문 표시합니다. 안쪽 호출에서 이 도시를 다시 후보로 고르지 않도록 호출 전에 표시합니다.

다음 도시를 방문 표시하기

8-4 Python 15행 · 세 상태를 바꿔 다음 호출하기

15

```
search(nxt, count+1, total+dist[last][nxt])
```

변수·상태	예제에서의 값
출발 상태	0, 1, 0
다음 이동 비용	2
새 호출	search(1, 2, 2)

세 상태를 바꿔 다음 호출하기

새 현재 도시는 nxt이고 방문 수는 한 개 늘어납니다. total에는 현재 도시 last에서 nxt로 가는 비용만 한 번 더합니다. last 0, count 1, total 0에서 nxt 1이면 새 호출은 search(1,2,2)입니다.

세 상태를 바꿔 다음 호출하기

8-4 Python 16행 · 돌아오면 방문 표시 지우기

16 `used[nxt] = False`

변수·상태	예제에서의 값
부모 상태	0, 1, 0
used 복구	[True, False, False]

돌아오면 방문 표시 지우기

안쪽 호출을 마치고 돌아왔으므로 nxt의 방문 표시를 지웁니다. 그래야 다른 순서에서도 이 도시를 고를 수 있습니다. 각 호출의 숫자 값은 따로 있으므로 total을 직접 빼서 복구하지 않습니다.

돌아오면 방문 표시 지우기

8-4 Python 17~18행 · 도시 0에서 시작하고 정답 받기

```
17        search(0, 1, 0)
18        return best
```

변수·상태	예제에서의 값
첫 호출	search(0, 1, 0)
완성 비용	9 또는 15
최종 반환	9

도시 0에서 시작하고 정답 받기

도시 0에 서 있고 도시 하나를 방문했으며 이동 비용은 0인 상태로 시작합니다. count가 0이 아닌 1인 이유는 출발 도시도 방문한 도시이기 때문입니다.

도시 0에서 시작하고 정답 받기

8-4 Python 전체 풀이 (1/2)

```python
def tsp_min(dist):
    n = len(dist)
    used = [False]*n
    used[0] = True
    best = 10**9
    def search(last, count, total):
        nonlocal best
        if total >= best: return
        if count == n:
            best = min(best, total+dist[last][0])
            return
```

최소 왕복 비용을 정수 하나로 반환한다. 방문 순서를 반환할 필요는 없다.

8-4 Python 전체 풀이 (2/2)

```
12            for nxt in range(1, n):
13                if not used[nxt]:
14                    used[nxt] = True
15                    search(nxt, count+1, total+dist[last][nxt])
16                    used[nxt] = False
17        search(0, 1, 0)
18        return best
```

최소 왕복 비용을 정수 하나로 반환한다. 방문 순서를 반환할 필요는 없다.

C에서도 함수의 세 상태를 같은 순서로 읽습니다

last, count, total

last는 현재 서 있는 도시입니다.

count는 지금까지 방문한 도시 수입니다.

total은 지금까지 이동한 비용입니다.

best는 안쪽 호출의 반환값으로 갱신합니다.

8-4 C 1~2행 · 여행 상태를 보조 함수에 전달하기

```
01  static int tour_go(const int d[][9], int n, int last,
02                      int count, int used[], int total, int best) {
```

변수·상태	예제에서의 값
last	현재 도시
count	방문한 도시 수
total	누적 비용

여행 상태를 보조 함수에 전달하기

tour_go는 여행 경로를 이어 가는 보조 함수입니다. 비용 표 d와 도시 수 n, 현재 도시 last를 받습니다. static은 이 파일 안에서 쓰는 함수라는 표시입니다.

여행 상태를 보조 함수에 전달하기

8-4 C 3~4행 · 중단하거나 왕복을 완성하기

```
03      if (total>=best) return best;
04      if (count==n) {int v=total+d[last][0]; return v<best?v:best;}
```

변수·상태	예제에서의 값
완성 전 total	5
귀환 비용	4
v	9

중단하거나 왕복을 완성하기

현재 비용이 이미 best 이상이면 이 경로를 끝내고 best를 반환합니다. 더 이동해도 비용은 줄지 않습니다.

중단하거나 왕복을 완성하기

8-4 C 5~6행 · 다음 도시 선택하고 표시하기

```
05        for (int x=1; x<n; ++x) if (!used[x]) {
06            used[x]=1;
```

변수·상태	예제에서의 값
고른 x	1
used[x]	0에서 1로

다음 도시 선택하고 표시하기

도시 x를 1부터 n-1까지 보고 아직 방문하지 않은 도시만 고릅니다. 도시 0은 마지막 귀환에 남겨 둡니다.

다음 도시 선택하고 표시하기

8-4 C 7행 · 새 상태와 갱신된 best 받기

```
07          best=tour_go(d,n,x,count+1,used,total+d[last][x],best);
```

변수·상태	예제에서의 값
새 상태	last=1, count=2, total=2
best	반환받아 저장

새 상태와 갱신된 best 받기

현재 도시를 x로, 방문 수를 count+1로, 비용을 total+d[last][x]로 바꾸어 다음 호출에 전달합니다. 반환된 더 좋은 최소 비용을 best에 다시 저장합니다.

새 상태와 갱신된 best 받기

8-4 C 8~10행 · 방문을 복구하고 결과 반환하기

```
08            used[x]=0;
09        } return best;
10    }
```

변수·상태	예제에서의 값
used[x] 복구	1에서 0으로
결과	best 반환

방문을 복구하고 결과 반환하기

안쪽 호출이 끝나면 방문 표시를 0으로 되돌립니다. 다른 순서에서 도시 x를 다시 쓸 수 있게 합니다.

방문을 복구하고 결과 반환하기

8-4 C 11~14행 · 출발점에서 시작하기

```
11  int tsp_min(const int dist[][9], int n) {
12      int used[9]={0}; used[0]=1;
13      return tour_go(dist,n,0,1,used,0,1000000000);
14  }
```

변수·상태	예제에서의 값
used[0]	1
시작 상태	last=0, count=1, total=0

출발점에서 시작하기

사이트가 호출하는 tsp_min 함수를 정의합니다. 이동 비용 표와 실제 도시 수를 받습니다.

출발점에서 시작하기

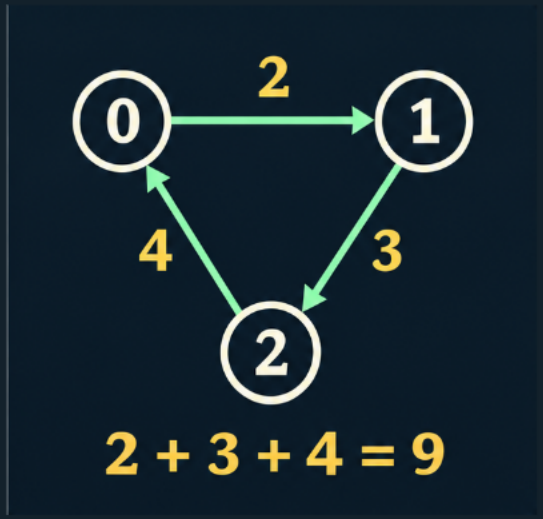
8-4 C 전체 풀이 (1/2)

```c
static int tour_go(const int d[][9], int n, int last,
                   int count, int used[], int total, int best) {
    if (total>=best) return best;
    if (count==n) {int v=total+d[last][0]; return v<best?v:best;}
    for (int x=1; x<n; ++x) if (!used[x]) {
        used[x]=1;
        best=tour_go(d,n,x,count+1,used,total+d[last][x],best);
        used[x]=0;
    } return best;
}
```

최소 왕복 비용을 정수 하나로 반환한다. 방문 순서를 반환할 필요는 없다.

8-4 C 전체 풀이 (2/2)

```c
int tsp_min(const int dist[][9], int n) {
    int used[9]={0}; used[0]=1;
    return tour_go(dist,n,0,1,used,0,1000000000);
}
```



현재 도시 0 · 방문 수 1 · 비용 0

찾은 최소 왕복 비용을 반환합니다.

최소 왕복 비용을 정수 하나로 반환한다. 방문 순서를 반환할 필요는 없다.

가능한 선택이 많으면 계산량도 커집니다

가지치기는 실제로 보는 갈래를 줄입니다.

배정 문제는 최악에 n!개의 완성 배정을 봅니다.

순회 문제는 출발점을 고정해도 (n-1)!개입니다.

작은 비용의 답을 먼저 찾으면 더 많은 갈래를 멈출 수 있습니다.

최악의 경우에는 여전히 많은 갈래가 남습니다.

다음에 어느 갈래를 볼지도 고를 수 있습니다

FIFO, LIFO, 최소 하한 우선

FIFO는 먼저 기다린 갈래부터 봅니다.

LIFO는 가장 최근 갈래부터 이어 봅니다.

최소 하한 우선은 비용을 가장 낮게 잡은 갈래부터 봅니다.

이번 기본 코드는 재귀 호출로 한 갈래를 깊게 봅니다.

하한을 더 강하게 만드는 다른 방법도 있습니다

도시별 최소 출발 비용을 한 번씩 더합니다.

모든 왕복 경로는 각 도시에서 한 번 출발합니다.

출발지마다 가장 싼 이동보다 싸게 떠날 수는 없습니다.

행의 최솟값을 더하면 왕복 비용의 하한을 얻습니다.

각 행의 최솟값을 빼 봅니다

출발 / 도착	0	1	2
0	∞	2	8
1	1	∞	3
2	4	6	∞

∞ 표시는 이번 계산에서 고를 수 없는 칸입니다.

각 행의 최솟값은 2, 1, 4

합은 2+1+4=7입니다.

각 행의 숫자에서 그 행의 최솟값을 뺍니다.

행에서 뺀 뒤 열도 확인합니다

출발 / 도착	0	1	2
0	∞	0	6
1	0	∞	2
2	0	2	∞

행에서 뺀 7에 열에서 뺄 2를 더해 하한 9를 얻습니다.

열 2의 최솟값은 2입니다.

열 0과 열 1의 최솟값은 0입니다.

모든 왕복은 각 도시에 한 번씩 도착합니다.

한 문장으로 현재 상태를 읽어 봅시다

search(1, 2)는 사람 1부터, 누적 비용 2입니다.

어떤 선택을 해 왔는지는 used에도 담겨 있습니다.

선택하면 표시하고 안쪽으로 들어갑니다.

돌아오면 표시를 지우고 다음 선택을 봅니다.

완성하면 best 갱신, 이길 수 없으면 바로 return