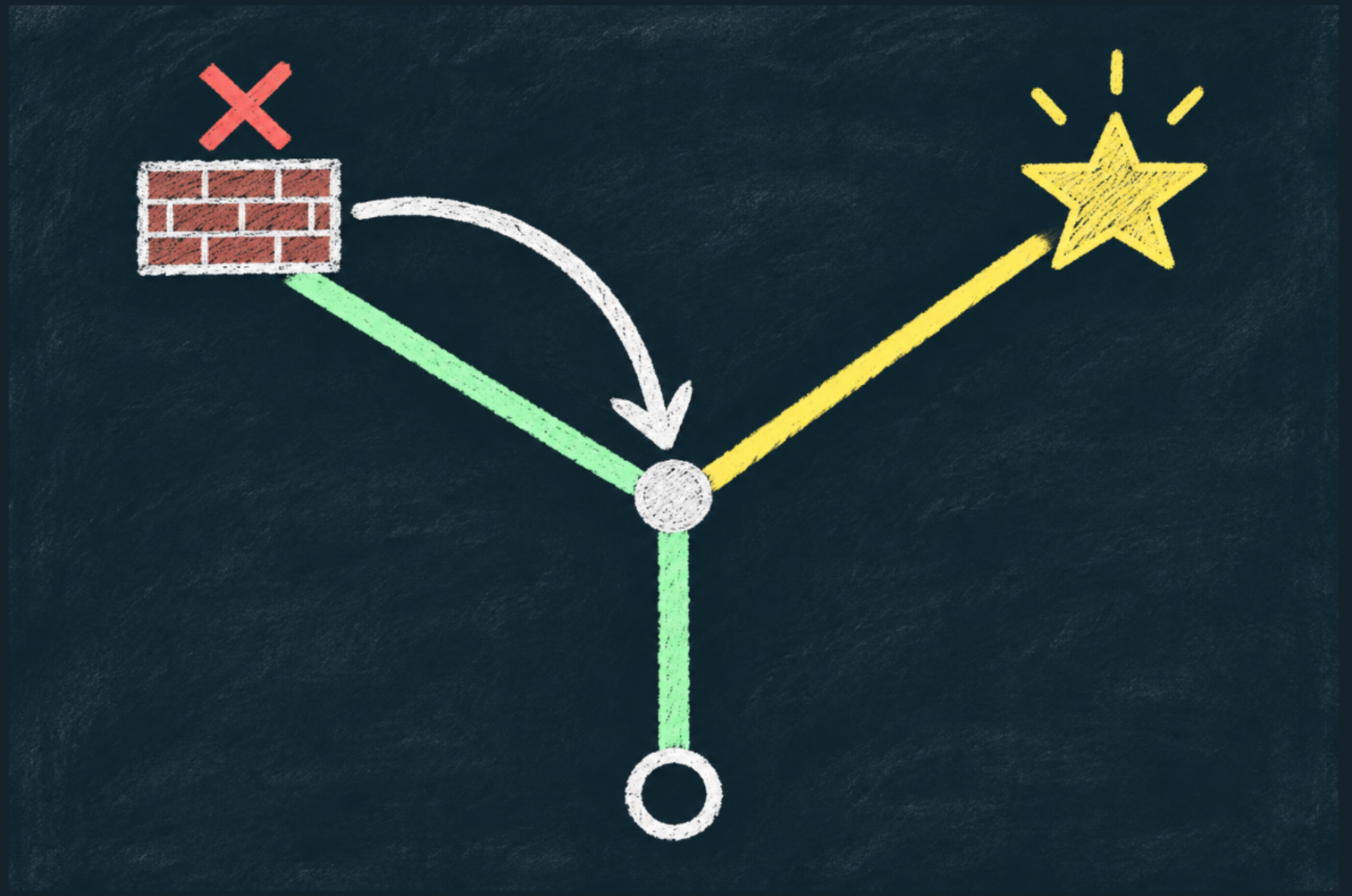
07

백트래킹

한 번 골라 보고, 돌아와서 다른 것을 골라 보기

막혔다면 마지막 선택으로 돌아옵니다



왼쪽 길을 골랐는데 막혀 있습니다.

갈림길로 돌아와 오른쪽 길을 가봅니다.

문제에서도 앞의 선택을 바꾸어 볼 수 있습니다.

이렇게 되돌아가며 찾는 방법이 백트래킹입니다.

도입 문제: 두 숫자를 문자로 바꾸기

전화기의 숫자 2에는 알파벳 a, b, c가, 숫자 5에는 j, k, l이 적혀 있다. 입력으로 두 자리 문자열 "25"가 주어진다.

2에서 첫 글자 하나를 고르고 5에서 둘째 글자 하나를 골라, 그 순서대로 이어 붙인다. 각 자리에서 반드시 한 글자씩 골라야 한다.

이 규칙으로 만들 수 있는 두 글자 문자열을 모두 구하시오.

예: "aj"는 가능하다. "ja"는 숫자 순서를 바꾸었으므로 불가능하다.

첫 자리는 2에서, 둘째 자리는 5에서



빈칸은 두 개입니다.

첫 칸: a, b, c 중 하나

둘째 칸: j, k, l 중 하나

어느 글자를 골랐는지 한 칸씩 기억합니다.

먼저 a로 시작하는 답을 만듭니다



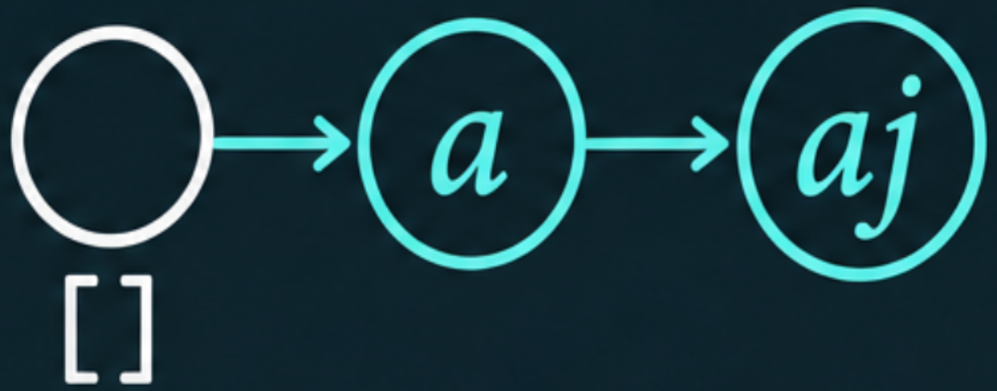
첫 글자로 a를 골랐습니다.

첫 칸의 a는 그대로 둡니다.

이제 둘째 칸에 넣을 j, k, l을 봅니다.

현재 만든 글자는 a입니다.

j를 붙이면 답 하나가 완성됩니다



a 뒤에 j를 붙였습니다.

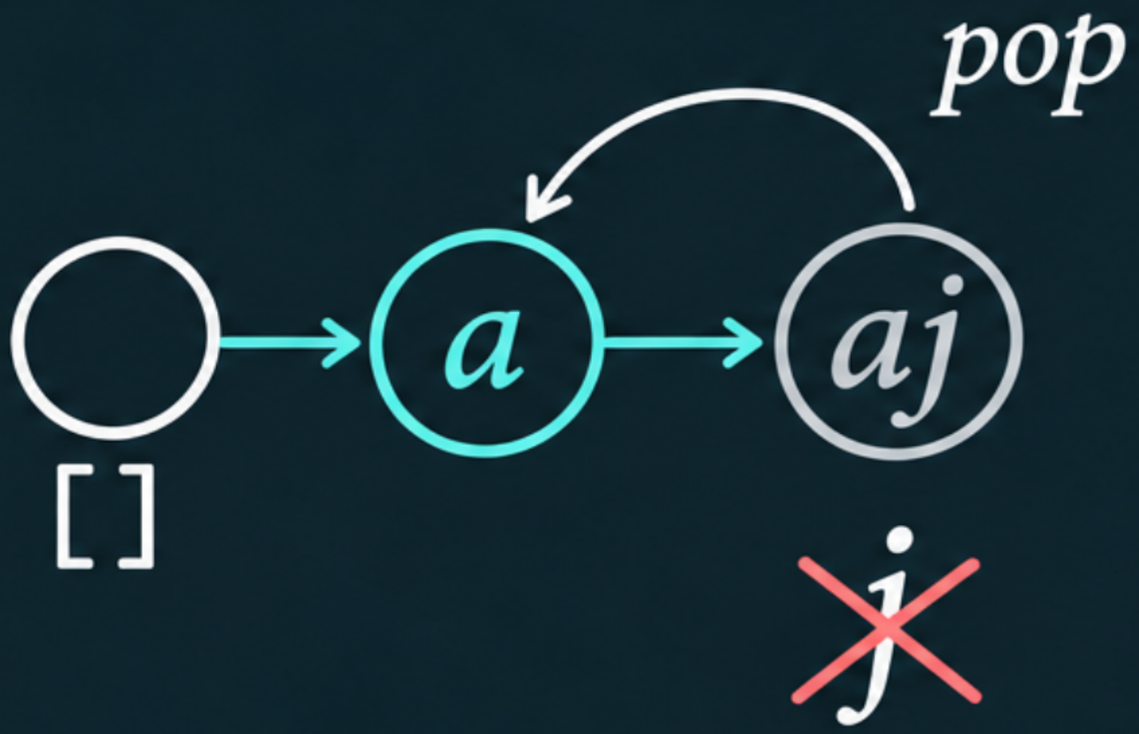
두 칸을 모두 채웠습니다.

완성한 aj를 답 목록에 적습니다.

aj

찾은 답: aj

마지막에 붙인 j만 지웁니다



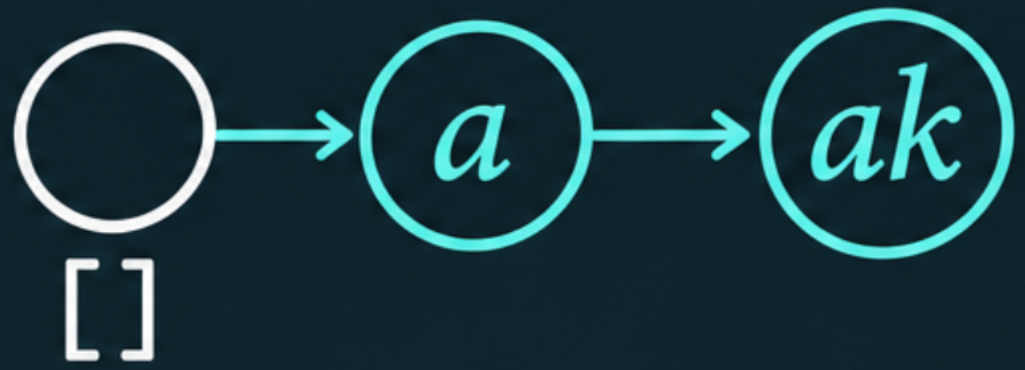
aj에서 j를 지웠습니다.

현재 글자는 다시 a입니다.

답 목록에 적은 aj는 지우지 않습니다.

둘째 칸을 고르기 전으로 돌아왔습니다.

그 자리에 이번에는 k를 넣습니다



aj ak

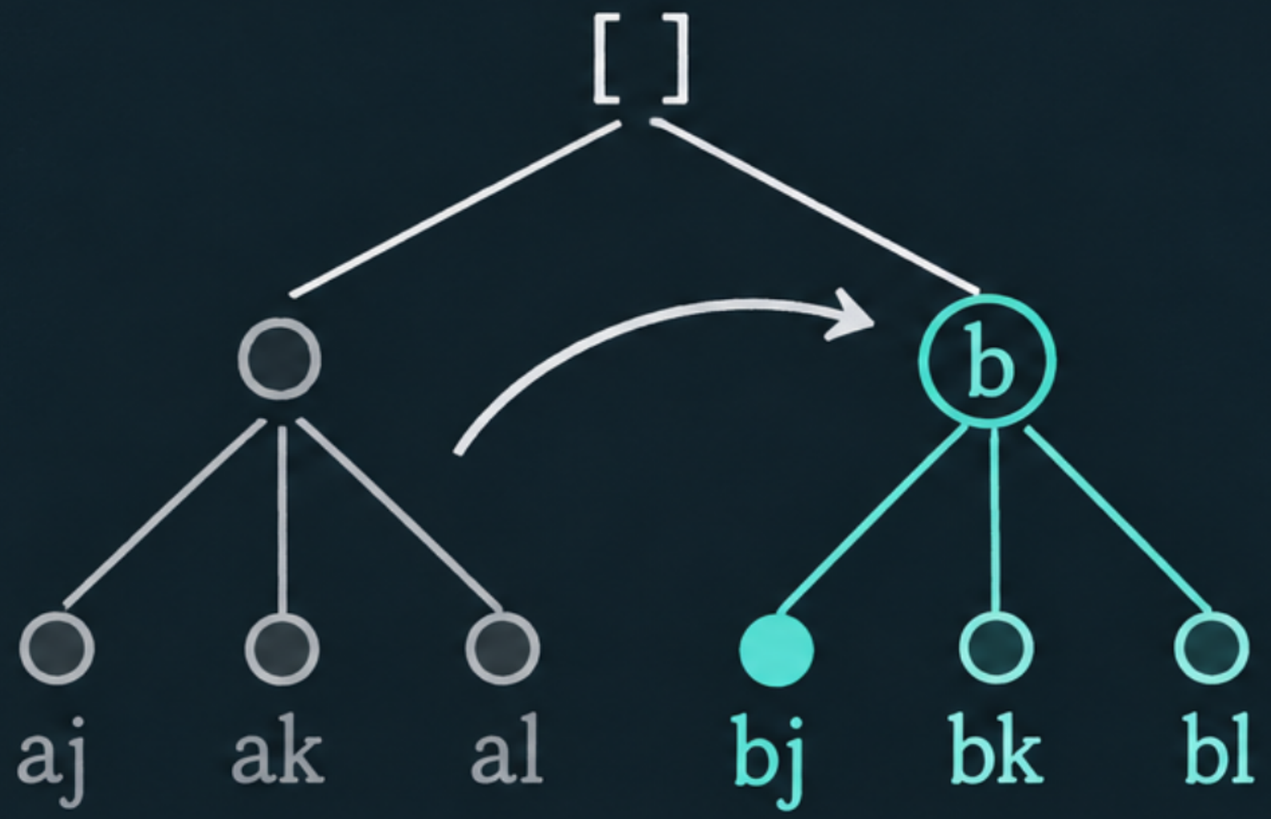
a 뒤에 k를 붙였습니다.

ak도 완성된 답입니다.

다시 k를 지우고 l을 붙일 수 있습니다.

a로 시작하는 답: aj, ak, al

a로 시작하는 답을 다 봤으면 b로 바꿉니다



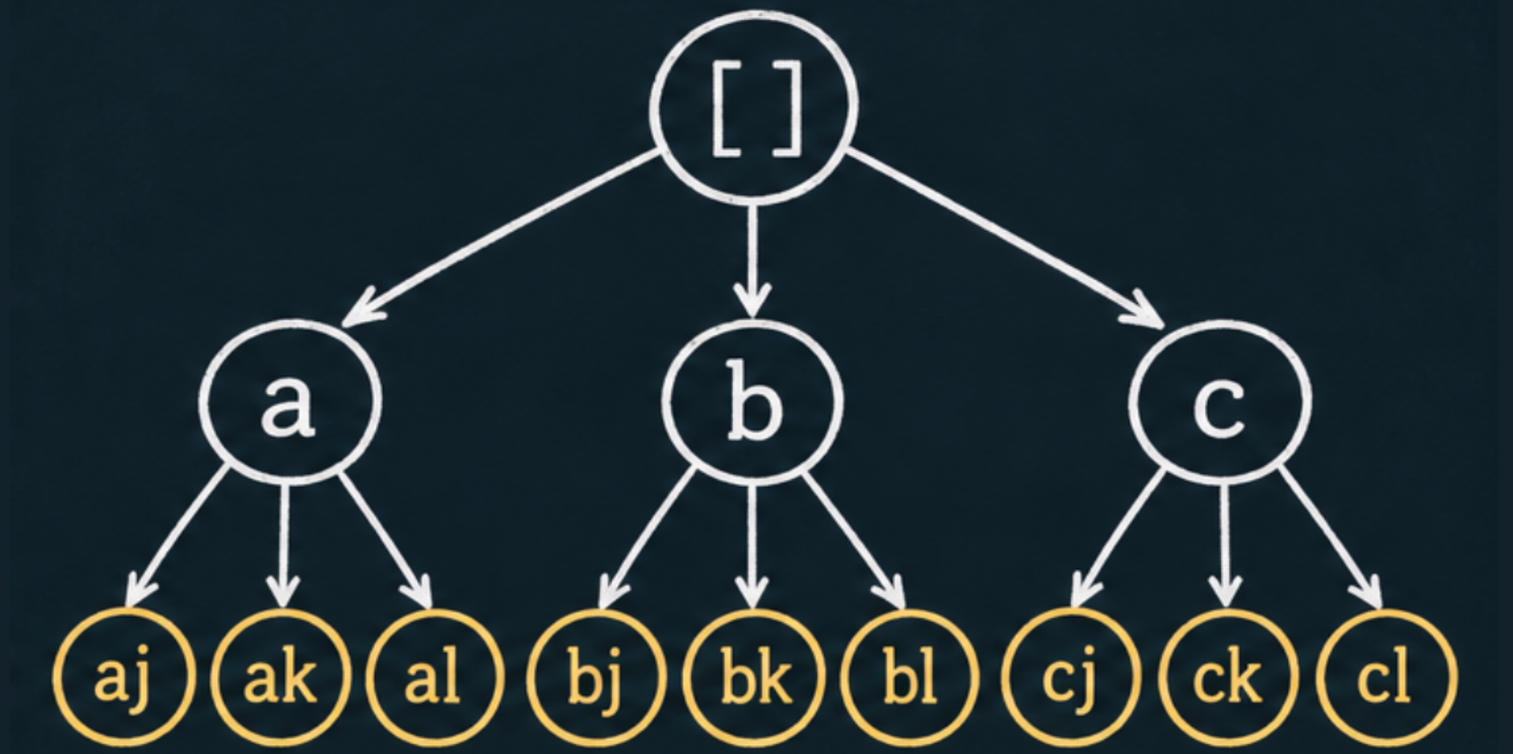
이제 첫 글자 a를 지웁니다.

첫 자리에 b를 놓습니다.

다시 둘째 자리 j, k, l을 고릅니다.

bj, bk, bl도 같은 방식으로 만듭니다.

가능한 답을 빠짐없이 찾았습니다



첫 글자마다 답 세 개가 있습니다.

a: aj, ak, al / b: bj, bk, bl

c: cj, ck, cl

총 아홉 문자열입니다.

코드에서 실패한 색을 지우는 위치

color[v] = c: 지금 점에 후보 색을 칠합니다.

paint(v+1): 다음 점부터 끝까지 칠할 수 있는지 확인합니다.

성공하면 1을 반환합니다. 실패하면 color[v] = 0으로 지우고 다음 색을 봅니다.

두 색이 모두 실패하면 앞의 점으로 돌아갑니다.

C 함수에서 n은 실제 점 수입니다

graph는 연결 표, n은 점 수, k는 쓸 수 있는 색 수입니다.

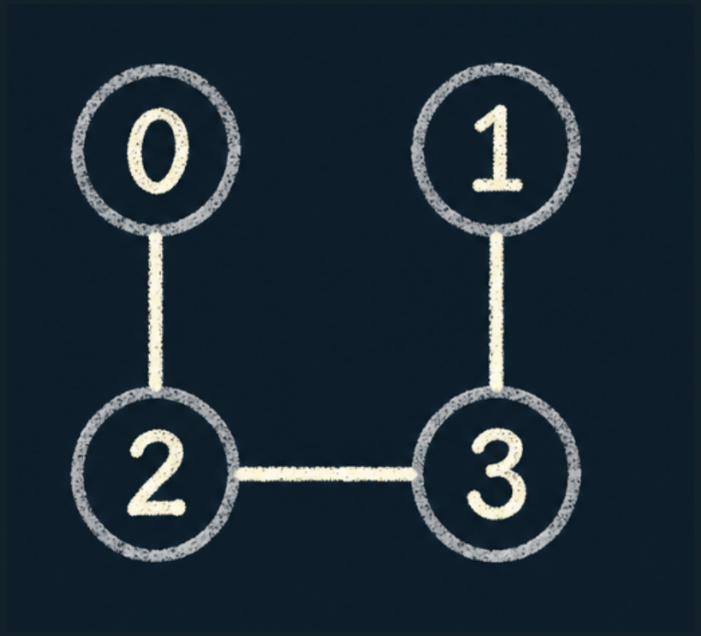
graph[][8]은 한 행의 저장 공간이 8칸이라는 C의 표시입니다. 앞의 n행·n열만 읽습니다.

color[8]={0}은 색을 저장할 공간 8칸을 모두 0으로 준비합니다.

입력 준비와 main 함수는 사이트가 맡습니다. 정해진 함수와 보조 함수를 작성합니다.

7-3 C ① 색 배열을 준비하고 첫 점을 맡깁니다

```
01  static int paint(const int g[][8], int n, int k,
02                   int v, int color[]) {
13  int can_color(const int graph[][8], int n, int k) {
14      int color[8]={0}; return paint(graph,n,k,0,color);
```



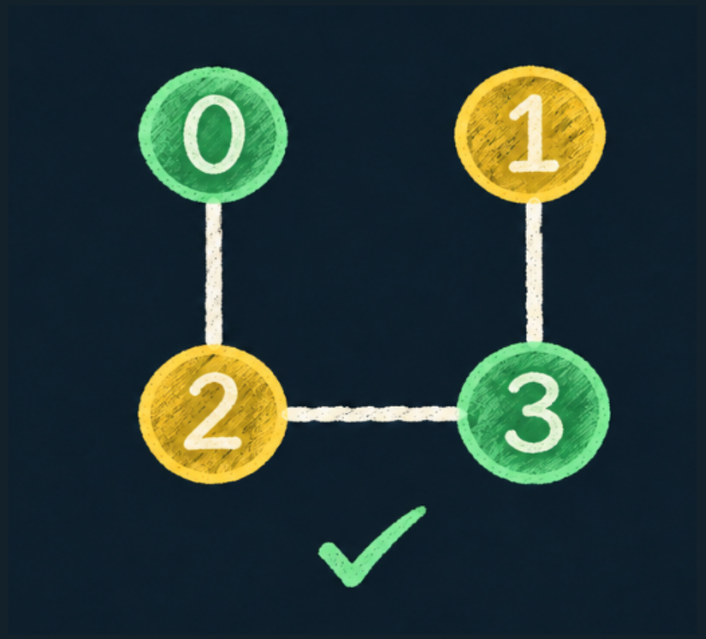
color[8]={0}: 여덟 칸을 모두 0으로 준비

실제 점은 앞 n개만 사용

paint에 v=0을 넘겨 0번부터 시작합니다.

7-3 C ② 완료 여부와 후보 색을 준비합니다

```
03        if (v==n) return 1;
04        for (int c=1; c<=k; ++c) {
05            int ok=1;
```



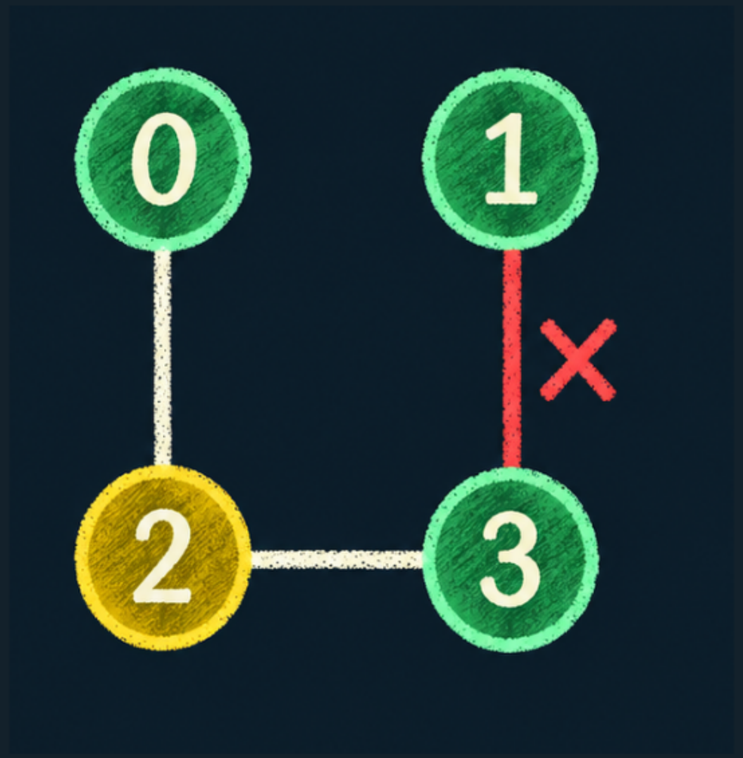
v==n이면 모든 점을 칠했으므로 1

그 전에는 c를 1부터 k까지 시험

++c는 c를 1씩 늘립니다. ok=1은 검사 전의 허용 표시입니다.

7-3 C ③ 두 조건이 모두 맞으면 충돌입니다

```
06          for (int u=0; u<v; ++u)
07              if (g[v][u] && color[u]==c) ok=0;
```



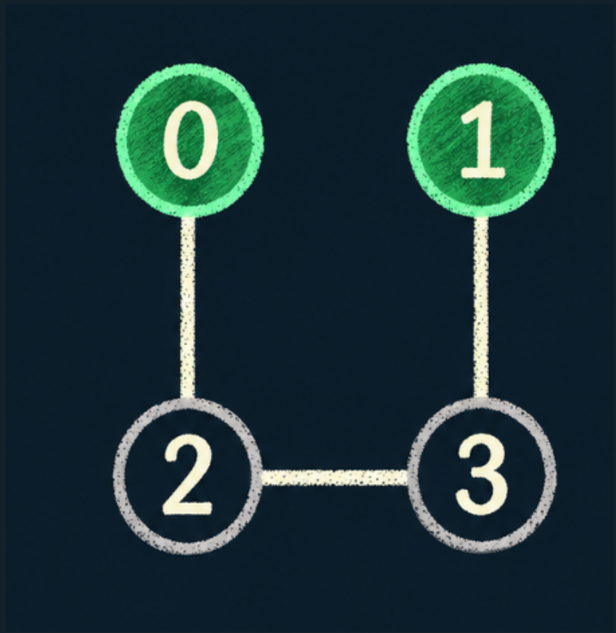
g[v][u]가 1이면 두 점이 연결됨

color[u]==c이면 같은 색

&&는 두 조건이 모두 맞는지 검사합니다.

7-3 C ④ 칠하기·성공·되돌리기를 연결합니다

```
08            if (ok) { color[v]=c;
09                if (paint(g,n,k,v+1,color)) return 1;
10                color[v]=0; }
11        } return 0;
```



ok가 1이면 색을 기록하고 다음 점을 맡김

실패하면 색을 0으로 되돌리고 다음 후보

모든 후보가 실패하면 0을 돌려주어 앞의 점이 선택을 바꿉니다.

7-3 C (앞부분)

```
1   static int paint(const int g[][8], int n, int k,
2                    int v, int color[]) {
3       if (v==n) return 1;
4       for (int c=1; c<=k; ++c) {
5           int ok=1;
6           for (int u=0; u<v; ++u)
7               if (g[v][u] && color[u]==c) ok=0;
```

이어서 다음 화면의 코드를 붙여 같은 함수를 완성합니다.

7-3 C (뒷부분)

```
8             if (ok) { color[v]=c;
9                 if (paint(g,n,k,v+1,color)) return 1;
10                color[v]=0; }
11        } return 0;
12    }
13    int can_color(const int graph[][8], int n, int k) {
14        int color[8]={0}; return paint(graph,n,k,0,color);
15    }
```

앞 화면에서 검사한 뒤, 색을 적고 다음 점으로 갑니다.

두 문제는 같은 순서로 찾고, 답을 다르게 모읍니다

7-2는 가능한 선택의 수를 묻습니다. 각 갈래에서 찾은 성공 수를 모두 더합니다.

7-3은 가능한지 묻습니다. 모든 점을 칠한 방법 하나를 찾으면 1을 반환합니다.

두 문제 모두 하나를 시험하고, 필요하면 앞의 선택으로 돌아와 다른 것을 봅니다.

문제에서 구하라는 답을 먼저 읽고, 마지막에 무엇을 반환할지 정합니다.

공통 동작은 고르기와 되돌리기입니다

첫 자리에 a를 놓고, 둘째 자리의 j·k·l을 차례로 넣어 봅니다.

a에서 만들 수 있는 세 답을 다 적으면 첫 글자를 b로 바꿉니다.

답 목록은 남겨 두고, 지금 시험 중인 글자만 지웠다가 바꿉니다.

하나를 고르기 → 뒤를 다 확인하기 → 돌아와 바꾸기

7-2: 합이 3이 되도록 숫자 고르기



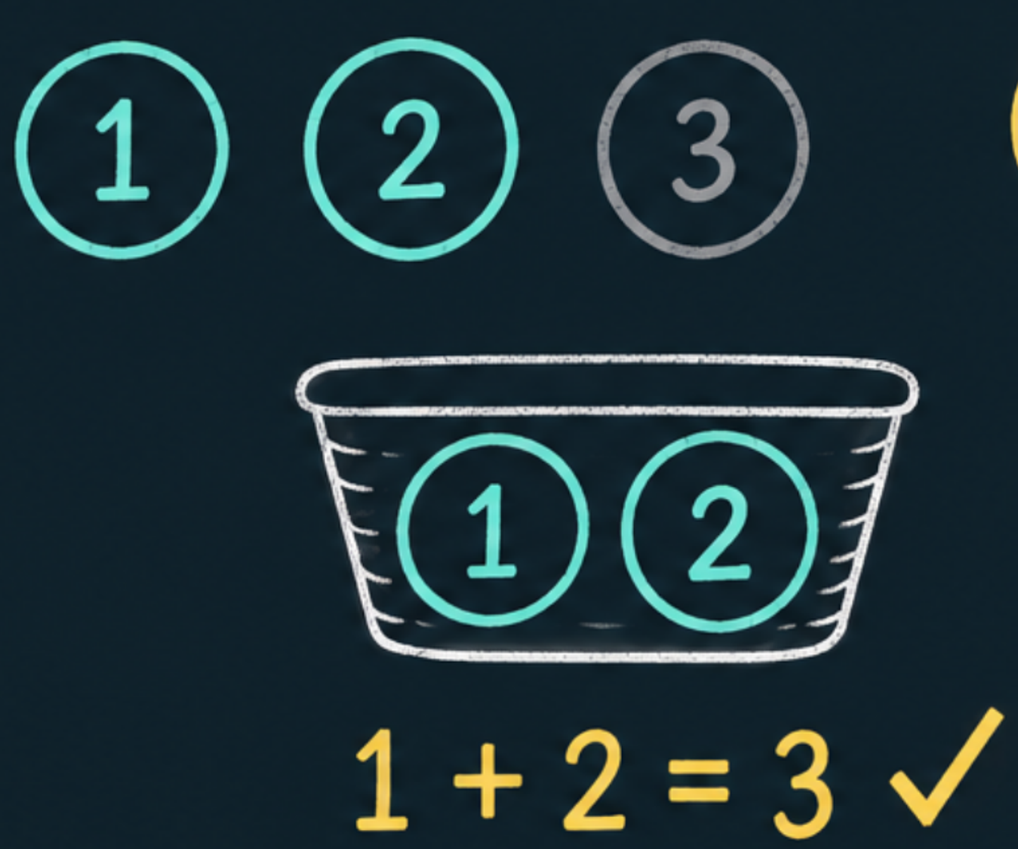
숫자 1, 2, 3이 하나씩 있습니다.

각 숫자를 고르거나 건너뛸 수 있습니다. 같은 숫자를 다시 쓰지는 않습니다.

고른 숫자의 합이 정확히 3이어야 합니다.

이렇게 고르는 방법은 모두 몇 가지일까요?

방법 ①: 1과 2를 함께 고르기



바구니에 1과 2를 넣습니다.

1 + 2 = 3입니다.

조건에 맞는 방법을 하나 찾았습니다.

지금까지 찾은 방법 수: 1

방법 ②: 3 하나만 고르기



이번에는 3만 넣습니다.

1과 2는 고르지 않았습니다.

바구니 안 숫자의 합은 역시 3입니다.

지금까지 찾은 방법 수: 2

문제가 요구하는 답은 2입니다

두 바구니 모두 합이 3입니다.





방법 ① 1 + 2 = 3

방법 ② 3 = 3

첫 바구니와 둘째 바구니는 서로 다른 고르기입니다.

조건에 맞는 서로 다른 고르기는 모두 2가지입니다.

확인하는 순서를 정합니다

먼저 1, 다음은 2, 마지막은 3을 봅니다.

각 숫자에서 먼저 건너뛴 경우를 보고, 그다음 고른 경우를 봅니다.

세 숫자의 결정을 마치면 합을 계산합니다. 합이 3인 경우만 셉니다.

이제 가능한 고르기 8가지를 빠짐없이 확인합니다.

먼저 1을 건너뜁니다



1을 넣지 않고 다음 숫자 2로 갑니다.

현재 바구니는 비어 있습니다.

2와 3을 어떻게 할지는 아직 정하지 않았습니다.

현재 합 0 / 다음에 볼 숫자 2

이어서 2도 건너뜁니다



이번에는 2를 넣지 않습니다.

1과 2를 모두 건너뛴 상태입니다.

마지막 숫자 3의 결정이 남아 있습니다.

현재 합 0 / 다음에 볼 숫자 3

① 아무것도 고르지 않은 경우



1 건너뜀 / 2 건너뜀 / 3 건너뜀

바구니가 비어 있으므로 합은 0입니다.

이번 선택에서 찾은 성공: 0가지

찾은 성공: 0가지

② 마지막 숫자 3을 고른 경우



1 건너뜀 / 2 건너뜀 / 3 고름

이번에는 3 하나가 있으므로 합은 3입니다.

이번 선택에서 찾은 성공: 1가지

찾은 성공: 1가지

3을 되돌려놓고 2로 돌아갑니다



3의 두 경우는 모두 확인했습니다.

3을 건너뛴 경우와 고른 경우를 다 보았습니다.

이제 3을 빼고, 앞에서 건너뛴 2를 골라 봅니다.

성공 기록 1은 그대로 남겨 둡니다.

③ 2를 고르고 3을 건너뛴 경우



1 건너뜀 / 2 고름 / 3 건너뜀

3을 빼고 2를 넣었으므로 합은 2입니다.

이번 선택에서 찾은 성공: 0가지

찾은 성공: 1가지

④ 2와 3을 함께 고른 경우



1 건너뜀 / 2 고름 / 3 고름

2 + 3 = 5이므로 목표 3과 다릅니다.

이번 선택에서 찾은 성공: 0가지

찾은 성공: 1가지

먼저 마지막에 넣은 3을 뺍니다



방금까지는 2와 3이 들어 있었습니다.

마지막에 넣은 3을 빼니 2만 남습니다.

2를 고른 상태에서 3의 두 경우를 다 보았습니다.

이제 2를 고르기 전으로 돌아갑니다.

2도 되돌려놓고 1로 돌아갑니다



1을 건너뛴 네 경우를 다 보았습니다.

2를 꺼내면 바구니가 다시 비어 있습니다.

이제 첫 숫자 1을 고르는 쪽을 확인합니다.

성공 기록은 여전히 1입니다.

⑤ 1을 고르고 2와 3을 건너뛴 경우



1 고름 / 2 건너뜀 / 3 건너뜀

바구니에 1만 있으므로 합은 1입니다.

이번 선택에서 찾은 성공: 0가지

찾은 성공: 1가지

⑥ 1과 3을 함께 고른 경우



1 고름 / 2 건너뜀 / 3 고름

1 + 3 = 4이므로 목표 3과 다릅니다.

이번 선택에서 찾은 성공: 0가지

찾은 성공: 1가지

3만 빼고 1은 그대로 둡니다



1을 고른 상태로 돌아왔습니다.

2를 건너뛴 아래의 두 경우를 모두 봤습니다.

이번에는 1을 둔 채 2를 골라 봅니다.

앞에서 고른 1을 유지하는 것이 중요합니다.

⑦ 1과 2를 함께 고른 경우



1 고름 / 2 고름 / 3 건너뜀

1 + 2 = 3이므로 성공입니다.

이번 선택에서 찾은 성공:
1가지

찾은 성공: 2가지

⑧ 세 숫자를 모두 고른 경우



1 고름 / 2 고름 / 3 고름

1 + 2 + 3 = 6이므로 목표 3과 다릅니다.

이번 선택에서 찾은 성공: 0가지

찾은 성공: 2가지

함수는 값을 받아 계산한 답을 돌려줍니다

```
def add(x, y):
    return x + y

answer = add(2, 3)
```

def는 이 이름의 함수를 만드는 줄입니다.

add(2,3)을 실행하면 x는 2, y는 3을 받습니다.

2+3을 계산한 5가 돌아와 answer에 들어갑니다.

함수를 만들기 → 값을 넣어 실행하기 → 결과 받기

search: 남은 숫자로 성공할 방법 세기

함수는 정해진 일을 실행하고, 그 결과를 알려주는 코드입니다.

여기서는 "남은 숫자를 골라 목표 합을 만드는 방법 세기"를 맡깁니다.

그 함수에 search라는 이름을 붙입니다. 답은 성공한 방법의 개수입니다.

이 예제에서는 목표 합을 3으로 정해 놓습니다.

3 하나만 남았고, 현재 합이 0이라면

1과 2는 건너뛰었습니다. 이제 3만 결정하면 됩니다.



3을 안 고름 → 최종 합 0 → 실패



3을 고름 → 최종 합 3 → 성공

두 경우를 확인했더니 성공한 방법은 한 가지입니다.

이 일을 맡은 search는 결과로 1을 돌려줍니다.

같은 3이 남아도, 현재 합이 2라면

이번에는 앞에서 2를 골랐습니다. 남은 숫자는 여전히 3입니다.



3을 안 고름 → 최종 합 2 → 실패



3을 고름 → 최종 합 5 → 실패

3을 안 골라도, 골라도 목표 합 3을 만들 수 없습니다.

이번 search의 결과는 성공 0가지입니다.

일을 맡기려면 두 가지를 알려줍니다

① 다음에 어디부터 볼까요? → 아직 결정하지 않은 숫자의 위치

② 지금까지 얼마를 골랐나요? → 이미 고른 숫자들의 합

이 두 정보를 받으면, 남은 숫자를 확인해서 성공 횟수를 셀 수 있습니다.

목표 합 3은 이 예제 전체에서 그대로 사용합니다.

숫자 3은 배열의 2번 칸에 있습니다

```
a[0] = 1
a[1] = 2
a[2] = 3
```

배열의 칸 번호는 0부터 시작합니다.

첫 칸은 0번, 둘째는 1번, 셋째는 2번입니다.

따라서 3부터 볼 때 알려줄 위치 번호는 2입니다.

a[2]는 "2번 칸에 들어 있는 값"이고, 그 값은 3입니다.

search(2, 0)을 그림으로 읽기



search(2, 0)

목표 합 = 3

지금까지 고른 숫자 없음
→ 현재 합 0

남은 숫자: 3

3을 안 고름

0 + 0 = 0

✗ 목표 3과 다름 → 실패

3을 고름

0 + 3 = 3

✓ 목표 3과 같음 → 성공

성공은 1가지 → 함수의 답은 1

search(2, 2)을 그림으로 읽기



search(2, 2)

목표 합 = 3

이미 고른 숫자: 2
→ 현재 합 2

남은 숫자: 3

3을 안 고름

2 + 0 = 2

✕ 목표 3과 다름 → 실패

3을 고름

2 + 3 = 5

✕ 목표 3과 다름 → 실패

성공은 0가지 → 함수의 답은 0

search(2, 3)을 그림으로 읽기

search(2, 3)

0번 칸 | 1번 칸 | 2번 칸

1 ✓ | 2 ✓ | 3

1과 2는 이미 고른 상태

이제 3을 볼 차례

목표 합 = 3

이미 고른 숫자: 1, 2
→ 현재 합 3

남은 숫자: 3

3을 안 고름

3 + 0 = 3

✓ 목표 3과 같음 → 성공

3을 고름

3 + 3 = 6

✗ 목표 3과 다름 → 실패

성공은 1가지 → 함수의 답은 1

더 볼 숫자가 없으면, 지금 합으로 판정합니다

처음 숫자: 1, 2, 3 | 목표 합: 3

1과 2를 고르고, 3은 건너뛰었음

1 ✓ 2 ✓ 3 ×

세 숫자 모두 결정 완료

더 볼 숫자: 없음

현재 합 = 1 + 2 = 3

✓ 목표 3과 같음 → 성공 1가지

1과 2와 3을 모두 골랐음

1 ✓ 2 ✓ 3 ✓

세 숫자 모두 결정 완료

더 볼 숫자: 없음

현재 합 = 1 + 2 + 3 = 6

✗ 목표 3과 다름 → 성공 0가지

현재 합은 이미 고른 숫자를 더한 값입니다.

숫자 2에서 맡을 일도 둘로 나눕니다

```
search(1, 0)
```

```
2를 건너뜀 → search(2, 0) → 1가지
2를 고름   → search(2, 2) → 0가지
```

1은 이미 건너뛰었습니다. 지금 1번 칸의 숫자 2를 봅니다.

2를 안 고르면 현재 합 0인 채 3부터 봅니다.

2를 고르면 현재 합 2인 채 3부터 봅니다.

search(1, 0)의 답은 1 + 0 = 1가지입니다.

처음에는 아무것도 고르지 않고 시작합니다

search(0, 0): 첫 칸의 숫자 1부터 보고, 현재 합은 0입니다.

1을 건너뛴 쪽의 성공: 3만 고르는 한 가지입니다.

1을 고른 쪽의 성공: 1과 2를 고르는 한 가지입니다.

처음 실행의 결과는 1 + 1 = 2가지입니다.

문제 7-2: 목표 합을 만드는 선택의 수

양의 정수 배열 a와 목표 합 target이 주어진다. a의 원소를 몇 개 골라 그 합을 target으로 만들려고 한다.

각 위치의 원소는 한 번만 고를 수 있다. 고른 순서는 구분하지 않지만, 같은 값이라도 서로 다른 위치의 원소를 골랐다면 다른 선택으로 센다. 아무것도 고르지 않은 선택도 포함하며 그 합은 0이다.

배열의 길이는 0~20, 각 원소는 1~50, target은 0~1000의 정수이다.

고른 원소의 합이 target인 선택의 수를 정수로 반환하는 subset_sum_count 함수를 작성하시오.

예: a=[1,2,3], target=3이면 [1,2]와 [3]의 두 선택이 있으므로 2를 반환한다.

Python: subset_sum_count(a, target) / C: subset_sum_count(a, n, target), n은 배열 길이

코드를 읽는 데 필요한 세 가지 표기

```
len([1, 2, 3])  → 3
3 == 3          → True
int(True)       → 1
```

len은 배열에 들어 있는 값의 개수를 셉니다.

==는 양쪽 값이 같은지 묻습니다.

int는 True를 1, False를 0으로 바꿉니다.

return int(total == target): 합이 맞으면 1, 다르면 0

계산을 시작하는 곳은 마지막 줄입니다

```
subset_sum_count([1,2,3], 3)
  a = [1,2,3]
  target = 3

  return search(0, 0)
```

사이트가 배열과 목표를 넣어 함수를 실행합니다.

함수 안에 search의 할 일을 정의합니다.

마지막 줄에서 첫 칸, 현재 합 0으로 실제 계산을 시작합니다.

search는 같은 배열과 목표를 읽고 위치와 합을 바꾸며 실행합니다.

7-2 ① 두 함수가 받는 값을 나눕니다

```
01  def subset_sum_count(a, target):
02      def search(i, total):
```



바깥 함수: 배열 a와 목표 target

안쪽 search: 다음 위치 i와 현재 합 total

예제의 배열은 [1,2,3], 목표 합은 3입니다.

7-2 ② 맨 마지막 줄에서 계산을 시작합니다

```
09      return search(0, 0)
```



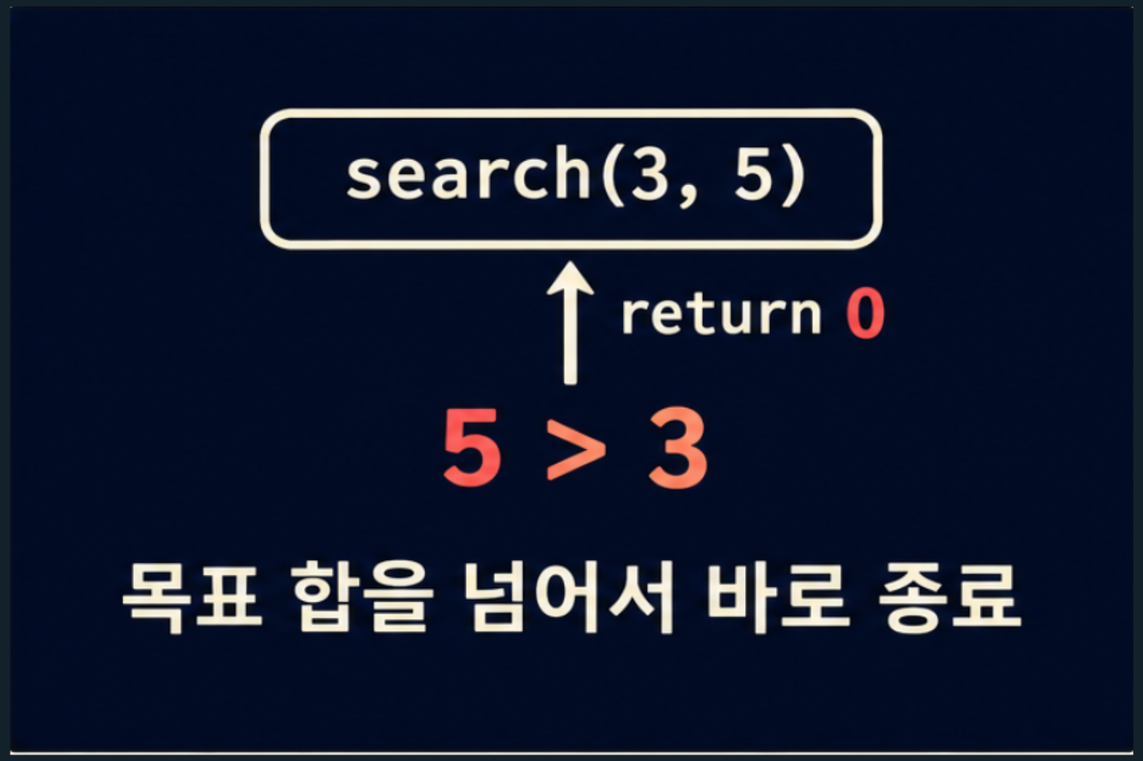
i=0: 첫 칸의 숫자 1부터 확인

total=0: 아직 고른 숫자가 없음

search가 일을 마쳐 알려준 답을 바깥 함수가 돌려줍니다.

7-2 ③ 합이 목표보다 크면 여기서 끝냅니다

```
03          if total > target:
04              return 0
```



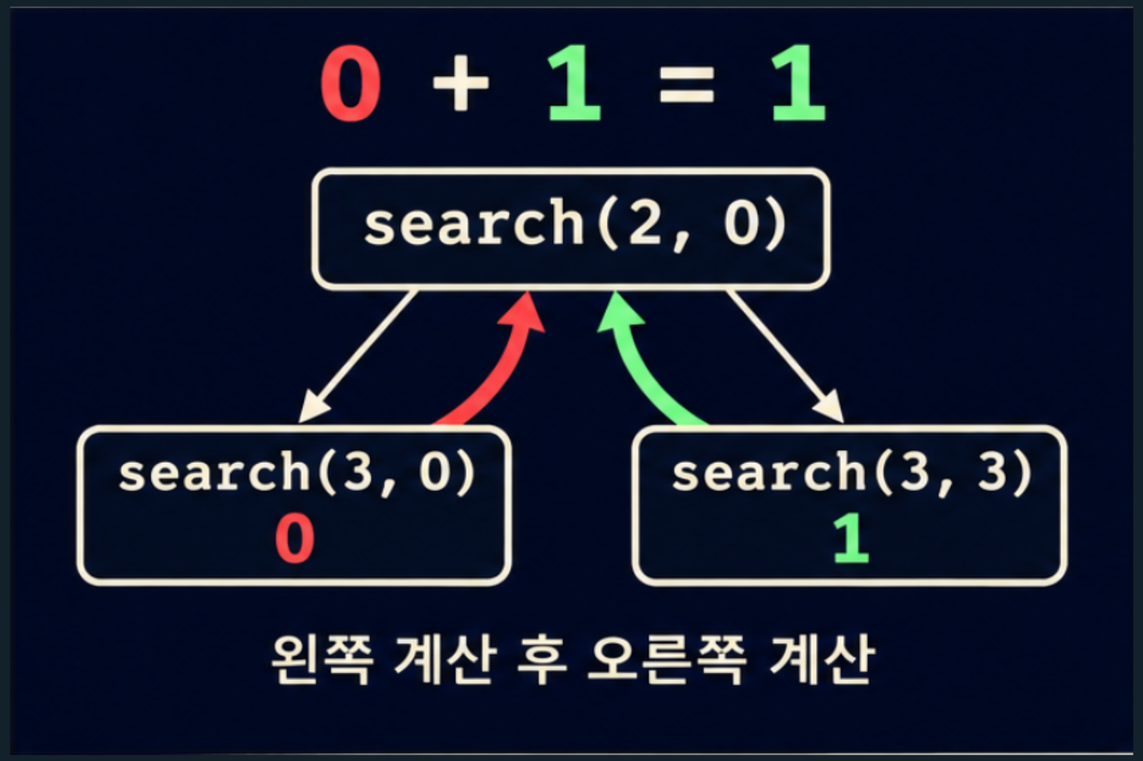
이 실행은 total=5, target=3

5>3이므로 return 0에서 종료

모든 입력 숫자가 양수여서, 더 골라도 합을 3으로 줄일 수 없습니다.

7-2 ④ 마지막 칸까지 봤다면 합을 비교합니다

```
05          if i == len(a):
06              return int(total == target)
```



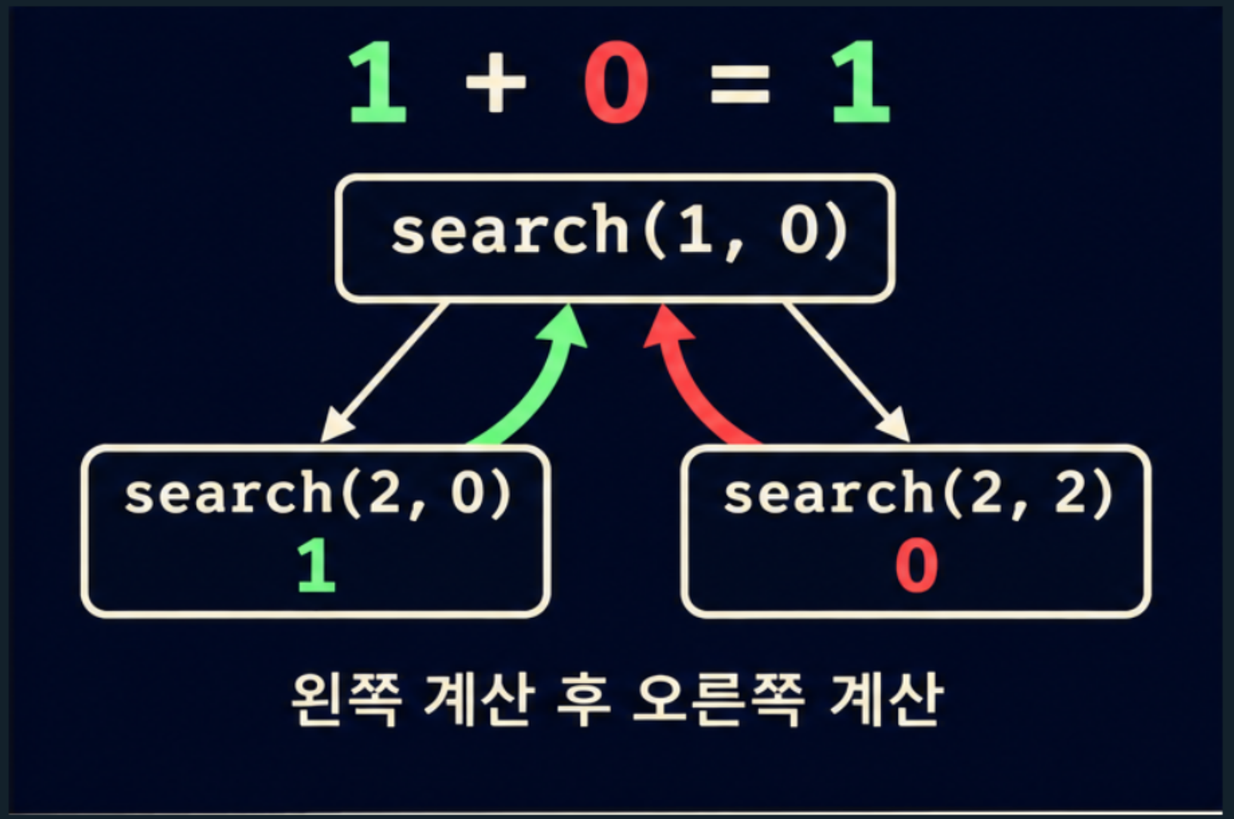
len(a)=3, 다음 위치 i=3 → 결정 완료

합 0이면 0, 합 3이면 1을 돌려줌

int(total == target)는 성공 여부를 1 또는 0으로 바꿉니다.

7-2 ⑤ 지금 숫자를 건너뛴 쪽을 먼저 계산합니다

```
07          return (search(i+1, total)
```



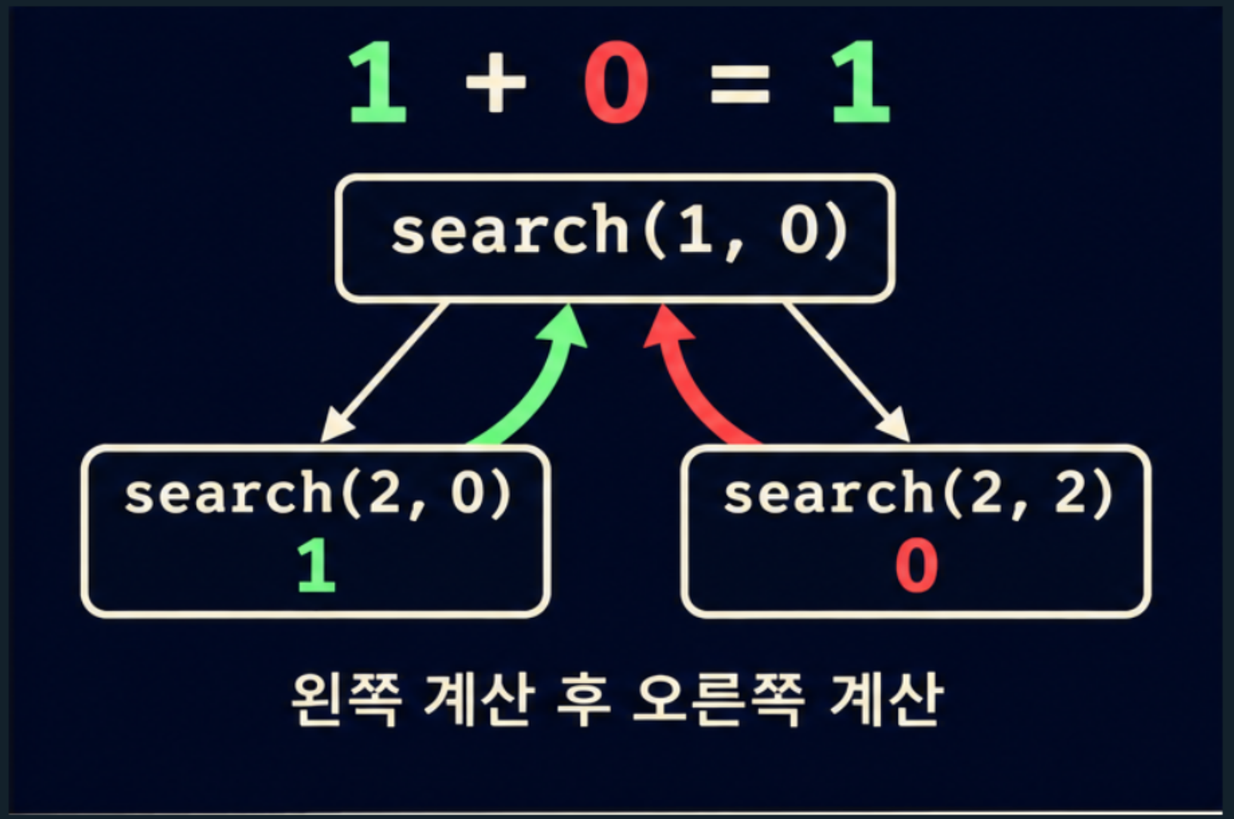
지금 i=1이면 a[i]는 숫자 2

i+1은 2, total은 0 그대로

search(2,0): 2를 넣지 않고, 다음 숫자 3부터 확인합니다.

7-2 ⑥ 지금 숫자를 고른 쪽은 합을 더합니다

```
08                      + search(i+1, total+a[i]))
```



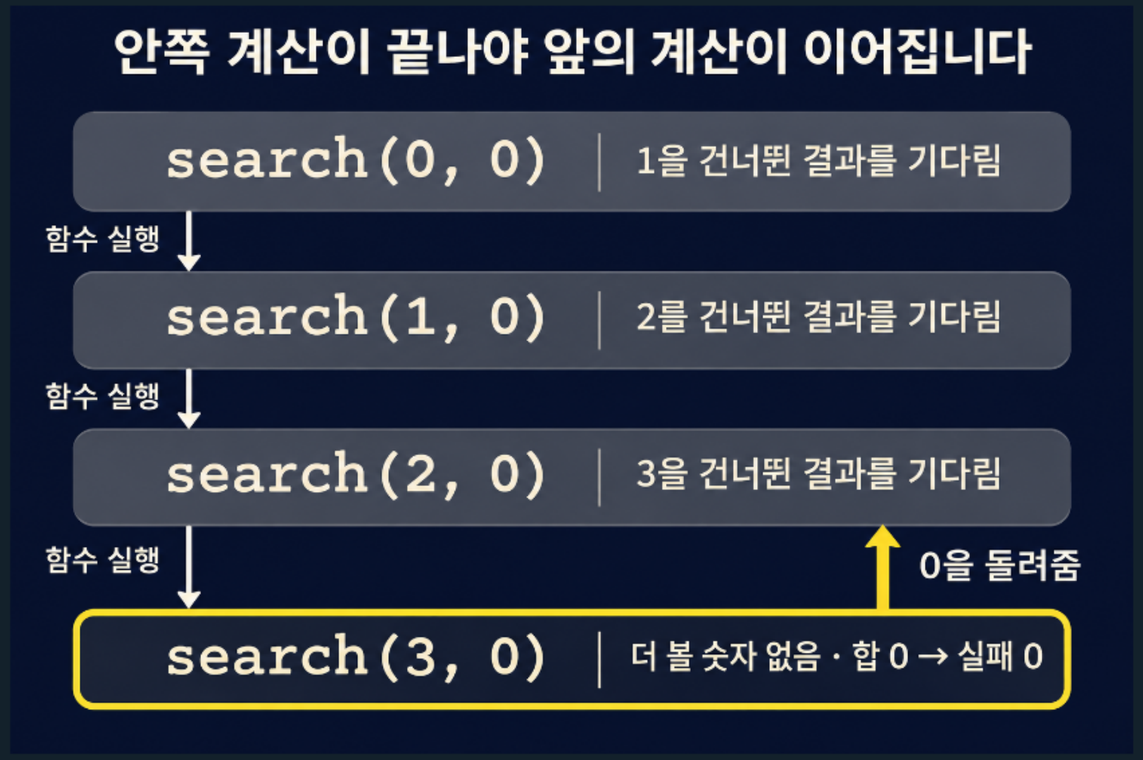
지금 i=1, total=0, a[i]=2

total+a[i]는 0+2=2

search(2,2): 2를 넣고, 현재 합 2로 다음 숫자 3을 봅니다.

7-2 ⑦ 안쪽 계산이 끝날 때까지 기다립니다

```
07          return (search(i+1, total)
08                  + search(i+1, total+a[i]))
```



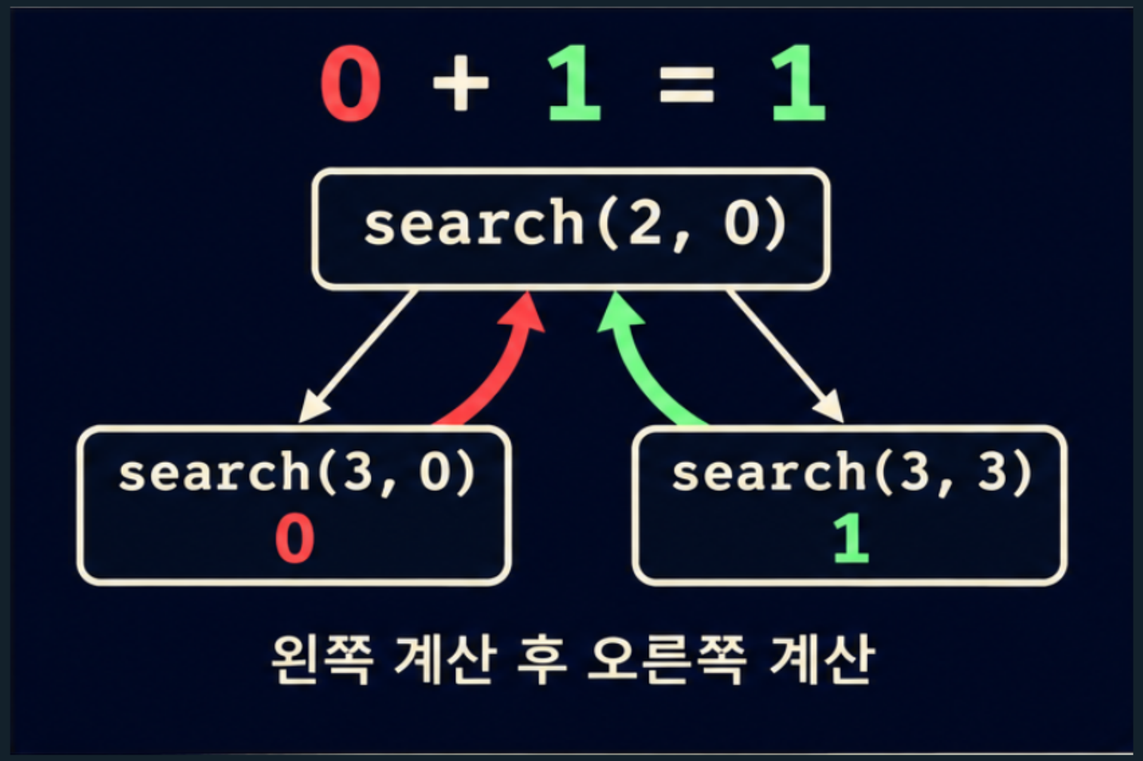
회색: 안쪽 호출의 답을 기다리는 실행

노랑: 지금 실제로 계산하는 실행

return 0을 만나면 그 호출을 시킨 바로 앞 실행으로 돌아갑니다.

7-2 ⑧ 마지막 숫자에서 받은 두 답을 더합니다

```
07          return (search(i+1, total)
08                  + search(i+1, total+a[i]))
```



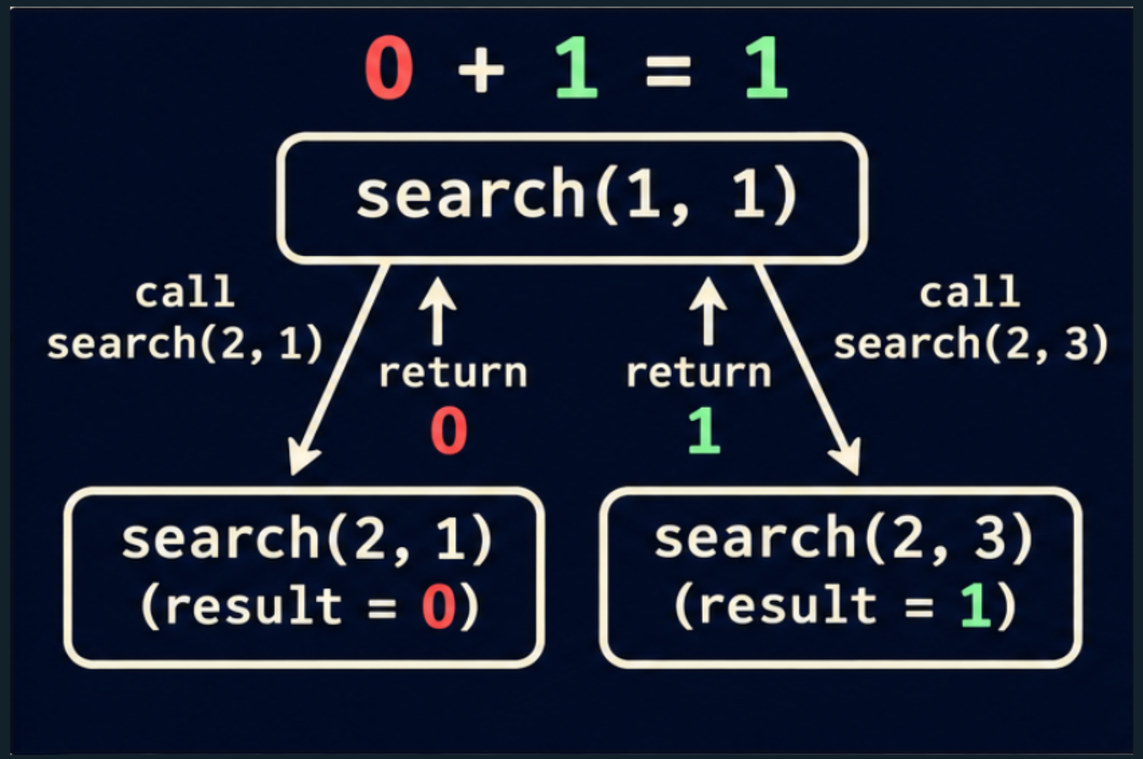
3을 안 고른 쪽이 알려준 값: 0

3을 고른 쪽이 알려준 값: 1

0+1=1은 “지금 상태에서 성공할 방법이 한 가지”라는 답입니다.

7-2 ⑨ 첫 숫자 1을 고른 경우도 같은 방식입니다

```
07          return (search(i+1, total)
08                  + search(i+1, total+a[i]))
```



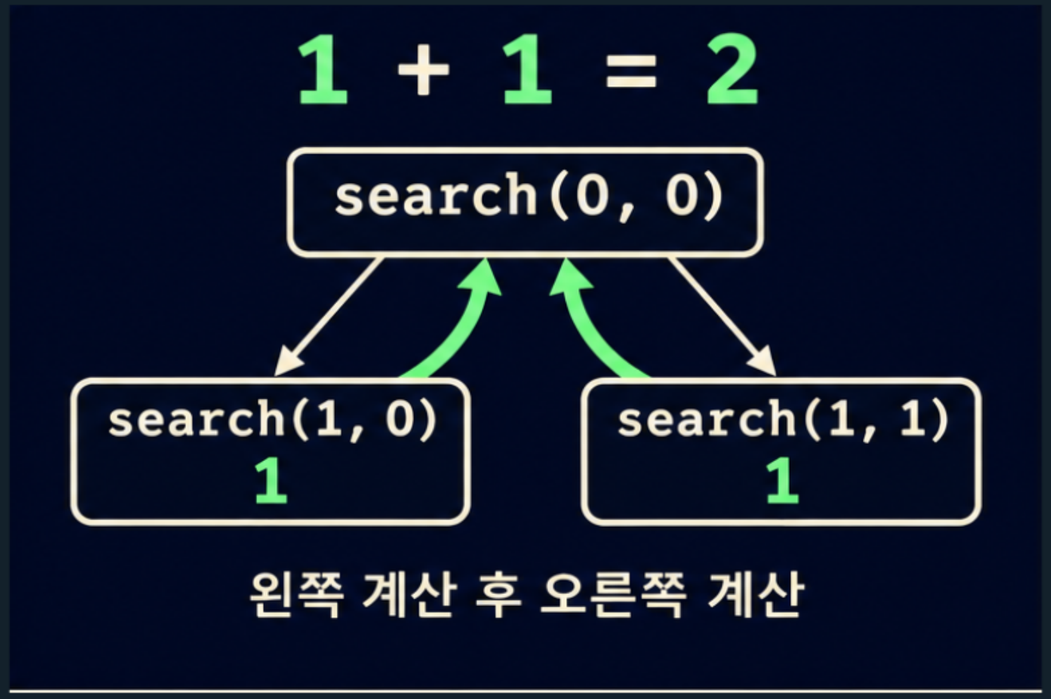
1을 이미 골라 현재 합은 1

2를 건너뛴 쪽 0 + 고른 쪽 1

1과 2를 함께 고르는 한 방법이 성공합니다.

7-2 ⑩ 첫 호출까지 답이 돌아오면 끝납니다

```
07          return (search(i+1, total)
08                  + search(i+1, total+a[i]))
09      return search(0, 0)
```



1을 건너뛴 쪽의 답: 1가지

1을 고른 쪽의 답: 1가지

search(0,0)이 2를 돌려주고, 바깥 함수도 2를 돌려줍니다.

7-2 Python: 두 선택의 개수를 더하기

```python
def subset_sum_count(a, target):
    def search(i, total):
        if total > target:
            return 0
        if i == len(a):
            return int(total == target)
        return (search(i+1, total)
                + search(i+1, total+a[i]))
    return search(0, 0)
```

search(i, total)은 i번부터 확인해 만들 수 있는 성공 선택의 수를 반환합니다.

코드와 방금 본 그림을 연결하기

i == len(a): 모든 숫자를 결정했으므로, 합이 맞으면 1, 다르면 0입니다.

search(i+1, total): 현재 숫자를 건너뛰고 다음 위치로 갑니다.

search(i+1, total+a[i]): 현재 숫자를 고르고 다음 위치로 갑니다.

두 호출이 알려준 성공 개수를 더해서 돌려줍니다.

합이 이미 목표를 넘었으면 그만 봅니다

```
a = [2, 4, 1], target = 3
2와 4를 고른 현재 합 = 6

1을 건너뛰면 6
1을 고르면   7
```

남은 숫자는 양수 1입니다.

고르든 건너뛰든 합이 3으로 줄어들 수 없습니다.

그래서 이 갈래의 성공 수는 0입니다.

더 진행해도 성공할 수 없는 갈래를 멈추는 것을 가지치기라고 합니다.

함수를 다시 부르면 계산할 합을 따로 넘깁니다

지금 합이 1일 때, 2를 고른 경우에는 다음 함수에 합 3을 넘깁니다.

3을 받은 함수가 끝나도, 일을 맡겼던 쪽의 합 1은 그대로입니다.

이제 2를 안 고른 경우를 볼 때는 원래 합 1에서 계속하면 됩니다.

각 실행의 total은 자기 계산에 쓰는 숫자입니다.

이 입력들도 같은 규칙으로 읽을 수 있습니다

a=[1,1], target=1: 앞의 1만 고르거나 뒤의 1만 고르므로 2입니다.

a=[], target=0: 아무것도 고르지 않은 선택 하나가 맞으므로 1입니다.

a=[2,4], target=3: 가능한 합은 0, 2, 4, 6이므로 0입니다.

값이 같아도 위치가 다르면 다른 선택으로 셉니다.

C에서도 같은 값을 함수에 넘깁니다

a는 입력 배열이고 n은 배열의 길이입니다. i는 다음 위치, total은 현재 합입니다.

subset_search는 탐색을 맡는 보조 함수입니다. 시작 함수보다 위에 따로 작성합니다.

subset_sum_count는 사이트가 부릅니다. 이 함수가 위치 0과 합 0으로 탐색을 시작합니다.

const는 배열을 읽기만 한다는 표시이고, static은 이 파일 안에서 쓰는 함수라는 표시입니다.

C 함수 선언에 붙은 단어 읽기

```
static int subset_search(
    const int a[], int n, int target,
    int i, int total)
```

이름 앞 int: 결과로 정수 하나를 돌려줍니다.

const int a[]: 정수 배열을 받아 읽습니다.

static: 이 파일에서 사용할 보조 함수로 둡니다.

실제 실행할 동작은 이어지는 중괄호 { } 안에 씁니다.

7-2 C ① 입력을 받고 보조 함수를 시작합니다

```
01  static int subset_search(const int a[], int n, int target,
02                           int i, int total) {
08  int subset_sum_count(const int a[], int n, int target) {
09      return subset_search(a,n,target,0,0);
```



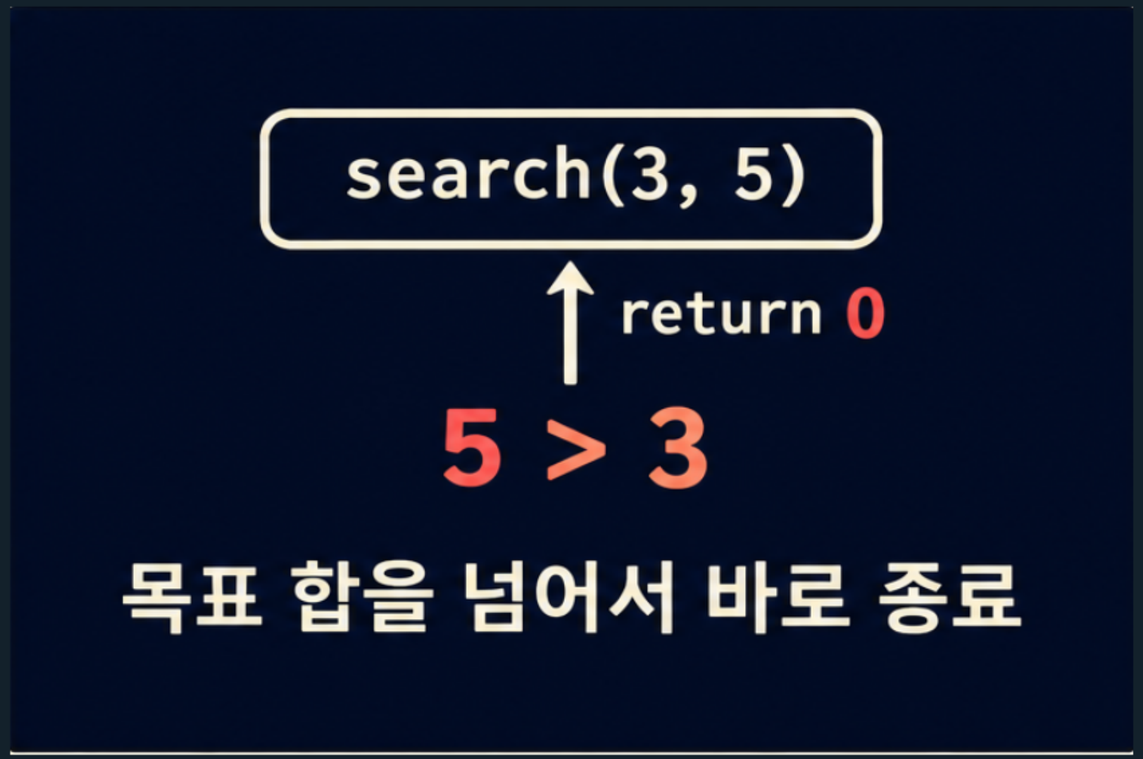
a,n,target은 배열과 길이와 목표

보조 함수에 i=0,total=0을 전달

C는 필요한 값을 괄호 안으로 함께 넘깁니다.

7-2 C ② 멈출 조건을 확인합니다

```
03      if (total>target) return 0;
04      if (i==n) return total==target;
```



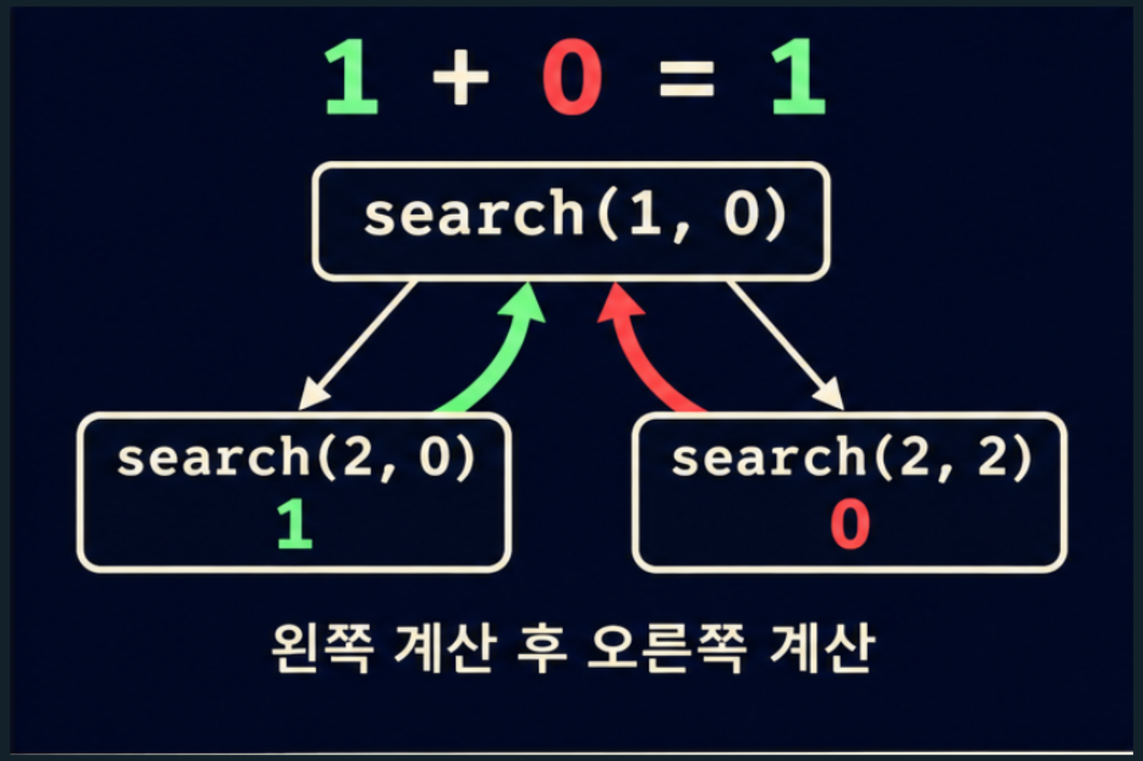
합이 목표 초과 → 성공 0

모두 결정함 → 합이 같은지 비교

C의 비교 결과는 맞으면 1, 틀리면 0입니다.

7-2 C ③ 건너뛴 쪽과 고른 쪽의 답을 더합니다

```
05        return subset_search(a,n,target,i+1,total)
06             + subset_search(a,n,target,i+1,total+a[i]);
```



첫 호출: 위치만 1 늘리고 합 유지

둘째 호출: 위치 1 증가, 지금 숫자도 합에 더함

두 정수 답을 더해 돌려주고 이 함수의 중괄호를 닫습니다.

7-2 C: 시작 함수와 탐색 함수

```
1   static int subset_search(const int a[], int n, int target,
2                            int i, int total) {
3       if (total>target) return 0;
4       if (i==n) return total==target;
5       return subset_search(a,n,target,i+1,total)
6            + subset_search(a,n,target,i+1,total+a[i]);
7   }
8   int subset_sum_count(const int a[], int n, int target) {
9       return subset_search(a,n,target,0,0);
10  }
```

두 subset_search 호출이 찾아낸 성공 수를 더해 반환합니다.

7-3: 이어진 점끼리 다른 색 칠하기



그림의 모든 점을 두 색으로 칠합니다.

점 하나에는 색 하나만 칠합니다.

선으로 바로 이어진 두 점은 색이 달라야 합니다.

모두 칠할 수 있으면 1, 끝내 불가능하면 0입니다.

같은 색을 써도 되는 두 점



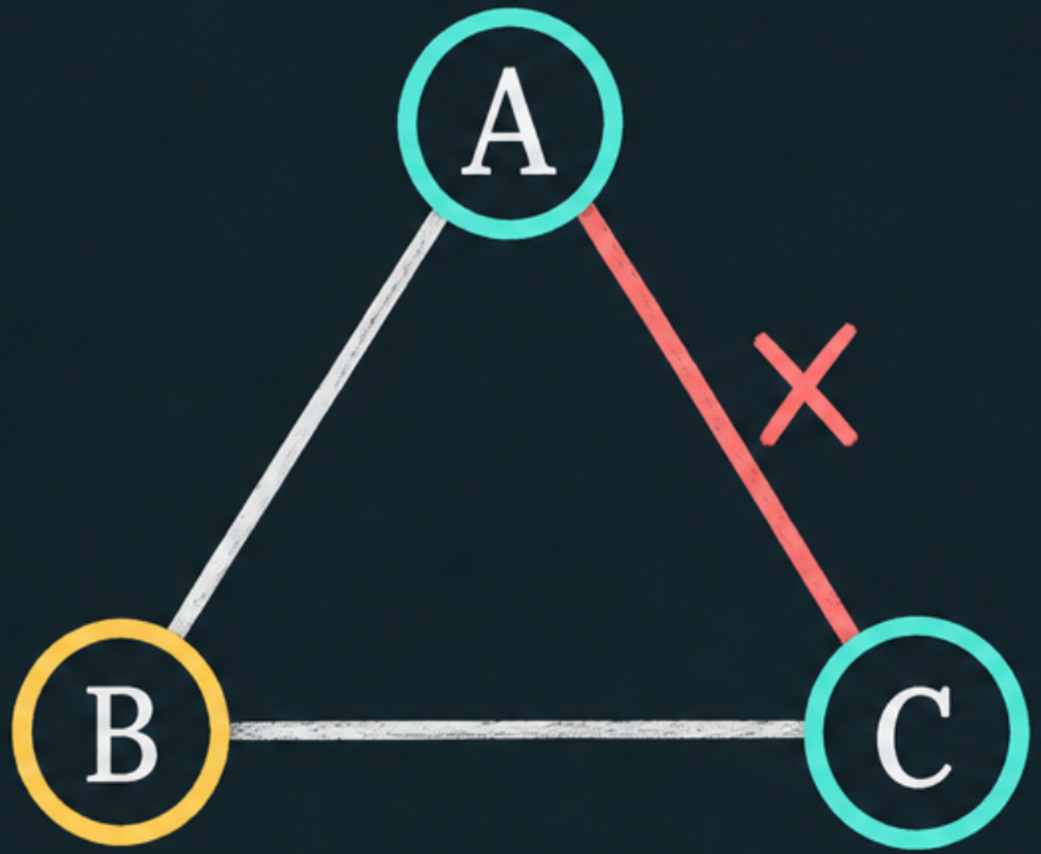
A와 C는 둘 다 초록입니다.

A와 C 사이에는 선이 없습니다.

검사할 것은 A-B와 B-C의 색입니다.

같은 색을 여러 점에 사용해도 됩니다.

삼각형에서는 두 색으로 막힙니다



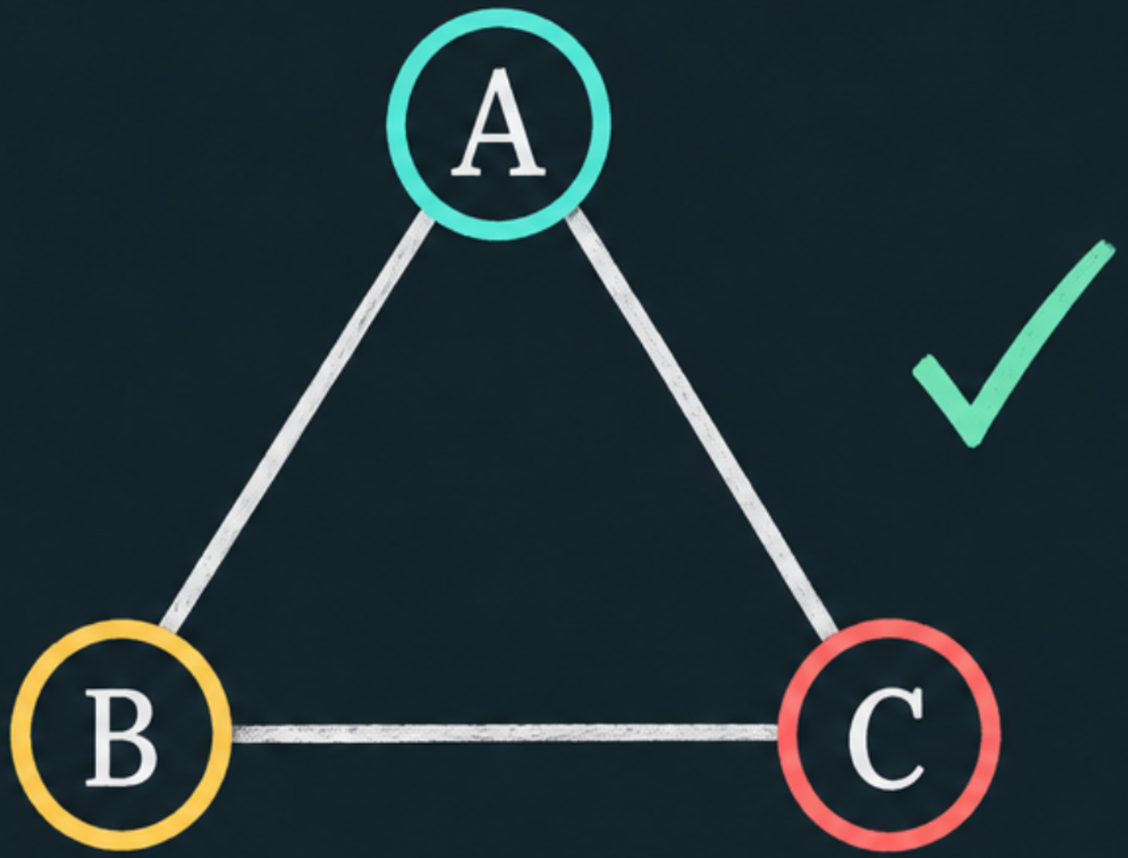
A를 초록, B를 노랑으로 칠했습니다.

C를 초록으로 칠하면 A와 같습니다.

C를 노랑으로 칠하면 B와 같습니다.

삼각형은 두 색으로 끝까지 칠할 수 없습니다.

색이 세 개면 삼각형도 가능합니다



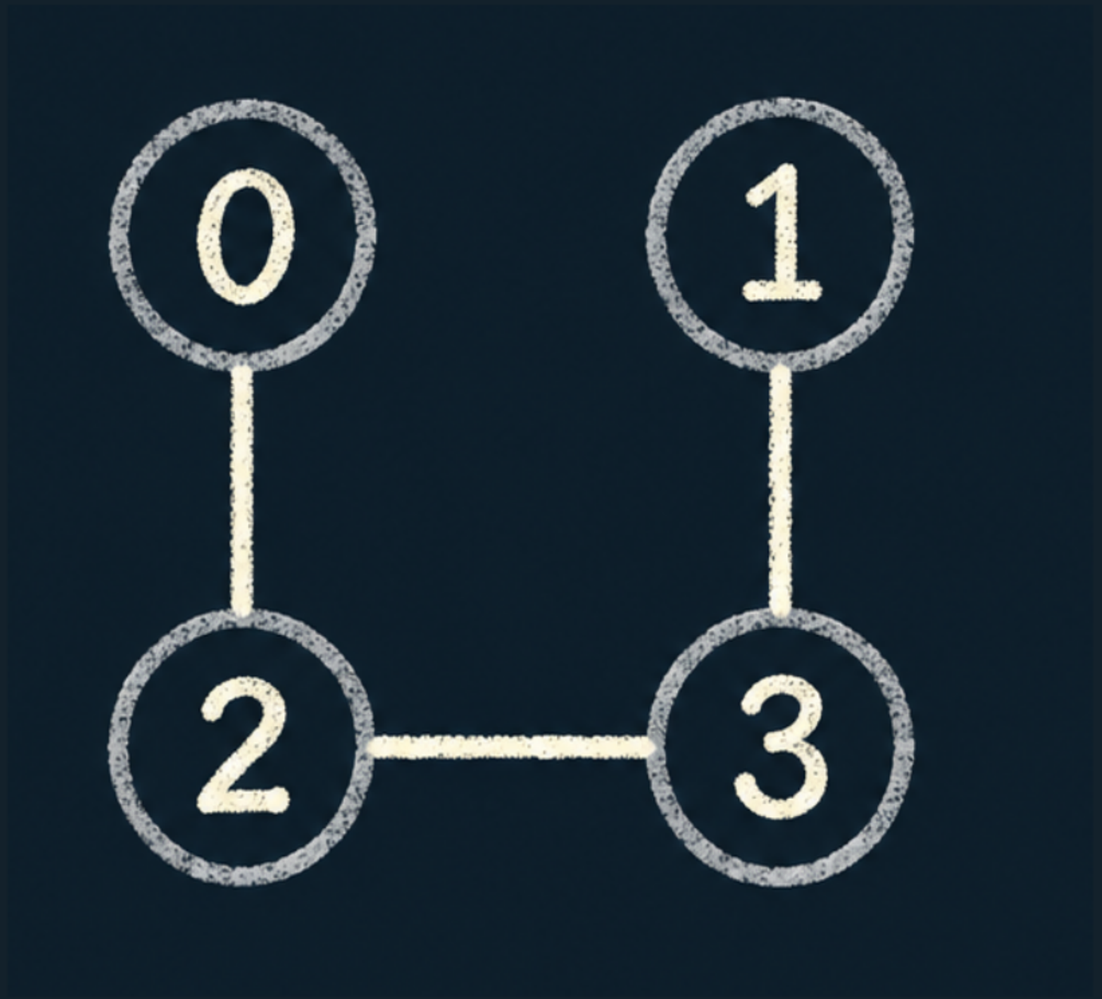
C에 세 번째 색을 칠합니다.

A-B, B-C, A-C의 색이 모두 다릅니다.

세 점을 빠짐없이 칠했습니다.

이때는 가능하므로 1을 반환합니다.

두 색으로 네 점을 칠하는 문제



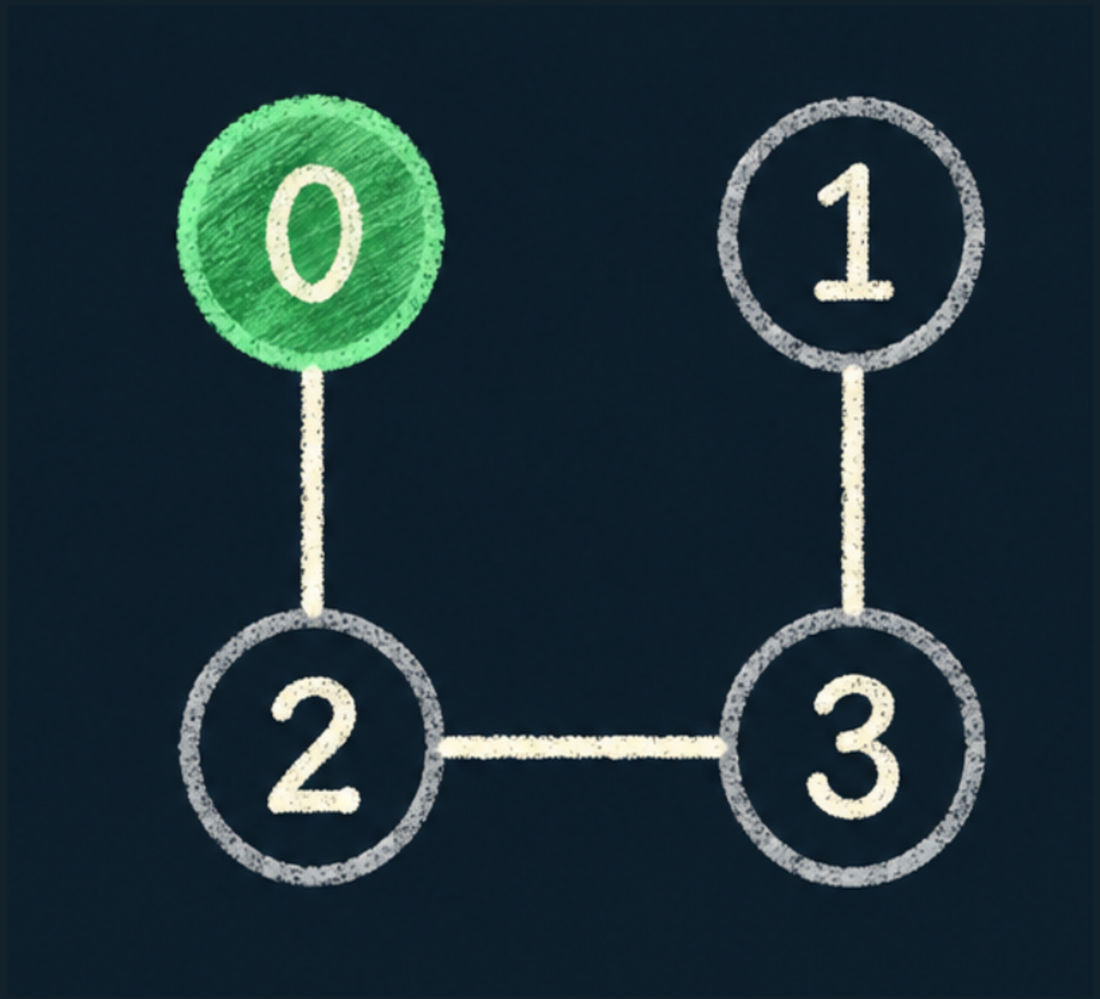
사용할 색은 초록과 노랑입니다.

선은 0-2, 2-3, 3-1에만 있습니다.

점 번호 0, 1, 2, 3 순서로 칠합니다.

각 점에서 초록을 먼저, 노랑을 나중에 시도합니다.

① 0번에 초록을 칠합니다



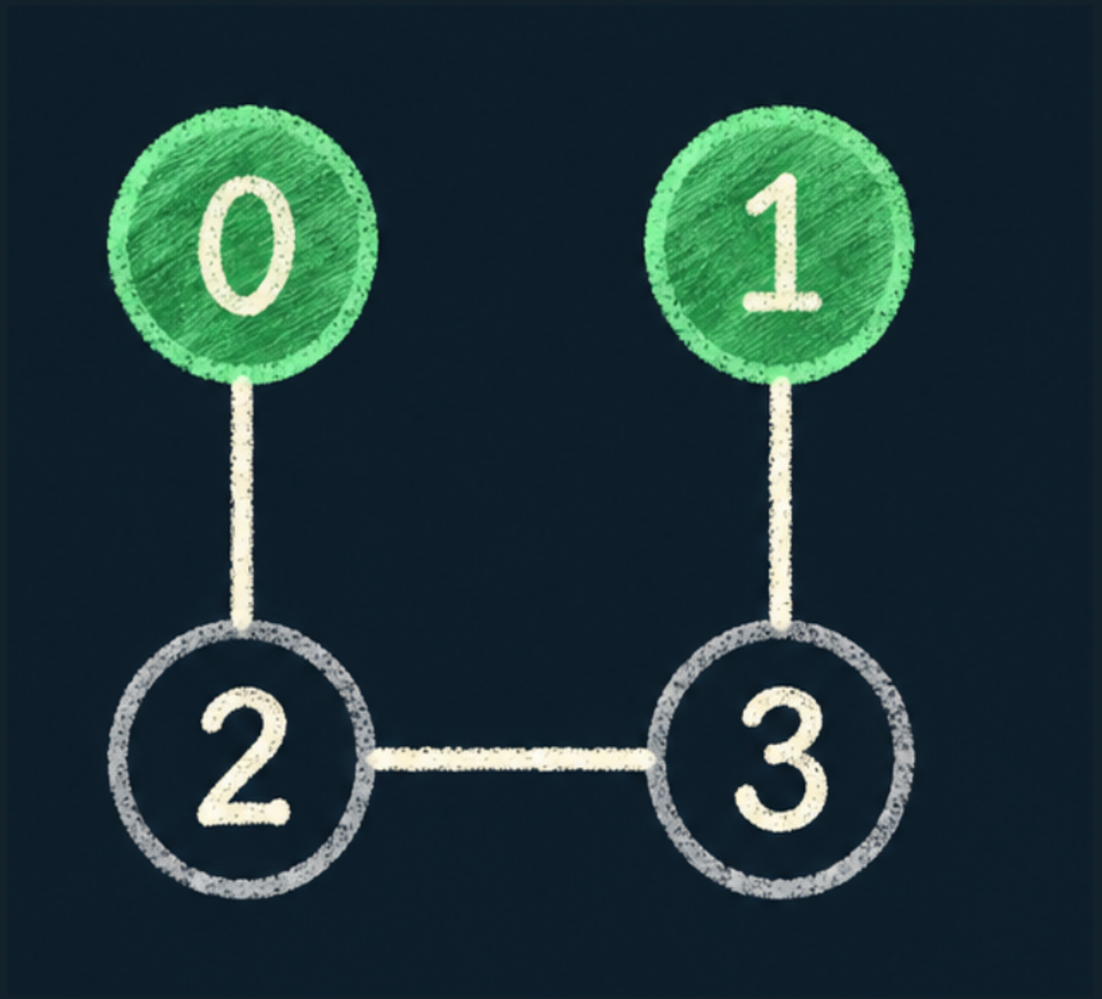
아직 칠한 이웃이 없습니다.

0번에 초록을 써도 충돌하지 않습니다.

0번의 색은 남겨두고 1번으로 갑니다.

현재 색: [초록, 비어 있음, 비어 있음, 비어 있음]

② 1번에도 초록을 칠합니다



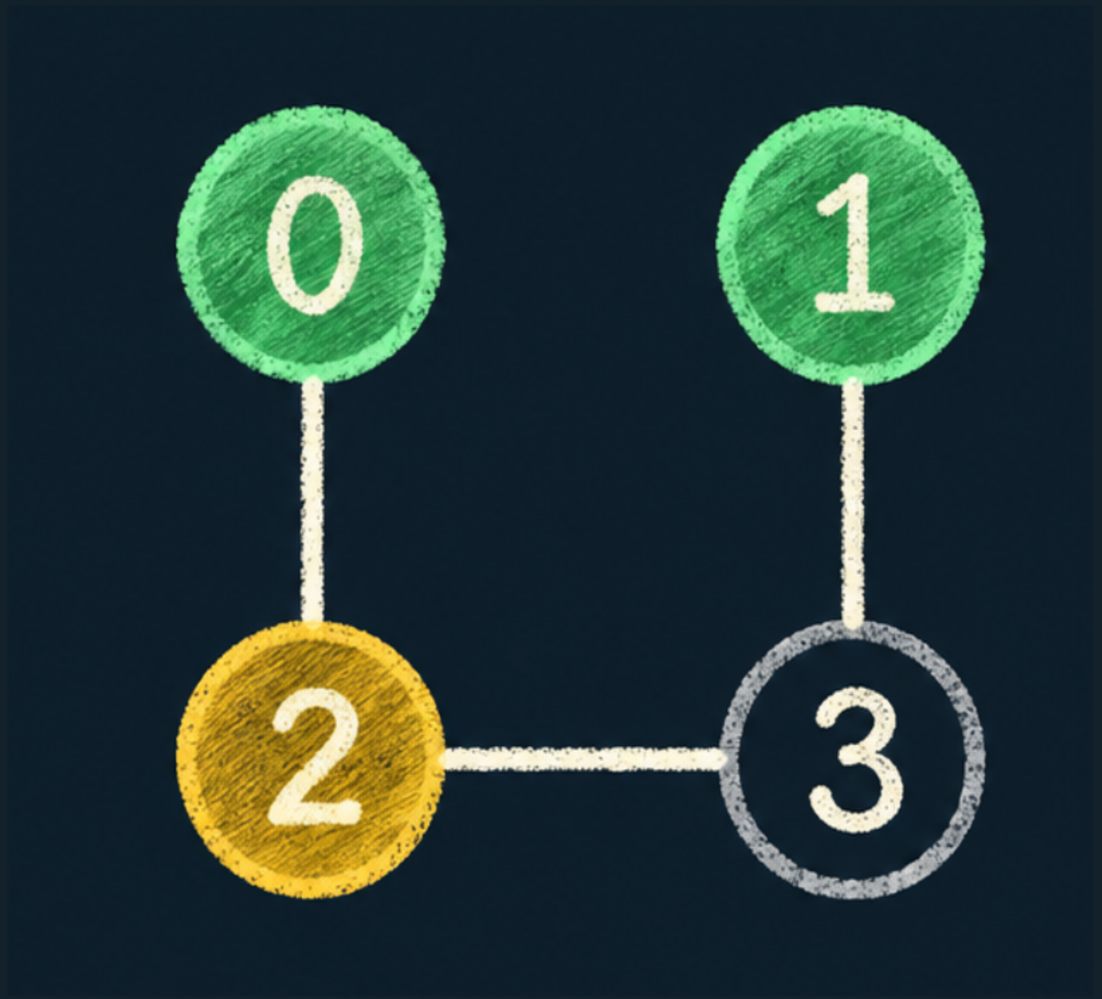
0번과 1번 사이에는 선이 없습니다.

둘의 색이 같아도 규칙에 맞습니다.

두 점을 그대로 두고 2번으로 갑니다.

현재 색: [초록, 초록, 비어 있음, 비어 있음]

③ 2번에는 노랑을 칠합니다



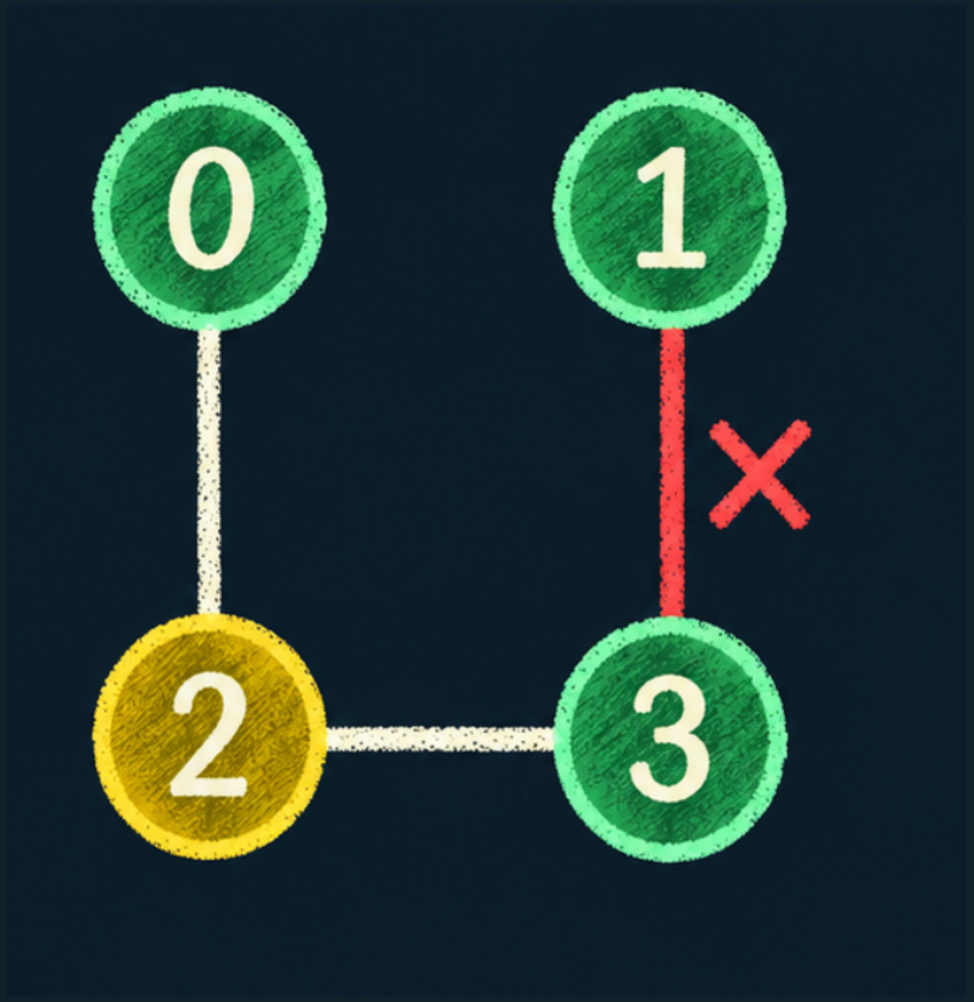
2번은 초록인 0번과 이어져 있습니다.

초록을 쓰면 0번과 충돌합니다.

다음 후보인 노랑은 괜찮습니다.

현재 색: [초록, 초록, 노랑, 비어 있음]

④ 3번에 초록을 시도하면 실패합니다



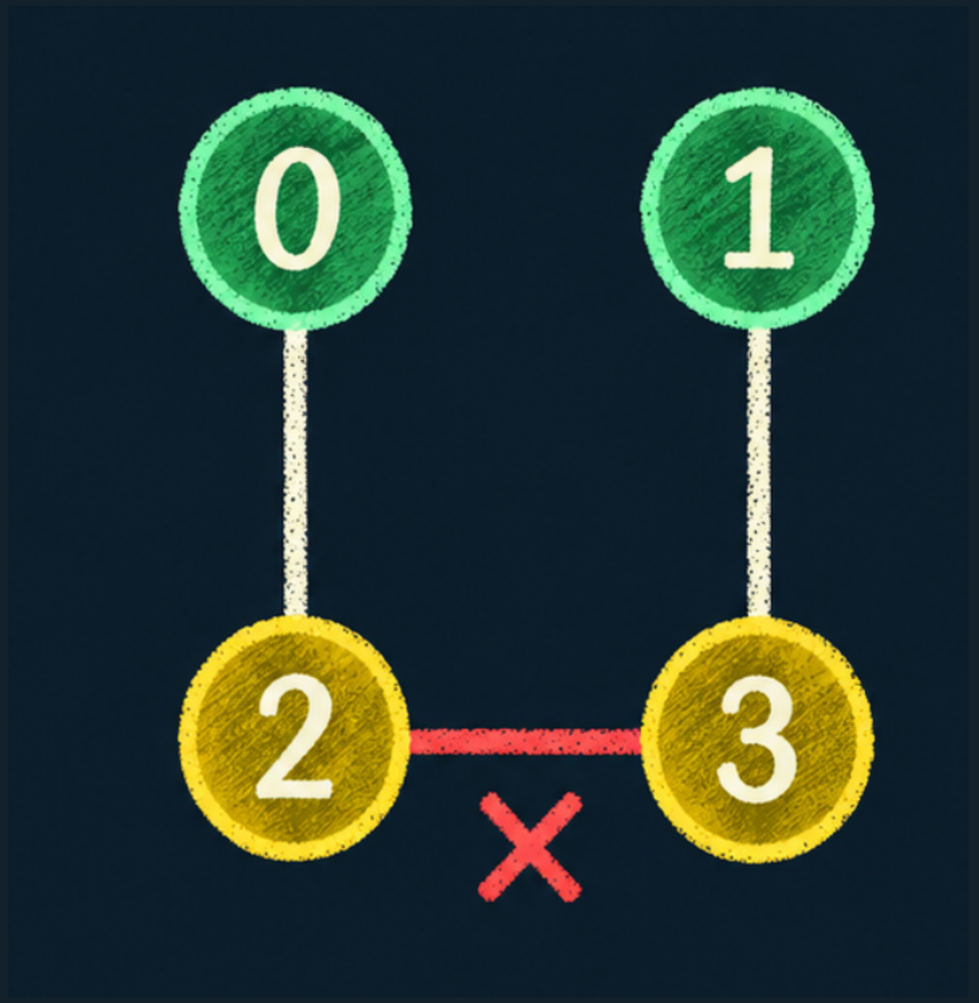
3번은 초록인 1번과 이어져 있습니다.

오른쪽 선의 양 끝이 모두 초록입니다.

이 후보는 사용할 수 없습니다.

3번에서 다음 후보인 노랑을 확인합니다.

⑤ 3번에 노랑을 시도해도 실패합니다



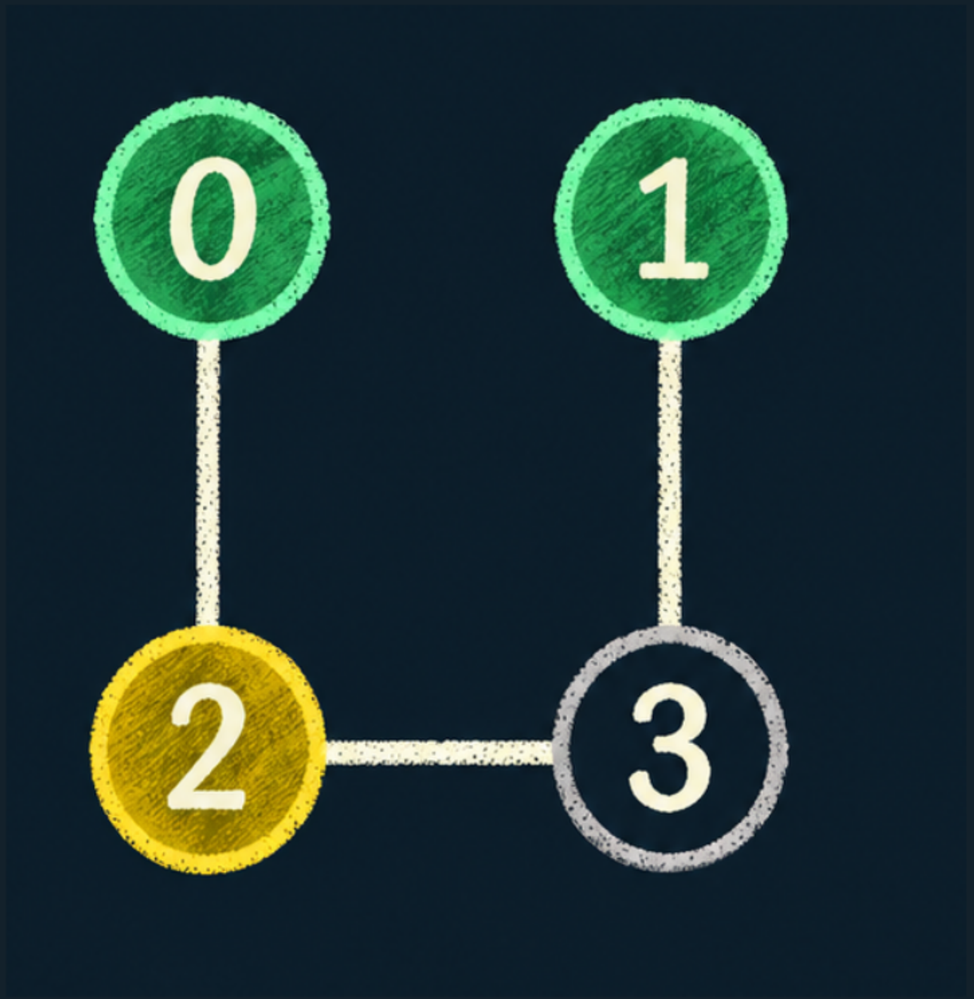
3번은 노랑인 2번과도 이어져 있습니다.

아래쪽 선의 양 끝이 모두 노랑입니다.

3번에 사용할 수 있는 색이 없습니다.

지금까지의 색 선택으로는 끝낼 수 없습니다.

⑥ 3번을 비우고 2번으로 돌아갑니다



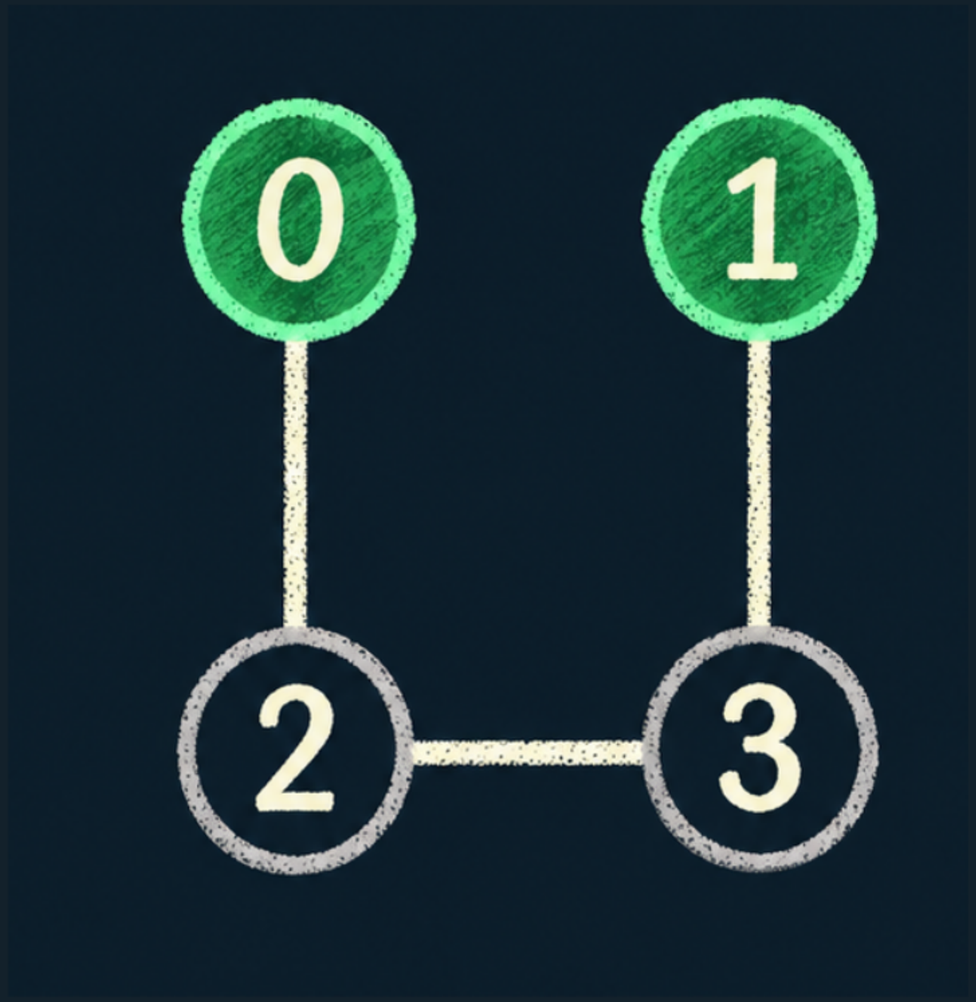
3번의 두 후보를 모두 확인했습니다.

3번에서 시도할 색을 모두 써 봤습니다.

바로 앞에서 칠한 2번의 선택을 다시 봅니다.

0번과 1번의 색은 아직 유지합니다.

⑦ 2번의 노랑을 지웁니다



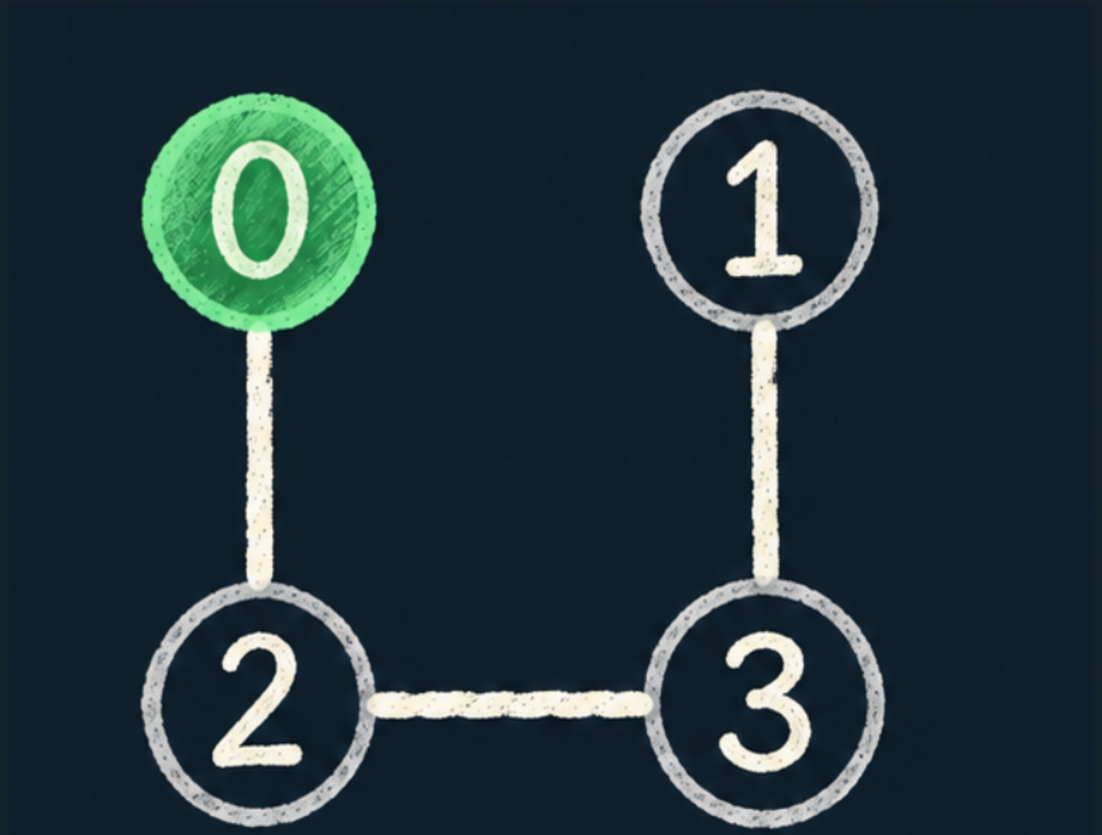
2번의 초록 후보는 이미 실패했습니다.

노랑을 골랐더니 뒤의 3번이 막혔습니다.

2번에 남은 후보가 없으므로 더 돌아갑니다.

2번을 비운 뒤 1번으로 돌아갑니다.

⑧ 1번의 초록을 지웁니다



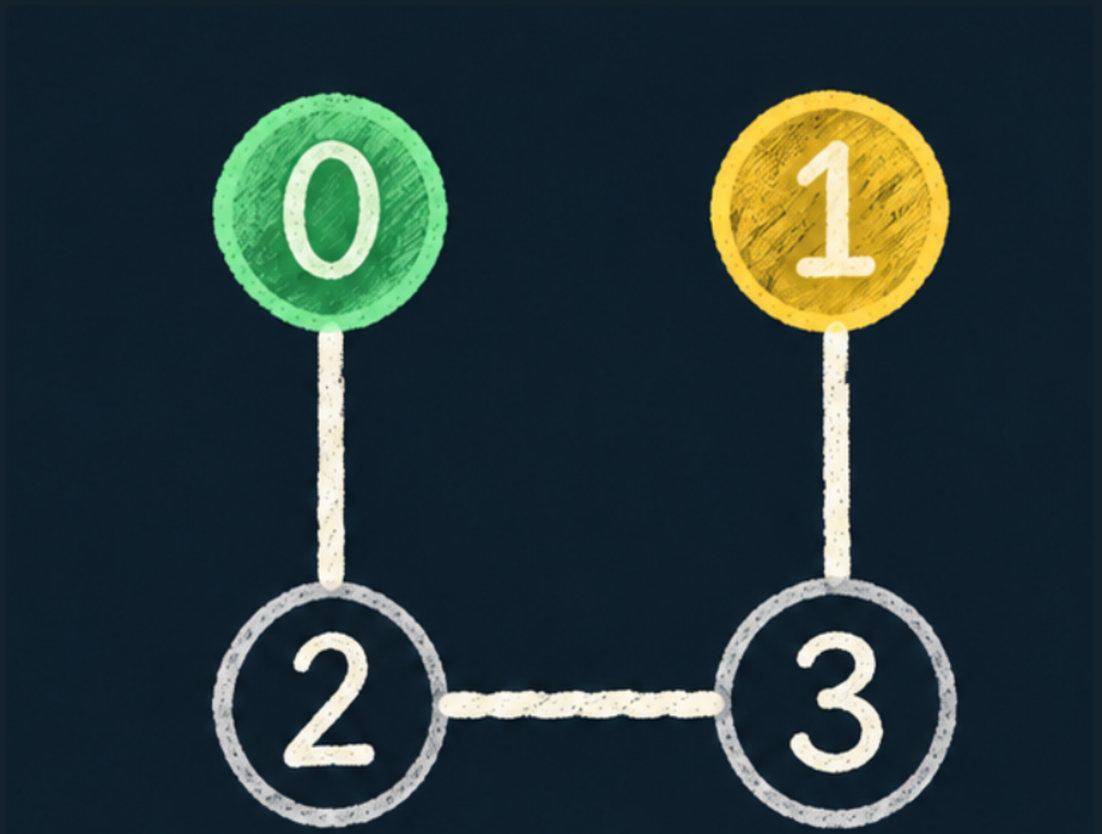
1번의 초록을 택한 뒤가 모두 실패했습니다.

아직 1번의 노랑은 시도하지 않았습니다.

0번의 초록은 그대로 유지합니다.

이제 1번의 다음 후보를 봅니다.

⑨ 1번에 노랑을 칠합니다



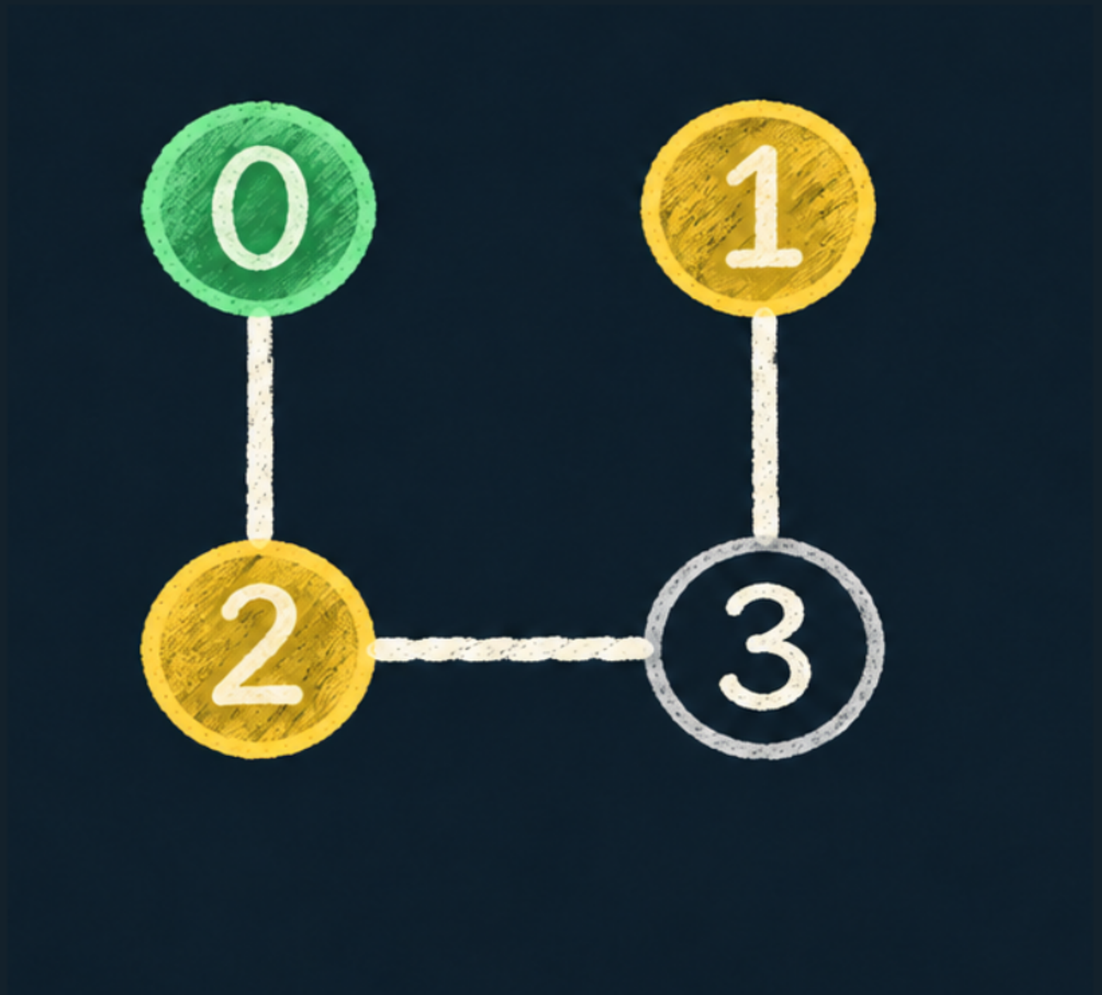
0번과 1번은 직접 이어져 있지 않습니다.

1번의 노랑 후보는 사용할 수 있습니다.

다시 다음 점 2번을 칠하러 갑니다.

현재 색: [초록, 노랑, 비어 있음, 비어 있음]

⑩ 2번에는 다시 노랑을 칠합니다



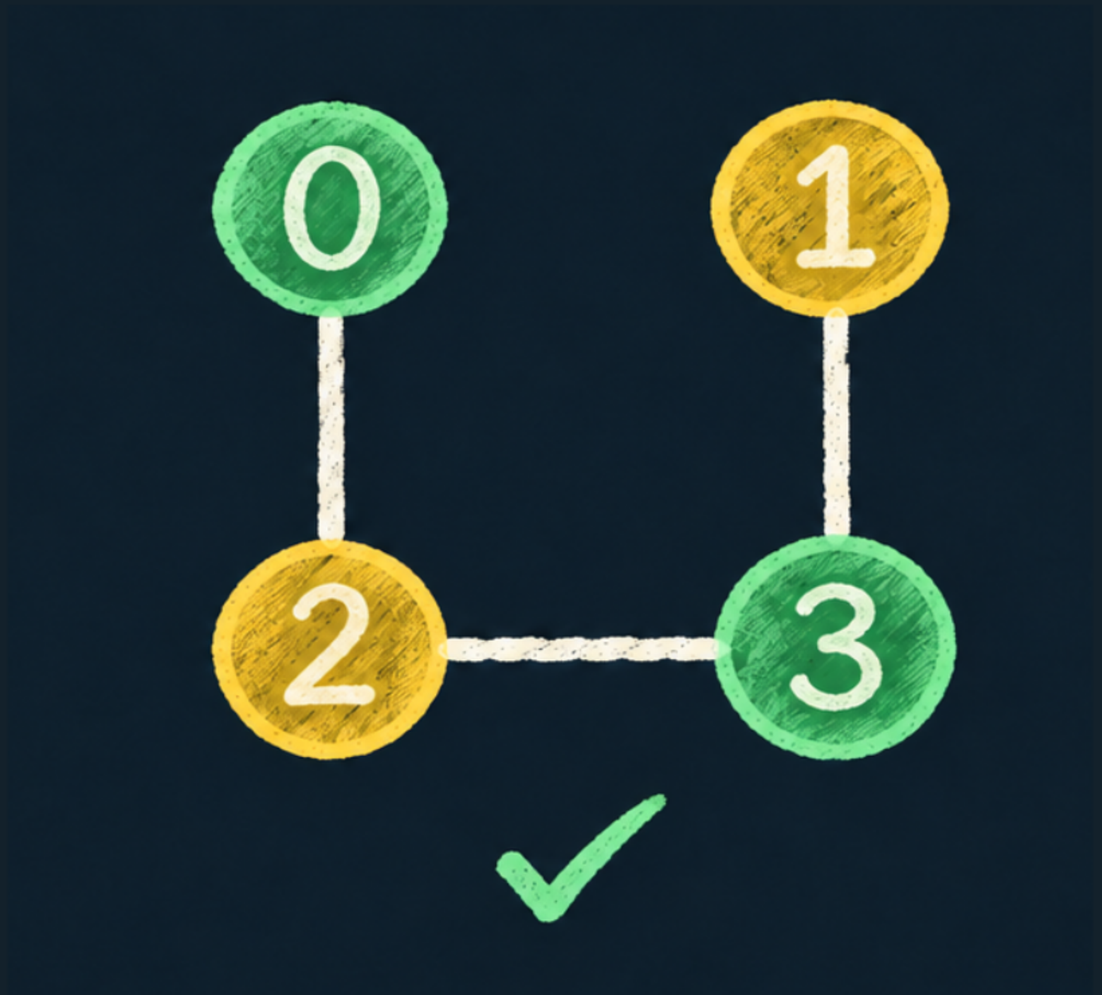
2번은 초록인 0번과 이어져 있습니다.

초록은 못 쓰지만 노랑은 쓸 수 있습니다.

1번과 2번 사이에는 선이 없습니다.

현재 색: [초록, 노랑, 노랑, 비어 있음]

⑪ 3번에 초록을 칠하면 완성됩니다



3번의 이웃 1번과 2번이 모두 노랑입니다.

3번에 초록을 칠해도 충돌하지 않습니다.

네 점을 모두 칠했습니다.

가능한 색칠을 찾았으므로 문제의 답은 1입니다.

완성한 그림에서 세 선을 다시 확인합니다

0–2: 초록과 노랑 / 2–3: 노랑과 초록 / 3–1: 초록과 노랑

모든 점에 색이 있고, 세 선의 양 끝도 모두 다른 색입니다.

조건에 맞는 색칠을 하나 찾았으므로 문제의 답은 1입니다.

이 문제는 가능한 색칠 하나만 찾으면 끝납니다.

문제 7-3: 모든 점을 조건에 맞게 색칠하기

n개의 점으로 이루어진 무방향 그래프를 k가지 색으로 칠하려고 한다. 모든 점에 색 하나를 칠하고, 선으로 직접 연결된 두 점의 색은 서로 달라야 한다.

연결 정보는 n×n 배열 graph에 주어진다. graph[i][j]=1이면 i번과 j번 점이 연결되고, 0이면 연결되지 않는다. 이 배열은 대칭이며 대각선 값은 0이다.

n은 1~8, k는 1~3의 정수이다. 주어진 색을 모두 쓸 필요는 없으며, 떨어져 있는 점도 빠짐없이 칠해야 한다.

이 조건으로 모든 점을 칠할 수 있으면 1, 없으면 0을 반환하는 can_color 함수를 작성하시오.

예: 세 점이 모두 서로 연결된 삼각형은 k=2이면 0, k=3이면 1을 반환한다.

Python: can_color(graph, k) / C: can_color(graph, n, k)

연결된 두 점은 표에 1로 적습니다

```
    0  1  2  3
0  [0, 0, 1, 0]
1  [0, 0, 0, 1]
2  [1, 0, 0, 1]
3  [0, 1, 1, 0]
```

방금 칠한 네 점의 연결을 적습니다.

graph[0][2] = 1
0번과 2번 사이에는 선이 있습니다.

graph[0][1] = 0
0번과 1번 사이에는 선이 없습니다.

한 행이 점 하나입니다. 1은 선이 있음, 0은 선이 없음입니다.

코드에서는 색을 숫자로 적습니다

색을 저장할 칸 네 개를 color=[0,0,0,0]으로 준비합니다.

0은 아직 빈 점, 1은 초록, 2는 노랑으로 정합니다.

완성 그림의 색은 color=[1,2,2,1]입니다.

color의 각 칸은 같은 번호의 점에 칠한 색입니다.

paint(3): 마지막 3번 점을 칠해 달라는 요청

앞의 색 [초록, 노랑, 노랑]을 유지하고 3번부터 확인합니다.



3번 점만 비어 있습니다.

3번에 초록을 칠하면 완성됩니다.

paint는 남은 점을 전부 칠할 수 있으면 1, 끝내 막히면 0을 돌려줍니다.

이 상태에서 paint(3)의 결과는 성공을 뜻하는 1입니다.

지금 후보 색을 앞의 점들과 비교합니다

3번에 초록을 써도 될까요?

0번: 연결 없음 → 통과

1번: 연결됨, 노랑 → 통과

2번: 연결됨, 노랑 → 통과

v는 지금 칠할 점 번호입니다.
여기서는 3입니다.

c는 지금 시험할 색입니다.
여기서는 초록인 1입니다.

u를 0,1,2로 바꾸며 앞에서
칠한 점을 확인합니다.

선이 있고 색도 같을 때만 그 후보를 버립니다.

현재 색을 적고, 다음 점을 칠해 봅니다

```
color[v] = c
if paint(v + 1) == 1:
    return 1
color[v] = 0
```

이번 색 c가 허용되는지 먼저 검사합니다.

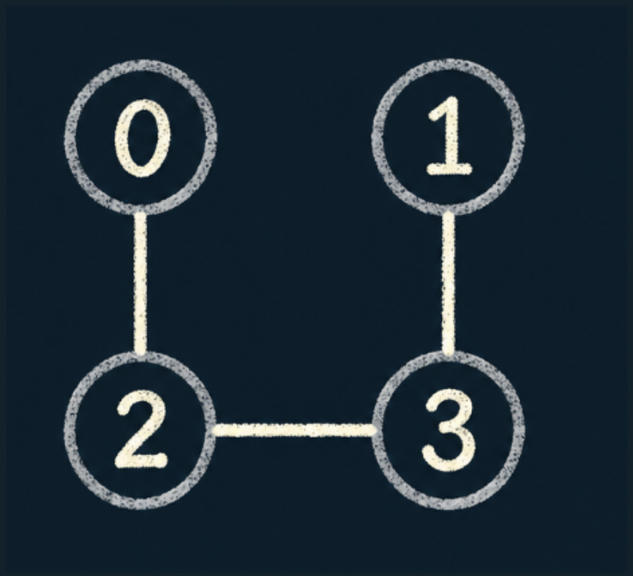
다음 점부터 성공하면 현재 함수도 1을 반환합니다.

다음 점들이 실패했을 때만 이번 색을 0으로 지웁니다.

모든 후보 색이 실패하면 현재 함수는 0을 반환해 앞 점으로 돌아갑니다.

7-3 ① 빈 색 배열을 만들고 0번부터 시작합니다

```
01  def can_color(graph, k):
02      n = len(graph)
03      color = [0]*n
04      def paint(v):
19      return paint(0)
```



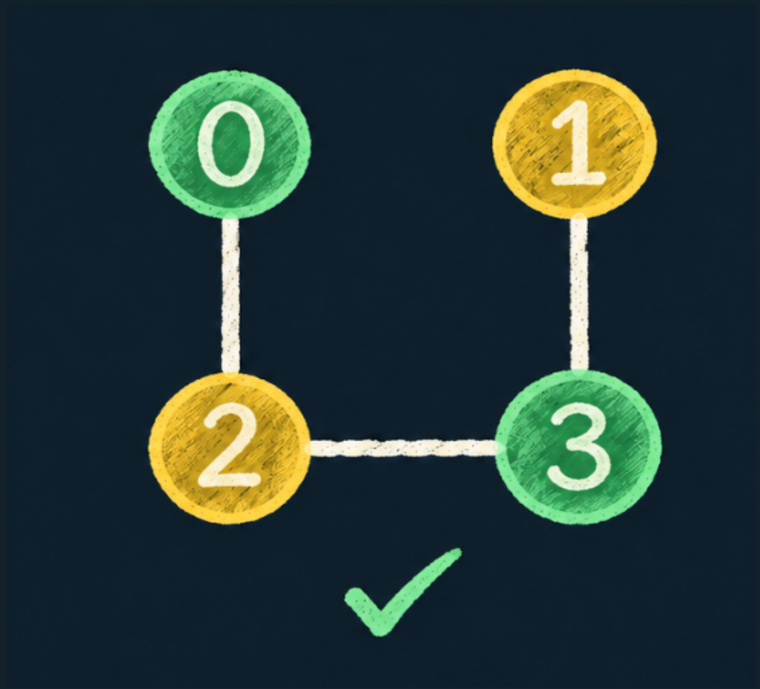
n=4, color=[0,0,0,0]

paint(0)이 첫 점의 색칠을 시작

0은 빈 색 표시입니다. 아직 어느 점에도 색을 칠하지 않았습니다.

7-3 ② 네 점을 모두 칠했으면 성공입니다

```
05          if v == n:
06              return 1
```



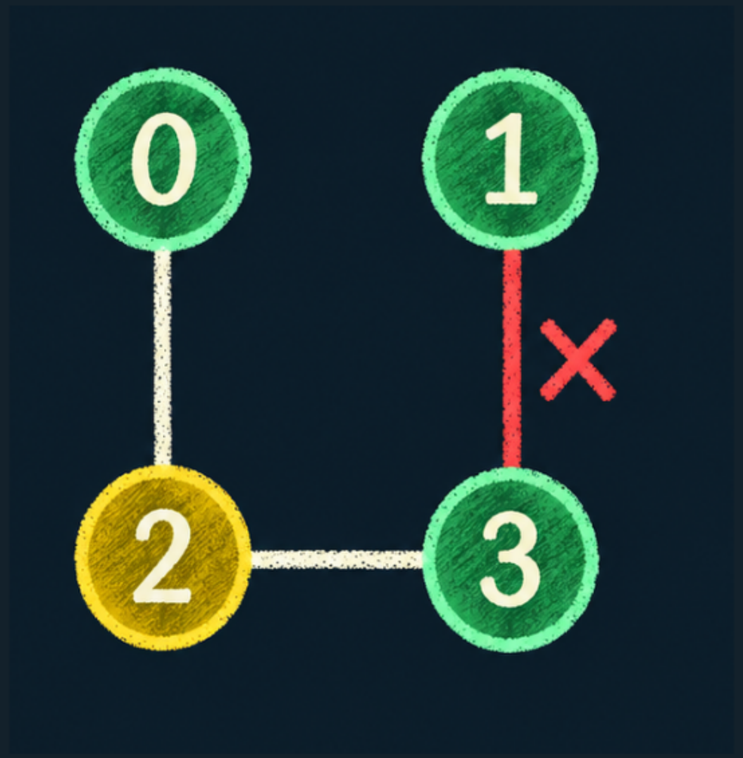
실제 점 번호는 0,1,2,3

다음 번호 v=4라면 모두 칠한 상태

v==n이면 배열을 더 읽지 않고 성공 1을 돌려줍니다.

7-3 ③ 이번 점에서 초록부터 시험합니다

```
07          for c in range(1, k+1):
08              allowed = True
```



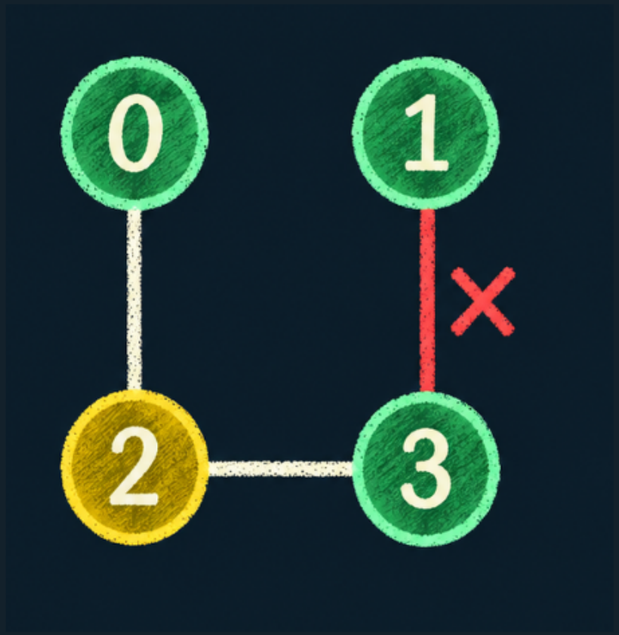
k=2이면 후보 c는 1,2 순서

각 후보마다 allowed=True로 시작

True로 시작한 뒤, 충돌을 찾으면 False로 바꿉니다.

7-3 ④ 연결되고 같은 색이면 후보를 버립니다

```
09                for u in range(v):
10                    if graph[v][u] == 1 and color[u] == c:
11                        allowed = False
12                        break
```



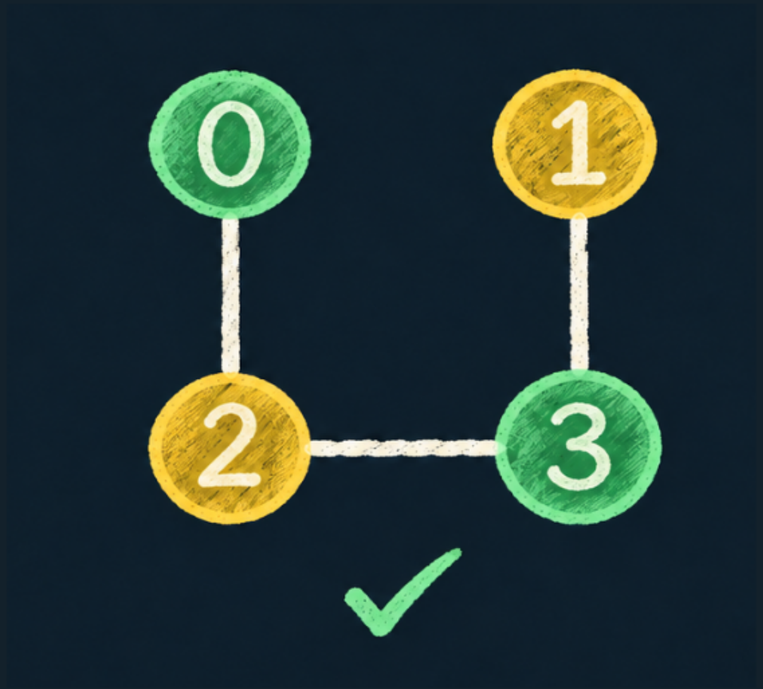
v=3, c=1인 초록을 검사

u=1은 연결되어 있고 색도 초록

두 조건이 모두 맞으므로 allowed=False, break입니다.

7-3 ⑤ 검사를 통과한 색을 실제로 기록합니다

```
13                if allowed:
14                    color[v] = c
```



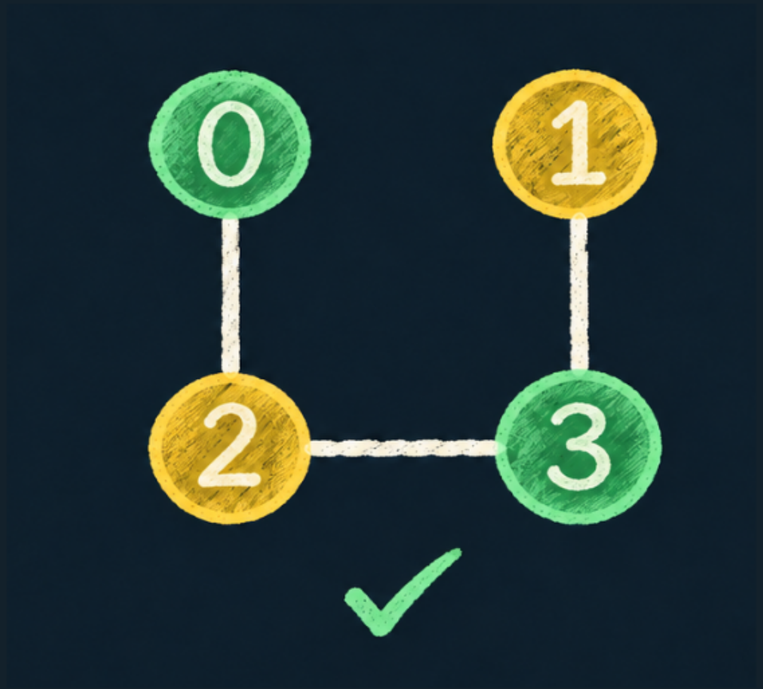
앞의 1번과 2번이 모두 노랑인 상태

3번의 초록 후보는 검사를 통과

color[3]=1로 초록을 기록한 뒤 다음 점을 확인합니다.

7-3 ⑥ 뒤까지 성공하면 지금 실행도 끝납니다

```
15                  if paint(v+1) == 1:
16                      return 1
```



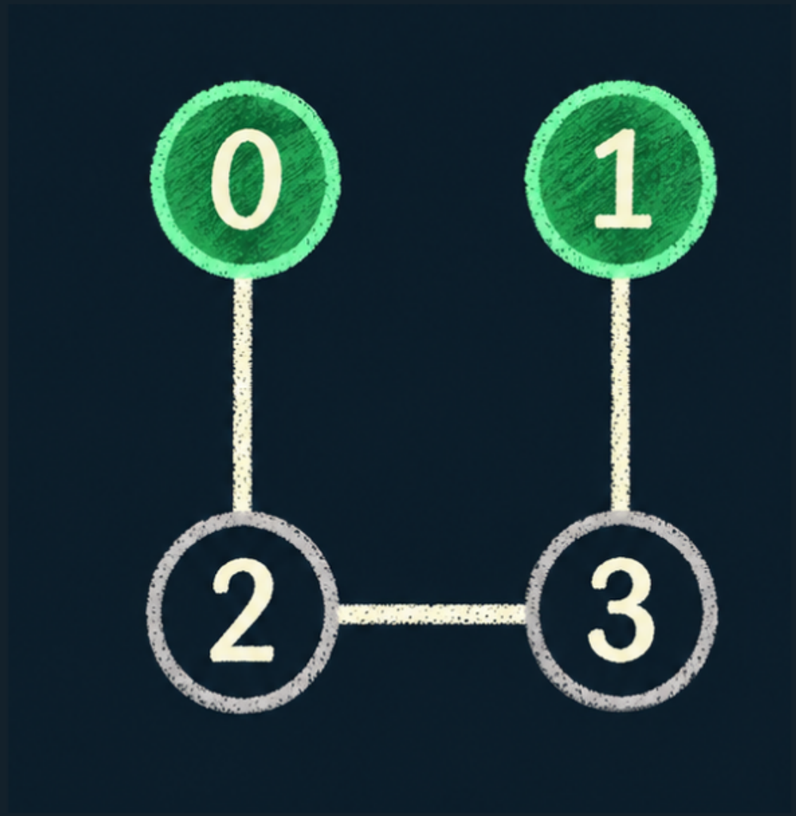
3번을 칠한 뒤 paint(4)를 실행

paint(4)가 성공 1을 돌려줌

return 1에서 끝나므로 아래의 색 지우기 줄은 실행하지 않습니다.

7-3 ⑦ 뒤가 실패했다면 방금 색을 지웁니다

```
17                    color[v] = 0
```



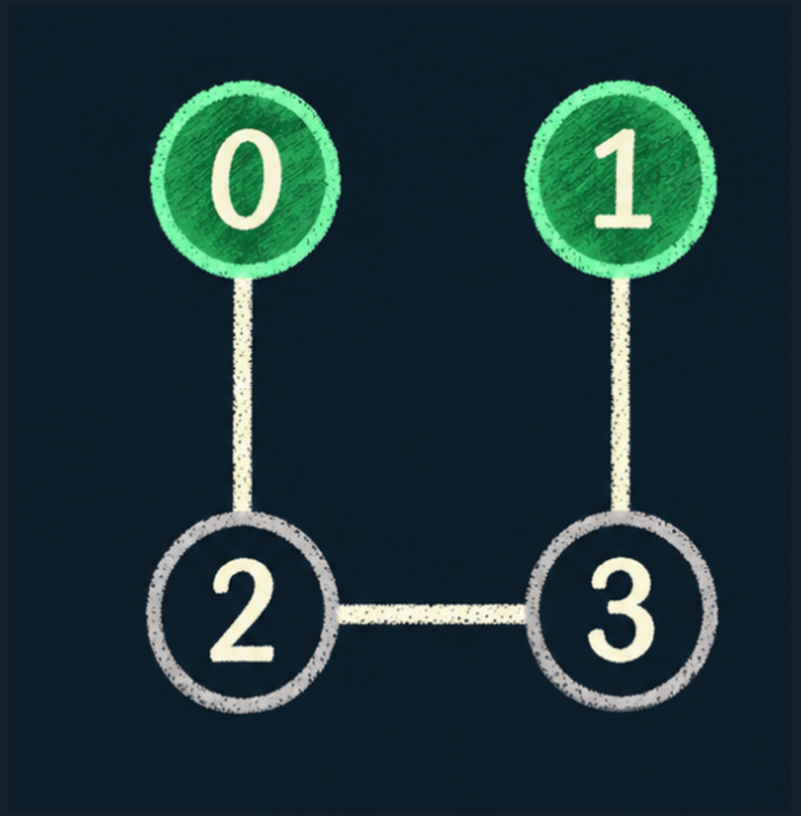
2번을 노랑으로 골랐지만 3번에서 막힘

paint(3)이 0을 알려주어 다음 줄로 내려옴

color[2]=0으로 지우고 2번의 다른 색을 시험합니다.

7-3 ⑧ 모든 색이 실패했을 때 0을 알려줍니다

```
18          return 0
```



이번 점의 후보 1과 2를 모두 확인

성공한 후보가 없어 반복문을 빠져나옴

return 0은 “현재 앞의 선택으로는 뒤를 완성할 수 없음”입니다.

7-3 Python (앞부분)

```python
def can_color(graph, k):
    n = len(graph)
    color = [0]*n
    def paint(v):
        if v == n:
            return 1
        for c in range(1, k+1):
            allowed = True
            for u in range(v):
                if graph[v][u] == 1 and color[u] == c:
                    allowed = False
                    break
```

이어서 다음 화면의 코드를 붙여 같은 함수를 완성합니다.

7-3 Python (뒷부분)

```
13              if allowed:
14                  color[v] = c
15                  if paint(v+1) == 1:
16                      return 1
17                  color[v] = 0
18          return 0
19      return paint(0)
```

앞 화면에서 검사한 뒤, 색을 적고 다음 점으로 갑니다.