완전 탐색

브루트 포스

가능한 후보를 생성하고 조건에 맞는 해를 찾는 방법

2장 학습 순서

한눈에 보는 완전 탐색

가능한 후보를 빠짐없이 검사

모든 경우를 체계적으로 확인하고 문제의 조건을 만족하는 해를 찾습니다.

Brute Force

모든 후보를 직접 검사

Naive Method

단순한 방법으로 탐색

Exhaustive Search

가능한 경우를 전수 조사

기본 탐색 방법

알고리즘 설계의 출발점

Python과 C로 같은 알고리즘 구현

후보를 생성하고 검사하는 과정은 두 언어에서 같습니다.

Python

리스트를 만들고 return으로 반환
itertools로 순열·조합 생성
문자열 비교: ==
큰 정수를 자동으로 처리

리스트와 표준 라이브러리 중심

C11

out 배열에 저장하고 개수 반환
반복문·재귀로 순열·조합 생성
문자열 비교: strcmp
int / long long의 범위 확인

배열의 용량과 반환값을 명시

실습 2강: Python 또는 C를 선택해 24문제 풀이

먼저 구별할 것: 입력·후보·결과

코드를 읽기 전에 · 모든 예제에 공통으로 쓰이는 구분

```
입력 → 후보 생성 → 조건 검사 → 결과 갱신 → 반환
```

예를 들어 [1,3,5,7]과 목표 8이 입력입니다. 두 위치를 고른 후보 (1,3)은 탈락하고 (1,7)은 통과합니다. pairs는 통과한 후보만 모읍니다.

```
인덱스 i=0  /  값 arr[i]=1  /  개수 len(arr)=4
```

위치 번호, 그 위치의 값, 전체 개수는 서로 다릅니다. 인덱스는 0부터 시작하므로 원소가 4개이면 마지막 위치는 3입니다.

```
return은 호출자에게 전달 / print는 화면에 표시
```

함수가 답을 print만 하고 return하지 않으면 Python 호출자는 None을 받습니다. 함수 제출 문제는 반환값으로, 전체 프로그램 문제는 표준 출력으로 채점하는지 확인합니다.

Python 들여쓰기: 처리 단위가 달라진다

공통 문법 · 각 예제에서 들여쓰기의 이유를 다시 확인합니다

```
for i ...:  →  for j ...:  →  if ...:
```

안으로 들어갈수록 현재 후보의 더 작은 단계를 처리합니다. if 안의 저장은 조건이 참일 때만, j 밖의 작업은 j 반복이 모두 끝난 뒤 실행됩니다.

```
초기화 위치 = 언제 새로 시작할 것인가?
```

전체 정답 count=0은 탐색 전에 한 번, 현재 후보 sum=0은 후보마다 한 번 둡니다. 초기화 위치를 바꾸면 이전 값을 잃거나 서로 다른 후보를 섞게 됩니다.

```
반복문 밖의 return = 모든 후보를 처리한 뒤 종료
```

return이 반복문 안에 있으면 첫 후보를 처리하고 함수가 끝날 수 있습니다. 탭과 공백을 섞지 말고 한 단계 공백 4칸으로 맞춥니다.

C 배열·포인터: 결과를 전달하는 두 경로

공통 문법 · { }로 문장 범위를 묶고 ;로 문장을 끝냅니다

```
int count = solve(input, out);
```

함수는 out 배열에 결과 값을 쓰고 count로 유효한 칸 수를 알려줍니다. 배열 용량과 실제 저장한 개수는 다르므로 out 전체를 무조건 출력하지 않습니다.

```
out[count++] = value;
```

현재 count 위치에 저장한 다음 count를 늘립니다. out[count]=value; count++;로 나누어 쓰면 같은 동작입니다. 이 수업의 짧은 표현은 해설에서 순서대로 풀어 읽습니다.

```
&count = 주소 / *count = 그 주소에 있는 값
```

재귀 함수 여러 개가 같은 결과 수를 수정할 때 주소를 전달합니다. 배열은 전달받은 공간에 쓸 수 있고, 정수 값만 넘기면 호출자 정수는 자동으로 바뀌지 않습니다.

C 함수 실행과 브라우저 채점

내려받는 .c 파일은 main을 포함합니다. 브라우저에는 함수와 보조 함수만 제출합니다.

```c
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int find_divisors(int n, int out[]); // implemented later

int main(void) {
  int out[10000];
  int count = find_divisors(10, out);
  for (int i = 0; i < count; i++)
    printf("%d ", out[i]);
  putchar('\n');
  return 0;
}
```

실행 결과: 1 2 5 10 · find_divisors 함수 정의를 이어 붙이면 완전한 프로그램입니다.

C 프로그램은 어디서 시작할까?

줄별 해설 1/3 · 무엇을 하는가 → 왜 필요한가

```
#include <...>
```

헤더는 함수·상수의 선언을 알려 줍니다. stdio.h는 입출력, string.h는 strcmp·strlen, stdlib.h는 llabs, limits.h는 INT_MAX·LLONG_MAX에 필요합니다.

```
int find_divisors(int n, int out[]);
```

함수의 이름·입력·반환형을 먼저 알리는 선언입니다. 이 한 줄은 구현이 아닙니다. 뒤에서 배우는 함수 본문을 함께 넣어야 실행 파일을 만들 수 있습니다.

```
int main(void) { ... }
```

일반 C 실행 파일은 main에서 시작합니다. 브라우저 실습에서 함수만 제출하는 경우에는 채점기가 main 역할을 하므로 문제의 제출 형식을 먼저 확인합니다.

C 프로그램은 어디서 시작할까?

줄별 해설 2/3 · 무엇을 하는가 → 왜 필요한가

```
int out[10000];
```

결과를 담을 정수 10,000칸을 확보합니다. 아직 유효한 약수가 들어 있는 것은 아닙니다. 함수가 채운 앞쪽 count칸만 읽어야 합니다.

```
int count = find_divisors(10, out);
```

10의 약수를 out에 써 달라고 호출합니다. 함수의 반환값 4는 약수 자체가 아니라 저장한 개수입니다. out에는 1, 2, 5, 10이 들어갑니다.

```
for (...) printf("%d ", out[i]);
```

i=0부터 count-1까지 유효한 결과만 출력합니다. %d는 정수 한 개의 출력 자리이며 out[i]가 그 자리에 들어갑니다.

C 프로그램은 어디서 시작할까?

줄별 해설 3/3 · 무엇을 하는가 → 왜 필요한가

```
putchar('\n'); return 0;
```

줄바꿈 문자를 출력한 뒤 프로그램을 정상 종료합니다. main의 0은 성공 상태이며, 약수의 개수 4와는 전혀 다른 반환값입니다.

C 프로그램은 어디서 시작할까?

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 실행 순서: main → find_divisors(10, out) → main으로 복귀 → 출력 → 종료.

02 최종 출력: 1 2 5 10. count=4이므로 out[4] 이후는 읽지 않습니다.

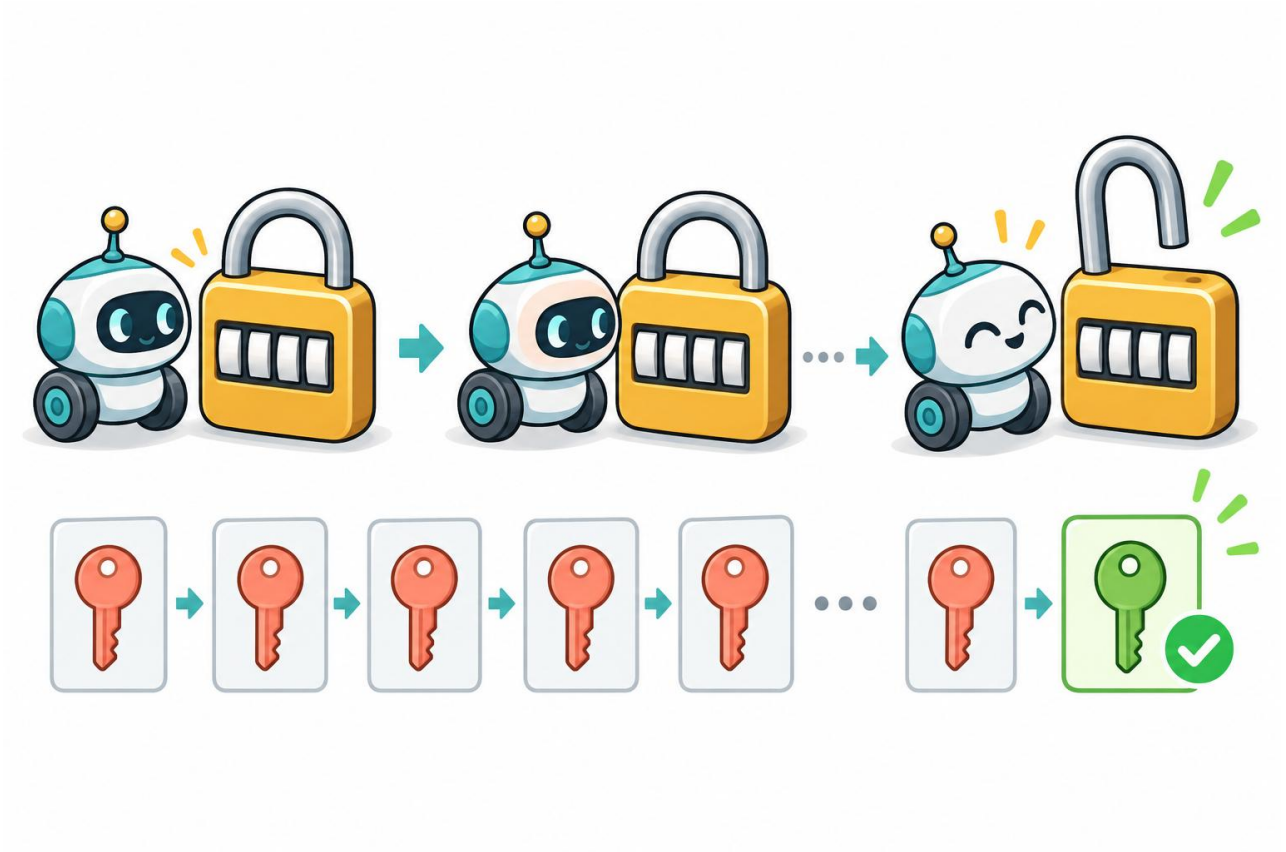
03 직접 확인: 함수 선언만 남기고 본문을 빼면 연결 단계에서 구현을 찾지 못합니다.

2.1

완전 탐색이란 무엇인가?

브루트 포스(Brute Force)의 핵심 개념을 자물쇠 비밀번호 찾기 비유로 이해합니다.

자물쇠 비밀번호 찾기



후보를 차례로 검사

0000부터 9999까지

앞자리 0도 그대로 유지

10 × 10 × 10 × 10

가능한 비밀번호는 10,000개

찾으면 종료

정답이 마지막이면

10,000번 모두 검사

그림은 검사 과정을 줄여 표현한 예시입니다. 각 자리에는 0~9가 들어갑니다.

완전 탐색의 핵심

모든 후보에 같은 조건 검사를 적용

전수 조사

빠짐없이 모두 확인

탐색 범위를 명확히 정의

정확성

범위 안의 해를 발견

후보 생성과 조건 검사가 정확해야 함

기초 도구

반복문·순열·조합

문제에 맞는 생성 방법 선택

자물쇠 비밀번호 찾기: 4중 반복문

secret_code = "8234" → 8,235회 시도 후 발견

lock_bruteforce.py

```
secret_code = "8234"
count = 0
for d1 in range(10):              # 첫 번째 자리 (0~9)
    for d2 in range(10):          # 두 번째 자리
        for d3 in range(10):      # 세 번째 자리
            for d4 in range(10):  # 네 번째 자리
                attempt = f"{d1}{d2}{d3}{d4}"
                count += 1
                if attempt == secret_code:
                    print(f"비밀번호: {attempt} ({count}회)")
                    exit()
```

네 자리 비밀번호: 중첩 반복의 실제 순서

줄별 해설 1/3 · 무엇을 하는가 → 왜 필요한가

```
secret_code = "8234"; count = 0
```

비밀번호는 네 자리 문자열로 취급합니다. "0007"처럼 앞의 0도 보존하기 위해서입니다. count는 이미 검사한 후보 수이며 검사 전에는 0입니다.

```
for d1 in range(10):
```

첫 자리 d1은 0부터 9까지입니다. range의 끝 10은 포함하지 않습니다. d1=0인 동안 아래의 세 반복문이 모두 끝나야 d1=1로 넘어갑니다.

```
for d2 ... / for d3 ... / for d4 ...
```

각 자리마다 다시 0~9를 고릅니다. 가장 안쪽 d4가 가장 빨리 바뀌어 0000, 0001, ..., 0009, 0010 순으로 후보가 생깁니다.

네 자리 비밀번호: 중첩 반복의 실제 순서

줄별 해설 2/3 · 무엇을 하는가 → 왜 필요한가

```
attempt = f"{d1}{d2}{d3}{d4}"
```

네 정수를 문자열 안에 순서대로 붙입니다. d1=0,d2=0,d3=0,d4=7이면 "0007"입니다. 네 숫자를 더하는 식이 아닙니다.

```
count += 1
```

이번 후보도 검사하므로 1을 더합니다. 첫 후보 0000을 검사할 때 count=1입니다. 후보의 숫자 값과 검사 횟수는 1만큼 차이 납니다.

```
if attempt == secret_code:
```

문자열 내용이 같은지 비교합니다. ==는 비교이고 =는 대입입니다. 일치할 때만 아래의 출력과 종료를 실행합니다.

네 자리 비밀번호: 중첩 반복의 실제 순서

줄별 해설 3/3 · 무엇을 하는가 → 왜 필요한가

```
print(...); exit()
```

찾은 비밀번호와 횟수를 출력하고 프로그램 전체를 끝냅니다. 이 예제의 exit()는 독립 실행용이며 함수 제출에서는 return으로 끝내야 합니다. break 하나는 d4 반복만 끝냅니다.

네 자리 비밀번호: 중첩 반복의 실제 순서

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 0000 → 1회, 0001 → 2회, 0009 → 10회, 0010 → 11회입니다.

02 8234를 검사하는 횟수는 8,235회입니다. 마지막 9999는 10,000회째입니다.

03 네 자리의 후보 수는 10×10×10×10입니다. 자릿수가 k라면 10의 k제곱으로 늘어납니다.

네 자리 자물쇠 · C

P13 / return = 시도 횟수

```c
int lock_attempts(const char secret[]) {
  int count = 0;
  for (int a = 0; a < 10; a++)
    for (int b = 0; b < 10; b++)
      for (int c = 0; c < 10; c++)
        for (int d = 0; d < 10; d++) {
          char attempt[5] = {'0' + a, '0' + b, '0' + c, '0' + d, '\0'};
          count++;
          if (strcmp(attempt, secret) == 0)
            return count;
        }
  return 0;
}
```

C에서는 char attempt[5]에 네 자리와 끝의 '₩0'을 저장하고 strcmp로 비교합니다. break는 가장 안쪽 반복문만 끝냅니다.

C 비밀번호: 문자 배열과 문자열 비교

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
int lock_attempts(const char secret[])
```

secret은 바꾸지 않을 입력 문자열입니다. 반환하는 int는 성공할 때까지 검사한 횟수입니다. 입력은 숫자 네 자리와 문자열 종료 문자로 구성된다고 가정합니다.

```
int count = 0;
```

아직 검사하지 않았으므로 0으로 시작합니다. 반복문 밖에 두어야 다음 후보에서도 누적 횟수가 유지됩니다.

```
for (int a=0; a<10; a++) ... d ...
```

네 반복문은 네 자리의 모든 조합을 만듭니다. 각 반복의 조건을 통과한 뒤 본문을 실행하고, 증가식으로 다음 숫자로 이동합니다.

C 비밀번호: 문자 배열과 문자열 비교

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
char attempt[5] = {'0'+a, ..., '\0'};
```

숫자 3에 문자 '0'의 값을 더하면 문자 '3'이 됩니다. 네 글자 뒤에 종료 문자 '₩0'를 넣어야 strcmp가 문자열의 끝을 알 수 있으므로 배열은 5칸입니다.

```
count++; if (strcmp(attempt, secret) == 0)
```

횟수를 늘린 뒤 두 문자열의 내용을 비교합니다. strcmp의 반환값 0은 같다는 뜻입니다. 배열에 ==를 쓰면 문자열 내용을 비교하는 것이 아닙니다.

```
return count; / return 0;
```

찾으면 즉시 함수 전체에서 나옵니다. 모든 후보를 검사한 뒤의 0은 찾지 못했다는 표시입니다. 유효한 네 자리 숫자 입력이면 반드시 찾습니다.

C 비밀번호: 문자 배열과 문자열 비교

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 secret="0007"이면 attempt가 0000부터 0007까지 만들어지고 반환값은 8입니다.

02 attempt[0..3]에는 '0','0','0','7', attempt[4]에는 '₩0'가 들어갑니다.

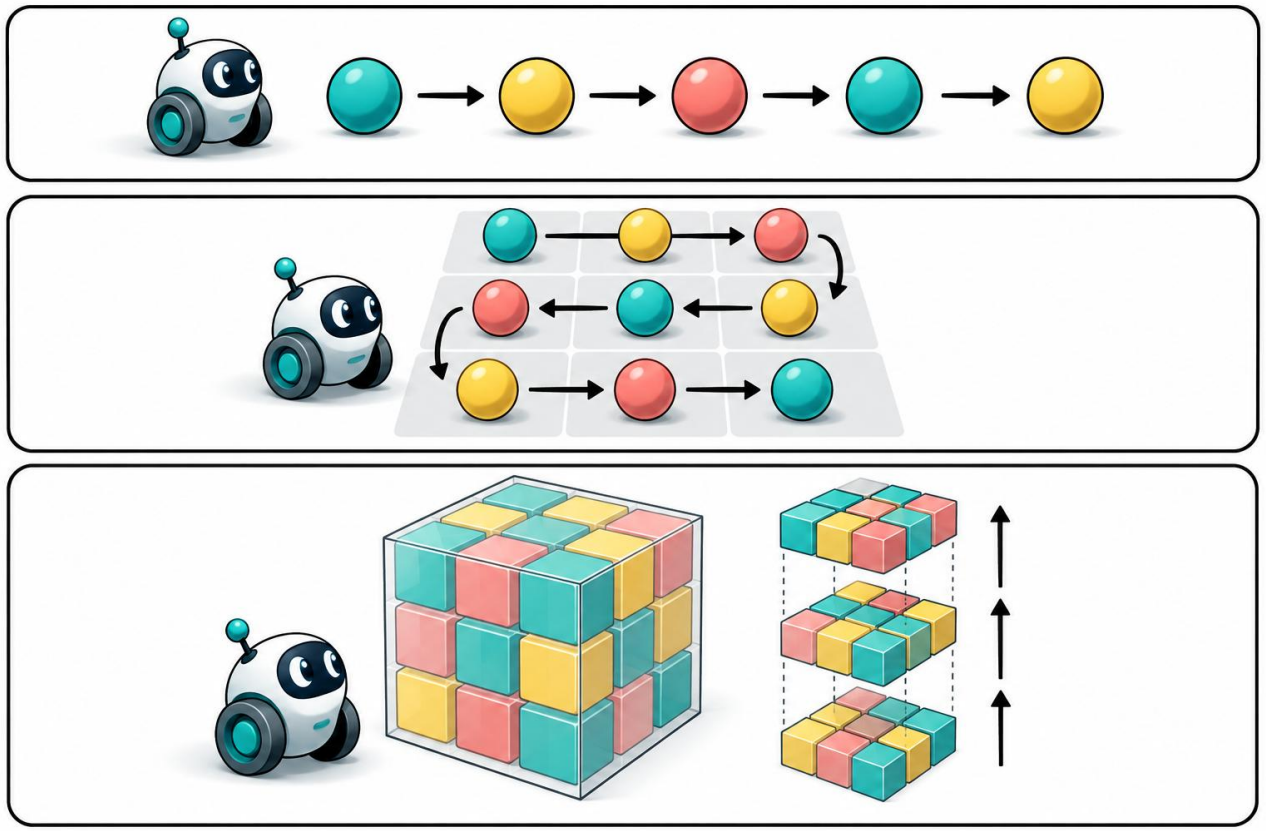
03 함수가 끝나도 호출자 프로그램 전체가 종료되는 것은 아닙니다. 호출자는 반환된 8로 다음 작업을 합니다.

2.2

단순 반복에서 다중 반복으로

찾아야 할 변수의 개수만큼 반복문을 중첩한다: 완전 탐색의 가장 중요한 패턴입니다.

선택 변수가 늘면 반복 범위도 늘어난다



한 축을 선택

한 반복문이 n번: O(n)

두 축을 선택

각각 n번 도는 두 반복문: $O(n^2)$

세 축을 선택

각각 n번 도는 세 반복문: $O(n^3)$

겹수만 세지 말고 각 반복문의 범위를 확인합니다. 6면 주사위 3개의 후보는 항상 216개입니다.

약수 찾기: O(n)

find_divisors.py

```
def find_divisors(n):
    divisors = []
    for i in range(1, n + 1):   # 1부터 n까지 모두 시도
        if n % i == 0:          # 나누어떨어지면 약수!
            divisors.append(i)
    return divisors

print(find_divisors(10))
```

출력 결과

10의 약수:

[1, 2, 5, 10]

시간 복잡도

O(n)

변수 1개 → 반복문 1개

약수 찾기: 후보·조건·저장을 분리하기

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
def find_divisors(n):
```

양의 정수 n을 받아 약수 목록을 돌려주는 함수입니다. 함수를 정의하는 것만으로 반복문이 실행되지는 않으며, 호출해야 실행됩니다.

```
divisors = []
```

찾은 약수를 담을 빈 리스트입니다. 호출할 때마다 새 목록을 만들어 이전 호출의 결과와 섞이지 않게 합니다.

```
for i in range(1, n + 1):
```

약수 후보 1부터 n까지를 전부 봅니다. range의 끝은 제외되므로 n을 포함하려면 n+1이 필요합니다. 0은 나눗셈의 제수가 될 수 있어 제외합니다.

약수 찾기: 후보·조건·저장을 분리하기

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
if n % i == 0:
```

%는 나눗셈의 나머지입니다. 10%2=0이면 나누어떨어지므로 2는 약수이고, 10%3=1이면 약수가 아닙니다.

```
divisors.append(i)
```

조건을 통과한 현재 후보 i를 목록 맨 뒤에 추가합니다. if 안에 있어야 약수만 저장하고, 반복문 안에 있어야 여러 약수를 모읍니다.

```
return divisors / print(find_divisors(10))
```

모든 후보를 다 본 뒤 목록을 반환합니다. print는 호출 결과 [1,2,5,10]을 화면에 보여줍니다. return을 반복문 안에 두면 첫 후보에서 끝납니다.

약수 찾기: 후보·조건·저장을 분리하기

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 n=10: i=1 → [1], i=2 → [1,2], i=3·4 → 변화 없음, i=5 → [1,2,5], i=10 → [1,2,5,10].

02 검사는 정확히 n번입니다. 따라서 시간은 O(n), 결과 저장 공간은 찾은 약수 개수에 비례합니다.

03 확인 문제: range(1,n)으로 바꾸면 자기 자신 n을 누락합니다. n=1에서도 정답은 [1]입니다.

약수 모두 찾기 · C

P14 / out[0..개수-1] = 약수, return = 개수 (용량 10,000)

```c
int find_divisors(int n, int out[]) {
  int count = 0;
  for (int i = 1; i <= n; i++)
    if (n % i == 0)
      out[count++] = i;
  return count;
}
```

코드 읽기

n % i == 0일 때 결과에 추가합니다. n 자체도 포함해야 합니다.

C 약수: 배열과 반환값 두 경로

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
int find_divisors(int n, int out[])
```

n은 양의 정수이고 out은 호출자가 준비한 결과 공간입니다. C에서는 배열 전체를 return하지 않고 out에 쓰고 개수를 반환합니다.

```
int count = 0;
```

저장한 결과 수이자 다음에 쓸 인덱스입니다. 배열 인덱스는 0부터 시작하므로 첫 저장 위치는 out[0]입니다.

```
for (int i=1; i<=n; i++)
```

i=1에서 시작해 n까지 포함합니다. Python range(1,n+1)과 같은 범위이며 한 번 검사한 뒤 i를 1 증가시킵니다.

C 약수: 배열과 반환값 두 경로

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
if (n % i == 0)
```

나머지가 0인 후보만 약수입니다. 중괄호가 없으므로 조건에 속하는 문장은 바로 다음 한 문장입니다.

```
out[count++] = i;
```

현재 count 위치에 i를 넣은 뒤 count가 1 커집니다. out[count]=i; count++;로 나누어 읽으면 됩니다. 처음부터 ++count를 쓰면 0번 칸을 건너뜁니다.

```
return count;
```

결과가 몇 칸인지 알려줍니다. n=10이면 out 앞 네 칸이 1,2,5,10이고 반환값은 4입니다. 호출자는 충분한 배열 용량을 보장해야 합니다.

C 약수: 배열과 반환값 두 경로

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 첫 약수 1: out[0]=1, count=1. 다음 약수 2: out[1]=2, count=2.

02 마지막 약수 10 저장 후 count=4입니다. 출력 반복 범위는 i<count여야 합니다.

03 괄호 }는 반복·함수의 범위를 닫는 표시입니다. 들여쓰기만으로 범위가 정해지는 Python과 다릅니다.

짝(Pair) 찾기: O(n²)

[1, 3, 5, 7, 9] 중 두 수의 합이 10이 되는 쌍 찾기

find_pairs.py

```
def find_pairs(arr, K):
    pairs = []
    for i in range(len(arr) - 1):        # 첫 번째 카드
        for j in range(i + 1, len(arr)): # 두 번째 (중복 방지)
            if arr[i] + arr[j] == K:
                pairs.append((arr[i], arr[j]))
    return pairs

print(find_pairs([1, 3, 5, 7, 9], 10))
```

결과

[(1, 9), (3, 7)]

O(n²)

n = 100 이면

약 10,000번 계산

for 안에 for → 반복문 2겹

두 수의 합: 왜 j는 i+1에서 시작할까?

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
def find_pairs(arr, K): / pairs = []
```

입력 목록과 목표 합을 받아 조건에 맞는 값의 쌍을 모읍니다. 여기서 서로 다른 원소는 서로 다른 위치의 원소를 뜻합니다.

```
for i in range(len(arr) - 1):
```

첫 위치를 0부터 끝에서 두 번째까지 고릅니다. 마지막 위치 뒤에는 두 번째 원소가 없으므로 첫 위치로 볼 필요가 없습니다.

```
for j in range(i + 1, len(arr)):
```

두 번째 위치는 반드시 첫 위치 뒤에서 고릅니다. i<j이므로 자기 자신과 짝짓거나 (i,j),(j,i)를 두 번 세지 않습니다.

두 수의 합: 왜 j는 i+1에서 시작할까?

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
if arr[i] + arr[j] == K:
```

현재 두 위치의 값을 꺼내 합을 비교합니다. 인덱스 i+j를 검사하는 것이 아니라 arr[i]+arr[j]를 검사합니다.

```
pairs.append((arr[i], arr[j]))
```

괄호 안 두 값을 튜플로 묶고 그 튜플 한 개를 결과 목록에 추가합니다. append에 두 개의 별도 인수를 넘기는 형태가 아닙니다.

```
return pairs / print(...)
```

모든 위치 쌍을 검사한 뒤 반환·출력합니다. [1,3,5,7,9], K=10이면 [(1,9),(3,7)]입니다. 5는 한 장뿐이므로 (5,5)는 없습니다.

두 수의 합: 왜 j는 i+1에서 시작할까?

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 n=5이면 i=0에서 j=1,2,3,4 / i=1에서 j=2,3,4 / i=2에서 j=3,4 / i=3에서 j=4입니다.

02 검사 수는 4+3+2+1=10=n(n-1)/2로 O(n²)입니다.

03 arr=[1,1,2], K=3이면 위치가 다른 두 쌍 때문에 (1,2)가 두 번 나옵니다. 값 중복 제거는 별도 요구사항입니다.

합이 K인 두 수 · C

P15 / out[][2] = 값의 쌍, return = 행 수 (최대 435)

```c
int find_pairs(const int A[], int n, int target, int out[][2]) {
  int count = 0;
  for (int i = 0; i < n; i++)
    for (int j = i + 1; j < n; j++)
      if (A[i] + A[j] == target) {
        out[count][0] = A[i];
        out[count++][1] = A[j];
      }
  return count;
}
```

코드 읽기

j를 i+1부터 시작하면 자기 자신과의 쌍, 역순 중복을 제외할 수 있습니다.

C 두 수의 합: 결과 한 행을 채우기

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
int find_pairs(..., int out[][2])
```

입력 A의 길이 n과 목표 target을 받습니다. out은 한 행에 정수 두 칸이 있는 결과 배열이며 반환값은 채운 행 수입니다.

```
int count=0; for (int i=0; i<n; i++)
```

count를 0으로 시작하고 첫 위치를 고릅니다. i=n-1일 때는 내부 반복이 0회이므로 Python의 n-1 상한과 결과는 같습니다.

```
for (int j=i+1; j<n; j++)
```

두 번째 위치를 뒤에서만 골라 두 번 세지 않습니다. 모든 위치 쌍을 정확히 한 번 검사합니다.

C 두 수의 합: 결과 한 행을 채우기

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
if (A[i] + A[j] == target)
```

배열에서 읽은 두 값의 합이 목표일 때만 결과를 씁니다. 조건이 거짓이면 count와 out을 그대로 둡니다.

```
out[count][0] = A[i];
```

현재 결과 행의 첫 번째 칸을 채웁니다. 아직 두 번째 값이 남아 있으므로 이 줄에서는 count를 늘리지 않습니다.

```
out[count++][1] = A[j]; return count;
```

같은 행의 두 번째 칸을 채운 후 다음 행으로 이동합니다. 함수 끝에서 유효한 행 수를 반환합니다. 결과 공간은 최대 n(n-1)/2행이 필요합니다.

C 두 수의 합: 결과 한 행을 채우기

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 A=[1,3,5,7,9], target=10: out[0]=[1,9], out[1]=[3,7], 반환값 2.

02 첫 칸을 쓴 직후 count++를 해 버리면 두 숫자가 서로 다른 행에 들어갑니다.

03 입력 합이 int 범위를 넘지 않는 조건에서 사용하는 코드입니다. 큰 정수 입력은 더 넓은 형으로 계산해야 합니다.

주사위 세 개의 합: O(n³)

dice_sum.py

```
def dice_sum(target):
    results = []
    for d1 in range(1, 7):          # 첫 번째 주사위
        for d2 in range(1, 7):      # 두 번째 주사위
            for d3 in range(1, 7):  # 세 번째 주사위
                if d1 + d2 + d3 == target:
                    results.append((d1, d2, d3))
    return results

print(dice_sum(10))
```

경우의 수 분석

6 × 6 × 6 = 216가지

합=10 → 27가지

O(6³) = O(216)

변수 3개 → 반복문 3개

주사위 세 개: 순서를 구별하는 탐색

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
def dice_sum(target): / results = []
```

목표 합과 같은 주사위 결과들을 모읍니다. 첫째·둘째·셋째 주사위가 구별되므로 (1,3,6)과 (6,3,1)은 다른 결과입니다.

```
for d1 in range(1, 7):
```

첫째 주사위 눈을 1~6에서 고릅니다. 0은 주사위 눈이 아니고 7은 range의 제외되는 끝입니다.

```
for d2 ... / for d3 ...
```

둘째·셋째도 각각 1~6에서 독립적으로 고릅니다. 같은 눈을 여러 번 쓸 수 있으므로 i+1 같은 제한을 넣지 않습니다.

주사위 세 개: 순서를 구별하는 탐색

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
if d1 + d2 + d3 == target:
```

216가지 후보 중 합이 목표인 후보만 통과시킵니다. 반복문 세 개는 후보 생성, if는 조건 검사 역할입니다.

```
results.append((d1, d2, d3))
```

현재 세 눈을 하나의 튜플로 묶어 저장합니다. d3 반복 안이므로 후보를 검사할 때마다 저장 여부를 결정합니다.

```
return results / print(dice_sum(10))
```

모든 후보를 검사한 후 결과 전체를 반환하고 출력합니다. 정답 하나만 찾고 return하면 나머지 가능한 결과를 잃습니다.

주사위 세 개: 순서를 구별하는 탐색

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 처음에는 (1,1,1),(1,1,2),…,(1,1,6), 다음은 (1,2,1)입니다.

02 target=3이면 [(1,1,1)], target=2이면 [], target=10이면 27개입니다.

03 눈의 수를 m으로 일반화하면 m^3번 검사합니다. 눈의 수가 고정된 6이면 검사 수는 항상 216입니다.

세 주사위의 합 · C

P16 / out[][3] = 주사위 눈, return = 행 수 (용량 216)

```c
int dice_sum(int target, int out[][3]) {
  int count = 0;
  for (int a = 1; a <= 6; a++)
    for (int b = 1; b <= 6; b++)
      for (int c = 1; c <= 6; c++)
        if (a + b + c == target) {
          out[count][0] = a;
          out[count][1] = b;
          out[count++][2] = c;
        }
  return count;
}
```

코드 읽기

각 반복문을 1부터 6까지 독립적으로 순회합니다. 합 10의 결과는 27개입니다.

C 주사위: 세 칸을 모두 쓴 다음 증가

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
int dice_sum(int target, int out[][3])
```

한 결과를 세 정수로 저장합니다. out의 한 행이 주사위 세 개의 눈이며 반환값은 결과 행 개수입니다.

```
int count=0; / for a,b,c = 1..6
```

count는 빈 결과에서 시작합니다. a를 고정하고 b를 고정한 상태에서 c가 1~6을 모두 돈 뒤 b가 바뀝니다.

```
if (a + b + c == target)
```

합이 맞는 후보에 대해서만 중괄호 안의 세 저장 문장을 실행합니다. 합이 다르면 다음 후보로 넘어갑니다.

C 주사위: 세 칸을 모두 쓴 다음 증가

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
out[count][0]=a; out[count][1]=b;
```

현재 결과 행의 첫 칸과 둘째 칸에 값을 넣습니다. 두 줄 모두 같은 count를 사용해야 하나의 결과가 됩니다.

```
out[count++][2]=c;
```

마지막 칸을 채우고 count를 증가시킵니다. 다음 정답은 다음 행에 저장됩니다. 최대 216행이면 모든 후보도 담을 수 있습니다.

```
return count;
```

out에서 유효한 행 수를 알려줍니다. 호출자는 0부터 count-1행까지만 사용합니다. 함수의 int 반환값은 주사위 합 자체가 아닙니다.

C 주사위: 세 칸을 모두 쓴 다음 증가

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 target=3: out[0]=[1,1,1], count=1. 나머지 후보는 저장하지 않습니다.

02 target=10의 (1,3,6)과 (3,1,6)은 모두 저장됩니다. 주사위에는 위치 구별이 있기 때문입니다.

03 연습: 세 저장 문장 뒤에 count++를 별도 한 줄로 옮겨 같은 동작인지 확인합니다.

중첩 반복문의 시간 복잡도

각 선택에 n가지 후보가 있고, 독립적인 선택 변수가 k개인 경우

변수 1개	변수 2개	변수 3개	변수 k개
$O(n)$	$O(n^2)$	$O(n^3)$	$O(n^k)$
반복문 1개	반복문 2개	반복문 3개	반복문 k개

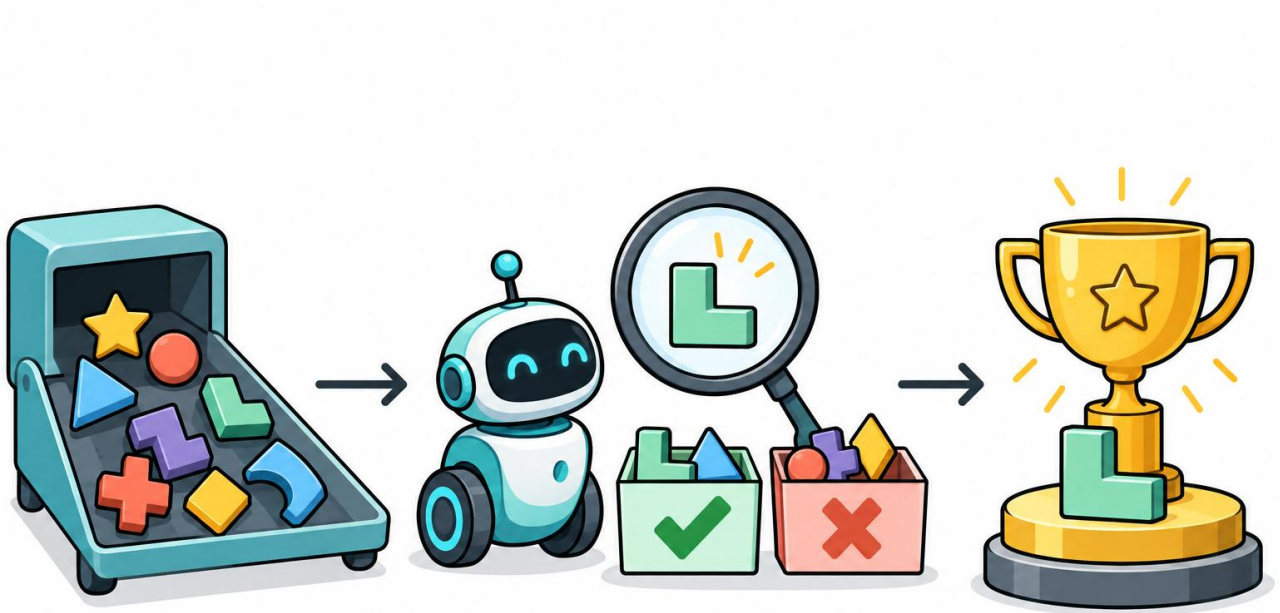
반복문 수와 각 반복문의 범위를 함께 확인합니다.

2.3

완전 탐색의 기본 구조

어떤 완전 탐색 문제든 통하는 3단계 템플릿: 후보 해 생성, 조건 검증, 해 선택.

후보 생성 · 조건 검사 · 해 선택



01 후보 생성

반복문·순열·조합으로
가능한 경우를 만든다

02 조건 검사

합, 무게, 일치 여부 등
문제의 조건을 확인한다

03 해 선택

하나를 반환하거나 모두 저장
또는 최솟값·최댓값을 갱신

최적값 문제는 첫 유효한 후보에서 끝내지 않고, 남은 후보도 확인합니다.

완전 탐색 알고리즘 템플릿

brute_force_template.pseudo

```
function BruteForceSearch(input):
    best_solution ← NULL                        // 최적해 저장 변수

    // [1단계] 후보 해 생성: 가능한 모든 경우 순회
    for each candidate in GenerateAll(input):
        // [2단계] 조건 검증
        if IsValid(candidate) then
            // [3단계] 해 선택
            // (A) 정답 하나만 필요하면:  return candidate
            // (B) 최적해가 필요하면:
            if candidate is better than best_solution then
                best_solution ← candidate
    return best_solution
```

백트래킹 · 분할 정복 · 동적 계획법 모두 이 구조에서 '비효율 탐색을 어떻게 줄일까?'를 고민하며 발전했습니다.

완전 탐색 틀: 첫 정답과 최적 정답의 차이

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
best_solution ← NULL
```

아직 답을 찾지 않았다는 상태입니다. 최소·최대 문제에서 초기값을 정할 때 실제 후보보다 불리한 값이나 미발견 표시가 필요합니다.

```
for each candidate in GenerateAll(input):
```

가능한 후보를 빠짐없이 생성하는 부분입니다. 카드 세 장이면 i<j<k, 부분집합이면 비트마스크처럼 후보의 구조에 맞춰 구현합니다.

```
if IsValid(candidate) then
```

후보가 문제의 제약을 만족하는지 검사합니다. 예를 들어 카드 합이 목표 이하인지 확인합니다. 후보 생성과 유효성 판단은 서로 다른 작업입니다.

완전 탐색 틀: 첫 정답과 최적 정답의 차이

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
return candidate / if candidate is better ...
```

아무 답 하나면 즉시 반환할 수 있습니다. 최댓값을 요구하면 지금 찾은 답보다 좋은 답이 뒤에 있을 수 있어 모든 후보를 검사해야 합니다.

```
best_solution ← candidate
```

지금까지 본 유효 후보 중 가장 좋은 답으로 갱신합니다. 조건에 맞는다는 이유만으로 덮어쓰면 더 나쁜 마지막 답이 남을 수 있습니다.

```
return best_solution
```

반복이 끝나야 전체 후보 중 최적이라고 말할 수 있습니다. 하나도 통과하지 않았다면 미발견 상태를 어떻게 출력할지 정해야 합니다.

완전 탐색 틀: 첫 정답과 최적 정답의 차이

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 목표 21, 후보 합 18→23→20: 18을 저장, 23은 탈락, 20으로 갱신합니다.

02 첫 번째 유효한 합 18에서 끝내면 최댓값 20을 놓칩니다.

03 세 단계로 말해 봅시다: 무엇을 전부 만들고, 어떤 조건으로 거르고, 무엇을 답으로 남기는가?

후보 생성과 최적값 갱신 · C

완전 탐색의 기본 구조를 블랙잭 문제에 적용한 함수 본문입니다.

```c
int best = 0;
for (int i = 0; i < n; i++)
  for (int j = i + 1; j < n; j++)
    for (int k = j + 1; k < n; k++) {
      int sum = cards[i] + cards[j] + cards[k];
      if (sum <= target) { // condition
        if (sum > best)
          best = sum; // best solution
      }
    }
return best;
```

코드 읽기

최적해가 필요하면 모든 유효 후보와 비교합니다. 첫 유효 후보가 최댓값은 아닙니다.

최댓값 갱신: 조건 두 개의 역할

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
int best = 0;
```

양의 카드 값에서 아직 고른 답이 없다는 기준값입니다. 유효한 조합이 없을 때 0을 돌려주는 문제 조건을 가정합니다.

```
for i ... / for j=i+1 ... / for k=j+1 ...
```

i<j<k인 서로 다른 세 위치를 고릅니다. 순서만 다른 같은 카드 조합을 한 번만 검사합니다.

```
int sum = cards[i] + cards[j] + cards[k];
```

현재 후보 세 장의 합을 계산합니다. sum은 매 후보마다 새로 계산하며 이전 후보 합을 누적하지 않습니다.

최댓값 갱신: 조건 두 개의 역할

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
if (sum <= target) {
```

먼저 상한을 넘지 않는지 검사합니다. sum==target만 허용하는 것이 아니므로 목표보다 작은 유효한 답도 포함합니다.

```
if (sum > best) best = sum;
```

유효 후보가 지금까지의 최댓값보다 큰 경우에만 갱신합니다. 더 작은 유효 후보는 best를 낮추지 못합니다.

```
return best;
```

세 반복문이 모두 끝난 뒤 답을 반환합니다. 코드 조각이므로 전체 함수 안에서 사용해야 합니다. main 없이 이 부분만 독립 실행하지 않습니다.

최댓값 갱신: 조건 두 개의 역할

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 cards=[5,6,7,8], target=20: 합은 18,19,20,21 순서이며 best는 0→18→19→20→20입니다.

02 마지막 21은 더 크지만 목표 초과라서 정답이 아닙니다.

03 경우의 수는 nC3, 각 합은 세 번의 값 접근으로 일정한 비용이므로 시간은 $O(n^3)$입니다.

2.4

순열과 조합 · itertools

Python itertools와 C의 반복문·재귀로 순열과 조합을 생성합니다.

itertools로 순열과 조합 생성

변수가 많을수록 반복문이 깊어집니다. itertools는 이 중첩을 한 줄로 대체해 줍니다.

중첩 반복문

before.py

```
n = len(arr)
for i in range(n):
    for j in range(i+1, n):
        for k in range(j+1, n):
            print(arr[i], arr[j], arr[k])
```

4개, 5개로 늘면 코드를 계속 고쳐야 함

itertools 사용

after.py

```
from itertools import combinations

for card in combinations(arr, 3):
    print(card)
# 숫자만 바꾸면 끝!
```

선택할 개수 r만 변경

itertools는 코드를 짧게 만들지만 실행 속도가 빨라지는 것은 아닙니다. 내부적으로는 모든 경우를 생성합니다.

세 장 고르기: 반복문과 combinations 연결

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
n = len(arr) / for i in range(n):
```

입력 원소 수를 구하고 첫 위치를 고릅니다. i가 끝부분이면 내부 반복이 0회가 되므로 잘못된 조합을 만들지는 않습니다.

```
for j in range(i+1, n):
```

둘째 위치를 첫째 뒤에서만 고릅니다. 같은 위치를 다시 고르지 않고 뒤집힌 순서도 생기지 않습니다.

```
for k in range(j+1, n):
```

셋째 위치를 둘째 뒤에서 고릅니다. i<j<k를 만족하는 위치 세 개만 생성합니다.

세 장 고르기: 반복문과 combinations 연결

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
print(arr[i], arr[j], arr[k])
```

현재 세 위치에 저장된 값을 출력합니다. 이 줄까지 도착했다는 것은 서로 다른 위치 세 개를 골랐다는 뜻입니다.

```
from itertools import combinations
```

표준 라이브러리의 조합 생성기를 가져옵니다. 설치가 필요한 외부 패키지가 아니며 위의 위치 선택을 대신 수행합니다.

```
for card in combinations(arr, 3): print(card)
```

후보를 한 개씩 받아 출력합니다. card는 원소 하나가 아니라 선택된 세 값의 튜플입니다. 두 코드의 출력 표시는 다르지만 선택한 조합은 같습니다.

세 장 고르기: 반복문과 combinations 연결

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 arr=[2,4,6,8]: 위치 (0,1,2),(0,1,3),(0,2,3),(1,2,3) → 값 (2,4,6),(2,4,8),(2,6,8),(4,6,8).

02 combinations(arr,3)는 반복 가능한 생성기입니다. list(...)를 쓰면 전부 메모리에 모으고, for로 바로 돌면 한 개씩 처리합니다.

03 원소가 두 개뿐이면 세 장 조합은 0개입니다. 코드는 자동으로 출력 없이 끝납니다.

세 장 선택하기 · C

P17 / out[][3] = 세 장, return = 행 수 (최대 120)

```c
int choose_three(const int A[], int n, int out[][3]) {
  int count = 0;
  for (int i = 0; i < n; i++)
    for (int j = i + 1; j < n; j++)
      for (int k = j + 1; k < n; k++) {
        out[count][0] = A[i];
        out[count][1] = A[j];
        out[count++][2] = A[k];
      }
  return count;
}
```

코드 읽기

j=i+1, k=j+1로 시작합니다. 원소가 세 개보다 적으면 결과는 빈 목록입니다.

C 세 장 선택: i<j<k를 배열로 저장

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
int choose_three(..., int out[][3])
```

입력 배열 A에서 세 위치를 골라 각 결과 행에 세 값을 저장합니다. out의 행 개수는 가능한 결과 수 이상이어야 합니다.

```
int count=0; for (int i=0; i<n; i++)
```

비어 있는 결과에서 시작합니다. 첫 위치를 하나 고르면 안쪽에서 가능한 나머지 두 위치를 모두 고릅니다.

```
for (int j=i+1; j<n; j++)
```

둘째는 첫째 뒤에서 시작합니다. i와 같은 위치가 들어가지 않습니다.

C 세 장 선택: i<j<k를 배열로 저장

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
for (int k=j+1; k<n; k++)
```

셋째는 둘째 뒤에서 시작합니다. 결과적으로 모든 위치 조합이 정확히 한 번 생깁니다.

```
out[count][0]=A[i]; out[count][1]=A[j];
```

같은 결과 행의 첫 두 칸을 채웁니다. count는 결과 행이고 i,j는 입력 위치입니다. 서로 다른 역할을 혼동하지 않습니다.

```
out[count++][2]=A[k]; return count;
```

셋째 값을 채운 후 행 수를 늘립니다. 마지막에 count를 반환하며 n<3이면 내부 본문이 실행되지 않아 0입니다.

C 세 장 선택: i<j<k를 배열로 저장

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 A=[2,4,6,8]이면 결과 네 행이고 반환값은 4입니다.

02 필요한 최대 행 수는 n(n-1)(n-2)/6입니다. 한 행의 폭 3과 전체 행 용량은 별개입니다.

03 직접 써 보기: out[count++][2]를 두 문장으로 풀면 out[count][2]=A[k]; count++;입니다.

순열 (Permutations)

순서가 다르면 다른 경우 → nPr (A, B) ≠ (B, A)

permutations.py

```
from itertools import permutations
items = ['A', 'B', 'C']
# 3개 중 2개를 뽑아 줄 세우기
result = list(permutations(items, 2))
print(len(result))   # 6
print(result)
```

출력 (총 6가지)

[('A','B'),('A','C'),('B','A'),('B','C'),('C','A'),('C','B')]

활용 사례

달리기 1·2·3등 뽑기

비밀번호 (1234 ≠ 4321)

외판원 순회 (방문 순서)

줄 세우기 문제

경우의 수: n! / (n-r)!

Python 순열: 뽑는 순서가 답에 포함된다

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
from itertools import permutations
```

순서 있는 선택을 생성하는 함수를 가져옵니다. 같은 입력 위치는 한 후보 안에서 두 번 선택하지 않습니다.

```
items = ['A', 'B', 'C']
```

입력 순서가 A,B,C인 세 원소입니다. 라이브러리는 이 입력 순서를 기준으로 후보를 차례대로 만듭니다.

```
permutations(items, 2)
```

세 원소 중 두 개를 골라 줄 세웁니다. 첫 자리는 3가지, 둘째는 이미 쓴 원소를 뺀 2가지이므로 6개입니다.

Python 순열: 뽑는 순서가 답에 포함된다

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
result = list(...)
```

생성되는 튜플들을 전부 리스트에 담습니다. n이 크면 결과 자체가 매우 커질 수 있으므로 반드시 list로 모을 필요는 없습니다.

```
print(len(result)); print(result)
```

첫 출력은 후보 수 6이고 다음 출력은 여섯 튜플입니다. ('A','B')와 ('B','A')는 순서가 달라 별도 후보입니다.

Python 순열: 뽑는 순서가 답에 포함된다

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 출력 순서: AB, AC, BA, BC, CA, CB. AA는 같은 위치를 반복 사용해서 나오지 않습니다.

02 길이 r인 순열 수는 n!/(n-r)!입니다. r=n이면 n!개가 됩니다.

03 입력에 값이 중복되면 위치는 달라도 같은 값의 튜플이 나올 수 있습니다. 자동 값 중복 제거 함수가 아닙니다.

직접 만드는 r개 순열 · C (1/2)

P18 / out[][6] = 선택 결과, return = 행 수 (최대 720)

```c
static void perm_visit(const int A[], int n, int r, int depth, int used[],
                       int path[], int out[][6], int *count) {
  if (depth == r) {
    for (int j = 0; j < r; j++)
      out[*count][j] = path[j];
    (*count)++;
    return;
  }
  for (int i = 0; i < n; i++) {
    if (used[i])
      continue;
    used[i] = 1;
    path[depth] = A[i];
    perm_visit(A, n, r, depth + 1, used, path, out, count);
    used[i] = 0;
  }
}
```

다음 장에서 이 보조 함수를 호출하는 공개 함수가 이어집니다.

C 순열 재귀: 선택 → 내려가기 → 되돌리기

줄별 해설 1/3 · 무엇을 하는가 → 왜 필요한가

```
perm_visit(... depth, used, path, out, count)
```

depth는 이미 채운 칸 수입니다. path는 만드는 중인 후보, used[i]는 입력 위치 i를 썼는지, count는 결과 수가 저장된 주소입니다.

```
if (depth == r) {
```

r칸을 모두 채웠으면 후보 하나가 완성된 것입니다. 더 내려가지 않고 저장한 뒤 호출한 곳으로 돌아가는 종료 조건입니다.

```
for (int j=0; j<r; j++) out[*count][j]=path[j];
```

현재 경로의 값을 결과 행에 복사합니다. 경로는 다음 탐색에서 바뀌므로 완성된 순간 별도 결과 공간에 복사해야 합니다.

C 순열 재귀: 선택 → 내려가기 → 되돌리기

줄별 해설 2/3 · 무엇을 하는가 → 왜 필요한가

```
(*count)++; return;
```

주소가 가리키는 결과 수를 1 증가시킨 뒤 이번 재귀 호출을 끝냅니다. 괄호 없는 *count++는 주소 자체를 이동시키므로 다른 코드입니다.

```
for (int i=0; i<n; i++) if (used[i]) continue;
```

다음 칸에 넣을 입력 위치를 처음부터 검사합니다. 이미 쓴 위치라면 이번 후보를 건너뛰어 중복 선택을 막습니다.

```
used[i]=1; path[depth]=A[i];
```

이 위치를 사용 중이라고 표시하고 현재 빈 칸에 값을 넣습니다. 예를 들어 depth=1이면 두 번째 칸에 씁니다.

C 순열 재귀: 선택 → 내려가기 → 되돌리기

줄별 해설 3/3 · 무엇을 하는가 → 왜 필요한가

```
perm_visit(..., depth+1, ..., count);
```

한 칸을 채웠으므로 다음 칸으로 내려갑니다. count는 이미 주소여서 &count를 다시 붙이지 않습니다. 모든 호출이 같은 결과 수를 공유합니다.

```
used[i]=0;
```

자식 호출이 돌아오면 현재 선택을 취소합니다. 다음 형제 후보에서도 이 원소를 사용할 수 있어야 합니다. path의 해당 칸은 다음 선택에서 덮어씁니다.

C 순열 재귀: 선택 → 내려가기 → 되돌리기

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 A=[1,2,3], r=2: 1 선택 → 2 선택 → [1,2] 저장 → 2 해제 → 3 선택 → [1,3] 저장 → 3 해제 → 1 해제.

02 그다음 첫 자리에 2를 고르고 [2,1],[2,3]을 만듭니다. used를 해제하지 않으면 뒤 후보들이 사라집니다.

03 종료 조건의 return은 모든 탐색을 끝내지 않습니다. 한 단계 위로 돌아가 다음 i를 계속 검사합니다.

직접 만드는 r개 순열 · C (2/2)

P18 / out[][6] = 선택 결과, return = 행 수 (최대 720)

```c
int make_permutations(const int A[], int n, int r, int out[][6]) {
  int path[6] = {0}, count = 0;
  int used[6] = {0};
  perm_visit(A, n, r, 0, used, path, out, &count);
  return count;
}
```

코드 읽기

재귀 깊이가 r이면 선택을 저장합니다. used 배열로 사용 여부를 표시하고 재귀가 끝나면 해제합니다.

C 순열 시작 함수: 재귀의 초기 상태

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
int make_permutations(..., int r, int out[][6])
```

호출자가 사용하는 시작 함수입니다. 이 예제 배열은 최대 6개 원소를 전제로 하므로 0≤r≤n≤6 범위를 지켜야 합니다.

```
int path[6]={0}, count=0;
```

만드는 중인 경로 여섯 칸과 결과 개수를 준비합니다. 초기 0이 답에 자동 포함되는 것은 아니며 실제 r칸만 채우고 복사합니다.

```
int used[6]={0};
```

모든 위치가 미사용인 상태입니다. 부분 초기화 문법 {0}은 첫 원소뿐 아니라 나머지도 0으로 초기화합니다.

C 순열 시작 함수: 재귀의 초기 상태

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
perm_visit(A,n,r,0,used,path,out,&count);
```

아직 채운 칸이 없으므로 depth=0으로 시작합니다. &count를 전달해 보조 함수가 이 지역 변수의 값을 수정할 수 있게 합니다.

```
return count;
```

모든 재귀 호출이 끝난 뒤 결과 수를 반환합니다. r=0이면 길이 0인 선택 한 가지가 있어 count=1입니다.

C 순열 시작 함수: 재귀의 초기 상태

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 n=3,r=2이면 시작 count=0 → 여섯 후보 저장 → 반환 6입니다.

02 out은 한 행 6칸인 배열이지만 실제로 읽을 열은 0..r-1입니다. 행 수는 nPr개 이상 확보합니다.

03 주의: 배열 크기 6을 n=10으로 호출한다고 자동 확장하지 않습니다. 입력 제한과 메모리 크기는 함께 설계해야 합니다.

조합 (Combinations)

순서가 달라도 같은 경우 → nCr (A, B) = (B, A)

combinations.py

```
from itertools import combinations
items = ['A', 'B', 'C']
# 3개 중 2개를 선택만 (순서 무관)
result = list(combinations(items, 2))
print(len(result))   # 3
print(result)
```

출력 (총 3가지)

[('A','B'), ('A','C'), ('B','C')]

활용 사례

로또 번호 추첨

대표 선수 3명 선발

메뉴 고르기

부분 집합 만들기

경우의 수: n! / (r!(n−r)!)

Python 조합: 뒤집어도 같은 선택

줄별 해설 1/1 · 무엇을 하는가 → 왜 필요한가

```
from itertools import combinations
```

순서를 구별하지 않는 선택을 생성합니다. 입력 위치가 증가하는 방향으로 고르기 때문에 AB와 BA를 둘 다 만들지 않습니다.

```
items=['A','B','C']; combinations(items,2)
```

세 원소 중 두 개를 고릅니다. A를 고르면 뒤의 B,C가 가능하고 B를 고르면 뒤의 C만 가능합니다.

```
result=list(...); print(len(result)); print(result)
```

결과를 모아 개수와 내용을 출력합니다. 출력은 3과 [('A','B'),('A','C'),('B','C')]입니다.

Python 조합: 뒤집어도 같은 선택

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 순열의 AB·BA는 조합에서는 같은 선택 한 개입니다. nCr=n!/(r!(n-r)!)로 계산합니다.

02 r=0은 빈 선택 한 개, r>n은 선택이 없어 0개입니다.

03 확인 질문: 발표 순서를 정하면 순열이고, 발표할 두 사람만 고르면 조합입니다.

직접 만드는 r개 조합 · C (1/2)

P19 / out[][6] = 선택 결과, return = 행 수 (최대 720)

```c
static void comb_visit(const int A[], int n, int r, int depth, int start,
                       int path[], int out[][6], int *count) {
  if (depth == r) {
    for (int j = 0; j < r; j++)
      out[*count][j] = path[j];
    (*count)++;
    return;
  }
  for (int i = start; i < n; i++) {
    path[depth] = A[i];
    comb_visit(A, n, r, depth + 1, i + 1, path, out, count);
  }
}
```

다음 장에서 이 보조 함수를 호출하는 공개 함수가 이어집니다.

C 조합 재귀: used 대신 start

줄별 해설 1/3 · 무엇을 하는가 → 왜 필요한가

```
comb_visit(... depth, start, path, out, count)
```

depth는 채운 칸 수, start는 다음으로 선택할 수 있는 가장 작은 입력 위치입니다. 이전보다 뒤의 위치만 골라 순서를 중복시키지 않습니다.

```
if (depth == r) {
```

r개를 모두 골랐으므로 후보가 완성됐습니다. 순열과 마찬가지로 저장하고 돌아가는 지점입니다.

```
for (...) out[*count][j]=path[j]; (*count)++;
```

현재 경로 r칸을 결과 행에 복사하고 유효 행 수를 늘립니다. 경로를 다음 후보가 덮어써도 저장된 결과는 유지됩니다.

C 조합 재귀: used 대신 start

줄별 해설 2/3 · 무엇을 하는가 → 왜 필요한가

```
return;
```

완성된 경로에서 더 선택하지 않고 한 단계 위로 돌아갑니다. 호출한 쪽의 for문은 다음 위치를 계속 검사합니다.

```
for (int i=start; i<n; i++)
```

이번 칸의 후보는 start부터 끝까지입니다. 0부터 시작하면 BA 같은 뒤집힌 조합이 다시 생깁니다.

```
path[depth]=A[i];
```

현재 칸에 선택한 값을 기록합니다. 다음 선택에서는 이 칸을 덮어쓰므로 별도 0 초기화로 되돌릴 필요가 없습니다.

C 조합 재귀: used 대신 start

줄별 해설 3/3 · 무엇을 하는가 → 왜 필요한가

```
comb_visit(...,depth+1,i+1,path,out,count);
```

다음 칸은 현재 위치 i보다 뒤에서 시작합니다. i+1이 중복 선택과 순서 중복을 동시에 막으므로 used 배열이 필요 없습니다.

C 조합 재귀: used 대신 start

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 A=[1,2,3], r=2: 1 선택(start=1)→2,3 → [1,2],[1,3]; 2 선택(start=2)→3 → [2,3].

02 3을 첫 값으로 고르면 다음 start=3이고 i<n을 만족하지 않아 아무 후보도 추가하지 않습니다.

03 더 효율적으로 남은 칸 수를 고려해 반복 상한을 줄일 수 있지만, 현재 코드는 이해하기 쉬운 기본형입니다.

직접 만드는 r개 조합 · C (2/2)

P19 / out[][6] = 선택 결과, return = 행 수 (최대 720)

```c
int make_combinations(const int A[], int n, int r, int out[][6]) {
  int path[6] = {0}, count = 0;
  comb_visit(A, n, r, 0, 0, path, out, &count);
  return count;
}
```

코드 읽기

재귀 깊이가 r이면 선택을 저장합니다. 다음 탐색의 시작 인덱스를 i+1로 전달합니다.

C 조합: 빈 경로에서 탐색 시작

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
int make_combinations(..., int out[][6])
```

조합을 요청하는 공개 함수입니다. 고정 배열 예제이므로 0≤r≤n≤6과 충분한 결과 행 용량을 가정합니다.

```
int path[6]={0}, count=0;
```

경로와 결과 수를 준비합니다. 순열과 달리 used 배열은 만들지 않습니다. 증가하는 위치만 선택하기 때문입니다
.

```
comb_visit(A,n,r,0,0,path,out,&count);
```

첫 번째 0은 채운 칸 수 depth, 두 번째 0은 시작 위치 start입니다. 비어 있는 경로에서 입력의 처음부터 선택합
니다.

C 조합: 빈 경로에서 탐색 시작

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
return count;
```

보조 함수가 수정한 결과 수를 반환합니다. n=4,r=2라면 여섯 결과 행을 사용합니다.

C 조합: 빈 경로에서 탐색 시작

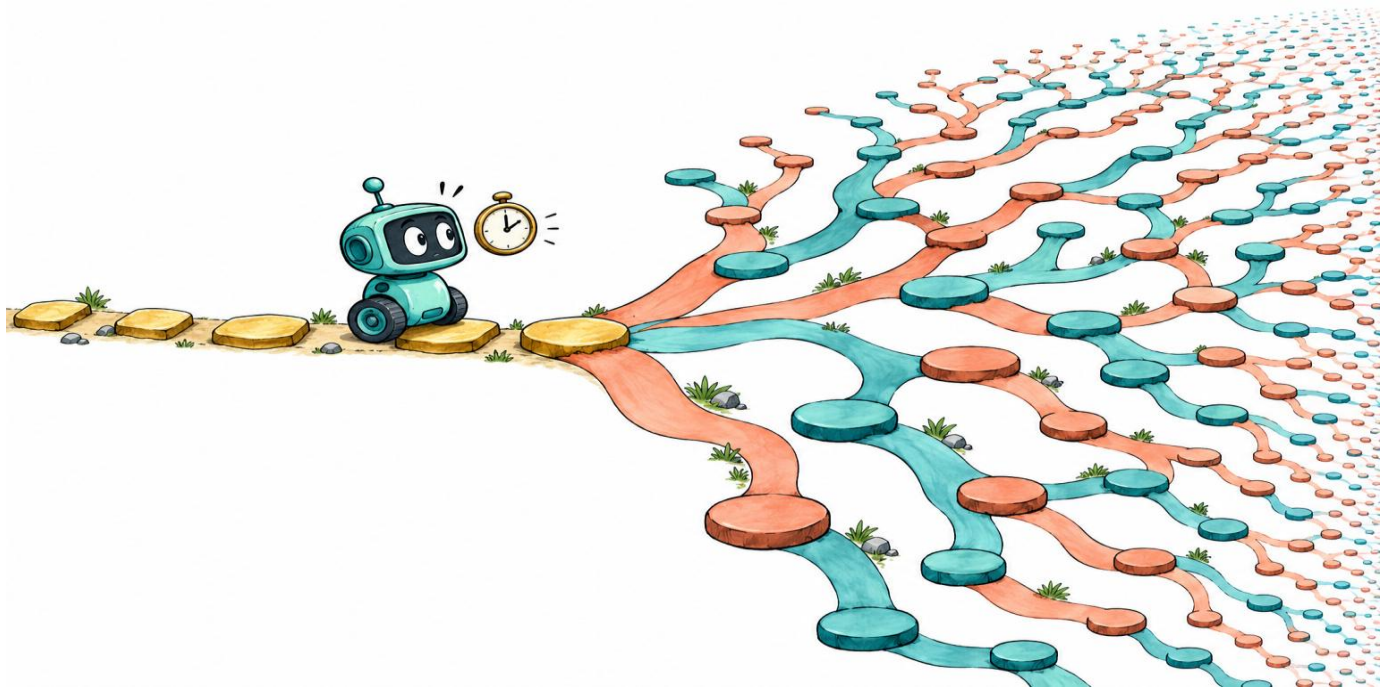
실행 추적 · 결과 검산 · 자주 틀리는 지점

01 n=4,r=2의 위치 쌍은 01,02,03,12,13,23입니다.

02 순열 12개와 달리 조합은 6개입니다. 같은 두 위치를 뒤집은 경우를 제외했기 때문입니다.

03 r=0일 때도 빈 선택 하나가 답이므로 1을 반환합니다. 빈 선택과 답 없음은 다릅니다.

순서가 바뀌면 다른 결과일까?



순열: 자리가 의미를 가진다

같은 세 차라도 1·2·3등이
바뀌면 다른 결과입니다.

`nPr = n! / (n−r)!`

조합: 선택한 묶음이 중요하다

딸기·초코 = 초코·딸기
같은 두 맛의 조합입니다.

`nCr = n! / (r! × (n−r)!)`

서로 다른 n개 중 r개를 중복 없이 선택할 때의 공식입니다.

비밀번호 해킹 (순열 활용)

1, 2, 3을 한 번씩 사용한 3자리 비밀번호 후보를 모두 만든다.

password_perm.py

```
from itertools import permutations
nums = [1, 2, 3]
# 3개 숫자를 모두 사용해 순서 있게 나열
passwords = list(permutations(nums, 3))
for pw in passwords:
    print(pw)
```

출력 결과 (3! = 6가지)

(1,2,3)	(1,3,2)
(2,1,3)	(2,3,1)
(3,1,2)	(3,2,1)

총 3! = 6가지 후보를 생성 → 모두 시도하면 반드시 정답을 찾습니다.

비밀번호 순열: 모든 숫자의 순서 바꾸기

줄별 해설 1/1 · 무엇을 하는가 → 왜 필요한가

```
from itertools import permutations; nums=[1,2,3]
```

입력 세 숫자를 각각 한 번씩 사용하는 예제입니다. 같은 숫자를 반복 허용하는 네 자리 자물쇠 예제와 후보 생성 규칙이 다릅니다.

```
passwords=list(permutations(nums,3))
```

세 개를 모두 골라 배열하는 3!=6개 후보를 만듭니다. 각 후보는 정수 세 개를 담은 튜플이지 세 자리 정수 하나가 아닙니다.

```
for pw in passwords: print(pw)
```

튜플을 하나씩 받아 출력합니다. (1,2,3)을 문자열 "123"으로 쓰려면 각 값을 문자로 바꿔 이어 붙이는 별도 과정이 필요합니다.

비밀번호 순열: 모든 숫자의 순서 바꾸기

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 출력 순서: (1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1).

02 111은 같은 위치를 재사용하므로 나오지 않습니다. 반복 허용 비밀번호라면 다른 생성 방식이 필요합니다.

03 튜플 1,2,3을 수 123으로 만들 때는 100×1+10×2+3처럼 자릿값을 이용합니다.

1·2·3 비밀번호 후보 · C

P20 / out = 비밀번호 6개, return = 6

```c
int password_candidates(const int A[], int n, int out[]) {
  int count = 0;
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      for (int k = 0; k < n; k++)
        if (i != j && i != k && j != k)
          out[count++] = 100 * A[i] + 10 * A[j] + A[k];
  return count;
}
```

코드 읽기

서로 다른 i,j,k만 사용하고 100*A[i]+10*A[j]+A[k]로 조립합니다.

C 비밀번호 후보: 자릿값으로 정수 만들기

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
int password_candidates(..., int out[]) / count=0
```

입력은 서로 다른 한 자리 숫자라고 가정합니다. 결과는 숫자 세 개의 배열 대신 세 자리 모양의 정수로 저장합니다.

```
for i=0..n-1 / for j=0..n-1 / for k=0..n-1
```

세 위치를 독립적으로 고릅니다. 이 상태에서는 같은 위치를 반복 고른 후보도 있으므로 뒤의 조건 검사가 필요합니다.

```
if (i!=j && i!=k && j!=k)
```

세 위치가 모두 다를 때만 통과합니다. &&는 모든 조건이 참이어야 한다는 뜻입니다. i<j<k로 제한하면 순열이 아니라 조합이 됩니다.

C 비밀번호 후보: 자릿값으로 정수 만들기

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
out[count++] = 100*A[i] + 10*A[j] + A[k];
```

첫 숫자를 백의 자리, 둘째를 십의 자리, 셋째를 일의 자리에 넣습니다. 결과를 저장한 다음 count를 늘립니다.

```
return count;
```

만들어진 순서 있는 후보 수를 반환합니다. n=3이면 6개입니다. 입력에 0이 있으면 012가 정수 12로 저장되어 앞의 0은 사라집니다.

C 비밀번호 후보: 자릿값으로 정수 만들기

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 A=[1,2,3] → out=[123,132,213,231,312,321], 반환값 6.

02 세 반복은 n^3번 후보를 검사하고 서로 다른 위치만 저장합니다. 값이 중복된 입력은 동일 정수를 여러 번 만들 수 있습니다.

03 앞의 0을 보존해야 하는 비밀번호라면 정수 대신 문자 배열을 사용해야 합니다.

아이스크림 메뉴 선택 (조합)

4가지 맛 중 2가지를 선택해 더블 컵을 만든다. (순서 무관)

icecream_comb.py

```
from itertools import combinations
flavors = ['초코', '바닐라', '딸기', '멜론']
# 4개 중 2개를 선택만 (순서 무관)
menus = list(combinations(flavors, 2))
for menu in menus:
    print(menu)
```

출력 결과 (4C2 = 6가지)

초코·바닐라	초코·딸기
초코·멜론	바닐라·딸기
바닐라·멜론	딸기·멜론

'초코+바닐라'와 '바닐라+초코'는 같은 메뉴 → 조합(combination)을 사용합니다.

아이스크림 조합: 이름 두 개를 한 후보로

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
from itertools import combinations
```

두 맛의 순서를 구별하지 않는 메뉴를 생성합니다. 초코+바닐라와 바닐라+초코는 같은 메뉴로 취급합니다.

```
flavors=['초코','바닐라','딸기','멜론']
```

네 개의 맛을 입력 순서대로 저장합니다. 여기서는 같은 맛 두 번 선택을 허용하지 않습니다.

```
menus=list(combinations(flavors,2))
```

네 맛 중 서로 다른 두 위치를 골라 4C2=6개 튜플을 만듭니다. 한 튜플이 한 메뉴입니다.

아이스크림 조합: 이름 두 개를 한 후보로

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
for menu in menus: print(menu)
```

여섯 메뉴를 하나씩 출력합니다. 메뉴를 미리 전부 저장할 필요가 없다면 combinations를 for에 바로 넣어도 됩니다.

아이스크림 조합: 이름 두 개를 한 후보로

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 초코·바닐라 / 초코·딸기 / 초코·멜론 / 바닐라·딸기 / 바닐라·멜론 / 딸기·멜론.

02 초코·초코는 나오지 않습니다. 동일 맛 중복을 허용한다면 문제 조건과 생성기를 바꿔야 합니다.

03 이 예제와 발표자 두 명 선택은 후보의 구조가 같은 조합 문제입니다.

아이스크림 두 가지 맛 · C

P21 / out[][2] = 맛 번호, return = 메뉴 수 (최대 190)

```c
int icecream_menus(int n, int out[][2]) {
  int count = 0;
  for (int i = 0; i < n; i++)
    for (int j = i + 1; j < n; j++) {
      out[count][0] = i;
      out[count++][1] = j;
    }
  return count;
}
```

코드 읽기

n개 중 2개를 고르면 n*(n-1)/2개의 메뉴가 만들어집니다.

C 메뉴: 문자열 대신 맛의 번호 저장

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
int icecream_menus(int n, int out[][2]); count=0
```

맛 이름 대신 0~n-1의 번호를 사용합니다. 반환하는 결과 행의 두 값은 맛 자체가 아니라 맛의 인덱스입니다.

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

첫 맛을 고릅니다. i=n-1에서는 뒤에 맛이 없으므로 내부 반복이 실행되지 않습니다.

```
for (int j=i+1; j<n; j++)
```

두 번째 맛을 뒤에서만 고릅니다. 같은 맛 두 번 선택과 뒤집힌 중복을 함께 제외합니다.

C 메뉴: 문자열 대신 맛의 번호 저장

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
out[count][0]=i; out[count++][1]=j;
```

두 번호를 같은 행에 저장한 뒤 결과 수를 늘립니다. 첫 줄에서 증가시키지 않는 이유는 두 번째 칸도 같은 행에 써야 하기 때문입니다.

```
return count;
```

만든 메뉴 개수를 반환합니다. n=4라면 6이고 호출자는 번호를 flavors 같은 이름 배열과 연결해 화면에 표시할 수 있습니다.

C 메뉴: 문자열 대신 맛의 번호 저장

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 out[0]=[0,1]은 초코·바닐라, out[5]=[2,3]은 딸기·멜론입니다.

02 번호를 저장하면 알고리즘은 문자열 복사 없이 선택 구조에 집중할 수 있습니다.

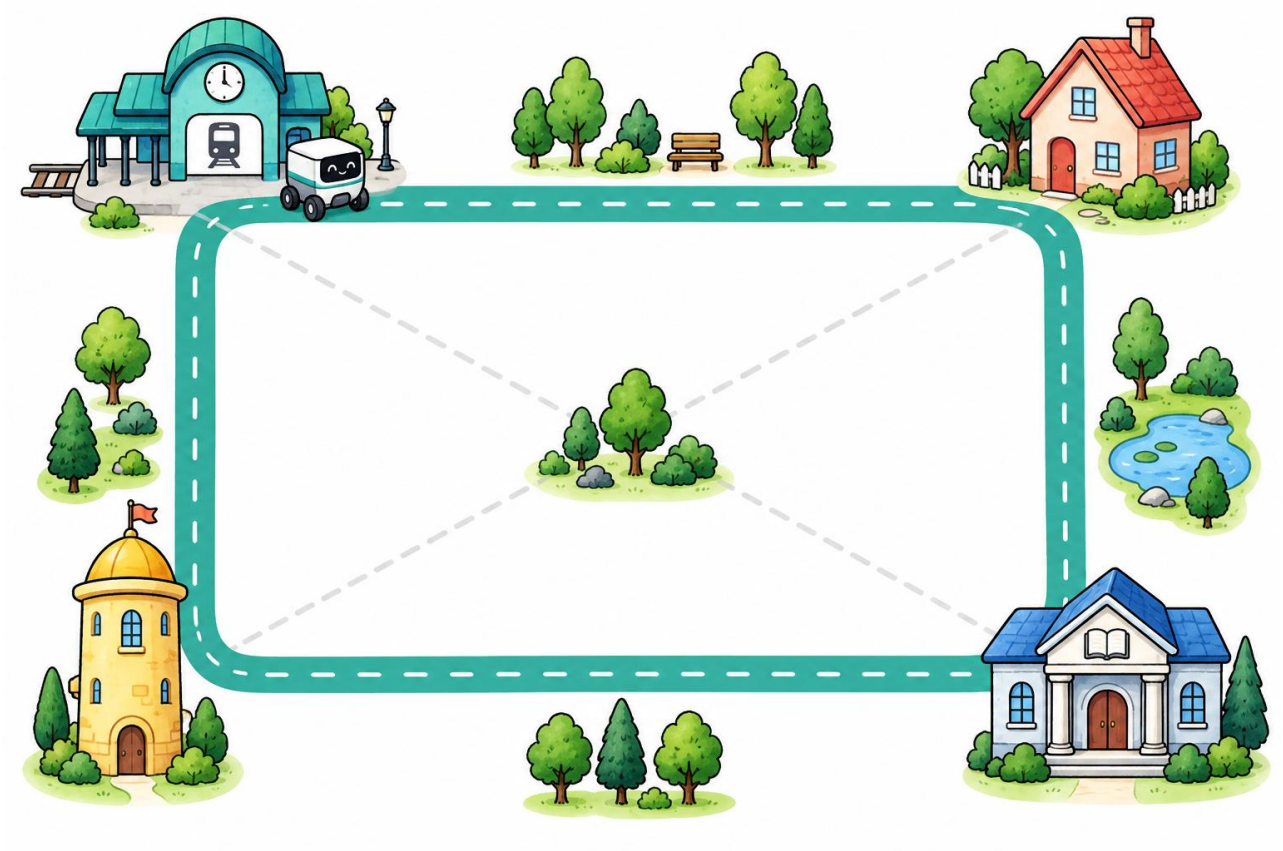
03 0부터 시작하는 배열 번호와 사람이 말하는 첫 번째·두 번째를 구별해서 설명해 봅시다.

2.5

예제로 배우는 완전 탐색

문자열 매칭, 최근접 쌍, 부분집합 생성: 완전 탐색의 전형적인 예제들을 살펴봅니다.

검색 창을 한 칸씩 옮겨 비교하기



후보는 시작 위치

패턴을 놓을 수 있는 위치를
왼쪽부터 하나씩 선택

창 안의 문자를 비교

하나라도 다르면
다음 시작 위치로 이동

맞는 위치를 모두 저장

첫 일치 뒤에도 계속 검사
겹쳐 나타나는 패턴도 확인

문자열 길이 n, 패턴 길이 m일 때 시작 위치는 0부터 n−m까지입니다. (0 < m ≤ n)

무차별 문자열 매칭

H E L L O ␣ W O R L D

i=0: L≠H i=1: L≠E i=2: LLO = LLO 발견

bf_string_match.py

```
def bf_string_match(text, pattern):
    n, m = len(text), len(pattern)
    found = []
    for i in range(n - m + 1):        # 가능한 모든 시작 위치
        match = True
        for j in range(m):            # 패턴 문자 하나씩 비교
            if text[i + j] != pattern[j]:
                match = False; break # 불일치 시 즉시 중단
        if match: found.append(i)
    return found                      # O(n*m)
```

문자열 검색: 시작 위치 i와 패턴 위치 j

줄별 해설 1/3 · 무엇을 하는가 → 왜 필요한가

```
def bf_string_match(text, pattern):
```

text 안에서 pattern이 등장하는 모든 시작 인덱스를 찾습니다. 같은 문자가 겹쳐 등장하는 경우도 포함하는 검색입니다.

```
n,m=len(text),len(pattern); found=[]
```

두 문자열 길이와 결과 목록을 준비합니다. n은 전체 길이, m은 비교할 패턴 길이로 서로 역할이 다릅니다.

```
for i in range(n-m+1):
```

시작할 수 있는 마지막 인덱스는 n-m입니다. 끝 위치 i+m-1이 n-1을 넘지 않아야 하므로 가능한 시작점은 0부터 n-m까지입니다.

문자열 검색: 시작 위치 i와 패턴 위치 j

줄별 해설 2/3 · 무엇을 하는가 → 왜 필요한가

```
match=True / for j in range(m):
```

현재 시작점에서는 일단 일치한다고 가정한 뒤 패턴의 0~m-1번째 문자를 확인합니다. match는 새 시작점마다 True로 재설정해야 합니다.

```
if text[i+j] != pattern[j]:
```

텍스트에서는 시작점 i만큼 이동한 위치, 패턴에서는 j를 읽습니다. i=2,j=1이면 text[3]과 pattern[1]을 비교합니다.

```
match=False; break
```

한 글자라도 다르면 현재 시작점은 탈락입니다. break는 j 반복만 끝내므로 이후 다음 시작점 i의 검사는 계속됩니다.

문자열 검색: 시작 위치 i와 패턴 위치 j

줄별 해설 3/3 · 무엇을 하는가 → 왜 필요한가

```
if match: found.append(i)
```

모든 문자를 통과했을 때 시작점 i를 저장합니다. 이 if는 j 반복 밖, i 반복 안에 있어야 한 시작점에 대해 한 번 판단합니다.

```
return found
```

모든 시작점을 검사한 뒤 목록을 반환합니다. 패턴이 더 길면 반복이 0회이고, 빈 패턴이면 이 구현은 0..n의 모든 경계를 반환합니다.

문자열 검색: 시작 위치 i와 패턴 위치 j

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 text="ababc", pattern="abc": i=0에서 a=a,b=b,a≠c로 탈락; i=1은 b≠a로 탈락; i=2는 a=a,b=b,c=c라서 2 저장.

02 비교 횟수는 최악에 (n-m+1)×m으로 O(nm)입니다. 불일치가 빠르면 실제 비교는 더 적습니다.

03 text="aaaa", pattern="aa"의 답은 [0,1,2]입니다. 성공 후 i를 m만큼 건너뛰면 겹치는 답을 놓칩니다.

문자열 매칭: 분석과 발전

run.py

```
T = "ababcabcababd"
P = "abc"
print(bf_string_match(T, P))
# 출력: [2, 5]  ← 패턴 등장 위치
```

시간 복잡도 분석

시작 위치: 최대 n-m+1개
각 위치: 최대 m번 비교
최악: (n-m+1) × m ≈ O(n×m)

완전 탐색 → 고급 알고리즘으로의 발전

Brute Force	O(nm)	KMP	O(n+m)	Boyer-Moore	최선: O(n/m)

DNA 시퀀싱에도 활용! 유전자 배열 ACGACAC...에서 특정 패턴 ACATA를 찾는 작업이 대표적입니다.

검색 예제: 출력 인덱스를 직접 검산

줄별 해설 1/1 · 무엇을 하는가 → 왜 필요한가

```
T = "ababcabcababd"
```

인덱스는 0부터 시작합니다. 앞부분은 0:a,1:b,2:a,3:b,4:c,5:a,6:b,7:c입니다.

```
P = "abc"
```

패턴 길이는 3입니다. 후보 시작점에서 연속된 세 글자가 정확히 a,b,c여야 통과합니다.

```
print(bf_string_match(T,P))
```

i=2와 i=5에서만 세 문자가 모두 일치하므로 [2,5]를 출력합니다. 두 번째와 다섯 번째 글자라는 뜻이 아니라 0부터 센 인덱스입니다.

검색 예제: 출력 인덱스를 직접 검산

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 T[2:5]="abc", T[5:8]="abc"로 결과를 다시 확인할 수 있습니다. 슬라이스의 끝은 제외됩니다.

02 T[0:3]="aba"이므로 두 글자가 같아도 정답이 아닙니다.

03 학생 확인: 일치한 문자열 자체를 반환하려면 어디를 바꿔야 할까요? 현재 코드는 found.append(i)로 위치를 저장합니다.

문자열의 모든 일치 위치 · C

P22 / out = 시작 인덱스, return = 개수 (용량 201)

```c
int bf_string_match(const char text[], const char pattern[], int out[]) {
  int n = (int)strlen(text), m = (int)strlen(pattern), count = 0;
  for (int i = 0; i <= n - m; i++) {
    int j = 0;
    while (j < m && text[i + j] == pattern[j])
      j++;
    if (j == m)
      out[count++] = i;
  }
  return count;
}
```

코드 읽기

시작 위치 i는 n-m까지입니다. C의 strlen 결과는 int에 담아 m>n일 때 음수 범위를 안전하게 처리하세요.

C 문자열 검색: j가 끝까지 갔는지 확인

줄별 해설 1/3 · 무엇을 하는가 → 왜 필요한가

```
int bf_string_match(..., int out[])
```

text와 pattern은 '₩0'로 끝나는 문자열입니다. 찾은 시작 위치를 out에 저장하고 개수를 반환합니다. strlen 선언을 위해 string.h가 필요합니다.

```
int n=(int)strlen(text), m=(int)strlen(pattern), count=0;
```

길이와 결과 수를 준비합니다. 이 수업 입력은 int에 담을 수 있는 길이라고 가정합니다. 문자열 종료 문자 자체는 길이에 포함되지 않습니다.

```
for (int i=0; i<=n-m; i++)
```

패턴이 텍스트를 벗어나지 않는 시작 위치만 검사합니다. i=n-m도 유효하므로 <=입니다.

C 문자열 검색: j가 끝까지 갔는지 확인

줄별 해설 2/3 · 무엇을 하는가 → 왜 필요한가

```
int j=0;
```

새 시작 위치에서 패턴 처음부터 비교합니다. 이전 시작점에서의 j를 이어 쓰면 일부 문자를 건너뛰게 됩니다.

```
while (j<m && text[i+j]==pattern[j]) j++;
```

범위 안이고 글자도 같을 때만 다음 글자로 갑니다. &&는 왼쪽이 거짓이면 오른쪽을 평가하지 않으므로 끝을 넘은 접근을 막습니다.

```
if (j==m) out[count++]=i;
```

패턴 길이만큼 일치했다면 시작 위치를 저장합니다. 중간 불일치이면 j<m이므로 저장하지 않습니다.

C 문자열 검색: j가 끝까지 갔는지 확인

줄별 해설 3/3 · 무엇을 하는가 → 왜 필요한가

```
return count;
```

저장된 위치 수를 돌려줍니다. out에는 최악에 n+1개 위치가 필요할 수 있습니다. 빈 패턴에 대한 이 구현의 결과도 경계 n+1개입니다.

C 문자열 검색: j가 끝까지 갔는지 확인

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 text="ababc", pattern="abc": i=0에서 j=2에 멈춤, i=1에서 j=0에 멈춤, i=2에서 j=3에 도달합니다.

02 반환값은 1이고 out[0]=2입니다. Python의 [2]와 같은 결과를 배열+개수로 표현했습니다.

03 while 조건의 ==를 !=로 바꾸면 일치하는 동안 진행한다는 의미가 뒤집힙니다.

가장 가까운 두 수 (Closest Pair)

[10, 3, 22, 15, 17, 9] → 차이가 가장 작은 두 수는? 모든 쌍을 검사하는 전형적 완전 탐색.

closest_pair.py

```
def closest_pair(arr):
    min_diff = float('inf')          # 무한대로 초기화
    best = None
    for i in range(len(arr) - 1):
        for j in range(i + 1, len(arr)):
            diff = abs(arr[i] - arr[j])
            if diff < min_diff:
                min_diff = diff
                best = (arr[i], arr[j])
    return best, min_diff
```

분할 정복의 '최근접 점 문제'로 발전 → O(n log n)으로 개선

실행 결과

[10,3,22,15,17,9]

↓

((10, 9), 1)

O(n²)

모든 쌍 비교

가장 가까운 두 수: 최솟값과 쌍을 함께 갱신

줄별 해설 1/3 · 무엇을 하는가 → 왜 필요한가

```
def closest_pair(arr):
```

서로 다른 두 위치의 값 차이가 가장 작은 쌍을 찾습니다. 유효한 쌍을 요구한다면 원소가 최소 두 개라는 조건이 필요합니다.

```
min_diff=float('inf'); best=None
```

처음에는 어떤 유한한 차이도 갱신할 수 있게 무한대로 시작합니다. best=None은 아직 쌍을 고르지 않았다는 표시입니다.

```
for i ... / for j in range(i+1,len(arr)):
```

i<j인 위치 쌍을 모두 검사합니다. 같은 위치끼리의 차이 0을 잘못 정답으로 고르지 않게 합니다.

가장 가까운 두 수: 최솟값과 쌍을 함께 갱신

줄별 해설 2/3 · 무엇을 하는가 → 왜 필요한가

```
diff=abs(arr[i]-arr[j])
```

값의 순서와 관계없는 거리를 원하므로 절댓값을 구합니다. 8-3과 3-8은 모두 거리 5입니다.

```
if diff<min_diff: min_diff=diff
```

더 작은 차이를 발견한 경우에만 기록을 바꿉니다. <=가 아니라 <이므로 동률이면 먼저 찾은 쌍을 유지합니다.

```
best=(arr[i],arr[j])
```

최솟값을 만든 두 값을 함께 저장합니다. 차이만 갱신하고 쌍을 갱신하지 않으면 결과가 서로 맞지 않게 됩니다.

가장 가까운 두 수: 최솟값과 쌍을 함께 갱신

줄별 해설 3/3 · 무엇을 하는가 → 왜 필요한가

```
return best,min_diff
```

쌍과 차이를 함께 반환합니다. 원소가 두 개보다 적으면 (None,inf)가 되므로 실제 사용에서는 입력 조건이나 예외 처리를 정해야 합니다.

가장 가까운 두 수: 최솟값과 쌍을 함께 갱신

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 arr=[8,3,6]: (8,3) 차이5 → best=(8,3); (8,6) 차이2 → best=(8,6); (3,6) 차이3 → 유지.

02 최종 반환은 ((8,6),2)입니다. 정렬 없이도 모든 쌍을 검사해 답을 구했습니다.

03 후보 수는 nC2이므로 O(n²), 결과 저장 외 추가 공간은 O(1)입니다.

가장 가까운 두 수 · C

P23 / out[0]=첫 값, out[1]=둘째 값, out[2]=최소 차이

```c
void closest_pair(const int A[], int n, long long out[3]) {
  out[2] = LLONG_MAX;
  for (int i = 0; i < n; i++)
    for (int j = i + 1; j < n; j++) {
      long long diff = llabs((long long)A[i] - A[j]);
      if (diff < out[2]) {
        out[0] = A[i];
        out[1] = A[j];
        out[2] = diff;
      }
    }
}
```

코드 읽기

C: llabs((long long)A[i]-A[j]). 갱신 조건은 <=가 아닌 <입니다.

C 가장 가까운 쌍: 형 변환을 먼저 하는 이유

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
void closest_pair(..., long long out[3])
```

반환값 대신 out 세 칸에 첫 값·둘째 값·최소 차이를 씁니다. n≥2이고 입력은 int 범위라고 가정합니다.

```
out[2]=LLONG_MAX;
```

가능한 가장 큰 long long 값으로 시작해 첫 실제 차이가 갱신되게 합니다. limits.h의 상수입니다.

```
for i ... / for j=i+1 ...
```

서로 다른 위치 쌍을 한 번씩 검사합니다. 배열 길이가 두 개보다 작으면 쌍 두 칸이 채워지지 않으므로 입력 제한이 중요합니다.

C 가장 가까운 쌍: 형 변환을 먼저 하는 이유

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
long long diff=llabs((long long)A[i]-A[j]);
```

먼저 A[i]를 넓은 형으로 바꾸고 빼야 int 뺄셈의 넘침을 피합니다. 뺄셈을 끝낸 뒤 변환하면 이미 발생한 넘침을 복구할 수 없습니다.

```
if (diff<out[2]) {
```

지금 기록보다 더 작은 차이만 채택합니다. 동률에서는 기존 쌍을 유지하는 규칙입니다.

```
out[0]=A[i]; out[1]=A[j]; out[2]=diff;
```

쌍과 그 차이를 세 칸에 함께 씁니다. 함수가 void이므로 호출자는 out 배열을 읽어 결과를 얻습니다.

C 가장 가까운 쌍: 형 변환을 먼저 하는 이유

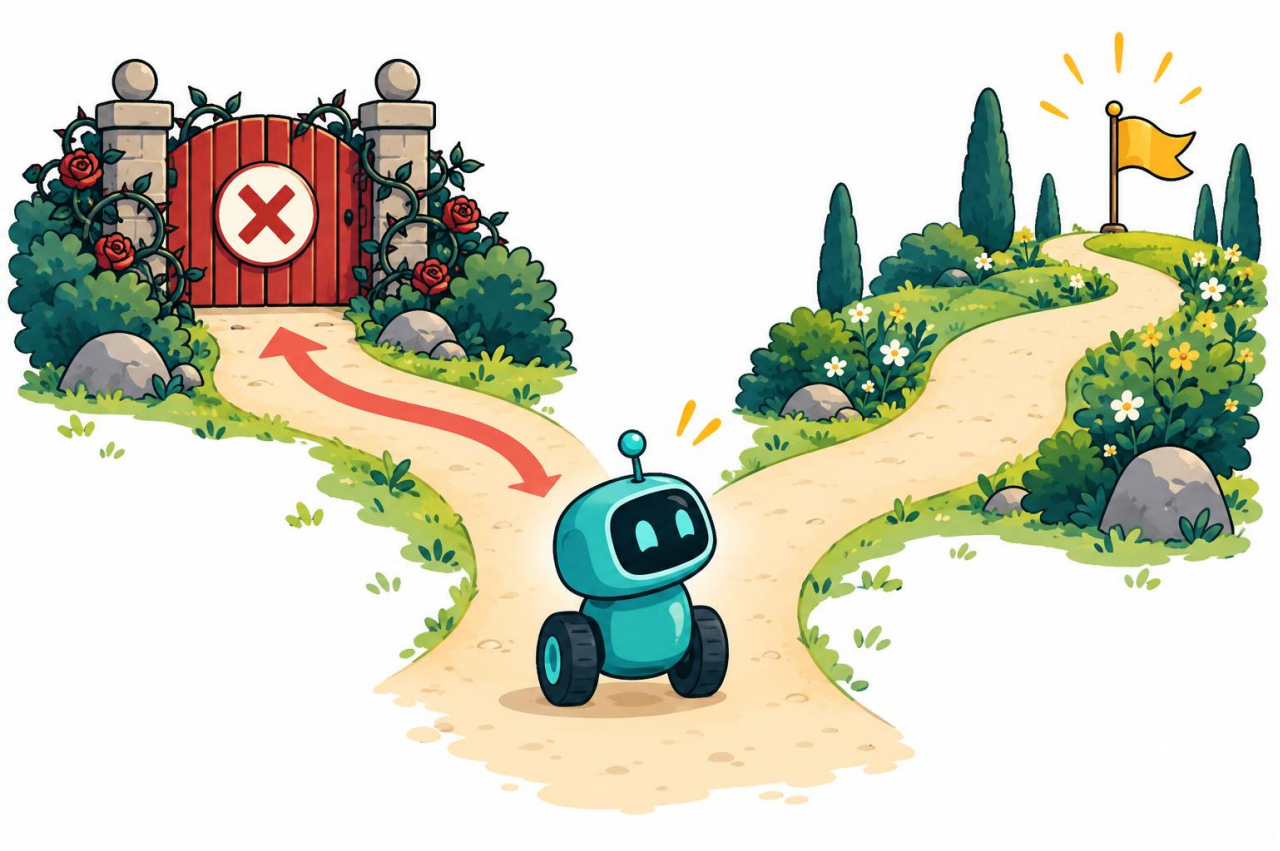
실행 추적 · 결과 검산 · 자주 틀리는 지점

01 A=[8,3,6]이면 최종 out=[8,6,2]입니다.

02 int 최솟값과 최댓값의 차이는 int 범위를 넘을 수 있습니다. 그래서 차이를 long long으로 계산합니다.

03 llabs를 선언하는 stdlib.h와 LLONG_MAX를 선언하는 limits.h를 포함해야 합니다.

부분집합은 원소마다 켜고 끄는 선택



넣기 또는 빼기

각 물건마다 독립적으로
두 가지 중 하나를 선택

물건 3개라면 $2^3 = 8$가지

아무것도 고르지 않는 경우와
모두 고르는 경우도 포함

비트 하나가 물건 하나

1이면 포함, 0이면 제외
마스크 하나가 부분집합 하나

그림의 사과와 물병을 고르고 손전등을 제외한 모습은 여덟 부분집합 중 하나입니다.

부분 집합 만들기 (Power Set)

{A, B, C} → 공집합 포함 총 $2^3 = 8$개의 부분집합. 비트마스크로 생성한다.

power_set.py

```
def power_set(arr):
    n = len(arr)
    subsets = []
    for i in range(1 << n):     # 0 ~ 2^n - 1
        cur = []
        for j in range(n):
            if i & (1 << j):    # j번째 비트가 1이면
                cur.append(arr[j])
        subsets.append(cur)
    return subsets
```

비트마스크 대응표

i	2진수	부분집합
0	000	{ }
1	001	{A}
2	010	{B}
3	011	{A,B}
5	101	{A,C}
7	111	{A,B,C}

1 << n = 2^n | i & (1 << j) = i의 j번째 비트 검사 | 비트 연산이 동적 계획법 상태 표현의 핵심

비트 연산을 처음 보는 학생을 위한 세 단계

줄별 해설 1/1 · 무엇을 하는가 → 왜 필요한가

```
1 << 0 = 1 / 1 << 1 = 2 / 1 << 2 = 4
```

이진수의 자리 값은 오른쪽부터 1,2,4,8입니다. 왼쪽으로 한 칸 이동할 때마다 값이 두 배가 되므로 j번째 자리만 켤 때 1<<j를 씁니다.

```
5 = 101₂   /   5 & 2 = 101₂ & 010₂ = 000₂
```

&는 같은 자리끼리 비교하여 둘 다 1인 곳만 남깁니다. 5에는 2의 자리가 꺼져 있으므로 결과는 0이고 두 번째 원소는 선택하지 않습니다.

```
5 & 4 = 100₂ = 4 → True
```

Python의 if는 정수 0을 거짓, 0이 아닌 정수를 참으로 봅니다. i & (1<<j) == 1로 바꾸면 j=1,2의 선택을 놓칩니다.

비트 연산을 처음 보는 학생을 위한 세 단계

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 Python에서 2**n이 거듭제곱입니다. 2^n은 XOR이며 2^3=1입니다. 코드의 1<<n은 n≥0에서 2**n과 같습니다.

02 &는 비트 AND, and는 논리 연산입니다. 마스크의 특정 자리를 검사할 때 and로 바꾸면 안 됩니다.

03 비트는 오른쪽부터 j=0,1,2이므로 CBA로 적힌 101은 A와 C 선택입니다.

부분집합 8개: 비트와 실제 결과를 연결

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
i=0: 000 → []     i=1: 001 → [A]
```

000은 아무 원소도 고르지 않는 공집합입니다. 001은 맨 오른쪽 j=0만 켜져 있으므로 arr[0]인 A를 고릅니다.

```
i=2: 010 → [B]    i=3: 011 → [A,B]
```

010은 B만, 011은 A와 B를 선택합니다. 결과 안의 원소 순서는 j를 0부터 검사하는 입력 순서입니다.

```
i=4: 100 → [C]    i=5: 101 → [A,C]
```

100은 세 번째 위치만 선택하고 101은 첫 번째와 세 번째 위치를 선택합니다. 숫자 i를 리스트에 넣는 것이 아닙니다.

부분집합 8개: 비트와 실제 결과를 연결

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
i=6: 110 → [B,C]  i=7: 111 → [A,B,C]
```

110은 B,C이고 마지막 111은 모든 원소를 선택합니다. 각 위치에 두 선택지가 있어 2×2×2=8가지입니다.

부분집합 8개: 비트와 실제 결과를 연결

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 최종 반환: [[],[A],[B],[A,B],[C],[A,C],[B,C],[A,B,C]]. 실제 Python에서는 A,B,C가 문자열이면 따옴표도 표시됩니다.

02 n=0이면 range(1)이 한 번 실행되어 [[]]을 반환합니다. 원소가 없어도 아무것도 고르지 않는 방법 한 가지는 있습니다.

03 입력 값이 중복되면 서로 다른 위치의 선택이 같은 값 목록이 될 수 있습니다. 수학의 집합으로 해석하려면 입력 원소가 서로 다르다고 가정합니다.

부분집합 들여쓰기: 세 개의 서로 다른 범위

줄별 해설 1/1 · 무엇을 하는가 → 왜 필요한가

```
for i ...:  →  cur=[]
```

마스크가 바뀔 때마다 새 cur가 필요합니다. 밖에서 한 번만 만들면 원소가 누적되고 같은 리스트 객체를 여러 번 저장하는 오류가 생깁니다.

```
for j ...:  →  if ...:  →  cur.append(arr[j])
```

선택된 원소마다 현재 부분집합에 추가합니다. 이 append는 if 안, j 반복 안, i 반복 안의 작업입니다.

```
subsets.append(cur)  →  return subsets
```

부분집합 저장은 j 반복 밖·i 반복 안입니다. 반환은 i 반복 밖입니다. 들여쓰기 한 단계의 차이가 처리 단위를 원소·부분집합·전체로 바꿉니다.

부분집합 들여쓰기: 세 개의 서로 다른 범위

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 subsets.append(cur)를 j 안으로 들이면 한 마스크를 여러 번 저장하고 같은 cur 참조가 겹칩니다.

02 시간: 마스크 2^n개 × 위치 n개 확인 = $O(n \cdot 2^n)$. 전체 결과 원소 수는 $n \geq 1$에서 $n \cdot 2^{n-1}$개입니다.

03 연습: arr=[10,20]일 때 네 결과와 i=2에서의 j별 & 결과를 먼저 적고 코드를 실행해 비교합니다.

부분집합: 코드 열 줄의 의미

줄별 해설 1/4 · 무엇을 하는가 → 왜 필요한가

```
def power_set(arr):
```

입력 원소를 넣거나 빼서 만들 수 있는 모든 선택을 반환합니다. arr=['A','B','C']에서 서로 다른 위치 세 개를 각각 선택·미선택합니다.

```
n = len(arr)
```

선택할 위치의 수입니다. 이 예제에서는 n=3이며 비트 세 개가 필요합니다. 값의 합이나 가장 큰 값을 뜻하지 않습니다.

```
subsets = []
```

완성된 부분집합들을 모으는 바깥 리스트입니다. 아직 어떤 선택도 처리하지 않았으므로 빈 목록에서 시작합니다.

부분집합: 코드 열 줄의 의미

줄별 해설 2/4 · 무엇을 하는가 → 왜 필요한가

```
for i in range(1 << n):
```

1을 왼쪽으로 n칸 옮기면 2의 n제곱입니다. n=3이면 8이므로 i는 0..7입니다. i 하나가 넣을 원소를 표시하는 비트마스크 한 개입니다.

```
cur = []
```

이번 마스크의 부분집합을 새로 만듭니다. 바깥 반복 안에 있어야 이전 선택이 섞이지 않고 각 결과가 서로 다른 리스트가 됩니다.

```
for j in range(n):
```

입력 위치 0..n-1을 한 번씩 검사합니다. j는 선택할 원소의 위치이며 arr[j]와 j번째 비트를 연결합니다.

부분집합: 코드 열 줄의 의미

줄별 해설 3/4 · 무엇을 하는가 → 왜 필요한가

```
if i & (1 << j):
```

1<<j는 j번째 자리만 1인 비트입니다. &는 두 수에서 함께 1인 자리만 남깁니다. 결과가 0이 아니면 i에서 그 원소를 넣으라고 표시한 것입니다.

```
cur.append(arr[j])
```

선택 비트가 켜져 있을 때 현재 원소의 값을 넣습니다. j 자체가 아니라 arr[j]를 넣어야 위치 번호 대신 실제 원소가 저장됩니다.

```
subsets.append(cur)
```

j 반복이 끝난 뒤 완성된 이번 선택을 한 번만 저장합니다. i=0의 cur=[]도 저장하므로 공집합을 빼먹지 않습니다.

부분집합: 코드 열 줄의 의미

줄별 해설 4/4 · 무엇을 하는가 → 왜 필요한가

```
return subsets
```

i의 모든 마스크를 처리한 뒤 바깥 결과 목록을 반환합니다. 반복 안에 return하면 첫 마스크의 공집합만 돌려주고 종료됩니다.

부분집합: 코드 열 줄의 의미

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 n=3이면 마스크 수는 8개입니다. 비트 표시 순서는 C B A이며 가장 오른쪽 비트가 arr[0]='A' 입니다.

02 i=5 (101), j=0: 101 & 001 = 001 → A 추가
j=1: 101 & 010 = 000 → 건너뜀
j=2: 101 & 100 = 100 → C 추가

03 이번 cur=['A','C']를 subsets에 넣습니다. & 결과 4도 참입니다. 반드시 1일 때만 참인 것이 아닙니다.

비트마스크로 모든 부분집합 · C

P24 / out[][8], sizes[] = 부분집합과 길이, return = 개수 (최대 256)

```c
int power_set(const int A[], int n, int out[][8], int sizes[]) {
  int count = 1 << n;
  for (int mask = 0; mask < count; mask++) {
    sizes[mask] = 0;
    for (int j = 0; j < n; j++)
      if (mask & (1u << j))
        out[mask][sizes[mask]++] = A[j];
  }
  return count;
}
```

코드 읽기

mask & (1u << j)가 참이면 A[j]를 선택합니다. 부분집합 수는 2^n, 모든 원소 검사까지 포함하면 O(n*2^n)입니다.

C 부분집합: 길이가 다른 결과를 저장하는 방법

줄별 해설 1/3 · 무엇을 하는가 → 왜 필요한가

```
int power_set(..., int out[][8], int sizes[])
```

부분집합마다 길이가 다르므로 out의 값과 sizes의 길이를 함께 저장합니다. 이 배열 예제는 $0 \le n \le 8$, 최소 2^n개 결과 행을 전제로 합니다.

```
int count = 1 << n;
```

마스크 개수 2^n을 계산합니다. $n \le 8$에서는 int에 안전하게 들어갑니다. 임의로 큰 n이나 음수만큼 시프트하는 코드는 허용되지 않습니다.

```
for (int mask=0; mask<count; mask++)
```

빈 선택 0부터 모든 비트가 켜진 count-1까지 순서대로 처리합니다. 각 mask 번호를 결과 행 번호로도 사용합니다.

C 부분집합: 길이가 다른 결과를 저장하는 방법

줄별 해설 2/3 · 무엇을 하는가 → 왜 필요한가

```
sizes[mask]=0;
```

현재 행에 저장한 원소 수를 0으로 초기화합니다. 이 값이 다음에 쓸 열 번호이기도 합니다.

```
for (int j=0; j<n; j++) if (mask & (1u<<j))
```

각 위치의 선택 여부를 비트 AND로 확인합니다. 1u는 unsigned 정수 1입니다. 결과가 0이 아니면 조건이 참입니다.

```
out[mask][sizes[mask]++] = A[j];
```

현재 행의 다음 빈 칸에 A[j]를 쓰고 해당 행 길이를 늘립니다. 저장과 증가를 두 문장으로 분리해서 읽어도 같습니다.

C 부분집합: 길이가 다른 결과를 저장하는 방법

줄별 해설 3/3 · 무엇을 하는가 → 왜 필요한가

```
return count;
```

총 부분집합 수를 반환합니다. 한 행의 유효 열은 sizes[행]개뿐입니다. 공집합 행의 길이 0을 보고 아무 값도 읽지 않아야 합니다.

C 부분집합: 길이가 다른 결과를 저장하는 방법

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 A=[10,20,30], mask=5(101): out[5][0]=10 → sizes[5]=1; out[5][1]=30 → sizes[5]=2.

02 mask=0의 sizes[0]=0입니다. out[0]에 초기화되지 않은 숫자가 있어도 읽지 않으면 문제 없습니다.

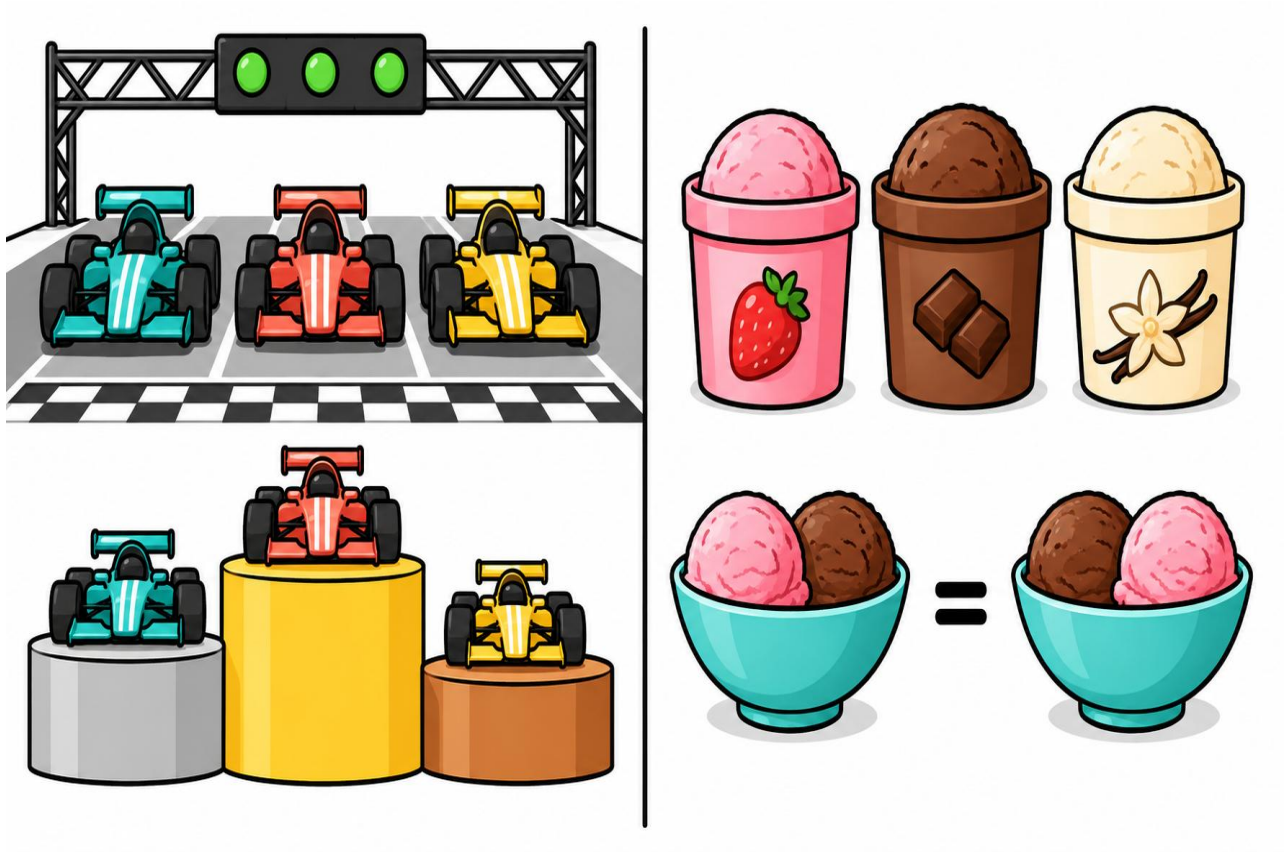
03 Python은 리스트 자체가 길이를 기억합니다. C에서는 sizes 배열이 그 역할을 명시적으로 맡습니다.

2.6

시간 복잡도와 조합적 폭발

완전 탐색의 치명적 약점: 입력이 조금만 커져도 경우의 수가 폭발적으로 증가합니다.

같은 입력 크기, 전혀 다른 후보 수



$n^2 = 400$

두 위치를 독립적으로 선택

$2^n = 1{,}048{,}576$

각 원소를 포함하거나 제외

$n! \approx 2.43 \times 10^{18}$

모든 원소의 순서를 나열

n = 20일 때의 비교입니다. 전체 비용은 후보 수 × 후보 하나의 처리 비용으로 판단합니다.

n에 따른 경우의 수 비교

n이 조금만 커져도 현실적으로 불가능한 계산량이 됩니다.

n	n! (팩토리얼)	2^n	n^n
5	120	32	3,125
10	3,628,800	1,024	10,000,000,000
20	2.43×10^{18}	1,048,576	1.05×10^{26}
30	2.65×10^{32}	1.07×10^{9}	2.06×10^{44}

n=20, 팩토리얼: 1초에 1억 번 계산하는 컴퓨터로도 약 770년

탐색 공간 계산 · C

P25 / out = [n!, 2^n, n^n]

```c
void search_space(int n, long long out[3]) {
  out[0] = out[1] = out[2] = 1;
  for (int i = 1; i <= n; i++) {
    out[0] *= i;
    out[1] *= 2;
    out[2] *= n;
  }
}
```

코드 읽기

곱셈 누적값은 1부터 시작합니다. C의 int 범위를 넘는 결과가 있으므로 long long을 사용합니다.

탐색 공간: 곱셈을 누적하는 세 가지 방식

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
void search_space(int n, long long out[3])
```

out[0]에 n!, out[1]에 2^n, out[2]에 n^n을 계산하는 비교용 함수입니다. 시간 복잡도 함수의 값을 작은 n에서 확인합니다.

```
out[0]=out[1]=out[2]=1;
```

곱셈의 시작값은 1입니다. 0으로 시작하면 무엇을 곱해도 0이라 후보 수를 계산할 수 없습니다.

```
for (int i=1; i<=n; i++)
```

곱셈을 n번 반복합니다. i는 1,2,…,n이므로 첫 번째 결과는 1부터 n까지 곱하게 됩니다.

탐색 공간: 곱셈을 누적하는 세 가지 방식

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
out[0] *= i;
```

out[0]=out[0]*i와 같습니다. n=4에서 1→1→2→6→24로 누적하여 4!을 구합니다.

```
out[1] *= 2; out[2] *= n;
```

매번 2를 곱하면 2^n, 매번 n을 곱하면 n^n입니다. 후보 수를 계산하는 이 함수 자체는 반복 n번으로 O(n)입니다.

탐색 공간: 곱셈을 누적하는 세 가지 방식

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 n=4의 최종 결과는 [24,16,256]입니다. 후보를 실제 생성하지 않고 개수만 계산한 것입니다.

02 long long도 무한하지 않습니다. 일반적인 64비트 signed 환경에서 n^n은 n=16까지 맞지만 17 ^{i} 누적의 최종값 17^{17}은 범위를 넘습니다.

03 따라서 이 함수를 큰 n의 정확한 수 계산기로 쓰면 안 됩니다. Python 큰 정수나 넘침 검사가 필요합니다.

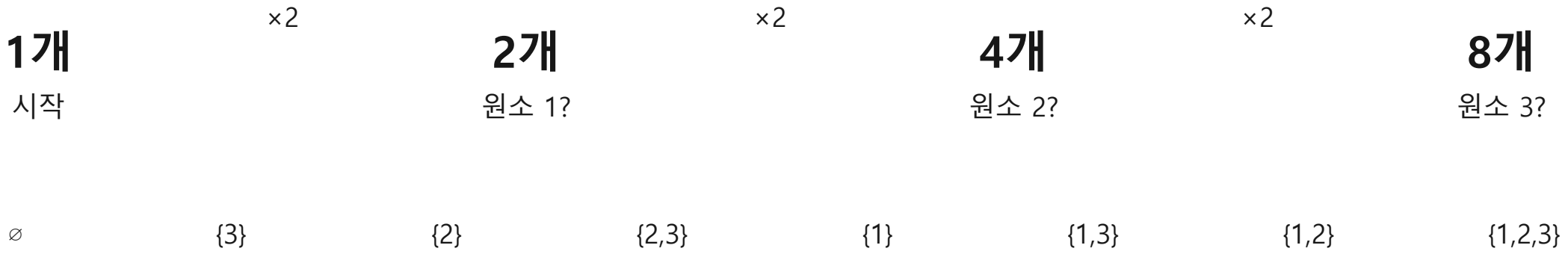
비밀번호 자릿수와 완전 탐색의 한계

숫자 한 자리를 늘리면 후보 수는 10배. 아래 시간은 초당 1만 번 검사하는 최악의 경우입니다.

4자리	**10,000**	초당 1만 번 → 1초
6자리	**1,000,000**	초당 1만 번 → 100초
8자리	**100,000,000**	초당 1만 번 → 약 2.8시간
12자리	**1,000,000,000,000**	초당 1만 번 → 약 3.2년
20자리	**10^{20}**	초당 1만 번 → 약 3.17억 년

부분집합 생성과 O(2ⁿ)

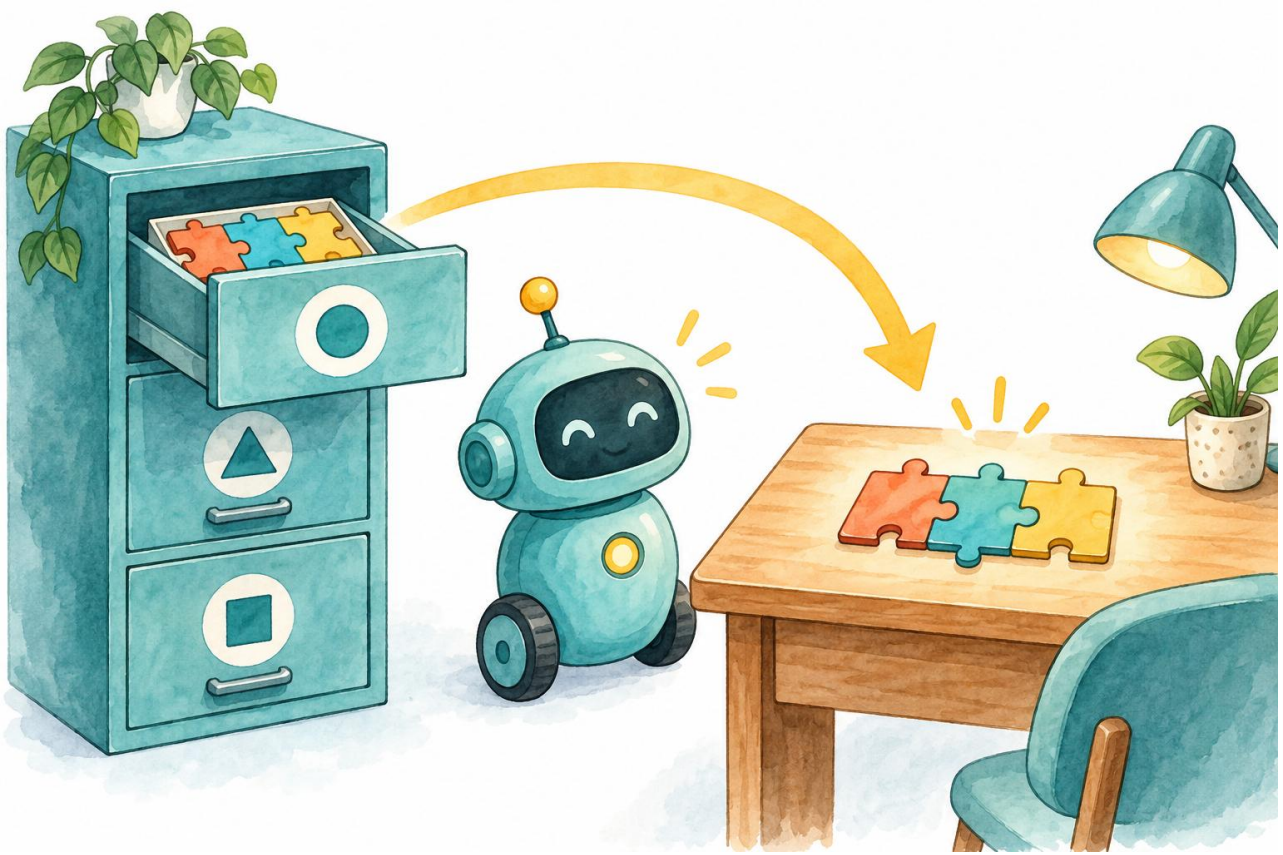
{1, 2, 3} → 각 원소를 포함하거나 포함하지 않거나 → $2^3 = 8$가지 (선택의 연속 = 지수적 폭발)



$2^3 = 8$개의 부분집합

원소 n개 → 2^n개의 부분집합 n=10: 1,024 n=20: 1,048,576 n=30: 약 10억

모든 도시를 방문하고 출발점으로 돌아오기



도시는 한 번씩 방문

출발점을 정하고
나머지 도시의 방문 순서를 생성

마지막에 출발점으로 복귀

마지막 도시에서 출발점까지의
이동 비용도 합산

가장 짧은 순회를 선택

4개 도시에서 출발점을 고정하면
나머지 순서는 3! = 6가지

그림의 사각 순회는 후보 하나입니다. 최단 경로라는 뜻은 아니며 비용으로 비교해야 합니다.

외판원 순회 (TSP): O(n!)

모든 도시를 한 번씩 방문하고 출발점으로 돌아오는 최단 경로 찾기.

도시 간 거리 행렬

	A	B	C	D
A	-	4	7	3
B	4	-	6	2
C	7	6	-	5
D	3	2	5	-

A→B→C→D→A	4+6+5+3 = 18
A→B→D→C→A	4+2+5+7 = 18
A→D→B→C→A	3+2+6+7 = 18
A→D→C→B→A	3+5+6+4 = 18

출발점 A 고정 → 3! = 6가지 순서

출발점 고정: (n-1)!개 순서 | 10개 도시: 9! = 362,880개

외판원 순회 최소 비용 · C (1/2)

P26 / return = 최소 왕복 비용

```c
static void tsp_visit(const int d[][8], int n, int depth, int last, int used[],
                      int cost, int *best) {
  if (depth == n) {
    int total = cost + d[last][0];
    if (total < *best)
      *best = total;
    return;
  }
  for (int next = 1; next < n; next++)
    if (!used[next]) {
      used[next] = 1;
      tsp_visit(d, n, depth + 1, next, used, cost + d[last][next], best);
      used[next] = 0;
    }
}
```

다음 장에서 이 보조 함수를 호출하는 공개 함수가 이어집니다.

외판원 재귀: 현재 경로의 상태를 전달

줄별 해설 1/3 · 무엇을 하는가 → 왜 필요한가

```
tsp_visit(d,n,depth,last,used,cost,best)
```

d는 거리표, depth는 방문한 도시 수, last는 현재 도시, cost는 지금까지 이동 비용입니다. best는 전체 탐색이 공유하는 최솟값 주소입니다.

```
if (depth == n) {
```

도시 n개를 모두 방문했을 때입니다. 아직 출발 도시로 돌아온 것은 아니므로 돌아오는 비용을 더해야 여행이 완성됩니다.

```
int total=cost+d[last][0];
```

마지막 도시에서 0번 도시로 돌아오는 간선의 비용을 더합니다. 이 항을 빼면 순환 여행이 아니라 단순 경로 비용이 됩니다.

외판원 재귀: 현재 경로의 상태를 전달

줄별 해설 2/3 · 무엇을 하는가 → 왜 필요한가

```
if (total<*best) *best=total; return;
```

완성된 여행이 더 싸면 전체 최솟값을 갱신하고 한 단계 위로 돌아갑니다. best 주소가 아니라 주소가 가리키는 값을 바꿉니다.

```
for (int next=1; next<n; next++)
```

출발지 0은 이미 방문했으므로 후보에서 제외합니다. 다음 도시를 1부터 n-1까지 검사합니다.

```
if (!used[next]) { used[next]=1;
```

아직 방문하지 않은 도시만 선택하고 방문 표시합니다. !는 논리 부정으로 0인 방문 상태를 참으로 바꿉니다.

외판원 재귀: 현재 경로의 상태를 전달

줄별 해설 3/3 · 무엇을 하는가 → 왜 필요한가

```
tsp_visit(...,depth+1,next,used,cost+d[last][next],best);
```

방문 수를 하나 늘리고 현재 도시를 next로 바꿉니다. 지금 이동한 간선 비용만 더한 새 cost를 자식 호출에 전달합니다.

```
used[next]=0;
```

자식 탐색이 끝나면 방문을 취소해 다른 경로에서 이 도시를 다시 고를 수 있게 합니다. cost는 값으로 전달하므로 별도 빼기 복구가 필요 없습니다.

외판원 재귀: 현재 경로의 상태를 전달

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 도시 0,1,2와 대칭 거리 d01=10,d02=15,d12=20이면 0→1→2→0 비용은 10+20+15=45입니다.

02 다른 경로 0→2→1→0도 45입니다. 출발지를 고정하면 후보 순서는 (n-1)!개입니다.

03 이 코드는 모든 도시 사이의 유효한 거리와 int에 들어가는 경로 비용을 가정합니다. 없는 길을 0으로 표시한 입력에는 추가 처리가 필요합니다.

외판원 순회 최소 비용 · C (2/2)

P26 / return = 최소 왕복 비용

```c
int tsp_min(const int dist[][8], int n) {
  int used[8] = {1}, best = INT_MAX;
  tsp_visit(dist, n, 1, 0, used, 0, &best);
  return best;
}
```

코드 읽기

used[0]=1로 시작합니다. 모든 도시를 방문한 순간 마지막 도시에서 0으로 돌아오는 비용을 더하세요.

외판원 시작: 0번 도시를 이미 방문한 상태

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
int tsp_min(const int dist[][8], int n)
```

거리표의 한 행 폭은 8칸입니다. 이 예제는 1≤n≤8의 완전한 거리표를 전제로 하며 최소 왕복 비용을 반환합니다
.

```
int used[8]={1}, best=INT_MAX;
```

used[0]만 1이고 나머지는 0으로 초기화됩니다. 출발지 0을 이미 방문했고 최선 비용은 아직 없으므로 최댓값에서 시작합니다.

```
tsp_visit(dist,n,1,0,used,0,&best);
```

depth=1은 도시 0을 센 값입니다. last=0, 이동 비용=0으로 시작하고 &best를 보내 보조 함수가 최솟값을 수정하게 합니다.

외판원 시작: 0번 도시를 이미 방문한 상태

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
return best;
```

모든 경로를 확인하고 최소 비용을 돌려줍니다. n=1이고 대각 거리 0이면 자기 도시만 있는 여행의 비용은 0입니다.

외판원 시작: 0번 도시를 이미 방문한 상태

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 depth를 0으로 시작하면 방문 수와 used 상태가 어긋납니다. 0번 도시를 센 depth=1이어야 합니다.

02 0번 출발을 고정하는 이유는 순환 경로의 시작점만 바꾼 중복을 줄이기 위해서입니다.

03 후보 수가 팩토리얼로 늘기 때문에 배열 8칸이라는 제한은 실행 시간과도 연결됩니다.

2.7

탐색의 최적화

고급 알고리즘들은 완전 탐색의 비효율을 줄이려는 노력 끝에 탄생했습니다.

완전 탐색의 계산량 줄이기

완전 탐색

모든 후보를 검사하는 구현을 기준으로
어떤 계산을 줄일 수 있는지 찾습니다.

백트래킹

불필요한 가지 제거

정답이 될 수 없는 후보는 중단

분할 정복

문제를 나누어 해결

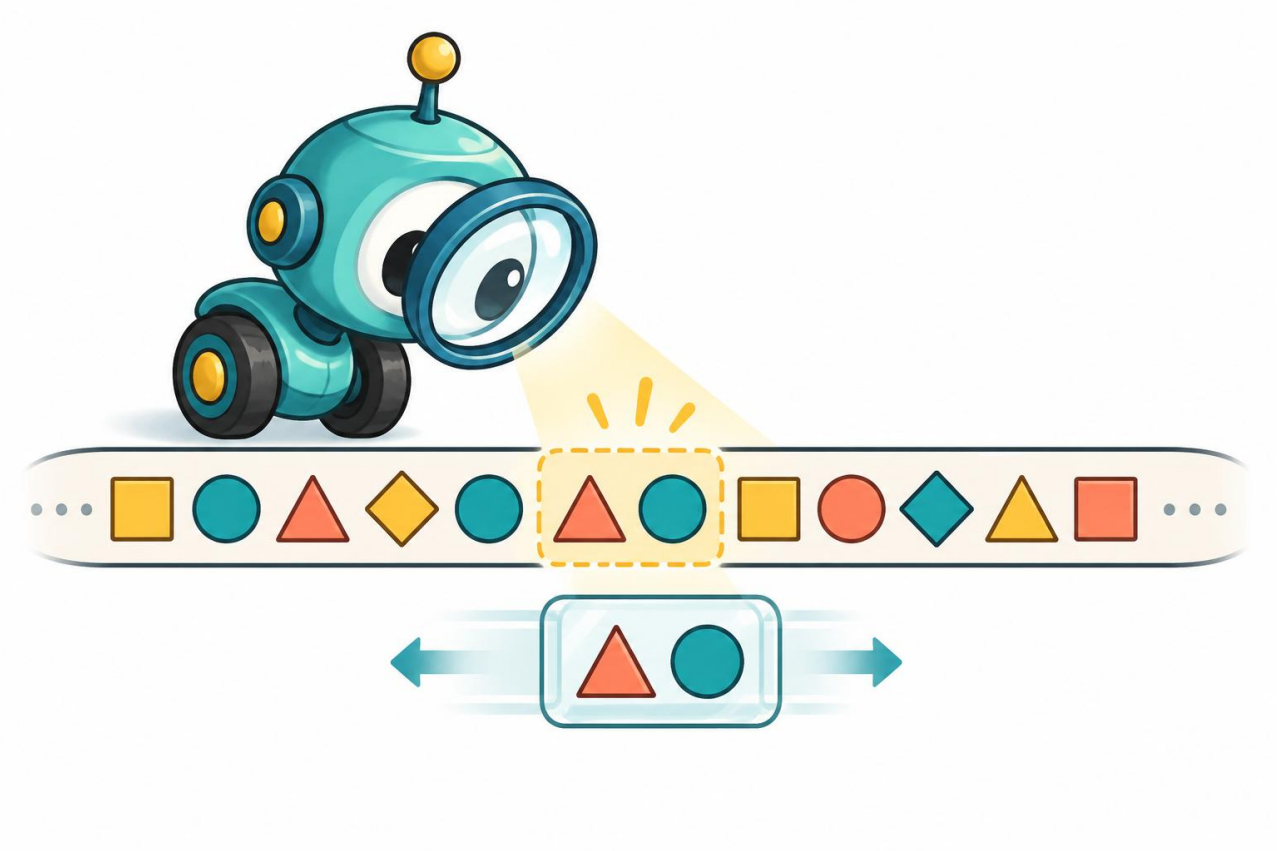
작은 문제의 해를 결합

동적 계획법

계산 결과 재사용

같은 부분 문제의 중복 계산 제거

답이 될 수 없는 가지에서 되돌아오기



목표: 부분집합의 합 10

완전 탐색은 선택을 완성한 뒤
합이 10인지 검사

중간 합이 10을 넘었다면?

남은 원소가 모두 음이 아닐 때
이 가지를 더 탐색하지 않는다

다른 선택은 계속 탐색

선택 상태를 복구하고
다음 가지를 확인한다

음수가 남아 있다면 합이 다시 줄 수 있으므로, 같은 가지치기를 그대로 적용하면 안 됩니다.

합이 target인 부분집합 · C

P27 / return = 조건을 만족하는 부분집합 수

```c
int subset_sum_count(const int A[], int n, int target) {
  int count = 0;
  for (unsigned mask = 0; mask < (1u << n); mask++) {
    int sum = 0;
    for (int j = 0; j < n; j++)
      if (mask & (1u << j))
        sum += A[j];
    if (sum == target)
      count++;
  }
  return count;
}
```

코드 읽기

모든 비트마스크에 대해 합을 0부터 계산합니다. target=0일 때 공집합을 빠뜨리지 마세요.

부분집합 합: 선택을 다 더한 후 검사

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
int subset_sum_count(..., int target); count=0
```

부분집합의 합이 target인 경우의 수를 셉니다. 값이 같은 원소도 위치가 다르면 다른 선택으로 취급합니다.

```
for (unsigned mask=0; mask<(1u<<n); mask++)
```

모든 선택 비트마스크를 나열합니다. n은 unsigned의 비트 폭보다 작아야 하며, 실제 실행 가능한 작은 크기여야 합니다.

```
int sum=0;
```

이번 부분집합의 합을 새로 시작합니다. 이 줄이 바깥 반복 밖에 있으면 이전 부분집합 합까지 섞이게 됩니다.

부분집합 합: 선택을 다 더한 후 검사

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
for j ... / if (mask & (1u<<j)) sum+=A[j];
```

선택된 비트에 해당하는 원소만 더합니다. 선택되지 않은 원소는 합에 영향을 주지 않습니다.

```
if (sum==target) count++;
```

모든 j를 처리한 뒤 완성된 합을 한 번 검사합니다. 안쪽 반복 중간에 검사하면 같은 부분집합을 여러 번 셀 수 있습니다.

```
return count;
```

조건을 만족한 마스크 개수를 반환합니다. mask=0의 합은 0이므로 target=0이면 공집합도 한 가지로 셉니다.

부분집합 합: 선택을 다 더한 후 검사

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 A=[1,2,3], target=3: mask=3(011)의 {1,2}, mask=4(100)의 {3} 두 가지이므로 답은 2입니다.

02 이 코드는 모든 마스크를 검사하는 기본형입니다. 앞의 가지치기 개념을 아직 구현한 코드는 아닙니다.

03 음수가 있으면 중간 합이 target보다 크다고 멈추면 안 됩니다. 예: 5를 더한 뒤 -2를 더해 3이 될 수 있습니다.

선형 검색과 이진 탐색

정렬된 전화번호부에서 이름 찾기

선형 검색

처음부터 한 줄씩 확인합니다.
최악에는 모든 항목을 봅니다.

전체 항목 수에 비례

O(n)

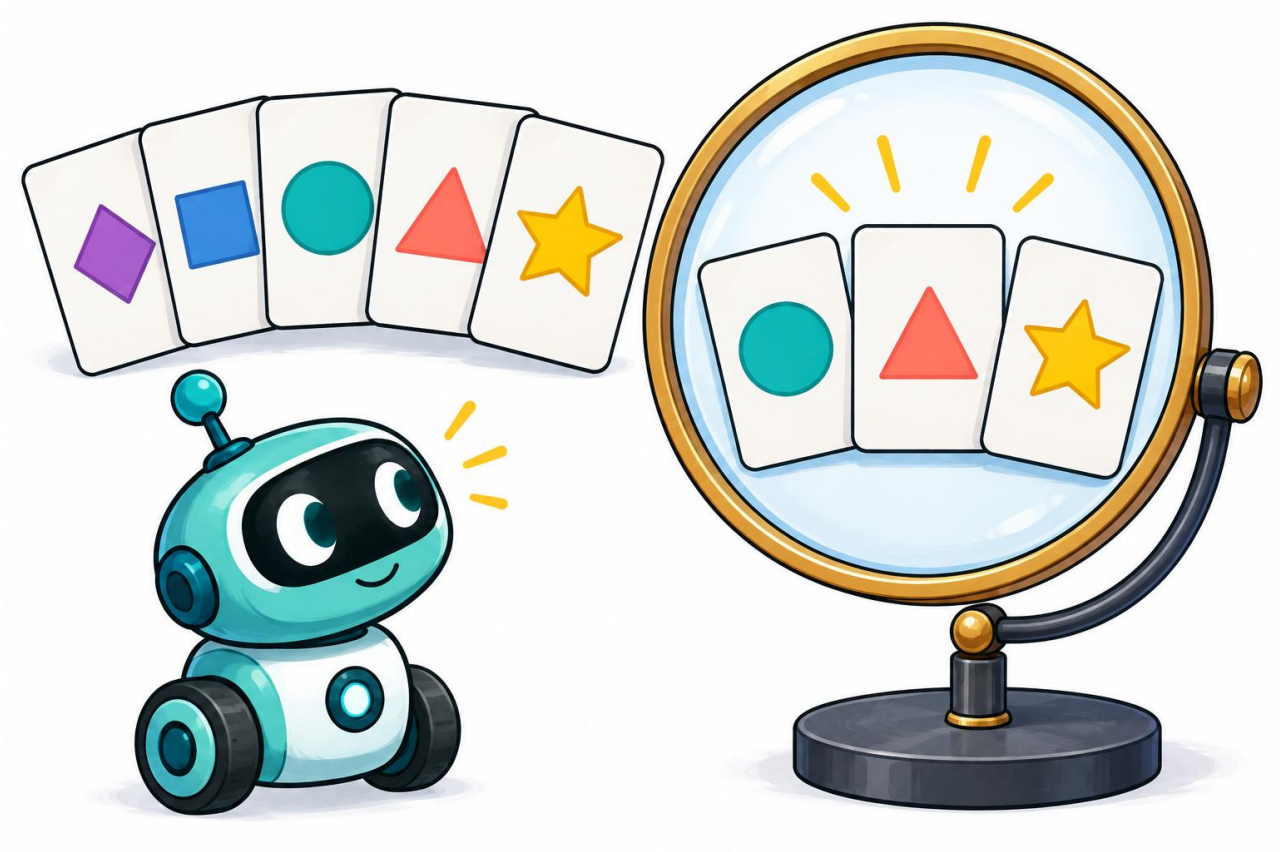
이진 탐색

중간 항목과 비교해 앞뒤를 판단합니다.
매번 남은 범위의 절반을 제외합니다.

정렬된 자료에서 사용

O(log n)

이미 구한 답은 저장했다가 다시 쓰기



같은 작은 문제가 반복된다

피보나치에서 F(3)처럼
같은 값을 여러 경로에서 요청

답을 구하면 저장한다

작은 문제의 입력을 기준으로
계산 결과를 기억

다시 요청되면 꺼내 쓴다

저장된 답이 있으면
동일한 계산을 반복하지 않는다

메모이제이션은 입력이 같은 부분 문제의 답을 재사용합니다. 저장 공간도 필요합니다.

기억력을 더하면 → 동적 계획법

피보나치 수열 $F(n) = F(n-1) + F(n-2)$

단순 재귀: $O(2^n)$ 상한 · 중복 계산

$F(5) = F(4) + F(3)$

$F(4) = F(3) + F(2)$ ← 다시 계산

$F(3) = F(2) + F(1)$ ← 다시 계산

$F(3)$, $F(2)$이 여러 번 호출 → 지수적 폭발

동적 계획법: $O(n)$: 기억

fib_dp.py

```
memo = {}
def fib(n):
    if n in memo:
        return memo[n]    # 기억!
    if n <= 1:
        return n
    memo[n] = fib(n-1) + fib(n-2)
    return memo[n]        # 저장!
```

메모이제이션: 같은 부분 문제의 계산 결과를 저장하고 재사용합니다.

피보나치 메모: 계산 결과를 재사용하기

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
memo = {}
```

n을 키, fib(n)을 값으로 저장하는 사전입니다. 함수 밖에 있어 재귀 호출들과 이후 호출이 같은 기록을 공유합니다.

```
def fib(n): / if n in memo: return memo[n]
```

이미 계산한 n이면 기록을 즉시 돌려줍니다. n in memo는 키의 존재 검사이지 값이 0이 아닌지 검사하는 것이 아닙니다.

```
if n<=1: return n
```

0과 1은 직접 답을 아는 종료 조건입니다. 여기서는 음수가 아닌 정수 n을 가정합니다. 종료 조건이 없으면 재귀가 계속 내려갑니다.

피보나치 메모: 계산 결과를 재사용하기

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
memo[n]=fib(n-1)+fib(n-2)
```

아직 계산하지 않은 경우만 두 작은 문제를 호출하여 더한 뒤 저장합니다. 저장 시점은 두 호출이 값을 돌려준 이후입니다.

```
return memo[n]
```

방금 저장한 값을 호출한 쪽에 돌려줍니다. 반환만 하고 저장하지 않으면 다음에 같은 n을 또 계산합니다.

피보나치 메모: 계산 결과를 재사용하기

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 fib(4) → fib(3) → fib(2) → fib(1)=1, fib(0)=0 → memo[2]=1 → memo[3]=2 → fib(2)는 기록 1 → memo[4]=3.

02 서로 다른 부분문제는 0..n입니다. 각 결과를 한 번 계산하므로 일반적인 산술 비용 모델에서 시간 O(n), 저장·재귀 공간 O(n)입니다.

03 매 호출 안에서 memo={}를 새로 만들면 기록이 공유되지 않습니다. 큰 n에서는 Python의 재귀 깊이 제한도 고려합니다.

기억하는 피보나치 · C

P28 / return = F(n), long long

```c
static long long fib_visit(int n, long long memo[], int ready[]) {
  if (n <= 1)
    return n;
  if (ready[n])
    return memo[n];
  memo[n] = fib_visit(n - 1, memo, ready) + fib_visit(n - 2, memo, ready);
  ready[n] = 1;
  return memo[n];
}
long long fib_memo(int n) {
  long long memo[71] = {0};
  int ready[71] = {0};
  return fib_visit(n, memo, ready);
}
```

C에서는 memo[71]과 ready[71]을 따로 두면 결과 0과 미계산 상태를 구분할 수 있습니다.

C 메모: 값 0과 미계산을 구분

줄별 해설 1/3 · 무엇을 하는가 → 왜 필요한가

```
fib_visit(int n, long long memo[], int ready[])
```

memo는 값, ready는 계산 완료 여부를 저장합니다. 실제 결과가 0일 수도 있으므로 memo[n]==0만으로 미계산을 판단하지 않습니다.

```
if (n<=1) return n;
```

기본값 fib(0)=0, fib(1)=1을 바로 반환합니다. 이 구현은 기본값을 ready에 기록하지 않아도 재귀 계산을 하지 않으므로 충분합니다.

```
if (ready[n]) return memo[n];
```

이미 계산했으면 저장된 값을 돌려줍니다. 이 줄 이후로 내려왔다는 것은 아직 해당 n을 계산하지 않았다는 뜻입니다.

C 메모: 값 0과 미계산을 구분

줄별 해설 2/3 · 무엇을 하는가 → 왜 필요한가

```
memo[n]=fib_visit(n-1,...)+fib_visit(n-2,...);
```

작은 두 문제의 값을 더해 저장합니다. C에서는 두 피연산자 호출 순서를 가정하면 안 되지만, 어느 쪽을 먼저 계산해도 같은 답이 나옵니다.

```
ready[n]=1; return memo[n];
```

값 저장이 끝난 뒤 완료 표시를 하고 반환합니다. 값이 준비되기 전에 ready를 켜면 다른 호출이 잘못된 값을 읽을 수 있습니다.

```
long long fib_memo(int n) {
```

외부에서 부르는 시작 함수입니다. 이 예제의 배열은 인덱스 0..70이므로 $0 \le n \le 70$ 범위를 전제로 합니다.

C 메모: 값 0과 미계산을 구분

줄별 해설 3/3 · 무엇을 하는가 → 왜 필요한가

```
long long memo[71]={0}; int ready[71]={0};
```

값과 완료 표시를 모두 초기화합니다. 한 번의 fib_memo 호출 안에서만 공유되고 다음 최상위 호출에서는 새 배열이 생깁니다.

```
return fib_visit(n,memo,ready);
```

준비한 배열을 보조 함수에 넘기고 그 계산 결과를 그대로 반환합니다. long long은 fib(70)을 담을 수 있습니다.

C 메모: 값 0과 미계산을 구분

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 n=4의 결과는 3입니다. fib(2),fib(3),fib(4)를 각각 계산하면 해당 ready 값이 1이 됩니다.

02 ready[0]이 0이어도 fib(0)은 첫 종료 조건에서 0을 반환하므로 오류가 아닙니다.

03 이 방법은 같은 작은 문제를 반복해서 푸는 비용을 줄입니다. 모든 재귀가 자동으로 빠른 것은 아닙니다.

탐색 방법과 대표 문제

01	**완전 탐색**	모든 후보 검사 / 기본 해법
02	**백트래킹**	가지치기 / N-Queens, 스도쿠
03	**분할 정복**	문제 분할 / 병합 정렬, 이진 탐색
04	**동적 계획법**	메모이제이션 / 피보나치, 배낭 문제

2.8

코딩 테스트

완전 탐색을 실전에 적용합니다. 입력 크기로 가능 여부를 판단하고 문제를 해결합니다.

코딩 테스트에서 확인하는 능력

주어진 문제를 제한 시간과 메모리 안에서 해결합니다.

01	**알고리즘적 사고**	문제의 조건에 맞는 효율적인 풀이 설계
02	**자료구조 활용**	자료의 저장 방식과 연산 비용을 고려
03	**문제 해결 능력**	요구사항을 정확히 읽고 코드로 구현
04	**실전 감각**	제한 시간 안에서 검증하고 디버깅

코딩 테스트에서 입력 받기

대량 입력에서 input()은 느리다 → sys.stdin.readline 을 사용하자

fast_input.py

```
import sys
input = sys.stdin.readline        # input()을 덮어씀!

n, m = map(int, input().split())            # "3 5"
data = list(map(int, input().split()))      # "10 20 30 40"

n = int(input())                            # N개의 줄 입력
data = [int(input()) for _ in range(n)]

grid = [list(map(int, input().split()))     # 2차원 배열
        for _ in range(n)]
```

Python 입력: 네 가지 패턴을 구별하기

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
import sys; input=sys.stdin.readline
```

표준 입력에서 한 줄을 읽는 함수를 input이라는 이름으로 사용합니다. input()은 줄 끝 개행을 포함할 수 있으며 아래 코드는 서로 다른 입력 패턴 예시입니다.

```
n,m=map(int,input().split())
```

한 줄 "3 5"를 공백으로 나눠 문자열 두 개를 만들고 각각 정수로 바꿔 n,m에 대입합니다. 값이 정확히 두 개여야 합니다.

```
data=list(map(int,input().split()))
```

한 줄 "10 20 30 40"을 정수 목록 [10,20,30,40]으로 만듭니다. map은 변환을 제공하고 list가 결과를 목록에 담습니다.

Python 입력: 네 가지 패턴을 구별하기

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
n=int(input()); data=[int(input()) for _ in range(n)]
```

먼저 줄 수 n을 읽고 이후 n개 줄에서 정수 하나씩 읽습니다. _는 반복 횟수만 필요해서 변수 값을 사용하지 않겠다는 관례입니다.

```
grid=[list(map(int,input().split())) for _ in range(n)]
```

행을 n번 읽어 행 목록을 다시 목록에 담습니다. 한 줄에 정수 여러 개가 있는 2차원 입력이며 각 행의 길이는 문제 조건으로 확인합니다.

Python 입력: 네 가지 패턴을 구별하기

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 입력이 "3 5" 한 줄이면 첫 패턴만 씁니다. 네 패턴을 모두 연달아 실행하라는 뜻이 아닙니다.

02 2×3 격자: "1 2 3" / "4 5 6" 두 줄 → [[1,2,3],[4,5,6]]. grid[1][2]=6입니다.

03 input()을 호출할 때마다 다음 줄로 이동합니다. 같은 줄을 여러 번 다시 읽는 함수가 아닙니다.

2차원 배열 입력과 순회 · C

P36 / return = 모든 원소의 합

```c
int grid_sum(const int grid[][8], int rows, int cols) {
  int sum = 0;
  for (int i = 0; i < rows; i++)
    for (int j = 0; j < cols; j++)
      sum += grid[i][j];
  return sum;
}
```

코드 읽기

행 i는 rows 미만, 열 j는 cols 미만입니다. 정사각형이라고 가정하면 직사각형 입력에서 틀립니다.

C 격자 합: 행과 열을 모두 방문

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
int grid_sum(const int grid[][8], int rows, int cols)
```

한 행의 실제 저장 폭은 8이고 사용 범위는 rows×cols입니다. 0≤rows,cols≤8, 합이 int 범위라는 조건으로 사용합니다.

```
int sum=0;
```

더하기의 시작값은 0입니다. 전체 격자 합을 누적해야 하므로 두 반복문 밖에 둡니다.

```
for (int i=0; i<rows; i++)
```

행 번호를 위에서 아래로 고릅니다. rows는 개수이므로 마지막 행 인덱스는 rows-1입니다.

C 격자 합: 행과 열을 모두 방문

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
for (int j=0; j<cols; j++) sum+=grid[i][j];
```

현재 행의 왼쪽부터 오른쪽까지 값을 더합니다. 새 행마다 j는 다시 0으로 초기화됩니다.

```
return sum;
```

모든 칸을 처리한 뒤 합을 반환합니다. 반복 횟수는 rows×cols입니다. 첫 행 뒤에 반환하면 나머지 행을 놓칩니다.

C 격자 합: 행과 열을 모두 방문

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 격자 [[1,2,3],[4,5,6]]: sum은 0→1→3→6→10→15→21입니다.

02 grid[][8]의 8은 행 사이 저장 간격과 연결됩니다. grid[2][3]을 같은 타입처럼 넘기면 안 됩니다.

03 별도 입력·출력이 없는 함수이므로 채점기가 배열을 전달하고 반환값을 확인하는 형태에 맞습니다.

C의 표준 입력 · 정수와 배열

Python의 input().split() 대신 scanf의 반환값과 배열 크기를 확인합니다.

```c
#include <stdio.h>
int main(void) {
  int n, target, cards[100];
  if (scanf("%d %d", &n, &target) != 2)
    return 1;
  if (n < 0 || n > 100)
    return 1;
  for (int i = 0; i < n; i++)
    if (scanf("%d", &cards[i]) != 1)
      return 1;
  // Use cards, n, target in the algorithm here.
  for (int i = 0; i < n; i++)
    printf("%d ", cards[i]);
  putchar('\n');
  return 0;
}
```

정수 변수에는 &를 붙입니다. scanf의 공백 구분 입력은 줄바꿈도 건너뜁니다.

C 1차원 입력: scanf는 읽은 개수를 반환

줄별 해설 1/3 · 무엇을 하는가 → 왜 필요한가

```
#include <stdio.h> / int main(void)
```

표준 입출력 선언을 가져오고 프로그램 시작 함수를 정의합니다. 뒤의 알고리즘 함수와 main의 역할을 구별합니다.

```
int n,target,cards[100];
```

카드 수, 목표, 카드 100칸을 준비합니다. 선언 직후 변수에 유효한 입력이 들어 있는 것은 아닙니다.

```
if (scanf("%d %d",&n,&target)!=2) return 1;
```

정수 두 개를 읽어 변수의 주소에 써 달라는 뜻입니다. 성공적으로 읽은 개수가 2가 아니면 잘못된 입력이므로 오류 상태 1로 끝냅니다.

C 1차원 입력: scanf는 읽은 개수를 반환

줄별 해설 2/3 · 무엇을 하는가 → 왜 필요한가

```
if (n<0 || n>100) return 1;
```

카드 수가 확보한 배열 범위를 넘는지 검사합니다. ||는 둘 중 하나라도 참이면 참입니다. 배열은 입력 수에 맞춰 자동 확장되지 않습니다.

```
for i ... / if (scanf("%d",&cards[i])!=1) return 1;
```

카드 n개를 각 칸의 주소에 저장합니다. 공백과 줄바꿈 모두 정수 사이의 구분으로 읽을 수 있습니다.

```
// Use cards, n, target ...
```

여기에 실제 문제 풀이를 연결하라는 자리 표시입니다. 현재 자료의 main은 읽은 값을 다시 출력하는 입출력 확인 예제입니다.

C 1차원 입력: scanf는 읽은 개수를 반환

줄별 해설 3/3 · 무엇을 하는가 → 왜 필요한가

```
for (...) printf("%d ",cards[i]); putchar('\n');
```

읽은 n개 값을 차례대로 출력하고 줄을 바꿉니다. printf에는 값, scanf에는 쓸 주소를 전달한다는 차이를 기억합니다.

```
return 0;
```

정상 종료를 뜻합니다. 카드 합의 답이 0이라는 의미가 아닙니다. 실제 답은 printf 등으로 별도 출력해야 합니다.

C 1차원 입력: scanf는 읽은 개수를 반환

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 입력: 첫 줄 3 21, 다음 줄 5 6 7 → n=3,target=21,cards=[5,6,7]. 현재 출력은 5 6 7입니다.

02 목표 21을 읽기만 했지 아직 블랙잭을 계산한 것은 아닙니다. 함수 호출을 연결해야 답을 구합니다.

03 &cards[i]를 cards[i]로 바꾸면 저장할 주소가 아니어서 정상적인 입력 코드가 아닙니다.

C의 표준 입력 · 2차원 배열

입력: 2 3 / 1 2 3 / 4 5 6 → 출력: 21

```c
#include <stdio.h>
int main(void) {
  int rows, cols, grid[8][8], sum = 0;
  if (scanf("%d %d", &rows, &cols) != 2)
    return 1;
  if (rows < 1 || rows > 8 || cols < 1 || cols > 8)
    return 1;
  for (int i = 0; i < rows; i++)
    for (int j = 0; j < cols; j++) {
      if (scanf("%d", &grid[i][j]) != 1)
        return 1;
      sum += grid[i][j];
    }
  printf("%d\n", sum);
  return 0;
}
```

브라우저 P36은 읽기가 끝난 grid를 인자로 받습니다. input_grid.c는 입력부터 실행합니다.

C 2차원 입력: 읽은 즉시 합산

줄별 해설 1/3 · 무엇을 하는가 → 왜 필요한가

```
#include <stdio.h> / int main(void)
```

입출력 함수를 사용하는 독립 실행 프로그램입니다. Python의 줄 단위 입력과 달리 scanf의 %d는 정수를 공백 구분으로 읽습니다.

```
int rows,cols,grid[8][8],sum=0;
```

행·열 수, 최대 8×8 저장 공간, 합계 변수를 준비합니다. 실제 사용할 영역은 입력의 rows×cols입니다.

```
scanf("%d %d",&rows,&cols)!=2 → return 1
```

행과 열 정수 두 개를 정상적으로 읽었는지 확인합니다. 실패한 변수를 그대로 사용하지 않도록 먼저 검사합니다
.

C 2차원 입력: 읽은 즉시 합산

줄별 해설 2/3 · 무엇을 하는가 → 왜 필요한가

```
if (rows<1 || rows>8 || cols<1 || cols>8) return 1;
```

이 프로그램이 지원하는 행·열 범위 1..8을 검사합니다. 입력이 저장 공간보다 크면 배열 바깥을 쓰게 되므로 중단합니다.

```
for i<rows / for j<cols
```

행별로 열을 순회합니다. i=0행의 모든 j를 처리한 뒤 i=1행으로 넘어갑니다.

```
scanf("%d",&grid[i][j]) / sum+=grid[i][j];
```

현재 칸을 성공적으로 읽은 뒤 합계에 더합니다. 실패하면 return 1로 종료하므로 읽지 못한 값을 더하지 않습니다.

C 2차원 입력: 읽은 즉시 합산

줄별 해설 3/3 · 무엇을 하는가 → 왜 필요한가

```
printf("%d\n",sum); return 0;
```

모든 칸을 더한 답을 한 번 출력하고 정상 종료합니다. 출력이 반복문 밖에 있어 중간 합이 여러 줄 나오지 않습니다.

C 2차원 입력: 읽은 즉시 합산

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 입력: 2 3 / 1 2 3 / 4 5 6 → 여섯 정수를 읽고 21을 출력합니다.

02 sum을 i 반복 안에서 0으로 만들면 마지막 행의 합 15만 남게 됩니다.

03 모든 칸을 저장할 필요 없이 합만 원한다면 임시 정수 한 개로 읽을 수도 있습니다. 여기서는 2차원 배열 학습을 위해 저장합니다.

완전 탐색의 연산량 추정

후보 수뿐 아니라 후보 하나를 처리하는 비용도 계산합니다.

01	**약 100만 번**	n^3, n = 100 / 대체로 시도 가능한 규모
02	**약 100만 개**	2^n, n = 20 / 후보 처리 비용도 확인
03	**수억 번 이상**	$n!$, n ≥ 12 / 제한 시간 초과 가능성
04	**수십억 번 이상**	탐색 범위 축소나 다른 알고리즘 검토

실제 실행 시간은 언어, 구현, 연산 종류와 채점 환경에 따라 달라집니다.

서로 다른 숫자 3개의 합

길이 N 리스트에서 서로 다른 3개를 골라 합이 target이 되는 경우의 수.

three_sum.py

```
from itertools import combinations
import sys
input = sys.stdin.readline
N = int(input())
arr = list(map(int, input().split()))
target = int(input())
count = 0
for c in combinations(arr, 3):
    if sum(c) == target:
        count += 1
print(count)
```

입력 / 출력

5
1 2 3 4 5
9

출력: 2

완전 탐색 가능?

N=100 →
$_{100}C_3$ = 161,700
수십만 번 = 매우 안전

세 수의 합: 정답 개수를 세는 전체 프로그램

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
from itertools import combinations / import sys
```

세 위치 선택 도구와 표준 입력 모듈을 가져옵니다. input=sys.stdin.readline으로 줄 입력 함수를 연결합니다.

```
N=int(input()); arr=list(map(int,input().split()))
```

첫 줄에서 원소 수, 다음 줄에서 정수 목록을 읽습니다. 이 코드는 입력에 정확히 N개가 있다고 가정하며 실제 길이를 별도로 검증하지 않습니다.

```
target=int(input()); count=0
```

세 번째 줄에서 목표 합을 읽습니다. count는 조건에 맞는 조합 수이며 합계 변수가 아닙니다.

세 수의 합: 정답 개수를 세는 전체 프로그램

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
for c in combinations(arr,3):
```

서로 다른 위치 세 개의 값을 튜플 c로 받습니다. 순서만 다른 조합은 한 번만 생성합니다.

```
if sum(c)==target: count+=1
```

세 값의 합이 정확히 목표일 때 후보 한 개를 셉니다. sum(c)는 이번 후보만 더하므로 이전 후보 합이 누적되지 않습니다.

```
print(count)
```

모든 후보 검사 후 개수만 출력합니다. 세 값 자체를 출력하라는 문제라면 출력 요구와 다른 답이 됩니다.

세 수의 합: 정답 개수를 세는 전체 프로그램

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 입력 N=5, arr=[2,3,5,7,8], target=15이면 (2,5,8),(3,5,7) 두 가지여서 2를 출력합니다.

02 입력 줄 순서는 N → 배열 → target입니다. N,target이 같은 줄인 블랙잭 예제와 다릅니다.

03 N<3이면 조합이 없어 count=0을 출력합니다. 값 중복이 있다면 위치가 다른 선택은 따로 셉니다.

세 수의 합 개수 · C

P29 / return = 경우의 수

```c
int three_sum_count(const int A[], int n, int target) {
  int count = 0;
  for (int i = 0; i < n; i++)
    for (int j = i + 1; j < n; j++)
      for (int k = j + 1; k < n; k++)
        if (A[i] + A[j] + A[k] == target)
          count++;
  return count;
}
```

코드 읽기

n=100이면 100C3=161,700개입니다. i<j<k의 범위를 지켜 중복을 제외합니다.

C 세 수의 합: 언제 count를 늘릴까?

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
int three_sum_count(..., int target); count=0
```

입력 배열과 목표를 받아 합이 일치하는 위치 조합 개수를 반환합니다. 결과 값들을 별도 배열에 저장하지 않습니다.

```
for i ... / for j=i+1 ... / for k=j+1 ...
```

증가하는 세 위치를 골라 같은 조합의 순서 중복과 위치 재사용을 막습니다.

```
if (A[i]+A[j]+A[k]==target) count++;
```

현재 세 값의 합을 한 번 계산해 일치하면 개수만 1 늘립니다. 조건문 밖에서 늘리면 모든 조합 수를 세게 됩니다.

C 세 수의 합: 언제 count를 늘릴까?

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
return count;
```

세 반복문이 끝난 뒤 개수를 반환합니다. n<3이면 내부 조건문이 실행되지 않아 0입니다. 정수 합과 개수가 int에 들어가는 입력 조건을 전제로 합니다.

C 세 수의 합: 언제 count를 늘릴까?

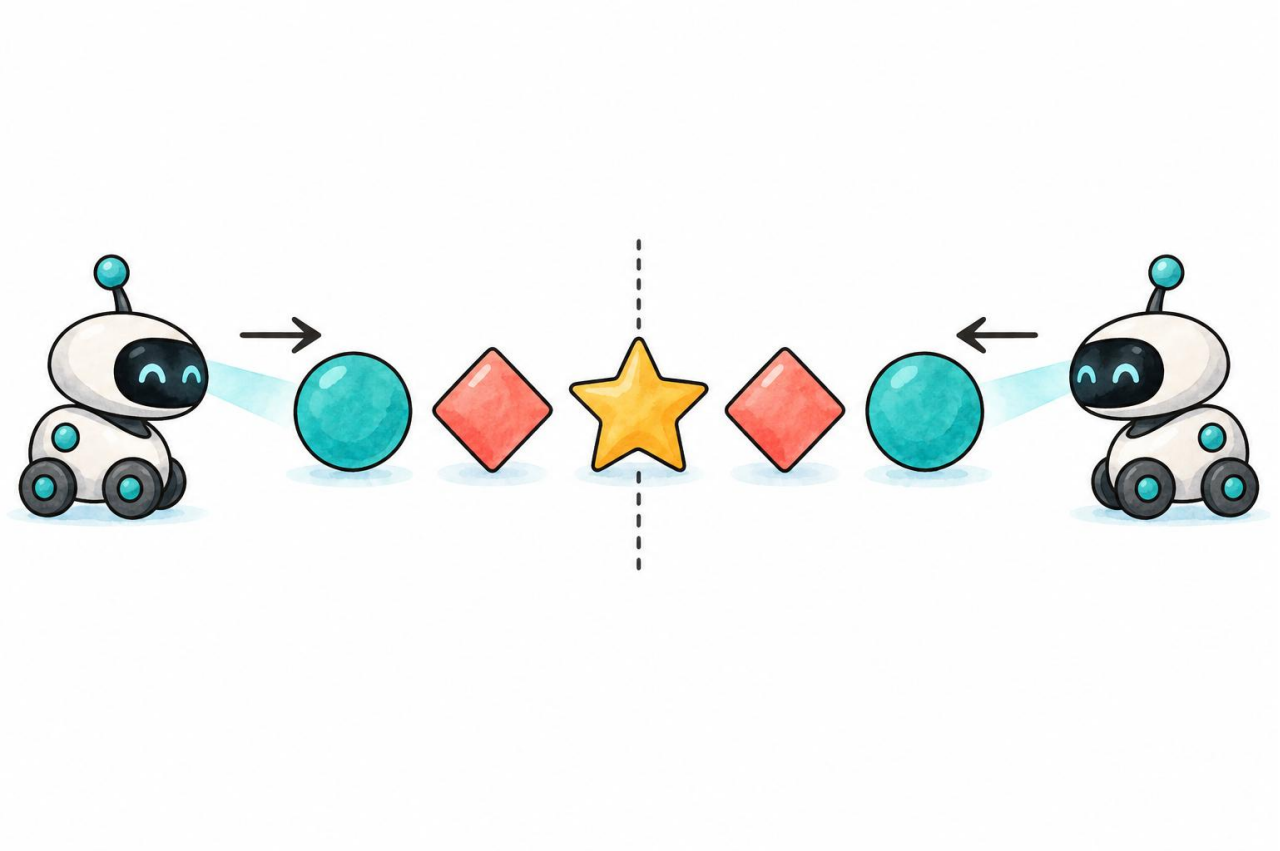
실행 추적 · 결과 검산 · 자주 틀리는 지점

01 A=[2,3,5,7,8], target=15 → 위치 (0,2,4),(1,2,3)이 통과하여 반환값 2.

02 같은 입력에서 세 값 자체를 제출해야 한다면 다음의 three_sum_values 같은 결과 배열 방식이 필요합니다.

03 문제를 읽을 때 답의 형태가 개수인지, 값 목록인지, 최댓값인지 먼저 표시합니다.

세 장을 골라 목표 이하의 최대 합 찾기



카드는 서로 다른 3장

고른 순서는 무시

i < j < k 또는 조합 사용

합이 목표를 넘으면 제외

목표 21에서 7 + 8 + 9 = 24

이 후보는 탈락

통과한 합 중 최대 선택

5 + 7 + 9 = 21

목표 이하이면서 최대

예: 카드 [5, 6, 7, 8, 9], 목표 21. 정답 합은 21이며 이를 만드는 조합은 여러 개일 수 있습니다.

블랙잭: M을 넘지 않는 최대 합

N장 중 3장을 골라 합이 M을 넘지 않으면서 가장 큰 경우를 찾는다.

blackjack.py

```
from itertools import combinations
import sys
input = sys.stdin.readline
def blackjack(cards, target):
    best = 0
    for combo in combinations(cards, 3):
        total = sum(combo)
        if total <= target and total > best:
            best = total
    return best
N, M = map(int, input().split())
cards = list(map(int, input().split()))
print(blackjack(cards, M))
```

예시 입력

5 21
5 6 7 8 9

가능한 조합

5+6+7 = 18
5+6+9 = 20
5+7+9 = 21 ← 최대

출력: 21

Python 블랙잭: 입력과 풀이 함수를 연결

줄별 해설 1/3 · 무엇을 하는가 → 왜 필요한가

```
from itertools import combinations / input=sys.stdin.readline
```

조합 생성 도구와 줄 입력을 준비합니다. import sys를 먼저 실행해야 sys 이름을 사용할 수 있습니다.

```
def blackjack(cards,target): / best=0
```

양의 카드 세 장으로 목표 이하의 가장 큰 합을 찾습니다. 유효한 선택이 없으면 0을 돌려준다는 기준으로 시작합니다.

```
for combo in combinations(cards,3):
```

서로 다른 카드 위치 세 개를 한 번씩 고릅니다. 같은 숫자의 카드가 두 장 있어도 서로 다른 위치라면 함께 선택할 수 있습니다.

Python 블랙잭: 입력과 풀이 함수를 연결

줄별 해설 2/3 · 무엇을 하는가 → 왜 필요한가

```
total=sum(combo)
```

현재 세 장의 합을 구합니다. 합은 조합마다 새로 계산하며 best와 구별합니다. best는 지금까지 본 최댓값입니다
.

```
if total<=target and total>best: best=total
```

목표를 넘지 않고 기존 답보다도 커야 갱신합니다. and는 두 조건이 모두 참이어야 한다는 의미입니다.

```
return best
```

모든 조합이 끝난 뒤 최댓값을 반환합니다. 들여쓰기가 for와 같아야 전체 후보를 검사합니다.

Python 블랙잭: 입력과 풀이 함수를 연결

줄별 해설 3/3 · 무엇을 하는가 → 왜 필요한가

```
N,M=map(int,input().split()); cards=list(...)
```

첫 줄 카드 수와 목표를 읽고 다음 줄 카드 목록을 읽습니다. N과 실제 카드 개수는 같다고 가정합니다.

```
print(blackjack(cards,M))
```

읽은 카드와 목표를 함수에 전달하고 반환된 최적 합을 출력합니다. 함수를 정의만 하고 호출하지 않으면 풀이가 실행되지 않습니다.

Python 블랙잭: 입력과 풀이 함수를 연결

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 cards=[5,6,7,8], target=20: 18→19→20은 갱신, 21은 목표 초과로 탈락. 반환 20.

02 합이 target과 같으면 더 좋아질 수 없으므로 조기 반환도 가능하지만 현재 기본형은 모든 조합을 검사합니다.

03 목표가 10이고 모든 세 장 합이 10을 넘으면 0입니다. 문제의 미발견 출력 규칙과 맞는지 확인합니다.

블랙잭 최대 합 · C

P30 / return = 최대 합 또는 0

```c
int blackjack(const int A[], int n, int target) {
  int best = 0;
  for (int i = 0; i < n; i++)
    for (int j = i + 1; j < n; j++)
      for (int k = j + 1; k < n; k++) {
        int total = A[i] + A[j] + A[k];
        if (total <= target && total > best)
          best = total;
      }
  return best;
}
```

코드 읽기

조건 검증 total<=target 후 best와 비교합니다. 최적값을 찾아야 하므로 첫 번째 유효 조합에서 종료하면 안 됩니다.

C 블랙잭: 유효성 검사와 비교를 동시에

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
int blackjack(..., int target); int best=0;
```

양의 카드에서 목표 이하 최대 합을 반환합니다. 후보가 없을 때의 기준 0을 유지합니다.

```
for i ... / for j=i+1 ... / for k=j+1 ...
```

서로 다른 세 위치를 오름차순으로 골라 중복된 순서를 제거합니다. n개 중 세 장을 고릅니다.

```
int total=A[i]+A[j]+A[k];
```

이번 조합의 합을 계산합니다. total은 이 후보의 값이고 best는 이전 후보까지의 최선입니다.

C 블랙잭: 유효성 검사와 비교를 동시에

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
if (total<=target && total>best) best=total;
```

상한을 지키면서 더 좋아진 경우에만 저장합니다. 둘 중 하나라도 거짓이면 기존 best가 유지됩니다.

```
return best;
```

탐색이 모두 끝난 뒤 최댓값을 반환합니다. 양의 카드 합이 int 범위에 들어가는 조건으로 사용합니다.

C 블랙잭: 유효성 검사와 비교를 동시에

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 합 18,19,20,21과 target=20이면 best는 0,18,19,20,20 순서입니다.

02 total>best만 검사하면 목표 초과 21을 잘못 선택합니다. total<=target만 보고 저장하면 더 작은 마지막 후보로 퇴보할 수 있습니다.

03 이 두 조건은 최적화 문제에서 유효한가·더 좋은가라는 서로 다른 질문입니다.

완전 탐색 연습 유형

01	**반복문**	$O(n^2)$, $O(n^3)$ / 모든 쌍과 세 원소 비교
02	**문자열 탐색**	$O(nm)$ / 패턴 매칭과 문자열 비교
03	**부분집합**	2^n개 후보 / 비트마스크, 선택·비선택
04	**조합 탐색**	nCr개 후보 / combinations, 합 조건
05	**2차원 배열**	$O(n^2m^2)$ 예 / 격자와 행렬 패턴
06	**최적화**	문제별로 다름 / 최솟값·최댓값 갱신

이중 · 삼중 반복문 연습

문제 1: 모든 (i,j) 쌍의 차이 (i<j)

pairs.py

```
A = [3, 1, 4, 2]
n = len(A)
for i in range(n - 1):
    for j in range(i + 1, n):
        print(f"{A[j]-A[i]}")
```

문제 1은 이중 반복문으로 모든 쌍을 순회하는 가장 기본적인 $O(n^2)$ 패턴입니다.

문제 2: 세 수의 합이 K (i<j<k)

triple.py

```
A = [2,3,5,7,8]; K = 15
for i in range(len(A)-2):
  for j in range(i+1, len(A)-1):
    for k in range(j+1, len(A)):
      if A[i]+A[j]+A[k] == K:
        print((A[i],A[j],A[k]))
```

출력

(2, 5, 8)
(3, 5, 7)

반복문 복습: 값의 차이와 세 값 출력

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
A=[3,1,4,2]; n=len(A)
```

첫 예제는 입력 순서를 그대로 유지합니다. 정렬하지 않았으므로 뒤쪽 값이 앞쪽 값보다 작을 수 있습니다.

```
for i in range(n-1): / for j in range(i+1,n):
```

첫 예제는 서로 다른 두 위치를 한 번씩 고릅니다. 가능한 쌍의 수는 6개입니다.

```
print(f"{A[j]-A[i]}")
```

뒤 위치의 값에서 앞 위치의 값을 뺍니다. 절댓값이 아니므로 (3,1)의 차이는 -2입니다. f 문자열이 결과를 출력용 문자열로 바꿉니다.

반복문 복습: 값의 차이와 세 값 출력

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
A=[2,3,5,7,8]; K=15
```

여기서부터 별도의 두 번째 프로그램입니다. 세미콜론은 같은 줄의 두 문장을 구분할 뿐 특별한 자료구조가 아닙니다.

```
for i<len(A)-2 / for j=i+1 .. len(A)-2 / for k=j+1 ..
```

세 값 예제에서는 뒤에 필요한 위치 수만큼 상한을 줄였습니다. 여전히 핵심 조건은 i<j<k입니다.

```
if A[i]+A[j]+A[k]==K: print((A[i],A[j],A[k]))
```

합이 목표인 경우 세 값을 튜플로 출력합니다. 앞의 개수 세기 예제와 달리 count를 늘리지 않고 해 자체를 보여줍니다.

반복문 복습: 값의 차이와 세 값 출력

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 첫 예제 출력: -2, 1, -1, 3, 1, -2 순서입니다. 이 결과는 절댓값 차이 목록이 아닙니다.

02 두 번째 예제 출력: (2,5,8), (3,5,7)입니다.

03 range의 끝은 제외됩니다. len(A)-2는 첫 위치가 마지막으로 가질 수 있는 인덱스보다 1 큰 값입니다.

모든 쌍의 차이 · C

P31 / out = 차이 목록, return = 개수 (최대 435)

```c
int pair_differences(const int A[], int n, int out[]) {
  int count = 0;
  for (int i = 0; i < n; i++)
    for (int j = i + 1; j < n; j++)
      out[count++] = A[j] - A[i];
  return count;
}
```

코드 읽기

[3,1,4,2]의 첫 차이는 1-3=-2입니다.

C 차이 목록: 부호까지 결과의 일부

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
int pair_differences(..., int out[]); count=0
```

각 위치 쌍에서 뒤 값-앞 값을 저장하는 함수입니다. 거리를 찾는 closest_pair와 달리 절댓값을 취하지 않습니다.

```
for i ... / for j=i+1 ...
```

입력 순서에서 앞·뒤 위치를 고정합니다. 배열을 정렬하면 어떤 값을 뒤에서 빼는지가 달라져 결과 순서와 부호가 바뀔 수 있습니다.

```
out[count++]=A[j]-A[i];
```

현재 차이를 결과 다음 칸에 쓰고 결과 수를 늘립니다. 여기서는 모든 쌍이 결과이므로 if가 없습니다.

C 차이 목록: 부호까지 결과의 일부

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
return count;
```

nC2개 결과의 길이를 반환합니다. 호출자는 충분한 out 용량과 뺄셈이 int 범위인 입력을 보장합니다.

C 차이 목록: 부호까지 결과의 일부

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 A=[3,1,4,2] → out=[-2,1,-1,3,1,-2], 반환값 6.

02 절댓값을 요구하는 문제로 바뀌면 계산식도 바꿔야 합니다. 함수 이름보다 문제의 정확한 정의를 먼저 읽습니다.

03 n=1이면 쌍이 없으므로 0을 반환하고 out은 읽지 않습니다.

합이 K인 세 수 나열 · C

P32 / out[][3] = 세 수, return = 행 수 (최대 1140)

```c
int three_sum_values(const int A[], int n, int target, int out[][3]) {
  int count = 0;
  for (int i = 0; i < n; i++)
    for (int j = i + 1; j < n; j++)
      for (int k = j + 1; k < n; k++)
        if (A[i] + A[j] + A[k] == target) {
          out[count][0] = A[i];
          out[count][1] = A[j];
          out[count++][2] = A[k];
        }
  return count;
}
```

코드 읽기

조건을 통과했을 때만 out[count]에 저장한 뒤 count를 증가시킵니다.

C 세 값 출력: 개수와 목록을 함께 만들기

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
int three_sum_values(..., int out[][3]); count=0
```

합이 목표인 세 값들을 결과 행에 담습니다. 반환값은 행 개수이며 한 행은 세 정수입니다.

```
for i ... / for j=i+1 ... / for k=j+1 ...
```

세 위치를 증가하는 순서로 골라 순서 중복 없이 모든 조합을 검사합니다.

```
if (A[i]+A[j]+A[k]==target) {
```

목표와 정확히 같은 합만 통과합니다. target 이하인 가장 큰 값 문제와 조건이 다릅니다.

C 세 값 출력: 개수와 목록을 함께 만들기

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
out[count][0]=A[i]; out[count][1]=A[j];
```

같은 결과 행에 첫째와 둘째 값을 씁니다. count는 아직 유지합니다.

```
out[count++][2]=A[k];
```

셋째 값을 쓴 후 다음 결과 행으로 이동합니다. 결과를 담을 수 있는 최대 행 수는 nC3입니다.

```
return count;
```

out의 유효 행 수를 반환합니다. 입력 합과 결과 수가 int에 들어가는 수업 범위를 가정합니다.

C 세 값 출력: 개수와 목록을 함께 만들기

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 A=[2,3,5,7,8], target=15 → out[0]=[2,5,8], out[1]=[3,5,7], 반환 2.

02 count만 반환하는 함수로는 어떤 조합이 정답인지 알 수 없습니다. 그래서 출력 배열이 추가되었습니다.

03 같은 값의 결과가 여러 번 나올 수 있는지는 위치 중복과 값 중복을 구분해서 판단합니다.

양끝에서 같은 모양인지 비교하기

그림의 원·마름모·별·마름모·원은 대칭입니다. 가운데 원소 하나는 남아도 됩니다.

앞뒤 순서가 같으면 회문

level, noon은 회문

hello는 회문이 아님

양끝에서 안쪽으로 이동

첫째와 마지막, 둘째와 끝에서

둘째를 차례로 비교

비교 기준을 먼저 확인

대소문자를 구분하면

Racecar는 회문이 아님

회문(Palindrome) 찾기

앞뒤로 읽어도 같은 문자열 찾기: s == s[::-1] 이면 회문.

palindrome.py

```
n = int(input())
result = []
for _ in range(n):
    s = input().strip()
    if s == s[::-1]:      # 앞뒤로 읽어도 같으면 회문
        result.append(s)
for r in result:
    print(r)
```

s[::-1] 슬라이싱 → 문자열을 뒤집는 파이썬 핵심 기법

입력 / 출력

입력:
5
level
hello
Racecar
noon
world

출력:
level
noon

회문 Python: 역순 문자열과 비교

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
n=int(input()); result=[]
```

읽을 단어 수를 정수로 받고 통과한 단어를 담을 목록을 준비합니다.

```
for _ in range(n):
```

단어를 n개 읽습니다. 반복 번호를 사용하지 않으므로 _라는 이름을 쓰는 관례입니다.

```
s=input().strip()
```

한 줄을 읽고 양 끝 공백·개행을 제거합니다. 이 예제는 공백 자체가 의미를 갖지 않는 단어 입력입니다. 문장 공백을 보존해야 하면 다른 처리가 필요합니다.

회문 Python: 역순 문자열과 비교

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
if s==s[::-1]:
```

슬라이스의 세 번째 값 -1은 뒤로 한 칸씩 이동하라는 뜻입니다. 시작·끝을 생략했으므로 문자열 전체를 뒤집고 원문과 비교합니다.

```
result.append(s)
```

앞뒤가 같으면 원래 단어를 저장합니다. 뒤집은 결과가 아니라 입력 문자열을 모으는 코드입니다.

```
for r in result: print(r)
```

입력 검사를 모두 끝낸 뒤 회문만 차례대로 출력합니다. 회문이 없으면 출력할 줄도 없습니다.

회문 Python: 역순 문자열과 비교

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 level → level과 같아 저장; apple → elppa와 달라 탈락; a → a와 같아 저장.

02 단어 길이 L에서 역순 복사와 비교는 O(L)입니다. 글자 하나나 빈 문자열도 이 판정에서는 회문입니다.

03 입력 문자열의 앞뒤 공백까지 판정해야 하는 문제라면 strip()이 내용을 바꾼다는 점을 주의합니다.

회문만 골라내기 · C

P33 / out = 회문인 문자열의 인덱스, return = 개수 (최대 30)

```c
int palindrome_words(const char *words[], int n, int out[]) {
  int count = 0;
  for (int i = 0; i < n; i++) {
    int left = 0, right = (int)strlen(words[i]) - 1, ok = 1;
    while (left < right)
      if (words[i][left++] != words[i][right--]) {
        ok = 0;
        break;
      }
    if (ok)
      out[count++] = i;
  }
  return count;
}
```

C에서는 양 끝의 인덱스를 안쪽으로 옮기며 비교합니다. Racecar와 racecar를 구분하세요.

C 회문: 양쪽 포인터를 가운데로

줄별 해설 1/3 · 무엇을 하는가 → 왜 필요한가

```
int palindrome_words(const char *words[], int n, int out[])
```

words[i]는 i번째 문자열의 시작 주소입니다. 결과는 단어 복사본이 아니라 회문인 단어의 인덱스를 저장합니다. 기본 ASCII 단어를 가정합니다.

```
int count=0; for (int i=0; i<n; i++)
```

결과 수를 준비하고 단어마다 새 검사를 시작합니다. 단어 개수 n과 단어 길이는 다른 값입니다.

```
int left=0, right=(int)strlen(words[i])-1, ok=1;
```

맨 앞과 맨 뒤 글자 위치를 정하고 일단 회문이라고 가정합니다. 문자열 종료 문자 '₩0'는 비교 대상에서 제외합니다.

C 회문: 양쪽 포인터를 가운데로

줄별 해설 2/3 · 무엇을 하는가 → 왜 필요한가

```
while (left<right)
```

두 위치가 만나거나 교차하기 전까지만 비교합니다. 홀수 길이의 가운데 글자는 자기 자신이므로 별도 비교할 필요가 없습니다.

```
if (words[i][left++] != words[i][right--])
```

현재 양 끝 글자를 비교한 뒤 left는 1 증가, right는 1 감소합니다. 비교와 이동을 나눠 읽으면 진행 방향을 이해하기 쉽습니다.

```
ok=0; break;
```

한 쌍이라도 다르면 회문이 아니므로 표시하고 현재 단어의 while을 끝냅니다. 다음 단어를 검사하는 for는 계속됩니다.

C 회문: 양쪽 포인터를 가운데로

줄별 해설 3/3 · 무엇을 하는가 → 왜 필요한가

```
if (ok) out[count++]=i; / return count;
```

통과한 단어의 번호를 저장하고 최종 개수를 반환합니다. 출력하려면 호출자가 words[out[k]]를 읽습니다.

C 회문: 양쪽 포인터를 가운데로

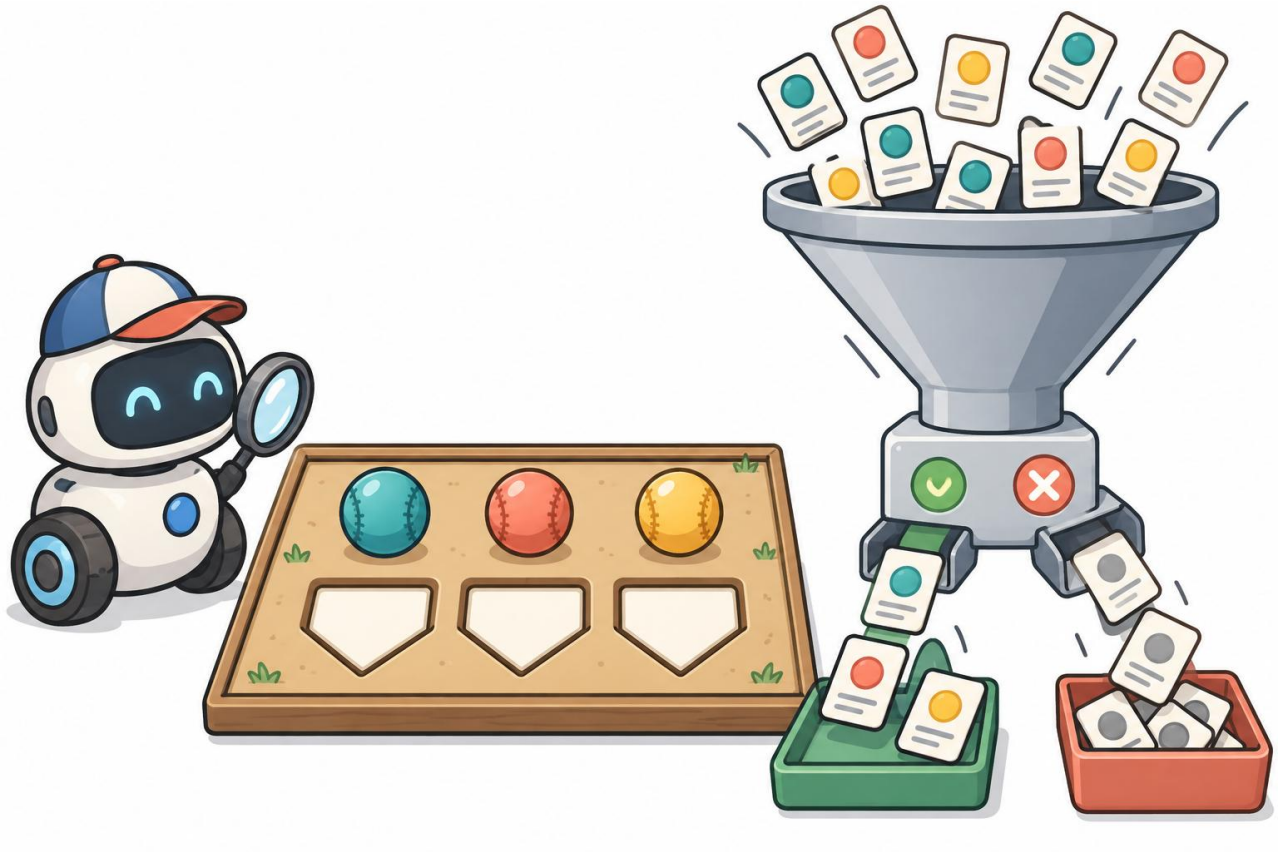
실행 추적 · 결과 검산 · 자주 틀리는 지점

01 level: (0,4)의 l=l → (1,3)의 e=e → (2,2)에서 멈춤 → 통과.

02 apple: (0,4)의 a≠e에서 즉시 탈락. words=["level","apple","a"]이면 out=[0,2], 반환 2.

03 C의 char는 UTF-8 한글 한 글자와 같지 않습니다. 이 양끝 바이트 비교 코드는 한글 회문 처리용으로 그대로 쓰면 안 됩니다.

무게 제한 안에서 가치가 가장 큰 묶음



물건은 통째로 선택

각 물건을 한 번 넣거나 제외
일부만 넣는 선택은 없음

무게와 가치는 다른 기준

총무게는 제한 이하인지 검사
총가치는 최댓값과 비교

예: 용량 10

(무게 4, 가치 5) + (6, 8)
총무게 10, 총가치 13

예제의 네 물건: (4,5), (6,8), (3,3), (5,6). 빈 선택도 허용하며 최대 가치는 13입니다.

배낭 문제 (0/1 Knapsack): O(N·2ⁿ)

N=20이면 부분집합 약 100만 개. 후보별 처리 비용과 실제 시간 제한도 확인합니다.

knapsack.py

```
from itertools import combinations
N, W = map(int, input().split())
items = [tuple(map(int, input().split()))
         for _ in range(N)]     # (무게, 가치)
max_val = 0
for r in range(1, N + 1):        # 1~N개 선택
    for combo in combinations(items, r):
        w = sum(it[0] for it in combo)
        v = sum(it[1] for it in combo)
        if w <= W:
            max_val = max(max_val, v)
print(max_val)
```

예시 입력

4 10
4 5
6 8
3 3
5 6

출력: 13

발전: 완전 탐색 O(N·2ⁿ) → 동적 계획법 O(NW)으로 개선 가능

배낭 Python: 무게는 제약, 가치는 목표

줄별 해설 1/3 · 무엇을 하는가 → 왜 필요한가

```
from itertools import combinations
```

물건을 고르는 모든 조합을 생성합니다. 같은 물건을 한 선택 안에서 여러 번 사용하는 무한 배낭 문제가 아닙니다.

```
N,W=map(int,input().split())
```

물건 수 N과 최대 무게 W를 읽습니다. W는 최대화하려는 답이 아니라 넘으면 안 되는 제한입니다.

```
items=[tuple(map(int,input().split())) for _ in range(N)]
```

N줄의 무게·가치를 튜플로 저장합니다. it[0]은 무게, it[1]은 가치이며 두 값의 순서를 바꾸면 문제 의미가 달라집니다.

배낭 Python: 무게는 제약, 가치는 목표

줄별 해설 2/3 · 무엇을 하는가 → 왜 필요한가

```
max_val=0; for r in range(1,N+1):
```

1개부터 N개까지 고르는 크기를 바꿉니다. 아무것도 안 고르는 가치 0은 초기값이 대표합니다. 용량과 무게는 음수가 아니라고 가정합니다.

```
for combo in combinations(items,r):
```

정확히 r개 물건을 고른 후보를 한 개씩 받습니다. 모든 r을 돌면 공집합을 제외한 2^n-1개 선택을 검사합니다.

```
w=sum(it[0] for it in combo)
```

현재 후보에 들어 있는 물건의 무게만 합합니다. 생성식의 it는 현재 물건 튜플 하나입니다.

배낭 Python: 무게는 제약, 가치는 목표

줄별 해설 3/3 · 무엇을 하는가 → 왜 필요한가

```
v=sum(it[1] for it in combo)
```

같은 후보의 가치들을 합합니다. 무게 합 w와 가치 합 v를 따로 계산해 제한과 목적을 분리합니다.

```
if w<=W: max_val=max(max_val,v)
```

용량 안에 드는 경우에만 현재 최선과 후보 가치를 비교합니다. 무거워서 탈락한 후보의 높은 가치는 답으로 쓸 수 없습니다.

```
print(max_val)
```

모든 선택을 비교한 뒤 최대 가치를 출력합니다. 선택한 물건 목록도 필요하다면 갱신할 때 combo를 함께 기록해야 합니다.

배낭 Python: 무게는 제약, 가치는 목표

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 물건 (무게,가치)=[(2,3),(3,4),(4,5)], 용량5: 첫째+둘째는 무게5 가치7로 가능, 첫째+셋째는 무게6으로 탈락.

02 단일 물건 가치의 최댓값 5보다 두 물건을 함께 고른 7이 좋습니다. 가장 가치 큰 물건만 고르면 놓칩니다.

03 선택마다 물건을 더하는 비용까지 포함한 전체 시간은 $O(n \cdot 2^n)$입니다. 후보 수 2^n과 처리 시간을 구별합니다.

0/1 배낭 완전 탐색 · C

P34 / return = 최대 가치

```c
int knapsack(const int weights[], const int values[], int n, int capacity) {
  int best = 0;
  for (unsigned mask = 0; mask < (1u << n); mask++) {
    int weight = 0, value = 0;
    for (int j = 0; j < n; j++)
      if (mask & (1u << j)) {
        weight += weights[j];
        value += values[j];
      }
    if (weight <= capacity && value > best)
      best = value;
  }
  return best;
}
```

마스크마다 무게와 가치를 따로 합산합니다. 모든 원소를 검사하는 구현의 시간은 O(n*2^n)입니다.

C 배낭: 하나의 마스크로 두 합을 계산

줄별 해설 1/3 · 무엇을 하는가 → 왜 필요한가

```
int knapsack(weights,values,n,capacity); best=0
```

같은 인덱스의 무게와 가치는 같은 물건입니다. 초기 0은 아무것도 고르지 않는 가치이며 무게·용량은 음수가 아니라고 가정합니다.

```
for (unsigned mask=0; mask<(1u<<n); mask++)
```

공집합까지 모든 선택을 만듭니다. n은 unsigned 비트 폭보다 작고 완전 탐색 가능한 작은 입력이어야 합니다.

```
int weight=0,value=0;
```

새 후보의 무게와 가치를 0부터 계산합니다. 매 마스크에서 초기화하므로 후보끼리 누적되지 않습니다.

C 배낭: 하나의 마스크로 두 합을 계산

줄별 해설 2/3 · 무엇을 하는가 → 왜 필요한가

```
for j ... / if (mask & (1u<<j)) {
```

j번째 물건을 선택한 비트이면 그 물건을 두 합에 반영합니다. 선택 비트 한 개로 무게와 가치를 함께 결정합니다.

```
weight+=weights[j]; value+=values[j];
```

같은 물건의 무게와 가치를 각각 더합니다. 인덱스를 서로 다르게 쓰면 존재하지 않는 가상의 물건을 계산하게 됩니다.

```
if (weight<=capacity && value>best) best=value;
```

무게 제한을 통과하고 가치가 더 큰 후보만 저장합니다. best에는 무게가 아니라 가치를 저장합니다.

C 배낭: 하나의 마스크로 두 합을 계산

줄별 해설 3/3 · 무엇을 하는가 → 왜 필요한가

```
return best;
```

최댓값을 반환합니다. 선택 물건 자체를 반환하지 않으며, 필요한 경우 best_mask도 갱신하여 복원할 수 있습니다.

C 배낭: 하나의 마스크로 두 합을 계산

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 weights=[2,3,4], values=[3,4,5], capacity=5: mask=3(011)은 무게5 가치7로 best=7.

02 mask=5(101)은 무게6 가치8이지만 탈락합니다. 최종 반환은 7입니다.

03 중간 무게 초과로 멈추는 최적화는 남은 무게가 음수가 아닐 때만 안전합니다. 현재 코드는 기본형 전체 합산입니다.

힌트와 맞지 않는 숫자 후보를 걸러내기

숫자와 위치가 모두 같음

스트라이크로 계산

숫자는 같지만 위치가 다름

볼로 계산

모든 힌트를 통과해야 한다

하나라도 어긋나면 탈락

통과한 후보의 개수를 센다

이 자료의 규칙: 0~9 중 서로 다른 숫자 3개, 앞자리 0 허용. 후보는 10 × 9 × 8 = 720개입니다.

숫자 야구 (Baseball)

0~9에서 서로 다른 숫자 3개로 구성 → 스트라이크/볼 힌트로 정답 후보를 추론.

baseball.py

```
from itertools import permutations
def get_sb(secret, guess):
    s = sum(1 for i in range(3) if secret[i] == guess[i])
    b = sum(1 for d in guess if d in secret) - s
    return s, b
hints = []
n = int(input())
for _ in range(n):
    line = input().split()
    hints.append((line[0], int(line[1]), int(line[2])))
count = 0
for perm in permutations('0123456789', 3):
    secret = ''.join(perm)
    if all(get_sb(secret, h[0]) == (h[1], h[2]) for h in hints):
        count += 1
print(count)
```

숫자 야구 Python: 모든 힌트와 일치하는 후보

줄별 해설 1/4 · 무엇을 하는가 → 왜 필요한가

```
from itertools import permutations / def get_sb(secret,guess):
```

중복 없는 세 문자의 후보를 만들고 두 문자열의 스트라이크·볼을 계산합니다. 이 수업 규칙은 0을 포함하고 맨 앞 0도 허용합니다.

```
s=sum(1 for i in range(3) if secret[i]==guess[i])
```

같은 위치의 같은 숫자마다 1을 더해 스트라이크 수를 셉니다. 숫자의 크기를 더하는 것이 아닙니다.

```
b=sum(1 for d in guess if d in secret)-s
```

공통 숫자 수에는 스트라이크도 포함돼 있으므로 s를 빼면 위치만 다른 볼 수가 됩니다. 추측도 서로 다른 세 숫자라는 조건이 필요합니다.

숫자 야구 Python: 모든 힌트와 일치하는 후보

줄별 해설 2/4 · 무엇을 하는가 → 왜 필요한가

```
return s,b
```

계산 결과를 두 값의 튜플로 돌려줍니다. 힌트의 (스트라이크,볼)과 그대로 비교할 수 있습니다.

```
hints=[]; n=int(input()); for _ in range(n):
```

힌트 목록과 개수를 준비하고 힌트 줄을 n개 읽습니다. n은 후보 숫자의 길이 3이 아니라 힌트 개수입니다.

```
line=input().split(); hints.append((line[0],int(line[1]),int(line[2])))
```

추측 숫자는 문자열로 보존하고 스트라이크·볼 개수만 정수로 바꿉니다. "012"를 int로 바꾸면 앞의 0을 잃습니다
.

숫자 야구 Python: 모든 힌트와 일치하는 후보

줄별 해설 3/4 · 무엇을 하는가 → 왜 필요한가

```
count=0; for perm in permutations('0123456789',3):
```

서로 다른 세 숫자의 순서 있는 후보 10×9×8=720개를 만듭니다. perm은 세 문자 튜플입니다.

```
secret=''.join(perm)
```

구분자 없는 빈 문자열로 세 문자를 이어 붙입니다. ('0','1','2')를 "012"로 만들어 get_sb에 전달합니다.

```
if all(get_sb(secret,h[0])==(h[1],h[2]) for h in hints):
```

현재 후보가 모든 힌트의 스트라이크·볼과 맞는지 검사합니다. 하나라도 어긋나면 all은 거짓이고 이후 힌트 평가를 멈춥니다.

숫자 야구 Python: 모든 힌트와 일치하는 후보

줄별 해설 4/4 · 무엇을 하는가 → 왜 필요한가

```
count+=1 / print(count)
```

모든 힌트를 통과한 후보만 하나 세고 마지막에 총개수를 출력합니다. 실제 정답 하나를 맞히는 것이 아니라 남은 가능성을 세는 문제입니다.

숫자 야구 Python: 모든 힌트와 일치하는 후보

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 secret="123", guess="132": 같은 위치는 '1' 하나라 s=1, 공통 숫자는 3개라 b=3-1=2입니다.

02 힌트가 "012 3 0" 하나면 가능한 후보는 "012" 한 개입니다. 힌트가 0개면 all의 빈 검사 결과가 참이라 720개가 남습니다.

03 다른 문제에서 숫자 1~9만 허용하면 후보는 9×8×7=504개입니다. 원문 규칙에 맞춰 후보 집합부터 바꿔야 합니다.

숫자 야구 후보 세기 · C (1/2)

P35 / return = 모든 힌트를 만족하는 후보 수

```c
int baseball_count(const char *guesses[], const int strikes[], const int balls[],
                   int n) {
  int count = 0;
  for (int a = 0; a < 10; a++)
    for (int b = 0; b < 10; b++)
      for (int c = 0; c < 10; c++) {
        if (a == b || a == c || b == c)
          continue;
        char candidate[3] = {'0' + a, '0' + b, '0' + c};
        int ok = 1;
```

다음 장에서 각 후보의 스트라이크·볼 조건을 검사합니다.

숫자 야구 C ① 후보와 검사 상태 준비

줄별 해설 1/2 · 무엇을 하는가 → 왜 필요한가

```
baseball_count(guesses,strikes,balls,n)
```

같은 인덱스 h의 추측 문자열·스트라이크·볼이 한 힌트입니다. n은 힌트 수이고 반환값은 가능한 후보 수입니다.

```
int count=0; / for a,b,c=0..9
```

세 자리 숫자 후보를 독립적으로 만듭니다. 이 단계에서는 1,000개이지만 중복 숫자를 다음 조건에서 제거합니다
.

```
if (a==b || a==c || b==c) continue;
```

어느 두 자리라도 같으면 현재 c 후보를 건너뜁니다. 중복 없는 후보 720개만 남습니다.

숫자 야구 C ① 후보와 검사 상태 준비

줄별 해설 2/2 · 무엇을 하는가 → 왜 필요한가

```
char candidate[3]={'0'+a,'0'+b,'0'+c};
```

숫자를 문자로 바꾸어 세 칸에 담습니다. 이 배열에는 종료 문자가 없으며 다음 코드가 인덱스 0..2만 직접 읽기 때문에 가능합니다.

```
int ok=1;
```

이번 후보가 아직 모든 힌트와 모순되지 않았다는 표시입니다. 다음 후보마다 다시 1로 시작해야 이전 탈락 결과가 남지 않습니다.

숫자 야구 C ① 후보와 검사 상태 준비

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 a=0,b=1,c=2이면 candidate는 '0','1','2'이고 유효 후보입니다. a=0,b=1,c=1은 continue로 탈락합니다.

02 candidate[3]에는 '₩0' 공간이 없습니다. 따라서 strlen(candidate)나 printf("%s",candidate)에 그대로 넘기면 안 됩니다.

03 이 페이지는 함수 앞부분입니다. 다음 페이지의 힌트 검사 코드와 합쳐야 중괄호가 완성됩니다.

숫자 야구 후보 세기 · C (2/2)

P35 / return = 모든 힌트를 만족하는 후보 수

```
        for (int h = 0; h < n && ok; h++) {
          int s = 0, common = 0;
          for (int i = 0; i < 3; i++) {
            if (candidate[i] == guesses[h][i])
              s++;
            for (int j = 0; j < 3; j++)
              if (candidate[i] == guesses[h][j])
                common++;
          }
          if (s != strikes[h] || common - s != balls[h])
            ok = 0;
        }
        if (ok)
          count++;
      }
  return count;
}
```

숫자를 문자열로 비교해야 012를 보존합니다. 공통 숫자 수에서 스트라이크를 빼면 볼입니다.

숫자 야구 C ② 힌트 검사와 후보 수 세기

줄별 해설 1/3 · 무엇을 하는가 → 왜 필요한가

```
for (int h=0; h<n && ok; h++)
```

힌트를 하나씩 보되 이미 모순된 후보라면 멈춥니다. 이 조기 종료는 뒤의 힌트가 앞의 모순을 없앨 수 없으므로 안전합니다.

```
int s=0, common=0;
```

현재 힌트에 대한 스트라이크와 공통 숫자 수를 초기화합니다. 힌트가 바뀔 때마다 다시 0이어야 합니다.

```
for i=0..2 / if (candidate[i]==guesses[h][i]) s++;
```

같은 위치의 문자가 같으면 스트라이크를 셉니다. 세 위치를 각각 검사합니다.

숫자 야구 C ② 힌트 검사와 후보 수 세기

줄별 해설 2/3 · 무엇을 하는가 → 왜 필요한가

```
for j=0..2 / if (candidate[i]==guesses[h][j]) common++;
```

후보의 현재 문자가 추측의 어디에 있는지 확인해 공통 숫자를 셉니다. 각 추측의 숫자는 서로 다르다고 가정합니다.

```
if (s!=strikes[h] || common-s!=balls[h]) ok=0;
```

스트라이크 또는 볼 중 하나라도 힌트와 다르면 후보를 탈락시킵니다. common에는 스트라이크도 포함되어 s를 빼야 볼이 됩니다.

```
if (ok) count++;
```

힌트를 모두 통과한 후보만 한 번 셉니다. 이 줄은 h 반복 밖이고 세 숫자를 고르는 반복 안입니다.

숫자 야구 C ② 힌트 검사와 후보 수 세기

줄별 해설 3/3 · 무엇을 하는가 → 왜 필요한가

```
return count;
```

세 자리 후보를 모두 검사한 뒤 가능한 개수를 반환합니다. 마지막 }들은 후보 반복과 함수의 범위를 닫습니다.

숫자 야구 C ② 힌트 검사와 후보 수 세기

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 candidate="123", guess="132": s=1, common=3, ball=2. 힌트 1S2B이면 통과, 2S1B이면 탈락합니다.

02 한 힌트에서 ok=0이 되면 h<n && ok가 거짓이 되어 나머지 힌트를 보지 않습니다.

03 후보별 힌트 최대 n개, 각 힌트는 세 자리의 고정 비교이므로 이 규칙에서는 O(720n)입니다.

완전 탐색 코딩 패턴 총정리

자주 나오는 코드 패턴 5가지: 이 형태만 익히면 대부분의 문제에 대응할 수 있습니다.

패턴	코드 형태	대표 문제
① 단일 for	for i in range(n):	약수·소수 판별
② 이중 for	for i in range(n-1): for j in range(i+1,n):	최근접 쌍, 두 수 합
③ combinations	for c in combinations(arr, r):	r개 선택 조합
④ permutations	for p in permutations(arr, r):	순서 있는 r개
⑤ 비트마스크	for i in range(1 << n):	모든 부분집합

Python · C 완전 탐색 패턴 대응표

Python	C11	실습
range(n)	for (int i=0; i<n; i++)	P14
combinations(A, 3)	i < j < k 반복문	P17 / P29
permutations(A, r)	used 배열 + 재귀	P18
combinations(A, r)	start 인덱스 + 재귀	P19
range(1 << n)	mask < (1u << n)	P24 / P34
s == s[::-1]	양 끝 문자를 안쪽으로 비교	P33

nPr = n!/(n-r)! · nCr = n!/(r!(n-r)!) · 부분집합 생성·기록은 $O(n \cdot 2^n)$

2장 핵심 정리

01	가능한 모든 후보를 체계적으로 검사한다.
02	선택 변수와 반복문의 범위를 확인한다.
03	후보 생성, 조건 검사, 해 선택으로 구현한다.
04	Python itertools 또는 C 반복문·재귀로 순열과 조합을 만든다.
05	n!, 2^n 등 후보 수와 후보당 처리 비용을 함께 계산한다.
06	가지치기와 문제 분할, 결과 재사용으로 계산을 줄인다.
07	입력 제한과 경우의 수를 계산해 완전 탐색 가능 여부를 판단한다.

완전 탐색의 시간 복잡도는?

quiz1.py

```
# 코드 A
for i in range(n):
    for j in range(n):
        pass

# 코드 B
for perm in permutations(arr, n):
    pass

# 코드 C
for i in range(1 << n):
    pass
```

A $O(n^2)$

이중 for문

B $O(n!)$

모든 순열

C $O(2^n)$

모든 부분집합

복잡도 퀴즈 해설: 후보 수와 만드는 비용

줄별 해설 1/1 · 무엇을 하는가 → 왜 필요한가

```
for i in range(n): / for j in range(n): pass
```

바깥 n회마다 안쪽 n회가 실행됩니다. 본문 실행 횟수는 n^2입니다. pass는 아무 일도 하지 않고 문법상 빈 본문을 채웁니다.

```
for perm in permutations(arr,n): pass
```

후보 순열은 n!개입니다. 후보 수를 묻는 답과 각 길이 n인 결과를 생성·저장·출력하는 전체 비용을 구별해야 합니다.

```
for i in range(1<<n): pass
```

0부터 2^n-1까지이므로 본문은 2^n번입니다. 각 마스크의 n개 비트를 검사하는 반복은 이 코드에 아직 없습니다.

복잡도 퀴즈 해설: 후보 수와 만드는 비용

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 n=3이면 A의 본문 9번, B의 후보 6개, C의 본문 8번입니다.

02 부분집합 목록을 실제로 만드는 power_set은 C의 반복 안에서 n개 위치를 보므로 $O(n \cdot 2^n)$입니다.

03 순열을 전부 길이 n의 배열로 복사해 저장하면 $O(n \cdot n!)$가 필요합니다. n!개의 후보라는 사실만 말하고 복사 비용을 숨기지 않습니다.

복잡도 퀴즈 · C 코드

work()는 O(1) 작업이라고 가정합니다. A와 C의 실행 횟수를 비교하세요.

```c
// A: n * n calls
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++) work();

// B: n! permutations, but writing each costs O(n)
// make_permutations(A, n, n, out);  // P18

// C: 2^n masks (n must fit the bit width)
for (unsigned mask = 0; mask < (1u << n); mask++)
  work();
```

코드 읽기

A: $\Theta(n^2)$ · B: 순열 수 $n!$, 전체 기록 $\Theta(n \cdot n!)$ · C: $\Theta(2^n)$

C 복잡도 퀴즈: work의 비용까지 곱하기

줄별 해설 1/1 · 무엇을 하는가 → 왜 필요한가

```
for (int i=0;i<n;i++) for (int j=0;j<n;j++) work();
```

work가 일정 시간이라면 전체 $O(n^2)$입니다. work 자체가 길이 n 배열을 읽는다면 $O(n^3)$으로 달라집니다.

```
make_permutations(A,n,n,out);
```

n!개 순열을 만들고 완성될 때마다 n개 원소를 out에 복사하므로 전체 출력 크기는 $n \cdot n!$입니다.

```
for (unsigned mask=0; mask<(1u<<n); mask++) work();
```

마스크는 2^n개입니다. work가 일정 시간이면 $O(2^n)$, 비트 n개를 검사하면 $O(n \cdot 2^n)$입니다. 시프트 가능한 n 범위를 전제로 합니다.

C 복잡도 퀴즈: work의 비용까지 곱하기

실행 추적 · 결과 검산 · 자주 틀리는 지점

01 반복문 개수만 세면 틀릴 수 있습니다. 각 반복의 범위와 본문에서 실제로 하는 작업을 함께 셉니다.

02 순열 저장은 out의 행 수도 n!개 필요합니다. 시간뿐 아니라 결과를 담을 메모리도 제한에 걸립니다.

03 수업 답안에서는 후보 수 → 후보 한 개 처리 비용 → 전체 시간 순서로 설명합니다.

순열과 조합 퀴즈

Q1	로또 6/45 당첨번호의 조합 수	**조합**	`45C6 = 8,145,060`
Q2	5명 중 금·은·동 메달리스트	**순열**	`5P3 = 60`
Q3	중복 없는 세 자리 숫자 비밀번호	**순열**	`10P3 = 720`
Q4	음식 5개 중 메뉴 2개 선택	**조합**	`5C2 = 10`

다음 장에서 다룰 알고리즘

3장 축소 정복	문제 크기를 줄이며 해결 / 삽입 정렬, 이진 탐색
4장 분할 정복	작은 문제를 나누어 재귀로 해결 / 병합 정렬, 퀵 정렬
7장 백트래킹	가능성 없는 경로를 제외 / N-Queens, 미로 탐색
5·6장 그리디와 DP	탐욕적 선택과 부분 문제 활용 / 배낭, 최소 신장 트리

완전 탐색 연습 순서

01	**반복문**	약수·소수 찾기, 모든 쌍 비교
02	**순열과 조합**	itertools 또는 used·start를 사용하는 C 재귀
03	**비트마스크**	1 << n, mask & (1 << j)로 부분집합 생성
04	**코딩 테스트**	완전 탐색 유형 문제를 직접 구현하고 검증
05	**최적화**	기존 코드에 가지치기를 추가해 계산량 비교

연습: 백준 1018, 2309, 2231, 7568 / 프로그래머스 타겟 넘버, 카펫

2장을 마치며

01 완전 탐색의 개념

02 단일·이중·삼중 반복문

03 후보 생성과 조건 검사, 해 선택

04 Python·C의 순열과 조합 구현

05 조합적 폭발과 시간 복잡도

06 탐색을 최적화하는 방법

NEXT

03

축소 정복

문제의 크기를 줄이며
해결하는 방법

2장 정리

완전 탐색

다음 시간: 축소 정복

2장 코딩 연습

학번 · 이름 · 분반으로 시작

실습 페이지에서 2장을 선택하고
연습문제부터 차례로 풉니다

막히면 단계별 힌트

사용할 문법과 풀이 순서를 보고
직접 코드를 이어 작성합니다

실행과 설명을 함께 연습

예제·추가 검사 후 경계 조건과
시간 복잡도를 설명합니다

실습: sangdon-park.github.io/algorithm-lab/?chapter=2