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

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;
}
```

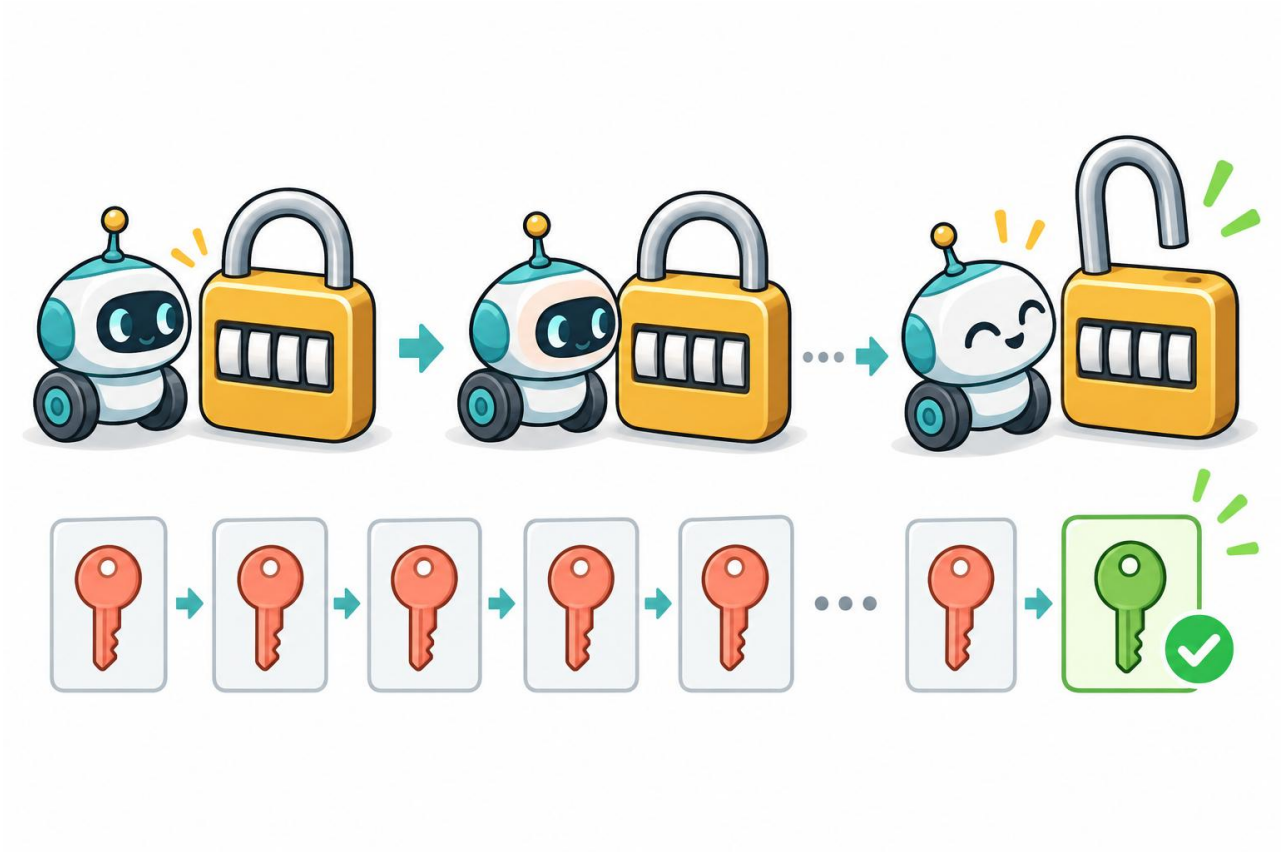
실행 결과: 1 2 5 10 · find_divisors 함수 정의를 이어 붙이면 완전한 프로그램입니다.

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 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;
}
```

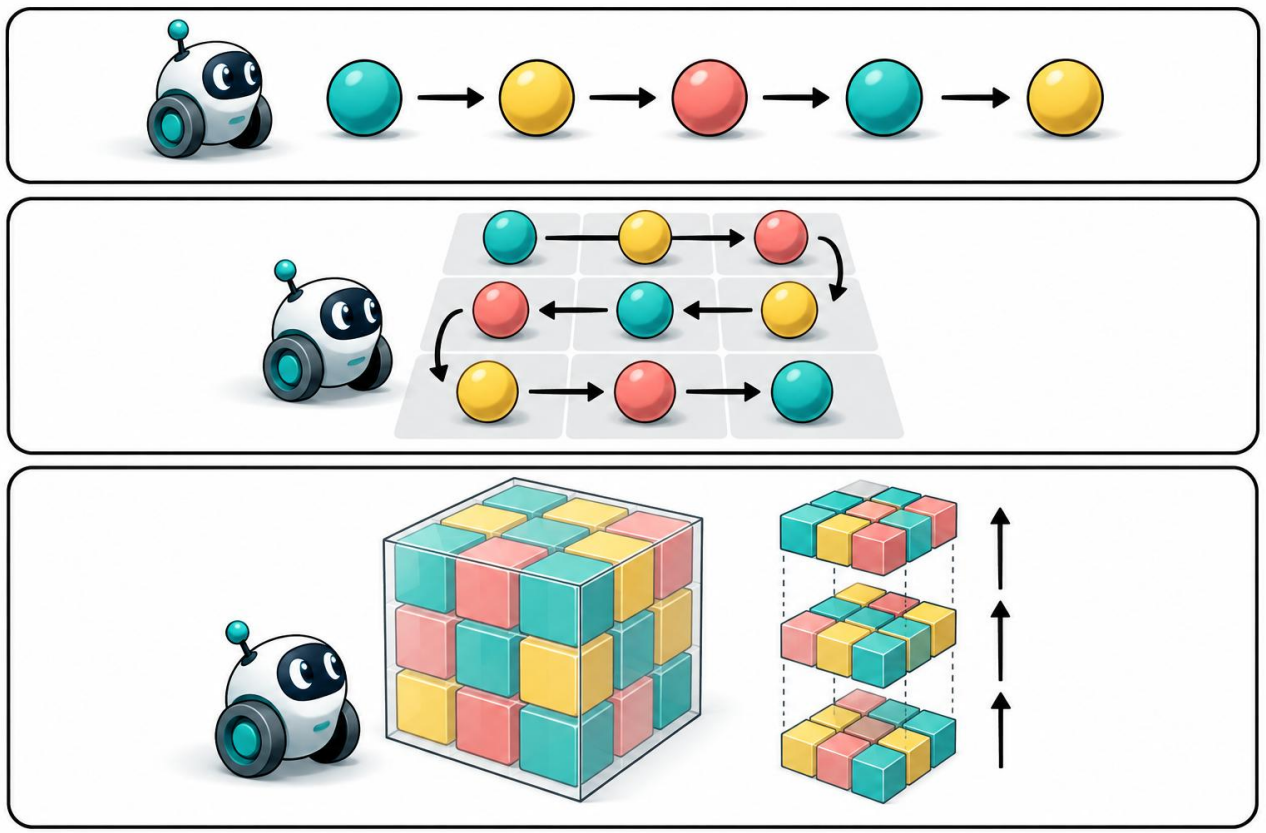
C에서는 char attempt[5]에 네 자리와 끝의 '₩0'을 저장하고 strcmp로 비교합니다. break는 가장 안쪽 반복문만 끝냅니다.

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 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.

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

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.

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

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.

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

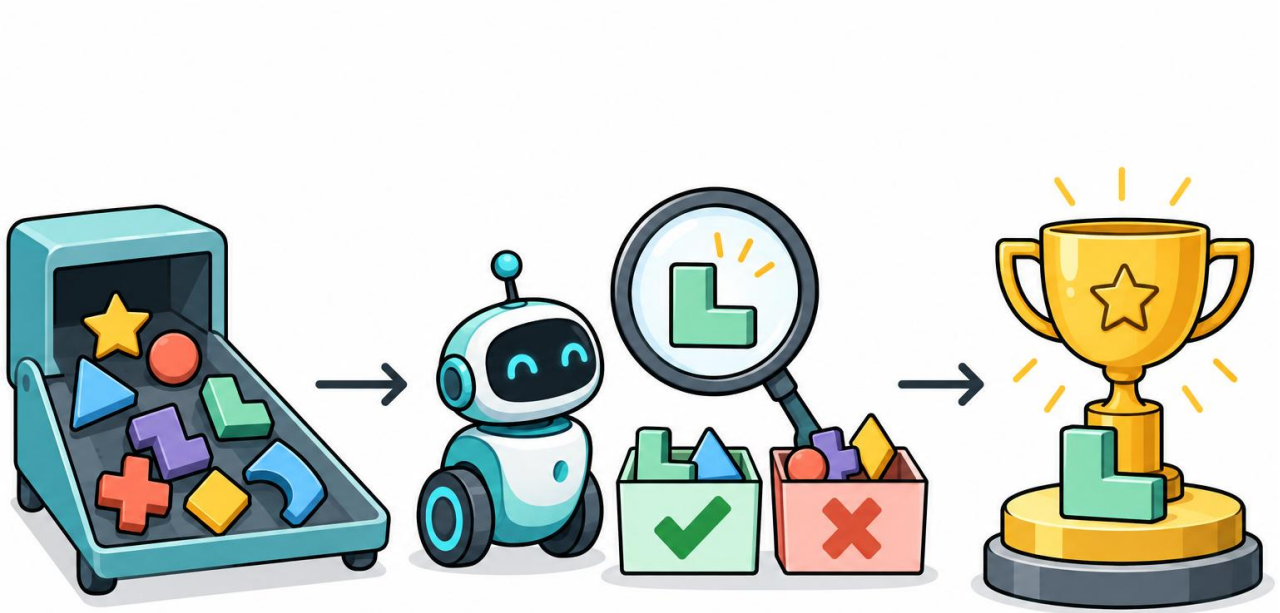
반복문 수와 각 반복문의 범위를 함께 확인합니다.

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.

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.

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 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.

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)!

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.

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.

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)!)

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.

직접 만드는 r개 조합 · C (2/2)

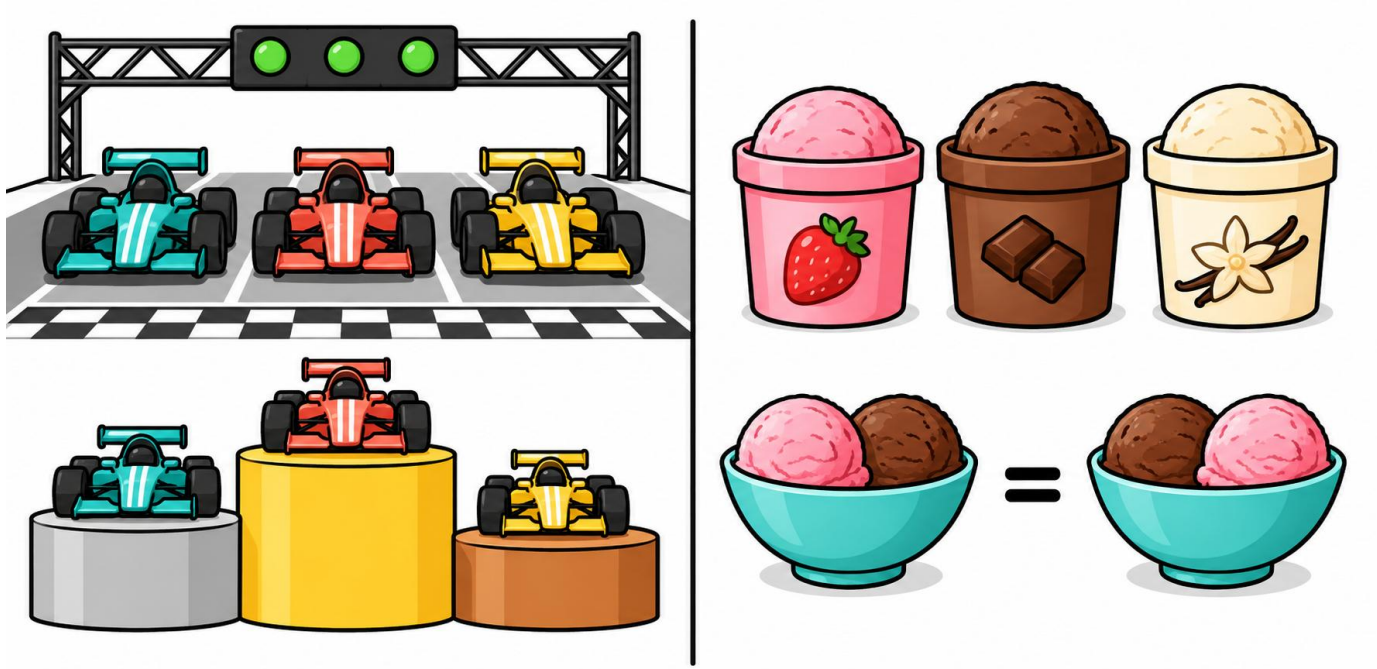
P19 / out[][6] = 선택 결과, return = 행 수 (최대 720)

```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.

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.

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.

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 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

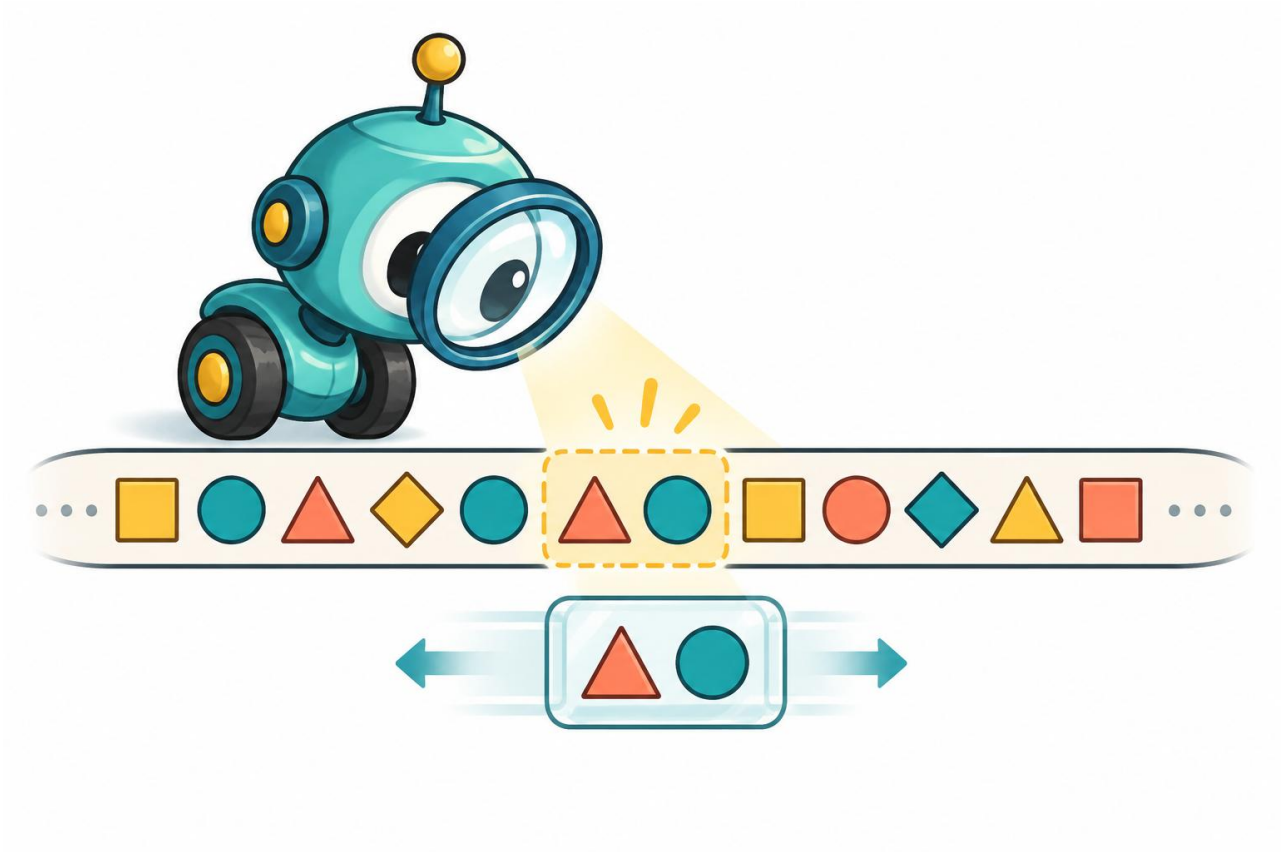
n개 중 2개를 고르면 n*(n-1)/2개의 메뉴가 만들어집니다.

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 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.

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.

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 · 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.

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

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

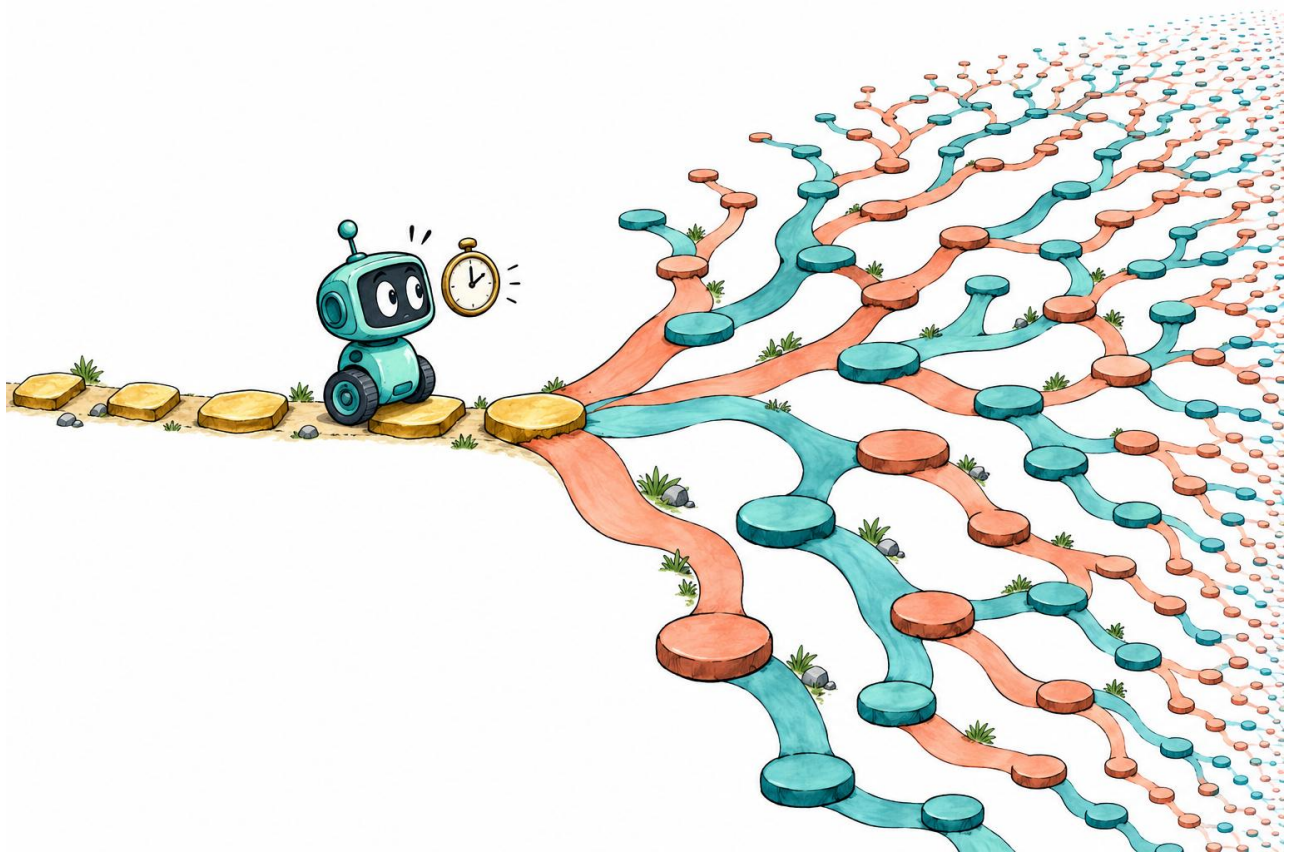
mask & (1u << j)가 참이면 A[j]를 선택합니다. 부분집합 수는 2^n, 모든 원소 검사까지 포함하면 O(n*2^n)입니다.

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.

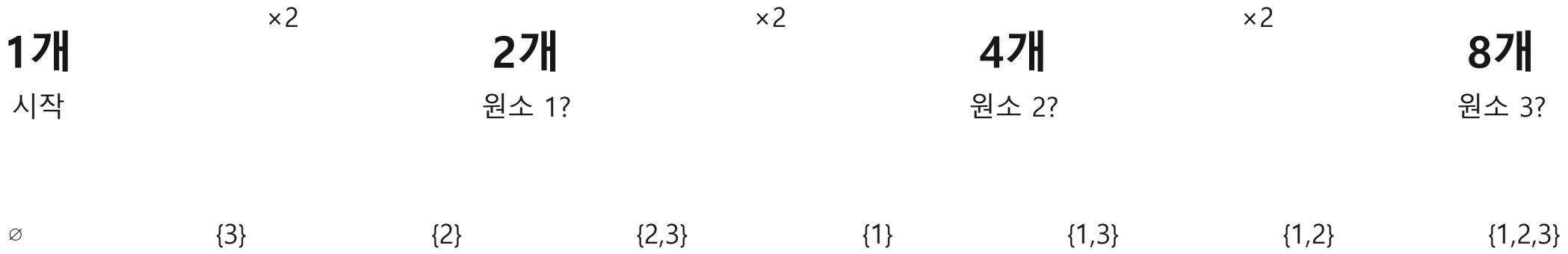
비밀번호 자릿수와 완전 탐색의 한계

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 $O(2^n)$

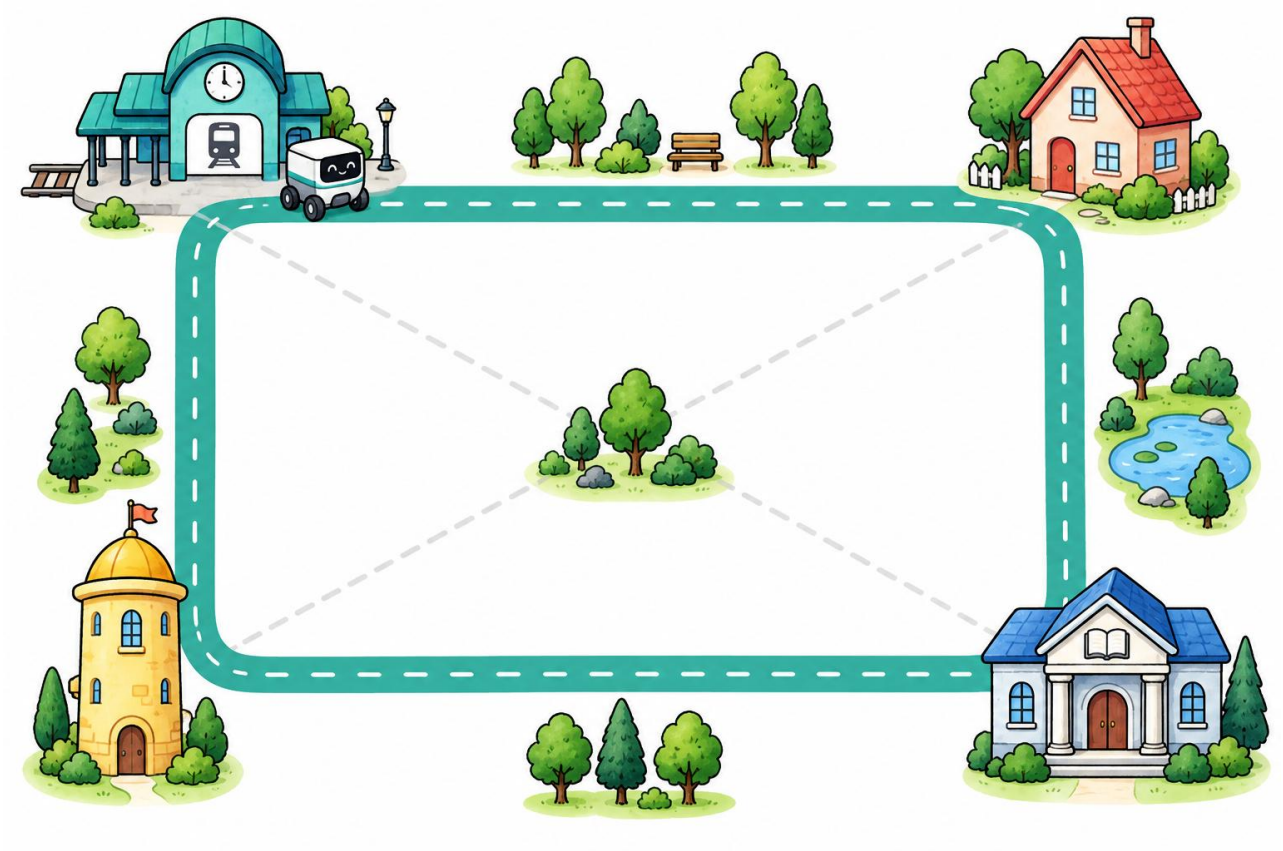
{1, 2, 3} → 각 원소를 포함하거나 포함하지 않거나 → $2^3 = 8$가지 (선택의 연속 = 지수적 폭발)



$2^3 = 8$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 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.

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

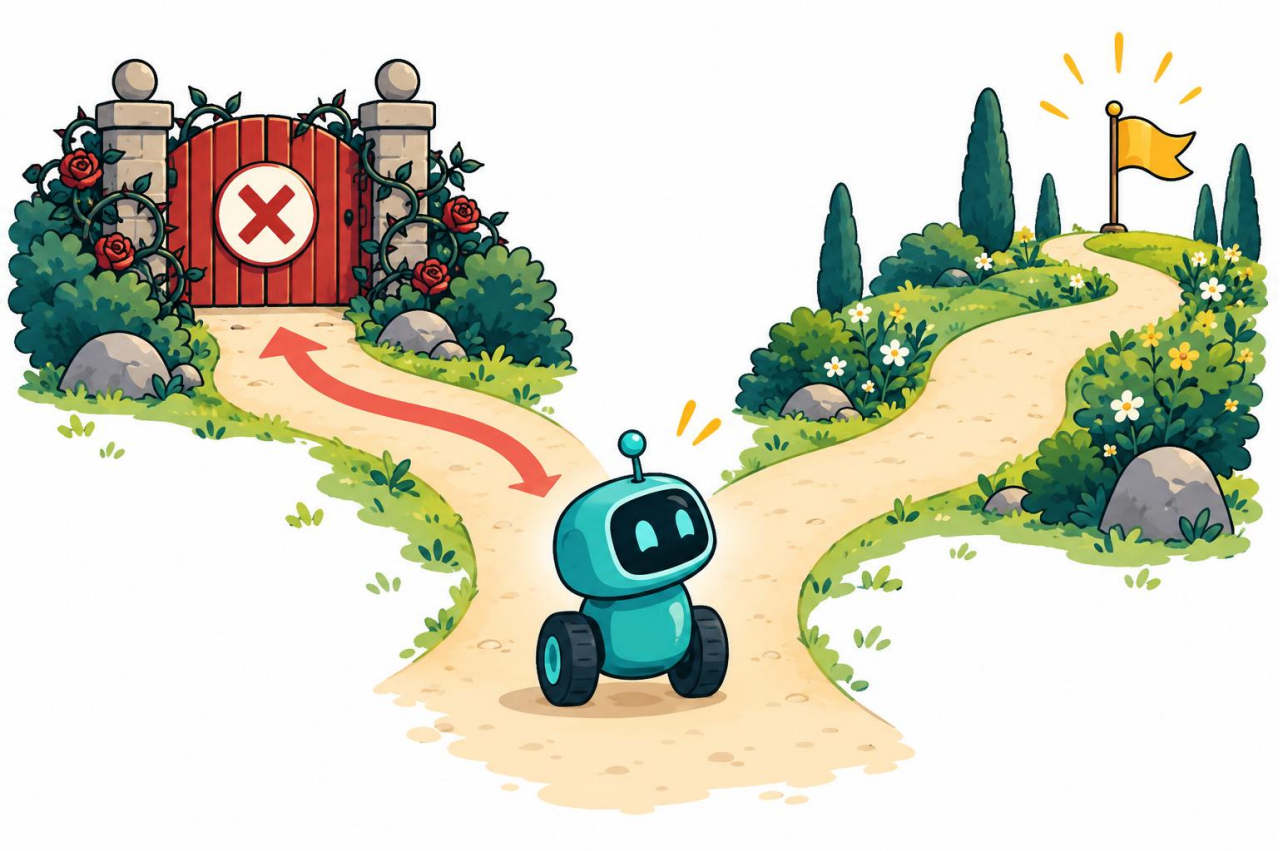
작은 문제의 해를 결합

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.

선형 검색과 이진 탐색

정렬된 전화번호부에서 이름 찾기

선형 검색

처음부터 한 줄씩 확인합니다.
최악에는 모든 항목을 봅니다.

전체 항목 수에 비례

O(n)

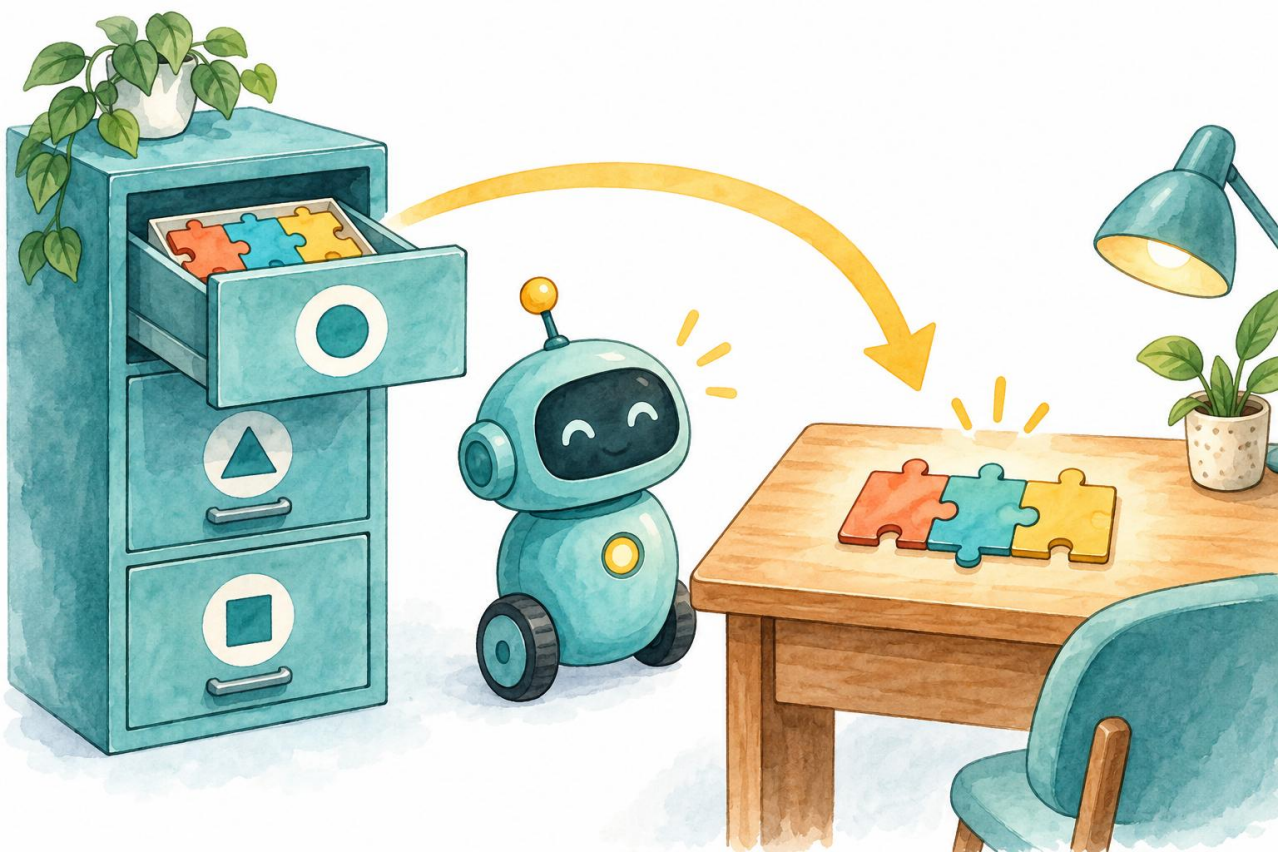
이진 탐색

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.

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.

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)]
```

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.

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.

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.

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 개수 · C

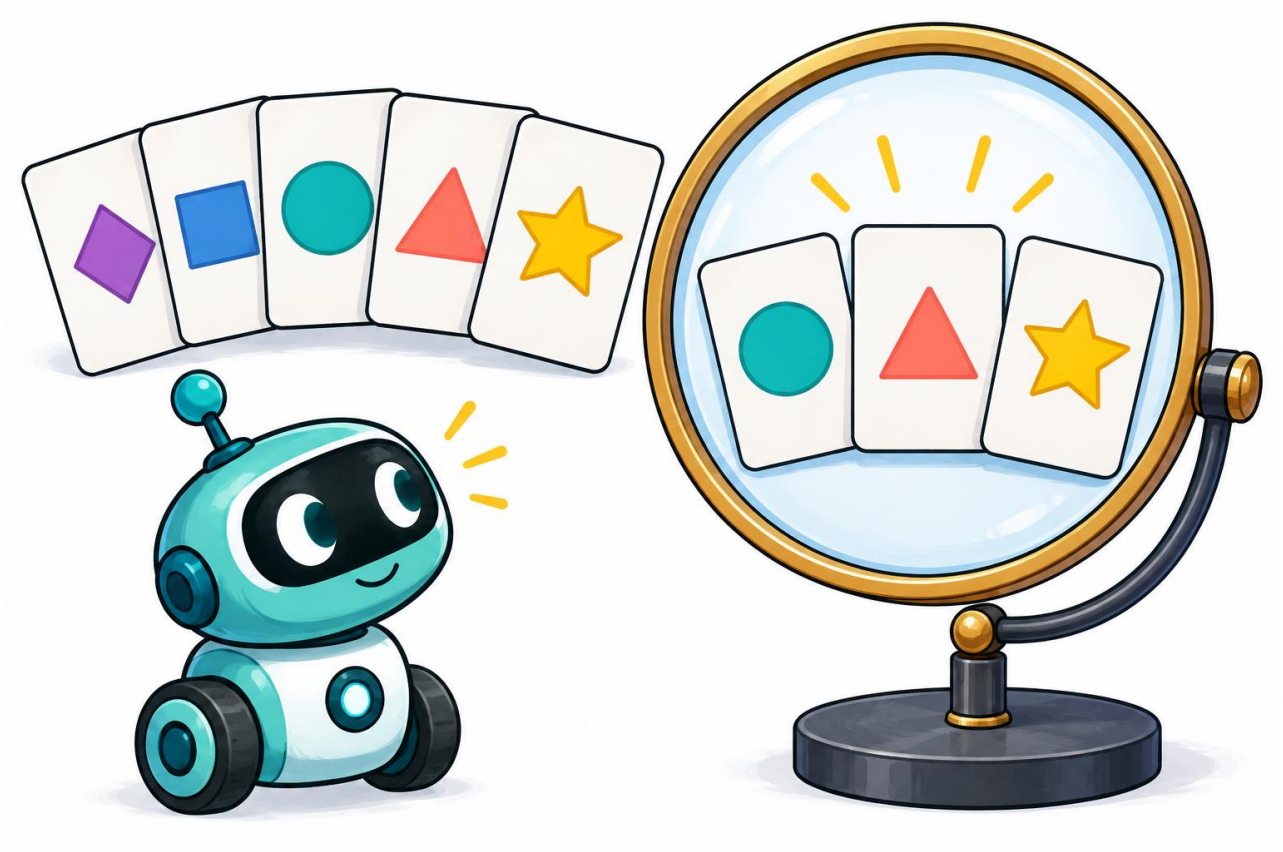
P29 / return = 경우의 수

```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.

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

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.

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)

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.

합이 K인 세 수 나열 · C

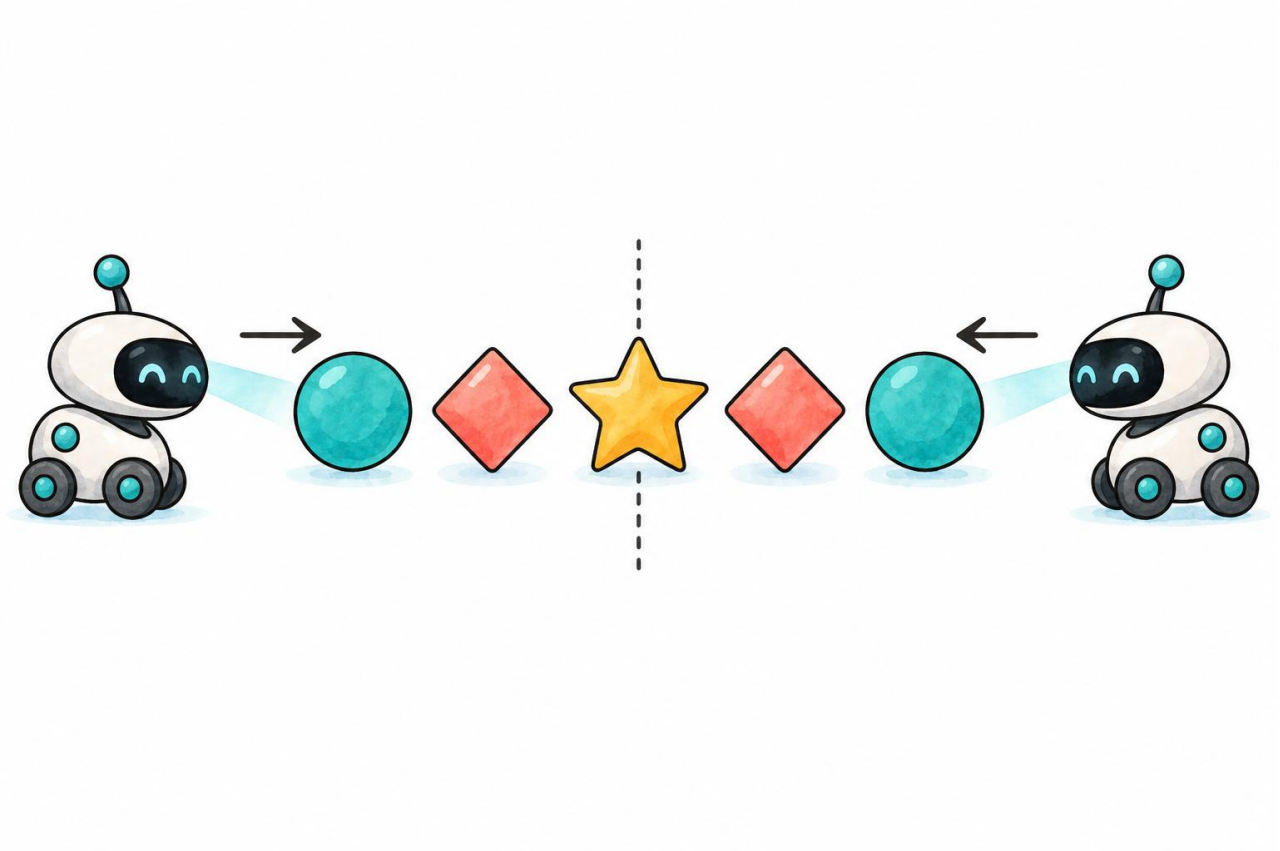
P32 / out[][3] = 세 수, return = 행 수 (최대 1140)

```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.

양끝에서 같은 모양인지 비교하기



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

그림의 원·마름모·별·마름모·원은 대칭입니다. 가운데 원소 하나는 남아도 됩니다.

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

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.

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

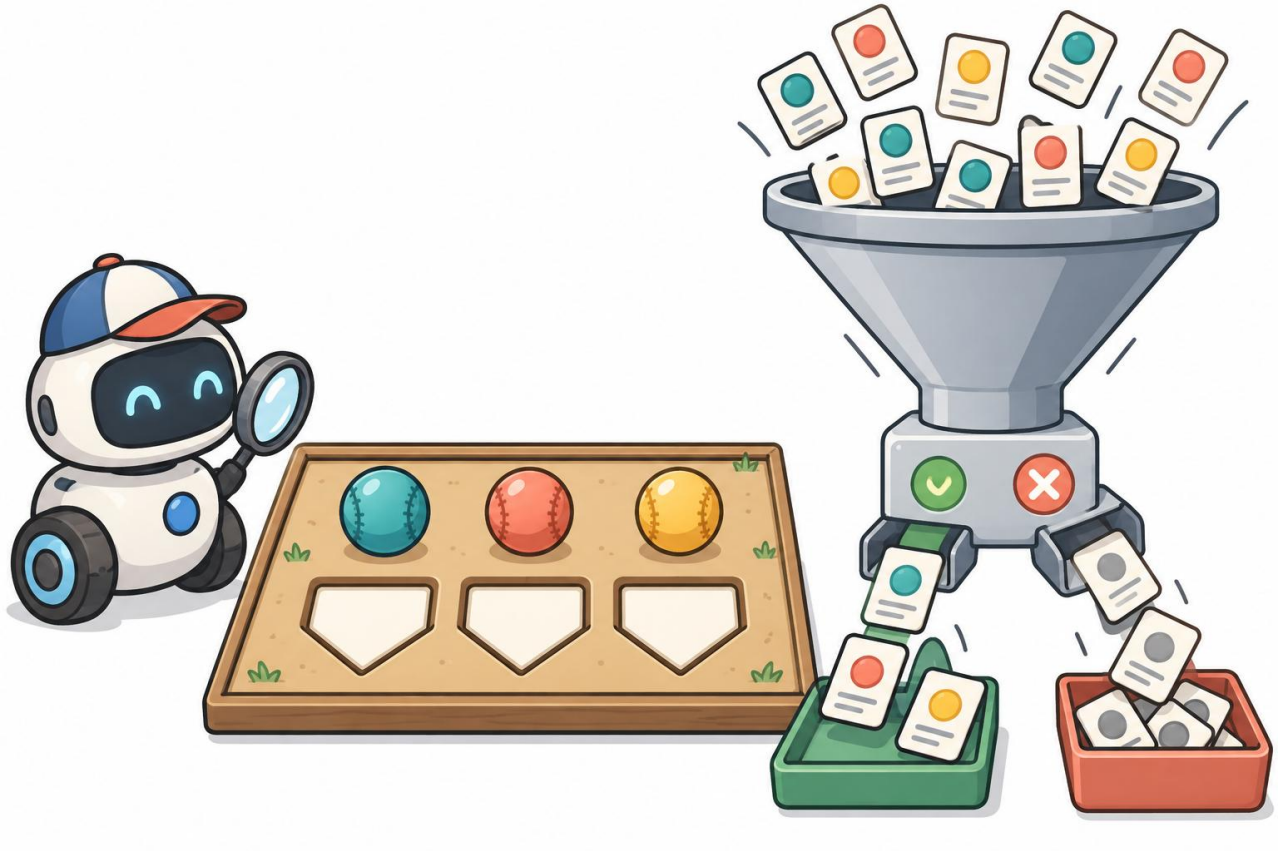
0/1 배낭 완전 탐색 · C

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.

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 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 후보 세기 · C (2/2)

P35 / return = 모든 힌트를 만족하는 후보 수

```c
        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.

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 · 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: $\Theta(2^n)$

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

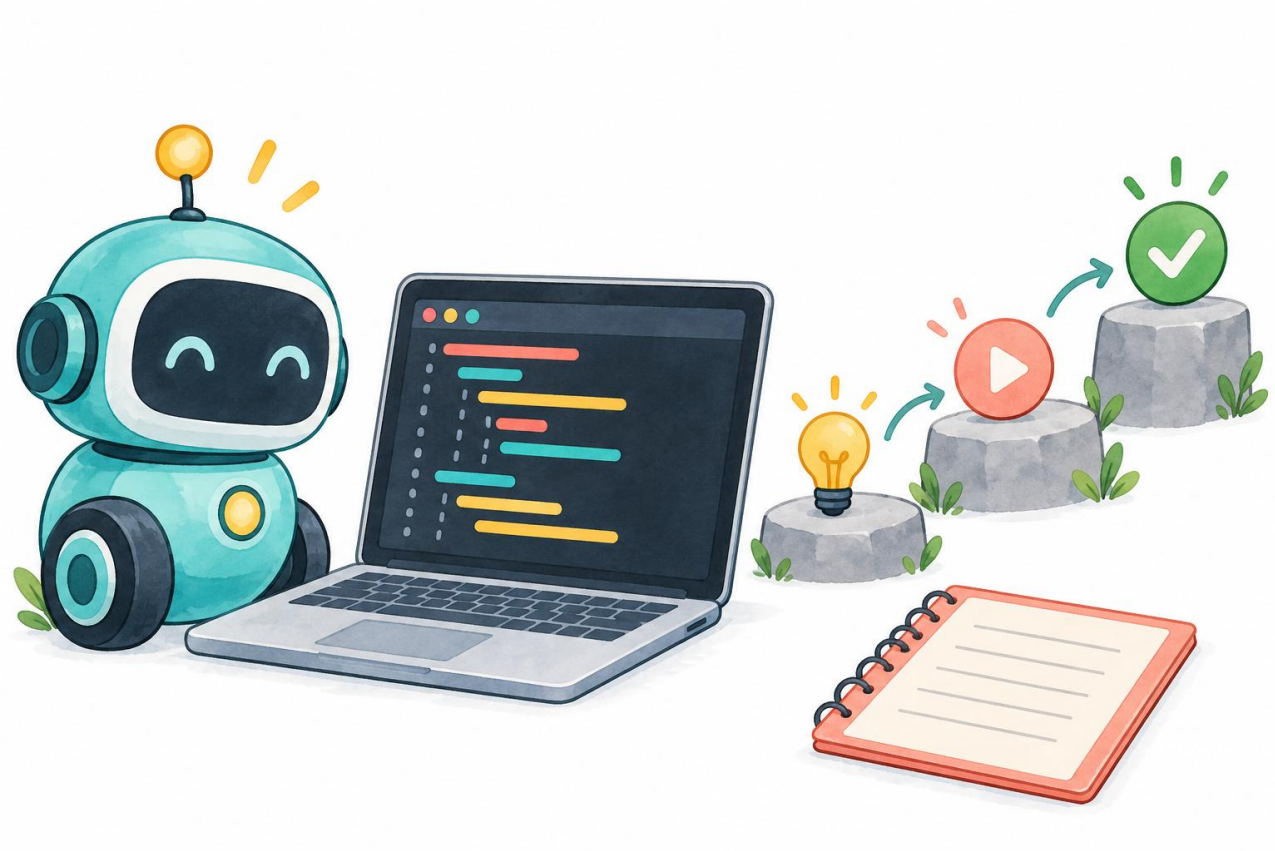
문제의 크기를 줄이며
해결하는 방법

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