그림으로 쉽게 설명하는 파이썬 알고리즘

3장

축소 정복 기법

Decrease and Conquer

정답을 찾기 위해 모든 곳을 다 뒤져야만 할까?

학습 목표

핵심 질문 “정답을 찾기 위해 모든 곳을 다 뒤져야만 할까?”

학습 포인트

축소 정복 기법의 기본 원리
문제를 같은 유형의 더 작은 문제로 바꾸어 반복적으로 해결하는 전략을 이해한다.

완전 탐색의 한계 극복
입력이 커질 때 폭발하는 계산량을 어떻게 줄일 수 있는지 학습한다.

문제 크기를 줄이는 3가지 전략
동일 크기 축소 · 동일 비율 축소 · 가변 크기 축소를 구분해 파악한다.

이번 장의 흐름

정의에서 출발해 세 가지 축소 유형을 구분하고, 재귀·반복 구현을 거쳐 탐색·거듭제곱·정렬·선택 문제로 확장합니다.

3.1

축소 정복이란 무엇인가

3.2

축소의 세 가지 유형

3.3

구현: 재귀와 반복

3.4

탐색 알고리즘

3.5

유클리드와 거듭제곱

3.6

정렬과 선택 문제

3.7

Coding Test

Q & A

거대한 문제를 마주했을 때

2장에서 배운 완전 탐색은 모든 경우를 하나하나 확인합니다. 입력 크기가 조금만 커져도 계산량이 폭발하죠. 이제 필요한 것은 문제의 크기 자체를 줄이는 기술입니다.

입력 크기가 커질수록 완전 탐색의 계산량은 기하급수적으로 늘어난다

2장: 완전 탐색
모든 경우를 하나하나 확인했다

한계: 조합적 폭발
입력이 커지면 계산량이 기하급수적으로 폭발

3장: 축소 정복
쪼개고, 버리고, 핵심만 남긴다

이제 필요한 것은 **문제의 크기 자체를 줄이는 기술**입니다.

SECTION 3.1

축소 정복이란 무엇인가?

3.1

문제의 크기를 단계적으로 줄여가며 해결 범위를 좁히는 전략을 알아봅니다.

§3.1 축소 정복

축소 정복(Decrease and Conquer)의 정의

정의 주어진 문제를 해결하기 위해 문제의 크기를 단계적으로 줄여가며 해결 범위를 좁히는 전략.

핵심 메커니즘

1. 주어진 문제를 약간 더 작은 하위 문제(sub-problem)로 바꾼다.
2. 하위 문제를 해결하고, 그 결과를 기반으로 전체 문제를 해결한다.
3. 직접 계산 가능한 크기가 될 때까지 이 과정을 반복한다.

축소 정복은 문제를 "같은 유형의 더 작은 문제"로 바꾼다는 점이 핵심입니다.

비유: 100장의 접시와 설거지

한 장씩 씻을 때마다 남은 접시는 99장, 98장으로 줄어든다

문제

100장을 한 번에 다 씻는 것은 불가능합니다. 문제 전체를 통째로 붙잡으면 어디서부터 손대야 할지 알 수 없습니다.

축소 + 정복

한 장을 씻고 나머지 99장에 같은 방법을 적용합니다. 문제의 크기를 하나씩 줄여가며 해결하는 방식입니다.

한 번에 하나씩 — **매 단계마다 문제의 크기가 정확히 1씩 줄어듭니다** (동일 크기 축소).

축소 정복 전략의 대표적인 예: 팩토리얼

$$n! = n \times (n-1)! \qquad (\text{단, } 0! = 1)$$

n!을 계산하려면 먼저 (n−1)!을 구한 뒤 그 결과에 n을 곱하면 됩니다. (n−1)!은 n!과 같은 형태이지만 크기가 하나 작아진 하위 문제입니다.

같은 유형의 더 작은 문제로 바꾸어 반복 — 크기가 충분히 작아져 직접 계산할 수 있을 때(0! = 1)까지 이어집니다.

SECTION 3.2

축소의 세 가지 유형

3.2

한 번에 문제의 크기를 얼마나 줄이느냐에 따라 축소 정복은 세 갈래로 나뉩니다.

§3.2 세 가지 유형

축소 정복의 세 가지 주요 유형

축소 정복 알고리즘은 한 단계에서 입력 크기를 얼마나 줄이는가에 따라 다음 세 가지로 구분됩니다.

유형	설명	대표 예제	시간 복잡도 예시
동일 크기 축소	문제의 크기를 매번 일정한 양(예: 1)씩 줄여나가는 방식	팩토리얼 계산, 순차 탐색, 삽입 정렬	O(n)
동일 비율 축소	문제의 크기를 일정한 비율(예: 1/2, 1/3)로 줄여나가는 방식	이진 탐색, 빠른 거듭제곱, 위조 코인(2/3분할)	O(log n)
가변 크기 축소	문제 크기가 상황에 따라 유동적으로 줄어드는 방식	유클리드 알고리즘 (GCD)	O(log min(a, b)) 또는 다양

동일 크기 · 동일 비율 · 가변 크기 — 감소 폭에 따른 세 가지 축소 방식

감소 폭이 클수록 단계 수가 줄어들지만, 한 단계의 비용과 분석 난이도는 올라갑니다.

§3.2 세 가지 유형

① 동일 크기 축소 (Decrease by a Constant)

문제를 매 단계마다 입력 크기를 1만 줄여가며 해결합니다. 한 요소를 처리하고 나머지를 다시 푸는 구조로, 단계가 많지만 구현이 단순하고 이해하기 쉽습니다.

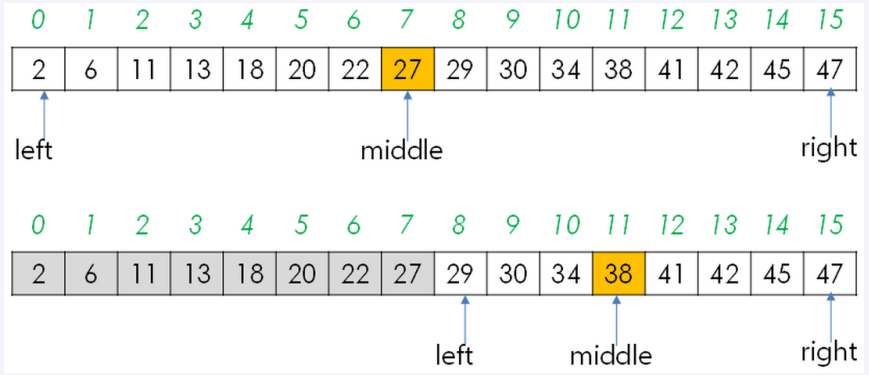
n → n − 1

단계 수 ≈ n
선형 시간 O(n)

대표 예 팩토리얼 계산 · 순차 탐색 · 삽입 정렬
단계가 많지만 각 단계가 값싸고, 재귀·반복 어느 쪽으로도 자연스럽게 구현됩니다.

② 동일 비율 축소 (Decrease by a Constant Factor)

매 단계마다 문제의 크기를 항상 동일한 비율(예: 1/2, 1/3, 1/10)로 줄여 나갑니다. 절반씩 버리므로 단계 수가 급격히 줄어듭니다.



매 단계 크기가 절반으로 감소한다

n → n / 2

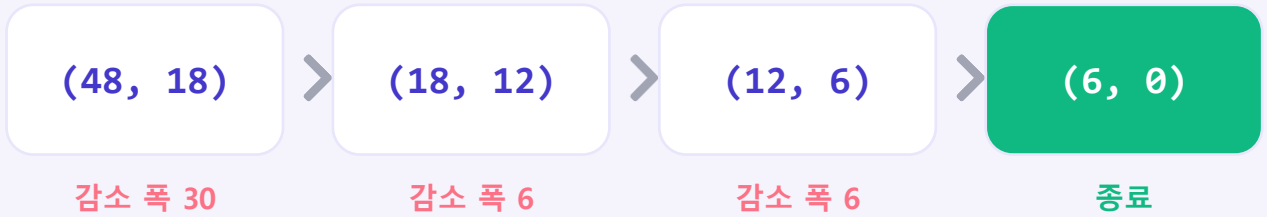
단계 수 ≈ $\log_2 n$
로그 시간 O(log n)

대표 예 이진 탐색 · 위조 동전 찾기 · 축소 정복 거듭제곱
n = 1,000,000이어도 단계 수는 20번 남짓입니다.

③ 가변 크기 축소 (Variable-Size Decrease)

각 단계에서 입력 크기를 일정하지 않게 줄여 나갑니다. 상황에 따라 감소 폭이 달라져 유연하지만, 분석은 다소 복잡합니다.

유클리드 알고리즘: gcd(48, 18)



(a, b) → (b, a mod b) — 줄어드는 폭이 매번 다릅니다.

n → ?

감소 폭이 입력에 의존
분석이 까다롭다

대표 예 유클리드 알고리즘 · 보간 탐색 · 퀵 셀렉트

최댓값을 찾을 때 일부 요소를 한 번에 제거하는 등의 변형된 접근도 여기에 속합니다.

축소 정복 알고리즘의 구현 방식 비교

같은 축소 전략이라도 재귀로 내려갈지 반복으로 쌓아 올릴지에 따라 코드의 모습이 달라집니다.

표 3.2 축소 정복 알고리즘의 구현 방식 비교

구분	하향식(Top-down)	상향식(Bottom-up)
방식	문제를 작게 나누어 재귀적으로 해결	가장 작은 문제부터 반복적으로 해결
사용 도구	재귀 함수	반복문
접근 방향	위에서 아래로 문제를 축소하며 접근	아래에서 위로 문제를 확장하며 접근
예시 알고리즘	팩토리얼(재귀), 피보나치(재귀), 이진 탐색 등	팩토리얼(반복), 피보나치(반복), 동적 계획법 등
장점	코드가 간결하고 문제 구조가 명확하게 드러남	성능이 좋고 메모리 사용량이 적음
단점	재귀 호출로 인해 오버헤드 발생 가능	구현이 다소 복잡할 수 있음 (특히 상태 저장 시)

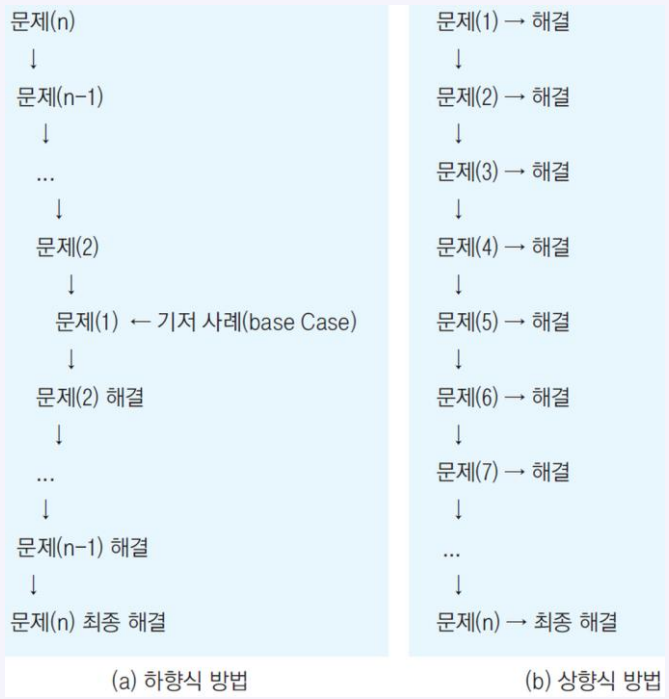
재귀 구현과 반복 구현의 구조 비교

어느 쪽을 택하든 문제를 줄여 나가는 논리는 같습니다 — 전개 방향만 다를 뿐입니다.

§3.2 세 가지 유형

하향식 방법과 상향식 방법의 비교

큰 문제에서 시작해 쪼개 내려가느냐, 작은 문제부터 쌓아 올리느냐의 차이입니다.



(a) 하향식 방법 (b) 상향식 방법

하향식 · 재귀 (Top-down)

큰 문제를 더 작은 하나의 부분 문제로 축소하고, 그것을 재귀적으로 해결한 뒤 결과를 결합·반환한다.

문제를 쪼개는 사고방식을 익히기에 좋다.

표현이 자연스럽다

상향식 · 반복 (Bottom-up)

가장 작은 문제부터 먼저 풀고, 그 결과를 바탕으로 점점 큰 문제를 해결해 나간다.

실행 성능과 안정성 면에서 유리하다.

실행이 효율적이다

분할 정복(Divide and Conquer)과의 비교

두 방법 모두 큰 문제를 더 작은 문제로 나누지만, 나누는 방식과 해결 방식이 다릅니다. 축소 정복을 분할 정복의 특수한 형태로 보기도 합니다.

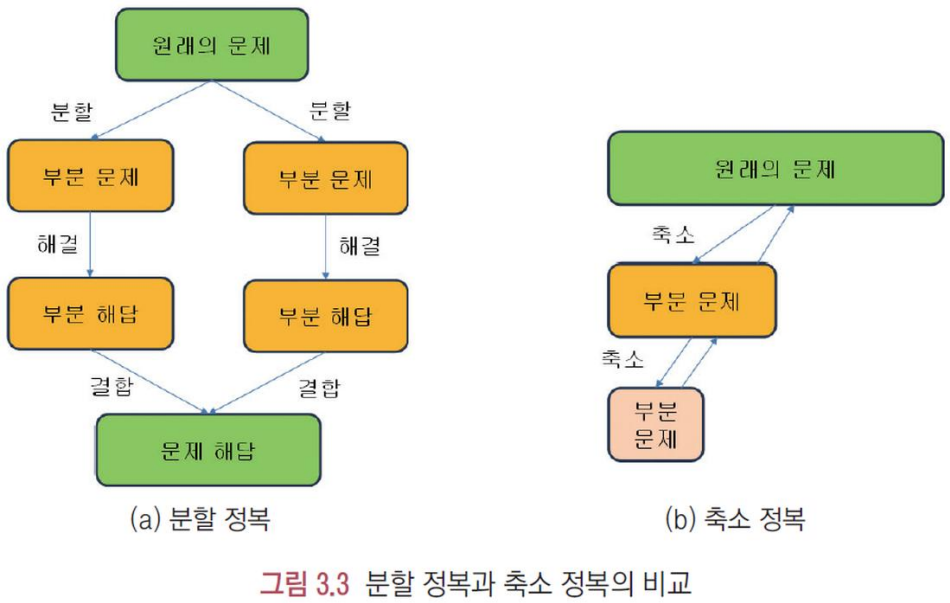


그림 3.3 분할 정복과 축소 정복의 비교

축소 정복은 한 갈래, 분할 정복은 모든 갈래로 내려간다

구분	축소 정복	분할 정복
나누는 방식	하나의 부분 문제로 축소	여러 개(보통 2개)로 분할
푸는 범위	부분 문제 중 하나만 해결	모든 부분 문제를 모두 해결
결합 단계	결합이 없거나 매우 단순	부분 해를 병합하는 단계 필요
재귀 호출 수	단계당 1회	단계당 2회 이상
대표 알고리즘	이진 탐색, 퀵 셀렉트	병합 정렬, 퀵 정렬
점화식 예	T(n) = T(n/2) + 1	T(n) = 2T(n/2) + n

SECTION 3.3

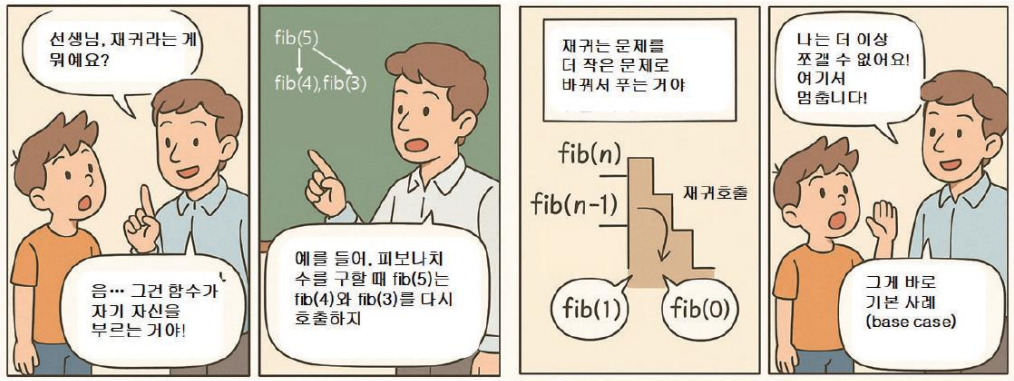
구현: 재귀와 반복

3.3

축소 정복은 하향식 재귀와 상향식 반복, 두 방향 모두로 구현할 수 있습니다.

재귀 (Recursion)

하향식 축소 정복은 문제를 더 작은 하나의 부분 문제로 축소하고, 그 결과를 이용해 원래 문제를 해결합니다.



기본 사례 (base case)

더 이상 쪼갤 수 없는 가장 작은 입력에 대한 즉각적인 해. 여기서 재귀가 멈춘다.

재귀 사례 (recursive case)

문제를 더 작은 동일 문제로 축소해 자기 자신을 호출하고, 그 결과로 해를 구성한다.

재귀 함수의 기본 골격

모든 재귀 함수는 "멈추는 조건"과 "줄여서 다시 부르는 조건" 두 가지로 이루어집니다.

recursion_template.pseudo

```
function Rec(n):
    // 기본 사례: 가장 작은 입력값에 대한 즉각적인 해
    if n이 충분히 작다면:
        return n에 대한 직접적인 해답

    // 재귀 사례: 문제를 더 작은 하위 문제로 축소
    smaller_problem = n을 더 작은 n'으로 변환
    sub_result = Rec(smaller_problem)

    // 하위 문제의 결과를 이용하여 현재 문제 해결
    result = sub_result를 이용하여 해답을 구성한다
    return result
```

기본 사례를 빠뜨리면 무한 재귀 — 호출 스택이 넘쳐 RecursionError가 발생합니다.

팩토리얼 계산 (재귀)

재귀는 본질적으로 순환적인 문제나 그러한 자료구조를 다루는 프로그램에 잘 맞습니다.

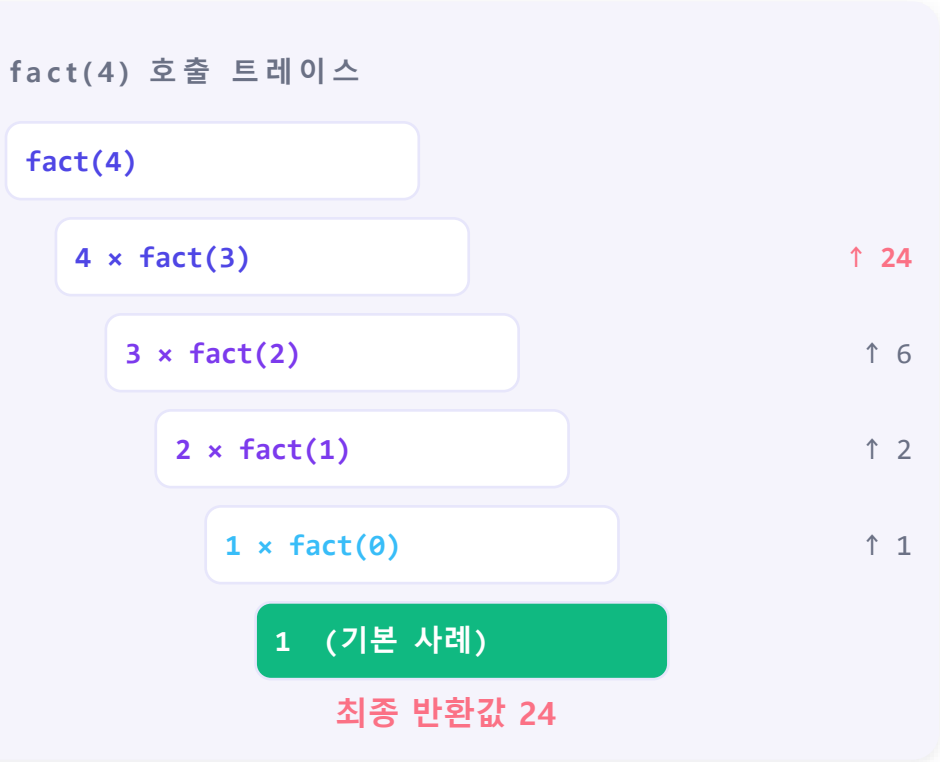
factorial.py

```python
def factorial(n):   # n은 0 이상의 정수
    if n == 0:      # 기본 사례
        return 1
    return n * factorial(n - 1)
```

동작 원리

n = 0에 도달할 때까지 호출을 쌓아 내려가고(하향), 기본 사례에서 값이 확정되면 곱셈을 수행하며 되돌아 올라옵니다(상향).

호출 깊이가 n이므로 재귀 스택도 n만큼 쌓입니다.



재귀 알고리즘의 복잡도 (팩토리얼)

n = 0이면 연산 없이 1을 반환하고 종료합니다. 그 외에는 곱셈 1회 + 재귀 호출 1회가 수행되므로 다음 점화식을 세울 수 있습니다.

$$T(n) = T(n - 1) + 1, \quad T(0) = 0$$

반복 대입으로 전개하기

$$\begin{aligned} T(n) &= T(n-1) + 1 \\ &= T(n-2) + 1 + 1 \\ &= T(n-3) + 1 + 1 + 1 \\ &\ \ \vdots \\ &= T(0) + n \end{aligned}$$

$T(0) = 0$ 이므로

$T(n) = O(n)$

입력 크기 n에 비례하는 선형 시간.
한 번에 1씩 줄이는 동일 크기 축소의
전형적인 결과입니다.

팩토리얼 계산 (반복)

작은 문제를 먼저 해결하고 그것을 바탕으로 큰 문제를 해결합니다. 즉 1! → 2! → … → n! 순으로 곱해 나가는 상향식 방법입니다.

factorial_iter.py

```python
def factorial_iter(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

O(n)

재귀와 같은 선형 시간이지만
스택을 쓰지 않아 더 안정적

재귀는 사고 표현에, 반복은 실행 성능에 유리 — 축소 정복은 둘 다로 쉽게 구현할 수 있습니다.

피보나치 수열 계산 (재귀)

피보나치 수열은 단순한 수학적 흥미를 넘어 자연 현상·컴퓨터 알고리즘·예술적 조화에서 중요한 역할을 하며, 황금비 φ와 밀접한 관련이 있습니다.

fib.py

```python
def fib(n: int):
    if n <= 1:      # 기본 조건: F(0)=0, F(1)=1
        return n
    return fib(n-1) + fib(n-2)   # 재귀 호출
```

$$F(n) = F(n-1) + F(n-2)$$

$$\phi \approx 1.618$$

황금비 (golden ratio)

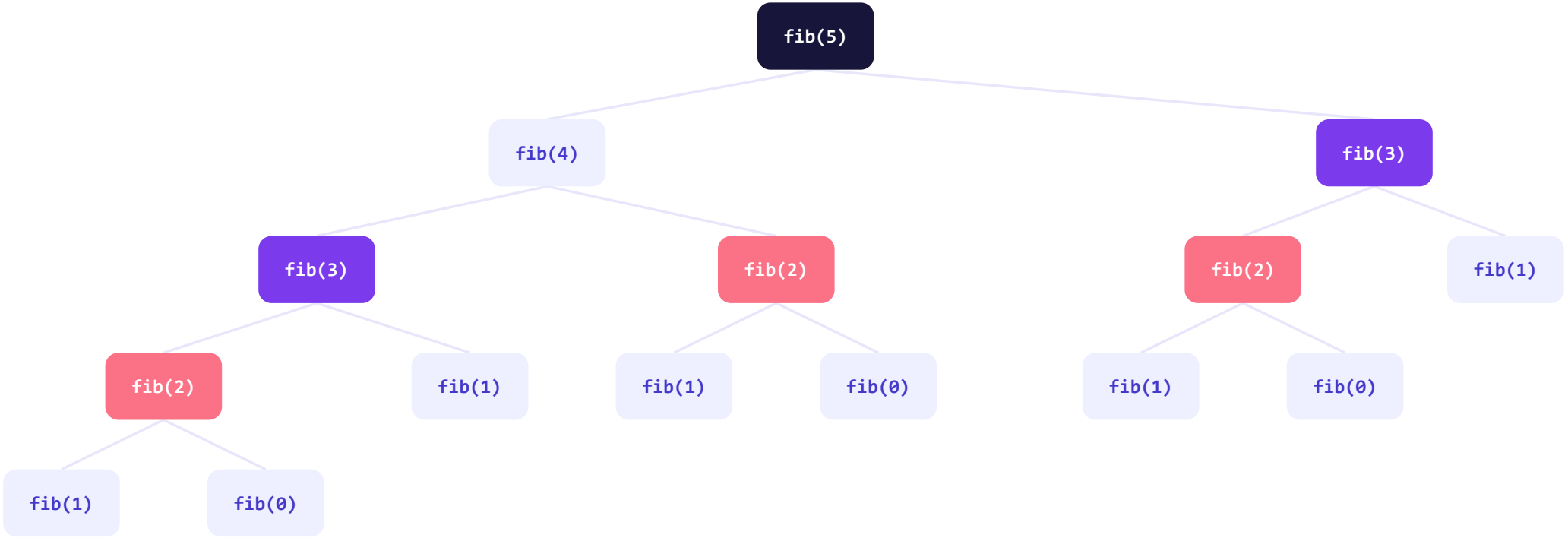
수열의 앞부분

F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	F(8)	F(9)
0	1	1	2	3	5	8	13	21	34

기본 조건

왜 비효율적인가: 중복 계산되는 부분 문제

작성하기 쉽고 직관적이지만, 호출 트리를 따라가 보면 같은 값이 여러 번 반복 계산되는 비효율이 명확히 드러납니다. fib(2)는 무려 세 번이나 계산됩니다.



fib(2) — 3회 중복 계산

fib(3) — 2회 중복 계산

동일한 부분 문제를 매번 처음부터 다시 계산합니다.

재귀 피보나치의 복잡도: 지수 시간

각 호출이 두 번의 재귀 호출을 하므로 다음 점화식을 세울 수 있습니다. (O(1)은 덧셈 비용)

`T(n) = T(n-1) + T(n-2) + O(1)`	점화식 세우기
`T(n) > T(n-1) + T(n-2)`	상수항 버리기
`T(n) > 2 × T(n-2)`	T(n−1) ≥ T(n−2) 이용
`T(n) > 2·2·…·2 × T(0)`	n/2 번 반복 전개
`T(n) > 2^(n/2)`	T(0) = 1

Ω(2^(n/2))

시간 복잡도의 하한이
지수 시간이 됩니다.

n = 50만 되어도
사실상 계산 불가능합니다.

해법 이미 계산한 값을 저장해 두었다가 재사용하면 중복 계산을 피할 수 있습니다.

피보나치 수열 계산 (반복)

값을 몇 개의 변수에 저장해 두고 재사용하면 중복 계산이 사라집니다. 각 항을 한 번씩만 계산하므로 시간 복잡도는 O(n)입니다.

fib2.py

```python
def fib2(n: int):
    a, b = 0, 1          # a=f(0), b=f(1)
    for _ in range(n):
        a, b = b, a + b
    return a
```

컴퓨터 성능이 아무리 발전해도
알고리즘의 효율성은 여전히 핵심적인 문제
임을 명확히 보여줍니다.

표 3.3 피보나치 수행 시간 비교

n	재귀적인 알고리즘	반복적인 알고리즘
40	105μs	4ns
60	107ms	6ns
80	1.83분	8ns
100	1.30일	10ns
120	3.65e+00년	12ns
160	3.83e+06년	16ns
200	4.02e+12년	20ns

피보나치 수행 시간 비교 — 재귀 vs 반복

그렇다면 항상 반복이 재귀보다 좋을까?

재귀의 가장 큰 장점은 문제를 작은 하위 문제로 나누어 푸는 과정을 자연스럽게 표현할 수 있다는 것입니다. 병합 정렬·퀵 정렬처럼 문제를 반으로 쪼개고 각 부분을 다시 정렬한 뒤 합치는 알고리즘이 대표적입니다.

merge_sort.py

```python
def merge_sort(A):
    if len(A) <= 1:              # 길이 1 이하 → 이미 정렬됨
        return A
    mid = len(A) // 2            # 반으로 나눌 기준 인덱스
    left = merge_sort(A[:mid])   # 왼쪽 재귀 정렬
    right = merge_sort(A[mid:])  # 오른쪽 재귀 정렬
    return merge(left, right)    # 정렬된 두 리스트 병합
```

표현력
분할·병합 구조를 코드가 그대로 보여준다

성능
반복은 호출 오버헤드·스택이 없다

선택 기준
구조가 재귀적이면 재귀, 단순 누적이면 반복

정답은 "상황에 따라 다르다" — 문제의 구조가 재귀적이면 재귀 구현이 훨씬 읽기 쉽고 짧습니다.

SECTION 3.4

탐색 알고리즘

3.4

순차 탐색·이진 탐색·보간 탐색은 각각 세 가지 축소 유형을 그대로 보여줍니다.

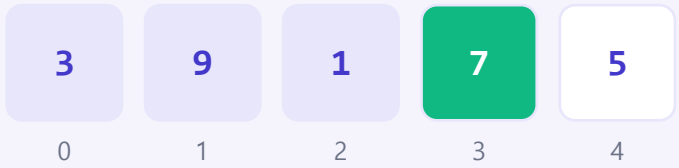
순차 탐색 (동일 크기 축소)

배열의 처음부터 끝까지 하나씩 값을 비교합니다. 한 번에 하나의 요소를 확인하므로 매 단계마다 확인할 요소가 1씩 줄어드는, 전형적인 동일 크기 축소입니다.

linear_search.py

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i     # 찾은 경우 인덱스 반환
    return -1            # 못 찾은 경우 -1 반환
```

target = 7 을 찾는 과정



4번의 비교 끝에 인덱스 3에서 발견
최악의 경우 n번 모두 비교해야 합니다.

최선 O(1)
첫 번째 원소가 정답

평균 O(n)
대략 n/2번 비교

최악 O(n)
마지막에 있거나 없음

이진 탐색 (동일 비율 축소)

정렬된 배열에서 중간값과 비교하여 탐색 범위를 절반으로 줄입니다. 문제의 크기가 n → n/2 → n/4 → … 로 축소됩니다.

binary_search.py

```python
def binary_search(arr, target, left, right):
    if left > right:
        return -1                # 찾지 못함
    mid = (left + right) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search(arr, target, mid+1, right)
    else:
        return binary_search(arr, target, left, mid-1)
```

전제 조건 배열이 정렬되어 있어야 합니다. 한 번의 비교로 후보의 절반을 통째로 버립니다.

이진 탐색의 시간 복잡도 분석

한 단계에서 비교 1회를 수행하고 문제 크기를 절반으로 줄이므로, 점화식을 전개하면 단계 수가 $\log_2 n$이 됩니다.

이진 탐색의 시간 복잡도 분석을 위하여 다음과 같이 점화식을 세울 수 있다.

$$T(n) = T(n/2) + 1 \qquad \text{// 단, } T(1) \approx 1$$

위의 식은 문제를 크기 n에서 시작하여 매번 절반(n/2) 크기로 줄이면서, 각 단계마다 1회의 일정한 연산이 수행된다는 의미이다. 이 점화식을 전개하면 다음과 같다.

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= [T(n/4) + 1] + 1 \\ &= [[T(n/8) + 1] + 1] + 1 \\ &= \dots \\ &= T(n/2^k) + k \end{aligned}$$

재귀가 멈추는 지점은 보통 T(1)이다. 따라서 이때, $n/2^k = 1$이 되는 순간 재귀가 종료된다. 이걸 방정식으로 풀면

$$\begin{aligned} & n/2^k = 1 \\ \rightarrow\ & 2^k = n \\ \rightarrow\ & k = \log_2 n \end{aligned}$$

최종 시간 복잡도 = O(log n)

$$T(n) = T(n/2) + 1 \quad (n > 1)$$
$$T(1) = 0$$

O(log n)

n = 1,000	약 10회
n = 1,000,000	약 20회
n = 10억	약 30회

보간 탐색 (가변 크기 축소)

찾는 값 x가 배열의 "어느 위치쯤"에 있을지 추정한 뒤 그 지점을 직접 비교합니다. 전화번호부에서 '홍'을 찾을 때 뒤쪽부터 펼치는 것과 같습니다.

$$pos = low + (x - S[low]) \times (high - low) / (S[high] - S[low])$$

이진 탐색과 무엇이 다른가

이진 탐색은 값과 무관하게 항상 한가운데를 봅니다.
보간 탐색은 값의 크기를 이용해 예상 위치로 곧장 점프하므로, 줄어드는 폭이 매 단계 달라집니다.

시간 복잡도

균등 분포일 때	$O(\log \log n)$
편향된 분포일 때	$O(n)$

데이터가 고르게 분포할수록 강력하지만, 치우친 데이터에서는 이진 탐색보다 느려질 수 있습니다.

보간 탐색 — 전체 소스

interpolation_search.py (1/2)

```python
def interpolation_search(arr, x):
    low = 0
    high = len(arr) - 1
    while low <= high and arr[low] <= x <= arr[high]:
        # 분모가 0이 되는 경우 방지
        if arr[high] == arr[low]:
            if arr[low] == x:
                return low
            else:
                break
        # 보간 공식으로 예상 위치 pos 계산
        pos = low + int((x - arr[low]) * (high - low)
                        / (arr[high] - arr[low]))
```

interpolation_search.py (2/2)

```python
        if pos < 0 or pos >= len(arr):
            break              # 안전 장치
        if arr[pos] == x:
            return pos
        elif arr[pos] < x:
            low = pos + 1      # 오른쪽 구간으로
        else:
            high = pos - 1     # 왼쪽 구간으로
    return -1                  # 찾지 못함

S = [10, 20, 30, 40, 50, 60, 70, 80, 90]
idx = interpolation_search(S, 70)
print(f"70는 인덱스 {idx}에 있습니다.")
```

실행 결과 `70는 인덱스 6에 있습니다.`

§3.4 탐색 알고리즘

세 가지 탐색 알고리즘 정리

같은 "탐색"이지만 문제 크기를 줄이는 방식이 다르고, 그 차이가 복잡도를 결정합니다.

알고리즘	축소 유형	전제 조건	평균 복잡도	최악 복잡도
순차 탐색	동일 크기 (n → n−1)	없음	O(n)	O(n)
이진 탐색	동일 비율 (n → n/2)	정렬된 배열	O(log n)	O(log n)
보간 탐색	가변 크기 (n → ?)	정렬 + 균등 분포	O(log log n)	O(n)

동일 크기

단순하지만 느리다. 정렬 없이도 쓸 수 있다.

동일 비율

정렬 비용을 감수할 만큼 강력하다.

가변 크기

데이터 분포를 알면 더 빠를 수 있다.

SECTION 3.5

유클리드와 거듭제곱

3.5

가변 크기 축소의 대표 알고리즘과, 절반씩 줄이는 거듭제곱 계산을 살펴봅니다.

유클리드 알고리즘 (가변 크기 축소)

두 정수 a, b (a ≥ b > 0)에 대해 각 단계마다 (a, b) → (b, a mod b)로 값을 갱신하고, b가 0이 될 때까지 반복합니다. 매 단계에서 a mod b는 b보다 작으므로 최소한 하나의 값이 계속 줄어듭니다.

gcd.py

```python
def gcd(a, b):
    if b == 0:
        print(f"b가 0이므로 종료. 최대공약수는 {a}")
        return a
    else:
        return gcd(b, a % b)
```

gcd(1071, 462)

gcd(1071, 462)	a mod b = 147
gcd(462, 147)	a mod b = 21
gcd(147, 21)	a mod b = 0
gcd(21, 0)	b = 0 → gcd = 21

복잡도 보통 한 단계에서 b는 절반 이하로 줄어듭니다. 따라서 전체 반복 횟수는 b의 크기에 대한 로그에 비례하며, 최악의 경우에도 O(log b)입니다. 평균적인 경우는 이보다 더 빠릅니다.

거듭제곱 계산 — 가장 먼저 떠오르는 방법

어떤 수 C를 n번 곱하는 문제입니다. 가장 단순한 방법은 C를 n번 반복해서 곱하는 것입니다. 이해하기 쉽고 구현도 간단하지만, 반복문이 n번 실행됩니다.

power.py

```
def power(C, n):
    result = 1
    for i in range(n):
        result = result * C
    return result
```

O(n)

n = 1,000,000이면 곱셈을 백만 번 수행합니다.

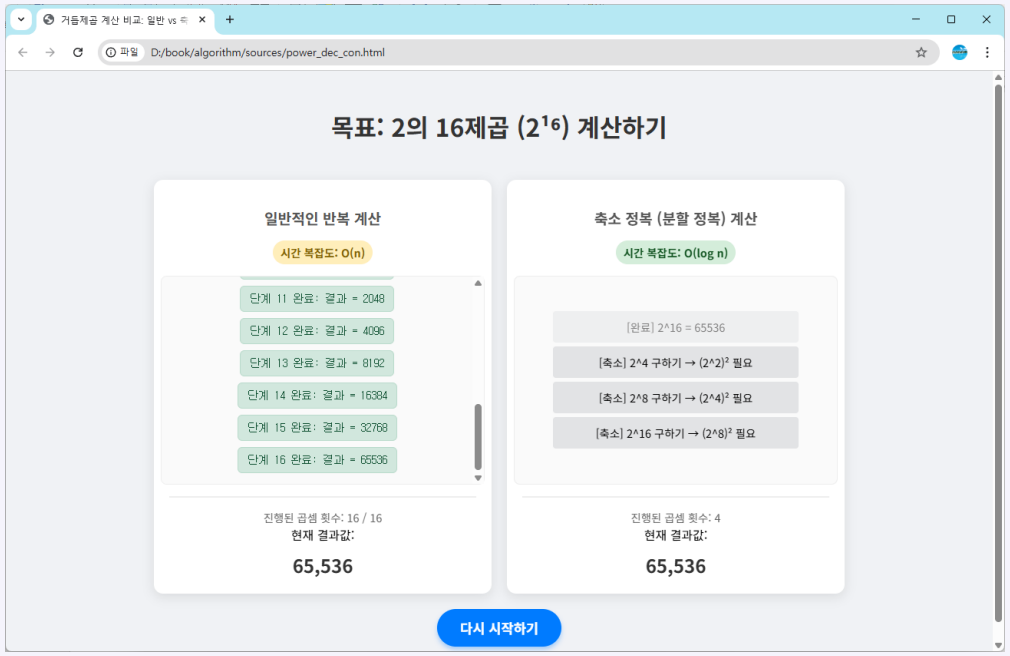
핵심 아이디어 — 큰 문제를 작은 문제로 나눈다

이 공식의 핵심은 $C^{n/2}$ 값을 한 번만 구하면, 그 값을 제곱하여 C^n 을 바로 알 수 있다는 점이다. 즉, 계산량이 절반으로 줄어든다. 예를 들어 2^{100} 을 구한다고 생각해 보자.

- 2^{100} 을 구하기 위해 2를 100번 곱할 필요가 없다. 2^{50} 만 알면 된다. $(2^{50} \times 2^{50})$
- 2^{50} 을 구하기 위해서는 2^{25} 만 알면 된다.
- 2^{25} 는 홀수이므로 $2^{12} \times 2^{12} \times 2$ 로 구한다.

축소 정복 거듭제곱의 동작 과정

n을 절반으로 줄여 내려간 뒤, 되돌아 올라오면서 제곱을 취합니다. 홀수 단계에서만 C를 한 번 더 곱합니다.



절반으로 내려갔다가 제곱하며 되돌아 올라온다

하강 (축소)
n → n/2 → n/4 → … → 0 까지 재귀 호출을 쌓는다

상승 (결합)
half × half 로 제곱, 홀수면 × C 를 한 번 더

결과
곱셈 횟수가 n번에서 $\log_2 n$번으로 줄어든다

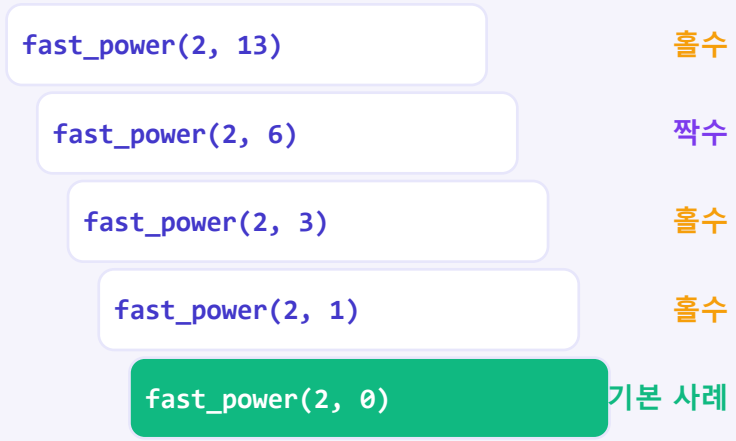
$$T(n) = T(n/2) + 1 \rightarrow O(\log n)$$

축소 정복을 이용한 거듭제곱

fast_power.py

```python
def fast_power(C, n):
    if n == 0:
        return 1
    # 축소: 문제 크기를 절반(n/2)으로 줄여 재귀 호출
    half = fast_power(C, n // 2)
    # 정복 및 결합
    if n % 2 == 0:      # n이 짝수일 때
        return half * half
    else:               # n이 홀수일 때
        return half * half * C
```

2^13 계산 과정



호출 5번으로 $2^{13} = 8192$ 를 계산합니다.

n을 한 번에 절반으로 줄인다 — 곱셈을 n번이 아니라 $\log_2 n$번만 수행하면 됩니다.

§3.5 유클리드·거듭제곱

축소 정복 거듭제곱의 시간 복잡도 분석

한 단계에서 문제를 절반(n/2)으로 줄이고 그 결과를 제곱하는 곱셈 1회를 수행합니다. (홀수일 때 한 번 더 곱하지만 상수로 취급)

$$T(n) = T(n/2) + 1 \quad (n > 1)$$

$$T(1) = 0 \quad \text{(기저 사례: 곱셈 필요 없음)}$$

이진 탐색과 동일한 형태의 점화식

매 단계 절반으로 줄이고 상수 시간의 일을 하므로, 전개하면 단계 수가 $\log_2 n$이 됩니다. 두 알고리즘은 겉모습이 전혀 다르지만 축소 구조가 같습니다.

O(log n)

	power	fast_power
n = 1,000	1,000회	10회
n = 100만	100만회	20회

실제 실행 시간 측정: 백문이 불여일견

파이썬의 time 모듈로 두 알고리즘의 실행 시간을 직접 측정해 봅시다.

measure.py

```python
import time

def measure_time(func, C, n, label):
    start = time.perf_counter()
    func(C, n)
    elapsed = time.perf_counter() - start
    print(f"[{label}] n={n} 걸린 시간: {elapsed:.6f}초")

measure_time(power_recursive, 2, 100000, "분할 정복")
measure_time(power_iterative, 2, 100000, "반복문")
```

측정 결과 (C = 2)

n	축소 정복	반복문	배수
100,000	0.000187초	0.141415초	**약 756배**
1,000,000	0.002338초	13.442806초	**약 5,749배**

n이 10배 커질 때
반복문은 약 95배 느려지고,
축소 정복은 거의 변하지 않습니다.

점근적 복잡도의 차이는 이론이 아니라 실측으로 드러납니다 — O(n)과 O(log n)의 간격은 n이 커질수록 벌어집니다.

SECTION 3.6

정렬과 선택 문제

삽입 정렬 · 위조 동전 찾기 · 퀵 셀렉트로 세 가지 축소 유형을 다시 확인합니다.

3.6

삽입 정렬 (동일 크기 축소)

크기 n 문제를 풀기 위해 n−1개가 이미 정렬되어 있다고 가정하고, n번째 원소를 그 정렬된 구간의 적절한 위치에 '삽입'하여 문제를 해결합니다.

insertion_sort.py

```python
def insertion_sort(A):
    for i in range(1, len(A)):
        key = A[i]              # 현재 비교할 값
        j = i - 1
        while j >= 0 and A[j] > key:
            A[j + 1] = A[j]     # 한 칸씩 뒤로 이동
            j -= 1
        A[j + 1] = key          # 알맞은 위치에 삽입
```

n−1개를 먼저 정렬하고 1개를 끼워 넣는다 — 문제 크기를 한 번에 1씩 줄이는 전형적인 동일 크기 축소입니다.

삽입 정렬의 시간 복잡도

최선의 경우 — 이미 정렬된 입력

외부 루프는 n−1번 실행되고, 각 단계에서 while 조건이 한 번 만에 거짓이 되어 비교 1회로 끝난다.

총 비교 횟수 = n − 1

O(n)

최악의 경우 — 역순으로 정렬된 입력

각 단계에서 앞의 자료들이 전부 한 칸씩 뒤로 이동해야 한다. i번째 반복에서 i번의 비교가 수행된다.

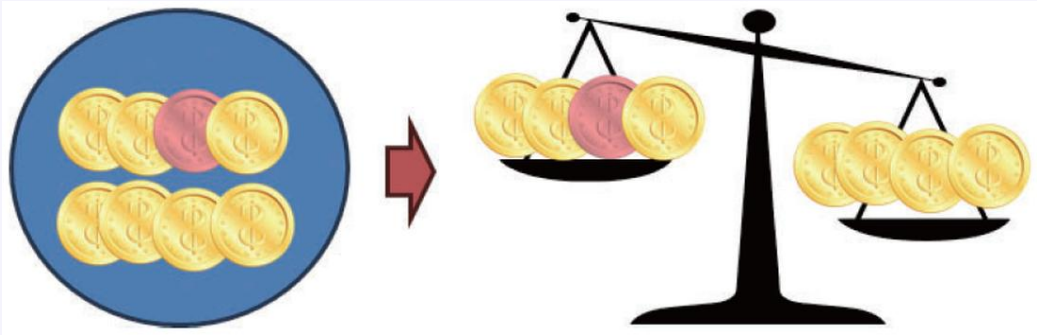
총 비교 = 1+2+…+(n−1) = n(n−1)/2

O(n²)

최선	평균	최악	공간
O(n)	O(n²)	O(n²)	O(1)

위조 동전 문제 (동일 비율 축소)

동일한 모양과 크기의 n개 동전 중 하나는 가짜로, 진짜보다 가볍습니다. 최소한의 저울질 횟수로 가짜 동전의 위치를 찾아내는 알고리즘을 설계하는 것이 목표입니다.



무게가 같다
따로 뺀 동전이 가짜

무게가 다르다
더 가벼운 더미에 가짜가 있다

반복
동전이 하나 남을 때까지 계속

탐색 범위가 매번 절반으로 — 동일 비율 축소 (n → n/2)입니다.

위조 동전 찾기 (2개로 나누기) — 전체 소스

find_fake_coin.py

```python
def find_fake_coin(coins, start=0):
    n = len(coins)
    if n == 1:              # 기저 사례: 하나뿐이면 그것이 가짜
        return start        # 원래 인덱스를 반환
    mid = n // 2
    group1 = coins[:mid]
    group2 = coins[mid:2*mid]
    remainder = coins[2*mid:]      # 홀수면 최대 1개
    if sum(group1) == sum(group2):
        return start + 2 * mid     # 남은 하나가 가짜
    elif sum(group1) < sum(group2):
        return find_fake_coin(group1, start)
    else:
        return find_fake_coin(group2, start + mid)
```

run.py

```python
coins = [1,1,1,1,0.9,1,1,1,1]
index = find_fake_coin(coins)
print(f"가짜 동전의 위치: {index}")
```

```
가짜 동전의 위치: 4 (무게: 0.9)
```

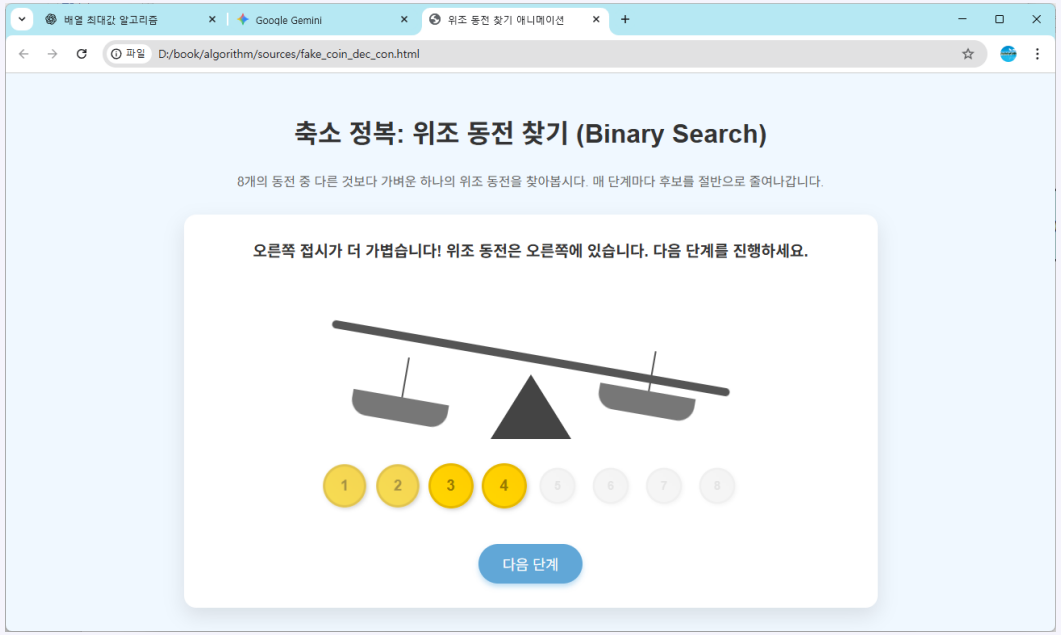
복잡도

$$T(n) = T(n/2) + 1$$
$$T(1) = 0$$

이진 탐색과 동일한 형태 → $O(\log_2 n)$

위조 동전 찾기 (2개로 나누기)

n개의 동전을 두 더미로 나눕니다. 홀수라면 한 개를 따로 빼 둡니다. 두 더미의 무게가 같으면 빼 둔 동전이 가짜이고, 다르면 가벼운 더미로 범위를 좁힙니다.



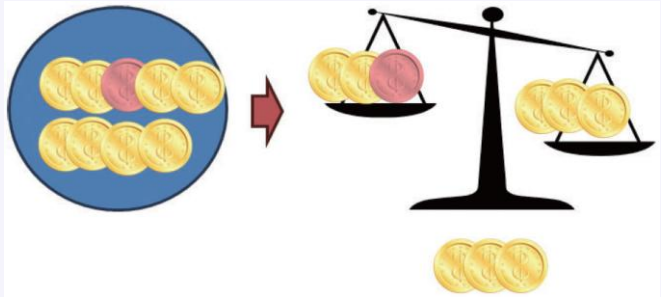
두 더미로 나누어 저울질 — 한 번에 후보의 절반이 사라진다

위조 동전 찾기 (3개로 나누기)

세 그룹으로 나누고 그중 두 그룹만 저울에 올립니다. 균형이면 나머지 한 그룹에, 기울면 가벼운 쪽에 가짜가 있습니다. 한 번의 저울질로 후보의 2/3를 버립니다.

find_fake_coin3.py

```python
def find_fake_coin(coins, left, right, weight_func):
    if left == right:       # 탐색 범위가 1개 → 발견
        return left
    size = (right - left + 1) // 3
    g1s, g1e = left, left + size - 1
    g2s, g2e = g1e + 1, g1e + size
    g3s, g3e = g2e + 1, right
    w = weight_func(coins[g1s:g1e+1], coins[g2s:g2e+1])
    if w == 0:              # 그룹1 == 그룹2
        return find_fake_coin(coins, g3s, g3e, weight_func)
    elif w < 0:             # 그룹1이 가벼움
        return find_fake_coin(coins, g1s, g1e, weight_func)
    else:                   # 그룹2가 가벼움
        return find_fake_coin(coins, g2s, g2e, weight_func)
```



복잡도

$T(n) = T(n/3) + 1$

$\rightarrow O(\log_3 n)$

2분할보다 상수배 빠릅니다.

k번째 작은 수 찾기 (선택 문제)

n개의 숫자 중 k번째로 작은 수를 찾는 문제입니다. 중앙값이나 상위 10% 경계값을 구하는 상황이 여기에 해당합니다. 정렬 후 k−1번 인덱스를 읽으면 되지만, 정렬에는 최소 O(n log n)이 듭니다.

정렬을 하지 않고도 이보다 빠르게 답을 찾을 수 있을까?

퀵 셀렉트의 핵심 아이디어 — 피벗 p 기준으로 파티션한 뒤

p = k−1

피벗 자체가 찾던 수. 탐색 종료.

p > k−1

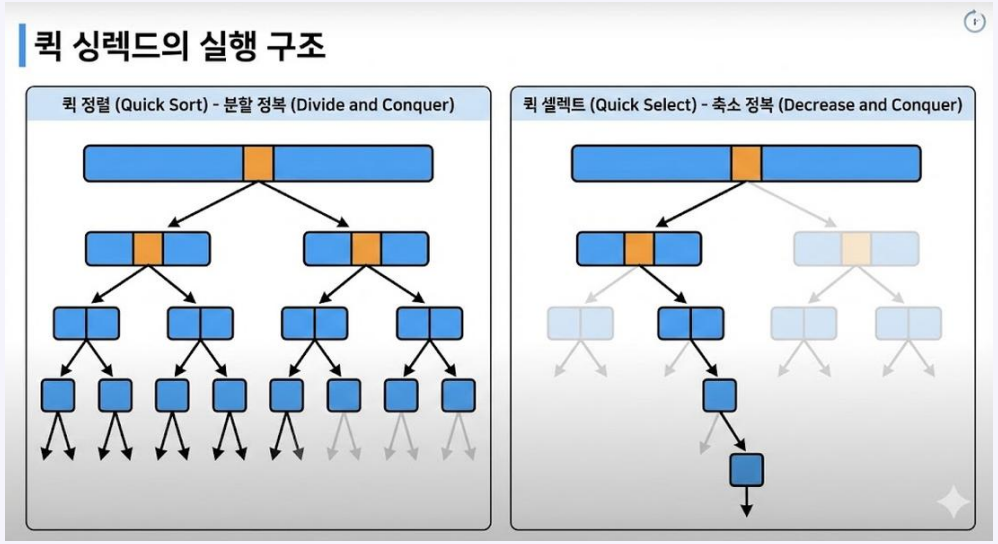
정답은 왼쪽에. 오른쪽은 버린다.

p < k−1

정답은 오른쪽에. 왼쪽은 버린다.

퀵 정렬 vs 퀵 셀렉트

퀵 셀렉트는 퀵 정렬의 원리를 빌려오되, 양쪽을 모두 정렬하는 대신 "정답이 있을 만한 한쪽만 남기는" 전형적인 축소 정복 전략을 사용합니다.



퀵 정렬 (분할 정복)

피벗을 기준으로 나뉜 두 부분 배열을 모두 재귀 호출하여 정렬한다. 두 갈래가 모두 살아남으므로 총 작업량이 n log n에 이른다.

$$T(n) = 2T(n/2) + n$$

퀵 셀렉트 (축소 정복)

정답이 있는 한쪽 부분 배열만 선택하여 재귀 호출한다. 나머지 반대쪽은 과감히 버린다. 평균 작업량이 n에 머문다.

$$T(n) = T(n/2) + n$$

버리는 것이 곧 알고리즘이다 — 필요 없는 절반을 버리는 순간 복잡도가 한 단계 내려갑니다.

퀵 셀렉트 — 전체 소스

quick_select.py (1/2)

```python
def quick_select(arr, k):
    # 피벗 선택 (편의상 중간 위치의 값)
    pivot = arr[len(arr) // 2]
    small, mid, big = [], [], []
    for x in arr:
        if x < pivot:
            small.append(x)
        elif x > pivot:
            big.append(x)
        else:
            mid.append(x)
```

quick_select.py (2/2)

```python
    # Case 1: small 안에 있을 때
    if k <= len(small):
        return quick_select(small, k)
    # Case 2: mid 안에 있을 때 (정답)
    elif k <= len(small) + len(mid):
        return pivot
    # Case 3: big 안에 있을 때
    else:
        return quick_select(big,
                            k - len(small) - len(mid))
```

run.py

```python
nums = [10, 4, 5, 8, 6, 11, 26]
print(f"3번째 작은 수: {quick_select(nums, 3)}")
```

실행 결과 **3번째 작은 수: 6**

정렬 없이 O(n) 평균 시간으로 찾았습니다.

퀵 셀렉트의 시간 복잡도 분석

파티션 자체에 O(n)이 들고, 그다음 한쪽만 재귀 호출합니다. 피벗이 얼마나 균형 있게 잡히느냐가 성능을 좌우합니다.

평균의 경우 — 피벗이 대략 절반을 가른다

$$T(n) = T(n/2) + n$$

$$n + n/2 + n/4 + \dots < 2n$$

$$O(n)$$

정렬(O(n log n))보다 빠릅니다.

최악의 경우 — 피벗이 항상 끝값

$$T(n) = T(n-1) + n$$

$$n + (n-1) + \dots + 1 = n(n+1)/2$$

$$O(n^2)$$

무작위 피벗 선택으로 회피할 수 있습니다.

실무에서는 피벗을 무작위로 고르거나 median-of-medians를 써서 최악의 경우를 회피합니다.

SECTION 3.7

Coding Test

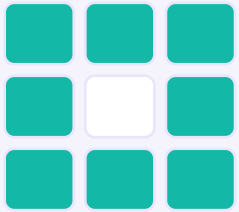
재귀적 사고를 직접 코드로 옮겨 봅니다. 세 문제 모두 축소 정복의 변주입니다.

3.7

Coding Test 3.1 — 별 모양 찍기

정사각형 별 무늬는 가운데가 비어 있고 그 주위를 동일한 패턴이 둘러싸며 점점 커집니다. 크기 N > 3인 경우 다음 규칙을 따릅니다.

1. 전체 패턴을 3등분하여 총 9개의 작은 영역으로 나눈다. (3×3 격자)
2. 가운데 영역은 공백으로 비워 두고, 나머지 8개 영역은 크기 N/3의 패턴으로 채운다.
3. 이 과정을 재귀적으로 반복한다.



가운데만 비우고 나머지 8칸을 재귀로 채운다

```
크기를 입력하시오: 9

*********
* ** ** *
*********
***   ***
* *   * *
***   ***
*********
* ** ** *
*********
```

Coding Test 3.1 — 전체 소스

make_star.py

```python
def make_star(n):
    if n == 1:
        return ['*']
    prev = make_star(n // 3)      # 작은 패턴 가져오기
    result = []
    for i in range(3):
        for line in prev:
            if i == 1:
                result.append(line + ' ' * (n // 3) + line)
            else:
                result.append(line * 3)
    return result

N = int(input("크기를 입력하시오: "))
for line in make_star(N):
    print(line)
```

기본 사례
n == 1이면 별 하나짜리 패턴 ['*']

재귀 사례
크기 n/3의 패턴을 먼저 얻는다

결합
가운데 줄만 공백을 끼워 넣고 나머지는 3번 반복

문제 크기 n → n/3 : 동일 비율 축소

</> §3.7 Coding Test

Coding Test 3.2 — 계단 오르기

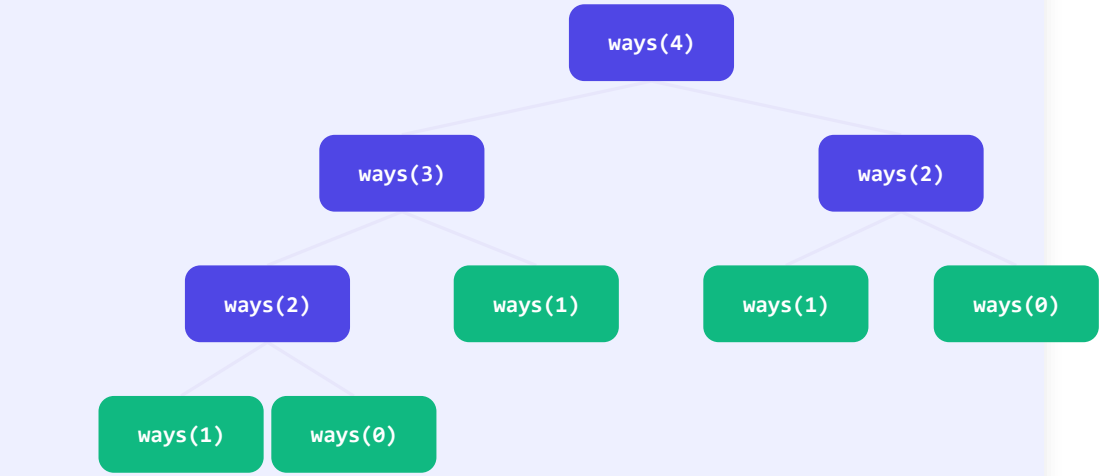
총 N개의 계단을 올라야 합니다. 한 번에 1칸 또는 2칸만 오를 수 있습니다. 꼭대기까지 올라가는 모든 경우의 수를 재귀 함수로 구하세요.

점화식 세우기

- 첫 번째 계단을 올라가는 법: 1가지
- 두 번째 계단을 올라가는 법: 2가지 (1+1 또는 2)
- k층에 올라가는 법 = (k−1)층에서 1칸 + (k−2)층에서 2칸

```
ways(k) = ways(k-1) + ways(k-2)
```

ways(4) 재귀 트리 — 답은 5



초록색 잎(기본 사례) 5개 = 경우의 수 5

Coding Test 3.2 — 전체 소스

ways.py

```python
def ways(n):
    if n == 0:
        return 1      # 정확히 도착
    if n < 0:
        return 0      # 불가능한 경로
    return ways(n - 1) + ways(n - 2)

N = int(input("계단 개수를 입력하세요: "))
print(ways(N))
```

```
계단 개수를 입력하세요: 4
5
```

관찰

피보나치 수열과 정확히 같은 점화식입니다. 따라서 순수 재귀는 지수 시간이며, 반복(또는 메모이제이션)으로 O(n)에 풀 수 있습니다.

주의 n이 조금만 커져도 중복 호출이 폭발합니다. fib 때와 같은 함정입니다.

$O(2^n) \rightarrow O(n)$

Coding Test 3.3 — 에너지 절약 로봇

로봇은 숫자 X에서 출발해 1로 줄어들어야 합니다. 세 가지 동작 중 하나를 쓸 수 있고 각 동작의 에너지 소모는 1입니다. 최소 에너지를 구하세요.

X − 1

언제나 가능

X ÷ 2

X가 2로 나누어떨어질 때

X ÷ 3

X가 3으로 나누어떨어질 때

min_energy.py

```python
def min_energy(x):
    if x == 1:
        return 0        # 이미 1이면 소모 없음
    energy = min_energy(x - 1)
    if x % 2 == 0:
        energy = min(energy, min_energy(x // 2))
    if x % 3 == 0:
        energy = min(energy, min_energy(x // 3))
    return energy + 1   # 한 번의 동작 추가
```

```
정수를 입력하시오: 12
결과: 3
```

12 → 4 → 2 → 1
(÷3, ÷2, ÷2 … 3회)

3장 정리

축소 정복은 "문제를 같은 유형의 더 작은 문제로 바꾼다"는 한 문장으로 요약됩니다. 무엇을 얼마나 버리느냐가 복잡도를 결정합니다.

축소 유형	크기 변화	대표 알고리즘	전형적 복잡도
동일 크기 축소	$n \to n - 1$	팩토리얼, 순차 탐색, 삽입 정렬	$O(n) \sim O(n^2)$
동일 비율 축소	$n \to n / c$	이진 탐색, 위조 동전, 빠른 거듭제곱	$O(\log n)$
가변 크기 축소	$n \to ?$	유클리드, 보간 탐색, 퀵 셀렉트	$O(\log n) \sim O(n)$

재귀 vs 반복

같은 전략을 하향식·상향식 중 어느 방향으로 전개할지의 문제

버리는 것이 힘

정답이 없는 쪽을 버릴수록 복잡도가 내려간다

분할 정복과의 차이

축소 정복은 한 갈래만, 분할 정복은 모든 갈래를 푼다

Q & A

다음 장에서는 문제를 여러 갈래로 쪼개 모두 푸는 분할 정복(Divide and Conquer)을 다룹니다.