그림으로 쉽게 설명하는 파이썬 알고리즘

CHAPTER 04

분할 정복 기법

Divide and Conquer

큰 문제를 잘게 쪼개어 정복하고, 다시 하나로 합친다

CONTENTS

이번 장에서 배울 내용

더불어 코딩 테스트 두 문제(색종이 만들기, 곱셈)를 분할 정복으로 직접 풀어본다.

4.1

분할 정복이란?

"나누어라, 그리고 정복하라"
큰 문제를 작은 문제로 쪼개어 해결하는 알고리즘 설계 전략

4.1

나누어라, 그리고 정복하라

고대 로마의 정치가이자 장군이었던 율리우스 카이사르(Julius Caesar)는 "나누어라, 그리고 정복하라"라는 유명한 말을 남겼다. 적을 여러 갈래로 분열시켜 세력을 약화시킨 뒤, 작아진 적들을 하나씩 각개격파한다면 승리는 훨씬 쉬워진다.

> 이 역사적 지혜를 알고리즘으로 계승한 것이 바로 분할 정복(divide and conquer) 기법이다.

Divide → Conquer → Combine

직관적 이해 — 거대한 책 요약하기

1,000페이지짜리 책을 혼자 요약하려면 막막하다. 하지만 친구 10명이 있다면 이야기가 달라진다.

나누기 (divide)

책을 100페이지씩 10등분하여 친구 10명에게 한 부분씩 나눠준다.

해결하기 (conquer)

각 친구는 맡은 100페이지만 읽고 요약한다. 1,000페이지는 불가능해도 100페이지는 금방이다.

합치기 (combine)

10개의 요약본을 순서대로 모아 붙인다. 1,000페이지 책의 최종 요약본이 완성된다.

큰 문제를 그대로 상대하면 어렵지만, 작게 쪼개면 각각은 쉽다 — 이것이 분할 정복의 핵심 통찰이다.

분할 정복의 3단계

분할 (Divide)

문제를 같은 유형의 더 작은 하위 문제 여러 개로 나눈다.

정복 (Conquer)

하위 문제를 재귀적으로 해결한다. 충분히 작아지면 직접 푼다.

결합 (Combine)

하위 문제의 해답을 조합해 원래 문제의 해답을 만든다.

분할 정복은 한 번만 나누는 알고리즘이 아니다

분할된 문제가 아직 커서 즉시 해결할 수 없다면 다시 분할한다. 아주 쉽게 풀릴 때까지 계속 분할한다.

divide_and_conquer.pseudo

```
DIVIDE_AND_CONQUER(P):
    if P의 크기가 충분히 작으면 then return SolveDirectly(P)
    P1, P2, ..., Pk <- Divide(P)                  // 분할
    Si <- DIVIDE_AND_CONQUER(Pi)   for i = 1..k   // 정복
    return Combine(S1, S2, ..., Sk)               // 결합
```

4.2

합병 정렬

4.2

1945년 폰 노이만이 제안한, 분할 정복의 모범 답안
나누는 과정은 단순하지만 합치는 과정에서 마법처럼 정렬이 완성된다

합병 정렬 (Merge Sort)

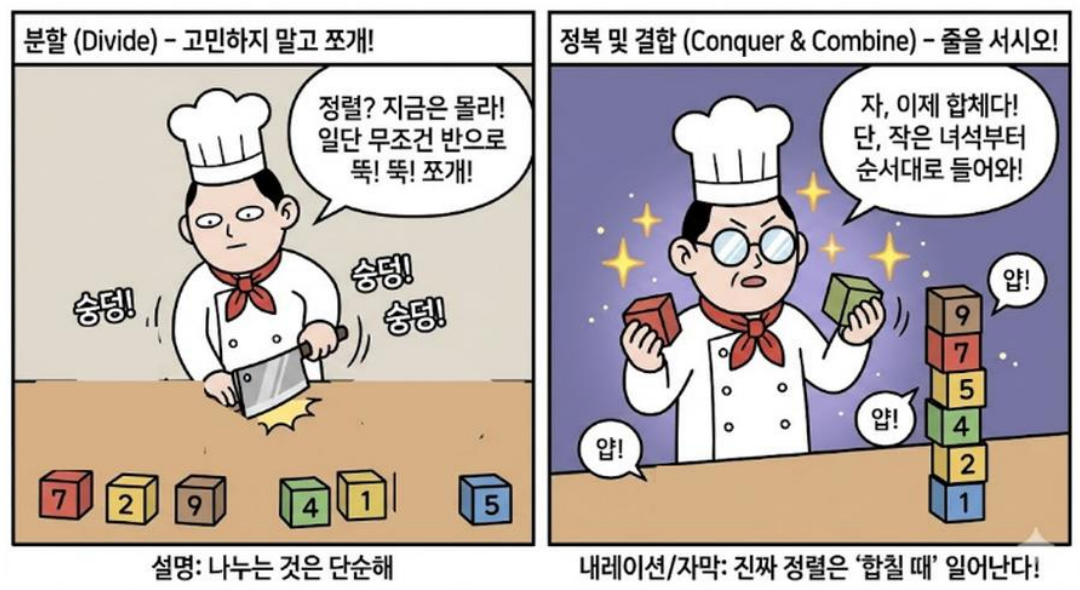
합병 정렬은 1945년 폰 노이만(John von Neumann)이 아이디어를 제안한 유서 깊은 알고리즘으로, 분할 정복의 모범 답안과도 같은 기법이다.

나누는 과정은 너무나 단순하지만, 합치는 과정에서 마법처럼 정렬이 완성된다.

1945
폰 노이만 제안

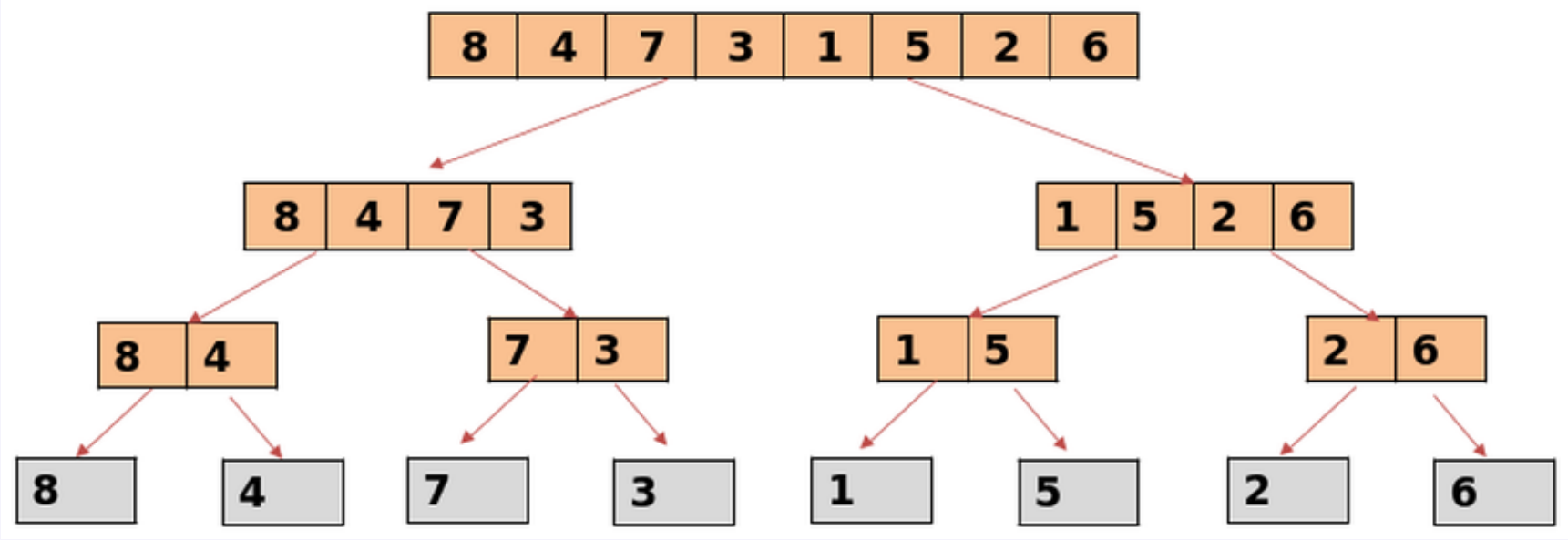
O(n log n)
모든 경우 동일

안정
stable sort



나누는 건 쉽다 (divide)

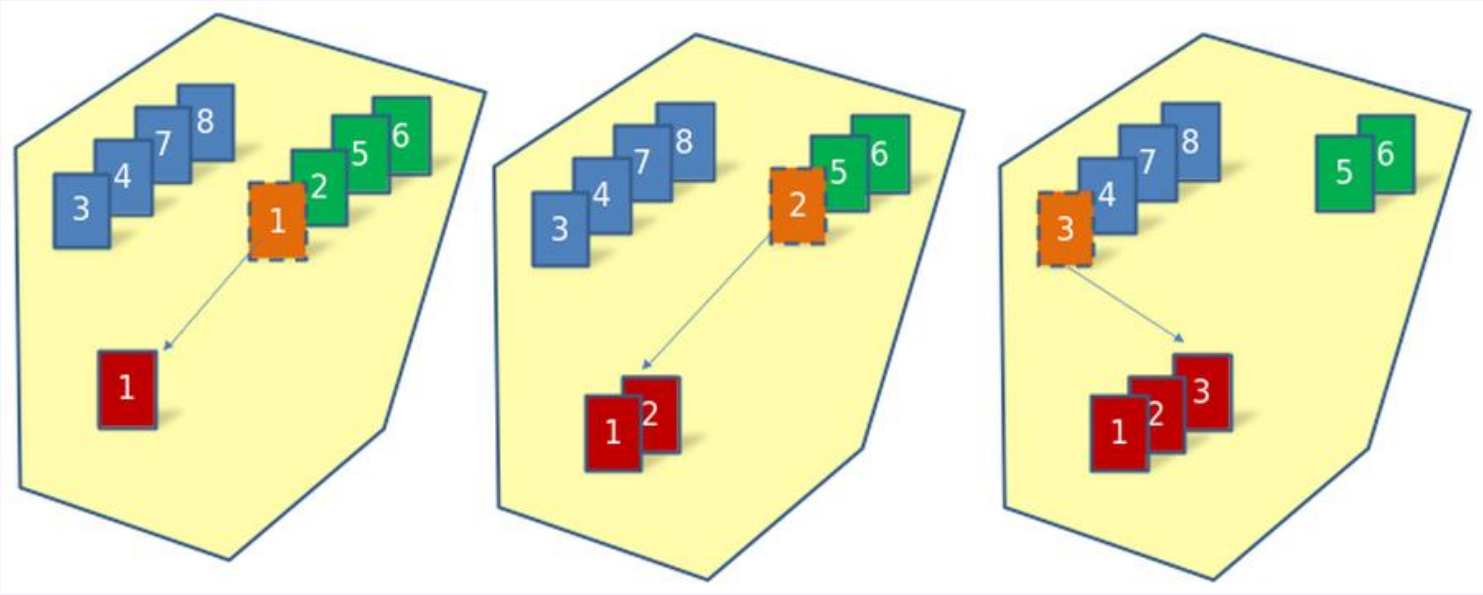
숫자 카드 8장이 섞여 있다. 합병 정렬은 먼저 카드를 반으로 나눈다. 그리고 다시 반으로, 또 반으로 — 카드가 한 장씩 남을 때까지 계속 나눈다. 여기까지는 아무런 비교도, 계산도 필요 없다.



길이 n인 배열은 $\log_2 n$ 번만 반으로 나누면 길이 1이 된다. 길이 1인 배열은 이미 정렬되어 있다 — 이것이 기저 사례다.

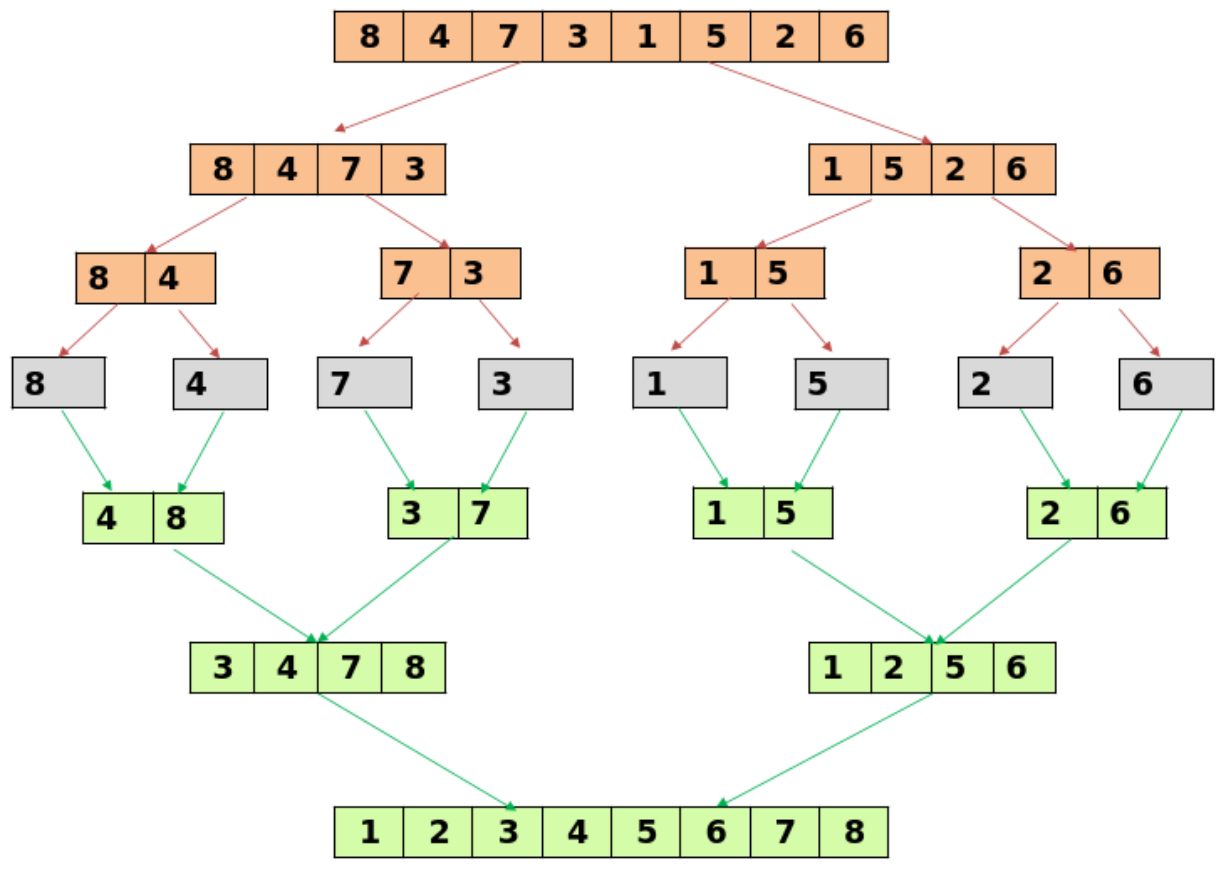
'결합'의 마법 (combine)

진짜 작업은 이제부터다. 흩어진 조각들을 다시 하나로 합쳐야 한다. 그런데 그냥 합치는 것이 아니라 "순서대로 정리하면서" 합친다. 이것을 합병(merge)이라고 한다.



두 개의 정렬된 배열을 합칠 때는, 양쪽 맨 앞 원소만 비교하면 된다. 더 작은 쪽을 꺼내 놓기를 반복하면 정렬된 하나의 배열이 만들어진다.

합병 정렬의 전체 과정



아래로 내려가며 분할

배열을 절반씩 쪼개 길이 1이 될 때까지 내려간다. 총 $\log_2 n$ 단계.

위로 올라오며 합병

두 정렬된 조각을 비교하며 하나로 합친다. 각 단계마다 n개 원소를 훑는다.

결과: O(n log n)

$\log_2 n$ 단계 × 각 단계 $O(n)$ 비용 = $O(n \log n)$.

합병 정렬 알고리즘 (의사코드)

merge_sort.pseudo

```
// 문제: 입력 리스트를 정렬한다.
// 입력: 리스트 A,  구간 [left, right]
// 출력: 정렬된 리스트 A

MERGE_SORT(A, left, right):
    if left < right then
        mid <- (left + right) / 2
        MERGE_SORT(A, left, mid)
        MERGE_SORT(A, mid + 1, right)
        MERGE(A, left, mid, right)
```

1. 배열의 크기가 2 이상인지 확인
2. 중간 지점을 계산
3. 왼쪽 부분을 재귀적으로 정렬
4. 오른쪽 부분을 재귀적으로 정렬
5. 정렬된 두 부분을 병합

핵심은 MERGE

분할은 인덱스 계산만으로 끝난다. 실제 정렬 작업은 전부 MERGE에서 일어난다. left < right 조건이 거짓이 되는 순간(원소 1개)이 재귀의 종료 조건이다.

합병 정렬 코드 — 분할부

merge_sort.py

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    # 1. 분할 (Divide): 크기가 1이 될 때까지 쪼갠다
    mid = len(arr) // 2
    left_group  = merge_sort(arr[:mid])
    right_group = merge_sort(arr[mid:])

    # 2. 정복 및 결합 (Conquer & Combine): 정렬하며 합친다
    return merge(left_group, right_group)
```

기저 사례

원소가 0개 또는 1개면 이미 정렬된 상태이므로 그대로 반환한다.

슬라이싱 분할

arr[:mid]와 arr[mid:]로 두 조각을 만든다. 파이썬에서는 새 리스트가 생성된다.

재귀 후 병합

두 조각이 각각 정렬되어 돌아오면, merge()가 하나로 합친다.

왜 mid = len(arr) // 2 인가?

정수 나눗셈(//)을 써야 인덱스가 정수로 유지된다. / 를 쓰면 실수가 되어 슬라이싱에서 오류가 발생한다.

합병 정렬 코드 — 병합부

merge.py

```python
def merge(left, right):
    result = []
    i = 0   # 왼쪽 그룹 인덱스
    j = 0   # 오른쪽 그룹 인덱스

    # 두 그룹을 비교하며 작은 것부터 차곡차곡 쌓는다
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:   # 등호가 안정 정렬을 보장!
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # 남은 요소들을 털어 넣는다
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

test.py

```python
test_arr = [7, 3, 5, 2, 8, 1, 4, 6]
print("정렬 전:", test_arr)

sorted_arr = merge_sort(test_arr)
print("정렬 후:", sorted_arr)
```

OUTPUT

```
정렬 전: [7, 3, 5, 2, 8, 1, 4, 6]
정렬 후: [1, 2, 3, 4, 5, 6, 7, 8]
```

병합 한 번의 비용은 두 조각의 길이 합 — 즉 O(n)이다.

§4.2 합병 정렬

합병 정렬의 중요한 특성 — 안정 정렬

합병 정렬에는 퀵 정렬 같은 다른 빠른 정렬이 갖지 못한 장점이 있다. 바로 안정 정렬(stable sort)이라는 점이다. 안정 정렬이란 "값이 같은 원소가 있을 때, 정렬 전의 순서가 정렬 후에도 유지되는 것"을 의미한다.

정렬 전 명단

- 90점 학생 A: 김철수
- 90점 학생 B: 이영희

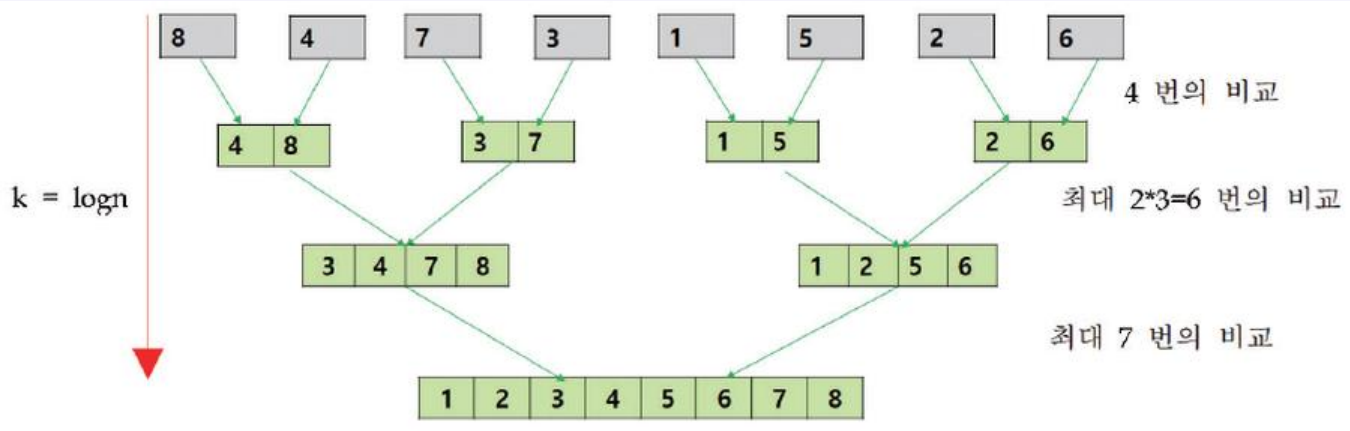
정렬 후 명단

- 90점 학생 A: 김철수
- 90점 학생 B: 이영희

순서가 뒤바뀌지 않는다

원래 명단에서 김철수가 이영희보다 앞에 있었다면, 합병 정렬 후에도 김철수는 여전히 앞에 위치한다. merge()에서 left[i] <= right[j] 의 등호(=)가 이 성질을 보장한다.

합병 정렬의 시간 복잡도



점화식

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

분할 $\log_2 n$ 단계 × 병합 O(n) 비용

$$T(n) = O(n \log n)$$

경우	시간 복잡도	설명
최악의 경우	O(n log n)	모든 요소를 정렬하는 데 O(n log n) 시간이 필요
평균적인 경우	O(n log n)	입력에 관계없이 동일한 연산을 수행
최선의 경우	O(n log n)	이미 정렬된 경우에도 동일한 단계를 수행

4.3

최대 구간 합

4.3

Maximum Subarray Sum

주가 등락폭 데이터에서 가장 큰 이익을 내는 연속 구간을 찾아라

§4.3 최대 구간 합

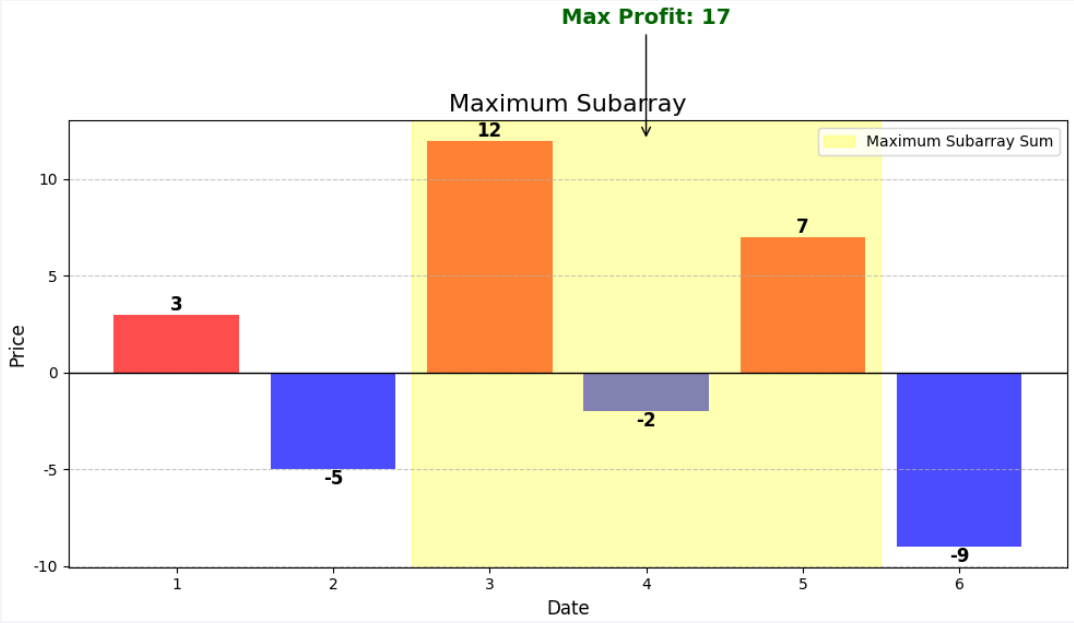
문제 — 가장 큰 이익을 내는 구간은?

A 회사의 지난 며칠간의 주가 등락폭(변동분) 데이터가 주어졌다. 목표는 가장 큰 이익을 낼 수 있는 연속된 구간을 찾는 것이다.

3	-5	12	-2	7	-9
1일	2일	3일	4일	5일	6일

처음부터 끝까지 들고 있었다면? 3 − 5 + 12 − 2 + 7 − 9 = 6

3일 차에 사서 5일 차에 팔았다면? 12 + (−2) + 7 = 17 ← 최대!



§4.3 최대 구간 합

직관적인 접근의 한계

가장 단순한 방법(완전 탐색)은 가능한 모든 시작일과 종료일을 다 계산해 보는 것이다. 데이터가 10개밖에 없다면 할 만하다.

1일 차에 샀을 때 → 2일 차부터 10일 차까지 (9가지)

2일 차에 샀을 때 → 3일 차부터 10일 차까지 (8가지)

…

9일 차에 샀을 때 → 10일 차에 파는 경우 (1가지)

$$9 + 8 + 7 + \cdots + 1 \approx (n - 1) \times n / 2$$

데이터가 10만 개라면 약 50억 번의 계산이 필요하다. $O(n^2)$은 현실적이지 않다.

완전 탐색의 비용

n = 100	약 5,000회
n = 1,000	약 50만 회
n = 100,000	약 50억 회

연산 횟수는 n의 제곱에 비례해 폭발한다.

§4.3 최대 구간 합

분할 정복 전략 — 셋 중 하나에는 반드시 있다!

배열을 정확히 절반으로 뚝 잘라본다. 우리가 찾는 최대 구간은 논리적으로 다음 세 가지 경우 중 하나에 반드시 속한다.

1

왼쪽 구역에 숨어 있다

자른 선의 왼쪽에 온전히 존재하는 경우. 재귀 호출로 해결한다.

2

오른쪽 구역에 숨어 있다

자른 선의 오른쪽에 온전히 존재하는 경우. 역시 재귀 호출.

3

가운데를 걸치고 있다

자른 선을 가로질러 양쪽을 모두 포함하는 경우. 직접 계산해야 한다.

세 값 중 최댓값이 정답

max(leftMax, rightMax, crossMax). 1번과 2번은 같은 문제의 절반 크기이므로 재귀로 처리하고, 3번만 별도로 계산하면 된다.

'걸친 구간' 처리하기

걸친 구간을 처리하는 부분이 분할 정복 알고리즘에서 가장 까다롭다. 핵심은 "걸친 구간은 반드시 중앙(mid)을 포함해야 한다"는 것이다. 우리가 찾는 것은 왼쪽 동네와 오른쪽 동네를 연결하는 다리 같은 구간이다.

mid에서 왼쪽으로 훑으며 최대 합(leftSum), mid+1에서 오른쪽으로 훑으며 최대 합(rightSum). 둘을 더하면 걸친 구간의 최대 합이다.

최대 구간 합 알고리즘 (메인)

max_subarray.pseudo

```
// 입력: 배열 A, 시작 인덱스 low, 끝 인덱스 high
// 출력: A[low..high] 내 최대 연속 부분 배열의 합

MaxSubarraySum(A, low, high):
    if low == high then    // 기저 사례: 원소가 하나
        return A[low]

    mid <- (low + high) / 2
    leftMax  <- MaxSubarraySum(A, low, mid)
    rightMax <- MaxSubarraySum(A, mid + 1, high)
    crossMax <- MaxCrossingSum(A, low, mid, high)

    return max(leftMax, rightMax, crossMax)
```

기저 사례
low == high 이면 원소가 하나뿐이므로 그 값이 곧 최대 구간 합이다.

분할
mid를 기준으로 왼쪽·오른쪽 두 개의 절반 문제로 나눈다.

정복
두 절반을 재귀 호출로 해결한다. 문제 크기는 매번 절반이 된다.

결합
걸친 구간을 계산하고, 세 후보 중 최댓값을 돌려준다.

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

최대 구간 합 알고리즘 (걸친 구간)

max_crossing.pseudo

```
MaxCrossingSum(A, low, mid, high):

    // 1. 왼쪽 부분 최대 합 (중앙에서 왼쪽으로 이동)
    leftSum <- -INF
    sum <- 0
    for i <- mid downto low do
        sum <- sum + A[i]
        if sum > leftSum then leftSum <- sum

    // 2. 오른쪽 부분 최대 합 (mid+1에서 오른쪽으로)
    rightSum <- -INF
    sum <- 0
    for j <- mid + 1 to high do
        sum <- sum + A[j]
        if sum > rightSum then rightSum <- sum

    // 3. 중앙을 걸친 전체 구간의 합
    return leftSum + rightSum
```

왜 −∞ 로 초기화하는가?

모든 원소가 음수인 배열에서도 올바른 최댓값을 찾기 위해서다. 0으로 초기화하면 "아무것도 안 고르는 것"이 최선이 되어 버린다.

mid를 반드시 포함한다

왼쪽 스캔은 mid에서 출발하고 오른쪽 스캔은 mid+1에서 출발하므로, 두 구간은 항상 경계선을 가로지른다.

비용: O(n)

파이썬 구현 — 걸친 구간 (combine)

max_crossing_sum.py

```python
def max_crossing_sum(arr, low, mid, high):

    # [중앙 -> 왼쪽] 으로 이동하며 최대 합 찾기
    left_sum = float('-inf')   # 음의 무한대로 초기화
    current_sum = 0
    for i in range(mid, low - 1, -1):   # mid부터 low까지 거꾸로
        current_sum += arr[i]
        if current_sum > left_sum:
            left_sum = current_sum

    # [중앙 -> 오른쪽] 으로 이동하며 최대 합 찾기
    right_sum = float('-inf')
    current_sum = 0
    for i in range(mid + 1, high + 1):
        current_sum += arr[i]
        if current_sum > right_sum:
            right_sum = current_sum

    return left_sum + right_sum
```

range의 끝은 포함되지 않는다

mid부터 low까지 거꾸로 돌려면 range(mid, low − 1, −1)로 써야 low가 포함된다.

한 번의 선형 스캔

왼쪽 스캔과 오른쪽 스캔 모두 각 원소를 한 번씩만 훑는다. 따라서 이 함수의 비용은 구간 길이에 비례하는 O(n)이다.

파이썬 구현 — 분할 정복 메인

max_subarray_sum.py

```python
def max_subarray_sum(arr, low, high):
    # 기저 사례(base case): 원소가 하나뿐일 때
    if low == high:
        return arr[low]

    # 분할(divide): 리스트를 반으로 나눔
    mid = (low + high) // 2

    # 정복(conquer): 왼쪽과 오른쪽 각각 재귀 호출
    left_max  = max_subarray_sum(arr, low, mid)
    right_max = max_subarray_sum(arr, mid + 1, high)

    # 결합(combine): 중앙을 걸친 구간 구하기
    cross_max = max_crossing_sum(arr, low, mid, high)

    return max(left_max, right_max, cross_max)
```

run.py

```python
# 주가 등락폭 데이터
stock_prices = [3, -5, 12, -2, 7, -9, 5, 1]
n = len(stock_prices)
max_profit = max_subarray_sum(stock_prices, 0, n - 1)
print(f"최대 구간 합(최대 이익): {max_profit}")
```

세 개의 후보

왼쪽 절반의 최대, 오른쪽 절반의 최대, 그리고 중앙을 걸친 최대. 파이썬 내장 max()로 셋 중 가장 큰 값을 고른다.

OUTPUT

```
주가 데이터: [3, -5, 12, -2, 7, -9, 5, 1]
최대 구간 합(최대 이익): 17
```

$$T(n) = O(n \log n)$$

시간 복잡도 분석

분할	배열을 반으로 나누는 데는 계산이 필요 없다. 인덱스 하나만 구하면 된다.	$O(1)$
정복	두 개의 절반 크기 문제를 재귀적으로 푼다.	$2T(n/2)$
결합	max_crossing_sum()이 중앙에서 양쪽으로 뻗어나가며 더한다. 배열 길이에 비례한다.	$O(n)$

$$T(n) = 2T(n/2) + O(n) \quad \Rightarrow \quad T(n) = O(n \log n)$$

합병 정렬과 완전히 동일한 형태의 점화식이다. 마스터 정리(§4.6)로 바로 풀 수 있다.

4.4

최근접 쌍 문제

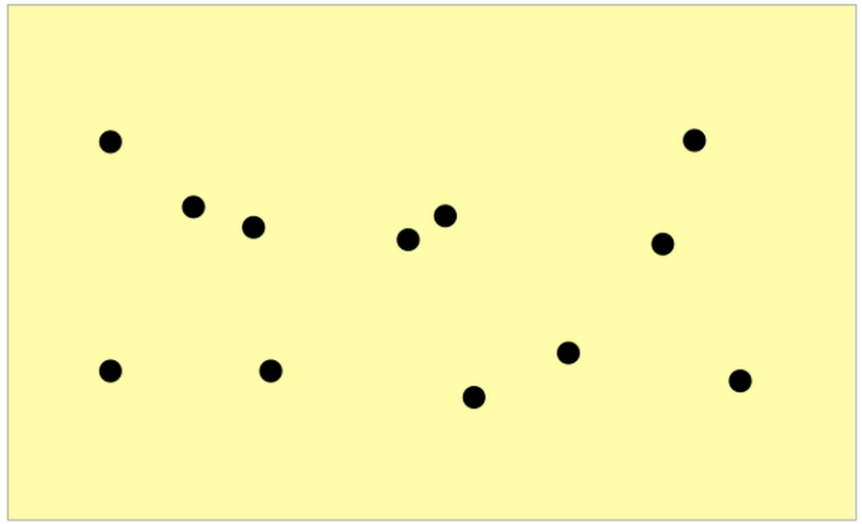
4.4

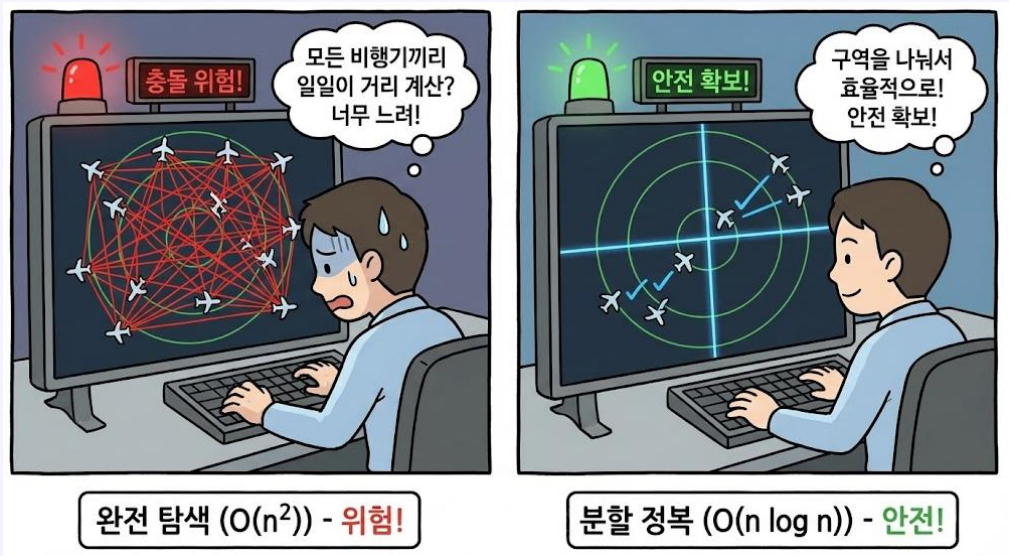
Closest Pair of Points
평면 위 n개의 점 중 가장 가까운 두 점을 찾아라

§4.4 최근접 쌍 문제

문제 — 평면 위 가장 가까운 두 점

2차원 평면에 n개의 점이 흩어져 있다. 이 중 거리가 가장 가까운 두 점을 찾는 것이 최근접 쌍 문제다. 항공 관제 시스템에서 충돌 위험이 있는 두 항공기를 찾거나, 지도 앱에서 가장 가까운 시설을 찾는 문제와 본질적으로 같다.





항공 관제 — 1초가 생명

레이더에 수백 대의 항공기가 잡힌다. 가장 가까운 두 대를 실시간으로 찾아내야 충돌을 막을 수 있다.

§4.4 최근접 쌍 문제

무식한 접근 — 모든 쌍을 다 재본다

가장 단순한 방법은 모든 점의 쌍에 대해 거리를 계산하고 최솟값을 고르는 것이다. 첫 번째 점은 나머지 (n−1)개와, 두 번째 점은 (n−2)개와 비교해야 한다.

$$(n-1) + (n-2) + \cdots + 1 = n(n-1)/2 = O(n^2)$$

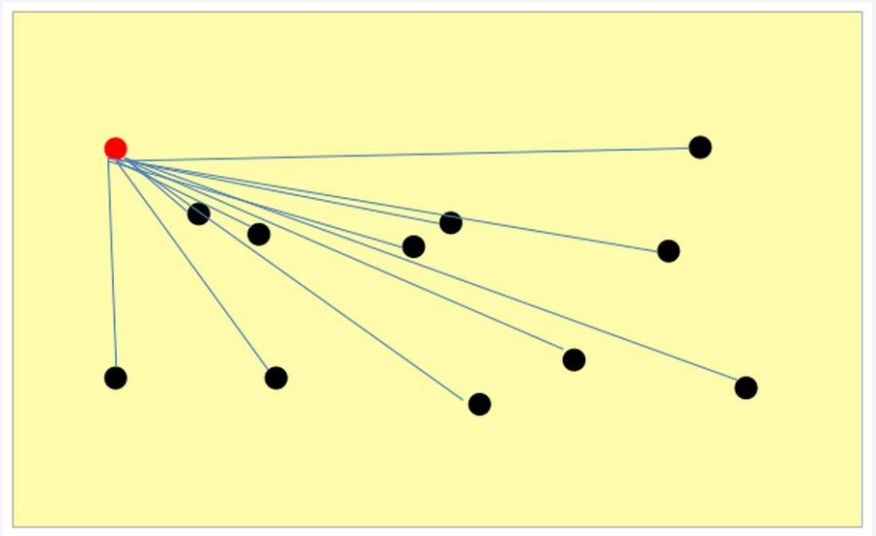
brute_force.py

```python
import math

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def distance(p1, p2):        # 두 점 사이의 거리
    return math.hypot(p1.x - p2.x, p1.y - p2.y)

def ClosestPairBruteForce(P):
    min_dist = float('inf')
    for i in range(len(P)):
        for j in range(i + 1, len(P)):
            min_dist = min(min_dist, distance(P[i], P[j]))
    return min_dist
```



점이 10만 개라면?

약 50억 번의 거리 계산이 필요하다.
매 프레임마다 이 계산을 반복할 수는 없다.

§4.4 최근접 쌍 문제

분할 정복 전략 — 평면을 반으로 가른다

x좌표를 기준으로 점들을 정렬한 뒤, 정확히 절반이 되는 지점에 수직선을 긋는다. 그러면 최근접 쌍은 다음 세 경우 중 하나에 속한다.

1

두 점 모두 왼쪽에

재귀 호출로 왼쪽 절반의 최소 거리 dL을 구한다.

2

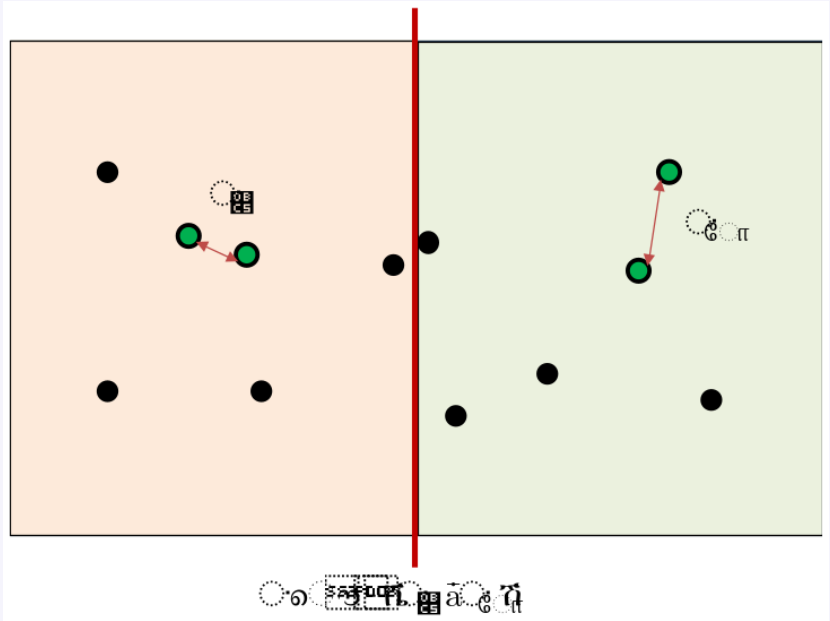
두 점 모두 오른쪽에

재귀 호출로 오른쪽 절반의 최소 거리 dR을 구한다.

3

경계선을 사이에 두고

한 점은 왼쪽, 다른 점은 오른쪽. 별도로 처리해야 한다.

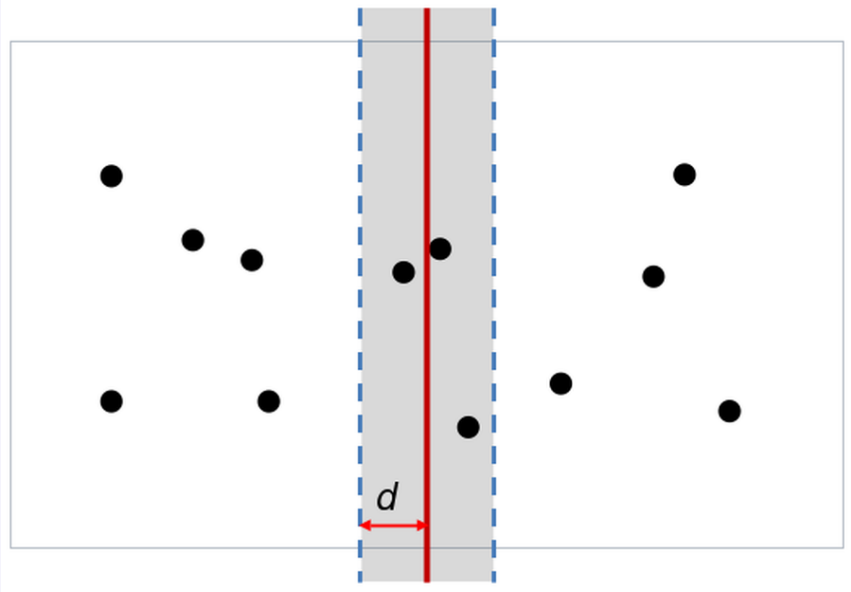


d = min(dL, dR)을 먼저 구한 뒤, 경계선 근처만 추가로 살펴본다.

§4.4 최근접 쌍 문제

경계선의 함정 — '띠(strip)'만 살펴보면 된다

경계선을 걸친 쌍을 모두 검사하면 다시 $O(n^2)$이다. 하지만 이미 d = min(dL, dR)을 알고 있다. d보다 가까운 쌍만 찾으면 되므로, 경계선에서 좌우로 d 이내에 있는 점들만 후보가 된다.



후보를 띠 안으로 좁힌다

경계선에서 수평 거리가 d 이상 떨어진 점은, 어떤 짝을 만나도 거리가 d를 넘는다. 검사할 필요조차 없다.

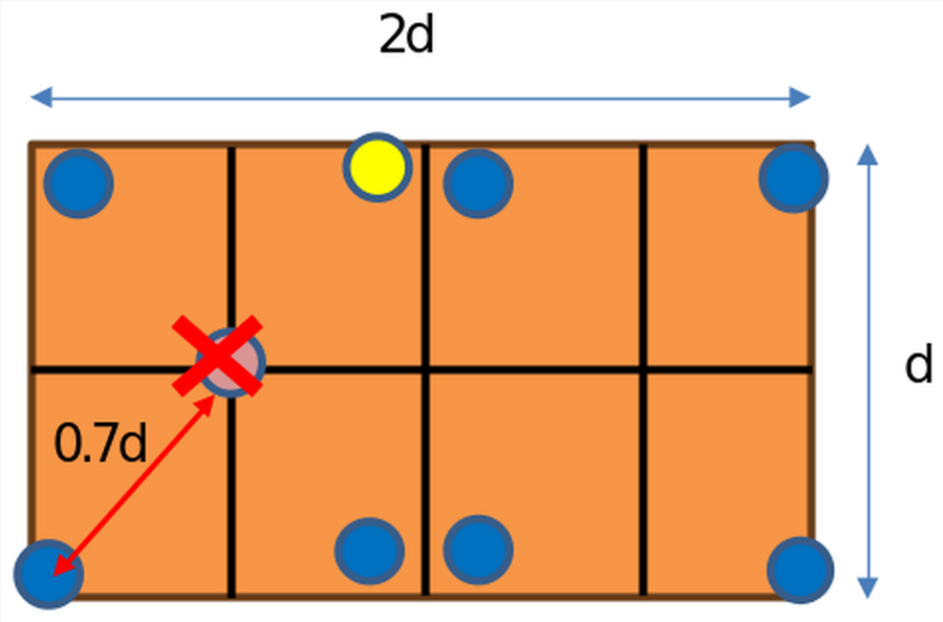
띠 안의 점을 y좌표로 정렬

띠 안의 점들을 y좌표 순으로 세우면, 자기보다 y가 d 이상 큰 점은 더 볼 필요가 없다.

그래도 걱정된다

띠 안에 점이 아주 많이 몰려 있으면? 여전히 $O(n^2)$이 아닐까?

증명 — 한 점당 최대 7개만 비교하면 된다



2d × d 직사각형

띠 안의 한 점 p에서 y 방향으로 d 이내의 영역은 가로 2d, 세로 d인 직사각형이다.

8개의 정사각형 칸

이 직사각형을 (d/2) × (d/2) 크기의 칸 8개로 나눈다.

한 칸에 점은 최대 1개

한 칸의 대각선 길이는 약 0.707d < d. 같은 칸에 두 점이 있으면 이미 d보다 가까운 쌍이 존재해 모순이다.

따라서 최대 7번 비교

직사각형 안의 점은 자기 자신 포함 최대 8개. 즉 한 점당 최대 7번만 비교하면 충분하다.

$$\text{띠 안 점 검사 비용} = O(n) \times 7 = O(n)$$

파이썬 구현 — 분할 정복 본체

closest_pair.py

```python
def ClosestPair(P):        # P는 x좌표로 정렬된 점 리스트
    if len(P) <= 3:
        return ClosestPairBruteForce(P)

    # 1. 분할: x축 기준으로 절반으로 나눈다
    mid = len(P) // 2
    Q = P[:mid]           # 왼쪽 절반
    R = P[mid:]           # 오른쪽 절반

    # 2. 정복: 재귀적으로 각각의 최소 거리를 구한다
    d_left  = ClosestPair(Q)
    d_right = ClosestPair(R)
    d = min(d_left, d_right)

    # 3. 결합: 경계선에서 거리 d 이내의 점만 추출
    mid_x = P[mid].x
    strip = [p for p in P if abs(p.x - mid_x) < d]

    # 4. 띠를 y좌표로 정렬 후, 앞쪽 6개까지만 비교
    strip.sort(key=lambda p: p.y)
    for i in range(len(strip)):
        for j in range(i + 1, min(i + 7, len(strip))):
            if distance(strip[i], strip[j]) < d:
                d = distance(strip[i], strip[j])
    return d
```

기저 사례: 점 3개 이하

점이 3개 이하면 나누는 비용이 오히려 크다. 그냥 완전 탐색으로 푼다.

strip 추출

리스트 컴프리헨션 한 줄로 후보를 골라낸다.

min(i + 7, len)

증명 덕분에 안쪽 반복은 상수 횟수로 끝난다.

실행 결과와 시간 복잡도

run.py

```
# 사용 예시
points = [
    Point(2, 3), Point(3, 4), Point(5, 1),
    Point(12, 10), Point(12, 30), Point(40, 50)
]

# x좌표 기준 정렬 필수!
points_sorted_by_x = sorted(points, key=lambda p: p.x)
result = ClosestPair(points_sorted_by_x)
print("가장 가까운 두 점 사이의 거리:", result)
```

OUTPUT

```
가장 가까운 두 점 사이의 거리: 1.4142135623730951
```

(2, 3)과 (3, 4) 사이의 거리 $\sqrt{2} \approx 1.414$ 가 정답이다.

분할	x좌표 기준으로 절반	O(1)
정복	두 개의 절반 문제 재귀	2T(n/2)
결합	strip 추출 + 상수 회 비교	O(n)

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

$$\Rightarrow$$

$$O(n \log n)$$

4.5

쿼드 트리

Quadtree
2차원 공간을 재귀적으로 4등분하는 자료 구조
이미지 압축부터 게임 충돌 감지까지

쿼드 트리의 기본 아이디어

쿼드 트리(quadtree)는 2차원 공간을 재귀적으로 4개의 구역(사분면)으로 분할하여 데이터를 효율적으로 표현하는 자료 구조다. 이진 탐색이 탐색 범위를 1/2씩 줄여나갔다면, 쿼드 트리는 평면을 1/4씩 줄여나간다.

① 확인

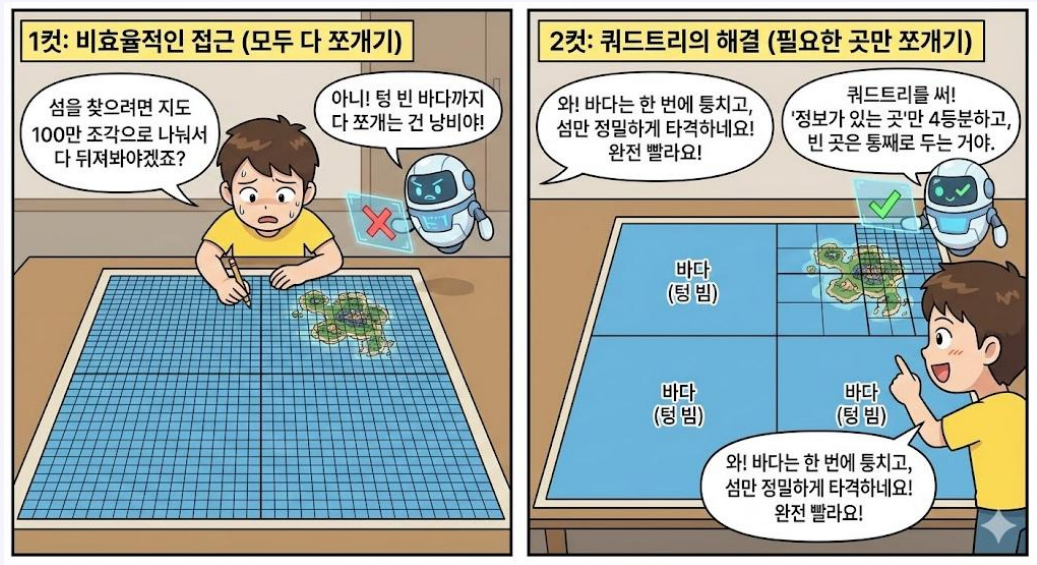
전체 영역을 확인한다.

② 기저 사례

데이터가 '단일 속성'(모두 같은 색이거나 텅 비어 있음)이면 더 이상 나누지 않고 하나의 노드로 표현한다.

③ 재귀 단계

'섞여 있다면' 영역을 정확히 4등분(좌상, 우상, 좌하, 우하)하여 각 부분에 대해 ①을 다시 수행한다.



이론에 그치지 않는다

게임의 물리 엔진(충돌 처리), 포토샵 같은 이미지 소프트웨어(압축)의 핵심 원리다.

응용 ① 흑백 이미지 압축

0(흰색)과 1(검은색)로 이루어진 이미지를 생각해보자. 넓은 영역이 모두 같은 색이라면, 픽셀 하나하나를 저장할 필요 없이 "이 영역 전체는 흰색"이라고 한 글자로 적으면 된다.

검사 (check)

현재 영역 (x, y)부터 size 만큼의 공간이 모두 같은 색(0 또는 1)인지 확인한다.

정복 (conquer)

모두 0이면 압축 결과에 0을, 모두 1이면 1을 기록하고 종료한다. 기저 사례다.

분할 (divide)

색이 섞여 있다면 괄호 '(' 를 기록하고, 영역을 4등분하여 좌상 → 우상 → 좌하 → 우하 순서로 재귀 호출한다.

압축의 정체

8×8 = 64개의 픽셀이 "(01((0001)000)0)" 라는 짧은 문자열로 바뀐다. 균일한 영역이 많을수록 압축률이 높아진다.

쿼드 트리 알고리즘 (의사코드)

quadtree_1.pseudo

```
// 입력: 2차원 데이터 I, 영역 좌표 (x, y), 영역 크기 size
// 출력: 해당 영역을 표현하는 트리 노드
ConstructQuadTree(I, x, y, size):

    // ① 현재 영역이 모두 같은 값(색)인지 검사
    initialValue <- I[x][y]
    isUniform <- true
    for i <- x to x + size - 1 do
        for j <- y to y + size - 1 do
            if I[i][j] != initialValue then
                isUniform <- false

    // ② 기저 사례: 균일하면 리프 노드 반환
    if isUniform then
        return CreateLeafNode(initialValue)
```

quadtree_2.pseudo

```
    // ③ 재귀 단계: 섞여 있다면 4등분
    else
        half <- size / 2
        root <- CreateInternalNode()

        root.topLeft     <- ConstructQuadTree(I, x,      y,      half)
        root.topRight    <- ConstructQuadTree(I, x,      y+half, half)
        root.bottomLeft  <- ConstructQuadTree(I, x+half, y,      half)
        root.bottomRight <- ConstructQuadTree(I, x+half, y+half, half)

        return root
```

$$\text{트리 깊이} \le \log_2(\text{size})$$

노드 하나가 곧 한 영역

리프 노드는 균일한 영역을, 내부 노드는 4개의 자식으로 쪼개진 영역을 뜻한다. 트리의 깊이는 최대 $\log_2(\text{size})$이다.

파이썬 구현 — 8×8 이미지 압축

quadtree_compress.py

```python
def quadtree_compress(x, y, n, image):

    # 1. Check: 현재 영역이 모두 같은 색인지 검사
    color = image[x][y]
    is_uniform = True
    for i in range(x, x + n):
        for j in range(y, y + n):
            if image[i][j] != color:
                is_uniform = False
                break
        if not is_uniform:
            break

    # 2. Conquer(Base Case): 모두 같은 색이면 색상 반환
    if is_uniform:
        return str(color)

    # 3. Divide: 섞여 있다면 4등분하여 재귀 호출
    half = n // 2
    result = "("
    result += quadtree_compress(x, y, half, image)
    result += quadtree_compress(x, y + half, half, image)
    result += quadtree_compress(x + half, y, half, image)
    result += quadtree_compress(x + half, y + half, half, image)
    result += ")"
    return result
```

example.py

```python
example_image = [
    [0,0,0,0, 1,1,1,1],
    [0,0,0,0, 1,1,1,1],
    [0,0,0,0, 1,1,1,1],
    [0,0,0,0, 1,1,1,1],
    [0,0,0,0, 0,0,0,0],
    [0,1,0,0, 0,0,0,0],   # 섞임!
    [0,0,0,0, 0,0,0,0],
    [0,0,0,0, 0,0,0,0],
]
print(quadtree_compress(0, 0, 8, example_image))
```

OUTPUT

```
(01((0001)000)0)
```

✓ 픽셀 64개 → 문자 16개

응용 ② 게임 개발과 충돌 감지

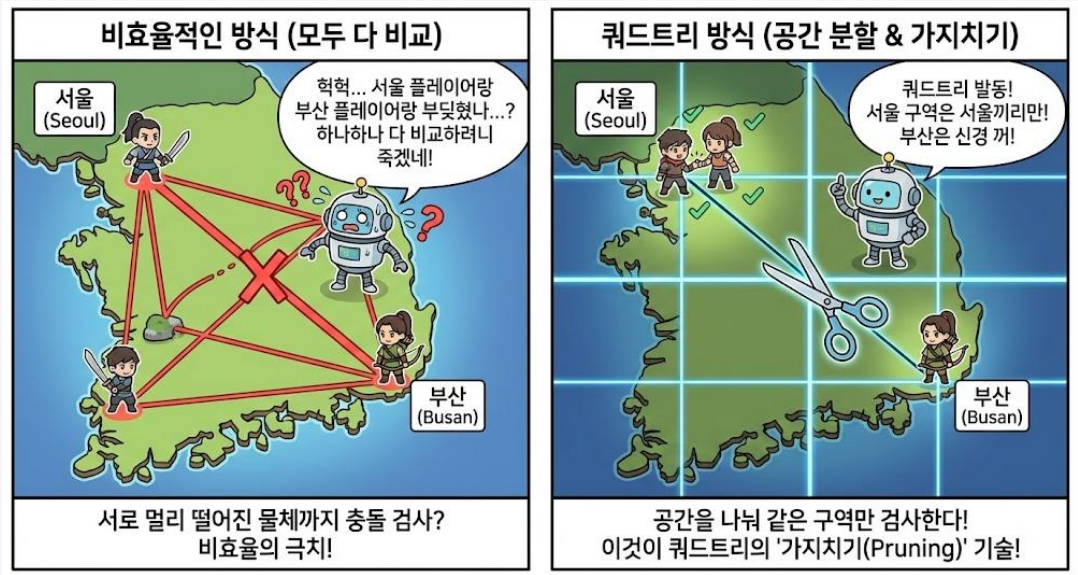
화면 속에는 수십 명의 캐릭터, 수백 발의 총알, 수천 개의 지형지물이 움직인다. 게임 엔진은 이 객체들이 서로 부딪혔는지 판별하는 충돌 감지(collision detection)를 매 프레임(약 0.016초)마다 수행해야 한다.

무식한 접근 — 브루트 포스

$$(n-1) + (n-2) + \cdots + 1 \approx n^2/2$$

객체 1,000개 → 매 프레임 약 50만 번 검사

1. **공간 분할** — 전체 맵을 4개의 사분면으로 나눈다. 한 구역에 오브젝트가 너무 많으면 그 구역을 다시 4개로 쪼갠다.
2. **객체 등록** — 모든 오브젝트는 자신의 위치에 따라 트리의 특정 노드(구역)에 등록된다.
3. **지역적 검사** — 충돌 검사를 할 때, 같은 노드(구역)에 속한 물체들끼리만 비교를 수행한다.



"서로 멀리 떨어져 있는 물체끼리는 충돌 검사를 할 필요가 없다"

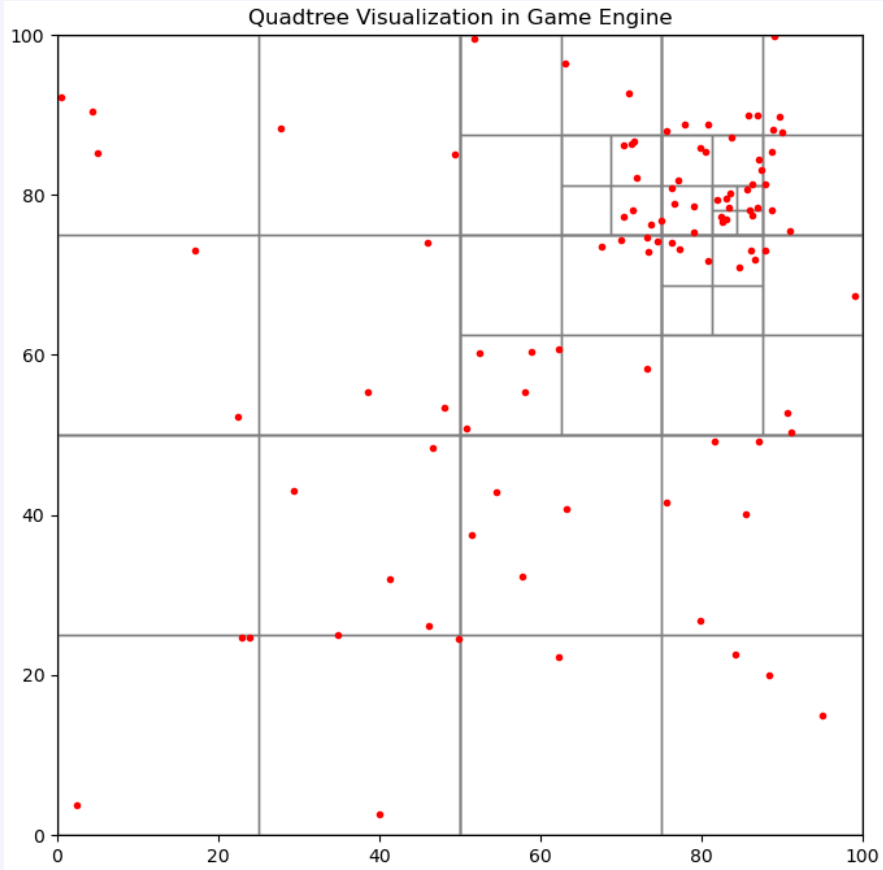
성능의 혁신과 LOD (Level of Detail)

쿼드 트리는 충돌 처리뿐만 아니라 렌더링 최적화 기술인 LOD(Level of Detail)에도 필수적이다. 플레이어의 시야(카메라)가 위치한 노드는 아주 작은 격자까지 분할하여 고화질 텍스처로 상세하게 그려내고, 시야에서 먼 노드는 더 이상 분할하지 않고 큰 격자 상태로 두어 저해상도로 그려낸다.

가까운 영역 깊게 분할 → 고해상도 렌더링

먼 영역 얕게 분할 → 저해상도 렌더링

결과 같은 프레임 예산으로 훨씬 넓은 세계를 그린다



4.6

마스터 정리

4.6

Master Theorem

분할 정복 점화식을 재귀 트리 없이 즉시 푸는 강력한 도구

§4.6 마스터 정리

점화식을 푸는 만능 도구

점화식을 풀기 위해 매번 재귀 트리를 그리거나 수학적 귀납법을 동원하는 것은 비효율적이다. 여기서 알고리즘 분석의 강력한 도구인 마스터 정리(Master Theorem)가 등장한다. 마스터 정리는 다음 형태의 점화식을 분석할 때 사용한다.

$$T(n) = a \cdot T(n/b) + f(n)$$

a

재귀 호출의 개수

문제를 몇 조각으로 나누어 각각 다시 푸는가

n / b

각 호출의 문제 크기

한 번 나눌 때마다 문제가 1/b 로 줄어든다

$f(n)$

분할·결합 비용

문제를 나누고 결과를 합치는 데 드는 추가 비용

§4.6 마스터 정리

마스터 정리의 세 가지 경우

핵심은 분할·결합 비용 f(n)을 기준값 n^(log_b a) 과 비교하는 것이다. 어느 쪽이 더 무거운지에 따라 답이 결정된다.

1

f(n)이 더 작다

`f(n) = O(n^(log_b a − ε))`

하위 문제들이 대부분의 시간을 쓴다.

$$T(n) = \Theta(n^{\log_b a})$$

2

둘이 비슷하다

`f(n) = Θ(n^(log_b a))`

분할·결합과 하위 문제가 균형을 이룬다.

$$T(n) = \Theta(n^{\log_b a} \cdot \log n)$$

3

f(n)이 더 크다

`f(n) = Ω(n^(log_b a + ε))`

쪼개고 합치는 데 시간이 더 걸린다.

$$T(n) = \Theta(f(n))$$

이 장에서 다룬 합병 정렬·최대 구간 합·최근접 쌍은 모두 a = 2, b = 2, f(n) = O(n) 이므로 Case 2에 해당한다.

적용 ① 이진 탐색

이진 탐색은 한 번 호출할 때마다 문제의 크기가 절반이 되고, 절반 지점(mid)을 계산하는 나눗셈 1번이 필요하다.

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

`a = 1`	재귀 호출 1회
`b = 2`	문제 크기를 절반으로
`f(n) = O(1)`	mid 계산 시간

마스터 정리 적용

기준값 계산

$n^{\log_2 1} = n^0 = 1$

f(n)과 비교

$f(n) = \Theta(1) = \Theta(n^0)$

판정

Case 2 (둘이 같은 정도)

$$T(n) = \Theta(\log n)$$

적용 ② 합병 정렬

합병 정렬은 문제를 2개의 부분 문제로 나누고, 각 부분 문제의 크기는 원본의 1/2이다. 나누고 합치는 데 걸리는 비용은 입력 크기 n에 비례한다.

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

`a = 2`	2개의 부분 문제로 나눈다
`b = 2`	각 부분 문제 크기는 1/2
`f(n) = cn`	병합 비용은 n에 비례 (c는 상수)

마스터 정리 적용

기준값 계산

$n^{\log_2 2} = n^1 = n$

f(n)과 비교

$f(n) = \Theta(n)$

판정

Case 2 (둘이 같은 정도)

$$T(n) = \Theta(n \log n)$$

§4.6 마스터 정리

한눈에 보는 점화식 정리

지금까지 만난 분할 정복 알고리즘들의 점화식을 마스터 정리로 정리하면 다음과 같다.

알고리즘	점화식	a	b	f(n)	경우	시간 복잡도
이진 탐색	$T(n) = T(n/2) + O(1)$	1	2	$O(1)$	Case 2	$O(\log n)$
합병 정렬	$T(n) = 2T(n/2) + O(n)$	2	2	$O(n)$	Case 2	$O(n \log n)$
최대 구간 합	$T(n) = 2T(n/2) + O(n)$	2	2	$O(n)$	Case 2	$O(n \log n)$
최근접 쌍	$T(n) = 2T(n/2) + O(n)$	2	2	$O(n)$	Case 2	$O(n \log n)$
카라츠바 (§4.7)	$T(n) = 3T(n/2) + O(n)$	3	2	$O(n)$	Case 1	$O(n^{1.585})$
스트라센 (§4.8)	$T(n) = 7T(n/2) + O(n^2)$	7	2	$O(n^2)$	Case 1	$O(n^{2.807})$

재귀 호출 횟수 a를 줄이는 것이 가장 강력한 최적화다 — 다음 두 절의 주제다.

4.7

카라츠바 알고리즘

4.7

Karatsuba Algorithm (1960)
큰 수의 곱셈에서 곱셈 4번을 3번으로 — 2,300년 만의 개선

문제의 분할 (divide)

아주 큰 두 수 X와 Y가 있다. 분할 정복을 적용하기 위해 이 수를 절반으로 쪼개서 표현할 수 있다. 예를 들어 X = 1234라면, 앞부분 12와 뒷부분 34로 나눈다. 일반화하면 N자리 수 X와 Y는 다음과 같이 표현된다 (m = N/2).

$$X = x_1 \cdot 10^m + x_0$$

$$Y = y_1 \cdot 10^m + y_0$$

예시: X = 1234, m = 2

$$1234 = 12 \cdot 10^2 + 34$$

이제 두 수의 곱을 전개해 보자.

$$X \cdot Y = (x_1 y_1) \cdot 10^{2m} + (x_1 y_0 + x_0 y_1) \cdot 10^m + (x_0 y_0)$$

곱셈이 x_1y_1, x_1y_0, x_0y_1, x_0y_0 로 총 4번 필요하다. $T(n) = 4T(n/2) + O(n) = O(n^2)$ — 개선이 없다!

카라츠바의 마법 — 대수적 트릭 (conquer)

1960년, 23세의 아나톨리 카라츠바는 곱셈을 3번만 하고도 원하는 값을 구해내는 대수적 트릭을 찾아냈다. 우리가 구해야 할 핵심 계수는 세 가지다.

A $= x_1 \cdot y_1$

B $= x_0 \cdot y_0$

C $= x_1 y_0 + x_0 y_1$

C를 곱셈 없이 구하는 법

$(x_1 + x_0)(y_1 + y_0)$ 를 한 번 곱해 보면 $x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0 = A + C + B$ 가 된다. 여기서 A와 B를 빼면 C가 남는다!

3번의 곱셈

$$M_1 = x_1 \cdot y_1$$

$$M_2 = x_0 \cdot y_0$$

$$M_3 = (x_1+x_0)(y_1+y_0)$$

결합 (combine)

$$C = M_3 - M_1 - M_2$$

$$X \cdot Y = M_1 \cdot 10^{2m} + C \cdot 10^{m} + M_2$$

카라츠바 알고리즘 (의사코드)

karatsuba.pseudo

```
// 입력: 두 개의 큰 정수 X, Y      // 출력: X × Y
KaratsubaMultiply(X, Y):

    // 1. 기저 사례: 한 자리 수라면 직접 곱함
    n <- max(length of X, length of Y)
    if n < 2 then
        return X * Y

    // 2. 분할(Divide): 숫자를 절반 크기(m)로 자름
    m <- n / 2
    x_high, x_low <- Split(X, m)
    y_high, y_low <- Split(Y, m)

    // 3. 정복(Conquer): 재귀 호출 3번
    z2 <- KaratsubaMultiply(x_high, y_high)
    z0 <- KaratsubaMultiply(x_low,  y_low)
    z1_raw <- KaratsubaMultiply(x_high + x_low, y_high + y_low)
    z1 <- z1_raw - z2 - z0

    // 4. 결합(Combine): 자릿수를 맞추어 더함
    return z2 * 10^(2m) + z1 * 10^m + z0
```

재귀 호출이 3번뿐

4번이 아니라 3번. 덧셈과 뺄셈이 몇 번 늘어나지만, 이들은 O(n)에 불과하다.

실무에서의 임계값

작은 수에서는 재귀 오버헤드가 더 크다. 보통 임계값을 10~100 자리 정도로 잡는다.

$$T(n) = 3T(n/2) + O(n)$$

파이썬 구현

karatsuba.py

```python
def karatsuba(x, y):
    # Base Case: 한 자리 수면 단순 곱셈
    if x < 10 or y < 10:
        return x * y

    n = max(len(str(x)), len(str(y)))
    m = n // 2

    # Divide: 10**m 을 이용해 절반으로 쪼갠다
    x_high, x_low = x // (10 ** m), x % (10 ** m)
    y_high, y_low = y // (10 ** m), y % (10 ** m)

    # Conquer: 재귀적으로 3번의 곱셈 수행
    a = karatsuba(x_high, y_high)
    b = karatsuba(x_low, y_low)
    sum_prod = karatsuba(x_high + x_low, y_high + y_low)

    # Combine: 중간항 = sum_prod - a - b
    mid = sum_prod - a - b
    return a * (10 ** (2 * m)) + mid * (10 ** m) + b
```

run.py

```python
num1 = 12345678901234567890
num2 = 98765432109876543210
print(karatsuba(num1, num2))   # 일반 곱셈 num1 * num2 와 비교
```

10 ** m 으로 자르기

몫(//)이 앞부분, 나머지(%)가 뒷부분이 된다.

OUTPUT

```
1219326311370217952
237463801111263526900
```

일반 곱셈 결과와 정확히 일치

RSA 암호의 핵심

수천 자리 정수 곱셈을 실용적으로 만든 알고리즘이다.

복잡도 분석 — 마스터 정리 적용

÷ 문제를 절반으로 나누었다 — `b = 2`

✕ 가장 비싼 연산인 곱셈(재귀 호출)을 4번에서 3번으로 줄였다 — `a = 3`

+ 덧셈·뺄셈·시프트 연산은 자릿수 n에 비례한다 — `f(n) = O(n)`

$$T(n) = 3T(n/2) + O(n)$$

$$\Rightarrow \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$$

n = 10,000 자리 비교

초등학교 곱셈

O(n²)

1억 회

카라츠바

O(n^1.585)

약 41만 회

약 240배 빠르다

자릿수가 커질수록 격차는 벌어진다.

4.8

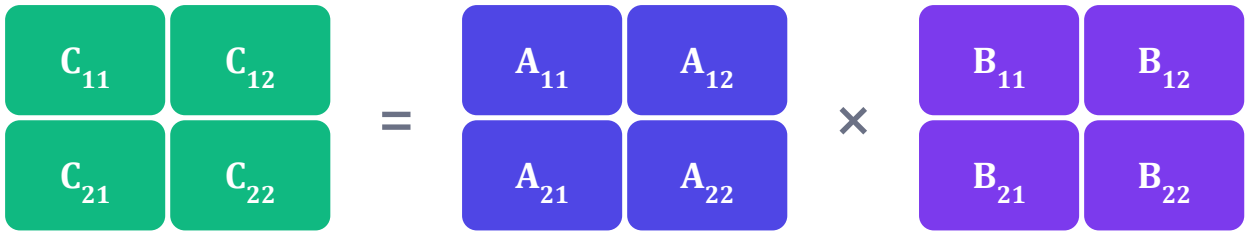
스트라센 알고리즘

4.8

Strassen Algorithm (1969)
행렬 곱셈에서 곱셈 8번을 7번으로

분할 정복의 시도 — 블록 행렬

두 개의 n × n 행렬을 곱하는 표준 알고리즘은 세 겹의 반복문으로 $O(n^3)$이 걸린다. 카라츠바 알고리즘에서 숫자를 절반으로 쪼갰듯이, 행렬도 4등분하여 블록 행렬(block matrix) 형태로 생각해보자.



각 결과 블록은 두 번의 블록 곱셈을 필요로 한다 — 모두 합쳐 8번

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \qquad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

아무런 개선이 없다 — $T(n) = 8T(n/2) + O(n^2) \Rightarrow \Theta(n^3)$

블록 곱셈이 8번, 블록 덧셈이 $O(n^2)$. 마스터 정리 Case 1에 의해 $n^{\log_2 8} = n^3$ — 원래 알고리즘과 똑같다.

스트라센의 발견 — "8번이 아니라 7번이면 된다"

1969년, 독일의 수학자 폴커 스트라센(Volker Strassen)은 이 견고해 보이는 벽을 허무는 방법을 찾아냈다. 카라츠바가 그랬던 것처럼, 행렬의 덧셈과 뺄셈을 조금 복잡하게 조합하면 곱셈 횟수를 8번에서 7번으로 줄일 수 있음을 발견했다.

7번의 블록 곱셈

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M_2 = (A_{21} + A_{22})\, B_{11}$$

$$M_3 = A_{11}(B_{12} - B_{22})$$

$$M_4 = A_{22}(B_{21} - B_{11})$$

$$M_5 = (A_{11} + A_{12})\, B_{22}$$

$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

결과 블록 조립 (combine)

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

$$C_{12} = M_3 + M_5$$

$$C_{21} = M_2 + M_4$$

$$C_{22} = M_1 - M_2 + M_3 + M_6$$

곱셈 1번을 덧셈 여러 번과 맞바꾼다

덧셈·뺄셈은 $O(n^2)$에 불과하지만, 곱셈은 재귀 호출이다. 재귀 호출 횟수 a를 줄이는 것이 압도적으로 이득이다.

복잡도 분석과 두 알고리즘의 공통점

$$T(n) = 7T(n/2) + O(n^2) \quad \Rightarrow \quad \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$$

	카라츠바 (§4.7)	스트라센 (§4.8)
대상	큰 정수의 곱셈	n × n 행렬의 곱셈
소박한 분할	곱셈 4번	블록 곱셈 8번
개선 후	곱셈 3번	블록 곱셈 7번
점화식	T(n) = 3T(n/2) + O(n)	T(n) = 7T(n/2) + O(n²)
시간 복잡도	O(n^1.585)	O(n^2.807)
기존 방법	O(n²)	O(n³)

두 알고리즘의 공통 교훈 — 값싼 덧셈을 늘려서라도 비싼 재귀 호출 횟수 a 를 줄여라.

4.9

코딩 테스트

분할 정복으로 실전 문제 두 개를 직접 풀어본다

4.9

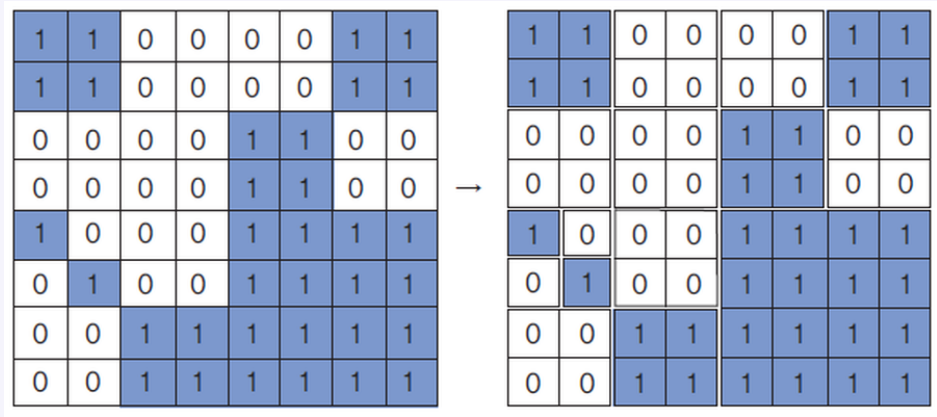
CT 4.1 색종이 만들기

색종이 만들기

여러 개의 정사각형 칸(N × N)으로 이루어진 종이가 있다. 각 칸은 하얀색 아니면 파란색이다. 전체 종이가 모두 같은 색이 아니면, 가로와 세로의 중앙을 잘라 똑같은 크기의 네 개의 N/2 × N/2 색종이로 나눈다. 이 과정을 잘라진 종이가 모두 같은 색이 되거나 한 칸이 될 때까지 반복한다.

최종적으로 하얀색 색종이와 파란색 색종이가 각각 몇 장인지 구하라.

- 쿼드 트리와 완전히 동일한 구조
- 검사(check) → 균일하면 카운트, 아니면 4등분
- 한 칸(n = 1)은 언제나 균일하므로 재귀는 반드시 끝난다



입력 (N = 8)

```
1 1 0 0 0 0 1 1   …   0 0 1 1 1 1 1 1
```

출력

하얀색 9장, 파란색 7장

풀이 — 재귀적으로 4등분하며 세기

solve.py

```python
import sys
sys.setrecursionlimit(10 ** 6)

def solve(x, y, n):
    global white_cnt, blue_cnt

    # 1. Check: 현재 영역의 첫 번째 색상 기준
    color = paper[x][y]
    for i in range(x, x + n):
        for j in range(y, y + n):
            if paper[i][j] != color:
                # 2. Divide: 4등분하여 재귀 호출
                nn = n // 2
                solve(x,      y,      nn)   # 좌상
                solve(x,      y + nn, nn)   # 우상
                solve(x + nn, y,      nn)   # 좌하
                solve(x + nn, y + nn, nn)   # 우하
                return

    # 3. Base Case: 반복문을 통과했다면 모두 같은 색
    if color == 0:
        white_cnt += 1
    else:
        blue_cnt += 1
```

run.py

```python
N = 8
white_cnt = blue_cnt = 0
solve(0, 0, N)
print(f"하얀색 색종이: {white_cnt}")
print(f"파란색 색종이: {blue_cnt}")
```

OUTPUT

```
하얀색 색종이: 9
파란색 색종이: 7
```

return 위치가 핵심

다른 색을 만나는 즉시 4등분 후 return 해야 한다. 그러지 않으면 아래의 카운트 코드까지 실행되어 잘못된 답이 나온다.

CT 4.2 거듭제곱 나머지

곱셈 — A^B mod C 를 눈 깜짝할 사이에에

세 개의 자연수 A, B, C가 주어진다. A를 B번 곱한 수, 즉 A^B를 C로 나눈 나머지를 구하라. 문제는 A와 B가 최대 2,147,483,647(약 21억)에 달하는 매우 큰 수라는 점이다. 단순한 반복으로는 21억 번의 곱셈이 필요하다.

지수를 절반으로

A^B = A^(B/2) × A^(B/2). 절반만 계산하면 나머지 절반은 공짜다.

B가 홀수라면

A^B = A^(B/2) × A^(B/2) × A. 남는 A 하나를 곱해준다.

모듈러 성질 활용

(a × b) mod c = ((a mod c) × (b mod c)) mod c. 매 단계 나머지를 취해 수가 커지지 않게 한다.

$$T(n) = T(n/2) + O(1) \quad \Rightarrow \quad \Theta(\log B)$$

예제

입력

```
10 11 12
```

출력

```
4
```

21억 번의 곱셈이
약 31번으로 줄어든다.
($\log_2$ 2,147,483,647 ≈ 31)

CT 4.2 거듭제곱 나머지

풀이 — 분할 정복 거듭제곱 (fast power)

power.py

```python
def power(a, b, c):
    # Base Case: 지수가 0이면 결과는 1
    if b == 0:
        return 1

    # Divide & Conquer: 지수를 절반으로 나눈다
    half = power(a, b // 2, c)

    # Combine: 절반의 결과를 제곱한다
    result = (half * half) % c

    # 지수가 홀수라면 a를 한 번 더 곱해준다
    if b % 2 == 1:
        result = (result * a) % c

    return result
```

run.py

```python
A, B, C = map(int, "10 11 12".split())
print(power(A, B, C))
```

OUTPUT 4

재귀 호출 추적

power(10, 11, 12)	b = 11 (홀수)
power(10, 5, 12)	b = 5 (홀수)
power(10, 2, 12)	b = 2 (짝수)
power(10, 1, 12)	b = 1 (홀수)
power(10, 0, 12)	return 1

깊이는 $\log_2 B$

B = 21억이어도 재귀 깊이는 31에 불과하다.

SUMMARY

4장 정리

분할 정복의 3단계

분할(divide) → 정복(conquer) → 결합(combine). 기저 사례에 도달할 때까지 재귀적으로 나눈다.

결합이 알고리즘의 핵심

합병 정렬은 나누기가 쉽고 합치기가 어렵다. 반대로 퀵 정렬은 나누기가 어렵고 합치기가 쉽다.

기하 문제에도 통한다

최대 구간 합과 최근접 쌍 모두 "경계를 걸친 경우"만 따로 처리하면 O(n log n)이 된다.

공간도 분할한다

쿼드 트리는 평면을 1/4씩 재귀 분할한다. 이미지 압축, 충돌 감지, LOD의 기반이다.

마스터 정리

T(n) = aT(n/b) + f(n) 꼴이면 f(n)과 n^(log_b a)의 크기만 비교하면 답이 나온다.

재귀 호출 횟수 a를 줄여라

카라츠바(4→3)와 스트라센(8→7)은 값싼 덧셈으로 비싼 곱셈을 대체했다.

CHAPTER 04

Q & A

질문을 나누어 봅시다.
큰 질문도 작게 쪼개면 답할 수 있습니다.