Chapter 9: Coding

왜 코딩인가?

데이터 사이언티스트에게 코딩은 **필수 역량**

데이터 전처리, API 연동, 파이프라인 구축 — 모두 코드로 구현됨

특히 소규모 기업에서는 분석부터 프로덕션까지 직접 담당

코딩 인터뷰 개요

- **형식**: 30~45분, HackerRank/CoderPad 또는 화이트보드
- **출제 범위**: 자료구조 조작(리스트, 트리, 그래프)과 알고리즘(재귀, DP)
- **평가 기준**: 풀이의 정확성 + Big-O 시간/공간 복잡도 분석 능력
- **핵심**: CS 기초가 소프트웨어/데이터 엔지니어와 협업 시 큰 강점이 됨

코딩 문제 접근 4단계

1. 문제 이해

- 문제를 되짚어 확인
- 입력 형식/범위 명확화
- 예시 입출력 직접 검증

2. 풀이 구상

- Brute-force 먼저 설명
- 비효율 원인 파악
- 최적화 방향 제시

3. 코딩

- 코드 작성하며 설명
- 의사코드 지양, 실제 코드 작성
- 변수명 명확하게

4. 검증

- 엣지 케이스 확인
- 테스트 케이스 실행
- 시간/공간 복잡도 분석

Space & Time Complexity

Big-O 표기법

Big-O 란?

- 알고리즘의 **최악의 경우**(worst-case) 성능 상한을 표현하는 표기법
- 입력 크기 n이 무한대로 갈 때 런타임/메모리 사용량의 **증가율**을 분석함
- 공간 복잡도도 동일한 방식으로 분석 — 예: 배열 복사 $O(N)$, 인접행렬 $O(N^2)$
- 실무에서도 중요: GPT-3 학습에 $12M 연산 비용이 들었듯, 효율 분석이 곧 비용 관리

Big-O 복잡도 클래스

복잡도	명칭	예시
$O(1)$	상수 시간	배열 인덱스 접근
$O(\log N)$	로그 시간	이진 탐색
$O(N)$	선형 시간	배열 순회(for문)
$O(N \log N)$	로그선형 시간	머지소트
$O(N^2)$	이차 시간	이중 for문(모든 쌍)
$O(2^N)$	지수 시간	N자리 이진수 재귀 생성
$O(N!)$	팩토리얼 시간	배열의 모든 순열 생성

이진 탐색 — 복잡도 분석 예시

정렬된 배열에서 특정 값을 찾는 알고리즘. 매 반복마다 탐색 범위를 절반으로 줄임.

```
def binary_search(a, k):
    lo, hi = 0, len(a) - 1
    best = lo
    while lo <= hi:
        mid = lo + (hi - lo) // 2
        if a[mid] < k:
            lo = mid + 1
        elif a[mid] > k:
            hi = mid - 1
        else:
            best = mid
            break
        if abs(a[mid] - k) < abs(a[best] - k):
            best = mid
    return best
```

- 시간: $O(\log N)$ — 반복마다 절반 축소

ML 알고리즘 복잡도 분석

Naive Bayes

- 학습: n개 데이터 $\times$ d개 피처 $\times$ k개 클래스
- **학습 시간**: $O(nkd)$ — 빈도 카운팅 기반
- **공간**: $O(kd)$ — 클래스별 확률 저장

Logistic Regression

- 가중치 벡터 β: $d \times 1$
- **학습 시간**: $O(nd)$ — 각 데이터에 가중치 곱셈
- **공간**: $O(d)$ — 가중치 벡터만 저장

Data Structures

자료구조

자료구조별 시간 복잡도 비교

자료구조	접근	탐색	삽입	삭제	최악 공간
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Stack	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Queue	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Hash Map	N/A	$O(1)$	$O(1)$	$O(1)$	$O(n)$
BST	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

> 평균(Average) 기준. 최악(Worst)은 Hash Map이 $O(n)$, BST도 $O(n)$까지 퇴화 가능.

Array (배열)

- **구조**: 메모리에 연속 저장되는 동일 타입 요소들의 시퀀스
- **강점**: 인덱스 접근 $O(1)$ — 가장 빠른 랜덤 액세스
- **약점**: 탐색/삭제 $O(N)$ (정렬 안 된 경우)
- **면접 패턴**: brute-force $O(n)$ 공간 → 배열 자체를 활용해 $O(1)$로 최적화
- **주의**: off-by-1 에러 — 배열 끝을 넘어 읽는 실수가 빈번함
- **DS 연결**: 벡터 = 1D 배열, 피처 행렬 X = 2D 배열 ($n \times d$)

Array — Python 리스트 컴프리헨션

Python 면접에서 자주 출제되는 **리스트 컴프리헨션(list comprehension)**:

```
# 첫 10개 짝수
a = [x * 2 for x in range(1, 11)]
# [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

# 누적합
c = [sum(a[:x]) for x in range(len(a) + 1)]
# [0, 2, 6, 12, 20, 30, 42, 56, 72, 90, 110]
```

- 표현력 높고 코드베이스에서 자주 사용됨
- 복잡한 로직은 가독성을 해칠 수 있으므로 적절히 사용

Linked List (연결 리스트)

- **구조**: 데이터와 포인터로 구성된 **노드(Node)** 의 연결 체인
- **종류**: 단일 연결, 이중 연결(doubly linked), 순환(circular)
- **강점**: head/tail에 삽입·삭제 $O(1)$
- **약점**: 인덱싱·탐색 $O(N)$ — 순차 접근만 가능
- **면접 패턴**: $O(N)$ 공간 → 기존 노드 재활용으로 $O(1)$ 최적화

Linked List — 역순 뒤집기

```
class Node:
    def __init__(self, val):
        self.val = val
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def reverse(self):
        prev = None
        curr = self.head
        while curr:
            next = curr.next    # 다음 노드 저장
            curr.next = prev    # 방향 뒤집기
            prev = curr         # prev 전진
            curr = next         # curr 전진
        self.head = prev
```

Stack & Queue

Stack (스택)

- **LIFO**: Last-In, First-Out
- push(추가)와 pop(제거)
- 재귀 연산에 주로 사용
- 배열 또는 연결 리스트로 구현

Queue (큐)

- **FIFO**: First-In, First-Out
- enqueue(추가)와 dequeue(제거)
- 반복적(iterative) 프로세스에 사용
- BFS 탐색의 핵심 자료구조

Stack 활용 — 괄호 균형 검사

```
def check_balance(s):
    left_side = ["(", "{", "["]
    right_side = [")", "}", "]"]
    stack = []
    for i in s:
        if i in left_side:
            stack.append(i)          # 여는 괄호 → push
        elif i in right_side:
            pos = right_side.index(i)
            if len(stack) == 0 or (left_side[pos] != stack[-1]):
                return False         # 짝 불일치
            else:
                stack.pop()          # 짝 일치 → pop
    return len(stack) == 0           # 모두 소진 → 균형
```

- `"({}(){}))"` → `True` / `"{}(})"` → `False`
- 핵심: 여는 괄호는 push, 닫는 괄호는 최상단과 비교 후 pop

Hash Map (해시 맵)

- **구조**: 키-값 쌍 저장. 해시 함수로 키 → 버킷 인덱스 매핑
- Python: `dict` 가 해시 맵 기능을 제공함
- **평균 성능**: 조회·삽입·삭제 모두 $O(1)$
- **충돌(collision)**: 서로 다른 키가 같은 인덱스를 가질 때 발생
- **실무 응용**: 데이터베이스 샤딩(sharding) — 해시 함수로 데이터가 어떤 DB에 저장될지 결정

Hash Map — Two Sum 문제

배열에서 합이 target인 두 원소가 존재하는지 판별:

```
def check_sum(a, target):
    d = {}                          # 해시 맵 생성
    for i in a:
        if (target - i) in d:       # 보수(complement) 존재 확인
            return True
        else:
            d[i] = i                # 현재 값 저장
    return False
```

- Brute-force: 이중 for문 $O(N^2)$
- Hash Map: 단일 for문 $O(N)$ — 보수를 $O(1)$에 조회
- 예: `[3, 1, 4, 2, 6, 9]`, target=11 → `True` (2+9=11)

Trees

트리

Tree 기본 개념

- **구조**: 루트 노드(root)와 자식 노드(children)의 계층 구조
- **이진 트리(binary tree)**: 각 노드에 최대 2개 자식(left, right)
- 탐색·삽입·삭제의 최악 런타임: $O(N)$

```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
```

- 면접 문제 대부분은 트리 순회(traversal) + 재귀 데이터 전달로 귀결됨

트리 순회 3가지

In-order (중위)

- **순서**: 왼쪽 → 루트 → 오른쪽
- BST에서 정렬된 순서를 얻을 때 사용
- `3, 12, 6, 4, 7, 9, 11, 5, 2, 8`

Pre-order (전위)

- **순서**: 루트 → 왼쪽 → 오른쪽
- 트리 복사, 직렬화에 활용
- `9, 12, 3, 4, 6, 7, 5, 11, 2, 8`

Post-order (후위)

- **순서**: 왼쪽 → 오른쪽 → 루트
- 트리 삭제, 후위 표기식에 활용
- `3, 6, 7, 4, 12, 11, 8, 2, 5, 9`

트리 순회 — In-order 코드

```
def inorder(node):
    if node is None:
        return []
    else:
        return inorder(node.left) + [node.val] + inorder(node.right)
```

- post/pre/in 은 **루트 처리 위치**를 가리킴
  - pre-order: 루트 **먼저** → 왼쪽 → 오른쪽
  - in-order: 왼쪽 → 루트 **중간** → 오른쪽
  - post-order: 왼쪽 → 오른쪽 → 루트 **마지막**
- Level-order(BFS 기반): `9, 12, 5, 3, 4, 11, 2, 6, 7, 8`

Binary Search Tree (BST)

- **핵심 성질**: 왼쪽 서브트리 $\leq$ 루트 $\leq$ 오른쪽 서브트리
- 균형(balanced) BST에서 탐색·삽입·삭제: $O(\log N)$
- 매 비교마다 탐색 범위를 **절반**으로 축소

탐색 예시 — 값 9 찾기:

1. 루트(8)와 비교 → 9 > 8 → 오른쪽으로 (절반 제거)
2. 노드(10)와 비교 → 9 < 10 → 왼쪽으로 (절반 제거)
3. 왼쪽 자식 없음 → 9는 트리에 없음

> **실무 연결**: B-tree(DB 인덱싱)는 BST의 일반화. 수백만 레코드도 깊이 4~5에서 조회 가능.

BST — 삽입 구현

```
class BST:
    def __init__(self, val):
        self.root = TreeNode(val)

    def insert(self, node, val):
        if node is not None:
            if val < node.val:
                if node.left is None:
                    node.left = TreeNode(val)
                else:
                    self.insert(node.left, val)
            else:
                if node.right is None:
                    node.right = TreeNode(val)
                else:
                    self.insert(node.right, val)
        else:
            self.root = TreeNode(val)
```

Heap (힙)

Max-Heap

- 부모 $\geq$ 자식 — 루트가 **최댓값**
- 최댓값 조회: $O(1)$
- 삽입/삭제: $O(\log N)$ (heapify)
- 탐색: $O(N)$

Min-Heap

- 부모 $\leq$ 자식 — 루트가 **최솟값**
- 최솟값 조회: $O(1)$
- 최솟값/최댓값이 핵심이고 임의 조회가 불필요할 때 사용

Heap — Python heapq 활용

K개의 최대 원소 찾기:

```
import heapq

k = 5
a = [13, 5, 2, 6, 10, 9, 7, 4, 3]
heapq.heapify(a)          # 힙 생성
heapq.nlargest(k, a)      # K-largest 반환
```

- heapify: $O(N)$에 배열을 힙으로 변환
- nlargest/nsmallest: 정렬 없이 상위/하위 K개를 효율적으로 추출

대표 면접 문제:

- K개의 최대/최소 원소 찾기
- 스트림 데이터에서 현재 중앙값(median) 구하기
- 거의 정렬된 배열 정렬하기

Graphs

그래프

Graph 기본 개념

- **구성**: 노드(vertex) + 간선(edge)
- **방향 그래프(directed)**: 간선에 방향 있음
- **무방향 그래프(undirected)**: 간선에 방향 없음
- **DAG**: 방향 비순환 그래프 — Airflow, Spark, TensorFlow 등에서 연산 흐름 표현

그래프 표현 방식

Adjacency Matrix (인접 행렬)

- $N \times N$ 행렬, 간선 유무를 0/1로 표시
- 두 노드 이웃 확인: $O(1)$
- 공간: $O(N^2)$ — 희소 그래프에 비효율적
- 예: Facebook 20억 유저 → $2B \times 2B$ 행렬 필요

Adjacency List (인접 리스트)

- 각 노드의 이웃 목록을 저장
- 두 노드 이웃 확인: 최악 $O(N)$
- 공간: 간선 수에 비례 — 희소 그래프에 효율적
- 실무 대부분의 그래프에 적합

Graph 구현 — 인접 리스트 기반

```
class Vertex:
    def __init__(self, val):
        self.val = val
        self.neighbors = {}

    def add_to_neighbors(self, neighbor, w):
        self.neighbors[neighbor] = w

    def get_neighbors(self):
        return self.neighbors.keys()

class Graph:
    def __init__(self):
        self.vertices = {}

    def add_vertex(self, val):
        new_vertex = Vertex(val)
        self.vertices[val] = new_vertex
        return new_vertex

    def add_edge(self, u, v, weight):
        if u not in self.vertices:
            self.add_vertex(u)
        if v not in self.vertices:
            self.add_vertex(v)
        self.vertices[u].add_to_neighbors(
            self.vertices[v], weight)
```

실무에서의 그래프

- **PageRank**: 웹페이지(노드) + 하이퍼링크(간선). 많은 고품질 링크를 받은 페이지 = 높은 순위
- **Computational Graph**: Airflow, Spark, TensorFlow에서 DAG로 연산 흐름 표현
  - 노드 = 연산, 간선 = 데이터(텐서) 흐름
  - **병렬화**: 독립 연산을 동시 실행 가능
  - **이식성**: 언어에 무관한 연산 표현

BFS vs DFS

BFS (너비 우선 탐색)

- **방식**: 큐(Queue) 사용, 이웃 먼저 처리
- **최적**: 최단 경로(shortest path) 탐색
- **장점**: 해를 반드시 찾음, 재귀 갇힘 없음
- **단점**: 메모리 사용 많음
- 런타임: $O(E + V)$

DFS (깊이 우선 탐색)

- **방식**: 재귀/스택, 한 경로 끝까지 탐색
- **최적**: 사이클 탐지, 위상 정렬
- **장점**: 메모리 적음, 먼 노드 빠르게 도달
- **단점**: 재귀에 갇힐 수 있음
- 런타임: $O(E + V)$

BFS 구현

```
def bfs(graph, v):
    n = len(graph.vertices)
    visited = [False] * (n + 1)
    queue = []
    queue.append(v)
    visited[v] = True
    while queue:
        curr = queue.pop(0)          # 큐에서 꺼냄 (FIFO)
        for i in graph.get_vertex(curr):
            if visited[i.val] == False:
                queue.append(i.val)
                visited[i.val] = True
    return visited
```

- 시작 노드 → 큐에 추가 → 이웃 노드를 순서대로 방문

DFS 구현

```
def dfs_helper(graph, v, visited):
    visited.add(v)
    for neighbor in graph.get_vertex(v).get_neighbors():
        if neighbor.val not in visited:
            dfs_helper(graph, neighbor.val, visited)
    return visited

def dfs(graph, v):
    visited = set()
    return dfs_helper(graph, v, visited)
```

- 시작 노드 → 이웃 하나를 끝까지 파고든 뒤 → 되돌아와서 다음 이웃 탐색
- `visited` 집합(set)으로 중복 방문 방지

Algorithms

알고리즘

Recursion (재귀)

- **정의**: 자기 자신을 직접 또는 간접 호출하는 함수
- **구성 요소**: 재귀 케이스(recursive case) + 기저 케이스(base case)
- **주의**: 중복 호출 → 지수적 런타임, 스택 오버플로 가능

```
def fib(n):
    if n == 0:
        return 0
    if n == 1 or n == 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

- `fib(6)` 호출 시 `fib(2)`가 여러 번 중복 계산됨 → 비효율

재귀 vs 반복 — 피보나치

```python
# 반복(iterative) 방식
def fib(n):
    if n == 0: return 0
    if n == 1 or n == 2: return 1
    prev2 = 0    # fib(n-2)
    prev = 1     # fib(n-1)
    res = 0
    for i in range(1, n):
        res = prev2 + prev
        prev2 = prev
        prev = res
    return res
```

- **재귀**: 표현은 간결하지만 $O(N)$ 재귀 호출 → 메모리 부담
- **반복**: 변수 2개만으로 $O(1)$ 추가 공간, 스택 오버플로 없음

Greedy Algorithm (탐욕 알고리즘)

- **핵심**: 매 단계에서 **지역 최적**(locally optimal) 선택
- 문제를 더 작은 하위 문제로 축소해나감

```
def minCoins(k):
    coins = [1, 5, 10, 25]
    n = len(coins)
    res = []
    i = n - 1
    while i >= 0 and k >= 0:
        if k >= coins[i]:
            k -= coins[i]
            res.append(coins[i])
        else:
            i -= 1
    return res
```

- 67센트 → 25+25+10+5+1+1 = 6개 동전
- **ML 연결**: 의사결정 트리가 각 분할에서 정보 이득(info gain)을 탐욕적으로 최대화

Dynamic Programming

동적 프로그래밍

DP의 두 가지 조건

DP가 적용 가능하려면 두 조건이 모두 필요함:

1. **최적 부분 구조(Optimal Substructure)**: 문제를 하위 문제로 분할하여 최적 해를 구성할 수 있음
2. **중복 부분 문제(Overlapping Subproblems)**: 같은 하위 문제가 여러 번 반복 등장함

피보나치가 DP를 만족하는 이유:

- `fib(n) = fib(n-1) + fib(n-2)` → 최적 부분 구조
- `fib(n)`과 `fib(n-1)` 모두 `fib(n-2)`를 필요로 함 → 중복

Top-down vs Bottom-up

Top-down (메모이제이션)

- 재귀 + 해시 테이블로 캐싱
- 큰 문제에서 작은 문제로 분할
- 필요한 하위 문제만 계산

```
dp = [0] * 1000
def fib(n):
    if n == 0 or n == 1:
        return n
    dp[n] = fib(n-1) + fib(n-2)
    return dp[n]
```

Bottom-up (타뷸레이션)

- 반복문 + 배열로 캐싱
- 작은 문제부터 큰 문제로 적립
- 모든 하위 문제를 순서대로 계산

```
def fib(n):
    dp = [0] * (n + 1)
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

Greedy vs DP — 0/1 Knapsack 비교

항목	A	B	C
가치(Value)	3	1.2	2
무게(Weight)	4	2	3
가치/무게	0.75	0.6	0.67

무게 한도 W = 5:

- **Greedy**: 가치/무게 비율 최고인 A 선택 → 가치 **3**, 잔여 무게 1 (추가 불가)
- **DP**: B + C 선택 → 가치 **3.2**, 무게 5 (전역 최적)

> Greedy = 지역 최적, DP = **전역 최적** 보장

Knapsack DP 구현

```
def knapsackDP(values, weights, max_weight):
    n = len(values)
    dp = [[0 for x in range(max_weight + 1)]
          for x in range(n + 1)]
    for i in range(n + 1):
        for w in range(max_weight + 1):
            if weights[i-1] <= w:
                dp[i][w] = max(
                    values[i-1] + dp[i-1][w - weights[i-1]],
                    dp[i-1][w]
                )
            else:
                dp[i][w] = dp[i-1][w]
    return dp[n][max_weight]
```

- 2D 배열 `dp[i][w]` : i번째 아이템까지 고려, 무게 한도 w일 때의 최대 가치
- 각 아이템마다 **"가져간다 vs 안 가져간다"** 중 최대 선택

DP의 실무 응용 — 강화학습

- **강화학습(RL)**: 특정 상태에서 어떤 행동을 취해야 기대 보상을 최대화하는지 학습
- **벨만 방정식(Bellman Equations)**: DP의 핵심 원리를 활용
  - 전체 보상을 하위 문제들의 보상 조합으로 분해
  - 각 하위 문제의 최적 보상을 결합 → 전역 최적 정책 도출
- DP는 학술적 연습이 아니라 **실제 ML에서 광범위하게 사용**되는 기법

Sorting

정렬 알고리즘

정렬 알고리즘이 면접에서 중요한 이유

- 정렬 자체를 코딩하라는 문제는 드물지만, **중간 단계로 활용**됨
- 입력을 정렬하면 숨겨진 구조가 드러나 문제가 단순해짐
- Mergesort와 Quicksort — 두 가지 핵심 알고리즘을 이해해야 함

알고리즘	평균 시간	최악 시간	공간	안정성
Mergesort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	안정
Quicksort	$O(N \log N)$	$O(N^2)$	$O(\log N)$	불안정

Mergesort (합병 정렬)

Divide-and-Conquer 전략:

1. 배열을 계속 반으로 나눔 → 원소 1개(기저 사례)까지
2. 정렬된 부분 배열을 반복적으로 합병(merge)

```
def mergesort(a, low, high):
    if low >= high:
        return a
    mid = (low + high - 1) // 2
    mergesort(a, low, mid)
    mergesort(a, mid + 1, high)
    merge_helper(a, low, high, len(a) // 2)
    return a
```

- **시간**: $O(N \log N)$ — 분할 $\log N$ 단계 $\times$ 합병 N
- **공간**: $O(N \log N)$ — 보조 배열 필요

Mergesort — merge_helper

```
def merge_helper(a, low, high, mid):
    if len(a) == 1:
        return a
    i = j = k = 0
    left = a[:mid]
    right = a[mid:]
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            a[k] = left[i]
            i += 1
        else:
            a[k] = right[j]
            j += 1
        k += 1
    while i < len(left):        # 왼쪽 잔여
        a[k] = left[i]; i += 1; k += 1
    while j < len(right):       # 오른쪽 잔여
        a[k] = right[j]; j += 1; k += 1
    return a
```

Quicksort (퀵 정렬)

피벗(pivot) 기반 분할:

1. 임의의 피벗 선택
2. 피벗보다 작은 원소는 왼쪽, 큰 원소는 오른쪽으로 swap
3. 좌우 부분에 대해 재귀적으로 반복

```
def helper(a, low, high):
    i = low - 1
    pivot = a[high]
    for j in range(low, high):
        if a[j] <= pivot:
            i += 1
            a[i], a[j] = a[j], a[i]
    a[i+1], a[high] = a[high], a[i+1]
    return i + 1

def quicksort(a, low, high):
    if len(a) == 1: return a
```

Matrix Multiplication (행렬 곱셈)

모든 ML 알고리즘은 결국 **행렬 연산**으로 귀결됨.

```
def matrix_multiply(A, B):
    m = [[0 for _ in range(len(B[0]))]
         for _ in range(len(A))]
    for i in range(len(A)):
        for j in range(len(B[0])):
            for k in range(len(B)):
                m[i][j] += A[i][k] * B[k][j]
    return m
```

- **시간**: $O(N^3)$ — 3중 for문
- **최적화**: 부분 행렬 곱으로 분할 → 병렬/분산 처리 가능
- **면접 배경**: 금융업계에서 선형대수 구현 + 최적화 역량을 동시 평가

핵심 요약 — 자료구조

자료구조	핵심 특성	언제 사용?
Array	$O(1)$ 인덱스 접근	순차 데이터, 인덱스 기반 접근
Linked List	$O(1)$ 삽입/삭제	빈번한 삽입/삭제, 크기 가변
Stack/Queue	LIFO/FIFO	재귀 시뮬레이션, BFS
Hash Map	$O(1)$ 평균 조회	빈도 카운팅, 빠른 룩업
BST	$O(\log N)$ 탐색	정렬된 데이터, DB 인덱싱
Heap	$O(1)$ 최대/최소 접근	Top-K, 스트림 중앙값
Graph	관계 모델링	네트워크, 경로 탐색

핵심 요약 — 알고리즘

알고리즘	핵심 아이디어	시간 복잡도
Binary Search	절반씩 탐색 범위 축소	$O(\log N)$
Recursion	자기 호출 + 기저 사례	문제 의존
Greedy	매 단계 지역 최적 선택	문제 의존
DP	하위 문제 결과 캐싱	문제 의존
Mergesort	분할-정복, 안정 정렬	$O(N \log N)$
Quicksort	피벗 기반, 평균 최고	평균 $O(N \log N)$
BFS/DFS	그래프 순회	$O(E + V)$