

# 알고리즘 전체 분류 지도

## 자료구조

### 1. 자료구조

## 알고리즘

1. 탐색
2. 정렬
3. 완전탐색
4. 그리디
5. 투포인터
6. 슬라이딩윈도우
7. 이분탐색
8. DP
9. 그래프알고리즘
10. 트리알고리즘
11. 문자열알고리즘
12. 비트마스킹
13. 시뮬레이션 x

## 자료구조

- 코테에서는 “자료구조 선택”이 곧 풀이의 절반입니다.
- 배열/리스트/해시/큐/힙/덱은 거의 매일 씁니다.

### 배열 (Array)

- 사고력 포인트
  - 인덱스 접근 O(1)
  - 삽입/삭제 느림
  - 2차원 배열은 시뮬레이션/그래프(격자 BFS)에서 자주 등장

```
int[] arr = new int[5];  
  
arr[0] = 10;  
int x = arr[0];  
arr.length;
```

### ArrayList

```
List<Integer> list = new ArrayList<>();

list.add(10);
list.add(1, 20);
list.get(0);
list.set(0, 100);
list.remove(0);
list.size();
list.contains(10);
```

## Stack (코테 권장: Deque로 스택 구현)

- Stack 클래스도 되지만, 코테에서는 보통 ArrayDeque를 스택으로 씁니다(성능/관례).
- 사고력 포인트
  - 괄호/문자열 처리, DFS(재귀 대체), 되돌리기(Undo)

[Stack]

```
Stack<Integer> stack = new Stack<>();

stack.push(10);
stack.pop();
stack.peek();
stack.isEmpty();
stack.size();
```

[Deque]

```
Deque<Integer> stack = new ArrayDeque<>();

stack.push(10); // 삽입
int a = stack.pop(); // 제거
int top = stack.peek();
boolean empty = stack.isEmpty();
int size = stack.size();
```

## Queue

- 사고력 포인트
  - BFS, 레벨 탐색, “최소 횟수/최단 거리(가중치 없는 경우)”

```
Queue<Integer> queue = new LinkedList<>();

queue.offer(10);
queue.poll();
queue.peek();
```

```
queue.isEmpty();
queue.size();
```

## Deque

- 양쪽 삽입/삭제 가능 → 슬라이딩 윈도우(최댓값/최솟값)에서 자주 사용

```
Deque<Integer> deque = new LinkedList<>();

deque.addFirst(1);
deque.addLast(2);
deque.pollFirst();
deque.pollLast();
deque.peekFirst();
deque.peekLast();
```

## PriorityQueue (Heap)

- 사고력 포인트
  - “가장 작은/큰 값” 반복 추출
  - 다익스트라, 스케줄링(우선순위 처리)
- 

```
PriorityQueue<Integer> pq = new PriorityQueue<>();

pq.offer(10);
pq.poll();
pq.peek();
pq.size();
pq.isEmpty();
```

### [최대 힙]

```
PriorityQueue<Integer> pq =
    new PriorityQueue<>(Collections.reverseOrder());
```

## Heap (= 완전이진트리)

- 힙은 “완전 이진 트리” 형태를 유지하면서, 루트가 항상 최소/최대가 되도록 만든 자료구조입니다.
  - Min-Heap(부모  $\leq$  자식): 루트 = 최소값
  - Max-Heap(자식  $\leq$  부모): 루트 = 최대값
- 사용법 : 우선순위 큐랑 비슷(아니 거의 동일)

핵심: 전체 정렬 X, 대신 1등(최소/최대)만 O(1)에 확인, O(logN)에 뽑기/넣기

## [여기서 중요한 점]

형제끼리, 다른 서브트리끼리의 크기 관계는 보장 안 함  
→ 그래서 “전체 정렬”이 아니라 “루트만 1등 보장”

## HashMap

- 사고력 포인트
  - 빈도수(counting), 중복체크, 인덱싱(값→위치)

```
HashMap<String, Integer> map = new HashMap<>();

map.put("a", 1);
map.get("a");
map.remove("a");
map.containsKey("a");
map.containsValue(1);
map.keySet();
map.values();
map.size();
```

## HashSet

- 사고력 포인트
  - “존재 여부 O(1)” + 중복 제거

```
HashSet<Integer> set = new HashSet<>();

set.add(1);
set.remove(1);
set.contains(1);
set.size();
```

\*\*\* [여기부터는 알고리즘] \*\*\*

## 탐색

- 사고력(판단 기준)
  - “최단 거리/최소 횟수” → BFS
  - “모든 경로/연결요소/깊게” → DFS

## DFS (깊이 우선)

☞ 재귀 or 스택 사용

```
void dfs(int node) {
    visited[node] = true;
    for(int next : graph.get(node)) {
        if(!visited[next]) dfs(next);
    }
}
```

## BFS (너비 우선)

### ☞ 최단거리 문제에 자주 사용

```
Queue<Integer> queue = new LinkedList<>();
queue.offer(start);
```

## 정렬

- 정렬 + 그리디 / 투포인터 / 이분탐색 조합이 매우 자주 나옵니다.
- 사고력 포인트
  - “정렬 후 조건 체크” 형태가 코테 단골

```
Arrays.sort(arr);
Collections.sort(list);
```

## 완전탐색 / 백트래킹(재귀를 말함)

- 모든 경우의 수를 탐색합니다.
- ↗ N이 작을 때 사용 (보통 8~10 이하)
- ↗ 백트래킹은 “가지치기”가 핵심

완전탐색 = for문으로 모두 순회

### 추가[순열]

```
void permutation(int depth) {
    if(depth == N) return;
}
```

## 그리디 (탐욕법)

### “지금 당장 최선 선택”

[사용 조건] : 선택이 전체 최적을 보장할 때

예시: 동전 문제 (큰 동전부터 사용)

---

## 투포인터

- 두 개의 인덱스를 사용합니다.
- ↗ 정렬된 배열에서 자주 사용
- 사고력(판단 기준)
  - “정렬된 배열에서 두 값/두 구간을 좁혀가며 찾는다”
  - 합/차/조건 만족 구간 찾기

```
int left = 0;
int right = arr.length - 1;

while(left < right) {
    int sum = arr[left] + arr[right];
}
```

---

## 슬라이딩 윈도우

- 구간 합 문제
- ↗ 부분합 문제
- 사고력(판단 기준)
  - “연속된 구간” + “오른쪽 확장/왼쪽 축소” 구조가 보이면 슬라이딩 윈도우

```
for(int right = 0; right < n; right++) {
    sum += arr[right];
}
```

---

## 이분탐색

- ↗ 정렬된 상태에서 탐색(정렬필수)
- 사고력(판단 기준)
  - “정답 범위를 반으로 줄일 수 있다”
  - “로그 시간 필요”
  - (추가) “정답이 어떤 값의 최소/최대”면 매개변수 탐색 가능성 큼

```
while(left <= right) {
    int mid = (left + right) / 2;
}
```

## DP (동적 계획법)

- 이전 값을 저장
- ↗ 점화식 찾는 능력 중요
- 사고력(판단 기준)
  - 같은 계산을 반복한다(중복 부분 문제)
  - 이전 상태로 현재를 만든다(최적 부분 구조)
  - “최대/최소/경우의 수”가 DP 단골

```
int[] dp = new int[n+1];
dp[0] = 0; // 초기값
dp[1] = 1;

for(int i=2;i<=n;i++){
    dp[i] = dp[i-1] + dp[i-2];
}
```

## 그래프 알고리즘

그래프는 “표현 + 탐색”이 50%입니다.

- DFS
- BFS
- 다익스트라
- 플로이드 워셜
- 유니온 파인드
- 위상정렬(DAG 문제)

## 다익스트라

↗ 가중치 최단거리(음수 가중치 X)

```
PriorityQueue<Node> pq = new PriorityQueue<>();
pq.offer(new Node(start,0));

while(!pq.isEmpty()){
    Node now = pq.poll();
}
```

## 플로이드 워셜

↗ 모든 정점 최단거리(N이 작을 때)

```

for(int k=0;k<n;k++)
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            dist[i][j] = Math.min(dist[i][j],
                                  dist[i][k]+dist[k][j]);
    
```

## 유니온 파인드

- 연결 여부/사이클 판별/MST
- A를 끝내야 B 가능" 같은 문제

```

int find(int x){
    if(parent[x] == x) return x;
    return parent[x] = find(parent[x]);
}
    
```

## 위상정렬 (DAG 문제)

### 선후 관계 / 작업 순서 문제

```

Queue<Integer> q = new LinkedList<>();

for(int i = 1; i <= n; i++){
    if(indegree[i] == 0) q.offer(i);
}
    
```

## 트리 알고리즘

### MST (최소 신장 트리)

#### 크루스칼

- ↗ 간선 정렬 + 유니온파인드

```

Arrays.sort(edges); // cost 기준

int total = 0;
for(Edge e : edges){
    if(find(e.u) != find(e.v)){
        union(e.u, e.v);
        total += e.cost;
    }
}
    
```

---

## 문자열 알고리즘

- KMP
  - 트라이
  - 아나그램
- 

## 비트마스킹

```
int bit = 1 << 3;
```

---

## 시뮬레이션

- 문제에서 하라는 대로 구현
  - 알고리즘이 안 떠오르고 “규칙대로 움직여라/바꿔라”가 핵심이면 시뮬레이션
  - 구현 실수 방지가 핵심(방향 배열 dx/dy 추천)
-