1. 알고리즘 분석

알고리즘 정의 및 기준

- **알고리즘 정의:** 특정한 작업을 수행하기 위한 **유한한 명령문의 집합**
- **알고리즘의 다섯 가지 기준**
    1. **입력(Input):** 외부에서 제공되는 자료가 있을 수 있다.
    2. **출력(Outpu): 적어도 한 가지의 결과**를 생성한다.
    3. **명확성(Definiteness):** 각각의 명령은 **명확하고 모호하고 않아야** 한다.
    4. **유한성(Finiteness):** 알고리즘의 명령대로 수행하면, 어떤 경우에도 **한정된 수의 단계 이후 반드시 종료**한다.
    5. **유효성(Effectiveness):** 원칙적으로 모든 명령은 종이와 연필만으로 수행될 수 있도록 **기본적이고 실행 가능**해야 한다.
- **프로그램 정의: 알고리즘 + 자료구조**

알고리즘 분석 (시간 복잡도)

- **분석의 초점**: 특별한 경우를 제외하고 **시간 복잡도**만을 분석한다.
- **주된 분석 기준: 최악의 경우 분석 (Worst-case analysis).**
- **점근 표기법 (Asymptotic Notation):** 실행 시간을 입력 크기 n 에 대한 함수로 표현하며, 복잡도를 간단히 1 개의 항으로 표현하기 위해 사용한다.
    - **빅오 표기법 (Big-O Notation, O):** 알고리즘의 최악의 경우 성능(상한)을 표현한다. 상수와 낮은 차수의 항은 무시한다.
    - **빅세타 표기법 (Θ):** 알고리즘의 성능이 특정 함수로 **정확히 묶일 때 (상한과 하한이 동일)** 사용되며, **대략적인 실행 시간을 가장 정확하게 표현**하고 있다.
- **시간 복잡도 비교 (효율성 순서)**

$$O(1) < O(\log N) < O(N) < O(N \log N) < O(N^2) < O(N^3) < O(2^N) < O(N!) < O(N^N)$$

- **주요 시간 복잡도 예시**
    - O(1) (**상수 시간**): **배열에서 원소 접근 (random access), 스택/큐의 삽입(push/enqueue)/삭제(pop/dequeue)**
    - O(logN) **(로그 시간): 이진 검색(binary search)**
    - O(N) **(선형 시간)**: 배열/연결리스트에서 검색, 최솟값/최댓값 찾기.

2. 재귀

재귀의 정의와 구조

- **재귀 (Recursion) 정의**: 어떤 함수가 **자기 자신을 호출**하여 문제를 해결하는 프로그래밍 기법
- **주요 요소:**
    - **기본 사례 (Base Case):** 재귀 호출이 **멈추는 조건**으로, 더 이상 자신을 호출하지 않고 값을 반환한다.(간단하면 직접 해결)
    - **재귀 사례 (Recursive Case):** 함수가 자신을 호출하여 문제를 **더 작은 하위 문제로 나누는 부분**. (일반적이면 더 작은 형태로 자기 자신 호출).
- **재귀 ↔ 반복**: 반복문으로 표현된 알고리즘은 재귀문으로, 재귀문은 반복문으로 표현될 수 있다.

재귀 함수의 단점 및 유용한 경우

- **재귀의 단점 (성능/메모리)**
    - **성능:** 스택 호출 오버헤드 (스택 관리 비용 추가)로 인해 반복문에 비해 실행 시간이 더 소요된다.
    - **메모리: 스택 프레임 사용**으로 인해 **스택 오버플로우 위험**이 있다.
    - **피보나치 함수 예:** 재귀는 중복 계산이 발생하여 실행 시간이 많이 소요된다. (반복은 O(N)).
- **재귀가 유용한 경우 (복잡한 문제): 트리, 그래프, 분할 정복** 문제 및 **하노이 탑** 문제. 이러한 문제에서 재귀를 사용하면 코드가 **간결**하고 **직관적**이다.

3. 배열

배열의 정의 및 특징

- **배열 (Array) 정의: 동일한 데이터 타입**의 요소들을 순서대로 저장하는 **고정된 크기**의 자료구조.
- **주요 특징:**
    - **선형 자료구조:** 같으 자료형의 데이터를 여러 개 나열한 구조.
    - **연속된 메모리 공간**에 순차적으로 데이터 저장.
    - 선언/생성할 때 크기를 정하면, 그 크기로 **고정**됨.
    - 인덱스(방 번호, 0 부터 시작)를 통해 원소 접근

배열의 성능 및 장단점

- **배열 접근 시간:**
    - **탐색 (Random Access):** O(1) (**상수 시간** 소요).
    - **삽입/삭제:** O(N) (**선형 시간** 소요) - **원소의 이동**이 발생하기 때문.
- **장점:**
    - 구현이 쉽고 메모리 관리가 편함.
    - 인덱스 사용으로 **탐색(검색)이 상수 시간** 소요 (빠르다)
- **단점:**
    - **크기를 변경할 수 없음** (변경하려면 새 배열 생성 및 데이터 이동 필요)
    - **메모리 낭비** 발생 가능성
    - **오버플로우** 발생 가능성
    - 삽입/삭제가 **비효율적**(O(N))이다.

희소 행렬 (Sparse Matrix)

- **정의: 대부분의 항이 0 으로 구성**된 행렬. (자바에서는 0 인 항의 개수가 50%를 상회하면 희소 행렬로 판정)
- **문제점: 메모리 공간의 낭비** 초래 및 큰 희소 행렬은 2 차원으로 표현하면 메인 메모리 크기를 초과할 수 있음
- **효율적 표현 (3 원소쌍 표현):** 0 이 아닌 각 요소를 행 인덱스 (i), 열 인덱스 (j), 값 (value)으로 나타내는 방식
    - **장점: 메모리 효율성** (0 이 아닌 요소만 저장).
    - **단점:** 특정 위치의 값을 찾으려면 리스트를 순회해야 하므로, **접근 속도가 느릴 수 있음**
- **전치 행렬 (Transpose Matrix) 생성** (3 원소쌍 기준):
    - 단순한 <i, j, value>에서 <j, i, value>로마 바꾸면 **행의 숫자가 정렬되지 않음.**
    - **단순 알고리즘 시간 복잡도:** O(columns * elements) = O(rows * columns^2).
    - **빠른 전치 행렬 알고리즘 시간 복잡도:** O (columns + elements).

4. 연결 리스트

단순 연결 리스트 (Singly Linked List)

- **정의:** 각 노드가 **데이터**와 다음 노드의 주소를 가진 포인터(링크)를 가지고 한 줄로 연결되어 있는 방식으로 데이터를 저장하는 자료 구조이다.
- **노드 구성:** <데이터 필드 + 링크 필드>
- **특징:** 마지막 노드의 링크 필드는 **널(null)**값을 가진다.
- **성능 비교 (배열 대비):**

연산	배열 (Array)	연결 리스트 (List)
Random access (원소 접근)	O(1)	O(N)
Insertion/Deletion (삽입/삭제)	O(N)	O(1)

- **포인터 변수 연산(ADL):** 포인터 변수 혹은 참조변수의 연산은 **비교, 할당문만 가능**하며, **사칙연산은 불가능**하다.
- **모든 노드를 처음부터 끝까지 방문하는 코드 (while 문):**

```
p <- L;
while (p != null) do {
 print p.data
 p <- p.link;
}
```

원형 연결 리스트 (Circular Linked List)

- **정의:** 리스트의 **마지막 노드의 링크가 첫 번째 노드를 가리키는** 연결 리스트.
- **특징:**
  - **null 이 존재하지 않는다.**
  - 한 노드에서 다른 모든 노드로의 **접근이 가능**하다.
- **변형된 구현 (Tail 포인터 사용):**
  - tail **포인터**만 사용하면 **선두에 원소 삽입** 및 **마지막에 원소 추가**가 **상수 시간(O(1))에 해결**된다.

이중 연결 리스트 (Doubly Linked List)

- **정의:** 선행 노드(이전 노드)와 후행 노드(다음 노드) 모두를 가리킬 수 있도록 **양방향 링크**를 가진 연결 리스트.
- **노드 구성:** data, llink(왼쪽 링크), rlink(오른쪽 링크) 필드.
- **장점:**
  - **양방향 탐색 가능.**
  - **삽입, 삭제 작업이 쉬워짐.**
- **단점:**
  - **추가 링크 공간 필요.**
  - 구현이 복잡함.

5. 재귀/반복 비교 및 시간 복잡도

기준	재귀 함수	반복문
성능	스택 호출 오버헤드(스택 관리 비용 추가)	오버헤드 없음, 효율적
메모리	스택 프레임 사용, 오버플로우 위험	고정 메모리, 효율적
가독성	간결, 재귀적 문제에 직관적	명시적 반복, 코드 길어질 수 있음
적합 문제	트리, 그래프, 분할 정복	선형 작업, 간단 반복
동일 문제 해결 시	시간과 공간 사용 면에서 덜 효율적(시간이 더 소요)	시간과 공간 사용 면에서 더 효율적(시간이 더 짧다)
이진 탐색	O(log N) (효율적)	O(log N) (효율적)
선형 탐색	O(N) (비효율적)	O(N) (비효율적)