ChatGPT

알고리즘 스터디 커리큘럼 (11주 계획)

1. 알고리즘(APS)란 무엇이며 왜 중요한가?

알고리즘은 어떤 문제를 해결하기 위한 단계적 절차를 의미한다 [1]. 문제의 입력을 받아 정의된 절차에 따라 계산을 수행하고 원하는 출력을 산출하는 일련의 **논리적인 단계의 모임**이라고 볼 수 있다 [2] [3]. 특히 **APS**(Algorithm Problem Solving)는 이러한 **알고리즘 문제 해결 방법**을 공부하는 분야로, 주어진 문제를 정확하고 효율적으로 풀어내는 프로그램을 작성하는 것이 목표이다 [4].

신입 개발자의 역량을 키우는 데에는 **포트폴리오**, **컴퓨터 기초 지식(CS)**, **코딩 테스트(알고리즘)** 세 가지 축이 중요한데, 그 중에서도 알고리즘 실력은 많은 IT 기업의 코딩 테스트를 대비하는 데 필수적이다. 실제로 네카라쿠배당토(네이버, 카카오, 라인 등 주요 기술 기업)부터 4대 SI 기업까지 대부분 신입 채용에 알고리즘 코딩 테스트를 포함하고 있다 [5]. **코딩 테스트 전형을 통과**하려면 알고리즘 문제 풀이 연습이 필수이며, 알고리즘 학습을 통해 **문제 구현력과 논리적 사고력**을 크게 향상시킬 수 있다 [6] [7]. 나아가 다양한 문제를 해결하면서 디버깅 능력도 함께 기를 수 있다.

온라인 저지(Online Judge)는 이러한 알고리즘 문제를 연습하기 위한 온라인 채점 시스템을 말한다 [8]. 대표적으로 Baekjoon OJ, CodeUp, Programmers, Codeforces 등의 사이트가 있으며, 코드 실행 결과와 효율성을 자동으로 채점해준다. 온라인 저지를 통해 시간/메모리 제한 내에서 동작하는지를 확인할 수 있고, 전 세계 사람들과 동일한 문제로 실력을 비교하거나 향상시킬 수 있다.

요약하면, **알고리즘 공부(APS)**를 하는 이유는 **코딩 테스트 대비**와 **문제 해결 능력 향상**이다. 우리의 스터디 목표는 코딩 테스트 합격 수준까지 알고리즘 한 축을 마스터하고, 나아가 실무에서 필요한 **문제 해결 사고방식**을 기르는 것이다.

2. 효율적인 문제 해결 접근법

알고리즘 문제를 해결할 때는 체계적인 접근법이 필요하다. 우선 문제를 읽고 **요구사항을 정확히 이해**해야 하며, 입력/출력과 조건을 명확히 파악해야 한다. 그런 다음 다음과 같은 단계별 접근을 권장한다:

- **Simplify & Brute-force**: 처음에는 문제를 간단히 만들어보거나 가능한 한 **직관적인 완전탐색(Brute-force)**이나 **시뮬레이션** 방법으로 해결해본다. 일단 **동작하는 해법**을 구한 뒤에 차차 개선하는 편이 좋다 (예: 작은 입력에서 완전탐색으로 정답을 구한 후, 최적화 방향을 모색). 처음부터 지나치게 복잡한 최적해를 노리기보다 **단기 목표**로 동작하는 코드를 작성하는 것이 중요하다.

- **Step by Step Refinement**: 큰 문제는 곧바로 풀기 어려우므로 **문제를 하위 문제로 분할**하고 단계별로 해결한다. 한 번에 모든 것을 구현하려 하지 말고, 기능 단위로 나누어 **모듈별로 설계**한다. 예를 들어, 입력 파싱 -> 핵심 로직 구현 -> 출력 형식 맞추기 등을 단계별로 처리한다. 이렇게 하면 디버깅이 수월해지고 실수를 줄일 수 있다.

- **Incremental Development**: 부분적으로 코드를 작성하며 수시로 **중간 결과를 출력**하거나 간단한 테스트를 수행해본다. 예를 들어, 그래프 문제에서는 일단 입력을 잘 받아 그래프 구조를 만들고, 간단한 출력으로 제대로 구성되었는지 확인한 뒤 DFS/BFS를 구현한다. 단계별로 검증하며 나아가면 **논리적 버그를 조기에 발견**할 수 있다.

- **Problem Decomposition (분할 설계)**: 복잡한 문제일수록 함수나 클래스 등을 활용하여 논리를 쪼개는 것이 좋다. 이를테면 백트래킹 문제에서는 재귀 함수를 설계하고, DP 문제에서는 점화식을 세우고 테이블을 정의하는 등 **큰 문제를 작은 단위로 설계**해야 효율적이다.

이러한 접근법을 통해 처음엔 비효율적이더라도 올바르게 작동하는 해법을 얻고, 이후에 최적화 아이디어(그리디, DP, 이분 탐색 등)를 적용하여 개선해 나가는 것이 바람직하다. 중요한 것은 **정확성**을 우선 확보한 후, **점진적으로 효율성을 개선**하는 사고 프로세스이다.

3. 시간 복잡도와 공간 복잡도 (Big-O 표기법)

시간 복잡도는 알고리즘이 실행되는 동안 수행하는 연산의 횟수를 나타낸다. 즉, 입력 크기와 실행 시간의 관계를 수학적으로 표현한 것이다 [9]. **공간 복잡도**는 알고리즘이 사용하는 메모리의 양을 뜻한다. 알고리즘의 효율성을 논할 때 일반적으로 **빅-오 표기법(Big-O notation)**으로 복잡도를 표현한다 [10] [11]. Big-O 표기법은 입력 크기가 커질 때 **가장 빠르게 증가하는 항**을 기준으로 알고리즘의 성능을 표기하는 방법으로, **최악의 경우 시간**의 상한을 나타낸다 [11].

예를 들어, `O(n)`은 입력 크기 n에 비례하여 실행 시간이 증가함을 의미하고, `O(n^2)`는 입력이 커지면 실행 시간이 n의 제곱에 비례하여 증가함을 뜻한다. 기본적으로 알고리즘을 설계할 때는 **시간 복잡도를 최소화**하여 **실행 시간을 줄이는 것**이 중요하다 [12]. 문제의 입력 범위를 보고 적절한 알고리즘을 선택할 수 있어야 하며, Big-O 분석을 통해 어느 정도 성능을 예상할 수 있어야 한다.

또한 **평균 시간 복잡도**와 **최악 시간 복잡도**를 구분하기도 한다. 예를 들어 퀵정렬은 평균적으로 `O(n log n)`이지만 최악의 경우 `O(n^2)`이 될 수 있다. 상황에 따라 **암묵적 상수**나 **캐시 효율** 등도 성능에 영향을 주지만, 빅-오 표기는 큰 그림에서 성능을 판단하는 데 유용하다.

공간 복잡도도 유의해야 한다. 메모리를 과도하게 쓰면 제한을 초과할 수 있으므로, **필요한 만큼만 메모리를 사용**하도록 코드를 작성해야 한다. 예를 들어 길이 n짜리 배열 두 개를 사용하는 알고리즘은 `O(n)` 공간을 차지한다. 재귀 알고리즘의 경우 **재귀 호출 스택**으로 인해 추가 메모리가 사용되기도 한다.

마지막으로, 일부 알고리즘이나 자료구조 연산은 **암ORT**분석 (Amortized Analysis)을 통해 평균 성능을 평가한다. 예를 들어 동적 배열(vector)에 원소를 반복 삽입하는 연산은 단순히 보면 재할당 시 `O(n)`이 걸리지만, **평균적으로는 삽입 연산당** `O(1)` 시간에 수행된다고 간주한다. 이런 **암묵적 평준화된 비용**도 이해해두면 좋다.

4. 알고리즘의 표현 방법과 평가 기준

알고리즘을 설계했다면, 그것을 명확히 표현하고 평가하는 것이 중요하다. 일반적으로 알고리즘을 표현하는 방법으로 **의사코드(Pseudocode)**와 **순서도(Flowchart)** 두 가지가 자주 활용된다 [13].

- **의사코드**: 특정 프로그래밍 언어 문법에 얽매이지 않고, 사람이 이해하기 쉬운 형태로 알고리즘 절차를 기술한 것이다 [14]. 마치 프로그램 코드처럼 보이지만 자연어에 가까운 서술로, 주요 알고리즘의 흐름과 제어 구조를 표현한다. 의사코드는 알고리즘의 로직을 빠르게 파악하고 언어별 구현으로 옮기기 전에 계획을 세우는 용도로 유용하다.

- **순서도**: 표준화된 도형과 화살표를 사용하여 알고리즘의 처리 과정을 **시각적으로 표현**한 것이다 [14]. 시작/종료, 처리, 입력, 분기 등의 도형을 통해 알고리즘의 흐름을 한눈에 볼 수 있다. 순서도는 알고리즘의 전체 구조를 그림으로 보여주므로 직관적인 이해를 돕고, 논리의 흐름을 점검하는 데 도움이 된다.

알고리즘을 평가할 때는 여러 **기준**을 고려해야 한다:

- **정확성**: 주어진 모든 입력에 대해 알고리즘이 **정확한 결과**를 산출하는가. 논리상 버그가 없고 문제 요구조건을 모두 만족해야 한다. 정확성은 알고리즘의 가장 기본적인 조건이다.

- **작업량(효율성)**: 위에서 다룬 **시간 복잡도**에 해당하는 개념으로, **얼마나 적은 연산으로 작업을 완료하는가**를 본다. 연산 횟수가 적을수록 좋으며, 빅-오 표기상 더 낮은 차수의 복잡도를 가지는 알고리즘이 효율적이다 [10].

- **메모리 사용량**: 알고리즘이 얼마나 추가적인 메모리를 사용하는지 평가한다. **공간 복잡도**가 낮을수록 대용량 입력 처리에 유리하다. 메모리를 과도하게 사용하면 실행이 불가능해질 수 있으므로, **인플레이스(in-place)** 알고리즘처럼 추가 공간을 최소화하는 기법도 고려 대상이다.

- **단순성**: 알고리즘이 얼마나 **구현하기 쉽고 이해하기 쉬운가**이다. 지나치게 복잡한 알고리즘은 실수 확률이 높고 유지보수가 어렵다. 비슷한 복잡도의 알고리즘이라면 로직이 단순한 쪽을 선호하는 것이 일반적이다. 예를 들어, 버블정렬은 로직이 단순하지만 퀵정렬은 복잡하다. 상황에 따라 단순한 방법이 오히려 실용적일 수 있다.

- **최적성**: 해당 문제가 요구하는 최소한의 연산량으로 풀리는지, 즉 **더 이상의 개선 여지가 없는가**를 따진다. 어떤 문제에 대한 알고리즘이 알려진 하한 시간복잡도와 동일하면 최적(almost optimal)이라고 볼 수 있다. 또한 최적해를 보장하는지 (특히 최적화 문제에서) 여부도 포함된다.

좋은 알고리즘은 위의 특성들을 균형 있게 만족한다. 코딩 테스트에서는 **정확성과 작업량(시간)**이 가장 우선시되며, 주어진 시간/메모리 내에 정확한 답을 출력해야 한다. 따라서 스터디 진행 중에도 항상 **이 알고리즘이 왜 필요한지, 복잡도가 어떤지** 자문하며, 더 나은 대안은 없는지 고민하는 습관을 들이자.

5. 기본 수학 알고리즘 연습

코딩 테스트 입문 단계에서는 복잡한 알고리즘 지식이 없어도 풀 수 있는 **기초 수학 문제**들을 먼저 다루는 것이 좋다 [15]. 이를 통해 프로그래밍 언어 문법에 익숙해지고 구현력을 키울 수 있다. 예를 들어 다음과 같은 간단한 계산 문제들이 있다:

- **합 공식**: 1부터 N까지의 합, 또는 N부터 M까지의 합을 구하는 문제. 이를 위한 수학 공식 `N*(N+1)/2` 등을 알고 있다면 반복문 없이도 결과를 얻을 수 있다 [16]. 이러한 수열 합 공식은 기본 중의 기본이다.

- **팩토리얼과 피보나치**: 간단한 재귀 사례로 자주 등장한다. `N!` (팩토리얼)은 1부터 N까지 곱이며, 재귀나 반복으로 구현 가능하다. **피보나치 수열**은 `F(n) = F(n-1) + F(n-2)`로 정의되는 유명한 수열로, 작은 N에 대해서는 재귀적으로 구현해볼 수 있다 [17]. 다만 N이 커지면 재귀만으로는 비효율적이므로 DP 기법을 적용해야 한다는 것도 곧 배우게 될 것이다.

- **약수, 배수와 소수**: 어떤 수의 **약수(divisor)**를 구하거나, 두 수의 **최대공약수(GCD)**와 **최소공배수(LCM)**를 구하는 문제는 수학의 기본이다. 최대공약수는 **유클리드 호제법**을 이용하면 효율적으로 계산할 수 있다. 예를 들어 GCD(192, 162)를 구할 때 192를 162로 나눈 나머지를 취하는 과정을 반복한다. **소수(prime)** 관련 문제도 빈출하는데, **에라토스테네스의 체** 알고리즘을 알면 1부터 N까지의 소수를 빠르게 찾아낼 수 있다 [18]. 에라토스테네스의 체는 불리언 배열을 활용해 합성수를 제거하는 대표적인 **O(N log log N)** 방법이다. 소수 판정, 소수의 개수 구하기 등의 문제에 적용된다.

- **조합과 순열**: n개 중 r개를 선택하는 **조합 (nCr)** 계산이나, 순서를 고려해 뽑는 **순열 (nPr)** 계산도 가끔 등장한다. 작은 n에 대해서는 수학 공식을 직접 구현하거나 재귀로 모든 조합/순열을 생성하는 brute-force를 할 수 있다. 하지만 n이 큰 경우에는 팩토리얼 계산과 동일하게 오버플로우나 큰 수 연산 이슈가 있으므로, 문제 상황에 맞게 활용한다. 참고로 Python에서는 `itertools` 모듈이 제공하는 `combinations`와 `permutations` 함수를 사용하면 편리하게 조합/순열을 구할 수 있다 [19].

以上와 같은 기본 수학 알고리즘은 특별한 자료구조나 알고리즘 지식 없이도 해결 가능하지만, **효율적인 구현**은 여전히 중요하다. 예를 들어 1부터 N까지 합을 매번 반복문으로 더하면 `O(N)` 시간이지만, 수식으로 계산하면 `O(1)`에 끝난다. 최대공약수도 두 수의 차를 계속 빼기보다 `%` 연산을 활용하면 더 빠르다. 이러한 기초 문제들을 풀며 **사고를 코드로 구현하는 연습**을 충분히 하면 이후 복잡한 알고리즘을 배울 때도 큰 도움이 된다 [16]. 또한 산술 오버플로우, 부동소수점 오류 같은 **수치 계산상의 주의점**도 함께 익힐 수 있다. (예: C/C++에서는 `int` 범위 초과 주의, 부동소수점 비교는 오차 고려 등)

6. 배열과 문자열 처리

배열(Array)은 가장 기본적인 데이터 구조로, 연속된 메모리 공간에 자료를 저장한다. 배열은 **인덱스**를 통해 임의 위치의 원소에 **상수시간 접근(O(1))**이 가능하여, **임의 접근(Random Access)**에 매우 효율적이다. 배열을 잘 활용하면 간단한 해시 역할을 하는 **직접 접근 테이블(DAT, Direct Address Table)**을 만들 수 있다. 예를 들어 크기가 100인 불리언 배열을 0으로 초기화해 두고, 어떤 값이 등장하면 해당 인덱스에 1을 표시함으로써 **O(1)** 시간에 특정 값의 존재 여부나 빈도를 체크할 수 있다. 이런 기법은 **해시를 쓰지 않고도** 간단한 카운팅이나 체크를 할 때 유용하다.

배열은 1차원뿐 아니라 **2차원 배열(행렬)** 형태로도 많이 등장한다. 2차원 배열을 순회(traverse)하는 방법에는 여러 가지 패턴이 있다:

- **행 우선 순회**: 이중 루프에서 바깥 루프를 행(row)으로, 안쪽 루프를 열(column)로 돌리는 방식이다. 2차원 배열을 행 단위로 왼쪽에서 오른쪽으로 한 줄씩 읽는다.
- **열 우선 순회**: 바깥 루프를 열로, 안쪽을 행으로 하여 열 단위로 내려가며 순회하는 방식이다. 행/열 우선 순회는 메모리 접근 패턴이 달라 성능에도 미세한 영향이 있을 수 있다 (행 우선이 일반적으로 지역성이 좋아 더 빠르다; C 계열 언어의 메모리 저장 방식이 행 우선이므로).
- **지그재그 순회**: 한 행은 왼쪽->오른쪽으로, 다음 행은 오른쪽->왼쪽으로 순회하는 식으로 지그재그 형태로 탐색한다. 문제에 따라 특이한 순회 방법이 필요할 때가 있으며, 구현 난이도를 높이기 위한 아이디어로 쓰인다.
- **델타 탐색(Delta Search)**: 2차원 격자에서 인접한 위치들을 탐색할 때, 상하좌우 또는 대각선 등 **이웃 좌표로 이동**하는 기법이다. 흔히 `dx[]`, `dy[]` 배열을 만들어 이동 방향들을 저장해두고 for문으로 인접한 좌표를 방문한다. 예를 들어 격자에서 상하좌우 이동을 위해 `dx = [-1,1,0,0]`, `dy = [0,0,-1,1]`로 두고 인덱스를 조합하면 현재 위치에서 이웃을 간편히 탐색할 수 있다. 이동 시에는 **유효 범위 체크**(boundary check)를 해서 배열 인덱스가 0 미만이나 N 이상이 되지 않는지 검증해야 한다.
- **전치 행렬(Transpose)**: 행렬의 행과 열을 바꾸는 작업으로, 2차원 배열 알고리즘에서 종종 요구된다. 전치를 하면 `A[i][j]`가 `A[j][i]` 자리로 가게 된다. 이 연산은 `N×M` 행렬을 새로운 `M×N` 행렬로 만들어야 하므로, 추가 메모리 또는 제자리(in-place) 알고리즘이 필요하다.

  Note: 2차원 배열 관련 문제에서는 인덱스 처리 실수가 잦으므로, 반복문 범위와 인덱스 유효성 검사를 철저히 하는 습관이 필요하다. 또한 (r,c)와 (x,y) 좌표 표기 혼용에 유의한다.

문자열(String)은 문자들의 연속(sequence)으로, 사실상 **문자 배열**로 볼 수 있다. 다만 Python이나 Java의 `String` 객체처럼 **불변(immutable)** 특성을 가진 고수준 문자열도 있다. C에서는 `char[]` 배열로 문자열을 표현하며, **문자** `'\0'` **(널 문자)**로 끝을 표시한다. 이를 **널 종료 문자열(null-terminated string)**이라고 하며, 문자열의 길이는 이 종료 표식을 통해 알 수 있다. 반면 Python의 문자열은 변경 불가능하고, 덧붙이거나 수정할 때 새로운 문자열이 생성된다.

문자열을 다룰 때 알아두면 좋은 기초사항:

- **아스키 코드(ASCII)와 유니코드**: 문자와 정수 코드 값의 관계를 이해해야 한다. 아스키는 영문자, 숫자, 일부 특수문자를 포함한 7비트 코드이며, 예를 들어 `'A'`는 65, `'a'`는 97, `'0'`은 48이다. C/C++에서는 문자형 `char`를 정수처럼 취급할 수 있어 `'A'+1`이 `'B'`가 되는 식의 연산이 가능하다. 유니코드는 보다 넓은 문자 집합(한글 등)을 표현하며, 파이썬의 `str`은 유니코드 문자 시퀀스로 동작한다.

- **고정 길이 vs 가변 길이**: C의 `char arr[100]`처럼 **고정 크기 배열**에 문자열을 저장하면, 최대 99글자(마지막은 널문자)까지 수용할 수 있고 그 이상은 잘린다. 반면 `std::string`이나 Python `str`처럼 **가변 길이** 문자열은 내용에 따라 동적으로 크기가 바뀐다. 코딩 테스트에서는 주로 가변 길이 문자열 타입을 쓰지만, C의 경우 입력 버퍼 크기에 유의해야 한다.

- **문자열 길이 제어와 종료**: C에서 문자열 처리 시 `strlen`으로 길이를 구하거나, 입력 함수 사용 시 버퍼 오버플로우를 방지해야 한다. C++의 `getline`이나 Python의 `input()`은 자동으로 안전하게 읽어준다. 문자

열을 다룰 때 **오프 바이라인 오류(Off-by-one error)**, 즉 인덱싱 실수로 마지막 문자를 건너뛰거나 널 문자를 잊는 오류를 조심한다.

- **정수 ↔ 문자열 변환**: 코딩하다 보면 숫자를 문자열로 바꾸거나 그 반대가 필요하다. 예를 들어 `'123'`이라는 문자열을 정수 123으로 바꾸려면 C에서는 `atoi` 함수를, C++에서는 `stoi`, Python에서는 `int()`를 사용한다. 반대로 정수를 문자열로 변환할 때는 `sprintf`(C), `to_string`(C++), `str()`(Python) 등을 사용한다. 이러한 기본 변환 함수 사용법을 알아두면 입출력 처리나 문자열 연산 문제를 풀 때 편리하다.

마지막으로, **입출력 스트림**에 대한 이해도 필요하다. 코딩 테스트에서는 입력 크기가 매우 큰 경우가 많으므로, 언어별로 **빠른 입출력 방법**을 숙지하는 것이 좋다. 예를 들어 C++에서는 `scanf/printf`나 `iostream`을 sync off 하고 `istreambuf_iterator`를 쓰는 등의 기법으로 속도를 높인다. Python에서도 기본 `input()` 대신 `sys.stdin.readline`을 쓰거나, 출력에 `sys.stdout.write`를 사용하는 것이 유리할 때가 있다. 자바는 `BufferedReader`/`BufferedWriter` 등을 쓴다. 이러한 방법들은 수백만 개의 숫자를 읽고 써야 할 때 시간초과를 예방해준다. 다만, 너무 이른 단계에서 최적화된 I/O에 집착할 필요는 없으며, **문제가 요구하는 경우에만 적용**하면 된다.

7. 주요 자료구조 (스택, 큐, 연결리스트, 해시, 힙)

효율적인 알고리즘을 작성하려면 **자료구조(data structure)**에 대한 이해가 필수적이다. 자료구조는 데이터를 저장하고 조직화하는 방식으로, 문제에 적합한 구조를 사용하면 코드 구현과 성능 면에서 큰 이점을 얻는다. 여기서는 알고리즘의 기본이 되는 주요 자료구조들을 정리한다 [20]:

- **스택(Stack)** – **후입선출(LIFO)** 구조로, 한 쪽 끝(top)에서만 자료를 넣고(push) 뺄(pop) 수 있는 형태다. 현실에서 접시 쌓기처럼 가장 나중에 넣은 것이 먼저 나오는 원리다 [21] [22]. 스택은 **함수 호출의 실행 흐름(콜 스택)**, **괄호 짝 검사**, **되돌리기(undo)** 기능, **DFS 구현** 등에 쓰인다. 예를 들어 **괄호 검증 문제**에서 여는 괄호를 스택에 넣고, 닫는 괄호가 나올 때 짝을 확인하며 pop하는 식으로 해결한다. **재귀 알고리즘**도 내부적으로는 시스템 스택을 사용하여 함수 호출을 관리한다. 일반적으로 스택의 기본 연산(push/pop)은 `O(1)`이며, 배열로도 구현되고 연결리스트로도 구현될 수 있다. 단 스택에는 최대 크기가 있을 수 있으며, 초과 시 Stack Overflow 오류가 발생한다 [23]. C++에서는 `<stack>` 컨테이너, 파이썬에서는 리스트(append/pop)나 `collections.deque`로 스택처럼 사용할 수 있다.

- **큐(Queue)** – **선입선출(FIFO)** 구조로, 한쪽 끝(rear)에서 삽입하고 반대쪽(front)에서 제거하는 형태다. 은행 대기줄처럼 먼저 온 사람이 먼저 나간다 [24]. 큐의 대표적인 활용은 **BFS(너비 우선 탐색)**에서 방문할 노드를 순서대로 저장하는 것과 **운영체제의 작업 스케줄링** 등에 있다 [25]. 큐의 연산(enqueue/dequeue)도 `O(1)`로 효율적이다. 큐는 **데크(Deque)**로 일반화되는데, 데크는 양쪽 끝에서 삽입/제거가 모두 가능한 구조다. 파이썬의 `collections.deque`는 데크 구현체로, 큐로도 스택으로도 사용 가능하다. 또한 **원형 큐(circular queue)** 개념도 있는데, 이는 배열로 큐를 구현할 때 front나 rear가 배열 끝에 도달하면 처음으로 돌아가도록 하는 방식으로 **메모리를 효율적으로 사용**한다 [26]. C++에서는 `<queue>` (및 `<deque>`)를 사용한다.

- **연결 리스트(Linked List)** – **노드(Node)**라 불리는 개별 원소들이 포인터로 서로 연결된 구조다. 배열과 달리 메모리상 연속되지 않아도 되며, 각 노드는 데이터와 다음 노드의 참조를 가진다. 연결 리스트는 **삽입/삭제가 O(1)**로 매우 효율적이지만 임의 위치 접근은 **O(n)**이 걸린다 (앞에서부터 차례로 가야 하므로). 따라서 연결 리스트는 **빈번한 삽입/삭제 작업**이 있는 경우 유용하며, **랜덤 액세스**가 필요할 땐 부적합하다. 실무에선 언어 제공 리스트(Vector<ArrayList> vs LinkedList 등) 선택시 이 점을 고려해야 한다. 알고리즘 문제에서는 연결 리스트를 직접 구현하는 경우는 드물지만, **LRU 캐시** 구현이나 **특정 패턴의 자료구조**를 만들 때 활용된다. C++에는 `<list>`로 이중 연결 리스트를 제공하고, 파이썬의 `list`는 배열 리스트이지만 `collections.deque`를 양방향 연결리스트에 가깝게 쓸 수 있다.

- **해시(Hash) / 해시테이블** – 키를 해시 함수로 변환한 인덱스로 대응시켜 **평균 O(1)** 시간에 검색/삽입을 지원하는 자료구조다. 파이썬의 `dict`, 자바의 `HashMap`, C++의 `unordered_map` 등이 해시테이블 구현체다. **해시 함수(Hash function)**는 임의 크기 키를 고정 길이의 해시값으로 매핑하는데, 잘 설계된 해시 함수는 키가 고르게 분포되어 충돌을 최소화한다. 해시 구조는 **키-값 쌍 데이터를 저장**하거나, 단순히 집합처럼 **원소 존재 여부 검사**에 널리 쓰인다. 예를 들어 전화번호 목록에서 특정 이름을 효율적으로 찾을 때 해시를 이용하면 좋다. 다만 해시테이블은 **충돌 발생 시 성능 저하**가 있을 수 있고, 메모리 사용량이 배열보다 크다. 그래도 보편적으로 평균적으로 매우 빠른 성능 덕분에 코딩 테스트에서 자주 활용되는 구조다 (예: 두 수 합 문제에서 보조 자료구조로 사용하여 `O(n)`에 푸는 등).

- **힙(Heap)과 우선순위 큐(Priority Queue)** – 힙은 **완전이진트리** 기반의 자료구조로, **부모 노드가 자식보다 항상 크거나 작은** (최대힙/최소힙) 특성을 가진다. 이를 이용한 **우선순위 큐**는 가장 높은(또는 낮은) 우선순위를 가진 요소를 빠르게 추출할 수 있는 구조다. 삽입과 삭제(최고 우선순위 추출)가 평균 `O(log n)`에 이루어진다. 코딩 테스트에서는 **최소 비용 문제**나 **그리디 알고리즘**에서 자주 등장한다. 예를 들어 **다익스트라 알고리즘**은 우선순위 큐로 최단 거리를 가진 노드를 효율적으로 뽑아 처리한다 [27]. 또 **허프만 코딩**과 같은 그리디 알고리즘에서도 최소힙을 이용해 가장 작은 두 원소를 반복적으로 합치는 과정이 있다. C++에서는 `<priority_queue>`로, Python에서는 `heapq` 모듈로 힙을 사용할 수 있다.

以上 자료구조들은 각자의 **장단점과 연산 복잡도**를 잘 이해해야 한다. 또한 대다수 언어는 이러한 기본 구조들을 표준 라이브러리(STL 등)로 제공하므로, **직접 구현보다는 내장 기능 활용**을 권장한다 [20]. 다만 학습을 위해 간단한 스택/큐/리스트 구현을 연습해 보는 것도 구조를 깊이 이해하는 데 도움이 된다. 본 스터디에서는 필요에 따라 구조별 간단한 구현도 다뤄볼 예정이다.

8. 재귀와 백트래킹 (Recursion & Backtracking)

재귀 호출(Recursion)은 자기 자신을 호출하는 함수로, 알고리즘 설계의 강력한 도구 중 하나이다. 재귀를 사용하면 복잡한 문제를 **간결한 구조**로 표현할 수 있는데, 수학적 귀납법처럼 **기저 조건(base case)**에서 답을 알고, 그보다 큰 경우에는 자기 자신을 호출하여 문제를 해결한다. 예를 들어 팩토리얼 `f(n) = n * f(n-1)`은 `f(1)=1`을 기저로 하는 재귀 정의이고, 피보나치 수 역시 재귀적으로 정의된다. 재귀 함수를 작성할 때는 언제 끝날지(기저 조건)를 명확히 하고, 함수를 호출할 때 문제의 크기가 줄어들도록 만들어 **무한 재귀에 빠지지 않게** 해야 한다.

재귀 실행 시 시스템은 **함수 호출 스택**을 사용한다. 매 함수 진입 시 지역 변수와 수행 위치 등이 스택 프레임에 저장되고, 함수가 끝나면 스택에서 내려온다 [28]. 이 때문에 **재귀 깊이**가 너무 깊으면 (예: 재귀 호출 10000번) 스택 오버플로(Stack overflow)나 성능 문제가 발생할 수 있다. C++의 기본 스택 크기로는 일반적으로 1000여 회 이상의 깊은 재귀 호출이 위험할 수 있다. 따라서 경우에 따라 **꼬리 재귀 최적화(tail recursion optimization)**가 없는 언어에서는 반복문으로 전환하거나, **명시적 스택 사용**으로 변환하기도 한다. 하지만 적절한 범위 내의 재귀는 코드를 매우 깔끔하게 만들어주며, 특히 **분할정복**과 **백트래킹**에서 유용하다.

백트래킹(Backtracking)은 해를 찾는 과정에서 **후보해를 구축하다가 조건에 맞지 않으면 되돌아(backtrack)** 가며 탐색하는 알고리즘 기법이다. 흔히 **DFS 깊이우선 탐색**을 기본으로 하며, 모든 가능한 해를 탐색하되 불필요한 분기는 조기에 가지치기(pruning)하여 **검색 공간을 줄이는 최적화 기법**을 동반한다. 백트래킹은 **결정 문제(decision problem)**나 **최적화 문제**를 해결하는 데 쓰이며, 아래와 같은 전형적인 예제가 있다:

- **N-Queen 문제**: N×N 체스판에 N개의 퀸을 서로 공격하지 못하게 놓는 방법을 찾는 문제로, 백트래킹으로 모든 배치를 탐색하면서 조건을 만족하지 않으면 바로 가지치기한다.
- **미로 찾기**: 미로에서 출구까지 가는 경로를 찾는 문제는 DFS/backtracking으로 시도 경로를 탐색하다 막히면 되돌아가는 방식으로 해결할 수 있다.
- **부분집합의 합(Subset Sum)**: 주어진 숫자 집합이 특정 합을 만들 수 있는지 찾는 문제. 모든 부분집합을 시도해보되, 현재 합이 목표를 초과하면 더 내려가지 않고 백트랙한다.
- **그래프 색칠하기(Map Coloring)**: 인접한 노드끼리는 같은 색을 쓰지 않도록 그래프를 몇 가지 색으로 칠하는 문제도 백트래킹으로 해볼 수 있다. 한 노드씩 색을 정해나가다가 위배 시 돌아간다.

백트래킹 구현의 핵심은 **재귀 호출로 상태 공간 트리를 깊이 탐색**하는 것이다. 각 재귀 호출 스택이 탐색 트리의 한 노드에 대응되고, 리턴하면서 상태를 이전으로 복구(restore)하는 과정을 거친다. 이때 가지치기를 얼마나 잘 하느냐에 성패가 달렸다. 가지치기는 문제의 제약을 활용하여 **불가능한 경로는 일찍 포기**하는 전략이다. (예: 부분집합 합에서 남은 숫자를 다 더해도 목표를 못 채우면 더 내려가지 않는다 등)

재귀와 백트래킹은 밀접한 관계가 있다. 백트래킹을 재귀 없이 구현하는 것도 가능하지만 오히려 재귀를 쓰면 자연스럽게 **탐색의 진행과 되돌아감**을 표현할 수 있다. 다만 백트래킹은 경우에 따라 **탐색 공간이 지수적으로 커질 수 있어 시간 복잡도가 매우 높으므로**, 입력 크기에 유의해야 한다. 또한 동일 문제를 백트래킹 대신 **동적계획법(DP)**으로 푸는 등 개선 여지가 없는지도 생각해봐야 한다. 백트래킹은 완전탐색에 기반하므로, 해답이 하나만 필요하거나 최적해가 필요하지 않은 결정 문제에 쓰는 것이 적합하며, **최적해를 구하는 문제**에는 각 탐색 branch에서 현재까지의 최적값을 저장하여 불필요한 탐색을 줄이는 기법(가지치기의 일종)을 추가하기도 한다.

9. 정렬 알고리즘 (Sorting)

여러 개의 데이터를 **정렬(Sort)**하는 것은 알고리즘의 기본 중 기본이다. 입력 데이터를 원하는 기준에 따라 **오름차순** 혹은 **내림차순**으로 배열하는 작업으로, 많은 문제의 전처리나 부분 절차로 등장한다. 정렬 알고리즘에는 다양한 방법이 있으며, 시간 복잡도 및 메모리 사용, 안정성 등에 차이가 있다. 주요 정렬 알고리즘은 다음과 같다:

- **버블 정렬(Bubble Sort)**: 인접한 두 원소를 비교하여 잘못된 순서라면 교환을 반복함으로써 큰 값이 차례로 끝으로 "거품처럼" 떠오르게 하는 방식이다. 구현이 단순하지만 **O(n^2)**로 느리며, 학습용으로 언급된다.

- **선택 정렬(Selection Sort)**: 매 단계마다 남은 요소 중 최솟값을 찾아 현재 위치와 교환하는 방식이다. 이 역시 **O(n^2)**이고, 작은 데이터에만 사용된다.

- **삽입 정렬(Insertion Sort)**: 필요 위치에 삽입하는 개념으로, 정렬된 부분 배열을 유지하면서 다음 원소를 알맞은 위치에 끼워 넣는다. 데이터가 거의 정렬되어 있는 경우 매우 효율적이지만, 최악의 경우 **O(n^2)**이다.

위의 단순 정렬들은 구현하기 쉽지만 **시간 복잡도가 높아** 큰 입력에는 부적합하다. 실용적으로 널리 쓰이는 정렬 알고리즘은 대부분 **O(n log n)**의 복잡도를 갖는다:

- **합병 정렬(Merge Sort)**: **분할 정복(divide and conquer)** 전략을 사용한다. 리스트를 절반씩 재귀적으로 분할하여 정렬한 뒤, 두 정렬된 리스트를 **병합(merge)**하면서 정렬을 완성한다. 안정적인 정렬이며, 시간 복잡도는 항상 `O(n log n)`으로 일정하지만, 추가 배열 공간 `O(n)`이 필요하다.

- **퀵 정렬(Quick Sort)**: 평균적으로 매우 빠른 정렬 알고리즘으로, 분할 정복을 사용하지만 제자리(in-place)에서 수행된다. 리스트에서 하나의 **피벗(pivot)**을 선택하고, 피벗보다 작은 요소들은 왼쪽으로, 큰 요소들은 오른쪽으로 보내 분할한다. 그런 다음 좌우 부분 리스트에 대해 재귀적으로 정렬을 수행한다. **분할(partition)** 과정의 구현 방법에 따라 **Hoare 파티셔닝**과 **Lomuto 파티셔닝** 알고리즘이 흔히 언급되는데 [29], Hoare 방식이 교환 횟수가 적어 보통 더 빠르다. 퀵정렬은 평균 `O(n log n)`이지만 피벗 선택이 나쁘면 최악 `O(n^2)`이 될 수 있다. 실전에서는 무작위 피벗 선택이나 Median-of-three 기법 등으로 최악 상황을 피한다. C++의 `std::sort`가 내부적으로 퀵정렬 기반 (일부 상황에선 introsort로 전환)이다.

- **힙 정렬(Heap Sort)**: 위에서 설명한 힙 자료구조(완전이진트리)를 활용한 정렬이다. 최대힙을 구성하면 루트에 가장 큰 값이 오므로 이를 제거하여 배열 끝에 놓고, 남은 요소로 다시 힙 조정(heapify)하는 과정을 반복한다. `O(n log n)` 보장에 추가 공간이 거의 필요 없지만, 데이터 접근 패턴이 불규칙해 캐시 효율이 떨어질 수 있어 실제 성능은 퀵정렬보다 떨어지기도 한다.

정렬 알고리즘을 공부할 때는 각 알고리즘의 **시간/공간 복잡도**, **제자리 여부(in-place)**, **안정성(stability)**을 함께 살펴봐야 한다. 안정정렬은 값이 같은 요소의 원래 상대 순서가 정렬 후에도 유지되는 성질로, 특정 상황 (예를 들어 여러 키로 정렬할 때)에서 중요하다. Merge Sort는 안정적이고, Quick Sort와 Heap Sort는 기본 형태는 불안정하다.

또한 특수한 경우에 **선형 시간 정렬**도 가능하다. **계수 정렬(Counting Sort)**은 데이터 값의 범위가 한정적일 때 유효하며, **기수 정렬(Radix Sort)**은 숫자나 문자열같이 자리수로 나눌 수 있는 데이터에 대해 자리수별로 안정 정렬을 적용해 정렬한다. 이러한 방법은 비교 기반이 아니므로 `O(n)` 혹은 `O(nk)` 등으로 동작하지만, 범위나 공간이 제한적일 때만 현실적으로 쓸 수 있다.

정렬은 많은 알고리즘 문제에서 **전처리 단계**로 등장한다. 예를 들어 **두 수 합** 문제에서 한 배열을 정렬한 후 투 포인터를 쓰거나, **그리디 알고리즘** 적용 전에 입력을 정렬하는 등의 아이디어가 흔하다. 따라서 다양한 정렬 알고리즘의 특성을 알고, 언어별 내장 정렬 함수를 사용할 줄 아는 것은 필수이다. 대부분의 언어 라이브러리 정렬은 최적화되어 있으므로, 면접 상황이 아니라면 직접 구현보다 **내장 함수 활용**을 우선하는 게 좋다 (단, Python의 `sorted`나 `list.sort()` 역시 팀소트 기반으로 `O(n log n)`임을 기억하자).

10. 탐색 알고리즘과 투 포인터 기법

어떤 데이터 모음에서 원하는 값을 **찾는 알고리즘(Searching)**도 기본 중 하나이다. 가장 단순한 방식은 처음부터 끝까지 훑는 **선형 탐색(Linear Search)**으로, 정렬 여부와 상관없이 적용 가능하지만 **O(n)** 시간이 걸린다. 더 빠른 탐색을 위해서는 정렬된 배열에 대해 **이분 탐색(Binary Search)**을 사용한다. 이분 탐색은 탐색 구간을 절반으로 줄여가며 찾는 방식으로, **O(log n)**에 원하는 값을 찾거나 존재하지 않음을 판단할 수 있다. 단, 이분 탐색을 쓰려면 데이터가 정렬되어 있어야 한다.

이분 탐색의 구현 포인트는 중간값 인덱스를 계산하고 (예: `mid = (low+high)//2`), 그 값과 찾는 값(target)을 비교하여 탐색 범위를 절반으로 좁히는 것이다. target이 중간값보다 작으면 오른쪽 절반을 버리고 왼쪽에서 계속 탐색하고, 크면 왼쪽을 버리고 오른쪽으로 간다. 값이 일치하면 위치를 반환한다. 이 과정에서 **루프 불변식**과 **종료 조건**을 정확히 다뤄야 한다. 인덱스 계산 시 오버플로우가 발생하지 않도록 `(low + high) // 2` 대신 `low + (high-low)//2`를 쓰는 등의 세부 팁도 있다.

이분 탐색은 검색뿐만 아니라, **Lower Bound / Upper Bound**를 찾는 응용도 중요하다. Lower bound는 찾고자 하는 값 이상이 처음 나타나는 위치, Upper bound는 찾는 값보다 큰 값이 처음 나오는 위치를 의미한다. 이를 이용하면 정렬된 배열에서 특정 값의 **개수**를 셀 수도 있다 (upper_bound - lower_bound로 계산). C++ `<algorithm>`에 `lower_bound`, `upper_bound` 함수가 제공되고, Python에서는 `bisect_left`, `bisect_right`가 동일한 기능을 한다. 이러한 함수들을 활용하면 이분 탐색을 직접 구현하지 않고도 편리하게 경계 값을 찾을 수 있다.

또 하나 알아둘 개념은 **파라메트릭 서치(Parametric Search)**이다. 결정 문제의 해를 판별하는 함수를 이용해 최적해를 **이분 탐색으로 찾아내는 기법**이다. 예를 들어, 어떤 조건을 만족하는 **최대값/최소값**을 구하는 문제에서, 값 X를 넣으면 "YES/NO"로 판정할 수 있다면, 그 Y/N 결과가 단조성을 띌 경우 이분 탐색으로 경계 지점을 찾아낸다. **예:** 특정 예산 내에서 만들 수 있는 최대 높이의 탑을 찾는 문제를 생각해보면, 높이 H로 탑을 세울 수 있는지를 결정 문제로 만들고, H를 이분 탐색하여 가장 높은 가능한 H를 찾는 식이다. 이러한 기법은 **최적화 문제를 결정 문제**로 바꾸어 푸는 아이디어로서, 이분 탐색의 활용도를 넓혀준다.

한편, 정렬된 배열이나 두 배열을 효율적으로 탐색할 때는 **투 포인터(Two Pointers)** 기법이 자주 사용된다. 투 포인터는 말 그대로 **두 개의 인덱스(포인터)를 이용**하여 배열을 탐색하는 방법이다. 일반적으로 하나의 배열 (또는 두 배열)에서 포인터를 움직이며 조건에 따라 증가/감소시켜 나간다. 몇 가지 활용 예시는 다음과 같다:

- **정렬된 두 배열 합치기**: Merge Sort의 merge 단계처럼, 각 배열에 포인터를 두고 작은 쪽을 선택해 결과에 넣으면서 포인터를 움직인다.

- **특정 합 찾기 (두 수의 합)**: 오름차순 배열에서 두 포인터를 양 끝에 두고 합에 따라 포인터를 조정한다. 합이 목표보다 작으면 왼쪽 포인터를 오른쪽으로, 합이 크면 오른쪽 포인터를 왼쪽으로 옮겨가며 찾는다. 이렇게 하면 **O(n)**에 두 수 합 문제를 풀 수 있다 (정렬 + 투포인터).
- **슬라이딩 윈도우(Sliding Window)**: 투 포인터의 한 형태로, 두 포인터로 현재 윈도우(구간)의 시작과 끝을 가리킨다. 한 포인터를 확장하며 조건을 만족하는 구간을 찾고, 다른 포인터를 움직여 구간을 축소한다. 예를 들어, **부분합 문제**(부분 연속 구간의 합이 특정 값을 넘는 최소 길이 등)는 슬라이딩 윈도우로 푼다. 윈도우 합이 목표보다 작으면 오른쪽 포인터를 늘려 합을 키우고, 목표 이상이면 왼쪽 포인터를 줄여 합을 줄이는 식으로 최적 해를 찾는다. 문자열의 **부분 문자열 문제**(예: 서로 다른 문자로 이루어진 가장 긴 부분문자열 등)도 슬라이딩 윈도우로 해결 가능하다.

투 포인터/슬라이딩 윈도우 기법은 코드가 비교적 간단하면서도 상당수의 배열/문자열 문제를 선형 시간에 해결하게 해준다. 다만, **정렬**이 필요하거나 데이터 특성을 알아야 적용 가능하므로 모든 상황에 쓰이진 않는다. 스터디에서는 투 포인터를 이용한 다양한 문제 (예: **BOJ 2470 두 용액** 등)를 풀어보며 이 기법의 패턴을 익힐 예정이다. 특히 슬라이딩 윈도우는 구간 합, 문자열 처리 등에서 유용하므로 꼭 연습해보자 [30] .

11. 문자열 알고리즘 (패턴 매칭 등)

문자열 관련 문제는 코딩 테스트에서 매우 흔하게 등장하는 분야다. 단순한 구현 문제부터 복잡한 텍스트 알고리즘까지 범위가 넓다. 여기서는 **문자열 패턴 매칭 알고리즘**과 기타 알아두면 좋은 알고리즘들을 다룬다.

- **브루트포스 패턴 검색**: 문자열에서 부분 문자열(패턴)을 찾는 가장 단순한 방법은, 패턴의 시작 위치를 0부터 차례로 시도하면서 패턴 전체가 일치하는지 확인하는 것이다. 이 방식은 **O(n*m)** (n=텍스트 길이, m=패턴 길이)으로, 최악의 경우 비효율적이다. 예를 들어 텍스트 "AAAAAA"에서 패턴 "AAAAB"를 찾을 때 불필요한 비교가 많이 일어난다.

- **KMP 알고리즘**: **Knuth-Morris-Pratt 알고리즘**은 대표적인 선형 시간 문자열 검색 알고리즘으로, 패턴 내의 접두사/접미사 일치 정보를 활용하여 불필요한 비교를 줄인다 [31] . KMP는 패턴에 대해 **부분 일치 테이블 (lps, longest prefix that is also suffix)**을 미리 계산하고, 텍스트를 스캔하면서 매칭이 깨지면 패턴을 처음부터 다시 비교하지 않고 lps 테이블을 참고해 이동한다. 이렇게 텍스트 인덱스는 한 번씩만 증가하고 뒤로 가지 않으므로, **O(n + m)**에 검색이 가능하다 [32] . KMP는 구현이 조금 복잡하지만, 패턴 매칭 문제가 나오면 매우 유용하다.

- **보이어-무어 알고리즘**: Boyer-Moore는 실전에서 매우 빠른 알고리즘으로, 패턴의 마지막 문자부터 비교해 나가며 **불일치 시 많은 문자를 건너뛸 수 있는**heuristic을 사용한다. **Bad Character 규칙**과 **Good Suffix 규칙** 두 가지 규칙으로 이동 거리를 계산하는데, 잘 맞으면 한 번에 패턴 길이보다 더 뛰어넘기도 한다. 평균 성능은 훌륭하지만 최악 경우 O(n*m)도 될 수 있다. 다만 실제 텍스트(예: 영문)에 대해서는 거의 선형에 가까운 속도를 보인다. Boyer-Moore도 KMP처럼 패턴에 대한 전처리(테이블 생성)가 필요하며, 구현이 복잡한 편이라 코테에서 직접 구현할 일은 드물지만 원리를 알아두면 좋다.

- **라빈-카프 알고리즘**: Rabin-Karp는 **해싱**을 이용한 문자열 검색 알고리즘이다. 패턴의 해시값과 텍스트의 각 부분 문자열 해시값을 비교해서 찾는다. 모든 위치에서 해시를 계산하면 `O(n*m)` 이지만 **롤링 해시(재귀적 해시)** 기법을 사용해 이전 위치 해시로 다음 위치 해시를 빠르게 계산하여 평균 `O(n+m)` 에 동작한다. 다만 해시 충돌 처리를 해야 하고, 최악의 경우 충돌이 많으면 성능이 떨어질 수 있다. Rabin-Karp의 장점은 동시에 여러 패턴을 찾는 등으로 확장하기 쉽다는 것이다. 기본 개념이 단순해, 예컨대 **회문 검사**나 **문자열 비교** 등의 문제 응용으로 등장하기도 한다.

위의 알고리즘들은 문자열 **검색**에 초점을 맞추지만, 문자열 알고리즘 분야에는 이 외에도 다양한 문제들이 있다. **트라이(Trie)**는 문자열 집합을 효율적으로 저장하고 검색하기 위한 트리형 자료구조로, 접두사 검색에 특히 유용하다. 예를 들어 많은 단어 중 특정 prefix로 시작하는 단어들을 빠르게 찾는 문제는 트라이로 해결 가능하다. (과거 카카오 코테에 트라이를 응용한 문제가 나온 바 있다.) 트라이는 메모리 사용이 크다는 단점이 있지만, **문자 단위로 가지를 뻗어나가며 단**

어를 구성하므로 검색이 문자열 길이에 비례한 시간이 걸리고, **자동완성**이나 **사전(Dictionary)** 구현 등에 사용된다. 스터디 후반부에 시간이 있다면 트라이도 다뤄볼 예정이다 33 .

또 한편, **문자열 압축 및 부호화** 알고리즘도 알고리즘 교양으로 알아둘 만하다. 예를 들어 **허프만 코딩(Huffman Coding)**은 greedy 알고리즘으로 문자열의 문자들을 0/1 비트의 코드로 바꿔 압축하는 기법이다. 출현 빈도가 낮은 문자는 긴 코드, 빈도가 높은 문자는 짧은 코드를 부여하여 전체 길이를 최소화한다. 허프만 트리라는 이진트리를 구성하며, **우선순위 큐**를 사용해 가장 빈도 낮은 노드 두 개를 합치는 것을 반복한다. 결과는 프리픽스 코드(어떤 코드도 다른 코드의 접두사가 아닌) 형태라 안정적으로 복원 가능하다. 허프만 코딩 자체가 코테에 직접 나오진 않지만, **greedy 전략의 예시**로 종종 언급된다.

마지막으로, 문자열 관련 작업 중 **암호화/인코딩**의 간단한 예로 **시저 암호(Caesar cipher)**를 들 수 있다. 이는 각 문자를 알파벳상 일정량 밀어 다른 문자로 치환하는 고전 암호다. 예를 들어 3글자 밀면 "ABC" -> "DEF"가 된다. 복호화는 반대로 밀면 된다. 시저 암호 문제는 주로 단순 구현 문제로 나오며, 파이썬에선 `chr`, `ord`를 활용하거나, 모듈러 연산으로 알파벳을 순환시키는 형태로 구현한다. 또한 **비트 XOR 연산을 이용한 암호화** 기법도 있는데, 어떤 키 값과 평문을 XOR하면 암호문이 되고, 다시 같은 키로 XOR하면 원문이 복원되는 성질을 이용한다. XOR은 **비트가 같으면 0, 다르면 1**이 되는 연산으로, 한 번 더 XOR하면 (a XOR b) XOR b = a가 되는 특징이 있다. 이 역시 간단한 구현문제로 등장할 수 있다.

정리하면, 문자열 파트에서는 **기본 구현 능력**(문자열 조작, 파싱 등)을 확실히 하고, 필요시 효율적인 알고리즘(KMP 등)을 적용할 수 있도록 준비해야 한다. 스터디 후반부에는 **고난도 문자열 문제**들도 몇 개 풀어보며 실전 감각을 키울 예정이다 34 .

12. 동적 계획법 (Dynamic Programming, DP)

동적 계획법(DP)은 알고리즘에서 매우 중요한 기법으로, 복잡한 문제를 **부분 문제로 나누어 푸는 최적화 기법**이다. 핵심 아이디어는 **중복되는 부분 문제(subproblem)**를 한 번만 풀고 결과를 재활용하여 전체 문제를 효율적으로 해결하는 데 있다. 이를 위해 **메모이제이션(Memoization)** 또는 **타뷸레이션(Tabulation)**을 사용한다.

- **탑다운 동적계획법 (메모이제이션)**: 재귀를 활용한 방법으로, 함수가 자기 자신을 호출하며 문제를 풀되, 이미 푼 부분 문제는 결과를 캐시에 저장해 두고 다시 풀지 않는 방식이다. 예를 들어 피보나치 수열을 재귀로 구현하면 지수 시간 복잡도가 되지만, **memo** 딕셔너리에 `F(n)` 값을 저장해두고 이미 구한 값은 바로 반환하면 `O(n)`에 계산 가능하다 17 . 이때 재귀 호출 관계를 **문제 해결 트리(call tree)**로 그려보면, 동일한 부분이 여러 번 계산되는 것을 막았음을 알 수 있다. 메모이제이션은 구현이 비교적 쉬워 많은 DP 문제에 활용된다.

- **바텀업 동적계획법 (타뷸레이션)**: 반복문을 활용한 방법으로, 작은 부분 문제 해답들을 먼저 계산하여 표(table)에 채워나가고, 이를 이용해 점차 큰 문제 해답을 구하는 방식이다. 보통 배열이나 2차원 테이블을 만들어 초기값(base)들을 설정하고 점화식에 따라 증가하는 방향으로 채운다. 예를 들어, 피보나치 수열의 DP는 `dp[0]=0, dp[1]=1`부터 시작해서 `dp[i] = dp[i-1] + dp[i-2]`를 i=2부터 n까지 반복하여 구한다. 재귀 호출이 없어 **스택 오버플로우 위험이 없고**, 명시적으로 계산 순서를 통제하기 때문에 무한루프 걱정도 적다.

DP를 성공적으로 적용하려면 **문제에 적합한 상태 표현과 점화식**을 찾는 것이 관건이다. 몇 가지 전형적인 DP 문제를 통해 개념을 살펴보자:

- **피보나치 수 계산**: 위에서 언급한 대로 `F(n)=F(n-1)+F(n-2)` 관계를 이용해 작은 값부터 차례로 계산하면 선형 시간에 구할 수 있다. 이 예시는 DP의 기본 원리를 잘 보여준다.

- **최적 경로 문제**: 예를 들어 삼각형 구조 숫자 배열에서 꼭대기부터 내려가면서 합의 최댓값 구하기 등은 DP로 해결된다. `dp[i][j] = max(dp[i-1][j-1], dp[i-1][j]) + value[i][j]` 같은 점화식을 세워 풀 수 있다.

- **최장 증가 부분 수열(LIS)**: 주어진 수열에서 부분 수열이 **증가하는 형태**로 가장 긴 길이를 찾는 문제다. 이 역시 DP로 풀린다 [9]. `dp[i]`를 **i번째 원소를 마지막으로 하는 LIS 길이**라고 정의하고, `dp[i] = 1 + max(dp[j] for j<i if arr[j] < arr[i])`로 점화식을 만들면 `O(n^2)`에 계산 가능하다. 입력이 크면 이 방법은 느리지만, LIS는 이분 탐색으로 `O(n log n)`에 개선하는 방법도 따로 존재한다 (길이만 구한다면).

- **최장 공통 부분 수열(LCS)**: 두 문자열이 주어졌을 때 둘의 공통된 부분 수열 중 가장 긴 것을 찾는 문제이다. 이 역시 이차원 DP로 유명하다. `dp[i][j]`를 **문자열 A의 i까지와 문자열 B의 j까지 고려했을 때 LCS 길이**라고 정의하면, 문자 일치 여부에 따른 점화식으로 `O(n*m)`에 풀 수 있다. LCS는 문자열 유사도 측정 등 다양한 곳에 응용된다.

- **0-1 배낭 문제(Knapsack)**: 무게와 가치가 부여된 물건들이 있을 때, 배낭에 넣을 수 있는 최대 무게 한도 내에서 **가치 합의 최댓값**을 구하는 문제다. 이 또한 DP의 고전 문제로, `dp[i][w]`를 **앞부터 i번째 물건까지 고려했을 때 무게 w로 담을 수 있는 최대 가치**로 정의하여 점화식을 세운다. 복잡도는 물건 수 * 무게한도로, 제한이 크면 비트마스크 DP나 최적화 기법이 필요하다.

이처럼 DP 문제들은 **상태 정의** (배열 인덱스, 부분 특성 등)와 **상태 전이(점화식)**를 세우는 것이 핵심이다. 연습을 통해 이 모델링 능력을 길러야 한다. 또한 DP 구현 시에는 배열 인덱스 범위나 초기값 설정에 신경 써야 한다. 경우에 따라 **메모리 절약 기법**으로 2차원 DP를 1차원 배열 하나로 공간 최적화하기도 한다. 예를 들어, 이전 행만 참조하면 되는 DP라면 1차원 배열을 쓰고 뒤에서 앞으로 업데이트하는 식으로 공간을 `O(n)`으로 줄일 수 있다.

마지막으로, DP는 정답만 구하면 되는 문제가 많지만, **최적해 구성 추적**을 물어보는 경우도 있다. 이때는 DP 테이블과 별도로 **추적용 포인터**나 경로 저장을 해야 한다. 예를 들어 LCS 문제에서 LCS 문자열 자체를 구하려면, dp 표를 역추적(backtrack)하는 방법을 쓴다. 이러한 기법까지 우리 스터디에서 깊게 다루지는 않을 수 있으나, 필요한 경우 소개할 예정이다.

동적 계획법은 분명 어렵지만, 한 번 개념을 잡으면 알고리즘 실력이 크게 향상되는 분야다. 스터디 중반부에 DP를 집중적으로 다루며, 다양한 난이도의 문제 (예: 계단 오르기, LIS, LCS, 문자열 편집거리 등)를 풀어볼 것이다 [35]. 또한 **메모이제이션 vs 타뷸레이션**의 선택, **점화식 도출 연습**, **상태 수 감소(메모리 압축)** 등의 주제도 함께 고민해보자.

13. 그래프와 트리 (Graph & Tree)

그래프(Graph)는 현실 세계의 관계나 연결성을 모델링하는 강력한 데이터 구조이다. 그래프는 **정점(vertex)**의 모음과 이들을 잇는 **간선(edge)**의 모음으로 구성된다. 예를 들어, 도시 간 도로망, 친구 관계망, 웹 링크 구조 등이 그래프로 표현될 수 있다. **트리(Tree)**는 **사이클이 없는 그래프**의 일종으로, 특별히 **계층적 구조**(부모-자식 관계)를 가지는 경우를 말한다. 트리는 정점 N개에 간선 N-1개를 가진 연결 그래프로 볼 수도 있다. 여기서는 트리와 그래프의 기본 개념과 **탐색 알고리즘**을 다룬다.

- **트리의 특징**: 트리는 루트(root)를 가진 계층 구조로 표현되기도 하고, 부모-자식 관계를 가지므로 **재귀적 속성**이 강하다. 이진 트리(Binary Tree)는 각 노드가 최대 두 명의 자식을 가지는 트리이며, 이진탐색트리(BST)는 왼쪽 자식 < 부모 < 오른쪽 자식의 키 크기 관계를 만족하는 특별한 트리이다. 트리는 **연결 리스트**와 같이 재귀적 정의로 이어진 자료구조로 간주할 수 있어, 재귀 알고리즘에 익숙해지기 좋은 구조다. 또한 **트리의 순회(Tree Traversal)** 방법으로 **전위순회(pre-order)**, **중위순회(in-order)**, **후위순회(post-order)**가 있다. 예를 들어 이진트리에서 전위순회는 "자기 자신 -> 왼쪽 서브트리 -> 오른쪽 서브트리" 순으로 방문한다. 중위순회는 BST의 경우 오름차순으로 노드를 방문하게 되는 특징이 있다.

- **트리의 구현과 표현**: 트리를 컴퓨터에서 표현하는 방법에는 **연결 리스트(포인터)** 방식과 **배열 인덱스** 방식이 있다. 포인터 방식은 각 노드 구조체가 자식에 대한 포인터를 들고 있는 형태로, C/C++의 트리 구현에 흔하다. 배열 방식은 특히 **완전 이진 트리**의 경우 편리한데, 배열의 인덱스를 노드로 삼고 `left_child = index*2`, `right_child = index*2+1` (또는 0-index라면 2+1, 2+2)로 표현한다. 힙이 그 대표적인 사례다. 문제에 따라 트리를 **인접 리스트** 형태(각 노드의 자식들을 리스트로)로 표현하거나, 부모를 가리키는 배열을 사용할 수도 있다. 트리에서 **최소 공통 조상(LCA, Lowest Common Ancestor)**를 찾아야 하는 등의 문제는 트리를 다루는 전형적인 알고리즘 중 하나인데, 이를 빠르게 구하기 위해 희소 테이블이나 오일러 투어 + RMQ 등 알고리즘이 존재하지만, 이는 고급 주제로 시간 되면 다뤄볼 수 있다.

- **그래프의 표현**: 그래프를 코딩할 때는 보통 **인접 행렬(Adjacency Matrix)**과 **인접 리스트(Adjacency List)** 두 가지 표현법을 쓴다. 인접 행렬은 NxN 2차원 배열을 만들어 `(i,j)`에 간선 존재 여부(또는 가중치)를 표기하는 방법이다. 구현이 간단하고 간선 존재 검사 `O(1)`로 빠르지만, 공간 복잡도가 `O(N^2)`이라 정점 많을 때는 비효율적이다. 인접 리스트는 각 정점마다 **연결된 이웃들의 리스트**를 저장하는 방식으로, 메모리를 간선 수에 비례하여 사용하며 대부분의 상황에 더 유리하다. 정점 수 V, 간선 수 E인 그래프에서 인접 리스트는 메모리 `O(V+E)`, 인접 행렬은 `O(V^2)`를 차지한다. 우리 스터디에서는 주로 인접 리스트를 사용할 것이다 (C++에선 `vector<int> graph[MAX]`, 파이썬에선 `adj = [[] for _ in range(n)]` 이런 식). 참고로 **양방향 그래프**는 리스트에 양쪽에 서로 추가하고, **단방향 그래프**는 한쪽만 추가한다.

- **그래프 탐색: DFS와 BFS** – 그래프에서 **모든 정점들을 체계적으로 방문**하거나, **특정 조건의 정점을 찾기 위한** 기본 탐색 알고리즘으로 DFS(Depth-First Search)와 BFS(Breadth-First Search)가 있다 [36] .

- **DFS(깊이우선탐색)**: 한 경로를 따라 갈 수 있을 만큼 끝까지 가보고, 더 갈 곳이 없으면 backtracking하여 다른 경로로 탐색하는 방식이다. 구현은 재귀 함수나 스택 자료구조로 할 수 있다. DFS는 **백트래킹**과도 유사한 형태로, 모든 가능성을 탐색하거나 연결 성분을 찾는 등에 사용된다. 그래프 문제에서 **사이클 탐지**, **탐색트리 구성**, **위상정렬(사이클 없는 유향 그래프에서)** 등에서 DFS를 활용한다. 또한 2차원 미로 문제 같은 곳에서 DFS로 한 길을 쭉 따라가며 풀기도 한다. 다만 DFS는 경로가 매우 깊어질 경우 recursion stack overflow 주의와, 무한 루프에 빠지지 않도록 **방문 체크(visited)**가 필요하다. 인접 리스트 그래프에서 DFS의 시간 복잡도는 인접 행렬이든 리스트든 `O(V+E)`이다.

- **BFS(너비우선탐색)**: 시작 정점으로부터 가까운 이웃들부터 차례로 탐색하는 방식으로, 구현은 큐(queue)를 사용한다. BFS는 **최단 거리**를 찾는 데 쓰이는데 (그래프 간선 가중치가 동일할 때), 먼저 방문되는 것이 최단 경로를 통한 방문이기 때문이다. 예를 들어 **미로의 최소 이동 횟수**나 **최소 단계 변화** 문제는 BFS로 풀면 효과적이다. BFS 역시 방문 체크가 필요하며, 선입선출로 인접 노드들을 큐에 넣어 탐색한다. BFS의 시간 복잡도도 `O(V+E)`로 그래프의 크기에 선형적이다. BFS는 또한 **이분 그래프 판별** (짝수 거리로 돌아오는지 검사) 같은 응용에도 쓰인다.

그래프와 트리는 알고리즘 문제의 **꽃**이라고 할 정도로 중요하다 [37] . 단순 구현부터 복잡한 응용까지 다양하다. 예를 들어 **트리 순회 결과 복원**, **그래프에서 연결 요소 개수 찾기** (DFS/BFS로), **오일러 경로**, **사이클 찾기** 등 많은 문제가 나올 수 있다. 스터디에서는 그래프의 기초를 다지고, DFS/BFS를 확실히 이해한 뒤, 이를 활용한 문제 (토마토 문제 BFS, 영역 구하기, 단지번호붙이기 등)를 풀어볼 것이다 [38] . 또한 **트리 문제**로는 이진트리의 직경, 트리의 높이, 트리 DP 등도 소개될 수 있다.

그래프 탐색 팁: 그래프 문제를 풀 땐 **방문 배열** 초기화와 **그래프 입력 처리**를 정확히 해야 한다. 또, 문제에서 노드 번호가 1-index인지 0-index인지, 연결 그래프인지, 방향성/가중치 여부 등을 꼼꼼히 확인해야 한다. BFS에서는 큐 초기화와 방문 표시 순서, DFS에선 재귀 종료 조건 등을 유의한다.

14. 그래프 알고리즘 (최단 경로, MST, 유니온파인드 등)

그래프를 활용한 고전 알고리즘들은 코딩 테스트에서 심심찮게 등장한다. 특히 **최단 경로(Shortest Path)** 문제와 **최소 신장 트리(MST)** 문제는 자료구조와 알고리즘의 총합적인 응용이라 할 수 있다.

- **다익스트라 알고리즘 (Dijkstra)**: **가중치 그래프**에서 한 출발점으로부터 모든 정점까지의 최단 거리를 구하는 알고리즘이다. 음의 간선이 없을 때 동작하며, BFS의 가중치 버전으로 볼 수 있다. 구현은 **우선순위 큐**(min-heap)를 이용하여, 거리 가장 작은 정점을 선택 -> 인접 간선 완화(Relaxation) -> 반복하는 방식이다. 인접 리스트 그래프에서의 시간 복잡도는 `O(E log V)`이다 (우선순위 큐 삽입/삭제가 log V). 이 알고리즘을 알면 **지도 경로 찾기**, **네트워크 지연 시간** 등의 문제를 해결할 수 있다. 예를 들어, 백준 1916번 최소비용 구하기 (단일 시작-단일 도착 최단경로) 문제는 다익스트라로 풀 수 있다 [27] . 스터디에서는 다익스트라 구현을 함께 실습해 볼 것이다.

- **벨만-포드 알고리즘**: 다익스트라와는 달리 **음의 가중치**가 있어도 사용할 수 있는 최단 경로 알고리즘이다. 동적계획법 원리를 사용하여, 최대 V-1번 간선 릴랙스를 반복한다 (간선 완화를 V-1회 반복하면 최단 거리가 확정됨). 시간 복잡도는 `O(V*E)`로, 다익스트라보다 느리다. 대신 **음수 사이클**을 검출할 수 있는 장점이 있다. (V번째 완화 때도 값이 줄어들면 음수 사이클 존재) 코딩 테스트에서는 음의 사이클 판단 문제로 종종 등장한다.

- **플로이드-워셜 알고리즘**: 모든 정점 쌍 간의 최단 거리를 구하는 DP 기반 알고리즘이다. 3중 루프를 돌며 `dp[k][i][j] = min(dp[k-1][i][j], dp[k-1][i][k] + dp[k-1][k][j])` 형태로 구현되며, 시간 복잡도는 `O(V^3)`이다. V가 작을 때만 사용 가능하지만, 구현이 간단하여 (코드 몇 줄) 코테에서 **그래프 모든 쌍 최단경로** 문제가 나오면 사용된다. 예를 들어, **경로상 거쳐야 하는 특별한 노드가 있는 최단경로** 문제 등은 플로이드-워셜로 풀기도 한다.

- **최소 신장 트리(MST)**: 가중치 무방향 그래프에서 **모든 정점을 연결하면서 전체 가중치 합이 최소**가 되는 간선 집합(트리)을 찾는 문제다. 대표 알고리즘으로 **크루스칼(Kruskal)**과 **프림(Prim)**이 있다. 크루스칼은 간선을 가중치 순으로 정렬한 뒤 작은 것부터 추가하되 **사이클이 생기지 않도록** 추가하는 방식이다 [20] . 이때 사이클 여부를 빠르게 판단하기 위해 **유니온-파인드(Union-Find, Disjoint Set)** 자료구조를 사용한다. 유니온-파인드는 각 정점이 어떤 연결 컴포넌트에 속해있는지 관리하며, find 연산 (대표 찾기)와 union 연산 (합치기)이 **거의 O(1)** (엄밀히는 inverse-Ackermann 함수 시간)로 매우 빠르다. 크루스칼 복잡도는 간선 정렬 O(E log E) + 유니온파인드 연산 O(E * α(V)). 프림 알고리즘은 특정 시작점에서 출발해, **하나의 MST 집합을 점차 키워가는** 방식이다. 매 단계 현재 트리에 인접한 간선 중 최소 간선을 선택해 트리에 추가한다. 구현은 가중치 기준 우선순위 큐로 가능하며 복잡도는 O(E log V)이다. MST는 도로 건설, 네트워크 연결 등 여러 시나리오에 적용된다.

- **유니온-파인드(Disjoint Set Union)**: 위에서 언급한 자료구조로, 서로소 집합들을 효율적으로 관리한다. **초기화** 때 각 원소가 자기만의 집합을 이루고, **union(x,y)** 연산으로 두 집합을 합치고, **find(x)** 연산으로 x가 속한 집합의 대표를 찾는다. 구현은 보통 배열과 재귀로 간단히 할 수 있고, 경로 압축(path compression)과 랭크 기반 union 최적화를 적용하면 거의 O(1) 성능을 낸다. DSU는 MST 외에도 **네트워크 연결 여부** 판단, **같은 그룹 여부** 판정 문제, **여러 소속 처리** 문제 등에서 두루 쓰인다. 가령 어떤 온라인 쿼리 문제에서 "A와 B가 같은 팀인가?"를 여러 번 물을 때 유니온파인드로 효율적으로 답할 수 있다.

- **기타 그래프 알고리즘**: 그래프엔 이 외에도 많은 알고리즘이 있다. **위상 정렬(Topological Sort)**은 사이클 없는 유향 그래프(DAG)에서 정점들을 선행 관계에 맞게 나열하는 것으로, 큐와 진입차수(indegree)를 사용해 구현한다. 작업 스케줄링 등에서 쓰이며, 난이도 중하 문제로 종종 나온다. **이분 그래프 판정**은 BFS/DFS로 인접 노드에 다른 색을 칠해가며 검증할 수 있다. **네트워크 유량(Max Flow)** 알고리즘 (에드몬드-카프, Dinic 등)은 고급 주제라 본 스터디 범위는 아니지만, 관심 있는 분들은 추가로 공부하면 좋다.

그래프 알고리즘은 개념도 중요하지만 구현 연습이 특히 필요하다. 다익스트라, 유니온파인드, 위상정렬 등은 **암기할 정도로 연습**해두면 실전에서 큰 강점이 된다. 스터디에서는 가능한 한 이러한 알고리즘들을 직접 구현해 보고, 관련 문제

들도 풀어볼 계획이다 [27] [39]. 예컨대 **최단 경로 문제**로는 백준 1753 최단경로(다익스트라), **MST 문제**로는 백준 1197 최소 스패닝 트리(크루스칼), **위상정렬 문제**로는 백준 2623 음악프로그램 등을 도전해볼 수 있다.

15. 그리디 알고리즘 (Greedy)

그리디 알고리즘은 매 단계에서 **당장 가장 좋아 보이는 선택 (greedy choice)**을 함으로써 최종 해답에 도달하는 알고리즘 설계 기법이다 [40]. 탐욕법은 직관적이고 구현이 간단한 경우가 많지만, **정당성 증명**이 중요하다. 즉, 이렇게 국지 최선의 선택을 해서 최종 결과가 항상 최적이라는 보장이 있어야 한다. 많은 문제들이 그리디로 풀리지만, 일부는 그리디로 접근하면 오답이 되는 경우도 있으므로 주의가 필요하다.

그리디의 고전적인 예와 코딩 테스트 단골 유형은 다음과 같다:

- **동전 거스름돈 문제**: 지폐/동전 단위가 1, 10, 50, 100원 처럼 배수 관계일 때 가장 큰 화폐 단위부터 거슬러주면 최소 개수 동전을 사용한다. 이 문제는 그리디의 정당성이 화폐 단위의 특정 속성(큰 단위가 작은 단위의 배수)에서 비롯된다는 점에서, 조건이 바뀌면 그리디가 최적을 보장 못 할 수도 있음을 보여준다.

- **활동 선택 문제**: 일정 시간이 겹치는 회의 중 **최대로 많은 회의를 참석**하려면, **종료 시간이 가장 빠른 회의부터 선택**하는 게 최선이다. 이것도 그리디로 풀리며, 각 단계에서 가장 빨리 끝나는 것을 선택하면 남은 시간에 최대한 많은 활동을 넣을 수 있기 때문이다. 비슷하게, **작업 스케줄링**이나 **강의실 배정** 문제 변형들이 있다.

- **배낭 문제의 Greedy 버전 (Fractional Knapsack)**: 앞서의 0-1 배낭은 DP로 풀어야 하지만, 물건을 쪼갤 수 있는 분할 가능한 배낭 문제에선 **가치 대비 무게 비율**이 높은 물건부터 담는 그리디 전략이 통한다. 이것도 코딩 테스트에서 응용될 수 있다 (예: 최대 이익으로 물건 팔기 등).

- **문자열/숫자 조작 문제**: 예를 들어 **가장 큰 수 만들기** (여러 숫자 조합하여 최대 숫자 찾기) 문제나, **자리수 제거** 문제 등이 그리디로 풀린다. 프로그래머스 "큰 수 만들기" 문제 등에서, 왼쪽부터 필요한만큼 남기고 불필요한 숫자를 제거하는 탐욕적 선택을 한다.

- **허프만 코딩**: 앞서 언급한 허프만 코딩 알고리즘은 매번 가장 빈도 낮은 두 노드를 합치는 그리디 선택으로 최적 압축 트리를 만든다. 이 또한 그리디 알고리즘의 한 예로, **우선순위 큐**를 사용한다.

그리디 알고리즘을 설계할 때는 **정렬**이 자주 선행된다. 예를 들어 활동 선택 문제도 종료시간 기준 정렬 후 탐욕 선택한다 [40]. 그러므로 문제에서 "최대/최소 ~"를 요구하고 어떤 정렬 기준이 떠오른다면 그리디를 의심해 볼 만하다. 다만, 앞서 언급했듯 그리디가 항상 최적해를 주는 것은 아니므로, 반례가 없는지 생각해보고 접근해야 한다.

코딩 테스트에 자주 출제되는 그리디 예시로 **회의실 배정**, **ATM 인출 시간 최소화** (사람별 인출 시간 정렬), **저울 문제** (추 무게 측정) 등이 있다. 우리 스터디에서도 그리디 문제를 여러 개 다루어 볼 것이다 [40]. 그리디는 난이도가 천차만별인데, 간단한 건 금방 풀리지만 어려운 건 정렬 기준을 떠올리기 어렵게 꼬아놓기도 한다. 다양한 유형을 접해보고 **경험적으로 탐욕적 선택 기준을 파악**하는 능력을 기르도록 하자.

16. 기타 고급 주제 및 자료구조

위에서 다룬 내용 외에도, 알고리즘 공부를 더 확장하고 싶다면 살펴볼 만한 주제들이 있다. 11주라는 기간 내에 모두 깊게 다루긴 어렵지만, 간략히 소개하고 추가 학습 방향을 제시한다:

- **비트 마스킹(Bit Masking)**: 정수를 이진수로 보고, 이진수의 각 비트를 집합의 원소 존재 여부처럼 활용하는 기법이다. 예를 들어 0부터 2^N-1까지의 비트값으로 N개의 원소 부분집합을 표현할 수 있다. 비트연산을 사용하면 부분집합 순회, 부분집합 합산 등을 빠르게 처리할 수 있다. 또한 특정 비트 토글(XOR)이나 합치기(OR), 교집합(AND) 등의 연산으로 효과적으로 상태를 다룬다. 동적계획법에서 **비트 DP** (예: 외판원 순회 문제 TSP

DP 해결) 등에 쓰이며, 코딩 테스트에도 가끔 활용된다. 비트 연산은 **상수 시간으로 동작**하므로, 사용법에 익숙해지면 코드 최적화에 도움을 준다.

- **세그먼트 트리(Segment Tree)**: 배열 같은 시퀀스 데이터에 대하여 구간 합, 구간 최댓값 등의 **범위 쿼리**를 빠르게 처리하고자 할 때 쓰이는 트리 구조다. 세그먼트 트리는 배열을 분할하여 구간에 대응하는 노드들을 구성, **O(log n)** 시간에 쿼리를 처리한다. 예를 들어 1~N 배열에 대해 매번 임의의 구간 합을 물을 때, 세그먼트 트리를 구축하면 빠르게 답할 수 있다. 또한 특정 원소 값 업데이트도 로그 시간에 가능하다. **펜윅 트리(Fenwick Tree)** 혹은 **BIT**도 유사한 기능을 제공하는 자료구조로 구현이 간단하고 자주 쓰인다 (Fenwick은 1-indexed 배열에서 트릭으로 구간합을 관리). 세그먼트 트리는 메모리를 4N 정도 사용하며, 구현은 조금 복잡하지만, 구간 질의 문제가 나오면 거의 유일한 해결법인 경우도 많다. 나아가 세그먼트 트리에 **Lazy Propagation**을 적용하면 구간 업데이트까지 효율적으로 수행할 수 있다 (나중에 한꺼번에 업데이트 처리). 이번 스터디에서는 시간 상 깊게 다루진 않겠지만, 관심 있다면 BOJ 2042 (구간 합 구하기) 문제로 연습해보길 권한다.

- **트라이(Trie)**: 앞서 문자열 파트에서 언급한 트라이는 문자열 집합의 **전형적인 자료구조**다. 문자 단위 트리로 구현되며, 모든 문자열의 공통 접두사들이 한 경로를 공유하기 때문에 검색이나 자동완성에 효율적이다. 전화번호 목록에서 어떤 번호가 다른 번호의 접두어인지를 판별하는 문제(프로그래머스 전화번호 목록 문제) 등에서 응용된다. 구현은 각 노드가 자식 포인터(보통 26개 또는 문자셋 크기)와 종료 마크를 가지는 형태로 한다.

- **AVL/Red-Black Tree 등 균형 이진 탐색 트리**: 표준 라이브러리의 `std::set`, `std::map` (C++), TreeMap (Java) 등은 내부적으로 **자기 균형 이진탐색트리**로 구현되어 삽입/삭제/탐색이 `O(log n)`을 보장한다. 이러한 구조를 직접 구현할 일은 거의 없지만, 알아두면 좋다. **이진 검색 트리(BST)**의 삽입/삭제가 편향되면 O(n)까지 가는 문제를 해결하기 위해, 높이를 일정하게 유지하려고 노드 재배치를 하는 것이 균형 BST의 핵심이다. 대표적인 기법이 AVL 트리와 레드-블랙 트리이다. (C++ `set`은 Red-Black Tree 사용). 또한 **Splay Tree** 같은 자가조정 트리, Treap 등도 이론적으로 존재한다. 이들 내용은 매우 깊지만, 결국 프로그래밍 대회에서는 주로 라이브러리 사용으로 대체된다.

- **공간 분할 알고리즘**: 분할정복의 한 예로 **퀵셀렉트(Quickselect)** 알고리즘이 있다. 이는 퀵정렬의 partition 개념을 이용해 **배열에서 k번째 원소를 선형 평균 시간**에 찾는 알고리즘이다. 정렬하지 않고도 순위 통계치를 구할 수 있다. 또 **분할정복을 이용한 거듭 제곱** (Exponentiation by squaring)은 `O(log n)`에 거듭 제곱을 계산하는 중요한 알고리즘이며, **분할정복을 이용한 행렬 제곱**은 피보나치 n번째 수를 로그 시간에 구하는 등 응용된다.

- **오프라인 쿼리 처리**: 간혹 입력으로 여러 쿼리를 한꺼번에 받아 처리하되, 온라인처럼 순서대로가 아니라 **처리를 효율화**할 수 있는 테크닉들이 있다. 예를 들어 **MO's 알고리즘**은 쿼리들을 sqrt 분할 정렬하여 캐시 효율을 높여 쿼리당 복잡도를 감소시키는 기법이다. **CDRQ(오프라인 쿼리)**는 Union-Find를 이용해 그래프 연결 쿼리를 오프라인으로 푸는 등, 문제 성질에 따라 오프라인 처리로 시간단축을 꾀할 수 있다. 이러한 기법은 특수한 경우에 쓰이므로 이번 과정에서는 다루지 않지만, 알고만 있자.

마지막으로, **프로세스와 메모리 구조**에 대한 이해도 간접적으로 도움이 된다. 예를 들어, **함수 호출 스택과 힙**의 차이를 알면 재귀나 동적 할당시 어떤 일이 일어나는지 알 수 있고, **캐시 지역성** 개념을 알면 연속 메모리(배열) vs 링크드 구조(리스트)의 성능 차이를 예상할 수 있다. 또한 C/C++의 경우 **메모리 관리**, Python의 경우 **레퍼런스, 가비지 컬렉션** 등도 알고리즘 구현 시 고려 포인트가 될 수 있다.

以上의 고급 주제들은 시간상 심화하지 못하지만, 스터디 이후에도 관심 있는 영역을 지속적으로 공부하면 좋다. 알고리즘 학습은 끝이 없으며, 새로운 문제를 접할 때마다 또 다른 개념을 익히게 될 것이다.

지금까지 초급-중급 수준의 알고리즘 스터디 커리큘럼을 챕터별로 정리했다. 본 스터디는 **11주 이내**에 주 2회 진행으로 계획되므로, 위에 열거한 모든 내용을 깊게 다루기는 어렵다. 따라서 각 주차별로 핵심 주제를 선정하고, 관련된 대표 문제들을 함께 풀어보는 방향으로 나아갈 것이다. 예를 들어:

- 1주차: 알고리즘 소개, 복잡도, 기본 구현 연습 (수학 문제 몇 개)
- 2주차: 자료구조 기본 (스택/큐/리스트 등) + 문자열 기본
- 3주차: 브루트포스 & 백트래킹 기초 연습 (재귀 익히기)
- 4주차: 정렬, 이분 탐색, 투포인터 등 배열 알고리즘
- 5주차: 동적 계획법 입문 (피보나치, DP 테이블 기본 문제)
- 6주차: 그래프 & BFS/DFS (기초 탐색 문제들)
- 7주차: 그래프 심화 (최단거리: 다익스트라 등, MST 등 소개)
- 8주차: 그리디 알고리즘 문제 풀이 연습
- 9주차: 구현 & 시뮬레이션 유형 집중 (여러 조건 처리 연습)
- 10주차: 복습 및 중급 문제 도전 (문자열 고급, 백트래킹 응용 등)
- 11주차: 모의 코딩 테스트 진행 및 약점 보완

이러한 플랜은 유동적이며, 스터디 진행 상황에 따라 조정될 수 있다. 중요한 것은 **문제 풀이를 통한 실전 역량 향상**이다. 알고리즘 이론을 공부하면서도, 반드시 해당 주제의 문제들을 풀어봐야 체득된다. 가능한 많은 문제를 다뤄보고, 서로의 풀이를 리뷰하며 **다른 시각의 접근법**도 배워나가자. 특히 구현력(버그 없이 코딩하기)은 반복된 연습만이 살 길이니, 작은 문제라도 많이 풀어보길 권장한다.

마지막으로 강조하자면, 코딩 테스트 대비를 위해서는 **5대장 문제 유형**인 **구현(시뮬레이션)**, **완전탐색(백트래킹)**, **그리디**, **DFS/BFS(그래프 탐색)**, **DP**를 골고루 접해보는 것이 좋다 [41]. 우리 스터디 커리큘럼은 이 모든 분야를 망라하도록 구성되었다. 11주 뒤에는 스터디원 모두가 기본 알고리즘에 자신감을 갖고, 낯선 문제를 만나도 어떻게 접근할지 생각해낼 수 있게 되기를 기대한다. 함께 꾸준히 노력하며 목표를 달성해 봅시다!

---

[1] [3] 알고리즘 - 위키백과, 우리 모두의 백과사전
https://ko.wikipedia.org/wiki/%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98

[2] [13] [14] [컴퓨터과학] 알고리즘의 표현 방법 : 순서도, 의사코드(슈도코드)
https://booksr143.tistory.com/10

[4] [6] [7] [9] [10] [11] [12] [CS] APS, 시간 복잡도 정리 — codelog
https://codingdialee.tistory.com/85

[5] [15] [16] [17] [18] [19] [20] [27] [29] [30] [33] [34] [35] [36] [37] [38] [39] [40] [41] 초급-중급 알고리즘 스터디 커리큘럼 추천 (3개월)
https://dev-dain.tistory.com/155

[8] 온라인 저지 - 위키백과, 우리 모두의 백과사전
https://ko.wikipedia.org/wiki/%EC%98%A8%EB%9D%BC%EC%9D%B8_%EC%A0%80%EC%A7%80

[21] [22] [23] [24] [25] [26] [28] [자료구조] 스택(stack), 큐(queue) - 코딩하는 경제학도
https://ssocoit.tistory.com/230

[31] 문자열 매칭 알고리즘 (KMP, Z-Array) | ialy's blog
https://ialy1595.github.io/post/string-match/

[32] (C++) 문자열 검색 알고리즘 : KMP 알고리즘 - 평생 공부 블로그
https://ansohxxn.github.io/algorithm/kmp/