ChatGPT

알고리즘 스터디: 오리엔테이션 및 기본 내용 정리

강의 시작에 앞서 (Before We Begin)

알고리즘 스터디에 오신 것을 환영합니다. 먼저, 강의 도입에 앞서 몇 가지 안내 사항을 말씀드리겠습니다. 강의 중 모든 내용을 전부 다루지 못할 수도 있으니 **궁금한 점은 언제든지 질문하고 소통해 주세요.** 함께 고민하며 배우는 과정이 중요합니다. 또한, Java 언어로 실습하는 분들은 **Java 버전을 미리 선택**해 주시기 바랍니다 (예: Java 8 또는 Java 11/17 등). 사용 환경에 따라 지원되는 라이브러리가 다를 수 있으므로 본인이 사용할 Java 버전을 알고 있는 것이 좋습니다.

개발 환경 및 코드 실행

프로그래밍 문제를 풀 때 사용하는 개발 환경은 각자 다를 수 있지만, 일반적으로 다음과 같은 도구들을 활용할 수 있습니다:

- **C++:** Visual Studio 또는 Visual Studio Code 등의 에디터를 주로 사용합니다. Visual Studio IDE로 C++를 개발하면 강력한 디버깅 기능과 편의성을 얻을 수 있지만, 문제마다 새로운 프로젝트를 생성해야 하는 번거로움이 있습니다.
- **Java:** IntelliJ IDEA나 Eclipse와 같은 IDE를 많이 사용합니다. 이들 또한 프로젝트 단위로 관리되기 때문에, 여러 문제를 풀다 보면 문제별로 프로젝트가 쌓여 관리가 복잡해질 수 있습니다.
- **Python:** PyCharm과 같은 IDE를 쓰거나, 가벼운 에디터로는 VS Code 등을 선호합니다. Python은 스크립트 언어이므로 파일 단위 실행이 쉬운 편입니다.

여러 문제를 효율적으로 관리하는 방법: 알고리즘 문제를 많이 풀다 보면 문제별로 여러 파일이 생기는데, **가능하면 한 문제를 하나의 파일(또는 하나의 폴더)**로 관리하는 것이 좋습니다. 예를 들어, Baekjoon 1000번 문제를 `1000.cpp`, `1000.java`, `1000.py`와 같이 **한 폴더 내에 여러 언어 소스코드로 함께 관리**할 수도 있습니다. 이렇게 하면 동일한 문제를 C++, Java, Python 등 여러 언어로 풀더라도 한 곳에 모아둘 수 있어 관리가 편리합니다.

VS Code를 활용한 통합 환경: 저는 개인적으로 Visual Studio Code에서 C++/Java/Python **모든 언어의 컴파일 및 실행 환경**을 구성하여 사용합니다. VS Code는 가볍고 여러 언어에 대해 자유롭게 커스터마이징이 가능한 강력한 에디터입니다 [1]. 초기 세팅에 다소 시간이 걸릴 수 있지만, 한 번 설정해 두면 문제 풀이 시 빠르게 코드를 작성하고 실행할 수 있는 장점이 있습니다. 특히 VS Code의 Tasks 또는 Code Runner 확장 등을 이용하면 단일 파일을 컴파일/실행하는 것을 자동화할 수 있습니다.

- C++의 경우 컴파일 옵션이나 디버깅 설정이 조금 복잡할 수 있으므로, 설정에 어려움이 있으면 강사에게 직접 문의해주세요. 대신 한 번 설정해두면 F5 키 하나로 현재 편집 중인 C++ 코드를 빌드하고 실행할 수 있습니다.
- Java와 Python은 비교적 설정이 간단합니다. Java는 `javac`로 컴파일 후 `java`로 실행하거나, VS Code의 Java Extension Pack을 이용하면 자동 빌드/실행이 가능합니다. Python은 별도 컴파일 과정 없이 `python` 인터프리터로 바로 실행되므로, VS Code 터미널 또는 실행 버튼을 통해 손쉽게 실행할 수 있습니다.

문제 폴더/파일 자동 생성: 많은 문제를 풀다 보면 일일이 폴더를 만들고 소스 파일을 생성하며, 필요한 경우 입력 파일까지 준비하는 과정이 번거롭게 느껴질 수 있습니다. 이럴 때는 **자동화 스크립트**를 작성하여 문제 폴더 및 파일을 템플릿으로 생성하는 방법도 고려할 수 있습니다. 예를 들어 문제 번호를 입력하면 `~/boj/1000/Main.cpp` 또는 `Main.java`, `main.py` 등을 자동으로 만들어주는 스크립트를 사용하면 편리합니다. 파일 구조와 템플릿을 한 번 정해두면, 새 문제 시작 시 시간을 절약할 수 있습니다.

요약하면, 자신에게 익숙한 IDE나 에디터를 사용하되 **여러 문제를 효율적으로 관리할 수 있는 방식**을 찾는 것이 중요합니다. 프로젝트 단위 관리가 번거롭다면 VS Code 같은 가벼운 도구에서 **단일 파일 위주로 코딩하고 실행하는 환경**을 구축해보세요. 필요하다면 자동화 도구의 도움을 받아 루틴 작업을 줄이고 코딩 자체에 집중할 수 있도록 합시다.

학습 목표 및 방향

이번 스터디의 **궁극적인 목표**는 **기업 코딩테스트에 대비한 알고리즘 및 프로그래밍 실력 향상**입니다. 이를 달성하기 위해 다음과 같은 세부 목표와 방향성을 갖고 학습할 예정입니다:

- **문제 해결 능력 향상:** 다양한 알고리즘 문제를 풀면서 문제 분석과 해결 능력을 키웁니다. 단순히 답만 맞추는 것이 아니라, **효율적인 알고리즘**을 고민하고 구현하는 연습을 합니다.
- **프로그래밍 언어 숙달:** C++14, Java 17, Python3 세 가지 언어를 모두 다룰 예정입니다. 각 언어의 문법과 특성을 이해하고 문제에 맞는 언어를 선택하여 활용하는 연습을 합니다. 3개월의 단기 집중 학습을 통해 **그래프 탐색, 최단 거리/최소 비용 문제**까지 다룰 **중급자 수준**을 목표로 합니다.
- **코드 리딩 및 리뷰 경험:** 알고리즘 실력 향상에는 다른 사람이 작성한 코드를 읽어보는 것도 큰 도움이 됩니다. 필요에 따라 **다른 수강생들의 코드**를 함께 리뷰하거나 참고하면서, **타인의 코드를 읽고 이해하는 능력**을 키워봅시다. 이는 실제 개발 직무에서 코드 리뷰 문화를 간접 체험하는 기회도 될 것입니다.
- **사고력과 컨텍스트 향상:** 문제를 해결하는 과정을 통해 **논리적인 사고력**을 기르고, 복잡한 문제도 작은 단위로 나누어 해결하는 **문제 분해 능력**을 강화합니다. 또한 매 문제마다 배운 개념들을 누적시켜 점진적으로 더 어려운 문제로 **생각의 컨텍스트를 확장**해 나갑니다.
- **보조 목표 (부수적인 역량):**
- 버전 관리 툴 사용: 코드 작성을 연습하면서 **Git과 GitHub** 사용에도 익숙해지도록 합니다. 예를 들어 본인의 코드 저장소를 만들어 매 문제 풀이를 commit하고 관리하면, 나중에 자신의 성장을 돌아보거나 협업 시에도 도움이 됩니다.
- 코딩 컨벤션 준수: C++, Java, Python 각 언어별로 권장되는 **코딩 스타일/컨벤션**이 있습니다. 예를 들어 Python의 PEP8 스타일 가이드, Java의 카멜케이스/파스칼케이스 규칙, C++의 네이밍 규칙 등이 있는데, 깨끗하고 일관된 코드를 작성하는 습관을 들이겠습니다. 이는 코드 가독성과 유지보수성에 매우 중요합니다.
- 공식 문서 읽는 습관: 새로운 함수나 라이브러리를 사용할 때 구글에서 찾은 **스니펫 코드에만 의존하지 말고 공식 문서(Docs)를 참조**하는 습관을 권장합니다. 예를 들어 Python에서 `sort` 함수를 쓴다면 Python 공식 문서에서 해당 함수의 동작과 복잡도를 확인해보는 식입니다. 문서를 통해 얻은 지식을 바탕으로 하면 더 깊은 이해와 신뢰할 수 있는 정보를 얻을 수 있습니다.

요약하면, 우리는 알고리즘 문제 해결력을 키우면서 부가적으로 실무에 필요한 개발 역량(버전 관리, 코드 스타일, 문서 활용 등)까지 함께 향상시키는 것을 지향합니다. **꾸준함과 적극성**을 가지고 3개월간 달려본다면 분명 많은 성장이 있을 것이라 기대합니다.

알고리즘이란 무엇인가?

스터디를 본격적으로 시작하기에 앞서, **알고리즘(Algorithm)**의 개념에 대해 간단히 짚고 넘어가겠습니다. 흔히 코딩을 처음 접하면 알고리즘이라고 하면 곧바로 떠올리는 것이 "코드"일 텐데요. 하지만 **알고리즘의 사전적 정의**를 보면 코드 그 자체라기보다는 **어떤 문제를 해결하기 위한 명확한 일련의 단계**를 의미합니다 [2]. 즉, **알고리즘은 특정 문제를 풀기 위한 제한된 개수의 잘 정의된 지시사항들의 모음**이라고 할 수 있습니다.

> **알고리즘**: "어떤 문제들을 해결하거나 **계산을 수행하기 위해 단계적으로 나열된 유한 개수의 명령어들**" [2]

중요한 것은, **알고리즘은 곧 효율성**과 연결된다는 점입니다. 현실에서 우리가 관심 갖는 알고리즘들은 **어느 정도 실용적이고 효율적인 것들**입니다 [3]. 무작정 동작하기만 하면 되는 것이 아니라 **얼마나 빠르고 자원을 적게 써서 동작하느냐**가 중요합니다. 초보자들이 흔히 하는 실수 중 하나는 효율성을 깊게 고민하지 않고 문제를 풀기 시작하는 것입니다. 하지만 좋은 알고리즘을 만들기 위해서는 적절한 **자료구조의 선택**, 그리고 경우에 따라 **수학적인 모델링과 분석**이 필요합

니다 [4]. 알고리즘은 단순한 코드 덩어리가 아니라, **문제를 해결하는 아이디어와 논리의 집합**이며, 그 **성능(시간, 공간적인 효율)**까지 고려된 것이라고 할 수 있습니다.

좋은 알고리즘이란 무엇인가? 일반적으로 다음 두 가지 자원을 얼마나 효율적으로 사용하는지가 기준이 됩니다: - **시간(Time):** 알고리즘이 문제를 해결하는 데 걸리는 **실행 시간** 또는 **연산 횟수**. (시간 복잡도로 분석) - **공간(Space):** 알고리즘이 사용하는 **메모리 공간**의 양. (공간 복잡도로 분석)

이 두 가지를 줄이는 것이 알고리즘 효율성의 핵심인데, 때로는 두 목표가 상충하기도 합니다. (예: 시간을 줄이기 위해 메모리를 더 쓰거나, 그 반대) 따라서 문제에 따라 **시간과 공간의 균형**을 고려한 최적의 알고리즘을 찾는 것이 중요합니다.

결론적으로, **알고리즘**이란 어떤 문제를 해결하는 단계적 방법이며, **좋은 알고리즘**이란 **정확하고 효율적(빠르고 경제적)**인 방법을 말합니다. 앞으로 진행할 스터디에서는 다양한 알고리즘을 배우고 구현하겠지만, 항상 그 알고리즘의 **복잡도(효율)**를 함께 분석하고 고민하는 습관을 들이도록 하겠습니다. 이렇게 하면 무작정 코딩하는 것이 아니라, 한 단계 깊이 있는 문제 해결 능력을 기르게 될 것입니다.

알고리즘 문제 풀이 플랫폼 활용

효율적인 학습을 위해서는 혼자서만 고민하기보다는 **온라인 judge 플랫폼**을 활용하여 꾸준히 연습하는 것이 좋습니다. 여기서는 대표적인 알고리즘 문제 풀이 사이트들과 활용법을 소개합니다.

백준 온라인 저지 (Baekjoon Online Judge)

백준 온라인 저지(BOJ)는 국내에서 가장 인기 있는 알고리즘 문제 풀이 사이트 중 하나로, **수천 개의 문제와 자동 채점 시스템**을 제공합니다 [5] [6]. 다음은 백준의 주요 특징과 활용 방법입니다:

- **다양한 문제와 분류:** 백준에는 쉬운 단계의 출력 문제부터 시작해서, 자료구조/알고리즘 분류별로 매우 다양한 문제가 있습니다. 상단 메뉴의 "문제 > 알고리즘 분류"에서 그래프, 동적계획법, 정렬 등 **원하는 알고리즘 유형의 문제를 모아 볼 수** 있습니다 [7]. 또한 ICPC 등 **각종 대회 기출 문제**도 모아서 풀어볼 수 있습니다 [8].
- **온라인 채점 및 풀이 공유:** 풀고 싶은 문제를 선택해서 코드를 제출하면, 백준 온라인 채점 시스템이 즉시 채점해 줍니다. 정답일 경우 **내 제출 코드가 "맞았습니다!!"**로 표시되고, 오답이면 어떤 케이스에서 틀렸는지 알려줍니다. 또한 자신이 맞힌 문제에 한해 **다른 사람들이 공개한 소스 코드**를 열람할 수 있어 참고가 가능합니다 [9]. 실행 시간이나 메모리 사용량이 적은 코드 위주로 살펴보며 배울 점을 찾는 것도 도움이 됩니다 [10].
- **그룹 및 문제집 기능:** 백준에서는 **그룹(Group)**을 만들어서 특정 사용자들과 함께 문제 풀기 할 수 있습니다 [11]. 그룹 내에서 **문제집**을 만들어 줄차적으로 문제를 풀게 하거나, **그룹 랭킹**을 통해 경쟁을 할 수도 있습니다 [12] [13]. 우리 스터디도 그룹 기능을 활용해 문제가 공유될 예정입니다. 그룹에 가입하면 해당 **문제 리스트(문제집)**를 차례로 풀어나가시면 됩니다 [14].
- **단계별로 풀어보기:** 백준의 유명한 기능 중 하나는 **"단계별로 풀어보기"**입니다. 이것은 프로그래밍 입문자를 위해 난이도 순으로 문제를 풀며 학습할 수 있도록 1단계부터 여러 단계를 마련해 둔 것입니다. 2025년 2월 기준으로 총 **57단계**까지 문제가 준비되어 있으며 [15], 기본 문법부터 자료구조, 알고리즘까지 폭넓게 다룹니다. **1단계부터 순서대로 30~35단계 이상**까지 성실히 풀어본다면, 일반적으로 **중급자 수준**의 문제 해결 능력을 갖추게 된다고 합니다 (물론 개인차는 있습니다). 단계별 문제를 풀면서 부족한 개념을 보완하고, 난이도를 서서히 높여가는 방식으로 실력을 쌓을 수 있으니 적극 활용해 보세요.
- **언어 설정:** 백준에서는 여러 프로그래밍 언어를 지원합니다. 설정 > 언어 메뉴에서 자신이 사용할 언어를 선택해두면 문제 제출 시 언어 선택이 편리합니다. 또한 한 문제를 여러 언어로 푸는 것도 가능하므로, 도전 삼아 동일한 문제를 C++/Java/Python으로 모두 풀어보는 것도 권장됩니다.
- **랭킹 및 경쟁 요소:** 백준은 솔브드(solved.ac)와 연동된 **티어(Tier)** 시스템을 갖추고 있어서, 문제를 풀면 브론즈부터 플래티넘까지 등급이 올라가는 **게이미피케이션 요소**가 있습니다 [16]. 또한 학교나 회사 이메일로 등록하면 **소속별 랭킹**을 확인할 수 있어 친구나 동료와 경쟁하는 재미도 있습니다 [12]. 이러한 요소들이 동기부여가 되어 꾸준한 문제 풀이에 도움이 됩니다.

요약하자면, **백준 온라인 저지**는 많은 양의 문제와 체계적인 분류, 그리고 경쟁 요소까지 갖춘 훌륭한 연습 플랫폼입니다. 우리 스터디에서도 백준 그룹과 문제집을 통해 진도를 나갈 예정이니, 아직 회원가입을 하지 않았다면 먼저 가입하고 본인의 아이디를 알려주세요. 앞으로 풀 문제와 진도는 그룹 문제집으로 공지하겠습니다.

Solved.ac (솔브드)

solved.ac은 백준 온라인 저지와 연동되는 부가 서비스로, **문제 난이도 정보와 추가 학습 도구**를 제공합니다. 백준 아이디로 solved.ac에 접속하여 연동하면, 백준 문제 리스트에 난이도(Tier)가 표시되고, 풀었던 문제 통계도 볼 수 있습니다 [17]. Solved.ac의 주요 기능은 다음과 같습니다:

- **난이도 및 티어 시스템:** Solved.ac은 백준의 방대한 문제들에 사용자 투표 기반 **난이도 평점**을 매겨서 브론즈~레벨의 **티어**로 구분해 둡니다 [18] [19]. 이를 통해 지금 풀만한 적정 난이도의 문제를 찾기가 쉬워집니다. 백준에서 문제를 여러 개 풀어 등급을 올리고 싶다면 solved.ac과 반드시 연동해야 합니다 [17]. 연동 후 문제를 풀면 내 티어가 계산되고 프로필에 반영됩니다.
- **CLASS 기능:** Solved.ac에는 알고리즘 학습 진도를 나타내는 **CLASS** 체계가 있습니다. 이것은 일종의 **큐레이션된 문제 셋**으로, Class 1, 2, 3... 숫자가 높아질수록 더 어려운 알고리즘 문제들로 구성되어 있습니다 [20]. 각 클래스별로 필수 문제를 일정 수 이상 풀면 다음 클래스 자격을 얻는 방식이며, 이는 일종의 알고리즘 **자격증 취득** 같은 느낌으로 동기부여가 됩니다 [20]. 초심자라면 Class 1부터 순차로 풀면서 알고리즘 문제 해결에 필요한 핵심 주제들을 차근차근 익힐 수 있습니다.
- **추천 문제 및 랜덤 마라톤:** Solved.ac 메인 페이지에는 **랜덤 마라톤**이라는 섹션이 있어 매주 무작위 문제를 추천해 줍니다 [21]. 난이도 범위는 점점 상승하도록 되어 있어 꾸준히 풀면 실력이 향상됩니다. (물론 무작위이다 보니 가끔 생소한 종류의 문제가 나올 수 있습니다.) 랜덤 마라톤 이외에도 사용자들이 많이 푼 문제나 오늘의 문제 등을 참고하여 문제를 고를 수도 있습니다.
- **스트릭(Streak):** Solved.ac에서는 **연속 해결 일수**를 스트릭으로 표시해 줍니다 [22]. 마치 GitHub 잔디처럼, 매일 문제를 풀면 스트릭 일수가 올라가고, 1일이라도 건너뛰면 초기화됩니다. 연속 100일 이상 풀면 특별한 프로필 배경을 주는 등 작은 보상도 있어, 가능하면 **"하루 한 문제"**라도 꾸준히 풀도록 재미 요소를 부여합니다 [22].
- **랭킹 및 경쟁:** Solved.ac 자체 랭킹 페이지에서 전체 사용자 중 내 순위를 볼 수 있고, 학교별/회사별 랭킹도 제공됩니다. 또한 **라이벌 기능**이 있어 친구나 경쟁하고 싶은 유저를 설정해두면 상대와의 문제 해결 수나 레이팅을 비교하면서 동기부여를 얻을 수 있습니다 (해당 기능은 solved.ac 웹에서 프로필 설정으로 가능합니다). 이러한 **랭킹 & 라이벌 기능**을 통해 서로 자극을 주고받으며 재미있게 실력을 키울 수 있습니다.

정리하면, **solved.ac**은 백준을 **더 체계적이고 재미있게 활용**할 수 있도록 도와주는 사이트입니다. 알고리즘 입문자 분들은 우선 백준의 "단계별로 풀기"나 solved.ac의 CLASS 문제들을 활용해 기본기를 다지고, 이후에는 solved.ac에서 제공하는 티어/난이도 정보를 참고하여 **자신의 약점 유형**을 찾아 풀어보는 식으로 학습을 진행하면 좋습니다 [23]. 특히 **매일 꾸준히 푸는 습관**을 들이면 실력 향상과 티어 상승에 큰 도움이 되니, Solved.ac 스트릭 기능을 적극 활용해 봅시다 [23].

프로그래머스 (Programmers)

프로그래머스는 국내 IT기업 코딩테스트 대비에 특화된 플랫폼입니다. 카카오, 라인 등 여러 기업들이 프로그래머스를 통해 공개 코딩테스트나 사전 과제를 진행해왔기 때문에, 실전 연습으로 많이 활용됩니다. 주요 특징은 다음과 같습니다:

- **코딩 테스트 연습 문제:** Programmers에는 난이도별 코딩 테스트 기출/유형 문제가 모여 있습니다. **Level 1~5**로 난이도를 표시하고 있어서, 초보자는 Level 1부터 차근차근 풀어볼 수 있습니다. 문제 스타일이 기업 출제 경향과 유사하기 때문에 **취업을 준비하시는 분들**에게 유용합니다. (예: 완전탐색, 해시, 스택/큐, 정렬, DFS/BFS 등의 섹션으로 문제가 나뉘어 있음)
- **온라인 테스트 환경:** 각 문제에는 웹상에서 코드를 작성하고 실행해볼 수 있는 편집기와 **채점 시스템**이 있습니다. 일부 문제는 라이브러리 사용의 제한 등이 있을 수 있으나 대부분 Python, Java 등으로 자유롭게 구현 가능

합니다. 또한 코드 실행 시 여러 **예제 테스트케이스**로 미리 검증해볼 수 있어, 채점 전에 충분히 테스트할 수 있습니다.
- **SQL 등 다양한 카테고리:** 알고리즘 문제뿐만 아니라 **SQL 문제**도 있어서 데이터베이스 쿼리 연습도 가능합니다. (일부 기업은 SQL 테스트도 병행하므로 SQL 공부도 해두면 좋습니다.)
- **기업 연계 채용 정보:** Programmers에서 제공하는 문제를 풀다 보면 해당 문제를 출제한 회사 정보를 볼 수도 있고, 상시 채용 연계 문제가 올라오기도 합니다. 물론 이것이 직접 취업으로 이어지지는 않겠지만, **실제로 기업들이 어떤 문제를 출제하는지 감을 잡는 용도**로 활용하면 좋습니다.

우리 스터디의 기본 문제풀이 플랫폼은 주로 **백준 및 solved.ac**가 되겠지만, 부차적으로 프로그래머스의 문제들도 참고할 수 있습니다. 특히 코딩테스트가 가까워지면 프로그래머스 고득점 Kit 문제나 기출 문제들을 풀어보는 것을 권장드립니다.

그 외 추천 플랫폼

- **삼성 SW Expert Academy (SWEA):** 삼성 공채 대비 알고리즘 사이트로, 삼성전자를 비롯한 계열사 코딩테스트 문제가 연습용으로 제공됩니다. 다만 전용 IDE(웹 or Eclipse 플러그인 등)를 사용하며, 문제마다 파일 입출력 형식이 정해져 있는 등 환경이 조금 다릅니다. 삼성 대비를 염두에 둔다면 한 번쯤 들어가 보는 것도 좋습니다.
- **Softeer:** 현대자동차그룹에서 운영하는 알고리즘 문제 사이트로, 실제 현대공채 코딩테스트도 이 플랫폼에서 보았습니다. 자동차 도메인 관련 문제 등 독특한 문제도 있습니다. UI나 사용법이 백준과 유사하니 필요하면 활용해 보세요.
- **CodeTree:** 비교적 최근 떠오르는 사이트로, 국내 기업 코테나 대회 문제를 다수 보유하고 있습니다. **메이커스 코딩 테스트** 등 이벤트도 열리고, UI도 직관적이라 연습에 도움이 됩니다.

정리: 여러 플랫폼이 있지만, **핵심은 꾸준함**입니다. 결국 어느 사이트에서든 **매일 한두 문제씩이라도 풀면서 감각을 유지**하는 것이 중요합니다. 백준/solved.ac을 메인으로 활용하되, 때때로 프로그래머스나 기타 플랫폼 문제도 풀며 다양한 유형에 익숙해지도록 합시다.

앞으로 스터디 진행 중에 적절한 플랫폼의 적절한 문제들을 골라 드릴 예정이며, 사용법에 익숙하지 않은 부분이 있으면 함께 공유하도록 해요.

시간 복잡도와 공간 복잡도

이제부터는 본격적인 **알고리즘 이론**의 핵심 주제들을 다뤄보겠습니다. 우선 알고리즘의 성능을 논할 때 빠지지 않는 개념인 **시간 복잡도(Time Complexity)**와 **공간 복잡도(Space Complexity)**에 대해 알아보겠습니다 [24] .

시간 복잡도 (Time Complexity)와 공간 복잡도 (Space Complexity)란?

시간 복잡도란 알고리즘의 입력 크기(n)가 증가할 때, 해당 알고리즘의 연산(작업) 횟수가 어떻게 증가하는지**를 나타내는 개념입니다** [25] **. 일반적으로 알고리즘 코드에서 기본적인 한 줄 연산(덧셈, 대입, 비교 등)을 1개의 연산으로 보고, 입력 크기에 따라 이런 연산이 몇 번 실행되는지를 함수 형태로 표현합니다. 예를 들어 입력 크기가 n일 때 연산 횟수가 $T(n) = 3n^2 + 5$와 같이 표현될 수 있습니다. 핵심은** 가장 높은 차수의 항**이 성능에 미치는 지배적인 영향인데, 이는 뒤에서 점근적 표기법을 통해 다룰 것입니다.

공간 복잡도는 알고리즘을 수행하는 데 필요한 메모리 공간의 양을, 입력 크기 n에 대한 함수로 표현한 것입니다 [26] . 예를 들어 배열 크기 n에 비례하여 추가 메모리를 사용하는 알고리즘은 공간 복잡도가 $O(n)$으로 표현될 수 있습니다. 흔히 공간 복잡도에서는 알고리즘 실행 중 필요한 **추가 메모리**(Auxiliary Space)에 집중합니다 (입력 자체가 차지하는 공간은 제외하거나 별도로 언급).

알고리즘을 평가할 때 시간과 공간 복잡도를 모두 고려해야 하지만, 일반적으로 코딩 테스트나 문제 풀이에서는 **시간 복잡도**를 더 중요한 요소로 여깁니다. 이는 온라인 저지에서 주어진 제한 시간 내에 알고리즘이 수행되어야 하기 때문입니

다. 물론 임베디드 환경이나 대용량 데이터 처리에서는 공간 복잡도도 매우 중요하지만, 우리의 문제 풀이 맥락에서는 우선 시간 복잡도에 중점을 둘 것입니다.

점근적 표기법 (Asymptotic Notation) – Big-O 등 표기

점근적 표기법은 시간/공간 복잡도를 **간단하게 표기**하기 위한 수학적 도구입니다. 앞서 언급한 $T(n) = 3n^2 + 5$ 같은 함수가 있다고 하면, 점근적 표기법을 사용하면 이를 **입력이 커질 때의 대략적인 성장 양상**으로 표현할 수 있습니다 [27]. 복잡한 다항식이나 식을 쓰는 대신, **가장 의미있는 성장 차수만 남겨** 표현하는 것이죠. 주로 사용하는 표기법에는 다섯 가지가 있습니다 [28]:

- **Big-O 표기법 (O)** – **점근적 상한**: 어떤 알고리즘의 **최악의 경우 시간 복잡도**를 상한으로 표현합니다. 예를 들어 $T(n) = 3n^2 + 5$라면 $T(n) = O(n^2)$로 표기합니다. Big-O는 가장 많이 쓰이는 표기법으로, 흔히 그냥 "이 알고리즘은 $O(n^2)$이다"처럼 알고리즘의 복잡도를 표현할 때 사용합니다 [29]. ($O(n^2)$라는 말은, n이 충분히 크면 $T(n)$이 n^2에 비례하여 증가한다는 의미입니다.)
- **Big-Ω 표기법 (Ω)** – **점근적 하한**: 알고리즘의 **최선의 경우** 혹은 **하한선**을 나타냅니다. 예를 들어 선택 정렬의 비교 연산 횟수는 최선도 n^2에 비례하므로 $\Omega(n^2)$입니다. 많이 쓰이지는 않지만, 이론적으로 알고리즘의 최소 복잡도를 논할 때 사용합니다.
- **Big-Θ 표기법 (Θ)** – **점근적 동일**: 상한과 하한이 모두 같은 경우를 말합니다. 만약 어떤 알고리즘이 $O(n)$이면서 동시에 $\Omega(n)$이라면 $\Theta(n)$이라고 표현합니다. 즉 **성장 양상이 정확히 n에 비례**한다는 뜻입니다 [30]. 보통 평균적인 복잡도가 상한과 같은 차수일 때 Θ로 표현하기도 합니다.
- **Little-o 표기법 (o)** – **엄격한 상한**: Big-O와 유사하지만, 엄격히 상한보다 작음을 나타냅니다. 예를 들어 $T(n) = 3n$은 $o(n^2)$입니다 (다항식 차수가 더 낮으므로). 알고리즘 분석에서는 드물게 쓰입니다.
- **Little-ω 표기법 (ω)** – **엄격한 하한**: Big-Ω의 엄격한 버전입니다. 이 또한 알고리즘 분석 실무에서는 거의 안 쓰입니다.

이 중 **가장 압도적으로 많이 사용되는 것은 Big-O 표기법**입니다 [29]. 실무적으로 우리가 관심 갖는 것은 "이 알고리즘이 **얼마나 느릴 수 있는가(최악의 복잡도)**"이기 때문입니다. 그래서 보통 **복잡도**를 말하면 Big-O를 가리키는 경우가 많습니다. 코딩 테스트에서도 문제 풀이 해설에 "이 풀이는 $O(n \log n)$입니다" 처럼 Big-O로 표현합니다.

앞으로도 별도 언급이 없으면 **시간 복잡도 = Big-O 기준의 최악 시간 복잡도**로 이해하시면 됩니다. 공간 복잡도도 $O(n)$, $O(1)$ 이런 식으로 Big-O로 표현할 것입니다.

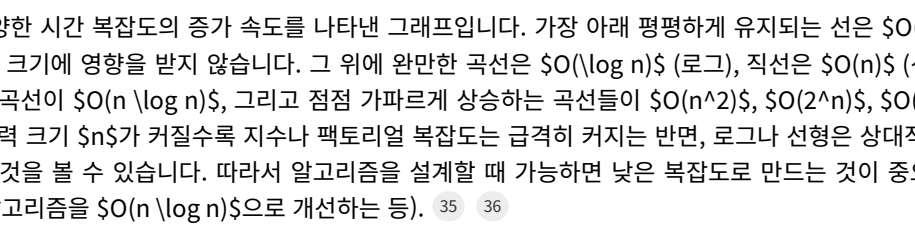
주요 시간 복잡도 클래스와 비교

알고리즘의 시간 복잡도를 흔히 **몇 가지 등급**으로 나눠볼 수 있습니다. 아래에 자주 등장하는 시간 복잡도 종류를 나열하고, 어떤 상황에서 나타나는지 간단한 예와 함께 살펴보겠습니다 [31] [32]:

- **$O(1)$ – 상수 시간:** 입력 크기와 무관하게 **항상 일정한 시간**이 걸리는 경우입니다. 예를 들어 배열의 첫 번째 원소를 반환하는 작업은 입력이 10개든 100만 개든 한 번 접근하면 끝나므로 $O(1)$입니다. 상수 시간 복잡도는 **가장 이상적**인 경우로, 입력이 커져도 수행 시간이 변하지 않습니다.
- **$O(\log n)$ – 로그 시간:** 입력 크기의 로그에 비례해 늘어나는 복잡도입니다. **이진 탐색(binary search)**이 대표적 예인데, 입력이 커질수록 탐색 범위를 절반씩 줄여나가므로 필요한 비교 횟수가 $\log_2 n$ 정도로 증가합니다. 로그 시간은 입력 2배 증가에 대해 연산 횟수가 한 번 더 늘어나는 식이므로 **상당히 효율적**인 축에 속합니다.
- **$O(n)$ – 선형 시간:** 입력 크기에 **정비례**해서 실행 시간이 증가하는 경우입니다. 예를 들어 길이 n인 배열의 모든 원소를 한 번씩 출력하는 알고리즘은 n개의 출력 연산이 필요하므로 $O(n)$입니다. 선형 시간 알고리즘은 대체로 루프 한 번 도는 구조이며, 입력이 커지면 그만큼 시간도 늘어납니다.
- **$O(n \log n)$** – 선형 로그 시간 **(또는** 준선형 시간**):** n에 $\log n$을 곱한 형태로 증가하는 경우입니다. 주로 **효율적인 정렬 알고리즘**들이 $O(n \log n)$ 복잡도를 가집니다 (퀵정렬 평균, 병합정렬 등). 이 복잡도는 선형보다는 느리지만, n^2보다는 훨씬 효율적입니다. 예를 들어 $n=1000$일 때 $\log_2 1000 \approx 10$ 정도이므로 $1000 \times 10 = 10000$ 수준의 연산으로 해결 가능한 셈입니다.

- **$O(n^2)$ – 이차 시간:** 실행 시간이 입력의 제곱에 비례해서 증가합니다. 흔히 **중첩된 두 개의 루프**를 사용하면 $O(n^2)$이 됩니다. 예를 들어 단순한 버블 정렬이나 이중 for문으로 모든 쌍을 비교하는 알고리즘 등이 이에 해당합니다. n^2 알고리즘은 n이 조금만 커져도 급격히 연산량이 증가하기 때문에, n이 만 단위만 되어도 10^8 수준(100 million)의 연산을 수행해야 합니다. (일반적으로 1억번 연산은 C++ 기준 대략 1~2초 정도로 볼 수 있습니다.)
- **$O(n^3)$ – 삼차 시간:** 이중 루프보다 하나 더 중첩된 **3중 루프** 구조 등에서 나타나며, 연산이 입력 크기의 세 제곱에 비례합니다. n이 1000만 넘어가면 사실상 현실적으로 수행이 불가능한 느린 복잡도입니다. ($1000^3 = 10^9$ 연산은 수십 초 이상 걸릴 수 있음)
- **$O(2^n)$ – 지수 시간:** 입력 크기가 커질 때 **지수적으로 (두 배, 두 배로) 시간 증가**를 나타냅니다. 예를 들어 재귀적으로 모든 부분집합을 생성하는 알고리즘은 2^n개의 부분집합을 열거하므로 $O(2^n)$입니다. 입력 크기가 조금만 커져도 감당이 안 되는 복잡도로, $n=20$이면 $2^{20} \approx 1,048,576$ (백만) 정도지만 $n=30$이면 10^9을 넘어버립니다. **완전탐색 (brute-force)** 중 가지치기가 없는 경우에 많이 나타납니다.
- **$O(n!)$ – 팩토리얼 시간:** 가장 비효율적인 복잡도 중 하나로, 입력 크기가 n일 때 $n!$ (팩토리얼)만큼의 조합을 모두 시도하는 경우입니다. 예를 들어 n개의 도시를 여행하는 모든 경로를 탐색(TSP 완전탐색)하면 $n!$ 가지 경로를 모두 고려해야 하므로 $O(n!)$입니다. $n=10$만 되어도 $10! = 3,628,800$으로 수백만 번 연산이 필요하고, $n=15$이면 $15! \approx 1.3 \times 10^{12}$로 현재 컴퓨터로도 불가능에 가깝습니다.

위 복잡도들 간의 **성장 속도 비교**를 그래프로 그려보면 다음과 같습니다 [33] [34] :



[33] [34] 다양한 시간 복잡도의 증가 속도를 나타낸 그래프입니다. 가장 아래 평평하게 유지되는 선은 $O(1)$ (상수 시간)으로 입력 크기에 영향을 받지 않습니다. 그 위에 완만한 곡선은 $O(\log n)$ (로그), 직선은 $O(n)$ (선형)이며, 그보다 가파른 곡선이 $O(n \log n)$, 그리고 점점 가파르게 상승하는 곡선들이 $O(n^2)$, $O(2^n)$, $O(n!)$ 등을 나타냅니다. 입력 크기 n가 커질수록 지수나 팩토리얼 복잡도는 급격히 커지는 반면, 로그나 선형은 상대적으로 완만하게 증가하는 것을 볼 수 있습니다. 따라서 알고리즘을 설계할 때 가능하면 낮은 복잡도로 만드는 것이 중요합니다 (예: $O(n^2)$ 알고리즘을 $O(n \log n)$으로 개선하는 등). [35] [36]

위 그래프에서 볼 수 있듯이, **시간 복잡도 측면에서 좋은 알고리즘 순**은 (좋음) $O(1)$ > $O(\log n)$ > $O(n)$ > $O(n \log n)$ > $O(n^2)$ > $O(2^n)$ > $O(n!)$ (나쁨) 순입니다 [34] . 현실적으로 코딩 테스트에서는 보통 n이 수백만까지 갈 수 있으므로, **$O(n \log n)$ 이하거나 $O(n)$ 이하의 알고리즘**이 요구되는 경우가 많습니다. $O(n^2)$ 알고리즘은 $n \approx 10^5$일 때는 10억 연산이 필요해 시간 초과가 날 수 있으므로, n의 크기에 따라 어느 복잡도가 적절할지 판단하는 것이 중요합니다.

추가로, **공간 복잡도** 측면의 일반적인 등급도 언급하면: - $O(1)$ 공간: 상수 공간 (추가 배열이나 리스트 거의 안 쓰는 경우) - $O(n)$ 공간: 입력 크기에 비례한 추가 공간 (예: 입력만한 크기의 추가 배열 사용) - $O(n^2)$ 공간 등: 2차원 DP 테이블 등 사용 시.

공간은 제한이 있는 경우가 아니면 관대하게 쓰는 편이지만, 보통 메모리 제한도 코딩 테스트에 명시됩니다 (예: 128MB 등). int 1개가 4바이트이므로, 128MB면 약 32 million 개 int 정도 저장 가능하다는 감을 가지고 코딩하면 됩니다.

마지막으로, **빅-O 표기법의 의미**를 오해하면 안 되는데요. $O(n)$이라는 것이 반드시 정확히 kn번 연산이라는 뜻이 아니고, $O(n)$ 범주에 속하는 여러 함수(예: n, $0.5n+10$, $2n - 5$ 등)를 모두 뭉뚱그려 표현한 것이라는 점입니다. 또한 빅-O는 상수를 무시하므로, 실제로 $O(n)$ 알고리즘이라도 $1000n$ 연산을 하는 알고리즘이 $2n$ 연산을 하는 알고리즘보다 항상 느리다는 뜻은 아닙니다. 하지만 입력 규모가 충분히 크다면 차수가 중요한 결정요소가 되므로, 우리가 빅-O로 복잡도를 분석하는 것은 **큰 그림에서 알고리즘 효율을 논하기 위함**임을 기억합시다 [27] [37] .

정리하면, **점근적 표기법(Big-O 등)**을 사용하면 알고리즘의 **성능 특성**을 간단히 표현할 수 있고, **여러 알고리즘을 복잡도 관점에서 비교**하기가 수월해집니다. 앞으로 문제 풀이를 할 때, "이 알고리즘은 입력 n에 대해 $O(n\log n)$이다"와 같이 복잡도를 분석/표기하는 습관을 들이도록 노력합시다.

기본 문법 및 구현 요소 (C++14, Java 17, Python 3)

이번 섹션에서는 알고리즘 문제를 풀 때 가장 기본적으로 알아야 할 **프로그래밍 문법과 테크닉**을 세 가지 언어별로 정리합니다. 특히 **입출력 방법, 기본 자료형과 연산, 배열/리스트 처리, 반복문 사용, 그리고 주의해야 할 언어별 특성** 등을 다룰 것입니다. 세 언어를 병행해서 다루지만, 모든 내용을 다 암기하려 하지 말고 각자 주력으로 사용할 언어를 기준으로 보고 필요한 부분에 교차 참조하는 형태로 읽으면 좋겠습니다.

표준 입력과 출력 (Standard I/O)

효율적인 입출력은 문제 해결에서 매우 중요합니다. 언어별로 콘솔 입출력을 다루는 방법과, 대량의 데이터를 처리할 때의 팁을 알아봅시다.

- **C++의 입력/출력:**
  C++에서는 `#include <iostream>` 헤더를 통해 `std::cin`과 `std::cout`을 사용하여 표준 입력과 출력을 처리합니다. 예를 들어 정수 두 개를 입력받아 합을 출력하는 코드는 다음과 같습니다:

  ```
  int a, b;
  std::cin >> a >> b;
  std::cout << a + b;
  ```

  C 언어 스타일의 `scanf/printf`도 `#include <cstdio>`로 사용 가능하나, C++ 스타일을 추천합니다. 다만 `cin/cout`은 기본 설정상 C의 입출력과 동기화되어 있어 속도가 상대적으로 느릴 수 있습니다. **빠른 입출력**이 필요할 경우 다음 구문을 코드 초반에 추가하면 좋습니다:

  ```
  std::ios::sync_with_stdio(false);
  std::cin.tie(NULL);
  ```

  이는 C++의 iostream과 C의 stdio의 동기화를 끊고, `cout`의 묶음을 풀어 (`tie`) 출력 성능을 높여줍니다. 이렇게 하면 `scanf/printf`보다도 큰 차이 없이 `cin/cout`을 빠르게 사용할 수 있습니다. 출력 시 줄바꿈은 `std::cout << "\n";` 또는 `std::endl`을 쓰는데, `std::endl`은 출력 버퍼를 강제 flush하므로 자주 쓸 경우 성능에 영향이 있습니다. 따라서 개행만 필요하면 `"\n"`을 사용하는 것이 좋습니다.

파일 입력을 사용하는 경우는 드뭅니다만, 필요한 경우 `freopen`을 써서 표준입력을 파일로 바꿀 수 있습니다:

```
freopen("input.txt", "r", stdin);
```

이렇게 하면 이후 `std::cin`은 파일 `input.txt`로부터 읽어오게 됩니다. (제출 전에 잊지 말고 주석 처리나 제거해야 함)

- **Java의 입력/출력:**
  Java는 `java.util.Scanner`를 이용한 입력이 가장 간단합니다:

  ```
  Scanner sc = new Scanner(System.in);
  int a = sc.nextInt();
  int b = sc.nextInt();
  System.out.println(a + b);
  ```

그러나 `Scanner`는 편리한 대신 매우 느립니다. 많은 양의 데이터를 빠르게 입력받아야 한다면 `BufferedReader`를 사용하는 것이 좋습니다:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String[] parts = br.readLine().split(" ");
int a = Integer.parseInt(parts[0]);
int b = Integer.parseInt(parts[1]);
System.out.println(a + b);
```

`BufferedReader`는 한 줄 단위로 입력을 받으므로, 공백으로 나뉜 숫자를 읽을 때는 위처럼 한 줄을 읽고 `split`하거나, `StringTokenizer`를 사용할 수 있습니다. 출력의 경우 `System.out.println`이 개행 출력, `System.out.print`가 개행 없는 출력입니다. Java도 출력 시 `System.out.flush()`로 강제 flush 가능하지만, 일반적으로 문제 풀이에서는 자동 flush로도 충분합니다.

Java에서 파일 입력을 하려면 `new FileInputStream("input.txt")` 등을 Scanner나 BufferedReader에 넣어주면 됩니다. 하지만 코딩 테스트 환경에서는 표준 입력으로 주어지므로, 파일 입출력은 연습할 때나 필요하지 실제 제출 코드에 쓰지는 않을 것입니다.

- **Python의 입력/출력:**
  Python에서는 `input()` 함수를 통해 한 줄의 입력을 문자열로 받습니다. 여러 개를 한꺼번에 받으려면 `input().split()`으로 공백 단위 분할을 하고, 필요하면 `map(int, ...)`으로 형 변환을 합니다. 예:

```
a, b = map(int, input().split())
print(a + b)
```

Python은 기본적으로 입력이 느린 편이어서, 입력 데이터가 많을 경우 `sys.stdin.readline()`을 사용하는 것이 좋습니다:

```
import sys
data = sys.stdin.readline().strip().split()
# 필요하면 map(int, data) 등 처리
```

이 방법은 `input()`보다 빠르게 동작합니다[38]. 출력은 `print()`가 편리하며, `print(..., end="")`로 개행을 막거나, `print(..., flush=True)`로 즉시 출력시킬 수도 있습니다. 다만 일반적으로 코딩 테스트 환경에서는 프로그램이 종료되면 출력 버퍼가 자동 flush되므로 `flush=True`를 쓸 일은 거의 없습니다.

Python에서 파일을 읽고 싶다면:

```
f = open("input.txt", "r")
data = f.read().split()
```

처럼 가능합니다. 하지만 이 또한 연습용이고 제출 시에는 표준 입력으로 받도록 해야겠지요.

EOF(End of File) 처리: 간혹 **입력 개수가 명시되지 않은 문제**가 있습니다 (예: 입력이 파일 끝까지 주어질 때까지 처리하는 문제). 이 경우 각 언어별로 EOF를 감지하는 방법: - C++: `while (std::cin >> x)` 형태로 쓰면 더 이상

읽을 데이터가 없을 때 반복문이 종료됩니다. - Java: Scanner의 `hasNext()` 메서드를 사용하거나, BufferedReader의 `readLine()`이 `null`을 반환하는지로 EOF를 판단합니다.

```
while(sc.hasNext()){
  String s = sc.next();
  ...
}
```

- Python: `for line in sys.stdin:` 루프로 파일 끝까지 읽을 수 있습니다. 또는 `try: input()`을 쓰고 `except EOFError: break`로 처리합니다.

입출력 버퍼와 성능: 프로그램이 실행되면 입력과 출력은 **버퍼(buffer)**를 통해 효율적으로 처리됩니다. 출력의 경우 버퍼가 가득 차거나 개행 문자를 만나거나 flush 명령이 올 때 실제로 출력됩니다. 그래서 `print()`로 출력할 때 개행 없이 하면 버퍼에 쌓아두었다가 한꺼번에 나중에 내보내기 때문에 **빈번한 출력 호출을 줄이고 버퍼링을 활용**하면 성능이 좋아집니다 [39]. C++의 `std::ios::sync_with_stdio(false)`는 C stdio와의 동기화를 끊어 더 빠르게 버퍼링하도록 해주고 [40], Java의 BufferedWriter (또는 System.out 자체도 어느 정도 버퍼링 됨), Python의 출력도 내부적으로 버퍼링되어 있습니다. 요약하면, **많은 양의 데이터를 처리할 땐** 표준 입출력도 **버퍼를 잘 활용**해야 시간 초과를 피할 수 있습니다. 필요 시 C++에서는 `getline`으로 한 줄씩 읽고 파싱하거나, Python에서도 `sys.stdout.write`로 한번에 문자열을 출력하는 등의 방법도 씁니다.

변수와 자료형 (Variables and Data Types)

세 언어는 **변수 선언 방식과 자료형**에서 큰 차이를 보입니다. 간략히 정리해봅시다.

- **C++: 정적 타입(static type) 언어**로, 변수를 선언할 때 타입을 반드시 명시해야 합니다. 예를 들어 `int x = 5;`, `double y = 3.14;` 처럼 씁니다. 주요 기본형으로 `int` (4바이트 정수), `long long` (8바이트 정수), `double` (8바이트 실수), `char` (1바이트 문자), `bool` (논리값) 등이 있습니다. C++14에서는 `auto` 키워드를 이용해 초기값을 보고 타입을 추론하게 할 수도 있습니다 (예: `auto z = 10;` 이면 z는 int로 추론).

C++은 메모리 관리 측면에서 **값 타입**과 **포인터/참조** 개념이 있습니다. 알고리즘 문제 풀이에서는 일반적으로 동적 메모리 할당(new/delete)을 직접 다룰 일은 드물고, `std::vector` 등의 컨테이너를 주로 쓰게 됩니다. 타입 변환 시에는 `(int)3.14` 또는 `static_cast<int>(3.14)`처럼 명시적으로 캐스팅합니다. **정수 오버플로우**에 주의해야 하는데, 예를 들어 `int` 범위를 넘어가는 값 (약 ±2.1e9) 계산시 이상한 값이 나오므로, 필요하면 `long long`을 사용해야 합니다.

- **Java:** C++처럼 **정적 타입 언어**이며, 기본 자료형(primitive type)과 참조형(reference type)이 있습니다. 기본형으로 `int` (4바이트), `long` (8바이트), `double` (8바이트 배정밀도 실수), `float` (4바이트 단정밀도 실수), `char` (2바이트 유니코드 문자), `boolean` 등이 있습니다. 변수 선언 시 타입을 써야 하고 (`int a = 5;` 등), 참조형 (예: `String`, 배열, 사용자 정의 클래스 등)은 객체에 대한 **레퍼런스**를 담습니다. `String`은 Java에서 객체(참조형)지만 워낙 자주 쓰이므로 기본형처럼 다뤄지기도 합니다.

Java는 C++과 달리 메모리 관리를 가비지 컬렉션이 자동으로 해주기 때문에, new로 생성한 객체를 지우는 delete 등이 필요 없습니다. 형 변환은 숫자 타입 간에는 자동/명시적 캐스팅 (`int i = (int)3.14;`), 객체 간에는 상속 관계에 따라 캐스팅 등이 있습니다. **오버플로우**는 Java에서도 역시 조심해야 하는데, int 범위를 넘으면 C++처럼 언더플

로우/오버플로우가 나지만 예외를 던지지는 않고 결과만 틀어지므로, 큰 수 계산에는 `long`을 써야 합니다. (Java에는 C++ `unsigned`가 없고 모두 부호있는 타입입니다.)

- **Python: 동적 타입(dynamic typing) 언어**입니다. 변수 선언 시에 타입을 지정하지 않으며, 할당되는 값에 따라 **런타임에 타입이 결정**됩니다. 예를 들어 `x = 5`라고 하면 x는 정수(int)로, `x = "Hello"`라고 하면 곧바로 문자열(str)로 타입이 바뀝니다. Python의 자료형은 기본적으로 **객체**이고, 정수도 무한 정밀도를 가지는 `int` (오버플로우가 없는 큰 정수), `float` (배정밀도 부동소수점), `str` (문자열), `bool`, `list`, `dict` 등 다양합니다.

Python은 개발자가 신경 쓸 메모리 관리가 거의 없지만, **가비지 컬렉션**으로 자동 메모리 해제를 합니다. 숫자 연산에서 **오버플로우가 없다는 점**은 큰 장점이지만, 그만큼 자동으로 큰 정수를 다루기 때문에 C++이나 Java보다 연산 속도가 느릴 수 있습니다. 형 변환은 `int("123")`, `float(5)` 이런 식으로 함수 호출 형태로 이뤄집니다.

세 언어 비교 요약: C++/Java는 컴파일 시 타입이 결정되는 정적 타입 언어이므로 **타입에 맞지 않는 연산은 컴파일 오류**가 납니다. Python은 실행 중에 타입이 결정되어 유연하지만, 반대로 잘못된 타입로 연산하면 **런타임 오류**가 납니다. 예를 들어 C++에서 `"text" + 5`는 컴파일 오류지만, Python에서는 `"text" + 5`를 실행해보기 전까지는 오류를 모릅니다 (실행하면 `TypeError` 예외 발생). 코딩 테스트에서는 정적/동적 타입의 유불리가 크게 작용하진 않지만, Python으로 풀 때는 타입 실수를 조심해야 하고, C++/Java로 풀 땐 미리 타입을 맞춰주는 노력이 필요합니다.

주의해야 할 연산 특성: - 정수 나눗셈의 **몫 연산**이 언어별로 다르게 동작합니다. **C++/Java**에서는 정수끼리 나누면 소수점 이하를 버린 **정수 몫**을 결과로 줍니다. 예를 들어 `5/2`는 2가 됩니다. 음수의 경우 C++14부터는 0쪽으로 버림(truncate)합니다. 예컨대 `-5/2`는 -2 (파이썬과 결과 다름에 주의). **Python**에서 `/` 연산자는 항상 **실수 나누기**를 수행하여 결과를 float로 줍니다. 그래서 `5/2`는 2.5가 됩니다. Python에서 정수 몫을 구하려면 `//` 연산자를 사용해야 하며, `5//2`는 2, `-5//2`는 -3 (내림(floor) 연산)입니다. 이런 차이 때문에 알고리즘 구현 시 **언어별 몫 연산 결과 차이**를 염두에 둬야 합니다. (예: C++로 구현한 코드와 Python 코드가 음수 나눗셈에서 다른 결과를 낼 수 있음) - **모듈로 연산 (%)**은 C++/Java/Python 모두 정수 나머지를 구합니다. 다만 음수에 대한 정의가 언어마다 조금 다릅니다. Python의 `%`는 수학적 모듈러(나머지가 항상 0 이상)로 정의되어 `-5 % 2 = 1`이지만, C++/Java에서는 -5 % 2 = -1이 됩니다. 코딩 테스트에서는 주로 양수 범위에서 쓰므로 크게 문제되진 않지만, 가끔 음수를 다루게 되면 이 차이를 인지해야 합니다. - **실수 오차:** Java와 C++의 `double`은 15~16자리 정도의 유효숫자를 가지므로, 소수 계산시 부동소수점 오차가 있을 수 있습니다. Python의 `float`도 같은 64-bit 부동소수점이라 마찬가지입니다. 따라서 실수 비교를 할 땐 오차 범위를 감안한 비교가 필요합니다. 또는 Python의 `decimal` 모듈이나 Fraction, C++의 `long double` 등을 상황에 따라 고려합니다.

배열과 리스트 (Arrays and Lists)

자료구조의 가장 기본인 **배열(array)** 혹은 **리스트(list)** 사용법을 알아보겠습니다. 각 언어별로 배열을 만드는 문법과 관련 메서드를 정리합니다.

- **C++에서 배열:**
  C++에서는 고정 크기 배열과 동적 배열(vector)를 모두 사용할 수 있습니다. **정적 배열**(스택에 할당되는 고정 배열)은 다음처럼 선언합니다:

```
int arr[100];    // 크기 100짜리 int 배열 (값은 초기화되지 않은 상태)
int arr2[5] = {1,2,3,4,5}; // 크기 5, 초기값 지정
```

  그러나 정적 배열은 크기가 컴파일시에 고정되어야 하고, 크기가 큰 경우 스택 오버플로우 위험이 있습니다. 그래서 보통 알고리즘 문제에서는 `std::vector`를 더 많이 활용합니다:

```cpp
#include <vector>
vector<int> vec;          // 빈 동적 배열
vector<int> vec2(10);     // 크기 10, 기본값 0으로 초기화
vector<int> vec3(5, 42);  // 크기 5, 모든 원소를 42로 초기화
```

`vector`는 크기가 동적으로 늘어나므로 원소를 추가할 때 `vec.push_back(value);`를 사용합니다. 또한 `vec.size()`로 현재 크기를 얻고, `vec.empty()`로 비어있는지 확인하는 등 다양한 메서드를 제공합니다.

배열/벡터의 길이를 구하는 방법: 정적 배열 `arr`의 길이는 컴파일 시 알 수 있으면 `sizeof(arr)/sizeof(int)`로 계산 가능하나, 보통 이렇게 하지 않고 그냥 상수를 알고 있을 때만 씁니다. `std::vector`의 경우 `vec.size()` 메서드를 사용하면 O(1) 시간에 길이를 얻을 수 있습니다.

기본 함수들: C++ `<algorithm>` 헤더에는 배열/벡터를 다루는 여러 유용한 함수들이 있습니다. - `sort(arr, arr+n)` 또는 `sort(vec.begin(), vec.end())`로 정렬[35]. - `reverse(vec.begin(), vec.end())`로 뒤집기. - `min_element(vec.begin(), vec.end())`는 최소값 위치 이터레이터를, `max_element`는 최대값 위치를 줍니다 (역참조하면 실제 값). - `accumulate(vec.begin(), vec.end(), 0)`은 합계를 계산 (세 번째 인자는 초기값이자 합산 타입 결정자). 또는 그냥 루프 돌며 합계 가능. - `vec.push_back(x)`는 끝에 원소 추가, `vec.pop_back()`은 끝 원소 제거. - `vec.resize(n)`으로 크기 조절, `vec.clear()`로 모두 제거. - C++11부터 `vec.emplace_back(args...)`도 있는데 push_back과 유사하나 객체를 제자리 생성(성능 미세 최적화, 거의 동일).

또한 2차원 배열은 `vector<vector<int>> matrix(N, vector<int>(M));`처럼 벡터의 벡터로 만들 수 있습니다. (후술)

- **Java에서 배열과 리스트:**
  Java의 배열은 객체로 취급됩니다. 배열 선언 및 생성은 다음과 같습니다:

```java
int[] arr = new int[100];   // 크기 100짜리 int 배열 (기본값 0으로 초기화)
int[] arr2 = {1,2,3,4,5};   // 크기 5, 리터럴로 초기화
```

  Java 배열은 `.length` 속성으로 길이를 얻습니다 (예: `arr.length`). 한 번 생성한 배열은 크기를 변경할 수 없습니다.

동적으로 크기를 다루려면 `ArrayList`를 사용하는 것이 일반적입니다:

```java
import java.util.*;
ArrayList<Integer> list = new ArrayList<>();
list.add(42);       // 원소 추가
int first = list.get(0);  // 인덱스로 접근
list.set(0, 7);     // 값 변경
list.remove(list.size()-1); // 마지막 원소 제거
```

`ArrayList`는 내부적으로 배열을 이용하지만, 자동으로 크기를 관리해줍니다. `list.size()` 메서드로 크기를 구하고, `list.isEmpty()`로 비어있는지 확인할 수 있습니다. **주의:** `ArrayList<int>`처럼 기본형을 직접 쓸 수 없고, `Integer`와 같은 wrapper class를 써야 합니다.

정렬과 유틸리티: - 배열의 정렬: `Arrays.sort(arr);`를 사용 (퀵/병합정렬 기반, $O(n \log n)$). - 리스트의 정렬: `Collections.sort(list);` (또는 `list.sort()` Java8+). - 최소/최대: 배열은 먼저 정렬하거나, 수동으로 탐색; 리스트는 `Collections.min(list)`, `Collections.max(list)` 제공. - 합계: `Arrays.stream(arr).sum();` (자바8 스트림) 또는 직접 루프 합산. - 역순 뒤집기: `Collections.reverse(list);`.

Java에서 2차원 배열은 `int[][] matrix = new int[N][M];` 처럼 선언합니다. 2차원 `ArrayList`도 `ArrayList<ArrayList<Integer>> matrix` 구조로 사용 가능합니다.

- **Python에서 리스트:**
  Python의 **list**는 가변 길이 배열과 같은 기능을 합니다. 선언은 리터럴 `[]`로 빈 리스트를 만들거나, 값을 넣어서 `[1,2,3]`으로 만들 수 있습니다. 예:

  ```
  arr = [0]*100      # 길이 100, 모든 값 0으로 초기화된 리스트
  arr2 = [1, 2, 3, 4, 5]   # 초기화 리스트
  ```

  단 위의 `[0]*100` 방식은 1차원에서만 안전합니다. 2차원 리스트를 초기화할 때 `[[0]*M]*N`은 **잘못된 방법**인데, 같은 리스트 객체가 N번 복사되므로 나중에 한 행을 수정하면 다른 행들도 같이 바뀝니다. 2차원은 comprehension으로 `[[0]*M for _ in range(N)]`를 사용해야 각 행이 별개의 리스트가 됩니다.

Python 리스트의 길이는 `len(arr)`로 구합니다. 주요 연산: - `arr.append(x)`로 마지막에 원소 추가. - `arr.pop()`으로 마지막 원소 반환 후 제거 (인덱스 지정 가능: `arr.pop(0)` 첫 원소 제거 등). - `arr.sort()`로 제자리 정렬 (오름차순), `sorted(arr)`는 새로운 정렬 리스트 반환. - `arr.reverse()`로 제자리 역순, 또는 `arr[::-1]` 슬라이싱으로 역순 복사본 얻기. - `min(arr)`, `max(arr)`, `sum(arr)` 내장 함수로 최소/최대/합계 구하기. - 존재 확인 `x in arr` (평균 $O(n)$). - 슬라이싱 `arr[a:b]`로 부분리스트 (얕은 복사).

Python 리스트는 동적 배열이므로, 랜덤 인덱스 접근과 끝 추가/삭제가 평균 $O(1)$로 빠릅니다. 다만 리스트의 `insert(0, x)` 같이 맨 앞에 삽입하거나 삭제하는 연산은 $O(n)$이 걸림에 유의해야 합니다 (deque 사용 고려).

튜플(tuple): 리스트와 달리 **불변(immutable)** 시퀀스 타입으로, 한 번 정하면 값을 변경할 수 없습니다. 튜플은 보통 여러 값을 한번에 리턴받거나 딕셔너리 키로 사용되는 등 특별한 경우 외엔 알고리즘 문제에서는 잘 안 씁니다.

요약: 배열/리스트는 가장 기본 자료구조로 세 언어 모두 자유롭게 다룰 수 있어야 합니다. C++에서는 가능한 `vector`를 쓰길 권장하며, Java에서는 배열과 ArrayList를 상황에 맞게 활용하고, Python에서는 기본 list를 사용하면 됩니다.

특히 **언어별 인덱싱 범위**를 조심해야 하는데, 세 언어 모두 **0-based index** (0부터 시작)입니다. 마지막 인덱스는 `length-1`입니다. 인덱스 범위를 벗어나면: - C++: 정적 배열은 위험 (쓰레기값, 오류), vector는 `at` 메서드 쓰면 범위 체크 예외, 그냥 `vec[i]`는 범위 체크 안 함 (조심). - Java: `ArrayIndexOutOfBoundsException` 예외 발생. - Python: `IndexError` 예외 발생.

또한 **얕은 복사(shallow copy)와 깊은 복사(deep copy)** 개념도 자료구조 다룰 때 필요합니다. C++은 대입 연산자가 알아서 복사해주지만 포인터 있을 땐 주의, Java는 객체 참조를 대입하면 같은 객체를 가리키고, Python도 리스트 등 가변 객체는 그냥 대입하면 참조 복사입니다. 그래서 Python에서 리스트 복사할 때 `new = old[:]` 또는 `list(old)` 등을 사용해야 새로운 객체가 생깁니다.

반복문과 제어문 (Loops and Control Statements)

반복문(loops)은 세 언어 모두 지원합니다. 기본적으로 `for` loop와 `while` loop, 그리고 필요시 `break`, `continue` 같은 제어를 합니다. 각 언어의 문법 차이를 살펴봅니다:

- **C++의 반복문:**
  C++의 전통적인 for문 구조: `for (초기식; 조건식; 증감식) { ... }`. 예:

  ```
  for (int i = 0; i < 10; ++i) {
    // i는 0부터 9까지
  }
  ```

  증감 연산자는 `i++` (후위 증가)나 `++i` (전위 증가) 모두 쓸 수 있지만, 차이는 미미하니 편한 것으로 쓰면 됩니다. **범위 기반 for문** (C++11부터)은 컨테이너에 직접 쓸 수 있어서 편합니다:

  ```
  vector<int> arr = {1,2,3};
  for (int x : arr) {
    cout << x;
  }
  ```

  위 코드는 `arr`의 모든 원소를 차례로 꺼내 x에 대입하며 루프를 돕니다. 참조로 받을 수도 있는데 (`for(int &x: arr)`) 이 경우 x를 수정하면 arr 원소도 바뀝니다.

while문: `while (조건) { ... }`이고, 조건이 참인 동안 계속 실행합니다. 또한 최소 한 번은 실행해야 할 경우 `do { ... } while(조건);` 구조도 있습니다.

루프 제어: `break;`를 만나면 해당 루프 즉시 종료, `continue;`는 현재 반복을 건너뛰고 다음 반복으로 진행.

- **Java의 반복문:**
  C/C++과 유사하게 `for (초기식; 조건식; 증감식) { ... }` 형태를 가집니다:

  ```
  for (int i = 0; i < 10; i++) {
    // 0~9 loop
  }
  ```

  증감 연산자, break, continue 모두 C++과 동일하게 동작합니다.

Java도 **향상된 for문 (for-each)**을 제공합니다:

```
int[] arr = {1,2,3};
for (int x : arr) {
  System.out.println(x);
}
```

`ArrayList` 등 Iterable 객체에도 사용 가능합니다. 이 for-each문은 내부적으로 iterator를 쓰므로 인덱스가 필요 없는 경우 유용합니다.

while문과 do-while문도 문법이 똑같습니다:

```
while (condition) { ... }
do { ... } while (condition);
```

- **Python의 반복문:**
  Python에는 C 스타일의 전통적인 for문이 없고, **for-each 형태**만 존재합니다. `for 변수 in (이터러블):` 구조로 사용합니다:

  ```
  for i in range(0, 10):
      # i = 0,1,...,9
  ```

  `range(n)`은 0부터 n-1까지, `range(a, b)`는 a부터 b-1까지의 정수 시퀀스를 생성합니다. 따라서 위는 전통 for와 같은 효과입니다. Python에서는 증감식이 없고, range나 리스트 등을 통해 반복합니다. 리스트 원소를 직접 반복:

  ```
  arr = [1,2,3]
  for x in arr:
      print(x)
  ```

  이런 식으로 사용합니다. 인덱스가 필요하면 `for i, val in enumerate(arr):`를 쓰면 인덱스와 값을 함께 얻을 수 있습니다.

while문: `while 조건:`으로 사용하며, 문법은 들여쓰기로 블록을 구분합니다:

```
i = 0
while i < 10:
    print(i)
    i += 1
```

do-while은 Python에 없지만 while True + break로 시뮬레이트 가능합니다.

제어문: `break`와 `continue` 키워드가 있으며, 기능은 다른 언어와 동일합니다. 다만 Python에는 특별히 루프가 한 번도 실행되지 않았을 때나 정상 종료했을 때 실행되는 `else`절도 있지만, 복잡한 로직 아니면 잘 쓰이지는 않습니다.

주의사항: - Python에서는 **들여쓰기(indent)**가 문법의 일부이므로, 탭/스페이스 혼용하지 말고 일정하게 맞춰야 합니다. 보통 스페이스 4칸이 표준입니다. - C++/Java에서는 블록 `{ ... }` 내에서 새로운 변수를 선언하면 블록을 벗어나면 소멸/범위를 벗어납니다. Python은 함수 내라면 블록과 상관없이 같은 지역 범위이므로, for문 안에서 선언한 변수도 루프 이후에 그대로 남아있습니다. (예: `for i in range(5): ...` 후에도 i 값이 4로 남아있음)

- 반복문은 잘못 작성하면 **무한 루프**에 빠질 수 있습니다. C++/Java에서는 CPU 100%로 먹으며 멈추지 않고, Python에서는 일정 시간 지나면 실행이 끝나지 않아 채점 서버에서 강제 종료될 것입니다. 항상 루프 조건이 언젠가 거짓이 되는지, break로 빠져나오는지 등을 점검해야 합니다.

2차원 배열과 행렬 처리

그래프나 DP문제를 풀다 보면 **2차원 배열(행렬)**을 많이 사용하게 됩니다. 2차원 배열은 앞서 언급한 대로 선언할 수 있으며, 순회 시 **이중 루프**를 사용합니다. 예를 들어 `N x M` 크기의 2D 배열을 모두 방문하려면:

- C++:

```
for(int i = 0; i < N; ++i){
  for(int j = 0; j < M; ++j){
    // access arr[i][j]
  }
}
```

- Java:

```
for(int i = 0; i < N; i++){
  for(int j = 0; j < M; j++){
    // access arr[i][j]
  }
}
```

- Python:

```
for i in range(N):
    for j in range(M):
        # access matrix[i][j]
```

많은 조건 분기와 델타 탐색: 2차원 격자에서 인접한 셀들을 탐색할 때는 상하좌우 방향으로 이동하곤 합니다. 이때 **델타 배열** 기법을 사용하면 편리합니다. 델타 배열이란 이동할 방향을 미리 배열로 정의해두고, 루프를 돌며 적용하는 방식입니다. 예를 들어 **상, 하, 좌, 우** 4방향의 델타를 정의해보겠습니다:

```
int dx[4] = {-1, 1, 0, 0};  // x(행) 방향 - 위, 아래
int dy[4] = {0, 0, -1, 1};  // y(열) 방향 - 왼쪽, 오른쪽

for(int x = 0; x < N; ++x){
  for(int y = 0; y < M; ++y){
    for(int k = 0; k < 4; ++k){
      int nx = x + dx[k];
      int ny = y + dy[k];
      if(nx >= 0 && nx < N && ny >= 0 && ny < M){
        // (nx, ny)는 (x,y)의 상하좌우 중 하나
      }
    }
  }
}
```

위 예시는 C++이지만, Java나 Python에서도 동일한 아이디어로 구현합니다. 파이썬 예:

```
dx = [-1, 1, 0, 0]
dy = [0, 0, -1, 1]

for x in range(N):
    for y in range(M):
        for k in range(4):
            nx = x + dx[k]
            ny = y + dy[k]
            if 0 <= nx < N and 0 <= ny < M:
                # process neighbor
```

이처럼 델타 배열을 쓰면 코드가 깔끔해지고, 방향을 추가/삭제하기도 편해집니다. (예: 대각선도 포함하려면 dx, dy에 8방향을 넣으면 됨)

경계 조건 체크: 2차원 배열을 다룰 때 항상 **인덱스 범위**를 벗어나지 않도록 조건을 넣어야 합니다 (위 코드의 `if` 처럼). 또한 배열의 가장자리를 다룰 때 실수가 없도록 주의해야 합니다. 예컨대 `(0, y)` 셀의 위쪽 이웃은 없으므로, 조건을 확인하지 않으면 잘못된 접근이 일어납니다.

메모리: 2차원 배열은 요소 개수가 `N*M`개가 됩니다. N이나 M 중 하나만 커도 메모리 사용이 커지므로, 선언 전에 문제 제한을 고려해야 합니다. 예를 들어 N=M=10^5인 2D 배열은 10^10개의 요소로 메모리상 불가능합니다. 이런 경우 보통 입력 형태가 달라지거나 하는 식으로 문제가 주어지므로, 2차원 배열 크기 선정은 문제 요구에 맞게 해야 합니다.

문자열과 문자 처리

기본 문법 파트에 명시적으로 적혀있지는 않지만, **문자열(String)** 처리는 문제 풀면서 굉장히 자주 나오는 주제입니다. 간략히 언어별 요점만 잡고 넘어가겠습니다:

- C++: `#include <string>`을 통해 `std::string` 클래스를 사용합니다. `string s = "hello";` 처럼 선언하고, `s.size()` 또는 `s.length()`로 길이, `s[i]`로 개별 문자 접근, `s.substr(pos, len)`으로 부분문자열 추출, `s.find("llo")`로 부분 문자열 찾기 등이 가능합니다. 덧붙이기는 `s += " world";` 혹은 `s.append(" world");`로 합니다. C-style 문자열 (`char*` 또는 `char[]`)도 쓸 수 있으나, 웬만하면 `std::string` 쓰는 게 안전합니다.
- Java: `String str = "hello";`처럼 리터럴로 생성하며, `str.length()`로 길이, `str.charAt(i)`로 문자 접근, `str.substring(a,b)`로 부분 문자열 (a부터 b-1까지), `str.indexOf("llo")`로 부분 문자열 위치 찾기 등이 됩니다. Java String은 **immutable**이라서 변경이 빈번하면 `StringBuilder`를 쓰는 게 좋습니다 (`StringBuilder sb = new StringBuilder(); sb.append(...);`).
- Python: 문자열 리터럴은 `"hello"`나 `'hello'` 둘 다 가능합니다. `len(s)`으로 길이, `s[i]`로 문자 (문자열에서 문자도 길이 1의 문자열로 취급), `s[a:b]`로 부분 문자열(slice) 등. Python 문자열도 immutable이라서, 덧붙이는 연산을 루프에서 누적하면 비효율적일 수 있으니 성능 필요 시 `list`로 쌓았다가 `''.join(list)`를 쓰기도 합니다.

이스케이프 시퀀스 (Escape Sequences): 문자열을 다룰 때 **특수 문자**를 나타내기 위한 표기법이 있습니다. - `\n` : 새 줄 (newline, line feed) – 출력상 줄바꿈 효과. 예: `cout << "Hello\nWorld";`는 두 줄에 걸쳐 출력. - `\t` : 탭 (tab) – 수평 탭 간격. - `\\` : 백슬래시 자체 – 백슬래시를 문자열에 넣고 싶을 때는 두 번 써야 하나 출력은 하나만 됩니다. - `\"` : 큰따옴표 문자 자체 – 문자열 리터럴을 큰따옴표로 둘러싸기 때문에, 내용에 큰따옴표를 넣으려면 \"로 이스케이프. - `\'` : 작은따옴표 문자 자체 – 마찬가지 논리.

예를 들어 `printf("She said \"Hello\"\n");`는 `She said "Hello"` 라고 출력되고 줄이 바뀝니다. Python에서도 `print("She said \"Hello\"")` 동일하게 동작합니다. Python에서는 문자열 리터럴 앞에 `r`을 붙이면 raw string이 되어 이스케이프를 처리하지 않지만, 문제에서 자주 쓰이진 않습니다.

문자 코드: 문자는 내부적으로 정수 (ASCII 코드 등)로 표현됩니다. C++에서 `char c = 'A'; int code = c;` 하면 65가 들어갑니다. Java도 `(int)'A'`는 65. Python은 `ord('A')`가 65, `chr(65)`가 'A'입니다. 이러한 ASCII/유니코드 코드값은 알파벳 대소문자 연산 등에 가끔 씁니다. (예: 'A'+1 = 'B', '0'+5 = '5' 문자 등)

형 변환과 연산상의 주의점 (Type Casting & Arithmetic Caveats)

앞서 변수/자료형 섹션에서 일부 다뤘지만, 초보자 분들이 흔히 놓치는 **언어별 연산 차이**나 **주의점**을 다시 요약합니다:

- **정수 나눗셈 몫**: 반복되지만 중요해서 다시 명시합니다. C++/Java는 **소수점 버림**, Python은 `/`는 **실수 반환**, `//`는 **내림**. 특히 Python의 `//`는 음수에서 C++ 결과와 1 차이날 수 있음.
- **나눗셈 후 연산**: `1/2*2`를 C++/Java에서는 02 = 0으로 계산하지만, Python에서는 0.52 = 1.0으로 다르게 나옵니다. 즉, 정수/정수 연산은 언어 따라 결과 다르니, 알고리즘상 나눗셈이 필요하면 의도한대로 구현해야 합니다 (예: 부동소수점 쓸지, 올림/내림 직접할지).
- **오버플로우**:
- C++: 정수 오버플로우 시 modulo 2^n로 넘어가서 잘못된 값이 됨 (UB는 아님, unsigned는 정의된 동작).
- Java: 정수 오버플로우 시 언더플로우되어 잘못된 값이 됨 (예외 없음).
- Python: 큰 정수 알아서 bigint로 처리 (사실상 오버플로우 없음, 메모리가 허용하는 한).
- 따라서, **아주 큰 수** (예: 피보나치 100번째를 int로 계산 등)은 C++/Java에선 overflow 나니, 미리 범위를 예측해 타입을 크게 쓰거나, 모듈러 연산 등으로 대응해야 합니다.
- **부동소수 비교**: `0.1 + 0.2 == 0.3`을 대부분 언어에서 false로 만듭니다 (표현 오차). 즉 실수 계산은 오차가 있으니, 문제에서 실수 오차 허용치를 주는 경우 `abs(a-b) < 1e-9` 이런 식으로 비교합니다. Python의 `math.isclose`도 유용.
- **논리 연산과 short-circuit**: C++/Java의 `&&`, `||`는 short-circuit (왼쪽 결과로 판단되면 오른쪽 실행 안 함). Python의 `and`, `or`도 마찬가지. 비트 논리연산 `&`, `|`와 혼동하지 않도록.
- **증감 연산자**: Python엔 `i++`가 없습니다 (`i += 1`로 해야). C++/Java 전위/후위 차이는 주변 다른 연산과 함께 쓰일 때 주의 (ex: `arr[i++] = ...` vs `arr[++i] = ...` 결과 다름).
- **삼항 연산자**: C++/Java에는 `cond ? expr1 : expr2`가 있지만 Python에는 없고 대신 `expr1 if cond else expr2` 형태로 사용합니다.

마지막으로, **언어별 표준 라이브러리 활용**에 익숙해지는 것도 중요합니다. 예를 들어: - C++: `<algorithm>`, `<map>`, `<set>`, `<queue>` 등 STL 사용. - Java: `java.util` 컬렉션들 (List, Map, Set 등) 사용. - Python: 내장 함수 (sorted, min 등)와 `itertools`, `math`, `collections` (deque, Counter) 같은 모듈 활용.

문제를 풀 때, **가능한 한 표준 라이브러리를 활용하여 간결하게 구현**하고, 직접 구현이 필요한 부분은 정확히 구현하도록 합니다. 다만 너무 라이브러리에 의존하면 핵심 로직 이해가 부족해질 수 있으니, 예를 들어 정렬 알고리즘을 처음 배울 땐 한 번 직접 구현(버블/병합 등)해보고, 실전에서는 내장 `sort`를 쓰는 식으로 병행하면 좋겠습니다.

표준 I/O 심화 및 팁

마지막으로, **표준 입출력과 관련된 조금 더 깊은 내용**과 팁을 짚고 마치겠습니다.

- **입출력 버퍼 작동 원리:**
  프로그램이 실행될 때 표준 입력(stdin)과 표준 출력(stdout)은 운영체제에 의해 **버퍼링**됩니다 [39]. 예를 들어 `printf`나 `cout`으로 출력하면 일단 버퍼에 데이터가 쌓이고, 버퍼가 가득 차거나 줄바꿈이 나오거나 flush 명령이 있을 때 실제 출력 장치(콘솔)에 나타납니다. 입력도 마찬가지로, 사용자가 키보드에 입력한 내용

이 Enter를 치기 전까지는 프로그램에 전달되지 않고 터미널 입력 버퍼에 쌓입니다. 이러한 버퍼링은 시스템 호출(IO)을 줄여 성능을 높이기 위한 메커니즘입니다.

이 때문에 가끔 `cout << "Prompt"; cin >> x;` 같은 코드를 짤 때 `Prompt`가 화면에 안 나오는 현상이 생길 수 있습니다. 개행이 없어서 버퍼에 머물기 때문인데, 이때는 `cout.flush()`를 호출하거나 `std::endl`을 써서 flush를 강제해야 합니다. 콘솔 프로그램에서는 흔하지 않지만, 대화형 문제에서는 flush 처리가 필요할 수 있습니다.

- **fast I/O (고속 입출력):**
  대량의 데이터 입출력이 있을 때 속도를 높이기 위해 **버퍼를 크게 쓰거나, 시스템 레벨 함수**를 직접 사용하기도 합니다. 예를 들어 C++에서는 `scanf/printf`보다 더 빠르게 하려고 `getchar_unlocked()` 등을 쓰거나, 아예 `read` 시스템콜을 써서 한번에 읽어와 파싱하는 고급 기법도 존재합니다. Python에서는 `sys.stdin.buffer` (바이너리 버퍼)로 읽어서 파싱하거나, PyPy 같은 JIT 인터프리터를 활용하기도 합니다. 하지만 이는 정말 입력량이 수백 MB 단위로 크지 않으면 대부분 필요 없습니다. 일반적인 코딩테스트에서는 `BufferedReader` / `scanf` 수준으로도 충분하며, 혹시 모자라면 언어를 바꾸는게 현실적일 때도 있습니다.

다만, **Python으로 매우 입출력이 많은 문제를 풀 땐** C++로 풀면 쉽게 통과될 수도 있습니다. 그런 경우 파이썬에서 `sys.stdin.readline`과 `sys.stdout.write`로 최적화해보고, 그래도 안 되면 PyPy3로 제출하거나, 결국 C++로 재도전할 수도 있습니다. 속도 면에서 C++이 유리한 경우가 분명히 있으니, 언어 선택도 전략적으로 해야 합니다.

- **EOF까지 입력 처리 팁:**
  앞서 언급한 EOF 처리에 하나 덧붙이면, C++에서 `while(cin >> x)`처럼 여러 개 값을 차례로 읽는 코드를 작성할 때, 만약 정수가 아닌 문자열 라인 단위로 EOF까지 처리해야 한다면 `string line; while(true){ if(!getline(cin, line)) break; ... }` 식으로 할 수 있습니다. Java는 `while(br.ready()){ ... }` 나 `while((line = br.readLine()) != null){ ... }` 패턴이 있고, Python은 간단히 `for line in sys.stdin:`이 편합니다.

- **출력 포맷 주의:**
  알고리즘 문제를 풀다 보면 출력에 공백/개행이 정확히 맞아야 정답 처리됩니다. **불필요한 출력**(예: 디버그용 프린트)을 남기면 오답 처리됩니다. 또한 C++ `endl`은 개행+flush라서 느릴 수 있고, Python `print`는 기본 개행 포함이라 `print(..., end="")`로 제어 가능합니다. 문제에서 요구하는 포맷 (예: "Case #1: ..." 이런 것)을 정확히 맞추는 연습이 필요합니다.

- **언어별 기본 입력 속도:**
  일반적으로 **C++ > Java > Python** 순으로 기본 입력처리가 빠릅니다. C++의 `scanf`/`cin.sync(false)`는 매우 빠른 편이고, Java의 `Scanner`는 느리지만 `BufferedReader`는 상당히 개선됩니다. Python은 느린 축에 속하지만, 한 줄로 몰아서 처리하거나 C 확장 모듈 (`numpy.fromfile` 등) 쓰는 등의 트릭이 있긴 합니다. 하지만 초보 단계에서는 이런 미세한 것보다, **알고리즘 개선**으로 시간 단축하는 데 집중하는 게 바람직합니다. I/O는 TLE 날 만큼만 최적화하면 됩니다.

마지막으로 강조하고 싶은 점은, **제가 강의에서 모든 걸 전부 다 다루지 않을 수 있다는 것**입니다. 위 내용도 매우 방대해 보이지만, 실전에서 겪게 될 상황의 일부일 뿐입니다. 만약 제 설명이 부족하거나 이해가 안 되는 부분이 있다면, **혼자서도 관련 자료나 레퍼런스를 찾아보고 공부하는 습관**을 들이세요. 필요하면 공식 문서나 신뢰할 만한 블로그, 유튜브 강의 등을 찾아보기를 권장합니다. 예를 들어 위에서 언급된 개념들을 정리한 블로그 글이나 위키백과 내용을 참고하면 더욱 분명하게 이해될 수도 있습니다. 스터디 중에도 자료들을 공유드리겠지만, **스스로 찾아보는 학습**이 가장 효과적임을 기억해주세요.

그래도 이미지나 도표가 도움이 될 만한 내용들은 최대한 삽입하려 노력했고, 핵심 개념마다 출처와 참고 링크를 달아 두었습니다. 필요할 때 해당 출처를 읽어보는 것도 좋겠습니다. 이 문서는 여러분이 두고두고 참고할 **보고서 스타일의 정리본**이 될 수 있도록 작성되었으니, 추후 PDF 등으로 만들어 옆에 놓고 보면서 공부하셔도 좋을 것 같습니다.

이것으로 오리엔테이션 및 기본 개념 정리를 마치겠습니다. 다음 시간부터는 실제 문제를 풀면서 오늘 언급된 개념들을 어떻게 활용하는지 다뤄보겠습니다. 궁극적으로 3개월 후에는 여러분이 **중급 알고리즘 문제도 막힘없이 해결하고, 여러 언어로 구현할 수 있는 수준**에 도달하기를 기대합니다. 함께 열심히 해보죠! [28] [25]

---

[1] 라즈이노 iOT :: 〖VS-Code〗 VS code처음 사용 설명서, C/C++ & Python 멀티 코딩 환경 셋팅하기! (코드러너, 디버깅기능 사용)
https://rasino.tistory.com/337

[2] [3] [4] [24] [25] [26] Complexity Analysis (1) - Computational Complexity (계산 복잡도)
https://jwoop.tistory.com/14

[5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [알고리즘] 백준 온라인 저지 사이트 소개
https://smartpro.tistory.com/2

[15] [16] [17] [23] 2025 개발자 취준생을 위한 백준 활용법 총정리 - 내일배움캠프 블로그
https://nbcamp.spartacodingclub.kr/blog/2025-%EA%B0%9C%EB%B0%9C%EC%9E%90-%EC%B7%A8%EC%A4%80%EC%83%9D%EC%9D%84-%EC%9C%84%ED%95%9C-%EB%B0%B1%EC%A4%80-%ED%99%9C%EC%9A%A9%EB%B2%95-%EC%B4%9D%EC%A0%95%EB%A6%AC-42692

[18] [19] [20] [21] [22] 알고리즘 입문자를 위한 백준/솔브닥(solved.ac) 사용백과
https://blog.koder.page/solvedac-guideline/

[27] [30] [37] Complexity Analysis (2) - Asymptotic Analysis (점근적 분석)
https://jwoop.tistory.com/15

[28] [29] 점근 표기법 - 위키백과, 우리 모두의 백과사전
https://ko.wikipedia.org/wiki/%EC%A0%90%EA%B7%BC_%ED%91%9C%EA%B8%B0%EB%B2%95

[31] [32] [33] [34] [35] [36] [38] [39] [40] Big O Cheat Sheet – Time Complexity Chart
https://www.freecodecamp.org/news/big-o-cheat-sheet-time-complexity-chart/