ChatGPT

알고리즘 스터디 10주차: 그리디 & 구현(시뮬레이션)

1. 서론

지난 9주차 **그래프 심화** 파트에서는 그래프 관련 알고리즘의 고급 주제를 다루었습니다. 예를 들어 **최단 경로 알고리즘**(다익스트라, 플로이드-워셜 등)과 **최소 신장 트리** 알고리즘(크루스칼, 프림)이 대표적이었습니다. 이러한 알고리즘들을 통해 가중치가 있는 그래프에서 효율적으로 최적 해를 찾는 방법을 학습했습니다. 특히 다익스트라 알고리즘과 프림 알고리즘은 **탐욕적 선택**을 활용하는 알고리즘이었고, 네트워크 흐름이나 위상 정렬 등도 그래프의 다양한 활용을 보여주었습니다. 지난 시간 내용을 요약하면:

- **최단 경로 알고리즘**: 한 정점에서 다른 정점들로의 최단 거리를 구하는 문제로, 다익스트라 알고리즘(O(E log V)) 등을 통해 효율적으로 풀이했습니다. 경우에 따라 벨만-포드 알고리즘이나 플로이드-워셜 알고리즘으로도 해결 가능한 상황을 살펴보았습니다.
- **최소 신장 트리(MST)**: 그래프의 모든 정점을 연결하면서 가중치 합이 최소인 트리를 찾는 문제로, Kruskal 및 Prim 알고리즘을 다루었습니다. 이를 통해 **그리디 알고리즘**의 개념(가장 작은 간선부터 선택 등)이 실제 문제에 적용되는 모습을 확인했습니다.
- **기타 그래프 심화 내용**: 위상 정렬(순서가 정해진 작업 나열 문제)이나 이분 그래프 판별, 네트워크 유량 등 고난도 주제에도 간략히 언급하며 그래프 이론의 폭을 넓혔습니다.

이처럼 그래프 심화 단원에서는 **여러 경로들 중 최적 경로를 찾는 알고리즘**과 **탐욕적 선택의 유용성**을 경험했습니다. 이번 **10주차**에서는 새롭게 **탐욕 알고리즘(Greedy)**의 일반 원리와, 별도의 알고리즘보다는 **구현력과 시뮬레이션** 능력이 중요한 **구현 문제**를 다룰 것입니다. 이를 통해 **문제 해결 전략의 다양성**을 배우고, 앞으로 이어질 문자열 알고리즘 등으로 나아갈 준비를 합니다.

2. 본론1 – 그리디 알고리즘

개념 정의

그리디 알고리즘(Greedy Algorithm) 또는 **탐욕 알고리즘**은 "매 단계에서 당장 가장 좋아 보이는(최적이라고 생각되는) 선택을 함으로써 최종적인 해답에 도달하는" 문제 해결 기법입니다. 이름 그대로 `탐욕스러운` 방식으로 현재 상황에서 최선의 결정만을 쫓아가는 전략입니다. 예를 들어, 동전 거스름돈을 줄 때 **가장 큰 화폐 단위부터** 거슬러 주는 것은 탐욕적 선택의 한 사례입니다. 그리디 알고리즘의 목표는 **국지적으로 최선인 선택**들을 모아서 **전체적으로도 최선인 해결책**을 얻는 것입니다.

그러나 이러한 접근이 **항상** 전역 최적해를 보장하지는 않습니다. 그리디는 부분적인 최적만 고려하기 때문에, 때로는 **눈앞의 이익을 쫓다가 전체 최적을 놓치는 상황**이 발생할 수 있습니다. 유명한 비유로, 당장 눈앞의 **1개의 마시멜로**를 선택하는 아이는 탐욕적인 결정이지만, **1분 기다리면 2개를 받는** 기회를 놓치는 꼴이 될 수 있습니다. 이처럼 현재 시점의 최선이 **장기적인 최선과 다를 수 있다**는 점을 항상 유의해야 합니다.

그럼에도 불구하고 **그리디 알고리즘**은 널리 쓰입니다. 가장 큰 이유는 **단순성과 속도**입니다. 동적 계획법(DP)은 모든 경우를 탐색하여 최적해를 찾지만 **시간/공간 비용**이 높을 수 있습니다. 반면, 그리디 알고리즘은 모든 경우를 다 보장하지는 못해도 **빠르게 근사해 또는 최적해에 도달**한다는 장점이 있습니다. 잘만 적용하면 **효율적(O(n log n) 내지 O(n))**으로 문제를 풀 수 있고, 복잡한 자료 구조나 많은 메모리도 필요 없는 경우가 많습니다. 요컨대, **"정확한 최적해"**가 요구되는 문제에서는 그리디 사용 전에 **조건 충족 여부**를 따져야 하지만, **제약을 만족하면 간단하고 빠른 해결책**이 될 수 있습니다.

알고리즘 성립 조건: 탐욕적 선택 속성 & 최적 부분 구조

어떤 문제가 그리디 알고리즘으로 해결 가능하려면 두 가지 핵심 **성립 조건**이 충족되어야 합니다:

- **탐욕적 선택 속성 (Greedy Choice Property)**: 이전 단계에서 한 선택이 이후 단계의 선택에 영향을 주지 않는다. 다시 말해, 매 순간의 지역 최선 선택이 앞으로의 과정에 제약이나 악영향을 주지 않아야 합니다. 그래야만 매 단계 탐욕적으로 고른 선택들을 취소하지 않고 이어나갈 수 있습니다. 이 속성이 있다면, **현재의 최적 선택이 전체 최적해 구성에 포함**된다고 믿고 진행할 수 있습니다.

- **최적 부분 구조 (Optimal Substructure)**: 문제에 대한 최종 최적해가 부분 문제들의 최적해로 구성된다. 즉, 전체 문제를 일부 부분으로 나누었을 때, **부분 문제 각각의 최적해들을 모아도 전체 최적해가 된다**는 성질입니다. 동적 계획법(DP)의 기본 전제가 되는 성질이기도 합니다. 그리디에서도 이 조건이 필요합니다. 왜냐하면 매 단계 결정할 때 남은 부분 문제가 다시 최적으로 풀릴 수 있어야, 앞선 선택과 합쳐 전체 최적해가 성립하기 때문입니다.

이 두 조건을 만족하는 문제에서는 그리디 알고리즘이 **전역적으로도 최적**인 해답을 제공할 수 있습니다. 반면 한 가지라도 성립하지 않으면, 탐욕적 접근은 오답을 낼 가능성이 높습니다. 예를 들어, **최적 부분 구조**가 성립하지 않는다면 부분적인 최선들을 합쳐도 전체 최선이 보장되지 않으므로 일반적으로 DP 등 **다른 방법**을 사용해야 합니다. 또한 **탐욕적 선택 속성**이 성립하지 않는 경우, 앞 단계의 탐욕 선택이 이후 선택지를 제한하거나 최적해 경로를 막아버려서 Greedy 해법이 실패하게 됩니다.

간단한 시각화: 아래 표는 몇 가지 예시 문제에 대해 그리디 알고리즘의 두 조건 성립 여부와, 실제로 탐욕 해법으로 최적해를 얻을 수 있는지 여부를 비교한 것입니다.

문제 유형	탐욕적 선택 속성	최적 부분 구조	탐욕 해법으로 최적 해결
회의실 배정 (한 회의실에 최대 회의)	O (성립)	O (성립)	O (가능)
동전 거스름돈 (일반적인 화폐 단위)	O (성립)	O (성립)	O (가능)
동전 거스름돈 (특이한 단위 구성)	X (불성립)	O (성립)	X (실패)
0/1 배낭 문제 (정수 무게 제한)	X (불성립)	O (성립)	X (탐욕으론 실패, DP 필요)

- 회의실 배정: 시작/종료 시간이 주어진 회의들 중 최대 개수를 선택하는 문제입니다. **탐욕적 선택 속성**과 **최적 부분 구조**가 모두 성립하여, 종료 시간이 가장 이른 회의부터 배정하는 탐욕 전략이 최적해를 보장합니다. (BOJ 1931번 문제)
- 동전 거스름돈 (일반): 한국 화폐처럼 큰 동전이 작은 동전의 배수 관계를 이루는 경우, **가장 큰 단위부터** 거슬러 주는 탐욕법이 최적입니다. 이 역시 두 조건이 충족되어 매 단계 최선이 전체 최선으로 이어집니다. (BOJ 11047번 문제)
- 동전 거스름돈 (특수): 만약 화폐 단위가 배수 관계가 아닌 경우를 생각해봅시다. 예를 들어 1원, 4원, 6원짜리 동전들이 있을 때 8원을 거슬러주면, 탐욕적으로는 6+1+1 = 3개 동전을 쓰지만 최적해는 4+4 = 2개로 가능했습니다. 이처럼 **탐욕적 선택 속성**이 깨지는 경우(한 번의 잘못된 선택이 이후 최적 구성을 방해) Greedy는 최적해를 찾지 못합니다.
- 0/1 배낭 문제: 무게 제한이 있는 배낭에 물건을 정수 단위로 담아 가치 합을 최대로 하는 문제입니다. 가치 대비 무게 비율 등 **탐욕적인 기준으로 선택**하면 일부 경우 최적해를 놓칩니다. 최적 부분 구조는 있지만 탐욕적 선택 속성이 없으므로 Greedy가 실패하고, DP로 해결해야 합니다. 반면 **분할 가능 배낭 문제(fractional knapsack)**의 경우에는 물건을 쪼갤 수 있어 탐욕 선택이 가능하며, 그리디로 최적해를 얻습니다.

요약하면, **탐욕 알고리즘을 적용하려면 해당 문제가 위 두 조건을 만족하는지 먼저 검증**해야 합니다. 만족한다면 그리디는 매우 효과적인 해결책이 되지만, 그렇지 않으면 탐욕으로 구한 해는 부분 최적에 머무를 위험이 큽니다.

그리디 알고리즘 적용 사례

Greedy 전략이 잘 들어맞는 대표적인 문제들은 많습니다. 그 중 익숙한 사례 몇 가지를 살펴보면:

- **활동 선택 문제 (회의실 배정 문제)**: 여러 활동(interval) 중 최대 개수를 수행하기 위해, "가장 빨리 끝나는 활동부터 선택" 하는 탐욕법을 쓸 수 있습니다. 이는 매 단계 **가장 빨리 끝나는 회의**를 골라 남은 시간에 최대 활동을 배치할 수 있게 해주며 최적해를 보장합니다.
- **동전 거슬러주기 문제**: 앞서 언급한 대로, 일반적인 화폐 단위에서는 "가치가 가장 큰 동전부터" 선택하는 전략이 최적입니다. 남은 금액을 줄이면서 동전 개수를 최소화하지요. 이 문제는 Greedy의 고전적인 예시로, BOJ 11047 **동전 0** 문제도 동일한 맥락입니다.
- **최소 신장 트리(MST)**: 그래프에서 MST를 구하는 Kruskal 알고리즘은 "남은 간선 중 가장 가중치 작은 간선부터 추가"하는 탐욕 전략입니다. 이 전략이 옳다는 것은 그래프 이론에서 **컷 속성**으로 증명됩니다(다음 절 참고). Prim 알고리즘도 "현재 트리에 가장 비용 작은 간선 추가"라는 greedy 선택입니다.
- **허프만 코딩**: 데이터 압축에서 문자별 가변 길이 코드를 할당하는 문제는 "가장 빈도가 낮은 두 노드를 계속 합치기"라는 탐욕적 합병으로 최적 코딩 트리를 만듭니다.
- **다익스트라 알고리즘**: 그래프 최단 경로에서, 방문하지 않은 정점 중 현재까지 알려진 최단 거리가 가장 짧은 정점을 선택해 인접 업데이트를 하는 과정도 일종의 탐욕입니다. (이때는 음수 간선이 없고 최적 부분 구조 성질이 만족되어야 올바르게 동작합니다.)

이 밖에도 **작업 스케줄링**, **배낭 문제(분할 가능)**, **문자열 기초적 정렬 문제** 등 다양한 곳에 Greedy가 쓰입니다. 핵심은 문제가 Greedy Choice Property와 Optimal Substructure를 내재하고 있는가를 판별하는 것입니다. 이러한 구조를 파악하면, **복잡한 완전 탐색이나 DP 없이도 손쉬운 규칙으로 문제 해결**이 가능합니다. Greedy의 매력은 바로 이 **간결함**과 **효율성**입니다.

그리디 알고리즘이 실패하는 경우와 반례

반대로, 탐욕 알고리즘이 통하지 않는 문제들은 어떤 특징을 가질까요? 일반적으로는 앞서 말한 조건 중 하나 이상을 만족하지 못하기 때문입니다. 몇 가지 전형적인 Greedy 실패 사례를 살펴보겠습니다.

- **0/1 Knapsack (배낭 문제)**: 물건을 쪼갤 수 없는 배낭 문제에서, 단위 무게당 가치가 가장 높은 물건부터 담는 탐욕적 접근은 항상 최적해를 주지 않습니다. 예를 들어 무게 5에 가치 50인 물건과 무게 4에 가치 40, 무게 1에 가치 1인 물건이 있고 배낭 용량이 5라면, 단위가치로는 둘 다 10이어서 탐욕 선택이 애매합니다. 하지만 경우에 따라 더 낮은 비율의 물건이 전체 가치를 높일 수 있습니다. 이 문제는 **탐욕적 선택 속성**이 없기에 DP로 풀어야 합니다.
- **특정 동전 거슬러주기**: 앞서 언급한 특수한 화폐 단위처럼, Greedy가 최적 보장을 못하는 경우입니다. 이런 경우는 **그리디 해가 부분 최적에 멈추고, 나중에 그 선택을 바꾸지 못해 최적해를 놓치는 패턴**입니다. 해결책은 보통 **DP 혹은 BFS**로 모든 경우를 탐색하는 것입니다.
- **가중치가 있는 활동 선택 문제**: 각 활동에 이익(가중치)이 붙고, 이익 합 최대화가 목표라면(Weighted Interval Scheduling), 단순히 짧은 활동부터 하는 Greedy는 실패합니다. 이 문제는 최적 부분 구조는 있지만 탐욕적 선택 속성이 없어 DP로 풀어야 합니다. (짧은 회의를 많이 하는 게 이익 합 최대라고 볼 수 없어서입니다.)
- **그래프 최단 경로 (음수 간선 존재)**: 다익스트라 알고리즘은 탐욕적이지만 음수 가중치가 있으면 실패합니다. 이는 탐욕 선택 속성이 깨지기 때문입니다. 이 경우 벨만-포드처럼 **모든 간선 반복 완화**를 해야 정확한 결과를 얻을 수 있습니다.

이처럼 **그리디 알고리즘의 적용이 실패할 수 있는 경우, 대안으로 동적 계획법(DP)이나 백트래킹/브루트포스 등의 포괄적인 탐색** 기법이 필요합니다. 현실에서는 Greedy로 빠르게 근사값을 구하고, DP로 정확한 값을 구하는 식으로 **상호 보완**하기도 합니다. 예를 들어, NP-난이도 문제에서는 최적해 대신 Greedy로 얻은 근사해로 만족하기도 합니다. 하지만 우리 알고리즘 스터디에서는 **Greedy가 언제 통하고 언제 통하지 않는지**를 명확히 구분하는 연습이 중요합니다.

탐욕 알고리즘의 정당성 증명 전략: 교환 논법 & 컷 속성

Greedy 알고리즘을 설계했다면, **왜 이 방법이 최적해를 보장하는지 증명**해야 할 때가 있습니다. 탐욕적 방법의 정당성을 증명하는 것은 때로 알고리즘 설계보다 어려울 수 있는데, 이를 위한 몇 가지 일반적인 **증명 전략**이 알려져 있습니다. 여기서는 대표적으로 **교환 논법(exchange argument)**과 **컷 속성(cut property)** 두 가지를 소개합니다.

- **교환 논법 (Exchange Argument)**: 가장 널리 쓰이는 Greedy 증명 기법입니다. 방법은 다음과 같습니다: 우선 어떤 **가상의 최적해**를 하나 가정합니다. 그리고 그리디 알고리즘이 만들어낸 해답과 최적해를 비교하면서, **필요하면 둘의 차이를 줄여가는 교환을 수행**합니다. 이때 교환은 최적해의 일부 선택을 그리디 해답의 선택으로 바꾸는 작업입니다. 교환을 통해 **해의 품질이 나빠지지 않음**을 보이면, 궁극적으로 그리디 해답으로도 최적해에 도달할 수 있음을 증명하게 됩니다. 예를 들어, 회의실 배정 문제의 증명에서, 만약 최적해가 그리디가 선택한 회의를 포함하지 않는다면 그 중 가장 빨리 끝나는 회의를 그리디 해의 회의로 교체해도 전체 개수는 줄지 않음을 보입니다. 이렇게 점진적으로 최적해를 Greedy 해와 동일하게 **교환**해 나가 최적성를 입증하는 것입니다.

- **컷 속성 (Cut Property)**: 이는 주로 **최소 신장 트리(MST)** 문제의 그리디 증명에 활용되는 개념입니다. 컷 속성은 그래프를 둘로 분할하는 임의의 **컷(cut)**에 착안합니다. 컷을 가로지르는 간선들 중 **가장 가중치가 작은 간선은 MST의 일부**라는 것이 컷 속성입니다. 이 성질을 이용하면, Kruskal 알고리즘처럼 가장 작은 간선부터 선택해 나가는 탐욕 전략이 옳음을 보장할 수 있습니다. 증명은 **귀류법**으로 진행되는데, 만약 최솟값 간선 e를 MST에 넣지 않고 대신 더 큰 간선 e'을 넣었다고 가정하면, 그 e'를 e로 대체함으로써 더 비용이 줄어든 트리를 만들 수 있어 모순이 생긴다는 식입니다. 결국 **모든 컷에 대해 안전한 선택**(가장 작은 간선 추가)이 가능하므로, 그리디 선택들이 누적되어도 최종적으로 MST 최적해가 완성됩니다. 컷 속성과 쌍을 이루는 **사이클 속성**(임의의 사이클에서 가장 가중치 큰 간선은 MST에 포함되지 않는다)도 MST 탐욕 증명에 쓰이며, Prim 알고리즘의 정당성 등에 활용됩니다.

이 밖에도 **"Greedy stays ahead"** 같은 개념적 증명, **프루프 바이 인덕션(귀납)** 등 다양한 접근이 상황에 따라 쓰입니다. 그러나 공통점은 **그리디 알고리즘 해답과 가상의 최적해를 비교**하여, Greedy 해가 열등하지 않음을 보이는 것입니다. 탐욕 알고리즘을 공부할 때는 각 문제별로 이러한 증명 아이디어까지 이해해두면 좋습니다. 이는 단순 암기보다는 "왜 이 선택이 항상 최적을 보장할까?"를 논리적으로 생각해보는 연습입니다.

시간 복잡도 및 구현 시 장단점

탐욕 알고리즘의 시간 복잡도는 문제마다 다르지만, 일반적으로 매우 양호한 편입니다. 많은 Greedy 해법은 **한 번 정렬(O(n log n)) 후 한 번 선형 스캔(O(n))**으로 끝나곤 합니다. 예를 들어 회의실 배정 문제도 회의 정렬(O(N log N)) + 한 번 탐색(O(N))으로 해결했습니다. 동전 거슬러주기 문제는 정렬할 필요도 없이 O(N) (동전 가짓수만큼 반복)으로 끝납니다. 만약 우선순위 큐(힙)을 사용한 탐욕 (예: Prim, Dijkstra) 경우에도 일반적으로 **O(N log N)** 정도의 시간으로 충분히 빠릅니다. 이처럼 **탐욕법은 현실적인 입력 크기에 대해 충분히 효율적**인 경우가 많습니다. 게다가 동적 계획법이 종종 지니는 **높은 메모리 사용량**(DP 테이블 등)도, Greedy는 그러한 테이블 없이 현재 상태만 유지하며 진행하므로 메모리 효율도 좋습니다.

장점을 정리하면:
- **단순한 로직**: 복잡한 상태 전환이나 많은 조건 없이, 정의한 규칙에 따라 순차 처리하므로 구현이 쉽습니다.
- **빠른 속도**: 결정 과정이 지역적이어서 일반적으로 많은 탐색을 하지 않고도 답을 도출합니다.
- **적은 자원 사용**: 추가 자료구조나 캐싱 없이 진행되므로 메모리 사용이 작습니다.
- **직관성**: 문제의 특성을 파악했다면 사람의 직관으로 풀이 전략을 예측하기 쉽습니다.

단점도 물론 존재합니다:
- **적용 범위 한정**: 모든 문제가 탐욕으로 풀리지는 않습니다. 문제에 그리디 조건이 성립하는지 판단이 모호할 때가 많고, 잘못 적용하면 틀린 답을 낼 위험이 있습니다.
- **정당성 증명이 필요**: Greedy 해법을 고안했다면 왜 최적해가 나오는지 증명 또는 강한 확신이 필요합니다. 그렇지 않으면 불안정한 방법이며, 증명이 어려운 경우도 많습니다.
- **전역 최적 보장 어려움**: 탐욕은 본질적으로 지역 판단이므로, 복잡한 제약이 있는 문제에서는 최적해를 놓치기 쉽습니

다. 이때는 백트래킹이나 DP 등 더 확실한 방법을 선택해야 합니다.
- **해법 도출의 어려움**: 어떤 문제에 탐욕 전략이 통할지 알아내는 것도 도전입니다. 기준 선택을 조금만 잘못 정하면 오답이 되므로, 적절한 탐욕 기준(정렬 키 등)을 찾는 것은 고도의 통찰을 요합니다.

결국 **Greedy 알고리즘은 "양날의 검"**입니다. 조건에 잘 맞으면 이보다 훌륭한 무기가 없지만, 섣불리 휘두르면 놓치는 것이 생깁니다. **문제의 구조적 특성을 파악하여 탐욕법이 적합한지 판단**하고, 적합하다면 안심하고 빠르게 풀면 됩니다. 반대로 애매하면 작은 반례라도 찾아보고, 조건 불만족 시 다른 방법으로 전환하는 유연함이 필요합니다.

예제 코드 – 회의실 배정 문제 (BOJ 1931)

탐욕 알고리즘의 전형적인 예제로 꼽히는 **회의실 배정** 문제를 간단한 코드로 구현해보겠습니다. 이 문제는 여러 회의의 시작시간과 종료시간이 주어질 때 **한 개의 회의실로 가장 많은 회의를 진행**하는 스케줄을 짜는 것입니다. 그리디 전략은 "종료 시간이 가장 이른 회의부터 배정"하는 것이었지요. 이를 코드로 나타내면 다음과 같습니다 (Java와 Python 두 가지 버전):

Python 구현:

```python
# 입력: N과 각 회의의 시작시간, 종료시간 쌍 (meetings 리스트)
meetings = sorted(meetings, key=lambda x: (x[1], x[0]))  # 종료시간 기준 정렬 (1차), 시작시간 (2차)
count = 0
current_end = 0
for start, end in meetings:
    if start >= current_end:
        count += 1
        current_end = end
print(count)  # 출력: 최대 사용할 수 있는 회의 개수
```

Java 구현:

```java
import java.util.*;
public class MeetingScheduler {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int N = sc.nextInt();
    int[][] meetings = new int[N][2];
    for (int i = 0; i < N; i++) {
      meetings[i][0] = sc.nextInt(); // 시작 시간
      meetings[i][1] = sc.nextInt(); // 종료 시간
    }
    // 종료 시간 기준, 같으면 시작 시간 기준 정렬
    Arrays.sort(meetings, new Comparator<int[]>() {
      public int compare(int[] a, int[] b) {
        if(a[1] == b[1]) return a[0] - b[0];
        return a[1] - b[1];
      }
    });
    int count = 0;
    int currentEnd = 0;
```

```
        for (int i = 0; i < N; i++) {
            if (meetings[i][0] >= currentEnd) {
                count++;
                currentEnd = meetings[i][1];
            }
        }
        System.out.println(count);
    }
}
```

위 코드는 간략화를 위해 입력 처리 등을 최소화했지만, 핵심 로직은 동일합니다. **정렬**로 가장 빨리 끝나는 회의 순으로 나열한 뒤, 순서대로 회의를 예약합니다. 각 회의를 선택할 때 이전에 선택한 회의의 종료시간 `current_end`를 기준으로 **겹치지 않는 가장 이른 다음 회의**를 찾아나가는 방식입니다. 이 탐욕적 선택으로 얻은 결과가 회의 최대 개수를 만족함은 앞서 설명한 대로 증명 가능합니다.

실습 문제

Greedy 알고리즘의 감을 더 익히기 위해 다음의 백준 온라인 저지(BOJ) 연습 문제들을 풀어보세요. 각각 위에서 언급한 개념을 활용한 문제들입니다.

- **BOJ 1931 – 회의실 배정**: 위에서 다룬 문제와 동일합니다. 탐욕 알고리즘으로 해결하세요.
- **BOJ 11047 – 동전 0**: 여러 동전들을 이용해 특정 금액을 최소 동전 개수로 거슬러주는 문제입니다. 가장 큰 단위 화폐부터 사용하는 Greedy 전략을 적용해보세요.
- **BOJ 11399 – ATM**: 사람들의 인출 시간 계산 문제로, 대기 시간을 최소화하려면 **짧은 작업부터 처리**하는 것이 핵심입니다. 이는 탐욕적으로 처리 순서를 정하는 문제입니다.

위 문제들은 모두 **탐욕적 정렬 기준을 무엇으로 잡는지가 포인트**입니다. 각 문제를 풀면서 Greedy의 사고 방식을 연습해봅시다. 가능하다면 왜 탐욕 해법이 최적인지도 스스로 설명해보세요.

또한 추가로 **반례가 존재하는 Greedy 시도**도 실험해보는 것을 권장합니다. 예를 들어 동전 문제에서 한국 화폐 단위가 아니었다면? ATM 문제에서 다른 순서로 하면 어떻게 손해가 발생하는지? 이런 생각을 통해 Greedy의 한계도 함께 이해할 수 있습니다.

간단한 팁: Greedy는 **정렬**과 자주 짝을 이룹니다. 어떤 기준으로 정렬했을 때 문제 상황을 가장 잘 반영하는지 고민해보는 습관을 가지세요. 그리고 현재 단계의 최선 선택이 이후 단계를 망치지 않는지 항상 따져보는 것, 그것이 탐욕 알고리즘 설계의 전부라고 해도 과언이 아닙니다.

---

3. 본론2 – 구현(시뮬레이션)

개념 이해: 구현 & 시뮬레이션 문제란?

구현(Implementation) 문제, 흔히 **시뮬레이션** 유형이라고도 부르는 문제들은 특별한 알고리즘보다는 **"문제에서 요구한 대로 정확히 동작을 코드로 옮기는 것"**이 핵심인 경우를 말합니다. 다시 말해, 주어진 시나리오나 규칙을 실제로 컴퓨터 안에서 **시뮬레이션**해보는 것입니다. 이때 요구사항을 하나하나 빠뜨리지 않고 충실히 재현하는 것이 가장 중요합니다.

구현형 문제의 예로는 다음과 같은 것들이 있습니다:

- **격자상의 이동**: 예를 들어 2차원 배열 맵에서 캐릭터를 움직이거나 게임을 진행하는 로직을 구현합니다. (뱀 게임, 로봇 청소기 등이 대표적)
- **이벤트 시뮬레이션**: 시간이나 순서에 따라 발생하는 이벤트들을 차례로 모사해봅니다. (예: 문제에서 "X초에 방향 전환" 등의 이벤트가 주어지면, 초 단위로 진행하면서 이벤트를 처리)
- **문자열 파싱/처리**: 문자열에 주어진 명령어를 해석해 자료구조를 조작하거나 결과를 출력합니다. (예: 배열에 대한 R, D 명령 처리하는 문제 등)
- **단순 완전 탐색**: 경우에 따라 해 볼 수 있는 모든 행동을 구현으로 시뮬레이션하는 것도 포함됩니다. (예: 게임에서 가능한 모든 움직임 따라 해보기 등)

구현 문제에서는 **자료 구조, 언어 문법 활용, 사소한 예외 처리** 등이 두루 요구됩니다. 알고리즘적인 고난도보다는, **정확함과 꼼꼼함**이 승부를 가릅니다. 실제로 "**삼성 SW 역량테스트**"라 불리는 코딩 테스트에서 이런 시뮬레이션 문제가 자주 등장하는 것으로 유명합니다. 이는 복잡한 알고리즘을 몰라도 풀 수 있지만, **조건 이해와 코드 구현 능력**을 검증하기 위함입니다.

시뮬레이션이라는 말 그대로, 문제 지문이 일러준 상황을 컴퓨터 안에 재현(simulate)하면 됩니다. 이를 위해서는 우선 **상태(state)**를 잘 정의해야 합니다. 상태란 시뮬레이션 시점에 우리가 추적해야 할 모든 정보를 뜻합니다. 예를 들어 뱀 게임이라면 현재 뱀의 머리/꼬리 위치와 몸의 좌표들, 진행 방향, 시간, 현재 점수 등이 상태가 될 것입니다. 로봇 청소기라면 로봇의 좌표와 방향, 청소 여부 판정 배열 등이 상태겠지요. 그리고 시뮬레이션 문제에서는 **상태가 시간이나 단계에 따라 어떻게 변하는지(전이)**를 모델링하는 것이 중요합니다. 매 초 혹은 한 턴마다 상태 변수들이 어떤 규칙으로 변하는지 알고리즘으로 구현합니다.

요약: 구현/시뮬레이션 문제는 한마디로 "특정 상황을 그대로 코드로 옮겨라" 입니다. 알고리즘 설계 능력보다는 **문제를 정확히 해석하고, 컴퓨터가 이해할 수 있는 형태로 옮기는 능력**이 중요합니다. 따라서 **디버깅 능력**, **예외 상황 처리**, **효율적인 코딩**이 요구됩니다. 난이도 높은 구현 문제일수록 고려해야 할 조건이 많고 코드도 길어지므로, 체계적으로 접근하는 습관을 들여야 합니다.

구현 전략: 자료구조 선택, 모듈화, 디버깅

어떤 시뮬레이션 문제를 받았을 때, 바로 코딩을 시작하기보다는 **다음과 같은 전략**으로 접근하면 실수를 줄일 수 있습니다:

1. **요구사항 정리**: 먼저 지문의 요구사항을 하나씩 정리하세요. 입력 형식, 출력 형식, 처리해야 할 조건들을 머릿속이나 종이에 순서대로 써봅니다. 예를 들어 "초기 로봇 위치는 (r,c)이고 북쪽을 바라본다", "매 턴마다 왼쪽으로 회전하며 앞칸 확인" 등 규칙을 단계별로 나열합니다.
2. **시나리오 세분화**: 전체 프로세스를 여러 단계로 나눠봅니다. **상태 변화**가 뚜렷한 문제라면 한 턴(또는 한 초)을 기준으로 어떤 일이 벌어지는지 기술합니다. 혹은 명령어 하나 단위로 상태가 어떻게 변하는지 적습니다. 이 과정을 통해 놓칠 수 있는 세부 조건을 발견하기도 합니다.
3. **필요한 변수/자료구조 설계**: 구현에 앞서, 어떤 데이터를 저장하고 어떻게 구조화할지 결정합니다. 2차원 그리드라면 `int[][]` 배열이 필요하고, 객체의 위치는 (x,y) 좌표 변수로 관리할 수 있습니다. 방향 이동이 빈번하다면 `dx[], dy[]` 배열을 정의해 방향 전환을 쉽게 하는 방법을 생각합니다. 또한 명령어 시뮬레이션의 경우 **큐(Deque)**나 **문자열 파싱용 배열/리스트** 등을 선택합니다. 예를 들어 BOJ 5430 **AC** 문제에서는 양방향 삭제에 유리한 덱(deque)을 쓰는 것이 핵심입니다.
4. **모듈화 및 함수화**: 시뮬레이션 로직이 길어질 경우, 코드의 일부를 함수로 빼거나 적절히 모듈화해서 작성합니다. 예를 들어 "좌회전하기" 동작, "한 칸 이동하기" 동작 등을 함수로 만들면 가독성이 높아집니다. 이는 디버깅에도 도움이 됩니다. 각 함수별로 동작을 단위 테스트하기 쉽기 때문입니다.
5. **예외 및 엣지 케이스 처리**: 구현 문제에서는 사소한 예외를 놓치면 오답이 됩니다. 예를 들어 시뮬레이션 도중 배열 인덱스가 범위를 벗어나는 경우(벽에 부딪히는 상황) 항상 조건문으로 걸러야 합니다. 입력이 0일 때 처리나, 출력 양식에 불필요한 공백/콤마가 들어가지 않도록 하는 것 등도 중요합니다. 여러 가지 경계 조건(최소값, 최대값, 빈 입력 등)을 직접 생각해보고 코드에 반영해야 합니다.

6. **디버깅**: 복잡한 구현일수록, 한 번에 완벽히 작성하기 어렵습니다. 적당한 중간 단계에서 출력이나 로그를 찍어보면서 상태 변화를 확인하는 것이 좋습니다. 예컨대 로봇 청소기 문제에서는 몇 턴 진행 후 로봇 위치와 방향을 출력해보며 예상과 맞는지 체크할 수 있습니다. 또한 작은 **예제 시나리오**를 스스로 만들어 손으로 시뮬레이션해보고, 프로그램 결과와 비교해보는 것도 큰 도움이 됩니다. (예: 뱀 문제에서 5x5 작은 보드로 직접 움직여보기)

이렇게 체계적으로 접근하면 **필요 이상으로 헤매지 않고 정확한 구현**에 다가갈 수 있습니다. 구현 문제는 "한 끗 차이" 실수가 많기 때문에, 애초에 치밀하게 설계하고 꼼꼼히 검증하는 습관이 중요합니다.

대표적인 구현(시뮬레이션) 문제 유형

위에서 언급한 사례들을 조금 더 구체적으로 살펴보겠습니다. 알고리즘 문제 풀이에서 자주 등장하는 시뮬레이션 유형들입니다:

- **격자(grid) 기반 시뮬레이션**: N×M 격자 맵이 주어지고, 그 위에서 캐릭터나 객체가 이동하거나 무엇인가를 수행하는 문제입니다. 예를 들어 **뱀(BOJ 3190)** 문제에서는 뱀이 매 초 머리를 움직이고 사과를 먹는 과정을 시뮬레이션해야 합니다. **로봇 청소기(BOJ 14503)** 문제도 로봇이 격자를 돌아다니며 청소하는 과정을 구현합니다. 이러한 문제에서는 방향 이동 처리가 중요합니다. 보통 방향을 상/우/하/좌 (또는 북/동/남/서) 4가지로 두고, `dx`, `dy` 배열을 설정해두면 유용합니다. 예를 들어 방향 인덱스 0,1,2,3을 북,동,남,서로 정하고 다음처럼 배열을 둘 수 있습니다:

```
int[] dx = {-1, 0, 1, 0};  // 행 변화: 북,동,남,서
int[] dy = {0, 1, 0, -1};  // 열 변화: 북,동,남,서
```

이렇게 하면 현재 방향 `dir`에서 왼쪽으로 회전하는 것은 `dir = (dir + 3) % 4` (혹은 -1이 아니면 -1)로 구현 가능하고, 오른쪽 회전은 `dir = (dir + 1) % 4`로 쉽게 처리할 수 있습니다. 이동은 `nx = x + dx[dir]; ny = y + dy[dir];`로 한 칸 전진이 됩니다. 격자 문제에서는 이런 **방향 관리 기법**과 **벽 검사(범위 검사)**가 핵심 포인트입니다. 또한 뱀 문제처럼 자기 몸과의 충돌 체크가 필요하면, 현재 뱀 몸 좌표들을 `큐`나 `리스트`에 저장해두고 이동 시 업데이트하는 로직을 구현합니다.

- **이벤트 기반 시뮬레이션**: **Snake(뱀)** 문제에서도 드러나지만, 어떤 이벤트가 시간 축에 따라 발생할 때, 그 시간까지 시뮬레이션을 진행하고 이벤트를 처리하는 방식입니다. 뱀 문제의 경우 X초에 방향전환 이벤트가 입력으로 주어지는데, 이를 처리하려면 **전체 시간을 while문으로 돌거나**, 이벤트 시점까지 미리 뱀을 움직이는 방식이 필요했습니다. 이벤트를 관리하기 위해 `Queue`나 `우선순위 큐`에 (발생시각, 이벤트내용) 형태로 저장해두고, 시뮬레이션 루프에서 현재 시간과 비교해 일치하면 이벤트를 꺼내 처리하는 식으로 구현할 수 있습니다.
  이처럼 시간에 따라 상태 변화가 예약된 문제는 **시계 흐름**을 코드에 녹여내야 합니다. 즉, `for (t=1; t<=T; t++)` 같은 루프를 돌며 tick 단위로 상태를 업데이트 하는 방법이 일반적입니다. 시간 단위가 크면 이벤트 간 간격을 건너뛰는 최적화도 필요할 수 있지만, 문제에 따라 적절히 결정합니다.

- **명령어 파싱 & 자료구조 시뮬레이션**: 입력으로 문자열 명령 시퀀스가 주어지고, 그에 따라 자료구조나 데이터를 조작하는 문제 유형입니다. 예로 **AC (BOJ 5430)** 문제가 있습니다. 이 문제는 문자로 이루어진 함수 시퀀스(`R`과 `D`)를 배열에 적용하는 것입니다. 문자열 처리와 덱(Deque)를 활용한 시뮬레이션이 요구되지요.
  이 유형에서는 **문자열을 효율적으로 파싱**하는 것과, **요구 기능을 지원하는 적절한 자료구조 선택**이 핵심입니다. AC 문제의 경우 정수를 보관하는 배열에 대해 앞뒤 제거(D)와 뒤집기(R)를 수행해야 하는데, 일반 리스트로 구현하면 R 연산 시 매번 리스트를 뒤집어야 해서 비효율적입니다. Greedy와도 연관되는 생각이지만, **굳이 실제로 뒤집지 않고, D 연산시 어느 쪽에서 뺄지만 결정하는 방식**으로 풀면 효율적입니다. 구체적으로, `reverse` 상태를 나타내는 불리언 변수를 두고 R 명령 시 그 값을 토글(toggle)만 합니다. 그리고 D 명령이 들어오면, `reverse == false`일 때는 앞쪽에서 pop, `reverse == true`일 때는 뒤쪽에서 pop을 하는 식입니다. 이렇게 하면 실제 배열을 뒤집지 않고도 동작을 시뮬레이션할 수 있어, 긴 명령어 시퀀스도 빠르게 처리 가

능합니다. Python에서는 `collections.deque`를 쓰면 양쪽 pop이 O(1)로 가능하고, Java에서는 `ArrayDeque` 혹은 `LinkedList`를 덱으로 활용할 수 있습니다.

예시 – AC 문제 간단 구현 (Python):

```
from collections import deque
commands = "RDD"  # 예시 명령어
data = deque([1, 2, 3, 4])  # 예시 배열
rev = False
error_flag = False
for c in commands:
    if c == 'R':
        rev = not rev
    elif c == 'D':
        if not data:
            error_flag = True
            break
        if rev:
            data.pop()          # 뒤집힌 상태: 뒤에서 제거
        else:
            data.popleft()     # 일반 상태: 앞에서 제거
if error_flag:
    print("error")
else:
    if rev:
        data.reverse()         # 최종 출력 시에는 실제 순서로 뒤집기
    print("[" + ",".join(map(str, data)) + "]")
```

위 코드는 하나의 테스트케이스에 대한 동작을 보여줍니다. 실제 BOJ에서는 여러 케이스를 처리해야 하지만 핵심 로직은 이렇습니다. **자료구조 선택과 구현 기술(문자열 처리)**이 성능을 좌우하는 전형적인 구현 문제입니다.

- **시뮬레이션 + 완전 탐색 결합**: 어떤 문제는 시뮬레이션 자체는 단순하지만 여러 초기 조건이나 여러 선택지를 다 시험해봐야 하는 경우가 있습니다. 이런 경우 2중 루프 이상의 완전 탐색과 시뮬레이션을 결합하게 되며, 코딩이 길고 복잡해질 수 있습니다. 하지만 기본 원리는 동일합니다: "요구된 동작을 정확히 구현"하고 이것을 여러 번 반복하는 것입니다. 대표적으로 퍼즐 해결이나 보드 게임 구현 문제 등이 그렇습니다.

시간 복잡도 분석 및 최적화 전략

구현 문제에서도 시간 복잡도 분석은 필요합니다. 왜냐하면 시뮬레이션을 글자 그대로 했다가는 **시간 초과**가 날 수 있는 경우도 있기 때문입니다. 일반적으로는 문제 입력 크기에 맞춰 **모든 동작을 1배속으로 다 해도** 괜찮게 설정되지만, 가끔 함정처럼 크거나 반복 많은 입력이 주어집니다.

예를 들어 **뱀(3190)** 문제의 경우, 최악에는 몇만 초 이상 게임이 진행될 수도 있습니다. (보드 크기 N 최대 100이라 뱀이 꼬불꼬불 움직이면 N^2까지 몸이 자라 이만큼 진행하기도 합니다.) 이러한 경우 초 단위로 while문을 돌리면 충분히 커버할 만하지만, **10^7 이상의 스텝**을 처리해야 한다면 언어에 따라 살짝 위험할 수도 있습니다. (C++처럼 빠른 언어는 버티지만 파이썬은 1천만 loop면 살짝 걱정해야 합니다.) 다행히 3190의 입력 제한은 그렇게 크진 않지만, 비슷한 상황에서라면 **시뮬레이션을 너무 느리게 짜지 않았는지** 확인해야 합니다.

구현 문제의 병목은 주로 **자료구조 연산과 반복 횟수**에서 나타납니다. AC 문제를 예로 들면, R이 나올 때마다 O(n) 리스트 뒤집기를 하면 명령어 길이 * 배열길이 만큼 시간이 들지만, 개선한 대로 하면 명령어 길이 + 배열길이 수준으로 해결됩니다. 따라서 구현을 할 때도 어떤 연산이 자주 반복되는지, 그 비용은 얼마인지 따져보고 개선 여지가 있으면 개선해야 합니다.

또한 입출력 양이 많다면 입출력을 빠르게 하는 방법을 고려해야 합니다. Java에서는 Scanner 대신 BufferedReader, BufferedWriter를 쓰는 것이 좋고, Python에서도 input() 대신 sys.stdin.readline을 사용하는 등 신경 씀으로써 전체 시간을 단축할 수 있습니다. 아무리 시뮬레이션 내용이 올바르게 구현되어도, I/O 병목으로 시간초과가 나는 경우도 있으니 유의합니다.

대응 전략 요약:
- 시뮬레이션 단계가 매우 많다면, 비효율적인 중첩 루프가 없는지 확인한다. (필요하면 수학적 계산으로 일부 단계를 건너뛰는 것도 가능하지만, 이는 고난도에 해당)
- 자료구조 연산의 복잡도를 고려한다. (리스트에서 맨 앞 원소 뽑는 O(n) 대신 deque 사용 등)
- 필요없는 연산을 줄인다. (조건 체크를 미리 끝내거나, 상태 변화가 없으면 루프 중단 등)
- 입출력 최적화도 잊지 않는다.

예를 들어 **로봇 청소기(14503)**의 경우 맵 크기가 50×50 이하여서 단순 반복도 거뜬합니다. 하지만 **AC(5430)** 문제는 배열 길이가 최대 100,000까지 갈 수 있어, 잘못하면 한 테스트에서 수십만번 이상의 연산이 중첩될 수 있습니다. 그러므로 AC 문제에서 `reverse`를 진짜 수행하면 최악의 경우 100,000길이 배열을 수만 번 뒤집어야 될 수도 있어 심각한 비효율입니다. 이걸 **논리적인 처리로 대체**한 것이 위의 deque + reverse-flag 아이디어였죠. 이런 식으로 각 문제마다 핵심 동작의 효율화를 고민하면 좋습니다.

예제 코드 – AC 문제 (BOJ 5430)

구현 및 시뮬레이션의 예제로, 문자열 명령을 파싱하여 배열을 조작하는 **AC** 문제의 코드를 살펴보겠습니다. 이 문제에서는 앞서 설명한 deque를 이용한 풀이 전략을 코드로 옮기는 것이 관건입니다. (Java와 Python 예시)

Python 구현:

```
from collections import deque
import sys

input = sys.stdin.readline  # 빠른 입력
T = int(input().strip())
for _ in range(T):
    commands = input().strip()
    n = int(input().strip())
    arr_input = input().strip()
    # 배열 파싱: "[]" 또는 "[1,2,3]"
    if n == 0:
        data = deque()
    else:
        # 양 끝 대괄호 제거하고, 쉼표로 분리
        arr_list = arr_input[1:-1].split(',')
        data = deque(arr_list)  # 문자열로 저장되지만 어차피 출력에도 사용
    error_flag = False
    rev = False
    for cmd in commands:
        if cmd == 'R':
```

```
                rev = not rev
            elif cmd == 'D':
                if not data:   # 비어있는데 삭제 시도 -> 에러
                    error_flag = True
                    break
                if rev:
                    data.pop()          # 뒤집힌 상태: 뒤에서 삭제
                else:
                    data.popleft()    # 정상 상태: 앞에서 삭제
        if error_flag:
            print("error")
        else:
            # 최종 출력
            if rev:
                data.reverse()
            # deque를 문자열로 출력 (요구 형식: [x,y,z])
            print("[" + ",".join(data) + "]")
```

Java 구현:

```
import java.io.*;
import java.util.*;
public class ACSimulator {
  public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StringBuilder output = new StringBuilder();
    int T = Integer.parseInt(br.readLine().trim());
    for(int t=0; t<T; t++){
      String commands = br.readLine().trim();
      int n = Integer.parseInt(br.readLine().trim());
      // 배열 입력 파싱
      String arrLine = br.readLine().trim();
      Deque<String> deque = new ArrayDeque<>();
      if(n > 0) {
        // 문자열 형태 "[x1,x2,...,xn]" 처리
        String nums = arrLine.substring(1, arrLine.length()-1);  // 양끝 [] 제거
        String[] numArr = nums.split(",");
        for(String num : numArr) {
          deque.add(num);
        }
      }
      boolean reversed = false;
      boolean errorFlag = false;
      for(char cmd : commands.toCharArray()){
        if(cmd == 'R') {
          reversed = !reversed;
        } else if(cmd == 'D') {
          if(deque.isEmpty()) {
            errorFlag = true;
            break;
```

```
                }
                if(!reversed) {
                    deque.pollFirst();
                } else {
                    deque.pollLast();
                }
            }
        }
        if(errorFlag) {
            output.append("error\n");
        } else {
            // 출력 시 덱 내용을 순서대로 혹은 반대로 출력
            if(reversed) {
                // 반전된 상태이면 덱을 뒤에서부터 출력
                Iterator<String> it = deque.descendingIterator();
                output.append("[");
                while(it.hasNext()){
                    output.append(it.next());
                    if(it.hasNext()) output.append(",");
                }
                output.append("]\n");
            } else {
                // 정상 상태이면 앞에서부터 출력
                output.append("[");
                Iterator<String> it = deque.iterator();
                while(it.hasNext()){
                    output.append(it.next());
                    if(it.hasNext()) output.append(",");
                }
                output.append("]\n");
            }
        }
    }
    System.out.print(output);
  }
}
```

위 코드들은 BOJ 5430 AC 문제의 핵심을 구현한 것입니다. Python 버전은 간결하지만, Java 버전은 입출력 성능을 위해 BufferedReader와 StringBuilder를 활용했고 출력 포맷을 맞추기 위해 Iterator를 활용하여 덱을 출력했습니다. 핵심 로직을 보면 **R 명령 처리시 내부 데이터의 순서를 뒤집는 대신 불리언 변수로 상태만 기록**하고, D 명령 처리시 그 변수에 따라 `pollFirst` 또는 `pollLast`를 호출하는 방식입니다. 이로써 수만 번의 R 연산에도 불구하고 실제 데이터 뒤집기는 최종 출력 시 한 번만 일어나며, D 연산도 덱의 앞뒤에서 O(1)로 일어나 효율적입니다.

이 문제를 통해 알 수 있듯이, **구현/시뮬레이션에서는 자료구조의 선택과 구현상의 최적화로 성능 차이가 크게 벌어질 수 있습니다.** 초반에 문제를 해석할 때 시간 복잡도를 한 번 판단해보고, 필요시 위와 같이 최적의 접근으로 코드 구조를 짜는 습관을 가져야 합니다.

실습 문제

시뮬레이션 유형의 이해를 돕기 위해 아래의 연습 문제들을 권장합니다. 직접 구현해보며 모듈화, 예외 처리, 자료구조 선택 등을 연습해보세요.

- **BOJ 3190 – 뱀**: 유명한 **Snake 게임** 구현 문제입니다. 뱀이 시간에 따라 움직이고, 사과를 먹으면 길이가 늘어나며, 벽 또는 자기 몸에 부딪히면 종료됩니다. 방향 전환 이벤트가 주어지므로, 시간을 차례로 진행하면서 시뮬레이션하세요. 뱀의 몸은 큐(`Queue`나 `Deque`)로 관리하면 편리합니다 (머리 추가, 꼬리 제거). 격자 범위를 벗어나는지, 머리가 몸에 부딪히는지 매 이동마다 검사해야 합니다. 이 문제는 **격자 이동 + 이벤트 처리**의 종합 연습이 됩니다.
- **BOJ 14503 – 로봇 청소기**: 로봇 청소기가 주어진 규칙에 따라 방을 청소하는 과정입니다. 시뮬레이션 규칙이 다소 복잡한데, 요약하면 "현재 위치 청소 -> 왼쪽 방향으로 탐색하며 빈 칸 있으면 회전+전진, 없으면 회전만 -> 네 방향 다 막혔으면 후진 or 종료" 입니다. 이 로직을 그대로 구현하면 됩니다. 중요한 것은 **방향 전환**과 **후진 로직**을 정확히 구현하는 것이며, 청소했는지 여부를 기록하기 위해 방문 체크용 배열(혹은 입력 배열을 활용)을 써야 합니다. 이 문제로 **모듈화**(예: 왼쪽으로 돌기 함수, 후진 함수 등) 연습을 해보세요.
- **BOJ 5430 – AC**: 위에서 예제로 다룬 문제입니다. 덱과 문자열 파싱을 연습하기 좋습니다. Java라면 Scanner 대신 BufferedReader로 입력 파싱하는 법도 익혀둘 만 합니다.
- (추가 도전) **BOJ 14891 – 톱니바퀴**, **BOJ 15685 – 드래곤 커브** 등: 삼성 기출의 구현 문제로, 톱니바퀴 회전 시뮬레이션이나 드래곤 커브 그리기 등 복잡한 구현 연습을 더 하고 싶다면 도전해보세요. 이들은 구현 난이도가 높지만 도전하면서 얻는 구현 감각이 클 것입니다.

이런 문제들을 풀 때는 **항상 종이에 시뮬레이션을 그려보는 습관**을 가지세요. 예를 들어 뱀 문제나 드래곤 커브 문제 등은 좌표평면이나 격자를 직접 그려서 움직임을 따라가 보면 규칙을 이해하고 실수할 부분을 미리 캐치하기 좋습니다. 복잡한 시뮬레이션 문제일수록 도식화, 표 작성 등으로 흐름을 정리하면 코딩할 때 큰 도움이 됩니다.

시각적 예시: 로봇 청소기 문제에서, 로봇의 이동 경로를 좌표로 나열해보거나 그림으로 표시하면 작동 과정을 한눈에 볼 수 있습니다. 예를 들어 예시 입력의 로봇 이동 경로를 기록한 표를 보면 다음과 같습니다:

```
(7,4) -> (7,3) -> (8,3) -> (8,4) -> (9,4) -> (9,5) -> ...
```

(임의의 예시 열거이며, 실제 방향 전환에 따라 좌표가 바뀜) 이런 식으로 스텝별 상태 추적표를 만들어보는 것도 일종의 시각화로, 복잡한 구현을 이해하는 데 도움이 됩니다. 실제로 많은 풀이 블로그들이 중간 과정을 그림이나 표로 정리하여 이해를 돕고 있습니다.

구현 문제 팁 요약

마지막으로 구현(시뮬레이션) 문제를 잘 풀기 위한 팁을 정리합니다:

- **꼼꼼한 조건 분석**: 지문의 요구를 한 문장도 빼먹지 말고 코드로 옮긴다는 마음가짐이 필요합니다. 조건 하나 빼먹으면 오답이 되기 쉽습니다.
- **단계별 구현과 테스트**: 한 번에 모든 코드를 작성하지 말고, 기능 단위로 나눠서 구현하고 작은 입력으로 테스트하면서 진행하세요.
- **적절한 자료구조 활용**: 문제에 따라 배열, 리스트, 덱, 세트, 맵 등을 적재적소에 써야 코드가 깔끔하고 빨라집니다. 또한 언어 내장 함수를 활용할 수 있다면 최대한 활용하세요(e.g., 파이썬의 `split`, `replace` 등).
- **예외 상황 처리**: 항상 **최소값, 최대값, 빈 값, 한계 상황** 등을 염두에 두고 방어적으로 코드를 작성합니다. 필요한 경우 if문을 추가하는 것을 두려워하지 마세요. 구현 문제에서 불필요한 일반화보다는 예외 하나하나 잡는 게 현실적입니다.
- **디버깅 습관**: 출력 형식이 정확한지, 공백/콤마 위치는 맞는지 등 사소한 부분도 실제 출력 예시와 하나하나 대조해보세요. 또한 틀렸을 경우 어떤 단계에서 잘못되었을지 로그를 찍어 추적하는 것도 좋은 방법입니다.

---

4. 마무리

이번 주차에서는 **탐욕 알고리즘과 구현(시뮬레이션) 문제 해결**에 대해 깊이 있게 살펴보았습니다. **그리디 알고리즘**은 문제 해결에 있어 단순하지만 강력한 도구입니다. 핵심 개념은 탐욕적 선택 속성과 최적 부분 구조이며, 이 두 조건이 만족될 때 **로컬 최적 -> 글로벌 최적**이 성립한다는 점을 배웠습니다. 그리디의 장단점을 이해함으로써, 언제 이 기법을 쓰고 언제 피해야 할지 감을 잡을 수 있습니다. 또한 Greedy 해법의 **정당성 증명**을 통해 알고리즘적 사고력을 기르고, 반례를 통해 그 한계도 체험해보았습니다.

구현 및 시뮬레이션 파트에서는 알고리즘보다는 **문제 해석과 실천적인 코딩 스킬**이 중요함을 알게 되었습니다. 복잡한 조건을 차근차근 코드로 옮기는 연습, 자료구조와 로직을 적절히 활용하는 방법, 그리고 꼼꼼한 디버깅과 예외 처리의 중요성을 다뤘습니다. 이것들은 코딩 테스트는 물론 실제 프로그래밍 작업에서도 매우 유용한 능력입니다. 시뮬레이션 문제를 풀며 얻은 **인내심과 집중력, 구현 감각**은 향후 더 복잡한 문제를 대할 때 큰 자산이 될 것입니다.

마지막으로, **핵심 개념 요약**: Greedy는 "항상 최선 같아 보이는 것을 고르는 전략"이며, 구현 문제는 "주어진 시나리오를 빠짐없이 코드로 재현하는 작업"입니다. 둘 다 문제 해결에 필수적인 방법론이니 만큼, 다양한 문제를 통해 계속 연습해보는 것이 중요합니다.

다음 주에는 **문자열 알고리즘**에 대해 예고해드리겠습니다. 문자열 알고리즘은 문자열 내에서 특정 패턴을 찾거나, 문자열을 효율적으로 처리하는 기법들을 의미합니다. 예고편을 살짝 보자면, **KMP 알고리즘**처럼 긴 문자열에서 패턴을 빠르게 찾는 방법, **트라이(Trie)** 자료구조를 이용한 문자열 집합 처리, 그리고 **접미사 배열/트리** 같은 심화 주제까지 다룰 계획입니다. 문자열은 모든 프로그래밍의 기본 자료형 중 하나인 만큼, 해당 알고리즘을 익히면 텍스트 처리 문제에서 큰 도움을 받을 것입니다.

그럼 이번 주차 내용을 잘 복습하시고, 실습 문제도 꼼꼼히 풀어보세요. **핵심은 개념의 이해와 직접 구현**입니다. 다음 주 **문자열 알고리즘** 강의에서 다시 뵙겠습니다! 열공하세요 👍.
