개요
알고리즘 문제 5개와 알고리즘 개념 범위 요약.

대표 문제 **1.** 버블 정렬 최적화
문제 정의
길이 n의 배열 A가 주어진다.
비교 기반, 제자리 정렬(in-place), 안정성(stable)을 유지하면서 배열을 오름차순으로 정렬한다.
목표는 정렬 결과를 바꾸지 않으면서, 불필요한 비교를 줄이는 것이다.

핵심 관찰
버블 정렬은 모든 패스를 끝까지 수행할 필요가 없다.
정렬 과정 중 교환이 더 이상 발생하지 않거나, 이미 정렬이 확정된 구간은 이후 패스에서 다시 비교하지 않아도 된다.

설계 1. 조기 종료
한 패스 동안 교환이 한 번도 발생하지 않으면 배열은 이미 정렬된 상태다.
따라서 이후 패스를 수행하지 않고 즉시 종료할 수 있다.

이 설계는 입력이 이미 정렬되어 있거나 거의 정렬된 경우에 불필요한 반복을 제거한다.
최선의 경우 수행 시간은 O(n)이 된다.

설계 2. 마지막 교환 위치 기반 범위 축소
각 패스에서 마지막으로 교환이 발생한 위치 이후의 구간은 이미 정렬된 상태다.
다음 패스에서는 이 구간을 비교 대상에서 제외할 수 있다.

패스가 반복될수록 비교 범위가 점점 줄어들며,
특히 거의 정렬된 입력에서 실질적인 수행 시간이 감소한다.

복잡도 정리
시간 복잡도는 최악의 경우 여전히 $O(n^2)$이다.
그러나 입력 상태에 따라 실제 수행 시간은 크게 달라진다.
공간 복잡도는 O(1)이며, 안정성은 유지된다.

이 문제에서 확인한 점
단순한 알고리즘이라도 입력의 상태를 명확히 정의하면
불필요한 연산을 제거하는 방향으로 설계를 수정할 수 있다.
문제 정의의 미세한 변화가 알고리즘의 실제 성능에 직접적인 영향을 준다는 점을 보여주는 사례다.

대표 문제 **2.** 삽입 정렬의 평균 시간 복잡도 분석
문제 정의
길이 n의 배열 A가 주어진다.
입력 배열은 모든 순열이 동일한 확률로 발생하는 균등 분포를 따른다고 가정한다.
삽입 정렬을 적용했을 때 평균 시간 복잡도를 계산하는 것이 목표다.

핵심 관찰

삽입 정렬의 수행 시간은 각 원소가 왼쪽으로 이동하는 횟수에 의해 결정된다.
이 이동 횟수는 배열에 존재하는 역전(inversion)의 개수와 직접적으로 연결된다.
따라서 평균 시간 복잡도는 평균 역전 수를 계산하는 문제로 환원할 수 있다.

분석 관점 1. 역전 수 기반 분석
임의의 두 인덱스 $i < j$에 대해, $A[i] > A[j]$일 확률은 1/2이다.
따라서 전체 역전 수의 기대값은
$n(n-1)/2$ 쌍 중 절반인 $n(n-1)/4$가 된다.

삽입 정렬에서 각 역전은 한 번의 이동 연산으로 해소된다.
비교 연산 수 또한 이동 횟수에 비례하므로,
평균 수행 시간은 $\Theta(n^2)$이 된다.

분석 관점 2. 원소 이동 거리 기반 분석
i번째 원소는 평균적으로 자신보다 앞에 있는 $i-1$개의 원소 중 절반보다 작다.
즉, 평균적으로 $(i-1)/2$만큼 왼쪽으로 이동한다.

전체 이동 횟수의 기대값은
$\Sigma_{(i=1 \to n)} (i-1)/2 = \Theta(n^2)$이다.
이 역시 삽입 정렬의 평균 시간 복잡도가 $\Theta(n^2)$임을 보여준다.

복잡도 정리
평균 시간 복잡도는 $\Theta(n^2)$이다.
최선의 경우(이미 정렬된 입력)는 $\Theta(n)$,
최악의 경우(역순 입력)는 $\Theta(n^2)$이다.
공간 복잡도는 $O(1)$이며, 안정성은 유지된다.

이 문제에서 확인한 점
평균 시간 복잡도 분석은 단순한 실행 경로 추적이 아니라
입력 분포에 대한 가정을 명확히 하는 것에서 시작된다.
같은 결과라도 역전 수 관점과 이동 거리 관점처럼
서로 다른 분석 틀로 동일한 결론에 도달할 수 있음을 확인했다.

대표 문제 **3. k-sorted** 배열 정렬

문제 정의
길이 n의 배열 A가 주어진다.
각 원소는 정렬 후 자신의 최종 위치로부터 최대 k만큼만 떨어져 있다(k-sorted).
이 제약을 이용해 전체 정렬보다 효율적인 알고리즘을 설계하는 것이 목표다.

핵심 관찰
모든 원소가 최대 k칸만 벗어나 있으므로,
각 위치에서의 정답 후보는 현재 위치를 기준으로 한 좁은 범위 안에만 존재한다.
따라서 전역적인 비교를 수행할 필요가 없다.

설계 1. 최소 힙 기반 정렬
크기 $k+1$의 최소 힙을 유지한다.
처음 $k+1$개의 원소를 힙에 넣고, 이후 배열을 순차적으로 읽으면서

힙의 최솟값을 하나씩 꺼내 정렬 결과에 배치한다.
새 원소를 힙에 추가하면서 이 과정을 반복한다.

이 방식은 항상 현재 위치에 올 수 있는 후보 중 최솟값을 선택한다는 점에서
문제의 제약을 정확히 반영한다.

시간 복잡도는 O(n log k),
공간 복잡도는 O(k)이다.

설계 2. 제한된 삽입 정렬
삽입 정렬을 그대로 사용하되,
각 원소의 이동 범위를 최대 k로 제한한다.
즉, 원소는 자신의 위치에서 최대 k칸까지만 왼쪽으로 이동한다.

이 경우 각 원소당 이동 비용은 O(k)이므로
전체 시간 복잡도는 O(nk)이다.
k가 충분히 작을 때는 단순하고 구현이 쉬운 장점이 있다.

두 해법의 비교
힙 기반 방식은 k가 커져도 안정적으로 O(n log k)를 유지한다.
제한된 삽입 정렬은 k가 매우 작을 때 상수 계수가 작아 실용적이다.
입력 제약의 크기에 따라 알고리즘 선택 기준이 달라진다.

복잡도 정리
힙 기반 방식: 시간 O(n log k), 공간 O(k)
제한 삽입 정렬: 시간 O(nk), 공간 O(1)

이 문제에서 확인한 점
입력에 추가 제약이 주어지면
기존의 일반적인 최적 알고리즘(O(n log n))이 항상 최선은 아니다.
문제 정의에서 제약을 정확히 활용하면
더 단순하거나 더 빠른 알고리즘을 선택할 수 있다.

대표 문제 **4.** 히스토그램에서 가장 넓은 직사각형

문제 정의
너비가 모두 1인 직사각형 막대들이 일렬로 주어지고, 각 막대의 높이는 서로 다르다.
이 히스토그램 안에서 만들 수 있는 직사각형들 중 넓이가 최대인 값을 구하는 것이 목표다.
직사각형은 연속한 막대 구간을 선택하고, 그 구간의 최소 높이를 높이로 갖는다.

핵심 관찰
최대 직사각형은 세 가지 경우 중 하나에 속한다.
왼쪽 절반에 완전히 포함되거나, 오른쪽 절반에 완전히 포함되거나, 가운데 경계를 가로지른다.
따라서 분할정복으로 문제를 나눌 수 있고, 관건은 “가운데를 가로지르는 최대 직사각형”을
효율적으로 계산하는 것이다.

분할정복 설계
구간 [l, r]에 대해 mid를 기준으로 [l, mid], [mid+1, r]로 분할한다.

각 절반에서의 최대 넓이는 재귀로 구한다.
그리고 가운데를 가로지르는 최대 넓이를 추가로 계산한 뒤, 세 값의 최댓값을 답으로 한다.

가운데를 가로지르는 최대 직사각형 계산 아이디어
초기 구간을 (i = mid, j = mid+1)로 두고, 현재 구간의 높이를 min(A[i], A[j])로 둔다.
현재 넓이는 (j - i + 1) * height이다.
이후 구간을 양쪽으로 넓히되, 확장할 때마다 height는 현재 구간의 최소 높이로 갱신된다.

효율을 위해 확장 방향은 다음 원칙으로 잡는다.
다음에 포함될 후보 막대 중 더 높은 쪽으로 먼저 확장하면, 최소 높이가 급격히 떨어지는 것을 늦출 수 있어 좋은 후보들을 빠르게 탐색할 수 있다.
즉, i-1과 j+1 중 존재하는 쪽을 보면서, 더 높은 막대 쪽으로 확장하고 height를 갱신하며 넓이를 업데이트한다.
이 과정을 i가 l에 도달하고 j가 r에 도달할 때까지 진행한다.

복잡도 분석
가운데 가로지르는 계산은 포인터가 양쪽으로 최대 (r-l+1)번만 이동하므로 O(n)이다.
재귀식은 T(n) = 2T(n/2) + O(n)이고, 따라서 전체 시간 복잡도는 O(n log n)이다.
추가 공간은 재귀 호출 스택 정도로 O(log n)이다.

이 문제에서 확인한 점
분할정복에서 핵심은 분할 자체가 아니라, 경계를 가로지르는 경우를 어떻게 “선형 시간”에 정복하느냐에 있다.
문제를 세 경우로 정확히 분해한 뒤, 가장 까다로운 경우(경계 포함)를 효율적으로 처리하면 전체 구조가 깔끔하게 닫힌다.

대표 문제 **5.** 편집 거리**(Edit Distance)**와 비용 가중치 확장

문제 정의
두 문자열 S, T가 주어진다.
삽입, 삭제, 대체 연산을 이용해 S를 T로 바꾸는 최소 비용을 계산한다.
기본 버전은 세 연산의 비용이 모두 1이라고 가정한다.
확장 버전은 삽입/삭제/대체의 비용이 서로 다를 수 있다고 가정하고, 그에 맞게 알고리즘을 수정한다.

핵심 관찰
문자열 변환은 prefix(앞부분)끼리의 변환으로 쪼개면 부분문제가 성립한다.
S의 앞 i글자를 T의 앞 j글자로 바꾸는 최소 비용을 D[i][j]로 두면,
마지막 연산(삽입/삭제/대체)만 결정하면 이전 상태로 항상 돌아갈 수 있다.

기본 DP 설계(비용이 모두 1인 경우)
D[i][j]는 다음 세 경우의 최소값으로 갱신된다.

1) 삭제: S[i]를 지운다
D[i][j] = D[i-1][j] + 1

2) 삽입: T[j]를 삽입한다
D[i][j] = D[i][j-1] + 1

3) 대체/일치: S[i]를 T[j]로 맞춘다
S[i] == T[j]면 비용 0, 다르면 비용 1
D[i][j] = D[i-1][j-1] + (S[i] == T[j] ? 0 : 1)

초기 조건은
D[0][j] = j (삽입만으로 길이 j 만들기)
D[i][0] = i (삭제만으로 길이 i 지우기)
이다.

확장: 연산 비용이 서로 다른 경우
삽입 비용 ci, 삭제 비용 cd, 대체 비용 cs로 둔다.
점화식에서 '+1'만 각 비용으로 바꾸면 된다.

삭제: D[i-1][j] + cd
삽입: D[i][j-1] + ci
대체/일치: D[i-1][j-1] + (S[i] == T[j] ? 0 : cs)

이 수정은 DP 구조를 바꾸지 않고, 문제 정의(비용 모델)의 변화를 그대로 반영한다.

복잡도 정리
DP 테이블 크기는 (|S|+1) x (|T|+1)이고, 각 칸은 O(1)에 계산된다.
따라서 시간 복잡도는 O(|S||T|), 공간 복잡도도 O(|S||T|)이다.
(추적 정보까지 저장하면 최소 연산 수열도 복원 가능하다.)

이 문제에서 확인한 점
DP는 "점화식 자체"보다 상태 정의가 먼저다.
문제를 prefix 변환 비용으로 정확히 정의하면,
비용 모델이 바뀌어도 점화식은 자연스럽게 확장된다.
즉, 문제 정의가 바뀌면 알고리즘은 같은 뼈대 위에서 수정되어야 한다는 점을 보여주는 사례다.

개념 범위 요약
정렬 알고리즘
비교 기반 정렬의 최악/평균/최선 복잡도 차이를 입력 상태 관점에서 이해한다.
입력 제약(k-sorted 등)이 주어질 경우 O(n log n) 정렬이 항상 최선은 아니다.

점근적 복잡도 표기
O, Θ, Ω 표기는 정확한 수행 시간 대신 성장률을 비교하기 위한 도구다.
입력크기와 가정이 명시되지 않은 복잡도 표기는 의미가 없다.

그리디 알고리즘
지역 최적 선택이 전체 최적으로 이어지기 위해 필요한 조건을 먼저 확인한다.
교환 논증이나 정렬 기준의 타당성이 설계의 핵심이다.
분할정복
문제를 나누는 것보다 경계를 포함하는 경우를 효율적으로 처리하는 것이 중요하다.
재귀식보다 병합 단계의 비용이 전체 복잡도를 좌우한다.

동적 계획법
점화식보다 상태 정의가 먼저다.
문제 정의(비용 모델, 허용 연산)가 바뀌어도 같은 상태 구조 위에서 확장된다.

백트래킹 / 분기 한정
해 공간 전체를 탐색하되, 제약이나 상한을 이용해 불필요한 분기를 제거한다.
문제 구조에 따라 탐색 순서와 가지치기 기준이 성능을 결정한다.

NP-완전 / 근사 알고리즘
정확해를 구하기 어려운 문제는 근사 비율과 가정이 알고리즘의 핵심이다.
문제의 난이도 분류가 설계 목표 자체를 바꾼다.