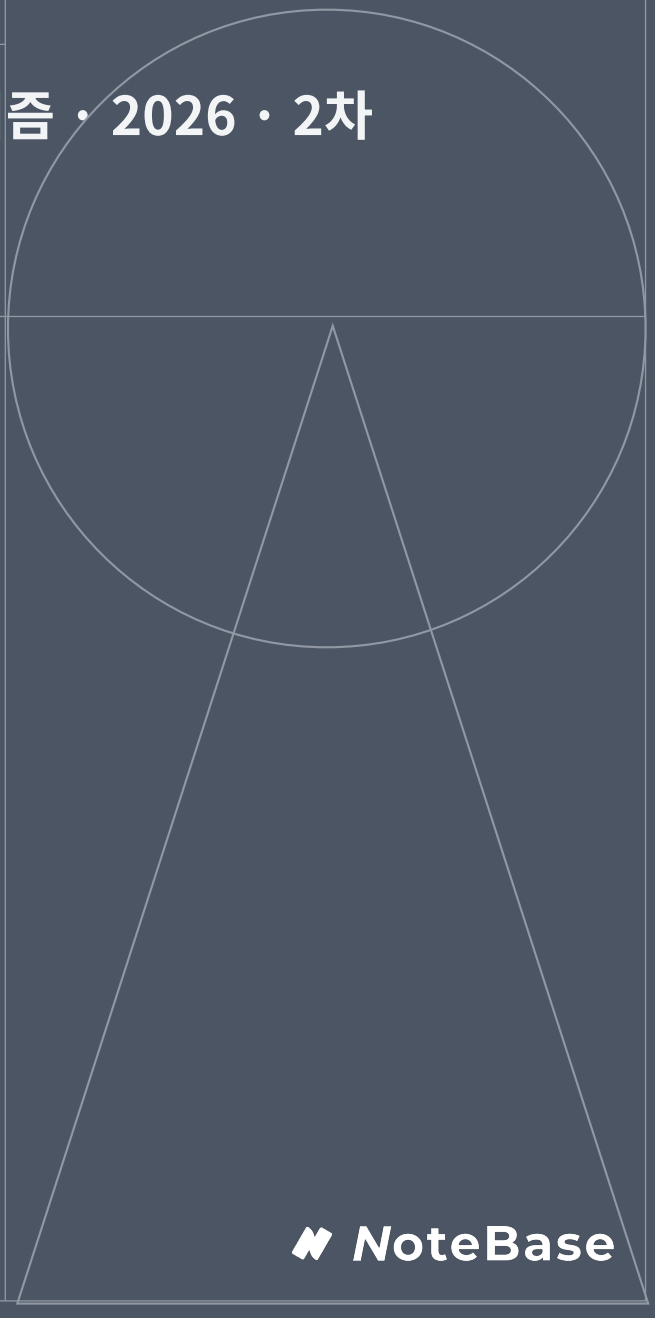
기출OX 해설집

공무원/국가직/7급 · 알고리즘 · 2026 · 2차



NoteBase

기출OX 해설집

공무원/국가직/7급 · 알고리즘 · 2026 · 2차

공무원/국가직/7급 · 알고리즘 · 2026 · 2차

문 1. 다음 중 레드-블랙 트리(red-black tree)의 특성으로 옳은 것만을 모두 고르면?

ㄱ. 모든 노드는 레드 또는 블랙이다.
ㄴ. 루트 노드는 레드이다.
ㄷ. 레드 노드의 자식 노드는 항상 블랙이다.
ㄹ. 모든 리프 노드(NIL)는 레드이다.

① ㄱ, ㄴ
② ㄱ, ㄷ
③ ㄴ, ㄷ
④ ㄷ, ㄹ

정답 2번

간단한 해설

레드-블랙 트리에서는 **루트와 NIL 리프가 모두 블랙**이다. 또한 **레드 노드의 자식은 블랙**이어야 하므로 부모와 자식이 연속해서 레드일 수 없다.

NoteBase

보기별 해설

ㄱ. O: 각 노드는 레드 또는 블랙 중 **하나의 색**을 갖는다. 색은 노드에 저장된 키의 값과 별도로 관리하는 정보이다.

ㄴ. X: 트리의 최상위 노드인 루트는 **블랙**이어야 한다. 따라서 '루트 노드는 레드이다'를 '루트 노드는 블랙이다'로 고친다.

ㄷ. O: 레드 노드의 자식은 **모두 블랙**이어야 한다. 즉, 부모와 자식이 모두 레드인 연결은 허용되지 않는다. 블랙 노드의 자식까지 항상 블랙이어야 한다는 뜻은 아니다.

ㄹ. X: NIL 리프는 실제 키를 저장하는 노드가 아니라 자식이 없는 위치를 나타내는 노드이며, **항상 블랙**으로 취급한다. 따라서 '모든 리프 노드(NIL)는 레드이다'를 '모든 리프 노드(NIL)는 블랙이다'로 고친다.

옳은 보기는 ㄱ, ㄷ이며, 이 조합은 ②에 해당한다.

NoteBase

문 2. 동적 계획법(dynamic programming)에 대한 설명으로 옳지 않은 것은?

① 부분 문제들 사이에는 의존적 관계가 없다.

② 동일한 부분 문제들을 다시 풀지 않도록 하기 위해 부분 문제들의 해를 저장한다.

③ 주어진 문제를 작은 문제로 나누어 해결하고 이를 이용하여 원래 문제의 해를 도출한다.

④ 이 기법을 적용할 수 있으려면 최적성의 원리(principle of optimality)가 성립하여야 한다.

정답 1번

간단한 해설

①의 '의존적 관계가 없다'는 설명이 잘못되었다. 동적 계획법은 **더 작은 부분 문제의 해를 이용**하며, 여러 계산에서 겹치는 부분 문제의 해를 저장해 재사용한다.

선지별 해설

① X: 동적 계획법에서는 한 부분 문제의 해를 구할 때 **더 작은 부분 문제의 해에 의존**할 수 있다. 예를 들어 피보나치 수열의 다섯 번째 값을 구하려면 네 번째와 세 번째 값이 필요하고, 네 번째 값을 구할 때도 세 번째 값이 필요하다. 이처럼 부분 문제가 서로 겹치고 그 결과를 이용하므로, 의존적 관계가 없다는 설명은 부적절하다.

② O: 한 번 구한 부분 문제의 해를 저장했다가 같은 부분 문제가 다시 필요할 때 **저장된 결과를 재사용**한다. 따라서 동일한 계산을 반복하는 낭비를 줄일 수 있다.

③ O: 원래 문제를 더 작은 부분 문제로 나누고, 그 해들을 결합하여 원래 문제의 해를 구한다. 이를 위해 어떤 작은 문제의 결과로 현재 문제를 해결할 수 있는지 관계를 정하고, 필요한 부분 문제의 결과를 이용한다.

④ O: 최적성의 원리는 **전체 문제의 최적해를 구성하는 부분해도 해당 부분 문제의 최적해**여야 한다는 뜻이다. 최적화 문제에 동적 계획법을 적용하려면 이러한 성질이 성립해야 작은 문제에서 구한 최적해를 이용해 전체 문제의 최적해를 도출할 수 있다.

NoteBase

문 3. 스택(stack)의 특징에 대한 설명으로 옳지 않은 것은?

① 나중에 들어온 데이터가 먼저 나가는 후입 선출 구조(LIFO, Last-In First-Out)로 입출력이 일어난다.

② 제일 먼저 입력된 데이터가 맨 아래에 쌓이고 가장 최근에 입력된 데이터가 가장 위에 쌓이는 구조로 되어 있다.

③ 함수 호출에서 복귀주소를 기억하는 데 스택을 사용하고, 함수는 수행이 끝나면 최근에 자신을 호출한 함수로 돌아간다.

④ push 연산은 스택에서 하나의 요소를 제거하는 연산으로 top이 가리키는 요소를 스택에서 꺼내서 외부로 건네주는 연산이다.

정답 4번

간단한 해설

④의 요소 제거 설명은 **pop 연산**에 해당한다. push는 스택의 맨 위에 새 요소를 삽입하는 연산이다.

선지별 해설

① O: 스택은 한쪽 끝에서 데이터를 넣고 꺼내므로 **나중에 넣은 데이터가 먼저 나오는 후입선출(LIFO)** 구조이다. 예를 들어 빈 스택에 A를 넣고 B를 넣으면, 꺼낼 때는 B가 먼저 나오고 그다음 A가 나온다.

② O: 스택에 새 데이터를 넣으면 기존 데이터 위에 쌓인다. 따라서 먼저 넣은 데이터는 아래에, 가장 최근에 넣은 데이터는 위에 놓이며, **top은 맨 위의 요소**를 가리킨다.

③ O: 함수가 다른 함수를 호출하면 호출한 곳으로 돌아갈 복귀주소를 스택에 보관한다. 예를 들어 A가 B를 호출하고 B가 C를 호출했다면, C는 B로 돌아가고 B는 A로 돌아간다. 이처럼 **나중에 호출된 함수부터 복귀**하는 순서가 스택의 후입선출 구조와 맞는다.

④ X: push는 스택의 맨 위에 새 요소를 삽입하는 연산이다. top이 가리키는 요소를 제거하여 외부로 건네주는 것은 **pop 연산**이다. 선지의 'push 연산'을 'pop 연산'으로 고쳐야 한다.

NoteBase

문 4. 다음 중 안정적 정렬(stable sort) 알고리즘에 해당하는 것만을 모두 고르면?

ㄱ. 버블 정렬(bubble sort)
ㄴ. 힙 정렬(heap sort)
ㄷ. 삽입 정렬(insertion sort)
ㄹ. 계수 정렬(counting sort)

① ㄴ, ㄷ
② ㄴ, ㄹ
③ ㄱ, ㄴ, ㄷ
④ ㄱ, ㄷ, ㄹ

정답 4번

간단한 해설

힙 정렬은 루트와 배열 끝의 원소를 교환하는 과정에서 **키가 같은 원소의 상대적 순서를 바꿀 수 있다**. 안정적 정렬은 이 순서를 정렬 전과 같게 유지해야 하므로 힙 정렬은 제외된다.

NoteBase

보기별 해설

안정적 정렬은 키가 같은 원소들의 상대적 순서를 보존하는 정렬이다. 예를 들어 키가 2인 원소 A와 B가 입력에서 A, B 순서였다면, 정렬 후에도 A가 B보다 앞에 있어야 한다.

ㄱ. 버블 정렬 — 안정적 정렬: 인접한 두 원소를 비교하여 앞 원소의 키가 더 클 때만 교환한다. **키가 같으면 교환하지 않으므로** 같은 키를 가진 원소들이 서로를 앞질러 순서가 뒤집히지 않는다.

ㄴ. 힙 정렬 — 비안정적 정렬: 최대 힙에서 루트의 최댓값을 배열 끝으로 옮기며 정렬한다. 이때 **떨어진 원소끼리 교환**하므로 같은 키의 입력 순서를 보존하지 못할 수 있다. 예를 들어 최대 힙인 [2A, 2B, 1]에서 루트와 끝을 교환하면 [1, 2B, 2A]가 되고, 남은 부분을 힙으로 복구하면 [2B, 1, 2A]가 된다. 다음 교환으로 [1, 2B, 2A]가 완성되어, 같은 키 2를 가진 A와 B의 순서가 뒤집힌다. 따라서 안정적 정렬에 포함되지 않는다.

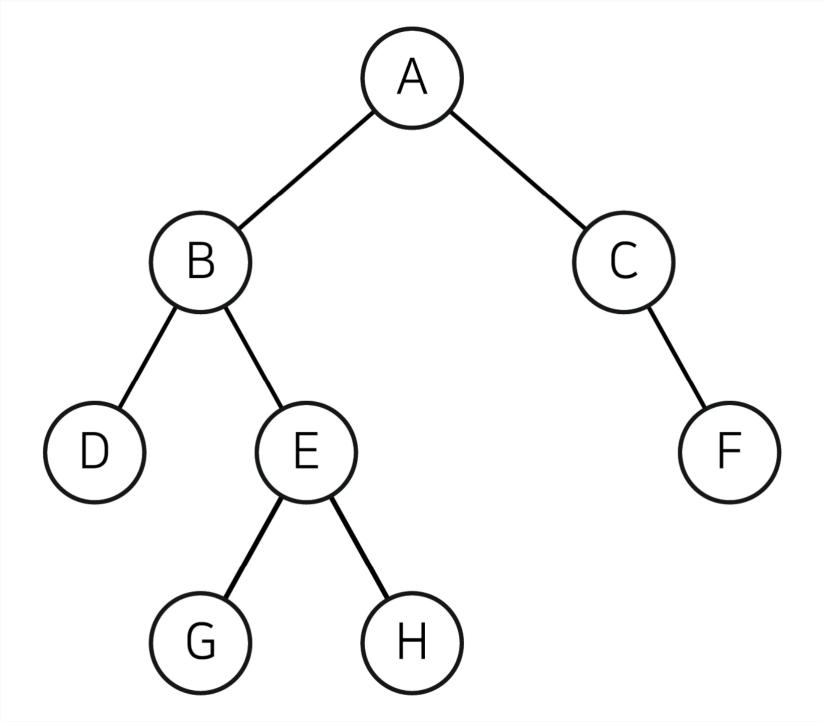
ㄷ. 삽입 정렬 — 안정적 정렬: 현재 원소를 앞쪽의 정렬된 부분에 삽입할 때, 현재 키보다 큰 원소만 오른쪽으로 이동시킨다. **같은 키의 기존 원소 뒤에 삽입**하므로 먼저 입력된 원소가 앞에 남는다.

ㄹ. 계수 정렬 — 안정적 정렬: 표준 계수 정렬은 키별 개수의 누적합으로 출력 위치를 구하고, 입력을 뒤에서부터 읽어 원소를 출력 배열에 배치한다. 같은 키의 원소를 하나 배치할 때마다 해당 키의 다음 출력 위치를 한 칸 앞당기므로, **나중에 입력된 원소가 뒤쪽에 배치**되어 기존 순서가 유지된다. 단순히 키별 개수만 세어 키 값을 다시 나열하는 방식과 달리, 원래 원소를 출력 배열로 옮기는 표준 구현을 기준으로 한다.

안정적 정렬에 해당하는 보기는 ㄱ, ㄷ, ㄹ이며, 이 조합은 ④에 제시되어 있다.

NoteBase

문 5. 다음 이진 트리(binary tree)의 단말 노드(leaf node) 개수와 간선(edge) 개수는?



① 단말 노드 개수: 3, 간선 개수: 7

② 단말 노드 개수: 3, 간선 개수: 8

③ 단말 노드 개수: 4, 간선 개수: 7

④ 단말 노드 개수: 4, 간선 개수: 8

정답 3번

간단한 해설

단말 노드는 **자식이 없는 노드**이므로 D·F·G·H의 4개이다. 전체 노드가 8개인 트리에서 간선은 루트를 제외한 각 노드에 하나씩 대응하므로 7개이다.

NoteBase

계산 과정

단말 노드 개수: 단말 노드는 **자식이 하나도 없는 노드**이다. 그림에서 D·F·G·H는 자식이 없으므로 단말 노드는 4개이다. C는 자식 F가 있고 E는 자식 G·H가 있으므로 단말 노드가 아니다. 단말 노드는 모두 같은 높이에 있을 필요가 없다.

간선 개수: 간선은 부모와 자식을 연결하는 선이다. 전체 노드는 A부터 H까지 8개이며, **루트 A를 제외한 각 노드는 부모와 연결되는 간선을 하나씩** 가진다. 따라서 간선 개수는 8 - 1 = 7개이다. 그림에서도 A–B, A–C, B–D, B–E, C–F, E–G, E–H의 7개를 확인할 수 있다.

단말 노드 4개와 간선 7개에 해당하는 선택지는 ③이다.

NoteBase

문 6. 다음 설명에 해당하는 알고리즘 기법은?

◯ 깊이 우선 탐색(DFS, Depth First Search)이 기본 골격이다.
◯ 해를 찾는 도중에 해가 아니면 되돌아가서 다시 해를 탐색한다.
◯ 최적화(optimization) 문제와 결정(decision) 문제를 해결할 수 있다.

① 백 트래킹(back tracking)
② k-NN(k-Nearest Neighbors)
③ 그리디(greedy) 알고리즘
④ 분할 정복(divide-and-conquer) 알고리즘

정답 1번

간단한 해설

조건을 만족하는 해가 될 수 없으면 이전 선택으로 되돌아가 다른 후보를 탐색하는 것이 백 트래킹의 핵심이다. 깊이 우선 탐색을 바탕으로 가능성 없는 탐색 경로를 일찍 제외한다.

공통 해설

백 트래킹은 해를 구성하는 선택을 하나씩 쌓아 가다가, 현재까지의 선택으로는 조건을 만족하는 해를 만들 수 없다고 판단되면 그 경로의 탐색을 중단하는 기법이다. 이때 **직전 선택을 취소하고 다른 후보를 선택**한다. 한 경로를 먼저 깊게 탐색한 뒤 되돌아온다는 점에서 깊이 우선 탐색이 기본 골격이다. 예를 들어 퀸을 서로 공격하지 않도록 놓는 문제에서 새 퀸이 앞서 놓은 퀸과 같은 열에 놓이면, 그 배치를 더 확장하지 않고 새 퀸을 거두어 다른 위치를 시도한다.

결정 문제에서는 조건을 만족하는 해가 존재하는지 탐색한다. 최적화 문제에서는 가능한 해들의 값을 비교하여 현재까지의 최선의 해를 갱신하며, 더 나은 해를 만들 수 없다고 판단되는 경로도 제외할 수 있다. 따라서 지문의 되돌아가기와 재탐색은 백 트래킹을 가리킨다.

NoteBase

문 7. 다음 중 문자열 매칭 알고리즘에 해당하는 것만을 모두 고르면?

ㄱ. 벨만 포드(Bellman Ford)
ㄴ. 보이어-무어(Boyer-Moore)
ㄷ. 다익스트라(Dijkstra)
ㄹ. KMP(Knuth, Morris, Pratt)

① ㄱ, ㄴ
② ㄱ, ㄷ
③ ㄴ, ㄷ
④ ㄴ, ㄹ

정답 4번

간단한 해설

보이어-무어와 KMP는 긴 문자열에서 **특정 패턴이 나타나는 위치를 찾는** 알고리즘이다. 보이어-무어는 패턴의 오른쪽부터 비교하고, KMP는 이미 일치한 부분의 정보를 활용해 불필요한 재비교를 줄인다.

NoteBase

보기별 해설

문자열 매칭은 본문 문자열에서 찾으려는 문자열인 패턴이 나타나는 위치를 찾는 작업이다. 보기의 알고리즘을 문자열 검색과 그래프의 최단 경로 탐색으로 구분한다.

ㄱ. 벨만 포드: **그래프의 최단 경로**를 구하는 알고리즘이다. 시작 정점에서 다른 정점까지의 거리 추정값을 간선을 따라 반복해서 갱신하므로, 문자열에서 패턴을 찾는 알고리즘에 해당하지 않는다.

ㄴ. 보이어-무어: 문자열 매칭 알고리즘이다. 본문과 패턴을 맞춘 뒤 **패턴의 오른쪽 문자부터** 비교한다. 불일치가 발생하면 문자와 패턴의 정보를 이용해 다음 비교를 위한 패턴의 이동 폭을 결정한다.

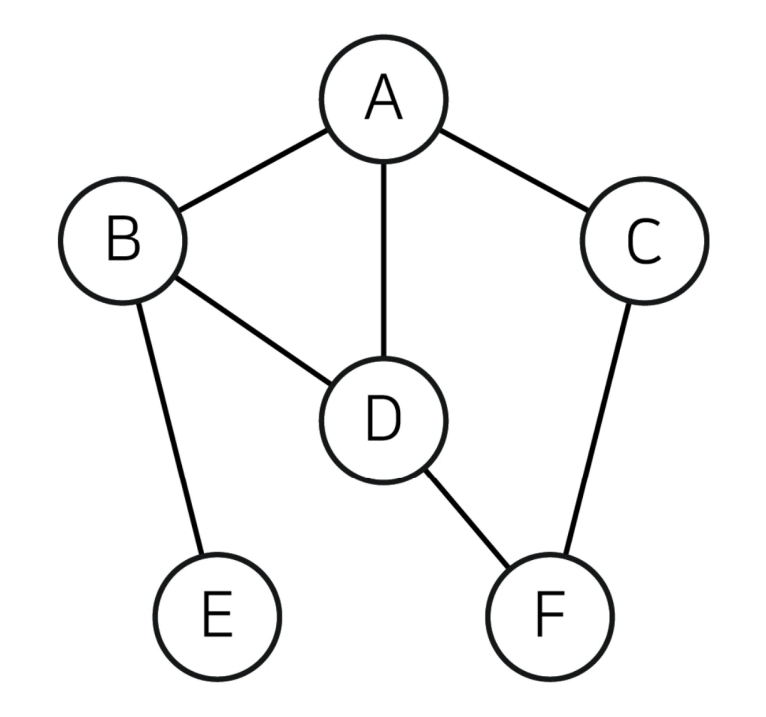
ㄷ. 다익스트라: **간선 가중치가 음수가 아닌 그래프**에서 시작 정점으로부터의 최단 경로를 구하는 알고리즘이다. 아직 최단 거리가 확정되지 않은 정점 중 거리 추정값이 가장 작은 정점을 선택하고 인접 정점의 거리를 갱신하므로, 문자열 매칭과는 목적이 다르다.

ㄹ. KMP: 문자열 매칭 알고리즘이다. 패턴의 **접두사와 접미사가 일치하는 정보**를 미리 구해 둔다. 비교 중 불일치가 발생하면 이 정보를 이용해 패턴의 다음 비교 위치를 정하므로, 본문에서 이미 확인한 문자를 불필요하게 다시 비교하는 일을 줄인다.

문자열 매칭 알고리즘에 해당하는 것은 ㄴ과 ㄹ이며, 이는 ④의 조합과 일치한다.

NoteBase

문 8. 다음 그래프에서 정점 A부터 깊이 우선 탐색을 수행할 때 가능한 정점들의 방문 순서로 옳은 것은?



① A → B → E → F → D → C

② A → B → D → C → E → F

③ A → D → B → E → F → C

④ A → D → F → E → B → C

정답 3번

간단한 해설

깊이 우선 탐색은 **현재 정점에서 더 갈 곳이 없을 때 되돌아간다**. E에서는 B를 거쳐 D까지 되돌아온 뒤 F로 진행하며, 되돌아가는 정점은 새 방문 순서에 다시 기록하지 않는다.

NoteBase

올바른 순서

A → D → B → E → F → C

A: 시작 정점 A를 먼저 방문한다. 깊이 우선 탐색은 **미방문 인접 정점으로 계속 진행**하고, 더 진행할 수 없을 때 이전 정점으로 되돌아가는 방식이다. 인접 정점의 선택 우선순위가 주어지지 않았으므로 A에 연결된 B·C·D 중 D를 먼저 선택할 수 있다.

D: A에서 D로 이동한다. D의 인접 정점은 A·B·F이고 A는 이미 방문했으므로, 아직 방문하지 않은 B와 F 중 B를 먼저 선택할 수 있다.

B: D에서 B로 이동한다. B의 인접 정점은 A·D·E이다. A와 D는 이미 방문했으므로 **남은 미방문 인접 정점 E**로 진행한다.

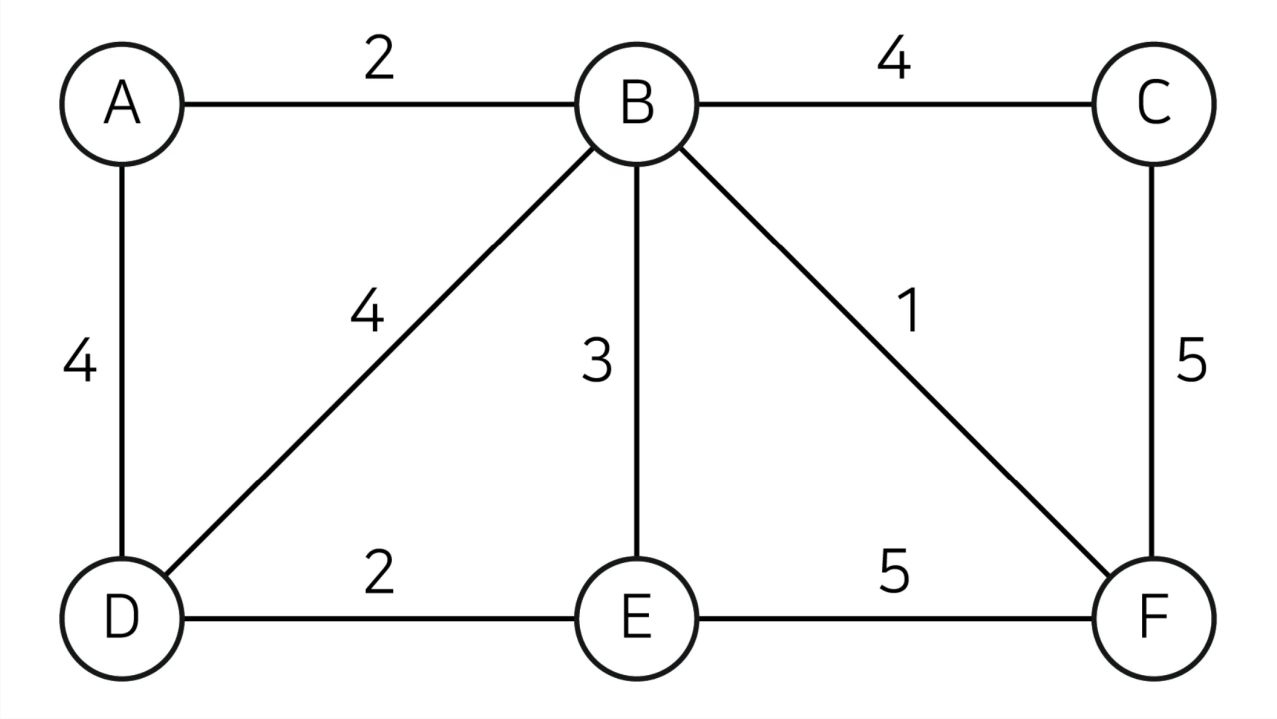
E: B에서 E로 이동한다. E에 연결된 정점은 이미 방문한 B뿐이므로 B로 되돌아간다. B에도 미방문 인접 정점이 없으므로 다시 D로 되돌아간다. 이때 B와 D는 **처음 방문하는 정점이 아니므로** 방문 순서에 다시 기록하지 않는다.

F: 되돌아온 D에는 아직 방문하지 않은 인접 정점 F가 남아 있으므로 F로 이동한다. 방문 기록에서 E 다음에 F가 나오는 것은 E에서 F로 직접 이동해서가 아니라, **되돌아온 D에서 탐색을 이어 갔기 때문**이다.

C: F의 인접 정점은 C와 D이다. D는 이미 방문했으므로 C로 진행한다. C에 연결된 A와 F도 이미 방문했으며, 이제 여섯 정점을 모두 방문했으므로 새로 기록할 정점은 없다.

NoteBase

문 9. 다음 그래프에 대해 정점 A를 시작으로 프림(Prim) 알고리즘을 수행했을 때, 생성된 최소 신장 트리의 간선 개수는?



① 4개

② 5개

③ 6개

④ 7개

정답 2번

간단한 해설

최소 신장 트리는 모든 정점을 사이클 없이 연결하므로 **간선 수가 정점 수보다 하나 적다**. 그래프의 정점은 A~F의 6개이므로 간선은 5개이다.

NoteBase

계산 과정

그림의 정점은 A, B, C, D, E, F로 총 6개이며, 모든 정점이 서로 경로로 연결되어 있다. 구하려는 값은 최소 신장 트리의 간선 개수이다.

최소 신장 트리는 모든 정점을 사이클 없이 연결하는 신장 트리 중 간선 가중치의 합이 최소인 것이다. 프림 알고리즘은 A 하나에서 시작하여, 현재 트리와 바깥 정점을 잇는 간선 중 가중치가 가장 작은 간선을 선택한다. 이때 **간선 하나를 추가할 때마다 새 정점 하나가 포함**된다. 따라서 정점 수가 n인 연결 그래프에서는 시작 정점을 제외한 나머지 정점을 포함시키는 데 간선이 $n-1$개 필요하다.

이 그래프에서는 시작 정점 A를 제외한 5개 정점을 추가하므로 간선 수는 6 - 1 = 5개이다. 가중치는 어떤 간선을 고를지 결정하지만 완성된 신장 트리의 간선 수를 바꾸지는 않는다. 계산 결과는 ②의 5개에 해당한다.

NoteBase

문 10. 다음 설명에 해당하는 정렬 알고리즘은?

◯ 배열을 정렬된 앞부분과 정렬되지 않은 뒷부분으로 나누어 처리한다.
◯ 최선의 경우 시간 복잡도는 $O(n)$이며 최악의 경우 시간 복잡도는 $O(n^2)$이다.

① 삽입 정렬
② 버블 정렬
③ 병합(merge) 정렬
④ 선택(selection) 정렬

정답 1번

간단한 해설

삽입 정렬은 **정렬된 앞부분에 다음 원소를 끼워 넣어** 정렬 구간을 늘린다. 이미 정렬되어 있으면 원소를 옮길 필요가 없어 선형 시간이 걸리지만, 역순이면 반복적인 비교와 이동 때문에 이차 시간이 걸린다.

핵심 풀이

삽입 정렬은 첫 원소를 정렬된 구간으로 보고, 뒷부분의 첫 원소를 꺼내 **앞부분의 알맞은 위치에 삽입**한다. 이 과정을 반복하면 정렬된 앞부분이 한 원소씩 늘어난다. 예를 들어 오름차순 정렬에서 앞부분이 [2, 5], 뒷부분이 [3, 1]이면 5를 오른쪽으로 옮기고 3을 삽입하여 앞부분은 [2, 3, 5], 뒷부분은 [1]이 된다.

원소 수를 n이라 하면, 이미 오름차순인 배열에서는 각 원소가 바로 앞 원소보다 작지 않은지만 확인하면 되므로 최선의 시간 복잡도는 $O(n)$이다. 반대로 서로 다른 원소가 역순으로 놓여 있으면 새 원소를 삽입할 때 정렬된 앞부분의 원소들을 모두 비교하고 옮겨야 한다. 앞부분이 길어질수록 이동 횟수도 늘어나므로 최악의 시간 복잡도는 $O(n^2)$이다.

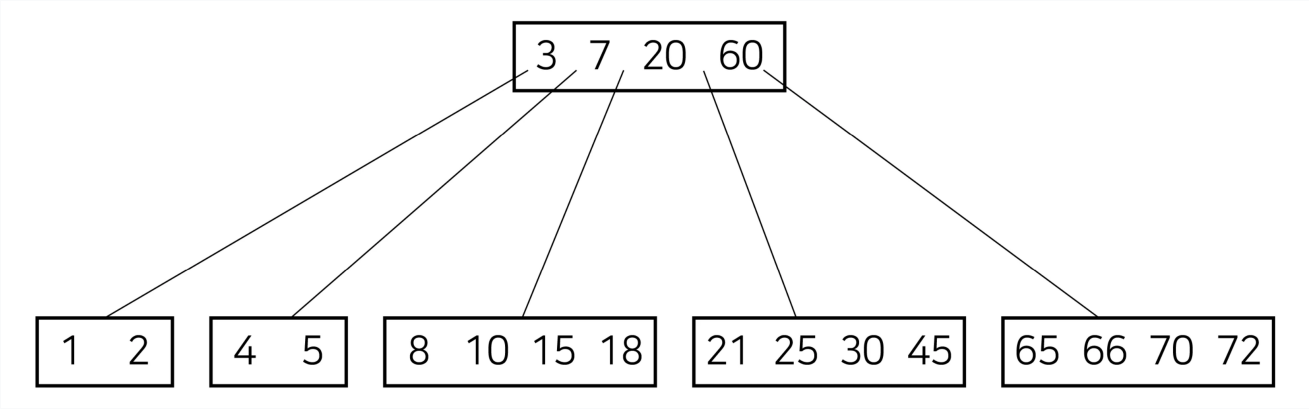
두 조건을 **함께 만족하는지** 구분해야 한다. 선택 정렬도 정렬된 앞부분을 늘리지만, 매번 미정렬 부분 전체에서 최솟값을 찾아야 하므로 이미 정렬되어 있어도 비교 횟수는 $O(n^2)$이다.

일반적인 오름차순 버블 정렬은 인접 원소를 비교·교환하여 큰 원소를 뒤로 보내므로 정렬된 뒷부분이 늘어난다. 교환이 없을 때 종료하도록 구현하면 최선에는 $O(n)$이 걸릴 수 있지만, 지문의 앞부분을 확장하는 처리 방식과 다르다.

병합 정렬은 배열을 나누어 각각 정렬한 뒤 두 정렬 구간을 합치는 방식이므로, 앞부분에 다음 원소를 삽입하는 처리와 구별된다.

NoteBase

문 11. 다음 B-트리에 “50”을 삽입할 때의 설명으로 옳지 않은 것은? (단, 각 노드는 최대 4개의 키를 가질 수 있다)



① 50이 루트 노드가 된다.

② 모든 리프 노드의 레벨은 동일하다.

③ 50을 삽입하면 오버플로우가 발생한다.

④ 50을 삽입할 때 분할(split) 연산이 필요하다.

정답 1번

간단한 해설

50을 삽입한 노드를 분할할 때 부모로 올라가는 키는 **중앙값 30**이다. 이어서 기존 루트도 분할되므로 새 루트의 키는 **50이 아니라 20**이 된다.

NoteBase

선지별 해설

B-트리는 각 노드의 키를 정렬하여 저장하고, 모든 리프 노드를 같은 깊이에 유지한다. 이 문제에서는 한 노드의 키가 4개를 넘으면 중앙 키를 부모로 올리고 나머지 키를 두 노드로 나누어 오버플로우를 해소한다.

① X: 50은 20보다 크고 60보다 작으므로 리프 노드 [21, 25, 30, 45]에 삽입된다. 삽입 직후 [21, 25, 30, 45, 50]을 분할하면 중앙 키 30이 부모로 올라가고, [21, 25]와 [45, 50]이 남는다. 기존 루트는 [3, 7, 20, 30, 60]이 되어 다시 분할된다. 이때 중앙 키 20이 새 루트로 올라가며, 그 자식 노드는 [3, 7]과 [30, 60]이 된다. 따라서 **새 루트의 키는 20**이고 50은 리프 노드에 남는다.

② O: **모든 리프 노드의 깊이가 같다**는 것은 B-트리의 성질이다. 리프 노드를 둘로 분할해도 두 노드는 원래 리프와 같은 깊이에 놓인다. 이어서 루트가 분할되면 새 루트가 한 단계 위에 생겨 모든 리프의 깊이가 함께 1씩 증가하므로, 삽입 후에도 리프들의 레벨은 동일하다.

③ O: 삽입 대상인 [21, 25, 30, 45]에는 이미 키가 4개 있다. 여기에 50을 넣으면 키가 5개가 되어 **최대 허용 개수 4개를 초과**하므로 오버플로우가 발생한다.

④ O: 넘친 키를 처리하려면 **노드를 분할하고 중앙 키를 부모로 올려야 한다**. 먼저 리프를 분할하여 30을 올린다. 기존 루트도 이미 키가 4개였으므로 30을 받은 뒤에는 키가 5개가 되어, 루트에 대해서도 분할 연산이 필요하다.

NoteBase

문 12. 다음 C언어 함수를 이용하여 rec(5)를 수행한 결괏값은?

```
int rec(int n) {
   if (n <= 0)
      return 0;
   else if (n <= 3)
      return n;
   else
      return rec(n-4) + rec(n-3) + rec(n-2) + rec(n-1);
}
```

① 9
② 10
③ 12
④ 15

정답 3번

간단한 해설

rec(5)는 rec(1), rec(2), rec(3), rec(4)의 반환값을 더한다. **rec(4)의 반환값이 6**이므로 rec(5)는 1 + 2 + 3 + 6 = 12를 반환한다.

NoteBase

계산 과정

함수의 매개변수 n은 각 호출에 전달된 정수이다. n이 0 이하이면 0을 반환하고, 앞 조건에 해당하지 않으면서 n이 3 이하이면 n을 그대로 반환한다. 따라서 rec(0), rec(1), rec(2), rec(3)의 반환값은 각각 0, 1, 2, 3이다. 이 호출들은 **재귀 호출 없이 종료**된다.

rec(4): n이 3보다 크므로 마지막 else에 따라 rec(0), rec(1), rec(2), rec(3)의 반환값을 더한다. 따라서 $\mathrm{rec}(4) = 0 + 1 + 2 + 3 = 6$이다. 이 6은 rec(5)를 계산할 때 필요한 **하위 호출의 반환값**이다.

rec(5): 마지막 else의 네 인수 $n-4$, $n-3$, $n-2$, $n-1$은 각각 1, 2, 3, 4이다. 따라서 rec(1), rec(2), rec(3), rec(4)의 반환값을 더하여 $\mathrm{rec}(5) = 1 + 2 + 3 + 6 = 12$를 얻는다. 여기서 더하는 것은 인수 1, 2, 3, 4 자체가 아니라 **각 호출이 돌려주는 값**이다.

최종 반환값 12는 ③의 값과 일치한다.

NoteBase

문 13. 배열 A = [5, 2, 4, 6, 1, 3]를 삽입 정렬로 오름차순 정렬할 때, 전체 과정에서 수행되는 key 값의 총 비교 횟수는? (단, 알고리즘은 다음과 같이 동작한다고 가정하며, 배열의 인덱스는 1부터 시작하고, n 값은 배열 A의 원소의 개수이다)

```
for i ← 2 to n do
    key ← A[i]
    j ← i - 1
    while j ≥ 1 and A[j] > key do
        A[j+1] ← A[j]
        j ← j - 1
    A[j+1] ← key
```

① 12

② 13

③ 14

④ 15

정답 1번

간단한 해설

key와 배열 원소의 비교는 **반복을 끝내는 거짓 비교도 포함**하되, j가 0이 되어 종료되면 추가하지 않는다. 각 삽입 단계의 비교 횟수는 1, 2, 1, 4, 4회이므로 총 12회이다.

NoteBase

계산 과정

비교 횟수는 while 조건의 A[j] > key를 실제로 평가한 횟수이다. j ≥ 1이라는 인덱스 범위 검사는 key 값의 비교에 포함하지 않는다. A[j] > key가 거짓이 되어 반복을 끝내는 비교는 포함하지만, **j가 0이면 key와 배열 원소를 더 비교하지 않는다**. 원소가 6개이므로 i = 2부터 6까지 차례로 삽입한다.

i = 2, key = 2: 앞의 원소 5와 비교한다. 5 > 2가 참이므로 5를 오른쪽으로 옮기고 j는 0이 된다. 비교는 1회이며, 삽입 후 배열은 [2, 5, 4, 6, 1, 3]이다.

i = 3, key = 4: 5 > 4는 참이므로 5를 옮긴다. 다음 2 > 4는 거짓이므로 멈춘다. **거짓인 마지막 비교까지 포함**하여 2회이며, 삽입 후 배열은 [2, 4, 5, 6, 1, 3]이다.

i = 4, key = 6: 5 > 6이 거짓이므로 원소를 옮기지 않고 멈춘다. 비교는 1회이며, 배열은 [2, 4, 5, 6, 1, 3]으로 유지된다.

i = 5, key = 1: 왼쪽의 6, 5, 4, 2와 차례로 비교한다. 모두 1보다 크므로 오른쪽으로 옮기며, 2를 옮긴 뒤 j가 0이 되어 종료된다. 비교는 4회이며, 삽입 후 배열은 [1, 2, 4, 5, 6, 3]이다.

i = 6, key = 3: 6, 5, 4와의 비교는 모두 참이므로 이들을 옮긴다. 이어 2 > 3이 거짓이 되어 멈춘다. 비교는 4회이며, 삽입 후 배열은 [1, 2, 3, 4, 5, 6]이다.

전체 key 값의 비교 횟수는 1 + 2 + 1 + 4 + 4 = 12회이며, ①의 12에 해당한다.

NoteBase

문 14. 위상 정렬(topological sort)에 대한 설명으로 옳은 것은?

① 깊이 우선 탐색 방식을 이용해서는 구현할 수 없다.

② 하나의 그래프에 한 개의 위상 정렬만 존재한다.

③ 사이클이 없는 모든 종류의 그래프에 적용할 수 있다.

④ 간선 $\langle i, j \rangle$를 가질 때 정점 i는 정점 j를 선행한다.

정답 4번

간단한 해설

위상 정렬은 방향 간선이 나타내는 **선행 관계를 지키도록** 정점을 나열하는 것이다. 간선 $\langle i, j \rangle$가 i에서 j로 향하면, 나열된 순서에서 i가 j보다 앞에 와야 한다.

선지별 해설

① X: 위상 정렬은 **깊이 우선 탐색으로 구현할 수 있다**. 방향 비순환 그래프에서 아직 방문하지 않은 인접 정점을 깊이 우선으로 탐색하고, 각 정점의 탐색이 끝나는 순서를 기록한다. 이 종료 순서를 뒤집으면 모든 방향 간선의 시작 정점이 끝 정점보다 앞에 오는 위상 정렬을 얻는다.

② X: 한 그래프에도 위상 정렬이 **여러 개 존재할 수 있다**. 예를 들어 간선이 A → C와 B → C뿐이면 A와 B는 모두 C보다 앞에 와야 하지만, A와 B 사이의 순서는 정해지지 않는다. 따라서 A → B → C와 B → A → C가 모두 가능한 위상 정렬이다. 이때 나열 순서의 화살표는 원래 그래프의 간선을 뜻하지 않는다.

③ X: 위상 정렬의 적용 대상은 사이클이 없는 모든 종류의 그래프가 아니라 **방향 비순환 그래프(DAG)**이다. 무방향 그래프는 간선에 따른 선행 방향이 정해져 있지 않다. 방향 그래프라도 사이클이 있으면 그 경로를 따라 선행 조건이 되돌아와 한 정점이 자기 자신보다 앞서야 하는 모순이 생기므로 위상 정렬이 불가능하다.

④ O: 방향 간선 $\langle i, j \rangle$는 i에서 j로 향하는 간선이다. 위상 정렬은 모든 방향 간선에 대해 **시작 정점을 끝 정점보다 앞에 배치**하므로 i가 j를 선행한다. 다만 i와 j가 반드시 서로 이웃한 위치에 놓여야 하는 것은 아니다.

NoteBase

문 15. 다음 파이썬 프로그램의 실행 결과는?

```
def search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target: return mid
        elif arr[mid] < target: low = mid + 1
        else: high = mid - 1
    return -1

if __name__ == "__main__":
    a = [2, 5, 8, 12, 16, 23, 38, 56, 91]
    print(search(a, 23))
```

① -1

② 4

③ 5

④ 6

정답 3번

간단한 해설

이 함수는 찾은 값 자체가 아니라 **0부터 세는 인덱스**를 반환한다. 이진 탐색으로 23을 찾았을 때 mid는 5이므로 출력값은 5이다.

NoteBase

계산 과정

search는 오름차순 배열에서 탐색 구간을 절반씩 줄이는 이진 탐색 함수이다. low와 high는 탐색 구간의 양 끝 인덱스이며, mid = (low + high) // 2로 가운데 인덱스를 구한다. //는 나눗셈의 몫을 구하는 연산이다. 배열 a의 원소는 9개이므로 처음에는 low = 0, high = 8이고, 찾을 값 target은 23이다.

첫 번째 반복: mid = (0 + 8) // 2 = 4이고 a[4] = 16이다. 16 < 23이므로 **인덱스 4 이하를 탐색 대상에서 제외**하고 low = mid + 1 = 5로 바꾼다. 이후 탐색 구간은 인덱스 5~8이다.

두 번째 반복: mid = (5 + 8) // 2 = 6이고 a[6] = 38이다. 38 > 23이므로 인덱스 6 이상을 제외하고 high = mid − 1 = 5로 바꾼다. 탐색 구간에는 인덱스 5만 남는다.

세 번째 반복: low = high = 5여도 **low <= high 조건을 만족**하므로 반복을 실행한다. mid = (5 + 5) // 2 = 5이고 a[5] = 23이므로 return mid에 따라 5를 반환하며 함수를 종료한다. 끝의 return −1은 탐색에 실패하여 반복문을 빠져나갔을 때만 실행된다.

print는 반환값 5를 출력한다. 23은 배열의 여섯 번째 원소이지만 인덱스는 5이며, 이 출력값은 ③의 5에 해당한다.

NoteBase

문 16. 다음은 어떤 문자열에 포함된 문자들의 빈도수를 나타낸 표이다. 허프만(Huffman) 코드를 적용하여 인코딩한 결과의 비트 수는?

문자	a	b	c	d	e	f
빈도수	7	20	10	5	50	6

① 98

② 202

③ 300

④ 331

정답 2번

간단한 해설

허프만 트리에서 두 노드를 합칠 때마다 그 아래 문자들의 코드 길이가 1비트씩 늘어난다. 따라서 **각 병합에서 얻은 빈도수의 합**이 전체 인코딩 길이이며, 이 표에서는 **202비트**이다.

NoteBase

계산 과정

표의 문자별 빈도수는 a가 7, b가 20, c가 10, d가 5, e가 50, f가 6이다. 전체 문자 수는 7 + 20 + 10 + 5 + 50 + 6 = 98개이며, 구할 값은 이 문자열을 허프만 코드로 표현하는 데 필요한 총 비트 수이다.

허프만 트리는 **현재 남은 노드 중 빈도수가 가장 작은 두 노드**를 골라 합치는 과정을 반복하여 만든다. 새 노드의 빈도수는 두 노드의 빈도수를 더한 값이다. 병합한 두 노드는 후보에서 빼고 새 노드를 넣어 다음 두 최소값을 고른다.

병합 과정은 다음과 같다.
5(d) + 6(f) = 11
7(a) + 10(c) = 17
11 + 17 = 28
20(b) + 28 = 48
48 + 50(e) = 98

두 노드를 합칠 때마다 그 아래에 속한 모든 문자의 코드 길이가 1비트씩 늘어난다. 따라서 각 병합은 해당 노드의 빈도수만큼 전체 비트 수를 늘리므로, **병합으로 얻은 빈도수를 모두 더하면** 총 비트 수가 된다.
11 + 17 + 28 + 48 + 98 = 202비트

이 값은 ②의 202와 일치한다.

NoteBase

문 17. 기수 정렬(radix sort)에 대한 설명으로 옳은 것만을 모두 고르면?

ㄱ. 비교 기반의 정렬 알고리즘이다.
ㄴ. 저장공간이 추가로 필요하다.
ㄷ. 안정적 정렬(stable sort) 알고리즘이다.
ㄹ. 시간 복잡도는 $O(n \log n)$이다.

① ㄱ, ㄷ
② ㄴ, ㄷ
③ ㄴ, ㄹ
④ ㄷ, ㄹ

정답 2번

간단한 해설

기수 정렬은 자릿값에 따라 자료를 분배하고 모으는 과정에서 **추가 저장공간**을 사용한다. 일반적인 기수 정렬은 같은 키를 가진 자료의 입력 순서를 유지하는 **안정적 정렬**이다.

보기별 해설

ㄱ. X: 기수 정렬은 두 자료의 키 전체를 비교하여 순서를 정하는 비교 기반 정렬이 아니다. 각 자릿값에 따라 자료를 버킷에 분배하고 순서대로 모으는 **비교 기반이 아닌 정렬**이다.

ㄴ. O: 자릿값별로 분배한 자료를 담을 버킷이나 보조 배열이 필요하므로 **추가 저장공간**을 사용한다.

ㄷ. O: 안정적 정렬은 같은 키를 가진 자료의 입력 순서를 유지하는 정렬이다. 일반적인 기수 정렬은 각 자릿수를 처리할 때 같은 자릿값을 가진 자료의 **상대적 순서를 유지**한다. 특히 낮은 자리부터 정렬하는 방식에서는 이 성질 덕분에 앞서 정렬한 낮은 자리의 순서가 보존된다. 전체 키가 같은 자료의 입력 순서도 유지된다.

ㄹ. X: 기수 정렬의 시간 복잡도는 자료 수뿐 아니라 자릿수와 기수에도 좌우된다. 자료 수를 n, 처리할 자릿수를 d, 한 자리에서 가능한 값의 수를 r이라고 하면, 일반적인 버킷 또는 계수 정렬 방식에서 한 자리의 처리는 $O(n + r)$이고 이를 d번 반복하므로 $O(d(n + r))$이다. **기수 r을 상수로 보면** $O(dn)$이며, 자릿수 d까지 일정하면 $O(n)$이다. 따라서 기수 정렬의 일반적인 시간 복잡도를 $O(n \log n)$으로 설명할 수 없다.

옳은 보기는 ㄴ, ㄷ이며, ②의 조합과 일치한다.

NoteBase

문 18. 다음은 0 이상의 모든 정수 n에 대하여 n!을 구하는 파이썬 코드이다. (가)에 들어갈 코드로 옳지 않은 것은?

```
def factorial(n):
    if (가):
        return 1
    return n * factorial(n - 1)
```

① $n == 0$

② n <= 0

③ $n == 1$

④ n <= 1

정답 3번

간단한 해설

'n == 1'만 종료 조건으로 두면 **n = 0에서 재귀 호출이 멈추지 않는다**. 입력에 0도 포함되므로, 0! = 1을 반환할 수 있도록 종료 조건이 0을 포함해야 한다.

공통 해설

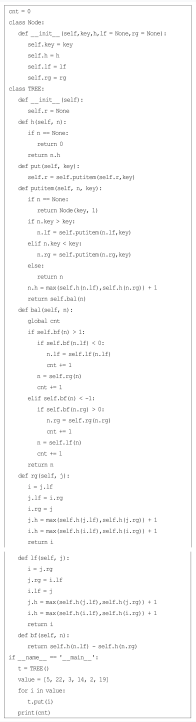
이 함수는 종료 조건이 참이면 1을 반환하고, 거짓이면 n을 1 줄여 자신을 다시 호출한 결과에 n을 곱한다. 팩토리얼은 0! = 1이고, 양의 정수에서는 $n! = n \cdot (n-1)!$이므로, 재귀 호출이 **0 또는 1에서 알맞게 종료**되도록 해야 한다. 특히 입력 범위가 0 이상의 모든 정수이므로 처음부터 n = 0인 경우도 처리해야 한다.

'n == 0'은 0에서 1을 반환하고, 양의 정수에서는 n을 줄여 결국 0에 도달하므로 올바르게 작동한다. 'n <= 0'도 주어진 입력 범위에서는 같은 방식으로 작동한다. 'n <= 1'은 0과 1에서 모두 1을 반환하는데, 0!과 1!은 모두 1이므로 이 조건도 사용할 수 있다.

반면 'n == 1'은 양의 정수 입력에서는 1에 도달하여 종료하지만, 0을 입력하면 n이 0 → -1 → -2 → …로 줄어들어 종료 조건을 만나지 못한다. 바깥 계산에 0을 곱하더라도 **재귀 호출의 결과를 먼저 구해야 하므로** 0이나 1이 정상적으로 반환되는 것은 아니다. 이 조건을 'n <= 1'로 바꾸면 0에서도 재귀 호출 없이 1을 반환한다.

NoteBase

문 19. 다음 파이썬 프로그램의 실행 결과는?



```
cnt = 0
class Node:
    def __init__(self,key,h,lf = None,rg = None):
        self.key = key
        self.h = h
        self.lf = lf
        self.rg = rg
class TREE:
    def __init__(self):
        self.r = None
    def h(self, n):
        if n == None:
            return 0
        return n.h
    def put(self, key):
        self.r = self.putitem(self.r,key)
    def putitem(self, n, key):
        if n == None:
            return Node(key, 1)
        if n.key > key:
            n.lf = self.putitem(n.lf,key)
        elif n.key < key:
            n.rg = self.putitem(n.rg,key)
        else:
            return n
        n.h = max(self.h(n.lf),self.h(n.rg)) + 1
        return self.bal(n)
    def bal(self, n):
        global cnt
        if self.bf(n) > 1:
            if self.bf(n.lf) < 0:
                n.lf = self.lf(n.lf)
                cnt += 1
            n = self.rg(n)
            cnt += 1
        elif self.bf(n) < -1:
            if self.bf(n.rg) > 0:
                n.rg = self.rg(n.rg)
                cnt += 1
            n = self.lf(n)
            cnt += 1
        return n
    def rg(self, j):
        i = j.lf
        j.lf = i.rg
        i.rg = j
        j.h = max(self.h(j.lf),self.h(j.rg)) + 1
        i.h = max(self.h(i.lf),self.h(i.rg)) + 1
        return i
    def lf(self, j):
        i = j.rg
        j.rg = i.lf
        i.lf = j
        j.h = max(self.h(j.lf),self.h(j.rg)) + 1
        i.h = max(self.h(i.lf),self.h(i.rg)) + 1
        return i
    def bf(self, n):
        return self.h(n.lf) - self.h(n.rg)
if __name__ == '__main__':
    t = TREE()
    value = [5, 22, 3, 14, 2, 19]
    for i in value:
        t.put(i)
    print(cnt)
```

① 1

② 2

③ 3

④ 4

정답 2번

간단한 해설

19를 삽입하면 노드 22의 왼쪽 자식 14의 오른쪽으로 가지가 늘어나 **LR 불균형**이 생긴다. 14에서 왼쪽 회전, 22에서 오른쪽 회전을 수행하며 **회전마다 cnt가 1씩 증가**하므로 출력값은 2이다.

NoteBase

계산 과정

실행 기준: 작은 키는 왼쪽, 큰 키는 오른쪽에 삽입한다. 빈 서브트리의 높이는 0, 새 노드의 높이는 1이며, 삽입 후 노드의 높이는 두 자식 높이 중 큰 값에 1을 더해 갱신한다. bf는 **왼쪽 높이 − 오른쪽 높이**이고, 값이 1보다 크거나 −1보다 작으면 bal에서 회전한다. cnt는 0에서 시작하여 회전 함수를 실행할 때마다 1씩 증가한다.

5·22·3 삽입: 5가 루트가 되고, 22는 5의 오른쪽 자식, 3은 왼쪽 자식이 된다. 22를 넣은 직후 5의 bf는 −1이고, 3까지 넣으면 0이다. 회전 조건에 해당하지 않아 cnt는 0이다.

14·2 삽입: 14는 22의 왼쪽 자식이 된다. 이때 22의 bf는 1, 루트 5의 bf는 1 − 2 = −1이므로 회전하지 않는다. 다음으로 2는 3의 왼쪽 자식이 된다. 3의 bf는 1이고, 5의 왼쪽과 오른쪽 서브트리 높이가 모두 2가 되어 bf는 0이다. 여전히 cnt는 0이다.

19 삽입: 19는 5의 오른쪽, 22의 왼쪽, 14의 오른쪽으로 내려가 14의 오른쪽 자식이 된다. 14의 bf는 0 − 1 = −1이므로 그대로 반환한다. 이어 노드 22에서는 왼쪽 높이가 2, 오른쪽 높이가 0이므로 bf가 2가 된다. **22는 왼쪽으로, 왼쪽 자식 14는 오른쪽으로 치우친 LR 상태**이므로 두 번의 회전이 필요하다.

첫 번째 회전: bal에서 22의 bf가 1보다 크고 왼쪽 자식 14의 bf가 0보다 작으므로 self.lf(n.lf)를 실행한다. lf는 오른쪽 자식을 위로 올리는 **왼쪽 회전**이다. 19가 올라가고 14가 그 왼쪽 자식이 되며, 22의 왼쪽 자식은 19로 바뀐다. 첫 번째 cnt += 1에 의해 cnt는 1이 된다.

두 번째 회전: 이어 self.rg(n)를 실행한다. rg는 왼쪽 자식을 위로 올리는 **오른쪽 회전**이므로 19가 올라가고 22가 그 오른쪽 자식이 된다. 이 서브트리는 19를 루트로 하여 왼쪽에 14, 오른쪽에 22를 둔다. 두 번째 cnt += 1에 의해 cnt는 2가 된다.

최종 출력: 전체 루트 5의 왼쪽 서브트리는 3과 그 왼쪽 자식 2로, 오른쪽 서브트리는 19와 그 두 자식 14·22로 구성된다. 양쪽 높이가 모두 2이므로 추가 회전은 없다. print(cnt)는 2를 출력하며, 이는 ②의 값과 일치한다.

NoteBase

문 20. 그래프의 임의의 한 점에서 출발하여 다른 모든 점을 한 번씩만 방문하고, 다시 시작점으로 돌아오는 경로 중 최단 경로를 찾는 문제는?

① 통 채우기(bin packing) 문제

② 정점 커버(vertex cover) 문제

③ 그래프 색칠하기(graph coloring) 문제

④ 여행자 문제(TSP, Traveling Salesman Problem)

정답 4번

간단한 해설

모든 정점을 한 번씩 방문하고 출발점으로 돌아오는 경로 중 **총 이동 거리가 가장 짧은 경로**를 찾는 것이 여행자 문제(TSP)이다.

공통 해설

여행자 문제(TSP)는 한 정점에서 출발하여 다른 모든 정점을 정확히 한 번씩 방문한 뒤 출발점으로 돌아오는 순회 경로 중, **전체 이동 거리의 합이 최소인 경로**를 찾는 문제이다. 출발점은 처음과 마지막에 나타나며, 나머지 정점은 중복해서 방문하지 않는다.

예를 들어 네 정점 A, B, C, D를 방문한다면 A → B → C → D → A처럼 출발점으로 돌아오는 경로가 후보가 된다. 이때 마지막 D → A 구간까지 포함한 총 거리를 다른 가능한 방문 순서의 총 거리와 비교한다. 문제의 '다른 모든 점을 한 번씩만 방문', '시작점으로 복귀', '최단 경로'는 이러한 여행자 문제의 방문 조건과 최소화 목표를 나타낸다.

NoteBase

문 21. 수행 시간 분석(run time analysis)에 관한 설명으로 옳지 않은 것은?

① 최악 경우(worst case) 분석은 상한의 의미가 있다.

② 최선 경우(best case) 분석은 수행 시간이 가장 적은 경우를 의미한다.

③ 평균 경우(average case) 분석은 모든 입력에 대한 수행 시간을 알고 계산하기 때문에 시간 복잡도의 척도로 많이 사용한다.

④ 상각(amortized) 분석은 일련의 연산을 수행하는 데 걸리는 총 시간을 연산 수로 나누어 계산한다.

정답 3번

간단한 해설

평균 경우 분석은 **입력의 발생 확률을 반영한 수행 시간의 기댓값**을 구한다. 모든 입력의 수행 시간과 확률분포를 파악하기 어려울 수 있으므로, 이를 쉽게 계산하여 널리 사용한다는 설명은 부적절하다.

선지별 해설

① O: 최악 경우 분석은 입력 크기가 같을 때 **수행 시간이 가장 긴 입력**을 기준으로 한다. 그 크기의 어떤 입력도 이 최대 수행 시간을 넘지 않으므로 수행 시간의 상한을 제시한다.

② O: 최선 경우 분석은 입력 크기가 같을 때 **수행 시간이 가장 짧은 입력**을 기준으로 한다. 예를 들어 순차 탐색에서 찾는 값이 첫 번째 위치에 있으면 한 번의 비교로 탐색이 끝난다. 다만 최선 경우의 시간만으로 다른 입력에서도 빠르게 실행된다고 보장할 수는 없다.

③ X: 평균 경우 분석은 같은 크기의 가능한 입력마다 수행 시간에 그 입력의 발생 확률을 곱한 뒤 합하여 **수행 시간의 기댓값**을 구한다. 모든 입력이 같은 확률로 발생한다고 가정할 때에만 수행 시간의 단순 산술평균이 된다. 실제 입력의 확률분포를 정하거나 각 경우의 수행 시간을 분석하기 어려울 수 있으므로, '모든 입력에 대한 수행 시간을 알고 계산하기 때문에 시간 복잡도의 척도로 많이 사용한다'는 설명은 부적절하다. 입력 분포에 관한 가정을 바탕으로 기대 수행 시간을 분석하는 방법으로 고쳐 이해해야 한다.

④ O: 상각 분석은 연산을 한 번씩 따로 평가하기보다 **연속된 연산 전체의 비용**을 평가한다. 총계법에서는 가능한 연산열의 총 수행 시간에 대한 상한을 구하고 이를 연산 수로 나누어 연산당 상각 비용을 얻는다. 일부 연산이 오래 걸리더라도 나머지 연산이 충분히 짧으면 연산당 비용은 작을 수 있다. 평균 경우 분석과 달리 입력의 발생 확률을 가정할 필요가 없다.

NoteBase

문 22. 문제를 해결하기 위한 알고리즘 개발 단계를 순서대로 바르게 나열한 것은?

(가) 알고리즘 구현
(나) 알고리즘 설계
(다) 알고리즘 분석
(라) 알고리즘 정확성 검증

① (나)→(다)→(라)→(가)
② (나)→(라)→(다)→(가)
③ (다)→(나)→(라)→(가)
④ (다)→(라)→(나)→(가)

정답 2번

간단한 해설

설계한 알고리즘의 효율성을 따지기 전에 **문제를 올바르게 해결하는지** 검증한다. 수행 시간과 메모리 사용량은 실제 코드를 작성하기 전에도 알고리즘의 처리 절차를 바탕으로 분석할 수 있다.

올바른 순서

(나) → (라) → (다) → (가)

(나) 알고리즘 설계: 문제의 입력과 요구되는 출력을 파악하고, 이를 연결하는 **구체적인 처리 절차**를 만든다. 검증과 분석의 대상이 되는 알고리즘이 먼저 마련되어야 한다.

(라) 알고리즘 정확성 검증: 설계한 절차가 허용된 입력에 대해 종료하고 요구되는 결과를 내는지 논리적으로 확인한다. 빠르게 실행되더라도 결과가 틀리면 문제를 해결한 것이 아니므로, 이 절차에서는 **정확성을 확인한 뒤 효율성을 분석**한다.

(다) 알고리즘 분석: 정확성이 확인된 알고리즘에 대해 입력 크기에 따른 수행 시간과 메모리 사용량을 평가한다. 연산 횟수와 필요한 저장 공간을 따지는 것이므로 **구현 전에도 분석할 수 있다**. 이를 통해 알고리즘의 자원 요구량이 적절한지 판단한다.

(가) 알고리즘 구현: 정확성과 효율성을 검토한 처리 절차를 프로그래밍 언어의 코드로 옮긴다. 앞 단계에서 마련하고 검토한 알고리즘을 컴퓨터에서 실행할 수 있는 프로그램으로 구체화하는 단계이다.

NoteBase

문 23. 그리디 알고리즘에 해당하는 것은?

① 크루스칼(Kruskal) 알고리즘

② 0/1 배낭(knapsack) 문제 풀이

③ 플로이드-워셜(Floyd-Warshall) 알고리즘

④ 최장 공통 부분 수열(LCS, Longest Common Subsequence) 구하기

정답 1번

간단한 해설

크루스칼 알고리즘은 **사이클을 만들지 않는 가장 가벼운 간선**을 차례로 선택하여 최소 신장 트리를 만든다. 각 단계에서 유리한 선택을 확정해 나가므로 그리디 알고리즘에 해당한다.

공통 해설

그리디 알고리즘은 각 단계에서 현재 기준으로 가장 유리한 선택을 하고, 그 선택을 되돌리지 않으면서 해를 구성하는 방법이다. 이 방법으로 전체 최적해를 얻으려면 **현재의 선택이 전체 최적해로 이어질 수 있다는 보장**이 있어야 한다. 단순히 눈앞의 이익이 큰 것을 고른다고 항상 최적해가 되는 것은 아니다.

크루스칼 알고리즘은 연결된 무방향 가중치 그래프에서 모든 정점을 연결하면서 간선 가중치의 합이 최소인 신장 트리를 구한다. 간선을 가중치가 작은 순서로 살펴보고, **서로 다른 연결 요소를 잇는 간선만 채택**하여 사이클을 피한다. 이미 같은 연결 요소에 속한 두 정점을 잇는 간선은 건너뛴다. 이렇게 현재 채택할 수 있는 가장 가벼운 간선을 확정해도 최소 신장 트리를 구성할 수 있으므로 그리디 전략이 성립한다.

0/1 배낭 문제 풀이, 플로이드-워셜 알고리즘, 최장 공통 부분 수열(LCS) 구하기는 대표적으로 동적 계획법을 적용하는 대상이다. 동적 계획법은 **부분 문제의 최적값을 저장하고 여러 후보를 비교·결합**하여 전체 최적값을 구한다. 현재 가장 유리해 보이는 선택 하나를 곧바로 확정하는 그리디 전략과 구별된다.

NoteBase

문 24. 정렬 알고리즘에 대한 설명으로 옳은 것은?

① 힙 정렬의 시간 복잡도는 $O(n^2)$이다.

② 퀵 정렬은 정렬할 전체 원소에 대해 정렬을 수행한 후 기준값을 중심으로 왼쪽 부분집합과 오른쪽 부분집합으로 분할한다.

③ 삽입 정렬은 입력에 민감한 알고리즘으로, 입력이 거의 정렬되어 있을 때 처리시간이 느리고, 입력이 역으로 정렬되어 있을 때는 최선의 경우로 빠르다.

④ 병합 정렬은 크기가 n인 입력을 n/2 크기로 분할하고, 각각에 대해 같은 방식으로 정렬을 수행한 후, 2개의 각각 정렬된 부분을 합병한다.

정답 4번

간단한 해설

병합 정렬은 입력을 나누어 각 부분을 재귀적으로 정렬한 뒤, **이미 정렬된 두 부분을 합병**한다. 합병할 때 두 부분의 맨 앞 원소 중 작은 것을 차례로 꺼내면 전체가 정렬된다.

선지별 해설

① X: 힙 정렬의 시간 복잡도는 통상 최악의 경우를 기준으로 $O(n\log n)$으로 나타낸다. 먼저 $O(n)$에 힙을 구성하고, 최댓값을 꺼내어 뒤에 배치한 뒤 힙을 복구하는 과정을 반복한다. 한 번의 힙 복구에 최대 $O(\log n)$이 걸리므로 전체 정렬은 **최악의 경우에도** $O(n\log n)$에 수행된다.

② X: 퀵 정렬은 전체 원소를 정렬한 후 나누는 것이 아니라 **기준값을 중심으로 먼저 분할**한 뒤, 왼쪽과 오른쪽 부분을 각각 재귀적으로 정렬한다. 오름차순이라면 기준값보다 작은 원소는 왼쪽에, 큰 원소는 오른쪽에 놓이도록 분할하며, 이 시점에 각 부분 내부까지 정렬되어 있을 필요는 없다.

③ X: 삽입 정렬은 앞쪽의 정렬된 부분에 다음 원소를 알맞게 끼워 넣는 방식이다. 입력이 **거의 정렬되어 있으면 원소를 옮기는 횟수가 적어 빠르다**. 이미 오름차순으로 정렬된 입력이 최선의 경우로 시간 복잡도는 $O(n)$이다. 서로 다른 원소가 역순으로 정렬되어 있으면 새 원소를 삽입할 때마다 앞의 원소들을 모두 한 칸씩 옮겨야 하므로 최악의 경우인 $O(n^2)$이 된다. 선지는 빠른 경우와 느린 경우를 반대로 설명했다.

④ O: 병합 정렬은 입력을 **두 부분으로 나누고 각각 정렬한 뒤 합병**하는 분할 정복 알고리즘이다. 원소가 하나인 부분은 이미 정렬된 것으로 보고, 재귀 호출에서 돌아오면서 정렬된 두 부분을 하나로 합친다. 예를 들어 [1, 4]와 [2, 3]을 합병하면 각 부분의 맨 앞 원소를 비교해 1, 2, 3, 4 순서로 꺼낸다. 입력 크기가 짝수이면 두 부분의 크기는 각각 $n/2$이고, 홀수이면 두 부분의 크기가 1만큼 다르도록 나눈다.

NoteBase

문 25. 다음 파이썬 프로그램의 실행 결과는?

```
class NodeBS(object):
    def __init__(self, vl = None):
        self.vl = vl
        self.root = None
        self.lf = None
        self.rg = None
    def _ad(self, vl):
        nd = NodeBS(vl)
        if vl > self.vl:
            self.rg = self.rg and self.rg._ad(vl) or nd
        elif vl < self.vl:
            self.lf = self.lf and self.lf._ad(vl) or nd
        else:
            print("node duplicated")
        return self
class Bst(NodeBS):
    def add_node(self, vl):
        if not self.root:
            self.root = NodeBS(vl)
        else:
            self.root._ad(vl)
    def order(self):
        current = self.root
        nodes, stack = [], []
        while stack or current:
            if current:
                nodes.append(current.vl)
                stack.append(current)
                current = current.lf
            else:
                current = stack.pop()
                current = current.rg
        return nodes
if __name__ == "__main__":
    bt = Bst()
    in_list = [6, 4, 8, 2, 5, 7, 9, 1, 3, 10]
    for i in in_list:
        bt.add_node(i)
    print(bt.order())
```

① [6, 4, 2, 1, 3, 8, 5, 7, 9, 10]

② [6, 4, 2, 1, 5, 8, 3, 7, 9, 10]

③ [6, 4, 2, 1, 3, 5, 8, 7, 9, 10]

④ [6, 4, 2, 1, 5, 3, 8, 7, 9, 10]

정답 3번

간단한 해설

order()는 현재 노드의 값을 **왼쪽 자식으로 이동하기 전에 기록**하므로 전위 순회이다. 6의 왼쪽 부분 트리를 모두 방문해야 오른쪽의 8로 넘어가므로, 3 다음에는 5가 기록된다.

NoteBase

올바른 순서

6 → 4 → 2 → 1 → 3 → 5 → 8 → 7 → 9 → 10

6: add_node()는 첫 값 6을 루트로 만들고, 이후 값은 현재 노드보다 작으면 왼쪽(lf), 크면 오른쪽(rg)에 삽입한다. order()는 루트에서 시작해 nodes에 값을 넣고, 해당 노드를 스택에 저장한 뒤 왼쪽으로 이동한다. 따라서 **루트 6을 먼저 기록**한다. 왼쪽으로 더 갈 수 없을 때는 스택에서 노드를 꺼내 그 노드의 오른쪽으로 이동하므로, 전체 방문 방식은 '루트 → 왼쪽 부분 트리 → 오른쪽 부분 트리'이다.

4: 4는 6보다 작으므로 6의 왼쪽 자식이다. 6에서 왼쪽으로 이동한 뒤 4를 기록하고 스택에 넣는다.

2: 2는 6과 4보다 작으므로 4의 왼쪽 자식이다. 4에서 다시 왼쪽으로 이동해 2를 기록하고 스택에 넣는다.

1: 1은 2의 왼쪽 자식이므로 다음에 기록된다. 1은 자식이 없어 왼쪽 이동이 끝나고, 스택에서 1을 꺼내도 오른쪽 자식이 없으므로 다시 2를 꺼낸다.

3: 3은 4보다 작고 2보다 크므로 2의 오른쪽 자식이다. 스택에서 꺼낸 2의 오른쪽으로 이동해 3을 기록한다. **스택에서 꺼내는 동작 자체는 값을 다시 기록하지 않는다.**

5: 3의 자식이 없으므로 3을 꺼낸 뒤, 스택에 남아 있는 4를 꺼내 오른쪽으로 이동한다. 5는 6보다 작고 4보다 크므로 4의 오른쪽 자식이다. 따라서 **6의 오른쪽으로 넘어가기 전에 5를 기록**한다.

8: 5의 방문을 마치면 스택에서 5를 꺼내고, 이어서 6을 꺼낸다. 6의 오른쪽 자식은 8이므로 8로 이동해 그 값을 기록하고 스택에 넣는다.

7: 7은 6보다 크고 8보다 작으므로 8의 왼쪽 자식이다. 8에서 왼쪽으로 이동해 7을 기록한다.

9: 7의 자식이 없으므로 스택에서 7을 꺼낸 뒤 8을 꺼내 오른쪽으로 이동한다. 9는 8의 오른쪽 자식이므로 다음에 기록된다.

10: 9의 왼쪽 자식이 없어 9를 스택에서 꺼내고, 그 오른쪽 자식인 10으로 이동해 값을 기록한다. 10도 자식이 없으므로 이를 꺼낸 뒤에는 스택이 비고 current도 None이 되어 반복이 끝난다. 반환된 nodes 리스트가 print()로 출력된다.

NoteBase