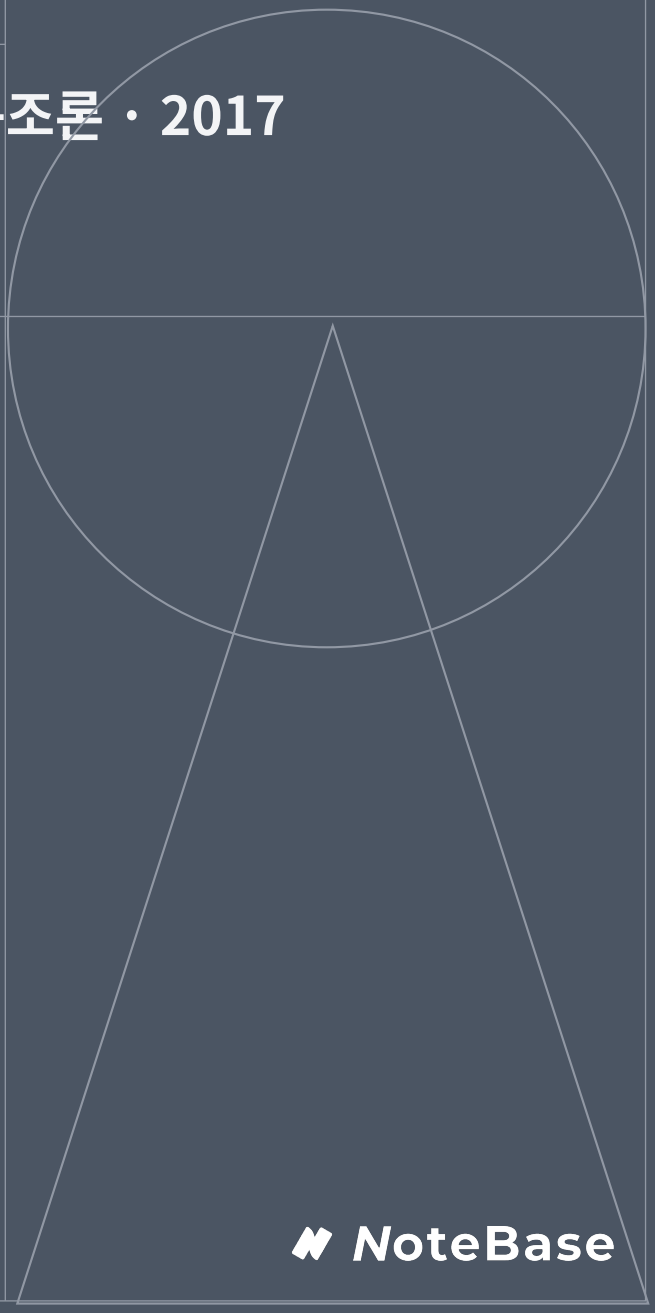
# 기출OX 해설집

공무원/국가직/7급 · 자료구조론 · 2017



NoteBase

# 기출OX 해설집

## 공무원/국가직/7급 · 자료구조론 · 2017

### 공무원/국가직/7급 · 자료구조론 · 2017

**문 1. 다음 C 언어 함수를 사용하여 function(9)를 수행했을 때, 반환되는 값은?**

```
int function(int n)
{
  if (n <= 1) return 1;
  if (n % 2 == 0)
    return n * function(n-1);
  else
    return n * function(n-3);
}
```

① 60

② 84

③ 540

④ 672

**정답 3번**

**출제 포인트**

재귀 함수의 종료 조건과 분기 조건을 정확히 추적해 최종 반환값을 계산할 수 있는지를 묻는다. 이 함수는 n이 짝수인지 홀수인지에 따라 다음 호출에서 n을 1 줄이는지 3 줄이는지가 갈리므로, 각 호출 단계에서 조건을 놓치지 않고 순서대로 대입하는 것이 핵심이다.

NoteBase

### 조건 정리

function(n)의 동작은 다음과 같다.
n이 1 이하이면 1을 반환하고 재귀를 종료한다.
n이 짝수이면 n * function(n-1)을 반환한다.
n이 홀수이면(위 두 조건에 해당하지 않으면) n * function(n-3)을 반환한다.
구하는 값은 function(9)이다.

### 사용할 공식

짝수 n에 대해 $f(n) = n \times f(n-1)$이 성립하고, 홀수 n(단, $n > 1$)에 대해 $f(n) = n \times f(n-3)$이 성립하며, $f(n) = 1\ (n \leq 1)$이 재귀의 종료 조건이다.

### 계산 과정

function(9): 9는 홀수이므로 9 * function(6)이다.
function(6): 6은 짝수이므로 6 * function(5)이다.
function(5): 5는 홀수이므로 5 * function(2)이다.
function(2): 2는 짝수이므로 2 * function(1)이다.
function(1): n이 1 이하이므로 1을 반환한다.
이제 역으로 대입한다.
function(2) = 2 * 1 = 2
function(5) = 5 * 2 = 10
function(6) = 6 * 10 = 60
function(9) = 9 * 60 = 540
따라서 function(9)의 반환값은 540이다.

### 선지 판단

① 60 → X, 이는 function(6)의 값이지 function(9)의 최종 반환값이 아니다.
② 84 → X, n이 짝수일 때도 매번 n-3을 적용하는 식으로 잘못 계산한 경우에 나올 수 있는 값으로, 홀수/짝수 분기를 혼동한 결과이다.
③ 540 → O, 홀수는 n-3, 짝수는 n-1로 정확히 분기하여 역대입하면 9 * 6 * 5 * 2 * 1 = 540이 된다.
④ 672 → X, 분기 조건을 반대로 적용(짝수에 n-3, 홀수에 n-1)하면 나올 수 있는 값으로 조건을 뒤바꾼 오류이다.

NoteBase

### 오답 포인트

가장 흔한 실수는 if (n % 2 == 0)의 짝수 조건과 else의 홀수 조건을 뒤바꿔 적용하는 것이다. 9는 홀수이므로 반드시 else branch인 n * function(n-3)으로 들어가야 하는데, 이를 n * function(n-1)로 착각하면 함수 6은 만들지만 이후 흐름이 완전히 달라져 540이 아닌 다른 값이 나온다. 또한 중간값(예: 60)을 최종 답으로 착각하지 않도록 역대입을 끝까지 마쳐야 한다.

### 다시 나오면 이렇게 판단

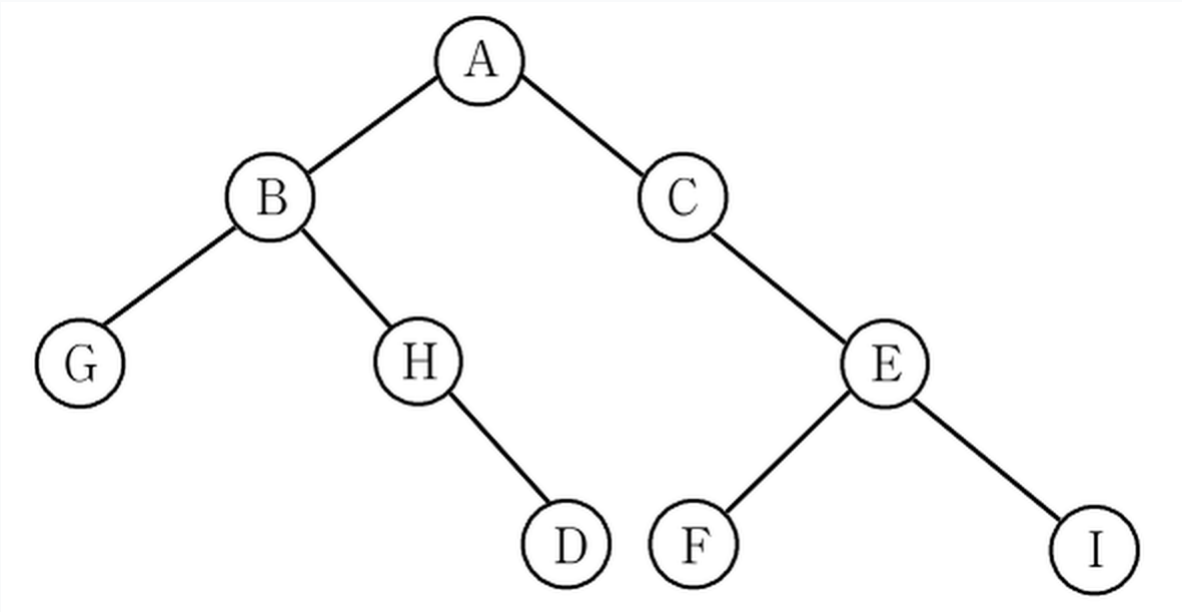
재귀 함수 문제는 가장 안쪽 호출(종료 조건에 가장 가까운 인자)부터 값을 확정한 뒤 바깥쪽으로 역대입하는 순서로 푼다. 매 단계마다 현재 n이 어느 조건문에 해당하는지(짝수인지 홀수인지, 종료 조건인지)를 표시해 가며 호출 체인을 적어 내려가면 분기를 혼동하지 않는다.

### 암기 포인트

n % 2 == 0은 짝수(2로 나눈 나머지가 0), else는 홀수를 의미한다는 것을 재귀 함수 문제에서 항상 먼저 표시해 둔다.

NoteBase

**문 2. 다음 이진 트리(binary tree)의 순회 방법으로 옳지 않은 것은?**




① 전위(preorder) 순회: A B G H D C E F I

② 중위(inorder) 순회: G B H D A C F E I

③ 후위(postorder) 순회: G D H B F E I C A

④ 레벨 순서(level order) 순회: A B C G H E D F I

**정답 3번**

**출제 포인트**

주어진 이진 트리 구조에서 전위·중위·후위·레벨 순서 네 가지 순회 결과를 직접 계산해 보고, 제시된 보기 중 계산 결과와 다른 하나를 찾아내는 문제이다. 트리 순회의 정의(방문 순서 규칙)를 정확히 적용할 수 있는지가 핵심이다.

**조건 정리**

트리 구조는 다음과 같다. 루트는 A이고, A의 왼쪽 자식은 B, 오른쪽 자식은 C이다. B의 왼쪽 자식은 G, 오른쪽 자식은 H이며, H는 오른쪽 자식으로 D를 가진다(왼쪽 자식 없음). C는 왼쪽 자식이 없고 오른쪽 자식으로 E를 가지며, E의 왼쪽 자식은 F, 오른쪽 자식은 I이다.

**사용할 공식**

전위 순회는 (루트 → 왼쪽 서브트리 → 오른쪽 서브트리) 순서로 방문한다. 중위 순회는 (왼쪽 서브트리 → 루트 → 오른쪽 서브트리) 순서로 방문한다. 후위 순회는 (왼쪽 서브트리 → 오른쪽 서브트리 → 루트) 순서로 방문한다. 레벨 순서 순회는 트리를 위에서 아래로, 같은 레벨에서는 왼쪽에서 오른쪽으로 방문한다.

NoteBase

**계산 과정**

전위 순회: A를 방문한 뒤 왼쪽 서브트리(B 이하)를 전위로, 이어서 오른쪽 서브트리(C 이하)를 전위로 방문한다. B 이하는 B, G, H, D 순이고 C 이하는 C, E, F, I 순이므로 전체는 A B G H D C E F I가 된다.
중위 순회: 왼쪽 서브트리(B 이하)를 중위로 먼저 방문하고 A, 그다음 오른쪽 서브트리(C 이하)를 중위로 방문한다. B 이하 중위는 G, B, H, D이고 C 이하 중위는 C, F, E, I이므로 전체는 G B H D A C F E I가 된다.
후위 순회: 왼쪽 서브트리(B 이하)를 후위로, 오른쪽 서브트리(C 이하)를 후위로 방문한 뒤 마지막에 A를 방문한다. B 이하 후위는 G, D, H, B이다(G를 먼저, 그다음 H의 서브트리인 D, H, 마지막 B). C 이하 후위는 E의 서브트리를 먼저 처리하는데 E의 왼쪽 자식 F, 오른쪽 자식 I를 방문한 뒤 E를 방문하므로 F, I, E가 되고 마지막에 C를 방문해 F I E C가 된다. 따라서 전체 후위 순회는 G D H B F I E C A가 되어야 한다. 그런데 ③번 보기는 G D H B F E I C A로, E와 I의 순서가 서로 바뀌어 있다(I가 E보다 먼저 방문되어야 하는데 E가 먼저 나옴). 이것이 옳지 않은 부분이다.
레벨 순서 순회: 1레벨 A, 2레벨 B C, 3레벨 G H E, 4레벨 D F I 순으로 방문하면 A B C G H E D F I가 되어 정확히 일치한다.

**선지 판단**

① 전위 순회 A B G H D C E F I: 계산 결과와 정확히 일치한다 → O.
② 중위 순회 G B H D A C F E I: 계산 결과와 정확히 일치한다 → O.
③ 후위 순회 G D H B F E I C A: 올바른 후위 순회는 G D H B F I E C A이다. E의 두 자식(F, I)을 모두 방문한 다음 E 자신을 방문해야 하므로 I가 E보다 앞서야 하는데, 제시된 보기는 F E I C 순서로 되어 있어 E와 I의 순서가 뒤바뀌었다 → X(틀린 표현 F E I C → 올바른 표현 F I E C).
③ 레벨 순서 순회 A B C G H E D F I: 계산 결과와 정확히 일치한다 → O.

**오답 포인트**

후위 순회에서 자주 하는 실수는 형제 노드 중 자기보다 먼저 등장해야 할 노드를 놓치고 부모를 먼저 적어버리는 것이다. E의 왼쪽 자식 F와 오른쪽 자식 I를 모두 방문한 뒤에야 E를 방문할 수 있는데, I를 빠뜨리고 F 다음 곧바로 E를 적으면 이번 문제의 오답 보기처럼 F E I C 순서가 되어버린다. 후위 순회는 항상 '왼쪽 자식, 오른쪽 자식, 그다음 자신'이라는 순서를 서브트리 단위로 재귀적으로 지켜야 한다.

**다시 나오면 이렇게 판단**

순회 순서를 묻는 문제는 각 보기를 통째로 외우려 하지 말고, 트리를 작은 서브트리 단위로 쪼개어 각 서브트리의 순회 결과를 먼저 구한 뒤 이를 조합하는 방식으로 검산한다. 특히 후위 순회는 마지막에 등장하는 노드가 항상 그 서브트리의 루트라는 점을 이용해, 주어진 보기의 마지막 글자부터 거꾸로 확인하면 오류를 빠르게 찾을 수 있다.

NoteBase

**암기 포인트**

전위=루트·왼쪽·오른쪽, 중위=왼쪽·루트·오른쪽, 후위=왼쪽·오른쪽·루트, 레벨 순서=위에서 아래로 왼쪽에서 오른쪽으로 한 층씩. 후위 순회에서는 어떤 노드든 자신의 두 자식을 모두 방문한 다음에야 등장한다는 규칙을 기준으로 삼아 검산한다.

NoteBase

**문 3. 다음과 같이 인접 행렬로 표현된 가중 그래프(weighted graph)에 대하여 최소 비용 신장 트리(minimum cost spanning tree)를 구성하였을 때, 최소 비용은?**

| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 5 | 1 | 2 |
| [1] | 5 | 0 | ∞ | 4 |
| [2] | 1 | ∞ | 0 | 3 |
| [3] | 2 | 4 | 3 | 0 |

① 6

② 7

③ 8

④ 9

**정답 2번**

**출제 포인트**

인접 행렬로 주어진 가중 그래프에서 최소 비용 신장 트리(MST)를 구성하고 그 총비용을 계산할 수 있는지를 묻는다. 정점이 4개(0,1,2,3)이므로 MST는 간선 3개로 구성된다.

**조건 정리**

인접 행렬에서 대각선은 0(자기 자신), ∞는 두 정점 사이에 간선이 없음을 뜻한다. 행렬을 읽으면 존재하는 간선과 가중치는 다음과 같다: (0,2)=1, (0,3)=2, (2,3)=3, (1,3)=4, (0,1)=5. (1,2)는 ∞이므로 간선이 없다.

**사용할 공식**

크루스칼(Kruskal) 알고리즘: 간선을 가중치 오름차순으로 정렬한 뒤, 사이클을 만들지 않는 간선만 순서대로 선택하여 정점 수 -1개(여기서는 3개)의 간선을 모으면 그 가중치 합이 최소 비용 신장 트리의 비용이다.

NoteBase

**계산 과정**

간선을 가중치 순으로 정렬하면 (0,2)=1, (0,3)=2, (2,3)=3, (1,3)=4, (0,1)=5 순이다.
1) (0,2)=1 선택 → {0,2} 연결, 비용 누적 1
2) (0,3)=2 선택 → {0,2,3} 연결, 사이클 없음, 비용 누적 1+2=3
3) (2,3)=3 은 0,2,3이 이미 모두 연결되어 있어 선택하면 사이클이 생기므로 제외
4) (1,3)=4 선택 → 정점 1이 합류하여 {0,1,2,3} 전부 연결, 사이클 없음, 비용 누적 3+4=7
간선 3개(정점 4개 -1)를 모두 선택했으므로 종료. 총 최소 비용은 7이다.

**선지 판단**

① 6 → X, 간선 3개를 어떻게 조합해도 사이클 없이 6을 만들 수 없다(최소 조합은 1+2+4=7).
② 7 → O, (0,2)=1, (0,3)=2, (1,3)=4를 선택한 신장 트리의 비용 합이 7로 최소이다.
③ 8 → X, 예를 들어 (0,2)=1, (0,3)=2, (2,3)=3를 모두 택하면 0-2-3 사이에 사이클이 생겨 신장 트리가 될 수 없으므로 8은 유효한 신장 트리 비용이 아니다.
④ 9 → X, (0,1)=5를 포함하면서 신장 트리를 구성할 수도 있으나(1+3+5=9) 이는 (1,3)=4를 쓰는 조합보다 비용이 크므로 최소 비용이 아니다.

**오답 포인트**

간선을 가중치 순으로 나열한 뒤 사이클 여부를 확인하지 않고 작은 값부터 무조건 다 더하면(1+2+3=6) 정점 3개가 서로 사이클을 이루는 상태를 신장 트리로 착각하게 되어 6이라는 오답이 나온다. 또한 정점 1과의 연결 간선으로 (1,3)=4 대신 (0,1)=5를 선택하면 9라는 오답이 나오므로, 정점 1을 가장 싼 간선으로 연결하는지 확인해야 한다.

**다시 나오면 이렇게 판단**

인접 행렬이 제시되면 먼저 존재하는 간선과 가중치만 뽑아 오름차순으로 정렬한다. 이후 각 간선을 순서대로 보면서 이미 연결된 정점끼리를 잇는 간선(사이클 유발)은 건너뛰고, 정점 수-1개의 간선이 모일 때까지 반복한다. 결과로 모인 간선들의 가중치 합이 곧 최소 비용이다.

**암기 포인트**

크루스칼 알고리즘: 간선을 가중치 오름차순 정렬 → 사이클을 만들지 않는 간선만 선택 → 정점 수-1개의 간선이 모이면 종료, 그 합이 MST 비용이다.

NoteBase

**문 4. 78개의 간선(edge)으로 이루어진 이진 트리(binary tree)가 가질 수 있는 최대 높이와 최소 높이의 차이는? (단, 루트 노드만 존재하는 이진 트리의 높이는 1이다)**

① 71

② 72

③ 73

④ 74

**정답 2번**

**출제 포인트**

노드 수가 정해진 이진 트리에서 높이가 취할 수 있는 최댓값과 최솟값을 구하는 문제이다. 이때 높이의 정의를 정확히 적용하는 것이 핵심인데, 이 문제는 '루트 노드만 있는 트리의 높이는 1'이라고 명시하여 높이를 간선 수가 아니라 노드 수(레벨 수) 기준으로 센다는 점을 알려 주고 있다.

**조건 정리**

간선(edge)이 78개이므로 노드 수는 간선 수보다 1개 많다. 이진 트리에서 노드 수를 $n$이라 하면 간선 수는 $n-1$이므로 $n-1=78$, 즉 $n=79$이다. 높이는 문제 조건에 따라 루트만 있을 때 1이 되도록 '레벨(노드) 수'로 정의된다.

**사용할 공식**

노드 수가 $n$일 때 이진 트리가 가질 수 있는 최대 높이는 모든 노드가 한 줄로 이어진 편향 트리(skewed tree)의 경우이며, 이때 높이는 $n$과 같다. 최소 높이는 각 레벨을 최대한 채운 포화(완전)에 가까운 트리에서 나오며, 레벨 수 기준 높이는 $h=\lceil \log_2(n+1) \rceil$로 계산한다(레벨 $h$까지 채운 포화 이진 트리의 최대 노드 수가 $2^h-1$이므로 $2^h-1 \geq n$을 만족하는 최소 $h$를 구하는 것과 같다).

**계산 과정**

최대 높이: 79개의 노드를 모두 한 줄로 연결한 편향 트리는 각 레벨에 노드가 하나씩만 있으므로 높이는 노드 수와 같은 79가 된다.
최소 높이: $2^h-1 \geq 79$를 만족하는 최소 $h$를 찾는다. $2^6-1=63<79$이고 2^7-1=127≥79이므로 최소 높이는 $h=7$이다(레벨 1~6까지 완전히 채운 63개 노드에 남은 16개 노드를 7번째 레벨에 배치하면 79개를 모두 담으면서 높이 7을 유지할 수 있다).
따라서 최대 높이와 최소 높이의 차이는 $79-7=72$이다.

NoteBase

### 선지 판단

① 71 → 오답: 79와 7의 차이를 잘못 계산하거나 최소 높이를 8로 잘못 구했을 때 나오는 값이다(79-8=71).
② 72 → 정답: 최대 높이 79, 최소 높이 7이므로 차이는 79-7=72이다.
③ 73 → 오답: 최소 높이를 6으로 잘못 구했을 때 나오는 값이다(79-6=73).
④ 74 → 오답: 최대 높이를 78(간선 수를 그대로 높이로 착각)로 두고 최소 높이를 4로 잘못 구하는 등의 계산 오류에서 나올 수 있는 값으로, 조건에 맞지 않는다.

### 오답 포인트

이 문제에서 가장 흔한 실수는 간선 수 78을 그대로 높이로 쓰거나, 노드 수 79를 구하지 않고 78을 기준으로 계산하는 것이다. 또한 최소 높이를 구할 때 $\log_2 n$을 그냥 반올림하거나 내림하여 6 또는 8로 잘못 구하는 경우가 많으므로, $2^h - 1 \geq n$을 만족하는 최소 $h$를 직접 대입해 확인하는 습관이 필요하다.

### 다시 나오면 이렇게 판단

먼저 간선 수에 1을 더해 노드 수 $n$을 구한다. 최대 높이는 편향 트리를 가정해 $n$과 같다고 둔다. 최소 높이는 $2^h - 1 \geq n$을 만족하는 최소 정수 $h$를 $h = 1, 2, 3, \ldots$ 순서로 대입해 직접 찾는다. 마지막으로 문제에서 높이를 노드 수 기준(루트만 있으면 1)으로 정의했는지, 간선 수 기준(루트만 있으면 0)으로 정의했는지 반드시 확인한 뒤 차이를 계산한다.

### 암기 포인트

노드 수 $n$인 이진 트리의 최대 높이(레벨 기준) = $n$(편향 트리). 최소 높이(레벨 기준) = $2^h - 1 \geq n$을 만족하는 최소 $h$, 즉 $h = \lceil \log_2 (n + 1) \rceil$. 간선 수 = 노드 수 $-1$.

NoteBase

**문 5. 1과 1000 사이의 정수로 구성된 이진 탐색 트리(binary search tree)에서 숫자 629를 탐색한다고 할 때, 옳지 않은 비교 순서는?**

① 19, 929, 172, 891, 224, 735, 405, 674, 537, 629

② 876, 807, 24, 186, 314, 717, 690, 428, 512, 629

③ 29, 829, 815, 137, 369, 504, 805, 589, 692, 629

④ 926, 87, 813, 318, 572, 826, 635, 598, 580, 629

**정답 4번**

### 출제 포인트

이진 탐색 트리(BST)에서 특정 값을 찾아가는 비교 순서가 실제로 하나의 트리에서 나올 수 있는 경로인지를 판단하는 문제이다. 각 선지는 루트에서 시작해 목표값 629와 비교한 노드들을 순서대로 나열한 것인데, 이 나열이 BST의 성질(왼쪽 서브트리는 더 작은 값, 오른쪽 서브트리는 더 큰 값)을 어긋나지 않고 지키는지를 확인해야 한다.

### 조건 정리

탐색 대상은 629이다. 각 선지는 629를 찾기 위해 차례로 비교한 노드값의 나열이며, 마지막 값은 항상 629 자신이다. 이진 탐색 트리에서는 루트에서 출발해 목표값보다 크면 왼쪽, 작으면 오른쪽으로 내려가므로, 한 번 어떤 노드를 지나면 그 다음에 나오는 노드는 그 노드가 정한 구간(하한과 상한) 안에 반드시 있어야 한다.

### 사용할 공식

구간 갱신 규칙을 사용한다. 하한을 $lo$, 상한을 $hi$라 하고 초기값은 $lo = -\infty$, $hi = +\infty$로 둔다. 다음 비교 노드값 $v$는 반드시 $lo < v < hi$를 만족해야 유효한 경로이다. 그 다음 목표값 629와 $v$를 비교해 $629 < v$이면 $hi = v$로, $629 > v$이면 $lo = v$로 구간을 갱신한다. 나열된 모든 노드에 대해 이 조건이 한 번이라도 깨지면 그 순서는 불가능한 탐색 경로이다.

NoteBase

**계산 과정**

①번: 19(lo=19) → 929(hi=929) → 172(19<172<929, lo=172) → 891(172<891<929, hi=891) → 224(172<224<891, lo=224) → 735(224<735<891, hi=735) → 405(224<405<735, lo=405) → 674(405<674<735, hi=674) → 537(405<537<674, lo=537) → 629(537<629<674, 도달). 모든 구간 조건이 지켜지므로 가능한 경로이다.
②번: 876(hi=876) → 807(hi=807, 807<876 성립) → 24(lo=24) → 186(24<186<807, lo=186) → 314(186<314<807, lo=314) → 717(314<717<807, hi=717) → 690(314<690<717, hi=690) → 428(314<428<690, lo=428) → 512(428<512<690, lo=512) → 629(512<629<690, 도달). 모든 조건이 지켜지므로 가능한 경로이다.
③번: 29(lo=29) → 829(hi=829) → 815(29<815<829, hi=815) → 137(29<137<815, lo=137) → 369(137<369<815, lo=369) → 504(369<504<815, lo=504) → 805(504<805<815, hi=805) → 589(504<589<805, lo=589) → 692(589<692<805, hi=692) → 629(589<629<692, 도달). 모든 조건이 지켜지므로 가능한 경로이다.
④번: 926(hi=926) → 87(lo=87) → 813(87<813<926, hi=813) → 318(87<318<813, lo=318) → 572(318<572<813, lo=572) → 826: 현재 구간은 (572, 813)인데 826은 813보다 커서 이 구간에 들어오지 못한다. 즉 572 다음에 826이 나오려면 826이 813보다 작아야 하는데 그렇지 않으므로, 813을 지난 뒤에는 어떤 값도 813을 넘어설 수 없다. 이 지점에서 이미 모순이 발생하므로 ④번은 하나의 이진 탐색 트리에서 나올 수 없는 비교 순서이다.

**선지 판단**

① 19, 929, 172, 891, 224, 735, 405, 674, 537, 629 → O, 매 단계 구간 조건을 모두 만족하는 가능한 경로이다.
② 876, 807, 24, 186, 314, 717, 690, 428, 512, 629 → O, 매 단계 구간 조건을 모두 만족하는 가능한 경로이다.
③ 29, 829, 815, 137, 369, 504, 805, 589, 692, 629 → O, 매 단계 구간 조건을 모두 만족하는 가능한 경로이다.
④ 926, 87, 813, 318, 572, 826, 635, 598, 580, 629 → X, 813을 지나면서 상한이 813으로 정해졌는데 그 다음에 813보다 큰 826이 나와 상한 조건을 어긴다 → 이 위치에서 826이 아니라 813보다 작은 값이 나와야 한다.

**오답 포인트**

이런 문제에서 흔히 저지르는 실수는 인접한 두 값만 비교하며 커지고 작아지는 것만 확인하는 것이다. 예를 들어 572에서 826으로 가는 것 자체는 572<826이라 언뜻 문제없어 보이지만, 그 이전에 813이라는 노드를 지나며 상한이 이미 813으로 좁혀졌다는 사실을 놓치면 오류를 발견하지 못한다. 각 노드를 지날 때마다 지금까지 좁혀진 구간(하한과 상한)을 계속 갱신해서 그 범위 안에 있는지를 확인해야 한다.

NoteBase

### 다시 나오면 이렇게 판단

이진 탐색 트리 비교 순서 문제는 하한과 상한 두 변수만 유지하면서 나열된 값을 하나씩 검사하면 빠르게 풀린다. 첫 값에서 시작해 목표값과 비교할 때마다 구간을 좁히고, 다음 값이 그 구간 밖에 있으면 그 순서는 불가능하다고 바로 판단한다.

### 암기 포인트

이진 탐색 트리에서 유효한 탐색 경로는 각 노드를 지날 때마다 목표값을 포함하는 구간이 점점 좁아져야 하고, 이후에 나오는 모든 노드는 그 좁혀진 구간 안에 있어야 한다는 것을 암기해 둔다.

NoteBase

**문 6. 단순 연결 리스트(singly linked list)의 노드를 나타내는 구조체 node에 다음 노드를 가리키는 포인터 link가 있다고 하자. 2개의 노드로 구성된 단순 연결 리스트에서 첫 번째 노드와 두 번째 노드의 사이에 새로운 노드를 삽입하면서 리스트의 순서를 역순으로 변환하고자 한다. p에는 리스트의 시작 주소가 저장되어 있으며, new에는 새로운 노드의 주소가 저장되어 있다고 가정할 때, 명령문의 실행 순서로 옳은 것은? (단, 역순 변환 후 p에는 리스트의 새로운 시작 주소가 저장되어 있어야 한다)**

① `p→link = new; p = p→link; new→link = p; new→link→link = NULL;`
② `p = p→link; p→link = new; new→link = p; new→link→link = NULL;`
③ `new→link = p; p→link = new; p = p→link; new→link→link = NULL;`
④ `new→link = p; p = p→link; p→link = new; new→link→link = NULL;`

**정답 4번**

### 출제 포인트

노드 A(첫 번째, p가 가리킴)와 노드 B(두 번째, A→link가 가리킴) 사이에 새 노드 new를 끼워 넣으면서 동시에 리스트 전체 순서를 뒤집는 문장 순서를 고르는 문제이다. 포인터 대입은 한 줄씩 실행되는 대로 이전 줄의 결과값이 다음 줄에 그대로 쓰이므로, 대입 순서가 바뀌면 원래 주소값이 덮어써져 사라지거나 노드가 유실된다.

### 조건 정리

초기 상태: p는 A를 가리키고 A→link는 B를 가리킨다(A→B). new는 아직 어디에도 연결되지 않은 새 노드 N의 주소이다. 목표 상태: 변환 후 p는 새로운 시작 주소를 가리켜야 하며, 원래 A와 B 사이에 N이 끼워진 A→N→B를 통째로 뒤집은 B→N→A 순서가 되어야 한다(즉 최종 첫 노드=B, 중간=N, 마지막=A→link=NULL).

### 사용할 공식

각 대입문을 왼쪽부터 순서대로 실행하면서, 대입이 일어나는 그 순간의 값(직전 줄까지 반영된 값)을 대입식 오른쪽에 사용한다는 원칙만 적용하면 된다. new→link→link = NULL은 new→link가 가리키는 노드(즉 A)의 link를 NULL로 만드는 문장이다.

### 계산 과정

④ new→link = p; → N→link = A (N이 A를 가리킨다).
p = p→link; → p는 원래 A→link 값인 B로 바뀐다(p = B).
p→link = new; → 이제 p는 B이므로 B→link = N (B가 N을 가리킨다).
new→link→link = NULL; → new→link는 A이므로 A→link = NULL.
결과: p(=B) → N → A → NULL. 세 노드가 모두 살아 있고 p가 새 시작 주소 B를 정확히 가리키므로 올바른 삽입 겸 역순 변환이 완성된다.

NoteBase

### 선지 판단

① p→link = new;(A→link = N) → p = p→link;(p = A→link = N, 즉 p가 N이 되어 원래 B로 가는 경로를 잃음) → new→link = p;(N→link = N, 자기 자신을 가리키는 순환 발생) → 완전히 잘못된 구조가 되어 X.
② p = p→link;(p = A→link = B) → p→link = new;(B→link = N) → new→link = p;(N→link = p인데 이때 p는 이미 B이므로 N→link = B, N과 B가 서로를 가리키는 순환 발생) → new→link→link = NULL;(N→link→link = B→link = NULL, B→link가 다시 NULL로 덮어써져 N이 리스트에서 끊어짐) → N 노드가 유실되어 X.
③ new→link = p;(N→link = A) → p→link = new;(A→link = N, 이 순간 A→link에 저장돼 있던 B의 주소가 사라짐) → p = p→link;(p = A→link = N) → new→link→link = NULL;(N→link→link = A→link = NULL) → B로 가는 연결이 통째로 사라져 B 노드가 유실되어 X.
④ 계산 과정과 같이 A, N, B 세 노드가 모두 B→N→A→NULL 순서로 정상 연결되고 p가 새 시작 주소 B를 가리키므로 O.

### 오답 포인트

이 문제의 함정은 'p = p→link;'를 언제 실행하느냐이다. new→link = p;를 먼저 실행해 N이 A를 가리키게 고정한 뒤에 p를 B로 옮기고 그다음 A→link(=p→link)를 new로 덮어써야, A의 원래 다음 노드였던 B의 주소가 사라지기 전에 필요한 연결(N→link=A)을 미리 확보할 수 있다. ②나 ③처럼 p→link = new;를 A→link 값(B의 주소)을 다른 곳에 옮겨 두기 전에 실행하면 B로 가는 경로가 영구히 사라진다.

### 다시 나오면 이렇게 판단

연결 리스트 포인터 대입 순서 문제는 항상 '지금 이 줄을 실행하면 어떤 값이 영구히 덮어써지는가'를 한 줄씩 손으로 표에 적어 추적한다. 특히 이번 문항처럼 어떤 노드의 원래 link 값(B의 주소)이 유일한 통로일 때는, 그 값을 다른 포인터에 옮겨 적어 두기 전에 원래 자리를 덮어쓰는 선지는 바로 소거한다.

### 암기 포인트

삽입+역순처럼 여러 포인터를 동시에 재배치할 때는 '원래 다음 노드 주소부터 안전한 곳에 옮겨 담고, 그다음에 현재 위치의 link를 덮어쓴다'는 순서를 지켜야 노드가 유실되지 않는다는 원칙을 기억해 둔다.

NoteBase

**문 7. 다음 자료에서 첫 행은 오름차순으로 정렬하기 위해 주어진 입력 값이고 마지막 행은 정렬이 완료된 후의 결과이다. 중간의 행들은 합병정렬(merge sort), 삽입정렬(insertion sort), 선택정렬(selection sort), 퀵정렬(quick sort) 알고리즘을 통하여 얻을 수 있는 중간 과정을 보여준다. 각 행에 사용된 정렬 알고리즘을 바르게 나열한 것은? (단, 퀵정렬에서 피봇은 가장 오른쪽에 있는 원소를 사용한다)**

| 입력 | 21 | 41 | 15 | 25 | 21 | 37 | 45 | 9 | 23 | 18 | 36 | 4 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ㉠ | 9 | 15 | 21 | 21 | 25 | 37 | 41 | 45 | 23 | 18 | 36 | 4 | 11 |
| ㉡ | 4 | 9 | 11 | 15 | 18 | 37 | 45 | 41 | 23 | 21 | 36 | 21 | 25 |
| ㉢ | 15 | 21 | 21 | 25 | 37 | 41 | 45 | 9 | 18 | 23 | 4 | 11 | 36 |
| ㉣ | 4 | 9 | 11 | 25 | 21 | 37 | 45 | 41 | 23 | 18 | 36 | 21 | 15 |
| 결과 | 4 | 9 | 11 | 15 | 18 | 21 | 21 | 23 | 25 | 36 | 37 | 41 | 45 |

① ㉠: 삽입정렬, ㉡: 선택정렬, ㉢: 합병정렬, ㉣: 퀵정렬

② ㉠: 삽입정렬, ㉡: 퀵정렬, ㉢: 합병정렬, ㉣: 선택정렬

③ ㉠: 합병정렬, ㉡: 선택정렬, ㉢: 삽입정렬, ㉣: 퀵정렬

④ ㉠: 합병정렬, ㉡: 퀵정렬, ㉢: 선택정렬, ㉣: 삽입정렬

**정답 1번**

**출제 포인트**

같은 입력 자료를 합병정렬·삽입정렬·선택정렬·퀵정렬로 각각 진행시켰을 때 나타나는 중간 상태를 보여 주고, 어느 행이 어떤 알고리즘의 중간 과정인지 구분하는 문제이다. 각 정렬 알고리즘이 자료를 처리해 나가는 방식(어디부터 어디까지가 확정되는지, 확정되지 않은 부분의 순서가 원래 순서와 같은지 다른지)의 차이를 이용해서 판단한다.

NoteBase

**자료 읽는 순서**

먼저 각 행에서 앞부분이 오름차순으로 정렬되어 있는지, 그리고 아직 정렬되지 않은 뒷부분이 입력의 원래 순서를 그대로 유지하고 있는지를 확인한다. 삽입정렬은 앞쪽 k개가 정렬된 상태이고 나머지는 입력에 있던 순서 그대로 남는다. 선택정렬은 앞쪽 k개가 전체 원소 중 가장 작은 값들로 채워지지만, 그 값을 채우는 과정에서 서로 떨어진 위치의 원소끼리 자리를 맞바꾸므로 뒷부분의 순서가 입력 순서와 달라진다. 합병정렬은 배열을 절반씩 나누어 각각 정렬한 뒤 합치므로, 서로 다른 구간이 부분적으로 정렬된 블록 형태로 동시에 나타날 수 있다. 퀵정렬은 피봇을 기준으로 작은 값과 큰 값을 나누어 자리를 바꾸므로, 정렬이 끝나지 않은 구간에서도 원소들이 원래 위치와 다른 곳으로 옮겨져 있다.

㉠: 9 15 21 21 25 37 41 45 | 23 18 36 4 11
앞의 8개(9,15,21,21,25,37,41,45)는 입력의 앞 8개 값(21,41,15,25,21,37,45,9)을 그대로 오름차순으로 배열한 것이고, 뒤의 5개(23,18,36,4,11)는 입력의 뒤 5개 값과 순서까지 완전히 같다. 즉 앞에서부터 하나씩 값을 이미 정렬된 부분에 끼워 넣어 나가고 아직 처리되지 않은 뒷부분은 입력 그대로 남아 있는 전형적인 삽입정렬의 중간 상태이다.

㉡: 4 9 11 15 18 | 37 45 41 23 21 36 21 25
앞의 5개(4,9,11,15,18)는 전체 13개 값 중 가장 작은 다섯 값이 순서대로 채워진 것이다. 반면 뒤쪽은 21이 두 곳(23 뒤와 맨 끝)에 나뉘어 있는 등 입력 순서와 다르게 뒤섞여 있는데, 이는 최솟값을 찾을 때마다 앞자리와 맞바꾸는 선택정렬 특유의 흔적이다(가장 작은 값 4는 원래 뒤쪽에 있었으나 맨 앞의 21과 자리를 바꾸며 그 21이 뒤쪽 어딘가로 이동한 것과 같은 방식).

㉢: 15 21 21 25 37 41 45 | 9 18 23 4 11 36
앞의 7개(15,21,21,25,37,41,45)는 입력의 앞 6개 값(21,41,15,25,21,37)을 정렬한 뒤 다음 값 45까지 포함해 하나의 정렬된 블록으로 합쳐진 상태이고, 뒤쪽 6개(9,18,23,4,11,36)는 입력 뒤쪽 값들(9,23,18,36,4,11)과 구성 원소는 같지만 순서가 바뀌어 있다. 이는 뒤쪽 구간에서도 이미 일부 부분합병이 이루어졌기 때문으로, 좌우를 각각 절반씩 나누어 정렬한 뒤 합치는 합병정렬의 중간 상태에 해당한다.

㉣: 4 9 11 | 25 21 37 45 41 23 18 36 21 15
앞의 3개(4,9,11)는 전체에서 가장 작은 세 값인데, 입력에서 4와 11은 맨 뒤쪽, 9는 중간에 있던 값들이다. 피봇(가장 오른쪽 원소)을 기준으로 작은 값들을 왼쪽으로 모으는 분할 과정을 거치면 이렇게 서로 멀리 떨어져 있던 작은 값들이 한꺼번에 앞쪽으로 이동해 확정될 수 있으며, 나머지 구간도 원래 순서와 다르게 뒤섞여 있어 분할 교환의 흔적이 남아 있는 퀵정렬의 중간 상태로 볼 수 있다.

NoteBase

### 선지별 해설

① ㉠ 삽입정렬, ㉡ 선택정렬, ㉢ 합병정렬, ㉣ 퀵정렬 → O. 위에서 확인한 네 행의 특징과 정확히 일치한다.
② ㉠ 삽입정렬, ㉡ 퀵정렬, ㉢ 합병정렬, ㉣ 선택정렬 → X. ㉡의 앞 5개(4,9,11,15,18)는 전체 최솟값들이 차례로 자리를 잡은 선택정렬의 결과이지 퀵정렬의 분할 결과가 아니다.
③ ㉠ 합병정렬, ㉡ 선택정렬, ㉢ 삽입정렬, ㉣ 퀵정렬 → X. ㉠은 뒷부분이 입력 순서와 완전히 같은 채로 앞 8개만 정렬되어 있으므로 삽입정렬의 특징이며, 여러 구간이 동시에 부분 정렬되는 합병정렬의 특징과는 다르다.
④ ㉠ 합병정렬, ㉡ 퀵정렬, ㉢ 선택정렬, ㉣ 삽입정렬 → X. ㉠은 합병정렬이 아니라 삽입정렬의 특징을 보이고, ㉡도 퀵정렬이 아니라 선택정렬의 특징을 보인다.

### 오답 포인트

㉠에서 앞부분만 보고 '정렬된 블록이 있으니 합병정렬'이라고 성급히 판단하기 쉬운데, 뒷부분이 입력의 원래 순서와 한 치의 어긋남 없이 완전히 같다는 점까지 확인해야 삽입정렬임을 확신할 수 있다. ㉡에서도 앞부분에 최솟값들이 모여 있다는 것만 보고 퀵정렬로 오인하지 않도록, 뒷부분에서 값들이 서로 자리를 맞바꾼 흔적(교환)이 있는지를 함께 살펴야 한다.

### 다시 나오면 이렇게 판단

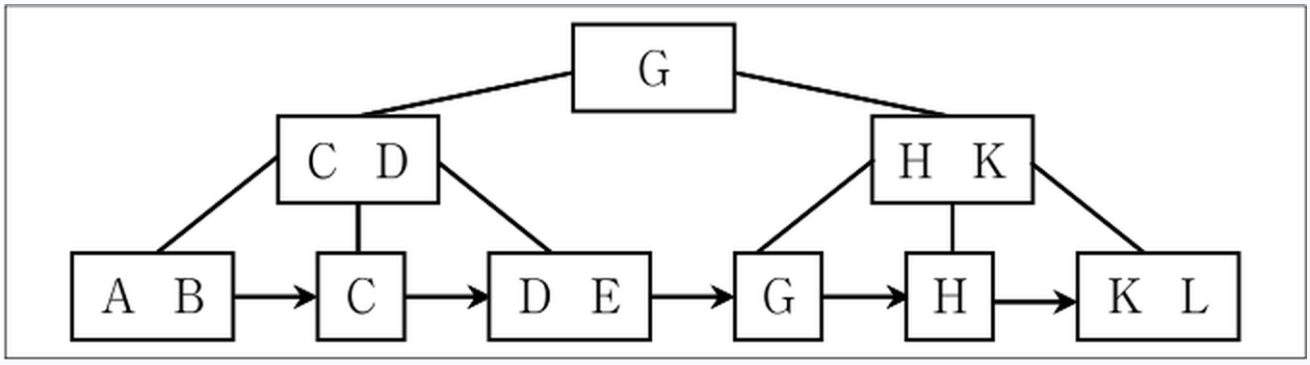
정렬 과정 추적 문제는 각 알고리즘의 '진행 중 불변식'을 기준으로 삼는다. 삽입정렬은 앞쪽 정렬 구간 뒤에 입력 순서 그대로인 미처리 구간이 붙어 있고, 선택정렬은 앞쪽에 전체 최솟값들이 채워지되 뒤쪽 순서가 교환으로 흐트러지며, 합병정렬은 배열 여러 구간에서 동시에 부분 정렬된 블록이 나타나고, 퀵정렬은 피봇 기준의 분할·교환으로 특정 구간의 순서가 통째로 뒤바뀐다. 이 네 가지 기준만 각 행에 대입해 보면 빠르게 구분할 수 있다.

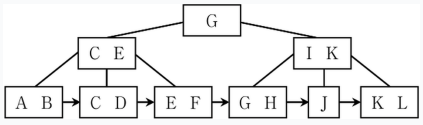
### 암기 포인트

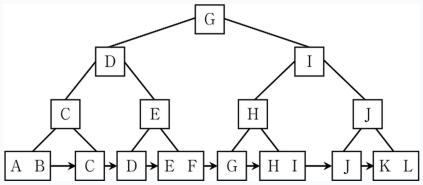
삽입정렬=앞은 정렬, 뒤는 입력 그대로. 선택정렬=앞은 최솟값들이 채워지고 뒤는 교환으로 뒤섞임.
합병정렬=여러 구간이 동시에 부분 정렬. 퀵정렬=피봇 기준 분할·교환으로 순서가 크게 뒤바뀜.

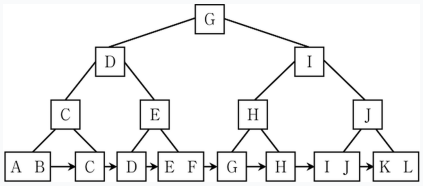
NoteBase

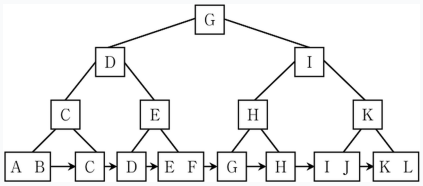
**문 8. 다음은 차수가 3인 B+ 트리(tree)의 초기 상태이다. 이 트리에 색인키 'F', 'I', 'J'를 순서대로 삽입한 후의 모습은? (단, 단말 노드는 최대 2개의 원소를 포함할 수 있으며, 단말 노드의 분할 시 중간 위치의 값은 오른쪽 노드에 포함되도록 한다)**




① 


② 


③ 


④ 


**정답 4번**

### 출제 포인트

차수가 3인 B+ 트리에서 단말 노드가 최대 2개의 원소만 가질 수 있다는 조건 아래, 키를 연속으로 삽입할 때 발생하는 단말 노드 분할과 그로 인한 상위(내부) 노드 분할이 연쇄적으로 일어나는 과정을 정확히 따라갈 수 있는지를 묻는다. 특히 단말 노드 분할 시에는 가운데 값이 오른쪽 노드로 복사되어 남지만, 내부 노드가 분할될 때는 가운데 값이 복사되지 않고 그대로 위로 올라가 사라진다는 점을 구분하는 것이 핵심이다.

NoteBase

**올바른 순서**

1) F 삽입: 루트 G에서 F는 G보다 작아 왼쪽 [C D]로, 다시 D 이상이므로 리프 [D E]로 이동해 [D E F]가 되어 3개로 넘친다. 규칙에 따라 가운데 값 E가 오른쪽에 포함되도록 [D] / [E F]로 나누고, E를 부모 [C D]에 올려 [C D E]가 되어 다시 넘친다. 내부 노드는 가운데 값이 그대로 위로 이동하므로 D가 루트로 올라가고 [C] / [E]로 나뉜다. 루트는 [D G]가 된다.
2) I 삽입: 루트 [D G]에서 I는 G 이상이므로 오른쪽 [H K]로, H 이상 K 미만이므로 리프 [H]로 이동해 [H I]가 되어 2개이므로 분할 없이 그대로 저장된다.
3) J 삽입: 루트 [D G]에서 J는 G 이상이므로 오른쪽 [H K]로, H 이상 K 미만이므로 리프 [H I]로 이동해 [H I J]가 되어 3개로 넘친다. 가운데 값 I가 오른쪽에 포함되도록 [H] / [I J]로 나누고, I를 부모 [H K]에 올려 [H I K]가 되어 다시 넘친다. 내부 노드 분할이므로 가운데 값 I가 그대로 위(루트)로 이동하고, 내부 노드는 [H] / [K]로 나뉜다. 이때 루트 [D G]에 I가 올라가 [D G I]가 되어 3개로 넘치므로, 다시 내부 노드 분할 규칙에 따라 가운데 값 G가 그대로 새 루트로 올라가고 기존 루트는 [D] / [I]로 나뉜다.
최종 결과: 새 루트는 G이며, 왼쪽 자식은 D(하위에 C, E를 두고 리프는 A B, C, D, E F), 오른쪽 자식은 I(하위에 H, K를 두고 리프는 G, H, I J, K L)이다.

**선지별 해설**

① X: 단말 노드 분할 규칙(가운데 값이 오른쪽 노드에 포함)과 내부 노드 분할 규칙(가운데 값이 위로 올라가며 복사되지 않음)을 지키지 않고 임의로 재구성한 형태로, 리프 구성(C D, E F 등)과 최상위 두 내부노드(C E, I K) 모두 위의 삽입 순서를 따를 때 나오지 않는다.
② X: 트리의 전체적인 골격(루트 G, 좌측 D, 우측 I)은 맞지만, I의 오른쪽 하위 내부 노드가 H와 J로 되어 있고 리프가 G, H I, J, K L로 구성되어 있다. 그러나 실제로는 J를 삽입할 때 리프 [H I]가 [H I J]로 넘쳐 [H]와 [I J]로 나뉘고 이때 위로 올라가는 값은 I이지 J가 아니므로, 내부 노드는 H, K가 되어야 하고 리프는 H, I J, K L이 되어야 한다. 따라서 오답이다.
③ X: 좌측 구조(D 아래 C, E; 리프 A B, C, D, E F)는 옳고 리프 구성(H, I J, K L)도 옳지만, I의 오른쪽 하위 내부 노드가 H와 J로 표시되어 있다. 리프 분할 시 위로 올라가는 값은 I이므로 내부 노드는 H와 K가 되어야 하는데 J로 잘못 표시되어 오답이다.
④ O: 좌측은 D 아래 C, E를 두고 리프가 A B, C, D, E F이며, 우측은 I 아래 H, K를 두고 리프가 G, H, I J, K L로 구성되어 위의 삽입 순서를 그대로 반영한다.

**오답 포인트**

가장 흔한 실수는 내부 노드가 분할될 때도 리프 노드처럼 가운데 값이 복사되어 남는다고 착각하는 것이다. 이 문제에서는 J를 삽입해 리프 [H I J]가 분할될 때 가운데 값 I가 상위 내부 노드로 올라가면서 원래 리프에는 남지 않고, 그 상위 내부 노드 [H K]에 I가 삽입되어 다시 넘치면 이번에는 내부 노드 분할이므로 가운데 값 I가 그대로 최상위로 이동하며 사라진다. 이 과정에서 내부 노드에 J가 남는다고 착각하면 ②나 ③처럼 H, J 구조를 답으로 고르게 된다.

NoteBase

### 다시 나오면 이렇게 판단

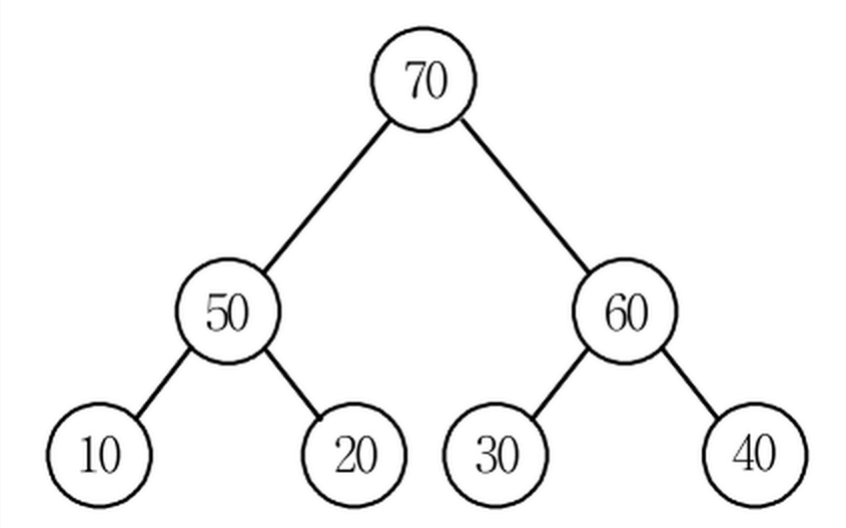
B+ 트리 삽입 문제가 나오면 먼저 단말 노드 분할 규칙(가운데 값의 처리 방식: 복사되어 남는지 여부)과 내부 노드 분할 규칙을 문제의 단서에서 각각 확인한다. 이 문제처럼 단말 노드는 가운데 값이 오른쪽에 포함(복사)되고, 내부 노드는 가운데 값이 복사 없이 그대로 위로 올라가는 표준 B+ 트리 규칙을 적용해 삽입을 하나씩 순서대로 시뮬레이션한 뒤, 선지의 상위 내부 노드에 어떤 값이 남아 있는지를 마지막에 반드시 대조한다.

### 암기 포인트

B+ 트리: 리프 노드 분할 시 가운데 값은 오른쪽 리프에 포함되며 부모로도 복사되어 올라간다(리프에는 남는다). 내부 노드 분할 시 가운데 값은 복사 없이 그대로 위로 이동하며 그 노드에는 남지 않는다. 연쇄 분할이 루트까지 전파되면 트리의 높이가 한 단계 증가한다.

NoteBase

**문 9. 그림과 같은 최대 힙(max heap)을 구성하기 위한 삽입 순서로 옳은 것은?**




① 50, 30, 40, 70, 60, 10, 20

② 10, 60, 50, 70, 20, 30, 40

③ 10, 20, 40, 30, 60, 50, 70

④ 50, 60, 30, 10, 20, 70, 40

**정답 4번**

**출제 포인트**

최대 힙(max heap)에 원소를 하나씩 삽입할 때는 완전이진트리의 맨 끝(배열의 마지막 위치)에 새 원소를 추가한 뒤, 그 원소를 부모 노드와 비교하여 부모보다 크면 위치를 맞바꾸는 상향 이동(sift-up, up-heap)을 부모보다 작거나 루트에 도달할 때까지 반복한다. 이 문제는 그림에 주어진 완전한 힙 배열(레벨 순서: 70, 50, 60, 10, 20, 30, 40)이 어떤 삽입 순서를 그대로 시뮬레이션했을 때 나오는지를 확인하는 문제이다. 배열 위치를 1번부터 번호를 매기면 각 위치 i의 부모는 i/2(정수 나눗셈)번 위치이다.

NoteBase

**올바른 순서**

④의 순서(50, 60, 30, 10, 20, 70, 40)로 하나씩 삽입하며 상향 이동을 적용하면 다음과 같다.
1) 50 삽입: [50]
2) 60 삽입: 끝에 추가하면 [50, 60], 60이 부모 50보다 크므로 교환 → [60, 50]
3) 30 삽입: [60, 50, 30], 부모(60)보다 작으므로 그대로
4) 10 삽입: [60, 50, 30, 10], 부모(50)보다 작으므로 그대로
5) 20 삽입: [60, 50, 30, 10, 20], 부모(50)보다 작으므로 그대로
6) 70 삽입: 끝에 추가하면 [60, 50, 30, 10, 20, 70], 부모(30)보다 크므로 교환 → [60, 50, 70, 10, 20, 30], 다시 부모(60)보다 크므로 교환 → [70, 50, 60, 10, 20, 30]
7) 40 삽입: [70, 50, 60, 10, 20, 30, 40], 부모(60)보다 작으므로 그대로
최종 배열이 70, 50, 60, 10, 20, 30, 40이 되어 그림의 힙과 정확히 일치한다.

**선지별 해설**

① 50, 30, 40, 70, 60, 10, 20 → X. 끝까지 시뮬레이션하면 70, 60, 40, 30, 50, 10, 20이 되어 2번째 위치가 60, 3번째 위치가 40이 되므로 그림의 50, 60과 다르다.
② 10, 60, 50, 70, 20, 30, 40 → X. 시뮬레이션하면 70, 60, 50, 10, 20, 30, 40이 되어 2번째와 3번째 자리의 값이 60과 50으로, 그림의 50과 60이 서로 뒤바뀐 형태가 된다.
③ 10, 20, 40, 30, 60, 50, 70 → X. 시뮬레이션하면 70, 40, 60, 10, 30, 20, 50이 되어 그림의 배열(70, 50, 60, 10, 20, 30, 40)과 여러 위치에서 값이 다르게 나온다.
④ 50, 60, 30, 10, 20, 70, 40 → O. 위 '올바른 순서'와 같이 단계별로 상향 이동을 적용하면 70, 50, 60, 10, 20, 30, 40이 그대로 만들어져 그림의 힙과 일치한다.

**오답 포인트**

단순히 최종 힙에 들어있는 숫자들이 선지에 모두 포함되어 있는지만 확인하고 순서를 확인하지 않으면 오답을 고르기 쉽다. 힙 삽입은 항상 '끝에 추가 후 부모와 비교하여 교환'이라는 규칙을 순서대로 그대로 따라가며 배열을 직접 만들어 봐야 하고, 중간에 교환이 일어나는 시점(부모보다 클 때만 위로 올라간다는 점)을 놓치면 위치가 뒤바뀐 오답(①, ②)이나 완전히 다른 배열(③)을 만들게 된다.

**다시 나오면 이렇게 판단**

완성된 힙 배열이 주어지고 삽입 순서를 찾는 문제는 각 선지를 처음부터 끝까지 직접 삽입 시뮬레이션(끝에 추가 → 부모와 비교 → 크면 교환, 부모에 도달할 때까지 반복)해서 결과 배열이 그림과 완전히 같은지 하나씩 대조한다. 배열 위치 번호를 1부터 매기고 i번째의 부모는 i/2(정수 나눗셈)라는 공식을 적어두면 교환 여부를 빠르게 판단할 수 있다.

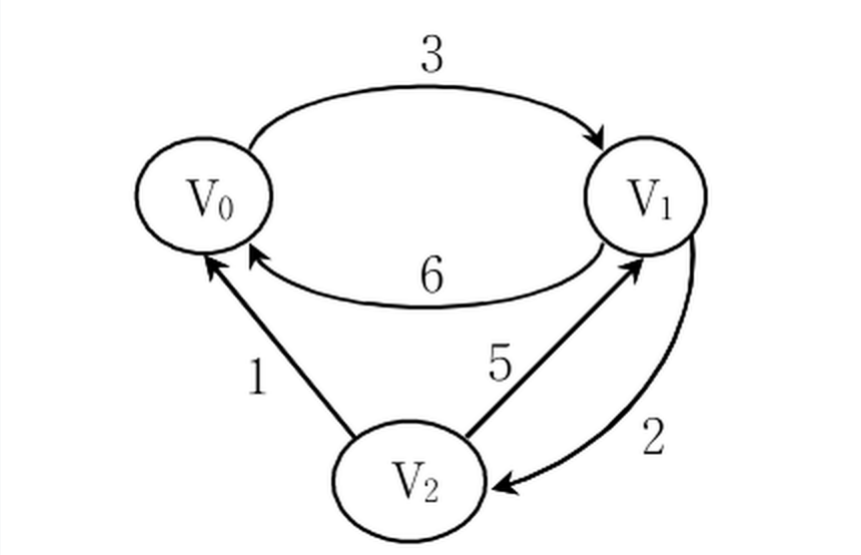
NoteBase

**암기 포인트**

최대 힙 삽입 규칙: 새 원소는 완전이진트리의 마지막 위치에 추가한 뒤, 부모 노드와 비교하여 자신이 더 크면 교환하고, 이를 부모보다 작아지거나 루트에 도달할 때까지 반복한다(상향 이동, sift-up). 배열로 표현할 때 i번째 노드의 부모는 i/2번째(정수 나눗셈) 위치이다.

NoteBase

**문 10. 다음 그래프에 대한 모든 쌍의 최단 경로를 구하기 위해 최단 거리 행렬 $D^{-1}$, $D^0$, $D^1$, $D^2$를 차례대로 구한다고 했을 때, $D^1$의 계산 결과는?**




①

| | [0] | [1] | [2] |
|---|---|---|---|
| [0] | 0 | 3 | ∞ |
| [1] | 6 | 0 | 2 |
| [2] | 1 | 4 | 0 |

②

| | [0] | [1] | [2] |
|---|---|---|---|
| [0] | 0 | 3 | 5 |
| [1] | 6 | 0 | 2 |
| [2] | 1 | 4 | 0 |

③

| | [0] | [1] | [2] |
|---|---|---|---|
| [0] | 0 | 3 | 5 |
| [1] | 3 | 0 | 2 |
| [2] | 1 | 4 | 0 |

NoteBase

④

| | [0] | [1] | [2] |
|---|---|---|---|
| [0] | 0 | 3 | ∞ |
| [1] | 6 | 0 | 2 |
| [2] | 1 | 5 | 0 |

**정답 2번**

### 출제 포인트

플로이드-워셜(Floyd-Warshall) 알고리즘으로 모든 쌍 최단 경로를 구하는 과정에서, 중간 경유 정점을 하나씩 추가해 갈 때 거리 행렬이 어떻게 갱신되는지를 정확히 계산할 수 있는지를 묻는다. D^{-1}은 경유 정점을 전혀 허용하지 않은 초기 인접 행렬이고, D^{k}는 정점 0부터 k까지를 경유지로 허용했을 때의 최단 거리 행렬이다.

### 조건 정리

그래프의 간선은 V0→V1=3, V1→V0=6, V2→V0=1, V2→V1=5, V1→V2=2이다. 정점이 3개(V0, V1, V2)이므로 행렬의 행과 열은 [0],[1],[2] 순서로 구성된다. 대각선(자기 자신으로의 거리)은 항상 0이고, 직접 연결되지 않은 칸은 무한대(∞)로 시작한다.

### 사용할 공식

$D^k[i][j] = \min(D^{k-1}[i][j], D^{k-1}[i][k] + D^{k-1}[k][j])$ 이다. 즉 정점 $k$를 새로 경유지로 허용했을 때, 기존 최단 거리와 $i \to k \to j$로 우회하는 거리 중 더 작은 값을 취한다.

### 계산 과정

1단계, D^{-1}(초기 행렬)을 세운다. [0]행: 0,3,∞ / [1]행: 6,0,2 / [2]행: 1,5,0.
2단계, D^{0}은 정점 0을 경유지로 허용해 갱신한다. D^{0}[1][2]=min(2, D^{-1}[1][0]+D^{-1}[0][2])=min(2,6+∞)=2로 변화 없음. D^{0}[2][1]=min(5, D^{-1}[2][0]+D^{-1}[0][1])=min(5,1+3)=4로 갱신된다. 나머지는 변화 없다. 따라서 D^{0}은 [0]행: 0,3,∞ / [1]행: 6,0,2 / [2]행: 1,4,0.
3단계, D^{1}은 정점 1을 경유지로 허용해 갱신한다. D^{1}[0][2]=min(D^{0}[0][2], D^{0}[0][1]+D^{0}[1][2])=min(∞, 3+2)=5로 갱신된다. D^{1}[2][0]=min(D^{0}[2][0], D^{0}[2][1]+D^{0}[1][0])=min(1, 4+6)=1로 변화 없음. 나머지 칸도 정점 1을 경유해도 더 짧아지지 않으므로 그대로 유지된다.
따라서 D^{1}은 [0]행: 0,3,5 / [1]행: 6,0,2 / [2]행: 1,4,0 이다.

NoteBase

### 선지 판단

① [0][2]가 여전히 ∞로 남아 있다. → 정점 1을 경유지로 허용하면 V0→V1→V2 경로(3+2=5)로 D^{1}[0][2]가 5로 줄어들어야 하는데 이를 반영하지 않은 D^{0} 단계의 값이므로 오답이다.
② [0]행: 0,3,5 / [1]행: 6,0,2 / [2]행: 1,4,0으로 계산 과정에서 도출한 D^{1}과 정확히 일치하므로 정답이다.
③ [1][0]이 3으로 되어 있다. → V1→V0의 실제 간선 가중치는 6이고, 정점 1을 경유해도(즉 자기 자신을 거쳐도) 더 짧아질 근거가 없으므로 6이 유지되어야 하는데 3으로 잘못 바뀐 값이라 오답이다.
④ [2][1]이 5로 되어 있다. → D^{0} 단계에서 이미 정점 0을 경유해 1+3=4로 갱신되었어야 할 값이 초기값 5 그대로 남아 있으므로 오답이다.

### 오답 포인트

가장 흔한 실수는 D^{0}에서 이미 갱신되어야 할 D^{0}[2][1](정점 0을 경유해 4로 줄어드는 값)을 초기값 5로 방치한 채 D^{1}을 계산하는 것이다(④의 함정). 또한 D^{1}에서 정점 1을 경유지로 새로 허용할 때 [0][2] 칸만 갱신 대상이라는 점을 놓치고 여전히 ∞로 두는 실수(①의 함정), 반대로 갱신 대상이 아닌 [1][0] 칸까지 임의로 바꾸는 실수(③의 함정)에 주의해야 한다.

### 다시 나오면 이렇게 판단

플로이드-워셜 문제는 D^{k-1}에서 D^{k}로 넘어갈 때 딱 한 개의 정점(k)만 새로 경유지로 허용된다는 점을 기억하고, 갱신 대상 칸은 오직 $D^{k-1}[i][k] + D^{k-1}[k][j]$가 기존 값보다 작아지는 칸뿐임을 행마다 직접 손으로 계산해 확인한다. 초기 인접 행렬(D^{-1})을 정확히 세우는 것이 전체 계산의 출발점이므로 방향(누가 누구로 가는 간선인지)을 먼저 표시하고 시작한다.

### 암기 포인트

$D^{k}[i][j] = \min(D^{k-1}[i][j], D^{k-1}[i][k] + D^{k-1}[k][j])$를 정점 개수만큼(0부터 n-1까지) 반복 적용하는 것이 플로이드-워셜의 핵심이며, 매 단계마다 경유지로 새로 허용되는 정점은 단 하나뿐이라는 점을 기억해 둔다.

NoteBase

**문 11. 다음은 연결 스택(linked stack)에서 새로운 노드를 삽입하는 C 언어 프로그램이다. ㄱ, ㄴ에 들어갈 내용으로 옳게 짝지은 것은? (단, top은 스택의 첫 번째 노드를 가리키는 포인터이며, top이 NULL이면 공백 스택이다)**

```
typedef struct node {
  int key;
  struct node *link;
} StackNode;

StackNode *top = NULL;

void push(int data) {
  StackNode *tmp = (struct node*)malloc(sizeof(StackNode));
  if (tmp == NULL) {
    printf("Memory allocation is failed.\n");
    exit(1);
  }
  tmp→key = data;
  if (top == NULL) {
    tmp→link = NULL;
    top = tmp;
  }
  else {
      ㉠ ;
      ㉡ ;
  }
}
```

| | ㉠ | ㉡ |
|---|---|---|
| ① | tmp→link = top | top = tmp |
| ② | top→link = tmp | tmp = top→link |
| ③ | top→link = tmp | tmp = top |
| ④ | tmp→link = top | top = tmp→link |

**정답 1번**

### 출제 포인트

연결 스택(linked stack)에서 이미 노드가 하나 이상 있는 상태(top != NULL)에 새 노드 tmp를 push할 때, 두 문장의 순서와 내용이 올바른지 묻는 문제이다. 연결 리스트의 맨 앞에 노드를 끼워 넣는 삽입 로직의 순서를 이해하고 있는지가 핵심이다.

NoteBase

### 올바른 순서

새 노드를 스택의 맨 앞(top 자리)에 끼워 넣으려면 반드시 다음 순서를 지켜야 한다.
1단계: 새 노드 tmp가 기존의 top이 가리키던 노드를 가리키도록 연결한다. 즉 tmp→link = top; 이 문장이 먼저 실행되어야 tmp가 기존 스택 전체와 연결된다.
2단계: 그 다음에 top이 새 노드 tmp를 가리키도록 갱신한다. 즉 top = tmp; 이 문장이 나중에 실행되어야 한다.
만약 순서를 바꾸어 top = tmp; 를 먼저 실행하면, top이 이미 tmp로 바뀌어 버린 뒤에는 원래 top이 가리키던 기존 스택의 첫 노드 주소를 알 수 없게 되어 tmp→link = top; 을 실행해도 tmp가 자기 자신을 가리키는 결과가 되고, 기존 노드들과의 연결이 끊어져 버린다. 따라서 ㄱ에는 tmp→link = top, ㄴ에는 top = tmp가 들어가야 한다.

### 선지별 해설

① ㄱ: tmp→link = top, ㄴ: top = tmp — O. 새 노드가 기존 top 노드를 가리키도록 먼저 연결한 뒤 top을 새 노드로 갱신하는 순서와 내용이 모두 올바르다.
② ㄱ: top→link = tmp, ㄴ: tmp = top→link — X. top→link = tmp는 기존 top 노드의 link 필드를 tmp로 바꾸는 문장인데, top 노드는 원래 다음 노드가 없는 스택의 첫 노드이거나 이미 다음 노드를 link로 갖고 있으므로 이 필드를 tmp로 덮어쓰면 그 뒤에 연결되어 있던 나머지 노드들과의 연결이 끊어진다. 또한 tmp = top→link는 tmp가 가리키던 새로 할당한 노드를 포기하고 top의 다음 노드를 가리키게 하는 것이어서, 애써 malloc한 새 노드가 스택 어디에도 연결되지 않고 유실된다.
③ ㄱ: top→link = tmp, ㄴ: tmp = top — X. ②와 마찬가지로 top→link = tmp가 기존 연결 구조를 깨뜨리며, tmp = top은 tmp가 top과 같은 노드를 가리키게 만들 뿐 새로 할당한 노드는 여전히 스택에 연결되지 않은 채 남아 데이터 삽입이 이루어지지 않는다.
④ ㄱ: tmp→link = top, ㄴ: top = tmp→link — X. ㄱ은 올바르게 새 노드를 기존 top에 연결하지만, ㄴ에서 top = tmp→link로 하면 top이 tmp가 아니라 tmp가 가리키는 노드(즉 원래의 top이었던 노드)를 다시 가리키게 되어, 결과적으로 top이 이전과 같은 노드를 가리키는 셈이 된다. 이렇게 되면 새로 삽입한 tmp 노드는 연결은 되어 있지만 top이 그 노드를 가리키지 않아 스택의 첫 노드로 인식되지 못한다.

### 오답 포인트

이 문제에서 가장 흔한 실수는 tmp→link = top과 top = tmp의 실행 순서를 뒤바꾸어 생각하는 것이다.
포인터를 다른 값으로 갱신하면 그 이전 값은 다시 되돌릴 수 없으므로, top을 tmp로 바꾸기 전에 반드시 tmp가 기존 top을 먼저 가리키도록 연결해야 한다는 점을 놓치기 쉽다. 또한 top→link = tmp처럼 top 노드 자체의 link를 건드리는 것과 tmp→link = top처럼 새 노드의 link를 건드리는 것을 혼동하지 않아야 한다.

### 다시 나오면 이렇게 판단

연결 리스트류 자료구조(스택, 큐, 리스트) 맨 앞에 새 노드를 삽입하는 코드가 나오면, 항상 '새 노드가 기존 구조를 먼저 가리키게 한 뒤 → 헤드/포인터가 새 노드를 가리키게 한다'는 두 단계 순서를 떠올린다. 순서가 하나라도 바뀌면 기존 연결이 끊기거나 새 노드가 유실된다는 점을 확인하며 판단한다.

NoteBase

**암기 포인트**

맨 앞 삽입은 '연결 먼저, 갱신 나중': tmp→link = top; → top = tmp; 순서를 통째로 외워 둔다.

NoteBase

**문 12. 다음 키(key)를 갖는 데이터들을 공백 AVL 트리에 차례대로 저장하였다. 생성된 AVL 트리를 T라고 할 때, 다음 설명 중에서 옳은 것은?**

12, 14, 9, 16, 19, 11

① T에 키가 25인 노드를 삽입하면 루트 노드가 바뀌게 된다.

② T에서 키가 9인 노드를 삭제하면 T의 높이가 1 감소한다.

③ T에 키가 7인 노드를 삽입하면 T의 높이가 1 증가한다.

④ 키가 9인 노드와 키가 16인 노드는 형제 노드이다.

**정답 4번**

### 출제 포인트

빈 AVL 트리에 12, 14, 9, 16, 19, 11을 순서대로 삽입했을 때 완성되는 트리의 실제 모양을 정확히 구성할 수 있는지, 그리고 그 트리에서 삽입·삭제·형제 관계를 올바르게 판단할 수 있는지를 묻는다.

### 조건 정리

AVL 트리는 모든 노드에서 왼쪽 서브트리 높이와 오른쪽 서브트리 높이의 차이(균형 인수)가 -1, 0, 1 중 하나여야 한다. 삽입 순서대로 이진 탐색 트리 규칙에 따라 키를 넣되, 어떤 노드의 균형 인수가 -2 또는 2가 되면 회전으로 재조정한다. 높이는 리프 노드를 0으로 두고 계산한다.

### 사용할 공식

균형 인수 $BF(v) = h(\text{left}(v)) - h(\text{right}(v))$이며 $|BF(v)| \le 1$을 항상 만족해야 한다. 삽입 위치의 조부모 노드에서 불균형(오른쪽-오른쪽, 왼쪽-왼쪽, 왼쪽-오른쪽, 오른쪽-왼쪽)이 발생하면 해당 유형에 맞는 단일 또는 이중 회전을 적용한다.

NoteBase

**계산 과정**

12를 삽입하면 12가 루트가 된다.
14를 삽입하면 12보다 커서 오른쪽 자식이 된다. 균형 인수 -1로 정상이다.
9를 삽입하면 12보다 작아서 왼쪽 자식이 된다. 균형 인수 0으로 정상이다. 이 시점 트리는 12(좌9, 우14)이다.
16을 삽입하면 12보다 크고 14보다 커서 14의 오른쪽 자식이 된다. 12의 균형 인수는 왼쪽 높이 1, 오른쪽 높이 2로 -1이라 회전이 필요 없다. 트리는 12(좌9, 우14(우16))이다.
19를 삽입하면 12, 14, 16을 거쳐 16의 오른쪽 자식이 되려 하는데, 이때 14의 균형 인수가 왼쪽 높이 0, 오른쪽 높이 2로 -2가 되어 불균형이 발생한다. 새 키가 불균형 노드(14)의 오른쪽 자식(16)의 오른쪽 서브트리에 들어가는 오른쪽-오른쪽 형태이므로 14를 축으로 왼쪽 회전을 수행한다. 그 결과 16이 새로운 부모가 되고 14와 19가 각각 16의 왼쪽, 오른쪽 자식이 된다. 트리는 12(좌9, 우16(좌14, 우19))가 된다.
11을 삽입하면 12보다 작아 9쪽으로, 9보다 커서 9의 오른쪽 자식이 된다. 9의 균형 인수는 0-1로 -1이라 정상이고, 12의 균형 인수도 왼쪽 높이 1(9-11), 오른쪽 높이 1(16-14,19)로 0이라 회전이 필요 없다.
최종 트리 T는 다음과 같다. 루트 12, 왼쪽 자식 9(오른쪽 자식으로 11을 둠, 왼쪽 자식 없음), 오른쪽 자식 16(왼쪽 자식 14, 오른쪽 자식 19)이다.

**선지 판단**

① T에 키 25를 삽입하면 25는 12, 16, 19를 거쳐 19의 오른쪽 자식이 된다. 19의 균형 인수는 0-1로 -1, 16의 균형 인수는 0-1로 -1, 12의 균형 인수는 좌측(9-11) 높이 1, 우측(16-14-19-25) 높이 2로 -1이 되어 모두 허용 범위 안이라 회전이 일어나지 않는다. 따라서 루트는 그대로 12이다 → 옳지 않은 설명이다.
② T에서 키 9를 삭제하면 9는 자식이 오른쪽의 11 하나뿐이므로 11이 그 자리로 올라와 12의 왼쪽 자식이 된다. 삭제 전 T의 높이는 12→9→11 경로가 2단계로 가장 길어 2였는데, 삭제 후에도 12→16→19(또는 14) 경로가 그대로 2단계로 남아 있어 트리의 높이는 여전히 2로 감소하지 않는다 → 옳지 않은 설명이다.
③ T에 키 7을 삽입하면 7은 12, 9를 거쳐 9의 왼쪽 자식이 된다. 9는 원래 오른쪽 자식 11만 있었는데 왼쪽에 7이 생겨 균형 인수가 0-0으로 오히려 더 균형 잡히고, 9를 루트로 하는 서브트리의 높이는 삽입 전후 모두 1로 변화가 없다. 따라서 전체 트리 T의 높이도 2로 그대로 유지된다 → 옳지 않은 설명이다.
④ 계산 과정에서 구성한 최종 트리 T를 보면 키 9와 키 16은 모두 루트 12의 자식으로, 같은 부모를 공유하는 형제 노드이다 → 옳은 설명이다.

**오답 포인트**

①과 ③은 삽입 후에도 해당 노드의 균형 인수가 -1에서 1 사이에 머무르면 회전이 일어나지 않는다는 점을 놓치기 쉽다. 균형 인수가 -2 또는 2가 되어야만 회전이 일어나고 트리 모양이나 높이가 바뀐다는 것을 확인해야 한다. ②는 삭제되는 노드의 형제 쪽 경로 길이를 따로 확인하지 않고 삭제된 쪽만 보고 높이가 줄었다고 성급히 판단하는 오류이다. 트리의 높이는 가장 긴 경로 기준이므로 다른 쪽 경로가 그대로면 높이는 유지된다.

NoteBase

### 다시 나오면 이렇게 판단

AVL 트리 문제는 반드시 삽입 순서대로 하나씩 이진 탐색 트리 규칙으로 배치한 뒤, 매 삽입마다 조상 노드들의 균형 인수를 확인해 -2 또는 2가 되는 지점을 찾아 그 유형(LL, RR, LR, RL)에 맞는 회전을 적용해 최종 트리를 그려 두어야 한다. 이후 각 선지는 그 최종 트리 그림 위에서 직접 삽입·삭제를 시뮬레이션해 높이 변화와 부모-자식 관계를 눈으로 확인하며 판단한다.

### 암기 포인트

균형 인수는 왼쪽 높이 빼기 오른쪽 높이이며 항상 -1, 0, 1 중 하나여야 한다. 균형 인수가 2 또는 -2가 될 때만 회전이 발생하고, 그 전까지는 삽입·삭제가 있어도 트리의 높이나 루트가 바뀌지 않을 수 있다. 트리의 높이는 가장 긴 경로 하나로 결정되므로 한쪽 경로만 보고 판단하면 안 된다.

NoteBase

**문 13. 다음 C 언어 함수의 시간 복잡도를 빅오(O) 표기법으로 표현한 것은? (단, n > 1)**

```
void testing(int n)
{
  int i, j, sum = 0;
  for (i = 0; i < n; i++)
    for (j = n; j > 1; j /= 2)
      sum += 1;
}
```

① O(n)

② $O(n^2)$

③ O(nlogn)

④ O(logn)

**정답 3번**

**출제 포인트**

이중 반복문에서 바깥 루프의 반복 횟수와 안쪽 루프의 반복 횟수를 각각 구한 뒤 곱해서 전체 시간 복잡도를 구하는 문제이다. 안쪽 루프처럼 변수를 매번 절반으로 줄여 가는 구조는 로그 형태의 반복 횟수를 만든다는 점을 파악하는 것이 핵심이다.

**조건 정리**

바깥 for문은 $i = 0$부터 $i < n$까지 1씩 증가하므로 총 $n$번 반복된다. 안쪽 for문은 $j = n$에서 시작하여 $j > 1$인 동안 $j$를 2로 나누는 연산(j /= 2)을 반복하므로, $j$가 $n$에서 시작해 1보다 커지지 않을 때까지 절반씩 줄어드는 횟수만큼 실행된다.

**사용할 공식**

값을 2로 계속 나누어 1 이하가 될 때까지 걸리는 반복 횟수는 $\log_2 n$에 비례한다. 즉 안쪽 루프의 반복 횟수는 $O(\log n)$이다. 바깥 루프가 $n$번 반복될 때마다 안쪽 루프가 매번 독립적으로 $O(\log n)$번 실행되므로 전체 반복 횟수는 $O(n) \times O(\log n) = O(n \log n)$이다.

NoteBase

### 계산 과정

안쪽 루프 한 번 실행 시 $j$의 값 변화는 $n \to n/2 \to n/4 \to \cdots \to 1$이 되므로 반복 횟수는 $k$를 $n/2^k \le 1$을 만족하는 최소 정수라 하면 $k = \log_2 n$이다. 이 안쪽 루프는 바깥 루프가 도는 $i = 0, 1, \ldots, n-1$의 매 단계마다 똑같이 $\log_2 n$번씩 반복되며, 안쪽 루프의 시작값 $j = n$이 바깥 루프의 $i$ 값과 무관하게 매번 고정되어 있으므로 총 sum 증가 횟수는 $n \times \log_2 n$번이다. 따라서 전체 시간 복잡도는 $O(n \log n)$이다.

### 선지 판단

① O(n) — 틀렸다. 바깥 루프만 고려한 값이며 안쪽 루프의 로그 반복을 반영하지 못한 표현이다.
② O(n^2) — 틀렸다. 안쪽 루프가 매번 $n$번이 아니라 $\log_2 n$번만 반복되므로 $n^2$이 아니라 $n \log n$이 되어야 한다.
③ O(nlogn) — 옳다. 바깥 루프 $n$회 × 안쪽 루프 $\log_2 n$회 = $n \log n$으로 전체 시간 복잡도와 정확히 일치한다.
④ O(logn) — 틀렸다. 안쪽 루프만의 반복 횟수이며 바깥 루프가 $n$번 도는 것을 반영하지 않은 표현이다.

### 오답 포인트

안쪽 루프의 종료 조건이 $j > 1$이고 $j$를 1씩 감소시키는 것이 아니라 2로 나누는 것이라는 점을 놓치면 안쪽 루프도 $n$번 도는 것으로 착각하여 $O(n^2)$을 고르게 된다. 반대로 안쪽 루프가 로그라는 점만 보고 바깥 루프의 존재를 무시하면 $O(\log n)$을 고르는 실수를 하게 된다. 두 루프가 서로 독립적으로 중첩되어 있으므로 각 루프의 반복 횟수를 곱해야 한다는 점을 항상 확인해야 한다.

### 다시 나오면 이렇게 판단

반복문의 증감식을 먼저 확인하여 $i++$처럼 1씩 증가·감소하면 반복 횟수가 $n$에 비례하고, j /= 2나 j *= 2처럼 값이 배수로 변하면 반복 횟수가 $\log n$에 비례한다는 것을 기준으로 삼는다. 중첩 루프의 전체 복잡도는 두 반복 횟수를 곱한 값이라는 점을 기계적으로 적용하면 빠르게 답을 구할 수 있다.

### 암기 포인트

변수가 1씩 증가·감소하는 루프는 $O(n)$, 변수가 2로 나누거나 곱해지는 루프는 $O(\log n)$이다. 이 둘이 중첩되면 $O(n \log n)$이 된다는 것을 대표 예시로 기억해 둔다.

NoteBase

**문 14. 다음은 원형 연결 리스트(circular linked list)의 길이를 구하는 C 언어 프로그램이다. ㉠ ~ ㉢에 들어갈 내용으로 옳게 짝지은 것은?**

```
typedef struct node {
  int key;
  struct node *link;
} list_node;
int length(list_node *ptr) {
  list_node *temp;
  int count = 0;
  if ( [ ㉠ ] ) {
    temp = ptr;
    do {
      count++;
      [ ㉡ ];
    } while ( [ ㉢ ] );
  }
  return count;
}
```

① ㉠: ptr != NULL, ㉡: temp = temp->link, ㉢: temp != ptr

② ㉠: ptr == NULL, ㉡: temp = temp->link, ㉢: temp == ptr

③ ㉠: ptr != NULL, ㉡: ptr = temp->link, ㉢: temp != ptr

④ ㉠: ptr == NULL, ㉡: ptr = temp->link, ㉢: temp == ptr

**정답 1번**

**출제 포인트**

원형 연결 리스트(circular linked list)에서 노드 개수를 세는 length 함수의 빈칸을 채우는 문제이다. 원형 리스트는 마지막 노드의 link가 다시 첫 노드(ptr)를 가리키므로, 순회를 멈추는 조건을 어떻게 세우는지가 핵심이다. 빈칸 ㉠은 리스트가 비어 있지 않은지 확인하는 조건, ㉡은 반복문 안에서 다음 노드로 이동하는 문장, ㉢은 do-while 반복을 멈추는 조건이다.

NoteBase

### 올바른 순서

① if(ptr != NULL) — 먼저 리스트가 빈 리스트(ptr이 NULL)가 아닌지 확인한다. 빈 리스트이면 순회 자체를 하지 않아야 하므로 count는 0으로 유지되고 그대로 반환된다.
② temp = ptr — 순회를 시작하기 전에 temp를 리스트의 시작 노드로 설정한다(문제에 이미 주어짐).
③ do { count++; temp = temp→link; } — do-while이므로 먼저 count를 1 증가시켜 현재 노드를 센 다음, temp를 다음 노드(temp→link)로 이동한다. 여기서 temp에 대입해야 순회 포인터가 실제로 앞으로 나아간다.
④ while(temp != ptr) — temp가 다시 시작 노드 ptr로 돌아올 때까지 반복한다. temp가 ptr과 같아지는 순간(한 바퀴를 다 돈 순간) 조건이 거짓이 되어 반복을 멈춘다.

### 선지별 해설

① ptr != NULL / temp = temp→link / temp != ptr — 모두 옳다. 빈 리스트가 아님을 확인(㉠)하고, do-while 안에서 temp를 다음 노드로 이동시키며(㉡) count를 누적하고, temp가 시작점 ptr로 돌아오기 전까지 반복(㉢)하므로 원형 리스트의 전체 노드 수를 정확히 센다.
② ptr == NULL / temp = temp→link / temp == ptr — ㉠이 틀렸다. ptr == NULL이면 리스트가 비어 있다는 뜻인데, 이 조건이 참일 때 순회 블록(do-while)에 들어가면 안 된다. 빈 리스트인데도 안으로 들어가 temp = ptr(즉 NULL)로 두고 count++와 temp→link를 실행하면 널 포인터 역참조가 발생해 오류가 난다. ㉢도 temp == ptr이면 시작하자마자(첫 반복에서 temp를 옮기기 전이 아니라 옮긴 후 비교하므로) 반복을 멈추는 조건이 되어 순회가 정상적으로 누적되지 않는다.
③ ptr != NULL / ptr = temp→link / temp != ptr — ㉡이 틀렸다. ptr = temp→link로 쓰면 순회 중에 시작 노드를 가리키는 ptr 자체가 계속 바뀌어 버린다. ㉢에서 temp != ptr로 종료 조건을 판단하는데, ptr이 함께 이동해 버리면 temp와 ptr이 우연히 같아지는 시점이 원래 의도한 '한 바퀴 완주' 시점과 달라져 개수를 잘못 센다. 이동해야 할 대상은 temp이지 ptr이 아니다.
④ ptr == NULL / ptr = temp→link / temp == ptr — ㉠, ㉡ 모두 틀렸다. 빈 리스트일 때 순회에 들어가는 오류(㉠)와 순회 기준점 ptr을 옮겨버리는 오류(㉡)가 겹쳐 있다.

### 오답 포인트

이 문제의 함정은 두 가지이다. 첫째, ㉠의 조건 방향(!= NULL vs == NULL)을 반대로 읽기 쉬운데, if 블록 안으로 들어가야 순회를 '수행'하는 것이므로 '리스트가 비어 있지 않을 때' 들어가야 한다. 즉 조건은 ptr != NULL이 되어야 한다. 둘째, ㉡에서 어떤 변수를 이동시켜야 하는지 헷갈리기 쉬운데, 순회 커서 역할은 temp가 하고 ptr은 시작점(종료 기준)으로 고정되어야 하므로 temp = temp→link가 맞고 ptr = temp→link는 틀리다.

NoteBase

### 다시 나오면 이렇게 판단

원형 리스트를 do-while로 순회하는 코드가 나오면, 먼저 '무엇이 시작점 기준(고정)이고 무엇이 이동하는 커서인지'를 구분한다. 시작점(ptr)은 고정해 두고 커서(temp)만 temp = temp→link로 이동시키며, 종료 조건은 커서가 다시 시작점과 같아지는 순간(temp == ptr이 참이 되면 멈춤, 즉 while 조건은 temp != ptr)으로 잡는다. 진입 조건은 자료구조가 비어 있지 않을 때(ptr != NULL) 순회 블록에 들어가는 것이 자연스럽다는 점도 함께 확인한다.

### 암기 포인트

원형 리스트 순회 3원칙: 시작점(ptr)은 고정, 커서(temp)만 이동, 종료는 '커서가 시작점으로 돌아왔는가'로 판단한다. do-while 구조에서는 몸체를 먼저 실행한 뒤 조건을 검사하므로, count++ 후 temp를 이동시키고 나서 temp != ptr을 검사하는 순서가 자연스럽다.

NoteBase

**문 15. 이중 연결 리스트(doubly linked list)의 노드를 나타내는 구조체 node에 이전 노드를 가리키는 포인터 llink와 다음 노드를 가리키는 포인터 rlink가 있다고 하자. 포인터 new_node가 가리키는 노드를 포인터 p가 가리키는 노드의 왼쪽에 삽입할 때, 문장의 수행 순서를 바르게 나열한 것은?**

ㄱ. p→llink→rlink = p→rlink;
ㄴ. p→rlink→llink = p→llink;
ㄷ. p→llink→rlink = new_node;
ㄹ. new_node→rlink = p;
ㅁ. p = new_node;
ㅂ. new_node→llink = p→llink;
ㅅ. p→llink = new_node;

① ㄱ → ㄴ
② ㅂ → ㅅ → ㄷ → ㄹ
③ ㄷ → ㅂ → ㄹ → ㅁ
④ ㄷ → ㄹ → ㅂ → ㅅ

**정답 4번**

### 출제 포인트

이중 연결 리스트에서 포인터 p가 가리키는 노드의 왼쪽에 new_node를 삽입할 때 필요한 포인터 연산이 무엇이고, 그 연산들 사이에 어떤 순서 제약이 있는지를 묻는 문제이다. 삽입 전의 연결 상태를 X ↔ p (X는 p의 왼쪽 노드, 즉 원래의 p→llink)라고 하면, 삽입 후에는 X ↔ new_node ↔ p가 되어야 한다. 이를 위해 필요한 연산은 new_node→llink = p→llink(ㅂ), new_node→rlink = p(ㄹ), p→llink→rlink = new_node(ㄷ), p→llink = new_node(ㅅ) 네 가지이며, 나머지 ㄱ · ㄴ · ㅁ은 이 삽입 절차에 쓰이지 않는 문장이다.

### 올바른 순서

핵심은 p→llink가 가리키는 원래 노드(X)의 값을 잃기 전에 그 값을 이용하는 연산부터 처리해야 한다는 점이다. p→llink 값을 읽어서 쓰는 연산은 ㅂ(new_node→llink = p→llink)과 ㄷ(p→llink→rlink = new_node) 두 가지이고, ㅅ(p→llink = new_node)은 p→llink 자체를 new_node로 덮어써 원래 값을 사라지게 만드는 연산이다. 따라서 ㅅ은 반드시 ㅂ과 ㄷ보다 뒤에 와야 한다. ㄹ(new_node→rlink = p)은 p→llink 값과 무관하게 언제 실행해도 결과가 같으므로 ㅂ · ㄷ과의 선후 관계는 자유롭다. 결국 { ㅂ, ㄹ, ㄷ }을 임의의 순서로 먼저 수행한 뒤 마지막에 ㅅ을 수행하면 삽입이 올바르게 완료되며, 선택지 4번의 ㄷ → ㄹ → ㅂ → ㅅ은 이 조건(ㅅ이 맨 마지막)을 만족하는 유효한 순서이다.

NoteBase

### 선지별 해설

① ㄱ → ㄴ: X → ㄱ(p→llink→rlink = p→rlink)은 p의 왼쪽 노드의 rlink를 p의 오른쪽 노드로 바꾸는 문장으로, 이는 p 노드 자체를 리스트에서 삭제할 때 쓰는 연산이며 new_node를 삽입하는 절차와는 무관하다. ㄴ(p→rlink→llink = p→llink) 역시 p의 오른쪽 노드의 llink를 바꾸는 삭제용 연산이다. 두 문장 모두 new_node를 전혀 사용하지 않으므로 삽입 순서가 될 수 없다.
② ㅂ → ㅅ → ㄷ → ㄹ: X → ㅂ(new_node→llink = p→llink)까지는 맞지만, 그 다음 ㅅ(p→llink = new_node)을 먼저 실행하면 p→llink가 new_node로 바뀌어 버린다. 이 상태에서 ㄷ(p→llink→rlink = new_node)을 실행하면 p→llink가 이미 new_node이므로 new_node→rlink = new_node가 되어 자기 자신을 가리키는 오류가 발생한다. ㅅ이 ㄷ보다 먼저 오면 원래 왼쪽 노드의 rlink를 고칠 수 없으므로 틀린 순서이다.
③ ㄷ → ㅂ → ㄹ → ㅁ: X → ㄷ, ㅂ, ㄹ까지는 삽입에 필요한 연산이 맞게 들어 있으나, 마지막의 ㅁ(p = new_node)은 p 포인터 자체가 new_node를 가리키도록 바꾸는 문장이다. 이렇게 되면 p는 더 이상 원래 가리키던 노드를 가리키지 않게 되어 리스트 구조가 깨지며, 정작 p→llink = new_node(ㅅ)가 빠져 있어 원래 p 노드의 llink가 여전히 옛 노드를 가리킨다. 따라서 삽입이 완성되지 않는다.
④ ㄷ → ㄹ → ㅂ → ㅅ: X → ㄷ(p→llink→rlink = new_node)으로 원래 왼쪽 노드가 new_node를 가리키게 하고, ㄹ(new_node→rlink = p)으로 new_node가 p를 가리키게 한다. 이 두 문장은 아직 p→llink 값이 바뀌기 전이므로 원래의 왼쪽 노드를 올바르게 참조한다. 이어서 ㅂ(new_node→llink = p→llink)으로 new_node가 원래 왼쪽 노드를 가리키게 하고, 마지막으로 ㅅ(p→llink = new_node)으로 p의 llink를 new_node로 바꾸어 삽입을 완성한다. p→llink 값을 필요로 하는 연산(ㄷ, ㅂ)이 모두 그 값을 바꾸는 연산(ㅅ)보다 앞에 배치되어 있으므로 올바른 순서이다.

### 오답 포인트

이런 유형에서 가장 흔한 실수는 p→llink를 새 값(new_node)으로 바꾸는 문장(ㅅ)을 너무 일찍 배치하는 것이다. p→llink는 삽입 절차 중간에 원래 왼쪽 노드를 가리키는 용도로 두 번(ㄷ, ㅂ) 더 쓰이므로, 이 값을 덮어쓰는 ㅅ은 반드시 그 두 문장 뒤에 와야 한다. 또한 ㄱ·ㄴ은 삭제 연산에 쓰이는 문장이고 ㅁ은 포인터 p 자체를 옮겨버리는 문장이라는 점을 혼동하지 않아야 한다.

### 다시 나오면 이렇게 판단

연결 리스트 삽입·삭제 순서 문제가 나오면 먼저 어떤 포인터 변수(p→llink, p→rlink 등)가 이후 단계에서 다시 읽히는지를 확인한다. 그 포인터 값을 덮어쓰는 문장은 그 값을 읽어서 쓰는 다른 모든 문장보다 반드시 뒤에 배치해야 하며, 이 제약을 만족하는 한 나머지 문장들 사이의 순서는 여러 가지가 가능함을 염두에 둔다.

### 암기 포인트

이중 연결 리스트에서 p의 왼쪽에 new_node를 삽입하는 절차는 new_node→llink = p→llink, new_node→rlink = p, p→llink→rlink = new_node, p→llink = new_node 네 단계이며, p→llink를 new_node로 바꾸는 마지막 단계는 항상 맨 뒤에 와야 한다는 점을 기억해 둔다.

NoteBase

**문 16. 다음 C 언어를 사용하여 원형 큐(circular queue)의 삽입(enqueue)과 삭제(dequeue) 연산을 구현한 것이다. front와 rear의 값이 0인 공백 큐에 6개의 값 1, 3, 2, 4, 8, 6을 차례대로 삽입하고, 여기에서 2개의 값을 삭제한 다음, 다시 3개의 값 5, 7, 9를 차례대로 삽입할 때, 배열 Q의 최종 모습은?**

```
int Q[8] = {0};
int front = 0;
int rear = 0;

void enqueue(int item)
{
  rear = (rear + 1) % 8;
  if (front == rear) {
    printf("Queue is full.\n");
    return;
  }
  Q[rear] = item;
}

int dequeue()
{
  int item;
  if (front == rear) {
    printf("Queue is empty.\n");
    exit(1);
  }
  else {
    front = (front + 1) % 8;
    item = Q[front];
    Q[front] = 0;
    return item;
  }
}
```

① 

| Q[0] | Q[1] | Q[2] | Q[3] | Q[4] | Q[5] | Q[6] | Q[7] |
|---|---|---|---|---|---|---|---|
| 0 | 7 | 9 | 2 | 4 | 8 | 6 | 5 |

② 

| Q[0] | Q[1] | Q[2] | Q[3] | Q[4] | Q[5] | Q[6] | Q[7] |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 8 | 6 | 5 | 7 | 9 | 0 |

③ 

| Q[0] | Q[1] | Q[2] | Q[3] | Q[4] | Q[5] | Q[6] | Q[7] |
|---|---|---|---|---|---|---|---|
| 7 | 9 | 0 | 2 | 4 | 8 | 6 | 5 |

④ 

| Q[0] | Q[1] | Q[2] | Q[3] | Q[4] | Q[5] | Q[6] | Q[7] |
|---|---|---|---|---|---|---|---|
| 9 | 0 | 2 | 4 | 8 | 6 | 5 | 7 |

**정답 3번**

### 출제 포인트

크기 8인 배열로 구현한 원형 큐에서 enqueue와 dequeue가 호출될 때마다 front와 rear가 먼저 (index + 1) % 8로 이동한 뒤 값을 쓰거나 읽는 방식임을 이해하고 있는지 묻는다. 이 코드는 rear를 먼저 옮긴 다음 그 위치에 저장하고, front도 먼저 옮긴 다음 그 위치의 값을 꺼내고 0으로 지우는 구조이므로, 큐가 실제로 채우는 칸은 Q[1]부터 시작하고 Q[0]는 마지막에 rear가 한 바퀴 돌아왔을 때만 채워진다는 점을 놓치기 쉽다.

NoteBase

### 조건 정리

초기 상태는 Q[8]={0,0,0,0,0,0,0,0}, front=0, rear=0이다. enqueue는 rear=(rear+1)%8을 먼저 계산하고 front==rear이면 가득 찬 것으로 보고 저장하지 않으며, 아니면 Q[rear]=item으로 저장한다. dequeue는 front==rear이면 공백으로 보고 종료하며, 아니면 front=(front+1)%8을 먼저 계산하고 item=Q[front]를 꺼낸 뒤 Q[front]=0으로 지운다. 수행 순서는 1, 3, 2, 4, 8, 6을 순서대로 enqueue한 다음 dequeue를 2회 호출하고, 마지막으로 5, 7, 9를 순서대로 enqueue하는 것이다.

### 사용할 공식

인덱스 이동 공식은 rear = (rear + 1) % 8과 front = (front + 1) % 8이다. 두 식 모두 나눗셈의 나머지를 이용해 인덱스가 7 다음에 다시 0으로 돌아가도록 만든다.

### 계산 과정

enqueue(1): rear=(0+1)%8=1, Q[1]=1.
enqueue(3): rear=2, Q[2]=3.
enqueue(2): rear=3, Q[3]=2.
enqueue(4): rear=4, Q[4]=4.
enqueue(8): rear=5, Q[5]=8.
enqueue(6): rear=6, Q[6]=6.
이 시점 배열은 Q[0..7]=0,1,3,2,4,8,6,0이고 front=0, rear=6이다.

dequeue() 1회차: front=(0+1)%8=1, item=Q[1]=1 반환, Q[1]=0.
dequeue() 2회차: front=(1+1)%8=2, item=Q[2]=3 반환, Q[2]=0.
이 시점 배열은 Q[0..7]=0,0,0,2,4,8,6,0이고 front=2, rear=6이다.

enqueue(5): rear=(6+1)%8=7, Q[7]=5.
enqueue(7): rear=(7+1)%8=0, Q[0]=7.
enqueue(9): rear=(0+1)%8=1, Q[1]=9.

최종적으로 front=2, rear=1이고 배열은 Q[0]=7, Q[1]=9, Q[2]=0, Q[3]=2, Q[4]=4, Q[5]=8, Q[6]=6, Q[7]=5이다.

NoteBase

### 선지 판단

① Q[0]=0, Q[1]=7 → 틀린 표현. rear가 0으로 되돌아왔을 때 enqueue(7)이 실행되어 Q[0]에는 0이 아니라 7이 저장되고, Q[1]에는 이후 enqueue(9)에 의해 9가 저장되므로 순서가 다르다.
② Q[0]=2 → 틀린 표현. dequeue로 지워진 자리(Q[1], Q[2])와 아직 값이 남아 있는 자리(Q[3]=2, Q[4]=4, …)의 위치를 뒤섞어 배열했으며, 실제로 Q[3]에 남아 있어야 할 2가 Q[0]에 놓여 있어 인덱스 이동 순서와 맞지 않는다.
③ Q[0]=7, Q[1]=9, Q[2]=0, Q[3]=2, Q[4]=4, Q[5]=8, Q[6]=6, Q[7]=5 → 옳은 표현. 계산 과정에서 도출한 최종 배열과 정확히 일치한다.
④ Q[0]=9 → 틀린 표현. enqueue(7)이 Q[0]에, enqueue(9)가 Q[1]에 저장되는 순서를 반대로 뒤집어, 더 나중에 삽입된 9를 Q[0]에 놓은 오류이다.

### 오답 포인트

가장 흔한 실수는 rear=(rear+1)%8을 값을 저장하기 전에 수행한다는 점을 놓치고 Q[rear]=item을 먼저 계산한 인덱스가 아니라 이전 rear 위치에 저장하는 것으로 착각하는 것이다. 또한 dequeue에서도 front를 먼저 옮긴 뒤 그 위치의 값을 꺼내므로, front가 가리키던 칸이 아니라 그다음 칸의 값이 꺼내진다는 점을 헷갈리면 안 된다. 마지막으로 rear가 7에서 0으로 넘어가는 순환 지점(enqueue(7)이 Q[0]에 저장되는 부분)을 놓치고 Q[0]을 계속 초기값 0으로 남겨 두는 오류도 흔하다.

### 다시 나오면 이렇게 판단

원형 큐 트레이싱 문제는 매 연산마다 rear 또는 front가 먼저 이동하는지, 이동한 후의 위치에 값을 쓰거나 읽는지를 표로 정리하며 한 단계씩 손으로 짚어 나간다. 이 코드처럼 'rear를 먼저 옮기고 그 자리에 저장'하는 방식과 '값을 먼저 넣고 rear를 나중에 옮기는' 방식은 결과가 다르므로, 코드의 순서를 그대로 따라가는 것이 핵심이다.

### 암기 포인트

이 코드는 rear=(rear+1)%8 → Q[rear]=item 순서(선증분 후저장)이고, dequeue는 front=(front+1)%8 → item=Q[front] → Q[front]=0 순서(선증분 후추출 후삭제)이다. 이 순서를 지켜 한 단계씩 배열 상태를 갱신하면 최종 결과를 정확히 구할 수 있다.

NoteBase

**문 17. 다음 C 프로그램의 출력 값으로 옳은 것은? (단, int 데이터 타입은 4 바이트로 표현되며, 배열 a의 시작 주소는 2293520이다)**

```
int main()
{
   int a[3][4] = {{10, 15, 20, 25}, {30, 35, 40, 45},
                  {50, 55, 60, 65}};
   printf("a+1 = %d, ", a+1);
   printf("*(*a+2) = %d, ", *(*a+2));
   printf("*(*(a+1)+2) = %d\n", *(*(a+1)+2));
   return 0;
}
```

① a+1 = 2293521, *(*a+2) = 50, *(*(a+1)+2) = 55

② a+1 = 2293536, *(*a+2) = 20, *(*(a+1)+2) = 40

③ a+1 = 2293521, *(*a+2) = 20, *(*(a+1)+2) = 40

④ a+1 = 2293536, *(*a+2) = 50, *(*(a+1)+2) = 55

**정답 2번**

**출제 포인트**

2차원 배열의 이름이 갖는 포인터 성질을 정확히 구분해서 이해하는지를 묻는다. 배열 이름 a는 *int*[4] 4개짜리 행을 가리키는 포인터이고, *a(즉 a[0])는 int 하나를 가리키는 포인터이다. 이 둘의 산술 연산 단위(스케일)가 다르다는 점이 핵심이다.

**조건 정리**

배열 선언은 int a[3][4]이고 초기값은 a[0]={10,15,20,25}, a[1]={30,35,40,45}, a[2]={50,55,60,65}이다. int는 4바이트이고 배열 a의 시작 주소는 2293520이다.

NoteBase

**사용할 공식**

a는 int[4] 타입 3개(행)로 이루어진 배열이므로 a+1은 한 행(4개의 int, 즉 4 × 4 = 16 바이트)만큼 이동한다. 따라서 a+1의 값은 2293520 + 16 = 2293536이다. 반면 *a는 a[0]의 첫 원소를 가리키는 int 포인터이므로 *a+2는 int 2개(2 × 4 = 8 바이트)만큼 이동하며, *(*a+2)는 그 위치에 저장된 값(정수) 자체를 의미한다.

**계산 과정**

*(*a+2)는 a[0]의 원소 중 인덱스 2, 즉 a[0][2]의 값이다. a[0]={10,15,20,25}이므로 a[0][2]=20이다.
*(*(a+1)+2)는 a+1이 가리키는 행, 즉 a[1]의 원소 중 인덱스 2, 즉 a[1][2]의 값이다. a[1]={30,35,40,45}이므로 a[1][2]=40이다.
정리하면 a+1=2293536, *(*a+2)=20, *(*(a+1)+2)=40이다.

**선지 판단**

① a+1=2293521, *(*a+2)=50, *(*(a+1)+2)=55 → X, a+1 값과 두 출력값 모두 틀렸다. a+1은 2293536이어야 하고 20, 40이어야 한다.
② a+1=2293536, *(*a+2)=20, *(*(a+1)+2)=40 → O, 계산 결과와 정확히 일치한다.
③ a+1=2293521, *(*a+2)=20, *(*(a+1)+2)=40 → X, 뒤의 두 값(20, 40)은 맞지만 a+1 값이 2293521로 틀렸다. a+1은 배열 원소 하나(1바이트)가 아니라 한 행(16바이트) 단위로 증가한다.
④ a+1=2293536, *(*a+2)=50, *(*(a+1)+2)=55 → X, a+1 값은 맞지만 뒤의 두 출력값이 틀렸다. 50과 55는 각각 a[2][0], a[2][1]에 해당하는 값으로 인덱스를 잘못 짚은 경우이다.

**오답 포인트**

가장 흔한 실수는 a+1을 배열 전체 시작 주소에서 단순히 1을 더한 값(2293521)으로 계산하는 것이다. a는 int형 변수가 아니라 int[4] 행을 가리키는 포인터이므로, a+1은 행 하나의 크기인 16바이트만큼 이동한다. 또 다른 실수는 *(*a+2)와 *(*(a+1)+2)의 행 인덱스를 혼동해 a[2]의 값(50, 55)을 가져오는 것인데, a+1은 두 번째 행(a[1])을 가리키므로 인덱스 1의 행에서 값을 읽어야 한다.

**다시 나오면 이렇게 판단**

2차원 배열 a[m][n]에서 a는 *int*[*n*] 타입 포인터, *a(=a[0])는 int 타입 포인터임을 먼저 구분한다. a+k는 k×n×(자료형 크기)바이트만큼, *a+k 또는 *(a+i)+k는 k×(자료형 크기)바이트만큼 이동한다는 스케일 차이를 기준으로 각 표현식이 몇 바이트를 이동하는지부터 계산한다.

**암기 포인트**

2차원 배열에서 a+1은 행 단위 이동(n×크기), *(a+i)+j 또는 *(*a+j)는 원소 단위 이동(크기)이며, *(*(a+i)+j)는 a[i][j]의 값과 같다는 대응 관계를 기억해 둔다.

NoteBase

**문 18. 다음 인접행렬이 표현하는 그래프의 부분 그래프 중에서 간선(edge)이 최소한 1개 이상인 연결 그래프의 개수는?**

| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 1 | 1 | 1 |
| [1] | 1 | 0 | 1 | 0 |
| [2] | 1 | 1 | 0 | 0 |
| [3] | 1 | 0 | 0 | 0 |

① 12

② 13

③ 14

④ 15

**정답 3번**

**출제 포인트**

인접행렬로 주어진 그래프에서 간선의 부분집합을 골라 만들 수 있는 모든 부분 그래프 중, 간선이 1개 이상이면서 그래프 전체가 하나로 연결된(고립된 성분이 없는) 경우의 수를 세는 문제이다. 정점 자체가 아니라 간선을 기준으로 부분집합을 만든다는 점과, 부분 그래프에 포함되지 않은 정점은 그래프에서 빠진 것으로 보고 연결성을 판단한다는 점이 핵심이다.

**조건 정리**

인접행렬을 보면 1이 표시된 칸은 [0]-[1], [0]-[2], [0]-[3], [1]-[2] 네 곳이다(행렬이 대칭이므로 무방향 그래프). 즉 원래 그래프의 간선은 e1=(0,1), e2=(0,2), e3=(0,3), e4=(1,2)의 4개이며, 정점 0을 중심으로 1·2·3이 연결되고 1과 2 사이에도 간선이 하나 더 있어 0-1-2가 삼각형을 이루고 0-3은 삼각형에 매달린 가지 형태이다. 부분 그래프는 이 4개의 간선 중 몇 개를 골라 만든 그래프이며, 고른 간선들이 서로 이어져 하나의 성분을 이루면 연결 그래프이다.

NoteBase

**사용할 공식**

간선이 4개이므로 간선의 부분집합은 $2^4 = 16$가지이며, 이 중 간선이 0개인 공집합(고립점만 있는 경우)을 제외하면 후보는 $16 - 1 = 15$가지이다. 각 후보에 대해 선택된 간선들이 그래프 이론적으로 하나의 연결 성분을 이루는지(즉 간선으로 이어진 정점들 사이에 서로 도달 가능한지) 직접 확인해 개수를 센다.

**계산 과정**

간선 1개짜리 부분 그래프: e1, e2, e3, e4 각각 두 정점만 잇는 하나의 연결 성분이므로 4가지 모두 연결 그래프이다.
간선 2개짜리 부분 그래프(총 6가지): {e1,e2}는 정점 0을 공유해 0-1, 0-2로 연결. {e1,e3}는 정점 0을 공유해 연결. {e1,e4}는 정점 1을 공유해 0-1-2로 연결. {e2,e3}는 정점 0을 공유해 연결. {e2,e4}는 정점 2를 공유해 0-2-1로 연결. 여기까지 5가지가 연결 그래프이다. 반면 {e3,e4}는 (0,3)과 (1,2)로 공유하는 정점이 없어 두 개의 분리된 성분이 되므로 연결 그래프가 아니다. 따라서 2개짜리는 6가지 중 5가지만 연결.
간선 3개짜리 부분 그래프(총 4가지): {e1,e2,e3}는 정점 0에서 1·2·3으로 뻗는 별 모양으로 연결. {e1,e2,e4}는 0-1-2 삼각형으로 연결. {e1,e3,e4}는 0-1, 0-3, 1-2로 정점 0,1,2,3이 모두 이어져 연결. {e2,e3,e4}도 0-2, 0-3, 1-2로 정점 0,1,2,3이 모두 이어져 연결. 4가지 모두 연결 그래프이다.
간선 4개짜리(원래 그래프 전체) 1가지도 당연히 연결 그래프이다.
전체 합계는 $4 + 5 + 4 + 1 = 14$가지이다.

**선지 판단**

① 12 → 간선 2개 조합에서 연결되지 않는 경우({e3,e4})를 이미 반영해도 12는 실제 집계값(14)보다 작아 계산 결과와 맞지 않는다. X
② 13 → 간선 2개 조합 중 연결되는 5가지 중 1가지를 더 빼거나 세는 과정에서 하나를 누락했을 때 나오는 값으로, 실제로는 5가지가 모두 연결되므로 맞지 않는다. X
③ 14 → 간선 1개(4가지)+2개 중 연결(5가지)+3개(4가지, 전부 연결)+4개(1가지)를 모두 더한 값과 정확히 일치한다. O
④ 15 → 이는 간선이 1개 이상인 전체 부분집합의 개수(공집합 제외, $2^4 - 1$)로, 연결되지 않는 {e3,e4} 경우까지 포함한 값이라 문제에서 요구하는 연결 그래프의 개수보다 1개 많다. X

**오답 포인트**

이 문제의 함정은 부분 그래프의 전체 개수(15가지)와 그중 연결 그래프인 개수(14가지)를 혼동하는 데 있다. 4개의 간선 중 (0,3)과 (1,2)만 골라 만든 부분 그래프는 정점 0-3 성분과 정점 1-2 성분으로 나뉘어 서로 연결되지 않으므로, 이 한 가지 경우만 제외하면 된다는 점을 놓치기 쉽다.

NoteBase

### 다시 나오면 이렇게 판단

먼저 인접행렬에서 1이 있는 칸을 모두 찾아 간선 목록을 만들고, 간선 개수가 n개이면 부분집합은 $2^n$가지, 공집합을 제외하면 $2^n - 1$가지임을 확인한다. 그다음 각 부분집합이 실제로 하나의 연결 성분을 이루는지 정점을 공유하는지 여부로 하나씩 점검하면서, 연결되지 않는 경우의 수만 골라내 전체에서 빼는 방식으로 계산한다.

### 암기 포인트

간선이 n개인 그래프의 간선 부분집합 총수는 $2^n$이고, 간선이 1개 이상인 부분집합은 $2^n - 1$이다. 부분 그래프의 연결 여부는 선택된 간선들이 정점을 공유하며 하나의 성분으로 이어지는지로 판단하며, 정점을 전혀 공유하지 않는 간선 조합만 연결이 끊긴 경우로 골라내면 계산이 빠르다.

NoteBase

**문 19. 기수 정렬(radix sort)을 수행하여 다음 원소들을 오름차순으로 정렬하고자 한다. 각 단계별 정렬 순서로 나타낼 수 없는 것은? (단, 정수 $K$는 $0 \le K \le 999$ 범위에 있고 각 자리수를 나타내는 3개의 서브키($K^1$, $K^2$, $K^3$)로 구성되어 있다고 생각한다. 여기서 $K^1$는 일의 자리 수, $K^2$는 십의 자리 수, $K^3$는 백의 자리 수를 의미하고, $0 \le K^i \le 9$이다)**

129, 308, 506, 92, 3, 841, 33

① 3, 506, 308, 129, 33, 841, 92

② 92, 3, 33, 129, 308, 506, 841

③ 841, 92, 3, 33, 506, 308, 129

④ 3, 33, 92, 129, 308, 506, 841

**정답 2번**

### 출제 포인트

기수 정렬(radix sort)은 LSD(least significant digit) 방식으로, 자리수가 작은 서브키부터 큰 서브키 순서로 안정 정렬(stable sort)을 반복 적용한다. 이 문제는 원소 129, 308, 506, 92, 3, 841, 33에 대해 일의 자리($K^1$) → 십의 자리($K^2$) → 백의 자리($K^3$) 순으로 버킷 정렬을 수행했을 때, 각 단계가 끝난 직후 나타날 수 있는 배열 순서를 정확히 계산할 수 있는지를 확인한다.

### 조건 정리

정렬 대상은 129, 308, 506, 92, 3, 841, 33이며 안정 정렬을 전제로 한다. $K^1$은 일의 자리, $K^2$는 십의 자리, $K^3$는 백의 자리이고 각 자리는 0부터 9까지의 버킷(0~9)에 원소를 순서대로 분배한 뒤 버킷 번호 0부터 9까지 차례로 이어붙여 다음 단계의 입력으로 사용한다.

### 사용할 공식

각 단계는 안정 버킷 정렬이다. 단계 t에서 자리수 값 $K^t$가 작은 것부터 큰 것 순서(0→9)로 버킷을 순회하되, 같은 버킷 안에서는 입력으로 들어온 순서를 그대로 유지한다.

NoteBase

**계산 과정**

초기 배열: 129, 308, 506, 92, 3, 841, 33.

1단계($K^1$, 일의 자리 기준): 129→9, 308→8, 506→6, 92→2, 3→3, 841→1, 33→3. 버킷 0부터 순서대로 모으면 1(841), 2(92), 3(3, 33), 6(506), 8(308), 9(129)가 되어 결과는 841, 92, 3, 33, 506, 308, 129이다.

2단계($K^2$, 십의 자리 기준, 1단계 결과를 입력으로 사용): 841→4, 92→9, 3→0, 33→3, 506→0, 308→0, 129→2. 버킷 0(3, 506, 308), 2(129), 3(33), 4(841), 9(92) 순으로 모으면 결과는 3, 506, 308, 129, 33, 841, 92이다.

3단계($K^3$, 백의 자리 기준, 2단계 결과를 입력으로 사용): 3→0, 506→5, 308→3, 129→1, 33→0, 841→8, 92→0. 버킷 0(3, 33, 92), 1(129), 3(308), 5(506), 8(841) 순으로 모으면 최종 결과는 3, 33, 92, 129, 308, 506, 841이다.

**선지 판단**

① 3, 506, 308, 129, 33, 841, 92 → O, 2단계(십의 자리 정렬)가 끝난 직후의 상태와 정확히 일치한다.
② 92, 3, 33, 129, 308, 506, 841 → X, 1단계·2단계·3단계 결과 어느 것과도 일치하지 않는다. 이 배열은 값 자체는 오름차순처럼 보이지만 92가 3, 33보다 앞에 있어 안정 기수 정렬의 어떤 중간 단계에서도 나타날 수 없는 순서이다.
③ 841, 92, 3, 33, 506, 308, 129 → O, 1단계(일의 자리 정렬)가 끝난 직후의 상태와 정확히 일치한다.
④ 3, 33, 92, 129, 308, 506, 841 → O, 3단계(백의 자리 정렬)까지 모두 끝난 최종 오름차순 결과와 일치한다.

**오답 포인트**

①③④를 오답으로 고르는 실수는 각 단계의 정렬 기준 자리수를 헷갈리거나, 안정 정렬에서 같은 버킷 내 순서(입력 순서 유지)를 무시하고 임의로 재배열해 대입하기 때문에 생긴다. 특히 ②처럼 최종 결과와 얼핏 비슷해 보이는 오름차순 배열이 제시되면, 실제로 존재하지도 않는 값의 순서를 정답 후보 최종 단계로 착각하기 쉬우므로 반드시 1단계부터 순서대로 직접 버킷에 넣어 확인해야 한다.

**다시 나오면 이렇게 판단**

이런 문제는 선지를 눈으로 비교하지 말고, 초기 배열에서 시작해 일의 자리 → 십의 자리 → 백의 자리 순서로 버킷 0~9에 원소를 직접 배치해 매 단계의 결과를 손으로 적어본다. 그렇게 얻은 세 개의 중간·최종 배열과 선지를 하나씩 대조하면, 어느 단계에도 해당하지 않는 선지를 쉽게 골라낼 수 있다.

**암기 포인트**

기수 정렬은 낮은 자리수부터 높은 자리수 순으로 안정 정렬을 반복하는 LSD 방식이며, 안정 정렬이므로 같은 버킷에 들어간 원소는 입력 순서를 그대로 유지한 채 이어붙인다는 점을 기억해 둔다.

NoteBase

**문 20. 행렬 $M = (a_{ij})$가 $|i-j| > 1$ 이면 0 값을 가질 때, 이러한 행렬을 삼중대각행렬(tridiagonal matrix)이라고 하며, 다음은 $6 \times 6$ 삼중대각행렬의 예를 보여준다. (단, $1 \le i \le m$, $1 \le j \le n$) 행렬 $M(m \times n)$의 0이 아닌 원소 $a_{ij}$를 열 우선(column major) 순서로 배열 $A$에 저장한다고 할 때, 0이 아닌 원소 $a_{ij}$가 배열 $A$에 저장되는 위치를 계산하는 수식으로 옳은 것은? (단, 배열 $A$의 시작주소는 $\alpha$이고, 행렬 $M$의 0인 원소는 배열에 저장하지 않는다)**

bmatrixa_11 & a_12 & 0 & 0 & 0 & 0 \ a_21 & a_22 & a_23 & 0 & 0 & 0 \ 0 & a_32 & a_33 & a_34 & 0 & 0 \ 0 & 0 & a_43 & a_44 & a_45 & 0 \ 0 & 0 & 0 & a_54 & a_55 & a_56 \ 0 & 0 & 0 & 0 & a_65 & a_66bmatrix

① $\alpha + 3 \cdot (j-1) + (i-j)$

② $\alpha + 3 \cdot (j-1) + (j-i)$

③ $\alpha + 3 \cdot (i-1) + (i-j)$

④ $\alpha + 3 \cdot (i-1) + (j-i)$

**정답 1번**

### 출제 포인트

삼중대각행렬(tridiagonal matrix)처럼 0이 아닌 원소가 대각선 주변에만 몰려 있는 희소행렬을, 0을 저장하지 않고 열 우선(column major) 순서로 배열에 압축 저장할 때의 주소 계산식을 구하는 문제이다. 각 열에 저장되는 원소의 개수와 순서를 정확히 파악하는 것이 핵심이다.

### 조건 정리

행렬 M은 |i-j|>1이면 값이 0이므로, 각 열 j에는 행 i가 j-1, j, j+1인 세 원소만 0이 아니다(단, 첫 열과 마지막 열은 2개). 저장은 열 우선이므로 1열의 원소들을 모두 저장한 뒤 2열, 3열 순으로 이어서 저장한다. 배열 A의 시작 주소는 α이고, 0인 원소는 아예 저장하지 않는다.

### 사용할 공식

열 우선 저장이므로 주소는 α + (원소 a_ij 이전에 저장된 원소 개수)이다. 1열은 원소가 2개($a_{11}$, $a_{21}$)이고, 2열부터는 각 열마다 원소가 3개씩($a_{j-1,j}$, $a_{jj}$, $a_{j+1,j}$)이다. j열 안에서 원소는 i가 작은 것부터, 즉 $i-j = -1, 0, 1$ 순서로 저장되므로 열 내 순번은 $(i-j)+1$이 된다. j열 이전(1열~(j-1)열)까지 저장된 원소 총 개수는 $3(j-1)-1$이지만, 열 내 순번의 $+1$과 상쇄되어 정리하면 최종 주소식은 $\alpha + 3 \cdot (j-1) + (i-j)$로 깔끔하게 정리된다.

NoteBase

### 계산 과정

실제 저장 순서를 나열하면 $a_{11}(0)$, $a_{21}(1)$|$a_{12}(2)$, $a_{22}(3)$, $a_{32}(4)$|$a_{23}(5)$, $a_{33}(6)$, $a_{43}(7)$|…이다(괄호 안은 α로부터의 오프셋). 이를 ①식 $3(j-1)+(i-j)$에 대입해 검증한다. $a_{11}$: $i=1, j=1 \rightarrow 3\cdot 0+0=0$, 실제 오프셋 0과 일치. $a_{21}$: $i=2, j=1 \rightarrow 0+1=1$, 일치. $a_{22}$: $i=2, j=2 \rightarrow 3+0=3$, 일치. $a_{32}$: $i=3, j=2 \rightarrow 3+1=4$, 일치. $a_{43}$: $i=4, j=3 \rightarrow 6+1=7$, 일치. 모든 원소에서 오프셋이 정확히 맞아떨어지므로 주소식은 $\alpha+3(j-1)+(i-j)$이다.

### 선지 판단

① α+3(j-1)+(i-j): 모든 원소 위치에서 실제 오프셋과 정확히 일치한다. 정답.
② α+3(j-1)+(j-i): i-j의 부호가 반대로 되어 있어, 예를 들어 $a_{12}$(i=1,j=2)에 대입하면 $3+1=4$가 되는데 실제 오프셋은 2이므로 틀림.
③ α+3(i-1)+(i-j): 열 번호 j 대신 행 번호 i를 기준으로 열 간격을 곱해, $a_{12}$에 대입하면 $0+(-1)=-1$이 되어 음수가 나오므로 틀림.
④ α+3(i-1)+(j-i): 마찬가지로 행 번호 i를 기준으로 곱해 $a_{21}$에 대입하면 $3+(-1)=2$가 되는데 실제 오프셋은 1이므로 틀림.

### 오답 포인트

열 우선 저장에서 열 간격(한 열에 몇 개씩 저장되는지)을 곱하는 대상은 반드시 열 번호 j여야 하는데, 행 번호 i에 곱하는 실수(③, ④)를 하기 쉽다. 또한 열 내 순서를 나타내는 항은 i-j(행이 작을수록 먼저 저장)인데 부호를 반대로 쓰는 실수(②)도 흔하다.

### 다시 나오면 이렇게 판단

희소행렬 압축 저장 문제는 먼저 저장 순서(행 우선인지 열 우선인지)를 확정하고, 기준이 되는 반복 단위(이 문제에서는 열)마다 몇 개의 원소가 들어가는지 센 다음, 몇 번째 열/행까지 지났는지(앞부분 개수)와 그 안에서 몇 번째인지(오프셋)를 나누어 더하는 방식으로 공식을 유도한다. 유도한 공식은 반드시 실제 표의 몇 개 원소(예: 첫 열, 둘째 열 시작 원소)에 대입해 검산한다.

### 암기 포인트

열 우선 저장 주소 = 시작주소 + (그 열 이전까지 저장된 원소 개수) + (그 열 안에서의 순번). 삼중대각행렬은 중간 열마다 3개씩 저장되므로 열 간격 계수는 3이고, 곱하는 대상은 항상 열 번호(j)이다.

NoteBase