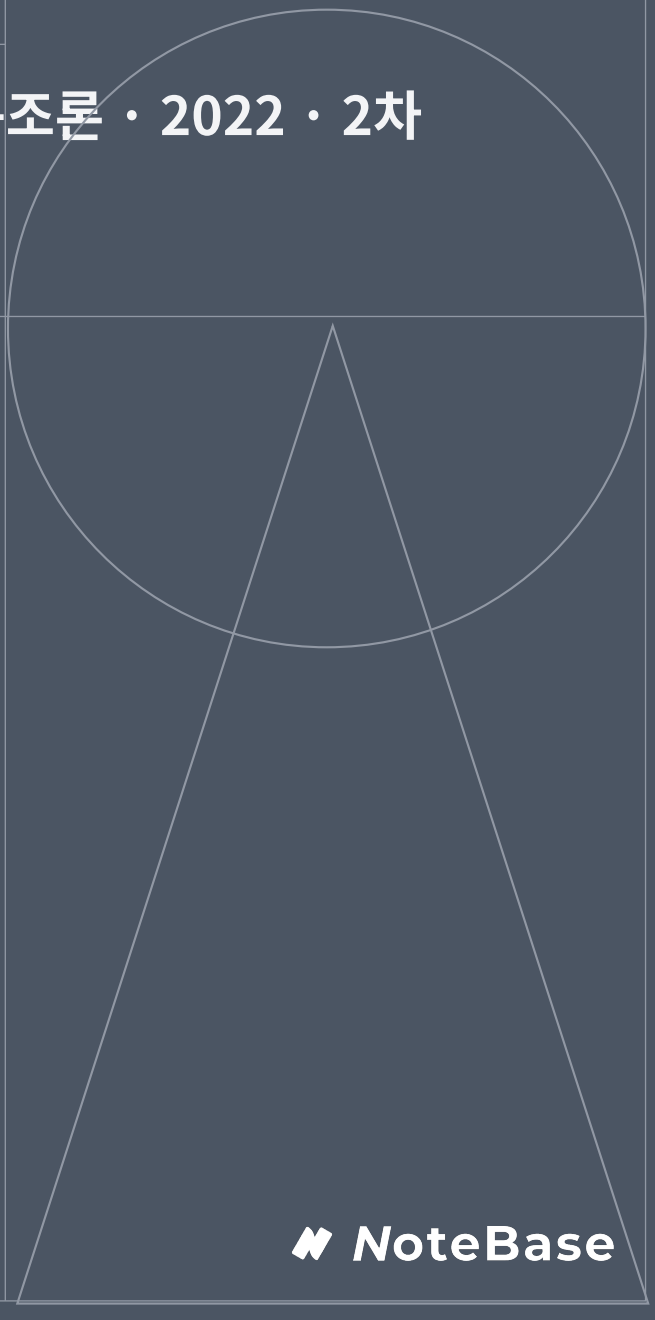
기출OX 해설집

공무원/국가직/7급 · 자료구조론 · 2022 · 2차



NoteBase

기출OX 해설집

공무원/국가직/7급 · 자료구조론 · 2022 · 2차

공무원/국가직/7급 · 자료구조론 · 2022 · 2차

문 1. 세 개의 노드를 가진 무방향 완전 그래프의 신장 트리(spanning tree)는 최대 몇 가지인가?

① 1

② 2

③ 3

④ 4

정답 3번

출제 포인트

이 문제는 완전 그래프 K_3에서 신장 트리가 몇 개 나오는지를 계산하는 문제이다.

조건 정리

주어진 그래프는 노드가 3개인 무방향 완전 그래프이다.
완전 그래프 K_3는 모든 두 노드가 간선으로 연결되어 있어 간선 수가 3개이다.
신장 트리는 모든 노드를 포함하면서 사이클이 없는 부분 그래프이므로 간선 수는 2개이어야 한다.

사용할 공식

완전 그래프 K_n의 신장 트리 수는 Cayley 공식 n^{n-2}로 구할 수 있다.
이 문제는 완전 그래프의 신장 트리 개수를 직접 묻고 있으므로 이 공식을 적용할 수 있다.
또는 K_3가 삼각형이므로 간선 3개 중 하나만 제거하면 사이클이 없어져 가능한 신장 트리가 3개라는 방식으로도 확인할 수 있다.

계산 과정

노드 수는 3이므로 $n = 3$이다.
Cayley 공식을 적용하면 $3^{3-2} = 3^1 = 3$이다.
직접 세어도 삼각형의 세 간선 중 어느 하나를 빼도 2개의 간선으로 모든 노드가 연결되므로 신장 트리는 3가지이다.

NoteBase

선지 판단

1번 X: 신장 트리는 1가지가 아니다. K_3에서는 제거하는 간선의 선택이 3가지이므로 올바르게는 3가지이다.
2번 X: 신장 트리는 2가지가 아니다. 두 가지만 세면 삼각형의 한 변을 제거하는 경우 하나를 빠뜨린 것이므로 올바르게는 3가지이다.
3번 O: K_3의 신장 트리 수는 $3^{3-2} = 3$이므로 옳다.
4번 X: 신장 트리는 4가지가 아니다. 간선이 3개뿐이어서 하나씩 제거하는 경우도 3가지뿐이므로 올바르게는 3가지이다.

오답 포인트

신장 트리를 단순히 간선 2개를 고르는 경우와 혼동하면 오답이 나온다. 이 문제에서는 선택된 간선 2개가 모든 노드를 연결하고 사이클이 없어야 하며, K_3에서는 그 조건을 만족하는 경우가 정확히 3가지이다.

다시 나오면 이렇게 판단

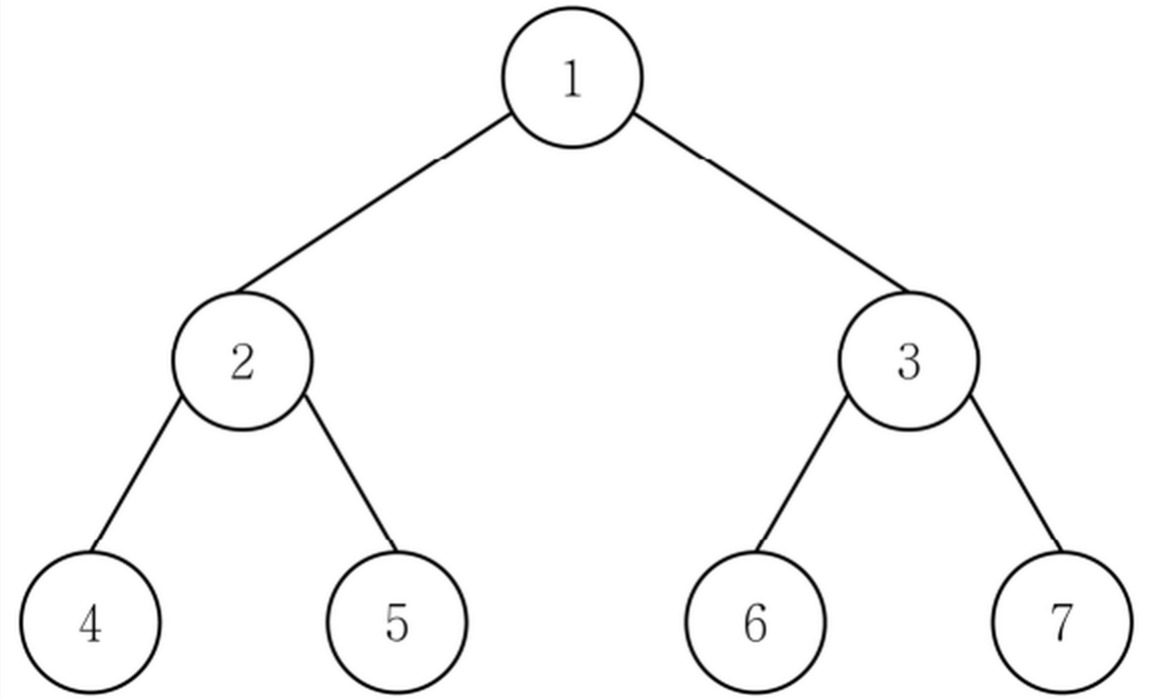
완전 그래프의 신장 트리 개수를 묻는 문제는 먼저 K_n인지 확인하고, 작은 그래프이면 직접 세고 일반형이면 n^{n-2}를 적용해 판단한다.

암기 포인트

완전 그래프 K_n의 신장 트리 수는 n^{n-2}이며, K_3는 3가지이다.

NoteBase

문 2. 다음 트리를 중위 순회한 결과로 옳은 것은?



① 2, 4, 5, 1, 3, 6, 7

② 2, 4, 5, 3, 6, 7, 1

③ 4, 2, 5, 1, 6, 3, 7

④ 4, 5, 2, 1, 6, 7, 3

정답 3번

출제 포인트

이 문제는 이진트리의 중위 순회가 왼쪽 서브트리 → 루트 → 오른쪽 서브트리 순서라는 점을 도식에 적용하는 문제이다.

NoteBase

선지별 해설

① X. 2를 가장 먼저 적은 점이 틀렸다. 중위 순회는 왼쪽 끝 노드 4부터 방문하므로 올바르게는 4, 2, 5, 1, 6, 3, 7이 된다.
② X. 루트 1을 맨 끝에 둔 점이 틀렸다. 중위 순회에서는 왼쪽 서브트리 4, 2, 5 뒤에 루트 1이 오고 그 다음 오른쪽 서브트리 6, 3, 7이 이어져야 한다.
③ O. 왼쪽 서브트리의 중위 순회 결과가 4, 2, 5이고 그 다음 루트 1, 오른쪽 서브트리의 중위 순회 결과가 6, 3, 7이므로 전체 결과는 4, 2, 5, 1, 6, 3, 7이 된다.
④ X. 왼쪽 서브트리를 4, 5, 2로 적은 점과 오른쪽 서브트리를 6, 7, 3으로 적은 점이 틀렸다. 이는 각 서브트리에서 루트를 마지막에 둔 순서이므로, 올바르게는 각 서브트리에서 루트가 두 자식 사이에 오는 4, 2, 5와 6, 3, 7이 된다.

오답 포인트

전위 순회는 루트가 먼저, 후위 순회는 루트가 마지막, 중위 순회는 루트가 가운데 온다는 차이를 섞으면 오답이 된다. 이 문항은 각 서브트리 안에서도 같은 원칙이 반복된다는 점을 놓치기 쉬운 함정을 노린 문제이다.

다시 나오면 이렇게 판단

중위 순회는 각 부분트리를 왼쪽 결과, 루트, 오른쪽 결과의 순서로 분해해 붙이면 된다. 따라서 먼저 왼쪽 서브트리 4, 2, 5를 확정하고 루트 1을 둔 뒤 오른쪽 서브트리 6, 3, 7을 이어 붙이면 된다.

암기 포인트

중위 순회는 왼-루-오이며 부모 노드는 항상 두 자식 사이에 놓인다.

NoteBase

문 3. 다음은 입력 개수 n에 대한 알고리즘 A ~ D의 수행시간 복잡도를 나타낸 것이다. 알고리즘 A ~ D를 수행시간 효율이 좋은 것부터 순서대로 나열한 것은?

알고리즘	수행시간 복잡도
A	$O(n \log n)$
B	$O(2^n)$
C	$O(n^2)$
D	$O(n!)$

① A, C, B, D

② A, C, D, B

③ C, A, B, D

④ C, A, D, B

정답 1번

출제 포인트

이 문제는 빅오 표기에서 각 시간복잡도의 증가율을 비교하여 수행시간 효율이 좋은 순서를 고를 수 있는지를 묻는다.

핵심 개념

빅오 표기법은 입력 크기 n이 커질 때 수행시간이 얼마나 빠르게 증가하는지를 상한 형태로 나타낸다.
수행시간 효율이 좋다는 말은 큰 n에서 증가율이 더 작은 복잡도라는 뜻이다.
이 문항의 비교 순서는 n log n < n^2 < 2^n < n! 이다.

NoteBase

선지별 해설

1번 O: A, C, B, D는 O(n log n), O(n^2), O(2^n), O(n!)의 순서이므로 증가율이 작은 것부터 큰 것까지 정확히 나열한 것이다.
2번 X: 틀린 부분은 D를 B보다 앞에 둔 순서이다. 실제로는 O(2^n)이 O(n!)보다 증가율이 작으므로 B가 D보다 앞서야 한다. 올바르게는 A, C, B, D이다.
3번 X: 틀린 부분은 C를 A보다 앞에 둔 순서이다. 실제로는 O(n log n)이 O(n^2)보다 증가율이 작으므로 A가 C보다 앞서야 한다. 올바르게는 A, C, B, D이다.
4번 X: 틀린 부분은 C와 A의 순서를 바꾼 점, 그리고 D를 B보다 앞에 둔 점이다. 실제로는 O(n log n) < O(n^2)이고 O(2^n) < O(n!)이므로 올바르게는 A, C, B, D이다.

오답 포인트

다항식과 지수함수의 비교는 익숙하지만 O(n log n)과 O(n^2)의 순서를 뒤집거나, O(2^n)과 O(n!)를 둘 다 매우 크다고만 보고 순서를 섞기 쉽다. 특히 2번과 4번은 n!이 2^n보다 더 빠르게 커진다는 점을 놓치면 고르기 쉽다.

다시 나오면 이렇게 판단

빅오 순서 비교는 큰 n에서의 증가율만 본다. 로그가 붙은 선형 시간은 제곱 시간보다 작고, 다항식은 지수함수보다 작으며, 지수함수는 팩토리얼보다 작다는 기준으로 정렬하면 된다.

암기 포인트

증가율 기본 순서는 n log n < n^2 < 2^n < n! 이다.

NoteBase

문 4. 이진 트리에 대한 설명으로 옳은 것만을 모두 고르면?

ㄱ. 공백 이진 트리의 높이를 0으로 본다면, 높이가 5인 이진 트리의 최대 노드 수는 15이다.
ㄴ. 9개의 노드를 가지고 있는 이진 트리는 8개의 간선을 가진다.
ㄷ. 루트 노드의 인덱스를 1로 하는 1차원 배열로 완전 이진 트리를 표현할 때, 인덱스 13인 노드의 부모 노드는 인덱스가 7이다.
ㄹ. 7개 노드로 구성할 수 있는 이진 트리의 최대 높이와 최소 높이의 차는 4이다.

① ㄱ, ㄴ
② ㄴ, ㄷ
③ ㄴ, ㄹ
④ ㄷ, ㄹ

정답 3번

출제 포인트

이 문제는 이진 트리의 높이 정의, 노드와 간선의 관계, 배열 표현에서의 부모 인덱스 규칙을 정확히 구별할 수 있는지를 묻는다.

핵심 개념

이진 트리에서 간선 수는 항상 노드 수보다 1개 적다.
루트를 인덱스 1로 두는 1차원 배열 표현에서는 부모 노드 인덱스가 $\lfloor i/2 \rfloor$이다.
공백 이진 트리의 높이를 0으로 보면 높이 h인 이진 트리의 최대 노드 수는 $2^h - 1$이다.

선지별 해설

① X: ㄴ은 맞지만 ㄱ이 틀리므로 전체 선택이 틀리다. ㄱ의 틀린 표현은 "최대 노드 수는 15이다"이다. 공백 이진 트리의 높이를 0으로 보면 높이 5의 최대 노드 수는 $2^5 - 1 = 31$이므로 올바르게는 "최대 노드 수는 31이다"이다.
② X: ㄴ은 맞지만 ㄷ이 틀리므로 전체 선택이 틀리다. ㄷ의 틀린 표현은 "인덱스 13인 노드의 부모 노드는 인덱스가 7이다"이다. 배열 표현에서 부모 인덱스는 $\lfloor 13/2 \rfloor = 6$이므로 올바르게는 "부모 노드의 인덱스는 6이다"이다.
③ O: ㄴ과 ㄹ이 모두 옳으므로 정답이다. ㄴ은 트리의 기본 성질상 노드가 9개이면 간선은 8개이므로 맞다. ㄹ은 7개 노드의 최대 높이는 한쪽으로만 이어진 경우 7이고 최소 높이는 가장 촘촘한 경우 3이어서 차이가 4이므로 맞다.
④ X: ㄹ은 맞지만 ㄷ이 틀리므로 전체 선택이 틀리다. ㄷ의 틀린 표현은 다시 "부모 노드의 인덱스가 7이다"라는 부분이다. 올바르게는 인덱스 13의 부모 노드는 인덱스 6이다.

NoteBase

오답 포인트

높이의 시작값과 배열 인덱스 규칙을 섞으면 틀리기 쉽다. 특히 ㄱ은 높이 정의를 바꿔도 최대 노드 수 공식을 잘못 적용하면 오답이 되고, ㄷ은 부모 인덱스를 $i/2$가 아니라 $\lfloor i/2 \rfloor$로 계산해야 한다는 점을 놓치기 쉽다.

다시 나오면 이렇게 판단

복합 선택지는 먼저 ㄱ, ㄴ, ㄷ, ㄹ을 각각 참거짓으로 분리해 판단한 뒤 조합을 맞춘다. 트리 문제에서는 노드 수와 간선 수 관계, 배열 표현의 부모 인덱스, 높이에 따른 최대 노드 수 공식을 순서대로 대입하면 된다.

암기 포인트

트리는 간선이 항상 노드보다 1개 적고, 배열 표현의 부모는 $\lfloor i/2 \rfloor$, 높이 h의 최대 노드 수는 $2^h - 1$이다.

NoteBase

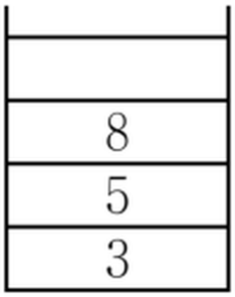
문 5. 크기가 4인 빈 스택에 다음 연산을 차례로 수행한 후의 스택 상태를 바르게 표현한 것은?

push(3) → push(7) → pop() → push(5) → push(8) → peek() → push(10) → pop()

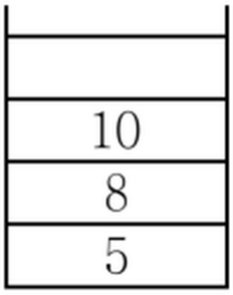
①



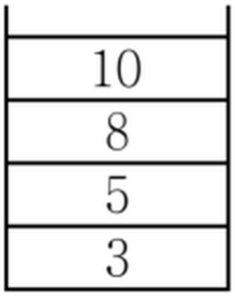
②



③



④



정답 2번

출제 포인트

이 문제는 스택의 후입선출 성질과 각 연산이 스택 상태를 어떻게 바꾸는지를 순서대로 추적할 수 있는지를 묻는다.

NoteBase

올바른 순서

1. push(3) -> push(7)로 스택은 아래부터 3, 7 상태가 된다.
2. pop()으로 7이 제거되고 push(5)로 아래부터 3, 5 상태가 된다.
3. push(8) 후 peek()를 수행하면 아래부터 3, 5, 8 상태가 되고 peek()는 조회만 하므로 스택은 변하지 않는다.
4. push(10)으로 아래부터 3, 5, 8, 10 상태가 된 뒤 마지막 pop()으로 10이 제거되어 최종 상태는 아래부터 3, 5, 8이 된다.

왜 이 순서인가: peek()는 top 원소를 읽기만 하고, 마지막 pop()은 직전에 들어간 10을 제거하므로 최종 스택에는 3, 5, 8만 남는다.

선지별 해설

1번: X. 이 그림은 최종 상태를 아래부터 3, 5만 남은 것으로 본다. push(8) 이후 peek()는 상태를 바꾸지 않으므로 8이 그대로 남아 있어야 한다. 틀린 표현은 8이 없는 최종 상태이고, 올바르게는 아래부터 3, 5, 8이 있어야 한다.

2번: O. 아래부터 3, 5, 8이 차 있고 맨 윗칸 하나가 비어 있는 상태이다. push(10)까지 하면 가득 차지만 마지막 pop()으로 10이 빠지므로 용량 4 스택에 3, 5, 8만 남는다.

3번: X. 이 그림은 아래부터 5, 8, 10이 남은 것으로 본다. 중간의 pop()으로 7이 제거된 뒤 3은 계속 바닥에 남아 있고, 마지막 pop()으로 10은 제거된다. 틀린 표현은 3이 사라지고 10이 남아 있는 점이며, 올바르게는 아래부터 3, 5, 8이 남는다.

4번: X. 이 그림은 마지막 상태를 아래부터 3, 5, 8, 10으로 본다. 그러나 마지막 연산이 pop()이므로 top인 10은 최종 상태에 남을 수 없다. 틀린 표현은 10이 남아 있는 점이고, 올바르게는 10이 제거된 아래부터 3, 5, 8 상태이다.

오답 포인트

peek()를 pop()처럼 착각하거나 마지막 pop()을 놓치면 4번처럼 10이 남는다고 보기 쉽다. 또 중간 pop()이 7만 제거한다는 점을 놓치면 3번처럼 바닥의 3이 사라진 것으로 잘못 판단하기 쉽다.

다시 나오면 이렇게 판단

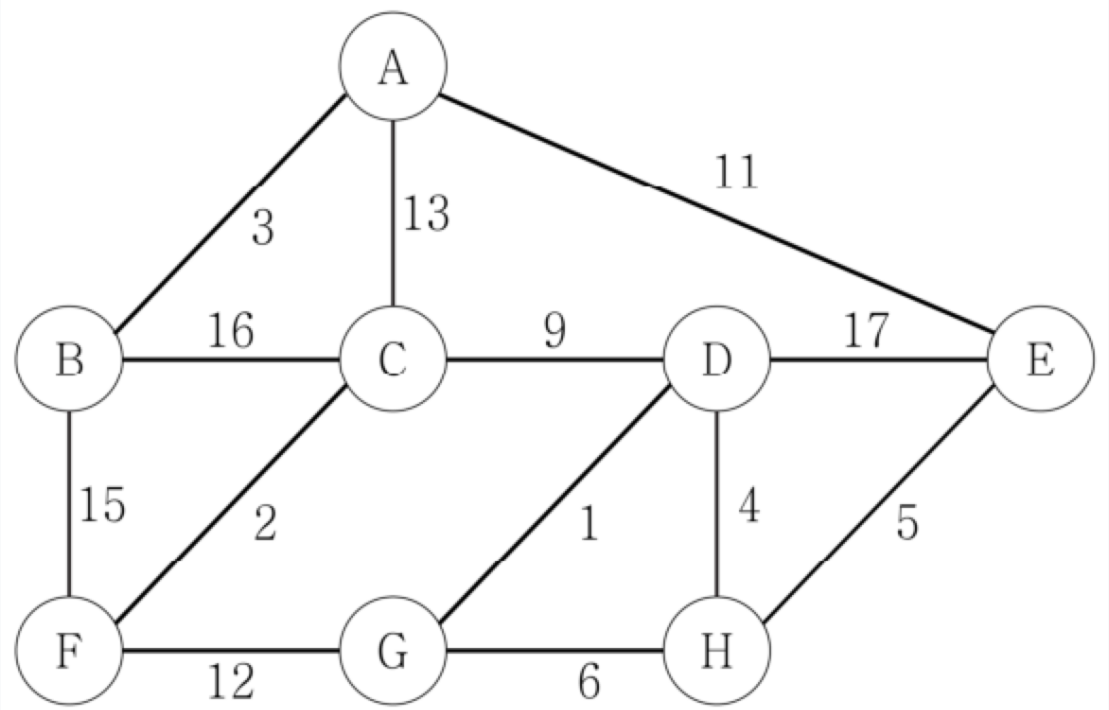
스택 문제는 연산을 한 줄씩 따라가면서 push는 top에 추가, pop은 현재 top 제거, peek는 조회만 하고 상태 변화 없음으로 구분해 기록하면 된다. 마지막 연산이 무엇인지까지 확인해야 최종 상태를 정확히 고를 수 있다.

암기 포인트

스택은 나중에 넣은 것이 먼저 나온다. peek는 그대로 두고, pop만 뺀다.

NoteBase

문 6. 그림은 정점 A ~ H 간의 거리를 나타낸 그래프이다. 다익스트라 (Dijkstra) 알고리즘을 이용하여 정점 A로부터 다른 모든 정점까지의 최단 경로를 구하고, 각 최단 경로의 거리를 짧은 것부터 순서대로 나열한 것은?



① 3, 11, 13, 15, 16, 20, 21

② 3, 11, 13, 15, 16, 21, 22

③ 3, 11, 13, 16, 18, 20, 21

④ 3, 11, 13, 16, 18, 21, 22

정답 1번

출제 포인트

다익스트라 알고리즘으로 정점 A에서 각 정점까지의 최단거리를 갱신하고, 그 최종 거리들을 오름차순으로 나열할 수 있는지를 묻는 문제이다.

조건 정리

A에서 직접 갈 수 있는 정점은 B=3, C=13, E=11이다.
최단거리를 줄이는 핵심 간선은 C-F=2, E-H=5, H-D=4, D-G=1이다.
직접 연결보다 우회 경로가 더 짧아지는 정점을 찾아야 한다.

NoteBase

사용할 공식

다익스트라 알고리즘은 아직 확정되지 않은 정점 중 현재 거리가 가장 짧은 정점을 하나씩 확정하면서 인접 정점의 거리를 갱신하는 방식이다.
이 문제는 모든 간선 가중치가 양수이므로 다익스트라 알고리즘을 그대로 적용할 수 있다.

계산 과정

초기값은 B=3, C=13, E=11이다.
B를 확정하면 F=18이 된다.
E를 확정하면 H=16, D=28이 된다.
C를 확정하면 F는 15로 줄고, D는 22로 줄어든다.
F를 확정하면 G=27이 된다.
H를 확정하면 D는 20으로 줄고, G는 22로 줄어든다.
D를 확정하면 G는 21로 다시 줄어든다.
따라서 A에서 각 정점까지의 최단거리는 B=3, E=11, C=13, F=15, H=16, D=20, G=21이다.

선지별 해설

① O: 최단거리들을 오름차순으로 쓰면 3, 11, 13, 15, 16, 20, 21이다. B는 3, E는 11, C는 13, F는 A-C-F로 15, H는 A-E-H로 16, D는 A-E-H-D로 20, G는 A-E-H-D-G로 21이므로 맞다.
② X: 마지막 값 22가 틀리다. G의 최단거리는 A-E-H-D-G 경로로 11+5+4+1=21이므로 22가 아니다. 틀린 표현은 마지막의 22이고, 올바르게는 21이다.
③ X: 다섯 번째 값 18이 틀리다. F의 최단거리는 A-B-F의 18이 아니라 A-C-F의 13+2=15이다. 틀린 표현은 18이고, 올바르게는 15이므로 전체 나열은 3, 11, 13, 15, 16, 20, 21이다.
④ X: 18과 22가 모두 틀리다. F는 15, G는 21이므로 이 선지는 우회 경로 갱신을 반영하지 못했다. 틀린 표현은 18, 22이고, 올바르게는 15, 21이다.

오답 포인트

직접 보이는 간선이나 처음 계산한 값만 유지하면 오답이 된다. 특히 F를 A-B-F=18로 고정하거나, G를 H를 거친 뒤 D를 다시 거쳐 21로 더 줄일 수 있다는 점을 놓치면 ③, ④, ② 같은 오답이 나온다.

다시 나오면 이렇게 판단

다익스트라 문제는 현재 가장 짧은 거리의 정점을 하나씩 확정하고, 그때마다 인접 정점의 값을 다시 계산해 더 짧아지면 즉시 갱신하는 순서로 판단한다. 초기값을 끝까지 붙들지 않고 갱신 여부를 끝까지 확인하면 된다.

암기 포인트

다익스트라는 가장 짧은 정점부터 확정하고, 확정할 때마다 주변 거리를 다시 줄여 본다는 방식으로 기억한다.

NoteBase

문 7. 다음 데이터를 차례대로 하나씩 입력받아 AVL 트리를 생성할 때, 필요한 회전을 적용 순서대로 바르게 나열한 것은?

6, 7, 8, 2, 1, 5, 4

① LL회전, LR회전, LR회전
② LL회전, LR회전, RL회전
③ RR회전, LL회전, LR회전
④ RR회전, LL회전, RL회전

정답 3번

출제 포인트

이 문제는 AVL 트리에 키를 순차 삽입할 때 처음 불균형이 생기는 조상 노드를 찾고, 그때 필요한 회전 유형을 순서대로 판단하는지를 묻는 문제이다.

올바른 순서

1. 8을 삽입하면 6-7-8이 오른쪽-오른쪽으로 이어져 6에서 RR회전이 일어난다.
2. 이어서 2와 1을 삽입하면 6의 왼쪽으로 2, 1이 연속되어 6에서 LL회전이 일어난다.
3. 그다음 5를 삽입하면 7의 왼쪽 부분트리에서 왼쪽-오른쪽 불균형이 생겨 7에서 LR회전이 일어난다.
4. 마지막 4를 삽입할 때는 균형 인수가 허용 범위를 넘지 않아 추가 회전이 없다.

왜 이 순서: AVL 삽입은 새 노드가 들어간 뒤 처음 불균형이 된 조상을 즉시 복구하므로, 실제 발생 순서는 RR -> LL -> LR로 고정된다.

NoteBase

선지별 해설

① X
- 틀린 이유: 첫 회전을 LL회전으로 본 부분이 틀리다. 6, 7, 8의 연속 삽입은 오른쪽-오른쪽 불균형이므로 RR회전이 먼저 일어난다.
- 올바르게는: RR회전, LL회전, LR회전이다.

② X
- 틀린 이유: 첫 회전을 LL회전으로 본 부분과 마지막 회전을 RL회전으로 본 부분이 틀리다. 5 삽입 뒤의 마지막 불균형은 7의 왼쪽 자식 쪽에서 생기는 좌우형이므로 LR회전이다.
- 올바르게는: RR회전, LL회전, LR회전이다.

③ O
- RR회전, LL회전, LR회전의 순서가 실제 재균형 과정과 일치한다. 8 삽입 시 RR, 1 삽입 시 LL, 5 삽입 시 LR이 발생하고 4 삽입 때는 추가 회전이 없다.

④ X
- 틀린 이유: 마지막 회전을 RL회전으로 본 부분이 틀리다. RL회전은 오른쪽 자식의 왼쪽 서브트리 때문에 생기는데, 이 문제의 마지막 불균형은 왼쪽 자식의 오른쪽 서브트리 때문에 생긴다.
- 올바르게는: RR회전, LL회전, LR회전이다.

오답 포인트

RR회전과 LL회전은 연속 방향만 보면 되지만, LR회전과 RL회전은 불균형 조상에서 새 노드까지의 경로를 두 단계로 끝까지 따라가야 한다. 이 문제는 마지막 단계에서 5가 7의 왼쪽 부분트리의 오른쪽 가지에 놓인다는 점을 놓치면 RL회전으로 잘못 보기 쉽다.

다시 나오면 이렇게 판단

AVL 회전 순서 문제는 각 삽입 직후 처음 불균형이 된 조상을 찾고, 새 노드가 그 조상의 왼쪽-왼쪽, 오른쪽-오른쪽, 왼쪽-오른쪽, 오른쪽-왼쪽 중 어느 경로에 놓였는지로 LL, RR, LR, RL을 바로 분류하면 된다.

암기 포인트

AVL 회전 이름은 불균형 조상에서 새 노드까지의 두 단계 경로를 그대로 읽는다. 오른쪽-오른쪽이면 RR, 왼쪽-왼쪽이면 LL, 왼쪽-오른쪽이면 LR이다.

NoteBase

문 8. 다음 C 코드는 원형 연결 리스트에 속한 모든 노드의 값을 더하여 출력한다. (가) ~ (다)에 들어갈 내용을 바르게 연결한 것은?

```
#include <stdio.h>
#include <malloc.h>
typedef struct Node {
   struct Node *a;
   int b;
} Node;

int listSum(Node *p) {
   int sum = 0;
   Node *s = p;
   do {
      sum += [ (가) ] ;
      p = [ (나) ] ;
   } while ( [ (다) ] );
   return sum;
}

int main() {
   int MAX = 5;
   int i, s;
   Node *p[MAX];

   for( i = 0; i < MAX; i++) {
      p[i] = (Node*)malloc(sizeof(Node));
      p[i]->b = i+1;
   }
   for( i = 0; i < MAX-1; i++)
      p[i]->a = p[i+1];
   p[MAX-1]->a = p[0];
   s = listSum(p[0]);
   printf("Sum = %d", s);
}
```

① (가): p.b, (나): p.a, (다): p != NULL

② (가): p->a, (나): p->b, (다): p != s

③ (가): p->b, (나): p->a, (다): p != NULL

④ (가): (*p).b, (나): (*p).a, (다): p != s

정답 4번

출제 포인트

이 문제는 포인터가 가리키는 구조체 멤버 접근 방식과 원형 연결 리스트의 순회 종료 조건을 함께 묻는 문제이다.

핵심 개념

구조체 포인터 `p`의 정수 멤버 `b`에 접근할 때는 `p->b` 또는 `(*p).b`를 사용한다.
구조체 포인터 `p`의 다음 노드 포인터 멤버 `a`에 접근할 때는 `p->a` 또는 `(*p).a`를 사용한다.
원형 연결 리스트는 마지막 노드가 다시 첫 노드를 가리키므로 종료 기준이 `NULL`이 아니라 시작 노드 재도달 여부이다.

NoteBase

선지별 해설

① X: `(가)`의 `p.b`는 포인터 변수에 점 연산자를 써서 잘못되었다. 올바르게는 `(*p).b` 또는 `p->b`이다. 또한 `(나)`의 `p.a`도 잘못되었다. 올바르게는 `(*p).a` 또는 `p->a`이다.
`(다)`의 `p != NULL`도 틀리다. 원형 연결 리스트는 마지막 노드의 다음이 `NULL`이 아니라 시작 노드이므로 종료되지 않는다.
② X: `(가)`의 `p->a`는 다음 노드의 주소를 더하게 되므로 합계 대상인 정수 값이 아니다. 올바르게는 `p->b` 또는 `(*p).b`이다.
`(나)`의 `p->b`는 정수 멤버인데 포인터 변수 `p`에 대입할 수 없다. 올바르게는 `p->a` 또는 `(*p).a`이다.
`(다)`의 `p != s` 자체는 원형 리스트 종료 조건으로 맞을 수 있으나, 앞의 두 칸이 이미 틀려 전체 선지는 오답이다.
③ X: `(가)`의 `p->b`와 `(나)`의 `p->a`는 각각 현재 노드 값 누적과 다음 노드 이동으로 맞다.
하지만 `(다)`의 `p != NULL`이 틀리다. 이 리스트는 `p[MAX-1]->a = p[0]`로 원형이므로 `NULL`을 만나지 않는다.
④ O: `(가)`의 `(*p).b`는 현재 노드의 값 `b`를 더하는 올바른 표현이다.
`(나)`의 `(*p).a`는 다음 노드로 이동하는 올바른 표현이다.
`(다)`의 `p != s`는 한 바퀴를 돌고 시작 노드 `s`로 돌아오기 전까지 반복하게 하므로 모든 노드를 정확히 한 번씩 더한다.

오답 포인트

출제자가 노린 함정은 두 가지이다. 첫째, 구조체 변수 접근 `.`과 구조체 포인터 접근 `->`를 섞어 쓰게 만드는 함정이다. 둘째, 선형 연결 리스트의 종료 조건인 `NULL` 검사를 원형 연결 리스트에도 그대로 적용하게 만드는 함정이다.

다시 나오면 이렇게 판단

포인터가 가리키는 구조체 멤버인지부터 확인하고, 값 필드인지 링크 필드인지 구분한다. 그다음 리스트가 선형인지 원형인지 확인해서 종료 조건이 `NULL`인지 시작 노드 재도달인지 판단한다.

암기 포인트

구조체 포인터 멤버 접근은 `p->field = (*p).field`이고, 원형 연결 리스트의 종료 기준은 `NULL`이 아니라 시작점 복귀이다.

NoteBase

문 9. 다음 C 코드를 수행하면 4개의 값이 출력된다. 출력 값 중 나머지 3개와 다른 값을 출력하는 C 코드의 라인 번호는?

라인 번호	C 코드
1	`#include <stdio.h>`
2	
3	`int main() {`
4	`    int array[] = {0,1,2};`
5	`    int *p = array+2;`
6	
7	`    printf("%d\n", *(p-1));`
8	`    printf("%d\n", *p-1);`
9	`    printf("%d\n", *(p--));`
10	`    printf("%d\n", *p);`
11	`}`

① 7

② 8

③ 9

④ 10

정답 3번

출제 포인트

이 문제는 포인터 산술, 역참조의 결합 방식, 후위 감소 연산의 평가 시점을 구분할 수 있는지를 묻는다.

사례 정리

array는 {0, 1, 2}이고 p는 처음에 array[2]를 가리킨다.
7행은 p를 한 칸 앞 원소 쪽으로 이동한 주소를 읽는다.
8행은 현재 p가 가리키는 값을 읽은 뒤 1을 빼고, 9행은 현재 값을 출력한 뒤 p를 한 칸 감소시킨다.

NoteBase

핵심 개념

*(p-1)은 p를 먼저 한 칸 이동한 뒤 그 위치의 값을 읽는 식이다.
*p-1은 (*p)-1로 해석되므로 현재 p가 가리키는 값에서 1을 뺀 결과이다.
*(p--)는 후위 감소 연산이므로 현재 p의 값을 먼저 사용하고, 식 평가 뒤 p 자체를 1 감소시킨다.

적용 기준

7행의 *(p-1)은 array[1]을 읽으므로 1이다.
8행의 *p-1은 (*p)-1이므로 2-1=1이다.
9행의 *(p--)는 감소 전 p가 가리키는 array[2]의 값 2를 출력하고, 그 뒤 p는 array[1]을 가리킨다.
10행의 *p는 감소된 뒤의 p를 읽으므로 다시 1이다.

선지별 해설

1번(7행): X. 7행의 *(p-1)은 array[1]을 읽어 1을 출력하므로 8행, 10행과 같다. 틀린 표현은 "7행이 다른 값을 출력한다"이고 올바르게는 "7행은 1을 출력한다"이다.
2번(8행): X. 8행의 *p-1은 *(p-1)이 아니라 (*p)-1로 계산된다. 현재 *p는 2이므로 결과는 1이다. 틀린 표현은 "8행이 다른 값을 출력한다"이고 올바르게는 "8행도 1을 출력한다"이다.
3번(9행): O. 9행의 *(p--)는 후위 감소라서 현재 p가 가리키는 array[2]의 값 2를 먼저 출력하고, 그 다음 p를 array[1]로 내린다. 따라서 9행만 2를 출력해 나머지 세 줄과 다르다.
4번(10행): X. 10행은 9행 실행 뒤 p가 array[1]을 가리키므로 1을 출력한다. 틀린 표현은 "10행이 다른 값을 출력한다"이고 올바르게는 "10행은 1을 출력한다"이다.

오답 포인트

*p-1을 *(p-1)로 오해하거나, p--가 먼저 감소한다고 착각하면 8행과 9행의 결과를 뒤바꾸게 된다.

다시 나오면 이렇게 판단

포인터 식은 괄호 위치와 후위 연산의 평가 시점을 먼저 본다. 현재 주소를 쓰는지, 이동한 뒤 주소를 쓰는지, 값을 읽은 뒤 감소하는지를 한 줄씩 분리하면 된다.

암기 포인트

*(p-1)은 한 칸 옆 값을 읽고, *p-1은 현재 값에서 1을 빼며, *(p--)는 현재 값을 읽고 나서 포인터를 내린다.

NoteBase

문 10. 다음 tree_height() 함수는 이진 트리의 루트 노드를 매개변수로 받아 트리의 높이를 반환한다. (가) ~ (다)에 들어갈 내용을 바르게 연결한 것은?

```
typedef struct node {
   int value;
   struct node *left;
   struct node *right;
} node;

int tree_height (node *ptr) {
   int left_h, right_h;
   if (ptr == NULL)
      return 0;
   else {
      left_h = tree_height(ptr->left);
      right_h = tree_height(ptr->right);
      if ( [ (가) ] )
         return [ (나) ] ;
      else
         return [ (다) ] ;
   }
}
```

① (가): left_h > right_h, (나): left_h, (다): right_h

② (가): left_h > right_h, (나): left_h + 1, (다): right_h + 1

③ (가): left_h < right_h, (나): left_h, (다): right_h

④ (가): left_h < right_h, (나): left_h + 1, (다): right_h + 1

정답 2번

출제 포인트

이 문제는 이진 트리의 높이를 재귀식으로 구현할 때 현재 노드의 1단계와 두 서브트리 높이의 최댓값을 함께 반영할 수 있는지를 묻는다.

사례 정리

빈 포인터는 높이 0으로 처리한다.
현재 노드의 높이는 왼쪽 높이와 오른쪽 높이 중 큰 값에 현재 노드 1단계를 더해 계산한다.

핵심 개념

이 함수의 핵심 재귀식은 `height(ptr) = max(height(ptr->left), height(ptr->right)) + 1`이다.
따라서 더 큰 서브트리 높이를 고른 뒤 현재 노드를 포함시키는 `+1`이 반드시 필요하다.

NoteBase

선지별 해설

1번 X: `(가)`를 `left_h > right_h`로 두는 방향은 맞지만 `(나)`와 `(다)`가 각각 `left_h`, `right_h`라서 현재 노드 1단계를 더하지 못한다. 이 선지는 서브트리 높이만 그대로 반환한다. 올바르게는 `return left_h + 1;`, `return right_h + 1;`이어야 한다.
2번 O: `(가)`를 `left_h > right_h`로 두면 더 큰 서브트리 높이를 선택할 수 있고, `(나)`와 `(다)`를 각각 `left_h + 1`, `right_h + 1`로 두면 현재 노드가 포함된다. 이는 `max(left_h, right_h) + 1`을 그대로 구현한 형태이다.
3번 X: `(가)`를 `left_h < right_h`로 두면 오른쪽 높이가 더 클 때 오히려 `left_h`를 반환하게 된다. 여기에 `+1`도 빠져 있어 현재 노드도 반영하지 못한다. 올바르게는 큰 쪽을 고르고 그 값에 1을 더해야 한다.
4번 X: 반환식에 `+1`을 붙인 점은 맞지만 `(가)`가 `left_h < right_h`이어서 작은 높이를 선택하는 구조가 된다. 이 선지는 `min(left_h, right_h) + 1`에 가깝다. 올바르게는 `left_h > right_h`일 때 `left_h + 1`, 그 외에는 `right_h + 1`을 반환해야 한다.

오답 포인트

트리 높이를 계산할 때 `max`만 떠올리고 현재 노드 1단계를 빼먹거나, 비교식 방향과 반환 대상을 함께 보지 않고 넘어가면 틀리기 쉽다. 특히 4번은 `+1`이 보여서 맞아 보이지만 실제로는 더 작은 높이를 반환한다.

다시 나오면 이렇게 판단

재귀로 높이, 깊이, 크기 같은 값을 구하는 코드는 먼저 종료조건을 확인하고, 그다음 반환식이 `현재 노드 반영값 + 하위 문제 결과` 구조인지 본다. 높이 문제이면 최종 형태가 `max(왼쪽, 오른쪽) + 1`인지 확인하면 된다.

암기 포인트

트리 높이 재귀는 `빈 트리 0, 현재 높이 = 큰 쪽 + 1`로 정리된다.

NoteBase

문 11. 10개의 정점을 가진 무방향 그래프가 가질 수 있는 간선의 최대 개수는? (단, 각 정점에서 자기 자신으로의 간선은 허용하지 않고, 두 정점 사이의 간선은 최대 1개이다)

① 42

② 43

③ 44

④ 45

정답 4번

출제 포인트

단순 무방향 그래프에서 최대 간선 수가 $\binom{n}{2} = \frac{n(n-1)}{2}$임을 적용하는 문제이다.

조건 정리

주어진 값: 정점 수는 10개이다.
구해야 할 값: 가질 수 있는 간선의 최대 개수이다.
주의할 조건: 자기 자신으로의 간선은 없고, 두 정점 사이의 간선은 최대 1개이므로 단순 무방향 그래프이다.

사용할 공식

공식은 $\binom{n}{2} = \frac{n(n-1)}{2}$이다.
이 공식을 쓰는 이유는 무방향 그래프의 최대 간선 수가 서로 다른 두 정점을 한 번씩 짝짓는 경우의 수와 같기 때문이다.

계산 과정

1. 정점 10개 중 서로 다른 두 정점을 고르는 경우의 수를 구한다.
2. $\binom{10}{2} = \frac{10 \times 9}{2}$로 계산한다.
3. 결과는 45이므로 최대 간선 수는 45개이다.

선지 판단

1번: X. 최대 간선 수는 45이므로 42는 틀리다. 틀린 표현은 값이 42라는 점이며, 올바르게는 45이다. 42는 경우의 수 계산 자체를 잘못했거나 일부 정점 쌍을 빠뜨린 값이다.
2번: X. 최대 간선 수는 45이므로 43은 틀리다. 틀린 표현은 값이 43이라는 점이며, 올바르게는 45이다. 43은 $\frac{10 \times 9}{2}$ 계산을 끝까지 하지 않거나 중간에 감산을 잘못한 경우 나올 수 있다.
3번: X. 최대 간선 수는 45이므로 44는 틀리다. 틀린 표현은 값이 44라는 점이며, 올바르게는 45이다. 44는 정점 쌍 하나를 누락해 세는 실수에서 나올 수 있다.
4번: O. $\binom{10}{2} = 45$이므로 최대 간선 수를 정확히 나타낸다.

NoteBase

오답 포인트

무방향 그래프의 최대 간선 수를 순서 있는 쌍으로 보지 않고, 서로 다른 두 정점의 조합으로 본다는 점이 핵심이다. 자기 루프 금지와 중복 간선 금지를 놓치면 공식을 잘못 적용한다.

다시 나오면 이렇게 판단

정점 수가 주어지고 최대 간선 수를 묻는 문제는 먼저 자기 루프와 중복 간선 허용 여부를 확인한 뒤, 단순 무방향 그래프이면 바로 $\binom{n}{2}$를 적용해 판단한다.

암기 포인트

단순 무방향 그래프의 최대 간선 수는 서로 다른 두 정점의 조합 수인 $\frac{n(n-1)}{2}$이다.

NoteBase

문 12. 회문(palindrome)은 영어 단어 radar와 같이 순서를 거꾸로 뒤집어도 원본과 동일한 단어를 의미한다. 다음 C 코드의 isPalindrome() 함수는 입력 문자열이 회문인 경우 1을 반환하고, 그렇지 않은 경우는 0을 반환한다. (가), (나)에 들어갈 내용을 바르게 연결한 것은?

```
#include <stdio.h>
char stack[100];
int top = -1;

void push(char ch) {stack[++top] = ch;}

char pop() {return stack[top--];}

int isEmpty() {return top == -1;}

int isPalindrome(char *str) {
   int i;
   for ( i = 0; str[i]; i++ )
      [      (가)      ] ;

   for ( i = 0; !isEmpty(); i++ )
      if ( [      (나)      ] )
         return 0;
   return 1;
}

int main() {
   char input[100];
   scanf("%s", input);
   printf("%s",
      isPalindrome(input) ? "True" : "False");
}
```

① (가): push(str[i]), (나): str[i] == pop()

② (가): push(str[i]), (나): str[i] != pop()

③ (가): push(str[top-i]), (나): str[i] == pop()

④ (가): push(str[top-i]), (나): str[i] != pop()

정답 2번

출제 포인트

이 문제는 스택의 후입선출 특성을 이용해 문자열을 역순으로 꺼내고, 이를 원래 문자열과 비교하여 회문 여부를 판정하는 흐름을 묻는 문제이다.

핵심 개념

회문은 앞에서 읽은 문자열과 뒤에서 읽은 문자열이 같은 문자열이다.
스택은 후입선출 구조이므로 문자를 입력 순서대로 push하면 pop할 때는 역순으로 나온다.
따라서 모든 문자를 push한 뒤 str[i]와 pop() 값을 차례로 비교할 때, 다른 순간이 있으면 회문이 아니다.

NoteBase

선지별 해설

① X: (가)의 push(str[i])는 맞지만 (나)의 "str[i] == pop()"가 틀렸다. 두 값이 같으면 계속 진행해야 하므로 return 0의 조건은 불일치여야 한다. 틀린 표현은 "=="이며, 올바르게는 "str[i] != pop()"이다.
② O: (가)에서 각 문자를 push(str[i])로 스택에 넣으면 pop()에서 역순으로 꺼낼 수 있다. (나)에서 str[i] != pop()일 때만 0을 반환해야 회문이 아닌 경우를 정확히 걸러낼 수 있다.
③ X: (가)의 "push(str[top-i])"가 틀렸다. top은 push 과정에서 변하는 스택 인덱스이므로 문자열 인덱스로 쓰면 처음부터 음수 인덱스가 될 수 있고, 원문을 순서대로 넣는 구조도 깨진다. 틀린 표현은 "str[top-i]"이며, 올바르게는 "str[i]"이다. (나)의 비교식 자체는 맞아도 (가)가 잘못되어 전체가 성립하지 않는다.
④ X: (가)의 "push(str[top-i])"가 틀렸다. 스택은 원문을 그대로 넣고 pop에서 역순을 얻어야 하므로, 올바르게는 "push(str[i])"이다. (나)의 "str[i] != pop()"는 맞지만 한 쌍 전체로는 정답이 아니다.

오답 포인트

스택이 역순을 만들어 준다는 사실을 놓치면 (가)에서 미리 역방향 인덱스를 써야 한다고 착각하기 쉽다. 또 return 0이 나오는 조건을 '같을 때'로 뒤집으면 회문 판정이 반대로 동작한다.

다시 나오면 이렇게 판단

문자열을 스택에 순서대로 넣고 다시 꺼내는 코드는 '원문 대 역순 비교'로 읽는다. 따라서 입력 단계는 str[i]를 그대로 push하고, 판정 단계는 원문 문자와 pop() 값이 다를 때 실패하도록 조건을 세우면 된다.

암기 포인트

회문 스택 판별은 '원문 push, 역순 pop, 다르면 실패'이다.

NoteBase

문 13. 다음 C 코드는 이진 탐색을 이용하여 정수 데이터를 탐색하는 함수이다. (가), (나)에 들어갈 내용을 바르게 연결한 것은? (단, key는 찾고자 하는 값, a[]는 오름차순으로 정렬된 정수 배열, n은 배열의 크기이다)

```
int search(int key, int a[], int n) {
    int mid;
    int left = 0;
    int right = n-1;
    while ( (가) ) {
        mid = (right + left) / 2;
        if (key == a[mid]) return mid;
        if ( (나) ) right = mid - 1;
        else left = mid + 1;
    }
    return -1;
}
```

① (가): left < right, (나): key < a[mid]

② (가): left < right, (나): key > a[mid]

③ (가): left <= right, (나): key < a[mid]

④ (가): left <= right, (나): key > a[mid]

정답 3번

출제 포인트

이 문제는 오름차순 배열에서 이진 탐색을 구현할 때 반복 종료 조건과 좌우 분기 조건을 정확히 고를 수 있는지를 묻는다.

핵심 개념

이진 탐색은 탐색 구간을 left와 right로 잡고, 구간이 남아 있는 동안만 반복한다. 오름차순 배열에서는 key가 a[mid]보다 작으면 왼쪽 구간을 남겨야 하므로 right를 mid - 1로 줄이고, key가 a[mid]보다 크면 오른쪽 구간을 남겨야 하므로 left를 mid + 1로 늘린다.

NoteBase

선지별 해설

1번: X. (가)의 "left < right"가 틀렸다. left와 right가 같아지는 마지막 한 칸도 검사해야 하므로 반복 조건은 "left <= right"이어야 한다. (나)의 "key < a[mid]"는 맞다.
2번: X. (가)의 "left < right"가 틀려 마지막 후보를 검사하지 못한다. (나)의 "key > a[mid]"도 틀렸다. key가 a[mid]보다 크면 오른쪽을 탐색해야 하므로 else 쪽에서 left = mid + 1이 되어야 하고, if 조건은 "key < a[mid]"이어야 한다.
3번: O. (가)의 "left <= right"는 탐색 구간이 한 칸 남은 경우까지 검사하게 하므로 맞다. (나)의 "key < a[mid]"도 맞다. 오름차순 배열에서 key가 중간값보다 작으면 왼쪽 절반만 남겨야 하므로 right = mid - 1로 갱신한다.
4번: X. (가)의 "left <= right"는 맞지만, (나)의 "key > a[mid]"가 틀렸다. key가 a[mid]보다 큰 경우는 오른쪽 절반으로 가야 하므로 left = mid + 1이 맞고, right = mid - 1을 수행하는 조건은 "key < a[mid]"로 고쳐야 한다.

오답 포인트

마지막 원소 하나가 남았을 때도 비교가 필요하다는 점을 놓치면 "left < right"를 고르게 된다. 또한 오름차순 배열이라는 조건을 잊으면 "작으면 왼쪽, 크면 오른쪽"이라는 분기 방향을 반대로 연결하기 쉽다.

다시 나오면 이렇게 판단

이진 탐색 코드를 보면 먼저 배열 정렬 방향을 확인하고, 다음에 "작을 때 어느 쪽 절반을 버리는가"를 본다. 그 뒤 반복 조건이 마지막 한 칸까지 포함하는지 확인하면 정답을 고를 수 있다.

암기 포인트

오름차순 이진 탐색은 "작으면 right 감소, 크면 left 증가, 반복은 left <= right"이다.

NoteBase

문 14. 다음 C 코드는 반복구조를 이용하여 n번째 피보나치 수를 구하는 함수이다. (가), (나)에 들어갈 내용을 바르게 연결한 것은? (단, n은 0 이상의 정수이다)

```
int fibonacci(int n){
    int x = 0, y = 1, z = 0;
    if (n < 2) return n;
    while ( (가) ) {
        (나) ;
        x = y;
        y = z;
    }
    return z;
}
```

① (가): n-- > 1, (나): z = x + y

② (가): n-- > 1, (나): z = x - y

③ (가): n-- >= 1, (나): z = x + y

④ (가): n-- <= 1, (나): z = x - y

정답 1번

출제 포인트

이 문제는 반복문으로 피보나치 수를 갱신할 때 반복 횟수와 다음 항 계산식을 정확히 대입할 수 있는지를 묻는 문제이다.

사례 정리

초깃값은 x=0, y=1, z=0이다.
반복 1회마다 z에 다음 피보나치 수를 만든 뒤 x=y, y=z로 한 칸씩 전진한다.
따라서 (나)는 두 직전 값의 합이어야 하고, n<2를 이미 반환했으므로 반복문은 n=2일 때 정확히 1번만 실행되어야 한다.

NoteBase

핵심 개념

피보나치 수열은 앞의 두 항의 합으로 다음 항을 만든다.
반복형 구현에서는 x와 y를 연속된 두 항으로 두고 z=x+y를 계산한 다음, x=y, y=z로 갱신한다.

적용 기준

빈칸 판단의 핵심은 두 가지이다.
첫째, 다음 항 계산식은 반드시 z = x + y이다.
둘째, n=2에서 한 번만 돌고 n=3에서 두 번 도는 조건이어야 하므로 while 조건은 n-- > 1이어야 한다.

선지별 해설

① O: while (n-- > 1)은 n=2에서 1회, n=3에서 2회 실행되어 필요한 횟수만큼 반복된다. z = x + y도 피보나치의 점화식과 일치한다.
② X: 틀린 부분은 z = x - y이다. 피보나치 수는 앞의 두 항의 차가 아니라 합으로 다음 항을 만드므로 올바르게는 z = x + y이다.
③ X: 틀린 부분은 n-- >= 1이다. 이 조건은 n=2일 때 두 번 실행되어 z가 2가 되므로 한 번 더 돈다. 올바르게는 n-- > 1이다.
④ X: 틀린 부분은 n-- <= 1과 z = x - y이다. n이 2 이상일 때 반복을 시작해야 하므로 <= 1은 방향부터 반대이고, 다음 항 계산도 차가 아니라 합이다. 올바르게는 n-- > 1, z = x + y이다.

오답 포인트

③처럼 >와 >=의 차이를 놓치면 반복 횟수가 한 번 더 늘어난다. ②와 ④는 피보나치의 핵심 규칙이 합이라는 점을 차로 바꾸어 놓은 함정이다.

다시 나오면 이렇게 판단

반복문 빈칸 문제는 먼저 초깃값이 무엇을 뜻하는지 확인하고, n=2나 n=3 같은 가장 작은 사례를 직접 대입해 반복 횟수와 갱신식이 맞는지 검산하면 된다.

암기 포인트

반복형 피보나치는 다음 항은 합, 반복 횟수는 n=2에서 1번이라는 기준으로 판단한다.

NoteBase

문 15. 다음 설명에 해당하는 노드 수 A, B, C의 합은? (단, 트리의 높이는 트리의 최대 레벨을 의미하며, 루트 노드의 레벨을 1로 한다)

◯ A = 높이가 5인 완전 이진 트리가 가질 수 있는 최소 노드 수
◯ B = 높이가 6인 포화 이진 트리의 노드 수
◯ C = 높이가 7인 완전 이진 트리가 가질 수 있는 최대 노드 수

① 142
② 206
③ 209
④ 222

정답 2번

출제 포인트

완전 이진 트리의 최소 노드 수와 포화 이진 트리의 노드 수 공식을 구분하여 합을 계산하는 문제이다.

조건 정리

A는 높이 5인 완전 이진 트리가 가질 수 있는 최소 노드 수이다.
B는 높이 6인 포화 이진 트리의 노드 수이다.
C는 높이 7인 완전 이진 트리가 가질 수 있는 최대 노드 수이다.
트리의 높이는 루트 노드의 레벨을 1로 하는 최대 레벨이다.

사용할 공식

완전 이진 트리의 높이가 h일 때 최소 노드 수는 2^{h-1}이다.
포화 이진 트리의 높이가 h일 때 노드 수는 $2^h - 1$이다.
완전 이진 트리의 높이가 h일 때 최대 노드 수는 $2^h - 1$이다.
이 공식을 쓰는 이유는 높이와 완전/포화 조건만으로 노드 수의 최소값과 최대값이 결정되기 때문이다.

계산 과정

1. $A = 2^{5-1} = 16$이다. 높이 5를 유지하는 완전 이진 트리의 최소 형태는 1레벨부터 4레벨까지는 모두 차고 5레벨에 노드가 1개만 있는 경우이다.
2. $B = 2^6 - 1 = 63$이다. 포화 이진 트리는 모든 레벨이 가득 찬다.
3. $C = 2^7 - 1 = 127$이다. 완전 이진 트리의 최대 노드 수는 같은 높이의 포화 이진 트리 노드 수와 같다.
4. 따라서 $A + B + C = 16 + 63 + 127 = 206$이다.

NoteBase

선지별 해설

① X: 계산 결과가 206이므로 142는 맞지 않다. 완전 이진 트리의 최소값이나 최대값 공식을 잘못 적용한 오답이다. 올바르게는 합이 206이다.
② O: $A = 16, B = 63, C = 127$이므로 합은 206이다.
③ X: 계산 결과가 206이므로 209는 맞지 않다. 완전 이진 트리의 최소 노드 수를 마지막 레벨까지 꽉 찬 경우와 혼동하면 이런 오차가 생긴다. 올바르게는 합이 206이다.
④ X: 계산 결과가 206이므로 222는 맞지 않다. 높이와 레벨 수를 혼동하여 2^h 계열 값을 크게 잡으면 나오는 오답이다. 올바르게는 합이 206이다.

오답 포인트

완전 이진 트리와 포화 이진 트리를 같은 개념으로 처리하면 A를 틀리기 쉽다. 완전 이진 트리의 최소 노드 수는 마지막 레벨에 노드가 1개만 있어도 높이가 유지된다는 점이 핵심 함정이다.

다시 나오면 이렇게 판단

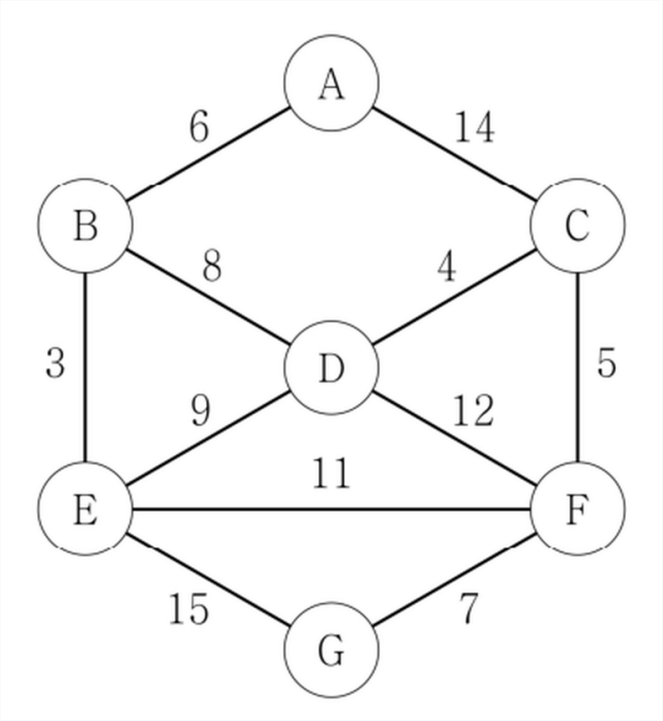
트리 노드 수 문제는 먼저 높이의 기준이 루트 레벨 1인지 확인한 뒤, 완전이면 최소와 최대를 나누어 보고 포화이면 바로 $2^h - 1$을 적용하면 된다.

암기 포인트

루트 레벨을 1로 둘 때 완전 이진 트리의 최소 노드 수는 2^{h-1}이고, 완전 이진 트리의 최대 노드 수와 포화 이진 트리의 노드 수는 모두 $2^h - 1$이다.

NoteBase

문 16. 다음 그래프에 대하여 Sollin 알고리즘으로 최소 비용 신장 트리를 구하고자 한다. 각 노드가 하나의 트리로 구성된 초기 상태에서 한 단계 처리한 이후의 결과로 두 개의 트리로 구성된 포리스트(forest)가 만들어졌을 때, 각 트리의 가중치 합 중 큰 값은?



① 9
② 12
③ 13
④ 16

정답 4번

출제 포인트

이 문제는 초기 상태에서 Sollin 알고리즘을 한 단계 적용할 때 각 노드가 선택하는 최소 간선을 묶어 포리스트를 만들고, 각 트리의 가중치 합을 계산하는 문제이다.

조건 정리

초기 상태에서는 각 노드가 각각 하나의 트리이다.
한 단계 처리에서는 각 트리가 자신에게 incident한 간선 중 가중치가 가장 작은 간선을 선택한다.
구해야 할 값은 한 단계 후 만들어진 두 트리의 가중치 합 중 큰 값이다.

NoteBase

계산 과정

1. 각 노드의 최소 간선은 A-B(6), B-E(3), C-D(4), D-C(4), E-B(3), F-C(5), G-F(7)이다.
2. 중복을 제거하면 실제로 선택되는 간선은 A-B(6), B-E(3), C-D(4), C-F(5), F-G(7)이다.
3. 따라서 포리스트는 {A, B, E}와 {C, D, F, G}의 두 트리로 합쳐진다.
4. 각 트리의 가중치 합은 {A, B, E}가 6+3=9, {C, D, F, G}가 4+5+7=16이다.
5. 큰 값은 16이다.

선지별 해설

① X: 9는 A-B(6), B-E(3)로 이루어진 {A, B, E} 트리의 가중치 합이다. 틀린 표현은 '큰 값이 9'라는 부분이고, 올바르게는 작은 값이 9이고 큰 값은 16이다.
② X: 12는 F가 최소 간선 C-F(5)를 두지 않고 D-F(12)를 잘못 보거나, G-F(7)를 빠뜨린 계산에서 나오는 값이다. 틀린 표현은 '큰 값이 12'라는 부분이고, 올바르게는 C-D, C-F, F-G가 선택되어 큰 값은 16이다.
③ X: 13은 한 단계 처리 결과에서 나올 수 없는 값이다. 선택 간선 집합이 A-B, B-E, C-D, C-F, F-G로 정해지므로 트리 가중치 합은 9와 16뿐이다. 틀린 표현은 '큰 값이 13'이라는 부분이고, 올바르게는 큰 값이 16이다.
④ O: C-D(4), C-F(5), F-G(7)가 묶여 {C, D, F, G} 트리가 되고 가중치 합은 16이다. 두 트리의 합 9와 16 중 큰 값이 16이므로 옳다.

오답 포인트

Sollin 알고리즘의 한 단계는 초기에는 각 노드가 자기 기준 최소 간선을 하나씩 고른다는 점을 놓치기 쉽다. D-F(12)처럼 더 큰 간선을 임의로 넣거나, G-F(7)를 빠뜨리면 12나 13 같은 오답이 나온다.

다시 나오면 이렇게 판단

초기 Boruvka형 문제는 각 노드의 최소 간선을 먼저 표시하고, 중복 간선을 합친 뒤 연결 성분별로 가중치 합을 계산하면 된다. 최종 답은 그 성분들의 합 중 발문이 요구한 값으로 정해진다.

암기 포인트

Sollin 1단계는 각 정점의 최소 간선 선택, 중복 제거, 연결 성분 합산의 순서로 정리된다.

NoteBase

문 17. 비어있는 레드 블랙 트리(red black tree)에 차례대로 키(key) 값 1, 3, 2, 4, 5, 6을 넣고 완성한 레드 블랙 트리를 전위 순회한 결과는? (단, 키를 저장하고 있는 노드만 순회하고, 순회 결과에서 키 값 뒤에 B를 붙이면 블랙 노드, R을 붙이면 레드 노드임을 나타낸다)

① 2B, 1B, 4R, 3B, 5B, 6R

② 2B, 1R, 4R, 3B, 5B, 6R

③ 4B, 2R, 1B, 3B, 5B, 6R

④ 4B, 2R, 1B, 3R, 5B, 6R

정답 1번

출제 포인트

레드블랙트리에 키를 순서대로 삽입할 때 회전과 재색칠을 반영한 최종 트리를 만든 뒤, 그 트리의 전위 순회 결과를 판단하는 문제이다.

핵심 정리

1, 3, 2를 넣으면 RL 형태가 되어 2가 루트가 되고 1R, 3R이 자식이 된다. 4를 넣으면 부모 3R와 삼촌 1R이 모두 red이므로 1과 3은 black, 2는 red가 되었다가 루트 규칙으로 다시 black이 된다. 5를 넣으면 3-4-5가 RR 형태이므로 4가 올라가고 3은 red가 된다. 6을 넣으면 부모 5R와 삼촌 3R이 모두 red이므로 3과 5는 black, 4는 red가 된다. 최종 트리는 루트 2B, 왼쪽 1B, 오른쪽 4R, 그 아래 3B, 5B, 6R 구조이므로 전위 순회는 2B, 1B, 4R, 3B, 5B, 6R이다.

선지별 해설

① O: 최종 트리가 2B를 루트로 하고 왼쪽에 1B, 오른쪽에 4R, 그 아래에 3B, 5B, 6R이 놓인다. 전위 순회는 루트-왼쪽-오른쪽 순서이므로 2B, 1B, 4R, 3B, 5B, 6R이 맞다.
② X: "1R"이 틀린 표현이다. 4를 삽입할 때 부모 3과 삼촌 1이 모두 red여서 재색칠이 일어나므로 1은 black이 된다. 올바르게는 2B, 1B, 4R, 3B, 5B, 6R이다.
③ X: "4B, 2R"로 시작하는 전위 순회가 틀린 표현이다. 모든 삽입과 보정을 끝내도 루트는 2B로 유지되며 4는 오른쪽 자식인 red 노드이다. 올바르게는 2B, 1B, 4R, 3B, 5B, 6R이다.
④ X: "2R"과 "3R"이 틀린 표현이다. 루트는 항상 black이어야 하므로 2는 black이고, 6 삽입 뒤 재색칠로 3도 black이 된다. 올바르게는 2B, 1B, 4R, 3B, 5B, 6R이다.

오답 포인트

새로 삽입한 노드가 red라는 사실만 기억하고 이후 재색칠 결과를 놓치면 ②처럼 1을 계속 red로 보거나 ④처럼 3을 red로 남겨 두기 쉽다. 또 회전이 일어난 뒤에도 루트 색은 black이라는 규칙을 놓치면 ③, ④처럼 루트 색이나 루트 자체를 잘못 잡게 된다.

NoteBase

다시 나오면 이렇게 판단

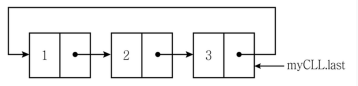
삽입할 때마다 먼저 BST 위치를 잡고, 부모가 red이면 삼촌의 색을 확인한다. 삼촌도 red이면 재색칠, 삼촌이 black이면 회전 후 재색칠을 적용한다. 모든 삽입이 끝난 뒤 최종 트리를 한 번 확정한 다음 전위 순회는 루트-왼쪽-오른쪽 순서로 읽으면 된다.

암기 포인트

레드블랙트리는 "부모와 삼촌이 모두 red이면 재색칠, 삼촌이 black이면 회전 후 재색칠"로 정리되고 루트는 항상 black이다.

NoteBase

문 18. 그림과 같은 원형 연결 리스트 myCLL을 만들기 위해 다음 파이썬 코드를 작성하였다. (가) ~ (다)에 들어갈 코드를 A ~ D에서 바르게 연결한 것은?




```
class Node:
    def __init__(self, data, next):
        self.data = data
        self.next = next

class CircularLinkedList:
    def __init__(self):
        self.last = None

    def insert(self, data):
        n = Node(data, None)
        if self.last is None:
            n.next = n
            self.last = n
        else:
            (가)
            (나)
            (다)

myCLL = CircularLinkedList()
myCLL.insert(1)
myCLL.insert(2)
myCLL.insert(3)
```

```
A. n.next = self.last
B. n.next = self.last.next
C. self.last = n
D. self.last.next = n
```

① (가): A, (나): C, (다): D

② (가): A, (나): D, (다): C

③ (가): B, (나): C, (다): D

④ (가): B, (나): D, (다): C

정답 4번

출제 포인트

이 문제는 원형 연결 리스트에서 self.last는 마지막 노드, self.last.next는 첫 노드를 가리킨다는 구조를 이용해 끝 삽입 코드의 올바른 연결 순서를 판단하는 문제이다.

NoteBase

선지별 해설

1번 X: (가)에 A를 넣으면 새 노드의 next가 마지막 노드를 가리켜 첫 노드로 연결되지 않는다. 틀린 표현은 "(가)=A"이며 올바르게는 "(가)=B"이다. 또한 (나)=C 뒤에 (다)=D를 두면 self.last가 이미 n으로 바뀐 뒤라 n.next=n이 되어 기존 원형 연결이 깨진다.
2번 X: (나)=D, (다)=C의 순서는 맞지만 (가)=A가 틀리다. 새 노드는 마지막 노드가 아니라 첫 노드인 self.last.next를 가리켜야 하므로 올바르게는 "(가)=B"이다.
3번 X: (가)=B는 맞지만 (나)=C를 먼저 수행하면 self.last가 새 노드로 바뀐 상태에서 (다)=D가 실행되어 self.last.next=n, 즉 n.next=n이 된다. 틀린 표현은 "(나)=C, (다)=D"의 순서이며 올바르게는 "(나)=D, (다)=C"이다.
4번 O: (가)=B로 새 노드가 첫 노드를 가리키고, (나)=D로 기존 마지막 노드가 새 노드를 가리키며, (다)=C로 마지막 포인터를 새 노드로 갱신하므로 그림과 같은 원형 연결 구조가 유지된다.

오답 포인트

수험생은 self.last가 마지막 노드라는 점은 기억하면서도 끝 삽입에서 먼저 이어야 할 대상이 첫 노드 self.last.next라는 점을 놓치기 쉽다. 또 self.last를 너무 일찍 n으로 바꾸면 이후 self.last.next 연산의 기준 노드가 바뀌어 자기 자신을 가리키는 잘못된 연결이 생긴다.

다시 나오면 이렇게 판단

원형 연결 리스트의 끝 삽입은 새 노드가 먼저 첫 노드를 가리키고, 기존 마지막 노드가 새 노드를 가리킨 뒤, 마지막 포인터를 새 노드로 옮기는 순서인지 확인하면 된다.

암기 포인트

끝 삽입 순서는 "새 노드→첫 노드, 기존 last→새 노드, last 갱신"이다.

NoteBase

문 19. 다음 C 코드는 단순 연결 리스트를 이용하여 스택의 pop() 함수를 구현한 것이다. (가), (나)에 들어갈 내용을 바르게 연결한 것은?

```
struct stack {
    int data;
    struct stack* link;
};
struct stack* top = NULL;
int pop(){
    struct stack* temp = top;
    int item;
    if (!temp){
        printf("Stack is empty.\n");
        exit(1);
    }
    item = temp->data;
    [ (가) ] ;
    free(temp);
    [ (나) ] ;
}
```

① (가) temp = temp->link, (나) return item

② (가) temp = temp->link, (나) return top->data

③ (가) top = temp->link, (나) return item

④ (가) top = temp->link, (나) return top->data

정답 3번

출제 포인트

이 문제는 단순 연결 리스트로 구현한 스택에서 pop 연산이 삭제 대상 노드를 안전하게 제거하고 기존 top의 값을 반환하는 순서를 정확히 적용할 수 있는지를 묻는 문제이다.

사례 정리

- top은 스택의 맨 위 노드를 가리킨다.
- pop은 현재 top 노드의 데이터를 꺼낸 뒤 그 노드를 삭제해야 한다.
- 삭제 전에는 다음 노드를 새 top으로 연결해야 하고, 반환값은 삭제된 노드의 data이어야 한다.

NoteBase

핵심 개념

- 연결 리스트 기반 스택의 pop = 현재 top 노드의 값을 보관하고 top을 다음 노드로 이동시킨 뒤 기존 노드를 해제하고 보관한 값을 반환하는 연산이다.
- 적용 요건: 삭제 대상 노드의 주소를 별도로 보관하고, 전역 포인터 top을 먼저 다음 노드로 갱신한 다음, 해제 후에는 삭제된 노드를 다시 참조하지 않아야 한다.

적용 기준

이 코드에서는 현재 top을 temp에 저장했으므로 (가)에는 전역 포인터 top을 다음 노드로 넘기는 문장이 들어가야 한다. 또한 삭제할 노드를 free한 뒤에는 이미 item에 저장해 둔 값을 반환해야 하므로 (나)에는 return item;이 들어가야 한다.

선지별 해설

① X: temp = temp->link는 지역 변수 temp만 다음 노드로 이동시킬 뿐 top은 그대로 남는다. 이어서 free(temp)를 하면 원래 삭제해야 할 노드가 아니라 다음 노드를 해제하게 되거나, top이 삭제된 노드를 계속 가리키는 문제가 생긴다. 틀린 표현은 (가)의 temp = temp->link이고, 올바르게는 top = temp->link이다. (나)의 return item은 삭제 직전 저장한 값을 반환하므로 맞다.
② X: (가)의 temp = temp->link가 잘못된 이유는 ①과 같다. 또한 return top->data는 pop으로 꺼낸 값이 아니라 현재 top이 가리키는 값에 의존하므로 반환 대상이 잘못된다. 틀린 표현은 (가)의 temp = temp->link와 (나)의 return top->data이고, 올바르게는 top = temp->link와 return item이다.
③ O: top = temp->link로 스택의 top을 다음 노드로 갱신한 뒤, free(temp)로 기존 맨 위 노드를 해제하고, return item으로 삭제된 노드의 원래 데이터를 반환하므로 pop의 동작 순서에 맞다.
④ X: (가)의 top = temp->link는 맞지만, return top->data는 삭제된 노드의 값이 아니라 새 top 노드의 값을 반환하게 된다. 원소가 하나뿐이었다면 top이 NULL이 되어 top->data 자체가 잘못된 참조가 된다. 틀린 표현은 (나)의 return top->data이고, 올바르게는 return item이다.

오답 포인트

temp와 top의 역할을 혼동하면 ①, ②를 고르기 쉽다. temp는 삭제 대상 노드를 잠시 보관하는 지역 포인터일 뿐이고, 실제 스택의 맨 위를 바꾸는 포인터는 top이다. 또한 pop의 반환값은 삭제 후의 top 값이 아니라 삭제되기 전 맨 위 원소라는 점을 놓치면 ④에 흔들린다.

다시 나오면 이렇게 판단

연결 리스트 삭제 코드가 나오면 먼저 어떤 포인터가 구조 전체의 시작점을 관리하는지 확인한다. 그다음 삭제 대상 노드의 값을 저장하고, 시작 포인터를 다음 노드로 갱신하고, 기존 노드를 해제한 뒤, 저장한 값을 반환하는 순서인지 보면 된다.

NoteBase

암기 포인트

연결 리스트 스택의 pop은 값 저장 → top 이동 → 기존 top 해제 → 저장값 반환의 순서이다.

NoteBase

문 20. 다음 조건에 따라 공백 해시 테이블에 키(key) 값 12, 15, 25, 34, 37, 75, 62, 33, 47, 55, 21, 42, 53을 순서대로 삽입할 때, 충돌 횟수가 가장 많은 키 값은? (단, 해시 함수 또는 선형 조사법을 적용하여 찾아간 버킷에 빈 슬롯이 없을 때마다 충돌이 발생한 것으로 본다)

◯ 해시 함수: h(x) = x mod 13
◯ 충돌 해결책: 선형 조사법(linear probing)
◯ 해시 테이블의 크기: 13개 버킷(0부터 12까지 인덱스를 가짐)
◯ 해시 테이블 버킷당 슬롯 수: 1개

① 21
② 42
③ 53
④ 62

정답 1번

출제 포인트

공백 해시 테이블에서 h(x)=x 13과 선형 조사법을 적용하여 각 키가 삽입될 때 만나는 연속 점유 버킷 수, 즉 충돌 횟수를 세는 문제이다.

조건 정리

주어진 키 삽입 순서는 12, 15, 25, 34, 37, 75, 62, 33, 47, 55, 21, 42, 53이다.
해시 함수는 h(x)=x 13이다.
버킷은 0부터 12까지 13개이고 버킷당 슬롯은 1개이다.
선형 조사법이므로 빈 버킷이 나올 때까지 다음 인덱스로 1칸씩 이동하며, 이미 찬 버킷을 한 번 만날 때마다 충돌 1회로 센다.

계산 과정

1. 초반 삽입 결과 12는 12번, 15는 2번, 25는 12번에서 충돌 1회 후 0번, 34는 8번, 37은 11번, 75는 10번에 들어간다.
2. 62는 62 13=10이므로 10→11→12→0이 모두 차 있어 충돌 4회 후 1번에 들어간다.
3. 33은 7번, 47은 47 13=8에서 8번이 차 있어 충돌 1회 후 9번, 55는 3번에 들어간다.
4. 21은 21 13=8이므로 8→9→10→11→12→0→1→2→3이 모두 차 있어 충돌 9회 후 4번에 들어간다.
5. 42는 42 13=3에서 3→4가 차 있어 충돌 2회 후 5번, 53은 53 13=1에서 1→2→3→4→5가 차 있어 충돌 5회 후 6번에 들어간다.

NoteBase

선지별 해설

① O: 21은 8번에서 시작해 4번에 들어갈 때까지 찬 버킷 9개를 연속해서 만나므로 보기들 중 충돌 횟수가 가장 많다.
② X: 42의 충돌 횟수는 2회이다. 3번과 4번만 차 있고 5번이 비어 있으므로 최대 충돌 키가 아니다. 틀린 표현은 '가장 많다'이고, 올바르게는 '42는 2회 충돌한다'이다.
③ X: 53의 충돌 횟수는 5회이다. 1번에서 시작해 1, 2, 3, 4, 5번이 차 있어 6번에 들어간다. 틀린 표현은 '가장 많다'이고, 올바르게는 '53은 5회 충돌한다'이다.
④ X: 62의 충돌 횟수는 4회이다. 10, 11, 12, 0번이 차 있어 1번에 들어간다. 틀린 표현은 '가장 많다'이고, 올바르게는 '62는 4회 충돌한다'이다.

오답 포인트

21은 시작 주소가 8번이라는 점만 보고 47과 비슷하게 처리하면 안 된다. 21은 12번 뒤에서 끝나지 않고 0번으로 순환하여 3번까지 계속 조사하므로 충돌 수가 9회까지 늘어난다. 또 선형 조사에서 처음 한 번만 충돌로 세는 것이 아니라, 이동 중 만나는 모든 점유 버킷을 충돌로 세어야 한다.

다시 나오면 이렇게 판단

선형 조사 문제는 각 후보 키의 해시 주소를 먼저 구한 뒤, 빈 버킷을 만날 때까지 실제 이동 경로를 인덱스 순서대로 적어 가며 점유 버킷 수를 센다. 인덱스가 12에서 끝나면 0으로 되돌아가는 순환도 반드시 포함한다.

암기 포인트

선형 조사 충돌 수는 처음 해시 주소부터 빈 칸 직전까지 만난 점유 버킷 수이다.

NoteBase

문 21. 다음 C 코드의 reverseList()는 단순 연결 리스트에 있는 노드의 순서를 역순으로 만드는 함수이다. (가) ~ (다)에 들어갈 내용을 바르게 연결한 것은?

```
struct node {
    int data;
    struct node *link;
} *head;

void reverseList() {
    struct node *prevNode, *curNode;
    prevNode = head;
    curNode = head;
    head = NULL;

    while ( (가) ) {
        curNode = curNode->link;
        prevNode->link = head;
        (나) ;
        (다) ;
    }
}
```

① (가): curNode != NULL, (나): head = curNode, (다): curNode = prevNode

② (가): curNode != NULL, (나): head = prevNode, (다): prevNode = curNode

③ (가): head != NULL, (나): head = curNode, (다): curNode = prevNode

④ (가): head != NULL, (나): head = prevNode, (다): prevNode = curNode

정답 2번

출제 포인트

이 문제는 단순 연결 리스트를 역순으로 만들 때 현재 노드의 다음 노드를 먼저 보관한 뒤, 현재 노드를 역순 리스트의 머리에 붙이고, 마지막에 작업 대상을 다음 노드로 넘기는 포인터 갱신 순서를 묻는다.

NoteBase

올바른 순서

1. 반복 조건 확인: 아직 처리할 노드가 있으므로 curNode != NULL인 동안 반복한다.
2. 다음 노드 보관: curNode = curNode->link로 현재 노드의 다음 노드를 미리 저장한다.
3. 링크 반전: prevNode->link = head로 현재 노드가 이미 뒤집힌 리스트의 맨 앞을 가리키게 만든다.
4. 새 머리 갱신: head = prevNode로 방금 처리한 노드를 역순 리스트의 새 머리로 만든다.
5. 작업 대상 이동: prevNode = curNode로 다음 반복에서 처리할 노드로 이동한다.

왜 이 순서인가: 다음 노드 주소는 링크를 끊기 전에 먼저 보관해야 하고, head는 현재 노드를 새 머리로 붙인 뒤에만 갱신될 수 있다.

선지별 해설

1번: X. (나)에서 head = curNode로 두면 방금 뒤집어 붙인 노드가 아니라 아직 처리하지 않은 다음 노드를 머리로 삼게 되어 역순 리스트 구성이 깨진다. 틀린 표현은 "head = curNode"이다. 올바르게는 head = prevNode이다. (다)의 curNode = prevNode도 다음 노드로 전진하지 못하므로 잘못이다. 올바르게는 prevNode = curNode이다.

2번: O. 반복 조건을 curNode != NULL로 두면 처리할 노드가 남아 있는 동안 반복한다. 본문에서 다음 노드를 먼저 저장한 뒤 prevNode->link = head, head = prevNode, prevNode = curNode 순으로 갱신하면 각 노드가 역순 리스트의 앞에 하나씩 붙는다.

3번: X. head는 반복 시작 전에 NULL로 초기화되므로 while(head != NULL)는 첫 검사에서 거짓이 되어 반복문이 한 번도 실행되지 않는다. 틀린 표현은 "head != NULL"이다. 올바르게는 curNode != NULL이다. (나)의 head = curNode도 새 머리 갱신 대상으로 맞지 않는다. 올바르게는 head = prevNode이다.

4번: X. (가)가 head != NULL이면 head를 NULL로 만든 직후 반복이 시작되지 못한다. 틀린 표현은 "head != NULL"이다. 올바르게는 curNode != NULL이다. (나), (다)는 2번과 같지만 반복 조건이 틀리므로 전체 연결은 성립하지 않는다.

오답 포인트

head를 순회 포인터로 착각하면 3번과 4번처럼 반복 조건을 head != NULL로 고르기 쉽다. 이 코드에서 head는 원래 리스트의 머리가 아니라 역순으로 재구성되는 새 리스트의 머리이므로 처음에 NULL이 되는 점이 함정이다. 또 다음 노드를 미리 저장한 뒤 head = prevNode로 붙여야 하는데, head = curNode로 착각하면 아직 처리하지 않은 노드를 새 머리로 잘못 올리게 된다.

다시 나오면 이렇게 판단

연결 리스트 반전 코드는 "다음 노드 보관 -> 현재 노드의 링크를 뒤집힌 리스트로 연결 -> head를 현재 노드로 갱신 -> 처리 포인터를 다음 노드로 이동"의 흐름으로 판단한다. 반복 조건은 새 머리인 head가 아니라 아직 처리할 노드를 가리키는 순회 포인터가 NULL인지로 결정된다.

NoteBase

암기 포인트

리스트 반전은 next 저장, link 반전, head 갱신, 포인터 전진의 순서이다. head는 결과 리스트의 머리이고 curNode는 미처리 노드를 가리킨다.

NoteBase

문 22. 다음 배열에 대하여, 위치 값이 증가함에 따라 키(key) 값도 이에 정비례하여 증가한다는 가정으로 탐색 위치를 계산하는 보간 탐색을 수행할 때, 키 45를 찾을 때까지 탐색 위치를 계산한 횟수는? (단, 탐색 위치 계산 시 소수점 이하는 버린다)

위치	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
키	2	3	4	7	13	18	22	27	32	42	45	70	72

① 2

② 3

③ 4

④ 5

정답 2번

출제 포인트

이 문제는 보간 탐색의 탐색 위치 계산식을 반복 적용해 목표 키를 찾을 때까지 위치를 몇 번 계산하는지를 묻는다.

조건 정리

주어진 값: 정렬된 키 배열의 위치는 0부터 12까지이고, 찾을 키는 45이다.
구해야 할 값: 키 45를 찾을 때까지 보간 탐색이 탐색 위치를 계산한 횟수이다.
주의할 조건: 탐색 위치 계산 결과의 소수점 이하는 버린다.

사용할 공식

보간 탐색의 위치는 $pos = low + \left\lfloor \frac{(key - A[low])(high - low)}{A[high] - A[low]} \right\rfloor$로 계산한다.
이 공식을 쓰는 이유: 키 값이 위치에 비례해 증가한다고 가정하므로, 양 끝 값 사이에서 목표 키 45가 들어갈 상대 위치를 비율로 추정하기 때문이다.

NoteBase

계산 과정

1. 처음 범위는 low=0, high=12이고 $A[0]=2$, $A[12]=72$이다. 따라서
$pos = 0 + \left\lfloor \frac{(45-2)(12-0)}{72-2} \right\rfloor = \lfloor 7.371... \rfloor = 7$이다.
2. 위치 7의 값은 27이므로 45보다 작다. 다음 범위는 low=8, high=12이고 $A[8]=32$, $A[12]=72$이다.
따라서 $pos = 8 + \left\lfloor \frac{(45-32)(12-8)}{72-32} \right\rfloor = 8 + \lfloor 1.3 \rfloor = 9$이다.
3. 위치 9의 값은 42이므로 45보다 작다. 다음 범위는 low=10, high=12이고 $A[10]=45$, $A[12]=72$이다.
따라서 $pos = 10 + \left\lfloor \frac{(45-45)(12-10)}{72-45} \right\rfloor = 10$이다.
4. 세 번째 위치 계산에서 바로 $A[10]=45$를 찾으므로 탐색 위치를 계산한 횟수는 총 3회이다.

선지 판단

1번 X: 계산 결과는 3회이므로 2회라는 값이 틀리다. 틀린 표현은 "2"이고 올바르게는 "3"이다. 두 번째 계산 위치 9에서 값 42를 확인한 뒤 아직 한 번 더 위치를 계산해야 하므로 2회로 끝나지 않는다.
2번 O: 세 번째 위치 계산에서 위치 10이 나오고 $A[10]=45$이므로 총 3회 계산한다. 계산 결과와 일치하므로 옳다.
3번 X: 계산 횟수는 4회가 아니라 3회이다. 틀린 표현은 "4"이고 올바르게는 "3"이다. 키를 찾은 뒤에 추가 계산이 한 번 더 필요하다고 잘못 세면 4회로 오인한다.
4번 X: 계산 횟수는 5회가 아니라 3회이다. 틀린 표현은 "5"이고 올바르게는 "3"이다. 보간 탐색의 범위 갱신 과정을 단계 수로 막연히 늘려 세면 큰 값의 오답이 나온다.

오답 포인트

탐색 위치를 계산한 횟수와 비교한 횟수, 또는 범위를 갱신한 단계를 서로 섞어 세면 오답이 된다. 또 소수점 이하를 버리는 조건을 놓치면 첫 위치부터 달라질 수 있다.

다시 나오면 이렇게 판단

보간 탐색 문제는 현재 low, high, 양 끝 키 값을 먼저 적고 공식으로 위치를 계산한 뒤, 그 위치의 값과 목표 키를 비교해 low 또는 high를 갱신한다. 이 과정을 목표 키를 찾을 때까지 반복하면서 위치를 계산할 때마다 1회씩 누적하면 된다.

암기 포인트

보간 탐색은 값의 비율로 위치를 추정하고, 소수점은 버리며, 위치를 계산할 때마다 횟수를 1회로 센다.

NoteBase

문 23. 다음 C 코드는 셸 정렬을 구현한 것이다. 코드의 출력 결과는?

```c
#include <stdio.h>
void sort(int arr[], int n) {
   int count = 1;
   int i, j, k, h, val;
   for (h = n/2; h > 0; h /= 2) {
      for (i = 0; i < h; i++)  {
         for (j = i+h; j < n; j += h) {
            val = arr[j];
            k = j;
            while (k > h-1 && arr[k-h] > val) {
               arr[k] = arr[k-h];
               k -= h;
            }
            arr[k] = val;

            if (count == 5)
               for (int t = 0; t < 6; t++)
                   printf("%d ", arr[t]);
            count++;
         }
      }
   }
}

int main() {
    int a[6] = {5, 9, 2, 3, 6, 7};
    sort(a, 6);
}
```

① 2 3 5 6 9 7

② 2 3 6 5 9 7

③ 3 6 2 5 9 7

④ 3 9 2 5 6 7

정답 2번

출제 포인트

이 문제는 셸 정렬의 gap별 삽입 과정을 추적하여 count == 5일 때 배열 상태를 찾는 문제이다.

올바른 순서

1. 초기 gap은 3이므로 부분 배열별 삽입 정렬을 수행한다. 배열은 [5, 9, 2, 3, 6, 7]에서 [3, 6, 2, 5, 9, 7]이 된다.
2. gap이 1로 줄어들면 일반 삽입 정렬이 시작된다. 첫 번째 삽입(j = 1) 뒤 배열은 [3, 6, 2, 5, 9, 7]로 그대로이다.
3. count는 내부 j 반복이 끝날 때마다 증가하므로 count == 5는 두 번째 삽입(j = 2) 직전 조건이다.
4. 값 2를 앞쪽에 삽입하면 배열이 [2, 3, 6, 5, 9, 7]이 되고 이 상태가 출력된다.

왜 이 순서: 이 코드는 정렬 완료 후가 아니라 각 삽입 직후에 count를 검사하므로 다섯 번째 출력 시점은 gap 1 단계의 두 번째 삽입 직후이다.

NoteBase

선지별 해설

1번 X: 2 3 5 6 9 7은 다섯 번째 삽입보다 더 진행된 상태이다. 틀린 표현은 셋째와 넷째 원소까지 이미 정렬되었다고 본 부분이며, 올바르게는 출력 결과가 2 3 6 5 9 7이다.
2번 O: gap 3 정렬 뒤 배열은 3 6 2 5 9 7이 되고, gap 1에서 j = 1은 변화가 없으며 j = 2에서 값 2가 맨 앞으로 삽입되어 2 3 6 5 9 7이 된다.
3번 X: 3 6 2 5 9 7은 gap 3 정렬까지 끝난 상태로, count == 5일 때보다 이른 중간 결과이다. 틀린 표현은 두 번째 삽입 결과가 아직 반영되지 않았다고 본 부분이며, 올바르게는 출력 결과가 2 3 6 5 9 7이다.
4번 X: 3 9 2 5 6 7은 첫 번째 gap 3 삽입만 반영한 상태이다. 틀린 표현은 9가 아직 두 번째 원소에 남아 있다고 본 부분이며, 올바르게는 출력 결과가 2 3 6 5 9 7이다.

오답 포인트

셸 정렬 문제에서 최종 정렬 결과를 바로 떠올리면 틀린다. 이 코드는 count == 5에서 중간 상태를 출력하므로 gap 3 정렬 3회와 gap 1 정렬의 첫 2회 삽입만 추적해야 한다.

다시 나오면 이렇게 판단

코드 추적형 정렬 문제는 반복문의 중첩 순서, 출력 조건 검사 위치, 카운터 증가 위치를 먼저 표시하면 된다. 그다음 출력이 걸리는 정확한 삽입 횟수까지만 배열 상태를 갱신하면 정답이 결정된다.

암기 포인트

셸 정렬 출력 추적은 gap 변화와 삽입 횟수를 함께 센다. count가 가리키는 것은 정렬 완료 시점이 아니라 각 삽입 직후 상태이다.

NoteBase

문 24. 루트 노드의 인덱스를 1로 하는 1차원 배열을 이용하여 최대 힙(max heap)을 구현한 후, 다음 데이터를 차례대로 하나씩 힙에 삽입하였다. 이후 힙 삭제 연산을 1회 수행한 후 배열의 인덱스 6에 저장된 데이터는?

15, 8, 10, 18, 22, 13, 26, 7

① 8
② 10
③ 13
④ 15

정답 2번

출제 포인트

이 문제는 최대 힙에 데이터를 순차 삽입한 뒤 삭제 연산 1회를 수행했을 때 배열의 특정 인덱스 값이 어떻게 바뀌는지를 묻는 문제이다.

조건 정리

- 루트 노드의 인덱스는 1이다.
- 삽입 순서는 15, 8, 10, 18, 22, 13, 26, 7이다.
- 최대 힙의 삭제 연산은 루트 삭제 후 마지막 원소를 루트로 올리고, 더 큰 자식과 교환하며 아래로 내려보내는 방식이다.
- 구해야 할 값은 삭제 1회 후 배열의 인덱스 6에 저장된 데이터이다.

계산 과정

1. 순차 삽입이 끝난 최대 힙 배열은 [26, 18, 22, 8, 15, 10, 13, 7]이다.
2. 루트 26을 삭제하고 마지막 원소 7을 루트로 올리면 [7, 18, 22, 8, 15, 10, 13]이 된다.
3. 루트 7은 자식 18, 22 중 더 큰 22와 교환되어 [22, 18, 7, 8, 15, 10, 13]이 된다.
4. 인덱스 3의 7은 자식 10, 13 중 더 큰 13과 교환되어 [22, 18, 13, 8, 15, 10, 7]이 된다.
5. 따라서 배열의 인덱스 6에 저장된 값은 10이다.

NoteBase

선지별 해설

① X: 틀린 표현은 '인덱스 6에 8이 저장된다'이다. 올바르게는 삭제 후 8은 인덱스 4에 있고, 인덱스 6에는 10이 저장된다.
② O: 삭제 후 최종 힙 배열이 [22, 18, 13, 8, 15, 10, 7]이므로 인덱스 6의 값은 10이다.
③ X: 틀린 표현은 '인덱스 6에 13이 저장된다'이다. 올바르게는 13은 재정렬 후 인덱스 3에 있고, 인덱스 6에는 10이 저장된다.
④ X: 틀린 표현은 '인덱스 6에 15가 저장된다'이다. 올바르게는 15는 인덱스 5에 있고, 인덱스 6에는 10이 저장된다.

오답 포인트

삽입이 끝난 힙 상태만 보고 답을 고르거나, 삭제 후 마지막 원소를 루트로 올린 뒤 한 번만 교환하고 멈추는 경우에 오답이 나온다. 이 문제는 삭제 후 아래로 내려가며 힙 조건을 끝까지 회복하는 과정을 놓치기 쉽다.

다시 나오면 이렇게 판단

힙 배열 문제는 삽입 완료 상태를 먼저 적고, 삭제 연산이 나오면 마지막 원소를 루트로 올린 뒤 더 큰 자식과 계속 교환하여 최종 배열을 확정한 다음 묻는 인덱스를 확인하는 순서로 판단한다.

암기 포인트

최대 힙 삭제는 '루트 삭제 → 마지막 원소 루트 이동 → 더 큰 자식과 반복 교환'의 흐름으로 정리된다.

NoteBase

문 25. 행 우선 순서로 저장되는 C 언어 2차원 배열 char a[3][5]가 선언되고 초기화되었을 때, a[2][3]과 다른 값을 가지는 것은?

① *(a+5*2+3)

② *(&a[0][0]+5*2+3)

③ (a[0]+5*2)[3]

④ (*(a+2))[3]

정답 1번

출제 포인트

이 문제는 2차원 배열 `a`가 행 우선으로 연속 저장될 때 배열명 `a`, 첫 행 `a[0]`, 그리고 원소 주소식이 각각 어떤 타입과 의미를 가지는지를 묻는 문제이다.

핵심 개념

`a`는 식에서 `char (*)[5]`로 decay하므로 1칸 증가할 때 문자 1개가 아니라 한 행 5칸만큼 이동한다.
`a[0]`은 첫 번째 행의 시작 원소를 가리키는 `char *`로 해석되므로 문자 단위로 이동한다.
따라서 `a[2][3]`은 `*(*(a+2)+3)`과 같고, 시작 원소 기준으로는 13번째 오프셋의 문자값이다.

선지별 해설

1번 X: `a+5*2+3`은 `a+13`과 같고, 여기서 `a`는 행 포인터이므로 13칸 이동은 문자 13칸이 아니라 행 13개 이동이다. `*(a+13)`은 `a[13]`에 해당하는 한 행 전체를 가리키는 식이지 `a[2][3]`의 문자값이 아니다. 틀린 표현은 `a`를 문자 포인터처럼 사용한 부분이다. 올바르게는 `*(&a[0][0]+5*2+3)` 또는 `(*(a+2))[3]`처럼 문자 단위 기준이나 행+열 기준으로 써야 한다.
2번 O: `&a[0][0]`은 첫 원소의 주소이므로 여기에 `5*2+3=13`을 더하면 `a[2][3]`의 주소가 된다. 이를 다시 `*`로 역참조하면 `a[2][3]`과 같은 문자값이 된다.
3번 O: `a[0]`은 첫 행의 첫 원소를 가리키는 포인터로 보고, 여기에 10을 더하면 세 번째 행의 시작 위치와 같은 연속 저장 위치에 도달한다. 그 상태에서 `[3]`을 취하면 결국 13번째 오프셋 원소인 `a[2][3]`과 같은 문자값이 된다.
4번 O: `a+2`는 세 번째 행을 가리키고, `*(a+2)`는 그 행 자체이다. 여기에 `[3]`을 취하면 세 번째 행의 네 번째 원소이므로 바로 `a[2][3]`과 같다.

오답 포인트

배열명 `a`와 첫 행 `a[0]`을 둘 다 "시작 주소"로만 외우면 1번에 속기 쉽다. 둘 다 시작점처럼 보이지만 `a`는 한 번 더 큰 단위인 행 포인터이고, `a[0]`은 문자 포인터라는 점이 핵심 차이이다.

NoteBase

다시 나오면 이렇게 판단

2차원 배열 식이 나오면 먼저 기준 포인터의 타입을 본다. 배열명 `a`이면 행 단위 이동, `&a[0][0]`이나 `a[0]`이면 문자 단위 이동으로 해석한 뒤 최종적으로 문자값을 읽는 식인지, 행 전체를 가리키는 식인지를 확인하면 된다.

암기 포인트

`a`는 행 단위, `a[0]`은 문자 단위, `a[2][3]`은 13번째 문자값이라는 구도로 정리된다.

NoteBase