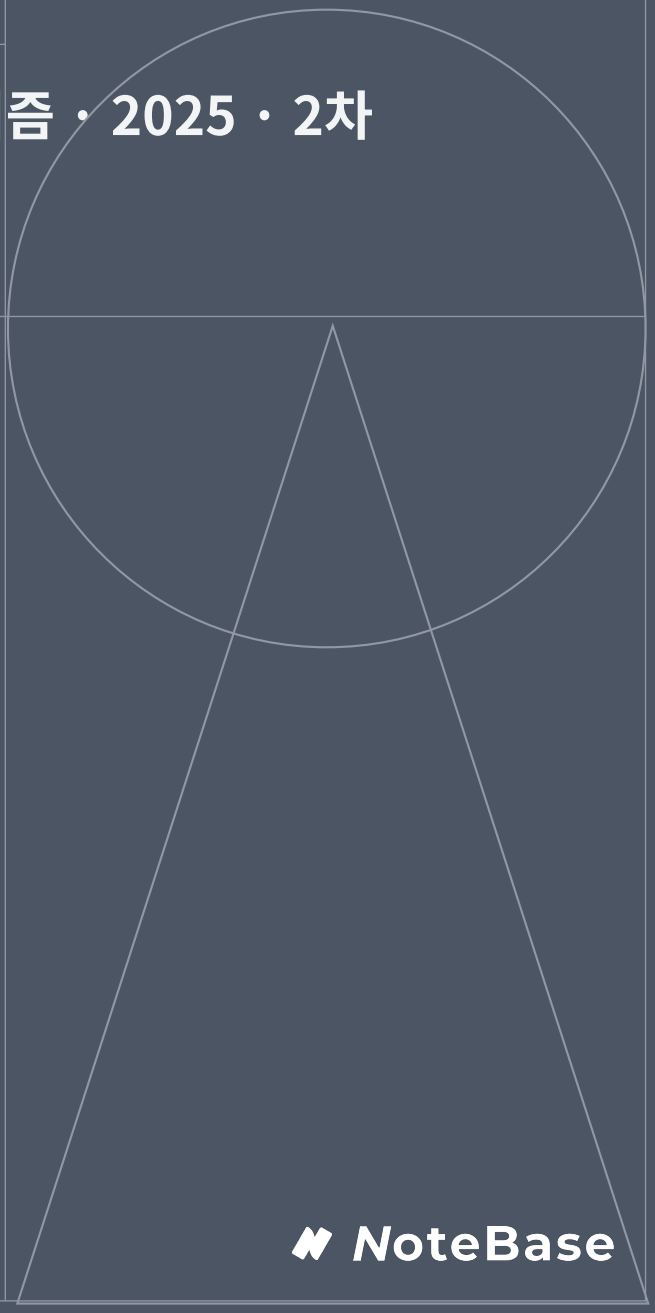
기출OX 해설집

공무원/국가직/7급 · 알고리즘 · 2025 · 2차



NoteBase

기출OX 해설집

공무원/국가직/7급 · 알고리즘 · 2025 · 2차

공무원/국가직/7급 · 알고리즘 · 2025 · 2차

문 1. 알고리즘의 조건에 대한 설명으로 옳지 않은 것은?

① 입력: 외부에서 제공되는 자료가 0개 이상이어야 한다.

② 출력: 적어도 1개 이상의 결과를 만들어야 한다.

③ 명확성: 명령어들은 현재 실행 가능한 연산이어야 한다.

④ 유한성: 한정된 수의 단계 후에는 반드시 종료되어야 한다.

정답 3번

출제 포인트

이 문제는 알고리즘의 기본 조건인 입력, 출력, 명확성, 유한성, 실행 가능성을 정확히 구분하는지를 묻는 문제이다.

핵심 개념

입력은 외부에서 주어지는 자료가 0개 이상 존재하는 성질이다. 출력은 처리 결과가 적어도 1개 이상 나와야 하는 성질이다. 명확성은 각 명령의 의미가 모호하지 않고 분명해야 하는 성질이다. 실행 가능성은 각 명령이 실제로 수행 가능한 기본 연산이어야 하는 성질이다. 유한성은 한정된 단계 안에서 반드시 종료되는 성질이다.

선지별 해설

① O. 알고리즘은 입력이 없을 수도 있으므로 입력 조건은 외부에서 제공되는 자료가 0개 이상이라는 설명이 맞다.
② O. 알고리즘은 문제 해결 결과를 내야 하므로 적어도 1개 이상의 출력을 만들어야 한다는 설명이 맞다.
③ X. 틀린 부분은 "명확성"이다. 제시된 내용인 "명령어들은 현재 실행 가능한 연산이어야 한다"는 명확성이 아니라 실행 가능성에 대한 설명이다. 명확성은 각 단계의 의미가 애매하지 않고 분명해야 한다로 고쳐야 한다.
④ O. 알고리즘은 무한히 반복되면 안 되므로 한정된 수의 단계 후에 반드시 종료되어야 한다는 설명이 맞다.

오답 포인트

명확성과 실행 가능성을 서로 바꾸어 외우면 ③에 쉽게 걸린다. 명확성은 명령의 뜻이 분명한지에 관한 조건이고, 실행 가능성은 그 명령을 실제 연산으로 수행할 수 있는지에 관한 조건이다.

NoteBase

다시 나오면 이렇게 판단

알고리즘 조건 문제는 먼저 입력 0개 이상, 출력 1개 이상, 유한성 종료 여부를 빠르게 확인하고, 남는 선지는 명확성인지 실행 가능성인지 기준을 나누어 판단하면 된다.

암기 포인트

명확성은 뜻이 분명함, 실행 가능성은 실제로 수행 가능함이다.

NoteBase

문 2. 점근 표기법에 대한 설명으로 옳은 것은?

① 알고리즘의 수행 시간을 대략적으로 나타내는 방법이다.

② O(Big-oh) 표기는 최소한의 알고리즘 수행 시간을 나타내기 위해 사용한다.

③ n개 정수 배열에 대한 순차 탐색 알고리즘은 $O(1)$ 시간 복잡도를 가진다.

④ 증가 함수 $10n^2 + 3n + 2$는 $O(n)$으로 표기할 수 있다.

정답 1번

출제 포인트

점근 표기법의 의미와 Big-O의 상한 개념, 순차 탐색의 시간 복잡도, 다항식의 최고차항 판별을 구별하는 문제이다.

핵심 개념

점근 표기법은 입력 크기 n이 커질 때 알고리즘의 수행 시간이나 공간 사용량이 어떤 비율로 증가하는지를 나타내는 방법이다. Big-O는 점근적 상한, Big-\Omega는 점근적 하한, Big-\Theta는 같은 차수의 점근적 경계를 뜻한다. 순차 탐색은 원하는 값이 뒤에 있거나 없으면 끝까지 확인하므로 최악 시간 복잡도가 $O(n)$이 된다. 다항식의 점근 표기에서는 최고차항만 남기므로 $10n^2 + 3n + 2$는 $O(n^2)$로 본다.

선지별 해설

① O: 점근 표기법은 알고리즘의 정확한 실행 시간을 초 단위로 재는 방법이 아니라, 입력 크기가 커질수록 수행 시간이 얼마나 빠르게 증가하는지를 대략적으로 나타내는 방법이므로 옳다.
② X: 틀린 부분은 \"최소한의\"이다. Big-O는 최소 수행 시간을 나타내는 표기가 아니라 점근적 상한을 나타내는 표기이다. 올바르게는 \"O(Big-O) 표기는 알고리즘 수행 시간의 점근적 상한을 나타내기 위해 사용한다\"이다. 최소한 또는 하한을 나타내는 표기는 Big-\Omega이다.
③ X: 틀린 부분은 \"$O(1)$\"이다. n개 정수 배열에 대한 순차 탐색은 최악의 경우 마지막 원소까지 확인하거나 찾지 못해 n번 비교하므로 시간 복잡도는 $O(n)$이다. 올바르게는 \"순차 탐색 알고리즘은 최악의 경우 $O(n)$ 시간 복잡도를 가진다\"이다.
④ X: 틀린 부분은 \"$O(n)$\"이다. 증가 함수 $10n^2 + 3n + 2$는 n^2항이 지배하므로 점근적으로 $O(n^2)$로 표기한다. 올바르게는 \"증가 함수 $10n^2 + 3n + 2$는 $O(n^2)$으로 표기할 수 있다\"이다.

오답 포인트

Big-O를 하한이나 최소 시간으로 착각하거나, 순차 탐색의 최선 경우 $O(1)$을 전체 시간 복잡도로 일반화하면 오답이 된다. 또한 다항식에서 상수항과 일차항만 보고 최고차항 n^2를 놓치면 ④를 잘못 고르게 된다.

다시 나오면 이렇게 판단

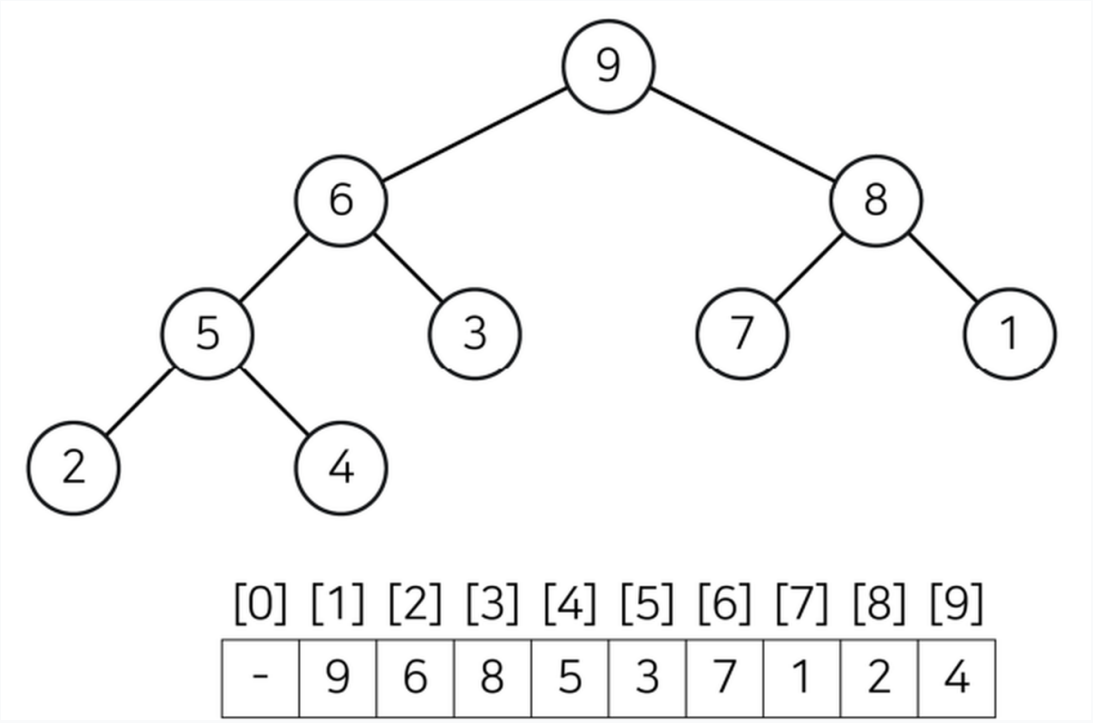
점근 표기 문제에서는 먼저 상한인지 하한인지 같은 경계의 종류를 확인하고, 탐색 알고리즘은 조건이 없으면 최악 시간 복잡도를 기준으로 판단한다. 다항식은 계수와 낮은 차항을 버리고 최고차항만 남겨 차수를 결정하면 된다.

암기 포인트

Big-O는 상한, Big-\Omega는 하한, 순차 탐색 최악은 $O(n)$, 다항식은 최고차항으로 판단한다.

NoteBase

문 3. 다음 그림과 같은 최대 힙(max heap)을 이용하여 힙 정렬을 수행할 때, 최댓값 9가 배열의 마지막 요소인 4와 교환된 다음, 힙 구조 유지를 위한 연산이 완료된 상태에서의 배열 내용으로 옳은 것은?



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

①	-	8	4	6	7	3	5	2	1	9
②	-	8	5	4	6	3	7	1	2	9
③	-	8	6	7	5	3	4	1	2	9
④	-	8	7	6	3	4	5	2	1	9

정답 3번

출제 포인트

최대 힙에서 루트와 마지막 원소를 교환한 뒤, down-heap으로 힙 속성을 복구하는 순서를 묻는 문제이다.

올바른 순서

1. 루트 9와 마지막 원소 4를 교환하면 배열은 `[-, 4, 6, 8, 5, 3, 7, 1, 2, 9]`가 된다.
2. 정렬이 끝난 9를 제외하고 힙 크기를 8로 줄인 뒤, 루트 4를 자식 6과 8 중 더 큰 8과 교환한다.
3. 그러면 배열은 `[-, 8, 6, 4, 5, 3, 7, 1, 2, 9]`가 된다.
4. 인덱스 3의 4를 자식 7과 1 중 더 큰 7과 다시 교환하면 최종 배열은 `[-, 8, 6, 7, 5, 3, 4, 1, 2, 9]`가 된다.

왜 이 순서: 최대 힙 복구는 현재 노드를 두 자식 중 더 큰 값과 바꾸는 방식으로 내려가므로, 4는 먼저 8과 바뀌고 그다음 7과 바뀐다.

NoteBase

선지별 해설

① X: 배열 `[-, 8, 4, 6, 7, 3, 5, 2, 1, 9]`는 두 번째 원소가 4가 되어 왼쪽 부분트리에서 최대 힙 조건이 깨진다.
틀린 부분: 루트에서 내려온 4가 인덱스 2에 남아 있고, 7과 5의 배치도 잘못되어 있다.
올바르게는: 첫 교환 뒤 4는 8과 먼저 바뀌고, 그다음 7과 바뀌어 `[-, 8, 6, 7, 5, 3, 4, 1, 2, 9]`가 된다.
② X: 배열 `[-, 8, 5, 4, 6, 3, 7, 1, 2, 9]`는 6이 인덱스 4로 내려가 있어 6과 5의 상대 위치가 잘못되었다.
틀린 부분: 인덱스 2에는 6이 와야 하는데 5가 올라와 있고, 인덱스 3에는 7이 와야 하는데 4가 남아 있다.
올바르게는: 루트의 자식 중 큰 값 8과 먼저 교환한 뒤, 4를 다시 7과 교환해야 한다.
③ O: 배열 `[-, 8, 6, 7, 5, 3, 4, 1, 2, 9]`는 4가 8과 7을 따라 두 단계 내려간 결과이며, 각 부모가 자식보다 크거나 같아 최대 힙 조건을 만족한다.
④ X: 배열 `[-, 8, 7, 6, 3, 4, 5, 2, 1, 9]`는 7과 6의 위치가 뒤바뀌어 원래 왼쪽 부분트리와 오른쪽 부분트리의 구조가 섞였다.
틀린 부분: 인덱스 2는 원래 루트의 왼쪽 부분트리에서 유지된 6이어야 하고, 인덱스 3은 4가 7과 교환된 자리여야 한다.
올바르게는: 최종 배열은 `[-, 8, 6, 7, 5, 3, 4, 1, 2, 9]`이다.

오답 포인트

루트에서 내려온 값 4를 단순히 한 번만 바꾸거나, 자식 둘 중 큰 값이 아니라 임의의 자식과 바꿔 버리면 오답이 된다. 이 문제의 함정은 `부모와 더 큰 자식의 교환` 원칙을 놓치게 만드는 데 있다.

다시 나오면 이렇게 판단

힙 정렬에서 루트와 마지막 원소를 교환한 뒤에는 정렬 구간을 하나 줄이고, 내려간 원소를 매 단계마다 두 자식 중 더 큰 값과 비교해 끝까지 내려보내면 된다.

암기 포인트

최대 힙 복구는 `루트 교환 후 큰 자식 쪽으로 연속 하강`으로 기억한다.

NoteBase

문 4. 다음은 초기상태 배열을 오름차순으로 정렬하는 단계들이다. 이와 같이 정렬을 수행하는 알고리즘으로 옳은 것은?

초기상태: [69, 10, 30, 3, 16, 8]
step 1: [3, 10, 30, 69, 16, 8]
step 2: [3, 8, 30, 69, 16, 10]
step 3: [3, 8, 10, 69, 16, 30]
step 4: [3, 8, 10, 16, 69, 30]
step 5: [3, 8, 10, 16, 30, 69]

① 선택 정렬(selection sort)
② 버블 정렬(bubble sort)
③ 합병 정렬(merge sort)
④ 기수 정렬(radix sort)

정답 1번

출제 포인트

이 문제는 선택 정렬의 진행 순서를 보고 다른 정렬과 구별하는 문제이다. 선택 정렬은 매 단계마다 미정렬 구간의 최솟값을 찾아 현재 위치와 교환한다.

올바른 순서

1. 전체 배열 [69, 10, 30, 3, 16, 8]에서 최솟값 3을 찾아 첫 번째 원소 69와 교환한다.
2. 남은 구간 [10, 30, 69, 16, 8]에서 최솟값 8을 찾아 두 번째 위치의 10과 교환한다.
3. 남은 구간 [30, 69, 16, 10]에서 최솟값 10을 찾아 세 번째 위치의 30과 교환한다.
4. 남은 구간 [69, 16, 30]에서 최솟값 16을 찾아 네 번째 위치의 69와 교환한다.
5. 남은 구간 [69, 30]에서 최솟값 30을 찾아 다섯 번째 위치에 둔다.

왜 이 순서인가: 앞쪽 정렬 완료 구간을 한 칸씩 늘리면서, 각 단계의 입력은 바로 직전 단계에서 확정된 앞부분을 제외한 미정렬 구간이기 때문이다.

NoteBase

선지별 해설

① O: 각 단계마다 미정렬 구간의 최솟값이 앞자리로 이동한다. step 1에서 3, step 2에서 8, step 3에서 10이 차례로 확정되므로 선택 정렬의 전형적인 진행이다.
② X: 버블 정렬은 인접한 두 원소를 반복 교환하여 큰 값을 뒤로 보내는 방식이다. 틀린 표현은 "버블 정렬"이다. 올바르게는 첫 번째 패스가 끝나면 최댓값 69가 맨 뒤에 위치하는 모습이 나타나야 한다.
③ X: 합병 정렬은 배열을 분할한 뒤 정렬된 부분배열을 합치면서 전체를 만든다. 틀린 표현은 "합병 정렬"이다. 올바르게는 중간 단계에서 앞부분만 확정되는 교환 패턴이 아니라, 분할과 병합 결과로 정렬된 부분열이 커지는 모습이 나타난다.
④ X: 기수 정렬은 자릿수별로 버킷에 분배하고 다시 모으는 과정을 반복한다. 틀린 표현은 "기수 정렬"이다. 올바르게는 1의 자리, 10의 자리 순서의 안정적 분배 결과가 나타나야 하며, 매 단계마다 최솟값 하나를 골라 앞에 두지는 않는다.

오답 포인트

앞쪽부터 정렬된 값이 늘어나는 모습만 보고 버블 정렬로 착각하기 쉽다. 그러나 버블 정렬은 인접 교환의 누적으로 최댓값이 뒤에서 먼저 확정되고, 선택 정렬은 미정렬 구간의 최솟값이 앞에서부터 확정된다는 점이 핵심 차이이다.

다시 나오면 이렇게 판단

정렬 과정 문제가 나오면 먼저 한 단계가 끝날 때 확정되는 위치가 앞쪽인지 뒤쪽인지 본다. 최솟값이 앞에서부터 하나씩 고정되면 선택 정렬이고, 최댓값이 뒤에서부터 하나씩 고정되면 버블 정렬로 판단할 수 있다.

암기 포인트

선택 정렬은 매 패스마다 최솟값 선택 후 앞자리 확정, 버블 정렬은 인접 교환 후 뒷자리 확정이다.

NoteBase

문 5. 다음 중위 표기법 수식을 후위 표기법 수식으로 옳게 변환한 것은?

(A * B) - (C / D) - E

① A B * C D E - / -
② A B * C D / - E -
③ A B * C D - / E -
④ * A B - / C D - E

정답 2번

출제 포인트

중위 표기식을 후위 표기식으로 바꿀 때 괄호, 연산자 우선순위, 그리고 같은 우선순위 연산자의 좌결합성을 정확히 적용하는지를 묻는 문제이다.

올바른 순서

1. `(A * B)`를 먼저 후위 표기식으로 바꾸면 `A B *`가 된다.
2. `(C / D)`를 후위 표기식으로 바꾸면 `C D /`가 된다.
3. 두 부분식을 첫 번째 뺄셈으로 결합하면 `A B * C D / -`가 된다.
4. 식 전체는 `((A * B) - (C / D)) - E`이므로 마지막에 `E`를 붙이고 다시 뺄셈을 적용해 `A B * C D / - E -`가 된다.

왜 이 순서: 뺄셈 `-`은 같은 우선순위에서 좌결합이므로 앞의 `((A * B) - (C / D))` 결과가 먼저 완성되어야 하고, 그 뒤에 `E`와 마지막 뺄셈이 붙는다.

NoteBase

선지별 해설

① X
`A B * C D E - / -`는 `D E -`가 먼저 계산된 뒤 `C / (D - E)`가 되는 형태이다. 원식의 `(C / D)`와 전혀 다르다. 틀린 표현은 `C D E - /` 부분이며, 올바르게는 `C D /`가 되어야 한다.

② O
`A B * C D / - E -`는 `(A * B)`를 `A B *`, `(C / D)`를 `C D /`로 만든 뒤, 첫 번째 뺄셈과 마지막 `- E`를 왼쪽에서 오른쪽으로 적용한 정확한 후위 표기식이다.

③ X
`A B * C D - / E -`는 `C D -`가 먼저 계산되어 `(C - D)`가 되고, 그 앞의 `A B *`와 나눗셈으로 결합되어 `((A * B) / (C - D)) - E`가 된다. 원식의 두 번째 괄호는 뺄셈이 아니라 나눗셈이다. 틀린 표현은 `C D - /` 부분이며, 올바르게는 `C D / -`가 되어야 한다.

④ X
후위 표기식은 연산자가 항상 해당 피연산자들 뒤에 와야 한다. 그런데 첫 기호가 `*`이므로 시작부터 후위 표기식 규칙을 어긴다. 틀린 표현은 맨 앞의 `*` 배치이며, 올바르게는 `A B *`처럼 피연산자 뒤에 연산자가 와야 한다.

오답 포인트

첫 번째 뺄셈과 두 번째 뺄셈을 한 번에 처리하거나, `(C / D)`를 `C D /`로 바꾸지 못하고 `C D -` 또는 `C D E - /`처럼 다른 부분식으로 읽으면 오답이 된다. 특히 `-`가 두 번 나온다고 해서 `E`를 중간에 먼저 끼워 넣는 실수가 자주 나온다.

다시 나오면 이렇게 판단

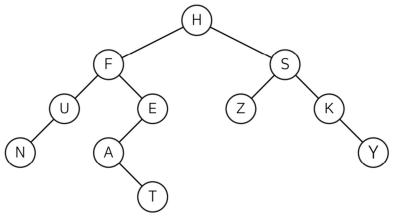
중위식을 후위식으로 바꿀 때는 괄호 안 부분식을 각각 독립적으로 후위화한 뒤, 바깥 연산자를 좌결합 순서대로 뒤에 붙이면 된다. 같은 우선순위의 `-`, `/`가 이어지면 왼쪽 부분식이 먼저 완성되는지를 확인하면 된다.

암기 포인트

후위 표기식은 `부분식 후위화 -> 바깥 연산자 뒤로`의 반복이며, 좌결합 연산자는 왼쪽 결과를 먼저 확정한다.

NoteBase

문 6. 그림과 같이 연결되어 있는 11개의 여행지를 관광객들이 모두 방문할 수 있도록 안내하기 위해 C언어로 구현한 Course() 함수를 다음과 같이 제시하고 있다. 여행지 H를 파라미터로 하여 알고리즘을 호출했을 때, 출력결과의 (가) ~ (라)에 들어갈 내용을 바르게 연결한 것은? (단, 상위 노드 기준으로 n->left는 왼쪽으로 순회하고 n->right는 오른쪽으로 순회한다)




```
typedef struct node {
    char name;
    struct node *left;
    struct node *right;
} node;

void Course(node *n) {
    if(n != null) {
        Course(n->left);
        Course(n->right);
        printf("%c->", n->name);
    }
}
```

○ 출력결과

(가) -> ____ -> (나) -> ____ -> (다) -> ____ ->

____ -> ____ -> ____ -> (라) -> ____ ->

① (가): H, (나): A, (다): Z, (라): K

② (가): H, (나): T, (다): S, (라): K

③ (가): N, (나): H, (다): U, (라): T

④ (가): N, (나): T, (다): E, (라): S

정답 4번

출제 포인트

이 문제는 C 함수의 재귀 호출 순서가 왼쪽 서브트리, 오른쪽 서브트리, 현재 노드인 후위 순회라는 점을 트리 구조에 대입해 출력 순서를 복원하는 문제이다.

NoteBase

선지별 해설

① X: (가)=N은 맞지만 (나)=A, (다)=Z, (라)=K가 틀리다. 실제 후위 순회는 N→U→T→A→E→F→Z→Y→K→S→H이므로 올바르게는 (나)=T, (다)=E, (라)=S이다.
② X: 루트 H는 두 서브트리를 모두 방문한 뒤 맨 마지막에 출력되므로 (가)=H가 될 수 없다. 또한 (다)는 S가 아니라 E이고 (라)는 K가 아니라 S이다. 올바르게는 (가)=N, (나)=T, (다)=E, (라)=S이다.
③ X: (가)=N은 맞지만 (나)=H와 (다)=U, (라)=T가 틀리다. H는 마지막 노드이고 U는 두 번째 노드이며 T는 세 번째 노드이다. 올바르게는 (나)=T, (다)=E, (라)=S이다.
④ O: 후위 순회 결과가 N→U→T→A→E→F→Z→Y→K→S→H이므로 (가)=N, (나)=T, (다)=E, (라)=S가 정확하다.

오답 포인트

전위 순회나 중위 순회와 혼동하면 상위 노드 H, F, S가 너무 앞에 나온다고 착각하기 쉽다. 특히 `printf`가 마지막이라는 점을 놓치면 루트 H가 처음 출력된다고 오판하게 된다.

다시 나오면 이렇게 판단

재귀 순회 문제는 함수 안에서 출력문 위치를 먼저 본다. 출력문이 두 재귀 호출 뒤에 있으면 후위 순회이고, 각 서브트리도 왼쪽-오른쪽-루트 순서로 순회를 적어 내려가면 된다.

암기 포인트

후위 순회는 left-right-root이며 출력문이 마지막이면 루트도 마지막이다.

NoteBase

문 7. 다음 데이터를 순서대로 삽입하여 불균형이 발생했을 때 회전 연산을 수행하여 균형 상태를 유지하는 AVL 트리를 구성하였다. 구성된 AVL 트리에서 단말 노드(leaf node)에 해당하는 값만을 모두 고르면?

50, 60, 70, 90, 80, 75, 73, 72, 78

① 72, 73, 90

② 72, 75, 90

③ 50, 73, 75, 90

④ 50, 72, 78, 90

정답 4번

출제 포인트

이 문제는 AVL 트리에 데이터를 순차 삽입할 때 불균형 유형에 맞는 회전을 적용하여 최종 단말 노드를 추적할 수 있는지를 묻는 문제이다.

올바른 순서

1. 50, 60, 70을 넣으면 50-60-70의 오른쪽-오른쪽 불균형이 생겨 50을 기준으로 좌회전하여 60이 루트가 된다.
2. 90을 넣으면 아직 균형 범위 안이므로 회전이 없다.
3. 80을 넣으면 70의 오른쪽-왼쪽 불균형이 생겨 70-90-80 부분에서 RL 회전을 하여 80이 부분트리 루트가 된다.
4. 75를 넣으면 60의 오른쪽-왼쪽 불균형이 생겨 루트 쪽에서 RL 회전을 하여 70이 루트가 된다.
5. 73을 넣으면 균형 범위 안이므로 회전이 없다.
6. 72를 넣으면 75의 왼쪽-왼쪽 불균형이 생겨 75를 기준으로 우회전하여 73이 부분트리 루트가 된다.
7. 78을 넣으면 80의 왼쪽-오른쪽 불균형이 생겨 73-80 부분에서 LR 회전을 하여 75가 부분트리 루트가 된다.

왜 이 순서: AVL 삽입은 매 단계마다 삽입 경로를 따라 가장 먼저 불균형이 난 조상 노드에서 회전해야 하므로, 마지막 단말 노드는 각 회전 뒤 부모-자식 관계가 어떻게 바뀌는지까지 따라가야 결정된다.

NoteBase

선지별 해설

① X: 73은 최종 트리에서 왼쪽 자식 72를 가지는 내부 노드이므로 단말 노드가 아니다. 틀린 표현은 '73이 단말 노드이다'이고, 올바르게는 73 대신 50이 단말 노드에 포함된다.
② X: 75는 마지막에 78을 삽입할 때 80의 왼쪽-오른쪽 불균형을 정리하는 과정에서 부분트리의 부모가 되므로 단말 노드가 아니다. 틀린 표현은 '75가 최종 단말 노드이다'이고, 올바르게는 75 대신 78이 단말 노드가 된다.
③ X: 73과 75를 모두 단말 노드로 본 것이 틀렸다. 72를 넣은 직후에는 75가 단말 노드일 수 있지만, 78을 넣고 회전한 최종 상태에서는 73과 75가 모두 내부 노드가 된다. 올바르게는 73, 75 대신 72와 78이 단말 노드에 들어간다.
④ O: 최종 AVL 트리는 루트 70, 왼쪽 부분트리 60-50, 오른쪽 부분트리 75-73-72와 80-78-90의 형태가 되므로 단말 노드는 50, 72, 78, 90이다.

오답 포인트

중간 단계의 단말 노드를 최종 단말 노드로 착각하면 틀린다. 특히 72를 넣은 직후의 단말 노드가 50, 72, 75, 90인데, 마지막 78 삽입 후 다시 회전이 일어나 75가 내부 노드로 바뀌는 점이 함정이다.

다시 나오면 이렇게 판단

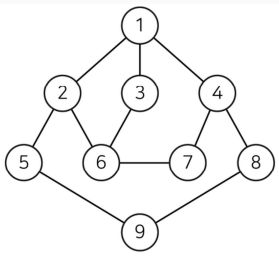
AVL 문제는 삽입할 때마다 경로를 따라 불균형 조상을 찾고, 불균형 유형이 LL, LR, RL, RR 중 무엇인지 먼저 정한 뒤 회전 후 부모-자식 관계를 다시 적어 본다. 마지막 삽입까지 끝난 뒤 자식이 없는 노드만 따로 추리면 단말 노드를 정확히 고를 수 있다.

암기 포인트

AVL 단말 노드는 중간 상태가 아니라 최종 회전까지 끝난 뒤의 자식 없는 노드 집합이다.

NoteBase

문 8. 그림과 같은 그래프를 대상으로 다음 알고리즘을 적용하였을 때, 결과에 대한 설명으로 옳지 않은 것은? (단, 탐색의 시작 정점은 1이고, Q는 큐 객체이며, Q.enqueue()와 Q.dequeue()는 큐 Q에 대한 삽입과 삭제 연산이고, 각 정점의 방문을 표시하는 것은 visited[] 배열을 이용한다)




```
_FS(v) {
    visited[v] <- YES;      // 정점 v에 대한 방문 표시
    Q.enqueue(v)
    while(!Q.empty()) {     // Q가 비어있지 않은 동안
        w <- Q.dequeue();
        for all u ∈ (w에 인접한 정점) {
            if(visited[u] != YES) {
                visited[u] <- YES;
                Q.enqueue(u);
            }
        }
    }
}
```

① 알고리즘의 수행에 따른 노드의 방문 순서는 1→2→5→9→8→4→7→6→3이다.

② 너비 우선 탐색(BFS, Breadth First Search) 알고리즘이다.

③ 시작 정점으로부터 가까운 정점들을 먼저 방문하고 멀리 떨어져 있는 정점들을 나중에 방문하는 탐색방법이다.

④ 알고리즘은 그래프의 모든 정점을 방문한다.

정답 1번

출제 포인트

이 문제는 큐를 사용하는 너비 우선 탐색의 층별 방문 원리와 방문 표시 시점이 enqueue 시점이라는 점을 묻는 문제이다.

NoteBase

선지별 해설

① X
제시된 순서는 1의 인접 정점인 3, 4보다 거리 2인 정점 5를 먼저 방문하므로 BFS의 층별 방문 원리와 맞지 않는다.
틀린 표현은 "1→2→5→9→8→4→7→6→3"이다. 올바르게는 인접 정점의 세부 처리 순서가 어떠하든 시작 정점 1에서 거리 1인 정점 2, 3, 4가 거리 2인 정점 5, 6, 7, 8보다 먼저 방문된다.

② O
큐를 사용하여 먼저 발견한 정점을 먼저 꺼내고 그 정점의 인접 정점을 확장하므로 너비 우선 탐색(BFS) 설명이 맞다.

③ O
BFS는 시작 정점에서 가까운 정점들을 먼저 방문하고 더 멀리 있는 정점들을 나중에 방문하는 방식이므로 진술이 맞다.

④ O
주어진 그래프는 시작 정점 1에서 모든 정점 2~9로 도달할 수 있는 연결 그래프이므로 이 알고리즘을 1에서 시작하면 모든 정점을 방문한다.

오답 포인트

깊이 우선 탐색처럼 한 갈래를 끝까지 따라가는 순서를 BFS 순서로 착각하기 쉽다. 이 문항의 함정은 ①처럼 5를 3, 4보다 먼저 놓아 BFS와 DFS를 혼동하게 만드는 데 있다.

다시 나오면 이렇게 판단

큐가 보이면 먼저 같은 거리의 정점들이 모두 처리되는지 확인한다. 시작 정점의 인접 정점보다 더 먼 정점이 먼저 나오면 BFS 결과가 아니다.

암기 포인트

BFS는 큐로 레벨 순서대로, DFS는 스택 또는 재귀로 한 갈래 우선으로 진행된다.

NoteBase

문 9. 다음 C언어 함수 ABC()에 대한 설명으로 옳은 것은?

```
void ABC(int A[], int n) {
    int i, j, tmp;
    for(i = n - 1; i > 0; i--) {
        j = 0;
        while(j < i) {
            if(A[j] < A[j + 1]) {
                tmp = A[j];
                A[j] = A[j + 1];
                A[j + 1] = tmp;
            }
            j = j + 1;
        }
    }
}
```

① 코드 수행 완료 후, A[0]에는 가장 큰 원소가 저장되어 있다.

② 코드 수행 완료 후, 배열 A의 원소는 오름차순으로 정렬되어 있다.

③ 최선의 경우와 최악의 경우의 수행 시간 복잡도가 다르다.

④ 코드는 삽입 정렬(insertion sort)을 구현한 것이다.

정답 1번

출제 포인트

이 문제는 버블 정렬의 비교·교환 방향과 그에 따른 정렬 순서, 시간 복잡도, 정렬 알고리즘 구분을 묻는다.

핵심 개념

버블 정렬은 인접한 두 원소를 반복 비교해 교환하는 정렬이다. 이 코드에서는 $A[j] < A[j+1]$일 때 교환하므로 작은 값이 오른쪽으로 밀리고 큰 값이 왼쪽으로 이동하여 최종적으로 내림차순이 된다. 또한 교환 여부를 검사해 조기 종료하는 장치가 없으므로 최선과 최악의 시간 복잡도는 같다.

NoteBase

선지별 해설

① O: 인접 원소를 비교하면서 왼쪽 값이 더 작으면 교환하므로 큰 값이 앞쪽으로 이동한다. 반복이 끝나면 배열은 내림차순이 되고, 따라서 $A[0]$에는 가장 큰 원소가 저장된다.
② X: 틀린 표현은 "오름차순"이다. 실제로는 $A[j] < A[j+1]$일 때 교환하므로 큰 값이 앞으로 와서 배열은 내림차순으로 정렬된다.
③ X: 틀린 표현은 "최선의 경우와 최악의 경우의 수행 시간 복잡도가 다르다"이다. 실제로는 교환이 없을 때 중단하는 조건이 없으므로 최선과 최악 모두 $O(n^2)$이다.
④ X: 틀린 표현은 "삽입 정렬(insertion sort)"이다. 실제로는 인접한 두 원소를 반복 비교·교환하는 버블 정렬을 구현한 것이다.

오답 포인트

버블 정렬은 보통 큰 값이 오른쪽 끝으로 이동하는 오름차순 예시로 많이 외워 두기 때문에, 비교 연산자 방향이 반대라는 점을 놓치면 ②를 고르기 쉽다. 또 조기 종료 최적화가 없는 기본형이라는 점을 놓치면 ③도 그럴듯하게 보인다.

다시 나오면 이렇게 판단

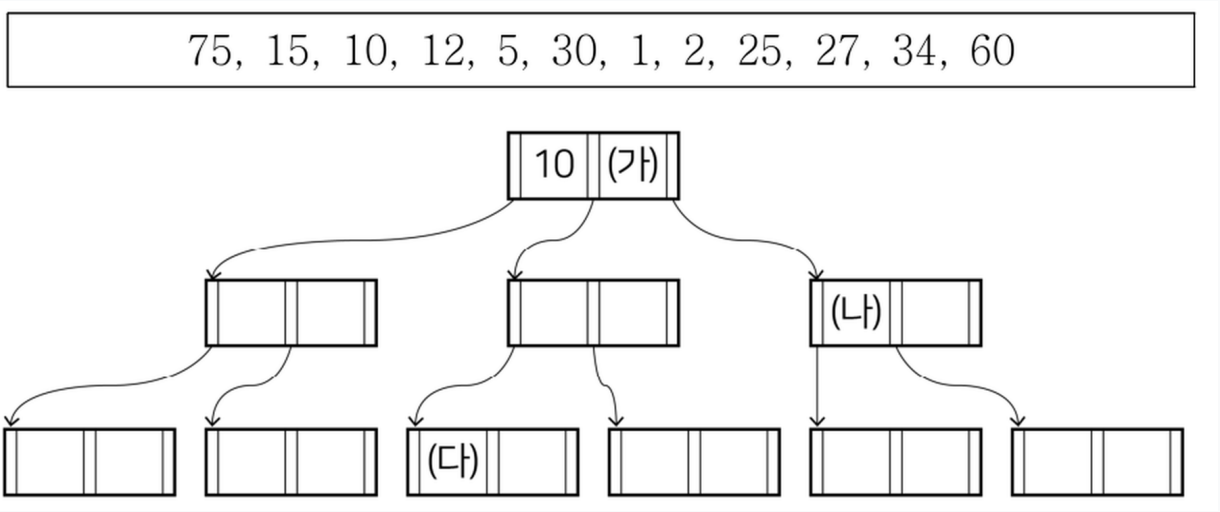
정렬 코드가 나오면 먼저 어떤 두 원소를 비교하는지와 교환 조건을 본다. 그다음 큰 값이 왼쪽으로 가는지 오른쪽으로 가는지 판단해 정렬 방향을 정하고, 마지막으로 조기 종료 조건 유무로 최선 시간 복잡도 변화가 있는지 확인한다.

암기 포인트

버블 정렬은 인접 비교·교환이다. $<$이면 큰 값이 앞으로 가서 내림차순, $>$이면 작은 값이 앞으로 가서 오름차순이 된다.

NoteBase

문 10. 다음 자료를 차수가 3인 B-트리에 순서대로 삽입하여 그림과 같은 B-트리를 생성하였을 때, (가) ~ (다)에 들어갈 값을 바르게 연결한 것은?



① (가): 25, (나): 34, (다): 12

② (가): 25, (나): 60, (다): 15

③ (가): 30, (나): 34, (다): 15

④ (가): 30, (나): 60, (다): 12

정답 4번

출제 포인트

차수 3 B-트리에서 한 노드가 가질 수 있는 최대 키 수가 2개라는 점과, 삽입 중 키가 3개가 되면 가운데 키를 부모로 올려 분할하는 과정을 묻는 문제이다.

NoteBase

선지별 해설

① X
- 2를 삽입하는 순간 기존 루트 [10, 15]가 분할되어 루트는 [10]이 된다. 이후 25, 27, 34, 60을 넣으면 최종 루트는 [10, 30]이 되므로 (가)=25는 틀리다.
- 오른쪽 리프 [34, 75]에 60을 넣으면 [34, 60, 75]가 되어 가운데 값 60이 부모로 올라가므로 (나)=34도 틀리다.
- 틀린 표현은 \"(가)=25, (나)=34\"이고, 올바르게는 \"(가)=30, (나)=60\"이다.

② X
- 최종 루트는 [10, 30]이므로 (가)=25는 틀리다.
- 가운데 내부 노드는 [15]이고 그 왼쪽 리프가 [12], 오른쪽 리프가 [25, 27]이므로 (다)=15도 틀리다.
- 틀린 표현은 \"(가)=25, (다)=15\"이고, 올바르게는 \"(가)=30, (다)=12\"이다.

③ X
- (가)=30은 맞지만, 오른쪽 내부 노드는 60이 승격된 결과 [60]이 되므로 (나)=34는 틀리다.
- 가운데 내부 노드 [15]의 왼쪽 자식은 [12]이므로 (다)=15도 틀리다.
- 틀린 표현은 \"(나)=34, (다)=15\"이고, 올바르게는 \"(나)=60, (다)=12\"이다.

④ O
- 75, 15, 10, 12, 5, 30, 1, 2를 순서대로 넣으면 루트가 [10]으로 정리되고, 오른쪽 부분트리에는 [15] 아래 [12]와 [30, 75]가 놓인다.
- 이어서 25, 27, 34를 넣으면 가운데 내부 노드는 [15, 30]이 되고 자식은 [12], [25, 27], [34, 75]가 된다.
- 마지막으로 60을 넣으면 [34, 60, 75]가 분할되면서 60이 부모로 올라가고, 부모 [15, 30]은 다시 분할되어 30이 루트로 올라간다. 따라서 (가)=30, (나)=60, (다)=12가 된다.

오답 포인트

차수 3을 최소 차수처럼 오해해서 한 노드에 3개 키를 둘 수 있다고 보면 승격 값이 모두 어긋난다. 또 마지막 60 삽입에서 34가 아니라 가운데 값 60이 부모로 올라간다는 점을 놓치면 (나)를 34로 잘못 고르기 쉽다.

다시 나오면 이렇게 판단

차수 3 B-트리는 노드당 최대 키 2개라는 점을 먼저 고정하고, 삽입할 때마다 키가 3개가 되는 순간 가운데 키를 부모로 올리면서 루트부터 최종 구조를 다시 적으면 된다.

암기 포인트

차수 3 B-트리는 최대 2키, 3키가 되면 가운데 값 승격이라는 규칙으로 정리된다.

NoteBase

문 11. 다음은 재귀 호출을 이용하여 피보나치(Fibonacci) 수열값을 찾는 파이썬 코드이다. 이 코드의 시간 복잡도는? (단, n은 0 이상의 정수이다)

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

① $O(n)$

② $O(n^2)$

③ $O(2^n)$

④ $O(\log n)$

정답 3번

출제 포인트

이 문제는 메모이제이션이 없는 단순 재귀 피보나치 함수의 호출 수를 점화식으로 세워 시간 복잡도를 판단하는 문제이다.

조건 정리

주어진 값: fib(n)은 $n = 0$이면 0, $n = 1$이면 1을 반환하고 그 외에는 fib(n-1) + fib(n-2)를 계산한다.
구해야 할 값: 이 코드의 시간 복잡도이다.
주의할 조건: 캐시나 동적 계획법이 없으므로 같은 부분문제를 여러 번 다시 계산한다.

사용할 공식

실행 시간을 $T(n)$이라 두면 $T(0) = T(1) = O(1)$, $T(n) = T(n-1) + T(n-2) + O(1)$이다.
이 공식을 쓰는 이유: $n \geq 2$인 호출은 fib(n-1)과 fib(n-2) 두 재귀 호출을 모두 수행하고, 현재 호출의 비교·덧셈 비용은 상수 시간이기 때문이다.

NoteBase

계산 과정

1. 기저 호출인 fib(0), fib(1)은 상수 시간이다.
2. 그 외의 호출은 두 하위 호출을 만들므로 실행 시간은 $T(n) = T(n-1) + T(n-2) + O(1)$을 따른다.
3. 이 점화식의 성장률은 피보나치 수와 같아서 $T(n) = \Theta(F_n) = \Theta(\varphi^n)$이다.
4. $\varphi \approx 1.618 < 2$이므로 보기 기준으로는 지수 시간인 $O(2^n)$이 맞다.

선지 판단

① X: $O(n)$은 한 번씩만 상태를 계산하는 반복문이나 메모이제이션 피보나치에서 성립한다. 이 코드는 같은 fib(k)를 여러 번 다시 호출하므로 선형 시간이 아니다. 틀린 표현은 O(n)이고, 올바르게는 중복 호출 때문에 지수 시간이다.
② X: $O(n^2)$는 다항 시간인데, 이 재귀 트리의 노드 수는 피보나치 수처럼 증가하므로 다항식으로 억제되지 않는다. 틀린 표현은 O(n^2)이고, 올바르게는 $\Theta(\varphi^n)$이며 보기에서는 $O(2^n)$이다.
③ O: 각 비기저 호출이 두 갈래로 분기하여 전체 호출 수가 피보나치 수열처럼 늘어난다. 따라서 정확한 성장률은 $\Theta(\varphi^n)$이고, 이를 포함하는 상한으로 $O(2^n)$이 맞다.
④ X: $O(\log n)$은 이진 탐색처럼 입력 크기가 매 단계 절반으로 줄 때 나타난다. 이 코드는 $n-1$, $n-2$로 줄어들 뿐 아니라 두 번 분기하므로 로그 시간이 아니다. 틀린 표현은 O(log n)이고, 올바르게는 지수 시간이다.

오답 포인트

n이 1씩 또는 2씩 줄어드니 선형에 가깝다고 오해하기 쉽지만, 핵심은 감소 폭이 아니라 같은 값을 반복 계산하는 분기 수이다. 메모이제이션 유무를 보지 않으면 ①이나 ②로 흔들리기 쉽다.

다시 나오면 이렇게 판단

재귀 시간 복잡도는 먼저 한 호출이 몇 개의 하위 호출을 만드는지 본다. 그다음 같은 부분문제가 캐시 없이 반복되는지 확인한다. 분기 재귀이면서 중복 계산이 있으면 선형·로그형보다 지수형을 우선 의심할 수 있다.

암기 포인트

단순 재귀 피보나치 = 중복 부분문제 + 2분기 재귀 = $\Theta(\varphi^n)$, 보기에서는 $O(2^n)$이다.

NoteBase

문 12. 분할 정복(divide-and-conquer) 방식의 정렬 알고리즘에 대한 설명으로 옳지 않은 것은?

① 2-way 합병 정렬의 분할은 분할 대상을 반으로 나눈다.

② 퀵정렬(quick sort)의 분할은 분할 대상에 대하여 피봇(pivot)을 기준으로 나눈다.

③ 합병 정렬과 다르게 퀵정렬에서는 모든 분할이 완료되면 정렬도 완료된다.

④ 합병 정렬은 추가 데이터 저장공간이 필요 없는 제자리 정렬 알고리즘이다.

정답 4번

출제 포인트

이 문제는 분할 정복 정렬에서 합병 정렬과 퀵정렬의 분할 기준, 정렬 완료 시점, 추가 공간 사용 여부를 구분하는 문제이다.

핵심 개념

2-way 합병 정렬은 입력 구간을 거의 절반씩 나누어 각각 정렬한 뒤 다시 합병하는 정렬이다. 결정 차이축은 중간 지점 기준 반분과 합병 단계의 존재이다. 키워드는 반분, 합병, 보조 배열이다.
퀵정렬은 피벗을 기준으로 작은 값과 큰 값을 양쪽으로 분할하며 재귀적으로 정렬하는 정렬이다. 결정 차이축은 피벗 기준 분할과 별도 합병 단계의 부재이다. 키워드는 pivot, partition, 제자리 정렬 가능이다.

선지별 해설

① O. 2-way 합병 정렬은 배열을 두 부분으로 반씩 나누는 방식이다. 분할 기준은 값의 크기가 아니라 구간의 중간 지점이므로 옳은 설명이다.
② O. 퀵정렬은 피벗보다 작은 원소와 큰 원소로 분할한다. 따라서 분할의 기준이 피벗이라는 설명은 맞다.
③ O. 퀵정렬은 분할 과정에서 피벗의 최종 위치가 확정되고 별도의 합병 단계가 없다. 반면 합병 정렬은 분할이 끝나도 합병을 해야 정렬이 완성되므로 이 선지는 옳다.
④ X. 틀린 표현은 \"추가 데이터 저장공간이 필요 없는 제자리 정렬 알고리즘\"이다. 일반적인 합병 정렬은 합병 과정에서 보조 배열이 필요하므로 제자리 정렬이 아니며, 올바르게는 \"합병 정렬은 보통 추가 저장공간이 필요한 정렬 알고리즘이다.\"이다.

오답 포인트

합병 정렬과 퀵정렬이 모두 분할 정복이라는 공통점만 보고 공간 특성까지 같다고 오해하기 쉽다. 이 문항은 ④에서 퀵정렬의 제자리 정렬 가능 성질을 합병 정렬에 바꿔 붙인 점이 함정이다.

다시 나오면 이렇게 판단

분할 기준이 중간 지점이면 합병 정렬이고 피벗이면 퀵정렬이다. 분할 뒤에 다시 합치는 단계가 필요하면 합병 정렬이고, 보조 배열 없이 끝난다고 하면 합병 정렬 설명으로는 틀린 진술일 가능성이 크다.

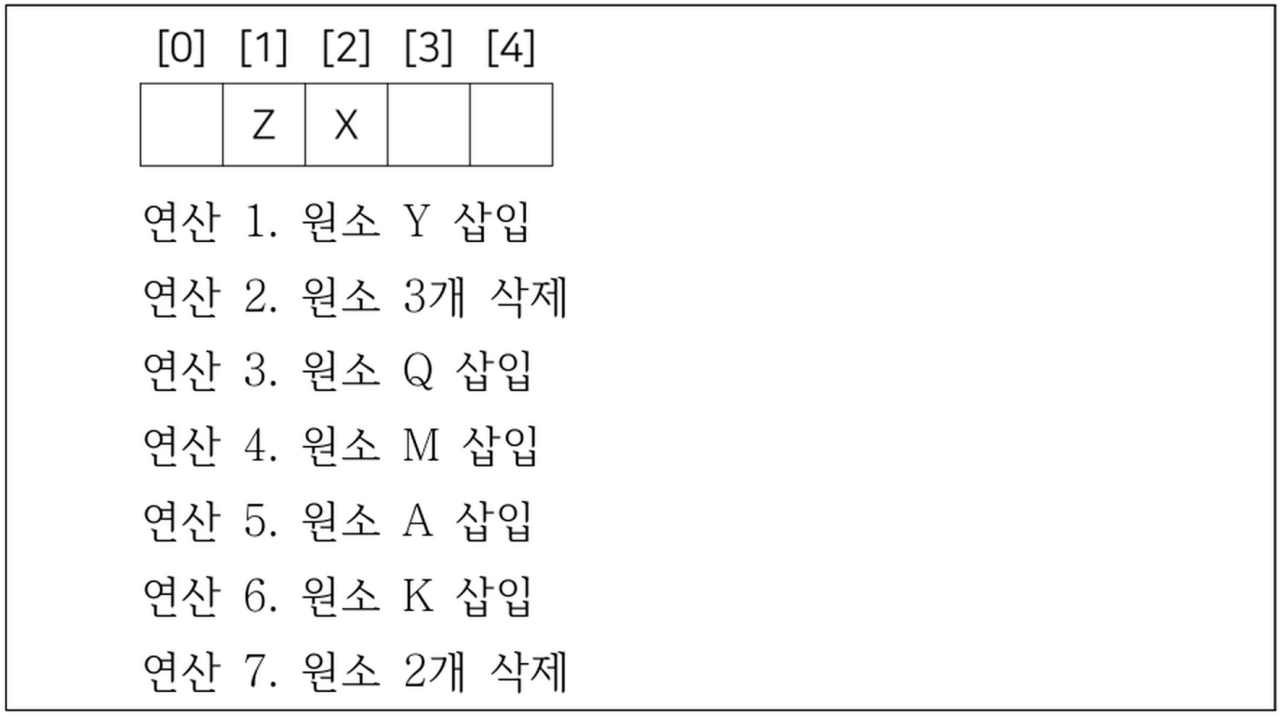
NoteBase

암기 포인트

합병 정렬은 반으로 나누고 다시 합치며 공간을 더 쓰고, 퀵정렬은 피벗으로 나누며 별도 합병이 없다.

NoteBase

문 13. 배열로 운영하는 원형 큐에 대해 다음과 같이 연산 1에서 7까지의 연산을 차례로 수행했을 때, 수행이 완료된 후 큐의 상태로 옳은 것은? (단, 현재 상태에서 front = 0, rear = 2이다)



① [0] [1] [2] [3] [4]

M				Q

② [0] [1] [2] [3] [4]

	A	K		

③ [0] [1] [2] [3] [4]

			Y	Q

④ [0] [1] [2] [3] [4]

M	A			

정답 2번

출제 포인트

배열로 구현한 원형 큐에서 삭제와 삽입이 일어날 때 front와 rear가 어떻게 이동하는지, 그리고 삭제 후 다시 원형으로 감기는 위치를 정확히 추적하는지를 묻는 문제이다.

NoteBase

올바른 순서

1. 초기 상태는 front=0, rear=2이고 원소는 [1]=Z, [2]=X이다.
2. 연산 1에서 Y를 삽입하면 rear가 3으로 이동하고 [3]=Y가 된다.
3. 연산 2에서 원소 3개를 삭제하면 Z, X, Y가 차례로 삭제되어 front=3, rear=3인 공백 큐가 된다.
4. 연산 3~6에서 Q, M, A, K를 삽입하면 [4]=Q, [0]=M, [1]=A, [2]=K가 되고 front=3, rear=2가 된다.
5. 연산 7에서 원소 2개를 삭제하면 front 다음 위치부터 삭제되므로 Q와 M이 제거되고 [1]=A, [2]=K만 남는다.

왜 이 순서인가: 삭제는 항상 front의 다음 칸에서 시작되고 삽입은 rear의 다음 칸에 이루어지므로, 배열의 모양보다 front와 rear의 이동 규칙이 우선이다.

선지별 해설

① X: [0]=M, [4]=Q 상태는 연산 6까지의 중간 상태에서 A와 K를 누락한 그림이거나 삭제 방향을 잘못 본 경우이다. 틀린 표현은 최종 상태를 M과 Q가 남는 것으로 본 점이다. 올바르게는 연산 7에서 Q와 M이 삭제되고 A와 K가 남는다.

② O: 연산 6까지 수행하면 큐의 순서는 Q, M, A, K이다. 연산 7에서 앞의 두 원소 Q와 M이 삭제되므로 배열에는 [1]=A, [2]=K만 남는다.

③ X: Y는 연산 2에서 삭제되는 세 원소 중 마지막 원소이므로 최종 상태에 남을 수 없다. 틀린 표현은 Y와 Q가 남는 것으로 본 점이다. 올바르게는 Y는 연산 2에서 제거되고 최종 상태는 A와 K이다.

④ X: M과 A가 남으려면 삭제가 rear 쪽에서 일어나야 하지만, 원형 큐의 삭제는 front 다음 위치에서 시작된다. 틀린 표현은 삭제 방향을 잘못 잡은 점이다. 올바르게는 Q와 M이 삭제되고 A와 K가 남는다.

오답 포인트

가장 흔한 실수는 front가 가리키는 칸에 원소가 있다고 보는 것과, 삭제 후 공백 큐가 되면 삽입 위치가 다시 [0]부터 시작한다고 단순화하는 것이다. 이 문항에서는 연산 2 후 공백 큐가 되어도 front=rear=3 상태를 유지한 채, 다음 삽입이 [4]부터 시작된다는 점이 함정이다.

다시 나오면 이렇게 판단

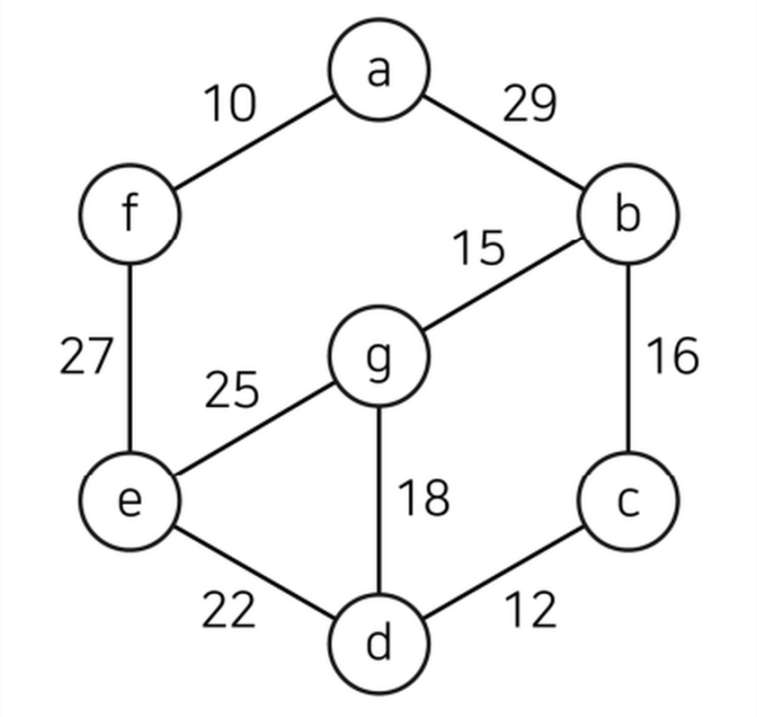
원형 큐 절차형 문제는 현재 원소 배열만 보지 말고 매 연산마다 삭제는 front를 한 칸 전진시키고 삽입은 rear를 한 칸 전진시킨다고 적어 가며 판단한다. 공백 큐 여부는 front와 rear가 같은지로 확인하고, 그 다음 삽입 위치를 다시 계산하면 된다.

암기 포인트

원형 큐는 삭제 때 front가 움직이고 삽입 때 rear가 움직이며, 공백 상태는 front=rear이다.

NoteBase

문 14. 다음 그래프를 대상으로 크루스칼(Kruskal) 알고리즘을 적용하여 최소 신장 트리(MST, Minimum Spanning Tree)를 구성하고자 할 때, 생성되는 MST에 5번째 추가되는 간선과 비용을 바르게 연결한 것은?



① 간선: (b, c), 비용: 16

② 간선: (d, g), 비용: 18

③ 간선: (d, e), 비용: 22

④ 간선: (e, g), 비용: 25

정답 3번

출제 포인트

이 문제는 가중치가 작은 간선부터 보되 사이클이 생기면 제외하는 Kruskal 알고리즘의 선택 순서를 정확히 적용할 수 있는지를 묻는 문제이다.

NoteBase

선지별 해설

① X
(b, c), 16은 네 번째로 추가된다. 앞에서 (a, f) 10, (c, d) 12, (b, g) 15가 먼저 선택되고, 그다음 (b, c) 16이 서로 다른 연결요소를 잇기 때문에 채택된다.
틀린 표현은 '5번째 추가'이다. 올바르게는 '(b, c), 16은 4번째로 추가된다.'이다.

② X
(d, g), 18은 비용 순서상 다음 후보이지만 이미 (b, g), (b, c), (c, d)를 통해 d와 g가 같은 연결요소에 있으므로 사이클이 생겨 추가되지 않는다.
틀린 표현은 '(d, g)가 MST에 추가된다'는 점이다. 올바르게는 '(d, g), 18은 사이클 때문에 제외된다.'이다.

③ O
(d, e), 22는 (d, g), 18이 제외된 뒤 남은 간선 중 가장 비용이 작고, 아직 트리에 연결되지 않은 e를 기존 연결요소에 붙이므로 5번째로 추가된다.

④ X
(e, g), 25는 (d, e), 22가 먼저 선택된 뒤에는 e와 g가 이미 같은 연결요소에 속해 사이클을 만든다. 따라서 5번째 추가 간선이 될 수 없다.
틀린 표현은 '(e, g), 25가 5번째로 추가된다'는 점이다. 올바르게는 '5번째 추가 간선은 (d, e), 22이고, (e, g), 25는 제외된다.'이다.

오답 포인트

Kruskal에서는 '가중치가 몇 번째로 작은가'와 'MST에 몇 번째로 추가되는가'를 구분해야 한다. 18은 다섯 번째로 작은 비용이지만 사이클을 만들어 제외되므로, 다섯 번째로 추가되는 간선은 그다음으로 채택 가능한 22가 된다.

다시 나오면 이렇게 판단

간선을 비용 오름차순으로 나열한 뒤, 각 단계마다 양 끝 정점이 이미 같은 연결요소인지부터 판단한다. 같은 연결요소이면 건너뛰고 다음 간선을 보며, '몇 번째 추가'는 '몇 번째 검사'와 다르다는 점을 유지하면 된다.

암기 포인트

Kruskal은 작은 간선부터 보되 사이클이면 버리고, 새 정점을 연결하는 간선만 채택한다.

NoteBase

문 15. 문자열 매칭을 위한 알고리즘으로 옳은 것은?

① 합병 정렬 알고리즘

② 라빈 카프(Rabin-Karp) 알고리즘

③ 백트래킹(backtracking) 알고리즘

④ 후위 순회(postorder traversal) 알고리즘

정답 2번

출제 포인트

이 문제는 문자열 매칭 알고리즘과 정렬·탐색·트리 순회 알고리즘을 구분할 수 있는지를 묻는 문제이다.

핵심 개념

문자열 매칭 알고리즘은 본문 문자열 안에서 패턴 문자열이 나타나는 위치를 찾는 알고리즘이다. 라빈 카프는 해시 값을 이용해 패턴과 부분 문자열을 비교하는 문자열 매칭 알고리즘이다. 합병 정렬은 데이터를 정렬하는 분할 정복 기반 정렬 알고리즘이다. 백트래킹은 가능한 해를 탐색하다가 조건에 맞지 않으면 되돌아가는 탐색 기법이다. 후위 순회는 이진 트리에서 왼쪽 서브트리, 오른쪽 서브트리, 루트 순으로 방문하는 순회 방식이다.

선지별 해설

① X: 합병 정렬 알고리즘은 원소들을 크기 순서로 정렬하는 알고리즘이므로 문자열 매칭 알고리즘이 아니다. 틀린 표현은 "문자열 매칭"에 해당한다고 보는 부분이다. 올바르게는 "합병 정렬 알고리즘은 정렬 알고리즘이다."이다.
② O: 라빈 카프(Rabin-Karp) 알고리즘은 패턴과 본문 부분 문자열의 해시 값을 비교하여 문자열 매칭을 수행하는 대표적인 알고리즘이므로 옳다.
③ X: 백트래킹(backtracking) 알고리즘은 조합 탐색, 상태 공간 탐색, 제약 충족 문제에 쓰이는 일반 탐색 기법이므로 문자열 매칭 알고리즘으로 보지 않는다. 틀린 표현은 "문자열 매칭을 위한"이라고 보는 부분이다. 올바르게는 "백트래킹 알고리즘은 되돌아가며 해를 탐색하는 탐색 알고리즘이다."이다.
④ X: 후위 순회(postorder traversal)는 트리 자료구조를 방문하는 방법이므로 문자열 패턴을 찾는 알고리즘이 아니다. 틀린 표현은 "문자열 매칭"에 해당한다고 보는 부분이다. 올바르게는 "후위 순회 알고리즘은 트리 순회 알고리즘이다."이다.

오답 포인트

보기들이 모두 "알고리즘"이라는 이름을 갖고 있어 적용 대상이 서로 다르다는 점을 놓치기 쉽다. 문자열 매칭은 문자열 안에서 패턴을 찾는 문제이고, 정렬은 순서를 바꾸는 문제이며, 백트래킹은 경우의 수를 탐색하는 기법이고, 후위 순회는 트리를 방문하는 방식이라는 대상 차이를 구분해야 한다.

NoteBase

다시 나오면 이렇게 판단

선지를 보면 먼저 그 알고리즘이 무엇을 처리하는지 판단하면 된다. 패턴 문자열을 본문에서 찾으면 문자열 매칭이고, 원소의 순서를 재배열하면 정렬이며, 경우의 수를 되짚어 탐색하면 백트래킹이고, 트리의 노드 방문 순서를 정하면 순회 알고리즘이다.

암기 포인트

라빈 카프는 해시 기반 문자열 매칭, 합병 정렬은 정렬, 백트래킹은 탐색, 후위 순회는 트리 순회이다.

NoteBase

문 16. 다음은 단일 출발점 최단 경로를 찾기 위한 다익스트라(Dijkstra) 알고리즘을 의사(pseudo) 코드로 기술한 것이다. 이 코드에서 (가)에 들어갈 내용으로 옳은 것은?

```
// 입력: 가중 방향 그래프 G, 인접 행렬 A, 시작 정점 s
// 출력: s로부터 다른 모든 정점까지의 최단 경로의 길이
Dijkstra(G, A, s) {
    U = {s};
    for(모든 정점 v ∈ G)
        D[v] = A[s][v];
    while(U != G) {
        D[w]가 최소인 정점 w ∈ (G - U)를 선택
        U = U ∪ {w};
        for(w에 인접한 모든 정점 v) {
            D[v] = [ (가) ];
        }
    }
}
```

① min(D[v], D[v] + A[v][w])

② min(D[v], D[w] + A[v][w])

③ min(D[v], D[v] + A[w][v])

④ min(D[v], D[w] + A[w][v])

정답 4번

출제 포인트

이 문제는 다익스트라 알고리즘에서 새로 확정한 정점 w를 이용해 인접 정점 v의 최단거리 추정값을 완화하는 갱신식을 묻는 문제이다.

올바른 순서

1. 시작 정점 s를 확정 집합 U에 넣고 $D[v] = A[s][v]$로 초기화한다.
2. U에 속하지 않은 정점 중 $D[w]$가 최소인 정점 w를 고른다.
3. 선택한 w를 U에 추가해 s에서 w까지의 최단거리를 확정한다.
4. w에 인접한 각 정점 v에 대해 기존 거리 $D[v]$와 w를 거쳐 가는 거리 $D[w] + A[w][v]$를 비교해 더 작은 값으로 갱신한다.

왜 이 순서: $D[w]$가 먼저 확정되어야만 w를 경유한 새 경로 길이 $D[w] + A[w][v]$를 신뢰하고 비교할 수 있다.

NoteBase

선지별 해설

① X: 틀린 부분은 $D[v] + A[v][w]$이다. 기존 추정거리 $D[v]$에 다시 간선 비용을 더하는 식은 w를 경유하는 새 경로를 계산하지 못한다. 올바르게는 $D[w] + A[w][v]$와 비교해야 한다.
② X: 틀린 부분은 $A[v][w]$이다. 다익스트라는 방금 확정한 정점 w에서 인접 정점 v로 가는 간선 비용을 써야 하므로 방향이 $w \to v$여야 한다. 올바르게는 $A[w][v]$이다.
③ X: 틀린 부분은 $D[v] + A[w][v]$이다. 출발점에서 w까지의 확정거리 $D[w]$를 사용해야 하는데, 아직 갱신 대상인 $D[v]$를 더하면 완화식이 아니다. 올바르게는 $D[w] + A[w][v]$이다.
④ O: 기존 값 $D[v]$와 w를 거쳐 가는 새 경로 길이 $D[w] + A[w][v]$ 중 작은 값을 택하는 완화식이므로 옳다. 다익스트라의 핵심 갱신 단계와 정확히 일치한다.

오답 포인트

가장 흔한 함정은 거리 배열 첨자와 간선 방향을 동시에 헷갈리는 점이다. $D[v]$는 갱신 대상이고, $D[w]$는 이미 최소값이 확정된 기준점이다. 또한 인접 행렬 항도 $A[v][w]$가 아니라 $A[w][v]$여야 한다.

다시 나오면 이렇게 판단

다익스트라 의사코드에서 새로 선택된 최소 정점이 보이면 먼저 그 정점을 w로 잡고, 다음 줄은 항상 기존값과 w를 경유한 새 경로 D[w] + 간선(w,v)를 비교하는 완화식인지 확인하면 된다.

암기 포인트

다익스트라 갱신은 기존 $D[v]$와 경유 $D[w] + A[w][v]$의 최소값 비교이다.

NoteBase

문 17. KMP(Knuth-Morris-Pratt) 알고리즘과 보이어-무어(Boyer-Moore) 알고리즘에 대한 설명으로 옳지 않은 것은?

① KMP 알고리즘은 패턴 문자열의 첫 문자부터 순차적으로 대상 문자열과 부분 문자열의 일치 여부를 판단한다.

② KMP 알고리즘은 패턴 문자열과 대상 문자열을 한 차례 비교한 후 얻을 수 있는 추가 정보로 비교 횟수를 감소시킨다.

③ 보이어-무어 알고리즘은 패턴 문자열의 마지막 문자부터 역순으로 대상 문자열과 부분 문자열의 일치 여부를 판단한다.

④ 보이어-무어 알고리즘은 전처리(preprocessing) 과정 없이 곧바로 일치 여부를 판단할 수 있다.

정답 4번

출제 포인트

이 문제는 KMP와 Boyer-Moore의 비교 방향과 전처리 사용 여부를 구분하는지를 묻는 문제이다.

핵심 개념

KMP는 패턴의 접두사와 접미사가 겹치는 정보를 전처리해 두고, 패턴의 앞쪽부터 비교하면서 이미 확인한 문자를 다시 많이 보지 않도록 하는 알고리즘이다.
Boyer-Moore는 패턴의 뒤쪽부터 비교하고, 불일치가 나면 나쁜 문자 규칙과 좋은 접미사 규칙에 따라 패턴을 크게 건너뛰는 알고리즘이다.
두 알고리즘 모두 전처리를 사용하지만, KMP는 실패 함수 계열 정보, Boyer-Moore는 bad character와 good suffix 계열 정보를 사용한다.

선지별 해설

① O: KMP는 패턴의 첫 문자부터 대상 문자열과 순차적으로 비교를 시작한다. 불일치가 생기면 실패 함수 정보를 이용해 다음 비교 위치를 정하므로, 진술의 방향 설명이 맞다.
② O: KMP는 한 번의 비교 과정에서 얻은 접두사·접미사 일치 정보를 활용해 불필요한 재비교를 줄인다. 이미 맞았던 부분을 다시 처음부터 대조하지 않는다는 점이 핵심이다.
③ O: Boyer-Moore는 패턴의 마지막 문자부터 역순으로 비교한다. 뒤쪽에서 불일치를 빨리 찾으면 패턴을 여러 칸 이동할 수 있으므로 진술이 옳다.
④ X: 틀린 부분은 "전처리(preprocessing) 과정 없이"이다. Boyer-Moore는 패턴에 대해 나쁜 문자 표와 좋은 접미사 정보를 미리 준비한 뒤 비교를 수행한다. 올바르게는 "보이어-무어 알고리즘은 전처리 후 패턴 문자열의 마지막 문자부터 역순으로 비교한다"이다.

NoteBase

오답 포인트

KMP와 Boyer-Moore를 모두 문자열 검색 알고리즘으로만 외우면 둘 다 단순 순차 비교라고 착각하기 쉽다. 특히 ④는 Boyer-Moore가 뒤에서 비교한다는 특징만 기억한 수험생에게 전처리 유무를 흐리게 만드는 함정이다.

다시 나오면 이렇게 판단

문자열 검색 알고리즘 비교 문제는 먼저 비교 시작 위치가 앞인지 뒤인지, 다음으로 어떤 전처리 정보를 저장하는지, 마지막으로 불일치 뒤 얼마나 건너뛰는지를 구분하면 정답을 빠르게 판단할 수 있다.

암기 포인트

KMP는 앞에서 비교하며 실패 함수 정보를 쓰고, Boyer-Moore는 뒤에서 비교하며 나쁜 문자·좋은 접미사 전처리를 쓴다.

NoteBase

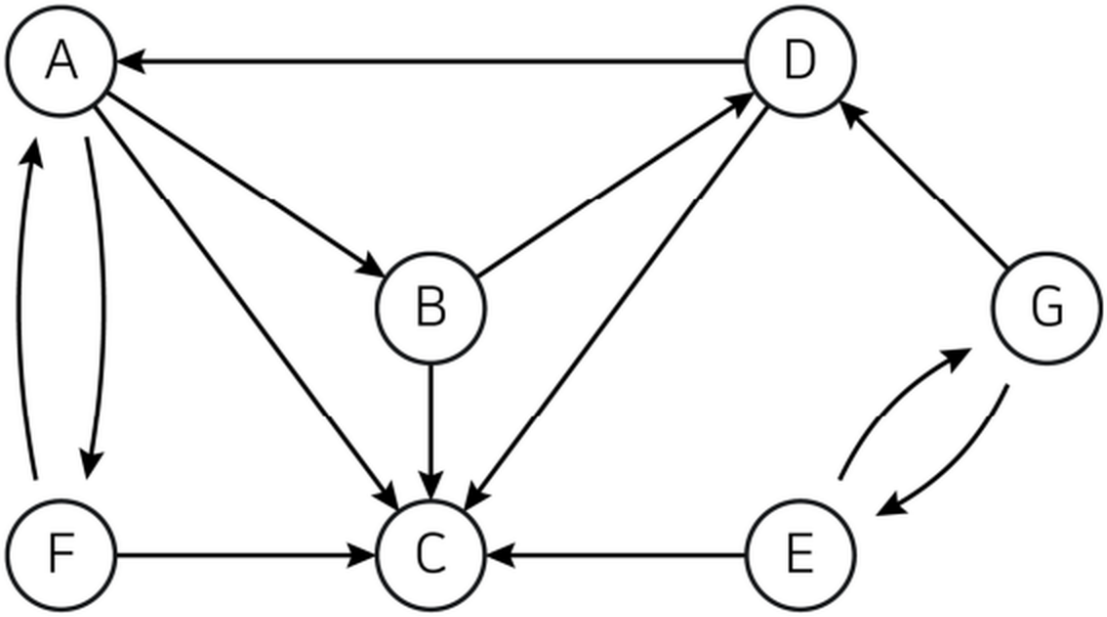
문 18. 다음 조건에 따라 주어진 그래프에 대해 깊이 우선 탐색(DFS, Depth First Search) 방법을 이용하여 트리를 만들 경우, 나올 수 있는 트리의 개수와 수행하는 백트래킹 횟수를 바르게 연결한 것은?

○ DFS의 시작 정점은 A로 하고, 정점의 접근 순서는 알파벳 순서로 한다.

○ DFS는 사이클을 형성하지 않으면서 백트래킹을 통해 시작 정점으로 돌아왔을 때, 시작 정점에서 방문하지 않은 접근 가능한 인접 정점이 없으면 종료한다.

○ DFS의 종료시점에 그래프에 방문하지 않은 정점들이 존재하면, 방문하지 않은 정점들 중 알파벳 순서가 가장 앞선 정점을 시작 정점으로 하여 DFS를 다시 시작한다.

○ 그래프 안에 방문하지 않은 정점이 존재하지 않을 때까지 DFS를 반복한다.



① 트리의 개수: 1, 백트래킹 횟수: 3

② 트리의 개수: 1, 백트래킹 횟수: 4

③ 트리의 개수: 2, 백트래킹 횟수: 3

④ 트리의 개수: 2, 백트래킹 횟수: 5

정답 4번

출제 포인트

이 문제는 방향 그래프에서 시작 정점 A, 알파벳순 인접 정점 선택, 미방문 정점이 남으면 가장 앞선 정점에서 DFS 재시작이라는 조건으로 DFS 포리스트와 백트래킹 횟수를 정확히 세는지를 묻는다.

NoteBase

선지별 해설

① X: 트리의 개수를 1개로 본 점이 틀리다. A에서 시작한 DFS로는 E와 G에 도달할 수 없으므로 첫 탐색 종료 뒤 E에서 DFS를 다시 시작해야 한다. 따라서 트리는 1개가 아니라 2개이다. 백트래킹 3회도 C→B, D→B, B→A, F→A, G→E의 5회를 반영하지 못한다.
② X: 백트래킹 횟수를 4회로 본 점이 틀리다. 첫 번째 트리에서 C에서 B로, D에서 B로, B에서 A로, F에서 A로 총 4회 백트래킹이 발생하고, 두 번째 트리에서 G에서 E로 1회가 더 발생한다. 따라서 4회가 아니라 5회이다.
③ X: 트리의 개수 2개는 맞지만 백트래킹 횟수 3회가 틀리다. DFS 트리 간선은 A-B, B-C, B-D, A-F, E-G로 잡히므로 되돌아오는 횟수는 각 말단 탐색 종료 때마다 총 5회 발생한다. 따라서 백트래킹 횟수는 3회가 아니라 5회이다.
④ O: A에서 시작하면 첫 번째 DFS 트리는 A, B, C, D, F를 포함하고, 미방문 정점이 남으므로 E에서 두 번째 DFS를 시작하여 E, G를 방문한다. 따라서 트리의 개수는 2개이다. 백트래킹은 C→B, D→B, B→A, F→A, G→E로 총 5회이므로 제시값이 맞다.

오답 포인트

첫 DFS가 끝났다고 해서 그래프 전체 탐색이 끝난 것으로 처리하면 트리 개수를 1개로 잘못 센다. 또 방문 스킵과 백트래킹을 혼동하면 A에서 C를 건너뛰는 장면을 백트래킹으로 오인하거나, 두 번째 트리의 G→E 복귀 1회를 빼먹기 쉽다.

다시 나오면 이렇게 판단

방향 그래프 DFS 문제는 시작 정점에서 실제로 도달 가능한 정점만 먼저 묶고, 막다른 정점에서 부모로 되돌아가는 횟수만 백트래킹으로 센다. 한 트리가 끝난 뒤 미방문 정점이 남으면 그때 DFS 포리스트의 트리 수를 1개 추가한다.

암기 포인트

DFS 포리스트의 트리 수는 재시작 횟수를 포함한 루트 수이고, 백트래킹 횟수는 DFS 트리에서 자식에서 부모로 실제 되돌아온 횟수이다.

NoteBase

문 19. 다음은 n개의 데이터에 대한 평균을 재귀적으로 계산하는 프로그램이다. (가) ~ (다)에 들어갈 내용을 바르게 연결한 것은? (단, n ≥ 1이다)

```
float AVG(float (가), int n) {
  if(n == 1) return (나);
    else
      return (data[n-1] + (n-1) * AVG(data, n-1)) / (다);
}
```

① (가): data[], (나): 0, (다): n

② (가): data[], (나): 1, (다): (n - 1)

③ (가): *data, (나): data[0], (다): n

④ (가): *data, (나): data[n - 1], (다): (n - 1)

정답 3번

출제 포인트

이 문제는 재귀적으로 평균을 갱신하는 식과 배열 매개변수의 의미를 연결할 수 있는지를 묻는다.

핵심 개념

함수 매개변수에서 `float data[]`와 `float *data`는 배열의 시작 주소를 받는 같은 의미이다.
원소가 1개일 때 평균은 그 원소 자신이므로 기저 조건의 반환값은 `data[0]`이어야 한다.
재귀식은 앞의 `n-1`개 평균에 마지막 원소 `data[n-1]`를 합쳐 전체 `n`개 평균으로 확장하므로
$AVG(data, n) = \frac{data[n-1] + (n-1) \times AVG(data, n-1)}{n}$가 된다.

NoteBase

선지별 해설

① X: `(가)=data[]`는 함수 매개변수로는 가능하지만 `(나)=0`이 틀리다. `n=1`이면 평균은 유일한 원소값이므로 0을 반환할 수 없다. 틀린 표현은 `0`이고, 올바르게는 `data[0]` 또는 `*data`이다.
② X: `(가)=data[]`는 가능하지만 `(나)=1`과 `(다)=n-1`이 틀리다. 원소 하나의 평균은 1이 아니라 `data[0]`이고, 앞의 `n-1`개 평균에 마지막 원소를 더해 `n`개 평균을 만들 때 분모는 `n`이어야 한다. 틀린 표현은 `1`, `n-1`이고, 올바르게는 `data[0]`, `n`이다.
③ O: `(가)=*data`는 `float *data`를 뜻하므로 배열 시작 주소를 받는 선언으로 맞다. `(나)=data[0]`는 `n=1`일 때 평균값과 같고, `(다)=n`을 넣으면 앞의 `n-1`개 평균과 `data[n-1]`를 합쳐 정확히 `n`개 평균이 계산된다.
④ X: `(가)=*data`는 가능하고 `n=1`에서 `(나)=data[n-1]`는 `data[0]`와 같으므로 기저 조건으로도 성립한다. 그러나 `(다)=n-1`이 틀리다. 전체 `n`개 평균을 구하는 식이므로 분모는 `n`이어야 한다. 틀린 표현은 `n-1`이고, 올바르게는 `n`이다.

오답 포인트

배열 매개변수 표기 `data[]`와 포인터 표기 `*data`의 차이에만 시선이 묶이면 기저 조건과 분모를 놓치기 쉽다. 이 문항의 함정은 `(가)`보다 `n=1`일 때 반환값과 평균 갱신식의 분모가 무엇인지에 있다.

다시 나오면 이렇게 판단

재귀 함수는 먼저 기저 조건이 문제의 의미와 맞는지 확인하고, 그다음 `n-1`개 결과를 이용해 `n`개 결과로 확장하는 식의 분자와 분모가 정확한지 확인하면 된다.

암기 포인트

재귀 평균은 `원소 1개면 그 값`, `n개 평균이면 분모는 n`이라는 두 줄로 정리된다.

NoteBase

문 20. 작업 선택 문제를 다루는 다음 알고리즘은 각 작업의 시작 시간 s와 완료 시간 f가 정해진 n개의 작업 t[0..n-1]가 주어질 때, 하나의 기계만 사용하여 충돌 없이 수행할 수 있는 최대 개수의 작업을 선택한다. 이 알고리즘의 (가), (나)에 들어갈 내용을 바르게 연결한 것은? (단, n ≥ 2인 정수, count는 선택한 작업의 개수, t[i].s는 작업 t[i]의 시작 시간, t[i].f는 작업 t[i]의 종료 시간이고, jobs 배열은 선택된 작업을 저장한다)

```
ActSelect(n, t[]) {
    (가) 의 오름차순으로 작업 t[]를 정렬;
    count = 0, last = 0, jobs[];
    jobs[count] = 0;
    for(i = 0; i < n; i++)
        if( (나) ) {
            count++;
            last = i;
            jobs[count] = last;
        }
    return jobs;
}
```

① (가) 시작 시간, (나) t[i].s >= t[last].f

② (가) 시작 시간, (나) t[i].f >= t[last].s

③ (가) 완료 시간, (나) t[i].s >= t[last].f

④ (가) 완료 시간, (나) t[i].f >= t[last].s

정답 3번

출제 포인트

활동 선택 문제에서 최대 개수의 작업을 고르는 탐욕 기준이 무엇인지 묻는 문제이다.

핵심 개념

활동 선택 문제의 핵심은 서로 겹치지 않는 작업을 많이 고르기 위해 가장 빨리 끝나는 작업부터 선택하는 데 있다. 따라서 정렬 기준은 시작 시간이 아니라 완료 시간이며, 다음 작업을 고를 조건은 현재 작업의 시작 시간이 직전에 선택한 작업의 완료 시간 이상인 것이다.

NoteBase

선지별 해설

① X: 틀린 부분은 (가)를 시작 시간으로 둔 점이다. 활동 선택의 탐욕 선택 기준은 가장 빨리 끝나는 작업을 먼저 고르는 것이므로 시작 시간 오름차순 정렬로는 최적해를 보장하지 못한다. 조건식 $t[i].s \geq t[last].f$ 자체는 맞지만 정렬 기준이 틀렸다. 올바르게는 완료 시간 오름차순으로 정렬해야 한다.

② X: 틀린 부분은 두 군데이다. 먼저 정렬 기준을 시작 시간으로 둔 것이 틀렸고, 다음 작업 선택 조건도 잘못되었다. $t[i].f \geq t[last].s$는 현재 작업의 종료 시각과 이전 작업의 시작 시각을 비교하므로 충돌 여부를 제대로 판정하지 못한다. 올바르게는 완료 시간 오름차순 정렬 후 $t[i].s \geq t[last].f$를 확인해야 한다.

③ O: 활동 선택 문제의 표준 탐욕 알고리즘은 완료 시간이 가장 이른 작업부터 본다. 그래서 (가)는 완료 시간이고, 이미 선택한 마지막 작업이 끝난 뒤에 시작하는 작업만 추가할 수 있으므로 (나)는 $t[i].s \geq t[last].f$가 된다.

④ X: 틀린 부분은 조건식 $t[i].f \geq t[last].s$이다. 이 식은 거의 항상 성립할 수 있어 작업이 실제로 겹치는지 걸러내지 못한다. 올바른 비교는 현재 작업의 시작 시각과 마지막 선택 작업의 완료 시각을 비교하는 $t[i].s \geq t[last].f$이다.

오답 포인트

시작 시간과 완료 시간을 대칭적으로 보아도 된다고 착각하면 ①이나 ④로 흔들린다. 이 문제의 함정은 정렬 기준과 충돌 판정 기준이 서로 다른 역할을 한다는 점이다. 정렬은 가장 빨리 끝나는 작업을 찾기 위한 것이고, 판정은 다음 작업의 시작 시각이 이전 작업의 완료 시각 뒤인지 확인하는 것이다.

다시 나오면 이렇게 판단

작업 선택 문제에서 '최대 개수'와 '충돌 없음'이 함께 나오면 먼저 완료 시간 오름차순 탐욕을 떠올린다. 그다음 새 작업을 넣는 조건은 항상 '새 작업 시작 시각 >= 마지막 선택 작업 완료 시각' 형태인지 확인하면 된다.

암기 포인트

활동 선택은 빨리 끝나는 것부터 고르고, 다음 것은 시작 시간이 직전 종료 시간 뒤에 있어야 한다.

NoteBase

문 21. 동적 프로그래밍(dynamic programming) 방법으로 최소의 원소단위 곱셈 횟수를 갖도록 연쇄 행렬곱셈(matrix-chain multiplication)의 순서를 정하려 한다. $c_{i,j}$를 행렬 M_i부터 M_j까지의 연쇄 행렬곱셈의 최소 비용이라고 할 때, $c_{i,j}$에 대한 관계식으로 옳은 것은? (단, $i < j$이고, 각 행렬의 차원이 $p_0, p_1, \cdots, p_n$일 때, 행렬 M_i의 크기는 $p_{i-1} \times p_i$이다)

① c_i,j=cases0 & if i=j \ _i ≤ k ≤ j-1\c_i,k+c_k+1,j+p_i-1p_kp_j\ & if i<jcases

② c_i,j=cases0 & if i=j \ _i ≤ k ≤ j-1\c_i,k+c_k,j+p_i-1p_k-1p_j\ & if i<jcases

③ c_i,j=cases0 & if i=j \ _i ≤ k ≤ j-1\c_i,k-1+c_k,j+p_i-1p_k-1p_j\ & if i<jcases

④ c_i,j=cases0 & if i=j \ _i ≤ k ≤ j-1\c_i,k+c_k,j+p_i-1p_kp_j\ & if i<jcases

정답 1번

출제 포인트

연쇄 행렬곱셈의 동적 계획법 점화식에서 분할 위치 k, 좌우 부분문제의 범위, 마지막 곱셈 비용 $p_{i-1}p_kp_j$를 정확히 잡는지를 묻는 문제이다.

조건 정리

주어진 값: $c_{i,j}$는 M_i부터 M_j까지의 최소 곱셈 비용이다.
구해야 할 값: $c_{i,j}$의 올바른 점화식이다.
주의할 조건: k에서 분할하면 왼쪽은 $M_i \sim M_k$, 오른쪽은 $M_{k+1} \sim M_j$가 되고, 두 결과 행렬의 크기는 각각 $p_{i-1} \times p_k$, $p_k \times p_j$이다.

사용할 공식

$i = j$이면 $c_{i,j} = 0$이다. $i < j$이면 $c_{i,j} = \min_{i \le k \le j-1} \{c_{i,k} + c_{k+1,j} + p_{i-1}p_kp_j\}$이다.
이 공식을 쓰는 이유는 연쇄 행렬곱셈이 분할 정복 구조를 가지기 때문이다. k에서 한 번 끊으면 왼쪽 최소 비용과 오른쪽 최소 비용을 각각 더하고, 마지막에 두 결과 행렬을 곱하는 비용만 추가하면 전체 최소 비용을 구할 수 있다.

계산 과정

1. $M_i \sim M_j$를 k에서 분할하면 부분문제는 $M_i \sim M_k$와 $M_{k+1} \sim M_j$이다.
2. 각 부분의 최소 비용은 각각 $c_{i,k}$, $c_{k+1,j}$이다.
3. 두 부분 결과의 크기는 $p_{i-1} \times p_k$, $p_k \times p_j$이므로 마지막 곱셈 비용은 $p_{i-1}p_kp_j$이다. 따라서 이 세 항을 더한 값의 최솟값을 취하는 ①이 맞다.

NoteBase

선지별 해설

① O: k에서 분할했을 때 왼쪽 비용 $c_{i,k}$, 오른쪽 비용 $c_{k+1,j}$, 마지막 곱셈 비용 $p_{i-1}p_kp_j$를 모두 정확히 반영한 점화식이다.
② X: 오른쪽 부분문제가 $c_{k,j}$로 되어 있어 M_k가 양쪽 부분문제에 겹친다. 곱셈 비용의 공통 차원도 p_k여야 하는데 p_{k-1}로 적었다. 틀린 표현은 $c_{k,j}$와 p_{k-1}이고, 올바르게는 $c_{k+1,j}$와 p_k이다.
③ X: 왼쪽 부분문제를 $c_{i,k-1}$, 오른쪽 부분문제를 $c_{k,j}$로 잡아 분할 기준이 한 칸씩 밀려 있다. 틀린 표현은 $c_{i,k-1}+c_{k,j}+p_{i-1}p_{k-1}p_j$이고, 올바르게는 $c_{i,k}+c_{k+1,j}+p_{i-1}p_kp_j$이다.
④ X: 마지막 곱셈 비용 $p_{i-1}p_kp_j$는 맞지만 오른쪽 부분문제가 $c_{k,j}$라서 M_k를 중복 포함한다. 틀린 표현은 $c_{k,j}$이고, 올바르게는 $c_{k+1,j}$이다.

오답 포인트

분할점 k를 잡은 뒤 오른쪽 구간이 $k+1$에서 시작한다는 점과, 마지막 곱셈의 공통 차원이 p_k라는 점을 헷갈리기 쉽다. ②와 ④는 부분문제 범위가 겹치는 함정이고, ③은 인덱스를 한 칸씩 밀어 놓은 함정이다.

다시 나오면 이렇게 판단

연쇄 행렬곱셈 점화식은 먼저 왼쪽 구간 $i\sim k$와 오른쪽 구간 $k+1\sim j$가 겹치지 않는지 확인하고, 마지막 곱셈을 $(p_{i-1}\times p_k)(p_k\times p_j)$로 적어 공통 차원이 p_k인지 점검하면 된다.

암기 포인트

행렬곱 연쇄 DP는 `왼쪽 $i\sim k$ + 오른쪽 $k+1\sim j$ + 마지막 비용 $p_{i-1}p_kp_j$`의 형태이다.

NoteBase

문 22. 동적 프로그래밍은 한 번 계산된 값을 메모리에 저장한 후, 저장된 값을 필요할 때마다 가져다 쓰는 방법이다. 동적 프로그래밍을 적용하여 구현한 다음 Java 프로그램에서 피보나치 수열 fibo(6)을 구하기 위해 저장되어 있는 fibo(3)의 결과를 가져다 쓰는 횟수는?

```
public class Fibonacci {
   static int[] dp = new int[100];

   public static void main(String[] args) {
      System.out.println(fibo(6));
   }

   public static int fibo(int num) {
      if(num == 0) {
         return 0;
      }
      if(num == 1) {
         return 1;
      }
      if(dp[num] != 0) {
         return dp[num];
      }
      return dp[num] = fibo(num - 1) + fibo(num - 2);
   }
}
```

① 0

② 1

③ 2

④ 3

정답 2번

출제 포인트

이 문제는 메모이제이션이 실제 호출 과정에서 언제 저장되고 언제 재사용되는지를 따라가며 fibo(3)의 메모 히트 횟수를 세는 문제이다.

조건 정리

주어진 값은 fibo(6)을 호출하는 재귀 코드와 dp 배열이다.
구해야 할 값은 이미 저장된 fibo(3) 값을 다시 읽어 오는 횟수이다.
주의할 조건은 fibo(3)을 처음 계산하는 호출은 재사용이 아니라 저장 과정이고, 재사용은 `if (dp[num] != 0)` 분기로 바로 반환될 때만 센다는 점이다.

NoteBase

사용할 원리

세어야 하는 횟수는 dp[3]이 저장된 뒤 `return dp[num];`로 끝나는 fibo(3) 호출 수이다.
`return dp[num] = fibo(num - 1) + fibo(num - 2);`에서 왼쪽 항 fibo(num - 1)이 먼저 끝까지 재귀로 내려간 뒤 오른쪽 항이 계산되므로 호출 트리는 왼쪽 자식부터 깊이 우선으로 탐색된다.
핵심 원리는 어떤 항이 이미 dp에 저장돼 있으면 그 호출은 `dp[num] != 0`에서 즉시 반환되어 그 아래 하위 항으로는 더 이상 내려가지 않는다는 점이다. 따라서 상위 항이 한 번 저장되면 그 안에 들어 있는 하위 항(예: fibo(3))의 재사용 횟수는 더 늘어나지 않는다.

계산 과정

호출 트리를 왼쪽 항부터 깊이 우선으로 들여쓰면 다음과 같다.
fibo(6) = fibo(5) + fibo(4)
fibo(5) = fibo(4) + fibo(3)
fibo(4) = fibo(3) + fibo(2) ← 여기서 fibo(3)이 처음 계산되어 dp[3]에 저장됨(저장이지 재사용 아님)
fibo(3) = fibo(2) + fibo(1)
fibo(2) = fibo(1) + fibo(0) ← dp[2] 저장
→ dp[3] 저장 후 dp[4] 저장
fibo(3) → dp[3] != 0 이므로 저장된 값 즉시 반환 ← 재사용 1회
→ dp[5] 저장
fibo(4) → dp[4] != 0 이므로 저장된 값 즉시 반환(이미 dp[4]에 있으므로 fibo(3)까지 내려가지 않음)
정리하면 fibo(3)은 fibo(4) 내부에서 단 한 번 계산되어 dp[3]에 2가 저장되고, 그 뒤 fibo(5)의 두 번째 항 fibo(3) 호출에서 dp[3]을 한 번 가져다 쓴다. 마지막으로 fibo(6)의 두 번째 항 fibo(4)는 이미 dp[4]에 저장돼 있어 그 자리에서 끊기므로 fibo(3)까지 다시 내려가지 않는다.
따라서 저장된 fibo(3) 결과를 가져다 쓰는 횟수는 1회이다.

선지별 해설

① X. 0회가 되려면 fibo(3)이 계산만 되고 한 번도 메모에서 재사용되지 않아야 한다. 실제로는 fibo(5)를 계산할 때 두 번째 fibo(3) 호출이 dp[3]을 바로 반환한다. 틀린 표현은 '0회'이고, 올바르게는 '1회'이다.
② O. fibo(3)은 처음 한 번 계산되어 저장되고, 그 뒤 fibo(5)의 나머지 항을 구할 때 정확히 한 번 메모에서 다시 읽힌다. 이후 fibo(6)에서 필요한 fibo(4) 자체가 이미 저장되어 있으므로 dp[3] 재사용은 더 늘어나지 않는다.
③ X. 2회가 되려면 저장된 dp[3]을 읽는 별도의 fibo(3) 호출이 두 번 있어야 한다. 실제 호출 흐름에서는 메모 히트가 fibo(5) 안에서 한 번만 발생한다. 틀린 표현은 '2회'이고, 올바르게는 '1회'이다.
④ X. 3회가 되려면 fibo(6) 계산의 여러 단계에서 fibo(3)까지 반복해서 다시 내려가야 한다. 하지만 fibo(4)와 fibo(5)도 차례로 저장되므로 이후 호출은 상위 값에서 끊긴다. 틀린 표현은 '3회'이고, 올바르게는 '1회'이다.

NoteBase

오답 포인트

처음 fibo(3)을 계산하는 장면까지 재사용 횟수에 포함하면 2회처럼 보이기 쉽다. 또 fibo(6)에서 마지막에 호출하는 fibo(4)가 내부에서 다시 fibo(3)을 부를 것이라고 생각하면 횟수를 과대계산하게 된다. fibo(4)는 이미 dp[4]에 저장돼 있어 즉시 반환되므로 그 아래 fibo(3)으로는 진입하지 않는다는 점을 놓치지 않아야 한다.

다시 나오면 이렇게 판단

메모이제이션 횟수 문제는 호출 트리를 끝까지 그리기보다 '처음 계산되어 저장되는 시점'과 '`dp[num] != 0`으로 바로 반환되는 시점'을 구분해서 센다. 상위 항이 이미 저장되면 그 아래 하위 항 재사용 횟수는 더 늘어나지 않는다고 판단하면 된다.

암기 포인트

동적 프로그래밍에서 처음 계산은 저장이고, 같은 값의 다음 호출부터가 재사용이다.

NoteBase

문 23. 배낭 문제(knapsack problem)에 대한 설명으로 옳은 것은?

① 0-1 배낭 문제와 분할 가능 배낭 문제는 다항식 시간(polynomial-time) 문제이다.

② 분할 가능 배낭 문제는 최적 부분 구조를 갖지만, 0-1 배낭 문제는 그렇지 않다.

③ 0-1 배낭 문제는 물건을 나눌 수 없어서 그리디(greedy) 알고리즘을 적용하여 해결할 수 있으며, 분할 가능 배낭 문제는 물건을 나눌 수 있어 동적 프로그래밍으로 해결해야 한다.

④ 분할 가능 배낭 문제는 단위 무게당 가치가 가장 높은 물건을 선택하는 방식이 항상 최적해를 보장하지만, 0-1 배낭 문제는 그렇지 않다.

정답 4번

출제 포인트

이 문제는 0-1 배낭 문제와 분할 가능 배낭 문제의 해결 가능성, 최적 부분 구조, 그리디 적용 가능성을 비교하는 문제이다.

핵심 개념

0-1 배낭 문제는 각 물건을 통째로 넣거나 빼야 하는 최적화 문제이며, 단위 가치순 그리디만으로는 항상 최적해가 보장되지 않는다. 분할 가능 배낭 문제는 물건을 쪼개 담을 수 있는 최적화 문제이며, 단위 무게당 가치가 큰 순서대로 선택하는 그리디가 항상 최적이다. 두 문제 모두 최적 부분 구조를 가지지만, 0-1 배낭 문제는 일반적으로 다항식 시간 문제로 보지 않고 동적 프로그래밍은 의사다항 시간 알고리즘이다.

선지별 해설

① X: 틀린 부분은 "0-1 배낭 문제와 분할 가능 배낭 문제는 다항식 시간 문제"라는 표현이다. 분할 가능 배낭 문제는 정렬 기반 그리디로 다항식 시간에 풀리지만, 0-1 배낭 문제는 일반적으로 NP-난해 문제로 알려져 있어 입력 크기에 대한 다항식 시간 알고리즘이 알려져 있지 않다.
② X: 틀린 부분은 "0-1 배낭 문제는 최적 부분 구조를 갖지 않는다"라는 표현이다. 0-1 배낭 문제도 남은 용량에 대한 부분문제로 나눌 수 있으므로 최적 부분 구조를 가진다. 올바른 표현은 "분할 가능 배낭 문제와 0-1 배낭 문제 모두 최적 부분 구조를 가진다"이다.
③ X: 틀린 부분은 "0-1 배낭 문제는 그리디 알고리즘으로 해결할 수 있으며, 분할 가능 배낭 문제는 동적 프로그래밍으로 해결해야 한다"라는 표현이다. 실제로는 반대이다. 분할 가능 배낭 문제는 단위 무게당 가치 기준 그리디가 최적해를 보장하고, 0-1 배낭 문제는 그리디만으로는 일반적으로 최적해를 보장하지 않아 동적 프로그래밍 등의 방법을 사용한다.
④ O: 분할 가능 배낭 문제는 물건을 나눌 수 있으므로 단위 무게당 가치가 가장 큰 것부터 채우는 그리디 선택이 항상 최적이다. 반면 0-1 배낭 문제는 물건을 쪼갤 수 없어 단위 가치가 높은 순서대로만 선택하면 전체 조합 최적해를 놓칠 수 있다.

NoteBase

오답 포인트

출제자는 "물건을 나눌 수 있는가"와 "그리디가 항상 최적인가"를 뒤섞어 헷갈리게 만든다. 특히 ③처럼 0-1과 분할 가능 배낭의 해결 전략을 서로 바꾸어 쓰거나, ②처럼 0-1 배낭에는 최적 부분 구조가 없다고 착각하면 오답이 된다.

다시 나오면 이렇게 판단

배낭 문제가 나오면 먼저 물건을 분할할 수 있는지 본다. 분할 가능이면 단위 무게당 가치 기준 그리디를 떠올리고, 분할 불가능한 0-1 배낭이면 그리디 최적 보장이 없고 동적 프로그래밍 계열 문제인지 판단한다.

암기 포인트

분할 가능 배낭은 가치밀도 그리디, 0-1 배낭은 조합 최적화와 동적 프로그래밍이다.

NoteBase

문 24. 알고리즘들에 대한 설명으로 옳지 않은 것은?

① 크루스칼의 최소 신장 트리 알고리즘은 가중치가 가장 작으면서 사이클을 만들지 않는 간선을 추가하여 트리를 구성한다.

② 프림(Prim)의 최소 신장 트리 알고리즘은 현재까지 만들어진 트리에 최소 가중치로 연결되는 정점을 트리에 추가한다.

③ 허프만(Huffman) 압축에서 파일에 빈번히 나타나는 문자는 긴 이진코드를 할당하고, 드물게 나타나는 문자는 짧은 이진코드를 할당한다.

④ 벨만-포드(Bellman-Ford)의 최단 경로 알고리즘은 음의 가중치를 갖는 간선이 존재하는 상황에서도 단일 출발점 최단 경로 문제를 해결하는 알고리즘이다.

정답 3번

출제 포인트

이 문제는 크루스칼, 프림, 허프만, 벨만-포드의 핵심 동작 원리를 정확히 구분하는지를 묻는 문제이다.

핵심 개념

크루스칼은 간선을 가중치 오름차순으로 보며 사이클이 생기지 않는 간선만 선택해 최소 신장 트리를 만드는 알고리즘이다.
프림은 현재 트리에 인접한 간선 중 최솟값을 택해 새로운 정점을 하나씩 붙여 최소 신장 트리를 확장하는 알고리즘이다.
허프만 압축은 자주 나오는 문자에 짧은 코드, 드물게 나오는 문자에 긴 코드를 주어 전체 비트 수를 줄이는 방식이다.
벨만-포드는 음의 가중치 간선이 있어도 단일 출발점 최단 경로를 구할 수 있는 알고리즘이다.

선지별 해설

① O: 크루스칼은 가장 가중치가 작은 간선부터 보되 사이클을 만드는 간선은 제외한다. 선지의 '가중치가 가장 작으면서 사이클을 만들지 않는 간선'이라는 설명이 정확하다.
② O: 프림은 현재까지 만들어진 트리와 연결되는 간선 중 최솟값을 선택해 새 정점을 추가한다. 선지의 '현재까지 만들어진 트리에 최소 가중치로 연결되는 정점'이라는 취지가 맞다.
③ X: 틀린 부분은 '빈번히 나타나는 문자는 긴 이진코드', '드물게 나타나는 문자는 짧은 이진코드'라는 표현이다. 실제로는 빈번히 나타나는 문자에 짧은 이진코드를, 드물게 나타나는 문자에 긴 이진코드를 할당한다.
④ O: 벨만-포드는 음의 가중치 간선이 있어도 단일 출발점 최단 경로를 구할 수 있다. 음의 사이클이 있으면 최단 경로가 확정되지 않는다는 추가 조건은 있지만, 선지의 진술 자체는 맞다.

NoteBase

오답 포인트

③은 허프만 부호의 길이 배정을 정반대로 바꿔 놓은 선지이다. 압축 알고리즘은 자주 나오는 문자에 더 적은 비트를 배정해야 전체 길이가 줄어드는데, 이를 반대로 읽으면 바로 오답이 된다.

다시 나오면 이렇게 판단

최소 신장 트리는 크루스칼이 '간선 기준', 프림이 '현재 트리 기준'으로 선택한다는 축으로 구분한다. 압축은 '자주 나오면 짧게', 최단 경로는 '음수 간선 허용이면 벨만-포드'라는 기준으로 판단한다.

암기 포인트

크루스칼은 간선 선택, 프림은 정점 확장, 허프만은 자주 나오면 짧게, 벨만-포드는 음수 간선 허용이다.

NoteBase

문 25. 다음 설명에 해당하는 알고리즘은?

◯ 주어진 문제에 대해 근사해나 일정 확률 이상의 최적해를 출력함
◯ 근사해나 간혹 틀린 해를 허용할 수 있는 경우에 사용 가능함
◯ 효율적인 결정론적 알고리즘이 밝혀져 있지 않은 문제에 효과적임

① 유클리드(Euclid) 알고리즘
② 스트라센(Strassen) 알고리즘
③ 몬테카를로(Monte Carlo) 알고리즘
④ 플로이드-워샬(Floyd-Warshall) 알고리즘

정답 3번

출제 포인트

이 문제는 몬테카를로 알고리즘의 핵심 특징인 확률적 수행, 오답 가능성 허용, 결정론적 효율 알고리즘 부재 상황에서의 활용을 묻는 문제이다.

핵심 개념

몬테카를로 알고리즘은 확률을 사용해 해를 구하며 일정 확률로 오답이 나올 수 있는 알고리즘이다. 판단 기준은 결과의 정확성을 항상 보장하는지 여부이다. 유클리드 알고리즘은 최대공약수를 정확히 구하는 결정론적 알고리즘이고, 스트라센 알고리즘은 행렬 곱셈을 더 빠르게 하지만 결과는 정확하며, 플로이드-워셜 알고리즘은 모든 정점 쌍 최단거리를 정확히 구하는 동적 계획법 알고리즘이다.

선지별 해설

① X. 틀린 부분은 이 설명을 유클리드 알고리즘에 대응시키는 점이다. 유클리드 알고리즘은 최대공약수를 나눗셈 반복으로 항상 정확하게 구하므로 근사해나 확률적 오답 허용과는 관련이 없다.
② X. 틀린 부분은 간혹 틀린 해를 허용하는 확률적 성격을 스트라센 알고리즘에 붙인 점이다. 스트라센 알고리즘은 분할정복으로 행렬 곱셈의 계산량을 줄이는 정확 알고리즘이지, 오답을 확률적으로 허용하는 알고리즘이 아니다.
③ O. 몬테카를로 알고리즘은 확률적 방법으로 해를 구하며 일정 확률로 오답이 나올 수 있다. 따라서 근사해나 드물게 틀린 답을 허용할 수 있고, 효율적인 결정론적 방법이 없는 문제에서 실용적으로 사용된다.
④ X. 틀린 부분은 이 설명을 플로이드-워셜 알고리즘에 대응시키는 점이다. 플로이드-워셜 알고리즘은 모든 정점 쌍 사이의 최단거리를 동적 계획법으로 정확히 계산하므로 확률적 근사 알고리즘이 아니다.

NoteBase

오답 포인트

빠른 알고리즘이라는 말만 보고 스트라센 알고리즘이나 플로이드-워셜 알고리즘을 고르면 틀리기 쉽다. 이 문항의 결정 단서는 속도가 아니라 일정 확률, 간혹 틀린 해 허용, 결정론적 알고리즘 부재라는 표현이다.

다시 나오면 이렇게 판단

선지나 설명에 확률, 오답 가능성, 근사해 허용이 함께 나오면 먼저 몬테카를로 계열인지 본다. 반대로 항상 정확한 값이나 경로, 최대공약수처럼 정확 해를 보장하면 결정론적 정확 알고리즘으로 판단한다.

암기 포인트

몬테카를로는 빠르지만 틀릴 수 있고, 유클리드·스트라센·플로이드-워셜은 결과 자체는 정확하다.

NoteBase