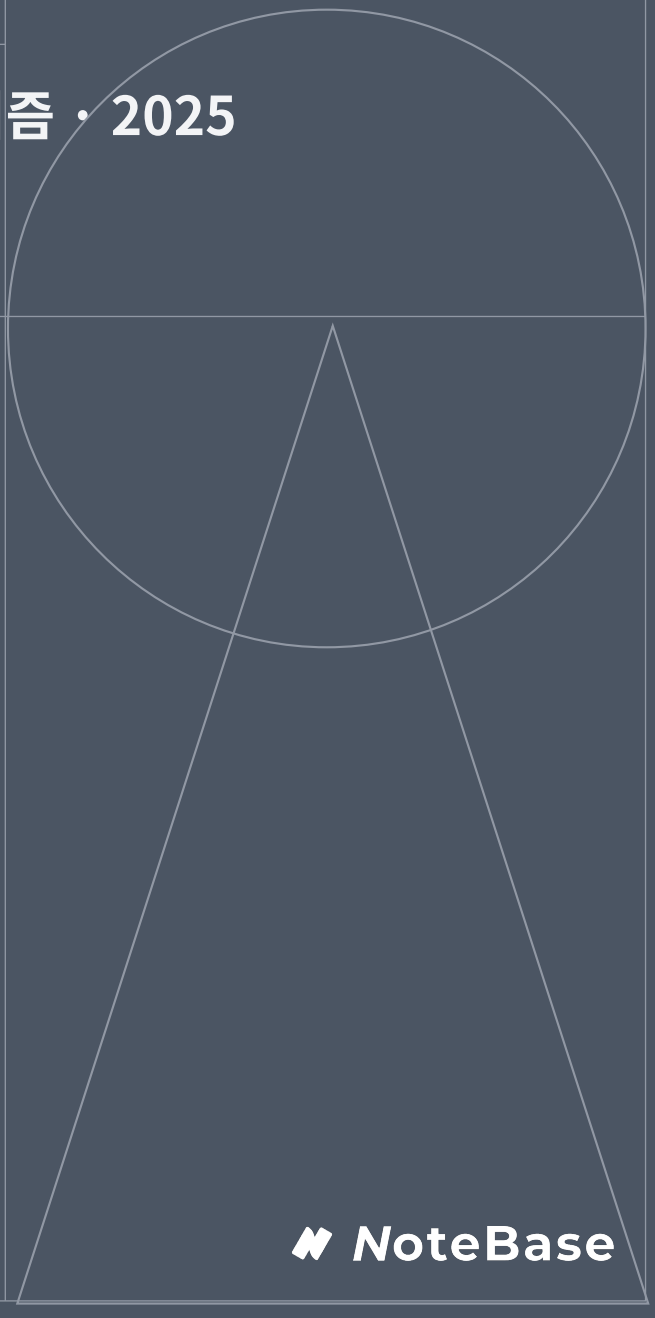
기출OX 해설집

공무원/국가직/9급 · 알고리즘 · 2025



NoteBase

기출OX 해설집

공무원/국가직/9급 · 알고리즘 · 2025

공무원/국가직/9급 · 알고리즘 · 2025

문 1. 다음 정수 열에서 연속 부분합의 최댓값은?

1, -3, 2, 4, -1, 5, -3, 2, 1, -2

① 5

② 8

③ 10

④ 15

정답 3번

간단한 해설

연속 부분합은 중간의 수를 빼지 않고 더해야 한다. 구간 2, 4, -1, 5의 합은 10이며, 각 위치에서 끝나는 연속 부분합의 최댓값을 비교해도 **전체 최댓값은 10**이다.

계산 과정

연속 부분합은 정수 열에서 서로 이웃한 수들을 하나 이상 골라 더한 값이다. 각 위치에서 끝나는 연속 부분합의 최댓값은 '현재 수부터 새로 시작한 합'과 '직전 위치의 최댓값에 현재 수를 더한 합' 중 큰 값이다. 직전까지의 최댓값이 **음수이면 버리고 현재 수부터 다시 시작**한다. 음수인 합을 앞에 붙이면 현재 수만 택할 때보다 합이 작아지기 때문이다.

첫 위치의 최댓값은 1이고, 두 번째 위치에서는 1 - 3 = -2이다. 세 번째 수 2에서는 앞의 -2를 버리고 새로 시작하므로 최댓값이 2가 된다. 이후 각 위치에서 끝나는 최댓값은 2 + 4 = 6, 6 - 1 = 5, 5 + 5 = 10, 10 - 3 = 7, 7 + 2 = 9, 9 + 1 = 10, 10 - 2 = 8이다. 따라서 각 위치에서 끝나는 최댓값은 순서대로 1, -2, 2, 6, 5, 10, 7, 9, 10, 8이다.

모든 연속 구간은 어느 한 위치에서 끝나므로, 위 값들 중 가장 큰 값이 전체 연속 부분합의 최댓값이다. 세 번째부터 여섯 번째 수까지 더하면 2 + 4 - 1 + 5 = 10이다. 일곱 번째부터 아홉 번째까지의 합은 -3 + 2 + 1 = 0이므로 아홉 번째까지 늘려도 합은 10이다. 최댓값 10은 ③의 값과 일치한다.

NoteBase

문 2. 이진 탐색(binary search) 알고리즘에 대한 설명으로 옳지 않은 것은?

① 탐색 범위가 절반씩 줄어든다.

② 탐색할 데이터는 순서 없이 저장되어 있다.

③ 시간 복잡도는 $O(\log n)$이다.

④ 1,000개의 데이터가 존재하는 경우, 최대 10회의 탐색으로 완료된다.

정답 2번

간단한 해설

이진 탐색은 데이터가 **정렬되어 있어야** 한다. 중앙값과 찾는 값을 비교했을 때 어느 쪽 범위를 제외할 수 있는지는 데이터의 정렬 순서를 근거로 판단하기 때문이다.

선지별 해설

① O: 중앙 원소와 찾는 값을 비교하고, 두 값이 다르면 찾는 값이 있을 수 있는 쪽만 남긴다. 오름차순이라면 찾는 값이 중앙값보다 작을 때 왼쪽을, 클 때 오른쪽을 탐색한다. 중앙 원소와 반대쪽 범위를 제외하므로 **남은 탐색 범위가 절반 이하로 줄어든다**.

② X: 탐색할 데이터는 순서 없이 저장된 것이 아니라 **오름차순이나 내림차순으로 정렬된 상태**여야 한다. 정렬되어 있지 않으면 중앙값보다 작은 값이 양쪽에 흩어져 있을 수 있으므로, 중앙값과의 비교만으로 한쪽 범위를 제외할 수 없다.

③ O: 데이터 수를 n이라 하면 탐색을 한 번 진행할 때마다 후보가 절반 이하로 줄어든다. 따라서 후보가 하나 정도 남을 때까지 필요한 반복 횟수는 $\log_2 n$에 비례한다. **중앙 원소에 바로 접근할 수 있는 배열**에서는 각 반복의 작업량이 일정하므로, 정렬 이후 이진 탐색 자체의 최악 시간 복잡도는 $O(\log n)$이다.

④ O: 중앙 원소를 한 번 확인하는 것을 탐색 1회로 센다. 데이터가 n개일 때 최대 탐색 횟수는 $\lfloor \log_2 n \rfloor + 1$이며, $\lfloor x \rfloor$는 x 이하의 가장 큰 정수를 뜻한다. 1,000개라면 $\lfloor \log_2 1{,}000 \rfloor + 1 = 10$이므로, **최대 10회의 중앙 원소 확인**으로 찾는 값의 존재 여부를 판단할 수 있다.

NoteBase

문 3. 동적 계획법(dynamic programming)으로 설계된 알고리즘의 동작 방식에 대한 설명으로 옳지 않은 것은?

① 큰 문제를 작은 문제로 나눌 수 있어야 한다.

② 작은 문제들이 반복해서 사용되며, 이 작은 문제들의 결괏값을 저장해 두고 활용한다.

③ 상향식(bottom-up) 접근 방식 또는 하향식(top-down) 접근 방식으로 구현할 수 있다.

④ 메모하기(memoization) 방법을 활용하여 연산과 탐색이 증가하고 실행속도가 저하된다.

정답 4번

간단한 해설

메모하기(memoization)는 이미 구한 작은 문제의 결괏값을 저장해 재사용하므로 **중복 연산과 반복 탐색을 줄여 실행 시간을 단축**한다. ④는 이 효과를 반대로 설명했다.

선지별 해설

① O: 동적 계획법은 큰 문제를 작은 부분 문제로 나누고, **부분 문제의 결과를 결합**하여 원래 문제의 결과를 구한다. 따라서 작은 문제의 결과로 큰 문제를 해결할 수 있는 구조가 필요하다.

② O: 같은 부분 문제가 여러 번 나타나므로, 한 번 구한 결괏값을 저장해 두었다가 **같은 부분 문제가 다시 필요할 때 재사용**한다. 예를 들어 피보나치 수를 재귀적으로 구하면 여러 경로에서 같은 항을 계산하게 되는데, 각 항의 값을 저장하면 이를 반복해서 계산하지 않아도 된다.

③ O: 상향식은 **작은 부분 문제부터 차례로 계산**하여 그 결과로 큰 문제를 해결한다. 하향식은 큰 문제에서 필요한 부분 문제를 재귀적으로 호출하고, 이미 구한 결과는 메모하기로 재사용한다. 두 방식 모두 부분 문제의 결과를 저장하여 활용할 수 있다.

④ X: 메모하기는 부분 문제를 계산하기 전에 저장된 결과가 있는지 확인하고, 있으면 다시 계산하지 않고 그 값을 사용한다. 따라서 '연산과 탐색이 증가하고 실행속도가 저하된다'는 설명은 **중복 연산과 반복 탐색을 줄여 실행속도를 높인다**로 고쳐야 한다. 결과를 저장할 메모리는 추가로 필요하지만, 같은 부분 문제를 반복해서 푸는 비용을 줄일 수 있다.

NoteBase

문 4. 다음 정렬 알고리즘 중에서 동일한 최악 시간 복잡도를 가진 것만을 모두 고르면?

ㄱ. 선택 정렬(selection sort)
ㄴ. 삽입 정렬(insertion sort)
ㄷ. 힙 정렬(heap sort)
ㄹ. 퀵 정렬(quick sort)

① ㄱ, ㄴ, ㄷ
② ㄱ, ㄴ, ㄹ
③ ㄱ, ㄷ, ㄹ
④ ㄴ, ㄷ, ㄹ

정답 2번

간단한 해설

퀵 정렬은 평균적으로 빠르더라도 **분할이 한쪽으로 치우치는 최악의 경우**에는 $O(n^2)$의 시간이 걸린다. 힙 정렬은 최악에도 $O(n \log n)$이므로 구별된다.

NoteBase

보기별 해설

n을 정렬할 원소의 수라고 할 때, 평균적인 실행 시간이 아니라 가장 불리한 입력에서의 시간 복잡도를 비교한다.

ㄱ. 선택 정렬: 최악 시간 복잡도는 $O(n^2)$이다. 정렬되지 않은 부분 전체를 살펴 최솟값을 하나씩 선택한다. **이미 정렬된 입력이라도 남은 부분을 계속 탐색**하므로, 비교 횟수는 처음에 n − 1회, 다음에 n − 2회처럼 줄어들며 전체적으로 제곱에 비례한다.

ㄴ. 삽입 정렬: 최악 시간 복잡도는 $O(n^2)$이다. 각 원소를 앞쪽의 정렬된 부분에 삽입하면서 더 큰 원소들을 뒤로 옮긴다. 오름차순으로 정렬할 때 **입력이 역순이면** 새 원소마다 앞선 원소를 모두 옮겨야 하므로, 이동 횟수가 1회, 2회, …, n − 1회로 늘어난다.

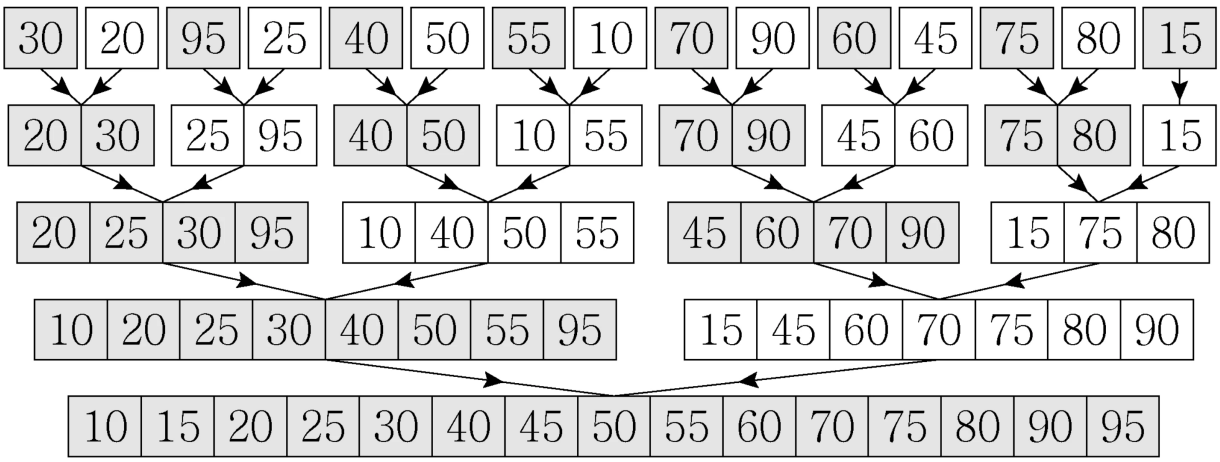
ㄷ. 힙 정렬: 최악 시간 복잡도는 $O(n\log n)$이다. 힙을 만든 뒤 루트의 원소를 꺼내고 힙을 복구하는 과정을 반복한다. 힙은 완전 이진 트리이므로 **복구할 때 내려가는 경로의 길이**가 $O(\log n)$이다. 힙 구성에는 $O(n)$이 들고, 원소를 꺼내며 복구하는 작업을 반복한 전체 시간은 $O(n\log n)$이다.

ㄹ. 퀵 정렬: 최악 시간 복잡도는 $O(n^2)$이다. 피벗을 기준으로 분할할 때마다 **피벗이 최솟값이나 최댓값으로 선택되면** 한쪽 부분은 비고 다른 쪽에는 나머지 원소가 모두 남는다. 처리할 부분의 크기가 하나씩만 줄어들면서 각 분할에 선형 시간이 들기 때문에, 전체 시간이 제곱에 비례한다.

최악 시간 복잡도가 $O(n^2)$로 같은 것은 ㄱ · ㄴ · ㄹ이며, ②의 조합에 해당한다.

NoteBase

문 5. 그림과 같은 과정을 포함하여 정렬을 실행하는 알고리즘은?



① 퀵 정렬

② 셸 정렬(shell sort)

③ 기수 정렬(radix sort)

④ 병합 정렬(merge sort)

정답 4번

간단한 해설

그림은 **정렬된 작은 묶음들을 두 개씩 합쳐** 더 큰 정렬된 묶음을 만드는 과정이다. 이렇게 병합을 반복하는 알고리즘이 **병합 정렬**이다.

공통 해설

병합 정렬은 작은 부분 배열을 각각 정렬한 뒤, **정렬된 두 부분 배열을 하나의 정렬된 배열로 합치는** 과정을 반복한다. 원소가 하나인 배열은 그 자체로 정렬되어 있으므로 병합의 출발점이 된다. 그림에서는 맨 위의 원소들이 아래로 내려가면서 두 개씩 정렬된 묶음을 이루고, 이 묶음들이 다시 합쳐진다.

그림 왼쪽의 [20, 30]과 [25, 95]를 합칠 때는 각 묶음에서 아직 꺼내지 않은 맨 앞 원소를 비교한다. 20과 25 중 20을 먼저 넣고, 다음에는 30과 25 중 25를 넣는다. 이어서 30을 넣고 남은 95를 붙이면 [20, 25, 30, 95]가 된다. 각 묶음이 이미 정렬되어 있어 **맨 앞 원소끼리 비교해 작은 것부터** 꺼내면 전체 순서를 유지할 수 있다.

오른쪽의 [75, 80]과 원소 하나인 [15]도 같은 방식으로 합쳐 [15, 75, 80]이 된다. 따라서 합칠 두 묶음의 길이가 반드시 같을 필요는 없다. 그림처럼 정렬된 묶음의 크기를 키워 가며 마지막에 전체를 하나로 합치는 과정은 병합 정렬에 해당한다.

NoteBase

문 6. 다음과 같은 배열 A에서 A[0]부터 삽입 연산을 차례대로 적용하여 이진 탐색 트리 T를 생성한 후, T를 전위(preorder) 순회 방법으로 방문한 값들을 배열 B에 B[0]부터 순차적으로 저장한 결과는?

A[] = {40, 20, 30, 10}

① B[] = {10, 20, 30, 40}

② B[] = {10, 30, 20, 40}

③ B[] = {40, 20, 10, 30}

④ B[] = {40, 30, 20, 10}

정답 3번

간단한 해설

40의 왼쪽 자식인 20 아래에 왼쪽 자식 10과 오른쪽 자식 30이 놓인다. 전위 순회는 **루트 → 왼쪽 서브트리 → 오른쪽 서브트리** 순서이므로, 20을 방문한 뒤에는 30보다 10을 먼저 방문한다.

올바른 순서

40 → 20 → 10 → 30

트리 구성: 이진 탐색 트리에 값을 삽입할 때는 루트부터 비교하여 **작으면 왼쪽, 크면 오른쪽**으로 내려가고, 해당 방향의 자식이 없는 위치에 새 노드를 넣는다. A[0]부터 삽입하므로 40이 루트가 된다. 20은 40보다 작아 40의 왼쪽 자식이 된다. 30은 40보다 작고 20보다 크므로 20의 오른쪽 자식이 된다. 10은 40과 20보다 모두 작으므로 20의 왼쪽 자식이 된다.

40 방문: 전위 순회는 각 서브트리에서 **루트를 먼저 방문**한 다음 왼쪽 서브트리와 오른쪽 서브트리를 차례로 순회한다. 전체 트리의 루트 40을 먼저 방문하여 B[0]에 저장한다.

20 방문: 40을 방문한 뒤 왼쪽 서브트리로 이동한다. 이 서브트리의 루트는 20이므로, 그 자식들보다 먼저 방문하여 B[1]에 저장한다.

10 방문: 20을 방문한 뒤에는 **왼쪽 서브트리부터** 순회하므로 10을 방문하여 B[2]에 저장한다. 10에는 자식이 없어 이 서브트리의 순회가 끝난다.

30 방문: 20의 왼쪽 서브트리를 마쳤으므로 오른쪽 자식 30을 방문하여 B[3]에 저장한다. 30에도 자식이 없고 40의 오른쪽 서브트리도 없으므로 순회를 마친다.

NoteBase

문 7. 30억 개의 정수를 갖는 배열에서 20개의 정수를 제외한 나머지가 모두 정렬되어 있다면, 이 배열을 가장 빠르게 정렬할 수 있는 알고리즘은?

① 퀵 정렬

② 선택 정렬

③ 병합 정렬

④ 삽입 정렬

정답 4번

간단한 해설

삽입 정렬은 **이미 정렬된 부분을 활용하여 순서가 어긋난 만큼만 원소를 이동**한다. 30억 개 중 20개를 제외한 나머지가 정렬되어 있으므로, 전체를 반복해서 분할하거나 최솟값을 찾는 방식보다 이 조건을 효과적으로 활용한다.

NoteBase

선지별 해설

원소 수를 n이라 하여, 일반적인 각 정렬 알고리즘의 비교·이동 작업량과 이미 정렬된 상태를 활용하는 정도를 비교한다. 여기서는 n이 30억이고, 20개를 제외한 원소들의 상대적인 순서가 이미 맞아 있다.

① 퀵 정렬: 피벗을 기준으로 원소를 나누고 각 부분을 재귀적으로 정렬한다. 평균 시간 복잡도는 $O(n \log n)$이며, 거의 정렬되어 있다는 조건만으로 분할 작업이 생략되지는 않는다. 피벗 선택에 따라 분할이 한쪽으로 치우치면 $O(n^2)$까지 느려질 수 있다.

② 선택 정렬: 아직 정렬하지 않은 부분에서 최솟값을 찾아 앞자리와 교환한다. **이미 정렬된 원소가 많아도 최솟값을 찾는 비교를 반복**하므로 시간 복잡도는 $O(n^2)$이다.

③ 병합 정렬: 배열을 나눈 뒤 정렬된 부분 배열들을 합친다. 일반적인 병합 정렬은 각 단계에서 전체 원소를 병합하고 이런 단계를 로그 횟수만큼 반복하므로 $O(n \log n)$이 걸린다. 이 문항의 거의 정렬된 상태를 삽입 정렬처럼 이동 횟수 감소로 직접 활용하지는 않는다.

④ 삽입 정렬: 앞부분을 정렬된 상태로 유지하면서 다음 원소보다 큰 원소들을 오른쪽으로 밀고, 그 자리에 다음 원소를 넣는다. 이미 순서가 맞으면 이동 없이 다음 원소로 넘어간다. 예를 들어 1, 2, 4, 3, 5에서는 3을 넣을 때 4만 한 칸 밀면 된다. 앞의 값이 뒤의 값보다 큰 원소 쌍을 '역전'이라 하고 그 개수를 I라 하면, 삽입 정렬의 시간 복잡도는 $O(n + I)$이다. 이 배열에서 역전은 순서가 어긋난 20개 원소 중 하나 이상을 포함하므로 그 수는 최대 $20n$ 이하이다. 따라서 **20을 고정된 수로 보면 선형 시간**에 처리할 수 있다. 배열 전체를 확인하고 필요한 원소를 이동하므로, 단순히 20번만 연산한다는 뜻은 아니다.

이처럼 대부분 정렬된 배열에서는 기존 순서를 활용해 역전된 쌍에 대해서만 이동하는 ④ 삽입 정렬이 가장 적합하다.

NoteBase

문 8. 알고리즘의 시간 복잡도에 대한 설명으로 옳은 것은?

① 널리 사용되는 것은 최악 시간 복잡도이다.

② 모든 알고리즘은 최악 시간 복잡도와 평균 시간 복잡도가 다르다.

③ 최악의 경우 실행 시간은 모든 입력에 대한 실행 시간의 하한을 제공한다.

④ 최선의 경우 실행 시간은 모든 입력에 대한 실행 시간의 상한을 제공한다.

정답 1번

간단한 해설

최악 시간 복잡도는 같은 크기의 입력 중 실행 시간이 가장 긴 경우를 기준으로 한다. 어떤 입력이 주어져도 실행 시간이 그 한계를 넘지 않는다는 보장을 제공하므로 알고리즘 분석에 널리 사용된다.

선지별 해설

최선·평균·최악을 비교할 때에는 입력 크기를 같게 두고, 입력 내용에 따라 달라지는 실행 시간을 살핀다.

① O: 최악 시간 복잡도는 같은 크기의 모든 입력 중 실행 시간이 가장 긴 경우를 분석한다. **입력의 확률분포를 가정하지 않고도** 실행 시간이 얼마나 길어질 수 있는지 보장할 수 있어 널리 사용된다.

② X: 최악 시간 복잡도와 평균 시간 복잡도가 **항상 다른 것은 아니다**. 평균 시간 복잡도는 입력이 나타날 확률을 고려한 실행 시간의 평균을 분석한다. 입력 내용과 관계없이 수행하는 연산 수가 같으면 최악과 평균 시간 복잡도도 같다. 예를 들어 원소가 n개인 배열에서 첫 원소를 기준으로 나머지 원소를 모두 비교하여 최댓값을 찾으면, 비교 횟수는 항상 n - 1회이다.

③ X: 최악의 경우 실행 시간은 같은 크기의 모든 입력에 대한 실행 시간의 **상한**을 제공한다. 그중 가장 긴 실행 시간이므로, 어느 입력의 실행 시간도 이를 초과하지 않는다. 따라서 '하한'을 '상한'으로 고쳐야 한다.

④ X: 최선의 경우 실행 시간은 같은 크기의 모든 입력에 대한 실행 시간의 **하한**을 제공한다. 그중 가장 짧은 실행 시간이므로, 어느 입력의 실행 시간도 이보다 짧지 않다. 따라서 '상한'을 '하한'으로 고쳐야 한다.

NoteBase

문 9. 다항적 시간 복잡도를 갖는 탐욕(greedy) 알고리즘으로 최적의 해를 구할 수 없는 것은?

① 부분 배낭(fractional knapsack) 문제

② 한 정점에서 다른 정점으로의 최단 경로 탐색 문제

③ 이진 트리의 최대합 경로 찾기 문제

④ 최소 신장 트리(MST, Minimum Spanning Tree) 생성 문제

정답 3번

간단한 해설

이진 트리에서 바로 다음 노드의 값이 크다고 해서 경로 전체의 합도 최대가 되는 것은 아니다. **각 서브트리의 경로 합을 먼저 구해 비교·결합**해야 하므로, 당장의 이득만으로 선택을 확정하는 탐욕 방법은 최적해를 보장하지 못한다.

NoteBase

선지별 해설

탐욕 알고리즘은 매 단계에서 가장 유리한 선택을 확정한다. 이 방법으로 최적해를 보장하려면 그 선택이 전체 최적해로 이어질 수 있어야 한다. 각 문제에서 이러한 선택이 가능한지를 비교한다.

① 부분 배낭 문제: **단위 무게당 가치가 큰 물건부터** 담는 탐욕 알고리즘으로 최적해를 구한다. 물건을 나누어 담을 수 있으므로, 가치 밀도가 낮은 물건을 담은 부분을 같은 무게의 가치 밀도가 높은 물건으로 바꾸면 총가치가 줄지 않는다. 따라서 높은 가치 밀도부터 담고 마지막 물건은 남은 용량만큼만 담으면 된다. 정렬과 순회로 다항 시간에 처리할 수 있다.

② 최단 경로 탐색 문제: **간선 가중치가 모두 음이 아닌 경우** 다익스트라 알고리즘을 적용할 수 있다. 출발점으로부터의 현재 거리 추정값이 가장 작은 미확정 정점을 선택해 거리를 확정하고, 그 정점을 거치는 경로로 이웃의 거리 추정값을 갱신한다. 음의 간선이 없으므로 나중에 다른 미확정 정점을 거쳐 돌아와도 확정한 거리보다 짧아질 수 없다. 이 탐욕적 확정 과정을 반복하면 목적지까지의 최단 경로를 다항 시간에 구한다. 단순히 현재 정점에서 가장 짧은 간선만 따라가는 방법과는 다르다.

③ 이진 트리의 최대합 경로 찾기 문제: **바로 다음 노드의 값만으로는 전체 경로 합을 판단할 수 없다**. 예를 들어 한 노드에서 아래로 이어지는 두 후보 경로의 값이 각각 9 → 1, 2 → 100이면, 다음 값이 큰 9를 택한 쪽의 이후 합은 10이지만 2를 택한 쪽은 102이다. 이처럼 뒤에 이어질 값들을 함께 고려해야 한다. 각 서브트리의 경로 합을 아래에서 위로 계산하고 필요한 경로를 비교·결합하는 동적 계획법으로 최적해를 구할 수 있지만, 당장의 값만 보고 선택을 확정하는 탐욕 알고리즘으로는 최적해를 보장하지 못한다. 이 문제 자체를 다항 시간에 풀 수 없다는 뜻은 아니다.

④ 최소 신장 트리 생성 문제: 연결된 무방향 가중 그래프에서 프림 또는 크루스컬 알고리즘으로 최적해를 구한다. 프림은 현재 트리와 바깥 정점을 잇는 간선 중 최소 가중치 간선을 추가하고, 크루스컬은 가중치가 작은 간선부터 확인하여 사이클을 만들지 않는 간선을 추가한다. 두 방법은 **선택한 간선을 포함하는 최소 신장 트리가 존재하도록** 트리를 확장하는 탐욕 알고리즘이며, 다항 시간에 수행된다.

탐욕 알고리즘으로 최적해를 보장할 수 없는 대상으로 묻는 것은 ③의 이진 트리 최대합 경로 찾기 문제이다.

NoteBase

문 10. 해시(hash) 함수가 h(k)=k mod 8이고 키값(k)이 15, 11, 5, 13, 22, 21 순서로 저장된 해시 테이블의 결과가 다음과 같은 경우에 사용된 충돌 해결 기법은?

인덱스	0	1	2	3	4	5	6	7
키값	22	21		11		5	13	15

① 체이닝(chaining)

② 선형 조사법(linear probing)

③ 분기 한정법(branch and bound)

④ 이차 조사법(quadratic probing)

정답 2번

간단한 해설

충돌이 나면 **바로 다음 인덱스를 차례로 확인**하며, 인덱스 7 다음에는 0으로 돌아간다. 키 22가 6·7을 지나 0에, 키 21이 5·6·7·0을 지나 1에 저장된 모습이 이 규칙을 보여 준다.

NoteBase

계산 과정

해시 함수 h(k)=k 8은 키를 8로 나눈 나머지를 최초 저장 위치로 정한다. 선형 조사법은 그 위치가 차 있으면 **인덱스를 1씩 증가시키며 처음 만나는 빈칸**에 저장한다. 인덱스는 0~7이므로 7 다음에는 0으로 돌아간다.

키 15: h(15)=15 8=7이다. 인덱스 7이 비어 있으므로 그곳에 저장한다.

키 11: h(11)=11 8=3이다. 인덱스 3이 비어 있으므로 그곳에 저장한다.

키 5: h(5)=5 8=5이다. 인덱스 5가 비어 있으므로 그곳에 저장한다.

키 13: h(13)=13 8=5이지만 인덱스 5에는 키 5가 있다. 다음 인덱스 6이 비어 있으므로 키 13을 6에 저장한다.

키 22: h(22)=22 8=6이다. 인덱스 6에는 키 13, 7에는 키 15가 있으므로 차례로 건너뛴다. **끝에 도달하면 처음으로 돌아가** 빈 인덱스 0에 키 22를 저장한다.

키 21: h(21)=21 8=5이다. 인덱스 5·6·7·0에는 각각 키 5·13·15·22가 있으므로 모두 건너뛴다. 그다음 빈 인덱스 1에 키 21을 저장한다.

최종 저장 위치는 0에 22, 1에 21, 3에 11, 5에 5, 6에 13, 7에 15로 그림과 일치한다. 충돌할 때마다 인접한 다음 칸을 확인한 것이므로 사용된 기법은 ② 선형 조사법이다.

NoteBase

문 11. 함수 $f(n)$에 대한 점근 표기법으로 옳지 않은 것은?

$f(n) = 7n\log n + \log n - 20n + 6$

① $f(n) = \Omega(n^2)$

② $f(n) = O(n^2)$

③ $f(n) = \Theta(n\log n)$

④ $f(n) + n^2 = O(n^2)$

정답 1번

간단한 해설

①의 하한 표기는 성립하지 않는다. $f(n)$은 $n\log n$**과 같은 차수**로 증가하므로, 더 빠르게 증가하는 n^2의 양의 상수배를 계속 웃돌 수 없다.

선지별 해설

점근 표기는 입력 크기 n이 충분히 커졌을 때의 증가율을 나타낸다. O는 상수배에 의한 상한, Ω는 상수배에 의한 하한, Θ는 두 조건을 모두 만족하는 범위이다. 주어진 함수에서 $\log n$, $-20n$, 6은 $n\log n$에 비해 영향이 작아진다. 따라서 $\frac{f(n)}{n\log n} \to 7$이며, 큰 n에서는 $f(n)$이 양수이다.

① X: $\frac{\log n}{n} \to 0$이므로 $\frac{f(n)}{n^2} \to 0$이다. $f(n) = \Omega(n^2)$이려면 어떤 양의 상수 c에 대해 충분히 큰 모든 n에서 $f(n) \geq cn^2$이어야 하는데, 비율이 0으로 가므로 이 조건을 만족할 수 없다. 하한의 기준을 $n\log n$**으로 고친** $f(n) = \Omega(n\log n)$은 성립한다.

② O: $n\log n$은 n^2보다 느리게 증가하므로, 충분히 큰 n에서 $f(n)$을 n^2의 일정한 양의 상수배 이하로 제한할 수 있다. 따라서 $f(n) = O(n^2)$이다. O는 **상한을 나타내며 정확히 같은 증가율을 요구하지 않는다**.

③ O: $\frac{f(n)}{n\log n}$이 양의 상수 7로 수렴하므로, 충분히 큰 n에서 $f(n)$은 $n\log n$의 두 양의 상수배 사이에 놓인다. **상한과 하한이 모두 성립**하므로 $f(n) = \Theta(n\log n)$이다.

④ O: $\frac{f(n)+n^2}{n^2} = 1 + \frac{f(n)}{n^2} \to 1$이다. 합에서 **더 빠르게 증가하는 n^2이 지배항**이 되므로 $f(n) + n^2 = \Theta(n^2)$이며, 이에 따라 $f(n) + n^2 = O(n^2)$도 성립한다.

NoteBase

문 12. 다음 문자열에 대하여 허프만 코딩(Huffman coding) 알고리즘으로 생성한 허프만 트리에서 루트(root) 노드부터 가장 깊은 단말(leaf) 노드까지 도달하기 위한 단순 경로상의 간선(edge) 개수는?

AAAEEBCCDDDDCCEEEAADDEEEBB

① 3

② 4

③ 5

④ 6

정답 1번

간단한 해설

허프만 트리는 매번 **빈도가 가장 작은 두 노드**를 합쳐 만든다. 이 문자열에서는 B와 C가 가장 깊은 단말이 되며, 루트에서 이들까지의 경로에는 **간선이 3개** 있다.

계산 과정

문자별 빈도: 주어진 문자열에서 A는 5회, B는 3회, C는 4회, D는 6회, E는 8회 나타난다. 각 문자를 이 빈도를 가중치로 갖는 단말 노드로 놓는다.

노드 결합: 허프만 코딩은 **남아 있는 노드 중 가중치가 가장 작은 두 노드**를 골라, 두 가중치의 합을 갖는 부모 노드로 묶는다. 먼저 B(3)와 C(4)를 합쳐 가중치 7인 노드를 만든다. 남은 가중치는 5, 6, 7, 8이므로 A(5)와 D(6)을 합쳐 11인 노드를 만든다. 이어 7과 E(8)을 합쳐 15인 노드를 만들고, 마지막으로 11과 15를 합쳐 26인 루트 노드를 만든다.

최대 깊이: 루트에서 B까지의 경로는 '루트(26) → 내부 노드(15) → 내부 노드(7) → B(3)'이다. C까지의 경로도 같은 두 내부 노드를 거치므로 간선이 3개이다. A, D, E까지는 각각 간선이 2개이므로, 가장 깊은 단말까지의 간선 개수는 **3개**이며 ①의 값과 일치한다. 전체 결합 횟수 4회가 아니라, 루트에서 하나의 단말까지 거치는 간선 수를 센다.

NoteBase

문 13. 그래프(graph) 구조로 데이터를 저장하고 그래프 알고리즘을 이용하여 문제를 해결하는 대표적인 예만을 모두 고르면?

ㄱ. 소셜 네트워크에서 친구 추천하기
ㄴ. 가장 적은 수의 동전으로 거스름돈 돌려받기
ㄷ. 최소 비용의 여행 경로 탐색하기
ㄹ. 정렬된 1차원 배열 구조에서 특정값 빠르게 찾기

① ㄱ, ㄷ
② ㄱ, ㄹ
③ ㄴ, ㄷ
④ ㄴ, ㄹ

정답 1번

간단한 해설

친구 추천은 사람 사이의 **연결 관계**를, 최소 비용 여행 경로 탐색은 장소 사이의 **이동 경로와 비용**을 이용한다. 사람이나 장소를 정점으로, 친구 관계나 이동 구간을 간선으로 나타내면 그래프 알고리즘으로 문제를 해결할 수 있다.

NoteBase

보기별 해설

그래프는 대상을 나타내는 정점과 정점 사이의 관계를 나타내는 간선으로 구성된다. 이 문제에서는 그래프로 표현할 수 있는 가능성보다, 관계나 경로를 그래프로 저장하고 처리하는 대표적인 활용인지를 구분한다.

ㄱ. 친구 추천: 사람을 정점으로, 친구 관계를 간선으로 나타낸다. 예를 들어 A와 B가 친구이고 B와 C가 친구라면, A와 아직 친구가 아닌 C를 추천 후보로 찾을 수 있다. 이처럼 **공통 친구나 연결 관계를 탐색**하므로 그래프의 대표적인 활용이다.

ㄴ. 동전 교환: 거스름돈 금액을 만들면서 사용한 동전 수를 최소화하는 문제이다. 대표적으로 **금액별 최소 동전 수를 구하는 동적 계획법**으로 해결하며, 동전 체계에 따라 큰 금액의 동전부터 고르는 탐욕법을 사용할 수도 있다. 그래프로 모델링하는 방법도 있지만, 그래프 구조로 데이터를 저장하는 대표적 활용을 묻는 이 문제의 분류에는 포함되지 않는다.

ㄷ. 최소 비용 여행 경로 탐색: 장소를 정점으로, 이동할 수 있는 구간을 간선으로 나타내고 간선에 이동 비용을 부여한다. 이동 조건에 맞는 경로 중 **간선 비용의 합이 가장 작은 경로**를 찾으므로 가중치 그래프를 활용하는 대표적인 예이다.

ㄹ. 정렬된 배열에서 값 찾기: **이진 탐색**을 대표적으로 사용한다. 가운데 원소와 찾는 값을 비교한 뒤, 정렬 순서를 이용해 값이 있을 수 없는 절반을 제외하는 과정을 반복한다. 배열의 정렬 순서와 인덱스를 이용하는 탐색이므로 그래프의 대표적인 활용에 해당하지 않는다.

그래프 구조와 그래프 알고리즘을 활용하는 대표적인 예는 ㄱ, ㄷ이며, 이 조합은 ①에 해당한다.

NoteBase

문 14. 다음 C 프로그램의 실행 결과는?

```
#include <stdio.h>
int count = 0;
int A[] = {1, 3, 5, 8, 12, 15, 20, 24, 30, 44, 52,
61, 64, 70, 81, 90};

int bin(int low, int high, int key) {
        int mid;
        count++;
        if (low > high) return 0;
        else {
             mid = (low + high) / 2;
             if (key == A[mid]) return 1;
             else if (key < A[mid])
                 return bin(low, mid - 1, key);
             else
                 return bin(mid + 1, high, key);
        }
}

int main() {
        bin(0, 15, 22);
        printf("count: %d, ", count);
        bin(0, 15, 52);
        printf("count: %d\n", count);

        return 0;
}
```

① count: 4, count: 8

② count: 4, count: 9

③ count: 5, count: 9

④ count: 5, count: 10

정답 3번

간단한 해설

count++가 종료 조건 검사보다 먼저 실행되므로 **탐색 실패로 끝나는 호출도 횟수에 포함**된다. 22를 탐색할 때 5회, 52를 탐색할 때 4회 호출되며, 전역변수 count를 초기화하지 않아 두 번째 출력은 **누적값 9**가 된다.

NoteBase

계산 과정

호출 횟수 계산 기준: count는 0에서 시작하는 전역변수이며 bin이 호출될 때마다 1씩 증가한다. 배열 인덱스는 0부터 15까지이고, low와 high는 현재 탐색 구간의 양 끝 인덱스이다. mid는 $\left\lfloor \frac{low + high}{2} \right\rfloor$로 정한다. 두 인덱스가 음이 아닌 정수이므로 C의 정수 나눗셈에서 소수 부분을 버린 결과와 같다. 오름차순 배열이므로 key가 A[mid]보다 작으면 왼쪽, 크면 오른쪽 구간을 탐색한다.

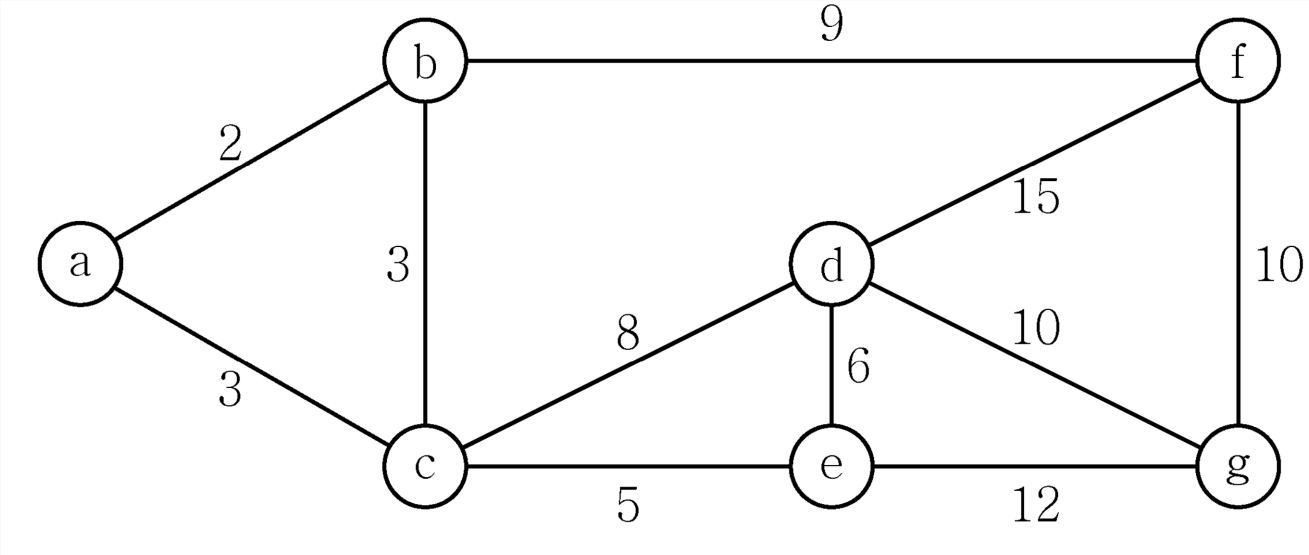
22 탐색: 탐색 구간은 [0, 15] → [0, 6] → [4, 6] → [6, 6] → [7, 6]으로 바뀐다. 앞의 네 호출에서 mid는 차례로 7, 3, 5, 6이고 A[mid]는 24, 8, 15, 20이다. 22는 24보다 작아서 왼쪽으로 이동한 뒤, 8·15·20보다 커서 오른쪽으로 이동한다. 다섯 번째 호출에서는 low가 7, high가 6이므로 탐색을 실패로 끝낸다. 이때도 **count++가 먼저 실행**되므로 count는 5가 되고 첫 printf는 'count: 5, '를 출력한다.

52 탐색: 앞선 탐색의 count = 5를 유지한 채 시작한다. 탐색 구간은 [0, 15] → [8, 15] → [8, 10] → [10, 10]으로 바뀐다. mid는 차례로 7, 11, 9, 10이고 A[mid]는 24, 61, 44, 52이다. 52는 24보다 커서 오른쪽, 61보다 작아서 왼쪽, 44보다 커서 오른쪽으로 이동한 뒤 네 번째 호출에서 발견된다. 해당 호출까지 count가 4 증가하며, **발견 즉시 반환하므로 추가 호출은 없다**.

최종 출력: 두 번째 printf에서 출력하는 값은 5 + 4 = 9이다. 따라서 한 줄에 'count: 5, count: 9'가 출력되며, ③의 값과 일치한다.

NoteBase

문 15. 다음 그래프에서 크루스칼(Kruskal) 알고리즘으로 최소 신장 트리(MST)를 생성할 때 다섯 번째로 추가되는 간선은? (단, 초기 MST는 공집합이다)



① (b, f)

② (c, d)

③ (d, e)

④ (d, g)

정답 1번

간단한 해설

크루스칼 알고리즘은 가중치가 작은 간선부터 검토하되 **사이클을 만드는 간선은 제외**한다. 네 간선을 추가한 뒤 가중치 8인 (c, d)는 건너뛰고, 아직 연결되지 않은 f를 잇는 (b, f)를 다섯 번째로 추가한다.

NoteBase

계산 과정

크루스칼 알고리즘은 모든 간선을 가중치의 오름차순으로 검토하고, 양 끝 정점이 서로 다른 연결 요소에 있을 때만 간선을 추가한다. 이미 연결된 두 정점을 다시 이으면 사이클이 생기므로 제외한다. 초기 간선 집합은 공집합이므로 각 정점은 서로 분리되어 있으며, 추가 횟수는 **검토한 간선이 아니라 실제로 채택한 간선**을 센다.

첫 번째 추가: 가중치가 가장 작은 (a, b)의 가중치는 2이다. 이를 추가하면 a와 b가 하나의 연결 요소가 된다.

두 번째 추가: (a, c)와 (b, c)는 모두 가중치가 3이다. 둘 중 먼저 검토한 간선을 추가하면 a, b, c가 연결된다. 나머지 하나는 a, b, c 사이에 사이클을 만들므로 제외한다. **동일 가중치 간선의 처리 순서가 달라도** 이 단계까지 추가된 간선은 2개이고 연결된 정점 집합은 같다.

세 번째 추가: 다음으로 작은 (c, e)의 가중치는 5이다. e는 a, b, c와 분리되어 있으므로 이 간선을 추가한다. 이제 a, b, c, e가 연결된다.

네 번째 추가: (d, e)의 가중치는 6이다. d가 기존 연결 요소와 분리되어 있으므로 추가한다. 이로써 a, b, c, d, e가 연결되고, 추가된 간선은 4개가 된다.

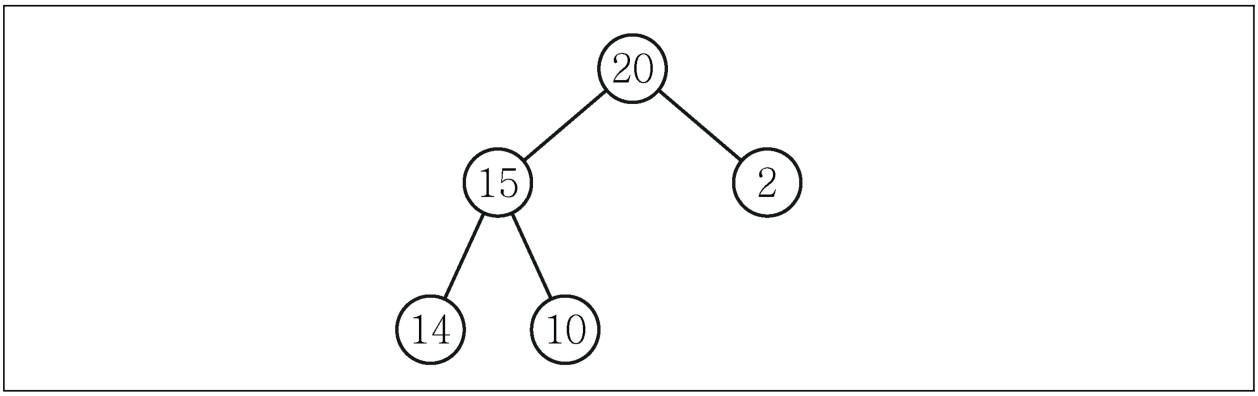
다음 검토: (c, d)의 가중치는 8이지만 c와 d는 이미 c → e → d 경로로 연결되어 있다. 이를 추가하면 c → e → d → c의 **사이클이 생기므로 제외**한다. 따라서 추가된 간선 수는 여전히 4개이다.

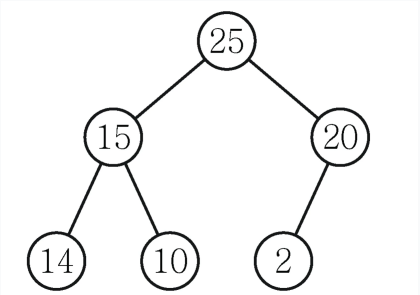
다섯 번째 추가: 그다음 간선 (b, f)의 가중치는 9이다. f는 아직 기존 연결 요소에 속하지 않으므로 사이클 없이 연결할 수 있다. 나머지 미검토 간선의 가중치는 모두 10 이상이므로 (b, f)가 다섯 번째로 추가된다.

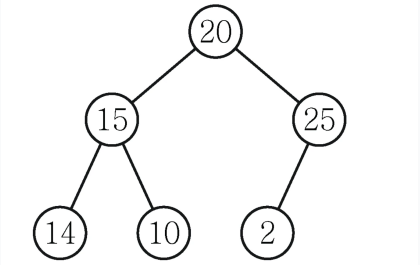
따라서 다섯 번째로 추가되는 간선은 (b, f)이며, ①에 해당한다.

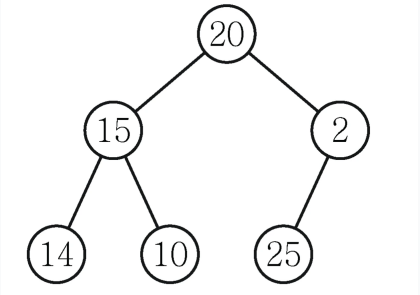
NoteBase

문 16. 다음 최대 힙(max heap)에 노드 25가 추가될 경우, 최대 힙 성질을 만족하도록 삽입 연산이 완료된 후의 구조로 옳은 것은?



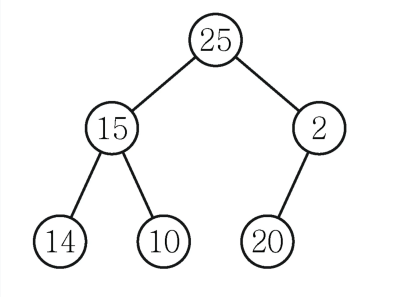
① 

② 

③ 

NoteBase

④



정답 1번

간단한 해설

새 노드 25는 먼저 2의 왼쪽 자식 자리에 삽입된다. 이후 **부모보다 큰 동안 위로 이동**하므로 2, 20과 차례로 자리를 바꾸어 루트가 되고, 20의 왼쪽 자식에는 2가 남는다.

계산 과정

삽입 위치: 최대 힙은 완전 이진 트리이며, 모든 부모의 값이 자식의 값 이상이어야 한다. 완전 이진 트리는 마지막 층을 제외한 층이 모두 차 있고, 마지막 층은 왼쪽부터 채워진다. 주어진 트리의 마지막 층에는 14와 10이 있으므로, 새 노드 25는 **2의 왼쪽 자식 자리**에 넣는다.

첫 번째 교환: 삽입한 25의 부모는 2이다. 25 > 2이므로 부모가 자식보다 작아 최대 힙 조건을 위반한다. **25와 2의 자리를 교환**하면 25는 루트 20의 오른쪽 자식이 되고, 2는 25의 왼쪽 자식이 된다.

두 번째 교환: 위로 올라온 25의 새 부모는 20이다. 여전히 25 > 20이므로 **25와 20의 자리도 교환**한다. 이제 25가 루트가 되고 20은 그 오른쪽 자식이 되며, 2는 20의 왼쪽 자식으로 남는다. 25는 더 이상 부모가 없으므로 위로 이동하는 연산을 끝낸다.

완료된 구조: 루트는 25, 그 왼쪽과 오른쪽 자식은 각각 15와 20이다. 15의 자식 14와 10은 그대로이고, 20의 왼쪽 자식은 2이다. 모든 부모가 자식보다 크고 완전 이진 트리 형태도 유지되며, 이 구조는 ①의 그림과 일치한다.

NoteBase

문 17. 백트래킹(backtracking)에 대한 설명으로 옳지 않은 것은?

① 스택(stack)을 이용하여 구현할 수 있다.

② 재귀적(recursive)으로 구현할 수 있다.

③ 제약 충족(constraint satisfaction) 문제에 유용하다.

④ 균형 탐색 트리(balanced search tree) 구조를 유지한다.

정답 4번

간단한 해설

백트래킹은 해가 될 가능성이 없는 경로에서 **이전 선택으로 되돌아가 다른 후보를 탐색**하는 기법이다. 탐색 트리의 **균형을 유지하는 기법은 아니다**.

선지별 해설

백트래킹은 해를 향해 선택을 하나씩 확장하다가, 현재 선택으로는 해를 얻을 수 없다고 판단되면 이전 단계로 돌아가 다른 선택을 시도하는 탐색 기법이다.

① O: 탐색 중인 경로의 선택이나 상태를 스택에 저장할 수 있다. 막다른 경로를 만나면 **가장 최근 선택부터 되돌리는** 동작이 스택의 후입선출 방식과 맞는다.

② O: 후보 하나를 선택하고 다음 단계의 탐색 함수를 다시 호출하는 **재귀 호출**로 구현할 수 있다. 더 진행할 수 없으면 호출에서 복귀하여 해당 선택을 취소하고 다음 후보를 시도한다. 이때 호출 스택이 돌아갈 위치와 각 호출의 실행 상태를 보관한다.

③ O: 제약 충족 문제는 변수에 값을 배정하되 주어진 제약을 모두 만족하는 해를 찾는 문제이다. 백트래킹은 값을 하나씩 배정하다가 **제약을 위반한 부분 해를 조기에 제외**할 수 있어 유용하다. 예를 들어 인접한 두 영역을 서로 다른 색으로 칠해야 한다면, 두 영역에 같은 색을 배정한 단계에서 그 선택을 취소하고 다른 색을 시도한다.

④ X: 백트래킹의 탐색 트리는 선택에 따라 가능한 상태가 어떻게 이어지는지를 나타내며, 가지의 수와 깊이는 문제와 선택에 따라 달라진다. 백트래킹은 **해가 될 수 없는 경로의 탐색을 중단**하지만, 트리의 높이나 가지 사이의 균형을 조정하지는 않는다. 따라서 '균형 탐색 트리 구조를 유지한다'는 설명은 '가능성이 없는 경로에서 되돌아가 다른 경로를 탐색한다'로 고쳐야 한다.

NoteBase

문 18. 문자열 매칭을 위한 알고리즘으로 옳지 않은 것은?

① 라빈-카프(Rabin-Karp)

② 벨만-포드(Bellman-Ford)

③ 보이어-무어(Boyer-Moore)

④ KMP(Knuth-Morris-Pratt)

정답 2번

간단한 해설

벨만-포드는 **가중 그래프의 단일 출발점 최단 경로**를 구하는 알고리즘이다. 텍스트에서 특정 패턴의 등장 위치를 찾는 문자열 매칭 알고리즘이 아니다.

선지별 해설

① O: 라빈-카프는 **해시값을 비교**하여 문자열을 매칭한다. 패턴의 해시값과 텍스트에서 패턴 길이만큼 잘라 낸 부분 문자열의 해시값을 비교한다. 해시값이 같아도 서로 다른 문자열일 수 있으므로 실제 문자도 비교하여 일치 여부를 확인한다.

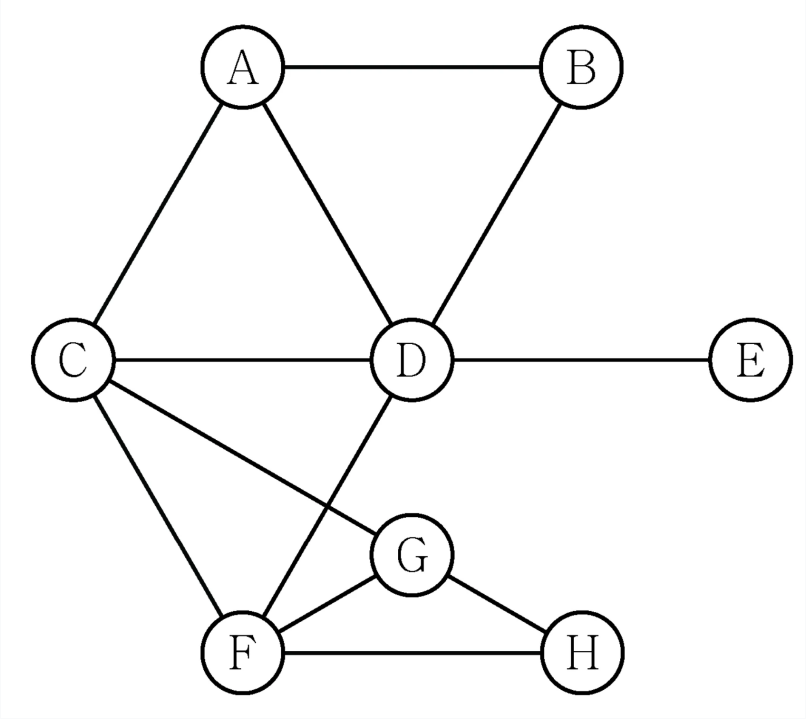
② X: 벨만-포드는 문자열 매칭이 아니라 **가중 그래프의 단일 출발점 최단 경로**를 구하는 알고리즘이다. 간선을 거쳐 이동할 때 경로의 가중치 합이 더 작아지면 거리를 갱신하는 과정을 반복한다. 따라서 텍스트와 패턴의 일치 여부를 찾는 알고리즘으로 분류할 수 없다.

③ O: 보이어-무어는 패턴의 **오른쪽 끝 문자부터** 텍스트와 비교하는 문자열 매칭 알고리즘이다. 불일치가 발생하면 불일치 문자나 이미 일치한 접미부에 관한 정보를 이용해 패턴을 이동시켜 불필요한 비교를 줄인다.

④ O: KMP는 패턴의 **접두부와 접미부가 일치하는 정보**를 미리 구해 이용하는 문자열 매칭 알고리즘이다. 일부 문자가 일치하다가 불일치하면 이 정보를 바탕으로 패턴의 다음 비교 위치를 정하므로, 텍스트의 비교 위치를 뒤로 되돌리지 않고 탐색을 이어 간다.

NoteBase

문 19. 다음 그래프에서 A부터 깊이 우선 탐색(depth first search)을 수행하는 경우, 가능한 정점의 방문 순서로 옳지 않은 것은?



① A → B → D → E → C → F → G → H

② A → C → F → H → G → E → D → B

③ A → C → G → H → F → D → E → B

④ A → D → B → E → C → G → H → F

정답 2번

간단한 해설

②의 순서에서 **E는 D와만 연결**되어 있으므로 E를 처음 방문하려면 D를 먼저 방문해야 한다. G에서 더 진행할 곳이 없으면 H를 거쳐 F로 되돌아간 뒤, 아직 방문하지 않은 D로 탐색을 이어 간다.

NoteBase

선지별 해설

깊이 우선 탐색은 현재 정점에 방문하지 않은 인접 정점이 있으면 그 정점으로 진행하고, 없으면 이전 정점으로 되돌아가는 방식이다. 방문 순서에는 각 정점을 처음 방문한 때만 기록하며, 되돌아가며 다시 거치는 정점은 기록하지 않는다. 인접 정점 중 어느 것을 먼저 선택할지 주어지지 않았으므로 여러 방문 순서가 가능하다.

① O: A → B → D → E로 진행한 뒤, E에는 방문하지 않은 인접 정점이 없으므로 **D로 되돌아간다**. 이어 D → C → F → G → H로 진행하면 제시된 방문 순서가 된다.

② X: A → C → F → H → G까지는 가능하다. G의 인접 정점 C·F·H는 모두 방문한 상태이므로 H를 거쳐 F로 되돌아간다. 이때 F에 아직 방문하지 않은 인접 정점 D가 남아 있으므로 D를 방문해야 한다. 특히 **E는 D와만 연결되어 있어 D보다 먼저 방문할 수 없다**. 가능한 순서의 한 예는 A → C → F → H → G → D → E → B이다. 이 순서에서는 D에서 E를 방문한 뒤 D로 되돌아가 B를 방문한다.

③ O: A → C → G → H → F → D → E까지는 연속된 두 정점 사이에 모두 간선이 있어 차례로 진행할 수 있다. E에서 더 진행할 곳이 없으면 **D로 되돌아가 미방문 정점 B**를 방문하므로 제시된 순서가 가능하다.

④ O: A → D → B로 진행한 뒤 B의 인접 정점 A·D가 모두 방문되었으므로 D로 되돌아간다. D에서 E를 방문하고 다시 D로 돌아온 다음 C → G → H → F로 진행한다. 따라서 **B와 E에서 각각 D로 되돌아가는 과정**을 거치면 제시된 방문 순서가 된다.

NoteBase

문 20. 다음 함수는 0보다 큰 자연수 n이 입력될 때, 1 2 3 4 … n 순서로 출력하는 재귀 함수이다. (가) ~ (다)에 들어갈 내용을 바르게 연결한 것은?

```
void fun(int n) {
    if ( (가) ) {
        (나)
    }
    (다)
}
```

① (가): n > 0, (나): printf("%d ", n);, (다): fun(n-1);

② (가): n > 0, (나): fun(n-1);, (다): printf("%d ", n);

③ (가): n > 1, (나): printf("%d ", n);, (다): fun(n-1);

④ (가): n > 1, (나): fun(n-1);, (다): printf("%d ", n);

정답 4번

간단한 해설

1부터 오름차순으로 출력하려면 **재귀 호출을 먼저 하고 현재 n을 나중에 출력**해야 한다. 출력문은 if문 밖에 있으므로 **n > 1**일 때만 재귀 호출하여, n이 1이면 추가 호출 없이 1을 출력하게 한다.

NoteBase

빈칸별 해설

(가): n > 1. 그림에서 (나)는 if문 안에 있고 (다)는 if문 밖에 있다. 1부터 출력하려면 **n이 1일 때 재귀 호출을 멈춰야** 하므로 조건은 n > 1이다. n > 0으로 두면 n이 1일 때도 fun(0)을 호출하여, if문 밖의 출력문에서 0까지 출력하게 된다.

(나): fun(n-1);. 현재 n보다 작은 수들을 먼저 출력하도록 **출력 전에 재귀 호출**한다. 호출된 함수가 끝나야 현재 함수의 다음 문장을 실행하므로, fun(n-1)이 1부터 n-1까지 출력한 뒤 현재 n을 출력할 수 있다.

(다): printf("%d ", n);. 재귀 호출이 끝난 뒤 현재 n을 출력한다. 이 문장은 **if문 밖에 있으므로** n이 1일 때도 실행된다. 예를 들어 fun(3)은 fun(2), fun(1)을 차례로 호출하고, fun(1)이 1을 출력한 뒤 돌아오면 fun(2)가 2를, 이어 fun(3)이 3을 출력한다. 반대로 (나)에 출력문을 넣으면 현재 n부터 출력하므로 요구한 오름차순이 되지 않는다.

(가)에 n > 1, (나)에 fun(n-1);, (다)에 printf("%d ", n);을 넣는 조합은 ④와 일치한다.

NoteBase