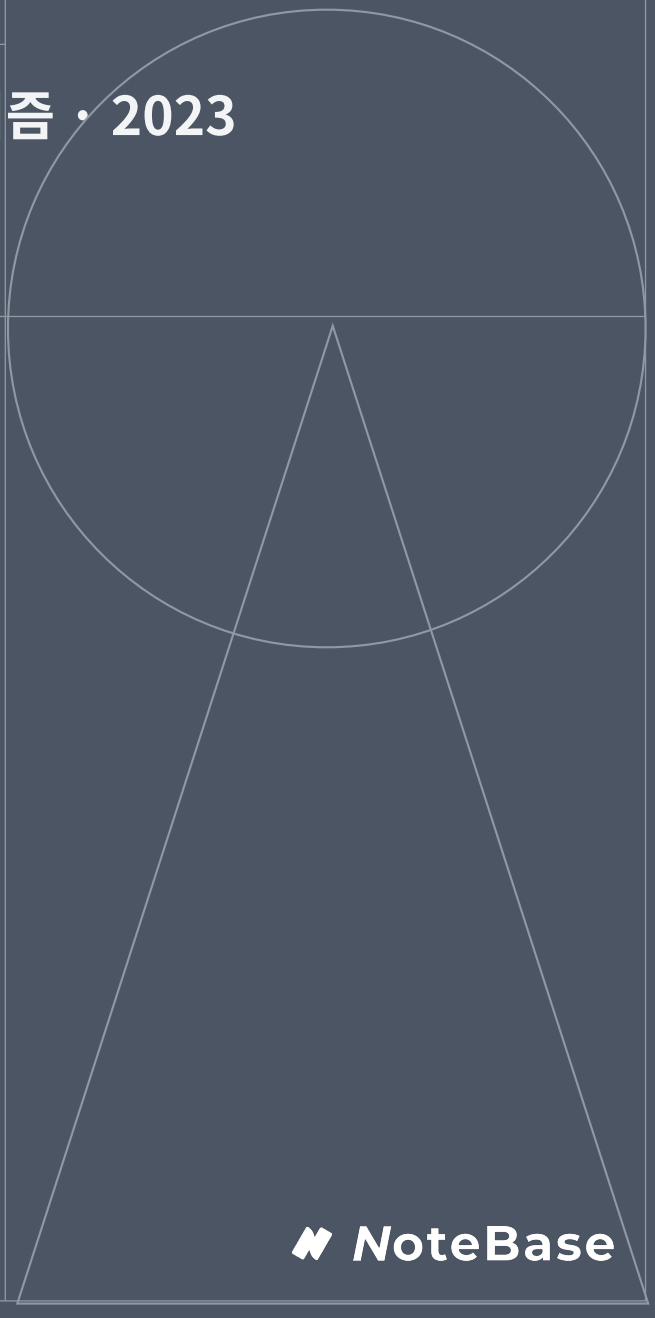
기출OX 해설집

공무원/국가직/9급 · 알고리즘 · 2023



NoteBase

기출OX 해설집

공무원/국가직/9급 · 알고리즘 · 2023

공무원/국가직/9급 · 알고리즘 · 2023

문 1. 알고리즘의 조건에 대한 설명으로 옳은 것만을 모두 고르면?

ㄱ. 모든 명령은 모호하지 않고 명확해야 한다.
ㄴ. 모든 명령은 실행 가능한 연산이어야 한다.
ㄷ. 모든 명령은 반복적으로 무한히 실행되어야 한다.

① ㄱ
② ㄱ, ㄴ
③ ㄴ, ㄷ
④ ㄱ, ㄴ, ㄷ

정답 2번

간단한 해설

알고리즘의 명령은 **명확하고 실행 가능**해야 하며, 전체 절차는 **유한한 단계 안에 종료**되어야 한다. ㄷ의 '무한히 실행되어야 한다'는 유한성 조건에 어긋난다.

보기별 해설

ㄱ. O: 명확성은 각 명령의 의미와 수행 방법이 **모호하지 않게 정해져 있어야 한다**는 조건이다. 예를 들어 '값을 적당히 증가시킨다'보다 '값을 1 증가시킨다'처럼 수행할 연산을 분명히 지정해야 한다.

ㄴ. O: 유효성은 각 명령이 실제로 **수행 가능한 기본 연산**이어야 한다는 조건이다. 명령의 뜻이 명확하더라도 수행할 수 없는 연산이라면 알고리즘의 단계로 사용할 수 없다.

ㄷ. X: 알고리즘은 **유한한 단계 안에 종료**되어야 한다는 유한성 조건을 충족해야 한다. 반복문을 사용할 수는 있지만, 반복도 결국 끝나야 한다. 따라서 '모든 명령은 반복적으로 무한히 실행되어야 한다'는 설명은 '전체 절차는 유한한 단계 안에 종료되어야 한다'로 고쳐야 한다.

옳은 보기는 ㄱ, ㄴ이며, 이 조합은 ②에 해당한다.

NoteBase

문 2. 알고리즘의 수행 시간 분석에 대한 설명으로 옳지 않은 것은?

① 알고리즘의 수행 시간은 컴퓨터 성능에 관계없이 명확하게 정의되어야 한다.

② 알고리즘의 시간복잡도는 수행하는 기본적인 연산 횟수를 입력 크기에 대한 함수로 표현한다.

③ 최선의 경우의 알고리즘 수행 시간은 모든 입력의 수행 시간에 대한 상한이 된다.

④ 알고리즘의 수행 시간 분석에는 최악의 경우, 평균의 경우, 최선의 경우가 있다.

정답 3번

간단한 해설

최선의 경우는 같은 크기의 입력 중 수행 시간이 가장 짧은 경우이므로, 모든 입력의 수행 시간에 대한 **하한**이 된다. 상한이 되는 것은 같은 크기의 입력 중 수행 시간이 가장 긴 **최악의 경우의 수행 시간**이다.

선지별 해설

① O: 실제 실행 시간을 초로 측정하면 컴퓨터 성능이나 실행 환경에 따라 결과가 달라진다. 알고리즘 자체의 효율을 비교하려면 기본 연산의 비용을 정한 계산 모형에서 **기본 연산 횟수**를 세어, 특정 컴퓨터의 성능에 의존하지 않도록 수행 시간을 정의한다.

② O: 시간복잡도는 비교·대입 등 분석의 기준이 되는 기본 연산이 몇 번 수행되는지를 **입력 크기의 함수**로 나타낸다. 이를 통해 입력이 커질 때 필요한 연산량이 어떻게 증가하는지 분석할 수 있다.

③ X: **같은 입력 크기**에서 최선의 경우의 수행 시간은 가능한 수행 시간 중 최솟값이므로 하한이고, 최악의 경우의 수행 시간은 최댓값이므로 상한이다. 따라서 '최선의 경우'를 '최악의 경우'로 고쳐야 한다. 예를 들어 원소 5개에서 목표값을 앞에서부터 비교하는 순차 탐색은 첫 원소가 목표값이면 1번, 마지막 원소가 목표값이면 5번 비교한다. 최소 비교 횟수인 1번을 모든 입력의 비교 횟수에 대한 상한으로 삼을 수는 없다.

④ O: 같은 입력 크기라도 입력 내용에 따라 수행 시간이 달라질 수 있으므로 최악·평균·최선의 경우로 나누어 분석한다. 최악은 가장 오래 걸리는 경우, 최선은 가장 짧게 걸리는 경우이다. 평균은 **각 입력이 나타날 확률**을 반영한 수행 시간의 기댓값이므로, 분석에 사용할 입력의 확률분포가 필요하다.

NoteBase

문 3. 다음은 그래프에서 너비 우선 탐색(breadth first search) 알고리즘이 동작하는 과정이다. (가) ~ (다)에 들어갈 내용을 바르게 연결한 것은?

[단계 1] 시작 정점을 'visited'로 표시하고 (가)에(서) (나)한다.
[단계 2] (가)에(서) 정점을 (다)하고, 제거한 정점의 인접 정점 중 아직 방문하지 않은 곳들은 'visited'로 표시하고 (가)에(서) (나)한다.
[단계 3] (가)가/이 비워질 때까지 [단계 2]를 반복한다.

① (가): Queue, (나): Dequeue, (다): Enqueue
② (가): Queue, (나): Enqueue, (다): Dequeue
③ (가): Stack, (나): Pop, (다): Push
④ (가): Stack, (나): Push, (다): Pop

정답 2번

간단한 해설

너비 우선 탐색은 시작 정점에서 가까운 정점부터 처리하므로 **먼저 넣은 정점을 먼저 꺼내는 큐**를 사용한다. 큐에 정점을 넣는 연산은 Enqueue, 꺼내는 연산은 Dequeue이다.

빈칸별 해설

(가) Queue: 너비 우선 탐색은 시작 정점에서 간선 수를 기준으로 가까운 정점부터 차례로 탐색한다. 이를 위해 **선입선출(FIFO)** 구조인 큐를 사용한다. 예를 들어 A를 넣고 B를 넣으면 A가 먼저 나온다. 스택은 나중에 넣은 B가 먼저 나오는 후입선출(LIFO) 구조이므로 이 처리 순서를 보장하지 않는다.

(나) Enqueue: 큐의 뒤쪽에 정점을 넣는 삽입 연산이다. [단계 1]에서는 시작 정점을 넣고, [단계 2]에서는 제거한 정점의 인접 정점 중 아직 방문하지 않은 정점들을 넣는다. 이때 **큐에 넣기 전에 visited로 표시**하므로 같은 정점이 여러 인접 정점을 통해 중복으로 들어가는 것을 막는다.

(다) Dequeue: 큐의 **맨 앞 정점을 꺼내는** 제거 연산이다. [단계 2]에서는 먼저 들어온 정점을 꺼낸 뒤 그 정점의 미방문 인접 정점들을 큐의 뒤쪽에 넣는다. 따라서 이미 대기 중인 정점들이 새로 넣은 정점들보다 먼저 처리된다. 큐가 비면 처리할 정점이 더 이상 없으므로 반복을 끝낸다.

(가) Queue, (나) Enqueue, (다) Dequeue의 연결은 ②에 해당한다.

NoteBase

문 4. 입력 크기 n에 대한 수행 횟수를 빅오(big-oh) 표기법으로 표현했을 때 옳지 않은 것은?

① $n^4 + 2^n + 5 \rightarrow O(n^4)$

② $3n^2 + 8n + 7 \rightarrow O(n^2)$

③ $2n^2 + 2n\log n + 4n \rightarrow O(n^2)$

④ $3n + 5n\log n + 2 \rightarrow O(n\log n)$

정답 1번

간단한 해설

①에서는 n^4보다 2^n이 더 빠르게 증가한다. **지수항을 다항식보다 낮은 차수로 취급할 수 없으므로**, 전체 수행 횟수는 $O(2^n)$으로 나타내며 $O(n^4)$으로는 상한을 잡을 수 없다.

선지별 해설

빅오 표기 $O(g(n))$는 충분히 큰 입력 크기에서 수행 횟수가 $g(n)$의 어떤 양의 상수배 이하가 된다는 뜻이다. 이 문제처럼 양의 항들을 더한 식은 **가장 빠르게 증가하는 항**을 기준으로 상한을 구할 수 있으며, 그 항의 상수 계수는 생략한다. 로그의 밑은 1보다 큰 고정 상수로 본다.

① X: 2^n은 지수함수이고 n^4은 다항함수이다. 입력 크기가 충분히 커지면 **지수함수가 다항함수보다 빠르게 증가**하므로, 2^n은 n^4에 어떤 고정 상수를 곱한 값도 결국 넘어선다. 따라서 $n^4 + 2^n + 5$를 $O(n^4)$으로 나타낼 수 없으며, $O(2^n)$으로 고쳐야 한다.

② O: $3n^2 + 8n + 7$에서는 이차항 $3n^2$이 일차항 $8n$과 상수항 7보다 빠르게 증가한다. 낮은 차수의 항들을 포함해도 충분히 큰 입력에서는 전체를 n^2의 상수배 이하로 묶을 수 있으므로 $O(n^2)$이다.

③ O: $\log n$은 n보다 느리게 증가하므로, $n\log n$**은 n^2보다 느리게 증가**한다. 따라서 $2n^2 + 2n\log n + 4n$에서는 이차항이 증가율을 결정하며, 전체 수행 횟수를 $O(n^2)$으로 나타낼 수 있다.

④ O: $n\log n$은 일차항 n과 상수항보다 빠르게 증가한다. 따라서 $3n + 5n\log n + 2$에서는 $5n\log n$**이 증가율을 결정**하며, 상수 계수 5를 생략하면 $O(n\log n)$이다.

NoteBase

문 5. 다음 의사코드(pseudo code)가 설명하는 정렬 알고리즘은?

○ 입력: 크기가 n인 배열 A
○ 출력: 정렬된 배열 A

```
for i = 0 to n-2 {
   min = i
   for j = i+1 to n-1 {
      if(A[j] < A[min])
         min = j
   }
   A[i] ↔ A[min]
}
return A
```

① 버블 정렬(bubble sort)
② 삽입 정렬(insertion sort)
③ 선택 정렬(selection sort)
④ 퀵 정렬(quick sort)

정답 3번

간단한 해설

미정렬 구간 전체에서 **최솟값의 위치를 찾은 뒤 맨 앞 원소와 교환**하는 방식은 선택 정렬이다. 내부 반복문에서는 위치만 갱신하고, 탐색이 끝난 뒤 원소를 교환한다.

NoteBase

공통 해설

선택 정렬은 아직 정렬되지 않은 구간에서 **가장 작은 원소를 선택하여 그 구간의 맨 앞에 놓는** 과정을 반복하는 정렬이다. 이 과정을 거칠 때마다 앞에서부터 한 원소씩 오름차순 위치가 확정된다.

주어진 코드에서 i는 이번에 위치를 확정할 자리이고, min은 현재까지 찾은 최솟값의 인덱스이다. 먼저 min = i로 두고, j를 i+1부터 n−1까지 이동시키며 나머지 원소를 확인한다. A[j] < A[min]이면 min = j로 갱신하므로, 내부 반복문이 끝났을 때 min에는 미정렬 구간의 최솟값 위치가 남는다.

이후 A[i]와 A[min]을 교환하여 최솟값을 i 위치에 놓는다. 핵심은 **탐색 중에는 인덱스만 바꾸고, 탐색이 끝난 뒤 교환한다**는 점이다. i를 0부터 n−2까지 증가시키며 이 작업을 반복하면 마지막 원소도 제자리에 남으므로 배열 전체가 정렬된다. 따라서 이 의사코드는 선택 정렬의 동작을 나타낸다.

예를 들어 배열이 [4, 2, 3]이면 첫 탐색에서 2의 위치를 찾아 맨 앞의 4와 교환하여 [2, 4, 3]이 된다. 다음에는 남은 [4, 3]에서 3을 선택하여 교환하므로 [2, 3, 4]가 된다.

NoteBase

문 6. 다음 fib() 함수는 피보나치 수열을 계산한다. fib(6)을 실행할 때, fib() 함수의 호출 횟수는? (단, fib(6)의 호출은 제외한다)

```
int fib(int n)
{
    if(n == 0) return 0;
    if(n == 1) return 1;
    return ( fib(n-1) + fib(n-2) );
}
```

① 21번
② 22번
③ 23번
④ 24번

정답 4번

간단한 해설

재귀가 끝나는 fib(0), fib(1)도 호출 횟수에 포함하며, 같은 인수로 다시 호출되면 매번 센다. 전체 호출은 25번이지만 **최초 fib(6) 호출 1번을 제외**하므로 24번이다.

NoteBase

계산 과정

호출 횟수의 기준: $T(n)$을 fib(n) 실행 시 **자기 자신을 포함한 전체 호출 횟수**라고 하자. n이 0 또는 1이면 바로 반환하므로 추가 재귀 호출은 없지만, 그 함수 자체는 한 번 호출된다. 따라서 $T(0) = T(1) = 1$이다.

점화식: n이 2 이상이면 fib(n)은 fib(n−1)과 fib(n−2)를 각각 호출한다. 두 호출에서 발생하는 모든 호출에 현재 호출 1번을 더하므로 $T(n) = 1 + T(n-1) + T(n-2)$이다. 이 코드에는 계산 결과를 저장해 재사용하는 처리가 없으므로, **같은 인수의 호출도 발생할 때마다 별도로 센다**.

호출 횟수 계산: 작은 인수부터 구하면 다음과 같다. 예를 들어 fib(2)는 자기 자신, fib(1), fib(0)을 합쳐 3번의 호출을 만든다.
$T(2) = 1 + 1 + 1 = 3$
$T(3) = 1 + 3 + 1 = 5$
$T(4) = 1 + 5 + 3 = 9$
$T(5) = 1 + 9 + 5 = 15$
$T(6) = 1 + 15 + 9 = 25$

제외 조건 적용: 25번에는 최초 fib(6) 호출도 들어 있다. 문제에서 제외하는 것은 **최초 호출 1번뿐**이므로, 나머지 호출 횟수는 25 − 1 = 24번이며 ④의 값과 일치한다.

NoteBase

문 7. (가)~(다)에 들어갈 점근 표기법은?

○ $n \geq n_0$인 모든 n에 대해 $c_1 g(n) \leq f(n) \leq c_2 g(n)$을 만족하는 양의 상수 c_1, c_2, n_0가 존재하기만 하면 f(n) = (가)이다.
○ $n \geq n_0$인 모든 n에 대해 $f(n) \leq cg(n)$인 조건을 만족하는 양의 상수 c와 n_0가 존재하기만 하면 f(n) = (나)이다.
○ $n \geq n_0$인 모든 n에 대해 $f(n) \geq cg(n)$을 만족하는 양의 상수 c와 n_0가 존재하기만 하면 f(n) = (다)이다.

① (가): Ω(g(n)), (나): O(g(n)), (다): Θ(g(n))
② (가): Ω(g(n)), (나): Θ(g(n)), (다): O(g(n))
③ (가): Θ(g(n)), (나): O(g(n)), (다): Ω(g(n))
④ (가): Θ(g(n)), (나): Ω(g(n)), (다): O(g(n))

정답 3번

간단한 해설

충분히 큰 입력에서 함수의 값을 **상한과 하한으로 함께 제한**하면 Θ 표기법이다. O는 점근적 상한을, Ω는 점근적 하한을 나타낸다.

빈칸별 해설

점근 표기법은 입력 크기 n이 충분히 커졌을 때 함수의 증가 정도를 나타낸다. 여기서 n_0는 부등식이 계속 성립하기 시작하는 기준이며, 주어진 양의 상수들은 n에 따라 바뀌지 않는다.

(가) $\Theta(g(n))$: 모든 $n \geq n_0$에서 $c_1 g(n) \leq f(n) \leq c_2 g(n)$이므로, $f(n)$은 $g(n)$의 양의 상수배 사이에 놓인다. **상한과 하한을 동시에 만족**하므로 두 함수의 점근적 증가 정도가 같다는 뜻이며, $f(n) = \Theta(g(n))$으로 나타낸다.

(나) $O(g(n))$: 모든 $n \geq n_0$에서 $f(n) \leq cg(n)$이므로, $g(n)$의 양의 상수배가 $f(n)$의 **점근적 상한**이 된다. 이는 위쪽 한계만 정한 조건이므로, $f(n)$이 $g(n)$보다 느리게 증가하는 경우도 포함한다. 따라서 $f(n) = O(g(n))$이다.

(다) $\Omega(g(n))$: 모든 $n \geq n_0$에서 $f(n) \geq cg(n)$이므로, $g(n)$의 양의 상수배가 $f(n)$의 **점근적 하한**이 된다. 이는 아래쪽 한계만 정한 조건이므로, $f(n)$이 $g(n)$보다 빠르게 증가하는 경우도 포함한다. 따라서 $f(n) = \Omega(g(n))$이다.

(가) · (나) · (다)는 각각 $\Theta(g(n))$, $O(g(n))$, $\Omega(g(n))$이므로 ③의 조합과 일치한다.

NoteBase

문 8. 다음 재귀함수 power()는 x^n을 계산한다. 빈칸에 들어갈 내용은?

```
double power(double x, int n)
{
   if (n == 0) return 1;
   return x * [        ]
}
```

① power(x, n);
② power(x, n-1);
③ power(x-1, n);
④ power(x-1, n-1);

정답 2번

간단한 해설

바깥에서 x를 한 번 곱하므로, 재귀호출은 나머지 곱인 x^{n-1}을 계산해야 한다. 따라서 **밑 x는 유지하고 지수 n만 1 줄이는** power(x, n-1)을 호출한다.

빈칸별 해설

이 함수는 n이 0 이상의 정수일 때, n == 0이면 1을 반환하고 그 밖에는 x를 재귀호출 결과에 곱한다. n > 0에서 $x^n = x \cdot x^{n-1}$이므로, 바깥에서 x를 한 번 곱한 뒤 필요한 값은 x^{n-1}이다.

빈칸: power(x, n-1);을 넣는다. **밑 x를 그대로 두고 지수만 1 줄이면** 필요한 거듭제곱을 계산하면서 n이 차례로 감소하여 종료 조건 n == 0에 도달한다. 예를 들어 power(2, 3)은 2 × power(2, 2), 2 × power(2, 1), 2 × power(2, 0)으로 이어진다. 마지막 호출이 1을 반환하면 각 호출의 반환값은 차례로 2, 4, 8이 된다.

power(x, n)은 인수가 변하지 않아 n > 0에서 종료 조건에 도달하지 못한다. power(x-1, n)과 power(x-1, n-1)은 밑을 바꾸므로 같은 x의 거듭제곱을 계산하는 관계를 유지하지 못한다. 필요한 재귀호출 power(x, n-1);은 ②에 해당한다.

NoteBase

문 9. 힙 정렬(heap sort)을 수행하기 위해 다음 데이터를 왼쪽부터 차례대로 하나씩 삽입하여 최소힙(min heap)을 구성하였다. 이후 루트를 한 번 삭제하고 최소힙 특성을 유지하기 위해 재조정한 후, 루트의 왼쪽 자식 노드의 값은?

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

① 3

② 4

③ 5

④ 6

정답 2번

간단한 해설

최소힙에서 루트를 삭제하면 마지막 노드를 루트로 옮긴 뒤 **두 자식 중 더 작은 값**과 교환하며 내려간다. 이 과정에서 9가 2, 4, 6과 차례로 교환되므로, 재조정 후 루트의 왼쪽 자식은 **4**이다.

NoteBase

계산 과정

최소힙은 완전이진트리이며, 각 부모의 값이 자식의 값보다 작거나 같다. 아래 배열은 트리를 위에서 아래로, 같은 높이에서는 왼쪽에서 오른쪽으로 읽은 것이다. 첫 번째 값이 루트이고 두 번째 값이 루트의 왼쪽 자식이다.

1. 순차 삽입: 새 값을 트리의 마지막 자리에 넣고, **부모보다 작으면 부모와 교환**하며 위로 올린다. 주어진 순서대로 삽입한 결과는 다음과 같다.
4 삽입: [4]
6 삽입: [4, 6]
8 삽입: [4, 6, 8]
3 삽입: [3, 4, 8, 6]
7 삽입: [3, 4, 8, 6, 7]
1 삽입: [1, 4, 3, 6, 7, 8]
5 삽입: [1, 4, 3, 6, 7, 8, 5]
2 삽입: [1, 2, 3, 4, 7, 8, 5, 6]
9 삽입: [1, 2, 3, 4, 7, 8, 5, 6, 9]

2. 루트 삭제: 루트의 1을 제거한 뒤, 완전이진트리의 모양을 유지하도록 **마지막 노드의 9를 루트로 이동**한다. 이때 배열은 [9, 2, 3, 4, 7, 8, 5, 6]이다.

3. 아래로 재조정: 9의 두 자식 2와 3 중 작은 2를 올린다. 이어서 9의 두 자식 4와 7 중 작은 4를 올리고, 마지막으로 9의 유일한 자식 6을 올린다. 각 교환 후 배열은 다음과 같다.
2와 교환: [2, 9, 3, 4, 7, 8, 5, 6]
4와 교환: [2, 4, 3, 9, 7, 8, 5, 6]
6과 교환: [2, 4, 3, 6, 7, 8, 5, 9]

재조정 후 루트는 2이고, **루트의 왼쪽 자식은 4**이다. 이는 ②의 값과 일치한다.

NoteBase

문 10. 다음 조건으로 퀵 정렬(quick sort)을 수행할 때, 처음 데이터 교환이 발생하는 배열의 인덱스 쌍은?

○ 데이터를 오름차순으로 정렬한다.
○ low는 왼쪽에서 오른쪽으로 탐색할 때, high는 오른쪽에서 왼쪽으로 탐색할 때 사용되는 변수이다.
○ 정렬할 데이터(A[])는 다음과 같으며, 피벗(pivot)의 초기값은 A[0]이고, low와 high의 초기값은 각각 1과 8이다.

i	0	1	2	3	4	5	6	7	8
A[i]	23	5	91	10	21	35	18	42	78

① 1, 8
② 2, 6
③ 2, 7
④ 3, 8

정답 2번

간단한 해설

피벗 23을 기준으로 왼쪽 탐색은 **인덱스 2의 91**에서, 오른쪽 탐색은 **인덱스 6의 18**에서 멈춘다. 두 값이 피벗을 기준으로 서로 반대쪽에 놓여 있으므로 먼저 교환한다.

NoteBase

계산 과정

초기 조건: 배열은 [23, 5, 91, 10, 21, 35, 18, 42, 78]이고, 피벗은 A[0]의 값인 23이다. low는 인덱스 1에서 오른쪽으로, high는 인덱스 8에서 왼쪽으로 탐색한다. 오름차순 정렬을 위해 피벗보다 작은 값은 왼쪽으로, 큰 값은 오른쪽으로 모은다.

low의 이동: 인덱스 1의 값 5는 피벗 23보다 작으므로 지나간다. 다음 인덱스 2의 값 91은 피벗보다 크므로 **low는 2에서 멈춘다**. 91은 피벗의 오른쪽으로 보내야 할 값이다.

high의 이동: 인덱스 8의 값 78과 인덱스 7의 값 42는 피벗 23보다 크므로 지나간다. 다음 인덱스 6의 값 18은 피벗보다 작으므로 **high는 6에서 멈춘다**. 18은 피벗의 왼쪽으로 보내야 할 값이다.

첫 교환: low = 2, high = 6으로 두 탐색 위치가 아직 교차하지 않았으므로 **A[2]의 91과 A[6]의 18을 교환**한다. 교환 후 배열은 [23, 5, 18, 10, 21, 35, 91, 42, 78]이 된다. 처음 교환하는 인덱스 쌍은 (2, 6)이며, ②의 값과 일치한다.

NoteBase

문 11. 재귀 알고리즘에 대한 설명으로 옳지 않은 것은?

① 재귀호출은 함수가 자기 자신을 호출하는 것이다.

② 재귀함수는 재귀호출이 끝나는 종료 조건이 있어야 한다.

③ 재귀함수로 작성된 병합 정렬(merge sort) 알고리즘은 반복문을 이용하여 구현할 수 있다.

④ 재귀함수는 실행 시간과 메모리 공간 사용 측면에서 반복문보다 효율성이 높다.

정답 4번

간단한 해설

재귀함수는 호출·복귀에 시간이 들고, 일반적으로 호출 상태를 저장할 **추가 스택 공간**이 필요하다. 따라서 실행 시간과 메모리 사용 면에서 **반복문보다 효율적이라고 일반화할 수 없다**.

선지별 해설

① O: 재귀호출은 **함수가 자기 자신을 다시 호출**하는 것이다. 현재 호출의 실행이 끝나기 전에 같은 함수의 새로운 호출이 시작되며, 새 호출이 끝나면 이전 호출의 실행을 이어 간다.

② O: 재귀함수가 정상적으로 끝나려면 **더 이상 재귀호출을 하지 않는 종료 조건**이 있어야 한다. 또한 재귀호출을 거듭하면서 그 조건에 도달해야 한다. 예를 들어 0 이상의 정수를 받아 1씩 줄여 호출한다면, 0에서 추가 호출 없이 반환하도록 하여 재귀를 끝낼 수 있다.

③ O: 병합 정렬은 **작은 구간부터 병합하는 반복문**으로도 구현할 수 있다. 각 원소를 길이 1의 정렬된 구간으로 보고, 이웃한 구간을 병합하여 길이 2, 4, 8의 구간으로 차례로 키운다. 이 과정을 전체 배열이 하나의 정렬된 구간이 될 때까지 반복하므로 재귀호출이 필수는 아니다.

④ X: 일반적인 재귀 구현에서는 함수 호출·복귀 비용이 발생하고, 아직 끝나지 않은 호출의 지역 변수와 복귀 위치 등을 호출 스택에 저장한다. 따라서 재귀 깊이가 깊어질수록 추가 공간이 필요하다. 같은 작업을 반복문으로 구현하면 이러한 재귀호출 비용을 줄일 수 있으므로, 재귀함수가 시간과 공간 면에서 더 효율적이라는 설명은 부적절하다. **효율은 알고리즘과 구현에 따라 달라지며**, 일반적으로 호출 비용과 스택 공간 측면에서는 반복문이 유리하다.

NoteBase

문 12. 동적 계획(dynamic programming) 알고리즘에 대한 설명으로 옳지 않은 것은?

① 최적화 문제에 적용할 수 있는 알고리즘이다.

② 상위 문제의 해를 분할하여 하위 문제의 해를 구한다.

③ 최적해를 구하는 방법을 재귀적으로 정의한다.

④ 한 번 계산된 부분 문제들의 해는 재사용을 위해 저장된다.

정답 2번

간단한 해설

②의 설명은 해를 구하는 방향을 거꾸로 서술했다. 동적 계획은 **하위 문제의 해를 결합하여 상위 문제의 해를 구한다**.

선지별 해설

① O: 동적 계획은 최솟값이나 최댓값을 구하는 최적화 문제에 적용할 수 있다. 전체 문제의 최적해를 부분 문제의 최적해로 구성할 수 있는 **최적 부분 구조**를 이용하며, 반복해서 등장하는 부분 문제의 결과를 저장해 계산을 줄인다.

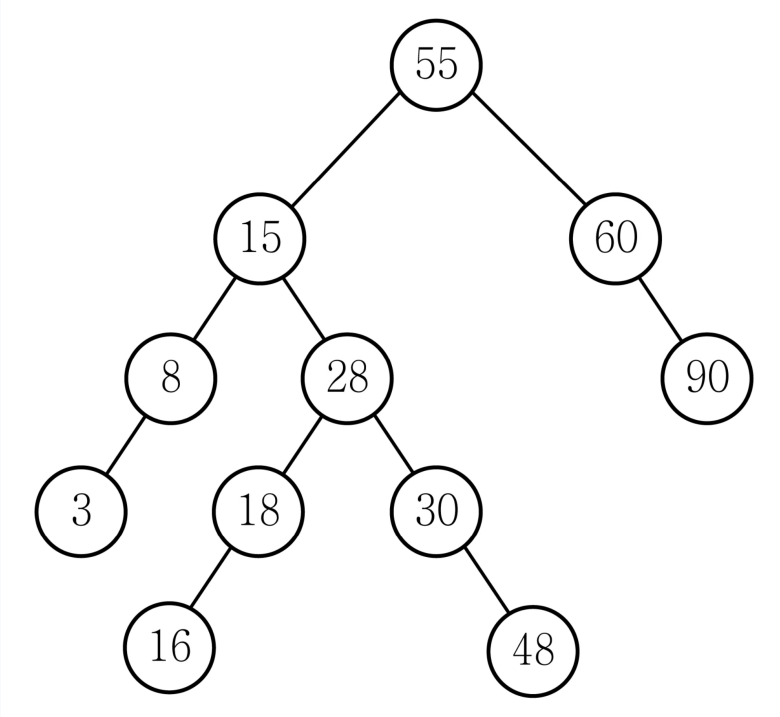
② X: 상위 문제의 해를 분할하여 하위 문제의 해를 구하는 것이 아니라, **하위 문제의 해를 결합하여 상위 문제의 해를 구한다**. 문제를 작은 부분 문제로 나누는 것과 이미 구한 해를 분할하는 것은 다르다. 예를 들어 동전 수를 최소화하는 문제에서는 작은 금액에 필요한 최소 동전 수를 이용해 더 큰 금액에 필요한 최소 동전 수를 구한다.

③ O: 현재 문제의 최적값을 더 작은 부분 문제의 최적값으로 표현하는 점화식을 세워, 최적해를 구하는 방법을 재귀적으로 정의한다. 이때 **재귀적 정의와 재귀 호출 구현은 구별**해야 한다. 점화식이 재귀적으로 정의되어 있어도 실제 계산은 반복문으로 작은 문제부터 진행할 수 있다.

④ O: 한 번 계산한 부분 문제의 해를 배열이나 표 등에 저장하고, 같은 부분 문제가 다시 필요하면 **저장된 결과를 재사용**한다. 이로써 동일한 부분 문제를 반복해서 계산하는 낭비를 줄인다.

NoteBase

문 13. 다음 이진 탐색 트리(binary search tree)에서 키 48을 검색할 때, 방문하는 노드의 순서는?



① 55－15－28－30－48

② 55－60－90－15－28－30－48

③ 55－15－8－3－28－18－16－30－48

④ 55－15－60－8－28－90－3－18－30－16－48

정답 1번

간단한 해설

이진 탐색 트리에서는 찾는 키가 현재 노드보다 **작으면 왼쪽, 크면 오른쪽** 자식으로 이동한다. 48은 55보다 작지만 15·28·30보다 크므로, 각 비교 결과에 따라 한쪽 가지로만 내려가며 검색한다.

NoteBase

올바른 순서

55 → 15 → 28 → 30 → 48

55: 루트 노드에서 검색을 시작한다. 이진 탐색 트리는 각 노드의 왼쪽 서브트리에 더 작은 키를, 오른쪽 서브트리에 더 큰 키를 둔다. 48은 55보다 작으므로 **왼쪽 자식 15**로 이동하며, 60이 있는 오른쪽 서브트리는 검색하지 않는다.

15: 48은 15보다 크므로 **오른쪽 자식 28**로 이동한다. 8이 있는 왼쪽 서브트리의 키는 15보다 작으므로 방문할 필요가 없다.

28: 48은 28보다 크므로 **오른쪽 자식 30**으로 이동한다. 18과 16이 있는 왼쪽 서브트리는 검색 대상에서 제외한다.

30: 48은 30보다 크므로 그림에서 **오른쪽 자식인 48**로 이동한다.

48: 현재 노드의 키가 찾는 키와 **같으므로 검색을 종료**한다.

NoteBase

문 14. 다음 정렬 알고리즘의 평균 시간복잡도를 바르게 연결한 것은?

(가) 선택 정렬 (나) 삽입 정렬
(다) 퀵 정렬 (라) 버블 정렬

① (가): O(n), (나): $O(n^2)$, (다): O(nlogn), (라): $O(n^2)$
② (가): O(n), (나): $O(n^2)$, (다): $O(n^2)$, (라): O(nlogn)
③ (가): $O(n^2)$, (나): O(nlogn), (다): O(n), (라): $O(n^2)$
④ (가): $O(n^2)$, (나): $O(n^2)$, (다): O(nlogn), (라): $O(n^2)$

정답 4번

간단한 해설

퀵 정렬은 분할과 재귀 정렬을 이용하므로 **평균 시간복잡도**가 $O(n\log n)$이다. 선택·삽입·버블 정렬은 평균적으로 비교나 이동 횟수가 자료 수의 제곱에 비례하여 $O(n^2)$이다.

대응 관계 해설

n은 정렬할 자료의 수이다. 각 알고리즘의 일반적인 비교 기반 구현에서 평균 시간복잡도를 연결한다.

(가) 선택 정렬: 정렬되지 않은 부분에서 최솟값을 찾아 맨 앞의 자료와 교환한다. 최솟값을 찾는 비교 횟수는 차례로 $n-1, n-2, \cdots$, 1회이며, 합은 $\frac{n(n-1)}{2}$이다. **입력의 정렬 상태와 관계없이 남은 부분을 탐색**하므로 평균 시간복잡도는 $O(n^2)$이다.

(나) 삽입 정렬: 앞쪽의 정렬된 부분에서 현재 자료가 들어갈 위치를 찾고, 그보다 큰 자료를 뒤로 밀어 삽입한다. 일반적인 평균 분석에서는 정렬된 부분의 길이에 비례하는 비교와 이동이 필요하다. 이를 자료마다 반복하므로 평균 시간복잡도는 $O(n^2)$이다. **이미 정렬된 입력에서의 최선 시간복잡도**인 $O(n)$과 구분한다.

(다) 퀵 정렬: 피벗을 기준으로 자료를 나눈 뒤 각 부분을 재귀적으로 정렬한다. 통상적인 평균 분석에서는 재귀 깊이가 $\log n$에 비례하고, 같은 깊이에서 분할에 드는 전체 작업량은 n에 비례하여 평균 시간복잡도는 $O(n\log n)$이다. **분할이 계속 한쪽으로 치우치는 최악의 경우**에는 $O(n^2)$이지만, 이 문제는 평균을 묻는다.

(라) 버블 정렬: 이웃한 두 자료를 비교하여 순서가 어긋나면 교환하고, 이 과정을 반복한다. 한 차례 순회할 때 가장 큰 자료가 뒤쪽의 제자리에 놓이지만, 나머지 자료를 정렬하려면 여러 차례 더 순회해야 한다. 평균적으로 **자료 수에 비례하는 순회를 반복**하므로 전체 비교·교환 작업의 시간복잡도는 $O(n^2)$이다.

따라서 (가) $O(n^2)$, (나) $O(n^2)$, (다) $O(n\log n)$, (라) $O(n^2)$의 연결이므로 ④와 일치한다.

NoteBase

문 15. (가)에 들어갈 내용은?

○ (가)는 사용되는 문자가 N개일 때, 문자열 검색을 빠르게 실행할 수 있도록 설계된 N진 트리이다.
○ (가)를 이용한 문자열 검색은 루트 노드에서 시작하여 탐색키의 첫 번째 문자에 연관된 링크를 따라 노드를 찾아가고 그 노드에서 다시 두 번째 문자에 연관된 링크를 따라 노드를 찾아가는 과정을 검색이 완료될 때까지 반복한다.

① AVL 트리(AVL tree)
② B+ 트리(B+ tree)
③ 레드-블랙 트리(red-black tree)
④ 트라이(trie)

정답 4번

간단한 해설

트라이는 **문자를 하나씩 따라가는 경로**로 문자열을 저장하고 검색한다. 루트에서 첫 번째 문자에 해당하는 링크를 선택한 뒤 다음 문자에 해당하는 링크를 차례로 따라가는 방식이 지문의 설명과 일치한다.

공통 해설

트라이(trie)는 문자열을 문자 단위로 나누어 저장하는 트리이다. 사용 가능한 문자가 N개이면 각 노드에서 다음 문자에 대응하는 링크를 최대 N개 둘 수 있으므로 N진 트리로 표현한다. 검색할 때는 루트에서 시작하여 **탐색키의 문자를 앞에서부터 하나씩** 읽고, 그 문자에 해당하는 링크를 따라 내려간다. 이 문자별 분기 규칙이 지문의 (가)를 트라이로 식별하는 기준이다.

예를 들어 'cat'과 'car'를 저장하면 루트에서 c, a를 따라가는 경로를 공유하고, 그다음에 t와 r로 갈라진다. 'cat'을 검색할 때는 c → a → t 순서로 이동한다. 필요한 링크가 없으면 검색에 실패하며, 모든 문자를 따라간 뒤에는 해당 위치에 **문자열의 끝을 나타내는 표시**가 있어야 그 문자열이 저장되어 있다고 판단한다.

NoteBase

문 16. 다음에서 설명하는 알고리즘 설계 기법에 해당하지 않는 것은?

문제를 해결할 때 여러 경우 중 하나를 결정해야 할 때마다 현재 순간에 최적이라고 생각되는 것을 선택하면서 문제의 최종해에 도달한다.

① 다익스트라(Dijkstra) 알고리즘
② 프림(Prim) 알고리즘
③ 플로이드-워셜(Floyd-Warshall) 알고리즘
④ 허프만(Huffman) 코딩 알고리즘

정답 3번

간단한 해설

플로이드-워셜 알고리즘은 경유할 수 있는 정점의 범위를 늘리면서 이미 구한 최단 거리를 이용하는 **동적 계획법**이다. 매 순간 가장 유리한 선택을 확정하는 **탐욕법**과 구별된다.

NoteBase

선지별 해설

지문의 설계 기법은 탐욕법이다. 각 단계에서 현재 기준으로 가장 유리한 선택을 확정하며 전체 해를 구성한다. 각 알고리즘이 해를 만드는 방식을 이 기준으로 구분한다.

① 다익스트라 알고리즘 — 탐욕법: 간선 가중치가 음수가 아닌 그래프에서 한 시작점으로부터의 최단 거리를 구한다. 아직 거리를 확정하지 않은 정점 중 **현재까지 알려진 거리가 가장 짧은 정점**을 선택하여 최단 거리를 확정하고, 그 정점을 거치는 경로로 이웃 정점의 거리 정보를 갱신한다.

② 프림 알고리즘 — 탐욕법: 연결된 가중 무방향 그래프에서 모든 정점을 연결하는 간선 가중치의 합이 최소인 신장 트리를 구한다. 현재 트리에 포함된 정점과 포함되지 않은 정점을 잇는 간선 중 **가중치가 가장 작은 간선**을 선택하여 트리를 확장한다.

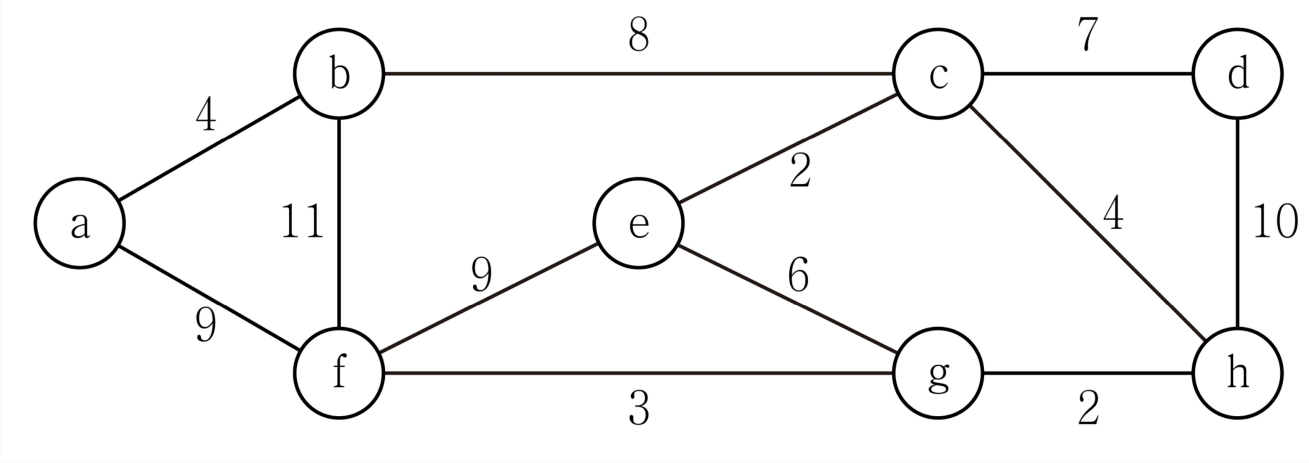
③ 플로이드-워셜 알고리즘 — 동적 계획법: 모든 정점 쌍 사이의 최단 거리를 구한다. 경유를 허용하는 정점을 하나씩 늘리면서, 각 정점 쌍의 기존 거리와 새로 허용한 정점을 거치는 경로의 거리를 비교하여 더 짧은 값으로 갱신한다. 이는 **이전에 구한 부분 문제의 결과를 조합**하는 방식이며, 현재 가장 유리한 선택 하나를 확정하는 탐욕법과 다르다.

④ 허프만 코딩 알고리즘 — 탐욕법: 기호의 출현 빈도를 이용하여 평균 코드 길이가 최소인 접두 부호를 구성한다. 현재 남아 있는 노드 중 **빈도 합이 가장 작은 두 노드**를 선택하여 하나의 노드로 합치고, 새 노드의 빈도를 두 빈도의 합으로 정한다. 이 선택과 결합을 반복하여 부호 트리를 만든다.

다익스트라·프림·허프만 코딩은 탐욕적 선택으로 해를 구성하고, 플로이드-워셜은 부분 문제의 결과를 조합한다. 따라서 지문의 탐욕법에 해당하지 않는 것은 ③의 플로이드-워셜 알고리즘이다.

NoteBase

문 17. 다음 그래프에서 Kruskal 알고리즘을 사용하여 최소 신장 트리(minimum spanning tree)를 찾을 때, 최소 신장 트리의 간선을 나열한 것은?



① (a,b) (a,f) (b,c) (c,d) (c,e) (f,g) (g,h)

② (a,b) (a,f) (b,c) (c,d) (e,g) (f,g) (g,h)

③ (a,b) (b,c) (c,d) (c,e) (c,h) (f,g) (g,h)

④ (a,b) (b,c) (c,d) (c,e) (e,g) (f,g) (g,h)

정답 3번

간단한 해설

Kruskal 알고리즘은 **가중치가 작은 간선부터 사이클을 만들지 않는 것만** 선택한다. (c,e), (g,h), (f,g), (a,b), (c,h)를 선택한 뒤에는 e와 g가 이미 e–c–h–g로 연결되어 있으므로, (e,g)는 제외하고 (c,d), (b,c)를 추가한다.

NoteBase

계산 과정

최소 신장 트리는 모든 정점을 사이클 없이 연결하면서 간선 가중치의 합을 최소로 하는 트리이다. 정점이 8개이므로 간선 8 − 1 = 7개를 선택한다. Kruskal 알고리즘에서는 처음에 각 정점을 별개의 연결 요소로 두고, 간선을 가중치 오름차순으로 검토하여 **서로 다른 연결 요소를 잇는 간선만** 추가한다. 같은 연결 요소 안의 두 정점을 이으면 기존 경로와 함께 사이클이 생긴다.

가중치 2: (c,e)와 (g,h)를 모두 선택한다. 각각 떨어져 있던 정점 두 개를 연결하므로 사이클이 생기지 않는다. 연결 요소는 {c,e}, {g,h}, {a}, {b}, {d}, {f}가 된다.

가중치 3: (f,g)를 선택한다. f를 {g,h}에 연결하여 {f,g,h}를 만든다.

가중치 4: (a,b)와 (c,h)를 모두 선택한다. (a,b)는 a와 b를 연결하고, (c,h)는 {c,e}와 {f,g,h}를 합친다. 두 간선의 검토 순서와 관계없이 사이클이 생기지 않으며, 연결 요소는 {a,b}, {c,e,f,g,h}, {d}가 된다.

가중치 6: (e,g)는 제외한다. e와 g 사이에는 이미 e–c–h–g 경로가 있으므로 이 간선을 추가하면 **e–c–h–g–e 사이클**이 생긴다.

가중치 7: (c,d)를 선택한다. 아직 따로 있던 d를 {c,e,f,g,h}에 연결한다.

가중치 8: (b,c)를 선택한다. 남아 있던 두 연결 요소 {a,b}와 {c,d,e,f,g,h}를 합쳐 모든 정점이 연결된다. 이때 선택한 간선이 7개이므로 종료하며, 가중치 9 이상인 나머지 간선은 추가하지 않는다.

선택된 간선은 (a,b), (b,c), (c,d), (c,e), (c,h), (f,g), (g,h)로, ③의 간선 집합과 일치한다. 총 가중치는 4 + 8 + 7 + 2 + 4 + 3 + 2 = 30이다.

NoteBase

문 18. 알고리즘 설계 기법 중 분할정복(divide-and-conquer)에 대한 설명으로 옳지 않은 것은?

① 이진 탐색(binary search)을 위해 분할정복을 적용할 수 없다.

② 분할정복을 사용한 대표적인 정렬 방법에는 병합 정렬(merge sort)이 있다.

③ 문제를 작은 문제로 분할하고 그 문제들의 해를 병합한다.

④ 분할된 문제들이 서로 중첩되지 않는 경우에 적합하다.

정답 1번

간단한 해설

이진 탐색은 **분할정복을 적용할 수 있는** 탐색 방법이다. 정렬된 자료의 중앙값과 찾는 값을 비교하여, 찾는 값이 있을 수 있는 절반의 구간에서 같은 탐색을 반복한다.

선지별 해설

① X: 이진 탐색은 **탐색 구간을 절반으로 줄여 같은 문제를 푸는** 분할정복의 사례이다. 자료가 오름차순으로 정렬되어 있다면, 찾는 값이 중앙값보다 작을 때는 왼쪽 구간만, 클 때는 오른쪽 구간만 탐색한다. 중앙값과 같으면 탐색을 끝낸다. 분할한 양쪽 구간을 모두 풀 필요는 없으므로, '분할정복을 적용할 수 없다'는 '분할정복을 적용할 수 있다'로 고쳐야 한다.

② O: 병합 정렬은 자료를 두 부분으로 나누고, 각 부분을 재귀적으로 정렬한 뒤 **정렬된 두 부분을 병합**한다. 예를 들어 [3, 1]과 [4, 2]를 각각 [1, 3]과 [2, 4]로 정렬한 다음, 두 부분의 앞쪽 값부터 비교해 작은 값을 꺼내면 [1, 2, 3, 4]가 된다. 문제의 분할, 부분 문제의 해결, 결과의 결합이 드러나는 대표적인 분할정복 알고리즘이다.

③ O: 분할정복은 원래 문제를 더 작은 같은 형태의 문제로 나누고, 각 부분 문제를 해결한 뒤 그 결과를 결합하여 원래 문제의 해를 구한다. 더 나눌 필요가 없는 작은 문제는 직접 해결한다. 다만 **결합 단계가 항상 복잡한 병합 연산일 필요는 없다**. 이진 탐색처럼 한쪽 부분 문제에서 얻은 탐색 결과를 그대로 반환할 수도 있다.

④ O: 분할정복은 부분 문제들이 서로 독립적이어서 **같은 부분 문제를 반복해서 풀 필요가 적을 때** 적합하다. 반대로 동일한 부분 문제가 반복해서 나타나면 중복 계산이 발생할 수 있으므로, 부분 문제의 해를 저장해 재사용하는 동적 계획법을 고려할 수 있다.

NoteBase

문 19. 입력으로 길이 n의 텍스트 문자열(T)과 길이 m의 패턴 문자열(P)이 있을 때, 문자열 매칭(string matching) 알고리즘에 대한 설명으로 옳지 않은 것은?

① 브루트-포스(brute-force) 방식의 수행 시간은 $O(mn)$이다.

② 라빈-카프(Rabin-Karp) 알고리즘은 P의 해시(hash) 값을 이용한다.

③ 보이어-무어(Boyer-Moore) 알고리즘은 P의 각 문자를 왼쪽에서 오른쪽으로 스캔하면서 T와 비교한다.

④ KMP(Knuth-Morris-Pratt) 알고리즘은 P의 각 문자에 대해 매칭 실패 시 비교를 다시 시작할 위치를 계산해 놓는다.

정답 3번

간단한 해설

보이어-무어 알고리즘은 패턴의 문자를 **오른쪽에서 왼쪽으로** 비교한다. 텍스트에서 패턴을 오른쪽으로 이동시키는 방향과, 한 위치에서 패턴 문자를 비교하는 방향은 구별해야 한다.

선지별 해설

① O: 브루트-포스 방식은 텍스트의 시작 위치를 한 칸씩 옮기며 패턴과 차례로 비교한다. 각 시작 위치에서 최대 m개의 문자를 비교하므로, 길이 n의 텍스트에 대한 **최악의 수행 시간**은 $O(mn)$이다.

② O: 라빈-카프 알고리즘은 **패턴 P와 길이가 같은 텍스트 구간의 해시 값**을 비교한다. 해시 값이 다르면 그 구간은 일치하지 않는다. 해시 값이 같더라도 서로 다른 문자열에서 같은 값이 나오는 해시 충돌이 가능하므로, 실제 문자도 비교하여 일치 여부를 확인한다.

③ X: 보이어-무어 알고리즘은 패턴 P의 **오른쪽 끝 문자부터 왼쪽으로** 텍스트 T의 대응 문자와 비교한다. 불일치가 생기면 불일치 문자나 이미 일치한 접미사에 관한 정보를 이용하여 패턴을 오른쪽으로 이동시킨다. 따라서 '왼쪽에서 오른쪽으로 스캔하면서'는 '오른쪽에서 왼쪽으로 스캔하면서'로 고쳐야 한다.

④ O: KMP 알고리즘은 패턴을 전처리하여 **불일치 시 이어서 비교할 패턴 위치**를 실패 함수 등의 형태로 계산해 둔다. 이미 일치한 부분의 접두사와 접미사 관계를 이용하므로, 텍스트의 비교 위치를 뒤로 되돌리지 않고 패턴의 비교 위치를 조정할 수 있다.

NoteBase

문 20. 백트래킹(backtracking)에 대한 설명으로 옳은 것만을 모두 고르면?

ㄱ. 너비 우선 탐색(breadth first search) 방식을 기반으로 한다.
ㄴ. 문제를 해결하는 과정에서 해를 더 이상 얻지 못하는 상황이 되면 직전 상황으로 되돌아가서 다시 해를 탐색하는 기법이다.
ㄷ. 적용할 수 있는 대표적인 문제로는 미로 찾기, N-여왕 문제 등이 있다.

① ㄱ
② ㄴ
③ ㄴ, ㄷ
④ ㄱ, ㄴ, ㄷ

정답 3번

간단한 해설

백트래킹은 한 경로를 따라 탐색하다가 해로 이어질 수 없으면 **이전 선택으로 되돌아가 다른 후보를 시도**하는 기법이다. 너비 우선 탐색이 아니라 **깊이 우선 탐색**을 바탕으로 한다.

보기별 해설

ㄱ. X: 백트래킹은 **깊이 우선 탐색**을 바탕으로 한다. 하나의 선택을 한 뒤 그 선택에서 이어지는 후보를 깊이 탐색하고, 해를 만들 수 없으면 되돌아간다. 시작점에서 가까운 깊이의 후보부터 차례로 탐색하는 너비 우선 탐색과 구별된다.

ㄴ. O: 현재까지의 선택으로 해를 더 이상 구할 수 없으면 **최근 선택을 취소**하고 이전 상태에서 다른 후보를 탐색한다. 이전 상태에서도 시도할 후보가 없으면 더 앞선 상태까지 되돌아간다. 이처럼 해가 될 수 없는 경로의 탐색을 중단하여 불필요한 탐색을 줄인다.

ㄷ. O: 미로 찾기에서는 막다른 길에 도달하면 갈림길로 돌아와 다른 길을 시도할 수 있다. N-여왕 문제에서는 N×N 체스판에 N개의 여왕을 서로 공격하지 않도록 배치한다. 한 행씩 여왕을 놓다가 **같은 열이나 대각선에 놓이지 않는 자리**가 없으면 앞서 놓은 여왕을 옮겨 다른 배치를 탐색한다. 두 문제 모두 선택과 되돌리기를 반복하는 백트래킹을 적용할 수 있다.

옳은 보기는 ㄴ, ㄷ이며, 이를 모두 포함한 선택지는 ③이다.

NoteBase