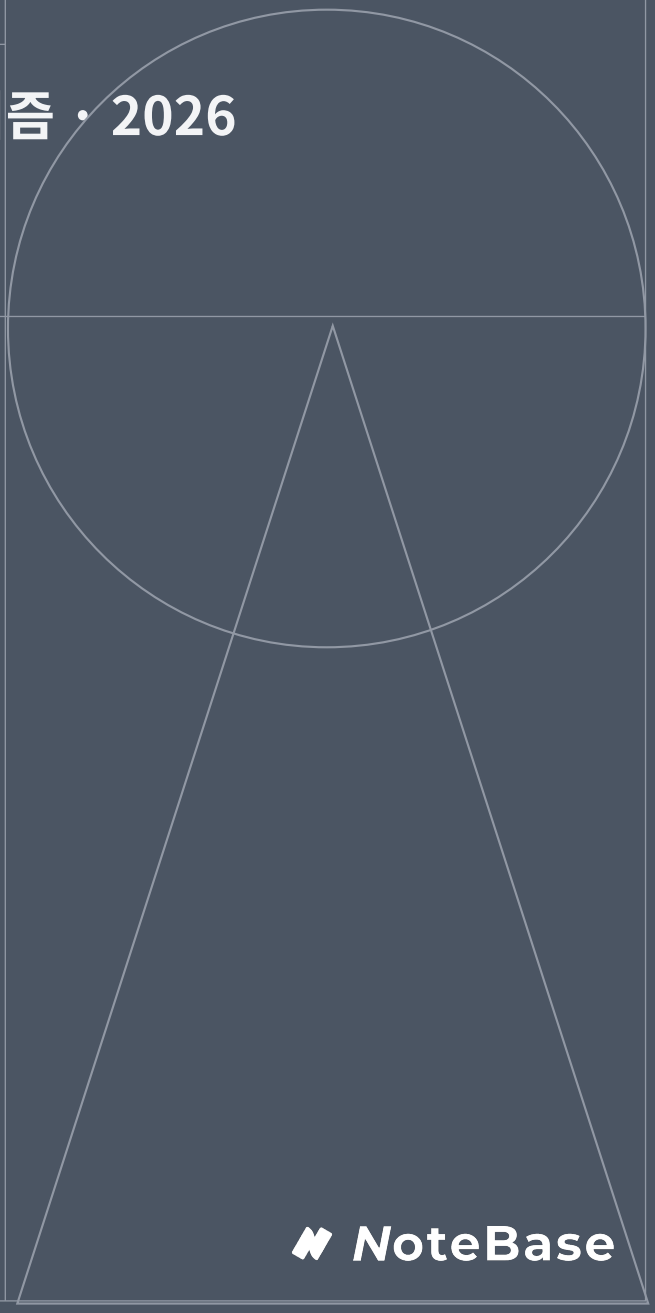
# 기출OX 해설집

공무원/국가직/9급 · 알고리즘 · 2026



NoteBase

# 기출OX 해설집

## 공무원/국가직/9급 · 알고리즘 · 2026

### 공무원/국가직/9급 · 알고리즘 · 2026

**문 1. 다음은 1 이상인 x에 대해 1부터 x까지의 합을 계산하는 C 함수이다. (가)에 들어갈 코드는?**

```
int sum(int x)
{
    if (x <= 1) return 1;
    return x + (가);
}
```

① x + 1

② x - 1

③ sum(x + 1)

④ sum(x - 1)

**정답 4번**

**간단한 해설**

1부터 x까지의 합은 x에 **1부터 x - 1까지의 합**을 더한 값이다. 따라서 남은 합을 구하는 재귀 호출 **sum(x - 1)**이 필요하다.

NoteBase

**빈칸별 해설**

(가): 함수 sum은 1부터 입력값까지의 합을 반환해야 한다. 현재 값 x는 반환식에서 이미 더하므로, 빈칸에는 **1부터 x - 1까지의 합**을 구하는 'sum(x - 1)'이 들어간다. 이는 $\text{sum}(x) = x + \text{sum}(x - 1)$이라는 재귀 관계를 구현한다.

입력값이 1이면 'if (x <= 1) return 1;'에서 1을 반환하고 종료한다. 입력값이 1보다 크면 재귀 호출마다 입력값이 1씩 줄어 **종료 조건에 도달**한다. 예를 들어 x = 3이면 입력값이 3 → 2 → 1로 줄어든다. 입력값 1에서 반환한 1을 이용해 입력값 2에서는 2 + 1 = 3을, 입력값 3에서는 3 + 3 = 6을 반환한다.

'x + 1'이나 'x - 1'은 단순한 산술식이므로 나머지 구간의 합을 구하지 못한다. 'sum(x + 1)'은 입력값을 늘려 1 이하라는 종료 조건에서 멀어진다. 따라서 필요한 코드는 'sum(x - 1)'이며, ④에 해당한다.

NoteBase

**문 2. 다음 설명에 해당하는 알고리즘은?**

◯ 문자열 매칭 알고리즘이다.
◯ 주어진 패턴과 텍스트에서 사용된 알파벳을 이용해 불일치 문자(bad character) 이동표를 만든다.
◯ 패턴을 이용해서 일치 접미부(good suffix) 이동표를 만든다.

① 다익스트라(Dijkstra) 알고리즘
② 라빈-카프(Rabin-Karp) 알고리즘
③ 보이어-무어(Boyer-Moore) 알고리즘
④ 플로이드-워셜(Floyd-Warshall) 알고리즘

**정답 3번**

**간단한 해설**

**불일치 문자 이동표와 일치 접미부 이동표**를 함께 사용하는 것이 보이어-무어 알고리즘의 특징이다. 비교 중 불일치가 발생하면 두 규칙으로 구한 이동량 중 큰 값을 이용해 패턴을 이동한다.

**공통 해설**

문자열 매칭은 텍스트에서 주어진 패턴과 같은 문자열을 찾는 작업이다. 보이어-무어(Boyer-Moore) 알고리즘은 패턴을 텍스트의 일부와 맞춰 놓고 **패턴의 오른쪽 끝부터 왼쪽으로** 문자를 비교한다. 불일치가 생기면 미리 만든 두 이동표를 이용해 다음 비교 위치를 정한다.

불일치 문자(bad character) 이동표는 불일치가 발생한 텍스트 문자와 패턴 안에서 그 문자가 나타나는 위치를 바탕으로 이동량을 정하는 데 쓰인다. 일치 접미부(good suffix) 이동표는 불일치가 나기 전에 이미 일치한 패턴 뒤쪽 부분을 활용한다. 이 부분이 다시 일치할 가능성이 있는 위치를 패턴에서 미리 분석하여 이동량을 정한다.

두 규칙으로 구한 이동량 중 **큰 값**을 택하면, 일치가 불가능한 위치를 건너뛰고 검색을 이어 갈 수 있다. 지문처럼 불일치 문자와 일치 접미부에 대한 이동표를 함께 준비하는 방식이 보이어-무어 알고리즘을 식별하는 기준이다.

NoteBase

**문 3. 분할 정복(divide and conquer) 방식의 정렬 알고리즘만을 모두 고르면?**

ㄱ. 버블 정렬(bubble sort)
ㄴ. 병합 정렬(merge sort)
ㄷ. 퀵 정렬(quick sort)

① ㄴ
② ㄱ, ㄴ
③ ㄱ, ㄷ
④ ㄴ, ㄷ

**정답 4번**

**간단한 해설**

병합 정렬과 퀵 정렬은 모두 정렬할 자료를 **작은 부분 문제로 나누어 재귀적으로 해결**하는 분할 정복 방식이다. 병합 정렬은 나눈 자료를 정렬한 뒤 병합하고, 퀵 정렬은 피벗을 기준으로 분할한 뒤 각 부분을 정렬한다.

**보기별 해설**

분할 정복은 큰 문제를 같은 형태의 작은 부분 문제들로 나누고, 각 부분 문제를 해결한 뒤 그 결과로 전체 문제를 해결하는 방식이다. 정렬에서는 나눈 각 부분에 같은 정렬 방법을 반복 적용하고, 원소가 하나 이하이면 더 나누지 않는다.

ㄱ. 버블 정렬: 분할 정복 방식에 해당하지 않는다. **인접한 두 원소를 비교하여 순서가 잘못되어 있으면 교환**하는 과정을 반복한다. 한 차례의 순회로 큰 원소를 뒤쪽에 확정하지만, 자료를 여러 부분 문제로 나누어 각각 재귀적으로 정렬하는 방식은 아니다.

ㄴ. 병합 정렬: 분할 정복 방식이다. 자료를 두 부분으로 나누고, 각 부분에 병합 정렬을 재귀적으로 적용한다. 이후 **정렬된 두 부분을 병합**하면서 전체의 순서를 맞춘다. 예를 들어 두 부분이 각각 1, 4와 2, 3으로 정렬되어 있다면, 앞쪽 원소부터 비교하여 1, 2, 3, 4로 합친다.

ㄷ. 퀵 정렬: 분할 정복 방식이다. 피벗을 기준으로 작은 값과 큰 값을 양쪽으로 분할하고, **분할된 각 부분에 퀵 정렬을 재귀적으로 적용**한다. 분할 단계에서 두 부분 사이의 크기 관계가 정해지므로, 병합 정렬처럼 정렬된 두 부분을 별도로 병합하는 과정은 필요하지 않다.

분할 정복 방식에 해당하는 것은 ㄴ과 ㄷ이며, 이 조합은 ④에 해당한다.

NoteBase

**문 4. 빅오($O$) 표기법에 대한 설명으로 옳지 않은 것은?**

① $O(n\log n)$은 2를 포함한다.

② $O(n\log n)$은 $3n\log n + 2n + 1$을 포함한다.

③ $O(n\log n)$은 $3n + \log n + 2$를 포함하지 않는다.

④ $O(n\log n)$은 $n^2 + 2n\log n + 3$을 포함하지 않는다.

**정답 3번**

**간단한 해설**

③의 $3n + \log n + 2$는 선형으로 증가하므로 $n\log n$의 상수배로 상한을 정할 수 있다. 빅오는 **점근적 상한**을 나타내므로, 더 느리게 증가하는 이 함수도 $O(n\log n)$에 포함된다.

**선지별 해설**

$O(n\log n)$은 충분히 큰 모든 $n$에 대해 $0 \le f(n) \le c \cdot n\log n$을 만족시키는 양의 상수 $c$가 존재하는 함수들의 집합이다. **증가율이 정확히 같을 필요는 없다.** 로그의 밑은 1보다 큰 고정 상수로 본다.

① O: 2는 입력 크기와 관계없이 일정한 상수이다. $n\log n$은 계속 커지므로 충분히 큰 $n$에서는 $2 \le n\log n$이 성립한다. 따라서 상수 함수 2도 $O(n\log n)$에 포함된다.

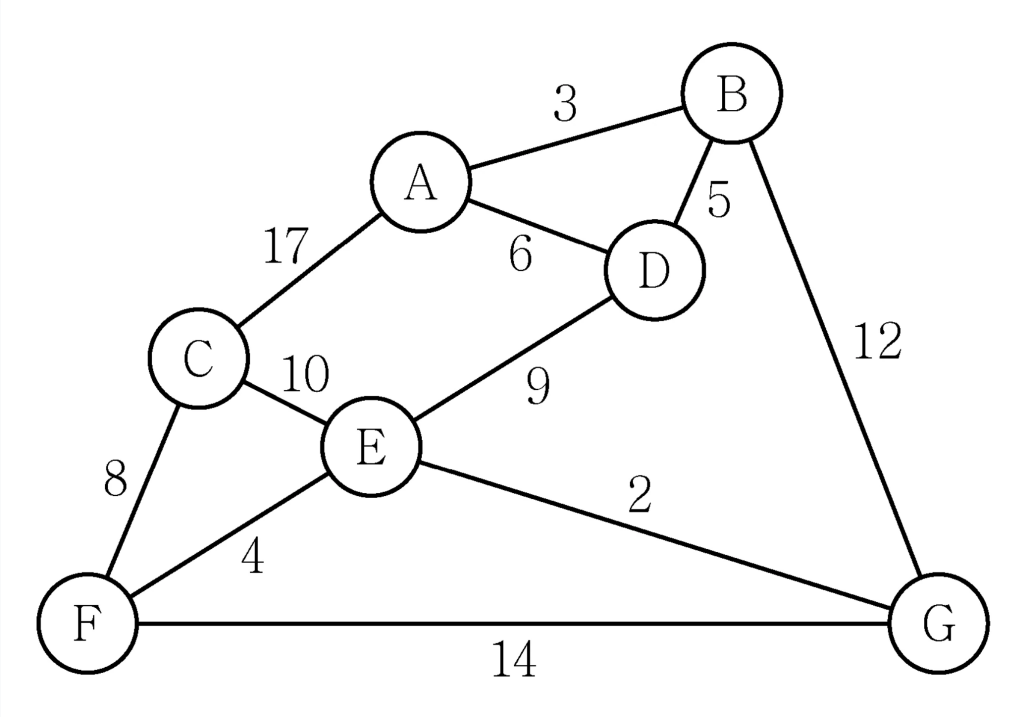
② O: $3n\log n + 2n + 1$에서 가장 빠르게 증가하는 항은 $3n\log n$이다. 나머지 $2n + 1$은 그보다 느리게 증가하므로, 전체를 $n\log n$의 일정한 상수배 이하로 묶을 수 있다. **상수 계수와 더 느리게 증가하는 항**은 이 상한을 바꾸지 않는다.

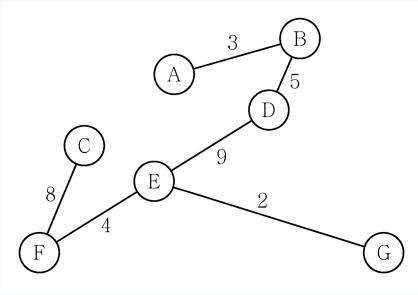
③ X: $3n + \log n + 2$에서 가장 빠르게 증가하는 항은 $3n$이므로 이 함수는 $O(n)$에 속한다. 충분히 큰 $n$에서는 $\log n \ge 1$이어서 $n \le n\log n$이 성립하므로, 선형 함수의 상한을 $n\log n$의 상수배로도 정할 수 있다. 따라서 선지의 표현은 **포함하지 않는다가 아니라 포함한다**로 고쳐야 한다.

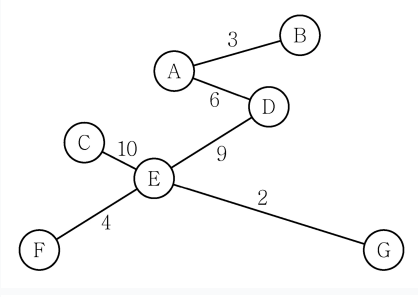
④ O: $n^2 + 2n\log n + 3$은 $n^2$ 이상이다. 그런데 $\frac{n^2}{n\log n} = \frac{n}{\log n}$은 $n$이 커질수록 한없이 커진다. 따라서 **어떤 고정된 상수배로도 상한을 정할 수 없어** 이 함수는 $O(n\log n)$에 포함되지 않는다.

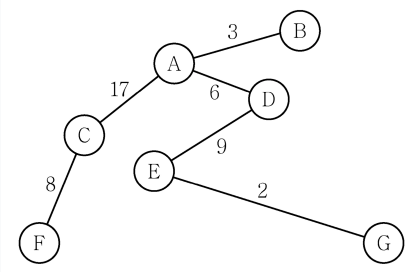
NoteBase

**문 5. 다음 그래프에서 크루스칼(Kruskal) 알고리즘을 사용하여 만든 최소 비용 신장 트리(minimum cost spanning tree)는?**



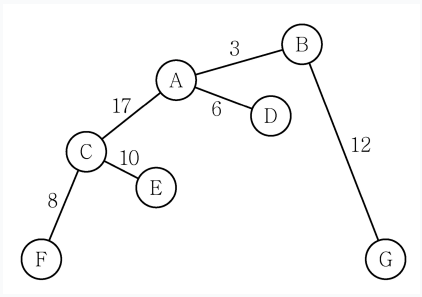

① 


② 


③ 


NoteBase

④




**정답 1번**

**간단한 해설**

크루스칼 알고리즘은 가중치가 작은 간선부터 선택하되 **사이클을 만드는 간선은 제외**한다. A–D(6)는 A–B–D가 이미 연결되어 있어 제외하고, D–E(9)로 두 연결 묶음을 합치면 총비용은 31이다.

**계산 과정**

풀이 기준: 최소 비용 신장 트리는 모든 정점을 사이클 없이 연결하면서 간선 가중치의 합을 최소로 하는 트리이다. 크루스칼 알고리즘은 전체 간선을 가중치의 오름차순으로 검토하고, **서로 다른 연결 묶음을 잇는 간선만 선택**한다. 정점이 7개이므로 간선 7 − 1 = 6개가 선택되면 완성된다.

작은 간선부터 선택: E–G(2), A–B(3), E–F(4), B–D(5)를 차례로 선택한다. E–F는 F를 E·G 묶음에, B–D는 D를 A·B 묶음에 추가하므로 사이클이 생기지 않는다. 이때 연결 묶음은 {E, F, G}, {A, B, D}, {C}이다.

A–D(6) 제외: A와 D는 이미 A–B–D 경로로 연결되어 있다. 여기에 A–D를 추가하면 **A–B–D–A의 사이클**이 생기므로 선택하지 않는다. 가중치가 작더라도 사이클을 만들면 제외해야 한다.

C–F(8) 선택: C는 아직 어느 다른 정점과도 연결되지 않았으므로 C–F를 선택한다. 이제 연결 묶음은 {C, E, F, G}와 {A, B, D} 두 개가 된다.

D–E(9) 선택: D와 E는 서로 다른 연결 묶음에 속하므로 D–E를 추가해도 사이클이 생기지 않는다. 이 간선으로 **두 묶음이 하나로 합쳐져** 모든 정점이 연결되고, 선택한 간선도 6개가 되어 종료한다. 더 큰 가중치의 C–E(10), B–G(12), F–G(14), A–C(17)는 추가하지 않는다.

결과: 선택한 간선은 E–G, A–B, E–F, B–D, C–F, D–E이며, 총비용은 2 + 3 + 4 + 5 + 8 + 9 = 31이다. 이 여섯 간선을 나타낸 그림이 ①이다.

NoteBase

**문 6. 충분히 큰 $n$에 대해서 수행 시간이 가장 많이 걸리는 시간 복잡도는?**

① $\Theta(2^n)$

② $\Theta(n^3)$

③ $\Theta(n!)$

④ $\Theta(n\log n)$

**정답 3번**

**간단한 해설**

팩토리얼 $n!$은 입력 크기가 1 증가할 때마다 점점 커지는 수를 곱한다. 따라서 충분히 큰 입력에서는 **고정된 배율로 증가하는 지수함수보다도 빠르게 증가**한다.

**선지별 해설**

$n$은 입력 크기이며, $\Theta(f(n))$은 충분히 큰 입력에서 수행 시간이 $f(n)$의 상수배 범위로 증가함을 나타낸다. 이 문제에서는 **입력 크기에 따른 증가율**을 기준으로 비교한다.

① $\Theta(2^n)$: 지수 시간이다. 입력 크기가 1 증가할 때마다 $2^n$은 2배가 된다. 충분히 큰 입력에서는 다항함수인 $n^3$보다 빠르게 증가하지만, 팩토리얼인 $n!$보다는 느리게 증가한다.

② $\Theta(n^3)$: 3차 다항 시간이다. 입력 크기가 2배가 되면 $n^3$은 8배가 된다. $n\log n$보다 빠르게 증가하지만, 충분히 큰 입력에서는 **지수함수보다 느리게 증가**한다.

③ $\Theta(n!)$: 팩토리얼 시간이다. $n! = 1 \cdot 2 \cdot \ldots \cdot n$이며, 입력 크기가 1 증가하면 기존 값에 $n+1$을 곱한다. $2^n$의 곱셈 배율은 2로 일정하지만, 팩토리얼의 곱셈 배율은 **입력 크기와 함께 계속 커진다**. 따라서 충분히 큰 입력에서는 $2^n$보다도 빠르게 증가한다.

④ $\Theta(n\log n)$: 입력 크기에 로그 인자를 곱한 시간이다. $\log n$은 $n^2$보다 느리게 증가하므로, $n\log n$은 $n^3$보다 느리게 증가한다. 제시된 후보 중 증가율이 가장 작다.

충분히 큰 입력에서 증가율은 $n\log n < n^3 < 2^n < n!$ 순이다. 따라서 수행 시간이 가장 많이 걸리는 시간 복잡도는 ③의 $\Theta(n!)$이다.

NoteBase

**문 7. 다음 의사코드에 해당하는 알고리즘 설계기법은?**

```
f[ ]: 모든 값이 0으로 초기화된 정수 배열
fib(n)
{
    if (f[n] == 0) {
        if (n == 1 or n == 2) f[n] = 1;
        else f[n] = fib(n - 1) + fib(n - 2);
    }
    return f[n];
}
```

① 그리디(greedy) 알고리즘

② 유전자(genetic) 알고리즘

③ 분기 한정(branch and bound) 기법

④ 동적 프로그래밍(dynamic programming)

**정답 4번**

**간단한 해설**

이미 구한 피보나치 수를 배열에 저장하고, 같은 값이 다시 필요하면 **재계산 없이 재사용**한다. 이는 메모이제이션을 사용하는 동적 프로그래밍의 특징이다.

NoteBase

**공통 해설**

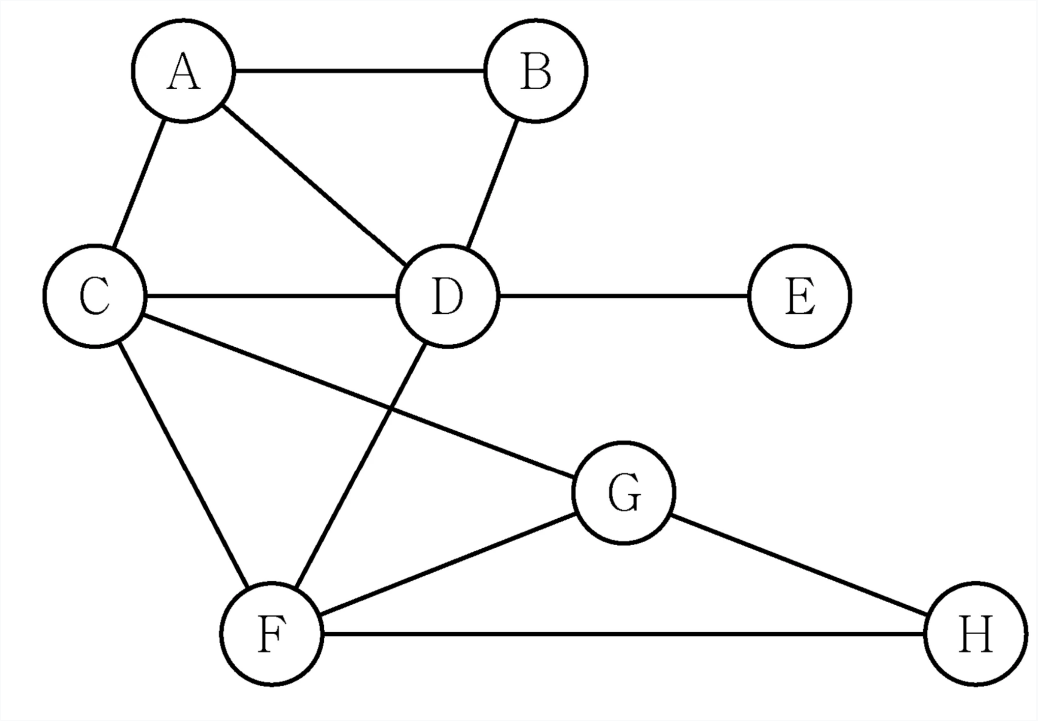
동적 프로그래밍은 문제를 작은 부분 문제로 나누어 풀고, 겹치는 부분 문제의 결과를 저장해 재사용하는 설계기법이다. 이처럼 필요한 부분 문제를 재귀적으로 풀면서 결과를 기억하는 구현을 **메모이제이션**이라고 한다.

의사코드에서 f[n]은 n번째 피보나치 수를 저장한다. 모든 값이 처음에 0이므로, f[n] == 0은 아직 그 값을 계산하지 않았다는 뜻이다. n이 1이나 2이면 1을 저장하고, 그 밖에는 fib($n-1$)과 fib($n-2$)의 합을 저장한다. 이미 값이 저장되어 있으면 조건문 내부를 건너뛰고 **저장된 f[n]을 바로 반환**한다. 따라서 같은 부분 문제의 계산을 반복하지 않는다.

예를 들어 fib(4)를 구하려면 fib(3)과 fib(2)가 필요하다. 이 중 fib(3)을 먼저 구하면서 f[2]에 1을 저장했다면, 이후 fib(2)를 호출할 때는 저장된 1을 바로 돌려준다. 단순히 재귀 호출을 한다는 점보다, 겹치는 부분 문제의 결과를 저장하고 재사용한다는 점에서 동적 프로그래밍에 해당한다.

NoteBase

**문 8. 다음 그래프의 A 정점부터 너비 우선 탐색(BFS, breadth first search)을 할 때, 가능한 정점의 방문 순서가 아닌 것은?**




① A → B → C → D → F → G → E → H

② A → B → D → C → E → F → G → H

③ A → C → B → D → F → G → E → H

④ A → C → D → E → B → F → H → G

**정답 4번**

**간단한 해설**

BFS는 시작점에서 **간선 수가 적은 정점부터** 방문한다. ④에서는 A와 직접 연결된 B보다 두 간선을 거쳐야 하는 E를 먼저 방문하므로 이 원칙에 어긋난다.

NoteBase

**선지별 해설**

너비 우선 탐색(BFS)은 선입선출 큐를 사용한다. A를 처리할 때 인접 정점 B·C·D를 큐에 넣고, 이후 큐 앞의 정점을 꺼내 아직 발견하지 않은 이웃을 큐 뒤에 넣는다. 그림에서 A와의 최단 거리는 B·C·D가 간선 1개, E·F·G가 간선 2개, H가 간선 3개이다. 따라서 가까운 거리의 정점을 모두 방문한 뒤 다음 거리로 넘어간다. 인접 정점을 살펴보는 순서가 지정되지 않았으므로 가능한 방문 순서는 여러 개이다.

① O: A의 이웃을 B·C·D 순서로 큐에 넣는다. B에서는 새로 발견할 정점이 없고, C에서는 F·G를, D에서는 E를 큐에 넣을 수 있다. 따라서 두 번째 거리의 정점은 **F·G·E 순서**가 되고, F를 처리할 때 발견한 H는 그 뒤에 방문하므로 제시된 순서가 가능하다.

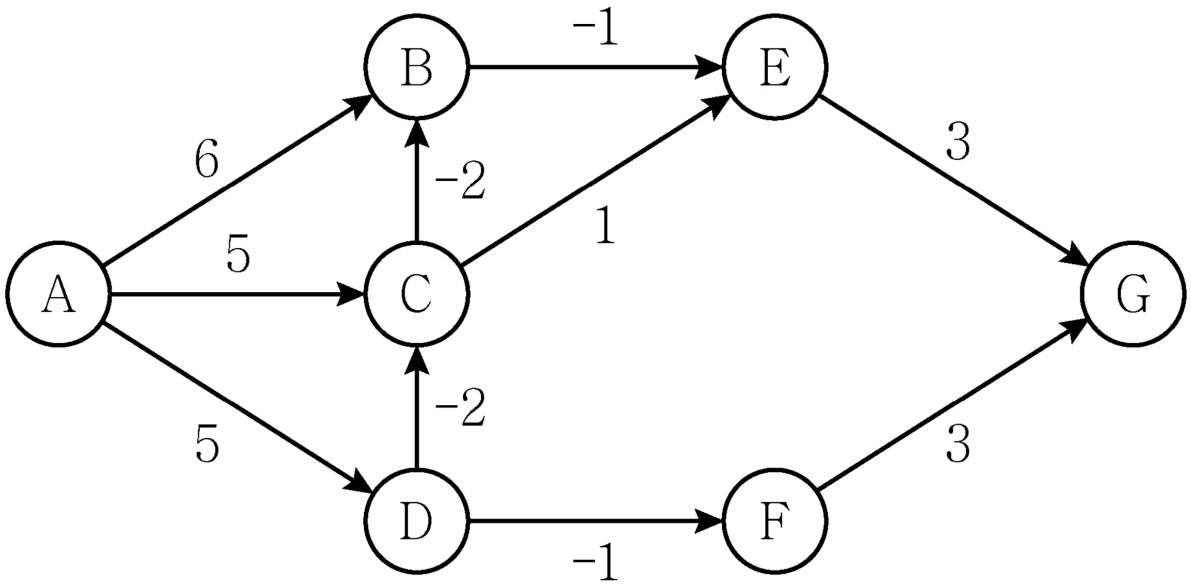
② O: A의 이웃을 B·D·C 순서로 큐에 넣는다. B에서는 새로 발견할 정점이 없고, D에서 E·F를 먼저 넣은 다음 C에서 G를 넣을 수 있다. 따라서 두 번째 거리의 정점은 **E·F·G 순서**가 된다. F에서 발견한 H는 이미 큐에 들어 있는 G 뒤에 놓이므로 제시된 순서가 가능하다.

③ O: A의 이웃을 C·B·D 순서로 큐에 넣는다. C에서 F·G를 먼저 넣고, B에서는 새 정점을 발견하지 않으며, D에서 E를 넣는다. 따라서 **C를 먼저 처리해도 B·D를 F·G보다 먼저 방문**하고, 그 뒤 F·G·E와 H를 방문할 수 있다.

④ X: A를 처리할 때 B·C·D가 모두 큐에 들어가므로 **B를 방문하기 전에 E를 방문할 수 없다**. 또한 C를 D보다 먼저 처리하면 F·G가 E보다 먼저 큐에 들어가며, 거리 3인 H를 거리 2인 G보다 먼저 방문할 수도 없다. 처음의 A → C → D를 유지하려면, 예를 들어 A → C → D → B → F → G → E → H처럼 고칠 수 있다.

NoteBase

**문 9. 다음 방향 그래프에 벨만－포드(Bellman－Ford) 알고리즘을 적용한 후, 각 정점과의 최단 거리 값을 바르게 연결한 것은? (단, 시작 정점은 A 정점이다)**




① A: 0, B: 1, C: 1, D: 5, E: −1, F: −1, G: 3

② A: 0, B: 1, C: 3, D: 5, E: 0, F: 4, G: 3

③ A: 0, B: 3, C: 2, D: 2, E: 3, F: 2, G: 6

④ A: 0, B: 3, C: 5, D: 5, E: 6, F: 4, G: 7

**정답 2번**

**간단한 해설**

직접 연결된 간선만 보지 않고 **음수 가중치를 포함한 우회 경로**도 비교한다. A→D→C→B→E의 거리는 5 − 2 − 2 − 1 = 0이어서, E까지의 최단 거리는 0이고 G까지는 0 + 3 = 3이다.

NoteBase

**계산 과정**

벨만－포드 알고리즘은 각 정점까지 알려진 거리를 두고, 모든 방향 간선을 반복 확인하여 더 짧은 경로가 발견되면 거리를 갱신한다. 이를 '완화'라고 한다. $d(v)$를 A에서 정점 v까지 현재 알려진 거리, $w(u, v)$를 u→v의 가중치라고 하면, $d(u) + w(u, v) < d(v)$일 때 $d(v) = d(u) + w(u, v)$로 갱신한다. 간선은 **화살표 방향으로만** 이용한다.

A와 초기화: 시작 정점 A의 거리는 0이고, 나머지 정점은 아직 경로를 찾지 못했다는 뜻으로 ∞로 초기화한다. 그림에는 A로 들어오는 간선이 없으므로 A의 최종 거리도 0이다.

D: D로 들어오는 간선은 A→D뿐이다. 가중치가 5이므로 A에서 D까지의 최단 거리는 0 + 5 = 5이다.

C: A→C로 바로 가면 거리는 5이다. A→D→C로 가면 5 − 2 = 3이므로 **D를 경유하는 경로**가 더 짧다. C의 최단 거리는 3이다.

B: A→B로 바로 가면 거리는 6이다. C까지의 최단 거리 3에 C→B의 가중치 −2를 더하면 3 − 2 = 1이다. 따라서 A→D→C→B를 이용한 B의 최단 거리는 1이다.

E: E로 들어오는 간선은 B→E와 C→E이다. B를 거치면 1 − 1 = 0이고, C에서 E로 바로 가면 3 + 1 = 4이다. **두 후보 거리 중 작은 값**인 0이 E의 최단 거리이다.

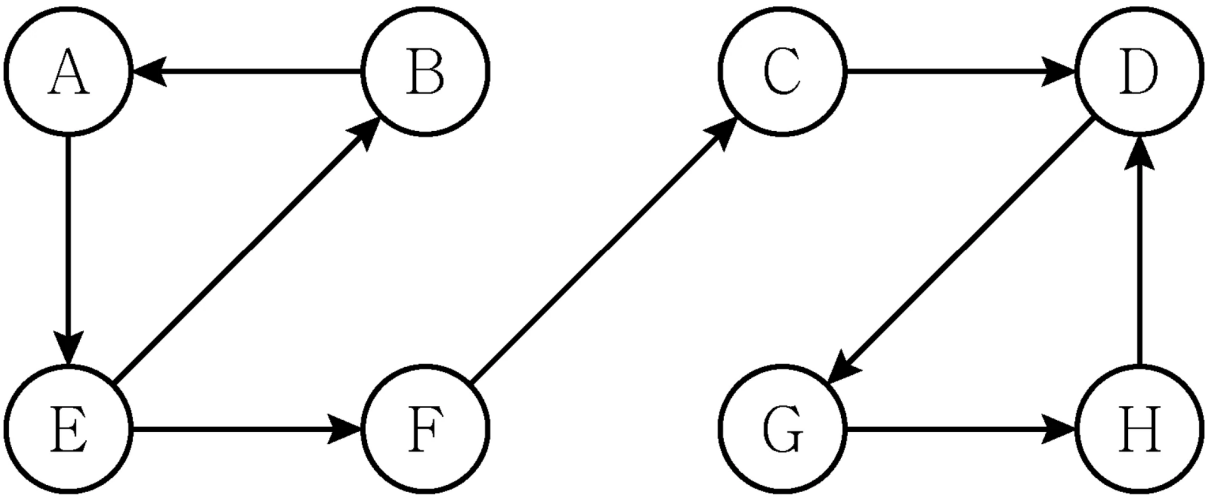
F: F로 들어오는 간선은 D→F뿐이다. D까지의 최단 거리 5에 가중치 −1을 더하면 5 − 1 = 4이므로 F의 최단 거리는 4이다.

G: E를 거치면 0 + 3 = 3이고, F를 거치면 4 + 3 = 7이다. 따라서 E를 거치는 경로가 더 짧으며 G의 최단 거리는 3이다.

최종 최단 거리는 A: 0, B: 1, C: 3, D: 5, E: 0, F: 4, G: 3으로, ②의 값과 일치한다.

NoteBase

**문 10. 다음 방향 그래프에서 2개 이상의 정점이 포함되어 있는 강한 연결 요소(strongly connected component)의 개수는?**




① 1

② 2

③ 3

④ 4

**정답 2번**

**간단한 해설**

화살표를 따라 서로 왕복할 수 있는 정점들은 **{A, B, E}**와 **{D, G, H}**로 묶인다. F와 C는 각각 단독 요소이므로, 정점이 2개 이상인 강한 연결 요소는 2개이다.

NoteBase

**도식 해설**

강한 연결 요소는 방향 간선을 따라 집합 안의 어느 정점에서 출발해도 다른 모든 정점에 도달할 수 있는 최대 정점 집합이다. 여기서 '최대'란 **서로 도달할 수 있는 정점을 더 포함할 수 없다**는 뜻이다. 따라서 같은 요소 안의 일부 정점을 따로 떼어 중복해서 세지 않는다.

{A, B, E}: A → E → B → A의 순환이 있다. 이 순환을 따라 세 정점 사이를 모두 왕복할 수 있으므로 하나의 강한 연결 요소이다.

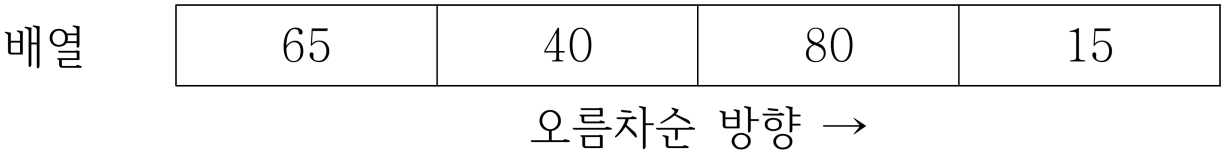
{D, G, H}: D → G → H → D의 순환이 있다. 세 정점이 서로 도달할 수 있으므로 또 하나의 강한 연결 요소이다.

F와 C: 두 순환 사이에는 E → F → C → D로 이어지는 경로가 있다. 그러나 오른쪽의 {D, G, H}에서 C나 F로 돌아가는 경로가 없고, C에서 F로, F에서 {A, B, E}로 돌아갈 수도 없다. **한 방향으로만 도달하는 관계**이므로 이 경로로 연결된 정점들을 하나의 강한 연결 요소로 합칠 수 없다. F와 C는 각각 {F}, {C}라는 단독 요소이다.

전체 강한 연결 요소는 {A, B, E}, {F}, {C}, {D, G, H}이다. 이 중 정점이 2개 이상인 것은 {A, B, E}와 {D, G, H}의 2개이다.

NoteBase

**문 11. 다음 배열에서 버블 정렬 알고리즘을 사용하여 오름차순 정렬했을 때, 자리바꿈의 총횟수는?**




① 3
② 4
③ 5
④ 6

**정답 2번**

**간단한 해설**

버블 정렬은 이웃한 두 값이 오름차순에 어긋날 때만 자리를 바꾼다. 주어진 배열에서는 각 회전에서 2회, 1회, 1회씩 교환하므로 **총 4회**의 자리바꿈이 발생한다.

**계산 과정**

그림의 초기 배열은 [65, 40, 80, 15]이다. 오름차순 버블 정렬에서는 이웃한 두 값을 비교하여 **왼쪽 값이 오른쪽 값보다 클 때만 교환**한다. 왼쪽에서 오른쪽으로 한 번 훑으면 가장 큰 값이 맨 오른쪽에 놓이므로, 다음 회전에서는 그 값을 제외하고 비교한다.

1회전: 65와 40을 교환하면 [40, 65, 80, 15]가 된다. 다음의 65와 80은 이미 오름차순이므로 교환하지 않는다. 이어 80과 15를 교환하면 [40, 65, 15, 80]이 된다. 자리바꿈은 2회이며, 가장 큰 값 80의 위치가 확정된다.

2회전: 80을 제외한 앞의 세 값을 비교한다. 40과 65는 교환하지 않고, 65와 15를 교환하여 [40, 15, 65, 80]이 된다. 자리바꿈은 1회이며, 65의 위치도 확정된다.

3회전: 남은 앞의 두 값 40과 15를 교환하면 [15, 40, 65, 80]이 되어 오름차순 정렬이 끝난다. 자리바꿈은 1회이다.

자리바꿈의 총횟수는 2 + 1 + 1 = 4회로, ②의 4에 해당한다. **비교했더라도 교환하지 않은 경우는 횟수에 포함하지 않는다.**

NoteBase

**문 12. 다음 표와 같이 분말이 있을 때, 40 kg 무게까지 허용가능한 배낭에 최대 이익을 얻을 수 있도록 분말을 넣는 알고리즘의 C 프로그램이 아래와 같다. (가), (나)에 들어갈 코드와 출력값을 바르게 연결한 것은? (단, 배낭에 각 분말 일부만 넣을 수도 있다)**

| 분말 종류 | 보유량(kg) | 이익 |
|---|---|---|
| A | 10 | 60 |
| B | 18 | 90 |
| C | 25 | 100 |
| D | 15 | 120 |

```
#include <stdio.h>

int main() {
    double wgt[] = {10, 18, 25, 15};
    double val[] = {60, 90, 100, 120};
    double ratio[4] = {}, W = 40, max_r,
           totalVal = 0.0;
    int i, max_i;

    for (i = 0; i < 4; i++)
        ratio[i] = (가);
    while (W > 0) {
        max_r = -1.0;
        max_i = -1;

        for (i = 0; i < 4; i++) {
            if (wgt[i] > 0 && ratio[i] > max_r) {
                max_r = ratio[i];
                max_i = i;
            }
        }
        if (max_i == -1) break;
        if (W >= wgt[max_i]) {
            W -= wgt[max_i];
            totalVal += val[max_i];
        }
        else {
            totalVal += val[max_i] * (나);
            break;
        }
        wgt[max_i] = 0;
    }
    printf("%.1f\n", totalVal);
    return 0;
}
```

① (가): val[i] / wgt[i], (나): W / wgt[max_i], 출력값: 255.0

② (가): val[i] / wgt[i], (나): wgt[max_i] / W, 출력값: 255.0

③ (가): val[i] / wgt[i], (나): W / wgt[max_i], 출력값: 220.0

④ (가): wgt[i] / val[i], (나): W / wgt[max_i], 출력값: 220.0

**정답 1번**

NoteBase

**간단한 해설**

분말을 나누어 넣을 수 있으므로 **1 kg당 이익이 큰 분말부터** 담는다. D와 A를 전량 넣고 남은 15 kg에 B의 일부를 넣으며, 이때 B의 이익은 전체 이익에 **담은 무게의 비율**을 곱한 값이므로 총이익은 255.0이다.

**빈칸별 해설**

(가): val[i] / wgt[i]. val[i]는 해당 분말 전체의 이익, wgt[i]는 전체 무게이며, 두 값을 나누면 1 kg당 이익이 된다. 분말을 나누어 담을 수 있으므로 이익이 낮은 분말을 같은 무게의 이익이 높은 분말로 바꾸면 총이익이 증가한다. 따라서 **단위 무게당 이익이 가장 큰 분말**부터 선택한다. 프로그램도 ratio[i]가 가장 큰 항목을 max_i로 선택하므로 이익을 무게로 나누어야 한다. 반대로 wgt[i] / val[i]의 최댓값을 고르면 단위 무게당 이익이 가장 작은 분말을 먼저 선택하게 된다.

(나): W / wgt[max_i]. W는 배낭의 남은 허용 무게이다. 선택한 분말 전체를 담을 수 없는 else 분기에서는 **남은 허용 무게를 분말 전체 무게로 나눈 비율**만큼만 담는다. 따라서 추가 이익은 전체 이익·담는 무게전체 무게이다. 이 분기에서는 W가 wgt[max_i]보다 작으므로 담는 비율은 1보다 작다. wgt[max_i] / W로 뒤집으면 전체 이익보다 큰 값을 더하게 된다.

출력값: A의 1 kg당 이익은 $\frac{60}{10} = 6$, B는 $\frac{90}{18} = 5$, C는 $\frac{100}{25} = 4$, D는 $\frac{120}{15} = 8$이다. 먼저 D 15 kg을 전량 담아 이익 120을 얻으면 W는 25 kg이 된다. 이어 A 10 kg을 전량 담으면 누적 이익은 180, W는 15 kg이 된다. 전량 담은 분말은 wgt[max_i] = 0으로 표시되어 다음 선택에서 제외된다. 다음으로 B를 선택하지만 보유량 18 kg을 모두 담을 수 없어 15 kg만 담는다. B에서 얻는 이익은 $90 \cdot \frac{15}{18} = 75$이므로 총이익은 120 + 60 + 75 = 255이다. 부분을 담은 뒤 break로 반복을 종료하므로 C는 담지 않는다. printf의 %.1f는 소수점 이하 한 자리까지 표시하므로 출력값은 255.0이다.

(가)의 val[i] / wgt[i], (나)의 W / wgt[max_i], 출력값 255.0을 연결한 것은 ①이다.

NoteBase

### 문 13. 다음 C 프로그램의 실행 결과에 포함되지 않는 것은?

```
#include <stdio.h>

int count = 0;

void f(int n, char from, char tmp, char to){
    count++;
    if (n == 1)
        printf("%d: 원판 %d를(을) %c에서 %c로 이동\n
", count, n, from, to);
    else {
        f(n - 1, from, to, tmp);
        printf("%d: 원판 %d를(을) %c에서 %c로 이동\n
", count, n, from, to);
        f(n - 1, tmp, from, to);
    }
}

int main() {
    f(3, 'a', 'b', 'c');
    return 0;
}
```

① 4: 원판 1를(을) c에서 b로 이동

② 4: 원판 3를(을) a에서 c로 이동

③ 5: 원판 1를(을) b에서 a로 이동

④ 7: 원판 1를(을) a에서 c로 이동

**정답 3번**

#### 간단한 해설

count는 **함수에 진입할 때마다** 증가하는 전역 변수이다. 원판 1을 b에서 a로 옮기기 전에는 원판 2를 처리하는 호출과 원판 1을 처리하는 호출이 연이어 실행되므로, 이때 출력되는 count는 5가 아니라 **6**이다.

NoteBase

**계산 과정**

호출 규칙: f의 인수 n은 옮길 원판의 수이며, from은 출발 기둥, tmp는 보조 기둥, to는 도착 기둥이다. count는 0에서 시작하는 전역 변수로, **모든 f 호출의 시작에서 1씩 증가**한다. n이 1이면 바로 출력하고, n이 1보다 크면 첫 재귀 호출을 끝낸 뒤 원판 n의 이동을 출력하고 두 번째 재귀 호출을 실행한다. 따라서 출력문의 count는 해당 호출에 처음 진입했을 때의 값이 아니라, 출력 시점까지 누적된 함수 호출 횟수이다.

첫 재귀 호출: main의 f(3, 'a', 'b', 'c')에 진입하면 count는 1이다. 이어 f(2, 'a', 'c', 'b')에 진입하여 2가 되고, f(1, 'a', 'b', 'c')에 진입하여 3이 된다. 이때 '3: 원판 1를(을) a에서 c로 이동'을 출력한다. 원판 2를 처리하던 호출로 돌아와도 count는 증가하지 않으므로 '3: 원판 2를(을) a에서 b로 이동'을 출력한다. 다음 f(1, 'c', 'a', 'b')에 진입하면 count가 4가 되어 '4: 원판 1를(을) c에서 b로 이동'을 출력한다.

원판 3의 이동: 첫 재귀 호출이 끝나면 최초의 f(3, 'a', 'b', 'c')로 돌아온다. **호출에서 복귀할 때는 count가 증가하지 않으므로** 값은 4이다. 따라서 '4: 원판 3를(을) a에서 c로 이동'을 출력한다.

두 번째 재귀 호출: f(2, 'b', 'a', 'c')에 진입하면 count가 5가 된다. 그러나 이 호출은 바로 출력하지 않고 먼저 f(1, 'b', 'c', 'a')를 호출하므로, **출력 전에 count가 6으로 증가**한다. 이때 '6: 원판 1를(을) b에서 a로 이동'을 출력한다. 돌아온 뒤에는 '6: 원판 2를(을) b에서 c로 이동'을 출력하고, 마지막 f(1, 'a', 'b', 'c')에 진입하여 '7: 원판 1를(을) a에서 c로 이동'을 출력한다.

실행 결과에는 ①·②·④의 문장이 모두 포함된다. b에서 a로 원판 1을 옮기는 문장의 앞 숫자는 6이므로, '5: 원판 1를(을) b에서 a로 이동'인 ③은 포함되지 않는다.

NoteBase

**문 14. 비어있는 이진 탐색 트리(binary search tree)에 다음 키값이 순서대로 입력되면, 15를 찾기 위해 방문해야 하는 노드의 개수는?**

50, 40, 30, 35, 20, 15, 25, 10

① 3

② 4

③ 5

④ 6

**정답 3번**

**간단한 해설**

이진 탐색 트리에서는 찾는 값이 현재 노드보다 작으면 왼쪽 자식으로 이동한다. 15의 탐색 경로는 50 → 40 → 30 → 20 → 15이며, **시작 노드와 찾은 노드까지 포함**해 5개를 방문한다.

**계산 과정**

이진 탐색 트리는 각 노드의 왼쪽 서브트리에 더 작은 키를, 오른쪽 서브트리에 더 큰 키를 둔다. 삽입할 때는 루트부터 키를 비교하며 내려가 **처음 만나는 빈 자리에 새 노드**를 연결한다.

트리 구성: 빈 트리에 처음 입력한 50이 루트가 된다. 이어서 40은 50의 왼쪽, 30은 40의 왼쪽에 놓인다. 35는 50과 40보다 작지만 30보다 크므로 30의 오른쪽에 놓인다. 20은 30의 왼쪽, 15는 20의 왼쪽에 놓인다. 25는 30보다 작고 20보다 크므로 20의 오른쪽에, 10은 15의 왼쪽에 놓인다.

15 탐색: 루트 50에서 시작한다. 15는 50, 40, 30, 20보다 차례로 작으므로 매번 왼쪽 자식으로 이동한다. 다음 노드 15에서 키가 일치하여 탐색을 끝낸다. 방문 경로는 50 → 40 → 30 → 20 → 15이다. 35와 25는 이 경로 밖에 있고, 10은 15에서 탐색이 끝나므로 방문하지 않는다.

방문 노드 수: 경로에 있는 노드는 50, 40, 30, 20, 15의 5개이다. 노드 사이의 이동은 4번이지만, 방문 개수에는 **처음 비교한 루트도 포함**한다. 따라서 방문해야 하는 노드는 5개로, ③의 값과 일치한다.

NoteBase

**문 15. 다음 배열에서 보간 탐색(interpolation search)으로 58을 찾고자 할 때, 첫 번째 탐색 위치는? (단, 배열에서 위치의 차이는 값의 차이에 비례한다는 가정하에 탐색 위치를 계산하며 소수점 이하는 반올림한다)**

| 위치 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 배열 | 3 | 7 | 12 | 22 | 32 | 58 | 67 | 80 | 87 | 89 |

① 2

② 4

③ 6

④ 8

**정답 3번**

**간단한 해설**

보간 탐색은 **값 차이의 비율을 위치 차이에 적용**하여 탐색 위치를 추정한다. 양 끝의 위치와 값을 이용하면 약 5.76이므로, 문제의 반올림 조건에 따른 첫 탐색 위치는 6이다.

**계산 과정**

그림에서 배열의 위치는 0부터 9까지이고, 양 끝의 값은 각각 3과 89이다. 보간 탐색은 정렬된 배열에서 **위치 차이가 값 차이에 비례한다**고 가정하여 찾는 값의 위치를 추정한다. 따라서 전체 값 차이 중 찾는 값까지의 차이가 차지하는 비율만큼, 왼쪽 끝에서 오른쪽으로 이동한 위치를 먼저 탐색한다.

찾는 값은 58이므로 왼쪽 끝 값과의 차이는 58 − 3 = 55이고, 양 끝 값의 차이는 89 − 3 = 86이다. 양 끝 위치의 간격은 원소 개수 10이 아니라 **9 − 0 = 9**이므로, 첫 탐색 위치를 p라 하면 $p = 0 + \frac{58-3}{89-3} \cdot (9 - 0)$이다. 이를 계산하면 $p = \frac{495}{86} \approx 5.76$으로, 왼쪽 끝 위치 0에서 약 5.76만큼 이동한 위치를 뜻한다.

소수점 이하는 반올림하므로 첫 탐색 위치는 6이며, 이는 ③의 값과 일치한다. 그림에서 58이 실제로 놓인 위치는 5이지만, 묻는 것은 그 실제 위치가 아니라 **비례식으로 추정한 첫 탐색 위치**이다.

NoteBase

**문 16. 다음과 같이 전체 버킷 개수가 13개이고 버킷당 1개의 슬롯을 가지는 비어있는 해시 테이블에 값 <7, 20, 28, 46, 81, 67, 4>를 순서대로 해시 함수를 사용하여 저장하였을 때, 버킷 번호 4에 저장되는 값은? (단, 해시 함수로 h(x) = x mod 13을 사용하며, 충돌 해결은 개방 주소 방법의 선형 조사법(linear probing)을 적용한다)**

| 버킷 번호 | 슬롯 |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | |
| 9 | |
| 10 | |
| 11 | |
| 12 | |

① 4

② 46

③ 67

④ 81

**정답 3번**

**간단한 해설**

67을 넣을 때 해시값은 2이지만, 버킷 2와 3에는 이미 28과 81이 있어 **처음 빈 버킷인 4에 67을 저장**한다. 마지막 값 4는 버킷 4가 차 있으므로 버킷 5에 저장된다.

NoteBase

**계산 과정**

그림의 버킷 번호는 0~12이고 각 버킷에는 값 하나만 저장할 수 있다. 해시 함수 h(x)=x 13은 값 x를 13으로 나눈 나머지를 최초 저장 위치로 정한다. 선형 조사법은 그 위치가 차 있으면 **버킷 번호를 하나씩 늘려 처음 만나는 빈 위치**에 저장하며, 기존 값을 덮어쓰지 않는다. 주어진 삽입 순서대로 위치를 구한다.

7 삽입: $h(7) = 7$이다. 버킷 7이 비어 있으므로 7을 저장한다.

20 삽입: h(20)=20 13=7이다. 버킷 7에는 7이 있어 충돌하므로 다음 버킷 8에 20을 저장한다.

28 삽입: h(28)=28 13=2이다. 버킷 2가 비어 있으므로 28을 저장한다.

46 삽입: h(46)=46 13=7이다. 버킷 7과 8에는 각각 7과 20이 있으므로 차례로 지나가고, 빈 버킷 9에 46을 저장한다.

81 삽입: h(81)=81 13=3이다. 버킷 3이 비어 있으므로 81을 저장한다.

67 삽입: h(67)=67 13=2이다. 버킷 2에는 28, 버킷 3에는 81이 있으므로 **2 → 3 → 4 순서로 조사**하여 빈 버킷 4에 67을 저장한다.

4 삽입: $h(4) = 4$이지만 버킷 4에는 이미 67이 있다. 따라서 67을 덮어쓰지 않고 다음 빈 버킷 5에 4를 저장한다.

최종적으로 버킷 번호 4에 저장되는 값은 67이며, ③의 값과 일치한다.

NoteBase

**문 17. 다음 두 문자열 A와 B의 편집 거리(edit distance)는? (단, 허용하는 문자열 연산은 삽입, 삭제, 교체 연산이다)**

◯ 문자열 A: algorithm
◯ 문자열 B: anthem

① 5
② 6
③ 7
④ 9

**정답 2번**

**간단한 해설**

편집 거리는 삽입·삭제·교체로 한 문자열을 다른 문자열로 바꾸는 **최소 연산 횟수**이다. algorithm은 교체 1회, 삭제 4회, 삽입 1회로 anthem이 되며, 최소 횟수는 6회이다.

NoteBase

**계산 과정**

편집 거리의 의미: 문자열 A는 algorithm으로 9글자이고, 문자열 B는 anthem으로 6글자이다.
삽입·삭제·교체는 각각 한 글자에 대한 연산이며, 각 연산을 1회로 센다. **길이 차이 3은 필요한 삭제 횟수와 삽입 횟수의 차이**일 뿐, 문자열을 같게 만드는 최소 연산 횟수는 아니다.

최소 횟수 계산 방법: $D(i, j)$를 A의 앞 i글자를 B의 앞 j글자로 바꾸는 최소 연산 횟수라고 하자. 빈 문자열로 바꾸려면 모두 삭제하고, 빈 문자열에서 만들려면 모두 삽입하므로 $D(i, 0) = i$, $D(0, j) = j$이다. 마지막 글자가 같으면 두 글자를 그대로 대응시켜 $D(i, j) = D(i-1, j-1)$로 계산한다. 다르면 삭제에 필요한 $D(i-1, j) + 1$, 삽입에 필요한 $D(i, j-1) + 1$, 교체에 필요한 $D(i-1, j-1) + 1$ 중 최솟값을 택한다. 각 칸이 앞서 계산한 최소 횟수를 이용하므로 가능한 편집 경로 중 최소를 구할 수 있다.

계산 결과: 위 규칙을 접두사의 길이가 짧은 것부터 적용한다. A 전체인 algorithm을 B의 접두사인 빈 문자열, a, an, ant, anth, anthe, anthem으로 바꾸는 최소 횟수는 차례로 9, 8, 8, 7, 6, 6, 6이다. 따라서 두 문자열 전체에 대한 값은 $D(9, 6) = 6$이다.

6회로 바꾸는 실제 과정: algorithm의 l을 n으로 교체하면 angorithm이 된다. 여기서 g, o, r, i를 각각 삭제하면 anthm이 되고, h와 m 사이에 e를 삽입하면 anthem이 된다. **교체 1회 + 삭제 4회 + 삽입 1회 = 6회**로, 앞에서 구한 최소 횟수를 실제로 달성한다.

두 문자열의 편집 거리는 6이므로 ②의 값과 일치한다.

NoteBase

**문 18. 다음 배열에 대해 아래 알고리즘을 적용하여 정렬하고자 한다. 3번째 for 루프를 수행할 때, 배열 내에서 교환되는 두 값은?**

| 위치 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 배열 | 9 | 21 | 54 | 32 | 77 | 45 | 19 | 83 | 12 | 3 |

```
Sort(A[], n): // n은 입력배열 A의 크기이다.
    for last ← (n - 1) downto 1
        A[0 ... last] 중 가장 큰 수 A[k]를 찾는다.
        A[k]와 A[last]의 값을 교환한다.
```

① 3, 54

② 21, 45

③ 21, 54

④ 32, 77

**정답 1번**

**간단한 해설**

이 알고리즘은 남은 구간의 **최댓값을 구간 맨 뒤의 값과 교환**한다. 앞선 두 번의 반복으로 83과 77이 뒤쪽에 배치되면, 세 번째 반복에서는 위치 0~7의 최댓값 54와 위치 7의 값 3을 교환한다.

NoteBase

**계산 과정**

배열 크기는 n = 10이고, last는 정렬할 구간의 마지막 위치이다. 매 반복에서 A[0]부터 A[last]까지의 최댓값을 찾아 A[last]와 교환하므로, **큰 값부터 뒤쪽 위치에 확정**된다. last는 9, 8, 7 순으로 줄어들며, 이미 확정된 뒤쪽 값은 다음 탐색에서 제외된다.

첫 번째 반복(last = 9): 위치 0~9의 최댓값은 위치 7의 83이다. 이를 위치 9의 3과 교환하면 배열은 [9, 21, 54, 32, 77, 45, 19, 3, 12, 83]이 된다. 83의 위치가 확정되고, 3은 위치 7로 이동한다.

두 번째 반복(last = 8): 위치 0~8의 최댓값은 위치 4의 77이다. 이를 위치 8의 12와 교환하면 배열은 [9, 21, 54, 32, 12, 45, 19, 3, 77, 83]이 된다. 이제 77과 83은 다음 탐색에서 제외된다.

세 번째 반복(last = 7): 위치 0~7의 값은 [9, 21, 54, 32, 12, 45, 19, 3]이다. 최댓값은 위치 2의 54이고, 구간 맨 뒤인 위치 7의 값은 **첫 번째 반복에서 이동해 온 3**이다. 따라서 A[2]와 A[7]의 값인 54와 3을 교환한다.

교환되는 두 값은 3과 54이므로 ①의 '3, 54'에 해당한다.

NoteBase

**문 19. 다음과 같이 오름차순 정렬을 수행하는 알고리즘은?**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 초기 상태 | 5 | 20 | 17 | 6 | 2 | 13 | 10 |
| 1단계 | 5 | 20 | 17 | 6 | 2 | 13 | 10 |
| 2단계 | 5 | 17 | 20 | 6 | 2 | 13 | 10 |
| 3단계 | 5 | 6 | 17 | 20 | 2 | 13 | 10 |
| 4단계 | 2 | 5 | 6 | 17 | 20 | 13 | 10 |
| 5단계 | 2 | 5 | 6 | 13 | 17 | 20 | 10 |
| 6단계 | 2 | 5 | 6 | 10 | 13 | 17 | 20 |

① 병합 정렬
② 삽입 정렬
③ 선택 정렬
④ 퀵 정렬

**정답 2번**

**간단한 해설**

왼쪽의 정렬된 부분에 **다음 원소를 알맞은 위치에 끼워 넣는** 삽입 정렬이다. 3단계에서는 6을 5와 17 사이에 넣고, 4단계에서는 2를 맨 앞으로 넣어 정렬된 부분을 하나씩 늘린다.

NoteBase

**공통 해설**

삽입 정렬은 첫 원소를 정렬된 부분으로 보고, 그다음 원소부터 차례로 가져와 앞부분의 알맞은 위치에 넣는다. 오름차순에서는 가져온 원소보다 큰 값들을 오른쪽으로 한 칸씩 밀어 자리를 만든다. 각 단계가 끝나면 **왼쪽의 정렬된 부분이 원소 하나만큼 늘어난다**는 것이 공통 기준이다.

1~3단계: 처음에는 5 하나가 정렬된 부분이다. 1단계에서 20은 5보다 크므로 그대로 둔다. 2단계에서는 17을 넣기 위해 20을 오른쪽으로 옮겨 앞부분이 5, 17, 20이 된다. 3단계에서는 6보다 큰 20과 17을 오른쪽으로 옮기고 6을 5 다음에 넣어 5, 6, 17, 20을 만든다.

4단계: 다음 원소인 2는 앞의 5, 6, 17, 20보다 모두 작다. 따라서 이 네 값을 오른쪽으로 한 칸씩 옮기고 2를 맨 앞에 넣어 2, 5, 6, 17, 20을 만든다. 아직 처리하지 않은 13과 10은 그대로 남아 있다.

5~6단계: 5단계에서는 13보다 큰 17과 20을 오른쪽으로 옮긴 뒤 13을 6과 17 사이에 넣는다. 6단계에서는 10보다 큰 13, 17, 20을 오른쪽으로 옮긴 뒤 10을 6 다음에 넣는다. 그 결과 전체가 2, 5, 6, 10, 13, 17, 20으로 정렬된다.

NoteBase

**문 20. 문자에 대한 빈도수가 다음과 같을 때, exam을 허프만(Huffman) 코드로 작성하면 비트 수는?**

| 문자 | a | b | c | e | m | x |
|---|---|---|---|---|---|---|
| 빈도수 | 8 | 5 | 3 | 10 | 6 | 1 |

① 10

② 11

③ 12

④ 14

**정답 1번**

**간단한 해설**

허프만 트리에서 **루트부터 문자까지의 간선 수**가 그 문자의 코드 길이이다. e·a·m은 각각 2비트, x는 4비트이므로 'exam'을 표현하는 데 2 + 4 + 2 + 2 = 10비트가 필요하다.

**계산 과정**

주어진 빈도수는 a가 8, b가 5, c가 3, e가 10, m이 6, x가 1이다. 허프만 코드는 **빈도수가 가장 작은 두 노드를 반복해서 합쳐** 이진 트리를 만든다. 합친 노드의 빈도수는 두 노드의 빈도수를 더한 값이며, 이후에는 이 노드도 다른 노드와 함께 비교한다.

트리 구성: x(1)와 c(3)를 합쳐 4를 만들고, 이 4와 b(5)를 합쳐 9를 만든다. 남은 노드 중 가장 작은 m(6)과 a(8)를 합쳐 14를 만든 뒤, 9와 e(10)를 합쳐 19를 만든다. 마지막으로 14와 19를 합친 33이 루트가 된다.

코드 길이: 각 간선에 0 또는 1을 부여하므로 **루트에서 해당 문자까지 지나는 간선 하나당 1비트**가 필요하다. 루트 → 19 → e는 2개, 루트 → 19 → 9 → 4 → x는 4개이다. a와 m은 각각 루트 → 14 → a, 루트 → 14 → m으로 간선이 2개이므로 둘 다 2비트이다.

총 비트 수: 'exam'에는 e, x, a, m이 한 번씩 나오므로 각 코드 길이를 더한다. 표의 빈도수는 트리를 만드는 데 사용한 값이며, 이 문자열의 비트 수를 구할 때 다시 곱하지 않는다. 따라서 2 + 4 + 2 + 2 = 10비트이며, ①의 10에 해당한다.

NoteBase