# 문제 01 · 객관식

완전 탐색(brute force)을 가리키는 다른 이름으로 볼 수 없는 것은?

① 탐욕적 선택 기법(greedy method)

② 전수 조사(exhaustive search)

③ 무차별 대입 기법

④ 순진한 방법(naive method)

⑤ 모든 경우 시도 기법

# 정답 및 해설 01

완전 탐색(brute force)을 가리키는 다른 이름으로 볼 수 없는 것은?

## 정답 ① 탐욕적 선택 기법(greedy method)

### 해설

완전 탐색은 가능한 경우를 빠짐없이 확인하는 전략이며 exhaustive search, naive method, 무차별 대입 등으로 불린다. 탐욕적 선택은 매 순간 가장 좋아 보이는 것만 고르고 되돌아보지 않는 전혀 다른 전략이다.

# 문제 02 · 객관식

완전 탐색에 대한 설명으로 옳지 않은 것은?

① 모든 경우를 확인하므로 정답을 놓치지 않는다

② 복잡한 수학 없이 반복문만으로 구현할 수 있다

③ 경우의 수가 많아도 실행 시간이 거의 늘지 않는다

④ 입력이 커지면 경우의 수가 급격히 늘어난다

⑤ 고급 알고리즘을 설계할 때 출발점으로 삼을 수 있다

# 정답 및 해설 02

완전 탐색에 대한 설명으로 옳지 않은 것은?

**정답 ③ 경우의 수가 많아도 실행 시간이 거의 늘지 않는다**

## 해설

완전 탐색의 치명적 약점이 바로 실행 시간이다. 경우의 수가 곧 실행 횟수이므로 입력이 조금만 커져도 시간이 폭발한다. 나머지 넷은 완전 탐색의 대표적인 성질이다.

# 문제 03 · 객관식

0부터 9까지의 숫자로 이루어진 4자리 자물쇠의 비밀번호 경우의 수는?

① 10,000가지

② 40가지

③ 5,040가지

④ 1,000가지

⑤ 100,000가지

# 정답 및 해설 03

0부터 9까지의 숫자로 이루어진 4자리 자물쇠의 비밀번호 경우의 수는?

**정답 ① 10,000가지**

**해설**

각 자리마다 10가지이므로 $10 \times 10 \times 10 \times 10 = 10^4 = 10{,}000$가지다. 자릿수가 하나 늘 때마다 경우의 수는 10배가 된다.

# 문제 04 · 객관식

다음 코드에서 출력되는 count 값은?

code.py

```
secret = "8234"
count = 0
for d1 in range(10):
    for d2 in range(10):
        for d3 in range(10):
            for d4 in range(10):
                attempt = f"{d1}{d2}{d3}{d4}"
                count += 1
                if attempt == secret:
                    print(count); exit()
```

① 8235

② 8234

③ 8236

④ 10000

⑤ 1

# 정답 및 해설 04

다음 코드에서 출력되는 count 값은?

code.py

```
secret = "8234"
count = 0
for d1 in range(10):
    for d2 in range(10):
        for d3 in range(10):
            for d4 in range(10):
                attempt = f"{d1}{d2}{d3}{d4}"
                count += 1
                if attempt == secret:
                    print(count); exit()
```

**정답 ① 8235**

## 해설

0000부터 순서대로 시도하며 시도할 때마다 count를 1 증가시킨다. 0000이 1회째이므로 8234는 8234 + 1 = 8235회째에 발견된다. 숫자 자체와 시도 횟수가 1만큼 차이 난다는 점이 핵심이다.

# 문제 05 · 객관식

완전 탐색이 정답을 놓치지 않는 근거로 가장 적절한 것은?

① 반복문이 언제나 정해진 횟수만큼만 돌고 끝나기 때문이다

② 경우의 수를 미리 계산해 두고 탐색을 시작하기 때문이다

③ 정답에 가까울 것 같은 후보부터 우선 확인하기 때문이다

④ 가능한 경우를 하나도 빠뜨리지 않고 모두 확인하기 때문이다

⑤ 조건을 만족하지 않는 후보를 미리 걸러내고 시작하기 때문이다

# 정답 및 해설 05

완전 탐색이 정답을 놓치지 않는 근거로 가장 적절한 것은?

**정답 ④ 가능한 경우를 하나도 빠뜨리지 않고 모두 확인하기 때문이다**

## 해설

탐색 공간을 빠짐없이 훑는다는 점이 정확성의 유일한 근거다. 정답에 가까운 것부터 보거나 미리 걸러내는 것은 휴리스틱이나 가지치기이며 완전 탐색 자체의 성질이 아니다.

# 문제 06 · 객관식

완전 탐색을 우선 시도해 볼 만한 상황으로 가장 적절한 것은?

① 입력의 크기가 미리 정해져 있지 않아 알 수 없는 경우

② 탐색해야 할 경우의 수가 수백만 이하로 계산되는 경우

③ 구해야 할 정답이 여러 개일 가능성이 있는 경우

④ 데이터가 미리 오름차순으로 정렬되어 있는 경우

⑤ 반복문을 두 겹 이상 사용할 수 없다는 제약이 있는 경우

# 정답 및 해설 06

완전 탐색을 우선 시도해 볼 만한 상황으로 가장 적절한 것은?

**정답 ② 탐색해야 할 경우의 수가 수백만 이하로 계산되는 경우**

## 해설

완전 탐색의 가능 여부를 가르는 것은 오직 탐색 공간의 크기다. 경우의 수를 먼저 계산해 수백만 이하이면 시도해 볼 만하다. 정렬 여부나 정답 개수는 판단 기준이 아니다.

# 문제 07 · 객관식

완전 탐색에서 중첩 반복문의 개수를 결정하는 것은?

① 문제에서 요구하는 정답의 개수

② 입력 데이터의 총 개수

③ 동시에 정해야 하는 변수의 개수

④ 사용할 수 있는 메모리의 크기

⑤ 리스트에 저장된 값의 최대 크기

# 정답 및 해설 07

완전 탐색에서 중첩 반복문의 개수를 결정하는 것은?

**정답 ③ 동시에 정해야 하는 변수의 개수**

### 해설

찾아야 할 변수가 하나면 반복문 하나, 둘이면 둘, 셋이면 셋이다. 변수의 개수와 중첩 깊이가 같다는 것이 이 절의 핵심 패턴이며, 그만큼 시간 복잡도의 차수도 올라간다.

# 문제 08 · 객관식

다음 코드에서 find_divisors(10)의 반환값은?

code.py

```
def find_divisors(n):
    divisors = []
    for i in range(1, n + 1):
        if n % i == 0:
            divisors.append(i)
    return divisors
```

① [1, 2, 5, 10]

② [1, 2, 4, 5, 10]

③ [1, 2, 5]

④ [2, 5, 10]

⑤ [1, 5, 10]

# 정답 및 해설 08

다음 코드에서 find_divisors(10)의 반환값은?

code.py

```
def find_divisors(n):
    divisors = []
    for i in range(1, n + 1):
        if n % i == 0:
            divisors.append(i)
    return divisors
```

**정답** ① **[1, 2, 5, 10]**

**해설**

1부터 10까지 모두 시도하여 10을 나누어떨어지게 하는 수를 모은다. 1, 2, 5, 10이 해당하며 1과 자기 자신도 약수이므로 포함된다.

# 문제 09 · 객관식

다음 코드의 시간 복잡도는? (n은 입력값)

code.py

```
def find_divisors(n):
    divisors = []
    for i in range(1, n + 1):
        if n % i == 0:
            divisors.append(i)
    return divisors
```

① $O(\sqrt{n})$

② $O(1)$

③ $O(\log n)$

④ $O(n^2)$

⑤ $O(n)$

# 정답 및 해설 09

다음 코드의 시간 복잡도는? (n은 입력값)

code.py

```python
def find_divisors(n):
    divisors = []
    for i in range(1, n + 1):
        if n % i == 0:
            divisors.append(i)
    return divisors
```

**정답 ⑤ O(n)**

### 해설

반복문이 1부터 n까지 정확히 n번 돈다. 참고로 √n까지만 검사하고 짝을 함께 구하는 방식으로 개선하면 O(√n)이 되지만, 이 코드는 그렇게 되어 있지 않다.

# 문제 10 · 객관식

다음 코드를 arr = [1, 3, 5, 7, 9], K = 10 으로 실행한 결과는?

code.py

```
def find_pairs(arr, K):
    pairs = []
    for i in range(len(arr) - 1):
        for j in range(i + 1, len(arr)):
            if arr[i] + arr[j] == K:
                pairs.append((arr[i], arr[j]))
    return pairs
```

① [(3, 7)]

② [(1, 9), (3, 7), (5, 5)]

③ [(1, 9), (9, 1), (3, 7), (7, 3)]

④ [(1, 9), (3, 7)]

⑤ [(5, 5)]

# 정답 및 해설 10

다음 코드를 arr = [1, 3, 5, 7, 9], K = 10 으로 실행한 결과는?

code.py

```python
def find_pairs(arr, K):
    pairs = []
    for i in range(len(arr) - 1):
        for j in range(i + 1, len(arr)):
            if arr[i] + arr[j] == K:
                pairs.append((arr[i], arr[j]))
    return pairs
```

**정답 ④ [(1, 9), (3, 7)]**

**해설**

j를 i+1부터 돌리므로 같은 쌍을 두 번 세지 않고 자기 자신과도 짝짓지 않는다. 합이 10인 쌍은 (1,9)와 (3,7) 두 개다.

# 문제 11 · 객관식

다음 코드에서 안쪽 반복문의 시작을 j = i + 1 로 둔 이유로 가장 적절한 것은?

code.py

```
def find_pairs(arr, K):
    pairs = []
    for i in range(len(arr) - 1):
        for j in range(i + 1, len(arr)):
            if arr[i] + arr[j] == K:
                pairs.append((arr[i], arr[j]))
    return pairs
```

① 인덱스가 리스트 범위를 벗어나지 않게 하기 위해

② 리스트를 정렬된 상태로 유지하기 위해

③ 같은 쌍이 두 번 세어지는 것을 막기 위해

④ 합이 K보다 커지는 경우를 미리 제외하기 위해

⑤ 시간 복잡도를 O(n)으로 낮추기 위해

# 정답 및 해설 11

다음 코드에서 안쪽 반복문의 시작을 j = i + 1 로 둔 이유로 가장 적절한 것은?

code.py

```
def find_pairs(arr, K):
    pairs = []
    for i in range(len(arr) - 1):
        for j in range(i + 1, len(arr)):
            if arr[i] + arr[j] == K:
                pairs.append((arr[i], arr[j]))
    return pairs
```

**정답 ③ 같은 쌍이 두 번 세어지는 것을 막기 위해**

## 해설

j를 0부터 돌리면 (1,9)와 (9,1)이 모두 세어지고 (1,1)처럼 자기 자신과 짝을 이루는 경우도 생긴다. i+1부터 시작하면 i < j가 보장되어 각 쌍이 한 번씩만 나온다. 복잡도는 여전히 $O(n^2)$이다.

# 문제 12 · 객관식

길이 n인 리스트에서 서로 다른 두 원소의 쌍을 모두 검사할 때 총 비교 횟수는?

① $n(n+1)/2$

② $n^2$

③ $n(n-1)/2$

④ $n-1$

⑤ $2n$

# 정답 및 해설 12

길이 n인 리스트에서 서로 다른 두 원소의 쌍을 모두 검사할 때 총 비교 횟수는?

**정답 ③ $n(n-1)/2$**

## 해설

i가 고정될 때마다 j가 하나씩 줄어들며 도므로 $(n-1) + (n-2) + \dots + 1 = n(n-1)/2$ 이다. 빅오로는 $O(n^2)$이다.

# 문제 13 · 객관식

다음 코드에서 안쪽 조건 검사가 실행되는 총 횟수는?

code.py

```
def dice_sum(target):
    results = []
    for d1 in range(1, 7):
        for d2 in range(1, 7):
            for d3 in range(1, 7):
                if d1 + d2 + d3 == target:
                    results.append((d1, d2, d3))
    return results
```

① 216회

② 18회

③ 36회

④ 27회

⑤ 666회

# 정답 및 해설 13

다음 코드에서 안쪽 조건 검사가 실행되는 총 횟수는?

code.py

```
def dice_sum(target):
    results = []
    for d1 in range(1, 7):
        for d2 in range(1, 7):
            for d3 in range(1, 7):
                if d1 + d2 + d3 == target:
                    results.append((d1, d2, d3))
    return results
```

**정답 ① 216회**

## 해설

주사위마다 6가지이므로 6 × 6 × 6 = 216회다. 27은 세 수의 합이 10이 되는 경우의 수이지 반복 횟수가 아니다. 반복 횟수와 조건을 통과한 개수를 구별해야 한다.

# 문제 14 · 객관식

찾아야 할 변수가 세 개이고 각 변수의 후보가 n가지일 때, 중첩 반복문으로 구현한 완전 탐색의 시간 복잡도는?

① $O(n \log n)$

② $O(n^2)$

③ $O(3n)$

④ $O(n^3)$

⑤ $O(3^n)$

# 정답 및 해설 14

찾아야 할 변수가 세 개이고 각 변수의 후보가 n가지일 때, 중첩 반복문으로 구현한 완전 탐색의 시간 복잡도는?

**정답 ④ $O(n^3)$**

## 해설

변수 하나마다 반복문 한 겹이 붙고 각 반복이 n번 도므로 $n \times n \times n = n^3$ 이다. 변수 개수가 지수 자리가 아니라 곱해지는 횟수로 들어간다는 점을 혼동하지 않아야 한다.

# 문제 15 · 객관식

완전 탐색의 3단계 템플릿을 순서대로 바르게 나열한 것은?

① 후보 해 생성 → 해 선택 → 조건 검증

② 조건 검증 → 후보 해 생성 → 해 선택

③ 해 선택 → 후보 해 생성 → 조건 검증

④ 후보 해 생성 → 조건 검증 → 해 선택

⑤ 조건 검증 → 해 선택 → 후보 해 생성

# 정답 및 해설 15

완전 탐색의 3단계 템플릿을 순서대로 바르게 나열한 것은?

**정답 ④ 후보 해 생성 → 조건 검증 → 해 선택**

## 해설

가능한 경우를 만들고(생성), 문제 조건을 만족하는지 보고(검증), 통과한 것을 답으로 삼거나 최적값과 비교한다(선택). 이 순서는 어떤 완전 탐색 문제에서도 동일하다.

# 문제 16 · 객관식

완전 탐색 템플릿에서 '조건 검증' 단계가 코드로 나타나는 형태로 가장 적절한 것은?

① 후보가 문제 조건을 만족하는지 판단하는 if 문

② 가능한 경우를 만들어 내는 for 문

③ 최적값을 담아 두는 변수의 초기화

④ 결과를 모아 두는 리스트의 append 호출

⑤ 함수의 마지막 return 문

# 정답 및 해설 16

완전 탐색 템플릿에서 '조건 검증' 단계가 코드로 나타나는 형태로 가장 적절한 것은?

**정답 ① 후보가 문제 조건을 만족하는지 판단하는 if 문**

## 해설

생성은 반복문이, 검증은 조건문이, 선택은 반환이나 최적값 갱신이 담당한다. 조건문은 만들어진 후보를 걸러 내는 거름망 역할을 한다.

# 문제 17 · 객관식

완전 탐색에서 '해 선택' 단계가 정답 하나만 필요한 경우와 최적해가 필요한 경우에 각각 어떻게 달라지는가?

① 하나만 필요하면 계속 비교·갱신하고, 최적해가 필요하면 즉시 반환한다

② 하나만 필요하면 즉시 반환하고, 최적해가 필요하면 계속 비교·갱신한다

③ 두 경우 모두 반드시 끝까지 탐색을 마친 뒤에 결과를 반환한다

④ 두 경우 모두 첫 번째 후보를 만나는 순간 그것을 반환한다

⑤ 하나만 필요하면 후보 생성 단계를 통째로 건너뛸 수 있다

# 정답 및 해설 17

완전 탐색에서 '해 선택' 단계가 정답 하나만 필요한 경우와 최적해가 필요한 경우에 각각 어떻게 달라지는가?

**정답 ② 하나만 필요하면 즉시 반환하고, 최적해가 필요하면 계속 비교·갱신한다**

## 해설

조건을 만족하는 답 하나면 충분한 문제는 찾는 즉시 return 하여 남은 탐색을 생략할 수 있다. 최적해를 구해야 하면 모든 후보를 끝까지 보며 최적값을 갱신해야 한다.

# 문제 18 · 객관식

완전 탐색의 후보 해 생성 단계가 반드시 지켜야 할 성질은?

① 생성한 후보를 정렬된 순서로 유지해야 한다

② 정답에 가까운 후보부터 생성해야 한다

③ 빠짐없이 그리고 중복 없이 생성해야 한다

④ 생성 개수가 입력 크기보다 작아야 한다

⑤ 생성 도중 조건을 만족하면 즉시 멈춰야 한다

# 정답 및 해설 18

완전 탐색의 후보 해 생성 단계가 반드시 지켜야 할 성질은?

**정답 ③ 빠짐없이 그리고 중복 없이 생성해야 한다**

## 해설

하나라도 빠지면 정답을 놓칠 수 있어 정확성이 무너지고, 중복이 생기면 같은 경우를 여러 번 세어 결과가 틀어지거나 시간이 낭비된다. 생성 순서나 정렬 여부는 요구 사항이 아니다.

# 문제 19 · 객관식

완전 탐색 구조가 고급 알고리즘의 출발점이 된다고 말하는 이유로 가장 적절한 것은?

① 고급 기법들이 완전 탐색의 비효율을 줄이는 방향으로 발전했기 때문이다

② 완전 탐색이 어떤 문제에서든 가장 빠른 방법으로 알려져 있기 때문이다

③ 고급 기법들이 완전 탐색보다 구현하기 훨씬 간단하기 때문이다

④ 고급 기법들이 완전 탐색과 똑같은 시간 복잡도를 갖기 때문이다

⑤ 완전 탐색이 정답 대신 근사한 답만 구해 주기 때문이다

# 정답 및 해설 19

완전 탐색 구조가 고급 알고리즘의 출발점이 된다고 말하는 이유로 가장 적절한 것은?

**정답 ① 고급 기법들이 완전 탐색의 비효율을 줄이는 방향으로 발전했기 때문이다**

## 해설

백트래킹은 가지치기를, 분할 정복은 문제 분할을, 동적 계획법은 기억을 더해 완전 탐색의 낭비를 줄인 것이다. 기준점이 되는 구조가 완전 탐색이다.

# 문제 20 · 객관식

순열과 조합을 구분하는 기준으로 가장 적절한 것은?

① 같은 원소를 두 번 이상 뽑는 것이 허용되는가

② 뽑아야 하는 원소의 개수가 미리 정해져 있는가

③ 뽑은 결과에서 순서가 다르면 다른 경우로 볼 것인가

④ 원본 데이터가 오름차순으로 정렬되어 있는가

⑤ 결과를 리스트와 튜플 중 어느 쪽으로 저장하는가

# 정답 및 해설 20

순열과 조합을 구분하는 기준으로 가장 적절한 것은?

**정답 ③ 뽑은 결과에서 순서가 다르면 다른 경우로 볼 것인가**

## 해설

순서를 구별하면 순열, 구별하지 않으면 조합이다. 1등과 2등을 정하는 문제는 순열, 대표 두 명을 뽑는 문제는 조합이다.

# 문제 21 · 객관식

다음 코드의 출력값은?

code.py

```
1 from itertools import permutations
2 items = ['A', 'B', 'C']
3 result = list(permutations(items, 2))
4 print(len(result))
```

① 3

② 6

③ 2

④ 9

⑤ 8

# 정답 및 해설 21

다음 코드의 출력값은?

code.py

```
1 from itertools import permutations
2 items = ['A', 'B', 'C']
3 result = list(permutations(items, 2))
4 print(len(result))
```

**정답 ② 6**

## 해설

3개 중 2개를 순서 있게 나열하므로 3P2 = 3 × 2 = 6이다. (A,B)와 (B,A)를 서로 다른 경우로 세기 때문에 조합보다 개수가 많다.

# 문제 22 · 객관식

다음 코드의 출력값은?

code.py

```
1 from itertools import combinations
2 items = ['A', 'B', 'C']
3 result = list(combinations(items, 2))
4 print(len(result))
```

① 3

② 6

③ 2

④ 9

⑤ 4

# 정답 및 해설 22

다음 코드의 출력값은?

code.py

```
1 from itertools import combinations
2 items = ['A', 'B', 'C']
3 result = list(combinations(items, 2))
4 print(len(result))
```

**정답 ① 3**

## 해설

3개 중 2개를 순서 없이 고르므로 3C2 = 3이다. (A,B)와 (B,A)를 같은 경우로 보기 때문에 순열의 절반이 된다.

# 문제 23 · 객관식

다음 중 조합(combination)으로 세어야 하는 상황은?

① 5명 중에서 회장과 부회장을 각각 한 명씩 정하는 경우

② 5개 메뉴 중 서로 다른 2개를 골라 세트를 만드는 경우

③ 서로 다른 숫자 3개를 이어 붙여 비밀번호를 만드는 경우

④ 네 개의 도시를 한 번씩 방문하는 순서를 정하는 경우

⑤ 달리기 경기에서 1등, 2등, 3등을 각각 정하는 경우

# 정답 및 해설 23

다음 중 조합(combination)으로 세어야 하는 상황은?

**정답 ② 5개 메뉴 중 서로 다른 2개를 골라 세트를 만드는 경우**

**해설**

세트에서는 초코·바닐라와 바닐라·초코가 같은 메뉴이므로 순서를 구별하지 않는다. 나머지 넷은 모두 순서가 결과를 바꾸므로 순열이다.

# 문제 24 · 객관식

서로 다른 n개에서 r개를 뽑아 순서 있게 나열하는 경우의 수를 나타내는 식은?

① $r \times (r-1) \times \dots \times 2 \times 1$

② $n \times (n-1) \times \dots \times (n-r)$ 의 곱

③ $n! / (r! \times (n-r)!)$

④ $n \times n \times \dots \times n$ (n을 r번 곱함)

⑤ $n \times (n-1) \times \dots \times (n-r+1)$

# 정답 및 해설 24

서로 다른 n개에서 r개를 뽑아 순서 있게 나열하는 경우의 수를 나타내는 식은?

**정답 ⑤ $n \times (n-1) \times \dots \times (n-r+1)$**

## 해설

첫 자리에 n가지, 다음 자리에 n−1가지 식으로 r개를 채우므로 n부터 시작해 r개의 수를 곱한다. 두 번째 식은 조합의 개수다.

# 문제 25 · 객관식

itertools를 사용해 완전 탐색을 구현했을 때 얻는 효과로 옳은 것은?

① 조건을 만족하지 않는 후보를 자동으로 건너뛴다

② 탐색해야 할 경우의 수가 줄어든다

③ 시간 복잡도의 차수가 한 단계 낮아진다

④ 중첩 반복문을 짧은 코드로 대체할 수 있다

⑤ 메모리를 전혀 사용하지 않고 동작한다

# 정답 및 해설 25

itertools를 사용해 완전 탐색을 구현했을 때 얻는 효과로 옳은 것은?

**정답 ④ 중첩 반복문을 짧은 코드로 대체할 수 있다**

## 해설

itertools는 코드를 짧고 수정하기 쉽게 만들어 줄 뿐 내부적으로는 모든 경우를 그대로 생성한다. 경우의 수도 복잡도도 달라지지 않는다.

# 문제 26 · 객관식

숫자 1, 2, 3을 한 번씩 모두 사용해 만들 수 있는 세 자리 수의 개수는?

① 3개

② 6개

③ 9개

④ 27개

⑤ 12개

# 정답 및 해설 26

숫자 1, 2, 3을 한 번씩 모두 사용해 만들 수 있는 세 자리 수의 개수는?

**정답 ② 6개**

### 해설

세 개를 모두 사용해 순서 있게 나열하므로 3! = 3 × 2 × 1 = 6개다. permutations(nums, 3)으로 생성되는 후보의 개수와 같다.

# 문제 27 · 객관식

4가지 맛 중 서로 다른 2가지 맛을 골라 더블 컵을 만든다. 담는 순서를 구별하지 않을 때 가능한 메뉴는 몇 가지인가?

① 6가지

② 12가지

③ 8가지

④ 16가지

⑤ 4가지

# 정답 및 해설 27

4가지 맛 중 서로 다른 2가지 맛을 골라 더블 컵을 만든다. 담는 순서를 구별하지 않을 때 가능한 메뉴는 몇 가지인가?

**정답 ① 6가지**

**해설**

서로 다른 두 맛을 고르므로 4×3÷2=6가지다. 같은 맛을 두 번 고르는 경우는 제외한다.

# 문제 28 · 객관식

로또처럼 45개의 번호 중 6개를 순서 없이 뽑는 경우의 수를 구하는 식으로 옳은 것은?

① $45! / 39!$

② $45! / (6! \times 39!)$

③ $45^6$

④ $45 \times 44 \times 43 \times 42 \times 41 \times 40$

⑤ $6! / 45!$

# 정답 및 해설 28

로또처럼 45개의 번호 중 6개를 순서 없이 뽑는 경우의 수를 구하는 식으로 옳은 것은?

**정답 ② 45! / (6! × 39!)**

## 해설

조합의 개수는 n! / (r! × (n−r)!) 이므로 45! / (6! × 39!) = 8,145,060이다. 네 번째 식은 순서를 구별한 45P6으로 이 값의 720배다.

# 문제 29 · 객관식

n개 원소로 만들 수 있는 순열의 개수가 n! 이라는 것이 의미하는 바로 가장 적절한 것은?

① 후보의 수가 n의 값과 무관하게 항상 일정하게 유지된다

② n이 아무리 커져도 후보의 수가 완만하게만 늘어난다

③ 후보의 수가 n의 크기에 정확히 정비례해서 늘어난다

④ 후보의 수가 n의 제곱에 비례하는 속도로 늘어난다

⑤ n이 조금만 커져도 후보의 수가 감당할 수 없이 늘어난다

# 정답 및 해설 29

n개 원소로 만들 수 있는 순열의 개수가 n! 이라는 것이 의미하는 바로 가장 적절한 것은?

**정답 ⑤ n이 조금만 커져도 후보의 수가 감당할 수 없이 늘어난다**

## 해설

10! 은 약 360만이지만 20! 은 약 $2.4 \times 10^{18}$ 이다. 팩토리얼은 이 장에서 다루는 증가 유형 중 가장 빠르며, 순열을 전부 생성하는 방식은 n이 10을 넘기면 곧 한계에 부딪힌다.

# 문제 30 · 객관식

0부터 9까지의 서로 다른 숫자 3개로 만드는 비밀번호의 개수는?

① 504개

② 1,000개

③ 120개

④ 30개

⑤ 720개

# 정답 및 해설 30

0부터 9까지의 서로 다른 숫자 3개로 만드는 비밀번호의 개수는?

**정답 ⑤ 720개**

## 해설

서로 다른 세 숫자를 순서 있게 배치하므로 10P3 = 10 × 9 × 8 = 720개다. 중복을 허용했다면 $10^3$ = 1,000개가 된다.

# 문제 31 · 객관식

길이 n인 텍스트에서 길이 m인 패턴을 찾을 때, 비교를 시작할 수 있는 위치의 개수는?

① n + m − 1개

② n × m − 1개

③ n − m + 1개

④ n − m − 1개

⑤ n × (m − 1)개

# 정답 및 해설 31

길이 n인 텍스트에서 길이 m인 패턴을 찾을 때, 비교를 시작할 수 있는 위치의 개수는 ?

**정답 ③ n − m + 1개**

## 해설

패턴이 텍스트를 벗어나지 않으려면 시작 위치가 0부터 n−m까지여야 하므로 n−m+1개다. n=11, m=3이면 9개의 시작 위치를 검사한다.

# 문제 32 · 객관식

다음 코드를 text = "ababcabcababd", pattern = "abc" 로 실행한 결과는?

code.py

```
def bf_match(text, pattern):
    n, m = len(text), len(pattern)
    found = []
    for i in range(n - m + 1):
        ok = True
        for j in range(m):
            if text[i + j] != pattern[j]:
                ok = False; break
        if ok: found.append(i)
    return found
```

① [5, 8]

② [0, 2, 5]

③ [2]

④ [2, 5]

⑤ []

# 정답 및 해설 32

다음 코드를 text = "ababcabcababd", pattern = "abc" 로 실행한 결과는?

code.py

```
def bf_match(text, pattern):
    n, m = len(text), len(pattern)
    found = []
    for i in range(n - m + 1):
        ok = True
        for j in range(m):
            if text[i + j] != pattern[j]:
                ok = False; break
        if ok: found.append(i)
    return found
```

**정답** ④ **[2, 5]**

### 해설

인덱스 2에서 abc, 인덱스 5에서 abc가 일치한다. 인덱스 0은 aba이므로 세 번째 문자에서 어긋난다.

# 문제 33 · 객관식

무차별 문자열 매칭의 최악의 경우 시간 복잡도는? (텍스트 길이 n, 패턴 길이 m)

① O(n / m)

② O(n + m)

③ O(n)

④ O(m)

⑤ O(n × m)

# 정답 및 해설 33

무차별 문자열 매칭의 최악의 경우 시간 복잡도는? (텍스트 길이 n, 패턴 길이 m)

**정답 ⑤ O(n × m)**

## 해설

시작 위치가 최대 n−m+1개이고 각 위치에서 최대 m번 비교하므로 곱해서 O(n×m)이다. KMP가 O(n+m), 보이어-무어가 평균 O(n/m)으로 이를 개선한 알고리즘이다.

# 문제 34 · 객관식

다음 코드에서 break 문이 하는 역할로 가장 적절한 것은?

code.py

```
def bf_match(text, pattern):
    n, m = len(text), len(pattern)
    found = []
    for i in range(n - m + 1):
        ok = True
        for j in range(m):
            if text[i + j] != pattern[j]:
                ok = False; break
        if ok: found.append(i)
    return found
```

① 한 글자라도 어긋나면 남은 비교를 생략하고 다음 시작 위치로 넘어간다

② 패턴을 찾으면 함수 전체를 즉시 끝내고 지금까지의 결과를 반환한다

③ 패턴 길이가 텍스트보다 길 때 발생하는 인덱스 오류를 막아 준다

④ 텍스트의 마지막 글자에 도달하면 바깥 반복문까지 함께 멈춘다

⑤ 이미 찾은 시작 위치를 다시 검사하지 않도록 건너뛰게 한다

# 정답 및 해설 34

다음 코드에서 break 문이 하는 역할로 가장 적절한 것은?

code.py

```
def bf_match(text, pattern):
    n, m = len(text), len(pattern)
    found = []
    for i in range(n - m + 1):
        ok = True
        for j in range(m):
            if text[i + j] != pattern[j]:
                ok = False; break
        if ok: found.append(i)
    return found
```

**정답 ① 한 글자라도 어긋나면 남은 비교를 생략하고 다음 시작 위치로 넘어간다**

## 해설

안쪽 반복문의 break이므로 그 시작 위치에 대한 비교만 중단된다. 한 글자만 달라도 그 위치는 이미 실패이므로 나머지를 볼 필요가 없다. 함수를 끝내는 것은 return이지 break가 아니다.

# 문제 35 · 객관식

다음 코드를 arr = [10, 3, 22, 15, 17, 9] 로 실행했을 때 반환되는 최소 차이는?

code.py

```
def closest_pair(arr):
    min_diff = float('inf')
    best = None
    for i in range(len(arr) - 1):
        for j in range(i + 1, len(arr)):
            diff = abs(arr[i] - arr[j])
            if diff < min_diff:
                min_diff = diff
                best = (arr[i], arr[j])
    return best, min_diff
```

① 7

② 2

③ 3

④ 5

⑤ 1

# 정답 및 해설 35

다음 코드를 arr = [10, 3, 22, 15, 17, 9] 로 실행했을 때 반환되는 최소 차이는?

code.py

```python
def closest_pair(arr):
    min_diff = float('inf')
    best = None
    for i in range(len(arr) - 1):
        for j in range(i + 1, len(arr)):
            diff = abs(arr[i] - arr[j])
            if diff < min_diff:
                min_diff = diff
                best = (arr[i], arr[j])
    return best, min_diff
```

**정답 ⑤ 1**

**해설**

10과 9의 차이가 1로 가장 작다. 15와 17의 차이 2가 그다음이다. 모든 쌍을 검사하는 $O(n^2)$ 방식이며 분할 정복을 적용하면 $O(n \log n)$까지 개선된다.

# 문제 36 · 객관식

다음 코드에서 min_diff를 float('inf')로 초기화한 이유로 가장 적절한 것은?

code.py

```
def closest_pair(arr):
    min_diff = float('inf')
    best = None
    for i in range(len(arr) - 1):
        for j in range(i + 1, len(arr)):
            diff = abs(arr[i] - arr[j])
            if diff < min_diff:
                min_diff = diff
                best = (arr[i], arr[j])
    return best, min_diff
```

① 정렬되지 않은 리스트도 처리할 수 있게 하기 위해

② 차이를 정수가 아닌 실수형으로 계산하도록 하기 위해

③ 두 수의 차이가 음수가 되는 경우를 미리 막기 위해

④ 리스트가 비어 있을 때 생기는 오류를 피하기 위해

⑤ 첫 번째로 계산되는 차이가 반드시 갱신되도록 하기 위해

# 정답 및 해설 36

다음 코드에서 min_diff를 float('inf')로 초기화한 이유로 가장 적절한 것은?

code.py

```
def closest_pair(arr):
    min_diff = float('inf')
    best = None
    for i in range(len(arr) - 1):
        for j in range(i + 1, len(arr)):
            diff = abs(arr[i] - arr[j])
            if diff < min_diff:
                min_diff = diff
                best = (arr[i], arr[j])
    return best, min_diff
```

**정답 ⑤ 첫 번째로 계산되는 차이가 반드시 갱신되도록 하기 위해**

### 해설

0으로 초기화하면 어떤 차이도 그보다 작지 않아 갱신이 일어나지 않는다. 최소값을 구할 때는 가능한 어떤 값보다 큰 값으로, 최대값을 구할 때는 반대로 초기화한다.

# 문제 37 · 객관식

원소가 n개인 집합의 부분집합 개수는? (공집합과 자기 자신 포함)

① $n^2$개

② $2^n$개

③ n!개

④ 2n개

⑤ $n(n-1)/2$개

# 정답 및 해설 37

원소가 n개인 집합의 부분집합 개수는? (공집합과 자기 자신 포함)

**정답 ② $2^n$개**

**해설**

각 원소마다 포함하거나 포함하지 않는 두 가지 선택이 있으므로 2를 n번 곱한다. 원소 3개면 8개, 10개면 1,024개, 20개면 약 100만 개다.

# 문제 38 · 객관식

다음 코드에서 arr = ['A', 'B', 'C'] 이고 i = 5 일 때 만들어지는 부분집합은?

code.py

```
def power_set(arr):
    n = len(arr)
    subsets = []
    for i in range(1 << n):
        cur = []
        for j in range(n):
            if i & (1 << j):
                cur.append(arr[j])
        subsets.append(cur)
    return subsets
```

① ['A', 'B', 'C']

② ['A', 'B']

③ ['B', 'C']

④ ['A', 'C']

⑤ ['C']

# 정답 및 해설 38

다음 코드에서 arr = ['A', 'B', 'C'] 이고 i = 5 일 때 만들어지는 부분집합은?

code.py

```
def power_set(arr):
    n = len(arr)
    subsets = []
    for i in range(1 << n):
        cur = []
        for j in range(n):
            if i & (1 << j):
                cur.append(arr[j])
        subsets.append(cur)
    return subsets
```

**정답 ④ ['A', 'C']**

**해설**

5를 이진수로 쓰면 101이다. 0번 비트와 2번 비트가 1이므로 arr[0]인 A와 arr[2]인 C가 선택된다.

# 문제 39 · 객관식

부분집합 생성 코드에서 1 << n 이 나타내는 값은?

① n / 2

② $n^2$

③ $2^n$

④ 2n

⑤ n!

# 정답 및 해설 39

부분집합 생성 코드에서 1 << n 이 나타내는 값은?

**정답 ③ $2^n$**

**해설**

1을 왼쪽으로 n비트 이동하면 2를 n번 곱한 것과 같다. 따라서 range(1 << n)은 0부터 $2^n-1$까지를 순회하며 모든 부분집합에 하나씩 대응한다.

# 문제 40 · 객관식

조합적 폭발(combinatorial explosion)을 가장 잘 설명한 것은?

① 반복문 안에서 예외가 연쇄적으로 발생해 프로그램이 멈추어 버리는 현상

② 입력이 커질수록 프로그램이 사용하는 메모리가 일정하게 늘어나는 현상

③ 확인해야 할 경우의 수가 입력 크기에 정비례해 완만하게 늘어나는 현상

④ 입력이 조금만 커져도 확인할 경우의 수가 감당할 수 없이 늘어나는 현상

⑤ 계산 결과의 자릿수가 넘쳐 값이 잘못 저장되어 버리는 현상

# 정답 및 해설 40

조합적 폭발(combinatorial explosion)을 가장 잘 설명한 것은?

**정답 ④ 입력이 조금만 커져도 확인할 경우의 수가 감당할 수 없이 늘어나는 현상**

**해설**

n이 20 정도만 되어도 $2^n$은 약 100만, n! 은 약 $2.4 \times 10^{18}$ 이 된다. 완전 탐색이 실패하는 거의 모든 경우가 이 현상 때문이다.

# 문제 41 · 객관식

n = 20 일 때 값이 가장 큰 것은?

① $2^n$

② $n!$

③ $n^2$

④ $n^3$

⑤ $n \log n$

# 정답 및 해설 41

n = 20 일 때 값이 가장 큰 것은?

**정답 ② n!**

### 해설

20! 은 약 $2.43 \times 10^{18}$, $2^{20}$ 은 약 100만, $20^2$ 은 400, $20^3$ 은 8,000이다. 팩토리얼이 압도적으로 크며 이 장에서 다루는 세 가지 폭발 유형 중 가장 가파르다.

# 문제 42 · 객관식

다음 중 중첩 반복문에서 나타나는 다항 시간 증가 유형은?

① $O(2^n)$

② $O(n^k)$

③ $O(n!)$

④ $O(n^n)$

⑤ $O(\log n)$

# 정답 및 해설 42

다음 중 중첩 반복문에서 나타나는 다항 시간 증가 유형은?

**정답 ② $O(n^k)$**

**해설**

반복문을 k겹 중첩하면 $O(n^k)$이 된다. $2^n$은 포함·비포함을 반복하는 선택 구조에서, $n!$ 은 순서를 나열하는 구조에서 나타난다.

# 문제 43 · 객관식

부분집합을 만드는 문제에서 $O(2^n)$이 나오는 근본적인 이유는?

① 부분집합의 크기가 원소마다 달라지기 때문이다

② 원소를 순서대로 나열하는 방법이 여러 가지이기 때문이다

③ 반복문을 n겹으로 중첩해야 하기 때문이다

④ 각 원소를 다른 모든 원소와 한 번씩 비교하기 때문이다

⑤ 원소마다 포함할지 말지 두 갈래의 선택이 반복되기 때문이다

# 정답 및 해설 43

부분집합을 만드는 문제에서 $O(2^n)$이 나오는 근본적인 이유는?

**정답 ⑤ 원소마다 포함할지 말지 두 갈래의 선택이 반복되기 때문이다**

## 해설

원소 하나가 늘 때마다 경우의 수가 두 배가 된다. 시작이 1가지, 원소 1개면 2가지, 2개면 4가지, n개면 $2^n$가지가 되는 구조다.

# 문제 44 · 객관식

서로 다른 원소 10개를 모두 나열하는 순열은 10!개다. 원소가 11개가 되면 순열 수는 몇 배가 되는가?

① 2배

② 11배

③ 10배

④ 100배

⑤ 121배

# 정답 및 해설 44

서로 다른 원소 10개를 모두 나열하는 순열은 10!개다. 원소가 11개가 되면 순열 수는 몇 배가 되는가?

**정답 ② 11배**

### 해설

11!=11×10!이므로 11배다. 10!의 십진수 값을 외우거나 계산할 필요는 없다.

# 문제 45 · 객관식

완전 탐색으로 비밀번호를 찾을 때, 자릿수가 하나 늘어나면 경우의 수는 어떻게 변하는가? (각 자리 0~9)

① 2배가 된다

② 10배가 된다

③ 10이 더해진다

④ 제곱이 된다

⑤ 변하지 않는다

# 정답 및 해설 45

완전 탐색으로 비밀번호를 찾을 때, 자릿수가 하나 늘어나면 경우의 수는 어떻게 변하는가? (각 자리 0~9)

**정답 ② 10배가 된다**

## 해설

자리마다 10가지 선택이 곱해지므로 자릿수가 하나 늘 때마다 10배가 된다. 4자리 1만, 6자리 100만, 8자리 1억으로 늘어난다.

# 문제 46 · 객관식

도시가 n개인 외판원 순회 문제에서 모든 방문 순서를 확인하는 완전 탐색의 시간 복잡도는?

① $O(n^2)$

② $O(2^n)$

③ $O(n!)$

④ $O(n^3)$

⑤ $O(n \log n)$

# 정답 및 해설 46

도시가 n개인 외판원 순회 문제에서 모든 방문 순서를 확인하는 완전 탐색의 시간 복잡도는?

**정답 ③ O(n!)**

## 해설

모든 도시를 한 번씩 방문하는 순서를 전부 나열하는 것이므로 순열의 개수만큼 확인해야 한다. 도시 4개면 24가지, 10개면 약 360만 가지, 20개면 약 $2.4 \times 10^{18}$ 가지다.

# 문제 47 · 객관식

서로 다른 도시 4개의 방문 순서를 모두 나열한다. 출발 도시는 고정하지 않으며, 역순도 다른 순서로 센다. 가능한 순서는 몇 가지인가?

① 64가지

② 16가지

③ 12가지

④ 24가지

⑤ 4가지

# 정답 및 해설 47

서로 다른 도시 4개의 방문 순서를 모두 나열한다. 출발 도시는 고정하지 않으며, 역순도 다른 순서로 센다. 가능한 순서는 몇 가지인가?

**정답 ④ 24가지**

### 해설

첫 도시 4가지, 다음 3가지, 다음 2가지, 마지막 1가지이므로 4!=24가지다.

# 문제 48 · 객관식

n = 20 인 순열을 모두 생성하는 계산이 현실적으로 불가능한 이유로 가장 적절한 것은?

① 메모리 용량이 순열 하나를 담기에 부족하기 때문이다

② 초당 1억 번을 계산해도 수백 년이 걸리기 때문이다

③ 파이썬의 재귀 깊이 제한에 걸리기 때문이다

④ 정수 자료형의 범위를 넘어서기 때문이다

⑤ 순열을 생성하는 함수가 20개까지만 지원하기 때문이다

# 정답 및 해설 48

n = 20 인 순열을 모두 생성하는 계산이 현실적으로 불가능한 이유로 가장 적절한 것은?

**정답 ② 초당 1억 번을 계산해도 수백 년이 걸리기 때문이다**

## 해설

20! 은 약 $2.43 \times 10^{18}$ 이고, 초당 1억 번을 처리해도 약 770년이 걸린다. 메모리나 자료형이 아니라 계산량 자체가 문제다.

# 문제 49 · 객관식

완전 탐색의 가능 여부를 판단할 때 가장 먼저 확인해야 할 것은?

① 탐색해야 할 경우의 수를 계산하는 일

② 사용할 반복문의 개수를 정하는 일

③ 입력 데이터가 정렬되어 있는지 확인하는 일

④ 재귀와 반복 중 어느 쪽으로 구현할지 정하는 일

⑤ 출력 형식이 무엇인지 확인하는 일

# 정답 및 해설 49

완전 탐색의 가능 여부를 판단할 때 가장 먼저 확인해야 할 것은?

## 정답 ① 탐색해야 할 경우의 수를 계산하는 일

### 해설

중요한 것은 입력 크기 n 자체가 아니라 그로부터 만들어지는 탐색 공간의 크기다. n이 20이어도 O(n²)이면 400에 불과하지만 O(n!)이면 계산이 불가능하다.

# 문제 50 · 객관식

완전 탐색에 가지치기(pruning)를 더한 기법은?

① 동적 계획법

② 분할 정복

③ 백트래킹

④ 그리디 알고리즘

⑤ 이진 탐색

# 정답 및 해설 50

완전 탐색에 가지치기(pruning)를 더한 기법은?

## 정답 ③ 백트래킹

### 해설

정답이 될 수 없다고 판단되는 가지를 미리 잘라 내어 탐색량을 줄이는 것이 백트래킹이다. N-Queens와 스도쿠가 대표적인 적용 사례다.

# 문제 51 · 객관식

완전 탐색에 기억(메모이제이션)을 더한 기법은?

① 완전 탐색

② 백트래킹

③ 분할 정복

④ 동적 계획법

⑤ 선형 탐색

# 정답 및 해설 51

완전 탐색에 기억(메모이제이션)을 더한 기법은?

## 정답 ④ 동적 계획법

### 해설

한 번 계산한 부분 문제의 결과를 저장해 두었다가 재사용한다. 피보나치 수열이 완전 탐색으로는 $O(2^n)$이지만 메모이제이션을 쓰면 $O(n)$이 되는 것이 대표적인 예다.

# 문제 52 · 객관식

양의 정수들에서 합이 10인 부분집합을 찾는다. 원소를 하나씩 고를 때 백트래킹으로 불필요한 탐색을 줄일 수 있는 이유는?

① 더하다가 이미 10을 넘으면 그 갈래를 더 이상 보지 않기 때문이다

② 부분집합을 만들지 않고 원소의 합만 따로 계산해서 비교하기 때문이다

③ 한 번 계산해 둔 합을 저장했다가 필요할 때 그대로 다시 쓰기 때문이다

④ 부분집합을 크기가 작은 것부터 차례대로 만들어 보고 멈추기 때문이다

⑤ 만들어야 하는 부분집합의 개수 자체가 $2^n$보다 훨씬 적기 때문이다

# 정답 및 해설 52

양의 정수들에서 합이 10인 부분집합을 찾는다. 원소를 하나씩 고를 때 백트래킹으로 불필요한 탐색을 줄일 수 있는 이유는?

**정답 ① 더하다가 이미 10을 넘으면 그 갈래를 더 이상 보지 않기 때문이다**

## 해설

원소가 모두 양수라서 합이 10을 넘으면 원소를 더 골라도 10으로 줄어들지 않는다. 따라서 그 갈래를 중단할 수 있다.

# 문제 53 · 객관식

다음 코드가 완전 탐색 방식의 피보나치보다 빠른 이유는?

code.py

```
memo = {}
def fib(n):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib(n-1) + fib(n-2)
    return memo[n]
```

① 덧셈 대신 곱셈을 사용해 계산량을 줄였기 때문이다

② 계산 도중 가망이 없는 값을 미리 걸러 내고 넘어가기 때문이다

③ 재귀 호출을 모두 반복문으로 바꾸어 구현했기 때문이다

④ 문제를 절반씩 나누어 따로 계산한 뒤 합치기 때문이다

⑤ 이미 계산한 값을 저장해 두고 다시 계산하지 않기 때문이다

# 정답 및 해설 53

다음 코드가 완전 탐색 방식의 피보나치보다 빠른 이유는?

code.py

```
memo = {}
def fib(n):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib(n-1) + fib(n-2)
    return memo[n]
```

**정답 ⑤ 이미 계산한 값을 저장해 두고 다시 계산하지 않기 때문이다**

## 해설

정의를 그대로 재귀로 옮기면 F(3)과 F(2)가 여러 번 다시 계산되어 $O(2^n)$이 된다. memo에 결과를 저장하면 각 값을 한 번씩만 계산하므로 O(n)이 된다.

# 문제 54 · 객관식

전화번호부에서 이름을 찾는 두 방법 중 완전 탐색에 해당하는 것과 그 시간 복잡도를 바르게 짝지은 것은?

① 첫 페이지부터 한 줄씩 확인 — $O(\log n)$

② 중간을 펼쳐 절반씩 버림 — $O(\log n)$

③ 첫 페이지부터 한 줄씩 확인 — $O(n)$

④ 중간을 펼쳐 절반씩 버림 — $O(n)$

⑤ 첫 페이지부터 한 줄씩 확인 — $O(n^2)$

# 정답 및 해설 54

전화번호부에서 이름을 찾는 두 방법 중 완전 탐색에 해당하는 것과 그 시간 복잡도를 바르게 짝지은 것은?

**정답 ③ 첫 페이지부터 한 줄씩 확인 — O(n)**

## 해설

완전 탐색에 해당하는 것은 처음부터 끝까지 훑는 선형 검색으로 O(n)이다. 절반씩 버리는 이진 탐색은 분할 정복의 사고가 적용된 방식으로 O(log n)이다.

# 문제 55 · 객관식

알고리즘 설계에서 완전 탐색을 먼저 구현해 보라고 권하는 이유로 가장 적절한 것은?

① 완전 탐색이 다른 어떤 기법보다 메모리를 적게 쓰기 때문이다

② 완전 탐색은 입력 크기와 무관하게 항상 같은 속도로 동작하기 때문이다

③ 완전 탐색은 코드가 길어져서 오히려 실수를 줄여 주기 때문이다

④ 완전 탐색이 대부분의 문제에서 가장 빠른 방법이기 때문이다

⑤ 문제의 구조를 파악하고 정답을 확인할 기준을 얻을 수 있기 때문이다

# 정답 및 해설 55

알고리즘 설계에서 완전 탐색을 먼저 구현해 보라고 권하는 이유로 가장 적절한 것은?

**정답 ⑤ 문제의 구조를 파악하고 정답을 확인할 기준을 얻을 수 있기 때문이다**

## 해설

느리더라도 확실히 맞는 답을 주므로, 최적화한 코드의 결과를 검증하는 기준으로 쓸 수 있고 문제의 구조도 드러난다. 그다음에 어디를 줄일지 고민하는 것이 올바른 순서다.

# 문제 56 · 객관식

코딩 테스트에서 입력이 많을 때 input() 대신 sys.stdin.readline 을 쓰는 이유는?

① 잘못된 입력이 들어오면 예외를 발생시키기 때문이다

② 입력값의 자료형을 자동으로 변환해 주기 때문이다

③ 줄 끝의 개행 문자를 자동으로 없애 주기 때문이다

④ 여러 줄을 한 번에 리스트로 만들어 주기 때문이다

⑤ 입력을 읽는 속도가 훨씬 빠르기 때문이다

# 정답 및 해설 56

코딩 테스트에서 입력이 많을 때 input() 대신 sys.stdin.readline 을 쓰는 이유는?

**정답 ⑤ 입력을 읽는 속도가 훨씬 빠르기 때문이다**

## 해설

대량 입력에서 input()은 느려 시간 초과의 원인이 된다. sys.stdin.readline은 개행 문자를 그대로 남기므로 문자열을 다룰 때는 strip()을 따로 붙여야 한다.

# 문제 57 · 객관식

다음 코드의 출력값은?

code.py

```
1 from itertools import combinations
2 arr = [1, 2, 3, 4, 5]
3 target = 9
4 count = 0
5 for c in combinations(arr, 3):
6     if sum(c) == target:
7         count += 1
8 print(count)
```

① 5

② 1

③ 3

④ 2

⑤ 10

# 정답 및 해설 57

다음 코드의 출력값은?

code.py

```
from itertools import combinations
arr = [1, 2, 3, 4, 5]
target = 9
count = 0
for c in combinations(arr, 3):
    if sum(c) == target:
        count += 1
print(count)
```

**정답 ④ 2**

## 해설

1부터 5까지에서 서로 다른 세 수를 골라 합이 9가 되는 경우는 (1,3,5)와 (2,3,4) 두 가지다. combinations는 순서를 구별하지 않으므로 (3,1,5) 같은 것은 따로 세지 않는다.

# 문제 58 · 객관식

다음 코드의 출력값은?

code.py

```
1 from itertools import combinations
2 def blackjack(cards, M):
3     best = 0
4     for c in combinations(cards, 3):
5         t = sum(c)
6         if t <= M and t > best:
7             best = t
8     return best
9 print(blackjack([5, 6, 7, 8, 9], 21))
```

① 22

② 20

③ 18

④ 24

⑤ 21

# 정답 및 해설 58

다음 코드의 출력값은?

code.py

```
from itertools import combinations
def blackjack(cards, M):
    best = 0
    for c in combinations(cards, 3):
        t = sum(c)
        if t <= M and t > best:
            best = t
    return best
print(blackjack([5, 6, 7, 8, 9], 21))
```

**정답 ⑤ 21**

## 해설

세 장의 합이 21을 넘지 않으면서 가장 큰 경우를 찾는다. 5+7+9 = 21이 조건을 만족하는 최대값이다. 6+8+9 = 23은 21을 넘으므로 제외된다.

# 문제 59 · 객관식

서로 다른 원소 6개에서 순서 없이 3개를 고른다. 같은 원소는 두 번 고르지 않는다. 확인할 조합은 몇 가지인가?

① 6가지

② 18가지

③ 120가지

④ 20가지

⑤ 216가지

# 정답 및 해설 59

서로 다른 원소 6개에서 순서 없이 3개를 고른다. 같은 원소는 두 번 고르지 않는다. 확인할 조합은 몇 가지인가?

**정답 ④ 20가지**

**해설**

6×5×4를 같은 세 원소의 나열 순서 3×2×1로 나누면 20가지다.

# 문제 60 · 객관식

다음 코드에서 회문인지 판단하는 s[::-1]의 의미는?

code.py

```
1 s = input().strip()
2 if s == s[::-1]:
3     print('회문')
4 else:
5     print('아님')
```

① 문자열에서 첫 글자를 제외한 나머지

② 문자열의 마지막 한 글자

③ 문자열을 거꾸로 뒤집은 결과

④ 문자열을 한 글자씩 나눈 리스트

⑤ 문자열의 길이에서 1을 뺀 값

# 정답 및 해설 60

다음 코드에서 회문인지 판단하는 s[::-1]의 의미는?

code.py

```
1 s = input().strip()
2 if s == s[::-1]:
3     print('회문')
4 else:
5     print('아님')
```

**정답 ③ 문자열을 거꾸로 뒤집은 결과**

**해설**

슬라이싱의 세 번째 값이 −1이면 뒤에서부터 한 칸씩 이동하며 읽으므로 문자열이 뒤집힌다. 뒤집은 결과가 원래와 같으면 앞뒤로 읽어도 같은 회문이다.

# 네 자리 자물쇠 탐색

네 자리 숫자 문자열 secret을 0000부터 순서대로 찾는 함수를 작성하시오. 네 중첩 반복문을 사용하고, 찾을 때까지 시도한 횟수를 반환하시오. 0000도 첫 시도이며 앞자리 0을 보존한다.

입력 `lock_attempts("8234")`
결과 `8235`

## 작성할 내용

탐색 순서와 종료 조건을 정하고 Python 또는 C로 함수를 작성하시오.
예시 입력의 처리 과정을 보이고, 경계 조건 하나와 시간 복잡도를 설명하시오.

`return = 시도 횟수`

# 네 자리 자물쇠 탐색

```python
def lock_attempts(secret):
    count = 0
    for a in range(10):
        for b in range(10):
            for c in range(10):
                for d in range(10):
                    count += 1
                    if f"{a}{b}{c}{d}" == secret:
                        return count
```

고정 네 자리: 최악 10,000개 후보.

# 네 자리 자물쇠 탐색

필요한 표준 헤더: <stdlib.h> <string.h> <limits.h> / 결과 배열은 호출자가 준비

```c
int lock_attempts(const char secret[]) {
  int count = 0;
  for (int a = 0; a < 10; a++)
    for (int b = 0; b < 10; b++)
      for (int c = 0; c < 10; c++)
        for (int d = 0; d < 10; d++) {
          char attempt[5] = {'0'+a, '0'+b, '0'+c, '0'+d, '\0'};
          count++;
          if (strcmp(attempt, secret) == 0) return count;
        }
  return 0;
}
```

고정 네 자리: 최악 10,000개 후보.

# 개념 및 계산

0~9로 이루어진 4자리 자물쇠를 0000부터 순서대로 모두 시도할 때, 마지막까지 갔다면 총 몇 회 시도한 것인지 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

0~9로 이루어진 4자리 자물쇠를 0000부터 순서대로 모두 시도할 때, 마지막까지 갔다면 총 몇 회 시도한 것인지 쓰시오.

## 정답 10,000회

### 해설

0000부터 9999까지 빠짐없이 시도하면 $10^4 = 10{,}000$회다. 정답을 찾는 즉시 멈추지 않고 끝까지 돌린 경우의 횟수다.

# 개념 및 계산

4자리 자물쇠를 초당 1,000회 시도할 수 있을 때, 최악의 경우 걸리는 시간을 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

4자리 자물쇠를 초당 1,000회 시도할 수 있을 때, 최악의 경우 걸리는 시간을 쓰시오.

## 정답 10초

### 해설

10,000 ÷ 1,000 = 10초다. 같은 속도로 8자리를 시도하면 1억 ÷ 1,000 = 10만 초, 약 28시간이 걸린다.

# 개념 및 계산

완전 탐색에서 '탐색 공간'이라는 말이 가리키는 것을 한 문장으로 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

완전 탐색에서 '탐색 공간'이라는 말이 가리키는 것을 한 문장으로 쓰시오.

## 정답 알고리즘이 확인할 모든 후보의 집합

### 해설

실행 시간은 후보 수와 후보당 처리 비용에 좌우된다. 코드를 작성하기 전에 두 값을 함께 살펴본다.

# 합이 K인 두 수의 목록

배열 A에서 i<j이며 A[i]+A[j]=target인 모든 값 쌍을 구하는 함수를 작성하시오. i, j가 증가하는 순서로 저장한다. 값이 같아도 인덱스가 다른 쌍은 별개다. 0≤n≤30, 값과 target은 −100~100이다.

입력 `find_pairs([1,3,5,7,9], 10)`
결과 `[[1,9],[3,7]]`

## 작성할 내용

탐색 순서와 종료 조건을 정하고 Python 또는 C로 함수를 작성하시오.
예시 입력의 처리 과정을 보이고, 경계 조건 하나와 시간 복잡도를 설명하시오.

`out[][2] = 값의 쌍, return = 행 수 (최대 435)`

# 합이 K인 두 수의 목록

```python
def find_pairs(A, target):
    result = []
    for i in range(len(A)):
        for j in range(i + 1, len(A)):
            if A[i] + A[j] == target:
                result.append([A[i], A[j]])
    return result
```

시간 $O(n^2)$.

# 합이 K인 두 수의 목록

필요한 표준 헤더: <stdlib.h> <string.h> <limits.h> / 결과 배열은 호출자가 준비

```c
int find_pairs(const int A[], int n, int target, int out[][2]) {
  int count = 0;
  for (int i = 0; i < n; i++)
    for (int j = i + 1; j < n; j++)
      if (A[i] + A[j] == target) {
        out[count][0] = A[i]; out[count++][1] = A[j];
      }
  return count;
}
```

시간 $O(n^2)$.

# 개념 및 계산

주사위 세 개를 던져 나온 눈의 합이 4가 되는 경우의 수를 쓰시오. (주사위는 서로 구별한다)

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

주사위 세 개를 던져 나온 눈의 합이 4가 되는 경우의 수를 쓰시오. (주사위는 서로 구별한다)

## 정답 3가지

### 해설

(1,1,2), (1,2,1), (2,1,1) 세 가지다. 삼중 반복문으로 216가지를 모두 만든 뒤 조건을 통과한 것만 세면 이 값이 나온다.

# 개념 및 계산

길이가 1,000인 리스트에서 서로 다른 두 원소의 모든 쌍을 검사하면 대략 몇 번 비교하게 되는지 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

길이가 1,000인 리스트에서 서로 다른 두 원소의 모든 쌍을 검사하면 대략 몇 번 비교하게 되는지 쓰시오.

**정답 약 50만 번 (1000 × 999 ÷ 2 = 499,500)**

## 해설

$n(n-1)/2$ 이므로 약 $n^2/2$ 이다. n이 10배가 되면 비교 횟수는 약 100배가 된다.

# 개념 및 계산

n≥2이다. i는 0부터 n−2까지, j는 i+1부터 n−1까지 순회한다. j의 범위를 0부터 n−1까지로 바꾸면 검사 횟수는 몇 배가 되는가?

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

n≥2이다. i는 0부터 n−2까지, j는 i+1부터 n−1까지 순회한다. j의 범위를 0부터 n−1까지로 바꾸면 검사 횟수는 몇 배가 되는가?

## 정답 정확히 2배

### 해설

원래는 n(n−1)/2회, 변경 후에는 (n−1)n회이다. 바깥 반복문의 범위는 그대로이므로 비율은 정확히 2이다.

# 블랙잭 최적 조합

양의 카드 값 A에서 서로 다른 세 장을 골라 합이 target 이하인 최대 합을 반환하는 함수를 작성하시오. 가능한 조합이 없으면 0이다. 0≤n≤100, 카드 값은 1~1,000, 0≤target≤3,000이다.

입력 `blackjack([5,6,7,8,9], 21)`
결과 `21`

`return = 최대 합 또는 0`

## 작성할 내용

탐색 순서와 종료 조건을 정하고 Python 또는 C로 함수를 작성하시오.
예시 입력의 처리 과정을 보이고, 경계 조건 하나와 시간 복잡도를 설명하시오.

# 블랙잭 최적 조합

```python
def blackjack(A, target):
    best = 0
    for i in range(len(A)):
        for j in range(i + 1, len(A)):
            for k in range(j + 1, len(A)):
                total = A[i] + A[j] + A[k]
                if total <= target and total > best:
                    best = total
    return best
```

시간 $O(n^3)$.

# 블랙잭 최적 조합

필요한 표준 헤더: <stdlib.h> <string.h> <limits.h> / 결과 배열은 호출자가 준비

```c
int blackjack(const int A[], int n, int target) {
  int best=0;
  for(int i=0;i<n;i++) for(int j=i+1;j<n;j++) for(int k=j+1;k<n;k++) {
    int total=A[i]+A[j]+A[k];
    if(total<=target && total>best) best=total;
  }
  return best;
}
```

시간 $O(n^3)$.

# 개념 및 계산

완전 탐색에서 후보 해를 만들어 낼 때 쓰이는 대표적인 도구 세 가지를 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

완전 탐색에서 후보 해를 만들어 낼 때 쓰이는 대표적인 도구 세 가지를 쓰시오.

## 정답 반복문, 순열, 조합

### 해설

변수가 적으면 중첩 반복문으로 충분하고, 순서를 정해야 하면 순열, 고르기만 하면 되면 조합을 쓴다. 부분집합이 필요하면 비트마스크를 함께 쓴다.

# r개 순열 직접 생성

서로 다른 정수 n개가 담긴 A에서 r개를 고르는 모든 순열을 직접 생성하시오. 0≤r≤n≤6 이다. 입력 인덱스 순으로 탐색하며 r=0이면 빈 순열 하나를 반환한다. 순열 생성 라이브러리는 사용하지 않는다.

입력 `make_permutations([1,2,3], 2)`
결과 `[[1,2],[1,3],[2,1],[2,3],[3,1],[3,2]]`

## 작성할 내용

탐색 순서와 종료 조건을 정하고 Python 또는 C로 함수를 작성하시오.
예시 입력의 처리 과정을 보이고, 경계 조건 하나와 시간 복잡도를 설명하시오.

`out[][6] = 선택 결과, return = 행 수 (최대 720)`

# r개 순열 직접 생성

```python
def make_permutations(A, r):
    result, path = [], []
    used = [False] * len(A)
    def dfs():
        if len(path) == r:
            result.append(path.copy())
            return
        for i in range(len(A)):
            if not used[i]:
                used[i] = True
                path.append(A[i])
                dfs()
                path.pop()
                used[i] = False
    dfs()
    return result
```

결과 수 nPr, 기록 시간 O(r·nPr), 재귀·사용 배열 O(n).

# r개 순열 직접 생성

필요한 표준 헤더: <stdlib.h> <string.h> <limits.h> / 결과 배열은 호출자가 준비

```c
static void perm_visit(const int A[], int n, int r, int depth, int used[], int path[],
    int out[][6], int *count) {
  if (depth==r) {
    for(int j=0;j<r;j++) out[*count][j]=path[j];
    (*count)++; return;
  }
  for (int i=0; i<n; i++) {
    if (used[i]) continue;
    used[i]=1; path[depth]=A[i];
    perm_visit(A,n,r,depth+1,used,path,out,count);
    used[i]=0;
  }
}
int make_permutations(const int A[], int n, int r, int out[][6]) {
  int path[6]={0}, count=0;
  int used[6]={0}; perm_visit(A,n,r,0,used,path,out,&count);
  return count;
}
```

결과 수 nPr, 기록 시간 O(r·nPr), 재귀·사용 배열 O(n).

# 개념 및 계산

5명 중에서 금메달, 은메달, 동메달을 받을 사람을 정하는 경우의 수를 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

5명 중에서 금메달, 은메달, 동메달을 받을 사람을 정하는 경우의 수를 쓰시오.

## 정답 60가지

### 해설

5P3 = 5 × 4 × 3 = 60이다. 누가 금인지 은인지가 달라지면 다른 결과이므로 순열로 센다.

# 개념 및 계산

5가지 음식 중 2가지를 골라 함께 먹는 경우의 수를 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

5가지 음식 중 2가지를 골라 함께 먹는 경우의 수를 쓰시오.

## 정답 10가지

### 해설

5C2 = (5 × 4) ÷ 2 = 10이다. 짜장면과 짬뽕을 고르는 것과 짬뽕과 짜장면을 고르는 것은 같은 선택이므로 조합으로 센다.

# r개 조합 직접 생성

서로 다른 정수 배열 A에서 r개를 고르는 모든 조합을 직접 생성하시오. 0≤r≤n≤6이다. 각 조합은 입력 인덱스 증가 순이며 r=0이면 빈 조합 하나다. 조합 생성 라이브러리를 쓰지 않는다.

입력 `make_combinations([1,2,3], 2)`
결과 `[[1,2],[1,3],[2,3]]`

## 작성할 내용

탐색 순서와 종료 조건을 정하고 Python 또는 C로 함수를 작성하시오.
예시 입력의 처리 과정을 보이고, 경계 조건 하나와 시간 복잡도를 설명하시오.

`out[][6] = 선택 결과, return = 행 수 (최대 20)`

# r개 조합 직접 생성

```python
def make_combinations(A, r):
    result, path = [], []
    def dfs(start):
        if len(path) == r:
            result.append(path.copy())
            return
        for i in range(start, len(A)):
            path.append(A[i])
            dfs(i + 1)
            path.pop()
    dfs(0)
    return result
```

결과 수 nCr, 결과 기록 O(r·nCr), 탐색 노드 수를 함께 고려한다.

# r개 조합 직접 생성

필요한 표준 헤더: <stdlib.h> <string.h> <limits.h> / 결과 배열은 호출자가 준비

```c
static void comb_visit(const int A[], int n, int r, int depth, int start, int path[],
    int out[][6], int *count) {
  if (depth==r) {
    for(int j=0;j<r;j++) out[*count][j]=path[j];
    (*count)++; return;
  }
  for (int i=start; i<n; i++) {
    path[depth]=A[i];
    comb_visit(A,n,r,depth+1,i+1,path,out,count);
  }
}
int make_combinations(const int A[], int n, int r, int out[][6]) {
  int path[6]={0}, count=0;
  comb_visit(A,n,r,0,0,path,out,&count);
  return count;
}
```

결과 수 nCr, 결과 기록 O(r·nCr), 탐색 노드 수를 함께 고려한다.

# 개념 및 계산

서로 다른 n개를 하나도 남기지 않고 모두 나열하는 순열의 개수를 n에 대한 식으로 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

서로 다른 n개를 하나도 남기지 않고 모두 나열하는 순열의 개수를 n에 대한 식으로 쓰시오.

## 정답 $n!$

### 해설

첫 자리에 n가지, 다음에 n−1가지 식으로 끝까지 곱한 값이다. 외판원 순회처럼 전체 순서를 정하는 문제에서 이 값이 그대로 탐색 공간이 된다.

# 개념 및 계산

r개를 고르는 재귀 조합 탐색에서 선택 개수를 3개에서 4개로 바꾸면 종료 조건의 목표 깊이는 어떻게 바뀌는가?

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

r개를 고르는 재귀 조합 탐색에서 선택 개수를 3개에서 4개로 바꾸면 종료 조건의 목표 깊이는 어떻게 바뀌는가?

**정답 목표 깊이를 3에서 4로 바꾼다.**

## 해설

선택한 원소의 수가 r에 도달하면 하나의 조합이 완성된다. 이 원리는 Python과 C에서 같다.

# 개념 및 계산

어떤 문제를 순열로 셀지 조합으로 셀지 판단할 때 스스로에게 던져야 할 질문을 한 문장으로 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

어떤 문제를 순열로 셀지 조합으로 셀지 판단할 때 스스로에게 던져야 할 질문을 한 문장으로 쓰시오.

## 정답  뽑은 것들의 자리를 서로 바꾸면 다른 경우가 되는가?

### 해설

달라지면 순열, 같으면 조합이다. 메달 색이 바뀌면 다른 결과지만 세트 메뉴는 순서를 바꿔도 같은 메뉴다.

# 문자열의 모든 일치 위치

텍스트 text에서 패턴 pattern의 모든 시작 인덱스를 무차별 매칭으로 구하시오. 겹치는 일치도 포함한다. 길이는 각각 0~50, 1~10이며 영문자로 구성된다. 일치가 없으면 빈 목록이다.

입력 `bf_string_match("ababcabcababd", "abc")`
결과 `[2,5]`

## 작성할 내용

탐색 순서와 종료 조건을 정하고 Python 또는 C로 함수를 작성하시오.
예시 입력의 처리 과정을 보이고, 경계 조건 하나와 시간 복잡도를 설명하시오.

`out = 시작 인덱스, return = 개수 (용량 201)`

# 문자열의 모든 일치 위치

```python
def bf_string_match(text, pattern):
    result = []
    for i in range(len(text) - len(pattern) + 1):
        j = 0
        while j < len(pattern) and text[i+j] == pattern[j]:
            j += 1
        if j == len(pattern):
            result.append(i)
    return result
```

시간 O((n−m+1)·m), n<m이면 즉시 빈 결과.

# 문자열의 모든 일치 위치

필요한 표준 헤더: <stdlib.h> <string.h> <limits.h> / 결과 배열은 호출자가 준비

```c
int bf_string_match(const char text[], const char pattern[], int out[]) {
  int n=(int)strlen(text), m=(int)strlen(pattern), count=0;
  for(int i=0;i<=n-m;i++) {
    int j=0;
    while(j<m && text[i+j]==pattern[j]) j++;
    if(j==m) out[count++]=i;
  }
  return count;
}
```

시간 O((n−m+1)·m), n<m이면 즉시 빈 결과.

# 가장 가까운 두 수

2≤n≤30인 정수 배열 A의 모든 i<j를 검사하여 차이의 절댓값이 최소인 값의 쌍과 차이를 구하시오. 동률이면 탐색 중 처음 찾은 쌍을 유지한다. 반환 형식은 [A[i],A[j],차이]다. 원소는 −2,000,000,000~2,000,000,000이다.

입력 `closest_pair([10,3,22,15,17,9])`
결과 `[10,9,1]`

## 작성할 내용

탐색 순서와 종료 조건을 정하고 Python 또는 C로 함수를 작성하시오.
예시 입력의 처리 과정을 보이고, 경계 조건 하나와 시간 복잡도를 설명하시오.

`out[0]=첫 값, out[1]=둘째 값, out[2]=최소 차이`

# 가장 가까운 두 수

```python
def closest_pair(A):
    best = None
    for i in range(len(A)):
        for j in range(i + 1, len(A)):
            gap = abs(A[i] - A[j])
            if best is None or gap < best[2]:
                best = [A[i], A[j], gap]
    return best
```

시간 $O(n^2)$.

# 가장 가까운 두 수

필요한 표준 헤더: <stdlib.h> <string.h> <limits.h> / 결과 배열은 호출자가 준비

```c
void closest_pair(const int A[], int n, long long out[3]) {
  out[2]=LLONG_MAX;
  for(int i=0;i<n;i++) for(int j=i+1;j<n;j++) {
    long long diff=llabs((long long)A[i]-A[j]);
    if(diff<out[2]) { out[0]=A[i]; out[1]=A[j]; out[2]=diff; }
  }
}
```

시간 $O(n^2)$.

# 비트마스크 부분집합 목록

서로 다른 정수 배열 A의 모든 부분집합을 비트마스크 증가 순서로 생성하시오. $0 \le n \le 8$ 이다. 각 부분집합 내부는 입력 인덱스 순이며 빈 부분집합도 포함한다.

입력 `power_set([1,2,3])`
결과 `[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]`

## 작성할 내용

탐색 순서와 종료 조건을 정하고 Python 또는 C로 함수를 작성하시오.
예시 입력의 처리 과정을 보이고, 경계 조건 하나와 시간 복잡도를 설명하시오.

`out[][8], sizes[] = 부분집합과 길이, return = 개수 (최대 256)`

# 비트마스크 부분집합 목록

```python
def power_set(A):
    result = []
    for mask in range(1 << len(A)):
        subset = []
        for j in range(len(A)):
            if mask & (1 << j):
                subset.append(A[j])
        result.append(subset)
    return result
```

시간 $O(n \cdot 2^n)$.

# 비트마스크 부분집합 목록

필요한 표준 헤더: <stdlib.h> <string.h> <limits.h> / 결과 배열은 호출자가 준비

```c
int power_set(const int A[], int n, int out[][8], int sizes[]) {
  int count=1<<n;
  for(int mask=0;mask<count;mask++) {
    sizes[mask]=0;
    for(int j=0;j<n;j++) if(mask & (1u<<j))
      out[mask][sizes[mask]++]=A[j];
  }
  return count;
}
```

시간 $O(n \cdot 2^n)$.

# 개념 및 계산

$n \ge m \ge 1$이다. 텍스트의 가능한 시작 위치를 모두 검사하며 각 위치의 첫 글자에서 불일치한다면 총 문자 비교 횟수는 얼마인가?

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

n≥m≥1이다. 텍스트의 가능한 시작 위치를 모두 검사하며 각 위치의 첫 글자에서 불일치한다면 총 문자 비교 횟수는 얼마인가?

## 정답 n−m+1회

### 해설

시작 위치는 0부터 n−m까지이며, 위치마다 한 번 비교한다.

# 개념 및 계산

두 수의 차이를 비교하며 더 작은 차이를 만날 때 best 쌍을 갱신한다. 반복 도중 best가 담고 있는 것은 무엇인가?

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

두 수의 차이를 비교하며 더 작은 차이를 만날 때 best 쌍을 갱신한다. 반복 도중 best가 담고 있는 것은 무엇인가?

## 정답 지금까지 검사한 쌍 중 차이가 가장 작은 쌍

### 해설

아직 검사하지 않은 쌍까지 포함한 최적해는 아니다. 모든 쌍을 확인한 뒤 전체 최적해가 된다.

# 개념 및 계산

원소가 10개인 집합의 부분집합 개수를 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

원소가 10개인 집합의 부분집합 개수를 쓰시오.

## 정답 1,024개

### 해설

$2^{10} = 1{,}024$다. 원소가 20개면 약 100만 개, 30개면 약 10억 개가 되어 곧 감당하기 어려워진다.

# 개념 및 계산

k가 고정된 중첩 반복, 모든 부분집합, 전체 순열의 후보 수 증가를 각각 빅오 표기로 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

k가 고정된 중첩 반복, 모든 부분집합, 전체 순열의 후보 수 증가를 각각 빅오 표기로 쓰시오.

## 정답 $O(n^k)$, $O(2^n)$, $O(n!)$

### 해설

각각 다항식, 지수, 팩토리얼 증가이다. 후보 한 개를 검사하는 비용은 별도로 곱한다.

# 개념 및 계산

n=3일 때 $n!$, $2^n$, $n^n$의 값을 각각 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

n=3일 때 $n!$, $2^n$, $n^n$의 값을 각각 쓰시오.

## 정답 6, 8, 27

### 해설

$3!=3\times2\times1=6$, $2^3=8$, $3^3=27$이다.

# 개념 및 계산

각 자리가 0~9인 8자리 비밀번호의 경우의 수를 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

각 자리가 0~9인 8자리 비밀번호의 경우의 수를 쓰시오.

## 정답 1억 가지 ($10^8$)

### 해설

자리마다 10가지가 곱해진다. 초당 1만 번씩 시도한다면 1억 ÷ 1만 = 1만 초, 약 3시간이 걸린다.

# 개념 및 계산

원소가 30개인 집합의 부분집합 개수를 거듭제곱식으로 쓰시오. 십진수로 계산하지 않아도 된다.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

원소가 30개인 집합의 부분집합 개수를 거듭제곱식으로 쓰시오. 십진수로 계산하지 않아도 된다.

## 정답 $2^{30}$개

### 해설

각 원소를 포함하거나 제외하는 두 선택이 있으므로 2를 30번 곱한 $2^{30}$개이다.

# 출발점을 고정한 외판원 순회

n×n 비용 행렬 dist가 주어진다. 0번 도시에서 출발해 모든 도시를 한 번씩 방문하고 0으로 돌아오는 최소 비용을 구하시오. 2≤n≤8, 대각선 비용은 0이고 나머지는 1~1,000이다. 모든 도시간 이동이 가능하며 비용은 비대칭일 수 있다.

입력 `tsp_min([[0,4,7,3],[4,0,6,2],[7,6,0,5],[3,2,5,0]])`
결과 `18`

`return = 최소 왕복 비용`

## 작성할 내용

탐색 순서와 종료 조건을 정하고 Python 또는 C로 함수를 작성하시오.
예시 입력의 처리 과정을 보이고, 경계 조건 하나와 시간 복잡도를 설명하시오.

# 출발점을 고정한 외판원 순회

```python
def tsp_min(dist):
    n = len(dist)
    used = [False] * n
    used[0] = True
    def dfs(last, depth, cost):
        if depth == n:
            return cost + dist[last][0]
        best = float('inf')
        for city in range(1, n):
            if not used[city]:
                used[city] = True
                value = dfs(city, depth + 1, cost + dist[last][city])
                best = min(best, value)
                used[city] = False
        return best
    return dfs(0, 1, 0)
```

후보 순서 (n−1)!개, 이 구현의 상한 O(n·(n−1)!).

# 출발점을 고정한 외판원 순회

필요한 표준 헤더: <stdlib.h> <string.h> <limits.h> / 결과 배열은 호출자가 준비

```c
static void tsp_visit(const int d[][8], int n, int depth, int last,
                      int used[], int cost, int *best) {
  if(depth==n) {
    int total=cost+d[last][0];
    if(total<*best) *best=total;
    return;
  }
  for(int next=1;next<n;next++) if(!used[next]) {
    used[next]=1;
    tsp_visit(d,n,depth+1,next,used,cost+d[last][next],best);
    used[next]=0;
  }
}
int tsp_min(const int dist[][8], int n) {
  int used[8]={1}, best=INT_MAX;
  tsp_visit(dist,n,1,0,used,0,&best);
  return best;
}
```

후보 순서 (n−1)!개, 이 구현의 상한 O(n·(n−1)!).

# 개념 및 계산

부분집합을 만드는 문제에서 원소가 하나 늘어나면 경우의 수는 몇 배가 되는지 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

부분집합을 만드는 문제에서 원소가 하나 늘어나면 경우의 수는 몇 배가 되는지 쓰시오.

## 정답 2배

### 해설

새 원소를 포함하는 경우와 포함하지 않는 경우로 갈라지므로 정확히 두 배가 된다. 이 두 배가 반복되어 $2^n$이 된다.

# 개념 및 계산

후보를 최대 1,048,576개 검사할 수 있다. 부분집합 후보가 $2^n$개일 때 가능한 n의 최댓값은? 참고: $2^{20}=1{,}048{,}576$이며 n이 1 늘면 후보 수는 2배다.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

후보를 최대 1,048,576개 검사할 수 있다. 부분집합 후보가 $2^n$개일 때 가능한 n의 최댓값은? 참고: $2^{20}=1{,}048{,}576$이며 n이 1 늘면 후보 수는 2배다.

## 정답 20

### 해설

n=20이면 한도와 같고, n=21이면 한도의 두 배가 되므로 최대는 20이다.

# 개념 및 계산

후보를 최대 1,000만 개 검사할 수 있다. 전체 순열 후보가 n!개일 때 가능한 n의 최댓값은? 참고: 10!=3,628,800, 11!=39,916,800이다.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

후보를 최대 1,000만 개 검사할 수 있다. 전체 순열 후보가 n!개일 때 가능한 n의 최댓값은? 참고: 10!=3,628,800, 11!=39,916,800이다.

## 정답 10

### 해설

10!은 한도 이하이고 11!은 한도를 넘는다. 팩토리얼 값은 문제에 제공된 수를 비교하면 된다.

# 개념 및 계산

컴퓨터가 2배 빨라지면 같은 시간 안에 검사할 수 있는 부분집합 문제의 원소 수 n은 얼마나 늘어나는가?

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

컴퓨터가 2배 빨라지면 같은 시간 안에 검사할 수 있는 부분집합 문제의 원소 수 n은 얼마나 늘어나는가?

## 정답 1개

### 해설

$2^{n+1}=2\times2^{n}$이므로 같은 후보당 비용에서 원소를 하나 더 처리한다. 하드웨어 속도 향상만으로 지수 증가를 없앨 수는 없다.

# 개념 및 계산

N-Queens와 스도쿠는 완전 탐색에서 발전한 어떤 기법의 대표 응용인지 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

N-Queens와 스도쿠는 완전 탐색에서 발전한 어떤 기법의 대표 응용인지 쓰시오.

## 정답 백트래킹

### 해설

두 문제 모두 규칙을 어기는 순간 그 갈래를 버리고 되돌아간다. 끝까지 만든 뒤 검사하는 완전 탐색보다 훨씬 적은 경우만 확인한다.

# 개념 및 계산

병합 정렬처럼 문제를 작은 문제로 나누어 각각 해결하고 결과를 합치는 기법의 이름을 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

병합 정렬처럼 문제를 작은 문제로 나누어 각각 해결하고 결과를 합치는 기법의 이름을 쓰시|오.

## 정답 분할 정복

### 해설

문제를 분할하고 작은 문제를 해결한 뒤 결합한다.

# 개념 및 계산

$F(n)=F(n-1)+F(n-2)$를 그대로 재귀 계산할 때와 메모이제이션할 때의 시간 복잡도 상한을 각각 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

$F(n)=F(n-1)+F(n-2)$를 그대로 재귀 계산할 때와 메모이제이션할 때의 시간 복잡도 상한을 각각 쓰시오.

## 정답 단순 재귀 $O(2^n)$, 메모이제이션 $O(n)$

### 해설

덧셈을 상수 시간으로 가정한다. 메모이제이션은 각 $F(k)$를 한 번만 계산한다.

# 개념 및 계산

동적 계획법이 중복 부분 문제를 처리하는 핵심 방법을 한 문장으로 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

동적 계획법이 중복 부분 문제를 처리하는 핵심 방법을 한 문장으로 쓰시오.

**정답** 부분 문제의 답을 저장하여 같은 계산에 재사용한다.

## 해설

상향식 테이블 채우기 또는 하향식 메모이제이션으로 구현할 수 있다.

# 개념 및 계산

각각 정확히 n번 도는 삼중 반복문의 가장 안쪽 연산은 n=100일 때 몇 번 실행되는가?

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

각각 정확히 n번 도는 삼중 반복문의 가장 안쪽 연산은 n=100일 때 몇 번 실행되는가?

**정답 1,000,000번**

## 해설

100×100×100=1,000,000이다.

98 단답형

# 개념 및 계산

완전 탐색에서 쓰는 후보 생성 방식 다섯 가지를 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

완전 탐색에서 쓰는 후보 생성 방식 다섯 가지를 쓰시오.

**정답** 단일 반복, 중첩 반복, 조합 생성, 순열 생성, 비트마스크

## 해설

Python과 C 모두 같은 탐색 원리를 구현할 수 있다.

# 0 1 배낭 구현

물건별 weights, values와 용량 capacity가 주어진다. 각 물건을 최대 한 번 골라 총 무게가 용량 이하인 최대 가치를 구하시오. 0≤n≤16, 무게·가치는 1~100, 0≤capacity≤300이다. 비트마스크로 모든 선택을 검사한다.

입력 `knapsack([4,6,3,5], [5,8,3,6], 10)`
결과 `13`

`return = 최대 가치`

## 작성할 내용

탐색 순서와 종료 조건을 정하고 Python 또는 C로 함수를 작성하시오.
예시 입력의 처리 과정을 보이고, 경계 조건 하나와 시간 복잡도를 설명하시오.

# 0 1 배낭 구현

```python
def knapsack(weights, values, capacity):
    best = 0
    for mask in range(1 << len(weights)):
        weight, value = 0, 0
        for j in range(len(weights)):
            if mask & (1 << j):
                weight += weights[j]
                value += values[j]
        if weight <= capacity and value > best:
            best = value
    return best
```

시간 $O(n \cdot 2^n)$.

# 0 1 배낭 구현

필요한 표준 헤더: <stdlib.h> <string.h> <limits.h> / 결과 배열은 호출자가 준비

```c
int knapsack(const int weights[], const int values[], int n, int capacity) {
  int best=0;
  for(unsigned mask=0;mask<(1u<<n);mask++) {
    int weight=0, value=0;
    for(int j=0;j<n;j++) if(mask & (1u<<j)) {weight+=weights[j]; value+=values[j];}
    if(weight<=capacity && value>best) best=value;
  }
  return best;
}
```

시간 $O(n \cdot 2^n)$.

100 단답형

# 개념 및 계산

회문(palindrome)이 무엇인지 한 문장으로 쓰시오.

답을 값·식 또는 한두 문장으로 쓰시오.

# 개념 및 계산

회문(palindrome)이 무엇인지 한 문장으로 쓰시오.

**정답** 앞에서 읽으나 뒤에서 읽으나 같은 문자열

### 해설

level과 noon이 예이다. 양 끝의 문자를 차례로 비교하여 확인할 수 있다.