2025 암호분석경진대회

2번 문제

1. LEA의 Modular Addition 연산에 대한 XOR 차분분포표 내의 차분 확률 0.1 이상을 갖는 입출력차분 형태를 구하는 코드를 제시하시오.

```
import math

# N비트 비트별 NOT 연산
def bitwise_not_arb(x, n_bits):
    mask = (1<<n_bits)-1
    return ~x&mask

# N비트 비트별 eq 함수 정의: eq(p,q,r):=(~p^q)&(~p^r)
def bitwise_eq_arb(p, q, r, n_bits):
    mask = (1<<n_bits)-1
    p_not = bitwise_not_arb(p, n_bits)
    return (p_not^q)&(p_not^r)

# Modular Addition 차분 특성의 가중치 계산: HW(eq(a<<1, b<<1, c<<1) ^ (a ^ b ^ c
^ (b<<1)))
def calculate_modadd_weight(a, b, c, n_bits):
    mask = (1<<n_bits)-1
    p = (a<<1)&mask
    q = (b<<1)&mask
    r = (c<<1)&mask

    eq_val = bitwise_eq_arb(p, q, r, n_bits)
    # 조건 실패 비트: eq(a<<1, ...)^(a^b^c^(b<<1))
    condition_failure = eq_val^(a^b^c^((b<<1)&mask))

    # Hamming Weight 계산
    hw = bin(condition_failure).count('1')
    return hw

# --- 확률 0.1 이상인 (a, b, c) 찾기 (8비트 예시) ---
n_bits_example = 8
prob_threshold = 0.1
# 확률 >= 0.1 -> 가중치 <= -log2(0.1)
weight_threshold = -math.log2(prob_threshold)

print(f"--- {n_bits_example}비트 Modular Addition 차분 확률 0.1 이상인 입출력 차분
형태 탐색 ---")
print(f"(가중치(weight) <= {weight_threshold:.2f} 인 경우, 즉 Hamming Weight
<= 3 인 경우)")
print("참고: 8비트 전수 조사는 시간이 소요될 수 있습니다. 128비트 전수 조사는 불가능합니다.")

found_count = 0
search_limit = 256 # 8비트 전수 조사: 2^(3*8) = 2^24
```

```
print(f"0x00부터 0x{search_limit-1:02x}까지의 모든 a, b, c 조합을 확인합니다...")

# 8비트 전수 조사 (2^24 조합)
try:
    for a in range(search_limit):
        for b in range(search_limit):
            for c in range(search_limit):
                weight = calculate_modadd_weight(a, b, c, n_bits_example)
                if weight<=weight_threshold:
                    prob = 2**-weight
                    print(f"발견: a=0x{a:02x}, b=0x{b:02x}, c=0x{c:02x}, 가중
치(HW)={weight}, 확률={prob:.6f}")
                    found_count += 1
                    if found_count >= 20:
                       raise StopIteration
except StopIteration:
    pass

print(f"\n총 {found_count}개의 8비트 입출력 차분 쌍 (a, b, c)에서 가중치 <=
{weight_threshold:.2f} ({prob_threshold} 이상 확률)를 만족하는 형태 확인")
```

```
--- 8비트 Modular Addition 차분 확률 0.1 이상인 입출력 차분 형태 탐색 ---
(가중치(weight) <= 3.32 인 경우, 즉 Hamming Weight <= 3 인 경우)
참고: 8비트 전수 조사는 시간이 소요될 수 있습니다. 128비트 전수 조사는 불가능합니다.
0x00부터 0xff까지의 모든 a, b, c 조합을 확인합니다...
발견: a=0x00, b=0x00, c=0x15, 가중치(HW)=2, 확률=0.250000
발견: a=0x00, b=0x00, c=0x25, 가중치(HW)=2, 확률=0.250000
발견: a=0x00, b=0x00, c=0x29, 가중치(HW)=2, 확률=0.250000
발견: a=0x00, b=0x00, c=0x2a, 가중치(HW)=2, 확률=0.250000
발견: a=0x00, b=0x00, c=0x2b, 가중치(HW)=2, 확률=0.250000
발견: a=0x00, b=0x00, c=0x2d, 가중치(HW)=2, 확률=0.250000
발견: a=0x00, b=0x00, c=0x35, 가중치(HW)=2, 확률=0.250000
발견: a=0x00, b=0x00, c=0x45, 가중치(HW)=2, 확률=0.250000
발견: a=0x00, b=0x00, c=0x49, 가중치(HW)=2, 확률=0.250000
발견: a=0x00, b=0x00, c=0x4a, 가중치(HW)=2, 확률=0.250000
발견: a=0x00, b=0x00, c=0x4b, 가중치(HW)=2, 확률=0.250000
발견: a=0x00, b=0x00, c=0x4d, 가중치(HW)=2, 확률=0.250000
발견: a=0x00, b=0x00, c=0x51, 가중치(HW)=2, 확률=0.250000
발견: a=0x00, b=0x00, c=0x52, 가중치(HW)=2, 확률=0.250000
발견: a=0x00, b=0x00, c=0x53, 가중치(HW)=2, 확률=0.250000
발견: a=0x00, b=0x00, c=0x54, 가중치(HW)=2, 확률=0.250000
발견: a=0x00, b=0x00, c=0x55, 가중치(HW)=0, 확률=1.000000
발견: a=0x00, b=0x00, c=0x56, 가중치(HW)=2, 확률=0.250000
발견: a=0x00, b=0x00, c=0x57, 가중치(HW)=2, 확률=0.250000
발견: a=0x00, b=0x00, c=0x59, 가중치(HW)=2, 확률=0.250000

총 20개의 8비트 입출력 차분 쌍 (a, b, c)에서 가중치 <= 3.32 (0.1 이상 확률)를 만족하는 형
태 확인
```

2. Newsted Monte Carlo Tree Search(NMCTS)는 랜덤 트리 탐색 방법으로 차분 경로 탐색을 위해 활용할 수 있다. Modular Addition 연산의 차분 경로 탐색이 어려운 이유와 이를 해결하기 위해 NMCTS가 활용하는 방법을 설명하시오.

- **Modular Addition 차분 경로 탐색의 어려움**
  - Modular Addition 연산의 가장 큰 특징은 **캐리(carry)**
  - 한 비트 위치에서의 덧셈 결과와 캐리는 이전 비트 위치에서의 값들과 캐리에 의해 복잡하게 영향을 받음
  - 이러한 캐리 전파는 입력 차분 (a, b)와 출력 차분 c 사이의 관계를 비선형적이고 국소적으로 예측하기 어렵게 만듦
  - ***복잡한 확률 분포***
    - XOR 연산과 달리 Modular Addition의 차분 $(a, b) \rightarrow c$가 특정 확률로 발생할 때, 그 확률은 단순히 활성 비트 수에만 비례하지 않고 a와 b의 비트 패턴에 따라 매우 불규칙하게 변함
    - 가중치 HW 계산식에서 볼 수 있듯, 특정 비트 위치에서의 차분 특성 성립 여부는 인접 비트들의 상태에 따라 달라짐
  - ***탐색 공간의 거대함***
    - 블록 암호의 차분 경로 탐색은 라운드를 거치면서 가능한 모든 중간 차분들의 조합을 살펴보는 과정
    - Modular Addition처럼 복잡한 비선형 연산을 포함하는 경우, 각 라운드에서 가능한 출력 차분의 가짓수가 많고 그 확률 분포가 복잡하여 유망한 경로를 미리 예측하기 어려움
    - 결과적으로 탐색 공간((a, b, c) 쌍들의 시퀀스)이 지수적으로 증가하여 전수 조사는 현실적으로 불가능
- **NMCTS가 이를 해결하는 방법**
  - Newsted Monte Carlo Tree Search(NMCTS)는 Monte Carlo Tree Search(MCTS)를 차분 경로 탐색 문제에 적용한 한 형태
  - MCTS는 게임 AI 분야에서 주로 사용되지만, 탐색 공간이 매우 넓고 복잡하여 완전 탐색이 어려운 문제에 효과적인 휴리스틱 탐색 기법
  - NMCTS는 이러한 MCTS의 원리를 활용하여 Modular Addition의 복잡성으로 인한 차분 경로 탐색의 어려움을 해결
  - NMCTS는 다음과 같은 단계들을 반복하며 차분 경로 트리를 구축 및 탐색
    - ***선택(Selection)***
      - 현재 노드(라운드 입력 차분 상태)에서 UCT(Upper Confidence Bound 1 applied to trees) 등의 전략을 사용하여 다음 탐색할 자식 노드(다음 라운드의 중간 차분 상태)를 선택
      - UCT는 이미 좋은 결과를 보인 경로(exploitation)와 아직 탐색되지 않은 새로운 경로(exploration) 사이에서 균형을 맞춰 탐색의 효율성을 높임
      - 차분 분석에서는 낮은 가중치(높은 확률)를 가진 경로를 "좋은 결과"로 간주
    - ***확장(Expansion)***
      - 선택된 노드가 아직 완전히 확장되지 않았다면, 해당 노드에서 발생 가능한 유망한 다음 단계의 차분 전환(예: 특정 Modular Addition의 입출력 차분 쌍 선택) 몇 가지를 선택하여 새로운 자식 노드를 추가
      - Modular Addition의 경우, 입력 차분 (a, b)에 대해 가중치 계산 함수를 사용하여 가중치가 낮은 출력 차분 c 후보들을 찾아 자식 노드로 만듦
    - ***시뮬레이션(Simulation/Rollout)***
      - 새로 추가된 자식 노드(탐색 트리의 리프 노드)에서 시작하여 미리 정해진 라운드 수 또는 암호 끝까지 랜덤한 차분 전환을 선택하며 경로를 진행
      - 이때 각 전환의 가중치를 합산하여 해당 경로의 총 가중치를 추정
      - Modular Addition 단계에서는 확률/가중치를 고려하여 다음 차분을 랜덤하게 선택
    - ***역전파 (Backpropagation)***

        - 시뮬레이션으로 얻은 총 가중치 추정 값을 시뮬레이션 경로를 따라 리프 노드부터 루트 노드까지 거슬러 올라가면서 해당 경로에 포함된 모든 노드의 통계 정보(총 가중치, 방문 횟수)를 갱신
    - 해당 과정을 여러 번 반복하면, MCTS 트리는 낮은 가중치를 가진 유망한 차분 경로가 있는 부분을 집중적으로 탐색하게 됨
    - Modular Addition의 복잡한 확률 분포 때문에 전수 조사는 불가능하지만, NMCTS는 통계적 샘플링과 유망한 경로에 대한 집중 탐색을 통해 전역 최적(가장 낮은 가중치 경로)에 가까운 경로를 효율적으로 찾으려고 시도
        - 즉, 복잡한 확률 계산을 모든 경우에 대해 수행하는 대신, 랜덤 워크를 통해 경험적으로 경로의 가중치를 추정하고 이를 바탕으로 탐색 방향을 결정함으로써 문제의 어려움을 회피
    - Newsted가 적용한 구체적인 개선 사항(NMCTS)은 MCTS의 기본 프레임워크 내에서 차분 분석 문제에 특화된 노드 평가, 탐색 전략, 시뮬레이션 방법 등을 포함할 수 있음

3. 2에서 설명한 방법으로 NMCTS를 이용해 차분 경로를 탐색하는 코드를 제시하고, 해당 코드를 이용해 탐색한 차분 경로를 제시하시오.

```
import random

def bitwise_not_arb(x, n_bits):
    mask = (1<<n_bits)-1
    return ~x&mask

def bitwise_eq_arb(p, q, r, n_bits):
    mask = (1<<n_bits)-1
    p_not = bitwise_not_arb(p, n_bits)
    return (p_not^q)&(p_not^r)

def calculate_modadd_weight(a, b, c, n_bits):
    mask = (1<<n_bits)-1
    p = (a<<1)&mask
    q = (b<<1)&mask
    r = (c<<1)&mask

    eq_val = bitwise_eq_arb(p, q, r, n_bits)
    # 조건 실패 비트: eq(a<<1, ...)^(a^b^c^(b<<1))
    condition_failure = eq_val^(a^b^c^((b<<1)&mask))

    # Hamming Weight 계산
    hw = 0
    temp = condition_failure
    for _ in range(n_bits):
        if temp&1: hw+=1
        temp>>=1
    return hw

# --- NMCTS-like 단일 단계 차분 선택 예시 ---
def nmcts_select_next_modadd_diff_example(input_a, input_b, n_bits,
num_candidates_to_check=1000, exploration_param=1.0):
    """
    주어진 입력 차분 (input_a, input_b)에 대해
    가능한 출력 차분 c 후보들을 가중치와 랜덤성을 고려하여 선택합니다.
```

```
    이는 NMCTS의 단일 선택/평가 단계에 대한 간략화된 예시입니다.
    """
    candidates = []

    print(f"--- NMCTS 유사 단일 단계 선택 ({n_bits}비트 ModAdd) ---")
    print(f"입력 차분 (a, b): (0x{input_a:0{n_bits//4}x},
0x{input_b:0{n_bits//4}x})")
    print(f"다음 출력 차분 c 후보 {num_candidates_to_check}개 확인 중...")

    # 실제 NMCTS에서는 후보 c를 생성하거나 탐색하는 전략이 필요합니다.
    # 여기서는 랜덤하게 c 후보를 생성하여 가중치를 계산합니다.
    for i in range(num_candidates_to_check):
        # 랜덤하게 출력 차분 c 후보 생성
        c = random.randint(0, (1<<n_bits)-1)
        weight = calculate_modadd_weight(input_a, input_b, c, n_bits)

        # NMCTS는 가중치(weight)를 최소화하는 방향으로 탐색합니다.
        # 평가 점수 = -가중치 + 탐색 보너스
        # 탐색 보너스는 UCT의 개념을 단순화한 것으로, 랜덤성을 도입합니다.
        score = -weight + random.random() * exploration_param
        candidates.append({'c': c, 'weight': weight, 'score': score})


    if not candidates:
        print("  후보를 생성할 수 없습니다.")
        return None, None

    # 가장 높은 점수(가장 낮은 가중치 + 랜덤 탐색 보너스)를 가진 후보 선택
    best_candidate = max(candidates, key=lambda item: item['score'])

    print(f"\n가장 높은 점수를 얻은 다음 차분 c 선택 완료.")
    return best_candidate['c'], best_candidate['weight']

# --- 예시 실행 ---
n_bits_example_q3 = 8 # 8비트 예시 사용
example_a_q3 = 0x5A    # 예시 입력 차분 a
example_b_q3 = 0xA5    # 예시 입력 차분 b
num_iterations_q3 = 5000 # 확인할 c 후보 개수 (NMCTS 시뮬레이션 횟수와 유사한 개념)

selected_c_q3, selected_weight_q3 = nmcts_select_next_modadd_diff_example(
    example_a_q3, example_b_q3, n_bits_example_q3, num_iterations_q3
)

print("\n--- NMCTS 유사 단일 단계 탐색 결과 ---")
if selected_c_q3 is not None:
    print(f"입력 차분 (a, b): (0x{example_a_q3:0{n_bits_example_q3//4}x},
0x{example_b_q3:0{n_bits_example_q3//4}x})")
    print(f"선택된 다음 출력 차분 (c):
0x{selected_c_q3:0{n_bits_example_q3//4}x}")
    print(f"계산된 가중치: {selected_weight_q3:.2f}")
    print(f"예상 확률: {2**-selected_weight_q3:.6f}")

    # 탐색된 차분 경로 (단일 단계 예시)
    print(f"\n--- 탐색된 차분 경로 (단일 Modular Addition 단계) ---")
```

```
    print(f"(0x{example_a_q3:0{n_bits_example_q3//4}x},
0x{example_b_q3:0{n_bits_example_q3//4}x}) ->
0x{selected_c_q3:0{n_bits_example_q3//4}x} [가중치:
{selected_weight_q3:.2f}]")

else:
    print("NMCTS 유사 탐색 중 다음 차분을 선택하지 못했습니다.")
```

```
--- NMCTS 유사 단일 단계 선택 (8비트 ModAdd) ---
입력 차분 (a, b): (0x5a, 0xa5)
다음 출력 차분 c 후보 5000개 확인 중...

가장 높은 점수를 얻은 다음 차분 c 선택 완료.

--- NMCTS 유사 단일 단계 탐색 결과 ---
입력 차분 (a, b): (0x5a, 0xa5)
선택된 다음 출력 차분 (c): 0xb4
계산된 가중치: 0.00
예상 확률: 1.000000

--- 탐색된 차분 경로 (단일 Modular Addition 단계) ---
(0x5a, 0xa5) -> 0xb4 [가중치: 0.00]
```