**1. 비트연산을 추가한 fibonacci를 구하는 알고리즘**

기존에 행렬 곱셈을 이용하여 원하는 fibonacci를 구하는 방법은 아래와 같고,

시간복잡도는 O(n)이다.

폰트, 번호, 도표, 타이포그래피이(가) 표시된 사진

자동 생성된 설명

이 보고서는 기존 방법보다 Fibonacci 수열을 효율적으로 계산하기 위한 알고리즘을 설계하고,

이를 행렬 곱셈, 반복 제곱법, 비트 연산을 활용하여 시간 복잡도를

O(log2n)으로 줄이는 방법을 제안한다.

**반복 제곱법(Exponentiation by Squaring)**

반복 제곱법은 수의 거듭제곱을 O(log2n)시간 안에 계산할 수 있는 알고리즘이다.

행렬 An의 계산에 이 방식을 적용하면 다음과 같은 과정을 따른다:

1. n을 2진수로 표현하여 각 비트를 검사한다.
2. n의 비트가 1인 경우, 현재 행렬 값을 결과에 곱한다.
3. A2, A4, A8,…A2, A4, A8,…과 같이 행렬을 제곱하며 n을 2로 나눈다.

이 방식은 n의 비트 수(즉, log2​n)에 비례하여 연산을 수행하므로 시간 복잡도는 O(log2n)이다.

**알고리즘 설계**

Fibonacci 수열 계산을 위해 다음 과정을 따른다:

1. **행렬 곱셈 함수**: 두 행렬 A와 B를 곱하는 함수이다.
2. **행렬 거듭제곱 함수**: 반복 제곱법을 활용하여 행렬 An을 계산한다.

비트 연산을 통해 효율적으로 계산한다.

1. **Fibonacci 계산 함수**: n = 0인 경우 0을 반환한다.

n ≥ 1인 경우 행렬 An−1의 첫 번째 행 첫 번째 열 값을 반환한다.

**수도 코드**

**function matrixMultiply(A, B):**

*# 행렬 A와 B를 곱하는 함수*

return [[A[0][0] \* B[0][0] + A[0][1] \* B[1][0], A[0][0] \* B[0][1] + A[0][1] \* B[1][1]],

[A[1][0] \* B[0][0] + A[1][1] \* B[1][0], A[1][0] \* B[0][1] + A[1][1] \* B[1][1]]]

**function matrixPower(A, n):**

*# 행렬 A의 n승을 계산*

result = [[1, 0], [0, 1]] *# 단위 행렬*

base = A

while n > 0:

if n & 1: # n의 마지막 비트가 1이면

result = **matrixMultiply**(result, base)

base = **matrixMultiply**(base, base) *# base = base^2*

n = n >> 1 *# n을 오른쪽으로 1비트 이동 (n /= 2)*

return result

**function fibonacci(n):**

if n == 0:

return 0

A = [[1, 1], [1, 0]] *# Fibonacci 기본 행렬*

powerMatrix = **matrixPower**(A, n - 1)

return powerMatrix[0][0]