컴퓨터공학과 20211507 김동건

행렬 곱셈을 사용하여 실습 1-3을 해결했다. 이를 활용하여 비트연산을 추가해 fibonacci를 구하는 알고리즘을 수도 코드로 작성하고 설명하시오. 시간 복잡도는 O(log 𝑛)으로 줄일 것.

실습 1-3에서 아래 행렬 곱셈을 이용하여 n번째 피보나치 수를 구할 수 있었다.

도표, 폰트, 화이트, 라인이(가) 표시된 사진

자동 생성된 설명

2 X 2 행렬을 n번 제곱하면, O(n)의 시간복잡도를 갖는다. 2 X 2 행렬과 2 X 2 행렬을 곱하는 데에는 상수시간이 소요되기 때문이다. 이제, 비트연산을 이용하여 행렬 곱셈을 O(n)번이 아닌 O(logn)번 하도록 하자. 행렬 곱셈도 마찬가지로, n 제곱을 구하기 위해 logn에 구할 수 있다. 예를 들어, 100번 째 피보나치 수를 구하자고 해보자. 그렇다면 주어진 행렬을 100 제곱 해야 한다. 이때, 100을 2진수로 나타내면 1100100이다. 이때, 1인 값은 64, 32, 4이다. 즉, 우리는 행렬의 64, 32, 4제곱을 계산한 다음, 그 세 행렬을 곱하여 100제곱을 만들 수 있다. 행렬의 1, 2, 4, 8, 16, ... , msb가 나타내는 수만큼의 행렬 제곱을 모두 구하는 데에는 O(logn)이 든다. 그 후, 비트가 1인 것들을 모아 곱하는 데에도 O(logn)이 들기 때문에 최종 시간복잡도 O(logn)에 행렬의 n제곱을 구할 수 있다.

이제, 이를 수도 코드로 작성하여 보자. 코드는 크게 세 부분으로 나뉜다. 1) 피보나치 수를 구해서 리턴하는 함수, 2) 행렬 끼리의 곱을 계산하는 함수, 3) 비트 연산을 이용한 거듭제곱으로 행렬의 n제곱을 구하는 함수  
  
function Fibo(n):  
 if n == 0 : return 1 # base case  
 elif n == 1 : return 1  
 matrix = [[1, 1], [1, 0]] # 주어진 행렬  
 result = matrix\_pow(matrix, n - 1)   
 return result[0][0] # n - 1번 곱한 뒤, (0, 0)에는 F\_n이 들어있다.

function matrix\_pow(matrix, exp) :  
 ret = [[1, 0], [0, 1]] # 항등행렬로 정의  
 while exp > 0 : # 비트 연산을 이용하여 거듭제곱 하는 부분  
 if exp & 1 : # 현재 exp 값이 홀수인 경우 한 번 더 곱해주어야 한다.  
 ret = multi(ret, matrix)  
 matrix = multi(matrix, matrix) # 지금까지 구한 제곱수를 제곱하여 지수를 2배한다.   
 exp >> = 1  
 return ret

function multi(a, b):  
 # m \* n 행렬 a와 n \* p 행렬 b를 곱하는 함수  
 # 이 문제에선 m, n, p가 모두 2이므로 상수 시간에 구할 수 있다.  
 m = a's row  
 n = a's column ( = b's row)  
 p = b's column  
 ret = n \* p matrix filled with 0  
 for i from 0 to m - 1  
 for j from 0 to p -1  
 for k from 0 to n - 1  
 ret[i][j] += a[i][k] \* a[k][j]  
 return ret

위와 같이 세 개의 함수를 이용하여 n번째 피보나치 수를 O(logn)에 구할 수 있다. matrix\_pow함수는 위의 비재귀 방식 말고도 재귀적으로도 구할 수 있다. 재귀적으로 구할 때에는 현재 b제곱을 구하고 있다면, floor(b/2) 제곱을 재귀적으로 구해 온 다음, 그 값을 제곱한다. 만약 b가 홀수라면 [[1, 1], [1, 0]]을 한 번 더 곱해준다. 다만, 시간은 비재귀에 비해 조금 느린편이고, 메모리 사용량이 많아 위의 코드는 비재귀 기준으로 작성하였다.