[알고리즘05반]_01_201701988_김수빈

Divide & Conquer

제출일 19.09.26 (목)

실행 환경 : java – eclipse 사용

```
C:₩Users₩김수빈>java -version
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)

C:₩Users₩김수빈>javac -version
javac 11.0.4

C:₩Users₩김수빈>
```

사용한 라이브러리

사용자 입력을 받기 위한 Scanner와 소수점 형식을 맞추기 위해 DecimalFormat, 매우 큰 정수를 처리하기 위한 BigInteger 라이브러리를 사용한다.

구현 내용

과제 조건 대로 첫 번째 사용자 입력에 따라 Fibonacci를 1이면 Recursion, 2면 Array, 3이면 Recursive squaring 구현 방법으로 연산한다. 두 번째 사용자 입력을 n으로 하여 for문을 통해 Fibonacci 0부터 n까지의 값을 구하고 각 Fibonacci 값을 구할 때 걸리는 시간과 함께 출력한다.

또 매우 큰 정수를 처리하기 위해 자료형 BigInteger을 사용했다.

Recursion 구현 방법인 경우,BigInteger n을 매개변수로 받아 n이 0 또는 1이면 n을 리턴 하고, 2 이상일 경우 재귀를 통해 n-1과 n-2를 인수로 넣은 recursion 호출 결과를 더해 리턴 한다.

Array 구현 방법의 경우, 매우 큰 정수를 처리할 수 있게 heap stack overflow를 막기 위해 길이가 3인 배열을 사용하였다. 인덱스 0과 1에 각각 0, 1을 채우고 인덱스 2부터 n까지는, 이전 인덱스 2개를 3으로 나눈 나머지 인덱스 값들을 더해 3으로 나눈 나머지 값 인덱스에 삽입하였다.

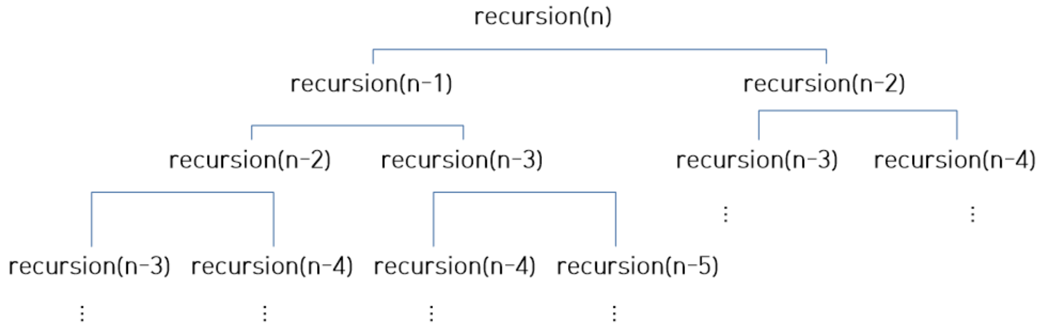
Recursive squaring 구현 방법의 경우 pseudo code에 따라 3가지 메소드를 사용하였다. Recursive_squaring은 BigInteger n을 매개변수로 받아 n이 2보다 작을 때는 n을 리턴 하고, 아니면 다음을 수행한다. matrix A를 정의하고 A와 n을 인수로 하는 pow 호출 결과의 0행, 1열의 값을 리턴 한다.

Pow는 2차원 행렬 A와 BigInteger n을 매개변수로 받아 A를 n만큼 제곱하는 메소드다. n이 1이면 A를 리턴 하고, 아니면 다음을 수행한다. N이 짝수일 경우, n을 2로 나눠 A를 n/2 제곱한 것과 A를 n/2 제곱한 것에 대해 mul을 호출하여 호출 결과를 리턴 한다. N이 홀수일 경우 A를 (n-1)/2 제곱한 것과 A를 (n+1)/2 제곱한 것에 대한 mul을 호출하고 그 결과를 리턴 한다.

Mul은 2차원 행렬 2개를 인수로 받아 곱셈 연산을 수행하는 메소드다.

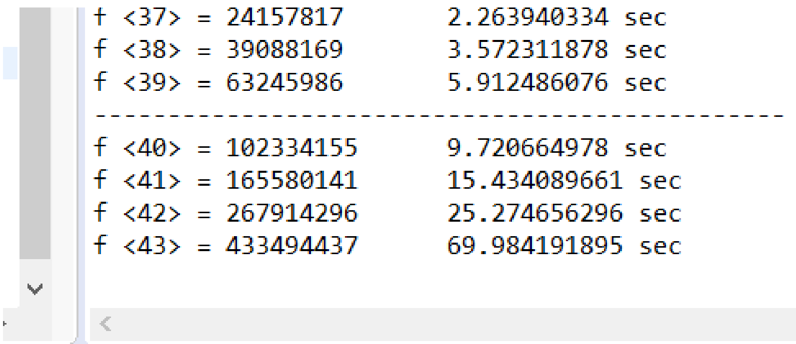
3가지 방법의 성능 비교는 다음과 같다.

Recursion을 사용하여 구현한 방법은 재귀를 통해 n-1과 n-2를 인수로 하는 recursion을 호출한다. 이 호출 방식을 트리로 나타내면 다음과 같다



이 경우 시간 복잡도는 트리의 노드 개수이다. 트리의 높이는 가장 왼쪽의 깊이에 따라 약 n이며, 높이가 n일 때 노드의 최대 개수는 $2^{n+1}-1$ 이다. 따라서 $O(2^n)$의 시간 복잡도를 갖게 된다 .exponential time을 가지므로,n이 커짐에 따라 성능이 급격히 떨어진다.

실제로 n이 43 만 돼도 다른 두 방법에 비해 급격히 느려짐을 확인하였다.



```
f <37> = 24157817       2.263940334 sec
f <38> = 39088169       3.572311878 sec
f <39> = 63245986       5.912486076 sec
-----------------------------------------------
f <40> = 102334155      9.720664978 sec
f <41> = 165580141      15.434089661 sec
f <42> = 267914296      25.274656296 sec
f <43> = 433494437      69.984191895 sec
```

배열을 사용한 방법은 for 문을 통해 입력된 n 까지 이전 두 개 인덱스의 값들을 더해 피보나치 연산을 수행한다. 따라서 $\Theta(n)$의 시간이 소요된다. recursion 보다는 훨씬 빠른 속도로 피보나치 50 까지 연산했다.

```
f <40> = 102334155      0.000133600 sec
f <41> = 165580141      0.000162600 sec
f <42> = 267914296      0.000293200 sec
f <43> = 433494437      0.000210800 sec
f <44> = 701408733      0.000209100 sec
f <45> = 1134903170     0.000219400 sec
f <46> = 1836311903     0.000223700 sec
f <47> = 2971215073     0.000213900 sec
f <48> = 4807526976     0.000132600 sec
f <49> = 7778742049     0.000127900 sec
-----------------------------------------------
f <50> = 12586269025    0.000126700 sec
```

Recursive squaring 은 $\begin{matrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{matrix} = \begin{matrix} 1 & 1 \\ 1 & 0 \end{matrix}^n$ 수식을 통해 피보나치 값을 연산하는 방법이다. Divide and Conquer 방식을 통해 행렬 제곱을 하여 구해진 행렬의 0 행 1 열 또는 1 행 0 열의 값을 반환한다. Divide and Conquer 방식을 사용하여 $\Theta(\log n)$ 의 시간 복잡도를 가진다.

배열 방법과 마찬가지로 recursion 보다 훨씬 빠른 속도로 피보나치 50 을 연산했다.

```
-----------------------------------------------
f <40> = 102334155      0.000100000 sec
f <41> = 165580141      0.000441200 sec
f <42> = 267914296      0.000159200 sec
f <43> = 433494437      0.000207600 sec
f <44> = 701408733      0.000106200 sec
f <45> = 1134903170     0.000258600 sec
f <46> = 1836311903     0.000206300 sec
f <47> = 2971215073     0.000174800 sec
f <48> = 4807526976     0.000153300 sec
f <49> = 7778742049     0.000124000 sec
-----------------------------------------------
f <50> = 12586269025    0.000108900 sec
```

문제 사이즈가 충분히 커졌을 때 recursive squaring 방식이 array 방식보다 빠름을 알 수 있다.

<>

<Array 구현 방식>

```
f <86> = 420196140727489673     0.000295000 sec
f <87> = 679891637638612258     0.000226300 sec
f <88> = 1100087778366101931    0.000379000 sec
f <89> = 1779979416004714189    0.000279600 sec
-----------------------------------------------
f <90> = 2880067194370816120    0.000243700 sec
```

<Recursive squaring 구현 방식>

```
f <81> = 37889062373143906      0.000178500 sec
f <82> = 61305790721611591      0.000154900 sec
f <83> = 99194853094755497      0.000196300 sec
f <84> = 160500643816367088     0.000136700 sec
f <85> = 259695496911122585     0.000165100 sec
f <86> = 420196140727489673     0.000146100 sec
f <87> = 679891637638612258     0.000159000 sec
f <88> = 1100087778366101931    0.000162700 sec
f <89> = 1779979416004714189    0.000184900 sec
-----------------------------------------------
f <90> = 2880067194370816120    0.000143000 sec
```

Strassen’s matrix multiplication pseudo code

```
SQUARE-MATRIX-MULTIPLY-STRASSEN(A, B)
n = A.rows
if n == 1
```

$C_{11} = A_{11} * B_{11}$

```
else
```

$S_1 = B_{12} - B_{22}$

$S_2 = A_{11} + A_{12}$

$S_3 = A_{21} + A_{22}$

$S_4 = B_{21} - B_{11}$

$S_5 = A_{11} + A_{22}$

$S_6 = B_{11} + B_{22}$

$S_7 = A_{12} - A_{22}$

$S_8 = B_{21} + B_{22}$

$S_9 = A_{11} - A_{21}$

$S_{10} = B_{11} + B_{12}$

$P_1 =$ SQUARE-MATRIX-MULTIPLY-STRASSEN(A_{11}, S_1)

$P_2 =$ SQUARE-MATRIX-MULTIPLY-STRASSEN(S_2, B_{22})

$P_3 =$ SQUARE-MATRIX-MULTIPLY-STRASSEN(S_3, B_{11})

$P_4 =$ SQUARE-MATRIX-MULTIPLY-STRASSEN(A_{22}, S_4)

$P_5 =$ SQUARE-MATRIX-MULTIPLY-STRASSEN(S_5, S_6)

$P_6 =$ SQUARE-MATRIX-MULTIPLY-STRASSEN(S_7, S_8)

$P_7 =$ SQUARE-MATRIX-MULTIPLY-STRASSEN(S_9, S_{10})

$C_{11} = P_4 + P_5 + P_6 - P_2$

$C_{12} = P_1 + P_2$

$C_{21} = P_3 + P_4$

$C_{22} = P_1 + P_5 - P_3 - P_7$

```
return C
```