Recursive Matrix Multiplication

201811567_주용한

함수 간략 설명

```
// n = 행, 열의 크기
void multiply(n,...){
                                           // n==1 일 때, 두 값을 곱해
   if(n==1) output += matrixA * matrixB;
                                           output 행렬에 //더하며 재귀
                                           탈출
    else{
                                           // n!= 1일 때, n/2를 대입해
      multiply(n/2,...);
8번의
                                           //함수를 재귀(n==1 이 될 때
      multiply(n/2,...);
까지)
      multiply(n/2,...);
      multiply(n/2,...);
      multiply(n/2,...);
      multiply(n/2,...);
      multiply(n/2,...);
      multiply(n/2,...);
    }
}
```

Recursive Matrix Multiplication 시간 복잡도

```
n = 1 일 때, output += matrixA * matrixB;

T(merge) T(conquer)

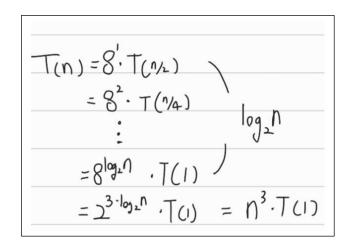
두 값을 곱한 후 결과를 output 행렬에 더한다

T(1) = C(상수)
```

Recursive Matrix Multiplication 시간 복잡도

n!= 1일 때, n 자리에 n/2 대입한 함수 8 번 재귀

```
T(n) = 8T(n/2)
= 8(8T(n/4))
= 8(8(8T(n/8)))
...
= 8^{\log(2,n)} T(1)
= 2^{3\log(2,n)} C
= C * n^{3}
```



big-O 시간 복잡도는 O(n^3) 이다.

Tiling 을 통한 공간 활용 개선

loop 을 통해 구현한 일반적인 함수의 간략 설명

```
for(int a = 0; k < n; a++){
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
        sum += matrixA[k][j] * matrixB[j][i]
        }
    }
}</pre>
```

함수가 호출되면 stack 메모리 위에 함수가 저장되고 함수가 끝나면 자동으로 stack 메모리 상에서 사라지게 된다.

위와 같은 코드로 함수를 만들면 모든 계산이 끝날 때까지 입력행렬의 모든 항을 읽고 stack 메모리에 쌓이게 된다. 하지만 Recursive Matrix Multiplication 의 구현 코드를 보면

각 항들을 바로 불러오는 게 아니라 n 의 값 즉 크기가 1 이 될 때까지 행렬을 잘게 나눈 다음 n==1 일 때 한 번만 계산하게 되며 자로 함수가 종료된다.

함수가 종료되면 stack 메모리에서 사라지게 되고 각각의 연산을 거치는 순간 바로 메모리에서 지워지게 된다.

따라서 Tiling 기법을 사용해 코드를 구현하면 모든 연산을 끝낼 때 까지 계속 메모리에 쌓이는 기존 방식과 다르게 연산 직후 메모리에서 사라져 높은 공간활용도를 갖을 수 있게 된다.