**Computer Algorithm**

**Chapter 02**

**Recursion**

**[실습] 재귀함수 구현**

**실습 목표**

* **재귀함수의 작동원리를 이해하고, 간단한 재귀함수를 직접 구현해볼 수 있다.**
* **반복문을 재귀함수로 변경할 수 있다.**

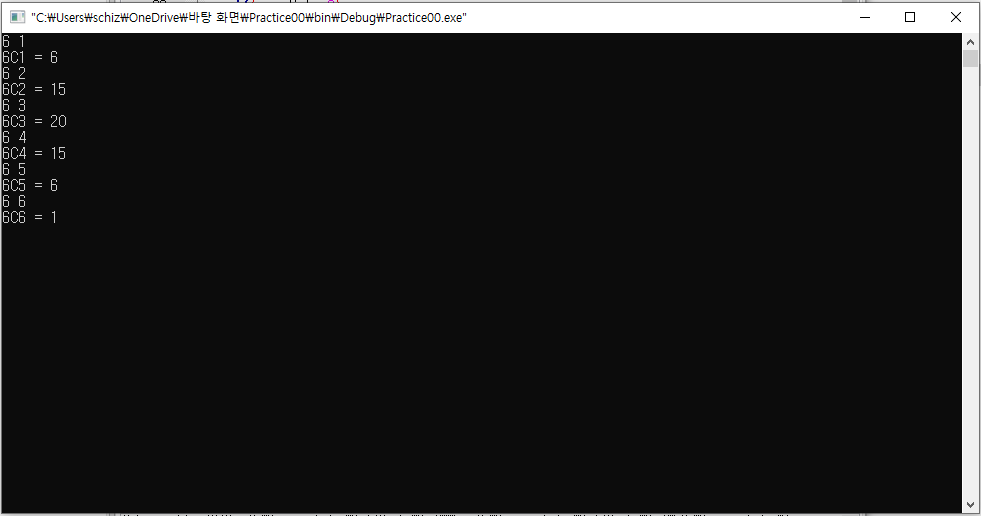
**요구사항**

* **실습과제 1) (25점)**

nCr 조합을 재귀를 통해 구현한다. 키보드에서 두 숫자 a, b를 입 력받아 aCb를 출력하는 프로그램을 재귀를 통해 구현한다.

구현은 배경지식을 참고하여 구현한다.

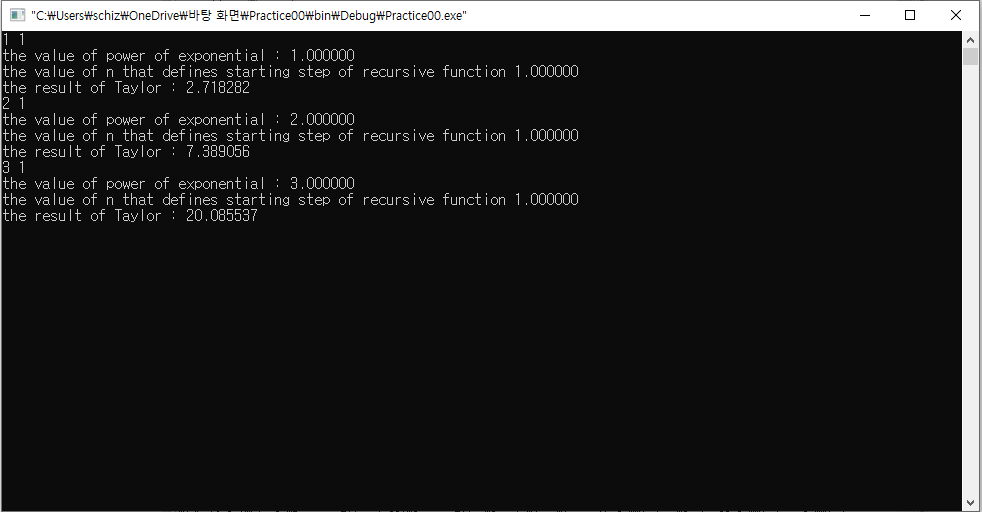
**//탈출설정시 0을 유의하라**



* **실습과제 2) (25점)**

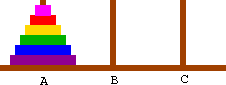
테일러 급수를 반복문이 아닌 재귀함수를 이용해 구현한다. 이 때 e^1, e^2, e^3, e^4, … , e^10을 재귀용법을 사용하여 화면에 출력하 는 프로그램을 작성한다. 테일러 급수에 관한 내용은 배경지식을 참고하도록 하며 여기서 급수 항의 숫자는 30으로 제한한다.

**//수학적으론 탈출설정이 무한이니 30으로 제한해야**



* **실습과제 3) (50점)**

하노이 타워문제를 재귀함수를 통해 해결한다.



하노이 타워 문제는 다음과 같다.

A에 있는 원반들을 B로 옮기려고 한다. 원반은 한개씩만 옮길 수 있고, 도중 작은 원반이 큰 원반 밑에 쌓여서는 안된다. A에 있는 원반을 B로 옮기는 방법을 출력한다.

입력은 C언어 표준 라이브러리를 통해서 입력받고, 원반의 갯수를 입력 받는다.

입력예시 : 3

출력예시 : move disk 0 from tray 1 to tray 3

(편의를 위해 A,B,C -> tray1,2,3)

move disk 1 from tray 1 to tray 2

move disk 0 from tray 3 to tray 2

move disk 2 from tray 1 to tray 3

move disk 0 from tray 2 to tray 1

move disk 1 from tray 2 to tray 3

move disk 0 from tray 1 to tray 3

/\*

예시에는 매개변수가 디스크갯수, 시작트레이, 목표트레이로 총 3개 이지만,

2개를 넘어서 3개를 입력하면 작은 디스크 위에 큰 디스크가 옮겨지는 현상이 발생.

중간에 경유하는 트레이가 다음 단계에서는 목표트레이가 되야하는 경우가 생긴다고

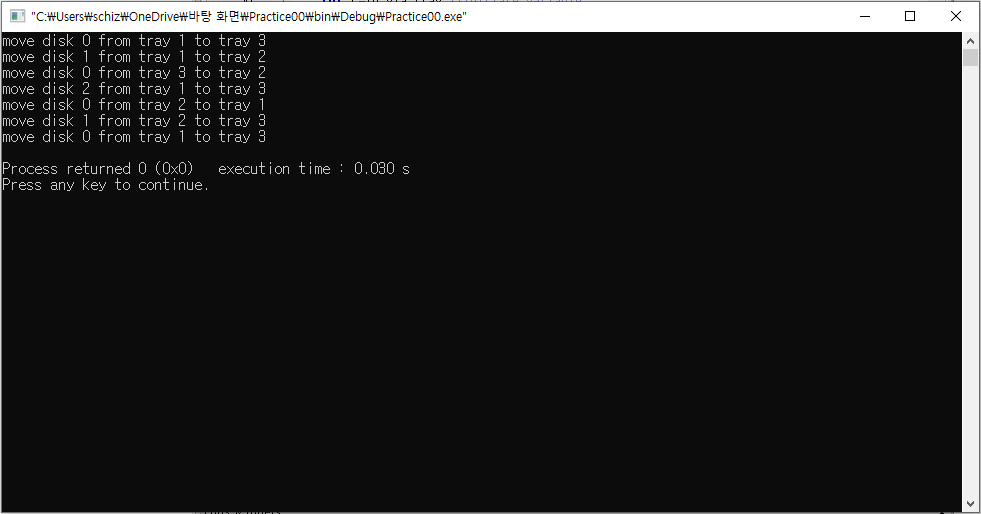
판단, 아쉽게도 수학적으로는 분석못해도 기계적으로 입력값을 하나더 추가함.

일단 문제는 해결했지만 문제의 해결과정을 온전히 이해못한 상태입니다.

높은 확률로 4이상의 경우에서는 문제를 일으킬 것을 생각됩니다.

더불어 아래 배경지식은 A트레이에서 B트레이로 옮겨가는 과정이지만 위의 예시는 A트레이에서 C트레이로 옮겨가는 예시라 주어진 예시에 맞추었습니다.

\*/



**배경지식**

재귀는 어떤 것을 정의할 때 자기 자신을 재참조하여 정의하는 것을 의미한다. 예를 들면 은 다음과 같이 정의할 수 있다.

위의 예에서 보다시피 을 자기 자신보다 낮은 차수를 가지는 부분집합 을 사용하여 정의하였으며 이러한 구조를 **재귀(Recurrence)**라고 한다.

대부분의 프로그래밍 언어에서 재귀호출을 허용하고 있으며 다음과 같은 형태로 구현된다..

**double** factorial (int *n*) // 재귀함수를 정의하는 부분

{

**if**( n == 0) { // 재귀함수의 탈출지점

**return** 1.0; //,

}

**return** (**double**) n \* factorial(*n-1*);

*//*

}

**void** main()

{

…

factorial(4); // 재귀함수를 호출하는 부분

…

}

위의 예에서 보이듯이, factorial(n) 함수를 차수가 낮은 재귀함수 factorial(n-1)와 탈출지점을 사용하여 정의하고 있으며, 위 재귀함수는 아래 그림과 같이 컴퓨터 내에서 실행된다.

본 실습에서 주어진 문제를 재귀용법을 통해 해결할 수 있도록 한다.

1)

nCr은 아래와 같이 정의된다.

은 combination(n, r) 의 형태의 함수로 구현할 수 있으며 이를 재귀호출을 함에 의해 구현이 가능한다. 아래는 코드의 일부이다. 이를 응용하여 나머지 구현을 완료할 수 있다.

**long** combination(int *n,* int *r*) // 재귀함수의 정의부분

{

**if**( r == 0 || r == n) {

**return** 1L; //

}

**return** ...

*//*

}

**void** main()

{

**int** i,j = 0;

**while**(scanf(“%d %d”, &i, &j) != EOF)

print(“%dC%d = %d\n”, i, j, combination(10,i)); // i!를 출력 i++;

}

**return** 0;

}

2)

를 테일러 전개(taylor series)를 하게 되면 다음과 같이 된다.

이때 재귀를 사용한 테일러 전개는 다음과 같이 정의할 수 있다.

위 식은 무한급수가 되지만 실제 계산을 하기 위해서는 충분히 유한한 횟수로 제한하여야 한다. 따라서 테일러 재귀를 통한 테일러 급수는 다음과 같이 정의할 수 있다.

위의 식을 재귀로 구현함에 의해 taylor 급수를 구현할 수 있다. 단 여기서 n의 수는 계속 증가하기 때문에 즉 1로 수렴하지 않으므로, 재귀 호출의 횟수(여기서는 n)를 제한함에 의해 재귀함수를 exit하도록 해야 한다.

3)

봉에 원반이 하나 있으면 다음과 같이 옮기면 된다.

Disk 0 : A 🡪 B

이 내용을 다음과 같이 출력한다.

// 프로그램 작성의 편의를 위해 A, B, C를 tray 1, tray 2, tray 3이라 한다.

Move disk 0 from tray 1 to tray 2.

봉에 원반이 n개 있으며 다음과 같이 옮기면 된다.

Disk {n-1..0} : A 🡪 C // 0부터 n-1까지의 원반을 A에서 C로 옮긴다.

Disk n : A 🡪 B. // 원반 n을 A에서 B로 옮긴다.

Disk {n-1..0} : C 🡪 B // 0부터 n-1까지의 원반을 C에서 B로 옮긴다.

여기서 Disk {n-1..0} : A 🡪 C와 Disk {n-1..0} : C 🡪 B 를 재귀로 구현하면 Hanoi 문제를 해결할 수 있다. Disk n : A 🡪 B. 는 “Move disk 0 from tray 1 to tray 2”를 출력하는 명령어를 작성하면 된다.

예를 들면 Disk {n-1..0} : A 🡪 C를 move(n-1, a, c)라는 함수로 작성한다고 하면 재귀용법을 사용하여 다음과 같은 형태로 구현할 수 있다.

void move(n, a, b)

... // 변수 초기화

... // 탈출지점 및 처리방법 정의

/\* 재귀를 사용하여 구현 \*/

move(n-1, a, c)

print("Move disk n from tray a to tray b")

move(n-1, c, b)

이 때 탈출 지점 (exit point)를 정확하게 정의하여 프로그램이 무한이 돌도록 하지 말아야 한다. 위 코드는 슈도코드로 c언어로 구현할 경우 다르게 표현된다는 점 또는 고려해야 한다.

**제출방법**

* 보고서 작성방법: 실습문제 번호별로 결과가 나온 화면의 내용을

캡쳐하여 보고서에 붙여 놓는다.

* 소스코드의 파일이름에 연습문제 번호를 붙이는 것을 잊지 않는다. 예) ex-1.c, ex-2.c
* 결과 보고서에 이름과 작성 날짜를 기입하는 것을 잊지 않는다. 예) 김웅섭\_2020\_09\_01.doc
* 실행결과를 보고서에 작성하여 소스코드와 함께 제출한다.
* 제출 마감 : e-class 제출 마감시간까지