컴퓨팅 사고와 SW 코딩

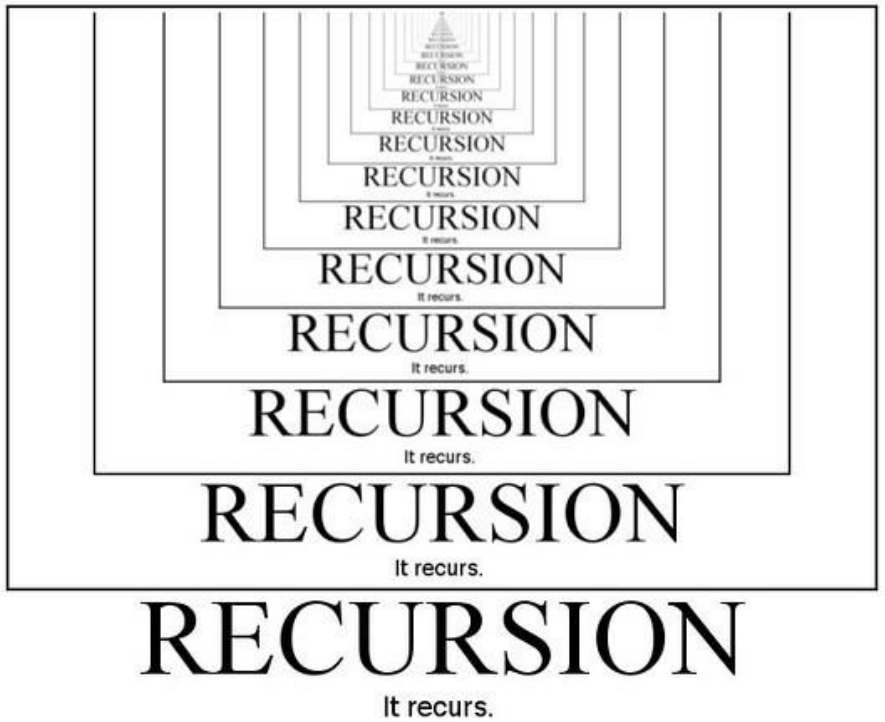
03

재귀

경북대학교 배준현 교수
(joonion@knu.ac.kr)

03. 재귀

- **재귀**: *recursion*
  - 어떤 문제를 정의할 때 **자기 자신을 참조**하여 정의하는 것



03. 재귀

재귀적으로 생각하기

- 재귀적 관계: *recurrence relation* = *recurrence equation*
  - 어떤 문제를 더 작은 부분 문제를 이용하여 정의하기 (재귀식, 점화식)
- 기저 조건: *base condition* = *exit condition*
  - 더 이상 재귀를 하지 않아도 답을 알 수 있는 종료 조건 찾기

$$n! = n \times (n-1)!$$

$$0! = 1$$

03. 재귀

- **재귀 함수: *recursive function***
  - **재귀적 관계를 이용하여 함수를 정의하는 것**

재귀 함수: 더 작은 값을 가지고 자기 자신을 호출하는 함수

재귀함수의 특성
가독성이 좋다
스택 메모리를 사용한다

반복문은 힙메모리를 사용한다

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

종료조건이 중요함!!

03. 재귀

- **재귀 함수와 호출 스택: *call stack***

stack: FIFO

  - **호출 스택: 현재 실행 중인 함수의 정보를 저장하고 있는 스택 자료구조**

```
fact(5)
  5 * fact(4)
      4 * fact(3)
          3 * fact(2)
              2 * fact(1)
                  1 * fact(0)
                      return 1
```

```
n = 0
return 1
```

```
n = 1
return 1 * fact(0)
```

```
n = 2
return 2 * fact(1)
```

```
n = 3
return 3 * fact(2)
```

```
n = 4
return 4 * fact(3)
```

```
n = 5
return 5 * fact(4)
```

call stack

03. 재귀

- **재귀와 반복: *recursion* .vs. *iteration***
  - **모든 재귀 함수는 반복문으로 구현할 수 있을까?**

반복하는 횟수는 동일한데, 반복문이 더 빠르다
컴퓨터 성능이 별로 안좋을 때에는 반복문으로 사용하는 것이 더 좋았다

```
def fact(n):
    if n == 0:
        return 1
    else:
        res = 1
        for i in range(1, n + 1):
            res *= i
        return res
```

s= [3, 7, 9, 2, 4, 8]

- 반복문
```
for i in range(0: len(s)):
   ss+=s[i]
```

- 재귀
```
def sum(s):
  if len(s)==1:
     return s[0]
  else
      return s[0]+sum(s[1:])
```

재귀함수와 반복문의 차이점
https://velog.io/@gillog/Algorithm-%EC%9E%AC%EA%B7%80%EC%99%80-%EB%B0%98%EB%B3%B5%EB%AC%B8

for문은 반드시 재귀함수로 바꿀 수 있다
함수만 여러개 있는게 functional programming(for문이 없다)

모든 재귀함수는 for문으로 변환할 수 있다

	반복문	재귀함수
기본	명령을 반복적으로 실행	함수 자체를 호출
체재	초기화, 조건, 루프 내 명령문 실행과 제어 변수 업데이트 포함	종료 조건만 지정(조건이 추가될 수 도 있음)
종료	설정한 조건에 도달 할 때까지 반복 실행	함수 호출 본문에 조건부가 포함, 재귀를 호출하지 않고 함수를 강제 반환
조건	제어 조건이 참이라면 무한 반복 발생	조건에 수렴하지 않을 경우 무한 재귀 발생
무한 반복	무한 루프는 CPU 사이클을 반복적으로 사용	무한 재귀는 스택 오버플로우 발생
스택 메모리	스택 메모리를 사용하지 않음	함수가 호출 될 때마다 새 로컬 변수와 매개 변수 집합, 함수 호출 위치를 저장하는데 사용
속도	빠른 실행	느린 실행
가독성	코드 길이가 길어지고 변수가 많아져 가독성이 떨어짐	코드 길이와 변수가 적어 가독성이 높아짐

03. 재귀

BOJ 10872: 팩토리얼

- 문제:
  - 0보다 크거나 같은 정수 N이 주어진다.
  - 이때, N!을 출력하는 프로그램을 작성하시오.
- 입력:
  - 첫째 줄에 정수 $N(0 \le N \le 12)$이 주어진다.
- 출력:
  - 첫째 줄에 N!을 출력한다.

03. 재귀

예제 입력	예제 출력
10	3628800
0	1

03. 재귀

BOJ 4564: 숫자 카드놀이

- 문제:
  - 숫자 카드놀이를 하려면 먼저 자연수 S를 골라야 한다.
  - 그 다음, 각 자리를 모두 곱하면서 한 자리 수가 나오기 전까지 계속해서 새로운 수를 만드는 게임이다.
  - 자연수 S가 주어졌을 때, 숫자 카드놀이를 하는 과정을 출력하는 프로그램을 작성하시오.

95
9 × 5 = 45
4 × 5 = 20
2 × 0 = 0

396
3 × 9 × 6 = 162
1 × 6 × 2 = 12
1 × 2 = 2

03. 재귀

- 입력:
  - 입력은 여러 테스트 케이스로 이루어져 있다.
  - 각 테스트 케이스는 숫자 카드놀이의 시작값 S로 이루어져 있다. $(1 \le S \le 100000)$
  - S는 0으로 시작하지 않으며, 입력의 마지막 줄에는 0이 하나 주어진다.
- 출력:
  - 0이 아닌 입력에 대해서, 숫자 카드놀이가 끝나기 전까지 나온 수를 공백으로 구분하여 출력한다.
  - 첫 값은 입력으로 주어진 값이다.

03. 재귀

예제 입력

```
95
396
28
4
40
0
```

예제 출력

```
95 45 20 0
396 162 12 2
28 16 6
4
40 0
```

03. 재귀

```
def solve(n):
    print(n, end = " ")
    s = str(n)
    if len(s) > 1:
        prod = 1
        for i in range(len(s)):
            prod *= int(s[i])
        solve(prod)
```

03. 재귀

- **콜라츠 추측: *Collatz conjecture***
  - 콜라츠 추측을 설명하면 다음과 같다.
  - 우선 다음과 같은 양의 정수 수열 x_i 를 생각하자.
    - 만약 x_i 가 짝수이면, $x_{i+1}=x_i/2$
    - 만약 x_i 가 홀수이면, $x_{i+1}=3*x_i+1$ 이다.
  - 콜라츠 추측은 이렇게 만든 수열은 결국 1이 된다는 것이다.
  - 과학자들은, 컴퓨터를 이용하여 첫 번째 수열이 2^{58} 보다 작으면, 이 추측은 참이라고 증명했다.
  - 콜라츠 추측은 흥미로운 현상이다. 이 법칙은 간단해보이지만, 수학적으로 아직까지 증명되어있지 않은 문제이다.
  - 우리는 이 추측이 옳다고 받아들이겠다.

03. 재귀

```
def collatz(n):
    if n == 1:
        return [1]
    elif n % 2 == 0:
        return [n] + collatz(n // 2)
    else:
        return [n] + collatz(3*n + 1)

for n in range(1, 11):
    print(collatz(n))
```

리스트형태로 합침

03. 재귀

- BOJ 6615: 콜라츠 추측
  - 문제:
    - 두개의 양의 정수를 준다.
    - 각각의 수에 대해서 콜라츠 추측으로 만든 수열을 생각하자.
    - 각각의 수열을 비교하였을때 처음으로 같은 숫자가 나왔을때 ,
      - 각각 몇번째 수열에서 만나는지 구해본다.
    - 문제의 편의를 위해, 이 수열은 1이 나오면 더이상 진행하지 않는다고 하자.
      (1 다음에 나올 수열을 생각하면, 1, 4, 2, 1, 4, 2, 1로 반복되기 때문이다.)

03. 재귀

- 입력:
  - 입력은 몇개의 테스트 케이스로 구성된다.
  - 각 테스트 케이스는 두개의 정수 A와 B가 주어진다. ($1 \le A, B \le 1{,}000{,}000$)
  - 마지막 줄은 두개의 0으로 구성된다.
- 출력:
  - 각각의 테스트 케이스마다 다음과 같은 문장을 한줄에 출력한다.
  - "A needs S_A steps, B needs S_B steps, they meet at C"
  - S_A와 S_B는 A와 B로 수열을 만들고, 처음으로 같은 숫자 C가 나왔을때
    - 각각의 수열에서 몇번째 인지 알려주는 숫자이다.

03. 재귀

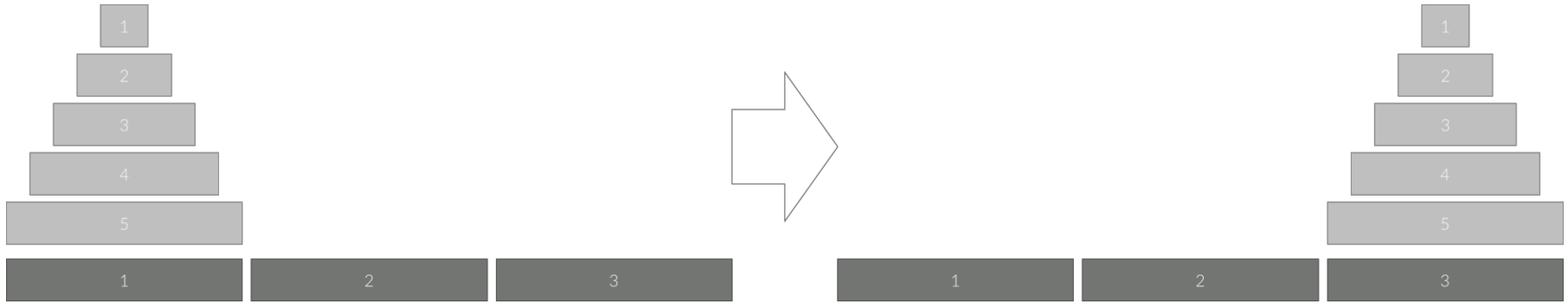
```
def solve(A, B):
    a = collatz(A)[::-1]
    b = collatz(B)[::-1]
    minlen = min(len(a), len(b))
    i = 0
    while True:
        if i == minlen or a[i] != b[i]:
            break
        i += 1
    return len(a) - i, len(b) - i, a[i - 1]
```

03. 재귀

```
while True:
    A, B = map(int, input().split())
    if A == 0 and B == 0:
        break
    a1, a2, a3 = solve(A, B)
    print(f"{A} needs {a1} steps, {B} needs {a2} steps, they meet at {a3}")
```

- BOJ 1914: 하노이 탑
  - 세 개의 장대가 있고 첫 번째 장대에는 반경이 서로 다른 n개의 원판이 쌓여 있다.
  - 각 원판은 반경이 큰 순서대로 쌓여있다.
  - 이제 수도승들이 다음 규칙에 따라 첫 번째 장대에서 세 번째 장대로 옮기려 한다.
    - 한 번에 한 개의 원판만을 다른 탑으로 옮길 수 있다.
    - 쌓아 놓은 원판은 항상 위의 것이 아래의 것보다 작아야 한다.
  - 이 작업을 수행하는데 필요한 이동 순서를 출력하는 프로그램을 작성하라.
    - 단, 이동 횟수는 최소가 되어야 한다.

(2^n)-1



03. 재귀

(2^n)-1

- 입력:
  - 첫째 줄에 첫 번째 장대에 쌓인 원판의 개수 N ($1 \le N \le 100$)이 주어진다.
- 출력:
  - 첫째 줄에 옮긴 횟수 K를 출력한다.
  - N이 20 이하인 입력에 대해서는 두 번째 줄부터 수행 과정을 출력한다.
  - 두 번째 줄부터 K개의 줄에 걸쳐 두 정수 A B를 빈칸을 사이에 두고 출력하는데, 이는 A번째 탑의 가장 위에 있는 원판을 B번째 탑의 가장 위로 옮긴다는 뜻이다.
  - N이 20보다 큰 경우에는 과정은 출력할 필요가 없다.

03. 재귀

예제 입력

```
3
```

예제 출력 (2^n)-1

```
7
1 3
1 2
3 2
1 3
2 1
2 3
1 3
```

03. 재귀

```
                    출발  목적지  경유지
def hanoi(n, src, dst, via):
    if n == 1:
        print(src, dst)
    else:
                          출발  경유지  목적지
        hanoi(n - 1, src, via, dst)
        print(src, dst)
                          경유지  목적지  출발
        hanoi(n - 1, via, dst, src)
```

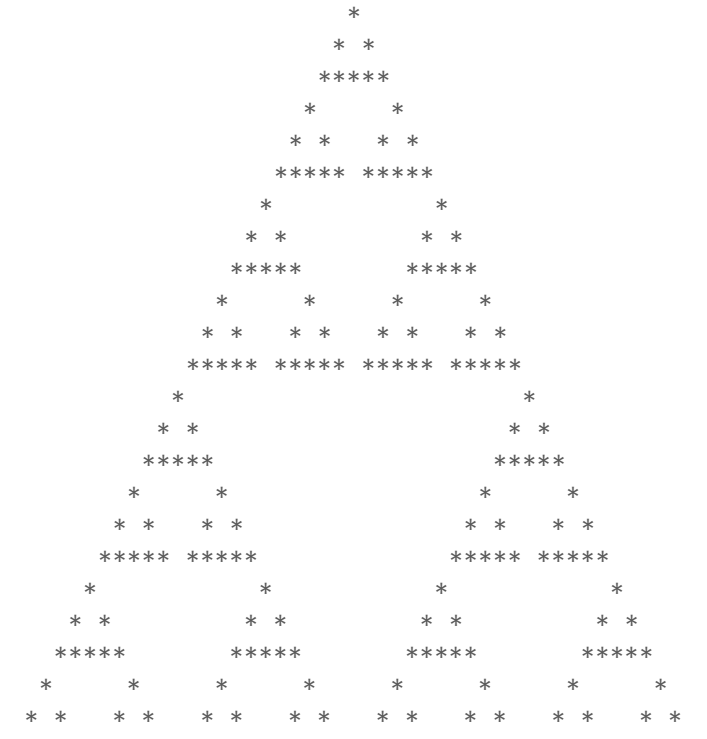
an=2(a(n-1))+1
a1=1
an=(2^n)-1

- BOJ 2448: 별 찍기 11
  - 문제:
    - 예제를 보고 규칙을 유추한 뒤에 별을 찍어 보세요.
  - 입력:
    - 첫째 줄에 N이 주어진다.
    - N은 항상 3×2^k 수이다. (3, 6, 12, 24, 48, ...) ($0 \le k \le 10$, k는 정수)
  - 출력:
    - 첫째 줄부터 N번째 줄까지 별을 출력한다.

예제 입력

24

예제 출력



```
def sierpinski(n, T, row, col):
    if n == 3:
        T[row][col] = 1
        T[row + 1][col - 1] = T[row + 1][col + 1] = 1
        for i in range(-2, 3):
            T[row + 2][col + i] = 1
    else:
        m = n // 2
        sierpinski(m, T, row, col)
        sierpinski(m, T, row + m, col - m)
        sierpinski(m, T, row + m, col + m)
```

03. 재귀

```
def solve(n):
    T = [[0] * (2*n - 1) for _ in range(n)]
    sierpinski(n, T, 0, n - 1)
    s = ""
    for i in range(N):
        for j in range(2*N - 1):
            s += "*" if T[i][j] == 1 else " "
        s += "\n"
    print(s)
```

03. 재귀

- **더 풀어볼 문제:**
  - BOJ 2057: **팩토리얼 분해**
  - BOJ 16953: A → B
  - BOJ 3943: **헤일스톤 수열**
  - BOJ 5393: **콜라츠**
  - BOJ 2270: **하노이 탑**
  - BOJ 2447: **별 찍기 10**

Any Questions?