



본 PSet 은 저의 강의 경험과 학생들의 의견 및 Stanford CS106 과 Harvard CS50 같은 강의에서 수집된 자료를 토대로 작성되었습니다.
본 PSet 에 문제점이나 질문 혹은 의견이 있다면, 저의 이메일(idebtor@gmail.com)로 알려 주시면 강의 개선에 많은 도움이 되겠습니다.

PSet - Recursion

목차

| | |
|---|----|
| Getting Started - Recursion..... | 1 |
| 제공되는 파일 | 2 |
| Step 1: 재귀 함수 구현하기 | 2 |
| Example 1: Factorial | 2 |
| Step 2: 재귀 함수 구현하기 | 5 |
| Example 2: GCD(Greatest Common Divisor) | 5 |
| Example 3: Fibonacci | 6 |
| Example 4: Bunny Ears | 7 |
| Example 5: Funny Ears | 7 |
| Example 6: Triangle | 8 |
| Example 7: Sum of digits..... | 8 |
| Example 8: Count 8..... | 8 |
| Example 9: Power N | 9 |
| 빌드를 위한 명령줄..... | 9 |
| 과제 제출 | 9 |
| 제출 파일 목록 | 9 |
| 마감 기한 & 배점..... | 10 |

Getting Started - Recursion

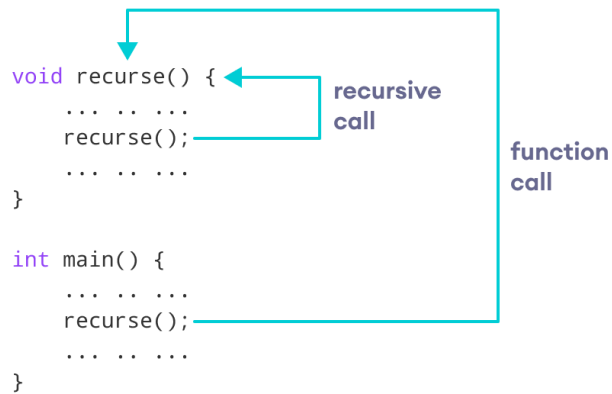
컴퓨터 과학에서의 **재귀(Recursion)**는 (iteration 과 달리) 어떠한 문제의 해결책을 동일한 문제의 **더 작은 인스턴스**의 해결책에 의존하는 방법입니다. 이 접근법은 여러 종류의 문제에 적용될 수 있으며, 재귀는 컴퓨터 과학의 중심 아이디어 중 하나입니다. 프로그램에 명시적 반복이 포함되어 있지 않더라도 무수히 많은 연산을 한정된 수의 재귀 프로그램으로 설명할 수 있습니다. 대부분의 컴퓨터 프로그래밍 언어는 [함수](#)가 프로그램 텍스트 내에서 함수 자신을 호출할 수 있도록 함으로써 재귀를 지원합니다.

(Resources: <https://en.wikipedia.org/wiki/Recursion> & www.codingBat.com)

재귀 알고리즘은 다음과 같은 용어로 표현됩니다.

1. **base case(s)**: 해결책을 재귀 없이 명시할 수 있는 경우
2. **recursive case(s)**: 해결책을 자신의 더 작은 버전으로 표현할 수 있는 경우

다음 그림은 재귀가 자기 자신을 반복적으로 호출하여 작동하는 방식을 보여줍니다.



제공되는 파일

이 PSet 에는 다음 파일들이 제공됩니다:

1. recursion.cpp, driver.cpp - 뼈대 코드
2. recursion.exe, recursion - 실행 답안 예시

Step 1: 재귀 함수 구현하기

강의에서 본 것과 같이 단순 팩토리얼 함수로 재귀를 연습합니다.

- factorial.cpp 라는 파일을 생성하고 이 예시를 실행해보세요.
- factorial() 함수에는 네 가지 스타일이 존재합니다. 이 네 가지 스타일을 다 시도한 후 마지막 양식인 “코드 예시 4”를 PSet 의 일부로 제출합니다.

Example 1: Factorial

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{fact}(n - 1) & \text{if } n > 0 \end{cases}$$

factorial(n) - iterative solution

```

long long unsigned factorial(n) {
    long long total = 1;
    for (int i = n; i > 1; i--) total *= i;
    return total;
}

```

factorial(n) - recursive solution

```

input: integer  $n$  such that  $n \geq 0$ 
output:  $[n \times (n-1) \times (n-2) \times \dots \times 1]$ 
    1. if  $n$  is 0, return 1
    2. otherwise, return  $[n \times \text{factorial}(n-1)]$ 
end factorial

```

```

factorial(1) = 1
factorial(2) = 2
factorial(3) = 6
factorial(4) = 24
factorial(5) = 120
factorial(8) = 40320
factorial(12) = 479001600
factorial(20) = 2432902008176640000

```

힌트:

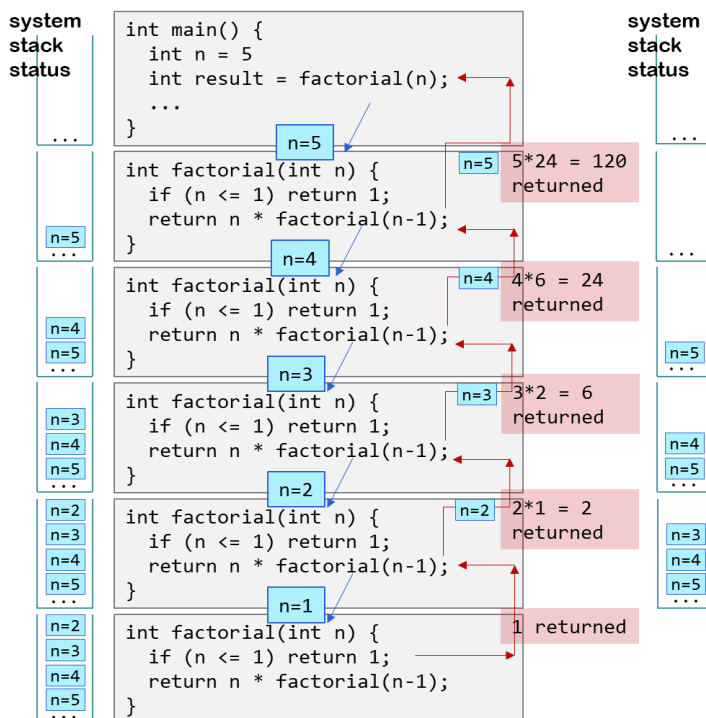
먼저 답을 즉각적으로 반환할 정도로 간단한 “base case”를 찾습니다 (이 예시에서는 $n == 1$ 또는 $n == 0$ 인 경우입니다). Base case 가 아닌 경우 (base case 에 접근하도록) $\text{factorial}(n-1)$ 을 재귀 호출합니다. 재귀 호출이 올바른 값을 반환하고 해당 값을 수정하여 결과를 낸다고 가정합니다.

이 문제를 재귀로 해결하려면 이 문제의 하위 문제를 알아내야 합니다. 문제를 세세하게 나눠봅시다.

1. $\text{factorial}(5) = 5 * \text{factorial}(4)$ aka $5! = 5 * 4!$ 입니다.
2. 더 나아가, $\text{factorial}(5) = 5 * (4 * \text{factorial}(3))$ 는 $5 * (4 * (3 * \text{factorial}(2)))$ 와 같고 이후도 동일한 방식으로 이어집니다.
3. $5 * 4 * 3 * 2 * 1$ 과 $1!$ 이 유일한 하위 문제가 될 때까지...
4. $\text{factorial}(1)$ 과 $\text{factorial}(0)$ 은 항상 1 이므로 base case 가 됩니다.

아래 그림에서 볼 수 있듯이 $\text{factorial}()$ 함수는 자기 자신을 호출하고 있습니다. 그러나, 매 호출마다 n 의 값을 1 씩 줄이고 있습니다. n 이 1 보다 작아지면, $\text{factorial}()$ 함수는 출력을 반환합니다.

함수를 마치지 않은 상태로 또 다른 함수를 계속해서 호출하면 함수의 현재 상태 혹은 n 의 값이 시스템 스택에 저장됩니다. 그런 다음 $\text{factorial}(n-1)$ 로부터 값이 반환되는 즉시 n 의 값이 시스템 스택에서 pop 되고 연산을 위해 되돌아옵니다.



이러한 사고방식을 통해 아래와 같은 팩토리얼 문제에 대한 재귀적인 해답을 작성할 수 있습니다:

코드 예시 1:

```
long long unsigned factorial(int n) {
    if (n == 1 || n == 0) return n;
    return n * factorial(n-1);
}
```

코드 예시 2:

```
long long unsigned factorial(int n) {
    return (n == 1 || n == 0) ? 1 : n * factorial(n - 1);    // using ternary operator
}
```

코드 예시 3:

```
#include <iostream>

long long unsigned factorial(int n) {
    std::cout << "n= " << n << std::endl;
    if (n == 1 || n == 0) return n;
    auto result = n * factorial(n - 1);
    std::cout << "n= " << n << "\tn!= " << result << std::endl;
    return result;
}

int main() {
    factorial(5);
}
```

실행 예시:

```
PS C:\GitHub\nowicx\psets\pset02recursion> g++ factorial.cpp -o factorial
PS C:\GitHub\nowicx\psets\pset02recursion> ./factorial
n= 5
n= 4
n= 3
n= 2
n= 1
n= 2    n!= 2
n= 3    n!= 6
n= 4    n!= 24
n= 5    n!= 120
PS C:\GitHub\nowicx\psets\pset02recursion> █
```

첫 번째 함수 호출은 **n = 5**에서 시작되지만, 4, 3, 2, 1의 모든 팩토리얼이 먼저 계산될 때까지 끝나지 않습니다. Main()에서는 함수를 한 번 호출하지만, 함수 내에서 **n = 1** 또는 base case에 도달할 때까지 함수 자신을 자체적으로 호출합니다.

코드 예시 4:

```
factorial(1) = 1
factorial(2) = 2
factorial(3) = 6
factorial(4) = 24
```

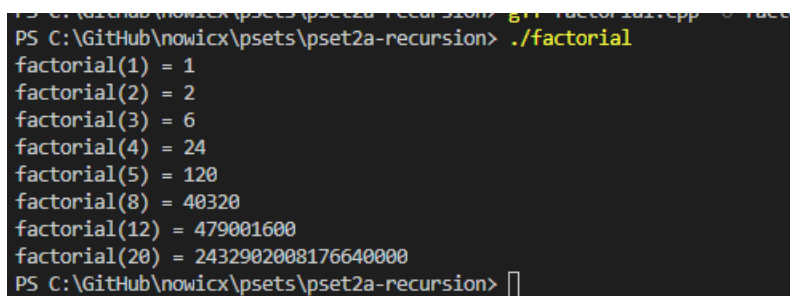
```
factorial(5) = 120
factorial(8) = 40320
factorial(12) = 479001600
factorial(20) = 2432902008176640000
```

```
#include <iostream>

long long unsigned factorial(int n) {
    if (n == 1 || n == 0) return n;
    auto result = n * factorial(n - 1);
    return result;
}

int main() {
    int n[] = {1, 2, 3, 4, 5, 8, 12, 20};
    for (auto x: n)
        std::cout << "factorial(" << x << ") = " << factorial(x) << std::endl;
    return 0;
}
```

실행 예시:



```
PS C:\GitHub\nowic\psets\pset2a-recursion> g++ factorial.cpp -o factorial
PS C:\GitHub\nowic\psets\pset2a-recursion> ./factorial
factorial(1) = 1
factorial(2) = 2
factorial(3) = 6
factorial(4) = 24
factorial(5) = 120
factorial(8) = 40320
factorial(12) = 479001600
factorial(20) = 2432902008176640000
PS C:\GitHub\nowic\psets\pset2a-recursion> 
```

손 또는 디버거를 이용해 코드를 하나하나 따라가보며 재귀의 개념을 이해하길 권장합니다.

“코드 예시 4”의 코드 스타일을 따라 이 pset의 나머지 재귀 코드를 완성하세요.

Step 2: 재귀 함수 구현하기

`factorial()`을 통해 재귀 함수를 연습해봤으니 더 많은 재귀 함수를 구현해봅시다.

1. 두 재귀 함수 `factorial()`과 `gcd()`는 다음 예제로 `recursion.cpp`에 이미 코딩되어 있습니다.
2. `recursion.cpp`와 `recursionDriver.cpp`에서 예시 1 ~ 2의 소스코드를 참조하고 `recursion.exe`를 실행해보면서 `recursion.cpp`에 `factorial()`을 포함한 나머지 예시를 구현하세요.

Example 2: GCD(Greatest Common Divisor)

$$\text{gcd}(x, y) = \begin{cases} x & \text{if } y = 0 \\ \text{gcd}(y, \text{remainder}(x, y)) & \text{if } y > 0 \end{cases}$$

최대 공약수에 대한 반복 관계($x \% y$ 는 x/y 의 나머지를 나타냅니다):

$$\begin{aligned} \text{gcd}(x, y) &= \text{gcd}(y, x \% y) & \text{if } y \neq 0 \\ &= \text{gcd}(x, 0) = x & \text{if } y = 0 \end{aligned}$$

```
gcd(x, y)
```

```
input: integer x, y such that  $x \geq y$ ,  $y > 0$ 
```

```
output: gcd of x and y
```

```
1. if y is 0, return x
```

```
2. otherwise, return [ gcd (y,  $x \% y$ ) ]
```

```
end gcd
```

예) $x = 27$ 과 $y = 9$ 의 반복 관계 계산

```
gcd(27, 9) = gcd(9, 27 % 9)
           = gcd(9, 0)
```

예) $x = 111$ 과 $y = 259$ 의 반복 관계 계산

```
gcd(111, 259) = gcd(259, 111 % 259)
              = gcd(259, 111)
              = gcd(111, 259 % 111)
              = gcd(111, 37)
              = gcd(37, 111 % 37)
              = gcd(37, 0)
              = 37
```

코드:

```
int gcd(int x, int y) {
    if (y == 0) return x;
    return gcd(y, x % y);
}
```

또는

```
int gcd(int x, int y) {
    return y == 0 ? x : gcd(y, x % y);
}
```

Example 3: Fibonacci

피보나치수열은 수학의 유명한 정의 중 하나이며 재귀적인 정의를 가지고 있습니다. 수열의 처음 두 값은 0 과 1(기본적인 2 개의 base case)입니다. 각 값은 이전 두 값의 합이므로 전체 수열은 0, 1, 1, 2, 3, 5, 8, 13, 21... 등등입니다.

$n = 0$ 이 수열의 시작을 나타내며 n 번째 피보나치 숫자를 반환하는 재귀 fibonacci(n) 방법을 정의하세요.

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(3) = 2
fibonacci(4) = 3
fibonacci(5) = 5
fibonacci(11) = 89
fibonacci(33) = 3524578
fibonacci(44) = 701408733 (계산에 시간이 조금 걸립니다.)
```

코드:

```
long long unsigned fibonacci(int n) {
    cout << "your code here\n";
}
```

Example 4: Bunny Ears

몇 마리의 토끼가 있습니다. 각 토끼는 두 개의 큰 귀를 가지고 있습니다. 모든 토끼의 총 귀의 개수를 재귀적으로 (loop 또는 곱셈 없이) 계산하려고 합니다.

```
bunnyEars(0) = 0
bunnyEars(1) = 2
bunnyEars(2) = 4
bunnyEars(3) = 6
bunnyEars(234) = 468
```

힌트:

먼저 base case(bunnies == 0)을 찾고, 그 경우에 0 을 반환합니다. Base case 가 아닌 경우, bunnyEars(bunnies-1)을 재귀 호출합니다. 재귀 호출이 올바른 값을 반환한다고 믿고 2 를 더하여 값을 고정합니다.

코드:

```
int bunnyEars(int bunnies) {
    if (bunnies == 0) return 0;

    // Recursive case: otherwise, make a recursive call with bunnies-1
    // (towards the base case), and fix up what it returns.

    cout << "your code here\n";
}
```

Example 5: Funny Ears

토끼와 독기가 1, 2, ... 번호대로 줄을 서있습니다. 홀수 토끼(1, 3, ...)는 평범하게 2 개의 귀를 가지고 있습니다. 짝수 독기(2, 4, ...)는 발이 올라가 있어 3 개의 귀를 가지고 있다고 볼 수 있습니다. 1, 2, ... n 까지 줄을 서있는 토끼와 독기의 귀의 개수를 재귀적으로 (loop 또는 곱셈 없이) 반환합니다.

```
funnyEars(0) = 0
funnyEars(1) = 2
funnyEars(2) = 5
funnyEars(3) = 7
funnyEars(4) = 10
funnyEars(9) = 22
funnyEars(10) = 25
```

코드:

```
int funnyEars(int funnies) {
    // your code here
}
```

Example 6: Triangle

블록으로 만든 삼각형이 있습니다. 맨 위의 행은 블록 1 개, 다음 행은 블록 2, 그다음 행은 블록 3 개 등등 동일한 방식으로 이어집니다. 주어진 행의 수를 이용해서 이러한 삼각형의 총 블록의 개수를 재귀적으로 (loop 또는 곱셈 없이) 계산하세요.

```
triangle(0) = 0
triangle(1) = 1
triangle(2) = 3
triangle(3) = 6
triangle(4) = 10
triangle(7) = 28
```

코드:

```
int triangle(int rows) {
    // your code here
}
```

Example 7: Sum of digits

음수가 아닌 정수 n 이 주어지면 각 자릿수의 합을 (loop 없이) 재귀적으로 반환합니다. 나머지(%) 10 을 하면 가장 오른쪽 자릿수($126 \% 10$ 은 6)를 구할 수 있는 반면 나누기(/) 10 을 하면 가장 오른쪽 자릿수($126 / 10$ 은 12)를 제거합니다.

```
sumDigits(126) = 9
sumDigits(12) = 3
sumDigits(1) = 1
sumDigits(10110) = 3
sumDigits(235) = 10
```

코드:

```
int sumDigits(int n) {
    // your code here
}
```

Example 8: Count 8

음수가 아닌 정수 n 이 주어지면 8 의 개수를 세어 반환합니다. 예를 들어, 818 인 경우 2 를 반환합니다. Loop 를 사용하지 마세요. 나머지(%) 10 을 하면 가장 오른쪽 자릿수($126 \% 10$ 은 6)를 구할 수 있는 반면 나누기(/) 10 을 하면 가장 오른쪽 자릿수($126 / 10$ 은 12)를 제거합니다.

```
count8(818) = 2
count8(8) = 1
count8(123) = 0
count8(881238) = 3
count8(48581) = 2
count8(888586198) = 5
count8(99899) = 1
```

코드:

```
int count8(int n) {
    // your code here
}
```


Example 9: Power N

Base 와 n 에 모두 1 이상의 값이 주어지면 base 의 n 제곱을 재귀적으로 (loop 없이) 계산합니다. 예를 들어, powerN(3, 2)는 9(3 의 제곱)입니다.

```
powerN(2, 5) = 32
powerN(3, 1) = 3
powerN(3, 2) = 9
powerN(3, 3) = 27
powerN(10, 2) = 100
powerN(10, 3) = 1000
```

코드:

```
long long powerN(int base, int n) {
    // your code here
}
```

빌드를 위한 명령줄

프로그램을 빌드하기 위해 다음과 같이 명령줄을 작성합니다.

```
g++ recursion.cpp driver.cpp -I../include -L../lib -lnowic -o recursion
```

이 명령줄은 다음 폴더 구조 및 파일이 존재하는 pset2a 폴더에서 작동합니다.

```
nowic/lib/libnowic.a
nowic/include/nowic.h
nowic/psets/pset2a/recursion.cpp
nowic/psets/pset2a/driver.cpp
```

과제 제출

- 소스 파일 상단에 아래와 같이 아너 코드 문장을 적고 서명하세요.
On my honor, I pledge that I have neither received nor provided improper assistance in the completion of this assignment.
서명: _____ 분반: _____ 학번: _____
- 제출하기 전에 코드가 제대로 컴파일되고 실행되는지 확인하세요. 제출 직전에 급하게 코드를 수정한 후 코드가 제대로 컴파일될 거라고 짐작하지 않는 게 좋습니다. “거의” 작동하는 코드도 틀린 것입니다.
- 과제가 컴파일 및 실행된다면, 마감 기한 전까지 과제의 일부만 완성했더라도 제출하기 바랍니다. 컴파일 및 실행되지 않는다면 제출하지 마세요. 마감 시간 이후 24 시간 이내 제출하면, 만점에서 25% 감점하고 채점합니다. 그 이상 늦은 것은 채점하지 않으며, 0 점 처리합니다.
- 제출 후, **마감 기한 전까지** 수정 및 재제출이 가능합니다. 파일 하나만 수정하더라도 해당 파일과 관련된 파일들을 모두 재제출해야 합니다. 재제출 횟수는 제한 없습니다. 마감 기한 전에 **가장 마지막으로** 제출된 파일을 채점할 것입니다.

제출 파일 목록

다음 파일들을 piazza 폴더에 제출하세요.

- factorial.cpp

- recursion.cpp
- driver.cpp

마감 기한 & 배점

- 마감 기한: 11:55 pm