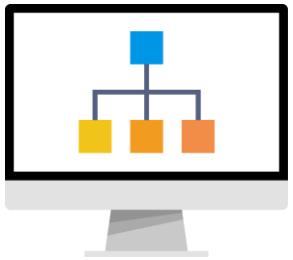


Data Structure & Algorithm
자료구조 및 알고리즘

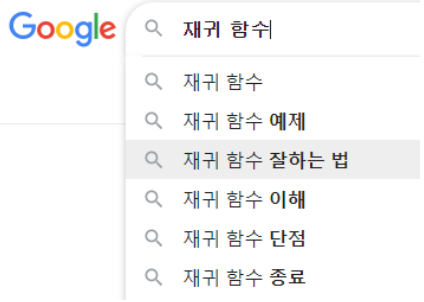
2. 재귀 (Recursion)

재귀란?

- 재귀 = 원래 자리로 되돌아 옴
- C 언어에서는 함수 내부에서 다른 함수를 호출할 수 있다.
- 만약 함수 내부에서 자기 자신을 호출하면?

```c
#include <stdio.h>

int sum(int n){
  if(n == 0) return 0;

  return n + sum(n - 1);
}

int main(){
  printf("%d\n", sum(5));
  return 0;
}
```

재귀 함수?



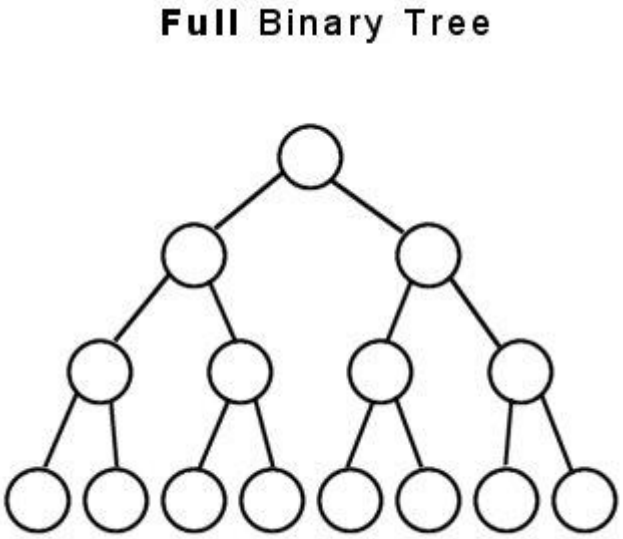
- 알고리즘 수업 시간에서 만나는 첫번째 장벽

- 왜 굳이 배워야 하나요?
  - 분할 정복이 가능한 어떤 문제들을 **간결한 코드**로 해결할 수 있어서.

- 재귀 함수로 구현한 코드를 for나 while루프로 바꿀 수 있다던데?
  - 가능하고, for나 while로 바꾸었을 때 수행 시간이나 메모리 사용 측면에서 더 좋을 때가 많음 (+ 이해하기도 쉬움).

재귀 함수

- 그럼 굳이 왜 배워야 하나요?
  - 많은 자료구조들이 재귀적인 형태를 가지고 있고 문제나 자료구조를 재귀적으로 바라보는 시야를 연습하기 위해



재귀함수의 기본적인 이해

```
void Recursive(void)
{
    printf("Recursive call! \n");
    Recursive( );
}
```

호출

```
void Recursive(void)
{
    printf("Recursive call! \n");
    Recursive( );
}
```

호출

```
void Recursive(void)
{
    printf("Recursive call! \n");
    Recursive( );
}
```

호출

복사본

복사본

복사본

복사본

재귀함수의 이해

원 본

```
void Recursive(void)
{
    printf("Recursive call! \n");
    Recursive( );
}
```

실제로는 코드 복사가
일어나지는 않는다!

```
Recursive call!
Recursive call!
Recursive call!
Recursive call!
Recursive call!
Recursive call!
Recursive call!
Recursive call!
Recursive call!
Recursive call!
Recursive call!
Recursive call!
Recursive call!
Recursive call!
Recursive call!
Recursive call!
Segmentation fault (core dumped)
```

재 진입

```
void Recursive(void)
{
    printf("Recursive call!\n");
    Recursive( );
}
```

재귀함수의 기본적인 이해

```
void Recursive(int num)
{
    if(num <= 0)        // 재귀의 탈출조건
        return;         //재귀의 탈출!
    printf("Recursive call! %d \n", num);
    Recursive(num-1);
}

int main(void)
{
    Recursive(3);
    return 0;
}
```

재귀함수의 간단한 예

실행결과

```
Recursive call! 3
Recursive call! 2
Recursive call! 1
```

재귀함수의 디자인 사례

- n을 받아서 0부터 n까지의 합을 구하는 방법

```c
#include <stdio.h>

int sum(int n){
  if(n == 0) return 0;

  return n + sum(n - 1);
}

int main(){
  printf("%d\n", sum(5));
  return 0;
}
```

재귀함수의 디자인 사례

- n을 받아서 0부터 n까지의 합을 구하는 방법

```c
#include <stdio.h>

int sum(int n){
  if(n == 0) return 0;

  return n + sum(n - 1);
}

int main(){
  printf("%d\n", sum(5));
  return 0;
}
```

sum 을 재귀 호출

재귀함수의 디자인 사례

- n을 받아서 0부터 n까지의 합을 구하는 방법

```c
#include <stdio.h>

int sum(int n){
  if(n == 0) return 0;

  return n + sum(n - 1);
}

int main(){
  printf("%d\n", sum(5));
  return 0;
}
```

종료 조건 (base case).

매 호출마다 n은 1씩 감소.
이 과정에서 base case로 접근 (n=0).

재귀함수의 디자인 사례

- n을 받아서 0부터 n까지의 합을 구하는 방법

```c
#include <stdio.h>

int sum(int n){
  if(n == 0) return 0;

  return n + sum(n - 1);
}

int main(){
  printf("%d\n", sum(5));
  return 0;
}
```

sum(5)

재귀함수의 디자인 사례

- n을 받아서 0부터 n까지의 합을 구하는 방법

```c
#include <stdio.h>

int sum(int n){
  if(n == 0) return 0;

  return n + sum(n - 1);
}

int main(){
  printf("%d\n", sum(5));
  return 0;
}
```

sum(5)
= 5 + sum(4)

재귀함수의 디자인 사례

- n을 받아서 0부터 n까지의 합을 구하는 방법

```c
#include <stdio.h>

int sum(int n){
  if(n == 0) return 0;

  return n + sum(n - 1);
}

int main(){
  printf("%d\n", sum(5));
  return 0;
}
```

sum(5)
= 5 + sum(4)
= 5 + (4 + sum(3))

재귀함수의 디자인 사례

- n을 받아서 0부터 n까지의 합을 구하는 방법

```c
#include <stdio.h>

int sum(int n){
  if(n == 0) return 0;

  return n + sum(n - 1);
}

int main(){
  printf("%d\n", sum(5));
  return 0;
}
```

sum(5)
= 5 + sum(4)
= 5 + (4 + sum(3))
= 5 + (4 + (3 + sum(2)))
= 5 + (4 + (3 + (2 + sum(1))))
= 5 + (4 + (3 + (2 + (1 + sum(0)))))

재귀함수의 디자인 사례

- n을 받아서 0부터 n까지의 합을 구하는 방법

```c
#include <stdio.h>

int sum(int n){
  if(n == 0) return 0;

  return n + sum(n - 1);
}

int main(){
  printf("%d\n", sum(5));
  return 0;
}
```

sum(5)
= 5 + sum(4)
= 5 + (4 + sum(3))
= 5 + (4 + (3 + sum(2)))
= 5 + (4 + (3 + (2 + sum(1))))
= 5 + (4 + (3 + (2 + (1 + sum(0)))))

재귀함수의 디자인 사례

- n을 받아서 0부터 n까지의 합을 구하는 방법

```c
#include <stdio.h>

int sum(int n){
  if(n == 0) return 0;

  return n + sum(n - 1);
}

int main(){
  printf("%d\n", sum(5));
  return 0;
}
```

sum(5)
= 5 + sum(4)
= 5 + (4 + sum(3))
= 5 + (4 + (3 + sum(2)))
= 5 + (4 + (3 + (2 + sum(1))))
= 5 + (4 + (3 + (2 + (1 + sum(0)))))
= 5 + (4 + (3 + (2 + (1 + 0))))

재귀함수의 디자인 사례

- n을 받아서 0부터 n까지의 합을 구하는 방법

```c
#include <stdio.h>

int sum(int n){
  if(n == 0) return 0;

  return n + sum(n - 1);
}

int main(){
  printf("%d\n", sum(5));
  return 0;
}
```

sum(5)
= 5 + sum(4)
= 5 + (4 + sum(3))
= 5 + (4 + (3 + sum(2)))
= 5 + (4 + (3 + (2 + sum(1))))
= 5 + (4 + (3 + (2 + (1 + sum(0)))))
= 5 + (4 + (3 + (2 + (1 + 0)))))

재귀함수의 디자인 사례

- n을 받아서 0부터 n까지의 합을 구하는 방법

```c
#include <stdio.h>

int sum(int n){
  if(n == 0) return 0;

  return n + sum(n - 1);
}

int main(){
  printf("%d\n", sum(5));
  return 0;
}
```

sum(5)
= 5 + sum(4)
= 5 + (4 + sum(3))
= 5 + (4 + (3 + sum(2)))
= 5 + (4 + (3 + (2 + sum(1))))
= 5 + (4 + (3 + (2 + (1 + sum(0)))))
= 5 + (4 + (3 + (2 + (1 + 0))))
= 5 + (4 + (3 + (2 + 1)))

재귀함수의 디자인 사례

- n을 받아서 0부터 n까지의 합을 구하는 방법

```c
#include <stdio.h>

int sum(int n){
  if(n == 0) return 0;

  return n + sum(n - 1);
}

int main(){
  printf("%d\n", sum(5));
  return 0;
}
```

sum(5)
= 5 + sum(4)
= 5 + (4 + sum(3))
= 5 + (4 + (3 + sum(2)))
= 5 + (4 + (3 + (2 + sum(1))))
= 5 + (4 + (3 + (2 + (1 + sum(0)))))
= 5 + (4 + (3 + (2 + (1 + 0))))
= 5 + (4 + (3 + (2 + 1)))
= 5 + (4 + (3 + 3))
= 5 + (4 + 6)
= 5 + 10
= 15

재귀의 특징

- 재귀는 큰 문제를 몇 개의 **작은 문제들**로 나누어 해결 가능
  - sum(5)를 5와 sum(4)를 더하는 것으로 **분할**
  - sum(4)는 4와 sum(3)을 더하는 것으로 **분할**
  - **...**
  - sum(1)을 1과 sum(0)을 더하는 것으로 **분할**
  - sum(0)은 쉬운 문제. 0을 돌려줌.

- 나뉘어진 **작은 문제들**은 원래의 문제와 정확히 같은 형태를 지님

- 반복문을 활용하는 방법과 항상 쌍으로 존재
  - 재귀를 활용하면 프로그래밍이 간단해지지만, 연산 시간이 비효율적으로 되는 경우가 많음 (개선 가능)

재귀함수의 디자인 사례

$$n! = n \times (n\text{-}1) \times (n\text{-}2) \times (n\text{-}3) \times \ldots \times 2 \times 1$$

$$(n\text{-}1)!$$

$$n! = n \times (n\text{-}1)!$$

$$f(n) = \begin{cases} n \times f(n-1) & \ldots. \; n \geq 1 \\ 1 & \ldots. \; n = 0 \end{cases}$$

```
if(n == 0)
    return 1;
```

```
else
    return n * Factorial(n-1);
```

팩토리얼의 재귀적 구현

```
int Factorial(int n)
{
    if(n == 0)
        return 1;
    else
        return n * Factorial(n-1);
}

int main(void)
{
    printf("1! = %d \n", Factorial(1));
    printf("2! = %d \n", Factorial(2));
    printf("3! = %d \n", Factorial(3));
    printf("4! = %d \n", Factorial(4));
    printf("9! = %d \n", Factorial(9));
    return 0;
}
```

팩토리얼 구현 결과

실행결과

```
1! = 1
2! = 2
3! = 6
4! = 24
9! = 362880
```

피보나치 수열 1: 이해

피보나치 수열

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 . . . .

피보나치 수열의 구성

수열의 n번째 값 = 수열의 n-1번째 값 + 수열의 n-2번째 값

피보나치 수열의 표현

$$fib(n) = \begin{cases} 0 & \ldots . n=1 \\ 1 & \ldots . n=2 \\ fib(n-1) + fib(n-2) & \ldots . \text{otherwise} \end{cases}$$

피보나치 수열 2: 코드의 구현

$$fib(n) = \begin{cases} 0 & \ldots\ldots n=1 \\ 1 & \ldots\ldots n=2 \\ fib(n-1)+fib(n-2) & \ldots\ldots \text{otherwise} \end{cases}$$

```
int Fibo(int n)
{
    if(n == 1)
        return 0;
    else if(n == 2)
        return 1;
    else
        return Fibo(n-1) + Fibo(n-2);
}
```

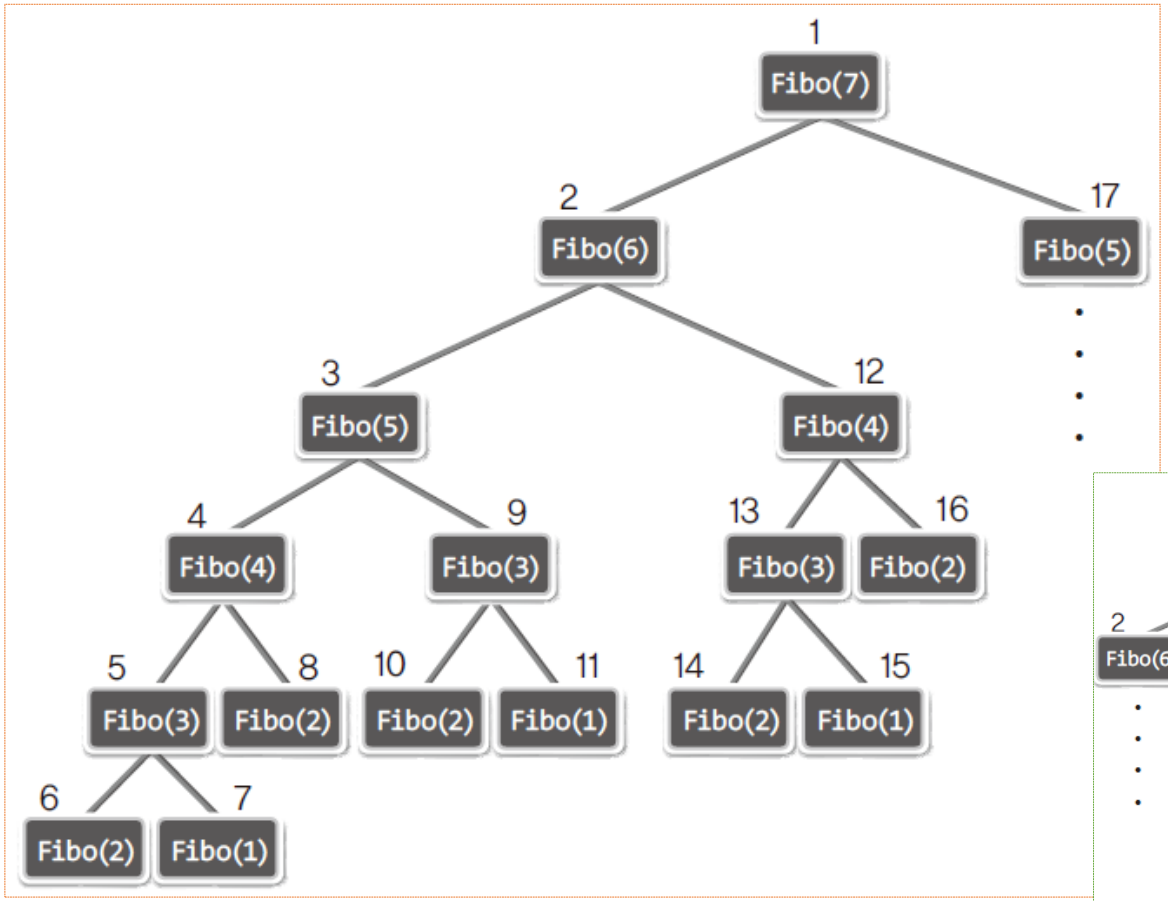
피보나치 수열 3: 완성된 예제

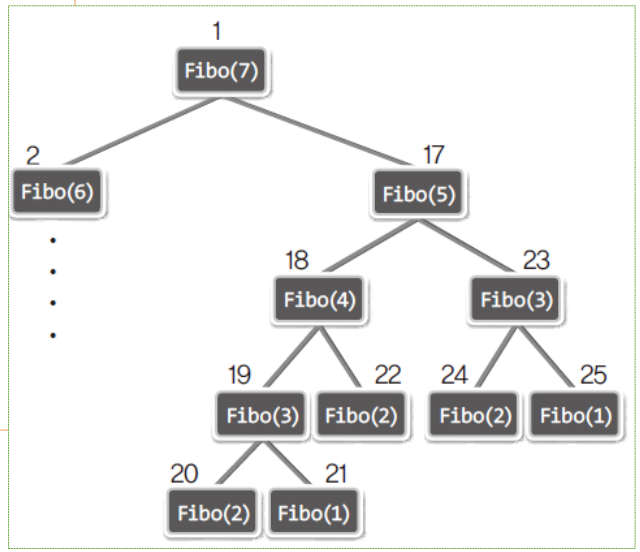
```
int Fibo(int n)
{
    if(n == 1)
        return 0;
    else if(n == 2)
        return 1;
    else
        return Fibo(n-1) + Fibo(n-2);
}

int main(void)
{
    int i;
    for(i=1; i<15; i++)
        printf("%d ", Fibo(i));

    return 0;
}
```

실행결과

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233
```

피보나치 수열 4: 함수의 흐름





이진 탐색 알고리즘의 구현 (반복문)

```
int BSearch(int ar[], int len, int target)
{
	int first = 0;		// 탐색 대상의 시작 인덱스 값
	int last = len-1;	// 탐색 대상의 마지막 인덱스 값
	int mid;

	while(first <= last)
	{
		mid = (first+last) / 2;		// 탐색 대상의 중앙을 찾는다.
		if(target == ar[mid])		// 중앙에 저장된 것이 타겟이라면
		{
			return mid;			// 탐색 완료!
		}
		else	// 타겟이 아니라면 탐색 대상을 반으로 줄인다.
		{
			if(target < ar[mid])
				last = mid-1;	// 왜 -1을 하였을까?
			else
				first = mid+1;	// 왜 +1을 하였을까?
		}
	}
	return -1;		// 찾지 못했을 때 반환되는 값 -1
}
```

이진 탐색 알고리즘의 재귀구현 1

이진 탐색의 알고리즘의 핵심

1. 탐색 범위의 중앙에 목표 값이 저장되었는지 확인
2. 저장되지 않았다면 탐색 범위를 반으로 줄여서 다시 탐색 시작

이진 탐색의 종료에 대한 논의

```
int BSearchRecur(int ar[], int first, int last, int target)
{
    if(first > last)
        return -1;     // -1의 반환은 탐색의 실패를 의미
    . . . .
}
```

first와 last는 각각 탐색의 시작과 끝을 나타내는 변수

이진 탐색 알고리즘의 재귀구현 2

탐색 대상의 확인!

```
int BSearchRecur(int ar[], int first, int last, int target)
{
    int mid;
    if(first > last)
        return -1;

    mid = (first+last) / 2;
    if(ar[mid] == target)
        return mid;
    . . . .
}
```

탐색의 대상에서 중심에 해당하는 인덱스 값 계산

타겟이 맞는지 확인!

이진 탐색 알고리즘의 재귀구현 3

계속되는 탐색

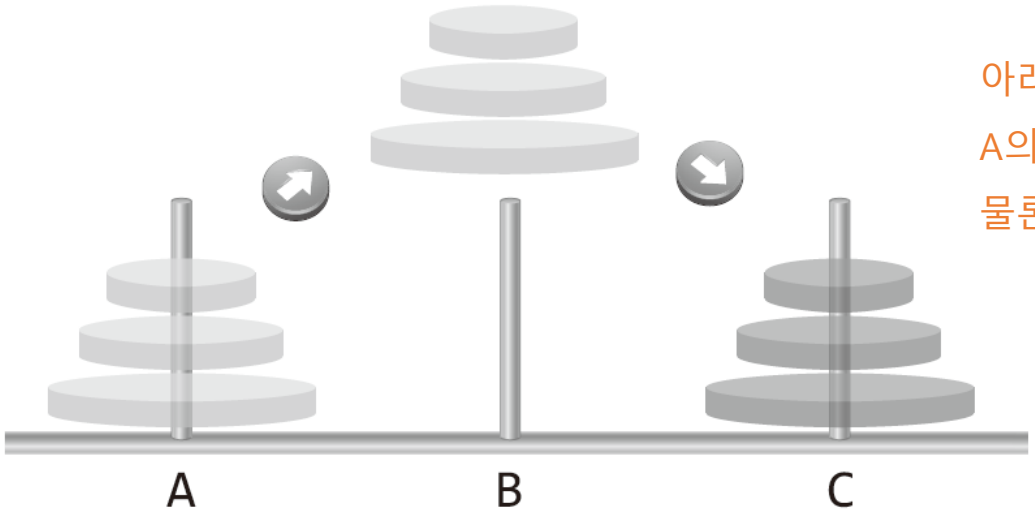
```
int BSearchRecur(int ar[], int first, int last, int target)
{
    int mid;
    if(first > last)
        return -1;
    mid = (first+last) / 2;

    if(ar[mid] == target)
        return mid;
    else if(target < ar[mid])
        return BSearchRecur(ar, first, mid-1, target);
    else
        return BSearchRecur(ar, mid+1, last, target);
}
```

앞부분을 대상으로 재 탐색

뒷부분을 대상으로 재 탐색

하노이 타워 문제의 이해

원반을 A에서 C로 이동하기

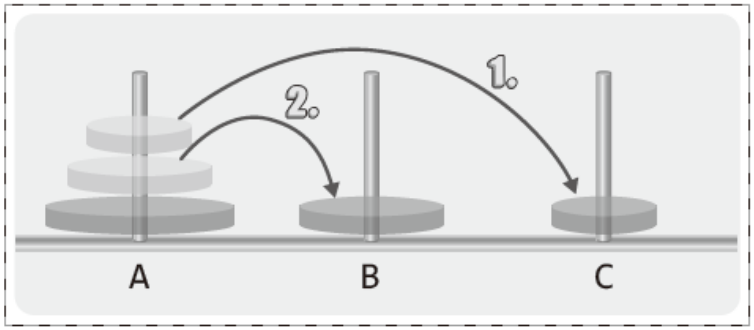


아래의 제약사항을 만족하면서
A의 모든 원반을 C로 옮기는 문제!
물론 옮기는 과정에서 B를 활용한다.

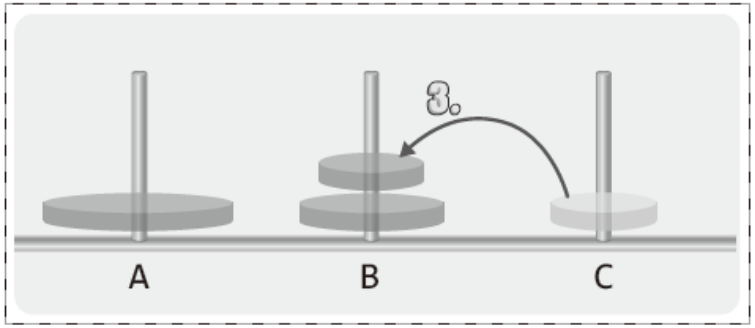
제약사항

- 원반은 한 번에 하나씩만 옮길 수 있습니다.
- 옮기는 과정에서 작은 원반의 위에 큰 원반이 올려져서는 안됩니다.

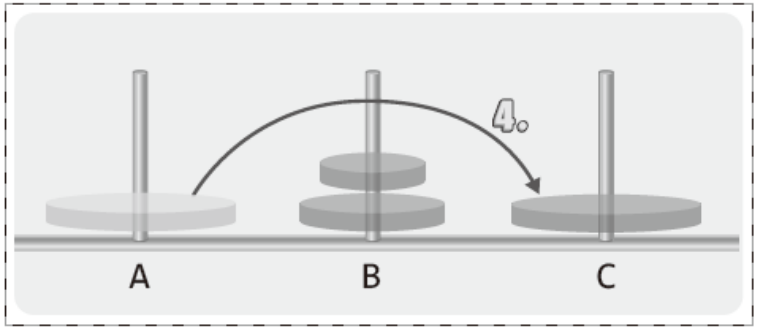
하노이 타워 문제 해결의 예 1



▶ [그림 02-8: 문제해결 1/5]

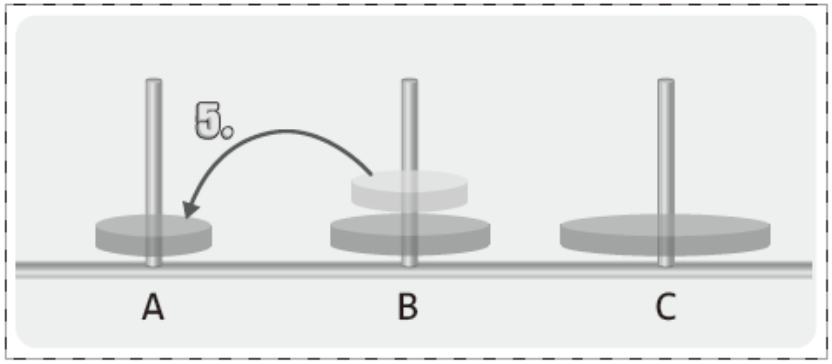


▶ [그림 02-9: 문제해결 2/5]

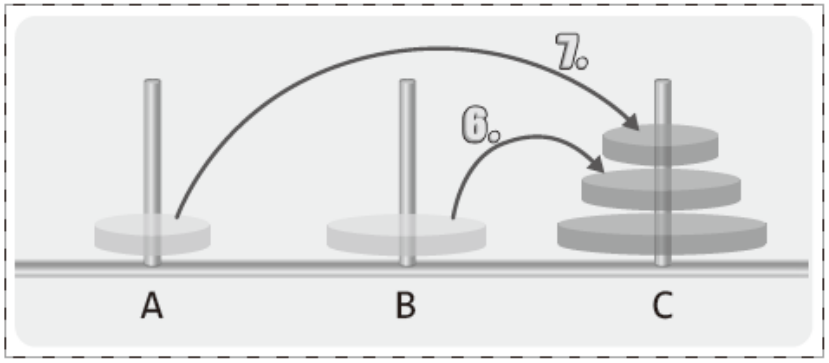


▶ [그림 02-10: 문제해결 3/5]

하노이 타워 문제 해결의 예 2



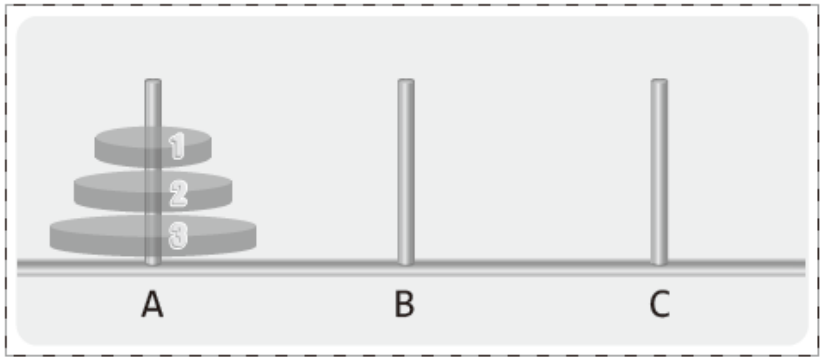
▶ [그림 02-11: 문제해결 4/5]



▶ [그림 02-12: 문제해결 5/5]

지금 보인 과정에서 반복의 패턴을 찾아야 문제의 해결을 위한 코드를 작성할 수 있다!.

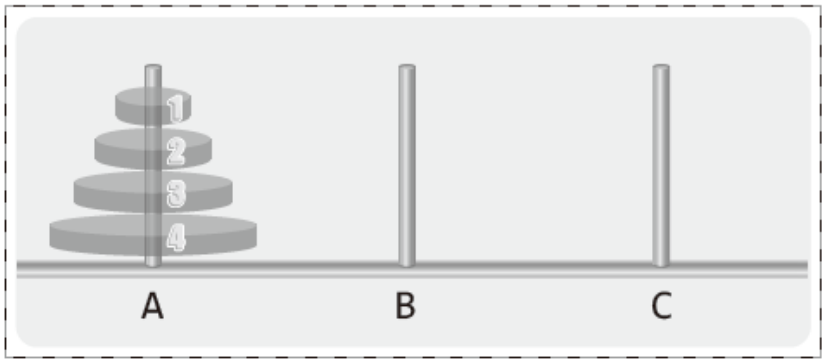
반복되는 일련의 과정을 찾기 위한 힌트



A의 세 원반을 C로 옮기기 위해서는 원반 3을 C로 옮겨야 한다. 그리고 이를 위해서는 원반 1과 2를 우선 원반 B로 옮겨야 한다.

▶ [그림 02-13: 원반이 3개인 하노이 타워]

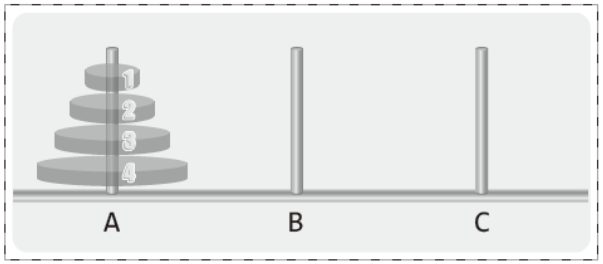
위와 아래의 두 예를 통해서 문제의 해결에 있어서 반복이 되는 패턴이 있음을 알 수 있다.



A의 네 원반을 C로 옮기기 위해서는 원반 4를 C로 옮겨야 한다. 그리고 이를 위해서는 원반 1과 2와 3을 우선 원반 B로 옮겨야 한다.

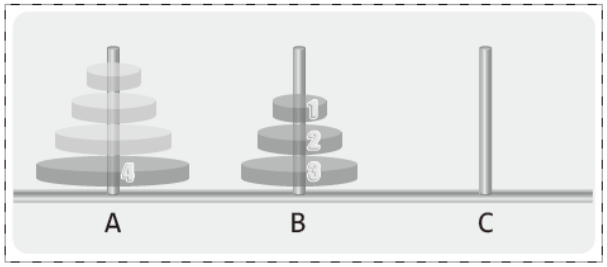
▶ [그림 02-14: 원반이 4개인 하노이 타워]

하노이 타워의 반복패턴 연구 1



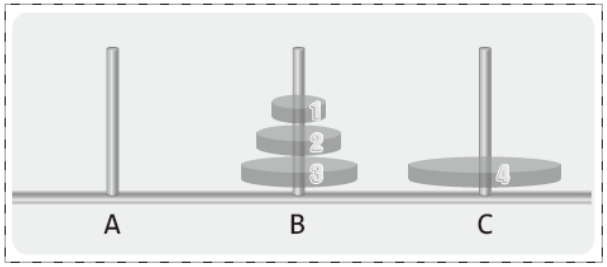
목적. 원반 4개를 A에서 C로 이동

▶ [그림 02-14: 원반이 4개인 하노이 타워]



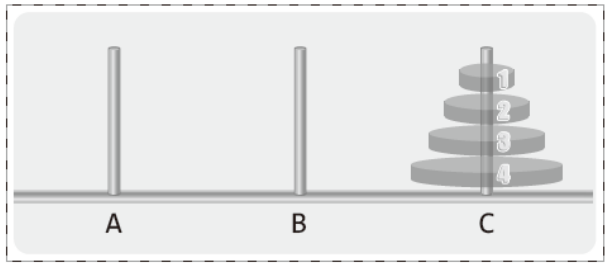
1. 작은 원반 3개를 A에서 B로 이동

▶ [그림 02-15: 반복패턴 1/3]



2. 큰 원반 1개를 A에서 C로 이동

▶ [그림 02-16: 반복패턴 2/3]



3. 작은 원반 3개를 B에서 C로 이동

▶ [그림 02-17: 반복패턴 3/3]

하노이 타워의 반복패턴 연구 2

목적. 원반 4개를 A에서 C로 이동

목적. 큰 원반 n개를 A에서 C로 이동

---

1. 작은 원반 3개를 A에서 B로 이동

1. 작은 원반 n-1개를 A에서 B로 이동

---

2. 큰 원반 1개를 A에서 C로 이동

2. 큰 원반 1개를 A에서 C로 이동

---

3. 작은 원반 3개를 B에서 C로 이동

3. 작은 원반 n-1개를 B에서 C로 이동

하노이 타워 문제의 해결 1

하노이 타워 함수의 기본 골격

```
void HanoiTowerMove(int num, char from, char by, char to)
{
    . . . .
}
```

원반 num의 수에 해당하는 원반을 from에서 to로 이동을 시키되 그 과정에서 by를 활용한다.

목적. 큰 원반 n개를 A에서 C로 이동 — HanoiTowerMove(num, from, by, to);

1. 작은 원반 n-1개를 A에서 B로 이동 — HanoiTowerMove(num-1, from, to, by);

2. 큰 원반 1개를 A에서 C로 이동 — printf(. . . .);

3. 작은 원반 n-1개를 B에서 C로 이동 — HanoiTowerMove(num-1, by, from, to);

하노이 타워 문제의 해결 2

목적. 큰 원반 n개를 A에서 C로 이동 — HanoiTowerMove(num, from, by, to);

1. 작은 원반 n-1개를 A에서 B로 이동 — HanoiTowerMove(num-1, from, to, by);

2. 큰 원반 1개를 A에서 C로 이동 — printf(. . . .);

3. 작은 원반 n-1개를 B에서 C로 이동 — HanoiTowerMove(num-1, by, from, to);

```
void HanoiTowerMove(int num, char from, char by, char to)
{
    if(num == 1)        // 이동할 원반의 수가 1개라면
    {
        printf("원반1을 %c에서 %c로 이동 \n", from, to);
    }
    else
    {
        HanoiTowerMove(num-1, from, to, by);
        printf("원반%d을(를) %c에서 %c로 이동 \n", num, from, to);
        HanoiTowerMove(num-1, by, from, to);
    }
}
```

하노이 타워 문제의 해결 3

```
void HanoiTowerMove(int num, char from, char by, char to)
{
    if(num == 1)        // 이동할 원반의 수가 1개라면
    {
        printf("원반1을 %c에서 %c로 이동 \n", from, to);
    }
    else
    {
        HanoiTowerMove(num-1, from, to, by);
        printf("원반%d을(를) %c에서 %c로 이동 \n", num, from, to);
        HanoiTowerMove(num-1, by, from, to);
    }
}

int main(void)
{
    // 막대A의 원반 3개를 막대B를 경유하여 막대C로 옮기기
    HanoiTowerMove(3, 'A', 'B', 'C');
    return 0;
}
```

실행결과

```
원반1을 A에서 C로 이동
원반2을(를) A에서 B로 이동
원반1을 C에서 B로 이동
원반3을(를) A에서 C로 이동
원반1을 B에서 A로 이동
원반2을(를) B에서 C로 이동
원반1을 A에서 C로 이동
```

재귀의 문제점 (1)

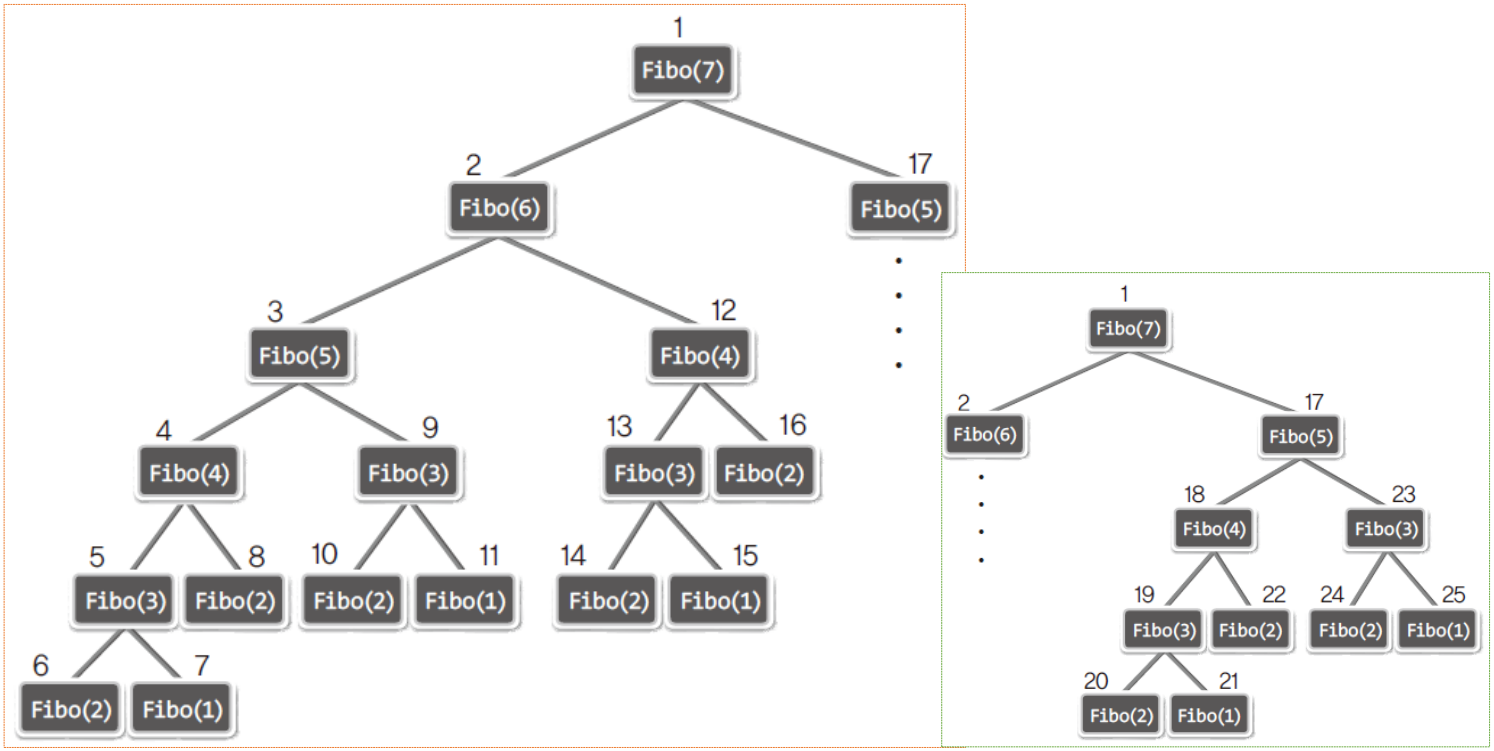
- 함수 호출로 인한 부하

```c
#include <stdio.h>

int sum(int n){
  if(n == 0) return 0;

  return n + sum(n - 1);
}

int main(){
  printf("%d\n", sum(5));
  return 0;
}
```

sum(5)
= 5 + sum(4)
= 5 + (4 + sum(3))
= 5 + (4 + (3 + sum(2)))
= 5 + (4 + (3 + (2 + sum(1))))
= 5 + (4 + (3 + (2 + (1 + sum(0)))))
= 5 + (4 + (3 + (2 + (1 + 0))))
= 5 + (4 + (3 + (2 + 1)))
= 5 + (4 + (3 + 3))
= 5 + (4 + 6)
= 5 + 10
= 15

재귀의 문제점 (2)

- 재귀를 이용한 해결 방법에 내재된 비효율성
  - 같은 계산의 반복



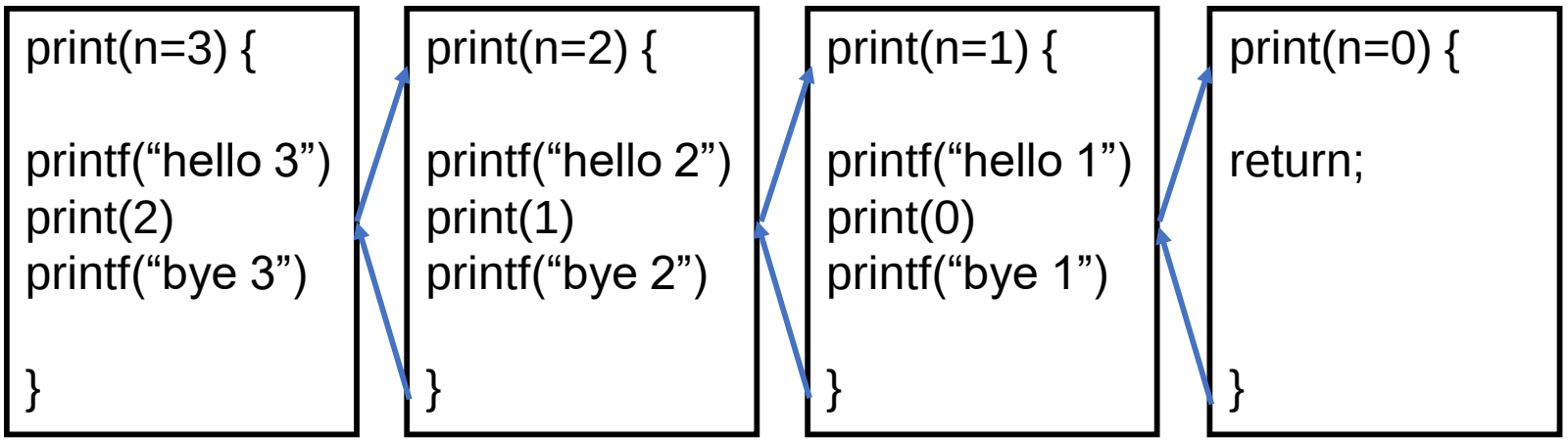
예제 문제

- 아래 코드의 수행 결과는?

```c
#include <stdio.h>

void print(int n) {
    if(n == 0) return;

    printf("hello %d\n",n);
    print(n - 1);
    printf("bye %d\n", n);
}

int main(){
    print(3);

    return 0;
}
```

예제 문제

- 아래 코드의 수행 결과는?




```c
#include <stdio.h>

void print(int n) {
    if(n == 0) return;

    printf("hello %d\n",n);
    print(n - 1);
    printf("bye %d\n", n);
}

int main(){
    print(3);

    return 0;
}
```

```
hello 3
hello 2
hello 1
bye 1
bye 2
bye 3
```

예제 문제

- 다음 코드의 차이점은?

```c
#include <stdio.h>

int sum(int n) {
    if(n == 0) return 0;

    return n + sum(n - 1);
}

int newsum(int n) {
    if(n <= 0) return 0;

    return n + newsum(n - 1);
}
```

요약

- 재귀 호출은 어렵다.

- 분할 정복: 문제 속에 존재하는 재귀성을 찾고
  1) 큰 문제를 작은 문제로 분할
  2) 이를 다시 더 작은 문제로 분할
  3) 반복하여 답을 알고있는 쉬운 문제를 푼다.

- 분할 정복이 가능한 문제를 재귀 호출을 활용하여 간결하게 프로그래밍할 수 있다.
  - 팩토리얼, 피보나치 수열, 이분 탐색, 하노이 탑 등