알고리즘 성능 분석, 재귀문

Data Structures and Algorithms

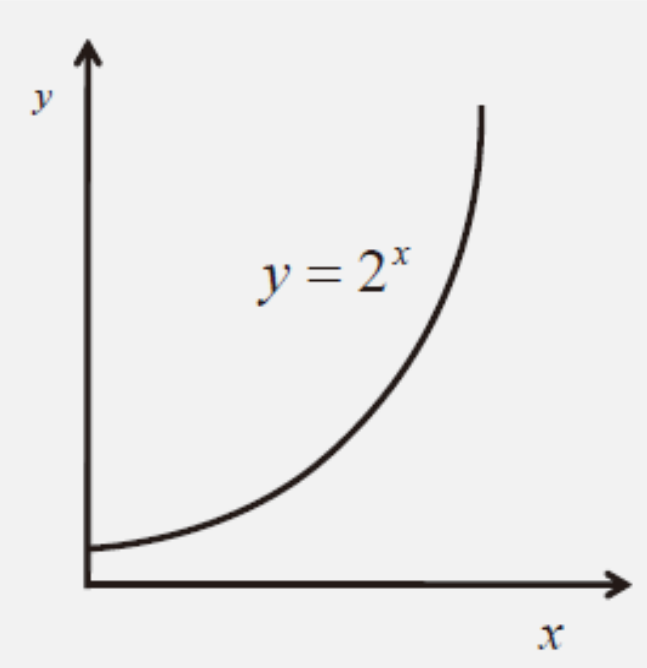
목차

- 알고리즘 분석
- 복잡도 계산: 이진 탐색 알고리즘
- 시간 복잡도의 일반화

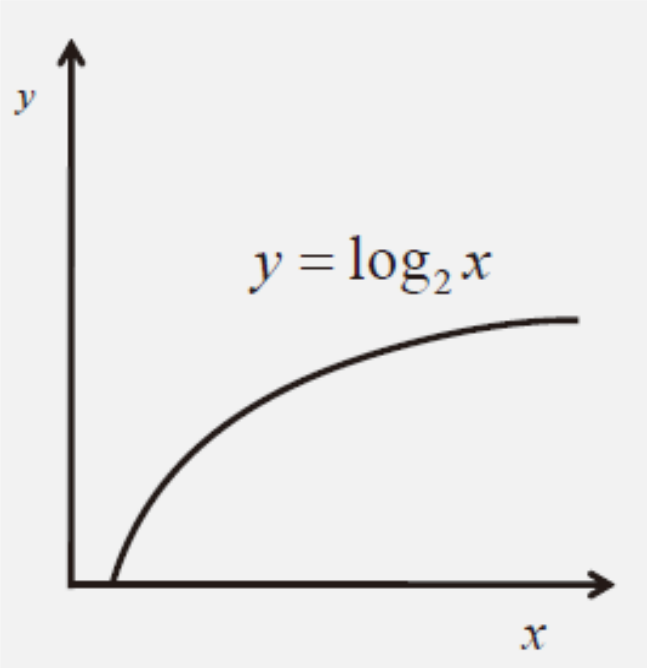
알고리즘의 성능 분석

배경지식

- x 축: 데이터 수
- y 축: 연산 횟수



지수식



로그식

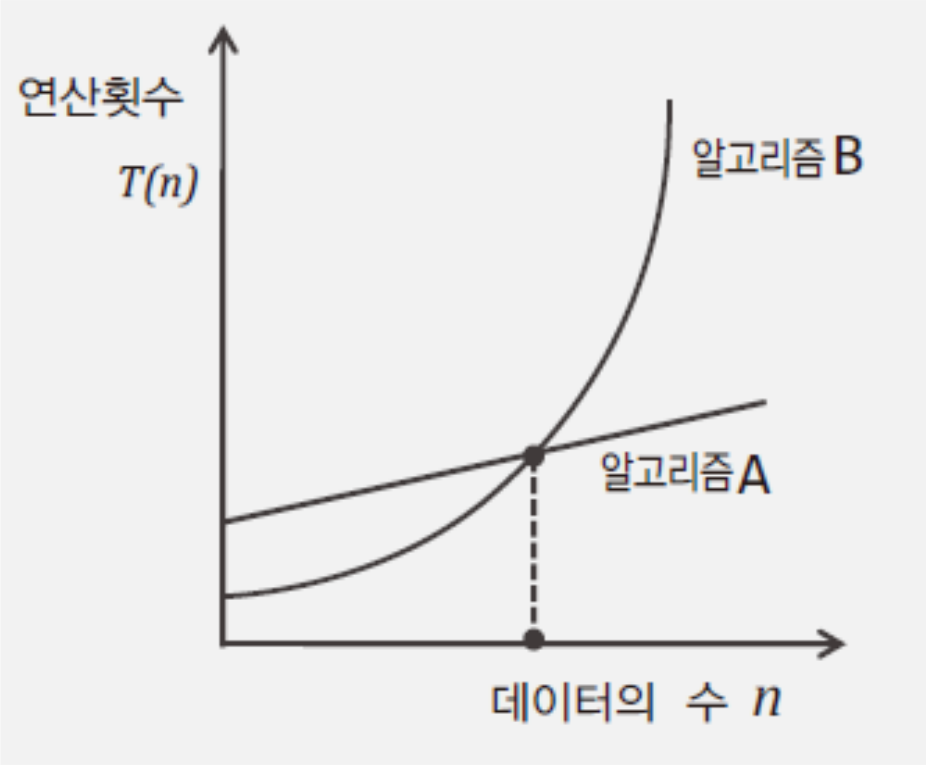
복잡도 Complexity

- 알고리즘을 평가하는 두 가지 요소
  - 시간 복잡도(time complexity) ➔ 얼마나 빠른가?
  - 공간 복잡도(space complexity) ➔ 얼마나 메모리를 적게 쓰는가?

- 시간 복잡도의 평가 방법
  - 중심이 되는 특정 연산의 횟수를 평가
  - 데이터의 수에 대한 연산횟수의 함수 T(n)을 구함

알고리즘의 수행 속도 비교 기준

- 데이터의 수가 적은 경우의 수행 속도는 큰 의미가 없음
- 데이터의 수에 따른 수행 속도의 변화 정도가 기준



Which algorithm is better?

순차 탐색 알고리즘의 시간 복잡도

- 핵심 연산은?

```
// 순차 탐색 알고리즘 적용된 함수
int LSearch(int ar[], int len, int target)
{
	int i;
	for(i=0; i<len; i++)
	{
		if(ar[i] == target)
			return i;		// 찾은 대상의 인덱스 값 반환
	}
	return -1;		// 찾지 못했음을 의미하는 값 반환
}
```

최상과 최악의 경우

- 운이 좋은 경우
  - 배열의 맨 앞에서 대상을 찾는 경우
  - ‘최상의 경우(best case)’는 관심 없음

- 운이 없는 경우
  - 배열의 끝에서 찾거나 대상이 저장되지 않은 경우
  - ‘최악의 경우(worst case)’에 관심 있음

평균적인 경우(Average Case)

- 가장 현실적인 경우
  - 일반적 상황에 대한 경우의 수
  - 알고리즘 평가에 도움이 됨
  - 계산이 어려움. 객관적 평가가 쉽지 않음

- 평균적인 경우의 복잡도 계산이 어려운 이유
  - 연출이 어려움
  - 증명이 어려움
  - 상황마다 다름 (최악의 상황의 결과는 늘 동일함)

순차 탐색 최악의 경우 시간 복잡도

데이터의 수가 *n*개일 때,

최악의 경우

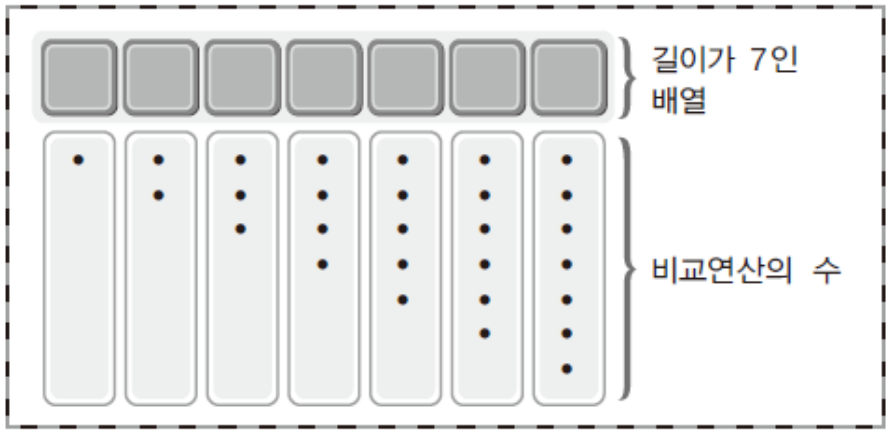
연산횟수는(비교연산의 횟수는) *n*

$T(n) = n$

최악의 경우를 대상으로 정의한 함수 $T(n)$

순차 탐색 평균적 경우 시간 복잡도

- 가정
  1. 탐색 대상이 배열에 존재 확률 50%
  2. 배열 요소에 존재 확률 동일
- 연산 횟수
  - 탐색 대상이 존재하지 않는 경우의 연산횟수는 n
  - 가정 2에 의해서 탐색 대상이 존재하는 경우의 연산횟수는 n/2



$$T(n) = n \times \frac{1}{2} + \frac{n}{2} \times \frac{1}{2} = \frac{3}{4}n$$

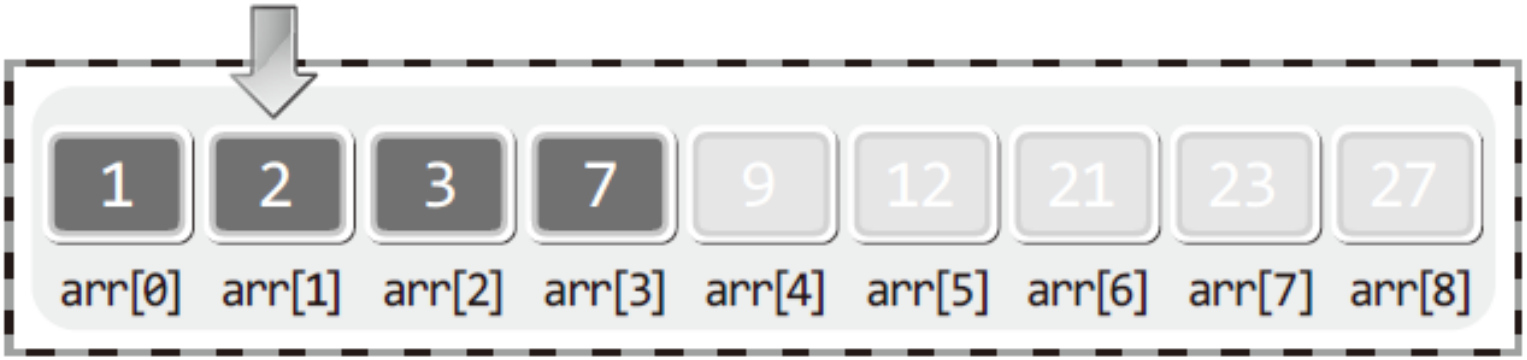
복잡도 계산

이진 탐색 알고리즘

이진 탐색 알고리즘: 3 찾기

- 첫 번째 시도
  1. 배열 인덱스의 시작은 0, 끝 8
  2. 0과 8의 합을 2로 나눔
  3. 나눈 결과 4를 인덱스로 하여 저장된 값이 3인지 확인

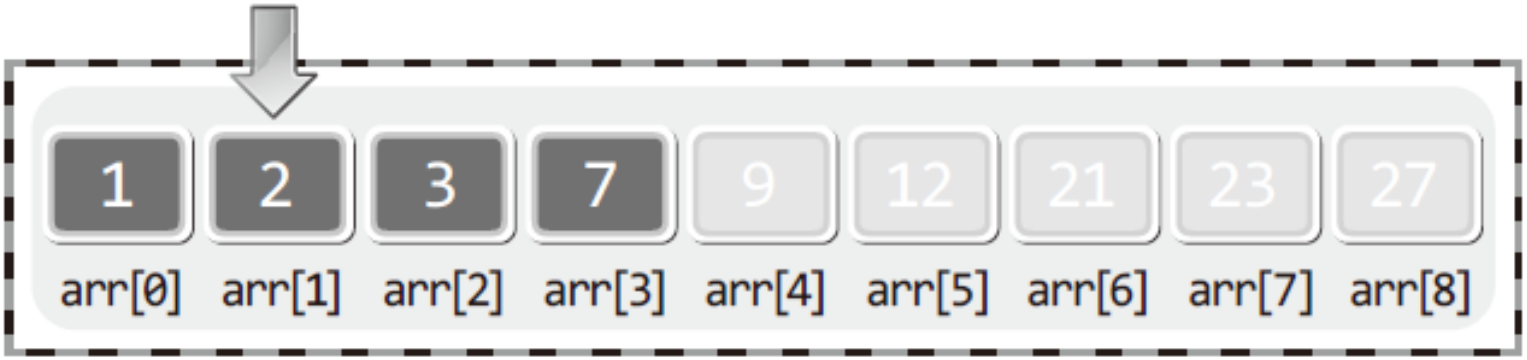
- 순차 탐색보다 좋은 성능
- 단, 정렬 필수



이진 탐색 알고리즘: 3 찾기

- 두 번째 시도:
  1. arr[4]의 9와 3의 대소 비교
  2. 대소 비교결과, arr[4]>3이므로 탐색 범위를 인덱스 0~3으로 제한!
  3. 첫 시도와 같이 0과 3의 합을 2로 나눔. 나머지는 버림
  4. 나눈 결과 1을 인덱스로 하여 저장된 값이 3인지 확인

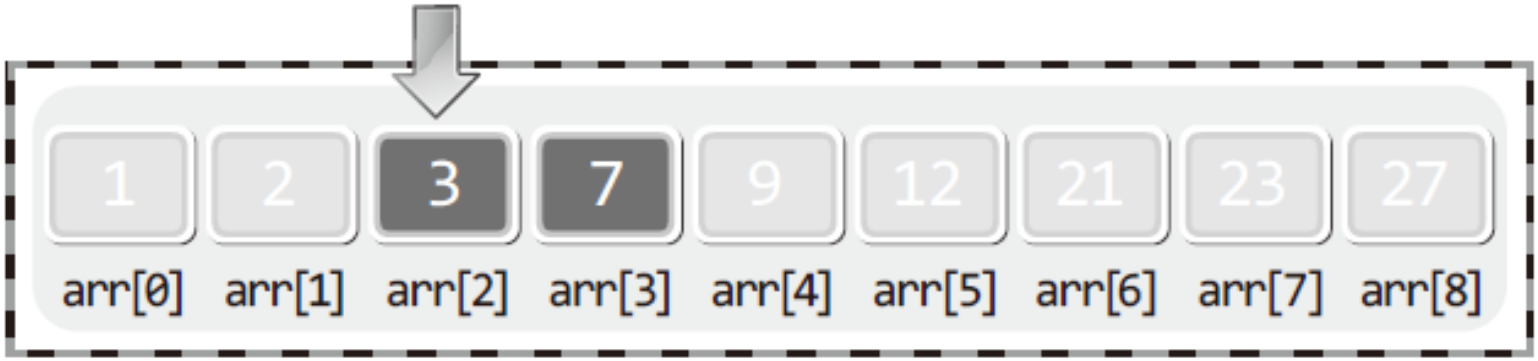
- 탐색의 대상, 첫 번째 시도 이후의 반



이진 탐색 알고리즘: 3 찾기

- 세 번째 시도:
  1. arr[1]의 2와 3의 대소 비교
  2. 대소 비교결과, arr[1]<3이므로 탐색 범위를 인덱스 2~3으로 제한!
  3. 2와 3의 합을 2로 나눔. 나머지는 버림
  4. 나눈 결과 2을 인덱스로 하여 저장된 값이 3인지 확인

- 탐색의 대상, 두 번째 시도 이후의 반

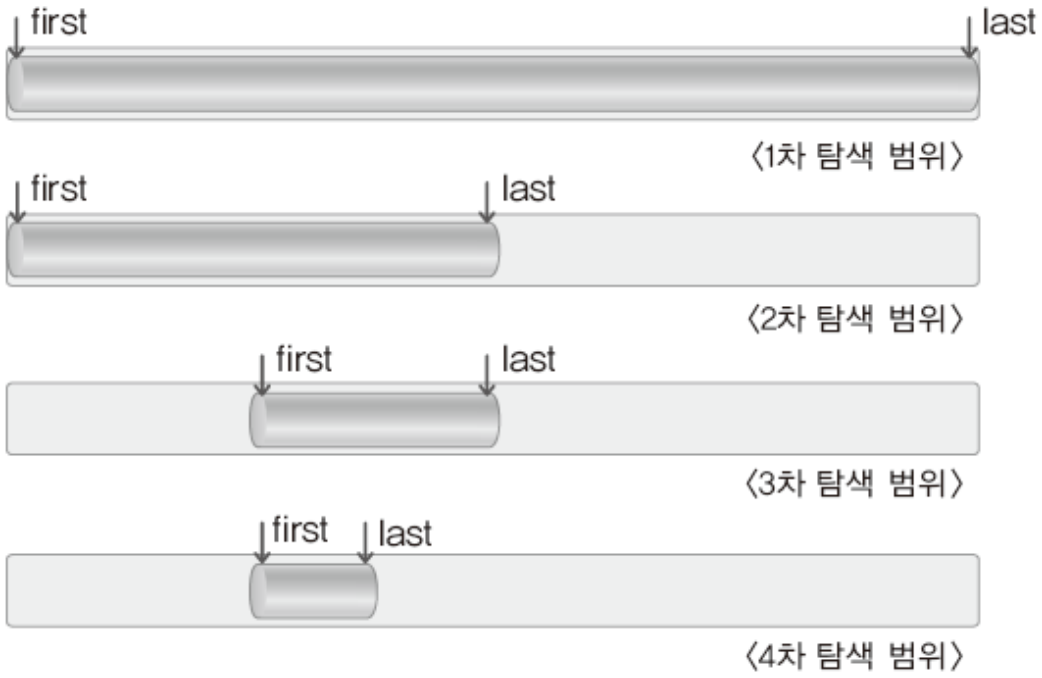


이진 탐색 알고리즘: 3 찾기

- 핵심: 과정마다 비교 대상의 범위가 ½씩 줄어듬

구현

- first와 last가 같다면
  - 탐색 대상이 하나란 뜻
- first와 last가
  역전될때까지 반복



- 기본 골격

```
while(first <= last)
{
    // 이진 탐색 알고리즘의 진행
}
```

구현

```
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;       // 중앙 인덱스
        if(target == ar[mid])       // 중앙에 저장된 것이 타겟이라면
        {
            return mid; // 탐색 완료!
        }
        else     // 타겟이 아니라면 탐색 대상을 반으로 줄임
        {
            if(target < ar[mid])
                last = mid-1;
            else
                first = mid+1;
        }
    }
    return -1;       // 찾지 못했을 때 반환되는 값 -1
}
```

최악의 경우 시간 복잡도

- 시간 복잡도 계산을 위한 핵심 연산은?

```
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;       // 중앙 인덱스
        if(target == ar[mid])       // 중앙에 저장된 것이 타겟이라면
        {
            return mid; // 탐색 완료!
        }
        else     // 타겟이 아니라면 탐색 대상을 반으로 줄임
        {
            if(target < ar[mid])
                last = mid-1;
            else
                first = mid+1;
        }
    }
    return -1;       // 찾지 못했을 때 반환되는 값 -1
}
```

?

최악의 경우 시간 복잡도

- 데이터의 수가 n개일 때, 최악의 경우 비교연산의 횟수는?
  - 데이터 수가 n개일 때의 탐색과정에서 1회의 비교연산 진행
  - 데이터 수가 n/2개일 때의 탐색과정에서 1회 비교연산 진행
  - 데이터 수가 n/4개일 때의 탐색과정에서 1회 비교연산 진행
  - 데이터 수가 n/8개일 때의 탐색과정에서 1회 비교연산 진행

  ⋮ n 이 정해지지 않음; 비교 횟수 확정 불가

  - 데이터 수가 1개일 때의 탐색과정에서 1회의 비교연산 진행

객관적 성능 비교 불가!

일반화 필요

시간 복잡도의 일반화

시간 복잡도의 일반화

- 비교 연산의 횟수의 예
  - 데이터 수가 8에서 1이 되기 까지 3번 비교하고 2로 나눔 (3회 비교)
  - 데이터가 1개 남았을 때, 마지막으로 비교연산 1회 진행 (1회 비교)

- 일반화 과정
  - n이 1이 되기까지 2로 나눈 횟수 k회, 비교연산 k회
  - 데이터가 1개 남았을 때, 마지막으로 비교연산 1회 진행

- 최악의 경우 시간 복잡도 함수 T(n)=k+1

시간 복잡도의 일반화

- n을 몇번 ½로 나눠야 1이 되는가?

$$n \times \left(\frac{1}{2}\right)^k = 1$$
$$n \times 2^{-k} = 1$$
$$n = 2^k$$
$$\log_2 n = \log_2 2^k$$
$$\log_2 n = k \cdot \log_2 2$$
$$\log_2 n = k$$

함수 T(n)= k+1 의 일반화

$$T(n) = k + 1$$
$$T(n) = \log_2 n + 1$$
$$T(n) = \log_2 n$$

Where did the +1 go?

Answer: Big Oh Notation

빅-오 표기법(Big-Oh Notation)

- 빅-오(Big-Oh): *T(n)*에서 가장 변화량이 큰 수식만 평가함

1 $T(n) = n^2 + 2n + 1$

2 $T(n) = n^2 + 2n$ 1차 근사치 식

3 $T(n) = n^2$ 2차 근사치 식

4 $O(n^2)$ T(n)의 빅-오

n이 증가함에 따라 2n+1 영향력 미미해짐

n	n^2	$2n$	$T(n)$	n^2의 비율
10	100	20	120	83.33%
100	10,000	200	10,200	98.04%
1,000	1,000,000	2,000	1,002,000	99.80%
10,000	100,000,000	20,000	100,020,000	99.98%
100,000	10,000,000,000	200,000	10,000,200,000	99.99%

빅-오 단순 계산법

- T(n)이 다항식인 경우, 최고차항의 차수가 빅-오

- 예

$T(n) = -n^2 + 2n + 9$?

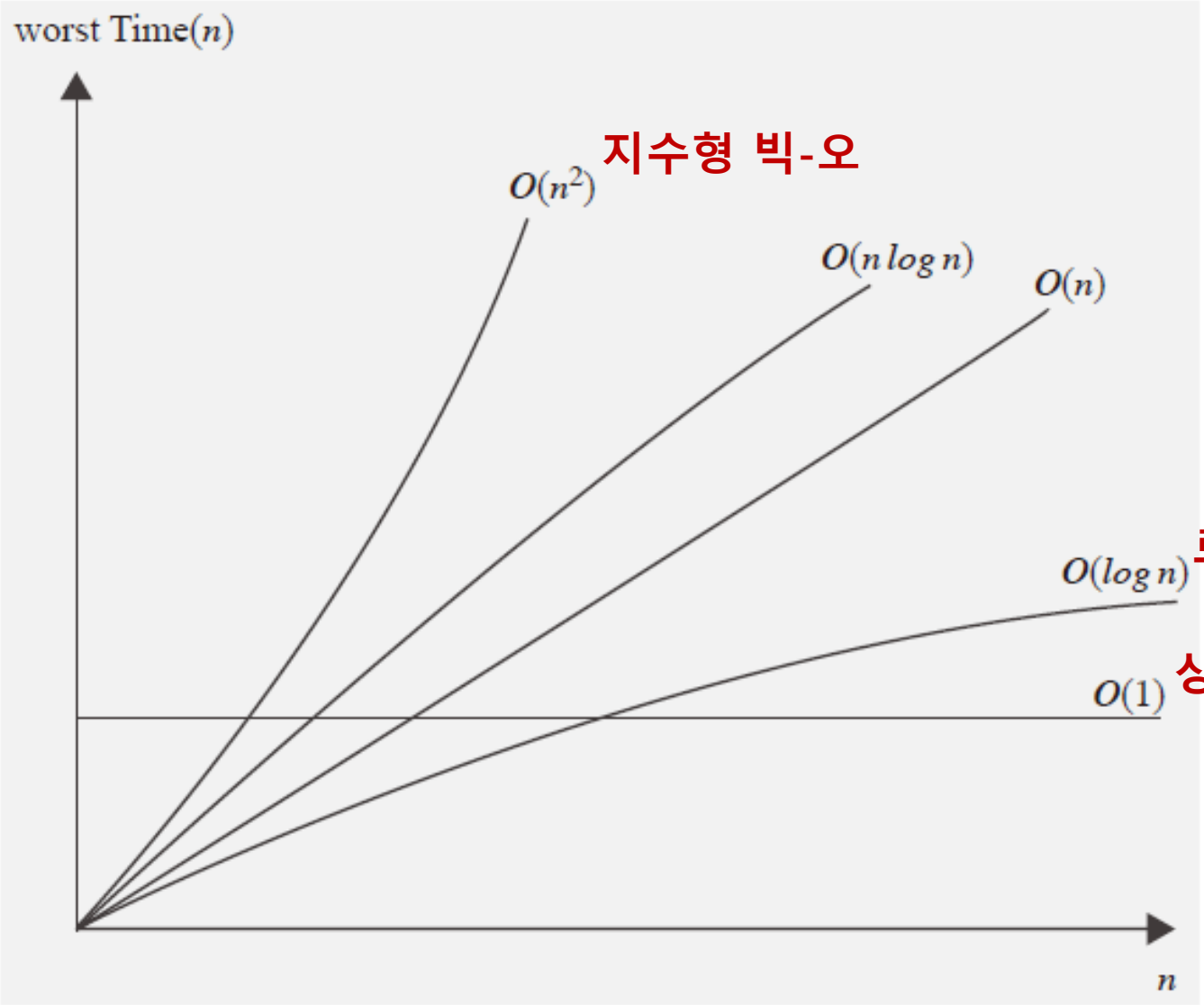
$T(n) = n^4 + n^3 + n^2 + 1$?

$T(n) = 5n^3 + 3n^2 + 2n + 1$?

- 일반화

$T(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n^1 + a_0$?

대표적 빅-오



순차 탐색 vs. 이진 탐색 알고리즘

알고리즘	복잡도	빅-오
순차 탐색	$T(n)$	?
이진 탐색	$T(n) = \log_2 n + 1$	?

최악의 경우에 연산의 횟수 BSWorstOpCount.c에서 결과 확인

n	순차 탐색 연산횟수	이진 탐색 연산횟수
500	500	9
5,000	5,000	13
50,000	50,000	16

순차 탐색 알고리즘 vs. 이진 탐색 알고리즘

- $T(n)=n$ ▶ $O(n)$

순차 탐색의 최악의 경우 시간 복잡도 **순차 탐색의 빅-오**

- $T(n)=log_2 n+1$ ▶ $O(log\, n)$

이진 탐색의 최악의 경우 시간 복잡도 **이진 탐색의 빅-오**

n	순차 탐색 연산횟수	이진 탐색 연산횟수
500	500	9
5,000	5,000	13
50,000	50,000	16

최악의 경우에 진행이 되는 연산의 횟수

이진 탐색의 연산횟수는 예제 BSWorstOpCount.c에서 확인한 결과

실용적 의미

- 수학적 정의
  - 두 개의 함수 f(n)과 g(n)이 주어졌을 때, 모든 n≥K에 대하여 f(n)≤Cg(n)을 만족하는 두 개의 상수 C와 K가 존재하면, f(n)의 빅-오는 O(g(n))임

$5n^2 + 100$ 의 빅오 $O(n^2)$ 의 **실용적 의미**

$5n^2 + 100$ 의 증가율은 아무리 커도 n^2

빅-오 판별의 예

$$Let\ \ f(n) = 5n^2 + 100,\ \ g(n) = n^2$$

$$\forall n \geqq K\ \ (e.g.\, K = 12)$$

$$5n^2 + 100 \leqq C \cdot n^2, \forall\, C > 6$$

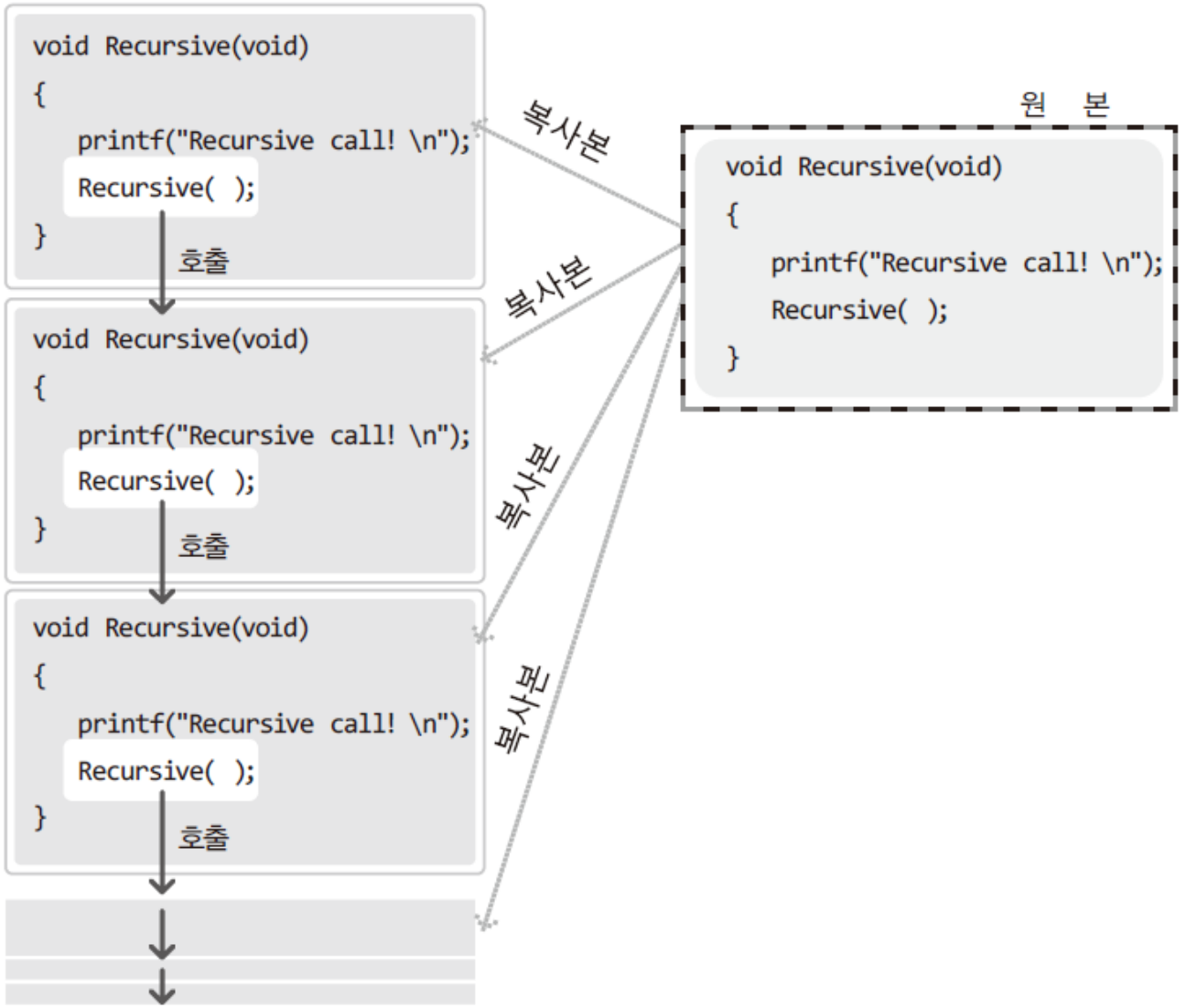
f(n) ≤ Cg(n)을 만족하는 두 개의 상수 C와 K가 존재한다면,

$$f(n) = O(g(n))$$

$$\therefore\ \ 5n^2 + 100 = O(n^2)$$

함수의 재귀적 호출의 이해

재귀함수의 도식화

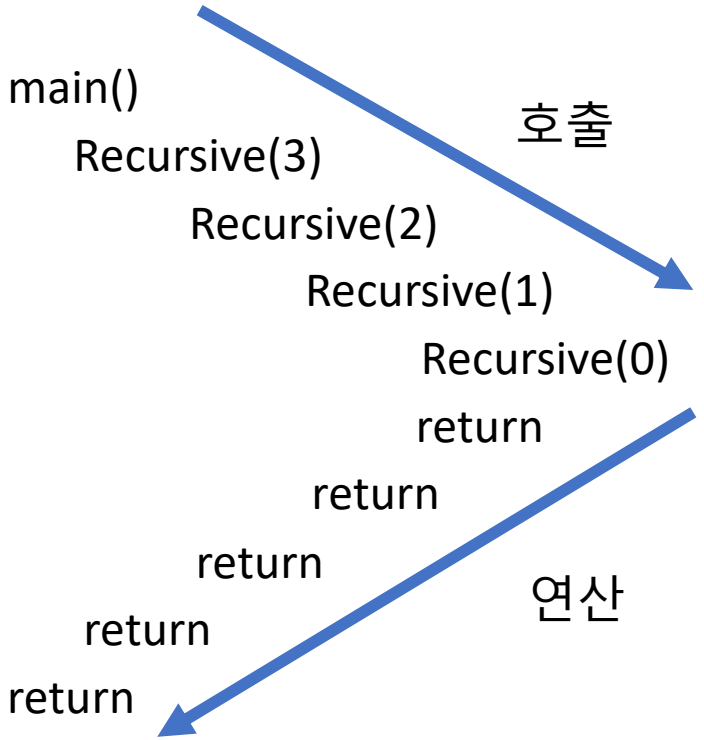

void Recursive(void)
{
printf("Recursive call! \n");
Recursive( );
}
호출
void Recursive(void)
{
printf("Recursive call! \n");
Recursive( );
}
호출
void Recursive(void)
{
printf("Recursive call! \n");
Recursive( );
}
호출
복사본
복사본
복사본
복사본
원 본
void Recursive(void)
{
printf("Recursive call! \n");
Recursive( );
}

재귀함수

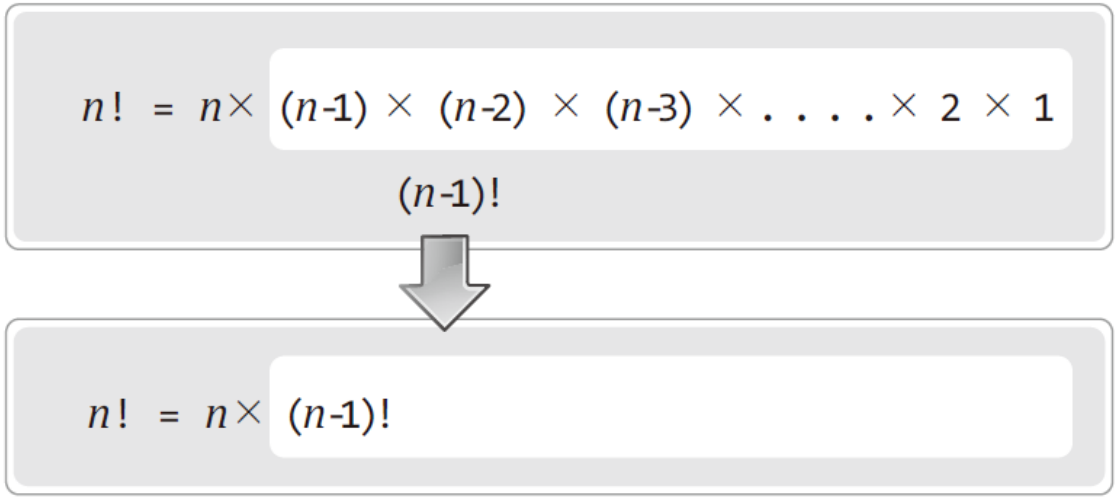
- 자기 자신을 다시 호출하는 함수

```
#include <stdio.h>
void Recursive(int num){
    if(num <= 0) // 재귀의 종료 조건
        return;
    printf("Recursive Call! %d \n", num);
    Recursive(num-1);
}

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



디자인 사례: 팩토리얼



$$f(n) = \begin{cases} n \cdot f(n-1) & n \geqq 1 \\ 1 & n = 0 \end{cases}$$

```
if(n == 0)
    return 1;
else
    return n * f (n-1);
```

팩토리얼의 재귀적 구현

```
#include <stdio.h>

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

int main()
{
    printf( " 1! = %d\n " , factorial(1));
    printf( " 1! = %d\n " , factorial(3));
    printf( " 1! = %d\n " , factorial(5));
    printf( " 1! = %d\n " , factorial(9));
    return 0;
}
```

```
1! = 1
3! = 6
5! = 120
9! = 362880
```

재귀의 활용

Fibonacci

피보나치

A series of numbers, starting from 0 where every number is the sum of the two numbers preceding it.

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

Named after

FIBONACCI

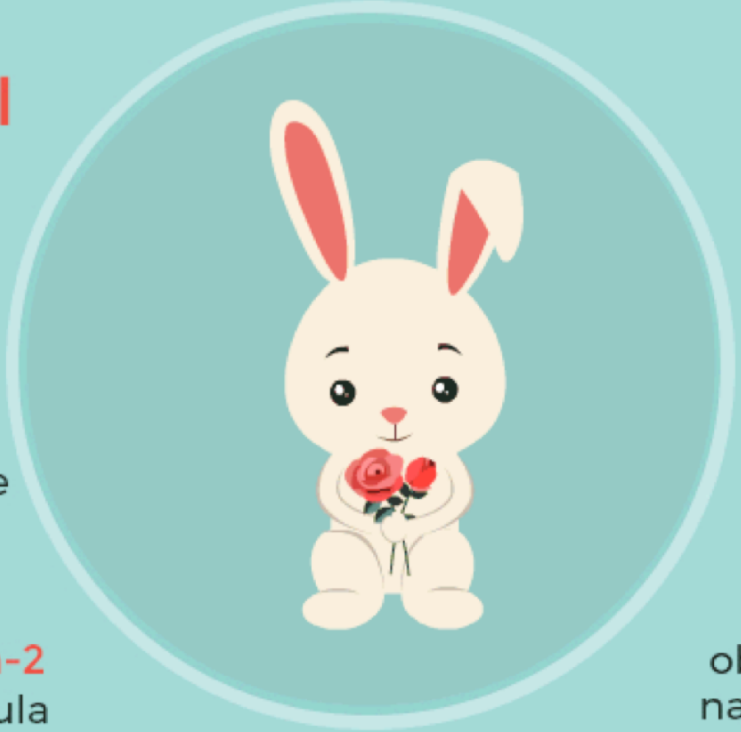
An Italian mathematician

Year 1202

The year it was first introduced to the western world in the book "Liber Abaci"

$x_n = x_{n-1} + x_{n-2}$

Mathematical formula



1.618

"Phi" or the "Golden Ratio" The ratio of any two consequent numbers of the sequence

Nature's code

Because it is observed in several natural phenomena

피보나치 수열 → 코드

- 수열: $0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ \ldots$
- 패턴: $n^{th}\text{수열} = (n-1)^{th}\text{ 수열} + (n-2)^{th}\text{ 수열}$
- 조건:

$$f(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ fib(n-1) + fib(n-2) & else \end{cases}$$

- 코드:

```
int fib(int n)
{
    if(   ?   )
        return    ?
    else if(   ?   )
        return    ?
    else
        return          ?
}
```

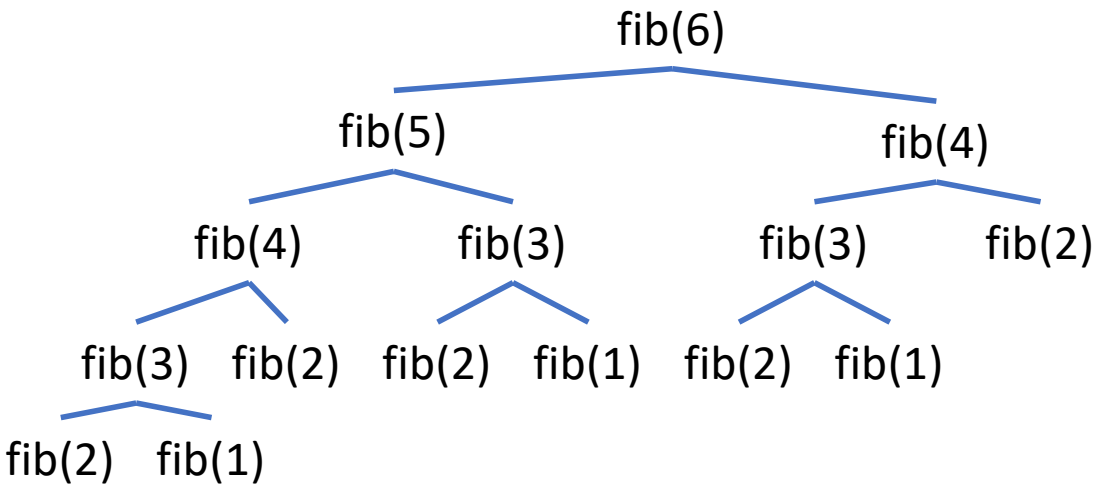
전체 코드와 흐름

```
#include <stdio.h>

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

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

    return 0;
}
```



지면 관계상 fib(6)부터 예시

재귀의 활용

이진 탐색

이진 탐색 알고리즘

- 이진 탐색의 알고리즘의 핵심
  1. 탐색 범위의 중앙에 목표 값이 저장되었는지 확인
  2. 저장되지 않았다면 탐색 범위를 반으로 줄임
  3. 1로 되돌아 감

이진 탐색 알고리즘의 재귀적 표현

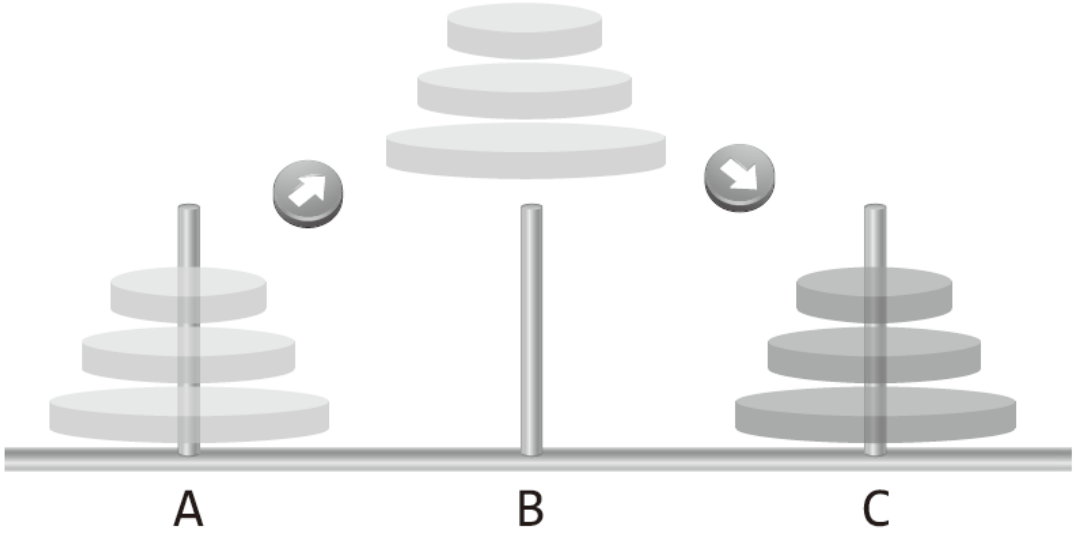
```
int BsearchRecur(int arr[], int first, int last, int tartget)
{
    if(first > last)  // first 탐색 시작, last 탐색 끝
        return -1;    // 탐색 실패

    mid = (first + last) / 2;  // 시작과 끝 사이의 중앙 인덱스 계산
    if(arr[mid] == target)     // 타겟 확인
        return mid;
    else if(target < arr[mid])
        // 앞부분을 대상으로 재 탐색
        return BsearchRecur(int arr[],   ?  ,   ?  , target);
    else
        // 뒷부분을 대상으로 재 탐색
        return BsearchRecur(int arr[],   ?  ,  ?  , target);
}
```

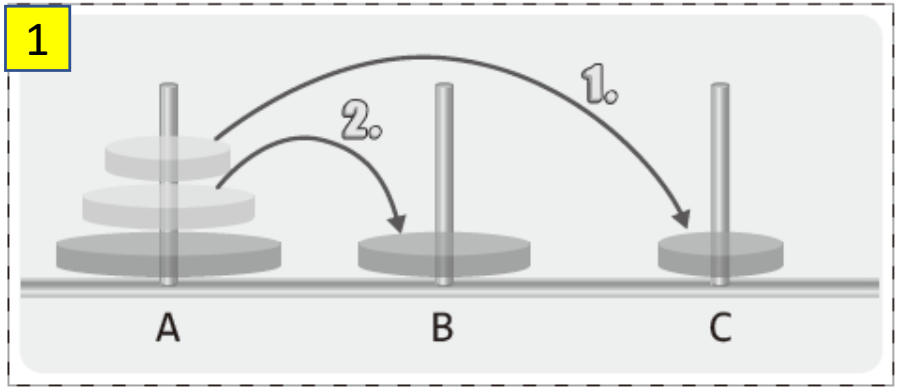
하노이 타워

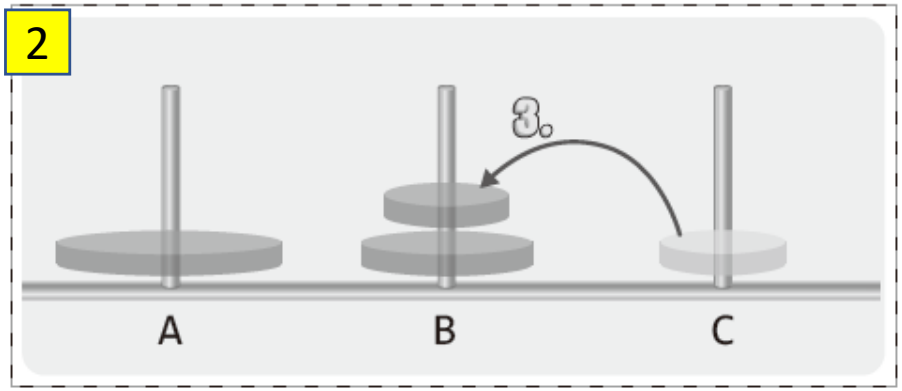
문제의 이해

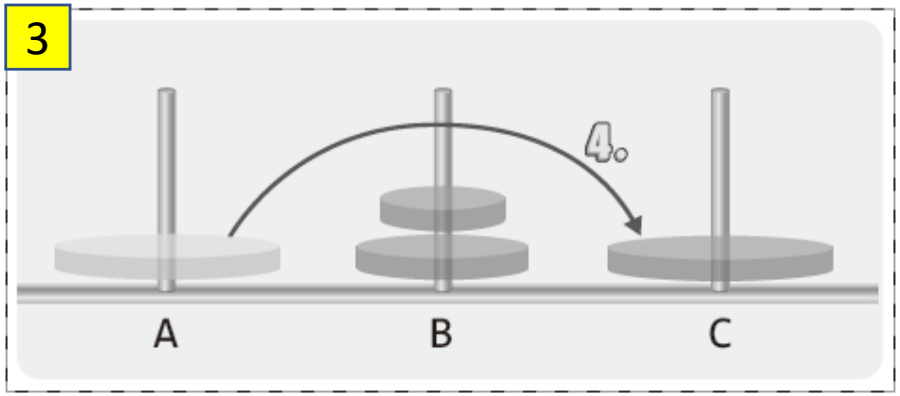
- 제약 사항
  - 원반은 한 번에 하나씩만 옮길 수 있음
  - 옮기는 과정에서 작은 원반의 위에 큰 원반을 올릴 수 없음
- 문제
  - 위의 제약 사항을 지키면서 원반을 A에서 C로 이동하기

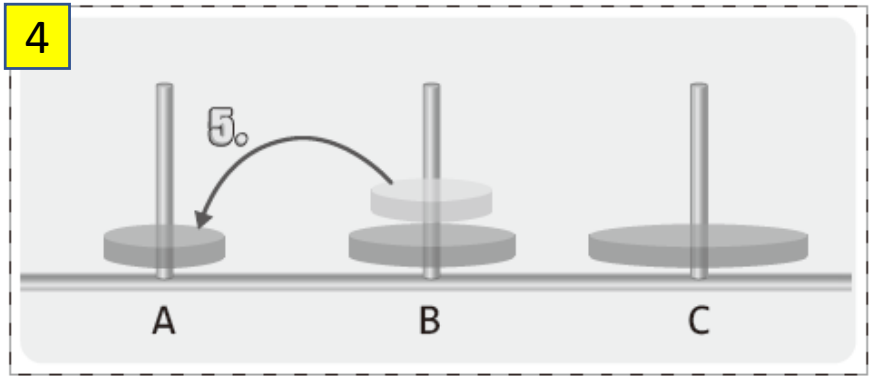


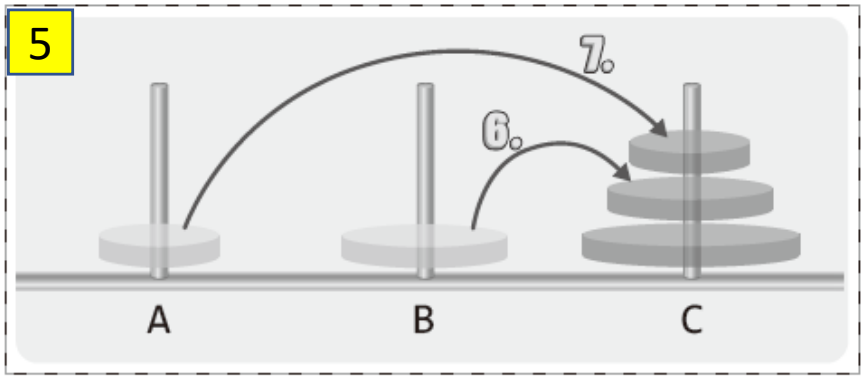
하노이 타워 해결 도식화





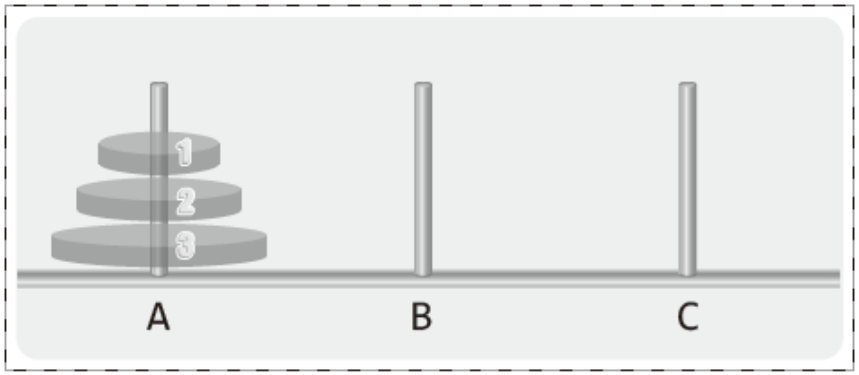






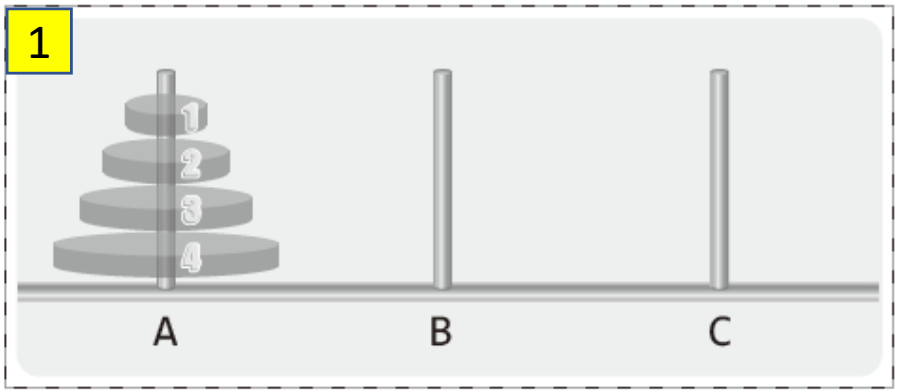
반복 패턴

- 원반 3개



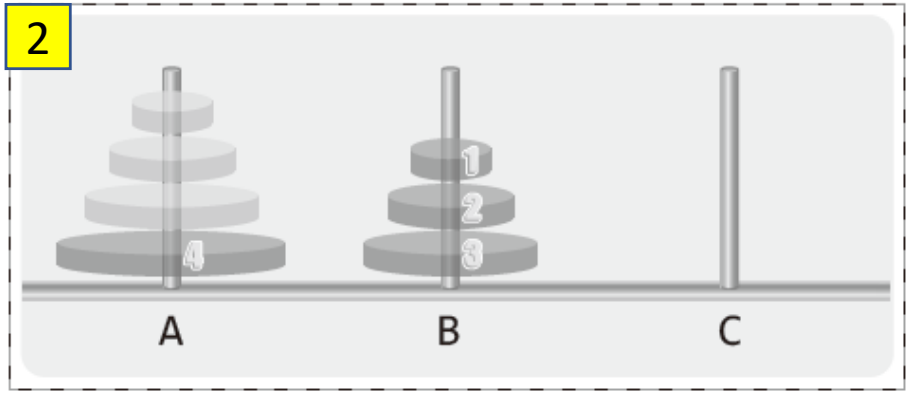
  - 원반 3을 C로 옮겨야 함
  - 그 전에 ?

반복 패턴



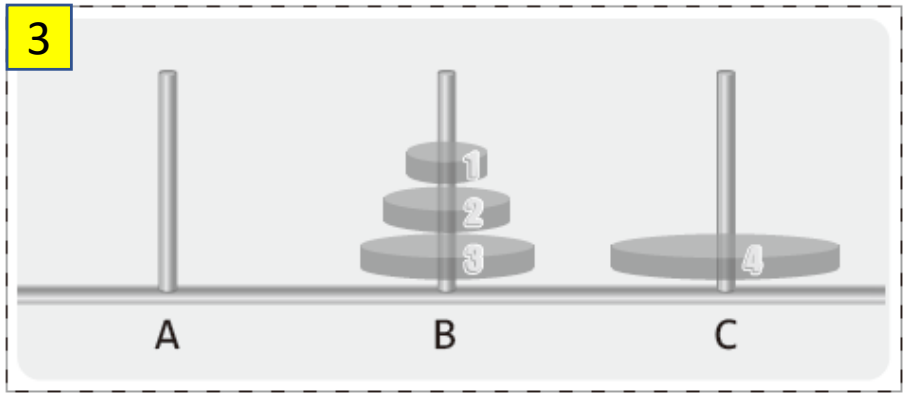
목적. 원반 4개를 A에서 C로 이동

Step 1



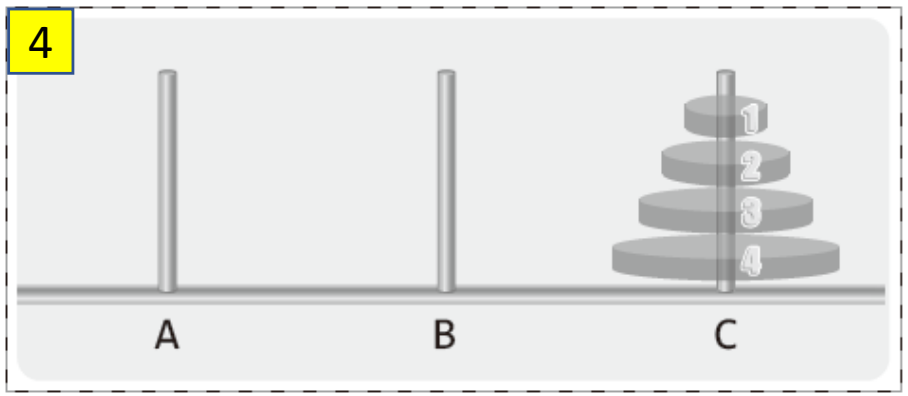
1. 작은 원반 3개를 A에서 B로 이동

Step 2



2. 큰 원반 1개를 A에서 C로 이동

Step 3



3. 작은 원반 3개를 B에서 C로 이동

Step 4

하노이 타워 함수의 기본 골격

```
// num: 원반 갯수, from 원반 시작 위치, by 경유 위치, to 최종 위치
void HanoiTowerMove(int num, char from, char by, char to)
{
      . . . .
}
```

목적. 큰 원반 n개를 A에서 C로 이동 `HanoiTowerMove(num,  ?  );`

1. 작은 원반 n-1개를 A에서 B로 이동 `HanoiTowerMove(num-1,  ?  );`

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

3. 작은 원반 n-1개를 B에서 C로 이동 `HanoiTowerMove(num-1,  ?  );`

코드

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)
        printf("원반 1을 %c에서 %c로 이동\n", from, to);
    else
    {
        HanoiTowerMove(num-1, from, to, by);
        printf("원반 %d을 %c에서 %c로 이동\n", num, from, to);
        HanoiTowerMove(num-1, by, from, to);
    }
}
```

코드

```
#include <stdio.h>

void HanoiTowerMove(int num, char from, char by, char to)
{
    if(num == 1)
        printf("원반 1을 %c에서 %c로 이동\n", from, to);
    else
    {
        HanoiTowerMove(num-1, from, to, by);
        printf("원반 %d을 %c에서 %c로 이동\n", num, from, to);
        HanoiTowerMove(num-1, by, from, to);
    }
}

int main()
{
    HanoiTowerMove(3, 'A', 'B', 'C');
    return 0;
}
```