2주차 3차시 알고리즘의 설계와 분석 II

【학습목표】

1. 시간 복잡도와 공간 복잡도를 이해할 수 있다.
2. 시간 복잡도의 점근적 표기법을 이해할 수 있다.
3. 재귀 알고리즘을 이해할 수 있다.

학습내용1 : 계산 복잡도와 표기법

직접 구현하지 않고서도 알고리즘의 효율성을 따져보는 기법으로 알고리즘의 복잡도 분석이 있다. 이 분석에는 두 가지 방법이 있는데 알고리즘의 수행 시간을 분석하는 시간 복잡도(time complexity)와 알고리즘이 사용하는 기억공간을 분석하는 공간 복잡도(space complexity)가 있다. 시간복잡도는 입력크기에 대한 함수로 표기하는데, 이 함수는 주로 여러개의 항을 가지는 다항식이다. 그래서 이를 단순한 함수로 표현하기 위해 점근적 표기를 사용한다. 이는 입력의 크기가 n이 무한대로 커질 때의 복잡도를 간단히 하기 위해서 사용하는 표기법으로 빅-오 표기법, 오메가 표기법, 세타 표기법이 있다.

1부터 100까지 더하는 데 필요한 알고리즘을 생각해보면
- 간단히 1부터 100까지 더하는 방법 : 1+2+3+4+5+6+……+100 = 5050
- n(n+1)/2 라는 수열에 대입 해보면 : (100+1)*50 = 5050

와 같이 계산할 수 있음을 알 수 있다.

1. 알고리즘의 성능 분석

예: 1부터 100까지 합을 구하는 문제

100번의 연산	3번의 연산
1+2+3……+100 = 5050	100(100+1)/2
100번의 연산(덧셈 100번)	3법의 연산 (덧셈1번, 곱셈1번, 나눗셈 1번)

학습내용2 : 시간복잡도의 점근적 표기법

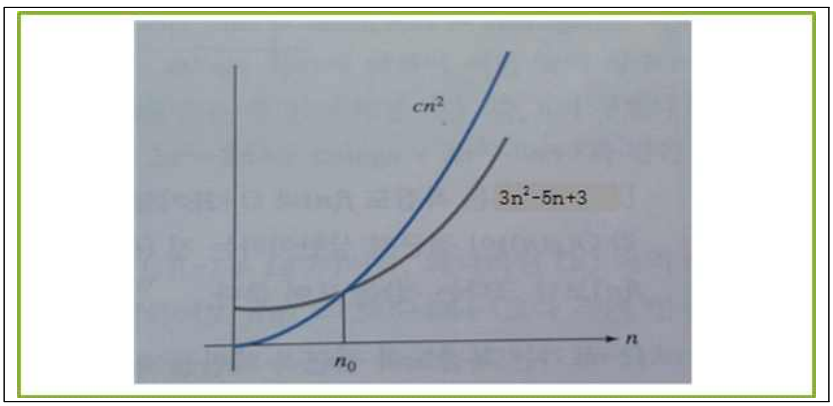
시간복잡도는 입력 크기에 대한 함수로 표기하는데, 이 함수는 주로 여러 개의 항을 가지는 다항식이다. 그래서 이를 단순한 함수로 표현하기 위해 점근적 표기(asymptotic notation)을 사용한다. 이는 입력 크기 n이 무한대로 커질 때 의 복잡도를 간단히 표현하기 위해 사용하는 표기법이다.
단순한 함수로 변환하는 예를 들면 3n3-15n+9n-15 일 때 n3 으로, 3n2-5n+3을 n2 으로 5n+6일 때 n으로와 같이 최고차의 항으로 단순화 시킨다. 이 단순화된 식에 상한, 하한, 동일한 증가율과 같은 개념을 적용하여 O-표기법(Big-Oh Notation), Ω 표기법(Big-Omega), Θ 표기법(Theta)과 같은 점근적 표기를 사용한다.
O-표기법(Big-Oh Notation)는 복잡도의 점근적 상한을 나타내고, Ω 표기법(Big-Omega)는 점근적 하한을 나타내며, Θ 표기법(Theta)는 점근적 상한과 하한이 동시에 적용되는 경우를 나타낸다.

1. 빅-오 표기법(Big-Oh Notation)

O-표기법(Big-Oh Notation)은 점근적 상한을 나타낸다.

예)
복잡도가 f(n)=3n2-5n+3이라면 f(n)의 O-표기는 O(n2)이다 먼저 f(n)의 단순화된 표현은 n2 이다 그리고 상한의 개념을 n이 증가함에 따라 부여하는 과정은 다음과 같다.

단순화된 함수 n2 에 임의의 상수 c를 곱한 cn2이 증가함에 따라 f(n)의 상한이 된다.



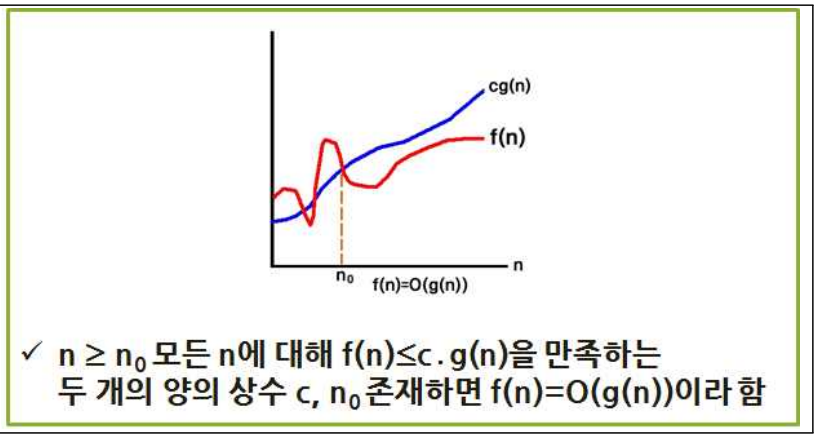
위의 그림에서 c=5 일 때 f(n)=3n2-5n+3과 5n2과의 교차점이 생기는데 이 교차점 이후 모든 n에 대해 , 즉 n이 무한대로 증가할 때 f(n)=3n2-5n+3은 5n2보다 절대로 커질 수 없다. 따라서 O(n2)이 3n2-5n+3의 점근적 상한이 되는 것이다. 위에서 c값을 5로 하였지만 굳이 c=5가 아니더라도 교차점 이후 상한 관계를 만족하는 어떤 양수가 존재한다면 f(n)=O(n2)으로 표기할 수 있다. 따라서 O-표기법에는 상수 c가 숨겨져 있다고 생각해도 좋다.

그러나 f(n)≠O(logn), f(n)≠O(n), f(n)≠O(nlogn)이다, 왜냐하면 O()속의 logn, n, nlogn 각각에 대하여 어떠한 양의 상수 c를 선택하여도 f(n)=3n2-5n+3보다 크게 만들 수 없기 때문이다. 즉 n이 무한대로 증가 하는데 clogn > 3n2-5n+3, cn > 3n2-5n+3, cnlogn > 3n2-5n+3과 같은 관계를 만족시키는 양의 상수 c가 존재 하지 않는다.

반면에 f(n)=O(n3), f(n)=O(n2)이다, 왜냐하면 O()속의 n3, 2n 각각에 대하여 양의 상수 c를 선택하면 cn3 > 3n2-5n+3, c2n > 3n2-5n+3 관계가 n이 무한대로 증가함에 따라 항상 성립하기 때문이다.

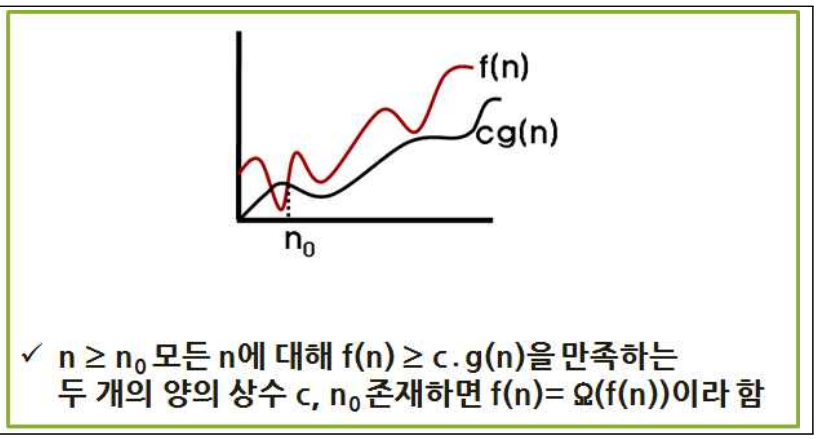
2. 점근적 상한과 하한

① 점근적 상한



② 점근적 하한(asymptotic lower bound)

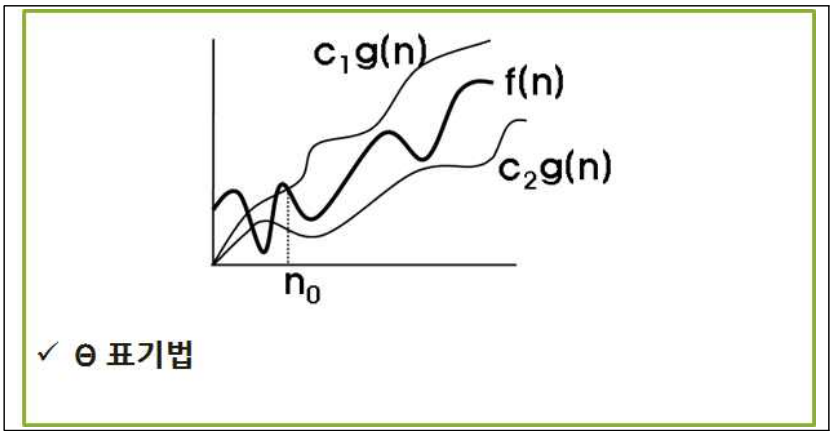
Ω 표기법는 복잡도의 점근적인 하한을 의미하는데 f(n)=3n2-5n+3의 Ω 표기는 Ω(n2)이다. f(n)= Ω(n2)은 n이 증가함에 따라 3n2-5n+3이 cn2보다 작을 수 없다는 의미이다. 이때 상수 c=1로 놓으면 된다. O 표기 때와 마찬가지로 Ω 표기도 복잡도 다항식의 최고차항만 계수 없이 취하면 된다.



따라서 f(n)= Ω(logn), f(n)= Ω(n), f(n)= Ω(nlogn)은 각각 성립한다. 왜냐하면 Ω()속의 logn, n, nlogn 각각에 대하여 어떠한 양의 상수 c를 선택하여도 f(n)=3n2-5n+3보다 작게 만들 수 있기 때문이다. 즉 n이 무한대로 증가 하는데 clogn < 3n2-5n+3, cn < 3n2-5n+3, cnlogn < 3n2-5n+3과 같은 관계를 만족시키는 양의 상수 c가 있다는 뜻이다.

반면 f(n)≠ Ω(n3), f(n)≠ Ω(2n)이다, 왜냐하면 Ω()속의 n3, 2n 각각에 대하여 어떤 양의 상수 c를 선택하여도 f(n)=3n2-5n+3보다 작게 만들 수 없기 때문이다. 즉 n이 증가함에 따라 cn3 < 3n2-5n+3, c2n < 3n2-5n+3 과 같은 관계를 만족시킬 양의 상수 c가 존재하지 않는다.

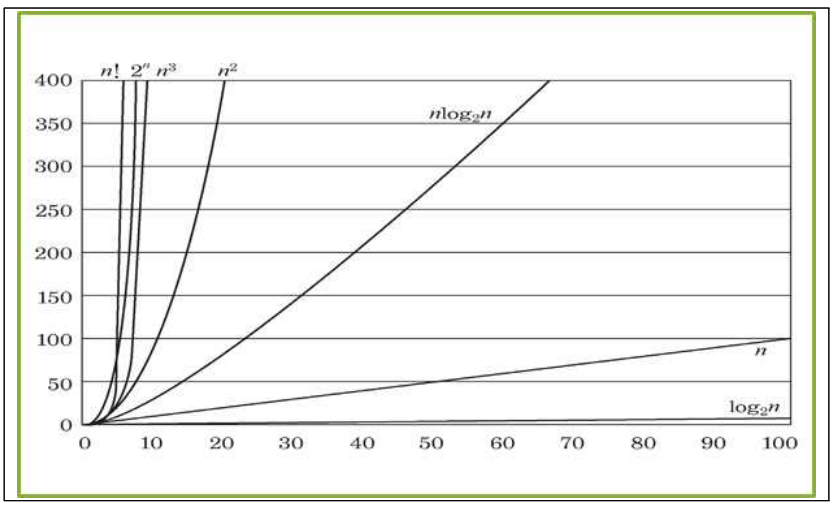
③ 점근적 상한과 하한



점근성능이란 입력의 크기 n이 커질 때의 알고리즘 성능을 의미하는 것으로, 점근성능을 표기하기 위한 방법으로 점근적 상한을 나타내는 f(n)=O(g(n)), 점근적 하한을 나타내는 f(n)=Ω(g(n)), 그리고 상한과 하한을 동시에 갖는 f(n)=Θ(g(n)) 등이 있다.

3. 빅오 복잡도 함수의 표현

빅오 복잡도 함수를 그래프로 표현하였다. 가로축은 n, 세로축은 n의 연산 결과 $n!$ 2^n n^3 n^2 $n\log_2 n$ n $\log_2 n$ 그래프, 즉 n이 커지면 커질수록, 시간 복잡도 함수는 다음과 같이 나타낼 수 있다(작을수록 빠르다).



복잡도는 일반적으로 빅-오 표기법사용하며 표현하며 아래 표는 시간 복잡도를 위해 자주 사용하는 빅-오 표기들이다.

시간	이름
O(1)	상수 시간(constant time)
O(logn)	로그(대수)시간(logarithmic time)
O(n)	선형 시간(linear time)
O(nlogn)	로그 선형 시간(log-linear time)
$O(n^2)$	제곱 시간(quadratic time)
$O(n^3)$	세제곱 시간(cubic time)
$O(n^k)$	다항식 시간(polynomial time)
$O(2^n)$	지수 시간(exponential time)

$$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(2^n) < O(n!)$$

4. 왜 효율적인 알고리즘이 필요한가?

$O(n^2)$	1,000	1백만	10억
PC	<1초	2시간	300년
슈퍼컴	<1초	1초	1주일

O(nlogn)	1,000	1백만	10억
PC	<1초	<1초	5분
슈퍼컴	<1초	<1초	<1초

10억 개의 숫자를 정렬하는데 PC에서 O(n2)알고리즘은 300년이 걸리는 반면O(nlogn) 알고리즘은 5분 만에 정렬한다. 입력크기가 작으면 어느 알고리즘을 사용해도 비슷한 수행시간이 걸리나 입력이 커지면 커질수록 그 차이는 더욱 커짐을 알 수 있다.

학습내용3 : 재귀 알고리즘

1) 재귀(순환)의 소개

재귀(순환)(recursion)이란 어떤 알고리즘이나 함수가 자신을 호출하여 문제를 해결하는 프로그래밍 기법이다. 이것은 처음에는 상당히 이상해 보이지만 순환은 효과적인 프로그래밍 기법중 하나이다.

순환은 본질적으로 순환적으로 정의된 문제나 자료구조를 다루는 프로그램에 적합하다.

2) 팩토리얼의 정의

$$n! = \begin{cases} 1 & n = 0 \\ n*(n-1)! & n \geq 1 \end{cases}$$

위의 정의에서 팩토리얼 n!을 정의 하는데 팩토리얼(n-1)!이 사용되었다 이러한 정의를 순환적이라 한다.

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

3) 피보나치 수열의 계산

일반적으로 순환을 사용하게 되면 단순하게 작성이 가능하며 가독성이 높아짐

- 그러나 똑같은 계산을 몇 번씩 반복하게 되면 계산시간이 엄청나게 길어질 수 있음

이러한 예로 피보나치수열을 계산해보면 아래와 같음

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

앞의 두 개의 숫자를 더해서 뒤의 숫자를 만들면 된다.

정의에 따라 수열을 만들어 보면 다음과 같다.

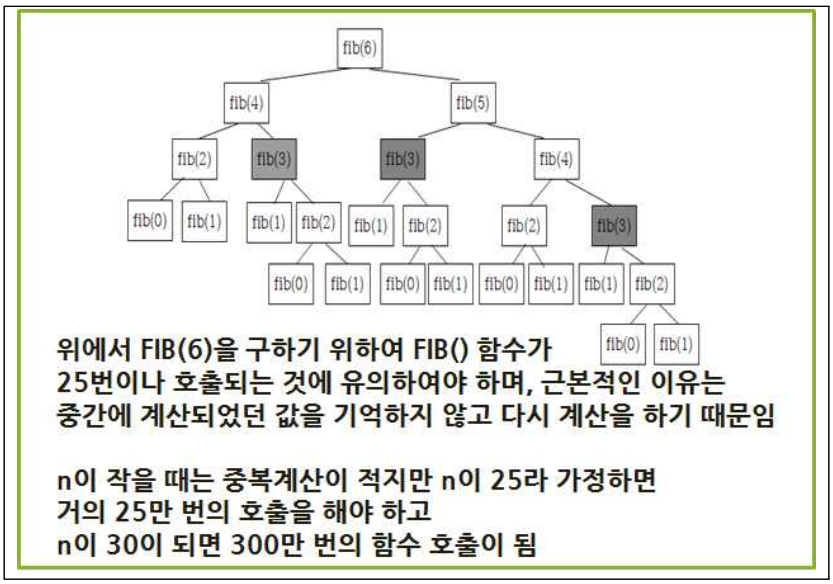
```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55.....

#include<stdio.h>

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

방금 전 예는 매우 단순하고 이해하기 쉽지만 비효율적이다.

- fib(6)으로 호출하였을 경우 fib(4)가 두 번이나 계산되고 fib(3)은 3번 계산이 되기 때문이다.
- 이런 현상은 순황이 커질수록 점점 심해지게 된다.



- 그래서 이와 같은 경우는 순환을 사용하지 않고 반복구조를 사용하면 좋은 결과를 얻을 수 있다.

* 반복구조 예

```
int fib_iter(int n)
{
        if( n<2) return n;
        else {
            int i, tmp, current=1, last=0;
      for(i=2; i<=n; i++){
      tmp=current;
      current=current + last;
      last=tmp;
        }
   return current;
  }
}
```

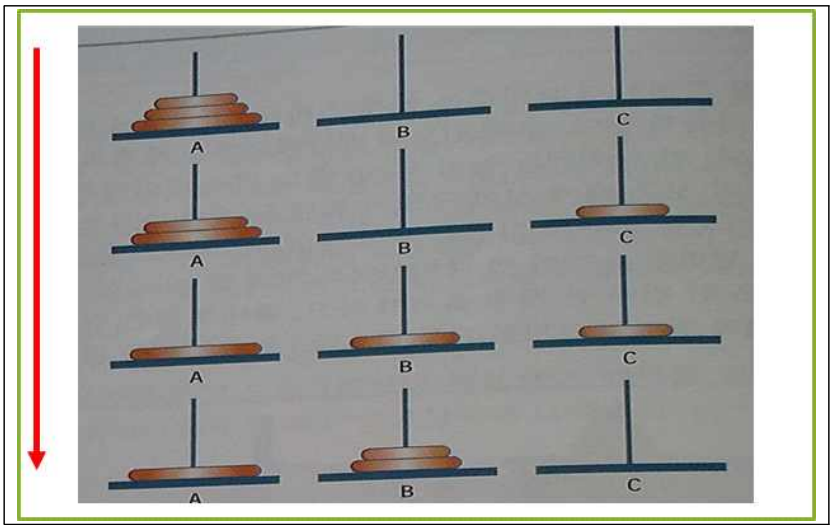
4) 하노이탑

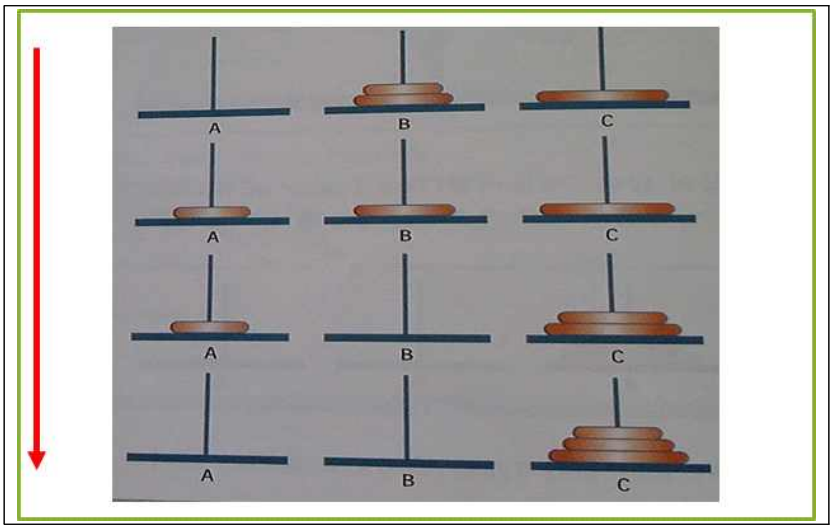
하노이 탑이란 1883년 프랑스 수학자 루카스(Lucas)에 의해 고안된 문제인데, 가운데 기둥을 이용해서 왼쪽 기둥에 놓인 크기가 다른 원판을 오른쪽 기둥으로 옮기는 문제였다. 이때 원판은 한 번에 한 개씩만 옮길 수 있으며, 작은 원판 위에 큰 원판이 놓일 수 없다는 조건이 따른다.

조건을 정리해보면

◎ 한 번에 하나의 원판만 이동할 수 있다.
◎ 맨 위에 있는 원판만 이동할 수 있다.
◎ 크기가 작은 원판 위에 큰 원판이 쌓일 수 없다.
◎ 중간의 막대를 임시적으로 이용할 수 있으나 앞의 조건을 지켜야 한다.

* 3개의 원판이 있는 경우에 아래와 같이 이동함

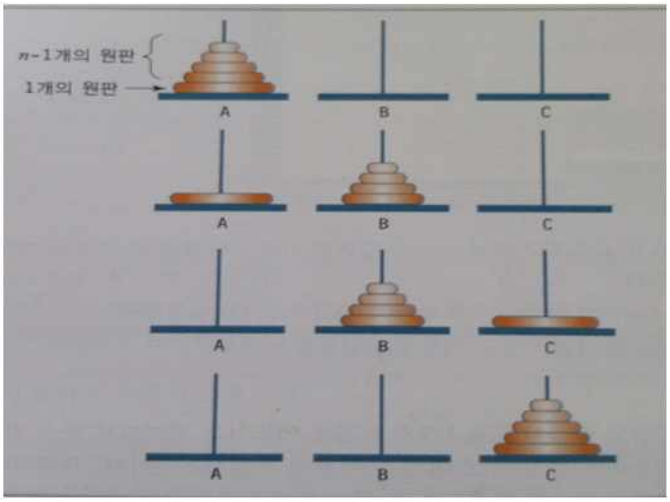




* 4개의 원판이 있는 경우에는 조금 더 복잡해짐
- 더 나아가 n개의 원판이 있는 경우를 해결하려면 상당히 복잡해진다.
- 이 문제는 순환적으로 생각하면 쉽게 해결할 수 있다.

- n개의 원판이 A에 쌓여 있는 경우 먼저 위에 쌓여있는 n-1개의 원판을 B로 옮긴 다음 제일 밑에 있는 원판을 C로 옮긴다.
- B에 있던 n-1개의 원판을 C로 옮긴다.

* 문제는 B에 쌓여있는 원판을 어떻게 C로 옮기느냐임
- 이 문제를 다음과 같이 알고리즘을 만들어 생각해 보자




```
// 막대 from에 쌓여 있는 n개의 원판을 막대 tmp를 사용하여
막대 to로 옮긴다.
void hanoi_tower(int n, char from, char tmp, char to)
{
            if (n==1){
               from에서 to로 원판을 옮긴다.
                     }
   else{
       hanoi_tower(n-1, from, to, tmp)
     // from의 맨 밑의 원판을 제외한 나머지 원판들을 tmp로 옮긴다.
       from에 있는 한 개의 원판을 to로 옮긴다.
       hanoi_tower(n-1, tmp, from to)
     // tmp의 원판들을 to로 옮긴다.
      }
}
```

```
#include <stdio.h>
void hanoi_tower(int n, char from, char tmp, char to)
 {
            if (n==1) printf("원판 1을 %c에서 %c로 옮긴다.₩n, from, to);
            else{
               hanoi_tower(n-1, from, to, tmp);
              printf("원판 %d를 %c에서 %c로 옮긴다.₩n", n, from, to);
              hanoi_tower(n-1, tmp, from, to);
            }
 }
main()
{
            hanoi_tower(4, 'A', 'B', 'C');
}
```

【학습정리】

1. 계산 복잡도와 표기법

- 직접 구현하지 않고서도 알고리즘의 효율성을 따져보는 기법으로 알고리즘의 복잡도 분석이 있다. 이 분석에는 두 가지 방법이 있는데 알고리즘의 수행 시간을 분석하는 시간 복잡도(time complexity)와 알고리즘이 사용하는 기억공간을 분석하는 공간 복잡도(space complexity)가 있다. 시간복잡도는 입력크기에 대한 함수로 표기하는데, 이 함수는 주로 여러개의 항을 가지는 다항식이다. 그래서 이를 단순한 함수로 표현하기 위해 점근적 표기를 사용한다. 이는 입력의 크기가 n이 무한대로 커질 때의 복잡도를 간단히 하기 위해서 사용하는 표기법으로 빅-오 표기법, 오메가 표기법, 세타 표기법이 있다.

2. 시간복잡도의 점근적 표기법

- O-표기법(Big-Oh Notation)는 복잡도의 점근적 상한을 나타내고, Ω 표기법(Big-Omega)는 점근적 하한을 나타내며, θ 표기법(Theta)는 점근적 상한과 하한이 동시에 적용되는 경우를 나타낸다.

3. 재귀 알고리즘

- 재귀(순환)(recursion)이란 어떤 알고리즘이나 함수가 자신을 호출하여 문제를 해결하는 프로그래밍 기법이다. 이것은 처음에는 상당히 이상해 보이지만 순환은 효과적인 프로그래밍 기법중 하나이다.