



# D.COM ALGORITHM STUDY

# 6강. 디피디피뿔!

# 오늘의 목표

첫번째, 복잡도에 대해 알아보시다!

두번째로, 대망의 다이나믹 프로그래밍에 대해 알아보시다!

#스압 주의!

가볍게,  
읽을거리




링크


## 개발 배우기가 정말 어려운 이유 [번역]

“개발을 배우는 과정에 대한 정말 좋은 글입니다!  
필독을 권합니다~!”

우선,  
**복잡도**에 대해 알아보시다  
Complexity



알고리즘이  
얼마나 효율적인지  
어떻게 표현할 수 있을까요?



알고리즘의 평가 방법에는  
**시간 복잡도와 공간 복잡도**  
두가지가 있습니다!

**시간 복잡도** → 알고리즘의 연산의 횟수

**공간 복잡도** → 알고리즘의 메모리 사용량



# 시간 복잡도

Time Complexity  
알고리즘의 연산의 횟수



# 시간 복잡도



시간 복잡도는 알고리즘의 연산의 횟수를 의미합니다.  
그렇다면 알고리즘의 연산의 횟수는 어떻게 구할 수 있을까요?

다음 3가지 경우로 나눌 수 있습니다.

1. 최선의 경우 Best Case
2. 평균적인 경우 Average Case
3. 최악의 경우 Worst Case

## 시간 복잡도



얼핏 보기에는 평균적인 경우가 이상적인 시간 복잡도로 보입니다.  
하지만 이는 알고리즘이 복잡할 때 구하기 매우 어려워집니다.

따라서 “최악의 경우”로 알고리즘의 연산 횟수를 파악합니다.

# 시간 복잡도

다음 1부터 N까지의 합을 구하는 3가지 알고리즘의  
연산의 횟수, 즉 시간 복잡도를 구해봅시다!



```
int sum = 0;
for (int i=1; i<=N; i++) {
    sum += i;
}
```



```
int sum = 0;
for (int i=1; i<=N; i++) {
    for (int j=1; j<=N; j++) {
        if (i == j) {
            sum += j;
        }
    }
}
```



```
int sum = 0;
sum = N*(N+1)/2;
```

# 시간 복잡도

반복문이 곧 연산 횟수를 결정합니다!  
N이 입력 데이터의 양이라면 연산 횟수는 다음과 같습니다!

$O(N)$

```
int sum = 0;
for (int i=1; i<=N; i++) {
    sum += i;
}
```

$O(N^2)$

```
int sum = 0;
for (int i=1; i<=N; i++) {
    for (int j=1; j<=N; j++) {
        if (i == j) {
            sum += j;
        }
    }
}
```

$O(1)$

```
int sum = 0;
sum = N*(N+1)/2;
```

#0의 의미는  
다음 슬라이드에서 알아보니다!

읽을거리



[링크](#)

알고리즘의 시간 복잡도와 Big-O 쉽게 이해하기

시간 복잡도에서 가장 영향력이 큰 부분을  
알아내는 방법입니다.  
어렵지 않습니다!

## 시간 복잡도

---

Q. 알고리즘의 수행 시간이 아닌 연산의 횟수가  
시간 복잡도인 이유는 무엇인가요?

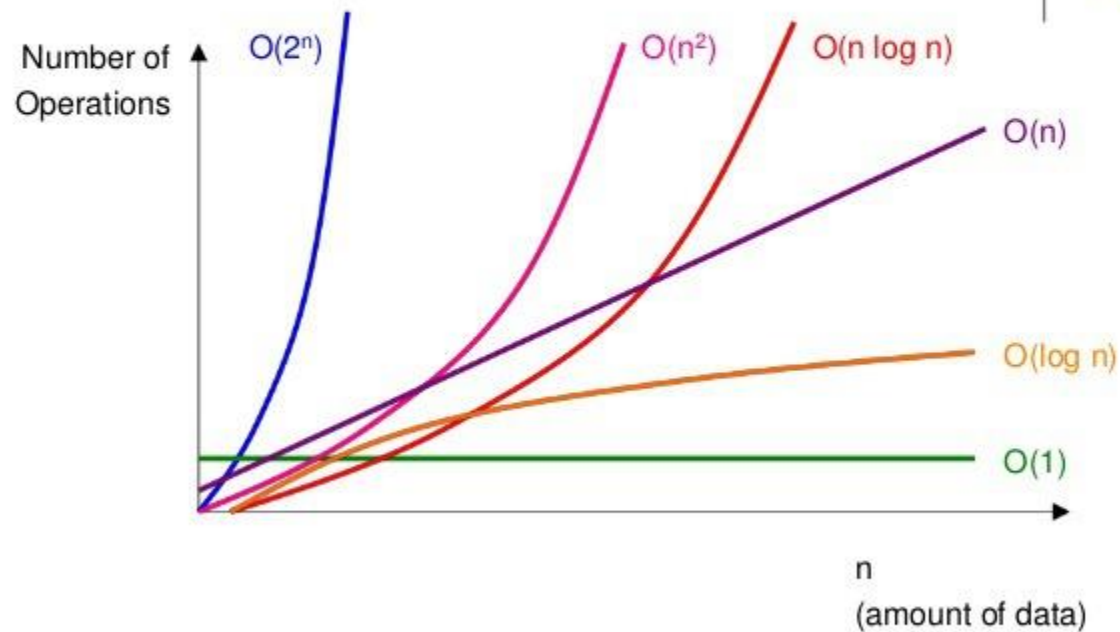
---

알고리즘의 수행시간을 기준으로 삼으면 다음과 같은 문제점이 있습니다.

1. 측정을 위해서는 구동 가능한 완성된 프로그램이 필요하다. 즉, 개발 전에 예측이 힘들다.
2. 컴퓨터마다 수행 시간 결과가 달라진다. Ex) 슈퍼 컴퓨터 VS 노트북

# 시간 복잡도

## Comparing Big O Functions



(C) 2010 Thomas J Cortina, Carnegie Mellon University

빅오 표기법의 종류를 나타낸 그래프입니다.

특히,  $O(n \log n)$  이상부터는 데이터의 양에 따라 연산 횟수가 **기하급수적**으로 증가합니다!

따라서, 데이터의 양이 많을 때 다음과 같은 시간 복잡도를 가지는 알고리즘의 사용에 주의하여야 합니다!

# 시간 복잡도



“대략 1억 번의 연산 횟수는 1초의 실행 시간이 걸립니다.”

대부분의 알고리즘 문제는 시간 제한과 최대 데이터 개수가 주어집니다.

따라서, 알고리즘을 구현하기 전 대략적인 시간 복잡도를 생각해 둔다면,  
이 문제에 적합한 알고리즘인지 판단할 수 있습니다!

#앞으로 문제를 풀 때 시간 복잡도를 생각하는 습관을 가집시다!



# 공간 복잡도

Space Complexity  
알고리즘의 메모리 사용량

# 공간 복잡도



공간 복잡도는 프로그램을 실행시켜 완료하는데 필요한 메모리의 양입니다.

하지만 보통 알고리즘 문제에서 메모리는 넉넉하게 주어지기 때문에  
크게 걱정하지 않아도 됩니다!

## 공간 복잡도



주로 배열의 크기를 통해 공간 복잡도를 계산합니다.


배열의 크기 x 자료형의 크기 = 배열이 사용한 공간의 크기



`int arr[1000]` =>  $1000 * 4B = 4000B$

`int arr[1000][1000]` =>  $1000 * 1000 * 4B = 4000000B = 4MB$

“좋은 알고리즘”이란  
주어진 상황과 문제에  
적합한 시간&공간 복잡도를 가진  
알고리즘을 의미합니다.



시간&공간 복잡도가 낮다고 무조건 좋은 알고리즘이 아닙니다!  
왜냐하면 대체로 복잡도가 낮은 알고리즘일수록 구현 난이도가 올라가기 때문입니다.  
즉, 간단한 문제에 굳이 복잡한 알고리즘을 적용하는 것은 비효율적입니다.  
“닭 잡는데 어찌 소 잡는 칼을 쓰는가”

이제,

# 다이나믹 프로그래밍

을 알아봅시다!



# 동적 계획법

Dynamic Programming

큰 문제를 작은 문제로  
나눠서 푸는 기법

# 동적 계획법

Dynamic Programming

---

이 기법은 Richard Bellman이 고안했으며,  
Dynamic이라는 단어가 멋있어 보여 선택했다고 합니다.  
즉, Dynamic Programming이라는  
용어 자체에 큰 의미는 없습니다.

# 동적 계획법

Dynamic Programming

---

“큰 문제를 작은 문제로 나눠서 푼다?”

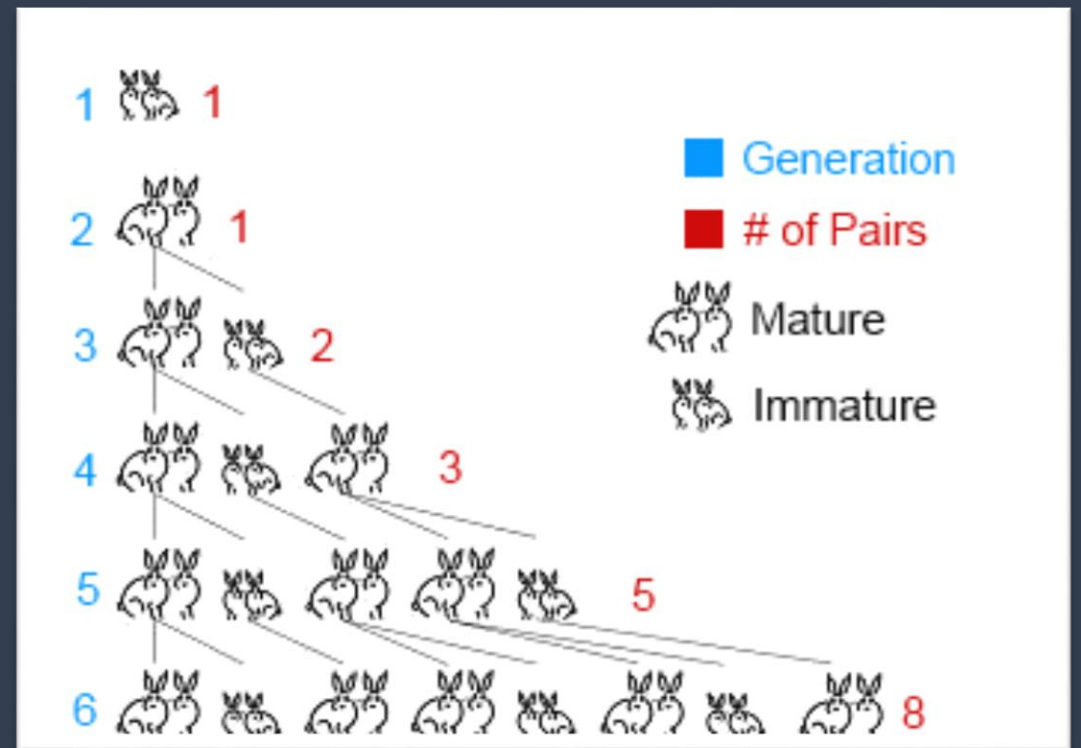
작은 문제의 정답을 구하다 보면,  
큰 문제의 정답을 구할 수 있습니다.  
다음 슬라이드에서 자세히 알아보시다!



고등학교 때 한번씩 보셨을  
**피보나치 수열**입니다!

우리는 이 문제를 풀기 위해  
‘점화식’이라는 개념을 도입했습니다!

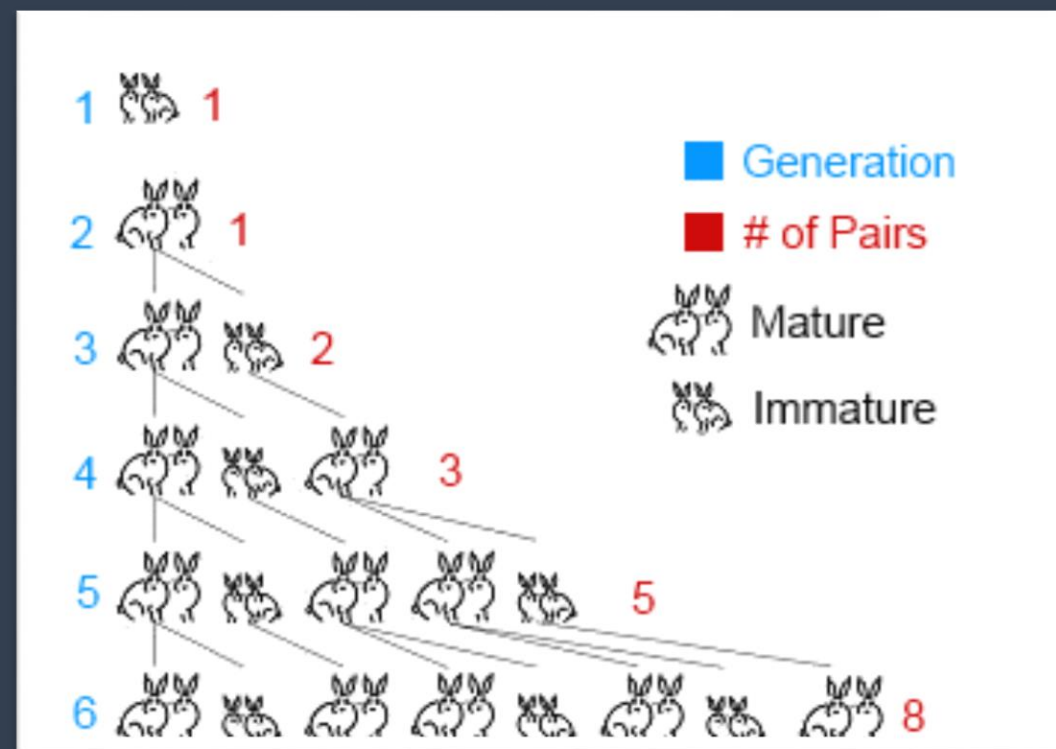
$$A_n = A_{n-1} + A_{n-2}$$



## 점화식이란

‘인접한 항들 사이의 관계식’을  
의미합니다.

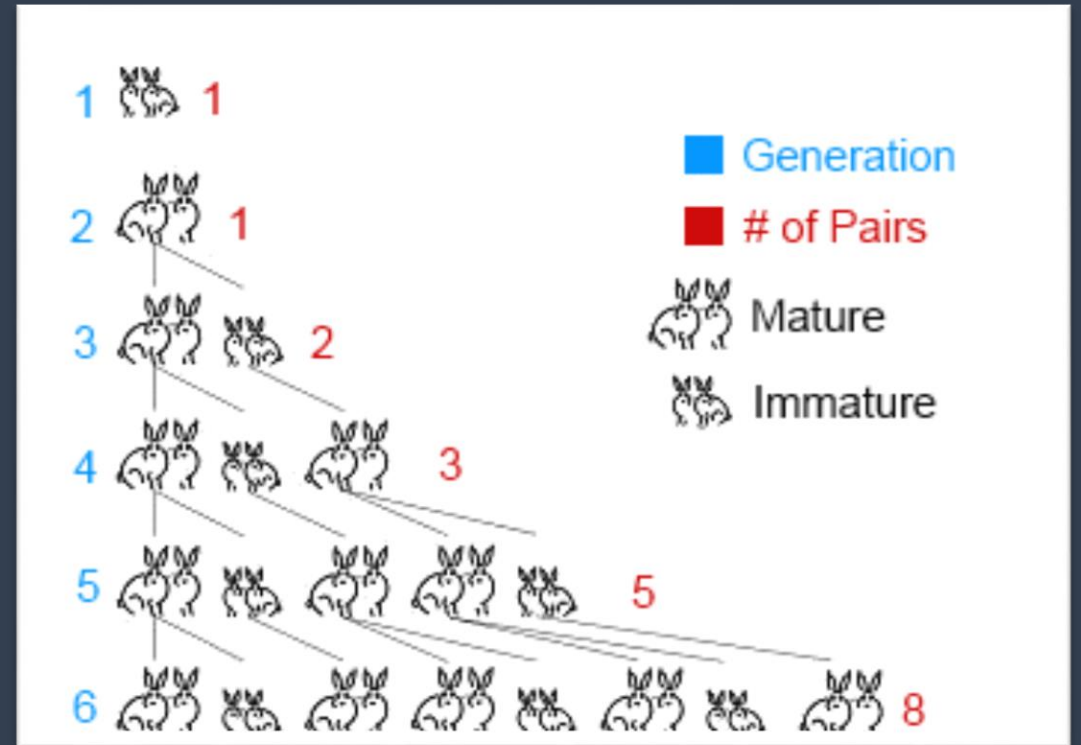
$$A_n = A_{n-1} + A_{n-2}$$



피보나치 수열의 점화식에 따라,  
N번째 항을 구하려면, N-1번째 항과 N-2번째 항을 알아야 하고,  
N-1 번째 항을 구하려면 N-2번째 항과 N-3번째 항을 알아야 하고,  
결국 N번째 항은 그보다 작은 항들을 품으로써 알 수 있습니다!

즉, 피보나치 N 문제를 구하기 위해서는  
그보다 작은 피보나치 문제들을 풀어야합니다!  
이것이 동적 계획법의 기초 아이디어입니다!  
구하기 어려운 큰 문제는  
구하기 쉬운 작은 문제들을 통해 풀 수 있습니다!

$$A_n = A_{n-1} + A_{n-2}$$



앞에서 알아본 원리를 어떻게 구현할 수 있을까요?

# 재귀(Recursion)을 이용하여 구현할 수 있습니다!

---

재귀에 대한 자세한 설명은 다음 링크를 참고해주세요!

<https://psychoria.tistory.com/28>

피보나치 수열을 재귀를 이용하여  
구현하면 다음과 같습니다.

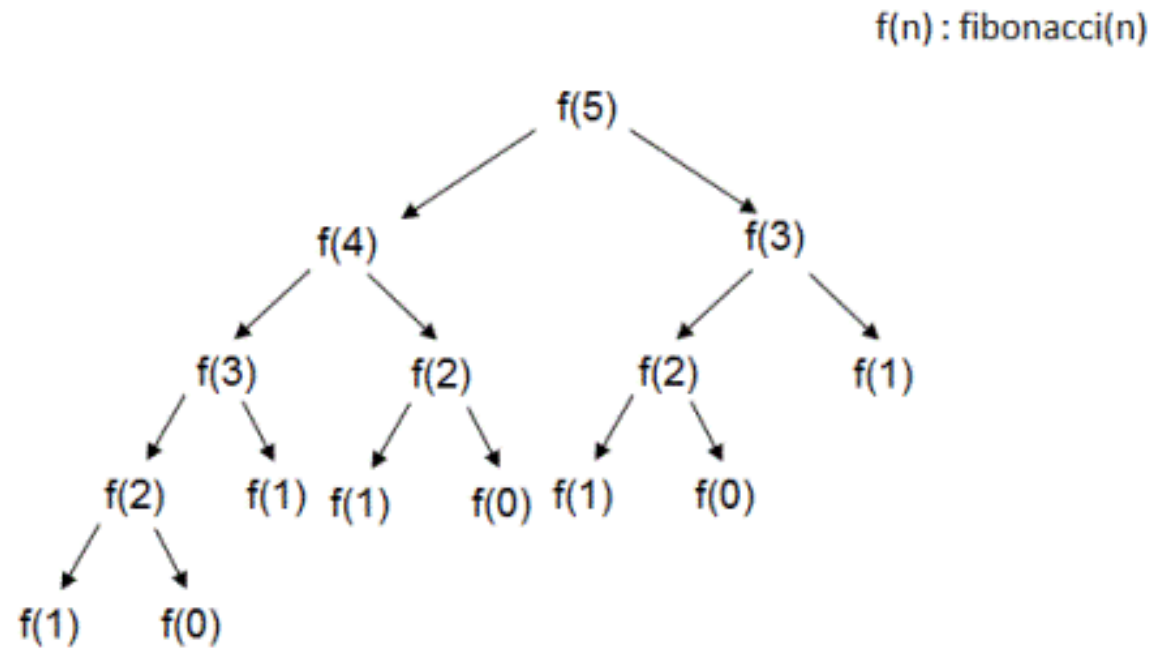
```
int Fibo(int N)
{
    if (N == 1)
    {
        return 0;
    }
    else if (N == 2)
    {
        return 1;
    }
    else
    {
        return Fibo(N - 1) + Fibo(N - 2);
    }
}
```

Q.

하지만 앞의 코드로 피보나치 50을 빠르게 구할 수 있을까요?

원하는 시간안에 구할 수 없다면,  
그 이유는 무엇일까요?

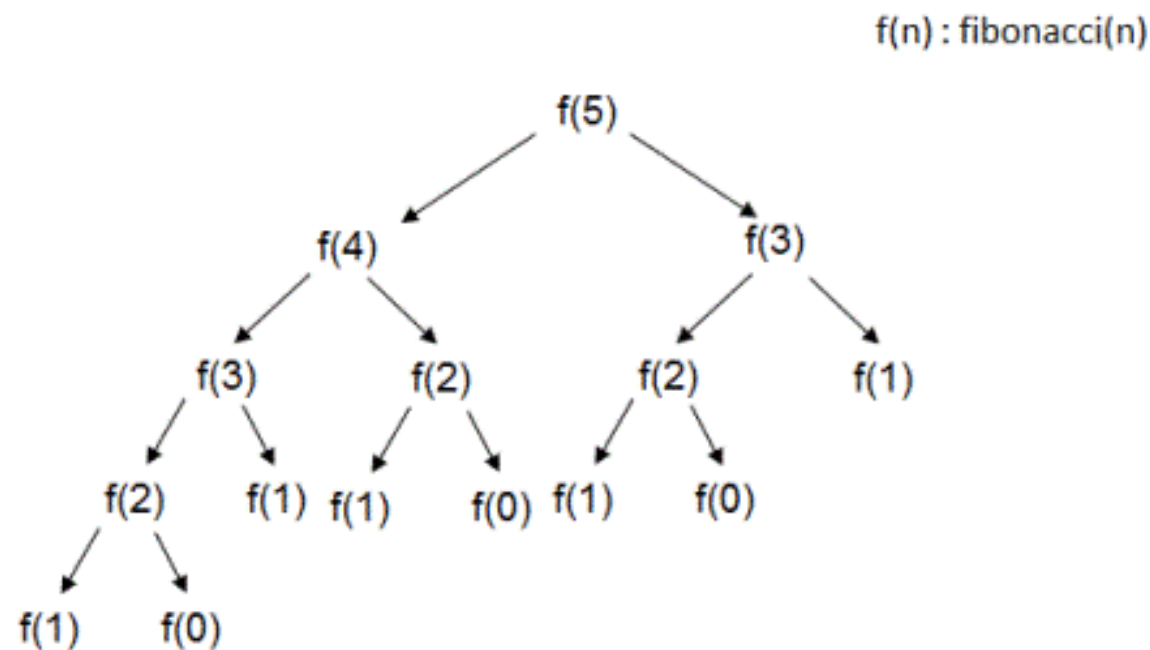
피보나치 5를 재귀로 구하는  
과정은 다음 그림과 같습니다!



Recursion tree generated for computing 5<sup>th</sup> number of fibonacci sequence



이미 호출했던 함수를 다시  
호출하는 것을 볼 수 있습니다.  
즉, 중복이 발생하고 있습니다!



Recursion tree generated for computing 5<sup>th</sup> number of fibonacci sequence

Q.

중복을 어떻게 피할 수 있을까요?

#개발자의 숙명입니다!

이미 한번 구한 값을 어딘가에  
메모해두면 되지 않을까요?

# 메모이제이션

Memoization

반복되는 결과를 메모해두는 것

#메모 '리' 제이션 아닙니다!

# 중복을 피해라!

#매우 중요합니다!

**메모이제이션**을 적용한 코드는 다음과 같습니다!  
한번 구한 값을 메모해두고 나중에 필요할 때 사용할 수 있습니다.

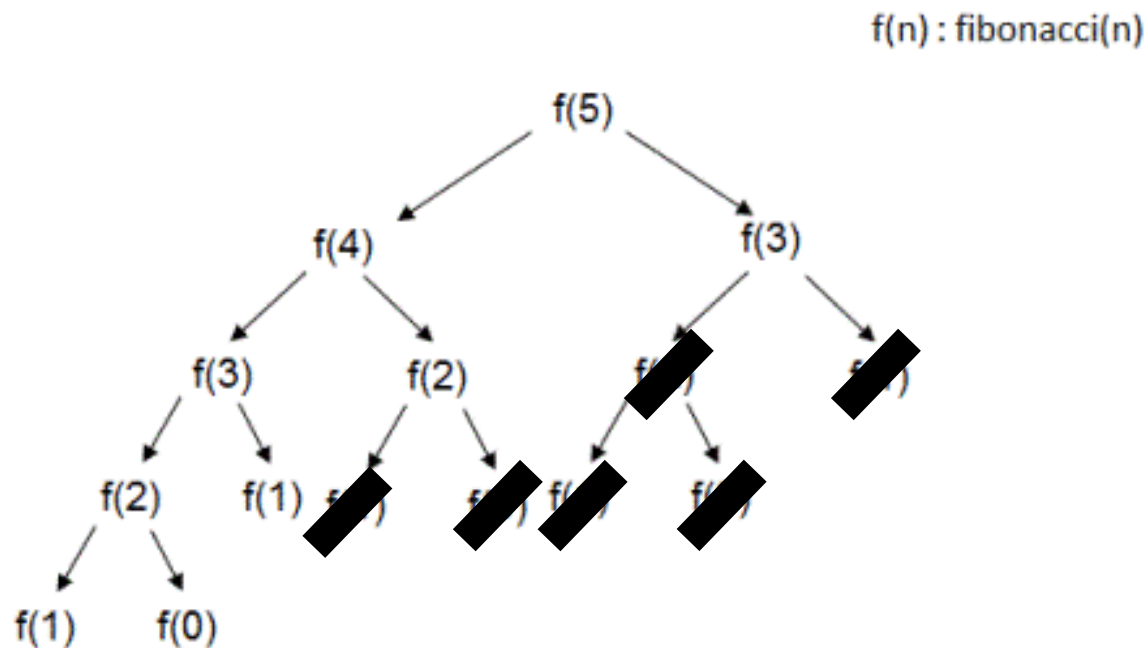
```
int Fibo(int N)
{
    if (N == 1)
    {
        return 0;
    }
    else if (N == 2)
    {
        return 1;
    }
    else
    {
        return Fibo(N - 1) + Fibo(N - 2);
    }
}
```



```
int memo[100];
int Fibo(int N)
{
    if (N == 1)
    {
        return 0;
    }
    else if (N == 2)
    {
        return 1;
    }
    else
    {
        if (memo[N] > 0)
        {
            return memo[N];
        }
        memo[N] = Fibo(N - 1) + Fibo(N - 2);
        return memo[N];
    }
}
```

# 메모의 힘!

**피보나치 수가 커질수록 더욱 진가를 발휘합니다!**



Recursion tree generated for computing 5<sup>th</sup> number of fibonacci sequence



연습 문제

2748번 피보나치 수 2

<https://www.acmicpc.net/problem/2748>

시간 제한에 유의 하주세요!



읽을거리



<https://moscowmule.tistory.com/4>

각 피보나치 알고리즘의 **시간 복잡도**를 알아봅시다!  
두 피보나치 수열 알고리즘의 성능 차이가 대체 얼마나 나길래  
메모이제이션을 사용하는 것이 바람직할까요?

다시, 동적계획법을 정의하면 다음과 같습니다.

큰 문제는 작은 문제들을 통해 푼다.  
단, 작은 문제의 정답을 구했으면, 어딘가에 메모해두자!

Q.

넵.. 동적 계획법의 개념은 알겠습니다..  
그래서 **어떻게** 구현하면 좋을까요?

DP를 구현하는 방식에는 **두가지**가 있습니다!

탑 다운 (Top-down)

바텀 업 (Bottom-Up)

# 탑 다운 (Top-down)

가장 큰 문제에서부터 점점 작은 문제를 풀어나가는 방식입니다.  
주로 재귀를 이용하여 구현합니다.

# 바텀 업 (Bottom-Up)

탑다운과 반대로, 작은 문제에서 점점 큰 문제를 풀어나가는 방식입니다.  
주로 반복문을 이용하여 구현합니다.

■      방금 배웠던 피보나치 수열을  
두가지 방식을 이용하여 구현해봅시다!


# 탑 다운 (Top-down)

```
int memo[100];
int Fibo(int N)
{
    if (N == 1)
    {
        return 0;
    }
    else if (N == 2)
    {
        return 1;
    }
    else
    {
        if (memo[N] > 0)
        {
            return memo[N];
        }
        memo[N] = Fibo(N - 1) + Fibo(N - 2);
        return memo[N];
    }
}
```

앞서 배운 방식과 같습니다!  
Fibo(N)을 구하기 위해, Fibo(N-1)과 Fibo(N-2)라는 작은 문제로 나눠 풉니다.  
이렇게 큰 문제에서 점점 작은 문제를 풀어나가다 보면 결국 큰 문제를 풀 수 있습니다.  
주로 재귀를 이용하여 구현합니다!



# 바텀 업 (Bottom-up)



```
int dp[100];
int Fibo(int N)
{
    dp[1] = 0;
    dp[2] = 1;
    for (int i = 3; i <= N; i++)
    {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[N];
}
```

Fibo(N)을 구하기 위해, Fibo(1)부터 풀어나갑니다.  
이렇게 작은 문제에서 점점 큰 문제를 풀어나가다 보면  
결국 큰 문제를 풀 수 있습니다.  
주로 반복문을 이용하여 구현합니다.



연습 문제  
2747번 피보나치 수

<https://www.acmicpc.net/problem/2747>

앞서 푼 피보나치 문제와 비슷합니다!  
단, 이번에는 바텀 업 방식을 이용해서 구현해봅시다!

# 탭 다운 vs 바텀 업

탭 다운 방식은 점화식을 표현하기 쉽다는 장점이 있고,  
**바텀 업** 방식은 재귀 함수를 사용하지 않기 때문에  
메모리와 처리시간을 절약할 수 있다는 장점이 있습니다.

각자 자신에게 잘 맞는 방법을 사용하시면 됩니다!  
간혹 두 방법 간의 구현 난이도 편차가 큰 문제도 있기에,  
두 방법 모두 알아 두셔야 합니다!

# DP 문제를 접근하는 방법

1. DP 문제인지 판단하기! (작은 문제로 쪼갤 수 있는가? 메모이제이션이 필요한가?)
2. 메모이제이션을 위한 배열 혹은 벡터를 만든다.
3. 큰 문제를 작은 문제로 표현하고, 점화식을 세운다.



연습 문제  
1463번 1로 만들기

<https://www.acmicpc.net/problem/1463>

우선 앞 슬라이드의 접근 방법을 생각해봅시다!  
그 후, 다음 슬라이드에서 같이 풀어봅시다!

## 1로 만들기 성공



시간 제한	메모리 제한	제출	정답	맞은 사람	정답 비율
2 초	128 MB	96210	31600	19974	31.945%

### 문제

정수  $X$ 에 사용할 수 있는 연산은 다음과 같이 세 가지 이다.

1.  $X$ 가 3으로 나누어 떨어지면, 3으로 나눈다.
2.  $X$ 가 2로 나누어 떨어지면, 2로 나눈다.
3. 1을 뺀다.

정수  $N$ 이 주어졌을 때, 위와 같은 연산 세 개를 적절히 사용해서 1을 만들려고 한다. 연산을 사용하는 횟수의 최솟값을 출력하시오.

### 입력

첫째 줄에 1보다 크거나 같고,  $10^6$ 보다 작거나 같은 정수  $N$ 이 주어진다.

### 출력

첫째 줄에 연산을 하는 횟수의 최솟값을 출력한다.

## 우선 문제를 읽어봅시다!

1463번 풀이

**1단계 DP 문제인지 확인하기!**

## 1단계 DP 문제인지 확인하기!

N을 1로 만드는 최소 횟수를 찾는 큰 문제는  
N/3, N/2, N-1을 1로 만드는 최소 횟수를 찾는 작은 문제로 생각할 수 있지 않을까요?  
=> 작은 문제를 통해 풀 수 있는가?

#DP에 익숙해지면 문제만 읽고도 DP 문제인지 감이 옵니다!



2단계 메모이제이션을 위한 배열을 만들자

## 2단계 메모이제이션을 위한 배열을 만들자

문제에서 정수 N은 1000000까지가 조건이므로,  
Int dp[1000005]로 전역 배열을 선언하자!

#전역 배열을 사용하면 유용합니다! 구체적인 이유는 다음 슬라이드에서 알아보시다.  
#배열 크기는 문제 조건보다 살짝 넉넉하게 잡으면 좋습니다.  
#항상, 배열 인덱스 초과에 주의합니다.

읽을거리



링크

## 왜 전역 배열을 사용해야 할까요?

“두 가지 장점이 있습니다.”

1. 선언과 동시에 자동으로 초기화 됩니다.
2. 지역배열 보다 더 많은 메모리를 할당 할 수 있습니다.

## 3단계 점화식을 세워보자

#가장 중요한 단계입니다.

## 1463번 풀이

앞에서 “N을 1로 만드는 큰 문제는  $N/3, N/2, N-1$ 을 1로 만드는 작은 문제로 풀 수 있을 것이다!”  
라 생각해보았습니다.

위 생각을 한번 구체화해봅시다!

N에게 가능한 경우는 **세가지**가 있습니다.

1. N이 3으로 나누어 떨어지면, 3으로 나눈다. (N은 자연수이기 때문)
2. N이 2로 나누어 떨어지면, 2로 나눈다. (N은 자연수이기 때문)
3. N에서 1로 뺀다.

$d[N]$ 을  $N$ 을 1로 바꾸는데 필요한 최소 연산 횟수라고 해봅시다.

그렇다면  $d[N]$ 은  $d[N/3]+1$ ,  $d[N/2]+1$ ,  $d[N-1]+1$  중에서의  
**최소값**이라 할 수 있지 않을까요?

왜냐하면 우리가 원하는 값은 최소 연산 횟수이기 때문입니다.

$d[i]$  ( $i \leq N$ )의 최소값을 구하다 보면 결국 원하는  $d[N]$ 을 구할 수 있습니다.

#아직 잘 이해가 가지 않는다면, 일단 슬라이드를 넘어가고,  
다음 문제를 풀고 다시 도전해보세요!

즉, 점화식은 다음과 같습니다.

$$d[n] = \min(d[n/3] + 1, d[n/2] + 1, d[n-1] + 1)$$



그리고 우리는 이미 작은 문제의 정답,  $D[1]=0$ ,  $D[2] = 1$ ,  $D[3] = 1$ 임을 알고 있습니다!  
이제 이를 탑 다운과 바텀 업 두가지 방식 중 하나로 구현하면 됩니다.

다음 페이지에서 좀 더 자세히 알아보시다!

1463번 풀이

# 바텀 업 Bottom Up

작은 문제에서 점점 큰 문제를 풀어나가 봅시다.

# 바텀 업

## Bottom Up

$d[1]=0$ ,  $d[2] = 1$ ,  $d[3] = 1$ 을 이미 알고 있습니다.  
이를 활용하여  $d[4]$ ,  $d[5]$ ,  $d[6]$  ... 를 차례대로 구한다면  
결국에는 원하는  $d[N]$ 을 구할 수 있습니다.

# 1463번 풀이

```
int dp[1000001];
int go(int N)
{
    for (int i = 2; i <= N; i++)
    {
        dp[i] = dp[i - 1] + 1;
        if (i % 2 == 0)
        {
            dp[i] = min(dp[i], dp[i / 2] + 1);
        }
        if (i % 3 == 0)
        {
            dp[i] = min(dp[i], dp[i / 3] + 1);
        }
    }
    return dp[N];
}
```

다음과 같이

**반복문**을 이용해서

**바텀 업** 방식으로 구현할 수 있습니다!

(주요 로직만 기술하였습니다. 완성된 정답 코드가 아닙니다!)

# 검증

바텀 업 방식의 시간 복잡도는 어떠할까요?  
과연 문제의 제한 조건 안에 풀 수 있을까요?

# 복잡도

단일 반복문으로 1~N까지 순회하므로  
시간 복잡도는  $O(N)$ 입니다.  
따라서 시간제한은 1초이고,  
정수 N은 최대 1000000(백만)까지로 주어지므로  
주어진 시간 안에 문제를 풀 수 있습니다.

# 탑 다운 Top Down

큰 문제에서 점점 작은 문제를 풀어나가 봅시다.

# 탑 다운 Top Down

$d[N]$ 에서  $d[N/3]$ ,  $d[N/2]$ ,  $d[N-1]$  값을 재귀적으로 구한다면  
 $d[N]$ 을 찾을 수 있습니다!



## 1463번 풀이

```
int d[1000001];
int go(int n) {
    if (n == 1) {
        return 0;
    }
    if (d[n] > 0) {
        return d[n];
    }
    d[n] = go(n-1) + 1;
    if (n%2 == 0) {
        int temp = go(n/2) + 1;
        if (d[n] > temp) {
            d[n] = temp;
        }
    }
    if (n%3 == 0) {
        int temp = go(n/3) + 1;
        if (d[n] > temp) {
            d[n] = temp;
        }
    }
    return d[n];
}
```

다음과 같이

**재귀**를 이용해서

**탑 다운** 방식으로 구현할 수 있습니다!

(주요 로직만 기술하였습니다. 완성된 정답 코드가 아닙니다!)

# 검증

탐 다운 방식의 시간 복잡도는 어떠할까요?  
특히 재귀 함수의 시간 복잡도는 어떻게 구할 수 있을까요?  
과연 문제의 제한 조건 안에 풀 수 있을까요?

# 복잡도

재귀 함수라고 어렵게 생각할 것 없습니다!  
N에서  $N/3$ ,  $N/2$ ,  $N-1$ 로 가는 복잡도는 각각 1입니다.  
그리고, 전체 데이터의 개수는 N개이므로,  
 $O(N)$ 의 시간 복잡도임을 알 수 있습니다.  
즉, 주어진 시간 안에 문제를 해결할 수 있습니다.

1463번 리뷰

# DP 문제를 접근하는 방법

1. DP 문제인지 판단하기! (작은 문제로 쪼갤 수 있는가? 메모이제이션이 필요한가?)
2. 메모이제이션을 위한 배열 혹은 벡터를 만든다.
3. 큰 문제를 작은 문제로 표현하고, 점화식을 세운다.



연습 문제  
11726번  $2 \times N$  타일링

<https://www.acmicpc.net/problem/11726>

우선 앞 슬라이드의 접근 방법을 생각해봅시다!  
그 후, 다음 슬라이드에서 같이 풀어봅시다!

11726번 풀이

**1단계 DP 문제인지 확인하기!**

# 1단계 DP 문제인지 확인하기!

N개의 타일을 채우는 문제는  
그보다 작은 타일을 채우는 문제로 풀 수 있지 않을까요?  
(DP 강의 자료에 있으니깐 DP 쓰겠지)

#DP에 익숙해지면 문제만 읽고도 DP 문제인지 감이 옵니다!

2단계 메모이제이션을 위한 배열을 만들자



## 2단계 메모이제이션을 위한 배열을 만들자

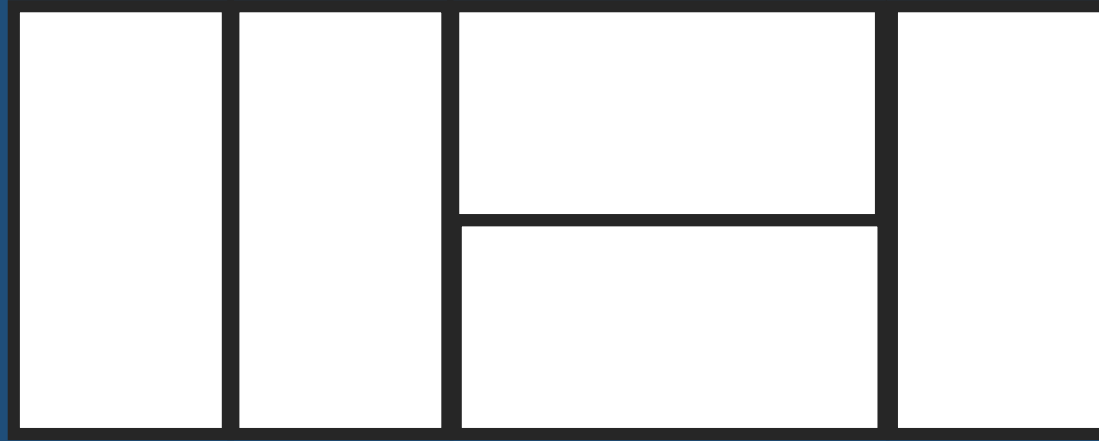
문제에서 정수 N은 1000까지가 조건이므로,  
Int dp[1005]로 전역 배열을 선언하자!

#전역 배열을 사용하면 유용합니다!  
#배열 크기는 문제 조건보다 살짝 넉넉하게 잡으면 좋습니다.  
#항상, 배열 인덱스 초과에 주의합니다.

## 3단계 점화식을 세워보자

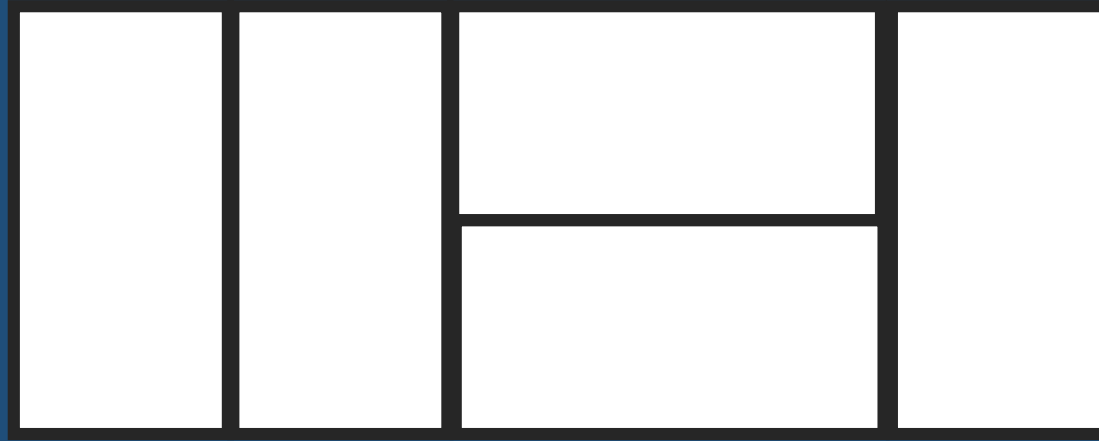
#가장 중요한 단계입니다.

# 11726번 풀이



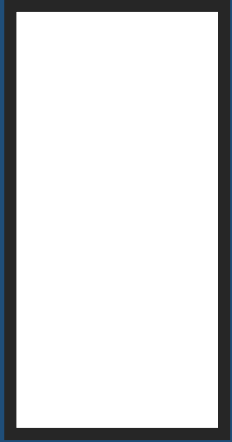
$d[n]$ 을  $2*n$  직사각형을 채우는 경우의 수라 해봅시다!

# 11726번 풀이



여기에서 어떤 점화식을 찾을 수 있을까요?  
잘 보이지 않는다면, 작은 숫자부터 직접 그려보며 풀어보는 것도 좋은 방법입니다!

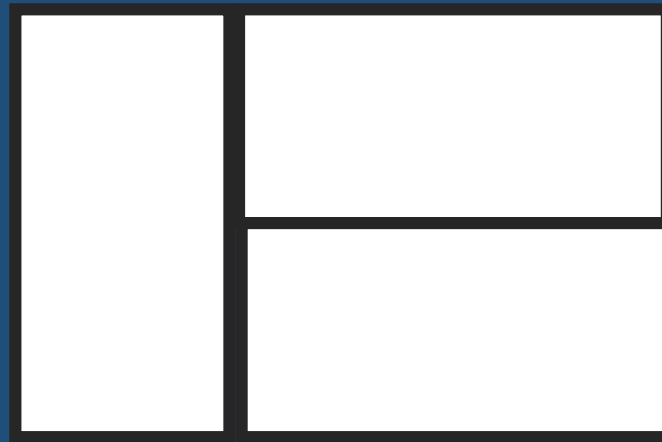
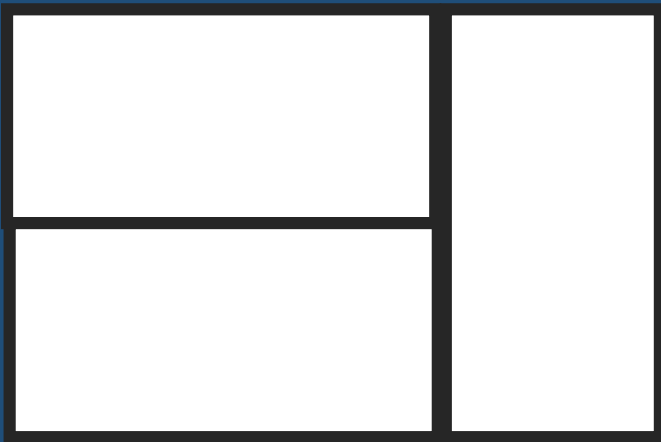
11726번 풀이



$d[1] = 1$

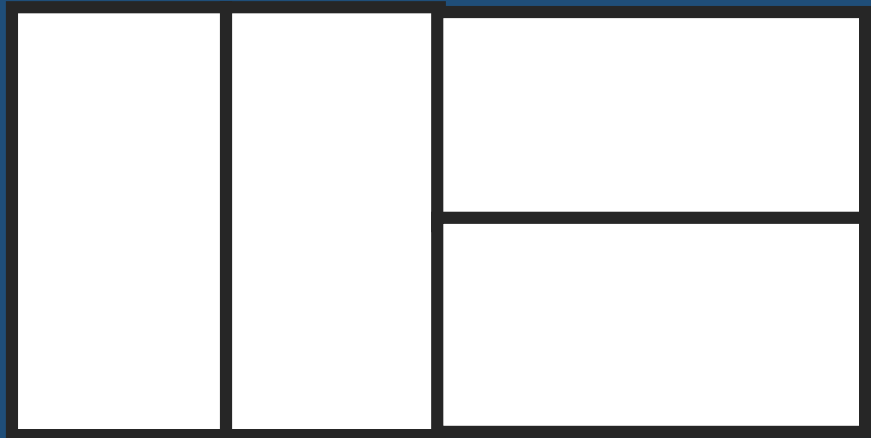
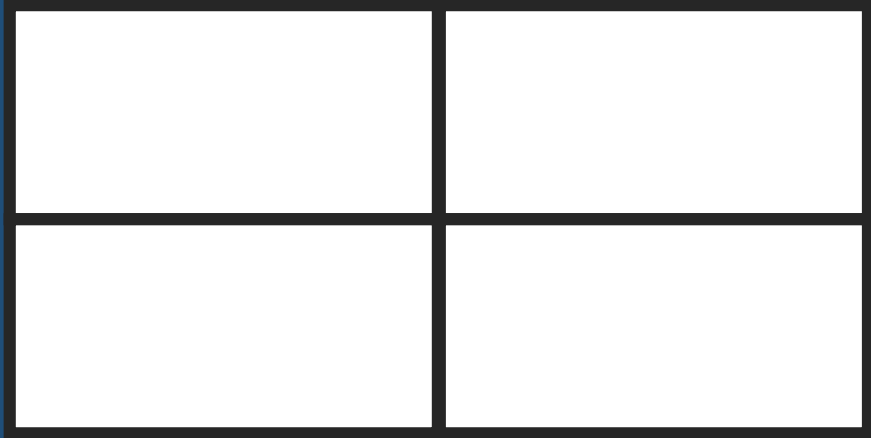


$d[2] = 2$

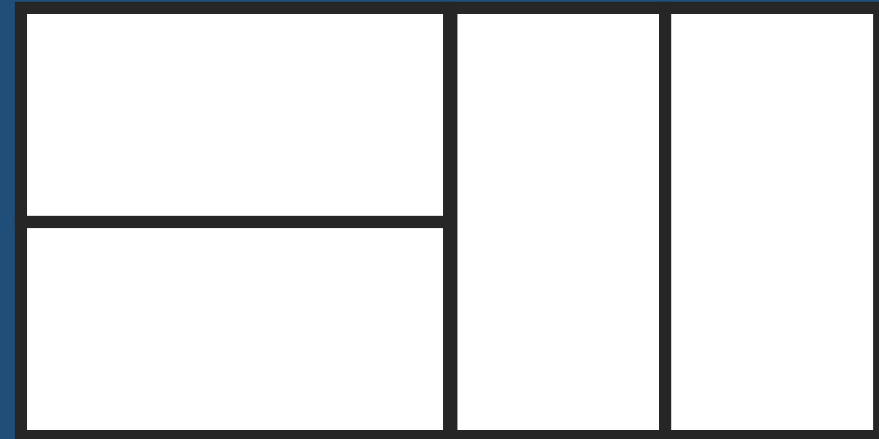


$d[3] = 3$

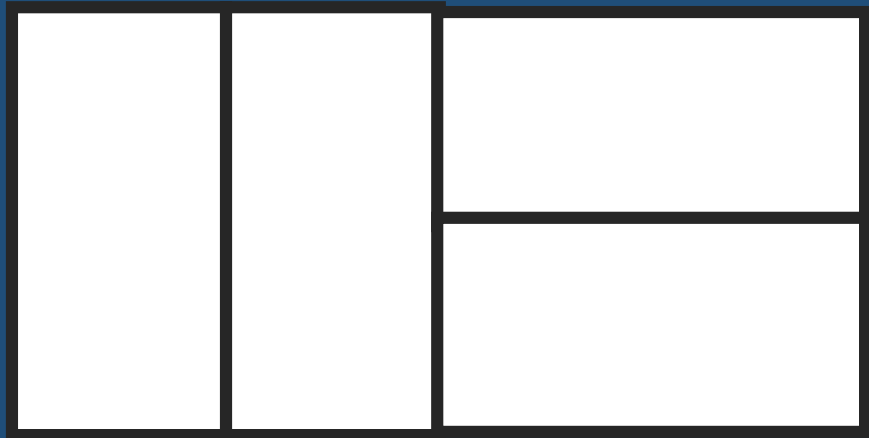
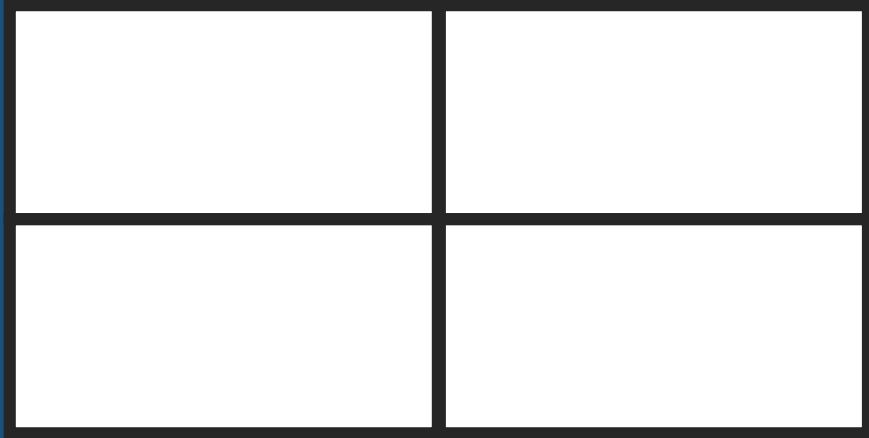
11726번 풀이



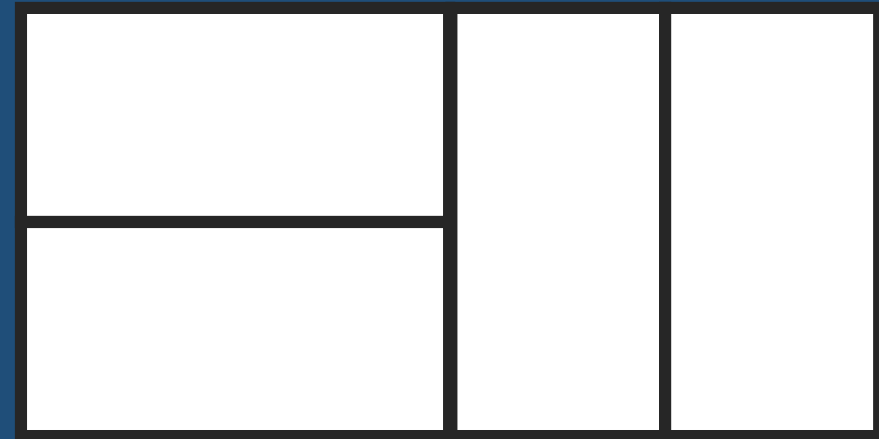
$d[4] = 5$



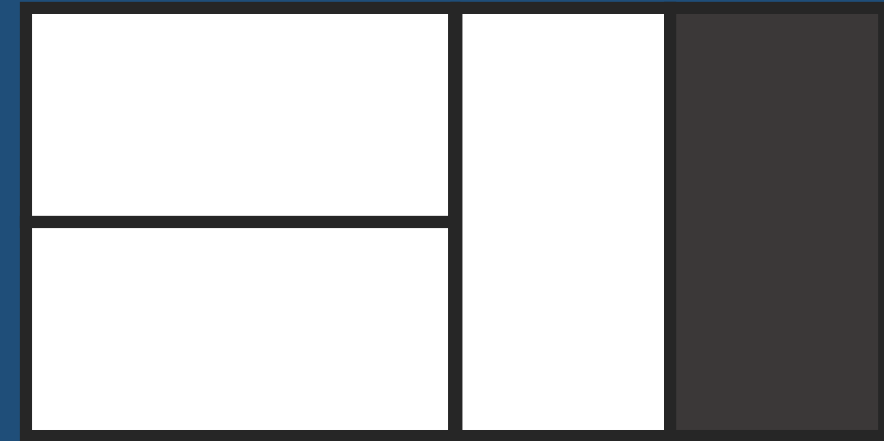
11726번 풀이



혹시 규칙을 눈치 채셨나요?



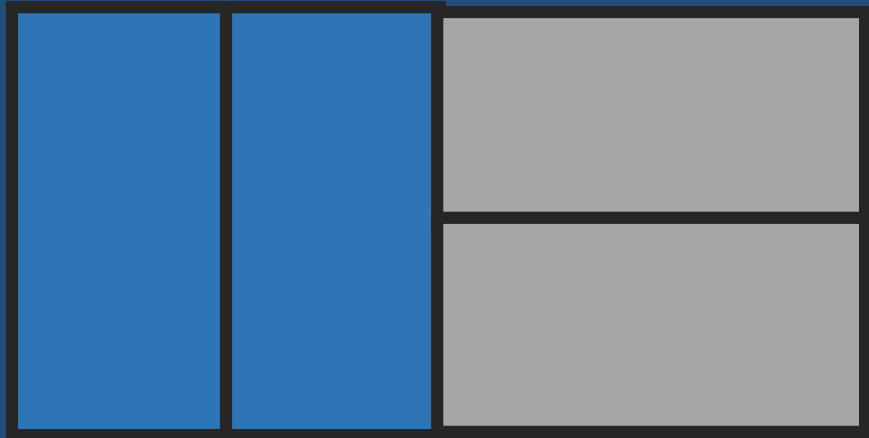
11726번 풀이



맨 오른쪽 도형은 반드시 다음 두가지 유형입니다!



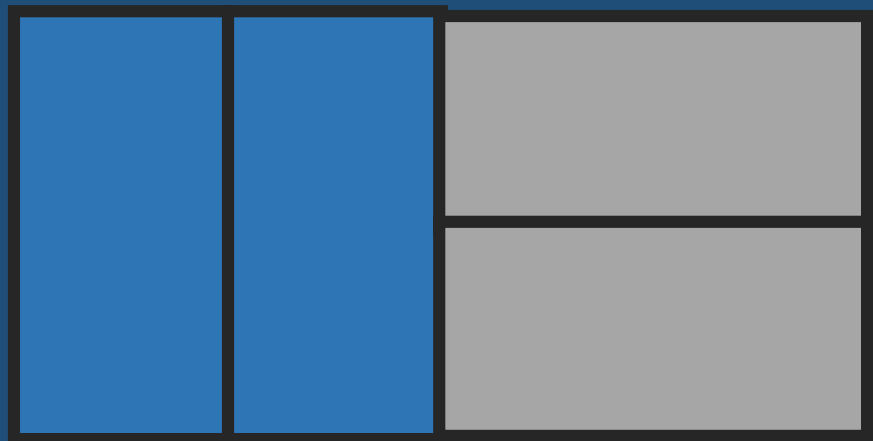
11726번 풀이



그렇다면 맨 오른쪽을 제외한 도형들은  
작은  $2 \times N$  타일 문제로 생각할 수 있습니다!



11726번 풀이



$\Rightarrow d[N-2]$



$\Rightarrow d[N-1]$

## 3단계 점화식을 세워보자

그렇다면 점화식은 어떻게 될까요?

## 3단계 점화식을 세워보자

$$d[n] = d[n-1] + d[n-2]$$

$$d[1] = 1, d[2] = 2$$

#어디선가 많이 본 점화식이지 않나요?

#풀이 코드는 생략하겠습니다!

# 검증

자신이 푼 알고리즘의 시간 복잡도를 직접 구해봅시다!  
주어진 시간 안에 문제를 풀이 할 수 있나요?



## 주의

### 출력

첫째 줄에  $2 \times n$  크기의 직사각형을 채우는 방법의 수를 10,007로 나눈 나머지를 출력한다.

출력에서 다음과 같이 N으로 나눈 나머지를 출력하라는 유형이 많이 있습니다.  
대부분 문제의 정답의 숫자가 매우 커 **오버 플로우**가 발생할 때 조건으로 붙습니다.



## 주의

```
for(int i=3;i<=data;i++){  
    d[i] = d[i-1] + d[i-2];  
    d[i] %= 10007;  
}
```

이때, 결과를 구하고 마지막에 나눌 필요 없이,  
구하는 중간에 계속 나눠줘도 상관 없습니다!

가능한 이유는 다음 문제를 참고해주세요!

<https://www.acmicpc.net/problem/10430>

# 정리

이번 시간에는  
**복잡도와 다이나믹 프로그래밍**에 대해 알아보았습니다!  
두 개념 모두 PS에서 매우 중요하므로 반드시 알아 두시기 바랍니다!

기초적인 다이나믹 프로그래밍 문제에서 중요한 것은,

**점화식**을 찾는 일입니다!

또한 **복잡도**를 계산하여

적절한 알고리즘인지 판단하는 연습도 중요합니다!

이번 과제를 통해서 숙달해 보시기 바랍니다!



# The End

수고하셨습니다!

다음 시간에는  
**수학**에 대해 알아보시다!