



3주차 알고리즘 세미나

PoolC 2021 1학기

● ● 지난시간 도전문제 – 감시(삼성 SW테스트 기출)

문제 요약 :

- $1 \leq N, M \leq 8$, 0은 빈 칸, 1~5는 cctv, 6은 벽, cctv의 최대 개수 ≤ 8
- cctv는 타입에 따라서 볼 수 있는 방향이 다르다
- cctv의 방향을 바꿔서 시야를 조정할 수 있다.
- cctv는 벽을 통과 할 수 없으나 다른 cctv는 통과 가능하다.
- cctv의 방향을 어떻게 설정해야 최대한 많은 곳을 볼 수 있을까?

고려해야 할 사항 :

- 시간복잡도?
- cctv의 정보 저장
- cctv가 보는 곳 탐색
- cctv가 볼 수 있는 범위 처리

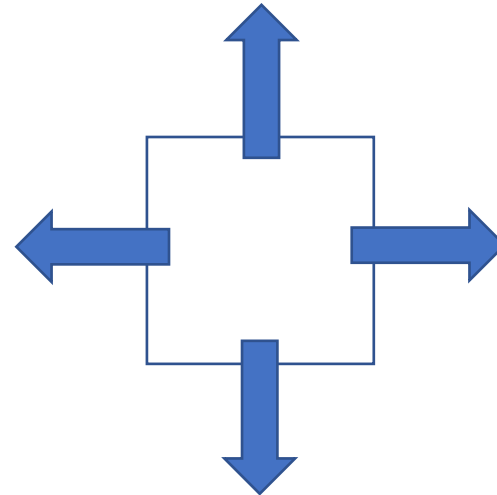
● ● 지난시간 도전문제 – 감시(삼성 SW테스트 기출)

- 시간복잡도

- $1 \leq N, M \leq 8$
- cctv의 최대 개수 : 8
- cctv가 가질 수 있는 방향 : 4
- 전체 방향 탐색 : $N+M$

$$4^8 \times (N+M) = 2^{20}$$

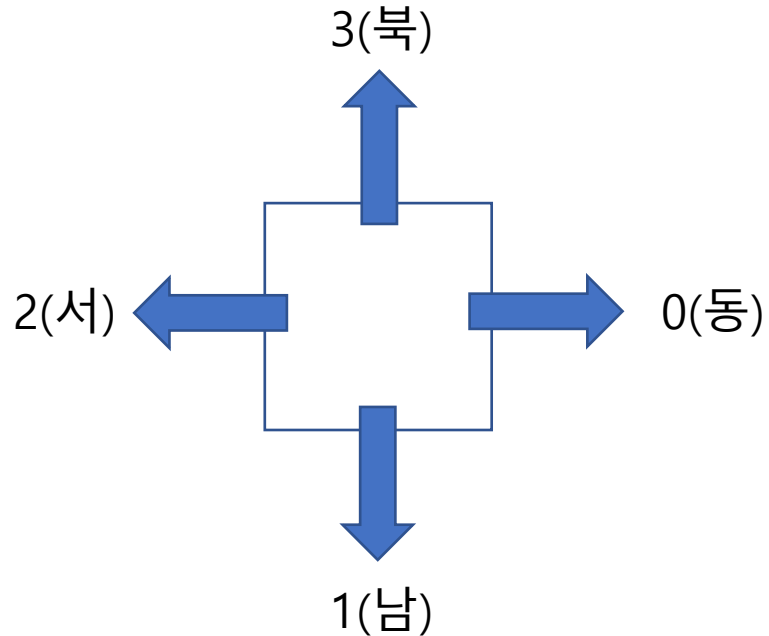
8개의 cctv에 대해서 각각 4개의 방향을 전부 고려



● ● 지난시간 도전문제 - 감시(삼성 SW테스트 기출)

- cctv의 정보 저장

- y좌표, x좌표, cctv의 종류, 현재 보고 있는 방향
dir



dir의 default는
0으로 임의 설정

info 구조체

- int y, x, type, dir

vector<info> cctv

- cctv들의 정보를 담을 vector

- 코드에서 방향 표시를 위해 임의로 숫자 부여

- 방향 전환 구현?

- $dir = (dir + 1) \% 4$

- 시계방향으로 90도 회전

● ● 지난시간 도전문제 - 감시(삼성 SW테스트 기출)

- cctv의 정보 저장

- y좌표, x좌표, cctv의 종류, 현재 보고 있는 방향



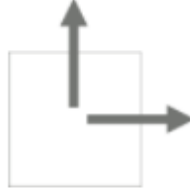
1번

{0}



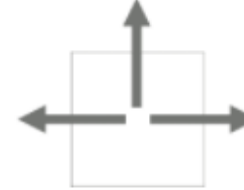
2번

{0,2}



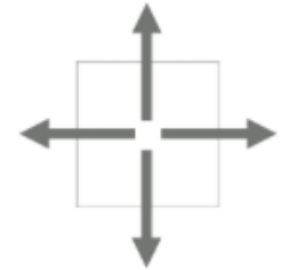
3번

{0,3}



4번

{0,2,3}



5번

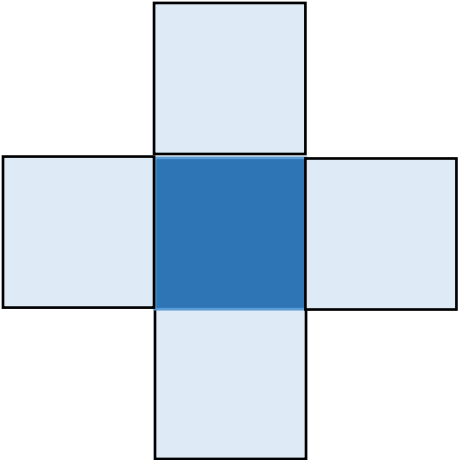
{0,1,2,3}

```
vector<int> cctv_dir[6] = { { }, {0}, {0,2}, {0,3}, {0,2,3}, {0,1,2,3} }
```

지난시간 도전문제 - 감시(삼성 SW테스트 기출)

- cctv가 보는 곳 탐색

- 현재 칸을 기준으로 상하좌우 4방향 탐색, dy, dx로 해결



```
if (board[y][x + 1] == 1)
    return true;
else if (board[y + 1][x] == 1)
    return true;
else if (board[y][x - 1] == 1)
    return true;
else if (board[y - 1][x] == 1)
    return true;
```

VS

```
int dx[4] = { 1, 0, -1, 0 };
int dy[4] = { 0, 1, 0, -1 };
for (int i = 0; i < 4; i++)
{
    if (board[y + dy[i]][x + dx[i]] == 1)
        return true;
}
```

코드의 간결성 UP

(y+dy[0], x+dx[0]) -> (y, x+1)

(y+dy[1], x+dx[1]) -> (y+1, x)

(y+dy[2], x+dx[2]) -> (y, x-1)

(y+dy[3], x+dx[3]) -> (y-1, x)

● ● 지난시간 도전문제 - 감시(삼성 SW테스트 기출)

- cctv가 보는 곳 탐색

- 현재 방향에서 cctv가 볼 수 있는 방향만 탐색

- 벽을 만나거나, 범위를 벗어나면 탐색 종료

- `vector<int> cctv_dir[6] = { {}, {0}, {0,2}, {0,3}, {0,2,3}, {0,1,2,3} }`

- `const int dx[4] = {1,0,-1,0}`

- `const int dy[4] = {0,1,0,-1}`

```
0 0 0 0 0 #
# 2 # # # #
0 0 0 0 6 #
0 6 # # 2 #
0 0 0 0 0 #
# # # # # 5
```

왼쪽 상단 2: ↔, 오른쪽 하단 2: ↔

[cctv가 보는곳을 탐색하는 함수]

```
for(int i=0; i<cctv_dir[type].size(); i++)
```

```
while(true)
```

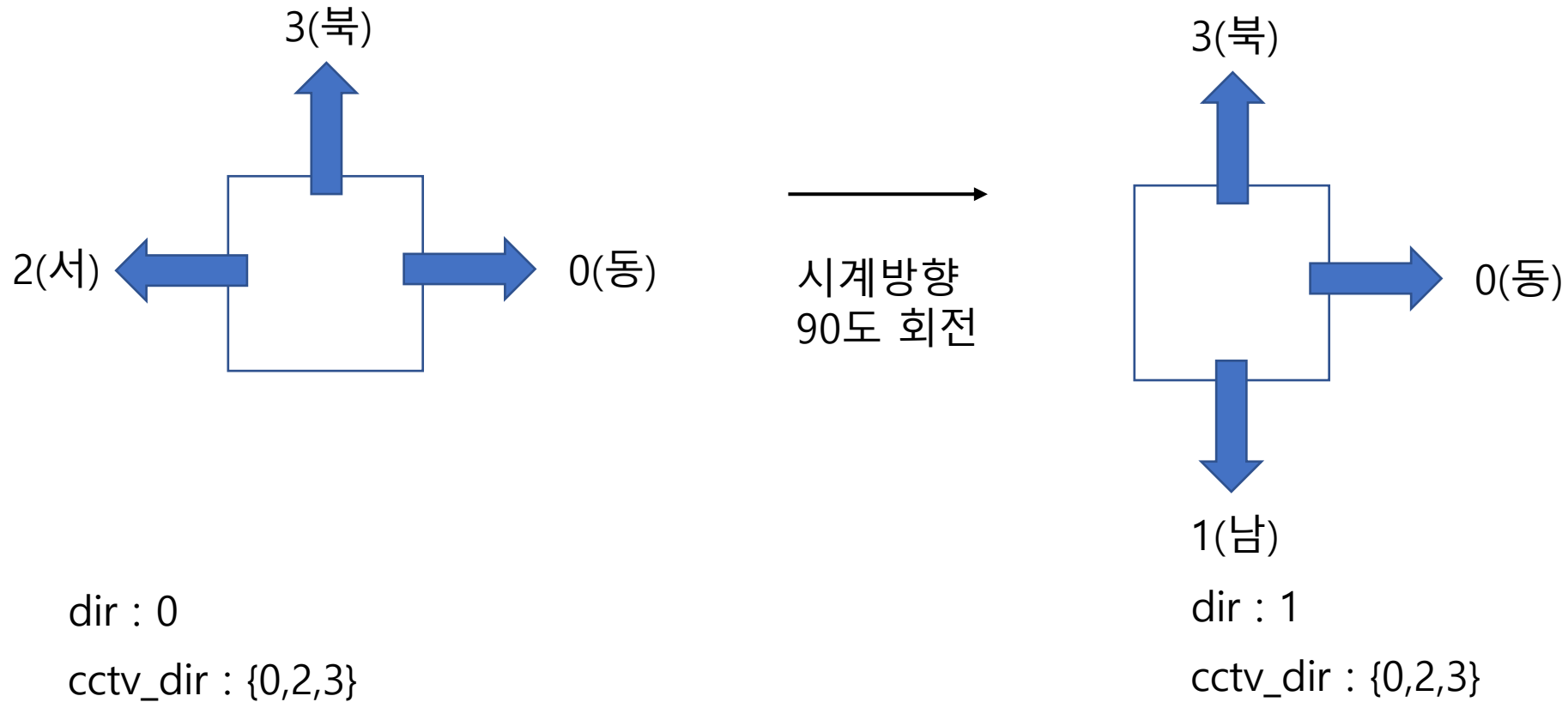
```
    y += dy[(cctv_dir[type][i] + dir) % 4]
```

```
    x += dx[(cctv_dir[type][i] + dir) % 4]
```

벽을 만나는지, 범위를 벗어나는지?

계속 방문 처리

● ● 지난시간 도전문제 - 감시(삼성 SW테스트 기출)



- 실제로 cctv_dir에 접근해서 값을 수정할 수 있지만 코드의 간결성을 위해
- 얼마나 회전했는지 나타내는 변수 dir을 더해서 실제 cctv를 회전한 것처럼 구현

● ● 지난시간 도전문제 - 감시(삼성 SW테스트 기출)

```
for(int i=0; i<cctv_dir[type].size(); i++)  
    while(true)  
        y += dy[(cctv_dir[type][i] + dir) % 4]  
        x += dx[(cctv_dir[type][i] + dir) % 4]
```

벽을 만나는지, 범위를 벗어나는지?

계속 방문 처리

cctv_dir[type] : 종류가 type인 cctv가
볼 수 있는 방향 vector

cctv_dir[type][i] : 종류가 type인 cctv가
볼 수 있는 방향 vector의 i번째 값

dir : 현재 cctv가 바라보는 방향
(처음 상태에서 얼마나 회전했는지)

(cctv_dir[type][i] + dir) : cctv가 볼 수 있는 방향 중 하나에 dir를 더해서

실제로 dir만큼 회전한 효과를 주기 위한것.

대신 방향은 0~3까지밖에 없으므로 % 4 처리

● ● 지난시간 도전문제 – 감시(삼성 SW테스트 기출)

- vector<int> cctv_dir[6] = { { }, {0}, {0,2}, {0,3}, {0,2,3}, {0,1,2,3} }
- vector<info> cctv
- const int dx[4] = {1,0,-1,0}
- const int dy[4] = {0,1,0,-1}
- const int INF = 987,654,321
- int MIN = INF
- int board[8][8]

```
void back_tracking (int index)
```

종료 조건(index == cctv.size())

최소값 갱신

```
for(int i=0; i<4; i++)
```

현재 cctv 방향에 대한 방문 처리

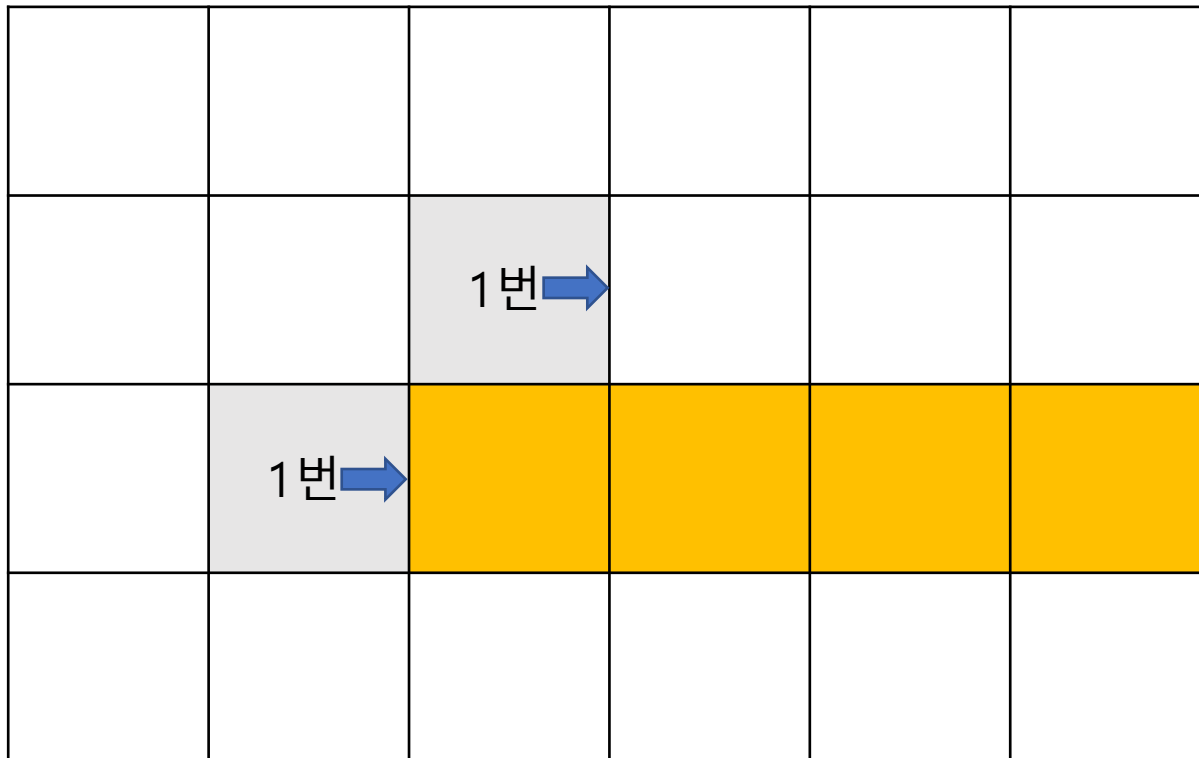
```
back_tracking(index+1)
```

현재 방향으로 방문했던 곳 다시 미방문 처리

시계방향으로 90도 방향 전환

● ● 지난시간 도전문제 - 감시(삼성 SW테스트 기출)

- 현재 cctv가 보는 방향 전부 방문 처리



```
for(int i=0; i<cctv_dir[type].size(); i++)
```

```
while(true)
```

```
    y += dy[(cctv_dir[type][i] + dir) % 4]
```

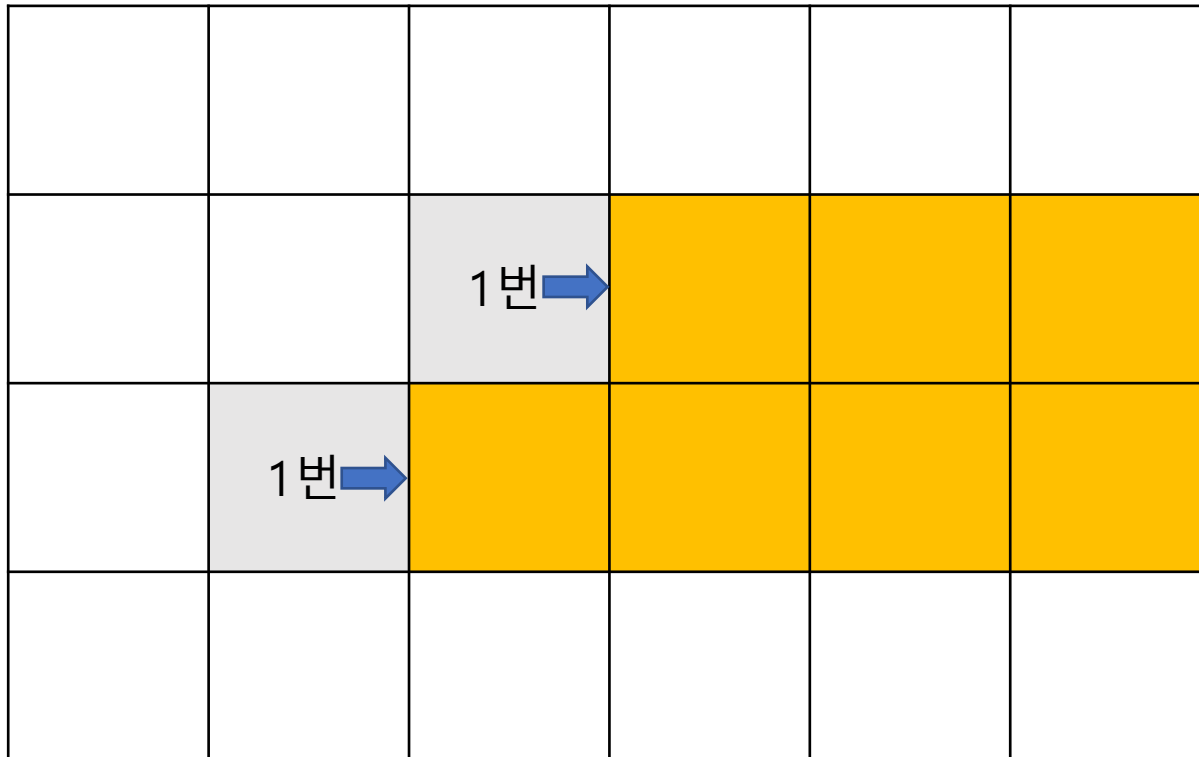
```
    x += dx[(cctv_dir[type][i] + dir) % 4]
```

벽을 만나는지, 범위를 벗어나는지?

계속 방문 처리

● ● 지난시간 도전문제 - 감시(삼성 SW테스트 기출)

- 현재 cctv가 보는 방향 전부 방문 처리



```
type = 1,    dir = 0 ,    i = 0
```

```
while(true)
```

```
    y += dy[(0 + 0) % 4]
```

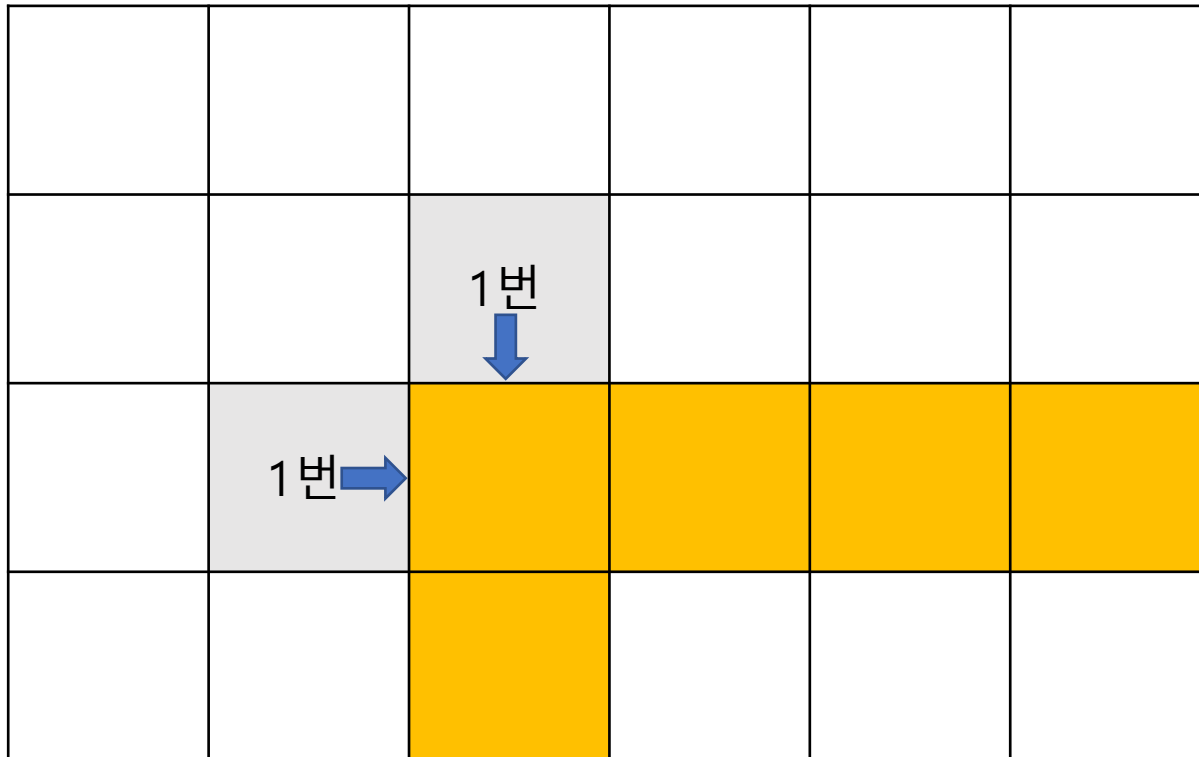
```
    x += dx[(0 + 0) % 4]
```

벽을 만나는지, 범위를 벗어나는지?

계속 방문 처리

● ● 지난시간 도전문제 - 감시(삼성 SW테스트 기출)

- 현재 cctv가 보는 방향 전부 방문 처리



```
type = 1,    dir = 1 ,    i = 0
```

```
while(true)
```

```
    y += dy[(0 + 1) % 4]
```

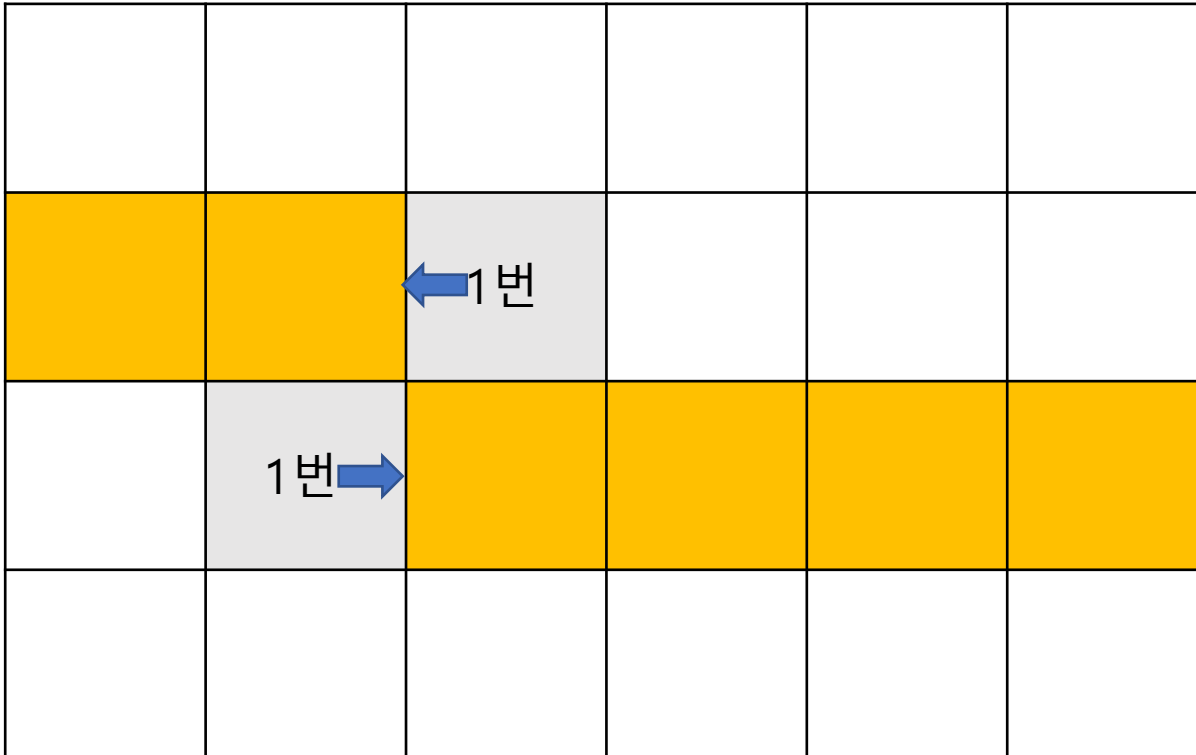
```
    x += dx[(0 + 1) % 4]
```

벽을 만나는지, 범위를 벗어나는지?

계속 방문 처리

● ● 지난시간 도전문제 - 감시(삼성 SW테스트 기출)

- 현재 cctv가 보는 방향 전부 방문 처리



```
type = 1,    dir = 2 ,    i = 0
```

```
while(true)
```

```
    y += dy[(0 + 2) % 4]
```

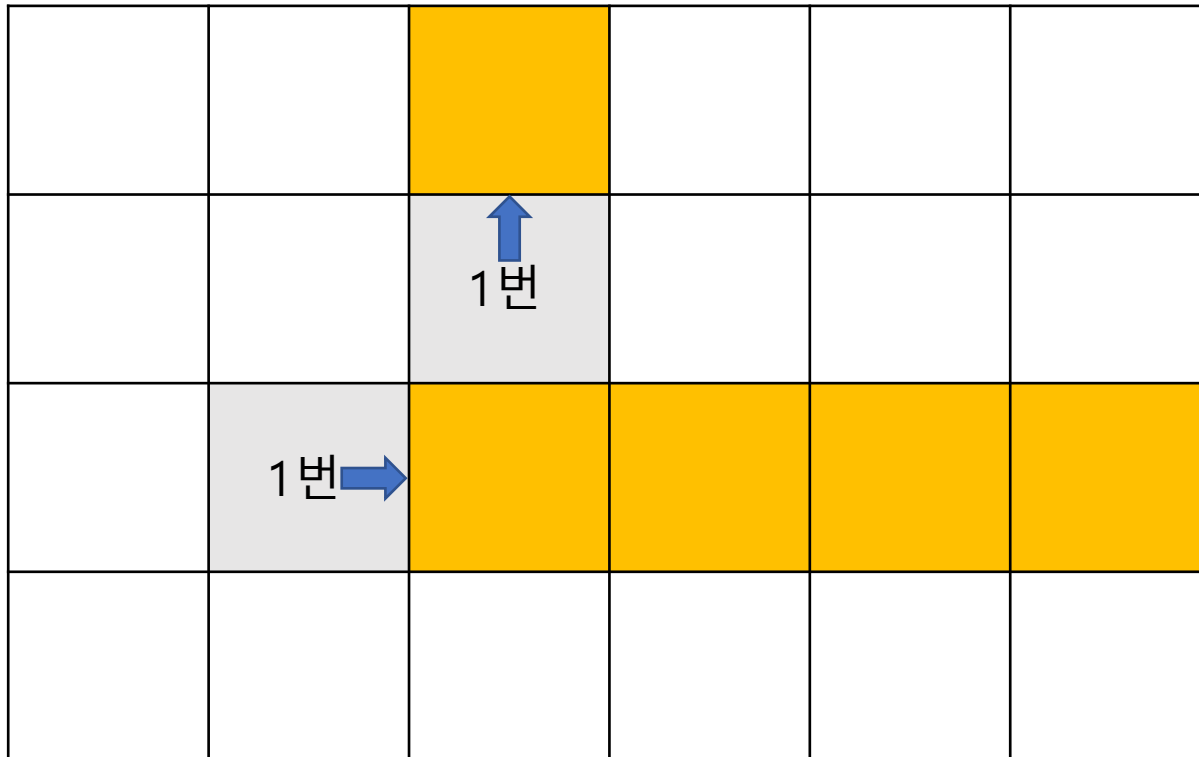
```
    x += dx[(0 + 2) % 4]
```

벽을 만나는지, 범위를 벗어나는지?

계속 방문 처리

● ● 지난시간 도전문제 - 감시(삼성 SW테스트 기출)

- 현재 cctv가 보는 방향 전부 방문 처리



```
type = 1,    dir = 3 ,    i = 0
```

```
while(true)
```

```
    y += dy[(0 + 3) % 4]
```

```
    x += dx[(0 + 3) % 4]
```

벽을 만나는지, 범위를 벗어나는지?

계속 방문 처리

● ● 지난시간 도전문제 - 감시(삼성 SW테스트 기출)

- 현재 cctv가 보는 방향 전부 방문 처리

벽을 만나는지, 범위를 벗어나는지?

1. 벽을 만난다 -> `board[y][x] == 6`
2. 범위를 벗어난다 -> **!inRange**
3. 둘중 하나라도 true면 break;

```
bool inRange(int y, int x)
    if(x<0 || y<0 || x>M-1 || y>N-1)
        return false
    return true
```

2차원 배열 탐색할때 정말 많이 쓰이는 함수

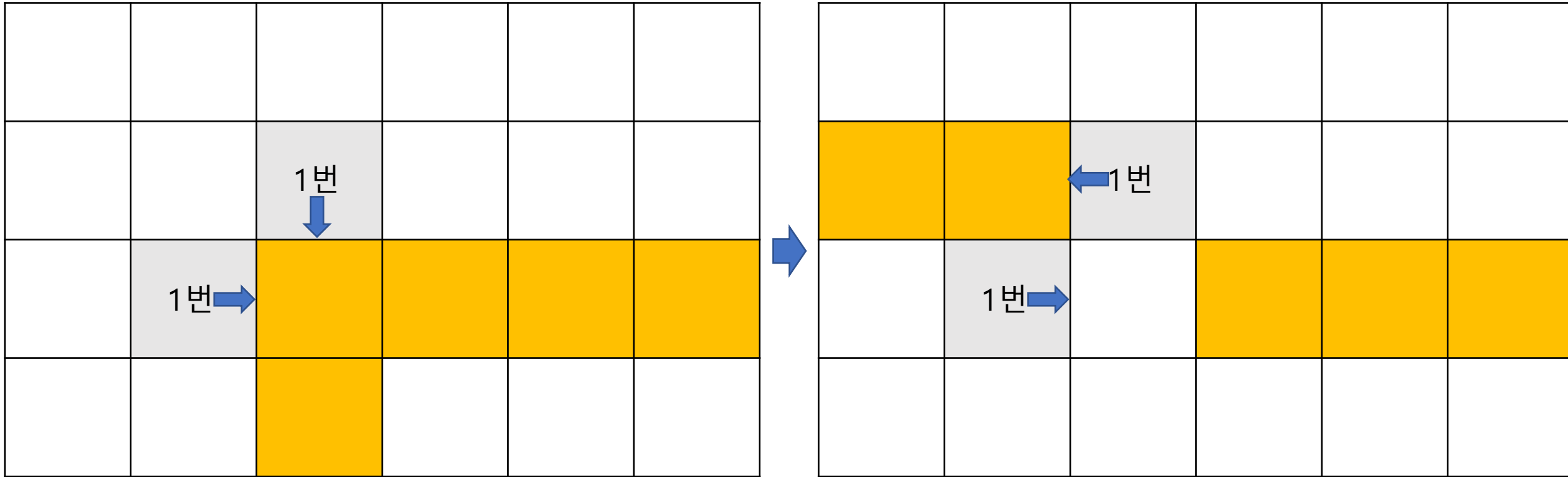
● ● 지난시간 도전문제 – 감시(삼성 SW테스트 기출)

- `vector<int> cctv_dir[6] = { { }, {0}, {0,2}, {0,3}, {0,2,3}, {0,1,2,3} }`
- `vector<info> cctv`
- `const int dx[4] = {1,0,-1,0}`
- `const int dy[4] = {0,1,0,-1}`
- `const int INF = 987,654,321`
- `int MIN = INF`
- `int board[8][8]`
- `bool checked[8][8]`
 - `checked[y][x]`가 true면 해당 칸은 cctv가 감시중

`checked[8][8]`을 bool로 선언해도 될까?

● ● 지난시간 도전문제 - 감시(삼성 SW테스트 기출)

- 현재 cctv가 보는 방향 전부 방문 처리



자기가 봤던 곳에 checked인 곳을 전부 !checked로 만들어준다

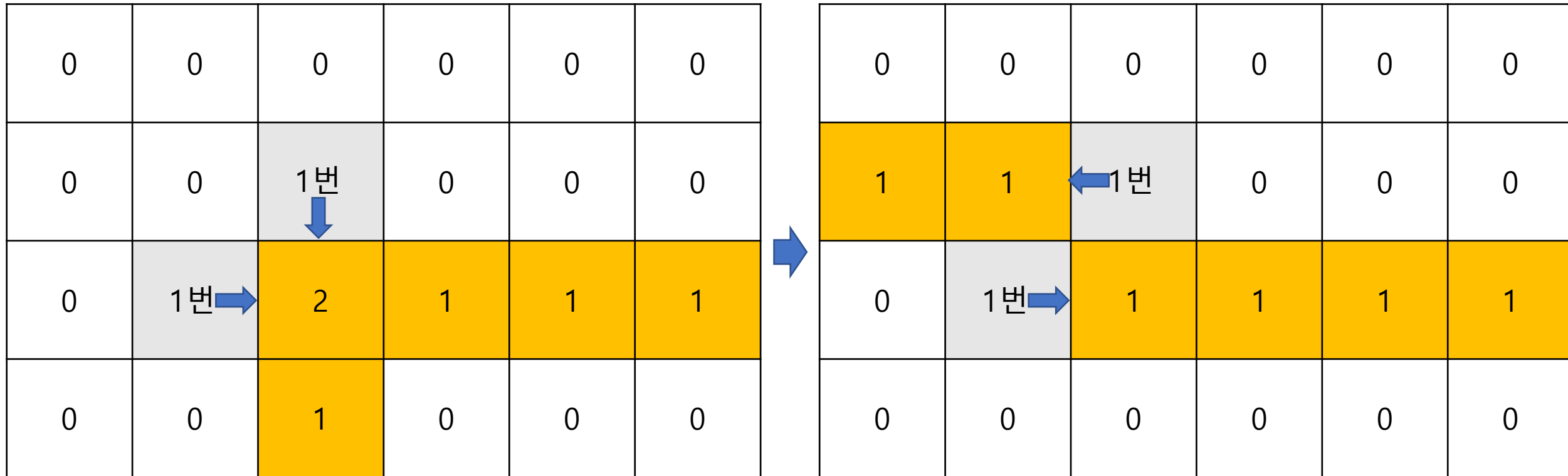
● ● 지난시간 도전문제 – 감시(삼성 SW테스트 기출)

- `vector<int> cctv_dir[6] = { { }, {0}, {0,2}, {0,3}, {0,2,3}, {0,1,2,3} }`
- `vector<info> cctv`
- `const int dx[4] = {1,0,-1,0}`
- `const int dy[4] = {0,1,0,-1}`
- `const int INF = 987,654,321`
- `int MIN = INF`
- `int board[8][8]`
- `int checked[8][8]`
 - `checked[y][x] > 0`이면, 해당 칸은 cctv가 감시중

`checked[8][8]`을 `int`로 선언하면?

● ● 지난시간 도전문제 - 감시(삼성 SW테스트 기출)

- 현재 cctv가 보는 방향 전부 방문 처리



자기가 봤던 곳에 checked>0인 곳을 전부 checked-- 해준다

● ● 지난시간 도전문제 - 감시(삼성 SW테스트 기출)

- 함수의 재사용

```
void back_tracking (int index)
```

```
    종료 조건(index == cctv.size())
```

```
        최소값 갱신
```

```
    for(int i=0; i<4; i++)
```

```
        현재 cctv 방향에 대한 방문 처리
```

```
        back_tracking(index+1)
```

```
        현재 방향으로 방문했던 곳 다시 미방문 처리
```

```
        시계방향으로 90도 방향 전환
```

```
void see(int index, bool isLight)
```

```
    for(int i=0; i<cctv_dir[type].size(); i++)
```

```
        while(true)
```

```
            y += dy[(cctv_dir[type][i] + dir) % 4]
```

```
            x += dx[(cctv_dir[type][i] + dir) % 4]
```

```
                벽을 만나는지, 범위를 벗어나는지?
```

```
            if(isLight)
```

```
                해당 칸 방문 처리
```

```
            else
```

```
                해당 칸 미방문 처리
```

이 문제를 통해 배웠던 점

- 4방향 처리를 임의의 숫자를 부여해 간편하게 할 수 있다. {0, 1, 2, 3}
- 각 방향마다 탐색을 dy, dx로 간편하게 할 수 있다. int dx[4] = {1,0,-1,0}
int dy[4] = {0,1,0,-1}
- 어떻게 배치해야 사각지대가 최소가 되는지 모르니 전부 다 해보기
- 현재 칸이 문제에서 주어진 범위 안에 있는지 함수로 판단하기
 - bool inRange(int y, int x)
- 방문 처리를 단순히 bool형 자료형이 아닌 int형으로 해서 간편하게 중복 처리

$$(y+dy[0], x+dx[0]) \rightarrow (y, x+1)$$
$$(y+dy[1], x+dx[1]) \rightarrow (y+1, x)$$
$$(y+dy[2], x+dx[2]) \rightarrow (y, x-1)$$
$$(y+dy[3], x+dx[3]) \rightarrow (y-1, x)$$

문제 요약 :

- $N \times N$ 행렬이 주어진다. $2 \leq N \leq 50$, $1 \leq M \leq 13$, 0이면 빈칸, 1이면 집, 2이면 치킨집
- 최대 M개의 치킨집을 골라서 도시의 치킨거리의 최소값을 구하자
- 치킨거리는 어떤 집에서 가장 가까운 치킨집까지의 거리
- 도시의 치킨거리는 모든 집의 치킨거리를 다 더한 값
- 치킨거리는 $|r_1 - r_2| + |c_1 - c_2|$ 로 구한다.

고려해야 할 사항 :

- 시간복잡도?
- 최대 M개?
- 집들의 좌표와 치킨집들의 좌표를 저장할 변수 필요

● ● 지난시간 도전문제 – 치킨배달

- 시간복잡도?

- 13개의 치킨집 중에서 M개를 선택. ${}_{13}C_M$

- 2N개의 집에서 각각 M개의 치킨집과 거리 비교 $2N \times M$

$${}_{13}C_M \times 2N \times M$$

- 최대 M개?

- 당연히 치킨집이 최대로 있어야 치킨거리가 최소화 될 가능성이 높다.

- M-1개의 치킨집으로 계산한 치킨거리에서, 거기에 1개의 치킨집을 추가한다 해도 치킨거리는 늘어나지 않는다.

- 따라서 최대 M개가 아닌 그냥 M개를 살려야 한다.

지난시간 도전문제 – 치킨배달

- 좌표 저장 변수?
 - `vector<pair<int, int> > house_pos`
 - `vector<pair<int, int> > chicken_pos`

vector	(1,4)	(2,0)	(3,7)	(6,4)	(6,7)	(0,5)	...
--------	-------	-------	-------	-------	-------	-------	-----

- N과 M이랑 비슷한 문제
 - 13개의 치킨집 중에서 중복없이 오름차순으로 선택하기
 - 만약 오름차순이 아니라면?
 - 13!의 경우의수
 - 치킨집을 뽑는데 순서는 중요하지 않다.

- 치킨거리를 실제로 구하자

집의 좌표 vector	(1,4)	(2,0)	(3,7)	(6,4)	(6,7)	(0,5)	...
치킨집의 좌표 vector	(3,2)	(4,5)	(3,3)	(1,1)	(6,3)	(0,4)	...

- 치킨거리는 $|r_1 - r_2| + |c_1 - c_2|$ 로 구한다.

2N x M번의 연산으로 최소값을 구할 수 있다.

지난시간 도전문제 - 치킨배달

- 오름차순으로 치킨집을 선택하기

```
void pick (int index)
```

```
    종료 조건(list.size == N)
```

```
        최소값 갱신
```

```
    for(int i=index; i<cnt; i++)
```

```
        치킨집 리스트에 i번째 추가
```

```
        pick (i+1)
```

```
        치킨집 리스트에서 i번째 제거
```

- 문제에서 중요했던 점

- 치킨집을 M개 뽑을 때 꼭 오름차순으로 중복없이 뽑아야 했던 것

- 치킨거리를 계산할 때 그냥 좌표끼리 비교만 해주면 됐다

문제 요약 :

- 10x10 크기의 행렬이 주어진다. 0과 1로 이루어져 있다.
- 크기가 1~5인 색종이가 각각 5개씩 있다.
- 1을 색종이로 덮어야 한다. 이때 색종이는 경계 안에 있어야 하고, 겹쳐서도 안되고 0을 덮으면 안된다.
- 색종이의 개수를 최소로 하고 싶다.

고려해야 할 사항 :

- 어떤 색종이를 붙여야 최소가 되는지 알 수 없으니 모든 색종이를 다 붙여본다. (완전탐색)
- 1의 위치들을 저장할 변수가 필요하다!
- 현재 각 색종이를 몇개씩 썼는지 저장할 변수가 필요하다!

● ● 지난시간 도전문제 - 색종이 붙이기

- 1의 위치들을 저장할 변수가 필요하다!
 - 왜? 0은 색종이가 있으면 안되고 문제에서 주어진 모든 1을 색종이로 덮어야 하므로
 - 불필요한 0까지 탐색하기 보단 1만 탐색하기
 - 1의 좌표들로 이루어진 vector에서 백트래킹하며 완전탐색
- 현재 각 색종이를 몇개씩 썼는지 저장할 변수가 필요하다!
 - 왜? 각 색종이별로 최대 5개까지만 쓸 수 있으므로
 - 만약 이미 5개를 쓴 상태에서 추가로 더 필요하다면 전의 색종이를 되돌리는 백트래킹으로 완전탐색

지난시간 도전문제 - 색종이 붙이기

- int board[10][10]
- int cnt[5] // cnt[0] : 길이가 1인 색종이를 사용한 횟수
- bool covered[10][10]
- const int INF = 987,654,321
- int MIN = INF
- int N = 10
- int M = 10

void pick (int index)

종료 조건(index == one.size())

최소값 갱신

현재 index번째 1의 좌표가 이미 cover되었는지?

pick (index+1)

for(int i=0; i<5; i++)

여기에 길이가 i+1인 색종이를 덮을수 있는지?

현재 index번째 1의 좌표를 기준으로 색종이를 덮는다

cnt[i]++

pick (index+1)

cnt[i]--

덮은 색종이를 치운다.

지난시간 도전문제 - 색종이 붙이기

현재 index번째 1의 좌표가 이미 cover되었는지?

pick (index+1)

- 왜 필요할까?

현재 1에서는 어떠한
색종이도 덮을 수 없다.

따라서 pick(index+1)을 호출하지
못하고 그대로 종료

1	1	0	0	0
1	1	0	0	0
0	0	0	0	0
0	0	1	1	0
0	0	1	0	0



1	1	0	0	0
1	1	0	0	0
0	0	0	0	0
0	0	1	1	0
0	0	1	0	0

for(int i=0; i<5; i++)

여기에 길이가 i+1인 색종이를 덮을수 있는지?

현재 index번째 1의 좌표를 기준으로 색종이를 덮는다

cnt[i]++

pick (index+1)

cnt[i]--

덮은 색종이를 치운다.

지난시간 도전문제 - 색종이 붙이기

현재 index번째 1의 좌표가 이미 cover되었는지?

pick (index+1)

1	1	0	0	0
1	1	0	0	0
0	0	0	0	0
0	0	1	1	0
0	0	1	0	0



1	1	0	0	0
1	1	0	0	0
0	0	0	0	0
0	0	1	1	0
0	0	1	0	0



1	1	0	0	0
1	1	0	0	0
0	0	0	0	0
0	0	1	1	0
0	0	1	0	0



1	1	0	0	0
1	1	0	0	0
0	0	0	0	0
0	0	1	1	0
0	0	1	0	0



1	1	0	0	0
1	1	0	0	0
0	0	0	0	0
0	0	1	1	0
0	0	1	0	0

현재 index번째 1의 좌표가 이미 cover되었는지?

pick (index+1)

return

for(int i=0; i<5; i++)

여기에 길이가 i+1인 색종이를 덮을수 있는지?

현재 index번째 1의 좌표를 기준으로 색종이를 덮는다

cnt[i]++

pick (index+1)

cnt[i]--

덮은 색종이를 치운다.

지난시간 도전문제 - 색종이 붙이기

여기에 길이가 i+1인 색종이를 덮을수 있는지?

- canCover

- board에서 벗어나는지?
- 길이가 i+1인 색종이를 더 쓸 수 있는지?
- 길이가 i+1인 색종이를 덮을 때 0이 있는지?
- 이미 색종이가 덮인 곳이 있는지?

```
bool canCover(int y, int x, int length)
```

```
( !inRange ) || ( cnt[length-1] >= 5 )
```

```
    return false
```

```
    for(int i=y; i<y+length; i++)
```

```
        for(int j=x; j<x+length; j++)
```

```
            ( board[i][j] == 0 ) || ( covered[i][j] )
```

```
                return false
```

```
    return true
```

지난시간 도전문제 - 색종이 붙이기

현재 index번째 1의 좌표를 기준으로 색종이를 덮는다

덮은 색종이를 치운다.

- Cover

- 덮을때도, 치울때도 같은 로직으로 작동한다.
- bool flag로 함수의 재사용

```
void Cover(int y, int x, int length, bool flag)
    for(int i=y; i<y+length; i++)
        for(int j=x; j<x+length; j++)
            covered[i][j] = flag
```

지난시간 도전문제 - 색종이 붙이기

현재 index번째 1의 좌표가 이미 cover되었는지?

pick (index+1)

return

for(int i=0; i<5; i++)

여기에 길이가 i+1인 색종이를 덮을수 있는지?

현재 index번째 1의 좌표를 기준으로 색종이를 덮는다

cnt[i]++

pick (index+1)

cnt[i]--

덮은 색종이를 치운다.

bool canCover(int y, int x, int length)

(!inRange) || (cnt[length-1] >= 5)

return false

for(int i=y; i<y+length; i++)

for(int j=x; j<x+length; j++)

(board[i][j] == 0) || (covered[i][j])

return false

return true

void Cover(int y, int x, int length, bool flag)

for(int i=y; i<y+length; i++)

for(int j=x; j<x+length; j++)

covered[i][j] = flag

● ● 지난시간 도전문제 - 색종이 붙이기

- 문제에서 중요했던 점
 - 무턱대고 10x10 배열을 탐색하지 말고 1의 좌표만 따로 저장한 다음 탐색하기
 - 현재 칸이 **covered**라면 색종이를 덮어볼 필요도 없이 다음 index로 넘어가기
 - 길이가 $i+1$ 인 색종이를 덮을 수 있는지 판단하기
 - 색종이를 덮고, 제거하는 과정이 동일하다는 것을 깨닫고 함수 재사용 하기

어려웠던 문제, 하지만 백트래킹의 기본 틀은 변하지 않는다.

다이나믹 프로그래밍이란? a.k.a 동적 계획법

- 완전탐색의 단점을 보완하는 방법
- 메모이제이션 : cache라는 메모리를 두어 계산의 중복을 막아준다.

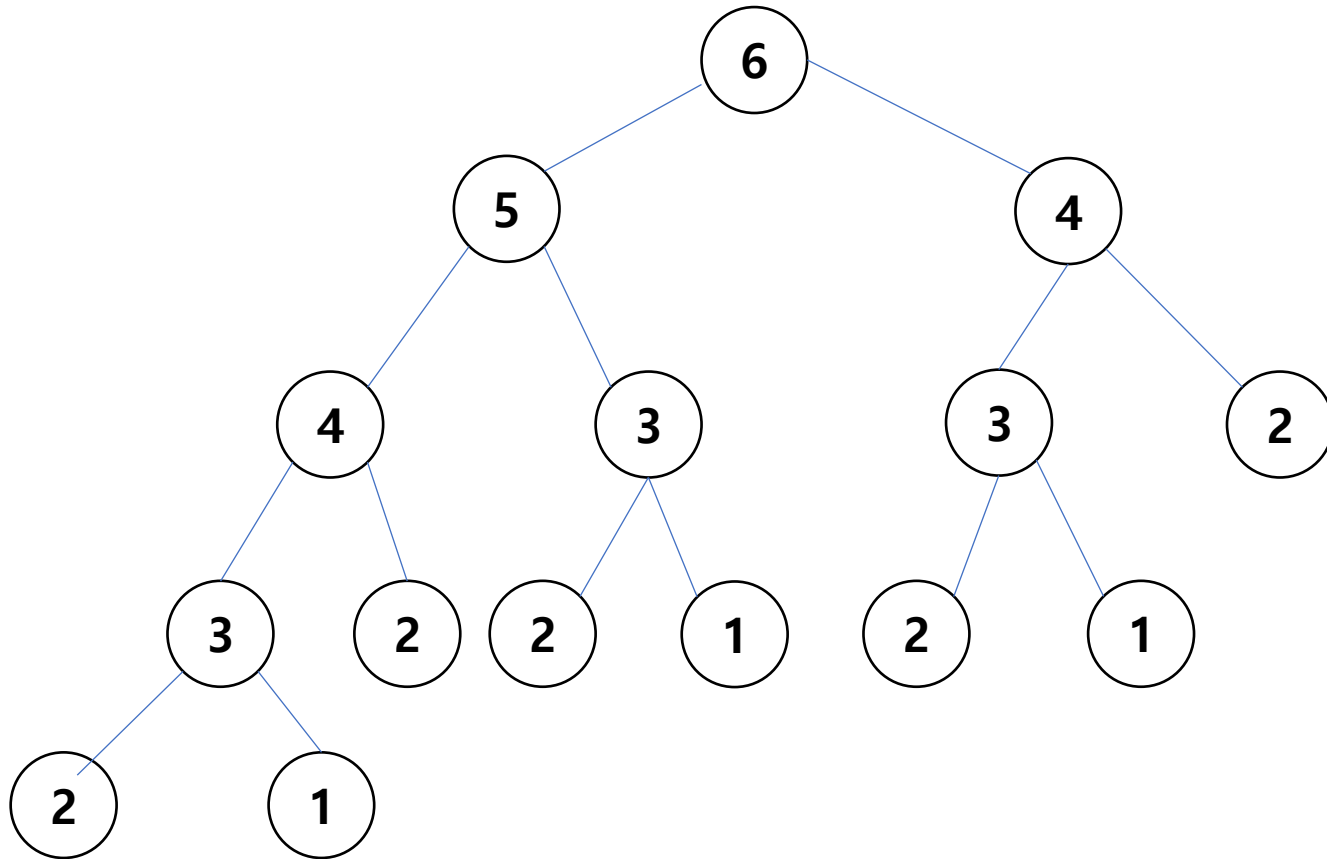
언제 써야 할까?

- 어떤 상태 공간에 똑같은 조건으로 방문 할 일이 있을때
- 계산의 중복이 일어날 수 있을때
- 대표적으로 경우의 수 구하기, 최소최대값 구하기, 등등

특징?

- Top-down, Bottom-up, 두 가지 방식이 있다.
- 완전탐색에서부터 출발하면 쉽다.
- 어려운 문제는 정말 어렵고, 쉬운 문제는 정말 쉽다.
- 어지간한 코딩테스트에서는 무조건 나온다

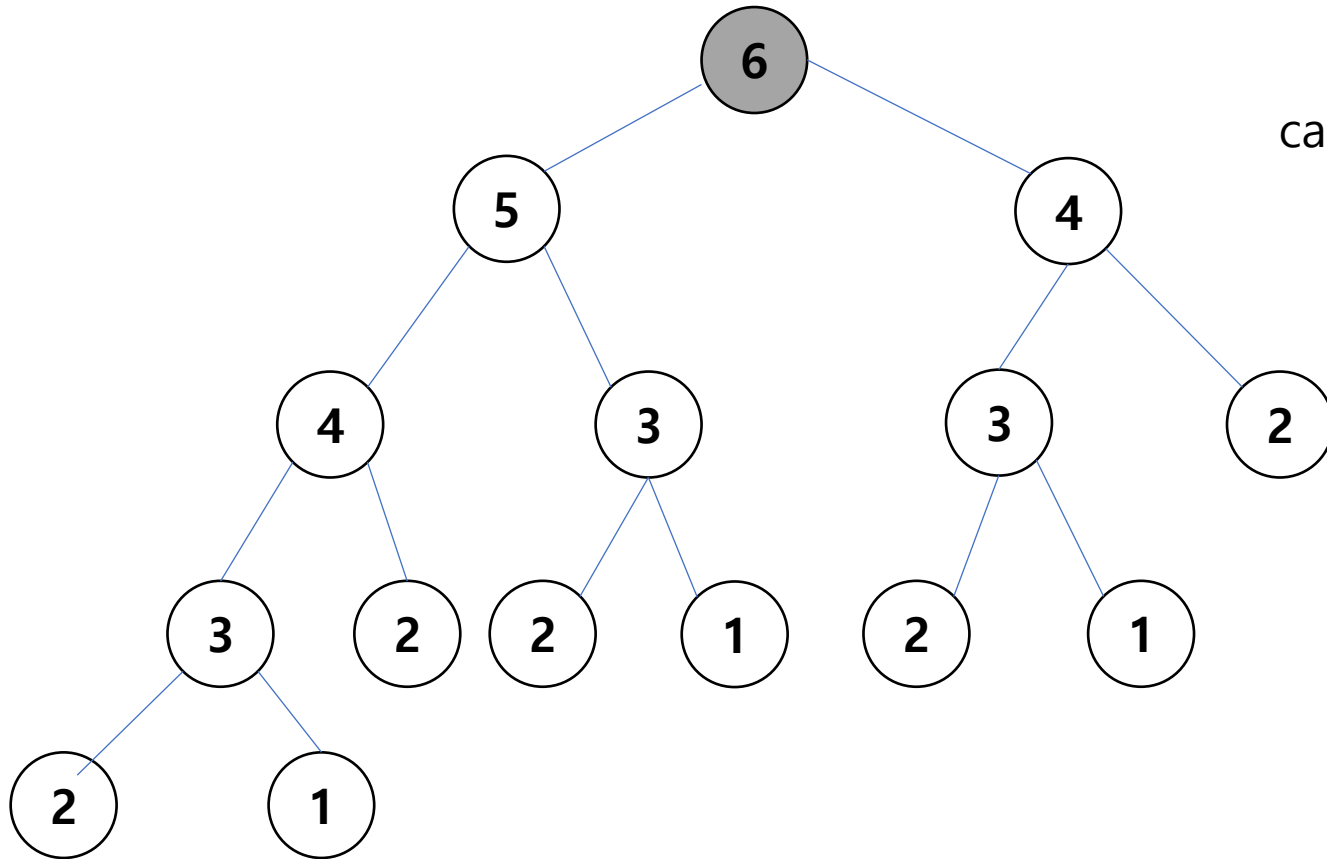
- 6번째 피보나치 수 계산



대략 $O(2^N)$

- 중복 되는 계산이 존재
- cache라는 메모리를 두어
다이나믹 프로그래밍 적용
- 이때 cache를 -1로 초기화 하는데
이는 아직 계산이 안됐음을 의미

- 6번째 피보나치 수 계산

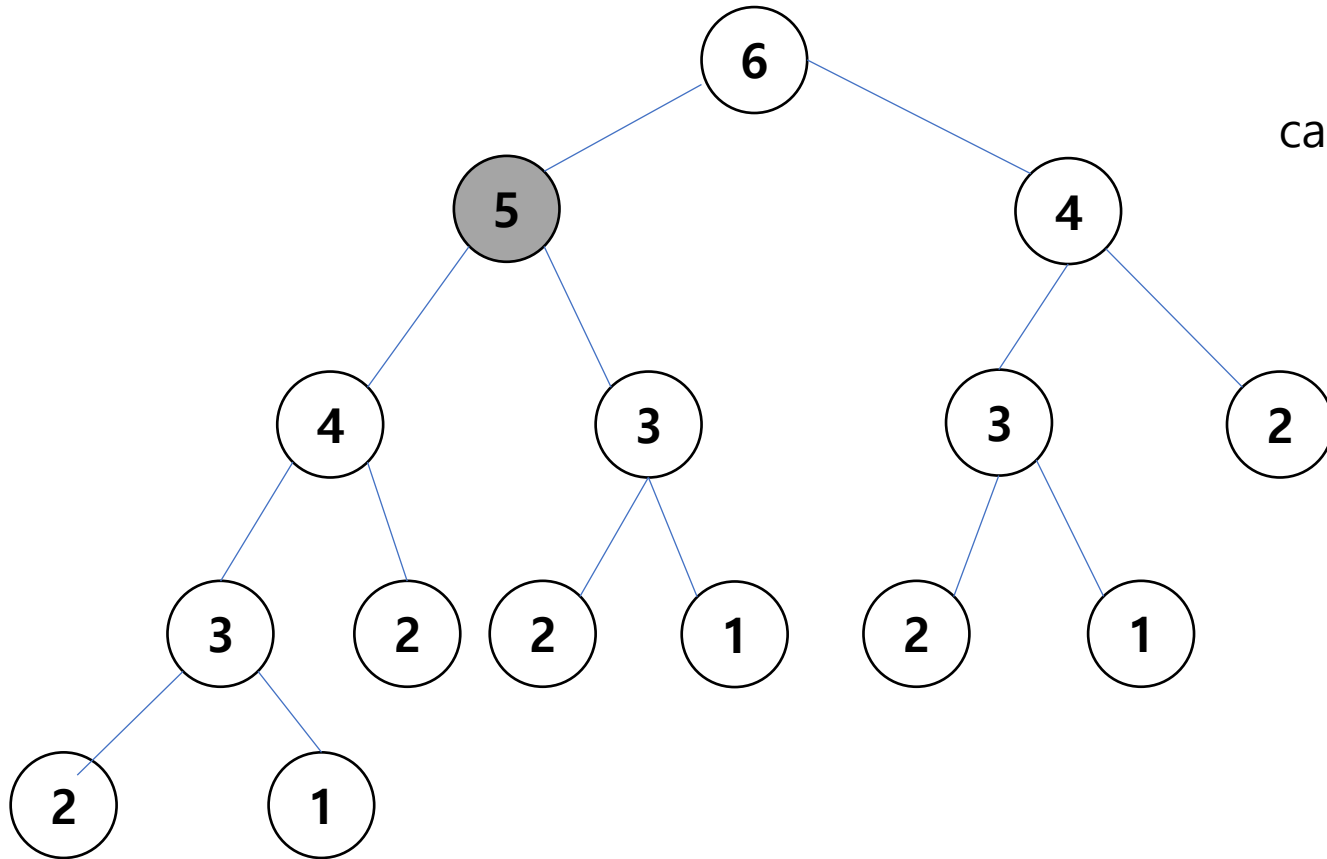


	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)
cache	1	1	-1	-1	-1	-1

F(1)과 F(2)는 더 이상 쪼갤 수 없으므로 미리 계산된 값.

재귀함수에서 종료조건(기저사례) 이라고 생각하면 된다!

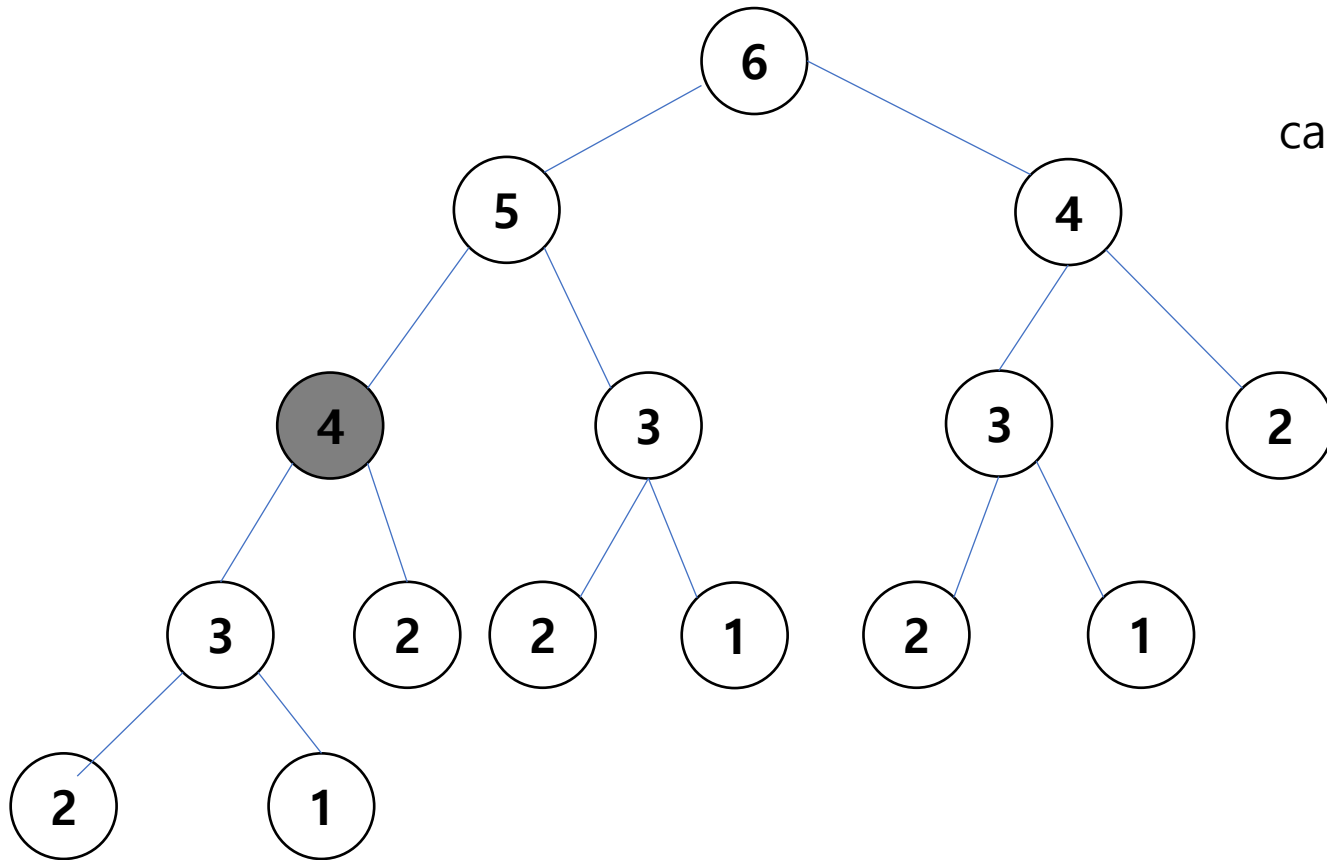
- 6번째 피보나치 수 계산



cache

F(1)	F(2)	F(3)	F(4)	F(5)	F(6)
1	1	-1	-1	-1	-1

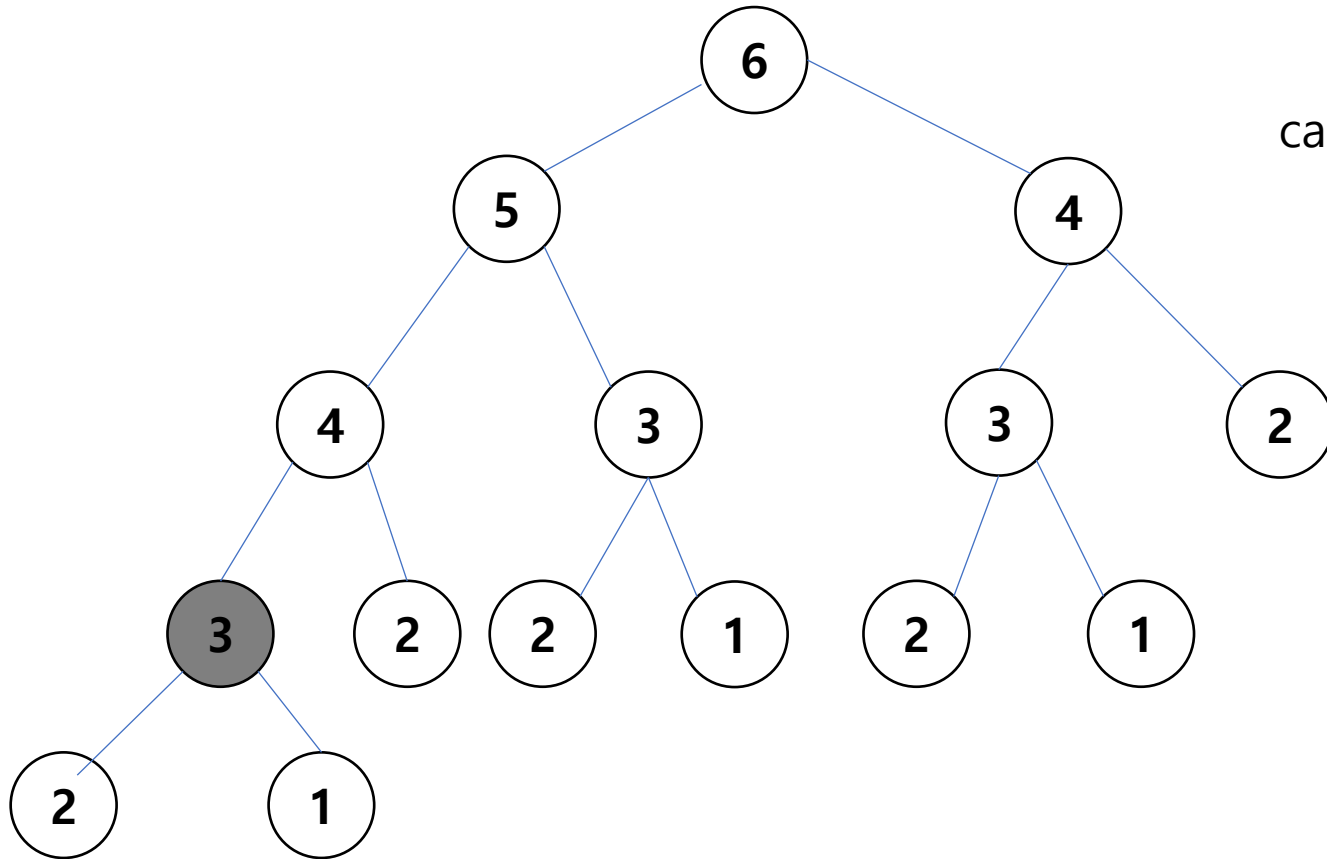
- 6번째 피보나치 수 계산



cache

F(1)	F(2)	F(3)	F(4)	F(5)	F(6)
1	1	-1	-1	-1	-1

- 6번째 피보나치 수 계산

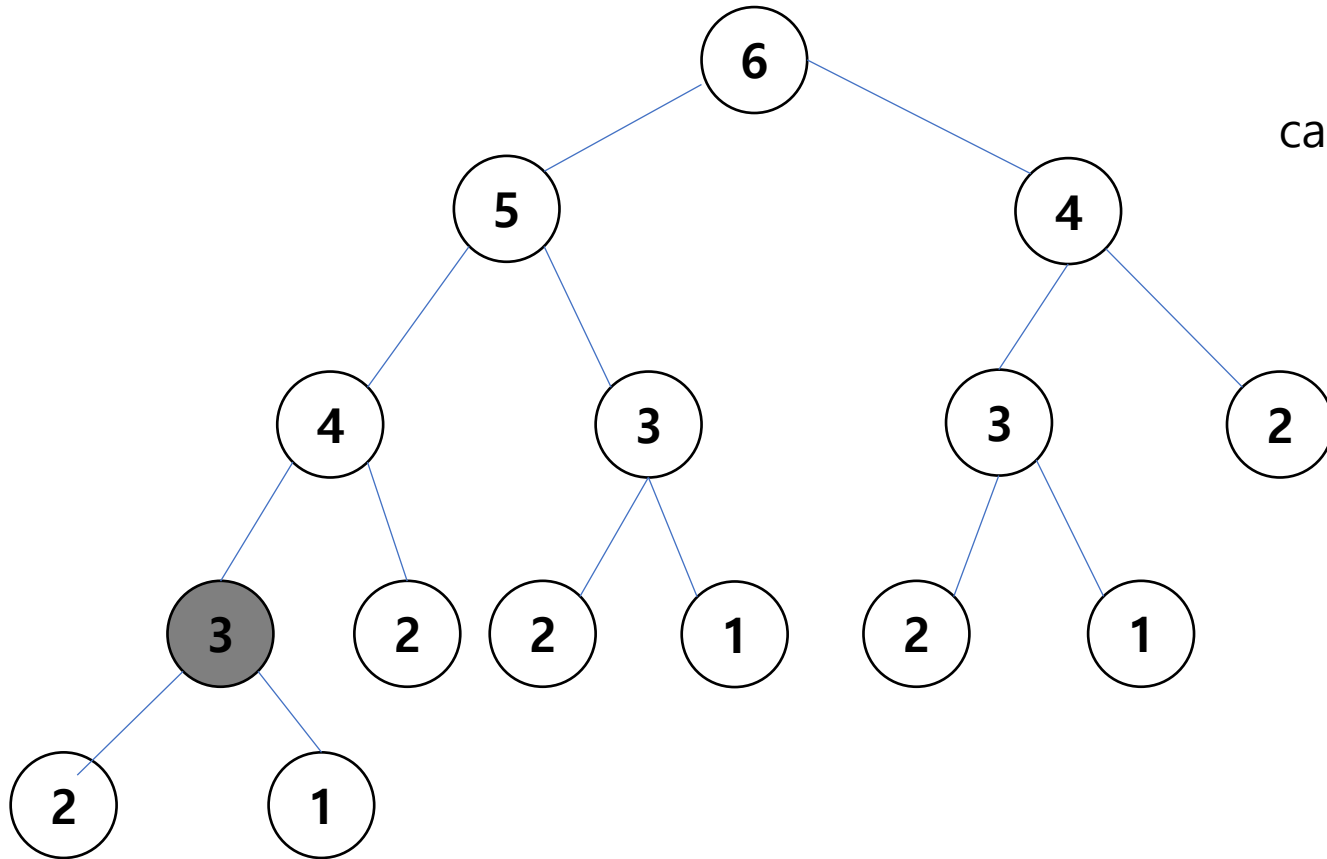


cache

F(1)	F(2)	F(3)	F(4)	F(5)	F(6)
1	1	-1	-1	-1	-1

- 만약 처음 초기화한 값인 -1이
아니면 이미 계산 되었다고 판단

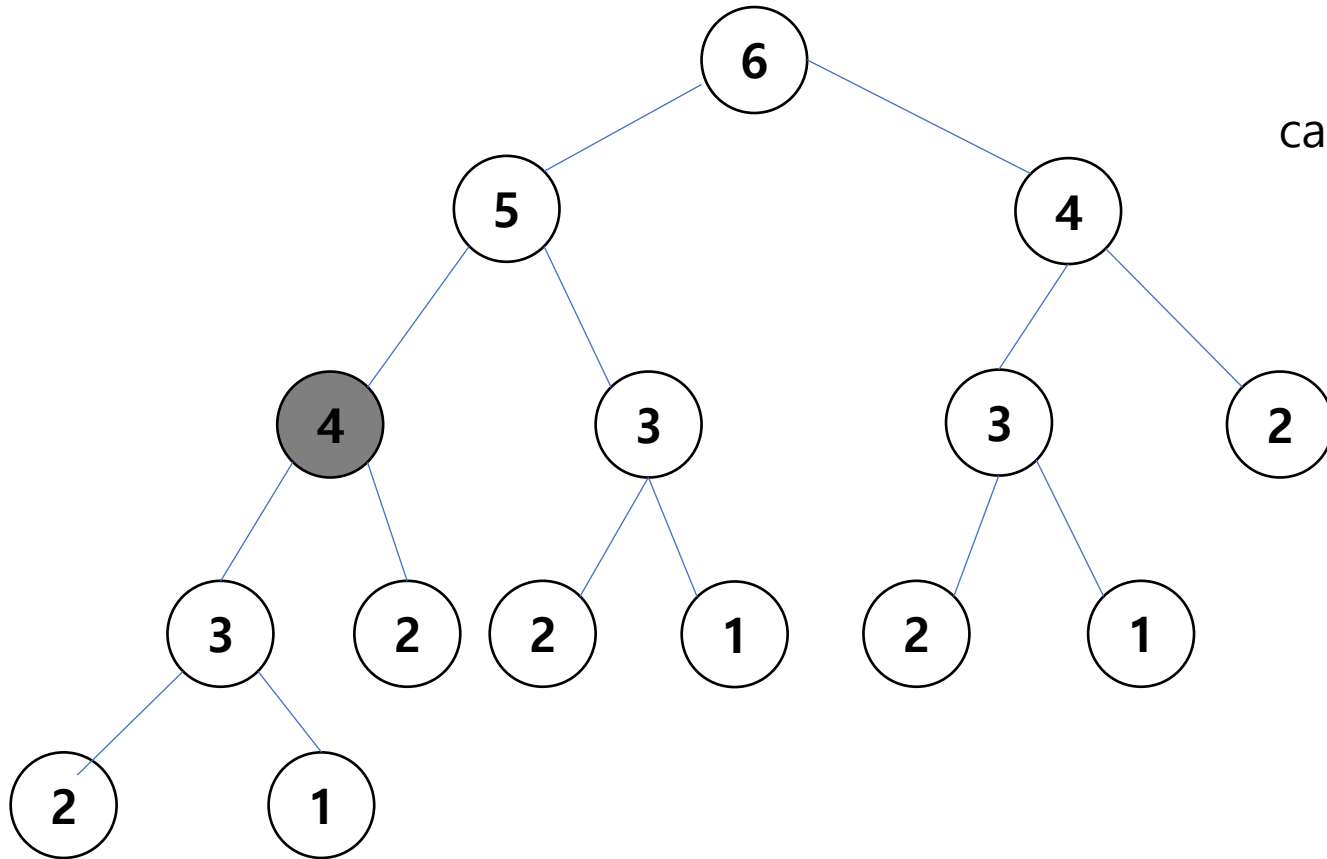
- 6번째 피보나치 수 계산



cache

F(1)	F(2)	F(3)	F(4)	F(5)	F(6)
1	1	2	-1	-1	-1

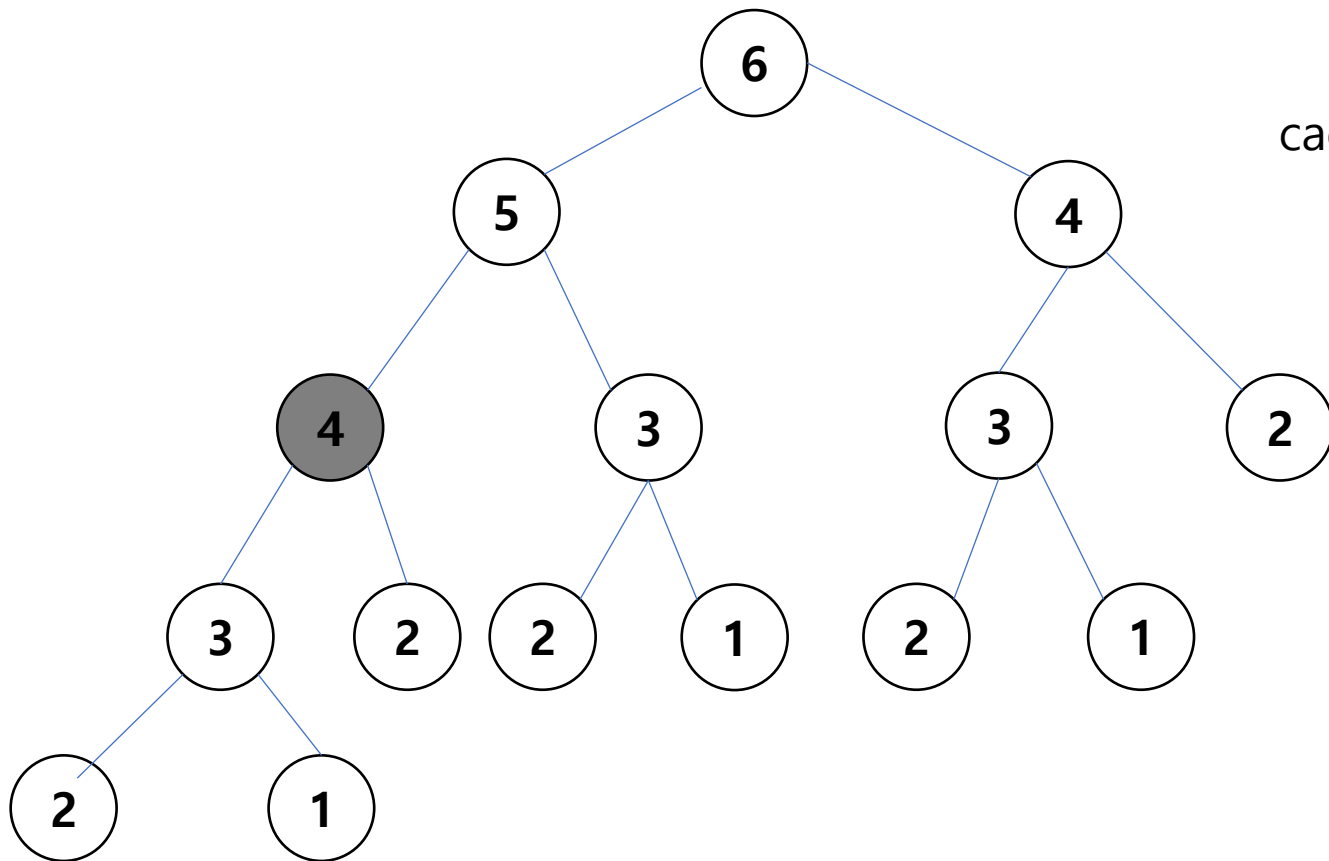
- 6번째 피보나치 수 계산



cache

F(1)	F(2)	F(3)	F(4)	F(5)	F(6)
1	1	2	-1	-1	-1

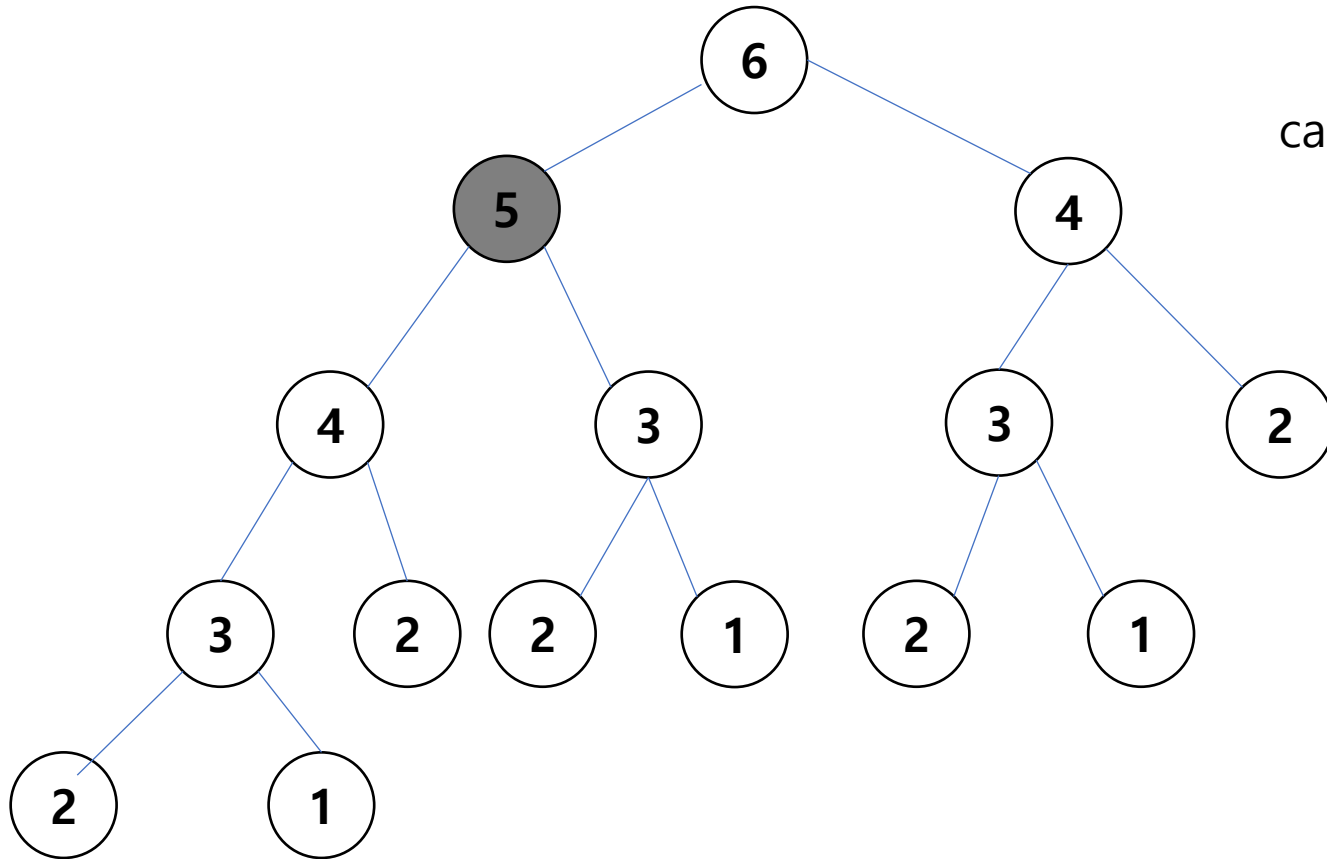
- 6번째 피보나치 수 계산



cache

F(1)	F(2)	F(3)	F(4)	F(5)	F(6)
1	1	2	3	-1	-1

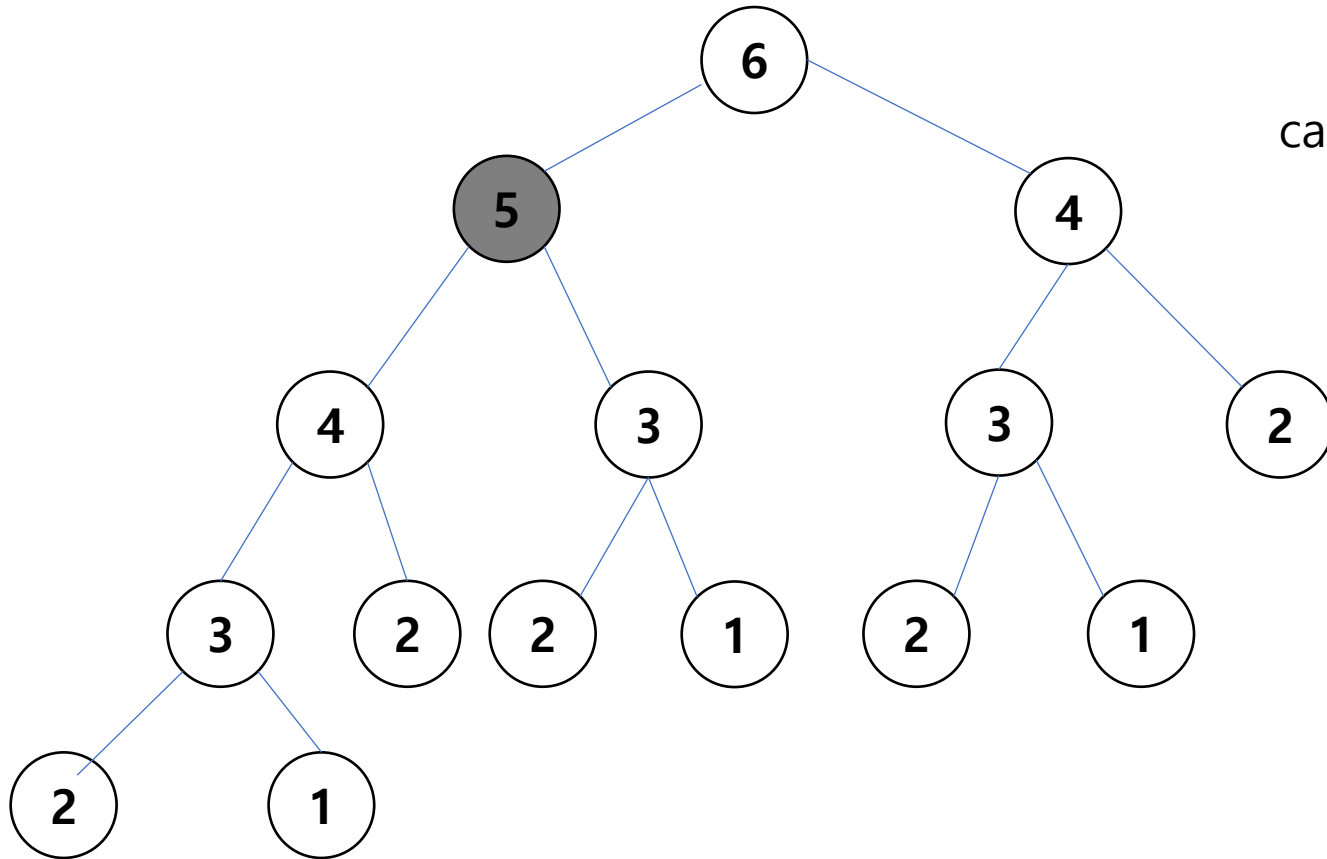
- 6번째 피보나치 수 계산



cache

F(1)	F(2)	F(3)	F(4)	F(5)	F(6)
1	1	2	3	-1	-1

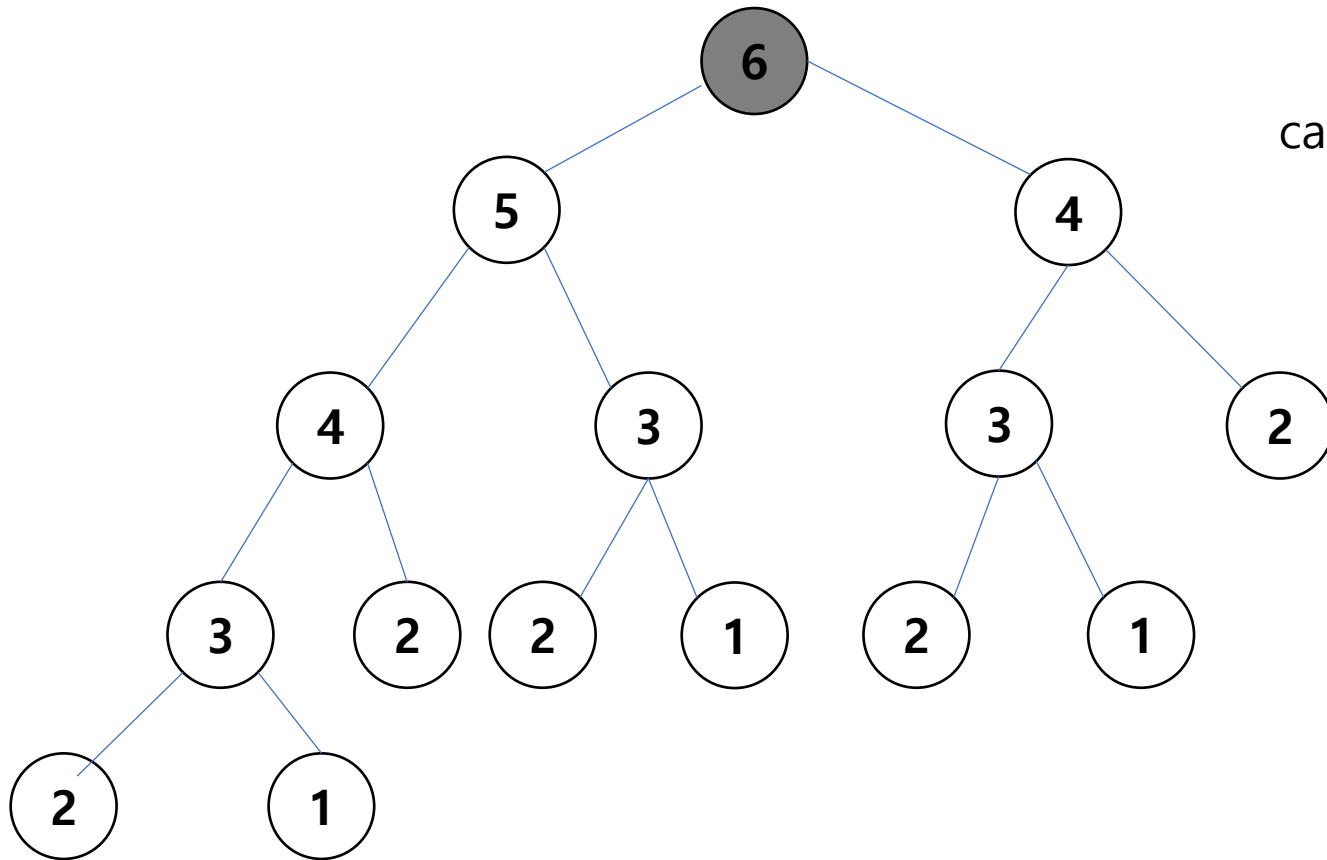
- 6번째 피보나치 수 계산



cache

F(1)	F(2)	F(3)	F(4)	F(5)	F(6)
1	1	2	3	5	-1

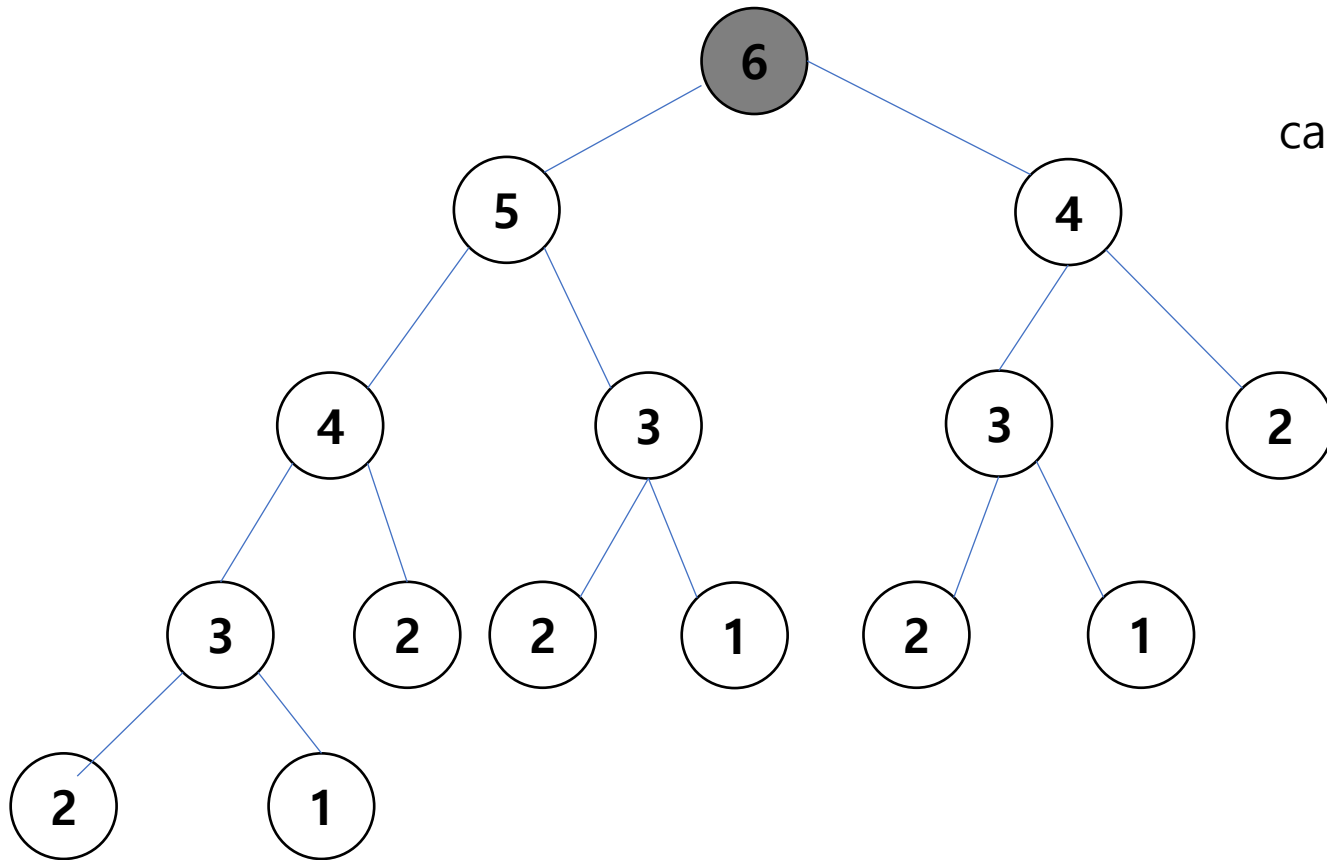
- 6번째 피보나치 수 계산



cache

F(1)	F(2)	F(3)	F(4)	F(5)	F(6)
1	1	2	3	5	-1

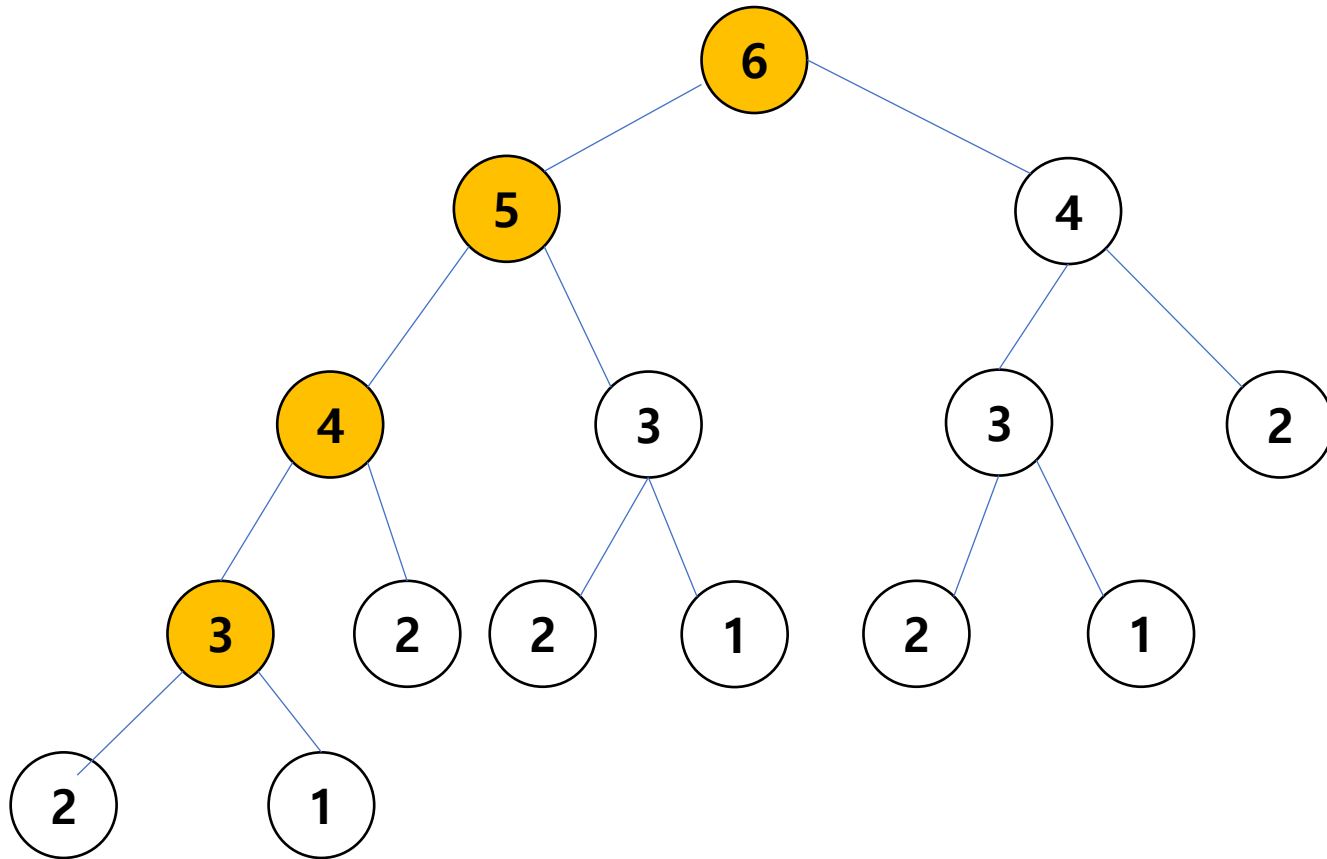
- 6번째 피보나치 수 계산



cache

F(1)	F(2)	F(3)	F(4)	F(5)	F(6)
1	1	2	3	5	8

- 6번째 피보나치 수 계산



완전탐색
 $O(2^N)$

vs

DP
 $O(N)$

미리 계산해 놓은 값으로
불필요한 재귀를 막는다.

Fib(6)을 계산하기 위해 cache[5],
cache[4], ... 를 이용

- 이와 같은 방법이 Top-down

- 실제 코드 구현은?

```
int Fib(int n)
```

```
    if(cache[n] != -1)
```

```
        return cache[n]
```

```
    if(n == 1 or n == 2)
```

```
        return cache[n] = 1
```

```
    return cache[n] = Fib(n-1) + Fib(n-2)
```

DP

비교

```
int Fib(int n)
```

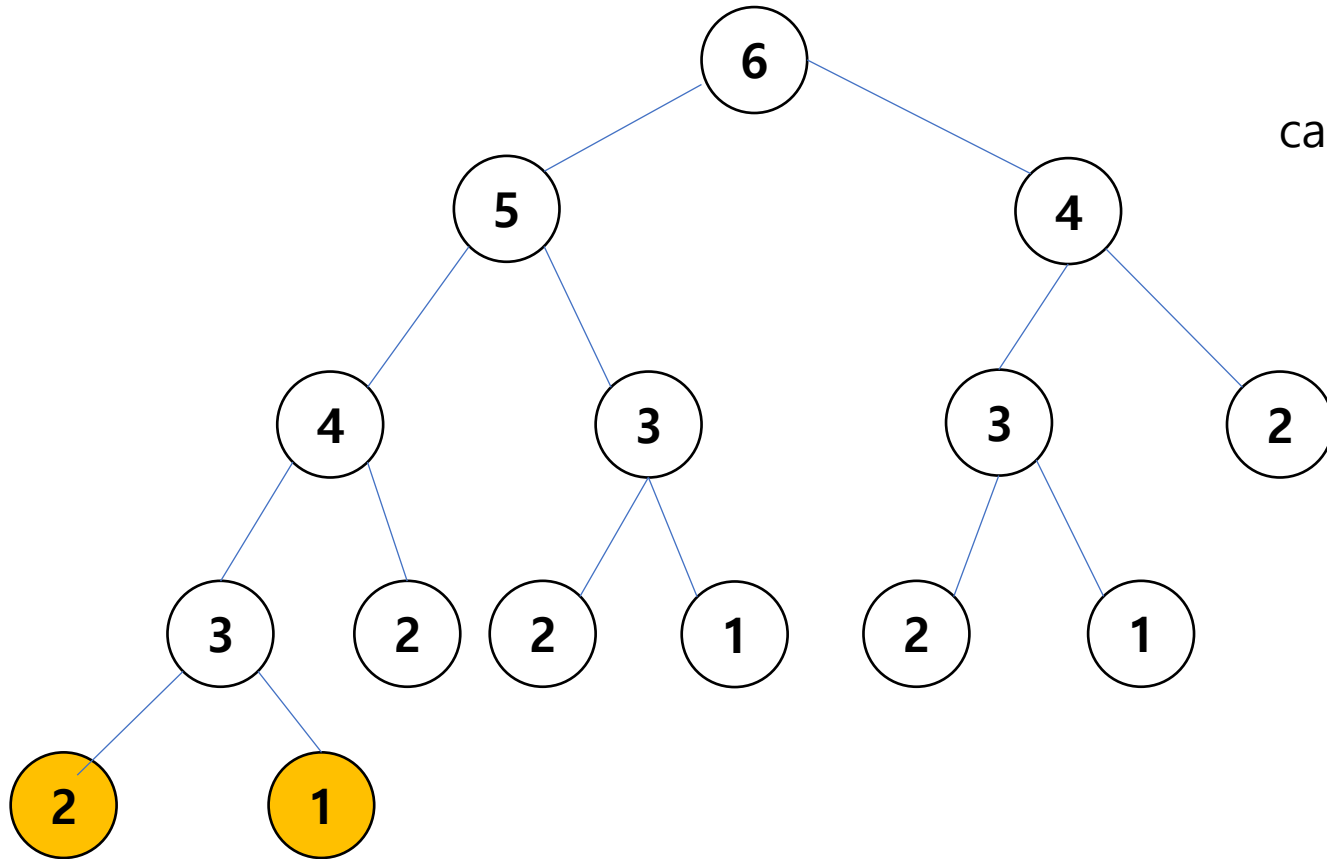
```
    if(n == 1 or n == 2)
```

```
        return 1
```

```
    return Fib(n-1) + Fib(n-2)
```

완전탐색

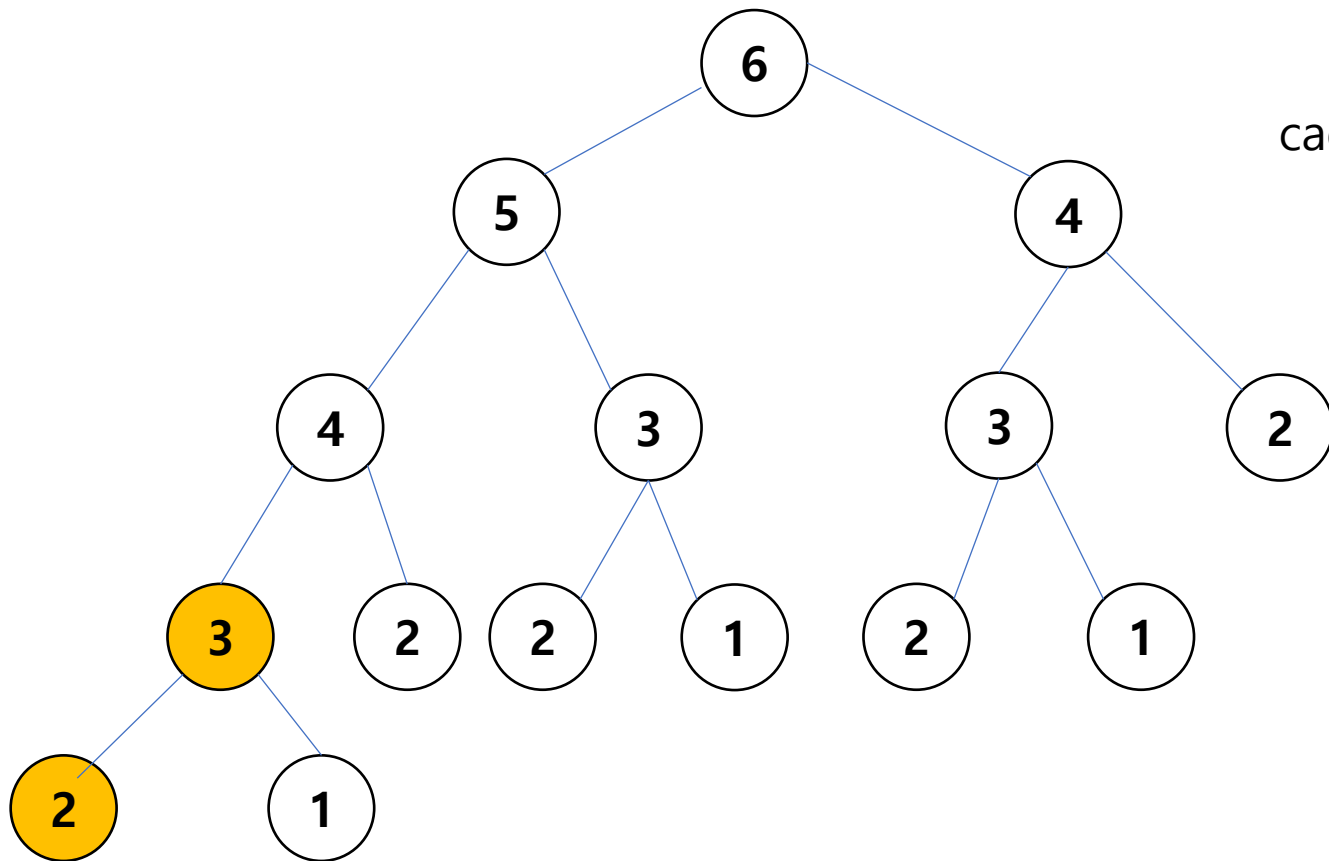
- 6번째 피보나치 수 계산



cache

F(1)	F(2)	F(3)	F(4)	F(5)	F(6)
1	1	-1	-1	-1	-1

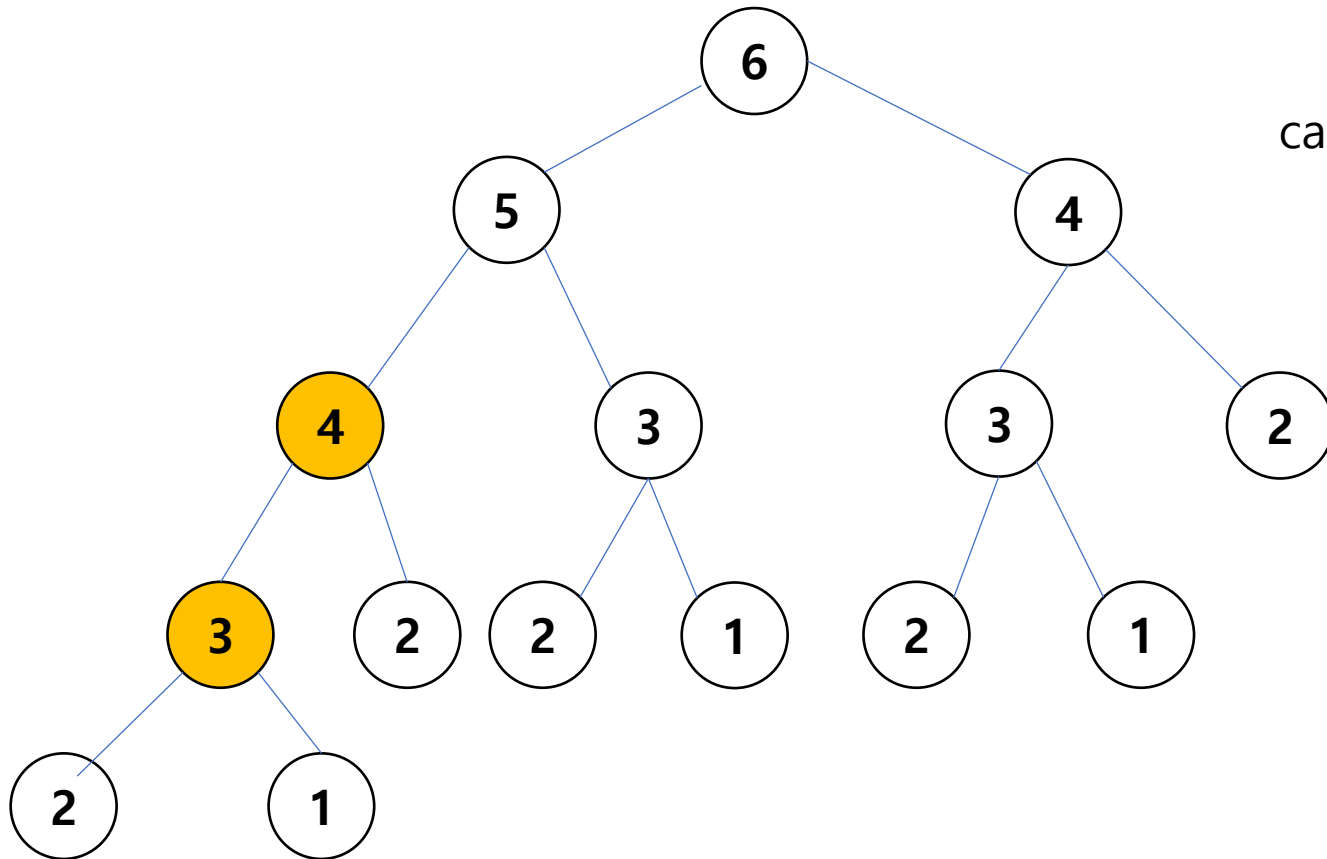
- 6번째 피보나치 수 계산



	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)
cache	1	1	2	-1	-1	-1

```
cache[3] = cache[2] + cache[1]
```

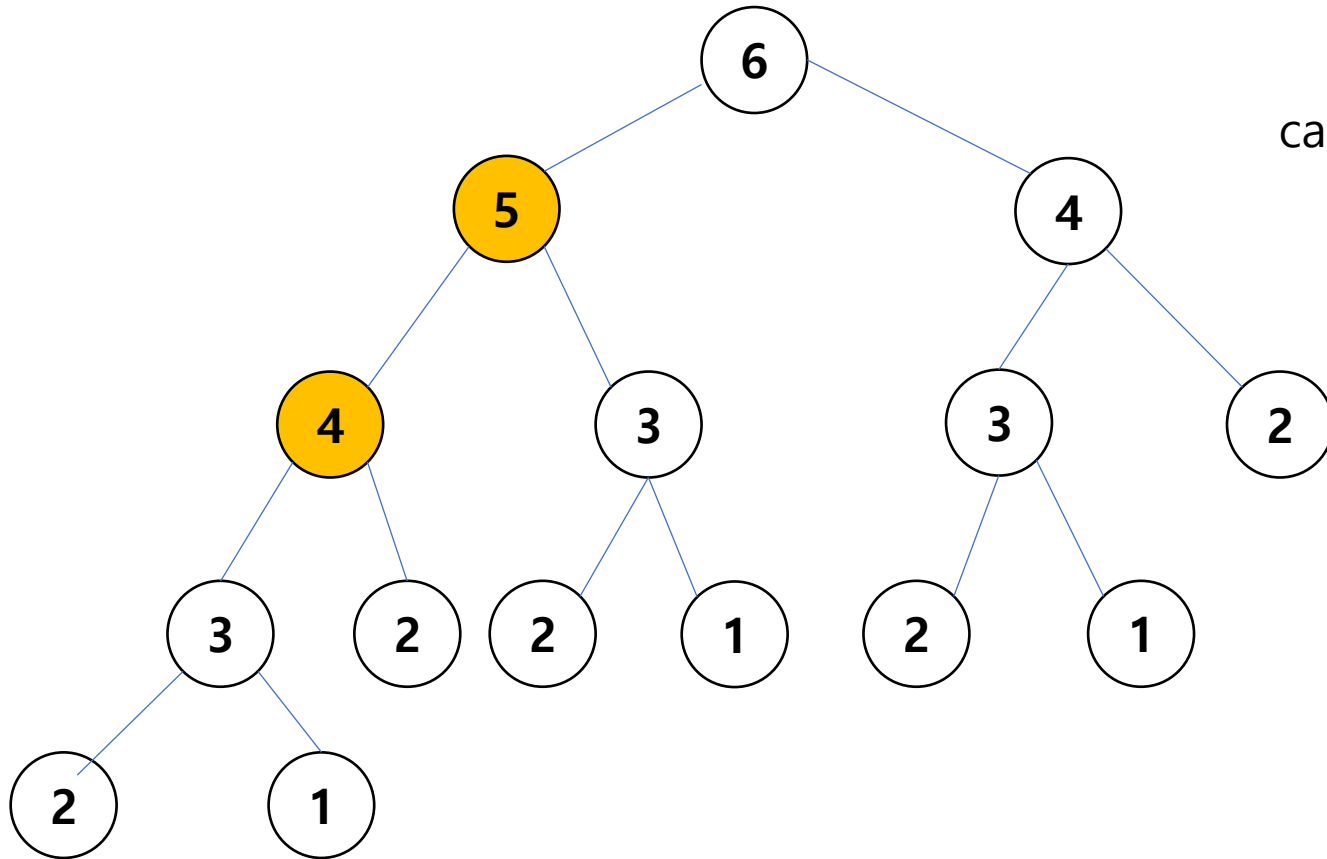
- 6번째 피보나치 수 계산



	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)
cache	1	1	2	3	-1	-1

$$\text{cache}[4] = \text{cache}[3] + \text{cache}[2]$$

- 6번째 피보나치 수 계산

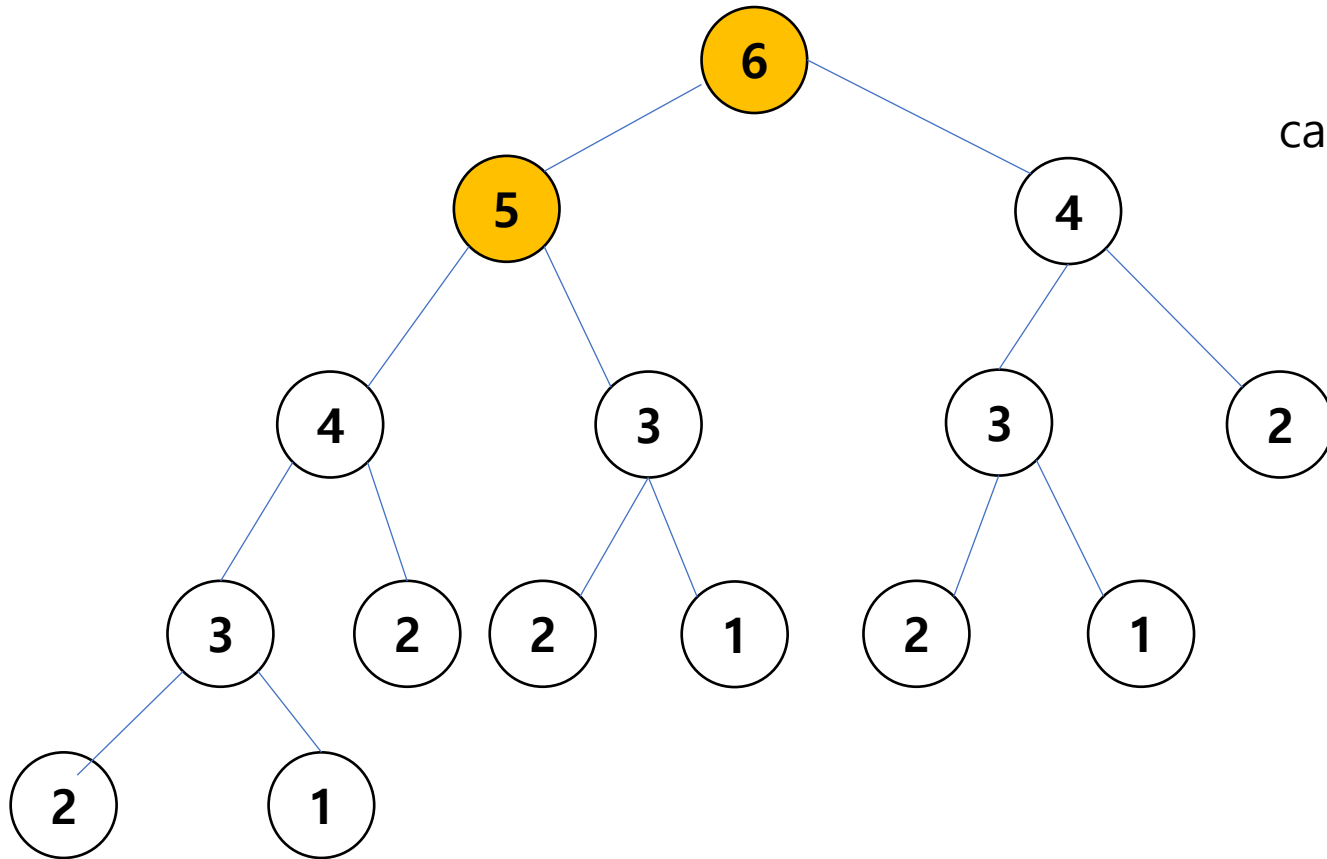


cache

F(1)	F(2)	F(3)	F(4)	F(5)	F(6)
1	1	2	3	5	-1

$$\text{cache}[5] = \text{cache}[4] + \text{cache}[3]$$

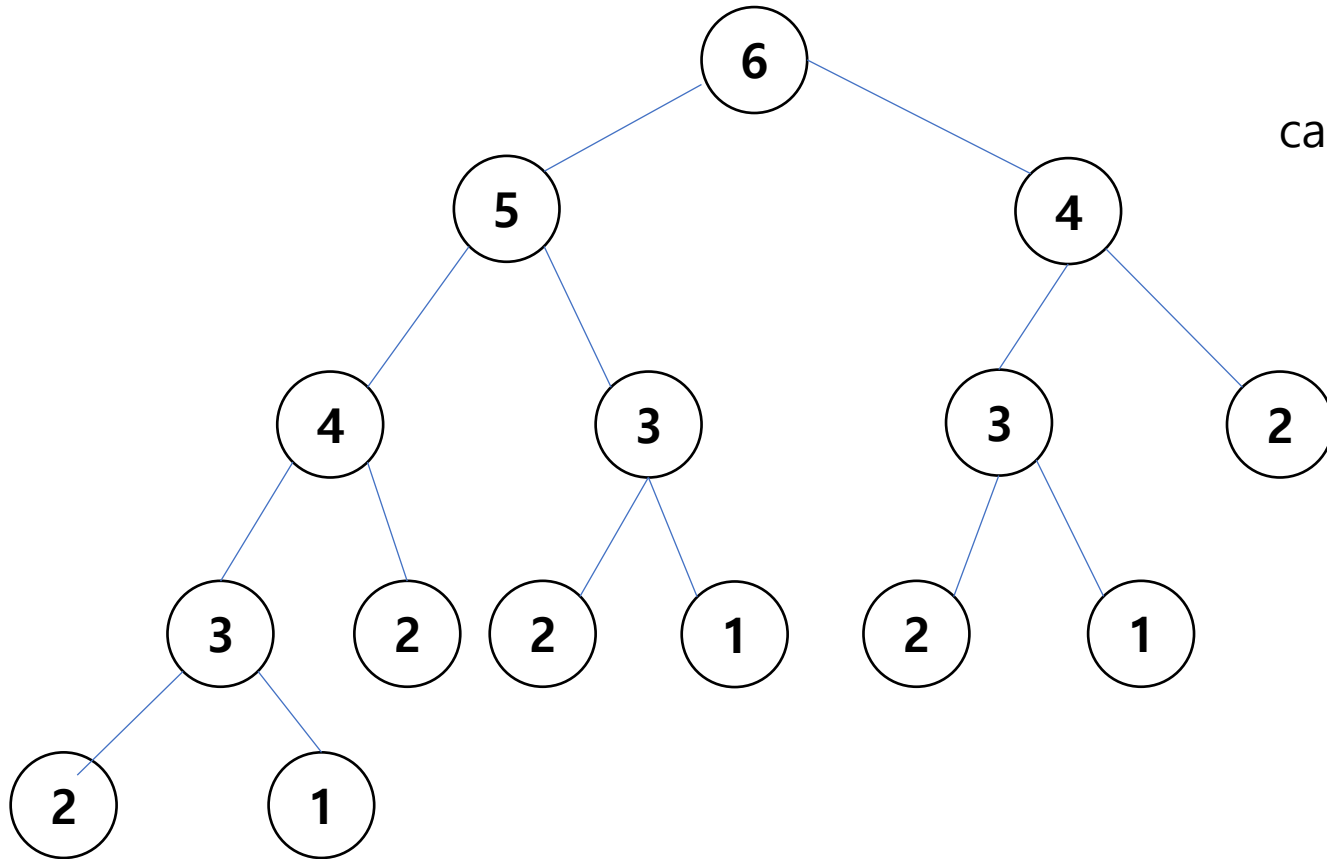
- 6번째 피보나치 수 계산



	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)
cache	1	1	2	3	5	8

$$\text{cache}[6] = \text{cache}[5] + \text{cache}[4]$$

- 6번째 피보나치 수 계산



cache

F(1)	F(2)	F(3)	F(4)	F(5)	F(6)
1	1	2	3	5	8

```
for(int i=3; i<=6; i++)  
    cache[i] = cache[i-1] + cache[i-2]
```

- 이와 같은 방법이 Bottom-up

다이나믹 프로그래밍(DP)

- Top-down VS Bottom-up

	Top-down	Bottom-up
시간	거의 동일	
구현 방식	재귀	반복문
메모리	모든 값을 저장	당장 계산에 필요한것만 저장

a	b	c			
1	1	2	3	5	8

	a	b	c		
1	1	2	3	5	8

	a	b	c		
1	1	2	3	5	8

```
a = 1, b = 1
for(int i=3; i<=6; i++)
    c = a+b
    a = b
    b = c
```

- 이와 같은 기법을 슬라이딩-윈도우 라고 한다!

- 메모이제이션 : 계산에 필요한 cache라는 메모리를 두어 계산의 중복을 막아준다.
- 어떤 상태 공간에 똑같은 조건으로 방문 할 일이 있을때(계산의 중복이 이루어 질 때)
- 큰 문제를 작은 문제로 나눌 수 있고 작은 문제의 답으로 큰 문제의 답을 구할 때
 - 좀더 쉽게 생각하는 방법?
 - cache를 수학에서의 함수라고 생각
 - $cache[x][y] = f(x,y)$
 - 재귀?

이분탐색(Binary Search)

이분탐색이란?

- 범위를 둘로 쪼개어 탐색하는 방법

언제 써야 할까?

- 범위 안에 있는 값들이 특정 순서대로 정렬되어 있을 때
- int형 범위를 넘어가는 숫자가 나올 때?

특징?

- 정렬을 기본으로 하는 경우가 많다
- "x의 최대값을 구하시오"를 "x가 y일때도 문제 조건에 성립할까?"로 바꾸는 발상의 전환
- 왼쪽범위, 중간값, 오른쪽범위 3개의 변수를 사용한다
- **오버플로우를 조심해야한다.**

이분탐색(Binary Search)

예제 : 1~1000까지 숫자중에서 x를 찾는 Up&Down 게임 진행.

1	2	...	x	...	999	1000
---	---	-----	---	-----	-----	------

여러가지 전략 : 절반씩 끊어서 찾기, 짝수만 찾기, 100단위로 끊어서 찾기, ...

무식하게 풀기(Brute Force) : 1부터 오름차순으로 1000까지 일일이 다 찾기

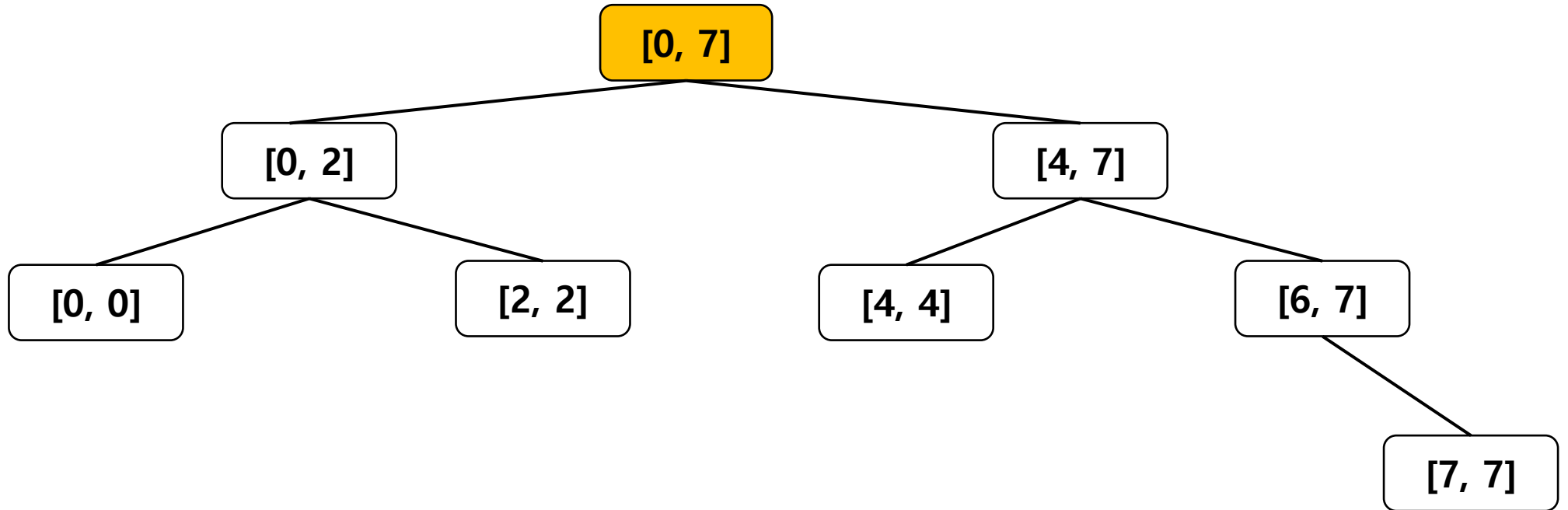
이분탐색(Binary Search) : 절반씩 끊어서 찾기

- 이분탐색을 쓸 수 있는 이유 : 배열이 정렬 되어 있다.

이분탐색(Binary Search)

- size가 8인 vector에서 4의 위치를 찾기

$$\text{mid} = (\text{l} + \text{r}) / 2$$

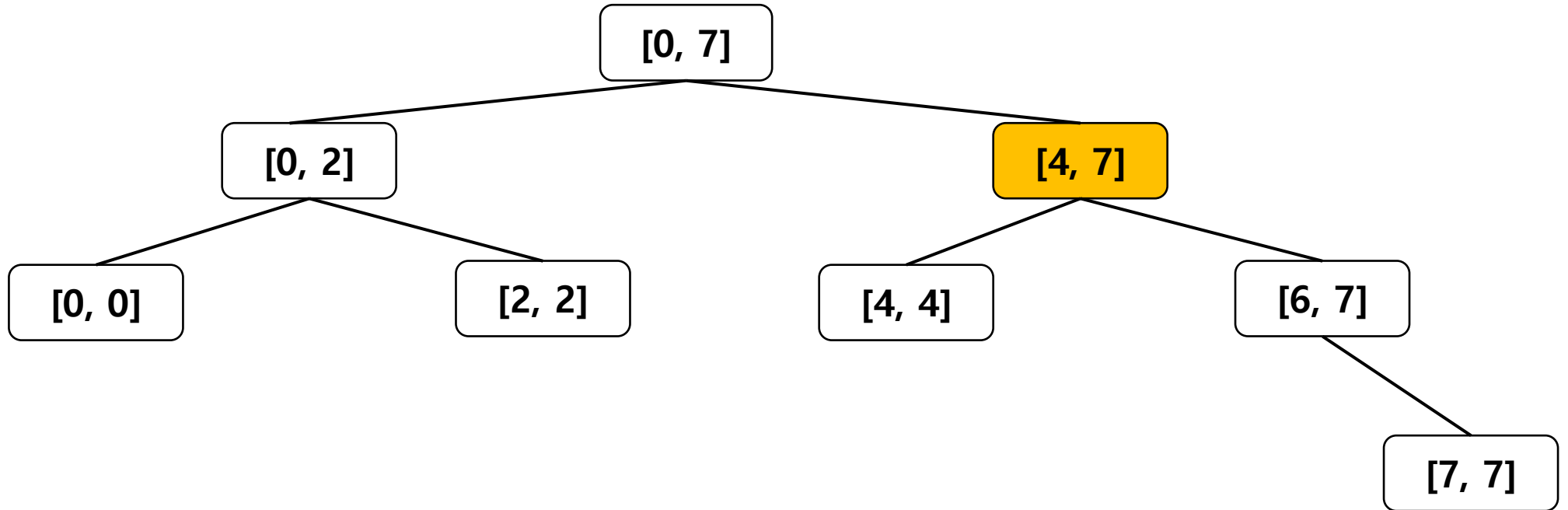


vector

-7	-3	1	3	4	9	10	12
----	----	---	---	---	---	----	----

이분탐색(Binary Search)

- size가 8인 vector에서 4의 위치를 찾기

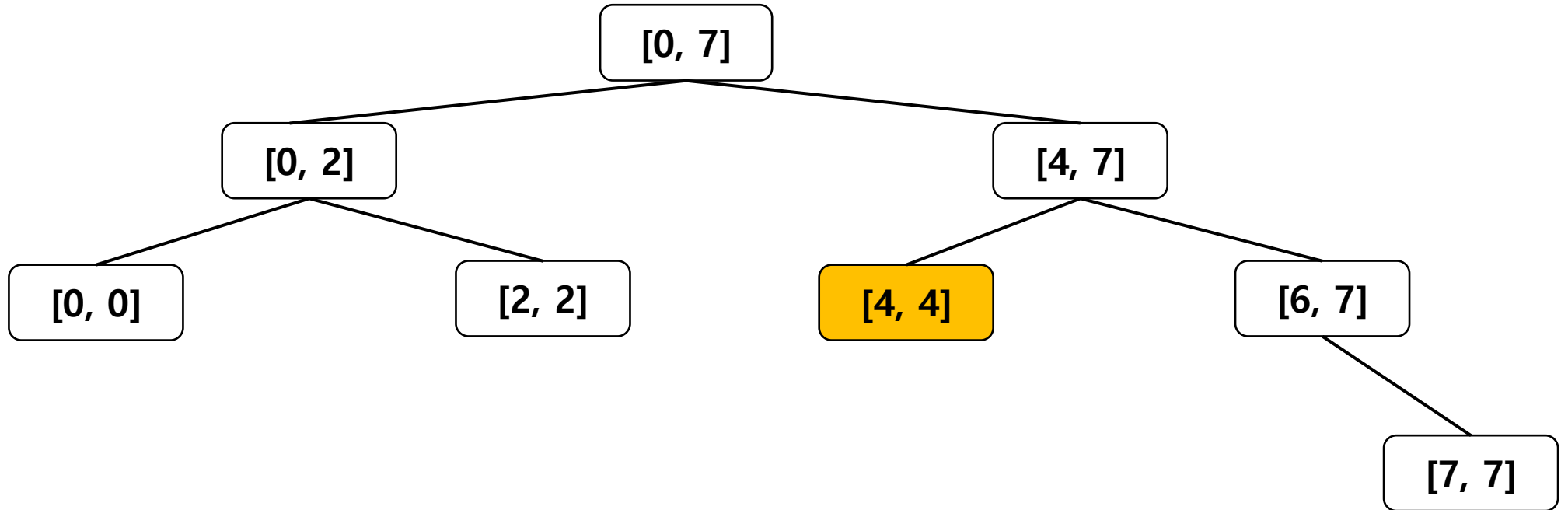


vector

-3	1	2	4	7	9	10	12
----	---	---	---	---	---	----	----

이분탐색(Binary Search)

- size가 8인 vector에서 4의 위치를 찾기



vector

-3	1	2	4	7	9	10	12
----	---	---	---	---	---	----	----

$O(\log N)$ 만에 탐색 완료

이분탐색(Binary Search)

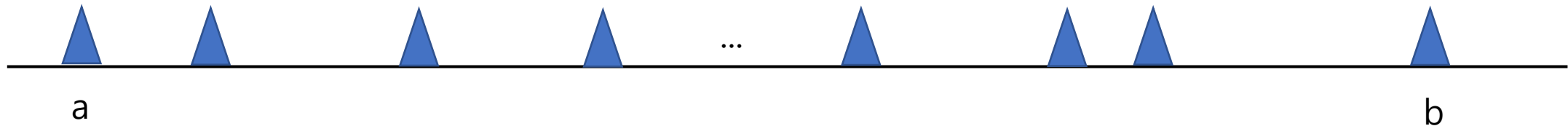
- size가 8인 vector에서 4의 위치를 찾기
 - 실제 구현?

int find (int l, int r, int x) : 범위가 [l , r] 일때 x의 위치 반환

```
int find (int l, int r, int x)
    int mid = (l+r)/2
    if(l > r)
        return -1
    if(list[mid] == x)
        return mid
    else if(list[mid] > x)
        return find(l, mid-1, x)
    else if(list[mid] < x)
        return find(mid+1, r, x)
```

```
bool find (int l, int r, int x)
    int mid = (l+r)/2
    if(l > r)
        return false
    if(list[mid] == x)
        return true
    else if(list[mid] > x)
        return find(l, mid-1, x)
    else if(list[mid] < x)
        return find(mid+1, r, x)
```

- "x의 최대값을 구하시오"를 "x가 y일때도 문제 조건에 성립할까?"로 바꾸는 발상의 전환



N개의 집을 M명의 사람이 하나씩 고른다. 고른 집 중에서 집과 집 사이의 거리가 가장 가까운값을 x 라고 할 때, x 의 최대값을 구하시오.

완전탐색? : x 가 1이 되도록 M개의 집을 택할 수 있을까?

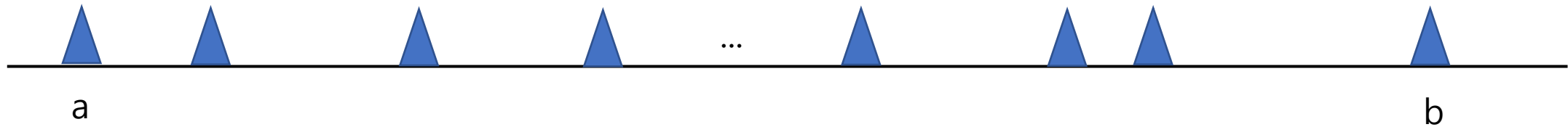
x 가 2가 되도록 M개의 집을 택할 수 있을까?

...

x 가 n 이 되도록 M개의 집을 택할 수 있을까?

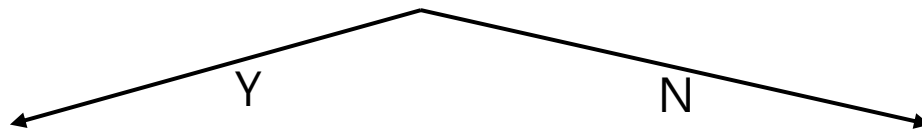
이분탐색(Binary Search)

- "x의 최대값을 구하시오"를 "x가 y일때도 문제 조건에 성립할까?"로 바꾸는 발상의 전환



N개의 집을 M명의 사람이 하나씩 고른다. 고른 집 중에서 집과 집 사이의 거리가 가장 가까운값을 x라고 할 때, x의 최대값을 구하시오.

이분탐색 : x가 mid가 되도록 M개의 집을 택할 수 있을까?



x가 $(r+mid)/2 \sim M$ 개의 집을~ ?

x가 $(l+mid)/2 \sim M$ 개의 집을~ ?

1. 1, 2, 3 더하기 : <https://www.acmicpc.net/problem/9095>

2. 1로 만들기 : <https://www.acmicpc.net/problem/1463>

● ● DP를 이용하는 문제 - 1,2,3 더하기

문제 요약 :

<https://www.acmicpc.net/problem/9095>

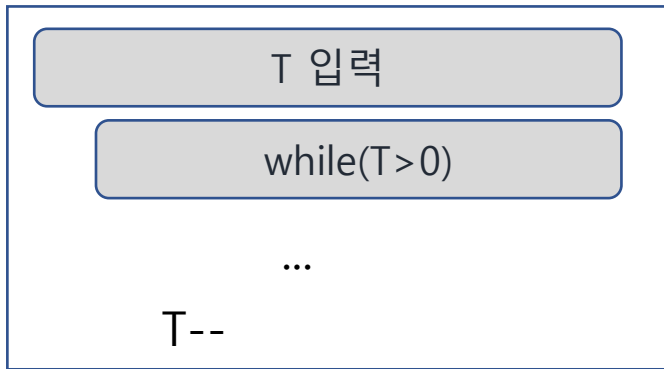
- $1 \leq N \leq 10$, 여러개의 테스트 케이스 존재
- 정수 N이 주어졌을 때, N을 1,2,3 의 합을 나타내는 방법의 수
- 구성하는 숫자가 같아도 순서가 다르면 다른 경우의 수
- DP로 경우의 수를 구하는 전형적인 문제

고려해야 할 사항 :

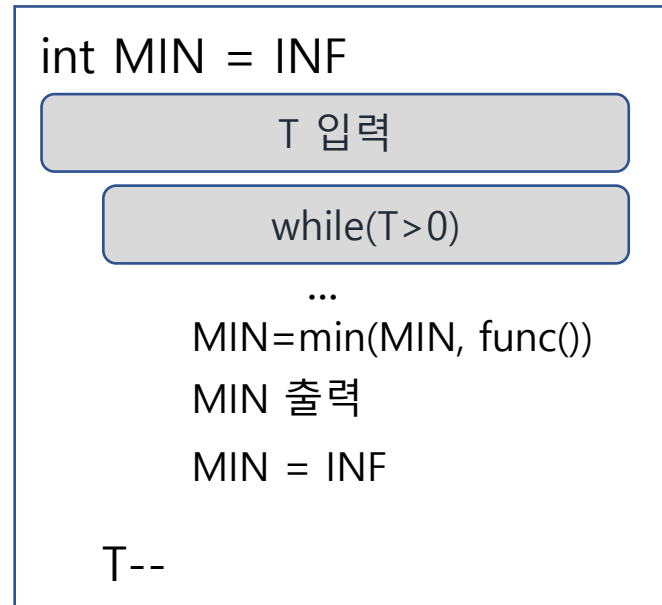
- 여러개의 테스트 케이스 처리
- 메모이제이션을 사용하기 위해 cache 배열을 -1로 미리 초기화 해준다.
- 각각의 테스트 케이스에 대해 cache를 초기화 해줘야 할까?

DP를 이용하는 문제 - 1,2,3 더하기

- 여러개의 테스트 케이스 처리



보통은 각각의 테스트 케이스 진입할때
마다 전역변수를 초기화 해주는게 좋다



vector도 마찬가지

● ● DP를 이용하는 문제 – 1,2,3 더하기

- 각각의 테스트 케이스에 대해 cache를 초기화 해줘야 할까?
 - N만 같으면 결과값은 똑같으니 필요없다.
- 메모이제이션을 사용하기 위해 cache 배열을 -1로 미리 초기화 해준다.
 - c++의 memset(배열, 값, sizeof(배열))
 - 값에는 0이나 -1 만 넣을 수 있다
 - fill()이나 단순 for문으로 초기화 해주어도 상관x

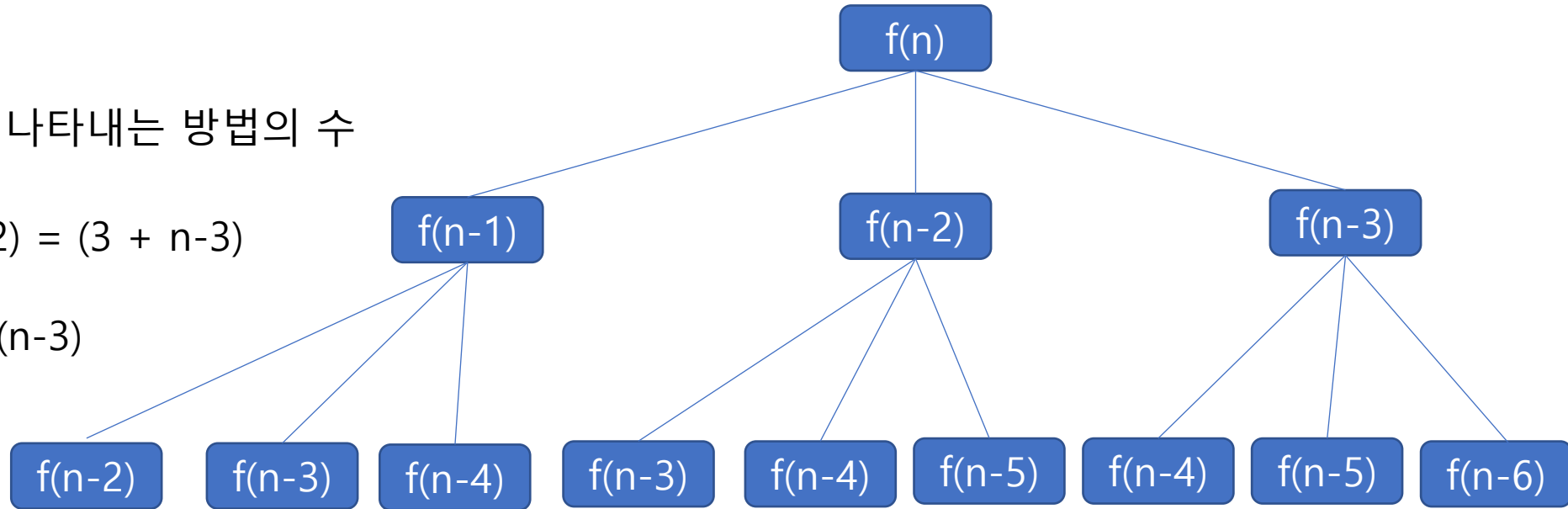
DP를 이용하는 문제 - 1,2,3 더하기

- Top-down

$f(n)$: n 을 1,2,3의 합으로 나타내는 방법의 수

$$n = (1 + n-1) = (2 + n-2) = (3 + n-3)$$

$$f(n) = f(n-1) + f(n-2) + f(n-3)$$



DP를 이용하는 문제 - 1,2,3 더하기

- Top-down

$$\begin{aligned}f(1) &= 1 \\f(2) &= 2 \\f(3) &= 4\end{aligned}$$

cache

-1	-1	-1	-1	-1	-1
----	----	----	----	----	----

int f(n)

if(cache[n] != -1)

return cache[n]

if(n==1)

return cache[n] = 1

else if(n==2)

return cache[n] = 2

else if(n==3)

return cache[n] = 4

return cache[n] = f(n-1)+f(n-2)+f(n-3)

1	2	4	-1	-1	-1
---	---	---	----	----	----

int f(n)

if(cache[n] != -1)

return cache[n]

return cache[n] = f(n-1)+f(n-2)+f(n-3)

<->

● ● DP를 이용하는 문제 – 1,2,3 더하기

- Bottom-up

$$f(1) = 1$$

$$f(2) = 2$$

$$f(3) = 4$$

```
cache[1] = 1
```

```
cache[2] = 2
```

```
cache[3] = 4
```

```
for(int i=4; i<=10; i++)
```

```
    cache[i] = cache[i-1] + cache[i-2] + cache[i-3]
```

● ● DP를 이용하는 문제 - 1로 만들기

문제 요약 :

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

- $1 \leq N \leq 10^6$
- 정수 N을 주어진 연산을 이용해 1로 만들때, 연산의 최소값을 구하시오
- DP로 최소값을 구하는 전형적인 문제

고려해야 할 사항 :

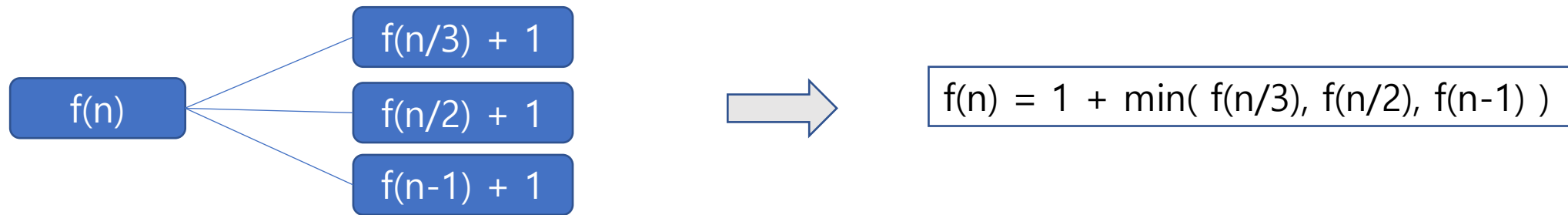
- 메모이제이션을 사용하기 위해 cache 배열을 -1로 미리 초기화 해준다.
- 경우의 수 말고 최소값은 어떻게 구할까?

DP를 이용하는 문제 - 1로 만들기

- Top-down

$f(n)$: n 을 1로 만드는데 필요한 연산횟수의 최소값

연산의 종류는 총 3가지



DP를 이용하는 문제 - 1로 만들기

- Top-down

```
int func(n)
```

```
    if(cache[n] != -1)
```

```
        return cache[n]
```

```
    if(n == 1)
```

```
        return cache[n] = 0
```

```
    int ret = INF
```

```
    if(n%3 == 0)
```

```
        ret = min(ret, func(n/3))
```

```
    if(n%2 == 0)
```

```
        ret = min(ret, func(n/2))
```

```
    return cache[n] = 1+min(ret,func(n-1))
```

- Bottom-up

```
cache[1] = 0
```

```
for(int i=2; i<=N; i++)
```

```
    cache[i] = cache[i-1]+1
```

```
    if(n%3 == 0)
```

```
        cache[i] = min(cache[i], cache[i/3] + 1)
```

```
    if(n%2 == 0)
```

```
        cache[i] = min(cache[i], cache[i/2] + 1)
```

이분탐색을 이용하는 문제 소개

1. 수 찾기 : <https://www.acmicpc.net/problem/1920>

2. 나무 자르기 : <https://www.acmicpc.net/problem/2805>

이분탐색을 이용하는 문제 - 수 찾기

문제 요약 : <https://www.acmicpc.net/problem/1920>

- $1 \leq N, M \leq 100,000$, $A[i]$ 는 int 범위 안에 있다.
- 주어진 배열 A에 M개의 수들이 있는지 알아내시오
- 출력은 각 숫자마다 한줄씩 이루어진다

고려해야 할 사항 :

- 완전탐색의 시간복잡도
- 이분탐색을 쓰기 위한 조건

이분탐색을 이용하는 문제 - 수 찾기

- 완전탐색의 시간복잡도
 - $O(100,000 \times 100,000)$
- 이분탐색을 쓰기 위한 조건
 - 탐색 영역이 일정한 기준으로 정렬 돼있어야 함
 - c++의 `sort()`, `algorithm` 헤더파일에 존재
 - $O(N \log N)$

오름차순 정렬

```
vector<int> list
    list에 숫자 입력
sort(list.begin(), list.end())
```

내림차순 정렬

```
bool compare(int a, int b)
    return a > b
```

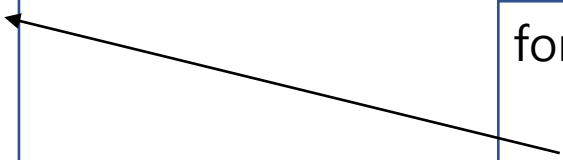
```
vector<int> list
    list에 숫자 입력
sort(list.begin(), list.end(), compare)
```


이분탐색을 이용하는 문제 - 수 찾기

- 실제 구현

```
bool find (int l, int r, int x)
{
    int mid = (l+r)/2
    if(l > r)
        return false
    if(list[mid] == x)
        return true
    else if(list[mid] > x)
        return find(l, mid-1, x)
    else if(list[mid] < x)
        return find(mid+1, r, x)
}
```

```
for(int i=0; i<M; i++)
{
    cin >> x
    cout << find(0, list.size()-1, x) << "\n"
}
```



이분탐색을 이용하는 문제 - 나무 자르기

문제 요약 :

<https://www.acmicpc.net/problem/2805>

- $1 \leq N \leq 1,000,000$, $1 \leq M \leq 2,000,000,000$, $0 \leq \text{나무의 높이} \leq 1,000,000,000$
- N개의 나무들을 일정 높이의 절단기로 잘랐을 때 적어도 M미터의 나무를 얻기 위한 높이의 최대값을 구하시오

고려해야 할 사항 :

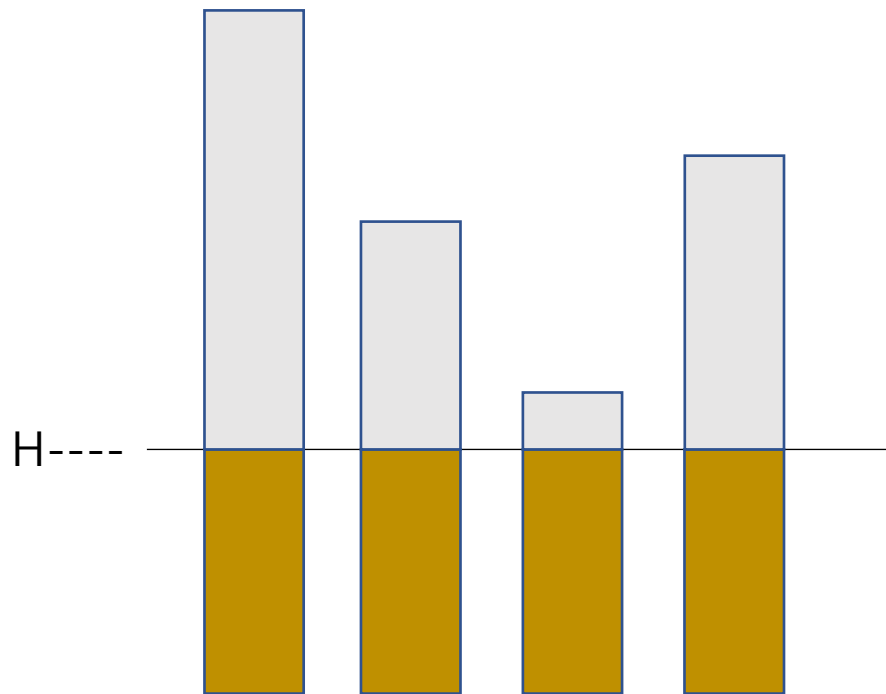
- 완전탐색의 시간복잡도
- 발상의 전환

이분탐색을 이용하는 문제 - 나무 자르기

- 발상의 전환

- H의 높이로 잘랐을 때, 적어도 M미터의 나무를 얻을 수 있나?
- H에 대해서 이분탐색

$$0 \leq H \leq 1,000,000,000$$

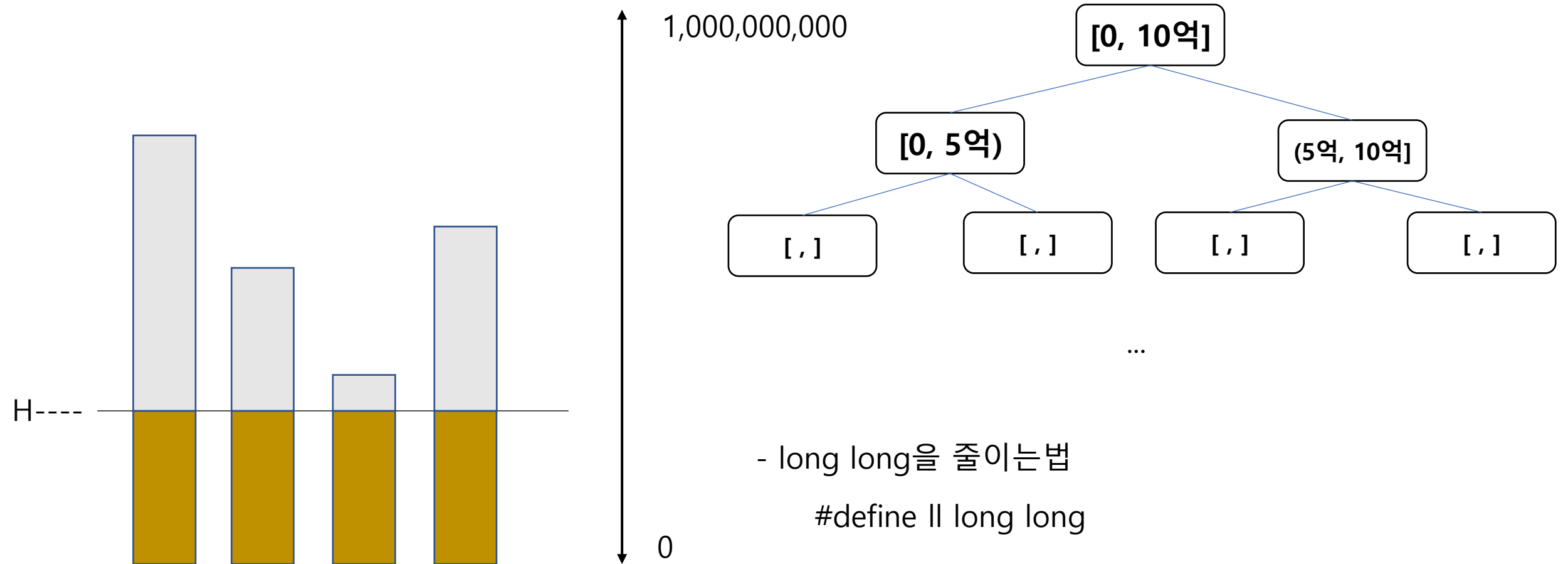


만약 H의 높이로 잘랐을 때,
M미터 이상의 나무를 얻는다면

H-1의 높이로 자를 이유가 있을까?

이분탐색을 이용하는 문제 - 나무 자르기

- 발상의 전환



이분탐색을 이용하는 문제 - 나무 자르기

- 실제 구현

ll bs(ll l, ll r) : H의 범위가 [l,r]일때, M미터 이상의 나무를 베는 H의 최대값을 반환

```
ll bs (ll l, ll r)
ll mid = (l+r)/2
    if(l > r)
        return 0
    else if(l==r)
        return calc(l) >= M ? l : 0
    if(calc(mid) < M)
        return bs(l,mid-1)
    else if(calc[mid] >= M)
        return max(bs(mid+1,r), mid)
```

ll calc(ll h) : 나무들을 h의 높이로 잘랐을 때 얻을 수 있는 나무의 합 반환

```
ll calc(ll h)
ll sum = 0
for(int i=0; i<N; i++)
    sum += max(list[i] - h, 0)
return sum
```



도전 문제!

1. 정수 삼각형 : <https://www.acmicpc.net/problem/1932>
2. 1로 만들기 2 : <https://www.acmicpc.net/problem/12852>
3. 가장 긴 증가하는 부분 수열: <https://www.acmicpc.net/problem/11053>
4. 입국심사 : <https://www.acmicpc.net/problem/3079>

● ● 자유로운 질문 및 토의!