



## 제6장

# 순환 (recursion)

**순환이란?**

# Recursion

## ❶ 자기 자신을 호출하는 함수

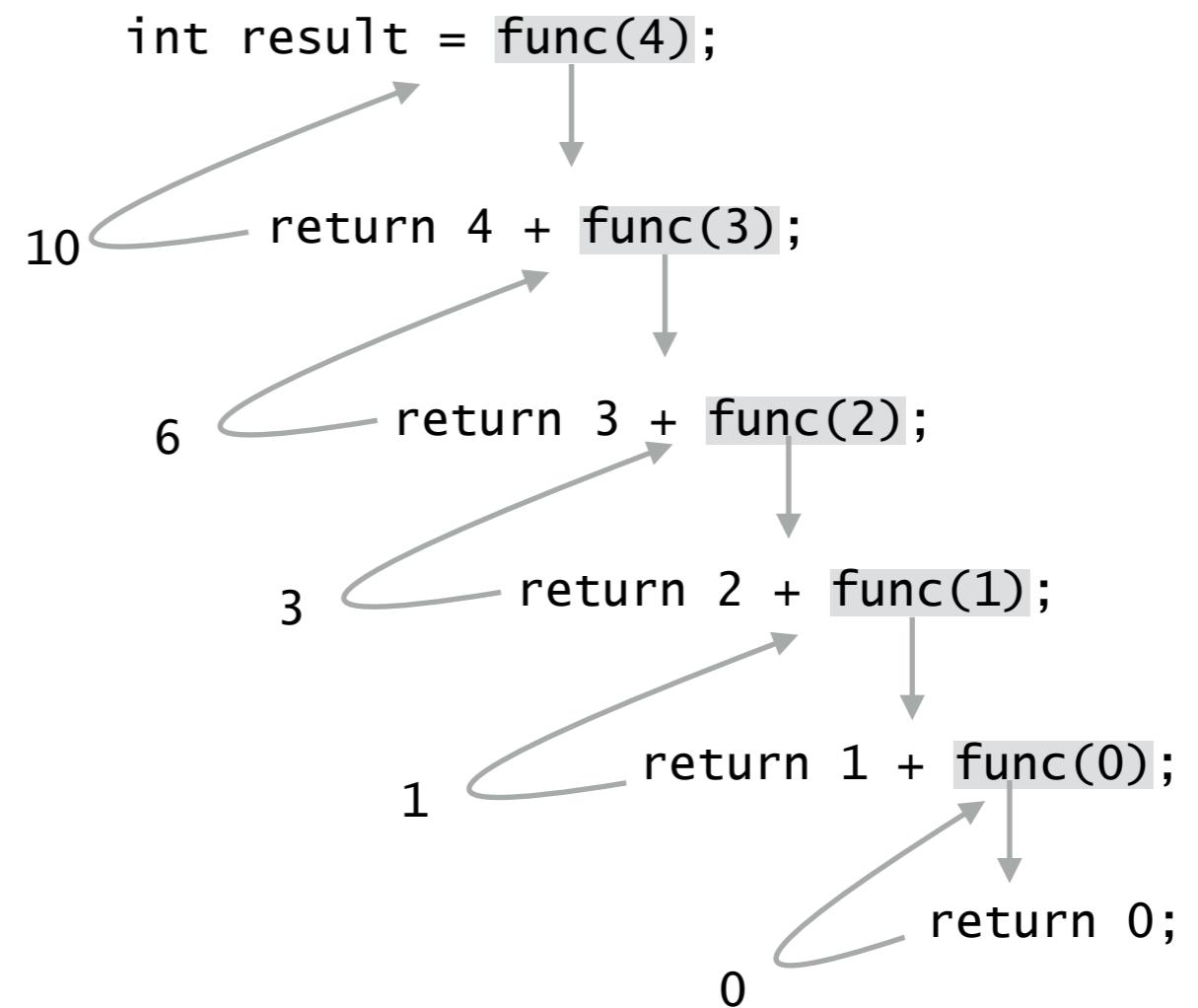
```
void func( ... )  
{  
    ...  
    func( ... );  
    ...  
}
```

# recursion은 항상 무한루프에 빠질까?

```
int main() {  
    int result = func(4);  
}
```

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

1~n까지의 합을 구한다.



# 무한루프에 빠지지 않으려면?

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

Base case: 적어도 하나의 recursion에 빠지지 않는 경우가 존재해야 한다.

Recursive case: recursion을 반복하다보면 결국 base case로 수렴해야 한다.

# recursion의 해석

이 함수의 mission은 0~n까지의 합을 구하는 것이다.

```
int func(int n) {  
    if (n==0)          n=0이라면 합은 0이다.  
        return 0;  
    else  
        return n + func(n-1);  
}
```

n이 0보다 크다면 0에서 n까지의 합은 0에서 n-1까지의 합에 n을 더한 것이다.

# 순환함수와 수학적귀납법

정리: `func(int n)`은 음이 아닌 정수  $n$ 에 대해서 0에서  $n$ 까지의 합을 올바로 계산한다.

증명:

1.  $n=0$ 인 경우:  $n=0$ 인 경우 0을 반환한다. 올바르다.
2. 임의의 양의 정수  $k$ 에 대해서  $n < k$ 인 경우 0에서  $n$ 까지의 합을 올바르게 계산하여 반환하고 가정하자.
3.  $n=k$ 인 경우: `func`은 먼저 `func(k-1)` 호출하는데 2번의 가정에 의해서 0에서  $k-1$ 까지의 합이 올바로 계산되어 반환된다. 메서드 `func`은 그 값에  $n$ 을 더해서 반환하므로 결국 0에서  $k$ 까지의 합을 올바로 계산하여 반환한다.

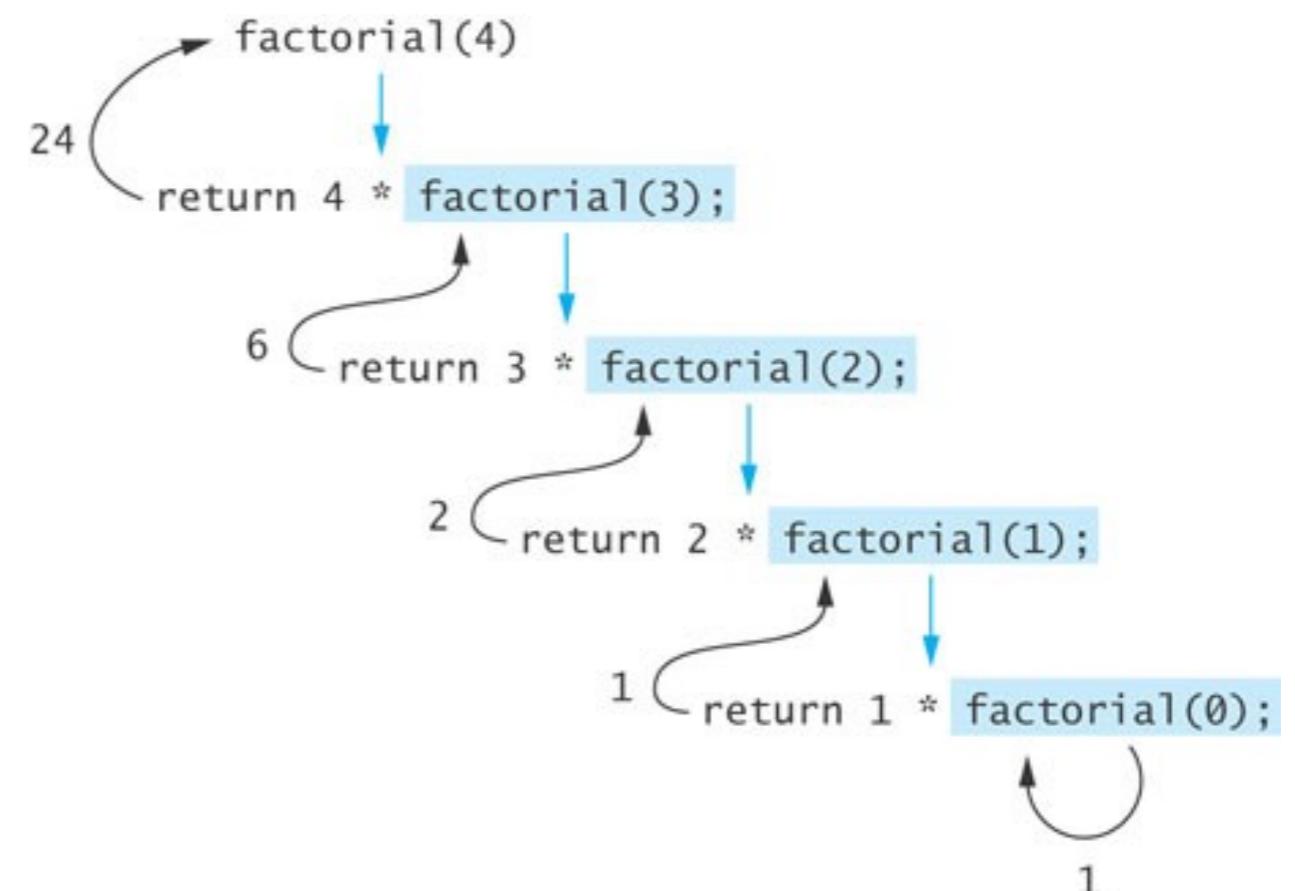
## Factorial: n!

$$0! = 1$$

$$n! = n \times (n-1)! \quad n > 0$$

# Factorial: n!

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



시간복잡도  $O(n)$

# 순환함수와 수학적귀납법

정리: `factorial(int n)`은 음이 아닌 정수  $n$ 에 대해서  $n!$ 을 올바로 계산한다.

증명:

1.  $n=0$ 인 경우:  $n=0$ 인 경우 1을 반환한다. 올바르다.
2. 임의의 양의 정수  $k$ 에 대해서  $n < k$ 인 경우  $n!$ 을 올바르게 계산한다고 가정하자.
3.  $n=k$ 인 경우를 고려해보자. `factorial`은 먼저 `factorial(k-1)` 호출하는데 2번의 가정에 의해  $(k-1)!$ 이 올바로 계산되어 반환된다. 따라서 메서드 `factorial`은  $k * (k-1)! = k!$ 을 반환한다.

$x^n$

$$\begin{aligned}x^0 &= 1 \\x^n &= x \cdot x^{n-1} \quad \text{if } n > 0\end{aligned}$$

```
double power(double x, int n) {  
    if (n==0)  
        return 1;  
    else  
        return x*power(x, n - 1);  
}
```

시간복잡도  $O(n)$

# Fibonacci Number

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2} \quad n>1$$

# Fibonacci Number

```
int fibonacci(int n) {  
    if (n<2)  
        return n;  
    else  
        return fibonacci(n-1) + fibonacci(n-2);  
}
```

시간복잡도?

# 최대공약수: Euclid Method

```
double gcd(int m, int n) {  
    if (m<n) {  
        int tmp=m; m=n; n=tmp; // swap m and n  
    }  
    if (m%n==0)  
        return n;  
    else  
        return gcd(n, m%n);  
}
```

$m \geq n$ 인 두 양의 정수  $m$ 과  $n$ 에 대해서  $m$ 이  $n$ 의 배수이면  $\text{gcd}(m, n)=n$ 이고,  
그렇지 않으면  $\text{gcd}(m, n)=\text{gcd}(n, m \% n)$ 이다.

## Euclid Method: 좀더 간단한 버전

$$\text{gcd}(p, q) = \begin{cases} p & \text{if } q=0 \\ \text{gcd}(q, p\%q) & \text{otherwise.} \end{cases}$$

```
int gcd(int p, int q) {  
    if (q==0)  
        return p;  
    else  
        return gcd(q, p%q);  
}
```

# **Recursive Thinking**

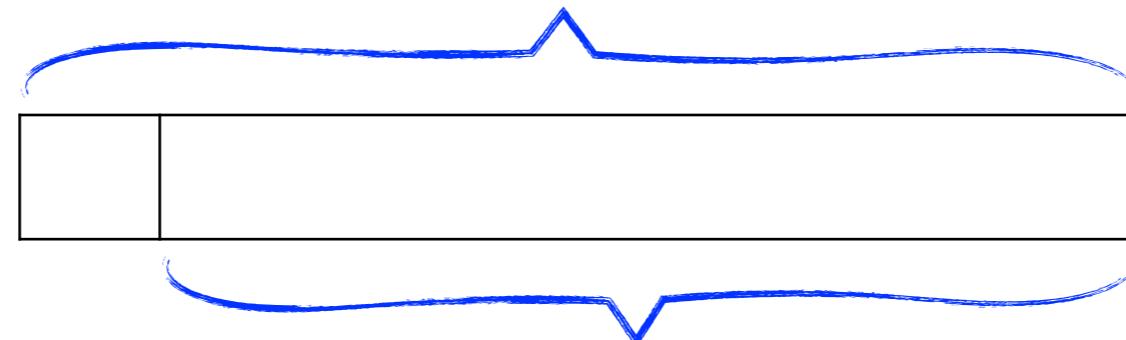
순환적으로 사고하기

# Recursion은 수학함수 계산에만 유용한가?

수학함수 뿐만 아니라 다른 많은 문제들을  
recursion으로 해결할 수 있다.

# 문자열의 길이 계산

이 문자열의 길이는

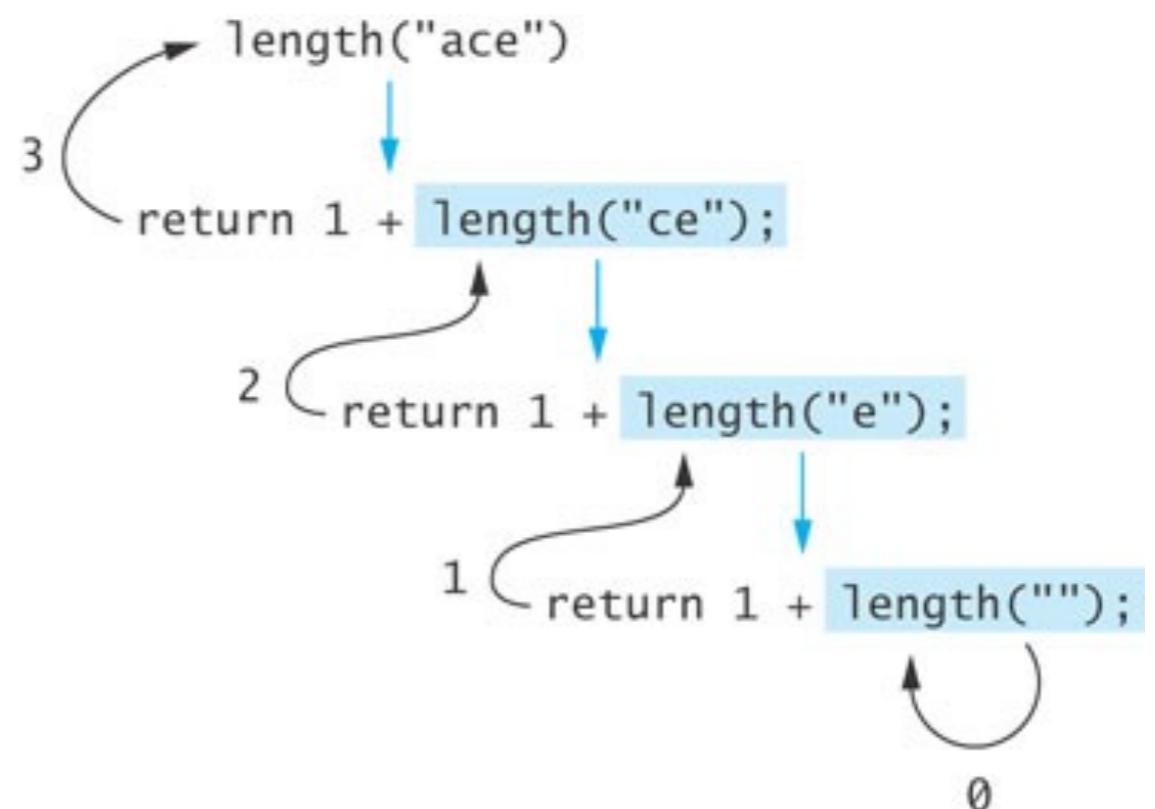


이 문자열의 길이 + 10이다.

```
if the string is empty  
    return 0;  
else  
    return 1 + the length of the string that excludes the first character;
```

# 문자열의 길이 계산

```
int length(char *str) {  
    if (*str == '\0')  
        return 0;  
    else  
        return 1 + length(str+1);  
}
```



## 문자열의 프린트

```
void printChars(char *str) {  
    if (*str == '\0')  
        return;  
    else {  
        printf("%c", *str);  
        printChars(str+1);  
    }  
}
```

# 문자열을 뒤집어 프린트

(1) 이 문자열을 뒤집어 프린트하려면



(3) 마지막으로 첫 글짜를 프린트 한다.

(2) 먼저 이 문자열을 뒤집어 프린트 한 후

```
void printCharsReverse(char *str) {  
    if (*str=='\0')  
        return;  
    else {  
        printCharsReverse(str+1);  
        printf("%c", *str);  
    }  
}
```

# 순차 탐색

data[0]에서 data[n-1] 사이에서 target을 검색한다.  
존재하면 배열 인덱스, 존재하지 않으면 -1을 반환한다.

```
int search(int data[], int n, int target,) {  
    if (n <= 0)  
        return -1;  
    else if (target==items[n-1])  
        return n-1;  
    else  
        return search(data, n-1, target);  
}
```

# 최대값 찾기

data[0]에서 data[n-1] 사이에서 최대값을 찾아  
반환한다.

```
int findMax(int n, int data[]) {  
    if (n==1)  
        return data[0];  
    else  
        return max(data[n-1], findMax(n-1, data));  
}
```

## 2진수로 변환하여 출력

음이 아닌 정수 n을 이진수로 변환하여 인쇄한다.

```
void printInBinary(int n) {  
    if (n<2)  
        printf("%d", n);  
    else {  
        printInBinary(n/2);  
        printf("%d", n%2);  
    }  
}
```

n을 2로 나눈 몫을 먼저 2진수로 변환하여 인쇄한 후

n을 2로 나눈 나머지를 인쇄한다.

# Disjoint Sets

배열 A의  $A[0], \dots, A[m-1]$ 과 배열 B의  $B[0], \dots, B[n-1]$ 에 정수들이 정렬되어 저장되어 있을 때 두 배열의 정수들이 disjoint한지 검사한다.

```
bool isDisjoint(int m, int A[], int n, int B[])
{
    if (m<0 || n<0)
        return true;
    else if (A[m-1]==B[n-1])
        return false;
    else if (A[m-1]>B[n-1])
        return isDisjoint(m-1, A, n, B);
    else
        return isDisjoint(m, A, n-1, B);
}
```

# Recursion vs. Iteration

- 모든 순환함수는 반복문(iteration)으로 변경 가능
- 그 역도 성립함. 즉 모든 반복문은 recursion으로 표현 가능함
- 순환함수는 복잡한 알고리즘을 단순하고 알기 쉽게 표현하는 것을 가능하게 함
- 하지만 함수 호출에 따른 오버헤드가 있음 (매개변수 전달, 액티베이션 프레임 생성 등)

# 순환적 알고리즘 설계

- 적어도 하나의 base case, 즉 순환되지 않고 종료되는 case가 있어야 함
- 모든 case는 결국 base case로 수렴해야 함

# 순환적 알고리즘 설계

암시적(implicit) 매개변수를  
명시적(explicit) 매개변수로 바꾸어라.

## 이진 탐색: Iterative Version

```
int binarySearch(int data[], int n, int target) {  
    int begin = 0, end = n-1;  
    while (begin<=end) {  
        int middle = (begin+end)/2;  
        if (data[middle] == target)  
            return middle;  
        else if (data[middle] > target)  
            end = middle - 1;  
        else  
            begin = middle + 1;  
    }  
    return -1;  
}
```

# 이진탐색: Recursion

items[begin]에서 items[end] 사이에서 target을 검색한다.

```
int binarySearch(int data[], int target, int begin, int end) {  
    if (begin>end)  
        return -1;  
    else {  
        int middle = (begin+end)/2;  
        if (data[middle] == target)  
            return middle;  
        else if (data[middle] > target)  
            return binarySearch(data, target, begin, middle-1);  
        else  
            return binarySearch(data, target, middle+1, end);  
    }  
}
```

data[begin]에서 data[end] 사이에서 합이 K가 되는 쌍이 존재하는지 검사한다. 데이터는 오름차순으로 정렬되어 있다고 가정한다.

```
bool twoSum(int data[], int begin, int end, int K) {  
    if (begin>=end)  
        return false;  
    else {  
        if (data[begin]+data[end] == K)  
            return true;  
        else if (data[begin]+data[end]<K)  
            return twoSum(data, begin+1, end, K);  
        else  
            return twoSum(data, begin, end-1, K);  
    }  
}
```

만약 중복 선택이 가능하다면 =을 빼면 됨

# **Towers of Hanoi**

하노이 타워

# Towers of Hanoi

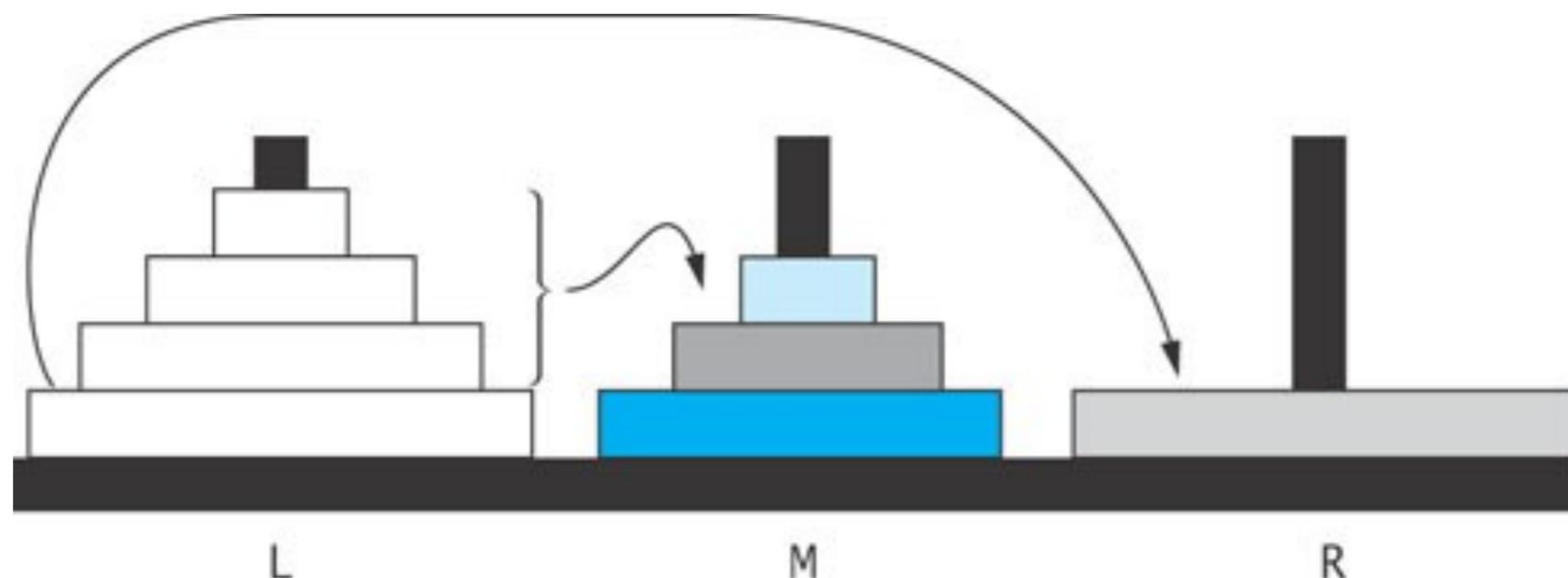


## Questions

- ⌚ Is it Possible ?
- ⌚ How many movements ?
- ⌚ How ?

## Is it possible ?

- $n=1$ 인 경우: 당연히 가능.
- $n < k$ 인 경우: 가능하다고 가정하자.
- $n=k$ 인 경우: 먼저 가장 큰 하나를 제외한 나머지  $k-1$ 개를 M으로 옮긴다. 다음으로 가장 큰 디스크를 L에서 R로 옮긴다. 그런 다음 좀 전에 M으로 옮겨둔  $k-1$ 개를 R로 옮긴다.



## How many movements ?

- $T(n)$ 을  $n$ 개의 디스크를 L에서 R로 옮길 때 필요한 movement의 횟수라고 하자.

$$T(1) = 1$$

$$T(n) = T(n-1)+1+T(n-1) = 2T(n-1)+1, \quad n>1 \text{인 경우}$$

- 점화식을 풀어보면  $T(n) = 2^n - 1$

# Recursive Algorithm

Move n Disks from the Starting Peg to the Destination Peg

if n is 1

    move disk 1 from the starting peg to the destination peg;

else

    move the top n - 1 disks from the starting peg to the temporary peg;

    move disk n from the starting peg to the destination peg;

    move the top n - 1 disks from the temporary peg to the destination peg;

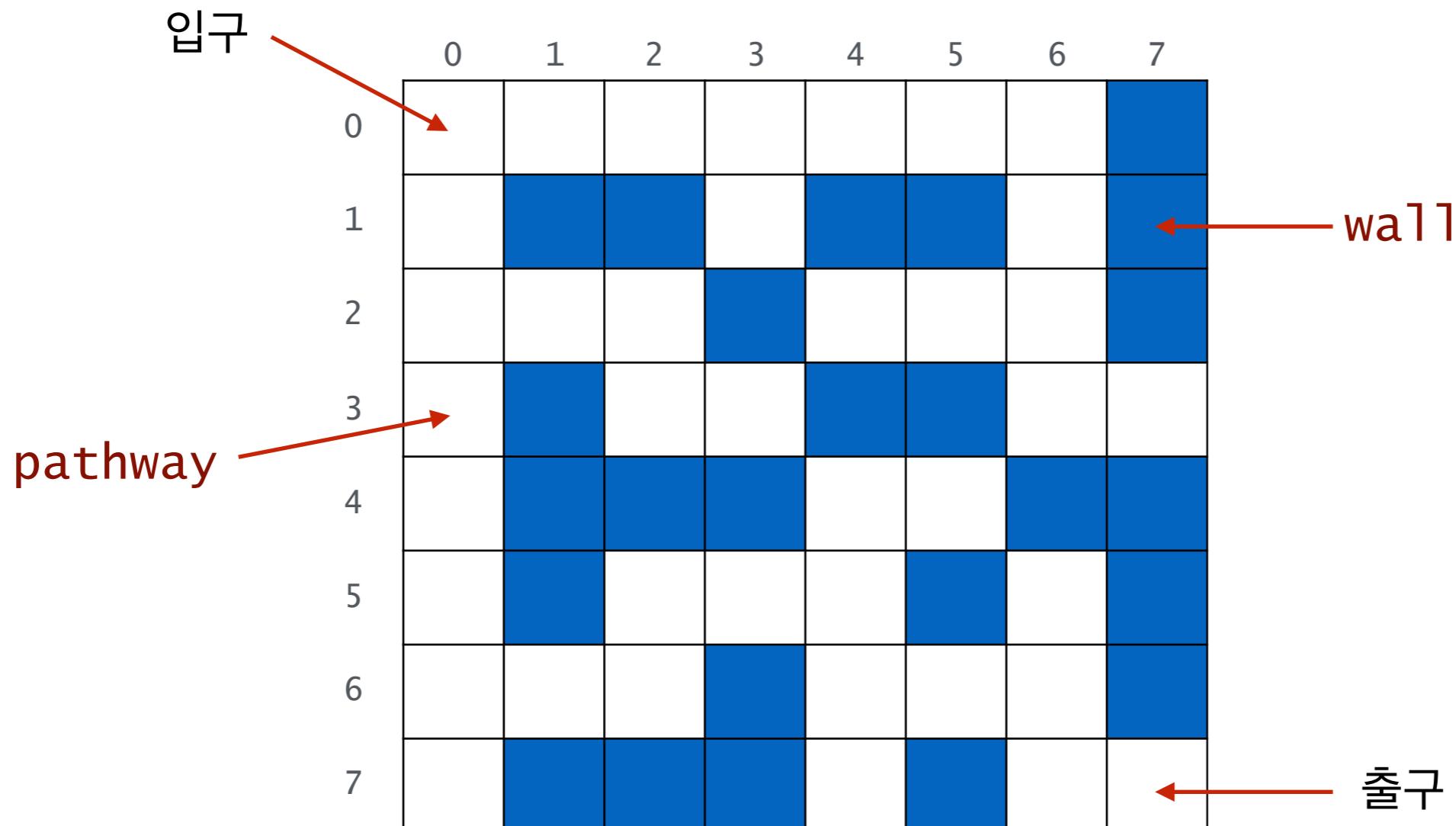
# Recursive Algorithm

```
void showMoves(int n, char start, char dest, char temp)
{
    if (n==1)
        printf("Move disk 1 from peg %c to peg %d\n", start, dest);
    else{
        showMoves(n-1, start, temp, dest);
        printf("Move disk %d from peg %c to peg %d\n", n, start, dest);
        showMoves(n-1, temp, dest, start);
    }
}
```

# Maze

미로찾기

# 미로찾기



# Recursive Thinking

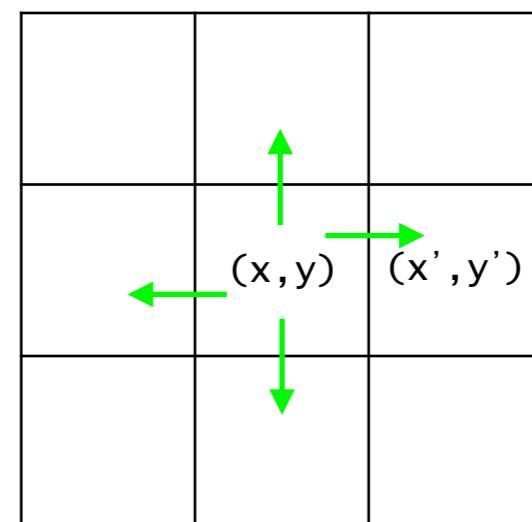
현재 위치에서 출구까지 가는 경로가 있으려면

- 1) 현재 위치가 출구이거나 혹은
- 2) 이웃한 셀들 중 하나에서 출구까지 가는 경로가 있거나

# 미로찾기(Decision Problem)

답이 yes or no인 문제

```
boolean findPath(x,y)
    if (x,y) is the exit
        return true;
    else
        for each neighbouring cell (x',y') of (x,y) do
            if (x',y') is a pathway cell
                if findPath(x',y')
                    return true;
    return false;
```



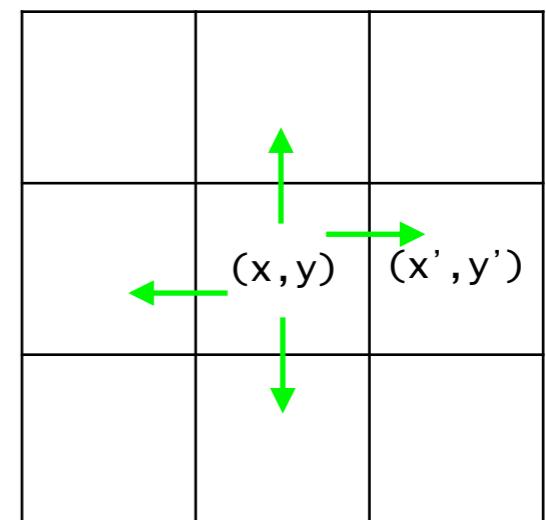
# Recursive Thinking

현재 위치에서 이미 가본 곳을 다시 지나지 않고 출구까지 가는 경로가 있으려면

- 1) 현재 위치가 출구이거나 혹은
- 2) 이웃한 셀들 중 하나에서 이미 가본 곳을 다시 지나지 않고 출구까지 가는 경로가 있거나

# 미로찾기

```
boolean findPath(x,y)
    if (x,y) is the exit
        return true;
    else
        mark (x,y) as visited;
        for each neighbouring cell (x',y') of (x,y) do
            if (x',y') is a pathway cell and not visited
                if findPath(x',y')
                    return true;
    return false;
```



```
boolean findPath(x,y)
    if (x,y) is either on the wall or a visited cell
        return false;
    else if (x,y) is the exit
        return true;
    else
        mark (x,y) as a visited cell;
        for each neighbouring cell (x',y') of (x,y) do
            if findPath(x',y')
                return true;
        return false;
```

# 미로찾기

```
int N=8;  
int maze[][][8] = {  
    {0, 0, 0, 0, 0, 0, 0, 1},  
    {0, 1, 1, 0, 1, 1, 0, 1},  
    {0, 0, 0, 1, 0, 0, 0, 1},  
    {0, 1, 0, 0, 1, 1, 0, 0},  
    {0, 1, 1, 1, 0, 0, 1, 1},  
    {0, 1, 0, 0, 0, 1, 0, 1},  
    {0, 0, 0, 1, 0, 0, 0, 1},  
    {0, 1, 1, 1, 0, 1, 0, 0}  
};
```

```
#define PATHWAY_COLOUR 0          // white  
#define WALL_COLOUR 1            // blue  
#define BLOCKED_COLOUR 2          // red  
#define PATH_COLOUR 3            // green
```

- PATH\_COLOR: visited이며 아직 출구로 가는 경로가 될 가능성이 있는 cell
- BLOCKED\_COLOR: visited이며 출구까지의 경로상에 있지 않음이 밝혀진 cell

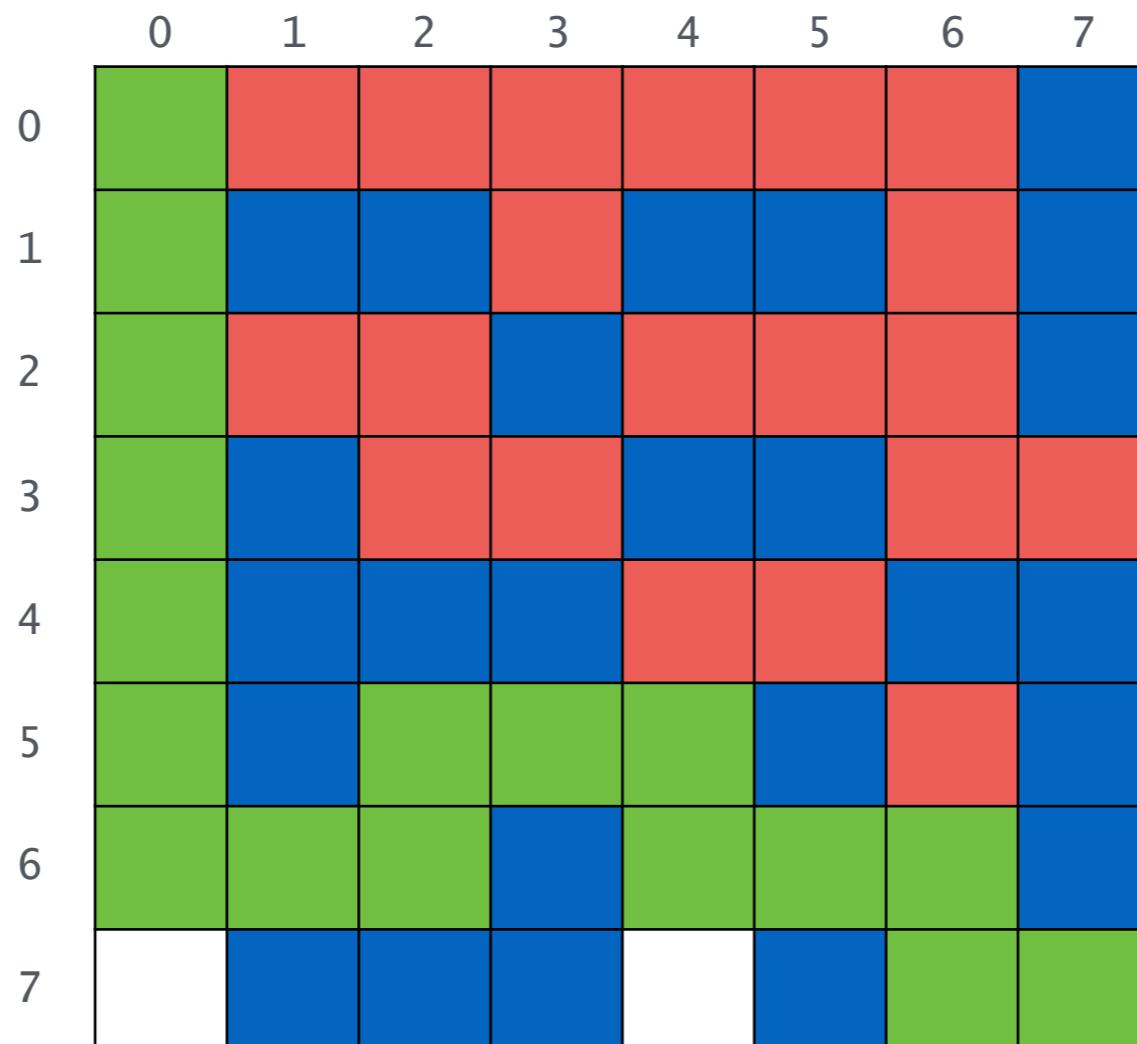
# 미로찾기

```
bool findMazePath(int x, int y) {  
    if (x<0 || y<0 || x>=N || y>=N || maze[x][y] != PATHWAY_COLOUR)  
        return false;  
    else if (x==N-1 && y==N-1) {  
        maze[x][y] = PATH_COLOUR;  
        return true;  
    }  
}
```

# 미로찾기

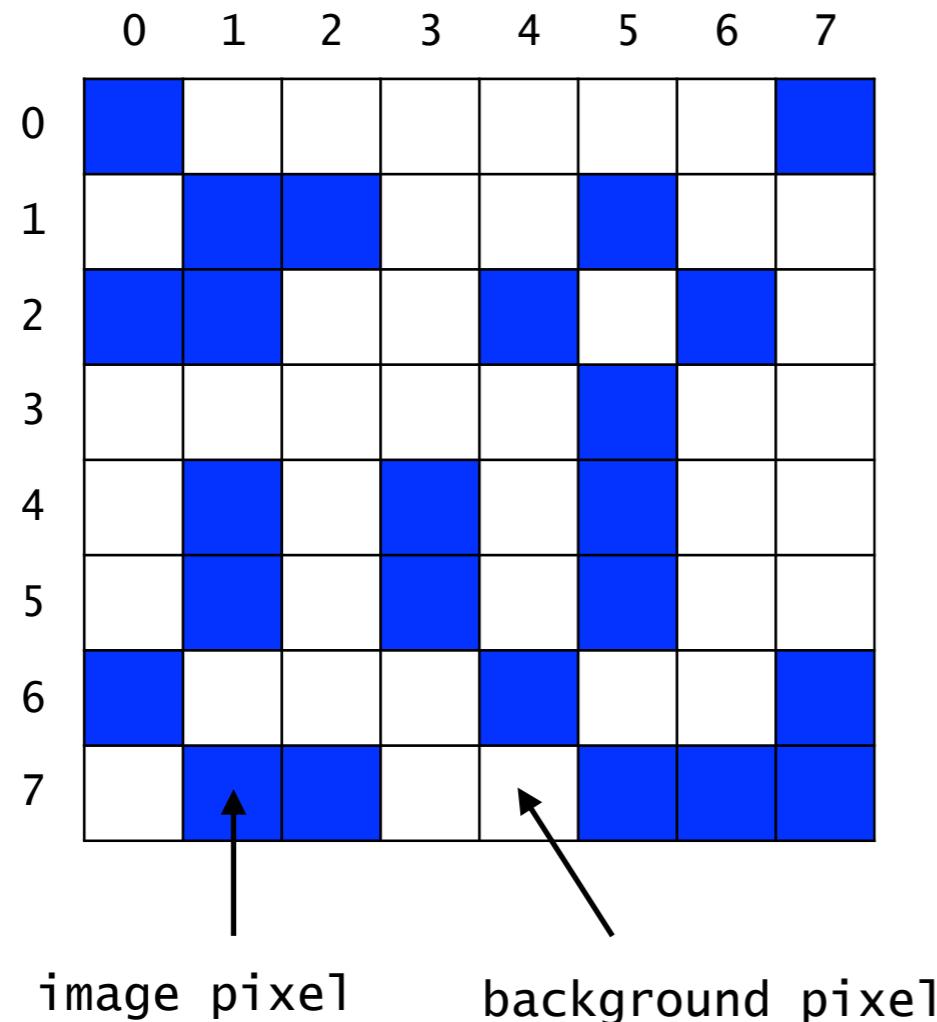
```
else {
    maze[x][y] = PATH_COLOUR;
    if (findMazePath(x-1,y) || findMazePath(x,y+1)
        || findMazePath(x+1,y) || findMazePath(x,y-1)) {
        return true;
    }
    maze[x][y] = BLOCKED_COLOUR;      // dead end
    return false;
}

int main() {
    printMaze();
    findMazePath(0,0);
    printMaze();
}
```



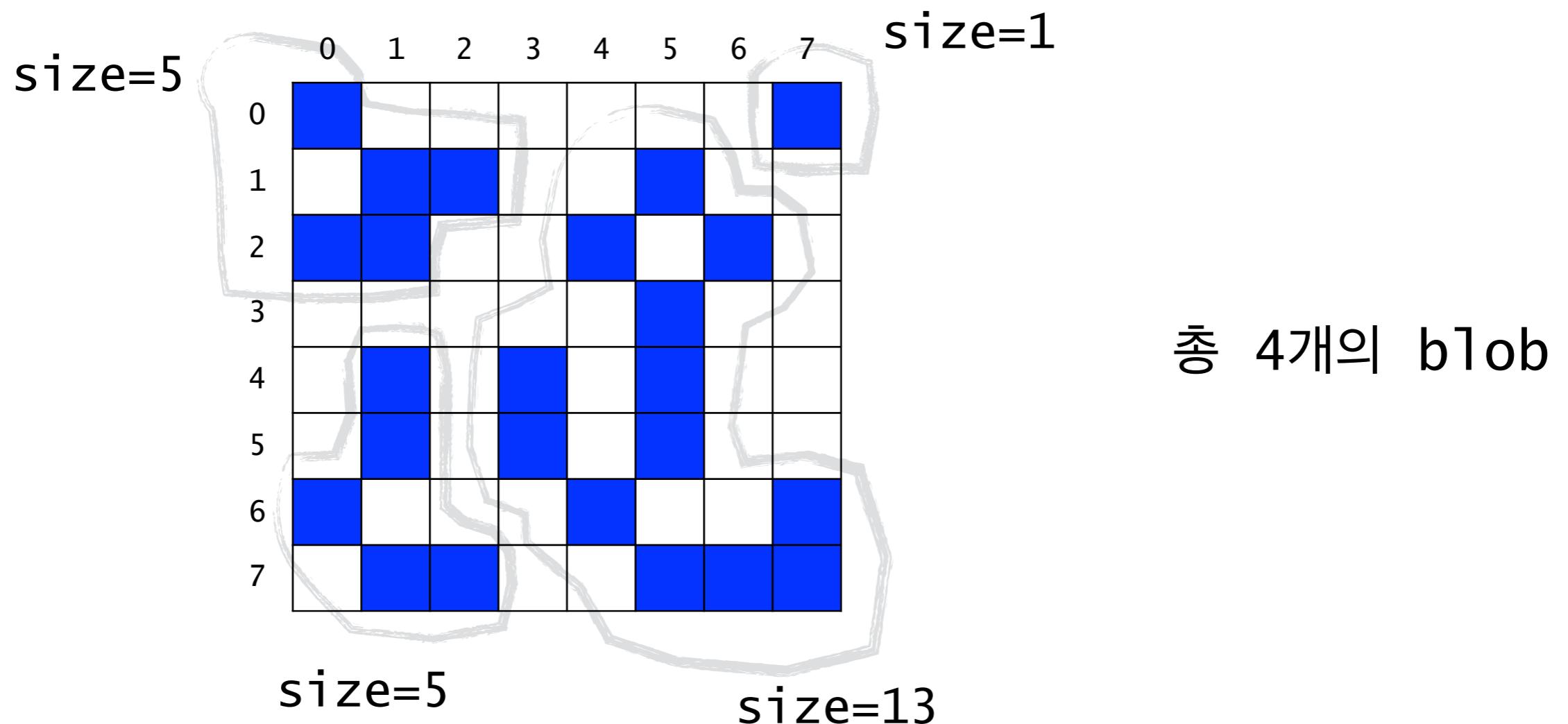
# **Counting Cells in a Blob**

# Counting Cells in a Blob



- Binary 이미지
- 각 픽셀은 background pixel이거나 혹은 image pixel
- 서로 연결된 image pixel들의 집합을 blob이라고 부름
- 상하좌우 및 대각방향으로도 연결된 것으로 간주

## Counting Cells in a Blob



# Counting Cells in a Blob

## • 입력:

- $N \times N$  크기의 2차원 그리드(grid)
- 하나의 좌표 (x,y)

## • 출력:

- 픽셀 (x,y)가 포함된 blob의 크기,
- (x,y)가 어떤 blob에도 속하지 않는 경우에는 0

# Recursive Thinking

현재 픽셀이 이 속한 blob의 크기를 카운트하려면

현재 픽셀이 image color가 아니라면

0을 반환한다

현재 픽셀이 image color라면

먼저 현재 픽셀을 카운트한다 ( $\text{count}=1$ ).

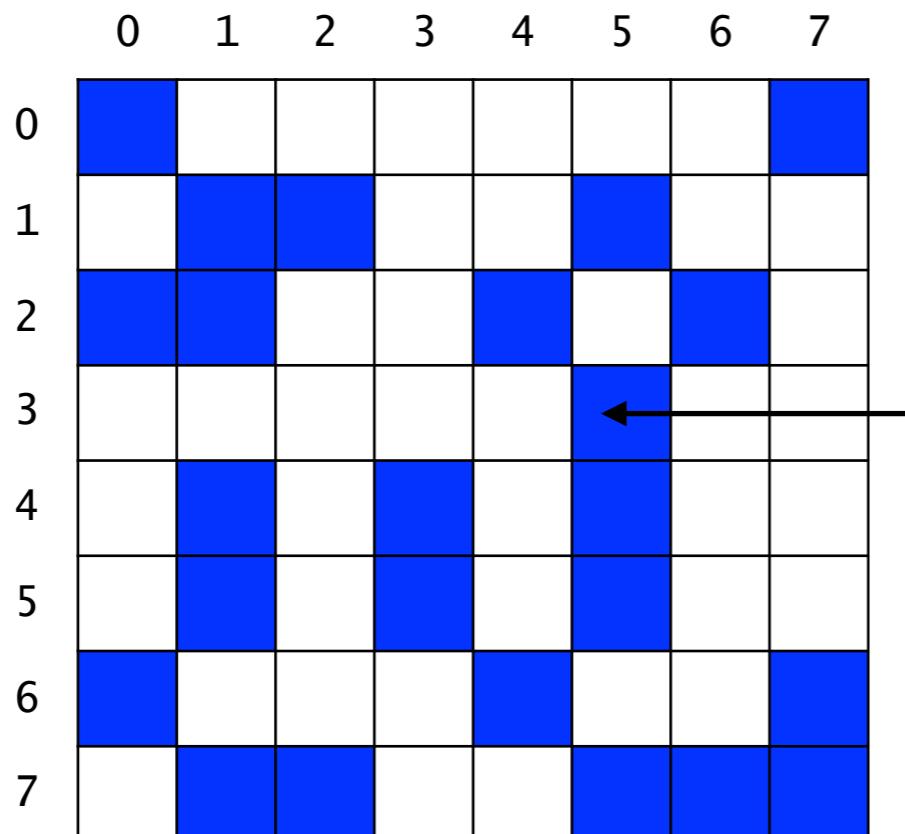
현재 픽셀이 중복 카운트되는 것을 방지하기 위해 다른 색으로 칠한다.

현재 픽셀에 이웃한 모든 픽셀들에 대해서

그 픽셀이 속한 blob의 크기를 카운트하여 카운터에 더해준다.

카운터를 반환한다.

# 순환적 알고리즘 (1)



$x=5$ ,  $y=3$ 이라고 가정. 즉 이  
픽셀이 포함된 blob의 크기를  
계산하는 것이 목적이다.

count = 0

## 순환적 알고리즘 (2)

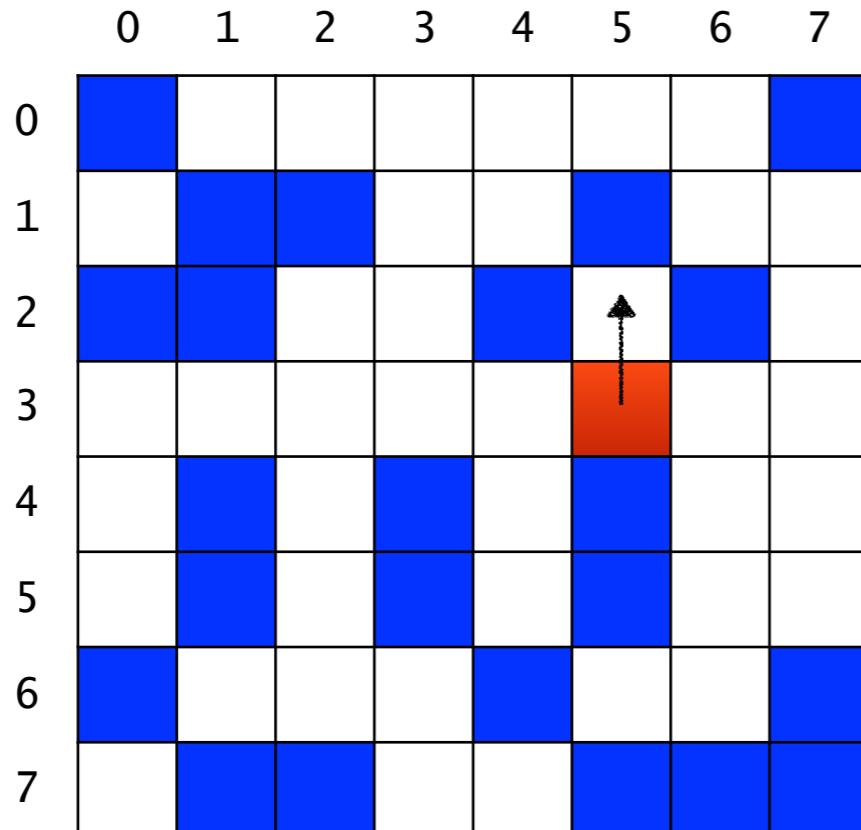
	0	1	2	3	4	5	6	7
0	Blue	White	White	White	White	White	White	Blue
1	White	Blue	Blue	White	White	Blue	White	White
2	Blue	Blue	White	White	Blue	White	Blue	White
3	White	White	White	White	White	Red	White	White
4	White	Blue	White	Blue	White	Blue	White	White
5	White	Blue	White	Blue	White	Blue	White	White
6	Blue	White	White	White	Blue	White	White	Blue
7	White	Blue	Blue	White	White	Blue	Blue	Blue

먼저 현재 `cell`을 다른 색으로 칠하고 `count`를 1증가한다. 이렇게 색칠하는 것은 이 픽셀이 중복 `count`되는 것을 방지하기 위해서이다.

`count = 1`

## 순환적 알고리즘 (3)

인접한 8개의 픽셀 각각에 대해서 순서대로 그 픽셀이 포함된 blob의 크기를 count 한다. 북, 북동, 동, 동남, ... 이런 순서로 고려한다.

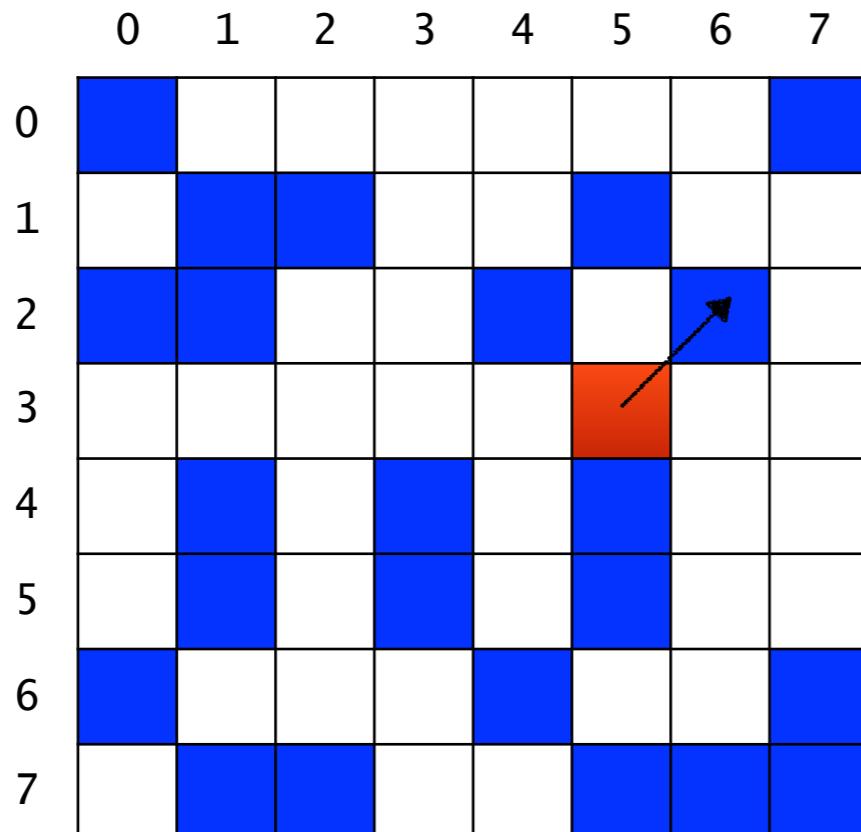


북쪽 픽셀이 포함된 blob의 크기는 0이다. 따라서 count값은 변화없다.

count = 1

## 순환적 알고리즘 (4)

인접한 8개의 픽셀 각각에 대해서 순서대로 그 픽셀이 포함된 blob의 크기를 count 한다. 북, 북동, 동, 동남, ... 이런 순서로 고려한다.

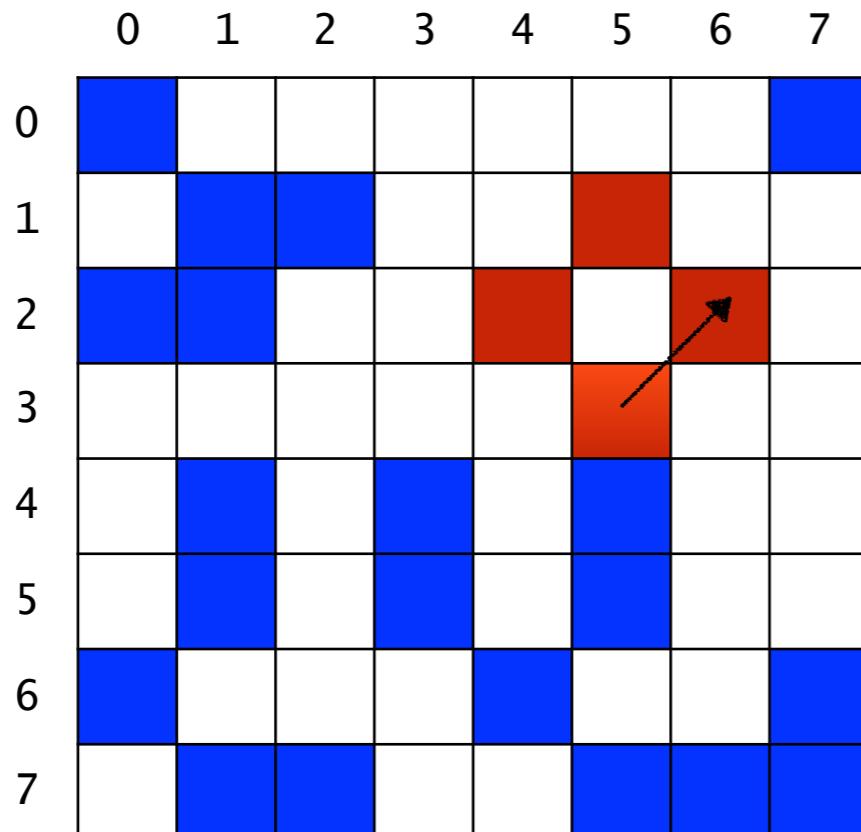


북동쪽 픽셀이 속한 blob을 count하고, count된 픽셀들을 색칠한다.

count = 1

## 순환적 알고리즘 (5)

인접한 8개의 픽셀 각각에 대해서 순서대로 그 픽셀이 포함된 blob의 크기를 count 한다. 북, 북동, 동, 동남, ... 이런 순서로 고려한다.

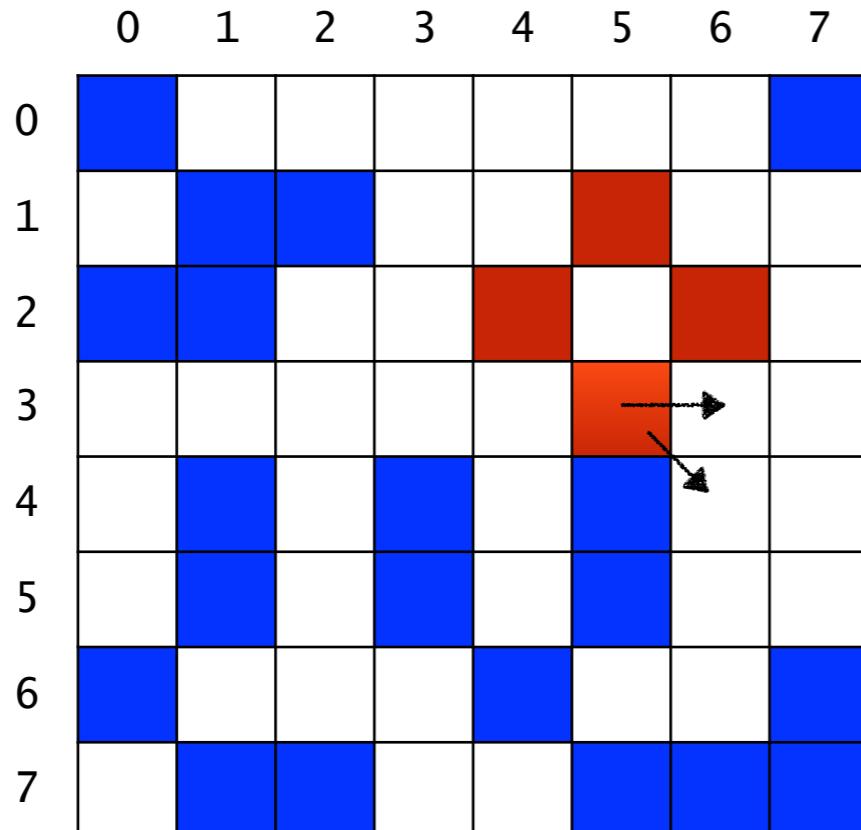


3개의 픽셀이 이 blob에 속한다.

$$\text{count} = 1+3=4$$

## 순환적 알고리즘 (5)

인접한 8개의 픽셀 각각에 대해서 순서대로 그 픽셀이 포함된 blob의 크기를 count 한다. 북, 북동, 동, 동남, ... 이런 순서로 고려한다.

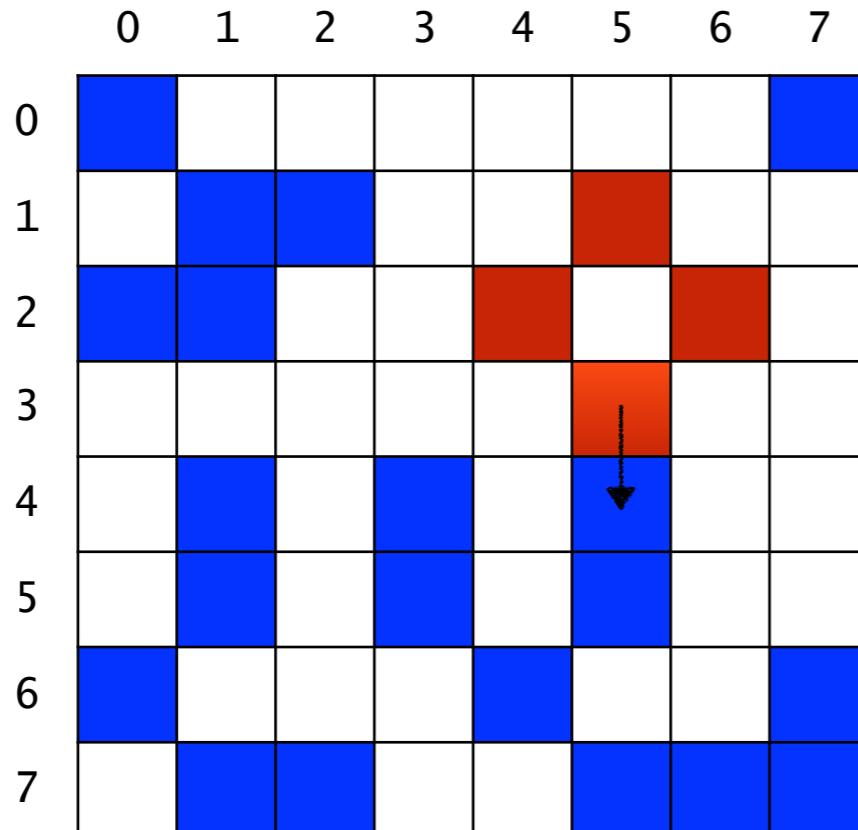


동쪽과 남동쪽 픽셀이 포함된 blob의 크기는 0이다. 따라서 count값은 변화없다.

count = 4

## 순환적 알고리즘 (6)

인접한 8개의 픽셀 각각에 대해서 순서대로 그 픽셀이 포함된 blob의 크기를 count 한다. 북, 북동, 동, 동남, ... 이런 순서로 고려한다.

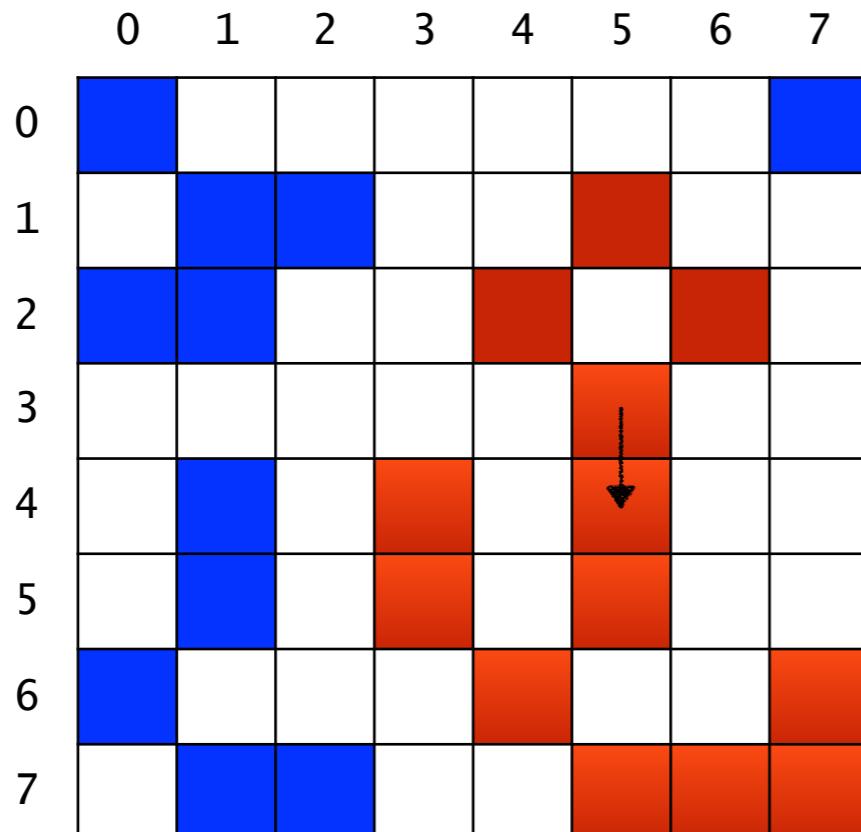


이제 남쪽 픽셀이 속한 blob을 count할 차례이다.

count = 4

## 순환적 알고리즘 (7)

인접한 8개의 픽셀 각각에 대해서 순서대로 그 픽셀이 포함된 blob의 크기를 count 한다. 북, 북동, 동, 동남, ... 이런 순서로 고려한다.

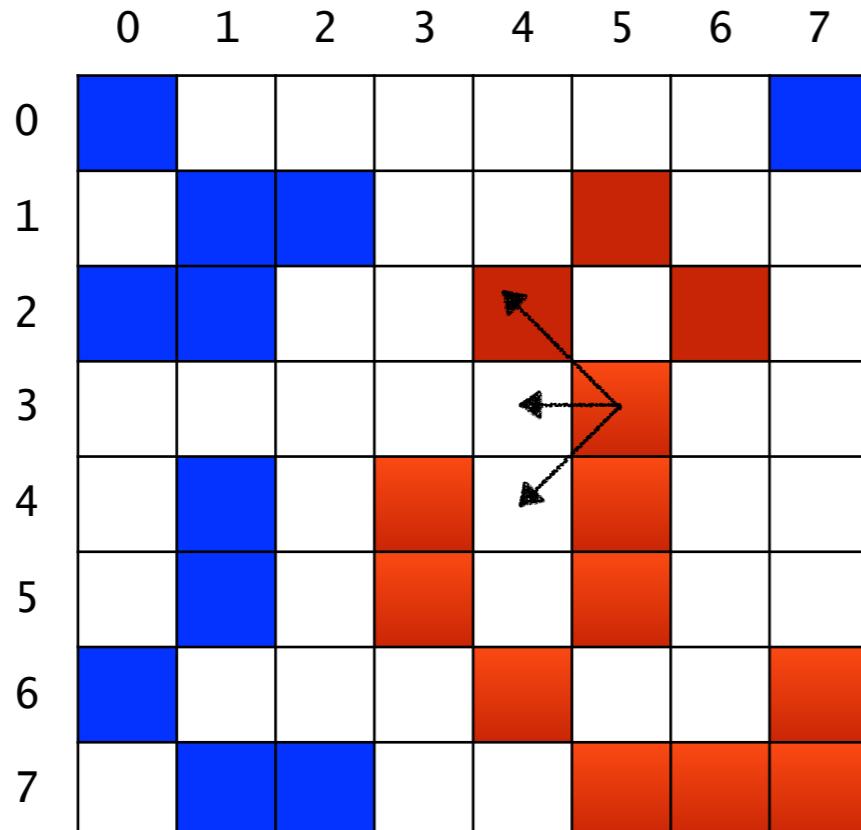


남쪽 픽셀이 속한 blob의 크기는 9이다. 카운트하고 색칠한다.

$$\text{count} = 4 + 9 = 13$$

## 순환적 알고리즘 (7)

인접한 8개의 픽셀 각각에 대해서 순서대로 그 픽셀이 포함된 blob의 크기를 count 한다. 북, 북동, 동, 동남, ... 이런 순서로 고려한다.



남서, 서, 북서 방향의 픽셀이 속한 blob은 없거나 혹은 이미 카운트되었다.

count = 13

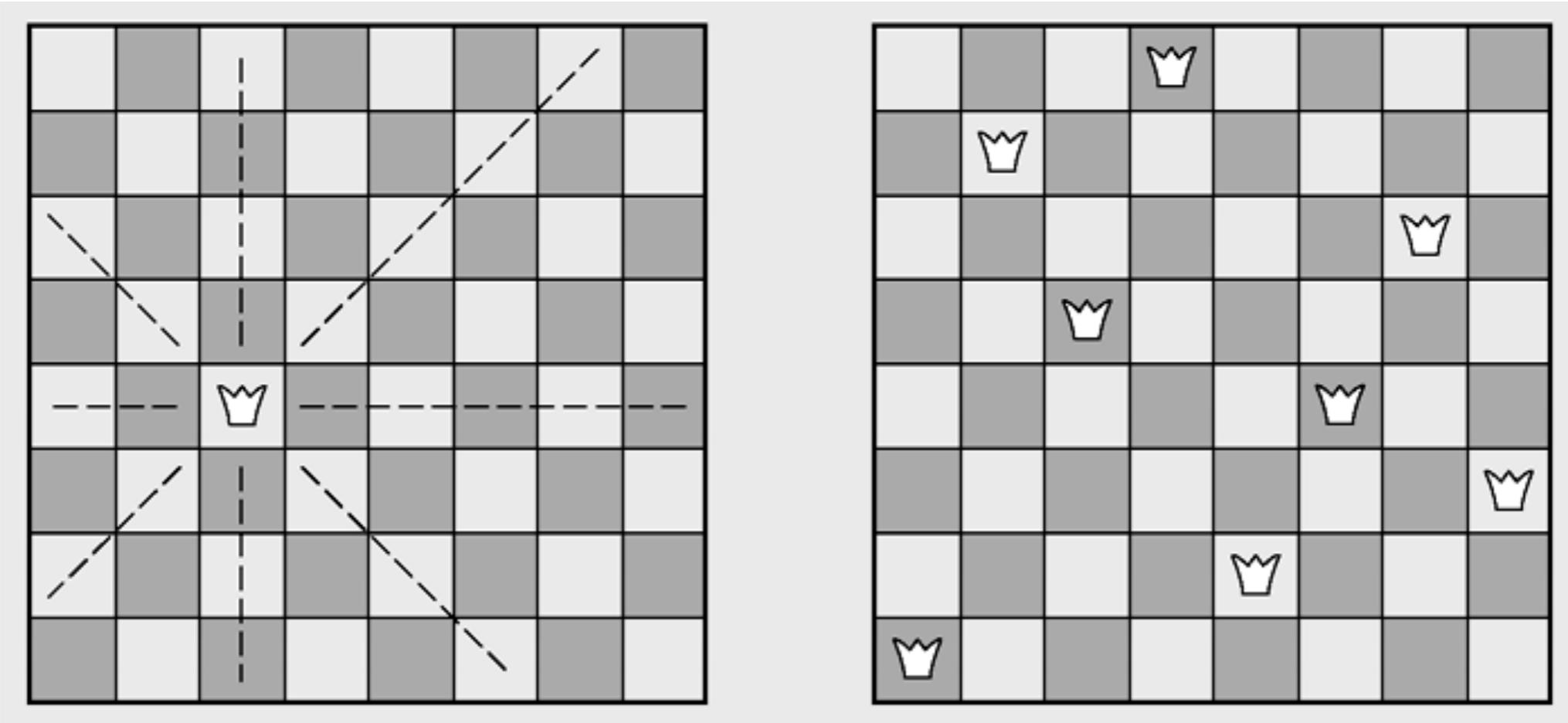
# Counting Cells in a Blob

```
#define BACKGROUND_COLOR 0
#define IMAGE_COLOR 1
#define ALREADY_COUNTED 2

int countCells(int x, int y) {
    /* 연습문제로 남겨 둠 */
}
```

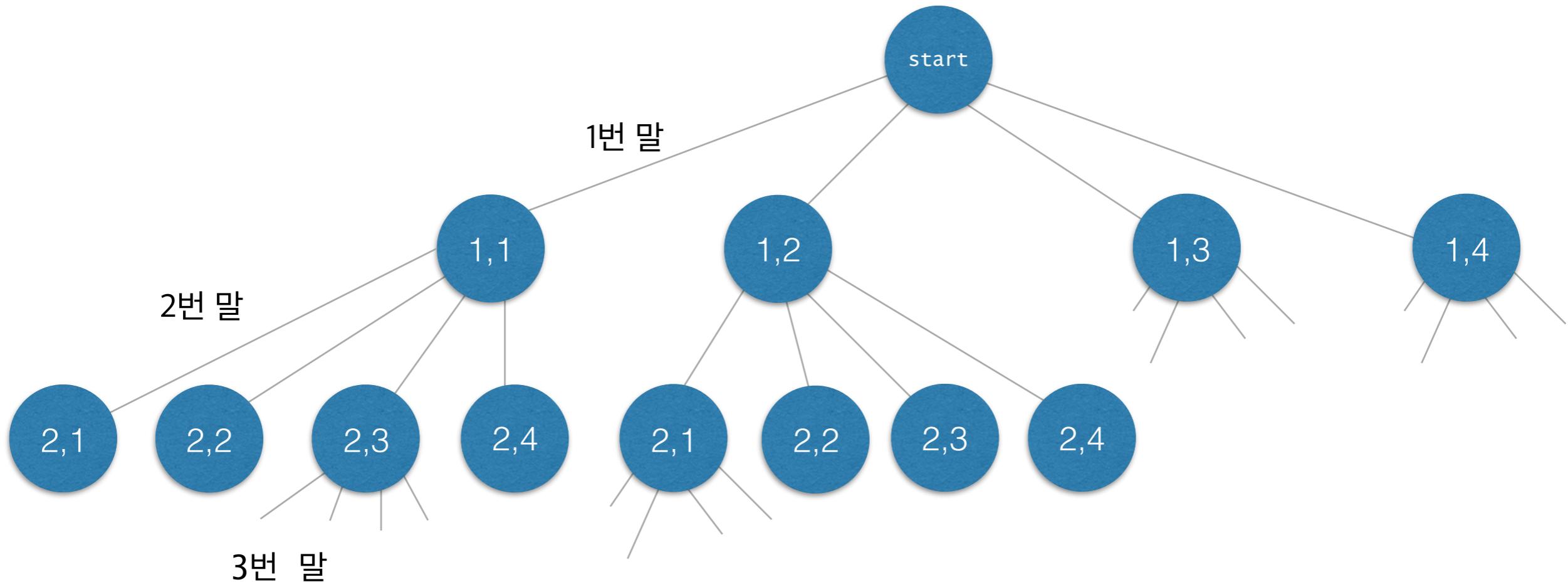
# **N-Queens**

# N-Queens Problem



The eight queens problem

# 상태공간트리



상태공간트리란 해를 포함하는 트리.

즉 해가 존재한다면 그것은 반드시 이 트리의 어떤 한 노드에 해당함  
따라서 이 트리를 체계적으로 탐색하면 해를 구할 수 있음

# 상태공간트리

상태공간 트리의 모든 노드를

탐색해야 하는 것은 아님

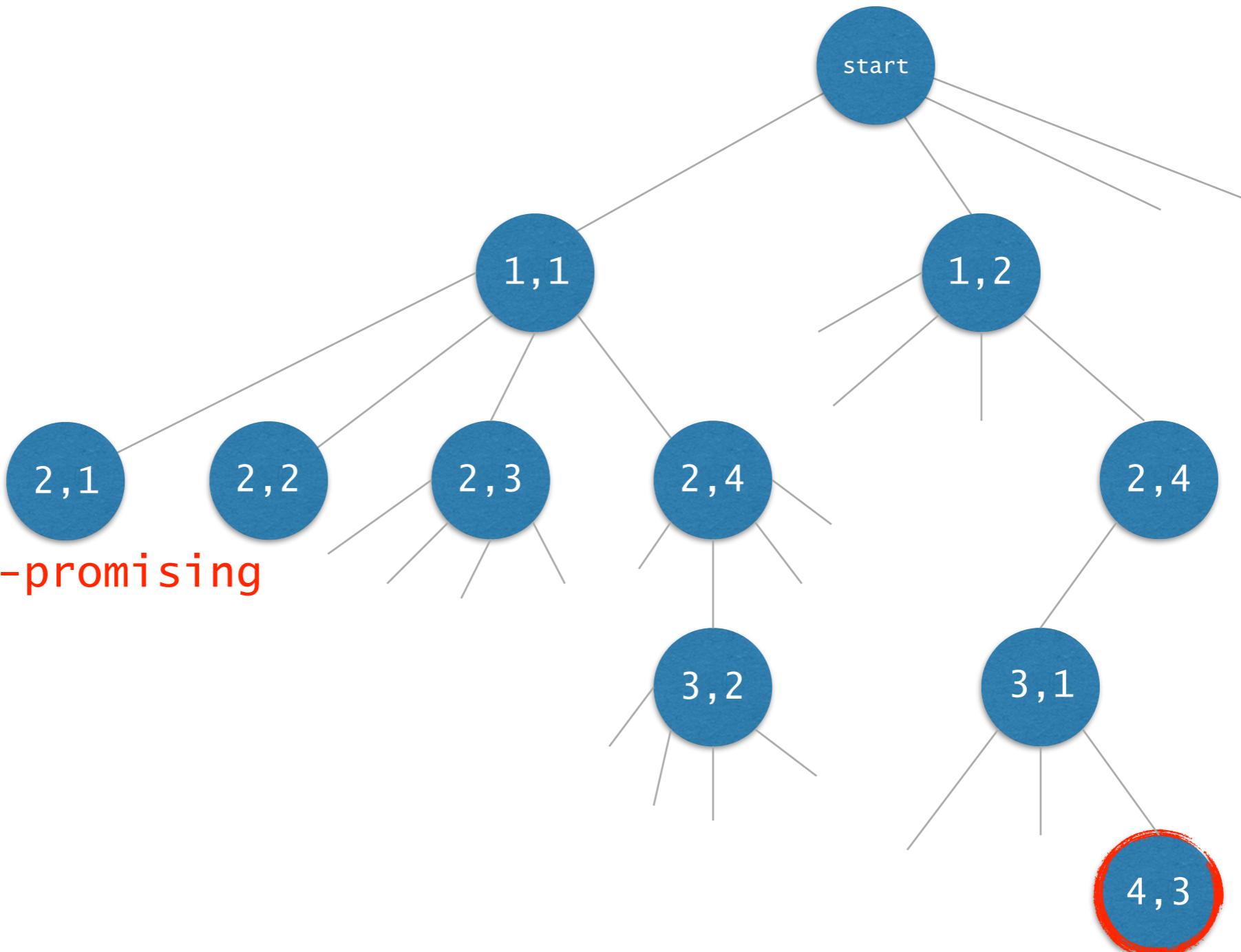
1번 말

2번 말

non-promising

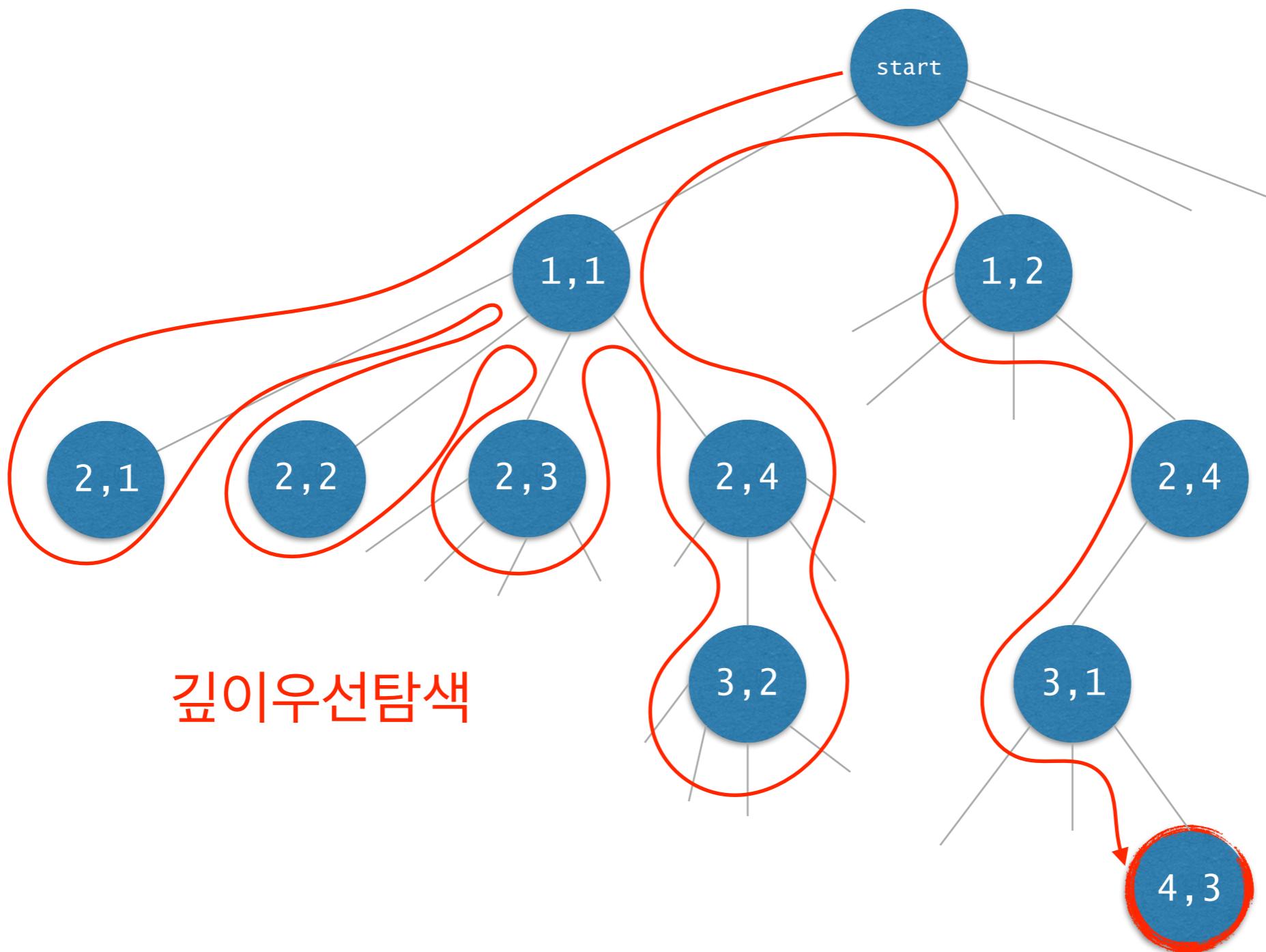
3번 말

4번 말



# 되추적 기법 (Backtracking)

- 상태공간 트리를 깊이 우선 방식으로 탐색하여 해를 찾는 알고리즘을 말한다.



# Design Recursion

매개변수는 내가 현재 트리의 어떤 노드에 있는지  
를 지정해야 한다.

```
return-type queens( arguments )
```

```
{
```

```
    if non-promising
```

```
        report failure and return;
```

```
    else if success
```

```
        report answer and return;
```

```
    else
```

```
        visit children recursively;
```

```
}
```

# Design Recursion

```
int cols[N+1];
return-type queens( int level )
{
    if non-promising
        report failure and return;
    else if success
        report answer and return;
    else
        visit children recursively;
}
```

매개변수 `level`은 현재 노드의 레벨을 표현하고, 1번에서 `level`번째 말이 어디에 놓였는지는 전역변수인 배열 `cols`로 표현하자.  
`cols[i]=j`는 `i`번 말이 (`i`행, `j`열)에 놓였음을 의미한다.

# Design Recursion

```
int cols[N+1];
bool queens( int level )
{
    if non-promising
        report failure and return;
    else if success
        report answer and return;
    else
        visit children recursively;
}
```

return-type은 일단 bool로 하자. 즉,  
성공이냐 실패냐를 반환한다.

# Design Recursion

```
int cols[N+1];
bool queens( int level )
{
    if (!promising(level))
        return false;
    else if success
        report answer and return;
    else
        visit children recursively;
}
```

노드가 어떤 경우에  
non-promising할까? 일단 이 문제는  
나중에 생각하자.

# Design Recursion

```
int cols[N+1];
bool queens( int level )
{
    if (!promising(level))
        return false;
    else if (level==N)
        return true;
    else
        visit children recursively;
}
```

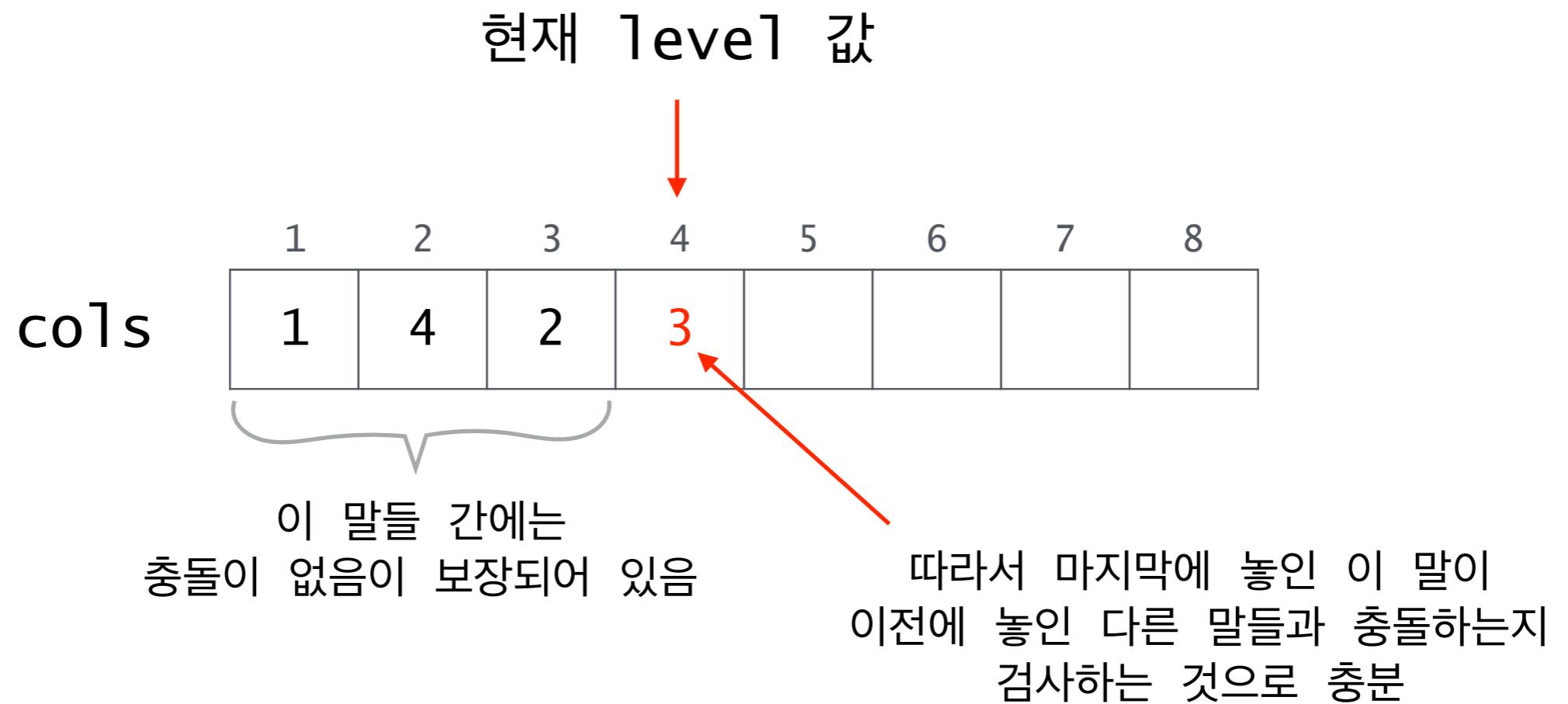
promising 테스트를 통과했다는 가정하에 level==N이면 모든 말이 놓였다는 의미이고 따라서 성공이다.

# Design Recursion

```
int cols[N+1];
bool queens( int level )
{
    if (!promising(level))
        return false;
    else if (level==N)
        return true;
    for (int i=1; i<=N; i++) {
        cols[level+1] = i;
        if (queens(level+1))
            return true;
    }
    return false;
}
```

level+1번째 말을 각각의 열에 놓은 후  
recursion을 호출한다.

# Promising Test



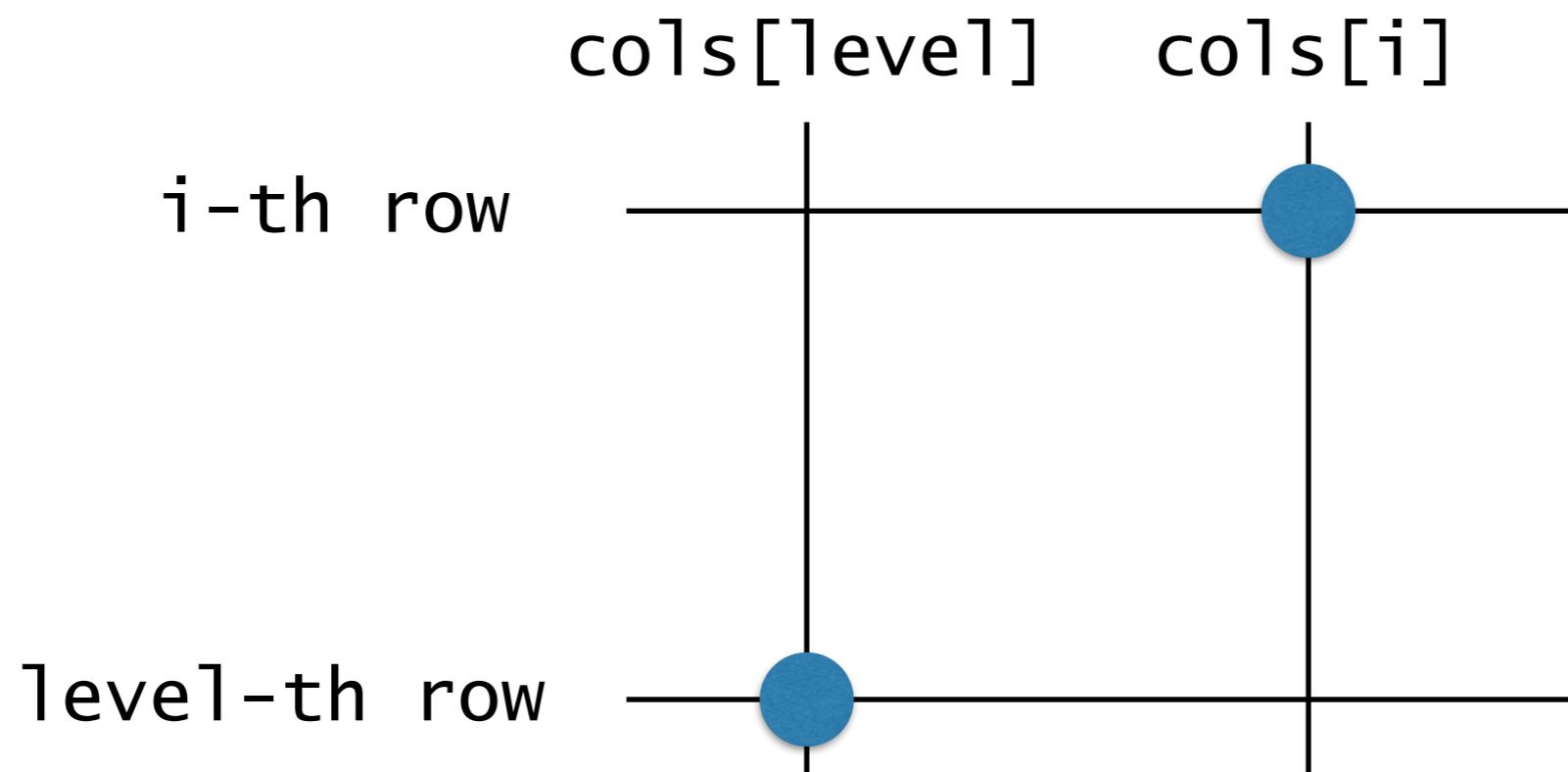
# Promising Test

```
bool promising( int level )
{
    for (int i=1; i<level; i++) {
        if (cols[i]==cols[level])
            return false;
        else if on the same diagonal
            return false;
    }
    return true;
}
```

같은 열에 놓였는지 검사

같은 대각선에 놓였는지 검사

# Promising Test



$$level-i = |cols[level]-cols[i]|$$

# Promising Test

```
bool promising(int level)
{
    for (int i=1; i<level; i++) {
        if (cols[i]==cols[level])
            return false;
        else if (level-i==Math.abs(cols[level]-cols[i]))
            return false;
    }
    return true;
}
```

같은 대각선에 놓였는지 검사

# N-Queen Problem

```
int cols[N+1];
bool queens( int level )
{
    if (!promising(level))
        return false;
    else if (level==N) {
        for (int i=1; i<=N; i++)
            printf("(%d, %d)", i, cols[i]);
        return true;
    }
    for (int i=1; i<=N; i++) {
        cols[level+1] = i;
        if (queens(level+1))
            return true;
    }
    return false;
}
```

처음에는 queens(0)로 호출한다.

멱집합

**powers**

- 임의의 집합 data의 모든 부분집합을 출력하라.

data = {a, b, c, d}

$\emptyset$   
a  
b  
c  
d  
a b  
a c  
a d  
b c  
...

$2^4=16$ 개

- ⦿ **{a,b,c,d,e,f}의 모든 부분집합을 나열하려면**
  - ⦿ a를 제외한 {b,c,d,e,f}의 모든 부분집합들을 나열하고
  - ⦿ {b,c,d,e,f}의 모든 부분집합에 {a}를 추가한 집합들을 나열한다.

- ☞  $\{b,c,d,e,f\}$ 의 모든 부분집합에  $\{a\}$ 를 추가한 집합들을 나열하려면
  - ☞  $\{c,d,e,f\}$ 의 모든 부분집합들에  $\{a\}$ 를 추가한 집합들을 나열하고
  - ☞  $\{c,d,e,f\}$ 의 모든 부분집합에  $\{a,b\}$ 를 추가한 집합들을 나열한다.

- ☞ **{c,d,e,f}의 모든 부분집합에 {a}를 추가한 집합들을 나열하려면**
  - ☞ {d,e,f}의 모든 부분집합들에 {a}를 추가한 집합들을 나열하고
  - ☞ {d,e,f}의 모든 부분집합에 {a,c}를 추가한 집합들을 나열한다.

Mission: S의 맥집합을 출력하라

powerSet(S)

if S is an empty set

    print nothing;

else

    let t be the first element of S;

    find all subsets of  $S - \{t\}$  by calling powerSet( $S - \{t\}$ );

    print the subsets;

    print the subsets with adding t;

이렇게 하려면  
powerSet 함수는 여러 개의  
집합들을 return해야 한다.  
어떻게?

Mission: S의멱집합을 구한 후 각각에 집합 P를 합집합하여 출력하라

powerSet(P, S)

if S is an empty set

    print P;

else

    let t be the first element of S;

    powerSet(P, S-{t});

    powerSet(P $\cup$ {t}, S-{t});

recursion 함수가 두 개의 집합을 매개변수로 받도록 설계해야 한다는 의미이다. 두 번째 집합의 모든 부분집합들에 첫번째 집합을 합집합하여 출력한다.

- $\{b,c,d,e,f\}$ 의 모든 부분집합에  $\{a\}$ 를 추가한 집합들을 나열하려면

- $\{c,d,e,f\}$ 의 모든 부분집합들에  $\{a\}$ 를 추가한 집합들을 나열하고
- $\{c,d,e,f\}$ 의 모든 부분집합에  $\{a,b\}$ 를 추가한 집합들을 나열한다.



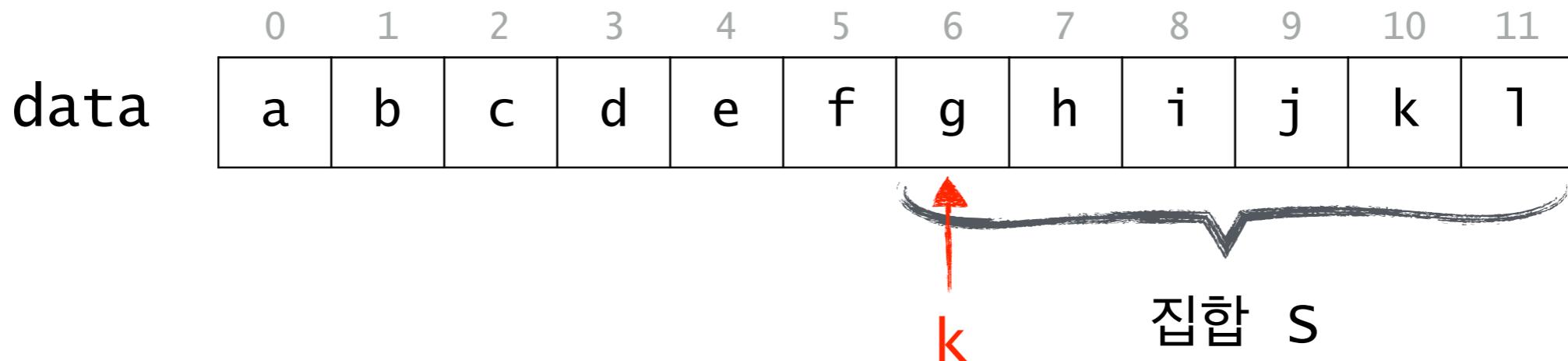
집합 S:  
k번째부터  
마지막 원소까지  
연속된 원소들이다.



집합 P:  
처음부터 k-1번째 원소들 중  
일부이다.

# 두 집합의 표현

Mission: S의 멱집합을 구한 후 각각에 집합 P를 합집합하여 출력하라



include

TRUE	FALSE	TRUE	TRUE	FALSE	FALSE						
------	-------	------	------	-------	-------	--	--	--	--	--	--

집합  $P=\{a, c, d\}$

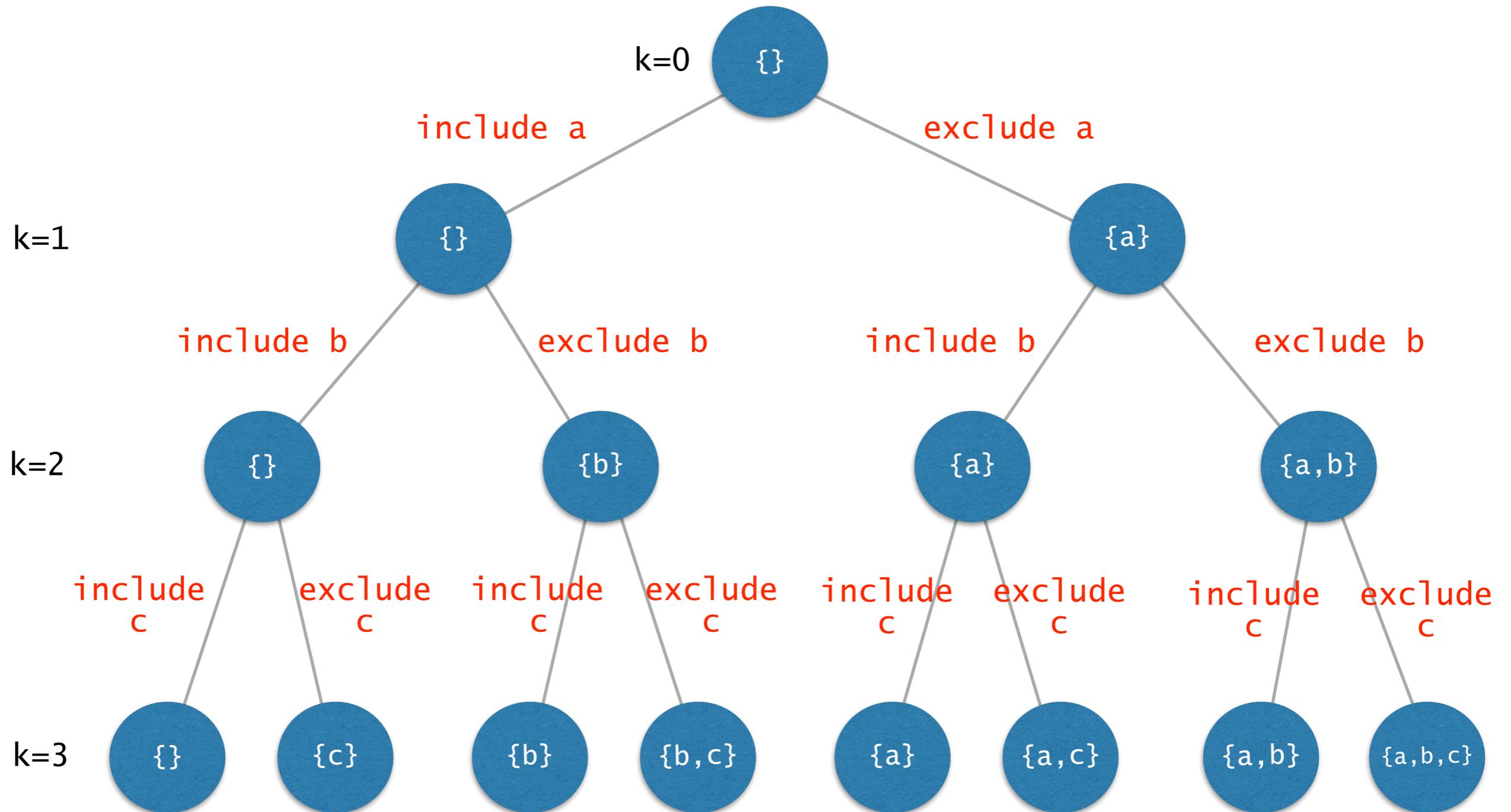
집합 S는  $\text{data}[k], \dots, \text{data}[n-1]$ 이고,  
집합 P는  $\underline{\text{include}[i]=\text{true}}, i=0, \dots, k-1$ ,인 원소들이다.

```
private static char data[] = {'a','b','c','d','e','f'};  
private static int n=data.length;  
private static boolean [] include = new boolean [n];  
  
public static void powerSet(int k) {  
    if (k==n) {  
        for (int i=0; i<n; i++)  
            if (include[i]) System.out.print(data[i] + " ");  
        System.out.println();  
        return;  
    }  
    include[k]=false;  
    powerSet(k+1);  
    include[k]= true;  
    powerSet(k+1);  
}
```

**Mission:**  
data[k], ..., data[n-1]의 멱집합을 구한 후 각각에 include[i]=true, i=0, ..., k-1, 인 원소를 추가하여 출력하라.

처음 이 함수를 호출할 때는 powerSet(0)로 호출한다. 즉 P는 공집합이고 S는 전체집합이다.

# 상태공간트리 (state space tree)



## 상태공간트리 (state space tree)

- ⦿ 해를 찾기 위해 탐색할 필요가 있는 모든 후보들을 포함하는 트리
- ⦿ 트리의 모든 노드들을 방문하면 해를 찾을 수 있다.
- ⦿ 루트에서 출발하여 체계적으로 모든 노드를 방문하는 절차를 기술한다.

```
private static char data[] = {'a','b','c','d','e'};  
private static int n=data.length;  
private static boolean [] include = new boolean [n];
```

```
public static void powerSet(int k) {  
    if (k==n) {  
        for (int i=0; i<n, i++)  
            if (include[i]) System.out.print(data[i] + " ");  
        System.out.println();  
        return;  
    }  
    include[k]=false;  
    powerSet(k+1);  
    include[k]= true;  
    powerSet(k+1);  
}
```

트리상에서 현재 나의 위치를 표현한다.

만약 내 위치가 리프노드라면

먼저 왼쪽으로 내려갔다가

이번엔 오른쪽으로 내려간다.

# **Permutation**

순열

# 순열(permutation)

- 임의의 집합 data에 대해서 원소들의 모든 가능한 순열을 출력하라.

```
data = {a, b, c, d}
```

a	b	c	d
a	b	d	c
a	c	b	d
a	c	d	b
a	d	b	c
a	d	c	b
b	a	c	d
b	a	d	c
b	c	a	d
...			

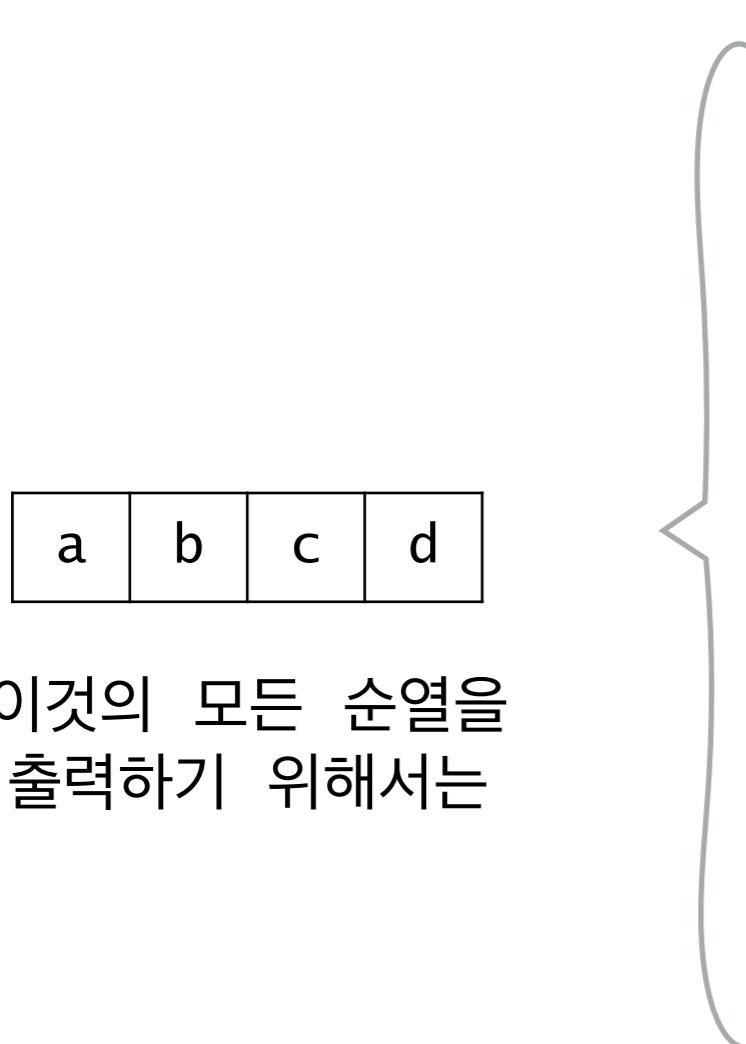
$4! = 24$ 개

# 순열(permutation)

## ④ $\{a,b,c,d\}$ 의 모든 순열

- ⓐ 첫 원소가 a이면서  $\{b,c,d\}$ 의 모든 순열
- ⓑ 첫 원소가 b이면서  $\{a,c,d\}$ 의 모든 순열
- ⓒ 첫 원소가 c이면서  $\{a,b,d\}$ 의 모든 순열
- ⓓ 첫 원소가 d이면서  $\{a,b,c\}$ 의 모든 순열

# 순열(permuation)



이것의 모든 순열을 출력하기 위해서는

a	b	c	d
---	---	---	---

b	a	c	d
---	---	---	---

c	b	a	d
---	---	---	---

d	b	c	a
---	---	---	---

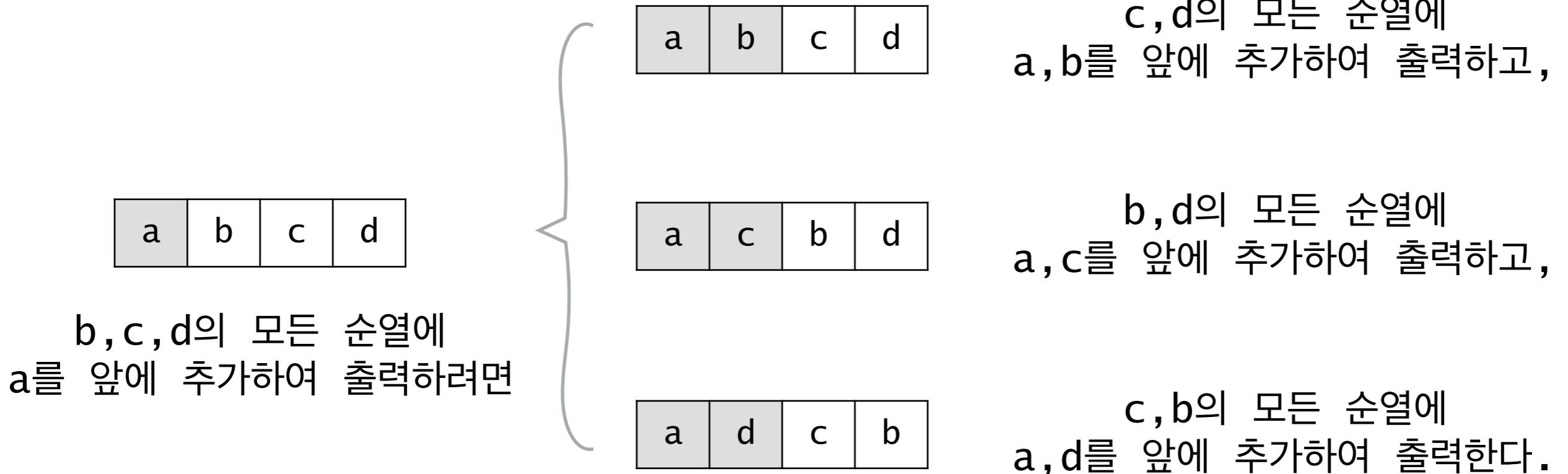
b, c, d의 모든 순열에  
a를 앞에 추가하여 출력하고,

a, c, d의 모든 순열에  
b를 앞에 추가하여 출력하고,

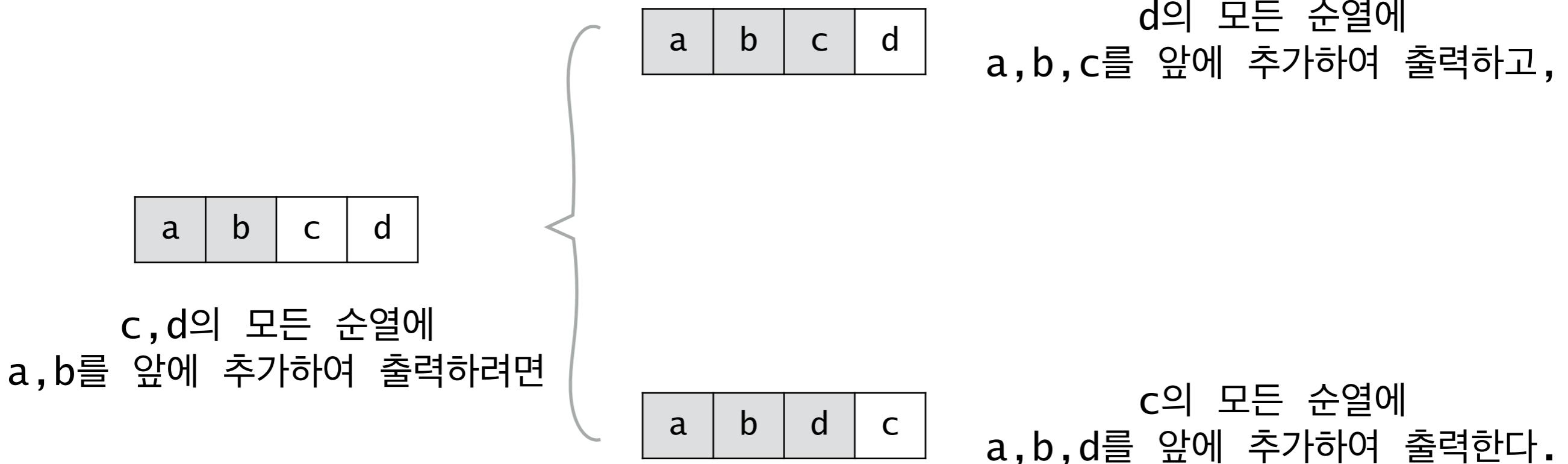
b, a, d의 모든 순열에  
c를 앞에 추가하여 출력하고,

b, c, a의 모든 순열에  
d를 앞에 추가하여 출력한다.

# 순열(permuation)



# 순열(permutation)



# 순열(permutation)

a	b	d	c
---	---	---	---



a	b	d	c
---	---	---	---

c의 모든 순열에  
a , b , d를 앞에 추가하여  
출력하려면

$\emptyset$ 의 모든 순열에  
a , b , d , c를 앞에 추가하여  
출력한다.

# 순열(permutation)

집합  $S=\{a,b,c,d\}$ 의 모든 순열을 출력하려면

$S$ 의 각 원소  $x$ 에 대해서

$S-\{x\}$ 의 모든 순열들을 생성한 다음 각각의 맨 앞에  $x$ 를 추가해서 출력한다.

# 순열(permutation)

$S - \{a\}$ 의 모든 순열들을 생성한 다음 각각의 맨 앞에  $a$ 를 추가해서 출력하려면

$S - \{a\}$ 의 각 원소  $y$ 에 대해서

$S - \{a, y\}$ 의 모든 순열들을 생성한 다음 각각의 맨 앞에  $(a, y)$ 를 추가해서 출력한다.

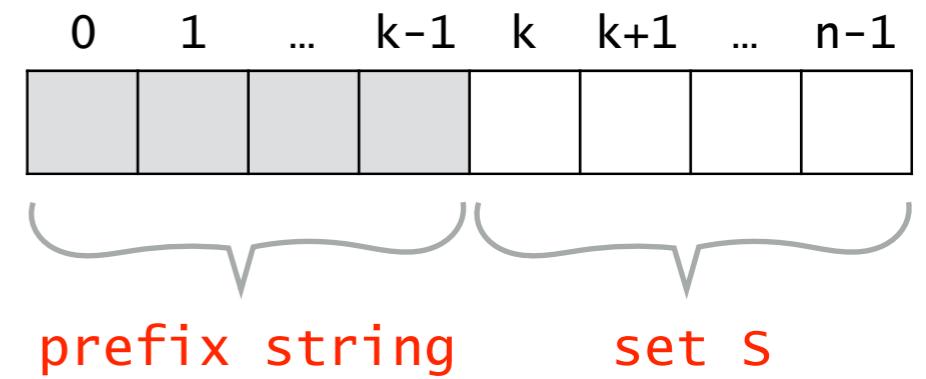
## The interface should be

Mission: S의 모든 순열들을 생성한 후 각각에 prefix string을 앞에 붙여서 출력한다.

```
void printPerm(a prefix string, a set S)
{
    if |S| is 0
        print the prefix string;
    else
        for each element x in S
            printPerm(the prefix string + x, S-{x});
}
```

# 순열(permutation)

```
char data[] = {'a','b','c','d'};  
int n=4;  
  
void perm(int k) {  
    if (k==n) {  
        print data[0…n-1];  
        return;  
    }  
    for (int i=k; i<n; i++) {  
        swap(data, k, i);           // swap data[k] and data[i]  
        perm(data, k+1, n);  
    }  
}
```



Is it okay ?

# 순열(permuation)

```
char data[] = {'a','b','c','d'};
```

```
int n=4;
```

```
void perm(int k) {
```

```
    if (k==n) {
```

```
        print data[0…n-1];
```

```
        return;
```

```
}
```

```
    for (int i=k; i<n; i++) {
```

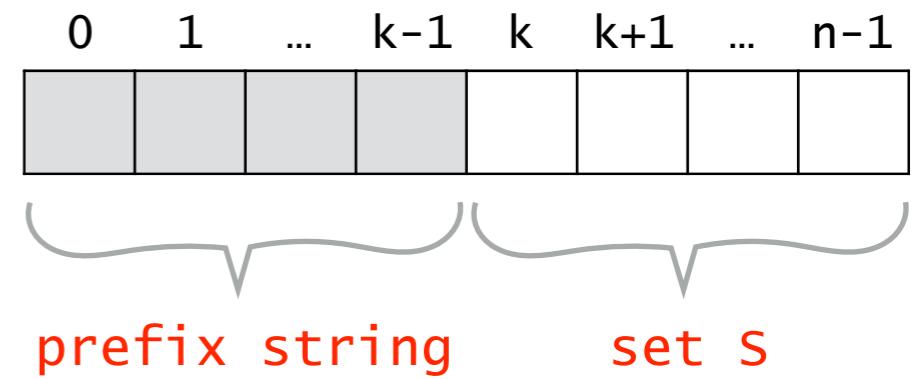
```
        swap(data, k, i);
```

```
        perm(data, k+1, n);
```

```
}
```

이 호출 이후에 데이터의 순서를 유지된다는 보장이 없음

```
}
```



# Designing recursion

- ⦿ recursion이 데이터를 변경할 때는 매우 조심해야한다.
- ⦿ 호출 전후에 데이터가 변경되지 않고 유지되도록 하는게 좋다.

# 순열(permutation)

```
void perm(int k)  {
    if (k==n)  {
        print data[0…n-1];
        return;
    }
    for (int i=k; i<n; i++)  {
        swap(data, k, i);
        perm(data, k+1, n);
        swap(data, k, i);
    }
}
```

**Tip:**  
리커전에 들어가기 전과 후에 데이터의  
동일성이 유지되도록 하라.

Mission:  
data[0..k-1]을 prefix로 하고,  
data[k..n]으로 만들 수 있는 모든 순열을  
프린트하되, 배열 data에 저장된 값들의 순서는  
그대로 유지한다.