



# 알고리즘 문제해결기법

제4주 Enumeration

## Enumeration

An **enumeration** is a complete, ordered listing of all the items in a collection. The term is commonly used in [mathematics](#) and [computer science](#) to refer to a listing of all of the [elements](#) of a [set](#).

Some sets can be enumerated by means of a **natural ordering** (such as 1, 2, 3, 4, ... for the set of [positive integers](#)), but in other cases it may be necessary to impose a (perhaps arbitrary) ordering.

# Enumeration

- 리스트
- 고정 차원의(fixed dimensional) 직교좌표공간
- 트리 혹은 그래프
- 부분집합, 순열(permutation)

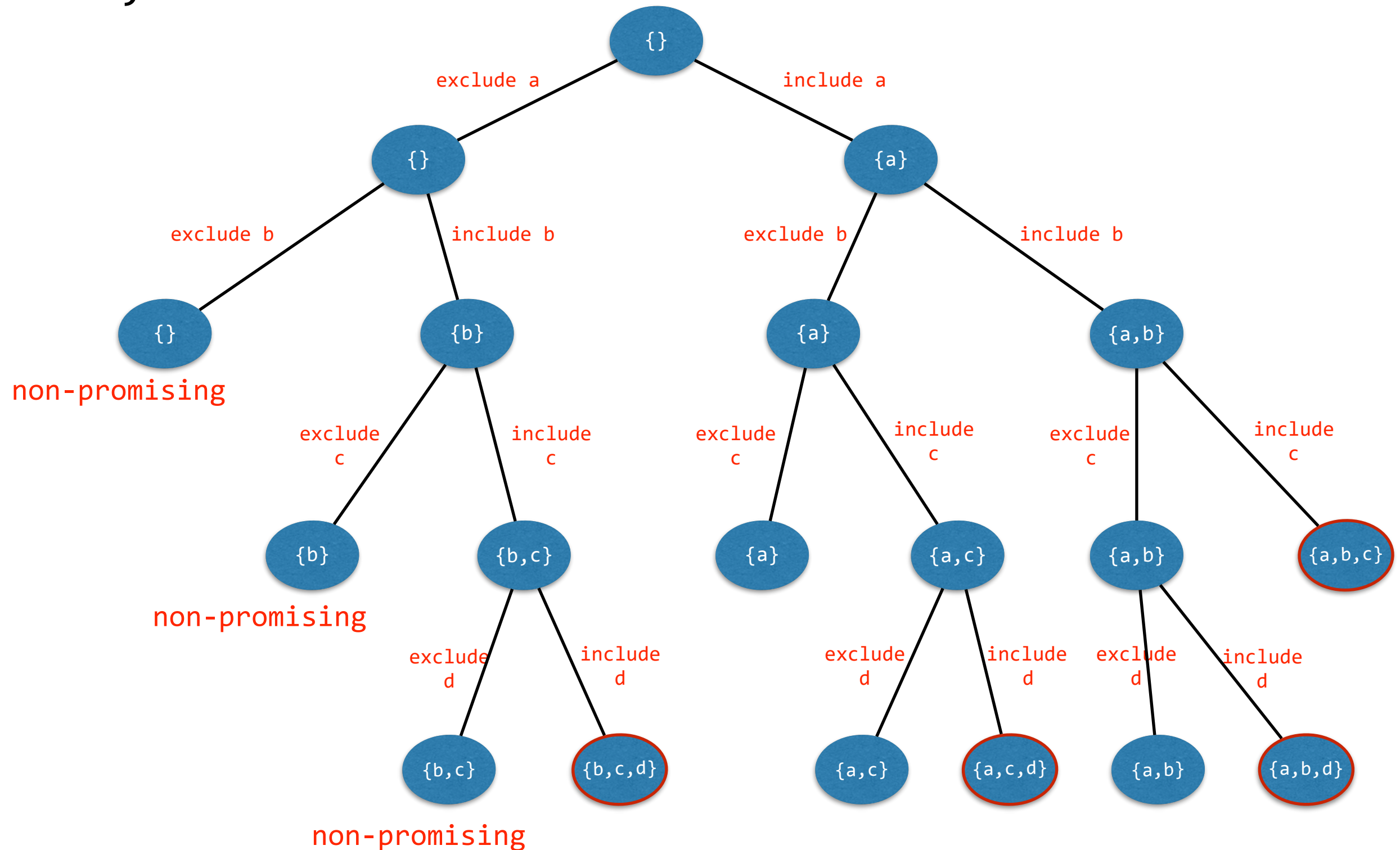
- 주어진 문제의 해와 **관련**이 있는 **모든 객체** 혹은 **상태**를 포함하는 트리
- 트리의 노드를 enumerate (혹은 traverse)함으로써 해를 찾을 수 있음
  - DFS, BFS, ...
  - 되추적 기법 (Backtracking)
  - 분기한정법 (branch and bound, pruning)
- 해공간(solution space) 트리, 탐색공간(search space) 트리 등으로 부르기도 함

## M out of N

- 크기가  $N$ 인 집합이 있다. 크기가  $M(\leq N)$ 인 모든 부분집합을 나열하는 프로그램을 작성하라.

$\{a, b, c, d\}$   
 $N=4, M=3$

## 상태공간트리 (state space tree)

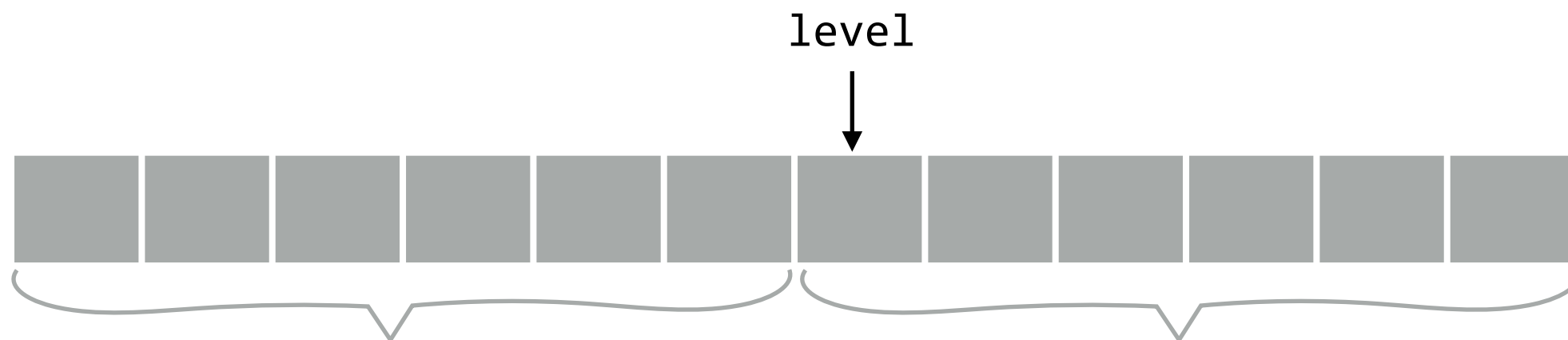


# M out of N

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

```
void M_OutOf_N(int level, int m) {
```

- level: 트리에서 나의 레벨
- include: 거기까지 내려오는 동안 어떤 원소를 선택하고 어떤 원소를 선택하지 않았는지 표시
- m: 추가로 선택해야할 원소의 개수



```
}
```

## M out of N

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

void M_OutOf_N(int level, int m)  {
    if (m==0) {
        for (int i=0; i<N; i++)
            if (include[i]) System.out.print(data[i] + " ");
        System.out.println();
    }
    else if (m>N-level) /* non-promising */
        return;
    else {
        include[level] = true;
        M_OutOf_N(level+1, m-1);
        include[level] = false;
        M_OutOf_N(level+1, m);
    }
}
```

data[level],...,data[N-1] 사이에서 m개를 선택하여 data[0],...,data[level-1] 사이의 데이터들 중에 include[i]가 true인 데이터들과 함께 출력한다.  
처음에는 M\_OutOf\_N(0, M)으로 호출한다.



연산	연산자	예: a = 60 = 00111100 = 0x3C b = 13 = 00001101 = 0x0D
두 정수 a, b를 비트별로 AND연산	c = a & b	c = 00001100 = 12 = 0x0C
두 정수 a, b를 비트별로 OR연산	c = a   b	c = 00111101 = 61 = 0x3D
두 정수 a, b를 비트별로 XOR연산	c = a ^ b	c = 00110001 = 49 = 0x31
정수 a의 비트별 NOT연산	c = ~a	c = 111...11000011 = -61 = 0xFFFFF3
정수 a를 왼쪽으로 b비트 쉬프트	c = a << b	c = 11110000 = 240 = 0xF0
정수 a를 오른쪽으로 b비트 쉬프트	c = a >> b	c = 00001111 = 15 = 0x0F

## Bit-mask를 이용한 집합의 표현

- 하나의 N비트 정수를 이용하여 크기가 N인 집합의 임의의 부분집합을 표현한다.  
*i번째 비트가 i번째 원소의 포함여부를 나타낸다.*
- $N < 32$ 인 경우 하나의 32비트 정수(int, unsigned int)로,  $N < 64$ 인 경우 하나의 64비트 정수(long in Java, long long in C)를 사용하여 각 비트마다 하나의 원소의 포함 여부를 표현
- *부호비트*에 주의 (overflow, carry), 32비트 정수와 64비트 정수를 섞어 쓰는 것에 주의
- C 혹은 C++의 경우 unsigned 타입 사용 가능

## Bit-mask를 이용한 집합의 표현

- 고객들이 원하는 토핑을 선택할 수 있는 피자집이 있다. 0~19번까지의 번호를 가지는 20가지의 선택가능한 토핑이 있다.

- 아무런 토핑도 포함하지 않는 공집합은

$$00000000000000000000_2 = 0$$

으로 표현된다.

- 20가지 토핑을 모두 포함하는 전체집합은

$$11111111111111111111_2 = 1048575_{10}$$

로 표현한다. 이 이진수는 다음과 같이 쉽게 구할 수 있다.

```
int fullPizza = (1 << 20) - 1;
```

# Bit-mask를 이용한 집합의 표현

```
int fullPizza = (1 << 20) - 1;
```

1

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$1 \ll 20$

0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$(1 \ll 20) - 1$

0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(혹은 `int fullPizza = 0xFFFF;`)

# Bit-mask를 이용한 집합의 표현

- 원소 추가: p번 토핑의 추가

```
toppings |= (1 << p);
```

p=10인 경우의 예

toppings

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(1<<10)

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

```
toppings |= (1<<10);
```

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

## Bit-mask를 이용한 집합의 표현

- 원소의 포함 여부: p번 토핑이 포함되어 있는지 검사

```
if ((toppings & (1<<p))>0)
    print(p + "th topping is in");
```

p=8인 경우의 예

toppings

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(1<<8)

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

toppings & (1<<8);

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# Bit-mask를 이용한 집합의 표현

- 원소의 삭제: p번 토핑의 삭제

`topping &= ~(1 << p);`

toppings

p=11인 경우의 예

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$(1 \ll 11)$

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$\sim(1 \ll 11)$

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

`toppings &= ~(1 << 11);`

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# Bit-mask를 이용한 집합의 표현

- 원소의 토글(toggle)

`topping ^= (1 << p);`

toppings

p=11인 경우의 예

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(1<<11)

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

`toppings ^= (1 << 11);`

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



## Bit-mask를 이용한 집합의 표현

- 두 집합에 대한 연산하기

```
int added = (a | b);           // 합집합
int intersected = (a & b);      // 교집합
int removed = (a & ~b);        // 차집합 (a-b)
int toggled = (a ^ b);         // a와 b중 하나에만 속한 원소들
```

## Bit-mask를 이용한 집합의 표현

- 집합의 크기 구하기

```
int bitCount(int x) {  
    if (x==0) return 0;  
    return x%2 + bitCount(x/2);  
}
```

언어	집합의 크기 구하기
gcc/g++	__builtin_popcount(toppings)
Visual C++	__popcnt(toppings)
Java	Integer.bitCount(toppings)



## Bit-mask를 이용한 집합의 표현

- 첫 번째 원소 찾기: 정수의 이진 표현에서 오른쪽에 붙어 있는 0의 개수

```
int indexOfFirstElement(int x) {  
    if (x==0) return -1;           // 공집합  
    else if (x%2==1) return 0;  
    return 1+indexOfFirstElement(x/2);  
}
```

언어	첫 번째 원소 찾기
gcc/g++	<code>__builtin_ctz(toppings)</code>
Visual C++	<code>_BitScanForward(&amp;index, toppings)</code>
Java	<code>Integer.numberOfTrailingZeros(toppings)</code>

## Bit-mask를 이용한 집합의 표현

- 주어진 집합의 모든 부분집합 순회하기

```
int pizza = 121;    // 임의의 집합
for (int subset=pizza; subset!=0; subset=((subset-1)&pizza))
{
    // subset은 pizza의 부분집합
}
```

만약 pizza가 full set이라면  
subset--로 충분하다.

- 주의:** 공집합은 방문하지 않음

## Problem 10

```
void solve(int N, int K) {  
    int set = (1 << N) - 1;           // 전체 자리수를 나타내는 집합  
    for ( ; set >= 0; set--) {  
        if (bitCount(set) == K)  
            printNumbersCorrespondingToBits(set);  
    }  
}
```

## 비교: recursion vs. bit map

- Bit map을 이용하는 방법이 연산의 속도 측면에서 빠름
- Recursion의 경우 함수 호출에 따른 오버헤드 때문에 느림
- 따라서 멍집합을 생성할 때는 bit map 방법이 빠름
- M-out-of-N 문제의 경우
  - bit map의 경우 항상  $2^N$ 개의 부분집합을 생성
  - Recursion의 경우 크기가 M 이하인 부분집합만을 생성
  - 따라서 M-out-of-N 문제에서는 M이 작거나 N에 가깝다면 recursion이 훨씬 빠름

## 문제 12: 스도쿠 [백준 2580번]

가로, 세로 각각 9개씩 총 81개의 작은 칸으로 이루어진 정사각형 판 위에서 이뤄지는데, 게임 시작 전 몇몇 칸에는 1부터 9까지의 숫자 중 하나가 쓰여 있다.

나머지 빈 칸을 채우는 방식은 다음과 같다.

- 1 각각의 가로줄과 세로줄에는 1부터 9까지의 숫자가 한 번씩만 나타나야 한다.
- 2 굵은 선으로 구분되어 있는 3x3 정사각형 안에도 1부터 9까지의 숫자가 한 번씩만 나타나야 한다.

	3	5	4	6	9	2	7	8
7	8	2	1		5	6		9
	6		2	7	8	1	3	5
3	2	1		4	6	8	9	7
8		4	9	1	3	5		6
5	9	6	8	2		4	1	3
9	1	7	6	5	2		8	
6		3	7		1	9	5	2
2	5	8	3	9	4	7	6	

## 문제 12: 스도쿠 [백준 2580번]

Empty slot의 개수

```
bool solve(struct pos p, int empty) {  
    if (!promising(p)) ← 위치 p에 놓인 값이 규칙을 위반하지 않는지 검사  
        return false;  
    if (empty == 0)  
        return true;  
  
    struct pos next = findEmptySlot(p); ← 다음의 빈 칸 찾기  
    for (int val = 1; val <= 9; val++) {  
        board[next.x][next.y] = val;  
        if (solve(next, empty - 1)) ← 거기에 1에서 9까지의 값을 차례로 넣고 recursion  
            return true;  
    }  
    board[next.x][next.y] = 0; ← 실패한 경우에는 그 자리를 비우고 false를 return  
    return false;  
}
```




# 시간을 줄이기 위한 아이디어

- 유일한 값으로 정해지는 곳을 미리 처리한다.
- DOF (Degree of Freedom)가 작은 셀부터 처리한다.

## 문제 12: 우회전 미로찾기

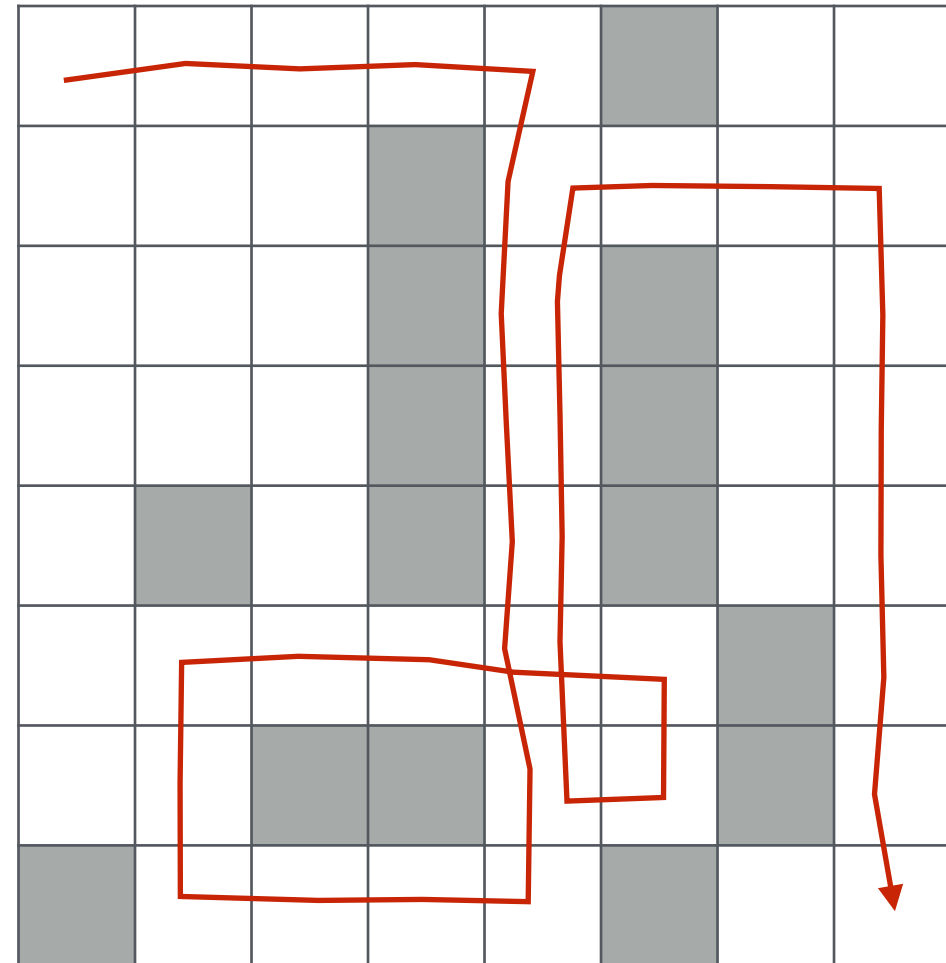
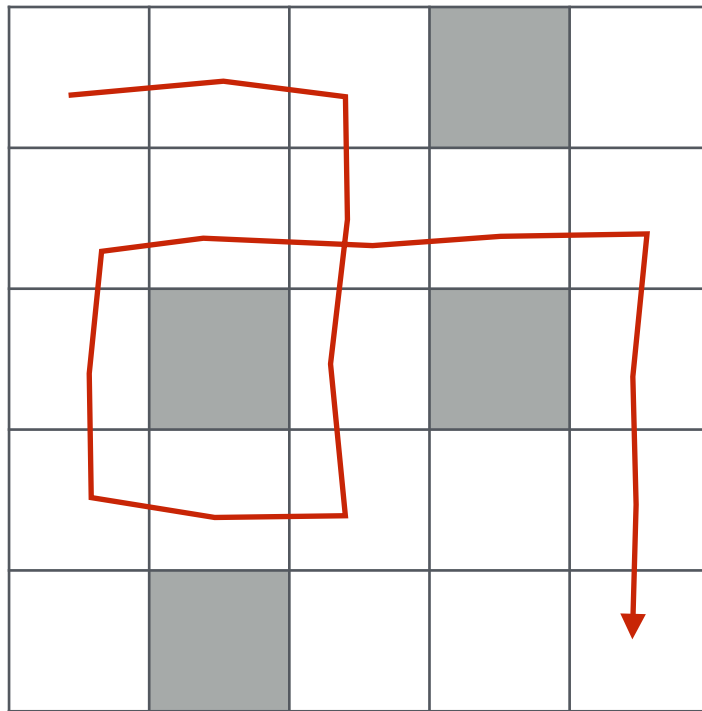
### 일반적인 미로찾기의 경우

```
boolean findPathInMaze(p)
  if p is not a permissible position (the wall or a visited cell)
    return false;
  if p is the exit
    return true;
  mark p as visited;
  for each neighboring cell q of p do
    if findPathInMaze(q)
      return true;
  return false;
```



왜 한 번 방문한 셀은 다시 방문하지 않아도 되는가?

## 문제 12: 우회전 미로찾기

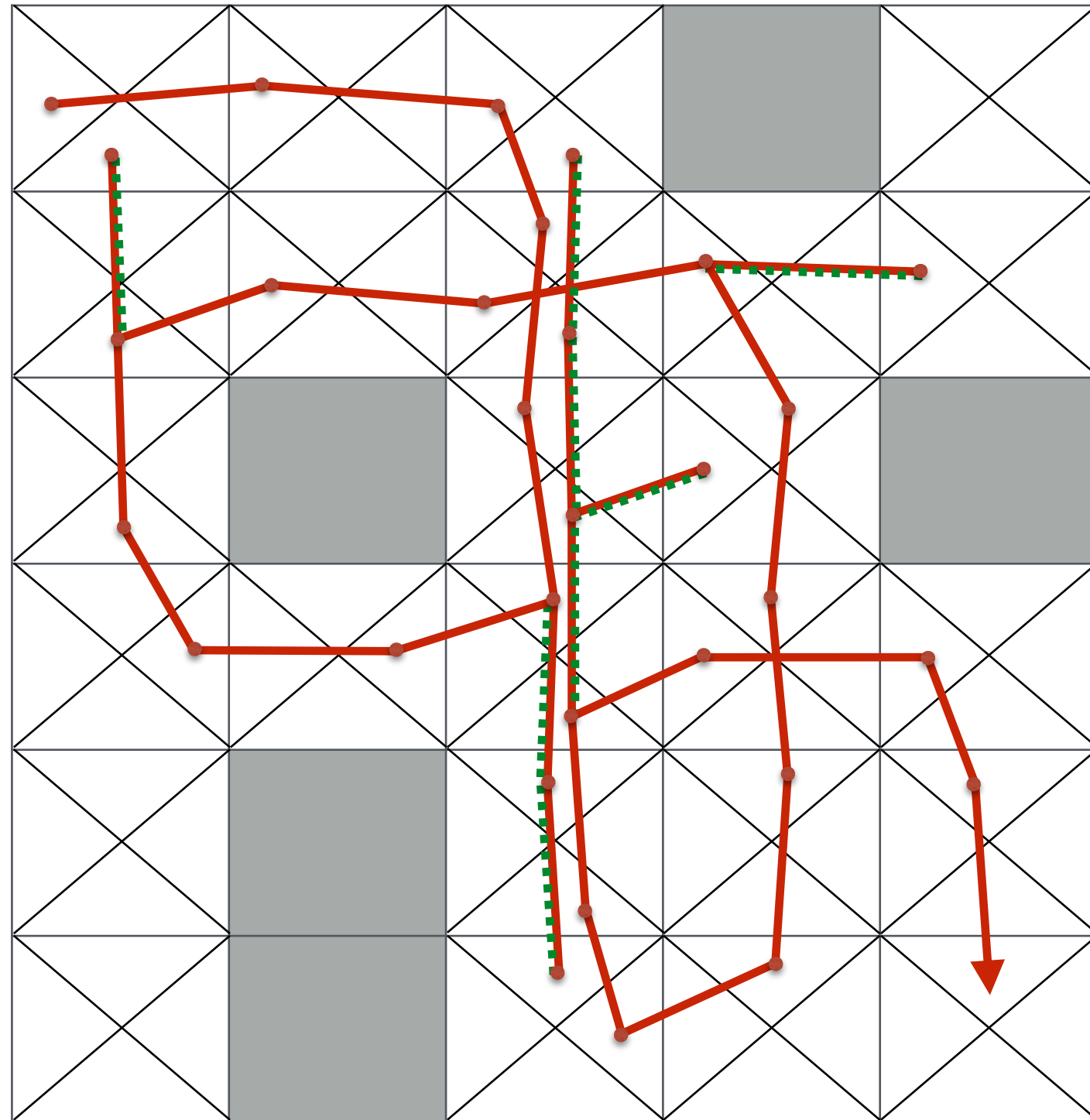


셀을 중복 방문할 수 있어야 한다.

## 문제 12: 우회전 미로찾기

- 한 지점을 여러 번 방문할 수 있다. 단, 매번 다른 방향에서...

## 문제 12: 우회전 미로찾기



.....  
backtrack된 경로

각 셀에 대해서  
4가지 진입방향에 대해 각각  
따로 방문 체크를 한다.

## 문제 12: 우회전 미로찾기

```
#define MAX 16

int board[MAX][MAX];
int n;
int visited[MAX][MAX][4]; ← 4가지 진입방향에 대해서 각각 따로 방문여부를 체크한다.

typedef struct {
    int x, y;           // coordinate,
    int d;              // entering direction,
                        // 0 from left, 1 from above, 2 from right, 3 from below
} Position;

void solve() {
    init();              // initialize the array 'visited'
    Position start = {.x = 0, .y = 0, .d = 0};
    if (maze(start)) printf("YES\n");
    else printf("NO\n");
}
```

## 문제 12: 우회전 미로찾기

```
bool maze(Position p) {
    if (p.x < 0 || p.x >= n || p.y < 0 || p.y >= n || board[p.x][p.y] != 0
        || visited[p.x][p.y][p.d] != 0)
        return false;
    if (p.x == n-1 && p.y == n-1)
        return true;
    visited[p.x][p.y][p.d] = 1;
    for (int turn = 0; turn < 2; turn++) { // 0 for straight, 1 for right turn
        Position q = next_pos(p, turn);
        if (maze(q))
            return true;
    }
    return false;
}

int offset[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
Position next_pos(Position p, int turn) { // 0 for straight, 1 for right turn
    Position q;
    q.x = p.x + offset[(p.d+turn)%4][0];
    q.y = p.y + offset[(p.d+turn)%4][1];
    q.d = (p.d+turn)%4;
    return q;
}
```