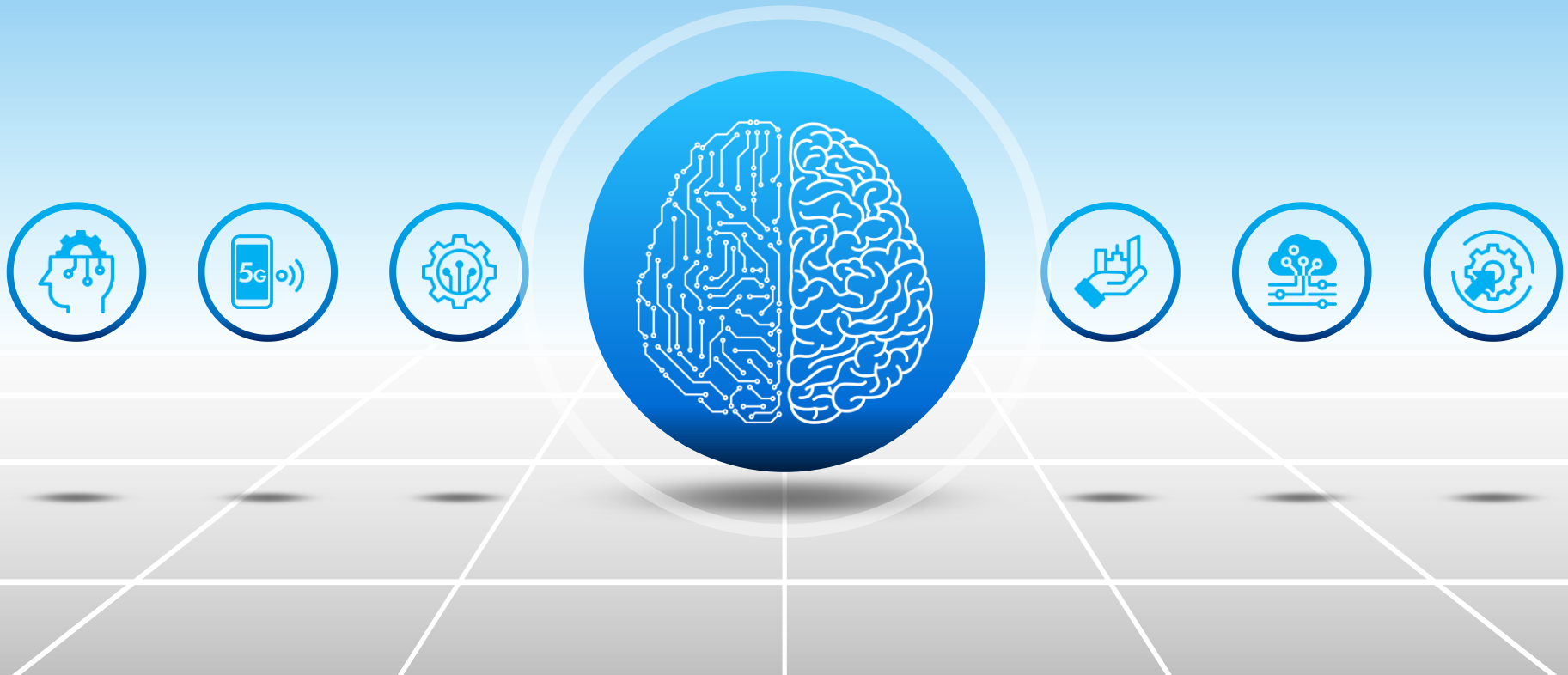


CIS3034 문제해결프로젝트

# 나머지와 소인수 문제해결



# 목 차

## CONTENTS



**I** 나눗셈과 나머지

**II** 소인수 분해

**II** 예제 풀이

# 나눔셈과 나머지



# 나눗셈과 나머지

- 프로그래밍 언어에서 정수의 나눗셈은 몫과 나머지 계산으로 수행된다.
- 실수의 경우 소수로 결과가 나오므로 나눗셈을 할 때 타입에 주의한다.

```
int div = A / B; //A를 B로 나눈 몫  
int mod = A % B; //A를 B로 나눈 나머지
```

- B가 0이면 Runtime Error가 발생하므로 주의한다.

# 나머지의 특징

- 나눗셈 연산의 결과는 몇 가지 중요한 특징을 갖는다.

```
int div = A / B; //A를 B로 나눈 몫  
int mod = A % B; //A를 B로 나눈 나머지
```

- $(div * B + mod == A)$  이 성립한다.
- A와 B가 음수가 아닐 때 나머지는 항상  $0 \sim (B-1)$  범위의 정수이다.
  - 수학적으로 음수일 때도 나머지가  $0 \sim (B-1)$  범위를 가져야 하지만, 실제로 프로그래밍 언어에서는 그렇지 않은 경우가 존재함.
- 나머지가 0이고 A와 B가 양수이면 A는 B의 배수이고, B는 A의 약수이다.

# 몫과 나머지의 규칙성

- 규칙적으로 변하는 수열을 같은 수로 나누면 대부분 규칙성을 갖는다. (이를 바탕으로 쉽게 문제해결이 가능)

1.  $a_i = i, m = 3$

a	1	2	3	4	5	6	7	8	9	10	11	12
나머지	1	2	0	1	2	0	1	2	0	1	2	0
몫	0	0	1	1	1	2	2	2	3	3	3	4

2.  $a_i = i^2, m = 7$

a	1	4	9	16	25	36	49	64	81	100	121	169	196
나머지	1	4	2	2	4	1	0	1	4	2	2	4	1

- 수열의 시작을 0부터 하면 규칙이 더 잘 보인다.
- 스도쿠보드 문제해결 실습

# 나머지의 닫힘 성질

- 문제의 조건상 나머지 값들을 계속 계산해야 할 때 유용함.

1. 더하기  $((a \bmod p) + (b \bmod p)) \bmod p = (a + b) \bmod p$

2. 곱하기  $((a \bmod p) \times (b \bmod p)) \bmod p = (a \times b) \bmod p$

3. 빼기  $((a \bmod p) - (b \bmod p) + p) \bmod p = (a - b + p) \bmod p$

# 나머지의 닫힘 성질

- 각 예제들을 계산해 보자.

a.  $(1 + 7 + 14 + 21 + \dots + 70) \bmod 5$

b.  $2^{64} \bmod 1000$

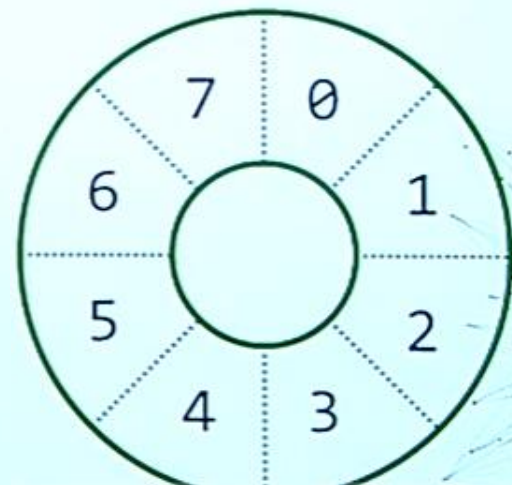
c. 다음 게임판에서  $x$ 번 칸에 있는 사람이 아래와 같이 이동하면 각각 어디에 있는가  
(단,  $m \leq 7, k \leq 7$ )

a. 시계 반대 방향으로  $m$  칸

b. 시계 방향으로  $k$  칸

c. 시계 반대 방향으로  $m$  칸 이동 후 시계 방향으로  $k$  칸

d. 시계 방향으로  $m$  칸 씩  $z$ 번 반복 이동



– Probing 문제해결 실습



# 소인수 분해



# 소수

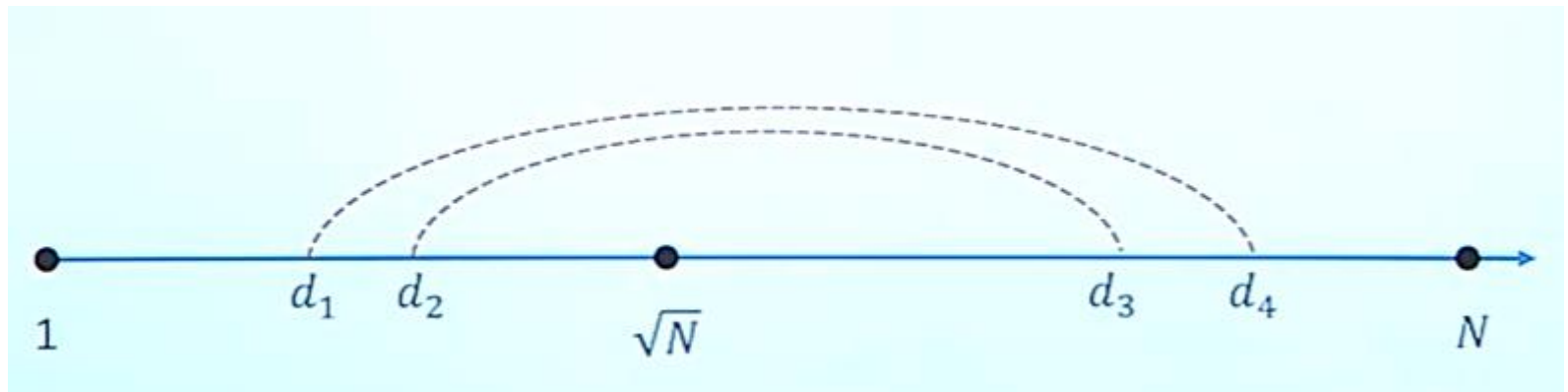
- 1과 자기 자신으로만 나누어 떨어지는 숫자
- 약수가 2개인 숫자
- 소인수 분해 결과가 자기 자신인 숫자

# 약수의 특징

- 아래와 같이 자연수  $N$ 을 두 자연수  $a, b$ 의 곱으로 나타낼 수 있을 때  $a$ 와  $b$ 는  $N$ 의 약수이다. (단,  $a \leq b$ )

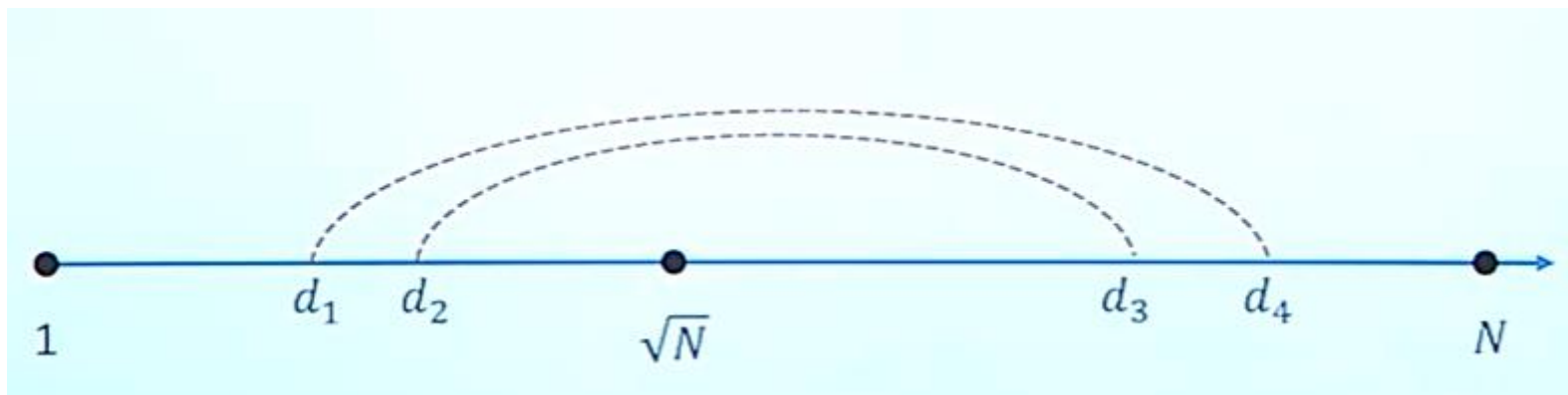
$$N = a \times b$$

- 자연수는 항상  $\sqrt{N}$ 이하의 자연수 하나와  $\sqrt{N}$  이상의 자연수 하나의 곱으로 표현된다.



# 소수의 판별

- 그러므로 자연수  $N$  이 소수인지 판별하기 위해서는  $2 \sim \sqrt{N}$  범위만 조사하면 됨.
  - 소수가 아니라면  $2 \sim \sqrt{N}$  범위에 약수가 발견되어야 함.
  - 만약  $2 \sim \sqrt{N}$  범위에 약수가 없으면,  $\sqrt{N} \sim N$  범위에도 약수가 없음이 보장됨.
  - 즉,  $2 \sim \sqrt{N}$  범위에 약수가 없으면 소수 판정.
- 위 방법은  $O(\sqrt{N})$ 의 시간 복잡도를 가짐.



# 소인수 분해

- 자연수 N을 소수들의 곱만으로 표현하는 것.
- 숫자를 소인수 분해하면...

1. 약수의 경우의 수를 쉽게 판별할 수 있다.
2. 숫자들의 최대 공약수 등을 계산하기 쉬워진다.

예시>

a.  $36 = 2 \times 2 \times 3 \times 3$

b.  $72 = 2 \times 2 \times 2 \times 3 \times 3$

c.  $60 = 2 \times 2 \times 3 \times 5$

# 소인수 분해 알고리즘

- 소수 약수를 차례로 구해가며 나누어 떨어지지 않을 때까지 계속 나누어 간다.

1. 소인수 분해 할 자연수를  $N$ , 소인수들의 집합을  $L$ 이라고 하자.
2. 약수 후보를  $M$ 이라고 하자. 초기값은 2이다.
3.  $M \leq \sqrt{N}$  이하 일 동안 아래 과정을 반복한다.
  1.  $M$ 이  $N$ 의 약수 인 동안 아래 과정을 계속 반복
    1.  $L$ 에  $M$ 을 추가한다.
    2.  $N$ 을  $M$ 으로 나눈다.
  2.  $M$ 을 1증가 시킨다.
4.  $L$ 에  $M$ 을 추가한다.
5.  $L$ 이 소인수들의 리스트가 된다.

- 위 방법은  $O(\sqrt{N})$ 의 시간 복잡도를 가짐.

## 소인수 분해 알고리즘

- 소인수 분해 알고리즘을 직접 수행해보자.

N	M	L
980		

## - 소인수분해 문제해결 실습

# 에라토스테네스의 체 알고리즘

- 한 자연수  $N$ 에 대하여  $1 \sim N$  범위 모든 숫자의 소수 여부를 구하는 전처리 알고리즘.

1.  $O(N \log \log N)$ 의 시간 복잡도로 모든 숫자의 소수 여부를 계산할 수 있다.
  1. 전처리 이후  $O(1)$ 만에 소수 여부를 검사할 수 있다.
2.  $O(N)$ 의 공간 복잡도를 가진다. 그러므로 너무 큰 숫자에는 적용할 수 없다.
3.  $1 \sim N$  범위 각각 모든 숫자들에 대해 가장 작은 소인수를 구해 둘 수 있다.
  1. 전처리 이후  $O(1)$ 만에 해당 소인수를 구해올 수 있다.
  2. 이를 활용해 어떤 자연수  $M$ 을  $O(\log_2 M)$ 만에 소인수분해 할 수 있다.



# 에라토스테네스의 체 알고리즘

- $1 \sim N$  범위의 모든 숫자의 소수 여부를 판별하기 위해 아래 과정을 수행한다.

1. 배열을 만들어  $2 \sim N$ 을 모두 소수라고 체크 해 둔다.
2.  $M$ 을  $2 \sim N$ 까지 차례로 대입하며 각각 아래 과정을 수행한다.
  1.  $M$ 이 소수가 아니라고 체크되어 있다면 건너 뛰고 종료한다.
  2.  $M^2 \sim N$ 범위에 있는  $M$ 의 배수들을 모두 소수가 아니라고 체크한다.  
이때 체크되는 숫자들의 입장에서는  $M$ 이 가장 작은 소인수가 된다.
3. 이후 배열에 체크 된 소수 여부를 사용하면 된다.

# 에라토스테네스의 체 알고리즘

- 다음 표로 에라토스테네스의 체 알고리즘을 수행해보자.

숫자	1	2	3	4	5	6	7	8	9	10
소수 여부	x									
소인수										

숫자	11	12	13	14	15	16	17	18	19	20
소수 여부										
소인수										

숫자	21	22	23	24	25	26	27	28	29	30
소수 여부										
소인수										

- 소수세기 문제해결 실습

# 예제 풀이



## 문제4A. 스도쿠보드

스도쿠란 아래의 그림 처럼 9행 9열의 보드에 1~9사이의 숫자들을 규칙에 맞게 채워넣는 게임을 말한다. 9행 9열의 보드에는 총 81개의 칸이 있는데 지수는 보통 각 칸을 1~81로 번호를 붙여서 구별한다. 가장 왼쪽 위의 (1행 1열의)칸이 1번 칸이 되고 그 이후 오른쪽으로 가면서 번호를 하나 증가시킨다. 그리고 해당 행이 다 차면 다음줄로 넘어가 이를 반복한다.

	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	10	11	12	13	14	15	16	17	18
3	19	20	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34	35	36
5	37	38	39	40	41	42	43	44	45
6	46	47	48	49	50	51	52	53	54
7	55	56	57	58	59	60	61	62	63
8	64	65	66	67	68	69	70	71	72
9	73	74	75	76	77	78	79	80	81

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

<스도쿠 판의 각 칸에 지수가 매긴 번호(왼쪽)와 각 칸이 속한 그룹(오른쪽) 그림>

스도쿠는 게임의 특징상 각 행과 각 열 그리고 3x3모양의 9칸으로 구성된 그룹에 1~9의 숫자가 하나씩만 들어가야 한다는 규칙이 있다. 그래서 각 칸에 숫자를 배치할 때 그 칸이 속한 그룹을 파악하고 규칙에 맞는지 확인하는 작업이 필요하다. 하지만 지수와 같이 스도쿠 게임을 하고 있는 예인이는 지수가 말하는 칸의 번호로 그 칸이 몇행 몇열에 위치한 칸이고 또 몇 번 그룹에 속한 칸인지 매번 계산하기가 힘이 들었다. 하지만 소심한 예인이는 자신보다 연니인 지수에게 이를 말하지 못하고 당신에게 도움을 요청했다. 지수가 말한 칸의 번호가 주어질 때, 그 칸이 속한 행, 열, 그룹의 번호를 계산해주는 프로그램을 작성해보자.

# 문제 4A. 스도쿠보드

## 입력 형식

이 문제는 여러개의 테스트케이스로 구성되어 있다.

첫 줄에는 테스트케이스의 수  $T$ 가 주어진다.  $T$ 는 1이상 100이하의 자연수 중 하나이다.

각 테스트케이스에는 지수가 말한 칸의 번호가 1부터 81까지의 자연수로 공백없이 한 줄에 주어진다.

## 출력 형식

모든 테스트케이스 별로 각 테스트케이스를 두 줄로 출력한다.

- 테스트케이스의 첫 줄에는 `Case #<id>` 형식으로 테스트케이스의 번호를 출력한다. 대소문자와 공백에 주의한다.
- 테스트케이스의 두 번째 줄에는 지수가 말한 칸의 행, 열, 그룹 번호를 공백으로 구분된 세 개의 자연수로 출력한다. `r c g`

# 문제4A. 스도쿠보드

예시 1

입력

```
5↵
1↵
2↵
3↵
4↵
5↵
```



출력

```
Case_#1:↵
1_1_1↵
Case_#2:↵
1_2_1↵
Case_#3:↵
1_3_1↵
Case_#4:↵
1_4_2↵
Case_#5:↵
1_5_2↵
```



# 문제4A. 스도쿠보드

```
#include <stdio>

using namespace std;

const int MAX_ROW = 9;
const int MAX_COL = 9;

class SudokuBoard{
public:
    // 칸의 번호로 행의 번호를 계산하는 메소드
    int getRowByIndex(int index) {

    }

    // 칸의 번호로 열의 번호를 계산하는 메소드
    int getColByIndex(int index) {

    }

    // 칸의 번호로 그룹 번호를 계산하는 메소드
    int getGroupByIndex(int index) {
        int r = getRowByIndex(index);
        int c = getColByIndex(index);
        return getGroupByPosition(r, c);
    }

    // 칸의 위치 (행, 열)로 그룹 번호를 계산하는 메소드
    int getGroupByPosition(int row, int column) {

    }

    // 칸의 위치 (행, 열)로 칸의 번호를 계산하는 메소드
    int getIndexByPosition(int row, int column) {

    }
};
```

```
void process(int caseIndex) {
    int index;
    scanf("%d", &index);

    SudokuBoard board = SudokuBoard();

    // 칸의 번호로 행, 열, 그룹 번호를 계산한다
    int row = board.getRowByIndex(index);
    int col = board.getColByIndex(index);
    int group = board.getGroupByIndex(index);

    printf("Case-#%d:\n", caseIndex);
    printf("%d-%d-%d\n", row, col, group);
}

int main() {
    int caseSize;
    scanf("%d", &caseSize);

    for (int caseIndex = 1; caseIndex <= caseSize; ++caseIndex) {
        process(caseIndex);
    }
}
```

## 문제4B. Probing

행사 기획자인 수정이는 이 번에 자신이 담당하게 된 행사에서 행운권 추첨을 기획하고 있다. 이번 행사에서는 최대 수천명 정도의 사람이 방문할 수 있기 때문에 어떻게 하면 공평하고 불규칙적으로 행운권 번호를 배정할 수 있을지가 큰 관건이었다. 하지만 똑똑한 수정이는 아래와 같은 아이디어를 냈다.

- 모든 행운권 번호는  $0 \sim (N-1)$ 의 정수로 총  $N$ 개이다.
- 모든 고객은 회원번호를 가지고 있으며 회원 번호는 자연수이다.
- 입장한 고객은 자신의 회원 번호를  $N$ 으로 나눈 나머지를 계산해 그 번호와 같은 행운권을 지급받는다.
  - 해당 행운권이 다른 사람에게 지급된 상황이라면 그것보다 +1한 번호를 지급받는다.
  - 아직 아무에게도 지급되지 않은 번호를 찾을 때 까지 행운권 번호를 +1씩 증가시켜가며 찾는다.
  - 만약  $(N-1)$ 번 행운권도 이미 지급된 상태라면 0번 행운권부터 다시 조회한다.
  - 이렇게 순서대로 조회하다가 가장 먼저 발견된 아직 지급되지 않은 행운권을 지급받는다.
- 고객들은 순서대로 한 명씩 입장하며 한번 지급된 행운권 번호는 교환할 수 없다.



<N=5000일 때 행운권 번호를 지급받는 예시>

수정이는 행사 중간에도 바뀔 예정이기에 당신에게 이를 자동화할 수 있는 프로그램을 작성해달라고 요청했다. 수정이를 도와주자.



# 문제 4B. Probing

## 입력 형식


첫 줄에는 준비한 행운권의 수  $N$ 과 입장 할 회원의 수  $M$ 이 공백으로 구분되어 주어진다.  $N$ 은 1이상 5,000이하의 자연수이며  $M$ 은 1이상 1,000이하의 자연수이다. 또한,  $M$ 은 항상  $N$ 이하의 값을 가진다.

이후 총  $M$ 줄에 걸쳐서 입장 한 회원들의 회원번호가 순서대로 주어진다. 각 회원번호는 1이상 1억 이하의 자연수이다.

## 출력 형식

입장한 회원들의 순서대로 해당 회원이 지급받게 될 행운권 번호를 한 줄에 하나 씩 정수 형태로 출력한다.

입/출력 예시

 : 공백     : 줄바꿈     : 탭

예시 1

입력

```
5000_5
2878
15092880
1
18762879
77787879
```

출력

```
2878
2880
1
2879
2881
```

# 문제 4B. Probing

```
#include <stdio>
#include <vector>

using namespace std;

class TicketTable {
public:
    vector<bool> used;
    int length;

    TicketTable(int length) {
        this->length = length;
        this->used.assign(length, false);
    }

    /**
     * 사용자의 회원 번호로 지급받게 될 행운권 번호를 계산하는 메소드
     */
    int findEmptyIndexByUserId(int userId) {
    }

    /**
     * 해당 행운권 번호가 이미 사용 중인지 여부를 반환하는 메소드
     */
    bool isEmpty(int ticketIndex) {
    }

    /**
     * 티켓 사용 여부를 갱신하기 위한 메소드
     */
    void setUsed(int index, bool status) {
    }
};
```

```
vector<int> getTicketNumbers(int n, int m, const vector<int>& ids) {
    vector<int> tickets;
    TicketTable table = TicketTable(n);

    for(int i = 0; i < m; i++) {
    }

    return tickets;
}

int main() {
    // n: 전체 티켓의 수
    // m: 요청 고객의 수
    int n, m;
    scanf("%d%d", &n, &m);

    vector<int> ids(m);
    for (int i = 0; i < m; ++i) {
        scanf("%d", &ids[i]);
    }

    vector<int> tickets = getTicketNumbers(n, m, ids);
    for (int i = 0; i < tickets.size(); ++i) {
        printf("%d\n", tickets[i]);
    }
}
```

# 문제4E. 소인수분해

자연수를 소인수 분해한 결과를 출력하는 프로그램을 작성해보자. 입력으로 주어진 자연수를 소인수분해 한 후 해당 숫자의 소인수들을 나열하면 된다.

## 입력 형식

첫 줄에는 테스트케이스의 수  $T$ 가 주어진다.  $T$ 는 1이상 100이하의 자연수이다.

각 테스트케이스는 한 줄로 구성되며 소인수 분해를 할 자연수가 주어진다. 이 자연수는 2이상 10억이하이다.

## 출력 형식

각 테스트케이스에 대한 정답을 두 줄씩 출력한다.

- 테스트케이스의 첫 줄에는 테스트케이스의 번호를 `##d:` 와 같은 형식으로 출력한다.
- 두 번째 줄에는 입력으로 주어진 숫자들의 소인수들을 공백으로 구분하여 오름차순으로 출력한다. 여러 번 곱해진 소인수는 그 횟수 만큼 출력한다.

입/출력 예시

 : 공백    : 줄바꿈    : 탭

예시 1

입력

```
3
24
28
21
```



출력

```
#1:
2 2 2 3
#2:
2 2 7
#3:
3 7
```



## 문제4E. 소인수분해

```
#include <cstdio>
#include <vector>

using namespace std;

/**
 * 자연수 N을 구성하는 모든 소인수를 반환하는 함수
 *
 * @param N
 * @return
 */
vector<long long> factorize(long n) {
}
```

```
void process(int caseIndex) {
    long long n;
    scanf("%lld", &n);

    vector<long long> factors = factorize(n);

    printf("#%d:\n", caseIndex);
    for (int i = 0; i < factors.size(); ++i) {
        if (i > 0) {
            printf(" ");
        }
        printf("%lld", factors[i]);
    }
    printf("\n");
}

int main() {
    int caseSize;
    scanf("%d", &caseSize);

    for (int caseIndex = 1; caseIndex <= caseSize; ++caseIndex) {
        process(caseIndex);
    }
}
```

# 문제4F. 소수세기

주어진 범위에 존재하는 소수의 수를 출력하는 프로그램을 작성해보자.

## 입력 형식

첫 줄에는 테스트케이스의 수  $T$ 가 주어진다.  $T$ 는 1이상 10이하의 자연수이다.

각 테스트케이스의 입력으로는 한 줄에 1이상 100만이하의 두 자연수  $L$ 과  $R$ 이 공백으로 구분되어 주어진다.

- $L$   $R$  순서로 주어지며  $L$ 은 항상  $R$ 이하의 값이다.

## 출력 형식

각 테스트케이스에 대하여 두 줄씩 정답을 출력한다.

- 테스트케이스의 첫 줄에는 테스트케이스의 번호를 `Case #<id>` 형식으로 출력한다.
- 두 번째 줄에는  $L$ 이상  $R$ 이하의 소수의 갯수를 공백없이 출력한다.

입력

```
3↵
2_10↵
50_100↵
100_1000↵
```

출력

```
Case_#1:↵
4↵
Case_#2:↵
10↵
Case_#3:↵
143↵
```

# 문제4F. 소수세기

```
#include <stdio>
#include <vector>

using namespace std;

//소인수 분해를 빠르게
class Sieve {
public:
    int maximumValue; //에라토스테네스의 체에서 다룰 가
    vector<bool> isPrime; //각 숫자별 소수 여부
    Sieve(int maximumValue) {
        this->maximumValue = maximumValue;
        this->isPrime.assign(maximumValue + 1, false);
        this->fillSieve();
    }

    /**
     *
     * @param num
     * @return 'num'이 소수라면 true, 그렇지 않으면 false
     */
    bool isPrimeNumber(int num) const {
        return this->isPrime[num];
    }

    /**
     * isPrime 배열의 값을 채우는 함수
     */
    void fillSieve() {
    }
};
```

```
vector<int> getAllPrimeNumbers(int from, int to, const Sieve& sieve) {
    vector<int> primes;

    for(int num = from; num <= to; num += 1) {
        if(sieve.isPrimeNumber(num)) {
            primes.push_back(num);
        }
    }

    return primes;
}

void process(int caseIndex, const Sieve& sieve) {
    int L, R;
    scanf("%d%d", &L, &R);

    vector<int> allPrimeNumbers = getAllPrimeNumbers(L, R, sieve);

    printf("Case #d:\n", caseIndex);
    printf("%d\n", (int)allPrimeNumbers.size());
}

int main() {
    const int MAX_VALUE = 1000000;
    Sieve sieve = Sieve(MAX_VALUE);

    int caseSize;
    scanf("%d", &caseSize);

    for (int caseIndex = 1; caseIndex <= caseSize; ++caseIndex) {
        process(caseIndex, sieve);
    }
}
```

# 실습 과제

- 6주차 4개 문제해결 프로그램 작성
  - 스도쿠보드
  - Probing
  - 소인수분해
  - 소수세기
- 실습과제 게시판에 업로드할 것
  - 4개의 cpp 파일
  - 파일명은 스도쿠보드.cpp, Probing.cpp, 소인수분해.cpp, 소수세기.cpp로 할 것
  - Cpp 파일에 코드를 설명할 수 있는 본인만의 주석을 작성할 것