



# week1 - 수학

## 0 Big O 표기법

### 1 합 공식

#### 1. 일반적인 방법

반복문을 활용하는 알고리즘 - 1부터 N까지의 합

반복문을 활용하는 알고리즘 - N부터 M까지의 합

시간 복잡도:  $O(n)$

#### 2. 등차수열 합 공식 (가우스 합 공식)

 등차수열 합 공식이란?

합 공식을 활용한 알고리즘 - 1부터 N까지의 합

합 공식을 활용한 알고리즘 - N부터 M까지의 합

시간 복잡도:  $O(1)$

\* 관련 문제

### 2 피보나치 수열

#### 1. 반복문으로 구현하기

반복문을 활용한 알고리즘

시간 복잡도:  $O(n)$

#### 2. 재귀함수로 구현하기

 재귀함수란?

재귀함수 알고리즘

시간 복잡도:  $O(2^n)$

#### 3. 동적 프로그래밍 기법 활용

 동적 프로그래밍과 메모이제이션

DP의 Memoization을 활용한 알고리즘

시간 복잡도:  $O(n)$

\* 관련 문제

### 3 약수

#### 1. 일반적인 방법

반복문과 조건문을 활용한 알고리즘

시간 복잡도:  $O(n)$

#### 2. 효율적인 방법

 sqrt() 메서드

sqrt() 메서드를 사용한 알고리즘

시간 복잡도:  $O(\sqrt{n})$

\* 관련 문제

### 4 최대공약수 GCD(Greatest Common Divisor)

### 1. 일반적인 방법

반복문을 활용한 알고리즘

시간 복잡도:  $O(n)$

### 2. 유클리드 호제법으로 최대공약수 구하기

 유클리드 호제법이란?

유클리드 호제법을 사용한 알고리즘

시간 복잡도:  $O(\log N)$

## 5 최소공배수 LCM(Least Common Multiple)

### 1. 일반적인 방법

반복문을 활용한 알고리즘

시간 복잡도:  $O(n)$

### 2. 유클리드 호제법으로 최소공배수 구하기

유클리드 호제법을 활용한 알고리즘

시간 복잡도:  $O(\log N)$

\* **4, 5** 관련 문제

## 6 소수

### 1. 일반적인 방법

N 보다 작은 자연수들로 나누는 알고리즘

시간 복잡도:  $O(n)$

### 2. 제곱근을 활용하는 방법

$\sqrt{N}$  이하의 자연수들로 나누는 알고리즘

시간 복잡도:  $O(\sqrt{N})$

### 3. 에라토스테네스의 체 방법론

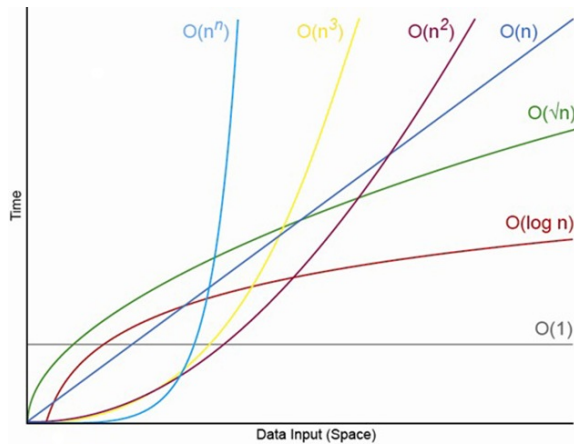
에라토스테네스의 체를 사용한 알고리즘

시간 복잡도:  $O(N \log(\log N))$

\* 관련 문제

## 0 Big O 표기법

알고리즘 성능을 수치화 하는 방법 중 하나인 시간 복잡도는 Big O로 측정하여 나타낼 수 있다.



## Big-O: functions ranking

BETTER

- $O(1)$  constant time
- $O(\log n)$  log time
- $O(n)$  linear time
- $O(n \log n)$  log linear time
- $O(n^2)$  quadratic time
- $O(n^3)$  cubic time
- $O(2^n)$  exponential time

WORSE

## 1 합 공식

- 1부터 N까지 or N부터 M까지의 합을 구하는 방법
  1. 일반적인 방법
  2. 등차수열 합 공식 (가우스 합 공식)

### 1. 일반적인 방법

#### 반복문을 활용하는 알고리즘 - 1부터 N까지의 합

```
int sum = 0;

for(int i = 1; i <= n; i++){
    sum += i;
}

System.out.println(sum);
```

- 반복문을 통해 N번 반복하며 sum을 갱신

#### 반복문을 활용하는 알고리즘 - N부터 M까지의 합

```
int sum = 0;
int start = Math.min(n, m);
int end = Math.max(n, m);

for(int i = start; i <= end; i++){
```

```
sum += i;
}
System.out.println(sum);
```

- 반복문의 시작과 끝 값을 설정하기 위해 n과 m 중 min과 max 값을 알아낸다.
- 반복문을 실행하며 sum 갱신

## 시간 복잡도: O(n)

- 필요한 계산 횟수가 입력 값의 크기와 비례한다.  
ex). 만약 입력 값이 10000이라면 for문을 10000번 돌려야 한다.
- 따라서 시간 복잡도 = O(n)

## 2. 등차수열 합 공식 (가우스 합 공식)

### 등차수열 합 공식이란?

첫째 항 a와 n번째 항 l을 알고 있을 때, 각 항이 일정한 값으로 증가한다면(= 등차수열이라면),

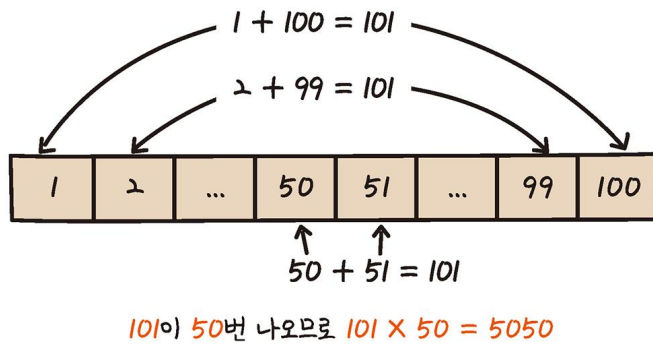
a부터 l까지의 합은  $n * (a + l) / 2$  이다.

#### ▼ 증명

a부터 l까지의 합 S를 정순과 역순으로 나열하여 더한 후 정리하면 도출 가능

$$\begin{aligned}
 S_n &= a + (a + d) + (a + 2d) + \cdots (l - d) + l \\
 +) S_n &= l + (l - d) + (l - 2d) + \cdots (a + d) + a \\
 \hline
 2S_n &= (a + l) + (a + l) + (a + l) + \cdots (a + l) + (a + l) \\
 \therefore S_n &= \frac{n(a + l)}{2}
 \end{aligned}$$

- ① 이를 1부터 N까지의 합으로 일반화 시키면,  $n * (n + 1) / 2$  가 된다.  
ex). 첫째 항 1부터 1씩 증가하여 100번째 항이 100인 수열의 총 합은  $100(100+1)/2=5050$ 이다.



- ② 이를 N부터 M까지의 합으로 일반화 시키면,  $(m - n + 1) * (n + m) / 2$  가 된다.

## 합 공식을 활용한 알고리즘 - 1부터 N까지의 합

```
System.out.println((n * (n + 1) / 2);
```

- 공식을 통해 한번에 반환 가능

## 합 공식을 활용한 알고리즘 - N부터 M까지의 합

```
int sumNtoM(int n, int m){
    return (m - n + 1) * (n + m) / 2;
}
System.out.println(sumNtoM(Math.min(n, m), Math.max(n, m)));
```

- min과 max만 판별하여 공식을 통해 한번에 반환 가능

## 시간 복잡도: O(1)

- 입력 값의 크기에 상관 없이 일정한 시간(상수 시간)이 소요된다.  
어떤 값을 입력하든 공식을 통해 한번만 계산하면 된다.

→ 따라서 시간 복잡도 = O(1)

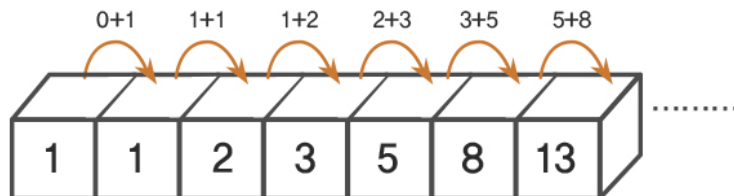
## \* 관련 문제

[BOJ 8393](#)

[프로그래머스 12912](#)

## 2 피보나치 수열

: 첫째 항, 둘째 항이 1이고 그 뒤의 항은 바로 앞의 두 항의 합인 수열



- 피보나치 수열 구현 방법
  1. 반복문으로 구현
  2. 재귀함수로 구현
  3. 동적프로그래밍 기법 활용

### 1. 반복문으로 구현하기

#### 반복문을 활용한 알고리즘

```
int num1, num2, sum;
num1 = 0;
num2 = 1;
sum = 1;

for(int i = 0; i < n; i++) // input 값 n
{
    System.out.print(sum + " ");
    sum = num1 + num2;
    num1 = num2;
    num2 = sum;
}
```

- 첫번째와 두번째 값을 1로 설정하기 위해 num1과 num2와 sum을 초기화한다.
- 반복문을 돌려서 앞의 두 항을 더해가며 sum 값을 구하고 num1과 num2를 갱신한다.

- sum 값을 num2에 갱신해준다.

## 시간 복잡도: $O(n)$

- 중복 계산 없이  $n$ 번 반복하면 된다.

→ 따라서 시간 복잡도 =  $O(n)$

## 2. 재귀함수로 구현하기

### 📌 재귀함수란?

: 자기 자신을 다시 호출하는 함수

### 재귀함수 알고리즘

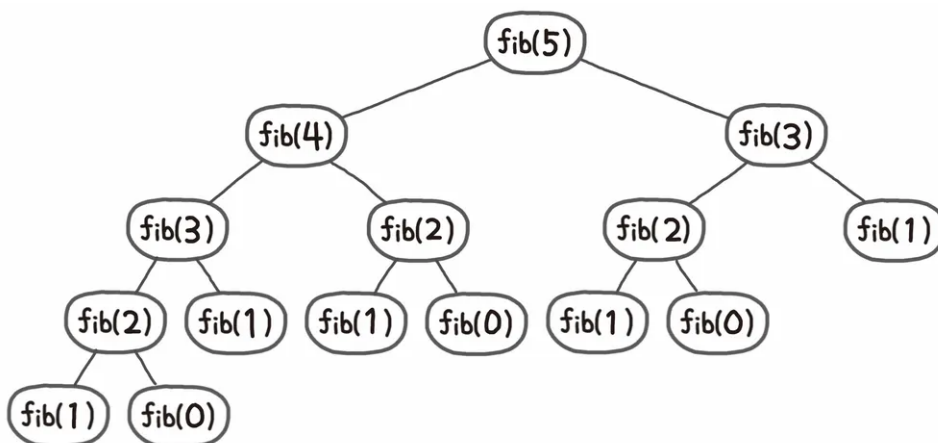
```
int fib(int n) {
    if(n <= 1) {
        return n;
    } else {
        return fib(n - 2) + fib(n - 1);
    }
}
```

- 다음 항을 알기 위해서 앞의 두 개의 항을 알아야 한다.

→  $n$ 번째 항을 알기 위해  $n-1$ ,  $n-2$ 번째 항을 재귀호출 한다.

## 시간 복잡도: $O(2^n)$

위의 코드를 그림으로 나타내면 다음과 같다.



top-down 형식

- 2번씩 트리의 높이 k만큼 반복하는 형태

즉, 구하려는 숫자가 커질 수록 호출이 2배로 증가하고 최대 값은 트리의 깊이인 N만큼 늘어남

➡ 따라서 시간 복잡도 =  $O(2^n)$

- 구하려고 하는 숫자가 커질수록 메모리 속도가 현저하게 떨어지므로 성능이 저하 됨

👉 중복되는 호출을 제거할 수 있는 방법인 **동적 프로그래밍**을 사용하여 해결할 수 있음

### 3. 동적 프로그래밍 기법 활용

#### 📌 동적 프로그래밍과 메모이제이션

**동적 프로그래밍이란?**

: Dynamic Programming이라고 하며, 큰 문제를 작은 문제로 나눠서 푸는 알고리즘

- 동적 프로그래밍의 특성 2가지
  - Overlapping SubProblem(겹치는 작은 문제)
 

: 큰 문제를 작은 문제로 쪼갤 때 작은 문제가 여러 번 재사용 되는 것  
ex). 피보나치 수열
  - Optimal Substructure(최적 부분 구조)
 

: 큰 문제의 정답을 작은 문제의 정답을 통해 풀이하는 것  
ex). 최단 경로 찾기

#### **메모이제이션(Memoization)**

: 동적 프로그래밍 기법 중 하나

작은 문제들을 하나씩 풀 때마다 정답을 저장(memo)해 두었다가, 이를 활용하여 더 큰 문제를 풀 때 작은 문제들을 다시 계산하지 않고 저장(memo)해 둔 값을 **재사용** 하는 방법

#### **DP의 Memoization을 활용한 알고리즘**

```
int[] memo;

int fiboDP(int n) {
    if (memo[k] != 0) { // 이미 계산된 값
        return memo[k];
    } else { // 처음 계산되는 값
```



```

        if (k <= 1) {
            // 0번 항과 1번 항은 1로 초기화
            memo[k] = 1;
        } else {
            // 3번째 항 부터는 계산하여 저장
            memo[k] = fiboDP(k - 1) + fiboDP(k - 2);
        }
        return memo[k];
    }
}

```

- 계산된 결과 값을 담을 memo 배열을 생성
- 이미 계산된 index에 대해서는 memo에 저장된 값을 return
- 처음 계산 되는 index의 경우에는 계산 후 memo 배열에 저장하고 반환

## 시간 복잡도: $O(n)$

- 계산된 값을 메모리에 저장해둠으로써 중복을 제거하여 필요 시마다 값을 불러오기만 하면 된다.
- 입력 값  $n$ 에 대해  $n$ 번의 함수 호출이 일어나고, 상수 시간의 계산이 소요된다.

➡ 따라서 시간 복잡도 =  $O(n)$

## \* 관련 문제

[BOJ 2747](#)

[BOJ 2748](#)

## 3 약수

: 어떤 수를 나누어 나머지가 없이 떨어지게 하는 수

ex). 8을 1, 2, 4, 8 로 나누면 나누어 떨어진다. 이때 1, 2, 4, 8은 8의 약수이다.

- $N$ 의 약수를 구하는 법
  1. 일반적인 방법 - 반복문과 조건문 사용
  2. 효율적인 방법 - sqrt 메서드 활용

## 1. 일반적인 방법

### 반복문과 조건문을 활용한 알고리즘

```
int n = 100;

for(int i = 1; i <= n; i++){
    if(n % i == 0){
        System.out.println(i + "는 약수 입니다.");
    }
}
```

- n을 1부터 n까지 하나씩 나누어 가며 나머지가 0인지 판별

### 시간 복잡도: $O(n)$

- 필요한 계산 횟수가 입력 값의 크기와 비례한다.  
ex). 만약 입력 값이 10000이라면 for문을 10000번 돌려야 한다.  
→ 따라서 시간 복잡도 =  $O(n)$

## 2. 효율적인 방법

- 1번(일반적인 방법)의 코드에서 `n % i == 0`의 의미는  $n/i$  한 수와  $i$ 의 곱은  $n$ 이라는 것이다.  
이는  $n$ 의 약수가  $i$ 일 때, 다른 하나의 약수는  $n/i$ 가 된다는 의미이다.
- 주어진  $N$ 에 대하여,  $\sqrt{N}$ 까지 수 중  $N$ 의 약수를 구하면 총 약수의 절반을 구할 수 있다.  
(약수가  $\sqrt{N}$ 인 경우는 제곱근이므로 1개로 카운트)



증명은 [여기](#) 참고

➡ 이를 활용하면 계산 횟수를 줄일 수 있다.

## 📌 sqrt() 메서드

: 제곱근(root)을 구하는 함수

- java.lang.Math 클래스의 함수
  - Math 클래스 메서드들은 모두 static 메서드이므로 import와 선언 없이 바로 사용 가능
- sqrt는 Square root를 의미
- 인자로 a를 전달하면 a의 제곱근이 double 형으로 return
- 인자로 0을 전달하면 0이 return
- 인자로 음수나 NaN(Not a Number)를 전달하면 NaN이 return

## sqrt() 메서드를 사용한 알고리즘

```
int n = 100;
int sqrt = (int) Math.sqrt(n);
ArrayList<Integer> arr = new ArrayList<>(); // 약수를 저장할 ArrayList

for(int i = 1; i <= sqrt; i++){
    if(n % i == 0){ // 약수 중 작은 수 저장
        arr.add(i);
        if(n / i != i){ // 약수 중 큰 수 저장
            arr.add(n / i);
        }
    }
}
```

- for문 조건을 sqrt 메서드를 사용한 변수로 정의
- `n % i == 0` 이면, 약수 중 작은 수인 i를 배열에 저장
- `if(n / i != i)` 가 true라면 n / i도 또 다른 약수가 되므로 배열에 저장
- `if(n / i != i)` 가 false라면 n의 제곱근에 해당하는 약수이므로 중복 저장하지 않음

#### ▼ 설명

- for문 조건을 sqrt 메서드를 사용한 변수로 정의
- 원래라면 1 ~ 100까지 돌아야 할 for문이 1 ~ 10 까지만 돌아도 약수 추출이 가능해졌다.
- 입력 값 100의 약수는 1, 2, 4, 5, 10, 20, 25, 50, 100 이렇게 9개이다.
- 10번의 for문을 돌며 9개의 약수 중 1을 구한다 (약수 중 작은 수)
- 1이 100의 약수라는 것을 알게 된 순간 100이라는 약수도 구할 수 있게 된다.
- 그 부분이 `if(n / i != i)` 부분이다. `100 / 1 != 1` 이라면 arr 리스트에 100 / 1 의 값을 넣어준다(약수 중 큰수)
- 이 과정을 반복한다.
- 10 번의 for 문을 돌며 9개의 약수 중 2을 구한다 (약수 중 작은 수)
- 2이 100의 약수라는 것을 알게 된 순간 50이라는 약수도 구할 수 있게 된다.
- 그 부분이 `if(n / i != i)` 부분이다. `100 / 2 != 2` 이면 arr 리스트에 100 / 2 의 값을 넣어준다(약수 중 큰수)
- 해당 부분은 약수 10을 주의 깊게 봐야한다 `10 * 10`이 100 이므로 10은 한번만 arr 리스트에 한번만 넣어야한다.
- 그 부분이 `if(n / i != i)` 부분이다. `100 / 10 != 10` false 이기 때문에 배열에 한번만 들어간다.
- for 문을 한번 돌 때 약수 중 작은 수 , 큰 수를 모두 arr 리스트에 넣기 때문에 for 문이 끝난 뒤 arr 리스트는
- 1, 100, 2, 50, 4, 25, 5, 20, 10 이런 식으로 값이 배열되어 있다.
- `arr.sort()`를 사용해 정렬을 진행한다.

### 시간 복잡도: $O(\sqrt{n})$

- 필요한 계산 횟수가 입력 값의 제곱근과 비례한다.

ex). 만약 입력 값이 100이라면 for문을 10번 돌리면 된다.

→ 따라서 시간 복잡도 =  $O(\sqrt{n})$

## \* 관련 문제

프로그래머스 12928

BOJ 1037

## 4 최대공약수 GCD(Greatest Common Divisor)

: 두 자연수의 공통된 약수 중 가장 큰 수를 의미

ex). 60과 48의 최대공약수는 12이다.

60과 48의 최대공약수 구하기

최대공약수는  
 $2 \times 2 \times 3 = 12$

2)	60	48
2)	30	24
3)	15	12
	5	4

5와 4는 서로소이다.

- 두 수의 최대공약수를 구하는 법
1. 일반적인 방법 - 반복문 돌리기
  2. 유클리드 호제법 이용

## 1. 일반적인 방법

### 반복문을 활용한 알고리즘

```
Scanner sc = new Scanner(System.in);
System.out.println("두 수를 입력하세요.");
a = sc.nextInt();
b = sc.nextInt();
```

```
int min = (a < b) ? a : b;
int gcd = 0;
for (int i = 1; i <= min; i++) {
    if (a % i == 0 && b % i == 0)
        gcd = i;
}
System.out.println("최대공약수 : " + gcd);
```

- 삼항연산자를 이용해 min과 max 값을 구분
- 1부터 min까지 반복하며 나눴을 때 나머지가 0인 값을 갱신

## 시간 복잡도: $O(n)$

- 필요한 계산 횟수가 입력 값의 크기와 비례한다.

ex). 만약 min 값이 6048이라면 for문을 6048번 돌려야 한다.

→ 따라서 시간 복잡도 =  $O(n)$

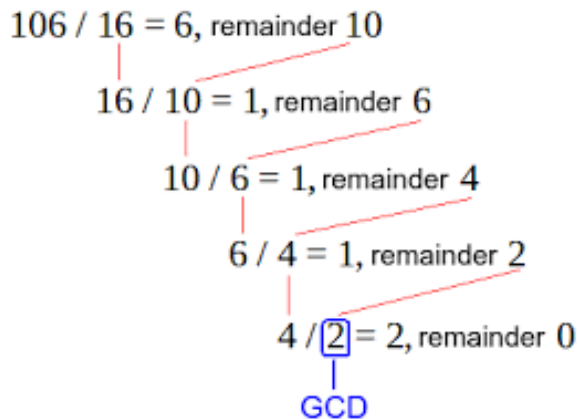
- 어떤 수의 약수가 매우 많을 경우 인수 분해 하는 데 시간을 많이 필요로 하게 되며, 두 수를 비교한 뒤 다시 곱해주는 것에도 많은 시간을 소비하게 되므로 아래의 유클리드 알고리즘을 활용하는 것이 더 효율적이다.

## 2. 유클리드 호제법으로 최대공약수 구하기

### 유클리드 호제법이란?

: 큰 숫자를 작은 숫자로 나누고, 그 나머지로 작은 숫자를 나누는 계산을 나머지가 0이 될 때까지 반복하는 것이다.

ex).




1. 두 수 중 큰 수를 작은 수로 나누어 나머지를 구한다. ( $106 \% 16 = 10$ )

2. 작은 수를 방금 구한 나머지로 나눈다. ( $16 \% 10 = 6$ )
3. 나머지가 0이 될 때까지 반복한다.
4. 나머지가 0이 될 때 나눴던 수가 최대공약수가 된다.

## 유클리드 호제법을 사용한 알고리즘

```
private int gcd(int max, int min) {
    while (max % min != 0) {
        int temp = max % min;
        max = min;
        min = temp;
    }
    return min;
}
```

- 두 개의 수 중 큰 수를 max로, 작은 수를 min으로 설정
- `max % min = N` 구하기
- `min == 0` 이면 `B` 는 최대공약수
- `N != 0` 이면 `max = min, min = N` 으로 대입하고 모듈러 연산 과정 반복 (recursive)

 모듈러 연산을 반복하는 recursive한 알고리즘이므로 재귀함수로도 구현할 수 있다.

```
public int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    return gcd(b, a % b);
}
```

## 시간 복잡도: $O(\log N)$

- 어떠한 수로 계속 나누어 나가는 과정을 반복한다.
- ➡ 따라서 시간 복잡도 =  $O(\log N)$
- cf). 재귀 호출의 경우 모듈러 연산을 반복하는 것보다 공간 복잡도가 늘어난다.

## 5 최소공배수 LCM(Least Common Multiple)

: 두 자연수의 공통된 배수 중 가장 작은 수를 의미

ex). 60과 48의 최소공배수는 240이다.  $(60 * 48) / 12 = 240$



최대공약수와 최소 공배수의 관계?

최소공배수 = 두 자연수의 곱 / 최대공약수

$$\begin{array}{r} G \overline{) \begin{array}{cc} A & B \\ a & b \end{array}} \end{array}$$

$$A = G \times a$$

$$B = G \times b$$

$$L = G \times a \times b$$

L: 최소공배수(LCM)

G: 최대공약수(GCD)

- 두 수의 최소공배수를 구하는 법 (최대공약수를 구하는 법과 동일)
1. 일반적인 방법 - 반복문 돌리기
  2. 유클리드 호제법 이용

### 1. 일반적인 방법

반복문을 활용한 알고리즘

```
int min = (a < b) ? a : b;
int gcd = 0;
for (int i = 1; i <= min; i++) {
    if (a % i == 0 && b % i == 0)
        gcd = i;
}
System.out.println("최소공배수 : " + a * b / gcd);
```

- 삼항연산자를 이용해 min과 max 값을 구분
- 최대공약수 gcd를 구하여 최소공배수 계산

시간 복잡도:  $O(n)$



- 필요한 계산 횟수가 입력 값의 크기와 비례한다.

→ 따라서 시간 복잡도 =  $O(n)$

## 2. 유클리드 호제법으로 최소공배수 구하기

### 유클리드 호제법을 활용한 알고리즘

```
// 최대공약수 gcd: 유클리드 호제법 이용
private int gcd(int max, int min) {
    while (max % min != 0) {
        int temp = max % min;
        max = min;
        min = temp;
    }
    return min;
}

// 최소공배수 lcm
int lcm(int a, int b) {
    return a * b / gcd(a, b);
}
```

- 유클리드 호제법을 활용하여 최대공약수를 구한다.
- 최소공배수는 주어진 두 수를 곱한 값을 유클리드 호제법을 통해서 구한 최대공약수로 나눈 값이 된다.

### 시간 복잡도: $O(\log N)$

- 어떠한 수로 계속 나누어 나가는 과정을 반복한다.

→ 따라서 시간 복잡도 =  $O(\log N)$

### \* 4, 5 관련 문제

[BOJ 2609](#)

[BOJ 5347](#)

[BOJ 1934](#)

[BOJ 13241](#)

[프로그래머스 12940](#)

## 6 소수

: 1보다 큰 자연수 중 1 과 그 수 자기 자신만을 약수로 갖는 자연수

즉, 소수의 약수는 반드시 2개이며, 그 중 하나는 반드시 1이고, 다른 하나는 자기 자신이다.

ex). 2는 1과 2만을 약수로 갖는 소수이지만, 10은 1, 2, 5, 10을 약수로 가지므로 소수가 아니다.

- 소수를 판별하거나 구하기 위한 알고리즘
  1. 가장 일반적인 방법
  2. 제곱근을 활용하는 방법
  3. 에라토스테네스의 체 방법론

### 1. 일반적인 방법

임의의 수  $N$ 이 1 과  $N$ 을 제외한 다른 수를 약수로 갖고 있다면 그 수는 소수가 아니고, 다른 약수가 없다면 그 수는 소수일 것이다.

→ 2 이상  $N$  미만의 수 중에 나누어 떨어지는 수가 존재한다면 소수가 아님을 이용하여 소수를 판별할 수 있다.

### N 보다 작은 자연수들로 나누는 알고리즘

```
// 0과 1 은 소수가 아니다
if(number < 2) {
    System.out.print("소수가 아닙니다");
    return;
}

// 2는 소수다
if(number == 2) {
    System.out.print("소수입니다");
    return;
}

for(int i = 2; i < number; i++) {
    // 소수가 아닌 경우
    if(number % i == 0) {
        System.out.print("소수가 아닙니다");
        return;
    }
}
```

```

    }
}

// 위 반복문에서 약수를 갖고 있지 않으면 소수
System.out.print("소수입니다");
return;
}

```

- number라는 (입력 받은) 수가 소수인지 아닌지 판별하는 로직
- case1) number가 0이나 1인 경우, 소수가 아님을 출력
- case2) number가 2인 경우, 소수임을 출력
- case3) number가 2 이상인 경우, 2부터 n 이하의 수까지 나누어 가며 소수 판별

## 시간 복잡도: $O(n)$

- 필요한 계산 횟수가 입력 값의 크기와 비례한다.  
반복문으로 N 미만의 수까지 모든 수를 검사하므로,  
➡ 따라서 시간 복잡도 =  $O(n)$

## 2. 제곱근을 활용하는 방법

소수를 판별한다는 것은 결국 1과 자기 자신을 제외한 다른 자연수를 약수로 갖고 있으면 안 된다는 의미다.

임의의 자연수  $N$  ( $N > 0$ )이 있다고 가정하자.

$p \times q = N$  을 만족할 때,  $p$ 와  $q$ 의 범위는  $1 \leq p, q \leq N$  이다.

그리고 이 때,  $p$ 와  $q$  중 하나는  $\sqrt{N}$  보다 작거나 같다.

▼ ex).

예를 들어  $N = 16$  일 때,  $N$ 을 두 수  $p$ 와  $q$ 의 곱으로 표현하면,

$N = 16 = 1 \times 16 = 2 \times 8 = 4 \times 4 = 8 \times 2 = 16 \times 1$  으로 나타낼 수 있다.

여기서 볼 수 있듯이,  $p$ 와  $q$ 는  $N$ 의 약수이며,  $p$ 의 값이 증가(감소)하면  $q$ 의 값이 감소(증가)한다.

따라서  $N$ 이 임의의 수( $p$ )로 나누어질 때, 나눈 임의의 수( $p$ )가  $\sqrt{N}$  보다 작거나 같다면 결국 몫에 해당하는 값( $q$ )은  $\sqrt{N}$  보다 클 수 밖에 없고, 두 수  $p$ 와  $q$ 는  $N$ 의 약수가 된다.

결과적으로  $p$ 와  $q$  중 하나는 반드시  $\sqrt{N}$  보다 작거나 같다는 것이다.

➡ 즉,  $\sqrt{N}$  이하의 자연수 중에 나누어 떨어지는 수가 있다면, 이는 1과  $N$  을 제외한 다른 자연수가  $N$ 의 약수라는 의미이므로 소수가 아니게 되는 것이다.

## $\sqrt{N}$ 이하의 자연수들로 나누는 알고리즘

```
// 0과 1은 소수가 아님
if(number < 2) {
    System.out.print("소수가 아닙니다");
    return;
}

// 2는 소수
if(number == 2) {
    System.out.print("소수입니다");
    return;
}

// 제곱근 함수 Math.sqrt()
for(int i = 2; i <= Math.sqrt(number); i++) {
    // 소수가 아닌 경우
    if(number % i == 0) {
        System.out.print("소수가 아닙니다");
        return;
    }
}

// 위 반복문에서 약수를 갖고 있지 않으면 소수
System.out.print("소수입니다");
return;
}
```

- 1번의 경우와 유사한 로직
- 반복문의 조건만 2 이상  $\sqrt{N}$  이하의 수로 변경하여, 나누어 떨어지는 수가 존재한다면 소수가 아님을 출력

## 시간 복잡도: $O(\sqrt{N})$

- 필요한 계산 횟수가 입력 값의 제곱근과 비례한다.

반복문으로  $\sqrt{N}$  이하의 수까지 모든 수를 검사하므로,

➡ 따라서 시간 복잡도 =  $O(\sqrt{N})$

- cf).  $N$ 까지의 모든 소수를 출력해야 하는 문제의 경우, 위와 같은 알고리즘을 활용하면  $O(\sqrt{N})$  알고리즘을  $N$ 번 반복하므로 시간 복잡도가  $O(N\sqrt{N})$ 이다.

위와 같은 알고리즘은 그다지 효율적인 알고리즘은 아니므로, 입력 값이 큰 경우 시간 초과가 될 수 있다.

👉 따라서 에라토스테네스의 체를 활용하여 시간을 줄일 수 있다.

### 3. 에라토스테네스의 체 방법론

: 2부터  $\sqrt{N}$  이하까지 반복하여  $N$  이하의 자연수들 중  $k$ 를 제외한  $k$ 의 배수들을 제거한다.

	2	3	4	5	6	7	8	9	10	Prime numbers
11	12	13	14	15	16	17	18	19	20	
21	22	23	24	25	26	27	28	29	30	
31	32	33	34	35	36	37	38	39	40	
41	42	43	44	45	46	47	48	49	50	
51	52	53	54	55	56	57	58	59	60	
61	62	63	64	65	66	67	68	69	70	
71	72	73	74	75	76	77	78	79	80	
81	82	83	84	85	86	87	88	89	90	
91	92	93	94	95	96	97	98	99	100	
101	102	103	104	105	106	107	108	109	110	
111	112	113	114	115	116	117	118	119	120	

출처 : 위키백과

#### 에라토스테네스의 체 방법론

1. 2부터 소수를 구하고자 하는 구간의 모든 수를 나열한다.
2. 2는 소수이므로 오른쪽에 2를 쓴다.
3. 자기 자신을 제외한 2의 배수를 모두 지운다.
4. 남아있는 수 가운데 3은 소수이므로 오른쪽에 3을 쓴다.
5. 자기 자신을 제외한 3의 배수를 모두 지운다.
6. 남아있는 수 가운데 5는 소수이므로 오른쪽에 5를 쓴다.
7. 자기 자신을 제외한 5의 배수를 모두 지운다.
8. 남아있는 수 가운데 7은 소수이므로 오른쪽에 7을 쓴다.

9. 자기 자신을 제외한 7의 배수를 모두 지운다.

10. 위의 과정을 반복하면 구하는 구간의 모든 소수가 남는다.

👉 2부터 시작해서 해당 수가 소수이면 표시를 하고, 그 수의 배수들은 소수 후보에서 제외한다.

따라서 2부터 N까지의 수 중에서 소수를 찾다고 했을 때, N의 제곱근보다 작은 소수의 배수를 모두 지우고 남는 수는 모두 소수이다.

## 에라토스테네스의 체를 사용한 알고리즘

```
if(num <= 1) return;

boolean[] arr = new boolean[num+1];

// 해당 index가 소수이면 true, 소수가 아니면 false
arr[0] = arr[1] = false;

for(int i = 2; i <= num; i += 1) { // 2 이상인 index에 대해 배열 초기화
    arr[i] = true;
}

// 2부터 숫자를 키워가며 배수들을 제외
for(int i = 2; i * i <= num; i += 1) {
    for(int j = i * i; j <= num; j += i) {
        arr[j] = false; // i의 배수이면 false 할당
    }
}

System.out.println("Prime number list: ");
for(int i = 0; i <= num; i += 1) {
    if(true == arr[i]) { // 소수인 경우(true)만 출력
        System.out.print(i + " ");
    }
}
}
```

- num 이하의 수를 담기 위한 boolean형 배열 arr 선언한다.
- 배열 arr의 index 값은 0 이상, num 이하의 자연수 값과 대응된다.
- 에라토스테네스 방법론을 사용하여 해당 index가 소수이면 값을 true로, 소수가 아니면 false로 설정한다.

## 시간 복잡도: $O(N \log(\log N))$

- 1차적으로, N 이하의 수에 대해 각각의 수들의 배수를 체에 거르는 시간이  $\log N$ 이므로, 이에 대한 시간 복잡도는  $O(N \log N)$ 이 된다.

▼ 설명

1 ~ x 까지의 수가 있는 칸을 체크하는 횟수를 대략적으로 따진다면,

$(x) + (x/2) + (x/3) + (x/4) + (x/5) + (x/6) + \dots + 1$  와 같다.

위의 수식은  $x(1 + 1/2 + 1/3 + \dots + 1/x)$  와 같이 표현할 수 있다.

위의 x로 묶인 괄호 안의 수열인  $1 + 1/2 + 1/3 + \dots + 1/x$  은 조화 수(Harmonic Number; 조화 수열에서 부분 합)라고 하며, 다음과 같이 발산한다.

$$H_x = \sum_{n=1}^x \frac{1}{n} = \ln x + \gamma + O\left(\frac{1}{x}\right)$$

이 때 감마는 상수 값이고,  $O(1/x)$ 는 big O와 같은 표기로, 수학에서의 함수 성장률의 상한선이다.

따라서 조화 수는 대략 자연로그(ln)의 형태를 따라간다.

참고) 조화 수에 관한 더 자세한 내용

log는 자연로그 ln으로 보기 때문에 N 이하의 소수에 대하여 체에 거르는 시간은  $\log N$ 이 되는 것이다.

- 에라토스테네스의 방법론에서는 N 이하의 모든 수의 배수를 체로 거르는 것이 아니라, N 이하의 소수에 대한 배수만 체로 거르는 것이다.

▼ 설명

앞서 조화 수를 통해 점근적 시간 복잡도  $O(N \log N)$ 을 도출해냈다.

이는 총 x개의 수에 대해 2일 때는  $(x/2)$ 개를, 3일 때는  $(x/3)$ 개를 체크한다는 의미이다.

하지만 에라토스테네스의 방법론에서는 x 이하의 모든 수를 체크하지 않는다. 1차적으로 걸러진 배열은 검사하지 않고 다음 반복문으로 넘어가게 된다. 이를 수식에서 제거하면 다음과 같은 식이 된다.

$$(x/2) + (x/3) + \cancel{(x/4)} + (x/5) + \cancel{(x/6)} + (x/7) + \cancel{(x/8)} + \cancel{(x/9)} + \dots + (x/(x-1))$$

즉, 중복되는 수들은 검사하지 않을 것이므로, 검사하는 수는 소수로 판정될 때 그의 배수들을 지워야 한다는 것이다. 이는 구간 내의 소수의 개수를 알아야 한다는 의미이다.

이 때, 가우스 소수 정리의 'x보다 작거나 같은 소수의 밀도는 대략  $1/\ln(x)$ 이다.'라는 성질을 활용한다. 해당 성질을 거꾸로 해석하면 x번째 소수는  $x \log(x)$ 라는 의미이다.

앞서 구한  $1/x$  의 합에서 중복되는 수를 제외한다면  $x$ 는 소수만 된다는 의미이므로,  
이는  $(1/2 + 1/3 + 1/5 + 1/7 + \dots)$  처럼 분모가 소수인 역수들의 합이 된다.

→ 따라서 시간 복잡도 =  $O(N \log(\log N))$

## \* 관련 문제

BOJ 1929 - 소수 구하기

BOJ 1978 - 소수 찾기

BOJ 4948 - 베르트랑 공준

BOJ 3896 - 소수 사이 수열

BOJ 6588 - 골드바흐의 추측

BOJ 9020 - 골드바흐의 추측