개발 역량강화를 위한 알고리즘 정복 CAMP

CHAPTER02 – Time Complexity & Loop Optimization

Dongyi Kim
dykim@codingmonster.net

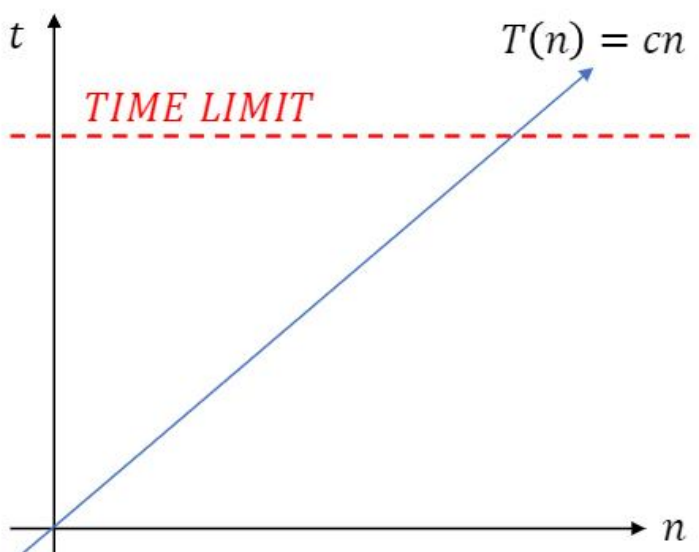
Time Cost

프로그램의 명령어는 모두 CPU에서 처리될 때 시간을 소요한다. 많은 명령어를 처리할 수록 전체 소요 시간이 증가하게 된다.

당연히 반복문은 반복 횟수에 비례해서 소요 시간이 증가한다.

간단한 코드도 반복 횟수가 많아지면 어느 수준 이상의 시간을 소요하게 된다.

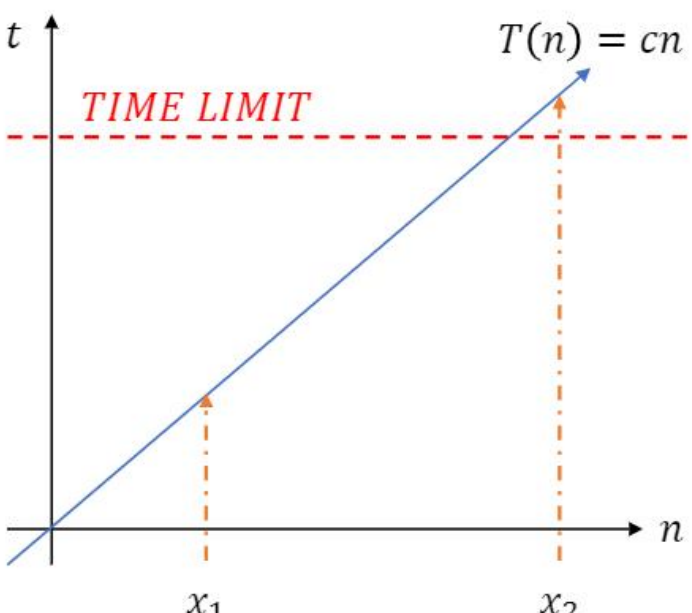
```
/**
 * 이 반복문은 N의 크기에 비례해 소요 시간이 증가한다.
 */
for(int i = 0 ; i < N; i++)
{

    SomeCode(); //반복문이 없는 상수시간 연산

}
```



Time Cost Anaysis

일반적으로 알고리즘을 설계 할 상황에서는 데이터의 수(혹은 크기)를 대략적으로 예측할 수 있는 경우가 대부분이다.

물론 코딩테스트 문제에서도 직/간접적으로 주어진다.



1) 데이터의 수 n이 항상 x_1이하라면, 이 알고리즘은 항상 제한시간 이내에 수행됨을 보장할 수 있는가?

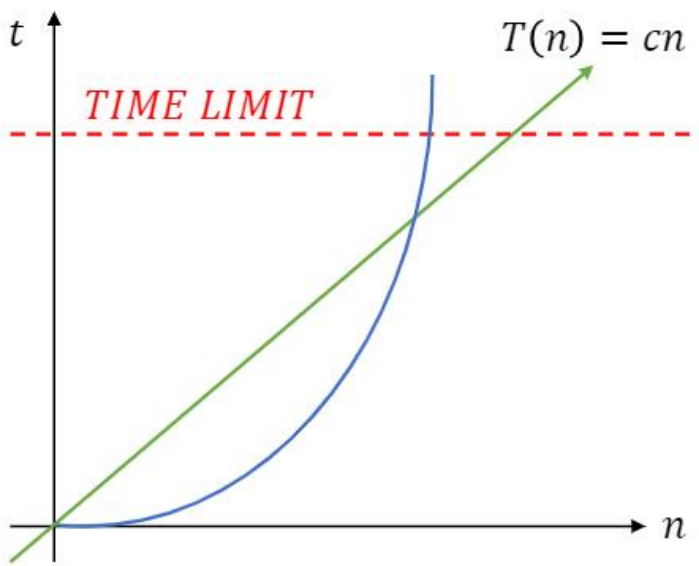
2) 데이터의 수 n이 항상 x_2정도라면 이 알고리즘은 제한시간을 초과하여 동작할 수 있다. 이 알고리즘을 파기하고 새 알고리즘을 설계해야 하는가?

Time Cost - Order of Function

연산량을 표현한 수식이 더 높은 오더(Order)를 가지는 알고리즘이
데이터 크기가 증가할 수록 더 가파르게 연산량이 증가한다.

Q. 더 높은 오더로 연산량이 표현되는 알고리즘은 항상 더 많은 연산을 수행하는가?

```
/**
 * 이 반복문은 N의 크기에 제곱-비례하게 소요 시간이 증가한다.
 */
for(int i = 0 ; i < N; i++)
{
    for(int j = 0 ; j < N; j++)
    {

        SomeCode(); //반복문이 없는 상수시간 연산

    }
}
```



Time Cost - Order of Function

연산량 함수의 오더가 실행 시간의 상승폭을 나타낼 수 는 있지만,

- 물론 가능하다면 낮은 오더의 연산량을 사용하는 것이 유리하다.
- 연산량 표현에 사용되는 기본 요소들과 명칭은 익혀 두자

실제 실행 시간에는 더 많은 요소들이 관여한다.

- 상수 시간 명령어들
- 데이터의 크기
- 입/출력 시간 등

극단적으로는…

- CPU vs GPU분산처리
- 멀티 쓰레딩과 병렬처리
- 하드웨어 스펙 …
- 언어적 성능
- 컴퓨터의 구조적인 이유

물론 대회나 시험에서는 이런 요소들은 통일하거나 큰 차이가 없도록 한다.

Functions	Name
$O(1)$	Constant
$O(\log_k N)$	Logarithm
$O(\sqrt[k]{N})$	k-Root
$O(N)$	Linear
$O(N^k)$	Polynomial
$O(a^N)$ $for\ a > 1$	Exponential
$O(\mathrm{N!})$	Factorial

Time Complexity of Algorithms

문제를 해결하기 위해 걸리는 시간과 입력 데이터의 관계를 함수 꼴로 표현한 것.
연산량(수행 시간)이 데이터의 크기에 대해 얼마나 민감하게 변하는지를 나타내는 지표.

일반적으로 Big-O표기법을 사용하여 입력에 대한 연산량을 나타낸다.

1. 상수는 모두 무시하고
2. 각 변수별로 Order가 가장 높은 항 만을 남겨 표현한다.

예:

1. $O(10^3 n^2 + 10^9 n) = O(n^2)$
2. $O(3N^2 \log_2 N + 2M^2 + N) = O(N^2 \log_2 N + M^2)$

일반적으로 시간 복잡도로 최악의 경우에 대한 연산량을 표현하지만, 최소나 평균 연산량을 표현하는 경우도 있다.

Time Complexity Analysis

데이터가 클 수록 더 작은 시간 복잡도를 가지는 알고리즘을 설계하는 것이 유리하다.

반대로, 데이터가 작을 수록 이런 점에서 자유로워진다.

시간 복잡도와 데이터의 최대 크기를 통해 대략적으로 어느 정도 수준의 연산량이 필요할지 예측해볼 수 있다.

보통의 PC에서 수천만 단위의 연산을 처리하면 1~2초 정도의 시간을 소요하게 된다. 이런 정보를 활용해 자신의 알고리즘을 대략적으로 평가할 수 있다. 물론 어느정도는 감각과 경험이 필요하다.

Q. N이 최대 20만인 데이터가 주어지는 문제에서, 내가 설계한 알고리즘이 $O(n^2)$의 시간 복잡도를 가진다면 잘 설계한 것일까?

Algorithm Optimization

알고리즘은 출력하고자 하는 값과 입력으로 주어지는 값에 따라서 다양하게 변형될 수 있다.

입력 데이터가 가지는 특징을 이용하여 알고리즘을 간소화할 수 있다.

- 정렬 여부
- 중복 존재 여부
- 입력 값의 범위와 크기
- 기타 수학적 특징 혹은 문제에 제시 된 규칙 등

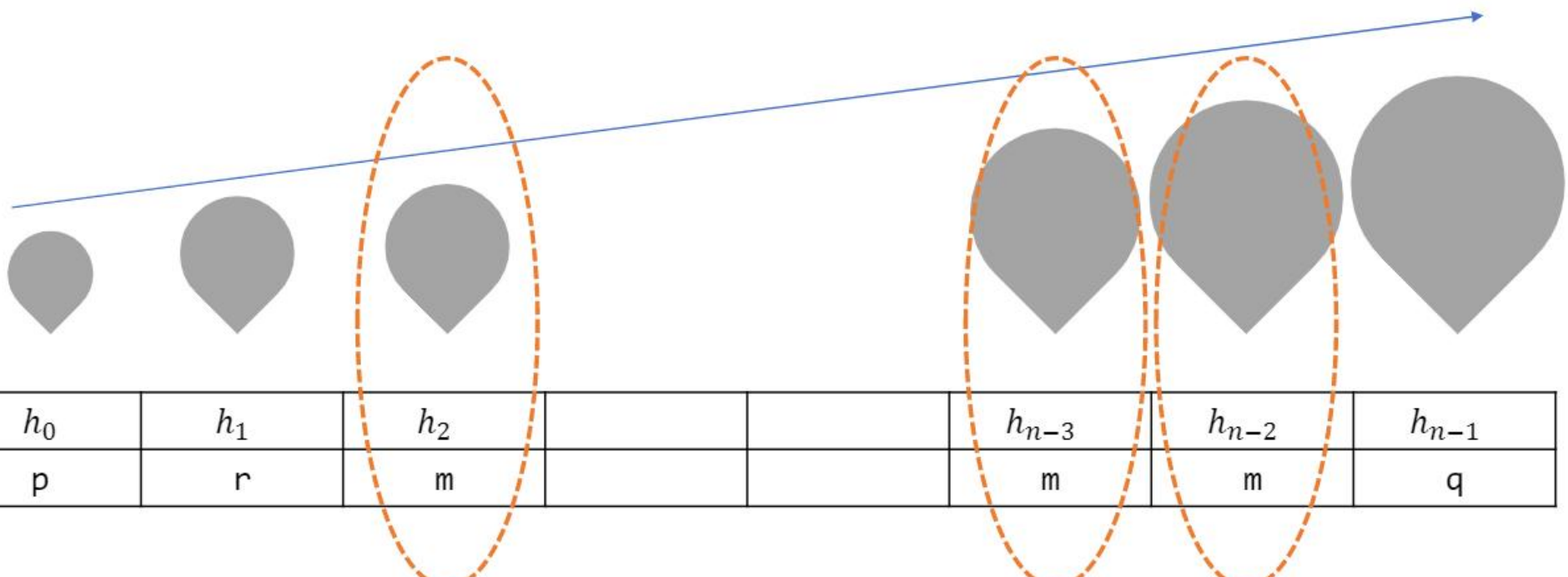
또한, 우리가 원하는 출력 데이터를 구하기위해 필요하지 않은 정보를 생략함으로서 기존의 알고리즘을 개선하거나 다른 알고리즘을 적용할 수 있게 된다.

- 예시
  - 구체적인 개수가 아닌 존재 여부만이 중요할 때
  - 여러가지 정답 후보들 중 하나만이 필요할 때
  - 등…

Problem 02A. 도토리 키 재기

주어진 조건을 만족하는 최대값을 탐색해야 한다.

왼쪽부터 탐색을 시작하는 것과 오른쪽부터 시작하는 것에 차이가 있을까?



$height$	h_0	h_1	h_2			h_{n-3}	h_{n-2}	h_{n-1}
$month$	p	r	m			m	m	q

Problem 02A. 도토리 키재기

```java
/**
 * 생일이 m월인 가장 큰 키의 도토리를 찾는 함수
 * @param height  각 도토리의 키
 * @param month   각 도토리의 출생 월
 * @param n   도토리의 수
 * @param m   찾고자 하는 달
 * @return    month[k] == m인 가장 큰 height[k]
 */
public static int getMaximumHeight(int[] height, int[] month, int n, int m)
{
    int maxHeight = -1; //생일이 m월인 사람이 아무도 없다면 -1일 것이다.
    for(int i = n-1; i>=0 ; i-=1)
    {   //모든 도토리의 출생 월 month[i]에 대해
        //키가 큰 도토리부터 고려하다가
        if(month[i] == m)
        {   // 생일이 일치하는 도토리가 등장했다! 저장하자
            maxHeight = height[i];

            //그 이후 (i가 작아지는 방향)에는 어차피 더 큰 키의 사람이 없다.
            //그러므로 더 수행할 필요가 없음이 자명하다.
            break;
        }
    }
    return maxHeight;
}
```

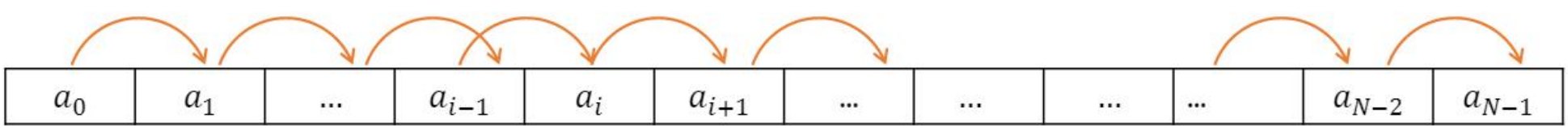
Problem 02B. 오름차순인가?

문제의 주어진 조건을 활용해 조금 더 효율적인 알고리즘을 설계해보자.

배열 `data[0] ~ data[N-1]`이 오름차순이다?

- `data[i] <= data[i+1]`를 만족하는 쌍 `<i, i+1>`이 총 `N-1`개 존재해야 한다.

⇔ `data[i] > data[i+1]`를 만족하는 쌍 `<i, i+1>`이 존재하지 않아야 한다.
⇔ 이런 쌍이 하나라도 존재하면 오름차순이 아니다.



Problem 02B. 오름차순인가?

변수의 역할을 정의함에 따라 로직의 구현이 달라질 수 있음에 유의하자.

```java
/**
 * 주어진 배열이 오름차순인지 검사하는 함수
 * @param data
 * @param n     데이터의 수
 * @return      data[0] ~ data[n-1]이 오름차순이라면 true, else false
 */
public static boolean isOrdered(int[] data, int n)
{
    int count = 0;
    for(int i = 0 ; i + 1 < n ; i ++)
    {



    }

    if(             ){

    }else{

    }
}
```

Problem 02C. 다양성

무작위 배열에서 서로 같은 값들을 찾아 한 번씩만 출력해주어야 한다.
또한 모두 정렬하여 출력해야 한다.

a	*a*	*e*	*c*	*a*	*b*	*a*	*a*	*b*	*d*	*c*	*c*

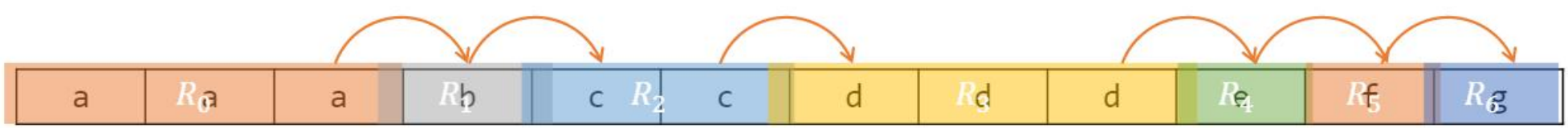
Problem 02C. 다양성

하지만 이 문제의 입력 데이터는 모두 오름차순으로 정렬되어 주어진다.

정렬된 데이터는 다양한 특징을 가지게 되어 처리가 용이 해진다.

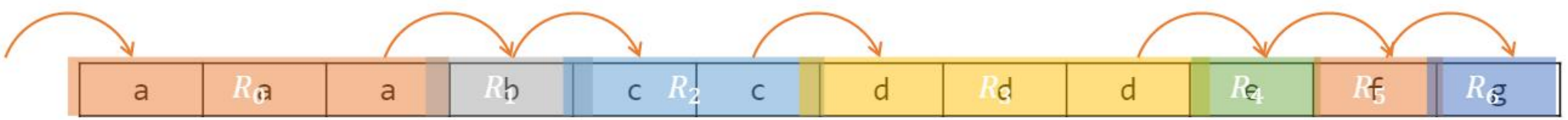
- 이런 점을 살려, 처리하기 번거로운 데이터는 정렬한 이후 처리하기도 한다.

데이터가 값을 기준으로 정렬되어 있다면, 서로 같은 값은 항상 인접해 있음이 보장된다.
즉, 서로 같은 값들끼리 하나의 범위(그룹)을 이룬다.



Problem 02C. 다양성

각 범위에 대한 대표 원소를 하나 정한 후,
각 그룹에 대해 대표 원소의 값을 한 번씩만 고려해주면 된다!



범위나 그룹처럼 하나의 인덱스로 표현하기 힘든 정보의 경우,
해당 정보를 대표할 수 있는 중복되지 않는 '기준'을 정하여 다루는게 좋다. (Primary Key?)

Problem 02C. 다양성

아래 조건식에 들어갈 내용을 고민해보자.

```
public static int getElementTypeCount(int[] data, int n)
{
    int countType = 0;
    for(int i = 0 ; i < n ; i ++)
    {
        if(                          )
        {

            countType += 1;

        }
    }

    return countType;
}
```

Problem 02D. 문자열의 비교

어떤 변수의 '배열'은 한 번에 비교하기 쉽지 않은 구조다.

- 물론 정의하기 나름이지만.

두 배열 내부 각 원소들을 일일이 비교하여 결정하는 경우가 일반적이다.

- 그렇기에 두 배열의 비교는 일반적으로 배열의 길이 만큼의 시간을 소모하는 연산이다.

배열 혹은 문자열을 다루는 내장 기능들과 함수들은 대부분이 길이에 비례한 시간을 소모한다.

- 꼭! 대략적인 시간복잡도는 알아 두는게 좋다.

Problem 02D. 문자열의 비교

두 단어 algorithm과 allergy를 생각해보자. 일반적으로 사전을 펼쳤을 때 어떤 단어가 더 앞에 있을까?

우리가 사전에서 단어를 찾아 나갈 때,

- 현재 페이지보다 앞에 있을지
- 현재 페이지보다 뒤에 있을지

판단하는 기준이 무엇일지 생각해보고 명확하게 그 과정을 정의해보자.

a	l	g	o	r	i	t	h	m

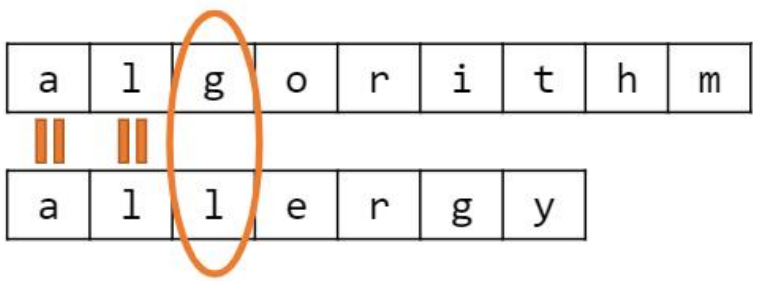
a	l	l	e	r	g	y

Problem 02D. 문자열의 비교

일반적으로 두 문자열을 앞에서부터 조회했을 때,

“서로 다른 문자가 등장하는 첫 위치에서의 두 문자의 사전 순 관계”를 두 문자열의 사전 순 관계로 정의한다.

- ‘A’ < ‘B’ < … < ‘Z’
- ‘0’ < ‘1’ < … < ‘9’
- ‘ㄱ’ < … < ‘ㅎ’
- ‘가’ < … < ‘힣’



Problem 02D. 문자열의 비교

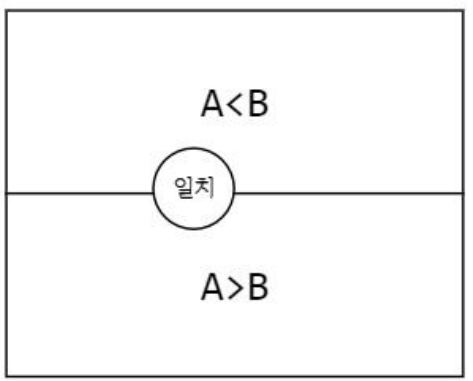
```
class MyString implements Comparable<MyString>
{
    private char[] characters;
    /**
     * this와 파라미터 o를 사전순으로 비교하여 결과를 -1, 0, 1로 반환하는 함수
     */
    @Override
    public int compareTo(MyString o) {
        int n = Math.min(this.characters.length, o.characters.length);









    }
}
```

Problem 02D. 문자열의 비교

완전 임의로 선택된 두 변수 A, B에 대해서 두 값이 일치할 확률은 상당히 적다.

- 물론 어떤 데이터인지에 따라 변수의 분포나 확률은 달라질 수 있다.
- 이 사실을 이용해 많은 것들을 할 수 있다.

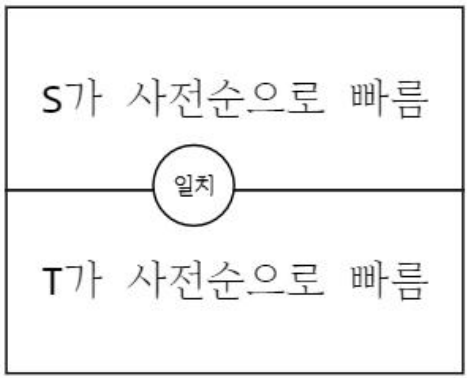
$$P(A = x|B = x) = \frac{1}{\text{해당 변수 정의역의 크기}}$$



Problem 02D. 문자열의 비교

그렇다면 두 배열 S, T의 모든 원소가 일치할 확률은 얼마나 적을까?

$$P(S = x|T = x) = \frac{1}{\text{타입 정의역의 크기}^{\text{배열의 길이}}}$$



우리가 현재 알고 싶은 정보가 '두 문자열(혹은 배열)이 같은가'일 때 사전순의 대소관계는 꼭 필요한 정보인가?

Problem 02D. 문자열의 비교

아래의 경우 두 배열은 일치하지 않다고 할 수 있다.

1. 두 배열의 길이가 다른 경우
2. 두 배열의 원소가 하나라도 다른 경우

```
public boolean equals(Object obj)
{
    if(!(obj instanceof MyString))
        return false;
    MyString o = (MyString)obj;









}
```

Problem 02E. 소수의 판별

입력으로 주어지는 숫자가 소수인지 판별해야 하므로, 소수가 가지는 특징을 이용한다.

- 1은 소수가 아니다.
- 소수는 1과 자기 자신만을 약수로 가진다.

즉, [1, N]범위에서 N의 약수의 수를 세어 2개가 된다면 소수다. 그렇지 않다면 소수가 아니다.

⇔ 1과 자기 자신을 제외하면 약수의 수가 0개여야만 한다!

짝수들은 2를 제외하고 모두 합성수라는 사실을 이용해 개선 할 수 있다.

Problem 02E. 소수의 판별

1. 연산량이 최대가 되기 위한 N의 조건은 무엇인가?

2. 이 함수의 시간 복잡도를 계산하시오.

3. 문제의 입력 조건 하에서 제한 시간내에 수행 될 수 있겠는가?

```
public static boolean isPrime(int N)
{
    if( N == 1) return false;   //1은 소수가 아니다
    else if( N == 2 ) return true; //2는 소수다
    else if( N % 2 == 0) return false; //나머지 짝수는 소수가 아니다

    int divisorConunt = 0;  //1과 자기 자신을 제외한 N의 약수의 수
    for(int i = 3; i < N; i+=2)
    {   //모든 N미만의 홀수 자연수 i에 대해
        if( N % i == 0 )
        {   //N의 약수가 되는 i에 대해 카운트
            divisorConunt += 1;

            //하나라도 발견된 순간 더 찾을 필요는 없다.
            break;
        }
    }

    if(divisorConunt > 0)
    {
        return false;
    }else{
        return true;
    }
}
```

Problem 02E. 소수의 판별

자연수 N의 한 약수 a를 생각해보자. N을 a로 나눈 몫을 b라고 하면 아래의 식이 성립한다.

$$N = ab \ (\text{단}, \ a \leq b)$$

즉, 한 자연수 N에 대해 서로 곱하여 N이 되는 약수 쌍 a, b가 **항상** 존재한다. 편의상 a는 b이하인 경우만 고려하자.

$(a \leq b)$라는 조건 하에서 N에 대한 a의 최대 값은 $a \leq \sqrt{N}$ 이다.

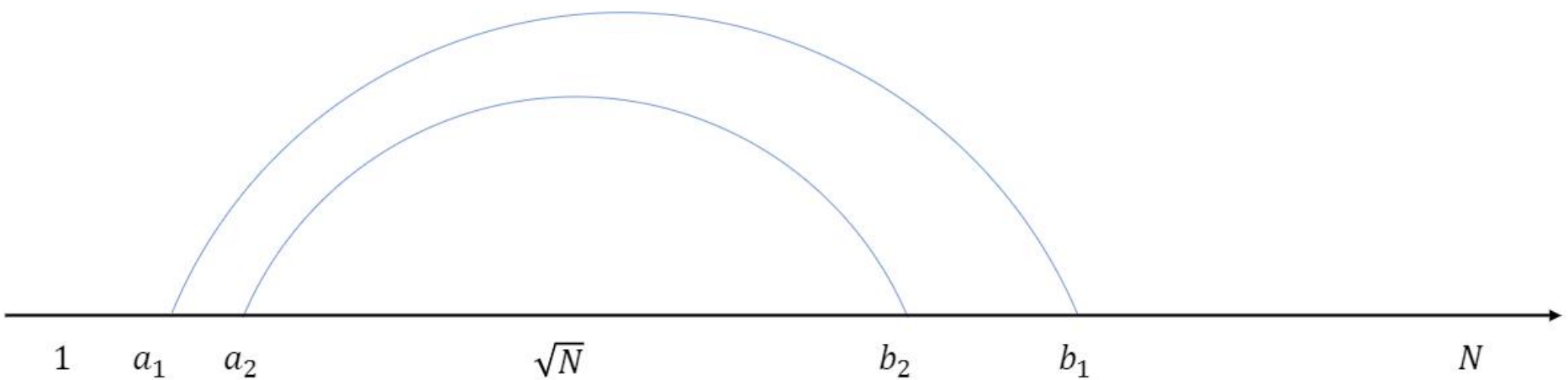
- 그러므로 $1 \leq a \leq \sqrt{N}$, $\sqrt{N} \leq b \leq N$ 이 된다.

Problem 02E. 소수의 판별

그러므로, N이 소수가 아니라면 $2 \le a \le \sqrt{N}$ 범위에 약수 a가 반드시 존재한다.

대우명제> $2 \le a \le \sqrt{N}$ 범위에 약수가 없다면 N은 소수다.

∴ [2, $\sqrt{N}$] 범위에 대해서만 약수의 존재 여부를 검사하면 소수/합성수를 구분할 수 있다.



Problem 02E. 소수의 판별

1) 이 알고리즘의 시간 복잡도를 계산해보자.

2) 아래 코드에서 for문의 종료 조건은 모순이 없는가?

3) 더 빠른 방법이 있을까?

```
public static boolean isPrime(int N)
{
    if( N == 1) return false;   //1은 소수가 아니다
    else if( N == 2 ) return true; //2는 소수다
    else if( N % 2 == 0) return false; //나머지 짝수는 소수가 아니다

    for(int i = 3; i*i <= N; i+=2)
    {   //모든 N미만의 홀수 자연수 i에 대해
        if( N % i == 0 )
        {   //이 범위에 약수가 존재한다면 소수일 수 없다.
            return false;
        }
    }
    //약수가 하나도 존재하지 않았다면.
    return true;
}
```

Problem 02F. 데스티니

두 데이터 i, j의 쌍(혹은 1:1관계) <i, j> 또한 데이터의 집합으로 볼 수 있다.

- 그러므로 유한하게 조회하거나 나열할 수 있다.

i가 [0, N-1]범위의 정수이고 j가 [0, M-1] 범위의 정수라고 한다면

- <0, 0>, <0, 1> , … , <0, M-1>
- <1, 0>, <1, 1> , … , <1, M-1>
- …
- <N-1, 0>, <N-1, 1>, … <N-1, M-1>

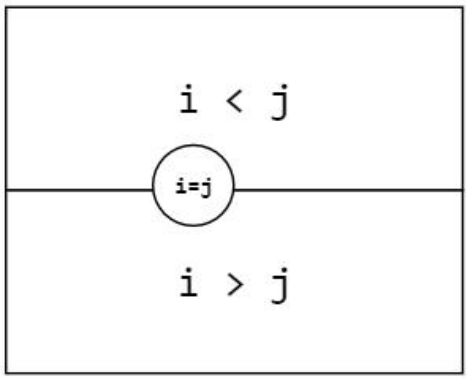
즉, 모든 경우를 반복문으로 조회해볼 수 있다.

```
for(int i = 0 ; i < n ; i ++)
{
    for(int j = 0 ; j < m ; j++)
    {
        System.out.printf("<%d, %d>\n", i, j);
    }
}
```

Problem 02F. 데스티니

또한 집합이기에 분류할 수 있다.

- `i < j` 인 경우
- `i = j` 인 경우
- `i > j` 인 경우



특정 상황에서는 두 데이터에 대한 관계 <i, j>에 대하여

- 어떤 관계인가
- 각 관계에 대해 무슨 작업을 할 것인가

에 따라서 조회가 불필요하거나, 중복적인 관계가 발생할 수 있다.

이런 경우는 제외하고 탐색하는 것이 당연히 효율적이다.

Problem 02F. 데스티니

가장 가까운 거리와, 그 거리를 가지는 쌍의 수를 한 번에 조회해보자.

```
int min_sqd = Integer.MAX_VALUE; //가장 거리가 가까운 두 점의 거리의 제곱
int min_cnt = 0;

for(int i = 0 ; i < n ; i ++)
{
    for(int j = 0 ; j < i ; j++)
    {









    }
}
```

Problem 02G. 버블 소트 구현하기

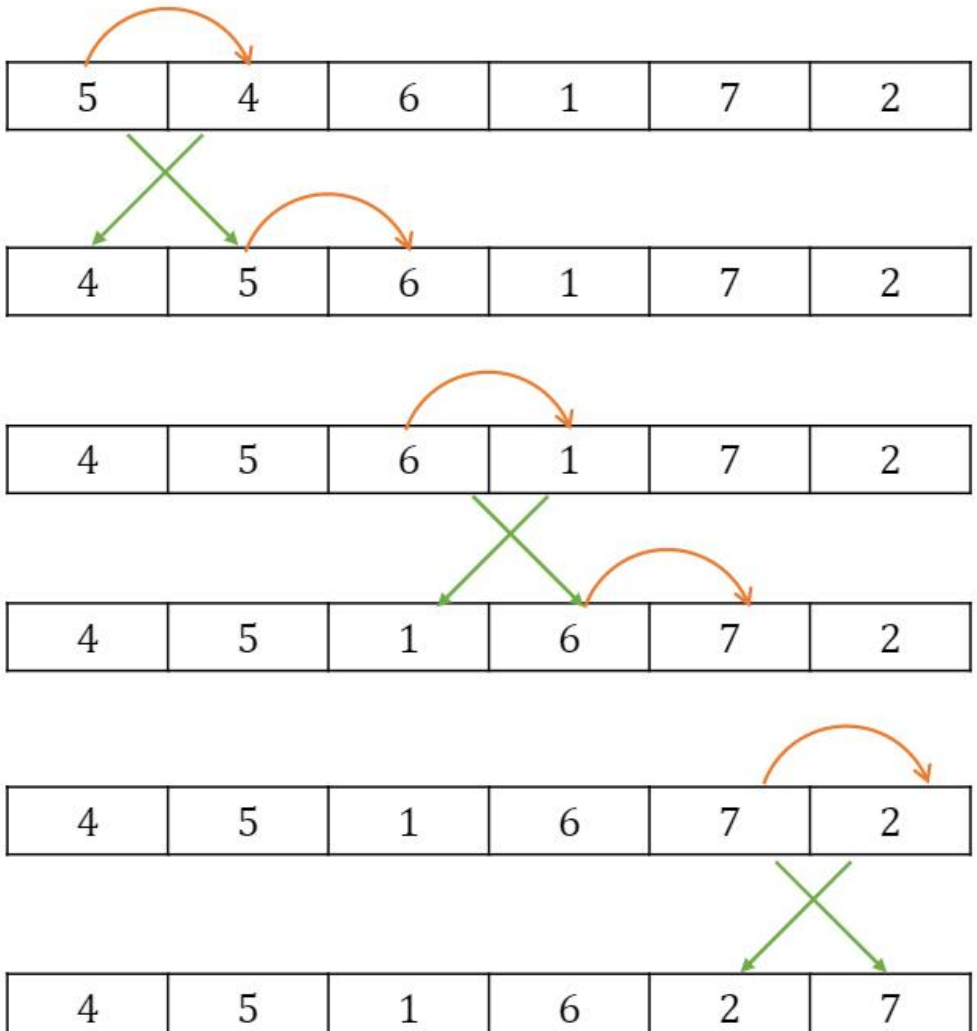
Algorithm>

앞에서부터 차례로 인접한 두 숫자를 비교하며

- a[j] <= a[j+1]면 continue;
- a[j] > a[j+1]면 두 숫자를 바꾼다.

가장 큰 숫자가 가장 뒤로 이동하게 된다.

이 과정을 N번 반복한다.



Problem 02G. 버블 소트 구현하기

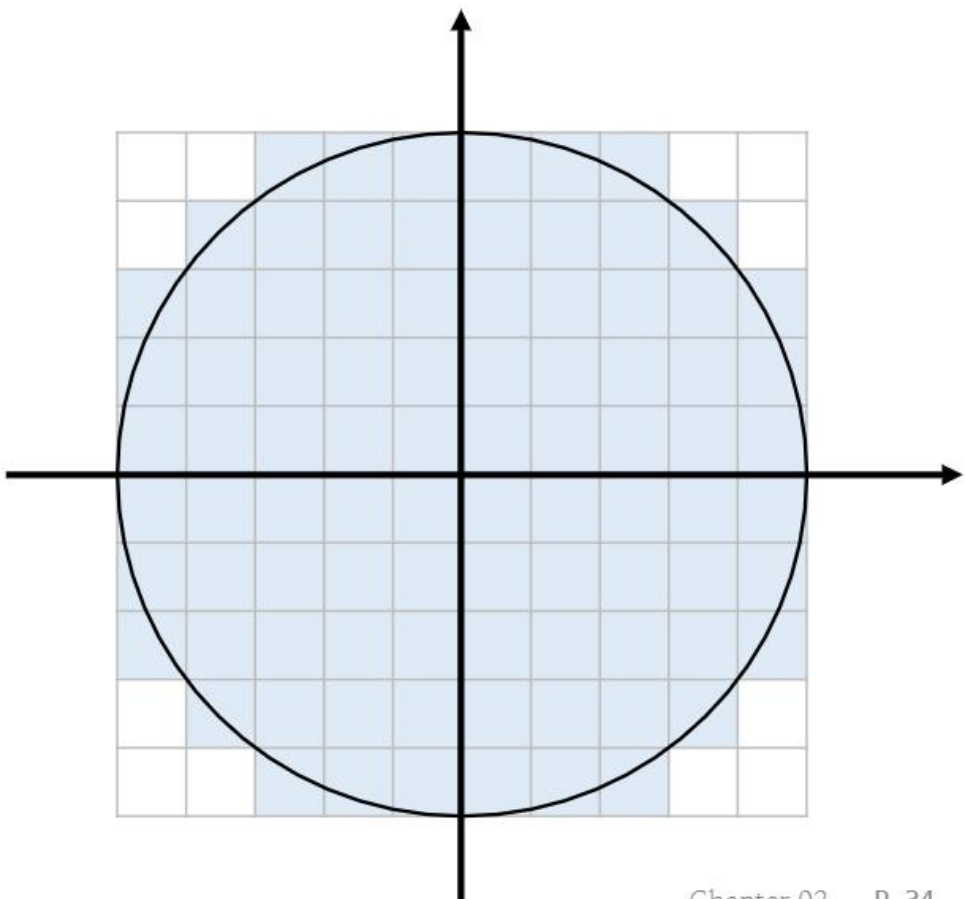
```
public static void bubbleSort(int[] data, int n)
{
    for(int i = 0 ; i < n ; i++)
    {

        for(int j = 0 ; j <          ; j ++)
        {

            if( data[j] > data[j+1] )
            {









            }
        }









    }
}
```

Problem 02H. 픽셀 수 세기

2차원 상에서 원의 위치는 정답에 영향을 주지 않으므로,
다루기 편할 것 같은 위치에 있다고 가정하자.

원의 중심이 원점에 있다고 가정하니,
각 사분면의 픽셀 수는 모두 같다.

∴ 생략 가능하다.



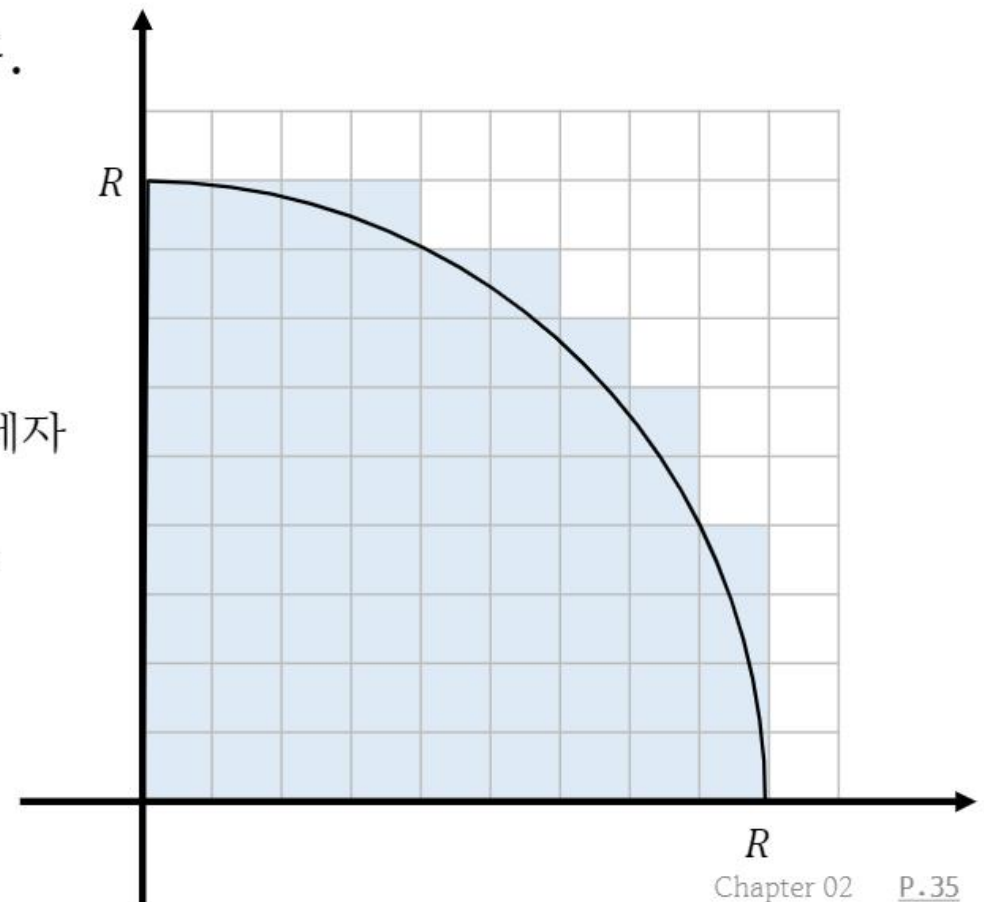
Problem 02H. 픽셀 수 세기

2차원 평면 상에서 격자의 수는 무한하게 많다.

하지만 정답이 될 수 있는 격자의 범위는 유한하다.

- x좌표나 y좌표가 R을 초과하는 점은 원 안에 포함될 리 없다.
- 정답의 후보에서 배제하자.

∴ R^2개의 격자들 중 원에 포함되는 픽셀의 수를 세자
  - 각 격자를 구분할 수 있는 기준을 정해야 한다.
  - 격자의 네 점 중 '왼쪽 아래 점의 좌표'를 사용해보자.



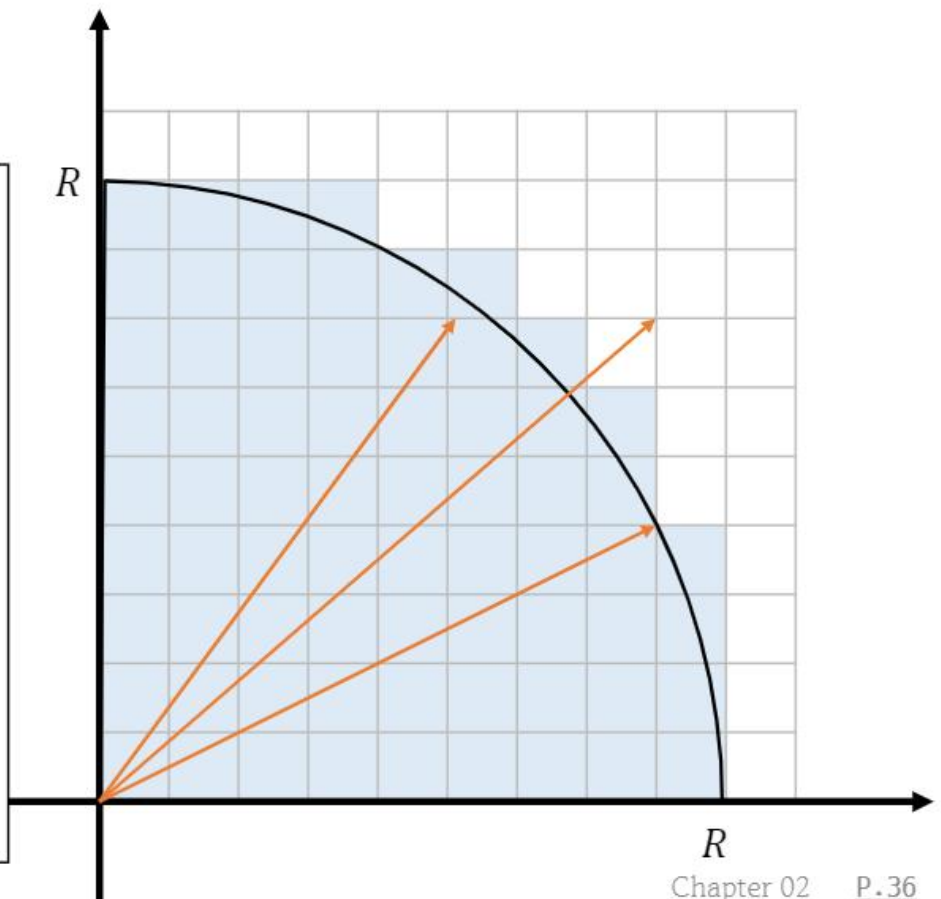
Problem 02H. 픽셀 수 세기

네 점 모두 원 안에 있거나, 두 개 이상의 변과 교차한다.

⇔ 왼쪽 아래 점(x,y) 이 원 안에 있다.

⇔ $\sqrt{(x-0)^2+(y-0)^2} < R$

```java
/**
 * 왼쪽 아래 좌표가 (x,y)인 픽셀이 반지름 R인 원에 포함되는가?
 * @param x
 * @param y
 * @param R
 * @return  포함된다면 true, else false
 */
public static boolean isInside(long x, long y, long R)
{
    long sqd = x *x + y * y ; //거리의 제곱
    if ( sqd < R * R )  //반지름의 제곱
    {   //원점과의 거리가 반지름보다 작다면, 원 안에 있다.
        return true;
    }
    return false;
}
```



Problem 02H. 픽셀 수 세기

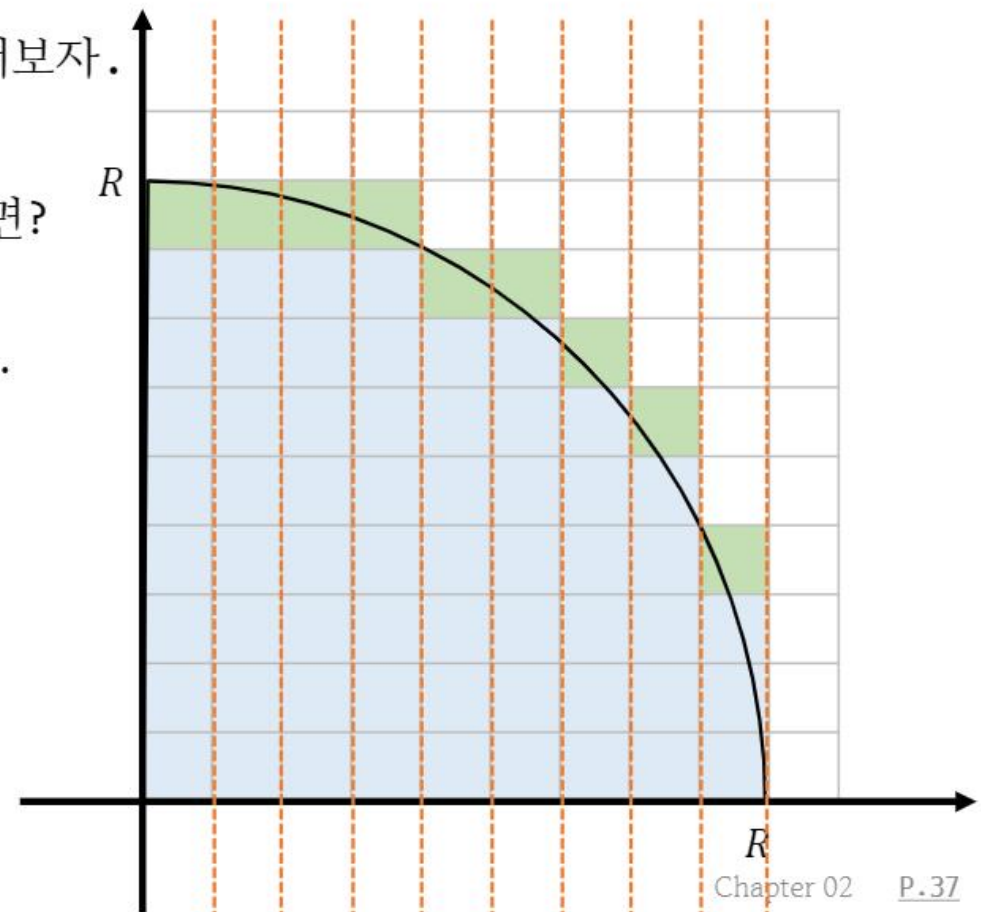
But, $R^2 = 4 \times 10^{10}$ 이다. 모든 격자를 일일이 순회하는 건 어렵다.

x좌표가 같은 격자끼리 하나의 그룹(열)으로 묶어보자.

각 그룹별로 가장 위의 격자의 위치를 알 수 있다면?
그 아래의 격자 수는 자명 해진다.

- 격자는 1x1크기로 정수 좌표계에 하나씩 존재하기 때문.

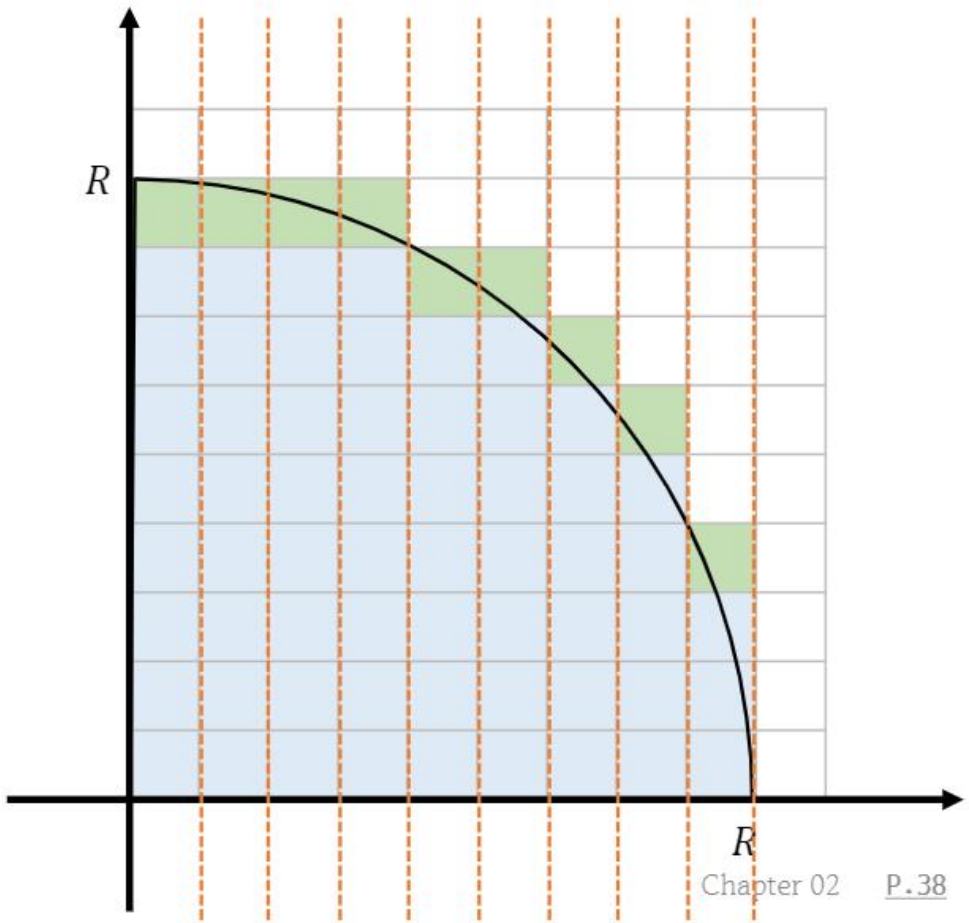
But, 가능한 y좌표가 최대 R개 이므로
각 열 별로 모든 y좌표를 탐색한다면 결국 $O(R^2)$



Problem 02H. 픽셀 수 세기

But, $R^2 = 4 \times 10^{10}$ 이다. 모든 격자를 일일이 순회하는 건 어렵다.

```
public static void testCase(int caseIndex) {
    long R = scanner.nextLong();
    long sum = 0; //1사분면에 존재하는 총 픽셀의 수
    for(long x = 0 ; x <= R; x ++)
    {
        long height = 0;
        for(long y = R; y >= 0; y --)
        {
            if(isInside(x, y, R))
            {   //위에서 부터 내려오다가
                //가장 최초로 원 안에 포함된 픽셀 (x, y)
                //이 그룹의 높이는 (y+1)이 된다.
                height = (y+1);
                break;
            }
        }
        sum += height; //너비는 1이므로
    }
    System.out.printf("#%d\n", caseIndex);
    System.out.printf("%d\n", sum * 4); //모든 사분면의 픽셀 수
}
```



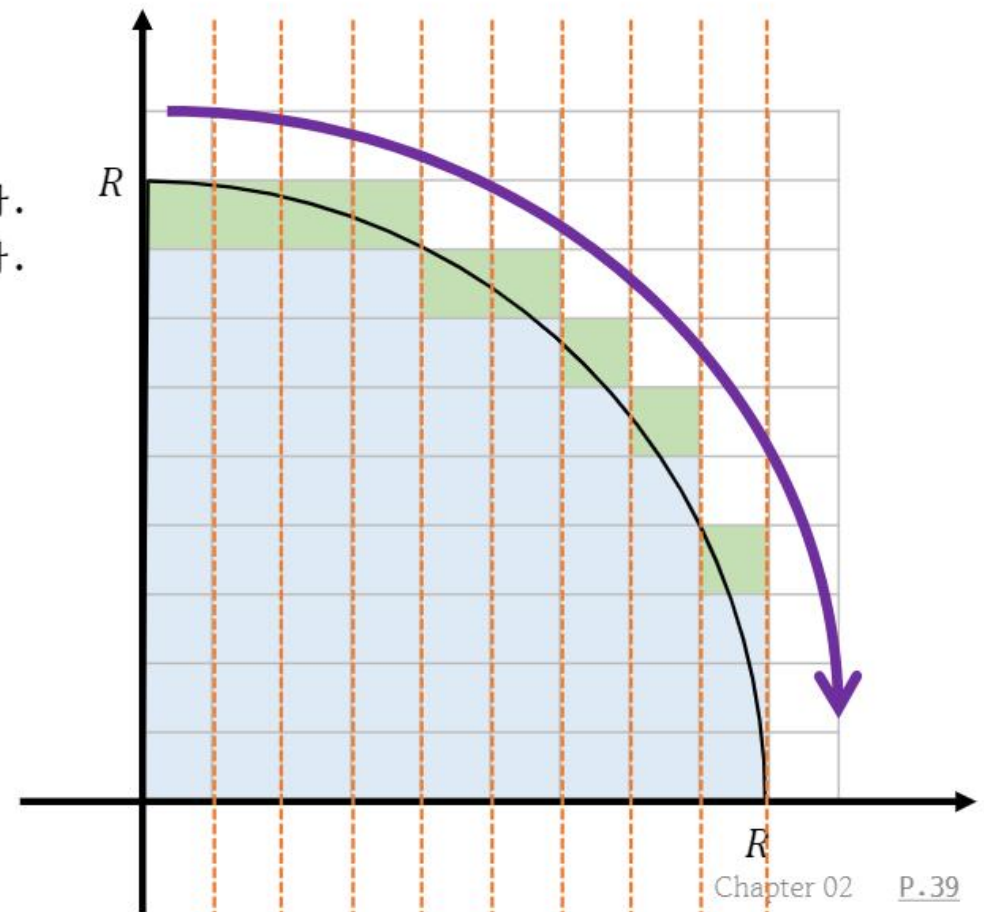
Problem 02H. 픽셀 수 세기

원점을 중심으로 가지는 원은 1사분면에서

- x좌표가 증가할 수 록 y좌표가 감소한다.

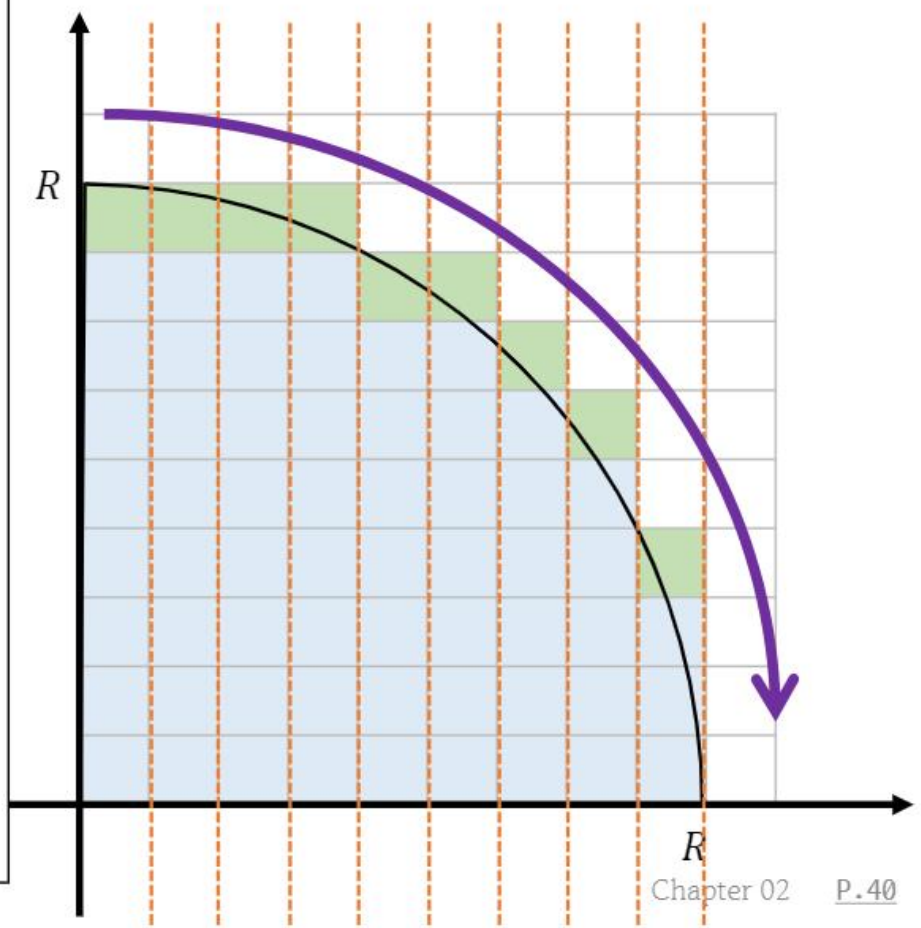
즉, 각 그룹의 높이는 내림차순 배열과 같다.

- 한 그룹의 높이는 자기 자신 왼쪽의 그룹 높이보다 작다.
- R부터가 아닌, 왼쪽 그룹의 높이부터 검사해도 충분하다.



Problem 02H. 픽셀 수 세기

```
public static void testCase(int caseIndex) {
    long R = scanner.nextLong();

    long sum = 0; //1사분면에 존재하는 총 픽셀의 수
    long y = R;
    for(long x = 0 ; x <= R; x ++)
    {
        long height = 0;
        for(          ; y >= 0; y --)
        {
            if(isInside(x, y, R))
            {   //위에서 부터 내려오다가
                //가장 최초로 원 안에 포함된 픽셀 (x, y)
                //이 그룹의 높이는 (y+1)이 된다.
                height = (y+1);
                break;
            }
        }

        sum += height; //너비는 1이므로
    }

    System.out.printf("#%d\n", caseIndex);
    System.out.printf("%d\n", sum * 4); //모든 사분면의 픽셀 수
```



Problem 02H. 픽셀 수 세기

상수 시간이 소요되는 F를 Basic Operation으로 정의했을 때,

1) F는 최대 몇 번 수행될 수 있는가?

2) 반지름 R에 대하여, 이 알고리즘의 시간 복잡도를 계산 하시오.

```
public static void testCase(int caseIndex) {
    long R = scanner.nextLong();

    long sum = 0; //1사분면에 존재하는 총 픽셀의 수
    long y = R;
    for(long x = 0 ; x <= R; x ++)
    {
        long height = 0;
        for(         ; y >= 0; y --)
        {
            if(isInside(x, y, R))
            {   //위에서 부터 내려오다가
                //가장 최초로 원 안에 포함된 픽셀 (x, y)
                //이 그룹의 높이 = (y+1)이 된다.
                height = (y+1);
                break;
            }
        }

        sum += height; //너비는 1이므로
    }

    System.out.printf("#%d\n", caseIndex);
    System.out.printf("%d\n", sum * 4);
```

F

Problem 02I. 정주행

주어진 배열을 정렬했을 때, 1씩 증가하는 연속 수열이 되는가?

물론 직접 (빠른) 정렬을 한 후, 차례로 1씩 증가하는지 검사해볼 수 도 있다.

- 과제로 남깁니다. 한 번 해보세요.

하지만 굳이 정렬을 해야 하기에 최소 $O(N \log_2 N)$의 시간이 소요된다.

또한, 정렬을 하면 안되거나 하기 싫을 때에는 사용할 수 없다.

다른 방법은 무엇이 있을까?

Problem 02I. 정주행

A를 입력 받은 배열, B를 어떤 정렬 된 연속 수열이라고 하자.

두 배열이 집합적으로 같다면 (모든 원소가 일치한다면),
A를 정렬하면 B가 된다.
-> A를 정렬하면 연속 수열이 된다.

즉, B와 같은 수열이 존재함을 보이면 된다.

Problem 02I. 정주행

주어진 배열의 최소값을 L, 최대값을 G라고 하자.
그렇다면, 배열에 L보다 작거나 G보다 큰 값은 존재하지 않는다.

배열에 [L, G]범위의 모든 정수가 하나씩 존재한다면, 정렬했을 때 연속 수열이 된다.

정수 [L, G] 범위에는 정확히 $M = (G - L + 1)$가지의 정수가 존재한다.

경우의 수를 나누어 보자.

Problem 02I. 정주행

1. M < N 인 경우
    1. 비둘기집 원리에 의해, 중복이 존재할 수 밖에 없다.
    2. 그러므로 연속 수열이 될 수 없다.

2. M > N 인 경우
    1. [L, G]범위의 연속 수열이 되기 위해서는 M개의 원소가 필요하다.
    2. 하지만 배열의 크기가 그 보다 작으므로 불가능하다.

3. M = N 인 경우
    1. [L, G]범위의 숫자가 정확히 하나씩 존재할 수 도 있다.
    2. 중복이 존재할 수 도 있다.

Problem 02I. 정주행

문제의 지문에 다음과 같은 조건이 있었다.

"철승이는 똑같은 에피소드를 두 번 보지 않는다."

⇔ 배열에 중복은 존재하지 않는다.

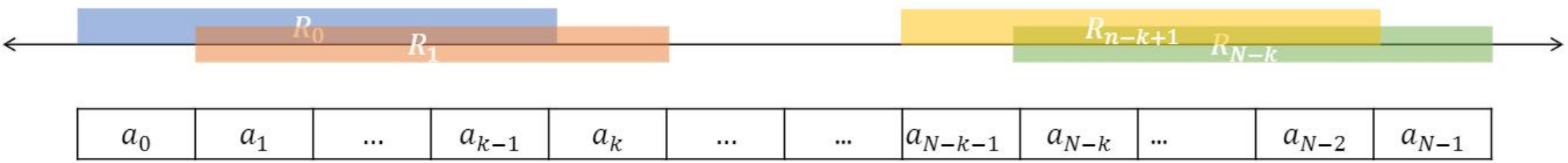
3. M = N 인 경우
    1. [L, G]빔위의 숫자가 정확히 하나씩 존재할 수 도 있다.
    2. ~~*중복이 존재할 수 도 있다.*~~

그러므로, (G-L+1)=N인 경우에는 주어진 배열을 정렬했을 때 항상 연속 수열이 된다.

Problem 02J. 승부조작

Sliding Window Method.

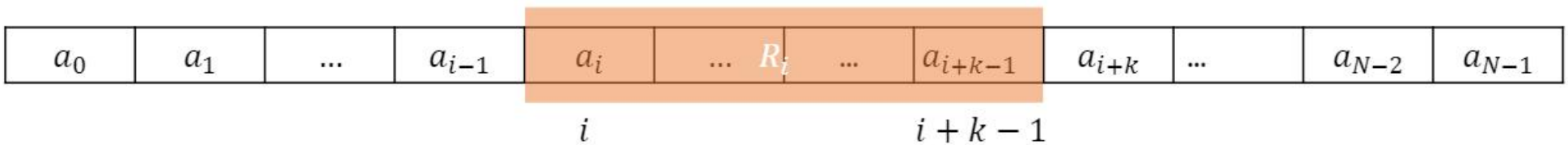
크기가 일정한 범위들을 한 방향으로 순서대로 조회하는 방법.



Problem 02J. 승부조작

각 범위 R_i는 배열의 [i, i+k-1] 인덱스를 포함한다.

모든 범위 R_i들 중 하나 이상의 범위에 대하여, 범위에 속한 원소들의 총 합이 짝수가 되는 경우가 하나 이상 존재하는지 검사하면 된다.



Problem 02J. 승부조작

아래와 같은 경우 시간 복잡도는 어떻게 될까?

```
int winCount = 0;
for(int i = 0 ; i + k - 1 < N; i++ )
{   // Ri := data[i] ~ data[i+k-1] 범위에 대하여
    long sum = 0;
    for(int j = i; j <= i + k -1; j ++)
    {   //범위 Ri의 모든 원소 data[j]의 합을 구한다.
        sum += data[j];
    }

    if( sum % 2 == 0)
    {
        winCount += 1;
        break;
    }
}
if(winCount > 0)
{
    //승리할 수 있다
}else{
    //승리할 수 없다.
}
```

Problem 02J. 승부조작

일일이 모든 범위를 매번 조회하는 것은 대략 $O(kN - k^2) \leq O(kN)$정도의 시간이 소요된다.

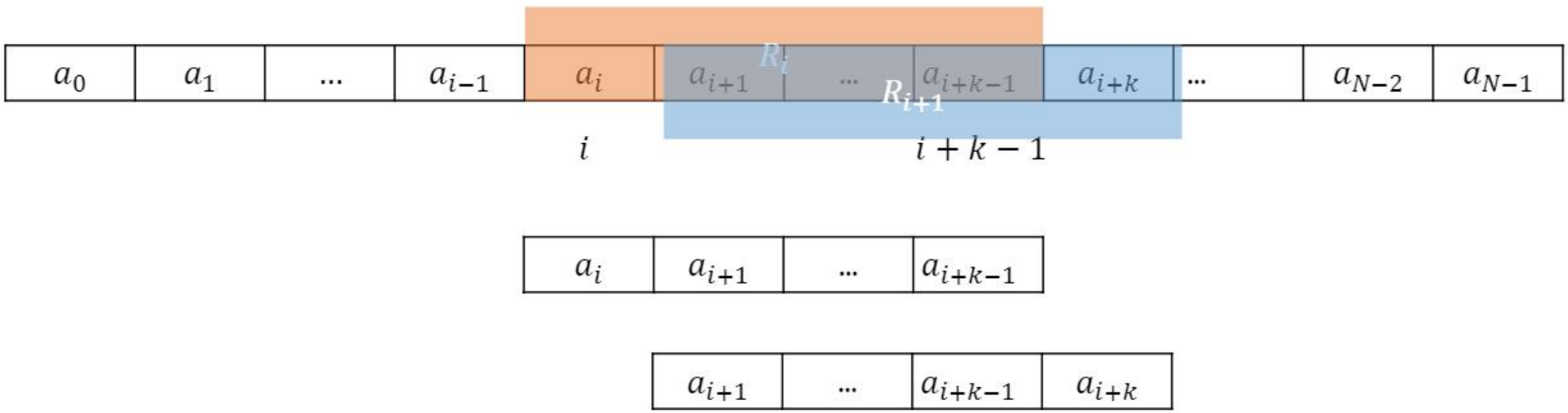
N과 k가 10만 이하의 자연수이므로, 최악의 경우 시간 제한 초과를 예상할 수 있다.

하지만 모든 범위에 대해 합을 검사 해야함은 자명하다. 어떻게 모든 범위를 빠르게 조회할 수 있을까?

Problem 02J. 승부조작

항상 길이가 k로 같은 범위를 순차적으로 조회하고 있다.

두 범위 R_i와 R_{i+1}의 대부분의 범위가 중복된다. 중복되는 정보들을 매번 다시 계산하지 않고 재활용할 수 있을까?



$(R_{i+1}$ 영역의 합) = (R_i영역의 합) - a_i + a_{i+k}

Problem 02J. 승부조작

1) 이 알고리즘의 시간 복잡도를 계산 하시오.

2) 범위들을 조회하는 문제들 중, 이 알고리즘이 적용될 수 있는 조건은 무엇인가?

```
int winCount = 0;
long sum = 0;

// 첫 (k-1)개의 원소에 대한 합을 계산한다.
for(int i = 0 ; i < k - 1 ; i++)
{
    sum += data[i];
}

for(int i = 0; i + k - 1 < N ; i++)
{   //영역의 왼쪽 끝 인덱스 i에 대해

    if(i > 0 )
    {   //영역을 벗어나게 되는 원소 제외
        sum -= data[i-1];
    }

    //새로 영역에 들어온 원소 추가
    sum = sum + data[i + k-1];

    if(sum % 2 == 0)
    {
        winCount += 1;
        break;
    }
}
```

Fin.

감사합니다.

* 이 PPT는 네이버에서 제공한 나눔글꼴과 아모레퍼시픽의 아리따부리 폰트를 사용하고 있습니다.