

Chapter. 03

알고리즘

모의 코딩테스트 풀이 Mock Coding Test

FAST CAMPUS
ONLINE

알고리즘 공채 대비반 I

강사. 류호석

Chapter. 03



모의 코딩테스트 풀이 - 1

Mock Coding Test Solution - 1

I 주의사항

실제 시험이라고 생각하고 5시간을 고정시켜 놓은 상태로

<https://www.acmicpc.net/category/detail/2338>

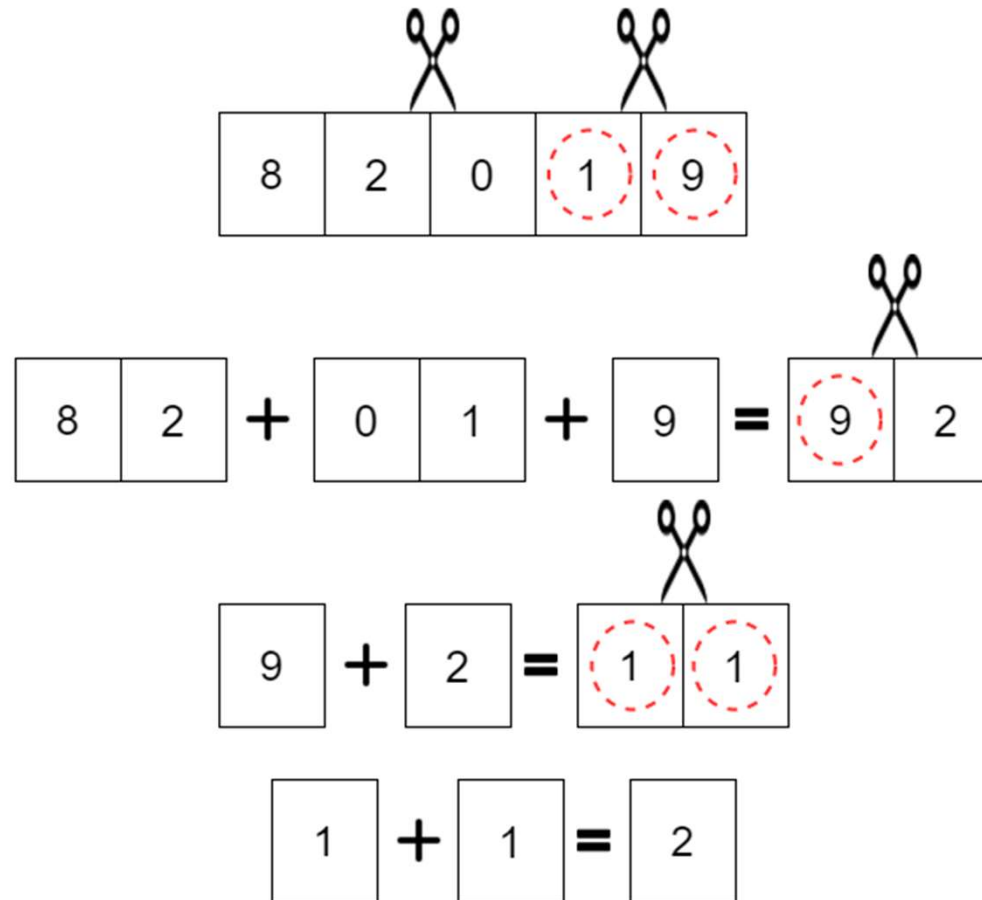
위 대회를 직접 경험해보신 다음에 이후 풀이 영상을 보세요.

어떤 부분이 어려운 것인지, 생각의 방향이 어떻게 다른 지를 직접 느껴보셔야 합니다.

I BOJ 20164 – 홀수 홀릭 호석

난이도: 2

$1 \leq$ 가지고 있는 수 $N < 10^9$



I 출제 의도

완전 탐색 접근을 통해서 모든 경우를 직접 하나하나 찾아내 보자.

본 문제에서 “경우”란, 조건을 만족하게 매 순간 수를 잘라서 시뮬레이션 하는 것을 의미한다.

수가 길어가 9자리 이기 때문에 전부 다 해보아도 경우의 수가 많지 않음을 알 수 있다.

I 완전 탐색 접근

관련 강의 - 01. 어떻게든 풀다. 완전 탐색 (Brute Force)

완전 탐색은 재귀 함수가 제 맛이다.

함수 디자인을 먼저 해보자.

I 완전 탐색 접근

완전 탐색은 **재귀 함수**가 제 맛이다.

함수 디자인을 먼저 해보자.

1. 매 순간 들고 있는 수, x
2. 시작부터 지금까지 얻은 점수, `total_odd_cnt`

```
static void dfs(int x, int total_odd_cnt) {
```

I 완전 탐색 접근

재귀 함수의 조건

1. 종료 조건 ➡ x 가 한 자리 수인 경우
2. 아닌 경우 ➡ 가능한 경우로 모두 잘라서 다음 수에 대해 재귀 호출

I 완전 탐색 접근

별개로, 있으면 좋을 것 같은

수에 포함된 홀수의 개수를 계산해주는 함수

```
static int get_odd_cnt(int x)
```

I 시간, 공간 복잡도 계산하기

수가 3개로 나뉘지고 합쳐질 때마다 크기가 매우 작아지기 때문에 모든 경우의 수가 1억을 절대 넘지 않음을 알 수 있다.

I 구현

```
// x 라는 수 안에 홀수의 개수를 return하는 함수
```

```
static int get_odd_cnt(int x){  
    int res = 0;  
    return res;  
}
```

```
// x 라는 수에 도달했으며, 이때까지 등장한 홀수 자릿수가 total_odd_cnt 만큼 있을 때, 남은 경우를 모두 잘라보는 함수
```

```
static void dfs(int x, int total_odd_cnt) {
```

```
    // 1. 만약 한 자리 수면 더 이상 작업을 반복할 수 없으므로 정답을 갱신하고 종료한다.
```

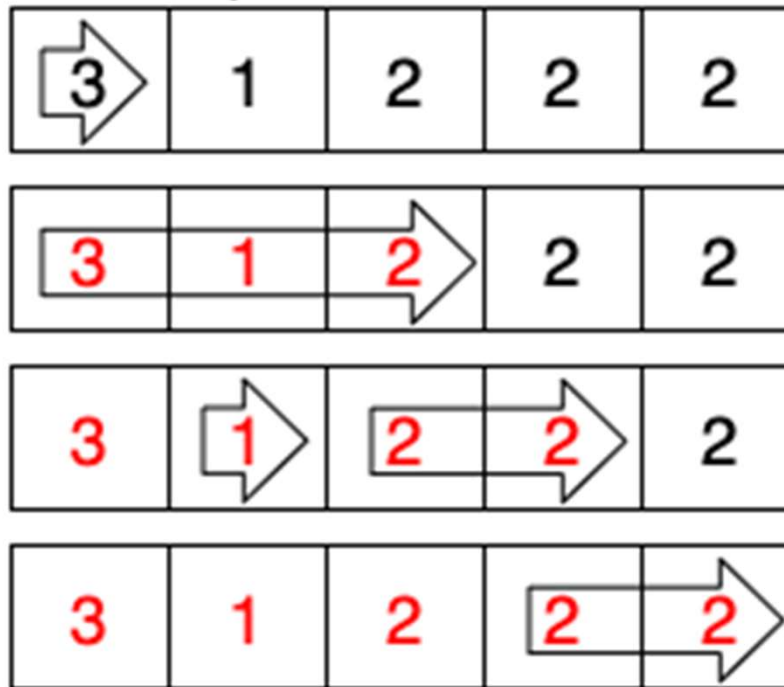
```
    // 2. 만약 두 자리 수면 2개로 나눠서 재귀 호출한다.
```

```
    // 3. 만약 세 자리 이상의 수면 가능한 3가지 자르는 방법을 모두 진행한다.
```

```
}
```

I BOJ 20165 – 인내의 도미노 장인 호석

난이도: 3

 $1 \leq \text{행 } N, \text{ 열 } M \leq 100$ $1 \leq \text{라운드 } R \leq 10,000$ $1 \leq \text{도미노의 길이} \leq 5$ 

I 출제 의도

문제를 해결하는 행위 자체가 “모든 순간을 올바르게 재현하기”

즉, **시뮬레이션**이다.

문제에서 주어지는 상황을 *올바르게* 이해하고, *구현해내는* 연습이 필요하다.

I 시뮬레이션의 핵심

상태를 표현하는 방법 정하기

1. 게임판의 상태를 표현하기

$$A[i][j] = \begin{cases} 0 \Rightarrow Fall\ Down \\ h > 0 \Rightarrow Not \end{cases}$$

$$backup[i][j] = h$$

I 시뮬레이션의 핵심

문제에 필요한 행위를 함수로 표현하기

필요한 행위

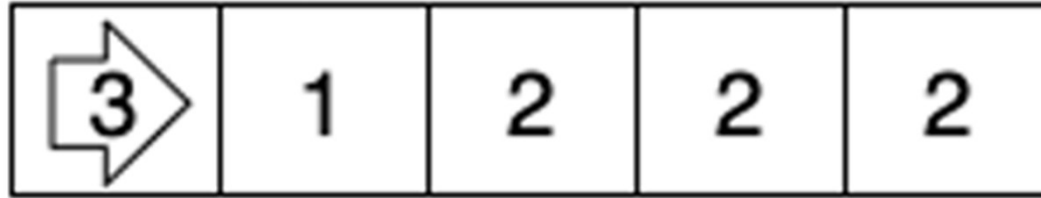
1. 공격, Attack
2. 수비, Defense

I 시뮬레이션의 핵심

1. 공격, Attack

- 넘어뜨린 도미노의 위치 (x, y)
- **특정 방향, dir** 으로 도미노를 밀기
- 밀린 도미노 높이를 이용해 연쇄적으로 밀기

I 시뮬레이션의 핵심



cnt:= 연쇄적으로 넘어질 도미노의 개수

Chapter. 03 모의 코딩테스트 풀이 - 1

I 시뮬레이션의 핵심

2. 수비, Defense

- 특정 위치의 상태를 **복원**하기

I 시간, 공간 복잡도 계산하기

한 번의 공격은 $O(N)$ 의 시간이, 한 번의 수비는 $O(1)$ 의 시간이 걸리므로,
총 시간 복잡도는 $O(R * N)$ 이 된다.

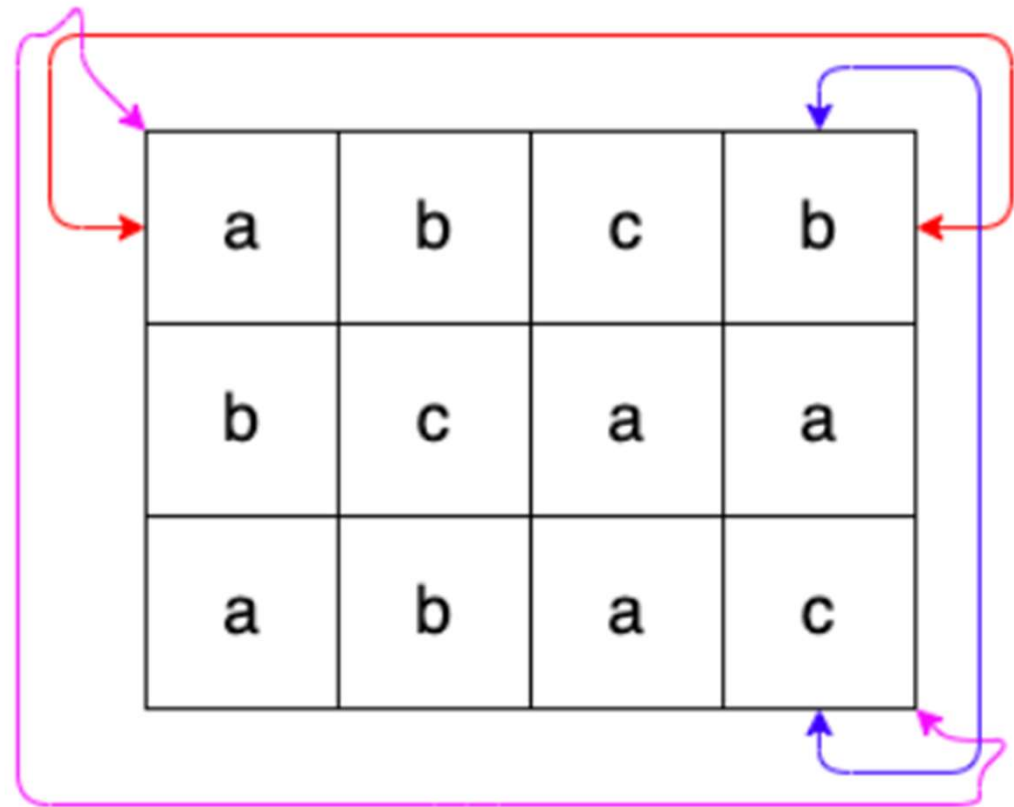
I 구현

```
// x 행 y 열에 있는 도미노를 dir 방향으로 밀어버리는 함수
static void attack(int x, int y, char dir) {
}

// 게임 순서에 맞게 진행시키는 함수
static void pro() {
}
```

I BOJ 20165 – 문자열 지옥에 빠진 호석

난이도: 3

 $3 \leq \text{행 } N, \text{ 열 } M \leq 10$ $1 \leq K \leq 1,000$ $1 \leq \text{신이 좋아하는 문자열의 길이} \leq 5$ 

I 출제 의도

문제에 주어진 조건에 맞게 탐색하는 효율적인 구현하기

이동을 통해 얻을 수 있는 **모든** 문자열은?

- 시작점 결정
- 4번까지의 이동이 가능 (신이 좋아하는 문자열은 최대 5글자)
- 매 이동마다 8가지 방향이 가능

I 생각의 흐름 - 필요한 자료구조는?

1. 어떤 문자열이 몇 번 등장할 수 있는 지를 기록할 변수

$M[\text{문자열 } s] = s\text{의 등장 횟수}$

모든 탐색을 미리 해서, M 이라는 자료구조를 가득 채워놓자.

I 생각의 흐름 - 필요한 자료구조는?

2. 신이 좋아하는 문자열을 입력 받을 때마다 등장 횟수 출력

모든 문자열의 등장 횟수를 M에 모두 기록해버렸으니까,

신이 좋아하는 문자열을 입력 받으면 **바로 출력이 가능하다!**

I 생각의 흐름 - 필요한 자료구조는?

★ 그렇다면 M이라는 자료구조는 어떤 것을 지원해야 하는가? ★

M에 수행할 연산

- 탐색 → “문자열 s”가 한 번 더 등장했어.
- 출력 → “문자열 s”는 몇 번 등장했니?

→ `HashMap<String, Integer> M` 을 사용하면 둘 모두 평균 $O(1)$

I 시간, 공간 복잡도 계산하기

탐색 경우의 수

= 시작 위치 * (4번의 이동, 매번 8방향 가능)

= $NM * 8^4 = 100 * 4,096 = 409,600$

1초 안에 충분히 수행이 가능하다!

I 구현

```
// 현재 위치 (i, j)와 지금까지 만든 경로 문자열 path, 그리고 그 길이 len
static void dfs(int i, int j, String path, int len) {
    /* TODO */
}

static void pro() {
    // 가능한 모든 단어의 등장 횟수 세기
    /* TODO */

    // 신이 좋아하는 문자열들에 대한 대답하기
    /* TODO */
}
```

I BOJ 20181 – 꿈틀꿈틀 호석 애벌레

난이도: 4

$1 \leq \text{먹이 개수 } N \leq 100,000$

$1 \leq K \leq 10^8$

$1 \leq \text{각 먹이의 만족도} \leq 10^8$



수가 크다. 꼭 정답의 최대치를 확인하자.
Long 이 필요함을 미리 깨달아야 한다.

N=9, K=6

1	5	4	4	2	3	10	3	5
---	---	---	---	---	---	----	---	---

1	5	4	4	2	3	10	3	5
---	---	---	---	---	---	----	---	---

만족도: 9

만족도: 15

만족도: 8

I 출제 의도

1. 기능성

➔ 문제의 상황을 시뮬레이션으로 바꿔서 모든 가능한 경우를 수행해보기

2. 효율성

➔ 탐색해야 하는 경우가 너무 많다 => DP를 떠올리는가?

➔ 추가로, 점화식에 필요한 값을 빠르게 계산해 내는 방법으로 Two Pointer 기법을 떠올리는가?

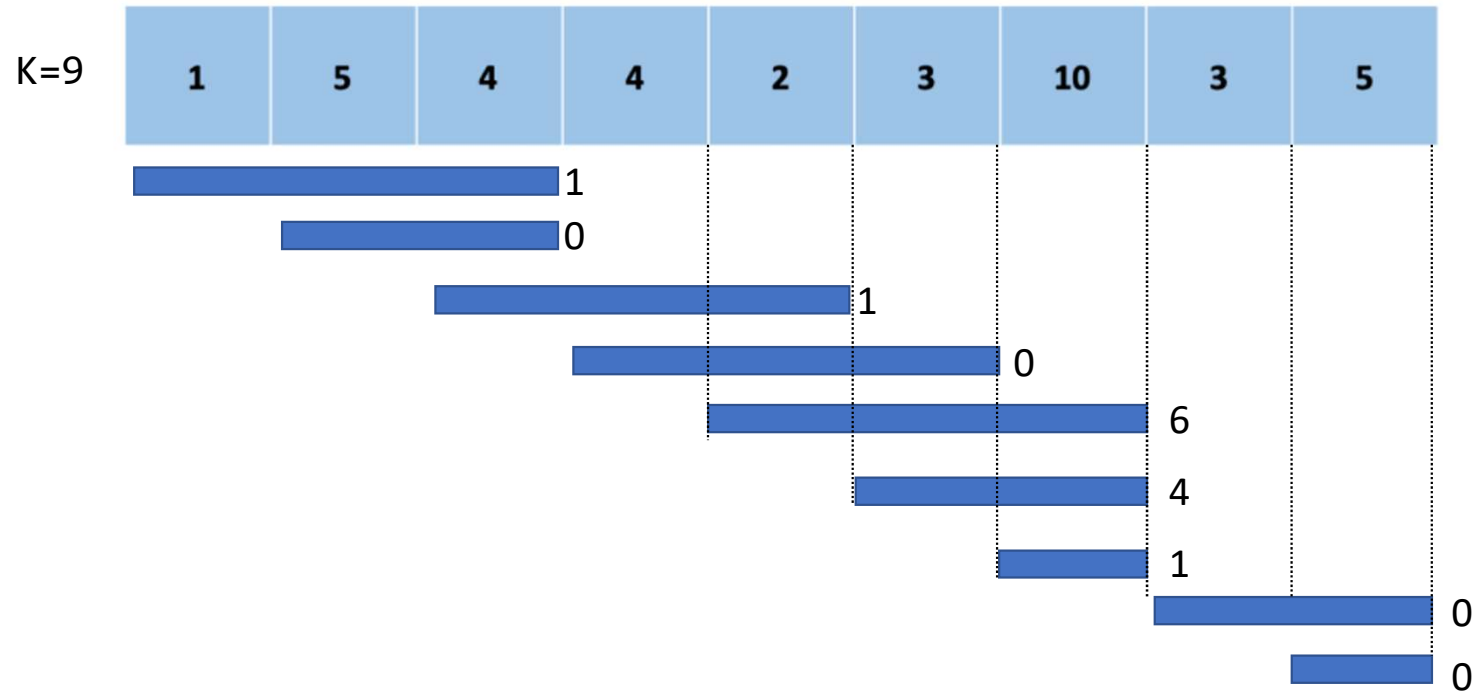
I 생각의 흐름 - 음식을 먹게 되는 구간들

음식을 먹기로 결정하면 그 순간부터 어디까지 먹게 되는지를 **“무조건 정해진다는 것”**을 알게 된다.

L 번째 음식에서 먹기 시작했을 때, 어디까지 먹으며 만족도는 얼마를 얻을 수 있을까?



I 생각의 흐름 - 음식을 먹게 되는 구간들



I 생각의 흐름 - 음식을 먹게 되는 구간들

L을 1부터 N까지 이동시키면서 매번 R을 계산한다고 하자.

L이 1만큼 증가하면 R은 어떻게 될까?

모든 음식이 양수의 만족도를 가지기 때문에, L이 커지면 R은 절대로 작아질 수 없다.

즉, 이전의 L에 대한 R값을 이용해서 다음 L에 대한 R을 계산할 수 있다.

K=9	1	5	4	4	2	3	10	3	5
-----	---	---	---	---	---	---	----	---	---

I 생각의 흐름 – DP라면?

정직한 순서로 문제를 풀어나가면 된다.

먼저, DP Table을 정의해보자.

$Dy[i]$:= i 번 위치까지 도착했을 때, 가능한 최대 만족도

I 생각의 흐름 - DP라면?

$Dy[i] := i$ 번 위치까지 도착했을 때, 가능한 최대 만족도

정답 $:= Dy[N]$

초기값: $Dy[0] = 0$

점화식:

$Dy[i] = \max(i\text{번 음식 안 먹기}, i\text{번 음식까지 먹어서 만족도 얻기})$

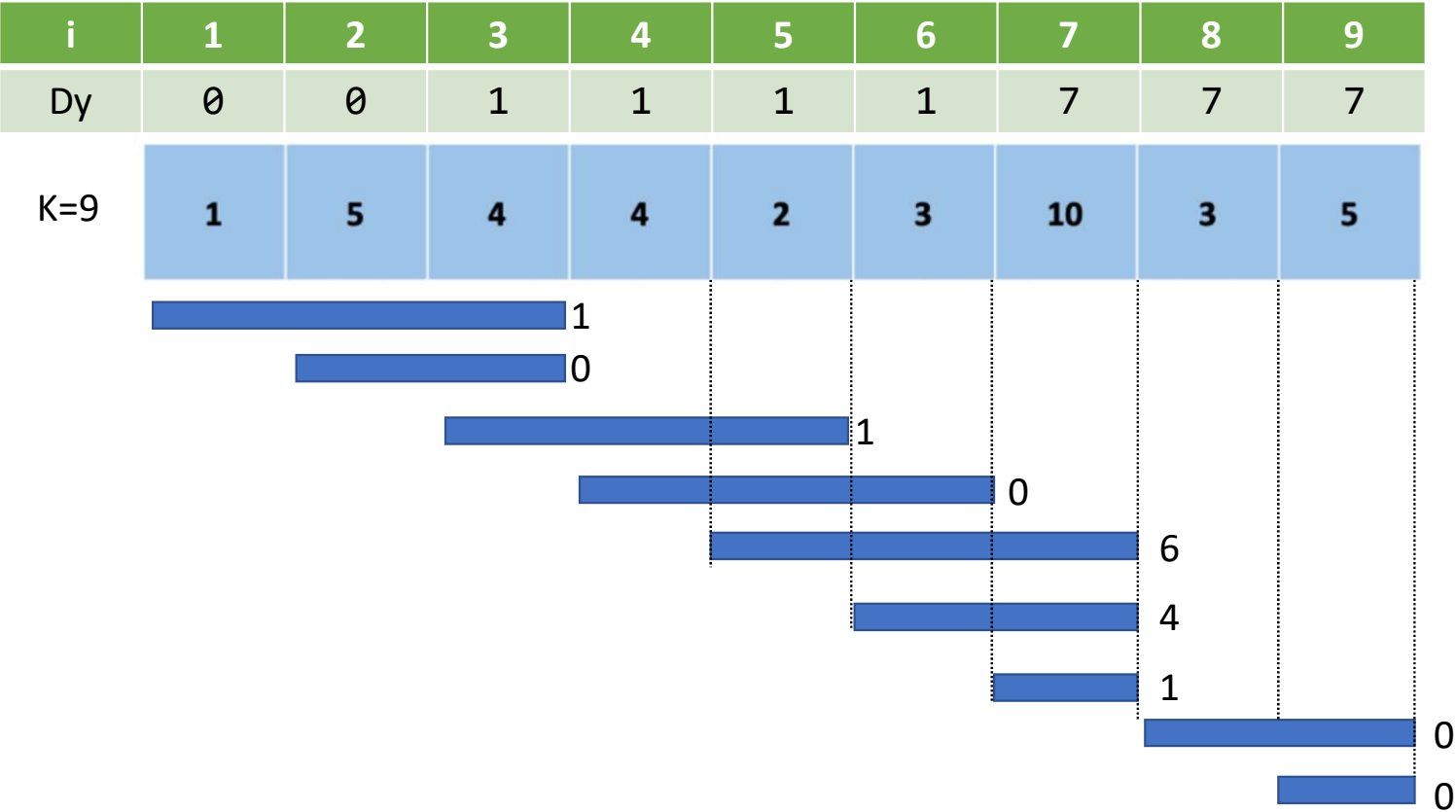
I 생각의 흐름 - 음식을 먹게 되는 구간들

i	1	2	3	4	5	6	7	8	9
Dy	v	v	v	v	v	v	v	v	?



$$\begin{aligned}
 Dy[9] &= \max \left(8\text{까지 먹고 } 9\text{는 안 먹기}, \max \begin{cases} 4\text{까지 먹고 } [5\sim 9] \text{ 먹기} \\ 5\text{까지 먹고 } [6\sim 9] \text{ 먹기} \\ 6\text{까지 먹고 } [7\sim 9] \text{ 먹기} \end{cases} \right) \\
 &= \max \left(Dy[8] + 0, \max \begin{cases} Dy[4] + 6 \\ Dy[5] + 4 \\ Dy[6] + 1 \end{cases} \right)
 \end{aligned}$$

I생각의 흐름 - 음식을 먹게 되는 구간들



I 생각의 흐름 - 전체 흐름

1. 음식을 먹게 되는 구간 $[L, R]$ 을 찾고, 같은 R 에 대해서는 뭉쳐서 저장해 놓는다.
2. R 을 1씩 증가시키면서, $Dy[R]$ 의 값을 계산해 나아간다.
3. 이때 점화식은, $\max(Dy[R-1], \max(Dy[L-1] + Eat(L, R)))$ 이다.
4. 구간을 Two Pointer로 찾고 저장해 놓는다면 시간 복잡도는 $O(N)$ 이 된다.

구현

```

static void pro() {
    long dyLeftMax = 0, sum = 0;
    for (int L = 1, R = 0; L <= N; L++){
        // dyLeftMax 에 max(Dy[1...(L-1)]) 값을 누적하기
        /* TODO */

        // L 에 맞는 R 계산하기
        /* TODO */

        // L, R 에 대해 얻는 만족도를 Dy[R] 에 갱신해주기
        /* TODO */

        // L을 한 칸 이동시킬 때의 sum 변화량 계산해주기
        /* TODO */

    }

    // 정답 계산하기
    /* TODO */
    System.out.println(dyLeftMax);
}

```

I BOJ 20181 – 골목대장 호석

난이도: 5

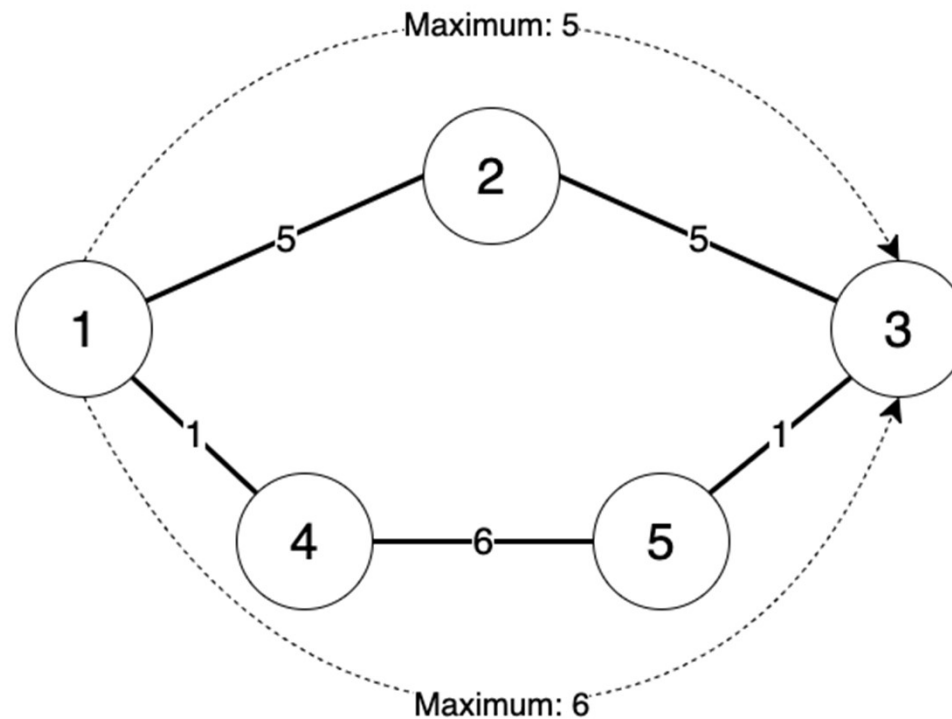
$1 \leq \text{교차로 개수}, N \leq 100,000$

$1 \leq \text{골목 개수}, M \leq 500,000$

$1 \leq \text{가진 돈}, C \leq 10^{14}$



수가 크다. 꼭 정답의 최대치를 확인하자.
Long 이 필요함을 미리 깨달아야 한다.



가진 돈
 $C = 10 \rightarrow \text{최대 } 5$
 $C = 9 \rightarrow \text{최대 } 6$

I 출제 의도

1. 기능성 ➔ 완전 탐색
2. 효율성 1 ➔ Dijkstra 알고리즘 활용
3. 효율성 2 ➔ Dijkstra 알고리즘 활용 + 이분 탐색

I 생각의 흐름 – 완전 탐색 먼저!

항상, 완전 탐색 방법부터 먼저 생각해보자.

DFS를 통해 가진 돈으로 시작점에서 도착점까지 갈 수 있는 모든 경로를 시도해보자.

가능한 경로는 최대 $N!$ 가지 가능하기 때문에 완전 탐색이 가능하다.

I 생각의 흐름 - 정답을 변수로 만들어보자.

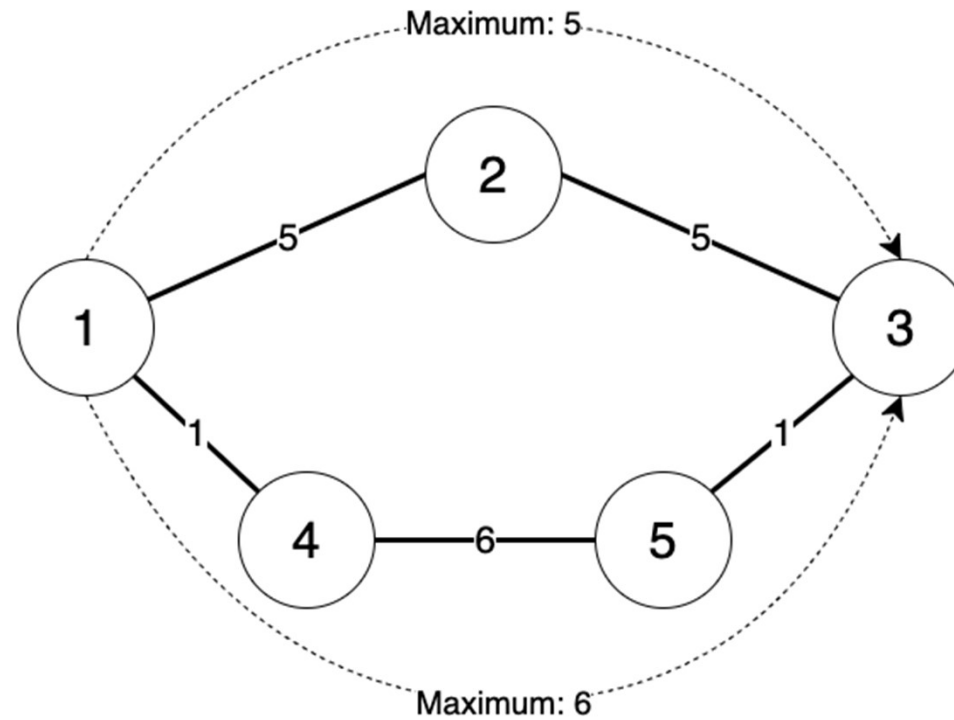
최단경로 → Dijkstra 알고리즘

But, 문제에서 요구하는 “최대 골목 비용”이란 것을 고려할 수가 없다.

따라서, Dijkstra 알고리즘을 쓰기 위해서 최대 골목 비용 x 를 결정해보자.

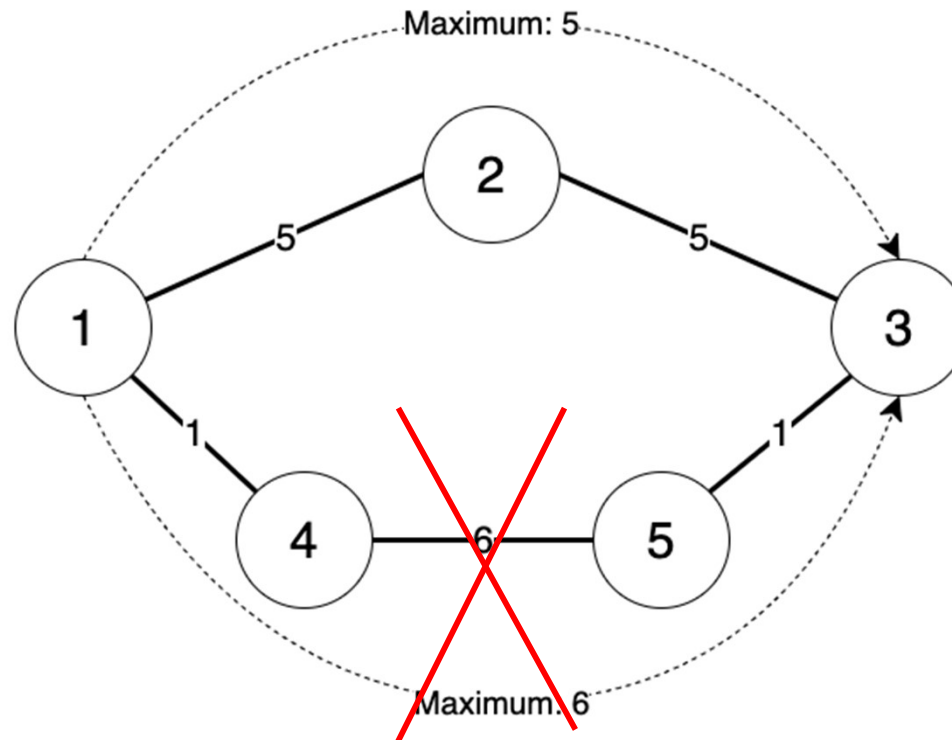
I 생각의 흐름 - 정답을 변수로 만들어보자.

최대 골목 비용 x 를 결정한 상황



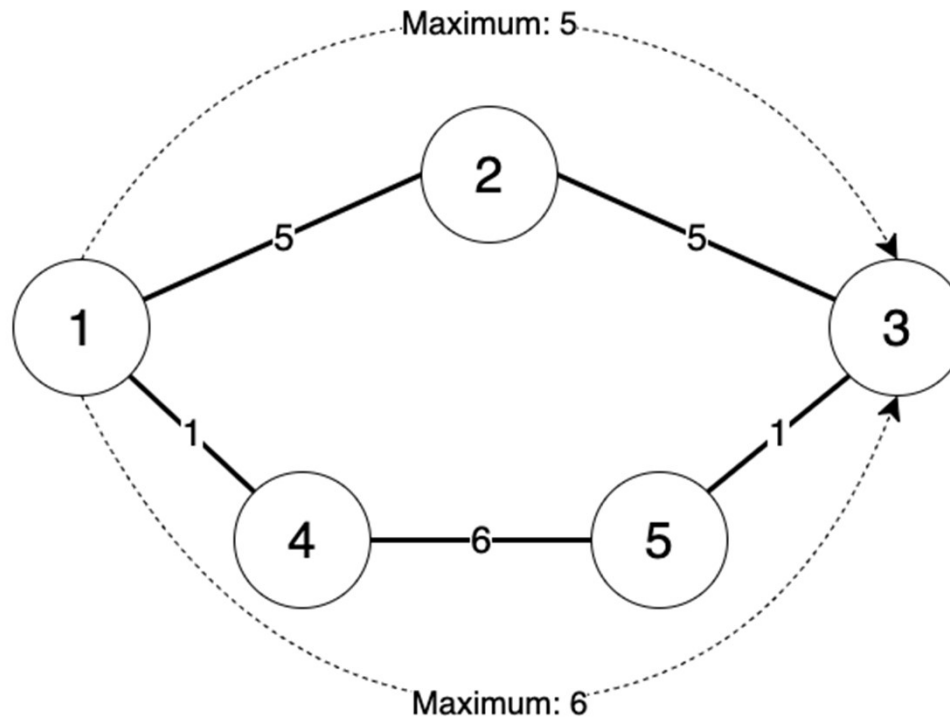
I 생각의 흐름 - 정답을 변수로 만들어보자.

최대 골목 비용 x 를 결정한 상황
 $x = 5$ 라고 하자.



I 생각의 흐름 - 정답을 변수로 만들어보자.

최대 골목 비용 x 를 결정한 상황
 $x = 6$ 라고 하자.




I 생각의 흐름 - 정답을 변수로 만들어보자.

실제 정답이 X^* 라고 하자.


X	1	2	...	$X^* - 1$	X^*	...	10^{14}
최소 비용			

I 생각의 흐름 - 정답을 변수로 만들어보자.

골목 최대치 x 가 늘어나면 사용 가능한 간선도 늘어나기 때문에,
최단거리는 점점 줄어들게 된다.




X	1	2	...	$X^* - 1$	X^*	...	10^{14}
최소 비용			




I 생각의 흐름 - 정답을 변수로 만들어보자.

그렇다면, 모든 x 에 대해 확인하지 말고 이분탐색을 통해 x^* 를 찾을 수 있지 않을까? 즉, Parametric Search이다!



x	1	2	...	$x^* - 1$	x^*	...	10^{14}
최소 비용			



I 시간, 공간 복잡도 계산하기

X^* 가 존재하는 정답 구간 : $[1, 10^{14}]$

한 번 x 를 정한 뒤의 최단 경로: $O(E \log V)$

즉, Dijkstra를 $\log 10^{14}$ 번 반복하면 된다.

I 구현

```

static boolean dijkstra(long x) {
    // x 라는 정답에 대해서 dijkstra 알고리즘을 수행하고, 가지고 있는 돈과 비교하는 함수
    /* TODO */
}

static void pro(){
    // 변수 초기화
    d = new long[n + 1];
    long left = 1, right = 10000000001, ans = right;

    // 정답에 대한 이분 탐색 시작
    /* TODO */

    // 정답 출력
    if (ans == 10000000001) ans = -1;
    out.println(ans);
    out.close();
}

```