Chapter 05.
동적 계획법 #2

Clip 01 | [10844] 쉬운 계단 수
2차원 동적배열
관심 대상 영역의 확장

Clip 02 | [1309] 동물원
동적배열의 상태관리

Clip 03 | [2096] 내려가기
동적배열의 메모리 최적화

Clip 04 | [2854] 문제 출제
동적계획법을 이용한 조합의 수

Clip 05 | [1149] RGB 거리
최소 비용을 찾는 동적계획법

Clip 06 | [1932] 정수 삼각형
동적계획법과 최대 최소 경로

Clip 07 | [15486] 퇴사2
확장된 동적 배낭 문제

Clip 08 | [11053] 가장 긴 증가하는 부분 수열
동적배열의 중첩 순회

Ch05. 동적 계획법 #2

1. [10844] 쉬운 계단 수

BOJ10844: 쉬운 계단 수

문제 요약

- 수의 각 숫자간 차이가 1이고, 첫번째 숫자가 0이 아닌 수를 계단수라고 한다
- 1<=N<=100 입력이 들어올 때 N 길이의 계단 수가 총 몇 개 인지 출력

BOJ10844: 쉬운 계단 수

문제 요약

입력 데이터
1

출력 데이터
9

1	2	3	4	5	6	7	8	9

BOJ10844: 쉬운 계단 수

문제 요약

입력 데이터
2

출력 데이터
17

10	12	21	23	32	34	43	45	
54	56	65	67	76	78	87	89	98

BOJ10844: 쉬운 계단 수

문제 분석

- 문제 요구사항: 계단수의 총 개수
  - d[N]: 길이가 N일 때 계단수의 총 개수
- 계단수를 생성할 때는 마지막에 있는 숫자에 영향을 받는다
  - 0과 9는 각각 {-1, +1} 방향으로 계단수를 만들 수 없기 때문
- 따라서 관심 대상에 마지막 숫자를 포함하여 동적배열을 구성해보자
  - d[N][L] = 길이가 N이고, 마지막 숫자가 L일 때 계단수의 총 개수

BOJ10844: 쉬운 계단 수

문제 분석

d[N][L] = 길이가 N이고, 마지막 숫자가 L일 때 계단수의 총 개수

- {N-1} 길이의 마지막 숫자 별 계단수를 모두 알고 있다고 가정하면 {N}번째 계단수를 계산할 수 있다

- (L == 0) d[N][L] = d[N-1][L + 1]
  - 마지막 숫자가 0이면 -1 방향으로 계단수를 만들 수 없다
- (1 <= L <= 8) d[N][L] = d[N-1][L - 1] + d[N - 1][L + 1]
- (L == 9) d[N][L] = d[N-1][L - 1]
  - 마지막 숫자가 0이면 +1 방향으로 계단수를 만들 수 없다

d[N][L] = 길이가 N이고, 마지막 숫자가 L일 때 계단수의 총 개수

BOJ10844: 쉬운 계단 수

구현

- (L == 0) d[N][L] = d[N-1][L + 1]
- (L == 9) d[N][L] = d[N-1][L – 1]
- (1 <= L <= 8) d[N][L] = d[N-1][L - 1] + d[N - 1][L + 1]

```
for (int i = 2; i <= n; i++) {
    for(int j = 0; j <= 9; j++) {
        switch (j) {
            case 0 -> d[i][j] = d[i - 1][j + 1] % MOD;
            case 9 -> d[i][j] = d[i - 1][j - 1] % MOD;
            default -> d[i][j] = (d[i - 1][j - 1] + d[i - 1][j + 1]) % MOD;
}}}
```

BOJ10844: 쉬운 계단 수

구현

- 길이가 1인 계단수는?
  - L == 0 은 문제의 정의에 의해 존재하지 않는다
  - 나머지는 각각 자기 자신의 숫자를 포함한 1개만 존재한다

```
for (int i = 1; i <= 9; i++) {
    d[1][i] = 1;
}
```

BOJ10844: 쉬운 계단 수

구현

- 최종 정답은 길이가 N일 때 계단 수이다
  - n번째 행의 모든 열을 더해서 출력을 하면 된다
  - 이때, 나머지 연산을 잊지 말고 포함해야 한다

```
int sum = 0;
for (int i = 0; i <= 9; i++) {
   sum = (sum + d[n][i]) % MOD;
}
System.out.println(sum);
```

Ch05. 동적 계획법 #2

2. [1309] 동물원

BOJ1309: 동물원

문제 요약

- N*2 칸의 공간이 주어짐 (1 <= N <= 100,000)
- 가로 세로 인접한곳에 사자를 배치할 수 없음
- 사자를 배치할 수 있는 경우의 수를 계산
  (한 마리도 배치 못하는 경우도 한가지로 취급)

BOJ1309: 동물원

문제 분석

- 문제 요구사항: 사자를 배치하는 경우의 수
  - d[N] = N * 2 배열에 사자를 배치하는 경우의 수
- {N-1}번째 경우를 이용하려고 하니 인접한 칸에 사자를 배치할 수 없다
  - 왼쪽에 사자가 있었다면:

    {N}에는 오른쪽에만 배치할 수 있다
  - 오른쪽에 사자가 있었다면:

    {N}에는 왼쪽에만 배치할 수 있다

BOJ1309: 동물원

문제 분석

- {N-1}번째 경우를 이용하려고 하니 인접한 칸에 사자를 배치할 수 없다
  - 왼쪽에 사자가 있었다면:
    {N}에는 오른쪽에만 배치할 수 있다
  - 오른쪽에 사자가 있었다면:
    {N}에는 왼쪽에만 배치할 수 있다
  - 두 칸 모두 사자가 없었다면?
    {N}에는 양쪽 모두 배치할 수 있다

BOJ1309: 동물원

문제 분석

- 문제를 행 단위로 쪼개서 생각해보자
  만들 수 있는 상태는 몇가지일까?
  - 왼쪽에 배치하기
  - 오른쪽에 배치하기
  - 둘 다 배치하지 않기

- 동적 배열의 차원을 늘려, 배치 상태 추가 정보를 부여한다
  - d[LEFT][N], d[RIGHT][N], d[NONE][N]

BOJ1309: 동물원

문제 분석

- 이제 아래 경우에 대해 점화관계를 생각해보자
- d[LEFT][N] = d[RIGHT][N-1] + d[NONE][N-1]
  왼쪽에 사자를 배치하려면, 이전에 오른쪽에 있거나 둘 다 비어 있어야 한다
- d[RIGHT][N] = d[LEFT][N-1] + d[NONE][N-1]
  오른쪽에 사자를 배치하려면, 이전에 왼쪽에 있거나 둘 다 비어 있어야 한다
- d[NONE][N] = d[LEFT][N-1] + d[RIGHT][N-1] + d[NONE][N-1]
  사자를 배치하지 않는다면, 이전에 모든 상태가 가능하다

BOJ1309: 동물원

구현 - 초기화

```java
final int NONE = 0;
final int LEFT = 1;
final int RIGHT = 2;

int[][] d = new int[3][n + 1];
d[LEFT][1] = 1;
d[RIGHT][1] = 1;
d[NONE][1] = 1;
```

- NONE, LEFT, RIGHT 상태를 상수로 정의
- 첫번째 행에 배치하는 경우는 각각 1가지로 대입

BOJ1309: 동물원

구현 - 초기화

```
for (int i = 2; i <= n; i++) {
    d[LEFT][i] = (d[RIGHT][i - 1] + d[NONE][i - 1]) % 9901;
    d[RIGHT][i] = (d[LEFT][i - 1] + d[NONE][i - 1]) % 9901;
    d[NONE][i] = (d[LEFT][i - 1] + d[RIGHT][i - 1] + d[NONE][i - 1]) % 9901;
}
```

- 왼쪽에 사자를 배치하려면, 이전에 오른쪽에 있거나 둘 다 비어 있어야 한다
- 오른쪽에 사자를 배치하려면, 이전에 왼쪽에 있거나 둘 다 비어 있어야 한다
- 사자를 배치하지 않는다면, 이전에 모든 상태가 가능하다

- 문제의 요구사항에 맞게 9901로 모듈러 연산을 취한다

Ch05. 동적 계획법 #2

3. [2096] 내려가기

BOJ2096: 내려가기

문제 요약

- N*3 칸의 공간이 주어짐 (1 <= N <= 100,000)
  0 <= (각 칸의 값) <= 9
- 인접한 칸으로만 이동이 가능함
  (단, 인덱스 범위를 넘어갈 순 없음)
  - 현재 [n][i] 칸에 있다면?
    - 다음은 [n+1][i-1], [n+1][i], [n+1][i+1]
- 경로의 합의 최소, 최대 구하기

BOJ2096: 내려가기

문제 요약

입력 데이터
3 1 2 3 4 5 6 4 9 0

출력 데이터
18 6

BOJ2096: 내려가기

문제 요약

입력 데이터
3 1 2 3 4 5 6 4 9 0

출력 데이터
18 6

BOJ2096: 내려가기

문제 요약

입력 데이터
3 1 2 3 4 5 6 4 9 0

출력 데이터
18 6

BOJ2096: 내려가기

문제 분석

- 문제 요구사항
  - 경로의 최대 합
  - 경로의 최소 합

- dmx[r][c] = (r, c) 까지 이동하는 경로의 최대 합
- dmn[r][c] = (r, c) 까지 이동하는 경로의 최소 합

BOJ2096: 내려가기

문제 분석

- dmx[r][c] = arr[r][c] +
  max(dmx[r-1][c-1], dmx[r-1][c], dmx[r-1][c+1])

- dmn[r][c] = arr[r][c] +
  min(dmn[r-1][c-1], dmn[r-1][c], dmn[r-1][c+1])

단, 좌표가 범위를 벗어나면 안된다
매번 비교로 구현할 수 있지만, 다른 방법으로 접근해 보자

BOJ2096: 내려가기

문제 분석

입력 배열		
[1]	[2]	[3]
1	2	3
4	5	6
4	9	0

dmx(경로의 최대 합)				
[0]	[1]	[2]	[3]	[4]
0	1	2	3	0
0				0
0				0

BOJ2096: 내려가기

문제 분석 – 최대 합

입력 배열

[1]	[2]	[3]
1	2	3
4	5	6
4	9	0

dmx(경로의 최대 합)

[0]	[1]	[2]	[3]	[4]
0	1	2	3	0
0	4+2			0
0				0

{0}열과 {4}열을 0으로 초기화, [1, 3] 범위에서 최대 합 계산

BOJ2096: 내려가기

문제 분석 – 최대 합

입력 배열

[1]	[2]	[3]
1	2	3
4	5	6
4	9	0

dmx(경로의 최대 합)

[0]	[1]	[2]	[3]	[4]
0	1	2	3	0
0	4+2	5+3		0
0				0

{0}열과 {4}열을 0으로 초기화, [1, 3] 범위에서 최대 합 계산

BOJ2096: 내려가기

문제 분석 – 최대 합

입력 배열

[1]	[2]	[3]
1	2	3
4	5	6
4	9	0

dmx(경로의 최대 합)

[0]	[1]	[2]	[3]	[4]
0	1	2	3	0
0	4+2	5+3	6+3	0
0				0

{0}열과 {4}열을 0으로 초기화, [1, 3] 범위에서 최대 합 계산

BOJ2096: 내려가기

문제 분석 – 최소 합

입력 배열

[1]	[2]	[3]
1	2	3
4	5	6
4	9	0

dmn(경로의 최소 합)

[0]	[1]	[2]	[3]	[4]
1,000,000	1	2	3	1,000,000
1,000,000	4+1			1,000,000
1,000,000				1,000,000

{0}열과 {4}열을 1,000,0000으로 초기화, [1, 3] 범위에서 최소 합 계산

BOJ2096: 내려가기

문제 분석 – 최소 합

입력 배열

[1]	[2]	[3]
1	2	3
4	5	6
4	9	0

dmx(경로의 최대 합)

[0]	[1]	[2]	[3]	[4]
1,000,000	1	2	3	1,000,000
1,000,000	4+1	5+1		1,000,000
1,000,000				1,000,000

{0}열과 {4}열을 1,000,0000으로 초기화, [1, 3] 범위에서 최소 합 계산

BOJ2096: 내려가기

문제 분석 – 최소 합

입력 배열

[1]	[2]	[3]
1	2	3
4	5	6
4	9	0

dmx(경로의 최대 합)

[0]	[1]	[2]	[3]	[4]
1,000,000	1	2	3	1,000,000
1,000,000	4+1	5+1	6+2	1,000,000
1,000,000				1,000,000

{0}열과 {4}열을 1,000,0000으로 초기화, [1, 3] 범위에서 최소 합 계산

BOJ2096: 내려가기

문제	결과
2096	메모리 초과

[1]	[2]	[3]
4	9	0

메모리 제한
4 MB (하단 참고)

[0]	[1]	[2]	[3]	[4]
1,000,000				1,000,000
1,000,000				1,000,000
1,000,000				1,000,000

메모리 제한

- Java 8: 256 MB
- Java 8 (OpenJDK): 256 MB
- Java 11: 256 MB
- Kotlin (JVM): 256 MB

{0}열과 {4}열을 1,000,0000으로 초기화, [1, 3] 범위에서 최소 합 계산

BOJ2096: 내려가기

문제 분석

- 사용하는 메모리 공간을 최적화 해야 한다
- 아이디어:
  {i}번째 줄의 결과를 구할 때 {i-1} 이외의 정보가 필요할까?
  → 동적 배열을 1차원으로 두고 처리해보자

BOJ2096: 내려가기

구현

- dmx[r][c] = arr[r][c] +
  max(dmx[r-1][c-1], dmx[r-1][c], dmx[r-1][c+1])

- dmn[r][c] = arr[r][c] +
  min(dmn[r-1][c-1], dmn[r-1][c], dmn[r-1][c+1])

- dmx[i] = input + max(dmx[i-1], dmx[i], dmx[i+1])
- dmn[i] = input + min(dmn[i-1], dmn[i], dmn[i+1])

BOJ2096: 내려가기

구현

- dmx[i] = input + max(dmx[i-1], dmx[i], dmx[i+1])
- dmn[i] = input + min(dmn[i-1], dmn[i], dmn[i+1])

하나의 행의 연산이 끝나기 전에 결과가 덮어쓰기 된다
행 단위로 최소 / 최대 를 구할 때 까지 값의 보존이 필요하다

BOJ2096: 내려가기

구현

- tmx[i] = input + max(dmx[i-1], dmx[i], dmx[i+1])
- tmn[i] = input + min(dmn[i-1], dmn[i], dmn[i+1])

```
for(int j = 1; j <= 3; j++) {
    dmx[j] = tmx[j];
    dmn[j] = tmn[j];
}
```

행 단위 최소 / 최대 를 구했다면 반복문으로 한번에 덮어쓰기 한다

BOJ2096: 내려가기

구현

- tmx[i] = input + max(dmx[i-1], dmx[i], dmx[i+1])
- tmn[i] = input + min(dmn[i-1], dmn[i], dmn[i+1])

```
for(int j = 1; j <= 3; j++) {
    dmx[j] = tmx[j];
    dmn[j] = tmn[j];
}
```

행 단위 최소 / 최대 를 구했다면 반복문으로 한번에 덮어쓰기 한다

BOJ2096: 내려가기

구현

```
for(int i = 1; i <= n; i++) {
    for(int j = 1; j <= 3; j++) {
        input = sc.nextInt();
        tmx[j] = Math.max(dmx[j - 1], Math.max(dmx[j], dmx[j + 1])) + input;
        tmn[j] = Math.min(dmn[j - 1], Math.min(dmn[j], dmn[j + 1])) + input;
    }
    for(int j = 1; j <= 3; j++) {
        dmx[j] = tmx[j];
        dmn[j] = tmn[j];
    }
}
```

Ch05. 동적 계획법 #2

4. [2854] 문제 출제

(hard)

BOJ2854: 문제 출제

문제 요약

- 문제마다 난이도가 부여되어 있음

- type1: 고정된 하나의 값 {i}로 부여된 난이도
- type2: 난이도를 {i}, {i+1} 둘 다 선택이 가능한 문제

- 모든 난이도별로 문제를 1개 출제할 때 (2 <= N <= 100,000)
  가능한 문제 조합의 수

BOJ2854: 문제 출제

문제 분석

입력 데이터
4 1 5 3 0 0 2 1

출력 데이터
33

- 난이도1: 1개
- 난이도2: 5개
- 난이도3: 3개
- 난이도4: 0개

- 난이도1~2: 0개
- 난이도2~3: 2개
- 난이도3~4: 1개

BOJ2854: 문제 출제

문제 분석

- 문제 요구사항: 가능한 출제 조합의 수
  - (이전 단계까지 조합의 개수) * (새로운 문제의 개수)
- 그런데, 변동 난이도로 인해 {i}가 되거나
  {i+1}이 되는 경우가 발생할 수 있어 복잡하다
- 우선 케이스를 분리해서 생각해보자

BOJ2854: 문제 출제

문제 분석

- [CASE1]
  난이도가 고정된 문제만 사용하는 경우
- [CASE2]
  난이도가 변동되는 문제를 {i} 로 사용한 경우
- [CASE3]
  난이도가 변동되는 문제를 {i+1}로도 사용한 경우

- 문제 요구사항: 가능한 출제 조합의 수
  - d[CASE][L]:
    난이도가 [1, L] 인 문제에서 CASE를 선택해서 출제하는 조합의 수

BOJ2854: 문제 출제

d[CASE][L]:
난이도가 [1, L] 인 문제에서 CASE를 선택해서 출제하는 조합의 수

문제 분석

- d[CASE1][L]
  난이도가 고정된 문제만 사용하여 난이도가 L인 문제의 개수

- d[CASE1][L] = examStatic[L] * (
  d[CASE1][L-1] + d[CASE2][L-1] + d[CASE3][L-1]
)

- 난이도가 {L-1}인 문제에서 조합의 수를 유도하면 된다
- 난이도가 고정된 문제는 다른 문제들에 영향을 주지 않으므로 단순하게 {L-1}의 모든 경우를 합산하면 된다

BOJ2854: 문제 출제

d[CASE][L]:
난이도가 [1, L] 인 문제에서 CASE를 선택해서
출제하는 조합의 수

문제 분석

- d[CASE2][L]
  난이도가 변동되는 문제를 {i} 로 사용한 경우
- d[CASE2][L] = examDynamic[L] * (
  d[CASE1][L-1] + d[CASE2][L-1] + d[CASE3][L-1]
)
- 난이도가 변동임에도 불구하고 {i}로만 한정해서 사용했다면?
  - 난이도가 고정된 문제와 동일하다고 생각할 수 있다
  - 따라서 고정인 문제와 비슷한 구조로 조합의 수를 계산할 수 있다

BOJ2854: 문제 출제

d[CASE][L]:
난이도가 [1, L] 인 문제에서 CASE를 선택해서 출제하는 조합의 수

문제 분석

- [CASE3]
  난이도가 변동되는 문제를 {i+1}로도 사용한 경우
- d[CASE3][L] = ?

1. [L-1] 단계에서 {i+1}만 고른 경우
2. [L-1] 단계에서 {i}, {i+1} 을 섞어서 사용한 경우

BOJ2854: 문제 출제

d[CASE][L]:
난이도가 [1, L] 인 문제에서 CASE를 선택해서 출제하는 조합의 수

문제 분석

- [CASE3]
  난이도가 변동되는 문제를 {i+1}로도 사용한 경우
- d[CASE3][L] = ?

1. [L-1] 단계에서 {i+1}만 고른 경우
   - examDynamic[L-1] * (d[CASE1][L-1] + d[CASE3][L-1])
   - CASE2가 없는 이유는?: {i+1}만 골랐기 때문에

BOJ2854: 문제 출제

d[CASE][L]:
난이도가 [1, L] 인 문제에서 CASE를 선택해서 출제하는 조합의 수

문제 분석

2. [L-1] 단계에서 {i}, {i+1} 을 섞어서 사용한 경우
   - 경우의 수는 [L-2]로 내려가서 참조해야 한다
   - [L-1]에서는 {i}로 뽑는 경우와 {i+1}을 뽑는 조합의 수는 (n * (n-1)) 식으로 계산할 수 있다
   - {L-1 에서 i와 i+1을 뽑는 조합의 수} * {L-2에서 만든 문제 수}
   - ((examDynamic[L-1]) * (examDynamic[L-1] – 1)) * (d[CASE1][L-2] + d[CASE2][L-2] + d[CASE3][L-2])

BOJ2854: 문제 출제

d[CASE][L]:
난이도가 [1, L] 인 문제에서 CASE를 선택해서 출제하는 조합의 수

문제 분석

- [CASE3]
  난이도가 변동되는 문제를 {i+1}로도 사용한 경우
- d[CASE3][L] =

1. [L-1] 단계에서 {i+1}만 고른 경우
   - examDynamic[L-1] * (d[CASE1][L-1] + d[CASE3][L-1])
2. [L-1] 단계에서 {i}, {i+1} 을 섞어서 사용한 경우
   - ((examDynamic[L-1]) * (examDynamic[L-1] – 1)) * (d[CASE1][L-2] + d[CASE2][L-2] + d[CASE3][L-2])

BOJ2854: 문제 출제

구현

- [CASE1]
  난이도가 고정된 문제만 사용하는 경우
- [CASE2]
  난이도가 변동되는 문제를 {i} 로 사용한 경우
- [CASE3]
  난이도가 변동되는 문제를 {i+1}로도 사용한 경우


```
for(int i = 2; i <= n; i++) {
    d[CASE1][i] = ((d[CASE1][i-1] + d[CASE2][i-1] + d[CASE3][i-1]) * examStatic[i]) % MOD;
    d[CASE2][i] = ((d[CASE1][i-1] + d[CASE2][i-1] + d[CASE3][i-1]) * examDynamic[i]) % MOD;
    d[CASE3][i] =
            (d[CASE1][i-1] + d[CASE3][i-1]) * examDynamic[i-1] % MOD +
            (examDynamic[i-1] * (examDynamic[i-1] - 1) % MOD)
                    * (d[CASE1][i-2] + d[CASE2][i-2] + d[CASE3][i-2])  % MOD;
}
long sum = d[CASE1][n] + d[CASE2][n] + d[CASE3][n];
```

Ch05. 동적 계획법 #2

5. [1149] RGB 거리

BOJ1149: RGB 거리

문제 요약

- N개의 집 [1, N] (2 <= N <= 1000)
- 집은 빨강, 초록, 파랑 중 하나의 색으로 칠해야 함
  - 칠하는 비용이 색마다 다름
- {i}번째 집은 {i-1}, {i+1} 번째 집과 색이 같지 않아야 함
- 모든 집을 칠하는 비용의 최솟값 출력

BOJ1149: RGB 거리

문제 요약

입력 데이터
3 26 40 83 49 60 57 13 89 99

출력 데이터
96

BOJ1149: RGB 거리

문제 요약

입력 데이터
3 1 100 100 100 1 100 100 100 1

출력 데이터
3

BOJ1149: RGB 거리

문제 분석

- 색상이 중복되지 않으면서, 최소비용을 선택하는 문제
- 문제 요구사항: 비용의 최소 값
  - d[n] = n번째 집까지의 비용의 최소 값
- {n}번째 집은 R / G / B 색상 중 하나일 수 있다
  - 단, {n} 번째 집에서 사용하지 않은 색상
- 마찬가지로 {n-1} 집도 R / G / B 색상 중 하나일 수 있다
  - 단, {n-2} 번째 집에서 사용하지 않은 색상

→ 색상을 관리하며 카운트 하도록 동적 배열을 확장한다

BOJ1149: RGB 거리

문제 분석

- d [n][COLOR] = n번째 집을 COLOR로 칠했을 때,
  최소로 하는 비용
- n번째 집을 R로 칠하는 경우
  - d[n][R] = d[n-1][G] + d[n-1][B] + costR
- n번째 집을 G로 칠하는 경우
  - d[n][G] = d[n-1][R] + d[n-1][B] + costG
- n번째 집을 B로 칠하는 경우
  - d[n][B] = d[n-1][R] + d[n-1][G] + costB

BOJ1149: RGB 거리

구현

```
final int R = 1, G = 2, B = 3;
for (int i = 1; i <= n; i++) {
    int r = sc.nextInt();
    int g = sc.nextInt();
    int b = sc.nextInt();
    d[i][R] = Math.min(d[i - 1][G], d[i - 1][B]) + r;
    d[i][G] = Math.min(d[i - 1][R], d[i - 1][B]) + g;
    d[i][B] = Math.min(d[i - 1][R], d[i - 1][G]) + b;
}
```

매 순간 {i}번째 R/G/B 비용을 입력 받으며 바로 계산할 수 있다

Ch05. 동적 계획법 #2

6. [1932] 정수 삼각형

BOJ1932: 정수 삼각형

문제 요약

- 정수로 이루어진 삼각형 (1 <= n <= 500)
- 수를 하나씩 선택하며 아래층으로 내려올 때,
  선택된 수의 합이 최대가 되는 경로 찾기
- 수는 {왼쪽 or 오른쪽} 대각선 중에서만 선택 가능

BOJ1932: 정수 삼각형

문제 분석

- 수는 {왼쪽 or 오른쪽} 대각선 중에서만 선택 가능
- 배열의 편한 관리를 위해 아래처럼 삼각형을 바꿔보자

```
        7
      3   8
    8   1   0
  2   7   4   4
4   5   2   6   5
```

```
7
3 8
8 1 0
2 7 4 4
4 5 2 6 5
```

수는 {아래 or 오른쪽 대각선} 중에서만 선택 가능

BOJ1932: 정수 삼각형

문제 분석

- 문제 요구사항: 최대가 되는 경로의 합
  - d[n]: n번째 줄까지 경로의 합 중 최대 값

- 1차원 동적배열로 바로 문제를 풀이할 수 있는가? (X)
  - {i+k} 번째 줄의 큰 값을 고르기 위해 {i}번째 줄에서 희생하고 작은 값을 골라야 할 수도 있다
  - 각 행에서 고른 수에 따라 상태가 최종 상태가 변한다

- d[n][i]: {n}번째 줄(행) {i}번째 수(열) 까지 경로의 합 중 최대 값

BOJ1932: 정수 삼각형

문제 분석

- d[i][j]: {i}번째 줄(행) {j}번째 수(열) 까지 경로의 합 중 최대 값

방법 1. 현재 위치를 기준으로 아래를 계산한다면?

- d[i + 1][j] = max(d[i][j] + a[i + 1][j], d[i + 1][j])
- d[i + 1][j + 1] = max(d[i][j] + a[i + 1][j + 1], d[i + 1][j + 1])

7	
3	8

7	
10	15

BOJ1932: 정수 삼각형

문제 분석

- d[i][j]: {i}번째 줄(행) {j}번째 수(열) 까지 경로의 합 중 최대 값

방법 2. 현재 위치에 값을, 이전 결과로 구한다면?

- d[i][j] = max(d[i-1][j-1], d[i-1][j]) + a[i][j]

3	8	
8	1	0

10	15	
18	?	?

BOJ1932: 정수 삼각형

구현

방법 1. 현재 위치를 기준으로 아래를 계산한다면?

```
for(int i = 1; i < n; i++) {
    for(int j = 1; j <= i; j++) {
        d[i + 1][j] = Math.max(d[i + 1][j], d[i][j] + a[i + 1][j]);
        d[i + 1][j + 1] = Math.max(d[i + 1][j + 1], d[i][j] + a[i + 1][j + 1]);
    }
}
```

BOJ1932: 정수 삼각형

구현

방법 2. 현재 위치에 값을, 이전 결과로 구한다면?

```
for(int i = 2; i <= n; i++) {
    for(int j = 1; j <= i; j++) {
        d[i][j] = Math.max(d[i - 1][j - 1], d[i - 1][j]) + a[i][j];
    }
}
```

Ch05. 동적 계획법 #2

7. [15486] 퇴사 2

BOJ15486: 퇴사 2

문제 요약

- N일동안 최대한 많은 돈을 벌 수 있도록
  상담 스케줄 정하기 (1 <= N <= 1,500,000)
- 상담의 길이 T, 보수 금액 P
  - (1 <= T <= 50), (1 <= P <= 1000)
- 동시에 여러 상담 진행 불가능
  종료일이 N일을 넘어가면 진행 불가능

BOJ15486: 퇴사 2

문제 분석

- 문제 요구사항: 최대 수익
  - d[n]: n일동안 상담을 했을 때 최대 수익

- 돈을 받으려면 해당 일까지 지정된 상담을 완료해야 한다.
  따라서 d[n] 은 진행중인 상담에 대한 금액이 들어가지 않는다

- 상담 완료일을 기준으로 점화 관계를 접근해보자

BOJ15486: 퇴사 2

문제 분석

- {i} 일마다 {T[i]} 길이의 상담이 주어진다
  - 이 상담은 {i + T[i]} 일에 종료가 된다
    - 종료 뒤에는 P[i] 의 보수를 받는다

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1	10								
2		20							
3			10						
4				20					
5					15				
6						40			
7							200		

BOJ15486: 퇴사 2

문제 분석

- {i} 일마다 {T[i]} 길이의 상담이 주어진다
  - 이 상담은 {i + T[i]} 일에 종료가 된다
    - 종료 뒤에는 P[i] 의 보수를 받는다

- d[i + t[i]] = max(d[i + t[i]], d[i] + p[i])
  - {i}번째 일을 하지 않고 여태까지 번 최대의 돈
  - 일이 시작되는 시점까지 번 돈 + {i}번째 일을 하고 번 돈

BOJ15486: 퇴사 2

문제 분석

- 문제 요구사항: 최대 수익
  - d[n]: n일동안 상담을 했을 때 최대 수익
- 만약 d[i] 보다 d[i-1] 이 더 크다면?
  - d[i] = d[i-1] 로 최대 수익을 갱신
- 정답은? d[n+1]
  - n일에 끝난 보수는 n일이 종료된 뒤에 받는다
    - 따라서 n+1 칸 에서 정답을 가져온다

BOJ15486: 퇴사 2

구현

```
for(int i = 1; i <= n + 1; i++) {
    d[i] = Math.max(d[i-1], d[i]);

    if(i + t[i] > n + 1) continue;
    d[i + t[i]] = Math.max(d[i + t[i]], d[i] + p[i]);
}
System.out.println(d[n + 1]);
```

- d[i + t[i]] = max(d[i + t[i]], d[i] + p[i])
  - {i}번째 일을 하지 않고 여태까지 번 최대의 돈
  - 일이 시작되는 시점까지 번 돈 + {i}번째 일을 하고 번 돈

Ch05. 동적 계획법 #2

8. [11053] 가장 긴 증가하는 부분 수열

BOJ11053: 가장 긴 증가하는 부분 수열

문제 요약

- 수열 중에서 부분 수열을 구해야 한다
- 부분 수열은 오름차순으로 증가해야 한다
- 만들 수 있는 부분 수열 중에 가장 긴 길이를 출력

BOJ11053: 가장 긴 증가하는 부분 수열

부분 수열?

원본 수열에서 원소를 뽑아 만든 새로운 수열

부분 문자열(Substring) : 연속성을 가진다
부분 수열 (Subsequence) : 연속성을 가지지 않아도 된다

ex) ABCDEF의 BCD : Substring(O) Subsequence(O)
ABCDEF의 ACF : Substring(X) Subsequence(O)

BOJ11053: 가장 긴 증가하는 부분 수열

문제 요약

입력 데이터
6 10 20 10 30 20 50

출력 데이터
4

BOJ11053: 가장 긴 증가하는 부분 수열

문제 분석

- 문제 요구사항: 부분 수열의 가장 긴 길이
    - d[i]: {i}번째 수까지 사용했을 때 가장 긴 부분 수열의 길이
- 위의 점화식으로 한번에 정답을 구할 수 있을까?
    - {i}번째 수를 고르지 않아야
      {i + k} 번째 수를 고를 수 있는 경우가 발생할 수 있다
- 그렇다면?
    - 반복문을 한번 더 돌면서 배열을 갱신해본다

BOJ11053: 가장 긴 증가하는 부분 수열

문제 분석

arr[]	10	20	10	30	20	50

d[]	0	0	0	0	0	0

BOJ11053: 가장 긴 증가하는 부분 수열

문제 분석

arr[]	10	20	10	30	20	50

i

d[]	1	0	0	0	0	0

i

d[i] 의 최소 길이는 1이다 (자기 자신)

BOJ11053: 가장 긴 증가하는 부분 수열

문제 분석

arr[]	10	20	10	30	20	50
	j	i				
d[]	1	2	0	0	0	0
	j	i				

j < i 까지 탐색하며, arr[j] < arr[i] 를 만족한다면?
d[i] 배열을 {d[j] + 1} 길이로 갱신한다. 단, 기존 값보다 작으면 무시

BOJ11053: 가장 긴 증가하는 부분 수열

문제 분석

arr[]	10	20	10	30	20	50
	j		i			

d[]	1	2	1	0	0	0
	j		i			

j < i 까지 탐색하며, arr[j] < arr[i] 를 만족한다면?
d[i] 배열을 {d[j] + 1} 길이로 갱신한다. 단, 기존 값보다 작으면 무시

BOJ11053: 가장 긴 증가하는 부분 수열

문제 분석

arr[]	10	20	10	30	20	50
d[]	1	2	1	0	0	0

j < i 까지 탐색하며, arr[j] < arr[i] 를 만족한다면?
d[i] 배열을 {d[j] + 1} 길이로 갱신한다. 단, 기존 값보다 작으면 무시

BOJ11053: 가장 긴 증가하는 부분 수열

문제 분석

arr[]	10	20	10	30	20	50
	j			i		
d[]	1	2	1	2	0	0
	j			i		

j < i 까지 탐색하며, arr[j] < arr[i] 를 만족한다면?
d[i] 배열을 {d[j] + 1} 길이로 갱신한다. 단, 기존 값보다 작으면 무시

BOJ11053: 가장 긴 증가하는 부분 수열

문제 분석

arr[]	10	20	10	30	20	50
		j		i		

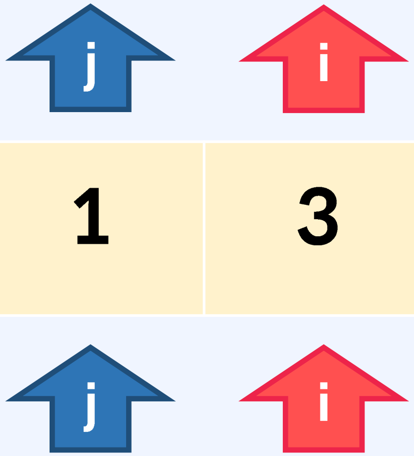
d[]	1	2	1	3	0	0
		j		i		

j < i 까지 탐색하며, arr[j] < arr[i] 를 만족한다면?
d[i] 배열을 {d[j] + 1} 길이로 갱신한다. 단, 기존 값보다 작으면 무시

BOJ11053: 가장 긴 증가하는 부분 수열

문제 분석

arr[]	10	20	10	30	20	50
d[]	1	2	1	3	0	0



j < i 까지 탐색하며, arr[j] < arr[i] 를 만족한다면?
d[i] 배열을 {d[j] + 1} 길이로 갱신한다. 단, 기존 값보다 작으면 무시

BOJ11053: 가장 긴 증가하는 부분 수열

문제 분석

arr[]	10	20	10	30	20	50

j i

d[]	1	2	1	3	2	0

j i

j < i 까지 탐색하며, arr[j] < arr[i] 를 만족한다면?
d[i] 배열을 {d[j] + 1} 길이로 갱신한다. 단, 기존 값보다 작으면 무시

BOJ11053: 가장 긴 증가하는 부분 수열

문제 분석

arr[]	10	20	10	30	20	50

d[]	1	2	1	3	2	0

j < i 까지 탐색하며, arr[j] < arr[i] 를 만족한다면?
d[i] 배열을 {d[j] + 1} 길이로 갱신한다. 단, 기존 값보다 작으면 무시

BOJ11053: 가장 긴 증가하는 부분 수열

문제 분석

arr[]	10	20	10	30	20	50
			j		i	
d[]	1	2	1	3	2	0
			j		i	

j < i 까지 탐색하며, arr[j] < arr[i] 를 만족한다면?
d[i] 배열을 {d[j] + 1} 길이로 갱신한다. 단, 기존 값보다 작으면 무시

BOJ11053: 가장 긴 증가하는 부분 수열

문제 분석

arr[]	10	20	10	30	20	50
				j	i	

d[]	1	2	1	3	2	0
				j	i	

j < i 까지 탐색하며, arr[j] < arr[i] 를 만족한다면?
d[i] 배열을 {d[j] + 1} 길이로 갱신한다. 단, 기존 값보다 작으면 무시

BOJ11053: 가장 긴 증가하는 부분 수열

문제 분석

arr[]	10	20	10	30	20	50
d[]	1	2	1	3	2	2

j < i 까지 탐색하며, arr[j] < arr[i] 를 만족한다면?
d[i] 배열을 {d[j] + 1} 길이로 갱신한다. 단, 기존 값보다 작으면 무시

BOJ11053: 가장 긴 증가하는 부분 수열

문제 분석

arr[]	10	20	10	30	20	50

j i

d[]	1	2	1	3	2	3

j i

j < i 까지 탐색하며, arr[j] < arr[i] 를 만족한다면?
d[i] 배열을 {d[j] + 1} 길이로 갱신한다. 단, 기존 값보다 작으면 무시

BOJ11053: 가장 긴 증가하는 부분 수열

문제 분석

arr[]	10	20	10	30	20	50

j i

d[]	1	2	1	3	2	3

j i

j < i 까지 탐색하며, arr[j] < arr[i] 를 만족한다면?
d[i] 배열을 {d[j] + 1} 길이로 갱신한다. 단, 기존 값보다 작으면 무시

BOJ11053: 가장 긴 증가하는 부분 수열

문제 분석

arr[]	10	20	10	30	20	50

d[]	1	2	1	3	2	4

j < i 까지 탐색하며, arr[j] < arr[i] 를 만족한다면?
d[i] 배열을 {d[j] + 1} 길이로 갱신한다. 단, 기존 값보다 작으면 무시

BOJ11053: 가장 긴 증가하는 부분 수열

문제 분석

arr[]	10	20	10	30	20	50

					j	i

d[]	1	2	1	3	2	4

					j	i

j < i 까지 탐색하며, arr[j] < arr[i] 를 만족한다면?
d[i] 배열을 {d[j] + 1} 길이로 갱신한다. 단, 기존 값보다 작으면 무시

BOJ11053: 가장 긴 증가하는 부분 수열

구현

```
int max = 1;
for(int i = 1; i <= n; i++) {
    d[i] = 1;
    for(int j = 1; j < i; j++) {
        if(arr[j] < arr[i]) {
            if(d[j] + 1 < d[i]) continue;
            d[i] = d[j] + 1;
            max = Math.max(max, d[i]);
        }
    }
}
```

j < i 까지 탐색하며,
arr[j] < arr[i] 를 만족한다면?

d[i] 배열을 {d[j] + 1} 길이로 갱신한다.
단, 기존 값보다 작으면 무시