내리막 길 풀이

완전탐색

- 완전탐색에서 시작

50	45	37	32	30
35	50	40	20	25
30	30	25	17	28
27	24	22	15	10

좌표 0, 0 에서 시작

←↑↓→ 4방향으로 이동 가능
단, 이동은 현재 값보다 낮은 값으로만 이동 가능

```
bruteforce(int y, int x)
    for (int i = 0; i < 4; ++i)
    {
        if (mat[y + dy[i]][x + dx[i]] < mat[y][x])
        {
            bruteforce(y + dy[i], x + dx[i]);
        }
    }
```

```
int dy[4] = { 0, 1, 0, -1 };
int dx[4] = { 1, 0, -1, 0 };
```

- 이동 방향의 값이 현재 위치의 값보다 낮다면 이동
  - mat[y + dy[i]][x + dx[i]] < mat[y][x]

완전탐색

- 만약 현재 위치가 도착 지점이라면

```
bruteforce(int y, int x)
    if (y == M - 1 and x == N - 1)
    {
        path += 1;
        return;
    }
```

기저 사례 : y의 값이 세로 M
x의 값이 가로 N일 때

  - 지나왔던 경로를 통해 이동이 가능한 것
  - 경로 += 1

- 이 방식은
  - 중복된 부분 문제를 다시 계산할 가능성이 큼
  - 한 번만 계산하도록 메모이제이션을 적용

동적 계획법

- 모든 경로에 대해 연산 횟수를 기억할 추가 배열을 정의

50	45	37	32	30
35	50	40	20	25
30	30	25	17	28
27	24	22	15	10

mat

-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1

cache

한번 연산했던 경로는 값이 -1이 아님
cache[y][x]의 값을 return

```
int& ret = cache[y][x];
if (ret != -1)
{
    return ret;
}
```

동적 계획법

- 연산을 이미 했다면 추가적인 연산을 하지 않도록 변경

50	45	37	32	30
35	50	40	20	25
30	30	25	17	28
27	24	22	15	10

mat

-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1

cache

```
for (int i = 0; i < 4; ++i)
{
    if (mat[y + dy[i]][x + dx[i]] < mat[y][x])
    {
        bruteforce(y + dy[i], x + dx[i]);
    }
}
```

```
int sum = 0;
for (int i = 0; i < 4; ++i)
{
    if (mat[y + dy[i]][x + dx[i]] < mat[y][x])
    {
        sum += dp(y + dy[i], x + dx[i]);  x[i]);
    }
}
```

동적 계획법

- 기저 사례 변경

```
if (y == M - 1 and x == N - 1)
{
    return 1;
}
```

기존 코드(pseudo)

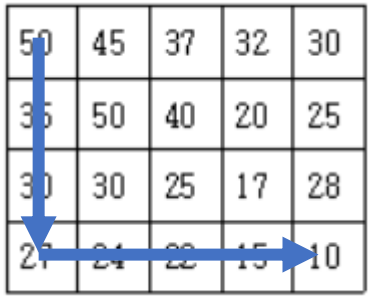
```
bruteforce(int y, int x)

    if (y == M - 1 and x == N - 1)
    {
        path += 1;
        return;
    }

    for (int i = 0; i < 4; ++i)
    {
        if (mat[y + dy[i]][x + dx[i]] < mat[y][x])
        {
            bruteforce(y + dy[i], x + dx[i]);
        }
    }
```

변경된 코드(pseudo)

```
dp(int y, int x)

    if (y == M - 1 and x == N - 1)
    {
        return 1;
    }

    int& ret = cache[y][x];
    if (ret != -1)
    {
        return ret;
    }

    int sum = 0;
    for (int i = 0; i < 4; ++i)
    {
        if (mat[y + dy[i]][x + dx[i]] < mat[y][x])
        {
            sum += dp(y + dy[i], x + dx[i]);
        }
    }
```

예시

- 경로가 존재할 시 해당 경로에 + 1 (-1 는 접근한 경로가 아닌 경우)



mat

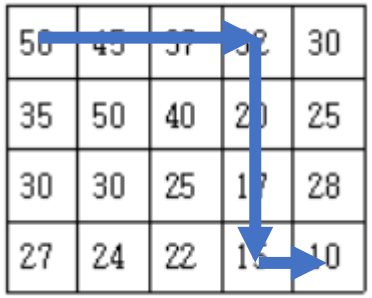
1	-1	-1	-1	-1
1	-1	-1	-1	-1
1	-1	-1	-1	-1
1	1	1	1	-1

cache

도착 지점은 바로 return

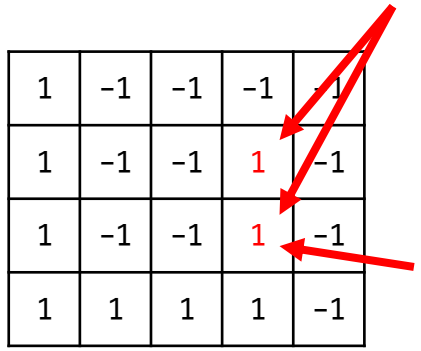
예시

- 다른 경로가 존재하면 다음과 같은 과정을 통해 연산



mat

재귀 호출로 cache[3][3]의 값을 리턴



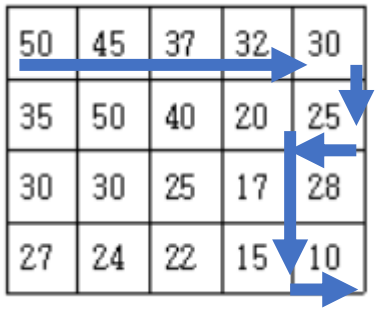
cache

한번 연산했던 경로는 값이 -1이 아님
cache[y][x]의 값을 return

```
int& ret = cache[y][x];
if (ret != -1)
{
    return ret;
}
```

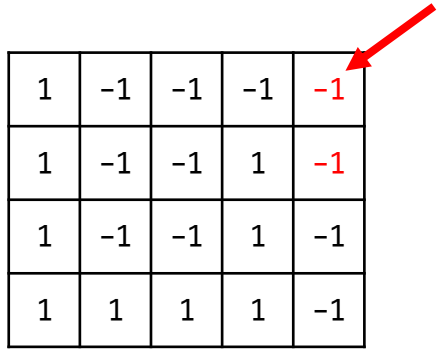
예시

- 다른 경로가 존재하면 다음과 같은 과정을 통해 연산



mat

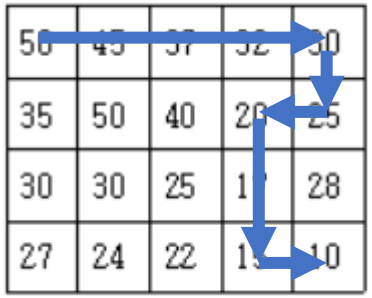
이동이 가능, 부분 문제 발생



cache

예시

- 다른 경로가 존재하면 다음과 같은 과정을 통해 연산



mat

1	-1	-1	-1	1
1	-1	-1	1	1
1	-1	-1	1	-1
1	1	1	1	-1

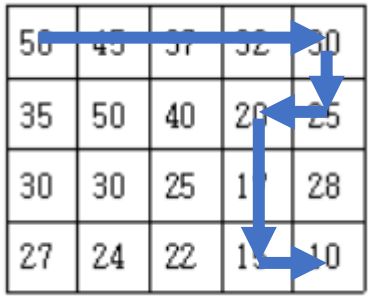
cache

한번 연산했던 경로,
cache[y][x]의 값을 return

예시

- 다른 경로가 존재하면 다음과 같은 과정을 통해 연산

모든 경로를 합침



mat

cache[0][3]은 경로↓, 경로→ 가 존재
가능한 경로만큼 sum에 합산

3	2	2	2	1
1	-1	-1	1	1
1	-1	-1	1	-1
1	1	1	1	-1

cache

```
int sum = 0;
for (int i = 0; i < 4; ++i)
{
    if (mat[y + dy[i]][x + dx[i]] < mat[y][x])
    {
        sum += dp(y + dy[i], x + dx[i]); x[i]);
    }
}
```

- 중복 연산을 줄여 시간 내에 결과 도출 가능