

Im4u 봄방학 캠프

DAY 3; Discrete Optimization Problems #3

Dynamic Programming (I)

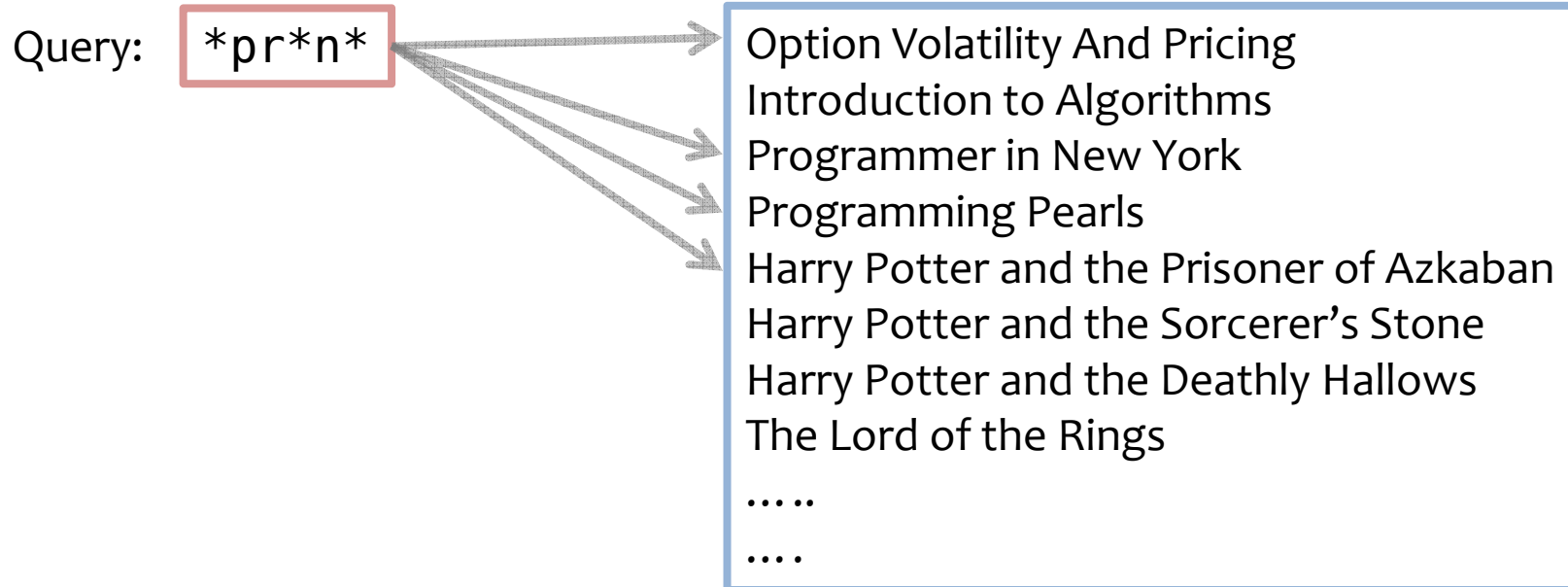
구종만

[jongman@gmail.com](mailto:jongman@gmail.com)

# 오늘 할 얘기

- 프로그래밍 대회에의 꽃, 동적 계획법
- 다양한 패턴들
  - Brute-Force 알고리즘 최적화하기
  - 최적화 문제 (Optimization Problem) 풀기
  - 경우의 수와 확률 문제 풀기
- with a lot of sample problems

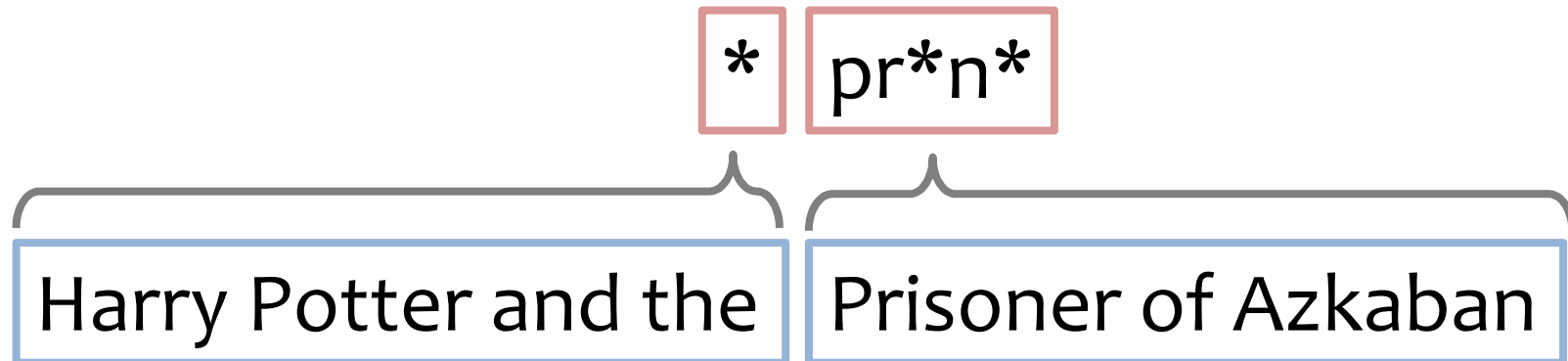
# Wildcards



- ? 는 한 글자에, \* 는 0+개의 글자에 매칭될 수 있다. 주어진 와일드카드에 매칭되는 제목의 수는?
- 제목과 와일드카드의 길이는 200 이하

# Wildcards: A Recursive Algorithm

- string  $s$  와 wildcard  $w$  가 주어졌을 때,  $s$  의  $n$  글자와 나머지,  $w$  의 첫 글자와 나머지로 각각 나눈다.



# Wildcards: A Recursive Algorithm

```
bool isMatched(string title, string wildcard) {  
    if(title.empty() && wildcard.empty()) return _____;  
    if(title.empty()) return _____;  
  
    for(int n = 0; n <= title.size(); ++n)  
        if(matches(wildcard[0], title.substr(0, n)))  
            if(isMatched(_____, _____))  
                return true;  
  
    return false;  
}
```

- 빈칸을 채워 봅시다 ^\_^
  - bool matches(char ch, string str)
  - 이 있다고 가정하고..

# Wildcards: A Recursive Algorithm

```
bool isMatched(string title, string wildcard) {  
    if(title.empty() && wildcard.empty()) return true;  
    if(title.empty()) return isAllStars(wildcard);  
  
    for(int n = 0; n <= title.size(); ++n)  
        if(matches(wildcard[0], title.substr(0, n)))  
            if(isMatched(title.substr(n), wildcard.substr(1)))  
                return true;  
  
    return false;  
}
```

- 빈칸을 채웠습니다 ^\_^

# Can We Brute-Force It?

\* \* \* \* \*

zzzzzzzzzzzzzzzzzzzz...zzzzzzzzzzzzzzza

- ...
- 물론, ad-hoc 최적화를 할 수 있지만, 논외로 두죠

# Optimizing Function Calls



- 빨간 경우의 답과 녹색 경우의 답은 다른가?
- 입력은 다른가?



## 같은 입력 == 같은 답

- isMatched("a", "z") 는 몇 번이나 호출될까?
- 이거 한 번만 계산하면 안되나요?

# Caching Function Values

```
string title, wildcard;  
int cache[100][100];  
// title.substr(t) and wildcard.substr(w) can be matched?  
bool isMatched(int t, int w) {  
    if(t == title.size() && w == wildcard.size()) return 1;  
    if(t == title.size()) return isAllStars(wildcard.substr(w));  
    if(cache[t][w] != -1) return cache[t][w];  
    for(int n = 0; n <= title.size() - t; ++n)  
        if(matches(wildcard[w], title.substr(t, n)))  
            if(isMatched(t+n, w))  
                return cache[t][w] = 1;  
    return cache[t][w] = 0;  
}
```

- `cache[ ][ ]` 는 모두 -1 로 초기화
- 특정 입력에 대한 계산 값을 테이블에 저장해 둬

# How Much Speedup?

- 최적화 이전

- 제목의 길이  $n$ , 와일드카드 길이  $m$  에 대해  $\text{bino}(n+m, m-1)$
- $\text{bino}(200, 99) \approx 9.05 \cdot 10^{58}$

- 최적화 이후

- 모든  $t, w$  의 조합에 대해  $O(n)$
- $t, w$  의 조합은  $O(n^2)$  개수 있음
- $O(n^3)$

# Memoization

- 재귀 호출로 문제를 푸는 과정에서, 한 번 계산한 결과 값을 두 번 계산하지 않도록 저장해 두는 기법
- (not memorization)
- Recurrence 만 있으면 언제든지 쓸 수 있다
  - **Recurrence** relation: 값이 자기 자신에 대해 재귀적으로 정의되는 함수

# Recurrence

- Factorial

$$f(0) = 1, f(n) = n \cdot f(n-1)$$

- Fibonacci Numbers

$$fib(0) = fib(1) = 1, fib(n) = fib(n-1) + fib(n-2)$$

- 머지 소트의 시간 복잡도

$$T(0) = T(1) = 1, T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$$

- 관계식과 base case 로 구성됨

# Solving A Recurrence

- 여러 경우, Recurrence 를 closed form 으로 바꿀 수도 있다.
  - Ex)  $T(0) = T(1) = 1, T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n \Rightarrow T(n) = O(n \lg n)$
- 하지만 못 푸는 경우가 더 많아요
  - 이런 경우에는 직접 각 인스턴스를 계산하는 수밖에 없죠
- 점화식을 어떻게 잡느냐에 따라 동적 계획법을 할 수 있느냐 없느냐가 갈리기도 합니다

# 동적 계획법 (Dynamic Programming)

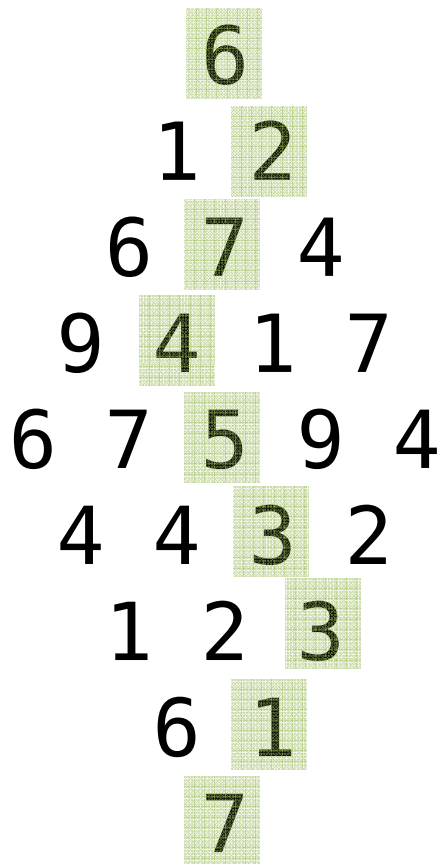
- 재귀적 문제 정의 + Memoization
  - 문제를 재귀적으로 정의한 뒤, 여러 번 계산되는 값을 저장해 두어 알고리즘의 속도를 높인다
- 컴퓨터 과학 사에 최악의 미스네이밍이라고 강력히 주장중
  - Dynamic이란 말도 당최 왜 나왔는지 모를 뿐더러...
  - Programming은 프로그래밍과 관련 없습니다 (동적 프로그래밍이라고 쓰지 맙시다)

# 동적 계획법의 사용처

- 처음에는 최적화 문제 (optimization problem) 들을 풀기 위해 고안
- 하지만
  - 완전 탐색을 최적화 한다거나
  - 확률/경우의 수 문제를 풀 때도 유용하게 사용
- 문제가 재귀적으로 정의되느냐! 가 포인트~!
  - 실제로 그렇지 않은 문제인데도 재귀적으로 바꿔버릴 수도 있다: 모델링의 힘



# Best Path On A Diamond



- 맨 윗칸에서 맨 아랫칸으로 내려오는 방법 중 숫자의 합을 최대화하는 방법은?
- 가운데 줄의 길이 =  $N$
- $N \leq 100$

# Can We Brute-Force It?

- 당근 안되겠죠
- 경로의 수를 세고 싶지만 대충 넘어갑시다 어차피 안될거 아는데. (이 슬라이드를 만들고 있는 지금은 목요일 새벽 5시)

# Best Path On A Diamond: Recursive Sol

|   |   |   |   |   |
|---|---|---|---|---|
| 6 | s | s | s | s |
| 1 | 2 | s | s | s |
| 6 | 7 | 4 | s | s |
| 9 | 4 | 1 | 7 | s |
| 6 | 7 | 5 | 9 | 4 |
| 4 | 4 | 3 | 2 | s |
| 1 | 2 | 3 | s | s |
| 6 | 1 | s | s | s |
| 7 | s | s | s | s |

```
int n, dia[201][101];

int bestpath(int y, int x) {
    if(x == n) return s;
    if(y == n*2-2) return dia[y][x];
    int delta = (y < n ? 1 : -1);
    return max(bestpath(y+1, x), bestpath(y+1, x+delta))
        + dia[y][x];
}

cout << bestpath(0, 0) << endl;
```

- 가장 무식한 완전 탐색에서부터 시작합니다
- s 는 무지 작은 수 (-987654321 쯤) 이라고 둡시다

# Introduce Memoization

```
int n, dia[201][101], cache[201][101];
int bestpath(int y, int x) {
    if(x == n) return s;
    if(y == n*2-2) return dia[y][x];
    int delta = (y < n ? 1 : -1);
    if(cache[y][x] != -1) return cache[y][x];
    return cache[y][x] = max(bestpath(y+1, x), bestpath(y+1, x+delta))
        + dia[y][x];
}
```

- Boom! 동적 계획법 알고리즘 탄생

$$C(y, x) = \max(C(y+1, x+1), C(y+1, x)) + dia[y, x]$$

- 시간 복잡도는 얼마일까요?

# 참 쉽죠?



# 주의, 주의!

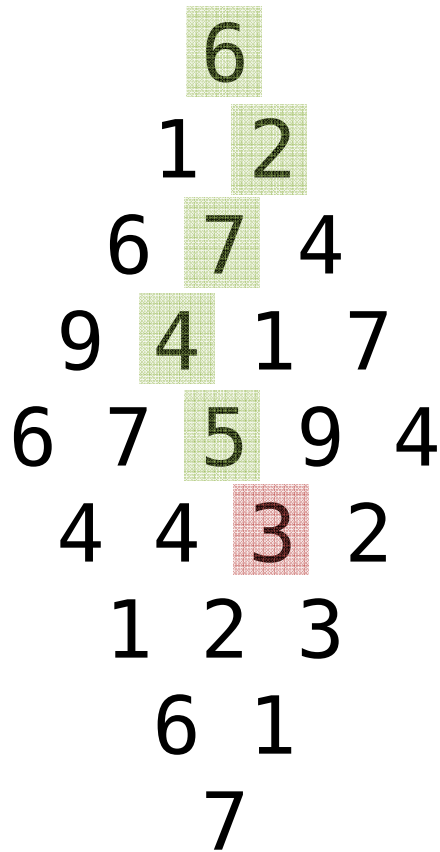
```
int bestpath(int y, int x, int current_path = 0) {  
    if(x == n) return s;  
    if(y == n*2-2) return max(max_path, current_path + dia[y][x]);  
    int delta = (y < n ? 1 : -1);  
    return bestpath(y+1, x, current_path + dia[y][x])  
        + bestpath(y+1, x+delta, current_path + dia[y][x]);  
}
```

- 이 알고리즘도 모든 경로를 다 보지만, 동적 계획법으로 바꿀 수는 없어요 (혹은 공간 복잡도가 무지 커지거나)

$$C(y, x, cp) = \max(C(y+1, x+\Delta, cp + dia[y, x]), C(y+1, x, cp + dia[y, x]))$$

- 부분 문제의 답은 이전까지의 답과 독립적이어야

## 그 말인즉슨..



- “3부터 끝까지 가는 경로 중 가장 합이 큰 것은?”
- “이러이러한 경로를 거쳐서 3 까지 왔다. 이 뒤를 잇는 경로 중 가장 합이 큰 것은?”

# Quantization

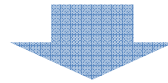


- 8개의 색깔만으로 재현한 그림
- Downsampling



# Quantization

1,744,755,4,897,902,890,6,777



4,757,757,4,899,899,899,4,757

- [1,1000] 구간의 정수 수열을,  $n$ 종류의 숫자만 사용하도록 다운샘플링하고 싶다.
  - $n \leq 10$ , 수열의 길이  $\leq 100$
  - 오차 제곱의 합을 최소화하는 방법은?

# 어떻게 재귀로 풀지?

- 망상A

- $1000^n$  으로 모든 공역을 만든 다음 가장 가까운 수로 매칭하자구..

- 망상B

- 암말 안하고 있음 옆사람이(혹은 종만이가) 풀어주겠지..

- 망상C

- 아 배고프다...

# Key Observation

- $6 \Rightarrow 10$  으로 바꿨는데  $11 \Rightarrow 9$  로 바꾸는 건 사상 최대의 \_\_\_\_\_ 이다.
- 수열을 이루는 수들의 \_\_\_\_\_ 는 상관없다.

# Key Observation

- 6 => 10 으로 바꿨는데 11 => 9 로 바꾸는 건 사상 최대의 \_개삽질\_ 이다.
- 수열을 이루는 수들의 \_순서\_ 는 상관이 없다.



## 따라~서

- 입력에 주어지는 수들을 정렬하고

1, 4, 6, 744, 755, 777, 890, 897, 902

- 이들을 적절히 n 묶음으로 나눈 후에, 각 묶음에서의 오류를 최소화하면 되지 않겠느냐~

1, 4, 6

5

744, 755, 777

760



890, 897, 902

896

# Recurrence

1, 4, 6, 744, 755, 777, 890, 897, 902



here=3, groups=2 (  ,  )

- here 부터 끝까지의 숫자는 아직 매핑이 안됐다
- 만들 수 있는 묶음의 개수 groups
- 첫 번째 묶음과 나머지로 나눈다면?

# 주어진 수열을 쪼개는 문제가 되는데..

```
int n;  
int sequence[100];  
  
int quantize(int here, int groups) {  
    if(here == n) return 0;  
    if(groups == 0) return INFINITE;  
    int ret = INFINITE;  
    for(int m = 1; here+m < n; ++m) {  
        int cand = solve(here, here+m-1) + quantize(here+m, groups-1);  
        ret = min(cand, ret);  
    }  
    return ret;  
}
```

- 주어진 수열을 첫 그룹과 나머지로 나눈다

$$C(\text{here}, \text{groups}) = \min_{m=0}^{\text{here}+m < n} [\text{solve}(\text{here}, \text{here} + m - 1) + C(\text{here} + m, \text{groups} - 1)]$$

- Memoization 은 추가 안 할게요..

# Coin Change

- JM나라에서는 1백원짜리, 3백원짜리, 5백원짜리, 9백원짜리, 2천원짜리 동전을 쓰고 있다 (거짓말)
- 3600원을 거슬러 줘야 하는데, 몇 가지 방법으로 거슬러 줄 수 있을까?
  - 1백원짜리 36개
  - 3백원짜리 12개
  - 1백원짜리 3개 + 3백원짜리 11개 + ...



# An Iterative Solution

```
int amt = 3600;
int ret = 0;
for(int one = 0; one*100 <= amt; ++one)
for(int three = 0; one*100 + three*300 <= amt; ++three)
for(int five = 0; one*100 + three*300 + five*500 <= amt; ++five)
for(int nine = 0; one*100 + three*300 + five*500 + nine*900 <= amt; ++nine)
for(int twenty = 0; one*100 + three*300 + five*500 + nine*900 + twenty*2000
<= amt; ++twenty)
    if(one*100 + three*300 + five*500 + nine*900 + twenty*2000 == amt)
        ++ret;
```

- 우와, 정말 무식하다

# A Recursive Solution

```
const int D = 5;
const int denom[D] = { 100, 300, 500, 900, 2000 };

int waysChange(int amt, int idx) {
    if(amt == 0) return 1;
    if(idx == D) return 0;
    int ret = 0;
    for(int thisCoin = 0; thisCoin * denom[idx] <= amt; ++thisCoin)
        ret += waysChange(amt - thisCoin * denom[idx], idx+1);
    return ret;
}
```

- Memoization 도입은 여전히 안 합니다 히히

# The Recurrence

$$\textit{denom} = \{100, 300, 500, 900, 2000\}$$

$$C(\textit{amt}, \textit{idx}) = \text{Ways of decomposing } \textit{amt} \text{ into } \textit{denom}[\textit{idx} \sim n-1]$$

$$= 1 \text{ (if } \textit{amt} = 0)$$

$$= \sum_{n=0}^{\textit{amt} / \textit{denom}[\textit{idx}]} C(\textit{amt} - \textit{denom}[\textit{idx}] \cdot n, \textit{idx} + 1) \quad (\text{otherwise})$$

# Diamonds

```
.....#.....
. #.#####. ....
.########.###. ....
. #.#####.#####. ....
.....#.....#####. ....
.....#####.###.#. ....
.....#####. ....
```

- Diamonds 는 가로 길이  $n$ , 세로 길이  $n$ 의 마름모꼴이다.
- 주어진  $(n, m)$  크기의 격자에서, 찾을 수 있는 다이아몬드의 개수는?

# Diamonds

- 다이아몬드 개수 세기
  - 각 칸에 대해, 주어진 칸을 맨 아래 칸으로 하는 다이아몬드의 개수를 세자
- Key Observation
  - 크기 5인 다이아몬드는 언제나 같은 칸에서 시작하는 크기 3 다이아몬드를 포함한다
  - 따라서, 각 칸을 맨 아래 칸으로 하는 다이아몬드의 최대 크기  $\text{maxSize}[y,x]$  를 구하자

```
.....#.....
..#.#####..
.########...
..#.######.
.....#.....
```

# Diamonds: Recurrence

$mxSize(y, x) = \text{max size of a diamond ending at } (y, x)$

```
.....#.....
.....###.....
.....#####.....
.....#####.....
.....#####.....
.....###.....
.....#.....
```

- 오늘의 마지막 문제니까 직접 풀어봅시다!

# Key Observation

- 크기  $n$ 인 다이아몬드는 사실  $n-2$ 인 다이아몬드 3개와 2칸으로 구성할 수 있다

```
...#...#...#...#...
...###...###...###...
...#####...#####...
...#####...#####...
...#####...#####...
...#####...#####...
...###...###...###...
...#...#...#...#...
```

- 크기가  $n$  이려면,  $(y-1, x-1)$ ,  $(y-1, x+1)$ ,  $(y-2, x)$  에 각각 크기  $n-2$ 인 다이아몬드가 있어야 한다

# Recurrence

$mxSize(y, x) = \text{max size of a diamond ending at } (y, x)$

$= 0$  (if  $cell[y, x] == '.'$ )

$= 1$  (if  $cell[y, x] == '#'$  and  $cell[y - 1, x] == '.'$ )

$= \max(mxSize(y - 1, x - 1), mxSize(y - 1, x + 1), mxSize(y - 2, x)) + 2$

- (max 가 0 일 경우 예외 처리가 필요합니다)
- Memoization 답은 간단하게 유도할 수 있다!



# Recursive Dynamic Programming

```
int n, m, cache[101][101];
char grid[101][101];

int maxSize(int y, int x) {
    if(grid[y][x] == '.') return 0;
    if(y < 2 || x == 0 || x+1 == m || grid[y-1][x] == '.') return 1;

    if(cache[y][x] != -1) return cache[y][x];

    int p = max(maxSize(y-2, x), mx(maxSize(y-1, x-1), maxSize(y-1, x+1)));

    if(p == 0 || grid[y-1][x] == '.') return cache[y][x] = 1;
    return cache[y][x] = p + 2;
}
```

- 하지만 꼭 재귀호출을 써야 할까?

# Iterative Dynamic Programming

```
int n, m, C[101][101];
char grid[101][101];
for(int y = 0; y < n; ++y)
    for(int x = 0; x < m; ++x) {
        if(grid[y][x] == '.')
            C[y][x] = 0;
        else if(x > 0 && x+1 < m && y >= 2 && grid[y-1][x] == '#') {
            int p = max(C[y-2][x], max(C[y-1][x-1], C[y-1][x+1]));
            if(p == 0)
                C[y][x] = 1;
            else
                C[y][x] = p + 2;
        }
    }
```

- `mxSize()` 는 `y` 가 더 작은 칸들만을 참조한다
- 특정 순서로 값을 계산해도 될 경우, 꼭 재귀호출을 쓸 필요는 없다

# Memoization vs. Iterative

## ■ Memoization

- 좀더 느리다
- 스택을 사용한다
- 연산 순서에 대해 고민할 필요가 없다
- 코드가 좀더 직관적이다
- 모든 부분문제를 계산할 필요 없을 경우 더 빠르다

## ■ Iterative

- 좀더 빠르다
- 스택을 사용X
- 연산 순서에 대해 고민해야 한다
- 코드가 좀 더 복잡해진다
- 모든 부분문제를 계산해야 할 경우 더 빠르다

# Lessons Learned

- 동적 계획법은 재귀적으로 정의되는 문제에는 어디나 사용될 수 있다
  - 완전 탐색 시간 단축
  - Optimization Problem 의 해결
  - 경우의 수와 확률 문제
- Recurrence (or 점화식)
  - 문제의 답을 자신에 대해 재귀적으로 정의한다
  - 각 입력에 대한 답을 하나의 부분문제라고 부른다

# Following Next Day ...

- 좀더 복잡한 주제들
  - Theoretical Foundations
  - 더 어려운 Optimization/Counting 문제들
  - 계산 게임 (Computational Games) with DP
  - DP with Exponential State Space
  - 최적화 문제의 답안 생성하기 etc.