2️⃣

알고리즘 : 동적 프로그래밍

동적 프로그래밍 (dynamic programming)

최적부분 구조를 가지며 재귀적으로 구현했을 때 중복 호출로 심각한 비효율이 발생하는 문제의 해결에 적합한 기법이다.

동적 프로그래밍을 사용하는 문제

관계중심으로 파악해 문제를 간명하게 볼 수 있지만, 심한 중복 호출이 일어나는 경우 재귀적 해법이 아닌 동적 프로그래밍으로 해결할 수 있다.
예를 들어 큰 문제의 해답에 작은 문제의 해답이 포함되어 있는 문제를 생각할 수 있다. 관계중심으로 파악하여 문제를 간명하게 정의 가능하다. (예로 펙토리얼을 생각할 수 있다.)

재귀 알고리즘을 사용해 바람직한 구현이 가능할 수 있지만, 심한 중복 호출이 일어나서 재귀적 구현이 매우 비효율적인 경우 동적 프로그래밍으로 해결할 수 있다.

재귀적 해법이 바람직한 예

재귀적으로 구현해도 심한 중복 호출이 발생하지 않는다.

- 퀵정렬, 병합정렬
- factorial(팩토리얼) 구하기
- 그래프의 깊이 우선 탐색 (DFS)

재귀적 해법이 치명적인 예

재귀적으로 구현하면 심한 중복 호출이 발생하므로 동적 프로그래밍이 필요하다.

- 피보나치 수 구하기
- 행렬 곱셈 최적 순서 구하기

동적 프로그래밍의 적용 조건

- 최적 부분구조(optimal substructure)
  큰 문제의 최적 솔루션에 작은 문제의 최적 솔루션이 포함된다.

- 재귀 호출시 중복(overlapping recursive calls)
  재귀적 해법으로 풀면 같은 문제에 대한 재귀 호출이 심하게 중복된다.

> 이 두가지 모두 해당되면 동적 프로그래밍을 사용해야 된다.

ex) 피보나치 수열은 두가지 모두 해당되므로 동적 프로그래밍을 사용하기에 적합하다.
ex) 팩토리얼은 두가지 모두 해당되지 않으므로 동적 프로그래밍을 사용하기에 부적합하다.

동적 프로그래밍 알고리즘 : Top-down 방식

하향식 동적프로그래밍(메모하기 방식의 동적 프로그래밍)은 재귀적으로 구현하되 함수의 앞부분에서 이미 해결한 적이 있는 문제인지 체크하여 이미 해결한 문제이면 함수를 더 이상 진행하지 않고 테이블에 있는 해를 리턴 한다.

구현

재귀적으로 구현하되, 호출된 적이 있으면 배열 f[]에 메모 해 두어(memoization = 메모리에 저장) 중복 호출 문제 해결할 수 있다. 즉, 함수의 앞부분에 이미 해결한 적이 있는 문제인지를 체크하는 부분을 두고, 이미 한 번 해결한 문제이면 함수를 더 이상 진행하지 않고 테이블에 있는 해를 리턴한다.

```
fib(n){
  if (f[n] ≠ 0) then return f[n];   // 호출된 적이 있으면
    else {
      if (n = 1 or n = 2)
      then f[n] ← 1;
    else f[n] ← fib(n-1) + fib(n-2);
    return f[n];
    }
}
```

동적 프로그래밍 알고리즘 : Bottom-up 방식

상향식 동적 프로그래밍은 작은 문제의 해부터 테이블에 저장해가면서 이들을 이용해 큰 문제들의 해를 구해나간다.
작은 문제의 해부터 테이블에 저장해나가면서 이들을 이용해 큰 문제들의 해를 구해나간다.
Top-down보다 흔하게 사용되는 방식이다.

구현

```
fibonacci(n) {
  f[1] <- 1;
    f[2] <- 1;

    for i <- 3 to n
      f[i] <- f[i-1] + f[i-2];
    return f[n];
}
```

=> 시간복잡도는 **Θ(n)**으로 선형시간이 걸린다.

동적 프로그래밍 예

행렬 경로 문제

양수 원소들로 구성된 n X n 행렬이 주어지고, 행렬의 좌상단에서 시작해 우하단까지 이동한다.

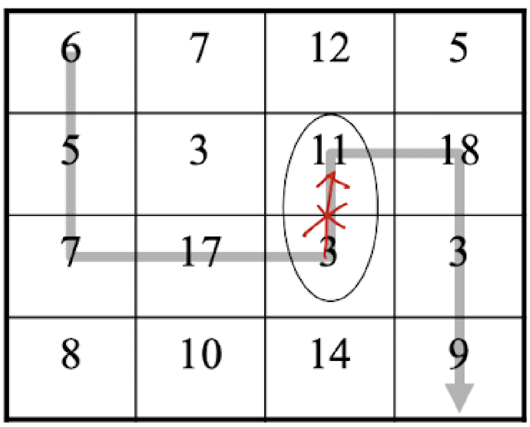
목표

- 행렬의 좌상단에서 시작해 우하단까지 이동하되, 방문한 칸에 있는 수들을 더한 값이 최대가 되도록 한다.

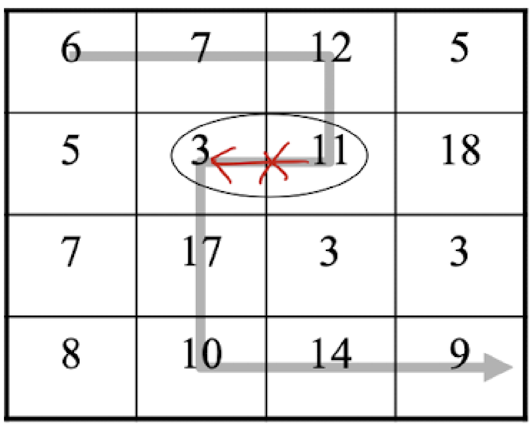
이동 방법

- 오른쪽이나 아래로만 이동할 수 있다.
- 왼쪽, 위쪽, 대각선 이동은 허용하지 않는다.

잘못된 이동 방법 예)

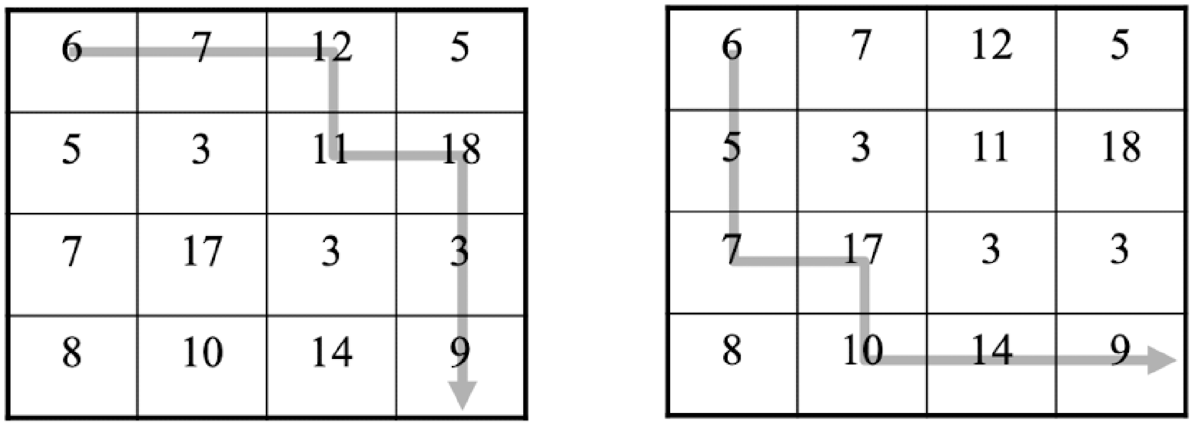


불법 이동 (상향)



불법 이동 (좌향)

올바른 이동 방법 예)



최적 부분구조의 재귀적 관계

cij는 (1, 1)에서 (i, j)에 이르는 최고점수
mij는 행렬의 원소 (i, j)의 값

```
        0                            if i = 0 or j = 0
cij =
        mij + max{ci,j-1, ci-1,j}    otherwise
```

최종적으로는 cn,n이 답이 된다.

행렬 경로 문제 - Recursive Algorithm

(1, 1)에서 (i, j)에 이르는 최고점수

```
matrixPath(i, j) {
  if (i = 0 or j = 0) then return 0;
  else return (mij + max(matrixPath(i-1, j), matrixPath(i, j-1)));
}
```

행렬 경로 문제 - 동적 프로그래밍 알고리즘

(1, 1)에서 (i, j)에 이르는 최고점수

```
matrixPath(i, j) {
  for i ← 0 to n
      c[i, 0] ← 0;
  for j ← 1 to n
      c[0, j] ← 0;
```

```
  for i ← 1 to n
    for j ← 1 to n
      c[i, j] ← mij + max(c[i-1, j], c[i, j-1]);
    return c[n, n];
}
```

=> 시간복잡도는 **Θ(n2)**이다. 행렬의 원소 수에 대해서는 선형시간이 소요된다.

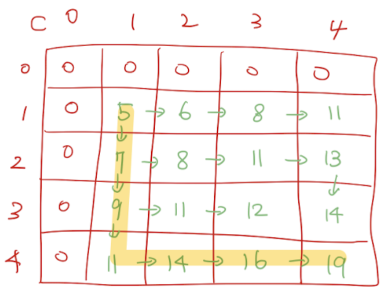
간단한 예제

Q. 다음과 같은 4 x 4 크기의 행렬 경로 문제를 푸는 과정 과 경로의 최대 점수를 구하시오.

4 x 4 행렬 m

i \ j	1	2	3	4
1	5	1	2	3
2	2	1	3	2
3	2	2	1	1
4	2	3	2	3

A.



최대 점수 = 19

돌 놓기 문제

3 x n 테이블의 각 칸에 숫자(양수 또는 음수)가 기록되어 있고, 이 테이블의 칸에 돌을 놓는다.

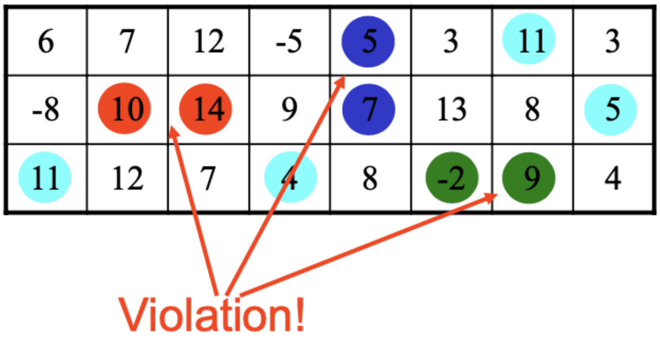
목표

- 돌이 놓인 자리에 있는 수의 합이 최대가 되도록 돌을 놓는다.

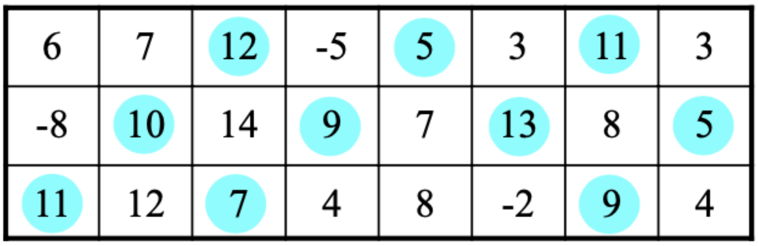
이동 방법

- 가로나 세로로 인접한 두 칸에 동시에 돌을 놓을 수 없다.
- 각 열에는 적어도 하나의 조약돌을 놓아야 한다.

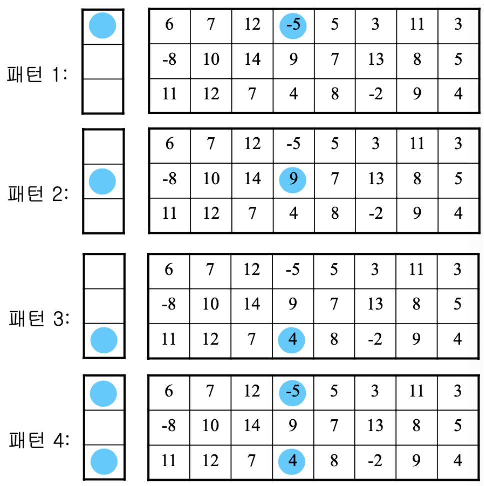
잘못된 방법 예)



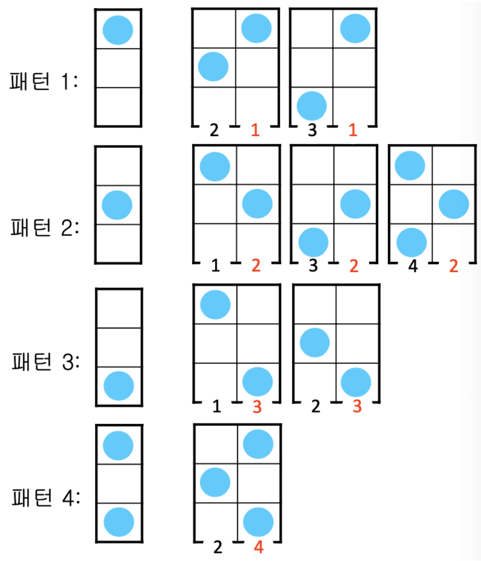
올바른 방법 예)



가능한 패턴



서로 양립할 수 있는 패턴



패턴 1 : 패턴 2, 3과 양립할 수 있다.
패턴 2 : 패턴 1, 3, 4와 양립할 수 있다.
패턴 3 : 패턴 1, 2와 양립할 수 있다.
패턴 4 : 패턴 2와 양립할 수 있다.

최적 부분구조의 재귀적 관계

cip : i열이 패턴p로 놓일 때의 최고점수
wip : i열이 패턴p로 놓일 때 i열에 돌이 놓은 곳의 점수 합
q : p와 양립하는 패턴

```
        w1p                     if i = 1
cij =
        max{ci-1,q} + wip       if i > 1
```

최종적으로는 cn1, cn2 , cn3, cn4 중 가장 큰 것이 답이다.

돌 놓기 문제 - Recursive Algorithm

wip : i열이 패턴 p로 놓일 때 i열에 돌이 놓인 곳의 점수 합. p∈{1, 2, 3, 4}

```
// i열이 패턴p로 놓일 때의 i열까지의 최대 점수 합 구하기
pebble(i, p) {
    if (i = 1)
      then return w1p;
    else {
      max ← -∞ ;
       for q ← 1 to 4 {
        if (패턴 q가 패턴 p와 양립)
          then {
```

```
                tmp ← pebble(i -1, q) ;
                if (tmp > max) then max ← tmp ; }
            }
          return (max + wi p ) ;
    }
}

// n 열까지 돌을 놓은 방법 중 최대 점수 합 구하기
pebbleSum(n) {
  return max { pebble(n, p) } ;
    // p = 1, 2, 3, 4
}
```

pebble(n, 1), ..., pebble(n, 4) 중 최대값이 돌 놓기 문제의 최종 답이다.

돌 놓기 문제 - 동적 프로그래밍 알고리즘

DP의 요건에 만족해야된다.

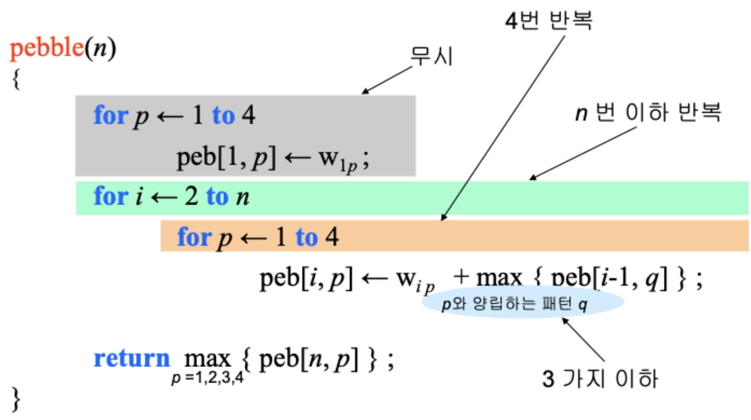
- Optimal substructure
  pebble(i, .)에 pebble(i-1, .)이 포함된다.
  즉, 큰 문제의 최적 솔루션에 작은 문제의 최적 솔루션이 포함된다.
- Overlapping recursive calls
  재귀적 알고리즘에 중복 호출 심하다.

```
pebble(n) {
  for p ← 1 to 4
  peb[1, p] ← w1p ;

    for i ← 2 to n
    for p ← 1 to 4
      peb[i, p] ← wip + max { peb[i-1, q] } ; // p와 양립하는 패턴 q
  return max { peb[n, p] } ;  // p =1,2,3,4
}
```

=> 시간복잡도는 Θ(n)이다.

+) 복잡도 분석



간단한 예제

Q. 다음과 같은 3 x 8 테이블에 대한 돌 놓기 문제에서 최대 점수를 구하시오.

3 x 8 테이블

i	1	2	3	4	5	6	7	8
	1	6	12	-5	5	3	9	2
	-14	9	14	9	7	13	6	4
	6	11	7	4	8	-2	7	3

A.

Peb	1	2	3	4	5	6	7	8
1	1	6+P3 12	12+P2 28	-5+P2 21	5+P2 49	3+P3 55	9+P2 79	2+P3 79
2	-14	9+P4 16	14+P1 26	9+P4 44	7+P3 39	13+P4 70	6+P1 61	4+P4 90
3	6	11+P1 12	7+P2 23	4+P1 32	8+P2 52	-2+P1 47	7+P2 77	3+P1 82
4	7	17+P2 3	19+P2 35	-1+P2 25	13+P2 57	1+P2 40	16+P2 86	5+P2 66

최대 점수 = 90

최장 공통 부분 순서 (LCS)

두 문자열에 공통적으로 들어있는 공통 부분 순서중 가장 긴 것을 찾는다.

부분순서 (subsequence)

문자열 abcd의 부분순서(subsequence)

a, ..., abc, ..., acd, ..., abcd

- 예시
  [bcdb]는 문자열 [abcbdab]의 subsequence

공통 부분 순서 (common subsequence)

문자열 abcd와 aabbdd의 공통 부분 순서(common subsequence)
a, ..., ad, ..., abd

- 예시
  [bca]는 문자열 [abcbdab]와 [bdcaba]의 common subsequence

최장 공통 부분 순서(LCS: longest common subsequence)

LSC란 Common subsequence들 중 가장 긴 것이다.

문자열 abcd와 aabbdd의 최장 공통 부분 순서(LCS)
abd

- 예시
  [bcba]는 string [abcbdab]와 [bdcaba]의 LCS

최적 부분구조의 재귀적 관계

두 문자열 Xm = [x1x2 ... xm]과 Yn = [y1y2 ... yn]에 대해

- xm = yn : Xm과 Yn의 LCS의 길이는 Xm-1과 Yn-1의 LCS의 길이보다 1이 크다.
- xm ≠ yn : Xm과 Yn의 LCS의 길이는 Xm과 Yn-1의 LCS의 길이와 Xm-1과 Yn의 LCS의 길이 중 큰 것과 같다.

cij : 두 문자열 Xi = [x1x2 ... xi]과 Yj = [y1y2 ... yj]의 LCS 길이

```
        0                              if i = 0 or j = 0
cij =   ci-1,j-1 + 1                   if i, j > 0 and xi = yj
        max{ci-1,j, ci,j-1} + wip      if i, j > 0 and xi ≠ yj
```

최종적으로 cmn이 답이다.

최장 공통 부분순서 - Recursive Algorithm

두 문자열 Xm과 Yn의 LCS 길이를 구한다.

```
LCS(m, n) {
  if (m = 0 or n = 0) then return 0;
  else if (xm= yn) then return LCS(m-1, n-1) + 1;
    else return max(LCS(m-1, n), LCS(m, n-1));
}
```

=> 엄청난 중복 호출이 발생한다.

최장 공통 부분순서 - 동적 프로그래밍 알고리즘

두 문자열 Xm과 Yn의 LCS 길이를 구한다.
부분 문제의 답을 배열C에 저장한다.
C[i, j] : Xi과 Yj의 LCS 길이를 저장한다.

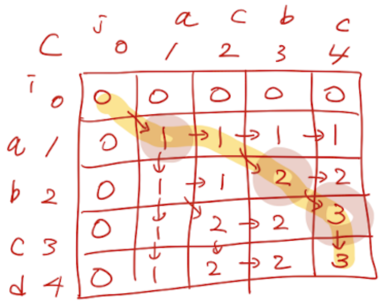
```
LCS(m, n) {
  for i ← 0 to m
      C[i, 0] ← 0;
    for j ← 0 to n
      C[0, j] ← 0;
    for i ← 1 to m
    for j ← 1 to n
          if (xi= yj)
              C[i, j] ← C[i-1, j-1] + 1;
      else
              C[i, j] ← max(C[i-1, j], C[i, j-1]);
     return C[m, n];
}
```

=> 시간복잡도는 Θ(mn)이다.

간단한 예제

Q. 다음 두 문자열 X4 =abcd와 Y4 =acbc의 LCS길이를 구하시오.

A.



LCS = abc
길이 = 3