동적 프로그래밍

HaRim Jung, Ph.D.
Visiting Professor / Senior Researcher
SKKU Institute for Convergence / Convergence Research Institute
Sungkyunkwan University, Korea

동적 프로그래밍 개요 (1/8)

❑ 동적 프로그래밍(dynamic programming)이란?

동적 프로그래밍 사용의 동기(motivation): 어떤 경우에 재사용이 필요할까?

- 복잡한 문제(problem)를 (1) 간단한 부분 문제(sub-problem)들로 분할하고, (2) 각 부분 문제에 대한 부분 해를 찾은 후 (필요 시) 재사용을 위해 **기록**(**tabulation** or memoization)하는 과정을 반복하여 모든 부분 해들을 찾고, (3) 이들을 통합하여 원래 문제의 해결책(solution)을 찾는 전략(**vs. 분할 정복 전략**)
  - 프로그래밍(programming)의 의미는 **과정**(**course**)**을 테이블에 기록**(**table-filling**), 동적(dynamic)의 의미는 **테이블이 갱신**(**update**)된다는 것

수학자인 리처드 벨만이 1940년대에 사용하던 용어였다. 1953년, 벨만은 큰 문제 안에 작은 문제가 중첩되어있는 문제를 해결하는 데 사용하는 이 방법을 '동적 계획법'이라 이름붙였다.
벨만은 그의 자서전, '태풍의 눈'에서 다음과 같이 말했다.

> 나는 RAND 코퍼레이션에서 1950년의 가을을 보냈다. 여기에서 내게 주어진 첫 과제는 다단계(multistage) 의사 결정 프로세스에 대해 적절한 용어를 명명하는 것이었다. '동적 계획법'이라는 이름이 어디에서 왔는지 궁금하지 않은가? 1950년대는 내가 수학에 대해 연구하기에는 좋지 못한 시기였다. 우리는 그 때 워싱턴에서 윌슨이라는 사람과 함께 일하고 있었다. 윌슨은 연구라는 것에 대해 굉장히 병적인 공포를 가지고 있었다. 사람들이 그 앞에서 연구에 대해 이야기를 꺼내면 그는 완전히 미치다시피 했다. 그러나 불행히도 RAND는 공군 소속의 회사였고, 윌슨은 그 공군의 간부인 국방 위원장이었다. 그래서 내가 RAND 안에 있었을 때 윌슨을 비롯한 공군들이 내가 수학에 대해 연구하는 것을 보이지 않게 막는다는 것을 알 수 있었다. 처음 올 때는 나는 위의 문제에 대해 '의사 결정 프로세스'라는 이름을 사용했지만, 여기에서 '프로세스(Process)'라는 단어를 사용하는데 여러가지 차질이 생겨버리고 만 것이다. 그래서 나는 사람들이 알지 못하게 '계획법(Programming)'이라는 단어를 붙였다. 또한 나는 이 프로세스가 다단계로 이루어져 있으며, 시가변적(time-varing)이며, '동적(Dynamic)'이다라는 개념(idea)이 전달되길 원했다. 이 단어야말로 내가 연구하는 알고리즘의 성질을 정확하게 짚어내었고, 게다가 윌슨에게도 피해를 입히지 않으며 공군도 이 단어에선 꼬투리를 잡지 못했으니 그야말로 일석이조의 효과를 누린 것이다.

출처: 위키 백과

- 일련의 연속적인 부분 해(local solution or current solution) 선택이 필요한 문제가 주어졌을 때, 선택을 해야 할 순간(step or sub-problem)마다 **기록된** 이전의 모든 선택들을 고려하여 **최적의 부분 해를 선택**함으로써 최종 해(global solution or final solution)를 도출하는 알고리즘

➡ **동적 프로그래밍은 탐욕적 방법과 마찬가지로 최적화 문제 해결에 사용될 수 있는 전략**

동적 프로그래밍 개요 (2/8)

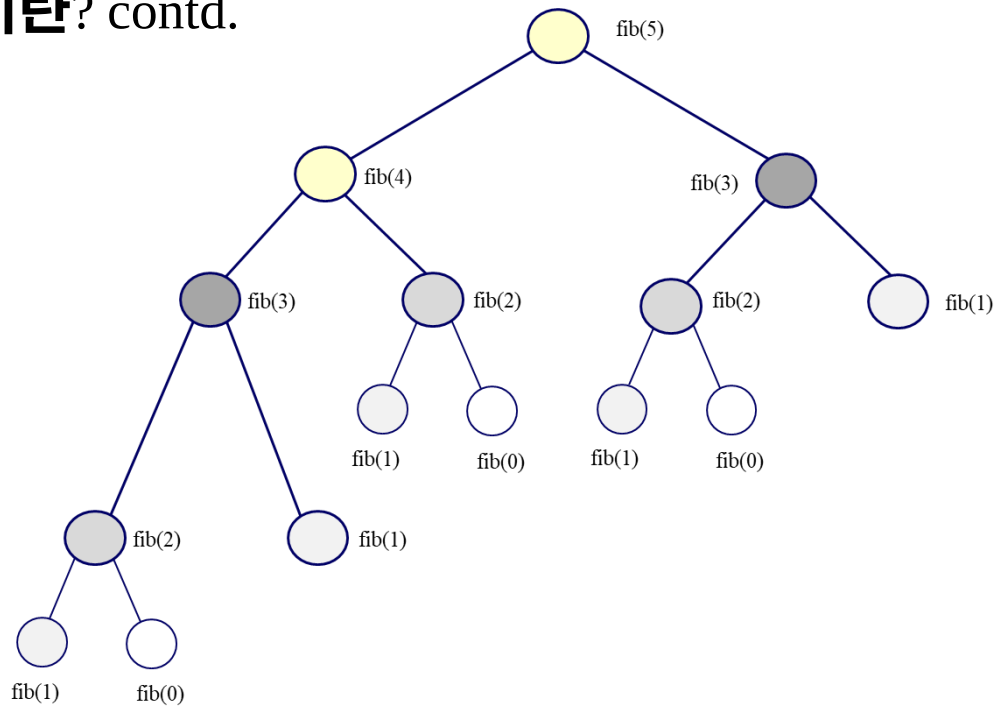
❑ 동적 프로그래밍(dynamic programming)이란? contd.

- **예: n번째 피보나치 수(Fibonacci number) 구하기**
  - 피보나치 수열의 순환식(recurrence: 점화식)

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2}, \text{ for } n \geq 2$$

  - 분할 정복 전략(recursion)

```
1  def fib(n):
2      if n == 0 or n == 1:
3          return n
4      return fib(n-1) + fib(n-2)
5
6  print(fib(5))
```



- 5번째 피보나치 수를 구하는 과정에서 **완벽히 동일한 부분 문제**가 중복(3번째 피보나치 수를 구하는 부분 문제 **fib(3)** 2 번, 2번째 피보나치 수를 구하는 부분 문제 **fib(2)** 3번, 1번째 피보나치 수를 구하는 부분 문제 **fib(1)** 5번, 0번째 피보나치 수를 구하는 부분 문제 **fib(0)** 3번) → **수행 시간: $O(2^n)$**
- 중복된 (완벽히 동일한) 부분 문제 해결 과정을 제거하게
- 된다면 **수행 시간을 선형 시간, i.e., O(n)으로 줄일 수 있음**
- 어떻게 중복된 부분 문제 해결 과정을 제거할 수 있을까?
  - » **부분 문제에 대한 해를 기록**한 후 해당 부분 문제와 완벽히 동일한 부분 문제 해결 시 **기록된 해를 (재)사용**

→ 두 가지 전략: 메모 전략 & 동적 프로그래밍

입력 크기 n에 대한 모든 경우의 수행 시간을 T(n)이라고 하면:
$T(n) = T(n-1) + T(n-2) + 1.$
위의 식을 전개하면:

$$T(n) = T(n-1) + T(n-2) + 1;$$
$$> 2 \times T(n-2);$$ T(n-1) + T(n–2) > T(n-2) + T(n-2) 이므로
$$> 2 \times 2 \times T(n-4);$$ T(n-3) + T(n–4) > T(n-4) + T(n-4) 이므로
$$\cdots$$
$$> 2 \times 2 \times \ldots \times 2 \times T(0) = 2^{\frac{n}{2}}.$$

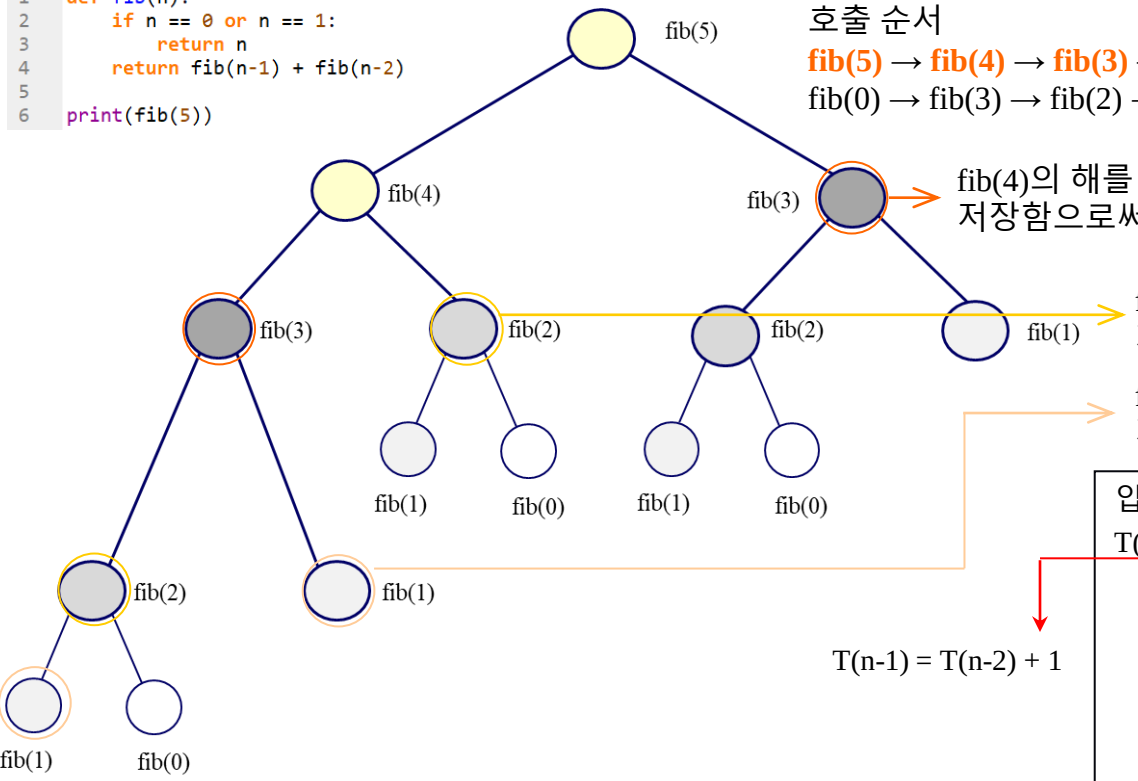
따라서 $T(n) = O(2^n)$.

동적 프로그래밍 개요 (3/8)

❑ 동적 프로그래밍(dynamic programming)이란? contd.

- **예: n번째 피보나치 수(Fibonacci number) 구하기 contd.**
  - **메모 전략(memoization)**
    - 특정 문제를 분할 정복 시 (특정 문제를 재귀 호출을 이용하여 문제의 최종 해를 찾을 시) 특정 부분 문제의 해를 찾게 되면 이를 기록해 놓고(memoization), 완벽히 동일한 문제의 해를 다시 찾아야 할 경우 함수를 호출하지 않고 저장된 해를 재사용하는 전략 NOTE: 분할 정복 및 메모 전략은 하향식(top-down) 문제 해결 방식(호출 순서를 생각해 보면 쉽게 알 수 있음)

```
1  def fib(n):
2      if n == 0 or n == 1:
3          return n
4      return fib(n-1) + fib(n-2)
5
6  print(fib(5))
```



호출 순서

fib(5) → **fib(4)** → **fib(3)** → **fib(2)** → **fib(1)** → **fib(0)** → fib(1) → fib(2) → fib(1) → fib(0) → fib(3) → fib(2) → fib(1) → fib(0) → fib(1)

fib(4)의 해를 찾는 과정에서 fib(3)이 호출되므로 fib(3)의 해 2를 저장함으로써 추후 fib(3)을 호출 시 저장된 결과 2를 사용

fib(3)의 해를 찾는 과정에서 fib(2)가 호출되므로 fib(2)의 해 1을 저장함으로써 추후 fib(2)를 호출 시 저장된 결과 1을사용

fib(2)의 해를 찾는 과정에서 fib(1)이 호출되므로 fib(1)의 해 1을 저장함으로써 추후 fib(1)을 호출 시 저장된 결과 1을사용

T(n-1) = T(n-2) + 1

입력 크기 n에 대한 모든 경우의 수행 시간을 T(n)이라고 하면:

T(n) = T(n-1) + 1; 여기서 1은 기록된 f(n-2)의 결과 접근 시간
= T(n-2) + 2; 여기서 2는 기록된 f(n-2), f(n-3)의 결과 접근 시간
= T(n-3) + 3;
...
= T(1) + n-1;
= T(0) + n.

T(0)은 1이므로, i.e., fib(0)은 한 번 호출이므로 T(n) = O(n)

동적 프로그래밍 개요 (4/8)

❑ 동적 프로그래밍(dynamic programming)이란? contd.

- **예: n번째 피보나치 수(Fibonacci number) 구하기 contd.**
  - **메모 전략을 이용한 피보나치 수 구하기 문제 해결을 위한 fib_memoized 함수 정의**

```
1   cache = {}
2   def fib_memoized(n):
3       if n in cache:
4           return cache[n]
5       if n == 0 or n == 1:
6           result = n
7           cache[n] = result
8           return cache[n]
9       result = fib_memoized(n - 1) + fib_memoized(n - 2)
10      cache[n] = result
11      return result
12
13  print(fib_memoized(7))
```

- **라인 1:** 기록(memo)을 위한 사전(dictionary) cache 선언(키는 n, 값은 피보나치 수)
- **라인 3-4:** 만약 fib_memoized(n)을 이미 호출하여 기록하였다면 해당 기록 값을 반환
- **라인 5-8**: 만약 n이 0 또는 1이라면 변수 result에 n의 값(0 또는 1)을 할당 후, n과 result 쌍을 cache에 기록 후 result 반환
- **라인 9-11**: 라인 3과 라인 5의 조건에 해당하지 않으면 fib_memoized(n-1)의 반환 값과 fib_memoized(m-2)의 반환 값의 합을 result에 할당 후, n과 result 쌍을 cache에 기록 후 result 반환

메모 전략을 하향식 동적 프로그래밍이라고도 지칭하는 경우도 있지만, 엄밀히 말하면 동적 프로그래밍이 아님
→ 따라서 메모 전략과 동적 프로그래밍을 서로 별개로 간주하는 것을 추천

동적 프로그래밍 개요 (5/8)

❑ 동적 프로그래밍(dynamic programming)이란? contd.

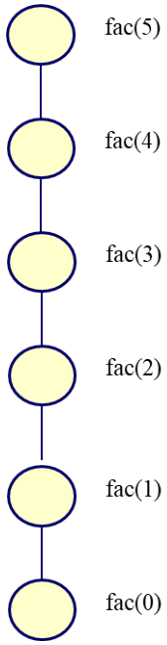
- **예: n번째 피보나치 수(Fibonacci number) 구하기 contd.**
  - **동적 프로그래밍: 어떠한 문제를 해결할 때 동적 프로그래밍을 사용하는가?**

1. 특정 문제를 분할 정복 시, 분할된 부분 문제들이 완벽히 동일하게 중복으로 존재해야 함(**overlapping sub-problem property**)
   - n번째 피보나치 수 구하기 문제와는 다르게 계승(factorial) 구하기, 합병 정렬(merge sort), 퀵 정렬(quick sort), 이진 검색(binary search) 등은 중복된 부분 문제가 존재하지 않음
     - » 예 1: 계승 구하기 문제
       - » 계승의 순환식

$$f_0 = 1$$
$$f_1 = 1$$
$$f_n = n \times f_{n-1}, \text{ for } n \geq 2$$

       - » 분할 정복 전략

```
1  def fac(n):
2      if n == 0 or n == 1:
3          return 1
4      return n * fac(n-1)
5
6  print(fac(5))
```



입력 크기 n에 대한 모든 경우의 수행 시간을 T(n)이라고 하면:
T(1) = 1;
T(n) = 1 + T(n-1).
위의 식을 전개하면:

```
T(n) = 1 +   T(n-1);
     = 2 +   T(n-2);
     = 3 +   T(n-3);
       ...
     = (n-2) + T(2);
     = (n-1) + T(1).
```

T(1)은 1이므로 T(n) = O(n)

       - » 부분 문제에 대한 해를 기록한 후 해당 부분 문제와 완벽히 동일한 부분 문제 해결 시 기록된 해를 재사용하자는 것이 동적 프로그래밍의 가장 중요한 동기인데 중복된 부분 문제가 존재하지 않으므로 (**disjoint sub-problem property**) 기록할 필요가 없음

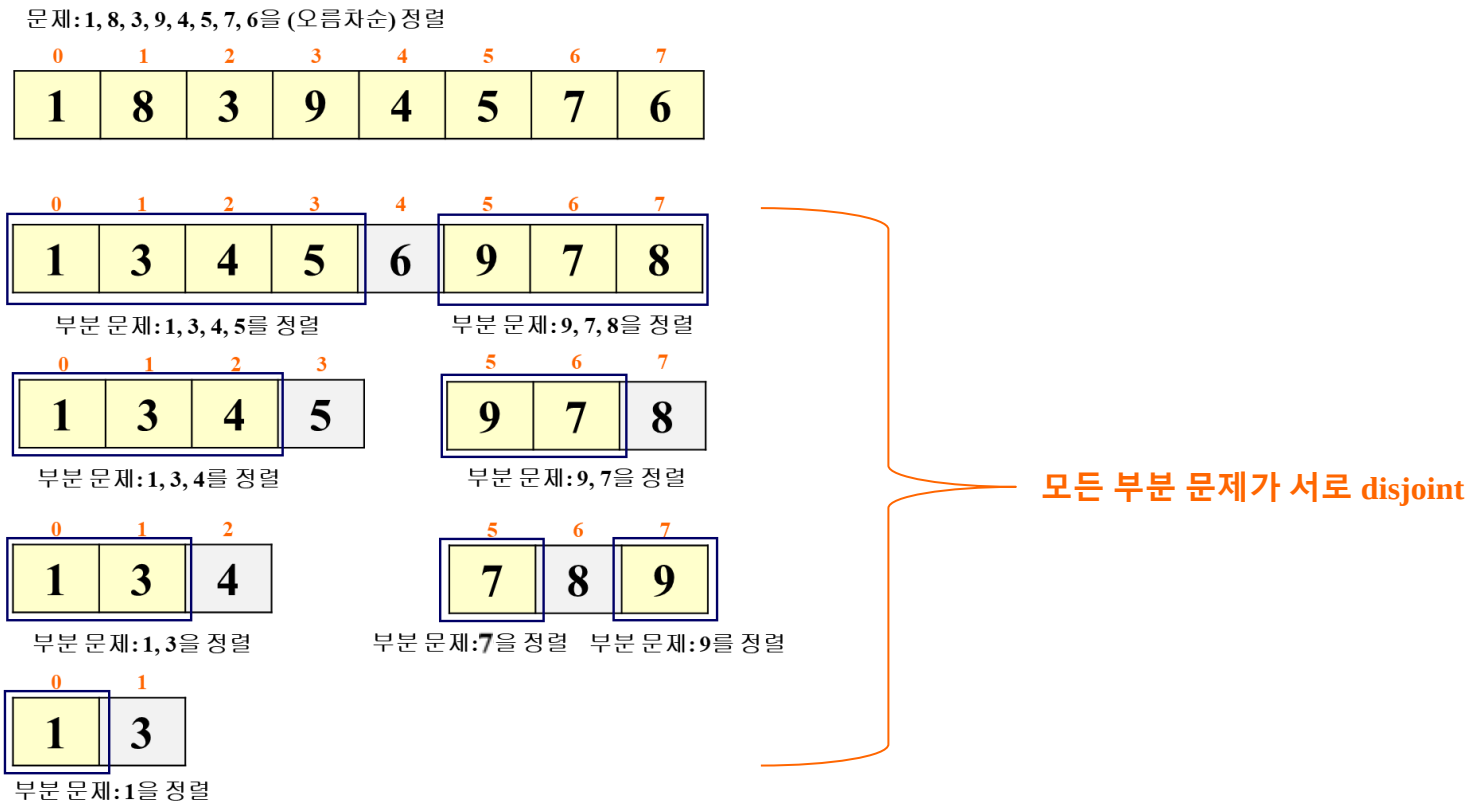
동적 프로그래밍 개요 (6/8)

❑ 동적 프로그래밍(dynamic programming)이란? contd.

- **예: n번째 피보나치 수(Fibonacci number) 구하기 contd.**

» 예 2: 퀵 정렬

» 동일한 부분 문제가 존재 하는가? No! 계승 문제와 마찬가지로 중복된 완벽히 동일한 부분 문제가 존재하지 않으므로 기록할 필요가 없음



- **결론 1**: 부분 문제로 분할 시 완벽히 동일한 부분 문제가 중복 될 경우에(**주어진 문제가 overlapping sub-problem 특성을 만족할 경우에**) 분할 정복 전략 대신 동적 알고리즘을 사용하면 수행 시간을 줄일 수 있음

동적 프로그래밍 개요 (7/8)

❑ 동적 프로그래밍(dynamic programming)이란? contd.

- **예: n번째 피보나치 수(Fibonacci number) 구하기 contd.**

2. 특정 문제를 분할 정복 시, 해당 문제에 대한 최적 해가 분할된 부분 문제들의 최적 해들로 구성이 되어야 함(**최적의 하위 구조: optimal sub-structure property**)

- 예: n번째 피보나치 수 구하기 문제: fib(5) = fib(4) + fib(3)

» 특정 문제에 대한 최적의 해란 다수의 해가 존재할 때 그 중 최적인 해를 의미하는데, 피보나치 수 구하기 문제의 경우 문제 fib(4)에 대한 해가 단 하나이므로 해당 해 3이 fib(4)의 최적 해이며, fib(3)에 대한 해도 단 하나이므로 해당 해 2가 fib(3)의 최적 해임. 이때 문제 fib(5)의 최적 해는 fib(4)의 최적 해와 fib(3)의 최적 해로 구성됨(i.e., 3 + 2)

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2}, \text{ for } n \geq 2$$

- **결론 2**: 주어진 문제가 **최적의 하위 구조 특성**을 만족할 경우 동적 프로그래밍을 사용할 수 있음

동적 프로그래밍 개요 (8/8)

❑ 동적 프로그래밍(dynamic programming)이란? contd.

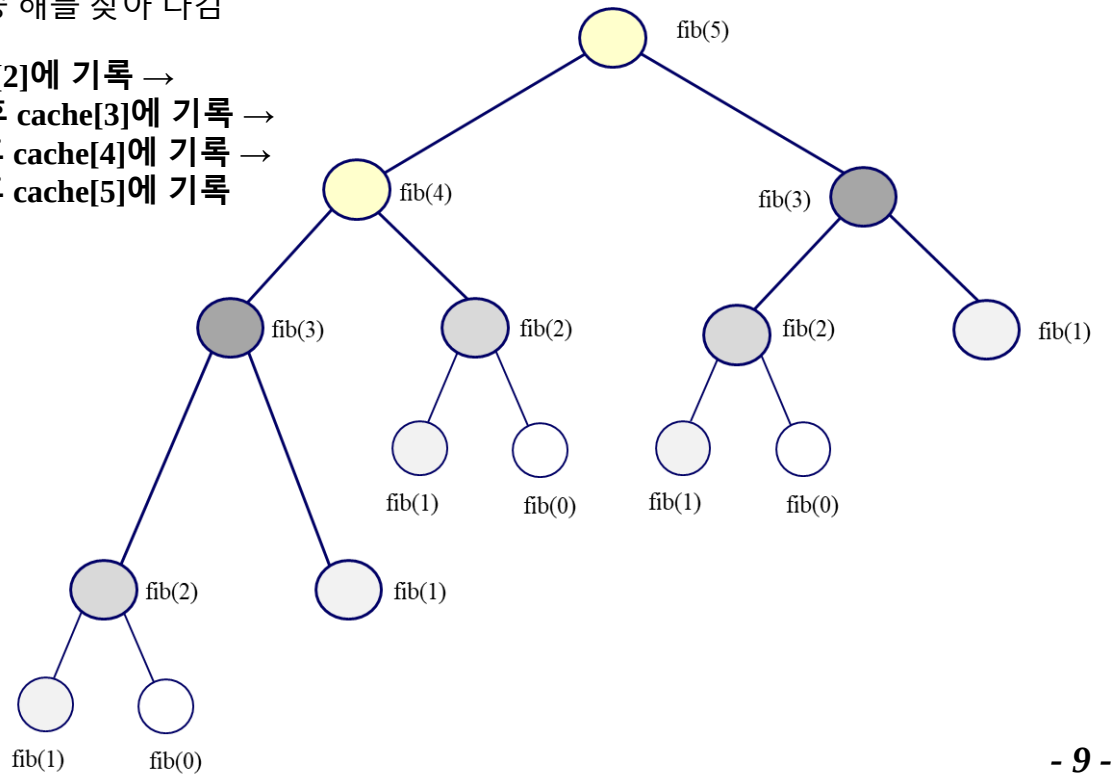
- **예: n번째 피보나치 수(Fibonacci number) 구하기 contd.**
  - **동적 프로그래밍**
    - 동적 프로그래밍은 주어진 문제를 분할 후 더 작은 부분 문제들의 해를 먼저 찾은 후 이들을 통합하여 더 큰 문제의 해를 찾는 **상향식(bottom-up)** 문제 해결 전략으로, 메모 전략과 마찬가지로 부분 문제들의 해를 **기록(tabulation)**하여 완벽히 동일한 부분 문제의 해를 다시 찾아야 할 경우 저장된 해를 재사용
    - 즉, 분할 정복 전략은 위에서 시작하여 문제의 최종 해를 찾기 위한 계산을 해 나가는 반면에, 동적 프로그래밍은 밑에서 시작하여 결과를 기록하면서 문제의 최종 해를 찾아 나감

2nd 피보나치 수: cache[0]의 값과 cache[1]의 값을 더한 후 cache[2]에 기록 →
3rd 피보나치 수: 기록된 cache[2]의 값과 cache[1]의 값을 더한 후 cache[3]에 기록 →
4th 피보나치 수: 기록된 cache[3]의 값과 cache[2]의 값을 더한 후 cache[4]에 기록 →
5th 피보나치 수: 기록된 cache[4]의 값과 cache[3]의 값을 더한 후 cache[5]에 기록

```
1  def fib_dp(n):
2    cache = [0 for _ in range(n + 1)]
3    cache[0] = 0
4    cache[1] = 1
5    for i in range(2, n + 1):
6      cache[i] = cache[i - 1] + cache[i - 2]
7    return cache[n]
8
9  print(fib_dp(5))
```

입력 크기 n에 대한 모든 경우의 수행 시간을 T(n)이라고 하면:

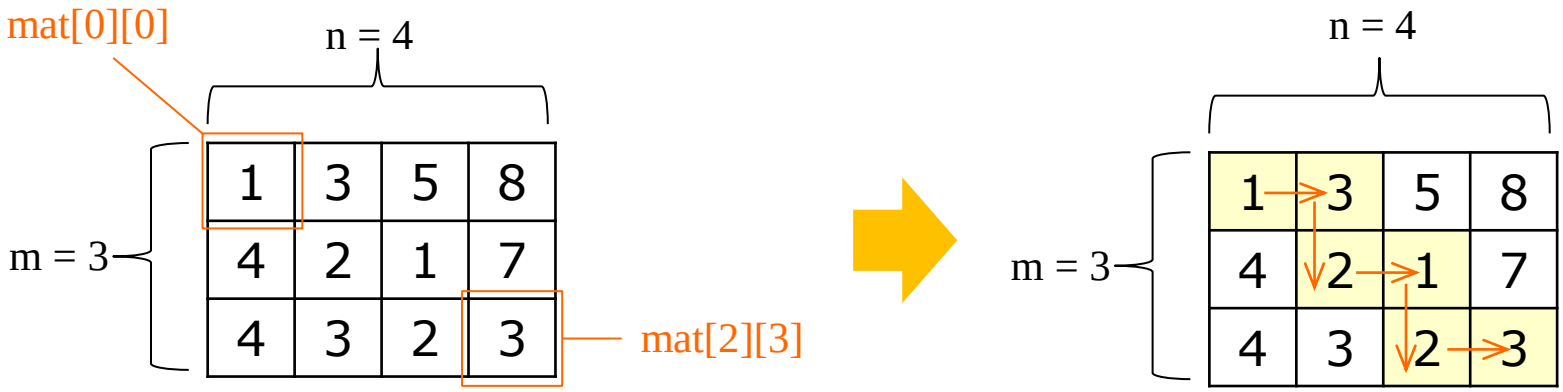
T(n)은 라인 3-4와 라인 5-6의 연산문의 총 수로 2 + (n – 1) = n + 1



동적 프로그래밍 vs. 분할 정복 전략(1/6)

❑ 행렬에서 최소 이동 비용 찾기(finding minimum cost in 2D matrix) 문제

- 정수들이 저장된 m × n 크기의 2차원 행렬 mat가 주어졌을 때 mat의 좌 상단(upper-left corner)으로부터 우 하단(lower-right corner)까지의 이동 비용을 최소화하는 문제로, mat의 각 셀 mat[i][j]에 저장된 정수는 해당 셀을 통과하는데 드는 비용을 나타내며 오른쪽이나 아래쪽 방향으로만 이동할 수 있음
  - 예: 아래의 그림의 행렬 mat에서 좌 상단 mat[0][0]으로부터 우 하단 mat[2][3]까지의 이동 비용을 최소화 할 수 있는 이동 경로는 mat[0][0] → mat[0][1] → mat[1][1] → mat[1][2] → mat[2][2] → mat[2][3]이고 비용은 12임



  - **관찰 1(Observation 1)**: (1) mat[i][j]에 도달하기 위해서는 반드시 mat[i][j-1] 혹은 mat[i-1][j]를 거쳐야 하며, (2) mat[i][j-1] 혹은 mat[i-1][j]까지는 최적의 경로로 이동해야 한다(**최적의 하위 구조 특성**).
    - 예: (1) mat[1][3]에 도달하기 위해서는 반드시 mat[1][2] 혹은 mat[0][3]을 거쳐야 하며, (2) mat[1][2] 혹은 mat[0][3]까지는 최적의 경로인 mat[0][0] → mat[0][1] → mat[1][1] → mat[1][2] (**비용은 7**) 혹은 mat[0][0] → mat[0][1] → mat[0][2] → mat[0][3] (**비용은 17**)을 거쳐야 함
    - 따라서 mat[0][0]으로 부터 mat[1][3]까지의 최소 이동 비용은 min(cost(mat[1][2]), cost(mat[0][3])) + mat[1][3]

mat[0][0]으로부터 mat[1][2]까지의 비용

mat[0][0]으로부터 mat[0][3]까지의 비용

동적 프로그래밍 vs. 분할 정복 전략(2/6)

- **행렬에서 최소 이동 비용 찾기**(finding minimum cost in 2D matrix) **문제** contd.
  - 행렬에서 최소 이동 비용 찾기 문제의 순환식

cost(mat[i][j]): mat[0][0]으로부터 mat[i][j]까지의 최소 이동 비용

$$cost(mat[i][j]) = \begin{cases} mat[i][j] & \text{if } i = 0 \text{ and } j = 0; \\ cost(mat[i][j-1]) + mat[i][j] & \text{if } i = 0; \\ cost(mat[i-1][j]) + mat[i][j] & \text{if } j = 0; \\ \min(\ cost(mat[i-1][j]), cost(mat[i-1][j])\ ) + mat[i][j] & \text{otherwise.} \end{cases}$$

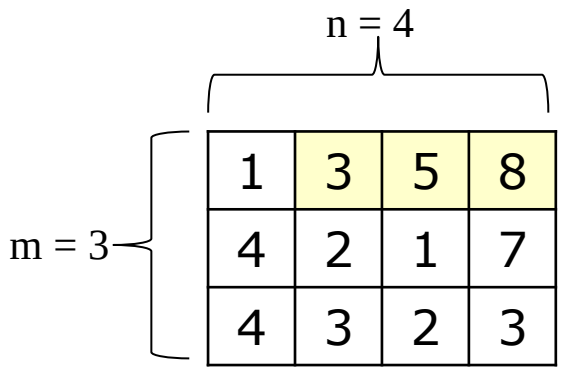
1. 만약 i = 0 and j = 0이라면 출발 셀 mat[0][0]이므로 mat[0][0]에 저장된 값
2. 만약 i = 0이라면 제일 위 행에 있는 셀들이므로 오른쪽으로만 이동해서 도착할 수 있으므로 각 셀의 최소 이동 비용은 자신의 바로 왼쪽에 위치하는 셀의 최소 이동 비용과 자신에게 저장된 값의 합

n = 4

m = 3

1	3	5	8
4	2	1	7
4	3	2	3

cost(mat[0][0]) = mat[0][0] = 1



cost(mat[0][1]) = cost(mat[0][0]) mat[0][1] = 1 + 3 = 4
cost(mat[0][2]) = cost(mat[0][1]) mat[0][2] = 4 + 5 = 9
cost(mat[0][3]) = cost(mat[0][2]) mat[0][3] = 9 + 8 = 17

동적 프로그래밍 vs. 분할 정복 전략(3/6)

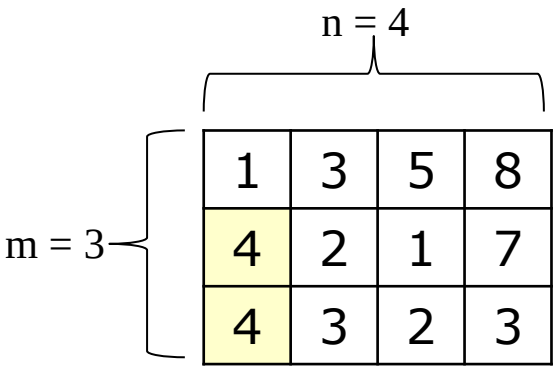
❑ 행렬에서 최소 이동 비용 찾기(finding minimum cost in 2D matrix) 문제 contd.

- 행렬에서 최소 이동 비용 찾기 문제의 순환식 contd.

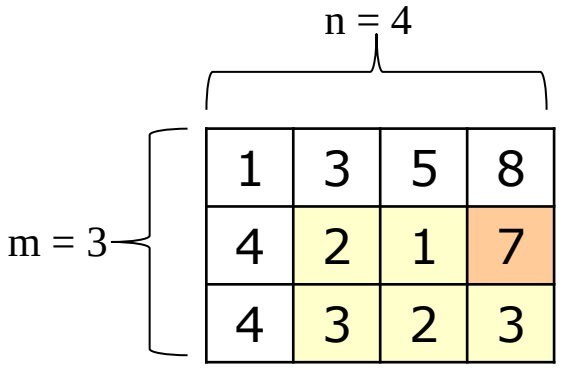
cost(mat[i][j]): mat[0][0]으로부터 mat[i][j]까지의 최소 이동 비용

$$cost(mat[i][j]) = \begin{cases} mat[i][j] & \text{if } i = 0 \text{ and } j = 0; \\ cost(mat[i][j-1]) + mat[i][j] & \text{if } i = 0; \\ cost(mat[i-1][j]) + mat[i][j] & \text{if } j = 0; \\ \min(\ cost(mat[i-1][j]), cost(mat[i-1][j])\ ) + mat[i][j] & \text{otherwise.} \end{cases}$$

3. 만약 j = 0이라면 제일 왼쪽 열의 셀들이므로 아래쪽으로만 이동해서 도착할 수 있으므로 각 셀의 최소 이동 비용은 자신의 바로 위에 위치하는 셀의 최소 이동 비용과 자신에게 저장된 값의 합
4. 위의 세 조건을 만족하지 않는다면 자신의 바로 위에 위치하는 셀의 최소 이동 비용과 자신의 바로 왼쪽에 위치한 셀의 최소 이동 비용 중 값이 더 작은 비용과 자신에게 저장된 값의 합



cost(mat[1][0]) = cost(mat[0][0]) mat[1][0] = 1 + 4 = 5
cost(mat[2][0]) = cost(mat[1][0]) mat[2][0] = 5 + 4 = 9



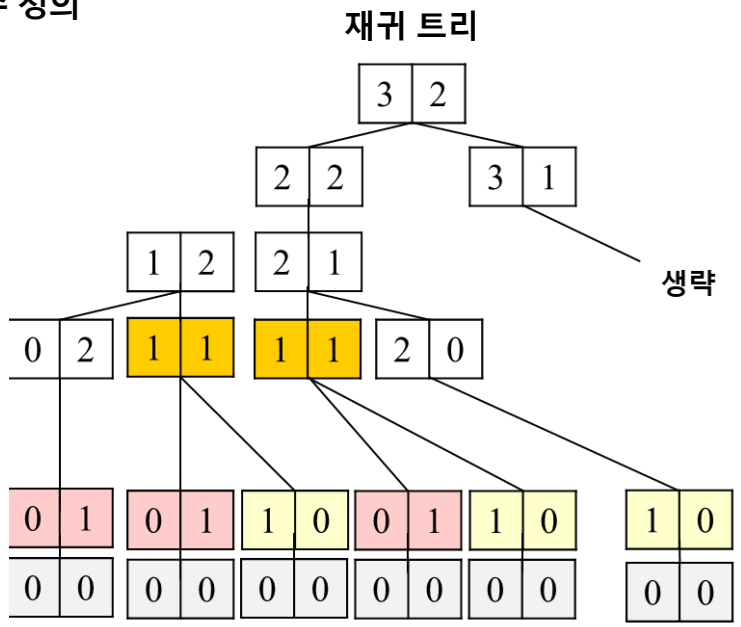
cost(mat[1][3]) =
min(cost(mat[0][3]), cost(mat[1][2])) + mat[1][3] =
7 + 7 = 14

동적 프로그래밍 vs. 분할 정복 전략(4/6)

❑ 행렬에서 최소 이동 비용 찾기(finding minimum cost in 2D matrix) 문제 contd.

- **분할 정복 전략을 이용하여 문제를 해결하기 위한 min_cost_recursion 함수 정의**

```
1   M = 3
2   N = 4
3
4   def get_min(a, b):
5     return a if a < b else b
6
7   def min_cost_recursion(cost, i, j):
8     if i == 0 and j == 0:
9       return mat[0][0]
10    if i == 0:
11      return min_cost_recursion(mat, 0, j-1) + mat[0][j]
12    if j == 0:
13      return min_cost_recursion(mat, i-1, 0) + mat[i][0]
14    a = min_cost_recursion(mat, i - 1, j)
15    b = min_cost_recursion(mat, i, j - 1)
16    return get_min(a, b) + mat[i][j]
17
18  mat = [[1, 3, 5, 8], [4, 2, 1, 7], [4, 3, 2, 3]]
19  print(min_cost_recursion(mat, M - 1, N - 1))
```



- **라인 1-2:** 행렬 사이즈 결정을 위한 변수 M과 N 선언 후 각각 3과 4로 초기화(3행 4열)
- **라인 4-5:** 두 값 중 작은 수를 반환하는 함수 get_min() 정의
- **라인 8-9**: 만약 i = 0 and j = 0이라면(현재 셀이 출발 셀이라면), mat[0][0] 값을 반환
- **라인 10-11**: 만약 i = 0이라면(현재 셀이 제일 위 행에 있는 셀이라면), (1) 바로 왼쪽 셀의 최소 이동 비용을 구하기 위해 재귀 호출 후, (2) 반환 값과 자신이 저장하고 있는 값의 합을 반환
- **라인 12-13**: 만약 j= 0이라면(현재 셀이 제일 왼쪽 열에 있는 셀이라면), (1) 바로 위쪽 셀의 최소 이동 비용을 구하기 위해 재귀 호출 후, (2) 반환 값과 자신이 저장하고 있는 값의 합을 반환
- **라인 14-16**: 라인 8, 라인 10, 라인 12의 세 조건을 만족하지 않는다면, (1) 자신의 바로 위에 위치하는 셀의 최소 이동 비용을 구하기 위해 재귀 호출하고, (2) 자신의 바로 왼쪽에 위치한 셀의 최소 이동 비용을 구하기 위해 재귀 호출 후, (3) 두 반환 값 중 값이 더 작은 비용과 자신에게 저장된 값의 합을 반환

동적 프로그래밍 vs. 분할 정복 전략(5/6)

❑ 행렬에서 최소 이동 비용 찾기(finding minimum cost in 2D matrix) 문제 contd.

- **메모 전략을 이용하여 문제를 해결하기 위한 min_cost_memo 함수 정의**

```
1   M = 3
2   N = 4
3   cache = [[0] * N for i in range(0, M)]
4
5   def get_min(a, b):
6     return a if a < b else b
7
8   def min_cost_memo(mat, i, j):
9     if cache[i][j] != 0:
10        return cache[i][j]
11    if i == 0 and j == 0:
12        cache[i][j] = mat[0][0]
13        return cache[i][j]
14    if i == 0:
15        cache[i][j] = min_cost_memo(mat, 0, j-1) + mat[0][j]
16        return cache[i][j]
17    if j == 0:
18        cache[i][j] = min_cost_memo(mat, i-1, 0) + mat[i][0]
19        return cache[i][j]
20    else:
21      a = min_cost_memo(mat, i - 1, j)
22      b = min_cost_memo(mat, i, j - 1)
23      cache[i][j] = get_min(a, b) + mat[i][j]
24    return cache[i][j]
25
```

- **라인 3:** 기록(memo)을 위한 2차원 리스트 cache 선언
- **라인 9-10:** **만약 현재 셀이 기록되어 있다면 만약 min_cost_memo(mat, i, j)를 이미 호출하여 기록하였다면 해당 기록 값을 반환**
- **라인 11-13**: 만약 i = 0 and j = 0이라면(현재 셀이 출발 셀이라면), mat[0][0] **값을 cache[0][0]에 기록 후** 반환
- **라인 14-16**: 만약 i = 0이라면(현재 셀이 제일 위 행에 있는 셀이라면), (1) 바로 왼쪽 셀의 최소 이동 비용을 구하기 위해 재귀 호출 후, (2) 반환 값과 자신이 저장하고 있는 값의 합을 **cache[0][j]에 기록 후** 반환
- **라인 17-19**: 만약 j= 0이라면(현재 셀이 제일 왼쪽 열에 있는 셀이라면), (1) 바로 위쪽 셀의 최소 이동 비용을 구하기 위해 재귀 호출 후, (2) 반환 값과 자신이 저장하고 있는 값의 합을 **cache[i][0]에 기록 후** 반환
- **라인 20-24**: 라인 9, 라인 11, 라인 14, 라인 17의 네 조건을 만족하지 않는다면, (1) 자신의 바로 위에 위치하는 셀의 최소 이동 비용을 구하기 위해 재귀 호출하고, (2) 자신의 바로 왼쪽에 위치한 셀의 최소 이동 비용을 구하기 위해 재귀 호출 후, (3) 두 반환 값 중 값이 더 작은 비용과 자신에게 저장된 값의 합을 **cache[i][j]에 기록 후** 반환

동적 프로그래밍 vs. 분할 정복 전략(6/6)

❑ 행렬에서 최소 이동 비용 찾기(finding minimum cost in 2D matrix) 문제 contd.

- **동적 프로그래밍을 이용하여 문제를 해결하기 위한 min_cost_dp 함수 정의**

```
1  M = 3
2  N = 4
3
4  def get_min(a, b):
5    return a if a < b else b
6
7  def min_cost_dp(mat, i, j):
8    cache = [[0] * N for i in range(0, M)]
9    cache[0][0] = mat[0][0]
10   for j in range(0, N):
11     cache[0][j] = cache[0][j - 1] + mat[0][j]
12   for i in range(0, M):
13     cache[i][0] = cache[i - 1][0] + mat[i][0]
14   for i in range(1, M):
15     for j in range(1, N):
16       cache[i][j] = get_min(cache[i - 1][j], cache[i][j - 1]) + mat[i][j]
17   return cache[M - 1][N - 1]
18
19 mat = [[1, 3, 5, 8], [4, 2, 1, 7], [4, 3, 2, 3]]
20 print(min_cost_dp(mat, M - 1, N - 1))
```

n = 4

mat

m = 3

1	3	5	8
4	2	1	7
4	3	2	3

cache

1	4	9	17
5	6	7	14
9	9	9	12

- **라인 8-9:** 최소 이동 비용 기록(table)을 위한 2차원 리스트 cache 선언 후 셀 cache[0][0]의 값은 mat[0][0]의 값으로, cache의 다른 셀들의 값은 0으로 초기화
- **라인 10-11:** mat의 제일 위 행에 존재하는 셀들의 최소 이동 비용 계산 및 cache[0][j]에 기록
- **라인 12-13**: mat의 제일 왼쪽 열에 존재하는 셀들의 최소 이동 비용 계산 미 cache[i][0]에 기록

각 셀의 최소 이동 비용은 바로 위 셀의 최소 이동 비용과 바로 왼쪽 셀의 최소 이동 비용을 알고 있어야 연산이 가능하므로, 제일 위 행의 셀들과 제일 왼쪽 셀들의 최소 이동 비용을 미리 계산하여 cache에 기록

- **라인 14-16**: 각 셀의 최소 이동 비용 계산 후 cache에 기록
- **라인 17**: 기록된 cache에서 우 하단 셀의 값(도착 셀까지의 최소 비용)을 반환

동적 프로그래밍 vs. 탐욕적 알고리즘(1/11)

❑ 거스름돈 계산하기 문제(coin change problem)

- 액면가(denomination)가 {1, 5, 6, 8}인 동전을 가지고 거스름돈 n (= 11)을 만들기 위한 **최소 동전 개수**는 몇 개이고 **최소 동전 개수를 위한 동전들의 조합**은 어떻게 되는가? 단, 각 액면가에 해당하는 동전의 개수는 무한하다고 가정
  - 1: 거스름돈 계산하기 문제는 부분 문제로 분할 시 완벽히 동일한 부분 문제가 중복 됨(overlapping sub-problem 특성을 만족함)
  - 2: 거스름돈 계산하기 문제는 최적의 하위 구조(optimal sub-structure) 특성을 만족함
    - 예를 들어 거스름돈 17를 만들기 위한 최소 개수의 동전 조합이 $1 \rightarrow 8 \rightarrow 8$이라면:
      - » 9(17 – 8)를 만들기 위한 최소 개수의 동전 조합은 $1 \rightarrow 8$ 이고;
      - » 1(9 – 8)을 만들기 위한 최소 개수의 동전 조합은 1이다.
  - 따라서 동적 프로그래밍 전략을 이용하여 거스름돈 계산하기 문제에 대한 최적의 해를 항상 찾을 수 있음
  - 탐욕적 알고리즘을 이용하여 거스름돈 계산하기 문제에 대한 최적의 해를 항상 찾을 수는 없을까? No
    - 탐욕적 알고리즘을 이용하여 특정 (최적화) 문제에 대해 항상 최적의 해를 찾는다는 것을 보장하기 위해서는 해당 **문제가 최적의 하위 구조 특성과 탐욕적 선택 특성(greedy choice property)를 동시에 만족**해야 함
    - **탐욕적 선택 특성**: 선택을 해야 할 순간마다 그 순간에 최적이라고 생각되는 것을 부분 해로 선택하는 과정을 반복하면 최종적으로 최적의 해를 찾을 수 있는 특성
      - » 거스름돈 계산하기 문제는 탐욕적 선택 특성을 만족하지 못함: 예를 들어 거스름돈 12를 만들기 위한 부분 해 선택 기준을 액면가가 가장 높은 동전 우선으로 선택한다고 했을 때, 탐욕적 알고리즘은 $8 \rightarrow 1 \rightarrow 1 \rightarrow 1 \rightarrow 1$을 선택하게 됨
    - 최적의 하위 구조 특성과 탐욕적 선택 특성을 쉽게 구분하는 법
      - » 최적의 하위 구조 특성은 최적인 최종 해를 살펴보니 부분 문제들의 최적 해들로 구성되어 있는 것이고, 탐욕적 선택 특성은 부분 문제들의 최적 해를 찾다 보니 최적인 최종 해를 찾은 것

동적 프로그래밍 vs. 탐욕적 알고리즘(2/11)

❑ 거스름돈 계산하기 문제(coin change problem) contd.

- 액면가(denomination)가 {1, 5, 6, 8}인 동전을 가지고 거스름돈 n (= 11)을 만들기 위한 **최소 동전 개수**는 몇 개이고 **최소 동전 개수를 위한 동전들의 조합**은 어떻게 되는가? 단, 각 액면가에 해당하는 동전의 개수는 무한하다고 가정
  - 동적 프로그래밍 전략이므로 다음과 같은 테이블 cache[k][n + 1]을 사용, 여기서 k는 동전 액면의 수(4 개: 1, 5, 6, 8)이고 n은 거스름돈의 총액
  - 또한 동전 시스템을 (Python) 리스트 demo = [1, 5, 6, 8]를 유지 (따라서 k = len(demo) = 4)
    - cache[i][j]에 기록되는 값의 의미는 액면가가 가장 낮은 동전(demo[0])으로부터 i 번째로 낮은 동전(demo[i])을 사용하여 (부분) 거스름돈 j를 만들기 위한 최소(최적) 동전 개수
    - 예를 들어, cache[1][8]에 기록되는 값의 의미는 액면가가 1(**demo[0]**), 5(**demo[1]**)인 동전을 사용하여 부분 거스름돈 8을 만들기 위한 최소(최적) 동전 개수이고, cache[3][11]은 액면가가 1, 5, 6(**demo[2]**), 8(**demo[3]**)인 동전을 사용하여 (최종) 거스름돈 11을 만들기 위한 최소(최적) 동전 개수임

	0	1	2	3	4	5	6	7	8	9	10	11
0 (1)												
1 (5)												
2 (6)												
3 (8)												

k = 4

(n + 1) = 12

동적 프로그래밍 vs. 탐욕적 알고리즘(3/11)

❑ 거스름돈 계산하기 문제(coin change problem) contd.

- 액면가(denomination)가 {1, 5, 6, 8}인 동전을 가지고 거스름돈 n (= 11)을 만들기 위한 **최소 동전 개수**는 몇 개이고 **최소 동전 개수를 위한 동전들의 조합**은 어떻게 되는가? 단, 각 액면가에 해당하는 동전의 개수는 무한하다고 가정
  - 거스름돈 계산하기 문제의 순환식

$$cache[i][j] = \begin{cases} 0 & \text{if } j = 0; \\ j & \text{if } i = 0; \\ cache[i-1][j] & \text{if } demo[i] > j; \\ \min(cache[i-1][j], 1 + cache[i][j - demo[i]]) & \text{otherwise.} \end{cases}$$

1. 만약 j = 0이라면 부분 거스름돈 0을 만들기 위한 최소 동전의 개수이므로 액면가 1인 동전 중 0 개가 필요하고(cache[0][0] = 0), 액면가 1, 5인 동전 중 0 개가 필요하고(cache[1][0] = 0), 액면가 1, 5, 6인 동전 중 0개가 필요하고(cache[2][0] = 0), 액면가 1, 5, 6, 8인 동전 중 0 개가 필요함(cache[3][0] = 0)
2. 만약 i = 0이라면 제일 위 행에 있는 셀들이므로 액면가 1인 동전만을 이용하여 (부분 혹은 최종) 거스름돈 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11을 만들기 위한 최소 동전 개수이므로 j임(예: 액면가 1인 동전만을 가지고 부분 거스름돈 9를 만들기 위한 최소 동전 개수는 9)

	0	1	2	3	4	5	6	7	8	9	10	11
0 (1)	0	1	2	3	4	5	6	7	8	9	10	11
1 (5)	0											
2 (6)	0											
3 (8)	0											

동적 프로그래밍 vs. 탐욕적 알고리즘(4/11)

❑ 거스름돈 계산하기 문제(coin change problem) contd.

- 액면가(denomination)가 {1, 5, 6, 8}인 동전을 가지고 거스름돈 n (= 11)을 만들기 위한 **최소 동전 개수**는 몇 개이고 **최소 동전 개수를 위한 동전들의 조합**은 어떻게 되는가? 단, 각 액면가에 해당하는 동전의 개수는 무한하다고 가정
  - 거스름돈 계산하기 문제의 순환식 contd.

$$cache[i][j] = \begin{cases} 0 & \text{if } j = 0; \\ j & \text{if } i = 0; \\ cache[i-1][j] & \text{if } demo[i] > j; \\ \min(\, cache[i-1][j],\ 1 + cache[i][\, j - demo[i]\,]\,) & \text{otherwise.} \end{cases}$$

3. 만약 demo[i] > j라면 동전의 액면가가 부분 거스름돈의 총액보다 더 높다는 의미임 (예를 들어, cache[1][1]은 액면가 1, 5인 동전을 이용하여 부분 거스름돈 1을 만들기 위한 최소 동전 개수인데 액면가 5인 동전(demo[1])은 사용할 수 없음) → 따라서 액면가가 1인 동전(demo[0])만을 가지고 부분 거스름돈 1을 만들기 위한 최소 동전 개수(부분 해)를 찾아야 하는데 이는 이미 바로 위의 셀인 cache[0][1]에 기록되어 있음(**overlapping sub-problem 특성을 만족함**)
   - » **즉, demo[i] > j라면 cache[i][j] = cache[i – 1][j]**

	0	1	2	3	4	5	6	7	8	9	10	11
0 (1)	0	1	2	3	4	5	6	7	8	9	10	11
1 (5)	0	1	2	3	4							
2 (6)	0											
3 (8)	0											

동적 프로그래밍 vs. 탐욕적 알고리즘(5/11)

❑ 거스름돈 계산하기 문제(coin change problem) contd.

- 액면가(denomination)가 {1, 5, 6, 8}인 동전을 가지고 거스름돈 n (= 11)을 만들기 위한 **최소 동전 개수**는 몇 개이고 **최소 동전 개수를 위한 동전들의 조합**은 어떻게 되는가? 단, 각 액면가에 해당하는 동전의 개수는 무한하다고 가정
  - 거스름돈 계산하기 문제의 순환식 contd.

$$cache[i][j] = \begin{cases} 0 & \text{if } j = 0; \\ j & \text{if } i = 0; \\ cache[i-1][j] & \text{if } demo[i] > j; \\ \min(\, cache[i-1][j],\ 1 + cache[i][\, j - demo[i]\,]\,) & \text{otherwise.} \end{cases}$$

4. 만약 $j \neq 0$이고 $i \neq 0$이고 demo[i] <= j라면 동전의 액면가가 부분 거스름돈의 총액과 같거나 낮다는 의미이고, 이 경우 cache[i-1][j]와 1 + cache[i][j – demo[i]] 중 동전 개수가 더 적은 것을 선택해야 함
   - » 예 1: cache[1][5]는 액면가 1, 5인 동전을 이용하여 부분 거스름돈 5를 만들기 위한 최소 동전 개수인데, **(1)** 액면가 1인 동전만을 가지고 부분 거스름돈 5를 만들기 위한 최소 동전 개수 5(cache[0][5]에 기록되어 있음)와 **(2)** 액면가 5인 동전 한 개 1와 0 (= 5 – 5)을 만들기 위한 최소 동전 개수 0 (cache[1][0]에 기록되어 있음)의 합 중 더 적은 1 + 0을 선택 → 따라서 cache[1][5] = 1
   - » 예 2: cache[1][6]은 액면가 1, 5인 동전을 이용하여 부분 거스름돈 6을 만들기 위한 최소 동전 개수인데, **(1)** 액면가 1인 동전만을 가지고 부분 거스름돈 6을 만들기 위한 최소 동전 개수 6(cache[0][6]에 기록되어 있음)과 **(2)** 액면가 5인 동전 한 개 1과 1 (= 6 – 5)을 만들기 위한 최소 동전 개수 1(cache[1][1]에 기록되어 있음)의 합 중 더 적은 1 + 1을 선택 → 따라서 cache[1][6] = 2
   - » 예 3: cache[1][10]은 액면가 1, 5인 동전을 이용하여 부분 거스름돈 10을 만들기 위한 최소 동전 개수인데, **(1)** 액면가 1인 동전만을 가지고 부분 거스름돈 10을 만들기 위한 최소 동전 개수 10(cache[0][10]에 기록되어 있음)과 **(2)** 액면가 5인 동전 한 개 1과 5 (= 10 – 5)를 만들기 위한 최소 동전 개수 1 (cache[1][5]에 기록되어 있음)의 합 중 더 적은 2을 선택 → 따라서 cache[1][10] = 2

동적 프로그래밍 vs. 탐욕적 알고리즘(6/11)

❑ 거스름돈 계산하기 문제(coin change problem) contd.

- 액면가(denomination)가 {1, 5, 6, 8}인 동전을 가지고 거스름돈 n (= 11)을 만들기 위한 **최소 동전 개수**는 몇 개이고 **최소 동전 개수를 위한 동전들의 조합**은 어떻게 되는가? 단, 각 액면가에 해당하는 동전의 개수는 무한하다고 가정
  - 거스름돈 계산하기 문제의 순환식 contd.

$$cache[i][j] = \begin{cases} 0 & \text{if } j = 0; \\ j & \text{if } i = 0; \\ cache[i-1][j] & \text{if } demo[i] > j; \\ \min(cache[i-1][j],\ 1 + cache[i][j - demo[i]]) & \text{otherwise.} \end{cases}$$

	0	1	2	3	4	5	6	7	8	9	10	11
0 (1)	0	1	2	3	4	5	6	7	8	9	10	11
1 (5)	0	1	2	3	4	1	2	3	4	5	2	3
2 (6)	0											
3 (8)	0											

액면가 6인 동전 한 개 1과 4 (= 10 – 6)를 만들기 위한 최소 동전 개수 4의 합

2와 5(= 1 + 4) 중 더 적은 2 선택

	0	1	2	3	4	5	6	7	8	9	10	11
0 (1)	0	1	2	3	4	5	6	7	8	9	10	11
1 (5)	0	1	2	3	4	1	2	3	4	5	2	3
2 (6)	0	1	2	3	4	1	1	2	3	4	2	2
3 (8)	0	1	2	3	4	1	1	2	1	2	2	2

0 번째 행을 제외한 각 행의 셀의 값을 위와 같은 방식으로 기록할 경우 최종 테이블

최적의 최종 결과는 cache[3][11]에 기록

동적 프로그래밍 vs. 탐욕적 알고리즘(7/11)

❑ 거스름돈 계산하기 문제(coin change problem) contd.

- 액면가가 $\{d_1, d_2, ..., d_k\}$인 동전을 가지고 거스름돈 n을 만들기 위한 최소 동전 개수를 찾기 위한 coin_change 함수 정의

```
1  def get_min(a, b):
2    return a if a < b else b
3
4  def coin_change(demo, n):
5      k = len(demo)
6      cache = [[0] * (n + 1) for i in range(0, k)]
7      for i in range(0, k):
8          cache[i][0] = 0
9      for j in range(1, n + 1):
10         cache[0][j] = j
11
12     for i in range(1, k):
13         for j in range(1, n + 1):
14             if demo[i] > j:
15                 cache[i][j] = cache[i - 1][j]
16             else:
17                 cache[i][j] = (
18                 get_min(cache[i - 1][j], 1 + cache[i][j - demo[i]])
19                 )
20
21     return cache[k-1][n]
22
23
24 demo = [1, 5, 10, 12, 50, 100, 500]
25 n = 16
26 print(coin_change(demo, n))
```

- 라인 1-2: 두 값 중 작은 수를 반환하는 함수 get_min() 정의
- 라인 5: 변수 k 선언 후 서로 다른 액면가를 가진 동전의 수 할당(예: 동전 시스템이 1, 5, 6, 8이면 k = 4)
- 라인 6: (부분·최종) 거스름돈을 만들기 위한 최소 동전 개수(부분·최종 해) 기록(table)을 위한 k 행 (n+1) 열 2차원 리스트 cache 선언 후 모든 셀의 값을 0으로 초기화 Note: (n + 1) 열인 이유는 부분 거스름돈 0원부터 시작하여 0, 1, 2, 3, ..., n원을 만들기 위한 최소 동전 개수를 기록해야 하기 때문

동적 프로그래밍 vs. 탐욕적 알고리즘(8/11)

❑ 거스름돈 계산하기 문제(coin change problem) contd.

- 액면가가 $\{d_1, d_2, ..., d_k\}$인 동전을 가지고 거스름돈 n을 만들기 위한 최소 동전 개수를 찾기 위한 coin_change 함수 정의 contd.

```
1  def get_min(a, b):
2    return a if a < b else b
3
4  def coin_change(demo, n):
5      k = len(demo)
6      cache = [[0] * (n + 1) for i in range(0, k)]
7      for i in range(0, k):
8          cache[i][0] = 0
9      for j in range(1, n + 1):
10         cache[0][j] = j
11
12     for i in range(1, k):
13         for j in range(1, n + 1):
14             if demo[i] > j:
15                 cache[i][j] = cache[i - 1][j]
16             else:
17                 cache[i][j] = (
18                 get_min(cache[i - 1][j], 1 + cache[i][j - demo[i]])
19                 )
20
21     return cache[k-1][n]
22
23
24 demo = [1, 5, 10, 12, 50, 100, 500]
25 n = 16
26 print(coin_change(demo, n))
```

- 라인 7-8: 만약 j = 0이라면 부분 거스름돈 0을 만들기 위한 최소 동전의 개수이므로 cache의 각 행의 j 열(0 열) 값을 0으로 기록(예: 동전 시스템이 1, 5, 6, 8이면 cache[0][0] = 0, cache[1][0] = 0, cache[2][0] = 0, cache[3][0] = 0)
- 라인 9-10: 만약 i = 0이면 i 행(0 행)의 각 열 값을 j로 기록(예: 동전 시스템이 1, 5, 6, 8이면 액면가 1인 동전만을 이용하여 (부분 혹은 최종) 거스름돈 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11을 만들기 위한 최소 동전 개수이므로 0 행 1 열 값 cache[0][1]을 1로, 0 행 2열 값 cache[0][2]를 2로, …, 0 행 11열 값 cache[0][11]을 11로 기록)

동적 프로그래밍 vs. 탐욕적 알고리즘(9/11)

❑ 거스름돈 계산하기 문제(coin change problem) contd.

- 액면가가 $\{d_1, d_2, ..., d_k\}$인 동전을 가지고 거스름돈 n을 만들기 위한 최소 동전 개수를 찾기 위한 coin_change 함수 정의 contd.

```
1  def get_min(a, b):
2    return a if a < b else b
3
4  def coin_change(demo, n):
5      k = len(demo)
6      cache = [[0] * (n + 1) for i in range(0, k)]
7      for i in range(0, k):
8          cache[i][0] = 0
9      for j in range(1, n + 1):
10         cache[0][j] = j
11
12     for i in range(1, k):
13         for j in range(1, n + 1):
14             if demo[i] > j:
15                 cache[i][j] = cache[i - 1][j]
16             else:
17                 cache[i][j] = (
18                 get_min(cache[i - 1][j], 1 + cache[i][j - demo[i]])
19                 )
20
21     return cache[k-1][n]
22
23
24 demo = [1, 5, 10, 12, 50, 100, 500]
25 n = 16
26 print(coin_change(demo, n))
```

- 라인 12 - 19: cache의 0 행을 제외한 각 행(각 행은 각 액면가 동전)들에 대해 다음과 같은 과정을 반복
  - 라인 13 – 19: 만약 각 행 i에 해당하는 액면가가 j 값(부분·최종 거스름돈)보다 높다면 cache[i][j]의 값을 (이미 기록된) cache[i – 1][j] 값으로 기록하고(라인 13 – 15), 만약 각 행 i에 해당하는 액면가가 j 값과 같거나 낮다면 경우 (이미 기록된) cache[i-1][j]와 1 + cache[i][j – demo[i]] 중 동전 개수가 더 적은 것을 선택하여 cache[i][j]의 값으로 기록
- 라인 21: cache의 우하단 셀의 값을 최종 결과로 반환

동적 프로그래밍 vs. 탐욕적 알고리즘(10/11)

❑ 거스름돈 계산하기 문제(coin change problem) contd.

- 액면가가 $\{d_1, d_2, ..., d_k\}$인 동전을 가지고 거스름돈 n을 만들기 위한 **최소 수의 동전 조합은 어떻게 출력**할까?
  - 만약 특정 셀 cache[i][j]의 값이 바로 위의 셀 cache[i-1][j]의 값으로 기록되었다면 현재 행 i에 해당하는 액면가 동전은 사용되지 않은 것이며, 그렇지 않다면 i에 해당하는 액면가 동전이 사용된 것임
- 액면가가 $\{d_1, d_2, ..., d_k\}$인 동전을 가지고 거스름돈 n을 만들기 위한 최소 수의 동전 조합 출력을 위한 print_coin_change 함수 정의

```
21  def print_coin_change(demo, cache, n):
22      i = len(demo) - 1
23      j = n
24      while j != 0:
25          if cache[i - 1][j] == cache[i][j] and i > 0:
26              i = i - 1
27          else:
28              print(demo[i])
29              j = j - demo[i]
```

  - 라인 21-22: cache의 행과 열을 위한 변수 i, j를 선언 후 마지막 행과 마지막 열로 초기화(**마지막 행과 마지막 열의 셀에 기록된 값은 최종 해**)
  - 라인 24: 최종 거스름돈 n (= j)이 0이 아닐 때까지 반복
    - 라인 25-26: 0 행을 제외한 i 행의 셀들 중 셀에 기록된 값이 바로 위의 셀 값으로 기록되었다면(라인 25의 조건을 만족한다면), i 행에 해당하는 액면가 동전(예: 최종 거스름돈 11을 만들기 위한 동전 중 액면가 8인 동전)은 사용되지 않았으므로 i – 1 행에 해당하는 액면가 동전(예: 최종 거스름돈 11을 만들기 위한 동전 중 액면가 6인 동전)을 고려
    - 라인 27-29: 위의 조건(라인 25의 조건)을 만족하지 않았다면 i 행에 해당하는 액면가 동전(예: 최종 거스름돈 11을 만들기 위한 동전 중 액면가 6인 동전)이 사용된 것이므로 출력하고 최종 거스름돈을 갱신(예: 최종 거스름돈 11 – 액면가 6 = 5)

동적 프로그래밍 vs. 탐욕적 알고리즘(11/11)

❑ 거스름돈 계산하기 문제(coin change problem) contd.

- 액면가가 $\{d_1, d_2, ..., d_k\}$인 동전을 가지고 거스름돈 n을 만들기 위한 **최소 수의 동전 조합은 어떻게 출력**할까? contd.

```
1  def get_min(a, b):
2    return a if a < b else b
3
4  def coin_change(demo, n):
5      k = len(demo)
6      cache = [[0] * (n + 1) for i in range(0, k)]
7      for i in range(0, k):
8          cache[i][0] = 0
9      for j in range(1, n + 1):
10         cache[0][j] = j
11
12     for i in range(1, k):
13         for j in range(1, n + 1):
14             if demo[i] > j:
15                 cache[i][j] = cache[i - 1][j]
16             else:
17                 cache[i][j] =(
18                 get_min(cache[i - 1][j], 1 + cache[i][j - demo[i]])
19                 )
20
21     print_coin_change(demo, cache, n)
22     return cache[k-1][n]
23
24 def print_coin_change(demo, cache, n):
25     i = len(demo) - 1
26     j = n
27     while j != 0:
28         if cache[i - 1][j] == cache[i][j] and i > 0:
29             i = i - 1
30         else:
31             print(demo[i], end = ' ')
32             j = j - demo[i]
33
34
35 demo = [1, 5, 10, 12, 50, 100, 500]
36 n = 16
37 coin_change(demo, n)
```

결과

```
10 5 1
```

수행시간 분석

라인 12 – 19에서 (0 행에 해당하는 액면가 동전을 제외한) 각 액면가 동전에 대해 (부분·최종) 거스름돈 1, 2, ..., n을 만들기 위한 최소 동전 수를 찾으므로 $O(k \times n)$
→ **제곱시간 알고리즘**