1 동적 계획법

1 동적 계획법

1 재귀 호출 알고리즘

- 다른 크기의 문제들이 서로 재귀적 관계를 가질 때 재귀 호출 알고리즘을 사용하면 자연스럽게 구현할 수 있는데 재귀적 구현이 효율적인 경우도 있고 그렇지 못한 경우도 있음

- 지나친 중복이 발생하기도 함

1 동적 계획법

2 재귀적 해법이 적합한 예

- 재귀적 해법이 바람직한 예
  - 퀵 정렬, 병합 정렬 등의 정렬 알고리즘
  - 계승(Factorial) 구하기
  - 그래프의 너비 우선 탐색 등

- 재귀적 해법이 치명적인 예
  - 피보나치 수 구하기
  - 행렬 곱셈 최적순서 구하기 등

3 동적 계획법

- 큰 문제의 해답에 작은 문제의 해답이 포함되어 있고 이를 재귀 호출 알고리즘으로 구현하면 지나친 중복이 발생하는 경우 이 재귀적 중복을 해결하는 방법
- 어떤 문제가 여러 단계의 반복되는 부분 문제로 이루어질 때, 각 단계에 있는 부분 문제의 답을 기반으로 전체 문제의 답을 구하는 방법
- 작은 문제들의 해를 먼저 구하여 저장하고 더 큰 문제의 해를 구할 때 작은 문제의 해를 반복 계산하지 않고 저장된 결과를 사용하는 방법
- 동적 프로그래밍이라고도 함

3 동적 계획법

- 동적 계획법을 이용하여 문제를 풀기 위해서는 그 문제가 **최적 부분 구조(Optimal Substructure)**를 가지고 있어야 함
- 최적 부분 구조란 전체 문제의 최적해가 부분 문제의 최적해로부터 만들어지는 구조

3 동적 계획법

- 예)

예를 들어 5개의 작은 문제로 쪼갤 수 있는 어떤 문제가 있다고 해보자. 쪼개진 문제의 해 5개를 모두 얻어야 이 문제의 해를 구할 수 있다면 이 문제는 최적 부분 구조를 갖추었다고 할 수 있다.

➔ 최적 부분 구조를 가진 문제는 재귀 호출을 사용해 문제를 풀 수 있음

➔ 재귀적 알고리즘은 간명하지만 때때로 엄청난 비효율을 초래하기도 함

4 피보나치 수 구하기

- 피보나치 수는 부분 문제의 답으로부터 본 문제의 답을 얻을 수 있으므로 최적 부분 구조로 이루어져 있음

$$f(n) = f(n-1) + f(n-2)$$
$$f(1) = f(2) = 1$$

- 아주 간단한 문제지만 동적 계획법의 동기와 구현이 다 포함되어 있음

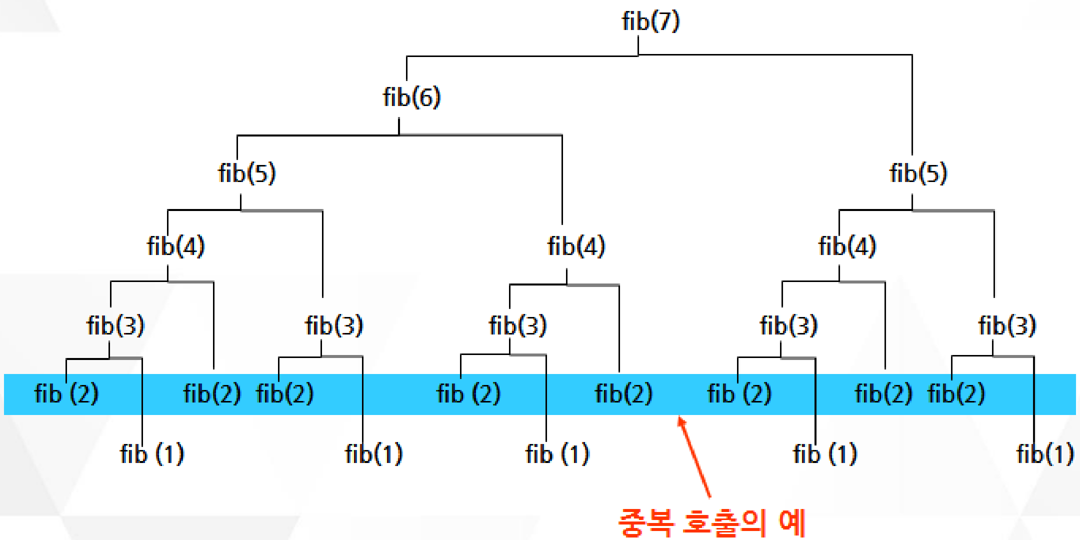
5 피보나치 수를 구하는 재귀 알고리즘

```
fib(n)
{
        if (n = 1 or n = 2) then
            return 1;
        else
           return (fib(n-1) +fib(n-2));
}
```

- ✓ 엄청난 중복 호출이 존재함
- ✓ 지수 함수에 비례하는 시간이 듬

6 피보나치 수열의 호출 트리

- fib(7)을 구하기 위해 fib(5)는 2번, fib(4)는 3번, fib(3)은 5번, fib(2)는 8번 호출됨



1 동적 계획법

7 재귀적 구현의 문제점

- 재귀적 구현으로 동일한 문제가 중복 호출됨
- 문제의 크기가 커짐에 따라 중복 호출도 증가함
- 한번만 구해서 저장해 놓았다가 나중에 다시 사용만 하면 되는데 매번 호출함으로써 비효율이 발생함

7 재귀적 구현의 문제점

- 예)

fib(3)을 처음 호출하고 결과를 얻었으면
이를 저장해두고 나중에 fib(3)이 필요할 때
저장한 것을 사용하면 됨

➔ 이런식으로 **부분 결과를 저장하면서 해를 구해가는 것이 동적 계획법임**

2 동적 계획법의 특징

2 동적 계획법의 특징

1 동적 계획법 기반의 알고리즘 동작 방식

1. 문제를 부분 문제로 나눔
2. 가장 작은 부분 문제부터 해를 구한 뒤 테이블에 저장함
3. 테이블에 저장되어 있는 부분 문제의 해를 이용하여 점차적으로 상위 부분 문제의 최적해를 구함

2 동적 계획법의 특징

1 동적 계획법 기반의 알고리즘 동작 방식

① 문제를 부분 문제로 분할함

- fib(n) 함수는 fib(n-1)과 fib(n-2)의 합
- fib(n-1)은 fib(n-2)와 fib(n-3)의 합
- fib(2)는 fib(1)과 fib(0)의 합
- fib(n)은 fib(n-1), fib(n-2), … fib(2), fib(1), fib(0) 의 부분 집합으로 나뉨

2 동적 계획법의 특징

1 동적 계획법 기반의 알고리즘 동작 방식

2 부분 문제로 나누는 일을 끝냈으면
가장 작은 부분 문제부터 해를 구함

3 그 결과는 테이블에 저장하고, 테이블에 저장된
부분 문제의 해를 이용하여 상위 문제의 해를 구함

테이블 인덱스	저장되어 있는 값
[0]	0
[1]	1
[2]	1
[3]	2
[4]	3
[5]	5
[6]	8
[7]	13
[8]	21
[9]	34
[10]	55
...	...
[n]	fib(n)

3 동적 계획법으로 피보나치 수 구하는 알고리즘

- 배열 f[ ]에 작은 것부터 저장해가면서 계산하는 방식

```
fibonacci(n)
{
        f[1] = f[2] = 1;
        for i = 3 to n
                f[i] = f[i-1] +f[i-2];
        return f[n];
}
```

✓ 선형시간에 끝남

- for 루프가 한번 돌 때마다 앞에서 구해 저장해 놓은 피보나치 수 2개를 배열에 가져다 더하면 됨
- 중복 호출이 야기한 비효율을 제거함

4 동적 계획법으로 풀 수 있는 문제의 특징

- 최적 부분 구조를 이룸
- 재귀적으로 구현했을 때 재귀 호출이 심하게 중복되어 심각한 비효율이 발생함
  - 거의 모든 재귀적 알고리즘은 최적 부분 구조를 구현

➡ **동적 계획법이 그 해결책**

- 참고

> 최적 부분 구조 : 큰 문제의 최적 솔루션에
> 작은 문제의 최적 솔루션이 포함됨

4 동적 계획법으로 풀 수 있는 문제의 특징

- 병합 정렬, 퀵 정렬 등도 큰 문제의 답이 작은 문제의 답을 포함하지만 이들의 재귀적 구현에서는 중복이 발생하지 않음
- 계승도 최적 부분 구조를 갖지만
이를 재귀적으로 구현한 것은 중복이 발생하지 않음

→ 동적 계획법의 대상이 되는 문제들과 다른 부분임

3 동적 계획법의 적용 예

3 동적 계획법의 적용 예

1 행렬 경로 문제 (예제)

- 양수로 이루어진 n×n 행렬이 주어지고 행렬의 왼쪽 위에서 시작해 한 칸씩 이동해 오른쪽 아래까지 도달함

- 이동 방법 (제약조건)
  - 오른쪽이나 아래쪽으로만 이동할 수 있음
  - 왼쪽, 위쪽, 대각선 이동은 허용하지 않음

- 목표
  - 행렬의 원소 (1, 1)에서 (n, n)까지 이동하는 모든 경로의 점수 중 가장 높은 점수를 구하는 문제

3 동적 계획법의 적용 예

1 행렬 경로 문제 (예제)

불법 이동의 예

✓ 오른쪽이나 아래쪽으로만 이동할 수 있음

6	7	12	5
5	3	11	18
7	17	3	3
8	10	14	9

[불법 이동 (상향)]

6	7	12	5
5	3	11	18
7	17	3	3
8	10	14	9

[불법 이동 (좌향)]

3 동적 계획법의 적용 예

1 행렬 경로 문제 (예제)

유효한 이동의 예

6	7	12	5
5	3	11	18
7	17	3	3
8	10	14	9

6	7	12	5
5	3	11	18
7	17	3	3
8	10	14	9

3

동적 계획법의 적용 예

1 행렬 경로 문제 (예제)

최적 부분 구조 존재 확인

- (1, 1)에서 원소 (i, j)까지 도달하는 경로들의 점수 중 최고점 구하기

원소 (i, j)에 도달하기 직전에
방문할 수 있는 원소는 (i-1, j)와 (i, j-1) 두 개임
원소 (i, j)는 방문하므로 원소 (i, j) 값은 반드시 더해짐

① (i-1, j)를 거쳐 (i, j)에 도달하는 점수
② (i, j-1)을 거쳐 (i, j)에 도달하는 점수
둘 중 큰 것에 원소 (i, j)의 점수를 더하면
원소 (i, j)까지의 최고 점수가 됨

문제 (i, j)의 최적해는
문제 (i-1, j)의 최적해와
문제 (i, j-1)의 최적해로
설명됨

➔ **최적 부분 구조임**

3 동적 계획법의 적용 예

1 행렬 경로 문제 (예제)

재귀적 관계 확인

- 변수 c_{ij}는 (1, 1)에서 (i, j)에 이르는 최고 점수
  최종적으로 구하는 것은 c_{nn}
  m_{ij}는 행렬의 원소 (i, j)의 값

$$
c_{ij} = \begin{cases} 0 & \text{if } i=0 \text{ 또는 } j=0 \\ m_{ij} + \max\{c_{i-1,j},\ c_{i,j-1}\} & \text{otherwise} \end{cases}
$$

3 동적 계획법의 적용 예

1 행렬 경로 문제 (예제)

재귀 호출 알고리즘

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

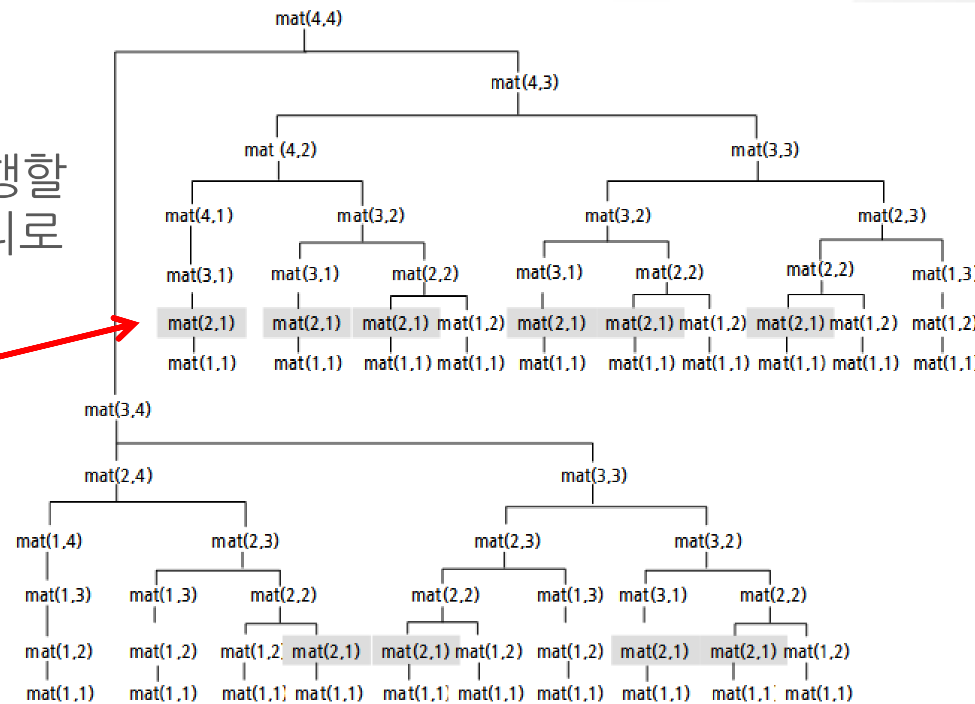
3 동적 계획법의 적용 예

1 행렬 경로 문제 (예제)

호출 트리의 예

- matrixPath(4, 4)를 수행할 때 재귀 호출 관계를 트리로 나타냄

중복 호출됨



- matrixPath(2, 1)은 10번 중복 호출됨

3

동적 계획법의 적용 예

1 행렬 경로 문제 (예제)

호출 트리의 예

- 4X4 행렬의 경우 25개의 문제를 포함하고 있는데 총 139번의 호출이 일어남
- 3X3 행렬은 9개의 문제를 포함하고 있으며 39번의 호출이 일어남

➔ 문제가 커지면 중복 호출이 지수 함수적으로 일어남

3 동적 계획법의 적용 예

1 행렬 경로 문제 (예제)

호출 트리의 예

- matrixPath( )에서 문제 크기가 커짐에 따라 중복 호출이 증가하는 모습

➔ 동적 프로그래밍하기 좋은 대상임

수행되는 matrixPath( )	matrixPath(2, 1)의 중복 호출 횟수
matrixPath(2, 2)	1
matrixPath(3, 3)	3
matrixPath(4, 4)	10
matrixPath(5, 5)	35
matrixPath(6, 6)	126
matrixPath(7, 7)	462
matrixPath(8, 8)	1,716
matrixPath(9, 9)	6,435

3 동적 계획법의 적용 예

1 행렬 경로 문제 (예제)

동적 계획법 알고리즘

- n×n 행렬에서 존재하는 부분 문제들의 총수는 고작 n^2개 인데 이 n^2을 아래에서부터(작은 것부터) 재귀적 관계를 이용해 구해나감

```
matrixPath(n)    ▷ (n, n)에 이르는 최고 점수
{
    for i ← 0 to n
        c[i, 0] ← 0;
    for j ← 1 to n
        c[0, j] ← 0;

1   for i ← 1 to n
        for j ← 1 to n                                  2
            c[i, j] ← m_ij + max(c[i-1, j], c[i, j-1]);

    return c[n, n];
}
```

3 동적 계획법의 적용 예

1 행렬 경로 문제 (예제)

동적 계획법 알고리즘의 수행시간

- matrixPath( ) 알고리즘의 수행시간은 ①의 for 루프가 지배함
- ②는 배열의 두 원소 중 큰 것을 고르는 작업과 덧셈이므로 상수 시간이 듦

➔ **총 수행시간은 $O(n^2)$**

- 행렬의 원소수 n^2에 대해서는 선형 시간임