다이나믹 프로그래밍

최백준 choi@startlink.io

다이나믹 프로그래밍

다이나믹 프로그래밍

Dynamic Programming

- 큰 문제를 작은 문제로 나눠서 푸는 알고리즘
- Dynamic Programming의 다이나믹은 아무 의미가 없다.
- 이 용어를 처음 사용한 1940년 Richard Bellman은 멋있어보여서 사용했다고 한다
- https://en.wikipedia.org/wiki/Dynamic_programming#History

다이나믹 프로그래밍

Dynamic Programming

- 두 가지 속성을 만족해야 다이나믹 프로그래밍으로 문제를 풀 수 있다.

1. Overlapping Subproblem
2. Optimal Substructure

Overlapping Subproblem

Overlapping Subproblem

- 피보나치 수
- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}\ (n \geq 2)$

Overlapping Subproblem

Overlapping Subproblem

- 피보나치 수
- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$ $(n \geq 2)$
- 문제: N번째 피보나치 수를 구하는 문제
- 작은 문제: N-1번째 피보나치 수를 구하는 문제, N-2번째 피보나치 수를 구하는 문제

Overlapping Subproblem

Overlapping Subproblem

- 문제: N번째 피보나치 수를 구하는 문제
- 작은 문제: N-1번째 피보나치 수를 구하는 문제, N-2번째 피보나치 수를 구하는 문제

- 문제: N-1번째 피보나치 수를 구하는 문제
- 작은 문제: N-2번째 피보나치 수를 구하는 문제, N-3번째 피보나치 수를 구하는 문제

- 문제: N-2번째 피보나치 수를 구하는 문제
- 작은 문제: N-3번째 피보나치 수를 구하는 문제, N-4번째 피보나치 수를 구하는 문제

Overlapping Subproblem

Overlapping Subproblem

- 큰 문제와 작은 문제는 상대적이다.

- 문제: N번째 피보나치 수를 구하는 문제
- 작은 문제: N-1번째 피보나치 수를 구하는 문제, N-2번째 피보나치 수를 구하는 문제

- 문제: N-1번째 피보나치 수를 구하는 문제
- 작은 문제: N-2번째 피보나치 수를 구하는 문제, N-3번째 피보나치 수를 구하는 문제

Overlapping Subproblem

Overlapping Subproblem

- 큰 문제와 작은 문제를 같은 방법으로 풀 수 있다.
- 문제를 작은 문제로 쪼갤 수 있다.

Optimal Substructure

Optimal Substructure

- 문제의 정답을 작은 문제의 정답에서 구할 수 있다.

- 예시
- 서울에서 부산을 가는 가장 빠른 길이 대전과 대구를 순서대로 거쳐야 한다면
- 대전에서 부산을 가는 가장 빠른 길은 대구를 거쳐야 한다.

Optimal Substructure

Optimal Substructure

- 문제: N번째 피보나치 수를 구하는 문제
- 작은 문제: N-1번째 피보나치 수를 구하는 문제, N-2번째 피보나치 수를 구하는 문제
- **문제의 정답을 작은 문제의 정답을 합하는 것으로 구할 수 있다.**
- 문제: N-1번째 피보나치 수를 구하는 문제
- 작은 문제: N-2번째 피보나치 수를 구하는 문제, N-3번째 피보나치 수를 구하는 문제
- **문제의 정답을 작은 문제의 정답을 합하는 것으로 구할 수 있다.**
- 문제: N-2번째 피보나치 수를 구하는 문제
- 작은 문제: N-3번째 피보나치 수를 구하는 문제, N-4번째 피보나치 수를 구하는 문제
- **문제의 정답을 작은 문제의 정답을 합하는 것으로 구할 수 있다.**

Optimal Substructure

Optimal Substructure

- Optimal Substructure를 만족한다면, 문제의 크기에 상관없이 어떤 한 문제의 정답은 일정하다.
- 10번째 피보나치 수를 구하면서 구한 4번째 피보나치 수
- 9번째 피보나치 수를 구하면서 구한 4번째 피보나치 수
- …
- 5번째 피보나치 수를 구하면서 구한 4번째 피보나치 수
- 4번째 피보나치 수를 구하면서 구한 4번째 피보나치 수

- 4번째 피보나치 수는 항상 같다.

다이나믹 프로그래밍

Dynamic Programming

- 다이나믹 프로그래밍에서 각 문제는 한 번만 풀어야 한다.
- Optimal Substructure를 만족하기 때문에, 같은 문제는 구할 때마다 정답이 같다.
- 따라서, 정답을 한 번 구했으면, 정답을 어딘가에 메모해놓는다.
- 이런 메모하는 것을 코드의 구현에서는 배열에 저장하는 것으로 할 수 있다.
- 메모를 한다고 해서 영어로 Memoization이라고 한다.

피보나치 수

Dynamic Programming

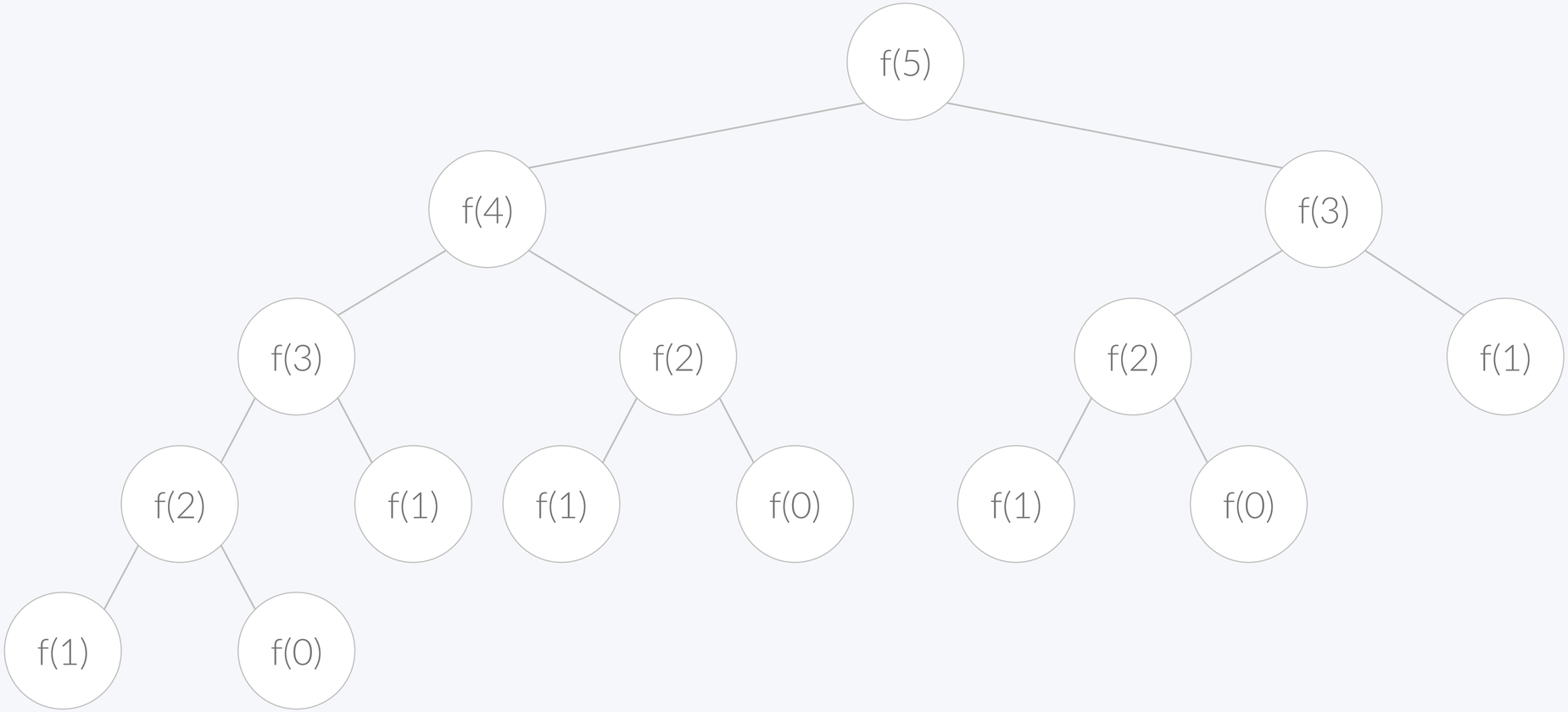
```
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

- 피보나치 수를 구하는 함수이다.

피보나치 수

Dynamic Programming

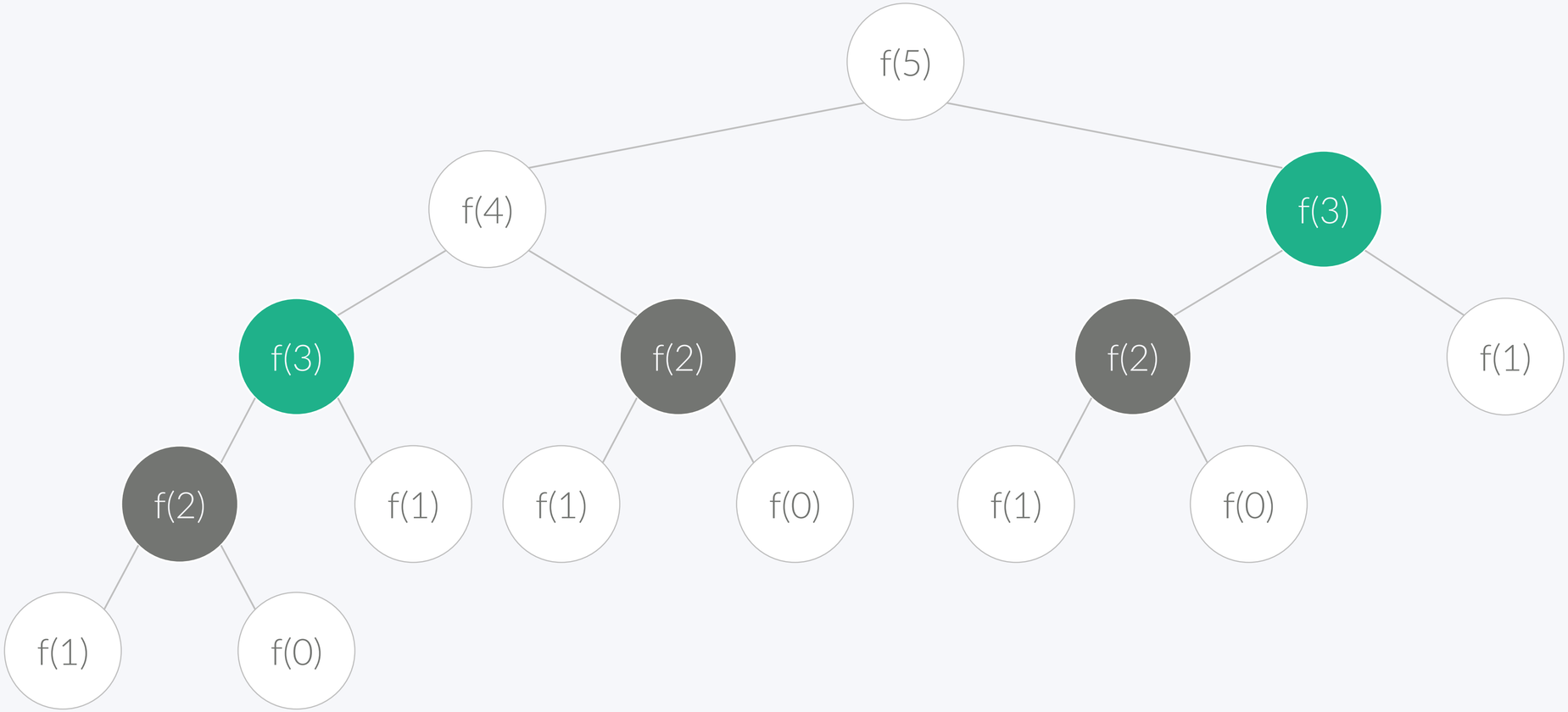
- fibonacci(5)를 호출한 경우 함수가 어떻게 호출되는지를 나타낸 그림



피보나치 수

Dynamic Programming

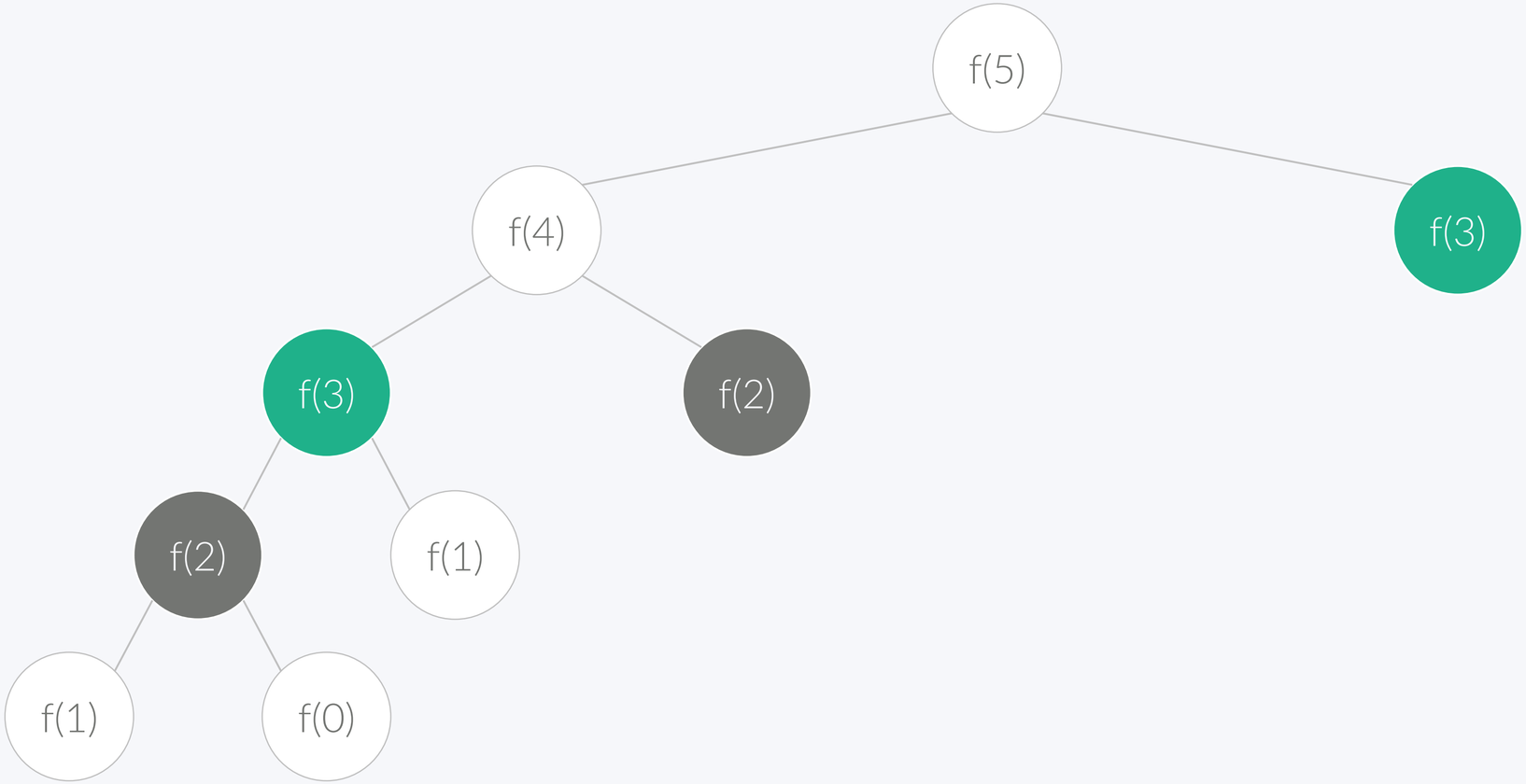
- 아래 그림과 같이 겹치는 호출이 생긴다.



피보나치 수

Dynamic Programming

- 한 번 답을 구할 때, 어딘가에 메모를 해놓고, 중복 호출이면 메모해놓은 값을 리턴한다.



피보나치 수

Dynamic Programming


```
int memo[100];
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    } else {
        memo[n] = fibonacci(n-1) + fibonacci(n-2);
        return memo[n];
    }
}
```

피보나치 수

Dynamic Programming


```
int memo[100];
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    } else {
        memo[n] = fibonacci(n-1) + fibonacci(n-2);
        return memo[n];
    }
}
```

피보나치 수

Dynamic Programming


```
int memo[100];
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    } else {
        if (memo[n] > 0) {
            return memo[n];
        }
        memo[n] = fibonacci(n-1) + fibonacci(n-2);
        return memo[n];
    }
}
```

다이나믹 프로그래밍

Dynamic Programming

- 다이나믹을 푸는 두 가지 방법이 있다.

1. Top-down
2. Bottom-up

Top-down

Dynamic Programming

1. 문제를 작은 문제로 나눈다.
2. 작은 문제를 푼다.
3. 작은 문제를 풀었으니, 이제 문제를 푼다.

Top-down

Dynamic Programming

1. 문제를 풀어야 한다.
    - fibonacci(n)
2. 문제를 작은 문제로 나눈다.
    - fibonacci(n-1)과 fibonacci(n-2)로 문제를 나눈다.
3. 작은 문제를 푼다.
    - fibonacci(n-1)과 fibonacci(n-2)를 호출해 문제를 푼다.
4. 작은 문제를 풀었으니, 이제 문제를 푼다.
    - fibonacci(n-1)의 값과 fibonacci(n-2)의 값을 더해 문제를 푼다.

Top-down

Dynamic Programming

- Top-down은 재귀 호출을 이용해서 문제를 쉽게 풀 수 있다.

Bottom-up

Dynamic Programming

1. 문제를 크기가 작은 문제부터 차례대로 푼다.
2. 문제의 크기를 조금씩 크게 만들면서 문제를 점점 푼다.
3. 작은 문제를 풀면서 왔기 때문에, 큰 문제는 항상 풀 수 있다.
4. 그러다보면, 언젠간 풀어야 하는 문제를 풀 수 있다.

Bottom-up

Dynamic Programming


```
int d[100];
int fibonacci(int n) {
    d[0] = 0;
    d[1] = 1;
    for (int i=2; i<=n; i++) {
        d[i] = d[i-1] + d[i-2];
    }
    return d[n];
}
```

Bottom-up

Dynamic Programming

1. 문제를 크기가 작은 문제부터 차례대로 푼다.
    - for (int i=2; i<=n; i++)
2. 문제의 크기를 조금씩 크게 만들면서 문제를 점점 푼다.
    - for (int i=2; i<=n; **i++**)
3. 작은 문제를 풀면서 왔기 때문에, 큰 문제는 항상 풀 수 있다.
    - d[i] = d[i-1] + d[i-2];
4. 그러다보면, 언젠간 풀어야 하는 문제를 풀 수 있다.
    - d[n]을 구하게 된다.