다이나믹 프로그래밍

190409(화) - 전현승

목차

- 다이나믹 프로그래밍
- 예시 – n번째 피보나치 수 구하기
- 11726 - 2×n 타일링 2
- 1149 - RGB거리

다이나믹 프로그래밍

- 일반적으로 주어진 문제를 풀기 위해서, 문제를 **여러 개의 하위 문제 (Subproblem, 부분 문제)**로 나누어 푼 다음, **그것을 결합하여 최종적인 목적에 도달**하는 것이다. (Wikipedia)

- Dynamic은 아무 의미 X (동적 프로그래밍이라는 용어도 거의 의미 X)
  - Richard Bellman : Dynamic이라는 단어가 멋있어보여서 사용했다
- "기억하며 풀기", "답 재활용하기" 등이 더 와닿는 듯

다이나믹 프로그래밍

- Overlapping Subproblem : 부분문제가 겹친다
- Optimal Substructure : 문제의 정답을 부분문제의 정답에서 구할 수 있다
- 답을 구하기 위해서 했던 계산을 또 하고 또 하고 계속해야 하는 류의 문제. 부분문제로 나누었을 때 이렇게 되면 된다

- 스펙트럼이 굉장히 넓다 (뇌피셜)
- 특정 알고리즘을 가리키는 용어가 아니라, 그냥 문제해결 생각방식으로 간주하는게 편함 (뇌피셜)

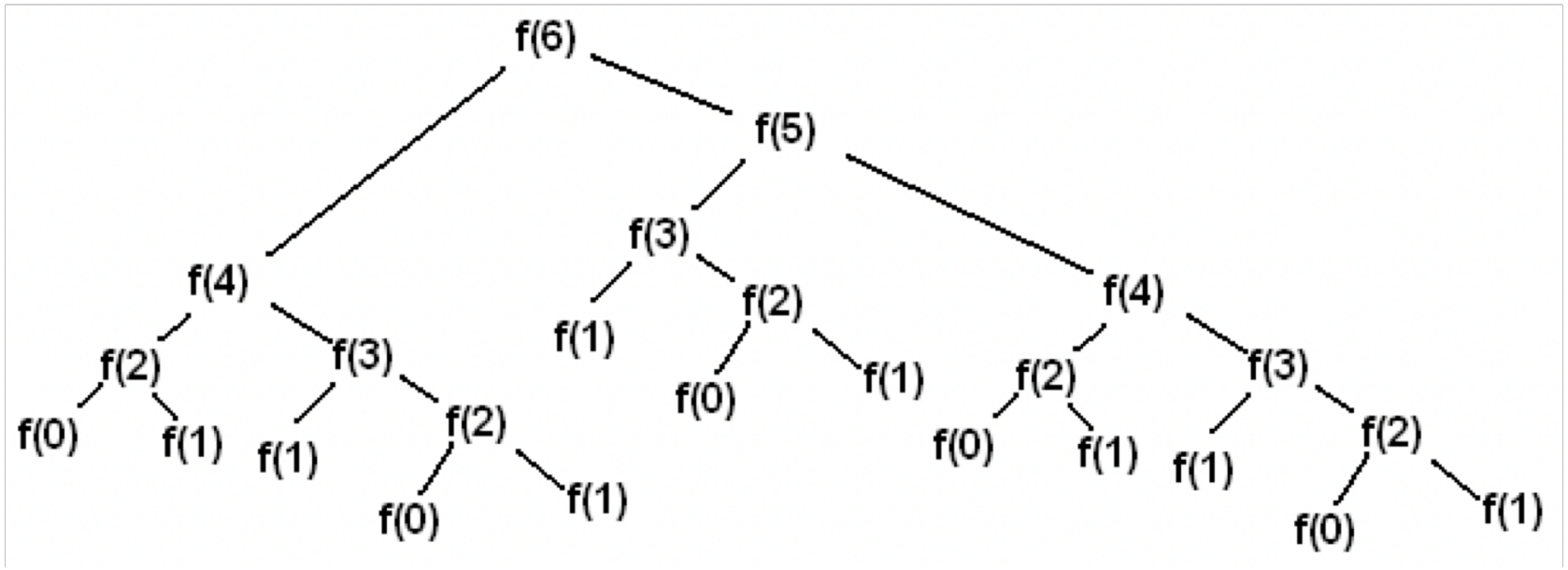
다이나믹 프로그래밍

- 구현 방식 : Top-down, Bottom-up

- Top-down : 문제를 부분문제로 나누고, 부분문제를 풀고, 문제를 푼다
  - 보통 재귀로 구현
- Bottom-up : 작은 부분문제부터 풀어나가다 보면 문제를 풀 수 있다
  - forloop

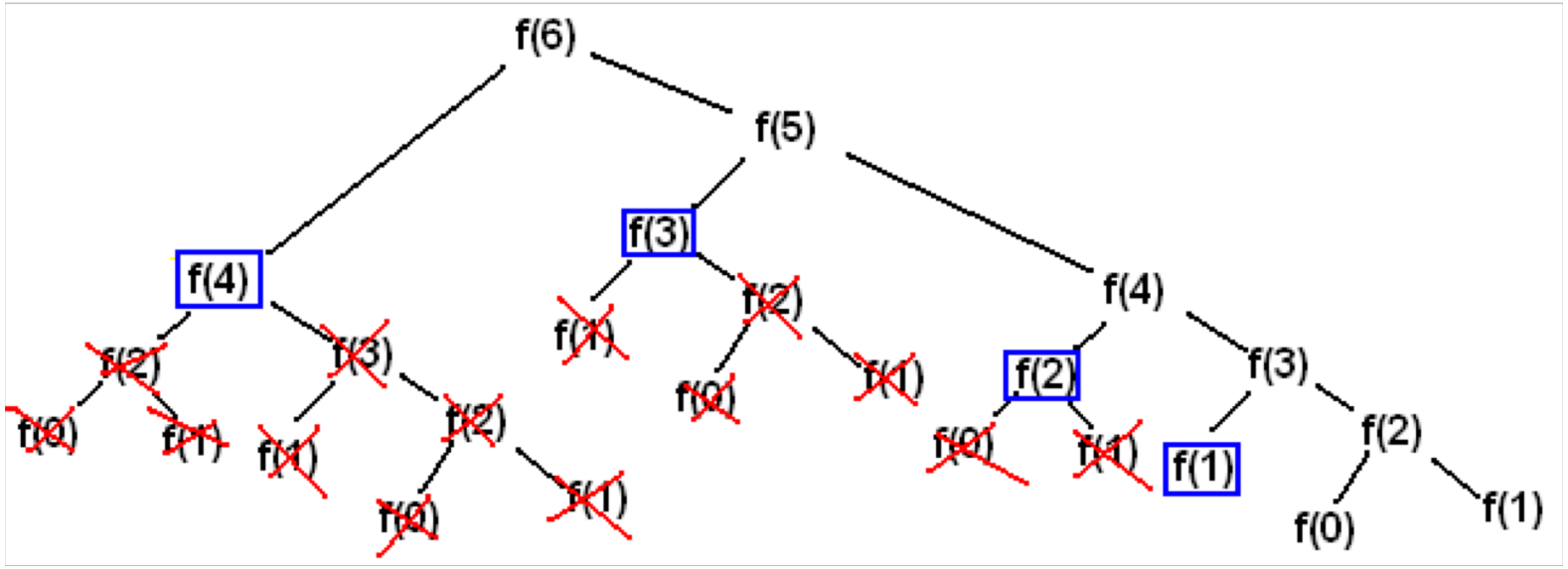
예시 – n번째 피보나치 수 구하기

- N=6인 경우를 예로 들면



예시 – n번째 피보나치 수 구하기

- 계산이 중복되는 부분이 많다



- f(4)를 구하기 위해 f(2) + f(3)을 두 번 똑같이 수행하고 있다.
- f(3), f(2)도 마찬가지

예시 – n번째 피보나치 수 구하기

- 중복되는 계산을 생략할 수 있지 않을까?
- f(4), f(3) 등 계산값을 저장해두고 풀면 이 연산을 줄일 수 있을 것
- "값을 메모해둔다" => Memoization

- Formal : "n번째 피보나치 수 구하는 문제"는 "n-2번째 피보나치 수 구하는 문제"와 "n- 1번째 피보나치 수 구하는 문제"로 나눌 수 있다.

예시 – n번째 피보나치 수 구하기

- Top-down

- f(6)를 구하는 큰 문제는 f(4)와 f(5)를 구하는 작은 문제로 나눌 수 있고, f(5)을 구하는 작은 문제는 f(3)와 f(4)을 구하는 더 작은 문제로 나눌 수 있고, ...
- f(4)를 이미 풀었으면 그 값을 참조하는 걸로 끝내고, 아직 안 풀었으면 풀어서 답 내놓고, 메모해놓는 것까지

예시 – n번째 피보나치 수 구하기

```
int memo[100] = {0, };  // Memoization 배열
int fibonacci(int n) {
        if(n < 2)
                return n;
        if (memo[n] != 0)
                // 메모해뒀는지 (한번 풀었던 문제인지) 확인 (0이 아니면 메모값이 있는 것)
                return memo[n]; // 메모 있으면 그거 쓰고
        memo[n] = fibonacci(n-1) + fibonacci(n-2); // 없으면 풀어서 메모해둔다
        return memo[n];
}
```

예시 – n번째 피보나치 수 구하기

- Bottom-up

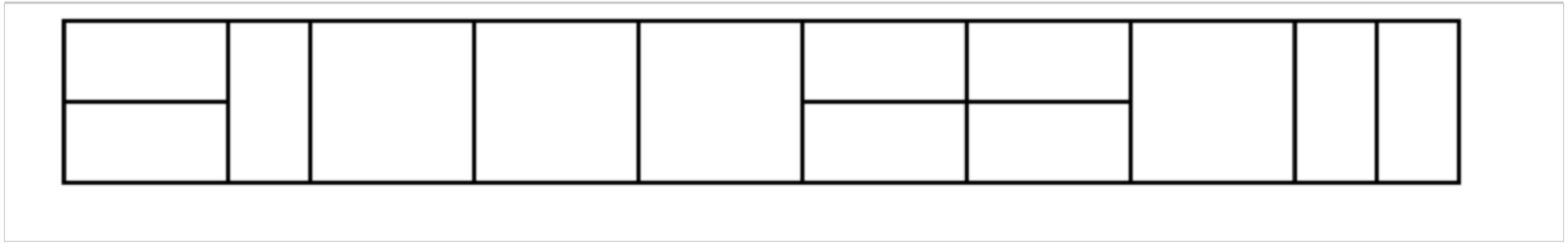
- f(1), f(2)를 구하면 f(3)을 구할 수 있고, f(2)와 f(3)을 구하면 f(4)를 구할 수 있다.
- f(1), f(2), f(3)을 메모해가면서 밑에서부터 올라가다 보면 f(6)도 구할 수 있다.

예시 – n번째 피보나치 수 구하기

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

11726 - 2×n 타일링 2

- 2×n 직사각형을 2×1과 2×2 타일로 채우는 방법의 수를 구하는 프로그램을 작성하시오. 아래 그림은 2×17 직사각형을 채운 한가지 예이다.



- 1초, n제한 1000

11726 - 2×n 타일링 2

- 엇갈리게 놓으면 안 된다는 것을 쉽게 알 수 있음

- 만약 2x(N-2) 칸까지 제대로 채웠다면 1x2 두개를 가로로 채우거나 2x2 블록을 오른쪽에 놓아 2xN 칸을 채울 수 있음

- 만약 2x(N-1) 칸까지 제대로 채웠다면 1x2 한 개를 세로로 채울 수 있음

- 위와 같이 채우는데 끝모양이 다 다르므로 다른 방법

11726 - 2×n 타일링 2

- 2*n번째 칸까지 채우는 방법의 수 =

  2*(n-2)번째 칸까지 채우는 방법의 수 * 2 +

  2*(n-1)번째 칸까지 채우는 방법의 수

11726 - 2×n 타일링 2

- Top-down

```
int f(int x) {
        if(x == 1) return 1;
        if(x == 2) return 3;
        return (2*f(x-2) + f(x-1)) % 10007;
}


int main() {
        int n; cin >> n;
        cout << f(n) << endl;
        return 0;
}
```

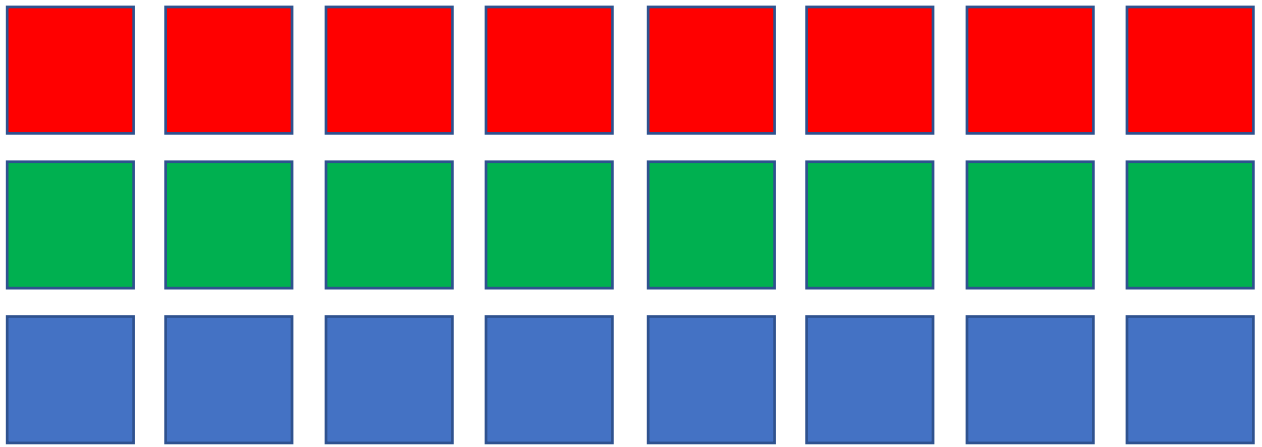
11726 - 2×n 타일링 2

- Bottom-up

```
dp[1] = 1;
dp[2] = 3;
for (int i = 3; i <= n; i++) {
        dp[i] = 2 * dp[i-2] + dp[i-1];
        dp[i] %= 10007;
}
```

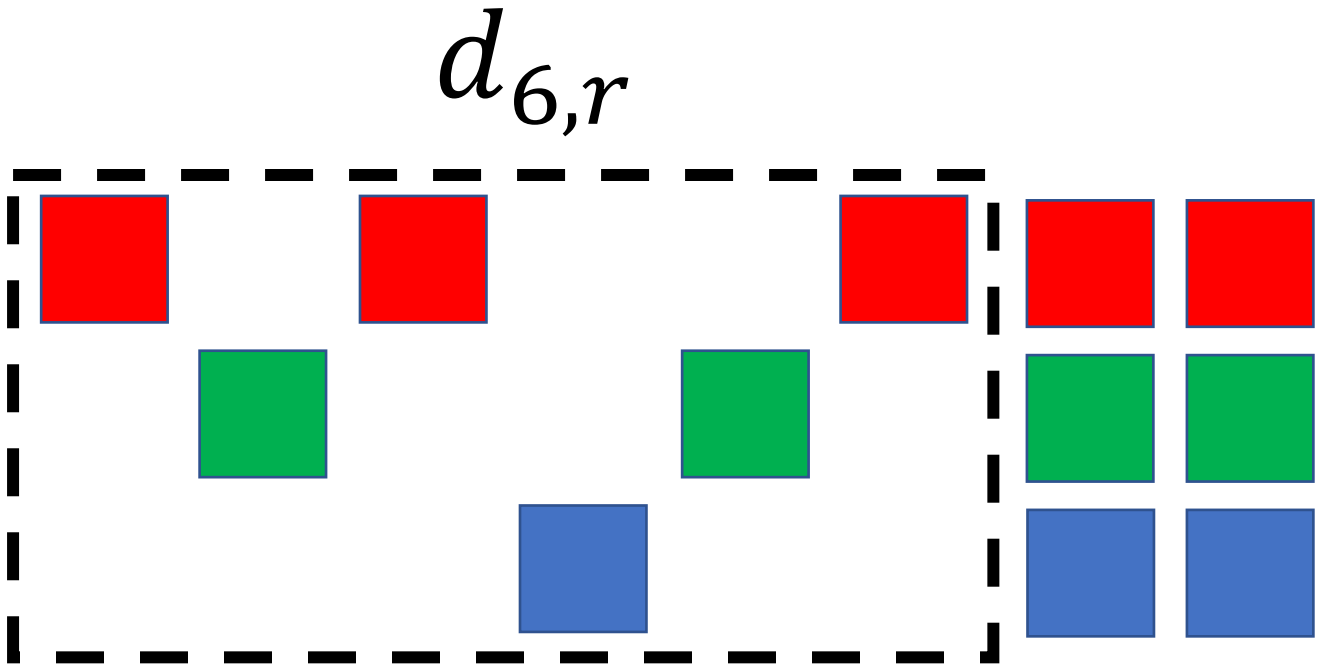
1149 - RGB거리

- N개의 집이 일렬로 나란히 붙어 있고, 각 집은 빨강, 초록, 파랑 중 하나의 색으로 칠해야 한다
- 각 집마다 어떤 색으로 칠하느냐에 따라 비용이 다름
- 인접한 집은 색이 달라야 함. 비용의 최솟값?



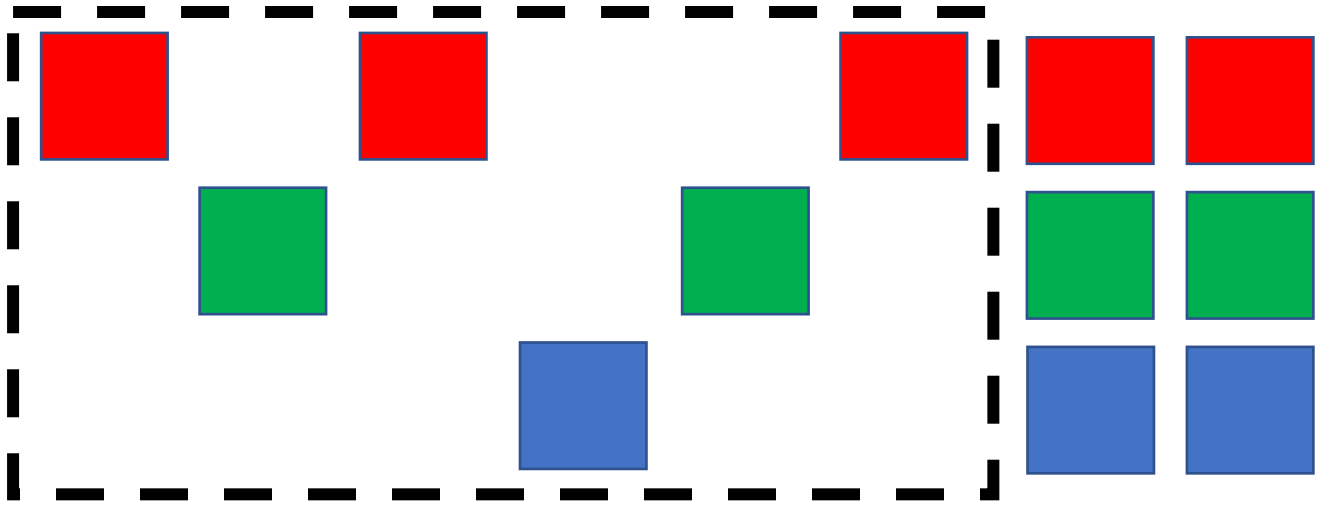
1149 - RGB거리

- $d_{i,c}$ = 1번부터 i번째 집까지 인접한 집끼리 색이 다르고, i번째 집의 색이 c인 경우 최소 비용



1149 - RGB거리

- $d_{i,r} = \min(d_{i-1,b},\ d_{i-1,g})$
- $d_{i,g} = \min(d_{i-1,b},\ d_{i-1,r})$
- $d_{i,b} = \min(d_{i-1,r},\ d_{i-1,g})$
- $ans = \min(d_{n,r}, d_{n,g}, d_{n,b})$



끝