03. Dynamic Programming 1

다이나믹 프로그래밍이란?

Dynamic Programming, 줄여서 DP라 부른다.

동적 계획법 - 문제를 어떤 형태로 변형시켜 쉽게 푸는 방법을 의미한다.

큰 문제를 작은 문제로 나눠서 푸는 알고리즘

DP의 2가지 필수 조건

1. Overlapping Subproblem

-> 큰 문제가 작은 문제로 세분화될 수 있고, 같은 방법으로 풀릴 때

2. Optimal Substructure

-> 큰 문제의 정답을 작은 문제의 정답들로 구성할 수 있을 때

Overlapping Subproblem

* 큰 문제가 작은 문제로 세분화될 수 있고, 같은 방법으로 풀릴 때

피보나치 수

$$\mathrm{Fib}(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) & \text{otherwise.} \end{cases}$$

Fib(n-1) = Fib(n-2) + Fib(n-3) Fib(n-2) = Fib(n-3) + Fib(n-4)

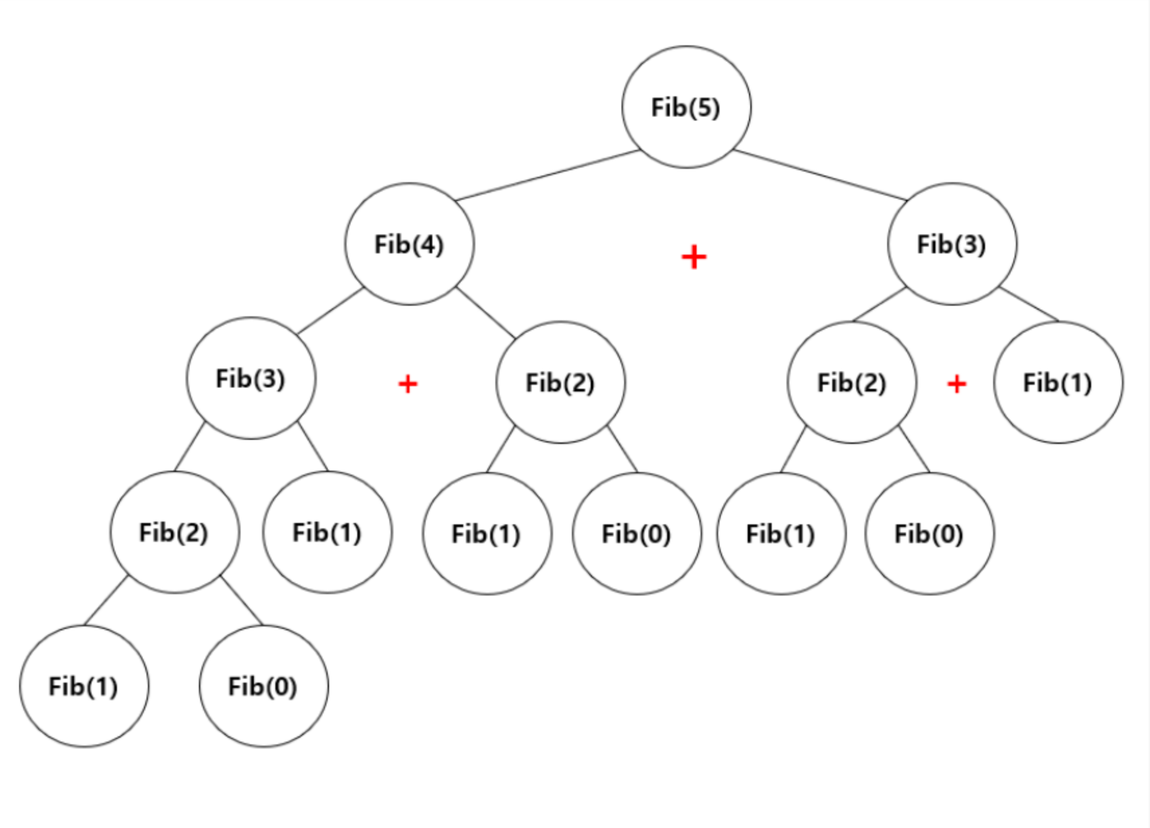
-> 큰 문제를 작은 문제의 합으로 구할 수 있다!

Optimal Substructure

* 큰 문제의 정답이 작은 문제들의 정답들로 구성할 수 있을 때

피보나치 수

Fib(5)를 구하는 과정 (재귀 호출)



피보나치의 재귀 함수 코드

```
int fib(int n)
{
        if (n == 0) return 0;
        else if (n == 1) return 1;
        return fib(n - 1) + fib(n - 2);
}
```

문제점 : 너무 잦은 함수 호출

-> 어차피 매번 답은 똑같은데, 시간을 단축시킬 방법은 없을까?

메모이제이션 (Memoization)

동일한 계산을 반복해야 할 때, 이전에 계산한 값을 메모리에 저장함

반복 수행 x -> 시간 복잡도 단축!

메모이제이션 (Memoization)

과정

1. 작은 정답들을 저장할 수 있는 dp table을 만든다.

2. 문제를 해결하면서 작은 정답이 나올 때마다 저장

3. 이전에 해결한 정답을 요구하면, 다시 계산하지 않고 리턴!

* Overlapping Subproblem 조건에 해당(큰 문제 해결 방법 = 작은 문제 해결 방법)

메모이제이션을 활용한 피보나치 코드

```
int dp[100];
int fib(int n)
{
        if (n == 0) return 0;
        else if (n == 1) return 1;
        else if (dp[n] != -1) return dp[n];
        return dp[n] = fib(n - 1) + fib(n - 2);
}
int main()
{
        ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);
        memset(dp, -1, sizeof(dp));
        cout << fib(5);
}
```

1. Top-Down 방법

큰 문제를 작은 문제로 나누고, 작은 문제의 정답을 이용해 큰 문제를 푸는 방법

위의 재귀함수 코드는 fib(n)을 먼저 넣어 작은 정답을 찾아나가는 구성

보통 재귀함수로 구현되면 Top-Down으로 생각하면 된다!

2. Bottom-Up 방법

1) 크기가 작은 문제들부터 차례대로 푼다.

2) 점점 크기를 키워나가며 문제를 푼다.

3) 1,2 번을 반복하다보면, 문제에서 원하는 정답을 구할 수 있다.

* 작은 문제들의 정답을 알고 있기 때문에 큰 문제의 정답도 구할 수 있다.

Top-Down vs Bottom-Up

거의 대부분의 DP 문제는 두 방법 모두 풀이법이 있다.

처음에는 헷갈리는 재귀(Top-Down) 방법 보다, 눈에 보이는 Bottom-Up 방식을 추천!

피보나치 수 7

BOJ 15624번

위 피보나치 코드를 이용해 구할 수 있다.

피보나치 수 7

BOJ 15624번

위 피보나치 코드를 이용해 구할 수 있다.

단, 구하는 과정에서의 오버플로우에 주의!

* 100번도 못가서 long long 범위를 초과하게 된다.

```
86번 째 피보나치 수 : 42019614072748967 3
87번 째 피보나치 수 : 67989163763861225 8
88번 째 피보나치 수 : 11000877836610193 1
89번 째 피보나치 수 : 17799794160047141 89
90번 째 피보나치 수 : 28800671943708161 20
91번 째 피보나치 수 : 46600466103755303 09
92번 째 피보나치 수 : 75401138047463464 29
93번 째 피보나치 수 : -62465836585876748 78
94번 째 피보나치 수 : 12935301461586715 51
95번 째 피보나치 수 : -49530535124290033 27
```

거스름돈

BOJ 14916번

2원, 5원 동전만 이용해 N원을 만드려 한다.

이때, 동전의 최소 개수는?

거스름돈

BOJ 14916번

5원을 최대한 많이 사용하는 것이 이득일까?

N = 12 -> 5 + 5 + 2 -> 3개

N = 13 -> 5 + 5 + 3 -> ?

거스름돈

BOJ 14916번

풀이 1. 다이나믹 프로그래밍

dp[i] : 2, 5원으로 i원을 만들 때 동전의 최소 개수라고 하자.

초기 값 : dp[2] = dp[5] = 1

거스름돈

BOJ 14916번

풀이 1. 다이나믹 프로그래밍

i원을 만드려면 i-2원 또는 i-5 원을 만들 수 있어야 한다.

i-2원을 만드려면 i-4원 또는 i-7원을 만들 수 있어야 한다.

i-5원을 만드려면 i-7원 또는 i-10원을 만들 수 있어야 한다.

-> 피보나치 수와 비슷한 형식!

거스름돈

BOJ 14916번

점화식 세워보기

초기 값 -> dp[i] = 1 (i = 2 or i = 5)

i원을 만드려면 i-2원 또는 i-5 원을 만들 수 있어야 한다. -> dp[i] = min(dp[i - 2], dp[i - 5]) + 1

거스름돈

BOJ 14916번

점화식을 이용한 코드

```
cin >> n;
memset(dp, -1, sizeof(dp));
dp[2] = 1; dp[4] = 2; dp[5] = 1;
for (int i = 6; i <= n; i++) {
        if (dp[i - 5] != -1) dp[i] = dp[i - 5] + 1;
        if (dp[i - 2] != -1) {
                if (dp[i] == -1) dp[i] = dp[i - 2] + 1;
                else dp[i] = min(dp[i], dp[i - 2] + 1);
        }
}
```

거스름돈

BOJ 14916번

풀이 2. 수학

N을 5로 나눈 나머지가 짝수일 때 -> 나머지는 2원으로 거슬러 준다.

N을 5로 나눈 나머지가 홀수일 때 -> 나머지에 5를 더한 후 2원으로 거슬러 준다.

ex) N = 13, 몫 : 2 나머지 : 3 -> 몫 : 1, 나머지 : 8 -> 1 + (8 / 2) = 5개

1로 만들기

BOJ 1463번

주어진 세 가지 연산을 이용해 1을 만드려고 할 때, 연산을 사용하는 횟수의 최솟값은?

1. X가 3으로 나누어 떨어지면, 3으로 나눈다.

2. X가 2로 나누어 떨어지면, 2로 나눈다.

3. 1을 뺀다.

1로 만들기

BOJ 1463번

점화식 세워보기

dp[i] : i를 1로 만들 때의 최소 연산 횟수라고 하자.

1. X가 3으로 나누어 떨어지면, 3으로 나눈다. -> dp[i] = dp[i / 3] + 1 (i % 3 = 0)

2. X가 2로 나누어 떨어지면, 2로 나눈다. -> dp[i] = dp[i / 2] + 1 (i % 2 = 0)

3. 1을 뺀다. -> dp[i] = dp[i-1]

위 3가지 경우 중 가장 낮은 값을 dp[i]로 초기화 한다.

1로 만들기

BOJ 1463번

점화식을 이용한 코드

```
cin >> n;
memset(dp, -1, sizeof(dp));
dp[1] = 0;
for (int i = 2; i <= n; i++)
{
        int tmp = dp[i - 1] + 1;
        if (i % 2 == 0) tmp = min(tmp, dp[i / 2] + 1);
        if (i % 3 == 0) tmp = min(tmp, dp[i / 3] + 1);
        dp[i] = tmp;
}
cout << dp[n] << "\n";
```

계단 오르기

BOJ 2579번

주어진 규칙에 따라 계단을 오를 때 얻을 수 있는 점수의 최댓값을 구하는 문제

1. 계단은 한 번에 최대 2칸 올라갈 수 있다.

2. 연속된 3개의 계단을 모두 밟아서는 안 된다.

3. 마지막 도착 계단은 반드시 밟아야 한다.

계단 오르기

BOJ 2579번

2. 연속된 3개의 계단을 모두 밟아서는 안 된다.

2번 조건이 없으면 모든 계단의 점수를 더한 것이 최댓값이 된다.

계단 오르기

BOJ 2579번

최대한 많은 계단을 밟는게 최선일까?

-> 가장 많이 밟으려면, 앞에 2칸 밟고 한 칸 건너뛰고를 반복하면 된다.

하지만 3번째 칸이 매우 큰 수라면? 3번째 칸을 반드시 밟아야 한다.

계단 오르기

BOJ 2579번

백트래킹으로 가능할까?

go(index, continue) : 현재 index 번 째 계단이고, 그 전 계단을 밟았는지 확인

그 전 칸을 밟았다면 2칸을 건너뛰고, 아니라면 1칸, 2칸 모두 가능함

매 계단마다 2가지 선택을 해줘야 한다 -> O(2^N) N이 최대 300이므로 불가능!

계단 오르기

BOJ 2579번

믿을 건 dp뿐!

-> dp 조건이 성립한다면, O(n)으로 가능하다.

1. Overlapping Subproblem — i번째 계단까지 밟으려면?

   -> i-1번째 계단 or i-2번째 계단까지밟아야 한다!

2. Optimal Substructure — i번째 계단까지의 최댓값?

   -> i-1 번째 or i-2 번째 계단까지의 최댓값 + i번째 계단 점수

계단 오르기

BOJ 2579번

여기까진 알겠는데.. 연속 처리는? 2차원 배열 이용

dp[i][0] : i-1번째 계단을 밟지 않은 i번째 계단까지의 최댓값

dp[i][1] : i-1번째 계단을 밟은 i번째 계단까지의 최댓값

계단 오르기

BOJ 2579번

점화식 세워보기

dp[i][0] : i-1번째 계단을 밟지 않은 i번째 계단까지의 최댓값

-> max(dp[i-2][0], dp[i-2][1]) + score[i]

dp[i][1] : i-1번째 계단을 밟은 i번째 계단까지의 최댓값

-> dp[i-1][0] + score[i]

2 x N 타일링

BOJ 11726번

2×n 크기의 직사각형을 1×2, 2×1 타일로 채우는 방법의 수를 구하는 문제

2 x N 타일링

BOJ 11726번

1, 2, 3 더하기

BOJ 9095번