# 동적 계획법

25.04.28 (월) 오후 5시 ~ 7시

# 목차

- **동적 계획법(Dynamic Programming, 이하 DP)**
  - A. **개념**
  - B. **문제 풀이 (10문제)**

# A. 개념

# DP A. 개념 | 01 정의와 개념 소개

- 동적계획법(Dynamic Programming, 이하 DP)은
  - 복잡한 문제를 작은 하위 문제로 나누고
  - 작은 하위 문제를 해결한 뒤
  - 작은 하위 문제들의 답을 이용해 복잡한 문제의 답을 구하는 기법이다.
- DP 기법을 이용해 풀 수 있는 문제는 **2가지 특징**이 나타난다.
  - 작은 문제들의 해답으로 큰 문제의 답을 만들어 나간다. → **최적 부분 구조**(Optimal Substructure)
  - 같은 하위 문제가 여러 번 반복해서 나타난다. → **중복되는 부분 문제**(Overlapping Subproblem)

# DP A. 개념 | 01 정의와 개념 소개

- **최적 부분 구조** (Optimal Substructure)
  - **큰 문제의 최적 해가, 작은 문제의 최적 해로부터 구성**될 수 있어야 한다.
  - 작은 문제의 최적 해를 이용해 큰 문제의 최적 해를 구성할 수 없다면, DP로는 못 푸는 문제이다.
    - 작은 문제의 최적 해를 수정해 큰 문제의 최적 해를 만들어야 한다면, 그건 DP로는 못 푸는 문제라는 의미다.
  - 예) 피보나치 수열

    $a_n = a \times a_0 + b \times a_1$ $(a_0 = 0, a_1 = 1)$로 나타낼 수 있다.

    $a_n$라는 큰 문제는 $a_0, a_1$라는 작은 문제의 해로 구성된다.

# DP A. 개념 | 01 정의와 개념 소개

- **중복되는 부분 문제** (Overlapping Subproblems)
  - **동일한 하위 문제를 여러 번 참조**한다.
  - 같은 입력 값을 가진 하위 문제는 항상 같은 결과가 나오는데, **메모이제이션 기법**을 활용해 이를 최적화할 수 있다.
    - 메모이제이션 기법은 **이미 계산한 하위 문제의 결과를 저장해서, 같은 계산을 반복하지 않도록 하는 기법**이다.
    - 중복 계산을 제거함으로써, 단순 재귀 구조에서 발생하는 지수 시간 복잡도를 선형 시간으로 줄일 수 있다.

  - 예) 피보나치 수열

    $a_n = a \times a_0 + b \times a_1\ (a_0 = 0, a_1 = 1)$에서 $a_0$는 a번, $a_1$은 b번 참조된다.

    메모이제이션 기법을 활용하면 참조 횟수를 줄일 수 있다.

# **DP** A. 개념 | 02 두 가지 구현 방식

- Top-down 과 Bottom-up 방식으로 DP 문제를 풀게 된다.

- Top-down 방식
  - **재귀와 메모이제이션**을 사용하여 구현한다.
  - 큰 문제를 호출하면서 필요한 작은 문제를 재귀 호출해서 푼다.
  - 하위 문제 결과를 캐시(dp 배열)에 저장해서 중복 계산을 막는다.
- Bottom-up 방식
  - **반복문**으로 구현한다.
  - 작은 문제부터 차례로 풀면서, 큰 문제를 계산한다.
  - 모든 하위 문제를 차례대로 계산한다.
  - 보통 dp 배열을 0부터 채워나간다.

# DP A. 개념 | 03 접근 방법

1. **DP 배열 정의** 🔥
    - **문제를 해결하기 위해 필요한 상태를 '정확하게' 정의한다.**
    - 이 단계에서 가장 중요한 것은 "dp 배열이 무엇을 의미하는가"를 자연어로 설명할 수 있어야 한다.
        - 예) dp[N]는 피보나치 수열의 N번째 항이다.
2. 초기값 설정
    - 가장 작은 하위 문제에 대한 답을 직접 정의한다. 문제의 가장 기초가 되는 상황은 직접 정의해줘야 한다.
        - 예) 피보나치 수열의 0번째 항과 1번째 항은 각각 0과 1이다. dp[0] = 0, dp[1]인 것이다.
3. **점화식 세우기** 🔥
    - **큰 문제와 작은 문제 간의 관계(=점화식)을 찾는다.**
    - **큰 문제를 작은 문제의 결과로 표현하는 관계식(=점화식)을 찾는다.**
        - 예) 피보나치 수열은 다음과 같은 점화 관계로 정의할 수 있다. dp[N] = dp[N – 1] + dp[N – 2]
4. 구현
    - Bottom-up 방식(반복문)이나 Top-down 방식(재귀 + 메모이제이션)으로 구현한다.

# B. 문제 풀이

# DP B. 문제 풀이 | 00 개요

- 이론적으로 Optimal Substructure, Overlapping Subproblem을 만족하면 DP로 풀 수 있는 문제이다.
  - 하지만 매번 확인하고 증명하는 것은 현실적으로 어렵다.
  - 그렇기에 실제 코딩테스트나 문제 풀이에서는, "이거 DP 문제일까?"를 감각적으로 잡아내는 능력이 훨씬 더 중요하다.
- DP 문제를 푸는 감각을 키워야 한다.
  - 문제를 많이 풀어보고, **"지금까지 풀었던 경험으로 이 문제도 DP로 풀 수 있을까?"** 를 빠르게 판단하는 감이 중요하다.

- DP로 풀리는 문제 10문항을 같이 풀어볼 텐데, 아래 두 가지 질문에 초점을 맞추셨으면 좋겠습니다.

1. **DP 배열을 어떻게 정의하면 좋을까?**
2. **점화식을 어떻게 작성하면 될까?**

# DP B. 문제 풀이 | 01 백준 1463번 1로 만들기

- **문제요약**
  - 정수 X에 사용할 수 있는 연산은 다음과 같이 세 가지 이다.
    1. X가 3으로 나누어 떨어지면, 3으로 나눈다.
    2. X가 2로 나누어 떨어지면, 2로 나눈다.
    3. 1을 뺀다
  - 정수 N이 주어졌을 때, 위와 같은 연산 세 개를 적절히 사용해서 1을 만들려고 한다. 연산을 사용하는 횟수의 최솟값을 출력하시오.
- **입력**
  - 정수 N이 주어진다. N은 1보다 크거나 같고, 1e6보다 작거나 같다.
- **출력**
  - 첫째 줄에 연산을 하는 횟수의 최솟값을 출력한다.

# DP B. 문제 풀이 | 01 백준 1463번 1로 만들기

- **풀이**
  - 예제를 보면서 감을 잡아보자.
  - 2는 3번째 연산(1 빼기)을 한번 적용하면 됨.
  - 10 이라는 문제를 풀기 위해서는
    - 3번째 연산(1 빼기)을 적용하고 : 9
    - 1번째 연산(3으로 나누기)을 적용하고 : 3
    - 1번째 연산(3으로 나누기)을 적용하면 됨 : 1
  - 10 이라는 문제를 풀기 위해 부분 문제(9)을 풀어야 한다는 걸 파악할 수 있음.
    - 10 이라는 문제를 풀기 위해 또 다른 부분 문제(5)를 푼 결과를 선택하지 않는 이유는 비용이 더 크거나 같기 때문임.

# DP B. 문제 풀이 | 01 백준 1463번 1로 만들기

- 추가적으로 N = 1, 2, 3, 4, 5, 6일 때 어떤 양상으로 진행되는지 확인해보자.
  - N = 1일 때, 굳이 연산을 할 필요가 없다. (애초에 N은 1보다 클 때 정의된다.)
  - N = 2일 때, 3번 연산(1을 빼기)을 하면 된다.
  - N = 3일 때, 1번 연산(3으로 나누기)을 한 번하면 된다.
  - N = 4일 때, 아래 결과 중 최솟값을 선택한다.
    - 3번 연산(1을 빼기)을 하면 된다. **그러면 N = 3일 때 푼 문제를 참고하면 된다.**
    - 2번 연산(2로 나누기)을 하면 된다. **그러면 N = 2일 때 푼 문제를 참고하면 된다.**
  - N = 5일 때, 아래 결과 중 최솟값을 선택한다.
    - 3번 연산(1을 빼기)을 하면 된다. **그러면 N = 4 일 때 푼 문제를 참고하면 된다.**
  - N = 6일 때, 아래 결과 중 최솟값을 선택한다.
    - 1번 연산(3으로 나누기)을 하면 된다. **그러면 N = 2일 때 푼 문제를 참고하면 된다.**
    - 2번 연산(2으로 나누기)을 하면 된다. **그러면 N = 3일 때 푼 문제를 참고하면 된다.**
    - 3번 연산(1을 빼기)을 하면 된다. **그러면 N = 5일 때 푼 문제를 참고하면 된다.**

# DP B. 문제 풀이 | 01 백준 1463번 1로 만들기

- dp 배열을 다음과 같이 정의하자.
  - dp[i] = (i를 1로 만들기 위한 연산의 최소 횟수)
- 초기값을 설정하자.
  - dp[1] = 0
- 점화식은 다음과 같이 정의할 수 있다.
  - dp[N] = min(dp[N – 1] + 1, dp[N/2] + 1, dp[N/3] + 1)
  - 당연히 dp[N/2] + 1 는 N이 2로 나누어 떨어질 때 정의되고, dp[N/3 + 1]도 마찬가지다.
- Bottom-up 방식으로 문제를 풀이(설명)했으니 반복문으로 구현하겠다.

# DP B. 문제 풀이 | 01 백준 1463번 1로 만들기

- **구현**

Line 5

dp 배열을 정의한다.

**dp[n] = (1로 만들기 위한 연산의 최소 횟수)**

Line 11 ~ 15

dp[N] = min(dp[N – 1] + 1, dp[N/2] + 1, dp[N/3] + 1)

**dp[N]이라는 문제를 풀려면**

**dp[N – 1], dp[N/2], dp[N/3] 문제를 참고하면 된다.**

**그 중 최솟값을 선택하면 최적인 선택이 된다.**

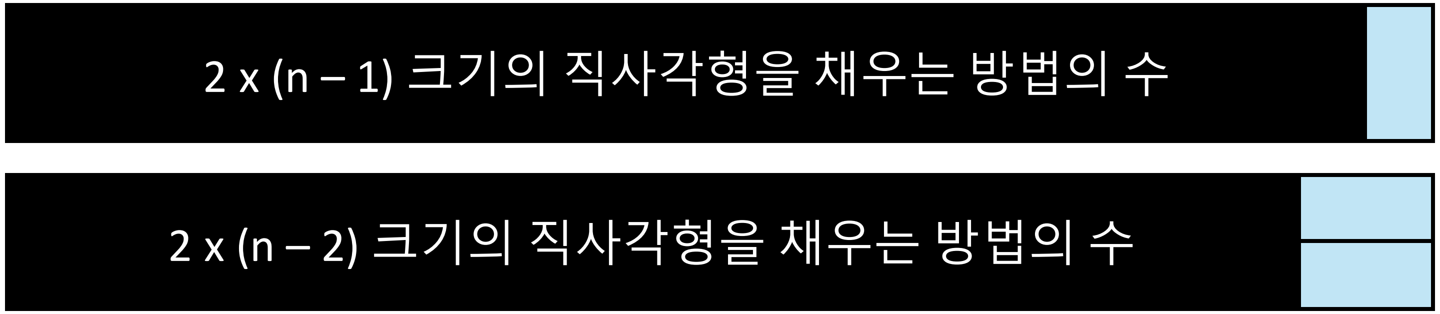
```cpp
1  #include <bits/stdc++.h>
2  using namespace std;
3  #define ll long long
4
5  ll N, dp[1'000'001];
6
7  int main(){
8      ios::sync_with_stdio(false);
9      cin >> N;
10     dp[1] = 0;
11     for(int i = 2; i <= N; i++){
12         dp[i] = dp[i-1] + 1;
13         if(i % 2 == 0) dp[i] = min(dp[i], dp[i/2] + 1);
14         if(i % 3 == 0) dp[i] = min(dp[i], dp[i/3] + 1);
15     }
16     cout << dp[N] << "\n";
17     return 0;
18 }
```

# DP B. 문제 풀이 | 02 백준 11726번 2×n 타일링

- **문제요약**
  - 2×n 크기의 직사각형을 1×2, 2×1 타일로 채우는 방법의 수를 구하는 프로그램을 작성하시오.
- **입력**
  - 첫째 줄에 n이 주어진다. (1 ≤ n ≤ 1,000)
- **출력**
  - 첫째 줄에 2×n 크기의 직사각형을 채우는 방법의 수를 10,007로 나눈 나머지를 출력한다.

# DP B. 문제 풀이 | 02 백준 11726번 2×n 타일링

- **풀이**
  - 예제를 보면서 감을 잡아보자.
  - n = 2일 때, 2 x 2 크기의 보드를 타일로 채우는 방법의 수는 2개이다.
  - n = 9일때, 2 x 9 크기의 보드를 타일로 채우는 방법의 수는 55개 라는데, 2x8일 때와 2x7일 때로 표현될 수 있지 않을까?
    - {2x9를 타일로 채우는 방법} = {2x8을 타일로 채우는 방법의 수} + {2x7을 타일로 채우는 방법의 수}
    - **2x8의 방법으로 만든 2x9 타일링과 2x7의 방법으로 만든 2x9 타일링에 교집합이 없다(독립)는 것을 이해 해야함.**
    - 그래야 2x6의 방법으로 만든 경우의 수를 2x9 타일링에서 취급 안 하는지 이해할 수 있음.
  - 즉, 2 x n 크기의 직사각형을 채우는 방법의 수는 아래의 합으로 구성됨.



# DP B. 문제 풀이 | 02 백준 11726번 2×n 타일링

- **풀이**
  - dp 배열을 정의해봅시다.
    - dp[n] = (2xn 크기의 직사각형을 채우는 방법의 수)
  - 초기값을 설정해봅시다.
    - dp[1] = 1, dp[2] = 2
  - 점화식을 세워봅시다.
    - dp[N] = dp[N – 1] + dp[N – 2]
  - 구현합시다.
    - Bottom-up 방식으로 문제를 풀이(설명)했으니 반복문으로 구현하겠다.

# DP B. 문제 풀이 | 02 백준 11726번 2×n 타일링

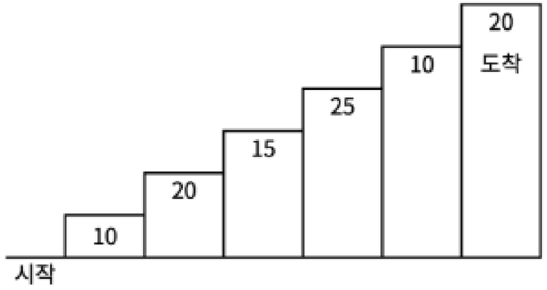
- **구현**

```cpp
#include <bits/stdc++.h>
using namespace std;

int dp[1001]; // dp[i] = 2 x i 크기의 직사각형을 채우는 방법의 수
int main(){
    cin.tie(0);
    cout.tie(0);
    ios::sync_with_stdio(false);

    dp[1] = 1;
    dp[2] = 2;
    int n; cin >> n;
    for(int i = 3; i <= n; i++){
        dp[i] = (dp[i-1] + dp[i-2])%10007;
    }
    cout << dp[n];
    return 0;
}
```

# DP B. 문제 풀이 | 03 백준 2579번 계단 오르기

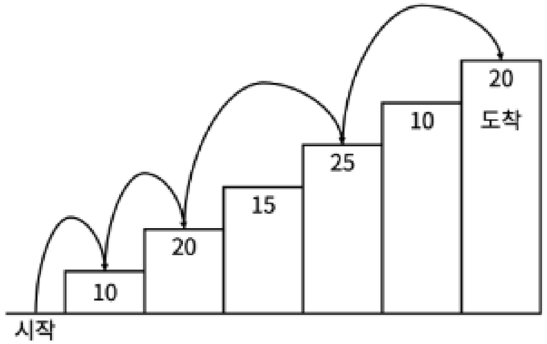
- **문제**

계단 오르기 게임은 계단 아래 시작점부터 계단 꼭대기에 위치한 도착점까지 가는 게임이다. <그림 1>과 같이 각각의 계단에는 일정한 점수가 쓰여 있는데 계단을 밟으면 그 계단에 쓰여 있는 점수를 얻게 된다.




<그림 1>

예를 들어 <그림 2>와 같이 시작점에서부터 첫 번째, 두 번째, 네 번째, 여섯 번째 계단을 밟아 도착점에 도달하면 총 점수는 10 + 20 + 25 + 20 = 75점이 된다.




<그림 2>

# DP B. 문제 풀이 | 03 백준 2579번 계단 오르기

- **문제**

계단 오르는 데는 다음과 같은 규칙이 있다.

1. 계단은 한 번에 한 계단씩 또는 두 계단씩 오를 수 있다. 즉, 한 계단을 밟으면서 이어서 다음 계단이나, 다음 다음 계단으로 오를 수 있다.
2. 연속된 세 개의 계단을 모두 밟아서는 안 된다. 단, 시작점은 계단에 포함되지 않는다.
3. 마지막 도착 계단은 반드시 밟아야 한다.

따라서 첫 번째 계단을 밟고 이어 두 번째 계단이나, 세 번째 계단으로 오를 수 있다. 하지만, 첫 번째 계단을 밟고 이어 네 번째 계단으로 올라가거나, 첫 번째, 두 번째, 세 번째 계단을 연속해서 모두 밟을 수는 없다.

각 계단에 쓰여 있는 점수가 주어질 때 이 게임에서 얻을 수 있는 총 점수의 최댓값을 구하는 프로그램을 작성하시오.

## 입력

입력의 첫째 줄에 계단의 개수가 주어진다.

둘째 줄부터 한 줄에 하나씩 제일 아래에 놓인 계단부터 순서대로 각 계단에 쓰여 있는 점수가 주어진다. 계단의 개수는 300이하의 자연수이고, 계단에 쓰여 있는 점수는 10,000이하의 자연수이다.

## 출력

첫째 줄에 계단 오르기 게임에서 얻을 수 있는 총 점수의 최댓값을 출력한다.

# DP B. 문제 풀이 | 03 백준 2579번 계단 오르기

- **문제 요약**
  - 계단이 N개 있다. (1번 ~ N번)
  - 각 계단에는 점수가 있고, 밟으면 그 점수를 얻는다.
  - 계단 오르는 규칙
    1. 한 번에 1칸 또는 2칸 씩 오를 수 있다.
    2. 연속된 세 계단을 밟을 수 는 없다.
    3. 마지막 계단은 반드시 밟아야 한다.
- **입력**
  - 계단의 개수(300 이하의 자연수)와 각 계단에 쓰여 있는 점수(10,000이하의 자연수)가 차례대로 주어진다.
- **출력**
  - 계단 오르기 게임에서 얻을 수 있는 총 점수의 최댓값을 출력해라.

5 MIN

# DP B. 문제 풀이 | 03 백준 2579번 계단 오르기

- **첫번째 풀이**

예제를 보면서 문제를 어떻게 풀면 좋을지 고민해보자.

첫번째 계단까지 오를 때, 최댓값은 10이다.

두번째 계단까지 오를 때, 최댓값은 10 + 20, 즉 30이다.

세번째 계단까지 오를 때, 최댓값은 10 + 15, 즉 25이다.

네번째 계단까지 오를 때, 최댓값은 어떻게 구할까?

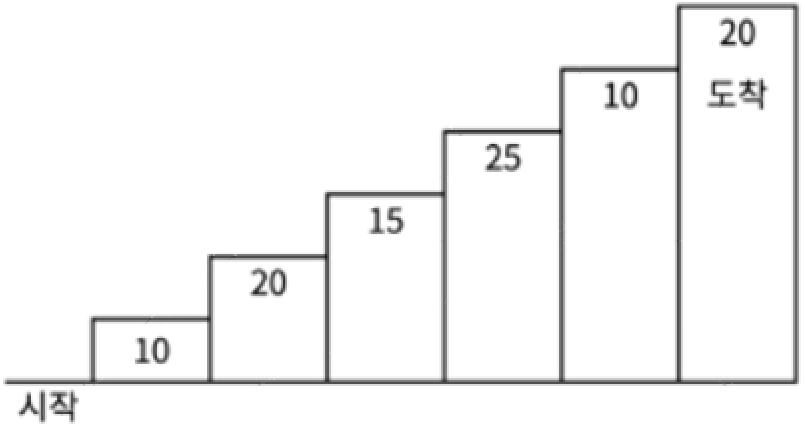
1. **두번째 계단까지 오를 때의 최댓값** + 네번째 계단의 점수
2. **첫번째 계단까지 오를 때의 최댓값** + 세번째 계단의 점수 + 네번째 계단의 점수

위 2개의 값 중에서 최댓값을 선택하면 된다.

다섯번째 계단까지 오를 때, 최댓값을 구하는 방법도 마찬가지이다.

1. **세번째 계단까지 오를 때의 최댓값** + 다섯번째 계단의 점수
2. **두번째 계단까지 오를 때의 최댓값** + 네번째 계단의 점수 + 다섯번째 계단의 점수

위 2개의 값 중에서 최댓값을 선택하면 된다.



<그림 1>

# DP B. 문제 풀이 | 03 백준 2579번 계단 오르기

- **첫번째 풀이**
  - dp 배열을 정의해봅시다.
    - dp[i] = (i번째 계단을 반드시 밟았을 때 얻을 수 있는 점수의 최댓값)
  - 초기값을 설정해봅시다.
    - dp[0] = 0, dp[1] = stair[1]
    - dp[2] = stair[1] + stair[2]
    - dp[3] = max(stair[1], dp[0] + stair[2]) + stair[3]
  - 점화식을 세워봅시다.
    - dp[i] = max(dp[i – 2], dp[i-3] + stair[i-1]) + stair[i] (단, i >= 4)
  - 구현합시다.
    - Bottom-up 방식으로 문제를 풀이(설명)했으니 반복문으로 구현하겠다.

# DP B. 문제 풀이 | 03 백준 2579번 계단 오르기

- **첫번째 구현**

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, dp[303], stair[303];

int main(){
    ios::sync_with_stdio(false);

    cin >> N;
    for(int i = 1; i <= N; i++) cin >> stair[i];
    dp[1] = stair[1];
    dp[2] = stair[1] + stair[2];
    dp[3] = max(stair[2] + stair[3], stair[1] + stair[3]);
    for(int i = 4; i <= N; i++){
        dp[i] = max(dp[i-3] + stair[i-1], dp[i-2]) + stair[i];
    }
    cout << dp[N] << "\n";
    return 0;
}
```

# **DP** B. 문제 풀이 | 03 백준 2579번 계단 오르기

- **두번째 풀이**
  - 첫번째 풀이에서는 계단을 의미하는 변수 i 로만 dp 배열의 상태를 관리하고 점화식을 작성하였다.
  - 그리고 나서 **점화식으로 문제의 규칙(3개 계단을 연속해서 밟으면 안 된다는 규칙)을 구현**했다.
  - 두번째 풀이에서는, **dp 배열에 또 다른 상태를 추가하면 dp 배열로 문제의 규칙을 표현할 수 있음을 보여줄 것이다.**

  - dp 배열을 다음과 같이 정의하면 된다.
  - dp[i][j] = (현재까지 j개의 계단을 연속해서 밟고 i번째 계단까지 올라섰을 때 점수 합의 최댓값)
  - dp[i][j]의 최댓값은 어떻게 구할 수 있을까?
    - j = 1인 경우, i번째를 밟기 전에 i-1번째 계단을 건너 뛰었다는 의미다. 따라서, dp[i][1] = max(dp[i-2][1], dp[i-2][2]) + stair[i] 이다.
    - j = 2인 경우, i번째를 밟기 전에 i-1번째 계단을 밟았다는 의미다. 따라서, dp[i][2] = dp[i-1][1] + stair[i] 이다.
  - 문제에서 출력 해야하는 답은 max(dp[n][1], dp[n][2])가 된다

# DP B. 문제 풀이 | 03 백준 2579번 계단 오르기

- **두번째 풀이**
  - dp 배열을 정의해봅시다.
    - dp[i][j] = (현재까지 j개의 계단을 연속해서 밟고 i번째 계단까지 올라섰을 때 점수 합의 최댓값)
  - 초기값을 설정해봅시다.
    - dp[1][1] = stair[1]
    - dp[2][1] = stair[2]
    - dp[2][2] = stair[1] + stair[2]
  - 점화식을 세워봅시다.
    - dp[i][1] = max(dp[i-2][1], dp[i-2][2]) + stair[i];
    - dp[i][2] = dp[i-1][1] + stair[i]
  - 구현합시다.
    - Bottom-up 방식으로 문제를 풀이했으니 반복문으로 구현하겠다.

# DP B. 문제 풀이 | 03 백준 2579번 계단 오르기

- **두번째 구현**

```cpp
#include <bits/stdc++.h>
using namespace std;
int dp[301][3];
int stair[301];

int main(){
    cin.tie(0);
    cout.tie(0);
    ios::sync_with_stdio(false);
    int T; cin >> T;
    for(int i = 1; i <= T; i++){
        cin >> stair[i];
    }
    dp[1][1] = stair[1];
    dp[2][1] = stair[2];
    dp[2][2] = stair[1] + stair[2];

    for(int i = 3; i <= T; i++){
        dp[i][1] = max(dp[i-2][1], dp[i-2][2]) + stair[i];
        dp[i][2] = dp[i-1][1] + stair[i];
    }

    cout << max(dp[T][1], dp[T][2]);
    return 0;
}
```

# DP B. 문제 풀이 | 03 백준 2579번 계단 오르기

- **첫번째 풀이와 두번째 풀이 차이?**
  - **결국은 dp 배열을 정의하고 점화식을 작성하는게, dp 문제를 푸는 방법이고 많은 시간이 소요되는 부분이다.**

| 항목 | 첫번째 풀이 | 두번째 풀이 |
| --- | --- | --- |
| dp 배열 정의 | dp[i] = i번째 계단을 밟았을 때 최대 점수 | dp[i][j] = i번째 계단을 밟았고, j개의 계단을 연속해서 밟은 상태에서의 최대 점수 |
| 상태 표현 | 상태 하나(i : 계단) | 상태 두개(i : 계단, j : 연속횟수) |
| 제약 조건 처리 | 점화식 설계로 3연속 금지 조건을 피함. | j라는 변수로 상태를 나눠서 3연속 불가 조건을 dp 배열로 방지함. |
| 점화식 구조 | dp[i] = max(dp[i-2], dp[i-3] + s[i-1]) + s[i] | dp[i][1] = max(dp[i-2][1], dp[i-2][2]) + s[i]<br>그리고<br>dp[i][2] = dp[i-1][1] + s[i] |

# DP B. 문제 풀이 | 04 백준 11053번 가장 긴 증가하는 부분 수열

- **문제요약**
  - 수열 A가 주어졌을 때, 가장 긴 증가하는 부분 수열을 구하는 프로그램을 작성하시오.
  - 예를 들어, 수열 A = {10, 20, 10, 30, 20, 50} 인 경우에 가장 긴 증가하는 부분 수열은 A = {10, 20, 10, 30, 20, 50} 이고, 길이는 4이다.
- **입력**
  - 첫째 줄에 수열 A의 크기 N (1 ≤ N ≤ 1,000)이 주어진다.
  - 둘째 줄에는 수열 A를 이루고 있는 Ai가 주어진다. (1 ≤ Ai ≤ 1,000)
- **출력**
  - 첫째 줄에 수열 A의 가장 긴 증가하는 부분 수열의 길이를 출력한다.

# DP B. 문제 풀이 | 04 백준 11053번 가장 긴 증가하는 부분 수열

- **풀이**
  - 문제의 예제를 보면서 파악해보자. 크기가 6인 수열 A = {10, 20, 10, 30, 20, 50} 가 있다.
  - 편의상 가장 긴 증가하는 부분 수열의 길이를 'LIS'라 하겠다. (LIS : Longest Increasing Subsequence)
  - dp 배열을 다음과 같이 정의해보자.
  - **dp[i] = (0번째부터 i번째까지의 부분 수열에 대한 LIS)**
    - i = 0일 때, LIS는 1이다. -> {10}
    - i = 1일 때, LIS는 2이다. -> {10, 20}
    - i = 2일 때, LIS는 2이다. -> {10, 20}
    - i = 3일 때, LIS는 3이다. -> {10, 20, 30}
    - i = 4일 때, LIS는 3이다. -> {10, 20, 30}
    - i = 5일 때, LIS는 4이다. -> {10, 20, 30, 50}

# DP B. 문제 풀이 | 04 백준 11053번 가장 긴 증가하는 부분 수열

- **풀이**
  - **dp[i] = (0번째부터 i번째까지의 부분 수열에 대한 LIS)**
  - DP 문제 풀이법의 결론은 **"dp 문제 = 상태 설정 + 전이 관계 세우기"**인데, 위처럼 dp 배열을 정의하면 관계를 찾기 어렵다.
  - 다음과 같이 정의해보자.
  - **dp[i] = (i번째 원소를 마지막으로 하는 LIS)** // 즉, 수열 A[0] ~ A[i]에서 **A[i]를 꼭 포함한** LIS를 저장하는 배열
  - dp[i]를 구하려면 다음과 같은 과정을 거치면 된다.
    - 0 <= j < i에 대해서
    - A[j] < A[i] 라면, dp[i] = max(dp[i], dp[j] + 1) 값을 계산한다. // 즉, 앞에서 A[i]보다 작은 값들 중 LIS 길이가 가장 긴 것 + 1 것이다.
  - **앞선 두 dp 배열 정의에서 상태를 정의하는 변수는 동일하나, dp 배열을 정의하는 방법이 서로 다르다.**
    - dp문제는 상태를 설정하고 전이 관계를 세우는게 가장 어렵고 시간이 많이 소요된다.

# DP B. 문제 풀이 | 04 백준 11053번 가장 긴 증가하는 부분 수열

- **구현**

```cpp
#include <bits/stdc++.h>
using namespace std;

int dp[1000];
int A[1000];

int main(){
    cin.tie(0);
    cout.tie(0);
    ios::sync_with_stdio(false);

    int N; cin >> N;
    for(int i = 0; i < N; i++) cin >> A[i];

    for(int i = 0; i < N; i++){
        dp[i] = 1;
        for(int j = 0; j < i; j++){
            if(A[i] > A[j]) dp[i] = max(dp[i], dp[j] + 1);
        }
    }

    cout << *max_element(dp, dp + 1000);
    return 0;
}
```

# DP B. 문제 풀이 | 05 백준 12865번 평범한 배낭

- **문제요약**
  - 준서가 여행에 필요하다고 생각하는 N개의 물건이 있다.
  - 각 물건은 무게 W와 가치 V를 가지는데, 해당 물건을 배낭에 넣어서 가면 준서가 V만큼 즐길 수 있다.
  - 아직 행군을 해본 적이 없는 준서는 최대 K만큼의 무게만을 넣을 수 있는 배낭만 들고 다닐 수 있다.
  - 준서가 최대한 즐거운 여행을 하기 위해 배낭에 넣을 수 있는 물건들의 가치의 최댓값을 알려주자.
- **입력**
  - 첫 줄에 물품의 수 N(1 ≤ N ≤ 100)과 준서가 버틸 수 있는 무게 K(1 ≤ K ≤ 100,000)가 주어진다.
  - 두 번째 줄부터 N개의 줄에 거쳐 각 물건의 무게 W(1 ≤ W ≤ 100,000)와 해당 물건의 가치 V(0 ≤ V ≤ 1,000)가 주어진다.
- **출력**
  - 한 줄에 배낭에 넣을 수 있는 물건들의 가치합의 최댓값을 출력한다.

5
MIN

# DP B. 문제 풀이 | 05 백준 12865번 평범한 배낭

- **풀이**
  - 어떤 것을 상태를 나타낼 변수로 잡아야 하고, dp 배열을 어떻게 정의할 지 고민해보자.
  - 일단, 우리가 출력해야 하는 것은 '배낭에 넣을 수 있는' '물건들의 가치합의' '최댓값'이다. 이를 토대로 상태 변수를 잡아보자.
  - 현재 넣을지 고려 중인 물건 번호 i, 현재 가방 무게 j 를 상태변수로 둘 수 있다.
    - i라는 변수를 잡아줌으로서 구간 [1, i]의 물건을 확인했음을 나타낸다. // 0-based 인덱스가 아닌 1-based 인덱스를 사용하겠음!
    - j라는 변수를 잡아줌으로서 배낭에 넣을 수 있는지를 체크할 수 있다.
  - dp[i][j] = (물건 1번부터 i번까지 고려했을 때, 무게 j **이하**로 담을 수 있는 가치의 최댓값)
    - 자연스럽게 정답은 dp[N][K] 가 된다.
    - dp[N][K] : 물건 1번부터 N번까지 고려했을 때, 무게 K 이하로 담을 수 있는 가치의 최댓값이기 때문이다.

# DP B. 문제 풀이 | 05 백준 12865번 평범한 배낭

- **풀이**
  - dp[i][j] = (물건 1번부터 i번까지 고려했을 때, 무게 j 이하로 담을 수 있는 가치의 최댓값)
  - dp[i][j] 를 이전 항으로 나타낼 수 있는지 점화식을 작성해보자.
  - i번째 물건의 무게 w[i]와 가치 v[i]에 대해 다음과 같은 선택이 가능하다. 두 경우의 최댓값을 선택하면 된다.
    - i번째 물건을 담는 경우
      - dp[i][j] = dp[i – 1][j – w[i]] + v[i] (단, j – w[i] >= 0)
      - i번째 물건을 선택해서 배낭에 넣을 때 얻을 수 있는 최대 가치는, dp[i – 1][j – w[i]]에다가 i번째 물건의 가치 v[i]를 더한 것이다.
      - dp[i – 1][j – w[i]] : 물건 1번부터 i – 1번까지 고려했을 때, 무게 j – w[i] 이하로 담을 수 있는가치의 최댓값
    - i번째 물건을 담지 않는 경우
      - dp[i][j] = dp[i-1][j]
      - i번째 물건을 고려했지만, 결국 담지 않았다는 의미의 점화식이다.
  - 이 풀이의 시간복잡도는 O(NK), 공간 복잡도가 O(NK)이다.

# DP B. 문제 풀이 | 05 백준 12865번 평범한 배낭

- **구현**

```cpp
#include <bits/stdc++.h>
using namespace std;

int dp[101][1000001];
int N, K;
int w[101], v[101];

int main(){

    cin >> N >> K;
    for(int i = 1; i <= N; i++) cin >> w[i] >> v[i];

    for(int i = 1; i <= N; i++){
        for(int j = 1; j <= K; j++){
            if(j-w[i]>=0){
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i]);
            }else{
                dp[i][j] = dp[i-1][j];
            }
        }
    }

    cout << dp[N][K] << "\n";
    return 0;
}
```

# DP B. 문제 풀이 | 06 백준 1149번 RGB 거리

- **문제요약**
  - RGB거리에는 집이 N개 있다. 거리는 선분으로 나타낼 수 있고, 1번 집부터 N번 집이 순서대로 있다.
  - 집은 빨강, 초록, 파랑 중 하나의 색으로 칠해야 한다. 각각의 집을 빨강, 초록, 파랑으로 칠하는 비용이 주어졌을 때, 아래 규칙을 만족하면서 모든 집을 칠하는 비용의 최솟값을 구해보자.
    1. 1번 집의 색은 2번 집의 색과 같지 않아야 한다.
    2. N번 집의 색은 N-1번 집의 색과 같지 않아야 한다.
    3. i(2 ≤ i ≤ N-1)번 집의 색은 i-1번, i+1번 집의 색과 같지 않아야 한다.
- **입력**
  - 첫째 줄에 집의 수 N(2 ≤ N ≤ 1,000)이 주어진다. 둘째 줄부터 N개의 줄에는 각 집을 빨강, 초록, 파랑으로 칠하는 비용이 1번 집부터 한 줄에 하나씩 주어진다. 집을 칠하는 비용은 1,000보다 작거나 같은 자연수이다.
- **출력**
  - 첫째 줄에 모든 집을 칠하는 비용의 최솟값을 출력한다.

5 MIN

# DP B. 문제 풀이 | 06 백준 1149번 RGB 거리

- **풀이**
  - 상태 변수를 어떤 걸로 잡고, 어떻게 dp 배열을 정의하면 될까?
  - 일단, 문제에서 비용을 출력하라고 했으니 **dp[상태변수] = ( ~~~ 비용) 꼴**이라고 잡아보자.
  - 우리는 문제를 정의할 때 **집(변수 i)**과 **색상(변수 j)**에 대한 조건을 확인해야 하므로, 각각을 상태변수로 잡아보자.
  - 그러면 다음과 같이 dp 배열을 정의할 수 있다.

  - dp[i][j] = i번째 집을 j색으로 칠했을 때의 최소 비용(j = 0 : 빨강, 1: 초록, 2 : 파랑)

# DP B. 문제 풀이 | 06 백준 1149번 RGB 거리

- **풀이**
  - dp[i][j] = i번째 집을 j색으로 칠했을 때의 최소 비용(j = 0 : 빨강, 1: 초록, 2 : 파랑)
  - i번째 집을 j색으로 칠할 때, (i-1)번째 집은 j색이 아니어야 하므로, (i-1)번째 집을 다른 두 색 중 하나로 칠한 최소 비용을 고르면 된다.

  - 점화식
    - $dp[i][0] \; = \; \min(dp[i-1][1], dp[i-1][2]) \; + \; cost[i][0]$
      - i번째 집을 빨간색으로 칠했을 때의 최소 비용은 i-1번째 집을 초록색이나 파란색으로 칠한 최소 비용에 i번째 집을 빨간색으로 칠했을 때 비용을 더하면 된다.
    - $dp[i][1] \; = \; \min(dp[i-1][0], dp[i-1][2]) \; + \; cost[i][1]$
      - i번째 집을 초록색으로 칠했을 때의 최소 비용은 i-1번째 집을 빨간색이나 파란색으로 칠한 최소 비용에 i번째 집을 초록색으로 칠했을 때 비용을 더하면 된다.
    - $dp[i][2] \; = \; \min(dp[i-1][0], dp[i-1][1]) \; + \; cost[i][1]$
      - i번째 집을 파란색으로 칠했을 때의 최소 비용은 i-1번째 집을 빨간색이나 초록색으로 칠한 최소 비용에 i번째 집을 파란색으로 칠했을 때 비용을 더하면 된다.

    - 정답은 $\min(\{dp[N-1][0], dp[N-1][1], dp[N-1][2]\})$ 이다.

# DP B. 문제 풀이 | 06 백준 1149번 RGB 거리

- **풀이**
  - dp 배열을 정의해봅시다.
    - dp[i][j] = i번째 집을 j색으로 칠했을 때의 최소 비용(j = 0 : 빨강, 1: 초록, 2 : 파랑)
  - 초기값을 설정해봅시다.
    - dp[0][0] = cost[0][0], dp[0][1] = cost[0][1], dp[0][2] = cost[0][2] = cost[0][2]
  - 점화식을 세워봅시다.
    - $dp[i][0] = \min(dp[i-1][1], dp[i-1][2]) + cost[i][0]$
    - $dp[i][1] = \min(dp[i-1][0], dp[i-1][2]) + cost[i][1]$
    - $dp[i][2] = \min(dp[i-1][0], dp[i-1][1]) + cost[i][2]$
  - 구현합시다.
    - Bottom-up 방식으로 문제를 풀이(설명)했으니 반복문으로 구현하겠다.

# DP B. 문제 풀이 | 06 백준 1149번 RGB 거리

- **구현**

```cpp
#include <bits/stdc++.h>
using namespace std;

int dp[1000][3];
int cost[1000][3];


int main(){
    cin.tie(0);
    cout.tie(0);
    ios::sync_with_stdio(false);

    int N; cin >> N;
    for(int i = 0; i < N; i++){
        cin >> cost[i][0] >> cost[i][1] >> cost[i][2];
    }

    dp[0][0] = cost[0][0];
    dp[0][1] = cost[0][1];
    dp[0][2] = cost[0][2];

    for(int i = 1; i < N; i++){
        dp[i][0] = min(dp[i-1][1], dp[i-1][2]) + cost[i][0];
        dp[i][1] = min(dp[i-1][0], dp[i-1][2]) + cost[i][1];
        dp[i][2] = min(dp[i-1][0], dp[i-1][1]) + cost[i][2];
    }
    cout << min(min(dp[N-1][0], dp[N-1][1]), dp[N-1][2]) << "\n";   // min({dp[N – 1][0], dp[N– 1][1], dp[N – 1][2]}) 와 결과 동일
    return 0;
}
```

# DP B. 문제 풀이 | 07 백준 9095번 1, 2, 3 더하기

- **문제 요약**
  - 정수 n이 주어졌을 때, n을 1, 2, 3의 합으로 나타내는 방법의 수를 구하는 프로그램을 작성하시오.
  - 단, 순서가 다르면 다른 경우로 센다.
- **입력**
  - 첫째 줄에 테스트 케이스의 개수 T가 주어진다. 각 테스트 케이스는 한 줄로 이루어져 있고, 정수 n이 주어진다. n은 양수이며 11 이하이다.
- **출력**
  - 각 테스트 케이스마다, n을 1, 2, 3의 합으로 나타내는 방법의 수를 출력한다.

# DP B. 문제 풀이 | 07 백준 9095번 1, 2, 3 더하기

- **풀이**
  - 상태변수를 잡을 만 한게 n밖에 보이지 않아 n으로 잡겠다.
  - 정수 n을 만들기 위해 어떤 작업을 할 수 있을까?
    - **만들어진 정수 n – 1**에 1을 붙이면 된다.
    - **만들어진 정수 n – 2**에 2를 붙이면 된다.
    - **만들어진 정수n – 3**에 3을 붙이면 된다.
  - 따라서, 점화식은 dp[n] = dp[n – 1] + dp[n – 2] + dp[n – 3] 이다. (단, n >= 4)

# DP B. 문제 풀이 | 07 백준 9095번 1, 2, 3 더하기

- **풀이**
  - dp 배열을 정의해봅시다.
    - dp[n] = 정수 n을 1, 2, 3의 합으로 나타내는 경우의 수
  - 초기값을 설정해봅시다.
    - dp[1] = 1, dp[2] = 2, dp[3] = 4
  - 점화식을 세워봅시다.
    - dp[n] = dp[n – 1] + dp[n – 2] + dp[n – 3]
  - 구현합시다.

# DP B. 문제 풀이 | 07 백준 9095번 1, 2, 3 더하기

- **구현**

```cpp
#include <bits/stdc++.h>
using namespace std;
int dp[1000001];


int main(){
    cin.tie(0);
    cout.tie(0);
    ios::sync_with_stdio(false);
    int T; cin >> T;
    dp[1] = 1;
    dp[2] = 2;
    dp[3] = 4;
    for(int i = 4; i <= 11; i++){
        dp[i] = dp[i-1] + dp[i-2] + dp[i-3];
    }

    while(T--){
        int N; cin >> N;
        cout << dp[N] << "\n";
    }
    return 0;
}
```

# DP B. 문제 풀이 | 08 백준 9251번 LCS

- **문제요약**
  - LCS(Longest Common Subsequence, 최장 공통 부분 수열)문제는 두 수열이 주어졌을 때, 모두의 부분 수열이 되는 수열 중 가장 긴 것을 찾는 문제이다.
  - 예를 들어, ACAYKP와 CAPCAK의 LCS는 ACAK가 된다.
- **입력**
  - 첫째 줄과 둘째 줄에 두 문자열이 주어진다. 문자열은 알파벳 대문자로만 이루어져 있으며, 최대 1000글자로 이루어져 있다.
- **출력**
  - 첫째 줄에 입력으로 주어진 두 문자열의 LCS의 길이를 출력한다.

# DP B. 문제 풀이 | 08 백준 9251번 LCS

- **풀이**
  - 문자열 A, B에 대해 LCS 길이를 찾아야 한다. 상태 변수를 어떻게 잡고 점화식을 작성하면 될까?
  - dp[i][j] = (A의 i번째 문자까지와 B의 j번째 문자까지의 LCS 길이)라고 하자. // 1-based 인덱스
  - 이렇게 dp 배열을 작성하면 자연스럽게 점화식도 작성이 가능하다.
  - A[i] == B[j] 이면, dp[i][j] = dp[i – 1][j – 1] + 1;
    - A의 i – 1번째 문자까지와 B의 j – 1번째 문자까지의 LCS길이에 1을 더하면,
    - A의 i번째 문자까지와 B의 j번째 문자까지의 LCS 길이가 된다.
  - A[i] != B[j] 이면, dp[i][j] = max(dp[i – 1][j], dp[i][j – 1])
    - 두 값 중 더 큰 값을 선택하면 된다.
  - 문제에서 원하는게 문자열 A, B의 LCS 길이를 출력하는 것이므로 dp[A.length()][B.length()]를 출력하면 된다.

# DP B. 문제 풀이 | 08 백준 9251번 LCS

- **구현**

```cpp
#include <bits/stdc++.h>
using namespace std;

string A, B;
int dp[1001][1001];

int main(){
  ios::sync_with_stdio(false);
  cin >> A >> B;
  int N = A.length(), M = B.length();
  A = '#' + A, B = '#' + B; // 1-based

  for(int i = 1; i <= N; i++){
    for(int j = 1; j <= M; j++){
      if(A[i] == B[j]){
        dp[i][j] = dp[i-1][j-1] + 1;
      }else{
        dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
      }
    }
  }
  cout << dp[N][M] << "\n";
  return 0;
}
```

# DP B. 문제 풀이 | 09 백준 2293번 동전 1

- **문제 요약**
  - n가지 종류의 동전이 있다. 각각의 동전이 나타내는 가치는 다르다.
  - 이 동전을 적당히 사용해서, 그 가치의 합이 k원이 되도록 하고 싶다. 그 경우의 수를 구하시오. 각각의 동전은 몇 개라도 사용할 수 있다.
  - 사용한 동전의 구성이 같은데, 순서만 다른 것은 같은 경우이다. (= 중복 조합)
- **입력**
  - 첫째 줄에 n, k가 주어진다.
  - 다음 n개의 줄에는 각각의 동전의 가치가 주어진다. 동전의 가치는 100,000보다 작거나 같은 자연수이다.
  - $1 \leq n \leq 100$, $1 \leq k \leq 10{,}000$
- **출력**
  - 첫째 줄에 경우의 수를 출력한다. 경우의 수는 2^31보다 작다.

# DP B. 문제 풀이 | 09 백준 2293번 동전 1

- **풀이**
  - 서로의 가치가 다른 n 종류의 동전을 활용해 그 합이 k가 되는 경우의 수를 출력하는 게 목표이다.
  - 상태 변수를 뭘로 잡고 dp 배열을 어떻게 정의하면 될까?
  - dp[i] = (i원을 만들 수 있는 경우의 수) 라고 정의하자.
  - 동전의 가치를 저장해둔 배열을 coin이라고 하자.
  - 그러면 다음과 같은 점화 관계를 작성할 수 있다.
  - $dp[j] \ += \ dp[j - coin[i]]$
    - (j원을 만드는 경우의 수) += (j – coin[i] 원을 만드는 경우의 수)
    - dp[0] = 1이다. 금액 0을 만드는 방법은 아무 동전도 쓰지 않은 것이다.

# DP B. 문제 풀이 | 09 백준 2293번 동전 1

- 구현

```cpp
#include <bits/stdc++.h>
using namespace std;

unsigned int dp[11000];
int coin[110];

int main(){
    cin.tie(0);
    cout.tie(0);
    ios::sync_with_stdio(false);

    int n, k; cin >> n >> k;
    for(int i = 1; i <= n; i++){
        cin >> coin[i];
    }
    dp[0] = 1;
    for(int i = 1; i <= n; i++){
        for(int j = coin[i]; j <= k; j++){
            dp[j] = dp[j] + dp[j-coin[i]];
        }
    }
    cout << dp[k];
    return 0;
}
```

# **DP** B. 문제 풀이 | 10 백준 2156번 포도주 시식

- **문제 요약**
  - 효주는 포도주 시식을 하려고 하는데, 여기에는 다음과 같은 두 가지 규칙이 있다.
    1. 포도주 잔을 선택하면 그 잔에 들어있는 포도주는 모두 마셔야 하고, 마신 후에는 원래 위치에 다시 놓아야 한다.
    2. 연속으로 놓여 있는 3잔을 모두 마실 수는 없다.
  - 1부터 n까지의 번호가 붙어 있는 n개의 포도주 잔이 순서대로 테이블 위에 놓여 있고,
    각 포도주 잔에 들어있는 포도주의 양이 주어졌을 때,
    효주를 도와 가장 많은 양의 포도주를 마실 수 있도록 하는 프로그램을 작성하시오.
- **입력**
  - 첫째 줄에 포도주 잔의 개수 n이 주어진다. ($1 \le n \le 10{,}000$) 둘째 줄부터 n+1번째 줄까지 포도주 잔에 들어있는 포도주의 양이 순서대로 주어진다. 포도주의 양은 1,000 이하의 음이 아닌 정수이다.
- **출력**
  - 첫째 줄에 최대로 마실 수 있는 포도주의 양을 출력한다.

# DP B. 문제 풀이 | 10 백준 2156번 포도주 시식

- **풀이**
  - 연속으로 3개의 포도주를 마시지 못할 때, 가장 많은 포도주를 마시는게 목표이다.
  - 상태 변수와 점화식은 어떻게 작성하면 될까?
  - dp[i] = (1번 ~ i번 포도주까지 고려했을 때, i번째 포도주를 "마실 수도 있고, 안 마실 수도" 있을 때의 최대 포도주 양) // 1-based

  - i번째 포도주를 마시지 않는다 → dp[i] = dp[i-1]
  - i번째 포도주를 마시고, i-1번째 포도주는 마시지 않는다 → dp[i] = dp[i-2] + wine[i]
  - i번째 포도주를 마시고, i-1번째 포도주도 마신다 (즉, i-2번째는 안 마셨던 경우) → dp[i] = dp[i-3] + wine[i-1] + wine[i]

  - 위 3가지 값 중 최댓값이 dp[i]이다.

# DP B. 문제 풀이 | 10 백준 2156번 포도주 시식

- **구현**

```cpp
#include <bits/stdc++.h>
using namespace std;

int wine[10001];
int dp[10001];

int main() {
    int n;
    cin >> n;

    for (int i = 1; i <= n; ++i) {
        cin >> wine[i];
    }

    dp[1] = wine[1];
    dp[2] = wine[1] + wine[2];
    dp[3] = max({dp[2], wine[1] + wine[3], wine[2] + wine[3]});

    for (int i = 4; i <= n; ++i) {
        dp[i] = max({
            dp[i - 1],
            dp[i - 2] + wine[i],
            dp[i - 3] + wine[i - 1] + wine[i]
        });
    }
    cout << dp[n] << '\n';
    return 0;
}
```

# 문제

- **DP 문제만 푸시면 됩니다!**
- **꼭 풀어야하는 Well-Known 문제만 꾹꾹 눌러담은 DP 문제집**
  - https://www.acmicpc.net/workbook/view/7319
- **DP 정복하고 싶은 사람만! *다른 자료구조와 알고리즘이 결합된 문제 다수**
  - https://www.acmicpc.net/workbook/view/2163

오늘 스터디는 여기서 마무리하겠습니다.

백준 많이 푸세요! 수고하셨습니다!