**<Baekjoon 1463> 1로 만들기**

Bottom-up은 for문을 이용해서, Top-down은 재귀를 이용해서 다이내믹 프로그래밍을 진행한다.

N을 구하는 문제를 구조가 같은 더 작은 수를 구하는 문제로 바꿀 수 있으면 다이내믹 프로그래밍을 생각한다.

**문제**

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**

첫째 줄에 1보다 크거나 같고, 106보다 작거나 같은 정수 N이 주어진다.

**출력**

첫째 줄에 연산을 하는 횟수의 최솟값을 출력한다.

예제입력1 2                예제입력2 10

예제출력1 1                예제출력2 3

<문제 풀이 방향>

ㄱ. 3으로 나누고, 2로 나누고, 1을 빼는 것 순서대로 적용해야 문제를 푼다는 생각은 좋지 않다.

ㄴ. 10을 예시로 들면 ㄱ이 좋지 않은 생각이라는 것을 알 수 있다.

ㄷ. 따라서 위 문제는 다이내믹 프로그래밍을 적용해야 한다는 것을 떠올려야 한다.

<DP Array 설계>

ㄹ. dp array에 담을 dp[n]은 ’n을 1로 만드는데 필요한 필요한 연산의 수’로 설정해야한다.

-> 이는 결국 문제에서 출력해야하는 결과가 된다.

-> dp array의 크기는 입력에서 주어진 N+1의 범위와 같다. 이는 0번째 index는 보통 사용하지 않기 때문이다.

<dp[n] 구하는 방법과 dp적용의 당위성>

ㅁ. dp[n]은 min( min( dp[n/3] , dp[n/2] ), dp[n-1] ) 으로 구할 수 있다.

-> n을 구하는 것을 n/3,n/2,n-1을 구하는 문제 중 하나로 쪼갤 수 있고, 문제 푸는 구조가 같으므로, 다이내믹 프로그래밍으로 풀 수 있다.

<시간복잡도 분석>

ㅂ. 차례대로 재귀또는 for문으로 호출하고 (O(N)), 함수또는 for문 내에서 단순 연산을 진행하므로 (O(1)), 시간복잡도는 O(N)이다.

**Remarks\_**

Top-Down (feat\_재귀)

ㄱ. 재귀 함수내에서 처음에 실행해주어야하는 조건은 다음과 같다.

=============================================================================

 if(n == 1)  return 0;  // n이 1이면 재귀함수 종료 (종료조건)

 if(dp[n]>0) return dp[n];  //  이미 dp칸이 채워져 있으면, 연산이 이미 진행되었으므로 이미 구한 값을 이용 (보초법)

dp[n] = 재귀함수(n-1) ~ // dp 배열에 재귀함수를 호출하여 칸을 채우기 (연산)

=============================================================================

ㄴ. dp를 전역배열로 두면 초기에 모두 0으로 초기화되어 있다는 장점이 있다.

-> 아직 재귀가 호출되지 않으면 0이 담겨있도록 하는것이 핵심이다.

Bottom-Up (feat\_for문)

ㄱ. for문에서 처음에 실행해주어야하는 조건은 다음과 같다.

=============================================================================

dp[1] =0; // 초기값 설정 (초기 조건)

for(int i=2;i<n;i++)  // (연산 범위 설정)

{

    dp[n] = dp[n-1] ~ // 이미 채운 낮은 dp배열 수를 이용하여 dp배열 채워나가기 (연산)

}

=============================================================================

**Answer\_**

<https://www.acmicpc.net/source/13571912>

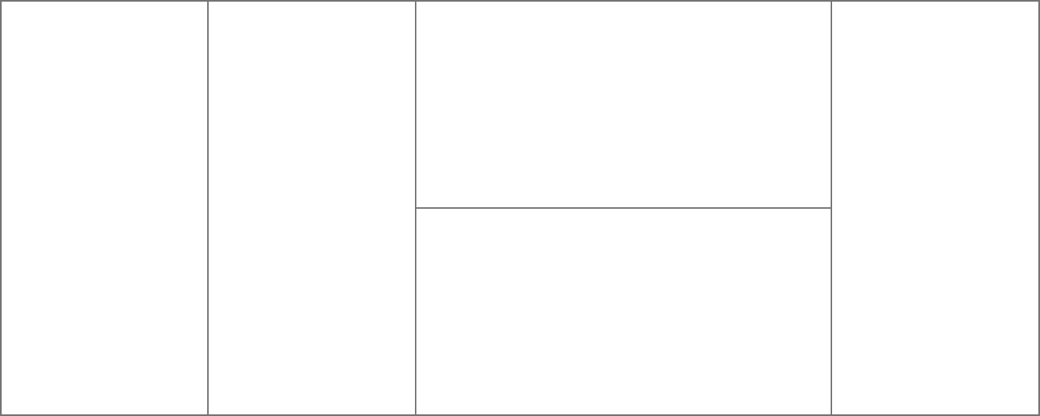
**<Baekjoon 11726, 11727> 2xn 타일링 1,2**

 dp[n]을 작은 문제로 쪼개는 연산식을 세울 때, 결국 점화식을 알아야 세울 수 있다.

**문제**

2×n 크기의 직사각형을 1×2, 2×1 타일로 채우는 방법의 수를 구하는 프로그램을 작성하시오.

아래 그림은 2×5 크기의 직사각형을 채운 한 가지 방법의 예이다.



**입력**

첫째 줄에 n이 주어진다. (1 ≤ n ≤ 1,000)

**출력**

첫째 줄에 2×n 크기의 직사각형을 채우는 방법의 수를 10,007로 나눈 나머지를 출력한다.

예제입력1 2                예제입력2 9

예제출력1 2                예제출력2 55

<문제 풀이 방향>

ㄱ. dp[n] = dp[n-1] + dp[n-2]를 통해서 구할 수 있다는 생각을 쉽게 할 수 있다.

<DP Array 설계>

ㄹ. dp array에 담을 dp[n]은 ’2xn칸을 채울 수 있는 경우의 수’로 설정해야한다.

<dp[n] 구하는 방법과 dp적용의 당위성>

ㅁ. dp[n]은 dp[n-1] + dp[n-2]으로 구할 수 있다.

-> n을 구하는 것을 n-1, n-2를 구하는 문제 중 하나로 쪼갤 수 있고, 문제 푸는 구조가 같으므로, 다이내믹 프로그래밍으로 풀 수 있다.

<시간복잡도 분석>

ㅂ. 차례대로 재귀또는 for문으로 호출하고 (O(N)), 함수또는 for문 내에서 단순 연산을 진행하므로 (O(1)), 시간복잡도는 O(N)이다.

**Remarks\_**

방법의 수를 10007로 나눈 나머지를 출력한다.

ㄱ. dp계산 도중에는 계속하여 재귀를 호출하거나, for문이 돌아가는 과정에서 int 범위를 벗어나는 지나치게 큰 값이 발생할 수 있다.

ㄴ. 이런 값이 발생하는지에 대해서 힌트를 주는 것이 어떠한 소수로 나눈 나머지를 출력하라는 것이다.

ㄷ. 큰 값을 dp array에 저장하는 대신, 이를 10007로 나눈 나머지를 계속해서 저장하는 것이 방법이다.

**Answer\_**

<https://www.acmicpc.net/source/13666428>

비슷한 문제  : Baekjoon 11267

<https://www.acmicpc.net/submit/11727/13666641>

**<Baekjoon 9095> 1,2,3 더하기**

N을 구하는 문제를 구조가 같은 더 작은 수를 구하는 문제로 바꿀 수 있으면 다이내믹 프로그래밍을 생각한다. (문제를 보자말자 명확한 점화식이 보이지 않아도!)

**문제**

정수 4를 1, 2, 3의 합으로 나타내는 방법은 총 7가지가 있다. 합을 나타낼 때는 수를 1개 이상 사용해야 한다.

* 1+1+1+1
* 1+1+2
* 1+2+1
* 2+1+1
* 2+2
* 1+3
* 3+1

정수 n이 주어졌을 때, n을 1, 2, 3의 합으로 나타내는 방법의 수를 구하는 프로그램을 작성하시오.

**입력**

첫째 줄에 테스트 케이스의 개수 T가 주어진다. 각 테스트 케이스는 한 줄로 이루어져 있고, 정수 n이 주어진다. n은 양수이며 11보다 작다.

**출력**

각 테스트 케이스마다, n을 1, 2, 3의 합으로 나타내는 방법의 수를 출력한다.

예제입력1                         예제출력1

3

4                                     7

7                                     44

10                                   274

<문제 풀이 방향>

ㄱ. 직접 경우의 수를 구하는 것은 예제입력에서 10만 봐도 어렵다는 것을 알 수 있다.

ㄴ. n을 1,2,3으로 표현하는 과정에서 마지막으로 더한 수에 따라서 n을 구하는 문제를 n-1,n-2,n-3을 구하는 문제로 바꿀 수 있다.

<DP Array 설계>

ㄹ. dp array에 담을 dp[n]은 ’n을 1,2,3의 조합으로 나타내는 방법의 수’로 설정해야한다.

<dp[n] 구하는 방법과 dp적용의 당위성>

ㅁ. dp[n]은 dp[n-1] + dp[n-2] + dp[n-3]으로 구할 수 있다.

-> n을 구하는 것을 n-1, n-2, n-3을 구하는 문제 중 하나로 쪼갤 수 있고, 문제 푸는 구조가 같으므로, 다이내믹 프로그래밍으로 풀 수 있다.

<시간복잡도 분석>

ㅂ. 차례대로 재귀또는 for문으로 호출하고 (O(N)), 함수또는 for문 내에서 단순 연산을 진행하므로 (O(1)), 시간복잡도는 O(N)이다.

**Remarks\_**

테스트케이스의 수 T가 주어진다.

ㄱ. 문제를 푸는 경우의 다양한 예제 입력 대신에 테스트케이스의 수가 주어지고, 여러 줄에 걸쳐서 2개 이상의 예제가 주어질 수 있다.

ㄴ.  while(testcasenum—){}을 통해서 각각의 테스트케이스의 입력을 받아서 주어진 문제를 풀면 된다.

ㄷ. 이때, 전역배열과 각종 변수들을 원래대로 초기화하여 각각의 테스트케이스가 다른 테스트케이스에 영향을 미치지 않도록 하는 것은 중요하다.

**Answer\_**

<https://www.acmicpc.net/source/13667245>

**<Baekjoon 11052> 카드 구매하기**

dp[n]을 구하는 연산을 할 때, dp의 원소가 아닌 다른 부가적인 요소를 불러와서 연산이 진행될 수도 있다.

**문제**

요즘 민규네 동네에서는 스타트링크에서 만든 PS카드를 모으는 것이 유행이다.

PS카드는 PS(Problem Solving)분야에서 유명한 사람들의 아이디와 얼굴이 적혀있는 카드이다. 각각의 카드에는 등급을 나타내는 색이 칠해져 있고, 다음과 같이 8가지가 있다.

* **전설카드**
* **레드카드**
* **오렌지카드**
* **퍼플카드**
* **블루카드**
* **청록카드**
* **그린카드**
* **그레이카드**

카드는 카드팩의 형태로만 구매할 수 있고, 카드팩의 종류는 카드 1개가 포함된 카드팩, 카드 2개가 포함된 카드팩, ... 카드 N개가 포함된 카드팩과 같이 총 N가지가 존재한다.

민규는 카드의 개수가 적은 팩이더라도 가격이 비싸면 높은 등급의 카드가 많이 들어있을 것이라는 미신을 믿고 있다. 따라서, 민규는 돈을 최대한 많이 지불해서 카드 N개 구매하려고 한다. 카드가 i개 포함된 카드팩의 가격은 Pi원이다.

예를 들어, 카드팩이 총 4가지 종류가 있고, P1 = 1, P2 = 5, P3 = 6, P4 = 7인 경우에 민규가 카드 4개를 갖기 위해 지불해야 하는 금액의 최댓값은 10원이다. 2개 들어있는 카드팩을 2번 사면 된다.

P1 = 5, P2 = 2, P3 = 8, P4 = 10인 경우에는 카드가 1개 들어있는 카드팩을 4번 사면 20원이고, 이 경우가 민규가 지불해야 하는 금액의 최댓값이다.

마지막으로, P1 = 3, P2 = 5, P3 = 15, P4 = 16인 경우에는 3개 들어있는 카드팩과 1개 들어있는 카드팩을 구매해 18원을 지불하는 것이 최댓값이다.

카드 팩의 가격이 주어졌을 때, N개의 카드를 구매하기 위해 민규가 지불해야 하는 금액의 최댓값을 구하는 프로그램을 작성하시오. N개보다 많은 개수의 카드를 산 다음, 나머지 카드를 버려서 N개를 만드는 것은 불가능하다. 즉, 구매한 카드팩에 포함되어 있는 카드 개수의 합은 N과 같아야 한다.

**입력**

첫째 줄에 민규가 구매하려고 하는 카드의 개수 N이 주어진다. (1 ≤ N ≤ 1,000)

둘째 줄에는 Pi가 P1부터 PN까지 순서대로 주어진다. (1 ≤ Pi ≤ 10,000)

**출력**

첫째 줄에 민규가 카드 N개를 갖기 위해 지불해야 하는 금액의 최댓값을 출력한다.

예제입력1 4        예제입력2 5            예제입력3 10                                          예제입력4 10

       1 5 6 7         10 9 8 7 6           1 1 2 3 5 8 13 21 34 55         5 10 11 12 13 30 35 40 45 47

예제출력1 10               예제출력2 50                        예제출력3 55                                                  예제출력4 50

<문제 풀이 방향>

ㄱ. 직접 경우의 수를 구하는 것은 예제입력에서 어렵다는 것을 알 수 있다.

ㄴ. 타일링 문제와 다른점은 마지막에 1개짜리를 구매한 것은 마지막에 2x1타일을 깐 것처럼 놓으면 겹치는 경우가 생길지에 대한 우려 때문이다.

ㄷ. 하지만, dp문제가 다 그렇듯, dp에서는 겹칠지에 대한 여부를 걱정하는 것이 아니라, ‘큰 문제가 작은 문제로 나뉘는지’, ‘큰 문제와 작은 문제의 푸는 방법이 같은지’만을 체크하면 된다.

ㄹ. 따라서 마지막에 1개짜리, 2개짜리, … n개짜리 중에 무엇을 샀는지에 대해서 신경을 써주고, 그 중 가장 많은 돈을 쓴 경우를 dp[n]에 넣어주면 된다.

<DP Array 설계>

ㄹ. dp array에 담을 dp[n]은 ’n개를 살 때 쓸 수 있는 최대 돈의 액수’로 설정해야한다.

<dp[n] 구하는 방법과 dp적용의 당위성>

ㅁ. dp[n]은 max{ dp[n-1]+amount[1] , dp[n-2]+amount[2] , dp[n-3]+amount[3] , … dp[0]+amount[n] } 으로 구할 수 있다.

-> n을 구하는 것을 n-1, n-2, n-3, … , 0을 구하는 문제 중 하나로 쪼갤 수 있고, 문제 푸는 구조가 같으므로, 다이내믹 프로그래밍으로 풀 수 있다.

-> 이 문제를 통해 dp[n]을 구할 때 dp[@]뿐만이 아니라 다른 배열에 들어있는 요소(돈 정보)를 바탕으로 연산이 진행되는 문제도 있다는 것을 알 수 있다.

<시간복잡도 분석>

ㅂ. 차례대로 재귀또는 for문으로 호출하고 (O(N)), 함수또는 for문 내에서 n개의 요소에 대한 비교를 진행하므로 (O(N)), 시간복잡도는 O(N^2)이다.

**Remarks\_**

dp[0]을 항상 채워야하는가

ㄱ. 보통은 dp[1]을 초기화하거나, n==1 이면 재귀함수를 종료하게 하는 등의 방법을 통해 1부터 dp배열을 초기화해 나간다.

ㄴ. 하지만, 이 문제 같은 경우는 dp[0]+amount[n]인 경우도 연산해야 하므로, dp[n]을 구하는 과정에서 dp[0]의 값을 필수로 한다.

ㄷ. dp[0]은 의미를 갖지 않고, 위의 연산을 수행하기위해서 필요한 값으로 초기화시키면 된다.

**Answer\_**

<https://www.acmicpc.net/source/13667880>

**<Baekjoon 2193> 이친수**

dp연산을 하는 도중에 마지막 요소에 따라서 dp[n]을 구하는 방법이 바뀌면, 2차원 dp array를 이용한다.

**문제**

0과 1로만 이루어진 수를 이진수라 한다. 이러한 이진수 중 특별한 성질을 갖는 것들이 있는데, 이들을 이친수(pinary number)라 한다. 이친수는 다음의 성질을 만족한다.

1. 이친수는 0으로 시작하지 않는다.
2. 이친수에서는 1이 두 번 연속으로 나타나지 않는다. 즉, 11을 부분 문자열로 갖지 않는다.

예를 들면 1, 10, 100, 101, 1000, 1001 등이 이친수가 된다. 하지만 0010101이나 101101은 각각 1, 2번 규칙에 위배되므로 이친수가 아니다.

N(1 ≤ N ≤ 90)이 주어졌을 때, N자리 이친수의 개수를 구하는 프로그램을 작성하시오.

**입력**

첫째 줄에 N이 주어진다.

**출력**

첫째 줄에 N자리 이친수의 개수를 출력한다.

예제입력1 3

예제출력1 2

<문제 풀이 방향>

ㄱ. 앞에서 다룬 dp문제들과는 달리 이 문제에서는 dp[n]을 구하는 식이 n-1번째 자리에 0,1 중에서 어떤 수가 들어갔는지에 따라서 변한다는 것을 알 수 있다.

ㄴ. 따라서 dp를 2차원 배열로 두어서, dp[k번째][어떠한 상태]를 나타낼 수 있다.

ㄷ. 이 문제에서는 어떠한 상태가 ’n-1번째 자리에 들어간 수’가 된다.

<DP Array 설계>

ㄹ. dp array에 담을 dp[n][k]는 n을 ’n자리 수에서 이친수의 개수’로, k를 ’n-1자릿수’로 설정해야한다.

<dp[n] 구하는 방법과 dp적용의 당위성>

ㅁ. dp[n]은 dp[n][0]+dp[n][1]로 구할 수 있다.

ㅂ. dp[n][0]은 dp[n-1][0]+dp[n-1][1], dp[n][1]은 dp[n-1][0]을 통해서 구할 수 있는데, 이때 dp[n][@]은 dp[n-1][@]을 구하는 문제로 쪼갤 수 있고,

     문제 푸는 구조가 같으므로, 다이내믹 프로그래밍으로 풀 수 있다.

<시간복잡도 분석>

ㅂ. 차례대로 재귀또는 for문으로 호출하고 (O(N)), 함수또는 for문 내에서 단순 연산을 진행하므로 (O(1)), 시간복잡도는 O(N)이다.

**Remarks\_**

'어떠한 상태’를 표현하기

ㄱ. dp문제를 풀 때, dp[n]을 쉽게 표현할 수 있는 문제(점화식이 명확하고, 케이스 분류를 안해도 되는 문제)는 쉬운 문제이다.

ㄴ. 수준 있는 dp문제는 대부분 케이스에 따라서 dp[n]을 표현하는 식이 다르게 나타난다.

ㄷ. 각각의 상황에 맞게 dp[n]을 구하는 식을 세울 때, if문을 통해서 dp[n][어떠한 상태]로 잘 나누는 것이 dp문제를 잘 풀 수 있는 핵심적인 방법이다.

언제 int를 쓰고, 언제 long long을 써야하는가

ㄱ. 이 문제는 10007과 같은 소수로 나눈 나머지를 출력하라고 하지 않았기 때문에, int로 출력하면 산술 overflow가 뜰 것이라는 것을 예상하기 힘들다.

ㄴ. 입력에서 주어지는 범위가 매우 크면 보통 long long이라고 두는 것이 맞다는 생각을 항상 해야한다.

**Answer\_**

<https://www.acmicpc.net/source/13685555>

**<Baekjoon 10844> 쉬운 계단 수**

dp연산을 하는 도중에 마지막 요소에 따라서 dp[n]을 구하는 방법이 바뀌면, 2차원 dp array를 이용한다.

**문제**

45656이란 수를 보자.

이 수는 인접한 모든 자리수의 차이가 1이 난다. 이런 수를 계단 수라고 한다.

세준이는 수의 길이가 N인 계단 수가 몇 개 있는지 궁금해졌다.

N이 주어질 때, 길이가 N인 계단 수가 총 몇 개 있는지 구하는 프로그램을 작성하시오. (0으로 시작하는 수는 없다.)

**입력**

첫째 줄에 N이 주어진다. N은 1보다 크거나 같고, 100보다 작거나 같은 자연수이다.

**출력**

첫째 줄에 정답을 1,000,000,000으로 나눈 나머지를 출력한다.

예제입력1 1                예제입력2 2

예제출력1 9                예제출력2 17

<문제 풀이 방향>

ㄱ. dp[n]을 구하는 식이 n-1번째 자리에 0~9 중에서 어떤 수가 들어갔는지에 따라서 변한다는 것을 알 수 있다.

ㄴ. 따라서 dp를 2차원 배열로 두어서, dp[k번째][어떠한 상태]를 나타낼 수 있다.

ㄷ. 이 문제에서는 어떠한 상태가 ’n-1번째 자리에 들어간 수’가 된다.

<DP Array 설계>

ㄹ. dp array에 담을 dp[n][k]는 n을 ’길이가 n인 계단 수의 개수’로, k를 ’n-1자릿수’로 설정해야한다.

<dp[n] 구하는 방법과 dp적용의 당위성>

ㅁ. dp[n]은 dp[n][0]+dp[n][1]+…+dp[n][9]로 구할 수 있다.

ㅂ. 임의의 dp[n][i]는(단, i는 0~9) dp[n-1][i-1] + dp[n-1][i+1]로 구할 수 있다. 이때 dp[n][@]은 dp[n-1][@]을 구하는 문제로 쪼갤 수 있고,

     문제 푸는 구조가 같으므로, 다이내믹 프로그래밍으로 풀 수 있다.

<시간복잡도 분석>

ㅂ. 차례대로 재귀또는 for문으로 호출하고 (O(N)), 함수또는 for문 내에서 단순 연산을  n번 진행하므로 (O(1) X O(N)), 시간복잡도는 O(N^2)이다.

**Remarks\_**

'어떠한 상태’를 표현하기

ㄱ. dp문제를 풀 때, dp[n]을 쉽게 표현할 수 있는 문제(점화식이 명확하고, 케이스 분류를 안해도 되는 문제)는 쉬운 문제이다.

ㄴ. 수준 있는 dp문제는 대부분 케이스에 따라서 dp[n]을 표현하는 식이 다르게 나타난다.

ㄷ. 각각의 상황에 맞게 dp[n]을 구하는 식을 세울 때, if문을 통해서 dp[n][어떠한 상태]로 잘 나누는 것이 dp문제를 잘 풀 수 있는 핵심적인 방법이다.

정답을 1,000,000,000으로 나눈 나머지를 출력하시오.

ㄱ. 앞의 문제와 달리 소수가 아닌 수로 나눈 나머지를 출력하는 문제도 있을 수 있다.

ㄴ. int형의 범위는 대략 20억 까지인데, 이 문제에서는 10억으로 나눈 나머지를 출력하라고 했으므로,

     dp[n]에 들어갈 값을 구한 후에 10억으로 나누어 주면, dp array를 int형 배열로 설정하여도 된다.

ㄷ. 어떠한 수로 나눈 나머지를 출력하라는 문제에서는 무조건 배열에 나눈 나머지를 저장해놓아도 된다. ( 이유는 정수론 )

**Answer\_**

<https://www.acmicpc.net/source/13669226>

**<Baekjoon 11057> 오르막 수**

dp연산을 하는 도중에 마지막 요소에 따라서 dp[n]을 구하는 방법이 바뀌면, 2차원 dp array를 이용한다.

**문제**

오르막 수는 수의 자리가 오름차순을 이루는 수를 말한다. 이때, 인접한 수가 같아도 오름차순으로 친다.

예를 들어, 2234와 3678, 11119는 오르막 수이지만, 2232, 3676, 91111은 오르막 수가 아니다.

수의 길이 N이 주어졌을 때, 오르막 수의 개수를 구하는 프로그램을 작성하시오. 수는 0으로 시작할 수 있다.

**입력**

첫째 줄에 N (1 ≤ N ≤ 1,000)이 주어진다.

**출력**

첫째 줄에 길이가 N인 오르막 수의 개수를 10,007로 나눈 나머지를 출력한다.

예제입력1 1                 예제입력2 2                예제입력3 3

예제출력1 10                예제출력2 55               예제출력3 220

<문제 풀이 방향>

ㄱ. dp[n]을 구하는 식이 n-1번째 자리에 0~9 중에서 어떤 수가 들어갔는지에 따라서 변한다는 것을 알 수 있다.

ㄴ. 따라서 dp를 2차원 배열로 두어서, dp[k번째][어떠한 상태]를 나타낼 수 있다.

ㄷ. 이 문제에서는 어떠한 상태가 ’n-1번째 자리에 들어간 수’가 된다.

<DP Array 설계>

ㄹ. dp array에 담을 dp[n][k]는 n을 ’길이가 n인 오르막 수의 개수’로, k를 ’n-1자릿수’로 설정해야한다.

<dp[n] 구하는 방법과 dp적용의 당위성>

ㅁ. dp[n]은 dp[n][0]+dp[n][1]+…+dp[n][9]로 구할 수 있다.

ㅂ. 임의의 dp[n][i]는(단, i는 0~9) dp[n-1][0] + … + dp[n-1][i]로 구할 수 있다. 이때 dp[n][@]은 dp[n-1][@]을 구하는 문제로 쪼갤 수 있고,

     문제 푸는 구조가 같으므로, 다이내믹 프로그래밍으로 풀 수 있다.

<시간복잡도 분석>

ㅂ. 차례대로 재귀또는 for문으로 호출하고 (O(N)), 함수또는 for문 내에서 단순 연산을  n번 진행하므로 (O(1) X O(N)), 시간복잡도는 O(N^2)이다.

**Remarks\_**

'어떠한 상태’를 표현하기

ㄱ. dp문제를 풀 때, dp[n]을 쉽게 표현할 수 있는 문제(점화식이 명확하고, 케이스 분류를 안해도 되는 문제)는 쉬운 문제이다.

ㄴ. 수준 있는 dp문제는 대부분 케이스에 따라서 dp[n]을 표현하는 식이 다르게 나타난다.

ㄷ. 각각의 상황에 맞게 dp[n]을 구하는 식을 세울 때, if문을 통해서 dp[n][어떠한 상태]로 잘 나누는 것이 dp문제를 잘 풀 수 있는 핵심적인 방법이다.

**Answer\_**

<https://www.acmicpc.net/source/13700168>

**<Baekjoon 9465> 스티커**

+ 2579

dp연산을 하는 도중에 마지막 상황에 따라서 dp[n]을 구하는 방법이 바뀌면, 상황을 표현할 방법을 정의한 뒤, 2차원 dp array를 이용한다.

**문제**

상근이의 여동생 상냥이는 문방구에서 스티커 2n개를 구매했다. 스티커는 그림 (a)와 같이 2행 n열로 배치되어 있다. 상냥이는 스티커를 이용해 책상을 꾸미려고 한다.

상냥이가 구매한 스티커의 품질은 매우 좋지 않다. 스티커 한 장을 떼면, 그 스티커와 변을 공유하는 스티커는 모두 찢어져서 사용할 수 없게 된다. 즉, 뗀 스티커의 왼쪽, 오른쪽, 위, 아래에 있는 스티커는 사용할 수 없게 된다.



모든 스티커를 붙일 수 없게된 상냥이는 각 스티커에 점수를 매기고, 점수의 합이 최대가 되게 스티커를 떼어내려고 한다. 먼저, 그림 (b)와 같이 각 스티커에 점수를 매겼다. 상냥이가 뗄 수 있는 스티커의 점수의 최댓값을 구하는 프로그램을 작성하시오. 즉, 2n개의 스티커 중에서 점수의 합이 최대가 되면서 서로 변을 공유 하지 않는 스티커 집합을 구해야 한다.

위의 그림의 경우에 점수가 50, 50, 100, 60인 스티커를 고르면, 점수는 260이 되고 이 것이 최대 점수이다. 가장 높은 점수를 가지는 두 스티커 (100과 70)은 변을 공유하기 때문에, 동시에 뗄 수 없다.

**입력**

첫째 줄에 테스트 케이스의 개수 T가 주어진다. 각 테스트 케이스의 첫째 줄에는 n (1 ≤ n ≤ 100,000)이 주어진다. 다음 두 줄에는 n개의 정수가 주어지며, 각 정수는 그 위치에 해당하는 스티커의 점수이다. 연속하는 두 정수 사이에는 빈 칸이 하나 있다. 점수는 0보다 크거나 같고, 100보다 작거나 같은 정수이다.

**출력**

각 테스트 케이스 마다, 2n개의 스티커 중에서 두 변을 공유하지 않는 스티커 점수의 최댓값을 출력한다.

예제입력1                                         예제출력1

2                         260

5                         290

50 10 100 20 40

30 50 70 10 60

7

10 30 10 50 100 20 40

20 40 30 50 60 20 80

<문제 풀이 방향>

ㄱ. 점수가 높은 스티커부터 떼는 식으로 직접적으로 점수를 계산하려고 하는 것은 미개한 짓이다.

ㄴ. dp[n]을 구하는 식이 n-1번째 열에서 위, 아래 스티커를 땠는지 여부에 따라서 변한다는 것을 알 수 있다.

ㄷ. 따라서 dp를 2차원 배열로 두어서, dp[k번째][어떠한 상태]를 나타낼 수 있다.

ㄹ. 이 문제에서는 어떠한 상태가 ’n번째 열에서 어떠한 스티커를 뗐는지 여부’가 된다.

ㅁ. n-1번째 열에서 스티커를 떼지 않았을 때 0, 위쪽 스티커를 떼었을 때를 1, 아래쪽 스티커를 떼었을 때를 2로 정의한다.

<DP Array 설계>

ㄹ. dp array에 담을 dp[n][k]는 n을 ’n번째 열까지 스티커를 떼었을 때 얻을 수 있는 점수의 최댓값’으로, k를 ’n자리에서 스티커를 뗀 상황’으로 설정해야한다.

<dp[n] 구하는 방법과 dp적용의 당위성>

ㅁ. dp[n]은 dp[n][0]+dp[n][1]+dp[n][2]로 구할 수 있다.

ㅂ. 임의의 dp[n][i]는(단, i는 0~2) i값에 따라서 구하는 방법이 바뀐다.

ㅅ. dp[n][0] 는 max(dp[n-1][0], dp[n-1][1], dp[n-1][2]) 로 구할 수 있다.

ㅇ. dp[n][1] 은 max(dp[n-1][0],dp[n-1][2])+stickerpoint[1][n]로 구할 수 있다.

ㅈ. dp[n][2]는 max(dp[n-1][0],dp[n-1][1])+stickerpoint[2][n]로 구할 수 있다.

ㅊ. 이때 dp[n][@]은 dp[n-1][@]을 구하는 문제로 쪼갤 수 있고, 문제 푸는 구조가 같으므로, 다이내믹 프로그래밍으로 풀 수 있다.

<시간복잡도 분석>

ㅂ. 차례대로 재귀또는 for문으로 호출하고 (O(N)), 함수또는 for문 내에서 단순 연산을 진행하므로 (O(1)), 시간복잡도는 O(N)이다.

**Remarks\_**

dp의 원소가 아닌 다른 부가적인 요소를 불러와서 연산이 진행될 수도 있다.

ㄱ. dp문제를 풀 때, 2차원 배열을 두어서 dp array를 설정해야 할 때, 부가적인 요소가 담긴 array와 햇갈리기 딱 좋다.

ㄴ. dp array는 [n번째][k어떠한 상황] 으로 표현되는 반면, scorearray[행][열]로 표현되기 때문이다.

-> ‘[ ]’안에 있는 인자들 간의 순서에 유의해야 한다.

**Answer\_**

<https://www.acmicpc.net/source/13701434>

2579 :

<https://www.acmicpc.net/source/14087018>

**<Baekjoon 2156> 포도주 시식**

dp연산을 하는 도중에 마지막 상황에 따라서 dp[n]을 구하는 방법이 바뀌면, 상황을 표현할 방법을 정의한 뒤, 2차원 dp array를 이용한다.

**문제**

효주는 포도주 시식회에 갔다. 그 곳에 갔더니, 테이블 위에 다양한 포도주가 들어있는 포도주 잔이 일렬로 놓여 있었다. 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.

효주는 될 수 있는 대로 많은 양의 포도주를 맛보기 위해서 어떤 포도주 잔을 선택해야 할지 고민하고 있다. 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.

예를 들어 6개의 포도주 잔이 있고, 각각의 잔에 순서대로 6, 10, 13, 9, 8, 1 만큼의 포도주가 들어 있을 때, 첫 번째, 두 번째, 네 번째, 다섯 번째 포도주 잔을 선택하면 총 포도주 양이 33으로 최대로 마실 수 있다.

**입력**

첫째 줄에 포도주 잔의 개수 n이 주어진다. (1≤n≤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.

예제입력1                                         예제출력1

6                         33

6

10

13

9

8

1

<문제 풀이 방향>

ㄱ. dp[n]을 구하는 식이 n번째 포도주를 마실지 말지 정하기 전에 연속으로 마신 잔의 수에 따라서 변한다는 것을 알 수 있다.

ㄷ. 따라서 dp를 2차원 배열로 두어서, dp[k번째][어떠한 상태]를 나타낼 수 있다.

ㄹ. 이 문제에서는 어떠한 상태가 ’n번째 잔 직전까지 연속으로 마신 잔의 수’가 된다.

ㅁ. n-1번째 잔을 마시지 않았을 때 0, n-1번째 잔을 마셨을 때를 1, n-2, n-1번째 잔을 마셨을 때를 2로 정의한다.

<DP Array 설계>

ㄹ. dp array에 담을 dp[n][k]는 n을 ’n번째 포도주 잔을 마실 때 마실 수 있는 양의 최댓값’으로, k를 ’n번째 잔 직전까지 연속으로 마신 잔의 수’으로 설정해야한다.

<dp[n] 구하는 방법과 dp적용의 당위성>

ㅁ. dp[n]은 dp[n][0]+dp[n][1]+dp[n][2]로 구할 수 있다.

ㅂ. 임의의 dp[n][i]는(단, i는 0~2) i값에 따라서 구하는 방법이 바뀐다.

ㅅ. dp[n][0] 는 max(dp[n-1][0], dp[n-1][1], dp[n-1][2]) 로 구할 수 있다.

ㅇ. dp[n][1] 은dp[n-1][0]+amount[n]로 구할 수 있다.

ㅈ. dp[n][2]는 dp[n-1][1]+amount[n]로 구할 수 있다.

ㅊ. 이때 dp[n][@]은 dp[n-1][@]을 구하는 문제로 쪼갤 수 있고, 문제 푸는 구조가 같으므로, 다이내믹 프로그래밍으로 풀 수 있다.

<시간복잡도 분석>

ㅂ. 차례대로 재귀또는 for문으로 호출하고 (O(N)), 함수또는 for문 내에서 단순 연산을 진행하므로 (O(1)), 시간복잡도는 O(N)이다.

**Remarks\_**

'어떠한 상태’를 표현하기

ㄱ. dp문제를 풀 때, dp[n]을 쉽게 표현할 수 있는 문제(점화식이 명확하고, 케이스 분류를 안해도 되는 문제)는 쉬운 문제이다.

ㄴ. 수준 있는 dp문제는 대부분 케이스에 따라서 dp[n]을 표현하는 식이 다르게 나타난다.

ㄷ. 각각의 상황에 맞게 dp[n]을 구하는 식을 세울 때, if문을 통해서 dp[n][어떠한 상태]로 잘 나누는 것이 dp문제를 잘 풀 수 있는 핵심적인 방법이다.

ㄹ. 어떠한 상황이 연속으로 무엇을 한 상태를 나타낼 때, ‘n번째까지 하였을 때 수’보다는 ’n-1번째까지 한 수’를 나타내는 것이 더 좋다.

-> 이 문제 같은 경우는 n-1번째까지 연속으로 마신 잔의 수를 k로 잡는 것이 좋다.

**Answer\_**

<https://www.acmicpc.net/source/13701887>

**<Baekjoon 11053> 가장 긴 증가하는 부분 수열(LIS)**

+ 11055, 11722, 11054, 1912

dp[n]을 출력에서 요구하는 값으로 바로 표현하기 어려우면, dp[n]을 ’n번째를 선택했을 때’로 전환하여 풀 수 있다.

**문제**

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.

예제입력1 6                                    예제출력1 4

       10 20 10 30 20 50

<문제 풀이 방향>

ㄱ. 앞의 문제와 이 문제가 다른 점은 dp[n]을 정할 때, ’n번째 수까지 고려했을 때, 가장 긴 부분 수열의 길이’로 두면 너무 복잡해진다는 것이다.

ㄴ. 보통의 dp문제는 dp[n]을 출력하려는 값과 일치하게 설정하면 되었지만, 그렇게 하면 문제가 지나치게 복잡해지는 경우도 있다.

ㄷ. 이에 대한 해결책은 dp[n]을 ’n번째 수를 마지막으로 하는 가장 긴 부분 수열의 길이’로 두는 것이다.

ㄹ. 이 경우에는 dp[1]~dp[n]중에서 가장 큰 값이 ’n번째 수까지 고려했을 때, 가장 긴 부분 수열의 길이’가 된다.

<DP Array 설계>

ㅁ. dp array에 담을 dp[n]을 ’n번째 수를 마지막으로 하는 가장 긴 부분 수열의 길이’으로 설정해야한다.

<dp[n] 구하는 방법과 dp적용의 당위성>

ㅂ. dp[n]은 수열 a[1],a[2],…a[n]중에서 j<n을 만족하고, a[j]<a[n]을 만족하는 j에 대해서 dp[n] = max(dp[j])+1로 구할 수 있다.

ㅅ. 이때 dp[n]은 dp[@]를 구하는 문제로 쪼갤 수 있고, 문제 푸는 구조가 같으므로, 다이내믹 프로그래밍으로 풀 수 있다.

<시간복잡도 분석>

ㅇ. 차례대로 재귀또는 for문으로 호출하고 (O(N)), 함수또는 for문 내에서 차례대로 비교를 진행하므로 (O(N)), 시간복잡도는 O(N^2)이다.

**Remarks\_**

’n까지 고려했을 때’와 ’n번째를 하였을 때’의 차이

ㄱ. dp문제를 풀 때, 문제 출력이 요구하는 n번째 까지 고려하였을 때 최댓값을 dp array에 저장하는 쉬운 문제이다.

ㄴ. 수준 있는 dp문제는 이와 같이 생각하면 너무 생각할 것이 많아지거나, 시간복잡도가 O(N^3)처럼 지나치게 커지는 경우가 다반사이다.

ㄷ. 생각을 전환하여서 n번째를 하였을 때에 대한 문제로 바꾸고, dp array에 담긴 값중에 최댓값을 고르는것이 dp문제를 잘 풀 수 있는 핵심적인 방법이다.

**Answer\_**

<https://www.acmicpc.net/source/14086247>

**<Baekjoon 1912> 연속합**

dp[n]을 출력에서 요구하는 값으로 바로 표현하기 어려우면, dp[n]을 ’n번째를 선택했을 때’로 전환하여 풀 수 있다.

**문제**

n개의 정수로 이루어진 임의의 수열이 주어진다. 우리는 이 중 연속된 몇 개의 수를 선택해서 구할 수 있는 합 중 가장 큰 합을 구하려고 한다. 단, 수는 한 개 이상 선택해야 한다.

예를 들어서 10, -4, 3, 1, 5, 6, -35, 12, 21, -1 이라는 수열이 주어졌다고 하자. 여기서 정답은 12+21인 33이 정답이 된다.

**입력**

첫째 줄에 정수 n(1 ≤ n ≤ 100,000)이 주어지고 둘째 줄에는 n개의 정수로 이루어진 수열이 주어진다. 수는 -1,000보다 크거나 같고, 1,000보다 작거나 같은 정수이다.

**출력**

첫째 줄에 답을 출력한다.

예제입력1 10                                                    예제출력1 33

       10 -4 3 1 5 6 -35 12 21 -1

<문제 풀이 방향>

ㄴ. 앞의 문제와 이 문제가 다른 점은 dp[n]을 정할 때, ’n번째 수까지 고려했을 때, 가장 긴 부분 수열의 길이’로 두면 너무 복잡해진다는 것이다.

ㄷ. 보통의 dp문제는 dp[n]을 출력하려는 값과 일치하게 설정하면 되었지만, 그렇게 하면 문제가 지나치게 복잡해지는 경우도 있다.

ㄹ. 이에 대한 해결책은 dp[n]을 ’n번째 수를 마지막으로 하는 가장 큰 연속합’로 두는 것이다.

ㅁ. 이 경우에는 dp[1]~dp[n]중에서 가장 큰 값이 ’n번째 수까지 고려했을 때, 가장 긴 부분 수열의 길이’가 된다.

<DP Array 설계>

ㅂ. dp array에 담을 dp[n]을 ’n번째 수를 마지막으로 하는 가장 큰 연속합’으로 설정해야한다.

<dp[n] 구하는 방법과 dp적용의 당위성>

ㅅ. a[n]은 a[n-1]을 마지막으로 하는 연속합에 포함이 되거나, a[n]을 시작으로 하는 새로운 연속합이 되어야한다.

ㅇ. 따라서, dp[n] = max(dp[n-1]+a[n], a[n])이 되어야한다.

ㅇ. 이때 dp[n]은 dp[@]를 구하는 문제로 쪼갤 수 있고, 문제 푸는 구조가 같으므로, 다이내믹 프로그래밍으로 풀 수 있다.

<시간복잡도 분석>

ㅂ. 차례대로 재귀또는 for문으로 호출하고 (O(N)), 함수또는 for문 내에서 단순 비교를 진행하므로 (O(1)), 시간복잡도는 O(N)이다.

**Remarks\_**

’n번째를 하였을 때’의 문제를 풀 때 조심할 점

ㄱ. dp문제를 풀 때, 문제 출력이 요구하는 n번째 까지 고려하였을 때 최댓값을 dp array에 저장하는 쉬운 문제이다.

ㄴ. 수준 있는 dp문제는 이와 같이 생각하면 너무 생각할 것이 많아지거나, 시간복잡도가 O(N^3)처럼 지나치게 커지는 경우가 다반사이다.

ㄷ. 생각을 전환하여서 n번째를 하였을 때에 대한 문제로 바꾸고, dp array에 담긴 값중에 최댓값을 고르는것이 dp문제를 잘 풀 수 있는 핵심적인 방법이다.

ㄹ. 이와 같은 문제는 마지막에 출력할 때, max(dp[1]~dp[n])인 경우가 많다.

ㅁ. 이때 음수의 값도 dp[i]에 들어갈 수 있으므로, answ를 initialize할 때, 조심해야한다.

**Answer\_**

<https://www.acmicpc.net/source/14086407>

Extra Problems\_

**<Baekjoon 1912> 연속합**

dp[n] = max(dp[n-1], dp[n-4], dp[n-9], …)

**<Baekjoon 9461> 파도반 수열**

dp[n] = dp[n-1] + dp[n-5]