

1. Unique Paths

2. Climbing Stairs

3. Coin Change

4. 가장 긴 서브시퀀스 증가

(Longest Increasing Subsequence)

문제 1) Unique Path

Problem

A robot is located at the top-left corner of a $m \times n$ grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time.

The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there? (7*3)

start						
						Finish

Note: m and n will be at most 100.

Example 1:

Input: $m = 3, n = 2$

Output: 3

Explanation: From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:

1. Right -> Right -> Down
2. Right -> Down -> Right
3. Down -> Right -> Right

Unique Path

Problem

1 (0,0)	1 (0,1)	1 (0,2)
1 (1,0)	2 (1,1)	3 (1,2)

Input : $m=3$, $n=2$

Output : 3

1. Right -> Right -> Down
2. Right -> Down -> Right
3. Down -> Right -> Right

Input : $m=7$, $n=3$

Output : 28

Example

1	1	1
1	2	3
1	3	6
1	4	10

문제 2) Climbing Stairs

Problem

You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Note: Given n will be a positive integer.

Example 1:

Input: 2

Output: 2

Explanation: There are two ways to climb to the top.

1. 1 step + 1 step

2. 2 steps

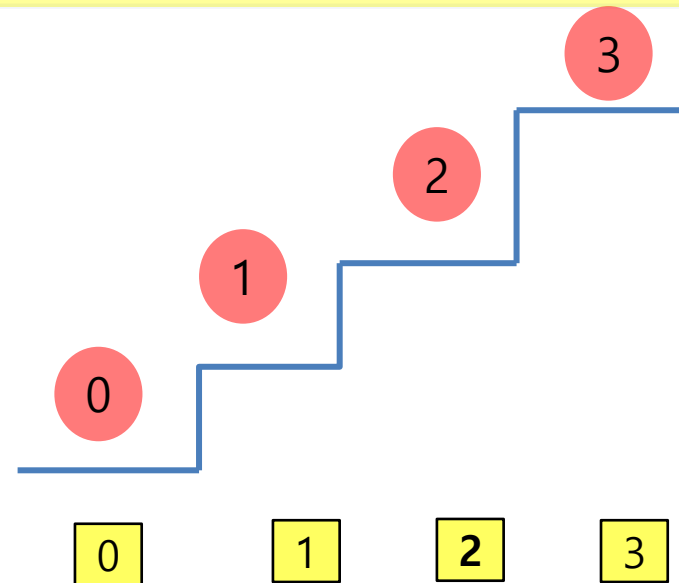
Climbing Stairs

Problem

Input: 3

Output: 3

1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step



Solution

$$dp[3] = dp[2] + dp[1]$$

$$dp[i] = dp[i-1] + dp[i-2]$$

문제 3) Coin Change

Problem

You are given coins of different denominations and a total amount of money amount. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

Example 1:

Input: coins = [1, 2, 5], amount = 11

Output: 3

Explanation: $11 = 5 + 5 + 1$

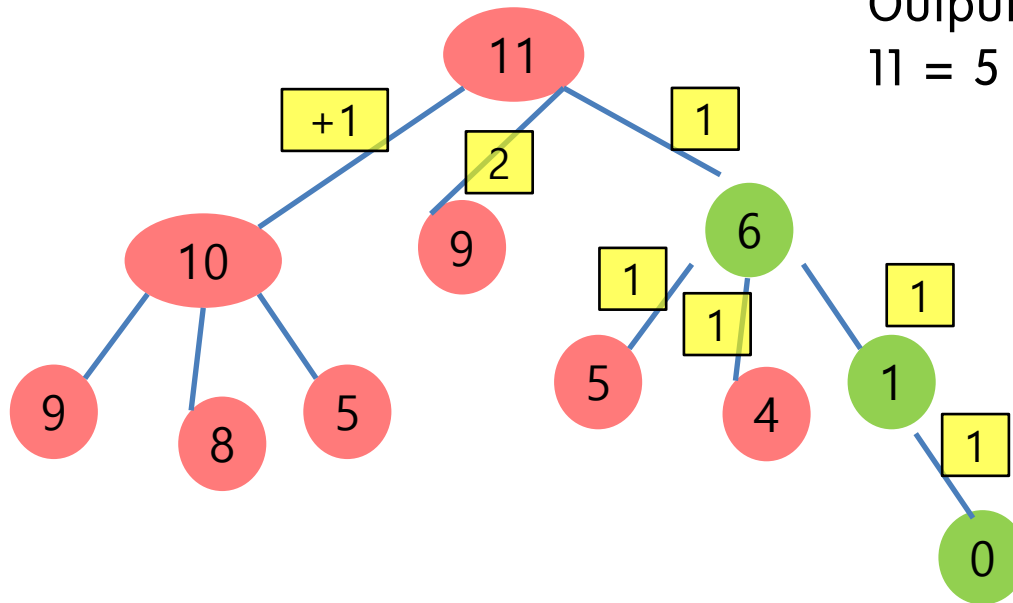
Coin Change

Problem

Input: coins = [1, 2, 5], amount = 11

Output: 3

11 = 5 + 5 + 1



dp[0]=0 을 기준으로 +1

Solution

0	1	2	3	4	5	6	7	8	9	1	1
0	1	1	1	1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2	2	2	2	2
0	1	1	2	2	1	2	2	3	3	2	3

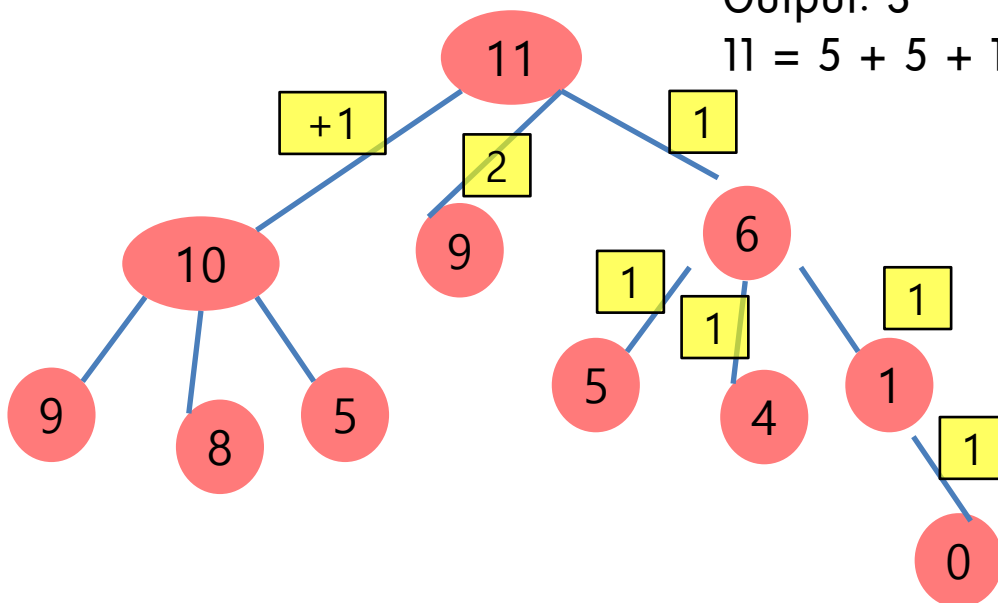
Coin Change

Problem

Input: coins = [1, 2, 5], amount = 11

Output: 3

11 = 5 + 5 + 1



$dp[i] = \text{Math.min}(dp[i], dp[i - \text{coins}[j]] + 1);$

$dp[1] = \text{Math.min}(dp[1], dp[1 - \text{coins}[j]] + 1);$

1-1 = $dp[0] = 0$

1-2 (X)

1-5 (X)

$dp[5] = \text{Math.min}(dp[5], dp[5 - \text{coins}[j]] + 1);$

5-1 = $dp[4] = 2$

5-2 = $dp[3] = 2$

5-5 = $dp[0] = 0$

Solution

0	1	2	3	4	5	6	7	8	9	10	11
0	12	12	12	12	12	12	12	12	12	12	12
0	1	1	2	2	1	2	2	3	3	2	3

가장 긴 서브시퀀스 증가 (Longest Increasing Subsequence)

설명

정수 배열 `nums`가 주어집니다.

정수가 증가하는 가장 긴 sub sequence의 길이를 반환합니다.

정수의 배열에서 연속적으로 증가하는 부분만 선택하면 됩니다.

예를 들어 `[3,6,2,7]`는 `[0,3,1,6,2,2,7]` 배열의 하위 시퀀스입니다

입출력

Input: `nums = [1,2,3,4,5,2,6,10,4,12]`

Output: 8

Input: `nums = [0,1,0,3,2,3]`

Output: 4

Input: `nums = [7,7,7,7,7,7,7]`

Output: 1

제한사항

$1 \leq \text{nums.length} \leq 2500$

$-10^4 \leq \text{nums}[i] \leq 10^4$

문제분석

0 1 2 3 4 5

nums {1, 2, 1, 4, 3, 3}

Seq {1, 2, 4, 0, 0, 0}

{1, 2, 3, 0, 0, 0}

1. 배열을 하나 더 만든다(seq[])
2. 연속적으로 증가되는 부분만 선택해서 저장
3. $1 < 2$, $2 < 4$
4. $2 > 1$, 되는 시점은 $seq[0] = nums[2]$

nums	1	2	1	4	3	3
DP	1	0	0	0	0	0

1. 배열을 하나 더 만든다(dp[])

문제분석



nums	1	2	1	4	3	3
DP	1	1	0	0	0	0
DP	1	2	0	0	0	0

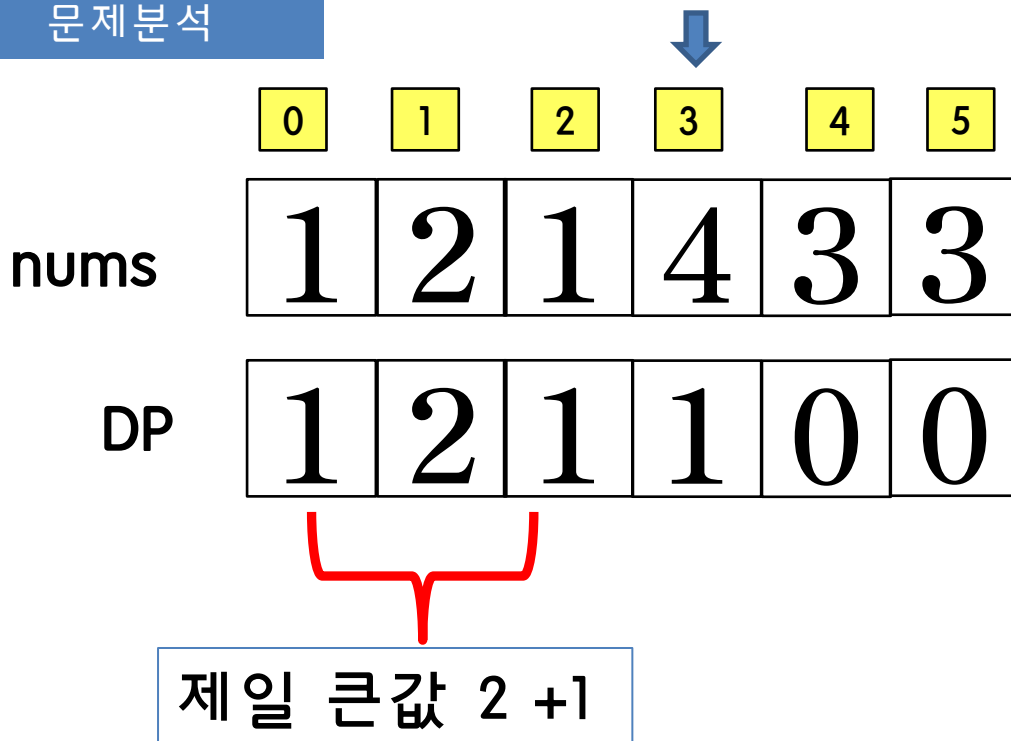
1. 배열을 하나 더 만든다(dp[])
2. 연속적으로 증가되는 부분만 +1 후 저장

문제분석

	0	1	2	3	4	5
nums	1	2	1	4	3	3
DP	1	2	1	0	0	0

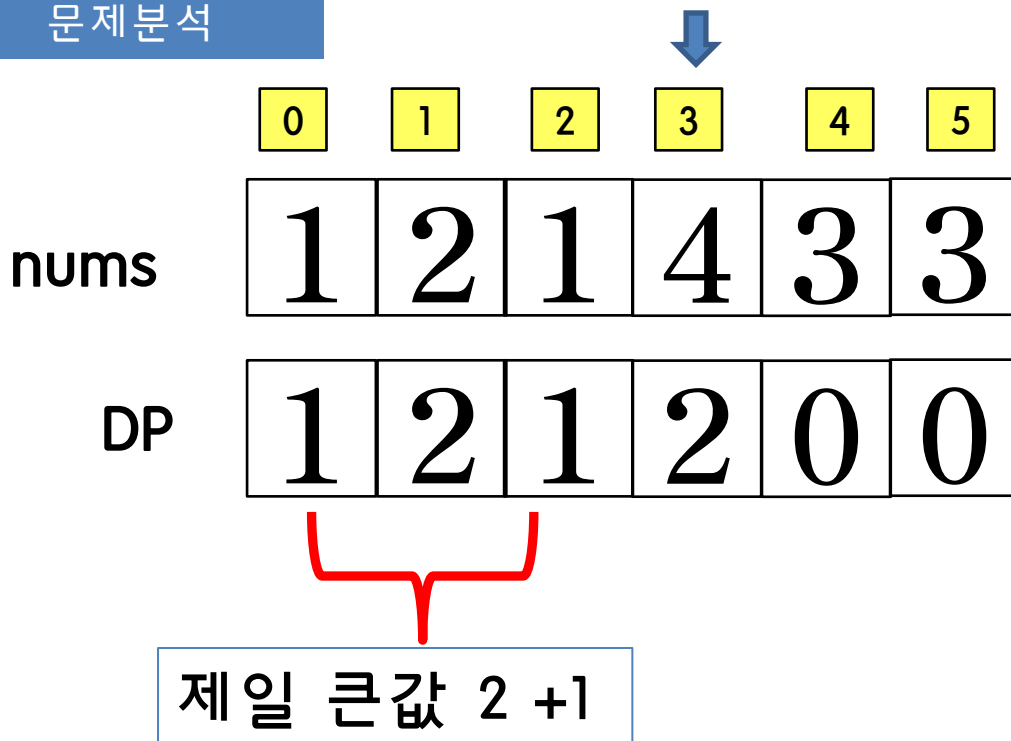
1. 배열을 하나 더 만든다(dp[])
2. 연속적으로 증가되는 부분만 +1 후 저장
3. Max값을 따로 저장하면서 비교

문제분석



1. 배열을 하나 더 만든다(dp[])
2. 연속적으로 증가되는 부분만 +1 후 저장
3. Max값을 따로 저장하면서 비교
4. `nums[3]==4`이므로 앞에서 제일 큰 dp값 2+1

문제분석




1. 배열을 하나 더 만든다(dp[])
2. 연속적으로 증가되는 부분만 +1 후 저장
3. Max값을 따로 저장하면서 비교
4. `nums[3]==4`이므로 앞에서 제일 큰 dp값 2+1

문제분석

	0	1	2	3	4	5
nums	1	2	1	4	3	3
DP	1	2	1	3	0	0

1. 배열을 하나 더 만든다(dp[])
2. 연속적으로 증가되는 부분만 +1 후 저장
3. Max값을 따로 저장하면서 비교
4. `nums[3]==4`이므로 앞에서 제일 큰 dp값 2+1

문제분석



	0	1	2	3	4	5
nums	1	2	1	4	3	3
DP	1	2	1	3	3	3

1. 배열을 하나 더 만든다(dp[])
2. 연속적으로 증가되는 부분만 +1 후 저장
3. Max값을 따로 저장하면서 비교
4. `nums[3]==4`이므로 앞에서 제일 큰 dp값 2+1

시간복잡도/공간복잡도 계산

시간복잡도

1. 대상(Source) : 문제에서 입력받은 파라미터(array 등) (속도)
Time Complexity : $O(N^2)$
대상 : `int[] nums`
이유 : for문 두번 실행

공간복잡도

2. 대상(Source) : 실제 사용되는 저장 공간을 계산(메모리 사용량)
예) 프로그램을 실행 및 완료하는데 필요한 저장공간
Space Complexity : $O(N)$
대상 : `int[] dp= new int[n];`
이유 : 추가적인 공간을 사용.

참고

- $O(1)$: 스택, 큐, Map
- $O(n)$: for문 => 데이터를 한번씩 다 호출하니까 (제일 많음)
- $O(\log N)$: sort, priorityQueue, binary Search Tree, Tree
- $O(k \log N)$: k번만큼 소팅하는 경우
- $O(n^2)$: 이중for문
- $O(m*n)$: 이중for문인데, n이 다른 경우 bfs, dfs 류 (예 n=100 인데 m=5인 경우)

동적 프로그래밍은 하나 이상의 **시퀀스** (예 : 배열, 행렬) 에 대한 **최적화** 방법입니다 .

동적 프로그래밍 판단 방법:

시퀀스 에 대한 작업을 수행하여 얻을 수 있는

최대 / 가장 긴 , 최소 / 가장 짧은 값 / 비용 / 이익을 요구 합니다 .

구현 방법

Top-down dp[] DFS+Memoization

Bottom up dp[] 0번부터 채우는 방식

1

dp 테이블 생성

```
int[][] dp = new int[rows + 1][cols + 1];
```

2

맞는 조건을 찾아내는 부분

```
for (int i = 1; i <= rows; i++) {  
    for (int j = 1; j <= cols; j++) {  
        if (matrix[i - 1][j - 1] == '1') {  
            점화식 찾기  
        }  
    }  
}
```

3

점화식 생성

```
for (int i = 1; i <= rows; i++) {  
    for (int j = 1; j <= cols; j++) {  
        if (matrix[i - 1][j - 1] == '1') {  
            점화식 찾기  
            dp[i][j] = Math.min(  
                Math.min(dp[i][j - 1], dp[i - 1][j]), dp[i - 1][j - 1]) + 1;  
            maxsqlen = Math.max(maxsqlen, dp[i][j]);  
        }  
    }  
}
```

Dp테이블의 row, col 를 직접 손으로 그린다.

항목명과 내가 생각한 규칙(subProgram)이 통하는지 구체적인 예를들어 적는다.
추상적으로 생각하면 안되고 꼭 구체적으로 작성해 본다.