Dynamic Programming

Dynamic programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems.

Linear (Single Sequence) DP
Double Sequence DP
Matrix (Coordination) DP
Knapsack DP
Game Theory DP
Interval DP

Linear (Single Sequence) DP

Give you a single sequence, you can select or skip an element in the sequence.

Number sequence 1234567
String ’abcdefg....’
Climb Stairs ....
A line of fences or houses to visit or paint....
Time or date in sequence....

DP 4 steps

1 Status

Each element in dp array should define a status. ie max steps, max sum etc. Some time we need use 2 dimension array.

2 transition equation (or choice)

From one status to next status. left-> right, right ->left? Or move from A diagonal line (i+1, j) (i, j+1) ...

3 start point

Init dp array, from (0) or (1) or (n) or (0, 0) as start point, or init line one, column one

4 end point

Final answer, max value etc, can be (0), or (n) or (m - 1, n - 1) or (0, n)...

70. Climbing Stairs

Easy 5598 178 Add to List Share

You are climbing a staircase. It takes `n` steps to reach the top.

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

Example 1:

```
Input: n = 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
```

Example 2:

```
Input: n = 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step
```

Let $dp[i]$ denotes the number of ways to reach on i^{th} step:

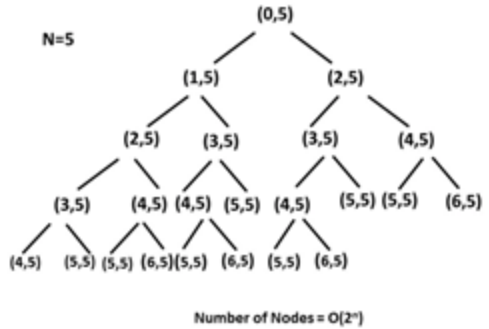
$$dp[i] = dp[i-1] + dp[i-2]$$

N=6

DP

0	1	2	3	5	8	13
0	1	2	3	4	5	6

```java
//recursive
public int climbStairs(int n) {
    return helper(n, new int[n + 1]);
}

public int helper(int n, int[] memo) {
    if (n <= 2) return n;
    if (memo[n] > 0) return memo[n];
    return memo[n] = helper(n - 1, memo) + helper(n - 2, memo);
}

//iterative
public int climbStairs(int n) {
    if (n < 2) return n;
    int[] dp = new int[n + 1];
    dp[1] = 1;
    dp[2] = 2;
    for (int i = 3; i <= n; i++)
        dp[i] = dp[i - 1] + dp[i - 2];
    return dp[n];
}

//iterative space O(1)
public int climbStairs(int n) {
    if (n <= 2) return n;
    int prev1 = 1, prev2 = 1, res = 0;
    for (int i = 2; i <= n; i++) {
        res = prev1 + prev2;
        prev1 = prev2;
        prev2 = res;
    }
    return res;
}
```



198. House Robber

Medium 6276 183 Add to List Share

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**.

Example 1:

```
Input: nums = [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
             Total amount you can rob = 1 + 3 = 4.
```

Example 2:

```
Input: nums = [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).
             Total amount you can rob = 2 + 9 + 1 = 12.
```

```java
class Solution {
    public int rob(int[] nums) {
        if(nums.length==0) return 0;
        if(nums.length==1) return nums[0];
        int dp[] = new int[nums.length];
        dp[0] = nums[0];
        dp[1] = Math.max(nums[0], nums[1]);
        for(int i = 2;i<nums.length; ++i){
            dp[i] = Math.max(nums[i] + dp[i-2], dp[i-1]);
        }
        return dp[nums.length-1];
    }
}
```

```java
Integer[] memo;
public int rob(int[] nums) {
    memo = new Integer[nums.length + 1];
    return rob(nums, nums.length - 1);
}

private int rob(int[] nums, int i) {
    if (i < 0) return 0;
    if (memo[i] != null) return memo[i];
    int res = Math.max(rob(nums, i - 1), rob(nums, i - 2) + nums[i]);
    return memo[i] = res;
}
```

Share

rdDict containing a list of **non-empty**
pace-separated sequence of one or

used multiple times in the

ontain duplicate words.

t", "code"]

ode" can be segmented as "leet

["apple", "pen"]

penapple" can be segmented as

to reuse a dictionary word.

ts", "dog", "sand", "and", "cat"]

o nested loops, and substring computation at
(n^3) time complexity.

array is $n + 1$.

0	1	2	3	4	5	6	7	8	9
*	L	e	e	t	c	o	d	e	*
T	F	F	F	F	T	F	F	F	T

```java
class Solution {
    //dp[i] = dp[j] && dict.contains(s.substring(j,i)) j=0..i-1
    public boolean wordBreak(String s, List<String> wordDict) {
        int N= s.length();
        boolean[] dp = new boolean[N+1];
        dp[0] = true;
        Set<String> set= new HashSet<String>(wordDict);
        for(int i = 1;i<=N; i++){
            for(int j= 0;j<i; j++){
                if(dp[j] && set.contains(s.substring(j,i))){
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[N];
    }
}
```

300. Longest Increasing Subsequence

Medium 6169 141 Add to List Share

Given an integer array `nums`, return the length of the longest strictly increasing subsequence.

A **subsequence** is a sequence that can be derived from an array by deleting some or no elements without changing the order of the remaining elements. For example, `[3,6,2,7]` is a subsequence of the array `[0,3,1,6,2,2,7]`.

Example 1:

```
Input: nums = [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is
[2,3,7,101], therefore the length is 4.
```

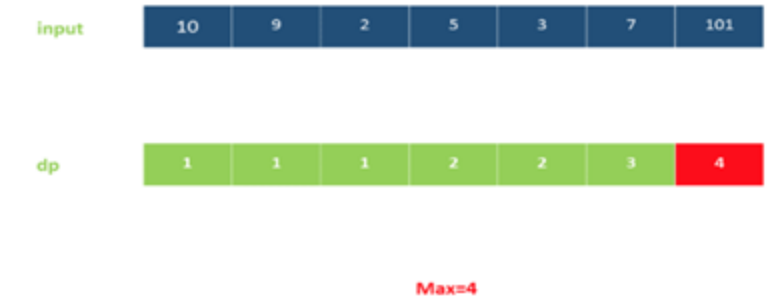
Example 2:

```
Input: nums = [0,1,0,3,2,3]
Output: 4
```

Example 3:

```
Input: nums = [7,7,7,7,7,7,7]
Output: 1
```

```java
class Solution {
    // dp[i] longest increasing subsequence from 0 to i,
    public int lengthOfLIS(int[] nums) {
        if(nums.length == 0) return 0;
        int[] dp=new int[nums.length];
        Arrays.fill(dp,1);
        int res = 1;
        for(int i = 1;i<nums.length; ++i){
            for(int j = 0;j<i; j++){
                if(nums[j] < nums[i]){
                    dp[i] = Math.max(dp[j] + 1, dp[i]);
                    res = Math.max(res,dp[i]);
                }
            }
        }
        return res;
    }
}
```



Multiple (Double) Sequence DP

Two Sequences, 2 Strings, 2 arrays use 2 dimension array to save the status

- **state: dp[i][j] first sequence first i characters or numbers, matching the second sequence first j characters or numbers.**

  longest common subsequence, or minimal edit distance

1143. Longest Common Subsequence

Medium 2682 34 Add to List Share

Given two strings `text1` and `text2`, return *the length of their longest* ***common subsequence***. If there is no **common subsequence**, return `0`.

A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

- For example, `"ace"` is a subsequence of `"abcde"`.

A **common subsequence** of two strings is a subsequence that is common to both strings.

Example 1:

```
Input: text1 = "abcde", text2 = "ace"
Output: 3
Explanation: The longest common subsequence is "ace" and its length is 3.
```

Example 2:

```
Input: text1 = "abc", text2 = "abc"
Output: 3
Explanation: The longest common subsequence is "abc" and its length is 3.
```

Example 3:

```
Input: text1 = "abc", text2 = "def"
Output: 0
Explanation: There is no such common subsequence, so the result is 0.
```

```java
class Solution {
    // f[i][j] is A(0..i-1) and B(0..j-1) longest common subsequence, we need find f[N1][N2]
    // f[i][j] = f[i-1][j-1] + 1| if A[i-1]= B[j-1]
 Otherwise  = max(f[i-1][j], f[i][j-1])
    public int longestCommonSubsequence(String text1, String text2) {
        int N1 = text1.length();
        int N2= text2.length();
        int[][] dp = new int [N1+1][N2+1];
        for(int i = 1;i<=N1; ++i){
            for(int j = 1;j<=N2; ++j){
                if(text1.charAt(i-1)==text2.charAt(j-1))
                    dp[i][j] = dp[i-1][j-1] + 1;
                else
                    dp[i][j] = Math.max(dp[i][j-1], dp[i-1][j]);
            }
        }
        return dp[N1][N2];
    }
}
```

这个版本的做法非常适合存储和输出 optimal path，也可以应用到 Longest Increasing Subsequence 中

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Homework

97. Interleaving String
72. Edit Distance
115. Distinct Subsequences
10. Regular Expression Matching
44. Wildcard Matching
1312. Minimum Insertion Steps to Make a String Palindrome
516. Longest Palindromic Subsequence
1216. Valid Palindrome III

Matrix (Coordination) DP

62. Unique Paths
63. Unique Paths II
120. Triangle
64. Minimum Path Sum
931. Minimum Falling Path Sum
1289. Minimum Falling Path Sum II
221. Maximal Square
85. Maximal Rectangle

63. Unique Paths II

Medium 2293 269 Add to List Share

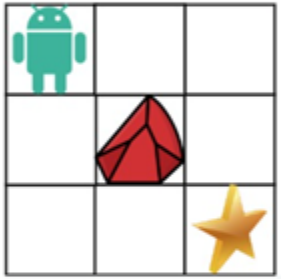
A robot is located at the top-left corner of a `m x n` grid (marked 'Start' in the diagram below).

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).

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and space is marked as `1` and `0` respectively in the grid.

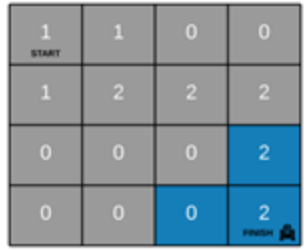
Example 1:



```
Input: obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]
Output: 2
Explanation: There is one obstacle in the middle of the 3x3 grid above.
There are two ways to reach the bottom-right corner:
1. Right -> Right -> Down -> Down
2. Down -> Down -> Right -> Right
```

Example 2:

```
Input: obstacleGrid = [[0,1],[0,0]]
Output: 1
```




```java
//recursive
public int uniquePathsWithObstacles(int[][] obstacleGrid) {
    int M = obstacleGrid.length, N = obstacleGrid[0].length;
    int[][] memo = new int[M][N];
    return helper(obstacleGrid, M - 1, N - 1, memo);
}

public int helper(int[][] grid, int i, int j, int[][] memo) {
    if (i == 0 && j == 0) return grid[i][j] == 1 ? 0 : 1;
    if (i < 0 || j < 0 || grid[i][j] == 1) return 0;
    if (memo[i][j] > 0) return memo[i][j];
    return memo[i][j] = helper(grid, i - 1, j, memo) + helper(grid, i, j - 1, memo);
}

//iterative
public int uniquePathsWithObstacles(int[][] obstacleGrid) {
    int M = obstacleGrid.length, N = obstacleGrid[0].length;
    int[][] dp = new int[M][N];
    for (int i = 0; i < M; i++) {
        if (obstacleGrid[i][0] == 1) break;
        dp[i][0] = 1;
    }
    for (int i = 0; i < N; i++) {
        if (obstacleGrid[0][i] == 1) break;
        dp[0][i] = 1;
    }
    for (int i = 1; i < M; i++)
        for (int j = 1; j < N; j++)
            dp[i][j] = obstacleGrid[i][j] != 1 ? dp[i - 1][j] + dp[i][j - 1] : 0;
    return dp[M - 1][N - 1];
}
}
```

120. Triangle

Medium 2503 285 Add to List Share

Given a `triangle` array, return *the minimum path sum from top to bottom.*

For each step, you may move to an adjacent number on the row below.

Example 1:

```
Input: triangle = [[2],[3,4],[6,5,7],[4,1,8,3]]
Output: 11
Explanation: The minimum path sum from top to bottom is 11
(i.e., 2 + 3 + 5 + 1 = 11).
```

Example 2:

```
Input: triangle = [[-10]]
Output: -10
```

Constraints:

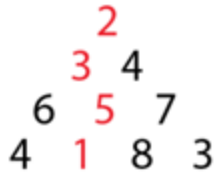
- `1 <= triangle.length <= 200`
- `triangle[0].length == 1`
- `triangle[i].length == triangle[i - 1].length + 1`
- `-10^4 <= triangle[i][j] <= 10^4`

Follow up: Could you do this using only `O(n)` extra space, where `n` is the total number of rows in the triangle?

```java
public class Solution {
    //dp 2d
    public int minimumTotal(List<List<Integer>> triangle) {
        int N = triangle.size();
        int[][] dp = new int[N + 1][N + 1];
        for (int i = N - 1; i >= 0; i--)
            for (int j = i; j >= 0; j--)
                dp[i][j] = Math.min(dp[i + 1][j], dp[i + 1][j + 1]) + triangle.get(i).get(j);
        return dp[0][0];
    }

    //dp 1d
    public int minimumTotal(List<List<Integer>> triangle) {
        int N = triangle.size();
        int[] dp = new int[N + 1];
        for (int i = N - 1; i >= 0; i--)
            for (int j = 0; j < triangle.get(i).size(); j++)
                dp[j] = Math.min(dp[j], dp[j + 1]) + triangle.get(i).get(j);
        return dp[0];
    }

    //recursive 2d
    public int minimumTotal(List<List<Integer>> triangle) {
        int N = triangle.size();
        return helper(0, 0, triangle, new int[N][N]);
    }

    private int helper(int row, int col, List<List<Integer>> triangle, int[][] memo) {
        if (row >= triangle.size()) return 0;
        if (memo[row][col] != 0) return memo[row][col];
        return memo[row][col] = Math.min(helper(row + 1, col, triangle, memo),
                                         helper(row + 1, col + 1, triangle, memo))
                                + triangle.get(row).get(col);
    }
}
```

no greedy

In row 2, number 4 is chosen over 3



The minimum sum path is 2+3+5+1=11

m/n	0	1	2	3
0	11	-	-	-
1	9	10	-	-
2	7	6	10	-
3	4	1	8	3

Filled DP table

64. Minimum Path Sum

$$\text{cost}(i, j) = \text{grid}[i][j] + \min\left(\text{cost}(i+1, j), \text{cost}(i, j+1)\right)$$

Medium 3995 77 Add to List Share

Given a `m x n grid` filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

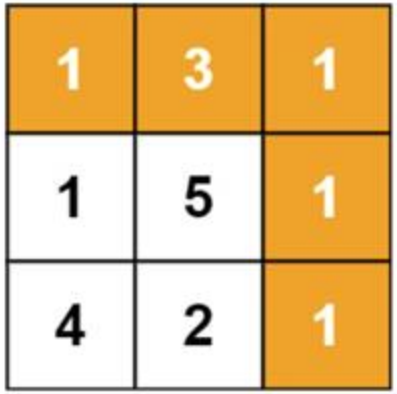
Original Array

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

dp

14	13	12	24
13	10	8	16
15	13	6	12
10	8	5	3

Example 1:




```
Input: grid = [[1,3,1],[1,5,1],[4,2,1]]
Output: 7
Explanation: Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.
```

Example 2:

```
Input: grid = [[1,2,3],[4,5,6]]
Output: 12
```

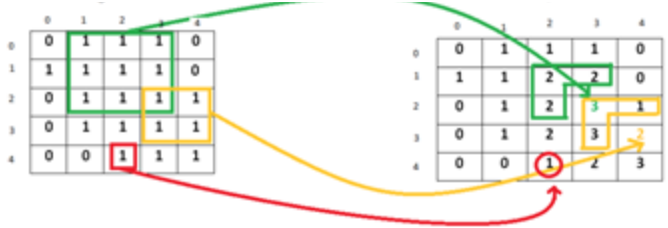
```java
//recursive
public int minPathSum(int[][] grid) {
    int M = grid.length, N = grid[0].length;
    int[][] memo = new int[M][N];
    return helper(grid, M - 1, N - 1, memo);
}

public int helper(int[][] grid, int i, int j, int[][] memo) {
    if (i == 0 && j == 0) return grid[0][0];
    if (i < 0 || j < 0) return Integer.MAX_VALUE;
    if (memo[i][j] > 0) return memo[i][j];
    return memo[i][j] = grid[i][j] + Math.min(helper(grid, i - 1, j, memo),
                                              helper(grid, i, j - 1, memo));
}

//iterative
public int minPathSum(int[][] grid) {
    int M = grid.length, N = grid[0].length;
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++) {
            if (i == 0 && j == 0) continue;
            if (i == 0) grid[i][j] += grid[i][j - 1];
            else if(j == 0) grid[i][j] += grid[i - 1][j];
            else grid[i][j] += Math.min(grid[i - 1][j], grid[i][j - 1]);
        }
    return grid[M - 1][N - 1];
}
```

221. Maximal Square

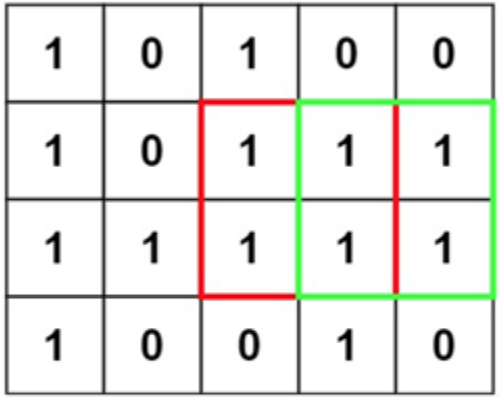
Medium 3921 97 Add to List Share

$$\mathrm{dp}(i,j) = \min\big(\mathrm{dp}(i-1,j), \mathrm{dp}(i-1,j-1), \mathrm{dp}(i,j-1)\big) + 1.$$

Given an `m x n` binary `matrix` filled with `0`'s and `1`'s, *find the largest square containing only* `1`'s *and return its area.*



Example 1:



Input: matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]
Output: 4

```java
public int maximalSquare(char[][] matrix) {
    int M = matrix.length, N = matrix[0].length;
    int[][] dp = new int[M][N];
    int max = 0;
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++) {
            if (i == 0 || j == 0)  {
                if (matrix[i][j] == '1') dp[i][j] = 1;
            } else {
                if (matrix[i][j] == '1')
                    dp[i][j]  = Math.min(Math.min(dp[i][j - 1],
                                                  dp[i - 1][j]),
                                                  dp[i - 1][j - 1]) + 1;
                }
                max = Math.max(dp[i][j], max);
            }
        return max*max;
    }
```

Example 2:

Input: matrix = [["0","1"],["1","0"]]
Output: 1

```java
public int maximalSquare(char[][] matrix) {
    int max = 0;
    for (int i = 0; i < matrix.length; i++)
        for (int j = 0; j < matrix[0].length; j++)
            if (matrix[i][j]=='1') {
                if (i > 0 && j > 0 && matrix[i - 1][j] > '0' &&
                    matrix[i][j - 1] > '0' && matrix[i - 1][j - 1] > '0') {
                    matrix[i][j]=(char)(Math.min(Math.min(matrix[i - 1][j],
                                                          matrix[i][j - 1]),
                                                          matrix[i - 1][j - 1]) + 1);
                }
                max = Math.max(matrix[i][j], max);
            }
    return max == 0 ? 0 : (int)Math.pow(max - '0', 2);
}
```

292. Nim Game

Description | Hints | Submissions | Discuss | Solution

Pick One

You are playing the following Nim Game with your friend: There is a heap of stones on the table, each time one of you take turns to remove 1 to 3 stones. The one who removes the last stone will be the winner. You will take the first turn to remove the stones.

Both of you are very clever and have optimal strategies for the game. Write a function to determine whether you can win the game given the number of stones in the heap.

Example:

```
Input: 4
Output: false
Explanation: If there are 4 stones in the heap, then you will never win the game;
             No matter 1, 2, or 3 stones you remove, the last stone will always be
             removed by your friend.
```

If dp[i] want to win the game, it must guarantee at least one of dp[i -1] dp[i -2] dp[i - 3] fail,

```java
private boolean canIWin(int n) {
    Boolean[] memo = new Boolean[n + 1];
    return dfs(n, memo);
}

private boolean dfs(int n, Boolean[] memo) {
    if (n < 0) return false;
    if (memo[n] != null) return memo[n];
    boolean res = false;
    for (int i = 1; i < 4; i++)
        if (n >= i) res |= !dfs( n: n - i, memo);
    return memo[n] = res;
}
```

```java
public boolean canWinNim(int n) {
    boolean[] dp = new boolean[Math.max(n + 1, 4)];
    dp[1] = dp[2] = dp[3] = true;
    for (int i = 4; i <= n; i++)
        dp[i] = !dp[i-1] || !dp[i-2] || !dp[i-3];
    return dp[n];
}


public boolean canWinNim(int n) {
    return n % 4 != 0;
}
```

486. Predict the Winner

Description | Hints | Submissions | Discuss | Solution

Pick One

Given an array of scores that are non-negative integers. Player 1 picks one of the numbers from either end of the array followed by the player 2 and then player 1 and so on. Each time a player picks a number, that number will not be available for the next player. This continues until all the scores have been chosen. The player with the maximum score wins.

Given an array of scores, predict whether player 1 is the winner. You can assume each player plays to maximize his score.

Example 1:

```
Input: [1, 5, 2]
Output: False
Explanation: Initially, player 1 can choose between 1 and 2.
If he chooses 2 (or 1), then player 2 can choose from 1 (or 2) and 5. If player 2 chooses 5, then p
So, final score of player 1 is 1 + 2 = 3, and player 2 is 5.
Hence, player 1 will never be the winner and you need to return False.
```

Example 2:

```
Input: [1, 5, 233, 7]
Output: True
Explanation: Player 1 first chooses 1. Then player 2 have to choose between 5 and 7. No matter whic
Finally, player 1 has more score (234) than player 2 (12), so you need to return True representing
```

Note:

1. 1 <= length of the array <= 20.
2. Any scores in the given array are non-negative integers and will not exceed 10,000,000.
3. If the scores of both players are equal, then player 1 is still the winner.

You have two choice, choose front or choose back.

dfs(i + 1) or dfs(j - 1) is for rest of the cards your opponent max score.

You can use your score - your opponent's score >= 0 to see if you can win

```
7   public boolean PredictTheWinner(int[] piles) {
8       Integer[][] memo = new Integer[piles.length][piles.length];
9       return dfs(piles, 0, piles.length - 1, memo) >= 0;
10  }
11
12  private int dfs(int[] piles, int i, int j, Integer[][] memo) {
13      if (i > j)  return 0;
14      if (memo[i][j] != null) return memo[i][j];
15      memo[i][j] = Math.max(piles[i] - dfs(piles, i + 1, j, memo),
16                            piles[j] - dfs(piles, i, j - 1, memo));
17      return memo[i][j];
18  }
```

```java
public boolean PredictTheWinner(int[] nums) {
    int n = nums.length;
    int[][] dp = new int[n][n];
    for (int i = 0; i < n; i++) { dp[i][i] = nums[i]; }
    for (int len = 1; len < n; len++) {
        for (int i = 0; i < n - len; i++) {
            int j = i + len;
            dp[i][j] = Math.max(nums[i] - dp[i + 1][j], nums[j] - dp[i][j - 1]);
        }
    }
    return dp[0][n - 1] >= 0;
}
```

Game Theory DP Homework

Leetcode 1025. Divisor Game
Leetcode 877. Stone Game
Leetcode 1140. Stone Game II
Leetcode 1406. Stone Game III
Leetcode 1510. Stone Game IV

Interval DP

DP on interval is an extension of linear dynamic programming. When dividing the problem in stages, it has a lot to do with the order in which the elements appear in the stage and which elements from the previous stage are merged.
Features of interval DP:
Merger : that is to integrate two or more parts, of course, it can also be reversed;
Features : able to decompose the problem into a form that can be combined in pairs;
Solution : Set the optimal value for the entire problem, enumerate the merge points, decompose the problem into two parts, and finally merge the optimal values of the two parts to get the optimal value of the original problem.
$dp(i,j) = \max(dp(i,j), dp(i,k) + dp(k+1,j) + cost)$

516. Longest Palindromic Subsequence

Medium 3159 220 Add to List Share

Given a string `s`, find *the longest palindromic* ***subsequence****'s length in* `s`.

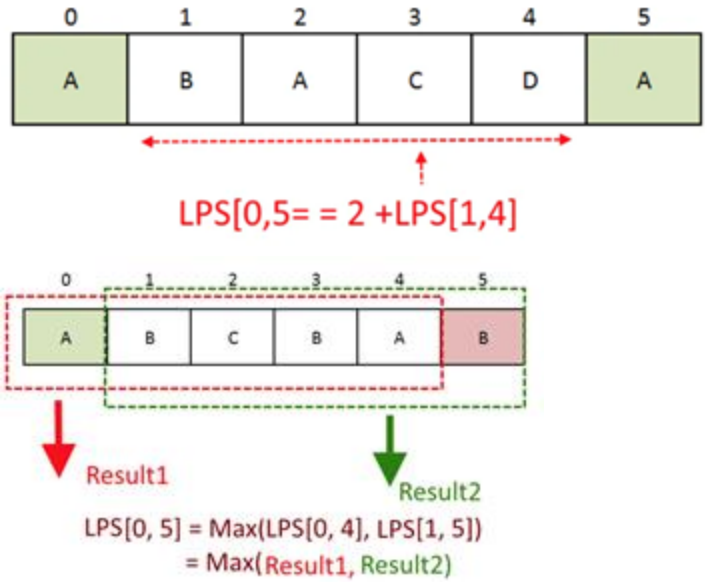
A **subsequence** is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

Example 1:

```
Input: s = "bbbab"
Output: 4
Explanation: One possible longest palindromic subsequence is "bbbb".
```

Example 2:

```
Input: s = "cbbd"
Output: 2
Explanation: One possible longest palindromic subsequence is "bb".
```




```java
Integer[][] memo;
public int longestPalindromeSubseq(String s) {
    int N = s.length();
    this.memo = new Integer[N][N];
    return dfs(s, 0, N - 1);
}

private int dfs(String s, int i, int j) {
    if (i > j) return 0;
    if (i == j) return 1;
    if (memo[i][j] != null) return memo[i][j];
    if (s.charAt(i) == s.charAt(j)) memo[i][j] = dfs(s, i + 1, j - 1) + 2;
    else memo[i][j] = Math.max(dfs(s, i + 1, j), dfs(s, i, j - 1));
    return memo[i][j];
}
```

Substring length = 6 (abacba)

	a	b	a	c	b	a
a	1	1	3	3	3	5
b		1	1	1	3	3
a			1	1	1	3
c				1	1	1
b					1	1
a						1

```
2   public int longestPalindromeSubseq(String s) {
3       int N = s.length();
4       int[][] dp = new int[N][N];
5       for (int i = N - 1; i >= 0; i--) {
6           dp[i][i] = 1;
7           for (int j = i + 1; j < N; j++) {
8               if (s.charAt(i) == s.charAt(j)) dp[i][j] = dp[i + 1][j - 1] + 2;
9               else dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
10          }
11      }
12      return dp[0][N - 1];
13  }
```

312. Burst Balloons

Hard 3615 109 Add to List Share

You are given `n` balloons, indexed from `0` to `n - 1`. Each balloon is painted with a number on it represented by an array `nums`. You are asked to burst all the balloons.

If you burst the `i^th` balloon, you will get `nums[i - 1] * nums[i] * nums[i + 1]` coins. If `i - 1` or `i + 1` goes out of bounds of the array, then treat it as if there is a balloon with a `1` painted on it.

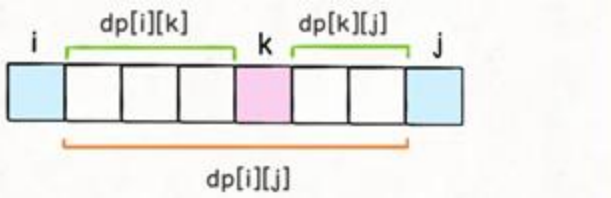
Return *the maximum coins you can collect by bursting the balloons wisely*.

Example 1:

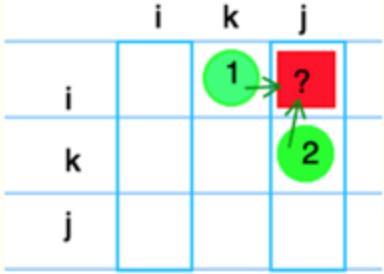
```
Input: nums = [3,1,5,8]
Output: 167
Explanation:
nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
coins =  3*1*5    +   3*5*8   +  1*3*8  + 1*8*1 = 167
```

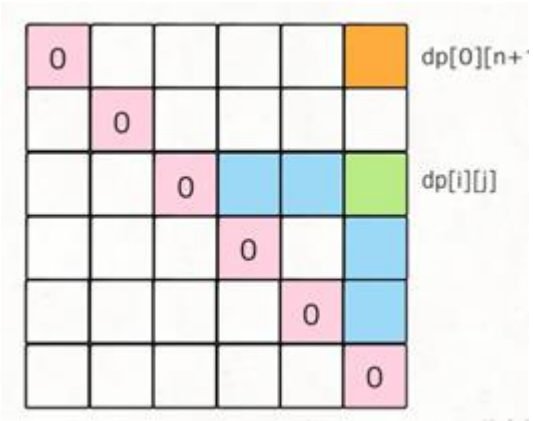
Example 2:

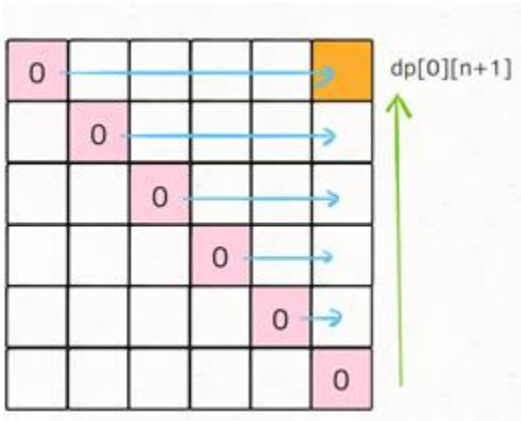
```
Input: nums = [1,5]
Output: 10
```



To caluate dp[I][j], we should calculate dp[I][k] and dp[k][j] first.







DFS + Memo

```
3   public int maxCoins(int[] nums) {
4       int N = nums.length;
5       Integer[][] memo = new Integer[N][N];
6       return dfs(nums, 0, N - 1, memo);
7   }
8   public int dfs(int[] nums, int start, int end, Integer[][] memo) {
9       if (start > end) return 0;
10      if (memo[start][end] != null) return memo[start][end];
11      int max = Integer.MIN_VALUE;
12      for (int i = start; i <= end; i++) {
13          int left = dfs(nums, start, i - 1, memo);
14          int right = dfs(nums, i + 1, end, memo);
15          int cur = get(nums, i) * get(nums, start - 1) * get(nums, end + 1);
16          max = Math.max(max, left + right + cur);
17      }
18      return memo[start][end] = max;
19  }
20  public int get(int[] nums, int i) {
21      if (i == -1 || i == nums.length) return 1;
22      return nums[i];
23  }
```

bottom up iterative DP

```
27    public int maxCoins(int[] nums) {
28        int[] ballons = new int[nums.length + 2];
29        int N = ballons.length;
30        ballons[0] = ballons[N - 1] = 1;
31        for (int i = 1; i < N - 1; i++) ballons[i] = nums[i - 1];
32        int[][] dp = new int[N][N];
33        for (int i = N - 2; i >= 0; i--) {
34            for (int j = i + 2; j < N; j++)
35                for (int k = i + 1; k < j; k++)
36                    dp[i][j] = Math.max(dp[i][j],
37                                        ballons[i] * ballons[k] * ballons[j] + dp[i][k] + dp[k][j]);
38        return dp[0][N - 1];
39    }
40    //  0 1 2 3  4   5
41    //  1 3 1 5  8   1
42  //0 1     3 30 159 167   (i=0, j=2, k=1, 1*3*1=3) (i=0, j=3, k=1, 1*3*5+15=30) (i=0, j=3, k=2, 1*1*5+3=8)     #1
43  //1 3       15 135 159   (i=1, j=3, k=2, 3*1*5=15) (i=1, j=4, k=2, 3*1*8+40=64) (i=1, j=4, k=3, 3*5*8+15=135) #2
44  //2 1          40  45    (i=2, j=4, k=3, 1*5*8+0+0=40) (i=2, j=5, k=3, 1*5*1+40+0=45) (i=2, j=5, k=4, 1*8*1+40+0=45)
45  //3 5              40    (i=3, j=5, k=4, 5*8*1+0+0=40)
46  //4 8
47  //5 1
48
49  //#1                (i=0, j=4, k=1, 1*3*8+135=159) (i=0, j=4, k=2, 1*1*8+3+40=51)(i=0, j=4, k=3, 1*5*8+30=70)
50  //                  (i=0, j=5, k=1, 1*3*1+159=162) (i=0, j=5, k=2, 1*1*1+3+45=49)(i=0, j=5, k=3, 1*5*1+30+40=75)
51  //                  (i=0, j=5, k=4, 1*8*1+159=167)
52  //#2                (i=1, j=5, k=2, 3*1*1+45=48) (i=1, j=5, k=3, 3*5*1+15+40=70) (i=1, j=5, k=4, 3*8*1+135=159)
53  }
```

Interval DP Homework

516. Longest Palindromic Subsequence
1216. Valid Palindrome III
1039. Minimum Score Triangulation of Polygon
1547. Minimum Cost to Cut a Stick

你们的好朋友Eddie

Knapsack DP

416. Partition Equal Subset Sum
474. Ones and Zeroes
494. Target Sum
1049. Last Stone Weight II
322. Coin Change
518. Coin Change 2
798. Backpack VII (LintCode)