Chapter 7 Dynamic Programming

DP问题特征：
最优解包含了其子问题的最优解 (optimal substructure)

- an optimal solution can be constructed from optimal solutions of its subproblems
- 在Graph中找A到B的最短路径，如果该路径经过C，则其C到B的路径肯定就是C到B的最短路径

子问题重叠性质 (overlapping subproblems)

- a problem is said to have **overlapping subproblems** if the problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems
- 某些子问题会被反复计算多次

解题二步曲：

- 定义子问题和定义递推关系（recurrence relation）
- 找到base case

fibonacci sequence
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

F(n) = F(n-1) + F(n-2)

```
F(50) = F(49) + F(48)
F(49) = F(48) + F(47)
F(48) = F(47) + F(46)
...
```

Top down(dfs + memoization)

```
var m = map(0 → 0, 1 → 1)
function fib(n)
    if key n is not in map m
           m[n] = fib(n - 1) + fib(n - 2)
    return m[n]
```

bottom-up

```
function fib(n)
       if n = 0
           return 0
       else
           var prevFib = 0, currFib = 1
           repeat n - 1 times
               var newFib = prevFib + currFib
```

```
                prevFib = currFib
                currFib  = newFib
        return currentFib
```

推荐阅读：
http://www.cs.cmu.edu/afs/cs/academic/class/15451-f18/www/
https://www.programiz.com/dsa/dynamic-programming

一维动态规划问题

Longest Increasing Subsequence
Given an unsorted array of integers, find the length of longest increasing subsequence.

Example:

```
Input: [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.
```

state[i]: **以第i个数结尾**的最长递增子序列（longest increasing subsequence）
state[i] = max(state[j] + 1), 其中 0 <= j < i 和 input[j] < input[i]
    对于**在i之前的元素j**，如果其值**小于第i个数的值**，则可以连接成一个**新的**最长递增子序列
    取这其中最大的，则为**以第i个数结尾**的最长递增子序列
Base case: 将所有的state初始值设为1

input = [10,9,2,5,3,7,101,18]
Init state = [1,1,1,1,1,1,1,1]

Final state = [1,1,1,2,2,3,4,4]

```
public int lengthOfLIS(int[] nums) {
 if(nums.length <= 1)
   return nums.length;

 // This will be our array to track longest sequence length
 int[] state = new int[nums.length];

 // Fill each position with value 1 in the array
 for(int i=0; i < nums.length; i++)
   state[i] = 1;

 for(int i = 1; i < nums.length; i++) {
   for(int j = 0; j < i; j++) {
     if(nums[j] < nums[i]) {
       if(state[j] + 1 > state[i]) {
         state[i] = state[j] + 1;
       }
     }
   }
 }

 int longest = 0;
 for(int i=0; i < state.length; i++)
   longest = Math.max(longest, state[i]);

 return longest;
}
```

https://leetcode.com/problems/number-of-longest-increasing-subsequence/submissions/

Coin Change
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.

Examples:

```
Input: coins = [1, 2, 5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1

Input: coins = [2], amount = 3
Output: -1
```

state[i]: amount为i时，所需的最少coins
state[i] = min(state[j] + 1)，其中j为**i - 某一个coin的值**
        对于每一个小于amount的值i，看看**i - 某一个coin的值**的state[j]
        取其中最小值为state[i]
Base case: state[coin值] = 1，其之前的值为不可取

coins = [3, 4, 7], amount: 9
Init state = [Max, Max, Max, 1, 1, Max, Max, 1, Max, Max]

Final state = [Max, Max, Max, 1, 1, Max, 2, 1, 2, 3]

```
public static int coinChange(int[] coins, int amount) {
      if (coins == null || coins.length == 0 || amount <= 0)
            return 0;

      int[] minNumber = new int[amount + 1];
```

```
        for (int i = 1; i <= amount; i++) {
            minNumber[i] = Integer.MAX_VALUE;

            for (int j = 0; j < coins.length; j++) {
                if (coins[j] <= i && minNumber[i - coins[j]] != Integer.MAX_VALUE)
                    minNumber[i] = Integer.min(minNumber[i], 1 + minNumber[i - coins[j]]);
            }
        }

    return minNumber[amount] == Integer.MAX_VALUE ? -1 : minNumber[amount];
}
```

House Robber
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.

Exampless:

```
Input: [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.

Input: [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.
```

state[i]: 对于前i个房子，抢匪以抢第i 个房子结束抢劫，所能获取的最大金额
state[i] = max(state[i - 2], state[i - 3]) + input[i]
Base case: state[0] = input[0], state[1] = input[1]

为了防止 i-3 out of index 添加 state[0] = 0

Input: [5,1,2,9,1]
Init state:
   5, 1, 2, 9, 1
0, 5, 1, 0, 0, 0

Final state:
   5, 1, 2, 9, 1
0, 5, 1, 7, 14, 8

```java
public int rob(int[] num) {
    if (num.length == 0) {
      return 0;
    }

    if (num.length == 1) {
      return num[0];
    }

    int[] state = new int[num.length + 1];
    state[0] = 0;
    state[1] = num[0];
    state[2] = num[1];

    for(int i = 2; i < num.length; i++) {
        state[i + 1] = Math.max(state[i - 1], state[i - 2]) + num[i];
    }
    return Math.max(state[num.length], state[num.length - 1]);
}
```

Delete and Earn
Given an array nums of integers, you can perform operations on the array.

In each operation, you pick any nums[i] and delete it to earn nums[i] points. After, you must delete every element equal to nums[i] - 1 or nums[i] + 1.

You start with 0 points. Return **the maximum number of points** you can earn by applying such operations.

Note:
The length of nums is at most 20000.
Each element nums[i] is an integer in the range [1, 10000].

Example:

```
Input: nums = [3, 4, 2]
Output: 6
Explanation:
Delete 4 to earn 4 points, consequently 3 is also deleted.
Then, delete 2 to earn 2 points. 6 total points are earned.

Input: nums = [2, 2, 3, 3, 3, 4]
Output: 9
Explanation:
Delete 3 to earn 3 points, deleting both 2's and the 4.
Then, delete 3 again to earn 3 points, and 3 again to earn 3 points.
9 total points are earned.
```

可以转化为House Robber题目

state[i]: pick或者delete小于等于**i** 的所有数字所产生的最大points
state[i] = max(count.get(i) + state[i - 2], state[i - 1])
其中count字典为**元素等i**的**元素和**（比如：5个3在数组中，则count.get(3) = 15）
Base case: state[0] = 0, state[1] = count[1]

Input: [2, 2, 3, 3, 3, 4]
Count: {0:0, 2: 4, 3: 9, 4: 4}
State: [0, 0, 0, 0, 0, ..., 0] 总共10001个0，因为数字取值为[1, 10000]，其中state[0]
也可以用Input最大的数字作为state的长度

State: [0, 0, 4, 9, 9, ..., 9]

部分难点在于coding

```java
public int deleteAndEarn(int[] nums) {
    // count可以用map来代替
    int[] count = new int[10001];
    for(int n : nums){
        count[n] += n;
    }

    int[] state = new int[10001];
    state[1] = count[1];

    for(int i = 2; i < 10001; i++) {
        state[i] = Math.max(count[i] + state[i - 2], state[i - 1]);
    }
    return state[10000];
}
```

Word Break
Given a non-empty string s and a dictionary wordDict containing a list of non-empty words, determine if s can be segmented into a space-separated sequence of one or more dictionary words.

Examples:

```
Input: s = "leetcode", wordDict = ["leet", "code"]
Output: true
Explanation: Return true because "leetcode" can be segmented as "leet code".

Input: s = "applepenapple", wordDict = ["apple", "pen"]
Output: true
Explanation: Return true because "applepenapple" can be segmented as "apple pen
apple". Note that you are allowed to reuse a dictionary word.
```

```
Input: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
Output: false
```

state[i]: 前i个字符串是否能有单词组成
state[i] = true 如果可以找到任意单词word[j]，满足state[i - word[j].length()] == true && substring(i - j. i + 1) is word
Base case: state[0] = true

Input: s = "leetcode", wordDict = ["leet", "code"]
Init state: [true, false, false, false, false, false, false, false, false]

Final state: [true, false, false, false, true, false, false, false, true]

```
public boolean wordBreak(String s, Set<String> dict) {
    boolean[] state = new boolean[s.length() + 1];
    state[0] = true;

    for(int i=1; i <= s.length(); i++){
        for(int j=0; j < i; j++){
            if(state[j] && dict.contains(s.substring(j, i))){
                state[i] = true;
                break;
            }
        }
    }

    return state[s.length()];
}
```

Maximum Length of Pair Chain
You are given n pairs of numbers. In every pair, the first number is always smaller than the second number.

Now, we define a pair (c, d) can follow another pair (a, b) if and only if b < c. Chain of pairs can be formed in this fashion.

Given a set of pairs, find the length longest chain which can be formed. You needn't use up all the given pairs. You can select pairs in any order.

Example:

```
Input: [[1,2], [2,3], [3,4]]
Output: 2
Explanation: The longest chain is [1,2] -> [3,4]
```

state[i]: 以index为 i 的pair结束的chain最长的长度
state[i] = max(state[j] + 1) 其中 j < i，即指代第 i 个pair之前的pair，且pairs[j][1] < pairs[i][0]

Base case: state[0] = 1

Init:
Input: [[1,2], [2,3], [3,4]]
State: [1, 0, 0]

i = 1，即以[2, 3]结尾的chain
j = 0, [1, 2] -> 因为pair[0][1] == pair[1][0]，那么pair[1]无法与pair[0]形成一个chain

i = 2，即以[3, 4]结尾的chain
j = 0, [1, 2] -> 因为pair[0][1] < pair[2][0]，那么pair[2]可以和pair[0]形成一个chain，此时state[2] = max(state[2], state[0] + 1) = max(0, 1 + 1) = 2
j = 1, [2, 3] -> 因为pair[1][1] == pair[2][0]，那么pair[2]无法与pair[1]形成一个chain

```
public int findLongestChain(int[][] pairs) {
    Arrays.sort(pairs, (a, b) -> a[0] - b[0]);
    int[] state = new int[pairs.length];
    Arrays.fill(state, 1);

    for (int j = 1; j < pairs.length; ++j) {
        for (int i = 0; i < j; ++i) {
            if (pairs[i][1] < pairs[j][0])
                state[j] = Math.max(state[j], state[i] + 1);
        }
    }
    // Jing: 可以直接返回最后一个state value
    int ans = 0;
    for (int x : state) if (x > ans) ans = x;
    return ans;
}
```

Palindrome Partitioning II
Given a string s, partition s such that every substring of the partition is a palindrome.

Return the minimum cuts needed for a palindrome partitioning of s.

Examples:
```
Input: "aab"
Output: 1
Explanation: The palindrome partitioning ["aa","b"] could be produced using 1 cut.
```

state[i]: 前i个字母形成palindrome partitions所需最小cut
state[i] = min(state[j] + 1)，其中0 <= j <= i且子字符串(j, i)是palindrome
如果子字符串(j, i)是palindrome，则考虑当前的state[j]；最后取最小的state[j]，用以推算state[i]
Base case: state[i] = i，即前i个字母至多需要i个cut去形成palindrome partitions，也就是把每一个character都划分为palindrome

Input: “aab”
state = [0, 1, 2]

‘a’
state[0,1,2]

‘aa’
state[0,0,2]

‘aab’
state[0,0,1]

如何去节省时间判断子字符串(j, i)是palindrome？
Cache

```
public int minCut(String s) {
    char[] c = s.toCharArray();
    int[] state = new int[c.length];
    boolean[][] pal = new boolean[c.length][c.length];

    for(int i = 0; i < c.length; i++) {
        int min = i;
        for(int j = 0; j <= i; j++) {
            if(c[j] == c[i] && (j + 1 > i - 1 || pal[j + 1][i - 1])) {
                pal[j][i] = true;
                min = j == 0 ? 0 : Math.min(min, cut[j - 1] + 1);
            }
        }
        state[i] = min;
    }
    return state[c.length - 1];
}
```

其它一维动态规划问题：
Longest Valid Parentheses
Climbing Stairs
Decode Ways
Unique Binary Search Trees
Unique Binary Search Trees II
Perfect Squares
Counting Bits
Can I Win
Unique Substrings in Wraparound String
Student Attendance Record II
Best Time to Buy and Sell Stock
2 Keys Keyboard

二维动态规划问题

2维矩阵类型

Minimum Path Sum
Given a m*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.

Example:
Input:

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

state[i][j]: 到达grid[i][j]所需最小path sum
state[i][j] = min(state[i][j - 1], state[i - 1][j]) + grid[i][j]
因为仅有两种到达grid[i][j]的方式，所以取其中的path sum最小的即可
Base case: state[0][0] = grid[0][0]

Input:
```
[
 [1,3,1],
 [1,5,1],
 [4,2,1]
]
```
Init State:
```
[
 [1,0,0],
 [0,0,0],
 [0,0,0]
]
-------------------------------------------------------------------
[
 [1,4,0],
 [2,0,0],
 [0,0,0]
]

[
```

 [1,**4**,5],
 [2,7,0],
 [0,0,0]
]

[
 [1,4,5],
 [**2**,7,0],
 [6,0,0]
]

```
public int minPathSum(int[][] grid) {
        int rows = grid.length;
        int cols = grid[0].length;
        for (int i = 0; i < rows; i++) {
                for (int j = 0; j < cols; j++) {
                        if (i == 0 && j != 0) {
                                grid[i][j] = grid[i][j] + grid[i][j - 1];
                        } else if (i != 0 && j == 0) {
                                grid[i][j] = grid[i][j] + grid[i - 1][j];
                        } else if (i == 0 && j == 0) {
                                grid[i][j] = grid[i][j];
                        } else {
                                grid[i][j] = Math.min(grid[i][j - 1], grid[i - 1][j]) + grid[i][j];
                        }
                }
        }

        return grid[rows - 1][cols - 1];
}
```

[Unique Paths II]
A robot is located at the top-left corner of a m*n grid. 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.

Now consider if some obstacles are added to the grids. How many unique paths would there be? An obstacle and empty space is marked as 1 and 0 respectively in the grid.

Note: m and n will be at most 100.

Example:

```
Input:
[
  [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
```

state[i][j]: 可以到达grid[i][j]的unique paths数目
state[i][j] = state[i][j - 1] + state[i - 1][j] 如果grid[i][j-1]和grid[i - 1][j]都不为1且不越界
...
Base case: state[0][0] = grid[0][0] == 0 ? 1 : 0

```java
public int uniquePathsWithObstacles(int[][] obstacleGrid) {
    int m = obstacleGrid.length;
    int n = obstacleGrid[0].length;
    int[][] state = new int[m][n];

    state[0][0] = obstacleGrid[0][0]==0 ? 1:0;
    if(state[0][0] == 0) return 0;

    for(int i=0; i<m; i++) {
        for(int j=0; j<n; j++) {
            if(obstacleGrid[i][j] == 1) {
              state[i][j] = 0;
            }else if(i == 0){
                if(j > 0) state[i][j] = state[i][j-1];
            }else if(j==0){
                if(i > 0) state[i][j] = state[i-1][j];
            }else { state[i][j] = state[i-1][j] + state[i][j-1]; }
        }
    }
    return state[m-1][n-1];
}
```

相似题目：
Dungeon Game
Triangle
Unique Paths

可转换为2维矩阵类型 I

该类问题其递推表达式只需涉及到state[i][j]附近的几个state，比如：state[i - 1][j], state[i][j - 1], state[i-1][j-1]

Edit Distance

Given two words word1 and word2, find the minimum number of operations required to convert word1 to word2.

You have the following 3 operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

Examples:

```
Input: word1 = "horse", word2 = "ros"
Output: 3
Explanation:
horse -> rorse (replace 'h' with 'r')
rorse -> rose (remove 'r')
rose -> ros (remove 'e')

Input: word1 = "intention", word2 = "execution"
Output: 5
Explanation:
intention -> inention (remove 't')
inention -> enention (replace 'i' with 'e')
enention -> exention (replace 'n' with 'x')
exention -> exection (replace 'n' with 'c')
exection -> execution (insert 'u')
```

state[i][j]: 将word1的前i个字符转化为word2的前j个字符所需最小操作

state[i][j] = state[i - 1][j - 1]   if word1[i] == word2[j]

                    1 + min(state[i][j - 1], state[i - 1][j], state[i - 1][j - 1]) if word1[i] != word2[j]

Base case: state[i][0] = i; state[0][j] = j

state[1][1] = 1 + min(state[1][0], state[0][1], state[0][0])

                    = 1 + min(1, 1, 0)

                    = 1

state[1][0]: 删除

state[0][1]: 增加

Final State:

	Empty	r	o	s
Empty	0	1	2	3
h	1	1	2	3
o	2	2	1	2
r	3	2	2	2

s	4	3	3	2
e	5	4	4	3

```
public int minDistance(String word1, String word2) {
    int m = word1.length();
    int n = word2.length();

    int[][] state = new int[m + 1][n + 1];
    for(int i = 0; i <= m; i++)
        state[i][0] = i;
    for(int i = 1; i <= n; i++)
        state[0][i] = i;

    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) {
            if(word1.charAt(i) == word2.charAt(j))
                state[i + 1][j + 1] = state[i][j];
            else {
                int a = state[i][j];
                int b = state[i][j + 1];
                int c = state[i + 1][j];
                // max(a, b, c)
                state[i + 1][j + 1] = a < b ? (a < c ? a : c) : (b < c ? b : c);
                state[i + 1][j + 1]++;
            }
        }
    }
    return state[m][n];
}
```

Interleaving String
Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2.

Examples:

```
Input: s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac"
Output: true

Input: s1 = "aabcc", s2 = "dbbca", s3 = "aadbbbaccc"
Output: false
```

state[i][j]: s3的前i+j个字符是否能够由s1的前i个字符和s2的前j个字符交织组成
state[i][j] = state[i - 1][j] && (s1[i - 1] == s3[i + j - 1]) 或者 state[i][j - 1] && (s2[j - 1] == s3[i + j - 1])
即：要么选择s3[i + j - 1]字符由s1[i - 1]组成，或由s2[j - 1]组成
Base case: state[0][0] = true

state[i][0] = state[i - 1][0] && (s1[i - 1] == s3[i - 1])
state[0][j] = state[0][j - 1] && (s2[j - 1] == s3[j - 1])

S3: “aadbbcbcac”
Final State:

	Empty	d	b	b	c	a
Empty	true	false	false	false	false	false
a	true(a)	false	false	false	false	false
a	true(aa)	true(aad)	true(aadb)	true(aadbb)	true(aadbbc)	false
b	false	false	true(aadbb)	false	true(aadbbcb)	false
c	false	false	true(aadbbc)	true(aadbbcb)	true(aadbbcbc)	true(aadbbcbca)
c	false	false	false	true(aadbbcbc)	false	true(aadbbcbcac)

```
public boolean isInterleave(String s1, String s2, String s3) {
    if ((s1.length()+s2.length()) != s3.length()) return false;

    boolean[][] matrix = new boolean[s1.length() + 1][s2.length() + 1];

    matrix[0][0] = true;

    for (int i = 1; i < matrix[0].length; i++){
        matrix[0][i] = matrix[0][i-1] && (s2.charAt(i-1) == s3.charAt(i-1));
    }

    for (int i = 1; i < matrix.length; i++){
        matrix[i][0] = matrix[i-1][0] && (s1.charAt(i-1) == s3.charAt(i-1));
    }

    for (int i = 1; i < matrix.length; i++){
        for (int j = 1; j < matrix[0].length; j++){
            matrix[i][j] = (matrix[i-1][j] && (s1.charAt(i-1) == s3.charAt(i+j-1)))
                    || (matrix[i][j-1] && (s2.charAt(j-1) == s3.charAt(i+j-1)));
        }
    }

    return matrix[s1.length()][s2.length()];
}
```

Distinct Subsequences

Given a string S and a string T, count the number of distinct subsequences of S which equals T.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters.

ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not.

Examples:

```
Input: S = "rabbbit", T = "rabbit"
Output: 3
Explanation:

As shown below, there are 3 ways you can generate "rabbit" from S.
(The caret symbol ^ means the chosen letters)

rabbbit
^^^^ ^^
rabbbit
^^ ^^^^
rabbbit
^^^ ^^^

Input: S = "babgbag", T = "bag"
Output: 5
Explanation:

As shown below, there are 5 ways you can generate "bag" from S.
(The caret symbol ^ means the chosen letters)

babgbag
^^ ^
babgbag
^^    ^
babgbag
^    ^^
babgbag
  ^  ^^
babgbag
    ^^^
```

state[i][j]: <u>S.substring(0, i) 的 **distinct subsequences** 等于 T.substring(0, j) 的个数</u>

state[i][j] = state[i - 1][j - 1] + state[i - 1][j] 如果 S[i] == T[j]

state[i - 1][j] 如果S[i] != T[j]

Ex: S="bab" T="b", state[3][1] = state[2][0] + state[2][1] = 1 + 1 = 2

"bab"-"b" "ba"-"" "ba"-"b"

拿最后一个b作为subsequence

拿第一个b作为subsequence

Base case: state[0][j] = 0; state[i][0] = 1

	Empty	b	a	g
Empty	1	0	0	0
b	1	1	0	0
a	1	1	1	0
b	1	2	1	0
g	1	2	1	1
b	1	3	1	1
a	1	3	4	1
g	1	3	4	5

```java
public int numDistinct(String s, String t) {
    int[][] state = new int[s.length()+1][t.length()+1];

    for(int i = 0; i <= s.length(); i++) {
        state[i][0] = 1;
    }

    for(int i = 0; i < s.length(); i++) {
        for(int j = 0; j < t.length(); j++) {
            if(s.charAt(i) == t.charAt(j)) {
                state[i+1][j+1] = state[i][j] + state[i][j+1];
            } else {
                state[i+1][j+1] = state[i][j+1];
            }
        }
    }

    return state[s.length()][t.length()];
}
```

Maximal Square

Given a 2D binary matrix filled with 0's and 1's, find the largest square containing only 1's and return its area.

Example:

```
Input:

1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0

Output: 4
```

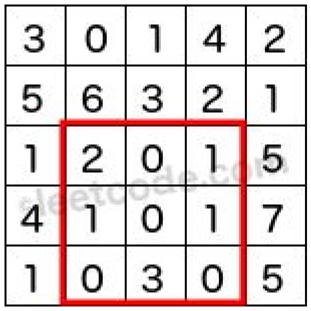
state[i][j]: 以第 i 行和第 j 列为右下角的Maximal Square边长
state[i][j] = **min**(state[i][j - 1], state[i - 1][j - 1], state[i - 1][j]) + 1 如果grid[i][j] == "1"
Base case: state[i][0] = grid[i][0] == "1" ? 1 : 0
state[0][j] = grid[0][j] == "1" ? 1 : 0

Final state:
1 0 1 0 0
1 0 1 1 1
1 1 1 2 2
1 0 0 1 0

Another input:
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 0 0 1 0
Another input state:
1 1 1 1 1
1 2 2 2 2
1 2 3 3 3
1 0 0 1 0

```
public int maximalSquare(char[][] matrix) {
    if(matrix.length == 0) return 0;
    int m = matrix.length, n = matrix[0].length, result = 0;
    int[][] state = new int[m+1][n+1];
    for (int i = 1 ; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if(matrix[i-1][j-1] == '1') {
                state[i][j] = Math.min(Math.min(state[i][j-1] , state[i-1][j-1]), state[i-1][j]) + 1;
                result = Math.max(state[i][j], result);
            }
        }
    }
    return result * result;
}
```

[Range Sum Query 2D - Immutable](#)
Given a 2D matrix matrix, find the sum of the elements inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).



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

The above rectangle (with the red border) is defined by (row1, col1) = (2, 1) and (row2, col2) = (4, 3), which contains sum = 8.

Example:

```
Given matrix = [
  [3, 0, 1, 4, 2],
  [5, 6, 3, 2, 1],
  [1, 2, 0, 1, 5],
  [4, 1, 0, 1, 7],
  [1, 0, 3, 0, 5]
]

sumRegion(2, 1, 4, 3) -> 8
sumRegion(1, 1, 2, 2) -> 11
sumRegion(1, 2, 2, 4) -> 12
```

state[i][j]: 由(0, 0)和(i, j)所限定的 rectangle 的 range sum
state[i][j] = state[i - 1][j] + state[i][j - 1] - state[i - 1][j - 1] + grid[i - 1][j - 1]
通过两个邻近矩阵来快速计算当前矩阵range sum
Base case: state[i][0] = 0; state[0][j] = 0

```
/**
 * Your NumMatrix object will be instantiated and called as such:
 * NumMatrix obj = new NumMatrix(matrix);
 * int param_1 = obj.sumRegion(row1,col1,row2,col2);
 */

private int[][] state;
```

```
public NumMatrix(int[][] matrix) {
    if(   matrix            == null
       || matrix.length     == 0
       || matrix[0].length == 0){
        return;
    }

    int m = matrix.length;
    int n = matrix[0].length;

    state = new int[m + 1][n + 1];
    for(int i = 1; i <= m; i++){
        for(int j = 1; j <= n; j++){
            state[i][j] = state[i - 1][j] + state[i][j - 1] -state[i - 1][j - 1] +
matrix[i - 1][j - 1] ;
        }
    }
}

public int sumRegion(int row1, int col1, int row2, int col2) {
    int iMin = Math.min(row1, row2);
    int iMax = Math.max(row1, row2);

    int jMin = Math.min(col1, col2);
    int jMax = Math.max(col1, col2);

    return state[iMax + 1][jMax + 1] - state[iMax + 1][jMin] - state[iMin][jMax +
1] + state[iMin][jMin];
}
```

Minimum ASCII Delete Sum for Two Strings
Given two strings s1, s2, find the lowest ASCII sum of deleted characters to make two strings equal.

Examples:

```
Input: s1 = "sea", s2 = "eat"
Output: 231
Explanation: Deleting "s" from "sea" adds the ASCII value of "s" (115) to the sum.
Deleting "t" from "eat" adds 116 to the sum. At the end, both strings are equal,
and 115 + 116 = 231 is the minimum sum possible to achieve this.

Input: s1 = "delete", s2 = "leet"
Output: 403
Explanation: Deleting "dee" from "delete" to turn the string into "let",
adds 100[d] + 101[e] + 101[e] to the sum. Deleting "e" from "leet" adds 101[e] to
the sum.
At the end, both strings are equal to "let", and the answer is 100 + 101 + 101 +
101 = 403. If instead we turned both strings into "lee" or "eet", we would get
answers of 433 or 417, which are higher.
```

state[i][j]: 让s1.substring(0, i)和s2.substring(0, j)变得一样所需最小代价
state[i][j] = state[i-1][j-1]   如果s1[i] == s2[j]
          min(state[i - 1][j] + s1[i], state[i][j - 1] + s2[j])   其它
Base case: state[0][0] = 0, state[i][0] = sum(...s1[i]), state[0][j] = sum(...s2[j])

	Empty	e	a	t
Empty	0	101	198	314
s	115	216	313	429
e	216	115	212	328
a	313	212	115	231

Delete Operation for Two Strings
Given two words word1 and word2, find the minimum number of steps required to make word1 and word2 the same, where in each step you can delete one character in either string.

Example:

```
Input: "sea", "eat"
Output: 2
Explanation: You need one step to make "sea" to "ea" and another step to make "eat"
to "ea".
```

state[i][j]: 让s1.substring(0, i)和s2.substring(0, j)变得一样所需最小代价
state[i][j] = state[i-1][j-1]   如果s1[i] == s2[j]
          min(state[i - 1][j] + 1, state[i][j - 1] + 1)   其它
Base case: state[i][0] = i, state[0][j] =j

Longest Palindromic Substring
Given a string s, find the longest palindromic substring in s. You may assume that the maximum length of s is 1000.

Example:

```
Input: "babad"
Output: "bab"
Note: "aba" is also a valid answer.

Input: "cbbd"
Output: "bb"
```

state[i][j]: i 到 j 的子字符串是否为palindromic string

state[i][j] = state[i + 1][j - 1]　　如果str[i] == str[j]
　　false　　其它
Base case: state[i][i] = true
　　state[i][i+1] = str[i] == str[i + 1]

Input: “babad”

	0	1	2	3	4
0	b	false	true	false	false
1		a	false	true	false
2			b	false	false
3				a	false
4					d

```java
public String longestPalindrome(String s) {
  int n = s.length();
  String res = null;

  boolean[][] dp = new boolean[n][n];

  for (int i = n - 1; i >= 0; i--) {
    for (int j = i; j < n; j++) {
      dp[i][j] = s.charAt(i) == s.charAt(j) && (j - i < 3 || dp[i + 1][j - 1]);

      if (dp[i][j] && (res == null || j - i + 1 > res.length())) {
        res = s.substring(i, j + 1);
      }
    }
  }

  return res;
}
```

作业：根据解题思路，删除紫色部分，写出和我们解题思路一样的代码；这一份代码把base case以及最后得出结论的逻辑都写在一起了

Code出处：
https://leetcode.com/problems/longest-palindromic-substring/discuss/2921/Share-my-Java-solution-using-dynamic-programming

Palindromic Substrings
Given a string, your task is to count how many palindromic substrings in this string.

The substrings with different start indexes or end indexes are counted as different substrings even they consist of same characters.

Example:

```
Input: "abc"
Output: 3
Explanation: Three palindromic strings: "a", "b", "c".

Input: "aaa"
Output: 6
Explanation: Six palindromic strings: "a", "a", "a", "aa", "aa", "aaa".
```

解题思路和上个题目一样，仅仅需要在最后求解时作出修改。HOW？

[Maximum Length of Repeated Subarray](#)
Given two integer arrays A and B, return the maximum length of an subarray that appears in both arrays.

Example:

```
Input:
A: [1,2,3,2,1]
B: [3,2,1,4,7]
Output: 3
Explanation:
The repeated subarray with maximum length is [3, 2, 1].
```

state[i][j]: 以A[i]和B[j]结尾的最长公共子数组的长度
state[i][j] = 1 + state[i - 1][j - 1]   如果A[i] == B[j]
                  0   其它
Base case: state[i][0] = 0, state[0][j] = 0

	Empty	3	3, 2	3, 2, 1	3, 2, 1, 4	3, 2, 1, 4, 7
Empty	0	0	0	0	0	0
1	0	0	0	1	0	0
1, 2	0	0	1	0	0	0
1, 2, 3	0	1	0	0	0	0
1, 2, 3, 2	0	0	2	0	0	0
1, 2, 3, 2, 1	0	0	0	3	0	0

```java
public int findLength(int[] A, int[] B) {
    if(A == null||B == null) return 0;
    int m = A.length;
    int n = B.length;
    int max = 0;

    int[][] dp = new int[m + 1][n + 1];
    for(int i = 0;i <= m;i++){
        for(int j = 0;j <= n;j++){
            if(i == 0 || j == 0){
                dp[i][j] = 0;
            } else{
                if(A[i - 1] == B[j - 1]) {
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                    max = Math.max(max,dp[i][j]);
                }
            }
        }
    }
    return max;
}
```
作业：根据解题思路，删除紫色部分，写出和我们解题思路一样的代码；这一份代码把base case以及最后得出结论的逻辑都写在一起了

Code出处：
https://leetcode.com/problems/maximum-length-of-repeated-subarray/discuss/109039/Concise-Java-DP:-Same-idea-of-Longest-Common-Substring

其它类似题目：
Scramble String
TODO: https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-transaction-fee

可转换为2维矩阵类型 II

该类问题其递推表达式需要考虑多个state的值，通常需要用个loop去遍历某个范围内的state，或者其它非邻近区域的state

Partition Equal Subset Sum
Given a non-empty array containing only positive integers, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

Note:
- Each of the array element will not exceed 100.
- The array size will not exceed 200.

Example:
```
Input: [1, 5, 11, 5]
Output: true
```

```
Explanation: The array can be partitioned as [1, 5, 5] and [11].

Input: [1, 2, 3, 5]
Output: false
Explanation: The array cannot be partitioned into equal sum subsets.
```

state[i][j]: 是否可以用前 i 个数字可以相加等于 j （不一定要取所有的前 i 个数字）

state[i][j] = state[i-1][j] 或 state[i-1][j-nums[i]]

state[i-1][j] 表示不取第 i 个数字到subset里去求和，使其等于 j

state[i-1][j-nums[i]] 表示取第 j 个数字到subset里从而使得其和等于 j

Ex:

State[3][6] = state[2][6] || state[2][6 - 11] = false

State[3][11] = state[2][11] || state[2][11 - 11]

Base case: state[0][0] = true, state[i][0] = true, state[0][j] = false

参考背包问题

```
public boolean canPartition(int[] nums) {
    int sum = 0;

    for (int num : nums) {
        sum += num;
    }
    // ex: 5 -> 101 & 1 = 1
    if ((sum & 1) == 1) {
        return false;
    }
    sum /= 2;

    int n = nums.length;
    boolean[][] dp = new boolean[n+1][sum+1];
    for (int i = 0; i < dp.length; i++) {
        Arrays.fill(dp[i], false);
    }

    dp[0][0] = true;

    for (int i = 1; i < n+1; i++) {
        dp[i][0] = true;
    }
    for (int j = 1; j < sum+1; j++) {
        dp[0][j] = false;
    }

    // ex: state[3][6] -> 6 >= 11 == false
    for (int i = 1; i < n+1; i++) {
        for (int j = 1; j < sum+1; j++) {
            dp[i][j] = dp[i-1][j];
            if (j >= nums[i-1]) {
                dp[i][j] = (dp[i][j] || dp[i-1][j-nums[i-1]]);
            }
        }
    }

    return dp[n][sum];
}
```

Ones and Zeroes
In the computer world, use restricted resource you have to generate maximum benefit is what we always want to pursue.

For now, suppose you are a dominator of m 0s and n 1s respectively. On the other hand, there is an array with strings consisting of only 0s and 1s.

Now your task is to find the maximum number of strings that you can form with given m 0s and n 1s. Each 0 and 1 can be used at most once.

Note:
- The given numbers of 0s and 1s will both not exceed 100
- The size of given string array won't exceed 600.

Example:
```
Input: Array = {"10", "0001", "111001", "1", "0"}, m = 5, n = 3
Output: 4
Explanation: This are totally 4 strings can be formed by the using of 5 0s and 3
1s, which are "10","0001","1","0"

Input: Array = {"10", "0", "1"}, m = 1, n = 1
Output: 2
Explanation: You could form "10", but then you'd have nothing left. Better form "0"
and "1".
```

state[i][j]: **最多**用 i 个 0 和 j 个 1 能够组成的string个数
state[i][j] = 1 + max(state[i - str_x_num_of_zeros][j - str_x_num_of_ones])
其中 str_x 指代 Array 里的任意字符串
Base case: state[0][0] = 0

尝试Debug dp的值，来理解
s -> ‘10’
Dp[m][n] = Math.max(1 + dp[m - 1][n - 1], Dp[m][n])
Dp[m-1][n-1] = Math.max(1 + dp[m-2][n - 2], Dp[m-1][n-1])

Dp[m-6][n-7] = Math.max(1 + dp[m-7][n-8], Dp[m-6][n-7])
…..
s -> '1'
Dp[m][n] = Math.max(1 + dp[m][n - 1], Dp[m][n])
Dp[m-1][n-1] = Math.max(1 + dp[m-1][n-2], Dp[m-1][n-1])
Dp[m-6][n-6] = Math.max(1 + dp[m-6][n-7], Dp[m-6][n-6])

{1, 0, 10}
Dp[0][1] = 1; Dp[1][0] = 1; Dp[1][1] = 1;

```
public int findMaxForm(String[] strs, int m, int n) {
    int[][] dp = new int[m+1][n+1];
    for (String s : strs) {
        int[] count = count(s);
        for (int i=m;i>=count[0];i--)
            for (int j=n;j>=count[1];j--)
                dp[i][j] = Math.max(1 + dp[i-count[0]][j-count[1]], dp[i][j]);
    }
    return dp[m][n];
}

public int[] count(String str) {
    int[] res = new int[2];
    for (int i=0;i<str.length();i++)
        res[str.charAt(i) - '0']++;
    return res;
 }
```

Predict the Winner
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.

Examples:

```
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 player 1 will be left with 1 (or 2).
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.
```

```
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 which number player 2 choose, player 1 can choose 233.
Finally, player 1 has more score (234) than player 2 (12), so you need to return
True representing player1 can win.
```

state[i][j]: 先选数字的player在选取 i 到 j 整数后比另外一个player**多获取**的 score
state[i][j] = max(input[i] - state[i + 1][j], input[j] - state[i][j - 1])
这里分别表示当前的player选取的是第一个整数还是最后一个整数
这里用减号的是因为state[i + 1][j]和state[i][j - 1]分别表示当前palyer的对手能获胜的分数
Base case: state[i][i] = nums[i]　　即只有一个数字时

```
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;
}
```

Maximum Vacation Days
If you could take vacations in some particular cities and weeks. How would you schedule the traveling to maximize the number of vacation days you could take? There are certain rules and restrictions you need to follow:

- You can only travel among **N** cities, represented by indexes from 0 to N-1. Initially, you are in the city indexed **0 on Monday.**
- The cities are connected by flights. The flights are represented as a N*N matrix (not necessary symmetrical), called flights representing the airline status from the city i to the city j. If there is no flight from the city i to the city j, **flights[i][j] = 0**; Otherwise, **flights[i][j] = 1.** Also, flights[i][i] = 0 for all i.
- You totally have K weeks (each week has 7 days) to travel. You can **only take flights at most once per day** and can **only take flights on each week's Monday morning.** Since flight time is so short, we don't consider the impact of flight time.
- For each city, you can only have restricted vacation days in different weeks, given an N*K matrix called days representing this relationship. For the value of **days[i][j]**, it represents the **maximum days you could take vacation in the city i in the week j.**

You're given the flights matrix and days matrix, and you need to output the maximum vacation days you could take during K weeks.

Examples:

```
Input:flights = [[0,1,1],[1,0,1],[1,1,0]], days = [[1,3,1],[6,0,3],[3,3,3]]
Output: 12
Explanation:
Ans = 6 + 3 + 3 = 12.

One of the best strategies is:
1st week : fly from city 0 to city 1 on Monday, and play 6 days and work 1 day.
(Although you start at city 0, we could also fly to and start at other cities since
it is Monday.)
2nd week : fly from city 1 to city 2 on Monday, and play 3 days and work 4 days.
3rd week : stay at city 2, and play 3 days and work 4 days.

Input:flights = [[0,0,0],[0,0,0],[0,0,0]], days = [[1,1,1],[7,7,7],[7,7,7]]
Output: 3
Explanation:
Ans = 1 + 1 + 1 = 3.

Since there is no flights enable you to move to another city, you have to stay at
city 0 for the whole 3 weeks.
For each week, you only have one day to play and six days to work.
So the maximum number of vacation days is 3.

Input:flights = [[0,1,1],[1,0,1],[1,1,0]], days = [[7,0,0],[0,7,0],[0,0,7]]
Output: 21
Explanation:
Ans = 7 + 7 + 7 = 21

One of the best strategies is:
1st week : stay at city 0, and play 7 days.
2nd week : fly from city 0 to city 1 on Monday, and play 7 days.
3rd week : fly from city 1 to city 2 on Monday, and play 7 days.
```

Note:

- N and K are positive integers, which are in the range of [1, 100].
- In the matrix flights, all the values are integers in the range of [0, 1].
- In the matrix days, all the values are integers in the range [0, 7].
- You could stay at a city beyond the number of vacation days, but you should work on the extra days, which won't be counted as vacation days.
- If you fly from the city A to the city B and take the vacation on that day, the deduction towards vacation days will count towards the vacation days of city B in that week.
- We don't consider the impact of flight hours towards the calculation of vacation days.

state[i][j]: week i时在city j，最多可以获得的vacation数目

state[i][j] = max(state[i - 1][k] + days[j][i])   其中 k = 0...N - 1 且可以从 city k 到 city j

Base case: state[0][0] = 0（其余的用负值表示还未被给值）

state[0][j] = days[j][0]   即：第一周在 city j 度过

```
public int maxVacationDays(int[][] flights, int[][] days) {
    int m = days[0].length;
    int n = days.length;
    int[][] dp = new int[m][n];

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (i == 0) {
                dp[i][j] = j == 0 || flights[0][j] != 0 ? days[j][i]: Integer.MIN_VALUE;
                continue;
            }
            dp[i][j] = Integer.MIN_VALUE;
            for (int k = 0; k < m; k++) {
                // (k == j || flights[k][j] == 1) ->  可以从city k 到 j
                if ((k == j || flights[k][j] == 1) && dp[i - 1][k] != Integer.MIN_VALUE) {
                    dp[i][j] = Math.max(dp[i - 1][k] + days[j][i], dp[i][j]);
                }
            }
        }
    }

    int max = 0;
    for (int i = 0; i < n; i++) {
        max = Math.max(max, dp[m - 1][i]);
    }
    return max;
}
```

Best Time to Buy and Sell Stock IV

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most k transactions.

Note: You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

Example:

```
Input: [2,4,1], k = 2
Output: 2
Explanation:
Buy on day 1 (price = 2) and sell on day 2 (price = 4), profit = 4-2 = 2.

Input: [3,2,6,5,0,3], k = 2
Output: 7
```

```
Explanation:
Buy on day 2 (price = 2) and sell on day 3 (price = 6), profit = 6-2 = 4. Then buy
on day 5 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.
```

state[k][i]: 到第 i 天时，如果交易 k 次，所能获得的最大收益
state[k][i] = max(state[k][i - 1], prices[i] - prices[j] + state[k - 1][j - 1]) 其中 0 =< j < i
state[k][i - 1] 表示不在第 i 天做任何交易，从而可以获得的最大收益
prices[i] - prices[j] + state[k - 1][j - 1] 表示在第 j 天购入股票并在第 i 天出售从而获得的最大收益
Base case: dp[0, j] = 0 没有交易则收益为0
dp[i, 0] = 0 只有一个价格则无法执行任何交易

```
public int maxProfit(int k, int[] prices) {
  int n = prices.length;
  if (n <= 1)
      return 0;

  //如果 k >= n/2，你可以每天都交易，但是这里你只在能够确保盈利的时候交易
  if (k >=  n/2) {
      int maxPro = 0;
      for (int i = 1; i < n; i++) {
            if (prices[i] > prices[i-1])
                  maxPro += prices[i] - prices[i-1];
      }
      return maxPro;
  }

  int[][] dp = new int[k+1][n];
  for (int i = 1; i <= k; i++) {
      int localMax = dp[i-1][0] - prices[0];
      for (int j = 1; j < n; j++) {
            dp[i][j] = Math.max(dp[i][j-1],  prices[j] + localMax);
            localMax = Math.max(localMax, dp[i-1][j] - prices[j]);
      }
  }
  return dp[k][n-1];
}
```

TODO:
https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-transaction-fee

Not Useful
K Inverse Pairs Array

其它类似问题：
Freedom Trail
Soup Servings