DP (chapter 5, 6)

```
/*
1. 递归：把问题的规模逐步变小
2. 动态规划：从小规模的问题计算到大规模的问题
*/
```

Triangle

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle

```
[
     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]
]
```

The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).

Note:

Bonus point if you are able to do this using only O(n) extra space, where n is the total number of rows in the triangle.

Answer:

```
/*
SOL 1: REC
sum[i][j] = sum[i + 1][j] + sum[i + 1][j + 1];
*/
public int minimumTotal1(List<List<Integer>> triangle) {
    int rows = triangle.size();
    int[][] sum = new int[rows][rows];
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < rows; j++) {
            sum[i][j] = Integer.MAX_VALUE;
        }
    }
    return dfs(triangle, 0, 0, sum);
}
public int dfs(List<List<Integer>> triangle, int row, int col, int[][] sum) {
    if (sum[row][col] != Integer.MAX_VALUE) {
        return sum[row][col];
    }

    if (row == triangle.size() - 1) {
        sum[row][col] = triangle.get(row).get(col); // bottom-up
    } else {
        int left = dfs(triangle, row + 1, col, sum);
        int right = dfs(triangle, row + 1, col + 1, sum);
        sum[row][col] = triangle.get(row).get(col) + Math.min(left, right);
    }
    return sum[row][col];
}
```

```
/*
SOL 2: DP
*/
public int minimumTotal1(List<List<Integer>> triangle) {
    if (triangle == null || triangle.size() == 0) {
        return 0;
    }

    int rows = triangle.size();
    int[][] sum = new int[rows][rows];

    for (int i = rows - 1; i >= 0; i++) {
        for (int j = 0; j <= i; j++) {
            if (i == rows - 1) {
                sum[i][j] = triangle.get(i).get(j);
            } else {
                sum[i][j] = triangle.get(i).get(j) + Math.min(sum[i + 1][j], sum[i + 1][j + 1]);
            }
        }
        return sum[0][0];
    }
}
```

Unique Paths
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). How many possible unique paths are there? Above is a 3 x 7 grid. How many possible unique paths are there?
Note: m and n will be at most 100.
Answer:

```
/*
time O(m*n)
*/
public class Solution {
   public int uniquePaths(int m, int n) {
       if (m == 0 || n == 0) {
           return 0;
       }

       int[][] path = new int[m][n];
       for (int i = 0; i < m; i++) {
           path[i][0] = 1;
       }
       for (int i = 0; i < n; i++) {
           path[0][i] = 1;
       }
       for (int i = 1; i < m; i++) {
           for (int j = 1; j < n; j++) {
               path[i][j] = path[i - 1][j] + path[i][j - 1];
```

```
            }
        }
        return path[m - 1][n - 1];
    }
}
```

Unique Paths II
Follow up for "Unique Paths":
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.
For example,
There is one obstacle in the middle of a 3x3 grid as illustrated below.

```
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
```

The total number of unique paths is 2.
Note: m and n will be at most 100.
Answer:

```
/*
e.g.
[
  [0, 1, 0, 1]
  [0, 0, 0, 0]
  [0, 0, 0, 0]
]
00 1 01 break
otherwise: 00 1 01 0 02 1 03 0 goes worng
*/
public class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        if (obstacleGrid == null || obstacleGrid.length == 0 || obstacleGrid[0].length == 0) {
            return 0;
        }

        int rows = obstacleGrid.length;
        int cols = obstacleGrid[0].length;
        int[][] sum = new int[rows][cols];
        for (int i = 0; i < rows; i++) {
            if (obstacleGrid[i][0] == 1) {
                break; // bug: sum[i][0] = 0;
            } else {
                sum[i][0] = 1;
            }
        }
        for (int i = 0; i < cols; i++) {
            if (obstacleGrid[0][i] == 1) {
```

```
                break;
            } else {
                sum[0][i] = 1;
            }
        }
        for (int i = 1; i < rows; i++) {
            for (int j = 1; j < cols; j++) {
                if (obstacleGrid[i][j] == 1) {
                    sum[i][j] = 0;
                } else {
                    sum[i][j] = sum[i - 1][j] + sum[i][j - 1];
                }
            }
        }
        return sum[rows - 1][cols - 1];
    }
}
```

Minimum Path Sum
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.
Answer:

```
public class Solution {
    public int minPathSum(int[][] grid) {
        if (grid == null || grid.length == 0 || grid[0].length == 0) {
            return 0;
        }

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

Climbing Stairs
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?
Answer:
redo method

```
public class Solution {
  public int climbStairs(int n) {
     if (n <= 1) {
       return n;
     }

     int[] path = new int[n];
     path[0] = 1;
     path[1] = 2;
     for (int i = 2; i < n; i++) {
        path[i] = path[i - 1] + path[i - 2];
     }
     return path[n - 1];
  }
}
```

Jump Game
Given an array of non-negative integers, you are initially positioned at the first index of the array. Each element in the array represents your maximum jump length at that position. Determine if you are able to reach the last index. For example: A = [2,3,1,1,4], return true. A = [3,2,1,0,4], return false.
Answer:
redo method:

```
public class Solution {
    public boolean canJump(int[] A) {
        if (A == null || A.length == 0) {
            return false;
        }

        int size = A.length;
        boolean[] jump = new boolean[size];
        jump[0] = true;
        for (int i = 1; i < size; i++) {
            for (int x = 1; x <= i; x++) {
                if (jump[i - x] && A[i - x] >= x) {
                    jump[i] = true;
                    break;
                }
            }
        }
        return jump[size - 1];
    }
}
```

Jump Game II
Given an array of non-negative integers, you are initially positioned at the first index of the array. Each element in the array represents your maximum jump length at that position. Your goal is to reach the last index in the minimum number of jumps. For example: Given array A = [2,3,1,1,4] The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)
Answer:

```
public class Solution {
    public int jump(int[] A) {
        int[] min = new int[A.length];
        for (int i = 1; i < A.length; i++) {
            min[i] = Integer.MAX_VALUE;
            for (int j = 0; j < i; j++) {
                if (min[j] != Integer.MAX_VALUE && j + A[j] >= i) {
                    min[i] = min[j] + 1;
                    break;
                }
            }
        }
        return min[A.length - 1];
    }
}
```

132. 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. For example, given s = "aab", Return 1 since the palindrome partitioning ["aa","b"] could be produced using 1 cut.
Answer:

```
public class Solution {
    public int minCut(String s) {
        if (s == null || s.length() == 0) {
            return 0;
        }

        boolean[][] isPalindromeArray2 = getIsPalindrome(s);
        // boolean[][] isPalindromeArray2 = new boolean[s.length()][s.length()];
        // isPalindromeArray2 = getIsPalindrome(s);

        int[] min = new int[s.length() + 1];
        min[0] = 0;
        for (int i = 1; i <= s.length(); i++) {
            min[i] = Integer.MAX_VALUE;
            for (int j = 1; j <= i; j++) {
                if (isPalindromeArray2[i - j][i - 1] && min[i - j] != Integer.MAX_VALUE) {
                    min[i] = Math.min(min[i], min[i - j] + 1);
                    // clabbal actual cut is 1, thus return minus 1 later
                }
            }
        }
```

```
        return min[s.length()] - 1;
    }
/*
    private boolean isPalindrome(String s, int start, int end) {
        for (int i = start, j = end; i < j; i++, j++) {
        // for (int i, int j; i < j; i++, j--) { // error: <identifier> expected
            if (s.charAt(i) != s.charAt(j)) {
                return false;
            }
            // bug: return true;
        }
        return true;
    }
*/
    private boolean[][] getIsPalindrome(String s) {
        boolean[][] isPalindromeArray = new boolean[s.length()][s.length()];
        for (int i = 0; i < s.length(); i++) {
            isPalindromeArray[i][i] = true;
        }
        for (int i = 0; i < s.length() - 1; i++) {
            isPalindromeArray[i][i + 1] = (s.charAt(i) == s.charAt(i + 1));
        }
        for (int len = 2; len < s.length(); len++) {
            for (int start = 0; start + len < s.length(); start++) {
                isPalindromeArray[start][start + len] = isPalindromeArray[start + 1][start + len - 1] &&
s.charAt(start) == s.charAt(start + len);
            }
        }
        return isPalindromeArray;
    }
}
```

Word Break

Given a string s and a dictionary of words dict, determine if s can be segmented into a space-separated sequence of one or more dictionary words. For example, given s = "leetcode", dict = ["leet", "code"]. Return true because "leetcode" can be segmented as "leet code".

Answer:

time O(NL) i 0-n j L 实际上是N(L^2)，至少要遍历一遍L个字母

```
/* leetcode "leet" "code" maxLen = 4
          i = 1 can[1] = false
          j = 1 can[1 - 1] = true
          word substring(1 - 1, 1) "l" not
          note: break statement only take you out of the inside for loop
          i = 2 can[2] = false
          j = 1 can[2 - 1] = false
          j = 2 can[2 - 2] = true substring(2 - 2, 2) "le" not
          i = 3 can[3] = false
          j = 1 can[3 - 1] = false
          j = 2 can[3 - 2] = false
```

```
        j = 3 can[3 - 3] = true substring(3 - 3, 3) "lee" not
        i = 4 can[4] = false
        j = 1 can[4 - 1] = false
        j = 2 can[4 - 2] = false
        j = 3 can[4 - 3] = false
        j = 4 can[4 - 4] = true substring(4 - 4, 4) "leet" can[4] = true
        i = 5 can[5] = false
        j = 1 can[5 - 1] true substring(5 - 1, 5) "c" not
        j = 2 can[5 - 2] false
        j = 3 can[5 - 3] false j = 4 can[5 - 4] false
        i = 6 can[6] = false
        j = 1 can[6 - 1] = false
        j = 2 can[6 - 2] = true substring(6 - 2, 6) "co" not
        j = 3 can[6 - 3] = false
        j = 4 can[6 - 4] = false
        i = 7 can[7] = false
        j = 1 can[7 - 1] = false
        j = 2 can[7 - 2] = false
        j = 3 can[7 - 3] = true substring(7 - 3, 7) "cod" not
        j = 4 can[7 - 4] = false
        i = 8 can[8] = false
        j = 1 can[8 - 1] = false j = 2 can[8 - 2] = false j = 3 can[8 - 3] = false
        j = 4 can[8 - 4] = true substring(8 - 4, 8) "code" can[8] = true
      */
      /*
      beginIndex -- the begin index, inclusive.
      endIndex -- the end index, exclusive.
      String str = "Hello";
      String a = str.substring(2, 4);  // a is "ll" (not "llo")
      String b = str.substring(0, 3);  // b is "Hel"
      String c = str.substring(4, 5);  // c is "o" -- the last char
      */
public class Solution {
   public boolean wordBreak(String s, Set<String> dict) {
      if (s == null || s.length() == 0) {
        return false;
      }

      int maxLen = getMaxLength(dict);
      boolean[] canSegment = new boolean[s.length() + 1]; // according to the definition of
substring
      canSegment[0] = true;
      for (int i = 1; i <= s.length(); i++) {
          canSegment[i] = false;
          for (int j = 1; j <= maxLen && j <= i; j++) {
              if (!canSegment[i - j]) {
                  continue;
              }
```

```
                String word = s.substring(i - j, i);
                if (dict.contains(word)) {
                    canSegment[i] = true;
                    break; // bug 1 break statement just take you out of the inside loop
                }
            }
        }
        return canSegment[s.length()];
    }

    private int getMaxLength(Set<String> dict) {
        int maxLen = 0;
        for (String word : dict) {
            maxLen = Math.max(maxLen, word.length());
        }
        return maxLen;
    }
}
```

Longest Increasing Subsequence
Given a sequence of integers, find the longest increasing subsequence (LIS). You code should return the length of the LIS. Example For [5, 4, 1, 2, 3], the LIS is [1, 2, 3], return 3 For [4, 2, 4, 5, 3, 7], the LIS is [4, 4, 5, 7], return 4
Answer:

```
/*
194567 follow the instruction of problem should return 14567
i = 0 f[0] = 1
i = 1 f[1] = 1 j = 0 nums[1] > nums[0] f[1] = max(1, f[0] + 1) = 2 max = 2
i = 2 f[2] = 1 j = 0 nums[2] > nums[0] f[2] = max(1, f[0] + 1) = 2 nums[2] < nums[1] max = 2
i = 3 f[3] = 1 j = 0 nums[3] > nums[0] f[3] = 2 nums[3] < nums[1] nums[3] > nums[2] f[3] =
max(f[3], f[2] + 1) = 3
i = 4 f[4] = 1 j = 0 nums[4] > nums[0] f[4] = 2 nums[4] < nums[1] nums[4] > nums[2] f[4] =
max(f[4], f[2] + 1) = 3 nums[4] > nums[3] f[4] = max(f[4], f[3] + 1) = 4
i = 5 f[5] = 1 j = 0 nums[5] > nums[0] f[5] = 2 nums[5] < nums[1] nums[5] > nums[2] f[5] =
max(f[5], f[2] + 1) = 3 nums[5] > nums[3] f[5] = max(f[5], f[3] + 1) = 4 nums[5] > nums[4] f[5] =
max(f[5], f[4] + 1) = 5
*/
public class Solution {
    public int longestIncreasingSubsequence(vector<int> nums) {

        int[] f = new int[nums.size()];
        int max = 0;
        for (int i = 0; i < nums.size(); i++) {
            f[i] = 1;
            for (int j = 0; j < i; j++) {
                if (nums[i] >= nums[j]) {
                    f[i] = Math.max(f[i], f[j] + 1);
                }
            }
```

```
            if (f[i] > max) {
                max = f[i];
            }
        }
        return max;
    }
}
```

Longest Common Subsequence
Given two strings, find the longest comment subsequence (LCS). Your code should return the length of LCS. Example For "ABCD" and "EDCA", the LCS is "A" (or D or C), return 1 For "ABCD" and "EACB", the LCS is "AC", return 2
Answer:

```
/*
sequence is able to jump
*/
public class Solution {
    /**
    * @param A, B: Two strings.
    * @return: The length of longest common subsequence of A and B.
    */
    public int longestCommonSubsequence(String A, String B) {
        int lena = A.length();
        int lenb = B.length();
        boolean[][] sameCount = new boolean[lena + 1][lenb + 1];
        // D[i][j] 定义为s1, s2的前i,j个字符串的最长common subsequence.
        for (int i = 0; i <= lena; i++) {
            for (int j = 0; j <= lenb; j++) {
                if (i == 0 || j == 0) {
                    sameCount[i][j] = 0;
                } else {
                    if (A.charAt(i - 1) == B.charAt(j - 1)) {
                        sameCount[i][j] = sameCount[i - 1][j - 1] + 1;
                    } else {
                        sameCount[i][j] = Math.max(sameCount[i][j - 1], sameCount[i - 1][j]);
                    }
                }
            }
        }
        return sameCount[A.length][B.length];
    }
}
```

Longest Common Substring
Given two strings, find the longest common substring.
Return the length of it.
Note
The characters in substring should occur continiously in original string. This is different with subsequence.

Example
Given A="ABCD", B="CBCE", return 2
Answer:

```
public class Solution {
    public int longestCommonSubstring(String A, String B) {
        if (A == null || B == null) { // if (A.length() == 0 || B.length() == 0)
            return 0;
        }

        int len = 0;
        int[][] sameCount = new int[A.length() + 1][B.length() + 1];
        for (int i = 0; i <= A.length(); i++) {
            for (int j = 0; j <= B.length(); j++) {
                if (i == 0 || j == 0) {
                    sameCount[i][j] = 0;
                } else {
                    if (A.charAt(i - 1) == B.charAt(j - 1)) {
                        sameCount[i][j] = sameCount[i - 1][j - 1] + 1;
                    } else {
                        sameCount[i][j] = 0;
                    }
                }
                len = Math.max(sameCount[i][j], len);
            }
        }
        return len;
    }
}
```

Edit Distance
Given two words word1 and word2, find the minimum number of steps required to convert word1 to word2. (each operation is counted as 1 step.)
You have the following 3 operations permitted on a word:
a) Insert a character
b) Delete a character
c) Replace a character
Answer:

```
/*
note: two sequence dp different with matrix dp
*/
/*
定义D[i][j] 为string1 前i个字符串到 string2的前j个字符串的转化的最小步。
1. 初始化： D[0][0] = 0; 2个为空 不需要转
2. D[i][0] = D[i - 1][0] + 1. 就是需要多删除1个字符
3. D[0][j] = D[0][j - 1] + 1. 就是转完后需要添加1个字符
a、给word1插入一个和word2最后的字母相同的字母，这时word1和word2的最后一个字母就一样了，此时编辑距离等于1（插入操作） + 插入前的word1到word2去掉最后一个字母后的编辑距离 D[i][j - 1] + 1
例子： 从ab --> cd
```

我们可以计算从 ab --> c 的距离，也就是 D[i][j - 1]，最后再在尾部加上d
b、删除word1的最后一个字母，此时编辑距离等于1（删除操作） + word1去掉最后一个字母到word2的编辑距离
D[i - 1][j] + 1
例子： 从ab --> cd
我们计算从 a --> cd 的距离，再删除b, 也就是 D[i - 1][j] + 1
c 、把word1的最后一个字母替换成word2的最后一个字母，此时编辑距离等于 1（替换操作） + word1和word2去掉最后一个字母的编辑距离。
这里有2种情况，如果最后一个字符是相同的，即是：D[i - 1][j - 1]，因为根本不需要替换，否则需要替换，就是
D[i - 1][j - 1] + 1
然后取三种情况下的最小距离
基于证明，当最后一个字符相同时，我们其实可以直接让D[i][j] = D[i - 1][j - 1].
*/

```
public class Solution {
    public int minDistance(String word1, String word2) {
        if (word1 == null || word2 == null) {
            return 0;
        }

        int len1 = word1.length();
        int len2 = word2.length();
        int[][] min = new int[len1 + 1][len2 + 1];

        for (int i = 0; i <= len1; i++) {
            for (int j = 0; j <= len2; j++) {
                if (i == 0) {
                    min[i][j] = j;
                } else if (j == 0) {
                    min[i][j] = i;
                } else {
                    if (word1.charAt(i - 1) != word2.charAt(j - 1)) {
                        //min[i][j] = Math.min(min[i][j - 1], min[i - 1][j]);
                        min[i][j] = Math.min(min[i - 1][j - 1], min[i][j - 1]);
                        min[i][j] = Math.min(min[i][j], min[i - 1][j]);
                        min[i][j]++;
                    } else {
                        min[i][j] = min[i - 1][j - 1];
                    }
                }
            }
        }
        return min[len1][len2];
    }
}
```

Distinct Subsequences
Given a string S and a string T, count the number of distinct subsequences of T in S.

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).
Here is an example:
S = "rabbbit", T = "rabbit"
Return 3.
Answer:

```
/*
在S中找T包含的所有字符（要按T的顺序），一共有几种组合?
S = "rabbbit" T = "rabbit"
rab1b2it
rab1b3it
rab2b3it
  01234567
   rabbbit
0 11111111
1r01111111
2a00111111
3b00012333
4b00001333
5i00000033
6t00000003
i = 0 j = 0 num[0][0] = 1
i = 1 j = 0 num[1][0] = 1 j = 1 S.charAt(1 - 1) == T.charAt(1 - 1) num[1][1] = 0 + num[0][0] = 1 j =
2 num[1][2] = 0 + num[0][2] = 0 ...
i = 2 j = 0 num[2][0] = 1 j = 1 num[2][1] += num[1][1] = 1 j = 2 num[2][2] += num[1][1] = 1 j = 3
num[2][3] += num[2][2] = 1
i = 3 j = 0 num[3][0] = 1 j = 1 num[3][1] += num[2][1] = 1 j = 2 num[3][2] += num[2][2] = 1 j = 3
num[3][3] += num[2][2] = 1 j = 4 num[3][4] += num[2][4] = 0
i = 4 j = 0 num[4][0] = 1 j = 1 num[4][1] += num[3][1] = 1 j = 2 num[4][2] += num[3][2] = 1 num[4]
[3] += num[3][2] = 1, num[4][3] += num[3][3] = 2 num[4][4] += num[3][3] = 1 num[4][5] += num[3]
[5] = 0
*/
public class Solution {
    public int numDistinct(String S, String T) {
        if (S == null || T == null) {
            return 0;
        }
        if (S.length() < T.length()) {
            return 0;
        }

        int[][] num = new int[S.length() + 1][T.length() + 1];
        for (int i = 0; i <= S.length(); i++) {
            for (int j = 0; j <= T.length(); j++) {
                if (i == 0 && j == 0) {
                    num[i][j] = 1;
                } else if (i == 0) {
                    num[i][j] = 0;
```

```
            } else if (j == 0) {
                num[i][j] = 1;
            } else {
                num[i][j] = 0;
                if (S.charAt(i - 1) == T.charAt(j - 1)) {
                    num[i][j] += num[i - 1][j - 1];
                }
                num[i][j] += num[i - 1][j];
            }

        }
    }
    return num[S.length()][T.length()];
  }
}
```

Interleaving String
Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2.
For example,
Given:
s1 = "aabcc",
s2 = "dbbca",
When s3 = "aadbbcbcac", return true.
When s3 = "aadbbbaccc", return false.
Answer:

```
/*
solution 1: Recursion with memory
*/
public class Solution {
   public static boolean isInterleave1(String s1, String s2, String s3) {
      if (s1 == null || s2 == null || s3 == null) {
         return false;
      }

      int len1 = s1.length();
      int len2 = s2.length();
      int len3 = s3.length();

      // The length is not equal, just return false.
      if (len1 + len2 != len3) {
         return false;
      }

      int[][][] memory = new int[len1 + 1][len2 + 1][len3 + 1];
      for (int i = 0; i <= len1; i++) {
         for (int j = 0; j <= len2; j++) {
            for (int k = 0; k <= len3; k++) {
               memory[i][j][k] = -1;
            }
```

```
            }
        }

        return recMemory(s1, 0, s2, 0, s3, 0, memory);
    }

    public static boolean recMemory(String s1, int index1, String s2,
            int index2, String s3, int index3, int[][][] memory) {
        int len1 = s1.length();
        int len2 = s2.length();
        int len3 = s3.length();

        if (index3 == len3 && index1 == len1 && index2 == len2) {
            return true;
        }

        if (memory[index1][index2][index3] != -1) {
            return memory[index1][index2][index3] == 1;
        }

        // 第一个字符，有2种可能：来自s1, 或是来自s2
        boolean ret = false;
        if (index1 < len1 && s1.charAt(index1) == s3.charAt(index3)) {
            ret = recMemory(s1, index1 + 1, s2, index2, s3, index3 + 1, memory);
        }

        // 如果不成功(首字母不来自于s1)，尝试另一种可能
        if (!ret && index2 < len2 && s2.charAt(index2) == s3.charAt(index3)) {
            ret = recMemory(s1, index1, s2, index2 + 1, s3, index3 + 1, memory);
        }

        memory[index1][index2][index3] = ret ? 1 : 0;
        return ret;
    }
}
/*
solution 2: dp
*/
/*
D[i][j]: 定义为s1 (前i个字符) s2(前j个字符) s3(i+j 个字符) 是不是交叉字符
递推公式： (s1.i == s3.(i+j) && D[i-1][j]) ll (s2.j == s3.(i+j) && D[i][j - 1])
分别从s1,s2两种可能性来匹配 ，两者有一个成立就行了。
s1 s3 首字母相同，继续查i -1 与 i + j -1 是否isInterleave1
s2 s3 首字母相同，继续查j -1 与 i + j -1 是否isInterleave1
初始化：D 0,0 就是true，因为都是空.
D[0][j] 就是判断一下str2与str3是不是尾字符相同，及D[0][j - 1]是不是true
D[i][0] 就是判断一下str1与str3是不是尾字符相同，及D[i - 1][0]是不是true
*/
/*
```

by yiming Apr/11/15

```
s1 = "aabcc",
s2 = "dbbca",
s3 = "aadbbcbcac",
i = 1 j = 1 s3.charAt(1) == s1.charAt(0), interleaved[0][1] s3.charAt(1) != s2.charAt(0),
interleaved[1][0] -> interleaved[1][1] = true
*/
/*
time O(N^2)
*/
public class Solution {
    public boolean isInterleave(String s1, String s2, String s3) {
        if (s1.length() + s2.length() != s3.length()) {
            return false;
        }

        boolean[][] interleaved = new boolean[s1.length() + 1][s2.length()];
        interleaved[0][0] = true;
        for (int i = 1; i <= s1.length(); i++) {
            if (s3.charAt(i - 1) == s1.charAt(i - 1) && interleaved[i - 1][0]) {
                interleaved[i][0] = true;
            }
        }
        for (int j = 1; j <= s2.length(); j++) {
            if (s3.charAt(j - 1) == s2.charAt(j - 1) && interleaved[0][j - 1]) {
                interleaved[0][j] = true;
            }
        }
        for (int i = 1; i <= s1.length(); i++) {
            for (int j = 1; j<= s2.length(); j++) {
                if (((s3.charAt(i + j - 1) == s1.charAt(i - 1)) && interleaved[i - 1][j]) || ((s3.charAt(i + j - 1)
== s2.charAt(j - 1)) && interleaved[i][j - 1])) {
                    interleaved[i][j] = true;
                }
            }
        }
        return interleaved[s1.length()][s2.length()];
    }

    public boolean isInterleave(String s1, String s2, String s3) {
        if (s1 == null || s2 == null || s3 == null) {
            return false;
        }

        int len1 = s1.length();
        int len2 = s2.length();
        int len3 = s3.length();

        if (len1 + len2 != len3) {
            return false;
```

```
        }

        boolean[][] D = new boolean[len1 + 1][len2 + 1];

        for (int i = 0; i <= len1; i++) {
            for (int j = 0; j <= len2; j++) {
                D[i][j] = false;
                if (i == 0 && j == 0) {
                    D[i][j] = true;
                    continue;
                }

                if (i != 0) {
                    D[i][j] |= s1.charAt(i - 1) == s3.charAt(i + j - 1) && D[i - 1][j];
                }

                if (j != 0) {
                    D[i][j] |= s2.charAt(j - 1) == s3.charAt(i + j - 1) && D[i][j - 1];
                }
            }
        }

        return D[len1][len2];
    }
}
```

Backpack
Given n items with size A[i], an integer m denotes the size of a backpack. How full you can fill this backpack?
Note
You can not divide any item into small pieces.
Example
If we have 4 items with size [2, 3, 5, 7], the backpack size is 11, we can select 2, 3 and 5, so that the max size we can fill this backpack is 10. If the backpack size is 12. we can select [2, 3, 7] so that we can fulfill the backpack.
You function should return the max size we can fill in the given backpack.
Answer:

```
/*
这个问题不能保证速度比搜索来的快
最坏情况：items的大小 1, 2, 4, 8, 16 … 对2的幂次数据，搜索和dp做速度都是一样，没有优化
time O(N*target)
1 2 3
-> 1 2 ［3］ 4 5 6 可以组成的和这么几种情况 用搜索 3 是由本身的3组成还是由 1 2 组成，重复运算dp来做可以优化
1 2 4
-> 1 2 3 4 5 6 7 三个数组成的和最坏的情况 此情况根本没有重复运算，所以dp没有任何优化
*/
/*
f[i][j] “前i”个数,取出一些能否凑成j, here j is the target number
```

A = [2, 3, 5, 7] m = 11
i = 0 j = 0 can[1][0] = can[0][0] = true 0 < A[0] j = 1 can[1][1] = can[0][1] = false 1 < A[0] j = 2 can[1][2] = can[0][2] = false 2 = A[0], can[0][2 - 2] = true can[1][2]
*/

```
public class Solution {
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @return: The maximum size
     */
    public int backPack(int m, int[] A) {
        boolean[][] can = new boolean[A.length + 1][m + 1];
        for (int i = 0; i <= A.length; i++) {
            for (int j = 0; j <= m; j++) {
                can[i][j] = false;
            }
        }
        can[0][0] = true;
        for(int i = 0; i < A.length; i++) {
            for (int j = 0; j <= m; j++) {
                can[i + 1][j] = can[i][j];
                if (j >= A[i] && can[i][j - A[i]]) {
                    can[i + 1][j] = true;
                }
            }
        }
        for (int i = m; i > 0; i--) {
            if (can[A.length][i]) {
                return i;
            }
        }
        return 0;
    }
}
```

Backpack II
Given n items with size A[i] and value V[i], and a backpack with size m. What's the maximum value can you put into the backpack?
Note
You cannot divide item into small pieces and the total size of items you choose should smaller or equal to m.
Example
Given 4 items with size [2, 3, 5, 7] and value [1, 5, 2, 4], and a backpack with size 10. The maximum value is 9.
Answer:

```
public class Solution {
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A & V: Given n items with size A[i] and value V[i]
```

```
     * @return: The maximum value
     */
    public int backPackII(int m, int[] A, int V[]) {
        int[][] nums = new int[A.length + 1][m + 1];

        for (int i = 0; i <= m; i++) {
            nums[0][i] = Integer.MIN_VALUE
        }

        for (int i = 0; i < A.length; i++) {
            nums[i][0] = 0;
        }

        for (int i = 1; i < A.length; i++) {
            for (int j = 1; j <= m; j++) {
                nums[i][j] = nums[i - 1][j];
                if (j >= A[i]) {
                    nums[i][j] = Math.max(nums[i][j], nums[i - 1][j - A[i]] + v[i]);
                }
            }
        }

        int store = Integer.MIN_VALUE;
        for (int i = m; i > 0; i--) {
            store = Math.max(nums[A.length][i], store);
        }
        return store;
    }
}
```

k Sum
Given n distinct positive integers, integer k (k <= n) and a number target.
Find k numbers where sum is target. Calculate how many solutions there are?
Answer:

```
/*
D[i][j][t]前i个数中，挑出j个数，组成和为t有多少方案
D[0][0][0]表示在一个空集中找出0个数，target为0，则有1个解，就是什么也不挑
（1）我们可以把当前A[i - 1]这个值包括进来，所以需要加上D[i - 1][j - 1][t - A[i - 1]]（前提是t - A[i
- 1]要大于0）
（2）我们可以不选择A[i - 1]这个值，这种情况就是D[i - 1][j][t]，也就是说直接在前i-1个值里选择
一些值加到target.
*/
public class Solution {
/**
* @param A: an integer array.
* @param k: a positive integer (k <= length(A)) * @param target: a integer
* @return an integer
*/
    public int kSum(int A[], int k, int target) {
```

```
        if (target < 0) {
            return 0;
        }

        int[][][] D = new int[A.length + 1][k + 1][target + 1];

        for (int i = 0; i <= A.length; i++) {
            for (int j = 0; j <= k; j++) {
                for (int m = 0; m <= target; m++) {
                    // 找0个数，目标为0，则一定是有1个解
                    if (j == 0 && t == 0) {
                        D[i][j][t] = 1;
                    } else if (!(i == 0 || j == 0 || t == 0)) {
                        D[i][j][m] = D[i - 1][j][m];
                        if (m >= A[i - 1]) {
                            D[i][j][m] += D[i - 1]D[j - 1][m - A[i - 1]];
                        }
                    }
                }
            }
        }
        return D[A.length][k][target];

    }
}
```

Minimum Adjustment Cost
Given an integer array, adjust each integers so that the difference of every adjcent integers are not greater than a given number target.
If the array before adjustment is A, the array after adjustment is B, you should minimize the sum of |A[i]-B[i]|
Note
You can assume each number in the array is a positive integer and not greater than 100
Example
Given [1,4,2,3] and target=1, one of the solutions is [2,3,2,3], the adjustment cost is 2 and it's minimal. Return 2.
Answer:

```
/*
D[i][v]: 把index = i的值修改为v，所需要的最小花费
当前index为v时，我们把上一个index从1-100全部过一次，取其中的最小值（判断一下前一个跟当前的是不是abs <= target）
time O(n*A*T)
*/
public class Solution {
/**
* @param A: An integer array.
* @param target: An integer.
*/
    public int MinAdjustmentCost(ArrayList<Integer> A, int target) {
```

```
        if (A == null || A.size() == 0) {
            return 0;
        }

        int[][] D = new int[A.size()][101];
        for (int i = 0; i < A.size(); i++) {
            for (int j  = 1; j <= 100; j++) {
                D[i][j] = Integer.MAX_VALUE;
                if (i == 0) {
                    D[i][j] = Math.abs(j - A.get(i));
                } else {
                    for (int k = 1; k <= 100; k++) {
                        if (Math.abs(j - k) > target) {
                            continue;
                        }
                        int dif = Math.abs(j - A.get(i)) + D[i - 1][k];
                        D[i][j] = Math.min(D[i][j], dif);
                    }
                }
            }
        }
        int ret = Integer.MAX_VALUE;
        for (int i = 1; i <= 100; i++) {
            ret = Math.min(ret, D[A.size() - 1][i]);
        }
        return ret;
    }
}
```

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, and there exists one unique longest palindromic substring.
Answer:

```
/*
状态表达式：D[i][j] 表示i,j这2个索引之间的字符串是不是回文。
递推公式： D[i][j] = if ( char i == char j) &&  （D[i + 1][j - 1]  ll  j - i <= 2））
(i, j)
推i,j的时候用到了i+1, j-1，其实意思就是在计算i,j时，关于同一个j-1的所有的i必须要计算过。
我们需要的是i+1, j - 1，实际上就是左下角的值。
1. 00
2. 00 01
      11
3. 00 01 02 (-> 11)
      11 12
        22
3. 00 01 02 03 (-> 12)
      11 12 13 (-> 22)
        22 23
          33
只要我们一列一列计算，就能够成功地利用这个动规公式。
注意：一行一行计算就会失败!
所以我们的循环的设计是这样的:
for (int j = 0; j < len; j++)
  { for (int i = 0; i <= j; i++) {}}
*/
public class Solution {
    public String longestPalindrome(String s) {
        if (s == null ll s.length() == 0) {
            return null;
        }

        String ret = null;
        int max = 0;
        boolean[][] D = new boolean[s.length()][s.length()];
        for (int j = 0; j < s.length(); j++) {
            for (int i = 0; i <= j; i++) {
                D[i][j] = s.charAt(i) == s.charAt(j) && (j - i <= 2 ll D[i + 1][j - 1]);
                if (D[i][j]) {
                    if (j - i + 1 > max) {
                        max = j - i + 1;
                        ret = s.substring(i, j + 1); // substring(i, j) i inclusive j exclusive
                    }
                }
            }
        }
        return ret;
    }
}
```

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.
Answer:

```
/*
time O(n)
dp[i][1] means we rob the current house and dp[i][0] means we don't,
num has num.length houses: 0~(num.length - 1); plus one more node to store the result after adding num[num.length - 1], thus the size should be defined num.length + 1
*/
/*
e.g. 一个数组，选出不相邻子序列，要求子序列和最大，
[4,9,6]=10
[4,10,3,1,5]=15
follow up问到如何测试，测试用例设计，时间空间复杂度
*/
public class Solution {
    public int rob(int[] num) {
        int[][] dp = new int[num.length + 1][2];
        for (int i = 1; i <= num.length; i++) {
            dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1]);
            dp[i][1] = num[i - 1] + dp[i - 1][0];
        }
        return Math.max(dp[num.length][0], dp[num.length][1]);
    }
}
```

200. Number of Islands
Given a 2d grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.
Example 1:
11110
11010
11000
00000
Answer: 1
Example 2:
11000
11000
00100
00011
Answer: 3
Answer:

by yiming Apr/11/15

```
/*
time O(n^2) space O(n^2)
11110
11010
11000
00000

i 0
j 0 (0, 0)1

(0, 0)1 dfs not visited (1, 0)1 (0, 1)1

(1, 0)1 dfs (2, 0)1 (1, 1)1
(0, 1)1 dfs (0, 2)1

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

(3, 0)1 dfs (3, 1)1 (4, 0)0
(1, 2)1 dfs (1, 3)0 (2, 2)0

(3, 1)1 dfs (3, 2)0 (4, 1)0
*/
public class Solution {
    public int numIslands(char[][] grid) {
        if (grid == null || grid.length == 0) {
            return 0;
        }

        int m = grid.length;
        int n = grid[0].length;
        int count = 0;
        boolean[][] visited = new boolean[m][n];

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == '1' && !visited[i][j]) {
                    dfs(grid, i, j, visited);
                    count++;
                }
            }
        }
        return count;
    }

    private void dfs(char[][] grid, int i, int j, boolean[][] visited) {
        if (i < 0 || i >= grid.length || j < 0 || j >= grid[0].length) {
            return;
```

```
        } else if (visited[i][j] || grid[i][j] != '1') { // here limit the bound of land
            return;
        }

        visited[i][j] = true;
        dfs(grid, i - 1, j, visited);
        dfs(grid, i + 1, j, visited);
        dfs(grid, i, j - 1, visited);
        dfs(grid, i, j + 1, visited);
    }
}
```