一.基本思想

一般来说，只要问题可以划分为规模更小的字问题，并且原问题的最优解中包含了子问题的最优解，则可以考虑用动态规划解决。动态规划的实质是分治思想和解决冗余。因此，动态规划是一种将问题实例分解为更小的/相似的子问题，并存储子问题的解，使得每个子问题只求解一次，最终获得原问题的答案，以解决最优化问题的算法策略。

与贪心法的关系：

1.与贪心法类似，都是将问题实例归纳为更小的、相似的子问题，并通过求解子问题产生一个全局最优解。

2.贪心法选择当前最优解，而动态规划通过求解局部子问题的最优解来达到全局最优解。

与递归法的关系：

1.与递归法类似，都是将问题实例归纳为更小的、相似的子问题。

2.递归法需要对子问题进行重复计算，需要耗费更多的时间与空间，而动态规划对每个子问题只求解一次。对递归法进行优化，可以使用记忆化搜索的方式。它与递归法一样，都是自顶向下的解决问题，动态规划是自底向上的解决问题。

递归问题——>重叠子问题——> 1.记忆化搜索（自顶向上的解决问题）；2.动态规划（自底向上的解决问题）

二.实际应用

70.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?

Note: Given *n* will be a positive integer.

Example 1:

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

Example 2:

Input: 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

动态规划解法：

```
class Solution {
public:
    int climbStairs(int n) {
        if(n == 1) return 1;
        if(n == 2) return 2;
        vector<int> memo(n+1, -1);
        memo[1] = 1;
        memo[2] = 2;
        for(int i = 3; i <= n; i++){
            memo[i] = memo[i-1] + memo[i-2];
        }
        return memo[n];
    }
};
```

120. 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.

空间复杂度：o(n^2)

```
class Solution {
public:
    int minimumTotal(vector<vector<int>>& triangle) {
        int n = triangle.size();
        vector<vector<int>> memo(n, vector<int>(n, -1));
        for(int j = 0; j < n; j++){
            memo[n-1][j] = triangle[n-1][j];
        }
        for(int i = n - 2; i >= 0; i--){
            for(int j = 0; j <= i; j++){
                memo[i][j] = min(memo[i+1][j], memo[i+1][j+1]) + triangle[i][
            }
        }
        return memo[0][0];
    }
};
```

空间复杂度：o(n)

```
class Solution {
public:
    int minimumTotal(vector<vector<int>>& triangle) {
        int n = triangle.size();
        vector<int> memo(n, -1);
        for(int j = 0; j < n; j++){
            memo[j] = triangle[n-1][j];
        }
        for(int i = n - 2; i >= 0; i--){
            for(int j = 0; j <= i; j++){
                memo[j] = min(memo[j], memo[j+1]) + triangle[i][j];
            }
        }
        return memo[0];
    }
};
```

64. 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.

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.`

空间复杂度：o(n^2)

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        int m = grid.size();
        int n = grid[0].size();
        vector<vector<int> > memo(m, vector<int>(n, -1));

        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                int tmp = 0;
                if((i - 1 >= 0) && (j-1 >= 0))
                    tmp = min(memo[i-1][j],memo[i][j-1]);
                else if(i - 1 >= 0)
                    tmp = memo[i-1][j];
                else if(j - 1 >= 0)
                    tmp = memo[i][j-1];
                else
                    tmp = 0;
                memo[i][j] = tmp + grid[i][j];
            }
        }
        return memo[m-1][n-1];
    }
};
```

空间复杂度：o(n)

```
class Solution {
public:
```

```
    int minPathSum(vector<vector<int>>& grid) {

     int m = grid.size();          int n = grid[0].size();
        vector<int> memo(n, -1);

        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                int tmp = 0;
                if((i - 1 >= 0) && (j-1 >= 0))
                    tmp = min(memo[j],memo[j-1]);
                else if(i - 1 >= 0)
                    tmp = memo[j];
                else if(j - 1 >= 0)
                    tmp = memo[j-1];
                else
                    tmp = 0;
                memo[j] = tmp + grid[i][j];
            }
        }
        return memo[n-1];
    }
};
```

343. Integer Break

Given a positive integer *n*, break it into the sum of **at least** two positive integers and maximize the product of those integers. Return the maximum product you can get.

For example, given *n* = 2, return 1 (2 = 1 + 1); given *n* = 10, return 36 (10 = 3 + 3 + 4).

Note: You may assume that *n* is not less than 2 and not larger than 58.

```
class Solution {
public:
    int integerBreak(int n) {
        vector<int> memo(n+1, -1);
        if(n == 2) return 1;
        for(int i = 3; i <= n; i++){
            for(int j = 2; j < i; j++){
                memo[i] = max(max(memo[i], j * (i-j)), j * memo[i-j]);
            }
        }
        return memo[n];
    }
};
```

279. Perfect Squares

Given a positive integer *n*, find the least number of perfect square numbers (for example, `1, 4, 9, 16, ...`) which sum to *n*.

Example 1:

Input: *n* =

```
12
```

Output: 3
Explanation:

```
12 = 4 + 4 + 4.
```

Example 2:

Input: *n* =

```
13
```

Output: 2
Explanation:

```
13 = 4 + 9.
```

```
class Solution {
public:
    int numSquares(int n) {
        vector<int> memo(n+1, n);
        if(n == 1) return 1;
        memo[0] = 0;
        memo[1] = 1;
        for(int i = 2; i <=n; i++){
            for(int j = 1; j *j <= i; j++){
                memo[i] = min(memo[i], memo[i-j*j]+1);
            }
        }
        return memo[n];
    }
};
```

91. Decode Ways

A message containing letters from `A-Z` is being encoded to numbers using the following mapping:

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

Given a **non-empty** string containing only digits, determine the total number of ways to decode it.

Example 1:

Input: `"12"`
Output: `2`
Explanation: `It could be decoded as "AB" (1 2) or "L" (12).`

Example 2:

Input: `"226"`
Output: `3`
Explanation: `It could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).`

```
class Solution {
public:
    int numDecodings(string s) {
        int n = s.size();
        if(n == 0) return 0;
        vector<int> memo(n+1, -1);
        memo[0] = 1;
        memo[1] = s[0] == '0'? 0 : 1;
        for(int i = 2;i <= n; i++){
            if(s[i-1] == '0'){
                if(s[i-2] == '1'|| s[i-2] == '2'){
                    memo[i] = memo[i-2];
                }
                else{
                    memo[i] = 0;
                }
            }else{
                if(s[i-2] == '1'|| (s[i-2] == '2' && s[i-1] <= '6')){
                    memo[i] = memo[i-1] + memo[i-2];
                }
                else{
                    memo[i] = memo[i-1];
                }
            }
        }
        return memo[n];
    }
```

```
};
```

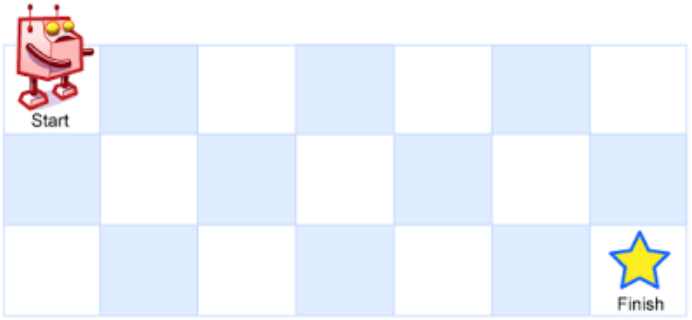
```
class Solution {
public:
    int numDecodings(string s) {
        int n = s.size();
        if(n == 0) return 0;
        vector<int> memo(n, -1);
        memo[0] = s[0] == '0'? 0 : 1;
        for(int i = 1;i < n; i++){
            if(s[i] == '0'){
                if(s[i-1] == '1'|| s[i-1] == '2'){
                    memo[i] = i-2 >= 0 ? memo[i-2] : 1;
                }
                else{
                    memo[i] = 0;
                }
            }else{
                if(s[i-1] == '1'|| (s[i-1] == '2' && s[i] <= '6')){
                    memo[i] = memo[i-1] + (i-2 >= 0 ? memo[i-2] : 1);
                }
                else{
                    memo[i] = memo[i-1];
                }
            }
        }
        return memo[n-1];
    }
};
```

62. 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 7 x 3 grid. How many possible unique paths are there?

Note: *m* and *n* will be at most 100.

Example 1:

Input: `m = 3, n = 2`
Output: `3`
Explanation:
`From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:`
`1. Right -> Right -> Down`
`2. Right -> Down -> Right`
`3. Down -> Right -> Right`

Example 2:

Input: `m = 7, n = 3`
Output: `28`

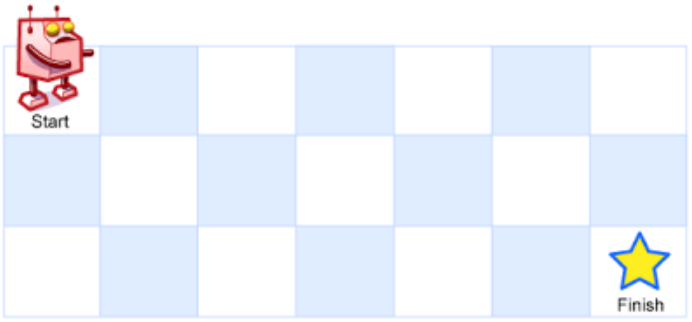
```
class Solution {
public:
    int uniquePaths(int m, int n) {
        if(m == 1 || n == 1) return 1;
        vector<vector<int> > memo(m, vector<int>(n));
        for(int i = 0; i < m; i++) memo[i][0] = 1;
        for(int j = 0; j < n; j++) memo[0][j] = 1;
        for(int i = 1; i < m; i++){
            for(int j = 1; j < n; j++){
                memo[i][j] = memo[i-1][j] + memo[i][j-1];
            }
        }
        return memo[m-1][n-1];
    }
};
```

63. Unique Paths II

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

Note: *m* and *n* will be at most 100.

Example 1:

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
```

```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size();
        int n = obstacleGrid[0].size();
        vector<vector<int> > memo(m, vector<int>(n));
        bool flag = true;
        for(int i = 0; i < m; i++){
            if(obstacleGrid[i][0]== 1) {
                memo[i][0] = 0;
                flag = false;
            }
```

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

213. House Robber II

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are **arranged in a circle.** That means the first house is the neighbor of the last one. Meanwhile, 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: `[2,3,2]`
Output: `3`
Explanation: `You cannot rob house 1 (money = 2) and then rob house 3 (money = 2),`
`because they are adjacent houses.`

Example 2:

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.`

空间复杂度O(n)

```
class Solution {
public:
    int robDiv(vector<int>& nums, int first, int last){
        int n = nums.size();
        vector<int> memo(n,0);
        memo[n-2] = nums[n-2];
        memo[n-1] = nums[n-1];
        int res= max(nums[n-1], nums[n-2]);
        for(int i = n-3; i >= first; i--){
            memo[i] = nums[i];
            res = max(res, memo[i]);
            for(int j = i + 2; j <=last; j++){
                int tmp = nums[i] + memo[j];
                memo[i] = max(tmp, memo[i]);;
                res = max(res, memo[i]);
            }
        }
        return res;
    }
    int rob(vector<int>& nums) {
        if(nums.size() == 0) return 0;
        if(nums.size() == 1) return nums[0];
        int n = nums.size();
        int res1 = robDiv(nums, 0, n-2);
        int res2 = robDiv(nums, 1, n-1);
        return max(res1, res2);
    }
};
```