2 Dynamic Programming

2.1 problem list

- 来源列表：LeetCode – 动态规划。

id	title	label
1	爬楼梯 easy	
2	买卖股票的最佳时机 easy	
3	使用最小花费爬楼梯 easy	
4	打家劫舍 easy	
5	区域和检索 – 数组不可变 easy	
6	不同路径 easy	
7	不同路径 II easy	
8	比特位计数 easy	
9	最小路径和 easy	

2.2 solution

2.2.1 p70 爬楼梯

动态规划

$O(N)$。

- 上 n 阶台阶的方案数 dp_i，可以等价：
  - 先上 1 阶，再上 $n-1$，即 dp_{i-1}。
  - 线上 2 阶，再上 $n-2$，即 dp_{i-2}。

```
class Solution {
public:
    int climbStairs(int n) {
        int *dp = new int[n + 1];
        dp[0] = dp[1] = 1;
        for (int i = 2; i <= n; i++) {
            dp[i] = dp[i - 2] + dp[i - 1];
        }
        return dp[n];
    }
};
```

2.2.2 p121 买卖股票的最佳时机

动态规划

$O(N)$。

- 考虑在第 i 天卖出。
  - 设 1 到 i 天的最低价格是 $cost_i$ 。
  - 则有利润为 $price_i - cost_i$。
- 对于 $cost_i$，对最低价格是否在第 i 天进行规划。
  - 在第 i 天，则为 $price_i$。
  - 不在，则在 1 到 $i-1$ 天，为 $cost_{i-1}$。
  - 即 $cost_i = min(price_i, cost_{i-1})$。

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int cost = INT_MAX, result = 0;
        for (auto price : prices) {
            cost = min(cost, price);
            result = max(result, price - cost);
        }
        return result;
    }
};
```

2.2.3 p746 使用最小花费爬楼梯

动态规划

$O(N)$。

设 $cost_i$ 为从第 i 层爬到顶的最小代价，同样考虑先走一步还是两步：

- 先走 1 步，$cost_{i+1}$。

- 先走 2 步，$cost_{i+2}$。

```
class Solution {
public:
    int minCostClimbingStairs(vector<int>& cost) {
        for (int i = cost.size() - 3; i >= 0; i--) {
            cost[i] += min(cost[i + 1], cost[i + 2]);
        }
        return min(cost[0], cost[1]);
    }
};
```

2.2.4 p198 大家劫舍

动态规划

相邻 3 间屋子必定至少偷一间。

所以这个问题等价于走楼梯问题，每次走 2 步或 3 步，问最大收益，做法同前面的题目。

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.size() <= 1) {
            return nums.empty() ? 0 : *nums.begin();
        }
        for (int i = nums.size(); i >= 0; i--) {
            int temp = 0;
            for (int j = 2; j <= 3; j++) {
                if (i + j < nums.size()) {
                    temp = max(temp, nums[i + j]);
                }
            }
            nums[i] += temp;
        }
        return max(nums[0], nums[1]);
    }
};
```

2.2.5 p303 区域和检索 - 数组不可变

动态规划

- **知识点**：$[l,r]$的和 = $[0,r]$ 的和 - $[0,l-1]$ 的和。
- 即区间和等于两个前缀和相减。
- $O(N)$ 预处理每一个前缀和。
- $O(1)$ 查询区间和。

```
class NumArray {
private:
    int *pre_sum, n;
public:
    NumArray(vector<int> nums) {
        if (nums.empty()) {
            return;
        }
        n = nums.size();
        pre_sum = new int[n];
        pre_sum[0] = nums[0];
        for (int i = 1; i < n; i++) {
            pre_sum[i] = pre_sum[i - 1] + nums[i];
        }
    }

    int sumRange(int i, int j) {
        return pre_sum[j] - (i == 0 ? 0 : pre_sum[i - 1]);
    }
};
```

2.2.6 p62 不同路径

动态规划

$O(NM)$。

设 $dp_{i,j}$ 为走到位置 $[i,j]$ 的方案数，则考虑上一步：

- $[i-1,j]$，则再之前的方案有 $dp_{i-1,j}$。
- $[i,j-1]$，则再之前的方案有 $dp_{i,j-1}$。

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

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

组合数

$O(N)$。

一共要往下走 $N-1$ 步，往右走 $M-1$ 。

等价于 $N+M-2$ 步中挑出 $N-1$ 步往下，即为组合数 C_{N+M-2}^{N-1}。

组合数直接乘会溢出，可以去一下自然对数，再转回来。

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        //C(n + m - 2, n - 1)
        long double result = 0.;
        for (int i = n; i < n + m - 1; i++) result += logl(i);
        for (int i = 1; i < m; i++) result -= logl(i);
        return int(expl(result) + 0.00000001);
    }
};
```

2.2.7 p63 不同路径2

动态规划

$O(NM)$ 。

- 同上一题，转移前判断一下是否是障碍物。

```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int n = obstacleGrid.size();
        if (n == 0) {
            return 0;
        }
        int m = obstacleGrid.begin()->size();
        if (m == 0) {
            return 0;
        }
        int ** dp = new int *[n];
        for (int i = 0; i < n; i++) {
            dp[i] = new int [m];
        }
        dp[0][0] = !obstacleGrid[0][0];
        for (int i = 0; i < n; i++) {
```

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

2.2.8 p338 比特位计数

动态规划

- 数字 i 的答案为 数字 i 去掉其二进制表示下最低位的 1 后的数字的答案 $+ 1$;
- **【知识点】**取一个数二进制下的最低位 1 ： $i\&-i$ 。

```
class Solution {
public:
    vector<int> countBits(int num) {
        vector<int> result(num + 1, 0);
        for (int i = 1; i <= num; i++) {
            result[i] = result[i - (i & -i)] + 1;
        }
        return result;
    }
};
```

2.2.9 p64 最小路径和

动态规划

$O(NM)$。

- 设 $dp_{i,j}$ 为走到 $[i,j]$ 的最小花费。
- 同样考虑最后一步是朝下的还是朝右的即可。

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        if (grid.empty() || grid.begin()->empty()) {
            return 0;
        }
        int n = grid.size(), m = grid.begin()->size();
        for (int i = 0; i < n; i++) {
```

```
            for (int j = !i; j < m; j++) {
                int temp = INT_MAX;
                if (i) temp = min(temp, grid[i - 1][j]);
                if (j) temp = min(temp, grid[i][j - 1]);
                grid[i][j] += temp;
            }
        }
        return grid[n - 1][m - 1];
    }
};
```