算法设计之动态规划

(1)70. 爬楼梯

Description

爬 n 阶楼梯，每次只能爬 1 或 2 个台阶。有多少种不同的方法可以爬到楼顶呢？

Solution

动态规划 O(n)

实质就是斐波那契数列

用 dp[i]表示到第i号台阶的方法数

注：没有0号台阶，是起点，dp[0]不是

到 1 号台阶方法有 dp[1] = 1

到 2 号台阶方法有 dp[2] = dp[1] + dp[0] // 从1号台阶上去或从起点上去

到 3 号台阶方法有 dp[2] = dp[2] + dp[1] // 从2号台阶上去或从1号台阶上去

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

class Solution {
public:
    int climbStairs(int n) {
        int dp[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];
    }
};

int main(){
    Solution solu;
    int n = 3;
    cout << solu.climbStairs(n);
    return 0;
}
```

---

(2)121. 买卖股票的最佳时机

Description

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。如果你最多只允许完成一笔交易（即买入和卖出一支股票），设计一个算法来计算你所能获取的最大利润。

Solution

遍历数组过程中
cost记录当前遍历到的最低的价格
result记录价格与cost最大差值
时间复杂度： O(n)

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int cost = INT_MAX, result = 0;
        for(int i=0; i<prices.size(); i++) {
            cost = min(cost, prices[i]);
            result = max(result, prices[i] - cost);
        }
        return result;
    }
};
```

```
int main(){
    Solution solu;
    vector<int> nums = {7,1,5,3,6,4};
    cout<<solu.maxProfit(nums);
    return 0;
}
```

---

(3)746. 使用最小花费爬楼梯

Description

数组的每个索引做为一个阶梯，第 i个阶梯对应着一个非负数的体力花费值 cost[i]。

你可以选择继续爬一个阶梯或者爬两个阶梯。在开始时，你可以选择从索引为 0 或 1 的元素作为初始阶梯。

找到达到楼层顶部的最低花费。

Solution

和第一题爬楼梯异曲同工

动态规划 O(n)

到达 0 号台阶最低花费 cost[0] = cost[0]

到达 1 号台阶最低花费 cost[1] = cost[1]

到达 2 号台阶最低花费 cost[2] = cost[2] + min(cost[0] , cost[1]) // 只能从 0 号或 1 号台阶出发

到达 3 号台阶最低花费 cost[3] = cost[3] + min(cost[1] , cost[2]) // 只能从 1 号或 2 号台阶出发

… …

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

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

```
int main(){
    Solution solu;
    vector<int> nums = {1, 100, 1, 1, 1, 100, 1, 1, 100, 1};
    cout<<solu.minCostClimbingStairs(nums);
    return 0;
}
```

---

(4)198. 打家劫舍

Description

每间房内都藏有一定的现金。可以偷窃多间房屋，但是要求被偷房屋不能相邻。
求能够偷窃到的最高金额。

Solution

动态规划

某种程度上偷窃到的最高金额就是要求所偷的房屋要尽可能的多
换句话说就是要求：隔 1家 或 2家 偷一次
所以这个问题等价于走楼梯花费最小问题，每次走2步或3步，问最大收益，

偷到 0 号房屋累积偷窃金额最高 nums[0] = nums[0]
偷到 1 号房屋累积偷窃金额最高 nums[1] = nums[1] // 0号房屋不能偷
偷到 2 号房屋累积偷窃金额最高 nums[2] = nums[2] + nums[0]
偷到 3 号房屋累积偷窃金额最高 nums[3] = nums[3] + max(nums[0], nums[1])
偷到 4 号房屋累积偷窃金额最高 nums[4] = nums[4] + max(nums[1], nums[2])
… …

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

class Solution {
public:
    int rob(vector<int>& nums) {
        int n = nums.size();
        if(n == 0) return  0;
        else if(n==1) return nums[0];
        else if(n==2) return max(nums[0], nums[1]);

        nums[2] += nums[0];
```

```
        for (int i=3; i<n; i++) {
            nums[i] += max(nums[i-3], nums[i-2]);
        }
        return max(nums[n-2], nums[n-1]);
    }
};

int main(){
    Solution solu;
    vector<int> nums = {2,7,9,3,1};
    cout << solu.rob(nums);
    return 0;
}
```

---

(5)303. 区域和检索 - 数组不可变

Description

给定一个整数数组 nums，求出数组从索引 i 到 j (i ≤ j) 范围内元素的总和，包含 i, j 两点。

Solution

很简单，就是和差 sum[i,j] = sum[0,j] - sum[0,i]
但是本题要学习指针的使用
通过new，使用指针创建数组

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

class NumArray {
private:
    int *pre_sum;

public:
    NumArray(vector<int> nums) {
        if (nums.empty())  return;

        int n = nums.size();
        pre_sum = new int[n+1];
        pre_sum[0] = 0;
        for (int i=0; i<n; i++) {
            pre_sum[i+1] = pre_sum[i] + nums[i];
```

```
        }
    }

    int sumRange(int i, int j) {
        return pre_sum[j+1]-pre_sum[i];
    }
};

int main() {
    NumArray numA({-2, 0, 3, -5, 2, -1});

    cout << numA.sumRange(0, 2) << endl;
    cout << numA.sumRange(2, 5) << endl;
    cout << numA.sumRange(0, 5) << endl;

    return 0;
}
```

---

(6)62. 不同路径

Description

一个 m x n 网格，一个机器人位于左上角，每次只能向下或者向右移动一步，机器人试图达到网格的右下角。

总共有多少条不同的路径？

Solution

因为只能向下或者向右移动

-> 网格上边界和左边界只有一种方法能到达。结论1： 到达(0,j)、(i,0)点方法数 dp[i,j] = 1

-> 要到达此外的某个点，只能通过该点左边或上边的格子出发。结论2：到达(i,j)点方法数 dp[i,j] = dp[i-1,j] + dp[i,j-1]

为了两个结论统一公式，我们使用n+1*m+1数组，上、左边界用0表示，起点变为（1,1）

其实这道题本质是数学中的概率统计，一共要往下走 n-1 步，往右走 m-1 步。等价于 (n-1)+(m-1) 步中挑出 n-1 步往下，即为组合数C(n-1)(n+m-1)

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

class Solution {
```

```
public:
    int uniquePaths(int m, int n) {
        if (m == 0 || n == 0) return 0;

        vector<vector<int>> dp(n+1,vector<int>(m+1,0));

        dp[1][1]=1; //在这里(1,1)表示起点
        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][m];
    }
};


int main() {
    Solution solu;
    cout << solu.uniquePaths(3,2);
    return 0;
}
```

---

(7)63. 不同路径 II

Description

一个 m x n 网格，一个机器人位于左上角，每次只能向下或者向右移动一步，机器人试图达到网格的右下角。

现在考虑网格中有障碍物。

总共有多少条不同的路径？

Solution

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
```

```
        if( obstacleGrid.empty() ) return 0;

        int n = obstacleGrid.size();
        int m = obstacleGrid[0].size();
        vector<vector<int>> dp(n+1,vector<int>(m+1,0));

        if( obstacleGrid[0][0]==0 ) dp[1][1]=1; //在这里(1,1)表示起点
        for(int i=1; i<=n; i++) {
            for(int j=1; j<=m; j++) {
                if(obstacleGrid[i-1][j-1]==0) {
                    dp[i][j] += (dp[i-1][j] + dp[i][j-1]);
                }
            }
        }

        return dp[n][m];
    }
};


int main() {
    Solution solu;
    vector<vector<int>> obstacleGrid = {{0,1,0},{0,1,0},{0,0,0}};
    cout << solu.uniquePathsWithObstacles(obstacleGrid);
    return 0;
}
```

---

(8)338. 比特位计数

Description

给定一个非负整数 num。对于 0 ≤ i ≤ num 范围中的每个数字 i ，计算其二进制数中的 1 的数目并将它们作为数组返回。例如，输入: 5，输出: [0,1,1,2,1,2]

Solution

方法一，利用如下规律，时间复杂度O(n)

```
i   二进制   1的数目   规律
0   0000      0
1   0001      1   = dp[0]+1

2   0010      1   = dp[0]+1
3   0011      2   = dp[1]+1
```

```
4  0100     1  = dp[0]+1
5  0101     2  = dp[1]+1
6  0110     2  = dp[2]+1
7  0111     3  = dp[3]+1

8  1000     1  = dp[0]+1
9  1001     2  = dp[1]+1
10  1010    2  = dp[2]+1
11  1011    3  = dp[3]+1
12  1100    2  = dp[4]+1
13  1101    3  = dp[5]+1
14  1110    3  = dp[6]+1
15  1111    4  = dp[7]+1
```

```
#include <vector>
#include <iostream>
using namespace std;

class Solution {
public:
    vector<int> countBits(int num) {
        vector<int> dp;
        dp.push_back(0);
        if(num==0) return dp;

        while(1) {
            int n = dp.size();
            for (int j=0; j<n; j++) {
                dp.push_back( dp[j]+1 );
                if(--num==0) return dp;
            }
        }
    }
};

void output(vector<int> nums) {
    for (auto e:nums) {
        cout << e << " ";
    }
}

int main() {
    Solution solu;
    output(solu.countBits(5));
    return 0;
}
```

方法二：还可以利用二进制的逻辑运算来解

(-i) 是数字i 二进制的 补码，即对i原码取反加1。例如 i=6=110，(-i) = 010

(i&-i) 数值为i的二进制数表示下最低位的 1 的权值。例如 i=6=110，(i&-i)=10=2
i-(i&-i) 是 数字 i 去掉其二进制表示下最低位的1后的数字

```
#include <vector>
#include <iostream>
using namespace std;

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


void output(vector<int> nums) {
    for (auto e:nums) {
        cout << e << " ";
    }
}

int main() {
    Solution solu;
    output(solu.countBits(15));
    return 0;
}
```

(9)64. 最小路径和

Description

给定一个包含非负整数的 m x n 网格，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。说明：每次只能向下或者向右移动一步。

Solution

结合题目不同路径、不同路径II
从左上角到右下角遍历
计算公式：grid[i][j] += min(grid[i-1][j], grid[i][j-1]);
不断更新每个方格数字和

```cpp
#include <vector>
#include <iostream>
#include <algorithm>
#include <limits>
using namespace std;

class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        if( grid.empty() ) return 0;

        int n = grid.size();
        int m = grid[0].size();

        for(int i=1; i<n; i++)  grid[i][0] += grid[i-1][0];
        for(int j=1; j<m; j++)  grid[0][j] += grid[0][j-1];

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


int main() {
    Solution solu;
    vector<vector<int>> grid = {{1,3,1},{1,5,1},{4,2,1}};
    cout<<solu.minPathSum(grid);
    return 0;
}
```