面试中LeetCode常见算法整理——动态规划

原创 God_Leek 2019-10-10 09:22:54 160 收藏 版权

分类专栏： LeetCode刷题之路 算法研讨之路 文章标签： LeetCode 动态规划

面试中LeetCode常见算法整理——动态规划

1. 斐波那契数列

（1）爬楼梯

70. Climbing Stairs

定义一个数组 dp 存储上楼梯的方法数（为了方便讨论，数组下标从 1 开始），dp[i] 表示走到第 i 个楼梯的方法数目。第 i 个楼梯可以从第 i-1 和 i-2 个楼梯再走一步到达，走到第 i 个楼梯的方法数为走到第 i-1 和第 i-2 个楼梯的方法数
之和。

$$dp[i] = dp[i-1] + dp[i-2]$$

考虑到 dp[i] 只与 dp[i - 1] 和 dp[i - 2] 有关，因此可以只用两个变量来存储 dp[i - 1] 和 dp[i - 2]，使得原来的 O(N) 空间复杂度优化为 O(1) 复杂度。

```
1  class Solution {
2  public:
3      int climbStairs(int n) {
4          vector<int> res(n+1, -1);
5          res[0] = 1;
6          res[1] = 1;
7          for (int i = 2; i <= n; ++i)
8              res[i] = res[i - 1] + res[i - 2];
9          return res[n];
10     }
11 };
12
13 class Solution2
14 public:
15     int climbSta  'int n) {
16         int dp_  ;
17         int dp_2 = 1;
18         if (n =
19             retu  dp_2;
20         if (n =
21             ret  p_1;
22         for (int i = 2; i <= n; ++i)
```

```
            int temp = dp_1;25
            dp_2 = temp;
        }
        return dp_1;
    }
};
```

```
            int dp_1 = dp_1 + dp_2;
```

（2）强盗抢劫

198. House Robber

定义 dp 数组用来存储最大的抢劫量，其中 dp[i] 表示抢到第 i 个住户时的最大抢劫量。由于不能抢劫邻近住户，如果抢劫了第 i -1 个住户，那么就不能再抢劫第 i 个住户，所以

$$dp[i] = max(dp[i-2] + nums[i], dp[i-1])$$

```
class Solution {
public:
    int rob(vector<int>& nums) {
        int dp_1 = 0; //表示上一次抢劫的
        int dp_2 = 0; //表示上上次抢劫的
        for (int i = 0; i < nums.size(); ++i)
        {
            int temp = dp_1;
            dp_1 = max(dp_2 + nums[i], dp_1); //当前抢劫的是上一次抢劫的与上上次抢劫的加上本次抢劫的最大值
            dp_2 = temp;
        }
        return dp_1;
    }
};
```

（3）强盗在环形区域抢劫

213. House Robber II

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

```
13      {
14          int dp_1 = 0;//前一个
15          int dp_2 = 0;//前前一个
16          int res = 0;
17          for (int i = l; i <= r; ++i)
18          {
19              int temp = dp_1;
20              dp_1 = max(dp_1, dp_2 + nums[i]);
21              dp_2 = temp;
22          }
23          return dp_1;
24      }
25  };
```

（4）信件错排

题目描述：
有 N 个信和信封，它们被打乱（也就是每封信都不在正确的信封中），求错误装信方式的数量。

解题思路：
定义一个数组 dp 存储错误方式数量，dp[i] 表示前 i 个信和信封的错误方式数量。假设第 i 个信装到第 j 个信封里面，而第 j 个信装到第 k 个信封里面。根据 i 和 k 是否相等，有两种情况：
i==k，交换 i 和 k 的信后，它们的信和信封在正确的位置，但是其余 i-2 封信有 dp[i-2] 种错误装信的方式。由于 j 有 i-1 种取值，因此共有 (i-1)*dp[i-2] 种错误装信方式。
i != k，交换 i 和 j 的信后，第 i 个信和信封在正确的位置，其余 i-1 封信有 dp[i-1] 种错误装信方式。由于 j 有 i-1种取值，因此共有 (i-1)*dp[i-1] 种错误装信方式。

综上所述，错误装信数量方式数量为：

$$dp[i] = (i-1) * dp[i-2] + (i-1) * dp[i-1]$$

这个思路好像更好理解：

当n个编号元素放在n个编号位置，元素编号与位置编号各不对应的方法数用dp[n]表示，那么dp[n-1]就表示n-1个编号元素放在n-1个编号位置，各不对应的方法数，其它类推.
　　第一步，把第n个元素放在一个位置，比如位置k，一共有n-1种方法；
　　第二步，放编号为k的元素，这时有两种情况：(1)把它放到位置n，那么，对于剩下的n-1个元素，由于第k个元素放到了位置n，剩下n-1个元素就有dp[n-2]种方法；(2)第k个元素不把它放到位置n，这时，对于这n-1个元素，有dp[n-1]种方法。

```
1   int lettersIncorrectlyArranged(int n)
2   {
3       vector<int> dp(n+1, 0);
4       dp[1] = 0;
5       dp[2] = 1;
6       for (int i = 3; i <= n; ++i)
7           dp[i] = (i - 1)*(dp[i-2] + dp[i-1]);
```

```
8       return dp[n];
9   }
```

（5）母牛生产

题目描述：
假设农场中成熟的母牛每年都生 1 头小母牛，并且永远不会死。第一年有 1 只小母牛，从第二年开始，母牛开始生小母牛。每只小母牛 3 年之后成熟又可以生小母牛。给定整数 N，求 N 年后牛的数量。

根据题目易得第i年的牛数量为

$$dp[i] = dp[i-1] + dp[i-3]$$

```
int cowProduction(int n)
{
    vector<int> count(n+1, 0);
    count[1] = 1;
    count[2] = 2;
    count[3] = 3;
    for (int i = 4; i <= n; ++i)
        count[i] = count[i - 1] + count[i - 3];
    return count[n];
}
```

2. 矩阵路径

（1）矩阵的最小路径和

64. Minimum Path Sum

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        if (grid.size() == 0)
            return 0;
        int rows = grid.size();
        int col  id[0].size();
        for(int  0;i<rows;++i)
            for  j = 0; j < cols; ++j)
            {
                if (i == 0 && j == 0)
                grid[i][j] = grid[i][j];
                else if (i == 0)
                grid[i][j] = grid[i][j] + grid[i][j - 1];
                if (j == 0)
                grid[i][j] = grid[i][j] + grid[i - 1][j];
                else
```

```
                grid[i][j] = grid[i][j] + min(grid[i][j-1], grid[i-1][j]);
        return grid[rows - 1][cols - 1];
    }
};
```

（2）矩阵的总路径数

62. Unique Paths

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

3. 数组区间

（1）子数组最大和

53. Maximum Subarray

一般解法

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int sum = INT_MIN;
        int cur
        for (int
        {
            cur
            sum = max(sum, curSum);
            if
                cur
        }
        return
    }
```

DP解法

DP[i]表示i之前并包含nums[i]的子数组的最大和，无非两种情况：DP[i] = DP[I-1] + nums[i]或者DP[i] = nums[i]，因此dp[i] = max(dp[i-1] + nums[i], nums[i])，取DP[i]的最大值即可。

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int n = nums.size();
        if (n == 0)
            return 0;
        int dp_i = nums[0];
        int res = dp_i;
        for (int i = 1; i < n; ++i)
        {
            int dp_pre = dp_i;
            dp_i = max(dp_pre + nums[i], nums[i]);
            res = max(res, dp_i);
        }
        return res;
    }
};
```

（2）数组中等差递增子区间的个数

413. Arithmetic Slices

dp[i] 表示以 A[i] 为结尾的等差递增子区间的个数。
在 A[i] - A[i - 1] == A[i - 1] - A[i - 2] 的条件下，{A[i - 2], A[i - 1], A[i]} 是一个等差递增子区间。如果 {A[i - 3], A[i - 2], A[i - 1]} 是一个等差递增子区间，那么 {A[i - 3], A[i - 2], A[i - 1], A[i]} 也是等差递增子区间，dp[i] = dp[i-1] + 1。

```
class Solution {
public:
    int numberOfArithmeticSlices(vector<int>& A) {
        int n = A.size();
        vector<int> dp(n, 0);
        //dp[i]表示以A[i]结尾的
        for (in  2; i < n; ++i)
        {
            if ( - A[i - 1] == A[i - 1] - A[i - 2])
                 = dp[i - 1] + 1;
        }
        return  late(dp.begin(), dp.end(), 0);
    }
};
```

（1）分割整数的最大乘积

343. Integer Break

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

（2）按平方数来分割整数

279. Perfect Squares

```
class Solution {
public:
    int numSquares(int n) {
        vector<int> dp(n+1, INT_MAX);
        dp[0] = 0; //注意初始值问题，要么dp全部初始化为INT_MAX，要么将dp[0]初始化为0
        int cnt = 1;
        for (int i = 1; i <= n; ++i)
        {
            if (cnt* cnt == i)
            {
                dp[i] = 1;
                ++cnt;
            }
            else
            {
                (int j = 1; j < cnt; ++j)
                dp[i] = min(dp[i], dp[i - j * j] + 1);
            }

        }
        return ;
    }
};
```


组成字母字符串

91. Decode Ways

```cpp
class Solution {
public:
    int numDecodings(string s) {
        if (s.empty() || s[0] == '0')
            return 0;
        //dp[i]表示0~i的numdecodings
        vector<int> dp(s.length()+1, 0);
        dp[0] = 1; //注意：一般这种情况都是需要特殊验证一下以确定dp[0]的初值
        dp[1] = 1; //第0个decoding
        for (int i = 2; i <= s.length(); ++i)
        {
            //第i-1个decoding
            //第1个decoding
            //判断当前是否为0，如果是0，则表示当前位不可以单独拿出来，dp[i] = 0;
            //如果不是0，则表示当前位可以单独拿出来，dp[i] = dp[i-1]
            dp[i] = s[i-1] == '0' ? 0 : dp[i - 1]; //注意下标问题，i-1表示当前位置
            //判断与前一位是否匹配
            if (s[i - 2] == '1' || (s[i - 2] == '2' && s[i - 1] <= '6'))
                dp[i] += dp[i - 2];
        }
        return dp[s.length()];
    }
};

//由于dp[i]只与dp[i-1]和dp[i-2]有关，所以可以将空间复杂度降为O(1)
class Solution {
public:
    int numDecodings(string s) {
        int n = s.length();
        if (n == 0 || s[0] == '0')
            return 0;
        int dp_i = 0;
        int dp_i_1 = 1; //dp[i-1]
        int dp_i_2 = 1; //dp[i-2]
        if (n == 1)
            return dp_i_1;
        for (int i = 1; i < n; ++i)
        {
            dp_i = s[i] == '0' ? 0 : dp_i_1;
            if (s[i - 1] == '1' || (s[i - 1] == '2' && s[i] <= '6'))
                dp_i += dp_i_2;
            dp_i_2 = dp_i_1;
            dp_i_1 = dp_i;
        }
        return dp_i;
    }
};
```

5. 最长递增子序列

（1）最长递增子序列

300. Longest Increasing Subsequence

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        int n = nums.size();
        if (n < 2)
            return n;
        int res = 1;
        vector<int> dp(n, 1); //dp[i]表示包含i之前的最长子串长度
        //关于什么时候用dp(n+1)什么时候用dp(n)取决于dp[i]的值和前几个值的相关度
        for (int i = 1; i < n; ++i)
        {
            for (int j = 0; j < i; ++j)
                if (nums[j] < nums[i])
                    dp[i] = max(dp[i], dp[j] + 1);
            res = max(res, dp[i]);
        }
        return res;
    }
};
```

（2）一组整数对能够构成的最长链

646. Maximum Length of Pair Chain

这个题跟上面的题思路类似，不过要先对整数对进行排序。

```
 class Solution {
public:
    int findLongestChain(vector<vector<int>>& pairs) {
        int n = pairs.size();
        if (n <2)
            ret ;
        vector< dp(n, 1); //dp[i]表示包含i的最长串长度
        int res
        sort(pa egin(), pairs.end(), cmp);
        for (int i = 1; i < n; ++i)
        {
            for  j = 0; j < i; ++j)
                pairs[i][0] > pairs[j][1])
                    dp[i] = max(dp[i], dp[j] + 1);
            res = max(res, dp[i]);
```

```
        return res;18      }
19
20  private:
21      static bool cmp(vector<int>& lhs, vector<int>& rhs)
22      {
23          if (lhs[1] == rhs[1])
24              return lhs[0] < rhs[0];
25          return lhs[1] < rhs[1];
26      }
27  };
28
```

（3）最长摆动子序列

376. Wiggle Subsequence

登录后复制

```
1   class Solution {
2   public:
3       int wiggleMaxLength(vector<int>& nums) {
4           int up = 1;
5           int down = 1;
6           int n = nums.size();
7           if (n < 2)
8               return n;
9           for (int i = 1; i < n; ++i)
10          {
11              if (nums[i] > nums[i - 1])
12                  up = down + 1;
13              else if (nums[i] < nums[i - 1])
14                  down = up + 1;
15          }
16          return max(up, down);
17      }
18  };
```

7. 0-1背包

8. 股票交易

举报