October 11

Basic Recursion Problems

Problem 1: Reverse a Linked List using Recursion

Problem Statement:

- Given the head of a singly linked list, reverse the list, and return the reversed list.

Link to problem:

https://leetcode.com/problems/reverse-linked-list/description/

Example 1:

- Input: head = [1,2,3,4,5]
- Output: [5,4,3,2,1]

Example 2:

- Input: head = [1,2]
- Output: [2,1]

Solution:

```
class Solution {
    public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }

        ListNode newHead = reverseList(head.next);  // Recursively reverse the rest
        head.next.next = head;  // Point the next node back to the current node
        head.next = null;  // Break the original next link

        return newHead;
    }
}
```

Explanation:

- Recursively reach the last node, and while returning back, reverse the pointers one by one.
- At each level of recursion, we set head.next.next = head to reverse the link, and finally return the new head of the reversed list.

Time Complexity:

- Each node is visited once, so the time complexity is **O(n)**, where n is the number of nodes

Space Complexity:

- The recursion adds one stack frame for each node, so the space complexity is O(n)**.**

Problem 2: Climbing Stairs

Problem Statement:

You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Link to problem:

https://leetcode.com/problems/climbing-stairs/

Example 1:

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

Example 2:

Input: n = 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

Solution:

```
class Solution {
   public int climbStairs(int n)
{
          int[] res = new int[n+1];  // Array to store Fibonacci numbers up to n
          res[0] = 1;  // Base case: res[0] = 1
          res[1] = 1;  // Base case: res[1] = 1
```

```
        for(int i = 2; i <= n; i++) {
            res[i] = res[i-1] + res[i-2];  // Fibonacci relation
}
}
```

Explanation:

- An array res of size n+1 is initialized to store the Fibonacci numbers up to n.
- The base cases are handled:
  - res[0] = 1: For n = 0, there's 1 way (no steps at all).
  - res[1] = 1: For n = 1, there's 1 way (taking 1 step).
- Then, starting from i = 2, we calculate each value of res[i] by summing the previous two values (res[i-1] + res[i-2]).
- This process continues until we fill up the array for all values up to n.

After the loop completes, the array res contains the Fibonacci sequence (or similar series) from res[0] to res[n].

Edge Cases:

- If the array is empty or contains only one price, no transactions can be made, so return `0`.
- If prices are continuously decreasing, the maximum profit is `0` because there is no day to sell the stock at a higher price than the buying price.

Time Complexity:

The loop runs from i = 2 to n, so the time complexity is **O(n)**.

Space Complexity:

The array res is of size n+1, so the space complexity is **O(n)**.

Problem 3: Power of Three

Problem Statement:

You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Link to problem:

https://leetcode.com/problems/power-of-three/description/

Example 1:

Input: n = 27
Output: true
Explanation: 27 = 3^3

Example 2:

Input: n = 0
Output: false
Explanation: There is no x where $3^x = 0$.

Solution:

```
class Solution {
    public boolean isPowerOfThree(int n) {
        if (n < 1) {
            return false;
        }
        if (n == 1) {
            return true;
        }
        return n % 3 == 0 && isPowerOfThree(n / 3);
    }
}
```

Explanation:

- The recursive approach keeps dividing n by 3. If n eventually becomes 1, it means that it is a power of 3.
- If n is less than 1 or not divisible by 3 at any step, return false.

Time Complexity:

Each recursive step reduces n by dividing it by 3, so the time complexity is **$O(\log_3(n))$**.

Space Complexity:

The depth of recursion is proportional to the number of times n can be divided by 3, so the space complexity is **$O(\log_3(n))$**.
