LeetCode题目精选

1. 两数之和

链接：https://leetcode-cn.com/problems/two-sum/

给定一个整数数组 nums 和一个目标值 target，请你在该数组中找出和为目标值的那 两个 整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。

```
给定 nums = [2, 7, 11, 15], target = 9

因为 nums[0] + nums[1] = 2 + 7 = 9
所以返回 [0, 1]
```

题解：

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];
            if (map.containsKey(complement)) {
                return new int[] { map.get(complement), i };
            }
            map.put(nums[i], i);
        }
        throw new IllegalArgumentException("No two sum solution");
    }
}
```

2. 爬楼梯

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

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢?

注意：给定 n 是一个正整数。

示例 1：

```
输入： 2
输出： 2
解释： 有两种方法可以爬到楼顶。
1. 1 阶 + 1 阶
2. 2 阶
```

示例 2：

```
输入： 3
输出： 3
解释： 有三种方法可以爬到楼顶。
1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶
```

题解：

```
public class Solution {
    public int climbStairs(int n) {
        if (n == 1) {
            return 1;
        }
        int[] dp = new int[n + 1];
        dp[1] = 1;
        dp[2] = 2;
        for (int i = 3; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
        return dp[n];
    }
}
```

3. 翻转二叉树

链接：https://leetcode-cn.com/problems/invert-binary-tree/

翻转一棵二叉树。

示例：

输入：

```
     4
   /   \
  2     7
 / \   / \
1   3 6   9
```

输出：

```
     4
   /   \
  7     2
 / \   / \
9   6 3   1
```

题解：

```
public TreeNode invertTree(TreeNode root) {
    if (root == null) {
        return null;
    }
    TreeNode right = invertTree(root.right);
    TreeNode left = invertTree(root.left);
    root.left = right;
    root.right = left;
    return root;
}
```

4. 反转链表

链接：https://leetcode-cn.com/problems/reverse-linked-list/

反转一个单链表。

示例:

```
输入: 1->2->3->4->5->NULL
输出: 5->4->3->2->1->NULL
```

题解：

```
public ListNode reverseList(ListNode head) {
    ListNode prev = null;
    ListNode curr = head;
    while (curr != null) {
        ListNode nextTemp = curr.next;
        curr.next = prev;
        prev = curr;
        curr = nextTemp;
    }
    return prev;
}
```

5. LRU缓存机制

链接：https://leetcode-cn.com/problems/lru-cache/

运用你所掌握的数据结构，设计和实现一个 LRU (最近最少使用) 缓存机制。它应该支持以下操作： 获取数据 get 和 写入数据 put 。

获取数据 get(key) - 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1。 写入数据 put(key, value) - 如果密钥不存在，则写入其数据值。当缓存容量达到上限时，它应该在写入新数据之前删除最近最少使用的数据值，从而为新的数据值留出空间。

进阶:

你是否可以在 O(1) 时间复杂度内完成这两种操作?

示例:

```
LRUCache cache = new LRUCache( 2 /* 缓存容量 */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // 返回  1
cache.put(3, 3);    // 该操作会使得密钥 2 作废
cache.get(2);       // 返回 -1 (未找到)
cache.put(4, 4);    // 该操作会使得密钥 1 作废
cache.get(1);       // 返回 -1 (未找到)
cache.get(3);       // 返回  3
cache.get(4);       // 返回  4
```

题解:

```
class LRUCache extends LinkedHashMap<Integer, Integer>{
    private int capacity;

    public LRUCache(int capacity) {
        super(capacity, 0.75F, true);
        this.capacity = capacity;
    }

    public int get(int key) {
        return super.getOrDefault(key, -1);
    }

    public void put(int key, int value) {
        super.put(key, value);
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry<Integer, Integer> eldest) {
        return size() > capacity;
    }
}

/**
 * LRUCache 对象会以如下语句构造和调用:
 * LRUCache obj = new LRUCache(capacity);
 * int param_1 = obj.get(key);
 * obj.put(key,value);
 */
```

6. 最长回文子串

链接: https://leetcode-cn.com/problems/longest-palindromic-substring/

给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

示例 1:

```
输入: "babad"
输出: "bab"
注意: "aba" 也是一个有效答案。
```

示例 2:

```
输入: "cbbd"
输出: "bb"
```

题解:

```
public String longestPalindrome(String s) {
    if (s == null || s.length() < 1) return "";
    int start = 0, end = 0;
    for (int i = 0; i < s.length(); i++) {
        int len1 = expandAroundCenter(s, i, i);
        int len2 = expandAroundCenter(s, i, i + 1);
        int len = Math.max(len1, len2);
        if (len > end - start) {
            start = i - (len - 1) / 2;
            end = i + len / 2;
        }
    }
    return s.substring(start, end + 1);
}

private int expandAroundCenter(String s, int left, int right) {
    int L = left, R = right;
    while (L >= 0 && R < s.length() && s.charAt(L) == s.charAt(R)) {
        L--;
        R++;
    }
    return R - L - 1;
}
```

7. 有效的括号

链接：https://leetcode-cn.com/problems/valid-parentheses/

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串，判断字符串是否有效。

有效字符串需满足： 1. 左括号必须用相同类型的右括号闭合。 2. 左括号必须以正确的顺序闭合。

注意空字符串可被认为是有效字符串。

示例 1:

```
输入: "()"
输出: true
```

示例 2:

```
输入: "()[]{}"
输出: true
```

示例 3:

```
输入: "(]"
输出: false
```

示例 4:

```
输入: "([)]"
输出: false
```

示例 5:

```
输入: "{[]}"
输出: true
```

题解:

```java
class Solution {

  // Hash table that takes care of the mappings.
  private HashMap<Character, Character> mappings;

  // Initialize hash map with mappings. This simply makes the code easier to read.
  public Solution() {
    this.mappings = new HashMap<Character, Character>();
    this.mappings.put(')', '(');
    this.mappings.put('}', '{');
    this.mappings.put(']', '[');
  }

  public boolean isValid(String s) {

    // Initialize a stack to be used in the algorithm.
    Stack<Character> stack = new Stack<Character>();

    for (int i = 0; i < s.length(); i++) {
      char c = s.charAt(i);

      // If the current character is a closing bracket.
      if (this.mappings.containsKey(c)) {

        // Get the top element of the stack. If the stack is empty, set a dummy value of '#'
        char topElement = stack.empty() ? '#' : stack.pop();


        // If the mapping for this bracket doesn't match the stack's top element, return false.
```

```
        if (topElement != this.mappings.get(c)) {
          return false;
        }
      } else {
        // If it was an opening bracket, push to the stack.
        stack.push(c);
      }
    }

    // If the stack still contains elements, then it is an invalid expression.
    return stack.isEmpty();
  }
}
```

8. 数组中的第K个最大元素

链接：https://leetcode-cn.com/problems/kth-largest-element-in-an-array/

在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

示例 1:

```
输入: [3,2,1,5,6,4] 和 k = 2
输出: 5
```

示例 2:

```
输入: [3,2,3,1,2,4,5,5,6] 和 k = 4
输出: 4
```

说明:

你可以假设 k 总是有效的，且 1 ≤ k ≤ 数组的长度。

题解：

```
import java.util.Random;
class Solution {
  int [] nums;

  public void swap(int a, int b) {
    int tmp = this.nums[a];
    this.nums[a] = this.nums[b];
    this.nums[b] = tmp;
  }


  public int partition(int left, int right, int pivot_index) {
    int pivot = this.nums[pivot_index];
    // 1. move pivot to end
    swap(pivot_index, right);
```

```
    int store_index = left;

    // 2. move all smaller elements to the left
    for (int i = left; i <= right; i++) {
      if (this.nums[i] < pivot) {
        swap(store_index, i);
        store_index++;
      }
    }

    // 3. move pivot to its final place
    swap(store_index, right);

    return store_index;
  }

  public int quickselect(int left, int right, int k_smallest) {
    /*
    Returns the k-th smallest element of list within left..right.
    */

    if (left == right) // If the list contains only one element,
      return this.nums[left];  // return that element

    // select a random pivot_index
    Random random_num = new Random();
    int pivot_index = left + random_num.nextInt(right - left);

    pivot_index = partition(left, right, pivot_index);

    // the pivot is on (N - k)th smallest position
    if (k_smallest == pivot_index)
      return this.nums[k_smallest];
    // go left side
    else if (k_smallest < pivot_index)
      return quickselect(left, pivot_index - 1, k_smallest);
    // go right side
    return quickselect(pivot_index + 1, right, k_smallest);
  }

  public int findKthLargest(int[] nums, int k) {
    this.nums = nums;
    int size = nums.length;
    // kth largest is (N - k)th smallest
    return quickselect(0, size - 1, size - k);
  }
}
```

9. 实现 Trie (前缀树)

实现一个 Trie (前缀树)，包含 insert, search, 和 startsWith 这三个操作。

示例:

```
Trie trie = new Trie();

trie.insert("apple");
trie.search("apple");   // 返回 true
trie.search("app");     // 返回 false
trie.startsWith("app"); // 返回 true
trie.insert("app");
trie.search("app");     // 返回 true
```

说明:

- 你可以假设所有的输入都是由小写字母 a-z 构成的。
- 保证所有输入均为非空字符串。

题解:

```
class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    // Inserts a word into the trie.
    public void insert(String word) {
        TrieNode node = root;
        for (int i = 0; i < word.length(); i++) {
            char currentChar = word.charAt(i);
            if (!node.containsKey(currentChar)) {
                node.put(currentChar, new TrieNode());
            }
            node = node.get(currentChar);
        }
        node.setEnd();
    }

    // search a prefix or whole key in trie and
    // returns the node where search ends
    private TrieNode searchPrefix(String word) {
        TrieNode node = root;
        for (int i = 0; i < word.length(); i++) {
           char curLetter = word.charAt(i);
           if (node.containsKey(curLetter)) {
                node = node.get(curLetter);
           } else {
                return null;
           }
        }
        return node;
    }


    // Returns if the word is in the trie.
```

```
    public boolean search(String word) {
        TrieNode node = searchPrefix(word);
        return node != null && node.isEnd();
    }
}
```

10. 编辑距离

链接：https://leetcode-cn.com/problems/edit-distance/

给定两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作： 1. 插入一个字符 2. 删除一个字符 3. 替换一个字符

示例 1:

```
输入: word1 = "horse", word2 = "ros"
输出: 3
解释:
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')
```

示例 2:

```
输入: word1 = "intention", word2 = "execution"
输出: 5
解释:
intention -> inention (删除 't')
inention -> enention (将 'i' 替换为 'e')
enention -> exention (将 'n' 替换为 'x')
exention -> exection (将 'n' 替换为 'c')
exection -> execution (插入 'u')
```

题解：

```
class Solution {
  public int minDistance(String word1, String word2) {
    int n = word1.length();
    int m = word2.length();

    // if one of the strings is empty
    if (n * m == 0)
      return n + m;

    // array to store the convertion history
    int [][] d = new int[n + 1][m + 1];

    // init boundaries
    for (int i = 0; i < n + 1; i++) {

      d[i][0] = i;
```

```
    }
    for (int j = 0; j < m + 1; j++) {
      d[0][j] = j;
    }

    // DP compute
    for (int i = 1; i < n + 1; i++) {
      for (int j = 1; j < m + 1; j++) {
        int left = d[i - 1][j] + 1;
        int down = d[i][j - 1] + 1;
        int left_down = d[i - 1][j - 1];
        if (word1.charAt(i - 1) != word2.charAt(j - 1))
          left_down += 1;
        d[i][j] = Math.min(left, Math.min(down, left_down));

      }
    }
    return d[n][m];
  }
}
```