Logic Questions

Question 1: ValidParathesis.java

Question 2: InvertBinaryTree.java

Question 3: TwoSum.java

tests: Main.java (Entry point)

and some other classes.

All code in the src folder

Question 1

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

Open brackets must be closed by the same type of brackets.
Open brackets must be closed in the correct order.
Note that an empty string is also considered valid.

Question 1 as example:

Input: "()"
Output: *true*

Question 1a:
Input: "()[]{}"
Output: true

Question 1b:
Input: "(]"
Output: false

Question 1c:
Input: "([)]"
Output: false

Question 1d:
Input: "{[]}"
Output: true

pseudocode:

use a stack, when encounter open parenthesis, push to stack, when encounter close parenthesis, pop from stack.

If the popped parenthesis is different type from encountered the close parenthesis, return False

At the end the program return True as the above loop does not return False.

Run time: O(n), space complexity: O(n), when n is the length of the input string.

Run time is O(n) because we only need to loop trough the string once. Space is O(n) as we need store previously visited parenthesis, and in the worst case it is O(n)

```
def is_valid(s):
        stack = []
        for char in s:
                if char is open_parathesis:
                        stack.push(char)
                else if char is close_parathesis:
                        head = stack.pop()
                        if same_type(char, head):
                                continue
                        else:
                               return false

        return true
```

Question 2

Invert a binary tree.

Input:

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

Output:

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

pseudocode:

use recursion:

```
def invert(root):
        if root == NULL:
                return
        else:
            temp = root.left
            root.left = root.right
            root.right = temp
            invert(root.left)         // recursion call
            invert(root.right)        // recursion call
```

run time: O(n), as for each node, as we visit each node exactly once. N is the number of nodes in the tree.

space complexity: O(n). Because when we use recursion, for each function call, we will add one entry in the call stack. In this case we will make exactly n function calls, and not until we reach the leaf node than we can pop stuff from the call stack. In the worst case, when tree is extremely unbalanced, we have n entries in the call stack. So the space complexity is exactly O(n), same as run time.

Question 3

Given an array of integers, return indices of the two numbers such that they add up to a specific target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

Output:

pseudocode

loop through array, use a dictionary to record the previous integers and their indices

key: previous saw numbers
value: the indices for previous saw values

If target – current_num in dictionary, then we have found the indices we need

```
def found_indices(array, target):
        if array == None or len(array) == 0:
                raise Exception(“Not valid input”)

        d = {}  # hashmap in java, dictionary in python
        for idx, num in enumerate(array):        # idx is the indices, num is the val
                if target – num in d:
                    return d.get(target-num), idx

                d[num] = idx        # record current value and index into the array

        raise Exception(“No soln found, should have exactly one soln”)
```

run time: O(n), we are visiting each number exactly once.

Space complexity: O(n), we are storing every number and their indices.

test cases:

Input: [], target = 5
Output: *exception*

Input: [1, 2], target = 5
Output: exception

Input: [1,5,6], target = 6
Output: 0,1

Input: [-1, 1, 2, 0], target = 0
Output: 0,1

Input: [1,-1, 5, 6, 8], target = 9
Output: 0,4