**Homework 2 Report: Stack-Based Solutions**

**Submitted by: Sriniketh Shankar (121113580)**

**Problem 1: Valid Parentheses Using Stack**

**Approach:**

The problem requires checking whether a given string consisting of parentheses is valid based on the correct ordering and matching of open and close brackets. The approach follows:

1. Utilize a stack to store open brackets.
2. Use a mapping dictionary to associate closing brackets with their respective open brackets.
3. Traverse the string:
   * If it's an open bracket, push it onto the stack.
   * If it's a closing bracket, pop the last open bracket from the stack and check if it matches.
   * If at any point the matching fails, return False.
4. At the end, if the stack is empty, return True, otherwise False.

**Complexity Analysis:**

* **Time Complexity:** O(n), where n is the length of the string. Each character is processed once.
* **Space Complexity:** O(n) in the worst case, when all characters are open brackets.

**Problem 2: Evaluating Postfix Expressions Using Stack**

**Approach:**

Postfix expressions are evaluated using a stack-based approach:

1. Split the input string into tokens.
2. Traverse the tokens:
   * If a number is encountered, push it onto the stack.
   * If an operator is encountered, pop the top two values from the stack and apply the operator.
   * Push the result back onto the stack.
3. At the end, the stack contains a single element, which is the final result.

**Complexity Analysis:**

* **Time Complexity:** O(n), where n is the number of tokens in the postfix expression.
* **Space Complexity:** O(n), as we may store all numbers in the stack before evaluating.

**Usefulness of Postfix Expressions and Conversion from Infix to Postfix**

**Usefulness of Postfix Expressions:**

Postfix notation is beneficial in several ways:

1. No Need for Parentheses: Unlike infix notation, postfix expressions do not require parentheses to dictate order of operations.
2. Efficient Evaluation: Postfix expressions can be evaluated using a single left-to-right scan, making them ideal for stack-based computation.

**Conversion from Infix to Postfix:**

The conversion of an infix expression to postfix follows these steps:

1. Initialize an empty stack for operators and an empty list for output.
2. Scan the infix expression left to right:
   * If an operand (number/variable) is encountered, add it to the output.
   * If an operator is encountered:
     + Pop operators from the stack to the output until the stack is empty or a lower precedence operator is found.
     + Push the current operator onto the stack.
   * If a left parenthesis is encountered, push it onto the stack.
   * If a right parenthesis is encountered, pop and output operators until a left parenthesis is found (discard the parentheses).
3. Pop any remaining operators in the stack and append them to the output.

**Conclusion:**

Both problems effectively use the stack data structure to solve the given tasks efficiently. The solutions ensure correctness and handle edge cases such as empty inputs, mismatched brackets, and division by zero (handled with integer division in Python).