Exercise No. 11

Topics Covered : Stack and Queue Date : 12.07.2024

Solve the following problems

|  |  |  |
| --- | --- | --- |
| **Q.**  **No.** | **Question Detail** | **Level** |
| **1** | **Minimum String Length After Removing Substrings**  **Problem Statement:** You are given a string s consisting only of uppercase English letters.  You can apply some operations to this string where, in one operation, you can remove any occurrence of one of the substrings "AB" or "CD" from s.  Return the minimum possible length of the resulting string that you can obtain.  Note that the string concatenates after removing the substring and could produce new "AB" or "CD" substrings.  **Example 1:**  **Input:** s = "ABFCACDB"  **Output:** 2  **Explanation:** We can do the following operations:   * Remove the substring "ABFCACDB", so s = "FCACDB". * Remove the substring "FCACDB", so s = "FCAB". * Remove the substring "FCAB", so s = "FC". So the resulting length of the string is 2.   It can be shown that it is the minimum length that we can obtain.  **Example 2:**  **Input:** s = "ACBBD"  **Output:** 5  **Explanation**: We cannot do any operations on the string so | Easy |

|  |  |  |
| --- | --- | --- |
|  | the length remains the same.  **Constraints:**  1 <= s.length <= 100  s consists only of uppercase English letters.  Program:  package com.HandsOn;  import java.util.Scanner;  public class MinimumStringLengthAfterRemovingSubstrings {  public static void main(String[] args) {  Scanner sc = new Scanner(System.***in***);  String s = sc.next();  System.***out***.println(*minLength*(s));  }  public static int minLength(String s) {  char arr[] = new char[s.length()];  int i =-1;  for (char ch : s.toCharArray()) {  if (i >= 0 && ((ch == 'B' && arr[i] == 'A') || (ch == 'D' && arr[i] == 'C'))) {  i--;  } else {  arr[++i] = ch;  }  }  return i + 1;  }  }  Output:  ABFCACDB  2 |  |
| **2** | **Infix To Postfix**  **Problem statement :** You are given a string 'exp' which is a valid infix expression.  Convert the given infix expression to postfix expression. **Note:** Infix notation is a method of writing mathematical expressions in which operators are placed between operands.  For example, "3 + 4" represents the addition of 3 and 4. Postfix notation is a method of writing mathematical expressions in which operators are placed after the operands.  For example, "3 4 +" represents the addition of 3 and 4. Expression contains digits, lower case English letters, ‘(’, ‘)’, ‘+’, ‘-’, ‘\*’, ‘/’, ‘^’.  **Example:**  **Input:** exp = ‘3+4\*8’  **Output:** 348\*+  **Explanation:**  Here multiplication is performed first and then the addition operation. Hence postfix expression is 3 4 8 \* +.  **Sample Input 1:**  3^(1+1)  **Expected Answer:**  311+^  **Output printed on console:**  311+^  **Explanation of Sample Input 1:** | Easy |

|  |  |  |
| --- | --- | --- |
|  | For this testcase, we will evaluate 'b' = (1+1) first. Hence it's equivalent postfix expression will be "11+".  Next we will evaluate 3^b. It's equivalent postfix expression will be "3b^".  Replacing 'b' with it's equivalent postfix we get "311+^".  **Sample Input 2:**  a+b+c+d-e  **Expected Answer**:  ab+c+d+e-  **Output printed on console:**  ab+c+d+e-  **Constraints:**  1 <= 'n' <= 5000  ‘n’, is the length of EXP  The expression contains digits, lower case English letters, ‘(’, ‘)’, ‘+’, ‘-’, ‘\*’, ‘/’, ‘^’.  Program:  package com.HandsOn;  import java.util.Scanner;  import java.util.Stack;  public class InfixtoPostfix {  static int Prec(char ch) {  switch (ch) {  case '+':  case '-':  return 1;  case '\*':  case '/':  return 2;  case '^':  return 3;  }  return -1;  }  static String infixToPostfix(String exp) {  String result = "";  Stack<Character> stack = new Stack<>();  for (int i = 0; i < exp.length(); ++i) {  char c = exp.charAt(i);  if (Character.*isLetterOrDigit*(c))  result += c;  else if (c == '(')  stack.push(c);  else if (c == ')') {  while (!stack.isEmpty() && stack.peek() != '(')  result += stack.pop();  stack.pop();  } else {  while (!stack.isEmpty() && *Prec*(c) <= *Prec*(stack.peek())) {  result += stack.pop();  }  stack.push(c);  }  }  while (!stack.isEmpty()) {  if (stack.peek() == '(')  return "Invalid Expression";  result += stack.pop();  }  return result;  }  public static void main(String[] args) {  Scanner scanner = new Scanner(System.***in***);  System.***out***.println("Enter an infix expression: ");  String exp = scanner.nextLine();  System.***out***.println("Infix expression: " + exp);  System.***out***.println("Postfix expression: " + *infixToPostfix*(exp));  scanner.close();  }  }  Output:  Enter an infix expression:  3+4\*8  Infix expression: 3+4\*8  Postfix expression: 348\*+ |  |
| **3** | **Evaluation of postfix expression**  **Problem statement :** An expression is called the postfix expression if the operator appears in the expression after the operands.  **Example :**  Infix expression: A + B \* C - D Postfix expression: A B + C D - \*  Given a postfix expression, the task is to evaluate the expression. The answer could be very large, output your answer modulo (10^9+7). Also, use modular division when required.  **Note**:   1. Operators will only include the basic arithmetic operators like '\*', '/', '+', and '-'. 2. The operand can contain multiple digits. 3. The operators and operands will have space as a separator between them. | Easy |

|  |  |  |
| --- | --- | --- |
|  | 4. There won’t be any brackets in the postfix expression.  **Sample input 1:**  2  2 3 1 \* + 9 -  1 2 3 + \* 8 -  **Sample output 1 :**  -4  -3  **Explanation of sample input 1: Test case 1:**  2 3 1 \* + 9 -  - : ( ) - ( )  9 : ( ) - (9)  + : ( ( ) + ( ) ) - (9)  '\*': ( ( ) + ( ( ) \* ( ) ) ) - (9)  1 : ( ( ) + ( ( ) \* (1) ) ) - (9)  3 : ( ( ) + ( (3) \* (1) ) ) - (9)  2 : ( (2) + ( (3) \* (1) ) ) - (9)  Result = (2 + 3) - 9 = 5 - 9 = -4  **Test case 2:**  1 2 3 + \* 8 -  - : ( ) - ( )  8 : ( ) - (8)  \* : ( ( ) \* ( ) ) - (8)  + : ( ( ) \* ( ( ) + ( ) ) ) - (8)  3 : ( ( ) \* ( ( ) + (3) ) ) - (8)  2 : ( ( ) \* ( (2) + (3) ) ) - (8)  1 : ( (1) \* ( (2) + (3) ) ) - (8)  Result = (1 \* 5) - 8 = 5 - 8 = -3  **Sample input 2 :** |  |

|  |  |  |
| --- | --- | --- |
|  | 1  100 200 + 2 / 5 \* 7 +  **Sample output 2:**  757  **Explanation of sample input 2:**  100 + 200 = 300  300 / 2 = 150  150 \* 5 = 750  750 + 7 = 757  **Constraints :**  1 <= T <= 100  1 <= N <= 10^3  1 <= NUM <= 100  Where ‘N’ denotes the length of postfix expression and ‘NUM’  denotes the operand.  Program:  package com.HandsOn;  import java.util.Scanner;  import java.util.Stack;  public class PostfixEvaluation {  static int evaluatePostfix(String exp)  {  Stack<Integer> stack = new Stack<>();  for (int i = 0; i < exp.length(); i++) {  char c = exp.charAt(i);  if (Character.*isDigit*(c))  stack.push(c - '0');  else {  int val1 = stack.pop();  int val2 = stack.pop();  switch (c) {  case '+':  stack.push(val2 + val1);  break;  case '-':  stack.push(val2 - val1);  break;  case '/':  stack.push(val2 / val1);  break;  case '\*':  stack.push(val2 \* val1);  break;  }  }  }  return stack.pop();  }  public static void main(String[] args)  {  Scanner scanner = new Scanner(System.***in***);  System.***out***.println("Enter an Postfix expression: ");  String exp = scanner.nextLine();  System.***out***.println("postfix evaluation: "  + *evaluatePostfix*(exp));  }  }  Output:  100 200 + 2 / 5 \* 7 +  757 |  |
| **4** | **Arithmetic Expression Evaluation**  **Problem statement**  You are given a string ‘expression’ consists of characters ‘+’, ‘-’, ‘\*’, ‘/’, ‘(‘, ‘)’ and ‘0’ to ‘9’, that represents an Arithmetic Expression in Infix Notation. Your task is to evaluate this Arithmetic Expression.  In Infix Notation, operators are written in-between their operands.  **Note :**   1. We consider the ‘/’ operator as the floor division. 2. Operators ‘\*’ and ‘/’ expression has higher precedence over operators‘+’ and ‘-’ 3. String expression always starts with ‘(‘ and ends with ‘)’. 4. It is guaranteed that ‘expression’ represents’ a valid   expression in Infix notation.   1. It is guaranteed that there will be no case that requires | Easy |

|  |  |  |
| --- | --- | --- |
|  | division by 0.   1. No characters other than those mentioned above are present in the string. 2. It is guaranteed that the operands and final result will fit in a 32-bit integer.   For example :  Consider string ‘expression’ = ‘((2+3)\*(5/2))’.  Then it’s value after evaluation will be ((5)\*(2)) = 10.  **Sample Input 1 :**  2  (2) ((2+3)\*(5/2))  **Sample Output 1 :**  2  10  **Explanation For Sample Input 1 :**  **Test case 1 :**  The value of the expression “(2)” will be 2 after evaluation.  **Test case 2 :**  See the problem statement for an explanation.  **Sample Input 2 :**  2  (121+(101+0)) (3\*(5+2)\*(10-7))  **Sample Output 2 :**  222  63  **Constraints :**  1 <= T <= 50  3 <= |expression| <= 10^4  Program:  package com.HandsOn;  import java.util.Scanner;  import java.util.Stack;  public class ArithmeticExpressionEvaluation {  static int precedence(char ch) {  switch (ch) {  case '+':  case '-':  return 1;  case '\*':  case '/':  return 2;  case '^':  return 3;  }  return -1;  }  static int applyOperation(int a, int b, char op) {  switch (op) {  case '+':  return a + b;  case '-':  return a - b;  case '\*':  return a \* b;  case '/':  return a / b;  }  return 0;  }  static String infixToPostfix(String exp) {  StringBuilder result = new StringBuilder();  Stack<Character> stack = new Stack<>();  for (int i = 0; i < exp.length(); ++i) {  char c = exp.charAt(i);  if (Character.*isDigit*(c)) {  result.append(c);  }  else if (c == '(') {  stack.push(c);  }  else if (c == ')') {  while (!stack.isEmpty() && stack.peek() != '(') {  result.append(stack.pop());  }  stack.pop();  } else {  result.append(' ');  while (!stack.isEmpty() && *precedence*(c) <= *precedence*(stack.peek())) {  result.append(stack.pop());  }  stack.push(c);  }  }  while (!stack.isEmpty()) {  result.append(stack.pop());  }  return result.toString();  }  static int evaluatePostfix(String exp) {  Stack<Integer> stack = new Stack<>();  for (int i = 0; i < exp.length(); i++) {  char c = exp.charAt(i);  if (c == ' ') continue;  if (Character.*isDigit*(c)) {  int n = 0;  // Extract the numeric value  while (Character.*isDigit*(c)) {  n = n \* 10 + (int)(c - '0');  i++;  if (i < exp.length())  c = exp.charAt(i);  else  break;  }  i--;  stack.push(n);  } else {  int val1 = stack.pop();  int val2 = stack.pop();  switch (c) {  case '+':  stack.push(val2 + val1);  break;  case '-':  stack.push(val2 - val1);  break;  case '\*':  stack.push(val2 \* val1);  break;  case '/':  stack.push(val2 / val1);  break;  }  }  }  return stack.pop();  }  public static void main(String[] args) {  Scanner scanner = new Scanner(System.***in***);  System.***out***.println("Enter the number of test cases: ");  int T = scanner.nextInt();  scanner.nextLine();  for (int t = 0; t < T; t++) {  System.***out***.println("Enter an infix expression: ");  String expression = scanner.nextLine();  String postfix = *infixToPostfix*(expression);  int result = *evaluatePostfix*(postfix);  System.***out***.println("The value of the expression " + expression + " is: " + result);  }  scanner.close();  }  }  Output:  Enter the number of test cases:  2  Enter an infix expression:  (2)  The value of the expression (2) is: 2  Enter an infix expression:  ((2+3)\*(5/2))  The value of the expression ((2+3)\*(5/2)) is: 10 |  |

|  |  |  |
| --- | --- | --- |
| **5** | **Evaluate Reverse Polish Notation** | Easy |
|  | **Problem Statement:** You are given an array of strings |  |
|  | tokens that represents an arithmetic expression in a Reverse |  |
|  | Polish Notation. |  |
|  | Evaluate the expression. Return an integer that represents |  |
|  | the value of the expression. |  |
|  | Note that: |  |
|  | The valid operators are '+', '-', '\*', and '/'. |  |
|  | Each operand may be an integer or another expression. |  |
|  | The division between two integers always truncates toward |  |
|  | zero. |  |
|  | There will not be any division by zero. |  |
|  | The input represents a valid arithmetic expression in a |  |
|  | reverse polish notation. |  |
|  | The answer and all the intermediate calculations can be |  |
|  | represented in a 32-bit integer. |  |
|  | **Example 1:** |  |
|  | **Input:** tokens = ["2","1","+","3","\*"] |  |
|  | **Output:** 9 |  |
|  | **Explanation:** ((2 + 1) \* 3) = 9 |  |
|  | **Example 2:** |  |
|  | **Input:** tokens = ["4","13","5","/","+"] |  |
|  | **Output:** 6 |  |
|  | **Explanation:** (4 + (13 / 5)) = 6 |  |
|  | **Example 3:** |  |
|  | **Input:** tokens = ["10","6","9","3","+","- |  |
|  | 11","\*","/","\*","17","+","5","+"] |  |
|  | **Output:** 22 |  |
|  | **Explanation:** ((10 \* (6 / ((9 + 3) \* -11))) + 17) + 5 |  |
|  | = ((10 \* (6 / (12 \* -11))) + 17) + 5 |  |
|  | = ((10 \* (6 / -132)) + 17) + 5 |  |
|  | = ((10 \* 0) + 17) + 5 |  |
|  | = (0 + 17) + 5 |  |
|  | = 17 + 5 |  |

|  |  |  |
| --- | --- | --- |
|  | = 22  **Constraints:**  1 <= tokens.length <= 10^4  tokens[i] is either an operator: "+", "-", "\*", or "/", or an integer in the range [-200, 200].  Program:  package com.HandsOn;  import java.util.Scanner;  import java.util.Stack;  class ReversePolishNotation {  public static int evalRPN(String[] tokens) {  Stack<Integer> stack = new Stack<>();  for (String token : tokens) {  if (token.equals("+") || token.equals("-") || token.equals("\*") || token.equals("/")) {  int b = stack.pop();  int a = stack.pop();  int result = 0;  switch (token) {  case "+":  result = a + b;  break;  case "-":  result = a - b;  break;  case "\*":  result = a \* b;  break;  case "/":  result = a / b;  break;  }  stack.push(result);  } else {  stack.push(Integer.*parseInt*(token));  }  }  return stack.pop();  }  public static void main(String[] args) {  Scanner scanner = new Scanner(System.***in***);  System.***out***.println("Enter the tokens of the RPN expression (separated by spaces):");  String input = scanner.nextLine();  // Split the input line into tokens  String[] tokens = input.split(" ");  int result = *evalRPN*(tokens);  System.***out***.println("The value of the expression is: " + result);  scanner.close();  }  }  Output:  Enter the tokens of the RPN expression (separated by spaces):  2 1 + 3 \*  The value of the expression is: 9 |  |
| **6** | **Decode String**  **Problem Statement:** Given an encoded string, return its decoded string.  The encoding rule is: k[encoded\_string], where the encoded\_string inside the square brackets is being repeated exactly k times. Note that k is guaranteed to be a positive integer.  You may assume that the input string is always valid; there are no extra white spaces, square brackets are well-formed, etc. Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, k. For example, there will not be input like 3a or 2[4].  The test cases are generated so that the length of the output will never exceed 105.  **Example 1:**  **Input:** s = "3[a]2[bc]"  **Output:** "aaabcbc"  **Example 2:**  **Input:** s = "3[a2[c]]"  **Output:** "accaccacc"  **Example 3:**  **Input:** s = "2[abc]3[cd]ef"  **Output:** "abcabccdcdcdef"  **Constraints:**   * 1 <= s.length <= 30 | Medium |

|  |  |  |
| --- | --- | --- |
|  | * s consists of lowercase English letters, digits, and square brackets '[]'. * s is guaranteed to be a valid input. * All the integers in s are in the range [1, 300].   Program:  package com.HandsOn;  import java.util.Stack;  import java.util.Scanner;  public class DecodeString {  public static String decodeString(String s) {  Stack<Integer> countStack = new Stack<>();  Stack<StringBuilder> stringStack = new Stack<>();  StringBuilder currentString = new StringBuilder();  int k = 0;  for (char ch : s.toCharArray()) {  if (Character.*isDigit*(ch)) {  k = k \* 10 + (ch - '0'); // Construct the number k  } else if (ch == '[') {  // Push the current number and current string to their stacks  countStack.push(k);  stringStack.push(currentString);  // Reset currentString and k for the next segment  currentString = new StringBuilder();  k = 0;  } else if (ch == ']') {  // Pop the number of times to repeat  int repeatTimes = countStack.pop();  // Pop the last built string  StringBuilder decodedString = stringStack.pop();  // Append the current built string repeated k times to the decoded string  for (int i = 0; i < repeatTimes; i++) {  decodedString.append(currentString);  }  // Update the current string to the decoded string  currentString = decodedString;  } else {  // Append current character to current string  currentString.append(ch);  }  }  return currentString.toString();  }  public static void main(String[] args) {  Scanner scanner = new Scanner(System.***in***);  System.***out***.println("Enter the encoded string:");  String s = scanner.nextLine();  System.***out***.println("Decoded string: " + *decodeString*(s));  scanner.close();  }  }  Output:  Enter the encoded string:  3[a]2[bc]  Decoded string: aaabcbc |  |
| **7** | **Simplify Path**  **Problem Statement:** Given an absolute path for a Unix- style file system, which begins with a slash '/', transform this path into its simplified canonical path.  In Unix-style file system context, a single period '.' signifies the current directory, a double period ".." denotes moving up one directory level, and multiple slashes such as "//" are interpreted as a single slash. In this problem, treat sequences of periods not covered by the previous rules (like "...") as valid names for files or directories.  The simplified canonical path should adhere to the following rules:  It must start with a single slash '/'.  Directories within the path should be separated by only one slash '/'.  It should not end with a slash '/', unless it's the root directory.  It should exclude any single or double periods used to denote current or parent directories.  Return the new path.  **Example 1:**  **Input:** path = "/home/" **Output:** "/home" **Explanation:**  The trailing slash should be removed.  **Example 2:**  **Input:** path = "/home//foo/" **Output:** "/home/foo" **Explanation:**  Multiple consecutive slashes are replaced by a single one. | Medium |

|  |  |  |
| --- | --- | --- |
|  | **Example 3:**  **Input:** path = "/home/user/Documents/../Pictures"  **Output:** "/home/user/Pictures"  **Explanation:**  A double period ".." refers to the directory up a level.  **Example 4:**  **Input:** path = "/../" **Output:** "/" **Explanation:**  Going one level up from the root directory is not possible.  **Constraints:**  1 <= path.length <= 3000  path consists of English letters, digits, period '.', slash '/' or '\_'.  path is a valid absolute Unix path.  Program:  package com.HandsOn;  import java.util.Scanner;  import java.util.Stack;  public class SimplifyPath {  public static String simplifyPath(String path) {  Stack<String> stack = new Stack<>();  String[] components = path.split("/");  for (String part : components) {  if (part.isEmpty() || part.equals(".")) {  // Skip empty parts or current directory indicators  continue;  } else if (part.equals("..")) {  // Pop the stack if we encounter a parent directory indicator  if (!stack.isEmpty()) {  stack.pop();  }  } else {  // Push valid directory names onto the stack  stack.push(part);  }  }  // Build the simplified path  StringBuilder simplifiedPath = new StringBuilder("/");  for (String dir : stack) {  simplifiedPath.append(dir).append("/");  }  // Remove the trailing slash if not the root directory  if (simplifiedPath.length() > 1) {  simplifiedPath.setLength(simplifiedPath.length() - 1);  }  return simplifiedPath.toString();  }  public static void main(String[] args) {  Scanner scanner = new Scanner(System.in);  System.out.println("Enter the Unix-style absolute path:");  String path = scanner.nextLine();  System.out.println("Simplified path: " + simplifyPath(path));  scanner.close();  }  }  Output:  Enter the Unix-style absolute path:  /home/user/Documents/../Pictures  Simplified path: /home/user/Pictures |  |
| **8** | **Remove All Adjacent Duplicates in String II**  **Problem Statement:** You are given a string s and an integer k, a k duplicate removal consists of choosing k adjacent and equal letters from s and removing them, causing the left and the right side of the deleted substring to concatenate together.  We repeatedly make k duplicate removals on s until we no longer can.  Return the final string after all such duplicate removals have been made. It is guaranteed that the answer is unique.  **Example 1:**  **Input:** s = "abcd", k = 2  **Output:** "abcd"  **Explanation:** There's nothing to delete.  **Example 2:**  **Input:** s = "deeedbbcccbdaa", k = 3  **Output:** "aa"  **Explanation:**  First delete "eee" and "ccc", get "ddbbbdaa" | Medium |

|  |  |  |
| --- | --- | --- |
|  | Then delete "bbb", get "dddaa" Finally delete "ddd", get "aa" **Example 3:**  Input: s = "pbbcggttciiippooaais", k = 2 Output: "ps"  **Constraints:**  1 <= s.length <= 10^5  2 <= k <= 10^4  s only contains lowercase English letters.  Program:  package com.HandsOn;  import java.util.Scanner;  import java.util.Stack;  public class RemoveAdjacentDuplicates {  public static String removeDuplicates(String s, int k) {  Stack<Pair> stack = new Stack<>();  for (char c : s.toCharArray()) {  if (!stack.isEmpty() && stack.peek().character == c) {  stack.peek().count++;  if (stack.peek().count == k) {  stack.pop();  }  } else {  stack.push(new Pair(c, 1));  }  }  StringBuilder result = new StringBuilder();  for (Pair pair : stack) {  for (int i = 0; i < pair.count; i++) {  result.append(pair.character);  }  }  return result.toString();  }  static class Pair {  char character;  int count;  Pair(char character, int count) {  this.character = character;  this.count = count;  }  }  public static void main(String[] args) {  Scanner scanner = new Scanner(System.***in***);  System.***out***.println("Enter the string:");  String s = scanner.nextLine();  System.***out***.println("Enter the value of k:");  int k = scanner.nextInt();  System.***out***.println("Resulting string after removing duplicates: " + *removeDuplicates*(s, k));  scanner.close();  }  }  Output:  Enter the string:  pbbcggttciiippooaais  Enter the value of k:  2  Resulting string after removing duplicates: ps |  |
| **9** | **Minimum Remove to Make Valid Parentheses**  **Problem Statement:** Given a string s of '(' , ')' and lowercase English characters.  Your task is to remove the minimum number of parentheses ( '(' or ')', in any positions ) so that the resulting parentheses string is valid and return any valid string.  Formally, a parentheses string is valid if and only if:  It is the empty string, contains only lowercase characters, or It can be written as AB (A concatenated with B), where A and B are valid strings, or  It can be written as (A), where A is a valid string.  **Example 1:**  **Input**: s = "lee(t(c)o)de)"  **Output:** "lee(t(c)o)de"  **Explanation:** "lee(t(co)de)" , "lee(t(c)ode)" would also be accepted.  **Example 2:**  **Input:** s = "a)b(c)d"  **Output:** "ab(c)d"  **Example 3:**  **Input:** s = "))(("  **Output:** ""  **Explanation:** An empty string is also valid. | Medium |

|  |  |  |
| --- | --- | --- |
|  | **Constraints:**  1 <= s.length <= 10^5  s[i] is either '(' , ')', or lowercase English letter.  Program:  package com.HandsOn;  import java.util.HashSet;  import java.util.Scanner;  import java.util.Set;  import java.util.Stack;  public class MinimumRemoveValidParentheses {  public static String minRemoveToMakeValid(String s) {  Stack<Integer> stack = new Stack<>();  Set<Integer> toRemove = new HashSet<>();  for (int i = 0; i < s.length(); i++) {  char c = s.charAt(i);  if (c == '(') {  stack.push(i);  } else if (c == ')') {  if (stack.isEmpty()) {  toRemove.add(i);  } else {  stack.pop();  }  }  }  while (!stack.isEmpty()) {  toRemove.add(stack.pop());  }  StringBuilder result = new StringBuilder();  for (int i = 0; i < s.length(); i++) {  if (!toRemove.contains(i)) {  result.append(s.charAt(i));  }  }  return result.toString();  }  public static void main(String[] args) {  Scanner scanner = new Scanner(System.***in***);  System.***out***.println("Enter the string:");  String s = scanner.nextLine();  System.***out***.println("Valid string after minimum removal: " + *minRemoveToMakeValid*(s));  scanner.close();  }  }  Output:  Enter the string:  a)b(c)d  Valid string after minimum removal: ab(c)d |  |
| **10** | **Score of Parentheses**  **Problem Statement:** Given a balanced parentheses string s, return the score of the string.  The score of a balanced parentheses string is based on the following rule:   * "()" has score 1. * AB has score A + B, where A and B are balanced parentheses strings. * (A) has score 2 \* A, where A is a balanced parentheses string.   **Example 1:**  **Input:** s = "()"  **Output:** 1  **Example 2:**  **Input:** s = "(())"  **Output:** 2  **Example 3:**  **Input**: s = "()()"  **Output:** 2  **Constraints:**  2 <= s.length <= 50  s consists of only '(' and ')'.  s is a balanced parentheses string.  Program:  package com.HandsOn;  import java.util.Scanner;  import java.util.Stack;  public class ScoreOfParentheses {  public static int scoreOfParentheses(String s) {  Stack<Integer> stack = new Stack<>();  stack.push(0);  for (char c : s.toCharArray()) {  if (c == '(') {  stack.push(0);  } else {  int innerScore = stack.pop();  int outerScore = stack.pop();  int scoreToAdd = Math.*max*(2 \* innerScore, 1);  stack.push(outerScore + scoreToAdd);  }  }  return stack.pop();  }  public static void main(String[] args) {  Scanner scanner = new Scanner(System.***in***);  System.***out***.println("Enter the balanced parentheses string:");  String s = scanner.nextLine();  System.***out***.println("Score of the parentheses string: " + *scoreOfParentheses*(s));  scanner.close();  }  }  Program:  Enter the balanced parentheses string:  ()()  Score of the parentheses string: 2 | Medium |
| **11** | **Reverse Substrings Between Each Pair of Parentheses**  **Problem Statement:** You are given a string s that consists of lower case English letters and brackets. | Hard |

|  |  |  |
| --- | --- | --- |
|  | Reverse the strings in each pair of matching parentheses, starting from the innermost one.Your result should not contain any brackets.  **Example 1:**  Input: s = "(abcd)" Output: "dcba"  **Example 2:**  Input: s = "(u(love)i)" Output: "iloveu"  Explanation: The substring "love" is reversed first, then the whole string is reversed**.**  **Example 3:**  Input: s = "(ed(et(oc))el)" Output: "leetcode"  Explanation: First, we reverse the substring "oc", then "etco", and finally, the whole string.  **Constraints:**  1 <= s.length <= 2000  s only contains lower case English characters and parentheses.  It is guaranteed that all parentheses are balanced.  Program:  package com.HandsOn;  import java.util.Scanner;  import java.util.Stack;  public class ReverseSubstringsBetweenParentheses {  public static String reverseParentheses(String s) {  Stack<StringBuilder> stack = new Stack<>();  StringBuilder current = new StringBuilder();  for (char c : s.toCharArray()) {  if (c == '(') {  stack.push(current);  current = new StringBuilder();  } else if (c == ')') {  current.reverse();  current = stack.pop().append(current);  } else {  current.append(c);  }  }  return current.toString();  }  public static void main(String[] args) {  Scanner scanner = new Scanner(System.***in***);  System.***out***.println("Enter the string with parentheses:");  String s = scanner.nextLine();  System.***out***.println("Resulting string after reversing substrings: " + *reverseParentheses*(s));  scanner.close();  }  }  Program:  Enter the string with parentheses:  (u(love)i)  Resulting string after reversing substrings: iloveu |  |
| **12** | **Count Subarrays With Fixed Bounds**  **Problem Statement**: You are given an integer array nums and two integers minK and maxK.  A fixed-bound subarray of nums is a subarray that satisfies the following conditions:  The minimum value in the subarray is equal to minK. The maximum value in the subarray is equal to maxK. Return the number of fixed-bound subarrays.  A subarray is a contiguous part of an array.  **Example 1:** | Hard |

|  |  |  |
| --- | --- | --- |
|  | **Input:** nums = [1,3,5,2,7,5], minK = 1, maxK = 5  **Output**: 2  **Explanation:** The fixed-bound subarrays are [1,3,5] and [1,3,5,2].  **Example 2:**  **Input:** nums = [1,1,1,1], minK = 1, maxK = 1  **Output**: 10  **Explanation:** Every subarray of nums is a fixed-bound subarray. There are 10 possible subarrays.  **Constraints:**  2 <= nums.length <= 10^5  1 <= nums[i], minK, maxK <= 10^6  Program:  package com.HandsOn;  import java.util.Scanner;  public class CountSubarraysWithFixedBounds {  public static long countSubarrays(int[] nums, int minK, int maxK) {  int n = nums.length;  int lastMinKIndex = -1;  int lastMaxKIndex = -1;  int leftBound = -1;  long count = 0;  for (int i = 0; i < n; i++) {  if (nums[i] < minK || nums[i] > maxK) {  leftBound = i;  lastMinKIndex = -1;  lastMaxKIndex = -1;  }  if (nums[i] == minK) {  lastMinKIndex = i;  }  if (nums[i] == maxK) {  lastMaxKIndex = i;  }  if (lastMinKIndex != -1 && lastMaxKIndex != -1) {  count += Math.*max*(0, Math.*min*(lastMinKIndex, lastMaxKIndex) - leftBound);  }  }  return count;  }  public static void main(String[] args) {  Scanner scanner = new Scanner(System.***in***);  System.***out***.println("Enter the number of elements in the array:");  int n = scanner.nextInt();  int[] nums = new int[n];  System.***out***.println("Enter the elements of the array:");  for (int i = 0; i < n; i++) {  nums[i] = scanner.nextInt();  }  System.***out***.println("Enter the value of minK:");  int minK = scanner.nextInt();  System.***out***.println("Enter the value of maxK:");  int maxK = scanner.nextInt();  long result = *countSubarrays*(nums, minK, maxK);  System.***out***.println("Number of fixed-bound subarrays: " + result);  scanner.close();  }  }  Output:  Enter the number of elements in the array:  6  Enter the elements of the array:  1 3 5 2 7 5  Enter the value of minK:  1  Enter the value of maxK:  5  Number of fixed-bound subarrays: 2 |  |
| **13** | **Maximum Frequency Stack**  **Problem Statement:** Design a stack-like data structure to push elements to the stack and pop the most frequent element from the stack.  Implement the FreqStack class:  FreqStack() constructs an empty frequency stack.  void push(int val) pushes an integer val onto the top of the stack.  int pop() removes and returns the most frequent element in the stack.  If there is a tie for the most frequent element, the element closest to the stack's top is removed and returned.  **Example 1:**  **Input**  ["FreqStack", "push", "push", "push", "push", "push", "push", "pop", "pop", "pop", "pop"]  [[], [5], [7], [5], [7], [4], [5], [], [], [], []]  **Output**  [null, null, null, null, null, null, null, 5, 7, 5, 4]  **Explanation**  FreqStack freqStack = new FreqStack(); | Hard |

|  |  |  |
| --- | --- | --- |
|  | freqStack.push(5); // The stack is [5] freqStack.push(7); // The stack is [5,7] freqStack.push(5); // The stack is [5,7,5] freqStack.push(7); // The stack is [5,7,5,7] freqStack.push(4); // The stack is [5,7,5,7,4] freqStack.push(5); // The stack is [5,7,5,7,4,5]  freqStack.pop(); // return 5, as 5 is the most frequent. The stack becomes [5,7,5,7,4].  freqStack.pop(); // return 7, as 5 and 7 is the most frequent, but 7 is closest to the top. The stack becomes [5,7,5,4].  freqStack.pop(); // return 5, as 5 is the most frequent. The stack becomes [5,7,4].  freqStack.pop(); // return 4, as 4, 5 and 7 is the most frequent, but 4 is closest to the top. The stack becomes [5,7].  **Constraints:**  0 <= val <= 10^9  At most 2 \* 10^4 calls will be made to push and pop.  It is guaranteed that there will be at least one element in the stack before calling pop.  Program:  package com.HandsOn;  import java.util.HashMap;  import java.util.Map;  import java.util.Scanner;  import java.util.Stack;  class FreqStack {  private Map<Integer, Integer> freqMap; // Map to store frequency of each element  private Map<Integer, Stack<Integer>> groupMap; // Map to store stacks of elements by their frequency  private int maxFreq; // Variable to keep track of the current maximum frequency  public FreqStack() {  freqMap = new HashMap<>();  groupMap = new HashMap<>();  maxFreq = 0;  }  public void push(int val) {  int freq = freqMap.getOrDefault(val, 0) + 1;  freqMap.put(val, freq);  if (!groupMap.containsKey(freq)) {  groupMap.put(freq, new Stack<>());  }  groupMap.get(freq).push(val);  if (freq > maxFreq) {  maxFreq = freq;  }  }  public int pop() {  int val = groupMap.get(maxFreq).pop();  freqMap.put(val, freqMap.get(val) - 1);  if (groupMap.get(maxFreq).isEmpty()) {  maxFreq--;  }  return val;  }  public static void main(String[] args) {  FreqStack freqStack = new FreqStack();  Scanner scanner = new Scanner(System.***in***);  while (true) {  System.***out***.println("Choose an operation:");  System.***out***.println("1: Push");  System.***out***.println("2: Pop");  System.***out***.println("3: Exit");  int choice = scanner.nextInt();  if (choice == 1) {  System.***out***.println("Enter value to push:");  int value = scanner.nextInt();  freqStack.push(value);  System.***out***.println("Pushed: " + value);  } else if (choice == 2) {  int poppedValue = freqStack.pop();  System.***out***.println("Popped: " + poppedValue);  } else if (choice == 3) {  break;  } else {  System.***out***.println("Invalid choice. Please try again.");  }  }  scanner.close();  }  }  Output:  Choose an operation:  1: Push  2: Pop  3: Exit  1  Enter value to push:  5  Pushed: 5  Choose an operation:  1: Push  2: Pop  3: Exit  1  Enter value to push:  7  Pushed: 7  Choose an operation:  1: Push  2: Pop  3: Exit  1  Enter value to push:  5  Pushed: 5  Choose an operation:  1: Push  2: Pop  3: Exit  1  Enter value to push:  7  Pushed: 7  Choose an operation:  1: Push  2: Pop  3: Exit  1  Enter value to push:  4  Pushed: 4  Choose an operation:  1: Push  2: Pop  3: Exit  1  Enter value to push:  5  Pushed: 5  Choose an operation:  1: Push  2: Pop  3: Exit  2  Popped: 5  Choose an operation:  1: Push  2: Pop  3: Exit  2  Popped: 7  Choose an operation:  1: Push  2: Pop  3: Exit  2  Popped: 5  Choose an operation:  1: Push  2: Pop  3: Exit  2  Popped: 4  Choose an operation:  1: Push  2: Pop  3: Exit  2  Popped: 7  Choose an operation:  1: Push  2: Pop  3: Exit  2  Popped: 5 |  |

