**Faculty of Computing**

**SE-314: Software Construction**

**Class: BESE 14AB**

# Lab 08: Recursion

**CLO-03:** Design and develop solutions based on Software Construction principles.  
**CLO-04:** Use modern tools such as Eclipse, NetBeans etc. for software construction.

**Date: 27th Oct 2025**

**Time: 09:00 AM** **- 12:50 PM   
 02:00 PM – 04:50 PM**

**Submitted By: Sikander Hayat Khan**

**CMS ID: 464544**

**Instructor: Dr. Sidra Sultana  
Lab Engineer: Mr. Aftab Farooq**

**Github Repo Link:** [**https://github.com/Sikander-Hayat-Khan/SC-LAB-8.git**](https://github.com/Sikander-Hayat-Khan/SC-LAB-8.git)

**Introduction:**

# Lab 08: Recursion

Students will have hands-on experience on designing, testing, and implementing recursive problems. Given a scenario, you will write the specifications and implement it by dividing into base case and recursive step. You may design helper methods to simplify your implementations. Write unit tests that check for compliance with the specifications.

Lectures on LMS regarding Recursion

## Lab Tasks

**Task 1: Recursive File Search**

**Objective:** The objective of this lab task is to create a Java program that recursively searches for a file within a directory and its subdirectories. This exercise will help you practice the principles of software construction and recursion.

**Instructions:**

1. Create a Java program that takes two command-line arguments: a directory path and a file name to search for.
2. Implement a recursive function to search for the specified file within the given directory and its subdirectories.
3. The program should display a message when it finds the file, including the full path to the file, or a message indicating that the file was not found.
4. Follow good coding practices, including meaningful variable names, comments, and modular code.
5. Implement error handling to handle cases where the specified directory does not exist or other exceptions may occur.
6. Use appropriate data structures and algorithms to efficiently search through the directory tree.
7. Test your program with different directory paths and file names to ensure its correctness and reliability.

Important: Do not forget to write the specifications and unit tests for the code.

**Mandatory Enhancements:**

1. Allow the program to search for multiple files in a single run.
2. Implement a feature to count the number of times a specific file appears within the directory and its subdirectories.
3. Provide an option to specify whether the search should be case-sensitive or case-insensitive.

**Task 2: Recursive String Permutations**

**Objective:** The objective of this lab task is to create a Java program that generates all permutations of a given string using a recursive algorithm. This exercise will help you practice recursion and algorithm design.

**Instructions:**

1. Create a Java program that generates all permutations of a given string using a recursive function.
2. Implement a recursive function **generatePermutations** that takes a string as input and returns a list of all its permutations.
3. Use a recursive approach to generate permutations. You can consider swapping characters in the string to create different permutations.
4. Follow good coding practices, including meaningful variable names, comments, and modular code.
5. Implement error handling to handle cases where the input string is empty or other exceptions may occur.
6. Analyze the time complexity of the recursive algorithm. How does the time complexity compare to an iterative solution for large strings?

**Mandatory Enhancements:**

1. Provide an option for the user to choose whether to include or exclude duplicate permutations, as some characters in the input string may be identical.
2. Implement a non-recursive algorithm for generating permutations and compare its performance with the recursive solution for large strings.

**Task 3: Testing Recursive Programs :**  
Design and implement JUnit test cases to verify the correctness and reliability of your recursive solutions from Task 1 and Task 2. Ensure tests cover base cases, recursive steps, and edge conditions such as invalid inputs or empty data. Analyze test results to confirm that all recursive functions behave as expected.

|  |
| --- |
| **Solution** |
| **Task 1:**  **Features implemented:**   * Recursively searches for file(s) in directory and subdirectories * Handles multiple filenames * Counts occurrences * Supports case-sensitive and case-insensitive search * Handles invalid directories gracefully   **Code:**  import java.io.File; import java.util.ArrayList; import java.util.List; import java.util.Scanner;  public class RecursiveFileSearch {  */\*\*  \* Recursively searches for a given file within the specified directory and its subdirectories.  \*  \* @param directory The directory in which to begin searching.  \* @param fileName The exact name of the file to search for.  \* @param caseSensitive If true, the search is case-sensitive; if false, it ignores case.  \* @param foundPaths A list that will store the absolute paths of all matching files found.  \*  \* @precondition directory is not null and represents an existing directory.  \* @postcondition All occurrences of the specified file within the directory tree are added to foundPaths.  \* @throws SecurityException if access to a directory is denied by the system.  \*/* // Recursive method to search for files  public static void searchFile(File directory, String fileName, boolean caseSensitive, List<String> foundPaths) {  if (directory == null || !directory.exists() || !directory.isDirectory()) {  return;  }   File[] files = directory.listFiles();  if (files == null) return;   for (File file : files) {  if (file.isDirectory()) {  // Recursive call  *searchFile*(file, fileName, caseSensitive, foundPaths);  } else {  if (caseSensitive) {  if (file.getName().equals(fileName)) {  foundPaths.add(file.getAbsolutePath());  }  } else {  if (file.getName().equalsIgnoreCase(fileName)) {  foundPaths.add(file.getAbsolutePath());  }  }  }  }  }   */\*\*  \* Main method to run the recursive file search program.  \* Prompts the user for a directory path, file names, and case-sensitivity option.  \* Displays results or appropriate error messages.  \*  \* @throws IllegalArgumentException if the directory does not exist or is invalid.  \*/* public static void main(String[] args) {  Scanner sc = new Scanner(System.*in*);  System.*out*.print("Enter directory path: ");  String directoryPath = sc.nextLine();  System.*out*.print("Enter file names to search (comma-separated): ");  String[] fileNames = sc.nextLine().split(",");  System.*out*.print("Case-sensitive search? (yes/no): ");  boolean caseSensitive = sc.nextLine().trim().equalsIgnoreCase("yes");  sc.close();   File directory = new File(directoryPath);   if (!directory.exists() || !directory.isDirectory()) {  System.*out*.println("Error: Directory does not exist!");  return;  }   for (String rawName : fileNames) {  String fileName = rawName.trim();  List<String> foundPaths = new ArrayList<>();  *searchFile*(directory, fileName, caseSensitive, foundPaths);   if (foundPaths.isEmpty()) {  System.*out*.println("File not found: " + fileName);  } else {  System.*out*.println("Found " + fileName + " " + foundPaths.size() + " time(s):");  for (String path : foundPaths) {  System.*out*.println(" → " + path);  }  }  }  } }  **Screenshot:** |
| **Task 2:**  **Features implemented:**   * Recursive permutation generator * Handles duplicates (optional inclusion/exclusion) * Non-recursive (iterative) version for performance comparison * Handles empty strings   **Code:**  import java.util.\*;  */\*\*  \* StringPermutations  \* ------------------  \* This program generates all permutations of a given string recursively and  \* optionally excludes duplicate permutations. A non-recursive version is also included.  \* It also measures and compares the execution time of both methods.  \*/* public class StringPermutations {   */\*\*  \* Recursively generates all permutations of the given string.  \*  \* @param str the remaining string to permute  \* @param prefix the accumulated prefix of characters chosen so far  \* @param result the set where generated permutations will be stored  \* @precondition str != null, result != null  \* @postcondition All permutations of the original string are added to result  \*/* public static void generatePermutations(String str, String prefix, Set<String> result) {  if (str.isEmpty()) {  result.add(prefix);  return;  }   for (int i = 0; i < str.length(); i++) {  char ch = str.charAt(i);  String remaining = str.substring(0, i) + str.substring(i + 1);  *generatePermutations*(remaining, prefix + ch, result);  }  }   */\*\*  \* Generates permutations using an iterative (non-recursive) Heap’s algorithm.  \*  \* @param str the input string  \* @return list of all generated permutations  \* @precondition str != null  \* @postcondition returns all permutations of str  \*/* public static List<String> generateIterative(String str) {  List<String> permutations = new ArrayList<>();  char[] arr = str.toCharArray();  int n = arr.length;  int[] c = new int[n];  permutations.add(new String(arr));   int i = 0;  while (i < n) {  if (c[i] < i) {  if (i % 2 == 0) {  char temp = arr[0];  arr[0] = arr[i];  arr[i] = temp;  } else {  char temp = arr[c[i]];  arr[c[i]] = arr[i];  arr[i] = temp;  }  permutations.add(new String(arr));  c[i]++;  i = 0;  } else {  c[i] = 0;  i++;  }  }  return permutations;  }   public static void main(String[] args) {  Scanner sc = new Scanner(System.*in*);  System.*out*.print("Enter a string: ");  String input = sc.nextLine();   if (input.isEmpty()) {  System.*out*.println("Error: String cannot be empty!");  sc.close();  return;  }   System.*out*.print("Include duplicate permutations? (yes/no): ");  boolean includeDuplicates = sc.nextLine().trim().equalsIgnoreCase("yes");   // --- Recursive permutation generation ---  Set<String> recursiveResult = includeDuplicates ? new LinkedHashSet<>() : new TreeSet<>();  long startRecursive = System.*nanoTime*();  *generatePermutations*(input, "", recursiveResult);  long endRecursive = System.*nanoTime*();  long recursiveTime = endRecursive - startRecursive;   // --- Iterative permutation generation ---  long startIterative = System.*nanoTime*();  List<String> iterativeResult = *generateIterative*(input);  long endIterative = System.*nanoTime*();  long iterativeTime = endIterative - startIterative;   // --- Display results ---  System.*out*.println("\n=== Recursive Permutations (" + recursiveResult.size() + " total) ===");  for (String s : recursiveResult) System.*out*.println(s);  System.*out*.printf("Recursive execution time: %.6f ms%n", recursiveTime / 1\_000\_000.0);   System.*out*.println("\n=== Iterative Permutations (" + iterativeResult.size() + " total) ===");  for (String s : iterativeResult) System.*out*.println(s);  System.*out*.printf("Iterative execution time: %.6f ms%n", iterativeTime / 1\_000\_000.0);   // --- Compare ---  System.*out*.println("\nPerformance Comparison:");  if (recursiveTime < iterativeTime)  System.*out*.println("Recursive version was faster!");  else if (iterativeTime < recursiveTime)  System.*out*.println("Iterative version was faster!");  else  System.*out*.println("Both took roughly the same time!");   sc.close();  } }  **Comment on Time Complexity:**   |  |  |  | | --- | --- | --- | | **String** | **Recursive Approach** | **Iterative Approach** | | **“Ali”** | **10.416800 ms** | **0.051400 ms** | | **“Ahmad”** | **10.035300 ms** | **0.036700 ms** |   **Screenshot:** |
| **Task 3:**  **Code:**  import org.junit.jupiter.api.Test; import static org.junit.jupiter.api.Assertions.\*; import java.io.File; import java.util.\*;  public class RecursiveTests {   // ---------- Tests for RecursiveFileSearch ----------  @Test  public void testFileExistsCaseInsensitive() {  List<String> found = new ArrayList<>();  File dir = new File(System.*getProperty*("user.dir"));  RecursiveFileSearch.*searchFile*(dir, "RecursiveFileSearch.java", false, found);  *assertFalse*(found.isEmpty(), "File should be found at least once");  }   @Test  public void testFileNotFound() {  List<String> found = new ArrayList<>();  File dir = new File(System.*getProperty*("user.dir"));  RecursiveFileSearch.*searchFile*(dir, "nonexistentfile123.txt", false, found);  *assertTrue*(found.isEmpty(), "Non-existent file should not be found");  }   // ---------- Tests for StringPermutations ----------  @Test  public void testPermutationsCount() {  Set<String> result = new HashSet<>();  StringPermutations.*generatePermutations*("abc", "", result);  *assertEquals*(6, result.size(), "3! permutations should be generated");  }   @Test  public void testPermutationsEmptyString() {  Set<String> result = new HashSet<>();  StringPermutations.*generatePermutations*("", "", result);  *assertTrue*(result.contains(""), "Empty string permutation should be itself");  }   @Test  public void testIterativeMatchesRecursive() {  Set<String> recursive = new HashSet<>();  StringPermutations.*generatePermutations*("abc", "", recursive);  List<String> iterative = StringPermutations.*generateIterative*("abc");  *assertEquals*(recursive.size(), iterative.size(), "Iterative and recursive should produce same number");  } }  **Screenshot:** |

### Deliverables:

Compile a single word document by filling in the solution part and submit this Word file on LMS.

In case of any problems with submissions on LMS, submit your Lab assignments by emailing it to [aftab.farooq@seecs.edu.pk.](mailto:aftab.farooq@seecs.edu.pk.)