**package** CriticalThinking.Module5;

**public** **class** RecursiveFactorials {

// Straight recursive calculation by submitting n-1 on each recursive call

**public** **static** **int** straightFactorail(**int** n) {

**if** (n == 0) {

**return** 1;

} **else** {

System.***out***.println(n + " \* " + (n-1));

**return** n \* *straightFactorail*(n - 1);

}

}

// Factorial by multiplying a ‘factorial’ accumulator value by n on each recursion, then submitting n-1 into the next iteration

**public** **static** **int** factorailWithAccumulator(**int** n, **int** factorial) {

**if** (n == 0) {

**return** factorial;

} **else** {

System.***out***.println(n + " \* " + factorial);

**return** *factorailWithAccumulator*(n - 1, n \* factorial);

}

}

**public** **static** **void** main(String[] args) {

**int** n1 = 5;

// Straight recursion

**int** factorial1 = *straightFactorail*(n1);

System.***out***.println("Factorial of 5 using straight recursion: " + factorial1);

// Recursion with an accumulator

**int** factorial2 = *factorailWithAccumulator*(n1, 1);

System.***out***.println("Factorial of 5 using an accumulator: " + factorial2);

**int** n2 = 7;

**int** factorial3 = *straightFactorail*(n2);

System.***out***.println("\nFactorial of 7 using straight recursion: " + factorial3);

**int** factorial4 = *factorailWithAccumulator*(n2, 1);

System.***out***.println("Factorial of 7 using an accumulator: " + factorial4);

}

}

A screenshot of a computer

Description automatically generated

The dominant operations in this algorithm are the two factorial recursive functions. The functions ‘straightFactorail()’ and ‘factorailWithAccumulator()’ both process once for each integer between 1 and n, so both process through n times.

‘straightFactorail()’ multiplies the current n by the recursion of n-1, so by the end we have n\*n-1\*n-2\*...\*2\*1, build up from 1 to n by one over each iteration. This shows that ‘straightFactorial()’ is of linear complexity, as the level of complexity increases linearly with n.

‘factorialWithAccumulator()’ takes the current n and multiplies it by the ‘accumulator’ value, which in this case is 1, then passes n-1 and the product of n\*accumulator as the new accumulator into the recursion so that the next iteration is n-1\*n\*accumulator and so on, so the end result would be 1\*2\*...n-2\*n-1\*n. This function is simply the reverse of ‘straightFactorial()’, and as such has linear complexity since it’s time complexity increases linearly for each integer in the value of n.

As both of these are linear, or O(n), and the contained operations is of constant time, namely the multiplication of two variables and the recursion call, the overall time complexity of this program is also O(n).

**package** CriticalThinking.Module5;

**import** javafx.application.Application;

**import** javafx.geometry.Insets;

**import** javafx.geometry.Pos;

**import** javafx.scene.Scene;

**import** javafx.scene.control.Button;

**import** javafx.scene.control.TextArea;

**import** javafx.scene.layout.VBox;

**import** javafx.stage.Stage;

**import** java.util.Random;

**public** **class** RandomArrayBackwards **extends** Application {

**private** String[] arr;

**private** Random random = **new** Random();

**private** TextArea inputArea;

**private** TextArea resultArea;

**private** Button displayButton;

**private** String result;

@Override

**public** **void** start(Stage primaryStage) {

primaryStage.setTitle("Array Portion Reverse");

// Create GUI components

displayButton = **new** Button("Display Random Portion");

displayButton.setOnAction(e -> displayRandomPortion());

inputArea = **new** TextArea();

inputArea.setPromptText("Enter a string");

resultArea = **new** TextArea();

resultArea.setPromptText("A portion will be displayed backwards");

resultArea.setEditable(**false**);

inputArea.setDisable(**true**);

// Create a VBox and add components

VBox vbox = **new** VBox(10);

vbox.setPadding(**new** Insets(20));

vbox.setAlignment(Pos.***CENTER***);

vbox.getChildren().addAll(inputArea, displayButton, resultArea);

Scene scene = **new** Scene(vbox, 300, 200);

primaryStage.setScene(scene);

primaryStage.show();

displayButton.requestFocus();

inputArea.setDisable(**false**);

}

**private** **void** displayRandomPortion() {

result = "";

String input = inputArea.getText();

arr = input.split("");

**if** (input.length() > 0) {

**int** start = random.nextInt(arr.length);

**int** end = random.nextInt(arr.length);

// Ensure start index is smaller than end index

**if** (start > end) {

**int** temp = start;

start = end;

end = temp;

}

displayArrayPortionBackwards(start, end);

resultArea.setText(result);

} **else** {

displayButton.setText("Please enter a string, then click the button");

}

}

**private** **void** displayArrayPortionBackwards(**int** start, **int** end) {

**if** (start > end) {

**return**;

}

result += arr[end];

displayArrayPortionBackwards(start, end - 1);

}

**public** **static** **void** main(String[] args) {

*launch*(args);

}

}

A screenshot of a computer

Description automatically generated with medium confidenceA screenshot of a computer

Description automatically generated with medium confidenceA screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidenceA screenshot of a computer

Description automatically generated with medium confidenceA screenshot of a computer

Description automatically generated with medium confidence

Breaking down this whole program, we have the import statements, declaration of variables, the start method, the displayRandomPortion method, the displayArrayPortionBackwards method, and the main method.

The import statements have no impact on time complexity as they happen during compilation. The variables that are declared but not instantiated with a value also have no impact on time complexity and the instantiation of the random variable as a Random is in constant time.

The Start method is then called and sets up the JavaFX components. During this step the creation of the components does not involve any computation complexity so this step does not impact the time complexity of the system.

The method displayRandomPortion that is called when the button is pressed. First it initializes the variable ‘result’ with an empty string, which takes constant time. It then takes the input text and splits it into an array with the split("") method and has a complexity proportional to the length of the input string, thus has a linear time complexity of O(n). It then runs into the ‘If’ statement that takes in the variables ‘end’ and ‘start’ to verify that ‘start’ is less than ‘end’. If not, it swaps the values to ensure that our start is always less than the end integer, which takes constant time.

Then the method displayArrayPortionBackwards is called and takes in the ‘start’ and ‘end’ variables. It then appends the element at index ‘end’ from the array ‘arr’ to the string ‘result’ starting at index 0, then recursively calls displayArrayPortionBackwrds passing in the same ‘start’ variable along with ‘end - 1’ which this method uses to append the next character to the right of the previous one. This method would iterate from the integers ‘end’ to ‘start’ so it would have a time complexity of (end - start). Since the difference between ‘start’ and ‘end’ it randomly generated we can say that that number is k, so it would have O(k) complexity. To take worst case scenario, we can assume that ‘start’ is randomly selected as ‘1’ and ‘end’ is randomly selected to be ‘n’ since ‘n’ is the largest random integer in range 1 - n. If this is the case, then this method would have complexity O(n).

The main method is called to initiate the launch(args) function and has no relevance to this analysis.

Given that the dominant operations in this program have time complexity O(n), we can say that this program has an overall complexity of O(n).