The estimated doubling ratio for my implementation is O(n), roughly ~2.0 since no algorithm exceeds that time complexity in the implementation. First, I check if the String is null; this is O(1). Second, I check whether the string representation is empty or blank. isEmpty is O(1) since it looks at length and isBlank is O(n) since it checks every character in the string until it hits the end or a non-whitespace character. Afterward, I convert the string representation into a char array using .toCharArray() to make it easier to process. This is an O(n) performance algorithm since the string is iterated through once and put into an array. Next, I iterate through the char array and make sure the amount of open parenthesis equals the amount of close parenthesis as this is a one-way iteration, going over each element once, the complexity is O(n). Thus, the first step of the program is O(n).

After some initial checks, the first major algorithm of my program begins which verifies, in-depth, if the String representation is valid. This algorithm consists of a for loop which goes through the char array “processed” and runs a bunch of checks. All the checks performed in the inner loop are constant time as they are size-independent. I check if a char is not a validInt and for two successive Integers. Simultaneously, there are gets and sets to an ArrayList which is then checked to make sure each level doesn’t have too many children. As this process is based on index and .size() and no iteration is preformed, it is of constant time. While the average case of ArrayList.add() is constant it becomes O(n) when the underlying array is doubled. To remedy this, the ArrayList is initialized to its maximum length of processed.length() so that no doubling is required. Get is O(1). As such, the time complexity of this algorithm is calculated to be O(n) since every single character in the string representation is iterated over once and only once.

After all, checks are complete, the main algorithm of the program begins, and a string representation of a tree is returned. This algorithm generates a List<List<Integer>> and adds elements/nodes to the correct level (index) based on the amount of open and close parentheses encountered. To do this every character of representation must be iterated over once through a for-loop. The level of each tree is initialized to a predetermined length so that minimal array doubling is required over the life of the array. The reason each level isn’t initialized to 2^index is the array size would be memory inefficient for large indexes. During each iteration, a couple of checks for exceptions and increments are done to the open and close parenthesis count which are size independent and in extension O(1). Since the highest time complexity algorithm in the program is O(n), in conclusion, the algorithm is O(n)/ linear (with a constant of 3).