**1)Remove Outermost Parentheses**

**Example 1:**

Input: s = "((()))"

Output: "(())"

Explanation: The input string is a single primitive: "((()))".

Removing the outermost layer yields: "(())".

**Example 2:**

Input: s = "()(()())(())"

Output: "(()())()"

Explanation: Primitive decomposition: "()" + "(()())" + "(())"

After removing outermost parentheses: "" + "()()" + "()"

Final result: "(()())()".

**DryRun:**

1. Create one StringBuffer
2. Create the openCount variable
3. Use **forEach loop** inside loop convert string into array using **toCharArray()** method
4. Use 4 conditions

**class Solution {**

**public String removeOuterParentheses(String s) {**

**StringBuilder result = new StringBuilder();**

**int openCount = 0;**

**for (char c : s.toCharArray()) {**

**if (c == '(') {**

**if (openCount > 0) {**

**result.append(c);**

**}**

**openCount++;**

**} else if (c == ')') {**

**openCount--;**

**if (openCount > 0) {**

**result.append(c);**

**}**

**}**

**}**

**return result.toString();**

**}**

**}**

**Time Complexity: O(n)**, since we are performing a single traversal of the string.  
**Space Complexity: O(1)**, since we are using a few variables to track the current state.

2)**Reverse Words in a String**

**Approach**

* Use a stack to push all the words in a stack
* Now, all the words of the string are present in the stack, but in reverse order
* Pop elements of the stack one by one and add them to our answer variable. Remember to add a space between the words as well.
* Here’s a quick demonstration of the same

**🧩 Summary of How It Works**

1. Adds a space at the end of the string.
2. Reads characters one by one. By creating 1 variable
3. Whenever a space is found → pushes the word to the stack.
4. After reading all words, pops each word from the stack to reverse order.
5. Prints the reversed sentence.

**CODE:**

**import java.util.\*;**

**class Test**

**{**

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

**{**

**String s = "TUF is great for interview preparation";**

**System.out.println("After reversing words: ");**

**System.out.println(s);**

**s += " ";**

**Stack<String> st = new Stack<String>();**

**int i;**

**String str = "";**

**for (i = 0;i < s.length();i++)**

**{**

**if (s.charAt(i) == ' ')**

**{**

**st.push(str);**

**str = "";**

**}**

**else**

**{**

**str += s.charAt(i);**

**}**

**}**

**String ans = "";**

**while (st.size() != 1)**

**{**

**ans += st.peek() + " ";**

**st.pop();**

**}**

**ans += st.peek(); *// The last word should'nt have a space after it***

**System.out.println("After reversing words: ");**

**System.out.print(ans);**

**}}**

**Approach:**

* We start traversing the string from the end until we hit a space. It indicates that we have gone past a word and now we need to store it.
* We check if our answer variable is empty or not
* If it’s empty, it indicates that this is the last word we need to print, and hence, there shouldn’t be any space after this word.
* If it’s empty we add it to our result with a space after it. Here’s a quick demonstration of the same

**NOTE**:

**s.split():**

**What is split()?**

* The split() method in Java **divides a string into parts (substrings)** based on a **delimiter** (a separator like space, comma, etc.).
* It returns an **array of strings**.

**Example:**String s = "Java is fun";

String[] arr = s.split(" ");

**Output**:arr = ["Java", "is", "fun"]

**NOTE**:

**What does "\\s+" mean?**

This is a **regular expression (regex)** pattern used inside split().

Let’s decode it:

| **Symbol** | **Meaning** |
| --- | --- |
| **\s->** | Matches any **whitespace** character (like space, tab, or newline) |
| **+->** | Means **“one or more”** occurrences |

* class Solution {
* public String reverseWords(String s) {
* s = s.trim();
* String[] words = s.split("\\s+");
* StringBuilder sb = new StringBuilder();
* for(int i = words.length - 1; i >= 0; i--){
* sb.append(words[i]);
* if(i > 0){
* sb.append(" ");
* }
* }
* return sb.toString();
* }
* }

**Time Complexity: O(N),** N~length of string

**Space Complexity: O(1),** Constant Space

**NOTE:**

**🧩 1️⃣ split() — breaks a string into words or parts**

**Definition:**

The split() method divides a string into **multiple substrings** based on a **delimiter (like space, comma, etc.)** and returns a **String array (String[])**.

**Syntax:**

String[] arr = s.split(" ");

**Example:**

String s = "Java is fun";

String[] words = s.split(" ");

for (String w : words) {

System.out.println(w);

}

**Output:**

Java

is

fun

🧠 **Explanation:**

* split(" ") → splits whenever it finds a **space**.
* Returns **String array** (each element is a word).
* split() is used when we want to **work with words**, not individual characters.

**🧩 2️⃣ toCharArray() — breaks a string into characters**

**Definition:**

It converts the entire string into a **character array (char[])**.

**Example:**

String s = "Java";

char[] ch = s.toCharArray();

for (char c : ch) {

System.out.println(c);

}

**Output:**

J

a

v

a

🧠 **Explanation:**

* Breaks **every character**, including spaces, into array elements.
* Used when we need to **traverse or modify characters** (like reversing).

**🧩 3️⃣ charAt(index) — gets one character at a time**

**Definition:**

Returns the character present at the given index.

**Example:**

String s = "Java";

System.out.println(s.charAt(2));

**Output:**

v

🧠 **Explanation:**

* Used when you want **only one character**.
* No new array is created — it just accesses one position directly.

**🧠 Important Note about charAt()**

* The **charAt()** method **always requires an index value** (an integer).
* This index tells **which character** in the string you want to access.
* Indexing in Java strings starts from **0** (zero-based indexing).

3)**Largest Odd Number in a String.**

class Solution {

    public String largestOddNumber(String num) {

        for (int i = num.length() - 1; i >= 0; i--) {

            int digit = num.charAt(i) - '0';  // Convert char to int

            if (digit % 2 != 0) {

                return num.substring(0, i + 1); // Substring up to last odd digit

            }

        }

        return ""; // No odd digit found}}

| **Expression** | **Start** | **End** | **Includes Indexes** | **Output** |
| --- | --- | --- | --- | --- |
| substring(0, 3) | 0 | 3 | 0, 1, 2 | "420" |
| substring(0, 4) | 0 | 4 | 0, 1, 2, 3 | "4205" |
| substring(0, 5) | 0 | 5 | 0, 1, 2, 3, 4 | "42053" |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4)**Longest Common Prefix**  **Example 1**  **Input:** str = ["flower", "flow", "flight"]  **Output:** "fl"  **Explanation:** All strings in the array begin with the common prefix "fl".  **Example 2**  **Input:** str = ["apple", "banana", "grape", "mango"]  **Output:** ""  **Explanation:** None of the strings share a common starting sequence, so the result is an empty string.  **NOTE:**  The **substring() method in Java** is used to extract a part of a string. It returns a new string that starts from the given startIndex and ends **before** the given end  Index.  **CODE:** |  |  |  |  |

class Solution {

    public String longestCommonPrefix(String[] strs) {

        if (strs == null || strs.length == 0) return "";

        String prefix = strs[0];  // Start with the first word as prefix

        for (int i = 1; i < strs.length; i++) {

            // Keep reducing prefix until the word starts with it

            while (!strs[i].startsWith(prefix)) {

                prefix = prefix.substring(0, prefix.length() - 1);

                if (prefix.isEmpty()) {

                    return "";

                }

            }

        }

        return prefix;

    }

}

**Time Complexity**

* n = number of strings in the array (strs.length)
* m = length of the first string (strs[0].length()) — assume the maximum length of any string is “**m**”

✅ **Time Complexity:** O(n \* m²) in the worst case

In the worst case, we **compare each character of the prefix**.

Prefix can be at most length m → O(m) comparisons per word.

**5)Isomorphic String**

**Problem Statement:**Given two strings s and t, determine if they are isomorphic. Two strings s and t are isomorphic if the characters in s can be replaced to get t.  
All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character, but a character may map to itself.

**Examples**

**Example 1**

**Input:** s = "paper", t = "title"

**Output:** true

**Explanation:** The characters in "s" can be mapped one-to-one to characters in "t":

'p' → 't', 'a' → 'i', 'e' → 'l', 'r' → 'e'

Since the mapping is consistent and unique for each character, the strings are isomorphic.

**Example 2**

**Input:** s = "foo", t = "bar"

**Output:** false

**Explanation:** 'f' → 'b' is fine, 'o' → 'a' for the first 'o', But the second 'o' in "s" would need to map to 'r' in "t", which conflicts with the earlier mapping of 'o' → 'a'

This inconsistency makes it impossible to convert "s" to "t" using a one-to-one character mapping.

class Solution {

    public boolean isIsomorphic(String s, String t) {

        if (s.length() != t.length()) return false;

// In Java, a char is based on Unicode (which can represent far more than 256 characters).

// But in many algorithmic problems like this one (LeetCode, etc.), the input is restricted to ASCII characters.

// ASCII has 256 possible values (0–255);

        int[] mapS = new int[256]; // mapping from s → t

        int[] mapT = new int[256]; // mapping from t → s

        for (int i = 0; i < s.length(); i++) {

            char ch1 = s.charAt(i);

            char ch2 = t.charAt(i);

            if (mapS[ch1] == 0 && mapT[ch2] == 0) {

                // First time mapping

                mapS[ch1] = ch2;

                mapT[ch2] = ch1;

            } else {

                // Already mapped, check consistency

                if (mapS[ch1] != ch2 || mapT[ch2] != ch1) {

                    return false;

                }

            }

        }

        return true;

    }

}

Two strings are **isomorphic** if each character in the first string maps to **exactly one character** in the second string, and vice versa.

* "egg" → "add" works because:
  + 'e' always maps to 'a'
  + 'g' always maps to 'd'
  + No conflicts.
* "foo" → "bar" fails because:
  + 'o' in "foo" maps to 'a' the first time, but the next 'o' tries to map to 'r'.
  + This violates the rule of one-to-one mapping → not isomorphic.

**🔹 Key words to remember:**

* “One-to-one mapping”
* “Each character must map consistently”
* “No conflicts allowed”

6) **Check if one string is rotation of another**

**Problem Statement**: Given two strings s and goal, return true if and only if s can become goal after some number of shifts on s.  
A shift on s consists of moving the leftmost character of s to the rightmost position. For example, if s = "abcde", then it will be "bcdea" after one shift.

**Example 1:**

**Input: s = "rotation", goal = "tionrota"**

**Output: true**

**Explanation: After multiple left shifts on "rotation", we get:**

**1st shift → "otationr"**

**2nd shift → "tationro"**

**3rd shift → "ationrot"**

**4th shift → "tionrota"**

**So the goal string can be obtained by rotating the original string.**

**Example 2:**

**Input: s = "hello", goal = "lohelx"**

**Output: false**

**Explanation:**

Even after all possible rotations of "hello", we cannot form "lohelx" due to the presence of an extra character 'x'. Hence, it's not possible.

**a)Brute force solution :;p**

class Solution {

*// Brute force approach to check if goal is a rotation of s*

public boolean rotateString(String s, String goal) {

*// Strings must be same length to be rotations of each other*

if (s.length() != goal.length()) {

return false;

}

*// Try all possible rotations of s*

for (int i = 0; i < s.length(); i++) {

String rotated = s.substring(i) + s.substring(0, i);

if (rotated.equals(goal)) {

*// Return true if a match is found*

return true;

}

}

return false;

}

}

**NOTE:**

**🔹 Summary Table**

| **i** | **s.substring(i)** | **s.substring(0, i)** | **rotated result** |
| --- | --- | --- | --- |
| 0 | "abcde" | "" | "abcde" |
| 1 | "bcde" | "a" | "bcdea" |
| 2 | "cde" | "ab" | "cdeab" |
| 3 | "de" | “abc" | "deabc" |
| 4 | "e" | "abcd" | "eabcd" |

**TimeComplexity:**O(n^2)

|  |  |
| --- | --- |
| String concatenation (+): | O(n) |

|  |  |
| --- | --- |
| equals() comparison:O(n) |  |
| **Space Complexity:** O(N) (for the space needed to store each rotated string.) |  |

**b)Optimal Solution:**

**Algorithm**

* Double the original string by joining it with itself, creating a new string like s + s.
* Look for the target string goalinside this new doubled string.
* If goal exists within the doubled string, then it's a valid rotation, return true.
* If it's not found, that means the original string cannot be rotated to match goal, return false

**CODE:**

class Solution {

    // Method to check if one string is a rotation of another

    public boolean rotateString(String s, String goal) {

        // Check if lengths of both strings are unequal

        if (s.length() != goal.length()) {

            // Return false if lengths don't match

            return false;

        }

        // Concatenate the string with itself

        String doubledS = s + s;

        // Check if the goal is a substring of the concatenated string

        return doubledS.contains(goal);

    }

}

// Main class to run the test cases

public class Main {

    public static void main(String[] args) {

        // Create instance of Solution

        Solution sol = new Solution();

        System.out.println(sol.rotateString("rotation", "tionrota"));

    }

}

**Time Complexity: O(N)**, because checking for a substring in s + s is linear in time.  
**Space Complexity: O(N)** for the space needed to store the concatenated string s + s.

**7)Check if two Strings are anagrams of each other**

**a)BruteForce Solution:**

import java.util.Arrays;

public class Solution {

public boolean isAnagram(String str1, String str2) {

// Step 1: If lengths differ → not anagram

if (str1.length() != str2.length())

return false;

// Step 2: Convert both to lowercase (optional)

str1 = str1.toLowerCase();

str2 = str2.toLowerCase();

// Step 3: Convert to char arrays

char[] arr1 = str1.toCharArray();

char[] arr2 = str2.toCharArray();

// Step 4: Sort both arrays

Arrays.sort(arr1);

Arrays.sort(arr2);

// Step 5: Compare sorted arrays

return Arrays.equals(arr1, arr2);}}

**b)Optimal solution:**

import java.util.\*;

public class Main {

public static boolean checkAnagrams(String str1, String str2) {

*// when both of the strings have different lengths*

if (str1.length() != str2.length())

return false;

int[] freq = new int[26];

for (int i = 0; i < str1.length(); i++) {

freq[str1.charAt(i) - 'A']++;

}

for (int i = 0; i < str2.length(); i++) {

freq[str2.charAt(i) - 'A']--;

}

for (int i = 0; i < 26; i++) {

if (freq[i] != 0)

return false;

}

return true;

}

**Time Complexity:** O(n) where n is the length of string

**Space Complexity:**O(1)

Each character in str1 increases its corresponding position count in freq.

Example:  
str1 = "CAT"

* 'C' - 'A' = 2 → freq[2]++
* 'A' - 'A' = 0 → freq[0]++
* 'T' - 'A' = 19 → freq[19]++

So now:

freq[0] = 1 (A)

freq[2] = 1 (C)

freq[19] = 1 (T)

**Step 4: Decrease for the second string**

for (int i = 0; i < str2.length(); i++) {

freq[str2.charAt(i) - 'A']--;

}

👉 For each letter in str2, we **decrease** the same frequency.

If str2 = "ACT":

* 'A' → freq[0]-- → 0
* 'C' → freq[2]-- → 0
* 'T' → freq[19]-- → 0

If both strings have the same letters,  
every frequency becomes **zero**.

**Step 5: Verify all frequencies are zero**

for (int i = 0; i < 26; i++) {

if (freq[i] != 0)

return false;

}

👉 If any frequency is **not zero**, that means one string had extra letters — so not anagrams.

8)

//BRUTEfORCE SOLUTION:

import java.util.\*;

class Solution {

    public String frequencySort(String s) {

        // Step 1: Count frequencies

        Map<Character, Integer> freq = new HashMap<>();

        for (char c : s.toCharArray()) {

            freq.put(c, freq.getOrDefault(c, 0) + 1);}

        // Step 2: Create buckets (index = frequency)

        List<Character>[] buckets = new List[s.length() + 1];

        for (char c : freq.keySet()) {

            int count = freq.get(c);

            if (buckets[count] == null)

                buckets[count] = new ArrayList<>();

            buckets[count].add(c);}

        // Step 3: Build result from high to low frequency

        StringBuilder sb = new StringBuilder();

        for (int i = s.length(); i > 0; i--) {

            if (buckets[i] != null) {

                for (char c : buckets[i]) {

                    sb.append(String.valueOf(c).repeat(i));}}}

        return sb.toString();}}

**DRYRUN:**

**🧩 Step 1:💡 Code in focus:**

Map<Character, Integer> freq = new HashMap<>();

for (char c : s.toCharArray()) {

freq.put(c, freq.getOrDefault(c, 0) + 1);

}

**🧩 Step 1.1: s.toCharArray()**

👉 This method converts the **entire string s** into a **character array (char[])** at once.

**Example:**

If

String s = "tree";

then after conversion:

s.toCharArray() → ['t', 'r', 'e', 'e']

✅ So now, we have an array of characters we can loop through.

**🧩 Step 1.2: for (char c : s.toCharArray())**

This is a **for-each loop** (also called an *enhanced for loop*).  
It iterates through each element of the array **one by one**.

Let’s trace it:

| **Iteration** | **c (current character)** |
| --- | --- |
| 1 | 't' |
| 2 | 'r' |
| 3 | 'e' |
| 4 | 'e' |

So inside the loop, c represents each character from the string in sequence.

**🧩 Step 1.3: freq.put(c, freq.getOrDefault(c, 0) + 1);**

This line counts how many times each character appears.

Let’s dry-run it with "tree":

| **Step** | **Character c** | **freq.getOrDefault(c, 0)** | **Updated value** | **freq Map after put()** |
| --- | --- | --- | --- | --- |
| 1 | 't' | not found → 0 | 0 + 1 = 1 | {t=1} |
| 2 | 'r' | not found → 0 | 0 + 1 = 1 | {t=1, r=1} |
| 3 | 'e' | not found → 0 | 0 + 1 = 1 | {t=1, r=1, e=1} |
| 4 | 'e' | found → 1 | 1 + 1 = 2 | {t=1, r=1, e=2} |

✅ Final result:

freq = {t=1, r=1, e=2}

**🧠 How the parts work together:**

| **Concept** | **What it does** |
| --- | --- |
| toCharArray() | Converts the *whole* string into an array of characters |
| for (char c : …) | Loops through *each character* in that array |
| freq.getOrDefault(c, 0) | Fetches the current count (or 0 if not found) |
| freq.put(c, …) | Updates the count in the map |

**🧩 Step 2: Code**

List<Character>[] buckets = new List[s.length() + 1];

for (char c : freq.keySet()) {

int count = freq.get(c);

if (buckets[count] == null)

buckets[count] = new ArrayList<>();

buckets[count].add(c);

}

**⚙️ Before this step:**

Let’s say our string is

s = "tree"

From Step 1, we already have:

freq = {t=1, r=1, e=2}

**🧠 Step 2 Explained:**

**🔹 Line 1:**

List<Character>[] buckets = new List[s.length() + 1];

* s.length() = 4
* So we create an **array of lists** with 5 slots (index 0 → 4).
* buckets = [null, null, null, null, null]
* Each buckets[i] will store **characters that appear i times**.

So:

* buckets[1] → characters that appear once
* buckets[2] → characters that appear twice
* buckets[3] → characters that appear three times
* and so on…

**🔹 Line 2–6: Loop through each character in the frequency map**

for (char c : freq.keySet()) {

int count = freq.get(c);

if (buckets[count] == null)

buckets[count] = new ArrayList<>();

buckets[count].add(c);

}

This loop:

* Goes through every key (character) in the freq map.
* Finds how many times it appears (count = freq.get(c)).
* Puts that character in the correct bucket.

**🧮 Let’s dry-run it:**

| **Character (c)** | **Frequency (count)** | **buckets[count] before** | **Action** | **buckets after** |
| --- | --- | --- | --- | --- |
| 't' | 1 | null | Create new list → add 't' | [null, [t], null, null, null] |
| 'r' | 1 | [t] | Add 'r' to same list | [null, [t, r], null, null, null] |
| 'e' | 2 | null | Create new list → add 'e' | [null, [t, r], [e], null, null] |

✅ After the loop:

buckets[1] = [t, r]

buckets[2] = [e]

buckets[3] = null

buckets[4] = null

**🧩 What did we just do?**

We created **“buckets” (groups)** based on frequency.

| **Frequency** | **Characters in that bucket** |
| --- | --- |
| 1 | t, r |
| 2 | e |

So now, it’s super easy to build the result string:

* Take characters from the **highest frequency bucket** first
* Repeat them according to their frequency
* Join them to form "eetr"

**🧠 Summary of how this step works:**

| **Step** | **Action** | **Example** |
| --- | --- | --- |
| 1 | Create buckets[] array of size s.length()+1 | [null, null, null, null, null] |
| 2 | Loop over each character in freq | 't', 'r', 'e' |
| 3 | Get how many times it appeared (count) | 't' → 1, 'r' → 1, 'e' → 2 |
| 4 | Add the character into buckets[count] | buckets[1]=[t,r], buckets[2]=[e] |
| 5 | Later used to build final answer | "eetr" |

**NOTE:**

**List<Character>[] buckets = new List[s.length() + 1]; here is there any difference between the ArrayList and List ?**

**🧩 Line in your code:**

List<Character>[] buckets = new List[s.length() + 1];

Here,

* List is **an interface** (it defines *what* a list can do).
* ArrayList is a **class** that *implements* the List interface (it defines *how* the list actually works).

**🔹 Difference between List and ArrayList**

| **Feature** | **List** | **ArrayList** |
| --- | --- | --- |
| **Type** | Interface | Concrete class |
| **Can you create an object?** | ❌ No (you can’t do new List()) | ✅ Yes (you can do new ArrayList()) |
| **Purpose** | Defines behavior (methods like add(), get(), size(), etc.) | Provides actual implementation using a resizable array |
| **Example** | List<Character> list; | ArrayList<Character> list = new ArrayList<>(); |
| **Flexibility** | Can later point to any class that implements List (e.g., ArrayList, LinkedList, etc.) | Fixed to ArrayList only |

**🧠 Why your code uses List[] and not ArrayList[]**

In this line:

List<Character>[] buckets = new List[s.length() + 1];

You’re saying:

"I want an array of *lists*, but I don’t care what kind of list (ArrayList, LinkedList, etc.) right now."

Later, when you need to actually store elements, you create an **ArrayList** for that bucket:

if (buckets[count] == null)

buckets[count] = new ArrayList<>();

So the **interface** (List) gives flexibility,  
and the **class** (ArrayList) gives the actual storage behavior.

**✅ Summary:**

| **Concept** | **Meaning** |
| --- | --- |
| List | Interface (general type) |
| ArrayList | A specific implementation of the List interface |
| new List() | ❌ Not allowed |
| new ArrayList<>() | ✅ Allowed |
| List<Character>[] buckets | Array that can hold *any kind of list* |
| buckets[count] = new ArrayList<>(); | Assign a real list implementation to use |

Would you like me to show the same concept with a **small example program** that uses both List and ArrayList to make it clearer?

**You said:**

List<Character>[] buckets = new List[s.length() + 1]; but they have created the new List

**ChatGPT said:**

Excellent catch again, Deb 💡 — that’s a very sharp observation!  
It **looks** like we’re creating a new List[...], but actually, there’s a **subtle difference** here. Let’s break it down carefully 👇

**🧩 The line in question:**

List<Character>[] buckets = new List[s.length() + 1];

**🔍 First, what this line really means:**

We are **not creating List objects** here.  
We’re only creating an **array that can *hold* references to List objects** later.

So:

* The **array itself** is created. ✅
* But the **lists inside it** (buckets[0], buckets[1], etc.) are **still null** ❌.

**🧠 Think of it like this:**

Imagine an **empty shelf** 🪣 with 5 slots.

buckets = [null, null, null, null, null]

You’ve only built the **shelf (array)**.  
You have *not* yet placed any **boxes (ArrayLists)** on it.

**🔹 Why new List[...] is allowed but new List() is not:**

* You cannot do:
* List<Character> list = new List<>(); // ❌ Not allowed

Because List is an **interface**, and interfaces cannot be directly instantiated.

* But you **can** create an array of interface references:
* List<Character>[] arr = new List[5]; // ✅ Allowed

Because you’re **not instantiating** the interface —  
you’re only creating an **array that can hold List references**.

**🪣 Later in your code:**

if (buckets[count] == null)

buckets[count] = new ArrayList<>();

Now you’re creating the actual ArrayList objects and putting them inside the array.

✅ After adding elements:

buckets = [null, [t, r], [e], null, null]

So finally, each buckets[i] holds a real ArrayList.

**✅ Summary**

| **Step** | **Code** | **What happens** |
| --- | --- | --- |
| 1 | new List[s.length() + 1] | Creates an **array** that can hold List references (all null initially) |
| 2 | new ArrayList<>() | Actually creates the **list object** inside a bucket |
| 3 | buckets[count] = new ArrayList<>(); | Stores that list into the array at the correct index |

👉 So in short:

new List[s.length() + 1] creates **an array of list references**, not actual List objects.

**TimeComplexity**: O(n)

**NOTE: we have used 3 for loops how it can have O(n) instead of O(n^3) TC?**

Step 1: Counting frequencies

for (char c : s.toCharArray()) {

freq.put(c, freq.getOrDefault(c, 0) + 1);

}

✅ Runs once for each character in s  
→ That’s n times (if n = s.length()).

Time: O(n)

🔹 Step 2: Filling buckets

for (char c : freq.keySet()) {

int count = freq.get(c);

...

}

Here we loop through unique characters, not the entire string.

Let’s say:

* In "tree" → unique chars = 3 (t, r, e)
* In "aaabb" → unique chars = 2 (a, b)

So this loop runs at most 26 times (for lowercase English letters),  
or generally O(k), where k = number of unique characters.

Since k ≤ n,  
👉 this step is also O(n) in the worst case.

🔹 Step 3: Building the result

for (int i = s.length(); i > 0; i--) {

if (buckets[i] != null) {

for (char c : buckets[i]) {

sb.append(String.valueOf(c).repeat(i));

}

}

}

Here:

* Outer loop runs from n down to 1 → O(n) times.
* Inner loops (over buckets) together process each character once.

Even though it looks nested, the total number of operations inside equals the number of characters in the string.

So overall: O(n) again.

**BUT 🧠 Why not O(n³) or O(n²)?**

O(n) + O(n) + O(n) = O(3n) → O(n)

**Final Answer:**

Even though we have three loops,  
➡️ Each loop runs *independently and linearly*,  
➡️ So the **overall time complexity is O(n)**.

**8)Maximum Nesting Depth of Parenthesis**

**Example 1:**

**Input:** s = "(1+(2\*3)+((8)/4))+1"

**Output:** 3

**Explanation:** Digit 8 is inside of 3 nested parentheses in the string.

**Example 2:**

**Input:** s = "(1)+((2))+(((3)))"

**Output:** 3

**Explanation:** Digit 3 is inside of 3 nested parentheses in the string.

class Solution {

    public int maxDepth(String s) {

        int depth = 0, maxDepth = 0;

        for (char c : s.toCharArray()) {

            if (c == '(') {

                depth++;

                maxDepth = Math.max(maxDepth, depth);

            } else if (c == ')')

{

                depth--;}}

        return maxDepth;

}}

**Time Complexity: O(n)**, where n is the length of the string.  
  
**Space Complexity: O(1)**, as only constant extra space is used.

class Solution

{

    public int maxDepth(String s)

    {

        int ans = 0;

        Stack<Character> st = new Stack<Character>();

        for (Character c : s.toCharArray())

        {

            if (c == '(')

            {

                st.push(c);

            } else if (c == ')')

          {

                st.pop();

            }

            ans = Math.max(ans, st.size());

        }

        return ans;

    }

}