**Basic and Easy String Problems**

**1.Remove Outer Parantheses.**

A valid parentheses string is either empty "", "(" + A + ")", or A + B, where A and B are valid parentheses strings, and + represents string concatenation.

For example, "", "()", "(())()", and "(()(()))" are all valid parentheses strings.

A valid parentheses string s is primitive if it is nonempty, and there does not exist a way to split it into s = A + B, with A and B nonempty valid parentheses strings.

Given a valid parentheses string s, consider its primitive decomposition: s = P1 + P2 + ... + Pk, where Pi are primitive valid parentheses strings.

Return s after removing the outermost parentheses of every primitive string in the primitive decomposition of s.

**Example 1:**

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

Output: "()()()"

Explanation:

The input string is "(()())(())", with primitive decomposition "(()())" + "(())".

After removing outer parentheses of each part, this is "()()" + "()" = "()()()".

**Example 2:**

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

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

Explanation:

The input string is "(()())(())(()(()))", with primitive decomposition "(()())" + "(())" + "(()(()))".

After removing outer parentheses of each part, this is "()()" + "()" + "()(())" = "()()()()(())".

**Example 3:**

Input: s = "()()"

Output: ""

Explanation:

The input string is "()()", with primitive decomposition "()" + "()".

After removing outer parentheses of each part, this is "" + "" = "".

**Brute Force Approach:**

**Approach 1 :** Using Stack

Initialize a Stack:

Use a Stack to track the nesting level of parentheses.

Iterate through the string:

If the character is '(':

If the stack is not empty, append it to the result (because it's not an outermost parenthesis).

Push '(' onto the stack.

If the character is ')':

op the top of the stack (to close the last opened '(').

If the stack is not empty, append ')' to the result (because it's not an outermost parenthesis).

Return the result:

The StringBuilder contains the modified string after removing the outermost parentheses.

public String removeOuterParentheses(String s) {  
 Stack<Character> st = new Stack<>();  
 StringBuilder sb = new StringBuilder();  
 for(char ch : s.toCharArray()){  
 if(ch == '('){  
 if(!st.isEmpty()){  
 sb.append(ch);  
 }  
 st.push(ch);  
 }  
 else{  
 st.pop();  
 if(!st.isEmpty()){  
 sb.append(ch);  
 }  
 }  
 }  
 return sb.toString();  
}

**Time Complexity**: O(N)

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

**Optimal Approach:**

**Approach 2** : Iterative approach

Use a Counter (count):

This variable keeps track of nested levels of parentheses.

Iterate through the string:

If ch == '(':

If count > 0, it means it's not an outermost '(', so append it to the result.

Increase count to track the level of nesting.

If ch == ')':

Decrease count (indicating the closure of a parenthesis group).

If count > 0, it means it's not an outermost ')', so append it to the result.

Return the result:

The StringBuilder contains the modified string after removing the outermost parentheses.

public String removeOuterParentheses1(String s)  
{  
 int cnt = 0;  
 StringBuilder ans = new StringBuilder();  
 for(int i=0;i<s.length();i++)  
 {  
 if(s.charAt(i) == '(')  
 {  
 if(cnt > 0)  
 ans.append(s.charAt(i));  
 cnt+=1;  
 }  
 else if(s.charAt(i) == ')')  
 {  
 cnt-=1;  
 if(cnt > 0)  
 ans.append(s.charAt(i));  
 }  
  
 }  
  
 return ans.toString();  
}

**Time Complexity**: O(N)

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

**2.** **Reverse words in a given string / Palindrome Check  
  
Example 1:**

Input: s=”this is an amazing program”

Output: “program amazing an is this”

**Example 2:**

Input: s=”This is decent”

Output: “decent is This”

**Solution 1(Brute Force)**

Intuition: We just need to print the words in reverse order. Can we somehow store them in reverse order of the occurrence and then simply add it to our answer

**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.

public static String reverseword(String input)  
{  
 String value = "";  
 Stack<String> store = new Stack<>();  
 String ans = "";  
 for(int i=0;i<input.length();i++)  
 {  
 if(input.charAt(i) == ' ')  
 {  
 if(!value.equals(""))  
 {  
 store.push(value);  
 }  
 value="";  
 }  
 else {  
 value = value + input.charAt(i);  
 }  
 }  
 if(!value.equals(""))  
 {  
 store.push(value);  
 }  
 while(store.size() != 1)  
 {  
 ans += store.peek()+" ";  
 store.pop();  
 }  
 ans += store.peek();  
 return ans;  
}

**Time Complexity:** O(N), Traversing the entire string

**Space Complexity:** O(N), Stack and ans variable

**Optimal Approach:**

Intuition: Notice, that we are using a stack in order to perform our task. Can we somehow not use it and reverse the words as we move through the string? Could we store a word in reverse order when we are adding it to our answer variable?

**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.

public static String reverseword1(String input)  
{  
 String ans = "";  
 String value = "";  
 for(int i=input.length()-1;i>=0;i--)  
 {  
 if(input.charAt(i) == ' ')  
 {  
 if(!value.equals(""))  
 {  
 ans += *reverseS*(value)+" ";  
 value = "";  
 }  
 }else  
 {  
 value += input.charAt(i);  
 }  
  
 }  
 if(!value.equals("")) {  
 ans += *reverseS*(value);  
 }  
  
 return ans.trim();  
}  
  
public static String reverseS(String value)  
{  
 char[] arr = value.toCharArray();  
 int start = 0;  
 int end = value.length()-1;  
 while(start < end)  
 {  
 char a = arr[start];  
 arr[start] = arr[end];  
 arr[end] = a;  
 start++;  
 end--;  
 }  
 return new String(arr);  
}

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

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

**3.** **Largest odd number in a string**

You are given a string num, representing a large integer. Return the largest-valued odd integer (as a string) that is a non-empty substring of num, or an empty string "" if no odd integer exists.

A substring is a contiguous sequence of characters within a string.

**Example 1:**

Input: num = "52"

Output: "5"

Explanation: The only non-empty substrings are "5", "2", and "52". "5" is the only odd number.

**Example 2:**

Input: num = "4206"

Output: ""

Explanation: There are no odd numbers in "4206".

**Example 3:**

Input: num = "35427"

Output: "35427"

Explanation: "35427" is already an odd number.

public String largestOddNumber(String num)  
{  
 String ans ="";  
 int index = findOdd(num);  
 for(int i=0;i<=index;i++)  
 {  
 ans+=num.charAt(i);  
 }  
 return ans;  
}  
public int findOdd(String num)  
{  
 for(int i=num.length()-1;i>=0;i--)  
 {  
 int value = num.charAt(i)-'0';  
 if(value%2==1)  
 {  
 return i;  
 }  
 }  
 return -1;  
}

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

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

**4.** **Longest Common Prefix**

Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string "".

**Example 1:**

Input: strs = ["flower","flow","flight"]

Output: "fl"

**Example 2:**

Input: strs = ["dog","racecar","car"]

Output: ""

Explanation: There is no common prefix among the input strings.

**Brute Force Approach:**

Traverse through every string and check every element of the same index and then return the ans.

**Optimal Approach:**

Sorting the array helps because the smallest and largest words determine the prefix.

The first and last words in sorted order will have the most differences.

Comparing just these two words gives the longest common prefix efficiently.

public static String longestCommonPrefix(String[] strs)  
{  
 String output="";  
 Arrays.*sort*(strs);  
 String first = strs[0];  
 String last = strs[strs.length-1];  
 for(int i=0;i<Math.*min*(first.length(),last.length());i++)  
 {  
 if(first.charAt(i) != last.charAt(i))  
 {  
 return output;  
 }  
 output += first.charAt(i);  
 }  
 return output;  
}

**Time Complexity:** O(n log n + m) . Sorting takes O(n log n), where n is the number of words.

Prefix comparison takes O(m), where m is the prefix length.

**Space Complexity:** O(1)

**5.** **Isomorphic String**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.

**Example 1:**

Input: s = "egg", t = "add"

Output: true

Explanation:

The strings s and t can be made identical by:

Mapping 'e' to 'a'.

Mapping 'g' to 'd'.

**Example 2:**

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

Output: false

**Example 3:**

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

Output: true

**Example 4:**

Input: s = "badc", t = "kikp"

Output: true

public boolean isIsomorphic(String s, String t)  
{  
 if(s.length() != t.length()) return false;  
 HashMap<Character,Character> map = new HashMap<>();  
 HashMap<Character,Character> mapped = new HashMap<>();  
 for(int i=0;i<s.length();i++)  
 {  
 char sChar = s.charAt(i);  
 char tChar = t.charAt(i);  
 if(map.containsKey(sChar) && map.get(sChar) != tChar)return false;  
 else if(mapped.containsKey(tChar) && mapped.get(tChar) != sChar)return false;  
 map.put(sChar,tChar);  
 mapped.put(tChar,sChar);  
 }  
 return true;  
}

**Time Complexity:** O(n) **Space Complexity:** O(n+n)

**6.** **Check whether one string is a rotation of another**

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 = "abcde", goal = "cdeab"

Output: true

**Example 2:**

Input: s = "abcde", goal = "abced"

Output: false

**Brute Force Approach:**

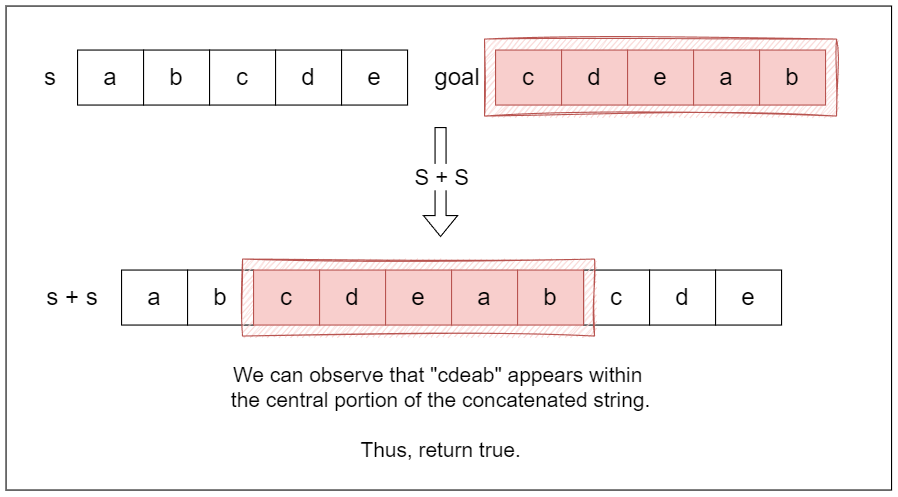
Rotate the string each time to left , and check if the strings are equal, you can rotate it to the size of the string .

**Optimal Approach:**

Instead of rotating the string and checking after each rotation, we can observe a relationship betweensandgoal. Ifgoalcan be formed by rotatings, it must be possible to findgoalas a substring in some version ofs

A clever way to exploit this is by concatenatingswith itself. Why? Because this effectively creates a string that contains all possible rotations ofswithin it. For example, ifs = "abcde", thens + s = "abcdeabcde". Notice how every possible rotation ofsappears somewhere in this concatenated string.

So, if goal can be obtained by rotatings, it must be a substring ofs + s. To implement this, we simply check if goal is a substring of the concatenated string. If it is, we returntrue; otherwise, we return false



public boolean rotateString(String s, String goal)  
{  
 if(s.length() != goal.length())return false;  
 return (s+s).indexOf(goal)>=0;  
}  
public boolean rotateString1(String s, String goal)  
{  
 return (s+s).contains(goal) && s.length()==goal.length();  
}

**Time complexity:**O(n)

Checking if the lengths of both strings are different takesO(n).

Concatenating the stringswith itself to createdoubledStringtakesO(n)because we are creating a new string that is twice the length ofs.

The substring find function is typically implemented using an algorithm that runs inO(n). This involves scanning thedoubledStringof length2nfor the substringgoalof lengthn. Since the search occurs in a string of size2n, the overall complexity for this operation remainsO(n).

Overall, the most significant operations are linear in terms ofn, resulting in a total time complexity ofO(n).

**Space complexity:**O(n)

The space used for thedoubledStringisO(n)since it stores a string that is double the size ofs(specifically,O(2⋅n)≈O(n)).

Thus, the overall space complexity is O(n) due to the concatenated string.

**7.** **Check if two strings are anagram of each other**

**Problem Statement:** Given two strings, check if two strings are anagrams of each other or not.

**Example 1:**

Input: CAT, ACT

Output: true

Explanation: Since the count of every letter of both strings are equal.

**Example 2:**

Input: RULES, LESRT

Output: false

Explanation: Since the count of U and T is not equal in both strings.

**Brute Force Approach:**

Sort both the string and compare each and every letter of both strings. If all letters matched then, print true. Otherwise, print false.

public static String SortString(String str)  
{  
 char c[] = str.toCharArray();  
 Arrays.*sort*(c);  
 return new String(c);  
}  
public static boolean checkAnagrams(String str1, String str2)  
{  
 // Case 1: when both of the strings have different lengths  
 if (str1.length() != str2.length())  
 return false;  
  
 str1 = *SortString*(str1);  
 str2 = *SortString*(str2);  
  
 // Case 2: check if every character of str1 and str2 matches with each other  
 for (int i = 0; i < str1.length(); i++)  
 {  
 if (str1.charAt(i) != str2.charAt(i))  
 return false;  
 }  
 return true;  
}

**Time Complexity:** O(nlogn) since sorting function requires nlogn iterations.

**Space Complexity:** O(1)

**Optimal Approach:**

Approach: Just count the frequency of every element in Str1 and iterate through Str2 and decrease the count of every element in the frequency array. Now iterate again, if the frequency at any point is not 0 this means, strings are not anagrams of each other.

public boolean isAnagram(String s, String t)  
{  
 if(s.length() != t.length())  
 return false;  
 int[] hash = new int[26];  
 for(int i=0;i<s.length();i++)  
 {  
 hash[(s.charAt(i)-'a')]++;  
 }  
 for(int i=0;i<s.length();i++)  
 {  
 hash[(t.charAt(i)-'a')]--;  
 }  
 for(int i=0;i<26;i++)  
 {  
 if(hash[i] != 0)  
 return false;  
 }  
  
 return true;  
}

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

**Space Complexity:** O(1)

**Medium String Problems**   
  
**1. Longest Palindromic Substring[Do it without DP]**

Given a string s, return the longest palindromic substring in s.

**Example 1:**

Input: s = "babad"

Output: "bab"

Explanation: "aba" is also a valid answer.

**Example 2:**

Input: s = "cbbd"

Output: "bb"

**Brute Force Approach:**

 **Generate All Substrings**

* Use two nested loops to generate all possible substrings of s.

 **Check for Palindrome**

* Use the isPalindrome function to check if a substring is a palindrome.

 **Keep Track of the Longest Palindrome**

* If a palindrome is found and its length is greater than the previous longest palindrome, update it.

public static boolean isPalindrome(String str) {  
 int left = 0, right = str.length() - 1;  
 while (left < right) {  
 if (str.charAt(left) != str.charAt(right)) {  
 return false;  
 }  
 left++;  
 right--;  
 }  
 return true;  
}  
  
public static String longestPalindrome(String s) {  
 int n = s.length();  
 String longest = "";  
   
 for (int i = 0; i < n; i++) {  
 String sub = "";   
 for (int j = i; j < n; j++) {  
 sub += s.charAt(j);   
 if (*isPalindrome*(sub) && sub.length() > longest.length()) {  
 longest = sub;  
 }  
 }  
 }  
 return longest;  
}

**Time Complexity:** **O(N³)** (due to generating substrings O(N²) and checking if they are palindromes O(N)). **Space Complexity:** O(1)

**Optimal Approach:**

Expanding around center.

Check for both odd and even size palindromes.

public static String longestPalindrome(String s)  
{  
 if(s.length() <= 1)  
 {  
 return s;  
 }  
 String maxStr = "";  
 for(int i=0;i<s.length()-1;i++)  
 {  
 String odd = *find*(s,i,i);  
 String even = *find*(s,i,i+1);  
  
 if(odd.length() > maxStr.length())  
 {  
 maxStr = odd;  
 }  
 if(even.length() > maxStr.length())  
 {  
 maxStr = even;  
 }  
 }  
 return maxStr;  
}  
  
public static String find(String s,int left,int right)  
{  
 String ans = "";  
 boolean flag = false;  
 while(left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right))  
 {  
 left--;  
 right++;  
 }  
 for(int i=left+1;i<right;i++)  
 {  
 ans += s.charAt(i);  
 }  
  
 return ans;  
}

**Time Complexity:** **O(N2)** **Space Complexity:** O(1)

**2.Count number of Substrings.**

Given a string of lowercase alphabets, count all possible substrings (not necessarily distinct) that has exactly k distinct characters.

**Example 1:**

Input: abc, k = 2

Output: 2

Possible substrings are {“ab”, “bc”}

**Example 2:**

Input: aba, k = 2

Output: 3

Possible substrings are {“ab”, “ba”, “aba”}

**Example 3:**

Input: aa, k = 1

Output: 3

Possible substrings are {“a”, “a”, “aa”}

**Brute Force Approach:**

Generate all possible substrings and check if it has exactly k distinct characters.

public int countSubstringsWithKDistinct(String s, int k) {  
 int count = 0;  
  
 // Generate all possible substrings  
 for (int i = 0; i < s.length(); i++) {  
 HashSet<Character> uniqueChars = new HashSet<>();  
 for (int j = i; j < s.length(); j++) {  
 uniqueChars.add(s.charAt(j));  
  
 // If exactly k distinct characters, increase count  
 if (uniqueChars.size() == k) {  
 count++;  
 }  
 // If more than k distinct characters, stop checking further  
 else if (uniqueChars.size() > k) {  
 break;  
 }  
 }  
 }  
  
 return count;  
}

**Time Complexity:** **O(N3)** **Space Complexity:** O(1)

**Optimal Approach:**

We use the formula:

exactlyK(s, K)=atMostK(s, K)−atMostK(s, K-1)\text{exactlyK(s, K)} = \text{atMostK(s, K)} - \text{atMostK(s, K-1)}exactlyK(s, K)=atMostK(s, K)−atMostK(s, K-1)

where:

* atMostK(s, K): Counts substrings with **at most** K distinct characters.
* atMostK(s, K-1): Counts substrings with **at most** K-1 distinct characters.
* Their difference gives us substrings with **exactly** K distinct characters.

public static int count(String a,int k)  
{  
 int left = 0;  
 int right = 0;  
 int count = 0;  
 HashMap<Character,Integer> freq = new HashMap<>();  
 while(right < a.length())  
 {  
 char Char = a.charAt(right);  
 freq.put(Char,freq.getOrDefault(Char,0)+1);  
 while(freq.size() > k)  
 {  
 char leftChar = a.charAt(left);  
 freq.put(leftChar,freq.get(leftChar)-1);  
 if(freq.get(leftChar) == 0)  
 freq.remove(leftChar);  
 left++;  
 }  
 count += right-left+1;  
 right++;  
 }  
 return count;  
}

**Why is count += (right - left + 1); used in the optimal approach?**

In the **Sliding Window (Two Pointers) approach**, the formula:

java

Copy code

count += (right - left + 1);

**counts all valid substrings ending at right that have at most K distinct characters**.

**Understanding the Formula**

At any step, the sliding window includes characters from left to right.

* Every substring that **starts anywhere from left to right** and **ends at right** is **valid**.
* The **number of these substrings** is **(right - left + 1)**.

**Example Walkthrough**

**Input:**

s = "aabac", k = 2

**Step-by-Step Expansion of Window**

| **Step** | **Window (left to right)** | **Unique Characters** | **Substrings Count (right - left + 1)** | **Total Count** |
| --- | --- | --- | --- | --- |
| r=0 | ["a"] | 1 | 1 | **1** |
| r=1 | ["aa"] | 1 | 2 | **3** |
| r=2 | ["aab"] | 2 | 3 | **6** |
| r=3 | ["aaba"] | 2 | 4 | **10** |
| r=4 | ["aabac"] | 3 (exceeds K, so shrink left) | - | - |

**Why Does This Work?**

For a **fixed right index**, the possible substrings are:

[left, right]

[left+1, right]

[left+2, right]

...

[right, right]

* There are (right - left + 1) substrings ending at right that satisfy the condition.
* This formula allows us to count efficiently **without generating substrings explicitly**.

**Time Complexity:** **O(N)** :Each character is added to and removed from the HashMap at most once.

**Space Complexity: O(K):** We store at most K keys in the HashMap.

**3.** **Sum of Beauty of all substring**

The beauty of a string is the difference in frequencies between the most frequent and least frequent characters.

For example, the beauty of "abaacc" is 3 - 1 = 2.

Given a string s, return the sum of beauty of all of its substrings.

**Example 1:**

Input: s = "aabcb"

Output: 5

Explanation: The substrings with non-zero beauty are ["aab","aabc","aabcb","abcb","bcb"], each with beauty equal to 1.

**Example 2:**

Input: s = "aabcbaa"

Output: 17  
  
**Brute Force (O(N³))**

* **Idea**: Generate all possible substrings, compute the frequency of each character, and find the beauty.
* **Steps**:
  1. Iterate over all possible substrings using two loops (O(N²)).
  2. Compute character frequencies for each substring (O(N)).
  3. Find the maximum and minimum frequency (O(26) ≈ O(1)) and compute beauty.
  4. Sum up all the beauties.
* public static int beautySum(String s) {  
   int n = s.length();  
   int totalBeauty = 0;  
    
   for (int i = 0; i < n; i++) {  
   for (int j = i; j < n; j++) {  
   int[] freq = new int[26]; // Frequency array for characters  
   for (int k = i; k <= j; k++) {  
   freq[s.charAt(k) - 'a']++;  
   }  
   totalBeauty += *calculateBeauty*(freq);  
   }  
   }  
   return totalBeauty;  
  }  
    
  private static int calculateBeauty(int[] freq) {  
   int maxFreq = 0, minFreq = Integer.*MAX\_VALUE*;  
   for (int f : freq) {  
   if (f > 0) {  
   maxFreq = Math.*max*(maxFreq, f);  
   minFreq = Math.*min*(minFreq, f);  
   }  
   }  
   return maxFreq - minFreq;  
  }
* **Time Complexity**: O(N3)O(N^3)O(N3) (since generating substrings takes O(N2)O(N^2)O(N2) and computing beauty takes O(N)O(N)O(N)).
* **Space Complexity**: O(1)O(1)O(1) (only a frequency array of size 26).

**. Optimized Brute Force (O(N²))**

* **Idea**: Instead of recalculating frequencies from scratch, maintain a frequency array that updates dynamically.
* **Steps**:
  1. Fix a **start index** and expand to the **end index** using an inner loop (O(N²)).
  2. Maintain a frequency array (O(26) ≈ O(1)) and update it as we expand.
  3. Calculate beauty in constant time using max and min frequencies.
* public int beautySum(String s)  
  {  
   int sum = 0;  
   String val = "";  
   for(int i=0;i<s.length();i++)  
   {  
   int[] hash = new int[26];  
   for(int j=i;j<s.length();j++)  
   {  
   hash[s.charAt(j)-'a']++;  
   sum += count(hash);  
   }  
   }  
   return sum;  
  }  
    
  public int count(int[] hash)  
  {  
   int maxi = 0;  
   int mini = 1000;  
   for(int i=0;i<hash.length;i++)  
   {  
   if(hash[i] > 0)  
   {  
   maxi = Math.*max*(maxi,hash[i]);  
   mini = Math.*min*(mini,hash[i]);  
   }  
   }  
   return maxi - mini;  
  }
* **Time Complexity**: O(N2)O(N^2)O(N2) (each substring is processed in O(1)O(1)O(1) after an update).
* **Space Complexity**: O(1)O(1)O(1) (frequency array of size 26).

**4.** **Implement Atoi**

Implement the myAtoi(string s) function, which converts a string to a 32-bit signed integer.

The algorithm for myAtoi(string s) is as follows:

Whitespace: Ignore any leading whitespace (" ").

Signedness: Determine the sign by checking if the next character is '-' or '+', assuming positivity if neither present.

Conversion: Read the integer by skipping leading zeros until a non-digit character is encountered or the end of the string is reached. If no digits were read, then the result is 0.

Rounding: If the integer is out of the 32-bit signed integer range [-231, 231 - 1], then round the integer to remain in the range. Specifically, integers less than -231 should be rounded to -231, and integers greater than 231 - 1 should be rounded to 231 - 1.

Return the integer as the final result.

**Example 1:**

Input: s = "42"

Output: 42

Explanation:

The underlined characters are what is read in and the caret is the current reader position.

Step 1: "42" (no characters read because there is no leading whitespace)

^

Step 2: "42" (no characters read because there is neither a '-' nor '+')

^

Step 3: "42" ("42" is read in)

^

**Example 2:**

Input: s = " -042"

Output: -42

Explanation:

Step 1: " -042" (leading whitespace is read and ignored)

^

Step 2: " -042" ('-' is read, so the result should be negative)

^

Step 3: " -042" ("042" is read in, leading zeros ignored in the result.

**Example 3:**

Input: s = "1337c0d3"

Output: 1337

Explanation:

Step 1: "1337c0d3" (no characters read because there is no leading whitespace)

^

Step 2: "1337c0d3" (no characters read because there is neither a '-' nor '+')

^

Step 3: "1337c0d3" ("1337" is read in; reading stops because the next character is a non-digit)

^

**Example 4:**

Input: s = "0-1"

Output: 0

^

**Example 5:**

Input: s = "words and 987"

Output: 0

**Approach:**

The goal is to parse the input string to extract an integer while handling:

Leading whitespace

Optional sign (+or-)

Conversion of digits into an integer

Clamping the result within the 32-bit integer range[-2^31, 2^31 - 1]

Approach :

Ignore Leading Whitespace: Skip over any leading spaces in the string.

Sign Detection: Check for a+or-sign immediately after the whitespace.

Convert Digits to Integer: Start reading consecutive digits, stopping when a non-digit is encountered.

Clamp to 32-bit Integer Range: If the integer exceeds2^31 - 1or goes below-2^31, clamp it to fit within these bounds.

Return the Result.

Solution Steps :

Define the bounds for 32-bit integers.

Remove leading whitespace using indexing.

Set the sign based on the first character, if it's+or-.

Traverse through each character of the string to build the integer.

If an overflow is detected during the process, clamp the result.

res <MAX INTEGER  
res \* 10 + digit < MAX INTEGER  
res < (MAX INTEGER-digit) / 10

public int myAtoi(String s)  
{  
 int i = 0;  
 int n = s.length();  
 int sign = 1;  
 int result = 0;  
  
 while(i<n && s.charAt(i) == ' ')i++;  
  
 if(i<n && (s.charAt(i) == '-' || s.charAt(i) == '+'))  
 {  
 sign = s.charAt(i) == '-'?-1:1;  
 i++;  
 }  
  
 while(i<n && s.charAt(i)-'0'>=0 && s.charAt(i)-'0'<=9)  
 {  
 int digit = s.charAt(i)-'0';  
 if(result > (Integer.*MAX\_VALUE*-digit)/10)  
 {  
 return sign == 1?Integer.*MAX\_VALUE*:Integer.*MIN\_VALUE*;  
 }  
 result = result \* 10 + digit;  
 i++;  
 }  
  
 return sign\*result;  
}

**Time Complexity: (O(n)),** where (n) is the length of the input string, as each character is processed once.

**Space Complexity: (O(1)),** as only a few variables are used to store intermediate results.

**5.** **Sort Characters by frequency**

Given a string s, sort it in decreasing order based on the frequency of the characters. The frequency of a character is the number of times it appears in the string.

Return the sorted string. If there are multiple answers, return any of them.

**Example 1:**

Input: s = "tree"

Output: "eert"

**Example 2:**

Input: s = "cccaaa"

Output: "aaaccc"

**Example 3:**

Input: s = "Aabb"

Output: "bbAa"

**Approach 1:**

Using HashMap and Sorting

Steps:

Count the Frequency

Use a HashMap<Character, Integer> to store character frequencies.

Sort the Characters by Frequency

Store the characters in a list.

Sort the list based on the values in the HashMap, in decreasing order.

Build the Result String

Iterate over the sorted list and append each character to the result string based on its frequency.

public String frequencySort(String s)  
{  
 Map<Character,Integer> freq = new HashMap<>();  
 for(char c:s.toCharArray())  
 {  
 freq.put(c,freq.getOrDefault(c,0)+1);  
 }  
  
 List<Character> charList = new ArrayList<>(freq.keySet());  
 charList.sort((a,b) -> freq.get(b) - freq.get(a));  
  
 StringBuilder result = new StringBuilder();  
 for(char c:charList)  
 {  
 result.append(String.*valueOf*(c).repeat(freq.get(c)));  
 }  
  
 return result.toString();  
}

**Time Complexity:** O(N log N) due to sorting, where N is the length of the string.

**Space Complexity:** O(N)

**Approach 2:**

Using Priority Queue (Max Heap)

Steps:

Count the Frequency

Use a HashMap<Character, Integer> to count occurrences of each character.

Use a Max Heap

A priority queue (PriorityQueue<Map.Entry<Character, Integer>>) is used to store character-frequency pairs.

The heap is ordered by frequency in decreasing order.

Build the Result String

Extract elements from the heap and append them to a StringBuilder.

public String frequencySort1(String s)  
{  
 StringBuilder result = new StringBuilder();  
 Map<Character,Integer> freq = new HashMap<>();  
 for(char c:s.toCharArray())  
 {  
 freq.put(c,freq.getOrDefault(c,0)+1);  
 }  
  
 PriorityQueue<Map.Entry<Character,Integer>> maxHeap = new PriorityQueue<>((a, b)->b.getValue() - a.getValue());  
 maxHeap.addAll(freq.entrySet());  
  
 while(!maxHeap.isEmpty())  
 {  
 Map.Entry<Character,Integer> entry = maxHeap.poll();  
 result.append(String.*valueOf*(entry.getKey()).repeat(entry.getValue()));  
 }  
  
 return result.toString();  
}

**Time Complexity:** O(N log K), where K is the number of unique characters.

**Space Complexity:** O(N)

**Approach 3:**

**Using Bucket Sort (Optimized for Large Strings)**

Steps:

Count the Frequency

Use a HashMap<Character, Integer> to store character frequencies.

Use Buckets to Store Characters by Frequency

Create an array of lists (List<Character>[] buckets).

Each index represents a frequency, and characters with that frequency are stored in that index.

Build the Result String

Iterate from the highest frequency index to the lowest and append characters to the result.

public static String frequencySortUsingBucketSort(String s) {  
 Map<Character, Integer> freqMap = new HashMap<>();  
 int maxFreq = 0;  
 for (char c : s.toCharArray()) {  
 freqMap.put(c, freqMap.getOrDefault(c, 0) + 1);  
 maxFreq = Math.*max*(maxFreq, freqMap.get(c));  
 }  
  
 List<Character>[] buckets = new List[maxFreq + 1];  
 for (int i = 0; i <= maxFreq; i++) {  
 buckets[i] = new ArrayList<>();  
 }  
 for (Map.Entry<Character, Integer> entry : freqMap.entrySet()) {  
 buckets[entry.getValue()].add(entry.getKey());  
 }  
  
 StringBuilder result = new StringBuilder();  
 for (int i = maxFreq; i > 0; i--) {  
 for (char c : buckets[i]) {  
 result.append(String.*valueOf*(c).repeat(i));  
 }  
 }  
  
 return result.toString();  
}

**Time Complexity : O(N)**, because sorting is avoided.

**Space Complexity**: **O(N)**

**6.Roman Number to Integer and vice versa.**

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

Symbol Value

I 1

V 5

X 10

L 50

C 100

D 500

M 1000

For example, 2 is written as II in Roman numeral, just two ones added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

I can be placed before V (5) and X (10) to make 4 and 9.

X can be placed before L (50) and C (100) to make 40 and 90.

C can be placed before D (500) and M (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer.

**Example 1:**

Input: s = "III"

Output: 3

Explanation: III = 3.

**Example 2:**

Input: s = "LVIII"

Output: 58

Explanation: L = 50, V= 5, III = 3.

**Example 3:**

Input: s = "MCMXCIV"

Output: 1994

Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.

public int romanToInt(String s)  
{  
 int curr = 0;  
 int prev = 0;  
 int ans = sendValue(s.charAt(0));  
 for(int i=1;i<s.length();i++)  
 {  
 prev = sendValue(s.charAt(i-1));  
 curr = sendValue(s.charAt(i));  
 if(curr <= prev)  
 {  
 ans += curr;  
 }  
 else  
 {  
 ans -= prev;  
 ans += curr - prev;  
 }  
 }  
 return ans;  
}  
  
  
public int sendValue(char c)  
{  
 switch(c)  
 {  
 case 'I':  
 return 1;  
 case 'V':  
 return 5;  
 case 'X':  
 return 10;  
 case 'L':  
 return 50;  
 case 'C':  
 return 100;  
 case 'D':  
 return 500;  
 case 'M':  
 return 1000;  
 default:  
 return 0;  
 }  
}

**Time Complexity : O(N).**

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

**7.** **Maximum Nesting Depth of Paranthesis.**

Given a valid parentheses string s, return the nesting depth of s. The nesting depth is the maximum number of nested parentheses.

**Example 1:**

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

Output: 3

**Example 2:**

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

Output: 3

**Example 3:**

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

Output: 3

public int maxDepth(String s)  
{  
 int count = 0;  
 int maxi = 0;  
 for(int i=0;i<s.length();i++)  
 {  
 if(s.charAt(i) == '(')  
 {  
 count++;  
 maxi = Math.*max*(maxi,count);  
 }  
 else if(s.charAt(i) == ')')  
 {  
 count--;  
 }  
 }  
 return maxi;  
}

**Time Complexity : O(N)**

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