**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:**

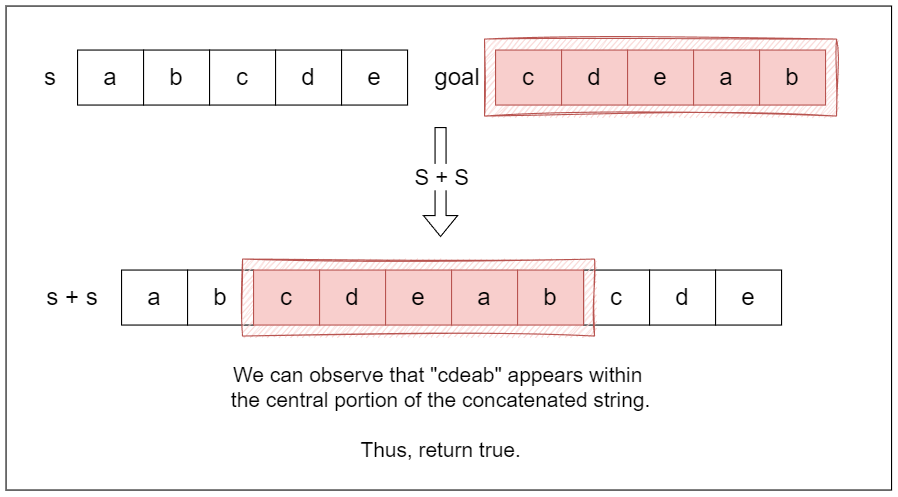
Rotate the string each time to left , and check if the strings are equal, you can rotate it to the size of the string .

**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.