**227. Basic Calculator II**

Given a string s which represents an expression, *evaluate this expression and return its value*.

The integer division should truncate toward zero.

You may assume that the given expression is always valid. All intermediate results will be in the range of [-231, 231 - 1].

**Note:** You are not allowed to use any built-in function which evaluates strings as mathematical expressions, such as eval().

**Input:** s = "3+2\*2"

**Output:** 7

**Input:** s = "3\*2+5/2-10 "

**Output:** 5

Sol:

Take 2 variable prevSign and no….prevSign will store the prev operator…and no will store the no in string

Iterate the String and if we find the character is operator then we check prevSign if it is

**+ then push no to stack**

**- then push -no to stack**

**\* then push(stack.pop() \* no)**

**/ then push(stack.pop() / no)**

And in last reinitialize these variable with

prevSign = ch;

and no = 0;

string s = "3\*2+5/2-10 ";

**prevSign = +**

**no = 0**

now iterate the string

1st char is 3 and it’s digit so we will form digit from string and assign its value to no variable

**prevSign = +**

**No = 3**

2nd char is \* which is operator so we will check the prevSign which is +

If it is + then we will simply push no to stack. **Stack = 3**

And assign curr char to prevSign and make no to 0

**prevSign = \***

**no = 0;**

3rd char is 2 which is digit assign its value to no so no = 2 now.

**prevSign = \***

**no = 2;**

4th char is + which is operator so we will check prevSign which is \*

If it is \* then first pop the value from stack and multiply with no and then push the resultant value to stack

So stack.pop() = 3 and no is 2 , We will multiply 3\*2 = 6 and then push 6 to stack, **stack = 6**

**prevSign = +**

**no = 0;**

5th char is 5 which is digit so assign this value to no

**prevSign = +**

**no = 5;**

6th char is / so we will check the prevSign which is +

So we will simply push the no to stack….**stack = 5, 6**

**prevChar = /**

**no = 0**

6th char is 2 which is digit so assign this value to no

**prevSign = /**

**No = 2**

7th char is – which is operator so we will check prevSign wich is /

So if operator is / then first we will do the pop

Stack.po() = 5 and divide this with no

So 5/2 = 2 so will push this 2 to stack, **stack = 2, 6**

**prevSign = -**

**n = 0**

8th char is 1 and then 0 so form the digit from it and assign to so no = 10

**prevSign = -**

**n = 1 0**

This is the last char so will go inside the if block and check the prevSign

prevSign = - so we will simply push -no which is -10 to stack. **Stack = -10, 2, 6**

Now we pop the element one by one and add to result

Result will be -10 + 2 + 6 = -2 and this is the result

**241. Different Ways to Add Parentheses**

Given a string expression of numbers and operators, return *all possible results from computing all the different possible ways to group numbers and operators*. You may return the answer in **any order**.

The test cases are generated such that the output values fit in a 32-bit integer and the number of different results does not exceed 104.

**Input:** expression = "2\*3-4\*5"

**Output:** [-34,-14,-10,-10,10]

**Explanation:**

(2\*(3-(4\*5))) = -34

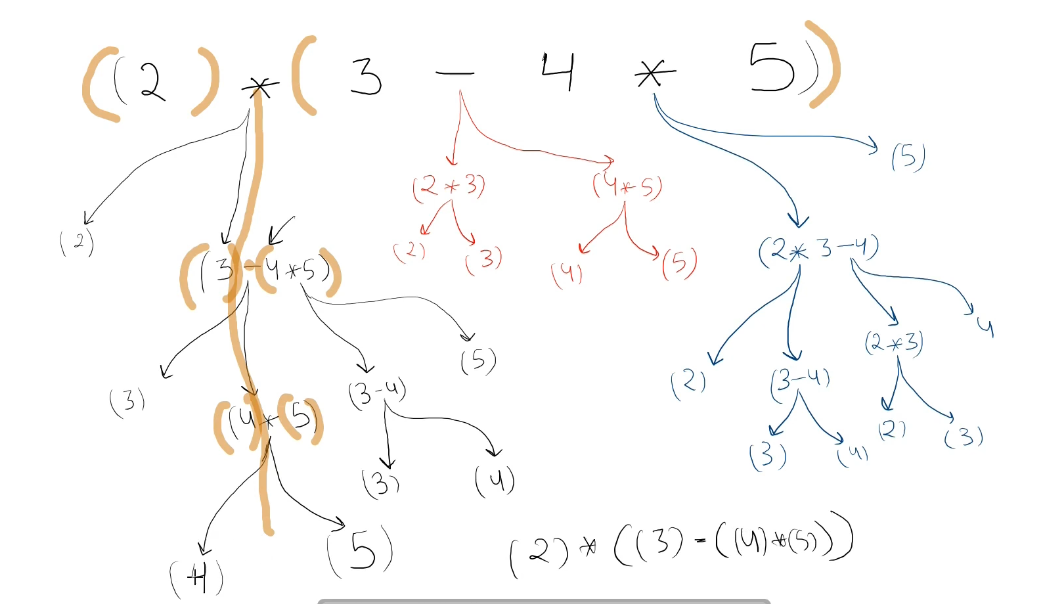
((2\*3)-(4\*5)) = -14

((2\*(3-4))\*5) = -10

(2\*((3-4)\*5)) = -10

(((2\*3)-4)\*5) = 10

Sol:



Iterate String, for every operator divide the String in 2 parts. String before operator and String after operator

Then do the recursive call for each left and right substring.

Suppose we store the left recursive call result into a and right recursive call result into b

Then we will iterate both result in loop

We will check if ch == ‘+’ then we will add x + y

We will check if ch == ‘-‘ then we will add x – y

We will check if ch = ‘\*’ then we will add x\*y

After all loop we will check if expression contains only number then res size will be 0

In this case we will simply add that number into result.

**22. Generate Parentheses**

Given n pairs of parentheses, write a function to *generate all combinations of well-formed parentheses*.

**Input:** n = 3

**Output:** ["((()))","(()())","(())()","()(())","()()()"]

**49. Group Anagrams**

Given an array of strings strs, group **the anagrams** together. You can return the answer in **any order**.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

**Input:** strs = ["eat","tea","tan","ate","nat","bat"]

**Output:** [["bat"],["nat","tan"],["ate","eat","tea"]]

**779. K-th Symbol in Grammar**

We build a table of n rows (**1-indexed**). We start by writing 0 in the 1st row. Now in every subsequent row, we look at the previous row and replace each occurrence of 0 with 01, and each occurrence of 1 with 10.

* For example, for n = 3, the 1st row is 0, the 2nd row is 01, and the 3rd row is 0110.

Given two integer n and k, return the kth (**1-indexed**) symbol in the nth row of a table of n rows.

**Input:** n = 1, k = 1

**Output:** 0

**Explanation:** row 1: 0

**Input:** n = 2, k = 1

**Output:** 0

**Explanation:**

row 1: 0

row 2: 01

**Sol:**

Draw grammar with n = 4 and k = 6

**0**

1. **1**

**0 1 1 0**

**0 1 1 0 1 0 0 1**

This is the generated grammar with n = 4. Now we need to find out value at index (4, 6) that is 0

If we see closely the grammar length is increasing 2 times if we change n to n+1. At n = 1 we have 1 element in grammar, at n = 2 we have 2, at n = 3 we have 4 and at n = 4 we have 8 elements.

If we observe closely n = 3 and n = 4 we see that

First Half of the 4th row is equal to 3rd row and another half of 4th row is just ^(XOR) of 1st half.

We divide the row in 2 half. If k <= mid then we can do the recursive call with n-1, k parameter. If k > mid then we just need to add the 1 ^ operator to the 1st half method calls.

**179. Largest Number**

Given a list of non-negative integers nums, arrange them such that they form the largest number and return it.

Since the result may be very large, so you need to return a string instead of an integer.

**Input:** nums = [3,30,34,5,9]

**Output:** "9534330"

**784. Letter Case Permutation**

Given a string s, you can transform every letter individually to be lowercase or uppercase to create another string.

Return *a list of all possible strings we could create*. Return the output in **any order**.

**Input:** s = "a1b2"

**Output:** ["a1b2","a1B2","A1b2","A1B2"]

Sol:

We will check whether char is digit or letter. If it is letter then we will make choice for small case and capital case.

If it is digit then we will have only one choice to add the digit. Once we add the digit then that digit is processed so remove the processed char from the input.

**5. Longest Palindromic Substring**

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

**Input:** s = "babad"

**Output:** "bab"

**Explanation:** "aba" is also a valid answer.

Sol:

Consider each character of the String as center of the string. And expand it in left and right direction till left and right char is same. This string can be of odd or even length. If it is odd length then ith char will be the center element. If string length is odd then I and i+1 both element will be the center.

**318. Maximum Product of Word Lengths**

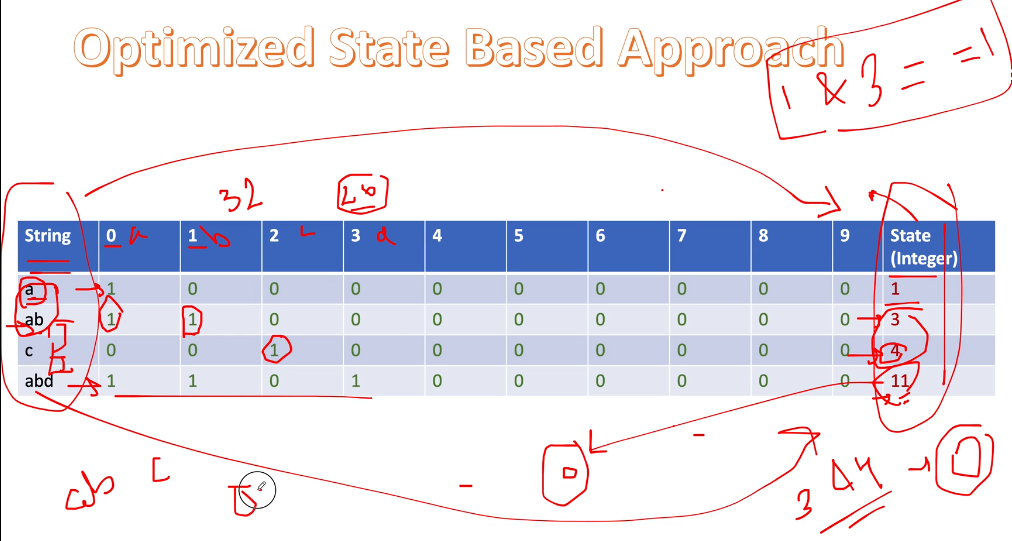
Given a string array words, return *the maximum value of* length(word[i]) \* length(word[j]) *where the two words do not share common letters*. If no such two words exist, return 0.

**Input:** words = ["abcw","baz","foo","bar","xtfn","abcdef"]

**Output:** 16

**Explanation:** The two words can be "abcw", "xtfn".

Sol:



Here we will use bitwise logic. In English we can have 26 char and in bitwise we have 32 bits. So we can accommodate the English words char into bit mask.

In this approach we assume that a points to 0, b points to 1, c points to 2 and so on…

If we want to find out if there is any char common in the words then we will do & operator and if it returns 0 then it means both words don’t have any common char. If & operator between both words return 1 then it means both words have common chars.

**1647. Minimum Deletions to Make Character Frequencies Unique**

A string s is called **good** if there are no two different characters in s that have the same **frequency**.

Given a string s, return*the****minimum****number of characters you need to delete to make*s***good****.*

The **frequency** of a character in a string is the number of times it appears in the string. For example, in the string "aab", the **frequency** of 'a' is 2, while the **frequency** of 'b' is 1.

**Input:** s = "aaabbbcc"

**Output:** 2

**Explanation:** You can delete two 'b's resulting in the good string "aaabcc".

Another way it to delete one 'b' and one 'c' resulting in the good string "aaabbc".

Sol:

Find the count of each char and put it in map. Then take a set to store and check the unique count. If set contains the count then delete the counts till set does not contain that count.

**1249. Minimum Remove to Make Valid Parentheses**

Given a string s of '(' , ')' and lowercase English characters.

Your task is to remove the minimum number of parentheses ( '(' or ')', in any positions ) so that the resulting *parentheses string* is valid and return **any** valid string.

Formally, a *parentheses string* is valid if and only if:

* It is the empty string, contains only lowercase characters, or
* It can be written as AB (A concatenated with B), where A and B are valid strings, or
* It can be written as (A), where A is a valid string.

Sol:

**1347. Minimum Number of Steps to Make Two Strings Anagram**

You are given two strings of the same length s and t. In one step you can choose **any character** of t and replace it with **another character**.

Return *the minimum number of steps* to make t an anagram of s.

An **Anagram** of a string is a string that contains the same characters with a different (or the same) ordering.

**Input:** s = "leetcode", t = "practice"

**Output:** 5

**Explanation:** Replace 'p', 'r', 'a', 'i' and 'c' from t with proper characters to make t anagram of s.

**792. Number of Matching Subsequences**

Given a string s and an array of strings words, return *the number of* words[i] *that is a subsequence of* s.

A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

* For example, "ace" is a subsequence of "abcde".
* **Input:** s = "abcde", words = ["a","bb","acd","ace"]
* **Output:** 3
* **Explanation:** There are three strings in words that are a subsequence of s: "a", "acd", "ace".

Sol:

Write the helper method to find out if string word is subsequence of s or not.

Then take 2 set. One set for subseq and another for notsubseq.

If notseq contains word then continue.

If suseq contains word then count++

Else…it means word can be subseq or not subseq

If it is subseq then count++ and add into subseq set else add into notsubset

**647. Palindromic Substrings**

Given a string s, return *the number of****palindromic substrings****in it*.

A string is a **palindrome** when it reads the same backward as forward.

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

**Input:** s = "aaa"

**Output:** 6

**Explanation:** Six palindromic strings: "a", "a", "a", "aa", "aa", "aaa".

**763. Partition Labels**

You are given a string s. We want to partition the string into as many parts as possible so that each letter appears in at most one part.

Note that the partition is done so that after concatenating all the parts in order, the resultant string should be s.

Return *a list of integers representing the size of these parts*.

**Input:** s = "ababcbacadefegdehijhklij"

**Output:** [9,7,8]

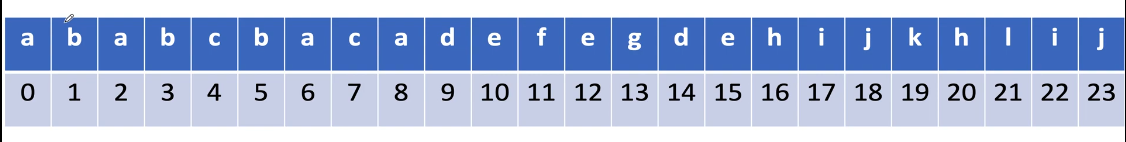
**Explanation:**

The partition is "ababcbaca", "defegde", "hijhklij".

This is a partition so that each letter appears in at most one part.

A partition like "ababcbacadefegde", "hijhklij" is incorrect, because it splits s into less parts.

Sol:



We find out the lastIndex of the each char.

a ------ 8, b-------- 5, c ------ 7, d ---------- 14, e ---------- 15, f ---------11, g ---------- 13, h --------- 19, I -----------22

j ---------23, k ----------20, l -----------21

Once we find out the lastIndex of each char, we will take 2 pointers I and j starting from 0

Find out the lastIndex of the char.

Suppose char is ‘a’ so its lastIndex is 8

Now we will increment the j counter and check the lastIndex of the char present at j++ index.

So b lastIndex is 5 which is lesser than 8 . we move to next char that is ‘a’ again. so just increment j counter.

If in any scenario end is greater than then we will update the with latest greater value.

And substring length will be j – i + 1 . will add this into result

And then we will assign I = j + 1 for the next substring.

**1209. Remove All Adjacent Duplicates in String II**

You are given a string s and an integer k, a k **duplicate removal** consists of choosing k adjacent and equal letters from s and removing them, causing the left and the right side of the deleted substring to concatenate together.

We repeatedly make k **duplicate removals** on s until we no longer can.

Return *the final string after all such duplicate removals have been made*. It is guaranteed that the answer is **unique**.

**Input:** s = "deeedbbcccbdaa", k = 3

**Output:** "aa"

**Explanation:**

First delete "eee" and "ccc", get "ddbbbdaa"

Then delete "bbb", get "dddaa"

Finally delete "ddd", get "aa"

**316. Remove Duplicate Letters**

Given a string s, remove duplicate letters so that every letter appears once and only once. You must make sure your result is **the smallest in lexicographical order** among all possible results.

**Input:** s = "bcabc"

**Output:** "abc"

Sol:

Find the last occurrence index of each element.

b ----- 3, c ----4, a -------2

We will take a array of Boolean type to check that char is visited or not. As we have to keep only element of each char. We don’t have to add duplicate char.

first iteration… add b into stack, stack will contain b

2nd iteration c will come. c is not visited, we mark it as visited.

Will check Stack.peek() > index of char……..and lastIndex[index of char] > i

As Stack.peek() > index will return false

So we will add this in stack. So stack will contain now bc

3rd iteration a will come. it is not in stack. We mark it visited.

As c > a and lastIndex of c which is 4 is greater than current index 2. Here both condition satisfied so we will pop the stack and mark the index of the poped element index to false.

Will do this for all char.

**402. Remove K Digits**

Given string num representing a non-negative integer num, and an integer k, return *the smallest possible integer after removing* k *digits from* num.

**Input:** num = "1432219", k = 3

**Output:** "1219"

**Explanation:** Remove the three digits 4, 3, and 2 to form the new number 1219 which is the smallest.

Sol:

Put every char into stack. Before putting it into stack. If stack.peek() > current element and k > 0 then pop the element from stack and do k- -

**767. Reorganize String**

Given a string s, rearrange the characters of s so that any two adjacent characters are not the same.

Return *any possible rearrangement of* s *or return* "" *if not possible*.

**Input:** s = "aab"

**Output:** "aba"

**Input:** s = "aaab"

**Output:** ""

Sol:

Store the count of the char into map. And then add all the char into maxheap.

While maxheap Size > 1 then we will remove 2 most frequent element from it. Decrement its count by 1 and we will append it one by one into maxheap if its count > 0.

At last we will have only one element in maxheap…check if it is having count more than1 then return empty string else add this char to result.

**1041. Robot Bounded In Circle**

On an infinite plane, a robot initially stands at (0, 0) and faces north. Note that:

* The **north direction** is the positive direction of the y-axis.
* The **south direction** is the negative direction of the y-axis.
* The **east direction** is the positive direction of the x-axis.
* The **west direction** is the negative direction of the x-axis.

The robot can receive one of three instructions:

* "G": go straight 1 unit.
* "L": turn 90 degrees to the left (i.e., anti-clockwise direction).
* "R": turn 90 degrees to the right (i.e., clockwise direction).

The robot performs the instructions given in order, and repeats them forever.

Return true if and only if there exists a circle in the plane such that the robot never leaves the circle.

**Input:** instructions = "GGLLGG"

**Output:** true

**Explanation:** The robot is initially at (0, 0) facing the north direction.

"G": move one step. Position: (0, 1). Direction: North.

"G": move one step. Position: (0, 2). Direction: North.

"L": turn 90 degrees anti-clockwise. Position: (0, 2). Direction: West.

"L": turn 90 degrees anti-clockwise. Position: (0, 2). Direction: South.

"G": move one step. Position: (0, 1). Direction: South.

"G": move one step. Position: (0, 0). Direction: South.

Repeating the instructions, the robot goes into the cycle: (0, 0) --> (0, 1) --> (0, 2) --> (0, 1) --> (0, 0).

Based on that, we return true.

**856. Score of Parentheses**

Given a balanced parentheses string s, return *the****score****of the string*.

The **score** of a balanced parentheses string is based on the following rule:

* "()" has score 1.
* AB has score A + B, where A and B are balanced parentheses strings.
* (A) has score 2 \* A, where A is a balanced parentheses string.
* **Input:** s = "(()())()"
* **Output:** 5

Sol:

**451. 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*.

**Input:** s = "tree"

**Output:** "eert"

**Explanation:** 'e' appears twice while 'r' and 't' both appear once.

So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a valid answer.

**Sol:**

Take a map and store the count of each char.

Then we will take

Queue<Character> maxHeap = **new** PriorityQueue<>((a, b) -> map.get(b) - map.get(a));

In this maxheap add all the key of the map.

Then iterate the maxheap till it is not empty.

Remove char from maxheap one by one and takes its count from the map and then print the char as many times as it is in map.

**678. Valid Parenthesis String**

Given a string s containing only three types of characters: '(', ')' and '\*', return true *if* s *is****valid***.

The following rules define a **valid** string:

* Any left parenthesis '(' must have a corresponding right parenthesis ')'.
* Any right parenthesis ')' must have a corresponding left parenthesis '('.
* Left parenthesis '(' must go before the corresponding right parenthesis ')'.
* '\*' could be treated as a single right parenthesis ')' or a single left parenthesis '(' or an empty string "".