**438. Find All Anagrams in a String**

Given two strings s and p, return *an array of all the start indices of*p*'s anagrams in*s. You may 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:** s = "cbaebabacd", p = "abc"

**Output:** [0,6]

**Explanation:**

The substring with start index = 0 is "cba", which is an anagram of "abc".

The substring with start index = 6 is "bac", which is an anagram of "abc".

**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

**394. Decode String**

Given an encoded string, return its decoded string.

The encoding rule is: k[encoded\_string], where the encoded\_string inside the square brackets is being repeated exactly k times. Note that k is guaranteed to be a positive integer.

You may assume that the input string is always valid; there are no extra white spaces, square brackets are well-formed, etc. Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, k. For example, there will not be input like 3a or 2[4].

**Input:** s = "3[a]2[bc]"

**Output:** "aaabcbc"

**Input:** s = "3[a2[c]]"

**Output:** "accaccacc"

**Explanation:**

We will take 2 stack.

One stack to store the frequency on string and other string itself.

Will take 2 variables. On variable to store frequency and other to store result.

K = 0

Res = “”

Iterate the String

In 1st iteration char is 3

char is digit no need to do anything..just stor this value to variable k. so k = 3

2nd char is ‘[‘, we will push no to numStack and result string to strStack

**numStack = 3**

**strStack = “”**

**reinitialize k and res for next opening bracket**

**k = 0;**

**res = new StringBuilder();**

3rd char is a, we will append char to temp = res

4th char is ‘]’ so we will take the take this res, and do the pop from the numStack which will give 3

Now pop from the strStack which will give empty StringBuilder.

Now append temp into result k times.

So result will be aaa

5th char is 2 so store this to a variable k = 2

6th char is [, we will push current num k to numStack. So numStack = 2

And push res into strStack….so strStack will contain……aaa

Int k = 0

Res = new StringBuilder()

7th char is b we will append this to res…res = b

8th char is c we will append this to res …. Res = bc

9th char is ‘]’

We will store the res into temp…………so temp = bc

Now we will pop frequency from frequency stakc…it will return 2

Then we will pop result from strStack which will return aaa

No append bc 2 time to result aaa

So appended string is aaabcbc and this is the result.

Steps:

Take 2 stack,,, 1 stack for frequency and other for result. stack name is freStack, resStack

Take 2 variables…1 for frequency and other for result. Variable name is k, res

1) if char is digit …..form the number from string and store its value to k

2) char is letter……. Append the char to res

3) if char is [ then push frequency and res to freStack, resStack resp.

4) if char is ] then store the res into temp....this is the whole string inside [ ] we need to repeat this string k times.. now pop the element from freStack and resStack.

Int count = freStack.pop()

StringBuilder res = resStack.pop();

Append the temp into res count times.

**211. Design Add and Search Words Data Structure**

Design a data structure that supports adding new words and finding if a string matches any previously added string.

Implement the WordDictionary class:

* WordDictionary() Initializes the object.
* void addWord(word) Adds word to the data structure, it can be matched later.
* bool search(word) Returns true if there is any string in the data structure that matches word or false otherwise. word may contain dots '.' where dots can be matched with any letter.
* WordDictionary wordDictionary = new WordDictionary();
* wordDictionary.addWord("bad");
* wordDictionary.addWord("dad");
* wordDictionary.addWord("mad");
* wordDictionary.search("pad"); // return False
* wordDictionary.search("bad"); // return True
* wordDictionary.search(".ad"); // return True
* wordDictionary.search("b.."); // return True

Sol:

Use Trie data structure.

**1396. Design Underground System**

An underground railway system is keeping track of customer travel times between different stations. They are using this data to calculate the average time it takes to travel from one station to another.

Implement the UndergroundSystem class:

* void checkIn(int id, string stationName, int t)
  + A customer with a card ID equal to id, checks in at the station stationName at time t.
  + A customer can only be checked into one place at a time.
* void checkOut(int id, string stationName, int t)
  + A customer with a card ID equal to id, checks out from the station stationName at time t.
* double getAverageTime(string startStation, string endStation)
  + Returns the average time it takes to travel from startStation to endStation.
  + The average time is computed from all the previous traveling times from startStation to endStation that happened **directly**, meaning a check in at startStation followed by a check out from endStation.
  + The time it takes to travel from startStation to endStation **may be different** from the time it takes to travel from endStation to startStation.
  + There will be at least one customer that has traveled from startStation to endStation before getAverageTime is called.

You may assume all calls to the checkIn and checkOut methods are consistent. If a customer checks in at time t1 then checks out at time t2, then t1 < t2. All events happen in chronological order.

UndergroundSystem undergroundSystem = new UndergroundSystem();

undergroundSystem.checkIn(45, "Leyton", 3);

undergroundSystem.checkIn(32, "Paradise", 8);

undergroundSystem.checkIn(27, "Leyton", 10);

undergroundSystem.checkOut(45, "Waterloo", 15); // Customer 45 "Leyton" -> "Waterloo" in 15-3 = 12

undergroundSystem.checkOut(27, "Waterloo", 20); // Customer 27 "Leyton" -> "Waterloo" in 20-10 = 10

undergroundSystem.checkOut(32, "Cambridge", 22); // Customer 32 "Paradise" -> "Cambridge" in 22-8 = 14

undergroundSystem.getAverageTime("Paradise", "Cambridge"); // return 14.00000. One trip "Paradise" -> "Cambridge", (14) / 1 = 14

undergroundSystem.getAverageTime("Leyton", "Waterloo"); // return 11.00000. Two trips "Leyton" -> "Waterloo", (10 + 12) / 2 = 11

undergroundSystem.checkIn(10, "Leyton", 24);

undergroundSystem.getAverageTime("Leyton", "Waterloo"); // return 11.00000

undergroundSystem.checkOut(10, "Waterloo", 38); // Customer 10 "Leyton" -> "Waterloo" in 38-24 = 14

undergroundSystem.getAverageTime("Leyton", "Waterloo"); // return 12.00000. Three trips "Leyton" -> "Waterloo", (10 + 12 + 14) / 3 = 12

**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

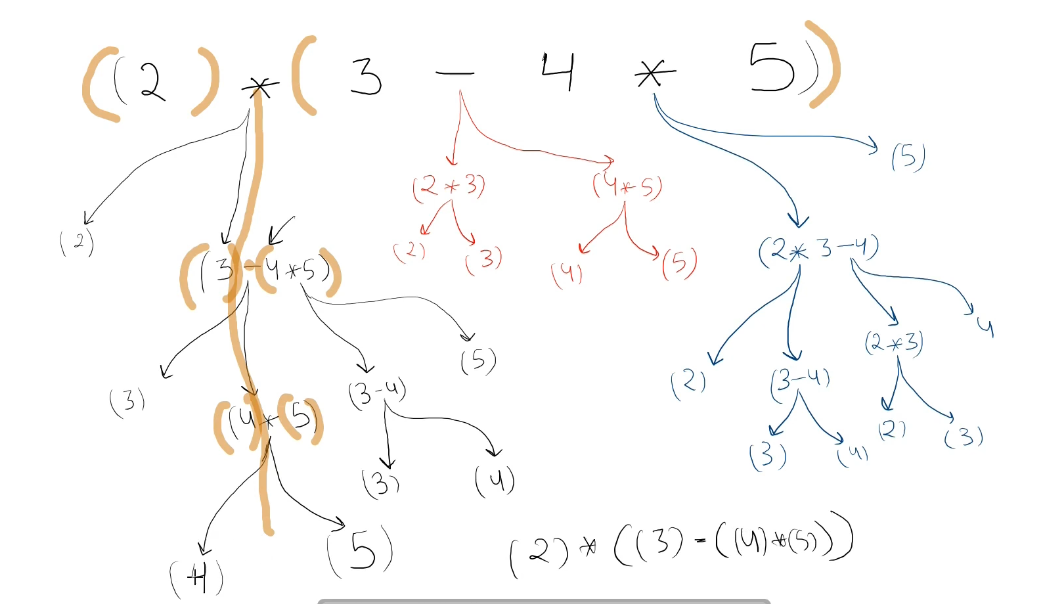
((2\*3)-(4\*5)) = -14

((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.

**890. Find and Replace Pattern**

Given a list of strings words and a string pattern, return *a list of* words[i] *that match* pattern. You may return the answer in **any order**.

A word matches the pattern if there exists a permutation of letters p so that after replacing every letter x in the pattern with p(x), we get the desired word.

Recall that a permutation of letters is a bijection from letters to letters: every letter maps to another letter, and no two letters map to the same letter.

**Input:** words = ["abc","deq","mee","aqq","dkd","ccc"], pattern = "abb"

**Output:** ["mee","aqq"]

**Explanation:** "mee" matches the pattern because there is a permutation {a -> m, b -> e, ...}.

"ccc" does not match the pattern because {a -> c, b -> c, ...} is not a permutation, since a and b map to the same letter.

Sol:

Iterate the word array and for each word check if it is isomorphic to patter or not. If it is then add that word to list.

**22. Generate Parentheses**

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

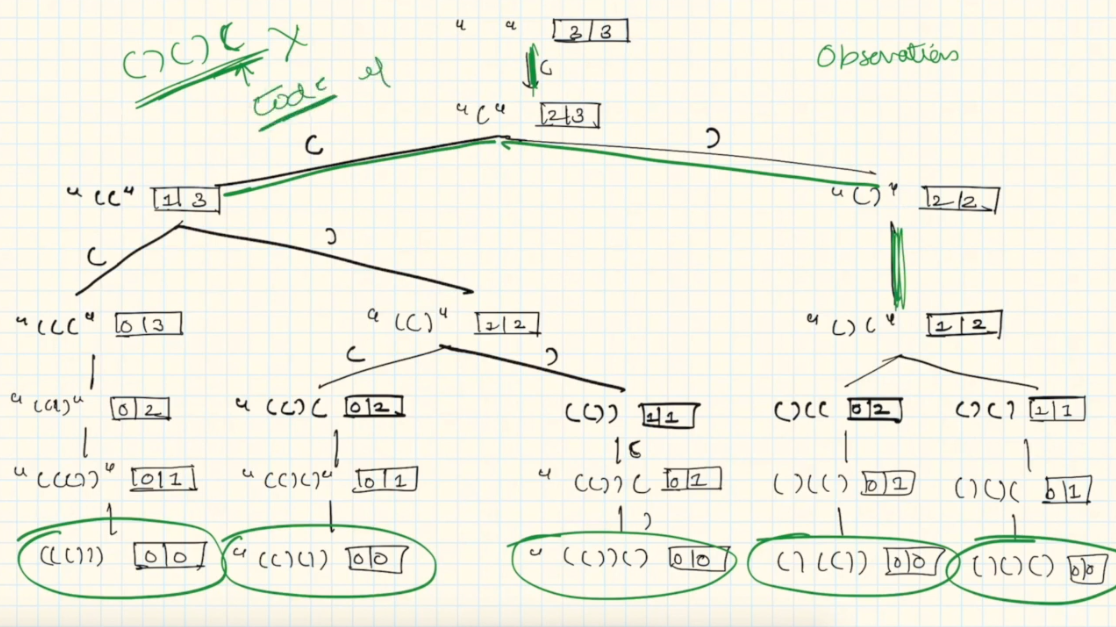
**Input:** n = 3

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

Here we are given n = 3;

It means we have to generate all the combination with 3 open brackets and 3 closed brackets.

Draw the recursive tree diagram with output = “”, open = 3, and closed = 3



In the diagram we can see that opening bracket is always coming. Closing bracket is coming only when closing bracket count is greater than opening bracket count. And when open == closed == 0 then we are getting the result.

**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"]]

**208. Implement Trie (Prefix Tree)**

A **[trie](https://en.wikipedia.org/wiki/Trie" \t "_blank)** (pronounced as "try") or **prefix tree** is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.

Implement the Trie class:

* Trie() Initializes the trie object.
* void insert(String word) Inserts the string word into the trie.
* boolean search(String word) Returns true if the string word is in the trie (i.e., was inserted before), and false otherwise.
* boolean startsWith(String prefix) Returns true if there is a previously inserted string word that has the prefix prefix, and false otherwise.
* Trie trie = new Trie();
* trie.insert("apple");
* trie.search("apple"); // return True
* trie.search("app"); // return False
* trie.startsWith("app"); // return True
* trie.insert("app");
* trie.search("app"); // return True

**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.

**17. Letter Combinations of a Phone Number**

Given a string containing digits from 2-9 inclusive, return all possible letter combinations that the number could represent. Return the answer in **any order**.

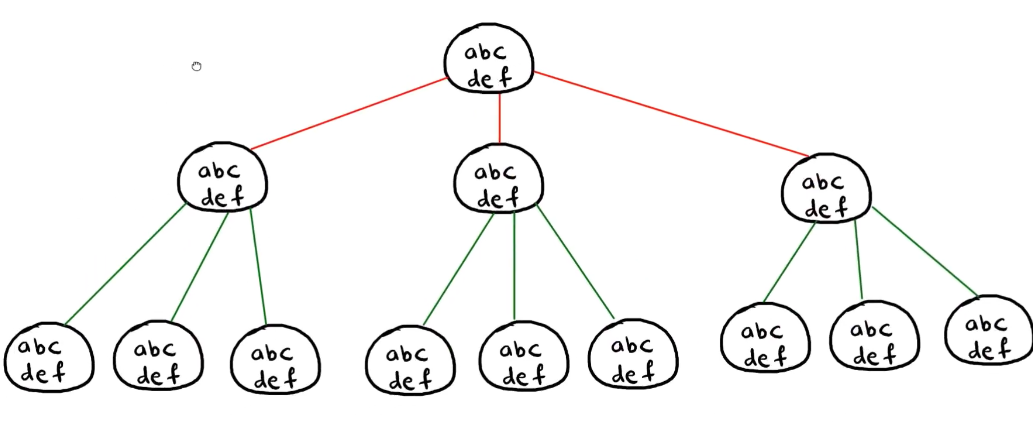
A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.



**Input:** digits = "23"

**Output:** ["ad","ae","af","bd","be","bf","cd","ce","cf"]

**Sol:**



First we will take a map of character and string.

We will store its key number as key in map and string present on that key as value in the map.

Here digit is given as “23”

So we will take first 2 from it and find out map where key is 2. We get the key value pair ‘2’, “abc”

So current String is “abc”. We will go through for loop till its length.

Then we will add its ith index char to String builder so stringbuilder will contain “a” and increase the i to i+1

So now I becomes 0+1 = 1. And 1st index in digit is 3 and its corresponding value in map is “def”

Now again it will go through loop and its ith char will be added into string builder. So now String builder will contain “ad”.

Its length is 2 which is equal to digits length.

It means we got one answer. We will add this in result and return.

Once we return we delete the last char from the string builder and we will back track.

So stringbuilder contains now “a” . it will e from “def “ this time. Again its length = 2 and we will add this in result.

Similarly it will execute for other chars.

**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.

**424. Longest Repeating Character Replacement**

You are given a string s and an integer k. You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most k times.

Return *the length of the longest substring containing the same letter you can get after performing the above operations*.

**Input:** s = "AABABBA", k = 1

**Output:** 4

**Explanation:** Replace the one 'A' in the middle with 'B' and form "AABBBBA".

The substring "BBBB" has the longest repeating letters, which is 4.

Sol:

Use variable size sliding window technique.

**1048. Longest String Chain**

You are given an array of words where each word consists of lowercase English letters.

wordA is a **predecessor** of wordB if and only if we can insert **exactly one** letter anywhere in wordA **without changing the order of the other characters** to make it equal to wordB.

* For example, "abc" is a **predecessor** of "abac", while "cba" is not a **predecessor** of "bcad".

A **word chain**is a sequence of words [word1, word2, ..., wordk] with k >= 1, where word1 is a **predecessor** of word2, word2 is a **predecessor** of word3, and so on. A single word is trivially a **word chain** with k == 1.

Return *the****length****of the****longest possible word chain****with words chosen from the given list of*words.

**Input:** words = ["a","b","ba","bca","bda","bdca"]

**Output:** 4

**Explanation**: One of the longest word chains is ["a","ba","bda","bdca"].

**318. Maximum Product of Word Lengths**

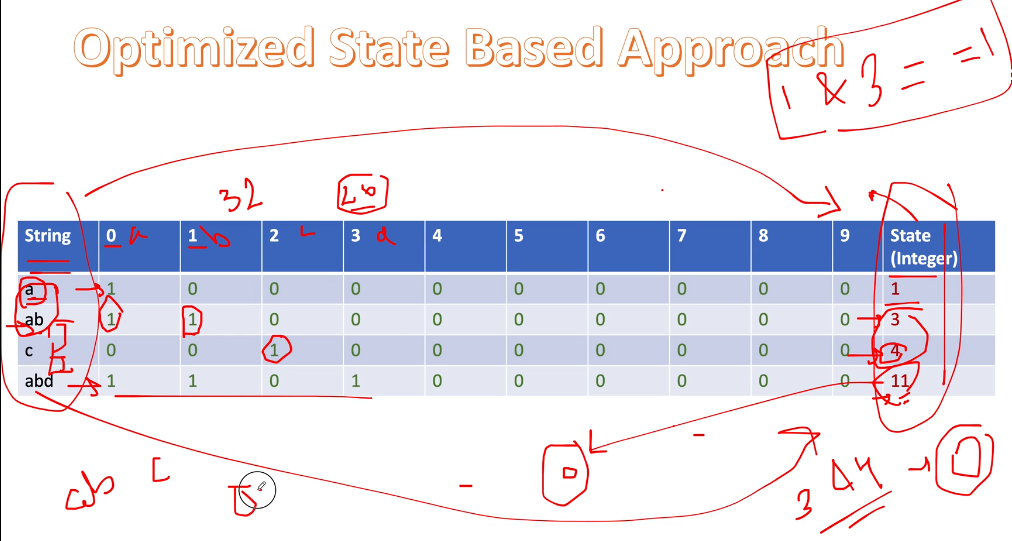
Given a string array words, return *the maximum value of* length(word[i]) \* length(word[j]) *where the two words do not share common letters*. If no such two words exist, return 0.

**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 cant 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.

**1239. Maximum Length of a Concatenated String with Unique Characters**

You are given an array of strings arr. A string s is formed by the **concatenation** of a **subsequence** of arr that has **unique characters**.

Return *the****maximum****possible length* of s.

**Input:** arr = ["cha","r","act","ers"]

**Output:** 6

**Explanation:** Possible longest valid concatenations are "chaers" ("cha" + "ers") and "acters" ("act" + "ers").

**921. Minimum Add to Make Parentheses Valid**

A parentheses string is valid if and only if:

* It is the empty string,
* 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.

You are given a parentheses string s. In one move, you can insert a parenthesis at any position of the string.

* For example, if s = "()))", you can insert an opening parenthesis to be "(**(**)))" or a closing parenthesis to be "())**)**)".

Return *the minimum number of moves required to make*s*valid*.

Sol:

If it is opening bracket then we will add into stack. Else we will check if top element of the stack is opening bracket. If it is then we will simply pop the element, else we will add into stack. In this way all the chars which are valid will be removed and only invalid char will be in the stack. Then return the size of the stack.

**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:

It is similar to question: add minimum to make the valid paranthesis. Here we take the stack of pair. In pair we store the invalid paranthesis and its index number. Once we get the invalid paranthesis we simpy delete that paranthesis from the string.

**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

**474. Ones and Zeroes**

You are given an array of binary strings strs and two integers m and n.

Return *the size of the largest subset of strs such that there are****at most***m0*'s and*n1*'s in the subset*.

A set x is a **subset** of a set y if all elements of x are also elements of y.

**Input:** strs = ["10","0001","111001","1","0"], m = 5, n = 3

**Output:** 4

**Explanation:** The largest subset with at most 5 0's and 3 1's is {"10", "0001", "1", "0"}, so the answer is 4.

Other valid but smaller subsets include {"0001", "1"} and {"10", "1", "0"}.

{"111001"} is an invalid subset because it contains 4 1's, greater than the maximum of 3.

**752. Open the Lock**

You have a lock in front of you with 4 circular wheels. Each wheel has 10 slots: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'. The wheels can rotate freely and wrap around: for example we can turn '9' to be '0', or '0' to be '9'. Each move consists of turning one wheel one slot.

The lock initially starts at '0000', a string representing the state of the 4 wheels.

You are given a list of deadends dead ends, meaning if the lock displays any of these codes, the wheels of the lock will stop turning and you will be unable to open it.

Given a target representing the value of the wheels that will unlock the lock, return the minimum total number of turns required to open the lock, or -1 if it is impossible.

**Input:** deadends = ["0201","0101","0102","1212","2002"], target = "0202"

**Output:** 6

**Explanation:**

A sequence of valid moves would be "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" -> "0202".

Note that a sequence like "0000" -> "0001" -> "0002" -> "0102" -> "0202" would be invalid,

because the wheels of the lock become stuck after the display becomes the dead end "0102".

**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]

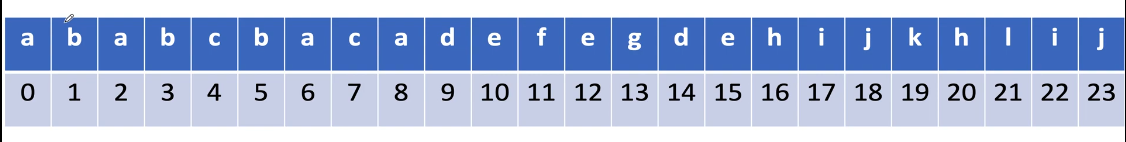
**Explanation:**

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.

**567. Permutation in String**

Given two strings s1 and s2, return true*if*s2*contains a permutation of*s1*, or*false*otherwise*.

In other words, return true if one of s1's permutations is the substring of s2.

**Input:** s1 = "ab", s2 = "eidbaooo"

**Output:** true

**Explanation:** s2 contains one permutation of s1 ("ba").

**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.

**151. Reverse Words in a String**

Given an input string s, reverse the order of the **words**.

A **word** is defined as a sequence of non-space characters. The **words** in s will be separated by at least one space.

Return *a string of the words in reverse order concatenated by a single space.*

**Note** that s may contain leading or trailing spaces or multiple spaces between two words. The returned string should only have a single space separating the words. Do not include any extra spaces.

**Input:** s = "the sky is blue"

**Output:** "blue is sky the"

**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**

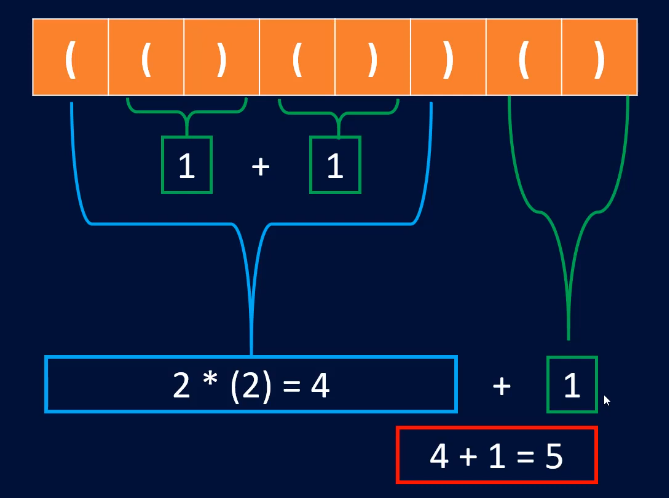
Given a balanced parentheses string s, return *the****score****of the string*.

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:

We will use stack.



We will iterate the string one by one.

If char is ‘(‘ then we will push 0 to the stack. If char is ‘)’ then we will do the calculation as follows.

For 1st char ‘(‘ we will push 0 to the stack…..**so in stack we have 0 as of now.**

For 2nd char ‘(‘ we will push 0 to the stack… **so in the stack we have 0, 0 as of now.**

For 3rd char ‘)’ we will pop the stack till stack.peek() != 0

Here stack.peek() = 0 so it will not go in this while loop.

Will check Math.max(2\*val, 1) it will return 1 so we will pop the stack and push val.

**So now stack will contain 0, 1**

For 4th char ‘(‘ we will push 0 to stack. **So in stack we have 0, 1, 0 as of now**

For 5th char ‘)’ we will pop the stack till stack.peek() != 0

Here stack.peek == 0 so it wont go in while loop

Will check Math.max(2\*val, 1) it will return 1 so we will pop the stack and push val.

**So now stack will contain 0, 1, 1**

For 6th char ‘)’ we will pop the stack till stack.peek() != 0 and add the val values. So val will be 1 + 1 = 2

**So stack will now contain 0**

Once we come out of while loop we check Math.max(2\*val, 1) which will return 4

We will pop the stack and push this val into stack

**So stack will contain now 4**

For 7th char ‘(‘ we will push 0 to stack. **So stack will contain now = 4, 0**

For 8th char ‘)’ we will pop the stak till stack.peek() != 0

It will not go in loop and outside loop we will check Math.max(2\*val, 1) which will return 1

Will pop the stack and push val into it.

**So stack will contain now 4, 1**

We have processed all the char of the string. In last we will sum up the stack values that will come 5 and will return it as result.

**1268. Search Suggestions System**

You are given an array of strings products and a string searchWord.

Design a system that suggests at most three product names from products after each character of searchWord is typed. Suggested products should have common prefix with searchWord. If there are more than three products with a common prefix return the three lexicographically minimums products.

Return *a list of lists of the suggested products after each character of*searchWord*is typed*.

**Input:** products = ["mobile","mouse","moneypot","monitor","mousepad"], searchWord = "mouse"

**Output:** [

["mobile","moneypot","monitor"],

["mobile","moneypot","monitor"],

["mouse","mousepad"],

["mouse","mousepad"],

["mouse","mousepad"]

]

**Explanation:** products sorted lexicographically = ["mobile","moneypot","monitor","mouse","mousepad"]

After typing m and mo all products match and we show user ["mobile","moneypot","monitor"]

After typing mou, mous and mouse the system suggests ["mouse","mousepad"]

**1202. Smallest String With Swaps**

You are given a string s, and an array of pairs of indices in the string pairs where pairs[i] = [a, b] indicates 2 indices(0-indexed) of the string.

You can swap the characters at any pair of indices in the given pairs **any number of times**.

Return the lexicographically smallest string that s can be changed to after using the swaps.

**Input:** s = "dcab", pairs = [[0,3],[1,2],[0,2]]

**Output:** "abcd"

**Explaination:**

Swap s[0] and s[3], s = "bcad"

Swap s[0] and s[2], s = "acbd"

Swap s[1] and s[2], s = "abcd"

**1081. Smallest Subsequence of Distinct Characters**

Given a string s, return *the lexicographically smallest subsequence of* s *that contains all the distinct characters of* s *exactly once*.

**Input:** s = "cbacdcbc"

**Output:** "acdb"

Sol:

This question is same as remove the duplicate character from string, which we had seen earlier.

**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.

**692. Top K Frequent Words**

Given an array of strings words and an integer k, return *the*k*most frequent strings*.

Return the answer **sorted** by **the frequency** from highest to lowest. Sort the words with the same frequency by their **lexicographical order**.

**Input:** words = ["i","love","leetcode","i","love","coding"], k = 2

**Output:** ["i","love"]

**Explanation:** "i" and "love" are the two most frequent words.

Note that "i" comes before "love" due to a lower alphabetical order.

**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 "".

**139. Word Break**

Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words.

**Note** that the same word in the dictionary may be reused multiple times in the segmentation.

**Input:** s = "leetcode", wordDict = ["leet","code"]

**Output:** true

**Explanation:** Return true because "leetcode" can be segmented as "leet code".