Strings

- Print anagrams together
- Longest even length substring
- Possible words from phone digits
- Alien Dictionary
- Longest Palindromic substring
- Design URL shortener
- Reverse each word in a given string
- Rabin Karp algo search pattern
- Rearrange characters
- Interleaved strings
- Generate IP addresses
- Circle of strings
- Length of longest substring
- Implement ATOI
- Search Pattern KMP algo
- Search Pattern Z algo
- Validate an IP address
- Largest numbers in K-swaps
- Excel sheet Part I
- Multiply two strings
- Count subsequences of type a^i, b^j, c^k
- String formation from substring
- Longest Prefix Suffix
- Number of distinct words with K max contiguous vowels
- Longest substring to form a palindrome
- IPL 2021 final
- Rank the permutations
- minimum times A has to be repeated s.t B is a substring of it.
- Smallest distinct window
- Longest Palindromic Substring in linear time
- Longest Valid Parantheses
- Restrictive Candy Crush
- Next higher pall no using same set of digits
- Sum of two large nums
- Column name from a given column number
- Add Binary Strings
- form a palindrome
- Count palindromic substrings of a string
- fact of large num.
- Count reversals
- Longest palindrome in a string
- Num following pattern
- Permutn of given string
- Substrings of length k with k-1 elements
- Repeated string Match

Sliding window

438. Find All Anagrams in a String

Medium 5733 228 Add to List Share

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.

Example 1:

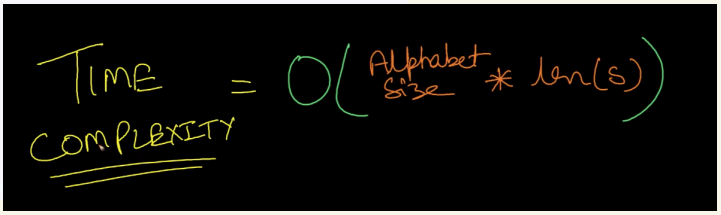
```
Input: s = "cbaebabacd", p = "abc"
Output: [0,6]
Explanation:
The substring with start index = 0 is "cba", which is an anagram of "abc".
The substring with start index = 6 is "bac", which is an anagram of "abc".
```

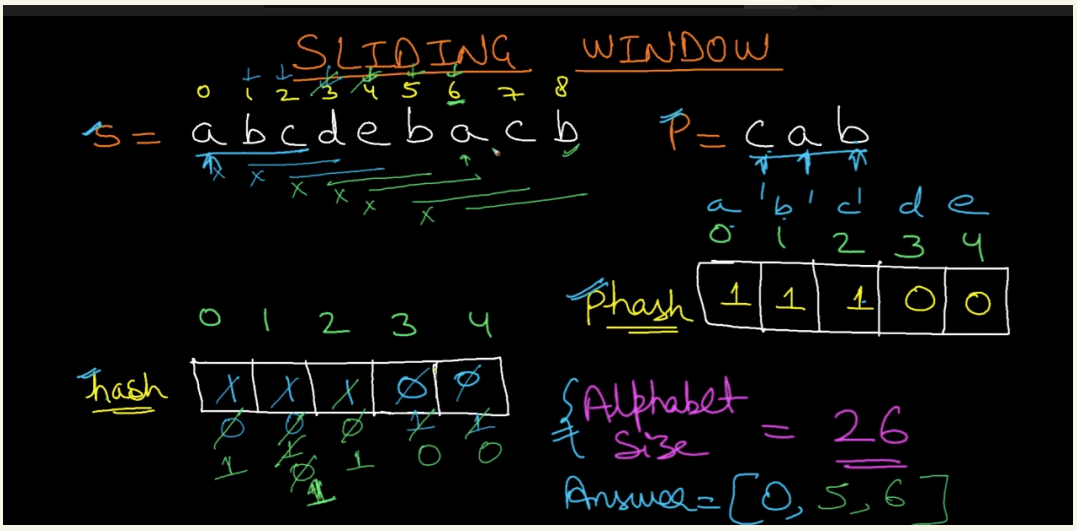
You need to record all the starting insides of anagrams of string b in string and you need to Return the vector.

We will us sliding window to solve this question

Example 2:

```
Input: s = "abab", p = "ab"
Output: [0,1,2]
Explanation:
The substring with start index = 0 is "ab", which is an anagram of "ab".
The substring with start index = 1 is "ba", which is an anagram of "ab".
The substring with start index = 2 is "ab", which is an anagram of "ab".
```






```cpp
class Solution {
public:
    vector<int> findAnagrams(string s, string p)
    {
        vector<int> pv(26,0), sv(26,0), res;
        if(s.size() < p.size()) return res;
// fill pv, vector of counters for pattern string and sv, vector of counters for the sliding window
        for(int i = 0; i < p.size(); ++i)
        {
            ++pv[p[i]-'a'];
            ++sv[s[i]-'a'];
        }
        if(pv==sv) res.push_back(0);
        //here window is moving from left to right across the string.
        //window size is p.size(), so s.size()-p.size() moves are made
        for(int i = p.size(); i < s.size(); ++i)
        {
             // window extends one step to the right. counter for s[i] is incremented
            ++sv[s[i]-'a'];
            // since we added one element to the right,one element to the left should be discarded.
            --sv[s[i-p.size()]-'a'];
            if(pv == sv)res.push_back(i-p.size()+1);
            /*
 if after move to the right the anagram can be composed,add new position of window's left point to the result. This comparison takes O(26), i.e O(1), since both vectors are of fixed size 26. Total complexity O(n)*O(1) = O(n)
            */
        }
        return res;
    }
};
```

242. Valid Anagram

Easy 3997 198 Add to List

Given two strings `s` and `t`, return `true` *if* `t` *is an anagram of* `s`, *and* `false` *otherwise.*

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.

Example 1:

```
Input: s = "anagram", t = "nagaram"
Output: true
```

```cpp
class Solution {
public:
    bool isAnagram(string s, string t)
    {
        bool ans=false;
        if(s.size()!=t.size()) return ans;
        vector<int> sv(26,0), tv(26,0);
        for(int i=0;i<s.size();i++)
        {
            ++sv[s[i]-'a'];
            ++tv[t[i]-'a'];
        }
        return sv==tv;
    }
};
```

This question also uses the concept Of sliding window.

49. Group Anagrams

Medium 8017 277 Add to List Share

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.

Example 1:

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

Example 2:

```
Input: strs = [""]
Output: [[""]]
```

Example 3:

```
Input: strs = ["a"]
Output: [["a"]]
```

```cpp
#define vs vector<string>
#define vvs vector<vs>
#define um unordered_map
class Solution{
    public:
    vvs groupAnagrams(vector<string>& S){
        if(S.size() == 1) return {{S[0]}};
        vvs result;
        um<string, vs> m;
        for(int  i = 0; i < S.size(); i++){
            string s = S[i];
            sort(S[i].begin(), S[i].end());
            m[S[i]].push_back(s);
        }
        for(auto i= m.begin(); i != m.end(); i++) result.push_back(i->second);
        return result;
    }
};
```

Use an `unordered_map` to group the strings by their sorted counterparts. Use the sorted string as the key and all anagram strings as the value.

```cpp
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        unordered_map<string, vector<string>> mp;
        for (string s : strs) {
            string t = s;
            sort(t.begin(), t.end());
            mp[t].push_back(s);
        }
        vector<vector<string>> anagrams;
        for (auto p : mp) {
            anagrams.push_back(p.second);
        }
        return anagrams;
    }
};
```

Moreover, since the string only contains lower-case alphabets, we can sort them using counting sort to improve the time complexity.

```cpp
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        unordered_map<string, vector<string>> mp;
        for (string s : strs) {
            mp[strSort(s)].push_back(s);
        }
        vector<vector<string>> anagrams;
        for (auto p : mp) {
            anagrams.push_back(p.second);
        }
        return anagrams;
    }
private:
    string strSort(string s) {
        int counter[26] = {0};
        for (char c : s) {
            counter[c - 'a']++;
        }
        string t;
        for (int c = 0; c < 26; c++) {
            t += string(counter[c], c + 'a');
        }
        return t;
    }
};
```

Great answer! I have made several improvements of your code. The new answer takes 52ms

1. use auto& rather then auto to avoid unnecessary copy
2. use std::move() to steal vector from map
3. use vector.reserve() to avoid reallocate

```
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        int n = strs.size();
        unordered_map<string, vector<string>> map;
        vector<vector<string>> ret;
        for (const auto& s : strs) {
            string t = s;
            sort(t.begin(), t.end());
            map[t].push_back(s);
        }
        ret.reserve(map.size());
        for (auto& p : map) {
            ret.push_back(std::move(p.second));
        }
        return ret;
    }
};
```

Print Anagrams Together

Medium Accuracy: 56.1% Submissions: 25856 Points: 4

Given an array of strings, return all groups of strings that are anagrams. The groups must be created in order of their appearance in the original array. Look at the sample case for clarification.

Example 1:

```
Input:
N = 5
words[] = {act,god,cat,dog,tac}
Output:
act cat tac
god dog
Explanation:
There are 2 groups of
anagrams "god", "dog" make group 1.
"act", "cat", "tac" make group 2.
```

Example 2:

```
Input:
N = 3
words[] = {no,on,is}
Output:
no on
is
Explanation:
There are 2 groups of
anagrams "no", "on" make group 1.
"is" makes group 2.
```

Your Task:
The task is to complete the function **Anagrams()** that takes a list of strings as input and returns a list of groups such that each group consists of all the strings that are anagrams.

Expected Time Complexity: O(N*|S|*log|S|), where |S| is the length of the strings.
Expected Auxiliary Space: O(N*|S|), where |S| is the length of the strings.

Stop loadi

```
class Solution{
  public:
    vector<vector<string> > Anagrams(vector<string>& arr) {
        //code here
        unordered_map<string,vector<string>>mp;
        vector<vector<string>>ans;
        for(auto it:arr){
            string temp=it;
            sort(it.begin(),it.end());
            mp[it].push_back(temp);
        }
        for(auto it:mp){
            ans.push_back(it.second);
        }
        return ans;

    }
};
```

Longest Even Length Substring

Medium Accuracy: 49.99% Submissions: 15790 Points: 4

For given string 'str' of digits, find length of the **longest** substring of 'str', such that the length of the substring is 2k digits and sum of left k digits is equal to the sum of right k digits.

Input:

The first line of input contains an integer T denoting the number of test cases. The description of T test cases follows.
Each test case contains a string string of length N.

Output:

Print length of the longest substring of length 2k such that sum of left k elements is equal to right k elements and if there is no such substring print 0.

Constraints:

$1 \le T \le 100$
$1 \le N \le 100$

Example:

Input:
2
000000
1234123

Output:
6
4

```cpp
#include <iostream>
using namespace std;

int main()
{
    int t,i;
    cin>>t;
    string s;
    while(t--)
    {
        cin>>s;
        int n=s.size(),ans=0,lsum=0,rsum=0,l,r;
        for(i=0;i<n;i++)
        {
            lsum=0,rsum=0;
            l=i;
            r=i+1;
            while(l>=0 and r<=n-1)
            {
                lsum+=s[l]-'0';
                rsum+=s[r]-'0';

                if(lsum==rsum) ans =max(ans,r-l+1);
                l--;
                r++;
            }
        }
        cout<<ans<<endl;
    }
}
```

[A $O(n^2)$ time and $O(1)$ extra space solution]

The idea is to consider all possible mid points (of even length substrings) and keep expanding on both sides to get and update optimal length as the sum of two sides become equal.

17. Letter Combinations of a Phone Number

Medium 8725 622 Add to List Share

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

Example 1:

```
Input: digits = "23"
Output:
["ad","ae","af","bd","be","bf","cd","ce","cf"]
```

Example 2:

```
Input: digits = ""
Output: []
```

```
class Solution {
private:
    unordered_map<char,vector<char>>table;
    void initializeTable(){
        table['1']={};
        table['2']={'a','b','c'};
        table['3']={'d','e','f'};
        table['4']={'g','h','i'};
        table['5']={'j','k','l'};
        table['6']={'m','n','o'};
        table['7']={'p','q','r','s'};
        table['8']={'t','u','v'};
        table['9']={'w','x','y','z'};
    }
    void backtrack(vector<string>&ans,string &curr,string digits)
    {
        if(digits.empty()){
            ans.push_back(curr);
        }
        else
        {
            vector<char>cnds=table[digits[0]];
            for(int i=0;i<cnds.size();i++)
            {
                curr.push_back(cnds[i]);
                backtrack(ans,curr,digits.substr(1));
                curr.pop_back();
            }
        }
    }
public:
    vector<string> letterCombinations(string digits)
    {
        initializeTable();
        vector<string>ans;
        if(!digits.empty()){
            string curr="";
            backtrack(ans,curr,digits);
        }
        return ans;
    }
};
```

GFG HARD Microsoft

Alien Dictionary

Hard Accuracy: 48.62% Submissions: 32142 Points: 8

Given a sorted dictionary of an alien language having N words and k starting alphabets of standard dictionary. Find the order of characters in the alien language.
Note: Many orders may be possible for a particular test case, thus you may return any valid order and output will be 1 if the order of string returned by the function is correct else 0 denoting incorrect string returned.

Example 1:

```
Input:
N = 5, K = 4
dict = {"baa","abcd","abca","cab","cad"}
Output:
1
Explanation:
Here order of characters is
'b', 'd', 'a', 'c' Note that words are sorted
and in the given language "baa" comes before
"abcd", therefore 'b' is before 'a' in output.
Similarly we can find other orders.
```

Expected Time Complexity: O(N * |S| + K) , where |S| denotes maximum length.
Expected Space Compelxity: O(K)

[LeetCode] 269. Alien Dictionary

There is a new alien language which uses the latin alphabet. However, the order among letters are unknown to you. You receive a list of **non-empty** words from the dictionary, where **words are sorted lexicographically by the rules of this new language**. Derive the order of letters in this language.

Example 1:
Given the following words in dictionary,

```
[
  "wrt",
  "wrf",
  "er",
  "ett",
  "rftt"
]
```

The correct order is: `"wertf"`.

Example 2:
Given the following words in dictionary,

```
[
  "z",
  "x"
]
```

The correct order is: `"zx"`.

Example 3:

Topological sort application

```cpp
class Solution{
    public:
    void dfs(int src, vector<int>& visited, vector<int>& topo, vector<int> adj[]){
        visited[src] = 1;
        for(auto i : adj[src]){
            if(!visited[i]){
                dfs(i,visited,topo,adj);
            }
        }
        topo.push_back(src);
    }
    string findOrder(string dict[], int N, int K){
        vector<int>adj[K];
        for(int i=1; i<N; i++){
            string s1 = dict[i-1];
            string s2 = dict[i];
            for(int j=0; j<min(s1.size(),s2.size()); j++){
                if(s1[j] != s2[j]){
                    int u = s1[j] - 'a';
                    int v = s2[j] - 'a';
                    adj[u].push_back(v);
                    break;
                }
            }
        }
        vector<int>visited(K,0);
        vector<int>topo;
        for(int i=0; i<K; i++)
            if(!visited[i]) dfs(i,visited,topo,adj);
        reverse(topo.begin(),topo.end());
        string s = "";
        for(int i=0; i<topo.size(); i++){
            s+=topo[i] + 'a';
        }
        return s;
    }
};
```

The idea is to create a graph of characters and then find topological sorting of the created graph. Following are the detailed steps.
1) Create a graph *g* with number of vertices equal to the size of alphabet in the given alien language. For example, if the alphabet size is 5, then there can be 5 characters in words. Initially there are no edges in graph.
2) Do following for every pair of adjacent words in given sorted array.
.....a) Let the current pair of words be *word1* and *word2*. One by one compare characters of both words and find the first mismatching characters.
.....b) Create an edge in *g* from mismatching character of *word1* to that of *word2*.
3) Print topological sorting of the above created graph.

The implementation of the above is in C++.

Time Complexity: The first step to create a graph takes O(n + alpha) time where n is number of given words and alpha is number of characters in given alphabet. The second step is also topological sorting. Note that there would be alpha vertices and at-most (n-1) edges in the graph. The time complexity of topological sorting is O(V+E) which is O(n + alpha) here. So overall time complexity is O(n + alpha) + O(n + alpha) which is O(n + alpha).

Exercise: The above code doesn't work when the input is not valid. For example {"aba", "bba", "aaa"} is not valid, because from first two words, we can deduce 'a' should appear before 'b', but from last two words, we can deduce 'b' should appear before 'a' which is not possible. Extend the above program to handle invalid inputs and generate the output as "Not valid".

5. Longest Palindromic Substring

Medium 15578 911 Add to List Share

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"
```

```cpp
/*
Here we are creating a function f which will simply traverse the string and store the max length of
longest palindromic substring. Note substring is not same as subsequence.
*/
class Solution {
public:
    void f(int &l,int &r, string &str,int &n, int &max_len, int &start)
    {
        while(l>=0 && r<str.length() && str[l]==str[r])
            {
                int len=r-l+1;
                if(len > max_len){
                    max_len = len;
                    start = l; //store the start index of the max length LPS
                }
                l--;r++; //expanding from centre towards left and right, if palindrome exists
            }
    }
    string longestPalindrome(string str)
    {
        string ans; //store the answer here
        int max_len = 0; //maxlength is 0 initially
        int start, end; //store the start and end pointers used in traversing the string.
        int n = str.size(); //length of the string.
        for(int i=0;i<n;i++) //for loop for every index i.
        {
            int l = i, r = i; // same centre odd LPS
            f(l,r,str,n,max_len,start);
            l = i; r = i+1; //start and end are same character but seperated EVEN LPS
            f(l,r,str,n,max_len,start);
        }
        return str.substr(start, max_len); return max length substr.
    }
};
```

Design a tiny URL or URL shortener

Medium Accuracy: 65.11% Submissions: 3466 Points: 4

Design a system that takes big URLs like "**http://www.geeksforgeeks.org/count-sum-of-digits-in-numbers-from-1-to-n/**" and converts them into a short **URL**. It is given that URLs are stored in database and every URL has an associated integer **id**. So your program should take an integer id and generate a URL.

A URL character can be one of the following

1. A lower case alphabet ['a' to 'z'], total 26 characters
2. An upper case alphabet ['A' to 'Z'], total 26 characters
3. A digit ['0' to '9'], total 10 characters

There are total 26 + 26 + 10 = 62 possible characters.

So the task is to convert an integer (database id) to a base 62 number where digits of 62 base are "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMN OPQRSTUVWXYZ0123456789"

Example 1:

```
Input:
N = 12345
Output:
dnh
12345
Explanation: "dnh" is the url for id 12345
```

Example 2:

```
Input:
N = 30540
Output:
h6K
30540
Explanation: "h6K" is the url for id 30540
```

Your Task:

```cpp
class Solution
{
    public:
    string idToShortURL(long long int n)
    {
        string s="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
        string nums="";
        while(n)
        {
            nums.push_back(s[n%62]);
            n/=62;
        }
        reverse(nums.begin(),nums.end());
        return nums;
    }
};
```

Expected Time Complexity: O(N)
Expected Auxiliary Space: O(1)

```cpp
class Solution
{
    public:
    string idToShortURL(long long int n)
    {
        string s="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
        string nums="";
        if(!n)return nums;
        if(n<62)
        {
            nums+=s[n];
            return nums;
        }
        while(n>=62)
        {
            nums=s[n%62]+nums;
            if((n/62)<62) nums=s[n/62]+nums;
            n/=62;
        }
    return nums;
    }
};
```

186 Reverse Words in a String II – Medium

Problem:

Given an input string, reverse the string word by word. A word is defined as a sequence of non-space characters.

The input string does not contain leading or trailing spaces and the words are always separated by a single space.

For example, Given s = “the sky is blue”, return “blue is sky the”.

Could you do it in-place without allocating extra space?

Thoughts:

The original problem Reverse Words in a String is quite straightforward, however the solution there might not be used in this problem.

The trick part is that this problem requires do it in-place. In place doesn’t mean that you cannot use O(1) space. It means you cannot have another array.

The solution is a little tricky, the idea is to reverse the whole string first, so “the sky is blue” becomes “eulb si yks eht”. This will make all the space in this string to be in the expected position. Next we need to reverse each word, so that it becomes “blue is sky the”.

```java
public class Solution {
    public void reverseWords(char[] s) {
        if (s.length == 0) {
            return;
        }
        reverse(s, 0, s.length - 1);
        int left = 0;
        for (int i = 0; i <= s.length; i ++) {
            if (i == s.length || s[i] == ' ') {
                reverse(s, left, i - 1);
                left = i + 1;
            }
        }
    }
    private void reverse(char[] s, int l, int r) {
        while (l < r) {
            char tmp = s[l];
            s[l] = s[r];
            s[r] = tmp;
            l ++;
            r --;
        }
    }
}
```

i → “The sky is blue”

→ o → "blue is sky the"

Can you do this in inplace?

O(1) space?

As you don't have extra space,

Reverse the whole string first

O(n) time

O(1) space

"eulb si yks eht" → blue is sky The ✓

Now iterate on the string & reverse(s.begin()+l, s.begin()+r)

Reverse each word in a given string

Medium Accuracy: 53.06% Submissions: 12092 Points: 4

Given a String. Reverse each word in it where the words are separated by dots.

Example 1:

```
Input:
S = "i.like.this.program.very.much"
Output:
i.ekil.siht.margorp.yrev.hcum
Explanation:
The words are reversed as
follows:"i" -> "i","like"->"ekil",
"this"->"siht","program" -> "margorp",
"very" -> "yrev","much" -> "hcum".
```

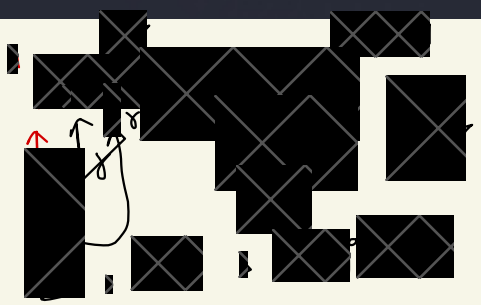
Example 2:

```
Input:
S = "pqr.mno"
Output:
rqp.onm
Explanation:
The words are reversed as
follows:"pqr" -> "rqp" ,
"mno" -> "onm"
```

Your Task:
You don't need to read input or print anything. Your task is to complete the function**reverseWords()**which takes the string S as input and returns the result string by reversing all the words separated by dots.

Expected Time Complexity:O(|S|).
Expected Auxiliary Space:O(|S|).

```
class Solution
{
  public:
    string reverseWords (string s)
    {
        int l=0,r=0,n=s.size();

        while(l<n)
        {
            while(r<n and s[r]!='.'){r++;}
            reverse(s.begin()+l,s.begin()+r);
            l=r+1;
            r=l;
        }
        return s;
    }
};
```



Search Pattern (Rabin-Karp Algorithm)

Medium Accuracy: 45.53% Submissions: 4415 Points: 4

Given two strings, one is a text string and other is a pattern string. The task is to print the indexes of all the occurences of pattern string in the text string. For printing, Starting Index of a string should be taken as 1.

Example 1:

```
Input:
S = "batmanandrobinarebat", pat = "bat"
Output: 1 18
Explanation: The string "bat" occurs twice
in S, one starts are index 1 and the other
at index 18.
```

â€‹**Example 2:**

```
Input:
S = "abesdu", pat = "edu"
Output: -1
Explanation: There's not substring "edu"
present in S.
```

Your Task:
You don't need to read input or print anything. Your task is to complete the function **search()** which takes the string S and the string pat as inputs and returns an array denoting the start indices (1-based) of substring pat in the string S.

Expected Time Complexity: O(|S|*|pat|).
Expected Auxiliary Space: O(1).

```
class Solution
{
    public:
        vector <int> search(string pattern, string text)
        {
            int k=0;
            vector<int> v;
            if(text.find(pattern)==string::npos) v.push_back(-1);
            for(int i=0;i<text.size()-pattern.size()+1;i++)
                if(text.substr(i,pattern.size())==pattern) v.push_back(i+1);
            if(v.empty()) v.push_back(-1);
            return v;
        }

};
```

Like the Naive Algorithm, Rabin-Karp algorithm also slides the pattern one by one. But unlike the Naive algorithm, Rabin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for following strings.

1) Pattern itself.
2) All the substrings of the text of length m.

Since we need to efficiently calculate hash values for all the substrings of size m of text, we must have a hash function which has the following property.

Hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say *hash(txt[s+1 .. s+m])* must be efficiently computable from *hash(txt[s .. s+m-1])* and *txt[s+m]* i.e., *hash(txt[s+1 .. s+m])= rehash(txt[s+m], hash(txt[s .. s+m-1]))* and rehash must be O(1) operation.

The hash function suggested by Rabin and Karp calculates an integer value. The integer value for a string is the numeric value of a string.

This is simple mathematics, we compute decimal value of current window from previous window.
For example pattern length is 3 and string is "23456"
You compute the value of first window (which is "234") as 234.
*How how will you compute value of next window "345"? You will do (234 – 2*100)* 10 + 5 and get 345.*

Time Complexity:
The average and best-case running time of the Rabin-Karp algorithm is O(n+m), but its worst-case time is O(nm). Worst case of Rabin-Karp algorithm occurs when all characters of pattern and text are same as the hash values of all the substrings of txt[] match with hash value of pat[]. For example pat[] = "AAA" and txt[] = "AAAAAAA".

String find is used to find the first occurrence of sub-string in the specified string being called upon. It returns the index of the first occurrence of the substring in the string from given starting position. The default value of starting position is 0.

Function Template:

- size_t find (const string& str, size_t pos = 0);
- size_t find (const char* s, size_t pos = 0);

Function parameters:

- str : The sub-string to be searched.
- s : The sub-string to be searched, given as C style string.
- pos : The initial position from where the string search is to begin.

Function Return:

- The function returns the index of the first occurrence of sub-string, if the sub-string is not found it returns string::npos(string::pos is static member with value as the highest possible for the size_t data structure).

```cpp
class Solution
{
    public:
        vector <int> search(string p, string s)
        {
            vector <int> v; //create an empty vector
            int i=0,j=0;//pointers i and j
            string r=""; //empty result string.
            while(j<s.size()) //travaerse the string s
            {
                r+=s[j]; //keep on addiing string elements in r.
                if(j-i+1==p.size()) //if size of pattern equal to difference between i and j
                {
                    if(r==p)v.push_back(i+1); //check if r==p then push it
                    r=r.substr(1); //else place first char of string in r as r and
                    i++; //increment i.
                }
                j++; //increment j
            }
            if(v.empty())return v={-1}; //if you didnt find any pattern retrn -1
            return v;
        }
};
```

Rearrange characters

Medium Accuracy: 46.45% Submissions: 8723 Points: 4

Expected Time Complexity : O(NlogN), N = length of String
Expected Auxiliary Space : O(number of english alphabets)

Given a string S with repeated characters. The task is to rearrange characters in a string such that no two adjacent characters are the same.
Note: The string has only lowercase English alphabets and it can have multiple solutions. Return any one of them.

Input : str = "**geeksforgeeks**" Output: 1
Explanation:
All the repeated characters of the given string can be rearranged so that no adjacent characters in the string is equal. Any correct rearrangement will show a output of 1.

Input : str = "bbbbb" Output: 0
Explanation :

Repeated characters in the string cannot be rearranged such that there should not be any adjacent repeated character.

Prerequisite : priority_queue.
The idea is to put the highest frequency character first (a greedy approach). We use a priority queue (Or Binary Max Heap) and put all characters and ordered by their frequencies (highest frequency character at root). We one by one take the highest frequency character from the heap and add it to result. After we add, we decrease the frequency of the character and we temporarily move this character out of priority queue so that it is not picked next time.
We have to follow the step to solve this problem, they are:
1. Build a Priority_queue or max_heap, **pq** that stores characters and their frequencies.
..... Priority_queue or max_heap is built on the bases of the frequency of character.
2. Create a temporary Key that will be used as the previously visited element (the previous element in the resultant string. Initialize it { char = '#' , freq = '-1' }
3. While **pq** is not empty.
..... Pop an element and add it to the result.
..... Decrease frequency of the popped element by '1'
..... Push the previous element back into the priority_queue if it's frequency > '0'
..... Make the current element as the previous element for the next iteration.
4. If the length of the resultant string and original string is not equal, print "not possible". Else print result.
Below is the implementation of above idea

Another approach :

Another approach is to fill all the even positions of the result string first, with the highest frequency character. If there are still some even positions remaining, fill them first. Once even positions are done, then fill the odd positions. This way, we can ensure that no two adjacent characters are the same.

```cpp
char getMaxCountChar(const vector<int>& count)
{
    int max = 0;
    char ch;
    for (int i = 0; i < 26; i++) {
        if (count[i] > max) {
            max = count[i];
            ch = 'a' + i;
        }
    }

    return ch;
}
```

```
string rearrangeString(string S)
{

    int n = S.size();
    if (!n)
        return "";

    vector<int> count(26, 0);
    for (auto ch : S)
        count[ch - 'a']++;

    char ch_max = getMaxCountChar(count);
    int maxCount = count[ch_max - 'a'];

    // check if the result is possible or not
    if (maxCount > (n + 1) / 2)
        return "";

    string res(n, ' ');

    int ind = 0;
    // filling the most frequently occurring char in the even
    // indices
    while (maxCount) {
        res[ind] = ch_max;
        ind = ind + 2;
        maxCount--;
    }
    count[ch_max - 'a'] = 0;
```

```
    // now filling the other Chars, first filling the even
    // positions and then the odd positions
    for (int i = 0; i < 26; i++) {
        while (count[i] > 0) {
            ind = (ind >= n) ? 1 : ind;
            res[ind] = 'a' + i;
            ind += 2;
            count[i]--;
        }
    }
    return res;
}

// Driver program to test above function
int main()
{
    string str = "bbbaa";
    string res = rearrangeString(str);
    if (res == "")
        cout << "Not valid string" << endl;
    else
        cout << res << endl;
    return 0;
}
```

Output

```
babab
```

Time complexity : O(n)

Space complexity : O(n+26) where 26 is the size of the vocabulary.

First approach using PQ

```
void rearrangeString(string str)
{
    int n = str.length();

    // Store frequencies of all characters in string
    int count[MAX_CHAR] = { 0 };
    for (int i = 0; i < n; i++)
        count[str[i] - 'a']++;

    // Insert all characters with their frequencies
    // into a priority_queue
    priority_queue<Key> pq;
    for (char c = 'a'; c <= 'z'; c++) {
        int val = c - 'a';
        if (count[val]) {
            pq.push(Key{ count[val], c });
        }
    }

    // 'str' that will store resultant value
    str = "";

    // work as the previous visited element
    // initial previous element be. ( '#' and
    // it's frequency '-1' )
    Key prev{ -1, '#' };

    // traverse queue
    while (!pq.empty()) {
        // pop top element from queue and add it
        // to string.
        Key k = pq.top();
        pq.pop();
        str = str + k.ch;
```

```
        // IF frequency of previous character is less
        // than zero that means it is useless, we
        // need not to push it
        if (prev.freq > 0)
            pq.push(prev);

        // make current character as the previous 'char'
        // decrease frequency by 'one'
        (k.freq)--;
        prev = k;
    }

    // If length of the resultant string and original
    // string is not same then string is not valid
    if (n != str.length())
        cout << " Not valid String " << endl;

    else // valid string
        cout << str << endl;
}

// Driver program to test above function
int main()
{
    string str = "bbbaa";
    rearrangeString(str);
    return 0;
}
```

358-rearrange-string-k-distance-apart

https://leetcode.com/problems/rearrange-string-k-distance-apart/description/

Given a non-empty string **s** and an integer **k**, rearrange the string such that the same characters are at least distance k from each other.

All input strings are given in lowercase letters. If it is not possible to rearrange the string, return an empty string "".

Example:

```
s = "aabbcc", k = 3

Result: "abcabc"

The same letters are at least distance 3 from each other.
```

```
s = "aaabc", k = 3

Answer: ""

It is not possible to rearrange the string.
```

```
s = "aaadbbcc", k = 2

Answer: "abacabcd"

Another possible answer is: "abcabcda"

The same letters are at least distance 2 from each other.
```

Thought Process

1. Map and MaxHeap
    i. Use map to record the count for each character, and use heap to poll the character with most count
    ii. Every time we poll a character out and append this character to our string builder and save the character to a queue for latter use
    iii. Time complexity O(n log(26)) or O(n)
    iv. Space complexity O(1)
2. Map and Index Map
    i. We store the count and valid index for each letter
    ii. Every time we insert a character to our result, we need to increase this character next valid position to i + k
    iii. We use a separate function to search the next character from 26 letters, where it has maximum count and it falls outside the valid index
    iv. Time complexity O(n)
    v. Space complexity O(1)

```java
class Solution {
    public String rearrangeString(String s, int k) {
        Map<Character, Integer> map = new HashMap<>();
        for (char c : s.toCharArray()) {
            map.put(c, map.getOrDefault(c, 0) + 1);
        }
        PriorityQueue<Map.Entry<Character, Integer>> maxHeap = new PriorityQueue<>((a, b)
        maxHeap.addAll(map.entrySet());
        StringBuilder sb = new StringBuilder();
        Queue<Map.Entry<Character, Integer>> wait = new LinkedList<>();
        while (!maxHeap.isEmpty()) {
            Map.Entry<Character, Integer> cur = maxHeap.poll();
            sb.append(cur.getKey());
            cur.setValue(cur.getValue() - 1);
            wait.offer(cur);
            if (wait.size() < k) continue;
            Map.Entry<Character, Integer> next = wait.poll();
            if (next.getValue() > 0) maxHeap.offer(next);
        }
        return sb.length() == s.length() ? sb.toString() : "";
    }
}
```

blocks[4]={cccc->aacc->aabc}

cnts—> 214

For all cnt in sorted_cnt

k=1

l=0

cbcacbac —>a2b1c4

2a1b4c -> Sort -> 4c2a1b

max_cnt=4

```cpp
// Time:  O(n)
// Space: O(n)

class Solution {
public:
    string rearrangeString(string str, int k) {
        int cnts [26] = {0};
        for (int i = 0;  i < str.length(); ++i) {
            ++cnts[str[i] - 'a'];
        }

        vector<pair<int, char>> sorted_cnts;
        for (int i = 0; i < 26; ++i) {
            sorted_cnts.emplace_back(cnts[i], i + 'a');
        }
        sort(sorted_cnts.begin(), sorted_cnts.end(), greater<pair<int, int>>());

        const auto max_cnt = sorted_cnts[0].first;
        string blocks[max_cnt];
        int i = 0;
        for (const auto& cnt : sorted_cnts) {
            for (int j = 0; j < cnt.first; ++j) {
                blocks[i].push_back(cnt.second);
                i = (i + 1) % max(cnt.first, max_cnt - 1);
            }
        }

        string result;
        for (int i = 0; i < max_cnt - 1; ++i) {
            if (blocks[i].length() < k) {
                return "";
            } else {
                result += blocks[i];
            }
        }
        result += blocks[max_cnt - 1];
        return result;
    }
};

// Time:  O(nlogc), c is the count of unique characters.
```

In the first solution we are using a function to rearrange our strings and we are returning a string as output. The input parameters are string s and the k which tells, the minimum distance of a character with it's next occurrence if it occurs more than once.

We are creating a count array of all the 26 characters and preinitialising the array with 0 and then with the for loop we are updating the count of every character in the count array.
Now we have a vector of pair of int key and char value and we are naming it as sorted_cnts. It will simply add all the characters in the vector from the end in sorted order. We can also use push_back.

We will now sort the count vector in decreasing order As we need the character who has highest count. So we will store the max_count in a variable which is first of first element of array. Then we will create a string blocks of the max count length. And now for every character in this sorted array, we will place the character in the blocks array till its count is over. And then we will update the index I as 1,2,3,0 as 4%4=0 then we check for 2a cccc->aacc-> I=2 it didn't become 0. At index 2 we will push b and we get -> aabc, k=1
We return null if we don't have enough characters
we add the characters into the result string
And for the last character we just add it and return result.

```cpp
// Space: O(c)
class Solution2 {
public:
    string rearrangeString(string str, int k) {
        if (k == 0) {
            return str;
        }

        unordered_map<char, int> cnts;
        for (const auto& c : str) {
            ++cnts[c];
        }

        priority_queue<pair<int, char>> heap;
        for (const auto& kvp : cnts) {
            heap.emplace(kvp.second, kvp.first);
        }

        string result;
        while (!heap.empty()) {
            vector<pair<int, char>> used_cnt_chars;
            int cnt = min(k, static_cast<int>(str.length() - result.length()));
            for (int i = 0; i < cnt; ++i) {
                if (heap.empty()) {
                    return "";
                }
                auto cnt_char = heap.top();
                heap.pop();
                result.push_back(cnt_char.second);
                if (--cnt_char.first > 0) {
                    used_cnt_chars.emplace_back(move(cnt_char));
                }
            }
            for (auto& cnt_char: used_cnt_chars) {
                heap.emplace(move(cnt_char));
            }
        }
        return result;
    }
};
```

if k=0 then return str, it is the base case

Create an unordered map of freq of chars

create a maxheap and store every count and its char In the heap as pair.

declare a string result and then till the heap is not empty cnt= string length - result length = string length initially as result=null string
Store the char at top of heap in cnt_char and pop that top from heap.
push it in result and if the count after decrementingthe top is greater than 0 place the char with the greater count in other max heap and lets call it as used max heap.
Move the character from heap 1 to heap2.

The approach to solving this problem is to count frequencies of all characters and consider the most frequent character first and place all occurrences of it as close as possible. After the most frequent character is placed, repeat the same process for the remaining characters.

1. Let the given string be str and size of string be n
2. Traverse str, store all characters and their frequencies in a Max Heap MH (implemented using priority queue). The value of frequency decides the order in MH, i.e., the most frequent character is at the root of MH.
3. Make all characters of str as '\0'.
4. Do the following while MH is not empty.
   - Extract the Most frequent character. Let the extracted character be x and its frequency be f.
   - Find the first available position in str, i.e., find the first '\0' in str.
   - Let the first position be p. Fill x at p, p+d,.. p+(f-1)d

Below is the implementation of the above algorithm.

```cpp
// C++ program to rearrange a string so that all same
// characters become at least d distance away using STL
#include <bits/stdc++.h>
#include <iostream>
using namespace std;
typedef pair<char, int> PAIR;

// Comparator of priority_queue
struct cmp {
    bool operator()(const PAIR& a, const PAIR& b)
    {
        if(a.second < b.second) return true;
          else if(a.second > b.second) return false;
          else return a.first > b.first;
    }
};

void rearrange(char* str, int d)
{
    // Length of the string
    int n = strlen(str);

    // A structure to store a character and its frequency
    unordered_map<char, int> m;

    // Traverse the input string and store frequencies of
    // all characters.
    for (int i = 0; i < n; i++) {
        m[str[i]]++;
        str[i] = '\0';
    }

    // max-heap
    priority_queue<PAIR, vector<PAIR>, cmp> pq(m.begin(),
                                               m.end());

    // Now one by one extract all distinct characters from
    // heap and put them back in str[] with the d
    // distance constraint
    while (pq.empty() == false) {
        char x = pq.top().first;

          // Find the first available position in str[]
          int p = 0;
        while (str[p] != '\0')
            p++;

          // Fill x at p, p+d, p+2d, .. p+(frequency-1)d
        for (int k = 0; k < pq.top().second; k++) {

              // If the index goes beyond size, then string
            // cannot be rearranged.
            if (p + d * k >= n) {
                cout << "Cannot be rearranged";
                exit(0);
            }
            str[p + d * k] = x;
        }
        pq.pop();
    }
}

int main()
{
    char str[] = "aabbcc";

      // Function call
    rearrange(str, 3);
    cout << str;
}
```

Algorithmic Paradigm: Greedy Algorithm
Time Complexity: Time complexity of above implementation is O(n + mLog(MAX)). Here n is the length of str, m is the count of distinct characters in str[] and MAX is the maximum possible different characters.
This article is contributed by **RAVI TEJA NAMA**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

767. Reorganize String

Medium 4078 168 Add to List Share

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*.

Example 1:

```
Input: s = "aab"
Output: "aba"
```

Example 2:

```
Input: s = "aaab"
Output: ""
```

```cpp
class Solution{
    public: string reorganizeString(string s)
    {
        unordered_map<char,int> m;
        priority_queue<pair<int,char>> pq;
        string output="";
        for(auto i:s) m[i]++; //store count of each character in s in a map
        for(auto &i:m) pq.push({i.second,i.first}); //push all the elements in a Max heap sorted by frequency of characters
        while(pq.size()>1)
        {
            auto top1 = pq.top(); //max count store in top1
            pq.pop();
            auto top2 = pq.top(); //second max count store in top2
            pq.pop();
            output+=top1.second; //add the char having highest count in output string
            output+=top2.second;//add the char having the second highest count in the output string
            if(--top1.first>0)  pq.push(top1); //if the frequency after decrementing is still greater than 0, push again
            if(--top2.first>0)  pq.push(top2); //if the frequency after decrementing is still greater than 0, push again
        }
        if(pq.size()) //if there is something left in pq, then
            if(pq.top().first==1) output+=pq.top().second; //if the frequemcy of top element is 1 then add that element
            else return ""; //otherwise, your string has multiple same characters and not enough characters which help.
        return output; //at the end return the output.
    }

};
```

Minimum characters to be added at front to make string palindrome

Hard Accuracy: 43.94% Submissions: 1166 Points: 8

Given string **str** of length **N**. The task is to find the minimum characters to be added at front to make string palindrome.

Note: A palindrome is a word which reads the same backward as forward. Example : madam.

Example 1:

Input:
S = "abc"
Output: 2
Explanation:
Add 'b' and 'c' at front of above string to make it palindrome : "cbabc"

```
8
9  class Solution {
10     public:
11     bool f(string s){
12         int i=0, j=s.size()-1;
13         while(i<=j) if(s[i++]!=s[j--]) return false;
14         return true;
15     }
16     int minChar(string str){
17         int i=0;
18         int count=0;
19         while(i<str.size()){
20             if(f(str.substr(0,i+1))) count = max(count,i+1);
21             i++;
22         }
23         return str.size()-count;
24     }
25 };
26
27
28
```

97. Interleaving String

Medium 3500 185 Add to List Share

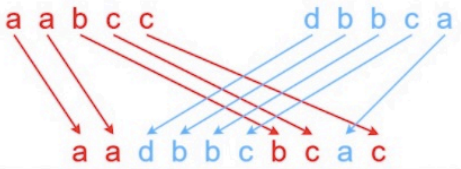
Given strings `s1`, `s2`, and `s3`, find whether `s3` is formed by an **interleaving** of `s1` and `s2`.

An **interleaving** of two strings `s` and `t` is a configuration where they are divided into **non-empty** substrings such that:

- $s = s_1 + s_2 + ... + s_n$
- $t = t_1 + t_2 + ... + t_m$
- $|n - m| <= 1$
- The **interleaving** is $s_1 + t_1 + s_2 + t_2 + s_3 + t_3 + ...$ or $t_1 + s_1 + t_2 + s_2 + t_3 + s_3 + ...$

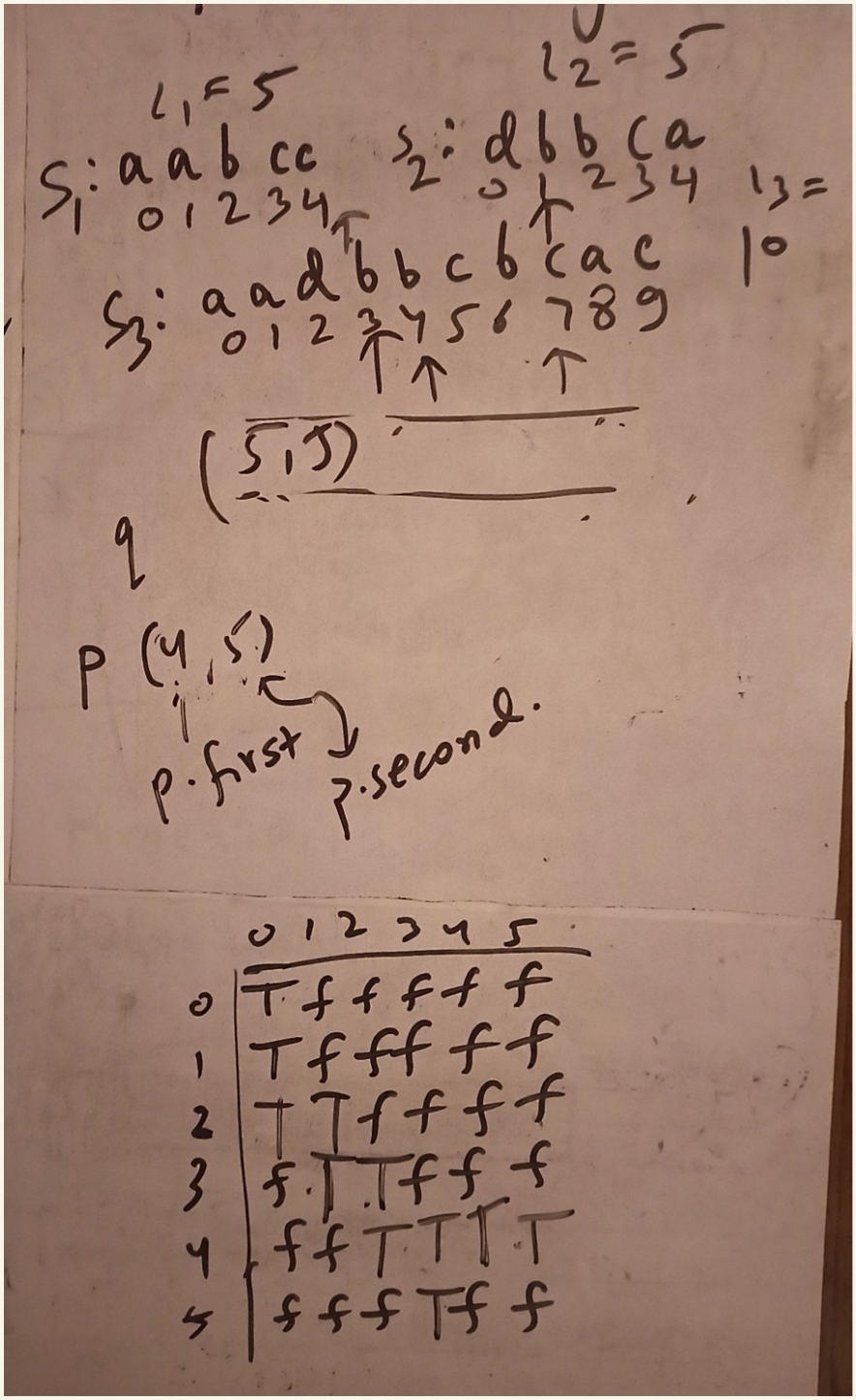
Note: `a + b` is the concatenation of strings `a` and `b`.

Example 1:



Input: s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac"
Output: true

```cpp
#define vb vector<bool>
#define vbb vector<vb>
class Solution {
    public:
    bool isInterleave(string s1, string s2, string s3)
    {
        int l1 = s1.size(), l2 = s2.size(), l3 = s3.size();
        if (l1 + l2 != l3) return false; //return false if sum of lengths of s1 and s2 is not s3
        vbb visited(l1 + 1, vb(l2 + 1, false));
        //create a 2D vector and preinitialise it with false.
        queue<pair<int, int>> q; //create a queue of pair of int and int.
        q.push({0,0});//push 0,0 in queue initially
        while(!q.empty()){ //till the queue not becomes empty
            auto p = q.front();q.pop(); //store queue front in p and pop queue front and do BFS
            if (p.first == l1 and p.second == l2)return true;
            if (visited[p.first][p.second]) continue; //if it is already visited, continue
            //if first is less than l1 and if string indexed at p.first in first string =
            if(p.first<l1 and s1[p.first]==s3[p.first+p.second]) q.push({p.first + 1, p.second});
            if(p.second<l2 and s2[p.second]==s3[p.first+p.second]) q.push({p.first, p.second + 1});
            visited[p.first][p.second] = true;
        }
        return false;
    }
};
```



We are doing a BFS and we are doing so in the interleaved fashion storing the two strings index counts in the pair
And we are checking for the string A and String B that if its corresponding index is matching, and then we push the one more index corresponding to the STRING and then if it matches in s3 the sum of first and second then we push the corresponding index.

and mark visited true, and do all the checking if the index is visited for the very first time and not before that .

Generate IP Addresses

Medium Accuracy: 43.42% Submissions: 14174 Points: 4

Given a string **S** containing only digits, Your task is to complete the function **genIp()** which returns a vector containing all possible combination of valid IPv4 ip address and takes only a string **S** as its only argument.
Note : Order doesn't matter.

For string 11211 the ip address possible are
1.1.2.11
1.1.21.1
1.12.1.1
11.2.1.1

Example 1:

```
Input:
S = 1111
Output: 1.1.1.1
```

Your Task:

Your task is to complete the function **genIp()** which returns a vector containing all possible combination of valid IPv4 ip address in sorted order and takes only a string **S** as its only argument.

Expected Time Complexity: O(N * N * N * N)
Expected Auxiliary Space: O(N * N * N * N)

```cpp
// } Driver Code Ends
/*You are required to complete this method*/

class Solution{
  public:
    vector<string> genIp(string &s) {
        vector<string>res;
        string ans;
        for(int a=1;a<=3;a++)
            for(int b=1;b<=3;b++)
                for(int c=1;c<=3;c++)
                    for(int d=1;d<=3;d++)
                        if(a+b+c+d==s.length()){
                            int w=stoi(s.substr(0,a)),x=stoi(s.substr(a,b)), y=stoi(s.substr(a+b,c)),
                            z=stoi(s.substr(a+b+c,d));
                            if(w<=255 and x<=255 and y<=255 and z<=255)
                                if((ans=to_string(w)+"."+to_string(x)+"."+to_string(y)+"."+
                                to_string(z)).size()==s.length()+3)
                                    res.push_back(ans);
                        }
        return res;
    }
};


} // } Driver Code Ends
```

Validate an IP Address

Medium Accuracy: 29.04% Submissions: 97285 Points: 4

Write a program to Validate an IPv4 Address. According to Wikipedia, IPv4 addresses are canonically represented in dot-decimal notation, which consists of four decimal numbers, each ranging from 0 to 255, separated by dots, e.g., 172.16.254.1 . The generalized form of an IPv4 address is **(0-255).(0-255).(0-255).(0-255).** Here we are considering numbers only from 0 to 255 and any additional leading zeroes will be considered invalid.

Your task is to complete the function **isValid** which returns 1 if the ip address is valid else returns 0. The function takes a string **s** as its only argument .

Example 1:

```
Input:
ip = 222.111.111.111
Output: 1
```

Example 2:

```
Input:
ip = 5555..555
Output: 0
Explanation: 5555..555 is not a valid
ip address, as the middle two portions
are missing.
```

Your Task:
Complete the function **isValid()** which takes the string **s** as an input parameter and returns 1 if this is a valid ip address otherwise returns 0.

Expected Time Complexity: O(N), N = length of string.
Expected Auxiliary Space: O(1)

```cpp
class Solution
{
    public:
        int isValid(string s)
        {
            int dotcount=0;
            //to store the count of the dots
            string a=""; //srtring wchich will store the answer
            for(int i=0; i<s.size(); i++)
            {
                if(s[i]>='0' && s[i]<='9') //if the element lies withing the 0-9 range
                {
                    //if string a is empty and string s first character is a 0,
                    //and i has not crossed size of string s and next character is not a dot
                    //then return a 0
                    if(a=="" and s[i]=='0' and i<s.size()-1 and s[i+1]!='.') return 0;
                    //otherwise keep on adding elemenet of string into a.
                    a+=s[i];
                }
                else if(s[i]=='.') //if the character is a dot
                {
                    dotcount++; //increment the dot count
                    if(a!="")
                    {
                        int c=stoi(a); //convert string to integer and if the int value
                        //is less than 0 or greater than 255 return a 0
                        if(c>255 or c<0) return 0;
                    }
                    //if the result string a is null then return a 0
                    else if(a=="") return 0;
                    //after encountered . if it is within the range 0 and 255 then you make a as 0.
                    a="";
                }
            }
            //at the end of the for loop, if the doc count is something other than 3, then return 0
            if(dotcount!=3) return 0;
            //if everything goes smooth, return a 1
            return 1;
        }
```

301. Remove Invalid Parentheses

Hard 4304 210 Add to List Share

Given a string `s` that contains parentheses and letters, remove the minimum number of invalid parentheses to make the input string valid.

Return *all the possible results*. You may return the answer in **any order**.

Example 1:

```
Input: s = "()())()"
Output: ["(())()","()()()"]
```

Example 2:

```
Input: s = "(a)())()"
Output: ["(a())()","(a)()()"]
```

Example 3:

```
Input: s = ")("
Output: [""]
```

Constraints:

- `1 <= s.length <= 25`
- `s` consists of lowercase English letters and parentheses

```cpp
class Solution {
public:
    bool isBalanced(string s)
    {
        int count=0;
        for(auto x: s){
            if(x=='(') count++;
            else if(x==')') count--;
            if(count<0) return false;
        }
        return count==0;
    }
    map<string,bool> visited;
    vector<string> removeInvalidParentheses(string s)
    {
        queue<string> q;
        vector<string> result;
        q.push(s);
        int found=0;
        while(!q.empty())
        {
            auto u= q.front(); q.pop();
            if(visited[u]) continue;
            visited[u]=true;
            if(isBalanced(u)) found=1, result.push_back(u);
            if(found) continue;
            for(int i=0;i<u.size();i++)
                if(u[i]=='(' or u[i]==')'){
                    auto v = u.substr(0,i)+u.substr(i+1,u.size());
                    q.push(v);
                }
        }
        return result;

    }
};
```

Circle of strings

Medium Accuracy: 44.86% Submissions: 10897 Points: 4

Given an array of lowercase strings **A[]** of size **N**, determine if the strings can be chained together to form a circle.
A string **X** can be chained together with another string **Y** if the last character of **X** is same as first character of **Y.** If every string of the array can be chained, it will form a circle.

For example, for the array arr[] = {"for", "geek", "rig", "kaf"} the answer will be Yes as the given strings can be chained as "for", "rig", "geek" and "kaf"

Example 1:

```
Input:
N = 3
A[] = { "abc", "bcd", "cdf" }
Output:
0
Explaination:
These strings can't form a circle
because no string has 'd'at the starting index.
```

Example 2:

```
Input:
N = 4
A[] = { "ab" , "bc", "cd", "da" }
Output:
1
Explaination:
These strings can form a circle
of strings.
```

Your Task:
You don't need to read input or print output. Your task is to complete the function **isCircle()** which takes the length of the array **N** and the array **A** as input parameters and returns **1** if we can form a circle or **0** if we cannot.

Expected Time Complexity: O(N)
Expected Auxiliary Space: O(N)

check these two concepts before proceeding..
Steps:
1) Making a class graph having member function (**addEdge, isConnected, DFS, isEulerian, canBeChained**) will be really helpful..It makes tracking through the code easier and if you happen to come and see after a few weeks, the code will be very much understandable.
2) Keep track on inBound Dependency of a node, you can make this in the constructor. Basically this will keep track on the inbound dependencies of a node,Inshort it'll keep track of nodes pointing to it.
3) Use Kosaraju's Algo or Tarjan's algo to find SCC (**Strongly Connected Components**). If you find some disconnected components return false. Reverse the graph and do the same stuff so we can be sure that the graph has all components connected and all the nodes can be traversed, then return true;
4) Construct the graph using the class and the string of words. Then check if isConnected is false, return false cuz there will be no Eulerian Circuit . Otherwise, return false if the dependency of a node is equal to its list size, which means its not pointing to anyone but itself getting pointed. Its a Euler Concept so be sure to read that first. Now if all shit goes well, we can safely return our bad boy as true

```cpp
#define vb vector<bool>
#define vvb vector<vb>
#define vi vector<int>
#define vvi vector<vi>
class Solution
{
    int graph[26][26];
    void dfs(int v, vb &visited)
    {
        visited[v] = 1;
        for(int i=0 ; i<26 ; i++)
            if(graph[v][i] and !visited[i])
                dfs(i,visited);
    }
    public:
    int isCircle(int N, vector<string> A)
    {
        memset(graph,0,sizeof graph);
        vi in(26,0); //indegree
        vi out(26,0); //outdegree
        for(int i=0 ; i<N ; i++)
        {
            int a = A[i][0]-'a';
            int b = A[i].back()-'a';
            in[a]++;
            out[b]++;
            graph[a][b] = 1;
        }
        for(int i=0;i<26;i++) if(in[i] != out[i]) return 0;
        vb visited(26,0);
        int i;
        for(i=0 ; i<26 ; i++) if(in[i]) break;
        dfs(i,visited);
        for(int i=0 ; i<26 ; i++) if(in[i] and !visited[i]) return 0;
        return 1;
    }
};
```

We are creating a Graph adjacency matrix of 26 cross 26 for all the Alphabets and then we are performing DFS on the graph.
If the particular character is visited, leave it, if it is not visited, mark It as visited if the value at the vertex indexed at I is set then visit its neighbours and perform recursive DFS calls.

IsCircle is checking, whether you can form euler circuit in the graph
Or not. We initialise the graph by 0 and then we create two vectors indegree and out degree and pre initialise both of them by 0.
Then we traverse the string array given to us and then for all strings
In this array we write a for loop and then we store the integer value associated with the first and last index of every string in a and b resp

Then we update in degree and out degree of both a and b by 1 and set the graph[a][b] as 1 this means we have stored it's entry in the graph. Now we will write one more for loop for every character, it will check, if at any point of time if the in degree of I is not equal to the out degree of I, then we will return 0. So it is kind of a trap. Now we will make a visited vector of bool type.
For evert character if in degree of some character is not 1 then we will break at that point and do a DFS on that character and now after this dis call gets completed, we will once again check ki if for any character there is. In degree but it is not yet visited we will return false.

At the last, if everything is ok, we will return true.

Length of the longest substring
Medium Accuracy: 50.99% Submissions: 29085 Points: 4

Given a string **S**, find the length of the longest substring without repeating characters.

Example 1:

```
Input:
S = "geeksforgeeks"
Output:
7
Explanation:
Longest substring is
"eksforg".
```

Example 2:

```
Input:
S = "abdefgabef"
Output:
6
Explanation:
Longest substring are
"abdefg" , "bdefga" and "defgab".
```

Your Task:
You don't need to take input or print anything. Your task is to complete the function **longestUniqueSubsttr()** which takes a string **S** as and **returns** the length of the longest substring.

Expected Time Complexity: O(|S|).
Expected Auxiliary Space: O(K) where K is constant.

Both are same questions one is GFG and one is Leetcode

3. Longest Substring Without Repeating Characters

Medium 20815 937 Add to List Share

Given a string `s`, find the length of the **longest substring** without repeating characters.

Example 1:

```
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the
length of 3.
```

Example 2:

```
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length
of 1.
```

Example 3:

```
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the
length of 3.
Notice that the answer must be a substring,
"pwke" is a subsequence and not a substring.
```

```cpp
class Solution {
public:
    int lengthOfLongestSubstring(string s)
    {
        vector<int> freq(128);
        //start from index 0 and now left and right both are 0
        int left = 0;
        int right = 0;
        int ans = 0;//here we will store the length of the longest substring without repeating characters
        while (right < s.length()) //until the right reaches the end of the string, keep going
        {
            char r = s[right]; //store character at right index in r
            freq[r]++; //update the frequency in the frequency vector
            while (freq[r] > 1){
            /*once you see that the frequency of the rightmost character becomes more than 1,
            stop aquiring and start releasing from the left until your freq becomes exactly 1*/
                char l = s[left]; //now store the leftmost character in the character l
                freq[l]--; //and now decrement the frequency by 1 as we have dropped off that character from our window.
                left++; //increment left as the new window is shifted one place towards the right.
            }
            //update your answer by the size of your window between right and left and store the max of the current ans and the new one.
            ans = max(ans, right - left + 1);
            //increment your right pointer too and keep on doing the same work
            right++;
        }
        //at the end return your answer
        return ans;
    }
};
```

The **atoi() function** in C takes a string (which represents an integer) as an argument and returns its value of type int. So basically the function is used to convert a string argument to an integer.

Syntax:

```
int atoi(const char strn)
```

Parameters: The function accepts one parameter **strn** which refers to the string argument that is needed to be converted into its integer equivalent.

Return Value: If strn is a valid input, then the function returns the equivalent integer number for the passed string number. If no valid conversion takes place, then the function returns zero.

Implement Atoi

Medium Accuracy: 32.9% Submissions: 87475 Points: 4

Your task is to implement the function **atoi**. The function takes a string(str) as argument and converts it to an integer and returns it.

Note: You are not allowed to use inbuilt function.

Example 1:

```
Input:
str = 123
Output: 123
```

Example 2:

```
Input:
str = 21a
Output: -1
Explanation: Output is -1 as all
characters are not digit only.
```

Your Task:
Complete the function **atoi()** which takes a string as input parameter and returns integer value of it. if the input string is not a numerical string then returns -1.

Expected Time Complexity: O(|S|), |S| = length of string str.
Expected Auxiliary Space: O(1)

Constraints:
1 ≤ length of S ≤ 10

```cpp
class Solution{
  public:
    int atoi(string str)
    {
      int n = str.size();
       int ans=0;
       if (str[0]=='-') //for negetive numbers
       {
            for (int i=1;i<n;i++) //start from i=1 as first index is minus
            {
                if (!(str[i]>='0' and str[i]<='9')) //if the character is not within 0 and 9
                    return -1; //return -1
                ans=ans*10+(str[i]-'0'); //keep on multiplying it with 10 and adding the character in int
            }
            return ans*-1; //return negetive answer
        }
        else //for positive numbers
        {
            for (int i=0;i<n;i++) //start from i=0 as first index is not minus
            {
                if (!(str[i]>='0' and str[i]<='9')) //if the character is not within 0 and 9
                    return -1; //return -1
                ans=ans*10+(str[i]-'0'); //keep on multiplying it with 10 and adding the character in int
            }
            return ans; //return answer
        }
    }
};
```

Search Pattern (KMP-Algorithm)

Medium Accuracy: 66.9% Submissions: 4845 Points: 4

Given two strings, one is a text string, **txt** and other is a pattern string, **pat**. The task is to print the indexes of all the occurences of pattern string in the text string. For printing, Starting Index of a string should be taken as 1.

Example 1:

```
Input:
txt = "batmanandrobinarebat", pat = "bat"
Output: 1 18
Explanation: The string "bat" occurs twice
in txt, one starts are index 1 and the other
at index 18.
```

Example 2:

```
Input:
txt = "abesdu", pat = "edu"
Output: -1
Explanation: There's not substring "edu"
present in txt.
```

Your Task:
You don't need to read input or print anything. Your task is to complete the function **search()** which takes the string txt and the string pat as inputs and returns an array denoting the start indices (1-based) of substring pat in the string txt.
Note: Return an empty list incase of no occurences of pattern. Driver will print -1 in this case.

Expected Time Complexity: O(|txt|).
Expected Auxiliary Space: O(|txt|).

Constraints:
$1 \le |txt| \le 10^5$
$1 \le |pat| < |S|$

The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to O(n). The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match. Let us consider below example to understand this.

```
txt = "AAAAABAAABA"
pat = "AAAA"

We compare first window of txt with pat
txt = "AAAAABAAABA"
pat = "AAAA"  [Initial position]
We find a match. This is same as Naive String Matching.

In the next step, we compare next window of txt with pat.
txt = "AAAAABAAABA"
pat =  "AAAA" [Pattern shifted one position]
This is where KMP does optimization over Naive. In this
second window, we only compare fourth A of pattern
with fourth character of current window of text to decide
whether current window matches or not. Since we know
first three characters will anyway match, we skipped
matching first three characters.

Need of Preprocessing?
An important question arises from the above explanation,
how to know how many characters to be skipped. To know this,
we pre-process pattern and prepare an integer array
lps[] that tells us the count of characters to be skipped.
```

Preprocessing Overview:

- KMP algorithm preprocesses pat[] and constructs an auxiliary **lps[]** of size m (same as size of pattern) which is used to skip characters while matching.
- **name lps indicates longest proper prefix which is also suffix.**. A proper prefix is prefix with whole string **not** allowed. For example, prefixes of "ABC" are "", "A", "AB" and "ABC". Proper prefixes are "", "A" and "AB". Suffixes of the string are "", "C", "BC" and "ABC".
- We search for lps in sub-patterns. More clearly we focus on sub-strings of patterns that are either prefix and suffix.
- For each sub-pattern pat[0..i] where i = 0 to m-1, lps[i] stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].

```
lps[i] = the longest proper prefix of pat[0..i]
          which is also a suffix of pat[0..i].
```

Note : lps[i] could also be defined as longest prefix which is also proper suffix. We need to use properly at one place to make sure that the whole substring is not considered.

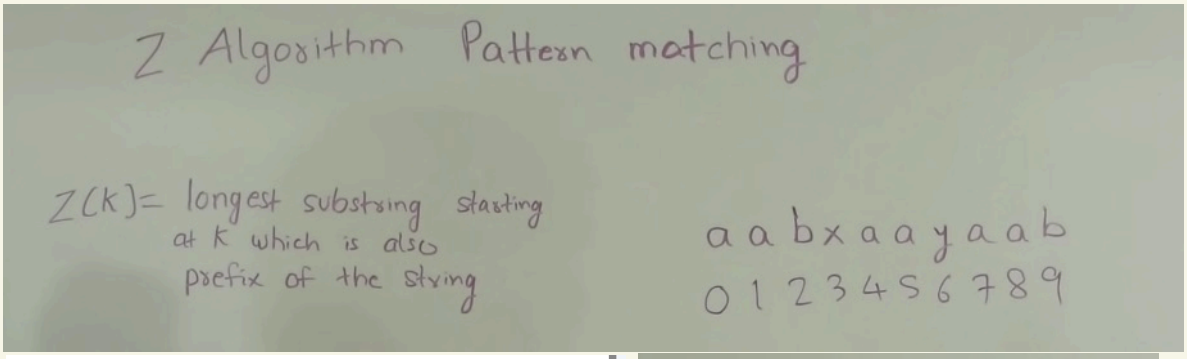
```
Examples of lps[] construction:
For the pattern "AAAA",
lps[] is [0, 1, 2, 3]

For the pattern "ABCDE",
lps[] is [0, 0, 0, 0, 0]

For the pattern "AABAACAABAA",
lps[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

For the pattern "AAACAAAAAC",
lps[] is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

For the pattern "AAABAAA",
lps[] is [0, 1, 2, 0, 1, 2, 3]
```

What we are using is we are creating a longest proper prefix LPS and storing the Frequency of repeating prefixes and if the pattern matches we are updating the length and incrementing I and if it is not matching we are checking if it is the first occurence or 0 length then we are setting LS[I] as 0 and moving I one step further and otherwise if The length is not 0 but pat[I] is not matching with pat[len] then in that case, we are updating the length of the string as ls[len-1] i.e. previous value in the LS array

Now we are creating a KMP search function where we are passing string as well as the Pattern in the function. Now we have stored their lengths and now, we will create a Vector of integer type LS of size pattern length. i.e. n.
Call LPS(pat,LS) and then
Now start from j=0 and I=0. And make a vector ans where we will store the occurences of the pattern in the string. Now we will traverse in the while loop from I=0 to m i.e. size of the string. If the text at ith location is similar to pattern at jth location then in that case You will increment both I and j

if your j has reached the pattern length then in that case, we will push i-j+1 in the answer That would be the first instance of the pattern in the string as we need the beginning of the pattern index which is I-j+1. Also please assign j=previous LS[j-1]
Else if I is less than m and string at index I is not matching with the pattern j. Then in that Case you have to check if j=0 then you have to increment I and otherwise you will update j as LS[j-1] Ie. Keep going back as you have already stored the repeated substring in the Prefix array.
At the end return the answer.

```
class Solution
{
    public:
    void lps(string pat, vector<int> &ls)
    {
        int len =0, i=1;
        ls[0]=0;
        while(i<pat.length())
        {
            if(pat[i]==pat[len]) ls[i++] = ++len;
            else
                if(len ==0) ls[i++] = 0;
                else len=ls[len-1];
        }
    }
    vector <int> search(string pat, string txt)
    {
        int n = pat.length();
        int m = txt.length();
        vector<int> ls(n);
        lps(pat, ls);
        int j=0,i=0;
        vector<int> ans;
        while(i<m)
        {
            if(txt[i]==pat[j]) i++,j++;
            if(j==n)
            {
                ans.push_back(i-j+1);
                j = ls[j-1];
            }
            else if(i<m and txt[i]!=pat[j])
            {
                if(!j) i++;
                else j = ls[j-1];
            }
        }
        return ans;
    }
};
```



Search Pattern (Z-algorithm)

Medium Accuracy: 69.82% Submissions: 4398 Points: 4

Given two strings, one is a text string and other is a pattern string. The task is to print the indexes of all the occurences of pattern string in the text string. For printing, Starting Index of a string should be taken as 1.

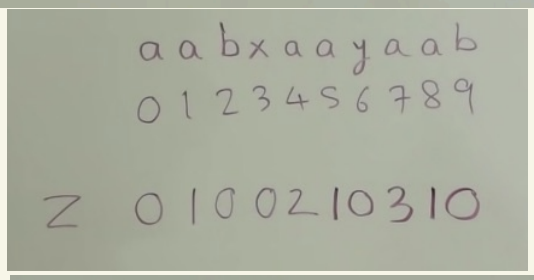
Example 1:

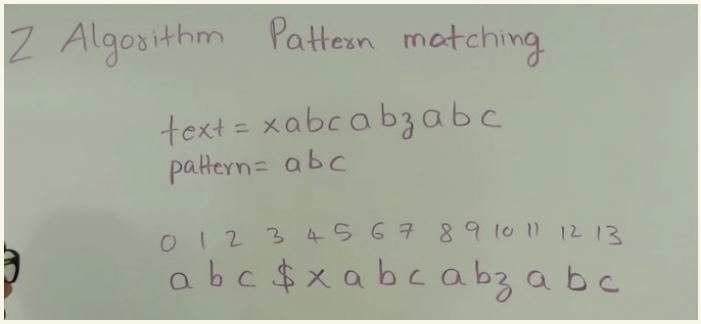
```
Input:
S = "batmanandrobinarebat", pat = "bat"
Output: 1 18
Explanation: The string "bat" occurs twice
in S, one starts are index 1 and the other
at index 18.
```

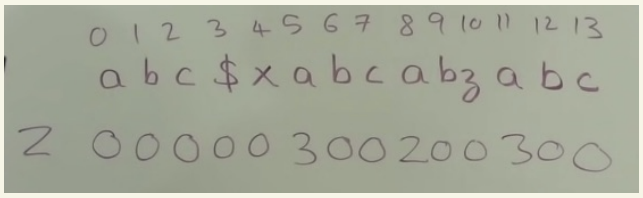
â€‹**Example 2:**

```
Input:
S = "abesdu", pat = "edu"
Output: -1
Explanation: There's not substring "edu"
present in S.
```

Expected Time Complexity: O(|S|).
Expected Auxiliary Space: O(|S|).








```cpp
class Solution
{
    public:
    vector<int> z(string &s)
    {
        int n = s.size();
        vector<int> z(n);
        for (int i=1,l=0,r=0;i<n;i++)
        {
            if (i<=r) z[i]=min(r-i+1, z[i-l]);
            while ( i+z[i]<n and s[z[i]]==s[i+z[i]] )
                z[i]++;
            if (i+z[i]-1>r) l=i, r=i+z[i]-1;
        }
        return z;
    }
    vector <int> search(string pattern, string text)
    {
        string concat = pattern+'#'+text;
        int sz1 = text.size(), sz2 = pattern.size();
        vector<int> v, lps = z(concat);
        for (int i=sz2+1; i<=sz1+sz2;i++)
            if (lps[i]==sz2) v.push_back(i-sz2);
        return v;
    }

};
```

Largest number in K swaps

Medium Accuracy: 46.92% Submissions: 30355 Points: 4

Given a number **K** and string **str** of digits denoting a positive integer, build the largest number possible by performing swap operations on the digits of **str** at most **K** times.

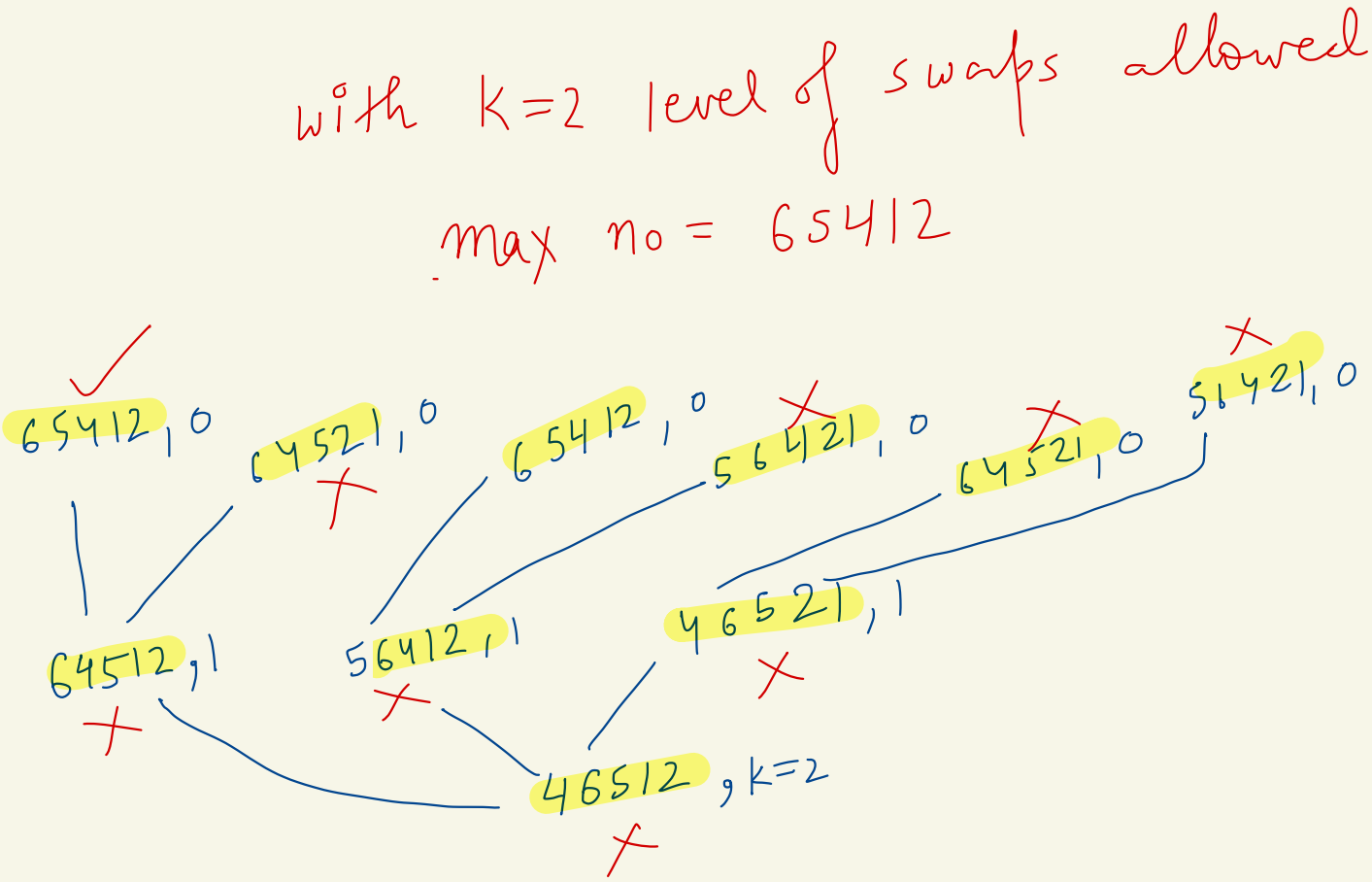
Example 1:

Input:
K = 4
str = "1234567"
Output:
7654321
Explanation:
Three swaps can make the
input 1234567 to 7654321, swapping 1
with 7, 2 with 6 and finally 3 with 5

Expected Time Complexity: O(n!/(n-k)!) , where n = length of input string
Expected Auxiliary Space: O(n)

```
class Solution
{
    public:
    void f(string str, string &max, int k, int pos)
    {
        if(!k) return;
        char maxm = str[pos];
        for(int i=pos+1;i<str.length();i++)
            if(maxm<str[i]) maxm=str[i];
        if(maxm!=str[pos]) k--;
        for(int i=str.length()-1; i>=pos ;i--)
            if(str[i]==maxm)
            {
                swap(str[i], str[pos]);
                if(str.compare(max) > 0) max = str;
                f(str,max,k,pos+1);
                swap(str[i],str[pos]);
            }
    }
    string findMaximumNum(string str, int k)
    {
       string max=str;
       f(str,max,k,0);
       return max;
    }
};
```

What we are doing is using a backtracking based solution



43. Multiply Strings

Medium 3920 1561 Add to List Share

Given two non-negative integers `num1` and `num2` represented as strings, return the product of `num1` and `num2`, also represented as a string.

Note: You must not use any built-in BigInteger library or convert the inputs to integer directly.

Positive Numbers only

Example 1:

```
Input: num1 = "2", num2 = "3"
Output: "6"
```

Example 2:

```
Input: num1 = "123", num2 = "456"
Output: "56088"
```

```
class Solution {
public:
    string multiply(string A, string B)
    {
        int n = A.length(),m = B.length();
        string res(n+m,'0');
        for(int i=n-1;i>=0;i--)
            for(int j=m-1;j>=0;j--)
            {
                int num = (A[i]-'0') * (B[j]-'0') + (res[i+j+1]-'0');
                res[i+j+1] = num%10+'0';
                res[i+j] += num/10;
            }
        for(int i=0;i<res.length();i++)
            if(res[i]!='0') return res.substr(i);
        return "0";
    }
};
```

Multiply two strings

Medium Accuracy: 26.16% Submissions: 86567 Points: 4

Given two numbers as strings s1 and s2. Calculate their Product.
Note: The numbers can be negative.

Numbers can be negetive

Example 1:

```
Input:
s1 = "33"
s2 = "2"
Output:
66
```

Example 2:

```
Input:
s1 = "11"
s2 = "23"
Output:
253
```

```
class Solution
{
   public:
   string multiplyStrings(string s1, string s2)
   {
        int n1 = s1.length(), n2=s2.length();
        if(n1<n2) return multiplyStrings(s2,s1);
        int z = 0,t1=0,t2=0;
        if(s1[0]=='-') z++,t1++;
        if(s2[0]=='-') z++,t2++;
        vector<int>v(n1+n2,0);
        for(int i=n1-1;i>=t1;i--)
        {
            int x = s1[i]-'0', c=0,k=n1+n2+i-n1;
            for(int j=n2-1;j>=t2;j--)
            {
                int y = x*(s2[j]-48)+c+v[k];
                v[k] = y%10;
                c = y/10;
                k--;
            }
            while(c>0)
            {
                int t = v[k] + c;
                v[k]=t%10;
                c = t/10;
                k--;
            }
        }
        int i = 0;
        while(i<n1+n2)
        {
            if(v[i]!=0)break;
            i++;
        }
        if(i==n1+n2)
        {
            string ans="0";
            return ans;
        }
        string ans;
        if(z==1) ans +="-";
        while(i<n1+n2)
        {
            ans += char(v[i]+48);
            i++;
        }
        return ans;
    }

};
```

This is the standard manual multiplication algorithm. We use two nested for loops, working backward from the end of each input number. We pre-allocate our result and accumulate our partial result in there. One special case to note is when our carry requires us to write to our sum string outside of our for loop.

At the end, we trim any leading zeros, or return 0 if we computed nothing but zeros.

```cpp
string multiply(string num1, string num2) {
    string sum(num1.size() + num2.size(), '0');

    for (int i = num1.size() - 1; 0 <= i; --i) {
        int carry = 0;
        for (int j = num2.size() - 1; 0 <= j; --j) {
            int tmp = (sum[i + j + 1] - '0') + (num1[i] - '0') * (num2[j] - '0') + carry;
            sum[i + j + 1] = tmp % 10 + '0';
            carry = tmp / 10;
        }
        sum[i] += carry;
    }

    size_t startpos = sum.find_first_not_of("0");
    if (string::npos != startpos) {
        return sum.substr(startpos);
    }
    return "0";
}
```

Longest Prefix Suffix

Medium Accuracy: 49.39% Submissions: 36492 Points: 4

Given a string of characters, find the length of the longest proper prefix which is also a proper suffix.

NOTE: Prefix and suffix can be overlapping but they should not be equal to the entire string.

Example 1:

```
Input: s = "abab"
Output: 2
Explanation: "ab" is the longest proper
prefix and suffix.
```

Example 2:

```
Input: s = "aaaa"
Output: 3
Explanation: "aaa" is the longest proper
prefix and suffix.
```

Your task:
You do not need to read any input or print anything. The task is to complete the function **lps()**, which takes a string as input and returns an integer.

Expected Time Complexity: O(|s|)
Expected Auxiliary Space: O(|s|)

Constraints:
$1 \leq |s| \leq 10^5$
s contains lower case English alphabets

```cpp
// } Driver Code Ends

//User function template for C++

class Solution{
  public:
    int lps(string s)
    {

        int n = s.length();
        int lps[n];
        int i=0, j=1;
        lps[0]=0;

        while(j<n)
        {
            if(s[i] == s[j]) lps[j++] = i++ + 1 ;
            else
                if(!i) lps[j++]=0;
                else i = lps[i-1];
        }
        return lps[n-1];
    }
};

// } Driver Code Ends
```

Dry run

for string abab
a does not match b so as i==0 so lps[1]=0
now lps array is like 00
i is at a,0 and j is at a,2,
now check that a matches a so lps becomes 0+1 = 1 and i and j are incrememted
i=1 and j=3 and lps array is 001
now b==b so again lps[3]=1+1 and i and j increments
lps becomes 0012 and i=2 and j=4 now j crosses the while loop so we return the lps[4-1]=lps[3]=2

```cpp
class Solution{
  public:
    int lps(string s)
    {

        int n = s.length();
        int lps[n];
        int i=0, j=1;
        lps[0]=0; //initialise lps array for index 0 as 0

        while(j<n)//iterate through the entire n using j
        {
            if(s[i] == s[j]) lps[j++] = i++ + 1 ; //increment i,j and update lps[j]=i+1
            else
                if(!i) lps[j++]=0; //if i reaches 0, then make lps[j] as 0 and increment j
                else i = lps[i-1]; //else store lps[i-1] in i
        }
        return lps[n-1];  //return lps[n-1] in the end.
    }
};
```

Longest substring to form a Palindrome

Hard Accuracy: 41.25% Submissions: 887 Points: 8

Given a string **S** which only contains lowercase alphabets. Find the length of the longest substring of **S** such that the characters in it can be rearranged to form a palindrome.

Example 1:

```
Input:
S = "aabe"
Output:
3
Explanation:
The substring "aab" can be rearranged to
"aba" which is the longest palindrome
possible for this String.
```

Example 2:

```
Input:
S = "adbabd"
Output:
6
Explanation:
The whole string “adbabd” can be
rearranged to form a palindromic substring.
One possible arrangement is "abddba".
Thus, output length of the string is 6.
```

We have to rearrange every length of substring and check Whether it is the longest palindromic subsequence or not and we have to return the number of changes we have to make in the string so that we are able to form the longest palindromic substring.

This question is different from LPS because, here you do not have to find the LPS, but you have arbitrary characters given in the string, and you have to shuffle them in order to make them a palindrome and then check that is this the best LPS you have got after shuffling and store it in a string answer and then keep on checking if you get any longest palindromic substring after rearranging the characters in the string offcourse.

Approach:
We have to keep on storing the characters and their frequency at each given point of time in the string and then try to place them in an order such that they become palindrome.

Suppose my I and j pointers are such
i=0 and j=5 and the string that I have processed is ababb then I am able to see that I have three b and 2 a So what I will do is I will place the b in the middle and try to place a on the left and right and I will do so if my a%2 is 0 to make an odd length palindrome.

aba
Then I will see that we have two more B left so we will place then on the left and right babab now we can see that Till now we have 5 length longest palindromic subsequence.

Can I use DP to solve this problem?
SO suppose from I to j you have found babab as LPS till now and it is Best having length 5, now suppose at 6 we have a character c so can we make it a palindrome again? Or the palindrome will be babab only So we are able to see a pattern here, if at a given point of time, the character at middle of the previous result is same as the character at the new index j then we will insert that into the result by Result = result.substr(0,mid+1)+s[j]+ result.substr(mid+1,j) then our string will become babbab

So storing previous result is looking to be useful. Now we will think that how to handle the odd length palindrome? So we see Once we have formed a palindrome in the previous result, all we need to care is about if the resultant string was even length, then no matter what just insert the upcoming character into middle as it will be placed in the centre, but if you have the string of odd length in that case, you have to think what you can do.

adbabd

a a1	
ad ->a,	a1, d1
Adb—>a,	a1, d1, b1
adba—>aa	a2,b1,d1
Adbab-->baab	a2,b2,d1
adbadb->dbaabd	a2,b2,d2

```cpp
class Solution {
  public:
    int longestSubstring(string s)
    {
        int result = 0, b=0;;
        map<int, int> m;          //To keep track of last index of each XOR operation
        m[b] = -1; //set the first mask as -1.
        for(int i=0;i<s.size(); i++){ //for every char in the string
            int t=s[i]-'a';
            /*Turn the temp-th bit on if character occurs odd number of times
            and turn off the temp-th bit off if the character occurs even number of times*/
            b^=(1<<t);
            if (m[b]) result = max(result,i - m[b]);//pal is found from index[mask] to i
            else m[b] = i;// If x is not found then add its position in the index dict.
            for(int j = 0; j < 26; j++){ // Check for the palindrome of odd length
                int b_ = b^(1<<j);//cancel the occur of a char if it occurs odd number times
                if (m[b_]) result =max(result,i - m[b_]);
            }
        }
        return result;
    }
};
```

Bitmasking Solution

Intuition:

```
s="aabe"
01100001-->a
01100010-->b
01100101-->e

we have a map which keeps everything in sorted order in a form of RB tree
insert <0,-1> in map
t=0 means for a
b=b xor (1<<t) means shift 1 on the left by t bits essentially 2^t so 00000001 xor 00000000 = 00000001
as m[b] is not present in the map, we skip the if part
b=1 now
add m[b]=i in the map
<1,0> is added in the map as b is 1 and i=0
map contains <0,-1> and <1,0> as of now
now run a for loop for all the 26 characters.
b_ = 00000001 xor 00000001 = 00000000

as <0,-1> is already in the map so we need to update the result as max(0,0-(-1)) = 1
no need to check for every j as none of them is present in the map as of now.
otherwise you can check
b_ = 00000001 XOR 00000010 = 11 not present in the map
     00000001 XOR 00000100 = 101 not present in the map
     and so on 1001 10001 100001 and 10000000...0000001 is not present in the map

let us proceed for the second iteration of the for loop i=1

again we see s[1]=a and t=0 again
as we see a was present in the map so we update result = max(1,1-0) = 1 so result=1
we need not go in the else part now
b will be 00000001 xor 00000010  as t has not changed and b was 1 and 1<<1 will be 10
so b = 00000011 i.e. 3
as m[3] is not in the map, we ignore this step and now, go to the else part which says insert it
insert m[3]=1 as i=1
```

```
map now has <0,-1>,<1,0> and <3,1>
for every character in the english alphabets,
check if 00000011 XOR 00000001= 10 is present in the map or not
as we can see 10 is not present in the map
now check if 00000011 XOR 00000010 is present in map or not, we see 00000001 is present in the map
so we update result as result = max(1,1-0) = 1 result remains same
no need to check as 00000011 XOR 00000100 i.e. 111 is not present likewise
00000011 XOR 00001000 i.e. 00001011 is not present in the map and so one.

now let us go for the third iteration i=2
s[2]=b and now t=1
b=1<<1 i.e. shift 1 by 1 bit i.e. b=2 or 00000010

as map does not have 2, so insert <2,2> in the map as i=2
map now has <0,-1>,<1,0>, <2,2> and <3,1>

    if (m[b]) result = max(result,i - m[b]);//pal is found from index[mask] to i
            else m[b] = i;// If x is not found then add its position in the index dict.
            for(int j = 0; j < 26; j++){ // Check for the palindrome of odd length
                int b_ = b^(1<<j);//cancel the occur of a char if it occurs odd number times
                if (m[b_]) result =max(result,i - m[b_]);
            }

now check if m[2] is present in the map, yes it is present, so
result = max(1,2-2) = 1
now for every character in the english dictionary

check if 00000010 XOR 00000001 is presnt in map, 3 is presnt, result = max(1,2-1) = 1
    if 00000010 XOR 00000010 is presnt in the map or not
            yes 0 is presnt result = max(1,2-(-1)) = 3 result=3
    if 00000010 XOR 00000100 is present in the map or not
            no 6 is not present
    if 00000010 XOR 00001000 is present in the map or not
                no 10 is not present
```

```
    this result=3 represents aba is the longest substring which is palindrome

we can see further running for e ki result remains same, i leave this as an exercise for you guys
```

Minimum times A has to be repeated such that B is a substring of it

Medium Accuracy: 51.73% Submissions: 1291 Points: 4

Given two strings **A** and **B**. Find minimum number of times A has to be repeated such that B is a Substring of it. If **B** can never be a substring then return **-1**.

Example 1:

```
Input:
A = "abcd"
B = "cdabcdab"
Output:
3
Explanation:
Repeating A three times (“abcdabcdabcd”),
B is a substring of it. B is not a substring
of A when it is repeated less than 3 times.
```

Example 2:

```
Input:
A = "ab"
B = "cab"
Output :
-1
Explanation:
No matter how many times we repeat A, we can't
get a string such that B is a substring of it.
```

```cpp
//User function Template for C++

class Solution {
  public:
    int minRepeats(string A, string B) {
        int a = A.size(), b = B.size(), k = b/a;
        for(int i=0; i<k-1; i++)
            A += A.substr(0, a);
        if(A.find(B) != string::npos)
            return k;
        A += A.substr(0, a);
        if(A.find(B) != string::npos)
            return k+1;
        return -1;
    }
};
```

KMP algorithm

686. Repeated String Match

Medium 1295 884 Add to List Share

Given two strings `a` and `b`, return *the minimum number of times you should repeat string* `a` *so that string* `b` *is a substring of it*. If it is impossible for `b` to be a substring of `a` after repeating it, return `-1`.

Notice: string `"abc"` repeated 0 times is `""`, repeated 1 time is `"abc"` and repeated 2 times is `"abcabc"`.

Example 1:

```
Input: a = "abcd", b = "cdabcdab"
Output: 3
Explanation: We return 3 because by repeating a
three times "abcdabcdabcd", b is a substring of
it.
```

Example 2:

```
Input: a = "a", b = "aa"
Output: 2
```

```cpp
class Solution {
public:
    int repeatedStringMatch(string a, string b)
    {
        vector<int> prefTable(b.size() + 1); // 1-based to avoid extra checks.
        for (auto sp = 1, pp = 0; sp < b.size();)
            if (b[pp] == b[sp] || pp == 0)  {
                pp = b[pp] == b[sp] ? pp + 1 : 0;
                prefTable[++sp] = pp;
            }
            else pp = prefTable[pp];
        for (auto i = 0, j = 0; i < a.size(); )
        {
            while (j < b.size() && a[(i + j) % a.size()] == b[j]) ++j;
            if (j == b.size()) return (i + j - 1) / a.size() + 1;
            j = prefTable[j];
            i += max(1, j - prefTable[j]);
        }
        return -1;
    }
};
```

Excel Sheet | Part - 1

Medium Accuracy: 47.89% Submissions: 12782 Points: 4

Given a positive integer N, return its corresponding column title as it would appear in an Excel sheet.
For N =1 we have column A, for 27 we have AA and so on.

Note: The alphabets are all in uppercase.

Example 1:

```
Input:
N = 51
Output: AY
```

```cpp
//User function template for C++

class Solution{
    public:
    string ExcelColumn(int N)
    {
        string ans="";
        while(N>0)
        {
            N-=1;
            char temp=(char)((int)'A'+N%26);
            ans=temp+ans;
            N/=26;
        }
        return ans;
    }
```

Count subsequences of type a^i, b^j, c^k

Medium Accuracy: 41.33% Submissions: 9351 Points: 4

Given a string S, the task is to count number of subsequences of the form $a^i b^j c^k$, where i >= 1, j >=1 and k >= 1.

Note:
1. Two subsequences are considered different if the set of array indexes picked for the 2 subsequences are different.
2. For large test cases, output value will be too large, return the answer MODULO 10^9+7

Example 1:

```
Input:
S = "abbc"
Output: 3
Explanation: Subsequences are abc, abc and abbc.
```

â€‹
Example 2:

```
Input:
S = "abcabc"
Output: 7
Explanation: Subsequences are abc, abc,
abbc, aabc abcc, abc and abc.
```

Your Task:
You don't need to read input or print anything. Your task is to complete the function **fun()** which takes the string S as input parameter and returns the number of subsequences which follows given condition.

Expected Time Complexity: O(Length of String).
Expected Auxiliary Space: O(1) .

```cpp
class Solution{
  public:
    int fun(string &s)
    {
        int mod = 1e9+7;
        int a = 0, ab = 0, abc = 0;
        for(auto& c: s)
            if(c=='a') a = (2*1LL*a + 0LL + 1)%mod;
            else if(c=='b') ab = (2*1LL*(ab) + 0LL + a)%mod;
            else if(c=='c')abc = (2*1LL*(abc) + 0LL + ab)%mod;
        return abc;
    }
};
```

String formation from substring

Medium Accuracy: 37.49% Submissions: 2569 Points: 4

Given a string 's', the task is to check if it can be constructed by taking a substring of it and appending multiple copies of the substring together.

Example 1:

```
Input: s = "ababab"
Output: 1
Explanation: It is contructed by
appending "ab" 3 times
```

Example 2:

```
Input: s = "ababac"
Output: 0
Explanation: Not possible to construct
```

User Task:
Your task is to complete the function **isRepeat ()** which takes a single string as input and returns 1 if possible to construct, otherwise 0. You do not need to take any input or print anything.

Expected Time Complexity: O(|s|)
Expected Auxiliary Space: O(|s|)

Constraints:
$1 <= |s| <= 10^5$

View Bookmarked Problems

Company Tags

Amazon MakeMyTrip

```cpp
// } Driver Code Ends
//User function template for C++
class Solution
{
    public:
    int isRepeat(string s)
    {
        int lps[s.length()] = {0};
        int i = 1;
        int len = 0;
        int n = s.length();
        int res  = 0;
        while(i < s.length() ){
            int k = i;
            while(s[i] == s[len]){
                lps[i] = len + 1;
                res = max(res,lps[i]);
                len++;i++;
                }
             if(s[i] != s[len]){
                if(len == 0){
                    lps[i] = 0;i++;
                }else{
                    len = lps[len - 1];
                }
            }

        }
        if(res == 0) return 0;
        string t = s.substr(0,n - res);
        string ans = "";
        int k = n/t.length();
        while(k--){
           ans += t;
        }
        if(ans == s) return 1;
        return 0;
    }
};
```

Number of distict Words with k maximum contiguous vowels

Hard Accuracy: 43.6% Submissions: 1410 Points: 8

Find the number of unique words consisting of lowercase alphabets only of length l that can be formed with at-most K contiguous vowels.

Example 1:

```
Input:
N = 2
K = 0
Output:
441
Explanation:
Total of 441 unique words are possible
of length 2 that will have K( =0) vowels
together, e.g. "bc", "cd", "df", etc are
valid words while "ab" (with 1 vowel) is
not a valid word.
```

Example 2:

```
Input:
N = 1
K = 1
Output:
26
Explanation:
All the english alphabets including
vowels and consonants; as atmost K( =1)
vowel can be taken.
```

Expected Time Complexity: O(N*K)
Expected Auxiliary Space: O(N*K)

```cpp
class Solution
{
  public:

    #define ll long long int
    const int MOD = 1e9 + 7;

    int kvowelwords(int n, int k) {
        vector<vector<ll>> dp(n+1, vector<ll>(k+1));

        for(int i = 0; i <= n; i++)
        {
            for(int j = 0; j <= k; j++)
            {
                if(i == 0) dp[i][j] = 1;
                else
                {
                    dp[i][j] = dp[i-1][k]*21 % MOD;
                    if(j > 0) dp[i][j] = (dp[i][j] + dp[i-1][j-1]*5 % MOD) % MOD;
                }
            }
        }
        return dp[n][k];
    }
};
```

IPL 2021 - Final

Hard Accuracy: 49.0% Submissions: 5074 Points: 8

IPL 2021 Finals are here and it is between the most successful team of the IPL Mumbai Indians and the team striving to garb their first trophy Royal Challengers Banglore. Rohit Sharma, captain of the team Mumbai Indians has the most experience in IPL finals, he feels lucky if he solves a programming question before the IPL finals. So, he asked the team's head coach Mahela Jayawardene for a question. Question is, given a string **S** consisting only of opening and closing parenthesis 'ie **'('** and **')'**, the task is to find out the length of the longest valid parentheses substring.

NOTE: The length of the smallest valid substring **( )** is **2**.

Example 1:

```
Input: S = "(()("
Output: 2
Explanation: The longest valid
substring is "()". Length = 2.
```

Example 2:

```
Input: S = "()(())("
Output: 6
Explanation: The longest valid
substring is "()(())". Length = 6.
```

Your Task:
You don't need to read input or print anything. Complete the function **findMaxLen()** which takes **S** as input parameter and returns the max length.

Constraints:
$1 \leq |S| \leq 10^5$

```cpp
// } Driver Code Ends
// User function template for c++
class Solution {
  public:
    int findMaxLen(string s) {
        int n = s.size();
        int open = 0, close = 0, ans = 0;
        for(int i=0; i<n; i++) {
            if(s[i]=='(') open++;
            else close++;
            if(open==close) ans = max(ans, open+close);
            if(close>open) open = 0, close = 0;
        }
        open = 0, close = 0;
        for(int i=n-1; i>=0; i--) {
            if(s[i]=='(') open++;
            else close++;
            if(open==close) ans = max(ans, open+close);
            if(open>close) open = 0, close = 0;
        }
        return ans;
    }
};

// } Driver Code Ends
```

Rank The Permutations

Medium Accuracy: 34.93% Submissions: 4173 Points: 4

Given a string, **S** find the rank of the string amongst all its permutations sorted lexicographically.The rank can be big. So print it modulo **1000003**.
Note: Return 0 if the characters are repeated in the string.

Example 1:

```
Input: S = "abc"
Output: 1
Explaination: It is the smallest
lexicographically string of the permutation.
```

Example 2:

```
Input: S = "acb"
Output: 2
Explaination: This is the second smallest
lexicographically string of the permutation.
```

Your Task:
You do not need to read input or print anything. Your task is to complete the function **rank()** which takes string S as input parameter and returns the rank modulo 1000003 of the string.

Expected Time Complexity: $O(|S|^2)$
Expected Auxiliary Space: $O(|S|)$

```cpp
// } Driver Code Ends
// User function Template for C++

class Solution{
public:
    int rank(string S){
        // code here
        string str = S;
        int mod = 1000003;
        set<char> st;
        for(auto e: S){
            st.insert(e);
        }
        if(st.size()!=S.size())
            return 0;
        sort(S.begin(), S.end());
        long long int count = 1;
        while(true){
            if(str==S){
                break;
            }
            next_permutation(S.begin(), S.end());
            count++;
        }
        return count%mod;
    }
};

} // } Driver Code Ends
```

Smallest distinct window

Medium Accuracy: 50.29% Submissions: 18199 Points: 4

Given a string '**s**'. The task is to find the **smallest** window length that contains all the characters of the given string at least one time.
For eg. A = "**aabcbcdbca**", then the result would be 4 as of the smallest window will be "**dbca**".

Example 1:

```
Input : "AABBBCBBAC"
Output : 3
Explanation : Sub-string -> "BAC"
```

Example 2:

```
Input : "aaab"
Output : 2
Explanation : Sub-string -> "ab"
```

Example 3:

```
Input : "GEEKSGEEKSFOR"
Output : 8
Explanation : Sub-string -> "GEEKSFOR"
```

Time, space : O(n) sliding window

```cpp
// } Driver Code Ends
class Solution{
    public:
    string findSubString(string str)
    {
        int n= str.length(); //size of the string
        unordered_map < char, int > m; //storing the frequency if each character in map
        int i=0, j=0, maxx= INT_MAX;
        string res;
        for (int i=0; i< n; i++) m[str[i]]= 0; //storing the frequency as 0 initially for all c
        int window_size=0; //maintaining the count
        while (i< n) //while the left is in the limit
        {
            if (!m[str[i]]) window_size++; //if not in map, increase window size.
            m[str[i]]++; //updte the frequency of the character by 1.
            if (window_size==m.size())  //if the # of distinct characters=window size
            {
                while (j<n and m[str[j]]>1) //if j has not reached end, and you have multiple occurences
                {
                    m[str[j]]--;  // shrink the window because duplicate elements
                    j++; //shift the window
                }
                if (maxx > (i-j+1)) //update the max
                {
                    maxx= i-j+1;
                    res= str.substr (j, i-j+1); //store result here
                }
            }
            i++;
        }
        return res;
    }
};
```

Longest valid Parentheses

Hard Accuracy: 48.35% Submissions: 14908 Points: 8

Given a string **S** consisting of opening and closing parenthesis '**(**' and '**)**'. Find length of the longest valid parenthesis substring.

A parenthesis string is valid if:

- For every opening parenthesis, there is a closing parenthesis.
- Opening parenthesis must be closed in the correct order.

Example 1:

```
Input: S = (((
Output: 2
Explaination: The longest valid
parenthesis substring is "()".
```

Example 2:

```
Input: S = )()())
Output: 4
Explaination: The longest valid
parenthesis substring is "()()".
```

Your Task:
You do not need to read input or print anything. Your task is to complete the function **maxLength()** which takes string S as input parameter and returns the length of the maximum valid parenthesis substring.

Expected Time Complexity: O(|S|)
Expected Auxiliary Space: O(|S|)

Constraints:

```cpp
class Solution{
public:
    int maxLength(string S){
        stack<int> s;
        s.push(-1);
        int count=0;
        for(int i=0;i<S.length();i++){
            if(S[i]=='(') s.push(i); //open bracket,push it's index on stack
            else{
                s.pop(); //pop the stack top as you have found first closed bracket.
                if(!s.empty()) //if stack is not empty
                    count=max(count , i-s.top()); //update the count as
                else s.push(i);
            }
        }
        return count;
    }
};
// (())()
// push -1 on the stack stack is -1
// -1 0 1
// i=2
// stack--> -1 0 1 pop top of stack.
// stack is not empty stack is -1 0
// count = max(0,2-0) =2
// now we saw ) , i=3
// pop stack
// stack becomes -1
// as the stack is not empty, we update count as max(2,3-(-1))
// count=4
// now we see ( we push 4 on stack
// stack is -1 4
// we see ) at i=5. we pop top
// stack is -1
// stack is not empty, count =max(4,5-(-1)) = max(4,6)
// count=6
```

Restrictive Candy Crush

Medium Accuracy: 50.67% Submissions: 17712 Points: 4

Given a string **s** and an integer **k**, the task is to reduce the string by applying the following operation:
Choose a group of **k** consecutive identical characters and remove them.

The operation can be performed any number of times until it is no longer possible.

Example 1:

```
Input:
k = 2
s = "geeksforgeeks"
Output:
gksforgks
Explanation:
Modified String after each step:
"geeksforgeeks" -> "gksforgks"
```

Time: O(n)
Space: O(k)

Example 2:

```
Input:
k = 2
s = "geegsforgeeeks"
Output:
sforgeks
Explanation:
Modified String after each step:
"geegsforgeeeks" -> "ggsforgeks" -> "sforgeks"
```

stack
string
Pair class

Your Task:

```cpp
// } Driver Code Ends

class Solution
{
    public:
    string Reduced_String(int k,string s)
    {
        string ans="";
        if(k==1)return ans;
        stack<pair<char,int>> st;
        for(int i=0;i<s.size();i++)
        {
            if(st.empty()) st.push({s[i],1});
            else if(st.top().first==s[i])
            {
                st.push({s[i],st.top().second+1});
                if(k==st.top().second)
                {
                    int x=k;
                    while(x--) {st.pop();}
                }
            }
            else if(st.top().first!=s[i]) st.push({s[i],1});

        }


        while(!st.empty())
        {
            ans+=st.top().first;
            st.pop();
        }
        reverse(ans.begin(),ans.end());
        return ans;
    }
};
```

Next higher palindromic number using the same set of digits

Medium Accuracy: 44.28% Submissions: 8877 Points: 4

Given a palindromic number **N** in the form of string. The task is to find the smallest palindromic number greater than **N** using the same set of digits as in **N**.

Example 1:

```
Input:
N = "35453"
Output:
53435
Explanation: Next higher palindromic
number is 53435.
```

Example 2:

```
Input: N = "33"
Output: -1
Explanation: Next higher palindromic number
does not exist.
```

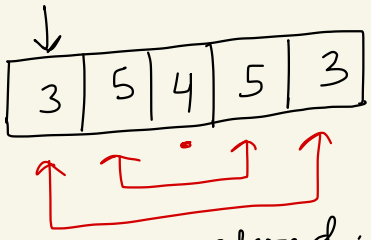
Your Task:
You don't need to read input or print anything. Your task is to complete the function **nextPalin()** which takes the string **N** as input parameters and returns the answer, else if no such number exists returns "-1".

Expected Time Complexity: O(|N|log|N|)
Expected Auxiliary Space: O(1)

Constraints:
$1 \le |N| \le 10^5$

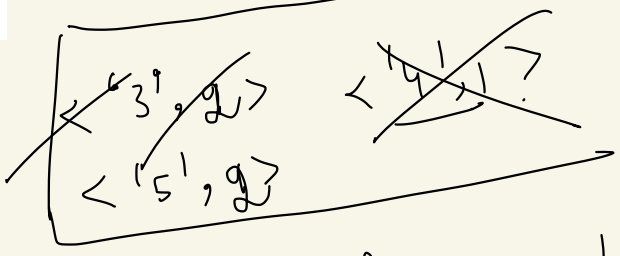
N = "35453"

output → next palindrome if possible using same characters of the string



s[0] ↳ temp.

use a map = unordered. <char, int> & store fr of each char



check if the count of next element in map is > 1

53435.

else return -1

↓

this approach won't work

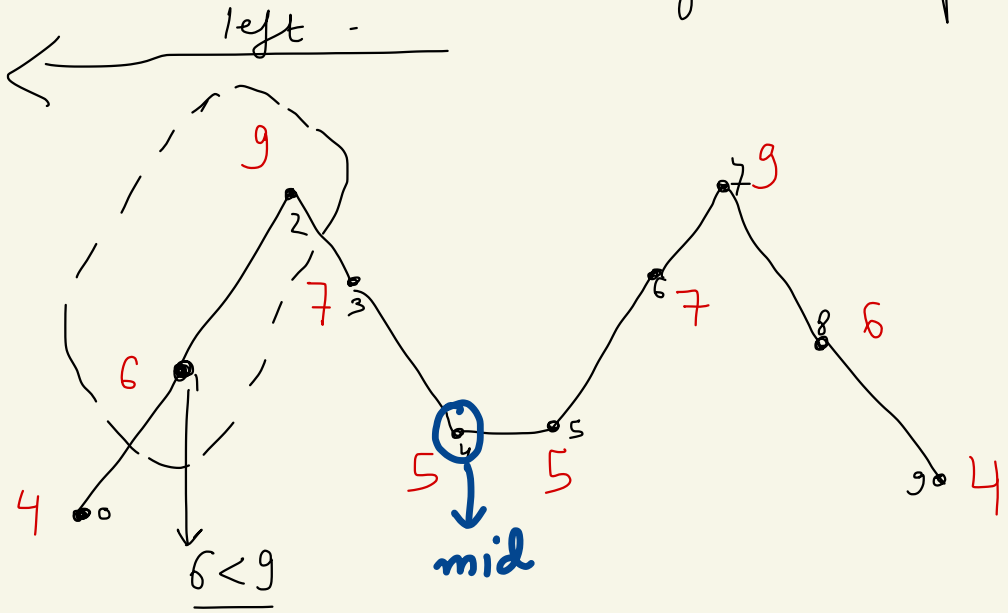
n = "121" if no of digits <= 3 → return -1
as next greater pal with same digits is not possible.

n = "469755794"

since the number can be so large it can't fit in long long int

we need to do some sort of string manipulation

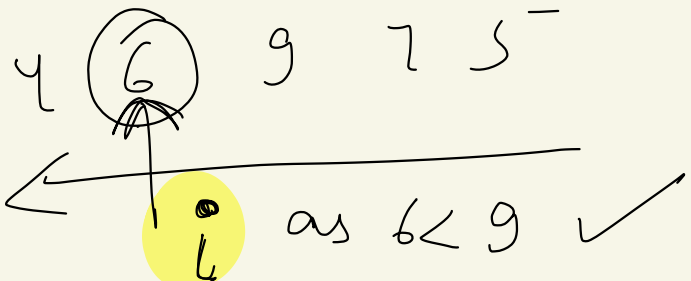
Idea.



1) find mid → $\frac{n}{2} - 1 = \frac{10}{2} - 1 = 4$

2) start traversing from mid → left side s.t.

we saw



3) Now start traversing from 9 to 5.
i.e i+1 to mid

search smallest digit greater than 6.
i.e next greater element on the right

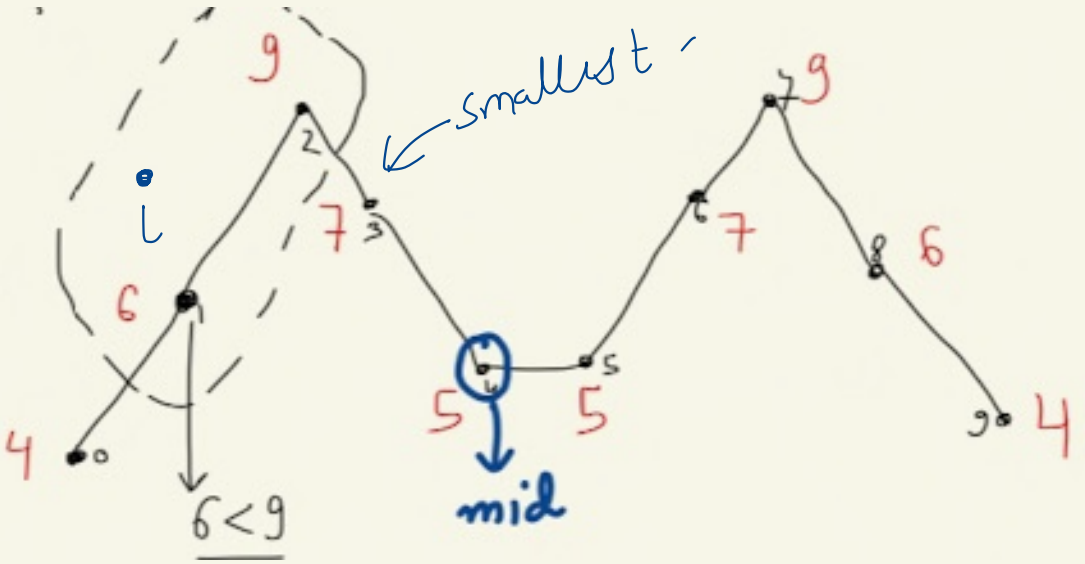
in range.

smallest

9 7 5

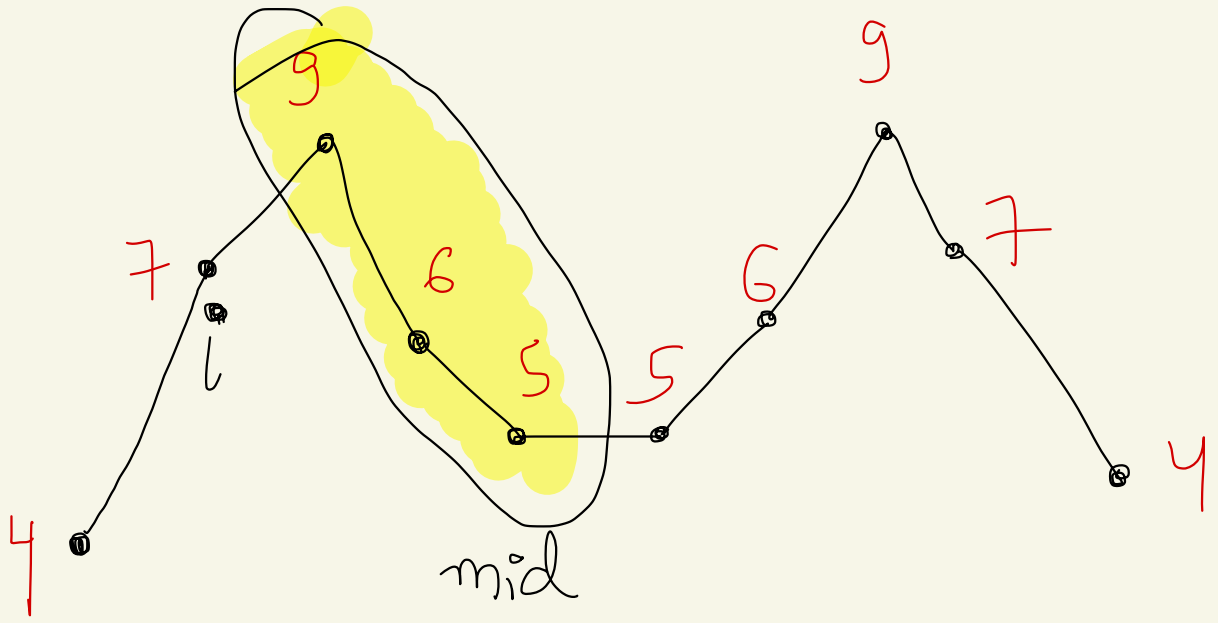
we see it's 7. ✓ as 7 > 6

if no such element is found
return -1 ✓



swap(i, smallest).

swap(n-1-i, n-1-smallest).



reverse elements in range (i+1, mid)

if n is even also we need to reverse.

range(mid+1, n-i-2)

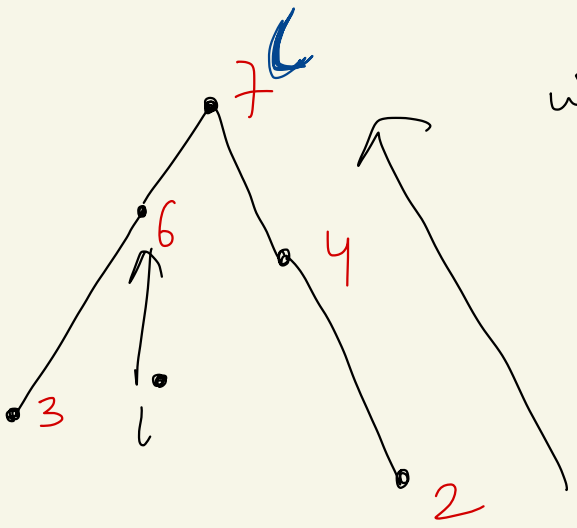
else if n is odd.

reverse

range(mid+2, n-i-2)

This Question is an extension of next permutation

Suppose you have a number



what will be the next permutation ?

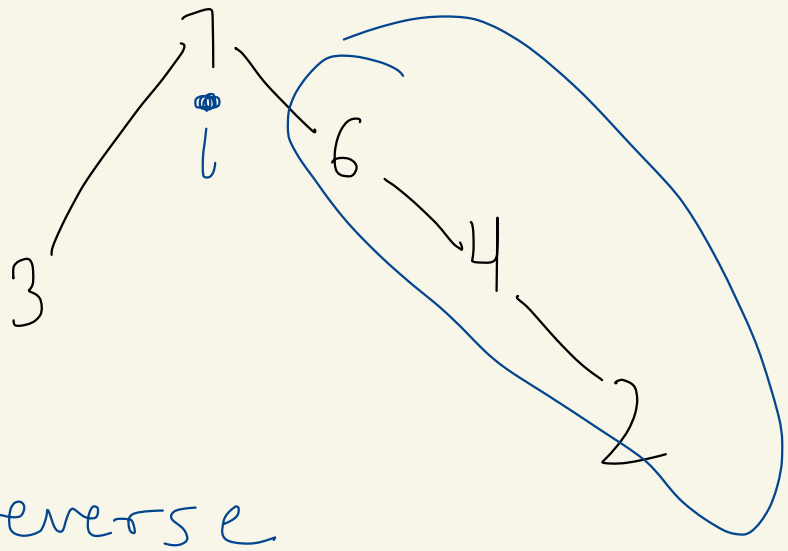
traverse from n-2 to 0 & find the first occurence where $a[i] < a[i+1]$

$4 < 2$ X

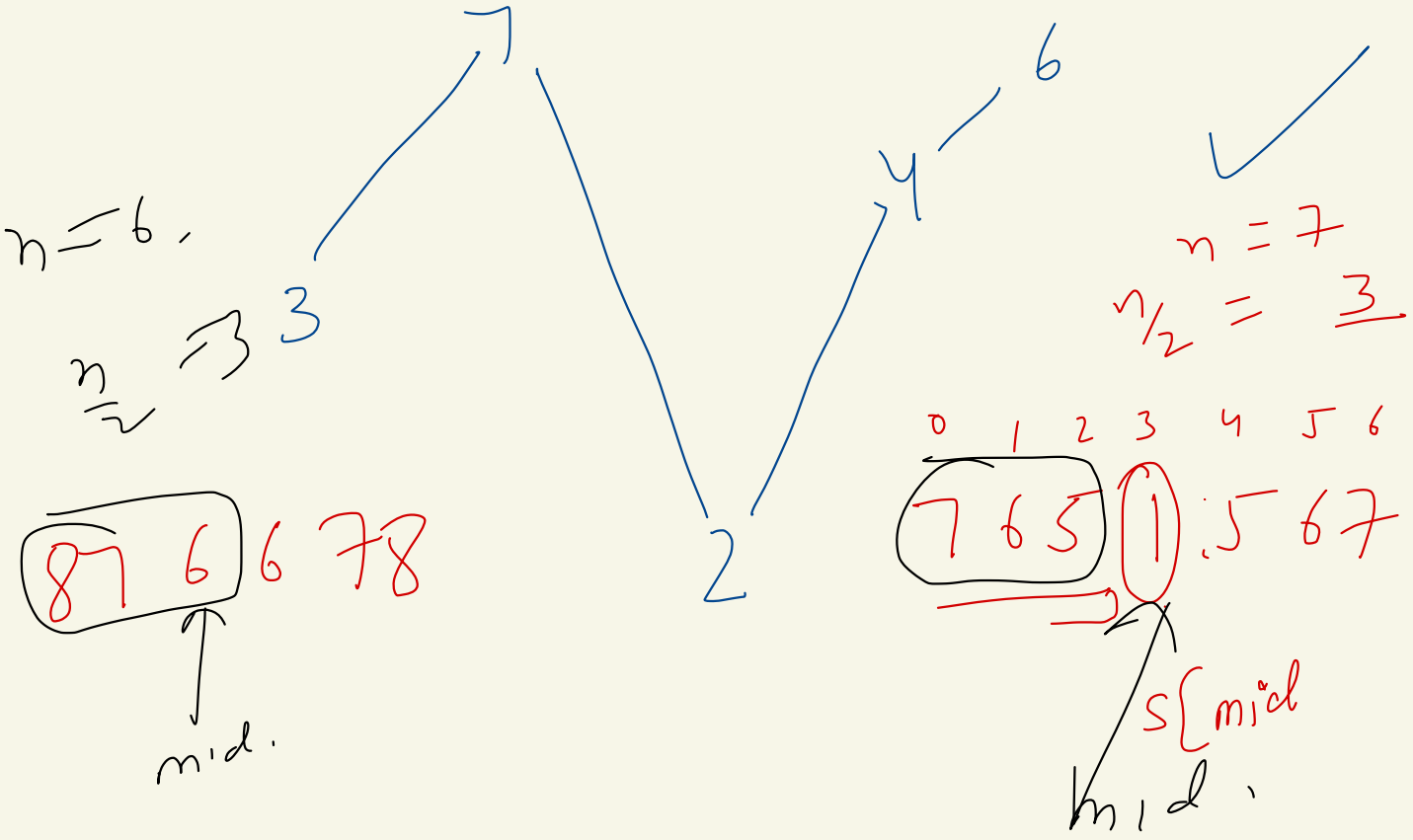
$7 < 4$ X

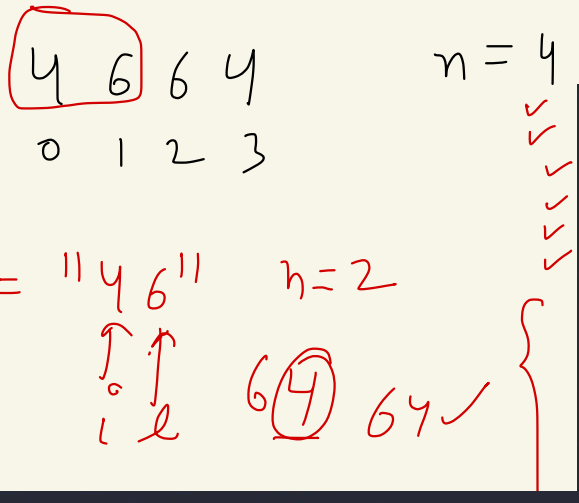
$6 < 7$ ✓

now swap $a[i]$ with the first occurence from n-1 to i+1 where $a[i] < a[k]$ ✓



reverse






```
string nextPalin(string s)
{
    int n = s.size();
    int mid=n/2;
    if(n<=3) return "-1";
    string final,half,result;
    if(!n%2) //n is even
    {
        half = s.substr(0,mid);
        result = nextPermutation(half);
        if(result=="-1") return result;
        final = result;
        reverse(result.begin(),result.end());
        final+=result;
    }
    else //n is odd
    {
        half = s.substr(0,mid);
        result = nextPermutation(half);
        if(result=="-1") return result;
        final = result;
        final = result + s[mid];
        reverse(result.begin(),result.end());
        final+=result;

    }
    return final;
}
```

```
string nextPermutation(string &a)
{
    int n=a.size(),l,i;
    for(i=n-2;i>=0;i--) if(a[i]<a[i+1]) break;
    if(i<0) return "-1";
    else
    {
        for(l=n-1;l>i;l--)if(a[i]<a[l]) break;
        swap(a[i],a[l]);
        reverse(a.begin()+i+1,a.end());
    }
    string ans="";
    for(int i=0;i<a.size();i++) ans+=a[i];
    return ans;
}
```

Next higher palindromic number using the same set of digits

Medium Accuracy: 44.28% Submissions: 8877 Points: 4

Given a palindromic number **N** in the form of string. The task is to find the smallest palindromic number greater than **N** using the same set of digits as in **N**.

Example 1:

```
Input:
N = "35453"
Output:
53435
Explanation: Next higher palindromic
number is 53435.
```

Example 2:

```
Input: N = "33"
Output: -1
Explanation: Next higher palindromic number
does not exist.
```

Your Task:
You don't need to read input or print anything. Your task is to complete the function **nextPalin()** which takes the string **N** as input parameters and returns the answer, else if no such number exists returns "-1".

Expected Time Complexity: O(|N|log|N|)
Expected Auxiliary Space: O(1)

```cpp
// } Driver Code Ends
//User function template for C++
class Solution{
    string nextPermutation(string &a){
        int n=a.size(),l,i;
        for(i=n-2;i>=0;i--) if(a[i]<a[i+1]) break;
        if(i<0) return "-1";
        else{
            for(l=n-1;l>i;l--)if(a[i]<a[l]) break;
            swap(a[i],a[l]);
            reverse(a.begin()+i+1,a.end());
        }
        string ans="";
        for(int i=0;i<a.size();i++) ans+=a[i];
        return ans;
    }
    public:
    string nextPalin(string s){
        int n = s.size();
        int mid=n/2;
        if(n<=3) return "-1";
        string final,half,result;
        half = s.substr(0,mid);
        result = nextPermutation(half);
        if(result=="-1") return result;
        final+=result;
        if(n&1) final+=s[mid];
        reverse(result.begin(),result.end());
        final+=result;
        return final;
    }
};

} // } Driver Code Ends
```

Sum of two large numbers

Medium Accuracy: 46.83% Submissions: 10213 Points: 4

Given two strings denoting non-negative numbers X and Y. Calculate the sum of X and Y.

Example 1:

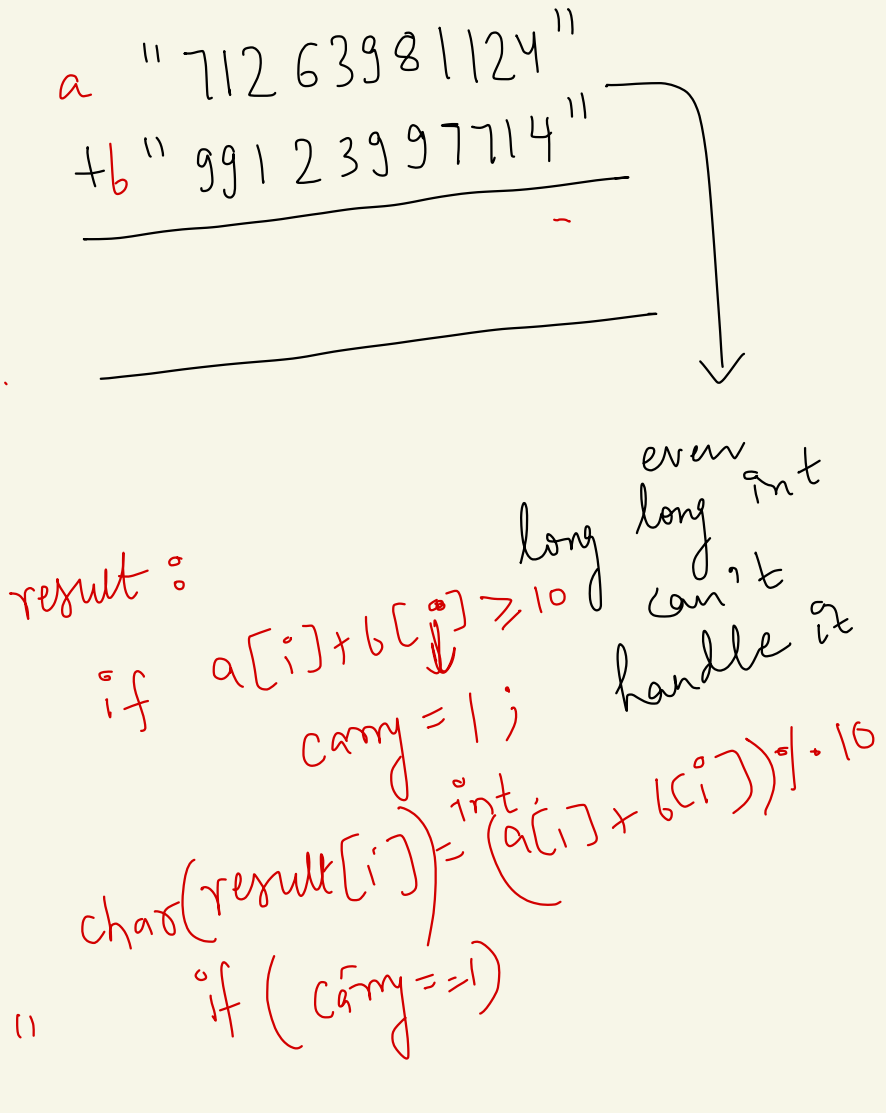
```
Input:
X = "25", Y = "23"
Output:
48
Explanation:
The sum of 25 and 23 is 48.
```

Example 2:

```
Input:
X = "2500", Y = "23"
Output:
2523
Explanation:
The sum of 2500 and 23 is 2523.
```

Your Task:
Your task is to complete the function **findSum()** which takes two strings as inputs and returns the string without leading zeros. You do not need to take any input or print anything.

Expected Time Complexity: O(|X| + |Y|)
Expected Auxiliary Space: O(1)



Sum of two large numbers

Medium Accuracy: 46.84% Submissions: 10214 Points: 4

Given two strings denoting non-negative numbers X and Y. Calculate the sum of X and Y.

Example 1:

```
Input:
X = "25", Y = "23"
Output:
48
Explanation:
The sum of 25 and 23 is 48.
```

Example 2:

```
Input:
X = "2500", Y = "23"
Output:
2523
Explanation:
The sum of 2500 and 23 is 2523.
```

Your Task:
Your task is to complete the function **findSum()** which takes two strings as inputs and returns the string without leading zeros. You do not need to take any input or print anything.

Expected Time Complexity: O(|X| + |Y|)
Expected Auxiliary Space: O(1)

```cpp
// User function template for C++
class Solution {
    private:
    string f(string &s){
        int i=0;
        while(i<s.size() and s[i]=='0') i++;
        if(i==s.size()) return "0";
        return s.substr(i);
    }
    public:
    string findSum(string x, string y)
    {
        int i=x.size()-1, j=y.size()-1,carry=0;
        string ans="";
        while(i>=0 or j>=0)
        {
            int sum = (i>=0?x[i--]-'0':0) + (j>=0? y[j--]-'0':0) + carry;
            carry=sum/10;
            ans+=to_string(sum%10);
        }
        if(carry) ans+=to_string(carry);
        reverse(ans.begin(),ans.end());
        return f(ans);
    }
};
} // } Driver Code Ends
```

to handle 0 wala case

ans = 00 00 | 23

while loop breaks.

Average Time: 20m
Your Time: 24m

Compile & Run

Description | Solution | Discuss (...) | Submissi... | C++ | Autocomplete

Example 1:

```
Input: num1 = "11", num2 = "123"
Output: "134"
```

Example 2:

```
Input: num1 = "456", num2 = "77"
Output: "533"
```

Example 3:

```
Input: num1 = "0", num2 = "0"
Output: "0"
```

Constraints:

- `1 <= num1.length, num2.length <= 10^4`
- `num1` and `num2` consist of only digits.
- `num1` and `num2` don't have any leading zeros except for the zero itself.

```cpp
class Solution {
    private:
    string f(string &s)
    {
        int i=0;
        while(i<s.size() and s[i]=='0') i++;
        if(i==s.size()) return "0";
        return s.substr(i);
    }
    public:
    string addStrings(string x, string y)
    {
        string ans="";
        int i=x.size()-1, j=y.size()-1, carry=0;
        while(i>=0 or j>=0)
        {
            int sum = (i>=0? x[i--]-'0':0) + (j>=0? y[j--]-'0':0) + carry;
            carry= sum/10;
            ans+= to_string(sum%10);
        }
        if(carry) ans+=to_string(carry);
        reverse(ans.begin(),ans.end());
        return f(ans);
    }
};
```

Form a palindrome

Medium Accuracy: 53.0% Submissions: 13990 Points: 4

Given a string, find the minimum number of characters to be inserted to convert it to palindrome.
For Example:
ab: Number of insertions required is 1. **b**ab or aba
aa: Number of insertions required is 0. aa
abcd: Number of insertions required is 3. **dcb**abcd

Example 1:

```
Input: str = "abcd"
Output: 3
Explanation: Inserted character marked
with bold characters in dcbabcd
```

Example 2:

```
Input: str = "aa"
Output: 0
Explanation:"aa" is already a palindrome.
```

Your Task:
You don't need to read input or print anything. Your task is to complete the function **countMin()** which takes the string **str** as inputs and returns the answer.

Expected Time Complexity: $O(N^2)$, N = |str|
Expected Auxiliary Space: $O(N^2)$

```cpp
// } Driver Code Ends
//User function template for C++

class Solution{
   public:
   int LCS(string A,string B)
   {
      int n = A.length(), m = B.length(), maxx=0, dp[n+1][m+1];
      for(int i=0;i<=n;i++)
          for(int j=0;j<=m;j++)
              if(i==0 or j==0) dp[i][j] = 0;
              else if(A[i-1]==B[j-1]){
                  dp[i][j] = 1+dp[i-1][j-1];
                  maxx = max(maxx,dp[i][j]);
              }
              else dp[i][j] = max(dp[i-1][j],dp[i][j-1]);
      return maxx;
   }
    int countMin(string str){
    string B = str;
    reverse(B.begin(),B.end());
    int num = LCS(str,B);
    return str.length()-num;
    }
};
// } Driver Code Ends
```

Count Palindrome Sub-Strings of a String

Medium Accuracy: 54.32% Submissions: 4342 Points: 4

Given a string, the task is to count all palindromic sub-strings present in it. Length of palindrome sub-string must be greater than or equal to 2.

Example

```
Input
N = 5
str = "abaab"
Output
3
Explanation:
All palindrome substring are : "aba" , "aa" , "baab"
```

Example

```
Input
N = 7
str = "abbaeae"
Output
4
Explanation:
All palindrome substring are : "bb" , "abba" ,"aea", "eae"
```

Expected Time Complexity : $O(|S|^2)$
Expected Auxilliary Space : $O(|S|^2)$

```
// } Driver Code Ends


int CountPS(char s[], int n)
{
    bool dp[n][n];
    memset(dp,0,sizeof dp);
    int count=0;
    for(int g=0;g<n;g++){
        for(int i=0,j=g;j<n;i++,j++){
            if(g==0) dp[i][j]=true;
            else if(g==1){
                if(s[i]==s[j]){
                    dp[i][j]=true;
                    count++;
                }
                else dp[i][j]=false;
            }
            else
                if(s[i]==s[j] and dp[i+1][j-1]==true){
                    dp[i][j]=true;
                    count++;
                }
                else dp[i][j]=false;
        }
    }
    return count;
}

```

33. Search in Rotated Sorted Array

Medium 12430 828 Add to List Share

There is an integer array `nums` sorted in ascending order (with **distinct** values).

Prior to being passed to your function, `nums` is **possibly rotated** at an unknown pivot index `k` (`1 <= k < nums.length`) such that the resulting array is `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]` (**0-indexed**). For example, `[0,1,2,4,5,6,7]` might be rotated at pivot index `3` and become `[4,5,6,7,0,1,2]`.

Given the array `nums` **after** the possible rotation and an integer `target`, return *the index of* `target` *if it is in* `nums`, *or* `-1` *if it is not in* `nums`.

You must write an algorithm with `O(log n)` runtime complexity.

Example 1:

```
Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4
```

Example 2:

```
Input: nums = [4,5,6,7,0,1,2], target = 3
Output: -1
```

```
class Solution {
    private:
    int BinarySearch(vector<int>&nums, int l,int h,int key)
    {
        int start=l;
        int end=h;
        int mid = start+(end-start)/2;
        while(start<=end){
            if(nums[mid]==key)return mid;
            else if(nums[mid]<nums[end])
                if(key>nums[mid] and key<=nums[end])
                    start = mid+1;
                else
                    end = mid-1;
            else
                if(key>=nums[start] and key<nums[mid])
                    end = mid-1;
                else
                    start = mid+1;
            mid = start+(end-start)/2;
        }
        return -1;
    }
    public:
    int search(vector<int>& nums, int target)
    {
        return BinarySearch(nums,0,nums.size()-1,target);
    }
};
```

363. Max Sum of Rectangle No Larger Than K

Hard 1826 104 Add to List Share

Given an `m x n` matrix `matrix` and an integer `k`, return *the max sum of a rectangle in the matrix such that its sum is no larger than* `k`.

It is **guaranteed** that there will be a rectangle with a sum no larger than `k`.

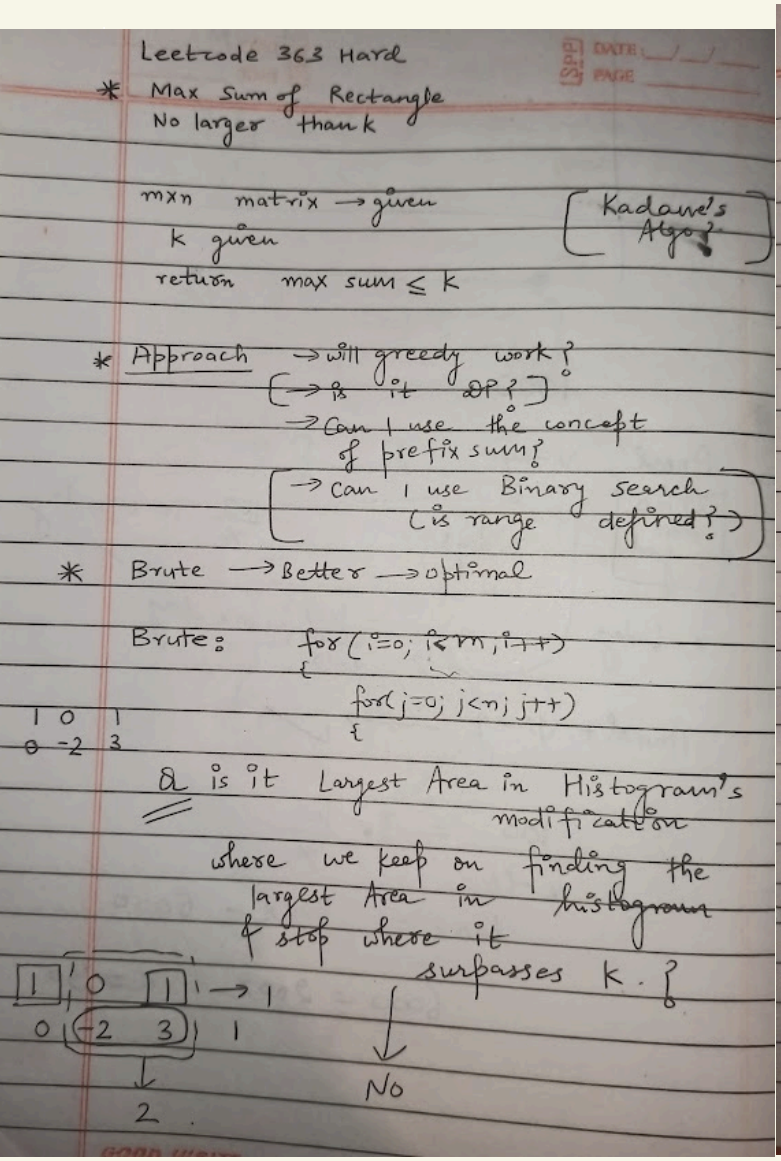
Example 1:

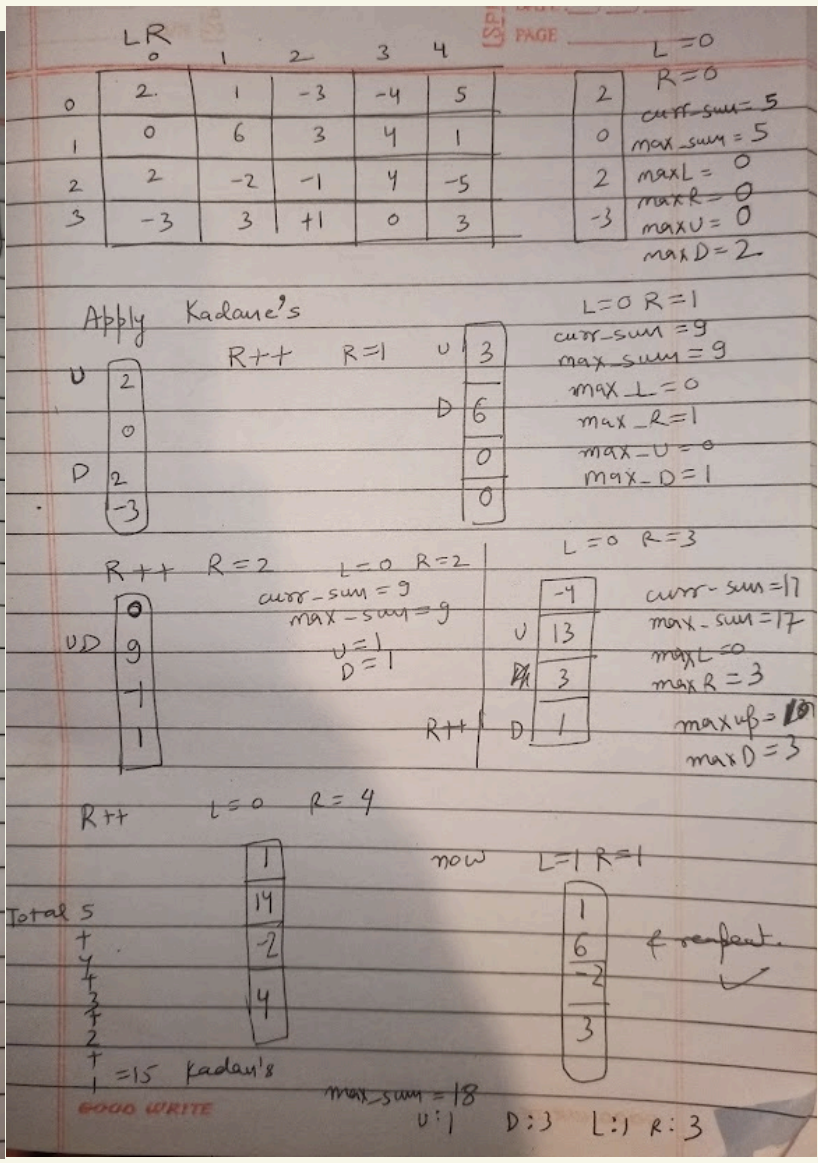
1	0	1
0	-2	3

```cpp
class Solution {
public:
    int maxSumSubmatrix(vector<vector<int>>& matrix, int k) {
        int m = matrix.size(), n = matrix[0].size();
        int res = INT_MIN;
        vector<int> row_sum(m, 0);
        for (int l = 0; l < n; ++l){
            fill(row_sum.begin(), row_sum.end(), 0);// in every iteration, we have to make all 0
            for (int r = l; r < n; ++r){
                // Kadane algorithm for a possible shortcut to next iteration or even an early return
                int cur_sum = 0, best_sum = INT_MIN;
                for (int i = 0; i < m; ++i){
                    row_sum[i] += matrix[i][r];
                    if (cur_sum < 0) cur_sum = row_sum[i];
                    else cur_sum += row_sum[i];
                    best_sum = max(best_sum, cur_sum);
                }
                if (best_sum == k) //return it.
                    return k;
                else if (best_sum < k) then keep on going.
                {
                    res = max(res, best_sum);
                    continue;
                }
                cur_sum = 0;
                set<int> s{0}; //create a set
                for (auto& sum: row_sum)
                {
                    cur_sum += sum;
                    auto it = s.lower_bound(cur_sum - k);
                    if (it != s.end())
                        res = max(res, cur_sum - *it);
                    if (res == k)
                        return k;
                    s.insert(cur_sum);
                }
            }
        }
        return res;
    }
};
```

kadane's algo

binary search





Factorials of large numbers

Medium Accuracy: 51.61% Submissions: 34391 Points: 4

Given an integer N, find its factorial.

Example 1:

```
Input: N = 5
Output: 120
Explanation : 5! = 1*2*3*4*5 = 120
```

Example 2:

```
Input: N = 10
Output: 3628800
Explanation :
10! = 1*2*3*4*5*6*7*8*9*10 = 3628800
```

Your Task:
You don't need to read input or print anything. Complete the function *factorial()* that takes integer N as input parameter and returns a list of integers denoting the digits that make up the factorial of N.

Expected Time Complexity : $O(N^2)$
Expected Auxilliary Space : O(1)

```cpp
// } Driver Code Ends
//User function template for C++

class Solution {
public:
    void multiply(vector<int> &v, int n){
        int carry = 0;
        for(int i=0; i<v.size(); i++){
            int data = v[i] * n + carry;
            v[i] = data % 10;
            carry=data/10;
        }
        while(carry){
            v.push_back(carry % 10);
            carry /= 10;
        }
    }
    vector<int> factorial(int N){
        vector<int> v = {1};
        for(int i=2; i<=N; i++){
            multiply(v,i);
        }

        reverse(v.begin(),v.end());
        return (v);
    }
};

// 5! = 120 n=5
// for loop will run from n=2 to n=5 i.e. 4 t
// and we pass vector v and the number i into
```

Design Circular | Design Snake Ga | Task Scheduler | Rearrange String | Maximum Numb | Moving Average | Problems - Leet | Inbox (34) - mat | Count the Re | Minimum numbe

practice.geeksforgeeks.org/problems/count-the-reversals0401/1#

Practice 831

Problem | Editorial | Submissions | Doubt Support | C++ (g++ 5.4) | Test against custom input

Count the Reversals

Medium Accuracy: 50.95% Submissions: 17146 Points: 4

Given a string **S** consisting of only opening and closing curly brackets **'{'** and **'}'**, find out the minimum number of reversals required to convert the string into a balanced expression.
A reversal means changing **'{'** to **'}'** or vice-versa.

Example 1:

```
Input:
S = "}{{}}{{{"
Output: 3
Explanation: One way to balance is:
"{{{}}{}}". There is no balanced sequence
that can be formed in lesser reversals.
```

â€‹**Example 2:**

```
Input:
S = "{{}{{}{{}}{{"
Output: -1
Explanation: There's no way we can balance
this sequence of braces.
```

Your Task:
You don't need to read input or print anything. Your task is to complete the function **countRev()** which takes the string S as input parameter and returns the minimum number of reversals required to balance the bracket sequence. If balancing is not possible, return -1.

Expected Time Complexity: O(|S|).
Expected Auxiliary Space: O(1).

```cpp
// Contributed By: Pranay Bansal// } Driver Code Ends

int countRev (string s){
    if (s.size()%2)return -1; //odd
    int open=0; int close=0,i=0;
    for (;i<s.size();i++)
        if (s[i]=='{') open++; //if you see open, open++
        else
            if(open==0) close++;
            else open--;
    return (int)(ceil(open/2.0)+ceil(close/2.0));
}
```

Average Time: 15m
Your Time: 2m 12s

Compile & Run | Submit

5. Longest Palindromic Substring

Medium 15791 930 Add to List Share

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"
```

Constraints:

- `1 <= s.length <= 1000`
- `s` consist of only digits and English letters.

```cpp
class Solution {
    public: //this function is checking, is the given range palindrome or not?
    void f(int &l,int &r, string &str,int &n, int &max_len, int &start){
        while(l>=0 and r<str.length() and str[l]==str[r]){ //check if it is in limit and matching.
            int len=r-l+1; //if so, update the length as string contained in range[l,r]
            if(len > max_len){ //update the max_len if you get a bigger len.
                start = l; //if you get so, start becomes the left side l and update your max_len
                max_len = len;
            }
            l--;r++; //increase l and decrease r.
        }
    }
    string longestPalindrome(string str){
        string ans; //where you are storing the result
        int max_len = 0; //here you are storing the length of the longest string.
        int start, end;//start and end.
        int n = str.size();//n=size of the string.
        for(int i=0;i<n;i++){ //for every character, run the for loop
            int l = i, r = i;//odd length palindrome
            f(l,r,str,n,max_len,start); //find length of in the max palindrome in range l,r
            l = i; r = i+1; //even length palindrome
            f(l,r,str,n,max_len,start); //find length of in the max palindrome in range l,r
        }
        return str.substr(start, max_len); //return such string, from the starting index to the length.
    }
};
```

Parsing of simple expressions

For the time being we only consider a simplified problem: we assume that all operators are **binary** (i.e. they take two arguments), and all are **left-associative** (if the priorities are equal, they get executed from left to right). Parentheses are allowed.

We will set up two stacks: one for numbers, and one for operators and parentheses. Initially both stacks are empty. For the second stack we will maintain the condition that all operations are ordered by strict descending priority. If there are parenthesis on the stack, than each block of operators (corresponding to one pair of parenthesis) is ordered, and the entire stack is not necessarily ordered.

We will iterate over the characters of the expression from left to right. If the current character is a digit, then we put the value of this number on the stack. If the current character is an opening parenthesis, then we put it on the stack. If the current character is a closing parenthesis, the we execute all operators on the stack until we reach the opening bracket (in other words we perform all operations inside the parenthesis). Finally if the current character is an operator, then while the top of the stack has an operator with the same or higher priority, we will execute this operation, and put the new operation on the stack.

After we processed the entire string, some operators might still be in the stack, so we execute them.

Here is the implementation of this method for the four operators $+ - * /$:

```cpp
bool delim(char c) {
    return c == ' ';
}

bool is_op(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/';
}

int priority (char op) {
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return -1;
}

void process_op(stack<int>& st, char op) {
    int r = st.top(); st.pop();
    int l = st.top(); st.pop();
    switch (op) {
        case '+': st.push(l + r); break;
        case '-': st.push(l - r); break;
        case '*': st.push(l * r); break;
        case '/': st.push(l / r); break;
    }
}
```

```cpp
int evaluate(string& s) {
    stack<int> st;
    stack<char> op;
    for (int i = 0; i < (int)s.size(); i++) {
        if (delim(s[i]))
            continue;

        if (s[i] == '(') {
            op.push('(');
        } else if (s[i] == ')') {
            while (op.top() != '(') {
                process_op(st, op.top());
                op.pop();
            }
            op.pop();
        } else if (is_op(s[i])) {
            char cur_op = s[i];
            while (!op.empty() && priority(op.top()) >= priority(cur_op)) {
                process_op(st, op.top());
                op.pop();
            }
            op.push(cur_op);
        } else {
            int number = 0;
            while (i < (int)s.size() && isalnum(s[i]))
                number = number * 10 + s[i++] - '0';
            --i;
            st.push(number);
        }
    }
```

```
    while (!op.empty()) {
        process_op(st, op.top());
        op.pop();
    }
    return st.top();
}
```

Thus we learned how to calculate the value of an expression in $O(n)$, at the same time we implicitly used the reverse Polish notation. By slightly modifying the above implementation it is also possible to obtain the expression in reverse Polish notation in an explicit form.

Unary operators

Now suppose that the expression also contains **unary** operators (operators that take one argument). The unary plus and unary minus are common examples of such operators.

One of the differences in this case, is that we need to determine whether the current operator is a unary or a binary one.

You can notice, that before an unary operator, there always is another operator or an opening parenthesis, or nothing at all (if it is at the very beginning of the expression). On the contrary before a binary operator there will always be an operand (number) or a closing parenthesis. Thus it is easy to flag whether the next operator can be unary or not.

Additionally we need to execute a unary and a binary operator differently. And we need to chose the priority of a unary operator higher than all of the binary operators.

In addition it should be noted, that some unary operators (e.g. unary plus and unary minus) are actually **right-associative**.

```
bool delim(char c) {
    return c == ' ';
}

bool is_op(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/';
}

bool is_unary(char c) {
    return c == '+' || c=='-';
}

int priority (char op) {
    if (op < 0) // unary operator
        return 3;
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return -1;
}

void process_op(stack<int>& st, char op) {
    if (op < 0) {
        int l = st.top(); st.pop();
        switch (-op) {
            case '+': st.push(l); break;
            case '-': st.push(-l); break;
        }
    } else {
        int r = st.top(); st.pop();
        int l = st.top(); st.pop();
        switch (op) {
            case '+': st.push(l + r); break;
            case '-': st.push(l - r); break;
            case '*': st.push(l * r); break;
            case '/': st.push(l / r); break;
        }
    }
}

int evaluate(string& s) {
    stack<int> st;
    stack<char> op;
    bool may_be_unary = true;
    for (int i = 0; i < (int)s.size(); i++) {
        if (delim(s[i]))
            continue;

        if (s[i] == '(') {
            op.push('(');
            may_be_unary = true;
        } else if (s[i] == ')') {
            while (op.top() != '(') {
                process_op(st, op.top());
                op.pop();
            }
            op.pop();
            may_be_unary = false;
        } else if (is_op(s[i])) {
            char cur_op = s[i];
            if (may_be_unary && is_unary(cur_op))
                cur_op = -cur_op;
            while (!op.empty() && (
                    (cur_op >= 0 && priority(op.top()) >= priority(cur_op)) ||
                    (cur_op < 0 && priority(op.top()) > priority(cur_op))
                )) {
                process_op(st, op.top());
                op.pop();
            }
            op.push(cur_op);
            may_be_unary = true;
        } else {
            int number = 0;
            while (i < (int)s.size() && isalnum(s[i]))
                number = number * 10 + s[i++] - '0';
            --i;
            st.push(number);
            may_be_unary = false;
        }
    }

    while (!op.empty()) {
        process_op(st, op.top());
        op.pop();
    }
    return st.top();
}
```

Right-associativity

Right-associative means, that whenever the priorities are equal, the operators must be evaluated from right to left.

As noted above, unary operators are usually right-associative. Another example for an right-associative operator is the exponentiation operator ($a \wedge b \wedge c$ is usually perceived as a^{b^c} and not as $(a^b)^c$).

What difference do we need to make in order to correctly handle right-associative operators? It turns out that the changes are very minimal. The only difference will be, if the priorities are equal we will postpone the execution of the right-associative operation.

The only line that needs to be replaced is

```
while (!op.empty() && priority(op.top()) >= priority(cur_op))
```

with

```
while (!op.empty() && (
        (left_assoc(cur_op) && priority(op.top()) >= priority(cur_op)) ||
        (!left_assoc(cur_op) && priority(op.top()) > priority(cur_op))
    ))
```

where `left_assoc` is a function that decides if an operator is left_associative or not.

Here is an implementation for the binary operators $+ - * /$ and the unary operators $+$ and $-$.

Number following a pattern

Number following a pattern

Medium Accuracy: 55.05% Submissions: 4391 Points: 4

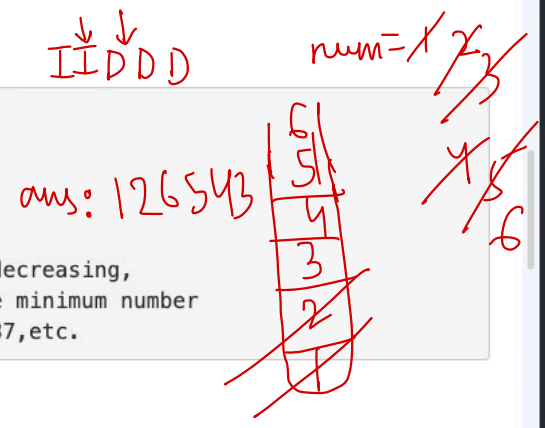
Given a pattern containing only I's and D's. I for increasing and D for decreasing.
Devise an algorithm to print the minimum number following that pattern.
Digits from 1-9 and digits can't repeat.

Example 1:

Input:
D
Output:
21
Explanation:
D is meant for decreasing,
So we choose the minimum number
among 21,31,54,87,etc.

Example 2:

Input:
IIDDD
Output:

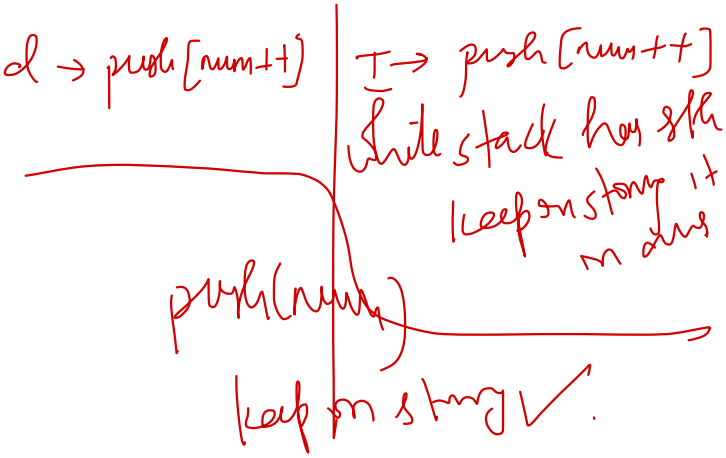



```
class Solution{
public:
    string printMinNumberForPattern(string s){
        string ans;
        stack<int> st;
        int num = 1;
        for(auto c:s){
            if(c == 'D'){
                st.push(num);
                num++;
            }
            else
            {
                st.push(num);
                num++;
                while(!st.empty())
                {
                    ans += to_string(st.top());
                    st.pop();
                }
            }
        st.push(num);
        while(!st.empty()){
            ans += to_string(st.top());
            st.pop();
        }
        return ans;
    }
};
```

We are taking a pattern here and the pattern contains only I and D's and now our task is to print the output as numbers.

If the input is IIDDD this means first two numbers are increasing and then next three numbers are decreasing, now one challenge is that we need to use minimum numbers, so once you see I you can keep on printing 1 and then if you see D then you have to push it into the stack. The idea of stack is that you will use minimum numbers and once you reach the end of the input string then you start popping the elements from the stack and start pushing it in the string end.

IIDDD
1
12
Push 3 in stack
push 4 in stack
Push 5 in stack
Reached end?
pop string top and Add 5 to string—>125
Pop string top and add 4 onto the string end—> 1254
pop string top and add 3 onto the string end—> 12543



Now it was the intuition, if the input begins with D push 1 followed by 2 in stack and then keep on adding numbers if you see other numbers in the input, then again do the same thing that you have to push the numbers

We are simply assigning num=1 and then doing case analysis if character is D then we do something else and if it is the I then you again do the same thing that you push the number in the stack and then you simply increment number.

Permutation of a given string

Permutations of a given string

Medium Accuracy: 49.85% Submissions: 36967 Points: 4

Given a string S. The task is to print all permutations of a given string in lexicographically sorted order.

Example 1:

```
Input: ABC
Output:
ABC ACB BAC BCA CAB CBA
Explanation:
Given string ABC has permutations in 6
forms as ABC, ACB, BAC, BCA, CAB and CBA .
```

Example 2:

```
Input: ABSG
Output:
ABGS ABSG AGBS AGSB ASBG ASGB BAGS
BASG BGAS BGSA BSAG BSGA GABS GASB
GBAS GBSA GSAB GSBA SABG SAGB SBAG
SBGA SGAB SGBA
Explanation:
Given string ABSG has 24 permutations.
```

```
// } Driver Code Ends
class Solution
{
    public:
        void f(string ip, string op, vector<string>&ans){
        int n = ip.length();
        if(n == 0){
            ans.push_back(op);
            return;
        }
        for(int i = 0;i<n;i++){
            string op1 = op;
            string ip1 = ip;
            op1.push_back(ip[i]);
            ip1.erase(ip1.begin()+i);
            f(ip1,op1,ans);
        }
        return;

    }
vector<string>find_permutation(string S)
{
    // Code here there
    vector<string> ans;
    string ip = S;
    string op = "";
    f(ip,op,ans);
    sort(ans.begin(), ans.end());
    return ans;
}
};
```

Create a vector of string type ans and then create two strings ip=S and a null string as op
Then call function f(ip, op, ans)
This function will do something.
Then we will sort the vector ans.
Now let us see what this function does
We are passing two strings ip and op into the function and passing the vector.
Then we are storing ip.length() in n. If n is 0 then we will simply push op into answer this means we have passed the ans vector as reference, so we need not worry about anything simply return, But If n>0 then in that case, we will run a for loop till the length of ip string.
Then we have two strings op1 and ip1 and we are copying op and ip into those and then we are pushing ip[I] in op[I] and then we are erasing ip1 from start to ith index and then we are calling f(ip1, op1,ans)

Longest palindrome in a string

Longest Palindrome in a String

Medium Accuracy: 49.2% Submissions: 48532 Points: 4

Given a string S, find the longest palindromic substring in S. **Substring of string S:** S[i . . . . j] where 0 ≤ i ≤ j < len(S). **Palindrome string:** A string which reads the same backwards. More formally, S is palindrome if reverse(S) = S. **Incase of conflict**, return the substring which occurs first (with the least starting index).

Example 1:

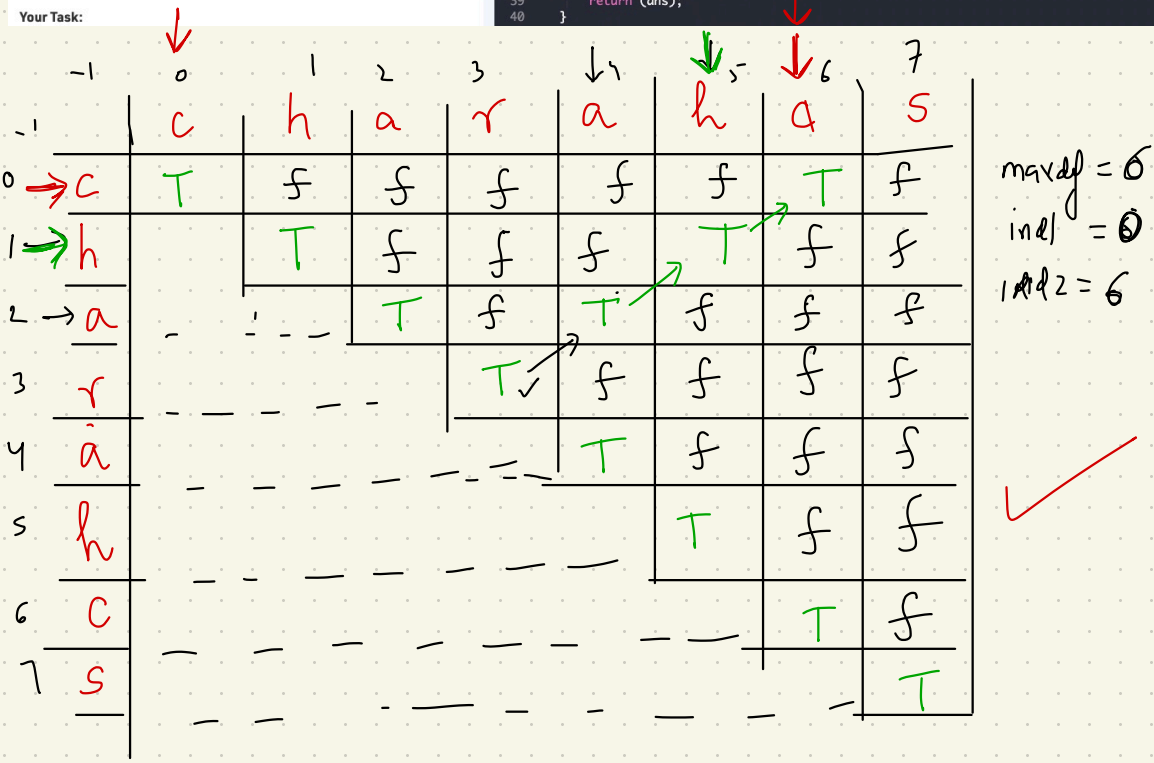
```
Input:
S = "aaaabbaa"
Output: aabbaa
Explanation: The longest Palindromic
substring is "aabbaa".
```

Example 2:

```
Input:
S = "abc"
Output: a
Explanation: "a", "b" and "c" are the
longest palindromes with same length.
The result is the one with the least
starting index.
```

Your Task:

```cpp
class Solution {
  public:
    string longestPalin (string S) {
        int n = S.length();
        vector<vector<bool>> dp(n,vector<bool>(n,false));
        for(int gap = 0; gap<n; gap++){
            for(int i=0,j=gap; j<n; i++,j++){
                if(gap == 0) dp[i][j] = true; //for 0 length palindrome
                else if(gap == 1){ //for single length palindrome
                    if(S[i] == S[j]) dp[i][j] = true; //if the char are same, then true
                    else dp[i][j] = false; //else false
                }
                else{ //for more than two length palindrome
                    if(S[i] == S[j]) dp[i][j] = dp[i+1][j-1]; //next row and prev col
                    else dp[i][j] = false;
                }
            }
        }

        int ind1 = -1, ind2 = -1, max_diff = -1;
        for(int i=0; i<n; i++)
            for(int j=0; j<n; j++)
                if(dp[i][j] and j-i > max_diff){
                    max_diff = j-i;
                    ind1 = i;
                    ind2 = j;
                }
        string ans{};
        for(int i = ind1; i <= ind2; i++){
            ans += S[i];
        }

        return (ans);
    }
```



Substrings of length k with k-1 elements

992. Subarrays with K Different Integers

Hard 2699 39 Add to List Share

Given an integer array `nums` and an integer `k`, return *the number of **good subarrays** of* `nums`.

A **good array** is an array where the number of different integers in that array is exactly `k`.

- For example, `[1,2,3,1,2]` has `3` different integers: `1`, `2`, and `3`.

A **subarray** is a **contiguous** part of an array.

Example 1:

```
Input: nums = [1,2,1,2,3], k = 2
Output: 7
Explanation: Subarrays formed with exactly 2 different integers: [1,2], [2,1], [1,2], [2,3], [1,2,1], [2,1,2], [1,2,1,2]
```

```cpp
class Solution {
public:

    int ok(vector<int>& A , int K){
        if(K==0) return 0;
        int n=A.size(),total=0,diff=0,j=0;
        vector<int>cnt(20002);
        for(int i=0;i<n;i++){
            if(cnt[A[i]]==0)diff++;
            cnt[A[i]]++;
            if(diff<=K) total+=(i-j+1);
            else{
                while(j<n and j<=i and diff>K){
                    cnt[A[j]]--;
                    if(cnt[A[j]]==0) diff--;
                    j++;
                }
                total+=(i-j+1);
            }
        }
        return total;
    }
    int subarraysWithKDistinct(vector<int>& A, int K) {
        return ok(A,K)-ok(A,K-1);
    }
};
```

Repeated string match

686. Repeated String Match

Medium 1313 889 Add to List Share

Given two strings `a` and `b`, return *the minimum number of times you should repeat string* `a` *so that string* `b` *is a substring of it*. If it is impossible for `b` to be a substring of `a` after repeating it, return `-1`.

Notice: string `"abc"` repeated 0 times is `""`, repeated 1 time is `"abc"` and repeated 2 times is `"abcabc"`.

Example 1:

```
Input: a = "abcd", b = "cdabcdab"
Output: 3
Explanation: We return 3 because by repeating a three times "abcdabcdabcd", b is a substring of it.
```

Example 2:

```
Input: a = "a", b = "aa"
Output: 2
```

Constraints:

```cpp
class Solution {
public:
    int KMP(const string& a, const string& b) {
        int m = a.size(); //size of string 1
        int n = b.size(); //size of string 2
        int k = 1 + int(ceil(double(n-1) / double(m))); //k
        string s;
        for(int i=0; i<k; i++) s += a; //add string a in itself k times
        int kmp[n]; //kmp array
        memset(kmp, 0, sizeof(kmp)); //set it 0.
        int i = 1, j = 0; //keep i at index 1 and j at index 0
        while(i < n) { //go till string2 length
            if(b[i] == b[j]) //if you see repeated instance of a char in b
                kmp[i++] = j++ +1; //store the next index of it in kmp[i] and increment i,j after
            else  //if they are not matching, then
                if(!j) i++; //if j is at start, increment i
                else j = kmp[j-1]; //if not, j will be whatever previously was it kmp[j-1]
        }
        int l=s.size(); //store size of string s in l
        i = 0, j = 0; //assign i,j as 0
        while(i+n<=j+l and j<n) //if b's length + i has not exceeded s'th length +j
            if(s[i]==b[j]) i++,j++; //if they are same, move both pointers one step ahead
            else
                if(!j) i++; //if j is at 0, just increase i
                else j = kmp[j-1]; //otherwise whatever is at kmp[j-1] store it in j
        if(j < n) return -1; //if j has not yet exceeded n, return -1
        return int(ceil(double(i) / double(m))); //return ceil(i/m) as we need these many repetetions
    }
    int repeatedStringMatch(string a, string b) {
        return KMP(a, b);
    }

};
```

Rolling hash method

686. Repeated String Match

Medium 1313 889 Add to List Share

Given two strings `a` and `b`, return *the minimum number of times you should repeat string* `a` *so that string* `b` *is a substring of it*. If it is impossible for `b` to be a substring of `a` after repeating it, return `-1`.

Notice: string `"abc"` repeated 0 times is `""`, repeated 1 time is `"abc"` and repeated 2 times is `"abcabc"`.

Example 1:

```
Input: a = "abcd", b = "cdabcdab"
Output: 3
Explanation: We return 3 because by repeating a three times "abcdabcdabcd", b is a substring of it.
```

Example 2:

```
Input: a = "a", b = "aa"
Output: 2
```

```cpp
class Solution {
public:
    int RollingHash(const string& a, const string& b){
        int m = a.size(),n = b.size(); //store the size of both strings
        int k = 1+int(ceil(double(n-1)/double(m))); //number of times you will copy a in s
        string s;
        for(int i=0;i<k;i++) s+=a; //concatenate a in s k times
        long M=1e9+7,hash=0,target=0,power=1;
        for(int i=0;i<n;i++){ //for every character in b
            hash=(hash*26+(long)(s[i]-'a'))%M; //calculate hash of that character modulo M
            target=(target*26+(long)(b[i]-'a'))%M; //calculate hash of char in s store in target
            if(i) power=(power*26) % M; // keep on increasing power modulo M
        }
        if(hash==target and s.substr(0,n)==b) return int(ceil(double(n)/double(m)));
        //return the number of times, if both hashes and target match,
        //( if a and b are same then only this condition will return)
        for(int i=1;i<m;i++){ //for every character in a from index 1
            hash=((hash-(long)(s[i-1]-'a')*power)*26 + (long)(s[i+n-1]-'a'))%M; //rolling hash method
            if(hash<0) hash+=M; //if negetive hash, roll the hash
            if(hash==target and s.substr(i,n)==b) return int(ceil(double(i+n)/double(m)));
            //return the number of times, if both hashes and target match and b is present in a
        }
        return -1; //return -1 if you can never find b in a :)
    }

    int repeatedStringMatch(string a, string b){
        return RollingHash(a, b);
    }
};
```

Binary Search, Stack, Queue, PQ, Segment Tree

35

Square root of a number ✓ Binary S.

minimum platform ✓ Sorting

minimum sum ✓ Sorting.

modular exponentiation of large numbers. ✓ Binary search.

Distribute n candies to k people ✓ → k+ logn time.

Coin piles ✓ → sorting prefix array

Painter's partition Problem ✓ → binary search, dp

Capacity to ship packages within D days ✓ → binary search.

Get minimum element from stack ✓ O(1) Space → hard → stack.

LRU cache → Stack

Celebrity problem ✓ → Stack

Max Rectangle ✓ → Stack

Next Greater Element ✓ → O(n) time → geeksforgeeks

Circular tour ✓ → Stack

Maximum rectangular area in a histogram ✓ → Stack

→ NLE & NSE combo using stack.

Stock span problem ✓ → dp.

max of min of every window size ✓

Rotten oranges → binary search

Steps by knight ✓ → bfs, queue

Longest sub-array with sum / by K → hashmap of remainder pefixsum

Smallest distinct window → hashmap & sliding window

first non-repeating character in a stream

Top K numbers in a stream

Geeky buildings

Count the reversals

Remove K digits

huffman encoding

Clone a stack w/o using extra space

Check mirror in n-ary tree

IPL 2021 match day 2.

Valid Expressions

Smallest number on left → greedy + Binary search.

Divide chocolate

split array largest sum

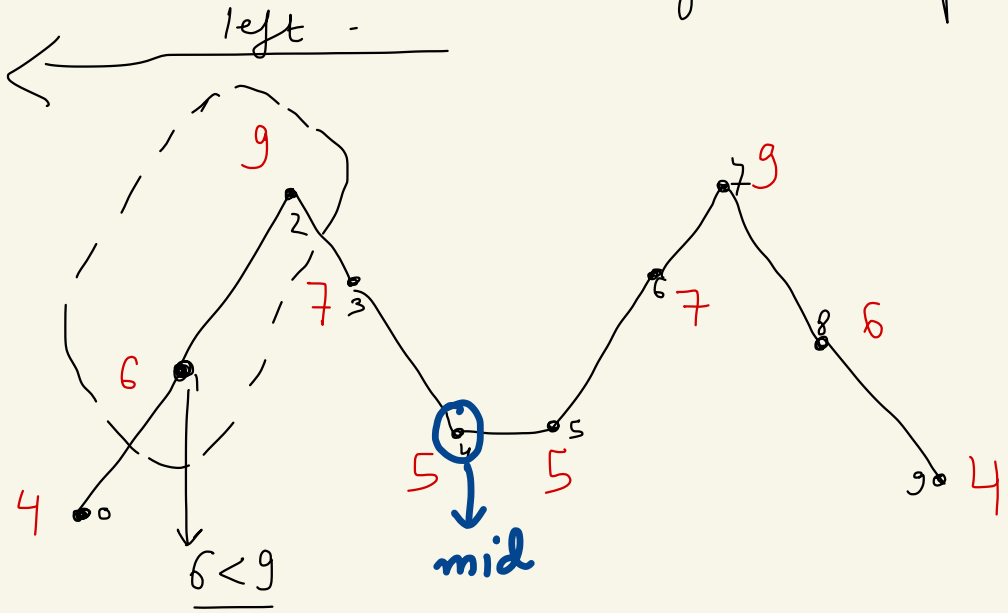
n = "121" if no of digits <= 3 → return -1
as next greater pal with same digits is not possible.

n = "469755794"

since the number can be so large it can't fit in long long int

we need to do some sort of string manipulation

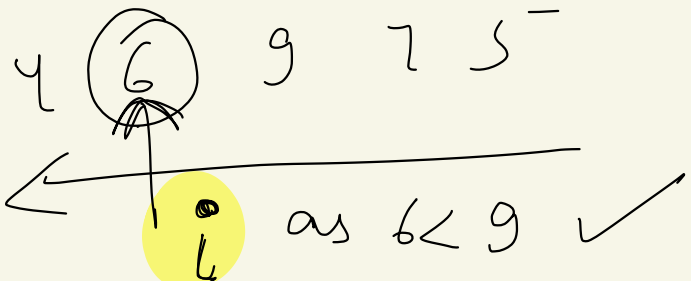
Idea.



1) find mid → $\frac{n}{2} - 1 = \frac{10}{2} - 1 = 4$

2) start traversing from mid → left side s.t.

we saw



3) Now start traversing from 9 to 5.
i.e i+1 to mid

search smallest digit greater than 6.
i.e next greater element on the right

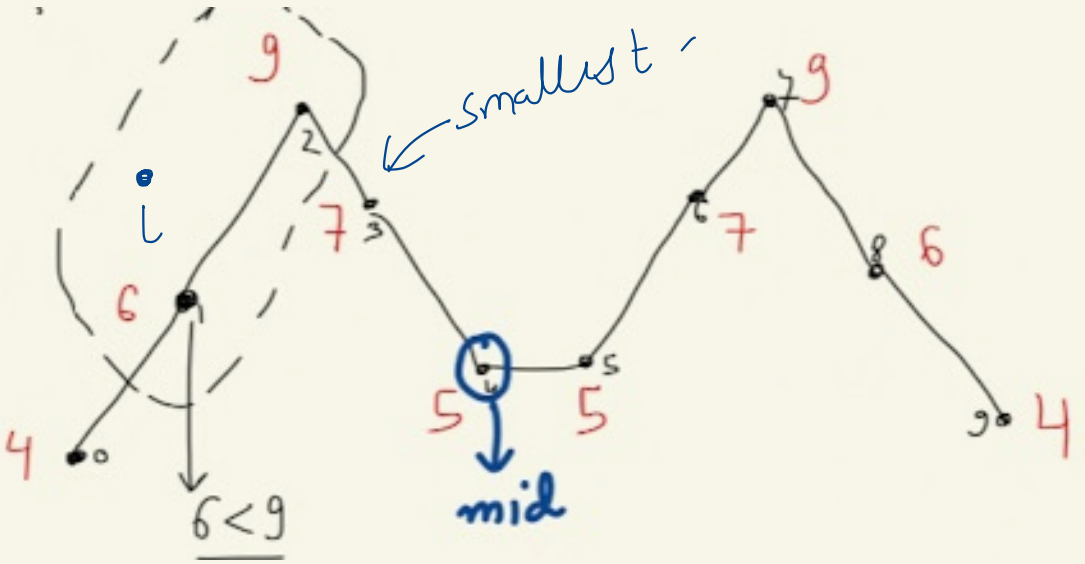
in range.

smallest

9 7 5

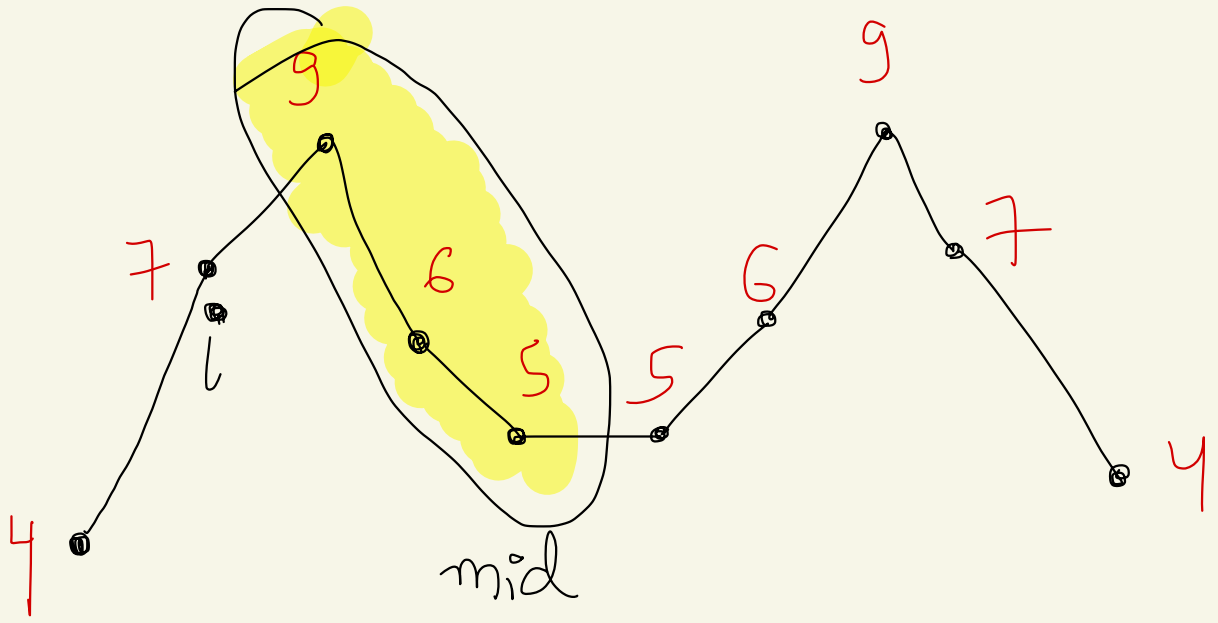
we see it's 7. ✓ as 7 > 6

if no such element is found
return -1 ✓



swap(i, smallest).

swap(n-1-i, n-1-smallest).



reverse elements in range (i+1, mid)

if n is even also we need to reverse.

range(mid+1, n-i-2)

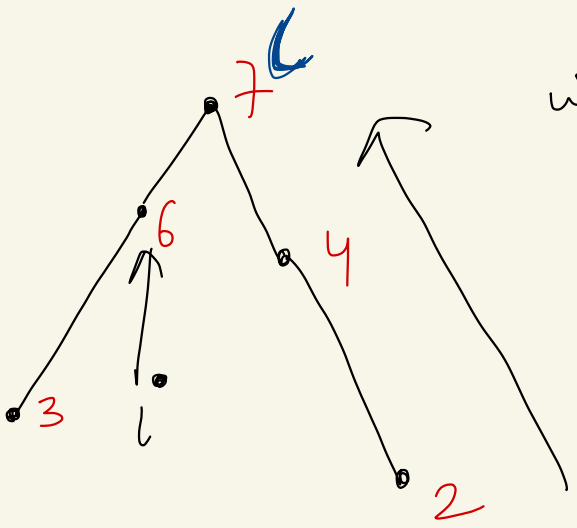
else if n is odd.

reverse

range(mid+2, n-i-2)

This Question is an extension of next permutation

Suppose you have a number



what will be the next permutation ?

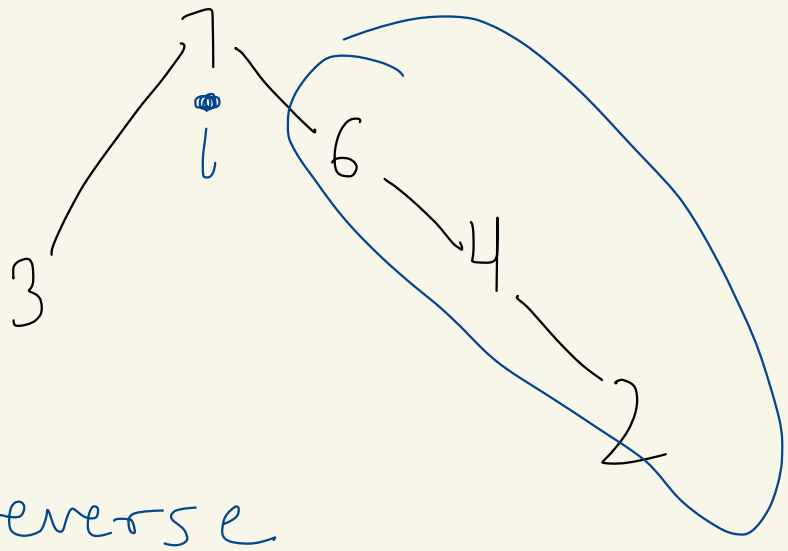
traverse from n-2 to 0 & find the first occurence where $a[i] < a[i+1]$

$4 < 2$ X

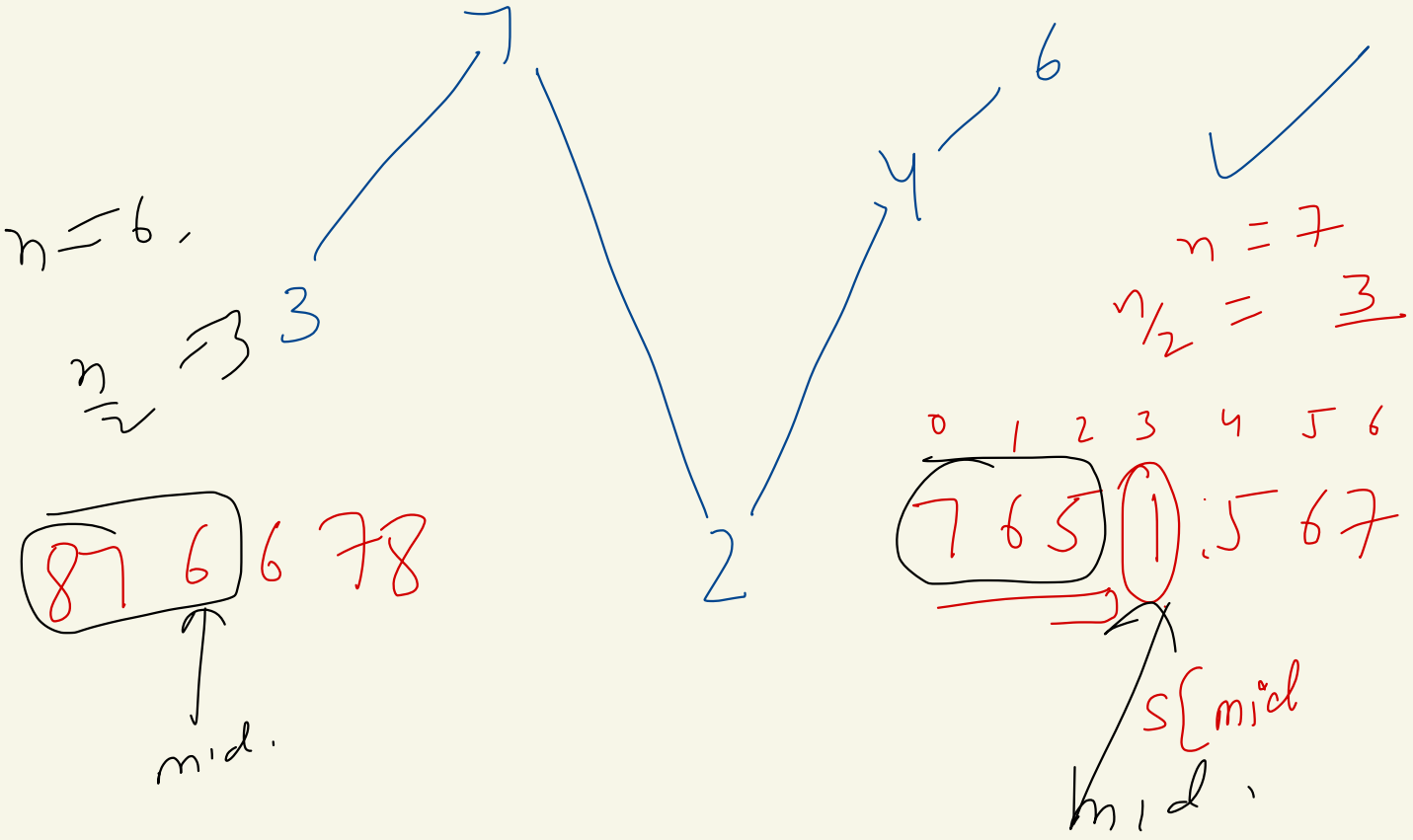
$7 < 4$ X

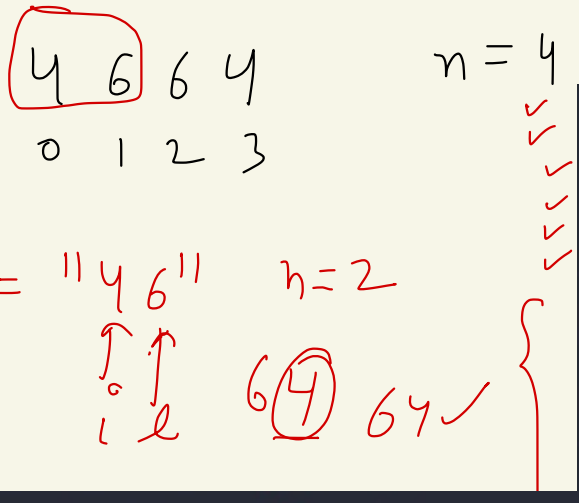
$6 < 7$ ✓

now swap $a[i]$ with the first occurence from n-1 to i+1 where $a[i] < a[k]$ ✓



reverse






```
string nextPalin(string s)
{
    int n = s.size();
    int mid=n/2;
    if(n<=3) return "-1";
    string final,half,result;
    if(!n%2) //n is even
    {
        half = s.substr(0,mid);
        result = nextPermutation(half);
        if(result=="-1") return result;
        final = result;
        reverse(result.begin(),result.end());
        final+=result;
    }
    else //n is odd
    {
        half = s.substr(0,mid);
        result = nextPermutation(half);
        if(result=="-1") return result;
        final = result;
        final = result + s[mid];
        reverse(result.begin(),result.end());
        final+=result;

    }
    return final;
}
```

```
string nextPermutation(string &a)
{
    int n=a.size(),l,i;
    for(i=n-2;i>=0;i--) if(a[i]<a[i+1]) break;
    if(i<0) return "-1";
    else
    {
        for(l=n-1;l>i;l--)if(a[i]<a[l]) break;
        swap(a[i],a[l]);
        reverse(a.begin()+i+1,a.end());
    }
    string ans="";
    for(int i=0;i<a.size();i++) ans+=a[i];
    return ans;
}
```

Next higher palindromic number using the same set of digits

Medium Accuracy: 44.28% Submissions: 8877 Points: 4

Given a palindromic number **N** in the form of string. The task is to find the smallest palindromic number greater than **N** using the same set of digits as in **N**.

Example 1:

```
Input:
N = "35453"
Output:
53435
Explanation: Next higher palindromic
number is 53435.
```

Example 2:

```
Input: N = "33"
Output: -1
Explanation: Next higher palindromic number
does not exist.
```

Your Task:
You don't need to read input or print anything. Your task is to complete the function **nextPalin()** which takes the string **N** as input parameters and returns the answer, else if no such number exists returns "-1".

Expected Time Complexity: O(|N|log|N|)
Expected Auxiliary Space: O(1)

```cpp
// } Driver Code Ends
//User function template for C++
class Solution{
    string nextPermutation(string &a){
        int n=a.size(),l,i;
        for(i=n-2;i>=0;i--) if(a[i]<a[i+1]) break;
        if(i<0) return "-1";
        else{
            for(l=n-1;l>i;l--)if(a[i]<a[l]) break;
            swap(a[i],a[l]);
            reverse(a.begin()+i+1,a.end());
        }
        string ans="";
        for(int i=0;i<a.size();i++) ans+=a[i];
        return ans;
    }
    public:
    string nextPalin(string s){
        int n = s.size();
        int mid=n/2;
        if(n<=3) return "-1";
        string final,half,result;
        half = s.substr(0,mid);
        result = nextPermutation(half);
        if(result=="-1") return result;
        final+=result;
        if(n&1) final+=s[mid];
        reverse(result.begin(),result.end());
        final+=result;
        return final;
    }
};

} // } Driver Code Ends
```

Sum of two large numbers

Medium Accuracy: 46.83% Submissions: 10213 Points: 4

Given two strings denoting non-negative numbers X and Y. Calculate the sum of X and Y.

Example 1:

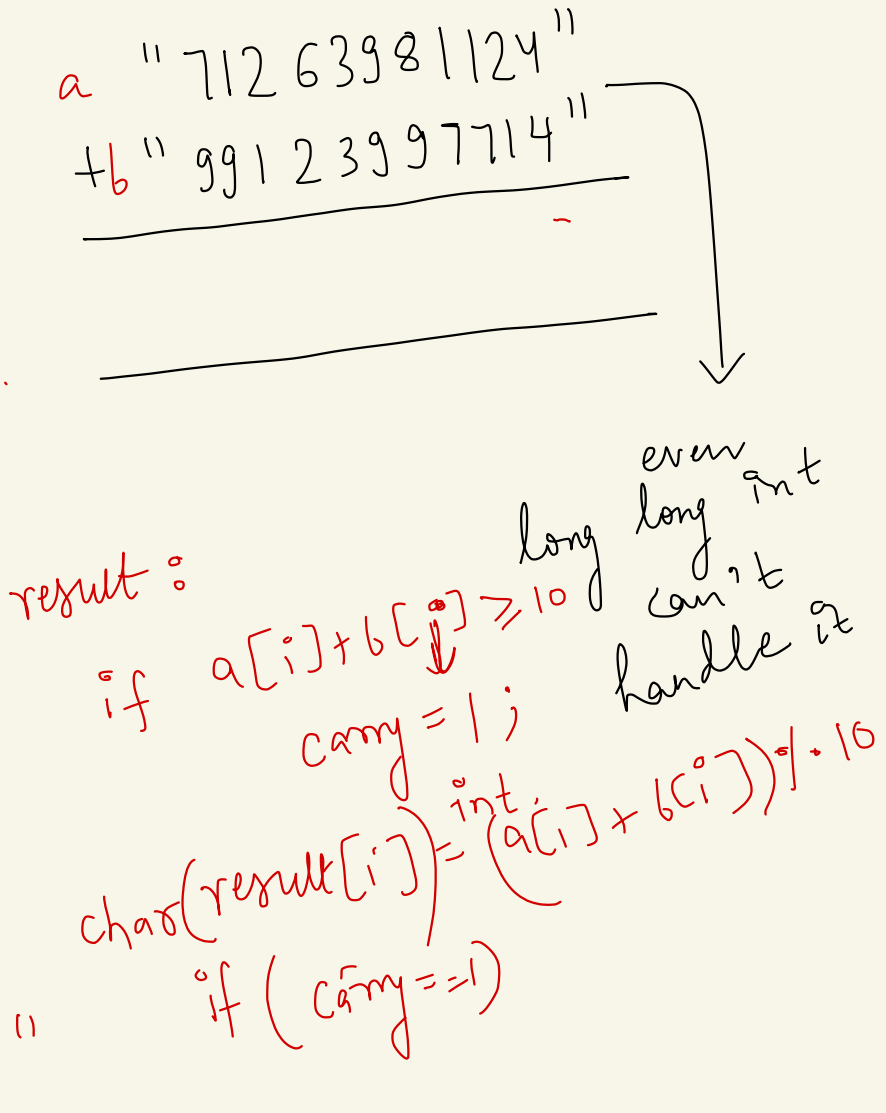
```
Input:
X = "25", Y = "23"
Output:
48
Explanation:
The sum of 25 and 23 is 48.
```

Example 2:

```
Input:
X = "2500", Y = "23"
Output:
2523
Explanation:
The sum of 2500 and 23 is 2523.
```

Your Task:
Your task is to complete the function **findSum()** which takes two strings as inputs and returns the string without leading zeros. You do not need to take any input or print anything.

Expected Time Complexity: O(|X| + |Y|)
Expected Auxiliary Space: O(1)



Sum of two large numbers

Medium Accuracy: 46.84% Submissions: 10214 Points: 4

Given two strings denoting non-negative numbers X and Y. Calculate the sum of X and Y.

Example 1:

```
Input:
X = "25", Y = "23"
Output:
48
Explanation:
The sum of 25 and 23 is 48.
```

Example 2:

```
Input:
X = "2500", Y = "23"
Output:
2523
Explanation:
The sum of 2500 and 23 is 2523.
```

Your Task:
Your task is to complete the function **findSum()** which takes two strings as inputs and returns the string without leading zeros. You do not need to take any input or print anything.

Expected Time Complexity: O(|X| + |Y|)
Expected Auxiliary Space: O(1)

```cpp
// User function template for C++
class Solution {
    private:
    string f(string &s){
        int i=0;
        while(i<s.size() and s[i]=='0') i++;
        if(i==s.size()) return "0";
        return s.substr(i);
    }
    public:
    string findSum(string x, string y)
    {
        int i=x.size()-1, j=y.size()-1,carry=0;
        string ans="";
        while(i>=0 or j>=0)
        {
            int sum = (i>=0?x[i--]-'0':0) + (j>=0? y[j--]-'0':0) + carry;
            carry=sum/10;
            ans+=to_string(sum%10);
        }
        if(carry) ans+=to_string(carry);
        reverse(ans.begin(),ans.end());
        return f(ans);
    }
};
} // } Driver Code Ends
```

to handle 0 wala case

ans = 00 00 | 23

while loop breaks.

Average Time: 20m
Your Time: 24m

Compile & Run

Description | Solution | Discuss (...) | Submissi... | C++ | Autocomplete

Example 1:

```
Input: num1 = "11", num2 = "123"
Output: "134"
```

Example 2:

```
Input: num1 = "456", num2 = "77"
Output: "533"
```

Example 3:

```
Input: num1 = "0", num2 = "0"
Output: "0"
```

Constraints:

- `1 <= num1.length, num2.length <= 10^4`
- `num1` and `num2` consist of only digits.
- `num1` and `num2` don't have any leading zeros except for the zero itself.

```cpp
class Solution {
    private:
    string f(string &s)
    {
        int i=0;
        while(i<s.size() and s[i]=='0') i++;
        if(i==s.size()) return "0";
        return s.substr(i);
    }
    public:
    string addStrings(string x, string y)
    {
        string ans="";
        int i=x.size()-1, j=y.size()-1, carry=0;
        while(i>=0 or j>=0)
        {
            int sum = (i>=0? x[i--]-'0':0) + (j>=0? y[j--]-'0':0) + carry;
            carry= sum/10;
            ans+= to_string(sum%10);
        }
        if(carry) ans+=to_string(carry);
        reverse(ans.begin(),ans.end());
        return f(ans);
    }
};
```

Form a palindrome

Medium Accuracy: 53.0% Submissions: 13990 Points: 4

Given a string, find the minimum number of characters to be inserted to convert it to palindrome.
For Example:
ab: Number of insertions required is 1. **b**ab or aba
aa: Number of insertions required is 0. aa
abcd: Number of insertions required is 3. **dcb**abcd

Example 1:

```
Input: str = "abcd"
Output: 3
Explanation: Inserted character marked
with bold characters in dcbabcd
```

Example 2:

```
Input: str = "aa"
Output: 0
Explanation:"aa" is already a palindrome.
```

Your Task:
You don't need to read input or print anything. Your task is to complete the function **countMin()** which takes the string **str** as inputs and returns the answer.

Expected Time Complexity: $O(N^2)$, N = |str|
Expected Auxiliary Space: $O(N^2)$

```cpp
// } Driver Code Ends
//User function template for C++

class Solution{
   public:
   int LCS(string A,string B)
   {
      int n = A.length(), m = B.length(), maxx=0, dp[n+1][m+1];
      for(int i=0;i<=n;i++)
          for(int j=0;j<=m;j++)
              if(i==0 or j==0) dp[i][j] = 0;
              else if(A[i-1]==B[j-1]){
                  dp[i][j] = 1+dp[i-1][j-1];
                  maxx = max(maxx,dp[i][j]);
              }
              else dp[i][j] = max(dp[i-1][j],dp[i][j-1]);
      return maxx;
   }
    int countMin(string str){
    string B = str;
    reverse(B.begin(),B.end());
    int num = LCS(str,B);
    return str.length()-num;
    }
};
// } Driver Code Ends
```

Count Palindrome Sub-Strings of a String

Medium Accuracy: 54.32% Submissions: 4342 Points: 4

Given a string, the task is to count all palindromic sub-strings present in it. Length of palindrome sub-string must be greater than or equal to 2.

Example

```
Input
N = 5
str = "abaab"
Output
3
Explanation:
All palindrome substring are : "aba" , "aa" , "baab"
```

Example

```
Input
N = 7
str = "abbaeae"
Output
4
Explanation:
All palindrome substring are : "bb" , "abba" ,"aea", "eae"
```

Expected Time Complexity : $O(|S|^2)$
Expected Auxilliary Space : $O(|S|^2)$

```
// } Driver Code Ends


int CountPS(char s[], int n)
{
    bool dp[n][n];
    memset(dp,0,sizeof dp);
    int count=0;
    for(int g=0;g<n;g++){
        for(int i=0,j=g;j<n;i++,j++){
            if(g==0) dp[i][j]=true;
            else if(g==1){
                if(s[i]==s[j]){
                    dp[i][j]=true;
                    count++;
                }
                else dp[i][j]=false;
            }
            else
                if(s[i]==s[j] and dp[i+1][j-1]==true){
                    dp[i][j]=true;
                    count++;
                }
                else dp[i][j]=false;
        }
    }
    return count;
}

```

33. Search in Rotated Sorted Array

Medium 12430 828 Add to List Share

There is an integer array `nums` sorted in ascending order (with **distinct** values).

Prior to being passed to your function, `nums` is **possibly rotated** at an unknown pivot index `k` (`1 <= k < nums.length`) such that the resulting array is `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]` (**0-indexed**). For example, `[0,1,2,4,5,6,7]` might be rotated at pivot index `3` and become `[4,5,6,7,0,1,2]`.

Given the array `nums` **after** the possible rotation and an integer `target`, return *the index of* `target` *if it is in* `nums`, *or* `-1` *if it is not in* `nums`.

You must write an algorithm with `O(log n)` runtime complexity.

Example 1:

```
Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4
```

Example 2:

```
Input: nums = [4,5,6,7,0,1,2], target = 3
Output: -1
```

```
class Solution {
    private:
    int BinarySearch(vector<int>&nums, int l,int h,int key)
    {
        int start=l;
        int end=h;
        int mid = start+(end-start)/2;
        while(start<=end){
            if(nums[mid]==key)return mid;
            else if(nums[mid]<nums[end])
                if(key>nums[mid] and key<=nums[end])
                    start = mid+1;
                else
                    end = mid-1;
            else
                if(key>=nums[start] and key<nums[mid])
                    end = mid-1;
                else
                    start = mid+1;
            mid = start+(end-start)/2;
        }
        return -1;
    }
    public:
    int search(vector<int>& nums, int target)
    {
        return BinarySearch(nums,0,nums.size()-1,target);
    }
};
```

363. Max Sum of Rectangle No Larger Than K

Hard 1826 104 Add to List Share

Given an `m x n` matrix `matrix` and an integer `k`, return *the max sum of a rectangle in the matrix such that its sum is no larger than* `k`.

It is **guaranteed** that there will be a rectangle with a sum no larger than `k`.

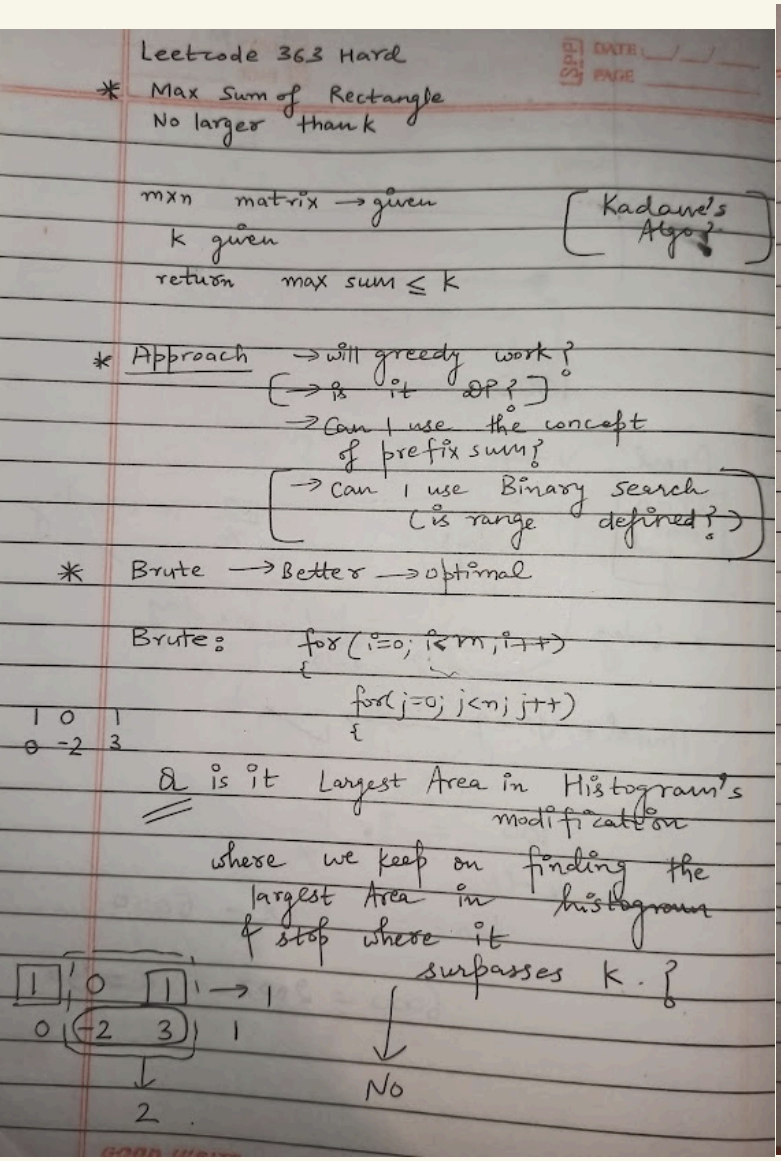
Example 1:

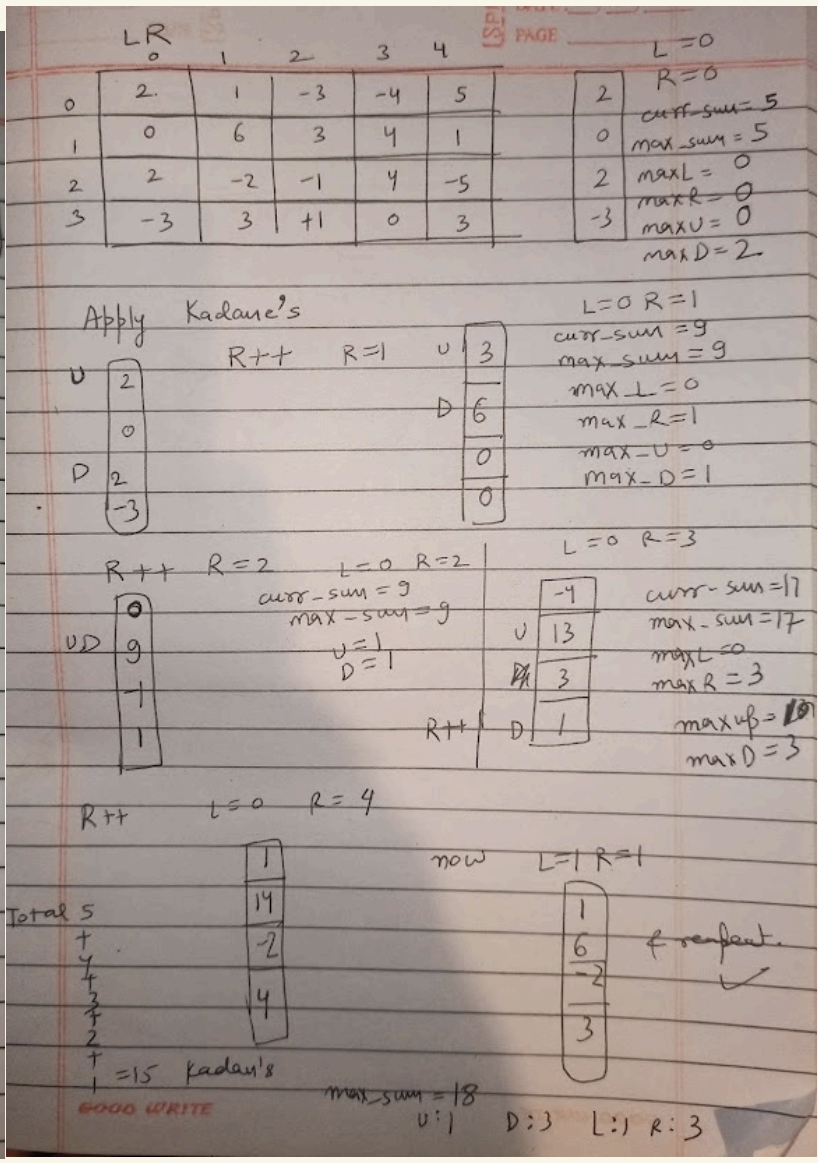
1	0	1
0	-2	3

```cpp
class Solution {
public:
    int maxSumSubmatrix(vector<vector<int>>& matrix, int k) {
        int m = matrix.size(), n = matrix[0].size();
        int res = INT_MIN;
        vector<int> row_sum(m, 0);
        for (int l = 0; l < n; ++l){
            fill(row_sum.begin(), row_sum.end(), 0);// in every iteration, we have to make all 0
            for (int r = l; r < n; ++r){
                // Kadane algorithm for a possible shortcut to next iteration or even an early return
                int cur_sum = 0, best_sum = INT_MIN;
                for (int i = 0; i < m; ++i){
                    row_sum[i] += matrix[i][r];
                    if (cur_sum < 0) cur_sum = row_sum[i];
                    else cur_sum += row_sum[i];
                    best_sum = max(best_sum, cur_sum);
                }
                if (best_sum == k) //return it.
                    return k;
                else if (best_sum < k) then keep on going.
                {
                    res = max(res, best_sum);
                    continue;
                }
                cur_sum = 0;
                set<int> s{0}; //create a set
                for (auto& sum: row_sum)
                {
                    cur_sum += sum;
                    auto it = s.lower_bound(cur_sum - k);
                    if (it != s.end())
                        res = max(res, cur_sum - *it);
                    if (res == k)
                        return k;
                    s.insert(cur_sum);
                }
            }
        }
        return res;
    }
};
```

kadane's algo

binary search





Factorials of large numbers

Medium Accuracy: 51.61% Submissions: 34391 Points: 4

Given an integer N, find its factorial.

Example 1:

```
Input: N = 5
Output: 120
Explanation : 5! = 1*2*3*4*5 = 120
```

Example 2:

```
Input: N = 10
Output: 3628800
Explanation :
10! = 1*2*3*4*5*6*7*8*9*10 = 3628800
```

Your Task:
You don't need to read input or print anything. Complete the function *factorial()* that takes integer N as input parameter and returns a list of integers denoting the digits that make up the factorial of N.

Expected Time Complexity : $O(N^2)$
Expected Auxilliary Space : O(1)

```cpp
// } Driver Code Ends
//User function template for C++

class Solution {
public:
    void multiply(vector<int> &v, int n){
        int carry = 0;
        for(int i=0; i<v.size(); i++){
            int data = v[i] * n + carry;
            v[i] = data % 10;
            carry=data/10;
        }
        while(carry){
            v.push_back(carry % 10);
            carry /= 10;
        }
    }
    vector<int> factorial(int N){
        vector<int> v = {1};
        for(int i=2; i<=N; i++){
            multiply(v,i);
        }

        reverse(v.begin(),v.end());
        return (v);
    }
};

// 5! = 120 n=5
// for loop will run from n=2 to n=5 i.e. 4 t
// and we pass vector v and the number i into
```

Design Circular | Design Snake Ga | Task Scheduler | Rearrange String | Maximum Numb | Moving Average | Problems - Leet | Inbox (34) - mat | Count the Re | Minimum numbe

practice.geeksforgeeks.org/problems/count-the-reversals0401/1#

Practice 831

Problem | Editorial | Submissions | Doubt Support | C++ (g++ 5.4) | Test against custom input

Count the Reversals

Medium Accuracy: 50.95% Submissions: 17146 Points: 4

Given a string **S** consisting of only opening and closing curly brackets **'{'** and **'}'**, find out the minimum number of reversals required to convert the string into a balanced expression.
A reversal means changing **'{'** to **'}'** or vice-versa.

Example 1:

```
Input:
S = "}{{}}{{{"
Output: 3
Explanation: One way to balance is:
"{{{}}{}}". There is no balanced sequence
that can be formed in lesser reversals.
```

â€‹**Example 2:**

```
Input:
S = "{{}{{}{{}}{{"
Output: -1
Explanation: There's no way we can balance
this sequence of braces.
```

Your Task:
You don't need to read input or print anything. Your task is to complete the function **countRev()** which takes the string S as input parameter and returns the minimum number of reversals required to balance the bracket sequence. If balancing is not possible, return -1.

Expected Time Complexity: O(|S|).
Expected Auxiliary Space: O(1).

```cpp
// Contributed By: Pranay Bansal// } Driver Code Ends

int countRev (string s){
    if (s.size()%2)return -1; //odd
    int open=0; int close=0,i=0;
    for (;i<s.size();i++)
        if (s[i]=='{') open++; //if you see open, open++
        else
            if(open==0) close++;
            else open--;
    return (int)(ceil(open/2.0)+ceil(close/2.0));
}
```

Average Time: 15m
Your Time: 2m 12s

Compile & Run | Submit

5. Longest Palindromic Substring

Medium 15791 930 Add to List Share

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"
```

Constraints:

- `1 <= s.length <= 1000`
- `s` consist of only digits and English letters.

```cpp
class Solution {
    public: //this function is checking, is the given range palindrome or not?
    void f(int &l,int &r, string &str,int &n, int &max_len, int &start){
        while(l>=0 and r<str.length() and str[l]==str[r]){ //check if it is in limit and matching.
            int len=r-l+1; //if so, update the length as string contained in range[l,r]
            if(len > max_len){ //update the max_len if you get a bigger len.
                start = l; //if you get so, start becomes the left side l and update your max_len
                max_len = len;
            }
            l--;r++; //increase l and decrease r.
        }
    }
    string longestPalindrome(string str){
        string ans; //where you are storing the result
        int max_len = 0; //here you are storing the length of the longest string.
        int start, end;//start and end.
        int n = str.size();//n=size of the string.
        for(int i=0;i<n;i++){ //for every character, run the for loop
            int l = i, r = i;//odd length palindrome
            f(l,r,str,n,max_len,start); //find length of in the max palindrome in range l,r
            l = i; r = i+1; //even length palindrome
            f(l,r,str,n,max_len,start); //find length of in the max palindrome in range l,r
        }
        return str.substr(start, max_len); //return such string, from the starting index to the length.
    }
};
```

Parsing of simple expressions

For the time being we only consider a simplified problem: we assume that all operators are **binary** (i.e. they take two arguments), and all are **left-associative** (if the priorities are equal, they get executed from left to right). Parentheses are allowed.

We will set up two stacks: one for numbers, and one for operators and parentheses. Initially both stacks are empty. For the second stack we will maintain the condition that all operations are ordered by strict descending priority. If there are parenthesis on the stack, than each block of operators (corresponding to one pair of parenthesis) is ordered, and the entire stack is not necessarily ordered.

We will iterate over the characters of the expression from left to right. If the current character is a digit, then we put the value of this number on the stack. If the current character is an opening parenthesis, then we put it on the stack. If the current character is a closing parenthesis, the we execute all operators on the stack until we reach the opening bracket (in other words we perform all operations inside the parenthesis). Finally if the current character is an operator, then while the top of the stack has an operator with the same or higher priority, we will execute this operation, and put the new operation on the stack.

After we processed the entire string, some operators might still be in the stack, so we execute them.

Here is the implementation of this method for the four operators $+ - * /$:

```cpp
bool delim(char c) {
    return c == ' ';
}

bool is_op(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/';
}

int priority (char op) {
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return -1;
}

void process_op(stack<int>& st, char op) {
    int r = st.top(); st.pop();
    int l = st.top(); st.pop();
    switch (op) {
        case '+': st.push(l + r); break;
        case '-': st.push(l - r); break;
        case '*': st.push(l * r); break;
        case '/': st.push(l / r); break;
    }
}
```

```cpp
int evaluate(string& s) {
    stack<int> st;
    stack<char> op;
    for (int i = 0; i < (int)s.size(); i++) {
        if (delim(s[i]))
            continue;

        if (s[i] == '(') {
            op.push('(');
        } else if (s[i] == ')') {
            while (op.top() != '(') {
                process_op(st, op.top());
                op.pop();
            }
            op.pop();
        } else if (is_op(s[i])) {
            char cur_op = s[i];
            while (!op.empty() && priority(op.top()) >= priority(cur_op)) {
                process_op(st, op.top());
                op.pop();
            }
            op.push(cur_op);
        } else {
            int number = 0;
            while (i < (int)s.size() && isalnum(s[i]))
                number = number * 10 + s[i++] - '0';
            --i;
            st.push(number);
        }
    }
```

```
    while (!op.empty()) {
        process_op(st, op.top());
        op.pop();
    }
    return st.top();
}
```

Thus we learned how to calculate the value of an expression in $O(n)$, at the same time we implicitly used the reverse Polish notation. By slightly modifying the above implementation it is also possible to obtain the expression in reverse Polish notation in an explicit form.

Unary operators

Now suppose that the expression also contains **unary** operators (operators that take one argument). The unary plus and unary minus are common examples of such operators.

One of the differences in this case, is that we need to determine whether the current operator is a unary or a binary one.

You can notice, that before an unary operator, there always is another operator or an opening parenthesis, or nothing at all (if it is at the very beginning of the expression). On the contrary before a binary operator there will always be an operand (number) or a closing parenthesis. Thus it is easy to flag whether the next operator can be unary or not.

Additionally we need to execute a unary and a binary operator differently. And we need to chose the priority of a unary operator higher than all of the binary operators.

In addition it should be noted, that some unary operators (e.g. unary plus and unary minus) are actually **right-associative**.

```
bool delim(char c) {
    return c == ' ';
}

bool is_op(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/';
}

bool is_unary(char c) {
    return c == '+' || c=='-';
}

int priority (char op) {
    if (op < 0) // unary operator
        return 3;
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return -1;
}

void process_op(stack<int>& st, char op) {
    if (op < 0) {
        int l = st.top(); st.pop();
        switch (-op) {
            case '+': st.push(l); break;
            case '-': st.push(-l); break;
        }
    } else {
        int r = st.top(); st.pop();
        int l = st.top(); st.pop();
        switch (op) {
            case '+': st.push(l + r); break;
            case '-': st.push(l - r); break;
            case '*': st.push(l * r); break;
            case '/': st.push(l / r); break;
        }
    }
}

int evaluate(string& s) {
    stack<int> st;
    stack<char> op;
    bool may_be_unary = true;
    for (int i = 0; i < (int)s.size(); i++) {
        if (delim(s[i]))
            continue;

        if (s[i] == '(') {
            op.push('(');
            may_be_unary = true;
        } else if (s[i] == ')') {
            while (op.top() != '(') {
                process_op(st, op.top());
                op.pop();
            }
            op.pop();
            may_be_unary = false;
        } else if (is_op(s[i])) {
            char cur_op = s[i];
            if (may_be_unary && is_unary(cur_op))
                cur_op = -cur_op;
            while (!op.empty() && (
                    (cur_op >= 0 && priority(op.top()) >= priority(cur_op)) ||
                    (cur_op < 0 && priority(op.top()) > priority(cur_op))
                )) {
                process_op(st, op.top());
                op.pop();
            }
            op.push(cur_op);
            may_be_unary = true;
        } else {
            int number = 0;
            while (i < (int)s.size() && isalnum(s[i]))
                number = number * 10 + s[i++] - '0';
            --i;
            st.push(number);
            may_be_unary = false;
        }
    }

    while (!op.empty()) {
        process_op(st, op.top());
        op.pop();
    }
    return st.top();
}
```

Right-associativity

Right-associative means, that whenever the priorities are equal, the operators must be evaluated from right to left.

As noted above, unary operators are usually right-associative. Another example for an right-associative operator is the exponentiation operator ($a \wedge b \wedge c$ is usually perceived as a^{b^c} and not as $(a^b)^c$).

What difference do we need to make in order to correctly handle right-associative operators? It turns out that the changes are very minimal. The only difference will be, if the priorities are equal we will postpone the execution of the right-associative operation.

The only line that needs to be replaced is

```
while (!op.empty() && priority(op.top()) >= priority(cur_op))
```

with

```
while (!op.empty() && (
        (left_assoc(cur_op) && priority(op.top()) >= priority(cur_op)) ||
        (!left_assoc(cur_op) && priority(op.top()) > priority(cur_op))
    ))
```

where `left_assoc` is a function that decides if an operator is left_associative or not.

Here is an implementation for the binary operators $+ - * /$ and the unary operators $+$ and $-$.

Number following a pattern

Number following a pattern

Medium Accuracy: 55.05% Submissions: 4391 Points: 4

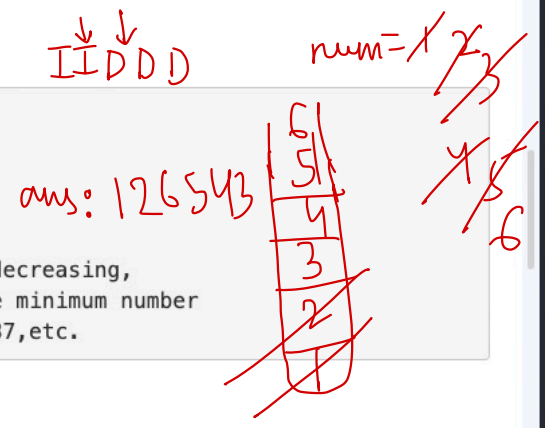
Given a pattern containing only I's and D's. I for increasing and D for decreasing.
Devise an algorithm to print the minimum number following that pattern.
Digits from 1-9 and digits can't repeat.

Example 1:

Input:
D
Output:
21
Explanation:
D is meant for decreasing,
So we choose the minimum number
among 21,31,54,87,etc.

Example 2:

Input:
IIDDD
Output:

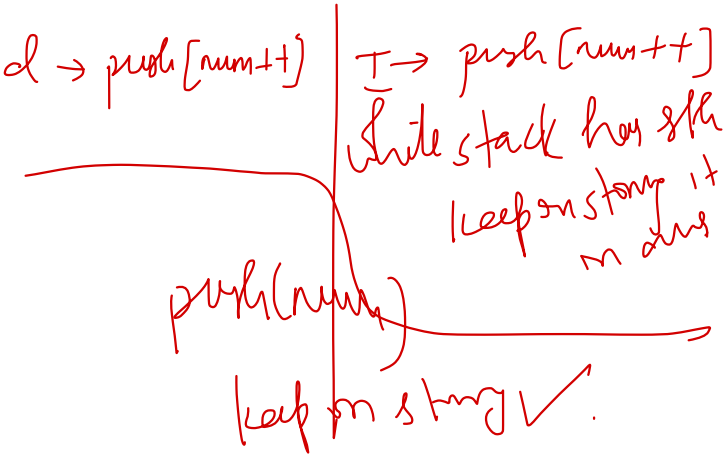



```
class Solution{
public:
    string printMinNumberForPattern(string s){
        string ans;
        stack<int> st;
        int num = 1;
        for(auto c:s){
            if(c == 'D'){
                st.push(num);
                num++;
            }
            else
            {
                st.push(num);
                num++;
                while(!st.empty())
                {
                    ans += to_string(st.top());
                    st.pop();
                }
            }
        st.push(num);
        while(!st.empty()){
            ans += to_string(st.top());
            st.pop();
        }
        return ans;
    }
};
```

We are taking a pattern here and the pattern contains only I and D's and now our task is to print the output as numbers.

If the input is IIDDD this means first two numbers are increasing and then next three numbers are decreasing, now one challenge is that we need to use minimum numbers, so once you see I you can keep on printing 1 and then if you see D then you have to push it into the stack. The idea of stack is that you will use minimum numbers and once you reach the end of the input string then you start popping the elements from the stack and start pushing it in the string end.

IIDDD
1
12
Push 3 in stack
push 4 in stack
Push 5 in stack
Reached end?
pop string top and Add 5 to string—>125
Pop string top and add 4 onto the string end—> 1254
pop string top and add 3 onto the string end—> 12543



Now it was the intuition, if the input begins with D push 1 followed by 2 in stack and then keep on adding numbers if you see other numbers in the input, then again do the same thing that you have to push the numbers

We are simply assigning num=1 and then doing case analysis if character is D then we do something else and if it is the I then you again do the same thing that you push the number in the stack and then you simply increment number.

Permutation of a given string

Permutations of a given string

Medium Accuracy: 49.85% Submissions: 36967 Points: 4

Given a string S. The task is to print all permutations of a given string in lexicographically sorted order.

Example 1:

```
Input: ABC
Output:
ABC ACB BAC BCA CAB CBA
Explanation:
Given string ABC has permutations in 6
forms as ABC, ACB, BAC, BCA, CAB and CBA .
```

Example 2:

```
Input: ABSG
Output:
ABGS ABSG AGBS AGSB ASBG ASGB BAGS
BASG BGAS BGSA BSAG BSGA GABS GASB
GBAS GBSA GSAB GSBA SABG SAGB SBAG
SBGA SGAB SGBA
Explanation:
Given string ABSG has 24 permutations.
```

```
// } Driver Code Ends
class Solution
{
    public:
        void f(string ip, string op, vector<string>&ans){
        int n = ip.length();
        if(n == 0){
            ans.push_back(op);
            return;
        }
        for(int i = 0;i<n;i++){
            string op1 = op;
            string ip1 = ip;
            op1.push_back(ip[i]);
            ip1.erase(ip1.begin()+i);
            f(ip1,op1,ans);
        }
        return;

    }
vector<string>find_permutation(string S)
{
    // Code here there
    vector<string> ans;
    string ip = S;
    string op = "";
    f(ip,op,ans);
    sort(ans.begin(), ans.end());
    return ans;
}
};
```

Create a vector of string type ans and then create two strings ip=S and a null string as op
Then call function f(ip, op, ans)
This function will do something.
Then we will sort the vector ans.
Now let us see what this function does
We are passing two strings ip and op into the function and passing the vector.
Then we are storing ip.length() in n. If n is 0 then we will simply push op into answer this means we have passed the ans vector as reference, so we need not worry about anything simply return, But If n>0 then in that case, we will run a for loop till the length of ip string.
Then we have two strings op1 and ip1 and we are copying op and ip into those and then we are pushing ip[I] in op[I] and then we are erasing ip1 from start to ith index and then we are calling f(ip1, op1,ans)

Longest palindrome in a string

Longest Palindrome in a String

Medium Accuracy: 49.2% Submissions: 48532 Points: 4

Given a string S, find the longest palindromic substring in S. **Substring of string S:** S[i . . . . j] where 0 ≤ i ≤ j < len(S). **Palindrome string:** A string which reads the same backwards. More formally, S is palindrome if reverse(S) = S. **Incase of conflict**, return the substring which occurs first (with the least starting index).

Example 1:

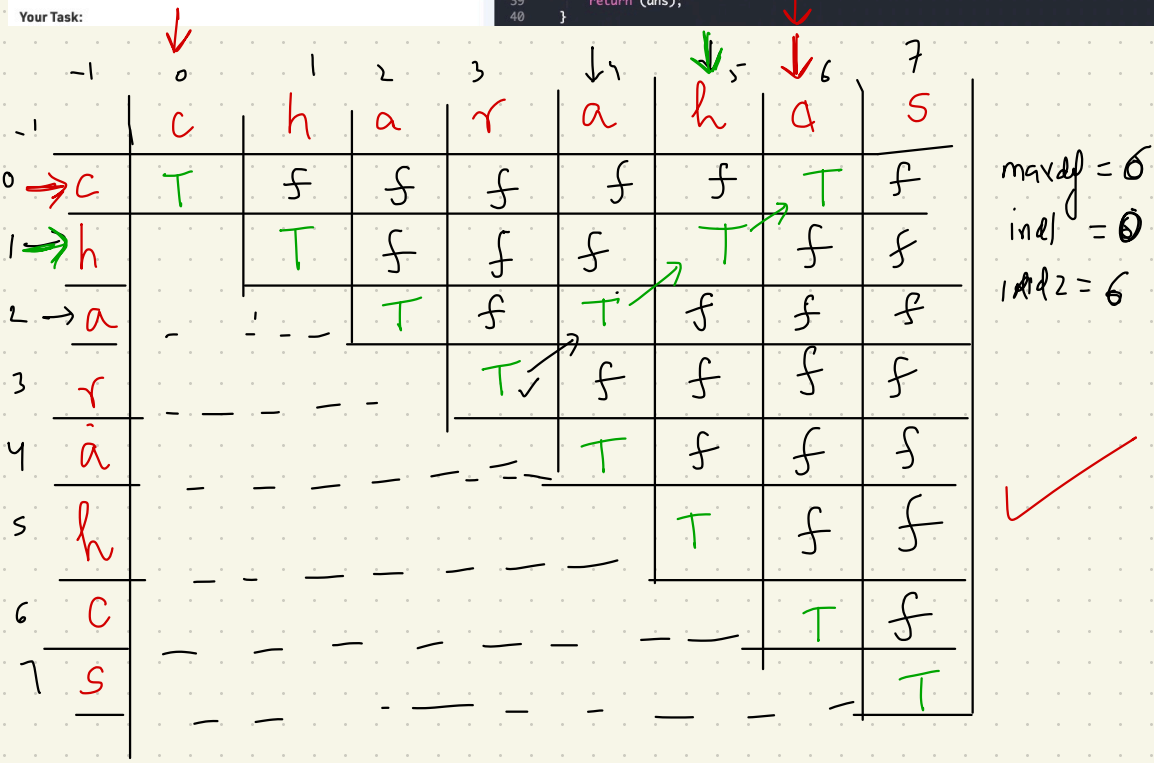
```
Input:
S = "aaaabbaa"
Output: aabbaa
Explanation: The longest Palindromic
substring is "aabbaa".
```

Example 2:

```
Input:
S = "abc"
Output: a
Explanation: "a", "b" and "c" are the
longest palindromes with same length.
The result is the one with the least
starting index.
```

Your Task:

```cpp
class Solution {
  public:
    string longestPalin (string S) {
        int n = S.length();
        vector<vector<bool>> dp(n,vector<bool>(n,false));
        for(int gap = 0; gap<n; gap++){
            for(int i=0,j=gap; j<n; i++,j++){
                if(gap == 0) dp[i][j] = true; //for 0 length palindrome
                else if(gap == 1){ //for single length palindrome
                    if(S[i] == S[j]) dp[i][j] = true; //if the char are same, then true
                    else dp[i][j] = false; //else false
                }
                else{ //for more than two length palindrome
                    if(S[i] == S[j]) dp[i][j] = dp[i+1][j-1]; //next row and prev col
                    else dp[i][j] = false;
                }
            }
        }

        int ind1 = -1, ind2 = -1, max_diff = -1;
        for(int i=0; i<n; i++)
            for(int j=0; j<n; j++)
                if(dp[i][j] and j-i > max_diff){
                    max_diff = j-i;
                    ind1 = i;
                    ind2 = j;
                }
        string ans{};
        for(int i = ind1; i <= ind2; i++){
            ans += S[i];
        }

        return (ans);
    }
```



Substrings of length k with k-1 elements

992. Subarrays with K Different Integers

Hard 2699 39 Add to List Share

Given an integer array `nums` and an integer `k`, return *the number of **good subarrays** of* `nums`.

A **good array** is an array where the number of different integers in that array is exactly `k`.

- For example, `[1,2,3,1,2]` has `3` different integers: `1`, `2`, and `3`.

A **subarray** is a **contiguous** part of an array.

Example 1:

```
Input: nums = [1,2,1,2,3], k = 2
Output: 7
Explanation: Subarrays formed with exactly 2 different integers: [1,2], [2,1], [1,2], [2,3], [1,2,1], [2,1,2], [1,2,1,2]
```

```cpp
class Solution {
public:

    int ok(vector<int>& A , int K){
        if(K==0) return 0;
        int n=A.size(),total=0,diff=0,j=0;
        vector<int>cnt(20002);
        for(int i=0;i<n;i++){
            if(cnt[A[i]]==0)diff++;
            cnt[A[i]]++;
            if(diff<=K) total+=(i-j+1);
            else{
                while(j<n and j<=i and diff>K){
                    cnt[A[j]]--;
                    if(cnt[A[j]]==0) diff--;
                    j++;
                }
                total+=(i-j+1);
            }
        }
        return total;
    }
    int subarraysWithKDistinct(vector<int>& A, int K) {
        return ok(A,K)-ok(A,K-1);
    }
};
```

Repeated string match

686. Repeated String Match

Medium 1313 889 Add to List Share

Given two strings `a` and `b`, return *the minimum number of times you should repeat string* `a` *so that string* `b` *is a substring of it*. If it is impossible for `b` to be a substring of `a` after repeating it, return `-1`.

Notice: string `"abc"` repeated 0 times is `""`, repeated 1 time is `"abc"` and repeated 2 times is `"abcabc"`.

Example 1:

```
Input: a = "abcd", b = "cdabcdab"
Output: 3
Explanation: We return 3 because by repeating a three times "abcdabcdabcd", b is a substring of it.
```

Example 2:

```
Input: a = "a", b = "aa"
Output: 2
```

Constraints:

```cpp
class Solution {
public:
    int KMP(const string& a, const string& b) {
        int m = a.size(); //size of string 1
        int n = b.size(); //size of string 2
        int k = 1 + int(ceil(double(n-1) / double(m))); //k
        string s;
        for(int i=0; i<k; i++) s += a; //add string a in itself k times
        int kmp[n]; //kmp array
        memset(kmp, 0, sizeof(kmp)); //set it 0.
        int i = 1, j = 0; //keep i at index 1 and j at index 0
        while(i < n) { //go till string2 length
            if(b[i] == b[j]) //if you see repeated instance of a char in b
                kmp[i++] = j++ +1; //store the next index of it in kmp[i] and increment i,j after
            else  //if they are not matching, then
                if(!j) i++; //if j is at start, increment i
                else j = kmp[j-1]; //if not, j will be whatever previously was it kmp[j-1]
        }
        int l=s.size(); //store size of string s in l
        i = 0, j = 0; //assign i,j as 0
        while(i+n<=j+l and j<n) //if b's length + i has not exceeded s'th length +j
            if(s[i]==b[j]) i++,j++; //if they are same, move both pointers one step ahead
            else
                if(!j) i++; //if j is at 0, just increase i
                else j = kmp[j-1]; //otherwise whatever is at kmp[j-1] store it in j
        if(j < n) return -1; //if j has not yet exceeded n, return -1
        return int(ceil(double(i) / double(m))); //return ceil(i/m) as we need these many repetetions
    }
    int repeatedStringMatch(string a, string b) {
        return KMP(a, b);
    }

};
```

Rolling hash method

686. Repeated String Match

Medium 1313 889 Add to List Share

Given two strings `a` and `b`, return *the minimum number of times you should repeat string* `a` *so that string* `b` *is a substring of it*. If it is impossible for `b` to be a substring of `a` after repeating it, return `-1`.

Notice: string `"abc"` repeated 0 times is `""`, repeated 1 time is `"abc"` and repeated 2 times is `"abcabc"`.

Example 1:

```
Input: a = "abcd", b = "cdabcdab"
Output: 3
Explanation: We return 3 because by repeating a three times "abcdabcdabcd", b is a substring of it.
```

Example 2:

```
Input: a = "a", b = "aa"
Output: 2
```

```cpp
class Solution {
public:
    int RollingHash(const string& a, const string& b){
        int m = a.size(),n = b.size(); //store the size of both strings
        int k = 1+int(ceil(double(n-1)/double(m))); //number of times you will copy a in s
        string s;
        for(int i=0;i<k;i++) s+=a; //concatenate a in s k times
        long M=1e9+7,hash=0,target=0,power=1;
        for(int i=0;i<n;i++){ //for every character in b
            hash=(hash*26+(long)(s[i]-'a'))%M; //calculate hash of that character modulo M
            target=(target*26+(long)(b[i]-'a'))%M; //calculate hash of char in s store in target
            if(i) power=(power*26) % M; // keep on increasing power modulo M
        }
        if(hash==target and s.substr(0,n)==b) return int(ceil(double(n)/double(m)));
        //return the number of times, if both hashes and target match,
        //( if a and b are same then only this condition will return)
        for(int i=1;i<m;i++){ //for every character in a from index 1
            hash=((hash-(long)(s[i-1]-'a')*power)*26 + (long)(s[i+n-1]-'a'))%M; //rolling hash method
            if(hash<0) hash+=M; //if negetive hash, roll the hash
            if(hash==target and s.substr(i,n)==b) return int(ceil(double(i+n)/double(m)));
            //return the number of times, if both hashes and target match and b is present in a
        }
        return -1; //return -1 if you can never find b in a :)
    }

    int repeatedStringMatch(string a, string b){
        return RollingHash(a, b);
    }
};
```