654. Maximum Binary Tree

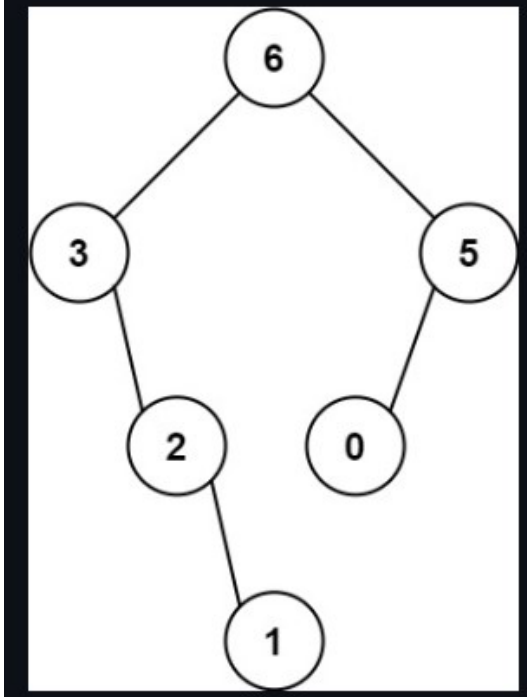
Medium

You are given an integer array `nums` with no duplicates. A **maximum binary tree** can be built recursively from `nums` using the following algorithm:

1. Create a root node whose value is the maximum value in `nums`.
2. Recursively build the left subtree on the **subarray prefix** to the **left** of the maximum value.
3. Recursively build the right subtree on the **subarray suffix** to the **right** of the maximum value.

Return *the **maximum binary tree** built from* `nums`.

Example 1:




```
Input: nums = [3,2,1,6,0,5]
Output: [6,3,5,null,2,0,null,null,1]
Explanation: The recursive calls are as follow:
```

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    TreeNode* constructMaximumBinaryTree(vector<int>& nums) {//loved this solution
        vector<TreeNode*>stack;
        for(int &num:nums){
            TreeNode* cur=new TreeNode(num);
            while(!stack.empty() and stack.back()->val<num){
                cur->left=stack.back();
                stack.pop_back();
            }
            if(!stack.empty()){
                stack.back()->right=cur;
            }
            stack.push_back(cur);
        }
        return stack.front();
    }
};
```

2130. Maximum Twin Sum of a Linked List

Medium

In a linked list of size `n`, where `n` is **even**, the `i^th` node (**0-indexed**) of the linked list is known as the **twin** of the `(n-1-i)^th` node, if `0 <= i <= (n / 2) - 1`.

- For example, if `n = 4`, then node `0` is the twin of node `3`, and node `1` is the twin of node `2`. These are the only nodes with twins for `n = 4`.

The **twin sum** is defined as the sum of a node and its twin.

Given the `head` of a linked list with even length, return *the **maximum twin sum** of the linked list.*

Example 1:


```
Input: head = [5,4,2,1]
Output: 6
Explanation:
Nodes 0 and 1 are the twins of nodes 3 and 2, respectively. All have twin sum = 6.
There are no other nodes with twins in the linked list.
Thus, the maximum twin sum of the linked list is 6.
```

Example 2:


```
Input: head = [4,2,2,3]
Output: 7
Explanation:
The nodes with twins present in this linked list are:
- Node 0 is the twin of node 3 having a twin sum of 4 + 3 = 7.
```

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    int pairSum(ListNode* head) {
        vector<int>a1;

        ListNode* i=head,*j=head;

        while(j&&j->next)
        {
            a1.push_back(i->val);
            i=i->next;
            j=j->next->next;
        }
        int ans=INT_MIN;
        int index=0;
        while(i)
        {
            ans=max(ans,a1[a1.size()-index-1]+i->val);
            i=i->next;
            index++;
        }
        return ans;
    }
};
/*the apporach is of O(n) tc and O(n) space*/
```

1008. Construct Binary Search Tree from Preorder Traversal

Medium

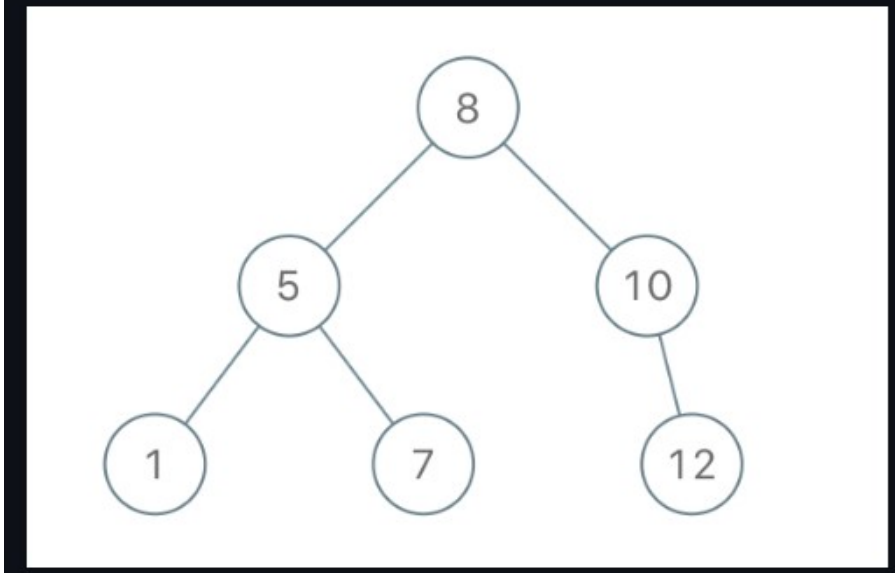
Given an array of integers preorder, which represents the **preorder traversal** of a BST (i.e., **binary search tree**), construct the tree and return *its root*.

It is **guaranteed** that there is always possible to find a binary search tree with the given requirements for the given test cases.

A **binary search tree** is a binary tree where for every node, any descendant of `Node.left` has a value **strictly less than** `Node.val`, and any descendant of `Node.right` has a value **strictly greater than** `Node.val`.

A **preorder traversal** of a binary tree displays the value of the node first, then traverses `Node.left`, then traverses `Node.right`.

Example 1:




```
Input: preorder = [8,5,1,7,10,12]
Output: [8,5,10,1,7,null,12]
```

Example 2:

```
Input: preorder = [1,3]
Output: [1,null,3]
```

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    TreeNode* bstFromPreorder(vector<int>& preorder) {//recursive stack,visualise for better understanding.
        int i=0;
        return build(preorder,i,INT_MAX);
    }
    TreeNode* build(vector<int>&a,int &i,int bound ){
        if(i==a.size()||a[i]>bound)return NULL;
        TreeNode*root=new TreeNode(a[i++]);
        root->left=build(a,i,root->val);
        root->right=build(a,i,bound);
        return root;
    }
};
```

1472. Design Browser History

Medium

You have a browser of one tab where you start on the `homepage` and you can visit another `url`, get back in the history number of `steps` or move forward in the history number of `steps`.

Implement the `BrowserHistory` class:

- `BrowserHistory(string homepage)` Initializes the object with the `homepage` of the browser.
- `void visit(string url)` Visits `url` from the current page. It clears up all the forward history.
- `string back(int steps)` Move `steps` back in history. If you can only return `x` steps in the history and `steps > x`, you will return only `x` steps. Return the current `url` after moving back in history at most `steps`.
- `string forward(int steps)` Move `steps` forward in history. If you can only forward `x` steps in the history and `steps > x`, you will forward only `x` steps. Return the current `url` after forwarding in history at most `steps`.

Example:

```
Input:
["BrowserHistory","visit","visit","visit","back","back","forward","visit","forward","back","back"]
[["leetcode.com"],["google.com"],["facebook.com"],["youtube.com"],[1],[1],[1],["linkedin.com"],[2],[2],[7]]
Output:
[null,null,null,null,"facebook.com","google.com","facebook.com",null,"linkedin.com","google.com","leetcode.com"]

Explanation:
BrowserHistory browserHistory = new BrowserHistory("leetcode.com");
browserHistory.visit("google.com");       // You are in "leetcode.com". Visit "google.com"
browserHistory.visit("facebook.com");     // You are in "google.com". Visit "facebook.com"
browserHistory.visit("youtube.com");      // You are in "facebook.com". Visit "youtube.com"
browserHistory.back(1);                   // You are in "youtube.com", move back to "facebook.com" return "facebook.com"
browserHistory.back(1);                   // You are in "facebook.com", move back to "google.com" return "google.com"
browserHistory.forward(1);                // You are in "google.com", move forward to "facebook.com" return "facebook.com"
browserHistory.visit("linkedin.com");     // You are in "facebook.com". Visit "linkedin.com"
browserHistory.forward(2);                // You are in "linkedin.com", you cannot move forward any steps.
browserHistory.back(2);                   // You are in "linkedin.com", move back two steps to "facebook.com" then to "google.com". r
browserHistory.back(7);                   // You are in "google.com", you can move back only one step to "leetcode.com". return "leet
```

Constraints:

- `1 <= homepage.length <= 20`
- `1 <= url.length <= 20`
- `1 <= steps <= 100`
- `homepage` and `url` consist of `'.'` or lower case English letters.
- At most `5000` calls will be made to `visit`, `back`, and `forward`.

```cpp
class doubly{
        public:
        string s;
        doubly *next,*prev;
        doubly(string url){
            s=url;
            prev=NULL;
            next=NULL;
        }
    };

class BrowserHistory {
public:
    //these design based questions are juat amazing, I knew that the browsor history is implemented using the linkedlists
    doubly *node,*curr;
    BrowserHistory(string homepage) {
        node=new doubly(homepage);
        curr=node;
    }

    void visit(string url) {//creating a new node for the new URL being visited, then we also assign the next and prev nodes to the existing curr and the next to the new node and then the current to the  new node as well
        doubly *NewNode= new doubly(url);
        curr->next=NewNode;
        NewNode->prev=curr;
        curr=NewNode;
    }

    string back(int steps) {//very easy implementation of the back opertion.
        while(steps&&curr->prev){
            curr=curr->prev;
            steps--;
        }
        return curr->s;
    }

    string forward(int steps) {
        while(steps&&curr->next){
            curr=curr->next;
            steps--;}
    return curr->s;
        }
};
```

1381. Design a Stack With Increment Operation

Medium

Design a stack that supports increment operations on its elements.

Implement the `CustomStack` class:

- `CustomStack(int maxSize)` Initializes the object with `maxSize` which is the maximum number of elements in the stack.
- `void push(int x)` Adds `x` to the top of the stack if the stack has not reached the `maxSize`.
- `int pop()` Pops and returns the top of the stack or `-1` if the stack is empty.
- `void inc(int k, int val)` Increments the bottom `k` elements of the stack by `val`. If there are less than `k` elements in the stack, increment all the elements in the stack.

Example 1:

```
Input
["CustomStack","push","push","pop","push","push","push","increment","increment","pop","pop","pop","pop"]
[[3],[1],[2],[],[2],[3],[4],[5,100],[2,100],[],[],[],[]]
Output
[null,null,null,2,null,null,null,null,null,103,202,201,-1]
Explanation
CustomStack stk = new CustomStack(3); // Stack is Empty []
stk.push(1);                          // stack becomes [1]
stk.push(2);                          // stack becomes [1, 2]
stk.pop();                            // return 2 --> Return top of the stack 2, stack becomes [1]
stk.push(2);                          // stack becomes [1, 2]
stk.push(3);                          // stack becomes [1, 2, 3]
stk.push(4);                          // stack still [1, 2, 3], Do not add another elements as size is 4
stk.increment(5, 100);                // stack becomes [101, 102, 103]
stk.increment(2, 100);                // stack becomes [201, 202, 103]
stk.pop();                            // return 103 --> Return top of the stack 103, stack becomes [201, 202]
stk.pop();                            // return 202 --> Return top of the stack 202, stack becomes [201]
stk.pop();                            // return 201 --> Return top of the stack 201, stack becomes []
stk.pop();                            // return -1 --> Stack is empty return -1.
```

```cpp
class CustomStack {
public://this code uses lazy incrementation of val(i.e val is passed down in another array at kth position and added during pop operation)
     vector<int> stack, inc;
    int n;
    CustomStack(int maxSize) {
        n = maxSize;
    }

    void push(int x) {
        if (stack.size() == n) return;
        stack.push_back(x);
        inc.push_back(0);
    }

    int pop() {
        int i = stack.size() - 1;
        if (i < 0) return -1;
        if (i > 0) {inc[i - 1] += inc[i];cout<<inc[i-1]<<"\n";}
        int res = stack[i] + inc[i];
        stack.pop_back();
        inc.pop_back();
        return res;
    }

    void increment(int k, int val) {
        int i = min(k, (int)stack.size()) - 1;
        if (i >= 0) inc[i] += val;
    }
};
```

921. Minimum Add to Make Parentheses Valid

Medium

A parentheses string is valid if and only if:

- It is the empty string,
- It can be written as `AB` (`A` concatenated with `B`), where `A` and `B` are valid strings, or
- It can be written as `(A)`, where `A` is a valid string.

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

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

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

Example 1:

```
Input: s = "())"
Output: 1
```

Example 2:

```
Input: s = "((("
Output: 3
```

```cpp
class Solution {
public:
    int minAddToMakeValid(string s) {//this wasn't supposed to be medium and maybe there could be other folllow ups of this problem by introducing other symbols.
        int res=0,balance=0;
         for(char c:s){
            balance+=(c=='(')?1:-1;
             if(balance==-1)res++,balance++;
        }
    return abs(res)+abs(balance);}
};
```

2375. Construct Smallest Number From DI String

Medium

You are given a **0-indexed** string `pattern` of length `n` consisting of the characters `'I'` meaning **increasing** and `'D'` meaning **decreasing**.

A **0-indexed** string `num` of length `n + 1` is created using the following conditions:

- `num` consists of the digits `'1'` to `'9'`, where each digit is used **at most** once.
- If `pattern[i] == 'I'`, then `num[i] < num[i + 1]`.
- If `pattern[i] == 'D'`, then `num[i] > num[i + 1]`.

Return *the lexicographically **smallest** possible string* `num` *that meets the conditions.*

Example 1:

```
Input: pattern = "IIIDIDDD"
Output: "123549876"
Explanation:
At indices 0, 1, 2, and 4 we must have that num[i] < num[i+1].
At indices 3, 5, 6, and 7 we must have that num[i] > num[i+1].
Some possible values of num are "245639871", "135749862", and "123849765".
It can be proven that "123549876" is the smallest possible num that meets the conditions.
Note that "123414321" is not possible because the digit '1' is used more than once.
```

Example 2:

```
Input: pattern = "DDD"
Output: "4321"
Explanation:
Some possible values of num are "9876", "7321", and "8742".
It can be proven that "4321" is the smallest possible num that meets the conditions.
```

```cpp
class Solution {
public:
    string smallestNumber(string pat) {
       int n=pat.length();
        // int maxx=min(n,9);
        int id=0;
        char minn='1';
        string res="1";
        for(int i=0;i<n;i++){
            char ch=pat[i];
            if(ch=='I'){
                res.push_back(++minn);
                id=res.size()-1;
            }
            else {
                res.push_back(++minn);
                for(int k=res.size()-1;k>id;k--){
                    swap(res[k],res[k-1]);
                }
            }
        }
    return res;}
};
```

2390. Removing Stars From a String

Medium

You are given a string `s`, which contains stars `*`.

In one operation, you can:

- Choose a star in `s`.
- Remove the closest **non-star** character to its **left**, as well as remove the star itself.

Return *the string after **all** stars have been removed.*

Note:

- The input will be generated such that the operation is always possible.
- It can be shown that the resulting string will always be unique.

Example 1:

```
Input: s = "leet**cod*e"
Output: "lecoe"
Explanation: Performing the removals from left to right:
- The closest character to the 1st star is 't' in "leet**cod*e". s becomes "lee*cod*e".
- The closest character to the 2nd star is 'e' in "lee*cod*e". s becomes "lecod*e".
- The closest character to the 3rd star is 'd' in "lecod*e". s becomes "lecoe".
There are no more stars, so we return "lecoe".
```

Example 2:

```
Input: s = "erase*****"
Output: ""
Explanation: The entire string is removed, so we return an empty string.
```

```cpp
class Solution {
public:
    string removeStars(string s) {
        string res;
        for(char c:s){
            if(c=='*'||c=='!'||c=='@'||c=='#'||c=='$'){
                res.pop_back();
            }
            else res+=c;
        }
        return res;

    }
};
```

946. Validate Stack Sequences

Medium

Given two integer arrays `pushed` and `popped` each with distinct values, return `true` *if this could have been the result of a sequence of push and pop operations on an initially empty stack, or* `false` *otherwise.*

Example 1:

```
Input: pushed = [1,2,3,4,5], popped = [4,5,3,2,1]
Output: true
Explanation: We might do the following sequence:
push(1), push(2), push(3), push(4),
pop() -> 4,
push(5),
pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1
```

Example 2:

```
Input: pushed = [1,2,3,4,5], popped = [4,3,5,1,2]
Output: false
Explanation: 1 cannot be popped before 2.
```

Constraints:

- `1 <= pushed.length <= 1000`
- `0 <= pushed[i] <= 1000`
- All the elements of `pushed` are **unique**.
- `popped.length == pushed.length`
- `popped` is a permutation of `pushed`.

```cpp
class Solution {
public:
    bool validateStackSequences(vector<int> &push, vector<int> &pop) {
        stack<int> s;
        for (auto i = 0, j = 0; i < push.size(); ++i) {
            s.push(push[i]);
            while (!s.empty() && s.top() == pop[j]) s.pop(), ++j;
        }
        return s.empty();
    }
};
```

2487. Remove Nodes From Linked List

Medium

You are given the `head` of a linked list.

Remove every node which has a node with a greater value anywhere to the right side of it.

Return *the* `head` *of the modified linked list.*

Example 1:


```
Input: head = [5,2,13,3,8]
Output: [13,8]
Explanation: The nodes that should be removed are 5, 2 and 3.
- Node 13 is to the right of node 5.
- Node 13 is to the right of node 2.
- Node 8 is to the right of node 3.
```

Example 2:

```
Input: head = [1,1,1,1]
Output: [1,1,1,1]
Explanation: Every node has value 1, so no nodes are removed.
```

Constraints:

- The number of the nodes in the given list is in the range `[1, 10^5]`.
- `1 <= Node.val <= 10^5`

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
// This looks like LIS in linkedlist
    ListNode* removeNodes(ListNode* head) {
        ListNode*curr=head;
        vector<int>s;
        while(curr){
            while(s.size()!=0&&s.back()<curr->val)s.pop_back();
            s.push_back(curr->val);
            curr=curr->next;
        }
        auto newnode=head;
        curr=head;
        for(auto ptr:s){
            newnode->val=ptr;
            curr=newnode;
            newnode=newnode->next;
        }
        curr->next=NULL;
        return head;
    }
};
```

901. Online Stock Span

Medium

Design an algorithm that collects daily price quotes for some stock and returns **the span** of that stock's price for the current day.

The **span** of the stock's price in one day is the maximum number of consecutive days (starting from that day and going backward) for which the stock price was less than or equal to the price of that day.

- For example, if the prices of the stock in the last four days is `[7,2,1,2]` and the price of the stock today is `2`, then the span of today is `4` because starting from today, the price of the stock was less than or equal `2` for `4` consecutive days.
- Also, if the prices of the stock in the last four days is `[7,34,1,2]` and the price of the stock today is `8`, then the span of today is `3` because starting from today, the price of the stock was less than or equal `8` for `3` consecutive days.

Implement the `StockSpanner` class:

- `StockSpanner()` Initializes the object of the class.
- `int next(int price)` Returns the **span** of the stock's price given that today's price is `price`.

Example 1:

```
Input
["StockSpanner", "next", "next", "next", "next", "next", "next", "next"]
[[], [100], [80], [60], [70], [60], [75], [85]]
Output
[null, 1, 1, 1, 2, 1, 4, 6]

Explanation
StockSpanner stockSpanner = new StockSpanner();
stockSpanner.next(100); // return 1
stockSpanner.next(80);  // return 1
stockSpanner.next(60);  // return 1
stockSpanner.next(70);  // return 2
stockSpanner.next(60);  // return 1
stockSpanner.next(75);  // return 4, because the last 4 prices (including today's price of 75) were less than or equal to today's pri
stockSpanner.next(85);  // return 6
```

```cpp
class StockSpanner {
public:

    stack<pair<int,int>>st;
//     prev-price-LRU freq as pairs
    int next(int price) {
        int res=1;
        while(!st.empty()&&st.top().first<=price){
            res+=st.top().second;
            st.pop();
        }
        st.push({price,res});
        return res;
    }
};
```

856. Score of Parentheses

Medium

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

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

- `"()"` has score `1`.
- `AB` has score `A + B`, where `A` and `B` are balanced parentheses strings.
- `(A)` has score `2 * A`, where `A` is a balanced parentheses string.

Example 1:

```
Input: s = "()"
Output: 1
```

Example 2:

```
Input: s = "(())"
Output: 2
```

Example 3:

```
Input: s = "()()"
Output: 2
```

Constraints:

- `2 <= s.length <= 50`
- `s` consists of only `'('` and `')'`.
- `s` is a balanced parentheses string.

```cpp
class Solution {
public:
    // again an edge case issue
    int scoreOfParentheses(string s) {
//         stack<int>st;
//         int i=0;
//         int res=0;
//         int mul=0;
//         while(i<s.length()){
//             while(s[i]=='('){
//                 mul++;
//                 i++;
//             }
//             res+=pow(2,mul);
//             while(s[i]==')'){
//                 mul--;
//                 i++;
//             }

//     return res;
//     }
     int depth = 0, res = 0;
        char prev = '(';

        for (const char &ch: s) {
            if (ch == '(') {
                depth++;
            } else {
                depth--;
                if (prev == '(')
                    res += 1<<depth;   //changed this part
            }

            prev = ch;
        }

        return res;
    }
};
// "((()(()))" 6
```

503. Next Greater Element II

Medium

Given a circular integer array `nums` (i.e., the next element of `nums[nums.length - 1]` is `nums[0]`), return *the **next greater number** for every element in* `nums`.

The **next greater number** of a number `x` is the first greater number to its traversing-order next in the array, which means you could search circularly to find its next greater number. If it doesn't exist, return `-1` for this number.

Example 1:

```
Input: nums = [1,2,1]
Output: [2,-1,2]
Explanation: The first 1's next greater number is 2;
The number 2 can't find next greater number.
The second 1's next greater number needs to search circularly, which is also 2.
```

Example 2:

```
Input: nums = [1,2,3,4,3]
Output: [2,3,4,-1,4]
```

Constraints:

- $1 <= nums.length <= 10^4$
- $-10^9 <= nums[i] <= 10^9$

```cpp
class Solution {
public:
//     stack problem....
    vector<int> nextGreaterElements(vector<int>& nums) {
        stack<int>s;
        vector<int>res(nums.size(),-1);
        for(int i=2*nums.size()-1;i>=0;i--){
            while(!s.empty()&&nums[s.top()]<=nums[i%nums.size()])s.pop();
            // res[i%nums.size()]=
            if(s.empty())res[i%nums.size()]=-1;
            else res[i%nums.size()]=nums[s.top()];
            s.push(i%nums.size());
        }
    return res;}
};
```

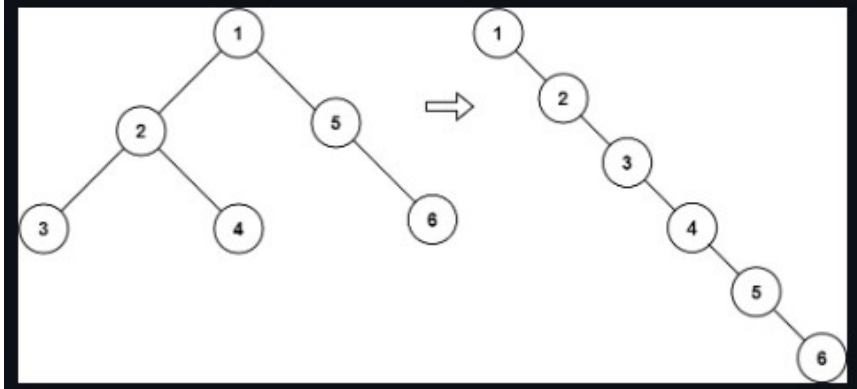
114. Flatten Binary Tree to Linked List

Medium

Given the `root` of a binary tree, flatten the tree into a "linked list":

- The "linked list" should use the same `TreeNode` class where the `right` child pointer points to the next node in the list and the `left` child pointer is always `null`.
- The "linked list" should be in the same order as a pre-order traversal of the binary tree.

Example 1:




```
Input: root = [1,2,5,3,4,null,6]
Output: [1,null,2,null,3,null,4,null,5,null,6]
```

Example 2:

```
Input: root = []
Output: []
```

Example 3:

```
Input: root = [0]
Output: [0]
```

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    // TreeNode* preorder(TreeNode*root){
    // }
    void flatten(TreeNode* root) {
        TreeNode*curr=root;
        while(curr){
            if(curr->left){
                TreeNode*prev=curr->left;
                while(prev->right)prev=prev->right;
                prev->right=curr->right;
                curr->right=curr->left;
                curr->left=NULL;
            }
            curr=curr->right;
        }
    }
};
```

394. Decode String

Medium

Given an encoded string, return its decoded string.

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

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

The test cases are generated so that the length of the output will never exceed 10^5.

Example 1:

```
Input: s = "3[a]2[bc]"
Output: "aaabcbc"
```

Example 2:

```
Input: s = "3[a2[c]]"
Output: "accaccacc"
```

Example 3:

```
Input: s = "2[abc]3[cd]ef"
Output: "abcabccdcdcdef"
```

Constraints:

- `1 <= s.length <= 30`
- `s` consists of lowercase English letters, digits, and square brackets `'[]'`.

```cpp
class Solution {
public://seems like a stack problem
    string decodeString(string s) {
        stack<string>chars;
        stack<int>nums;
        string res;
        int num=0;
        for(char c:s) {
            if(isdigit(c)) {
                num=num*10+(c-'0');
            }
            else if(isalpha(c)) {
                res.push_back(c);
            }
            else if(c =='[') {
                chars.push(res);
                nums.push(num);
                res="";
                num=0;
            }
            else if(c==']') {
                string tmp=res;
                for(int i=0;i<nums.top()-1;++i) {
                    res+=tmp;
                }
                res=chars.top()+res;
                chars.pop(); nums.pop();
            }
        }
        return res;
    }
};
```

1209. Remove All Adjacent Duplicates in String II

Medium

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

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

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

Example 1:

```
Input: s = "abcd", k = 2
Output: "abcd"
Explanation: There's nothing to delete.
```

Example 2:

```
Input: s = "deeedbbcccbdaa", k = 3
Output: "aa"
Explanation:
First delete "eee" and "ccc", get "ddbbbdaa"
Then delete "bbb", get "dddaa"
Finally delete "ddd", get "aa"
```

Example 3:

```
Input: s = "pbbcggttciiippooaais", k = 2
Output: "ps"
```

Constraints:

- `1 <= s.length <= 10^5`

```cpp
class Solution {
public:
    string removeDuplicates(string s, int k) {
        stack<pair<char,int>>st;
//          char,freq;
        int n=s.length();
        for(int i=0;i<n;i++){
            if(st.empty()||st.top().first!=s[i])st.push({s[i],1});
            else{
                st.top().second++;
                if(st.top().second==k)st.pop();
            }
        }
        string res="";
        while(!st.empty()){
            for(int i=0;i<st.top().second;i++)res+=st.top().first;
            st.pop();
        }
        reverse(res.begin(),res.end());
        return res;
    }
};
```

143. Reorder List

Medium

You are given the head of a singly linked-list. The list can be represented as:

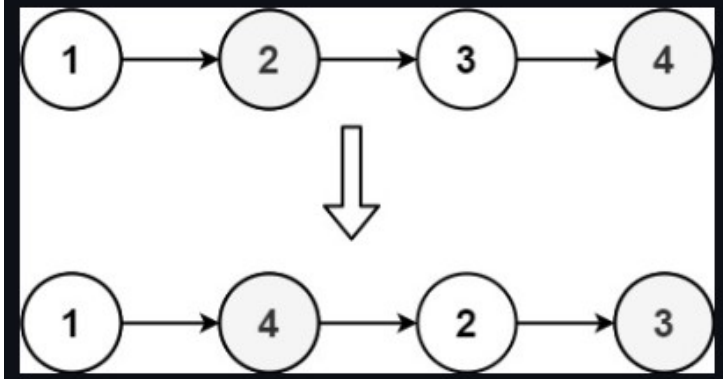
$$L_0 \rightarrow L_1 \rightarrow \ldots \rightarrow L_{n-1} \rightarrow L_n$$

Reorder the list to be on the following form:

$$L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \ldots$$

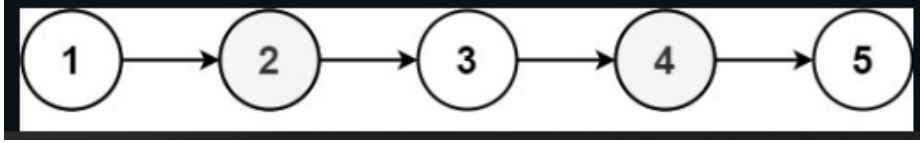
You may not modify the values in the list's nodes. Only nodes themselves may be changed.

Example 1:




```
Input: head = [1,2,3,4]
Output: [1,4,2,3]
```

Example 2:



```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    void reorderList(ListNode* head) {
      if(!head||!(head->next)||!(head->next->next))return;/*edge cases of the linked list ,basically we cant reorder the list cuz , theres no need of reordering when there is just 1,2,3 elements in the list */
        ListNode* ptr=head;
        int size=0;



            stack<ListNode *>st;// we push eevery node into the list
          while(ptr!=NULL){
             st.push(ptr);
              size++;
              ptr=ptr->next;
          }
          ListNode *pptr=head;
          for(int i=0;i<size/2;i++){/*and we also connect the alternative nodes to the top of the stack */
             ListNode *ele=st.top();
              st.pop();
              ele->next=pptr->next;
              pptr->next=ele;
              pptr=pptr->next->next;
          }
          pptr->next=NULL;//end of the list has to be assigned with NULL


    }
};
```

155. Min Stack

Medium

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the `MinStack` class:

- `MinStack()` initializes the stack object.
- `void push(int val)` pushes the element `val` onto the stack.
- `void pop()` removes the element on the top of the stack.
- `int top()` gets the top element of the stack.
- `int getMin()` retrieves the minimum element in the stack.

You must implement a solution with `O(1)` time complexity for each function.

Example 1:

```
Input
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

Output
[null,null,null,null,-3,null,0,-2]

Explanation
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top();    // return 0
minStack.getMin(); // return -2
```

```cpp
class MinStack {
public:
    vector<pair<int,int>>p;
    MinStack() {

    }

    void push(int val) {
        if(p.empty()){
            p.push_back({val,val});
        }
        else {
            p.push_back({val,min(p.back().second,val)});
        }
    }

    void pop() {
     p.pop_back();
    }

    int top() {
        return p.back().first;
    }

    int getMin() {
        return p.back().second;
    }
};

```

150. Evaluate Reverse Polish Notation

Medium

You are given an array of strings `tokens` that represents an arithmetic expression in a Reverse Polish Notation.

Evaluate the expression. Return *an integer that represents the value of the expression*.

Note that:

- The valid operators are `'+'`, `'-'`, `'*'`, and `'/'`.
- Each operand may be an integer or another expression.
- The division between two integers always **truncates toward zero**.
- There will not be any division by zero.
- The input represents a valid arithmetic expression in a reverse polish notation.
- The answer and all the intermediate calculations can be represented in a **32-bit** integer.

Example 1:

```
Input: tokens = ["2","1","+","3","*"]
Output: 9
Explanation: ((2 + 1) * 3) = 9
```

Example 2:

```
Input: tokens = ["4","13","5","/","+"]
Output: 6
Explanation: (4 + (13 / 5)) = 6
```

Example 3:

```
Input: tokens = ["10","6","9","3","+","-11","*","/","*","17","+","5","+"]
Output: 22
Explanation: ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
```

```cpp
class Solution {
public:
    //lol RPN is same as evaluating a postfix notation
    int evalRPN(vector<string>& tokens) {
        stack<int>st;
        for(auto i:tokens){
            if(i=="+"||i=="-"||i=="*"||i=="/"){
                int p1=st.top();
                st.pop();
                int p2=st.top();
                st.pop();
                if(i=="+")p1=p1+p2;
                if(i=="-")p1=p2-p1;
                if(i=="*")p1=p1*p2;
                if(i=="/")p1=p2/p1;
                st.push(p1);
            }
            else st.push(stoi(i));
        }
    return st.top();}
};
```

2216. Minimum Deletions to Make Array Beautiful

Medium

You are given a **0-indexed** integer array `nums`. The array `nums` is **beautiful** if:

- `nums.length` is even.
- `nums[i] != nums[i + 1]` for all `i % 2 == 0`.

Note that an empty array is considered beautiful.

You can delete any number of elements from `nums`. When you delete an element, all the elements to the right of the deleted element will be **shifted one unit to the left** to fill the gap created and all the elements to the left of the deleted element will remain **unchanged**.

Return *the **minimum** number of elements to delete from* `nums` *to make it beautiful.*

Example 1:

```
Input: nums = [1,1,2,3,5]
Output: 1
Explanation: You can delete either nums[0] or nums[1] to make nums = [1,2,3,5] which is beautiful. It can be proven you need at least
```

Example 2:

```
Input: nums = [1,1,2,2,3,3]
Output: 2
Explanation: You can delete nums[0] and nums[5] to make nums = [1,2,2,3] which is beautiful. It can be proven you need at least 2 del
```

Constraints:

- `1 <= nums.length <= 10^5`
- `0 <= nums[i] <= 10^5`

```cpp
class Solution {
public:
    // int minDeletion(vector<int>& nums) {//totally wrong approach for the problem ..lol
    //     multiset<pair<int,int>>s;
    //     for(int i=0;i<nums.size();i++){
    //         s.insert({i,nums[i]});
    //     }
    //     for(auto &[i,p]:s){
    //         if(i%)
    //     }
    // }
    int minDeletion(vector<int>& nums){// pretty easy question just that the writings are made overcomplicated
        // here we keep track of the number of deletions to be made , which would only be when nums[i]==nums[i+1]
        if(size(nums)==1)return 1;
        int del=0;
        for(int i=0;i-del+1<size(nums);i+=2){
            del+=(nums[i-del]==nums[i-del+1]);
        }
        return (size(nums)-del)%2+del;
            //then in the end we also gotta check if the final array
//size is even or not , if not reduce one element from the end and make it even
        // U see it actually means u gotta calculate the number of deletions to be done so that u can have an even sized distinct array ..
    }
};
```

316. Remove Duplicate Letters

Medium

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

Example 1:

```
Input: s = "bcabc"
Output: "abc"
```

Example 2:

```
Input: s = "cbacdcbc"
Output: "acdb"
```

Constraints:

- `1 <= s.length <= 10^4`
- `s` consists of lowercase English letters.

Note: This question is the same as 1081: https://leetcode.com/problems/smallest-subsequence-of-distinct-characters/

```cpp
class Solution {
public:
    string removeDuplicateLetters(string s) {
        vector<int> lastIndex(26, 0);
        for (int i = 0; i < s.length(); i++){
            lastIndex[s[i] - 'a'] = i; // track the lastIndex of character presence
        }

        vector<bool> seen(26, false); // keep track seen
        stack<char> st;

        for (int i = 0; i < s.size(); i++) {
            int curr = s[i] - 'a';
            if (seen[curr]) continue; // if seen continue as we need to pick one char only
            while(st.size() > 0 && st.top() > s[i] && i < lastIndex[st.top() - 'a']){
                seen[st.top() - 'a'] = false; // pop out and mark unseen
                st.pop();
            }
            st.push(s[i]); // add into stack
            seen[curr] = true; // mark seen
        }

        string ans = "";
        while (st.size() > 0){
            ans += st.top();
            st.pop();
        }
        reverse(ans.begin(), ans.end());
        return ans;
    }
};
```

735. Asteroid Collision

Medium

We are given an array `asteroids` of integers representing asteroids in a row.

For each asteroid, the absolute value represents its size, and the sign represents its direction (positive meaning right, negative meaning left). Each asteroid moves at the same speed.

Find out the state of the asteroids after all collisions. If two asteroids meet, the smaller one will explode. If both are the same size, both will explode. Two asteroids moving in the same direction will never meet.

Example 1:

```
Input: asteroids = [5,10,-5]
Output: [5,10]
Explanation: The 10 and -5 collide resulting in 10. The 5 and 10 never collide.
```

Example 2:

```
Input: asteroids = [8,-8]
Output: []
Explanation: The 8 and -8 collide exploding each other.
```

Example 3:

```
Input: asteroids = [10,2,-5]
Output: [10]
Explanation: The 2 and -5 collide resulting in -5. The 10 and -5 collide resulting in 10.
```

Constraints:

- `2 <= asteroids.length <= 10^4`
- `-1000 <= asteroids[i] <= 1000`

```cpp
class Solution {
public:
//     very likely a stack problem..
//
    // vector<int> asteroidCollision(vector<int>& arr) {
    //     int n=arr.size();
    //     vector<int>s;
    //     for(int i=0;i<n;i++){
    //         if(s.size()==0||(s.back()<0&&arr[i]<0)||(s.back()>0&&arr[i]>0))s.push_back(arr[i]);
    //         else {
    //             while(s.size()!=0&&s.back()<0&&arr[i]>0||s.back()>0&&arr[i]<0){
    //                 if(abs(s.back())==abs(arr[i]))s.pop_back();
    //                 else if(abs(s.back())<abs(arr[i])){
    //                     s.pop_back();
    //                     s.push_back(arr[i]);
    //                 }
    //                 else break;
    //             }
    //         }
    //     }
    // return s;}
            vector<int> asteroidCollision(vector<int>& asteroids) {
                    vector<int> res;
                    for(int a : asteroids) {
                        bool destroyed = false;
                        while(res.size() && res.back() > 0 && a < 0 && !destroyed) {
                                if(res.back() >= -a) destroyed = true;
                                if(res.back() <= -a) res.pop_back();
                        }
                        if(!destroyed) res.push_back(a);
                    }
                        return res;
                }
};
```

1996. The Number of Weak Characters in the Game

Medium

You are playing a game that contains multiple characters, and each of the characters has **two** main properties: **attack** and **defense**. You are given a 2D integer array `properties` where `properties[i] = [attack_i, defense_i]` represents the properties of the i^{th} character in the game.

A character is said to be **weak** if any other character has **both** attack and defense levels **strictly greater** than this character's attack and defense levels. More formally, a character `i` is said to be **weak** if there exists another character `j` where $attack_j > attack_i$ and $defense_j > defense_i$.

Return *the number of **weak** characters.*

Example 1:

```
Input: properties = [[5,5],[6,3],[3,6]]
Output: 0
Explanation: No character has strictly greater attack and defense than the other.
```

Example 2:

```
Input: properties = [[2,2],[3,3]]
Output: 1
Explanation: The first character is weak because the second character has a strictly greater attack and defense.
```

Example 3:

```
Input: properties = [[1,5],[10,4],[4,3]]
Output: 1
Explanation: The third character is weak because the second character has a strictly greater attack and defense.
```

Constraints:

- $2 <= properties.length <= 10^5$
- `properties[i].length == 2`

```cpp
class Solution {
public:
    int numberOfWeakCharacters(vector<vector<int>>& properties) {
       sort(properties.begin(),properties.end(),[] (vector<int>& a,vector<int>& b){
           if(a[0]==b[0])return a[1]>b[1];
           return a[0]<b[0];
       });
        int ans=0,var=INT_MIN;
        for(int i=properties.size()-1;i>=0;i--){
            if(properties[i][1]<var)ans++;
            var=max(properties[i][1],var);
        }
    return ans;}
};
/*what were doing here is that we should sort the first parameter and then check for the no.of second parameter greater than the recent answer.*/
```

71. Simplify Path

Medium

Given a string `path`, which is an **absolute path** (starting with a slash `'/'`) to a file or directory in a Unix-style file system, convert it to the simplified **canonical path**.

In a Unix-style file system, a period `'.'` refers to the current directory, a double period `'..'` refers to the directory up a level, and any multiple consecutive slashes (i.e. `'//'`) are treated as a single slash `'/'`. For this problem, any other format of periods such as `'...'` are treated as file/directory names.

The **canonical path** should have the following format:

- The path starts with a single slash `'/'`.
- Any two directories are separated by a single slash `'/'`.
- The path does not end with a trailing `'/'`.
- The path only contains the directories on the path from the root directory to the target file or directory (i.e., no period `'.'` or double period `'..'`)

Return *the simplified* ***canonical path***.

Example 1:

```
Input: path = "/home/"
Output: "/home"
Explanation: Note that there is no trailing slash after the last directory name.
```

Example 2:

```
Input: path = "/../"
Output: "/"
Explanation: Going one level up from the root directory is a no-op, as the root level is the highest level you can go.
```

Example 3:

```cpp
class Solution {
public:
    string simplifyPath(string path) {
        vector<string>stack;
        for(int i=0;i<path.size();i++){
            if(path[i]=='/')continue;
            string temp;
            while(i<path.size()&&path[i]!='/'){temp+=path[i];i++;}
            if(temp==".")continue;
            else if (temp==".."){
                if(!stack.empty()){stack.pop_back();}
            }
            else stack.push_back(temp);
        }
        if(stack.size()==0)return "/";
        string res="";
        while(!stack.empty()){
            res="/"+stack.back()+res;
            stack.pop_back();
        }
        return res;
    }
};
```

456. 132 Pattern

Medium

Given an array of `n` integers `nums`, a **132 pattern** is a subsequence of three integers `nums[i]`, `nums[j]` and `nums[k]` such that `i < j < k` and `nums[i] < nums[k] < nums[j]`.

Return `true` *if there is a* ***132 pattern*** *in* `nums`, *otherwise, return* `false`.

Example 1:

```
Input: nums = [1,2,3,4]
Output: false
Explanation: There is no 132 pattern in the sequence.
```

Example 2:

```
Input: nums = [3,1,4,2]
Output: true
Explanation: There is a 132 pattern in the sequence: [1, 4, 2].
```

Example 3:

```
Input: nums = [-1,3,2,0]
Output: true
Explanation: There are three 132 patterns in the sequence: [-1, 3, 2], [-1, 3, 0] and [-1, 2, 0].
```

Constraints:

- `n == nums.length`
- `1 <= n <= 2 * 10^5`
- `-10^9 <= nums[i] <= 10^9`

```cpp
class Solution {
public:
// as for the tradition of writing brute-force
    // bool find132pattern(vector<int>& arr) {
    //     int n=arr.size();
    //     for(int i=1;i<n-1;i++){
    //         if(arr[i-1]<arr[i+1]&&arr[i+1]<arr[i])return true;
    //     }
    // return false;
    // }
// sorry,my bad understanding of the problem
    bool find132pattern(const std::vector<int>& nums) {
    int n = nums.size();
    if (n < 3) {
        return false;
    }

    std::stack<int> stk;
    int s3 = INT_MIN;

    for (int i = n - 1; i >= 0; i--) {
        if (nums[i] < s3) {
            return true;
        } else {
            while (!stk.empty() && nums[i] > stk.top()) {
                s3 = stk.top();
                stk.pop();
            }
        }
        stk.push(nums[i]);
    }

    return false;
}
};
```

402. Remove K Digits

Medium

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

Example 1:

```
Input: num = "1432219", k = 3
Output: "1219"
Explanation: Remove the three digits 4, 3, and 2 to form the new number 1219 which is the smallest.
```

Example 2:

```
Input: num = "10200", k = 1
Output: "200"
Explanation: Remove the leading 1 and the number is 200. Note that the output must not contain leading zeroes.
```

Example 3:

```
Input: num = "10", k = 2
Output: "0"
Explanation: Remove all the digits from the number and it is left with nothing which is 0.
```

Constraints:

- `1 <= k <= num.length <= 10^5`
- `num` consists of only digits.
- `num` does not have any leading zeros except for the zero itself.

```cpp
class Solution {
public:
    string removeKdigits(string num, int k) {
        string res="";

        for(auto c : num){                                          //iterating over the whole string
                            while(res.length() && res.back()>c && k){  //while k>0 and there's still characters in res and last char is greater than current char
                res.pop_back();
                k--;
            }

            if(res.length() || c!='0')        //to ensure that res doesnt have leading 0's
                res.push_back(c);
        }

        while(res.length() && k){             //if k is still greater than 0, delete characters from back
            k--;
            res.pop_back();
        }

        return res.empty()? "0" : res;         //returning solution
    }
};
```

936. Stamping The Sequence

Hard

You are given two strings `stamp` and `target`. Initially, there is a string `s` of length `target.length` with all `s[i] == '?'`.

In one turn, you can place `stamp` over `s` and replace every letter in the `s` with the corresponding letter from `stamp`.

- For example, if `stamp = "abc"` and `target = "abcba"`, then `s` is `"?????"` initially. In one turn you can:

```
<ul>
	<li>place <code>stamp</code> at index <code>0</code> of <code>s</code> to obtain <code>"abc??"</code>,</li>
	<li>place <code>stamp</code> at index <code>1</code> of <code>s</code> to obtain <code>"?abc?"</code>, or</li>
	<li>place <code>stamp</code> at index <code>2</code> of <code>s</code> to obtain <code>"??abc"</code>.</li>
</ul>
Note that <code>stamp</code> must be fully contained in the boundaries of <code>s</code> in order to stamp (i.e., you canno
```

We want to convert `s` to `target` using **at most** `10 * target.length` turns.

Return *an array of the index of the left-most letter being stamped at each turn*. If we cannot obtain `target` from `s` within `10 * target.length` turns, return an empty array.

Example 1:

```
Input: stamp = "abc", target = "ababc"
Output: [0,2]
Explanation: Initially s = "?????".
- Place stamp at index 0 to get "abc??".
- Place stamp at index 2 to get "ababc".
[1,0,2] would also be accepted as an answer, as well as some other answers.
```

Example 2:

```
Input: stamp = "abca", target = "aabcaca"
Output: [3,0,1]
Explanation: Initially s = "???????".
- Place stamp at index 3 to get "???abca".
```

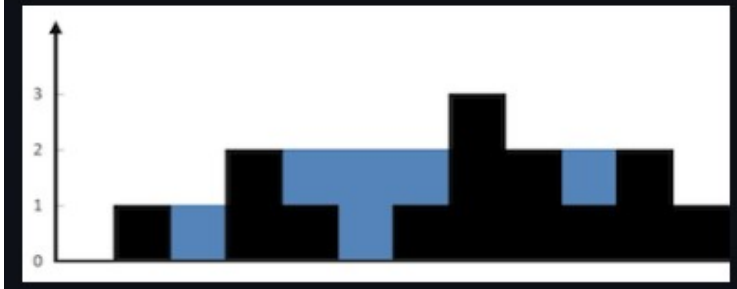
```cpp
class Solution {
public:
    vector<int> movesToStamp(string stamp, string target) {//this is votrubac's solution
      vector<int> res;
      auto total_stamp = 0, turn_stamp = -1;
      while (turn_stamp) {
          turn_stamp = 0;
          for (int sz = stamp.size(); sz > 0; --sz)
              for (auto i = 0; i <= stamp.size() - sz; ++i) {
                  auto new_stamp = string(i, '*') + stamp.substr(i, sz) + string(stamp.size() - sz - i, '*');
                  auto pos = target.find(new_stamp);
                  while (pos != string::npos) {
                      res.push_back(pos);
                      turn_stamp += sz;
                      fill(begin(target) + pos, begin(target) + pos + stamp.size(), '*');
                      pos = target.find(new_stamp);
                  }
              }
          total_stamp += turn_stamp;
      }
      reverse(begin(res), end(res));
      return total_stamp == target.size() ? res : vector<int>();
    }
};
```

42. Trapping Rain Water

Hard

Given `n` non-negative integers representing an elevation map where the width of each bar is `1`, compute how much water it can trap after raining.

Example 1:




```
Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
Explanation: The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain
```

Example 2:

```
Input: height = [4,2,0,3,2,5]
Output: 9
```

Constraints:

- `n == height.length`
- `1 <= n <= 2 * 10^4`
- `0 <= height[i] <= 10^5`

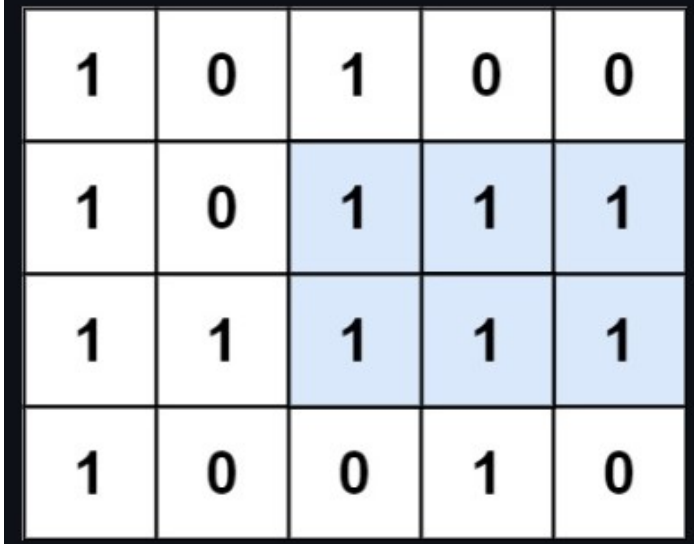
```cpp
class Solution {
public:
    int trap(vector<int>& height) {//two pointer approach
             int left_max = INT_MIN;
        int right_max = INT_MIN;

        int r = height.size()-1;
        int l = 0;

        int ans = 0;
        while(l<=r){
            if(left_max <= right_max){
                if(left_max <= height[l]){
                    left_max = height[l++];
                    continue;
                }
                ans += (left_max - height[l++]);
            }else{
                if(right_max <= height[r]){
                    right_max = height[r--];
                    continue;
                }
                ans += (right_max - height[r--]);
            }
        }
        return ans;
       /*
       int trap(vector<int>& height)//stacks O(n)solution
{
    int ans = 0, current = 0;
    stack<int> st;
    while (current < height.size()) {
        while (!st.empty() && height[current] > height[st.top()]) {
            int top = st.top();
            st.pop();
            if (st.empty())
                break;
            int distance = current - st.top() - 1;
            int bounded_height = min(height[current], height[st.top()]) - height[top];
            ans += distance * bounded_height;
        }
        st.push(current++);
    }
    return ans;
    */
```

85. Maximal Rectangle

Hard

Given a `rows x cols` binary `matrix` filled with `0`'s and `1`'s, find the largest rectangle containing only `1`'s and return *its area*.

Example 1:




```
Input: matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]
Output: 6
Explanation: The maximal rectangle is shown in the above picture.
```

Example 2:

```
Input: matrix = [["0"]]
Output: 0
```

Example 3:

```
Input: matrix = [["1"]]
```

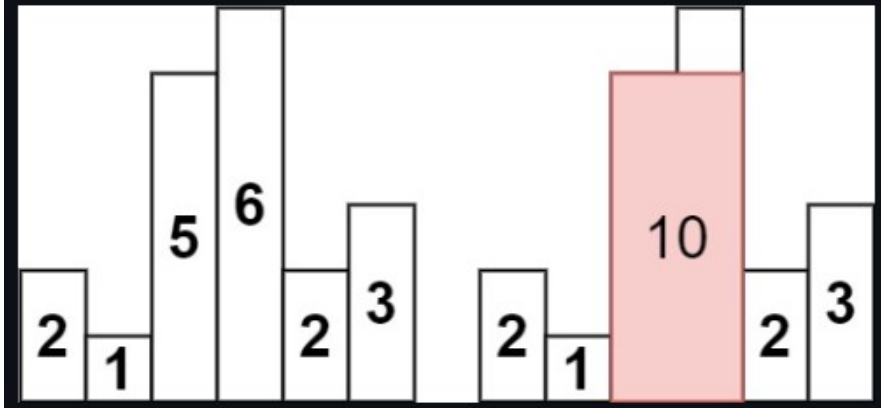
```cpp
class Solution {
public:
//     hooked up the solution of largest rectangle area
    int largestRectangleArea(vector < int > & histo) {
            stack < int > st;
            int maxA = 0;
            int n = histo.size();
            for (int i = 0; i <= n; i++) {
                while (!st.empty() && (i == n || histo[st.top()] >= histo[i])) {
                int height = histo[st.top()];
                st.pop();
                int width;
                if (st.empty())
                    width = i;
                else
                    width = i - st.top() - 1;
                maxA = max(maxA, width * height);
                }
                st.push(i);
            }
            return maxA;
            }
            int maximalRectangle(vector<vector<char>>& matrix) {
                int maxiArea = 0, n = matrix.size(), m = matrix[0].size();
                vector<int> height(m,0);

                for(int i=0;i<n;i++){
                    for(int j=0;j<m;j++){
                        if(matrix[i][j] == '1') height[j]++;
                        else height[j] = 0;
                    }
                    int area = largestRectangleArea(height);
                    maxiArea = max(maxiArea,area);
                }
                return maxiArea;
            }
};
```

84. Largest Rectangle in Histogram

Hard

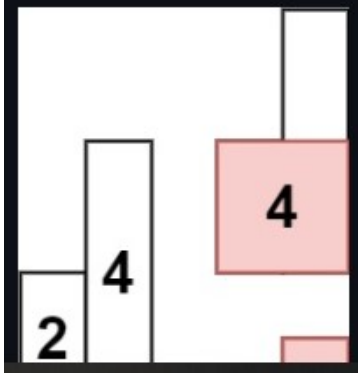
Given an array of integers `heights` representing the histogram's bar height where the width of each bar is `1`, return *the area of the largest rectangle in the histogram.*

Example 1:




```
Input: heights = [2,1,5,6,2,3]
Output: 10
Explanation: The above is a histogram where width of each bar is 1.
The largest rectangle is shown in the red area, which has an area = 10 units.
```

Example 2:



```cpp
class Solution {
public:
    int largestRectangleArea(vector<int>& height) {
        // a very interesting problem here , so will be using a monotonic stack to solve the problem
        vector<pair<int,int>>s;//we'll be storing (start index , height) of each block of histogram.
        int res=0,n=size(height);
        for(int i=0;i<n;i++){
            int start=i;
            while(s.size()!=0&&s.back().second>height[i]){
                int id=s.back().first,h=s.back().second;
                s.pop_back();
                res=max(res,h*(i-id));//h is the height that it has extended , till i from start, which is stored in the stack.
                start=id;
            }
            s.push_back({start,height[i]});
        }
        //cout<<res<<" ";
        for(int i=0;i<s.size();i++){//check the remaining values in the stack, these will be the ones that can be extended till the end. so area would be height*(end -start)
            int h=s[i].second;
            res=max(res,h*(n-s[i].first));
        }
    return res;}
};
```

224. Basic Calculator

Hard

Given a string `s` representing a valid expression, implement a basic calculator to evaluate it, and return *the result of the evaluation*.

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

Example 1:

```
Input: s = "1 + 1"
Output: 2
```

Example 2:

```
Input: s = " 2-1 + 2 "
Output: 3
```

Example 3:

```
Input: s = "(1+(4+5+2)-3)+(6+8)"
Output: 23
```

Constraints:

- `1 <= s.length <= 3 * 10^5`
- `s` consists of digits, `'+'`, `'-'`, `'('`, `')'`, and `' '`.
- `s` represents a valid expression.
- `'+'` is **not** used as a unary operation (i.e., `"+1"` and `"+(2 + 3)"` is invalid).
- `'-'` could be used as a unary operation (i.e., `"-1"` and `"-(2 + 3)"` is valid).
- There will be no two consecutive operators in the input.
- Every number and running calculation will fit in a signed 32-bit integer.

```cpp
#define LL long long
class Solution {
public:

    int solve(string s){
        LL sum=0;
        int sign=1;
        int n=s.size();
        stack<pair<int,int>>st;
        for(int i=0;i<n;i++){
            char c=s[i];
            if(isdigit(c)){
                LL num=0;
                while(i<n&&isdigit(s[i])){
                    num=((num*10)-'0'+s[i]);
                    i++;
                }
                i--;
                sum+=(sign*num);
                sign=1;
            }
            else if(c=='('){
                st.push({sum,sign});// save previous state.
                sum=0;
                sign=1;
            }
            else if(c==')'){
                sum=(st.top().first)+(st.top().second*sum);
                // sum=0;
                // sign=1;
                st.pop();
            }
            else if(c=='-')sign*=-1;
        }
    return sum;
    }
    int calculate(string s) {

        return solve(s);
    }
};
```

2589. Minimum Time to Complete All Tasks

Hard

There is a computer that can run an unlimited number of tasks **at the same time**. You are given a 2D integer array `tasks` where `tasks[i] = [start_i, end_i, duration_i]` indicates that the `i^th` task should run for a total of `duration_i` seconds (not necessarily continuous) within the **inclusive** time range `[start_i, end_i]`.

You may turn on the computer only when it needs to run a task. You can also turn it off if it is idle.

Return *the minimum time during which the computer should be turned on to complete all tasks.*

Example 1:

```
Input: tasks = [[2,3,1],[4,5,1],[1,5,2]]
Output: 2
Explanation:
- The first task can be run in the inclusive time range [2, 2].
- The second task can be run in the inclusive time range [5, 5].
- The third task can be run in the two inclusive time ranges [2, 2] and [5, 5].
The computer will be on for a total of 2 seconds.
```

Example 2:

```
Input: tasks = [[1,3,2],[2,5,3],[5,6,2]]
Output: 4
Explanation:
- The first task can be run in the inclusive time range [2, 3].
- The second task can be run in the inclusive time ranges [2, 3] and [5, 5].
- The third task can be run in the two inclusive time range [5, 6].
The computer will be on for a total of 4 seconds.
```

Constraints:

- `1 <= tasks.length <= 2000`

```cpp
//     //mycurrent approach would be to sort the tasks based on their end time, and then maybe chech which ones
//     int findMinimumTime(vector<vector<int>>& tasks) {
//         sort(tasks.begin(),tasks.end(),[](const auto &e1,const auto &e2){
//             return e1[1]==e2[2]?e1[0]<e2[0]:e1[1]<e2[1];
//         });
//         unordered_map<int,bool>mp;//this stores which time is used
//         int count =0;
//         for(int i=0;i<tasks.size();i++){
//             int s=tasks[i][0],e=tasks[i][1],d=tasks[i][2];
//             int used=0;
//             for(int j=s;j<=e;j++) {//this is to utilise the already used time in the ith task , so that we dont use more time for the ith task than needed
//                 if(mp.count(j))used++;
//             }
//             for(int t=e;used<d;t--){//we start assigning tasks from the end inorder to use it efficiently
//                 if(!mp.count(t)){ //here we need to take account only if there is a new time slice that wasn't used by the previous users
//                     used++;
//                     mp[t]=1;//we store that this time slice has been used
//                     count++;
//                 }
//             }
//         }
//     return count;}


//     /// damnn i liked this problem for some reason and it was pretty challenging
// };
class Solution {
public:
    int findMinimumTime(vector<vector<int>>& tasks) {
        sort(tasks.begin(),tasks.end(),[](vector<int>& a,vector<int>& b){
            return (a[1]!=b[1])?a[1]<b[1]:a[0]<b[0];
        });
        unordered_map<int,bool> used;       // time_slice... used or not ...

        int count=0;
        for(auto& task:tasks){
            int usedTime=0;
            for(int t=task[0];t<=task[1];t++)   if(used.count(t))   usedTime++;      // have used this time slice

            for(int t=task[1];usedTime<task[2];t--){
                if(!used.count(t)){         // this time slice is available
                    usedTime++;             // duration for which I run it
                    used[t]=1;              // use this timeslice
                    count++;                // time for which my computer is on
                }
            }
        }
        return count;
    }
```

32. Longest Valid Parentheses

Hard

Given a string containing just the characters `'('` and `')'`, return *the length of the longest valid (well-formed) parentheses substring*.

Example 1:

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

Example 2:

```
Input: s = ")()())"
Output: 4
Explanation: The longest valid parentheses substring is "()()".
```

Example 3:

```
Input: s = ""
Output: 0
```

Constraints:

- `0 <= s.length <= 3 * 10^4`
- `s[i]` is `'('`, or `')'`.

```cpp
class Solution {
public:

    int longestValidParentheses(string s) {//o(n) dp  solution
        s = ")" + s;
        int curMax = 0;
        vector<int> longest(s.size(),0);
        for(int i=1; i < s.length(); i++){
            if(s[i] == ')' && s[i-longest[i-1]-1] == '('){
                    longest[i] = longest[i-1] + 2 + longest[i-longest[i-1]-2];
                    curMax = max(longest[i],curMax);
            }
        }
        return curMax;
    }
};
```

20. Valid Parentheses

Easy

Given a string `s` containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

Example 1:

```
Input: s = "()"
Output: true
```

Example 2:

```
Input: s = "()[]{}"
Output: true
```

Example 3:

```
Input: s = "(]"
Output: false
```

Constraints:

- `1 <= s.length <= 10^4`
- `s` consists of parentheses only `'()[]{}'`.

```cpp

class Solution {
public:
bool check(char a,char b){
    if(a=='('&&b==')'||a=='['&&b==']'||a=='{'&&b=='}')return true;
    return false;
}
   bool isValid(string s) {//we'll be using stack to track the parenthesis in the  string
   stack<char>st;
       for(int i=0;i<s.length();i++){
           if(s[i]=='('||s[i]=='{'||s[i]=='[')st.push(s[i]);
           else if(st.empty()||!check(st.top(),s[i]))return false;
           else st.pop();
       }
    return st.empty();
   }
};
```

844. Backspace String Compare

Easy

Given two strings `s` and `t`, return `true` *if they are equal when both are typed into empty text editors.* `'#'` means a backspace character.

Note that after backspacing an empty text, the text will continue empty.

Example 1:

```
Input: s = "ab#c", t = "ad#c"
Output: true
Explanation: Both s and t become "ac".
```

Example 2:

```
Input: s = "ab##", t = "c#d#"
Output: true
Explanation: Both s and t become "".
```

Example 3:

```
Input: s = "a#c", t = "b"
Output: false
Explanation: s becomes "c" while t becomes "b".
```

Constraints:

- `1 <= s.length, t.length <= 200`
- `s` and `t` only contain lowercase letters and `'#'` characters.

Follow up: Can you solve it in `O(n)` time and `O(1)` space?

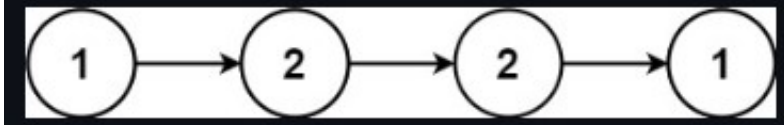
```cpp
class Solution {
public:
    bool backspaceCompare(string s, string t) {
        string s1,s2;
        for(int i=0;i<s.length();i++){
            if(s[i]=='#'){
                if(s1.length()>0)s1.pop_back();
            }
            else s1.push_back(s[i]);
        }
        for(int i=0;i<t.length();i++){
            if(t[i]=='#'){
                if(s2.length()>0)s2.pop_back();
            }
            else s2.push_back(t[i]);
        }

    return s1==s2;
    }
};
```

234. Palindrome Linked List

Easy

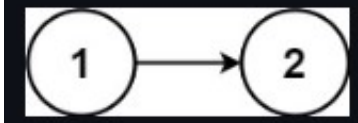
Given the `head` of a singly linked list, return `true` *if it is a palindrome or* `false` *otherwise.*

Example 1:




```
Input: head = [1,2,2,1]
Output: true
```

Example 2:




```
Input: head = [1,2]
Output: false
```

Constraints:

- The number of nodes in the list is in the range `[1, 10^5]`.
- `0 <= Node.val <= 9`

Follow up: Could you do it in `O(n)` time and `O(1)` space?

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* getmid(ListNode*head){
        ListNode*slow=head,*fast=head;
        while(fast && fast->next){
            slow=slow->next;
            fast=fast->next->next;
        }
        return slow;
    }
    ListNode* reversal(ListNode*head){
        ListNode* prev=NULL,*curr=head,*temp=NULL;
        while(curr != NULL){
            temp=curr->next;
            curr->next=prev;
            prev=curr;
            curr=temp;
        }
        return prev;
    }
    bool comparelist(ListNode*head1,ListNode*head2){
        while(head1 && head2){
            if(head1->val!=head2->val)return false;
            head1=head1->next;
            head2=head2->next;
        }
        return true;
    }
    bool isPalindrome(ListNode* head) {
        ListNode *h1=head,*h2,*mid;
        mid=getmid(head);
        h2=reversal(mid);
        return comparelist(h2,h1);
    }
};
```

232. Implement Queue using Stacks

Easy

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (`push`, `peek`, `pop`, and `empty`).

Implement the `MyQueue` class:

- `void push(int x)` Pushes element x to the back of the queue.
- `int pop()` Removes the element from the front of the queue and returns it.
- `int peek()` Returns the element at the front of the queue.
- `boolean empty()` Returns `true` if the queue is empty, `false` otherwise.

Notes:

- You must use **only** standard operations of a stack, which means only `push to top`, `peek/pop from top`, `size`, and `is empty` operations are valid.
- Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

Example 1:

```
Input
["MyQueue", "push", "push", "peek", "pop", "empty"]
[[], [1], [2], [], [], []]
Output
[null, null, null, 1, 1, false]

Explanation
MyQueue myQueue = new MyQueue();
myQueue.push(1); // queue is: [1]
myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)
myQueue.peek(); // return 1
myQueue.pop(); // return 1, queue is [2]
myQueue.empty(); // return false
```

Constraints:

- `1 <= x <= 9`
- At most `100` calls will be made to `push`, `pop`, `peek`, and `empty`.

```cpp
class MyQueue {//amazing easy question ,blown!
public:
        stack<int>arr1;
        stack<int>arr2;
    MyQueue() {
    }

    void push(int x) {
        while(!arr1.empty()){
            arr2.push(arr1.top());
            arr1.pop();
        }
        arr1.push(x);
        while(!arr2.empty()){
            arr1.push(arr2.top());
            arr2.pop();
        }
    }

    int pop() {
        int r=arr1.top();
        arr1.pop();
        return r;
    }

    int peek() {
        return arr1.top();
    }

    bool empty() {
        return arr1.empty();
        //return arr1.size()==0?true:false;
    }

};
```

2696. Minimum String Length After Removing Substrings

Easy

You are given a string `s` consisting only of **uppercase** English letters.

You can apply some operations to this string where, in one operation, you can remove **any** occurrence of one of the substrings `"AB"` or `"CD"` from `s`.

Return *the **minimum** possible length of the resulting string that you can obtain.*

Note that the string concatenates after removing the substring and could produce new `"AB"` or `"CD"` substrings.

Example 1:

```
Input: s = "ABFCACDB"
Output: 2
Explanation: We can do the following operations:
- Remove the substring "ABFCACDB", so s = "FCACDB".
- Remove the substring "FCACDB", so s = "FCAB".
- Remove the substring "FCAB", so s = "FC".
So the resulting length of the string is 2.
It can be shown that it is the minimum length that we can obtain.
```

Example 2:

```
Input: s = "ACBBD"
Output: 5
Explanation: We cannot do any operations on the string so the length remains the same.
```

Constraints:

- `1 <= s.length <= 100`
- `s` consists only of uppercase English letters.

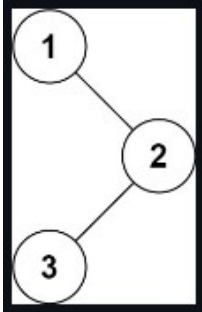
```cpp
class Solution {
public:
    int minLength(string s) {
        // unordered_map<int,int>m;
        // int n=s.length();
        // while(s.find("AB")!=string::npos||s.find("AB")!=string::npos){
        //     auto iA=s.find("AB");
        //     auto iC=s.find("CD");
        //     if(iA<iC)s.erase(iA,2);
        //     else s.erase(iC,2);
        // }
        // // cout<<n<<endl;
        // // cout<<m[0]<<" "<<m[1]<<"\n";
        // // int res=n-(m[0]*2+m[1]*2);
        // return s.length();
        stack<char>st;
        int n=s.length();
        for(int i=0;i<n;i++){
            char c=s[i];
            if(st.empty())st.push(c);
            else {
                int last=st.top();
                if(c=='B'||c=='D'){
                    if(c=='B'){
                        if(last=='A')st.pop();
                        else st.push(c);
                    }
                    else if(c=='D'){
                        if(last=='C')st.pop();
                        else st.push(c);
                    }
                }
                else st.push(c);
            }
        }
        return st.size();
    }
};
```

144. Binary Tree Preorder Traversal

Easy

Given the `root` of a binary tree, return *the preorder traversal of its nodes' values.*

Example 1:




```
Input: root = [1,null,2,3]
Output: [1,2,3]
```

Example 2:

```
Input: root = []
Output: []
```

Example 3:

```
Input: root = [1]
Output: [1]
```

Constraints:

- The number of nodes in the tree is in the range `[0, 100]`.
- `-100 <= Node.val <= 100`

Follow up: Recursive solution is trivial, could you do it iteratively?

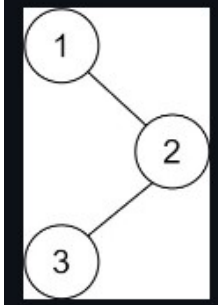
```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    vector<int>res;
    void func(TreeNode* root){
        if(!root)return;
        else{
            res.push_back(root->val);
            func(root->left);
            func(root->right);
        }
    }
    vector<int> preorderTraversal(TreeNode* root) {
        func(root);
        return res;}
};
```

145. Binary Tree Postorder Traversal

Easy

Given the `root` of a binary tree, return *the postorder traversal of its nodes' values*.

Example 1:




```
Input: root = [1,null,2,3]
Output: [3,2,1]
```

Example 2:

```
Input: root = []
Output: []
```

Example 3:

```
Input: root = [1]
Output: [1]
```

Constraints:

- The number of the nodes in the tree is in the range `[0, 100]`.
- `-100 <= Node.val <= 100`

Follow up: Recursive solution is trivial, could you do it iteratively?

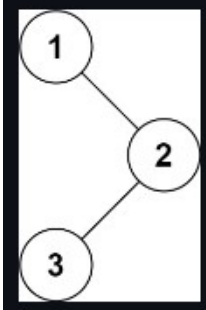
```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
private:
        vector<int>res;
    void post(TreeNode*root){
        if(root->left!=NULL)post(root->left);
        if(root->right!=NULL)post(root->right);
        res.push_back(root->val);
    }
public:
    vector<int> postorderTraversal(TreeNode* root) {
        if(!root)return {};
        post(root);
        return res;
    }
};
```

94. Binary Tree Inorder Traversal

Easy

Given the `root` of a binary tree, return *the inorder traversal of its nodes' values.*

Example 1:




```
Input: root = [1,null,2,3]
Output: [1,3,2]
```

Example 2:

```
Input: root = []
Output: []
```

Example 3:

```
Input: root = [1]
Output: [1]
```

Constraints:

- The number of nodes in the tree is in the range `[0, 100]`.
- `-100 <= Node.val <= 100`

Follow up: Recursive solution is trivial, could you do it iteratively?

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
   /* vector<int> inorderTraversal(TreeNode* root) {//normal recursive solution
        vector<int>res;
        inorder(root,res);
        return res;
    }
    void inorder(TreeNode* root,vector<int>&res){
        if(root!=NULL){
            inorder(root->left,res);
            res.push_back(root->val);
            inorder(root->right,res);
        }
    }*/
    //iterative solution using stack
    vector<int> inorderTraversal(TreeNode* root){
        vector<int>res;
        TreeNode* curr=root;
        stack<TreeNode*>st;
        while(curr!=NULL or !st.empty()){
            while(curr!=NULL){
                st.push(curr);
               curr=curr->left;
            }
            curr=st.top();
            st.pop();
            res.push_back(curr->val);
            curr=curr->right;
        }
        return res;
    }
};
```