Minimum window substr (cont.)

```
    right +=1
return "" if ans[0]== float("inf") else
    s[ans[1]: ans[2]+1]
```

Group anagrams together

```
def group-anagrams(strs):
    res= defaultdict(list)
    for s in strs:
        count=[0]*26
        for c in s:
            count[ord(c)-ord("a")] +=1
        res[tuple(count)].append(s)
    return res.values()
```

Valid parentheses

```
def isValid(s):
    stack=[]
    bracket-map={")": "(", "]": "["}
    for c in s:
        if c in bracket-map:
            if stack & stack[-1] == bracket_map[c]:
                stack.pop()
            else: return False
        else:
            stack.append(c)
    return True if not stack else False
```

Valid palindrome

```
def isPalindrome(s):
    l,r = 0, len(s)-1
    while l<r:
        while l<r & not s[l].isalnum():
            l+=1
        while r>l & not s[r].isalnum():
            r-=1
        if s[l].lower()!= s[r].lower():
            return False
        l, r = l+1, r-1
    return True
```

longest palindromic substr

```
def longestPalindrSubstr(s):
    def expandAroundCenter(left,right)
        while left>=0 &right <len(s) &
                s[left] == s[right]:
            left -=1
            right +=1
        return right-left-1
    if not s:
        return ""
    start,end=0,0
    for i in range(len(s)):
        len1=expandAroundCent(i, i)
        len2= expandAroundCent(i,i+1)
        max-len= max(len1, len2)
        if max-len >end-start:
            start = i-(max-len-1)//2
            end = i + max-len//2
    return s[start:end+1]
```

Number of palindromic substrs

```
def countSubstr(s):
    def expandAroundCent(left,right)
        count=0
        while left>=0 & right<len(s) &
                s[left] == s[right]:
            count +=1
            left -=1
            right +=1
        return count
    total-pal = 0
    for i in range(len(s)):
        total-pal += expandAroundCent(i,i)
        total-pal += expandAroundCent(i,i+1)
    return total-pal
```

Encode and Decode Strings (list of str to str & vice versa)

```
def encode(strs):
    res = ""
    for s in strs:
        res += str(len(s)) +"#"+s
    return res
def decode(str):
    res, i = [], 0
    while i< len(str):
        j=i
        while str[j]!='#':
            j +=1
        length= int(str[i:j])
        res.append(str[j+1:j+1+length])
        i = j+1 + length
    return res
```

TREE

Maximum depth of binary tree

```
from collections import deque
def maxDepth(root):
    if not root: return 0
    queue = deque([root])
    depth =0
    while queue:
        depth +=1
        cnt= len(queue)
        for i in range(cnt):
            node=queue.popleft()
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
    return depth
```

Same tree (binary tree)

```
from collections import deque
def isSame(p, q):
    queue= deque([(p,q)])
    while queue:
        node1,node2= queue.popleft()
        if not node1 and not node2:
            continue
        if not node1 or not node2
            or node1.val != node2.val:
            return False
        queue.append((node1.left,
                      node2.left))
~~return~~
        queue.append(node1.right,
                      node2.right))
    return True
```

Invert binary tree (swap left and right children)

```
def invert(root):
    if not root:
        return None
    queue = deque([root])
    while queue:
        node=queue.popleft()
        node.left, node.right=
            node.right, node.left
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
    return root
```

Binary tree max. path sum

```
def maxPathSum(root):
    res = [root.val]
    def dfs(root):
        if not root:
            return 0
        leftMax= dfs(root.left)
        rightMax= dfs(root.right)
        leftMax = max(leftMax,0)
        rightMax=max(rightMax,0)
        res[0]=max(res[0], root.val
            + leftMax + rightMax)
        return root.val + max(leftmax,
                              rightMax)
    dfs(root)
    return res[0]
```

Binary tree level order traversal

```
def levelorder(root):
    if not root:
        return []
    result=[]
    queue=deque([root])
    while queue:
        level=[]
        cnt= len(queue)
        for i in range(cnt):
            node= queue.popleft()
            level.append(node.val)
            if node.left:
                queue.append(node.
                    left)
            if node.right:
                queue.append(node.
                    right)
        result.append(level)
    retur result
```

Serialize and deserialize binary tree (encode tree as string & decode it)

```
def serialize(root):
    def helper(node):
        if not node:
            return ['null']
        left = helper(node.left)
        right= helper(node.right)
        return [str(node.val)]+
                left+right
    return ','.join(helper(root))
def deserialize(data):
    def helper(nodes):
        val =nodes.pop(0)
        if val == 'null':
            return None
        node= TreeNode(int(val))
        node.left=helper(nodes)
        node.right= helper(nodes)
        return node
    node_list = data.split(',')
    return helper(node_list)
```

Subtree of another tree (is t a subtree of s)

```
def isSubtree(s, t):
    def isSameTree(s,t):
        if not s & not t:
            return True
        if not s or not t or
                s.val != t.val:
            return False
        return isSameTree(s.left,
            t.left) and
            isSameTree(s.right,
                t.right)
    if not s:
        return False
    if isSameTree(s,t):
        return True
    return self.isSubtree(s.left,
        t) or self.isSubtree(s.right,
        t)
```

<u>Note</u>: Inorder traversal:
left tree → node — right tree
Preorder:
node → left tree → right tree
Postorder:
left tree → right tree → node

Construct Binary Tree from Preorder and Inorder Traversal