1.ReverseWordString:

+ revese full string, find words and reverse each of them

2.Maximum Size Subarray Sum Equals k

+ consider k as subtraction totalSum - <sum from 0 to i> where i - array index and totalSum is the sum of first i items.

+Save previous ith sum to map

3.Longest Palindromic Substring

+ use DP and consider even(i-1,i) and odd(i-1,i+1) substrings as polyndrom.

4.Product of Array Except Self

+ Use resulting array to calculation. First get product(i) = product(i-1)\*num(i-1) product(0) = 1. Then product(n-1) = right \* product(n-1) where right = 1 for n-1,then right = right \* num(i)

 5.Find the Duplicate Number

+ Sort the item and find same adjacent items.

6.Intersection of Two Linked Lists

class ListNode {  
 int val;  
 ListNode next;  
  
 ListNode(int x) {  
 val = x;  
 next = null;  
 }  
  
 ListNode(int x, ListNode n) {  
 this(x);  
 this.next = n;  
 }  
}  
  
public class IntersectionTwoLinkedLists {  
 private int lenOf(ListNode head) {  
 int len = 0;  
 while (head != null) {  
 len += 1;  
 head = head.next;  
 }  
 return len;  
 }  
  
 public ListNode getIntersectionNode(ListNode headA, ListNode headB) {  
 //boundary check  
 if (headA == null || headB == null) return null;  
  
 ListNode a = headA;  
 ListNode b = headB;  
  
 //if a & b have different len, then we will stop the loop after second iteration  
 while (a != b) {  
 //for the end of first iteration, we just reset the pointer to the head of another linkedlist  
 a = a == null ? headB : a.next;  
 b = b == null ? headA : b.next;  
 }  
 return a;  
 }  
}

- make while until list1 != list2 list1 = list1.next and list2 = list2.next,

If list1 == null list1 = head1 and if list2 == null list2 = head2 Floyd algorithm????? – fix it!!!

7.Symmetric Tree

Create method for x and y to compare x.left and y.right and x.right and y.left. To pass (root, root) as first parameters to this method.

8.Binary Tree Vertical Order Traversal

public List<List<Integer>> verticalOrder(TreeNode root) {

List<List<Integer>> res = new ArrayList<>();

if (root == null) {

return res;

}

Map<Integer, ArrayList<Integer>> map = new HashMap<>();

Queue<TreeNode> q = new LinkedList<>();

Queue<Integer> cols = new LinkedList<>();

q.add(root);

cols.add(0);

int min = 0;

int max = 0;

while (!q.isEmpty()) {

TreeNode node = q.poll();

int col = cols.poll();

if (!map.containsKey(col)) {

map.put(col, new ArrayList<Integer>());

}

map.get(col).add(node.val);

if (node.left != null) {

q.add(node.left);

cols.add(col - 1);

min = Math.min(min, col - 1);

}

if (node.right != null) {

q.add(node.right);

cols.add(col + 1);

max = Math.max(max, col + 1);

}

}

for (int i = min; i <= max; i++) {

res.add(map.get(i));

}

return res;

Graph Valid Tree:

Use union-find algorithm with rank and path-compression

9.Alien Language

**import** collections  
**class** Solution:  
 **def** alienOrder(self, words):  
 pre = collections.defaultdict(set)  
 suc = collections.defaultdict(set)  
  
 **for** pair **in** zip(words, words[1:]):  
 **for** a, b **in** zip(\*pair):  
 **if** a != b:  
 suc[a].add(b)  
 pre[b].add(a)  
 **break** chars = set(**''**.join(words))  
 *#get first char without predecesesor* charToProcess = chars - set(pre)  
 order = **''  
 while** charToProcess:  
 ch = charToProcess.pop()  
 order += ch  
 **for** b **in** suc[ch]:  
 pre[b].discard(ch)  
 **if not** pre[b]: *# if processed all predecesesors let's add curret char to charToProcess* charToProcess.add(b)  
 **return** order \* (set(order) == chars)

10. **Kth Largest Element in an Array**

**class** Solution:  
 **def** quick\_select(self, arr, l, r, k):  
 **if** 0 < k <= r - l + 1:  
 pos = self.partition(arr, l, r)  
 **if** pos - l == k - 1:  
 **return** arr[pos]  
 **if** pos - l > k - 1:  
 **return** self.quick\_select(arr, l, pos - 1, k)  
 **return** self.quick\_select(arr, pos + 1, r, k - pos + l - 1)  
  
 **def** partition(self, arr, l, r):  
 i = l  
 x = arr[r]  
 **for** j **in** range(l, r):  
 **if** arr[j] > x:  
 arr[i],arr[j] = arr[j],arr[i]  
 i += 1  
 arr[i], arr[r] = arr[r], arr[i]  
 **return** i  
  
 **def** findKthLargest(self, nums, k):  
 **return** self.quick\_select(nums, 0, len(nums) - 1, k)  
  
sol = Solution()  
print(sol.findKthLargest([3,2,1,5,6,4], 2))

**11. WordLadder||**

**import** scala.collection.mutable.ListBuffer  
  
**case class** WordNode(word: String, numSteps: Int, pre: WordNode = **null**)  
  
**object** FindLaddersApp **extends** App {  
 **private def** log(last: WordNode): Unit = {  
 **val** sbf = **new** StringBuilder()  
 **var** cur = last  
 **while**(cur != **null**){  
 sbf.insert(0, s"**$**{cur.word}(**$**{cur.numSteps})")  
 cur = cur.pre  
 }  
 *println*(sbf.result())  
 }  
  
 **def** findLadders(beginWord: String, endWord: String, wordList: List[String]): List[List[String]] = {  
 **var** result = *List*.*empty*[List[String]]  
 **var** queue = *List*.*empty*[WordNode]  
 queue = *WordNode*(beginWord,1) :: queue  
 **var** minStep = 0  
 **var** visited = *Set*.empty[String]  
 **var** unvisited = *Set*.empty[String]  
 unvisited = unvisited ++ wordList  
 **var** preNumSteps = 0  
 **while**(queue.nonEmpty){  
 **var** top = queue.head  
 queue = queue.tail  
 **val** word = top.word  
 **val** currNumSteps = top.numSteps  
 *log*(top)  
 *println*(s"currNumStep = **$**currNumSteps")  
 **var** continue = **false  
 if**(word == endWord){  
 **if**(minStep == 0) minStep = top.numSteps  
 **if**(top.numSteps == minStep && minStep !=0) {  
 //nothing  
 **val** t = **new** ListBuffer[String]()  
 t += top.word  
 **while**(top.pre !=**null**){  
 t.+=:(top.pre.word)  
 top = top.pre  
 }  
 result = result :+ t.toList  
 continue = **true** }  
 }  
 **if**(!continue){  
 **if**(preNumSteps < currNumSteps){  
 unvisited = unvisited -- visited  
 }  
 *println*(s"preNumSteps = **$**preNumSteps")  
 preNumSteps = currNumSteps  
 **val** arr = word.toArray  
  
 **for**(i <- 0 until arr.length) {  
 **for**(c <- 'a' to 'z'){  
 **val** temp = arr(i)  
 **if**(arr(i) != c){  
 arr(i) = c  
 }  
  
 **val** newWord = String.*valueOf*(arr)  
 **if**(unvisited.contains(newWord)) {  
 queue = queue :+ *WordNode*(newWord, top.numSteps + 1, top)  
 visited = visited + newWord  
 }  
 arr(i)=temp  
 }  
 }  
  
 }  
 }  
 result  
 }  
 *println*(*findLadders*("hit", "cog", *List*("hot","dot","dog","lot","log","cog")))  
}