**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 items 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)

**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. WordLadder2**

**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")))  
}

**12.Best Time Buy Sell Stock**

def maxProfit(prices: Array[Int]): Int = {

if (prices.isEmpty) 0

else {

var maxprofit = 0

var minprice = Int.MaxValue

for (i <- 0 to prices.length - 1) {

if (minprice > prices(i)) {

minprice = prices(i)

}

else if ((prices(i) - minprice) > 0) {

maxprofit = (prices(i) - minprice) max maxprofit

}

}

maxprofit

}

}

**Design HitCounter**

class HitCounter {

private val elapsedPeriod = 300

private val times = Array.ofDim[Int](elapsedPeriod)

private val hits = Array.ofDim[Int](elapsedPeriod)

/\*\* Record a hit.

@param timestamp - The current timestamp (in seconds granularity). \*/

def hit(timestamp: Int) {

val index = timestamp % elapsedPeriod

if(times(index) != timestamp){

times(index) = timestamp

hits(index) = 1

} else hits(index) += 1

}

/\*\* Return the number of hits in the past 5 minutes.

@param timestamp - The current timestamp (in seconds granularity). \*/

def getHits(timestamp: Int): Int = {

times.zip(hits).filter(th => timestamp - th.\_1 < elapsedPeriod).map(\_.\_2).sum

}

}

**Binary Search Tree Iterator**

public class BSTIterator {

private List<TreeNode> stack = new LinkedList<>();

public BSTIterator(TreeNode root) {

pushAll(root);

}

/\*\*

\* @return whether we have a next smallest number

\*/

public boolean hasNext() {

return !stack.isEmpty();

}

public int next() {

TreeNode tmpNode = stack.remove(stack.size()-1);

pushAll(tmpNode.right);

return tmpNode.val;

}

private void pushAll(TreeNode node) {

for (; node != null; stack.add(node), node = node.left);

}

}

**10. Find a celebrity**

public int findCelebrity(int n) {  
 int canditate = 0;  
 for(int i = 1; i< n;i++){  
 if(knows(canditate,i)){  
 canditate = i;  
 }  
 }  
 for(int i = 0; i< n;i++){  
 if((i!=canditate) &&(knows(canditate,i) || !knows(i, canditate))) return -1;  
 }  
 return canditate;  
}

**Minimum Sliding Window**

public static String minWindow(String s, String t) {  
 if(s == null || t == null || s.length() < t.length()) return "";  
 int map[] = new int[128];  
 for(int i = 0; i< t.length(); i++){  
 map[t.charAt(i)] += 1;  
 }  
 int counter = t.length();  
 int begin = 0,end = 0;  
 int d = Integer.*MAX\_VALUE*;  
 int head = 0;  
 while(end < s.length()){  
 if(map[s.charAt(end)] > 0) counter -=1;  
 map[s.charAt(end)] -= 1;  
 end += 1;  
 while(counter == 0){  
 if(end - begin < d) {  
 d = end - begin;  
 head = begin;  
 }  
 if(map[s.charAt(begin)] == 0) counter += 1;  
 map[s.charAt(begin)] += 1;  
 begin += 1;  
  
 }  
 }  
 return d == Integer.*MAX\_VALUE* ? "": s.substring(head, head + d);  
}

**GenerateParentheses**

GenerateParentheses

**private def** generate(s: Array[Char], pos:Int,n: Int,open:Int,close:Int, res:ListBuffer[String]):Unit = {  
 **if**(close == n) {  
 res += s.mkString("")  
 }  
 **else** {  
 **if**(open > close){  
 s(pos) = ')'  
 *generate*(s, pos + 1, n, open, close + 1, res)  
 }  
 **if**(open < n){  
 s(pos) = '('  
 *generate*(s, pos + 1, n, open + 1, close, res)  
 }  
 }  
}  
  
**def** generateParenthesis(n: Int): List[String] = {  
 **val** res = **new** ListBuffer[String]()  
 *generate*(Array.*ofDim*[Char](2\* n),0, n, 0, 0,res)  
 res.toList  
}

**99. Evaluate Division**

1. x/y = 2 x → y is edge with weight 2.

2. create the graph with x/y and y/x edges.

3. Use DFS to find a/c path and calculate the path weight.

**91. Decode Ways**

1. Represent num\_Decoding(s) = num\_Decoding(s,1:k) +

num\_Decoding(s,k+1,n)

2. Use memoization to cache num\_Decoding for a substring

3. Take into account to consider 01 and 100… as invalid string

**621. Task Scheduler**

1. Sort the tasks by its frequences. Let start from bigger frequencies and run the inner loop by the cooling time taking the pending tasks from 25 — i task and increase the time by 1,

2. Again sort the frequences array in the descending order in each cycle iteration.

3. Run outer loop until the arr[25] is 0

**560. Subarray Sum Equals K**

1. Consider sum(subarray(0,j)) + k = sum(subarray(0, i)) for j < i.

2. Store a frequency of sum(subarray) in map.

3. if sum(subarray) — k in map to sum up frequency of it with total.

**468. Validate IP Address**

Use regular expression to extract IP parts:

res = re.match(r'^([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})$', ip)

res = re.search('^([0-9abcdefABCDEF]{1,4}):([0-9abcdefABCDEF]{1,4}):([0-9abcdefABCDEF]{1,4}):([0-9abcdefABCDEF]{1,4}):([0-9abcdefABCDEF]{1,4}):([0-9abcdefABCDEF]{1,4}):([0-9abcdefABCDEF]{1,4}):([0-9abcdefABCDEF]{1,4})$', ip)

Check that there is no leading zeros in IP4: 001 and Ip address has to start from a > 0.

**289. Game of Life**

1. Allocate a buffer for calculating with 2 rows in size.

2. Put i-th row in a buffer. If a buffer reached size of 2 rows it would copy 0-row back to original array and append new row from array.

**46. Permutations**

1.create the function backtrack(first):

- check if first = len of arr then it returns

- iterate from first to end:

a) swap i and first and call backtrack recursively

b) swap i and first back

**151. Reverse Words in a String**

1. copy a string in inverse order.

2. iterate through the string and push the char to a stack

3. if it meets space then to pop the chars to result list — if the result list is not empty, it appends space before chars.

4. join the result result

**127. Word Ladder**

1. Use BFS for search endWord from beginWord

2. Store visited words to a dict to prevent repeative using of it.

3. Store in a queue a tuple (word,length)

4. When adding new neighbours to a queue it add theirs to visited dict and check if added word is endWord.

**380.** [**Insert Delete GetRandom O(1)**](https://leetcode.com/problems/insert-delete-getrandom-o1)

1. Declare an array for values and a map for value and its position in the array.

2. While inserting new value it has to check if new value already exists. If not exist it will add this value to the end of this array

3.While removing a value it swap last item with the value and remove last item in the array and the value in the map.

**79. Word Search**

1. Use backtracking to check if a word in list.

2. if backtrack function it checks if word in position equals a character in the matix. If so it calls this with i+1,i-1,j-1, j+1 characters in matrix by adding the used characters in seen before each recursive call and then it removes it later.

3. In main function it iterates through all characters and call the backtracking for each character.

**FriendCircles**

1. Use FindUnion with path compression and rank.
2. If m[i][j] == 1 or m[p1]m[p2] == 1 let’s union friends
3. Filter out the unique parent values. There are circle size

**Word Frequencies**

cat words.txt | tr -s ' ' '\n' | awk '{nums[$1]++}END{for(word in nums) print word, nums[word]}' | sort -rn -k2