**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 - 1 == 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);  
}

**Generate Parentheses**

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. Sum up items.

2. if sum – k in map than add frequency to result.

3. Store current sum to frequency map (sum -> frequency)

**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 count

- 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

**127. Word Ladder**

1. Use BFS for search endWord from beginWord

2. Store visited words to a dict to prevent repeated 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

**923. 3Sum With Multiplicity**

1. Use 2 pointer technic

2. Sort array, take i-th item, and look for target – a[i] from i + 1 to len(arr)-1.

3. if a[j] + a[k] == target – a[i] it has 2 cases:

3.1 if a[j] == a[k] num = (k – j + 1)\*(k-j) / 2

3.2 in another case left is count of a[j] and right is count of a[k]:

**while** j + 1 < k **and** arr[j] == arr[j + 1]:  
 left += 1  
 j += 1  
right = 1  
**while** k - 1 > j **and** arr[k] == arr[k - 1]:  
 right += 1  
 k -= 1  
num += left \* right

* 1. num += left \* right

**394. Decode String**

1. recursive decent

**class** Decoded:  
 **def** \_\_init\_\_(self):  
 self.pos = 0  
 self.res = **''  
  
  
class** Solution:  
 **def** is\_digit(self, ch):  
 **return '0'** <= ch <= **'9'  
  
 def** is\_char(self, ch):  
 **return 'a'** <= ch <= **'z' or 'A'** <= ch <= **'Z'  
  
 def** str(self, s, decoded):  
 **if** decoded.pos == len(s):  
 **return ""  
 if** self.is\_char(s[decoded.pos]):  
 j = decoded.pos  
 **while** decoded.pos < len(s) **and** self.is\_char(s[decoded.pos]):  
 decoded.pos += 1  
 decoded.res += s[j: decoded.pos]  
  
 **def** rep\_or\_str(self, s, decoded):  
 **if** decoded.pos == len(s):  
 **return ""  
 if** self.is\_char(s[decoded.pos]):  
 self.str(s, decoded)  
 **return** j = decoded.pos  
 **while** decoded.pos < len(s) **and** self.is\_digit(s[decoded.pos]):  
 decoded.pos += 1  
 k = int(s[j: decoded.pos])  
 decoded.pos += 1  
 prev = decoded.pos  
 **for** i **in** range(k):  
 decoded.pos = prev  
 **while** s[decoded.pos] != **']'**:  
 self.str(s, decoded)  
 **if** self.is\_digit(s[decoded.pos]):  
 self.rep\_or\_str(s, decoded)  
 **elif** self.is\_char(s[decoded.pos]):  
 self.str(s, decoded)  
 decoded.pos += 1  
  
 **def** decodeString(self, s: str) -> str:  
 **if not** str:  
 **return ""** decoded = Decoded()  
 **while** decoded.pos < len(s):  
 self.rep\_or\_str(s, decoded)  
 **return** decoded.res

**528. Random Pick with Weight**

1. Consider array of weight sums: w[0], w[0] + w[1],…

2. Take random value from the interval [0, total)

3. Search least sum > random value using binary search:

While lo != hi:

If a >= array[mid] lo = mid + 1

Else hi = mid – 1

Return lo

**755. Pour Water**

1. Go to the left to look up the leftmost min:

While height[i+d] <= height[i]

2. Do the same for the right side.

3. in another case to increment height[k]:

**class** Solution:  
 **def** find\_min(self, heights, v, d):  
 i = v  
 res = -1  
 *#look up leftmost/rightmost min* **while** 0 <= (i + d) < len(heights) **and** heights[i + d] <= heights[i]:  
 **if** heights[i + d] < heights[i]:  
 res = i + d  
 i += d  
 **if** res != -1:  
 **return** res  
 **return** -1  
  
 **def** pourWater(self, heights, V: int, K: int):  
 **if not** heights **or** V == 0:  
 **return** heights  
 **for** v **in** range(V):  
 left = self.find\_min(heights, K, -1)  
 **if** left >= 0:  
 heights[left] += 1  
 **else**:  
 right = self.find\_min(heights, K, 1)  
 **if** right >= 0:  
 heights[right] += 1  
 **else**:  
 heights[K] += 1  
  
 **return** heights

**973. K Closest Points to Origin**

1. Use quick\_select to find k-th closest point to origin

2. partition uses forward and backward iterations

3. quick\_select uses mid - r + 1 and check if k < or > mid – l + 1 then choose the part having K. For right part it changes K = K – (mid –l +1) = K –mid + l -1

**103. Binary Tree Zigzag Level Order Traversal**

1. Use recursive helper function to traverse tree in order.

2. Use map as function parameter and level -> list

3. Function calling itself for left child and right child passes map and level + 1.

4. If level is even it appends node.val to list looking up in map by level. In other case it inserts node.val at the begging of list.

**332. Reconstruct Itinerary**

1. Put a ticket departure as key and list of indexes of tickets to map. Sort a list in lexical order.

2. Use dfs to walk around all tickets to find unique path from all tickets so dfs takes one of parameters used\_tickets and other parameters are dep, seen and path.

3. Call dfs until use all of tickets for a path

4. Reuse path in the same recursive level by reseting the path after calling of dfs.

**347. Top K Frequent Elements**

1. Make map element to its frequency

2. Create min heap of size k and if it reaches size k then it remove min element if it has less frequency than adding one has.

**Search in rotated Sorted array**

1. Use binary search
2. If leftmost <= middle and target is out of left half – it returns right half
3. If middle <= rightmost and target is out of the right half – it returns left half
4. If leftmost <= middle – it returns left half
5. If middle <= rightmost – it returns right half

**947. Most Stones Removed with Same Row or Column**

1. Use disjoin set union to store each x and y coordinate separately –

Let’s y will be in 10000+ y. if maximum coordinate is 10000

2. stones may hold in one component

3. So union x,y coordinates of stones

4. max moves = number of stones – number of components. In each component is left one stone so number of components are number of left stones.

**341. Flatten Nested List Iterator**

**class** NestedIterator:  
 **def** \_\_init\_\_(self, lst):  
 self.q = []  
 self.flatten(lst)  
  
 **def** flatten(self, lst):  
 **for** ni **in** lst:  
 **if** ni.isInteger():  
 self.q.append(ni.getInteger())  
 **else**:  
 self.flatten(ni.getList())  
  
 **def** next(self):  
 a = self.q[0]  
 self.q.pop(0)  
 **return** a  
  
 **def** hasNext(self):  
 **return** len(self.q)

**692. Top K Frequent Words**

1. Count word frequency in map

2. Use priority queue to get frequency minimum or lower lexicographical word if frequencies are equal

3.Add word to list and reverse list

**Course Schedule**

1. Create graph as adjacent list, list visit is with 3 states: 0 is if vertex is not seem,1 is wa seen, -1 is if vertex in cycle
2. Before each traversation we mark vertex as cycle part then as seen.
3. Use dfs to traverse graph
4. **def** canFinish(self, numCourses: int, prerequisites):  
    **if not** numCourses:  
    **return False** graph = [[] **for** \_ **in** range(numCourses)]  
    visit = [0 **for** i **in** range(numCourses)]  
    **for** i **in** range(len(prerequisites)):  
    graph[prerequisites[i][0]].append(prerequisites[i][1])  
     
    **def** dfs(v, graph, visit):  
    **if** visit[v] == 1:  
    **return True  
    if** visit[v] == -1:  
    **return False** visit[v] = -1  
    **for** u **in** graph[v]:  
    **if not** dfs(u, graph, visit):  
    **return False** visit[v] = 1  
    **return True  
     
    for** v **in** range(len(graph)):  
    **if not** dfs(v, graph, visit):  
    **return False  
    return True**

**895. Maximum Frequency Stack**

1. Create map value to frequency, map frequency to stack

2. If we push value we will add value to map : a -> (frequency + 1). If frequency > maxFrequency, we will update maxFrequency and put maxFrequency -> stack, push value to stack

3. When we pop value, we pop value from stack in second map, and decsreas value frequency in first map.

1. **ZigZag Conversion**

**def** convert(self, s, numRows):  
 **if not** s **or not** numRows:  
 **return** s  
 r = 0  
 i = 0  
 m = defaultdict(list)  
 **while** i < len(s):  
 **while** r < numRows **and** i < len(s):  
 m[r].append(s[i])  
 i += 1  
 r += 1  
 r -= 2  
 **while** r > 0 **and** i < len(s):  
 m[r].append(s[i])  
 i += 1  
 r -= 1  
 r = 0  
 res = **''  
 for** i **in** range(numRows):  
 **for** j **in** range(len(m[i])):  
 res += m[i][j]  
 **return** res

**BasicCalculator||**

class Solution:

def prec(self, op):

if op == '+' or op == '-':

return 0

return 1

def eval(self, stack, ops):

while len(stack) > 0:

zn = stack.pop()

a = ops.pop()

b = ops.pop()

if zn == '+':

ops.append(a + b)

elif zn == '-':

ops.append(b - a)

elif zn == '\*':

ops.append(a \* b)

else:

ops.append(b // a)

return ops.pop()

def postfix(self, s):

j = 0

stack = []

buf = []

while j < len(s):

if '0' <= s[j] <= '9':

i = j

while i < len(s) and '0' <= s[i] <= '9':

i+= 1

buf.append(s[j:i])

j = i

else:

if s[j] in ['+','-','\*','/']:

while len(stack) > 0 and self.prec(s[j]) <= self.prec(stack[len(stack) - 1]):

buf.append(stack.pop())

stack.append(s[j])

j += 1

while len(stack) > 0:

buf.append(stack.pop())

return buf

def calculate(self, s):

if not s:

return None

stack = []

postfix = self.postfix(s)

i = 0

while i < len(postfix):

x = postfix[i]

if x.isdecimal():

stack.append(int(x))

else:

a = stack.pop()

b = stack.pop()

if x == '+':

stack.append(a + b)

elif x == '-':

stack.append(b - a)

elif x == '\*':

stack.append(a \* b)

else:

stack.append(b // a)

i += 1

return stack.pop**()**

**974. Subarray Sums Divisible by K**

1. Make prefix sum array modulo K

2. Count frequency of meet remainders in array from 1.

So N such arrays may be combinated N choose 2 ways to get Subarray(I, j) % k == 0

1. Sum up such combination by v \* (v-1) / 2 + …
2. **class** Solution:  
    **def** subarraysDivByK(self, arr, k):  
    **if not** arr **or** len(arr) == 0 **or** k == 0:  
    **return** 0  
    p = [0]  
    **for** x **in** arr:  
    p.append((p[-1] + x) % k)  
     
    freq = Counter(p)  
    **return** int(sum(v \* (v - 1) // 2 **for** v **in** freq.values()))