Q1. Fruit into Baskets

As in Approach 1, we want the longest subarray with at most two different "types" (values of tree[i]). Call these subarrays valid.

Say we consider all valid subarrays that end at index j. There must be one with the smallest possible starting index i: lets say opt(j) = i.

Now the key idea is that opt(j) is a monotone increasing function. This is because any subarray of a valid subarray is valid.

**Algorithm**

Let's perform a sliding window, keeping the loop invariant that i will be the smallest index for which [i, j] is a valid subarray.

We'll maintain count, the count of all the elements in the subarray. This allows us to quickly query whether there are 3 types in the subarray or not.

class Solution {

public int totalFruit(int[] tree) {

Map<Integer, Integer> count = new HashMap<Integer, Integer>();

int res = 0, i = 0;

for (int j = 0; j < tree.length; ++j) {

count.put(tree[j], count.getOrDefault(tree[j], 0) + 1);

while (count.size() > 2) {

count.put(tree[i], count.get(tree[i]) - 1);

if (count.get(tree[i]) == 0) count.remove(tree[i]);

i++;

}

res = Math.max(res, j - i + 1);

}

return res;

}

Complexity: O(N)

Q2. Next Closest Time  
We have up to 4 different allowed digits, which naively gives us 4 \* 4 \* 4 \* 4 possible times. For each possible time, let's check that it can be displayed on a clock: ie., hours < 24 and mins < 60. The best possible time != start is the one with the smallest cand\_elapsed = (time - start) % (24 \* 60), as this represents the time that has elapsed since start, and where the modulo operation is taken to be always non-negative.

For example, if we have start = 720 (ie. noon), then times like 12:05 = 725 means that (725 - 720) % (24 \* 60) = 5 seconds have elapsed; while times like 00:10 = 10 means that (10 - 720) % (24 \* 60) = -710 % (24 \* 60) = 730 seconds have elapsed.

Also, we should make sure to handle cand\_elapsed carefully. When our current candidate time cur is equal to the given starting time, then cand\_elapsed will be 0 and we should handle this case appropriately.

class Solution {

public String nextClosestTime(String time) {

int start = 60 \* Integer.parseInt(time.substring(0, 2));

start += Integer.parseInt(time.substring(3));

int ans = start;

int elapsed = 24 \* 60;

Set<Integer> allowed = new HashSet();

for (char c: time.toCharArray()) if (c != ':') {

allowed.add(c - '0');

}

for (int h1: allowed) for (int h2: allowed) if (h1 \* 10 + h2 < 24) {

for (int m1: allowed) for (int m2: allowed) if (m1 \* 10 + m2 < 60) {

int cur = 60 \* (h1 \* 10 + h2) + (m1 \* 10 + m2);

int candElapsed = Math.floorMod(cur - start, 24 \* 60);

if (0 < candElapsed && candElapsed < elapsed) {

ans = cur;

elapsed = candElapsed;

}

}

}

return String.format("%02d:%02d", ans / 60, ans % 60);

}

}

* Time Complexity: O(1). We all 4^4 possible times and take the best one.
* Space Complexity: O(1)

**OR**

class Solution {

public String nextClosestTime(String time) {

HashSet<Integer> set = new HashSet<Integer>();

set.add(new Integer(time.charAt(0)- '0'));

set.add(new Integer(time.charAt(1)- '0'));

set.add(new Integer(time.charAt(3)- '0'));

set.add(new Integer(time.charAt(4)- '0'));

int hours = Integer.parseInt(time.substring(0,2));

int mins = Integer.parseInt(time.substring(3));

while(true) {

mins++;

hours = (hours + mins/60) % 24;

mins = mins % 60;

if(set.contains(hours/10) &&

set.contains(hours%10) &&

set.contains(mins/10) &&

set.contains(mins%10)) break;

}

return String.format("%02d", hours) + ":" + String.format("%02d", mins);

}

}

Q3. License Key Formatting   
public String licenseKeyFormatting(String s, int K) {

StringBuilder sb = new StringBuilder();

s = s.replace("-", "");

String s1 = s.toUpperCase();

Stack stk = new Stack();

for(int i = 0; i<s1.length();i++)

stk.push(s1.charAt(i));

int count = 0;

char[] s2 = s1.toCharArray();

while(!stk.empty())

{ count++;

if(count != K+1)

{ sb.append(stk.peek());

stk.pop();

}

if(count == K+1)

{ count = 0;

sb.append("-");

continue;

} }

return sb.reverse().toString();

}

Q4. Unique Email Addresses  
class Solution {

public int numUniqueEmails(String[] emails) {

Set<String> seen = new HashSet();

for (String email: emails) {

int i = email.indexOf('@');

String local = email.substring(0, i);

String rest = email.substring(i);

if (local.contains("+")) {

local = local.substring(0, local.indexOf('+'));

}

local = local.replaceAll(".", "");

seen.add(local + rest);

}

return seen.size();

}

}

Q5. K Empty Slots

Approach: Sliding Window [Accepted]

**Intuition**

As in *Approach #2*, we have days[x] = i for the time that the flower at position x blooms. We wanted to find *candidate* intervals [left, right] where days[left], days[right] are the two smallest values in [days[left], days[left+1], ..., days[right]], and right - left = k + 1.

Notice that these candidate intervals cannot intersect: for example, if the candidate intervals are [left1, right1] and [left2, right2] with left1 < left2 < right1 < right2, then for the first interval to be a candidate, days[left2] > days[right1]; and for the second interval to be a candidate, days[right1] > days[left2], a contradiction.

That means whenever whether some interval can be a candidate and it fails first at i, indices j < i can't be the start of a candidate interval. This motivates a sliding window approach.

**Algorithm**

As in *Approach #2*, we construct days.

Then, for each interval [left, right] (starting with the first available one), we'll check whether it is a candidate: whether days[i] > days[left] and days[i] > days[right] for left < i < right.

If we fail, then we've found some new minimum days[i] and we should check the new interval [i, i+k+1]. If we succeed, then it's a candidate answer, and we'll check the new interval [right, right+k+1]

class Solution {

public int kEmptySlots(int[] flowers, int k) {

int[] days = new int[flowers.length];

for (int i = 0; i < flowers.length; i++) {

days[flowers[i] - 1] = i + 1;

}

int ans = Integer.MAX\_VALUE;

int left = 0, right = k+1;

search: while (right < days.length) {

for (int i = left+1; i < right; ++i) {

if (days[i] < days[left] || days[i] < days[right]) {

left = i;

right = i + k + 1;

continue search;

}

}

ans = Math.min(ans, Math.max(days[left], days[right]));

left = right;

right = left + k + 1;

}

return ans < Integer.MAX\_VALUE ? ans : -1;

}

}

Q5. **Longest Substring with At Most Two Distinct Characters**

The main idea is to maintain a sliding window with 2 unique characters. The key is to store the last occurrence of each character as the value in the hashmap. This way, whenever the size of the hashmap exceeds 2, we can traverse through the map to find the character with the left most index, and remove 1 character from our map. Since the range of characters is constrained, we should be able to find the left most index in constant time.

public int lengthOfLongestSubstringTwoDistinct(String s) {

if(s.length() < 1) return 0;

HashMap<Character,Integer> index = new HashMap<Character,Integer>();

int lo = 0;

int hi = 0;

int maxLength = 0;

while(hi < s.length()) {

if(index.size() <= 2) { ///For K chars replace 2 with k

char c = s.charAt(hi);

index.put(c, hi);

hi++;

}

if(index.size() > 2) { ///For K chars replace 2 with k

int leftMost = s.length();

for(int i : index.values()) {

leftMost = Math.min(leftMost,i);

}

char c = s.charAt(leftMost);

index.remove(c);

lo = leftMost+1;

}

maxLength = Math.max(maxLength, hi-lo);}//end of while

return maxLength;

Q6. K Empty Slots

class Solution {

/\*\*

\* Runtime Complexity: O(n logn)

\* Spatial Complexity: O(n)

\*/

public int kEmptySlots(int[] flowers, int k) {

TreeSet<Integer> state = new TreeSet<Integer>();

for(int day = 0; day < flowers.length; day++) {

state.add(flowers[day]);

Integer left = state.lower(flowers[day]);

Integer right = state.higher(flowers[day]);

if ((left != null && flowers[day] - left == k + 1) ||

(right != null && right - flowers[day] == k + 1))

return day+1;

}

return -1;

}

}

Q7. Evaluate Division

(1) Build the map, the key is dividend, the value is also a map whose key is divisor and value is its parameter. For example, a / b = 2.0, the map entry is <"a", <"b", 2.0>>. To make searching and calculation easier, we also put b / a = 0.5 into the map.  
(2) for each query, use DFS to search divisors recursively

public class Solution {

public double[] calcEquation(String[][] equations, double[] values, String[][] query) {

// build the map

Map<String, Map<String, Double>> map = new HashMap<>();

for(int i = 0; i < equations.length; i++) {

if(!map.containsKey(equations[i][0])) map.put(equations[i][0], new HashMap<>());

map.get(equations[i][0]).put(equations[i][1], values[i]);

if(!map.containsKey(equations[i][1])) map.put(equations[i][1], new HashMap<>());

map.get(equations[i][1]).put(equations[i][0], 1 / values[i]);

}

// search dividend and divisor using DFS

double[] res = new double[query.length];

for(int i = 0; i < query.length; i++) {

double[] para = new double[]{1.0};

if(calculate(map, query[i][0], query[i][1], para, new HashSet<>(), 1.0)) res[i] = para[0];

else res[i] = -1.0;

}

return res;

}

// DFS

private boolean calculate(Map<String, Map<String, Double>> map, String num1, String num2,

double[] para, Set<String> visited, double res) {

if(!map.containsKey(num1) || !map.containsKey(num2) || visited.contains(num1)) return false;

if(num1.equals(num2)) {

para[0] = res;

return true;

}

visited.add(num1);

for(Map.Entry<String, Double> entry: map.get(num1).entrySet()) {

if(calculate(map, entry.getKey(), num2, para, visited, res \* entry.getValue())) return true;

}

visited.remove(num1); // backtracking

return false;

}

}

Q8. Minimum cost to hire K workers

**"1. Every worker in the paid group should be paid in the ratio of their quality compared to other workers in the paid group."**  
So for any two workers in the paid group,  
we have wage[i] : wage[j] = quality[i] : quality[j]  
So we have wage[i] : quality[i] = wage[j] : quality[j]  
We pay wage to every worker in the group with the same ratio compared to his own quality.

**"2. Every worker in the paid group must be paid at least their minimum wage expectation."  
For every worker, he has an expected ratio of wage compared to his quality.**

So to minimize the total wage, we want a small ratio.  
So we sort all workers with their expected ratio, and pick up K first worker.  
Now we have a minimum possible ratio for K worker and we their total quality.

As we pick up next worker with bigger ratio, we increase the ratio for whole group.  
Meanwhile we remove a worker with highest quality so that we keep K workers in the group.  
We calculate the current ratio \* total quality = total wage for this group.

We redo the process and we can find the minimum total wage.  
Because workers are sorted by ratio of wage/quality.  
For every ratio, we find the minimum possible total quality of K workers.

**Time Complexity**  
O(NlogN) for sort.  
O(NlogK) for priority queue.

public double mincostToHireWorkers(int[] q, int[] w, int K) {

double[][] workers = new double[q.length][2];

for (int i = 0; i < q.length; ++i)

workers[i] = new double[]{(double)(w[i]) / q[i], (double)q[i]};

Arrays.sort(workers, (a, b) -> Double.compare(a[0], b[0]));

double res = Double.MAX\_VALUE, qsum = 0;

PriorityQueue<Double> pq = new PriorityQueue<>();

for (double[] worker: workers) {

qsum += worker[1];

pq.add(-worker[1]);

if (pq.size() > K) qsum += pq.poll();

if (pq.size() == K) res = Math.min(res, qsum \* worker[0]);

}

return res;

}

Q9. Guess the word

**Thought**  
If there are exact matches matches between our guess word wordlist[rand] and the secret word, those words do not share matches matches with wordlist[rand] cannot be the secret word.

**Reference**  
list.toArray(T[] a)  
Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

private final static int WORD\_LENGTH = 6;

public void findSecretWord(String[] wordlist, Master master) {

int index = new Random().nextInt(wordlist.length);

String wordGuessed = wordlist[index];

List<String> curlist = new LinkedList<>();

int numMatches = master.guess(wordGuessed);

if (numMatches == WORD\_LENGTH)

return;

for (int i = 0; i < wordlist.length; i++) {

if (i != index && countMatches(wordlist[i], wordGuessed) == numMatches) {

curlist.add(wordlist[i]);

}

}

wordlist = curlist.toArray(new String[curlist.size()]);

findSecretWord(wordlist, master);

}

private int countMatches(String word, String wordGuessed) {

int numMatches = 0;

for (int i = 0; i < WORD\_LENGTH; i++) {

if (word.charAt(i) == wordGuessed.charAt(i))

numMatches++;

}

return numMatches;

}

Q 10. Find and Replace String  
 public String findReplaceString(String S, int[] indexes, String[] sources, String[] targets) {

List<int[]> sorted = new ArrayList<>();

for (int i = 0 ; i < indexes.length; i++)

sorted.add(new int[]{indexes[i], i});

Collections.sort(sorted, Comparator.comparing(i -> -i[0]));

for (int[] ind: sorted) {

int i = ind[0], j = ind[1];

String s = sources[j], t = targets[j];

if (S.substring(i, i + s.length()).equals(s)) S = S.substring(0, i) + t + S.substring(i + s.length());

}

return S;

}

Q11. Robot Room Cleaner

We cannot move backward directly, so we need to turn left, turn left, move forward, turn right, and turn right to mock moving backward while the orientation doesn't change. We maintain a variableorientation that indicates the current orientation.  
The room is modeled as a grid but no cells are given, so we have to assume we start at (0, 0).  
Please note that when we detect that the current cell is cleaned(visited) before, we turn right before 'continue'.

private final static int[][] directions = {{0, -1}, {-1, 0}, {0, 1}, {1, 0}};

public void cleanRoom(Robot robot) {

cleanRoomDFS(robot, new HashSet<>(), 0, 0, 0);

}

private void cleanRoomDFS(Robot robot, Set<String> visited, int x, int y, int orientation) {

visited.add(x + "," + y);

robot.clean();

for (int i = 0; i < 4; i++) {

int nx = directions[orientation][0] + x;

int ny = directions[orientation][1] + y;

if (visited.contains(nx + "," + ny)) {

robot.turnRight();

orientation++;

orientation %= 4;

continue;

}

if (robot.move()) {

cleanRoomDFS(robot, visited, nx, ny, orientation);

robot.turnRight();

robot.turnRight();

robot.move();

robot.turnLeft();

robot.turnLeft();

}

robot.turnRight();

orientation++;

orientation %= 4;

}

}

**OR**

**class** **Solution** {

int[][] dirs = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

**public** void cleanRoom(Robot robot) {

Set<**String**> visited = **new** HashSet<>();

visited.add("0-0");

dfs(robot, 0, 0, 0, visited);

}

// [x, y] is the relative position from the initial point

**private** void dfs(Robot robot, int x, int y, int curDir, Set<**String**> visited) {

robot.clean();

**for** (int i = 0; i < 4; i++) {

int nextDir = (curDir + i) % 4;

int **new**X = x + dirs[nextDir][0];

int **new**Y = y + dirs[nextDir][1];

**if** (!visited.contains(**new**X + "-" + **new**Y) && robot.move()) {

visited.add(**new**X + "-" + **new**Y);

dfs(robot, **new**X, **new**Y, nextDir, visited);

}

robot.turnRight();

}

robot.turnRight();

robot.turnRight();

robot.move();

robot.turnRight();

robot.turnRight();

}

}

Q12. Bricks falling when hit

#### Approach #1: Reverse Time and Union-Find [Accepted]

**Intuition**

The problem is about knowing information about the connected components of a graph as we cut vertices. In particular, we'll like to know the size of the "roof" (component touching the top edge) between each cut. Here, a cut refers to the erasure of a vertex.

As we may know, a useful data structure for joining connected components is a disjoint set union structure. The key idea in this problem is that we can use this structure if we work in reverse: instead of looking at the graph as a series of sequential cuts, we'll look at the graph after all the cuts, and reverse each cut.

**Algorithm**

We'll modify our typical disjoint-set-union structure to include a dsu.size operation, that tells us the size of this component. The way we do this is whenever we make a component point to a new parent, we'll also send it's size to that parent.

We'll also include dsu.top, which tells us the size of the "roof", or the component connected to the top edge. We use an ephemeral "source" node with label R \* C where all nodes on the top edge (with row number 0) are connected to the source node.

For more information on DSU, please look at Approach #2 in the [article here](https://leetcode.com/articles/redundant-connection/).

Next, we'll introduce A, the grid after all the cuts have happened, and initialize our disjoint union structure on the graph induced by A (nodes are grid squares with a brick; edges between 4-directionally adjacent nodes).

After, if we get an cut at (r, c) but the original grid[r][c] was always 0, then we couldn't have had a meaningful cut - the number of dropped bricks is 0.

Otherwise, we'll look at the size of the new roof after adding this brick at (r, c), and compare them to find the number of dropped bricks.

Since we were working in reverse time order, we should reverse our working answer to arrive at our final answer.

class Solution {

public int[] hitBricks(int[][] grid, int[][] hits) {

int R = grid.length, C = grid[0].length;

int[] dr = {1, 0, -1, 0};

int[] dc = {0, 1, 0, -1};

int[][] A = new int[R][C];

for (int r = 0; r < R; ++r)

A[r] = grid[r].clone();

for (int[] hit: hits)

A[hit[0]][hit[1]] = 0;

DSU dsu = new DSU(R\*C + 1);

for (int r = 0; r < R; ++r) {

for (int c = 0; c < C; ++c) {

if (A[r][c] == 1) {

int i = r \* C + c;

if (r == 0)

dsu.union(i, R\*C);

if (r > 0 && A[r-1][c] == 1)

dsu.union(i, (r-1) \*C + c);

if (c > 0 && A[r][c-1] == 1)

dsu.union(i, r \* C + c-1);

}

}

}

int t = hits.length;

int[] ans = new int[t--];

while (t >= 0) {

int r = hits[t][0];

int c = hits[t][1];

int preRoof = dsu.top();

if (grid[r][c] == 0) {

t--;

} else {

int i = r \* C + c;

for (int k = 0; k < 4; ++k) {

int nr = r + dr[k];

int nc = c + dc[k];

if (0 <= nr && nr < R && 0 <= nc && nc < C && A[nr][nc] == 1)

dsu.union(i, nr \* C + nc);

}

if (r == 0)

dsu.union(i, R\*C);

A[r][c] = 1;

ans[t--] = Math.max(0, dsu.top() - preRoof - 1);

}

}

return ans;

}

}

class DSU {

int[] parent;

int[] rank;

int[] sz;

public DSU(int N) {

parent = new int[N];

for (int i = 0; i < N; ++i)

parent[i] = i;

rank = new int[N];

sz = new int[N];

Arrays.fill(sz, 1);

}

public int find(int x) {

if (parent[x] != x) parent[x] = find(parent[x]);

return parent[x];

}

public void union(int x, int y) {

int xr = find(x), yr = find(y);

if (xr == yr) return;

if (rank[xr] < rank[yr]) {

int tmp = yr;

yr = xr;

xr = tmp;

}

if (rank[xr] == rank[yr])

rank[xr]++;

parent[yr] = xr;

sz[xr] += sz[yr];

}

public int size(int x) {

return sz[find(x)];

}

public int top() {

return size(sz.length - 1) - 1;

}

}