1. anagrams

// assumption: coded in ascii

public class Solution{

public boolean anagram(String s, String t){

if(s == null || t == null){

return true;

}else if(s.length() == 0 || t.length() == 0){

return true;

}else if(s.length() != t.length()){

return false;

}

int[] array = new int[256];

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

array[(int) t.charAt(i)] ++;

}

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

array[(int) s.charAt(j)] --;

if(array[(int) s.charAt(j)] < 0){

return false;

}

}

return true;

}

}

2. compare strings

// assumption: ascii coding

public class Solution{

public boolean compare(String A, String B){

if(A == null){

return false;

}else if(B == null){

return true;

}else if(A.length() < B.length()){

return false;

}

int[] count = new int[256];

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

count[A.charAt(i)]++;

}

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

count[B.charAt(j)]--;

if(count[B.charAt(j)] < 0){

return false;

}

}

return true;

}

}

3. Strstr

public class Solution{

public int strStr(String source, String target){

if(source == null || target == null){

return -1;

}else if(source.length() == 0 && target.length() == 0){

return 0;

}else if(target.length() > source.length()){

return -1;

}

int targetHash = 0;

int sourceHash = 0;

int prime = 31;

int largePrime = 101;

int seed = 1;

// hash target

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

targetHash = hash(targetHash, target.charAt(i), prime, largePrime);

sourceHash = hash(sourceHash, source.charAt(i), prime, largePrime);

if(i > 0){

seed = hash(seed, 0, prime, largePrime);

}

}

// if target == source at index = 0, return 0, need to compare two strings as well to avoid collision

if(targetHash == sourceHash && equals(target, source, 0)){

return 0;

}

// sliding window

for(int i = 1; i < source.length() - target.length() + 1; i++){

// get rid of the last element

sourceHash = nonNegative(sourceHash - source.charAt(i - 1) \* seed % largePrime, largePrime);

// add the next element

sourceHash = hash(sourceHash, source.charAt(i + target.length() - 1), prime, largePrime);

if(sourceHash == targetHash && equals(target, source, i)){

return i;

}

}

return -1;

}

private int hash(int multiply, int add, int prime, int largePrime){

return (multiply \* prime % largePrime + add) % largePrime;

}

private int nonNegative(int hash, int largePrime){

if(hash < 0){

hash += largePrime;

}

return hash;

}

private boolean equals(String target, String source, int index){

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

if(target.charAt(i) != source.charAt(i + index)){

return false;

}

}

return true;

}

}

4. Anagrams

public class Solution{

public List<String> anagrams(String[] strs) {

if(strs == null || strs.length == 0){

return new ArrayList<String>();

}

List<String> result = new ArrayList<String>();

Map<String, List<String>> strsMap = new HashMap<String, List<String>>();

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

char[] array = strs[i].toCharArray();

Arrays.sort(array);

String sortedString = new String(array);

// new

if(!strsMap.containsKey(sortedString)){

List<String> currentString = new ArrayList<String>();

currentString.add(strs[i]);

strsMap.put(sortedString, currentString);

}else{

strsMap.get(sortedString).add(strs[i]);

}

}

for(String str: strsMap.keySet()){

if(strsMap.get(str).size() > 1){

result.addAll(strsMap.get(str));

}

}

return result;

}

}

5. Longest common substring

public class Solution{

public int longestCommonSubstring(String A, String B) {

if(A == null || A.length() == 0 || B == null || B.length() == 0){

return 0;

}

int[][] result = new int[A.length()][B.length()];

int longest = 0;

// initialization

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

if(A.charAt(i) == B.charAt(0)){

result[i][0] = 1;

}

longest = Math.max(longest, result[i][0]);

}

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

if(B.charAt(j) == A.charAt(0)){

result[0][j] = 1;

}

longest = Math.max(longest, result[0][j]);

}

// dp

for(int i = 1; i < A.length(); i++){

for(int j = 1; j < B.length(); j++){

if(A.charAt(i) == B.charAt(j)){

result[i][j] = result[i - 1][j -1] + 1;

}else{

result[i][j] = 0;

}

longest = Math.max(longest, result[i][j]);

}

}

return longest;

}

}

6. Remove element

public class Solution{

public int removeElement(int[] A, int element) {

if(A == null || A.length == 0){

return 0;

}

// quicksort ish method

// left : not element)

// right: (elememt

int left = 0;

int right = A.length - 1;

while(left <= right){

if(A[left] != element){

left++;

}else if(A[right] == element){

right--;

}else{

// left is element and right is not element

swap(A, left, right);

left++;

right--;

}

}

return left;

}

private void swap(int[] array, int left, int right){

int tmp = array[left];

array[left] = array[right];

array[right] = tmp;

}

}

7. Longest common prefix

public class Solution{

public String longestCommonPrefix(String[] strs) {

if(strs == null || strs.length == 0){

return new String();

}

// get length length

int length = Integer.MAX\_VALUE;

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

length = Math.min(length, strs[i].length());

}

// create substring

char[] result = new char[length];

// initialize to be the first string’s substring

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

result[i] = strs[0].charAt(i);

}

// iterate all remaining

int end = length; // exclusive

for(int i = 1; i < strs.length; i++){

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

if(result[j] != strs[i].charAt(j)){

end = Math.min(end, j);

break;

}

}

}

return new String(result, 0, end);

}

}

8. Subarray sun

public class Solution {

public ArrayList<Integer> subarraySum(int[] nums) {

if(nums == null || nums.length == 0){

return new ArrayList<Integer>();

}

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

ArrayList<Integer> result = new ArrayList<>();

map.put(0, -1); // subarray sum end at -1 (inclusive)

int sumEndAtI = 0;

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

sumEndAtI += nums[i]; // inclusive

Integer index = map.get(sumEndAtI);

if(index == null){

// new value

map.put(sumEndAtI, i);

}else{

result.add(index + 1);

result.add(i);

return result;

}

}

return result;

}

}

9. Remove duplicates

public class Solution{

public int removeDuplicates(int[] nums) {

if(nums == null || nums.length == 0){

return 0;

}

int slow = 0;

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

if(nums[slow] != nums[i]){

slow++;

swap(nums, slow, i);

}

}

return (slow + 1);

}

private void swap(int[] array, int left, int right){

int tmp = array[left];

array[left] = array[right];

array[right] = tmp;

}

}

10. Merge sorted arrary

public class Solution{

public void mergeSortedArray(int[] A, int m, int[] B, int n) {

if(A == null || B == null){

return;

}

int finalIndex = m + n - 1;

int aIndex = m - 1;

int bIndex = n - 1;

while(finalIndex >= 0){

// move from a

// case 1. there is no b

// case 2. there is a and a is larger

if(bIndex < 0 || aIndex >=0 && A[aIndex] > B[bIndex]){

A[finalIndex] = A[aIndex];

finalIndex--;

aIndex--;

}else{

// move from b

A[finalIndex] = B[bIndex];

finalIndex--;

bIndex--;

}

}

return;

}

}

11. Product of array excluding itselft

public class Solution {

public ArrayList<Long> productExcludeItself(ArrayList<Integer> A) {

if(A == null || A.size() == 0){

return new ArrayList<Long>();

}

// from left to right

// leftProduct: product of everything from left excluding i

long leftProduct = 1;

ArrayList<Long> result = new ArrayList<>();

for(int i = 0; i < A.size(); i++){

result.add(leftProduct);

leftProduct \*= A.get(i);

}

// right to left

// rightProduct: product of everything from right excluding i

long rightProduct = 1;

for(int i = A.size() - 1; i >= 0; i--){

long currentLeftProduct = result.get(i);

long finalProduct = currentLeftProduct \* rightProduct;

// update

result.set(i, finalProduct);

rightProduct \*= A.get(i);

}

return result;

}

}

12. First missing positive

public class Solution{

// assumption: no duplicates

public int firstMissingPositive(int[] A) {

if(A == null || A.length == 0){

return 1;

}

Set<Integer> result = new HashSet<>();

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

result.add(A[i]);

}

for(int i = 1; i < A.length + 1; i++){

if(!result.contains(i)){

return i;

}

}

return A.length + 1;

}

public int firstMissingPositiveII(int[] A) {

if(A == null || A.length == 0){

return 1;

}

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

// swap the element to the original position

while(A[i] != i+1 && A[i] < A.length + 1 && A[i] > 0 && A[A[i] - 1] != A[i]){

swap(A, A[i] - 1, i);

}

}

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

if(A[i] != i + 1){

return i + 1;

}

}

return A.length + 1;

}

private void swap(int[] array, int left, int right){

int tmp = array[left];

array[left] = array[right];

array[right] = tmp;

}

}

13. 3Sum closest

public class Solution{

public int threeSumClosest(int[] numbers ,int target) {

// number is not null and numbers.length >= 3

Arrays.sort(numbers);

int closest = numbers[0] + numbers[1] + numbers[2];

for(int i = 0; i < numbers.length - 2; i++){

int fix = numbers[i];

int left = i + 1;

int right = numbers.length - 1;

while(left < right){

int leftElement = numbers[left];

int rightElement = numbers[right];

int currentSum = fix + leftElement + rightElement;

if(currentSum - target > 0){

right--;

}else if(currentSum - target < 0){

left++;

}else{

return target;

}

if(Math.abs(currentSum - target) < Math.abs(closest - target)){

closest = currentSum;

}

}

}

return closest;

}

}

14. 3sum

public class Solution{

public ArrayList<ArrayList<Integer>> threeSum(int[] numbers){

// corner case

if(numbers == null || numbers.length < 3){

return new ArrayList<ArrayList<Integer>>();

}

Arrays.sort(numbers);

ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

for(int i = 0; i < numbers.length - 2; i++){

if(i != 0 && numbers[i] == number[i - 1]){

continue;

}

int left = i + 1;

int right = numbers.length - 1;

while(left < right){

int sum = numbers[i] + numbers[left] + numbers[right];

if(sum == 0){

ArrayList<Integer> current = new ArrayList<>();

current.add(numbers[i]);

current.add(numbers[left]);

current.add(numbers[right]);

result.add(current);

// remove duplicates

while(left + 1 < numbers.length && right - 1 >= i + 1){

if(numbers[left + 1] == numbers[left]){

left++;

}else if(numbers[right - 1] == numbers[right]){

right--;

}else{

break;

}

}

left++;

right--;

}else if(sum < 0){

// too small

left++;

}else{

// too large

right--;

}

}

}

return result;

}

}

15. 2sum

public class Solution {

public int[] twoSum(int[] numbers, int target) {

// assumption: no duplicate

if(numbers == null || numbers.length < 2){

return new int[] {-1, -1};

}

for(int i = 0; i < numbers.length - 1; i++){

for(int j = i + 1; j < numbers.length; j++){

int sum = numbers[i] + numbers[j];

if(sum == target){

return new int[] {i + 1, j + 1};

}

}

}

return new int[] {-1, -1};

}

public int[] twoSum(int[] numbers, int target) {

// assumption: no duplicate answer

if(numbers == null || numbers.length < 2){

return new int[] {-1, -1};

}

// current number -> index

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

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

int complement = target - numbers[i];

Integer mapComplementIndex = map.get(complement);

if(mapComplementIndex == null){

// nothing in the map that can add up to the target

map.put(numbers[i], i);

}else{

return new int[] {mapComplementIndex + 1, i + 1};

}

}

return new int[] {-1, -1};

}

}

16. Quicksort

public class Solution {

public int[] quickSort(int[] array) {

if(array == null || array.length <= 1){

return array;

}

quickSortHelper(array, 0, array.length - 1);

return array;

}

private void quickSortHelper(int[] array, int left, int right){

// termination

if(left >= right){

return;

}

int pivotIndex = partition(array, left, right);

quickSortHelper(array, left, pivotIndex - 1);

quickSortHelper(array, pivotIndex + 1, right);

return;

}

private int partition(int[] array, int left, int right){

int pivotIndex = getPivot(left, right);

int pivotElement = array[pivotIndex];

// partition

// swap pivot to the end

swap(array, pivotIndex, right);

// from left to right - 1, swap everything larger than pivot to the right

int leftIndex = left;

int rightIndex = right - 1;

while(leftIndex <= rightIndex){

if(array[leftIndex] < pivotElement){

leftIndex++;

}else if(array[rightIndex] >= pivotElement){

rightIndex--;

}else{

swap(array, leftIndex, rightIndex);

leftIndex++;

rightIndex--;

}

}

// swap back pivot

swap(array, leftIndex, right);

// return pivotIndex

return leftIndex;

}

private int getPivot(int left, int right){

return left + (int) (Math.random() \* (right - left + 1));

}

private void swap(int[] array, int left, int right){

int tmp = array[left];

array[left] = array[right];

array[right] = tmp;

}

}

17. Combinations

public class Solution {

public List<List<Integer>> combine(int n, int k) {

// corner case

if(k > n){

return new ArrayList<List<Integer>>();

}

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

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

combineHelper(n, k, 1, current, result);

return result;

}

private void combineHelper(int n, int k, int start, List<Integer> current, List<List<Integer>> result){

// termination

if(current.size() == k){

result.add(new ArrayList(current));

}

for(int i = start; i <= n; i++){

current.add(i);

combineHelper(n, k, i + 1, current, result);

current.remove(current.size() - 1);

}

return;

}

}

18. combination sum

public class Solution {

public List<List<Integer>> combinationSum(int[] candidates, int target) {

if(candidates == null || candidates.length == 0){

return new ArrayList<List<Integer>>();

}

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

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

Arrays.sort(candidates);

combinationSumHelper(candidates, target, 0, current, result);

return result;

}

private void combinationSumHelper(int[] candidates, int target, int index, List<Integer> current, List<List<Integer>> result){

// stop one step before

if(index == candidates.length - 1){

if(target % candidates[index] == 0){

for(int i = 0; i < target / candidates[index]; i++){

current.add(candidates[index]);

}

result.add(new ArrayList(current));

for(int i = 0; i < target / candidates[index]; i++){

current.remove(current.size() - 1);

}

}

return;

}

// recursion

for(int i = 0; i <= target / candidates[index]; i++){

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

current.add(candidates[index]);

}

combinationSumHelper(candidates, target - i \* candidates[index], index + 1, current, result);

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

current.remove(current.size() - 1);

}

}

}

}

19. topological ordering

class DirectedGraphNode {

int label;

ArrayList<DirectedGraphNode> neighbors;

DirectedGraphNode(int x){

this.label = x;

this.neighbors = new ArrayList<DirectedGraphNode>();

}

}

public class Solution {

// assumption: no cycles

public ArrayList<DirectedGraphNode> topSort(ArrayList<DirectedGraphNode> graph) {

if(graph == null || graph.size() == 0){

return new ArrayList<DirectedGraphNode>();

}

// find head

// the only node with no neighbor is head

Map<DirectedGraphNode, Integer> child = new HashMap<>();

for(DirectedGraphNode node: graph){

for(DirectedGraphNode neighbor: node.neighbors){

Integer count = child.get(neighbor);

if(count == null){

child.put(neighbor, 1);

}else{

child.put(neighbor, count + 1);

}

}

}

// bfs

Deque<DirectedGraphNode> queue = new LinkedList<>();

ArrayList<DirectedGraphNode> result = new ArrayList<>();

// which one is not in the hash set

for(DirectedGraphNode node: graph){

if(!child.containsKey(node)){

queue.add(node);

result.add(node);

}

}

while(!queue.isEmpty()){

DirectedGraphNode current = queue.pollLast();

int currentLevelSize = current.neighbors.size();

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

DirectedGraphNode neighbor = current.neighbors.get(i);

// check if it is in the last layer

Integer count = child.get(neighbor);

if(count == 1){

queue.offerFirst(neighbor);

result.add(neighbor);

}else{

child.put(neighbor, count - 1); \

}

}

}

return result;

}

}

20. Word Ladder

public class Solution {

public int ladderLength(String start, String end, Set<String> dict) {

if(dict == null || dict.size() == 0){

if(canTransform(start, end)){

return 2;

}else{

return 0;

}

}

Deque<String> queue = new LinkedList<String>();

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

dict.add(end);

queue.offerFirst(start);

visited.add(start);

int length = 1;

while(!queue.isEmpty()){

int levelLength = queue.size();

length++;

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

String current = queue.pollLast();

ArrayList<String> next = getNext(current, dict);

for(String nextString: next){

if(nextString.equals(end)){

return length;

}else if(!visited.contains(nextString)){

visited.add(nextString);

queue.offerFirst(nextString);

}

}

}

}

return 0;

}

private ArrayList<String> getNext(String current, Set<String> dict){

ArrayList<String> result = new ArrayList<>();

for(String dictString: dict){

if(canTransform(dictString, current)){

result.add(dictString);

}

}

return result;

}

private boolean canTransform(String a, String b){

if(a.length() != b.length()){

return false;

}

int flag = 0;

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

if(a.charAt(i) != b.charAt(i)){

flag++;

}

}

return (flag == 1);

}

}

21. N Queens

public class Solution {

public ArrayList<ArrayList<String>> solveNQueens(int n) {

if(n < 1){

return new ArrayList<ArrayList<String>>();

}

ArrayList<Integer> current = new ArrayList<>();

ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

solveNQueensHelper(n, current, result);

ArrayList<ArrayList<String>> finalResult = convertToSolution(result, n);

return finalResult;

}

private void solveNQueensHelper(int n, ArrayList<Integer> current, ArrayList<ArrayList<Integer>> result){

// termination

if(current.size() == n){

result.add(new ArrayList(current));

return;

}

// recursion

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

if(checkNoConflict(i, current)){

current.add(i);

solveNQueensHelper(n, current, result);

current.remove(current.size() - 1);

}

}

return;

}

private boolean checkNoConflict(int number, ArrayList<Integer> current){

for(int i = 0; i < current.size(); i++){

// conflict

// case 1. column

// case 2. diagonal

if(number == current.get(i) || Math.abs(number - current.get(i)) == current.size() - i){

return false;

}

}

return true;

}

private ArrayList<ArrayList<String>> convertToSolution(ArrayList<ArrayList<Integer>> result, int n){

ArrayList<ArrayList<String>> finalResult = new ArrayList<ArrayList<String>>();

for(int i = 0; i < result.size(); i++){

// loop over answers

ArrayList<String> oneFinalResult = new ArrayList<String>();

for(int j = 0; j < n; j++){

// loop over rows

String rowString = "";

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

if(k == result.get(i).get(j)){

rowString += "Q";

}else{

rowString += ".";

}

}

oneFinalResult.add(rowString);

}

finalResult.add(oneFinalResult);

}

return finalResult;

}

}

21. subsets

class Solution {

public ArrayList<ArrayList<Integer>> subsets(int[] nums) {

if(nums == null){

return new ArrayList<ArrayList<Integer>>();

}

ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

ArrayList<Integer> current = new ArrayList<>();

Arrays.sort(nums); // assume nums are not sorted

subsetsHelper(nums, 0, current, result);

return result;

}

private void subsetsHelper(int[] nums, int index, ArrayList<Integer> current, ArrayList<ArrayList<Integer>> result){

// termination

if(index == nums.length){

result.add(new ArrayList(current));

return;

}

// recursion

// add nums[index]

current.add(nums[index]);

subsetsHelper(nums, index + 1, current, result);

current.remove(current.size() - 1);

// do not add nums[index];

subsetsHelper(nums, index + 1, current, result);

}

// way 2

public ArrayList<ArrayList<Integer>> subsets(int[] nums) {

if(nums == null){

return new ArrayList<ArrayList<Integer>>();

}

ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

ArrayList<Integer> current = new ArrayList<>();

Arrays.sort(nums); // assume nums are not sorted

result.add(current);

subsetsHelper(nums, 0, current, result);

return result;

}

private void subsetsHelper(int[] nums, int index, ArrayList<Integer> current, ArrayList<ArrayList<Integer>> result){

result.add(new ArrayList(current));

for(int i = index; i < nums.length; i++){

current.add(nums[i]);

subsetsHelper(nums, i + 1, current, result);

current.remove(current.size() - 1);

}

}

}

22. subsets with duplicates

class Solution {

public ArrayList<ArrayList<Integer>> subsetsWithDup(ArrayList<Integer> S) {

if(S == null){

return new ArrayList<ArrayList<Integer>>();

}

Integer[] array = S.toArray(new Integer[0]);

Arrays.sort(array);

ArrayList<Integer> current = new ArrayList<>();

ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

subsetsWithDupHelper(array, 0, current, result);

return result;

}

private void subsetsWithDupHelper(Integer[] array, int index, ArrayList<Integer> current, ArrayList<ArrayList<Integer>> result){

result.add(new ArrayList(current));

for(int i = index; i < array.length; i++){

// dedup

if(i == index || array[i] != array[i - 1]){

current.add(array[i]);

subsetsWithDupHelper(array, i + 1, current, result);

current.remove(current.size() - 1);

}

}

}

}

23. permutations

class Solution {

public ArrayList<ArrayList<Integer>> permute(ArrayList<Integer> nums) {

if(nums == null){

return new ArrayList<ArrayList<Integer>>();

}

ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

Integer[] array = nums.toArray(new Integer[0]);

permuteHelper(array, result, 0);

return result;

}

private void permuteHelper(Integer[] array, ArrayList<ArrayList<Integer>> result, int index){

// termination

if(index == array.length){

result.add(new ArrayList(Arrays.asList(array)));

return;

}

// recursion

for(int i = index; i < array.length; i++){

swap(array, i, index);

permuteHelper(array, result, index + 1);

swap(array, i, index);

}

}

private void swap(Integer[] array, int left, int right){

Integer tmp = array[left];

array[left] = array[right];

array[right] = tmp;

}

}

24. permutations II

public class Solution{

public ArrayList<ArrayList<Integer>> permuteUnique(ArrayList<Integer> nums){

if(nums == null){

return new ArrayList<ArrayList<Integer>>();

}

ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

Integer[] array = nums.toArray(new Integer[0]);

Arrays.sort(array);

permuteUniqueHelper(array, result, 0);

return result;

}

private void permuteUniqueHelper(Integer[] array, ArrayList<ArrayList<Integer>> result, int index){

// termination

if(index == array.length){

result.add(new ArrayList(Arrays.asList(array)));

return;

}

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

for(int i = index; i < array.length; i++){

//if(!set.contains(array[i])){

if(i > index && array[i] == array[i - 1]){

continue;

}else{

// set.add(array[i]);

swap(array, i, index);

permuteUniqueHelper(array, result, index + 1);

swap(array, i, index);

}

}

}

private void swap(Integer[] array, int left, int right){

int tmp = array[left];

array[left] = array[right];

array[right] = tmp;

}

}

25. word ladder II

// time : O(n^4)

// space : O(n^2)

public class Solution{

public List<List<String>> findLadders(String start, String end, Set<String> dict){

if(dict == null){

return new ArrayList<List<String>>();

}

dict.add(end);

// bfs

List<List<String>> result = new ArrayList<List<String>>();

Deque<List<String>> queue = new LinkedList<List<String>>();

List<String> startingState = new ArrayList<String>();

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

visited.add(start);

startingState.add(start);

queue.offerFirst(startingState);

while(!queue.isEmpty()){

int levelSize = queue.size();

boolean flag = false; // test whether or not we have reached end

Set<String> set = new HashSet<>();

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

List<String> current = queue.pollLast();

List<List<String>> nextStrings = getNextStrings(current, dict);

for(List<String> next : nextStrings){

if(visited.contains(next.get(next.size() - 1))){

continue;

}

if(!flag && next.get(next.size() - 1).equals(end)){

flag = true;

result.add(next);

}else if(flag && next.get(next.size() - 1).equals(end)){

result.add(next);

}else{

queue.offerFirst(next);

}

set.add(next.get(next.size() - 1));

}

}

for(String string : set){

visited.add(string);

}

if(flag){

return result;

}

}

return result;

}

private List<List<String>> getNextStrings(List<String> current, Set<String> dict){

List<List<String>> next = new ArrayList<List<String>>();

String currentLast = current.get(current.size() - 1);

for(String dictString : dict){

if(differOne(dictString, currentLast)){

current.add(dictString);

next.add(new ArrayList(current));

current.remove(current.size() - 1);

}

}

return next;

}

private boolean differOne(String dictString, String current){

int index = 0;

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

if(dictString.charAt(i) != current.charAt(i)){

index++;

}

}

return index == 1;

}

}

public class Solution{

public ArrayList<ArrayList<String>> solveNQueens(int n) {

if(n < 1){

return new ArrayList<ArrayList<String>>();

}

Deque<ArrayList<Integer>> queue = new LinkedList<ArrayList<Integer>>();

ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

// initialize queue

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

ArrayList<Integer> current = new ArrayList<>();

current.add(i);

queue.offerFirst(current);

}

// bfs

int level =0;

while(!queue.isEmpty()){

int levelSize = queue.size();

level++;

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

ArrayList<Integer> current = queue.pollLast();

if(level == n){

result.add(current);

}else{

for(int j = 0; j < n; j++){

if(checkNoConflict(j, current)){

current.add(j);

queue.offerFirst(new ArrayList(current));

current.remove(current.size() - 1);

}

}

}

}

}

ArrayList<ArrayList<String>> finalResult = convertToSolution(result, n);

return finalResult;

}

private boolean checkNoConflict(int number, ArrayList<Integer> current){

for(int i = 0; i < current.size(); i++){

if(number == current.get(i) || Math.abs(current.get(i) - number) == current.size() - i){

return false;

}

}

return true;

}

private ArrayList<ArrayList<String>> convertToSolution(ArrayList<ArrayList<Integer>> result, int n){

ArrayList<ArrayList<String>> finalResult = new ArrayList<ArrayList<String>>();

for(int i = 0; i < result.size(); i++){

ArrayList<String> oneFinalResult = new ArrayList<String>();

for(int j = 0; j < n; j++){

String row = "";

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

if(k == result.get(i).get(j)){

row += "Q";

}else{

row += ".";

}

}

oneFinalResult.add(row);

}

finalResult.add(oneFinalResult);

}

return finalResult;

}

}

public class Solution {

public boolean anagram(String s, String t) {

if (s == null || t == null) {

return false;

}

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

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

Character c = s.charAt(i);

if (count.containsKey(c)) {

count.put(c, count.get(c) + 1);

} else {

count.put(c, 1);

}

}

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

Character c = t.charAt(i);

if (!count.containsKey(c)) {

return false;

} else {

count.put(c, count.get(c) - 1);

if (count.get(c) < 0) {

return false;

}

}

}

return true;

}

}

public class Solution {

public boolean compareStrings(String A, String B) {

//deal with null situation

if (B == null) {

return true;

} else if (A == null) {

return false;

}

int[] count = new int[26]; //input is UPPER CASE letters

//count letters in A

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

count[A.charAt(i) - ‘A’]++;

}

//count letters in B

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

count[B.charAt(i) - ‘A’]--;

if (count[B.charAt(i) - ‘A’] < 0) {

return false;

}

}

return true;

}

class Solution {

public int strStr(String source, String target) {

if (target == null) {

return 0;

} else if (source == null) {

return -1;

} else if (target.length() == 0) {

return 0;

}

for (int i = 0; i < source.length() - target.length() + 1; i++) { //be careful about boundary condition!!!

String subStr = source.substring(i, i + target.length()); //not sure about syntax

if (subStr.compareTo(target) == 0) {

return i;

}

}

return -1;

}

}

public class Solution {

public List<String> anagrams(String[] strs) {

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

if (strs == null) {

return ret;

}

Map<String, List<String>> seen = new HashMap<String, List<String>>();

for (String s : strs) {

Strings sorted = sort(s); //typo!!

if (seen.containsKey(sorted)) {

seen.get(sorted).add(s);

} else {

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

container.add(s);

seen.put(sorted, container);

}

}

for (String s : seen.keySet()) { //not sure about syntax

if (seen.get(s).size() > 1) {

ret.addAll(seen.get(s));

}

}

return ret;

}

private String sort(String s) {

if (s == null) {

return null;

}

char[] c = s.toCharArray();

Arrays.sort(c);

return new String(c);

}

}

public class Solution {

public int longestCommonSubstring(String A, String B) {

if (A == null || B == null) {

return 0;

}

int ret = 0;

int n = A.length();

int m = B.length();

int[][] o = new int[n + 1][m + 1];

//o[][] records longest common substring ending at

//i - 1, j - 1 position of A and B

for (int i = 1; i <= n; i++) {

for (int j = 1; j <= m; j++) {

if (A.charAt(i - 1) == B.charAt(j - 1)) {

o[i][j] = o[i - 1][j - 1] + 1;

}

ret = Math.max(ret, o[i][j]);

}

}

return ret;

}

}

public class Solution {

public int removeElement(int[] array, int target) {

if (array == null) {

return 0;

}

int i = 0;

int j = array.length - 1;

while (i <= j) {

if (array[i] == target) {

swap(array, i, j--);

} else {

i++;

}

}

return j + 1;

}

private void swap(int[] a, int i, int j) {

int tmp = a[i];

a[i] = a[j];

a[j] = tmp;

}

}

public class Solution {

public String longestCommonPrefix(String[] strs) {

String ret = “”;

if (strs == null || strs.length == 0) {

return ret;

}

int l = minLen(strs);

//look at first l chars

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

char cur = strs[0].charAt(i);

//look at all chars at position i

for (int j = 1; j < strs.length; j++) {

if (cur != strs[j].charAt(i)) {

return ret;

}

}

ret += cur;

}

return ret;

}

//return min length of strings in strs

private int minLen(String[] strs) {

int l = 0;

if (strs == null || strs.length == 0) {

return l;

}

l = strs[0].length();

for (int i = 1; i < strs.length; i++) {

l = Math.min(l, strs[i].length());

}

return l;

}

}

public class Solution {

public ArrayList<Integer> subarraySum(int[] nums) {

ArrayList<Integer> ret = new ArrayList<Integer>();

if (nums == null) {

return ret;

}

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

int cumSum = 0; //assume sum of array is smaller than Integer.MAX\_VALUE

seen.put(cumSum, -1); //in case we have a subarray of sum of zero from 0

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

cumSum += nums[i];

if (seen.containsKey(cumSum)) {

ret.add(1 + seen.get(cumSum));

ret.add(i);

break;

} else {

seen.put(cumSum, i);

}

}

return ret;

}

}

public class Solution {

public int removeDuplicates(int[] nums) {

if (nums == null || nums.length == 0) {

return 0;

}

int i = 0; //pointer for unique elements

for (int j = 1; j < nums.length; j++) {

if (nums[i] != nums[j]) {

exch(nums, ++i, j);

}

}

return i + 1;

}

private void exch(int[] nums, int i, int j) {

if (nums == null) {

return;

}

int tmp = nums[i];

nums[i] = nums[j];

nums[j] = tmp;

}

}

public class Solution {

public void mergeSortedArray(int[] A, int m, int[] B, int n) {

if (B == null || n == 0) {

return;

}

//assume A is not null, and that A.length >= m + n

int i = m - 1; //pointer for A

int j = n - 1; //pointer for B

int k = m + n - 1; //pointer for combined array

//fill the combined array backwards

while (k >= 0) {

if (i >= 0 && j >= 0) {

A[k--] = A[i] > B[j] ? A[i--] : B[j--];

} else {

A[k--] = i >= 0 ? A[i--] : B[j--];

}

}

}

}

public class Solution {

public ArrayList<Long> productExcludeItself(ArrayList<Integer> A) {

ArrayList<Long> ret = new ArrayList<Long>();

if (A == null || A.size() == 0) {

return ret; //assume return empty list when input size is 0

}

ArrayList<Long> cumProdArray = new ArrayList<Long>();

long cumProd = 1;

ret = new ArrayList<Long>(A.size()); //initialize an empty list of capacity A.size()

//traverse forward

for (int i = 0; i < A.size() - 1; i++) {

cumProd \*= A.get(i);

cumProdArray.add(cumProd);

}

//traverse backward

cumProd = 1;

for (int i = A.size() - 1; i >= 1; i--) {

ret.add(0, cumProd \* cumProd.get(i - 1));

cumProd \*= A.get(i);

}

ret.add(0, cumProd);

return ret;

}

}

public class Solution {

public int firstMissingPositive(int[] A) {

if (A == null) {

return 1;

}

int n = A.length;

//first convert nonpositive numbers to n + 1

//such that they won’t influence positve to negative conversion

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

if (A[i] <= 0) {

A[i] = n + 1;

}

}

//scan from the start, use abs(element) - 1 as index to convert

//element being pointed to to a negative value

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

int abs = Math.abs(A[i]);

if (abs <= n) {

A[abs - 1] = -Math.abs(A[abs - 1]);

}

}

//output first index whose element is positive

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

if (A[i] > 0) {

return i + 1;

}

}

return n + 1;

}

}

public class Solution {

public int threeSumClosest(int[] numbers, int target) {

if (numbers == null || numbers.length < 3) {

return 0; //assume return 0 for null or array size is less than 3

}

int ret = numbers[0] + numbers[1] + numbers[2];

Arrays.sort(numbers);

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

int cur = numbers[i];

int left = i + 1;

int right = numbers.length - 1;

//use two pointers to scan the array

//when sum is smaller, move left pointer to right

//when sum is larger, move right pointer to left

while (left < right) {

int curSum = numbers[left] + numbers[right] + cur;

if (Math.abs(curSum - target) < Math.abs(ret - target)) {

ret = curSum;

}

if (curSum == target) {

return target;

} else if (curSum > target) {

right--; //larger value is on the right

} else {

left++; //smaller value is on the left

}

}

}

return ret;

}

}

public class Solution {

public int firstMissingPositive(int [] A) {

if (A == null || A.length == 0) {

return 1;

}

int n = A.length;

Set<Integer> seen = new HashSet<Integer>();

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

seen.add(A[i]);

}

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

if (!seen.contains(i + 1)) {

return i + 1;

}

}

return n + 1;

}

}

public class Solution {

private void exch(int[] A, int i, int j) {

int tmp = A[i];

A[i] = A[j];

A[j] = tmp;

}

public int firstMissingPositive(int[] A) {

if (A == null || A.length == 0) {

return 1;

}

//swap each element to the position pointed by itself

//suppose index starts at 1

int n = A.length;

int i = 0;

while (i < n) {

if (A[i] == i + 1 || A[i] <= 0 || A[i] > n || A[i] == A[A[i] - 1]) {

i++;

} else {

exch(A, i, A[i] - 1);

}

}

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

if (A[i] != i + 1) {

return i + 1;

}

}

return n + 1;

}

}

public class Solution {

public ArrayList<ArrayList<Integer>> threeSum(int[] numbers) {

ArrayList<ArrayList<Integer>> ret = new ArrayList<ArrayList<Integer>> ();

if (numbers == null || numbers.length < 3) {

return ret;

}

Arrays.sort(numbers); //array must be sorted so we can search for solution

int n = numbers.length;

//think about duplicates

//[0,0,0,0]

//[-1, -1, -1, 0, 0, 1, 1, 1]

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

//skip duplicate solutions containing same numbers[i]

if (i != 0 && numbers[i] == numbers[i - 1]) {

continue;

}

int cur = numbers[i];

int left = i + 1;

int right = n - 1;

while (left < right) {

int curSum = cur + numbers[left] + numbers[right];

if (curSum < 0) {

left++;

} else if (curSum > 0) {

right--;

} else {

ArrayList<Integer> sol = new ArrayList<Integer> ();

sol.add(cur);

sol.add(numbers[left]);

sol.add(numbers[right]);

ret.add(sol);

//duplicates may exist

while (right > left && numbers[right] == numbers[right - 1]) {

right--;

}

while (left < right && numbers[left + 1] == numbers[left]) {

left++;

}

right--;

left++;

}

}

}

return ret;

}

}

public class Solution {

public int[] twoSum(int[] numbers, int target) {

int[] ret = new int[2];

if (numbers == null || numbers.length < 2) {

return ret;

}

int n = numbers.length;

//assume one and only one solution exists for all input

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

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

complement.put(target - numbers[i], i + 1);

}

//scan input again, look for the complement

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

int cur = numbers[i];

if (complement.containsKey(cur)) {

int complementIdx = complement.get(cur);

if (i + 1 == complementIdx) {

continue; //for case like [4, 4], 8

} else {

//ascending order

ret[0] = Math.min(i + 1, complementIdx);

ret[1] = Math.max(i + 1, complementIdx);

break; //only one solution may exist

}

}

}

return ret;

}

}

public class Solution {

public int[] quickSort(int[] array) {

if (array == null) {

return array;

}

quickSort(array, 0, array.length - 1);

return array;

}

private void quickSort(int[] a, int lo, int hi) {

//lo: lower boundary (inclusive)

//hi: upper boundary (inclusive)

if (lo >= hi) {

return;

}

int pivot = partition(a, lo, hi);

quickSort(a, lo, pivot - 1);

quickSort(a, pivot + 1, hi);

}

private int partition(int[] a, int lo, int hi) {

int pivot = (int) (Math.random() \* (hi - lo + 1)) + lo;

exch(a, pivot, lo);

int i = lo; //left pointer

int j = hi + 1; //right pointer

while (true) {

while(a[++i] < a[lo]) {

if (i == hi) {

break;

}

}

while(a[lo] < a[--j]) {

if (j == lo) {

break;

}

}

if (i >= j) {

break;

} else {

exch(a, i, j);

}

}

exch(a, lo, j);

return j;

}

private void exch(int[] a, int i, int j) {

int tmp = a[i];

a[i] = a[j];

a[j] = tmp;

}

}

public class Solution {

/\*\*

\* @param n: Given the range of numbers

\* @param k: Given the numbers of combinations

\* @return: All the combinations of k numbers out of 1..n

\*/

public List<List<Integer>> combine(int n, int k) {

// write your code here

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

if (n <= 0 || k <= 0 || n < k) {

return ret;

}

if (k == 1) {

for (int i = 1; i <= n; i++) {

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

sol.add(i);

ret.add(sol);

}

return ret;

}

if (n == k) {

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

for (int i = 1; i <= n; i++) {

sol.add(i);

}

ret.add(sol);

return ret;

}

//now solve the problem in two scenarios

//solutions with n, solutions without n

List<List<Integer>> solWithoutN = combine(n - 1, k);

ret.addAll(solWithoutN);

List<List<Integer>> solWithN = combine(n - 1, k - 1);

for (List<Integer> l : solWithN) {

l.add(n);

}

ret.addAll(solWithN);

return ret;

}

}

public class Solution {

public List<List<Integer>> combine(int n, int k) {

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

if (n <= 0 || k <= 0 || n < k) {

return ret;

}

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

combine(ret, cur, 1, n, k);

return ret;

}

/\*

\* use DFS to traverse recursion tree, use ret to store all results

\* use cur to store solution from root to current node

\*/

private void combine(List<List<Integer>> ret, List<Integer> cur, int start, int n, int k) {

if (cur.size() == k) {

ret.add(new ArrayList(cur)); //must initialize a new list

//otherwise we are just inserting a duplicate element many times

return;

}

for (int i = start; i <= n; i++) {

cur.add(i);

combine(ret, cur, i + 1, n, k);

cur.remove(cur.size() - 1); //remove last element added

}

}

}

public class Solution {

public List<List<Integer>> combinationSum(int[] candidates, int target) {

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

if (candidates == null || candidates.length == 0 || target <= 0) {

return ret; //all input numbers are positive

}

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

Arrays.sort(candidates); //sorting is critical

combinationSum(ret, cur, target, candidates, 0, candidates.length - 1);

return ret;

}

private void combinationSum(List<List<Integer>> ret, List<Integer> cur, int t, int[] a, int lo, int hi) {

if (t == 0) {

//found a solution

ret.add(new ArrayList(cur));

return;

}

if (lo > hi || a[lo] > t) {

//numbers are in ascending order

//if the smallest is larger than t, then no need for further action

return;

}

//two possible solutions

//with a[lo]

cur.add(a[lo]);

combinationSum(ret, cur, t - a[lo], a, lo, hi);

cur.remove(cur.size() - 1);

//without a[lo]

combinationSum(ret, cur, t, a, lo + 1, hi);

}

}

public class Solution {

//this solution may result in stackoverflow

public ArrayList<DirectedGraphNode> topSort(ArrayList<DirectedGraphNode> graph) {

ArrayList<DirectedGraphNode> ret = new ArrayList<DirectedGraphNode> ();

if (graph == null || graph.size() == 0) {

return ret;

}

//assume no cycle exists

topSort(ret, graph);

return ret;

}

private void topSort(ArrayList<DirectedGraphNode> ret, ArrayList<DirectedGraphNode> g) {

if (g.size() == 1) {

//add the root node

ret.add(0, g.get(0));

return;

}

//add all nodes with no children to head of ret

//then remove these nodes from g, and also remove them from the neighbors

ArrayList<DirectedGraphNode> newG = new ArrayList<DirectedGraphNode> ();

for (DirectedGraphNode n : g) {

if (n.neighbors == null || n.neighbors.size() == 0) {

ret.add(0, n);

} else {

newG.add(n);

}

}

for (DirectedGraphNode n : newG) {

//update neighbors, worst case time complexity O(n^2)

n.neighbors.removeAll(ret);

}

g = newG;

topSort(ret, g);

}

}

public class Solution {

//this solution is bottom up

public ArrayList<DirectedGraphNode> topSort(ArrayList<DirectedGraphNode> graph) {

ArrayList<DirectedGraphNode> ret = new ArrayList<DirectedGraphNode> ();

if (graph == null || graph.size() <= 0) {

return ret;

}

//assume no cycle exists

//assume every node has a unique label, otherwise we can create one

//for them

topSort(ret, graph);

return ret;

}

private void topSort(ArrayList<DirectedGraphNode> ret, ArrayList<DirectedGraphNode> g) {

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

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

Map<Integer, DirectedGraphNode> labelNode = new HashMap<Integer, DirectedGraphNode> ();

countChildren(g, childCount);

findParents(g, parents);

linkLabel(g, labelNode);

while(!childCount.isEmpty()) {

ArrayList<Integer> cur = new ArrayList<Integer> ();

for (Integer i : childCount.keySet()) {

if (childCount.get(i) == 0) {

cur.add(i);

ret.add(0, labelNode.get(i)); //always add to head

//because we are going backwards

}

}

//remove nodes with no children from childCount

for (Integer i : cur) {

childCount.remove(i);

}

//update childCount

for (Integer i : cur) {

for (DirectedGraphNode p : parents.get(i)) {

childCount.put(p.label, childCount.get(p.label) - 1);

}

}

}

}

/\*count number of children for each node\*/

private void countChildren(ArrayList<DirectedGraphNode> g, Map<Integer, Integer> childCount) {

for (DirectedGraphNode n : g) {

//assume no duplicate in input

assert(!childCount.containsKey(n.label));

childCount.put(n.label, n.neighbors == null ? 0 : n.neighbors.size());

}

}

private void findParents(ArrayList<DirectedGraphNode> g, Map<Integer, ArrayList<DirectedGraphNode>> parents) {

for (DirectedGraphNode n : g) {

//for every node, we create empty list for them

parents.put(n.label, new ArrayList<DirectedGraphNode> ());

}

for (DirectedGraphNode n : g) {

assert(n.neighbors != null);

for (DirectedGraphNode child : n.neighbors) {

assert(parents.containsKey(child.label));

parents.get(child.label).add(n);

}

}

}

private void linkLabel(ArrayList<DirectedGraphNode> g, Map<Integer, DirectedGraphNode> labelNode) {

for (DirectedGraphNode n : g) {

assert(!labelNode.containsKey(n.label));

labelNode.put(n.label, n);

}

}

}

public class Solution {

//top down solution

public ArrayList<DirectedGraphNode> topSort(ArrayList<DirectedGraphNode> graph) {

ArrayList<DirectedGraphNode> ret = new ArrayList<DirectedGraphNode> ();

if (graph == null || graph.size() <= 0) {

return ret;

}

//assume no cycle exists

//assume every node has a unique label, otherwise we can create one

//for them

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

Map<Integer, DirectedGraphNode> labelNode = new HashMap<Integer, DirectedGraphNode> ();

countParents(graph, parentCount); //not graph != g

linkLabel(graph, labelNode);

while(!parentCount.isEmpty()) {

ArrayList<Integer> cur = new ArrayList<Integer> ();

for (Integer i : parentCount.keySet()) {

if (parentCount.get(i) == 0) {

cur.add(i);

ret.add(labelNode.get(i)); //always add to tail

//because we are going from roots

}

}

//remove nodes with no parent from parentCount

for (Integer i : cur) {

parentCount.remove(i);

}

//update parentCount

for (Integer i : cur) {

for (DirectedGraphNode c : labelNode.get(i).neighbors) {

parentCount.put(c.label, parentCount.get(c.label) - 1);

}

}

}

return ret;

}

/\*count number of parent for each node\*/

private void countParents(ArrayList<DirectedGraphNode> g, Map<Integer, Integer> parentCount) {

//initialization

for (DirectedGraphNode n : g) {

parentCount.put(n.label, 0);

}

for (DirectedGraphNode n : g) {

//assume no duplicate in input

assert(n.neighbors != null);

for (DirectedGraphNode c : n.neighbors) {

parentCount.put(c.label, parentCount.get(c.label) + 1);

}

}

}

private void linkLabel(ArrayList<DirectedGraphNode> g, Map<Integer, DirectedGraphNode> labelNode) {

for (DirectedGraphNode n : g) {

assert(!labelNode.containsKey(n.label));

labelNode.put(n.label, n);

}

}

}

public class Solution {

//top down solution with BFS

public ArrayList<DirectedGraphNode> topSort(ArrayList<DirectedGraphNode> graph) {

ArrayList<DirectedGraphNode> ret = new ArrayList<DirectedGraphNode> ();

if (graph == null || graph.size() <= 0) {

return ret;

}

//assume no cycle exists

//assume every node has a unique label, otherwise we can create one

//for them

Map<Integer, Integer> parentCount = countParents(graph); //not graph != g

Map<Integer, DirectedGraphNode> labelNode = linkLabel(graph);

//here we do not record level #, so BFS doesn’t have to be level-order

/\*

//level-order BFS

Queue<DirectedGraphNode> toVisit = new LinkedList<DirectedGraphNode>();

for (Integer i : parentCount.keySet()) {

if (parentCount.get(i) == 0) { //add head to toVisit

toVisit.offer(labelNode.get(i));

}

}

while(!toVisit.isEmpty()) {

int curLevelSize = toVisit.size();

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

DirectedGraphNode n = toVisit.poll();

ret.add(n);

//update parentCount

assert(n.neighbors != null);

for (DirectedGraphNode child : n.neighbors) {

parentCount.put(child.label, parentCount.get(child.label) - 1);

if (parentCount.get(child.label) == 0) {

//it is guaranteed that each child is put into toVisit only once

toVisit.offer(child);

}

}

}

} \*/

//non-level-order BFS

Queue<DirectedGraphNode> toVisit = new LinkedList<DirectedGraphNode>();

for (Integer i : parentCount.keySet()) {

if (parentCount.get(i) == 0) {

toVisit.offer(labelNode.get(i));

}

}

while (!toVisit.isEmpty()) {

DirectedGraphNode cur = toVisit.poll();

ret.add(cur);

//assume neighbors not null

for (DirectedGraphNode n : cur.neighbors) {

parentCount.put(n.label, parentCount.get(n.label) - 1);

if (parentCount.get(n.label) == 0) {

toVisit.offer(n);

}

}

}

return ret;

}

/\*count number of parent for each node\*/

private Map<Integer, Integer> countParents(ArrayList<DirectedGraphNode> g) {

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

//initialization

for (DirectedGraphNode n : g) {

parentCount.put(n.label, 0);

}

for (DirectedGraphNode n : g) {

//assume no duplicate in input

assert(n.neighbors != null);

for (DirectedGraphNode c : n.neighbors) {

parentCount.put(c.label, parentCount.get(c.label) + 1);

}

}

return parentCount;

}

private Map<Integer, DirectedGraphNode> linkLabel(ArrayList<DirectedGraphNode> g) {

Map<Integer, DirectedGraphNode> labelNode = new HashMap<Integer, DirectedGraphNode>();

for (DirectedGraphNode n : g) {

assert(!labelNode.containsKey(n.label));

labelNode.put(n.label, n);

}

return labelNode;

}

}

public class Solution {

/\*\*

\* @param start, a string

\* @param end, a string

\* @param dict, a set of string

\* @return an integer

\*/

public int ladderLength(String start, String end, Set<String> dict) {

// write your code here

if (start.equals(end)) {

return 2;

} else if (dict.size() == 0) {

return 0;

}

//bfs

Deque<String> toVisit = new LinkedList<String> ();

int lvl = 0;

Set<String> dictCopy = cloneDict(dict); //we will remove visisted points, so make a copy to retain original input

//Set has no clone method

toVisit.offerLast(start);

assert(!dictCopy.contains(end)); //assume end is not in dict

dictCopy.add(end);

dictCopy.remove(start);

while (!toVisit.isEmpty()) {

lvl++;

int currentLevelSize = toVisit.size();

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

String cur = toVisit.pollFirst();

//WRONG: dictCopy.remove(cur);

//results will be different depending on traversal order

if (cur.equals(end)) {

return lvl;

}

addChildren(cur, toVisit, dictCopy);

}

}

return 0;

}

private Set<String> cloneDict(Set<String> dict) {

Set<String> copy = new HashSet<String> ();

for (String i : dict) {

copy.add(i);

}

return copy;

}

/\*check and add all unvisited children that are in dictCopy to toVisit \*/

private void addChildren(String cur, Deque<String> toVisit, Set<String> dictCopy) {

ArrayList<String> toRemove = new ArrayList<String> ();

for (String candidate : dictCopy) { //Set supports iterator and enhancer for!

if (canTransform(candidate, cur)) { //use canTransform rather than

//construct all possible children

toVisit.offerLast(candidate);

toRemove.add(candidate);

}

}

for (String s : toRemove) {

dictCopy.remove(s);

}

}

private boolean canTransform(String a, String b){

if(a.length() != b.length()){

return false;

}

int flag = 0;

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

if(a.charAt(i) != b.charAt(i)){

flag++;

if (flag > 1) {

break;

}

}

}

return (flag == 1);

}

}

public class Solution {

/\*\*

\* @param start, a string

\* @param end, a string

\* @param dict, a set of string

\* @return an integer

\*/

//optimize for large dictionary

public int ladderLength(String start, String end, Set<String> dict) {

// write your code here

if (start.equals(end)) {

return 2;

} else if (dict.size() == 0) {

return 0;

}

//bfs

Deque<String> toVisit = new LinkedList<String> ();

int lvl = 0;

Set<String> dictCopy = cloneDict(dict); //keep original dict

//Set has no clone method

toVisit.offerLast(start);

assert(!dictCopy.contains(end)); //assume end is not in dict

dictCopy.add(end);

dictCopy.remove(start);

while (!toVisit.isEmpty()) {

lvl++;

int currentLevelSize = toVisit.size();

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

String cur = toVisit.pollFirst();

if (cur.equals(end)) {

return lvl;

}

addChildren(cur, toVisit, dictCopy);

}

}

return 0;

}

private Set<String> cloneDict(Set<String> dict) {

Set<String> copy = new HashSet<String> ();

for (String i : dict) {

copy.add(i);

}

return copy;

}

/\*check and add all unvisited children that are in dictCopy to toVisit \*/

private void addChildren(String cur, Deque<String> toVisit, Set<String> dictCopy) {

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

for (char c = 'a'; c <= 'z'; c++) {

String newStr = cur.substring(0, i) + c + cur.substring(i + 1, cur.length());

if (dictCopy.contains(newStr)) {

toVisit.offerLast(newStr);

dictCopy.remove(newStr); //once we know we are gonna

//visit a specific node, it will never be visited again

}

}

}

}

}

//recursion

class Solution {

/\*\*

\* Get all distinct N-Queen solutions

\* @param n: The number of queens

\* @return: All distinct solutions

\* For example, A string '...Q' shows a queen on forth position

\*/

ArrayList<ArrayList<String>> solveNQueens(int n) {

// write your code here

ArrayList<ArrayList<String>> ret = new ArrayList<ArrayList<String>>();

//if (n <= 2) { //should be able to automatically handle n = 1, n = 2

if (n <= 0) {

return ret;

}

ArrayList<String> cur = new ArrayList<String>();

solveNQueens(ret, cur, 0, n);

return ret;

}

private void solveNQueens(ArrayList<ArrayList<String>> ret, ArrayList<String> cur, int start, int n) {

//start means number of queens that have been placed

if (start == n) {

ret.add(new ArrayList(cur));

return;

}

//i denotes position to be examined for current row start

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

//each queen occupies one row, one column, one diagonal

String next = getNextPlacement(cur, i, n);

if (next == null) {

continue;

}

cur.add(next);

solveNQueens(ret, cur, start + 1, n);

cur.remove(cur.size() - 1);

}

}

//see if we can place a queen at position i

//if not, return null

private String getNextPlacement(ArrayList<String> cur, int i, int n) {

for (int j = 0; j < cur.size(); j++) {

int curCol = getCurrentCol(cur.get(j));

//check if i and curCol are same column

if (i == curCol) {

return null;

}

//check if i and curCol are on same diagonal

if ((cur.size() - j) == Math.abs(i - curCol)) {

return null;

}

}

String ret = "";

for (int j = 0; j < n; j++) {

ret += j == i? "Q" : ".";

}

return ret;

}

private int getCurrentCol(String s) {

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

if (s.charAt(i) == 'Q') {

return i;

}

}

return -1; //Q not found

}

};

//iterative

class Solution {

/\*\*

\* Get all distinct N-Queen solutions

\* @param n: The number of queens

\* @return: All distinct solutions

\* For example, A string '...Q' shows a queen on forth position

\*/

ArrayList<ArrayList<String>> solveNQueens(int n) {

// write your code here

ArrayList<ArrayList<String>> ret = new ArrayList<ArrayList<String>>();

if (n <= 0) {

return ret;

}

//use DFS to traverse all possible solutions

Deque<ArrayList<String>> stack = new LinkedList<ArrayList<String>>();

stack.addLast(new ArrayList<String>());

while(!stack.isEmpty()) {

ArrayList<String> cur = stack.removeLast();

if (cur.size() == n) {

//this is the leaf node (one solution)

ret.add(cur);

} else {

//add children

addChildren(stack, cur, n);

}

}

return ret;

}

//add all children to top of stack

private void addChildren(Deque<ArrayList<String>> stack, ArrayList<String> cur, int n) {

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

String next = getNextPlacement(cur, i, n);

if (next == null) {

continue;

}

cur.add(next);

stack.addLast(new ArrayList<String>(cur));

cur.remove(cur.size() - 1);

}

}

//return a string with 'Q' at column i if no conflict with previous placements

//otherwise return null

private String getNextPlacement(ArrayList<String> cur, int i, int n) {

for (int row = 0; row < cur.size(); row++) {

int col = getCol(cur.get(row));

//check if on same column or same diagonal

if ((col == i) || ((cur.size() - row) == Math.abs(col - i))) {

return null;

}

}

String ret = "";

for (int j = 0; j < n; j++) {

ret += j == i? "Q" : ".";

}

return ret;

}

//return position of 'Q' in a string

//return -1 if not found

private int getCol(String s) {

if (s== null) {

return -1;

}

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

if (s.charAt(i) == 'Q') {

return i;

}

}

return -1;

}

};

class Solution {

/\*\*

\* @param S: A set of numbers.

\* @return: A list of lists. All valid subsets.

\*/

public ArrayList<ArrayList<Integer>> subsets(int[] nums) {

// write your code here

ArrayList<ArrayList<Integer>> ret = new ArrayList<ArrayList<Integer>>();

if (nums == null || nums.length == 0) {

ret.add(new ArrayList<Integer>());

return ret; //return list with an empty list when input is empty

}

int[] sorted = new int[nums.length];

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

sorted[i] = nums[i];

}

Arrays.sort(sorted);

return subsets(sorted, sorted.length);

}

private ArrayList<ArrayList<Integer>> subsets(int[] sorted, int n) {

ArrayList<ArrayList<Integer>> ret = new ArrayList<ArrayList<Integer>>();

if (n == 0) {

ret.add(new ArrayList<Integer>());

} else {

ArrayList<ArrayList<Integer>> noLast = subsets(sorted, n - 1);

ArrayList<ArrayList<Integer>> withLast = new ArrayList<ArrayList<Integer>>();

for (ArrayList<Integer> i : noLast) {

ArrayList<Integer> iWithLast = new ArrayList<Integer>(i);

iWithLast.add(sorted[n - 1]);

withLast.add(iWithLast);

}

ret.addAll(noLast);

ret.addAll(withLast);

}

return ret;

}

}

//another recursive method

//tree where nodes have multiple children

class Solution {

/\*\*

\* @param S: A set of numbers.

\* @return: A list of lists. All valid subsets.

\*/

public ArrayList<ArrayList<Integer>> subsets(int[] nums) {

// write your code here

ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

if (nums == null || nums.length == 0) {

result.add(new ArrayList<Integer>());

return result;

}

int[] sorted = new int[nums.length];

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

sorted[i] = nums[i];

}

Arrays.sort(sorted); //to make sure elements are in non-descending order

ArrayList<Integer> current = new ArrayList<Integer>();

subsets(result, sorted, current, 0);

return result;

}

private void subsets(ArrayList<ArrayList<Integer>> result, int[] sorted, ArrayList<Integer> current, int index) {

result.add(new ArrayList<Integer>(current));

for (int i = index; i < sorted.length; i++) {

//assume all unique elements

current.add(sorted[i]);

subsets(result, sorted, current, i + 1);

current.remove(current.size() - 1);

}

}

}

//straight recursion

class Solution {

/\*\*

\* @param S: A set of numbers.

\* @return: A list of lists. All valid subsets.

\*/

public ArrayList<ArrayList<Integer>> subsetsWithDup(ArrayList<Integer> S) {

// write your code here

ArrayList<ArrayList<Integer>> ret = new ArrayList<ArrayList<Integer>>();

if (S == null || S.size() == 0) {

ret.add(new ArrayList<Integer>());

return ret; //return list with an empty list when input is empty

}

int[] sorted = new int[S.size()];

for (int i = 0; i < S.size(); i++) {

sorted[i] = S.get(i);

}

Arrays.sort(sorted);

return subsetsWithDup(sorted, sorted.length);

}

private ArrayList<ArrayList<Integer>> subsetsWithDup(int[] sorted, int n) {

ArrayList<ArrayList<Integer>> ret = new ArrayList<ArrayList<Integer>>();

if (n == 0) {

ret.add(new ArrayList<Integer>());

} else {

int dupCount = 1;

for (int i = n - 2; i >= 0; i--) {

if (sorted[i + 1] == sorted[i]) {

dupCount++;

} else {

break;

}

}

ArrayList<ArrayList<Integer>> noLast = subsetsWithDup(sorted, n - dupCount);

ArrayList<ArrayList<Integer>> withLast = new ArrayList<ArrayList<Integer>>();

for (ArrayList<Integer> i : noLast) {

ArrayList<Integer> iWithLast = new ArrayList<Integer>(i);

for (int j = 0; j < dupCount; j++) {

iWithLast.add(sorted[n - 1]);

withLast.add(new ArrayList(iWithLast));

}

}

ret.addAll(noLast);

ret.addAll(withLast);

}

return ret;

}

}

//another recursive method

//tree with multiple children

class Solution {

/\*\*

\* @param S: A set of numbers.

\* @return: A list of lists. All valid subsets.

\*/

public ArrayList<ArrayList<Integer>> subsetsWithDup(ArrayList<Integer> S) {

// write your code here

ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

if (S == null || S.size() == 0) {

result.add(new ArrayList<Integer>());

return result;

}

int[] sorted = new int[S.size()];

for (int i = 0; i < S.size(); i++) {

sorted[i] = S.get(i);

}

Arrays.sort(sorted); //to make sure elements are in non-descending order

ArrayList<Integer> current = new ArrayList<Integer>();

subsetsWithDup(result, sorted, current, 0);

return result;

}

private void subsetsWithDup(ArrayList<ArrayList<Integer>> result, int[] sorted, ArrayList<Integer> current, int index) {

result.add(new ArrayList<Integer>(current));

for (int i = index; i < sorted.length; i++) {

//assume all unique elements

if (i > index && sorted[i] == sorted[i - 1]) {

continue;

} else {

current.add(sorted[i]);

subsetsWithDup(result, sorted, current, i + 1);

current.remove(current.size() - 1);

}

}

}

}

//use DFS

//tree has n branches for each node

class Solution {

/\*\*

\* @param S: A set of numbers.

\* @return: A list of lists. All valid subsets.

\*/

public ArrayList<ArrayList<Integer>> subsetsWithDup(ArrayList<Integer> S) {

// write your code here

ArrayList<ArrayList<Integer>> ret = new ArrayList<ArrayList<Integer>>();  
 if (S == null || S.size() == 0) {

ret.add(new ArrayList<Integer>());

return ret;

}

int[] sorted = new int[S.size()];

for (int i = 0; i < S.size(); i++) {

sorted[i] = S.get(i);

}

Arrays.sort(sorted);

ArrayList<Integer> cur = new ArrayList<Integer>();

subsetsWithDup(ret, cur, sorted, 0, sorted.length - 1);

return ret;

}

private void subsetsWithDup(ArrayList<ArrayList<Integer>> ret, ArrayList<Integer> cur, int[] sorted, int lo, int hi) {

ret.add(new ArrayList<Integer>(cur));

for (int i = lo; i <= hi; i++) {

//if (i > 0 && sorted[i] == sorted[i - 1]) {

//we do not examine elements that are same as sorted[lo - 1]

if (i > lo && sorted[i] == sorted[i - 1]) {

continue;

}

cur.add(sorted[i]);

subsetsWithDup(ret, cur, sorted, lo + 1, hi);

cur.remove(cur.size() - 1);

}

}

}

class Solution {

/\*\*

\* @param nums: A list of integers.

\* @return: A list of permutations.

\*/

//time complexity: 1! + 2! + 3! + ...

public ArrayList<ArrayList<Integer>> permute(ArrayList<Integer> nums) {

// write your code here

ArrayList<ArrayList<Integer>> ret = new ArrayList<ArrayList<Integer>>();

if (nums == null || nums.size() == 0) {

return ret;

}

ret.add(new ArrayList<Integer>());

for (int i = 0; i < nums.size(); i++) {

//determine number to be inserted

Integer cur = nums.get(i);

ArrayList<ArrayList<Integer>> insertedList = new ArrayList<ArrayList<Integer>>();

//insert the number to all possible positions, there are i + 1 possible positions

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

if (j != i) {

//create new list for insertion

for (ArrayList<Integer> curL : ret) {

ArrayList<Integer> insertedCurL = new ArrayList<Integer>(curL);

insertedCurL.add(j, cur);

insertedList.add(insertedCurL);

}

} else {

//last position, insert in place

for (ArrayList<Integer> curL : ret) {

curL.add(j, cur);

}

}

}

ret.addAll(insertedList);

}

return ret;

}

}

//recursive method for permutation

//without a temporary list

class Solution {

/\*\*

\* @param nums: A list of integers.

\* @return: A list of permutations.

\*/

public ArrayList<ArrayList<Integer>> permute(ArrayList<Integer> nums) {

// write your code here

ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

if (nums == null || nums.size() == 0) {

return result;

}

Integer[] numsArray = nums.toArray(new Integer[0]);

permute(result, numsArray, 0);

return result;

}

private void permute(ArrayList<ArrayList<Integer>> result, Integer[] numsArray, int start) {

if (start == numsArray.length) {

result.add(new ArrayList<Integer>(Arrays.asList(numsArray)));

} else {

//iterate over all elements

//choose every one as leading element

for (int i = start; i < numsArray.length; i++) {

swap(numsArray, start, i);

permute(result, numsArray, start + 1);

swap(numsArray, start, i);

}

}

}

private void swap(Integer[] a, int i, int j) {

Integer tmp = a[i];

a[i] = a[j];

a[j] = tmp;

}

}

//BFS, iterative

class Solution {

/\*\*

\* @param nums: A list of integers.

\* @return: A list of unique permutations.

\*/

public ArrayList<ArrayList<Integer>> permuteUnique(ArrayList<Integer> nums) {

// write your code here

ArrayList<ArrayList<Integer>> ret = new ArrayList<ArrayList<Integer>>();

if (nums == null || nums.size() == 0) {

return ret;

}

ret.add(new ArrayList<Integer>());

for (int i = 0; i < nums.size(); i++) {

//determine number to be inserted

int cur = nums.get(i);

ArrayList<ArrayList<Integer>> insertedList = new ArrayList<ArrayList<Integer>>();

//insert the number at all possible positions

//for unique numbers there are i + 1 possible positions

//for duplicate numbers, only insert on left side

for (ArrayList<Integer> curL : ret) {

for (int j = 0; j <= curL.size(); j++) {

if (j > 0 && curL.get(j - 1) == cur) {

break;

} else {

//create new list for insertion

ArrayList<Integer> insertedCurL = new ArrayList<Integer>(curL);

insertedCurL.add(j, cur); //only add at left side

insertedList.add(insertedCurL);

}

}

}

ret = insertedList;

}

return ret;

}

}

//DFS, recursive

class Solution {

/\*\*

\* @param nums: A list of integers.

\* @return: A list of unique permutations.

\*/

public ArrayList<ArrayList<Integer>> permuteUnique(ArrayList<Integer> nums) {

// write your code here

ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

if (nums == null || nums.size() == 0) {

return result;

}

Integer[] numsArray = nums.toArray(new Integer[0]);

Arrays.sort(numsArray); //sorting is necessary

permuteUnique(result, numsArray, 0);

return ret;

}

private void permuteUnique(ArrayList<ArrayList<Integer>> result, Integer[] numsArray, int start) {

if (start == numsArray.length) {

result.add(new ArrayList<Integer>(Arrays.asList(numsArray)));

} else {

for (int i = start; i < numsArray.length; i++) {

if (i > start && (numsArray[i].equals(numsArray[i - 1]) || numsArray[i].equals(numsArray[start])) {

//why do have to look at not only the one before

//but also the one pointed to by start?

//because the original assumption that array is sorted

//may not hold after swapping and a recursive call

continue;

} else {

swap(numsArray, i, start);

permuteUnique(result, numsArray, start + 1);

swap(numsArray, i, start);

}

}

}

}

private void swap(Integer[] a, int i, int j) {

Integer tmp = a[i];

a[i] = a[j];

a[j] = tmp;

}

}

//BFS, iterative

public class Solution {

/\*\*

\* @param start, a string

\* @param end, a string

\* @param dict, a set of string

\* @return a list of lists of string

\*/

public List<List<String>> findLadders(String start, String end, Set<String> dict) {

// write your code here

List<List<String>> result = new ArrayList<List<String>>();

assert(start != null && end != null);

assert((!dict.contains(start)) && (!dict.contains(end)));

if (start.equals(end)) {

List<String> startEnd = new ArrayList<String>();

startEnd.add(start);

startEnd.add(end);

result.add(startEnd);

return result;

}

Queue<List<String>> q = new LinkedList<List<String>>();

List<String> firstList = new ArrayList<String>();

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

boolean foundAnswer = false;

firstList.add(start);

q.offer(firstList);

visited.add(start); //make sure start won't be visited again

while(!q.isEmpty()) {

int currentLevelSize = q.size();

Set<String> childrenSameLevel = new HashSet<String>();

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

List<String> current = q.poll();

String word = current.get(current.size() - 1);

List<String> children = getChildren(word, visited, dict);

for (String child : children) {

if (child.equals(end)) {

foundAnswer = true;

List<String> shortestPath = new ArrayList<String>(current);

shortestPath.add(end);

result.add(shortestPath);

break; //end shows up at most once for each word

} else {

List<String> intermediatePath = new ArrayList<String>(current);

intermediatePath.add(child);

q.offer(intermediatePath);

childrenSameLevel.add(child);

}

}

}

if (foundAnswer) {

break;

}

for (String child : childrenSameLevel) {

visited.add(child); //make sure no duplicate visit from different levels

}

}

return result;

}

//return all child words in dict that have not been visited before

private List<String> getChildren(String word, Set<String> visited, Set<String> dict) {

List<String> children = new ArrayList<String>();

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

for (char c = 'a'; c <= 'z'; c++) {

String child = word.substring(0, i) + c + word.substring(i + 1, word.length());

if (dict.contains(child) && (!visited.contains(child))) {

children.add(child);

}

}

}

return children;

}

}