Search In Rotated Array

Suppose a sorted array is rotated at some pivot unknown to you beforehand.
(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2). You are given a target value to search. If found in the array return its index, otherwise return -1. No duplicate exists in the array.

```
public class Solution {
    public int search(int[] A, int target) {
        int result = -1;
        if(A == null || A.length == 0)
            return result;
        for(int i = 0; i < A.length; i++){
            if (A[i] == target){
                result = i;
                break;
            }
        }
        return result;
    }
}//run-time: O(N)
```

```
public class Solution {
    public int search(int[] A, int target) {
        int left = 0, right = A.length - 1;
        while (left <= right){
            int middle = (left + right) / 2;
            if (target == A[middle])
                return middle;
            if (A[middle] < A[right]){
                if (target > A[middle] && target <= A[right])
                    left = middle + 1;
                else
                    right = middle - 1;
            }
            else{
                if (target < A[middle] && target >= A[left])
                    right = middle - 1;
                else
                    left = middle + 1;
            }
        }
        return -1;
    }
}//run-time: O(logN)
```

Search In Rotated Sorted Array II

Follow up for "Search in Rotated Sorted Array": What if duplicates are allowed? Would this affect the run-time complexity? How and why? Write a function to determine if a given target is in the array.

```
public class Solution {
    public boolean search(int[] A, int target) {
        if (A == null || A.length == 0)
            return false;
        for (int i = 0; i < A.length; i++){
            if (A[i] == target){
                return true;
            }
        }
        return false;
    }
}//run-time: O(N)
```

```
public class Solution {
    public boolean search(int[] A, int target) {
        int left = 0, right = A.length - 1;
        while (left <= right){
            int middle = (left + right) / 2;
            if (target == A[middle])
                return true;
            if (A[middle] > A[left]){
                if (A[middle] > target && A[left] <= target)
                    right = middle - 1;
                else
                    left = middle + 1;
            }
            else if (A[middle] < A[left]){
                if (A[middle] < target && A[right] >= target)
                    left = middle + 1;
                else
                    right = middle - 1;
            }
            else
                left ++;
        }
        return false;
    }
}//run-time: O(N)
```

Subsets

Given a set of distinct integers, S, return all possible subsets.

Note: Elements in a subset must be in non-descending order.
The solution set must not contain duplicate subsets.

```
public class Solution {
    public ArrayList<ArrayList<Integer>> subsets(int[] S) {
        ArrayList<ArrayList<Integer>> subset = new ArrayList<ArrayList<Integer>>();
        if (S == null || S.length == 0)
            return subset;
        Arrays.sort(S);//Sorts the specified array into ascending numerical order.
        for (int i = 0; i < S.length; i++){
            ArrayList<ArrayList<Integer>> temp = new ArrayList<ArrayList<Integer>>();
            for (ArrayList<Integer> a: subset){
                temp.add(new ArrayList<Integer>(a));
            }
            for (ArrayList<Integer> a: temp){
                a.add(S[i]);
            }
            ArrayList<Integer> single = new ArrayList<Integer>();
            single.add(S[i]);
            temp.add(single);
            //could not just add S[i] since S[i] is int, but temp is ArrayList<Integer>
            subset.addAll(temp);
        }
        subset.add(new ArrayList<Integer>());
        return subset;
    }
}
```

Subsets II

Given a collection of integers that might contain duplicates, S, return all possible subsets.

```
public class Solution {
    public ArrayList<ArrayList<Integer>> subsetsWithDup(int[] num) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        if (num == null || num.length == 0)
            return result;
        Arrays.sort(num);
        ArrayList<ArrayList<Integer>> temp = new ArrayList<ArrayList<Integer>>();
        for (int i = num.length - 1; i >= 0; i--){
            //get the existing array in result
            if (i == num.length - 1 || num[i] != num[i + 1] || temp.size() == 0){
                temp = new ArrayList<ArrayList<Integer>>();
                for (int j = 0; j < result.size(); j++){
                    temp.add(new ArrayList<Integer>(result.get(j)));
                }
            }
            //add one element to temp
            for (ArrayList<Integer> a: temp)
                a.add(0, num[i]);
            //add the new single number to the temp
            if (i == num.length - 1 || num[i] != num[i + 1]){
            //have to write "i == num.length - 1" at first, or the runtime will be exceeded
                ArrayList<Integer> test = new ArrayList<Integer>();
                test.add(num[i]);
                temp.add(test);
            }
            //add all the arrays in the temp into result
            for (ArrayList<Integer> a: temp){
                result.add(new ArrayList(a));
            }
        }
        //add the empty block at the end of the arrayList
        result.add(new ArrayList<Integer>());
        return result;
    }
}
```

Combination Sum

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T. The same repeated number may be chosen from C unlimited number of times. Note: All numbers (including target) will be positive integers.

Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ … ≤ ak).

The solution set must not contain duplicate combinations.

For example, given candidate set 2,3,6,7 and target 7, A solution set is: [7] [2, 2, 3]

```
public class Solution {
    public ArrayList<ArrayList<Integer>> combinationSum(int[] candidates, int target) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        if (candidates == null || candidates.length == 0)
            return result;
        Arrays.sort(candidates);
        ArrayList<Integer> temp = new ArrayList<Integer>();
        combinationSum(result, temp, candidates, target, 0, 0);
        return result;
    }
    public void combinationSum(ArrayList<ArrayList<Integer>> result, ArrayList<Integer> temp, int[]
candidates, int target, int step, int sum){
        if (sum == target){
            if (!result.contains(temp))
                result.add(new ArrayList<Integer>(temp));
            return;
        }
        if (sum > target)
            return;
        for (int i = step; i < candidates.length; i++){
            temp.add(candidates[i]);
            combinationSum(result, temp, candidates, target, i, sum + candidates[i]);
            temp.remove(temp.size() - 1);//what's that for?
        }
    }
}
```

Combination Sum II

Given a collection of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T. Each number in C may only be used once in the combination. Note: All numbers (including target) will be positive integers. Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ … ≤ ak). The solution set must not contain duplicate combinations. For example, given candidate set 10,1,2,7,6,1,5 and target 8, A solution set is: [1, 7] [1, 2, 5] [2, 6] [1, 1, 6]

```
public class Solution {
    public ArrayList<ArrayList<Integer>> combinationSum2(int[] num, int target) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        if (num == null || num.length == 0)
            return result;
        Arrays.sort(num);
        ArrayList<Integer> temp = new ArrayList<Integer>();
        combinationSum2(num, target, 0, result, temp);
        return result;
    }
    public void combinationSum2(int[] num, int target, int step, ArrayList<ArrayList<Integer>> result,
ArrayList<Integer> temp){
        if (target == 0){
            result.add(new ArrayList<Integer>(temp));
            return;
        }
        if (target < 0 || step >= num.length)
            return;
        for (int i = step; i < num.length; i++){
            //avoid the duplicate solution of array
            if (i > step && num[i] == num[i - 1])
                continue;
            temp.add(num[i]);
            combinationSum2(num, target - num[i], i + 1, result, temp);
            temp.remove(temp.size() - 1);
        }
    }
}
```

Valid Sudoko

Determine if a Sudoku is valid. The Sudoku board could be partially filled, where empty cells are filled with the character '.'.

```
public class Solution {
    public boolean isValidSudoku(char[][] board) {
        //test whether each row is valid or not
        for (int i = 0; i < board.length; i++){
            HashSet<Character> test = new HashSet<Character>();
            for (int j = 0; j < board[0].length; j++){
                if (board[i][j] != '.' && !test.add(board[i][j]))
          //Adds the specified element to this set if it is not already present.return boolean
          //could not use contains, which could just check if it is already present, but could not add into it
                    return false;
            }
        }
        //test whether each column is valid or not
        for (int i = 0; i < board[0].length; i++){
            HashSet<Character> test = new HashSet<Character>();
            for (int j = 0; j < board.length; j++){
                if (board[j][i] != '.' && !test.add(board[j][i]))
                    return false;
            }
        }
        //test whether each block is valid or not
        for (int i = 0; i < 3; i ++){//could not use i += 3
            for (int j = 0; j < 3; j ++){
                HashSet<Character> test = new HashSet<Character>();
                for (int k = i * 3; k < i * 3 + 3; k++){
                    for (int l = j * 3; l < j * 3 + 3; l++){
                        if (board[k][l] != '.' && !test.add(board[k][l]))
                            return false;
                    }
                }
            }
        }
        return true;
    }
}
```

Sudoko Solver

Write a program to solve a Sudoku puzzle by filling the empty cells. Empty cells are indicated by the character '.'.

```
public class Solution {
    public void solveSudoku(char[][] board) {
        solveSudoku(board, 0, 0);
    }
    public boolean solveSudoku(char[][] board, int i, int j){
        if (j >= 9)
            return solveSudoku(board, i + 1, 0);
        if (i == 9)
            return true;
        if (board[i][j] == '.'){
            for (int k = 1; k <= 9; k++){
                board[i][j] = (char)(k + '0');
                if (isValid(board, i, j)){
                    if (solveSudoku(board, i, j + 1))
                        return true;
                }
                board[i][j] = '.';
            }
        }
        else{
            return solveSudoku(board, i, j + 1);
        }
        return false;
    }
    public boolean isValid(char[][] board, int i, int j){
        for (int k = 0; k < 9; k++){
            if (k != j && board[i][k] == board[i][j])
                return false;
        }
        for (int k = 0; k < 9; k++){
            if (k != i && board[k][j] == board[i][j])
                return false;
        }
        for (int row = i / 3 * 3; row < i / 3 * 3 + 3; row++){
            for (int col = j / 3 * 3; col < j / 3 * 3 + 3; col++){
                if (row != i && col != j && board[row][col] == board[i][j])
                    return false;
            }
        }
        return true;
    }
}
```

Largest Rectangle In Histogram

Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram. For example, Given height = [2,1,5,6,2,3], return 10.

```
public class Solution {
    public int largestRectangleArea(int[] height) {
        int area = 0;
        Stack<Integer> stack = new Stack<Integer>();
        for (int i = 0; i < height.length; i++){
            if (stack.isEmpty() || height[stack.peek()] < height[i])
                stack.push(i);
            else{
                int start = stack.pop();
                int width = stack.isEmpty() ? i : i - stack.peek() - 1;
                area = Math.max(area, height[start] * width);
                i--;
            }
        }
        //for the last index case
        while(!stack.isEmpty()){
            int start = stack.pop();
            int width = stack.isEmpty() ? height.length : height.length - stack.peek() - 1;
            area = Math.max(area, height[start] * width);
        }
        return area;
    }
}
//runtime: O(N)
//find the height which is smaller than the left one, then calculate the area before this height to find the max area
```

Maximal Rectangle

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing all ones and return its area.

```
public class Solution {
    public int maximalRectangle(char[][] matrix) {
        int row = matrix.length;
        if (row == 0)
            return 0;
        int col = matrix[0].length;//have to write here not with the row since the boundary case should be defined first
        int[][] ones = new int[row][col];
        //the ones matrix is used to count the continuous number of 1 in each row
        for (int i = 0; i < row; i++){
            for (int j = 0; j < col; j++){
                if (matrix[i][j] == '1'){
                    if (j == 0)
                        ones[i][j] = 1;
                    else
                        ones[i][j] = ones[i][j - 1] + 1;
                }
            }
        }
        int max = 0;
        for (int i = 0; i < row; i++){
            for (int j = 0; j < col; j++){
                int minHeight = i;
                int minWidth = ones[i][j];
                while(minHeight >= 0){
                    //have to consider the upper row value
                    minWidth = Math.min(minWidth, ones[minHeight][j]);
                    int area = minWidth * (i - minHeight + 1);
                    max = Math.max(max, area);
                    minHeight--;
                }
            }
        }
        return max;
    }
}
```

N Queens

The n-queens puzzle is the problem of placing n queens on an n×n chessboard such that no two queens attack each other. Given an integer n, return all distinct solutions to the n-queens puzzle.
Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space respectively.

```
public class Solution {
    public ArrayList<String[]> solveNQueens(int n) {
        ArrayList<String[]> result = new ArrayList<String[]>();
        int[] location = new int[n]; //use array to present the location of Q
        solveNQueens(n, result, location, 0);
        return result;
    }
    public void solveNQueens(int n, ArrayList<String[]> result, int[] location, int current){
        if (current == n)
            printBoard(location, result, n);
        else{
            for (int i = 0; i < n; i++){
                location[current] = i;
                if (isValid(location, current))
                    solveNQueens(n, result, location, current + 1);
            }
        }
    }
    // N queens rule: no two queens share the same row, column, or diagonal
    public boolean isValid(int[] location, int current){
        for (int i = 0; i < current; i++){
            if (location[i] == location[current] || Math.abs(location[current] - location[i]) == (current - i))
                return false;
        }
        return true;
    }
    public void printBoard(int[] location, ArrayList<String[]> result, int n){
        String[] ans = new String[n];
        for (int i = 0; i < n; i++){
            String row = new String();
            for (int j = 0; j < n; j++){
                if (location[i] == j)
                    row += "Q";
                else
                    row += ".";
            }
            ans[i] = row;
```

```
            }
            result.add(ans);
        }
}
```

N Queens II

Follow up for N-Queens problem. Return the total number of distinct solutions.

```
public class Solution {
    int count;
    public int totalNQueens(int n) {
        count = 0;
        int[] location = new int[n];
        totalNQueens(n, location, 0);
        return count;
    }
    public void totalNQueens(int n, int[] location, int current){
        if (current == n){
            count ++;
            return;
        }
        else{
            for (int i = 0; i < n; i++){
                location[current] = i;
                if (isValid(location, current))
                    totalNQueens(n, location, current + 1);
            }
        }
    }
    public boolean isValid(int[] location, int current){
        for (int i = 0; i < current; i++){
            if (location[i] == location[current] || Math.abs(location[i] - location[current]) == (current - i))
                return false;
        }
        return true;
    }
}
```

Spiral Matrix

Given a matrix of m x n elements (m rows, n columns), return all elements of the matrix in spiral order.
For example, Given the following matrix: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]. You should
return [1,2,3,6,9,8,7,4,5].

```
public class Solution {
    public ArrayList<Integer> spiralOrder(int[][] matrix) {
        ArrayList<Integer> spiral = new ArrayList<Integer>();
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0)
            return spiral;
        int row = matrix.length, col = matrix[0].length;
        int x = 0, y = 0;
        while (row > 0 && col > 0){
            if (row == 1){
                for (int i = 0; i < col; i++)
                    spiral.add(matrix[x][y++]);
                break;
            }
            if (col == 1){
                for (int i = 0; i < row; i++)
                    spiral.add(matrix[x++][y]);
                break;
            }
            for (int i = 0; i < col - 1; i++)
                spiral.add(matrix[x][y++]);
            for (int i = 0; i < row - 1; i++)
                spiral.add(matrix[x++][y]);
            for (int i = 0; i < col - 1; i++)
                spiral.add(matrix[x][y--]);
            for (int i = 0; i < row - 1; i++)
                spiral.add(matrix[x--][y]);
            x ++;
            y ++;
            row -= 2;
            col -= 2;
        }
        return spiral;
    }
}
```

Next Permutation

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers. If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order). The replacement must be in-place, do not allocate extra memory. Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column. 1,2,3 → 1,3,2; 3,2,1 → 1,2,3; 1,1,5 → 1,5,1

```
public class Solution {
    public void nextPermutation(int[] num) {
        if (num.length <= 1)
            return;
        for (int i = num.length - 2; i >= 0; i--){
            if (num[i] < num[i + 1]){
                int j;
                for (j = num.length - 1; j >= i + 1; j--){
                    if (num[i] < num[j])
                        break;
                }
                //swap two integers in binary, using XOR
                num[i] = num[i] ^ num[j];
                num[j] = num[i] ^ num[j];
                num[i] = num[i] ^ num[j];
                Arrays.sort(num, i + 1, num.length);
            //Sorts the specified range of the array into ascending order.(int fromIndex, int toIndex)
                return;
            }
        }
        for (int i = 0; i < num.length / 2; i++){
            int temp = num[i];
            num[i] = num[num.length - i - 1];
            num[num.length - i - 1] = temp;
        }
        return;
    }
}
```

Palindrome Partitioning

Given a string s, partition s such that every substring of the partition is a palindrome. Return all possible palindrome partitioning of s. For example, given s = "aab", Return [["aa","b"], ["a","a","b"]]

```
public class Solution {
    public ArrayList<ArrayList<String>> partition(String s) {
        ArrayList<ArrayList<String>> result = new ArrayList<ArrayList<String>>();
        if (s == null || s.length() == 0)
            return result;
        ArrayList<String> temp = new ArrayList<String>();
        partition(s, result, temp, 0);
        return result;
    }
    public void partition(String s, ArrayList<ArrayList<String>> result, ArrayList<String> temp, int start){
        if (start == s.length()){
            result.add(new ArrayList<String>(temp));
            return;
        }
        for (int i = start + 1; i <= s.length(); i++){
            String subString = s.substring(start, i);
            if (isPalindrome(subString)){
                temp.add(subString);
                partition(s.substring(i), result, temp, start);
                temp.remove(temp.size() - 1);
            }
        }
    }
    public boolean isPalindrome(String s){
        int left = 0, right = s.length() - 1;
        while (left < right){
            if (s.charAt(left) != s.charAt(right))
                return false;
            left ++;
            right --;
        }
        return true;
    }
}
```

Palindrome Partitioning II

Given a string s, partition s such that every substring of the partition is a palindrome.Return the minimum cuts needed for a palindrome partitioning of s. For example, given s = "aab", Return 1 since the palindrome partitioning ["aa","b"] could be produced using 1 cut.

```
public class Solution {
    public int minCut(String s) {
        int min = 0;
        if (s == null || s.length() == 0)
            return min;
        int[] cuts = new int[s.length() + 1];
        //s.substring(i, j) is palindrome if matrix[i][j] == true
        boolean[][] matrix = new boolean[s.length()][s.length()];
        for (int i = 0; i < s.length(); i++)
            //the max number of cuts for each s.substring(i)
            cuts[i] = s.length() - i;
        for (int i = s.length() - 1; i >= 0; --i){
            for (int j = i; j < s.length(); ++j){
                if ((s.charAt(i) == s.charAt(j) && (j - i < 2)) || (s.charAt(i) == s.charAt(j) && matrix[i +
1][j - 1])){
                    //two situations
                    //1: i and j are neighbors or the same
                    //2: the chars between i and j are also palindrome
                    matrix[i][j] = true;
                    cuts[i] = Math.min(cuts[i], cuts[j + 1] + 1);
                }
            }
        }
        min = cuts[0];
        return min - 1;
    }
}
```

Longest Palindromic Substing

Given a string S, find the longest palindromic substring in S. You may assume that the maximum length of S is 1000, and there exists one unique longest palindromic substring.

```
public class Solution {
    public String longestPalindrome(String s) {
        if (s.isEmpty())
            return null;
        if (s.length() == 1)
            return s;
        String maxLen = s.substring(0, 1);
        for (int i = 0; i < s.length(); i++){
            //the case that the string is palindrome with i
            String temp = longestPalindrome(s, i, i);
            if (temp.length() > maxLen.length())
                maxLen = temp;
            //the case that the string is palindrome with i & i + 1
            temp = longestPalindrome(s, i, i + 1);
            if (temp.length() > maxLen.length())
                maxLen = temp;
        }
        return maxLen;
    }
    public String longestPalindrome(String s, int begin, int end){
        while (begin >= 0 && end < s.length() && s.charAt(begin) == s.charAt(end)){
            begin--;
            end++;
        }
        return s.substring(begin + 1, end);
    }
}
```

Permutation Sequence

The set [1,2,3,…,n] contains a total of n! unique permutations. By listing and labeling all of the permutations in order, We get the following sequence (ie, for n = 3):"123""132""213""231""312""321"
Given n and k, return the kth permutation sequence. Note: Given n will be between 1 and 9 inclusive.

```
public class Solution {
    public String getPermutation(int n, int k) {
        //the total number of sequence for n is n * (n - 1)!
        ArrayList<Integer> num = new ArrayList<Integer>();
        int sum = 1;
        for (int i = 1; i <= n; i++){
            num.add(i);
            sum *= i;
        }
        sum /= n;
        k--;
        StringBuffer temp = new StringBuffer();
        for (int i = 1; i <= n; i++){
            //use step to know the specific group that k is located
            int step = k / sum;
            temp.append(num.get(step));//Returns the element at the specified position in this list.
            num.remove(num.get(step));
            if (i == n)
                break;
            k %= sum;
            sum /= (n - i);
        }
        return temp.toString();
    }
}
```

First Missing Positive

Given an unsorted integer array, find the first missing positive integer. For example, Given [1,2,0] return 3, and [3,4,-1,1] return 2. Your algorithm should run in O(n) time and uses constant space.

```
public class Solution {
    public int firstMissingPositive(int[] A) {
        Arrays.sort(A);
        int index = 0;
        int n = 1;
        while (index < A.length && A[index] <= 0){//have to set the index < A.length to make the
boundary
            index ++;
        }
        for (int i = index; i < A.length; i++){
            if (i > 0 && A[i] == A[i - 1])//have to set i > 0 to make the boundary
                continue;//consider the duplicate solution in the array
            else if (A[i] != n)
                return n;
            else
                n++;
        }
        return n;
    }
}//runtime: O(NlogN) maybe?
public class Solution {
    public int firstMissingPositive(int[] A) {
        int n = A.length, i = 0;
        while (i < n){
            if (A[i] > 0 && A[i] <= n && A[i] - 1 != i && A[A[i] - 1] != A[i]){//why those situations?
                int temp = A[A[i]-1];
                A[A[i]-1] = A[i];
                A[i] = temp;
            }
            else
                i++;
        }
        for(int j = 0; j < n; j++){
            if(A[j] != j + 1)
                return j + 1;
        }
        return n+1;
    }
}//runtime: O(N)
```

Clone Graph

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.
OJ's undirected graph serialization: Nodes are labeled uniquely.
We use # as a separator for each node, and , as a separator for node label and each neighbor of the node.
As an example, consider the serialized graph {0,1,2#1,2#2,2}.
The graph has a total of three nodes, and therefore contains three parts as separated by #.
First node is labeled as 0. Connect node 0 to both nodes 1 and 2.
Second node is labeled as 1. Connect node 1 to node 2.
Third node is labeled as 2. Connect node 2 to node 2 (itself), thus forming a self-cycle.

```
/**
 * Definition for undirected graph.
 * class UndirectedGraphNode {
 *         int label;
 *         List<UndirectedGraphNode> neighbors;
 *         UndirectedGraphNode(int x) { label = x; neighbors = new ArrayList<UndirectedGraphNode>(); }
 * };
 */
public class Solution {
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
        if (node == null)
            return null;
        HashMap<Integer, UndirectedGraphNode> clone = new HashMap<Integer,
UndirectedGraphNode>();
        return cloneGraph(node, clone);
    }
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode node, HashMap<Integer,
UndirectedGraphNode> clone){
        if (clone.containsKey(node.label)){
            return clone.get(node.label);
        }
        UndirectedGraphNode newNode = new UndirectedGraphNode(node.label);
        clone.put(newNode.label, newNode);
        for (UndirectedGraphNode temp: node.neighbors){
            newNode.neighbors.add(cloneGraph(temp, clone));
        }
        return newNode;
    }
}
```

Word Break

Given a string s and a dictionary of words dict, determine if s can be segmented into a space-separated sequence of one or more dictionary words. For example, given s = "leetcode", dict = ["leet", "code"]. Return true because "leetcode" can be segmented as "leet code".

```
public class Solution {
    public boolean wordBreak(String s, Set<String> dict) {
        //1D dynamic programming
        boolean[] test = new boolean[s.length() + 1];
        test[0] = true;
        for (int i = 0; i < s.length(); i++){
            if (!test[i])
                continue;
            for (String a: dict){
                int len = a.length();
                int end = i + len;
                if (end > s.length())
                    continue;//dict set could larger than s
                if (test[end])
                    continue;
                if (s.substring(i, end).equals(a))
                    test[end] = true;
            }
        }
        return test[s.length()];
    }
}
```