```
public class Recursion {
    //
**************************Recursion*************************************************************
******
    // Generate all binary strings
    class Solution {
        public static ArrayList<String> generateString(int k) {
            // Write your code here.
            ArrayList<String> ls = new ArrayList<>();
            String str = "";
            binary(0, -1, k, ls, str);
            return ls;
        }

        public static void binary(int ind, int prev, int n, ArrayList<String> ls, String temp) {
            if (ind == n) {
                ls.add(temp);
                return;
            }
            binary(ind + 1, 0, n, ls, temp + "0");
            if (prev == -1 || prev == 0) {
                binary(ind + 1, 1, n, ls, temp + "1");
            }
        }
    }

    // Generate Paranthesis
    class Solution {

        public List<String> AllParenthesis(int n) {
            // Write your code here
            List<String> ls = new ArrayList<>();
            f(0, 0, n, ls, "");
            return ls;

        }

        public void f(int open, int close, int n, List<String> ls, String path) {
            if (close == n) {
                ls.add(path);
                return;
            }
            if (open > close) {
                f(open, close + 1, n, ls, path + ")");
            }
            if (open < n) {
                f(open + 1, close, n, ls, path + "(");
            }
        }
```

```
}

// Print all subsequences/Power Set
class Solution {
    public List<String> AllPossibleStrings(String s) {
        // Code here
        List<String> answer = new ArrayList<>();
        int n = s.length();
        int tot = 1 << n;
        for (int num = 0; num < tot; num++) {
            String ans = "";
            for (int i = 0; i < n; i++) {
                int ind = n - 1 - i;
                int check = num & (1 << ind);
                if (check != 0) {
                    ans += s.charAt(i);
                }
            }
            if (!ans.equals(""))
                answer.add(0, ans);
        }
        Collections.sort(answer);
        return answer;
    }
}
// Learn All Patterns of Subsequences …
// Count all subsequences with sum K
class Solution {
    class TUF {
        static int findWaysUtil(int ind, int target, int[] arr, int[][] dp) {
            if (target == 0)
                return 1;
            if (ind == 0)
                return arr[0] == target ? 1 : 0;
            if (dp[ind][target] != -1)
                return dp[ind][target];
            int notTaken = findWaysUtil(ind - 1, target, arr, dp);
            int taken = 0;
            if (arr[ind] <= target)
                taken = findWaysUtil(ind - 1, target - arr[ind], arr, dp);
            return dp[ind][target] = notTaken + taken;
        }

        static int findWays(int[] num, int k) {
            int n = num.length;
            int dp[][] = new int[n][k + 1];
            for (int row[] : dp)
                Arrays.fill(row, -1);
            return findWaysUtil(n - 1, k, num, dp);
```

```
        }
    }

    class TUF {
        static int findWays(int[] num, int k) {
            int n = num.length;
            int[][] dp = new int[n][k + 1];
            for (int i = 0; i < n; i++) {
                dp[i][0] = 1;
            }
            if (num[0] <= k)
                dp[0][num[0]] = 1;
            for (int ind = 1; ind < n; ind++) {
                for (int target = 1; target <= k; target++) {
                    int notTaken = dp[ind - 1][target];
                    int taken = 0;
                    if (num[ind] <= target)
                        taken = dp[ind - 1][target - num[ind]];
                    dp[ind][target] = notTaken + taken;
                }
            }
            return dp[n - 1][k];
        }
    }
}

// Check if there exists a subsequence with sum K
class Solution {
    class TUF {
        static boolean subsetSumUtil(int ind, int target, int[] arr, int[][] dp) {
            if (target == 0)
                return true;
            if (ind == 0)
                return arr[0] == target;
            if (dp[ind][target] != -1)
                return dp[ind][target] == 0 ? false : true;
            boolean notTaken = subsetSumUtil(ind - 1, target, arr, dp);
            boolean taken = false;
            if (arr[ind] <= target)
                taken = subsetSumUtil(ind - 1, target - arr[ind], arr, dp);
            dp[ind][target] = notTaken || taken ? 1 : 0;
            return notTaken || taken;
        }
    }

    class TUF {
        static boolean subsetSumToK(int n, int k, int[] arr) {
            boolean dp[][] = new boolean[n][k + 1];
            for (int i = 0; i < n; i++) {
```

```
                dp[i][0] = true;
            }
            if (arr[0] <= k)
                dp[0][arr[0]] = true;
            for (int ind = 1; ind < n; ind++) {
                for (int target = 1; target <= k; target++) {
                    boolean notTaken = dp[ind - 1][target];
                    boolean taken = false;
                    if (arr[ind] <= target)
                        taken = dp[ind - 1][target - arr[ind]];
                    dp[ind][target] = notTaken || taken;
                }
            }
            return dp[n - 1][k];
        }
    }
}

// Combination Sum
class Solution {
    public void findCombinations(int ind, int arr[], int target, List<List<Integer>> ans,
List<Integer> ds) {
        if (ind == arr.length) {
            if (target == 0) {
                ans.add(new ArrayList<>(ds));
            }
            return;
        }
        if (arr[ind] <= target) {
            ds.add(arr[ind]);
            findCombinations(ind, arr, target - arr[ind], ans, ds);
            ds.remove(ds.size() - 1);
        }
        findCombinations(ind + 1, arr, target, ans, ds);
    }

    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>> ans = new ArrayList<>();
        findCombinations(0, candidates, target, ans, new ArrayList<>());
        return ans;
    }
}

// Combination Sum-II
class Solution {
    public List<List<Integer>> combinationSum2(int[] candidates, int target) {
        List<List<Integer>> ans = new ArrayList<>();
        Arrays.sort(candidates);
        findCombinations(0, candidates, target, ans, new ArrayList<>());
```

```
        return ans;
    }

    static void findCombinations(int ind, int[] arr, int target, List<List<Integer>> ans,
List<Integer> ds) {
        if (target == 0) {
            ans.add(new ArrayList<>(ds));
            return;
        }
        for (int i = ind; i < arr.length; i++) {
            if (i > ind && arr[i] == arr[i - 1]) {
                continue;
            }
            if (arr[i] > target)
                break;
            ds.add(arr[i]);
            findCombinations(i + 1, arr, target - arr[i], ans, ds);
            ds.remove(ds.size() - 1);
        }
    }
}

// Subset Sum-I
class Solution {
    ArrayList<Integer> subsetSums(ArrayList<Integer> arr, int N) {
        // code here
        ArrayList<Integer> res = new ArrayList<Integer>();
        printSubsetSums(arr, res, 0, N, 0);
        return res;
    }

    void printSubsetSums(ArrayList<Integer> arr, ArrayList<Integer> res, int ind, int N,
int sum) {
        if (ind >= N) {
            res.add(sum);
            return;
        }
        printSubsetSums(arr, res, ind + 1, N, sum + arr.get(ind));
        printSubsetSums(arr, res, ind + 1, N, sum);
    }
}

// Subset Sum-II
class Solution {
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        Arrays.sort(nums);
        List<List<Integer>> ansList = new ArrayList<>();
        findSubsets(0, nums, new ArrayList<>(), ansList);
        return ansList;
```

```
        }

        public void findSubsets(int ind, int[] nums, List<Integer> ds, List<List<Integer>>
ansList) {
            ansList.add(new ArrayList<>(ds));
            for (int i = ind; i < nums.length; i++) {
                if (i != ind && nums[i] == nums[i - 1]) {
                    continue;
                }
                ds.add(nums[i]);
                findSubsets(i + 1, nums, ds, ansList);
                ds.remove(ds.size() - 1);
            }
        }
    }

    // Combination Sum - III
    class Solution {
        public List<List<Integer>> combinationSum3(int k, int n) {
            int arr[] = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
            List<List<Integer>> ls = new ArrayList<>();
            List<Integer> ans = new ArrayList<>();
            f(0, 0, k, n, arr, ls, ans);
            return ls;
        }

        public void f(int ind, int curr, int c, int n, int a[], List<List<Integer>> ls, List<Integer>
ans) {
            if (curr == c || ind == a.length) {
                if (n == 0 && ans.size() == c) {
                    ls.add(new ArrayList<>(ans));
                    return;
                }
                return;
            }
            if (a[ind] <= n) {
                ans.add(a[ind]);
                f(ind + 1, curr + 1, c, n - a[ind], a, ls, ans);
                ans.remove(ans.size() - 1);
            }
            f(ind + 1, curr, c, n, a, ls, ans);
        }
    }

    // Letter Combinations of a Phone numb…
    class Solution {
        // Function to find list of all words possible by pressing given numbers.
        static ArrayList<String> possibleWords(int a[], int N) {
            // your code here
```

```
        String keyboard[] = new String[] { "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "
wxyz" };
        ArrayList<String> ls = new ArrayList<>();
        String temp = "";
        getCombinations(0, N, keyboard, ls, temp, a);
        return ls;
    }

    static void getCombinations(int ind, int n, String map[], ArrayList<String> ls, String
temp, int a[]) {
        if (ind == n) {
            ls.add(temp);
            return;
        }
        for (int i = 0; i < map[a[ind] - 2].length(); i++) {
            getCombinations(ind + 1, n, map, ls, temp + map[a[ind] - 2].charAt(i), a);
        }
    }
}
// Palindrome Partitioning
class Solution {
    public List<List<String>> partition(String s) {
        List<List<String>> res = new ArrayList<>();
        List<String> path = new ArrayList<>();
        solve(0, s, path, res);
        return res;
    }

    public void solve(int index, String s, List<String> path, List<List<String>> res) {
        if (index == s.length()) {
            res.add(new ArrayList<>(path));
            return;
        }
        for (int i = index; i < s.length(); i++) {
            if (isPal(s, index, i)) {
                path.add(s.substring(index, i + 1));
                solve(i + 1, s, path, res);
                path.remove(path.size() - 1);
            }
        }
    }

    public boolean isPal(String s, int start, int end) {
        while (start <= end) {
            if (s.charAt(start) != s.charAt(end)) {
                return false;
            }
            start++;
            end--;
```

```
            }
            return true;
        }
    }
    // Word Search
    class Solution {
        public boolean exist(char[][] board, String word) {
            int n = board.length;
            int m = board[0].length;
            int dx[] = new int[] { 0, 1, 0, -1 };
            int dy[] = new int[] { -1, 0, 1, 0 };
            int l = word.length();
            boolean flag = false;
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < m; j++) {
                    if (word.charAt(0) == board[i][j]) {
                        flag = f(i, j, dx, dy, 0, l, board, n, m, word);
                        if (flag == true) {
                            return true;
                        }
                    }
                }
            }
            if (flag == false) {
                return false;
            } else
                return true;
        }

        public boolean f(int row, int col, int dx[], int dy[], int ind, int len, char board[][], int n,
int m,
                String word) {
            if (ind == len - 1) {
                return true;
            }
            // System.out.println("Row: "+row+" Col: "+col+" index: "+ind);
            boolean res = false;
            char origin = board[row][col];
            board[row][col] = '.';
            for (int i = 0; i < 4; i++) {
                int nr = row + dx[i];
                int nc = col + dy[i];
                if (nr >= 0 && nr < n && nc >= 0 && nc < m && word.charAt(ind + 1) ==
board[nr][nc]) {
                    if (f(nr, nc, dx, dy, ind + 1, len, board, n, m, word) == true) {
                        return true;
                    }
                }
            }
```

```
        board[row][col] = origin;
        return res;
    }
}
// N Queen
class Solution {
    public List<List<String>> solveNQueens(int n) {
        List<List<String>> ans = new ArrayList<>();
        char board[][] = new char[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                board[i][j] = '.';
            }
        }
        dfs(0, board, ans);
        return ans;
    }

    public void dfs(int col, char board[][], List<List<String>> ans) {
        // Base Case
        if (col == board.length) {
            ans.add(construct(board));
            return;
        }

        for (int row = 0; row < board.length; row++) {
            if (validate(row, col, board) == true) {
                board[row][col] = 'Q';
                dfs(col + 1, board, ans);
                board[row][col] = '.';
            }
        }
    }

    public boolean validate(int row, int col, char board[][]) {
        int r = row, c = col;
        while (r >= 0 && c >= 0) {
            if (board[r][c] == 'Q') {
                return false;
            }
            c--;
        }
        int r1 = row, c1 = col;
        while (r1 >= 0 && c1 >= 0) {
            if (board[r1][c1] == 'Q') {
                return false;
            }
            r1--;
            c1--;
```

```
            }
            int r2 = row, c2 = col;
            while (r2 < board.length && c2 >= 0) {
                if (board[r2][c2] == 'Q') {
                    return false;
                }
                r2++;
                c2--;
            }
            return true;
        }

        public List<String> construct(char board[][]) {
            List<String> res = new ArrayList<>();
            for (int i = 0; i < board.length; i++) {
                String t = new String(board[i]);
                res.add(t);
            }
            return res;
        }
    }
    // Rat in a Maze
    class Solution {
        public static ArrayList<String> findPath(int[][] m, int n) {
            // Your code here
            ArrayList<String> res = new ArrayList<String>();
            int dx[] = new int[] { 0, 1, 0, -1 };
            int dy[] = new int[] { -1, 0, 1, 0 };
            char ds[] = new char[] { 'L', 'D', 'R', 'U' };
            boolean vis[][] = new boolean[n][n];
            if (m[0][0] == 0) {
                return res;
            }
            dfs(0, 0, vis, m, n, res, "", dx, dy, ds);
            return res;

        }

        public static void dfs(int row, int col, boolean vis[][], int matrix[][], int n,
ArrayList<String> res,
                String temp, int dx[], int dy[], char ds[]) {
            if (row == n - 1 && col == n - 1) {
                res.add(temp);
                return;
            }
            // System.out.println("Path Taken "+temp+" Position:"+"["+row+","+col+"]");
            vis[row][col] = true;
            for (int i = 0; i < 4; i++) {
                int nr = row + dx[i];
```

```
            int nc = col + dy[i];
            if (nr >= 0 && nr < n && nc >= 0 && nc < n && matrix[nr][nc] == 1 &&
vis[nr][nc] == false) {
                dfs(nr, nc, vis, matrix, n, res, temp + ds[i], dx, dy, ds);
            }
        }
        vis[row][col] = false;
    }
  }
  // Word Break
  class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        int n = s.length();
        Set<String> hs = new HashSet<>();
        for (String temp : wordDict) {
            hs.add(temp);
        }
        return f(s, hs);
    }

    public boolean f(String s, Set<String> hs) {
        int wordlen = s.length();
        if (wordlen == 0) {
            return true;
        }
        for (int i = 1; i <= wordlen; i++) {
            if (hs.contains(s.substring(0, i)) && f(s.substring(i, wordlen), hs)) {
                return true;
            }
        }
        return false;
    }
  }
  // M Coloring Problem

  class Solution   {
    // Function to determine if graph can be coloured with at most M colours
    // such
    // that no two adjacent vertices of graph are coloured with same colour.
    public boolean graphColoring(boolean graph[][], int m, int n) {
        int color[] = new int[n];
        for (int i = 0; i < n; i++) {
            color[i] = 0;
        }
        if (graphColoringUtil(graph, m, color, 0, n) == false) {
            return false;
        }
        return true;
    }
```

```
    boolean graphColoringUtil(boolean graph[][], int m, int color[], int ind, int n) {
        if (ind == n) {
            return true;
        }
        for (int c = 1; c <= m; c++) {
            if (isSafe(ind, graph, color, c, n)) {
                color[ind] = c;
                if (graphColoringUtil(graph, m, color, ind + 1, n) == true)
                    return true;
                color[ind] = 0;
            }
        }
        return false;
    }

    boolean isSafe(int ind, boolean graph[][], int color[], int c, int n) {
        for (int i = 0; i < n; i++)
            if (graph[ind][i] && c == color[i]) {
                return false;
            }
        return true;
    }
}
// Sudoko Solver
class Solution {
    public void solveSudoku(char[][] board) {
        sudoku(board);
    }

    public boolean sudoku(char board[][]) {
        int n = board.length;
        int m = board[0].length;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (board[i][j] == '.') {
                    for (char ch = '1'; ch <= '9'; ch++) {
                        if (isValid(i, j, board, ch) == true) {
                            board[i][j] = ch;
                            if (sudoku(board) == true) {
                                return true;
                            } else {
                                board[i][j] = '.';
                            }
                        }
                    }
                    return false;
                }
            }
```

```
        }
        return true;
    }

    public boolean isValid(int row, int col, char board[][], char ch) {
        for (int i = 0; i < 9; i++) {
            if (board[row][i] == ch) {
                return false;
            }
            if (board[i][col] == ch) {
                return false;
            }
            if (board[3 * (row / 3) + (i / 3)][3 * (col / 3) + (i % 3)] == ch) {
                return false;
            }
        }
        return true;
    }
}
// Expression Add Operators
public class Solution {
    public List<String> addOperators(String num, int target) {
        List<String> result = new ArrayList<String>();
        if (num == null || num.length() == 0) {
            return result;
        }
        f(result, "", num, target, 0, 0, 0);
        return result;
    }

    public void f(List<String> result, String path, String num,
            int target, int pos, long eval, long multed) {
        if (pos == num.length()) {
            if (target == eval) {
                result.add(path);
            }
            return;
        }
        for (int i = pos; i < num.length(); i++) {
            if (i != pos && num.charAt(pos) == '0') {
                break;
            }
            long cur = Long.parseLong(num.substring(pos, i + 1));
            if (pos == 0) {
                f(result, path + cur, num, target, i + 1, cur, cur);
            } else {
                f(result, path + "+" + cur, num, target, i + 1, eval + cur, cur);
                f(result, path + "-" + cur, num, target, i + 1, eval - cur, -cur);
                f(result, path + "*" + cur, num, target, i + 1, eval - multed + multed * cur,
```

```
multed * cur);
                }
            }
        }
    }
}
```