Recursion questions and solutions

List of questions on Leetcode

https://leetcode.com/list/ps92gfo5

Given an array nums of distinct integers, return all the possible permutations. You can return the answer in any order.

```
class Solution {
    List<List<Integer>> res = new ArrayList();
    public List<List<Integer>> permute(int[] nums) {
        List<Integer> numbers = new ArrayList();
        for(int x: nums) numbers.add(x);
        dfs(numbers, 0);
        return res;
    }

    void dfs(List<Integer> numbers, int start) {
     if(start == numbers.size()-1) {
      res.add(new ArrayList(numbers));
      return;
     }
     for(int next = start; next < numbers.size(); next++) {
      Collections.swap(numbers, start, next);  // swap
      dfs(numbers, start+1);
      Collections.swap(numbers, start, next);  // un swap
     }
    }
}
```

Given a collection of numbers, nums, that might contain duplicates, return all possible unique permutations in any order.

```
class Solution {
    public List<List<Integer>> permuteUnique(int[] nums) {
        List<Integer> numList  = Arrays.stream( nums ).boxed
().collect( Collectors.toList() );
        Collections.sort(numList);
        List<List<Integer>> res = new ArrayList();
        dfs(numList,0,res);
        return res;
    }
    void dfs(List<Integer> numList, int start, List<List<Inte
ger>> res) {
        if(start == numList.size()-1) {
            res.add(new ArrayList(numList));
            return;
        }
        Set<Integer> set = new HashSet();
        for(int next = start; next < numList.size(); next++)
{
            if(!set.add(numList.get(next))) continue;
            Collections.swap(numList, start, next);
            dfs(numList, start+1, res);
            Collections.swap(numList, start, next);
        }
    }
}
```

Given two integers n and k, return all possible combinations of k numbers chosen from the range [1, n]. You may return the answer in any order.

```
class Solution {
    List<List<Integer>> res = new ArrayList();
```

```
    public List<List<Integer>> combine(int n, int k) {
        dfs(1,n, k, new ArrayList());
        return res;
    }

    void dfs(int start, int n, int k, List<Integer> list) {
        if(k == 0) {
            res.add(new ArrayList(list));
            return;
        }
        for(int next = start; next <= n - k + 1; next++) {
            list.add(next);
            dfs(next+1,n,k-1,list);
            list.remove(list.size()-1);
        }
    }
}
```

Given an array of distinct integers candidates and a target integer target, return a list of all unique combinations of candidates where the chosen numbers sum to target. You may return the combinations in any order.

`The same number may be chosen from candidates an unlimited number of times. Two combinations are unique if the frequency of at least one of the chosen numbers is different.`

```
class Solution {
    public List<List<Integer>> combinationSum(int[] candidate
s, int target) {
        List<List<Integer>> res = new ArrayList();
        dfs(candidates, 0, target, new ArrayList(), res);
        return res;
    }


    void dfs(int[] a, int i, int target, List<Integer> list,
```

```
List<List<Integer>> res) {
        if(i >= a.length) return;

        if(target < 0) return;

        if(target == 0) {
            res.add(new ArrayList<Integer>(list));
            return;
        }
        list.add(a[i]);
        dfs(a,i,target-a[i],list,res);
        list.remove(list.size()-1);
        dfs(a,i+1,target,list,res);
    }
}


// approach 2

class Solution {
    public List<List<Integer>> combinationSum(int[] nums, int
target) {
        List<List<Integer>> list = new ArrayList<>();
        backtrack(list, new ArrayList<>(), nums, target, 0);
        return list;
    }

    private void backtrack(List<List<Integer>> list, List<Int
eger> tempList, int [] nums, int remain, int start){
        if(remain < 0) return;
        else if(remain == 0) list.add(new ArrayList<>(tempLis
t));
        else{
            for(int i = start; i < nums.length; i++){
                tempList.add(nums[i]);
                backtrack(list, tempList, nums, remain - nums
```

```
[i], i); // not i + 1 because we can reuse same elements
                tempList.remove(tempList.size() - 1);
            }
        }
    }
}
```

Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target.

- Each number in candidates may only be used once in the combination.
- Note: The solution set must not contain duplicate combinations.

```
class Solution {
    List<List<Integer>> res = new ArrayList();
    public List<List<Integer>> combinationSum2(int[] candidat
es, int target) {
        Arrays.sort(candidates);
        dfs(candidates,0,target, new ArrayList());
        return res;
    }

    private void dfs(int[] a, int i, int target, List<Integer
> temp) {
        if(target == 0) {
            res.add(new ArrayList(temp));
            return;
        }
        if(i == a.length || target < 0) {
            return;
        }

        for(int j = i; j < a.length; j++) {
            if(j > i && a[j-1] == a[j]) continue;
```

```
            temp.add(a[j]);
            dfs(a,j+1,target-a[j],temp);
            temp.remove(temp.size()-1);
        }

    }
}
```

Find all valid combinations of k numbers that sum up to n such that the following conditions are true:

- Only numbers 1 through 9 are used.
- Each number is used at most once.
- Return a list of all possible valid combinations. The list must not contain the same combination twice, and the combinations may be returned in any order.

```
class Solution {
    List<List<Integer>> res = new ArrayList();
    public List<List<Integer>> combinationSum3(int k, int n)
{
        dfs(k,n,1,new HashSet());
        return res;
    }

    void dfs(int k, int sum, int start, Set<Integer> set) {
        if(k == 0 && sum == 0) {
            res.add(new ArrayList(set));
            return;
        }
        if(k == 0 || sum == 0) return;

        for(int i = start; i <= 9; i++) {
            if(!set.contains(i)) {
                set.add(i);
                dfs(k-1, sum-i,i+1,set);
```

```
                set.remove(i);
            }
        }
    }
}
```

Given an integer array nums of unique elements, return all possible subsets (the power set).

- The solution set must not contain duplicate subsets. Return the solution in any order.

```
class Solution {
    List<List<Integer>> res = new ArrayList<>();
    public List<List<Integer>> subsets(int[] nums) {
        dfs(new ArrayList<>(), nums, 0);
        return res;
    }

    private void dfs(List<Integer> tempList, int [] nums, int
start){
        res.add(new ArrayList<>(tempList));
        for(int i = start; i < nums.length; i++){
            tempList.add(nums[i]);
            dfs(tempList, nums, i + 1);
            tempList.remove(tempList.size() - 1);
        }
    }
}
```

Given an integer array nums that may contain duplicates, return all possible subsets(the power set).

- The solution set must not contain duplicate subsets. Return the solution in any order.

```
class Solution {
    List<List<Integer>> res = new ArrayList<>();
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        Arrays.sort(nums);
        dfs(new ArrayList<>(), nums, 0);
        return res;
    }

    private void dfs(List<Integer> tempList, int [] nums, int
start){
        res.add(new ArrayList<>(tempList));

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

            if(i > start && nums[i] == nums[i-1]) continue;

            tempList.add(nums[i]); // add cur number
            dfs(tempList, nums, i + 1);
            tempList.remove(tempList.size() - 1);
        }
    }
}
```

Given a string s, partition s such that every substring of the partition is a palindrome

- Return all possible palindrome partitioning of s.

```
class Solution {
    List<List<String>> res = new ArrayList();
    boolean[][] isPal;

    public List<List<String>> partition(String s) {
        isPal = new boolean[s.length()][s.length()];
        dfs(s,0,new ArrayList<String>());
```

```
        return res;
    }

    private void dfs(String s, int i, List<String> temp) {
        if(i == s.length()) {
            res.add(new ArrayList<String>(temp));
            return;
        }
        for(int j = i; j < s.length(); j++) {
            if(s.charAt(i) == s.charAt(j) && (j-i < 3 || isPa
l[i+1][j-1]) ) {
                isPal[i][j] = true;
                temp.add(s.substring(i,j+1));
                dfs(s,j+1,temp);
                temp.remove(temp.size()-1);
            }
        }
    }
}
```

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

```
class Solution {
    List<String> res = new ArrayList();
    public List<String> generateParenthesis(int n) {
        dfs("",n,n);
        return res;
    }

    void dfs(String s, int open, int close) {
        if(open == 0 && close == 0) {
            res.add(s);
            return;
        }
```

```
        if(open != 0) {
            dfs(s+"(", open-1, close);
        }
        if(close > open) {
            dfs(s+")", open, close-1);
        }
    }
}
```

Given a string containing digits from 2-9 inclusive, return all possible letter combinations that the number could represent. Return the answer in any order.

- A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

```
class Solution {
    public List<String> letterCombinations(String digits) {
        String[] KEYS = { "", "", "abc", "def", "ghi", "jkl",
"mno", "pqrs", "tuv", "wxyz" };
        LinkedList<String> q = new LinkedList();
        if(digits.length() == 0) return q;
        q.add("");
        for(char d: digits.toCharArray()) {
            int idx = d - '0';
            for(int i = q.size(); i > 0; i--) {
                String prev = q.poll();
                for(char c: KEYS[idx].toCharArray()) {
                    q.add(prev+c);
                }
            }
        }
        return q;
    }
}
```

The n-queens puzzle is the problem of placing n queens on an n x n chessboard such that no two queens attack each other.

- Given an integer n, return all distinct solutions to the n-queens puzzle. You may return the answer in any order.
- Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space, respectively.

```
class Solution {
    // a cell belongs to a column, leftToRightDiag and rightT
oLeftDiag, So these boolean array will be used to check if qu
een is already present or not.
    boolean[] isQueenAtCol, isQueenAtLeftToRightDiag, isQueen
AtRightToLeftDiag;

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

    // this will maintain position of queen for each row
    Map<Integer, Integer> rowToColumMapForQueen = new HashMap
();

    public List<List<String>> solveNQueens(int n) {
        isQueenAtCol = new boolean[n];
        isQueenAtLeftToRightDiag = new boolean[2*n-1];
        isQueenAtRightToLeftDiag = new boolean[2*n-1];
        dfs(0,n);
        return res;
    }

    private void dfs(int curRow, int n) {
        if(curRow == n) {
            List<String> temp = new ArrayList();
            for(int row = 0; row < n; row++) {
                char[] charArray = new char[n];
                Arrays.fill(charArray, '.');
```

```
                charArray[rowToColumMapForQueen.get(row)] =
'Q';
                temp.add(new String(charArray));
            }
            res.add(temp);
            return;
        }
        for(int col = 0; col < n; col++) {
            if (isQueenAtCol[col] ||
                isQueenAtLeftToRightDiag[curRow + col] ||
                isQueenAtRightToLeftDiag[curRow - col + n -
1]) continue;

            // pick an option
            // put queen at cur position
            isQueenAtCol[col] = true;
            isQueenAtLeftToRightDiag[curRow + col] = true;
            isQueenAtRightToLeftDiag[curRow - col + n - 1] =
true;
            rowToColumMapForQueen.put(curRow,col);

            dfs(curRow+1,n);

            isQueenAtCol[col] = false;
            isQueenAtLeftToRightDiag[curRow + col] = false;
            isQueenAtRightToLeftDiag[curRow - col + n - 1] =
false;
            rowToColumMapForQueen.remove(curRow);
        }
    }
}
```

Sudoku solver. Write a program to solve a Sudoku puzzle by filling the empty cells.

- A sudoku solution must satisfy all of the following rules:

- Each of the digits 1-9 must occur exactly once in each row.
- Each of the digits 1-9 must occur exactly once in each column.
- Each of the digits 1-9 must occur exactly once in each of the 9 3×3 sub-boxes of the grid.
- The '.' character indicates empty cells.

```
class Solution {
    class Cell{
        int i, j;
        Cell(int i, int j) {this.i = i;this.j=j;}
    }

    Set<Integer>[] rowValSet = new HashSet[9];
    Set<Integer>[] colValSet = new HashSet[9];
    Set<Integer>[] boxValSet = new HashSet[9];
    List<Cell> emptyCells = new ArrayList();

    public void solveSudoku(char[][] board) {
        init(board);
        dfs(board,0);
    }

    boolean dfs(char[][] board, int start) {
        if(start == emptyCells.size()) {
            return true;
        }
        Cell cell = emptyCells.get(start);
        int k = 3*(cell.i/3) + cell.j/3;
        for(int v = 1; v <= 9; v++) {
            if(rowValSet[cell.i].contains(v) || colValSet[cel
l.j].contains(v) || boxValSet[k].contains(v)) continue;
            int charCode = v+'0';
            board[cell.i][cell.j] = (char) charCode;
            rowValSet[cell.i].add(v);colValSet[cell.j].add
```

```
(v);boxValSet[k].add(v);
            if(dfs(board, start+1)) return true;
            board[cell.i][cell.j] = '.';
            rowValSet[cell.i].remove(v);colValSet[cell.j].rem
ove(v);boxValSet[k].remove(v);
        }
        return false;
    }

    void init(char[][] board) {
        for(int i = 0; i < 9; i++) {
            rowValSet[i] = new HashSet();
            colValSet[i] = new HashSet();
            boxValSet[i] = new HashSet();
        }

        for(int i = 0; i < 9; i++) {
            for(int j = 0; j < 9; j++) {
                if(Character.isDigit(board[i][j])) {
                    rowValSet[i].add(board[i][j]-'0');
                    colValSet[j].add(board[i][j]-'0');
                    int k = 3*(i/3) + j/3;
                    boxValSet[k].add(board[i][j]-'0');
                } else {
                    emptyCells.add(new Cell(i,j));
                }
            }
        }
    }
}
```

Given an m x n grid of characters board and a string word, return true if word exists in the grid.

- The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell

may not be used more than once.

```
class Solution {
    boolean[][] visited;
    int[][] dirs = {{1,0},{0,1},{-1,0},{0,-1}};
    public boolean exist(char[][] board, String word) {
        visited = new boolean[board.length][board[0].length];

        for(int i = 0; i < board.length; i++){
            for(int j = 0; j < board[i].length; j++){
                if((word.charAt(0) == board[i][j]) && dfs(boa
rd, word, i, j, 0)){
                    return true;
                }
            }
        }

        return false;
    }

    private boolean dfs(char[][]board, String word, int i, in
t j, int index){
        if(index == word.length()){
            return true;
        }

        if(i >= board.length || i < 0 || j >= board[i].length
|| j < 0 || board[i][j] != word.charAt(index) || visited[i]
[j]){
            return false;
        }

        visited[i][j] = true;

        for(int[] dir: dirs) {
```

```
            int ni = i + dir[0];
            int nj = j + dir[1];
            if(dfs(board, word, ni, nj, index+1)) return tru
e;
        }

        visited[i][j] = false;
        return false;
    }
}
```