A general approach to backtracking questions in Java (Subsets, Permutations, Combination Sum, Palindrome Partitioning)

This structure might apply to many other backtracking questions, but here I am just going to demonstrate Subsets, Permutations, and Combination Sum.

Subsets : <https://leetcode.com/problems/subsets/>

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

**List**<**List**<Integer>> **list** = **new** ArrayList<>();

Arrays.sort(nums);

backtrack(**list**, **new** ArrayList<>(), nums, 0);

**return** **list**;

}

**private** void backtrack(**List**<**List**<Integer>> **list** , **List**<Integer> tempList, **int** [] nums, **int** start){

**list**.add(**new** ArrayList<>(tempList));

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

tempList.add(nums[i]);

backtrack(**list**, tempList, nums, i + 1);

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

}

}

Subsets II (contains duplicates) : <https://leetcode.com/problems/subsets-ii/>

**public** **List**<**List**<Integer>> subsetsWithDup(**int**[] nums) {

**List**<**List**<Integer>> **list** = **new** ArrayList<>();

Arrays.sort(nums);

backtrack(**list**, **new** ArrayList<>(), nums, 0);

**return** **list**;

}

**private** void backtrack(**List**<**List**<Integer>> **list**, **List**<Integer> tempList, **int** [] nums, **int** start){

**list**.add(**new** ArrayList<>(tempList));

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

**if**(i > start && nums[i] == nums[i-1]) **continue**; // skip duplicates

tempList.add(nums[i]);

backtrack(**list**, tempList, nums, i + 1);

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

}

}

Permutations : <https://leetcode.com/problems/permutations/>

**public** **List**<**List**<Integer>> permute(**int**[] nums) {

**List**<**List**<Integer>> **list** = **new** ArrayList<>();

// Arrays.sort(nums); // not necessary

backtrack(**list**, **new** ArrayList<>(), nums);

**return** **list**;

}

**private** void backtrack(**List**<**List**<Integer>> **list**, **List**<Integer> tempList, **int** [] nums){

**if**(tempList.size() == nums.length){

**list**.add(**new** ArrayList<>(tempList));

} **else**{

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

**if**(tempList.contains(nums[i])) **continue**; // element already exists, skip

tempList.add(nums[i]);

backtrack(**list**, tempList, nums);

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

}

}

}

Permutations II (contains duplicates) : <https://leetcode.com/problems/permutations-ii/>

**public** **List**<**List**<Integer>> permuteUnique(**int**[] nums) {

**List**<**List**<Integer>> **list** = **new** ArrayList<>();

Arrays.sort(nums);

backtrack(**list**, **new** ArrayList<>(), nums, **new** **boolean**[nums.length]);

**return** **list**;

}

**private** void backtrack(**List**<**List**<Integer>> **list**, **List**<Integer> tempList, **int** [] nums, **boolean** [] used){

**if**(tempList.size() == nums.length){

**list**.add(**new** ArrayList<>(tempList));

} **else**{

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

**if**(used[i] || i > 0 && nums[i] == nums[i-1] && !used[i - 1]) **continue**;

used[i] = **true**;

tempList.add(nums[i]);

backtrack(**list**, tempList, nums, used);

used[i] = **false**;

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

}

}

}

Combination Sum : <https://leetcode.com/problems/combination-sum/>

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

**List**<**List**<Integer>> **list** = **new** ArrayList<>();

Arrays.sort(nums);

backtrack(**list**, **new** ArrayList<>(), nums, target, 0);

**return** **list**;

}

**private** void backtrack(**List**<**List**<Integer>> **list**, **List**<Integer> tempList, **int** [] nums, **int** remain, **int** start){

**if**(remain < 0) **return**;

**else** **if**(remain == 0) **list**.add(**new** ArrayList<>(tempList));

**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);

}

}

}

Combination Sum II (can't reuse same element) : <https://leetcode.com/problems/combination-sum-ii/>

**public** **List**<**List**<Integer>> combinationSum2(**int**[] nums, **int** target) {

**List**<**List**<Integer>> **list** = **new** ArrayList<>();

Arrays.sort(nums);

backtrack(**list**, **new** ArrayList<>(), nums, target, 0);

**return** **list**;

}

**private** void backtrack(**List**<**List**<Integer>> **list**, **List**<Integer> tempList, **int** [] nums, **int** remain, **int** start){

**if**(remain < 0) **return**;

**else** **if**(remain == 0) **list**.add(**new** ArrayList<>(tempList));

**else**{

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

**if**(i > start && nums[i] == nums[i-1]) **continue**; // skip duplicates

tempList.add(nums[i]);

backtrack(**list**, tempList, nums, remain - nums[i], i + 1);

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

}

}

}

Palindrome Partitioning : <https://leetcode.com/problems/palindrome-partitioning/>

**public** List<List<String>> partition(String s) {

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

backtrack(list, new ArrayList<>(), s, 0);

**return** list;

}

**public** void backtrack(List<List<String>> list, List<String> tempList, String s, int start){

**if**(start == s.length())

list.add(new ArrayList<>(tempList));

**else**{

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

**if**(isPalindrome(s, start, i)){

tempList.add(s.substring(start, i + 1));

backtrack(list, tempList, s, i + 1);

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

}

}

}

}

**public** boolean isPalindrome(String s, int low, int high){

**while**(low < high)

**if**(s.charAt(low++) != s.charAt(high--)) **return** false;

**return** true;

}