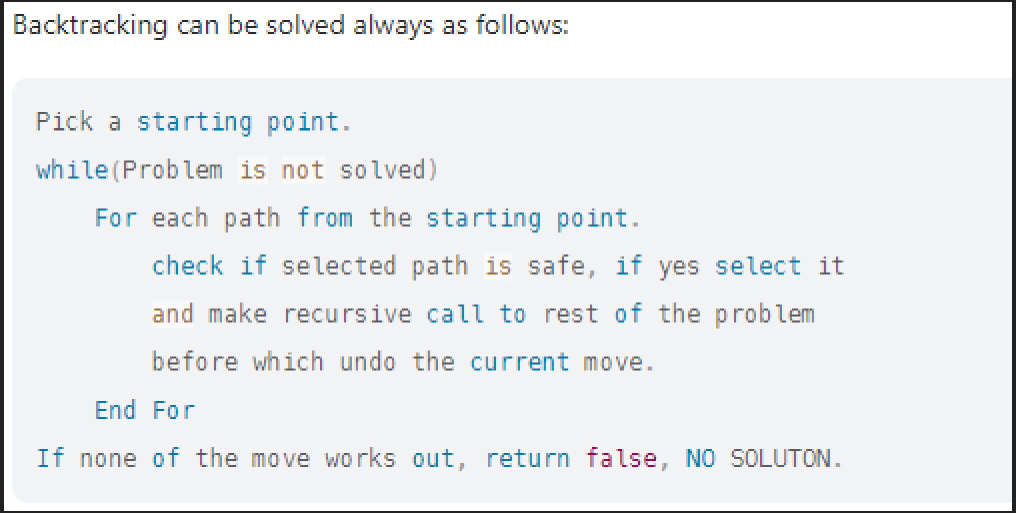
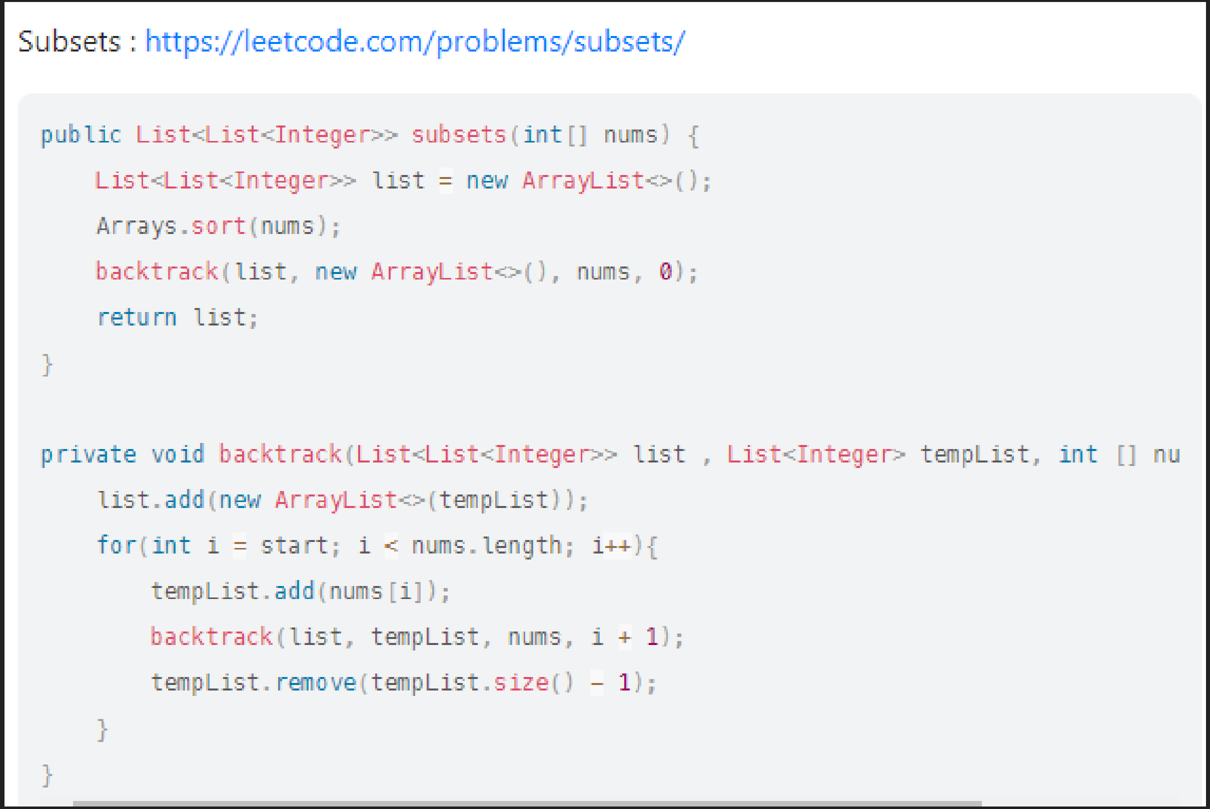
Recursion backtracking


Backtracking can be solved always as follows:

```
Pick a starting point.
while(Problem is not solved)
    For each path from the starting point.
        check if selected path is safe, if yes select it
        and make recursive call to rest of the problem
        before which undo the current move.
    End For
If none of the move works out, return false, NO SOLUTON.
```

Subsets

https://leetcode.com/problems/subsets/



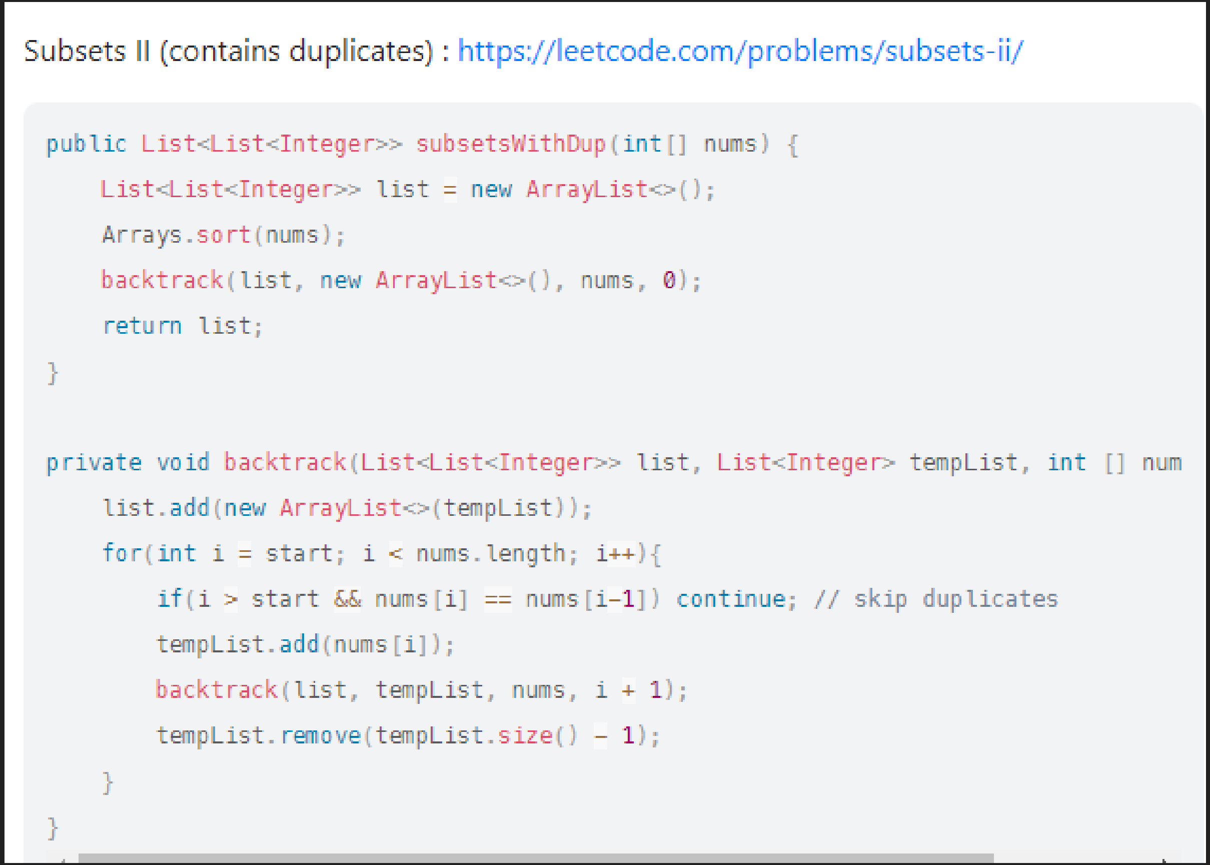
Subsets : https://leetcode.com/problems/subsets/

```
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 [] nu
    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 2

https://leetcode.com/problems/subsets-ii/

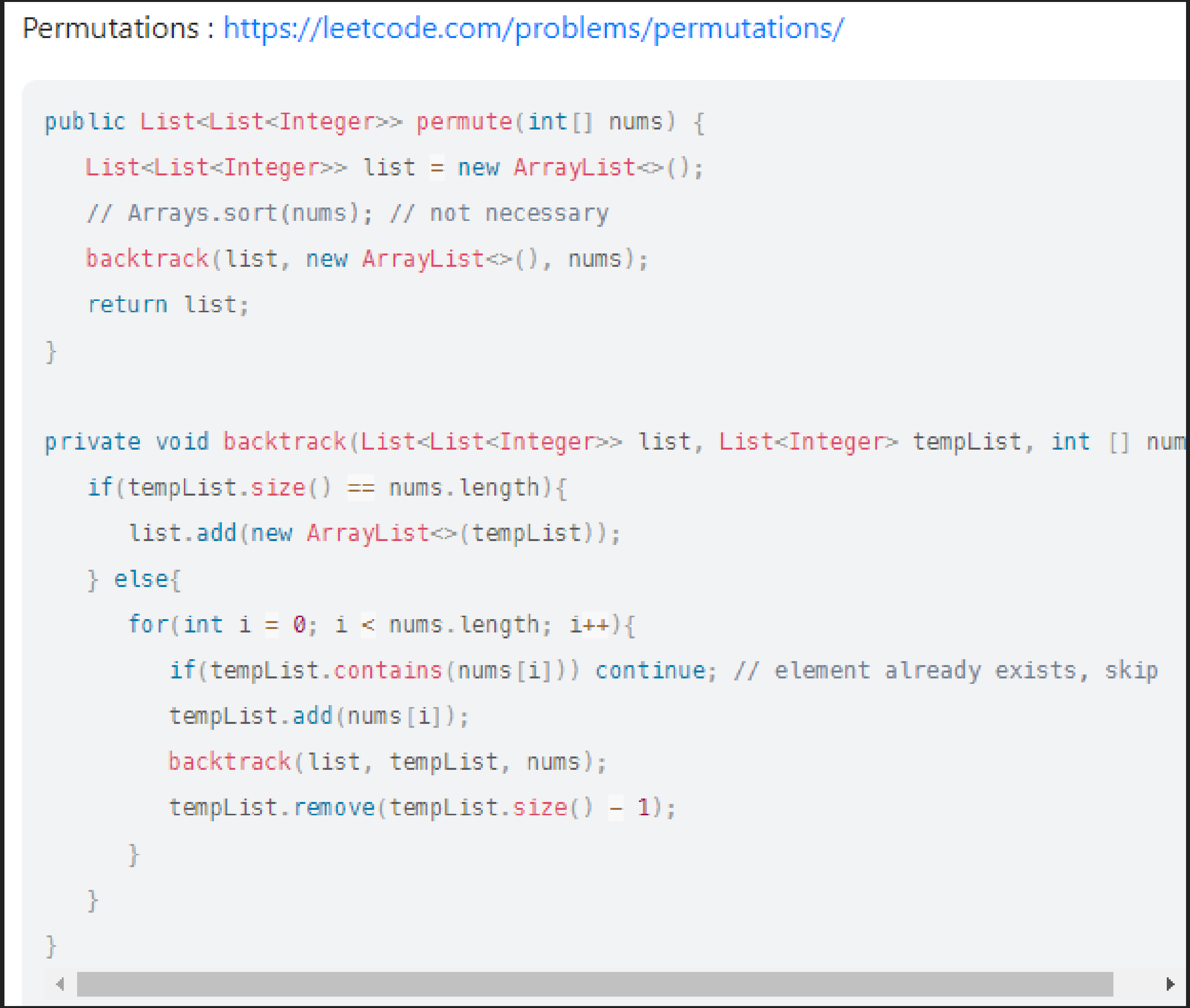

Subsets II (contains duplicates) : https://leetcode.com/problems/subsets-ii/

```
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 [] num
    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/



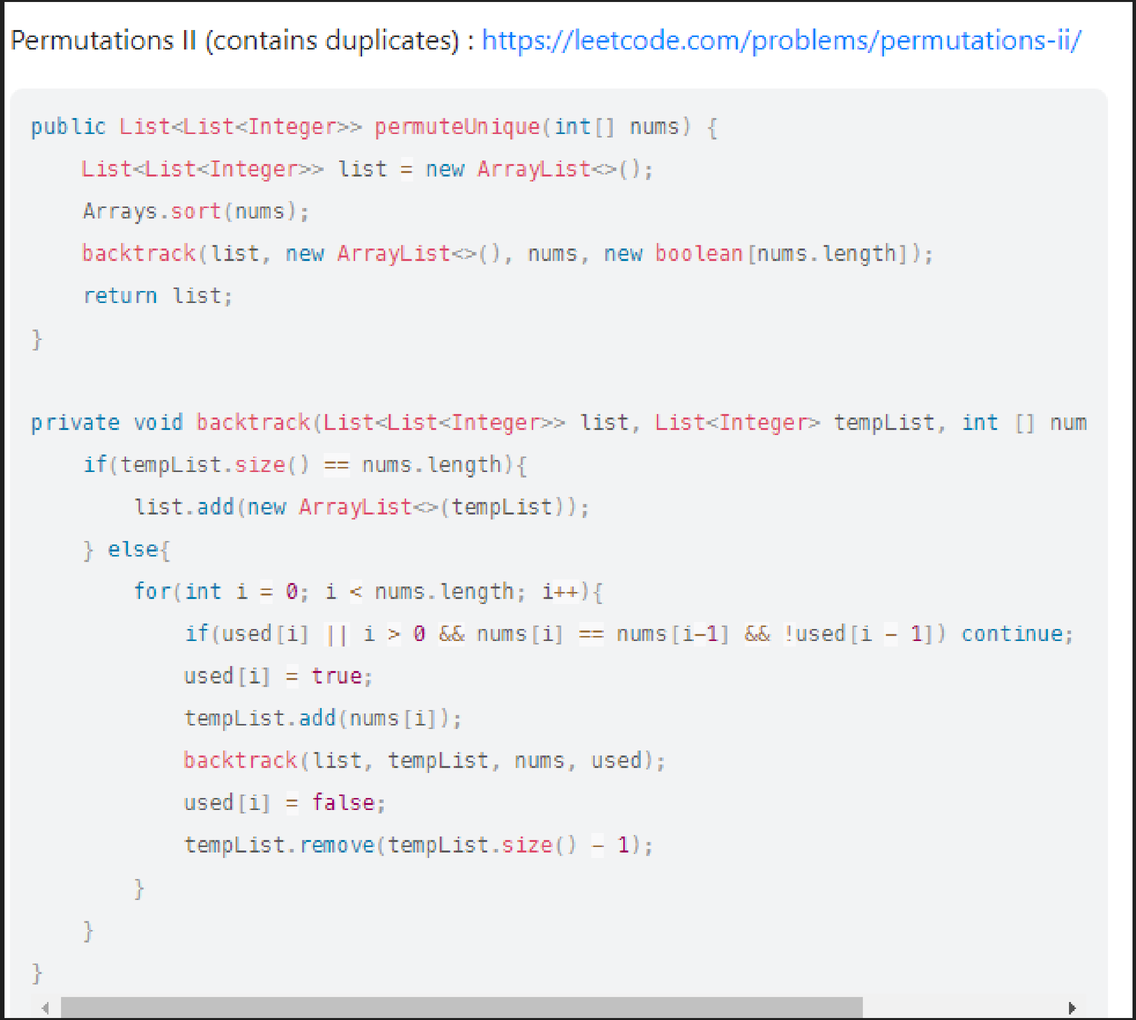
Permutations : https://leetcode.com/problems/permutations/

```
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 [] num
   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 2

https://leetcode.com/problems/permutations-ii/



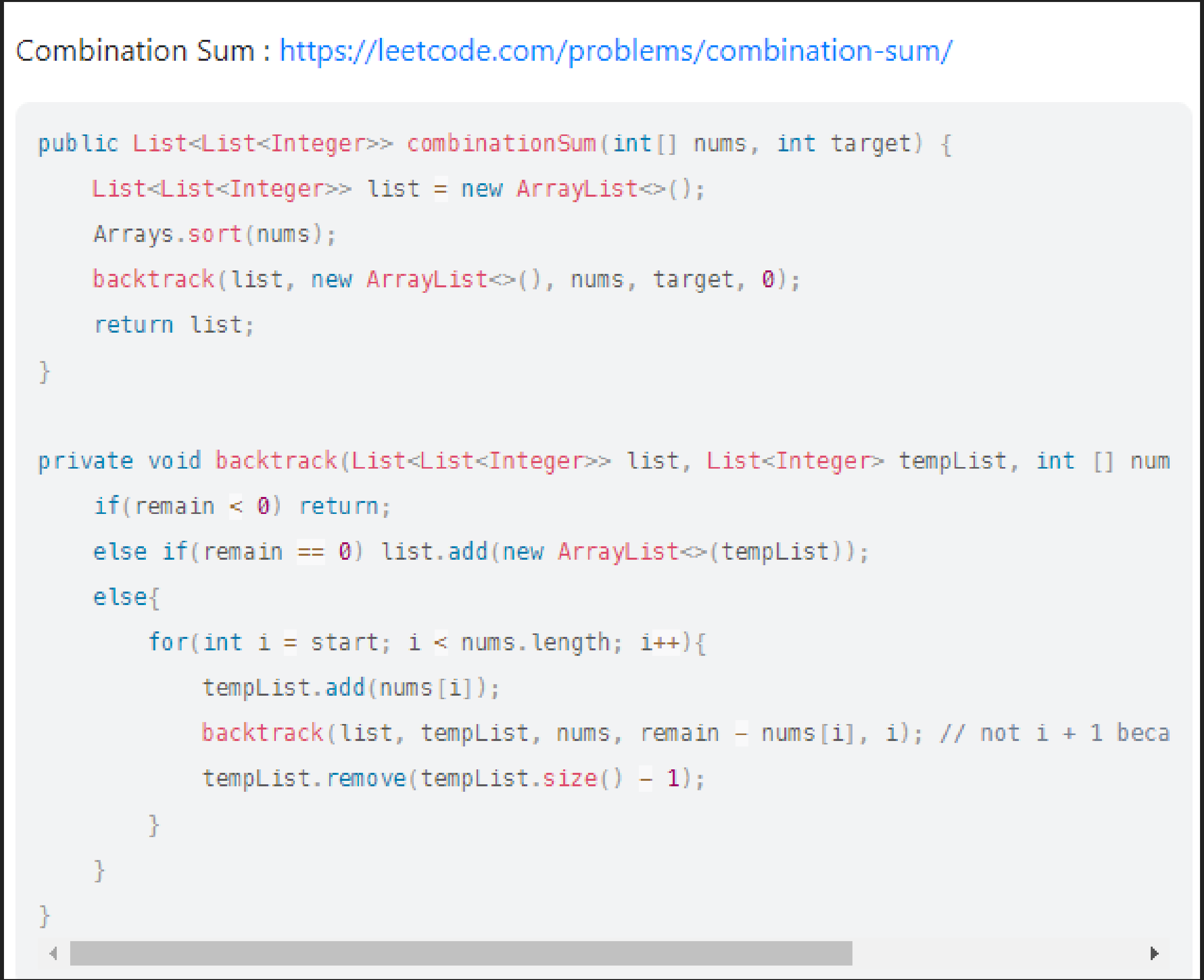
Permutations II (contains duplicates) : https://leetcode.com/problems/permutations-ii/

```
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 [] num
    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/



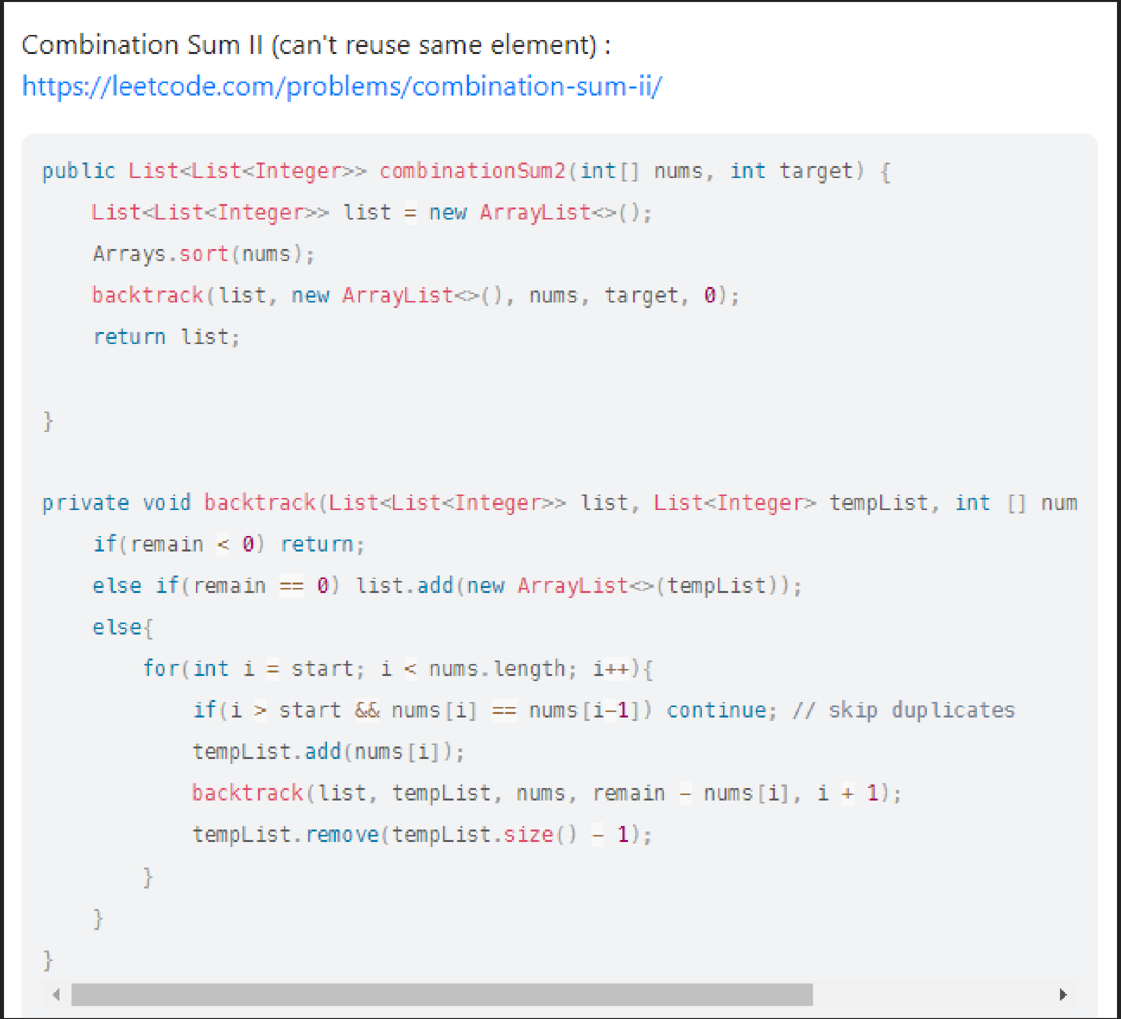
Combination Sum : https://leetcode.com/problems/combination-sum/

```
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 [] num
    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 beca
            tempList.remove(tempList.size() - 1);
        }
    }
}
```

Combination sum 2

https://leetcode.com/problems/combination-sum-ii/



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

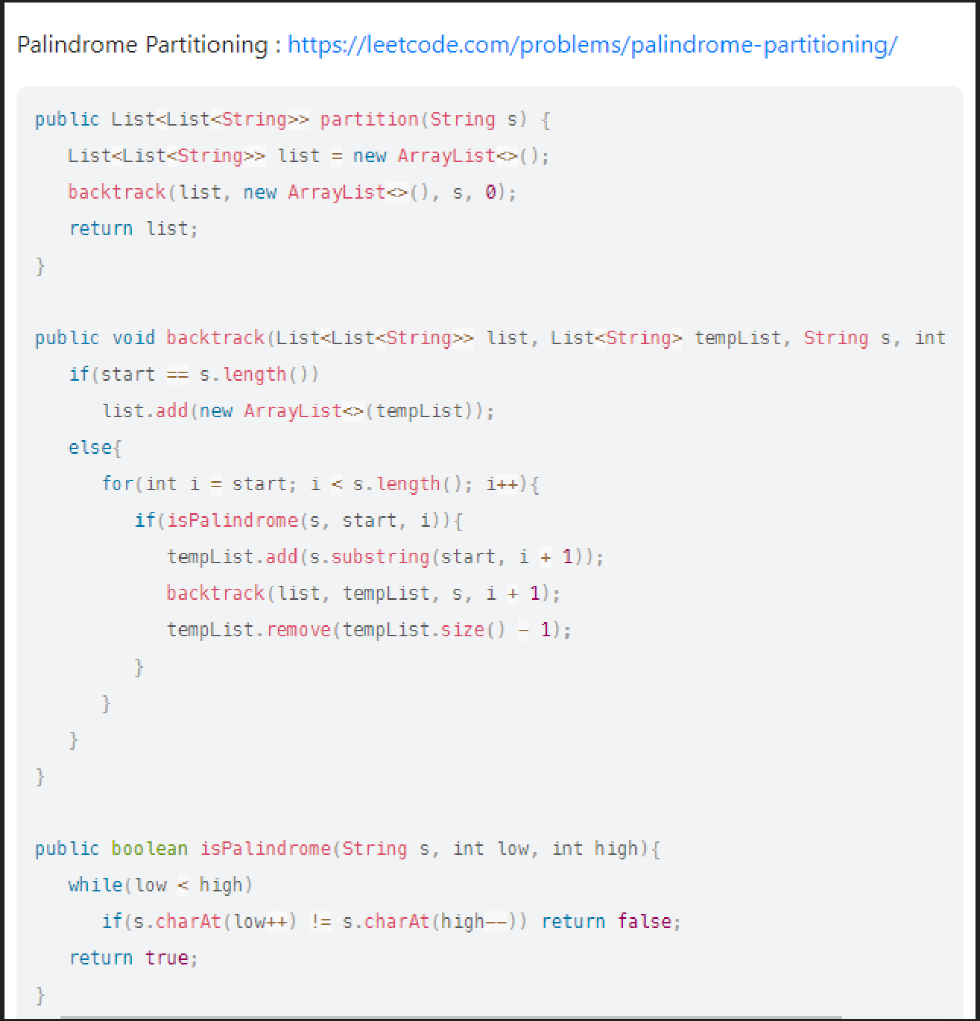
```java
public List<List<Integer>> combinationSum2(int[] nums, int target) {
    List<List<Integer>> list = new ArrayList<>();
    Arrays.sort(nums);
    backtrack(list, new ArrayList<>(), nums, target, 0);
    return list;

}

private void backtrack(List<List<Integer>> list, List<Integer> tempList, int [] num
    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/

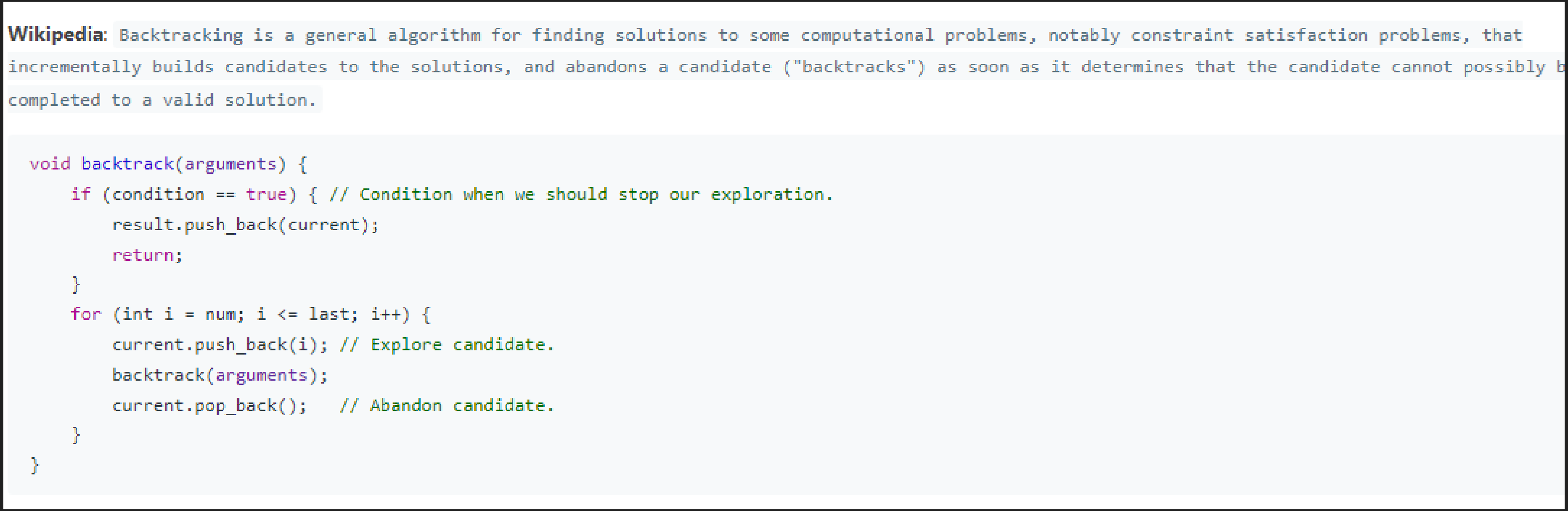


Palindrome Partitioning : https://leetcode.com/problems/palindrome-partitioning/

```
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
   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;
}
```



Wikipedia: `Backtracking is a general algorithm for finding solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly b completed to a valid solution.`

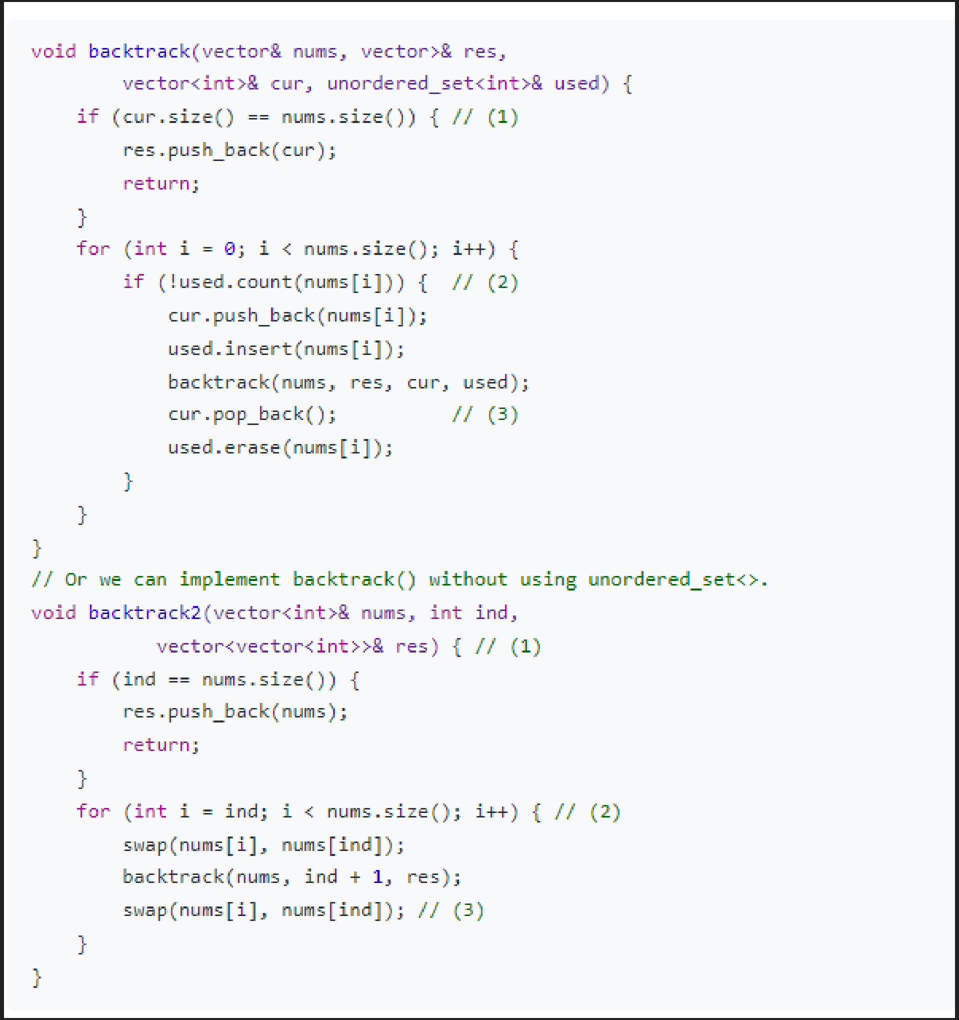
```
void backtrack(arguments) {
    if (condition == true) { // Condition when we should stop our exploration.
        result.push_back(current);
        return;
    }
    for (int i = num; i <= last; i++) {
        current.push_back(i); // Explore candidate.
        backtrack(arguments);
        current.pop_back();   // Abandon candidate.
    }
}
```

One thing to remember before we can jump to some backtracking problems:

1. **Permutation**: can be thought of number of ways to order some input.
    - Example: permutations of ABCD, taken 3 at a time (24 variants): ABC, ACB, BAC, BCA, ...
2. **Combnation**: can be thought as the number of ways of selecting from some input.
    - Example: combination of ABCD, taken 3 at a time (4 variants): ABC, ABD, ACD, and BCD.
3. **Subset**: can be thought as a selection of objects form the original set.
    - Example: subset of ABCD: 'A', 'B', 'C', 'D,' 'A,B' , 'A,C', 'A,D', 'B,C', 'B,D', 'C,D', 'A,B,C', ...

From now let's start to apply this algorithm to solve some backtracking problems.

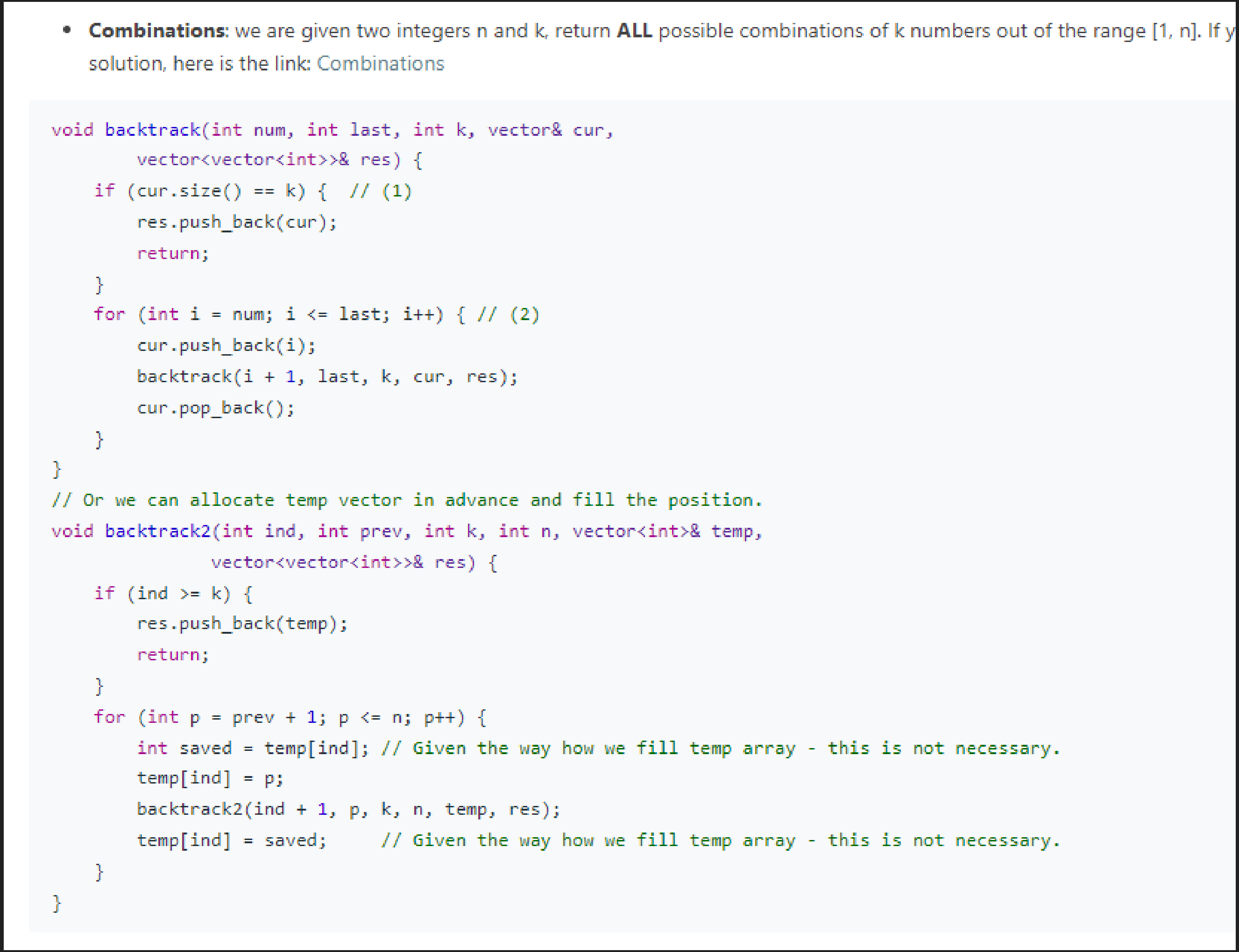
- **Permutations**:
  this set of problems related to generating (subset of) all possible permutations. Let's have a look at fist problem: Permutations
  In this problem we should return **ALL** the possible permutations from **DISTINCT** integer array.



```cpp
void backtrack(vector& nums, vector>& res,
        vector<int>& cur, unordered_set<int>& used) {
    if (cur.size() == nums.size()) { // (1)
        res.push_back(cur);
        return;
    }
    for (int i = 0; i < nums.size(); i++) {
        if (!used.count(nums[i])) {  // (2)
            cur.push_back(nums[i]);
            used.insert(nums[i]);
            backtrack(nums, res, cur, used);
            cur.pop_back();          // (3)
            used.erase(nums[i]);
        }
    }
}
// Or we can implement backtrack() without using unordered_set<>.
void backtrack2(vector<int>& nums, int ind,
           vector<vector<int>>& res) { // (1)
    if (ind == nums.size()) {
        res.push_back(nums);
        return;
    }
    for (int i = ind; i < nums.size(); i++) { // (2)
        swap(nums[i], nums[ind]);
        backtrack(nums, ind + 1, res);
        swap(nums[i], nums[ind]); // (3)
    }
}
```

Another variation of the problem is Permutations II.
The only difference between first problem is that we **MAY** have **DUPLICATES** in the input array.

```
void backtrack(vector& nums, vector& cur, vector>& res,
         unordered_map& hmap) {
    if (cur.size() == nums.size()) { // (1)
        res.push_back(cur);
        return;
    }
    for (auto& [num, freq] : hmap) { // (2)
        if (freq == 0) continue;     // (3)
        freq--;
        cur.push_back(num);
        dfs(nums, cur, res, hmap);
        cur.pop_back();              // (4)
        freq++;
    }
}
// Iterate over the original list, but check if the previous element is the same as current.
// We need to make this check because using the same element will give us the same result as last iteration.
void backtrack2(vector<int>& nums, vector<int>& temp,
                  vector<vector<int>>& res, unordered_map<int, int>& freq) {
        if (temp.size() == nums.size()) {
            res.push_back(temp);
            return;
        }
        for (int i = 0; i < nums.size(); i++) {
            if (freq[nums[i]] == 0 || (i != 0 && nums[i] == nums[i - 1])) continue;
            temp.push_back(nums[i]);
            freq[nums[i]]--;
            backtrack(nums, temp, res, freq);
            freq[nums[i]]++;
            temp.pop_back();
        }
    }
```



- **Combinations:** we are given two integers n and k, return **ALL** possible combinations of k numbers out of the range [1, n]. If y
  solution, here is the link: Combinations

```
void backtrack(int num, int last, int k, vector& cur,
        vector<vector<int>>& res) {
    if (cur.size() == k) {  // (1)
        res.push_back(cur);
        return;
    }
    for (int i = num; i <= last; i++) { // (2)
        cur.push_back(i);
        backtrack(i + 1, last, k, cur, res);
        cur.pop_back();
    }
}
// Or we can allocate temp vector in advance and fill the position.
void backtrack2(int ind, int prev, int k, int n, vector<int>& temp,
               vector<vector<int>>& res) {
    if (ind >= k) {
        res.push_back(temp);
        return;
    }
    for (int p = prev + 1; p <= n; p++) {
        int saved = temp[ind]; // Given the way how we fill temp array - this is not necessary.
        temp[ind] = p;
        backtrack2(ind + 1, p, k, n, temp, res);
        temp[ind] = saved;     // Given the way how we fill temp array - this is not necessary.
    }
}
```

4. Backtrack: the same as in earlier examples.

- **Subsets**: we are given an integer array of unique elements, return **all** possible subsets (the power set)
  Subsets

```
void dfs(int ind, vector& nums, vector& cur, vector>& res) {
    res.push_back(cur); // (1)
    for (int i = ind; i < nums.size(); i++) { // (2)
        cur.push_back(nums[i]);
        dfs(i + 1, nums, cur, res);
        cur.pop_back(); // (3)
    }
}
```

Let's check the steps again:

1. Now we are adding element to the result unconditionally. This is because we need to generate the su
2. The same as in previous examples: we are using new element on each dfs() call.
3. Backtrack: the same as in previous example.
   The implementation will be a bit different if the input array has duplicates Subsets II , but we already

```
void dfs(int ind, vector& cur, vector>& res,
        vector& nums) {
    res.push_back(cur);
    for (int i = ind; i < nums.size(); i++) {
        if (i > ind && nums[i] == nums[i - 1]) continue;
        cur.push_back(nums[i]);
        dfs(i + 1, cur, res, nums);
        cur.pop_back();
    }
}
```