Recursion_2

@kbbhatt04

@August 15, 2023

Combination Sum - 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.

- Approach
    - Optimal
        - Initially, the index will be 0, target as given and the data structure(vector or list) will be empty
        - Now there are 2 options viz to pick or not pick the current index element
        - If you pick the element, again come back at the same index as multiple occurrences of the same element is possible so the target reduces to `target – arr[index]` (where `target - arr[index] >= 0`)and also insert the current element into the data structure
        - If you decide not to pick the current element, move on to the next index and the target value stays as it is
        - Time Complexity: $O(2^{target} * k)$ where k is average length
        - Space Complexity: $O(k * x)$ where k is the average length and x is the number of combinations

```
# Python3
# Optimal Solution
class Solution:
    def combinationSum(self, candidates, target):
        ans = []
        temp = []
        def fun(index, n, target):
            if(n == index):
                if(target == 0):
                    ans.append(temp[:])
                return

            if(candidates[index] <= target):
                temp.append(candidates[index])
                fun(index, n, target - candidates[index])
                temp.pop()

            fun(index+1, n, target)

        fun(0, len(candidates), target)
        return ans
```

```
// C++
// Optimal Solution
void findCombination(int ind, int target, vector < int > & arr,
      if (ind == arr.size()) {
        if (target == 0) {
          ans.push_back(ds);
        }
        return;
      }
      // pick up the element
      if (arr[ind] <= target) {
        ds.push_back(arr[ind]);
        findCombination(ind, target - arr[ind], arr, ans, ds);
        ds.pop_back();
      }
```

```
        findCombination(ind + 1, target, arr, ans, ds);
    }
```

Combination Sum - 2

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.

- Approach
    - Brute-force
        - Sort the array because the ans should contain the combinations in sorted order
        - Generate all combinations that add up to target
        - Add all these combinations to a Set
        - Time Complexity: $O(2^n * k)$ + Time to convert set to list, where k is average length of ds
        - Space Complexity: $O(k * x)$ where k is average length of ds and x is the number of combinations
    - Optimal
        - Before starting the recursive call make sure to sort the elements because the ans should contain the combinations in sorted order and should not be repeated
        - Initially, we start with the index 0, at index 0 we have n – 1 way to pick the first element of our sub-sequence
        - Check if the current index value can be added to our ds. If yes add it to the ds and move the index by 1. While moving the index skip the consecutive repeated elements because they will form duplicate sequences

- Reduce the target by arr[i], and call the recursive call for fun(nums, n, idx + 1, target – nums[i], ds, ans) after the call make sure to pop the element from the ds
- if(arr[i] > target) then terminate the recursive call because there is no use to check as the array is sorted in the next recursive call the index will be moving by 1 all the elements to its right will be in increasing order
- Time Complexity: $O(2^n * k)$ where k is average length of ds
- Space Complexity: $O(k * x)$ where k is average length of ds and x is the number of combinations

```
# Python3
# Brute-force Solution
class Solution:
    def fun(self, nums, n, ind, target, ds, ans):
        if ind == n:
            if target == 0:
                ans.add(tuple(ds[:]))
            return

        ds += nums[ind],
        self.fun(nums, n, ind+1, target-nums[ind], ds, ans)
        ds.pop()

        self.fun(nums, n, ind+1, target, ds, ans)

    def combinationSum2(self, nums, target):
        nums.sort()
        ans = set()
        self.fun(nums, len(nums), 0, target, [], ans)
        ans = [list(i) for i in ans]
        return ans
```

```
# Python3
# Optimal Solution
class Solution:
    def fun(self, nums, n, ind, target, ds, ans):
        if target == 0:
            ans += ds[:],
            return

        for i in range(ind, n):
            if i > ind and nums[i] == nums[i - 1]:  continue
            if nums[i] > target:    break
            ds += nums[i],
            self.fun(nums, n, i+1, target-nums[i], ds, ans)
            ds.pop()

    def combinationSum2(self, nums, target):
        nums.sort()
        ans = []
        self.fun(nums, len(nums), 0, target, [], ans)
        return ans
```

```
// C++
// Optimal Solution
#define v_int vector<int>
#define v_v_int vector<vector<int>>

class Solution {
public:
    void fun(int ind, v_int nums, v_int& temp, v_v_int& ans, in
    {
        if(target == 0)
        {
            ans.push_back(temp);
            return;
        }

        for(int i = ind; i < n; i++)
        {
            if(i > ind && nums[i] == nums[i-1])   continue;
            if(nums[i] > target)  break;
            temp.push_back(nums[i]);
            fun(i + 1, nums, temp, ans, n, target-nums[i]);
            temp.pop_back();
        }
    }

    v_v_int combinationSum2(v_int& nums, int target) {
        vector<vector<int>> ans;
        vector<int> temp;
        sort(nums.begin(), nums.end());
        fun(0, nums, temp, ans, nums.size(), target);
        return ans;
    }
};
```

Subset Sum - 1

Return all subset sums of the given array in non-decreasing order

- Approach
  - Brute-force
    - Generate all subsets
    - Calculate sum of all subsets and insert it into a list
    - Sort and return the list
    - Time Complexity: $O(2^n + 2^n)$
    - Space Complexity: $O(2^n)$
  - Optimal

- Incrementally add and subtract element (using Take/Not Take logic)
- Time Complexity: $O(2^n)$
- Space Complexity: $O(2^n)$

```python
# Python3
# Brute-force Solution
from typing import List

def subsetSum(num: List[int]) -> List[int]:
    subs = []
    temp = []

    def fun(index):
        if index == len(num):
            subs.append(temp[:])
            return

        temp.append(num[index])
        fun(index + 1)
        temp.pop()

        fun(index + 1)

    fun(0)
    ans = [sum(i) for i in subs]
    ans.sort()
    return ans
```

```python
# Python3
# Optimal Solution
def func(num, n, ind, temp_sum, ans):
    if ind == n:
        ans += temp_sum,
        return
    func(num, n, ind+1, temp_sum+num[ind], ans)
    func(num, n, ind+1, temp_sum, ans)

def subsetSum(num):
    ans = []
    func(num, len(num), 0, 0, ans)
    ans.sort()
    return ans
```

```
// C++
// Optimal Solution
void func(vector<int> num, int n, int ind, int temp_sum, vector<
    if (ind == n) {
        ans.push_back(temp_sum);
        return;
    }
    func(num, n, ind+1, temp_sum+num[ind], ans);
    func(num, n, ind+1, temp_sum, ans);
}

vector<int> subsetSum(vector<int> &num){
    vector<int> ans;
    func(num, num.size(), 0, 0, ans);
    sort(ans.begin(), ans.end());
    return ans;
}
```

Subsets - 2

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.

- Approach

- Optimal
    - Sort the input array
    - Make a recursive function that takes the input array, the current subset, the current index and a list of list/ vector of vectors to contain the answer
    - Try to make a subset of size n during the nth recursion call and consider elements from every index while generating the combinations
    - Only pick up elements that are appearing for the first time during a recursion call to avoid duplicates
    - Once an element is picked up, move to the next index
    - The recursion will terminate when the end of the array is reached
    - While returning backtrack by removing the last element that was inserted
    - Time Complexity: $O(2^n * k)$ where k is the time taken to insert every subset in answer list
    - Space Complexity: $O(2^n * k)$ to store every subset of average length k. Auxiliary space is $O(n)$ if n is the depth of the recursion tree.

```
# Python3
# Optimal Solution
class Solution:
    def subsetsWithDup(self, nums):
        ans = []
        ds = []
        def findSubsets(ind):
            ans.append(ds[:])
            for i in range(ind, len(nums)):
                if i > ind and nums[i] == nums[i - 1]:
                    continue
                ds.append(nums[i])
                findSubsets(i + 1)
                ds.pop()
        nums.sort()
        findSubsets(0)
        return ans
```

```
// C++
// Optimal Solution
#define v_int vector<int>
#define v_v_int vector<vector<int>>

class Solution {
   private:
      void findSubs(int ind, v_int& nums, v_int& ds, v_v_int& ar
         ans.push_back(ds);
         for (int i = ind; i < nums.size(); i++) {
            if (i != ind && nums[i] == nums[i - 1]) continue;
            ds.push_back(nums[i]);
            findSubs(i + 1, nums, ds, ans);
            ds.pop_back();
         }
      }
   public:
      v_v_int subsetsWithDup(v_int & nums) {
         v_v_int ans;
         v_int ds;
         sort(nums.begin(), nums.end());
         findSubs(0, nums, ds, ans);
         return ans;
      }
};
```

Combination Sum - 3

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.

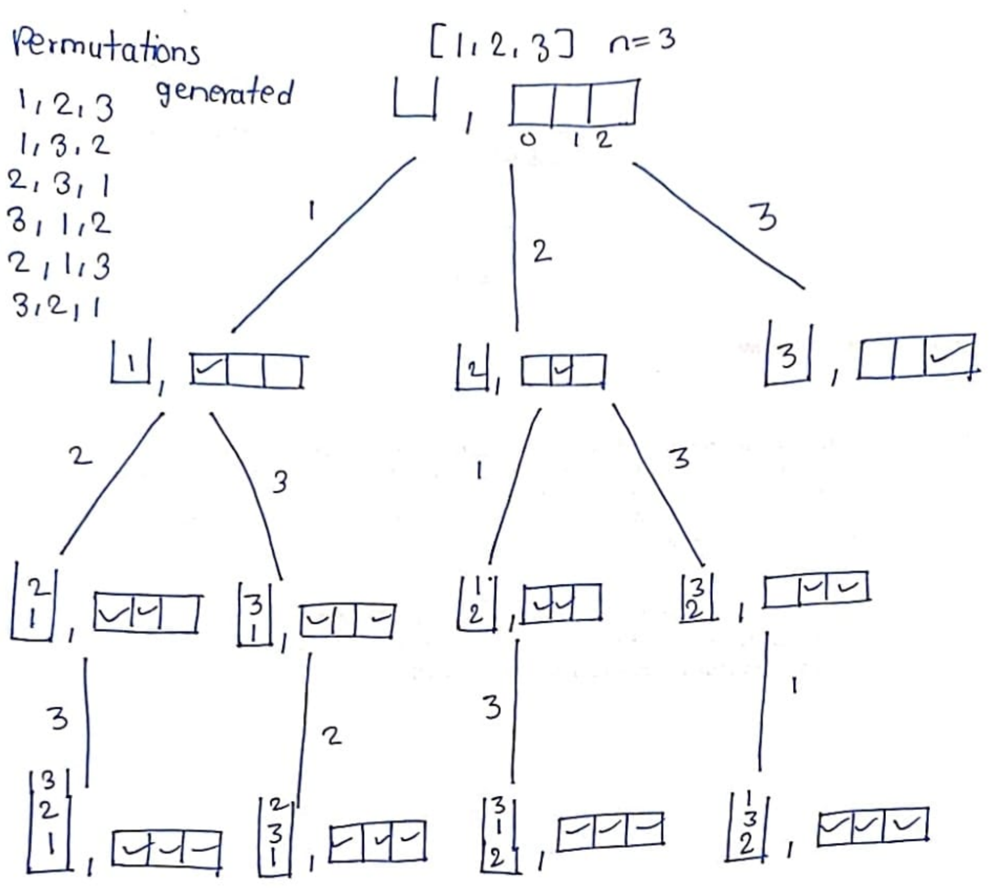
- Approach
    - Optimal
        - Same logic as Combination Sum - 2
        - Time Complexity: $O(2^n * k)$ where k is average length of ds
        - Space Complexity: $O(k * x)$ where k is average length of ds and x is the number of combinations

```
# Python3
# Optimal Solution
class Solution:
    def combinationSum3(self, k: int, n: int) -> List[List[int]
        nums = [1,2,3,4,5,6,7,8,9]
        ans = []

        def fun(index, k, target, temp_ds, temp_sum):
            if index == 9:
                if len(temp_ds) == k:
                    if temp_sum == target:
                        ans.append(temp_ds[:])
                return

            temp_ds += nums[index],
            temp_sum += nums[index]
            fun(index + 1, k, target, temp_ds, temp_sum)
            temp_ds.pop()
            temp_sum -= nums[index]

            fun(index + 1, k, target, temp_ds, temp_sum)
        fun(0, k, n, [], 0)
        return ans
```

```
// C++
// Optimal Solution
#define v_int vector<int>
#define v_v_int vector<vector<int>>
class Solution {
public:
    void fun(v_int nums, int ind, int k, int n, v_int& temp,
                        int temp_sum, v_v_int& ans) {
        if (ind == 9) {
            if (temp.size() == k) {
                if (temp_sum == n) {
                    ans.push_back(temp);
                }
            }
            return;
        }

        temp.push_back(nums[ind]);
        temp_sum += nums[ind];
        fun(nums, ind+1, k, n, temp, temp_sum, ans);
        temp_sum -= nums[ind];
        temp.pop_back();

        fun(nums, ind+1, k, n, temp, temp_sum, ans);
    }

    vector<vector<int>> combinationSum3(int k, int n) {
        v_int nums = {1,2,3,4,5,6,7,8,9};
        v_v_int ans;
        v_int temp;
        fun(nums, 0, k, n, temp, 0, ans);
        return ans;
    }
};
```

Print all permutations of an array

- Approach
  - Brute-force
    - Run a for loop starting from 0 to nums.size() – 1
    - Check if the frequency of i is unmarked, if it is unmarked then it means it has not been picked and then we pick and make sure it is marked as picked afterwards
    - Call the recursion with the parameters to pick the other elements when we come back from the recursion make sure you throw that element out and unmark that element in the map



- Time Complexity: $O(n! * n)$
- Space Complexity: $O(n)$

- Optimal
  -
  - Time Complexity: $O(n! * n)$
  - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    ans = []
    ds = []
    def recurPermute(self, nums: List[int], freq: List[int]):
        if len(self.ds) == len(nums):
            self.ans.append(self.ds.copy())
            return
        for i in range(len(nums)):
            if not freq[i]:
                self.ds.append(nums[i])
                freq[i] = 1
                self.recurPermute(nums, freq)
                freq[i] = 0
                self.ds.pop()

    def permute(self, nums: List[int]) -> List[List[int]]:
        self.ans = []
        self.ds = []
        freq = [0] * len(nums)
        self.recurPermute(nums, freq)
        return self.ans
```

```
// C++
// Brute-force Solution
#define v_int vector<int>
#define v_v_int vector<vector<int>>
class Solution {
  private:
    void recurPermute(v_int& ds, v_int& nums, v_v_int& ans, int
      if (ds.size() == nums.size()) {
        ans.push_back(ds);
        return;
      }
      for (int i = 0; i < nums.size(); i++) {
        if (!freq[i]) {
          ds.push_back(nums[i]);
          freq[i] = 1;
          recurPermute(ds, nums, ans, freq);
          freq[i] = 0;
          ds.pop_back();
        }
      }
    }
  public:
    v_v_int permute(vector < int > & nums) {
      v_v_int ans;
      v_int ds;
      int freq[nums.size()];
      for (int i = 0; i < nums.size(); i++) freq[i] = 0;
      recurPermute(ds, nums, ans, freq);
      return ans;
    }
};
```

```
# Python3
# Optimal Solution
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        def fun(index):
            if(index == n):
                ans.append(nums[:])
                return
            for i in range(index, n):
                nums[i], nums[index] = nums[index], nums[i]
                fun(index+1)
                nums[i], nums[index] = nums[index], nums[i]

        n = len(nums)
        ans = []
        fun(0)
        return ans
```

```
// C++
// Optimal Solution
#define v_int vector<int>
#define v_v_int vector<vector<int>>

class Solution {
public:
    void fun(v_int& nums, int index, v_v_int& ans) {
        if (index == nums.size()) {
            ans.push_back(nums);
            return;
        }
        for (int i = index; i < nums.size(); i++) {
            swap(nums[index], nums[i]);
            fun(nums, index+1, ans);
            swap(nums[index], nums[i]);
        }
    }
    v_v_int permute(v_int& nums) {
        v_v_int ans;
        fun(nums, 0, ans);
        return ans;
    }
};
```

Template

- Approach
    - Brute-force
        -
        - Time Complexity: $O(n^3)$
        - Space Complexity: $O(1)$

- Better
  -
  - Time Complexity: $O(n^3)$
  - Space Complexity: $O(1)$
- Optimal
  -
  - Time Complexity: $O(n^3)$
  - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
```

```
# Python3
# Better Solution
```

```
# Python3
# Optimal Solution
```

```
// C++
// Optimal Solution
```