40. Combination Sum II

40. Combination Sum II

Medium

344698Add to ListShare

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.

Example 1:

```
Input: candidates = [10,1,2,7,6,1,5], target = 8
Output:
[
[1,1,6],
[1,2,5],
[1,7],
[2,6]
]
```

Example 2:

```
Input: candidates = [2,5,2,1,2], target = 5
Output:
[
[1,2,2],
[5]
]
```

Constraints:

- `1 <= candidates.length <= 100`
- `1 <= candidates[i] <= 50`
- `1 <= target <= 30`

-
  ```
      class Solution:
          def combinationSum2(self, candidates: List[int], target: int) ->
  ```

```
List[List[int]]:
            if sum(candidates)<target:
                return []
            res = []
            visited = [False]*len(candidates)
            self.combinationalSum(sorted(candidates),target,res,0,
[],visited)
            return res


        def combinationalSum(self,candidates,target,res,idx,asf,visited):
            if target==0:
                if asf not  in res:
                    res.append(asf)
                return
            if target<0 or idx==len(candidates):
                return
            for i in range(idx,len(candidates)):
                if i>idx and candidates[i]==candidates[i-1]:
                    continue
                if visited[i]==False and candidates[i]<=target:
                    visited[i]=True
                    self.combinationalSum(candidates, target-
candidates[i], res, i+1, asf+[candidates[i]],visited)
                    visited[i]=False
```

Approach 1: Backtracking with Counters

Intuition

> As a reminder, backtracking is a general algorithm for finding all (or some) solutions to some computational problems. The idea is that it ***incrementally*** builds candidates to the solutions, and abandons a candidate ("backtrack") as soon as it determines that the candidate cannot lead to a final solution.

In our problem, we could *incrementally* build the combination by adding numbers one at a time, and once we find the current combination is not valid, we *backtrack* (by abondoning the last number we added to the combination) and try another candidate.

As we mentioned before, this problem is an extention of an earlier problem called 39. Combination Sum. As it turns out, we could build the solutions upon the solutions to the problem of 39. Combination Sum, by incorporating the differences between the problems.

There are two differences between this problem and the earlier problem:

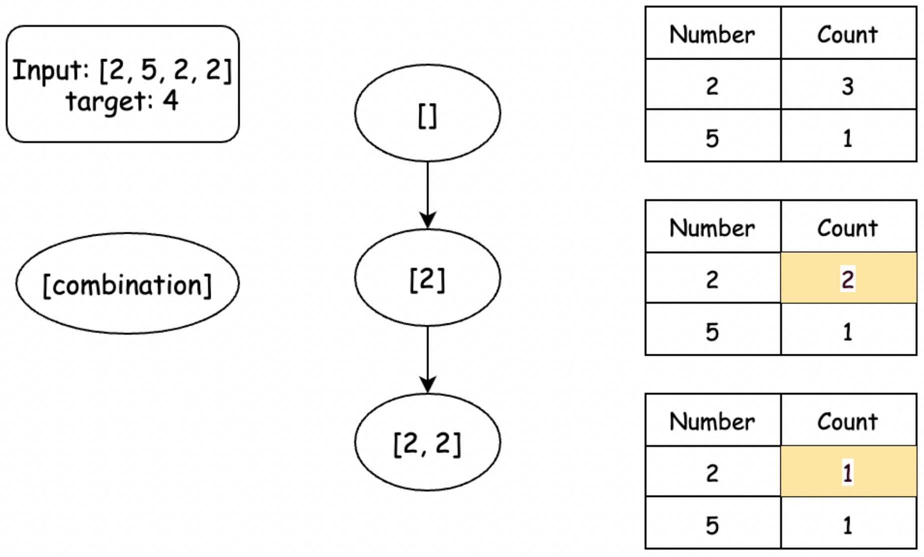
- In this problem, each number in the input is **not** unique. The implication of this difference is that we need some mechanism to avoid generating **duplicate** combinations.

- In this problem, each number can be used only **once**. The implication of this difference is that once a number is chosen as a candidate in the combination, it will not appear again as a candidate later.

There are several ways to adapt the solutions of [39. Combination Sum] to solve this problem.

> In this approach, we will present a solution with the concept of **counter**. Rather than treating each number as a candidate, we treat groups of unique numbers as candidates.

To demonstrate the idea, we showcase how it works with a concrete example in the following graph:



As one can see from the above graph, if we treat each appearance of the number `2` as a candidate, then we would generate multiple instances of the same combination of `[2, 2]`. For instance, the first and second appearances of the number `2` will lead to the same combination as the second and the third appearances of the number `2`.

> However, we could count the appearance of each unique number. And then we can use the generated *counter* table during the construction of the combination.

For instance, starting from the empty combination, we first pick the number `2` as the first candidate into the combination. In the counter table, we then update the count of the number `2`, which remains 2 instances rather than 3. In the next step, again we pick another instance of the number `2` into the combination. With this pick, we reach the desired target number which is `4`.

> As one can see, with the counter table, at each step, we could ensure that each combination we generate would be **unique** at the end.

Algorithm

Here are a few steps on how we can implement the above intuition:

- First of all, we build a counter table out of the given list of numbers.

- We would then use this counter table during our *backtracking* process, which we define as the function `backtrack(comb, remain, curr, candidate_groups, results)`. In order to keep the **state** of each backtracking step, we use quite a few parameters in the function, which we elaborate as follows:
    - `comb`: the combination we built so far at each step.
    - `remain`: the remaining sum that we need to fill, in order to reach the target sum.
    - `curr`: the cursor that points to the current group of number that we are using from the counter table.
    - `counter`: the current counter table.
    - `results`: the final combinations that have the target sum.
- At each invocation of the backtracking function, we first check if we reach the target sum (*i.e.* `sum(comb) = target`), and if we should stop the exploration simply because the sum of current combination goes **beyond** the desired target.
- If there is still some remaining sum to fill, we will then iterate through the current counter table to pick the next candidate.
    - Once we pick a candidate, we then continue the exploration by invocating the `backtrack()` function with the **updated** states.
    - **More importantly**, at the end of each exploration, we need to **revert** the state we updated before, in order to start off a clean slate for the next exploration. It is due to this *backtracking* operation, the algorithm got its name.

```python
class Solution:
    def combinationSum2(self, candidates: List[int], target: int) -> List[List[int]]:

        def backtrack(comb, remain, curr, counter, results):
            if remain == 0:
                # make a deep copy of the current combination
                #   rather than keeping the reference.
                results.append(list(comb))
                return
            elif remain < 0:
                return

            for next_curr in range(curr, len(counter)):
                candidate, freq = counter[next_curr]

                if freq <= 0:
                    continue
                # add a new element to the current combination
                comb.append(candidate)
                counter[next_curr] = (candidate, freq-1)

                # continue the exploration with the updated combination
                backtrack(comb, remain - candidate, next_curr, counter,
results)

                # backtrack the changes, so that we can try another
candidate
                counter[next_curr] = (candidate, freq)
                comb.pop()

        results = []  # container to hold the final combinations
        counter = Counter(candidates)
        # convert the counter table to a list of (num, count) tuples
        counter = [(c, counter[c]) for c in counter]

        backtrack(comb = [], remain = target, curr = 0,
                  counter = counter, results = results)

        return results
```