Combination Sum

Difficulty	Medium
Category	BackTracking
Question	https://leetcode.com/problems/combination-sum/
Solution	https://youtu.be/GBKI9VSKdGg
Status	Done

Question

> 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.
>
> The test cases are generated such that the number of unique combinations that sum up to `target` is less than `150` combinations for the given input.

Example

Example 1:

```
Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]
Explanation:
2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times.
7 is a candidate, and 7 = 7.
These are the only two combinations.
```

Example 2:

```
Input: candidates = [2,3,5], target = 8
Output: [[2,2,2,2],[2,3,3],[3,5]]
```

Example 3:

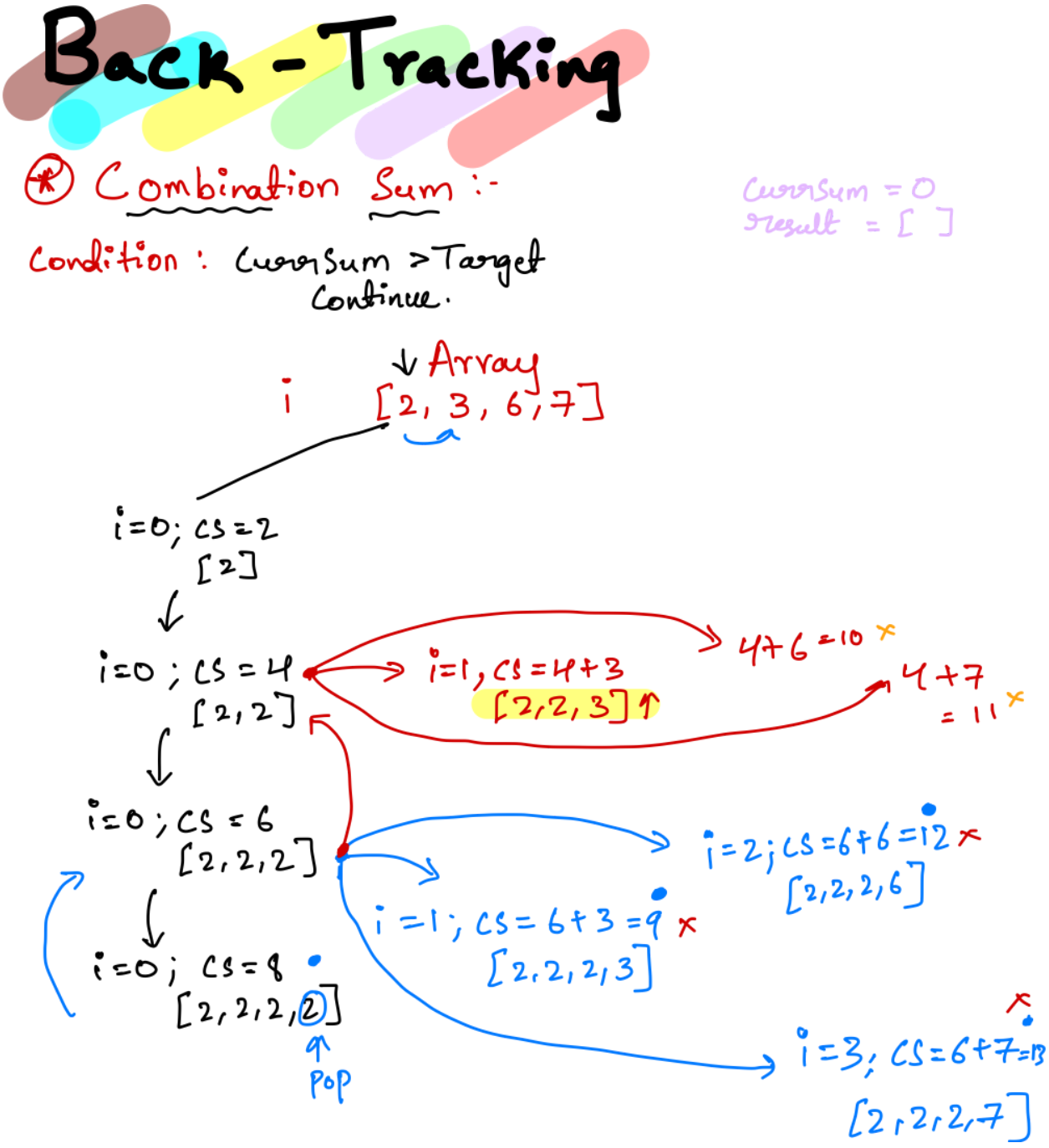
```
Input: candidates = [2], target = 1
Output: []
```

Idea

Back-tracing, starting from the target value, check if subtract from each one of the candidate result in ≥ 0. If so, it means that there's a possible path. Continue to subtract from the current candidate and so on. If the target turns into 0, it means that there's a combination sum achieved.

▼ Special Note

1. Remove the last element entered to reverse the state of the current possible combination sum so that it can experiment other candidates from the same state
2. Keep track of a index value, so that the child candidate iteration doesn't start from zero. Instead, it start from the current index and so on. Therefore, eliminate the duplicates like [2,2,3], [3,2,2], and [2,3,2]



Solution

```
def combinationSum(candidates, target):
    # Initialize a list to store the valid combinations
    result = []
```

```
# Define a backtracking function
def backtrack(start, target, current_combination):
    # Base case: If the target becomes zero, add the current combination to the result
    if target == 0:
        result.append(list(current_combination))
        return
    # Explore all possible candidates
    for i in range(start, len(candidates)):
        # If the candidate is greater than the remaining target, skip it
        if candidates[i] > target:
            continue
        # Add the candidate to the current combination
        current_combination.append(candidates[i])
        # Recursively call the function with the updated target and combination
        backtrack(i, target - candidates[i], current_combination)
        # Remove the last element (backtrack) to explore other possibilities
        current_combination.pop()

# Start the backtracking from the beginning of the candidates list
backtrack(0, target, [])

return result
```

Explanation

> Certainly! Let's break down the code for finding unique combinations that sum to the target:
>
> ```
> def combinationSum(candidates, target):
>     # Initialize a list to store the valid combinations
>     result = []
> ```
>
> - We start by defining the `combinationSum` function, which takes two arguments: `candidates`, a list of distinct integers, and

`target`, the target integer we want to reach with combinations. `result` is an empty list where we'll store the valid combinations.

```
    # Define a backtracking function
    def backtrack(start, target, current_combination):
```

- Next, we define a recursive `backtrack` function. It has three parameters:
  - `start`: This parameter keeps track of the index from which we should start considering candidates. It helps avoid duplicate combinations.
  - `target`: This parameter represents the remaining target we need to reach with the current combination.
  - `current_combination`: This is a list that holds the current combination of candidates we are considering.

```
        # Base case: If the target becomes zero, add the current com
bination to the result
        if target == 0:
            result.append(list(current_combination))
            return
```

- The base case of the recursion is when `target` becomes zero. This means we have found a valid combination, and we add a copy of the `current_combination` to the `result` list.

```
        # Explore all possible candidates
```

```
        for i in range(start, len(candidates)):
```

- We then loop through the candidates, starting from the index `start`. This loop helps us consider all possible candidates for the current position in the combination.

```
            # If the candidate is greater than the remaining target,
skip it
            if candidates[i] > target:
                continue
```

- We check if the current candidate is greater than the remaining `target`. If it is, we skip this candidate because it cannot be part of a valid combination.

```
            # Add the candidate to the current combination
            current_combination.append(candidates[i])
```

- If the candidate is valid, we add it to the `current_combination`.

```
            # Recursively call the function with the updated target
and combination
            backtrack(i, target - candidates[i], current_combinatio
n)
```

- We make a recursive call to the `backtrack` function with the updated values:
    - `i`: The index at which we're considering the candidate.

    - `target - candidates[i]`: The updated remaining target after adding the candidate.
    - `current_combination`: The updated combination with the candidate.

```
            # Remove the last element (backtrack) to explore other p
ossibilities
            current_combination.pop()
```

- After returning from the recursive call, we remove the last element from the `current_combination`. This is the "backtracking" step, which allows us to explore other possibilities and not include the same candidate multiple times in a single combination.

```
    # Start the backtracking from the beginning of the candidates li
st
    backtrack(0, target, [])

    return result
```

- Finally, we start the backtracking process by calling the `backtrack` function with an initial `start` index of 0, the original `target`, and an empty `current_combination`. After the backtracking process is complete, we return the `result` list containing all the unique combinations that sum to the target.

This code effectively explores all possible combinations of candidates, using backtracking to find valid combinations and

> backtracking when necessary to explore other possibilities. It ensures that each combination is unique, even if the same candidate is used multiple times.