Subsets With Duplicates (easy)

Problem Statement

Given a set of numbers that might contain duplicates, find all of its distinct subsets.

Example 1:

```
Input: [1, 3, 3]
Output: [], [1], [3], [1,3], [3,3], [1,3,3]
```

Example 2:

```
Input: [1, 5, 3, 3]
Output: [], [1], [5], [3], [1,5], [1,3], [5,3], [1,5,3], [3,3], [1,3,3], [3,3,5], [1,5,3,3]
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
const find_subsets = function(nums) {
  subsets = [];
  // TODO: Write your code here
  return subsets;
};


console.log(`Here is the list of subsets: ${find_subsets([1, 3, 3])}`)
console.log(`Here is the list of subsets: ${find_subsets([1, 5, 3, 3])}`)
```

RUN SAVE RESET

Solution

This problem follows the Subsets pattern and we can follow a similar **Breadth First Search (BFS)** approach. The only additional thing we need to do is handle duplicates. Since the given set can have duplicate numbers, if we follow the same approach discussed in Subsets, we will end up with duplicate subsets, which is not acceptable. To handle this, we will do two extra things:

1. Sort all numbers of the given set. This will ensure that all duplicate numbers are next to each other.
2. Follow the same BFS approach but whenever we are about to process a duplicate (i.e., when the current and the previous numbers are same), instead of adding the current number (which is a duplicate) to all the existing subsets, only add it to the subsets which were created in the previous step.

Let's take Example-2 mentioned above to go through each step of our algorithm:

```
Given set: [1, 5, 3, 3]
Sorted set: [1, 3, 3, 5]
```

1. Start with an empty set: [[]]
2. Add the first number (1) to all the existing subsets to create new subsets: [[], [1]];
3. Add the second number (3) to all the existing subsets: [[], [1], [3], [1,3]].
4. The next number (3) is a duplicate. If we add it to all existing subsets we will get:

```
    [[], [1], [3], [1,3], [3], [1,3], [3,3], [1,3,3]]
```

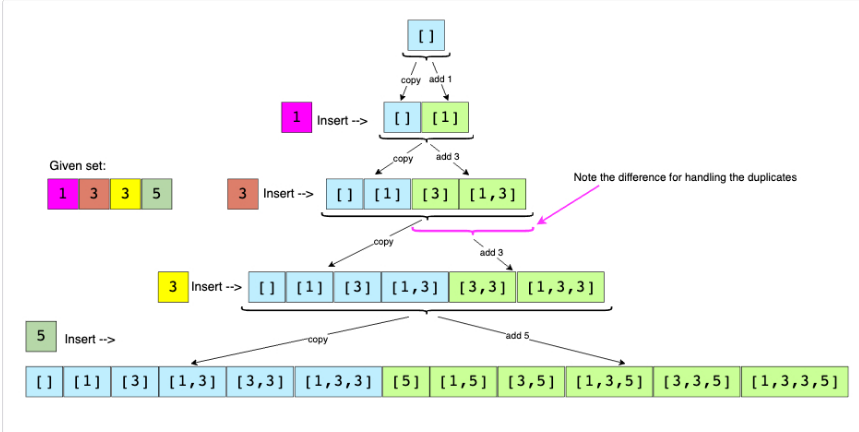
```
We got two duplicate subsets: [3], [1,3]
Whereas we only needed the new subsets: [3,3], [1,3,3]
```

To handle this instead of adding (3) to all the existing subsets, we only add it to the new subsets which were created in the previous (3rd) step:

```
    [[], [1], [3], [1,3], [3,3], [1,3,3]]
```

5. Finally, add the forth number (5) to all the existing subsets: [[], [1], [3], [1,3], [3,3], [1,3,3], [5], [1,5], [3,5], [1,3,5], [3,3,5], [1,3,3,5]]

Here is the visual representation of the above steps:



Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```javascript
function find_subsets(nums) {
  // sort the numbers to handle duplicates
  nums.sort();
  const subsets = [];
  subsets.push([]);
  let startIndex = 0,
    endIndex = 0;
  for (i = 0; i < nums.length; i++) {
    startIndex = 0;
    // if current and the previous elements are same, create new subsets only from the subsets
    // added in the previous step
    if (i > 0 && nums[i] === nums[i - 1]) {
      startIndex = endIndex + 1;
    }
    endIndex = subsets.length - 1;
    for (j = startIndex; j < endIndex + 1; j++) {
      // create a new subset from the existing subset and add the current element to it
      const set = subsets[j].slice(0);
      set.push(nums[i]);
      subsets.push(set);
    }
  }
  return subsets;
}


console.log('Here is the list of subsets: ');
let result = find_subsets([1, 3, 3]);
```

RUN | SAVE | RESET

Time complexity

Since, in each step, the number of subsets could double (if not duplicate) as we add each element to all the existing subsets, the time complexity of the above algorithm is $O(2^N)$, where 'N' is the total number of elements in the input set. This also means that, in the end, we will have a total of $O(2^N)$ subsets at the most.

Space complexity

All the additional space used by our algorithm is for the output list. Since at most we will have a total of $O(2^N)$ subsets, the space complexity of our algorithm is also $O(2^N)$.

Create

Where to Go from Here