Non-Constructible Change

Understanding the problem

We are given an array of positive integers, which represent the values of coins that we have in our possession. The array could have duplicates. We are asked to write a function that returns the minimum amount of change that we cannot create

Approach 1: Brute Force

Iterate from 1 to the maximum amount of change that I can create, which is the value that all of the coins in the input array sum up to. At each iteration, find out if we can sum up to the current value using our coins; if we can't then the minimum amount of change is found. If we get out of the loop without returning the result, return maximum amount of change + 1.

To find out if there is a subset in the input array that adds up to a target value, I can first sort the array in ascending order, then iterate through every integer in the sorted array, starting from the last one; for each integer, compare it with the target value, if the integer is equal to the target value, then the subset is found; if the integer is smaller than the target value, subtract it from the target value and update the target value to the result, move to the next integer. If the very beginning of the input array is reached and the target value is still larger than 0, then I find the value that no subset can add up to.

Implementation

```
function nonConstructibleChange(coins) {
  coins.sort((a, b) => a - b);
  const maximumSum = coins.reduce((sum, value) => sum + value, 0);

  for (let sum = 1; sum < maximumSum; sum++) {
    let currentSum = sum;
    for (let i = coins.length - 1; i >= 0; i--) {
      const currentValue = coins[i];
      if (currentValue <= currentSum) {
        currentSum -= currentValue;
      }
      if (currentSum === 0) {
        break;
      }
    }

    if (currentSum > 0) {
      return sum;
    }
  }
```

```
    return maximumSum + 1;
  }
```

Time & Space Complexity

O(m * n) time | O(1) space, where n is the number of coins and m is the sum that all of the coins add up to.

The O(nlog(n)) runtime of the sorting step is not reflected in the final time complexity, because the nested loops take O(m * n) time. The input array contains only positive integers, so the minimum value we can have is `1`. Suppose the number of coins is `10` and all the coins are `1`. The sum of all the coins is going to be `10`. Therefore `m` must be equal to or greater than `n`. When `m = n`, `O(n * log(n) + m * n) = O(n * log(n) + n * n) = O(n * n)`, since `n * n` is always greater than `n * log(n)`. When `m > n`, `O(n * log(n) + m * n) = O(n * (log(n) + m)) = O(n) * O(log(n) + m) = O(n * m)`, since `(m + log(n)) < (2 * m)`.

Approach 2: Optimized

If we don't know how to solve an array problem, the first thing we usually can do is sort the input array and see what that looks like. The sample input is `[5, 7, 1, 1, 2, 3, 22]`. After sorting it, we get `[1, 1, 2, 3, 5, 7, 22]`. Let's walk through the sorted array and try to solve the problem.

```
[1, 1, 2, 3, 5, 7, 22]
 ^
```

Now we are at index 0, and we get `1`, so the amount of change we can create is `1`, we move on to the next integer.

```
[1, 1, 2, 3, 5, 7, 22]
    ^
```

The change we can create is `2 (1 + 1)`.

```
[1, 1, 2, 3, 5, 7, 22]
       ^
```

The change we can create are `4 (1 + 1 + 2)` and `3 (1 + 2)`. That means we are able to make any amount of change from `1` to `4`.

```
[1, 1, 2, 3, 5, 7, 22]
          ^
```

The change we can create are `7 (1 + 1 + 2 + 3)`, `6 (1 + 2 + 3)` and `5 (2 + 3)`, so we can make any amount of change from `1` to `7`.

```
[1, 1, 2, 3, 5, 7, 22]
             ^
```

The change we can create are `12 (1 + 1 + 2 + 3 + 5)`, `11 (1 + 2 + 3 + 5)`, `10 ( 2 + 3 + 5)`, `9 (1 + 1 + 2 + 5)` and `8(1 + 2 + 5)`, so now we can make any amount of change from `1` to `12`.

```
[1, 1, 2, 3, 5, 7, 22]
                ^
```

The change we can create are `19 (1 + 1 + 2 + 3 + 5 + 7)`, `18 (1 + 2 + 3 + 5 + 7)`, `17 (2 + 3 + 5 + 7)`, `16 (1 + 1 + 2 + 5 + 7)`, `15 (1 + 2 + 5 + 7)`, `14 (1 + 1 + 2 + 3 + 7)`, `13 (1 + 2 + 3 + 7)` so now we can make any amount of change from `1` to `19`. This also means, we can make previous amount of change plus the current integer, which is `7`. We were able to make `12`, add `7` to it, we get `19`; we were also able to make `11`, add `7` to it, we get `18`; `10 + 7 = 17`; `9 + 7 = 16`; ...

```
[1, 1, 2, 3, 5, 7, 22]
                   ^
```

The change we can create are `19 + 22 = 41`, `18 + 22 = 40`, `17 + 22 = 39`, `16 + 22 = 38`,… `1 + 22 = 23`, `22` and the previous amount of change: `19` … `1`. We are not able to make `20`, `21`. So we find the minimum amount of change we cannot create, which is `20`.

Let's look at another example:

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

The change we can make is `1`.

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

The change we can make are `1` and `2`.

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

The change we can make are `1`, `2`, `3`, `4`, `5`.

Now we replace `3` with `4` to see what happens.

```
[1, 1, 4]
 ^
```

The change we can make is `1`.

```
[1, 1, 4]
    ^
```

The change we can make are `1` and `2`.

```
[1, 1, 4]
       ^
```

The change we can make are `1`, `2`, `4`, `5`. We cannot make `3`, because the new value we get is `4`, which means the minimum new amount of change we can make is `4`. If we replace `4` with `2` or `1`, than we can make `3`.

We can arrive at the following conclusion:

Imagine we have a set of coin called `U`. In this set we have now one coin and its value is 1: `U = {1}`. Imagine we have another value called `C`, representing the change we can create with our coins: `C = 1`. We have also `V` which represents the new coin we add to our set. If `V > C + 1`, we cannot make `C + 1` change; if `V <= C + 1`, we can make from `C` to `C + V` change, and the minimum amount of change we cannot create is `C + V + 1`.

So to solve the problem, what we need to do is:

- Sort the input array in ascending order,
- Use a variable to keep track of the current change we create. Initiate it to 0.
- Iterate through every integer in the sorted array.
  - For every integer, compare it to the current change, if it is greater than the current change, we found the minimum amount of change we cannot make, which is current change we make plus 1, return the result; otherwise, add the integer to the current change.
- If we get out of the loop without returning the result, return current change we make plus 1.

Implementation

JavaScript:

```
function nonConstructibleChange(coins) {
  coins.sort((a, b) => a - b);

  let currentChangeCreated = 0;
  for (const coin of coins) {
    if (coin > currentChangeCreated + 1) {
      return currentChangeCreated + 1;
    }

    currentChangeCreated += coin;
  }

  return currentChangeCreated + 1;
}
```

Go:

```
package main

import (
	"sort"
)

func NonConstructibleChange(coins []int) int {
	sort.Ints(coins)

	currChangeCreated := 0

	for _, coin := range coins {
		if coin-currChangeCreated > 1 {
			break
		}

		currChangeCreated += coin
	}
```

```
        return currChangeCreated + 1
}
```

Time & Space Complexity

O(nlog(n)) time | O(1) space, where n is the number of coins.