Planters Problem

Reasoning

We must first move the biggest plant to a new planter, since it's pot will be fillable by later plants. We could make this easier by first sorting both arrays, from largest to smallest. Then we move the biggest from the array of plants, by adding it with its new pot size to the output array, then we would remove its new pot from the new pot array by moving the starting pointer for the new pot array up. We would go through this for each pot in the pots array, checking if the plant could be moved either to the next pot in the new pot array or one of the previous pots in the old pot array. We would return true if we get through the whole thing, and false otherwise.

Psudocode

```
function move_plants (old_pots, new_pots) {
    old_pots = mergeSort(old_pots)
    new_pots = mergeSort(new_pots)

    old_pot_pointer = 0
    new_pot_pointer = 0

    for i from 0 to length of old_pots {
        pot = old_pots[i]
        new_pot = new_pots[new_pot_pointer]
        old_pot = old_pots[old_pot_pointer]
        if (pot < new_pot) {
            output.append(new_pot)
            increase new_pot_pointer by one
        }
        // if the plant can fit in said pot and the old pot pointer is
before our current pointer
        else if (pot < old_pot and i > old_pot_pointer) {
            output.append(old_pot)
            increase old_pot_pointer by one
        }
        else {
            return false
        }
    }
}
```

Proof

This algorithm will work, since when we reach each plant in the now sorted array, it is the biggest plant left un-potted. This means that it can take the largest possible pot, without distrupting any other plant that needs to be potted. And if it does distrupt another plant, then there is no way to

pot both plants. We check both the potted array, and the input array, up to the point that we have repotted to, so if it cannot fit in either, then there is no way to pot it.

Running Time

Estimate

$$O(n \log n)$$

Reasoning

```
function move_plants (old_pots, new_pots) {
    old_pots = mergeSort(old_pots) // Takes O(n log n) time
    new_pots = mergeSort(new_pots) // Takes O(n log n) time

    old_pot_pointer = 0
    new_pot_pointer = 0

    for i from 0 to length of old_pots { // Takes O(n) time
        pot = old_pots[i]
        new_pot = new_pots[new_pot_pointer]
        old_pot = old_pots[old_pot_pointer]
        if (pot < new_pot) {
            output.append(new_pot)
            increase new_pot_pointer by one
        }
        // if the plant can fit in said pot and the old pot pointer is
before our current pointer
        else if (pot < old_pot and i > old_pot_pointer) {
            output.append(old_pot)
            increase old_pot_pointer by one
        }
        else {
            return false
        }
    }
}
```

The ammount of time this takes is $O(n \log n) + O(n) + O(1)$ which is equivilent to $O(n \log n)$ time. This is because the longest part is sorting the arrays, and the rest of my algorithm only takes $O(n)$ time.