#5: Combination Sum – (CS 4A: Fall 2024)

Kelly Dempster: Due Date: 10/27/24

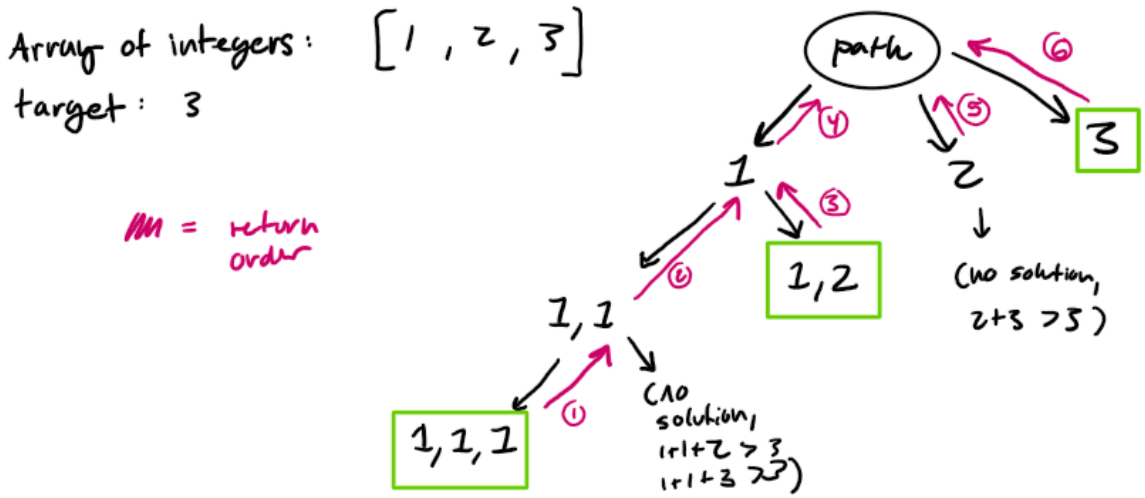
The problem statement:

Given an array of integers and a sum target, find all possible/unique combinations of the numbers in the array which add up to the target. A number can be used as many times as necessary. Show the combinations in a list, or output to console.

1. High-Level Description

To solve this problem, the array of integers will be sorted from least to greatest before calling the recursive solution function (titled *CombinationSum*). The sorted array, target (now called difference), created path list, and starting index of zero are passed into the first call which starts the recursion. A number is chosen from the array and subtracted from the target. The base case is when the "difference" is equal to zero, meaning the pathway taken adds up exactly to the target. If not yet at the base case, the function will move through a for-loop, which starts at the passed in index. In the loop, if the next number to be added doesn't go over the target (difference is greater than or equal to zero), then the number will be added to the path list. Then, a recursive call is made with the new path list and the index of the number in the array which was just added. This forms a new branch or possible solution. If the branch ends up not reaching a solution, the call is returned back to where the number was added, the number is removed, and the process repeats with the next number in the array.

The backtracking part is where the number is added to the path, that path is tried to see if it reaches a solution, and if no solution is reached, it backtracks by removing the number from the path and trying again with the next number. See the example below:



2. Model

The path list, difference from the target, the array of numbers, and the index are passed between each call. What the passed index does is make the pool of numbers possible to chose from smaller. Instead of creating a new array of numbers every time the pool is made smaller, it passes a new starting point in the array for the function to continue from, which saves space and processing time, and part of why the array is sorted before being used.

3. Java Source Code

```
import java.util.ArrayList;
import java.util.Arrays;
public class ComboSum {

    /* Requires an array of positive, non-zero integers WITHOUT repeating
      numbers */
    public static void CombinationSum (int[] array, int target) {
        ArrayList<Integer> path = new  ArrayList<Integer>();

        Arrays.sort(array);

        System.out.println("Solutions: ");
        // index zero for start since all numbers are valid candidates
        CombinationSum(path, array, target, 0);
        return;
    } // END combination sum (caller)

    /* Helper: Difference starts as target, and has numbers gradually
     * subtracted from it, until it reaches zero (meaning a path has been
     * found), it runs out of possible combinations, or the difference is
     * negative, meaning the path overshot the target.  */
    public static void CombinationSum (ArrayList<Integer> path, int[] array,
                                       int difference, int index) {
        if (difference == 0){
            PrintPath(path);
            return;
        }

        for (int i = index; i < array.length; i++) {
            if (difference - array[i] >= 0){

                path.add(array[i]);

                // recursive call with next valid option
                CombinationSum(path, array, difference - array[i], i);

                // backtracks by popping the last number added to the list
```

```
                // (undoing the add action)
                path.remove(path.size() - 1);

            }
        }
    } // END combination sum

    public static void PrintPath(ArrayList<Integer> path) {
        int i;
        for (i = 0; i < path.size() - 1; i++) {
            System.out.print(path.get(i) + " + ");
        }
        System.out.print(path.get(i));
        System.out.println();
    } // END PrintPath


} // END class
```

4. Testing Program

```
public static void main(String[] args) {

    System.out.println("Test 1: Target 3");
    int[] test1 = {1, 2, 3};
    int sum1 = 3;
    CombinationSum(test1, sum1);

    System.out.println("\n\nTest 2: No possible solutions");
    int[] test2 = {7, 6, 9};
    int sum2 = 4;
    CombinationSum(test2, sum2);

    System.out.println("\n\nTest 3: Larger Array and sum");

    int[] test3 = {4, 2, 3, 8, 7};
    int sum3= 16;
    CombinationSum(test3, sum3);

} // END driver main
... used in ComboSum class
```

5. Example Inputs and Outputs

Array: 1, 2, 3 Target: 3

Output:

1 + 1 + 1
1 + 2
3

Array: 7, 6, 9 Target: 4

Output: (None, no solution)

Array: 4, 2, 3, 8, 7 Target: 16

Output:

2 + 2 + 2 + 2 + 2 + 2 + 2 + 2
2 + 2 + 2 + 2 + 2 + 2 + 4
2 + 2 + 2 + 2 + 2 + 3 + 3
2 + 2 + 2 + 2 + 4 + 4
2 + 2 + 2 + 2 + 8
2 + 2 + 2 + 3 + 3 + 4
2 + 2 + 2 + 3 + 7
2 + 2 + 3 + 3 + 3 + 3
2 + 2 + 4 + 4 + 4
2 + 2 + 4 + 8
2 + 3 + 3 + 4 + 4
2 + 3 + 3 + 8
2 + 3 + 4 + 7
2 + 7 + 7
3 + 3 + 3 + 3 + 4
3 + 3 + 3 + 7
4 + 4 + 4 + 4
4 + 4 + 8
8 + 8

- **Examples Directly from Running Program:**

Test 1: Target 3
Solutions which add to 3:
1 + 1 + 1
1 + 2
3

Test 2: No possible solutions
Solutions which add to 4:

Test 3: Larger Array and sum
Solutions which add to 16:
2 + 2 + 2 + 2 + 2 + 2 + 2 + 2

```
2 + 2 + 2 + 2 + 2 + 2 + 4
2 + 2 + 2 + 2 + 2 + 3 + 3
2 + 2 + 2 + 2 + 4 + 4
2 + 2 + 2 + 2 + 8
2 + 2 + 2 + 3 + 3 + 4
2 + 2 + 2 + 3 + 7
2 + 2 + 3 + 3 + 3 + 3
2 + 2 + 4 + 4 + 4
2 + 2 + 4 + 8
2 + 3 + 3 + 4 + 4
2 + 3 + 3 + 8
2 + 3 + 4 + 7
2 + 7 + 7
3 + 3 + 3 + 3 + 4
3 + 3 + 3 + 7
4 + 4 + 4 + 4
4 + 4 + 8
8 + 8

Process finished with exit code 0
```