Recursion Problem Solutions: Group B

1. Given a number n, generate all distinct ways to write n as the sum of positive integers. For example, with $n = 4$, the options are 4, 3 + 1, 2 + 2, 2 + 1 + 1, and 1 + 1 + 1 + 1.

The main challenge in solving this problem is figuring out how to avoid getting duplicate solutions like 3 + 1 and 1 + 3. The approach I've outlined in this solution works by always generating the numbers in nonincreasing order, so we will never generate solutions like 1 + 3.

Here's a solution that works by strengthening the recursion. We will actually answer the harder question "list all distinct ways to write n as the sum of positive integers, where all the integers in the sum are less than or equal to k." Given this, we can make the following insight:

- There is just one way to write 0 as the sum of integers up to and including k: namely, it's the empty sum of no numbers.
- Otherwise, the sum must have at least one integer in it. For each possible first integer i (which must be in the range of 1 to the minimum of n and k), try writing the number as the sum of i, plus all possible ways of summing to $n - i$ using numbers no greater than i.

This gives the following recursive solution:

```
function allPartitionsOf(n) {
      return restrictedAllPartitions(n, n);
}
function restrictedAllPartitions(n, k) {
      if n is zero, the only solution is the empty sum.
      for i from 1 to the minimum of n and k, inclusive:
            record all solutions of the form i + restrictedAllPartitions(n - i, i)
      return those solutions
}
```

This solution will recompute many subproblems, so you could consider using memoization or DP to try to avoid the recomputations. This unfortunately increases the memory usage, so it's up to you to decide whether it's a good idea or not.

2. In a binary tree, a *common value subtree* is a complete subtree where every node has the same value. (A complete subtree is a subtree consisting of a node and all its children). Determine the largest common value subtree in a nonempty binary tree.

For notational simplicity, let's call a common value subtree a CVS. The key observation necessary here is the following:

- A tree with just one node is a CVS whose value is the value in the root.
- A tree with root r and subtrees t_1, t_2, …, t_n is a CVS if each of t_1, t_2, …, t_n is a CVS and has the same value as the root's value.

This means that we can do a bottom-up pass over the tree and find all of the CVS's as we go. We can then determine which CVS found this way is the largest. The pseudocode below splits this into several passes over the tree just for simplicity, and can be significantly space-optimized by combining everything together into one pass.

```
function findLargestCVS(t) {
       annotateTreeSizes(t)
       annotateTreeCVS(t)
       return largestCVSIn(t)
}
function annotateTreeSizes(t) {
       if t is a leaf, set t.size = 1
       else:
         call annotateTreeSizes(c) for each child c of t.
         set t.size = 1 + sum(c.size) for each child c of t
}
function annotateCVS(t) {
       if t is a leaf, set t.isCVS to true.
       else:
         call annotateCVS(c) for each child c of t.
         set t.isCVS to whether c.value = t.value and c.isCVS for each child c of t.
}
function largestCVSIn(t) {
       if t.isCVS return t.size
       else
         let dₖ = largestCVSIn(cₖ) for each child cₖ of t.
         return argmax{ dₖ.size } for all dₖ.
}
```

This code runs in time O(n), where n is the number of nodes in the trees. To see this, note that every function call takes time O(1 + num children), so summing up across all nodes in the tree we can charge O(1) total work to each node in the tree. Summing up across all n nodes gives a runtime of O(n).

3. Suppose you have a multiway tree where each node has an associated integer value. Find a set of nodes with the maximum possible sum, subject to the constraint that you cannot choose a node and any of its children at the same time.

The insight necessary to solve this problem is to split the problem into two cases – first, solving this problem when the root node *is* included in the solution, and second when the root node is *not* included in the solution. If we include the root node, then all of its children must not be included, and we'd like optimal solutions for each of its subtrees subject to the restriction that their root nodes aren't included. If we exclude the root node, then for each child, we should take the best solution possible, whether or not we choose to include the root node.

If we code this up using a naïve recursion, we get this solution:

```
function maxSum(t) {
      return maxSumUnrestricted(t)
}

// Gives the largest possible sum that can be made with the tree rooted at t when
// there are no restrictions on whether t must be included.
function maxSumUnrestricted(t) {
      return the max of maxSumRestricted(t, true) and maxSumRestricted(t, false);
}

// Gives the largest possible sum that can be made with the tree rooted at t
// subject to the restriction that the root of t either must be or must not be
// included
function maxSumRestricted(t, include) {
   if t is null, return 0.

   if mustInclude is true:
      return t.value + sum of maxSumRestricted(c, false) for all children c of t.

   if mustInclude is false:
      return the sum of maxSumUnrestricted(c) for all children c of t
}
```

This recursion is correct but highly inefficient – we'll end up recomputing the same subproblems on many subtrees, leading to a very slow runtime. However, there are only $2n$ possible unique calls in a tree of n nodes – for each node, we can either include it or exclude it – and so we can memoize the results or use dynamic programming to compute the results bottom-up. If we do this, the net runtime is only $O(n)$ because we do $O(1)$ work per node $O(n)$ total times, though it now requires $O(n)$ storage space.

4. Suppose that you have a rectangular box with several vertical partitions in it. The ends of the box are open. Suppose you pour an enormous volume of water onto the box. Water will become trapped between the partitions in the box. Given the heights of the partitions, determine the heights of the water between the partitions.

There are many possible solutions to this problem. First, I'll describe a simple recursive solution to the problem. Then, I'll give an alternate solution that is more efficient.

One observation that might be useful in solving this problem is that if you know where the largest partition is, you can split the box in half at that partition. Once you've done that, you can do the following:

- Find the largest element within that piece.
- Fill the partitions between that element and the maximum element to the top with water.
- Recursive repeat this process on the other half of the array.

A more efficient solution is the following. Start off by finding the largest partition, as before. Split the array into a left half and a right half. The logic in these halves are the same, just run in different directions, so I'll only give it for the left half. Walk from the left side of the array toward the largest element, creating an array tracking, for each partition, the height of the largest partition to its left. This is initially 0, and afterwards can be filled in by taking the maximum of the previous partition and the maximum value so far. Once this is filled in, each partition's water height can be determined by the height of the largest partition to its left. This approach requires $O(n)$ time to find the maximum value initially, then $O(n)$ time to compute the partition heights (which are equal to the water heights), for a total of $O(n)$ time.