1. In the *sum-of-subsets problem* you are given as input:
- a set of n positive integers (weights) $\{w_1, w_2, w_3, \ldots, w_n\}$, and
- a target sum, W

with the task of finding all subsets that sum to the target sum, W.

Consider an instance of the sum-of-subsets problem: For the weights of { 5, 6, 10, 11, 16 }, find all the subsets adding to 21.

To solve a problem using backtracking, you need to answer the following questions:

a) What should the state-space tree look like? (i.e.,What would the "for each child c" loop iterate over?)

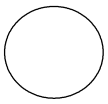
b) What state information is needed at each node?

Global Problem-Instance Information

	1	2	3	4	5
weights:	5	6	10	11	16

n: 5 W: 21

Starting Node - original call to start recursive backtracking function



c) Any alternate state-space tree which might be better for exploring subsets?

d) Without some pruning criteria (check for "promising" child node), how many nodes are in both of the above state-space trees?

e) What criteria can be used to determine if a child node (c) is NOT promising?

f) What information is needed by our promising function?

2. Consider customizing the Backtrack template for the sum-of-subsets problem. Use a single, "global", current-node state which is updated before we go down to the child (via a recursive call) and undone when we backtrack to the parent.

```
Backtrack( recursionTreeNode p ) {

    treeNode c;
    for each child c of p do                                    # each c represents a possible choice
        if promising(c) then                                        # c is "promising" if it could lead to a better solution
            if c is a solution that's better than best then     # check if this is the best solution found so far
                best = c                                        # remember the best solution
            else
                Backtrack(c)                                        # follow a branch down the tree
            end if
        end if
    end for
} // end Backtrack
```