Handling "Hard" Problems: For many optimization problems (e.g., TSP, knapsack, job-scheduling), the best known algorithms have run-time's that grow exponentially. Thus, you could wait centuries for the solution of all but the smallest problems!

Ways to handle these "hard" problems:

- Find the best (or a good) solution "quickly" to avoid considering the vast majority of the \(2^n\) worse solutions, e.g., Backtracking (Chapter 5) and Branch-and-Bound (Chapter 6)
- See if a restricted version of the problem meets your needed that might have a tractable (polynomial, e.g., \(\Theta(n^3)\)) solution. e.g., Fractional Knapsack problem, TSP problem satisfying the triangle inequality
- Use an approximation algorithm to find a good, but not necessarily optimal solution

**Backtracking** (chapter 5) general idea:

- Search the "state-space tree" using depth-first search to find a suboptimal solution quickly
- Use the best solution found so far to prune partial solutions that are not "promising," i.e., cannot lead to a better solution than one already found.

The goal is to prune enough of the state-space tree (exponential is size) that the optimal solution can be found in a reasonable amount of time. However, in the worst case, the algorithm is still exponential.

1. Consider an instance of the coin-change problem: For coin types of \{1, 5, 10, 12, 25, 50\} and a change amount of 29-cents, determine the **fewest number** of coins for this amount of change.

   - a) Draw a little more of the recursive state-space tree.
   - b) How would you recognize a solution?
   - c) What would be the first solution found with a depth-first search of this state-space tree?

<table>
<thead>
<tr>
<th>Order of Coins</th>
<th>Change Amount 29-cents</th>
<th>Change Amount 399-cents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fewest # coins</td>
<td>Execution time (sec.)</td>
</tr>
<tr>
<td>1 5 10 12 25 50</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>50 25 12 10 5 1</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

   d) Why does the order of coins make such a big difference in the number of nodes in the state-space tree?
The general recursive backtracking algorithm for optimization problems (e.g., TSP, knapsack, job-scheduling) looks something like:

```
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
            best = c # remember the best solution
        else
            Backtrack(c) # follow a branch down the tree
    end if
end for

} // end Backtrack
```

General Notes about Backtracking:

- The depth-first nature of backtracking only stores information about the current branch being explored so the memory usage is "low"
- Each node of the state-space (recursive-call) tree maintains the state of a partial solution. In general the partial solution state consists of potentially large arrays that change little between parent and child. To avoid having multiple copies of these arrays, a single "global" state is maintained which is updated before we go down to the child (via a recursive call) and undone when we backtrack to the parent.
- We could use the concept of backtracking without recursion by using a stack to maintain a collection of unexplored choices. Thus, we would simulate the run-time stack to drive the backtracking algorithm.

2. Consider customizing the above Backtrack template for the coin-change problem.
   a) What would the "for each child c" loop iterate over? (What problem instance information is needed?)
   b) What two criteria can be used to determine if a child node (c) is NOT promising?
   c) What state information is needed at each node?
   d) What information is needed by our promising function?