1. The textbook solves the coin-change problem with the following code (note the "set-builder-like" notation):

\[ \{ c \mid c \in \text{coinValueList} \text{ and } c \leq \text{change} \} \]

Results of running this code:

- Change Amount: 63
- Coin types: [1, 5, 10, 25]
- Run-time: 70.689 seconds
- Fewest number of coins: 6
- Number of Backtracking Nodes: 67,716,925

I removed the fancy set-builder notation and replaced it with a simple if-statement check:

```python
def recNC(change, coinValueList):
    global backtrackingNodes
    backtrackingNodes += 1
    minCoins = change
    if change in coinValueList:
        return 1
    else:
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + recNC(change - i, coinValueList)
            if numCoins < minCoins:
                minCoins = numCoins
        return minCoins
```

Results of running this code:

- Change Amount: 63
- Coin types: [1, 5, 10, 25]
- Run-time: 45.815 seconds
- Fewest number of coins: 6
- Number of Backtracking Nodes: 67,716,925

a) Why is the second version so much "faster"? The first version builds a new list of coin values on each of 67,716,925 recursive calls (except base cases). Second version use if-statement to avoid building a new list.

b) Why does it still take a long time? --- still 67,716,925 recursive calls.

2. To speed the recursive backtracking algorithm, we can prune unpromising branches. The general recursive backtracking algorithm for optimization problems (e.g., fewest number of coins) looks something like:

```python
Backtrack(recursionTreeNode p) {
    for each child c of p do
        if promising(c) then
            if c is a solution that's better than best then
                best = c
            else
                Backtrack(c)
        end if
    end if
} // end Backtrack
```

General Notes about Backtracking:
- The depth-first nature of backtracking only stores information about the current branch being explored on the run-time stack, so the memory usage is "low" eventhough the # of recursion tree nodes might be exponential (2^n).
- Each node of the search-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 reference to a single "global" array can be maintained which is updated before we go down to the child (via a recursive call) and undone when we backtrack to the parent.

a) For the coin-change problem, what defines the current state of a search-space tree node?
4. As with Fibonacci, the coin-change problem can benefit from dynamic program since it was slow due to solving the same problems over-and-over again. Recall the general idea of dynamic programming:

- Solve smaller problems before larger ones
- Store their answers
- Look-up answers to smaller problems when solving larger subproblems, so each problem is solved only once

a) To solve the coin-change problem using dynamic programming, we need to answer the questions:

- What is the smallest problem?
- Where do we store the answers to the smaller problems?
Dynamic Programming Coin-change Algorithm:

I. Fills an array fewestCoins from 0 to the amount of change. An element of fewestCoins stores the fewest number of coins necessary for the amount of change corresponding to its index value.

For 29-cents using the set of coin types \{1, 5, 10, 12, 25, 50\}, the dynamic programming algorithm would have previously calculated the fewestCoins for the change amounts of 0, 1, 2, ..., up to 28 cents.

II. If we record the best, first coin to return for each change amount (found in the “minimum” calculation) in an array bestFirstCoin, then we can easily recover the actual coin types to return.

\[
fewestCoins[29] = \text{minimum}(fewestCoins[28], fewestCoins[24], fewestCoins[19], fewestCoins[17], fewestCoins[4]) + 1 = 2 + 1 = 3
\]

Extract the coins in the solution for 29-cents from bestFirstCoin[29], bestFirstCoin[24], and bestFirstCoin[12]

b) Extend the lists through 32-cents.

c) What coins are in the solution for 32-cents? 10, 10, 12
1. Consider the following sequential search (linear search) code:

<table>
<thead>
<tr>
<th>Textbook's Listing 5.1</th>
<th>Faster sequential search code</th>
</tr>
</thead>
<tbody>
<tr>
<td>def sequentialSearch(alist, item):</td>
<td>def linearSearch(aList, target):</td>
</tr>
<tr>
<td>&quot;&quot;&quot; Sequential search of unordered list &quot;&quot;&quot;</td>
<td>&quot;&quot;&quot; Returns the index of target in aList or -1 if target is not in aList &quot;&quot;&quot;</td>
</tr>
</tbody>
</table>
| pos = 0 | for position in range(len(aList)):
| found = False | if target == aList[position]:
| while pos < len(alist) and not found:
| if alist[pos] == item:
| found = True
| else:
| pos = pos + 1
| return found | return position |
| | return -1 |

a) What is the **basic operation** of a search? **Comparison** items

b) For the following aList value, which target value causes linearSearch to loop the fewest ("best case") number of times?

```
10  loop once  O(1)
```

<table>
<thead>
<tr>
<th>aList:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>15</td>
<td>28</td>
<td>42</td>
<td>60</td>
<td>69</td>
<td>75</td>
<td>88</td>
<td>90</td>
<td>93</td>
<td>97</td>
</tr>
</tbody>
</table>

```
10  loop once  O(1)
```

c) For the above aList value, which target value causes linearSearch to loop the most ("worst case") number of times? **97 for any item not in aList**

```
O(n)
```

d) For a **successful search** (i.e., target value in aList), what is the "average" number of loops?

```
~5  \( O(\frac{n}{2}) = O(n) \)
```

e) The above version of linear search assumes that aList is sorted in ascending order. When would this version perform better than the original linearSearch at the top of the page?

*Gets to stop early if we find a list item greater than the target since the list is sorted.*
2. Consider the following binary search code:

<table>
<thead>
<tr>
<th>Textbook's Listing 5.3</th>
<th>Faster binary search code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>def binarySearch(alist, item):</code></td>
<td><code>def binarySearch(target, lyst):</code></td>
</tr>
<tr>
<td><code>  first = 0</code></td>
<td><code>  &quot;Returns the position of the target</code></td>
</tr>
<tr>
<td><code>  last = len(alist) - 1</code></td>
<td><code>  item if found, or -1 otherwise.&quot;&quot;</code></td>
</tr>
<tr>
<td><code>  found = False</code></td>
<td><code>  left = 0</code></td>
</tr>
<tr>
<td><code>  while first &lt;= last and not found:</code></td>
<td><code>  right = len(lyst) - 1</code></td>
</tr>
<tr>
<td><code>    midpoint = (first + last) // 2</code></td>
<td><code>  while left &lt;= right:</code></td>
</tr>
<tr>
<td><code>    if alist[midpoint] == item:</code></td>
<td><code>      midpoint = (left + right) // 2</code></td>
</tr>
<tr>
<td><code>      found = True</code></td>
<td><code>      if target == lyst[midpoint]:</code></td>
</tr>
<tr>
<td><code>    else:</code></td>
<td><code>        return midpoint</code></td>
</tr>
<tr>
<td><code>      if item &lt; alist[midpoint]:</code></td>
<td><code>      elif target &lt; lyst[midpoint]:</code></td>
</tr>
<tr>
<td><code>        last = midpoint - 1</code></td>
<td><code>        right = midpoint - 1</code></td>
</tr>
<tr>
<td><code>      else:</code></td>
<td><code>        left = midpoint + 1</code></td>
</tr>
<tr>
<td><code>        first = midpoint + 1</code></td>
<td><code>      return -1</code></td>
</tr>
<tr>
<td><code>  return found</code></td>
<td></td>
</tr>
</tbody>
</table>

a) "Trace" binary search to determine the worst-case basic total number of comparisons?

<table>
<thead>
<tr>
<th>loop</th>
<th>worst-case # elements remaining</th>
<th>left</th>
<th>right</th>
<th>midpoint</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>n</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>51</td>
</tr>
<tr>
<td>2</td>
<td>n/2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>51</td>
</tr>
<tr>
<td>3</td>
<td>n/4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
</tbody>
</table>

b) What is the worst-case big-oh for binary search? \( O(\log_2 n) \)

\( \frac{\log_2 n}{x} = n \)

\( 2^x = n \)

c) What is the best-case big-oh for binary search? \( O(1) \) one of these

d) What is the average-case (expected) big-oh for binary search? \( O(\log_2 n) \)
e) If the list size is 1,000,000, then what is the maximum number of comparisons of list items on a successful search? \( \frac{\log_2 1000000}{20} \)
f) If the list size is 1,000,000, then how many comparisons would you expect on an unsuccessful search? \( 20 \)