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

```python
def recMC(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 + recMC(change - i, coinValueList)
        if numCoins < minCoins:
            minCoins = numCoins
    return minCoins
```

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

```python
def recMC(change, coinValueList):
global backtrackingNodes
backtrackingNodes += 1
minCoins = change
if change in coinValueList:
    return 1
else:
    for i in coinValueList:
        numCoins = 1 + recMC(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**: 70.689 seconds
- **Fewest number of coins**: 6
- **Number of Backtracking Nodes**: 67,716,925

I doesn’t build a new list of coins that doesn’t exceed change amount.

a) Why is the second version so much “faster”? 


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:

```java
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 for
} // 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?
b) When would a "child" tree node NOT be promising?

If we already have a solution (say 4 coins), then stop if we've given back 3 coins and still have a positive change amount.

3. Consider the output of running the backtracking code with pruning (next page) twice with a change amount of 63 cents.

<table>
<thead>
<tr>
<th>Change Amount</th>
<th>Coin types: [1, 5, 10, 25]</th>
<th>Change Amount</th>
<th>Coin types: [25, 10, 5, 1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run-time</td>
<td>0.036 seconds</td>
<td>Run-time</td>
<td>0.003 seconds</td>
</tr>
<tr>
<td>Fewest number of coins</td>
<td>6</td>
<td>Fewest number of coins</td>
<td>6</td>
</tr>
<tr>
<td>The number of each type of coins is:</td>
<td>number of 1-cent coins is 3</td>
<td>The number of each type of coins is:</td>
<td>number of 25-cent coins is 2</td>
</tr>
<tr>
<td>number of 5-cent coins is 0</td>
<td>number of 10-cent coins is 1</td>
<td>number of 5-cent coins is 0</td>
<td></td>
</tr>
<tr>
<td>number of 25-cent coins is 2</td>
<td>number of 1-cent coins is 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of Backtracking Nodes: 4831</td>
<td>Number of Backtracking Nodes: 310</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a) Explain why ordering the coins from largest to smallest produced faster results.

Left-side above finds all 1st solution first which is not good for pruning. Right-side which finds the greedy solution first.

b) For coins of [30, 25, 12, 10, 5, 1] typical timings:

<table>
<thead>
<tr>
<th>Change Amount</th>
<th>Run-Time (seconds)</th>
<th>Number of Tree Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>399</td>
<td>8.88</td>
<td>2,015,539</td>
</tr>
<tr>
<td>409</td>
<td>55.17</td>
<td>12,093,221</td>
</tr>
<tr>
<td>419</td>
<td>318.56</td>
<td>72,558,646</td>
</tr>
</tbody>
</table>

Why the exponential growth in run-time?

Still a lot of redundant calculations in the tree.

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? 0 amount of change

• Where do we store the answers to the smaller problems?
backtrackingNodes = 0  # profiling variable to track number of state-space tree nodes
def solveCoinChange(changeAmt, coinTypes):
    def backtrack(changeAmt, numberOfEachCoinType, numberOfCoinsSoFar, solutionFound, bestFewestCoins, bestNumberOfEachCoinType):
        global backtrackingNodes
        backtrackingNodes += 1
        for index in range(len(coinTypes)):
            smallerChangeAmt = changeAmt - coinTypes[index]
            if promising(smallerChangeAmt, numberOfCoinsSoFar + 1, solutionFound, bestFewestCoins):
                if smallerChangeAmt == 0:  # a solution is found
                    if (not solutionFound or numberOfCoinsSoFar + 1 < bestFewestCoins):  # check if its best
                        bestFewestCoins = numberOfCoinsSoFar + 1
                        bestNumberOfEachCoinType = [] + numberOfEachCoinType
                        bestNumberOfEachCoinType[index] += 1
                        solutionFound = True
                else:  # call child with updated state information
                    smallerChangeAmtNumberOfEachCoinType = [] + numberOfEachCoinType
                    smallerChangeAmtNumberOfEachCoinType[index] += 1
                    solutionFound, bestFewestCoins, bestNumberOfEachCoinType = backtrack(smallerChangeAmt, smallerChangeAmtNumberOfEachCoinType, numberOfCoinsSoFar + 1, solutionFound, bestFewestCoins, bestNumberOfEachCoinType)
                    return solutionFound, bestFewestCoins, bestNumberOfEachCoinType
            else:
                return False
        # end def backtrack
    
def promising(changeAmt, numberOfCoinsReturned, solutionFound, bestFewestCoins):
        if changeAmt < 0:
            return False
        elif changeAmt == 0:
            return True
        else:  # changeAmt > 0
            if solutionFound and numberOfCoinsReturned + 1 >= bestFewestCoins:
                return False
            else:
                return True
        # Body of solveCoinChange

    numberOfEachCoinType = []  # set-up initial "current state" information
    numberOfCoinsSoFar = 0
    solutionFound = False
    bestFewestCoins = -1
    bestNumberOfEachCoinType = None

    numberOfEachCoinType = []
    for coin in coinTypes:
        numberOfEachCoinType.append(0)
    numberOfCoinsSoFar = 0
    solutionFound = False
    bestFewestCoins = -1
    bestNumberOfEachCoinType = None

    solutionFound, bestFewestCoins, bestNumberOfEachCoinType = backtrack(changeAmt, numberOfEachCoinType, numberOfCoinsSoFar, solutionFound, bestFewestCoins, bestNumberOfEachCoinType)

    return bestFewestCoins, bestNumberOfEachCoinType
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.

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

a minimum for 29 given by 5-cent coin

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?
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>
</table>
| def sequentialSearch(alist, item): 
  """ Sequential search of unorder list """ 
  pos = 0
  found = False 
  while pos < len(alist) and not found: 
    if alist[pos] == item: 
      found = True
    else: 
      pos = pos+1
  return found |
| def linearSearch(alist, target): 
  """Returns the index of target in alist 
or -1 if target is not in alist"""
  for position in range(len(alist)): 
    if target == alist[position]: 
      return position
  return -1 |

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

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

```
     0  1  2  3  4  5  6  7  8  9  10
aList: 10 15 28 42 60 69 75 88 90 93 97
```

\[ O(1) \text{ best case} \]

c) For the above alist value, which target value causes linearSearch to loop the most ("worst case") number of times?

For any unsuccessful search \[ O(n) \]

d) For a successful search (i.e., target value in aList), what is the "average" number of loops? \[ O(n/2) = O(n) \]

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?
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>def binarySearch(alist, item):</td>
<td>def binarySearch(target, lyst):</td>
</tr>
<tr>
<td>first = 0</td>
<td>&quot;&quot;&quot;&quot;Returns the position of the target</td>
</tr>
<tr>
<td>last = len(alist)-1</td>
<td>item if found, or -1 otherwise.&quot;&quot;&quot;&quot;</td>
</tr>
<tr>
<td>found = False</td>
<td>left = 0</td>
</tr>
<tr>
<td></td>
<td>right = len(lyst) - 1</td>
</tr>
<tr>
<td>while first&lt;=last and not found:</td>
<td>while left &lt;= right:</td>
</tr>
<tr>
<td>midpoint = (first + last)//2</td>
<td>midpoint = (left + right) // 2</td>
</tr>
<tr>
<td>if alist[midpoint] == item:</td>
<td>if target == lyst[midpoint]:</td>
</tr>
<tr>
<td>found = True</td>
<td>return midpoint</td>
</tr>
<tr>
<td>else:</td>
<td>elif target &lt; lyst[midpoint]:</td>
</tr>
<tr>
<td>if item &lt; alist[midpoint]:</td>
<td>right = midpoint - 1</td>
</tr>
<tr>
<td>last = midpoint-1</td>
<td>else:</td>
</tr>
<tr>
<td>else:</td>
<td>left = midpoint + 1</td>
</tr>
<tr>
<td>first = midpoint+1</td>
<td>return -1</td>
</tr>
<tr>
<td>return found</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># elements remaining</th>
<th>worst-case</th>
<th>left</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>midpoint</th>
<th>0</th>
<th>100</th>
<th>0</th>
<th>right</th>
<th>n-1</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&quot;n&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

c) What is the best-case big-oh for binary search? $O \left( 1 \right)$

d) What is the average-case (expected) big-oh for binary search? $O \left( \log_2 n \right)$

e) If the list size is 1,000,000, then what is the maximum number of comparisons of list items on a successful search?

$$ \sqrt{2^{20}} \quad 2.0 \text{ comparisons} $$

f) If the list size is 1,000,000, then how many comparisons would you expect on an unsuccessful search?

$\sim 20$
3. Hashing Motivation and Terminology:

a) Sequential search of an array or linked list follows the same search pattern for any given target value being searched for, i.e., scans the array from one end to the other, or until the target is found.

If \( n \) is the number of items being searched, what is the average and worst case big-oh notation for a sequential search?

\[
\begin{align*}
\text{average case } & \mathcal{O} \left( \frac{1}{n} \right) \\
\text{worst case } & \mathcal{O} \left( \frac{1}{n} \right)
\end{align*}
\]

b) Similarly, binary search of a sorted array (or AVL tree) always uses a fixed search strategy for any given target value. For example, binary search always compares the target value with the middle element of the remaining portion of the array needing to be searched.

If \( n \) is the number of items being searched, what is the average and worst case big-oh notation for a search?

\[
\begin{align*}
\text{average case } & \mathcal{O} \left( \frac{1}{\log n} \right) \\
\text{worst case } & \mathcal{O} \left( \frac{1}{\log n} \right)
\end{align*}
\]

Hashing tries to achieve average constant time (i.e., \( O(1) \)) searching by using the target’s value to calculate where in the array/Python list (called the hash table) it should be located, i.e., each target value gets its own search pattern. The translation of the target value to an array index (called the target’s home address) is the job of the hash function. A perfect hash function would take your set of target values and map each to a unique array index.

<table>
<thead>
<tr>
<th>Set of Keys</th>
<th>Hash function</th>
<th>Hash Table Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>hash(John Doe) = 6</td>
<td>0</td>
</tr>
<tr>
<td>Philip East</td>
<td>hash(Philip East) = 3</td>
<td>1</td>
</tr>
<tr>
<td>Mark Fienup</td>
<td>hash(Mark Fienup) = 5</td>
<td>2</td>
</tr>
<tr>
<td>Ben Schafer</td>
<td>hash(Ben Schafer) = 8</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

Philip East 3-2939
Mark Fienup 3-5918
John Doe 3-4567
Ben Schafer 3-2187

a) If \( n \) is the number of items being searched and we had a perfect hash function, what is the average and worst case big-oh notation for a search?

\[
\begin{align*}
\text{average case } & \mathcal{O} \left( \frac{1}{n} \right) \\
\text{worst case } & \mathcal{O} \left( \frac{1}{n} \right)
\end{align*}
\]

4. Unfortunately, perfect hash functions are a rarity, so in general many target values might get mapped to the same hash-table index, called a collision.

Collisions are handled by two approaches:

- **open-address** with some rehashing strategy: Each hash table home address holds at most one target value. The first target value hashed to a specify home address is stored there. Later targets getting hashed to that home address get rehashed to a different hash table address. A simple rehashing strategy is linear probing where the hash table is scanned circularly from the home address until an empty hash table address is found.

- **chaining**, **closed-address**, or **external chaining**: all target values hashed to the same home address are stored in a data structure (called a bucket) at that index (typically a linked list, but a BST or AVL-tree could also be used). Thus, the hash table is an array of linked list (or whatever data structure is being used for the buckets)