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

$\{c \mid c \in \text{coinValueList and } c \leq \text{change}\}$

```
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
```

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:

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

a) Why is the second version so much "faster"? doesn't build a new list of coins that doesn't exceed change amount

b) Why does it still take a long time? redundant problems -- whole recursion tree

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:

```
Backtrack( recursionTreeNode p ) {
    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
```

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.

```
Change Amount: 63  Coin types: [1, 5, 10, 25]
Run-time: 0.036 seconds
Fewest number of coins 6
The number of each type of coins is:
number of 1-cent coins is 3
number of 5-cent coins is 0
number of 10-cent coins is 1
number of 25-cent coins is 2
Number of Backtracking Nodes: 4831
```

```
Change Amount: 63  Coin types: [25, 10, 5, 1]
Run-time: 0.003 seconds
Fewest number of coins 6
The number of each type of coins is:
number of 25-cent coins is 2
number of 10-cent coins is 1
number of 5-cent coins is 0
number of 1-cent coins is 3
Number of Backtracking Nodes: 310
```

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

Left-side above finds all 1-¢ solution first which is not good for pruning V.S. right-side which finds the greedy solution first.

b) For coins of [50, 25, 12, 10, 5, 1] typical timings:

Change Amount	Run-Time (seconds)	Number of Tree Nodes
399	8.88	2,015,539
409	55.17	12,093,221
419	318.56	72,558,646

Why the exponential growth in run-time?

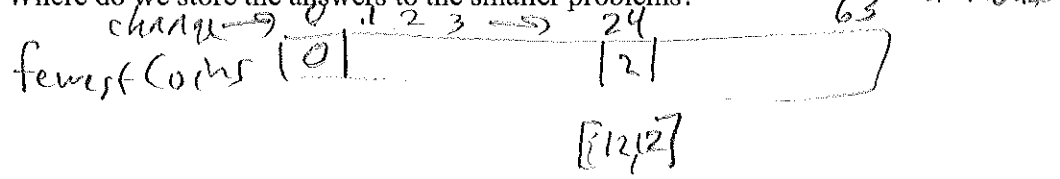
Still a lot of redundant calculations in the tree.

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?



[12/27]

```
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
    # 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
```

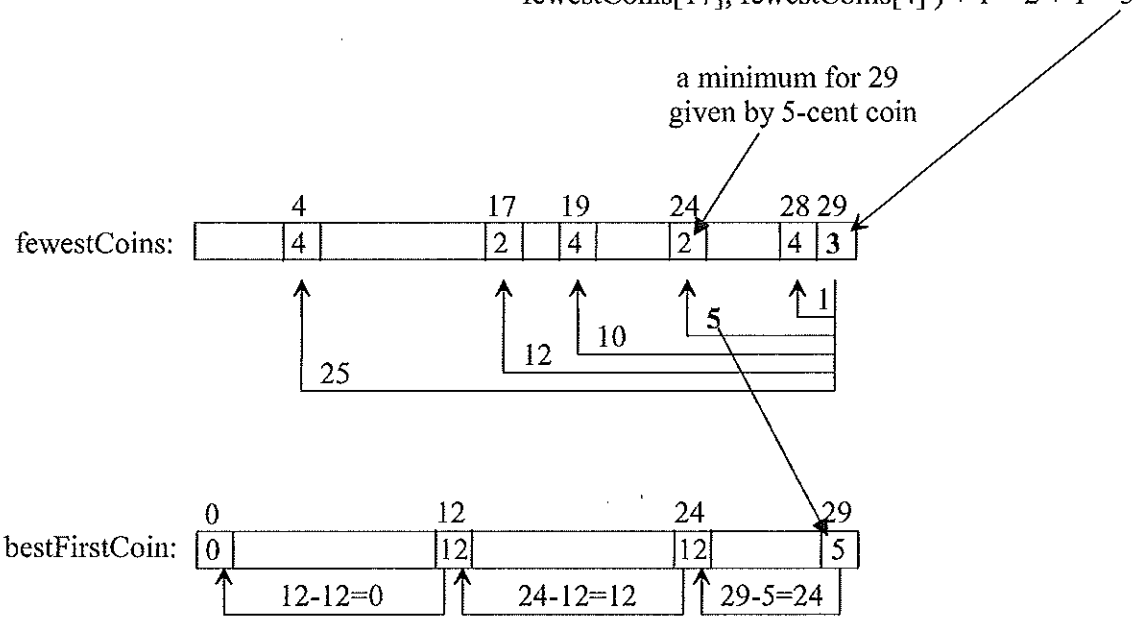
Dynamic Programming Coin-change Algorithm:

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.

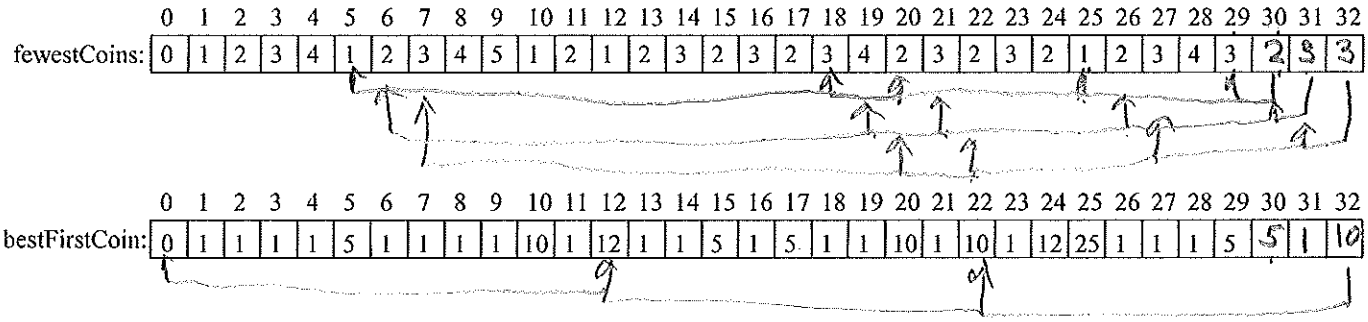
II. If we record the best, first coin to return for each change amount (found in the "minimum" calculation) in an array bestFirstCoin, then we can easily recover the actual coin types to return.

fewestCoins[29] = 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:

Textbook's Listing 5.1	Faster sequential search code

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? 10

O(1) best case

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

c) For the above `aList` value, which `target` value causes `linearSearch` to loop the most ("worst case") number of times? 97 or any unsuccessful search O(n)

d) For a *successful search* (i.e., `target` value in `aList`), what is the "average" number of loops? $n/2$

$O(\frac{n}{2}) = O(n)$

Textbook's Listing 5.2	Faster sequential search code

def orderedSequentialSearch(alist, item):
  """ Sequential search of order list """
  pos = 0
  found = False
  stop = False
  while pos < len(alist) and not found and not stop:
  if alist[pos] == item:
  found = True
  else:
  if alist[pos] > item:
  stop = True
  else:
  pos = pos+1
 	
  return found
	def linearSearchOfSortedList(target, aList):
	  """Returns the index position of target in
	  sorted aList or -1 if target is not in aList"""
	  breakOut = False
	  for position in range(len(aList)):
	  if target <= aList[position]:
	  breakOut = True
	  break
	
	  if not breakOut:
	  return -1
	  elif target == aList[position]:
	  return position
	  else:
	  return -1

if =:

elif <

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:

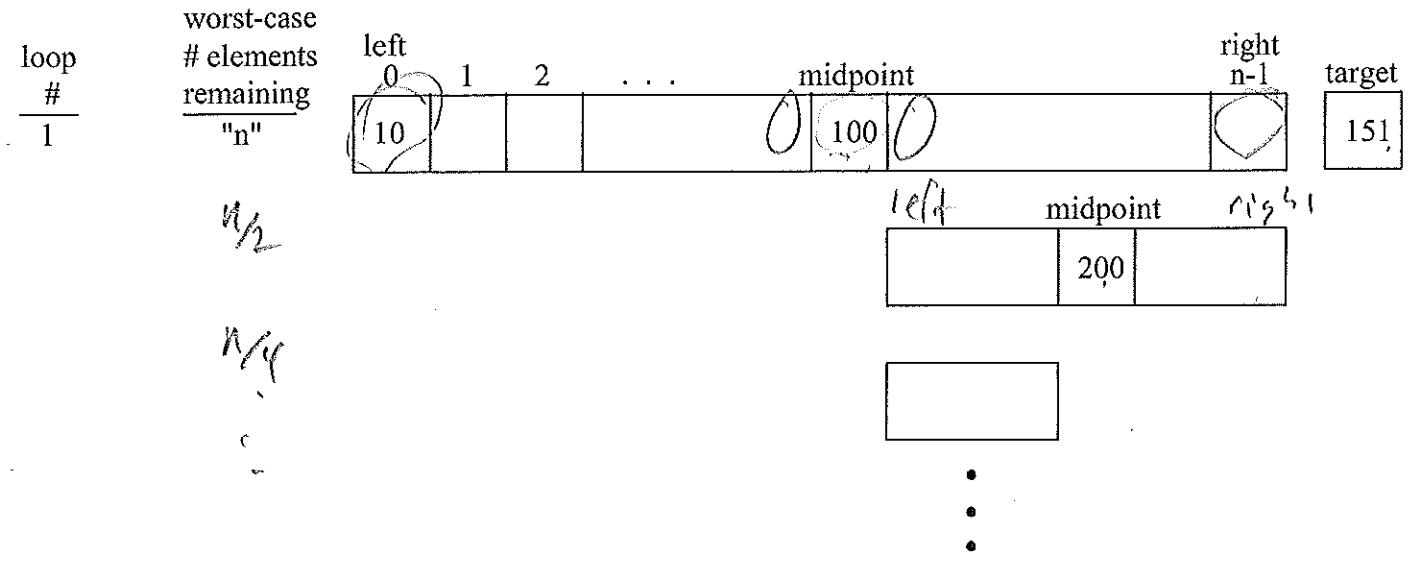
Textbook's Listing 5.3

```
def binarySearch(alist, item):
    first = 0
    last = len(alist)-1
    found = False

    while first<=last and not found:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
            found = True
        else:
            if item < alist[midpoint]:
                last = midpoint-1
            else:
                first = midpoint+1

    return found
```

Faster binary search code

```
def binarySearch(target, lyst):
    """Returns the position of the target
       item if found, or -1 otherwise."""
    left = 0
    right = len(lyst) - 1
    while left <= right:
        midpoint = (left + right) // 2
        if target == lyst[midpoint]:
            return midpoint
        elif target < lyst[midpoint]:
            right = midpoint - 1
        else:
            left = midpoint + 1
    return -1
```

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



b) What is the worst-case big-oh for binary search? $O(\log_2 n)$ $2^? = n$

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

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 maximun number of comparisons of list items on a *successful search*?

2^{20} 20 comparison

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

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?

average case $O(\ n\ )$

worst case $O(\ n\ )$

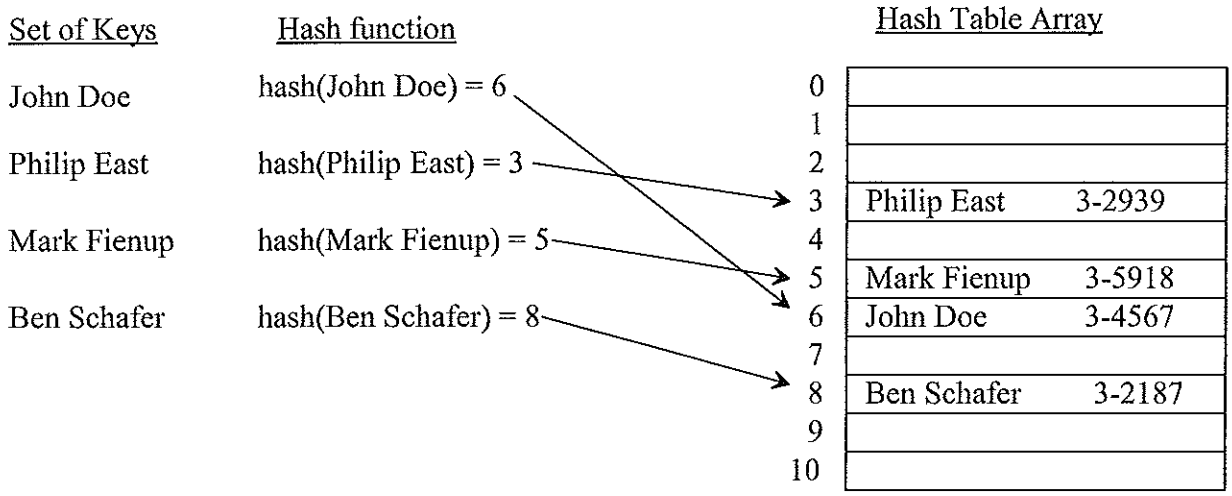
b) Similarly, binary search of a sorted array (or AVL tree) always uses a fixed search strategy for any given target value. For example, binary search always compares the target value with the middle element of the remaining portion of the array needing to be searched.

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

average case $O(\ \log_2 n\ )$

worst case $O(\ \log_2 n\ )$

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.



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?

average case $O(\ 1\ )$

worst case $O(\ 1\ )$

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)