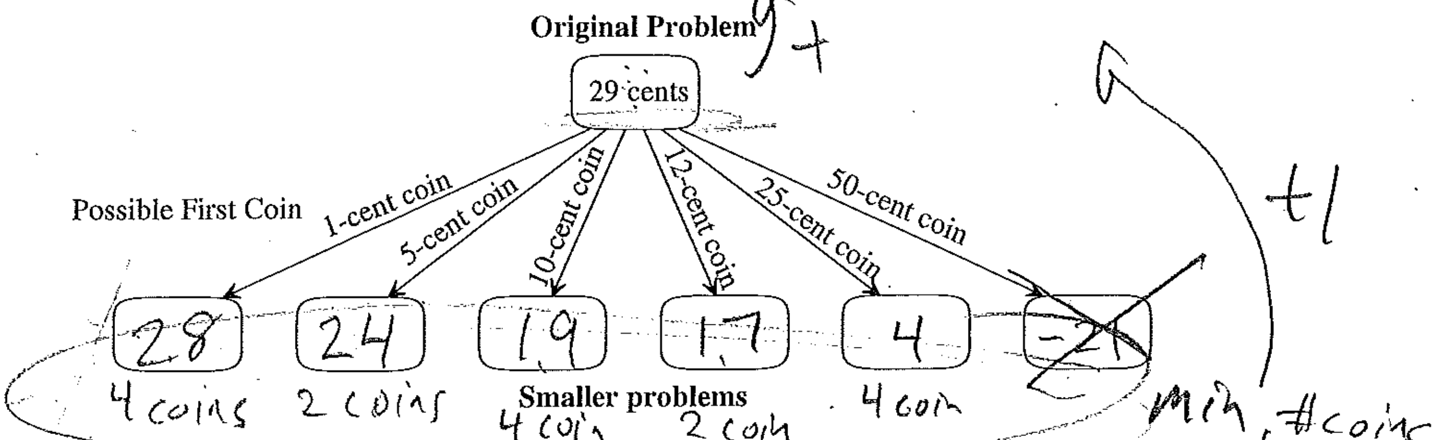
After we give back the first coin, which smaller amounts of change do we have?

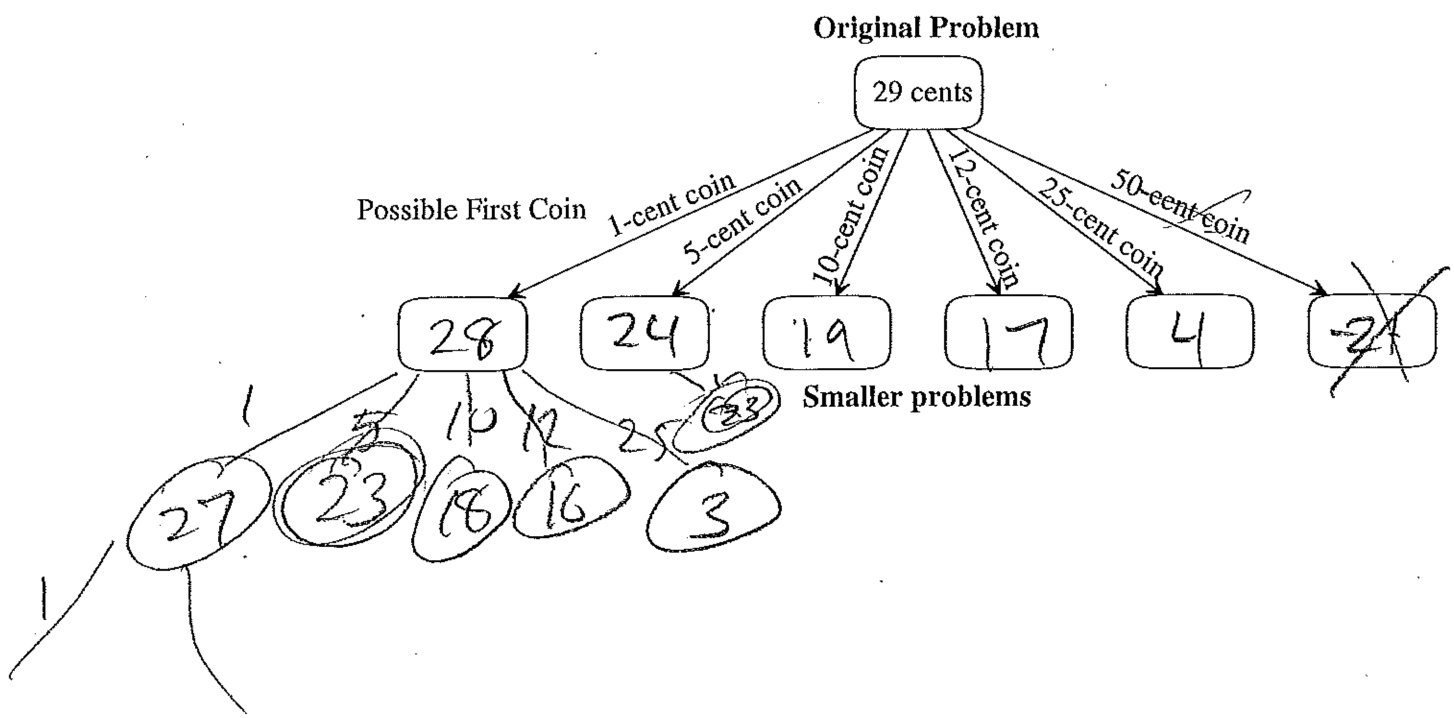


4. If we knew the fewest number of coins needed for each possible smaller problem, then how could determine the fewest number of coins needed for the original problem?

5. Complete a recursive relationship for the fewest number of coins.

$$\overset{29}{\text{FewestCoins(change)}} = \begin{cases} \min\limits_{\text{coin} \in \text{CoinSet and coin} \le \text{change}} (\text{FewestCoins}(\text{change} - \text{coin})) + 1 & \text{if change} \notin \text{CoinSet} \\ 1 & \text{if change} \in \text{CoinSet} \end{cases}$$

6. Complete a couple levels of the recursion tree for 29-cents change using the set of coins {1, 5, 10, 12, 25, 50}.



Data Structures Lecture 13 Name: Mark F.

1. The textbook solves the coin-change problem with the following code (note the "set-builder-like" notation): {c | c ∈ coinValueList and c ≤ 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"? The 1st version rebuilds the list of "valid" coins on each of 67 million calls. The 2nd version use the original list with an if-statement check.

b) Why does it still take a long time? As with recursive fibonacci we solve the smaller change problems from scratch each time we encounter them. Thus, the recursion tree grows exponentially.

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? currentChange amount remaining and coins previously given back to get to this current tree node.

b) When would a "child" tree node NOT be promising?

(1) change amount goes negative (29¢ change + 50¢ coin → -21)
(2) if tree node cannot do better than a previously found solution
(see attached page)

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.

With [1,5,10,25] the 1st solution found has 29-coins (i.e. all pennies) but with [25,10,5,1] the 1st solution found is a 5-coin solution that's more helpful in pruning the tree.

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?

Recalculation of smaller problems many times.

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:

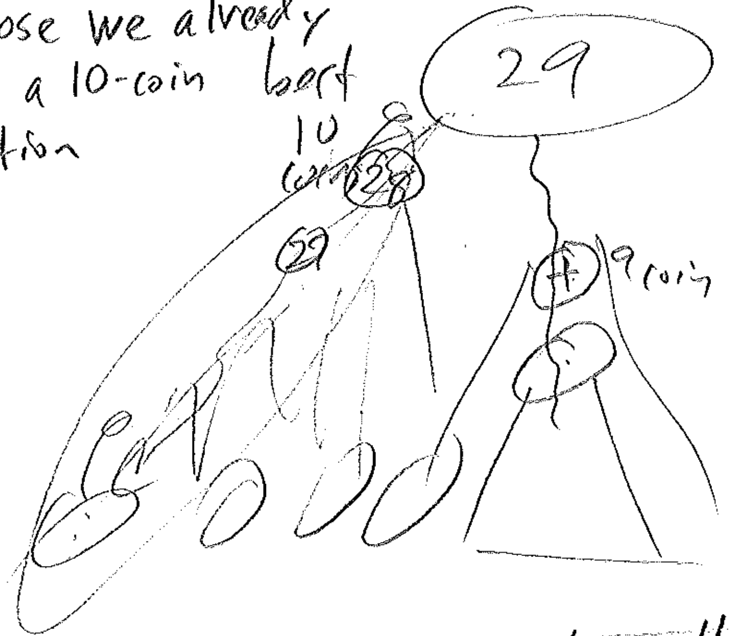
- What is the smallest problem? 0
- Where do we store the answers to the smaller problems?

Change amount → 0 1 2 3 19 24 28 29

fewest coins → 0 2 3

50 10 5 1

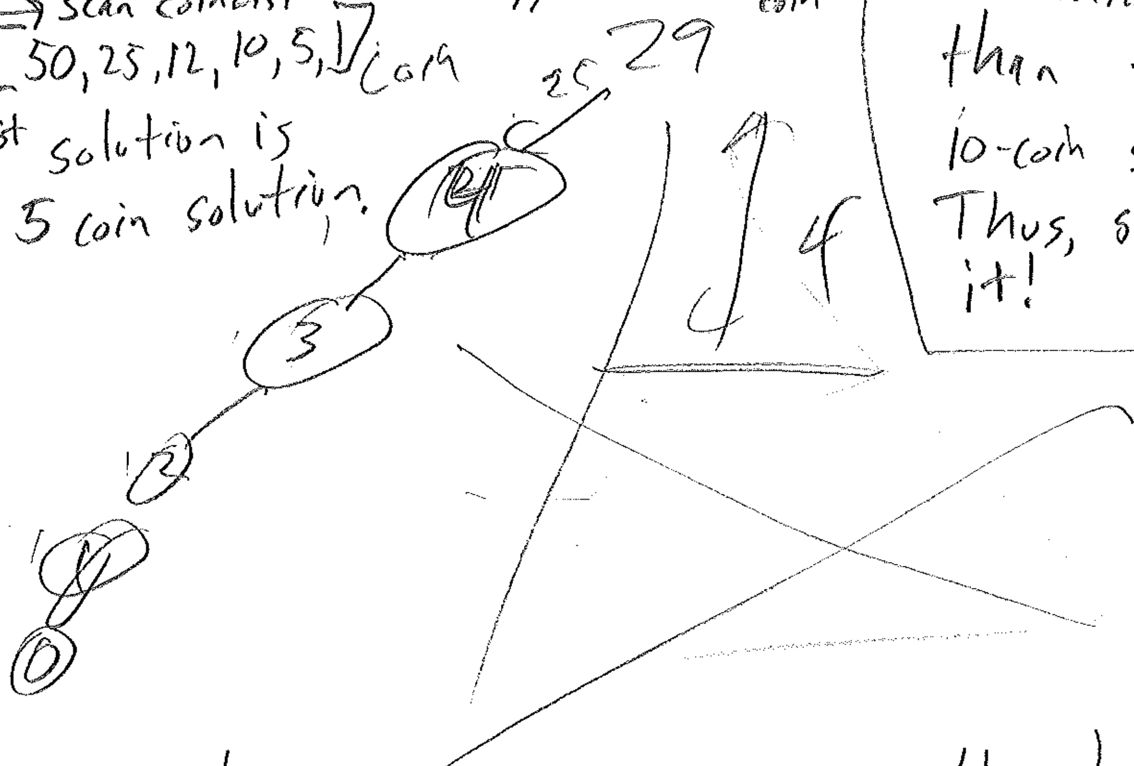
Suppose we already have a 10-coin best solution



If we are following a branch and already have given back 9 coin with more to return (ie., position change amount), then it cannot do better than the best 10-coin solution. Thus, stop following it!

→ scan coinList from biggest to smallest coin
[50, 25, 12, 10, 5, 1] coin

1st solution is a 5 coin solution.



Thus, we can prune all other branches to 4-coins at most.

```
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                            9                        10
            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
```

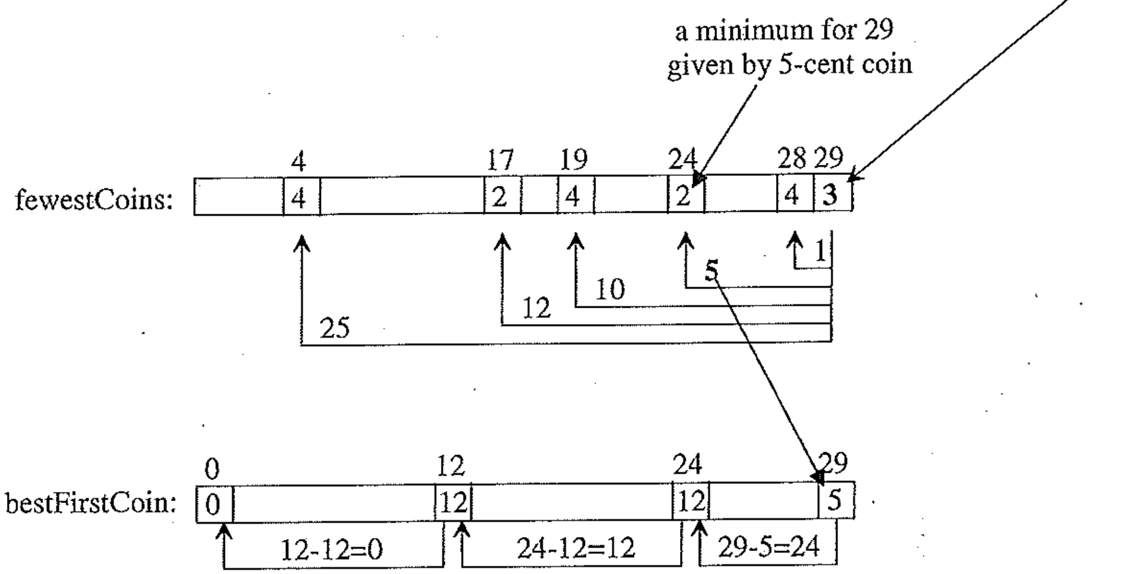
Dynamic Programming Coin-change Algorithm:

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.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
fewestCoins:	0	1	2	3	4	1	2	3	4	5	1	2	1	2	3	2	3

	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
fewestCoins:	2	3	4	2	3	2	3	2	1	2	3	4	3	2	3	3

25 12 10 5 1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
bestFirstCoin:	0	1	1	1	1	5	1	1	1	1	10	1	12	1	1	5	1

	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
bestFirstCoin:	5	1	1	10	1	10	1	12	25	1	1	1	5	5	1	10

12¢ 10¢ 10¢

stop at 0

start here at 32¢

c) What coins are in the solution for 32-cents?

10¢, 10¢, 12¢

Data Structures Lecture 14 Name: Mark F.

1. Consider the following sequential search (linear search) code:

Textbook's Listing 5.1

```
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
```

Faster sequential search code

```
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 of items: target with list item.

b) For the following `aList` value, which `target` value causes `linearSearch` to loop the fewest ("best case") number of times? 10

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

$O(1)$ best case

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

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

$O(n)$

Textbook's Listing 5.2

```
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
```

2 comparison

Faster sequential search code

```
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
```

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?

Stops early if run across list item thats bigger than target item.

2. Consider the following binary search code:

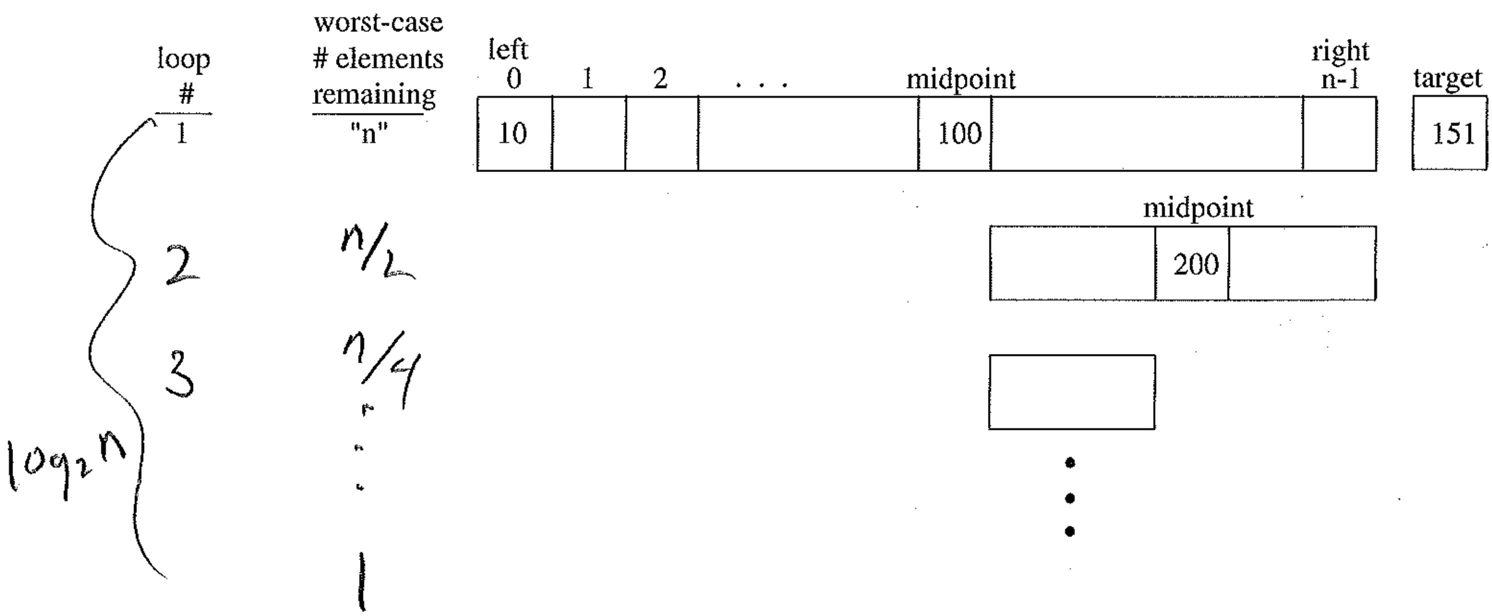
Textbook's Listing 5.3

```
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?

c) What is the best-case big-oh for binary search?

d) What is the average-case (expected) big-oh for binary search?

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

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