Data Structures (CS 1520) Homework #3 Due: March 2 (Saturday) at 11:59 PM

Objective: To gain experience implementing linked data structures by implementing a cursor-based list using doubly-linked nodes.

To start the homework: Download and extract the file hw3.zip from
http://www.cs.uni.edu/~fienup/cs1520s19/homework/

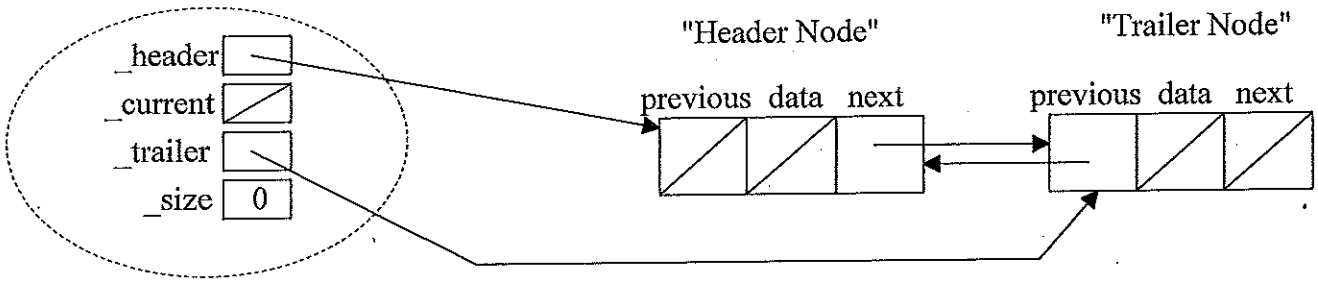
The hw3.zip file contains:

- the Node class (in the node.py module) and the Node2Way class (in the node2way.py module)
- the skeleton CursorBasedList class (in the cursor_based_list.py module) which you will complete
- the cursorBasedListTester.py file that you can use to interactively test your CursorBasedList class.

Part A: Recall that in a **cursor-base list** a *cursor* (indicating the *current item*) can be moved around the list with the cursor being used to identify the region of the list to be manipulated. We will insert and removing items relative to the current item. A *current item* must always be defined as long as the list is not empty.

Cursor-based operations	**Description of operation**
`L.getCurrent()`	Precondition: the list is not empty. Returns the current item without removing it or changing the current position.
`L.hasNext()`	Precondition: the list is not empty. Returns True if the current item has a next item; otherwise return False.
`L.next()`	Precondition: hasNext returns True. Postcondition: The current item has moved right one item
`L.hasPrevious()`	Precondition: the list is not empty. Returns True if the current item has a previous item; otherwise return False.
`L.previous()`	Precondition: hasPrevious returns True. Postcondition: The current item has moved left one item
`L.first()`	Precondition: the list is not empty. Makes the first item the current item.
`L.last()`	Precondition: the list is not empty. Makes the last item the current item.
`L.insertAfter(item)`	Inserts item after the current item, or as the only item if the list is empty. The new item is the current item.
`L.insertBefore(item)`	Inserts item before the current item, or as the only item if the list is empty. The new item is the current item.
`L.replace(newValue)`	Precondition: the list is not empty. Replaces the current item by the newValue
`L.remove()` -str- -- len--	Precondition: the list is not empty. Removes and returns the current item. Making the next item the current item if one exists; otherwise the tail item in the list is the current item unless the list in now empty.

The `cursor_based_list.py` file contains a skeleton `CursorBasedList` class. You MUST uses a doubly-linked list implementation with a *header* node and a *trailer* node. **All "real" list items will be inserted between the header and trailer nodes to reduce the number of "special cases"** (e.g., inserting first item in an empty list, deleting the last item from the list, etc.). An empty doubly-linked list implementation with a *header* node and a *trailer* node looks like:



Use the provided `cursorBasedListTester.py` program to test your list.

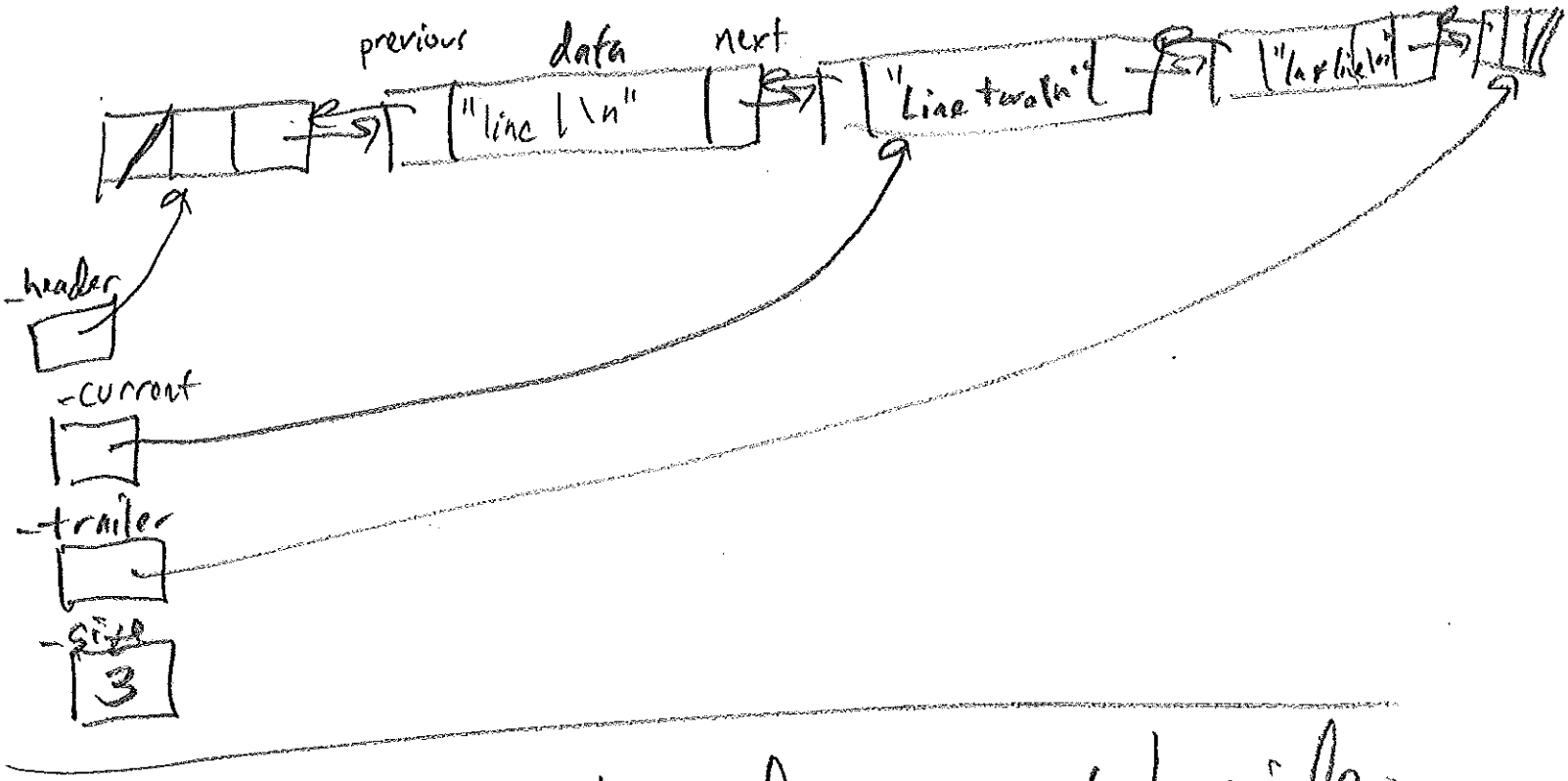
myFile.txt

```
line 1 \n
Line two\n
last line\n
```

Main: ① ask to open or create file

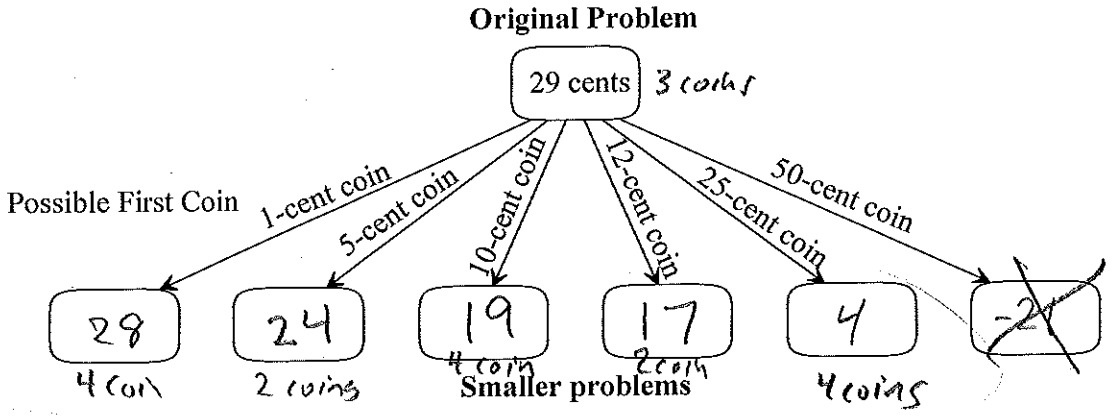
② Main menu loop - Part A tester

③ save list back to file



No class Thursday - watch video for lecture 14

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

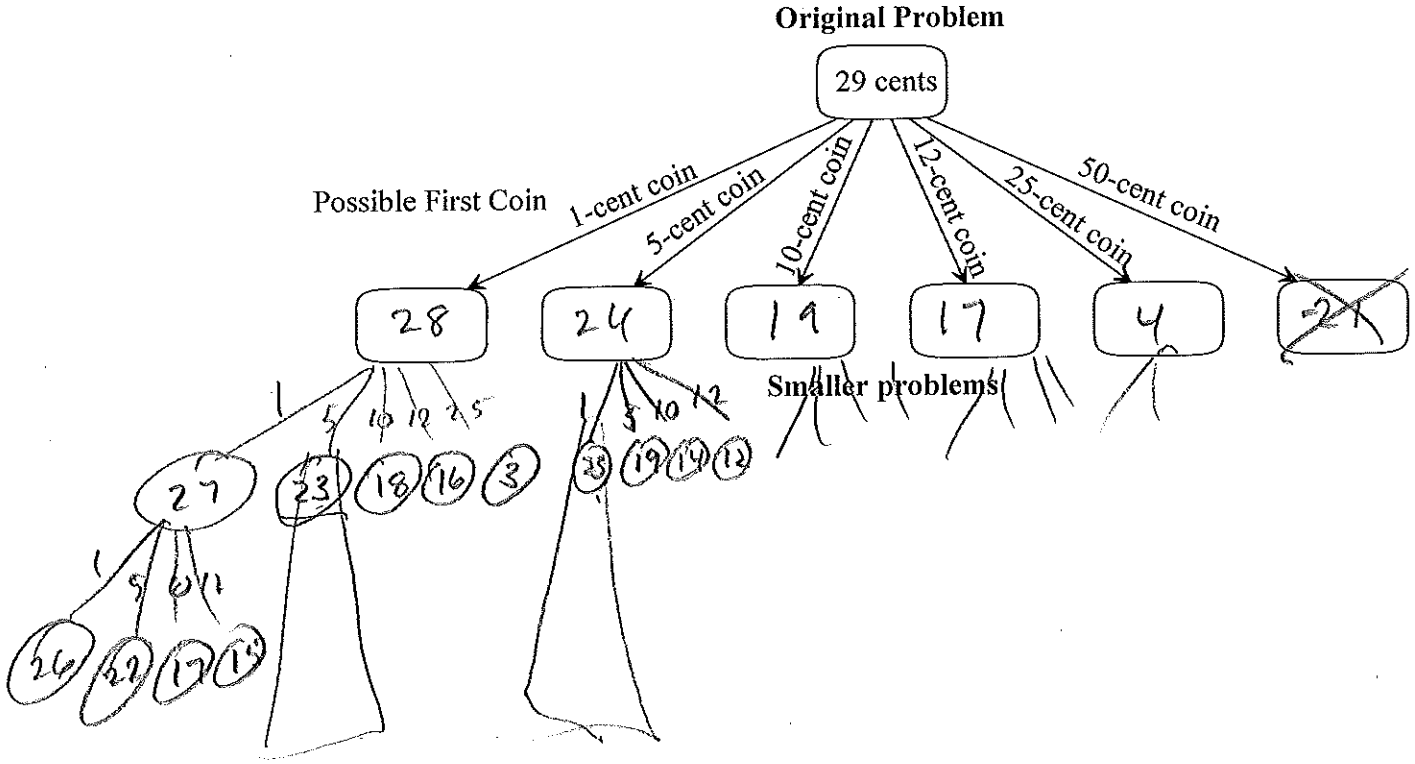


4. If we knew the fewest number of coins needed for each possible smaller problem, then how could determine the fewest number of coins needed for the original problem?

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

$$\text{FewestCoins}(change) = \begin{cases} \min_{coin \in CoinSet \text{ and } coin \le change}(\text{FewestCoins}(change - coin)) + 1 & \text{if } change \notin CoinSet \\ 1 & \text{if } change \in CoinSet \\ 0 & \text{if } change = 0 \end{cases}$$

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



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 first version must create a new list of "valid" coin types, but the second version just uses the coinValueList with an if-statement.

b) Why does it still take a long time? Still does a lot of redundant calculations and has 67,716,925 calls to the function.

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:

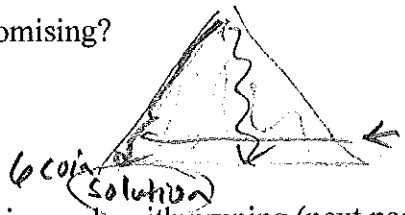
```
Backtrack( recursionTreeNode p ) {
    for each child c of p do                          # each c represents a possible choice
        if promising(c) then                          # c is "promising" if it could lead to a better solution
            if c is a solution that's better than best then   # 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?



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.

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?

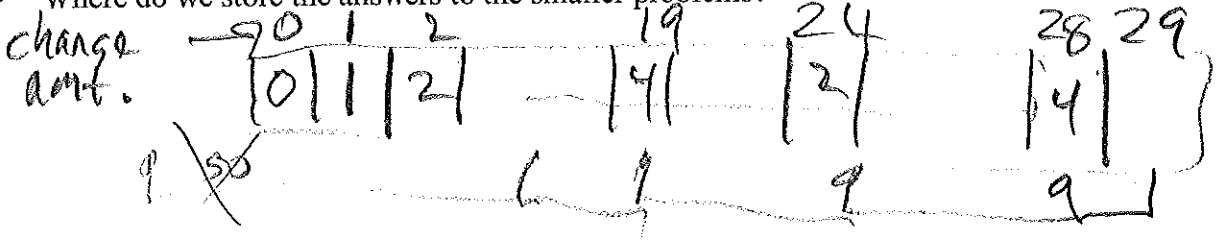
4. As with Fibonacci, the coin-change problem can benefit from dynamic program since it was slow due to solving the same problems over-and-over again. Recall the general idea of dynamic programming:

- 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 change amount 0 coins

- Where do we store the answers to the smaller problems?



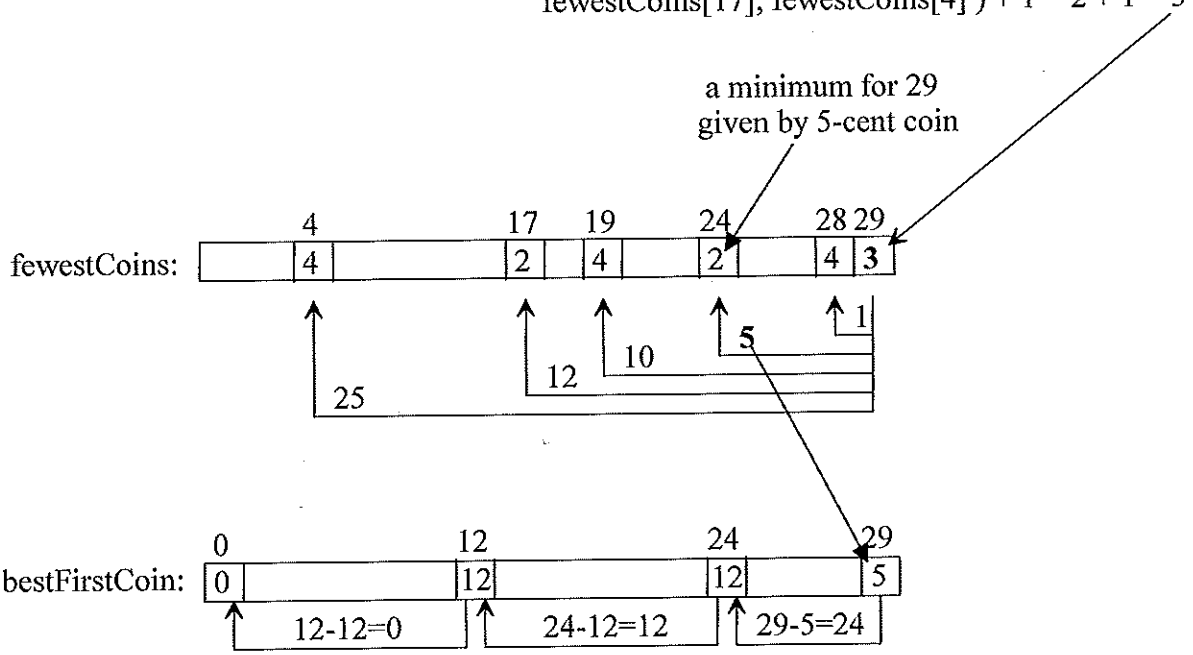
Dynamic Programming Coin-change Algorithm:

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.

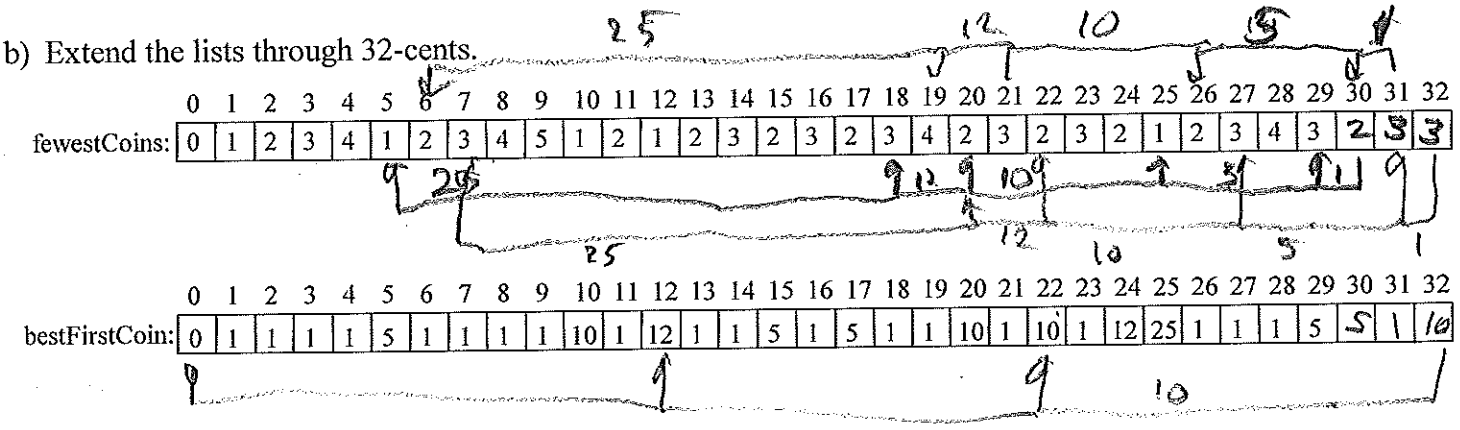
II. If we record the best, first coin to return for each change amount (found in the "minimum" calculation) in an array bestFirstCoin, then we can easily recover the actual coin types to return.

fewestCoins[29] = 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? Compare two items

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

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?

O(n) where n = len(aList)

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	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

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?

code at top of page always checks all items on any unsuccessful search. Bottom version gets to stop "early" if target item > any item in list.

2. Consider the following binary search code:

Textbook's Listing 5.3

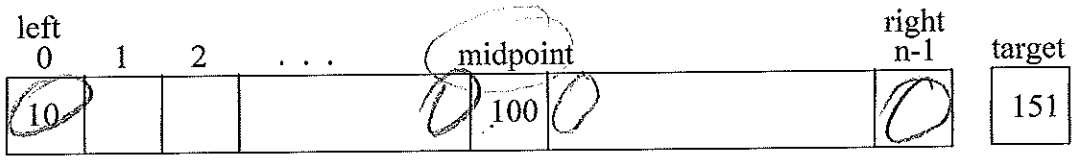
```
def binarySearch(alist, item):
    first = 0
    last = len(alist)-1
    found = False

    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
```

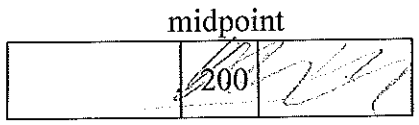
Faster binary search code

```
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?

loop #	worst-case # elements remaining "n"
1	
2	$\frac{n}{2}$
3	$\frac{n}{4}$
4	$\frac{n}{8}$
	1





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

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*?

$\log_2 1000000$ $2^{10} = 1024$ $2^{20} = 1024 \times 1024$ ~20

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

~20