1. The textbook's ordered list ADT uses a singly-linked list implementation. I added the _size, _tail, _current, _previous, and _currentIndex attributes:

OrderedList Object

The search(targetItem) method searches for targetItem in the list. It returns True if targetItem is in the list; otherwise it returns False. Additionally, it has the side-effects of setting _current, _previous, and _currentIndex. The complete search(targetItem) method code for the OrderedList is:

```python
class OrderedList:
    def search(self, targetItem):
        if self._current != None and self._current.getData() == targetItem:
            return True

        self._previous = None
        self._current = self._head
        self._currentIndex = 0
        while self._current != None:
            if self._current.getData() == targetItem:
                return True
            elif self._current.getData() > targetItem:
                return False
            else: # inch-worm down list
                self._previous = self._current
                self._current = self._current.getNext()
                self._currentIndex += 1

        return False
```

a) What's the purpose of the "elif self._current.getData() > targetItem:" check?

Consider the add(item) method with the precondition: item is not in the list.

b) Write the precondition check at the start of the add(item) method.

c) Suppose you are adding the item value of 's'. Update the above picture for this "normal" case, and number the steps in the drawing.

d) What special cases need to be considered for the add method?
2. A recursive function is one that calls itself. Complete the recursive code for the countDown function that is passed a starting value and proceeds to count down to zero and prints "Blast Off!!!".

**Hint**: The countDown function, like most recursive functions, solves a problem by splitting the problem into one or more simpler problems of the same type. For example, countDown (10) prints the first value (i.e., 10) and then solves the simpler problem of counting down from 9. To prevent "infinite recursion", if-statements are used to check for trivial base case(s) of the problem that can be solved without recursion. Here, when we reach a countDown (0) problem we can just print "Blast Off!!!".

```python
""" File: countDown.py """

def main():
    start = eval(input("Enter count down start: "))
    print("\nCount Down:")
    countDown(start)

def countDown(count):

main()
```

**Program Output:**

```
Enter count down start: 10
Count Down:
10
9
8
7
6
5
4
3
2
1
Blast Off!!!
```

a) Trace the function call countDown (5) on paper by drawing the run-time stack and showing the output.

b) What do you think will happen if your call countDown (-1)?

c) Why is there a limit on the depth of recursion?
3. The non-recursive `__str__` method for `OrderedList` object below would return: "(head) a c m (tail)"

```
def __str__(self):
    resultStr = "(head) "
    current = self._head
    while current != None:
        resultStr += str(current.getData()) + " "
        current = current.getNext()
    return resultStr + "(tail)"
```

We can thing of building the string for the list as "a" + (string for the rest of the list)

a) Complete the recursive `strHelper` function in the `__str__` method for our `OrderedList` class.

```
def __str__(self):
    """ Returns a string representation of the list with a space between each item. """
    def strHelper(current):

        # Start of __str__ method execution
        return "(head) " + strHelper(self._head) + "(tail)"
```

4. Some mathematical concepts are defining by recursive definitions. One example is the Fibonacci series:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

After the second number, each number in the series is the sum of the two previous numbers. The Fibonacci series can be defined recursively as:

\[
\begin{align*}
    F_{0} &= 0 \\
    F_{1} &= 1 \\
    F_{N} &= F_{N-1} + F_{N-2} & \text{for } N \geq 2.
\end{align*}
\]

a) Complete the recursive function:

b) Draw the call tree for `fib(5)`.  
c) On my office computer, the call to fib(40) takes 22 seconds, the call to fib(41) takes 35 seconds, and the call to fib(42) takes 56 seconds. How long would you expect fib(43) to take?

d) How long would you guess calculating fib(100) would take on my office computer?

e) Why do you suppose this recursive fib function is so slow?

f) What is the computational complexity? $O(\ )$

g) How might we speed up the calculation of the Fibonacci series?

5. A VERY POWERFUL concept in Computer Science is dynamic programming. Dynamic programming solutions eliminate the redundancy of divide-and-conquer algorithms by calculating the solutions to smaller problems first, storing their answers, and looking up their answers if later needed instead of recalculating them.

We can use a list to store the answers to smaller problems of the Fibonacci sequence. To transform from the recursive view of the problem to the dynamic programming solution you can do the following steps:

1) Store the solution to smallest problems (i.e., the base cases) in a list
2) Loop (no recursion) from the base cases up to the biggest problem of interest. On each iteration of the loop we:

   • solve the next bigger problem by looking up the solution to previously solved smaller problem(s)
   • store the solution to this next bigger problem for later usage so we never have to recalculate it

a) Complete the dynamic programming code:

```python
def fib(n):
    """Dynamic programming solution to find the nth number in the Fibonacci sequence."""
    # List to hold the solutions to the smaller problems
    fibonacci = []

    # Step 1: Store base case solutions
    fibonacci.append(  )
    fibonacci.append(  )

    # Step 2: Loop from base cases to biggest problem of interest
    for position in range( ):
        fibonacci.append(  )

    # return nth number in the Fibonacci sequence
    return
```

Running the above code to calculate fib(100) would only take a fraction of a second.

b) One tradeoff of simple dynamic programming implementations is that they can require more memory since we store solutions to all smaller problems. Often, we can reduce the amount of storage needed if the next larger problem (and all the larger problems) don't really need the solution to the really small problems, but just the larger of the smaller problems. In fibonacci when calculating the next value in the sequence how many of the previous solutions are needed?
Objective: To understand recursion by writing simple recursive solutions.

To start the lab: Download and unzip the lab5.zip file from eLearning.

Part A: Recall: We modified the textbook's ordered list ADT that uses a singly-linked list implementation by adding the _size, _tail, _current, _previous, and _currentIndex attributes:

```
# Non-recursive code we are replacing

def search(self, targetItem):
    if self._current != None and \\  
    self._current.getData() == targetItem:
        return True

    self._previous = None
    self._current = self._head
    self._currentIndex = 0
    while self._current != None:
        if self._current.getData() == targetItem:
            return True
        elif self._current.getData() > targetItem:
            return False
        else:  # inchworm down list
            self._previous = self._current
            self._current = self._current.getNext()
            self._currentIndex += 1
        return False
```

def search(self, targetItem):
    def searchHelper():
        
        """Recursive helper function that moves down the linked list.
        It has no parameters, but uses self._current, self._previous,
        and self._currentIndex.""
        
        # Add code here

```
# Start of search - do not modify below code
if self._current != None and \\  
self._current.getData() == targetItem:
    return True

self._previous = None
self._current = self._head
self._currentIndex = 0
return searchHelper()  # return the result of searchHelper
```

a) What are the base case(s) for the searchHelper that halt the while-loop of the non-recursive search code?

b) What are the recursive case(s) for the searchHelper that replaces the while-loop of the non-recursive search code?

c) Complete the recursive searchHelper function in the search method of our OrderedList class in ordered_linked_list.py. Run the ordered_linked_list.py test code at the bottom of the class, or test it with the listTester.py program.
Part B: Recall that Lecture 7 and Section 6.6 discussed a very “non-intuitive”, but powerful list/array-based approach to implement a priority queue, call a binary heap. The list/array is used to store a complete binary tree (a full tree with any additional leaves as far left as possible) with the items being arranged by heap-order property, i.e., each node is ≤ either of its children. An example of a min heap “viewed” as a complete binary tree would be:

```
    6
   / \
  [2] [3]
 /     /
15    10    20    50
/
114
/
300
```

Python List actually used to store heap items:
```
0 1 2 3 4 5 6 7 8 9 10
| | | | | | | | | | |
| | | | | | | | | | |
```
Recall the General Idea of `insert(newItem)`:
- append `newItem` to the end of the list (easy to do, but violates heap-order property)
- restore the heap-order property by repeatedly swapping the `newItem` with its parent until it percolates up to the correct spot

Recall the General Idea of `delMin()`:
- remember the minimum value so it can be returned later (easy to find - at index 1)
- copy the last item in the list to the root, delete it from the right end, decrement size
- restore the heap-order property by repeatedly swapping this item with its smallest child until it percolates down to the correct spot
- return the minimum value

Originally, we used iteration (i.e., a loop) to percolate up (see `percUp`) and percolate down (see `percDown`) the tree. (textbook code below)

```python

## NON-RECURSIVE CODE WE ARE REPLACING

def percUp(self, i):
    while i // 2 > 0:
        if self.heapList[i] < self.heapList[i // 2]:
            tmp = self.heapList[i // 2]
            self.heapList[i // 2] = self.heapList[i]
            self.heapList[i] = tmp
            i = i // 2

## RECURSIVE CODE

def percDown(self, i):
    while (i * 2) <= self.currentSize:
        mc = self.minChild(i)
        if self.heapList[i] > self.heapList[mc]:
            tmp = self.heapList[i]
            self.heapList[i] = self.heapList[mc]
            self.heapList[mc] = tmp
            i = mc
```

For part B, I want you to complete the recursive `percUpRec` and recursive `percDownRec` methods in `binHeap.py`. Run the `binHeap.py` file which has test code at the bottom to test both methods.

After you have correct code for both parts of the lab, submit a `lab5.zip` containing your code on eLearning. If you do not get done today, then submit it by next week’s lab period.

(If you have extra time, work on previous labs or homeworks!)
1. Consider the coin-change problem: Given a set of coin types and an amount of change to be returned, determine the fewest number of coins for this amount of change.

   a) What "greedy" algorithm would you use to solve this problem with US coin types of \{1, 5, 10, 25, 50\} and a change amount of 29-cents?

   b) Do you get the correct solution if you use this algorithm for coin types of \{1, 5, 10, 12, 25, 50\} and a change amount of 29-cents?

2. One way to solve this problem in general is to use a divide-and-conquer algorithm. Recall the idea of Divide-and-Conquer algorithms.

   Solve a problem by:
   - dividing it into smaller problem(s) of the same kind
   - solving the smaller problem(s) recursively
   - use the solution(s) to the smaller problem(s) to solve the original problem

   a) For the coin-change problem, what determines the size of the problem?

   b) How could we divide the coin-change problem for 29-cents into smaller problems?

   c) If we knew the solution to these smaller problems, how would be able to solve the original problem?
3. After we give back the first coin, which smaller amounts of change do we have?

**Original Problem**

```
29 cents
```

Possible First Coin

```
1-cent coin
5-cent coin
10-cent coin
25-cent coin
50-cent coin
```

**Smaller problems**

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.

```
FewestCoins(change) = \begin{cases} 
\min(\text{FewestCoins}( \text{coin } \in \text{CoinSet and coin } \leq \text{change} \bigg) ) + 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\}.

**Original Problem**

```
29 cents
```

Possible First Coin

```
1-cent coin
5-cent coin
10-cent coin
25-cent coin
50-cent coin
```

**Smaller problems**
Objective: To gain experience utilizing a previously implementing data structure (i.e., cursor-based list) and writing user-friendly and robust programs.

To start the homework you must have completed Homework #3. Copy your hw3 folder and rename it hw4. It should already contain:
- the Node class (in the node.py module) and the Node2Way class (in the node2way.py module)  
- your completed CursorBasedList class (in the cursor_based_list.py module)  
- the cursorBasedListTester.py file that you used to interactively test your CursorBasedList class.

Text-editor Program
Once you have your CursorBasedList class finished (HW #3), you are to write a simple text-editor program that utilizes your CursorBasedList class.

When your text-editor program starts, it should ask for a text-file name (.txt) to edit. If the file name exists, it should load the file into an initially empty CursorBasedList object by reading each line from the file and use the insertAfter method to append the line to the list. Each node in the list will hold a single line of the text file. If the text-file name specified at startup does not exist, an empty CursorBasedList object is created to model editing a new file.

Regardless of whether you loaded a file or just created an empty list, a menu-driven loop very similar to the cursorBasedListTester.py program should allow you to edit the file’s content by modifying the list. You should NOT need to modify your CursorBasedList class. Only create a CursorBasedList object and use its methods. Make sure that your editor does not violate any preconditions of the CursorBasedList methods, so your editor is robust, i.e., does not crash when editing.

When done editing, the lines of data contained in the nodes of the CursorBasedList are written back to the text file.

Your text-editor program should present a menu of options that allows the user to:
- navigate and display the first line, i.e., the first line should be the current line
- navigate and display the last line, i.e., the last line should be the current line
- navigate and display the next line, i.e., the next line should become the current line. If there is no next line, tell the user and don’t change the current line
- navigate and display the previous line. Similarly, if there is no previous line, tell the user and don’t change the current line.
- insert a new line before the current line
- insert a new line after the current line
- delete the current line and have the line following become the current line. If there is no following line, the current line should be the last line.
- replace the current line with a new line
- exit and save the current list back to a text file

Warning: When you load a text file into your list nodes, you can leave the ‘n’ characters on the end of each line of text. However, remember to add a ‘n’ character to end of inserted lines or replacement lines.

Implement AND fully test your text-editor program. Part of your grade will be determined by how robust your text-editor runs (i.e., does not crash) and how user-friendly/intuitive your program is to use. You are required to submit a brief (less than a page) User's manual on how to use your text-editor.

For extra credit, your program may provide do one or more of the following additional text-editor functionality:
- Find word and Find next occurrence
- Replace a specified word/string on the current line by another word/string
- Copy and Paste a line, etc.

Be sure to include these additional features in your User's manual.
Data Structures (CS 1520)  Homework #4  Due: Oct. 10 (Saturday) at 11:59 PM

On eLearning (Course Content | Unit #1 | Homework #3 subfolder), submit a single .zip file, hw3.zip containing the following:

- the Node class (in the node.py module) and the Node2Way class (in the node2way.py module)
- the completed CursorBasedList class (in the cursor_based_list.py module)
- text_editor.py - your text-editor program
- user_manual.txt - a text-file is fine, but you can submit any format I can open (.docx, .rtf, .pdf)

Note: No design document needed for this homework.

(If you miss the deadline, you can still submit it without a late penalty. However, there will be a homework 5, etc. and you don’t want to get too far behind!)
1. The textbook solves the coin-change problem with the following code (note the “set-builder-like” notation):
   \[
   \{ c | c \in \text{coinValueList} \text{ and } c \leq \text{change} \}
   \]

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 [c for c in coinValueList if c <= change]:
            numCoins = 1 + recMC(change - i, coinValueList)
            if numCoins < minCoins:
                minCoins = numCoins
            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

a) Why is the second version so much "faster"?

b) Why does it still take a long time?

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 for
} // end Backtrack
```

# each c represents a possible choice
# c is "promising" if it could lead to a better solution
# check if this is the best solution found so far
# remember the best solution
# follow a branch down the tree

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" eventthough 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.

<table>
<thead>
<tr>
<th>Change Amount: 63</th>
<th>Coin types: [1, 5, 10, 25]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run-time: 0.036 seconds</td>
<td></td>
</tr>
<tr>
<td>Fewest number of coins 6</td>
<td></td>
</tr>
<tr>
<td>The number of each type of coins is:</td>
<td></td>
</tr>
<tr>
<td>number of 1-cent coins is 3</td>
<td></td>
</tr>
<tr>
<td>number of 5-cent coins is 1</td>
<td></td>
</tr>
<tr>
<td>number of 10-cent coins is 1</td>
<td></td>
</tr>
<tr>
<td>number of 25-cent coins is 2</td>
<td></td>
</tr>
<tr>
<td>Number of Backtracking Nodes: 4831</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Change Amount: 63</th>
<th>Coin types: [25, 10, 5, 1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run-time: 0.003 seconds</td>
<td></td>
</tr>
<tr>
<td>Fewest number of coins 6</td>
<td></td>
</tr>
<tr>
<td>The number of each type of coins is:</td>
<td></td>
</tr>
<tr>
<td>number of 25-cent coins is 2</td>
<td></td>
</tr>
<tr>
<td>number of 10-cent coins is 1</td>
<td></td>
</tr>
<tr>
<td>number of 5-cent coins is 0</td>
<td></td>
</tr>
<tr>
<td>number of 1-cent coins is 3</td>
<td></td>
</tr>
<tr>
<td>Number of Backtracking Nodes: 310</td>
<td></td>
</tr>
</tbody>
</table>

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:

<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?

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?
Data Structures

Lecture 13

backtrackingNodes = 0  # profiling variable to track number of state-space tree nodes

def solveCoinChange(changeAmt, coinTypes):
    def backtrack(changeAmt, numberOfCoinTypes, 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 = [1] + numberOfEachCoinType
                        bestNumberOfEachCoinType[index] += 1
                        solutionFound = True
                else:  # call child with updated state information
                    smallerChangeAmtNumberOfEachCoinType = [1] + 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.

\[
fewestCoins[29] = \min\{fewestCoins[28], fewestCoins[24], fewestCoins[19], fewestCoins[17], fewestCoins[4]\} + 1 = 2 + 1 = 3
\]

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

b) Extend the lists through 32-cents.

```
fewestCoins: [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32]
bestFirstCoin: [0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```

c) What coins are in the solution for 32-cents?
Objective: To gain experience with a simple, recursive, divide-and-conquer implementation and how to develop faster solutions using an iterative dynamic programming implementation.

To start the lab: Download and unzip the lab6.zip file from eLearning.

Part A: We've seen several recursive “divide-and-conquer” algorithms: fibonacci and coin-change problem. The general idea of divide-and-conquer algorithms is:
- dividing the original problem into small problem(s) (e.g., fib(n-1) and fib(n-2))
- solving the smaller problem(s) recursively
- combine the solution(s) to smaller problem(s) to solve the original problem (e.g., return fib(n-1) + fib(n-2))

Mathematics has several simple recursively defined functions. For example, the factorial function can be recursively defined as:

\[ n! = n \times (n-1)! \quad \text{for} \quad n \geq 1, \quad \text{and} \quad 0! = 1 \]

Implement a recursive factorial \( n \) function using this recursive definition and test it with several small examples (e.g., \( 3! = 3 \times 2! = 3 \times 2 \times 1! = 3 \times 2 \times 1 \times 0! = 3 \times 2 \times 1 = 6 \), and \( 5! = 5 \times 4 \times 3 \times 2 \times 1 = 120 \)).

In Discrete Structures (CS 1800) you used (or will use) the binomial coefficient formula:

\[ C(n,k) = \frac{n!}{k!(n-k)!} \]

(2)

to calculate the number of combinations of “n choose k,” i.e., the number of ways to choose \( k \) objects from \( n \) objects. For example, when calculating the number of unique 5-card hands from a standard 52-card deck (e.g., \( C(52, 5) \)) we need to calculate \( 52! / 5! \times 47! = 2,598,960 \).

Implement the function \( C(n, k) \) using formula (2) above and your recursive factorial function.

Part B: One problem with using formula (2) in most languages is that \( n! \) grows very fast and overflows the integer representation before you can do the division to bring the value back to a value that can be represented. (NOTE: Python does not suffer from this problem, but lets pretend that it does.)

For example, when calculating \( C(52, 5) \) we need to calculate \( 52! / 5! \times 47! \). However, the value of \( 52! = 80,658,175,170,943,878,571,660,636,856,403,766,975,289,505,440,883,277,824,000,000,000,000 \) is much, much bigger than can fit into a 64-bit integer representation. Fortunately, another way to view \( C(52, 5) \) is recursively by splitting the problem into two smaller problems by focusing on:
- the hands containing a specific card, say the ace of clubs, and
- the hands that do not contain the ace of clubs.

For those hands that do contain the ace of clubs, we need to choose 4 more cards from the remaining 51 cards, i.e., \( C(51, 4) \). For those hands that do not contain the ace of clubs, we need to choose 5 cards from the remaining 51 cards, i.e., \( C(51, 5) \). Therefore, \( C(52, 5) = C(51, 4) + C(51, 5) \).

In general, (NOTE: When implementing your recursive code, be sure to use DC for the recursive calls)

\[ C(n,k) = C(n-1,k-1) + C(n-1,k) \quad \text{for} \quad 1 \leq k \leq (n-1), \quad \text{and} \]
\[ C(n,k) = \begin{cases} 1 & \text{for} \quad k = 0 \text{ or } k = n \end{cases} \]

(3)

Implement the recursive “divide-and-conquer” binomial coefficient function using equation (3). Call your function \( DC(\text{\textit{n}, k}) \) for “divide-and-conquer”. Notice the difference in run-time between calculating the binomial coefficient using \( C(24, 12) \) vs. \( DC(24, 12) \), \( C(26, 13) \) vs. \( DC(26, 13) \), and \( C(28, 14) \) vs. \( DC(28, 14) \).

Part C: Much of the slowness of your “divide-and-conquer” binomial coefficient function, \( DC(n, k) \), is due to redundant calculations performed due to the recursive calls. For example, the recursive calls associated with \( DC(5, 3) = 10 \) would be:
Pascal's triangle (named for the 17th-century French mathematician Blaise Pascal, and for whom the programming language Pascal was also named) is a "dynamic programming" approach to calculating binomial coefficients. Recall that dynamic programming solutions eliminate the redundancy of recursive divide-and-conquer algorithms by calculating the solutions to smaller problems first, storing their answers, and looking up their answers if later needed instead of recalculating it. Abstractly, Pascal's triangle relates to the binomial coefficient as in:

\[
\begin{array}{ccccccc}
C(0,0) & C(1,0) & C(1,1) & C(2,0) & C(2,1) & C(2,2) & C(3,0) \\
0 & 1 & 1 & 1 & 1 & 1 & 1 \\
& 2 & 3 & 6 & 4 & 1 & 0 \\
& & 5 & 10 & 10 & 5 & 1 \\
\end{array}
\]

For Part C, your job is to implement the "dynamic programming" binomial coefficient function using a Python list of lists to store Pascal's triangle (e.g., pascalTriangle[row #][col #]) and loops (no recursion needed). Call your function DP(n, k) for "dynamic programming". Hints for Part C:

- Review the dynamic programming fibonacci example from Lecture 9. File lab6/fibonacci.py contains the recursive divide-and-conquer, and two dynamic programming versions of fibonacci. The first dynamic programming version, fib_DP, stores the answers to all of the smaller problems. The second dynamic programming version, fib_DP2, reduces the amount of memory used by only storing the answers for the previous two smaller problems.

Notice the difference in run-time between calculating the binomial coefficient using DC(24, 12) vs. DP(24, 12), DC(26, 13) vs. DP(26, 13), and DC(28, 14) vs. DP(28, 14).

**Part D: EXTRA CREDIT** - Your function DP_Extra_Credit(n, k) should use the idea of fib_DP2 to avoid storing all the answers to the smaller problems. Notice that the calculation of the next row in the picture above only needs the previous row and none of the older rows.

After you have correct code for all parts of the lab, submit a lab6.zip containing your code on eLearning. If you do not get done today, then submit it by next week's lab period.
1. Consider the following sequential search (linear search) code:

**Textbook's Listing 5.1**

```
def sequentialSearch(alist, item):
    """Sequential search of unordered 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?

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

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<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>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>97</td>
</tr>
<tr>
<td>aList:</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>
</tr>
<tr>
<td></td>
<td>93</td>
<td>97</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

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;Returns the position of the target item if found, or -1 otherwise.&quot;</td>
</tr>
<tr>
<td>last = len(alist)-1</td>
<td>left = 0</td>
</tr>
<tr>
<td>found = False</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>#</th>
<th>worst-case # elements remaining</th>
<th>left</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>midpoint</th>
<th>n-1</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;n&quot;</td>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100</td>
<td></td>
<td>151</td>
</tr>
</tbody>
</table>

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 maximum 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?
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?

   \[
   \text{average case } O(\quad) \\
   \text{worst case } O(\quad)
   \]

   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?

   \[
   \text{average case } O(\quad) \\
   \text{worst case } O(\quad)
   \]

   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>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>hash(John Doe) = 6</td>
</tr>
<tr>
<td>Philip East</td>
<td>hash(Philip East) = 3</td>
</tr>
<tr>
<td>Mark Fienup</td>
<td>hash(Mark Fienup) = 5</td>
</tr>
<tr>
<td>Ben Schafer</td>
<td>hash(Ben Schafer) = 8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hash Table Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

   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?

   \[
   \text{average case } O(\quad) \\
   \text{worst case } O(\quad)
   \]

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 specific 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)
5. Consider the following examples using open-address approach with a simple rehashing strategy of linear probing where the hash table is scanned circularly from the home address until an empty hash table address is found.

<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>Andrew Berns</td>
<td>hash(Andrew Berns) = 3</td>
<td>4</td>
</tr>
<tr>
<td>Sarah Diesburg</td>
<td>hash(Sarah Diesburg) = 3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Philip East 3-2939</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mark Fienup 3-5918</td>
</tr>
<tr>
<td></td>
<td></td>
<td>John Doe 3-4567</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ben Schafer 3-2187</td>
</tr>
</tbody>
</table>

a) Assuming open-address with linear probing where would Andrew Berns and then Sarah Diesburg be placed?

Common rehashing strategies include the following.

<table>
<thead>
<tr>
<th>Rehash Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear probing</td>
<td>Check next spot (counting circularly) for the first available slot, i.e., (home address + (rehash attempt #)) % (hash table size)</td>
</tr>
<tr>
<td>quadratic probing</td>
<td>Check the square of the attempt-number away for an available slot, i.e., (home address + ((rehash attempt #)² + (rehash attempt #)/2) % (hash table size), where the hash table size is a power of 2 and the offset hash returns an odd value between 1 and the hash table size</td>
</tr>
<tr>
<td>double hashing</td>
<td>Use the target key to determine an offset amount to be used each attempt, i.e., (home address + (rehash attempt #) * offset) % (hash table size), where the hash table size is a power of 2 and the offset hash returns an odd value between 1 and the hash table size</td>
</tr>
</tbody>
</table>

b) Assume quadratic probing, insert "Andrew Berns" and "Sarah Diesburg" into the hash table.

<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>Andrew Berns</td>
<td>hash(Andrew Berns) = 3</td>
<td>4</td>
</tr>
<tr>
<td>Sarah Diesburg</td>
<td>hash(Sarah Diesburg) = 3</td>
<td>5</td>
</tr>
<tr>
<td>(3-2740)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3-7395)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|                   |                                 | 5                |
|                   |                                 |                  |
|                   |                                 | Ben Schafer 3-2187|
|                   |                                 | 6                |
|                   |                                 | 7                |
|                   |                                 | Philip East 3-2939|
|                   |                                 | Mark Fienup 3-5918|
|                   |                                 | John Doe 3-4567  |
c) Assume double hashing, insert “Andrew Berns” and “Sarah Diesburg” into the hash table.

<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></td>
</tr>
<tr>
<td>Philip East</td>
<td>hash(Philip East) = 3</td>
<td></td>
</tr>
<tr>
<td>Mark Fienup</td>
<td>hash(Mark Fienup) = 5</td>
<td></td>
</tr>
<tr>
<td>Ben Schafer</td>
<td>hash(Ben Schafer) = 0</td>
<td></td>
</tr>
<tr>
<td>Andrew Berns</td>
<td>hash(Andrew Berns) = 3</td>
<td></td>
</tr>
<tr>
<td>(3-2740)</td>
<td>rehash_offset(Andrew Berns) = 1</td>
<td></td>
</tr>
<tr>
<td>Sarah Diesburg</td>
<td>hash(Sarah Diesburg) = 3</td>
<td></td>
</tr>
<tr>
<td>(3-7395)</td>
<td>rehash_offset(Sarah Diesburg) = 3</td>
<td></td>
</tr>
</tbody>
</table>

| Andrew Berns         | hash(Andrew Berns) = 3     |                  |
| (3-2740)             | rehash_offset(Andrew Berns) = 1 |      |
| Sarah Diesburg       | hash(Sarah Diesburg) = 3   |                  |
| (3-7395)             | rehash_offset(Sarah Diesburg) = 3 |      |

d) For the above double-hashing example, what would be the sequence of hashing and rehashing addresses tried for Sarah Diesburg if the table was full? For the above example, \((\text{home address} + (\text{rehash attempt #} \times \text{offset}) \mod \text{hash table size})\) would be: \((3 + (\text{rehash attempt #}) \times 3) \mod 8\)

<table>
<thead>
<tr>
<th>Rehash Attempt #</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>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

e) Indicate whether each of the following rehashing strategies suffer from primary or secondary clustering.
- **primary clustering** - keys mapped to a home address follow the same rehash pattern
- **secondary clustering** - rehash patterns from initially different home addresses merge together

<table>
<thead>
<tr>
<th>Rehash Strategy</th>
<th>Description</th>
<th>Suffers from:</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear probing</td>
<td>Check next spot (counting circularly) for the first available slot, i.e., ((\text{home address} + (\text{rehash attempt #})) \mod \text{hash table size}))</td>
<td>primary clustering</td>
</tr>
<tr>
<td>quadratic probing</td>
<td>Check a square of the attempt-number away for an available slot, i.e., ((\text{home address} + ((\text{rehash attempt #})^2 + (\text{rehash attempt #})/2)) \mod \text{hash table size}), where the hash table size is a power of 2)</td>
<td>primary clustering</td>
</tr>
<tr>
<td>double hashing</td>
<td>Use the target key to determine an offset amount to be used each attempt, i.e., ((\text{home address} + (\text{rehash attempt #}) \times \text{offset}) \mod \text{hash table size}), where the hash table size is a power of 2 and the offset hash returns an odd value between 1 and the hash table size)</td>
<td>primary clustering</td>
</tr>
</tbody>
</table>

6. Let \(\lambda\) be the load factor (\# item/hashing table size). The average probes with linear probing for insertion or unsuccessful search is: \(\left(\frac{1}{2}\right)\left(1 + \left(\frac{1}{(1-\lambda)^2}\right)\right)\). The average for successful search is: \(\left(\frac{1}{2}\right)\left(1 + \left(\frac{1}{(1-\lambda)}\right)\right)\).

a) Why is an unsuccessful search worse than a successful search?
The average probes with quadratic probing for insertion or unsuccessful search is:
\[
\left(\frac{1}{1-\lambda}\right) - \lambda - \log_e(1 - \lambda)
\]

The average probes with quadratic probing for successful search is:
\[
1 - \left(\frac{1}{2}\right) - \log_e(1 - \lambda)
\]

Consider the following table containing the average number probes for various load factors:

<table>
<thead>
<tr>
<th>Probing Type</th>
<th>Search outcome</th>
<th>0.25</th>
<th>0.5</th>
<th>0.67</th>
<th>0.8</th>
<th>0.99</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Probing</td>
<td>unsuccessful</td>
<td>1.39</td>
<td>2.50</td>
<td>5.09</td>
<td>13.00</td>
<td>5000.50</td>
</tr>
<tr>
<td></td>
<td>successful</td>
<td>1.17</td>
<td>1.50</td>
<td>2.02</td>
<td>3.00</td>
<td>50.50</td>
</tr>
<tr>
<td>Quadratic Probing</td>
<td>unsuccessful</td>
<td>1.37</td>
<td>2.19</td>
<td>3.47</td>
<td>5.81</td>
<td>103.62</td>
</tr>
<tr>
<td></td>
<td>successful</td>
<td>1.16</td>
<td>1.44</td>
<td>1.77</td>
<td>2.21</td>
<td>5.11</td>
</tr>
</tbody>
</table>

b) Why do you suppose the "general rule of thumb" in hashing tries to keep the load factor between 0.5 and 0.67?

7. Allowing deletions from an open-address hash table complicates the implementation. Assuming linear probing we might have the following

<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>Andrew Berns</td>
<td>hash(Andrew Berns) = 3</td>
<td>4</td>
</tr>
<tr>
<td>Sarah Diesburg</td>
<td>hash(Sarah Diesburg) = 3</td>
<td>5</td>
</tr>
</tbody>
</table>

Philip East 3-2939
Andrew Berns 3-2740
Mark Fienup 3-5918
John Doe 3-4567
Sarah Diesburg 3-7395
Ben Schafer 3-2187

a) If "Mark Fienup" is deleted, how will we find Sarah Diesburg?

b) How might we fix this problem?
1. The Map/Dictionary abstract data type (ADT) stores key-value pairs. The key is used to look up the data value.

<table>
<thead>
<tr>
<th>Method call</th>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d = ListDict()</td>
<td><strong>init</strong> (self)</td>
<td>Constructs an empty dictionary</td>
</tr>
<tr>
<td>d[&quot;Name&quot;] = &quot;Bob&quot;</td>
<td><strong>setitem</strong> (self, key, value)</td>
<td>Inserts a key-value entry if key does not exist or replaces the old value with value if key exists.</td>
</tr>
<tr>
<td>temp = d[&quot;Name&quot;]</td>
<td><strong>getitem</strong> (self, key)</td>
<td>Given a key return it value or None if key is not in the dictionary</td>
</tr>
<tr>
<td>del d[&quot;Name&quot;]</td>
<td><strong>delitem</strong> (self, key)</td>
<td>Removes the entry associated with key</td>
</tr>
<tr>
<td>if &quot;Name&quot; in d:</td>
<td><strong>contains</strong> (self, key)</td>
<td>Return True if key is in the dictionary; return False otherwise</td>
</tr>
<tr>
<td>for k in d:</td>
<td><strong>iter</strong> (self)</td>
<td>Iterates over the keys in the dictionary</td>
</tr>
<tr>
<td>len(d)</td>
<td><strong>len</strong> (self)</td>
<td>Returns the number of items in the dictionary</td>
</tr>
<tr>
<td>str(d)</td>
<td><strong>str</strong> (self)</td>
<td>Returns a string representation of the dictionary</td>
</tr>
</tbody>
</table>

```
from entry import Entry
class ListDict(object):
    """Dictionary implemented with a Python list."""
    def __init__(self):
        self._table = []

    def __getitem__(self, key):
        """returns the value associated with key or returns None if key does not exist."""
        entry = Entry(key, None)
        try:
            # NOTE: Python list index method
            # errors on unsuccessful search
            index = self._table.index(entry)
            return self._table[index].getValue()
        except:
            return None

    def __delitem__(self, key):
        """removes the entry associated with key."""
        entry = Entry(key, None)
        try:
            # NOTE: Python list index method
            # errors on unsuccessful search
            index = self._table.index(entry)
            self._table.pop(index)
        except:
            return

    def __str__(self):
        """Returns string repr. of the dictionary""
        resultStr = "[
        for item in self._table:
            resultStr = resultStr + " " + str(item)
        return resultStr + "]"

    def __iter__(self):
        """Iterates over keys of the dictionary""
        for item in self._table:
            yield item.getKey()
```

```
class Entry(object):
    """A key/value pair.""

    def __init__(self, key, value):
        self._key = key
        self._value = value

    def getKey(self):
        return self._key

    def getValue(self):
        return self._value

    def setValue(self, newvalue):
        self._value = newvalue

    def __eq__(self, other):
        if not isinstance(other, Entry):
            return False
        return self._key == other._key

    def __str__(self):
        return str(self._key) + ";" + str(self._value)
```

a) Complete the code for the __contains__ method.

    def __contains__(self, key):

b) Complete the code for the __setitem__ method.

    def __setitem__(self, key, value):
2. Dictionary implementation using hashing with chaining -- an UnorderedList object at each slot in the hash table.

```python
from entry import Entry
from unordered_linked_list import UnorderedList

class ChainingDict(object):
    """Dictionary implemented using hashing with chaining."""
    def __init__(self, capacity = 8):
        self._capacity = capacity
        self._table = []
        for index in range(self._capacity):
            self._table.append(UnorderedList())
        self._size = 0
        self._index = None
    def _contains_(self, key):
        """Returns True if key is in the dictionary or False otherwise."""
        if self._index:
            return self._table[self._index].search(entry)
        entry = Entry(key, None)
        return entry.getValue()
    def _getitem_(self, key):
        """Returns the value associated with key or returns None if key does not exist."""
        if key in self:
            entry = Entry(key, None)
            entry = self._table[self._index].remove(entry)
            self._table[self._index].add(entry)
            return entry.getValue()
        else:
            return None
    def _delitem_(self, key):
        """Removes the entry associated with key."""
        if key in self:
            entry = Entry(key, None)
            entry = self._table[self._index].remove(entry)
            self._size -= 1
    def _setitem_(self, key, value):
        """Inserts an entry with key/value if key does not exist or replaces the existing value with value if key exists.""
        entry = Entry(key, value)
        if key in self:
            entry = self._table[self._index].remove(entry)
            entry.setValue(value)
        else:
            self._size += 1
            self._table[self._index].add(entry)
    def __len__(self):
        return self._size
    def __str__(self):
        result = "HashDict: capacity = " + \
                 str(self._capacity) + ", load factor = " + \
                 str(self._size / self._capacity)
        for i in range(self._capacity):
            result += "\nRow " + str(i) + "::" + str(self._table[i])
        return result
    def __iter__(self):
        """Iterates over the keys of the dictionary""
```

ChainingDict Object

<table>
<thead>
<tr>
<th>_size</th>
<th>13</th>
<th>_capacity</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>_index</td>
<td>4</td>
</tr>
</tbody>
</table>

Python list of UnorderedList objects containing Entries

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a) In _getitem_, why is the entry = Entry(key, None) object created?

b) In _getitem_, where does self._index receive its value?

c) What single modification was needed to the UnorderedList's remove method?

d) Complete the _iter_ method.
Objective: To experiment with searching and get a feel for the performance of hashing.

To start the lab: Download and unzip the lab7.zip file from eLearning.

**Part A:** Comparison of searching techniques.

a) Open and run the `timeLinearSearch.py` program that times the LinearSearch algorithm imported from `LinearSearch.py`. Observe that it creates a list, `evenList`, that holds 10,000 sorted, even values (e.g., `evenList = [0, 2, 4, 6, 8, ..., 19996, 19998]`). It then times the searching for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 so half of the searches are successful and half are unsuccessful. How long does it take to linear search for target values from 0, 1, 2, 3, 4, ..., 19998, 19999?

b) Open and run the `timeBinarySearch.py` program that times the binarySearch algorithm imported from `binarySearch.py`. How long does it take to binary search for target values from 0, 1, 2, 3, 4, ..., 19998, 19999?

c) Open and run the `timeListDictSearch.py` program that times the ListDict dictionary ADT in `list_dictionary.py`. The ListDict implementation uses a single Python list for storing dictionary entries. The `timeListDictSearch.py` program adds the 10,000 even values (`i.e., 0, 2, 4, 6, 8, ..., 19996, 19998`) to a ListDict object, and then times the searching for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 so half of the searches are successful and half are unsuccessful. How long does it take to search for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 in the ListDict?

d) Open and run the `timeListDictSearch.py` program that times the ListDict dictionary ADT in `list_dictionary2.py`. The ListDict implementation uses a single Python list for storing dictionary entries, BUT does not use `try-except` to recover from unsuccessful search of Python list's index method. The `timeListDictSearch.py` program adds the 10,000 even values (`i.e., 0, 2, 4, 6, 8, ..., 19996, 19998`) to a ListDict object, and then times the searching for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 so half of the searches are successful and half are unsuccessful. How long does it take to search for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 in the ListDict?

e) Open and run the `timeChainingDictSearch.py` program that times the ChainingDict dictionary ADT in `chaining_dictionary.py`. The `timeChainingDictSearch.py` program adds the 10,000 even values (`i.e., 0, 2, 4, 6, 8, ..., 19996, 19998`) to a ChainingDict with 16,384 slots (`i.e., load factor of 0.61`), and then times the searching for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 so half of the searches are successful and half are unsuccessful. How long does it take to search for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 in the ChainingDict?

f) Explain the relative performance results of searching using linear search, binary search, a ListDict, and ChainingDict. (Think about their big-oh notations and their constants of proportionality “c”)

g) The Python for loop allows traversal of built-in data structures (strings, lists, tuple, etc) by an `iterator`. To accomplish this with our data structures we need to include an `__iter__`-method (e.g., `ListDict` class from Lecture 15 in `lab7/list_dictionary.py`). In general an `__iter__`-method, must loop down the data structure and yield each item in the data structure. See the end of `UnorderedList` and `ListDict` classes for examples of their `__iter__`-methods. **Complete the `__iter__` code for the ChainingDict** (`lab7/chaining_dictionary.py`) and **OpenAddrHashDict** (`lab7/open_addr_hash_dictionary.py`) classes.
Part B: a) Open and run the `timeOpenAddrHashDictSearch.py` program that times the OpenAddrHashDict dictionary ADT in `open_addr_hash_dictionary.py`. The `timeOpenAddrHashDictSearch.py` program adds the 10,000 even values (i.e., 0, 2, 4, 6, 8, ..., 19996, 19998) to an OpenAddrHashDict with 16,384 (2^14) slots (i.e., load factor of 0.61) using linear probing, and then times the searching for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 so half of the searches are successful and half are unsuccessful. How long does it take to search for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 in the OpenAddrHashDict?

b) Place the even values (i.e., 0, 2, 4, 6, 8, ..., 16382, 16384, 16386, 16388, ..., 19998) in the hash table below. Value 0 is stored at home address 0, value 2 is stored at home address 2, ..., value 16,382 is stored at home address 16,382, but values 16,384 to 19,998 will have collisions. Now, think about the number of probes needed to searching for target values from 0, 1, 2, 3, 4, ..., 19998, 19999. Why does the above timing of searching for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 take so long with a load factor of only 0.61?

```
 0
 1
 2
 3
 4
 5
 6
 7
16,380
16,381
16,382
16,383
```

(c) Experiment with changing the load factor of the HashTable by increasing the hash table size to 32,768 (2**15) for a load factor of 0.31, and 65,536 for a load factor of 0.15. Completing the following table:

<table>
<thead>
<tr>
<th>Linear Probing</th>
<th>Hash Table Size (Load Factor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time with 10,000 items in hash table (sec.)</td>
<td>16,384 or 2**14 (0.61)</td>
</tr>
</tbody>
</table>

(d) In `timeOpenAddrHashDictSearch.py` modify the construction of evenHashTable so it uses quadratic probing instead of linear probing (i.e., `evenHashTable = OpenAddrHashTable(2**14, hash, False)`). Completing the following table:

<table>
<thead>
<tr>
<th>Quadratic Probing</th>
<th>Hash Table Size (Load Factor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time with 10,000 items in hash table (seconds)</td>
<td>16,384 (0.61)</td>
</tr>
</tbody>
</table>

e) Explain why quadratic probing performs better than linear probing.

After you have answers and correct code for all parts of the lab, submit a lab7.zip containing your code on eLearning. If you do not get done today, then submit it by next week's lab period.
Remember to save your lab7 files for later usage on homework assignments!
1. The Dictionary implementation using open-address hashing was the `OpenAddrHashDict` class in lab7.zip.

```python
from entry import Entry
class OpenAddrHashDict(object):
    EMPTY = None  # class variables shared by all objects of the class
    DELETED = True

def __init__(self, capacity = 8, hashFunction = hash,
             linear = True):
    self._table = [OpenAddrHashDict.EMPTY] * capacity
    self._size = 0
    self._hash = hashFunction
    self._homeIndex = -1
    self._actualIndex = -1
    self._linear = linear
    self._probeCount = 0

def __getitem__(self, key):
    """Returns the value associated with key or
    returns None if key does not exist.""
    if key in self:
        return self._table[self._actualIndex].getValue()
    else:
        return None

def __delitem__(self, key):
    """Removes the entry associated with key.""
    if key in self:
        self._table[self._actualIndex] = OpenAddrHashDict.DELETED
        self._size -= 1

def __setitem__(self, key, value):
    """Inserts an entry with key/value if key does not exist or
    replaces the existing value with value if key exists.""
    entry = Entry(key, value)
    if key in self:
        self._table[self._actualIndex] = entry
    else:
        self._table[self._actualIndex] = entry
        self._size += 1

def __contains__(self, key):
    """Return True if key is in the dictionary; return False otherwise"""
    entry = Entry(key, None)
    self._probeCount = 0
    # Get the home index
    self._homeIndex = abs(self._hash(key)) % len(self._table)
    self._actualIndex = self._homeIndex
    # Stop searching when an empty cell is encountered
    while self._actualIndex < len(self._table):
        self._probeCount += 1
        if self._table[self._actualIndex] == OpenAddrHashDict.EMPTY):
            self._actualIndex = self._actualIndex
            return False  # an empty cell is found, so key not found
        elif self._table[self._actualIndex] == entry:
            self._actualIndex = self._actualIndex
            return True
        # Calculate the index and wrap around to first position if necessary
        self._actualIndex = self._actualIndex % len(self._table)
        else:  # Quadratic probing
            index = (self._homeIndex + self._actualIndex ** 2 + self._actualIndex) // 2) % len(self._table)
            return False  # tried all the slots in the hash table and did not find key

def __len__(self):
    return self._size

def __str__(self):
    resultStr = "[
    for item in self._table:
        if not item in (OpenAddrHashDict EMPTY, OpenAddrHashDict.DELETED):
            resultStr = resultStr + " " + str(item)
        return resultStr + "]"

def __iter__(self):
    """Iterates over the keys of the dictionary""
```

a) Complete the `__iter__` method.
2. All simple sorts consist of two nested loops where:
   - the outer loop keeps track of the dividing line between the sorted and unsorted part with the sorted part growing by one in size each iteration of the outer loop.
   - the inner loop's job is to do the work to extend the sorted part's size by one.

Initially, the sorted part is typically empty. The simple sorts differ in how their inner loops perform their job.

Selection sort is an example of a simple sort. Selection sort's inner loop scans the unsorted part of the list to find the maximum item. The maximum item in the unsorted part is then exchanged with the last unsorted item to extend the sorted part by one item.

At the start of the first iteration of the outer loop, initial list is completely unsorted:

<table>
<thead>
<tr>
<th>Unsorted Part</th>
<th>Empty Sorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>myList: 25 35 20 40 90 60 10 50 45</td>
<td></td>
</tr>
</tbody>
</table>

The inner loop scans the unsorted part and determines that the index of the maximum item, maxIndex = 4.

<table>
<thead>
<tr>
<th>Unsorted Part</th>
<th>Sorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>myList: 25 35 20 40 90 60 10 50 45</td>
<td></td>
</tr>
</tbody>
</table>

maxIndex = 4  lastUnsortedIndex = 8

After the inner loop (but still inside the outer loop), the item at maxIndex is exchanged with the item at lastUnsortedIndex. Thus, extending the Sorted Part of the list by one item.

<table>
<thead>
<tr>
<th>Unsorted Part</th>
<th>Sorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>myList: 25 35 20 40 45 60 10 50 90</td>
<td></td>
</tr>
</tbody>
</table>

maxIndex = 4  lastUnsortedIndex = 8

a) Write the code for the outer loop

b) Write the code for the inner loop to scan the unsorted part of the list to determine the index of the maximum item

c) Write the code to exchange the list items at positions maxIndex and lastUnsortedIndex.

d) What is the big-oh notation for selection sort?
3. *Bubble sort* is another example of a simple sort. Bubble sort's inner loop scans the unsorted part of the list comparing adjacent items. If it finds adjacent items out of order, then it exchanges them. This causes the largest item to "bubble" up to the "top" of the unsorted part of the list.

At the start of the first iteration of the outer loop, initial list is completely unsorted:

```
<table>
<thead>
<tr>
<th>Unssorted Part</th>
<th>Empty Sorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>myList: 25 35 20 40 90 60 10 50 45</td>
<td></td>
</tr>
</tbody>
</table>
```

The inner loop scans the unsorted part by comparing adjacent items and exchanging them if out of order.

```
<table>
<thead>
<tr>
<th>Unssorted Part</th>
<th>Sorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>myList: 25 35 20 40 90 60 10 50 45</td>
<td></td>
</tr>
</tbody>
</table>

in order, so don't exchange

<table>
<thead>
<tr>
<th>Unssorted Part</th>
<th>Sorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>myList: 25 20 35 40 90 60 10 50 45</td>
<td></td>
</tr>
</tbody>
</table>

out of order, so exchange

<table>
<thead>
<tr>
<th>Unssorted Part</th>
<th>Sorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>myList: 25 20 35 40 60 90 10 50 45</td>
<td></td>
</tr>
</tbody>
</table>

in order, so don't exchange

<table>
<thead>
<tr>
<th>Unssorted Part</th>
<th>Sorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>myList: 25 20 35 40 60 10 90 50 45</td>
<td></td>
</tr>
</tbody>
</table>

out of order, so exchange

<table>
<thead>
<tr>
<th>Unssorted Part</th>
<th>Sorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>myList: 25 20 35 40 60 10 50 90 45</td>
<td></td>
</tr>
</tbody>
</table>

out of order, so exchange

<table>
<thead>
<tr>
<th>Unssorted Part</th>
<th>Sorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>myList: 25 20 35 40 60 10 50 45 90</td>
<td></td>
</tr>
</tbody>
</table>
```

After the inner loop (but still inside the outer loop), there is nothing to do since the exchanges occurred inside the inner loop.

a) What would be the worst-case big-oh of bubble sort?

b) What would be true if we scanned the unsorted part and didn’t need to do any exchanges?
4. Another simple sort is called insertion sort. Recall that in a simple sort:
- the outer loop keeps track of the dividing line between the sorted and unsorted part with the sorted part growing by one in size each iteration of the outer loop.
- the inner loop's job is to do the work to extend the sorted part's size by one.

After several iterations of insertion sort's outer loop, a list might look like:

<table>
<thead>
<tr>
<th>Sorted Part</th>
<th>Unsorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>10 20 35 40 45 60 25 50 90 • • •</td>
<td></td>
</tr>
</tbody>
</table>

In insertion sort the inner-loop takes the "first unsorted item" (25 at index 6 in the above example) and "inserts" it into the sorted part of the list "at the correct spot." After 25 is inserted into the sorted part, the list would look like:

<table>
<thead>
<tr>
<th>Sorted Part</th>
<th>Unsorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>10 20 25 35 40 45 60 50 90 • • •</td>
<td></td>
</tr>
</tbody>
</table>

Code for insertion is given below:

```python
def insertionSort(myList):
    """Rearranges the items in myList so they are in ascending order""
    for firstUnsortedIndex in range(1,len(myList)):
        itemToInsert = myList[firstUnsortedIndex]
        testIndex = firstUnsortedIndex - 1
        while testIndex >= 0 and myList[testIndex] > itemToInsert:
            myList[testIndex+1] = myList[testIndex]
            testIndex = testIndex - 1
        # Insert the itemToInsert at the correct spot
        myList[testIndex + 1] = itemToInsert
```

a) What is the purpose of the testIndex >= 0 while-loop comparison?

b) What initial arrangement of items causes the is the overall worst-case performance of insertion sort?

c) What is the worst-case $O(\cdot)$ notation for the number of item moves?

d) What is the worst-case $O(\cdot)$ notation for the number of item comparisons?

e) What initial arrangement of items causes the is the overall best-case performance of insertion sort?

f) What is the best-case $O(\cdot)$ notation for insertion sort?
Data Structures (CS 1520) Homework #5A Due: October 24 (Saturday) at 11:59 PM

Objective: Become more proficient at implementing sorting algorithms.

To start the homework: Download and extract the file hw5A.zip from eLearning at Course Content | Unit 1 | Homework #5.

Part A: Recall the insertion sort code from lecture 16 and lab 8:

```python
def insertionSort(myList):
    """Rearranges the items in myList so they are in ascending order""
    for firstUnsortedIndex in range(1, len(myList)):
        itemToInsert = myList[firstUnsortedIndex]
        testIndex = firstUnsortedIndex - 1
        while testIndex >= 0 and myList[testIndex] > itemToInsert:
            myList[testIndex+1] = myList[testIndex]
            testIndex = testIndex - 1
        # Insert the itemToInsert at the correct spot
        myList[testIndex + 1] = itemToInsert
```

It sorts in ascending order by inserting the first item from the unsorted part into the sorted part on the left end of the list. For example, the inner-loop takes 25 and "inserts" it into the sorted part of the list "at the correct spot."

<table>
<thead>
<tr>
<th>Sorted Part</th>
<th>Unsorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5</td>
<td>6 7 8 9 10 25</td>
</tr>
</tbody>
</table>

After 25 is inserted into the sorted part, the list would look like:

<table>
<thead>
<tr>
<th>Sorted Part</th>
<th>Unsorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5</td>
<td>6 7 8 9 10 25</td>
</tr>
</tbody>
</table>

For Part A, I want you to modify and improve insertion sort (still sorting in ascending order) by:
- build the sorted part on the right end of the list and insert the last unsorted item into it repeatedly,
- speedup the inner-while-loop by:
  - starting the sort by finding and swapping the largest item with the right-most item. This only needs to be done once at the start of the sort before the nested outer and inner loops.
  - simplify the while-loop condition since the itemToInsert will always be ≤ the right-most item

Include a timing program as in lab8.zip for your program that times your sorting algorithm several time with different initial orderings of 15,000 list items. The initial orderings of items are: descending order, ascending order, random order, and random order again to check for consistence. Report the times and compare your times to the original insertion sort from lab8. You can use the included hw5A/insertionSortComparison.docx (or .pdf) file. Also, include an explanation of the timing comparisons.

On eLearning (Course Content | Unit #2 | Homework #5 subfolder), submit a single .zip file, hw5A.zip, containing the following:
- the original insertion sort file insertionSort.py
- the completed improved insertion sort file improvedInsertionSort.py
- the completed insertionSortComparison.docx (or .pdf) file

(If you miss the deadline, you can still submit it without a late penalty. However, I there will be a homework 6, etc. and you don’t want to get too far behind!)
To start the homework: Download and extract the file hw5B.zip from cLearning at Course Content | Unit 1 | Homework #5.

Part B: In lecture 17 and lab 8 we wrote a simple heap sort that used the BinHeap class. Unfortunately, my lab 8 timings for 400,000 random integers were: 6.69 seconds for heap sort, 2.44 seconds for merge sort, and 1.24 seconds for quick sort. A couple factors make this simple heap sort slower:

1. this heap sort used a BinHeap object (e.g., myHeap=BinHeap()) which adds an extra layer of method calls (e.g., myHeap.insert(item) and myHeap.delMin()) which merge sort and quick sort did not have.
2. this heap sort uses percUp (in myHeap.insert(item)) which swaps the inserted item repeatedly with its parent until the inserted item satisfies the heap-order property. Each swap in percUp requires three moves.

We can re-implement the percUp to reduce moves by repeatedly comparing the parents to the itemToInsert, but instead of swapping just move the parents down (1 move each). When we find the correct spot, insert the itemToInsert once into the heap.

3. similarly, this heap sort uses percDown (in myHeap.delMin()) which moves the last item to the root and swaps this item repeatedly with its smaller child until it satisfies the heap-order property. Each swap in percDown requires three moves. Similar to the above discussion for percUp, we could modify the percDown to reduce the moves by repeatedly comparing the last item (i.e., the parent) with the smallest child, but instead of swapping just move the child up (1 move each). When we find the correct spot, insert the last item once into the heap.

4. this heap sort does not use the buildHeap method which takes as a parameter an unordered list and builds the heap from it by percDown the non-leaf items to build bigger and bigger heaps until all the items are in a single heap.
For Part B, we can improve heap sort (still sorting in ascending order) by:

1. eliminating the usage of the BinHeap class by using the list parameter (e.g., myList) to store items rearranged as a heap and “in-line” the code for heap methods right into the heap-sort code, so your heap-sort code calls no methods, but local functions instead.

2. improve the building of the initial “max-heap” containing all of the list items (called heapifying the list) by:
   - use index 0 to hold the root of the max-heap. This changes the relationship of parent and child indexes from that of BinHeap’s. A node at index i has: a left child at i*2+1, a right child at i*2+2, and a parent at (i-1)/2
   - use the concept employed in the “buildHeap” method in the BinHeap class: (1) think of the leaves as mini max-heaps of size 1 and (2) percolate down the non-leaves starting at index (len(myList)//2 - 1) through index 0 into the smaller heaps below to form bigger-and-bigger heaps.
   - while percolating down the non-leaves, don’t repeatedly “swap” down a branch of the heap (e.g., 3 moves per swap), but instead “insert” the non-leaf item into the correct spot in the branch by shifting larger children up the branch (one move per shift) before inserting at the correct spot.

3. after heapifying, use the max-heap of all list items in the unsorted part to perform a selection-sort like sort.

After heapifying the list in the above step we have the max. item at index 0:

```
0 1 2 3 4 5 6 7 8
myList: 60 50 25 40 45 20 10 35 15
```

"swap" max. item in heap at index 0 with lastUnsortedIndex to grow sorted part of the list

```
0 1 2 3 4 5 6 7 8
myList: 15 50 25 40 45 20 10 35 60
```

percolate 15 down to restore heap-order across unsorted part

```
0 1 2 3 4 5 6 7 8
myList: 50 45 25 40 15 20 10 35 60
```

Fortunately, both the heapify step and the modified selection-sort steps only need a commonly called permuteDown function. Your task will be to complete the heapifyList and permuteDown functions in the hw5B/improvedHeapSort.py file.

Include a timing program as in lab8.zip for your improved heap sort that times the sorting of a randomly generated list of a user-specified size. Report the times and compare your times to the original heap sort from lab8 on 400,000 items and 800,000 items. You can use the included hw5B/heapSortComparison.docx (or .pdf) file.

On eLearning (Course Content | Unit #2 | Homework #5 subfolder), submit a single .zip file, hw5B.zip, containing the following:

- your original heap sort file heapSort.py from lab 8
- the completed improved heap sort file improvedHeapSort.py
- the timeHeapSortComparison.py file (unchanged from original hw5B.zip)
- the completed heapSortComparison.docx (or .pdf) file

(If you miss the deadline, you can still submit it without a late penalty. However, I there will be a homework 6, etc. and you don’t want to get too far behind!)
1. So far, we have looked at simple sorts consisting of nested loops. The $\#$ of inner loop iterations $n(n-1)/2$ is $O(n^2)$. Consider using a min-heap to sort a list. (methods: BinHeap(), insert(item), delMin(), isEmpty(), size())

a) If we insert all of the list elements into a min-heap, what would we easily be able to determine?

**General idea of Heap sort:**

1. Create an empty heap

2. Insert all $n$ list items into heap

3. delMin heap items back to list in sorted order

b) What is the overall $O(\cdot)$ for heap sort?

2. Another way to do better than the simple sorts is to employ divide-and-conquer (e.g., Merge sort and Quick Sort). Recall the idea of **Divide-and-Conquer** algorithms. Solve a problem by:

   * dividing problem into smaller problem(s) of the same kind
   * solving the smaller problem(s) recursively
   * use the solution(s) to the smaller problem(s) to solve the original problem

In general, a problem can be solved recursively if it can be broken down into smaller problems that are identical in structure to the original problem.

a) What determines the “size” of a sorting problem?

b) How might we break the original problem down into smaller problems that are identical?

c) What base case(s) (i.e., trivial, non-recursive case(s)) might we encounter with recursive sorts?

d) How do you combine the answers to the smaller problems to solve the original sorting problem?

e) Consider why a recursive sort might be more efficient. Assume that I had a simple $n^2$ sorting algorithm with $n = 100$, then there is roughly $100^2 / 2 = 5,000$ amount of work. Suppose I split the problem down into two smaller sorting problems of size 50.

   * If I run the $n^2$ algorithm on both smaller problems of size 50, then what would be the approximate amount of work?

   * If I further solve the problems of size 50 by splitting each of them into two problems of size 25, then what would be the approximate amount of work?
3. The general idea merge sort is as follows. Assume “n” items to sort.
   - Split the unsorted part in half to get two smaller sorting problems of about equal size = n/2
   - Solve both smaller problems recursively using merge sort
   - “Merge” the solutions to the smaller problems together to solve the original sorting problem of size n

   a) Fill in the merged Sorted Part in the diagram.
   b) Describe how you filled in the sorted part in the above example?

4. Merge sort is substantially faster than the simple sorts. Let’s analyze the number of comparisons and moves of merge sort. Assume “n” items to sort.

   a) On each level of the above diagram write the WORST-CASE number of comparisons and moves for that level.
   b) What is the WORST-CASE total number of comparisons and moves for the whole algorithm (i.e., add all levels)?
   c) What is the big-oh for worst-case?
5. **Quick sort** general idea is as follows.
   - Select a "random" item in the unsorted part as the **pivot**
   - Rearrange (*partitioning*) the unsorted items such that:
     - Quick sort the unsorted part to the left of the pivot
     - Quick sort the unsorted part to the right of the pivot

a) Given the following **partition** function which returns the index of the pivot after this rearrangement, complete the recursive **quicksortHelper** function.

```python
def quicksort(lyst):
    quicksortHelper(lyst, 0, len(lyst) - 1)

def quicksortHelper(lyst, left, right):
    def partition(lyst, left, right):
        middle = (left + right) // 2
        pivot = lyst[middle]
        lyst[middle] = lyst[right]
        lyst[right] = pivot
        boundary = left
        for index in range(left, right):
            if lyst[index] < pivot:
                temp = lyst[index]
                lyst[index] = lyst[boundary]
                lyst[boundary] = temp
                boundary += 1
        temp = lyst[boundary]
        lyst[boundary] = lyst[right]
        lyst[right] = temp
        return boundary
```

b) For the list below, trace the first call to partition and determine the resulting list, and value returned.

<table>
<thead>
<tr>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>26</td>
<td>93</td>
<td>17</td>
<td>50</td>
<td>31</td>
<td>44</td>
<td>55</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>left</th>
<th>right</th>
<th>index</th>
<th>boundary</th>
<th>pivot</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

b) What initial arrangement of the list would cause partition to perform the most amount of work?

c) Let “n” be the number of items between left and right. What is the worst-case $O(\_\_\_\_)$ for partition?
d) What would be the overall, worst-case \( O(\ ) \) for Quick Sort?

e) Ideally, the pivot item splits the list into two equal size problems. What would be the big-oh for Quick Sort in the best case?

f) What would be the big-oh for Quick Sort in the average case?

g) The textbook's partition code (Listing 5.15 on page 225) selects the first item in the list as the pivot item. However, the above partition code selects the middle item of the list to be the pivot. What advantage does selecting the middle item as the pivot have over selecting the first item as the pivot?
Objectives: You will gain experience:
- get a feel for simple sorts: selection, bubble, and insertion sorts
- get a feel for advanced sorts: heap, quick, and merge sorts

To start the lab: Download and unzip the lab8.zip file from eLearning.

Part A: The lab8.zip file you downloaded and extracted contains the following sorting algorithms which all sort in ascending order (i.e., from smallest to largest):
- bubbleSort.py - bubble sort code which does not check if it can stop early
- bubbleSortB.py - bubble sort code which stops early if no swapping is needed during a scan of the unsorted part
- insertionSort.py - the insertion sort
- selectionSort.py - the selection sort code we developed in class

Each program runs the sorting algorithm several times with different initial orderings of 10,000 list items. The initial orderings of items are: descending order, ascending order, random order, and random order again to check for consistence. Complete the following timings by running each program.

<table>
<thead>
<tr>
<th>Type of sorting algorithm</th>
<th>Initial Ordering of Items</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Descending</td>
</tr>
<tr>
<td>bubbleSort.py</td>
<td></td>
</tr>
<tr>
<td>bubbleSortB.py</td>
<td></td>
</tr>
<tr>
<td>insertionSort.py</td>
<td></td>
</tr>
<tr>
<td>selectionSort.py</td>
<td></td>
</tr>
</tbody>
</table>

Study the code and answer the following questions about the sorting algorithms:

a) Why does the bubbleSort algorithm take less time on the ascending ordered list than the descending ordered list?

b) Why does the bubbleSortB algorithm take A LOT less time on the ascending ordered list than the descending ordered list?

c) Why does the insertionSort algorithm take A LOT less time on the ascending ordered list than the descending ordered list?

d) Why does the insertionSort algorithm take less time on the descending ordered list than the bubbleSort algorithm on the descending ordered list?
c) Why does the selectionSort algorithm take less time on the descending ordered list than the insertionSort algorithm on the descending ordered list?

Part B: Advanced Sort comparison

a) Complete the heap sort function in `lab8/heapSort.py` which contains the template for the heap sort algorithm discussed in class. Recall the steps of the algorithm:

```
myList            unsorted list with n items
                   ↓
heap with n items
                   ↓
myList            sorted list with n items
```

Steps:
1. Create an empty heap
2. Insert all n list items into heap
3. delMin heap items back to list in sorted order

b) Time the heap sorting algorithm using `lab8/timeHeapSort.py` on 100,000 random items, 200,000 random items, and 400,000 random items.

<table>
<thead>
<tr>
<th># Items</th>
<th>Your Heap Sort Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td></td>
</tr>
<tr>
<td>200,000</td>
<td></td>
</tr>
<tr>
<td>400,000</td>
<td></td>
</tr>
</tbody>
</table>

c) Explain the $O(\cdot)$ for your heap sort algorithm?
d) The general idea merge sort is as follows. Assume "n" items to sort.
   - Split the unsorted part in half to get two smaller sorting problems of about n/2
   - Solve both smaller problem recursively using merge sort
   - "Merge" the solution to the smaller problems together to solve the original sorting problem of size n

![Diagram of merge sort](image)

The textbook's merge sort is in mergesort.py. Use the timeMergeSort.py program to run merge sort on a list of random items. Complete the following timing table:

<table>
<thead>
<tr>
<th>Random # Items</th>
<th>textbook's Merge Sort Timings</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td></td>
</tr>
<tr>
<td>200,000</td>
<td></td>
</tr>
<tr>
<td>400,000</td>
<td></td>
</tr>
</tbody>
</table>

e) Recall the general idea of Quick sort is as follows. Assume "n" items to sort.
   - Select a "random" item in the unsorted part as the pivot
   - Rearrange (called partitioning) the unsorted items such that:

   ![Diagram of quicksort](image)

   - Quick sort the unsorted part to the left of the pivot
   - Quick sort the unsorted part to the right of the pivot

The lecture 17 quick sort is in quicksort.py. Use the timeQuickSort.py program to run quick sort on a list of random items. Complete the following timings to get a feel for the "speed" of quicksort.

<table>
<thead>
<tr>
<th># Items</th>
<th>Lecture 17 Quick Sort Timings</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td></td>
</tr>
<tr>
<td>200,000</td>
<td></td>
</tr>
<tr>
<td>400,000</td>
<td></td>
</tr>
</tbody>
</table>

All three advanced sorting algorithms are $O(n \log n)$ on initially random data. Why do you suppose quick sort is the fastest advanced sort on random items?
After you have answers and correct code for all parts of the lab, submit a lab8.zip containing your code on eLearning. If you do not get done today, then submit it by next week’s lab period.

Remember to save your lab8 files for later usage on homework assignments!

**Part C: EXTRA CREDIT**

a) Write (pencil-and-paper below) a variation of bubble sort that:
- sorts in descending order (largest to smallest)
- builds the sorted part on the left-hand side of the list, i.e.,

<table>
<thead>
<tr>
<th>Sorted Part</th>
<th>Unsorted Part</th>
</tr>
</thead>
</table>

(Your code does NOT need to stop early if a scan of the unsorted part has no swaps)

```python
def bubbleSortC(myList):
    # Inner loop scans from right to left
    # across the unsorted part swapping
    # adjacent items that are "out of order"
```

b) Implement and test your `bubbleSortC` code.
Test 2 Review Topics

Test 2 will be Thursday October 29th in class. It will be closed-book and notes, except for one 8.5” x 11” sheet of paper (you can use front and back) containing any notes that you want AND the Python Summary handout (in front of you packet).

The test will cover Chapters 4 and 5. The following topics (and maybe more) with be covered:

Chapter 4: Recursion
Recursive functions: base-case(s), recursive case(s), tracing recursion via run-time stack or recursion tree, “infinite recursion”
Costs and benefits of recursion
Recursive examples: countDown, OrderedList __str__ method, fibonacci, factorial, binomial coefficient
Divide-and-Conquer technique of solving a problem. Examples: fibonacci, coin-change problem
Backtracking technique of solving a problem: Examples: coin-change problem, maze (textbook)
General concept of dynamic programming solutions for recursive problems that repeatedly solve the same smaller problems over and over. Example fibonacci, coin-change problem, binomial coefficient

Chapter 5: Searching and Sorting
Sequential/Linear search: code and big-oh analysis
Binary Search: code and big-oh analysis
Python List implementation (ListDict) of dictionaries and big-oh analysis
Hashing terminology: hash function, hash table, collision, load factor, chaining/closed-address/external chaining, open-address with some rehashing strategy: linear probing, quadratic probing, primary and secondary clustering hashing implementation of dictionaries (ChainingDict and OpenAddrHashDict) and their big-oh analysis
General idea of simple sorts
Simple sorts: selection, bubble, insertion sorts and their big-oh analysis
Advanced sorts and their big-oh analysis: heap sort, quick sort and merge sort
Data Structures - Test 2

Question 1. (10 points) What is printed by the following program? Output:

```python
def recFn(myString, index):
    print(index)
    if index == 0:
        return "here"
    elif index == 1:
        return "there"
    elif index > 4:
        return recFn(myString, index-1) + myString[index]
    else:
        return myString[index] + recFn(myString, index-2)

print("result =", recFn("abcdefgh", 5))
```

Run-time Stack

<table>
<thead>
<tr>
<th>ref_addr</th>
<th>myString</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Question 2. (10 points) Write a recursive Python function to compute the following mathematical function, $G(n)$:

- $G(n) = n$ for all value of $n \leq 3$ (e.g., $G(2)$ is 2)
- $G(4) = 6$ if $n = 4$
- $G(n) = G(n-5) + G(n-4) + G(n-2)$ for all $n$ values $> 4$.

```python
def G(n):
```

Question 3. a) (7 points) For the above recursive function $G(n)$, complete the calling-tree for $G(9)$.

```
G(9)
```

```
G(4)

G(5)
```

```
G(7)
```

b) (2 points) What is the value of $G(9)$?

c) (1 point) What is the maximum height of the run-time stack when calculating $G(9)$ recursively?
Question 4. Consider the following simple sorts discussed in class -- all of which sort in ascending order.

```python
def bubbleSort(myList):
    for lastUnsortedIndex in range(len(myList)-1, 0, -1):
        for testIndex in range(lastUnsortedIndex):
            if myList[testIndex] > myList[testIndex+1]:
                temp = myList[testIndex]
                myList[testIndex] = myList[testIndex+1]
                myList[testIndex+1] = temp

def insertionSort(myList):
    for firstUnsortedIndex in range(1, len(myList)):
        itemToInsert = myList[firstUnsortedIndex]
        testIndex = firstUnsortedIndex - 1
        while testIndex >= 0 and myList[testIndex] > itemToInsert:
            myList[testIndex+1] = myList[testIndex]
            testIndex = testIndex - 1
        myList[testIndex + 1] = itemToInsert

def selectionSort(aList):
    for lastUnsortedIndex in range(len(aList)-1, 0, -1):
        maxIndex = 0
        for testIndex in range(1, lastUnsortedIndex + 1):
            if aList[testIndex] > aList[maxIndex]:
                maxIndex = testIndex
        # exchange the items at maxIndex and lastUnsortedIndex
        temp = aList[lastUnsortedIndex]
        aList[lastUnsortedIndex] = aList[maxIndex]
        aList[maxIndex] = temp
```

<table>
<thead>
<tr>
<th>Timings of Above Sorting Algorithms on 10,000 items (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of sorting algorithm</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>bubbleSort.py</td>
</tr>
<tr>
<td>insertionSort.py</td>
</tr>
<tr>
<td>selectionSort.py</td>
</tr>
</tbody>
</table>

a) (5 points) Explain why selectionSort's timings are roughly the same regardless of the initial ordering of items.

b) (5 points) Explain why insertionSort on a descending list (14.2 s) takes longer than insertionSort on an ascending list (0.004 s).

c) (5 points) Explain why bubbleSort is O(n²) in the worst-case, where n is the size of the list being sorted.
Question 5. (20 points) In class we discussed the insertionSort code shown in question 3 on page 2 which sorts in ascending order (smallest to largest) and builds the sorted part on the left-hand side of the list.

For this question write a variation of insertion sort that:
- sorts in **descending order** (largest to smallest), and
- builds the **sorted part on the right-hand side of the list**, i.e.,

<table>
<thead>
<tr>
<th>Unsorted Part</th>
<th>Sorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>myList:</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3 4 5</td>
<td>6 7 8</td>
</tr>
<tr>
<td>· · 25</td>
<td>40 90 60</td>
</tr>
<tr>
<td>50 45 35 20</td>
<td>10</td>
</tr>
</tbody>
</table>

```python
def insertionSortVariation(myList):
```

Question 6. Recall the general idea of Heap sort which uses a min-heap (class BinHeap with methods: BinHeap(), insert(item), delMin(), isEmpty(), size()) to sort a list.

**General idea of Heap sort:**
1. Create an empty heap
2. Insert all n list items into heap
3. delMin heap items back to list in sorted order

```
from bin_heap import BinHeap

def heapSort(myList):
    myHeap = BinHeap() # Create an empty heap
```

a) (5 points) Complete the code for heapSort so that it **sorts in descending order**

b) (5 points) Determine the overall $O(\cdot)$ for your heap sort and briefly justify your answer. Let $n = \text{len}(\text{myList})$. 

111
Question 7. Two common rehashing strategies for open-address hashing are linear probing and quadratic probing:

| quadratic probing | Check the square of the attempt-number away for an available slot, i.e.,
|                   | [home address + (rehash attempt #)^2 + (rehash attempt #))/2] % (hash table size), where the hash table size is a power of 2. Integer division is used above |

a) (10 points) Insert “Andrew Berns” and then “Sarah Diesburg” using Linear (on left) and Quadratic (on right) probing.

Hash Table with Linear Probing

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ben Schafer</td>
<td>John Doe</td>
<td></td>
<td>Mark Fienup</td>
<td>Philip East</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hash function

- hash(John Doe) = 2
- hash(Philip East) = 5
- hash(Mark Fienup) = 4
- hash(Ben Schafer) = 1
- hash(Andrew Berns) = 1
- hash(Sarah Diesburg) = 4

Hash Table with Quadratic Probing

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ben Schafer</td>
<td>John Doe</td>
<td></td>
<td>Mark Fienup</td>
<td>Philip East</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

b) In open-address hashing (e.g., pictures above), the average number probes/compares for various load factors is:

<table>
<thead>
<tr>
<th>Probing Type</th>
<th>Search outcome</th>
<th>0.25</th>
<th>0.5</th>
<th>0.67</th>
<th>0.8</th>
<th>0.99</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Probing</td>
<td>unsuccessful</td>
<td>1.39</td>
<td>2.50</td>
<td>5.09</td>
<td>13.00</td>
<td>5000.50</td>
</tr>
<tr>
<td></td>
<td>successful</td>
<td>1.17</td>
<td>1.50</td>
<td>2.02</td>
<td>3.00</td>
<td>50.50</td>
</tr>
<tr>
<td>Quadratic Probing</td>
<td>unsuccessful</td>
<td>1.37</td>
<td>2.19</td>
<td>3.47</td>
<td>5.81</td>
<td>103.62</td>
</tr>
<tr>
<td></td>
<td>successful</td>
<td>1.16</td>
<td>1.44</td>
<td>1.77</td>
<td>2.21</td>
<td>5.11</td>
</tr>
</tbody>
</table>

The "general rule of thumb" tries to keep the load factor (i.e., # items / hash-table size) between 0.5 and 0.67.
- (4 points) Why don't you want the load factor to exceed 0.67?
  - (4 points) Why don't you want the load factor to be less than 0.5?

c) (7 points) In closed-address hashing (e.g., ChainingDict picture below), if the load factor is close to 1, say 0.99, would you expect the average-case number of probes/compares to be between 1 and 2? (Justify your answer)