Computer Science 222, Fall 2012
Laboratory 11 – Searching and Sorting, List Functions

Learning objectives:

- Perform linear search and binary search on data.
- Compare efficiency of searches.
- Perform sort of data.
- Compare efficiency of built-in Python sort versus insertion sort.
- Use Python's built-in list functions

Download the files `dataSorted.txt`, `searchTest.py` and `sortTest.py` from Oaks.

1. Linear Search

Save your work in a file `algorithms.py`. The file `dataSorted.txt` contains series of numbers in sorted order on multiple lines of a file. Each line may contain 1 to many numbers. Write a function `readData(filename)` that accepts the name of a file, builds a list of the data, and returns the list. Write another function `isInLinear(srchVal, values)` that accepts a value and a list of comparable values, and it returns a boolean. It should return `True` if the value is found in values and `False` otherwise. Your code should perform an efficient linear search. Add code to `main()` to test your functions.

2. Binary Search

Add a function `isInBinary(srchVal, values)` that works the same way as `isInLinear()` but instead performs a binary search of the data. Add code to `main()` to test your function.

3. Comparative analysis of searches

Download and run the program `searchTest.py`. This code searches for several different elements in a small list and a large list. It outputs the runtime (in seconds) of your linear search code vs. your binary search code. Make observations about the output as part of your header comment for `algorithms.py`.

4. Sorting

Add a function `insertionSort(values)` to `algorithms.py`, that accepts a list of values and sorts them in place using the insertion sort algorithm.

Briefly, the insertion sort algorithm works like this. Assume that the first element in the list is sorted. Examine each remaining element in turn. Move the element to its left until either it is larger than (or equal to) the element to its left, or it is the first element in the list. (Try to move it to the left in as efficient a way as possible.) You are done when you have sorted the last element in the original list in this way.

For example, suppose the list contains [4, 2, 3, 1, 5]. Assume that the 4 is sorted. Since 2 is not larger than 4, move the 2 to the left. Since 2 now is the first element in the list, stop. You now have [2, 4, 3, 1, 5], and you are ready to sort the 3. Since 3 is not larger than 4, move 3 to the left. Since 3 is larger than 2, stop. Now you have [2, 3, 4, 1, 5], and you are ready to sort the 1. Since 1 is not larger than 4, 3, or 2, move the 1 to the left three times. Since 1 now is the first element in the list, stop. You now have [1, 2, 3, 4, 5]. Since 5 is larger than 4, stop.

You're done.

5. Comparative analysis of sorts

Download and run the program `sortTest.py`. This code sorts a small list and a large list using your sort and Python's built-in list method `sort()`. Make observations about the output as part of your header comment for `algorithms.py`.

6. Using Python's built-in list functions

Write a program, `noDuplicates.py`, that creates a list containing duplicate entries. Print the list, call `removeDuplicates(list)`, and print the list again.

The function `removeDuplicates(list)` uses list functions available in Python (p. 345) to remove all duplicate entries from `list`, resulting in a list with a single copy of each entry. For example, if the original list is `[1,2,3,1,2,3,4,5,4,5,6,6,6,7,1,7]`, the final list is `[1,2,3,4,5,6,7]`. This function removes the duplicates in the original list and does not return anything.

Upload the files:

`algorithms.py`   `noDuplicates.py`