22C:16 Homework 10

Due via ICON on Monday, May 2nd, 4:59 pm

Submit the solutions to all 4 problems, but we will grade some 2 problems of our choice from your submission.

1. A sequence of numbers

   $$x_1, x_2, \ldots, x_n$$

   is called *bitonic* if there is an integer j, $1 \leq j \leq n$, such that $x_1 < x_2 < \cdots < x_j$ and $x_j > x_{j+1} > \cdots > x_n$. In other words, a bitonic sequence consists of a bunch of numbers at the beginning that are in strictly ascending order followed by a bunch of numbers in strictly descending order.

   Write a function called `findBitonicMax` that takes a list `L` of numbers that forms a bitonic sequence and returns an index of the maximum number in `L`. Of course, it is possible to find the maximum number by simply doing a left-to-right scan of the elements in `L`. A much more efficient way would be to solve this problem by using a "binary search" like algorithm. Specifically, you could examine the element in the middle of `L` along with its two neighbors on either side and based on what you see, you can determine if (i) the maximum is the middle element, (ii) the maximum is on the left side of the middle element, and (iii) the maximum is on the right side of the middle element.

   Implement this "binary search" like algorithm (think recursion!) using the following function header:

   ```
   def findBitonicMax(L, first, last):
   ```

   This function is meant to return an index of the maximum element in `L[first:last+1]`. To find the maximum element in all of `L`, the user would have to call this function as `findBitonicMax(L, 0, len(L)-1)`.

2. This problem is also about bitonic sequences (see Problem 1). Write a function called `searchBitonic` that takes a list `L` that consists of a bitonic sequence and a key value `k` and determines if `k` is an element of `L`; the function simply returns `True` or `False` depending on whether it found `k` in `L`.

   To do this efficiently, you should call the function `findBitonicMax` first to find an index `j` such that `L[0:j+1]` is strictly ascending and `L[j:len(L)]` is strictly descending. Then you can do a binary search on `L[0:j+1]` and a separate binary search on `L[j:len(L)]`. You should make sure that `searchBitonic` runs in time proportional to $\log_2 n$, in the worst case.

   Use the following function header:

   ```
   def searchBitonic(L, k, first, last):
   ```

   This function is meant to search for `k` in `L[first:last+1]`. To search for `k` in all of `L`, the user would have to call this function as `searchBitonic(L, k, 0, len(L)-1)`.

3. In this problem you will use the file `words.dat` that I posted for Homework 9. Also, you will use the `makeNeighbors` function that you made for Problem 1 in Homework 9.

   You are to write a program that reads two 5-letter words from the input. Let us call these words s and t. Your program should compute and outout a sequence of words that starts with s and ends with t such that each intermediate word in the sequence is obtained from the previous word in the sequence by replacing exactly one letter in the word. You may assume that s and t appear in `words.dat` and every word in the output is required to belong to `words.dat`. It is possible that there is no such sequence from s to t and in this case your program should produce a message saying so.

This is a fun application of recursion that is usually taught in more advanced classes (e.g., 22C:21 or 22C:31). However, the algorithm that you can use to solve this problem is fairly natural and corresponds to how you would explore a maze. The algorithm is called *depth first search* and you will find many expositions of this on the web.

The basic idea is to start from word s, mark it "visited" and travel to an (arbitrary) neighbor; let us call this neighbor v. On arriving at v, you mark it "visited" and then move on to an arbitrary not-yet-visited neighbor. The process continues in this manner, until you arrive at a node that has *no* not-yet-visited neighbors. You have now reached a dead end and it is time to *backtrack* to the most recently visited word that has one or more not-yet-visited neighbors. This process is naturally implemented by making use of recursion.

4. This problem asks you to compare the running times of *selection sort* and *merge sort*. Use the Python code posted one the course website for both of these algorithms.

   To perform this experiment, let us start with a positive integer n that will be the size of the list you want sorted. Randomly generate m lists, each of size n, and sort each of these lists first using selection sort and then using merge sort. Compute the average running time of selection sort, with the average taken over m runs. Similarly, compute the average running time of merge sort. My suggestion is to use $m = 20$; but depending on how slow your selection sort function is running, you may have to reduce this down to $m = 10$. For n, my suggestion is to use values 100,000 to 190,000 in increments of 10,000.

   An easy way to generate random (unsorted) sequences of numbers is to use the `shuffle` function from the `random` module.

   After you have finished your experiment, make two plots, one showing the running time of selection sort and the other showing the running time of merge sort. Write a paragraph discussing your results. Make sure your write-up address the following issues: (i) which of the algorithms is faster and how do the relative times change with increasing n, and (ii) do the plots of the running times you see in your experiments correspond to what you have learned about the running times of these algorithms in class.
