Sorting Out Quicksort

sort /sôrt/
verb
gerund or present participle: sorting
1. arrange systematically in groups; separate according to type, class, etc.
"the mail was sorted"

2. (INFORMAL) resolve (a problem or difficulty).
"the problem with the engine was soon sorted"

This homework requires you to implement insertion sorting and quicksort functions (methods) and associated code to test and analyze the sorting routines on various data files. As always for our programming assignments, you can write your code in C++, Java, or Python 3. You will use system calls to print out the time required by sections of your code; we'll provide examples and guidance on how to do this. Your version of quicksort will use the "tuning" discussed in class where small lists are not sorted recursively with quicksort but instead are sorted with insertion sort.

Part of this homework will be like a lab exercise or experiment where you'll run your code on different data files, study the time taken by your sorting algorithm, and answer some questions. You'lll submit a PDF file with the answers to those questions. (MARK: they can submit two files to GradeScope, right?) But you will also submit your code to GradeScope where it will be run against test-cases to show that it sorts correctly. (GradeScope will not test for efficiency or time-complexity.)

What You Will Code

Here are instructions for what we want you to code.

- Write a function `insertionsort()` that takes 3 parameters: a list, a *start* index, and an *end* index. The function will sort the portion of the list from index *start* through *end* (inclusive) using insertion sort. The list parameter will be a Python list, a Java List or a C++ vector., and it will store integer values. The function will not return anything (void or the equivalent). List elements will be sorted in non-descending order. Lists in these languages are zero-indexed, so if *start* is 0 then sorting starts at the beginning of the list. If *end* is -1 then sorting is done to the last element of the list.

- Write a function `is_sorted()` that can be used to check your sorting method by verifying that the portion of the between index *start* and *end* are correctly sorted. It takes 3 parameters: a list, a *start* index, and an *end* index, that work as described above. This function must be in a separate file from your sorting code; see below for more information on this. It will print "yes" or "no" to the console to show if the list is sorted or not.

- Write a function `quicksort()` that takes 4 parameters: a list, a *start* index, an *end* index, and an int value *minsize*. This function will work like the insertion sort function except that it will use quicksort. (See next item for details about partitioning.) The first 3 parameters serve the same role as for the insertion sort function. The 4th parameter is used to control what quicksort does when it's processing a small list, that is one that has $\leq$ *minsize* elements. If *minsize* > 1 and the size of the sub-list between *start* and *end* is $\leq$ *minsize*, then that sub-list is sorted with your insertion sort function. If *minsize* ≤ 1 or the size of the sublist is $>$ *minsize*, then insertion sort is not used on the

small sub-list. (Observe that passing the function a *minsize* value ≤ 1 means that your quicksort will not use insertion sort at all.)

- Your quicksort function should call a separate `partition()` function. You may choose Lomuto's or Hoare's algortihm (both covered in the text book); put a comment at the start of your function stating which you're using. Your code should randomly choose an element in the sub-list and swap it into the right position for use as the partition element, as discussed in class. However, be prepared to comment out this randomizaton (or disable it somehow) for one of the experiments you'll do (more info below on this). (You do not need to use any of the approaches we'll discuss after Module 1 for finding the median or something close to it for partitioning.)

- You should have some kind of driver program that reads two values from the command line: the first is the name of a file in the current directory that contains integer values, and the second is the *minsize* value for quicksort. This program will read integer values from the file into a list and sort the entire list with your quicksort function, using the *minsize* value. It will then call the `is_sorted()` method on the entire list, which will print "yes" or "no" to the console to show if the list is sorted or not. The input file can contain integer values on the same line or on separate lines.
  This is the program that you will submit to GradeScope (along with any other code files needed to make it work) to show your algorithms work. You will probably also use this program (or modified versions of it) to do the experirments listed in the next section.

The Exercises

Coming soon!

What's Below is The Text From Wiring (will be modified)

Input

The input file will begin with one line containing integers J and C, the number of junction points and the number of possible connections respectively. The next J lines will each specify the name of a junction point along with the type (breaker, switch, light, outlet, box). When a switch is listed, the lights that need to be behind that switch will be always listed next to indicate this dependency. There will only be one breaker box. The next C lines specify the connections by providing the name of two junction points and the cost between them.

Output

Output the cost of the minimum wiring for the house that adheres to all of the constraints

Sample Input

```
6 8
b1 breaker
j1 box
s1 switch
l1 light
l2 light
o1 outlet
b1 j1 5
b1 o1 1
j1 s1 1
j1 o1 2
o1 l1 1
l1 l2 2
s1 l1 6
s1 l2 1
```

Sample Output

```
7
```