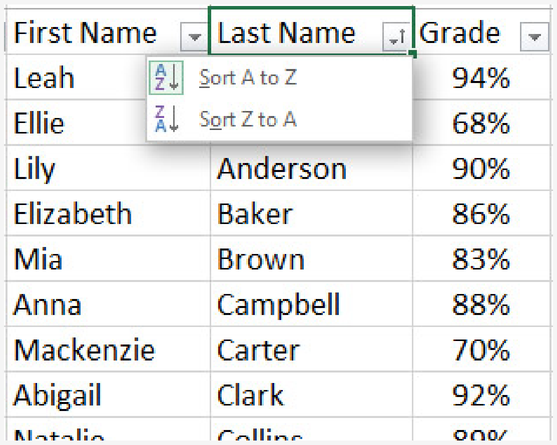
LAB 07 : SORT DESIGN

Due Saturday at 5:00 PM MST

This week, we will create a design for a sort algorithm.

Program Description

Sorting is the process of taking an unordered list and making it ordered. For example, alphabetizing is a sorting process. Organizing a to-do list from least important to most important is also a sorting process. In each case, we are changing the sequence of the elements in a list to conform to some criterion.



First Name	Last Name	Grade
Leah		94%
Ellie		68%
Lily	Anderson	90%
Elizabeth	Baker	86%
Mia	Brown	83%
Anna	Campbell	88%
Mackenzie	Carter	70%
Abigail	Clark	92%
Natalie	Collins	80%

Sorting elements in a collection is a widely used operation in many programming contexts. Lists of files are sorted before they are presented to the user in a FileOpen dialog box. Elements in a database are often sorted in order to facilitate more efficient retrieval. Anytime a binary search is utilized to quickly find an element in an array, the array needs to be sorted. There are literally hundreds of sorting algorithms identified by computer scientists. This week, we will describe one.

Algorithm in Plain English

Imagine a teacher sorting a collection of assignments by the student's last name. She picks up the stack of papers (representing the unsorted array) and looks for the student's name that sorts last (the highest value). Because the stack is unsorted, she must perform a linear search. When the highest value is found, she places it on her desk (representing the sorted portion of the array) and continues by looking for the next highest value. As she continues, the stack of papers in her hand (the unsorted array) gets smaller and the stack of papers on the desk (the sorted array) gets larger. When there are no papers remaining in her hand, the stack on the desk is the resulting sorted list.

Detailed Description of the Algorithm

A central idea of the sort is the pivot which separates the side of the array that is already sorted from the side that remains to be sorted. In the first iteration, `i_pivot` refers to the right-most slot (`i_pivot = num - 1`). This represents the slot where the largest element in the unsorted side of the array must go.

Next, it is necessary to find the index of the largest element to the left of the pivot. Two variables are required: `i_check` representing the index in the unsorted side of the array that is currently being searched, and `i_largest` representing the largest element found thus far. If the element under `i_check` is greater than the element under `i_largest`, then `i_largest` receives the value under `i_check`. This continues as `i_check` goes from the slot one to the left of `i_pivot` all the way to slot 0. Now all that remains is to swap the element under `i_pivot` with the element under `i_largest` if the element under `i_largest` happens to be bigger than the one under `i_pivot`.

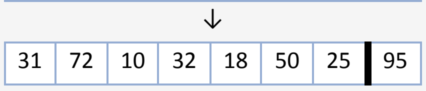
Example

To demonstrate how this works, consider the following eight numbers to be sorted. Initially the list is unsorted and no items are in their proper location. We will represent `i_pivot` with the thick vertical bar.

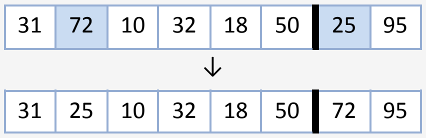
31	72	10	32	18	95	25	50

Looking at the array to the right of `i_pivot`, we discover that 95 is the largest value. Thus, 95 and 50 are swapped.

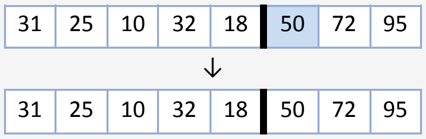
31	72	10	32	18	95	25	50



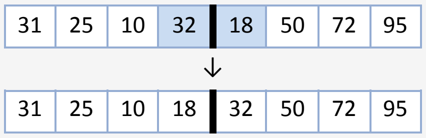
The largest value is now 72 in slot 1. It is swapped with 25 in slot 6.



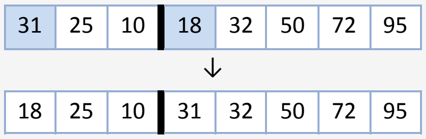
Since the search space includes the element under `i_pivot`, we discover that 50 is the highest. Nothing happens here.



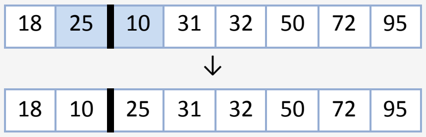
The next highest value is 32, so slots 3 and 4 are swapped.



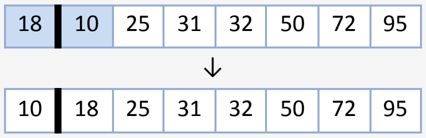
A search through {31, 25, 10, 18} yields slot 0 with the greatest value. Thus we swap 31 and 18.



The next search through {18, 25, 10} yields slot 1 with the highest value. Thus we swap 25 and 10.



The last step is to search through {18, 10}. Since the value under slot 0 is larger, we perform the last swap of the sort.



There is just one element left in the unsorted side of the pivot. This element is guaranteed to be the smallest item in the array and, by itself, is sorted. Thus, the entire array is sorted!

Assessment

To submit this assignment, three things are needed: a pseudocode program, the algorithmic efficiency, and a program trace. This will be submitted through I-Learn as a **single-file PDF**.

As with two weeks ago, please use the "Comments..." field to answer the following questions:

- How long did it take for you to complete this assignment?
- What was the hardest part of the assignment?
- Was there anything unclear about the instructions or how you were to complete this lab?

Pseudocode Program

Write the pseudocode for our sort algorithm. Note that your pseudocode must match the algorithm described above. Not just any sorting algorithm will work!

Your program must do the following:

- Prompt the user for a file containing a collection of words.
- Read the file in JSON format.
- Perform a sort on the list of words.

- Display the contents of the list on the screen.

Make sure your pseudocode uses a fixed-width font such as Consolas or Courier. Also, make sure you single-space your code. You might also need to set the space-before and space-after to 0.

Algorithmic Efficiency

You are required to compute the algorithmic efficiency of this sorting algorithm. Both name the efficiency (such as O(log n)) and give a rationale as to why it is what you say it is.

Program Trace

You will also need to perform a program trace on the sort. This can be done with pen and paper. If you do it this way, take a picture and attached it to your submission document. Alternatively, you can do with word processor or a spreadsheet. In either case, perform a trace using following data set:

```
{
  "array": [ "26", "6", "90", "55" ]
}
```

Assessment

Your grade for this activity will be according to the following rubric:

	Exceptional 100%	Good 90%	Acceptable 70%	Developing 50%	Missing 0%
Efficiency 20%	It is unambiguous that the correct algorithmic efficiency was determined	Efficiency determination and rationale are correct	Insufficient rationale or incorrect efficiency	There exists an informal discussion of the algorithmic efficiency	Algorithmic efficiency was not computed for this problem
Trace 30%	The trace is correct	The trace is correct except for one or two minor errors	One major error occurred or the trace is not detailed enough to be helpful	An attempt was made to trace the output, but it did not follow any of the guidelines	No program trace exists
Design Quality 40%	The most elegant and correct solution is found	The design completely covers the problem definition	One aspect of the problem definition is missing, one aspect of the design will not work as expected, or the wrong sort was used	Elements of the solution are present	Pseudocode is missing or the provided solution does not resemble the problem definition
Professionalism 10%	Professional, beautiful, elegant, single-spaced, using a fixed-width font	Everything is clear and legible, pseudocode used correctly	Misspelling, smudge, incorrect pseudocode, or examples of unprofessionalism	At least one aspect of the design is too messy to read or is not pseudocode	Difficult or impossible to read