

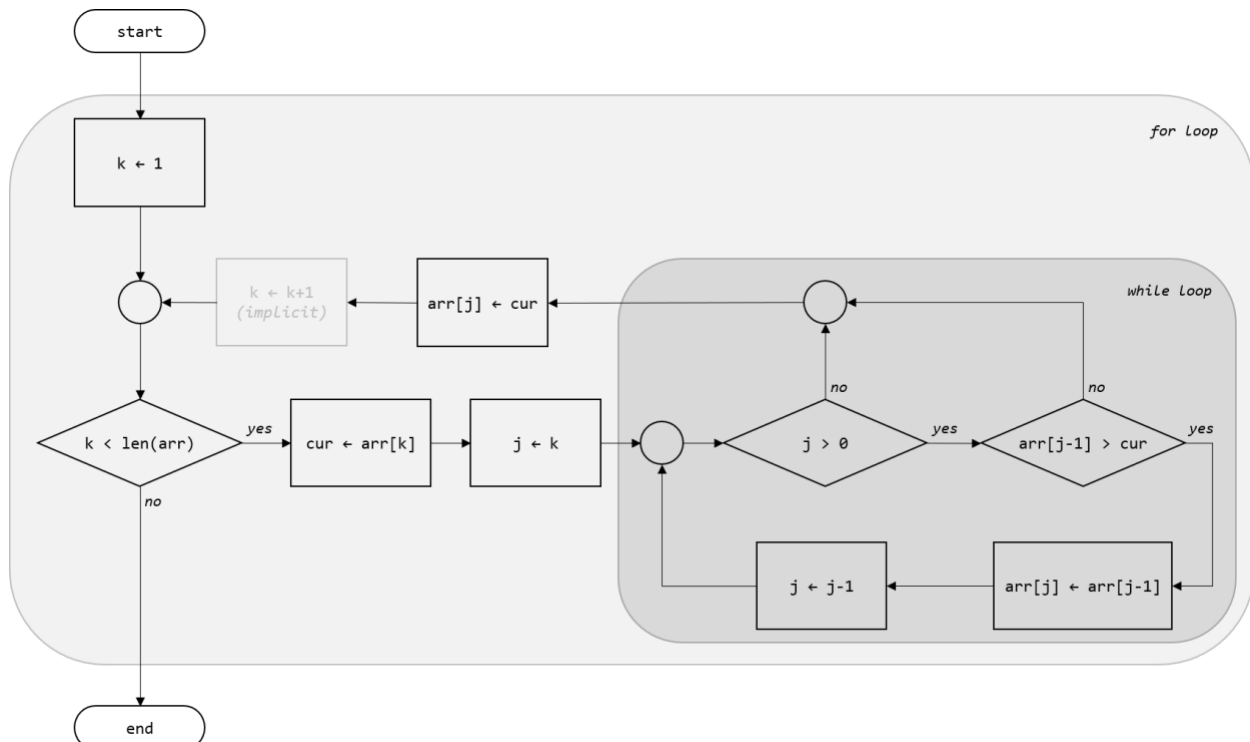
## CSCI 241 Data Structures

### Project 1: A Battle of Sorts

In class, we have traced through the insertion sort algorithm as implemented in Python. Recall from the demonstration that insertion sort works by considering each value in the array starting at position 1 (not position 0). It then determines the correct location for that value by comparing it to the values to its left, shifting right any greater values it encounters. You know you have found the correct location for the value as soon as the value to your left is not greater than what you are trying to place. This is the approach that we worked through in class, and there is a video example available on the course website. To the right is the code that we used in class, which is already included in the skeleton file.

```
def insertion_sort(arr):  
    for k in range(1, len(arr)):  
        cur = arr[k]  
        j = k  
        while j > 0 and arr[j-1] > cur:  
            arr[j] = arr[j-1]  
            j = j - 1  
        arr[j] = cur
```

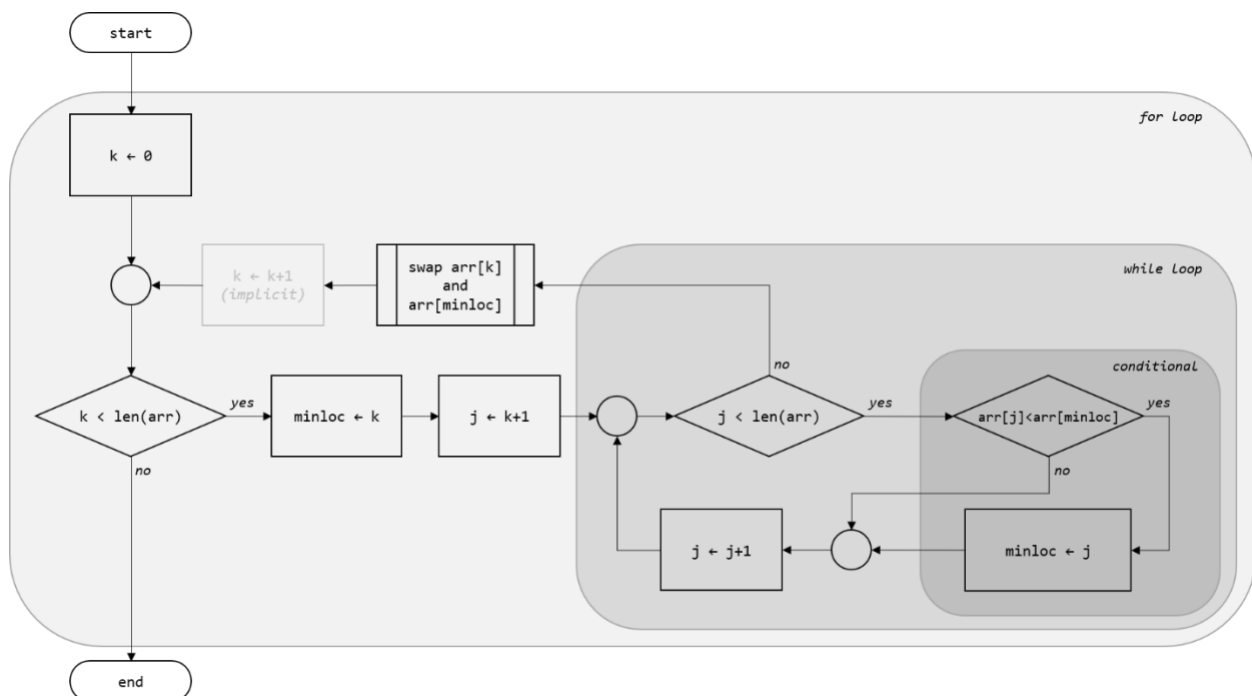
Another way to view the same algorithm is with a flow chart like the one below. This view shows the logical flow of the program and the actions that are taken and decisions that are made along the way. To trace the program using the flow chart, create an array of values and begin at the node labeled "start." Follow the arrows and execute any actions contained in rectangle nodes along the way. When you encounter a diamond-shaped node, answer the question in it and take the path labeled with the answer. Note the shaded regions that correspond to the for loop and while loop of the Python implementation. The leftward arrow symbol represents the assignment operation. For example,  $k \leftarrow 1$  means that the variable  $k$  is assigned the value 1.



Another common approach to sorting is called selection sort. Instead of looking at a value and determining where to place it in the array, we look at a cell and determine which value should go in it. Starting at cell 0 (not cell 1), locate the minimum value in the cells indexed from 0 to  $n-1$ . Remember the index of the cell containing the minimum value. Once you know which cell has the minimum value, swap the values in that cell and cell 0. Repeat this for cell 1, locating the minimum value in the cells indexed 1 to  $n-1$ . Repeat until you have placed a value in every cell. Like insertion sort, this has the effect of building a sorted subarray on the left side of the array. Unlike insertion sort, the sorted portion is never changed, because having considered every candidate for those cells, we know they have been placed correctly. The figure to the right illustrates the structure of selection sort's implementation in Python. You'll provide the missing parts for your submission.

```
def selection_sort(arr):
    for 
        
        
        while 
            if 
                
            
        
    
```

Below is the flow chart that visually represents the selection sort algorithm. Trace this the same way you did for insertion sort: follow the arrows beginning at the start node and execute the instructions and answer the questions along the way. The new  symbol in this flow chart represents a multi-step process. Here, it's used for the swap operation that we covered in section 1.1.



In the skeleton file for this project, we provide the full implementation for insertion sort, a pass placeholder to be replaced with your selection sort implementation, and a main section that generates identical arrays to be sorted by each method. The count and order of the values in the arrays are determined by command-line arguments provided when you execute the program in Terminal or PowerShell. For example, invoking

```
python Sort.py 10 increasing
```

would generate two arrays with identical increasing sequences of length 10, such as

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

One of those arrays will be sent as a parameter to the `insertion_sort` function and the other will be sent as a parameter to the `selection_sort` function. We have already provided the main section that parses the arguments from the command line, generates two identical arrays as specified, and passes those arrays separately to `insertion_sort` and `selection_sort`. Note the commented-out print statements at the bottom of the main section. Begin by testing several runs with sizes small enough to verify by hand by uncommenting those lines and examining the arrays after they are sorted. Once you are confident that your selection sort implementation is working correctly, you're ready to begin the larger runs we'll use for counting instructions.

The major result of this project is to learn how the two sorting approaches differ in performance for various cases of input, and how those differences change as the size of the input grows.

Specifically, we consider arrays with values that initially are increasing, decreasing, or randomly ordered. For each of these orientations, we consider sizes of 10000, 20000, 30000, 40000, and 50000 elements. This means that we will invoke the `Sort.py` program fifteen times (three orientations, each with five sizes).

To measure performance, we will count how many times each line of code is executed for each function. Python has a module that will do this automatically. Rather than invoking `python` as usual and as illustrated above, we invoke it with the `trace` module and instruct it to count the lines as it traces execution. The command

```
python -m trace --count -C . Sort.py 10000 random
```

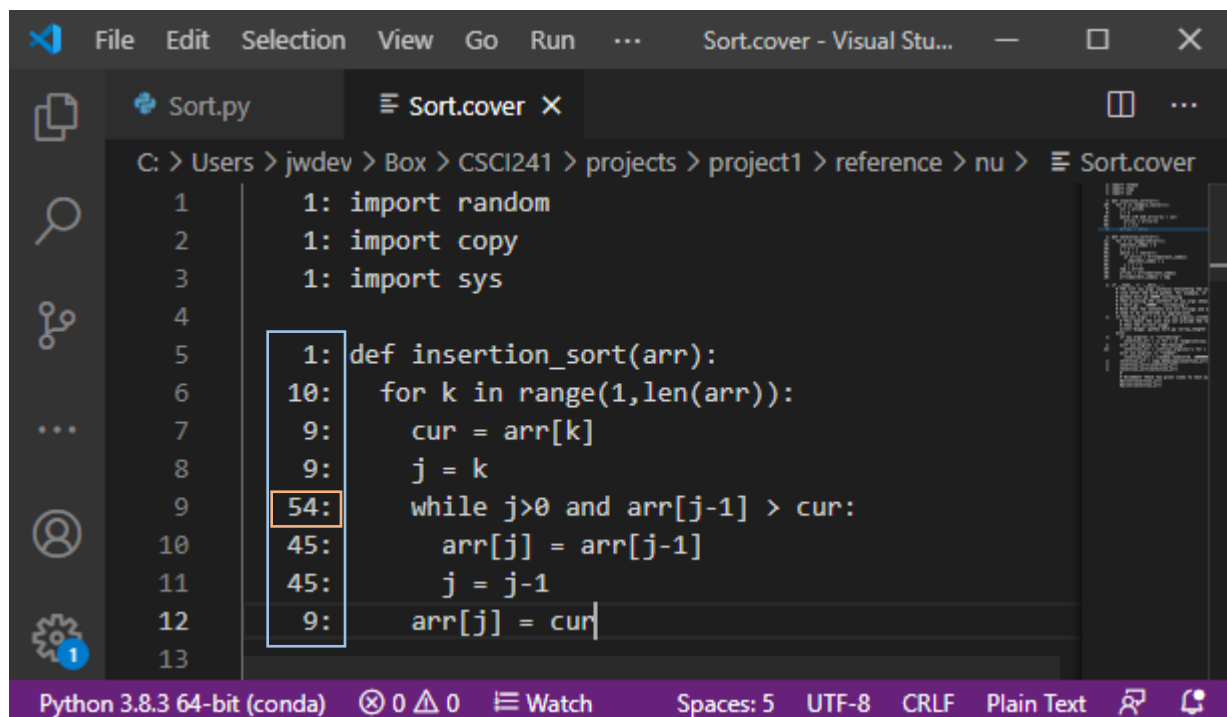
will sort 10000-element arrays of initially random values, as expected. The other components of the command mean the following:

**-m trace** Loads the trace module that knows how to observe your running program.

**--count** Instructs the trace module to count how many times each line is executed.

-C . Instructs the trace module to place the files containing the line counts in the current folder. In most terminal environments, a single dot means “this directory” and two dots means “the directory above me.” Here, the -C flag indicates that the next argument (the dot) is the folder where the count files should go.

When your program finishes, a file called Sort.cover will appear alongside your Sort.py file. The cover file is a copy of Sort.py, but the beginning of each line shows the number of times that line was executed. Below is a cropped image of the Sort.cover file showing only insertion sort for a small array of 10 initially decreasing values:



```
1: import random
1: import copy
1: import sys
1: def insertion_sort(arr):
10:     for k in range(1,len(arr)):
9:         cur = arr[k]
9:         j = k
54:         while j>0 and arr[j-1] > cur:
45:             arr[j] = arr[j-1]
45:             j = j-1
9:         arr[j] = cur
```

To tally the instruction count for insertion sort with this size and orientation, we add up the numbers in the blue box above, which indicate how many times each line was executed. Most students will accomplish this by copying the numbers to a spreadsheet and summing them with a formula (see the bonus opportunity below). Note that the number in the red box must be doubled, because the while conditional contains two statements:  $j > 0$  and  $\text{arr}[j-1] > \text{cur}$ . While it’s true that the second clause will occasionally be skipped when  $j \neq 0$ , this occurrence is rare for this project and the small difference will have no impact on your observations. To keep the project simple, we double the count for line 9 in all cases. Here, that means we use 108 for line 9’s count instead of 54. No other counts in either insertion sort or selection sort should be doubled, because all other lines are single statements.

### 5-point Bonus Opportunity

Instead of manually running the program fifteen times and copying the counts to a spreadsheet to tally and graph, write a second Python program that executes the `Sort.py` program with the correct parameters using loops. Note that every time your bonus program executes `Sort.py`, `Sort.cover` is overwritten. You will need to employ some mechanism to process the counts in `Sort.cover` automatically before the next execution destroys them. This second program should execute all fifteen of the `Sort.py` runs, reading the contents of `Sort.cover` and extracting, summing, and graphing the counts for each function as described. If you pursue this bonus, all your data and the corresponding graphs should be generated by a single execution of your bonus program. You may import and use whatever Python modules you like to manage the data and generate the graphs. You can then use a word processor to present and discuss those graphs.

In addition to your implementation and the optional bonus, you will submit three PDF files for this project. Each file should be a single page that is dedicated to a specific initial orientation of data: increasing, decreasing, or random. For each case, provide a graph using the five array sizes for the x-axis values and instruction counts on the y-axis. The graph should plot two lines: the instruction counts for insertion sort at each size and the instruction counts for selection sort at each size. Below each graph, answer in a single brief paragraph the appropriate set of questions from the list below. Even if you are already familiar with formal analytic presentations such as Big-Oh (which we will cover soon), we ask that you use prose with specific but informal terminology.

1. **Initially increasing data:** What about the insertion sort algorithm (the process, not the code) makes it take fewer steps than selection sort? Why is the line for insertion sort straight (but not flat) instead of curved like the line for selection sort?
2. **Initially decreasing data:** Why are these lines shaped more similarly than they are in the initially increasing case? What about this input case has such a dramatic impact on insertion sort's performance?
3. **Initially random data:** Why does insertion sort take fewer steps than selection sort for this case? Include an explanation of why the approaches differ by the amount they do.

*(see next page)*

## Submission Expectations

All files submitted via Gradescope

1. **Sort.py**: A file containing the two sorting functions and the main section that we provided. Be certain that your functions are named exactly as provided in the skeleton file, including capitalization and underscores. Please include no personally identifying marks in the file.
2. **Bonus.py**: Optionally, include your program to generate the counts and graphs automatically.
3. **Increasing.pdf**: A one-page document containing one graph and one paragraph that explains why the plots in the graph differ or are similar when the **data are initially in increasing order**. The graph should contain two plots, one for insertion sort and one for selection sort. Each point on the plots should represent the number of instructions executed for the relevant sorting function. Your prose response should be brief, informal, and complete; a student from outside of our discipline should be able to read your document and have a reasonable understanding of the differences between the two algorithms and the performance impacts caused by those differences.
4. **Decreasing.pdf**: A one-page document containing one graph and one paragraph that explains why the plots in the graph differ or are similar when the **data are initially in decreasing order**. The graph should contain two plots, one for insertion sort and one for selection sort. Each point on the plots should represent the number of instructions executed for the relevant sorting function. Your prose response should be brief, informal, and complete; a student from outside of our discipline should be able to read your document and have a reasonable understanding of the differences between the two algorithms and the performance impacts caused by those differences.
5. **Random.pdf**: A one-page document containing one graph and one paragraph that explains why the plots in the graph differ or are similar when the **data are initially in random order**. The graph should contain two plots, one for insertion sort and one for selection sort. Each point on the plots should represent the number of instructions executed for the relevant sorting function. Your prose response should be brief, informal, and complete; a student from outside of our discipline should be able to read your document and have a reasonable understanding of the differences between the two algorithms and the performance impacts caused by those differences.

As with anything you submit at William & Mary, you should be proud of these documents as a representation of your capabilities.