

Name: Yangchen Ye (yye02)
Assignment: Project 1: A Battle of Sorts
Date Submitted: Saturday, February 2, 2019 12:57:07 PM EST
Current Grade: Needs Grading

Submission Field:
There is no student submission text data for this assignment.

Comments:
There are no student comments for this assignment.

Files:
 Original filename: Writeup.pdf
 Filename: Project 1 A Battle of Sorts_yye02_attempt_2019-02-02-12-57-07_Writeup.pdf


 Original filename: Sort.py
 Filename: Project 1 A Battle of Sorts_yye02_attempt_2019-02-02-12-57-07_Sort.py


CSCI 241 Data Structures


Project 1: A Battle of Sorts


Your submission for project 1 will be graded according to the following rubric. You will receive a score in each category. Your total project score will be the sum of all category scores. Categories and their weights may change with each project.

	Exemplary 100%	Good 90%	Satisfactory 75%	Marginal 60%	Unacceptable 0%
Adherence to Specifications 20%	There are no deviations from the specifications . Items are named correctly. All files are present in the submission. The implementation and writeup both reflect a deep understanding of the project's purpose and how the specifications reinforce the relevant material.	Most components of the submission adhere to specifications . Some issues remain. Examples may include spelling or case of specified names or other editorially minor issues.	Many components of the submission conform to specifications , but key elements are incorrect. Examples may include incorrect performance measurements, algorithmic deviations from what we cover in class, or use of built-in Python features instead of custom implementations.	The submission attempts to follow the instructions, but fundamental issues persist. There is a clear misunderstanding as to the specifications and how they relate to the material at hand.	The submission may be substantially incomplete. It may be missing specified components. It may implement items in ways other than specified or covered. The submission conveys either illiteracy or ignorance of the specifications.
Your Score 20 / 20	<div> <div></div> <div></div> <div></div> <div></div> <div></div> </div>				

	Exemplary 100%	Good 90%	Satisfactory 75%	Marginal 60%	Unacceptable 0%
Functionality 30%	The submission passes all test cases. Output is formatted correctly. There is no unspecified output. Main code is conditioned to be separate from class or function definitions.	The submission produces the correct result, but minor issues such as formatting.	There is evidence that the implementation does not consider problematic cases or special cases. The program may crash on a few test cases.	The implementation is fundamentally incorrect. Multiple cases fail, or the program crashes in multiple scenarios. The submission fails to convey a sufficient understanding of the material.	The program crashes on most inputs. It contains errors that imply it could not have run during development. Of the test cases that do not crash, many fail with incorrect results.
Your Score 30 /30					

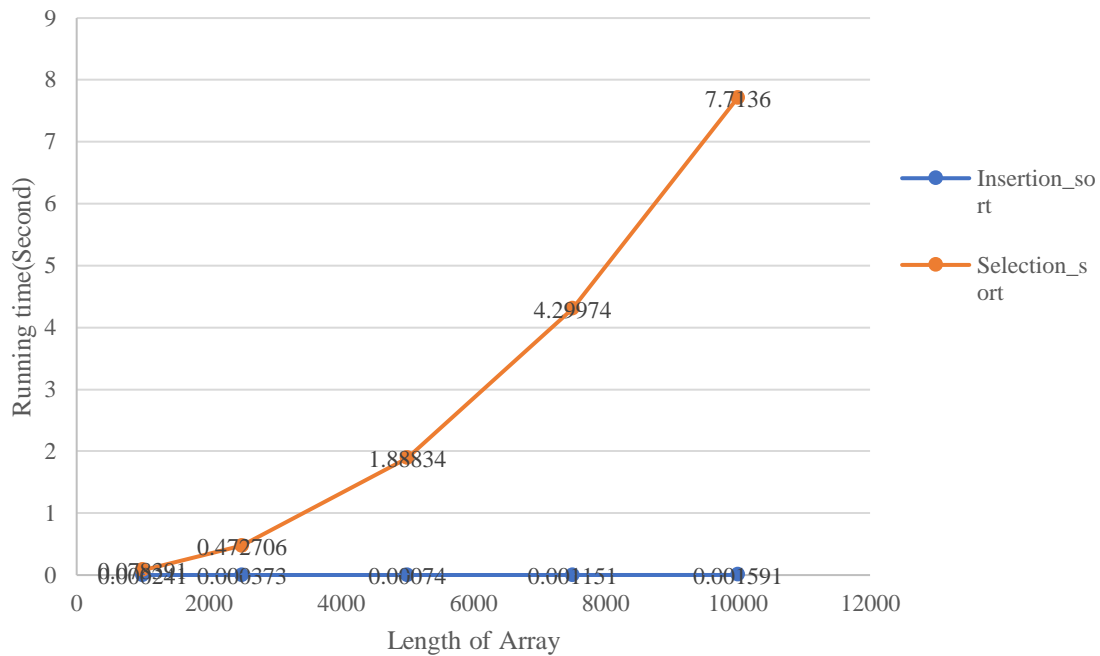
	Exemplary 100%	Good 90%	Satisfactory 75%	Marginal 60%	Unacceptable 0%
Performance 10%	The implementation takes advantage of all opportunities to improve performance.	The implementation is generally efficient, but some steps could be taken to improve performance.	The implementation misses many opportunities to improve performance. Steps are taken without regard to their cost.	The implementation is naïve with respect to performance. Some sections of code take significantly longer than the algorithms described in class.	The implementation takes so long to run as to interfere with grading. It contains unnecessary loops or repetition of steps. If sections of code perform at all, they are extremely inefficient.
Your Score 10 / 10					

	Exemplary 100%	Good 90%	Satisfactory 75%	Marginal 60%	Unacceptable 0%
Readability 10%	Variable, function, and class names are descriptive. Comments explain sections of code where necessary. Indentation is consistent and deliberate.	Code is well-structured, but names could be more descriptive. Some sections of code are not obvious and could benefit from additional comments.	There is repeated code that should be moved to functions. Names are not reflective of what they reference. Inconsistent white space. The implementation contains unnecessary special cases.	The code uses inappropriate features for the problem at hand. Complex implementations are not commented. Names are poorly chosen. Repeated code degrades reliability. Functions return values inappropriately.	The implementation conveys a fundamental misunderstanding of basic programming structures, regardless of language. Loops and conditionals are inappropriately interchanged. Incorrect variables are used for computation. Sections of code are unreachable
Your Score 10 /10					

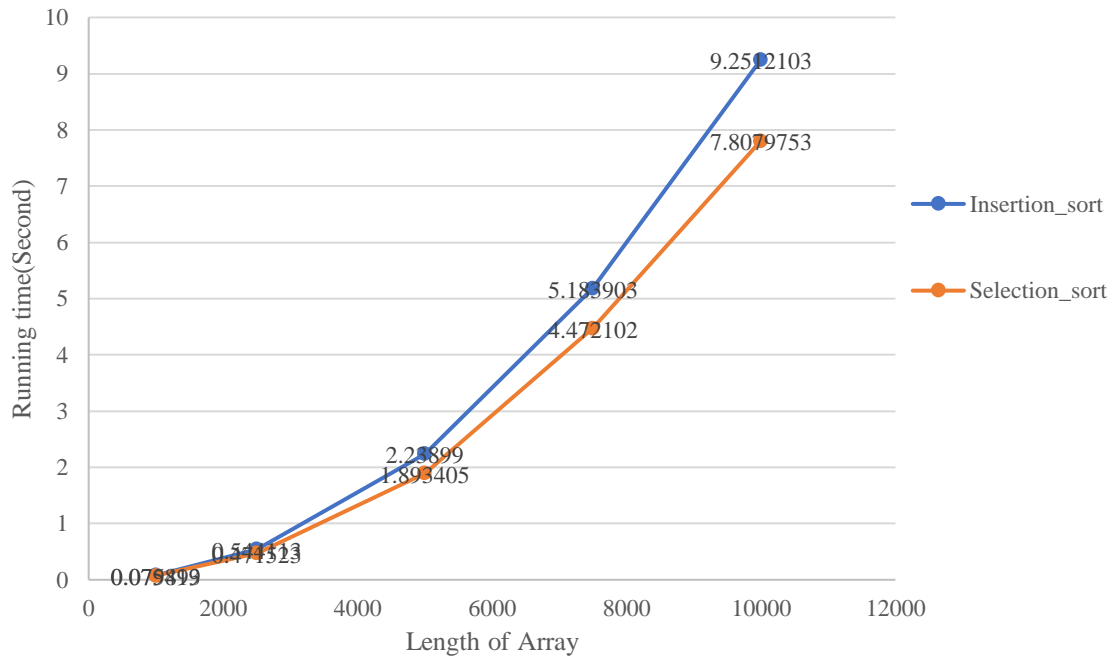
	Exemplary 100%	Good 90%	Satisfactory 75%	Marginal 60%	Unacceptable 0%
Writeup 30%	The writeup indicates a deep understanding of the material at hand. It employs illustrations, prose explanations, and examples to justify claims. It describes in detail the algorithmic steps that lead to performance variations (or for projects 2 and later, formal performance analysis). The document contains few if any grammatical errors.	The writeup is content-complete and generally correct, but presentation is lacking. It may fail to justify some claims with illustrations or examples. It indicates that the general concepts are clearly understood, but minor improvements in presentation would benefit the reader.	The writeup covers the specified material, but in a minimal way. It draws some incorrect conclusions, or offers incorrect or incomplete justifications. Minimal effort to distinguish this submission among others is present.	The writeup is not structured as a prose document, or contains only cursory responses to the required items. The writeup appears to have been an afterthought of the implementation instead of an integral component of the project. No effort to distinguish this submission among others is present.	The writeup is missing or fails to address the required items. If it is present, it is so brief as to indicate a lack of sufficient time, analysis, and construction of rational argument.
Your Score 30/30					

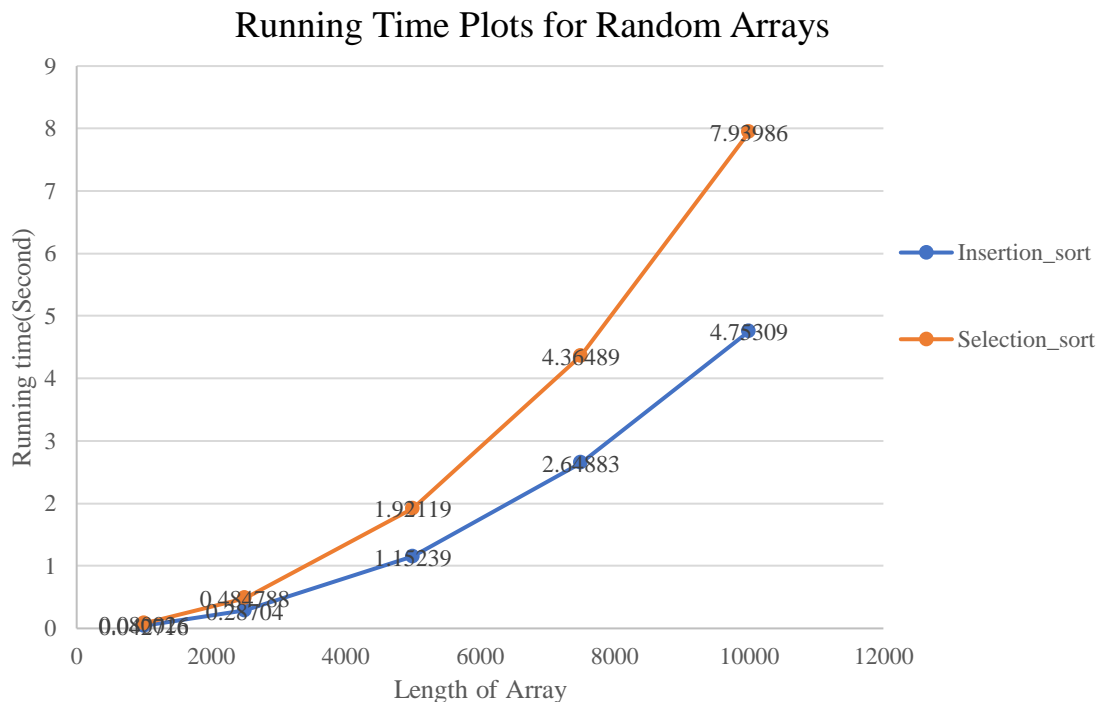
A bit more fine grained y axis for the increasing array runtimes would be better to see what exactly is happening with the Insert sort. Also your placement of the graphs makes it harder for the reader since they need to keep referring back to the graph when referenced in your writeup. It would be better to have the discussion of the graph next to the actual graph.

Running Time Plots for Increasing Arrays



Running Time Plots for Decreasing Arrays





Prose explanation:

It can first be observed that the performance of selection sort is roughly identical in the three types of input, which I think is because no matter what the input is like, selection sort always goes through the same process. The *while* loop inside the *for* loop always go through the whole sub-array after the index k in the for loop, making up for $\frac{n^2-n}{2}$ loops, and inside every turn of the *while* loop the *if* statement is executed regardless of whether the *minimalindex* is updated, which makes up for a total $\frac{n^2-n}{2}$ compares. After each *while* loop ends, there's always one movement of values, swapping the value in index k and index *minimalindex*, making up for a total n movement of values. Therefore, the execution of selection sort is steady, comprised of $\frac{n^2-n}{2}$ compares and n movement.

Excellent analysis and discussion for why selection sort is so consistent.

While the performance of selection sort is roughly the same, insertion sort shows obviously different efficiency in dealing with different types of input. This is because the *while* loop in the insertion sort function is executed only if $arr[j-1] > cur$, which is dependent on the feature of the input.

In the first graph where the input array is already sorted, the *while* loop in the insertion sort is never executed because every $arr[j] \leq cur$, so the movement of value inside the body of the *while* loop is never executed. In this case, the only work that insertion sort does is looping through the *for* loop and assign the value of $arr[j]$ to itself, which is very fast because the *cur* and $arr[j]$ are already in alias in this case. Therefore, when the input is

increasing already, insertion sort does n loops through the array and n compares and n movements, while selection sort does $\frac{n^2-n}{2}$ loops inside which it executes $\frac{n^2-n}{2}$ compares,

and n movement of values. Insertion sort does significantly less work than selection sort, which explains why in the first graph insertion sort is drastically faster than selection sort.

In the second graph where the input array is decreased, insertion sort performs slightly less efficient than selection sort. In this type of input, each $arr[j-1] > cur$ is satisfied in each while loop, so the *while* loop is executed a total $\frac{n^2-n}{2}$ times, with a total $\frac{n^2-n}{2}$ compares. Further, in each loop, there's a movement of values from $arr[j-1]$ to $arr[j]$, and one movement in each *for* loop, making up for a total $\frac{n^2-n}{2} + n$ movements. Therefore, in the second case, insertion sort does the same time of loops and compares as selection sort but does more movements of values than selection sort. This explains the difference of two sorts functions in the second case.

Last, in the third graph where the input array is random, insertion sort performs more efficiently than selection sort to a moderate degree. The reason for this, I think, lies in that random array is the intermediate state between increasing and decreasing. In a random array, it is very likely that some sections of it is already increasing so that the *while* loop in the insertion sort need not be executed. Thus, overall the insertion sort will probably do less times of *while* loops, less times of compare accordingly, but more times of movements of values, because anyway the insertion sort will do some *while* loops and do some more movements of values besides the n movements in the *for* loop. The reason why insertion sort performs better in the random input case is probably that moving values of *int* type is much cheaper and faster than comparing two values of *int* type. This can be confirmed by comparing the pattern in the first and second graph: when the times of compares drops, the efficiency increases dramatically, while when the times of movements increases to roughly the same proportion, the decrease of efficiency is relatively small. This also implies that when moving values is much more expensive than comparing values, selection sort will probably be faster than insertion sort.

To sum up:

1. Increasing:

Insertion sort: n loops, n compares, n movements

Selection sort: $\frac{n^2-n}{2}$ loops, $\frac{n^2-n}{2}$ compares, n movements

Result: Insertion sort significantly faster than selection sort

2. Decreasing:

Insertion sort: $\frac{n^2-n}{2}$ loops, $\frac{n^2-n}{2}$ compares, $n + \frac{n^2-n}{2}$ movements

Selection sort: $\frac{n^2-n}{2}$ loops, $\frac{n^2-n}{2}$ compares, n movements

Well done with with this discussion of why insertion sort performs faster on random arrays than selection sort.

Superb job with this discussion for why insertion sort runs slower than selection sort. Your in-depth analysis and exploration for why this is the case was very well done and explained.

Nice job summarizing
your results

Result: Insertion sort slightly slower than selection sort

3. Random:

Insertion sort: less than $\frac{n^2-n}{2}$ loops, less than $\frac{n^2-n}{2}$ compares, less than $n + \frac{n^2-n}{2}$,
more than n movements

Selection sort: $\frac{n^2-n}{2}$ loops, $\frac{n^2-n}{2}$ compares, n movements

Result: Insertion sort moderately faster than selection sort

Overall, great job with this writeup. One thing to keep in mind is how you display your graphs, such as in your increasing graph it is unclear what exactly insert sort is doing and having the graph zoomed in more would help clear up that confusion. Aside from that keep up the great work!

```

import time
import random

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

def selection_sort(arr):
    for k in range(len(arr)):
        minimalindex = k
        j = k+1
        while j < len(arr):
            if arr[j] < arr[minimalindex]:
                minimalindex = j
            j = j+1
        temp = arr[k]
        arr[k] = arr[minimalindex]
        arr[minimalindex] = temp

if __name__ == '__main__':
    array_insertion = []; array_selection = []
    arraylength = [1000, 2500, 5000, 7500, 10000]
    for k in arraylength:
        array_insertion = []; array_selection = []
        for j in range(k):
            array_insertion.append(j)
            array_selection.append(j)
        start = time.clock()
        insertion_sort(array_insertion)
        end = time.clock()
        print(str(k) + ' ' + 'increasing' + ' ' + "insertion" +
' {:.20f}'.format(end-start))
        start = time.clock()
        selection_sort(array_selection)
        end = time.clock()
        print(str(k) + ' ' + 'increasing' + ' ' + "selection" +
' {:.20f}'.format(end-start))
        array_insertion = []; array_selection = []
        for j in range(k, 0, -1):
            array_insertion.append(j)
            array_selection.append(j)

```

```

start = time.clock()
insertion_sort(array_insertion)
end = time.clock()
print(str(k) + ' ' + 'decreasing' + ' ' + "insertion" +
'{:.20f}'.format(end-start))
start = time.clock()
selection_sort(array_selection)
end = time.clock()
print(str(k) + ' ' + 'decreasing' + ' ' + "selection" +
'{:.20f}'.format(end-start))
array_insertion = []; array_selection = []
for j in range(k):
    rand = random.randint(0,99999)
    array_insertion.append(rand)
    array_selection.append(rand)
start = time.clock()
insertion_sort(array_insertion)
end = time.clock()
print(str(k) + ' ' + 'random' + ' ' + "insertion" +
'{:.20f}'.format(end-start))
start = time.clock()
selection_sort(array_selection)
end = time.clock()
print(str(k) + ' ' + 'random' + ' ' + "selection" +
'{:.20f}'.format(end-start))

```

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

Testing elapsed time: 00:03:38.6128622.