Benchmarks for Insertion, Selection, and Quicksort Algorithms

Michael Stewart

Colorado State University Global

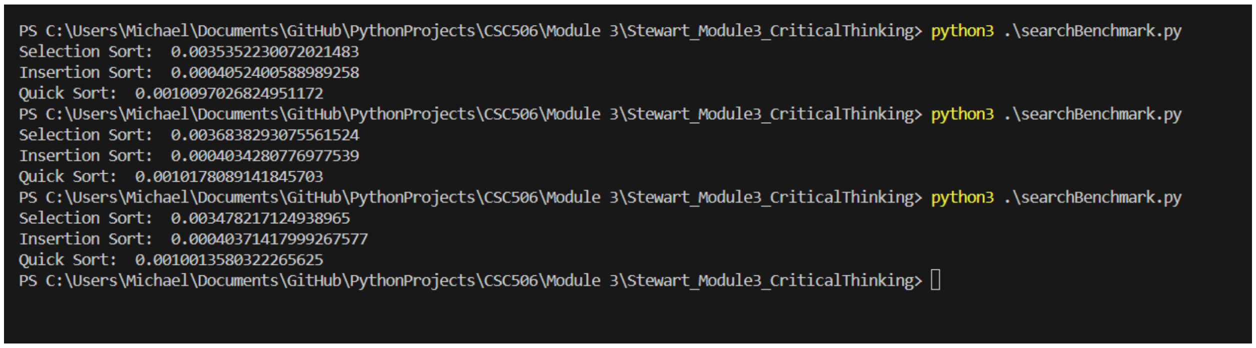
CSC506-1: Design and Analysis of Algorithms

Dr. Isaac K. Gang

10 December 2023

Benchmarks for Insertion, Selection, and Quicksort Algorithms

This week, we were able to define different algorithms and benchmark them using Python. I chose to benchmark the selection sort, insertion sort, and quick sort. I chose the selection sort because it might be close to the simplest sorting algorithm. Going element by element and checking if it is the minimum in the parts of the array that are unsorted is the most intuitive way to sort arrays. Next, I chose the insertion sorting algorithm because it is similar to the selection algorithm, but you are instead searching the sorted part of the array rather than the unsorted part of the array. Finally, I chose to compare the quick sort algorithm since I had an example of benchmarking it for our discussion post.



```
PS C:\Users\Michael\Documents\GitHub\PythonProjects\CSC506\Module 3\Stewart_Module3_CriticalThinking> python3 .\searchBenchmark.py
Selection Sort:  0.0035352230072021483
Insertion Sort:  0.00040524005889892578
Quick Sort:  0.0010097026824951172
PS C:\Users\Michael\Documents\GitHub\PythonProjects\CSC506\Module 3\Stewart_Module3_CriticalThinking> python3 .\searchBenchmark.py
Selection Sort:  0.0036838293075561524
Insertion Sort:  0.0004034280776977539
Quick Sort:  0.0010178089141845703
PS C:\Users\Michael\Documents\GitHub\PythonProjects\CSC506\Module 3\Stewart_Module3_CriticalThinking> python3 .\searchBenchmark.py
Selection Sort:  0.003478217124938965
Insertion Sort:  0.00040371417999267577
Quick Sort:  0.0010013580322265625
PS C:\Users\Michael\Documents\GitHub\PythonProjects\CSC506\Module 3\Stewart_Module3_CriticalThinking> 
```

1. **Insertion Sort: 0.000405ms**
2. **Quick Sort: 0.001009ms A difference of 0.000604 or 60% slower**
3. **Selection Sort: 0.003535ms A difference of 0.002526 or 71% slower**

Selection Sort

Design

While designing the selection sort I realized that it is a couple of simple nested for loops. I went through the whole array and methodically found the minimum value inside the unsorted portion of the array. I had some issues with handling the data where the array would fill itself with zeroes, and I had the wrong way of swapping the key and minimum value in the last line.

Results

The algorithm ended up having around 3.5ms of runtime when sorting an array of 500 integers 0-9. This was slower than all of the other algorithms. This result makes sense because the selection sort, even though the simplest, is not very efficient. Since the best-case scenario of sorting for the insertion algorithm is O(n), it would make sense for the selection algorithm to have a slower time when its best-case scenario is O(n^2).

Insertion Sort

Design

While designing the insertion sort algorithm, I realized that this algorithm is a better version of the selection sorting algorithm. While the insertion sort algorithm uses two for loops, the insertion sort algorithm, however, has a for loop and a while loop. This may seem like a very small change, but this means that the while loop can stop and doesn't have to go through its whole range of iterations depending on how well-sorted the array already is.

Results

With the insertion sort algorithm, it was an interesting surprise to see how fast the insertion sort was. I knew that the insertion sort's best-case time complexity was excellent, but even with an average of randomly generated integers it still was very efficient.

Quick Sort

Design

The design for this was a lot shorter of a time period than the previous ones. I had discussed this previously in a discussion forum and worked through some issues my implementation had. I had some struggles with the ranges I needed to use for the different sections of the array, but eventually got a functional implementation. Some of my classmates and I did some analysis and decided that the partition part of my algorithm could be improved so it is not optimal.

Results

The quick sort ended up in between the selection and insertion sort and the selection sort in terms of speed. I think this is because the quick-sort algorithm is able to be efficient with large arrays, but the best-case scenario is not better than the insertion-sort algorithm. If the partition part of the algorithm was more optimal and the range of random numbers increased then they might get closer together in terms of speed.

Conclusion

In conclusion, it was a good learning experience to learn about these algorithms and practice making them more efficient and seeing the results of different modifications.

References

GeeksforGeeks. (2023b, August 28). *Python program for Insertion Sort*. GeeksforGeeks. https://www.geeksforgeeks.org/python-program-for-insertion-sort/

GeeksforGeeks. (2023d, August 28). *Python program for Selection Sort*. GeeksforGeeks. https://www.geeksforgeeks.org/python-program-for-selection-sort/

GeeksforGeeks. (2023b, August 28). *Python program for Quicksort*. GeeksforGeeks. https://www.geeksforgeeks.org/python-program-for-quicksort/

Lysecky, R. (2019a). Chapter 1. In F. Vahid (Ed.), *Design and Analysis of Algorithms*. essay, Zyante Inc.