Sorting Algorithm Analysis: Quicksort

Advanced Topics in Computer Science

Theo Demetriades

2021-04-3

0. Abstract

The Quicksort algorithm developed for this project performed about as well as would be expected. Since Quicksort has a “Big O” of nlog(n), it should outperform all O(n^2) sorting algorithms when given an array of any significant size. When tested against and compared to two O(n^2) algorithms (Selection Sort and Insertion Sort), the Quicksort outperformed them for not only the largest array sizes, but for *every* size. The two algorithms that Quicksort did not “beat” were the modified “Counting” sort and Python’s built-in `sort()` method. Also included in this paper are two extensions after the primary discussion of the Python Quicksort’s design and performance. First, there is a visual presentation of the Quicksort algorithm using Python’s turtle module. Next, there is optimization. The three goals of the optimization portion of this project were to sort through an array of size one million and range one million in less than one second, to do it faster than the “counting sort” previously mentioned, and finally to do it faster than Python’s built-in `sort()` method. Two of the three goals were reached with the help of Cython. All of the code used for this project can be found on GitHub at https://github.com/tdeme/sorting_algorithm_analysis.

1. Introduction

Sorting algorithms are among the most fundamental and commonly used algorithms in the field of computer science. There is a wide range of sorting algorithms, all with varying degrees of simplicity and efficiency. This paper compares five sorting algorithms, with a special focus on Quicksort.

First, Quicksort’s general strategy will be outlined, followed by the actual source code for the pure-Python implementation of the Quicksort used throughout this project. Next, Quicksort’s performance will be analyzed, first theoretically with a Big-O analysis, and then experimentally using data collected using Python’s `time` module. The findings of these analyses will also be compared with those of the four other Python sorting algorithms for context. The paper will then shift to the “extensions.” The goal of the first extension is to create an easy-to-understand and accurate visualization of Quicksort, and the goal of the second extension is to create the fastest possible Quicksort. The paper ends with a discussion of the results and what they suggest about when Quicksort should be used (or not) over other sorting algorithms.

2. Introduction to the Quicksort Algorithm

Quicksort is considered to be one of the most efficient sorting algorithms, and is used in the libraries of many popular programming languages such as Java and C++. Quicksort works by choosing a pivot, then reordering the array such that all the elements less than the pivot are on its left, and all the elements to its right are greater (elements equal to the pivot can be placed on either side depending on the implementation). The reordering process is sometimes implemented as a helper function called `partition`. Then, the elements to the left and right are sorted the same way recursively. Note: there are many different strategies of choosing the “pivot.” The implementations used in this project choose the last element of the array as the pivot. When sorting an already-sorted array, choosing the first or last element for the pivot causes the run-time to degrade to the worst case of O(n^2). If one is working with data that may already be sorted, they should consider using the "median of three" strategy for choosing the pivot (choosing the median of the first, middle, and last elements for the pivot), which would prevent the run-time degradation to O(n^2) for already-sorted arrays. The following text-diagram outlines a potential series of steps Quicksort would use to sort an unsorted array:

Original, unsorted array: `[4, 2, 0, 1]`
This array, called `array`, has four elements ranging from `array[0]` to `array[3]`.

```
                  0  1  2  3
Choose the pivot: [4, 2, 0, 1], pivot=1
```
`array[3]` is chosen to be the pivot since it is the last element.

Find the index of the first element greater than the pivot from the left, index will be called `lefti`:

```
 0  1  2  3
[4, 2, 0, 1], pivot=1, lefti=0
```

Find the index of the first element less than *or equal to* than the pivot from the right, index will be called `righti`:

```
 0  1  2  3
[4, 2, 0, 1], pivot=1, lefti=0, righti=2
```

If `lefti < righti`, in other words, if they have not yet crossed, swap the two elements:

```
 0  1  2  3
[0, 2, 4, 1], pivot=1, lefti=0, righti=2
```

Find the next item greater than the pivot from the left:

```
 0  1  2  3
[0, 2, 4, 1], pivot=1, lefti=1, righti=2
```

Find the next item less than the pivot from the right:

```
 0  1  2  3
[0, 2, 4, 1], pivot=1, lefti=1, righti=0
```

Since `lefti` and `righti` have crossed, the partition is complete, and it is time to place the pivot in its place inside the array (at index `lefti`):

```
 0  1  2  3
[0, 1, 2, 4]
```

The list happens to be sorted after one partition, but this is not usually the case. In a realistic scenario, `quicksort(array, start, lefti-1)` and `quicksort(array, lefti+1, stop)` would be called recursively to sort the elements on either side of the pivot in the same way outlined above.

3. Source Code

The following is the pure-Python implementation of Quicksort used in the tests and analysis for this project:

```
def quicksort(array, start, stop):
#First, the base-cases
    if stop-start<1: #The array is sorted if its length is ≤ 1.
        return
    if stop-start==1: #If the array size is 2...
        if array[start]<=array[stop]: #If it is sorted, return
            return
        else: #Else, swap the two elements and return
              array[start], array[stop] = array[stop],
array[start]
            return

    pivot = array[stop] #Define pivot to be the last element

    lefti = start #Initialize lefti to start
    righti = stop-1 #Stop-1 because we don’t want to check pivot

    while lefti<righti: #Until they meet…
         #Find first number greater than pivot from left!
        while array[lefti]<=pivot and lefti<=righti:
            lefti+=1
       #Find first number less than or equal to pivot from right
        while array[righti]>pivot and righti>lefti:
            righti-=1
        if lefti<righti: #Swap!
            array[lefti], array[righti] = array[righti], array[lefti]

    #Swap the pivot with array[lefti] to finish partition
    array[lefti], array[stop] = array[stop], array[lefti]

    #Swap items on either side of the pivot recursively
    quicksort(array, start, lefti-1)
    quicksort(array, lefti+1, stop)
```

Note that this implementation does not use the “median-of-three” strategy to choose the pivot because including it did not seem to improve the average performance on unsorted data (which is what was used for testing throughout this project).

The following, since it is referred to throughout this paper, is the Python source-code for the “counting” sort algorithm which will later be compared with the Quicksort implemented above.

```
def counting_sort(array):
#This algorithm uses a dictionary to count the occurrences of each
int
    new = [] #Initialize the sorted list to be returned
```

```
occurrences = {} #Initialize the dictionary to "count"
for integer in l: #Traverse the list, counting each number
    if integer in occurences.keys():
        occurrences[integer]+=1
    else: #if the key value doesn't yet exist, initialize to 1
        occurrences[integer] = 1
for i in range(len(l)): #Reconstruct the list, now sorted
    if i in occurences.keys():
        for _ in range(occurrences[i]):
            new.append(i)
return new
```

4. Big-O Analysis

Now that the Quicksort algorithm has been outlined both in concept and in Python code, its runtime can be predicted with a Big-O analysis. A more thorough analysis can be found in the first reference on the final page of this paper.

Average case: Quicksort is known to have an average time complexity of O(nlogn). The logn part of the algorithm predictably comes from its recursive nature, splitting the array in two smaller pieces (of roughly equal size on average) with each call of the function, which implies that the partition process is linear. In other words, incrementing `lefti` and `righti` takes only n operations. Therefore, the average time complexity is n operations for the partition times $\log_2 n$ levels of recursion, or nlogn.

Worst case: The worst case complexity occurs when the difference in size between the two "smaller pieces" is as big as possible. If the "bigger piece" is only decreasing its size a couple elements (worst case one) at a time, many more layers of recursion will be necessary than if the two pieces were both of roughly size n/2. Quicksort implementations which choose the first or last element for the pivot are known to degrade to the worst case $O(n^2)$ when sorting already-sorted arrays, because the chosen pivot is always either the greatest or smallest element in the array or subarray to be sorted. When the pivot is a maximum or minimum element, it is the only one left out of the next call of the function, resulting in an approximate runtime of O(n) = n + O(n-1) + O(n-2)... which comes out to an order of n^2. For this reason, as mentioned earlier, other strategies for choosing the pivot have been developed, such as picking the median of the first, middle, and last elements.

Best case: The best case occurs when each pair of recursive calls sorts subarrays of equal size (about n/2). This would result in an approximate runtime of T(n) = n*[T(n/2)+T(n/2)]. When an T(n) equation is defined recursively like this, splitting into two T(n/2), that part of the equation can be expressed as log(n) (can think of it as $\log_2 n$). This means that the best-case time complexity of Quicksort is O(nlogn).

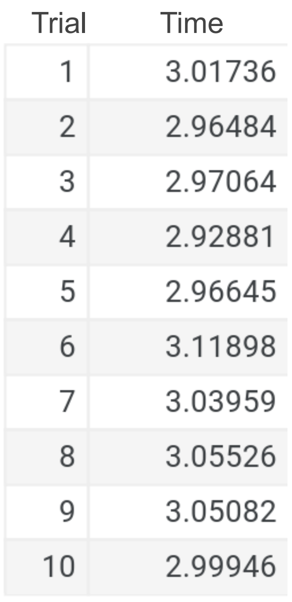
5. Performance Data

We can check whether the algorithm behaves as predicted, O(nlogn), by using Python's `time` module. For this project, the pure-Python implementation of Quicksort was timed for sorting

Python lists of sizes zero to one million. The following is some of the raw data collected when sorting arrays of size zero to nine thousand - ten data points in total with a “step” of one thousand.

Number of Values	Time to Sort (seconds)
0	0
1000	0.001195
2000	0.00268
3000	0.00404
4000	0.00599
5000	0.00842
6000	0.00932
7000	0.01142
8000	0.01299
9000	0.01428

The next table shows the results of ten trials each sorting an array of size one million, and the average result across all trials.



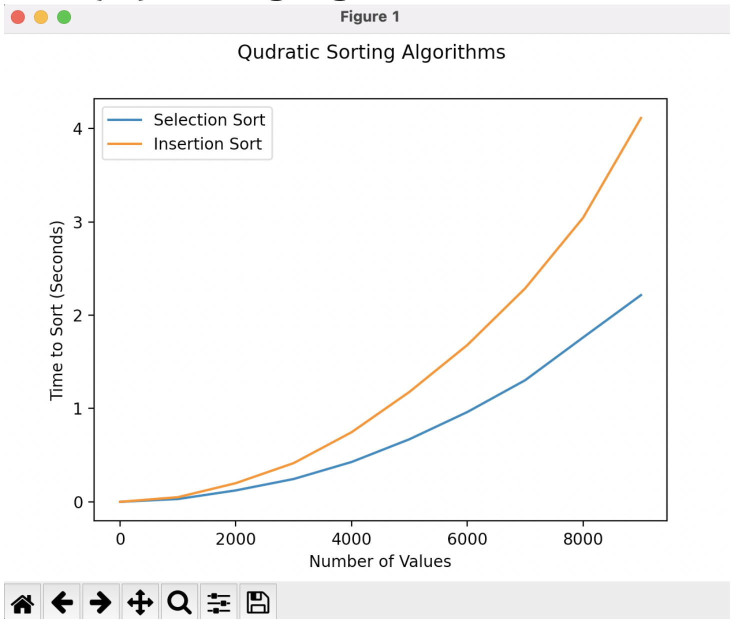
Trial	Time
1	3.01736
2	2.96484
3	2.97064
4	2.92881
5	2.96645
6	3.11898
7	3.03959
8	3.05526
9	3.05082
10	2.99946

Average: 3.01122 seconds

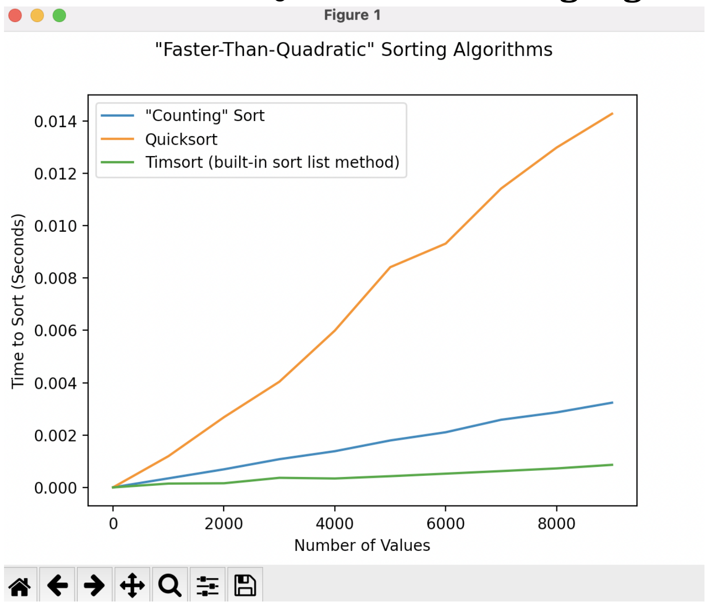
6. Graphical Results

When the data are graphed, it is clear that the performance is less-than-quadratic, which is immediately a good sign. However, to determine whether the performance is actually O(nlogn), some closer analysis may be necessary, since nlogn graphs can sometimes be difficult to tell apart from linear graphs. The following graphs come straight from `comparisons.py`, the program written for this project using the matplotlib.pyplot library to help visualize the performance of several sorting algorithms, including the Quicksort implementation outlined in part 3. The graphs are ordered in ascending efficiency.

6.1 O(n^2) Sorting Algorithms



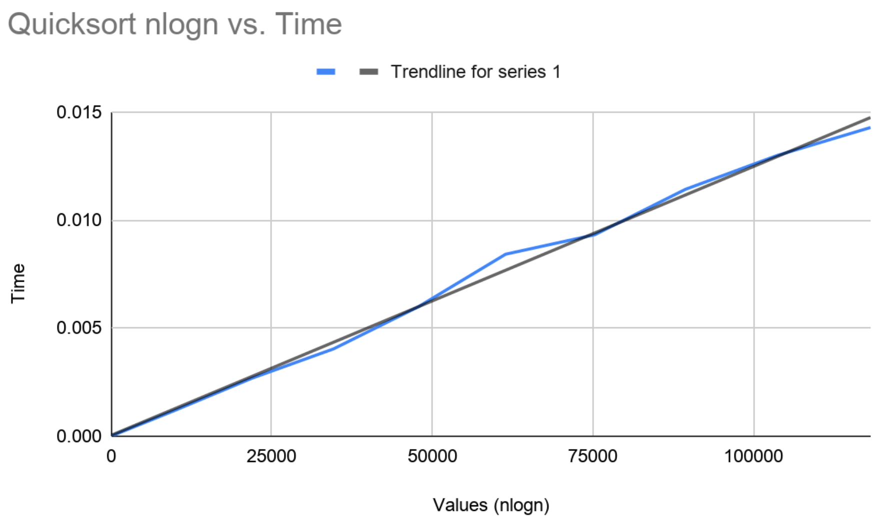
6.2 "Less-Than-Quadratic" Sorting Algorithms



As is made clear by this graph, Python's built-in `sort()` list method is by far the fastest algorithm tested. Interestingly, the simple implementation of the "Counting" sort algorithm also performed quite well.

6.3 Quicksort # of Values (nlogn) vs Time Graph

As mentioned in the introduction to part 6, it can be tricky to tell an O(nlogn) algorithm just by its execution time, or even just by its graph. To help make Quicksort's time complexity more visually obvious, here is a graph of data on the same interval for Quicksort, aided with nlogn scaling on the x-axis as well as a best-fit line. If the Quicksort's performance is really on the order of nlogn, the line should look straight.



The line is nearly straight after all, so the Quicksort implemented does have a performance on the order of nlogn.

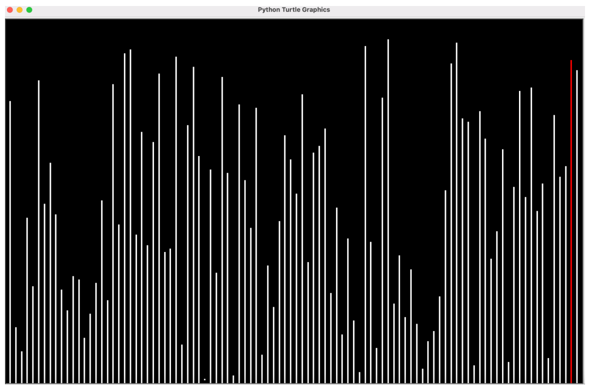
7. Extensions

As mentioned before, there were two main extensions for this project: making a better visual representation of the Quicksort algorithm and making a faster Quicksort. Both goals were a success, however, as always, there is still room for improvement.

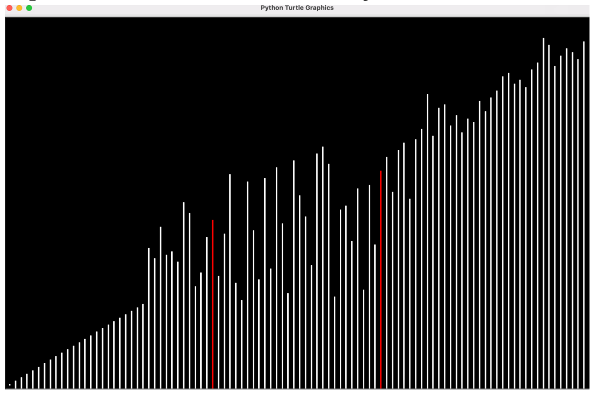
7.1 Visual Representation With `turtle`

For the first extension, the objective was to create a `turtle` graphic that shows the process of sorting through a random list using the Quicksort algorithm. To do this, several lines (`turtle` objects, really) are initialized. The heights of the lines represent values in an array, which must be sorted. In addition to the graphic, the program displays live counters in the terminal for the number of comparisons and swaps used throughout the sorting process. The source code is quite long, so it will not be included in this paper, but it can be found in the project source code on GitHub (link in abstract).

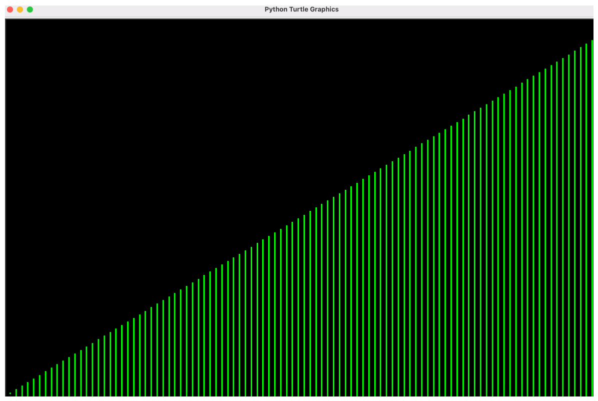
The resulting `turtle` graphic is shown below:



`turtle` objects of random height created to represent the unsorted array.



`turtle` objects being swapped are colored red.



The "array" of `turtle` objects is colored green upon completion.

7.2 Optimization

For the final coding part of this project, there were three goals for an improved Quicksort algorithm. The first goal was for the algorithm to be able to sort through an array of size one million in less than a second, next, to do it faster than the "counting" sort, and finally, to do it faster than the built-in Python method equivalent. The data used for testing in this section is still unsorted, so a median-of-three implementation was not pursued.

The first question that must be asked is: what is limiting the speed of the algorithm? Clearly it performs better than Selection and Insertion Sort, but when compared to Python's built-in `sort()`, the current algorithm is quite slow. So, the next natural question is: what makes Python's built-in methods faster than user-implemented ones? The answer is that the built-in methods are actually referring back to C code.

7.2.1 Surpassing the Language Limitation

It is relatively well-known that Python is a "slow" language, and it turns out that what slows it down is also what made it great in the first place. Python objects, like the list, can often take any type of data. For example, code like `my_list = ['cat', 4, True, [1, 2]]` works just fine, even though each of the list's elements is a different type of data. The tradeoff is that, when the code is interpreted, the object's elements' data types cannot be assumed by the computer. In many other programming languages, this is not the case. For example, in C, the programmer must define data types upon declaration. Sometimes type declaration can be a hassle, but it also has major benefits. One of the biggest benefits happens to be performance, which is why incrementing a variable like `i` in Python is done faster with a `for` loop than manually with a `while` loop - the for loop calls back to C code to do the incrementing!

So how does C apply to this project? As it turns out, there are tools which allow a programmer to use Python for its convenience, while maintaining most of C's speed and efficiency. Python libraries like Numba and PyPy use what are called "just-in-time" compilers (JIT for short). Cython was used for this project. Instead of JITs like Numba and PyPy, Cython can be thought of as a "superset" language of Python. In other words, pure-Python can be run using Cython, but the programmer also has the option of using C-like data-typing to enhance the performance of their code. The result is something much closer to what really happens when a Python function or method is called (such as a `for` loop or the `sort()` method).

7.2.2 Source Code

The following is the source code of the "Cythonized" Quicksort algorithm whose performance will then be analyzed:

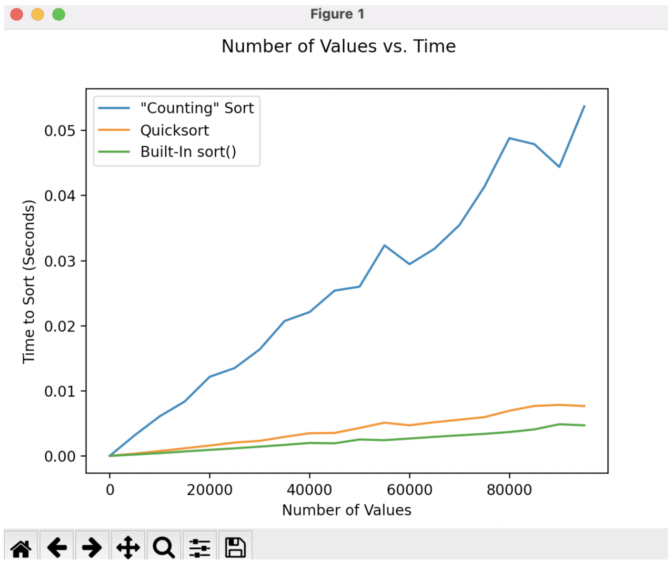
*Please note that the comments in this code will focus on the Cython-specific parts.

```
import numpy as np
cimport numpy as np

#cpdef implies the function is hybrid python and C. To define a pure
C #function, use cdef (use def to define Python functions like
usual).
#(void since the function only edits an existing object)
cpdef void quicksort(np.int64_t[:] array, const int start, const int
stop):
#np.int64[:] is a single-dimensional NumPy array of integers
    if stop-start<1:
        return
    if stop-start==1:
        if array[start]<=array[stop]:
                return
            else:
                array[start], array[stop] = array[stop], array[start]
                return
#C objects can be defined using cdef
    cdef int pivot = array[stop]
    cdef int lefti = start
    cdef int righti = stop-1

    while lefti<righti:
        while array[lefti]<=pivot and lefti<=righti:
            lefti+=1
        while array[righti]>pivot and righti>lefti:
            righti-=1
        if lefti<righti:
            array[lefti], array[righti] = array[righti], array[lefti]

    array[lefti], array[stop] = array[stop], array[lefti]

    quicksort(array, start, lefti-1)
    quicksort(array, lefti+1, stop)
```

7.2.3 Performance

As expected, the Cythonized Quicksort performed much better than the pure-Python implementation, after only the very minor changes outlined above. It is worth keeping in mind, though, that the Big-O does not change - the efficiency gains came from specifying the data types of as many objects and functions as possible, not changes to the actual logic.

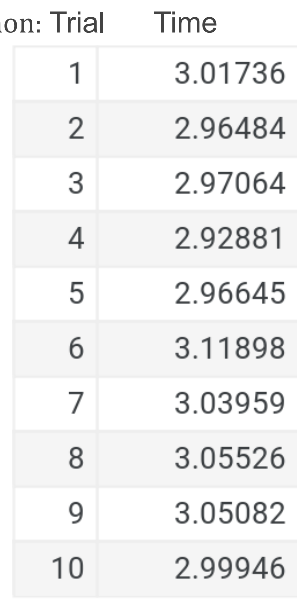
*Please note that all algorithms were "improved" by being given numpy arrays instead of Python lists for the analysis in this section to "even the playing field" as much as possible. The "Counting" Sort was also slightly Cythonized.



Now, some of the weaknesses of the "Counting" Sort become apparent. First, its memory-intensive nature causes inconsistent results depending on array size, and second, its simple implementation left much less room for Cython to improve its performance. On the other hand, the graph shows that the Quicksort's time to sort has decreased from the last graph, even though the array sizes have increased by a factor of ten. The Quicksort is finally performing better than the Counting Sort, but still not as well as the built-in `sort()` method.

The following tables compare the performance of the Cython Quicksort to the pure-Python Quicksort for an array size of one million.

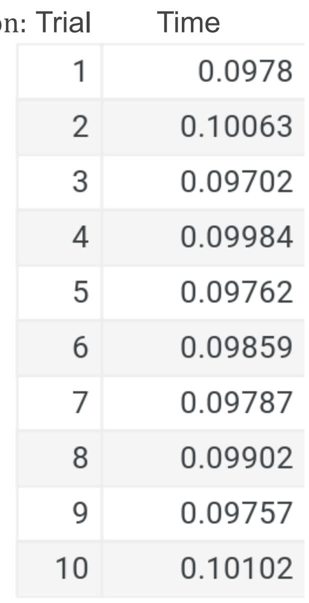
pure-Python:



Trial	Time
1	3.01736
2	2.96484
3	2.97064
4	2.92881
5	2.96645
6	3.11898
7	3.03959
8	3.05526
9	3.05082
10	2.99946

Average: 3.01122 seconds

Cython:



Trial	Time
1	0.0978
2	0.10063
3	0.09702
4	0.09984
5	0.09762
6	0.09859
7	0.09787
8	0.09902
9	0.09757
10	0.10102

Average: 0.09869 seconds

That's about 30x faster!

In the end, the first two of the three goals were reached and surpassed quite easily, but the third goal of "beating" the Python `sort()` method was not achieved. While the Cythonized Quicksort that takes NumPy arrays as input *is* faster than the Python `list.sort()` method, it is not faster than NumPy's `array.sort()`, which is what it should really be compared to. What would it take to build a sorting algorithm faster than NumPy's `array.sort()`? Well, just like Python's `list.sort()`, NumPy's `array.sort()` is written in C, which is almost definitely faster than even Cython-enhanced Python. Note that the average time for NumPy's `array.sort()` to sort through an array of one million integers (with range one million) was on the order of ~0.05 seconds. If the Quicksort were written entirely in C, that might be enough to give it the edge, but it would likely take more than that since the NumPy's sort method is highly optimized for its own data types. Instead, it could be possible to achieve a faster algorithm using Python's `multiprocessing` module, or the Cython tools `nogil`, `parallel`, and `prange`, which are all ways of "parallelizing" code.

In Python, programmers can use the `threading` module to create "threads," but they are not actually executed in parallel because of the "Global Interpreter Lock," or GIL for short. In Cython, using the `with nogil` context manager "releases the GIL," allowing for parallel thread execution. Alternatively, one could parallelize Quicksort by using Python's `multiprocessing` module to create a new `Process` object for each recursive call.

Example code:

```
import multiprocessing
...
process1 = multiprocessing.Process(target=quicksort, args=(array,
start, lefti-1))
process1.start()
process2 = multiprocessing.Process(target=quicksort, args=(array,
lefti+1, stop))
process2.start()
```

Unfortunately, this was attempted but the logic was broken and the bug(s) was/were never found. The goal of using parallel programming to "beat" NumPy's `array.sort()` was ultimately abandoned, but would be an interesting project for the future.

8. Discussion

The results of this project consistently show that the Quicksort algorithm is an efficient way to sort data, and that there are many ways to improve its simplest implementation. The Quicksort made for this project greatly outperformed quadratic sorting algorithms such as Selection Sort and Insertion Sort, and, with some tweaks, could even compete with Python's built-in `sort()` method. However, when using Python, it is probably only useful to implement a Quicksort algorithm for the sake of practice and education, since it is not as fast or convenient as the already-available Python or NumPy `sort()` methods, when using their respective data types. To write a faster sorting algorithm would likely require one or more of the following: using a faster language (such as C) or parallelizing the algorithm using `multiprocessing`, `nogil`, `prange`, or some other similar tool.

In the future, it may be useful to include other sorting algorithms in the comparisons. Hybrid sorting algorithms such as Timsort looked particularly interesting during the research for this project, especially since it turned out to be the fastest Python list-sorting algorithm tested. If one does choose to implement their own Quicksort algorithm to use, there are things they should consider before starting. First, if it is possible that already-sorted data will be passed, they should use the median-of-three version of the algorithm to avoid frequent degradation to $O(n^2)$ performance. Second, if possible, always use NumPy arrays since that is the easiest way to greatly improve performance. Lastly, if further optimization is required, one should consider using Cython to type their information and, if possible, parallelize their code for the specific number of processors in their computer (in which case performance may even pass the built-in methods).

9. Summary

In this paper, Quicksort was introduced as an efficient, yet conceptually simple sorting algorithm which is widely used in the real world. An example outline of how Quicksort might go about sorting an array was shown using a text-based diagram, followed by actual Python code for the implementation used in testing. Next, Quicksort's Big-O was determined analytically, then supported experimentally with the help of Python's `matplotlib` and `time` modules. Quicksort outperformed the O(n^2) sorting algorithms in the preliminary tests, as expected, but was outperformed by both the "Counting" Sort and Python's built-in `list.sort()` method. Next, a better visual representation of the Quicksort algorithm was made using Python's `turtle` module. Finally, the experimental analysis was repeated using improved versions of each sorting algorithm with the help of Cython and NumPy arrays. For these tests, the improved Quicksort outperformed the improved "Counting" Sort, but was still slower than the "default" sorting method, which was the NumPy `array.sort()` in this case. However, the goal of sorting an array of size one million in less than one second was achieved and sorted the data about 30x faster than the original pure-Python implementation. The idea for further improvement by parallelization was described, but not pursued. Finally, the results of the project were discussed, and it was concluded that the default sorting method is usually fastest, unless one implements their efficient sorting algorithm in C (or some other lower-level programming language) or with multiprocessing (or multithreading with nogil), in addition to being the most convenient.

10. References

Wikipedia entry on Quicksort, at https://en.wikipedia.org/wiki/Quicksort.

Helper functions and class for turtle extension, at https://www.crashwhite.com/advtopicscompsci/materials/assignments/activities-projects/project-sorting_algorithm_analysis.pdf.

Python source code for the list object, at https://github.com/python/cpython/blob/master/Objects/listobject.c.

NumPy.sort() documentation, at https://numpy.org/doc/stable/reference/generated/numpy.sort.html.

Cython documentation, at https://cython.readthedocs.io/en/latest/.