Analysis - Sorting: Putting your affairs in order

Deeraj Gurram

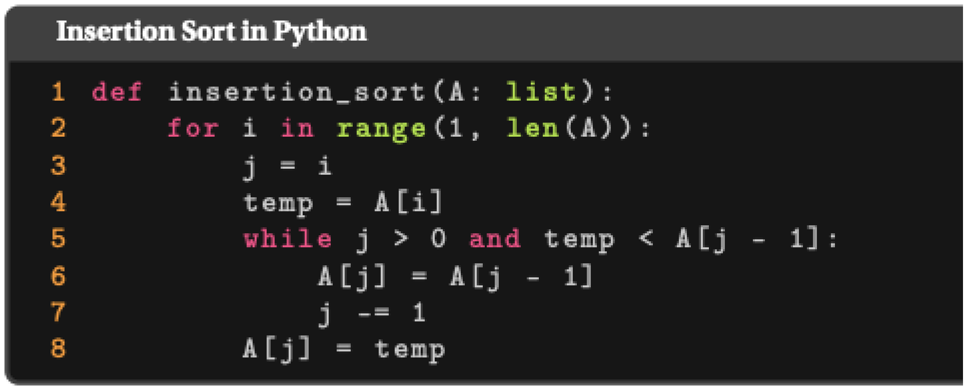
Fall 2021

1 Introduction

In this assignment, we were asked to create 4 different functions that implemented different sorting algorithms. We return the number of moves and compares from each algorithm, and from the results, we can predict which algorithms are best under different circumstances. In this writeup, I'll be examining the results of each function and how these functions compare to each other. Some questions I'll be answering include under which circumstances each algorithm performs best and how each function compares to each other. The rest of this writeup includes the code I based my programs off of, an analysis of the results/graphs, and a conclusion that sums up my findings.

2 Code

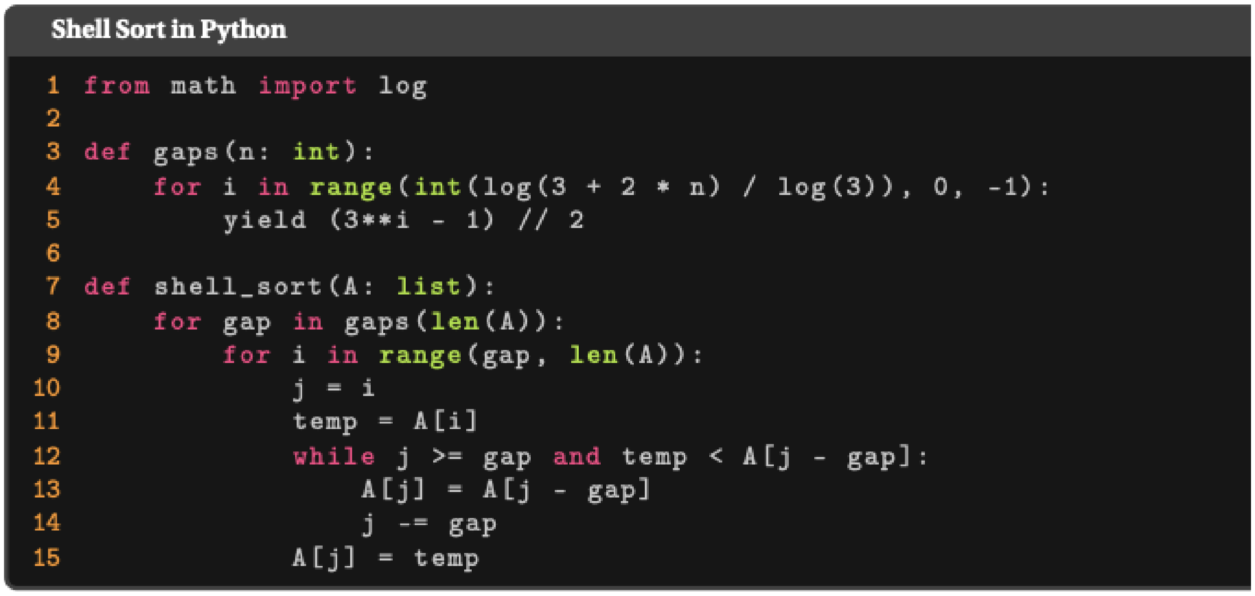
Pseudocode for Insertion Sort algorithm from Dr. Long



Insertion Sort in Python

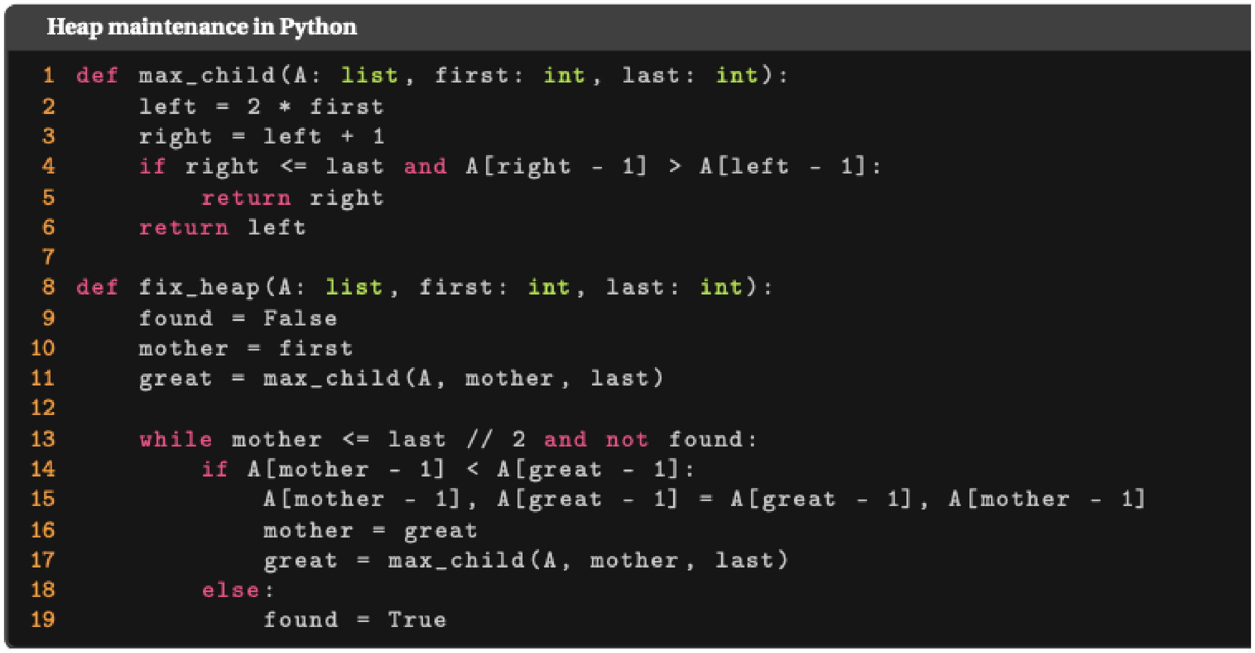
```
def insertion_sort(A: list):
    for i in range(1, len(A)):
        j = i
        temp = A[i]
        while j > 0 and temp < A[j - 1]:
            A[j] = A[j - 1]
            j -= 1
        A[j] = temp
```

Pseudocode for Shell Sort algorithm from Dr. Long

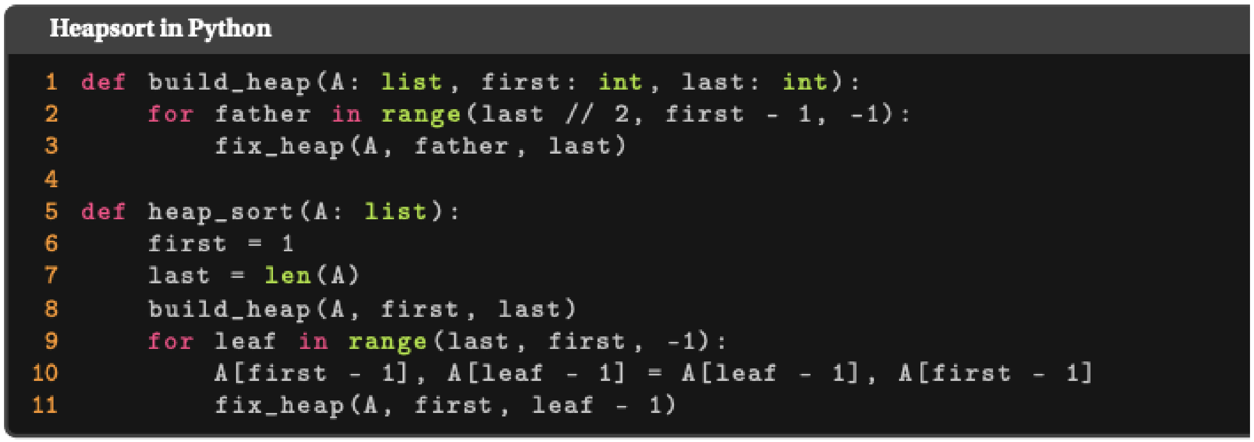

Shell Sort in Python

```python
from math import log

def gaps(n: int):
    for i in range(int(log(3 + 2 * n) / log(3)), 0, -1):
        yield (3**i - 1) // 2

def shell_sort(A: list):
    for gap in gaps(len(A)):
        for i in range(gap, len(A)):
            j = i
            temp = A[i]
            while j >= gap and temp < A[j - gap]:
                A[j] = A[j - gap]
                j -= gap
            A[j] = temp
```

Pseudocode for Heap Sort algorithm from Dr. Long


Heap maintenance in Python

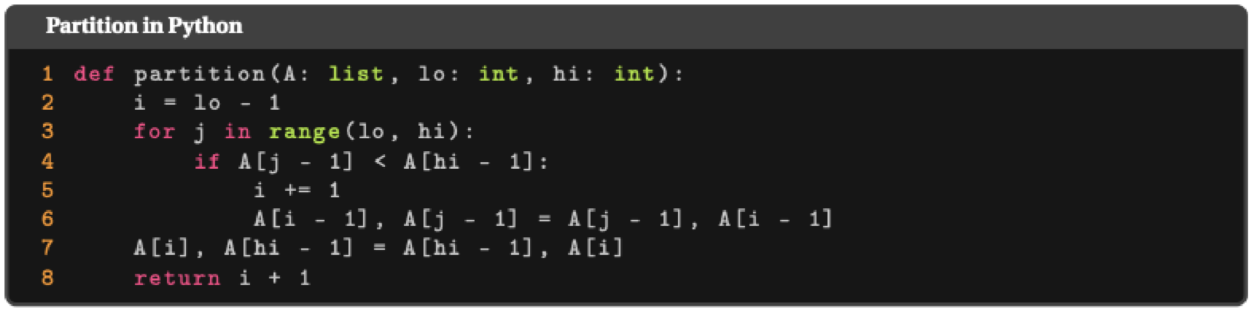
```python
def max_child(A: list, first: int, last: int):
    left = 2 * first
    right = left + 1
    if right <= last and A[right - 1] > A[left - 1]:
        return right
    return left

def fix_heap(A: list, first: int, last: int):
    found = False
    mother = first
    great = max_child(A, mother, last)

    while mother <= last // 2 and not found:
        if A[mother - 1] < A[great - 1]:
            A[mother - 1], A[great - 1] = A[great - 1], A[mother - 1]
            mother = great
            great = max_child(A, mother, last)
        else:
            found = True
```



Heapsort in Python

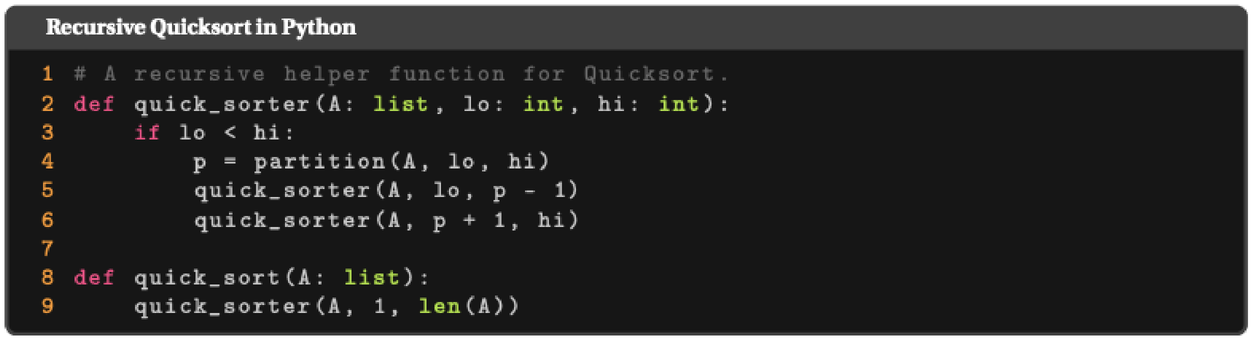
```python
def build_heap(A: list, first: int, last: int):
    for father in range(last // 2, first - 1, -1):
        fix_heap(A, father, last)

def heap_sort(A: list):
    first = 1
    last = len(A)
    build_heap(A, first, last)
    for leaf in range(last, first, -1):
        A[first - 1], A[leaf - 1] = A[leaf - 1], A[first - 1]
        fix_heap(A, first, leaf - 1)
```

Pseudocode for Quick Sort algorithm from Dr. Long



Partition in Python

```python
def partition(A: list, lo: int, hi: int):
    i = lo - 1
    for j in range(lo, hi):
        if A[j - 1] < A[hi - 1]:
            i += 1
            A[i - 1], A[j - 1] = A[j - 1], A[i - 1]
    A[i], A[hi - 1] = A[hi - 1], A[i]
    return i + 1
```

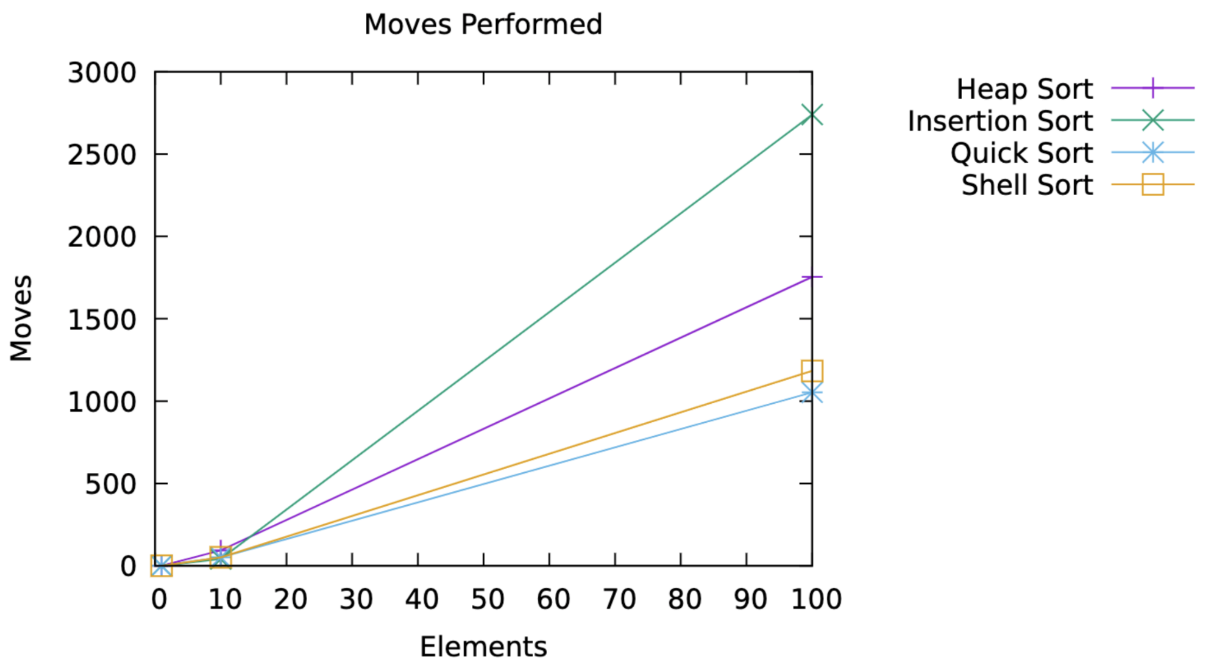


Recursive Quicksort in Python

```python
# A recursive helper function for Quicksort.
def quick_sorter(A: list, lo: int, hi: int):
    if lo < hi:
        p = partition(A, lo, hi)
        quick_sorter(A, lo, p - 1)
        quick_sorter(A, p + 1, hi)

def quick_sort(A: list):
    quick_sorter(A, 1, len(A))
```

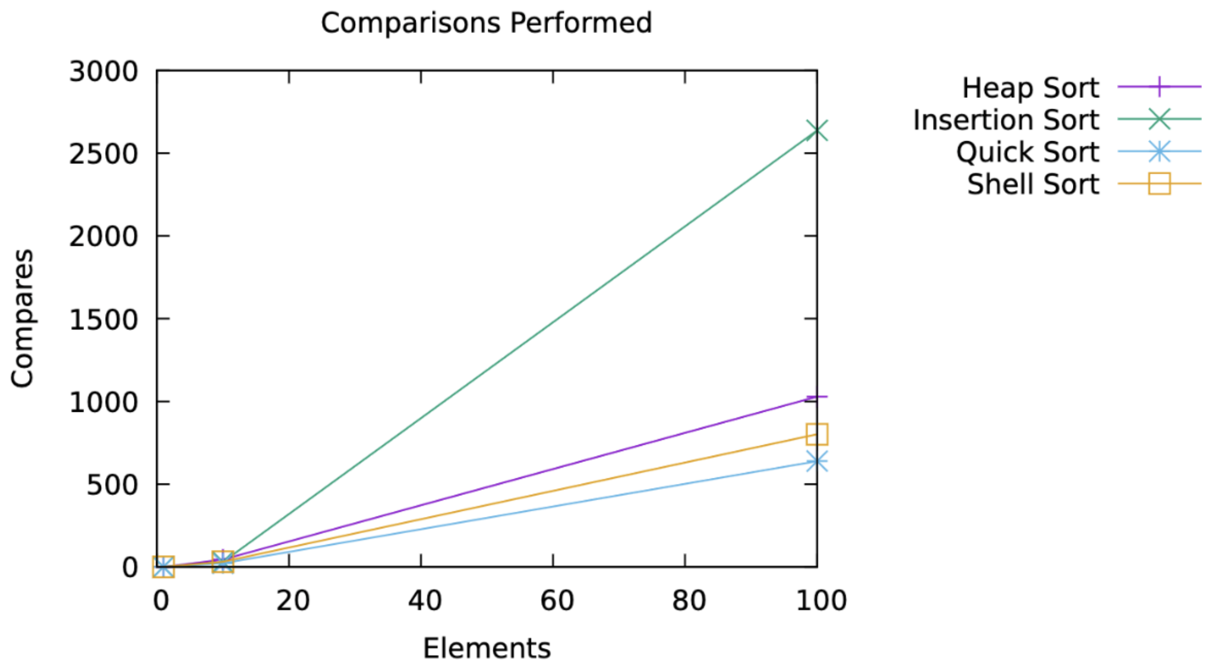
3 Analysis

Moves performed when looping over 100 elements



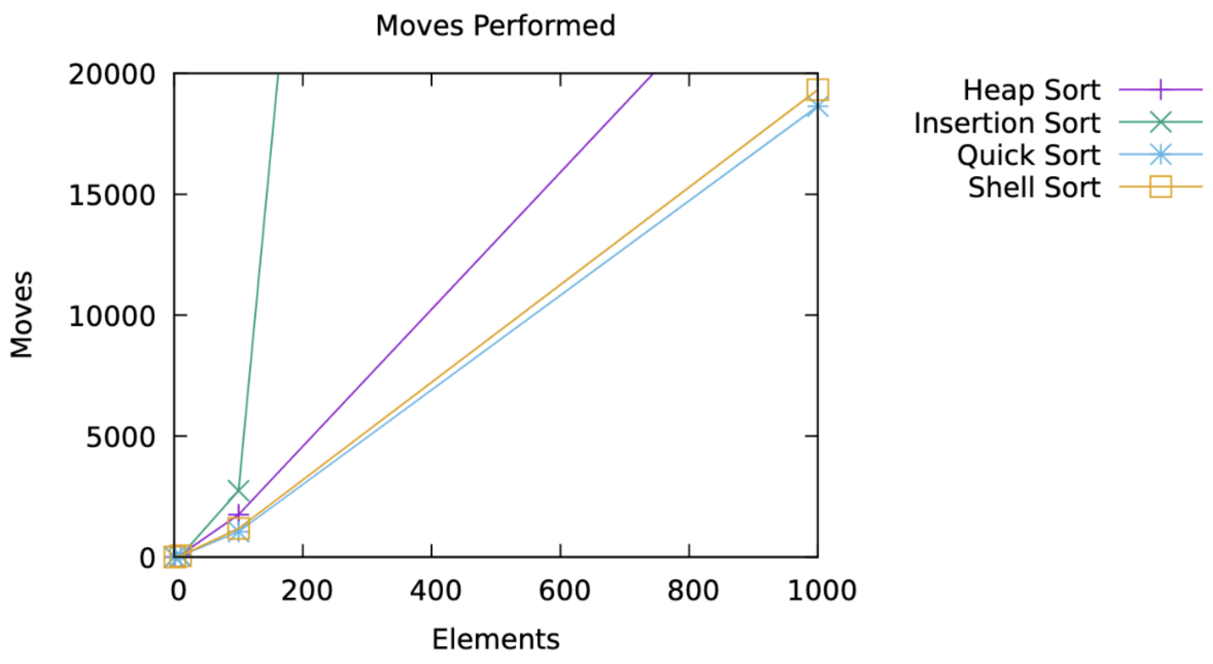
From this graph, we can see that the quicksort algorithm's number of moves computed is the fastest, with shell in second, heap in third, and insertion last. Though the insertion sort algorithm is faster at a length of 10 elements, it is quickly overtaken as the number of elements increases. Similarly, the shell sort algorithm is very close to the quicksort algorithm, though quicksort steadily outpaces shell sort near the end of the elements.

Comparisons performed when looping over 100 elements



The insertion sort algorithm's comparisons are very similar to insertion's number of moves. This occurs because my implementation of the insert.c code increments moves and compares almost at exactly the same time, though the number of moves is higher by around 100 values. Similarly, the shell sort has a comparable number of comparisons to its number of moves though it is separated by around 300 values. However, the following is true as seen in the graph: the quicksort algorithm returns the smallest amount of compares, with shell sort in second, heap sort in third, and insertion sort in last place. This trend will be consistent when looking at any number of elements greater than 20 as seen in the graph.

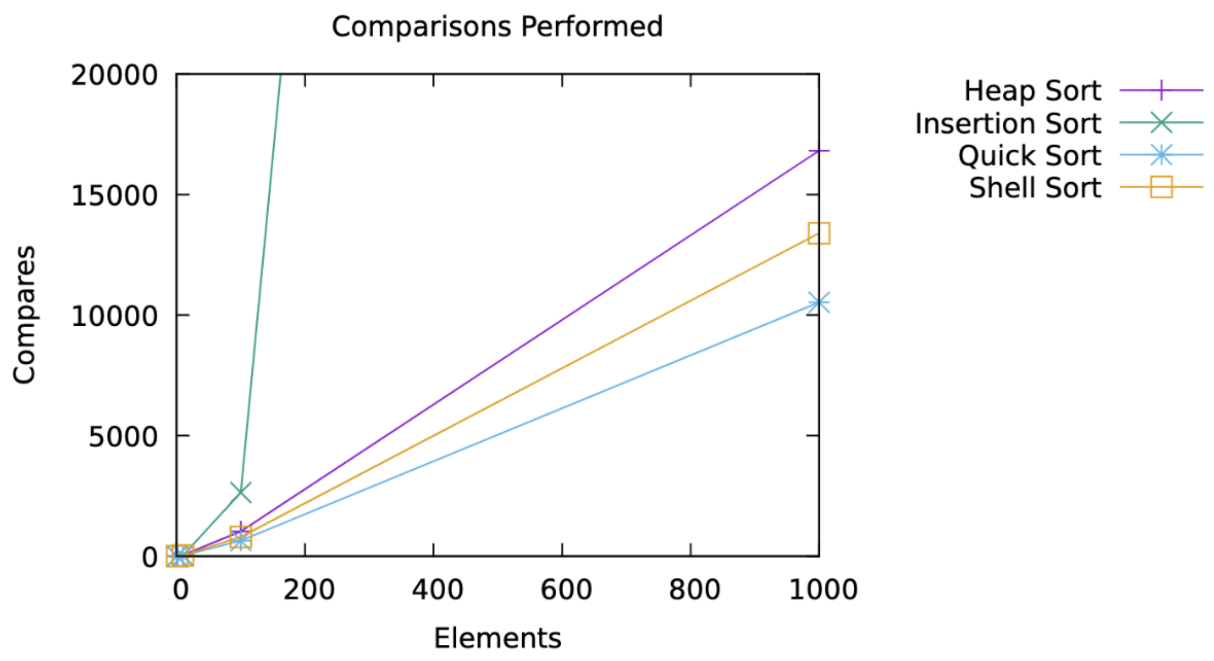
Moves performed when looping over 1000 elements



Note: I limited the y range so the graph could clearly delineate the lines for the other sorting algorithms.

From this graph, we can again see that the quicksort algorithm is the fastest, with shell sort closely following it, heap in a mediocre third, and insert in last place. Quicksort ends with around 16,000 moves which is great when compared to insertion sort's almost 300,000 moves for the same number of elements. From this, we can see that the recursive solution saves much more memory and time when compared to the simple, intuitive solution.

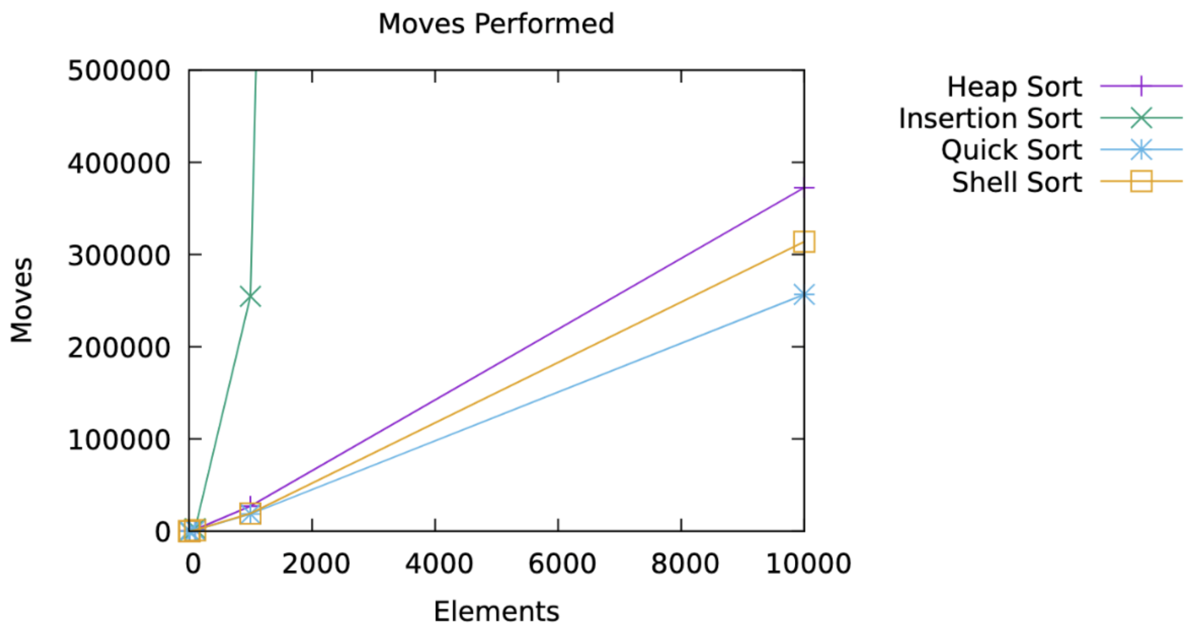
Comparisons performed when looping over 1000 elements



Note: I limited the y range so the graph could clearly delineate the lines for the other sorting algorithms.

The similarity of the Insertion sort's moves and compares is caused by the similarity in the number of moves and compares by the functions implementation. In quicksort, shell sort, and heap sort's case though, the number of compares is lower than the number of moves by almost half due to the algorithm's implementation, and the location at which swap() and cmp() were called.

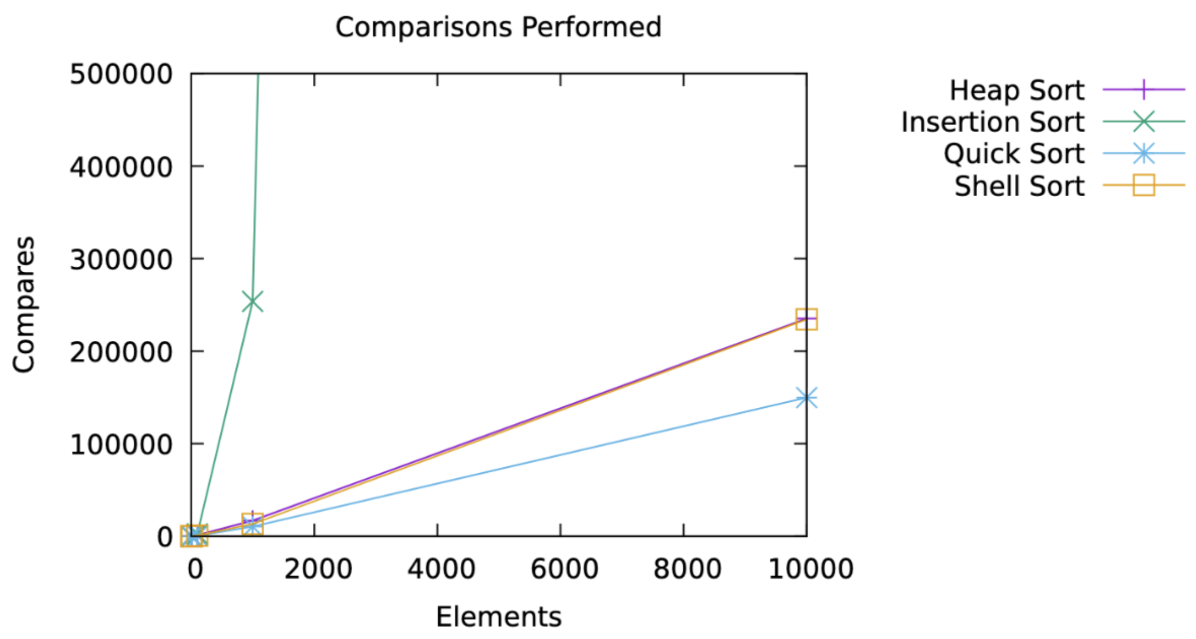
Moves performed when looping over 10,000 elements



Note: I limited the y range so the graph could clearly delineate the lines for the other sorting algorithms.

From this graph we see that the quicksort, shell sort, and heap sort all perform within a similar range, with insertion sort reaching much higher levels. Again, quick sort beats shell sort, and the difference is much more pronounced at a higher number of elements. Though the two algorithms are still close, the quicksort is noticeably faster and uses less moves.

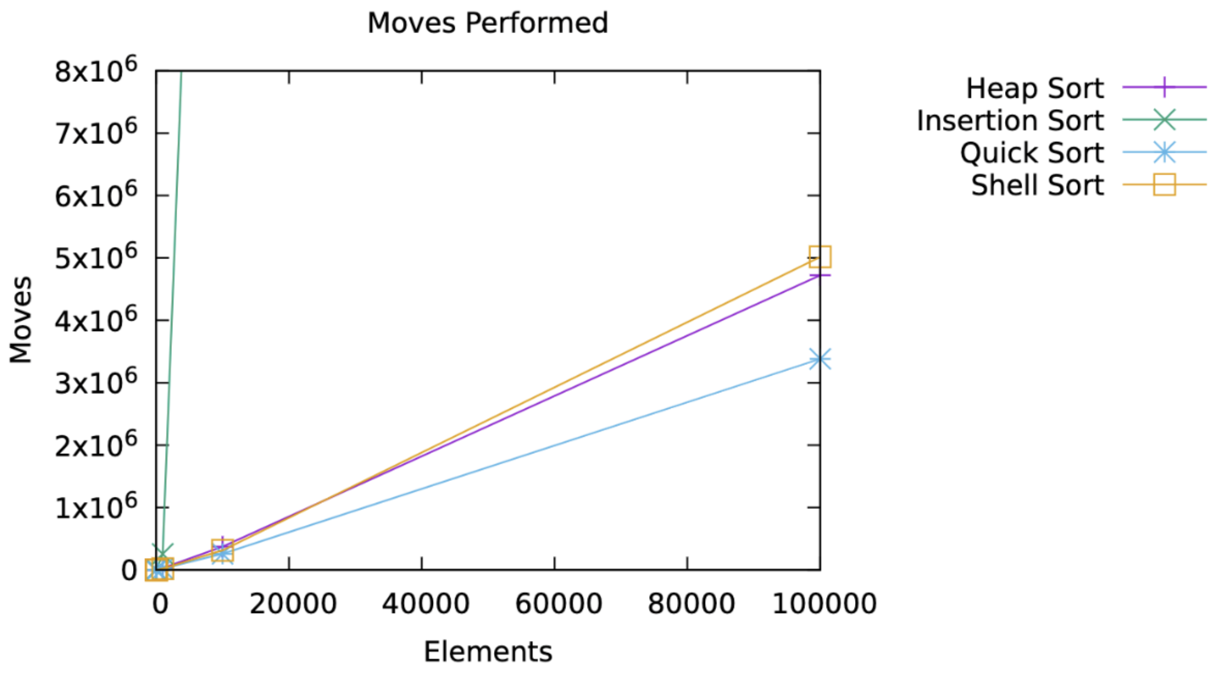
Comparisons performed when looping over 10,000 elements



Note: I limited the y range so the graph could clearly delineate the lines for the other sorting algorithms.

From this graph we see that insertion sort and quicksort are still in their expected places, however the number of compares for heap sort and shell sort are almost interchangeable. While heap sort previously appeared to be less efficient than shell sort, we can now see that at a higher number of elements, heap sort begins to overtake the efficiency of shell sort.

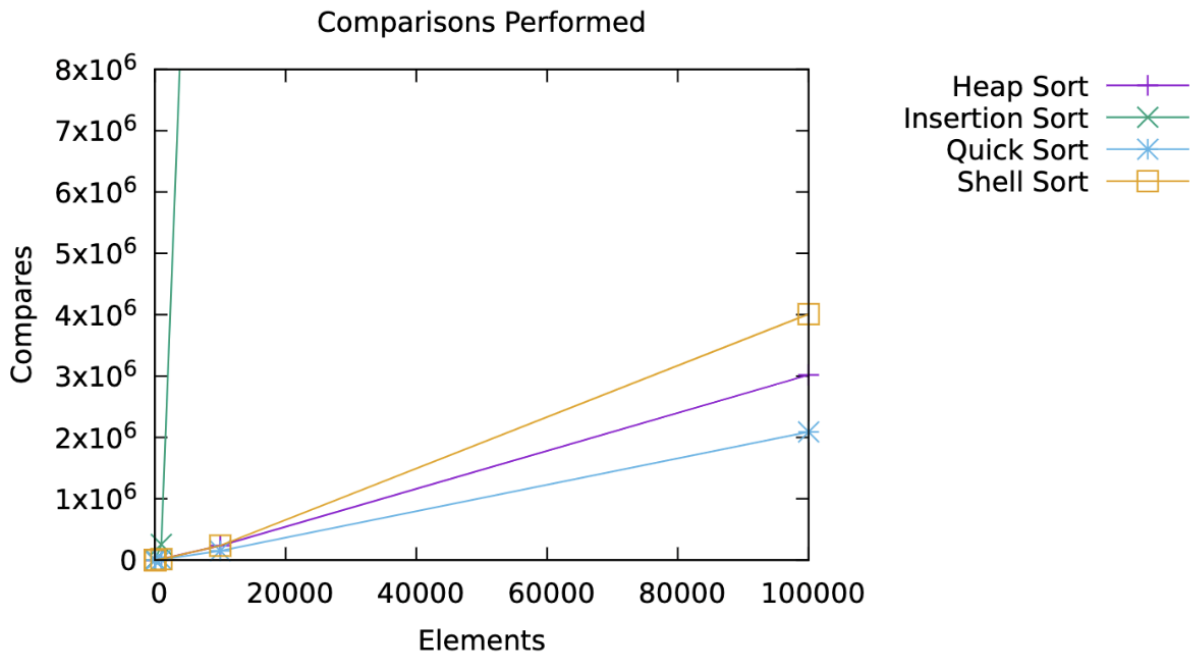
Moves performed when looping over 100,000 elements



Note: I limited the y range so the graph could clearly delineate the lines for the other sorting algorithms.

From this graph, we see that quicksort and insertion sort are still performing as expected while heap sort has now thoroughly overtaken shell sort for second place. If we examine the graph closely, we see that heap and shell sort seem to switch places in efficiency several times throughout their plotting, noticeable moments occur at what appears to be 20,000 elements and close to 33,000 elements. However, after 40,000 elements, it’s clear to see that heap sort is more efficient than shell sort.

Comparisons performed when looping over 100,000 elements



Note: I limited the y range so the graph could clearly delineate the lines for the other sorting algorithms.

In contrast to the extremely close race between shell sort and heapsort in the moves graph for 100,000 elements, we can clearly see heap sort's efficiency overtake shell sort at around 15,000 elements. As we noticed in the comparisons graph for 10,000 elements, heap and shell sort were almost interchangeable. However, we now see that it simply took more elements to clearly delineate the difference in their efficiency. Quicksort and insertion sort are still performing as expected however. Interestingly, in this graph we see that the difference between the efficiency of quicksort and heapsort is similar between heap sort and shell sort at 100,000 elements.

4 Conclusion

From the analysis made on the graphs as well as the known time complexity of the code, we can conclude that at an extremely low number of elements (less than 10 elements) insertion sort is the fastest algorithm. However, at any number of elements above 10, quicksort is clearly the most efficient. Interestingly, the analysis of these graphs have proved that while shell sort is more efficient at less than 10,000 elements, the heap sort algorithm quickly overtakes shell sort and becomes more efficient. This behavior was completely unexpected to me, as I expected the shell sort to stay in second place in terms of its efficiency. However, after analyzing the time complexity of heap sort and shell sort, we see why this is the case. When comparing heap sort's best time complexity of Ω(n log(n)) to shell sort's best time complexity of Ω(n log(n)) in the best case, and O(n^2) in the worst case, it's clear to see why heap sort overtakes shell sort at larger element sizes. Though the two algorithms perform at a similar level in their best cases, when the number of elements increases, shell sort's time complexity becomes O(n^2) which is quickly outperformed by heap sort. When comparing the best and worst performers, we see that the quicksort has a time complexity of Ω(n log(n)) while insertion sort has a complexity of Ω(n) at low elements and O(n^2) at any higher number of elements. As such, we can list the best performances of each sorting algorithm as follows:

- Quick sort is most efficient at > 10 elements
- Insertion sort is most efficient at < 10 elements
- Heap sort is most efficient at > 40,000 elements
- Shell sort is most efficient at < 40,000 elements