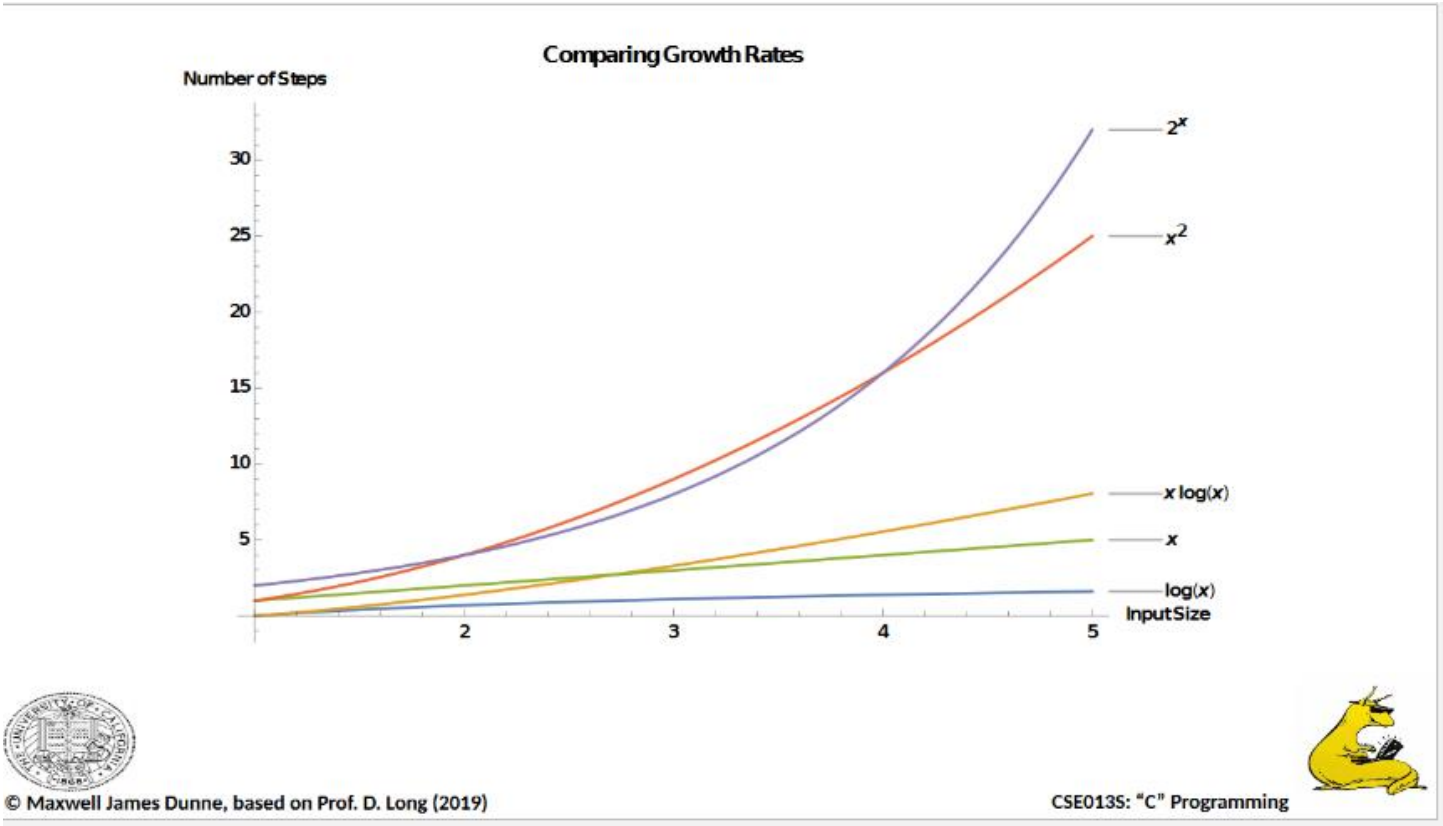
Write-Up
Assignment #5 : Sorting
Despina Patronas

Time Complexity



Bubble Sort:

```
def Bubble_Sort(arr):
  for i in range(len(arr) - 1):
    j = len(arr) - 1
    while j > i:
      if arr[j] < arr[j - 1]:
        arr[j], arr[j - 1] = arr[j - 1], arr[j]
      j -= 1
  return
```

Bubble Sort (pseudocode)

Worst case: **O(x^2)**

In the worst case, bubble sort will have an exponential time complexity due to the nature of the comparisons:
Formula for swaps ~= comparisons = (x(x-1))/2

Bubble sort falls short in that it is one directional and completely dependent on successive passes of the entire Array for the sake of placing only one element in the correct position.

Best Case: **O(x)**

In a perfect world, the bubble sort can achieve a linear time complexity with few or no swaps needed.

Average Case: **O(x^2)**

Even with optimizing the bubble sort to eliminate the sorted elements at the end, the time complexity of bubble sort averages out at O(n^2) making it the worst sorting algorithm to implement.

Space Complexity:

The constant aka overhead when executing sorting algorithms differs. This constant is defined as the space used by the algorithm which is also known as space complexity. Optimizing code can decrease the overhead constant however, constant speedup is overshadowed by the algorithms time complexity.

Space Complexity Note:

Bubble sort may be the worst in terms of time complexity but it has excellent space complexity, since the code is straightforward only needing incremental variables. So the constant is simply 1 for bubble sort.

Shell Sort : **exchanges items in array across a gap**

```
def gap(n):
  while n > 1:
    n = 1 if n <= 2 else 5 * n // 11
    yield n
```

gap (pseudocode)

```
def Shell_Sort(arr):
  for step in gap(len(arr)):
    for i in range(step, len(arr)):
```



```
      for j in range(i, step - 1, -step):
        if arr[j] < arr[j - step]:
          arr[j], arr[j - step] = arr[j - step], arr[j]
  return
```

Shell Sort (pseudocode)

Worst case: between O(x^2) and (x^3/2)

Worst case depends on the gap sequence used. In general shell sort follows insertion sorts footsteps with worst case being Quadratic with the nested looping structures.

Best Case: O(x log x)

In the best case, inner loop will never proc since there will not be a condition to swap. This inner loop becomes a constant 0(1). The middle loop is dependent on the outer loop which is constantly being halved. Therefore the O(x log x) is achievable since comparisons for each increment == size of array

x/2 + x/4 + x/8 + … + x/x consolidates into x (log x)

Average Case: between O(log x) ^2 and O(x^4/3)

Average case will depend on the gap used. Shell sort iterated through a high amount of moves and compares similar to bubble sort.

Space complexity note:

For Shell sort the space complexity is constant at O(1) similar to bubble sort, there is a less demanding overhead involved.

Binary Insertion Sort (derivation of insertion sort)

```
def Binary_Insertion_Sort(arr):
  for i in range(1, len(arr)):
    value = arr[i]
    left = 0
    right = i

    while left < right:
      mid = left + ((right - left) // 2)

      if value >= arr[mid]:
        left = mid + 1
      else:
        right = mid

    for j in range(i, left, -1):

      arr[j - 1], arr[j] = arr[j], arr[j - 1]

  return
```

Worst case: O(x^2)

Insertion binary sort is handicapped by the iteration of its outer loop and the inner loops swaps for insertion to take place. (only swaps elements adjacent to each other). The cost of moves is what slows this algorithm down significantly

Best Case: O(x log x)

Compared to insertion sort, binary insertion sort can do MUCH less compares at (O(log x) compares)
If compares is a huge make up, binary would be preferred over regular insertion sort.
The best case is determined by the constant division by 2 boosting the performance in best case.

Average Case: O(x^2)

In the long run, the high move count in binary insertion cause its average performance to falter.

Space Complexity note:

Binary Insertion sort has little overhead, at a constant of O(1)

Quick Sort:

```
def Partition(arr, left, right):
  pivot = arr[left]
  lo = left + 1
  hi = right

  while True:
    while lo <= hi and arr[hi] >= pivot:
      hi -= 1

    while lo <= hi and arr[lo] <= pivot:
      lo += 1

    if lo <= hi:
      arr[lo], arr[hi] = arr[hi], arr[lo]
    else:
      break

  arr[left], arr[hi] = arr[hi], arr[left]
  return hi


def Quick_Sort(arr, left, right):
  if left < right:
    index = Partition(arr, left, right)
    Quick_Sort(arr, left, index - 1)
    Quick_Sort(arr, index + 1, right)
  return
```

A Less Simple Algorithm

- On Average
  - $O(n \log(n))$
- Worst case
  - $O(n^2)$

```
def Partition(arr, left, right):
  pivot = arr[left]
  lo = left + 1
  hi = right

  while True:
    while lo <= hi and arr[hi] >= pivot:
      hi -= 1

    while lo <= hi and arr[lo] <= pivot:
      lo += 1

    if lo <= hi:
      arr[lo], arr[hi] = arr[hi], arr[lo]
    else:
      break

  arr[left], arr[hi] = arr[hi], arr[left]
  return hi

def Quick_Sort(arr, left, right):
  if left < right:
    index = Partition(arr, left, right)
    Quick_Sort(arr, left, index - 1)
    Quick_Sort(arr, index + 1, right)
  return
```

Worst case: O(x^2)

In the worst case, quick sort is not able to perform well due to continuously bad partition choices. Repeated bad pivots, will repeated divide the array disproportionately and thus lose the divide and conquer advantage which makes this algorithm so strong. To prevent this from happening a random partition choice will yield increased performance. A median pivot will also prove good for time complexity, but increase space complexity significantly as choosing a median is a O(x) operation

Best Case: O(x log x)

As described above, eliminated the effect of bad partitions by either randomizing or selecting median as pivot will cause quick sort to perform at its best.

Average Case: O(x log x)

Easily achievable average case by preventing the problems that cause worst case performance (above)

Space complexity note:

Depending on the implementation, quick sort can average between O(x log x) overhead or higher O(x).

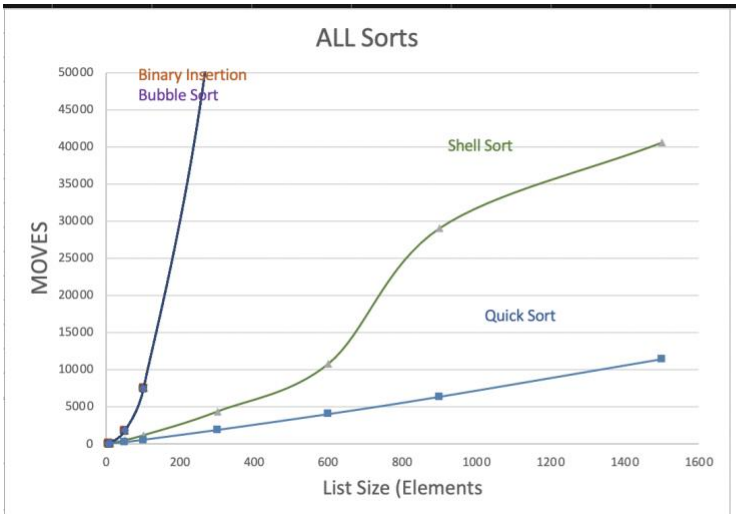
Take-Aways: When a program is required to sort a larger algorithm the time complexity becomes vital. Quick sort is able to consistently achieve both a great average on time complexity while also minimizing the space complexity.

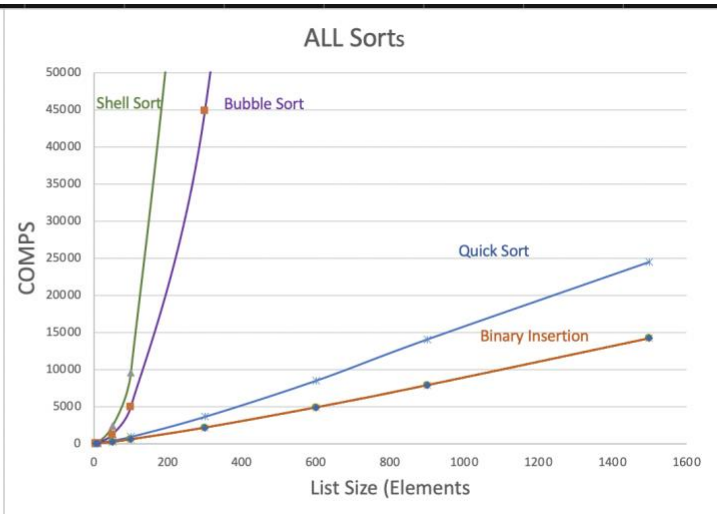
A small array can get away with using a less efficient algorithm with an optimized code for the best constant space complexity and be efficient enough for intended application.

Each sorting algorithms will scale differently based on moves, comparisons, and overhead required to accomplish the task. Choosing the best one will depend on the size of the list to sort and memory available.

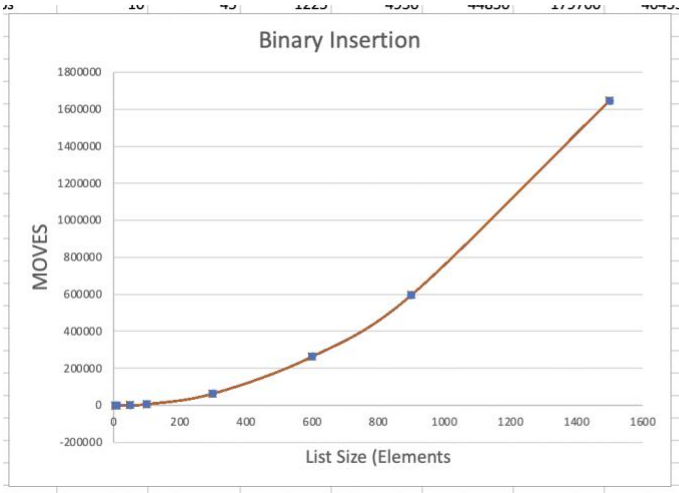
NOTE: Credits to Professor Max James Dunne, Lecture slides 19 and 20 providing the graphs, pseudo code, and information regarding time complexity and sorting algorithm analysis.

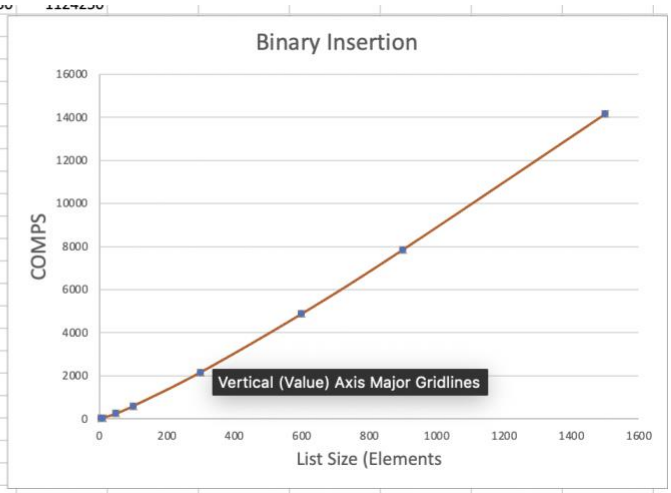
Summary of Data: Here is some graphical data on the numbers from each sort generated by my sorting program at different element length intervals:



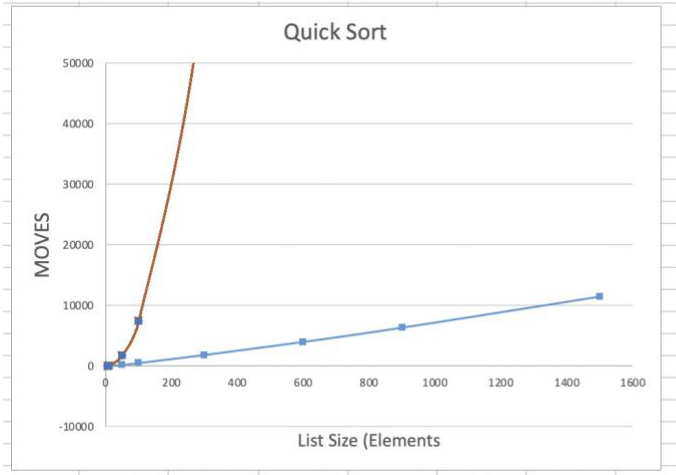


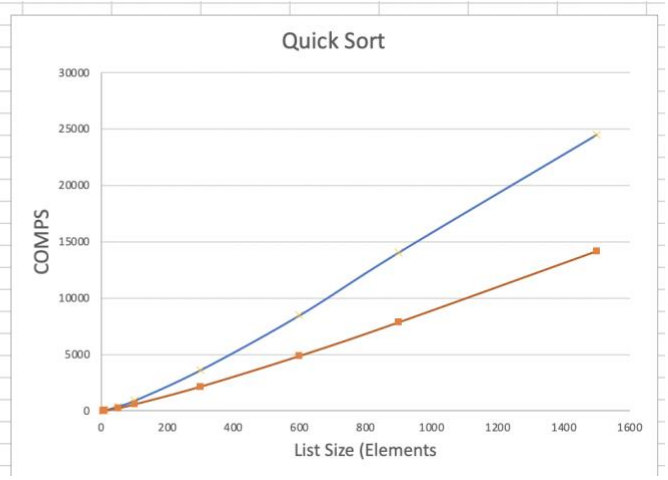
Binary insertion has an out of control Exponential count for moves
More reasonable almost linear comps count as it goes into the larger numbers range



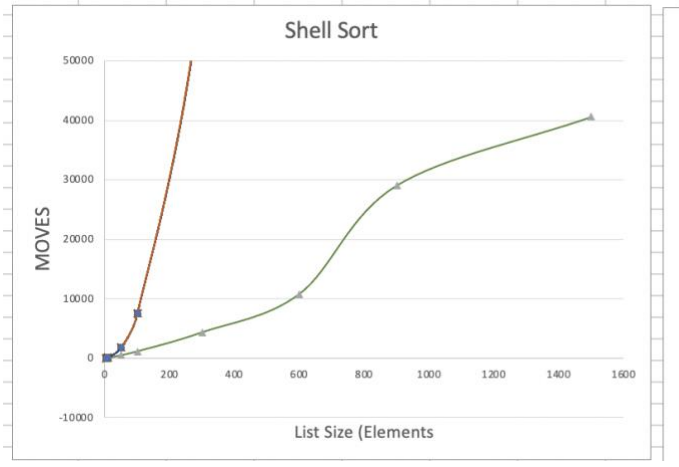


Quick sort compared to binary insertion
shows how much more efficient the moves count in large numbers
Comps are decent, almost linear next to binary insertions strong comp count

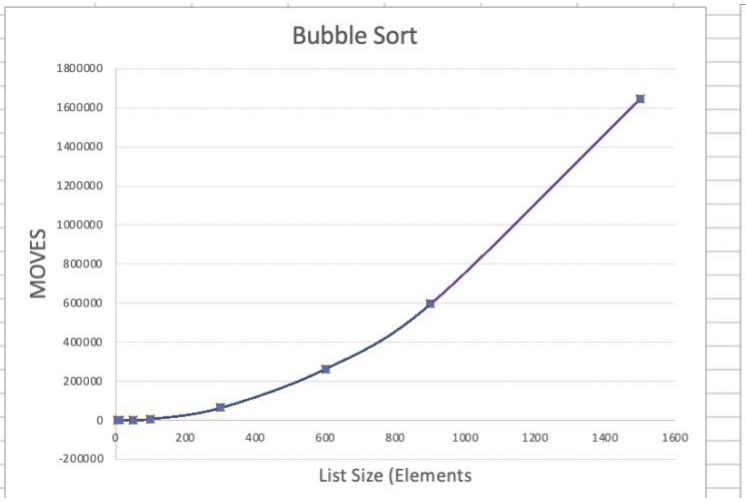


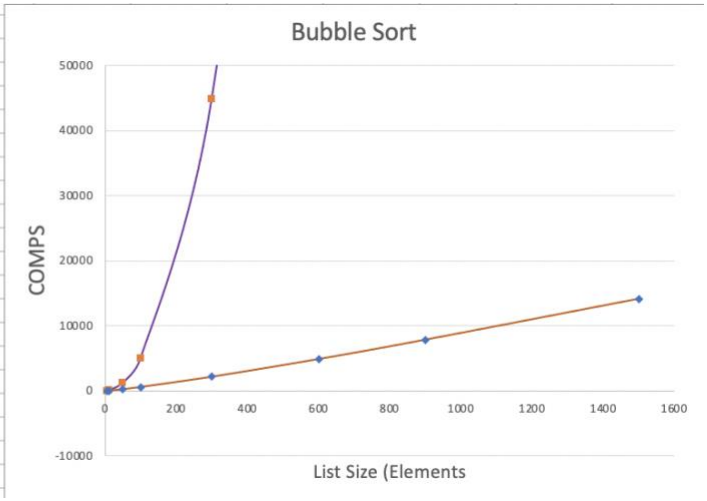


Shell sort Compared to Binary Insertion
The Moves count of shell sort weirdly tappers off here
The Comps are exponential



Bubble Sort Moves (which is a replica of the Binary insertion moves count) is exponential
Its Comps are also staggeringly exponential compared to binary insertion.





Conclusion: all sorting comparison sorting algorithms have pros and cons in its number of operations (moves vs swaps)

The best that can be done for a comparison sort is O(n log n) performance as seen by Quick Sort.
Arguably the least efficient sorting algorithm is Bubble sort which underperforms in both moves and comps.