****

**A screen shot of a computer

Description automatically generated**

**Selection Sort**

The basic premise of selection sort is that starting from the left most element, we compare that element to every element that comes after it to find the “best” element in the subarray. During each iteration, selection sort guarantees that at least one element will be sorted correctly, resulting in us to having to do one less comparison each time. Although this means that not exactly n comparisons are being conducted each iteration, we still generalize it to approximately n because the relationship is still linear. Therefore, the hypothesis of time complexity is that for an input of size N it will take approximately N \* N or **O(N^2)** time. We expect allocations to remain constant as no recursive calls are eating up memory on the stack, and the only local variables we define are one time allocation (counters and such). Therefore space complexity is **O(1)**.

If the time complexity truly were N^2 we would expect 10000 comparisons and swaps for 100 classes, however we note that ProfileSorts only records about 5000. However, ProfileSorts does confirm that the relationship between the input and the number of compares + swaps (which represents time complexity) is quadratic. In fact, since we are rounding the number of comparisons up to N per iteration, the time complexity relationship is slightly better than a basic quadratic x^2 function. In conclusion, ProfileSorts confirms our initial hypothesis that the relationship between N and time taken follows a quadratic behavior or approximately (N^2) and that space complexity is constant.

A screenshot of a graph

Description automatically generated

X-axis: Size of Input (N); Y-axis: Time Taken (Comparisons + Swaps)

**Bubble Sort**

Bubble sort operates on a similar theory to selection sort, in that we will end up passing through the array N times, each time making one less comparison because Bubble Sort also guarantees at least one item will be sorted each pass. Bubble sort is an iterative algorithm, therefore any other additional memory allocated is independent of the size of the input. Therefore, space complexity must be **O(1).** Again, we can create a hypothesis that argues a time complexity of N passes by approximately N comparisons 🡪 **O(n^2)**.

Using the predictive model of n^2 we would expect 10 000 comparisons + swaps for 100 classes, but again we are short a couple thousand (7000). However, once again we can note that the relationship between input and time taken is roughly quadratic. We can attribute the slight discrepancies to two factors 1) the approximation of # of comparisons per pass and 2) the fact bubble sort can “finish early” in the case that no swaps are made. However, we recall that time complexity analysis looks only at the worst case and note that our ProfileSort data values represent average case. Also notice that space complexity remains 3 for all the tests runs. Therefore, the hypothesis that bubble sort’s time complexity is quadratic and space complexity is constant holds true.

A graph on a graph

Description automatically generated

X-axis: Size of Input (N); Y-axis: Time Taken (Comparisons + Swaps)

**Insertion Sort**

Just like bubble sort and selection sort, insertion will guarantee that one element is sorted each time. The main difference is that insertion will focus only on one element at a time and compare it to every other element in the unsorted subarray. Therefore, we hypothesize that Insertion Sort is O(n^2) time complexity once again. Insertion sort is also an iterative algorithm, therefore any other additional memory allocated is independent of the size of the input. Space complexity is O(1).

Following the same logic as the other O(n^2) algorithms we expect to see about 10 000 comparisons and swaps but only see 5 000. However, we once again note that the relationship is roughly quadratic in nature. The generalization of the time complexity to n^2 visualizes the worst case for insertion sort, and our experimental data points follow that behavior very closely. We also see that allocations are constant, consistent with our initial claims of time complexity of O(n^2) and space complexity of O(1).

A graph on a graph

Description automatically generated

**Quicksort**

Quick sort utilizes a divide and conquer method, splitting the array into two subarrays and recursively splitting the resulting subarrays. This process will repeat log (n) times, until all subarrays are of size 1 or 2. From that point approximately n comparisons will be made on all subarrays of size 2. We can hypothesis quick sort to have a time complexity of **O(n log(n))**. Unlike the previous sorts Quicksort is recursive, and each recursive call will allocate memory on the stack. Each recursive call spawns two more recursive calls and if we use an in place method (we do all our operations on a single array and don’t create excess subarrays), space complexity is O(log (n). This is what I did in my implementation. In the case that we don’t use an in-place method and create extra subarrays, space complexity will be **O(n log(n) )**, which is the hypothesis I will argue here.

**Time Complexity**

An estimation of comparisons + swaps for n = 100 is around 665 and the recorded experimental value is around 800. We can see below that the time taken (comparisons and swaps) is only slightly worse than n log(n). The recorded behavior roughly mimics that of a linear logarithmic graph, supporting our initial hypothesis of a time complexity of O (n log(n))

**A graph on a graph

Description automatically generated**

**Space Complexity – Argued using PDF data because my implementation utilized an in place method with O(log n) space complexity**

We would expect the time complexity to be O(n log n). For n=100 we expect around 600 allocations but instead get 808 allocations. For n = 1000 we expect around 9965 allocations and record 8224. Since each recursive call spawns two more, we expect space required to be at least log (n). Also, since we are creating a sub array with each recursive call to work with we also require a linear amount of memory n. Therefore, this is consistent with the hypothesis that space complexity is O(log n).