HW3 algorithms

Daniel Zawacki, Alexander Vu, Robert Matzkin

February 2024

1 partition in Quicksort

For both implementations, an array of size 5000 was filled with random numbers between the range of -1000000 to 1000000. After running our sort function, a second function ran a pointer through the whole array starting at 0, and only returned false if the number ahead was not greater than or equal to the number the pointer is on. This process was run multiple times for confirmation. This test handles both positive and negative numbers in a random order when passed to the implementation, allowing for a wide range of cases to be tested by both versions, and handles the edge case wherein there are duplicate elements.
for the run time analysis, see the figure below.

2 binary search sqrt

```
BinarySearch(int x, lo, high, precision)
if x=0 or x=1:
  return x
if x<0:
    throw error
mid = (lo+high)/2
sq = mid^2
if abs(sq-x)<=precision: //if correct
    return mid
elif sq < x: //if less
    return BinarySearch(x, mid, high, precision)
else: //if more
    return BinarySearch(x, lo, mid, precision)
```

This algorithm starts by handling base cases and errors. if $x = 0, 1$ or $x < 0$, the algorithm returns 0, 1,or error respectively, as the square roots of 0 and 1 are 0 and 1 and negative numbers don't have real square roots. From there, the algorithm computes the middle of the provided range and creates three cases. First, if the middle of the range squared is within the precision range of the target x(i.e. $mid^2 - x \to 0$), then the middle is a sufficiently close approximation to the square root of x. second, if mid squared is too much less than x, then by extension the middle is too much less than the square root of x, and becomes the new low to be called again. Thirdly, if mid squared is too much more than x, then by extension the middle is too much more than the square root of x, and becomes the new high to be called again. This algorithm does assume that the square root of x is within the range provided, but so long as it is, he cases it creates will only ever find the square root or get closer too it without passing it over.

Because the size f the range provided would be cut roughly in half with each subsequent call, the time complexity of this algorithm would be O($log(n)$), where n is the size of the range between lo and high. To be specific, this means the run time is not linear with the input size itself, which is always constant.

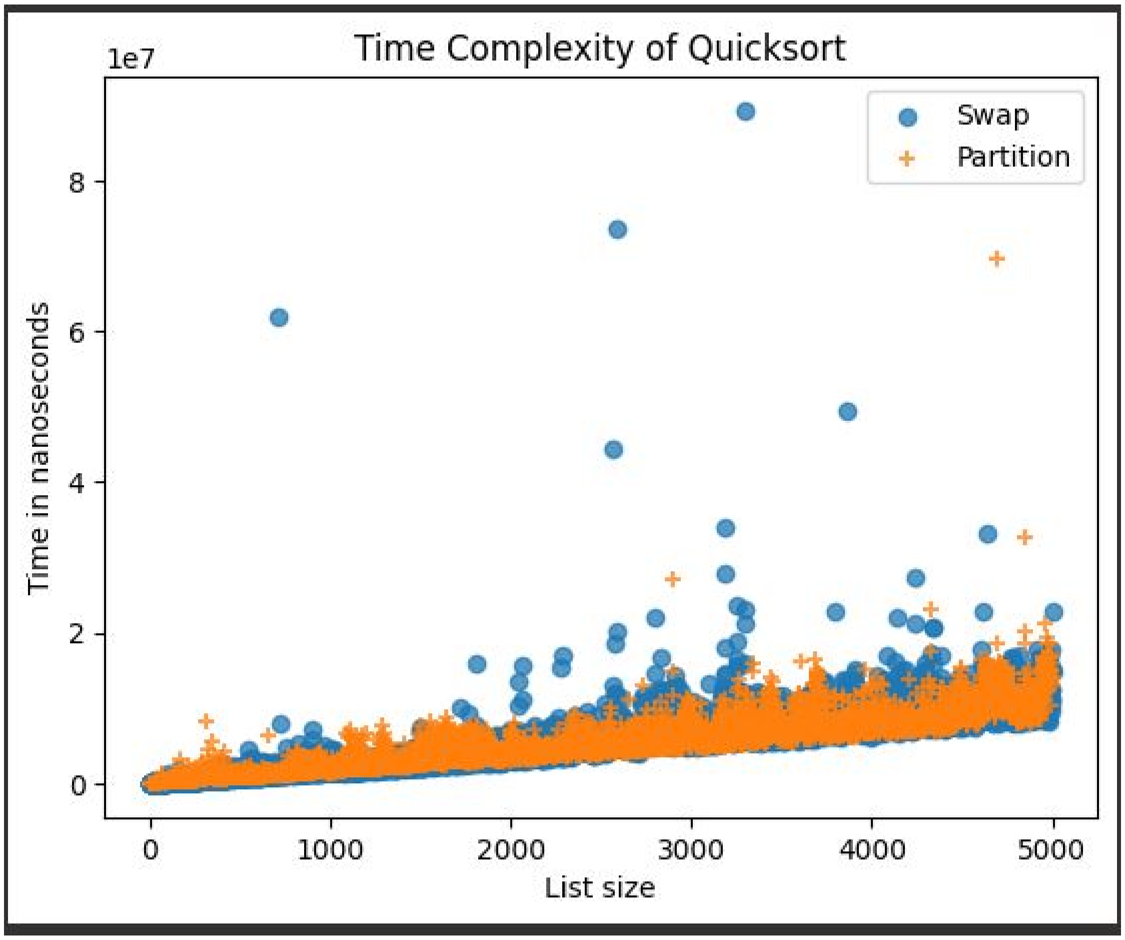

Figure 1: This graph plots randomly generated sample arrays of up to size 5000, along with how long it takes to sort them in nanoseconds.

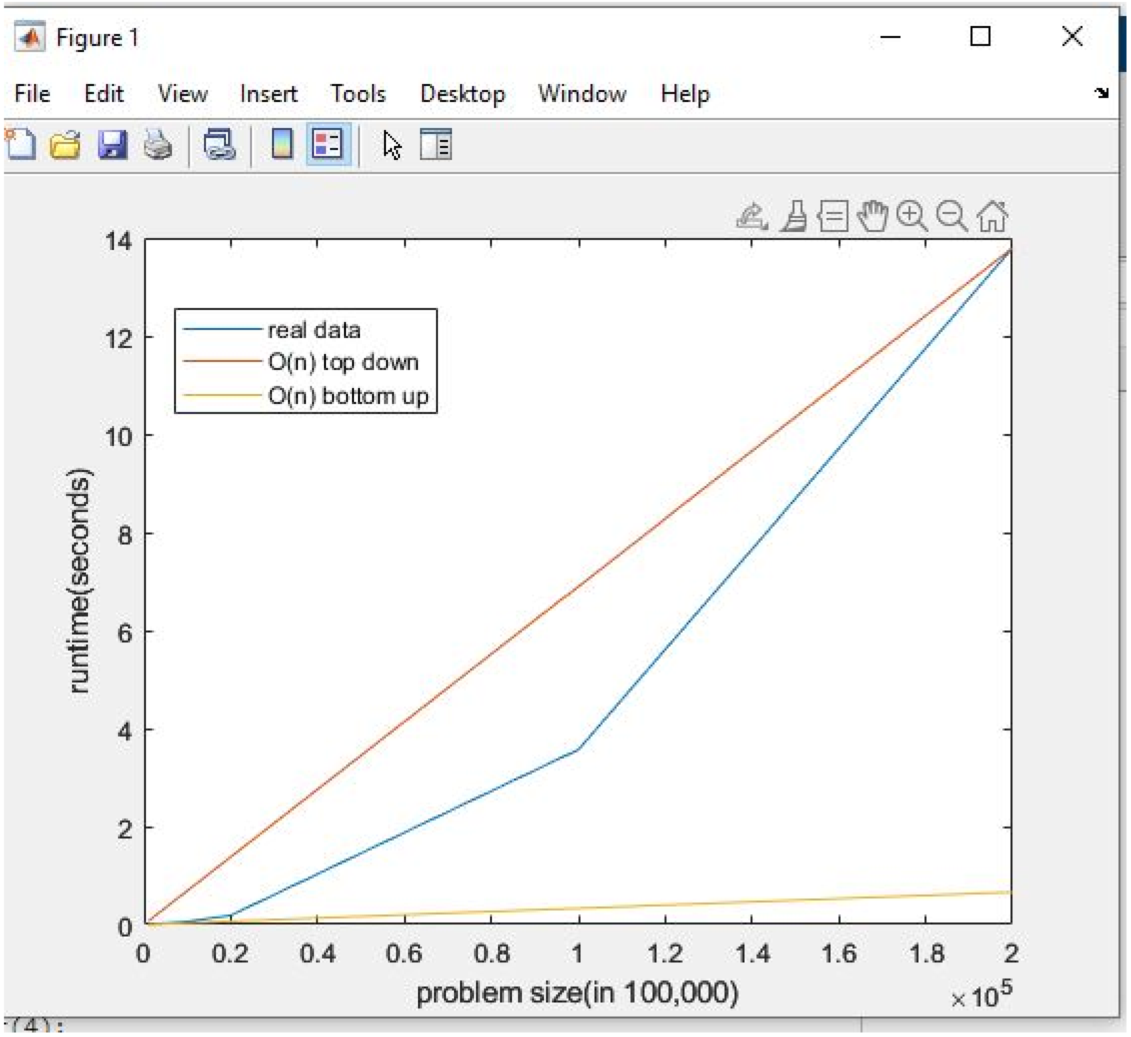

Figure 2: Blue is real data, red is O(n), found by using the data from the high end and linearly dividing down, orange is O(n) found by starting from the low end of the data and linearly multiplying upwards. Though still starting notably slower, the data does seem to more closely follow the trend of O(n) with the coefficient found using the upper extreme of the data