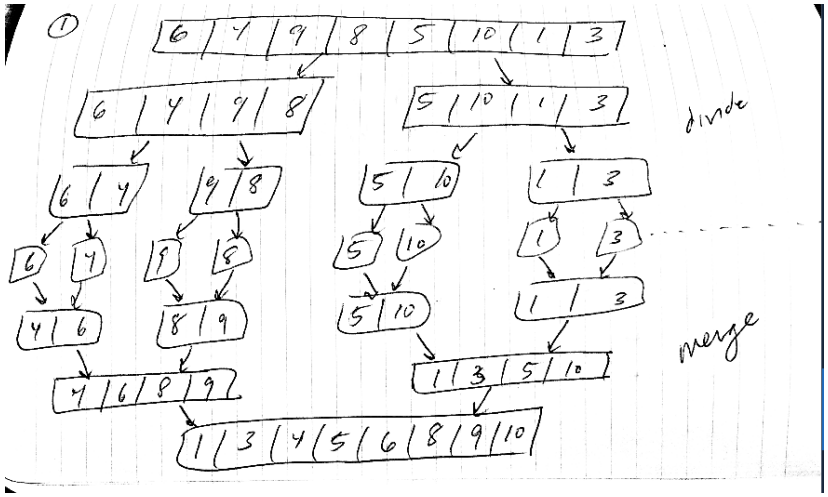
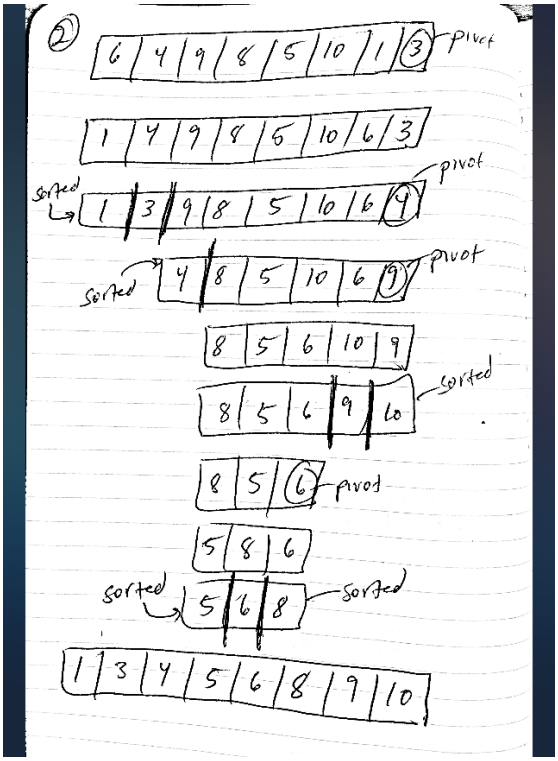
Brian Bauman
**CSC 421 Assignment 2**

1.




2.



3.

recursiveSort A from index 1 to n

where recursiveSort is defined as

```
key = A[n]
recursiveSort A from 1 to (n - 1)
i = n - 1
while i > 0 and A[i] > key do
      A[i + 1] = A[i]
      i = i - 1
A[i + 1] = key
```

In this pseudocode, the running time T(n) is equal to T(n - 1) [the recursive call on an array that is one element smaller] plus (n - 1) comparisons performed in the while loop.

```
T(n) = T(n - 1) + n - 1
T(n) = T(n - 2) + n - 2 + n - 1
T(n) = T(n - 3) + n - 3 + n - 2 + n - 1
T(n) = T(n - 4) + n - 4 + n - 3 + n - 2 + n - 1

T(n) = T(n - i) + i*n - (1 + 2 + 3 + … + i)
```

When the size of the subproblem is 1, no comparisons need be made. As such, T(1) = 0. So, T(n - i) = 0 when (n - i) = 1 or when i = n - 1. Plugging this value of i back in, we get

```
T(n) = 0 + (n - 1)*n - (1 + 2 + 3 + … + n - 1)
     = n^2 - n - (n/2)*(1 + n - 1)
     = n^2 - n - (n^2)/2
     = (n^2)/2 - n
```

4.

```
i = (n + 1) / 2
if (size of A == 1 && A[i] != i) return false (exit condition)
if (A[i] > i) then
      recurse on A[1..(i - 1)]
else if (A[i] < i) then
      recurse on A[(i + 1)..n]
else return true (match condition)
```

This algorithm works because if A[i] > i, it is impossible for any value A[j], where j > i, to be equal to j, because j would need to grow faster than A[j] grows in order to satisfy this condition. As the integers are strictly increasing, the slowest that A[j] can grow is by 1 (the next integer), which is also the fastest j can grow (array indices). By the reverse logic, if A[i] < i, it is impossible for any value A[j], where j < i, to be equal to j.

Because this algorithm reduces the size of the problem set by a factor of two with each recursive call (either taking the top or bottom half of the array), the result is returned in O(lg n) time.

5.

a.

max(A, 1, n) can be shown with the following pseudocode

```
mid = n / 2
leftMax = recursively find the maximum value in A[1..mid]
rightMax = recursively find the maximum value in A[mid + 1..n]

if (leftMax < rightMax) return rightMax
else return leftMax
```

This algorithm has a running time, T(n), of 2*T(n/2) [the cost of recursing on each subarray] + 1 [the cost of the final comparison between the maximums returned from each subarray].

```
T(n) = 2*T(n/2) + 1
T(n/2) = 2*T(n/2^2) + 1

T(n) = 2^2*T(n/2^2) + 2
T(n) = 2^3*T(n/2^3) + 3
T(n) = 2^i*T(n/2^i) + i
```

T(n) = 1 when n = 2 (we don't need to recurse here, we can just compare the nubmers).

As such, T(n/2^i) = 1 when n/2^i = 2, or when n = 2*(2^i), or when i = lg(n/2). Plugging this back in, we get

```
T(n) = n/2 + lg(n/2) = (1/2)*n + lgn - 1
```

Asymptotically, this algorithm finishes in O(n) time.

b.

No, this algorithm does not perform better. This is because, although it recurses on subarrays, this algorithm still needs to inspect every element in each half of the array, just as the incremental approach would. As such, they both run in O(n) time.

6.

a.

Insertion sort sorts an array of length in in O(n^2) time in the worst case. For a sublist of length k, insertion sort will sort it in O(k^2) time in the worst case. If this must be performed (n/k) times, we can expect the worst case time to be O((k^2) * (n/k)), which reduces to O(nk).

b.

In merge sort, the merging step takes 2*k operations for the first merge (of two sublists of length k), the second merge step takes 4*k operations, then 8*k, etc., until we reach a fully merged list. If the full array is of size n, we can expect to require ~lg(n) - lg(k) merges, as the fact that we are starting with sublists of length k saves us from merging together sublists that are smaller in length than k (hence the subtraction of lg(k)). Because each merge step must run through all n of the elements (though it is doing so in separate sublists), the complexity of this problem is equivalent to O( n * (lg(n) - lg(k)) ). Written differently, this is O(nlg(n/k)).

c.

To determine this, we compare the time complexities with eachother:

```
nk + nlg(n/k) vs. nlg(n)
```

nlg(n/k) will always be smaller than nlg(n) (for values of k > 1). As such, we must determine when the nk term will become larger than nlg(n). By comparison, we can see that this is when

```
nk >= nlg(n), or

k >= lg(n)
```

7.

a.

I'm having trouble explaining this one, but I am sure that it requires computing the convex hull, as that was discussed in class and in this section.

b.

Assuming that you have split the problem into two subproblems in (a), where the ghosts on one side are equal to the ghostbusters on one side, you can simply reapply the same algorithm for this subset, finding a beam-line that splits the subset into two sections where the number of ghosts and ghostbusters are equal. If (a) ran in O(nlgn) time, this would mean calculating this for each addtional ghostbuster (n total), bringing the running time to O(n^2lgn).