1 Quiz

Note that all indexes used are zero-based and ranges/sublists are done using [inclusive...exclusive] convention. Also, python is used as pseudo-code because it is essentially pseudo-code already.

1.1 Section 2.2 Problem 2

```python
def selection_sort(ls):
    for i in range(len(ls)-1):
        min_j = None
        min_val = None
        for j in range(i, len(ls)):
            if min_val == None or ls[j] < min_val:
                min_j = j
                min_val = ls[j]
        ls[i], ls[min_j] = ls[min_j], ls[i]
        print(i, min_j, ls)
    return ls
```

During an iteration of the inner loop, the loop invariant is that the elements before the current index (i in the above pseudocode) are in sorted order before and after the inner loop is ran. Another invariant is that all elements to the left of the current index are less than the elements to the right (and under) the current index. Stated more rigorously, the sublist $A[0...i]$ is sorted, and $\forall x \in A[0...i], \forall y \in A[i...N] : x < y$.

It does not need to be ran on the final element of the array because there are no possible elements for it to be swapped with. This can also be shown using the second invariant, under which we know that the final element must be the greatest.

Best and worst case are both $\Theta(n^2)$, because the minimization step has to be ran completely every inner loop taking an average of $n/2$ steps, and the outer loop runs $n - 1$ times.

1.2 Section 2.3 Problem 5

```python
def binary_search(ls, item):
    '''Assumes ls is a list of numbers sorted increasing '''
    if len(ls) == 1:
        if ls[0] == item:
            return 0
        else:
            raise Exception('Not found')
    midpoint = len(ls)//2
    if ls[midpoint] == item:
        return midpoint
    elif ls[midpoint] < item:
        return midpoint + binary_search(ls[midpoint:], item)
    elif ls[midpoint] > item:
        return binary_search(ls[:midpoint], item)
```

Worst case running time must be $\Theta(lg\ n)$ because each step either reduces the size of the remaining list to be half of its previous size or terminates. Because the program is recursive, we can be certain of this because every recursive call calls the function with the list sliced in half, taking either the first or last half.

1.3 Section 2.3 Problem 6

If the values are stored in an array, no, because the swapping that takes place will still take linear time. However, if the values are stored in a linked list this would not be a problem, but unfortunately binary search cannot run in $lg\ n$ on a linked list. As such, it seems that you cannot obtain $\Theta(lg\ n)$ by combining insertion sort and binary search.

2 Homework (optional)

2.1 Section 2.1 Problem 1

Dividing n items into n/k sublists of length k means that insertions sort will be called n/k times on lists of length k. Each of those calls will cost $\Theta(k^2)$ so n/k of them will cost $\Theta(k^2 \times \frac{n}{k}) = \Theta(nk)$

2.2 Section 2.1 Problem 2

Starting with the n/k sorted sublists of length k, we can write the recurrence for this as

$$F(n) = \begin{cases} \Theta(1) & \text{if } n \le k, \\ 2T(n/2) + \Theta(n) & \text{otherwise.} \end{cases}$$

Using the same analysis as shown in the textbook, we arrive at $\Theta(n\ lg(n/k))$. I think.

2.3 Section 2.1 Problem 3

If k is a constant factor of n, the worst case time for the modified algorithm becomes $\Theta(n^2 + n\,lg(n/n)) = \Theta(n^2)$, which is worse than standard merge sort.

Now I can tell that the nk term means that the only choice for k is $lg\,n$ and below, so let's try $k = \Theta(lg\,n)$. This gives $\Theta(n\,lg\,n + n\,lg(\frac{n}{lg\,n})) = \Theta(n\,lg\,n + n\,lg\,n - lg\,lg\,n) = \Theta(n\,lg\,n)$

2.4 Section 2.2 Problem 1

You also need to prove that no elements are removed or deleted. I'm not sure of an elegant mathy way to express this better that multiset equivalence, but knowing that the "exchange" operation preserves this is good enough.