Exercise 1. *Design a greedy algorithm for the interval scheduling problem.*

Solution. First, sort the classes by ending time. Add the first class to the schedule. Next, iterate through the remaining classes adding one to the schedule if its start time is after the previous class's end time. The resulting schedule will have the maximum number of classes possible (for one room to hold).

```
// Input: Array of classes C
// Output: Maximum number of classes in one room

procedure main:
    sort_by_end_time(C) // n log(n) ops
    sched += C[0] // 2 ops
    for i in 1 to size(C): // n times
        if C[i].start >= sched.last.end: // 1 op
            sched += C[i] // 2 ops

    return size(sched) // 1 op
```

We can see that running time is approximately $n \log n + 2 + n \cdot 2 + 1 \in \mathcal{O}(n \log n)$.

Let the optimal schedule be O with m classes and the greedy schedule be G be non-optimal with n classes; thus, we know that $m < n$. Let $0 \leq k \leq m$ be the first index where $O[k] \neq G[k]$. We know that we can substitute the class at $G[k]$ in for the class at $O[k]$ because it is the earliest ending class that can be placed at that time and thus, does not conflict with the rest of O. Now, G and O are the same up until $k + 1$; this process can be continued until G and O are equivalent for 0 to m. We know O will still have further classes left because $n > m$ yet this not possible since $G[m - 1]$ is latest possible class that can be scheduled (all later classes are conflicting or non-existent). Thus, G must be the optimal schedule.

Exercise 2. *Consider the segment covering problem discussed in class. Design a greedy algorithm to solve the problem, and prove that the greedy solution is the optimal solution.*

Solution. First, sort the array elements by right endpoint in ascending order. Iterate through the sorted array, and select the first element which has a left endpoint which covers the beginning of the target segment. Repeat this process comparing the previously selected element's right endpoint (instead of the beginning of the target's left endpoint) with the left endpoints of the sorted array elements. When then end point of the selected element covers the right endpoint of the target segment end point, return the number of selected elements.

```
// Input: array S of line segments and segment l which must be
   ↪ covered
// Output: minimum number of segments from S needed to cover l

procedure main:
    sort_by_end(S) // n log(n) ops
    low_bound = l.start // 1 op
    segments = [] // 1 op
    while low_bound < l.end: // n times (worst)
        for i in 0 to size(S)-1: // n times
            if S[i].start <= low_bound: // 1 op
                segments += S[i] // 1 op
                low_bound = S[i].end // 1 op
                remove S[i] // 1 op
                break // 1 op

    return size(segments) // 1 op
```

We can see that the running time is approximately $n \log n + 2 + n \cdot n \cdot (1 + (4)/2)) + 1 \in \mathcal{O}(n^2)$.

Let G be the greedy solution with m elements and O be the optimal solution with n elements; we know that $m \geq n$. Let $0 \leq k \leq n$ be the index of the first element that is not same in G and O. Since $G[k]$ covers the maximum area while still covering the left end point of what is currently covered (or the beginning the target segement if it is the first element), we know that we can swap it with $O[k]$ and O will still cover the target segment. Now, $k+1$ will be the index of the first different elements.

Continue this process until the elements of G have been swapped for all of O (up to $O[n]$). Since O is still a valid solution, we know that $O[n]$ covers the right endpoint of the target segment; according to the above greedy algorithm, we can see that the last element of G is the first element that covers the right endpoint of the target segment. Thus, we know that $O[n] = G[m]$ and that $n = m$; it follows that G is optimal.