Example Solution: Divide And Conquer and Dynamic Programming

Feel free to work in groups of at most 4 for these - if you have a group of more than 4, please run it by me first. If you do work in a group, please include the names of those that you worked with. **However: each student should submit a separate copy where the solutions have been written by yourself.**

1. In this problem, we are given an array A of length n and want to count the number of inversions within the list. Recall that an inversion is a pair (i, j) such that $i < j$ and $A[i] > A[j]$. Construct an algorithm that determines the number of inversions in $O(n \log n)$ time.
   Solution: We will use a Divide And Conquer strategy to solve this problem. Given A of length n, we will create two arrays $B = A\left[1 : \frac{n}{2}\right]$ and $C = A\left[\frac{n}{2} : n\right]$. We will recurse on B and C to determine the number of inversions in each, and then count the number of inversions that cross B and C and return the total sum of inversions. As part of our recursion, we will sort B and C to make it easier to count the inversions that cross them. To count the inversions, we loop through both B and C, comparing corresponding elements. Since both B and C are already sorted by means of our recursion, when we have an inversion, we know that all remaining elements in B must also form inversions with the current element in C. On the other hand, if the current element in B doesn't form an inversion with the current element in C, then no remaining elements in C would form an inversion with the current element of B either. The following pseudocode implements this idea:

```
def inversionCount(A):
    if len(a) == 1:
        return (A, 0)
    else:
        (B, I1) = inversionCount(A[1:n/2])
        (C, I2) = inversionCount(A[n/2:n])
        Initialize i = 0 and j = 0 as the start of B and C
        Initialize sortedA as an empty list
        Initialize I3 as 0
        while the ends of both B and C are not reached:
            if B[i] < C[j]:
                Add B[i] to sortedA and increment i
            else if B[i] > C[j]:
                Add C[j] to sortedA, increment j,
                    and increase I3 by len(B)-i
            else: # If they are equal
                Add C[j] to sortedA and increment j
        Add whatever is left in B and C to sortedA
        return (sortedA, I1 + I2 + I3)
```

   Let $T(n)$ be the runtime of the above algorithm for an input of length n. Then $T(n) = 2T\left(\frac{n}{2}\right) +$ however much time it will take to merge and count the inversions in the middle. Since we are only iterating through B and C once each at most, this gives us $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$, which we know from class gives us $T(n) = O(n \log n)$.

2. In this problem, we are given a list of intervals $I_1 = (s_1, e_1, v_1),\ \ldots,\ I_n = (s_n, e_n, v_n)$ where s_i, e_i and v_i are the start time, end time, and value of I_i. Construct an algorithm that determines the maximum value non-conflicting collection of intervals from the input.
Solution: Let $\textsc{Opt}(j)$ denote the sum of the optimal collection of intervals from the sublist $I_1, \ldots, I_j$. Note that in the optimal solution, either I_j was included in the sum or it wasn't. If it was not included, then $\textsc{Opt}(j) = \textsc{Opt}(j-1)$. Otherwise, $\textsc{Opt}(j) = \textsc{Opt}(p(j))$ where $p(j)$ is the index of the latest job that ends before I_j starts, or 0 if no such job exists. Hence

$$\textsc{Opt}(j) = \max(\textsc{Opt}(j-1), v_j + \textsc{Opt}(p(j))). \tag{1}$$

To calculate $\textsc{Opt}(n)$, we create an array A such that $A[j] = \textsc{Opt}(j)$. We initialize $A[0] = 0$ (since 0 value can be obtained from 0 intervals), and then set $A[j] = \max(A[j-1], v_j + A[p(j)])$ for $j = 1$ to n (in that order). We can precalculate p in linear time (left as an exercise, obviously a student cannot leave this open but it's a good problem for you all, so I recommend thinking about it). Hence, since it takes only a constant amount of time to calculate $A[j]$ if $A[0],\ \ldots,\ A[j-1]$ have all already been calculated, we have that A can be computed in linear time. Since $A[j]$ follows the recursive formula in Eq.1, we have that $A[n] = \textsc{Opt}(n)$, giving us a linear time solution.