Consider the Longest Increasing Subsequence Problem from class. In this problem, we are given a list of numbers $x_1, ..., x_n$ and want to find the longest increasing subsequence (not necessarily consecutive) within this list. In class, we first considered the tree of possible solutions:

We argued that the optimal solution would be somewhere at the bottom level of this tree, but that there would be $2^n$ nodes at the bottom and thus it would be infeasible to check all of them. So we came up with a few pruning rules that would reduce the number of solutions we’d have to feasibly check:

i) Any sequence that doesn’t even form an increasing sequence can be pruned.

ii) If two sequences on the same depth have the same length, prune the one that ends in a larger number.

iii) If two sequences on the same depth end in the same number, prune the one with less length.

1. Convince me that each of the pruning rules are valid - that is, even after pruning off the described subtrees at every level, we will still have the optimal solution somewhere on the bottom level of the remaining subtrees.
2. In class we tried to use i) and iii) to develop an algorithm. Technically the one we came up with in class ended up being $O(n^3)$ to compute. Let $\text{Opt}(j, k) = \text{the longest increasing subsequence of the first } j \text{ numbers that ends on } x_k$. Show that we can compute an array $A$ of size $n \times n$ such that $A[j, k] = \text{Opt}(j, k)$ in $O(n^2)$ time.

3. Let’s try to use i) and ii) to develop an algorithm. Let $\text{Opt}(j, k) = \text{the minimal last number of length } k \text{ subsequences from the first } j \text{ numbers.}$ Show that we can compute an array $A$ of size $n \times n$ such that $A[j, k] = \text{Opt}(j, k)$ in $O(n^2)$ time.