1. (30%) Medians and order statistics: If we had an algorithm that finds the median of a sequence in linear time (worst case) – findMedian(A, p, r) which returns the index of the median of the sequence $A[p..r]$, describe a worst-case linear time algorithm that finds any order statistics. Provide a brief runtime analysis.

**Answer:** This is not very different from the selection algorithm, except that at every stage we first find the median and then use it as a pivot for Select (same select that we showed in class).

The runtime is as follows: The boundary condition is $T(1) = d$ (line 1). Otherwise we perform two linear operations – finding the median and partitioning, and since every partition is guaranteed to cut the array in half, the recurrence runtime is $T(n) = T(n/2) + O(n)$ which we know is linear. The median finding adds a linear term to the original select, but it doesn’t change the asymptotic runtime since both select and partition are linear.

2. Binary search trees:
   (a) (15%). Let x be a leaf node in a binary search tree T. Let y be x’s parent. Show that y.key is either the smallest key in T larger than x.key or the largest key in T smaller than x.key.

**Answer:** We know that x is a leaf. If $y < x$ then x is on y’s right subtree, but x is a leaf so it’s all of y’s right subtree. We already saw (HW3) that if y has a right subtree, then its successor is the minimum node on that subtree. Since x is the only node on the subtree, it’s certainly the minimum too, hence y’s successor. if $y > x$ this is the exact symmetric argument with respect to the predecessor.

(b) (15%) Is the following claim true or false? Explain: in order to determine whether two binary search trees are identical one has to perform an in-order walk on them and compare the results.

**Answer:** No, because the inorder gives you the sorted order of the nodes, which is identical for any BST with the same keys (regardless of how they are positioned on the tree).

3. (40%) Dynamic programming: Given an array A of n numbers, the maximum subarray problem is the task of finding the contiguous subarray $A[i..j]$ of numbers which has the largest sum. For example, if $A = \{-2, 1, -3, 4, -1, 2, 1, -5, 4\}$ then the subarray that gives the maximum sum is $\{4, -1, 2, 1\}$ with sum 6 (emphasized in bold font). Let us define MS(i) as the maximum sum subarray that ends at $A[i]$ (and must include $A[i]$). For example, in a 1-based index, $-MS(1) = \{-2\}$. $MS(2) = \{1\}$ (since concatenating -2 and 1 gives a smaller sum, so MS(2) includes only A[2]). In other words – for MS(i) we ask ourselves which one is better – for $A[i]$ to extend MS(i-1) or be its own subarray.

(a) Show that the problem has the optimal substructure (Hint: $A[i]$ either extends the maximum sub-array that ends in $A[i-1]$ or alternatively, includes only $A[i]$ itself. Use a cut-and-paste argument for MS(i-1) with respect to MS(i)).

**Answer:** If $A[i]$ extends MS(i-1), then MS(i-1) is the maximum subarray sum that ends in $A[i]$. If not, we could find a larger MS(i-1), concatenate $A[i]$ and get a better sum. If $A[i]$ is its own subarray, it can only happen if MS(i-1) is negative. In this case it’s not worth it to add to $A[i]$.
(b) Define a recursive algorithm that calculates MS(i), that can be used as a basis for a dynamic programming calculation. Remember to also return the overall maximum sum. It doesn’t have to be MS(n) (why?).

Answer: \( MS(i) = \max(MS(i-1) + A[i], A[i]) \) (for all i). The final answer to the algorithm is \( \max(MS(i)) \) for all i.

(c) Based on that, calculate MS(i) for every index in the array above.

Answer: \( MS = \{-2, 1, -2, 4, 3, 5, 6, 1, 5\} \) (the index that ends the max. subarray sum is emphasized in bold fonts).