1. (20%) **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.

**Solution:** We can run it to select the pivot for select instead of selecting it at random. In this case we’ll always have the pivot located in the middle of the partition, therefore the runtime is guaranteed to be linear: $T(N) = T(N/2) + O(N)$ (linear according to the master theorem).

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.

   **Solution:** This is just another way to define the successor/predecessor. If $x$ is a left child, then its parent $y$ has a larger key. Since $x$ is a leaf, then it has no subtree so its successor has to be the lowest ancestor s.t. the subtree $x$ resides on is the left, which is exactly $y$. The same argument (symmetric) holds if $x$ is a right child.

   (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.

   **Solution:** False. In-order walk gives us the sorted set of keys, which is identical for any two BSTs with the same set of keys but a different layout.

3. (30%) **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 $MS(i)$).

   **Solution:** Given an optimal solution $S$ that ends in $A[i]$, let us look at the subarray ending in $A[i-1]$. If $MS(i-1)$ were not optimal we could have made a better solution and attach $A[i]$ to it, resulting in a better solution overall, contradicting the assumption that $S$ is optimal. It doesn’t matter if $A[i]$ is its own solution or is attached to a subarray. The argument stays the same.
(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).

4. (20%) **Greedy Algorithms:** Consider a set of points on the real line:

![Points on the real line](image)

We want to find the minimum number of unit intervals that cover all the points.

Here is a simple greedy algorithm to solve the problem:

1. Consider the points in sorted order from left to right: \( \{p_1, p_2, \ldots, p_n\} \)
2. Use the unit interval \( i_1 = [p_1, p_1 + 1] \) (the interval starting at the leftmost point and extends one unit to the right).
3. Repeat the same process with all the points not covered by \( i_1 \), until all the points are covered.

When applied on the example above we get the following:

![Greedy algorithm](image)

The greedy choice here says that there must be an optimal solution that contains \( i_1 = [p_1, p_1 + 1] \). Show that it is safe to select this interval. (**Hint:** \( p_1 \) must be covered somehow, right?...)

**Solution:** This is the same argument as the activity selection problem we saw in class. Say \( S \) is an optimal solution. The first point has to be covered somehow, so there must be an interval in our set that contains \( p_1 \). If it is \( [p_1, p_1 + 1] \) we’re done. Otherwise, we can replace that interval with \( [p_1, p_1 + 1] \). \( p_1 \) is still covered but we "pulled" the first interval to the right, so it must cover everything covered before and even more. A similar argument holds for any other uncovered point later down the line.