Homework 3

1. Design an O(log(n)) time algorithm to find the peak value p of a unimodal array A.

If the size of A is three or smaller, solve this problem any straightforward way. Otherwise, begin by finding $a_{n/2}$, and comparing to $a_{(n/2 - 1)}$ and $a_{(n/2 + 1)}$. We will call these elements middle, lower, and upper. If middle is smaller than lower, recursively repeat this procedure on the subarray from a0 to middle. If middle is smaller than upper, recursively repeat this procedure on the subarray from middle to an. If neither of those two cases is true, it holds that middle is greater than lower and upper, meaning that it must be the peak entry p.

Pseudocode for the above solution is as follows:

```
peakFinder(A) {
      if (A.size() ≤ 3) return A[floor(A.size() / 2)];
      middle = A[floor(A.size() / 2)];
      lower = A[floor(A.size() / 2) - 1];
      upper = A[floor(A.size() / 2) + 1];

      // Using Python Array Slicing notation:
      // A[start:end] # items start through end-1
      // A[start:]    # items start through the rest of the array
      // A[:end]      # items from the beginning through end-1
      // A[:]         # a copy of the whole array
      if (middle < lower) return peakFinder(A[:upper]);
      if (middle < upper) return peakFinder(A[middle:]);
      return middle;
}
```

2. In the SELECTION algorithm, will the algorithm still work in linear time if the input is divided into groups of seven instead of groups of five?

In class we showed that the running time for the Selection Algorithm was as follows:

```
T(n) = T(max(|A₁|, |A₂|)) + T(n / groupSize) + n and
max(|A₁|, |A₂|) = n - (n / groupSize) * (1/2) * ceil(1/2 * groupSize).

When groupSize = 5:
T(n) = T(7/10 * n) + T(1/5 * n) + n = T(9/10 * n) + n = O(n).
```

It then follows that when groupSize = 7, we can show if the running time is still linear with the following equation:

```
T(n) = T(n - (n/7) * (1/2) * 4) + T(n/7) + n
T(n) = T(n - (4/14)n) + T(2/14 * n) + n
T(n) = T((10/14 + 2/14) * n) + n
T(n) = T(6/7 * n) + n
T(n) = O(n)
```

Therefore, yes, SELECTION is still linear when groupSize is seven.

3. Given a multiple selection problem:

A. Design an O(n log(n)) time algorithm for the problem.

Simply sort A in ascending order using a canonical merge sort algorithm. Then, iterate through all m integers and return the value of A located at the 1-based index of k_i (For example, if $k_i = 1$, return the first element of A). This algorithm will take the time to sort A plus the time to iterate through the m integers, or because $m \leq n$,
T(n) = O(n log(n)) + O(m) = O(n log(n)) time.

B. Design an O(nm) time algorithm for the problem.

Given the single selection algorithm that runs in linear time, simply run single selection once for each of the m integers provided. This will take at worst linear time once for each of the m values, or O(nm) time.

C. Improve your algorithm to O(n log(m)) time.

Apply the selection algorithm to find the value of the $k_{(m/2)}$-th smallest value. At the end of your search, the array A will be split into three partitions. Values less than the $k_{(m/2)}$-th smallest value, the $k_{(m/2)}$-th smallest value itself, and values greater than the $k_{(m/2)}$-th smallest value. Then you split the problem into two sub-problems and recursively search for the smaller k values in the portion of A smaller than the $k_{(m/2)}$-th smallest value and larger k values in the portion of A that is greater. This would require
T(n) = O(n log(m)) time.

Pseudocode for the above algorithm is as follows, given K = the set of m values:

```
mid = K[K.size()/2]
multipleSelection(A, K, mid) {
        pivot = A good pivot value;
        A1 = all elements of A < pivot;
        A2 = all elements of A > pivot;
        i = A1.size() + 1;
        if mid < i { return multipleSelection(A1, K, mid); }
        if mid > i { return multipleSelection(A2, K, mid - i); }
        if mid = i {
                return [
                multipleSelection(A1, K[:K.size()/2 + 1], K[K.size()/4]),
                pivot,
                multipleSelection(A2, K[K.size()/2:], K[K.size()*(3/4)])
                ];
        }
}
```

4. Given a black-box function that can tell if x is a feasible value:

A. Design an O(n log(n)) time algorithm to find all feasible values of A by calling the function O(log(n)) times.

Sort the array A using a canonical merge sort, taking O(n log(n)) time. Then call the black box function on the middle value of A. If the middle value of A is a feasible value, then recursively apply this algorithm to the upper half of A. Otherwise, apply this algorithm recursively to the lower half of A. After at most log(n) calls to the function, you will have either the largest feasible value or the smallest value which is not feasible, depending on the result to the latest call to the black box. Then, all values smaller than this value are feasible, and all values larger are not feasible. This algorithm takes O(n log(n) + log(n)) = O(n log(n)) time. I suspect the algorithm could be made even more efficient by checking the middle value against the two values immediately above and below it as well, but that would not change the worst-case running time of the algorithm.

B. Improve your algorithm to O(n) time such that the total number of calls on the black-box function is still at most O(log(n)).

Begin by using the selection algorithm to find the median of A in linear time. Then, partition A into three groups, A1, the median, and A2. Run the black box on the median value. If it is a feasible value, then recursively apply this algorithm on A2. If the result of the recursive call is an empty set, return the median value, otherwise return the resulting number. If median is not a feasible value, simply return the result of a recursive call to A1. After log(n) calls, this will provide you with the largest feasible value. Finally, partition A once more using this value as the pivot, and return the set of all values less than or equal to this new pivot. This algorithm takes T(n) = T(n/2) + n time, which was shown in class to be equal to O(2n) time, plus the final n to partition the values, leads to a total running time of O(3n) = O(n) time.

Pseudocode for the algorithm described above is as follows:

```
feasibleValues(A) {

    // Return the only element of A if feasible, NULL if otherwise.
    if (A.size() == 2) return blackBox(A[0]) ? A[0] : NULL;

    // Partition A such that A1 < med < A2
    med = median(A);
    double[] A1;
    double[] A2;
    for each a in A {
        if (a < med) A1.append(a);
        if (a > med) A2.append(a);
    }

    result = blackBox(med);
    if (result) {
        toReturn = feasibleValues(A2);
        if (toReturn != NULL) { return toReturn; }
        else { return result; }
    }
    else {
        return feasibleValues(A1);
    }
}
```