Math 182: Problem Set 3
Kenny Guo

Question 1:

Given a collection of intervals of real numbers, $[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]$, say that a set of points $P \subset \mathbb{R}$ stabs the collection of intervals if for each interval $[a_i, b_i]$ there is some point $p \in P$ which is in the interval—that is, for which $a_i \leq p \leq b_i$. Such a set P is called a stabbing set for the set of intervals. Design an efficient algorithm which, given a list of intervals, finds a stabbing set of minimum size.

Example. Suppose the intervals are $[1, 2], [1, 4], [0, 3], [2.5, 8], [-1, 3], [7, 10]$ then the minimum number of points in a stabbing set is 2 and one stabbing set of size 2 is $\{1.5, 7\}$.

Sort the intervals by their ending values. Store the ending value of the first interval and stab it. Scan over the rest of the intervals. If the starting point of a new interval is greater than the current ending value stored, make that interval's ending value the new one, and stab it. Repeat until all intervals are scanned.

```
StabbingSet(I):
    n = len(I)
    Sort I by right endpoints
    curr_end = null
    P = empty set
    for i = 1, 2, 3, ... n:
        if curr_end == null or I[i].left > curr_end:
            curr_end = I[i].right
            Add curr_end to P
    return P
```

The runtime is $O(n \log n)$.

Proof. Sorting takes $O(n \log n)$. Then, every iteration of the loop takes constant work (comparisons, assignment, adding), so another $O(n)$ work. $O(n \log n)$ then dominates. □

This algorithm produces a stabbing set of minimum size.

Proof. First, note the algorithm does produce a stabbing set. After sorting, the first interval is stabbed to produce the first curr_end. Then for every subsequent interval:

- If the interval's left endpoint is greater than the current end, we add a stabbing point for its right endpoint, so it gets stabbed
- If the interval's left endpoint is less than or equal to curr_end, then curr_end stabs this interval, since its right endpoint is greater than curr_end by sorting.

Thus, all intervals are stabbed by P.

Now, we show optimality by induction. The base cases with no and one interval are trivially satisfied. Now assume that for some n, our algorithm produces a minimum stabbing set for any collection of less than n intervals. Consider a set of n intervals. Consider the first interval, $[a_1, b_1]$.

We claim that there exists a minimum stabbing set that contains b_1. Let P be an minimum stabbing set, so there is some $p \in P \cap [a_1, b_1]$. Replace p with b_1. Suppose an interval $i \neq 1$ was also stabbed by p, so $a_i \leq p \leq b_i$. Since $p \leq b_1 \leq b_i$ by sorting of the right endpoints, we also have that $a_i \leq b_1 \leq b_i$, so b_1 also stabs it. Since the size of the stabbing set did not change, this set is still optimal.

Thus, consider an minimum stabbing set for the n intervals with b_1. Remove b_1 from the stabbing set and drop all intervals it stabs, which is at least 1. By the inductive hypothesis, our algorithm finds a minimum stabbing set for the remaining intervals, whose cardinality must match that of the optimal stabbing set on the remaining intervals. Thus, the algorithm produces an optimal stabbing set for the n intervals, closing the induction. □

Question 2:

Consider the following algorithm.

The Stoogesort algorithm (by Jeff Erickson)

```
Sort(A):
    Stoogesort(A, 1, length(A))

Stoogesort(A, i, j):
    if j - i = 1 and A[i] > A[j]:
        Swap(A, i, j)
    if j - i > 1:
        t = floor((j - i + 1)/3)
        Stoogesort(A, i, j - t)
        Stoogesort(A, i + t, j)
        Stoogesort(A, i, j - t)

Swap(A, i, j):
    temp = A[i]
    A[i] = A[j]
    A[j] = temp
```

Does the algorithm work correctly? If so, prove it. If not, find a list which it does not sort.

Yes, the algorithm produces a correct sort.

Proof. We argue by induction. The base cases of 1 element and 2 elements are trivially satisfied, where the latter comes from the Swap performed immediately. Thus, assume that for some n, Stoogesort correctly sorts arrays of size less than n. Consider an array of size n.

We have on the first call of Stoogesort that $i = 1$ and $j = n$, so $j - 1 > 1$, and the three recursive calls are performed. The first Stoogesort is called on the subarray $A[1 : n - t]$ and sorts it by the IH. After this sort, we know the top t largest elements lie within the range $A[1 + t, n]$, since

- If a largest element was in $A[n-t+1, n]$, it remains untouched and still lies within $A[1+t, n]$
- The worst case is if all t largest elements were in $A[1: n-t]$. In which case, they would get sorted to the t top positions in this range, i.e. $A[n-2t+1: n-t]$. This is a subset of $A[1+t: n]$, since

$$\begin{aligned} t = floor(n/3) &\leq n/3 \\ \implies 3t &\leq n \\ \implies 1+t &\leq n-2t+1. \end{aligned}$$

The second call on $A[1+t, n]$ therefore moves the top t largest elements into the t rightmost positions, $A[n-t+1: n]$, where they are also sorted, by the IH. Finally, the third call on the range $A[1: n-t]$ sorts the (now) lowest $n-t$ elements in A, by the IH. Thus, all of A gets sorted. □

Question 3:

You are trying to focus a microscope. Assume that there are n focus settings on a dial, one of which has the least blurriness. If you turn the dial too far forward or too far back from that setting, the image will become increasingly blurry. Design an algorithm to efficiently find the setting with the best focus. Mathematically, represent the settings by $1, \ldots, n$. There is a function Blurriness(i) that reports an integer representing the blurriness of the image when the knob is at setting i. There is a unique number $k \in \{1, 2, \ldots, n\}$ that minimizes this function. The function Blurriness(i) is strictly decreasing for $i \leq k$, and strictly increasing for $i \geq k$. Design an algorithm to find k. For full credit, this algorithm should run in $O(\log n)$ time.

Write a base case that chooses the index with the least Blurriness for arrays of size 2 or less. For larger arrays, divide and conquer with a bisection search. Take the midpoint and compare its blurriness to its right neighbor. If midpoint is larger, call the algorithm on the neighbor to the right end. If the midpoint is smaller, call the algorithm on the left end to the midpoint. Return the resulting index.

```
BisectionSearch(low, high):
    if high - low == 0:
        return low
    if high - low == 1:
        if Blurriness(high) > Blurriness(low):
            return low
        else:
            return high

    mid = floor((low+high)/2)
    if Blurriness(mid) < Blurriness(mid + 1):
        return BisectionSearch(low, mid)
    else:
        return BisectionSearch(mid+1, high)
```

The runtime is $O(\log n)$.

Proof. For $n \geq 3$, the algorithm performs a constant amount of work (compute mid, compare Blurriness of mid and neighbor, and assuming Blurriness computation cost is independent of n) and performs at most one recursive call on half of the input. Thus, the cost recursion is

$$T(n) = T(n/2) + O(1),$$

which by the Master Theorem, gives $O(\log n)$. □

This algorithm is gives the index with the lowest blurriness.

Proof. We argue by induction. The base cases for $n = 1, 2$ are trivially satisfied. Thus, assume the algorithm returns the lowest blurriness index for inputs of size less than n. Consider an input of size n. Since Blurriness is unimodal across the indices:

- If Blurriness(mid) ¡ Blurriness(mid + 1), Blurriness is strictly increasing across these two indices, so the global minimum must be within $1, 2, \ldots,$ mid.
- Likewise, if the reverse inequality holds, Blurriness is strictly decreasing across these two, so the global minimum must be to the right of mid.

Either way, the recursive call is called on an input of size less than n which contains the global minimum, so by the IH, the global minimum is returned, and we close the induction. □

Question 4:

Given a list of length n, find all elements of L that occur in L strictly more than $n/4$ times. You cannot assume that the elements of L are ordered (so you can't sort them); similarly, you should not assume that you have access to any kind of hash map. The only operation you have access to is checking if two elements are the same, which can be done in $O(1)$ time. For full credit, your algorithm should run in $O(n \log n)$ time.

Set up a base case for 1 element lists that simply returns the element. Then, for larger size lists, divide and conquer. Split the list into halves and call the algorithm on each, returning for each half all the elements that occur strictly more than $(n/2)/4$ times within those halves. Then, combine their results, and perform a linear scan for each of the candidates to see if it appears strictly more than $n/4$ times in the whole list.

```
FindElements(L):
    n = len(L)

    if n == 1:
        return [L[1]]

    mid = floor(n/2)
    L1 = FindElements(L[1:mid])
    L2 = FindElements(L[mid+1:n])
```

```
combined = L1.append(L2)

for candidate in combined:
    tally = 0
    for i = 1, 2, ..., n:
        if L[i] == candidate:
            tally++
    if 4 * tally <= n:
        Remove candidate from combined
return combined
```

This algorithm runs in $O(n \log n)$ time.

Proof. For the run of the algorithm, it recursively calls itself twice on half its input size. Since there can also be at most 3 elements within a list with strictly more than $n/4$ occurrences, $L1$ and $L2$ combined have at most 6 unique candidates, a constant amount. For each of these candidates, the algorithm performs a linear scan, and checks if it exceeds $n/4$ count, for a total additional cost of $O(n)$. Thus, the timecost recursion is

$$T(n) = 2 * T(n/2) + O(n)$$

which is balanced and by the Master Theorem, gives $O(n \log n)$ runtime. □

This algorithm finds all elements of L with strictly more than $n/4$ occurrences.

Proof. We argue by induction. The base case for $n = 1$ is trivially satisfied, since the algorithm returns the single element. Thus, assume the algorithm correctly returns the elements that appear more $k/4$ times for lists of size $k < n$. Consider a list of size n. The algorithm calls itself on $L[1{:}\, mid]$ and $L[mid + 1{:}\, n]$.

We first claim that if an element, say x, appears more than $n/4$ times in L, it is in either $L1$ or $L2$. Assume that it is not in $L1$. Let $N1$ and $N2$ be the number of occurrences of x i the subarrays $L[1{:}\, mid]$ and $L[mid + 1{:}\, n]$, respectively. Thus, $N1 \leq floor(n/2)/4$. Thus, we see

$$\begin{aligned} & N1 + N2 > n/4 \\ \implies & N2 > n/4 - N1 \\ \implies & N2 > [n - floor(n/2)]/4. \end{aligned}$$

This, since $L[mid + 1{:}\, n]$ has $n - floor(n/2)$ elements, by the IH, x must be included in $L2$ by the recursive call.

The algorithm then loops over these candidates. By a standard loop invariant argument over the inner loop (i), we can show that tally counts the number of times each candidate shows up in L, and if it doesn't exceed $n/4$, it is removed. Thus, the algorithm indeed returns all elements that exceed more than $n/4$ occurrences, and we close the induction. □