Problem Set 2

All parts are due Thursday, September 20 at 11PM.

Name: Kevin Jiang

Collaborators: Name1, Name2

Problem 2-1.

(a) WLOG, assume that $e_0 = A[k]$ in $A_0 = A[0 : n/2]$ is the smaller endpoint. If there exists an $A[j]$ greater than e_0, then its index lies either in $0 \leq j \leq k$ or $k < j \leq n/2$. If j lies in the former case, then $A[j]$ to e_1 is a longer zipline than e_0 to e_1, which is a contradiction. If j lies in the latter case, then Kevel's zipline journey will prematurely decelerate. Therefore, e_0 is the largest element in A_0. Since the case for e_1 is symmetrical to e_0, $e_i = \max(A_i)$. $\square$

(b) We propose the following efficient divide-and-conquer algorithm to find Kevel the zipline of maximum possible length in a given input array $A[0 : n]$ of n distinct integers:

Given an array of n heights, the maximum possible length zipline is either in the first half ($A[0 : n/2]$), the second half ($A[n/2 : n]$), or includes the middle element $A[n/2]$ with at least one element from both halves. In the first two cases, we can recursively call the original algorithm with its corresponding half. In the third case, we search for the maximum element in the first half (e_0) and the maximum element in the second half (e_1). For both e_0 and e_1, we do a linear search to the right and left end, respectively, and compare to find the maximum zipline length.

This algorithm is correct because there are only three possible cases for where the longest zipline length can lie. Note that any two consecutive heights in the array can be ziplined, so a longest zipline length has to exist. The recursive method will split the array in half each time and end in the base case of three elements, where the longest length is either the first two elements, the last two elements, or all three elements. The third case returns the longest length because of the fact that one endpoint must be a maximum of one half of the array (proved in part (a)). By doing a linear search for the longest zipline for each max, we are guaranteed to find the longest length.

The algorithm has a running time of $O(n \log n)$. In the first two cases of the algorithm, we split the array of n elements into two arrays of length $n/2$. Thus, the amount of work needed for these two cases is $2T(n/2)$. In the third case, we search for the largest element in both halves and then perform linear searches to compare, which is

$O(n/2) + O(n/2) + O(n) + O(n) + O(1) = O(n)$ work. Therefore, the recurrence relation is $T(n) = 2T(n/2) + O(n)$. By the Master Theorem, we have $n^{\log_2 2} = n$, which is the second case of the Theorem. Hence, the runtime is $O(n \log n)$. $\square$

Problem 2-2.

The algorithm is as follows:

For each of the n panels, add the heights (h_i, h_{i+1}) into a list as individual lists of length two. The second element should indicate if it was a "starting height" (1) or "ending height" (-1) in the panel:

$$[(h_1, h_2), (h_3, h_4), \cdots (h_{2n-1}, h_{2n})] \Rightarrow [[h_1, 1], [h_2, -1], \cdots, [h_{2n-1}, 1], [h_{2n}, -1]]].$$

Apply merge sort to sort the heights in increasing order based on h_i. Now, keep a running counter c that adds one to c if h_i is a starting height or subtracts one if it is an ending height. The instantaneous count at height h_i is then appended to the end of $[h_i, (1 \text{ or } -1)]$. We can loop through the $2n$ endpoints again, combine consecutive two terms into one interval (a, b), and create a tuple with the interval and the current running count c at the height a. We can then loop over the intervals and delete zero length intervals.

This algorithm is correct since it correctly defines the intervals and associates their respective number of panel shadings. Since every interval is due to a panel starting or ending, by breaking the heights (h_i, h_{i+1}) of each panel, we get every possible starting and ending point for the intervals. Sorting the list of points gives us the intervals (by taking consecutive terms) that we want. The method of using a running count keeps track of the number of panel shadings accurately because every panel is either in or out of the interval. By adding one to c, we are indicating that a panel is now in this interval. The count c will keep this panel until it has ended at some height h_i, at which c will subtract one. Therefore, the count c will always indicate the number of panels whose height range lies in the corresponding interval. The special case where panel edges meet is accounted for since we are removing the zero length intervals after we apply the running count.

The amount of work needed for this algorithm is $O(cn + n \log n) = O(n \log n)$ since we loop through the $2n$ points three times (add the starting/ending heights to a list, append the running count to each height, and create a tuple for the formatting requested in the problem), delete zero length intervals, and apply merge sort ($O(n \log n)$ algorithm).

Problem 2-3. In a regular dynamic array, the insertion and removal of items at the back of the array can be done in amortized constant time by allocating a space of $2n$ elements to store n items. Therefore, to achieve amortized constant time for insertion/removal at the front and the back of the array, we can allocate a space of $3n$ items, where the n items are stored in the middle third. In other words, we have n empty items in the front and back of our array. We can use a pointer to indicate the first element of our array.

This data structure supports worst-case constant time index lookup since we can do an $O(1)$ arithmetic operation on the pointer to find the desired index. It also supports amortized constant time insertion at both ends since we only resize when $n = 3^i$, which is $\Theta(1 + 3 + 9 + 27 + \cdots + n) = \Theta(n) = O(1)$ amortized for n insert rights. For removal, whenever we have $n < \text{size}/9$, we can resize to $\text{size}/3$. Hence, it is $O(1)$ amortized for both insert/remove-left/right.

Problem 2-4.

(a) Suppose we have two points q_i, q_j such that$||q_i - q_j|| < \delta$. WLOG let $i < j$. Note that the x-coordinates are bounded by the width of the vertical strip, and the y-coordinates are at most δ apart. Thus, consider the box

$$B = \{(x, y)|x \in [x^* - \delta, x^* + \delta], y \in [q_i[1], q_i[1] + \delta]\}.$$

We must have $q_j \in B$. Moreover, $q_k \in B$ for $i \leq k \leq j$ due to the y-coordinate. We claim that at most 10 points q_k could lie in the box.

Suppose we split the box into two squares of side length δ. In each half, the points must be at least δ apart or else δ is no longer $\min(dL, dR)$. Consider each point as a circle with radius $\delta/2$. The minimum area any point will contribute to the square is then just a quarter circle of area $\frac{1}{4} \cdot \pi \left(\frac{\delta}{2}\right)^2 = \frac{\pi\delta^2}{16}$ (a corner point). Since the area of the square is δ^2, the maximum number of points that can fit is $\dfrac{\delta^2}{\frac{\pi\delta^2}{16}} \approx 5$. Therefore, the maximum number of points that can fit into B is 10. Note that 10 points is a rather loose upper bound, but we have shown that $|i - j|$ is at most a fixed constant.

(b) In the first two cases, we can recursively call the function, so the total work is $2T(n/2)$. In the third case, we only need to check at most $\binom{10}{2}$ pairs of points for $n-9$ sets of 10 consecutive points. Thus, the work needed is $45n - 405 = O(n)$. Since we are in the balanced case of the Master Theorem, the running time for $\boxed{T(n) = 2T(n/2) + O(n)}$ is $\boxed{O(n \log n)}$. The work needed to sort the points is also $O(n \log n)$, so nothing changes.

(c) Submit your implementation to `alg.mit.edu/PS2`