Problem Set 2

All parts are due on September 20, 2018 at 11PM. Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on the course website, and any code should be submitted for automated checking on `alg.mit.edu`.

Problem 2-1. [25 points] **Zipline of Death**

For his upcoming beyond-the-grave performance, Kevel Enievil wants to build an awesomely long zipline in the 6.006 mountain range. We represent the heights of the mountains in this range as an array of n distinct integers, $A[0] \dots A[n-1]$. Assume that these heights are all *distinct*, and that n is a power of 2.

Kevel needs to decide on two mountains as the endpoints of the zipline. To avoid premature deceleration, he needs to ensure that the endpoint mountains are higher than any mountains in between. Precisely, in the array A, he must find a contiguous subarray $A[i] \dots A[j]$ for integers i, j ($0 \le i \le j < n$) such that $A[i]$ and $A[j]$ are larger than all elements $A[k]$ strictly in between ($i < k < j$). Note that subarrays of size 2 or less automatically satisfy this criterion (it would just not be a very impressive stunt). Furthermore, subarrays of size > 2 allow Kevel to zipline if and only if their largest two elements are the two endpoints.

For example, if the array is

$$[4, 1, 2, 5, 3, 7, 6]\,,$$

Kevel can build a zipline from 4 to 5 (having length 4) or a zipline from 5 to 7 (having length 3).

In this problem, you'll develop a divide-and-conquer algorithm to find Kevel a zipline of maximum possible length.

(a) Suppose you split array A into two halves, $A_0 = A[0 : n/2]$ and $A_1 = A[n/2 : n]$, and suppose that the longest zipline has one endpoint e_0 in A_0 and one endpoint e_1 in A_1. Let $e_i = \min(e_0, e_1)$ be the smaller endpoint. Prove that $e_i = \max(A_i)$.

(b) Describe an 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. (For any algorithm you describe in this class, you should **argue that it is correct**, and **argue its running time**.) For full points, your algorithm should have a worst-case running time of $O(n \log n)$.

Problem 2-2. [20 points] **Glass Shadows**

Cale Dhihuly has created a new form of art called ShadowGlass! Panels of glass having the same width and varying heights are mounted vertically, sticking out of a wall, and the shadows cast on the opposite wall (by perfectly horizontal light) are determined by how many panels are present at each height. See Figure 1. While finalizing the design, Cale wants a method to compute these shadow intervals and the number of panels shading each interval, since this distribution determines the aesthetic appeal of the installation.

Given an input array of panels, where a panel $p = (h_1, h_2)$ extends from height h_1 to height $h_2 > h_1$, Cale wants an efficient algorithm to produce the set of shadow intervals created by the panels, along with the number of panels shading each interval. An element of the output $((a, b), k)$ would correspond to a shadow interval from height a to height b (where $a < b$) shaded by k panels. In the example below, the panels are specified by height intervals $[(3, 9), (0, 6), (4, 7)]$, and the shadow intervals—together with the number of panels shading each—may be described as:

$$[((0, 3), 1), ((3, 4), 2), ((4, 6), 3), ((6, 7), 2), ((7, 9), 1)].$$

This toy example is easy enough to compute by hand, but Cale would prefer not to process a longer lists of panels manually. Design an $O(n \log n)$ algorithm to help Cale determine the number of overlapping panels at each shadow interval given an input of n panels. **Hint:** use sorting!

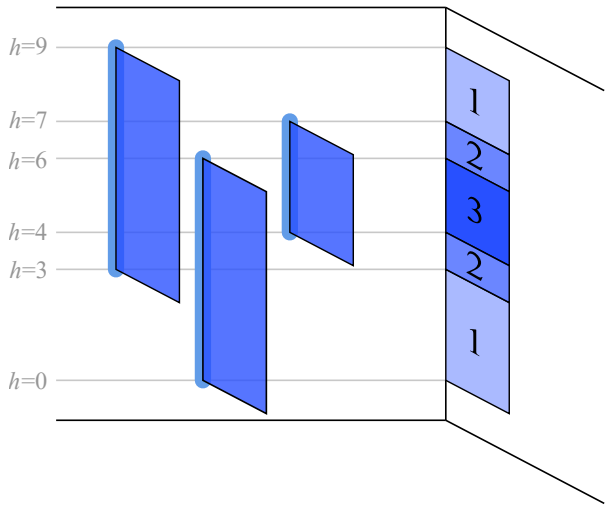

Figure 1: An example of Cale Dhihuly's ShadowGlass art installation.

Problem 2-3. [15 points] **Double-Ended Sequence Operations**

The dynamic array data structure supports worst-case constant-time indexing—the element in slot i may be located in $O(1)$ time for any $0 \leq i <$ length—as well as insertion and removal of items at the back of the array (the largest index) in amortized constant time. However, insertion and deletion at the front of a dynamic array (at index 0) are not efficient: every entry must be moved over to maintain the sequential order of entries, taking linear time.

On the other hand, the linked-list data structure supports insertion and removal operations at both ends in worst-case constant time, but at the expense of linear-time indexing.

Show that we can have the best of both worlds: design a data structure to store a sequence of items that supports **worst-case** constant time index lookup, as well as **amortized** constant time insertion and removal at both ends. Your data structure should use $O(n)$ space to store n items.

Problem 2-4. [40 points] **Closest Pair of Points**

Given a set P of points $\{p_0, p_1, \ldots, p_{n-1}\}$, where p_i has coordinates (x_i, y_i), we wish to find the pair (p_i, p_j) of points in P whose squared Euclidean distance $\|p_i - p_j\|^2 = (x_i - x_j)^2 + (y_i - y_j)^2$ is the smallest. The naïve brute-force solution to this problem is to compute the distance between all $\binom{n}{2}$ pairs of points, and take the minimum, which takes $\Theta(n^2)$ time.

In this problem, we will do better via the following divide-and-conquer algorithm. First, we sort the points by their x coordinates. Next, we divide this sorted array into left and right halves of size $n/2$. Say the left half has all the points with x coordinates $< x^*$ (where x^* is the median x coordinate) and the right half has all the points with x coordinates $\geq x^*$. The closest pair of points must fall into one of three cases:

1. both points are in the left half;
2. both points are in the right half; or
3. one point is in the left half and one point is in the right half.

To handle the first two cases, the algorithm recursively calls itself on each half. To handle the third case, observe that we only care about finding the closest pair of points, so we only care about finding points that are closer than the best pairs found by the recursive calls on the left or right halves. Let d_L and d_R be the distances between the closest pair of points in the left and right halves respectively, and let $\delta = \min(d_L, d_R)$. If there is a closer-than-δ pair containing one point from each half, then both points must lie within a width-2δ vertical strip centered on the dividing vertical line $x = x^*$ separating the two halves; see Figure 2.

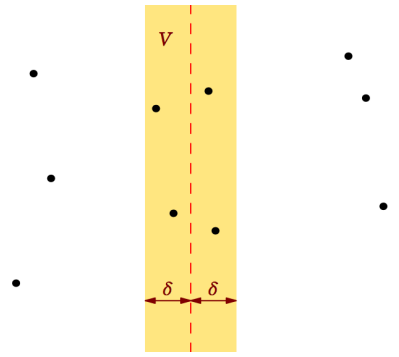

Figure 2: It suffices to consider a vertical strip V of width 2δ centered at $x = x^*$.

To find the closest point pair within this vertical strip, we first identify the points within the strip by a linear scan, checking each point for an x coordinate in $[x^* - \delta, x^* + \delta]$. In subproblem **(a)**, we show that we can find the closest pair of points within the vertical strip by checking the distance between only a linear number of point pairs!

(a) [10 points] Suppose we sort the k points in the vertical strip by y coordinate, so that the points in y-order are $(q_0, q_1, \ldots, q_{k-1})$. Show that, if $\|q_i - q_j\| < \delta$ (a pair of points are closer than a pair we've found so far), then $|i - j|$ is at most some fixed constant m (the pair appear within m places of each other in the y-coordinate order).

(b) [5 points] What is the recurrence and running time of the algorithm described above? (Note that the algorithm described is not optimal. You **do not** need to improve upon the algorithm for full points, but you may do so if you are feeling adventurous!)

(c) [25 points] Write a Python function `closest_pair(points)` that returns the smallest **squared Euclidean distance** between any pair of input points, following the algorithm above. You may assume that there are at least two points, and that no pair of points have the same x or y coordinate. You can download a code template containing some test cases from the website. You are allowed to use any built-in Python sort functionality. Submit your code online at `alg.mit.edu`.

```
def squared_distance(p, q):
    '''Returns the squared distance between points p and q'''
    (px, py), (qx, qy) = p, q
    return (px - qx)**2 + (py - qy)**2

def closest_pair(points):
    '''
    Input:  points | tuple of at least 2 points of the form (x, y)
    Output: smallest squared distance between any pair of points
    '''
    ##################
    # YOUR CODE HERE #
    ##################
    return 0
```