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, $A[0] \ldots A[n-1]$. 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 $A[i] \ldots A[j]$ 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 $e_i = \max(A_i)$.

Solution: Without loss of generality, assume $i = 0$. If $e_0 \neq \max(A_0)$, then we have two cases:

- If $\max(A_0)$ were to the right of e_0, then (e_0, e_1) wouldn't be a valid zipline.
- If $\max(A_0)$ were to the left of e_0, then we could make a longer zipline by moving e_0 to the left to the next larger value e_0' in A_0 (which exists because $\max(A_0)$ is to the left). The new zipline (e_0', e_1) would still be a valid zipline because the added elements strictly between e_0' and e_1 are $\le e_0 < e_1$.

Thus, in either case, we have a contradiction, so in fact $e_0 = \max(A_0)$ as claimed.

Rubric:

- 10 points total
- 5 points for case where $\max(A_0)$ is to the right
- 5 points for case where $\max(A_0)$ is to the left

(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 $O(n \log n)$.

Solution: The largest zipline subarray is either entirely within A_0, entirely within A_1, or it spans across the middle. The hardest case is the last, when the zipline has endpoints in both A_0 and A_1. In this case, we know from part (a) that one of the endpoints is $\min\big(\max(A_0), \max(A_1)\big)$.

Algorithm:

1. If $|A| \leq 2$, return A.
2. Split A into two halves, A_0 and A_1.
3. Recursively solve the problem within A_0 and within A_1.
4. At every level of recursion, check for a zipline crossing over the middle.
   - Calculate $\max(A_0)$ and $\max(A_1)$.
   - Suppose that $\max(A_i) = \min\big(\max(A_0), \max(A_1)\big)$.
   - Start at the middle of A and walk linearly through the other half, A_{1-i}. Stop when you reach the first element bigger than $\max(A_i)$ (e.g., $\max(A_{1-i})$).
5. Compare the two recursive solutions to the crossing solution, and return the one of largest length.

Correctness. We consider all possible locations for the longest zipline: within A_0, within A_1, or spanning both. In the first two cases, we find the correct answer by induction on $|A|$. In the last case, we know from part (a) that $\max(A_i)$ is one of the endpoints. We therefore cannot include any elements $> \max(A_i)$ strictly within the zipline, so we must stop when we reach such an element. Conversely, all elements we include strictly within the zipline are less than $\max(A_i)$ which is less than the other endpoint, so the zipline is valid.

Running time. At each recursive step, we recurse on two subproblems of half the size, and spend $O(n)$ time to find the two maxima and to walk linearly through one half. Thus we obtain the recurrence

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

By Case 2 of Master Theorem (as in merge sort), this solves to $T(n) = O(n \log n)$.

Rubric:

- 15 points total
- 8 points for the algorithm
- 4 points for correctness argument
- 3 points for running time argument

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!

Solution: Take the start and end point of each interval, and sort these indices, maintaining the additional information of whether each index is a start or end. Iterate through the indicies consecutively, maintaining a counter of the number of open intervals. When a start index is encountered, the counter is incremented by 1, and when an end index is encountered, the counter is decremented by 1. The value of the counter between the processing of each pair of indices is the number of overlapping pieces in the interval, since it includes all the pieces starting before the index (corresponding to each increment of the counter) but not the pieces also ending before the index (corresponding to each decrement of the counter).

Sorting the $2n$ indices takes $O(n \log n)$ time, and iterating through the $2n$ indices and outputting the number of overlapping pieces at each interval takes $O(n)$, so the algorithm runs in $O(n \log n)$ time.

Rubric:

- 5 points for applying sorting

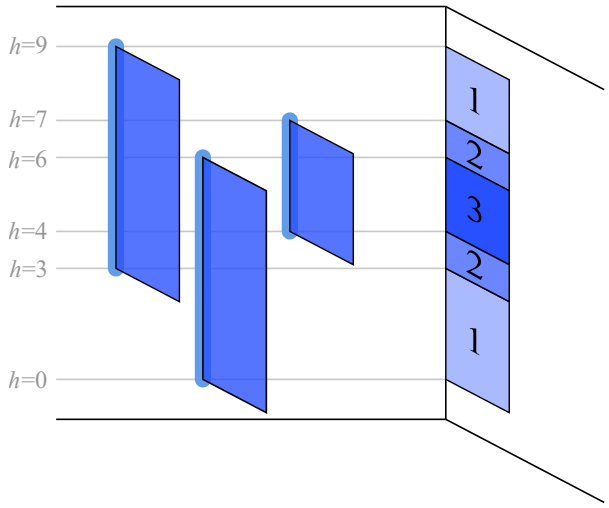

Figure 1: An example of Cale Dhihuly's ShadowGlass art installation.

- 10 points for correctness analysis
- 5 points for correct running time analysis
- Partial credit may be awarded

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.

Solution: There are many possible solutions. One solution modifies the two-stack queue implementation described in lecture, where care needs to be taken when popping from an empty stack, or pushing to a full stack. An alternative approach would be to store the queued items in the middle of an array rather than at the front, leaving a linear number of extra slots at both the beginning and end whenever rebuilding occurs, guaranteeing that linear time rebuilding only occurs once every $\Omega(n)$ operations.

For example, whenever reallocating space to store a sequence of n elements, copy them to the middle of a length $m = 3n$ array. To insert or remove an item to the beginning or end of the

sequence, add or remove an element at the appropriate end in constant time. If no free slot is exists during an insertion, at least a linear number of insertions must have happened since the last rebuild, so we can afford to rebuild the array. If removing an item brings the ratio n/m of items to array size to below $1/6$, at least $m/6 = \Omega(n)$ removals must have occurred since the last rebuild, so we can again afford to rebuild the array.

A linear number of operations between expensive linear time rebuilds ensures that each dynamic operation takes at most amortized $O(1)$ time. To support array indexing in constant time, we maintain the index location i of the left-most item in the array and the number of items n stored in the array, both of which can be maintained in worst-case constant time per update. To access the jth item stored in the queue sequence using zero-indexing, confirm that $i + j < n$ and return the item at index $i + j$ of the array container in worst-case constant time.

Rubric:

- 9 points for data structure description
- 4 points for amortized analysis for dynamic operations
- 2 points for constant time indexing
- Partial credit may be awarded

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.

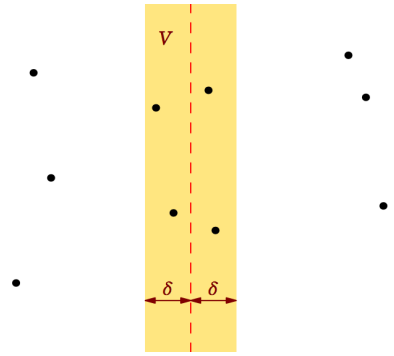

Figure 2: It suffices to consider a vertical strip V of width 2δ centered at $x = x^*$.

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).

Solution: Consider points q_i and q_j for which $\|q_i - q_j\| < \delta$. Then q_i and q_j must both reside in a rectangle R that has height δ and width 2δ, symmetric about the halfway dividing line. How many points could exist within this rectangle? All points within the left half of the rectangle (which is a $\delta \times \delta$ square) are at least δ away from each other, as are all points within the right half of the rectangle. We can pack at most four points within a $\delta \times \delta$ square subject to no pair being closer than δ from each other: specifically, positioned at the four corners of the square. Thus, at most eight points exist in R, four in each half, which must appear consecutively (in some order) in the y-order. Thus $|i - j| \leq 7$. See Figure 3.

Another way to prove a constant bound on m is an area argument. As above, each $\delta \times \delta$ square on either side of the dividing line contain only points that are at least distance δ from each other. This means that we can draw a disk of radius $\delta/2$ centered on each point in the square, and these disks are guarenteed not to overlap. Each point's disk overlaps a non-zero area of the square which contains its center. In fact, the smallest area a point's disk must overlap the square is when the point is located at the corner of the square, overlapping a quarter of the area of the disk, $\pi(\delta/2)^2/4 = \pi\delta^2/16$. Since the square has area δ^2, and each point's disk overlaps at least $\pi\delta^2/16$ of its area, there

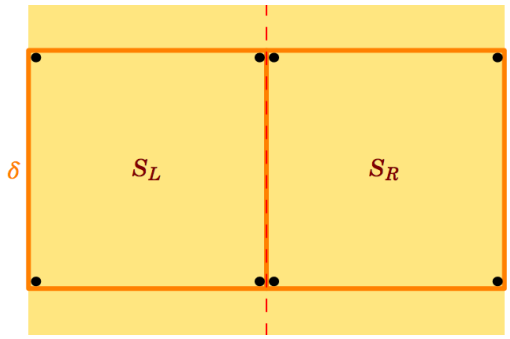

Figure 3: Bounding squares

can be at most $16/\pi < 6$ points in either square. Thus at most 10 points may exist in the $2\delta \times \delta$ rectangle, so $|i - j| \leq 9$. This is a weaker bound than discussed above, but is more rigerous and is sufficient to prove the claim.

Rubric:

- 3 points for bounding the area to consider
- 7 points for proving a specific constant m.
- Partial credit may be awarded.

(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!)

Solution: Because we sort at every recursive step, we have:

$$T(n) = 2T(n/2) + O(n \log n).$$

By Case 2 of the Master Theorem, this solves to $O(n \log^2 n)$.

Rubric:

- 3 points for recurrence
- 2 points for runtime analysis

(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`.

Rubric:

- This part automatically graded at `alg.mit.edu/PS2`.

Solution:

```
def squared_distance(p, q):
    (px, py), (qx, qy) = p, q
    return (px - qx)**2 + (py - qy)**2

def base_cases(points):
    if len(points) == 2:
        (p1, p2) = points
        return squared_distance(p1, p2)
    if len(points) == 3:
        (p1, p2, p3) = points
        return min(squared_distance(p1, p2),
                   squared_distance(p2, p3),
                   squared_distance(p3, p1))
    return None

def closest_pair_strip(strip, dsq):
    strip.sort(key = lambda p: p[1])        # O(n log n) Sort by y
    for i in range(len(strip)):             # O(n)
        for j in range(i + 1, len(strip)):  # O(1) Loop
            p, q = strip[i], strip[j]
            if (p[1] - q[1])**2 >= dsq:     # stop when y seperation > dsq;
                break                       # by (a), this will occur quickly
            dsq = min(dsq, squared_distance(p, q))
    return dsq

def closest_pair_sorted_x(points):
    if len(points) < 4:
        return base_cases(points)
    mid = len(points) // 2
    dsq = min(closest_pair_sorted_x(points[:mid]),
              closest_pair_sorted_x(points[mid:]))
    mid_x = points[mid][0]
    strip = [p for p in points if (p[0] - mid_x)**2 < dsq]
    return min(dsq, closest_pair_strip(strip, dsq))

def closest_pair(points):
    points.sort(key = lambda p: p[0])       # O(n log n) Sort by x
    return closest_pair_sorted_x(points)    # T(n)
```