CS120: Intro. to Algorithms and their Limitations Hesterberg & Vadhan

Problem Set 2

Harvard SEAS - Fall 2022 *Due: Wed Sep. 21, 2022 (11:59pm)*

Your name: Nicole Chen
Collaborators: Prin Pulkes, Jayden Personatt, Sid Bharthulwar
No. of late days used on previous psets: 0
No. of late days used after including this pset: 0

Please review the Syllabus for information on the collaboration policy, grading scale, revisions, and late days.

1. (reductions) The purpose of this exercise is to give you practice formulating reductions and proving their correctness and runtime. Consider the following computational problem:

Input	: Points $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ in the $\mathbb{R}^2$ plane that are the vertices of a convex polygon (in an arbitrary order) whose interior contains the origin
Output	: The area of the polygon formed by the points

Computational Problem AreaOfConvexPolygon

(a) Show that AreaOfConvexPolygon$\leq_{O(n),n}$ Sorting. Be sure to analyze both the correctness and runtime of your reduction.

In this part and the next one, you may assume that a point $(x, y) \in \mathbb{R}^2$ can be converted into polar coordinates (r, θ) in constant time.

You may find the following useful:

- The polar coordinates (r, θ) of a point (x, y) are the unique real numbers $r \geq 0$ and $\theta \in [0, 2\pi)$ such that $x = r\cos\theta$ and $y = r\sin\theta$. Or, more geometrically, $r = \sqrt{x^2 + y^2}$ is the distance of the point from the origin, and θ is the angle between the positive x-axis and the ray from the origin to the point.
- The area of a triangle is $A = \sqrt{s(s-a)(s-b)(s-c)}$ where a, b, c are the side lengths of the triangle and $s = \frac{a+b+c}{2}$ (Heron's Formula).

Solution

We first describe the algorithm.

- Given an array of points P, check the size of the array. If $n \leq 2$, return $\perp$
- For each $i \in [n-1]$, convert each point in P to their polar form and store that information in a new array P' such that $P' = (\theta_0, (r_i, (x_0, y_0))), ..., (\theta_{n-1}, (r_{n-1}, (x_{n-1}, y_{n-1})))$. θ_i, $\theta \in [0, 2\pi)$, for $P[i]$ is the angle between the positive x axis and the ray from the origin to the point. r_i is the length of the ray from the origin to the point.
- Sort P' according to θ from the oracle call to Sorting. This should yield a sorted array called S

- Create a new variable a intended to store the sum of the areas of each miniature triangle
- For each $i \in [n-1]$, determine s_i for $S[i]$ and $S[i+1]$ respectively. s_i is the third side of the triangle, and we use the distance formula to calculate it: $s_i = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$. Then, use Heron's Formula to determine the area of each mini triangle formed by the sides r_i, r_{i+1}, and s_i. For each pairing, add the area to the running area total in a.
- After iterating through n pairings, return the running area total a.

Correctness

- We take that the input P of points are the vertices of a convex polygon whose interior contains the origin. Because the input forms a convex polygon, when each point is converted to its respective polar coordinates and sorted according to its angle to the x-axis, we can draw edges connecting each point to the next (based on the sorted order). Doing so will produce a convex polygon.
- Every convex polygon centered at the origin with n vertices is comprised of n triangles with a point at the origin and two adjacent vertices from the original polygon such that the sum of the area of these n triangles equals the area of the polygon. This is because for each vertex in the polygon, we can draw a line connecting that point to the origin (r_i). Therefore, each triangle is comprised of the two rays formed by connecting adjacent points to the origin. The third side is formed by connecting adjacent points together. These three sides are represented as r_i, r_{i+1}, and s_i in the above description of the algorithm.
- To determine whether two vertices are adjacent, we determine how close two points are in relation to the angle their ray creates with the origin. Therefore, the algorithm returns a sorted array S of points based on their angles such that two adjacent vertices determine a side of the original polygon as their angles are the closest in value.
- Each triangle therefore has a point at the origin and two adjacent vertices. The sides of each triangle are therefore determined by the distance between the adjacent vertices and the distances between the adjacent vertices and the origin (the ray of each vertex).
- If the algorithm returns a real number, it has summed of the area of each of the mini triangles that comprise the original polygon and therefore returns the area of the original polygon.
- If the input array has a size less than 3, the points cannot form a proper polygon (a polygon must have a size of at least 3). We have proven that we cannot return a real area A so we return $\perp$ for an array of points that cannot properly form a convex polygon.

Run Time

- Because we utilize a sorting algorithm in the computation problem AreaOfConvex-Polygon, there is a reduction such that the oracle that solves Sorting is utilized in AreaofConvexPolygon. This oracle is called once on a size of n. We therefore say that the call to `Sorting` runs $T_{sorting}(n)$ time.

- The reduction algorithm take $\mathcal{O}(n)$ time. It take $\mathcal{O}(n)$ time to determine θ_i and r_i for each point and store the angle and ray in a new array P', assuming that conversion to polar coordinates takes $O(1)$ time, so when we iterate n times, we get a run time of $O(n)$.
- It takes $O(1)$ time to create a new variable a
- It takes $\mathcal{O}(n)$ time to calculate s_i for each pairing and to sum the area of each miniature triangle per each pairing. This is because each calculation takes $O(1)$ time, so when we do this n times, we get a run time of $\mathcal{O}(n)$.
- Summing the above run time produces the aggregate $O(n)+O(n)+O(1)+T_{sorting}(n)$ which simplifies down to $O(n) + T_{sorting}(n) = O(n)$ for reduction.
- Therefore, there exists a reduction from AreofConvexPolygon to Sorting, where the reduction algorithm calls the oracle only once on an input of size n and the reduction algorithm runs in time $\mathcal{O}(n)$. Therefore, referencing **Definition 4.3 in Class Lectures**, AreaOfConvexPolygon$\leq_{O(n),n}$ Sorting.

(b) Deduce that AreaOfConvexPolygon can be solved in time $O(n \log n)$.

- We refer to our conclusions in (a). The reduction algorithm runs $O(n)$ time. If we take the Sorting Algorithm to be `MergeSort` we get a run time of $O(n)+O(n \log(n))$. Therefore, the total run time is $O(n(\log n))$.

(c) Let Π and Γ be arbitrary computational problems, and suppose that there is a reduction from Π to Γ that runs in time at most $g(n)$ and makes at most $k(n)$ oracle calls, all on instances of size at most $f(n)$. Show that if Γ can be solved in time at most $T(n)$, then Π can be solved in time at most $O(g(n) + k(n) \cdot T(f(n)))$. Note that the case $k(n) = 1$ was stated and proved in class; the case $k(n) > 1$ is useful as well, such as in Part 1d below.

- If Γ can be solved in time $O(T(n))$, then with an input of size $f(n)$, a call to the oracle will take at most $T(f(n))$ by rules of substitution $n = f(n)$.
- There are $k(n)$ oracle calls and because each oracle call takes $O(T(f(n)))$ time, $k(n)$ calls will take at most $k(n) \cdot T(f(n))$ time.
- Outside of the oracle calls, the computation problem Π runs in time at most $g(n)$ (ie. the reduction algorithm takes time $g(n)$).
- Therefore, we sum the run time of the reduction algorithm and the run time of the calls to the oracle to determine the run time of Π, where we get $O(g(n) + k(n) \cdot T(f(n)))$

(d) (*challenge; extra credit) Come up with a way to avoid conversion to polar coordinates and any other trigonometric functions in solving AreaOfConvexPolygon in time $O(n \log n)$. Specifically, design an $O(n)$-time reduction that makes $O(1)$ calls to a Sorting oracle on arrays of length at most n, using only arithmetic operations $+$, $-$, $\times$, $\div$, and $\sqrt{\ }$, along with comparators like $<$ and $==$. (Hint: first partition the input points according to which quadrant they belong in, and consider $\tan\theta$ for a point with polar coordinates (r, θ).)

- Given an array of points P, check the size of the array. If $n \leq 2$, return $\perp$

- For each $i \in [n-1]$, we group the input points according to which quadrant they belong and we calculate $\tan(\theta_i)$ where θ_i is the angle formed between the x axis and the ray connecting the origin to the point p_i. Note: we aren't actually finding θ_i and calculating $\tan(\theta_i)$ accordingly. Instead, we calculate $\tan(\theta_i)$ with $\frac{y_i}{x_i}$. To group the points according to which quadrant they belong, we look at whether x is negative or not and whether y is negative or not. We group points where x and y are positive in group 0 (Quadrant 1); we group points where x is negative and y is positive in group 1 (Quadrant 2); we group points where x is negative and y is negative in group 2 (Quadrant 3); lastly, we group points where x is positive and y is negative in group 3 (Quadrant 4). Through grouping and calculating $\tan(\theta_i)$, we get the array $P' = ((((\frac{y_i}{x_i}, x_i, y_i), (\frac{y_m}{x_m}, x_m, y_m), ...), ((\frac{y_n}{x_n}, x_n, y_n), ...), ((\frac{y_p}{x_p}, x_p, y_p), ...), ((\frac{y_q}{x_q}, x_q, y_q), ...))$ where $i, m, p, q < n$. This step takes $O(n)$ time because we iterate through all points in P to calculate $\frac{y_i}{x_i}$ and to insert each point into its respective group in P'.
- Sort P' according to $\frac{y_i}{x_i}$ according to the oracle call to Sorting, whereby in Group 1, we sort in ascending order (smallest to largest); in Group 2, we sort in descending order (largest to smallest); in Group 3, we sort in ascending order; and in Group 4, we sort in descending order. This is because $\frac{y_i}{x_i}$ increases as θ increases in the first quadrant. $\frac{y_i}{x_i}$ decreases as θ increases in the second quadrant. $\frac{y_i}{x_i}$ increases as θ increases in the third quadrant. And, $\frac{y_i}{x_i}$ decreases as θ decreases in the fourth quadrant. We make $O(1)$ calls to the Sorting oracle on an array with total number of elements n. This newly sorted array is called S. It must be noted that if the $\frac{y_i}{x_i}$ has $x_i = 0$ then that point (x_i, y_i) will be sorted at the very end of each group.
- Create a new variable a intended to store the sum of the areas of each miniature triangle created by the origin and two adjacent points.
- For each $i \in [n-1]$ in S, calculate the area of a miniature triangle with vertices at the origin, the coordinates specified at $S[i]$, and the coordinates specified at $S[i+1]$ respectively. The area can be determined by the equation $A = |\frac{1}{2} \cdot (x_i(y_{i+1}) + x_{i+1}(-y_i))|$ based on the general equation $A = |\frac{1}{2} \cdot (x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)|$. For each pairing, add the area to the running area total in a.
- After iterating through n pairings, return the running area total a. All of this takes $O(n)$ time.
- Therefore, the reduction algorithm is $O(n) + O(n) = O(n)$ time.

Similar techniques to what you are using in this problem are used in algorithms for other important geometric problems, like finding the Convex Hull of a set of points, which has applications in graphics and machine learning.

2. (augmented binary search trees) The purpose of this problem is to give you experience reasoning about correctness and efficiency of dynamic data-structure operations, on variants of binary-search trees.

   Specifically, we will work with *selection data structures*. We have seen how binary search trees can support min queries in time $O(h)$, where h is the height of the tree. A generalization is *selection* queries, where given a natural number q, we want to return the q'th smallest element of the set. So `DS.select(0)` should return the key-value pair with the minimum key among those stored by the data structure `DS`, `DS.select(1)` should return the one with the second-smallest key, `DS.select(n-1)` should return the one with the maximum key if the set is of size n, and `DS.select((n-1)/2)` should return the median element if n is odd.

   In the Roughgarden text (§11.3.9), it is shown that if we *augment* binary search trees by adding to each node v the size of the subtree rooted at v, then Selection queries can be answered in time $O(h)$.[1]

   (a) In the Github repository, we have given you a Python implementation of size-augmented BSTs supporting search, insertion, and selection, and with a stub for `rotate`. One of the implemented functions (`search`, `insert`, or `select`) has a correctness error, and another one is too slow (running in time linear in the number of nodes of the tree rather than in the height of tree). Identify and correct these errors. You should provide a text explanation of the errors and your corrections, as well as implement the corrections in Python.

   - `insert` is too slow because it runs in $O(n)$ time rather than $O(h)$ time. This is because while the function iterates and properly inserts the node based on its key in $O(h)$ time (either accesses the left or the right subtree and therefore accesses one node per level of the tree), the call to `calculate_sizes` takes $O(n)$ time because that function iterates through every node in the tree. Thus, $O(h) + O(n)$ produces a run time of $O(n)$ instead of $O(h)$. This can be corrected by simply updating the size of the tree after each recursive call by assignment (takes $O(1)$ time). There is no need to update the size of the tree when a new node is being created because initialization automatically sets size to 1.
   - `select` is incorrect because when the function recursively calls `select` on the right subtree for trees where the index is greater than the size of the left subtree, the index remains the same, not adjusting for the fact that we've essentially subtracted out the entire left subtree. In other words, there is a mismatch between the index `select` is passing in after each recursive call and the alternative index `left_size` that each recursive call uses to compare the original index against. An example of this is if we have a tree whose left subtree is of size 2. Say we are looking for the node at the third index. Because the size of the left subtree is 2, we already now that there are 3 nodes smaller than the node we are looking for (in the left subtree and the root). Therefore, when searching the right subtree, we should be looking for the smallest node there (indexed at 0), rather than the fourth smallest one there (with an index of 3). Recursive iterations of `select` on the left subtree are not impacted

[1] Note that the Roughgarden text uses a different indexing than us for the inputs to Select. For Roughgarden, the minimum key is selected by Select(1), whereas for us it is selected by Select(0).

because the index looks at the i'th smallest element of the set so it automatically looks in the smallest direction. The correction here is that when `select` is called on the right subtree, the index needs to subtract out the size of the left subtree and the size of the root node: `select(index - left_size - 1)`

(b) Describe (in pseudocode or pictures) how to extend `rotate` to size-augmented BSTs, and argue that your extension maintains the runtime $O(1)$. Prove that your new rotation operation preserves the invariant of correct size-augmentations. (That is, if every node's size attribute had the correct subtree size before the operation, then the same is true after the operation.)

```
1  Rotate(self,d,c)
     Input  : Two strings d, c where d determines the direction of the rotation and can
              either be "L" or "R" and c determines the child of T to perform the rotation
              on and can either be "L" or "R"
     Output : The root of the tree/subtree
2  if c = "L" then
3  |  s = self.left
4  |  if d = "L" then
5  |  |  r = s.right; r' = r.right; s.right = None; r.left = None;
6  |  |  self.left = r; r.right = s; s.left = r';
7  |  else
8  |  |  r = s.left; r' = r.right; s.left = None; r.right = None;
9  |  |  self.left = r; r.right = s; s.left = r';
10 else
11 |  s = self.right
12 |  if d = "L" then
13 |  |  r = s.right; r' = r.left; s.right = None; r.left = None;
14 |  |  self.right = r; r.left = s; s.right = r';
15 |  else
16 |  |  r = s.left; r' = r.right; s.left = None; r.right = None;
17 |  |  self.right = r; r.right = s; s.left = r';
18 temp = r.size;
19 r.size = s.size;
20 s.size = s.size - temp + r'.size;
21 return self
```

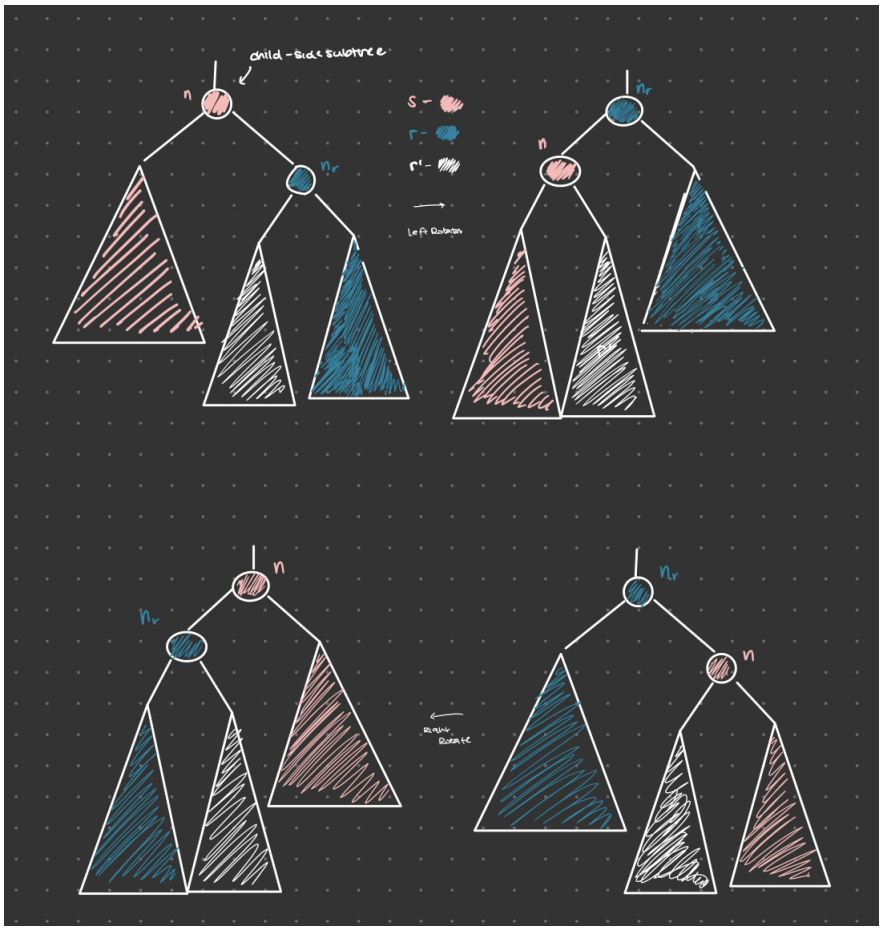
Algorithm 1: Rotate

Description of Algorithm

- Consider three trees s, r, r' where s is the child-side subtree of the root with its right or left subtree removed (the right subtree is removed if we are rotating left and the left subtree is removed if we are rotating right). The node n is the root of s and is where `rotate` moves around. In the diagram below, the entirety of s is colored in pink. The subtree that rotates into the n's place is called r with root node n_r. r does not have a subtree in the direction of the rotation. Instead, the subtree of r in the direction of the rotation is called r' because we removed r' from r such that r' and r are disjoint trees. In the diagram below, r is represented in blue while r' is

represented in white. The initializations for all these subtrees is documented in the pseudocode above on Lines 5, 8, 13, and 16. It's important to note that n and n_r are not initialized in the code because of redundancy. We use s, r to denote n and n_r. However, the difference is that n and n_r reference all child nodes underneath it while s and r only reference the child nodes as shown in the diagram below. This was done through re-assignment as shown in the pseudo code. Thus, when we refer to $s.size$ or $r.size$, we are referring only to the size of the tree rooted at s or r after reassignments have been made (shaded in the diagram).

- When we rotate the tree, n_r, the root of r, takes the place of n through re-assignment and the subtree of r opposite of the direction that we rotate moves up the tree with r. The subtree of r in the direction that we rotate (r') becomes a subtree of s, and s becomes a subtree of r in the direction that we rotate. Rotation is documented in the pseudocode above on Lines 6, 9, 14, and 17. The diagram below also showcases this. It's important to note that after rotation, s and r still refer to the same subtrees; however, n and n_r have different child nodes and subtrees.
- After rotation, we update the size of n and n_r through referencing the sizes of s, r, and r'. This is further elaborated in the **Size Preservation** section below.



Run Time

- The run time for `rotate` is $O(1)$ because operations such as assignment and summation are called as shown through the pseudo code above. Therefore, although there are multiple basic operations called multiple times (c times), we would have a run time of $O(1)$ still.

Size Preservation

- Assuming that before rotation, every node within the BST had its size correctly updated, we prove that the operation preserves the invariant of correct size-augmentations. Because we know that before rotation, every node with the BST had its size correctly updated, all nodes beneath n_r and n have their size correctly updated even after rotation because none of the nodes had children or subtrees altered. Therefore, our proof centers on the question of whether the size of n_r and n correctly reflect correct size-augmentations.
- We first calculate the size of n_r after rotation ($size(n_{r,rotated})$), by dividing the tree r into its two subtrees after rotation. The subtrees we will be using to analyze the size of n_r and n are s, r, and r'. On one end of the branch is the subtree s plus r'. On the other side is one of r's original subtrees (this subtree is unchanged because `rotate` moves r in one direction so the subtree in the opposite direction remains the same). Therefore, $size(n_{r,rotated}) = size(s) + size(r') + size(r_{left}|r_{right}) + 1$. The original size of n before rotation is equivalent to the above equation. This is represented in the pseudo code on line 19. The code sets $size(n_{r,rotated})$ equal to the prior size of n, which is represented in the code as $s.size$ because the prior $s.size$ assignment has yet to be altered. We also store the size of n_r before rotation in `temp` (represented as $r.size$ because assignment has yet to be altered) so that we can access this size later on when calculating the new size of n.

$$size(n_{r,rotated}) = size(n_{original}) \tag{1}$$

- To calculate the size of n after the rotation ($size(n_{rotated})$), we see that one of its subtrees is r'. Therefore, the $size(n_{rotated})$ is the sum of its r' and s. This is represented in the pseudo code on line 20. In the code, $size(s) = s.size - temp$ which equates to $size(n_{rotated}) = size(n_{original}) - size(r)$ because the original size of n, represented in the code as s has yet to be updated by the reassignments (setting certain subtrees to None). This equation is the same as below:

$$size(n_{rotated}) = size(r') + size(s) \tag{2}$$

- The other subtree of r ($r_{left,rotated}|r_{right,rotated}$) remains the same after rotation because `rotate` moves r in one direction so the subtree in the opposite direction of the rotation remains the same.

$$size(r_{left,rotated}|r_{right,rotated}) = size(r_{left}|r_{right}) \tag{3}$$

- We prove that the new rotation operation preserves the invariant of correct size-augmentation by assessing whether the subtrees of n_r (after rotation) correctly sum up to the size of n_r (after rotation) according to the equation:

$$size(n_r) = size(n_r.left) + size(n_r.right) + 1 \tag{4}$$

We do this through substituting (2) for one of the r_r's subtrees and (3) for the other subtree. We therefore get $size(n_r) = size(r') + size(s) + size(r_{left}|r_{right}) + 1$, which is equivalent to (1) as established above. Therefore, we have proven that the new rotation operation preserves the invariant of correct size-augmentation.

(c) Implement `rotate` in size-augmented BSTs in Python in the stub we have given you.

Food for thought (do read - it's an important take-away from this problem): This problem concerns size-augmented binary search trees. In lecture, we discussed AVL trees, which are balanced binary search trees where every vertex contains an additional *height* attribute containing the length of the longest path from the vertex to a leaf (height-augmented). Additionally, every pair of siblings in the tree have heights differing by at most 1, so the tree is height-balanced. Note that if we augment a binary search tree both by size (as in the above problem) and by height (and use it to maintain the AVL property), then we create a dynamic data structure able to perform `search`, `insert`, and `select` all in time $O(\log n)$.