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:
Collaborators:
No. of late days used on previous psets:
No. of late days used after including this pset:

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).

Proof. We can solve the problem with the following pseudocode:

```
AreaOfConvexPolygon $\{A\}$
Input   : A set of coordinates $A = \{(x_0, y_0), \dots, (x_{n-1}, y_{n-1})\}$ in any order
Output  : The area of the convex polygon formed by $A$
foreach $i = 0, \dots, n-1$ do
    $A'_i$ = PolarCoordinates($x_i, y_i$)
$A'$ = Sort($A'$);
foreach $i = 0, \dots, n-2$ do
    $c$ = LawOfCosines($A'_i, A'_{i+1}$)
    TotalArea += HeronsFormula($A'_i, A'_{i+1}, c$)
TotalArea += HeronsFormula($A'_0, A'_{n-1}$, LawOfCosines($A'_i, A'_{i+1}$))
return *TotalArea*
```

Here, we sort based on the θ values of each polar coordinate, so that the closest angle to 0 is first and the closest angle to 2π is last; this ensures that when we compute areas, we never overlap. Once we do this, we can find the three side lengths of every triangle formed with two vertices being adjacent in our sort and one value being the origin. The side lengths end up being r_i, r_{i+1} from the list of coordinates, and a third side which can be calculated by the Law of Cosines; then, we can use Heron's formula to find the area of each triangle. (To be honest, this isn't even necessary - we could just use the area formula $\frac{1}{2}ab\sin C$, where C is the positive difference between the angle measures θ. But I wanted to use the framework provided here.) When we add up all these values, we get the total area.

This algorithm ends up being correct because a convex polygon is fully divided by applying lines radiating out from an interior point to every vertex, and we account for every triangle formed here (because lines don't intersect, we clearly form n triangles, and note there are n leftmost vertices that we measure). Now note that every step besides the sort happens in constant time - the conversion to polar coordinates is guaranteed to be $O(1)$, computation of the Law of Cosines and then Heron's formula all come from basic arithmetic operations, and then we just add all our results. So across n sets of coordinates, that all turns out to be $O(n)$, which means that the sort is what dominates our overall runtime. □

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

Proof. As discussed above, our algorithm can be reduced to a sorting algorithm where that algorithm is bounded by time $O(n)$ for other arithmetic operations. We know that `MergeSort` runs in time $O(n \log n)$, so that works; the algorithm can be solved in time $O(n \log n)$. □

(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 (d) below.

Proof. There is not much to show here, to be honest. We accept that Π can be solved by an algorithm with runtime $g(n)$ which makes $k(n)$ oracle calls, each of which has runtime $T(f(n))$ for an input of size $f(n)$. Then the overall runtime is just the greater of $O(g(n))$ and $O(k(n) \cdot T(f(n)))$. This is easily expressed as just $O(g(n)) + k(n) \cdot T(f(n)))$. The crucial idea here is that if $k(n)$ is a constant or small factor, we can call the oracle several times and not experience much loss in our runtime compared with other algorithms. □

(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).)

Proof. The idea here is straightforward - use a series of if statements to sort our points (x_i, y_i) by quadrant; this is easily doable in $O(1)$ time by requiring x_i, y_i to have some greater or lesser relationship to 0. In particular, we can require points in Quadrant I to have $x_i \geq 0, y_i > 0$, Quadrant II to have $x_i < 0, y_i \geq 0$, Quadrant III with $x_i \leq 0, y_i < 0$ and Quadrant IV with $x_i > 0, y_i \leq 0$. Then - taking care to consider the $y = 0$ case - we compute the slope of the line from each point through the origin, and sort by axis on this parameter. This again is an $O(1)$ algorithm done n times. At this point, we have every single point sorted within their quadrants by what would be θ if we were in polar coordinates. Combining the four quadrants back into one list - all of this is still on $O(n)$ time - we are ready to apply the area formulas from above. We can use the Euclidean distance formula to find each c_i and compute areas with Heron's formula, which completes the idea. Therefore this is an $O(n)$ reduction which makes $O(1)$ (in particular, 4) calls to the Sorting oracle on arrays of length less than n. □

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.

   Proof. The function incorrectly implemented was `select`. Whenever we travel right on a BST, we must keep in mind that any entry on the right of a node is always larger than every element on the left of the node. Therefore, every time the `select` algorithm travels right on the tree, it should reduce the index it's looking for by the size of the left subtree plus one for the size of the parent node; this way we are searching for the appropriately large node in our new subtree. Originally, the recursive call did not decrease the index asked for on the right side of any tree - by decreasing the index the prescribed amount, all test cases came out correctly.

   The function implemented inefficiently was `insert`. Originally, the algorithm recalculated the sizes of every single node; however, this is unnecessary as we only need to compute the new sizes for parents of the inserted node. Luckily, there is only one of these per row in the tree, which means that we can do this in $O(h)$ time rather than $O(n)$ if we are smart. In particular, the way to do this is to add one to the size measurement of every node that we encounter before we have to return the node for the recursive algorithm to work; since we only encounter $O(h)$ nodes in our algorithm, this is a faster way to solve the problem. $\square$

   (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

[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).

size attribute had the correct subtree size before the operation, then the same is true after the operation.)

```
rotate $\{tree, direction, child\_side\}$
Input    : A BST, the way that we want to rotate,
           and the child of the BST we rotate on
Output   : The rotated BST
$antidirection = \text{Reverse}(direction)$
$pivot\_node = tree.child\_side$
$top\_node = pivot\_node.antidirection$
$pivot\_node.antidirection = top\_node.direction$
$top\_node.direction = pivot\_node$
$top\_node.size = pivot\_node.size$
$pivot\_node.size -= (top\_node.antidirection.size + 1)$
$tree.child\_side = top\_node$
return $tree$
```

The pseudocode above works in $O(1)$. In fact, we can be sure about this because it entirely consists of BST operations - naming a child or a size - and assignment. The BST operations are all just moving up or down a list of pointers, so we can be sure that this is $O(1)$, and we covered that assignment is a basic operation in class. This happens a set number of times - in 8 lines, here - so we are in $O(1)$ time.

The size-augmentations are all correct here as well. Note that every single node in a rotated BST keeps its same children except the ones denoted *pivot_node* and *top_node* (as no other node has its children replaced in the pseudocode above). So we need only examine these two to be sure that all nodes have the correct sizes. *top_node* goes where *pivot_node* was, and nothing is deleted from the tree; this means that *top_node* should have the same size as *pivot_node* originally did. Then *pivot_node* loses the child that wasn't rotated to *top_node*; also, *top_node* is no longer a child of *pivot_node* (though it loses no children besides those two classes). Therefore we must remove the size measurement of *pivot_node*'s unrotated child and subtract one for *top_node* (and of course, we note the case where *pivot_node* had no unrotated child and we just subtract 1). Therefore our size-augmentation is correct as long as the original measurement was correct.

(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 $O(\log n)$.