CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Problem Set 2

Harvard SEAS - Fall 2024 *Due: Wed 2024-09-25 (11:59pm)*

Your name: Ethan Veghte
Collaborators:
No. of late days used on previous psets: 0
No. of late days used after including this pset: 2

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

1. (designing reductions) The purpose of this exercise is to give you practice designing 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 (and you may use them without proof):

- 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).

Step 1: Convert points to Polar coordinates $\Rightarrow O(1)$
Step 2: Use a sort function to put the indices in a sorted order according to their θ *values.* $\Rightarrow O(Sorting)$
Step 3: Rely on Heron's Formula to calculate the areas of the triangles from the origins to the indices. By doing the sequential pairs in the ordered list. That is if Heron's Formula is algorithm A(x, y) we do $A([0],[1]) + A([1],[2]) + \ldots A([n-2],[n-1])$. *This means that we perform Heron's Formula n times then n additions making it* $O(2n)$. $\Rightarrow O(n)$

In essence the sorting algorithm is a ***reduction*** *from AreaOfConvexPolygon. This is because there exists an algorithm that solves* Π *(AreaOfConvexPolygon), using a subroutine oracle (fast sorting methods) that solves* Γ *(sorting of indices based on* θ*). Since*

this demonstrates Sorting as a reduction of AreaOfConvexPolygon, then we know notion of reductions then we know AreaOfConvexPolygon$\leq_{O(n),n}$ Sorting.

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

The deduction is then easy as since AreaOfConvexPolygon$\leq_{O(n),n}$ Sorting, and sorting can be done in $O(n \log n)$, and AreaOfConvexPolygon uses sorting as a reduction then AreaOfConvexPolygon can be solved in time $O(n \log n)$.

(c) (*challenge; extra credit; optional[1]) 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 the slope of the line from a vertex (x,y) to the origin.)

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. (composition of reductions) The purpose of this exercise is to give you practice in working with the abstract definition of reductions.

Let Π, Γ, Λ be computational problems, and suppose that $\Pi \leq_{T_1(n),q_1(n)\times h_1(n)} \Gamma$ and $\Gamma \leq_{T_2(n),q_2(n)\times h_2(n)} \Lambda$, for non-decreasing functions $T_1, T_2, q_1, q_2, h_1, h_2$. Determine functions T_3, q_3, h_3 in terms of $T_1, T_2, q_1, q_2, h_1, h_2$ such that

$$\Pi \leq_{O(T_3(n)),q_3(n)\times h_3(n)} \Lambda,$$

and justify your answers. (Hint: you can take $h_3(n) = h_2(h_1(n))$.)

$h_3(n) = h_2(h_1(n))$: First off this is the size of the oracle calls. So with Γ is a reduction of Π as shown by $\Pi \leq_{T_1(n),q_1(n)\times h_1(n)} \Gamma$ the oracle used is of size at most $h_1(n)$ with input size n. The next step is that Λ is a reduction of Γ and the size of its oracle calls are $h_2(n)$, for input of size n, which means that in terms of Π and Λ the oracle calls are first of size $h_1(n)$ with respect to reduction Γ, but those calls of size $h_1(n)$ are then called on the further reduction Λ whose oracle calls are of size $h_2(\{input\})$ which means the input to Λ-oracle are of size $h_1(n)$ resulting in $h_3(n) = h_2(h_1(n))$.

$q_3(n) = q_1(n) \cdot q_2(h_1(n))$: This is a similar notion as above except that instead of the size of oracle calls being the runtime variable in question it is the number of oracle calls. First, Π has $q_1(n)$ oracle calls for reduction Γ with input size n, and Γ has $q_2(n)$ oracle calls for reduction Λ with input size n. This means that for Λ as a reduction of Π the input size of the number of oracle calls ($q_2\{input\}$) for Λ-oracle is $h_1(n)$ (i.e. $q_2(h_1(n))$), but for every Λ-oracle call there is a Γ-oracle call so the $q_2(h_1(n))$ oracle calls happen $q_1(n)$ times. This

[1]This problem is meant to be done based on your enjoyment/interest and only if you have time. It won't make a difference between N, L, R-, and R grades (meaning it will only impact whether an R gets increased to an R+), and course staff will deprioritize questions about this problem at office hours and on Ed.

means that Λ *as a reduction of* Π *has has* $q_3(n) = q_1(n) \cdot q_2(h_1(n))$ *oracle calls.*

$T_3(n) = T_1(n) + [q_1 \cdot T_2(h_1(n))]$*: Based on* $\Pi \leq_{T_1(n),q_1(n)\times h_1(n)} \Gamma$ *we know there are* q_1 *calls to reduction* Γ*, and those reduction calls take* $T_2(n)$ *time for input size* n*. However, the inputs to oracle calls are of size* $h_1(n)$*, thus there are* $q_1(n)$ *calls that take* $T_2(h_1(n))$ *time (i.e.* $q_1 \cdot T_2(h_1(n))$ *). However, the first reduction to* Γ *take time* $T_1(n)$ *in it of itself, because it is the total time required to complete this entire process of transforming the instance of* Π *into instances of* Γ *and making queries to the* Γ*-oracle. Thus combined* $T_3(n) = T_1(n) + [q_1 \cdot T_2(h_1(n))]$

3. (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 $O(h)$.[2]

   (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, another one is too slow (running in time that's (at least) linear in the number of nodes of the tree rather than linear in the height of tree), and the third is correct.

   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.

   `search`: Correct implementation

   `insert`: Inefficient implementation
   This implementation is inefficient because of the `self.calculate.size()` *call within the function. This is a* $O(n)$ *operation making it inherently inefficient compared to the rest of the algorithm. Instead if the size was simply adjusted incrementally down the branch that the insertion occurred this becomes an* $O(logn)$ *operation. To fix this I simply added an incremental update at the end of the function. That way as it works it way back up the tree (*$O(logn)$ *length) the sizes are updated. The correct implementation is:*

```
def insert(self, key):
    if self.key is None:
```

[2]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).

```
        self.key = key
    elif self.key > key:
        if self.left is None:
            self.left = BinarySearchTree(self.debugger)
        self.left.insert(key)
    elif self.key < key:
        if self.right is None:
            self.right = BinarySearchTree(self.debugger)
        self.right.insert(key)
    self._size =
        1 + (self.left.size if self.left else 0)
        + (self.right.size if self.right else 0)
    return self
```

`select`: Incorrect implementation
There are two wrong parts. The first wrong part of the implementation was dealing with if the ind variable was larger than the size of the left child (i.e. the ind-th smallest key is not in the left child). Previously it was

```
if left_size < ind and self.right is not None:
return self.right.select(ind)
```

This meant that when going into the right child it was now only looking for the ind-th key in the right subtree. However, we don't want the ind-th key in the right subtree we want the ind-th key in the entire tree. We have already ruled out some amount of the tree because we know it is not in the left child at this point so the new ind-th value we want to look for is $ind - left_size - 1$*. However, we also must make sure that the ind-th element of the tree being entered exists at all. Since we know that the initial* $ind < self.size - 1$ *we cannot guarantee that when we recur into the right child that will be maintained. I added this if condition as*

```
if ind >= self.size:
```

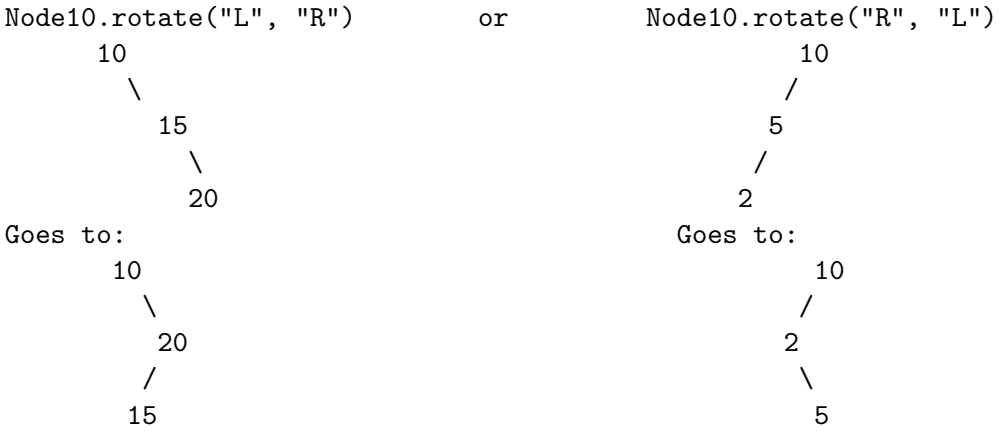
which is redundant for the first case but necessary in later recurrences. The final implementation:

```
    def select(self, ind):
left_size = 0
if self.left is not None:
    left_size = self.left.size
if ind == left_size:
    return self
if ind >= self.size:
    return None
if left_size > ind:
    return self.left.select(ind)
```

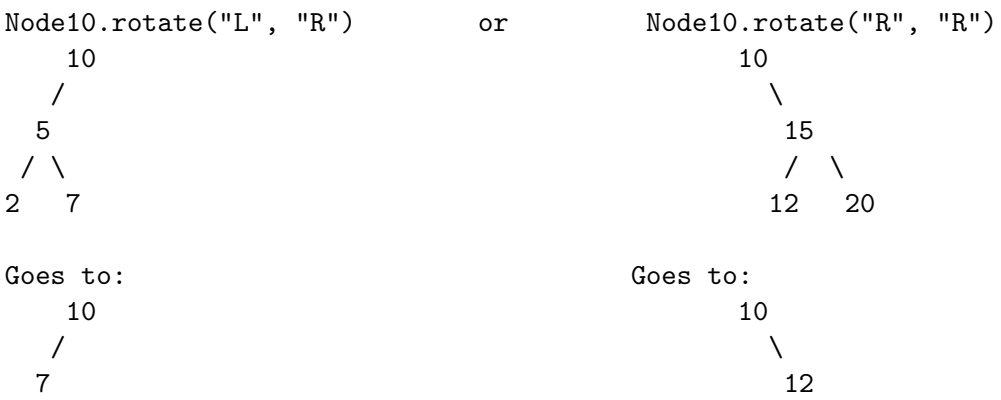
```
        if left_size < ind and self.right is not None:
            return self.right.select(ind - left_size - 1)
        return None
```

(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.)

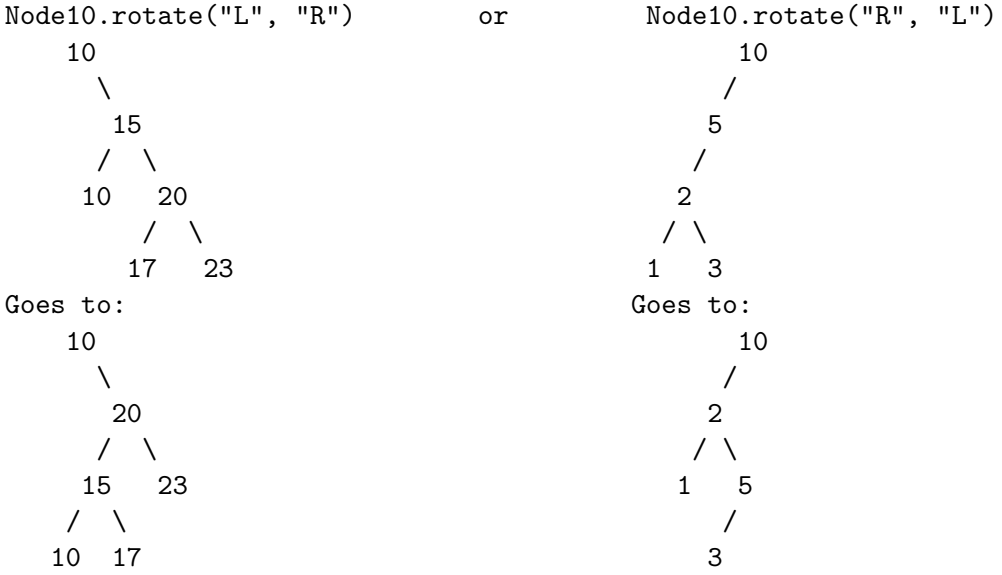
With size-augmented BSTs (each node has a size attribute), the `rotate` function operates by taking 3 arguments. 1: the tree (`self`). 2: the direction. 3: the child side. The implementation must recognize 3 cases. The first is if the child of the tree origin node has one child itself. In this case the rotation direction must be the opposite to the side the child is on for it to be possible (see below). That is because you cannot rotate further left if it only has a child on the left side, and vice versa.



The second case is if the specified child of the tree origin node side has 2 children of its own. It is still simple if the rotation and the specified child of the origin node are the same side (i.e. "R" and "R" and vice versa). This just means that everything is just easily rotated over. A grandchild becomes the new child and the other becomes a great-grandchild (see below).



The other circumstance is if the child side and the rotation side differ. This gets more complicated if the grandchild has 2 children as well because the output is no longer just a single direction branch. The branch sides will need to be switched in order to maintain the integrity of the BST. As a result, when this case is updated the grandchild's child of `self` *that is to the same side of the child as the rotation is attached to the opposite side of its grandparent node as it was to its parent (i.e. "R" rotation means the R child of the child node is attached as the left child to the grandparent node and vice versa). See below for a diagram.*



The pseudo code looks like: *This maintains $O(1)$ because the number of operations is constant. The algorithm is made up of the act of rotation as well as updating the sizes. The rotation step is a constant as it a constant set of commands on only specific nodes. The updating of sizes is constant because all of the subtrees of the child (rotated away/down) and grandchild (rotated up) did not have their sizes changed in any way. This means the updating of sizes can be done by relying on calls to those sizes in their subtrees.* $size = 1 + size_{left} + size_{right}$.
The pseudo code looks like:

```
3 arguments (tree origin, direction of rotation, child)
    if direction is L
        check for child is not None and that the child.right is not None
        set tempt root var to be child.right
        shift parent down to left and make its parent the temp root var
            (also update new root's pointer to the parent)
```

```
            update child size and new root size using established
            right and left sizes

            update pointers from origin node to update nodes

        if direction is R
            check for child is not None and that the child.right is not None
            set tempt root var to be child.right
            shift parent down to left and make its parent the temp root var
                (also update new root's pointer to the parent)

            update child size and new root size using established
            right and left sizes

            update pointers from origin node to update nodes
```

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

```
def rotate(self, direction, child_side):
# set a child variable
if child_side == "L":
    child = self.left
elif child_side == "R":
    child = self.right
else:
    return self  # Invalid child side, return the current node

# Left rotation
if direction == "L":
    if child is not None and child.right is not None:
        new_root = child.right
        child.right = new_root.left
        if new_root.left:
            new_root.left.parent = child
        new_root.left = child

        # Update sizes
        child.size = 1 +
            (child.left.size if child.left else 0) +
            (child.right.size if child.right else 0)
        new_root.size = 1 +
            (new_root.left.size if new_root.left else 0) + (new_root.right.size i

        # Update root pointers
        if child_side == "L":
            self.left = new_root
```

```
            else:
                self.right = new_root

    # Right rotation
    elif direction == "R":
        # Perform right rotation on the identified child
        if child is not None and child.left is not None:
            new_root = child.left
            child.left = new_root.right
            if new_root.right:
                new_root.right.parent = child
            new_root.right = child

            # Update sizes
            child.size = 1 +
                (child.left.size if child.left else 0) +
                (child.right.size if child.right else 0)
            new_root.size = 1 +
                (new_root.left.size if new_root.left else 0) +
                (new_root.right.size if new_root.right else 0)

            # Update root pointers
            if child_side == "L":
                self.left = new_root
            else:
                self.right = new_root

    # Return updated tree/subtree
    return self
```

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 $O(\log n)$.

4. (reflection) We aim for cs1200 to be a collaborative learning community. Describe two concrete ways in which you have supported, or will try to support, your classmates' learning in the course. Be specific, connecting your answer to the structure of cs1200.

   Note: As with the previous psets, you may include your answer in your PDF submission, but the answer should ultimately go into a separate Gradescope submission form.

   I have found a solid PSET working group. This week unfortunately I was not able to find time to work with them, but I find this material extremely valuable to talk out. The problems are

not that complex or confusing in the end, it is just the initial steps to wrap your head around a problem and talking with others is helpful (e.g. talking through the jargon-y questions). I hope to be able to start next week's pset early and so that when I find time to work with the pset group i have found I can be helpful to others as well as myself by talking them out. Beyond the PSET working group I also went to Anurag's office hours, and felt he did a very good job of encouraging us to to try to teach if we had an answer to someone else's question. Yes, office hours was helpful for my own questions towards Anurag or the TFs but it also was very helpful from a collaborative sense. I will continue utilizing that in the future.

5. Once you're done with this problem set, please fill out this survey so that we can gather students' thoughts on the problem set, and the class in general. It's not required, but we really appreciate all responses!