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

Problem Set 2

Harvard SEAS - Fall 2023 *Due: Tue Sep. 27, 2023 (11:59pm)*

Your name: Jean Yves Gatwaza
Collaborators: None
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).

(b) Deduce that AreaOfConvexPolygon 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 $h(n)$. Show that if Γ can be solved in time at most $T(n)$, then Π can be solved in time at most $O(g(n) + k(n) \cdot T(h(n)))$. Note that the case $k(n) = 1$ was stated in class; the case $k(n) > 1$ is useful as well, such as in Part 1d below.

(d) (*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

[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.

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.

Solution

(a) To show that AreaOfConvexPolygon reduces to sorting in $O(n)$ time, let's first define a reduction process that solves the problem using sorting. The reduction process I came up with is as follows:

i. **Convert the points to polar coordinates:** We will convert the points to polar coordinates using the formula $r = \sqrt{x^2 + y^2}$ and $\theta = \arctan(\frac{y}{x})$. This conversion can be done in $O(1)$ time for each point as given in the problem.

ii. **Sort the points by their θ:** We will sort the points by their angle θ using a comparison based sorting algorithm. This will arrange the points in a counter-clockwise order around the origin, making it easy to calculate the area of the polygon.

iii. **Calculate the area of the polygon:** We will calculate the area of the polygon by calculating the area of the triangles formed by the origin and two consecutive points in the sorted list. We can use the formula given above: $A = \sqrt{s(s-a)(s-b)(s-c)}$ where a, b, c are the side lengths of the triangle and $s = \frac{a+b+c}{2}$. This can be done in $O(n)$ time since we have the points sorted in a counter-clockwise order.
The area of the ploygon will be the sum of the the areas A_i of the triangles formed by the origin $O(0,0)$ and two consecutive points $P_i(x_i, y_i)$ and $P_{i+1}(x_{i+1}, y_{i+1})$ in the sorted list.

iv. **Return the area of the polygon**
Proof of correctness:

We are going to prove how the reduction process is correct by proving that it returns the correct area of the polygon.

A. **Converting the points to polar coordinates:** This conversion is correct because the formula $r = \sqrt{x^2 + y^2}$ and $\theta = \arctan(\frac{y}{x})$(mathematically) correctly converts the points to polar coordinates.

B. **Sorting based on Polar angles:** We will assume our sorting algorithm is correct, as we are using it as an oracle, and correctly sorts the points based on their polar angles (We can assume we use the angles as keys and sort the points based on the keys).

C. **Calculating the area of the polygon:** We will prove by contradiction that the area of the polygon is the sum of the areas of the triangles formed by the origin and two consecutive points in the sorted list:
Let's assume that the area of the polygon is not the sum of the areas of the

triangles formed by the origin and two consecutive points in the sorted list. This means that there is a triangle T formed by the origin and two consecutive points in the sorted list that is not part of the polygon. This is a contradiction because the polygon is convex, meaning that all the points in the polygon are connected by straight lines. This means that the area of the polygon is the sum of the areas of the triangles formed by the origin and two consecutive points in the sorted list.

D. **Returning the area of the polygon:** This is correct because we have calculated the area of the polygon correctly.

Runtime analysis:

A. **Converting the points to polar coordinates:** This conversion can be done in $O(1)$ time for each point as given in the problem. Since we have n points, the total runtime for this step is $O(n)$.

B. **Sorting based on Polar angles:** We will assume our sorting algorithm is correct, as we are using it as an oracle, and correctly sorts the points based on their polar angles (We can assume we use the angles as keys and sort the points based on the keys). Since we have n points, the total runtime for this step is $O(n \log n)$.

C. **Calculating the area of the polygon:** We can calculate the area of each triangle in $O(1)$ time. Since we have n points, the total runtime for this step is $O(n)$.

D. **Returning the area of the polygon:** This is correct because we have calculated the area of the polygon correctly. Since we have n points, the total runtime for this step is $O(1)$.

Therefore, we conclude that AreaOfConvexPolygon is reductible to $O(n)$; n Sorting because we have shown that it can be solved using $O(n)$ calls to a Sorting oracle on arrays of length at most n.

(b) To show that AreaOfConvexPolygon can be solved in $O(n \log n)$ time, let's analyze the steps of our solution.

i. **Converting the points to polar coordinates:** This conversion is $O(n)$ as each conversion is $O(1)$ and we have n points.

ii. **Sorting based on Polar angles:** This step is $O(n \log n)$ as we are sorting n points.

iii. **Calculating the area of the polygon:** This step is $O(n)$ as we are calculating the area of n triangles and each triangle's area can be calculated in $O(1)$ time.

From this we can see that the core part of the reductrion is the sorting, which takes $O(n \log n)$ time. The conversion to polar cordinates and the calculation of the area of the polygon take $O(n)$ time. Therefore, we conclude that AreaOfConvexPolygon can be solved in time $O(n \log n)$. because $O(n \log n)$ is the dominant term in the runtime of the reduction process.

(c) To show that if Γ can be solved in time at most $T(n)$, then Π can be solved in time at most $O(g(n) + k(n) \cdot T(h(n)))$, we will use the reduction process we defined in part A.

The reduction itself runs in time $O(n)$

- For each of the $k(n)$ oracle calls during the reduction, problem Γ needs to be solved on an instance of size at most $h(n)$. And because problem γ can be solved in times at most $T(n)$, each oracle call thus takes $O(T(h(n)))$ time.
  We have $k(n)$ oracle calls, so the total time for the oracle calls is $O(k(n) \cdot T(h(n)))$.
- The overall time complexity to solve problem Π will be the sum of the time complexity of the reduction and the time complexity of the oracle calls. This will give us
$$O(g(n) + k(n) \cdot T(h(n)))$$

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 $O(h)$.[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.

   (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.)

   (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

[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).

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 $O(\log n)$.

Solution

(a) The function *select* is incorrect because it is supposed to return the ind*th* key in the tree considering an inorder traversal of the tree–visiting nodes in ascending order of their keys. This means that going to the right subtree, it should modify the index to be $ind - left_size - 1$ as nodes including the current node and $left_size$ nodes in the left subtree should not be skipped. This is the correction I made to the function.:

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

The *insert* function on the other hand, is too slow because after inserting a new key, it calls the *calculatesizes* method that updates the size attribute for every node in the tree, which recalculates the sizes for all nodes and results in a runtime of $O(n)$ instead of $O(h)$. To optimize the function, I chose to update only the size attribute of the particular node we are visiting based on its left and right children's sizes. This is the correction I made to the function:

```
def insert(self, key):
if self.key is None:
    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)
# Here, we shall only update the size of the current node
#based on its children. Count the current node and then
#add the size of its left and right subtrees
self.size = 1 + (self.left.size if self.left else 0)
```

```
        + (self.right.size if self.right else 0)
        return self
```

As a result, in the worst case, the insert method will take $O(h)$ time, where h is the height of the tree. This is because the insert method will have to traverse the tree from the root to the inserted node, updating the size attribute of each node on the path.

(b) **Concept:** Left Rotation is performed at a node x, where the right child of x, i.e., y, becomes the new parent, and the left child of y becomes the right child of x.

Extension to Size-Augmented BSTs: In size-augmented BSTs, each node has an additional attribute, `size`, representing the number of nodes in its subtree, including itself. When performing rotations, we need to ensure that the `size` attribute of the affected nodes is correctly updated to maintain the size-augmented property.

Pseudocode:

```
def left_rotate(x):
    y = x.right  # y is the right child of x
    if y is None:
        return x  # Left rotation is not possible if the right child is Non

    x.right = y.left  # Set the left child of y as the right child of x
    if y.left is not None:
        y.left.parent = x  # Update the parent pointer of y's left child

    y.parent = x.parent  # Update y's parent pointer
    if x.parent is None:
        root = y  # y becomes the new root if x was the root
    elif x == x.parent.left:
        x.parent.left = y  # y replaces x as the left child of x's parent
    else:
        x.parent.right = y  # y replaces x as the right child of x's parent

    y.left = x  # Set x as the left child of y
    x.parent = y  # Update x's parent pointer

    # Update the size attribute of x and y
    x.size = 1 + size(x.left) + size(x.right)
    y.size = 1 + size(y.left) + size(y.right)

    return root  # Return the possibly updated root of the tree
```

Correctness and Runtime Analysis:

- **Correctness:** The rotation preserves the Binary Search Tree property, and the `size` attributes of the nodes x and y are correctly updated based on their new children after the rotation, preserving the size-augmented property.
- **Runtime:** The rotation involves a constant number of pointer updates and arithmetic operations to update the `size` attributes, implying a runtime of $O(1)$.

Proof of Preserving Size-Augmented Property:

- Before the rotation, the `size` attribute of each node correctly represents the number of nodes in its subtree.
- After the rotation, the `size` attributes of only the nodes involved in the rotation (x and y) need to be updated, ensuring that every node's `size` attribute has the correct subtree size after the operation.

This approach guarantees that the `size` augmentation invariant is preserved during the rotation, ensuring the correctness of the size-augmented BSTs after the rotation operation.

(c) ********** Implemented in ps2.py **********