CS 266 Homework 4

Zachary DeStefano, PhD Student, 15247592

Due Date: May 8

Problem 4.11

Give an example of a 2-dimensional linear program that is bounded, but where there is no lexicographically smallest solution.

This happens in the case where an entire edge of the bounded region is the smallest solution.
Here is a simple example of that. These are the constraints:
$x \leq 4$ and $x \geq 0$
$y \leq 4$ and $y \geq 0$
Our objective function to minimize is then y.
The whole edge where $y = 0$ is the best solution.

Problem 4.14

4.14 Here is a paranoid algorithm to compute the maximum of a set A of n real numbers:
What is the worst-case running time of this algorithm?
What is the expected running time (with respect to the random choice in line 3)?

Worst-case running time

In the worst-case, it recurses every time and checks $n - 1$ elements as well as checks all $n - 1$ elements in the paranoid step. Thus, if you let c and k be constants, then we have

$$T(n) = c + T(n-1) + k * (n-1)$$

We can simplify this to

$$T(n) = c * n + k + T(n-1)$$

After doing out the recursion, we end up with

$$T(n) = c * (n + (n-1) + (n-2) + ... + 1) + k * n$$

We have an arithmetic series so we can simplify this to the following

$$T(n) = c\frac{n(n+1)}{2} + k * n$$

Simplifying this into big-Oh notation, we have

$$T(n) = O(n^2)$$

Average running time

In the average case, the check of the all the elements again only occurs if we picked the maximum element of the current set. If the current set has n elements then this probability is $\frac{1}{n}$. The recursion will happen regardless of which element we picked.

This makes our equation the following:

$$T(n) = k + c\frac{1}{n}(n-1) + T(n-1)$$

We can replace the $(n-1)$ by n for simplifying and we will then have

$$T(n) \leq k + T(n-1)$$

Expanding this out we have

$$T(n) \leq k * n$$

Thus we now have

$$T(n) = O(n)$$

This gives us the expected running time.

Problem 5.1

We can use the Divide and Conquer Theorem for this problem, which states the following:
Let $a \geq 1, b > 1$ and $c, k \geq 0$ be constants.
If $T(n)$ defined on nonnegative integers by $T(n) = aT(\frac{n}{b}) + c * n^k$
T(n) can be bounded asymptotically as follows:
if $a > b^k$ then $T(n) = \Theta(n^{log_b(a)})$
if $a = b^k$ then $T(n) = \Theta(n^k log(n))$
if $a < b^k$ then $T(n) = \Theta(n^k)$

For this problem, it holds that $a = 2$, $b = 4$, $c = 2$, and $k = 0$.
It then happens that $a > b^k$, thus our solution is $Q(n) = \Theta(n^{log_b(a)})$.
As it happens $log_4(2) = \frac{1}{2}$, thus we have $Q(n) = \Theta(n^{\frac{1}{2}})$
This proves that $Q(n) = O(\sqrt{n})$ and $Q(n) = \Omega(\sqrt{n})$.

TODO: Proof with points and query rectangle

Problem 5.10

In some applications one is interested only in the number of points that lie in a range rather than in reporting all of them. Such queries are often referred to as range counting queries. In this case one would like to avoid having an additive term of O(k) in the query time.
a. Describe how a 1-dimensional range tree can be adapted such that a range counting query can be performed in O(logn) time. Prove the query time bound.
b. Using the solution to the 1-dimensional problem, describe how ddimensional range counting queries can be answered in O(logd n) time. Prove the query time.
c.* Describe how fractional cascading can be used to improve the running time with a factor of O(logn) for 2- and higher-dimensional range counting queries.

Part A

Assuming that in our range tree, each node says the total number of subnodes, it is easy to have an O(log n) algorithm for range counting.

In the 1-d case, we just have a binary search tree, so we just have to do the following:
1. Transverse the Binary Search Tree
2. After you get to a split vertex:
- Traverse the left side and every time you have to make a left turn, add the right node's subnode count to the range count.

- Traverse the right side and every time you have to make a right turn, add the left node's subnode count to the range count.

Running time:
There is a constant number of operations per level and there are $log_2(n)$ levels, thus the total running time is $O(logn)$.

Part B

You make sure to have binary search trees that report how many nodes are in subtrees.

For the d-dimensional case, once you get to a split vertex as well as other vertices, you report the number of matching y-coordinates by going through the linked binary search trees. For the next dimension, you do the same thing and so on.

Running time:
In the first dimension, there are $log_2(n)$ levels. For each level, we will do at most $log_2(n)$ operations to get the number in the next level. This means that for d dimensions, we will end up with $O(log^d(n))$ operations.

Part C