CS 266 Homework 4

Zachary DeStefano, PhD Student, 15247592

Due Date: May 8

Problem 4.11

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 $y \geq 0$
Objective Function: y
The whole edge where $y = 0$ is the best solution.

Problem 4.14

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 $Q(n) = \Omega(\sqrt{n})$.

Here is the proof of lower bound:
Take 7 points lined up in a row.
The first split will get a node followed by 3 points in each subtree.
The next split will get a node followed by two leaves.
There are 2 splits and a constant number of operations per split,
so it is lower bounded by 2, and $floor(\sqrt{7}) = 2$

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

First, construct a range tree where each node says the total number of subnodes.
In the 1-d case, we just have a binary search tree where each node contains the number of subnodes in its data. Here is the algorithm:

GetRangeCount(Binary Search Tree B, x_{min}, x_{max}):
1. Transverse the Binary Search Tree
2. Intialize rangeCount to zero.
3. Considering x_{min} and x_{max} when 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 rangeCount
- Traverse the right side and every time you have to make a right turn,
add the left node's subnode count to rangeCount
4. return rangeCount

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

Part B

Do the same initial construction as Part A, where you make a range tree where each node contains the number of subnodes. You recurse though into the next dimension and only start counting though with the subtree in the last dimension. We will be referencing the algorithm GetRangeCount from part A. Here is the recursive algorithm:

```
Get_dDimensionalRangeCount(rangeTree T, range R, dimension d):
- if d==1
-     return GetRangeCount(T, R_min, R_max)
- else
-     rangeCount = 0
-     nextRange = R without first dimension
-     Traverse first dimension of T considering first dimension of R
-     When split vertex arrives:
-           Traverse left side:
-               Every time you make a left turn:
-                   Let nodeTree = the (d-1) dimensional subtree at the right node
-                    rangeCount = rangeCount + ...
-                      Get_dDimensionalRangeCount(nodeTree,nextRange,d-1)
-          Traverse right side:
-               Every time you make a right turn:
-                   Let nodeTree = the (d-1) dimensional subtree at the left node
-                   rangeCount = rangeCount + ...
-                    Get_dDimensionalRangeCount(nodeTree,nextRange,d-1)
-      return rangeCount
```

Running time:

In the first dimension, there are $log_2(n)$ levels. In the worst case at each level of the first dimension, we will recurse into the subtree at the next dimension.

Let $T(n, d)$ be the running time for n points with d dimensions.
We have just shown then that $T(n, d) \leq log_2(n) \cdot T(n, d-1)$
We know from part A that $T(n, 1) \leq log_2(n)$
Putting these together, it holds that $T(n, d) = (log_2(n))^d$, thus
our running time is $O(log^d(n))$

Part C

We will follow the construction in section 5.6 of the range tree with fractional cascading incorporated. We can traverse the tree in the exact same initial way Part B, but for the last two dimensions, we will incorporate fractional cascading to get the range counts to be added to the total range count.

At the point in the fractional cascading algorithm when we report the points in A(v) whose y-coordinate is in the range $[y : y']$, we will modify this to find the index at y and the index at y' to get the number of points in the range. Both of these can be done in constant time since we are keeping track of the range.

Since we are not reporting k points and doing a constant number of operations, the total query time for the last two dimensions is $O(log(n))$ instead of $O(log(n) + k)$

The previous dimensions will still have $log(n)$ query time, thus the total running time is $O(log^{d-1}(n))$. This improves the running time by a $log(n)$ factor as required.