Problem Set 3

Both theory and programming questions are due **Thursday, October 6** at **11:59PM**. Please download the .zip archive for this problem set, and refer to the `README.txt` file for instructions on preparing your solutions.

We will provide the solutions to the problem set 10 hours after the problem set is due, which you will use to find any errors in the proof that you submitted. You will need to submit a critique of your solutions by **Friday, October 7th, 11:59PM**. Your grade will be based on both your solutions and your critique of the solutions.

Problem 3-1. [45 points] **Range Queries**

Microware is preparing to launch a new database product, NoSQL Server, aimed at the real-time analytics market. Web application analytics record information (e.g., the times when users visit the site, or how much does it take the server to produce a HTTP response), and can display the information using a Pretty Graph™[1], so that CTOs can claim that they're using data to back their decisions.

NoSQL Server databases will support a special kind of index, called a *range index*, to speed up the operations needed to build a Pretty Graph™ out of data. Microware has interviewed you during the fall Career Fair, and immediately hired you as a consultant and asked you to help the NoSQL Server team design the range index.

The range index must support fast (sub-linear) insertions, to keep up with Web application traffic. The first step in the Pretty Graph™ algorithm is finding the minimum and maximum values to be plotted, to set up the graph's horizontal axis. So the range index must also be able to compute the minimum and maximum over all keys quickly (in sub-linear time).

(a) [1 point] Given the constraints above, what data structure covered in 6.006 lectures should be used for the range index? Microware engineers need to implement range indexes, so choose the simplest data struture that meets the requirements.

1. Min-Heap
2. Max-Heap
3. Binary Search Tree (BST)
4. AVL Trees
5. B-Trees

Solution: AVL trees are the only data structure that supports sub-linear insertions, as well as sub-linear minimum and maximum queries.

[1] U.S. patent pending, no. 9,999,999

(b) [1 point] How much time will it take to insert a key in the range index?

1. $O(1)$
2. $O(\log(\log N))$
3. $O(\log N)$
4. $O(\log^2 N)$
5. $O(\sqrt{N})$

Solution: Inserting a node in an AVL tree takes $O(\log N)$ time. See the lecture notes on AVL trees.

(c) [1 point] How much time will it take to find the minimum key in the range index?

1. $O(1)$
2. $O(\log(\log N))$
3. $O(\log N)$
4. $O(\log^2 N)$
5. $O(\sqrt{N})$

Solution: Finding the minimum key in an AVL tree takes $O(\log N)$ time. See the lecture notes on AVL trees.

(d) [1 point] How much time will it take to find the maximum key in the range index?

1. $O(1)$
2. $O(\log(\log N))$
3. $O(\log N)$
4. $O(\log^2 N)$
5. $O(\sqrt{N})$

Solution: Finding the maximum key in an AVL tree takes $O(\log N)$ time. See the lecture notes on AVL trees.

The main work of the Pretty Graph™ algorithm is drawing the bars in the graph. A bar shows how many data points there are between two values. For example, in order to produce the visitor graph that is the hallmark of Google Analytics, the range index would record each time that someone uses the site, and a bar would count the visiting times between the beginning and the ending of a day. Therefore, the range index needs to support a fast (sub-linear time) COUNT(l, h) query that returns the number of keys in the index that are between l and h (formally, keys k such that $l \leq k \leq h$).

Your instinct (or 6.006 TA) tells you that COUNT(l, h) can be easily implemented on top of a simpler query, RANK(x), which returns the number of keys in the index that are smaller or equal to x (informally, if the keys were listed in ascending order, x's rank would indicate its position in the sorted array).

(e) [1 point] Assuming $l < h$, and both l and h exist in the index, COUNT(l, h) is

1. $\textsc{Rank}(l) - \textsc{Rank}(h) - 1$
2. $\textsc{Rank}(l) - \textsc{Rank}(h)$
3. $\textsc{Rank}(l) - \textsc{Rank}(h) + 1$
4. $\textsc{Rank}(h) - \textsc{Rank}(l) - 1$
5. $\textsc{Rank}(h) - \textsc{Rank}(l)$
6. $\textsc{Rank}(h) - \textsc{Rank}(l) + 1$
7. $\textsc{Rank}(h) + \textsc{Rank}(l) - 1$
8. $\textsc{Rank}(h) + \textsc{Rank}(l)$
9. $\textsc{Rank}(h) + \textsc{Rank}(l) + 1$

Solution: $\textsc{Rank}(h) - \textsc{Rank}(l) + 1$. We want to count all the keys x such that $l \leq x \leq h$, which is equivalent to all the keys $x \leq h$ minus all the keys $x < l$. $\textsc{Rank}(h)$ counts all keys x s.t. $x \leq h$. $\textsc{Rank}(l)$ counts all keys s.t. $x \leq l$, and will include l. So we want $\textsc{Rank}(h) - (\textsc{Rank}(l) - 1) = \textsc{Rank}(h) - \textsc{Rank}(l) + 1$

(f) [1 point] Assuming $l < h$, and h exists in the index, but l does not exist in the index, $\textsc{Count}(l, h)$ is

1. $\textsc{Rank}(l) - \textsc{Rank}(h) - 1$
2. $\textsc{Rank}(l) - \textsc{Rank}(h)$
3. $\textsc{Rank}(l) - \textsc{Rank}(h) + 1$
4. $\textsc{Rank}(h) - \textsc{Rank}(l) - 1$
5. $\textsc{Rank}(h) - \textsc{Rank}(l)$
6. $\textsc{Rank}(h) - \textsc{Rank}(l) + 1$
7. $\textsc{Rank}(h) + \textsc{Rank}(l) - 1$
8. $\textsc{Rank}(h) + \textsc{Rank}(l)$
9. $\textsc{Rank}(h) + \textsc{Rank}(l) + 1$

Solution: $\textsc{Rank}(h) - \textsc{Rank}(l)$. We want to count all the keys x such that $l \leq x \leq h$, which is equivalent to all the keys $x \leq h$ minus all the keys $x < l$. $\textsc{Rank}(h)$ counts all keys x s.t. $x \leq h$. $\textsc{Rank}(l)$ counts all keys s.t. $x \leq l$, which is equivalent to all the keys $x < l$ since l is not in the index. So $\textsc{Rank}(h) - \textsc{Rank}(l)$ is the right answer.

(g) [1 point] Assuming $l < h$, and l exists in the index, but h does not exist in the index, $\textsc{Count}(l, h)$ is

1. $\textsc{Rank}(l) - \textsc{Rank}(h) - 1$
2. $\textsc{Rank}(l) - \textsc{Rank}(h)$
3. $\textsc{Rank}(l) - \textsc{Rank}(h) + 1$
4. $\textsc{Rank}(h) - \textsc{Rank}(l) - 1$
5. $\textsc{Rank}(h) - \textsc{Rank}(l)$
6. $\textsc{Rank}(h) - \textsc{Rank}(l) + 1$

7. $\text{RANK}(h) + \text{RANK}(l) - 1$
8. $\text{RANK}(h) + \text{RANK}(l)$
9. $\text{RANK}(h) + \text{RANK}(l) + 1$

Solution: $\text{RANK}(h) - \text{RANK}(l) + 1$. We want to count all the keys x such that $l \leq x \leq h$, which is equivalent to all the keys $x \leq h$ minus all the keys $x < l$. $\text{RANK}(h)$ counts all keys x s.t. $x \leq h$. $\text{RANK}(l)$ counts all keys s.t. $x \leq l$, and will include l. So we want $\text{RANK}(h) - (\text{RANK}(l) - 1) = \text{RANK}(h) - \text{RANK}(l) + 1$

(h) [1 point] Assuming $l < h$, and neither l nor h exist in the index, $\text{COUNT}(l, h)$ is

1. $\text{RANK}(l) - \text{RANK}(h) - 1$
2. $\text{RANK}(l) - \text{RANK}(h)$
3. $\text{RANK}(l) - \text{RANK}(h) + 1$
4. $\text{RANK}(h) - \text{RANK}(l) - 1$
5. $\text{RANK}(h) - \text{RANK}(l)$
6. $\text{RANK}(h) - \text{RANK}(l) + 1$
7. $\text{RANK}(h) + \text{RANK}(l) - 1$
8. $\text{RANK}(h) + \text{RANK}(l)$
9. $\text{RANK}(h) + \text{RANK}(l) + 1$

Solution: $\text{RANK}(h) - \text{RANK}(l)$. We want to count all the keys x such that $l \leq x \leq h$, which is equivalent to all the keys $x \leq h$ minus all the keys $x < l$. $\text{RANK}(h)$ counts all keys x s.t. $x \leq h$. $\text{RANK}(l)$ counts all keys s.t. $x \leq l$, which is equivalent to all the keys $x < l$ since l is not in the index. So $\text{RANK}(h) - \text{RANK}(l)$ is the right answer.

Now that you know how to reduce a $\text{COUNT}()$ query to a constant number of $\text{RANK}()$ queries, you want to figure out how to implement $\text{RANK}()$ in sub-linear time. None of the tree data structures that you studied in 6.006 supports optimized $\text{RANK}()$ out of the box, but you just remembered that tree data structures can respond to some queries faster if the nodes are cleverly augmented with some information.

(i) [1 point] In order to respond to $\text{RANK}()$ queries in sub-linear time, each node $node$ in the tree will be augmented with an extra field, $node.\gamma$. Keep in mind that for a good augmentation, the extra information for a node should be computed in $O(1)$ time, based on other properties of the node, and on the extra information stored in the node's subtree. The meaning of $node.\gamma$ is

1. the minimum key in the subtree rooted at $node$
2. the maximum key in the subtree rooted at $node$
3. the height of the subtree rooted at $node$
4. the number of nodes in the subtree rooted at $node$
5. the rank of $node$

6. the sum of keys in the subtree roted at $node$

Solution: The number of nodes in the subtree rooted at $node$ (also known as subtree size).

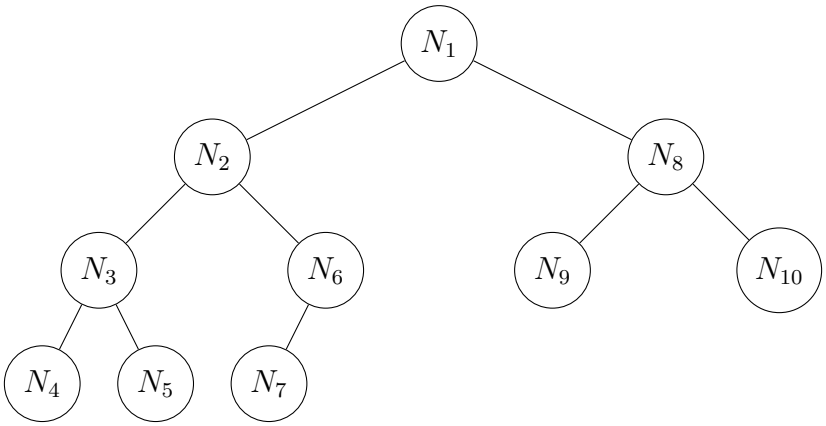
γ cannot be the node's rank, because that cannot be computed solely based on information in the node's subtree. Augmentations that report key information (minimum, maximum, sum) don't reveal any information about a node's rank. A tree's height is only roughly related to the number of nodes in the tree.

(j) [1 point] How many extra *bits* of storage per node does the augmentation above require?

1. $O(1)$
2. $O(\log(\log N))$
3. $O(\log N)$
4. $O(\log^2 N)$
5. $O(\sqrt{N})$
6. $O(N)$

Solution: The number of nodes in a subtree is at most N, so the γ field in each node needs to be $O(\log N)$ bits wide to be able to store numbers between 0 and N.

The following questions refer to the tree below.



(k) [1 point] $N_4.\gamma$ is

1. 0
2. 1
3. 2
4. the key at N_4

Solution: 1. A leaf's subtree has exactly one node.

(l) [1 point] $N_3.\gamma$ is

1. 1
2. 2
3. 3
4. the key at N_4
5. the key at N_5
6. the sum of keys at $N_3 \ldots N_5$

Solution: 3

(m) [1 point] $N_2.\gamma$ is

1. 2
2. 3
3. 4
4. 6
5. the key at N_4
6. the key at N_7
7. the sum of keys at $N_3 \ldots N_5$

Solution: 6

(n) [1 point] $N_1.\gamma$ is

1. 3
2. 6
3. 7
4. 10
5. the key at N_4
6. the key at N_{10}
7. the sum of keys at $N_1 \ldots N_{10}$

Solution: 10. The questions should show the pattern for computing $node.\gamma$, which is

$$\begin{aligned} node.\gamma &= 1 + \gamma(left) + \gamma(right) \\ \text{NIL}.\gamma &= 0 \end{aligned}$$

(o) [6 points] Which of the following functions need to be modified to update γ? If a function does not apply to the tree for the range index, it doesn't need to be modified. (True / False)

1. INSERT

2. DELETE
3. ROTATE-LEFT
4. ROTATE-RIGHT
5. REBALANCE
6. HEAPIFY

Solution: ROTATE-LEFT, ROTATE-RIGHT, and REBALANCE

γ needs to be updated in the same situations where a node's height is updated. The AVL methods that call UPDATE-HEIGHT are ROTATE-LEFT, ROTATE-RIGHT, and REBALANCE. INSERT and DELETE rely on the rotation methods to update the height.

(p) [1 point] What is the running time of a COUNT() implementation based on RANK()?

1. $O(1)$
2. $O(\log(\log N))$
3. $O(\log N)$
4. $O(\log^2 N)$
5. $O(\sqrt{N})$

Solution: $O(\log N)$, which can be achieved by the algorithm below.

```
RANK(tree, k)
 1  r = 0
 2  node = tree.root
 3  while node ≠ NIL
 4      if k < node.key
 5          node = node.left
 6      else
 7          if node.left ≠ NIL
 8              r = r + 1 + node.left.γ
 9          else
10              r = r + 1
11          if node.key == k
12              return r
13          node = node.right
14  return r
```

The structure is very similar to that of FIND – in the worst case, RANK visits all the nodes on a path from the root to a leaf, and does a constant amount of computation at each node on the path.

After the analytics data is plotted using Pretty Graph™, the CEO can hover the mouse cursor over one of the bars, and the graph will show a tooltip with the information represented by that bar. To

support this operation, the range index needs to support a LIST(l, h) operation that returns all the keys between l and h as quickly as possible.

LIST(l, h) cannot be sub-linear in the worst case, because LIST$(-\infty, +\infty)$ must return all the keys in the index, which takes $\Omega(n)$ time. However, if LIST only has to return a few elements, we would like it to run in sub-linear time. We formalize this by stating that LIST's running time should be $T(N) + \Theta(L)$, where L is the length of the list of keys output by LIST, and $T(N)$ is sub-linear.

Inspiration (or your 6.006 TA) strikes again, and you find yourself with the following pseudocode for LIST.

```
LIST(tree, l, h)
1  lca = LCA(tree, l, h)
2  result = []
3  NODE-LIST(lca, l, h, result)
4  return result

NODE-LIST(node, l, h, result)
1  if node == NIL
2      return
3  if l ≤ node.key and node.key ≤ h
4      ADD-KEY(result, node.key)
5  if node.key ≥ l
6      NODE-LIST(node.left, l, h, result)
7  if node.key ≤ h
8      NODE-LIST(node.right, l, h, result)

LCA(tree, l, h)
1  node = tree.root
2  until node == NIL or (l ≤ node.key and h ≥ node.key)
3      if l < node.key
4          node = node.left
5      else
6          node = node.right
7  return node
```

(q) [1 point] LCA most likely means

1. last common ancestor
2. lowest common ancestor
3. low cost airline
4. life cycle assessment

5. logic cell array

Solution: Visiting the Wikipedia page for LCA shows that "lowest common ancestor" is the only computer science-related interpretation of LCA.

(r) [1 point] The running time of $\text{LCA}(l, h)$ for the trees used by the range index is

1. $O(1)$
2. $O(\log(\log N))$
3. $O(\log N)$
4. $O(\log^2 N)$
5. $O(\sqrt{N})$

Solution: $O(\log N)$, the height of the tree.

(s) [1 point] Assuming that ADD-KEY runs in $O(1)$ time, and that LIST returns a list of L keys, the running time of the NODE-LIST call at line 3 of LIST is

1. $O(1)$
2. $O(\log(\log N))$
3. $O(\log N)$
4. $O(\log^2 N)$
5. $O(\sqrt{N})$
6. $O(1) + O(L)$
7. $O(\log(\log N)) + O(L)$
8. $O(\log N) + O(L)$
9. $O(\log^2 N) + O(L)$
10. $O(\sqrt{N}) + O(L)$

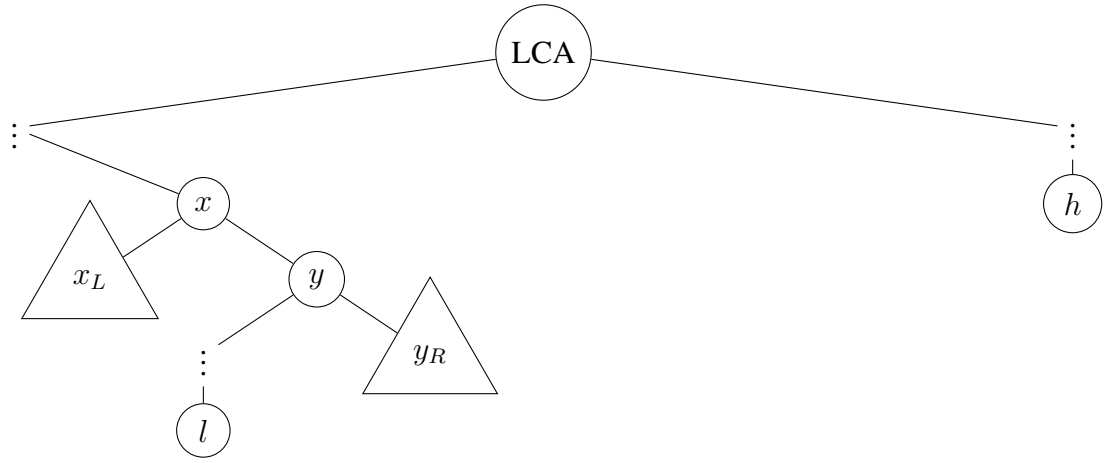
Solution: $O(\log N) + O(L)$.

The NODE-LIST call at line 3 performs a pre-order traversal starting at the least common ancestor of the nodes containing keys l and h. If the keys l and/or h are not in the BST, we use the positions of imaginary nodes that would be created by inserting l and/or h using the BST insertion algorithm (no re-balancing).

The traversal ignores some sub-trees. Line 5 prunes the left sub-tree of any node whose key is $\leq l$ because, by the BST invariant, the keys of all the nodes in the sub-tree will be smaller than $node.key$, which is $\leq l$. In a symmetric fashion, line prunes sub-trees where all the nodes' keys are guaranteed to be greater than h.

NODE-LIST spends $O(1)$ time for each node it visits, so the total running time is $O(V)$, where V is the number of visited nodes. We split up $V = L + X$, where L is the number of nodes returned from LIST, and X is the number of nodes that are visited, but not output (extra nodes, or overhead). We argue that $X = O(h)$, where h is the height of the BST. Because the BST is an AVL, we use $h = O(\log N)$ to conclude that the total running time of the NODE-LIST call in LIST is $O(\log(N) + L)$.

We'll sketch a proof that the extra nodes in the left sub-tree of the LCA node are all on the path from the LCA to the l node. A symmetric proof covers the right sub-tree of the LCA node. The proof refers to the tree below.



Every node on the path from the LCA node to the l node falls under one of the following two cases.

- The node l belongs to the node's right sub-tree. Let x be the node's key, and x_L be the node's left sub-tree. Since the l node is in the x node's left sub-tree, it follows that $l > x$ (by theBST invariant), so the x_L sub-tree will be pruned by line 5 in NODE-LIST. So NODE-LIST will never "stray" to the left of the path from the LCA node to the l node.
- The node l belongs to the node's left sub-tree. Let y be the node's key, and y_R be the node's right sub-tree. Since the l node is in the x node's right sub-tree, it follows that $l < y$ (by the BST's invariant). Every node in y_R will have a key greater than y (BST invariant), so all the keys in y_R are greater than l. Furthermore, because y_R is in LCA's left sub-tree, all the keys in y_R are smaller than the key in LCA, which is $\leq h$. It follows that all the nodes in y_R belong to the $(l, h]$ interval, and their keys will be returned from NODE-LIST. So, whenever NODE-LIST "strays" to the right of the path from the LCA node to the l node, it will output all the nodes it encounters.

The two cases above show that all the extra nodes (visited but not added to the result) in the LCA node's left sub-tree are on the path from LCA to l, which means $X = O(h)$. A symmetric proof can be used to show that all the extra nodes in the LCA node's right sub-tree are on the path from LCA to h.

(t) [1 point] Assuming that ADD-KEY runs in $O(1)$ time, and that LIST returns a list of L keys, the running time of LIST is

1. $O(1)$
2. $O(\log(\log N))$

3. $O(\log N)$
4. $O(\log^2 N)$
5. $O(\sqrt{N})$
6. $O(1) + O(L)$
7. $O(\log(\log N)) + O(L)$
8. $O(\log N) + O(L)$
9. $O(\log^2 N) + O(L)$
10. $O(\sqrt{N}) + O(L)$

Solution: $O(\log N) + O(L)$. According to the previous questions, LCA takes $O(\log N)$ time, and the NODE-LIST call takes $O(\log N) + O(L)$ time.

(u) [20 points] Prove that LCA is correct.

Solution: We will argue that LCA(l, h) returns the least-common ancestor, defined as the root of the smallest sub-tree that contains both l and h. If the keys l and/or h are not in the BST, we augment the tree with the imaginary nodes that would be created by inserting l and/or h using the BST insertion algorithm (no re-balancing).

Based on the definition above, we will prove that LCA is correct in two stages: first we prove that LCA maintains the invariant that l and h belong to the sub-tree rooted at $node$, then we prove that LCA returns the root of the smallest sub-tree that contains both l and h.

LCA **maintains the invariant that** l **and** h **belong to the sub-tree rooted at** $node$. $node$ starts out at the tree's root, so the invariant starts out to be true at line 2. The loop continues as long as both l and h are smaller than $node$'s key, or l and h are both bigger than $node$'s key. If l and h are smaller, then the condition on line 3 will be true, so both l and h are in $node$'s left sub-tree, by the BST invariant, so line 4 preserves the invariant. Similarly, if l and h are larger, they're both in $node$'s right sub-tree, so line 6 preserves the invariant.

LCA **returns the root of the smallest sub-tree that contains both** l **and** h. Proof by contradiction:

Let r be the key of the node returned by LCA, and c be the key of the node that is the correct answer. By the invariant above, l and h are in the sub-tree rooted at the r node, so the c node is in the sub-tree rooted at r.

If c is in the r node's left sub-tree, it follows that $c < r$. Since l and h are in the sub-tree rooted at the c node, they are in r's left sub-tree, so $l < r$ and $h < r$ by the BST invariant. But in that case, the condition on line 2 will be true when idnode is the r node, so the r node should not be returned from LCA. Contradiction.

The case where c is in the r node's right sub-tree is symmetrical, as both l and h will be in r's sub-tree and $l > r$, $h > r$.

Problem 3-2. [55 points] **Digital Circuit Layout**

Your AMDtel internship is off to a great start! The optimized circuit simulator cemented your reputation as an algorithms whiz. Your manager capitalized on your success, and promised to deliver the Bullfield chip a few months ahead of schedule. Thanks to your simulator optimizations, the engineers have finished the logic-level design, and are currently working on laying out the gates on the chip. Unfortunately, the software that verifies the layout is taking too long to run on the preliminary Bullfield layouts, and this is making the engineers slow and unhappy. Your manager is confident in your abilities to speed it up, and promised that you'll "do your magic" again, in "one week, two weeks tops".

A chip consists of logic gates, whose input and output terminals are connected by wires (very thin conductive traces on the silicon substrate). AMDtel's high-yield manufacturing process only allows for horizontal or vertical wires. Wires must not cross each other, so that the circuit will function according to its specification. This constraint is checked by the software tool that you will optimize. The topologies required by complex circuits are accomplished by having dozens of layers of wires that do not touch each other, and the tool works on one layer at a time.

(a) [1 point] Run the code under the python profiler with the command below, and identify the method that takes up most of the CPU time. If two methods have similar CPU usage times, ignore the simpler one.

```
python -m cProfile -s time circuit2.py < tests/10grid_s.in
```

Warning: the command above can take 15-60 minutes to complete, and bring the CPU usage to 100% on one of your cores. Plan accordingly. If you have installed PyPy successfully, you can replace `python` with `pypy` in the command above for a roughly 2x speed improvement.

What is the name of the method with the highest CPU usage?

Solution: The first line in the profiler output identifies `intersects` as the method with the largest CPU usage.

(b) [1 point] How many times is the method called?

Solution: The first line in the profiler output indicates that `intersects` is called 187590314 times.

The method that has the performance bottleneck is called from the `CrossVerifier` class. Upon reading the class, it seems that the original author was planning to implement a *sweep-line* algorithm, but couldn't figure out the details, and bailed and implemented an inefficient method at the last minute. Fortunately, most of the infrastructure for a fast sweep-line algorithm is still in place. Furthermore, you notice that the source code contains a trace of the working sweep-line algorithm, in the `good_trace.jsonp` file.

Sweep-line algorithms are popular in computational geometry. Conceptually, such an algorithm sweeps a vertical line left to right over the plane containing the input data, and performs operations when the line "hits" point of interest in the input. This is implemented by generating an array

containing all the points of interest, and then sorting them according to their position along the horizontal axis (x coordinate).

Read the source for `CrossVerifier` to get a feel for how the sweep-line infrastructure is supposed to work, and look at the good trace in the visualizer that we have provided for you. To see the good trace, copy `good_trace.jsonp` to `trace.jsonp`

```
cp good_trace.jsonp trace.jsonp
```

On Windows, use the following command instead.

```
copy good_trace.jsonp trace.jsonp
```

Then use Google Chrome to open `visualizer/bin/visualizer.html`

The questions below refer to the fast sweep-line algorithm shown in `good_trace.jsonp`, not to the slow algorithm hacked together in `circuit2.py`.

(c) [5 points] The x coordinates of points of interest in the input are (True / False)

1. the x coordinates of the left endpoints of horizontal wires
2. the x coordinates of the right endpoints of horizontal wires
3. the x coordinates of midpoints of horizontal wires
4. the x coordinates where horizontal wires cross vertical wires
5. the x coordinates of vertical wires

Solution: The x coordinates of both endpoints of horizontal wires, and of vertical wires. The algoritm doesn't handle wire midpoints specially. Crossing points cannot possibly be points of interest, because they are the algorithm's output, not input.

(d) [1 point] When the sweep line hits the x coordinate of the left endpoint of a horizontal wire

1. the wire is added to the range index
2. the wire is removed from the range index
3. a range index query is performed
4. nothing happens

Solution: The wire is added to the range index when the sweep line hits the x coordinate of the left endpoint.

(e) [1 point] When the sweep line hits the x coordinate of the right endpoint of a horizontal wire

1. the wire is added to the range index
2. the wire is removed from the range index
3. a range index query is performed
4. nothing happens

Solution: The wire is removed from the range index when the sweep line hits the X coordinate of the right endpoint.

(f) [1 point] When the sweep line hits the x coordinate of the midpoint of a horizontal wire

1. the wire is added to the range index
2. the wire is removed from the range index
3. a range index query is performed
4. nothing happens

Solution: Nothing happens.

(g) [1 point] When the sweep line hits the x coordinate of a vertical wire

1. the wire is added to the range index
2. the wire is removed from the range index
3. a range index query is performed
4. nothing happens

Solution: A range index is performed.

(h) [1 point] What is a good invariant for the sweep-line algorithm?

1. the range index holds all the horizontal wires to the left of the sweep line
2. the range index holds all the horizontal wires "stabbed" by the sweep line
3. the range index holds all the horizontal wires to the right of the sweep line
4. the range index holds all the wires to the left of the sweep line
5. the range index holds all the wires to the right of the sweep line

Solution: The invariant for the sweep-line algorithm is that the range index holds all the horizontal wires "stabbed" by the sweep line.

(i) [1 point] When a wire is added to the range index, what is its corresponding key?

1. the x coordinate of the wire's midpoint
2. the y coordinate of the wire's midpoint
3. the segment's length
4. the x coordinate of the point of interest that will remove the wire from the index

Solution: The only correct answer is "the Y coordinate of the wire's midpoint". Only horizontal wires are added to the range index, and all their points have the same Y coordinate.

Modify `CrossVerifier` in `circuit2.py` to implement the sweep-line algorithm discussed above. If you maintain the current code structure, you'll be able to use our visualizer to debug your implementation. To use our visualizer, first produce a trace.

```
TRACE=jsonp python circuit2.py < tests/5logo.in > trace.jsonp
```

On Windows, run the following command instead.

```
circuit2_jsonp.bat < tests/5logo.in > trace.jsonp
```

Then use Google Chrome to open `visualizer/bin/visualizer.html`

(j) [1 point] Run your modified code under the python profiler again, using the same test case as before, and identify the method that takes up the most CPU time.

What is the name of the method with the highest CPU usage? If two methods have similar CPU usage times, ignore the simpler one.

Solution: The first line in the profiler output identifies `count` as the method with the highest CPU usage.

(k) [1 point] How many times is the method called?

Solution: The first line in the profiler output indicates that `count` is called 20000 times.

(l) [40 points] Modify `circuit2.py` to implement a data structure that has better asymptotic running time for the operation above. Keep in mind that the tool has two usage scenarios:

- Every time an engineer submits a change to one of the Bullhorn wire layers, the tool must analyze the layer and report the number of wire crossings. In this late stage of the project, the version control system will automatically reject the engineer's change if it causes the number of wire crossings to go up over the previous version.
- Engineers working on the wiring want to see the pairs of wires that intersect, so they know where to focus ther efforts. To activate this detailed output, run the tool using the following command.
  `TRACE=list python circuit2.py < tests/6list_logo.in`
  On Windows, run the following command instead.
  `circuit2_list.bat < tests/6list_logo.in`

When your code passes all tests, and runs reasonably fast (the tests should complete in less than 60 seconds on any reasonably recent computer), upload your modified `circuit.py` to the course submission site.

Solution: The solution archive on the course Web site contains the staff's solution and secret test cases.

6.006 Introduction to Algorithms
Fall 2011