1 Problem Solving: Quiz #1

1.1 Quiz #1, Problem 2
(a) Problem: Given an $O(n)$ time algorithm for checking whether an arbitrary binary tree is a binary search tree.

The most important thing for this part of the problem was to remember the invariant of a binary search tree. At each node $x$ in the tree, we require that

\[
\begin{align*}
    y.\text{key} < x.\text{key} & \quad \forall y \in \text{SUBTREE}(x.\text{left}) \\
    y.\text{key} > x.\text{key} & \quad \forall y \in \text{SUBTREE}(x.\text{right})
\end{align*}
\]

The problem asks you to write code to check this condition, for every $x$, in $O(n)$ time. (It would be easy to do this directly from the definition in $\Theta(n^2)$ time, but we must do better.)

About three-quarters of the class made the same mistake on this problem, which was to check the condition on the left subtree (for example) by just checking $x.\text{left}.\text{key} \leq x.\text{key}$. This is insufficient because it does not imply that $x.\text{key}$ is greater than every key in the subtree. Indeed, $x.\text{left}$ will usually be one of the middle elements in that subtree, so nearly half of all the elements in the subtree are larger than this, and we are not checking that $x$’s key is larger than these.

On the other hand, it is possible to verify that the condition holds for the subtree by checking only a single $y$ (for each $x$), namely, the $y$ that is the maximum in that subtree.\footnote{Actually, implementing this check would actually run in $O(n)$ time, at least on a perfectly balanced tree, by the analysis we did for BUILD-HEAP.}

We can find this $y$ for each node $x$ in one a few different ways:
– Compute the minimum and maximum value of each subtree, bottom-up.

If we first recurse on the left subtree and get \((min_L, max_L)\), then \(max_L\) will be \(y\), so we just need to check \(max_L \leq x.key\). Likewise, we recurse on the right subtree to get \((min_R, max_R)\) and then compare \(x.key \leq min_R\). Finally, we return the minimum and maximum from this subtree as \((\min\{min_L, x\}, \max\{x, max_R\})\).\(^2\)

– Enumerate the nodes of the tree using SUCCESSOR. When we get to node \(x\), we will have just come from \(y\). If we keep track of the key from the last visited node during the traversal, we can check the BST condition with a single comparison per node.

– Perform an in-order traversal of the tree, copying the keys into an array. The resulting tree will be sorted if and only if the BST condition is satisfied.

Indeed, notice that \(y\) and \(x\) end up adjacent to each other in the array. (The reason that we can check the condition on the whole left-subtree just by comparing to \(y\) is the same reason we can check that an array is sorted — which naively, requires comparing every pair of elements — only by comparing adjacent elements.)

An alternative solution for this problem is to recurse through the tree, keeping track of the constraints implied by all the ancestors of the current node. For each subtree, the ancestors imply that the keys must lie in a range \((a, b)\) for some \(a\) and \(b\).

Specifically, the value \(a\) is the largest of the keys of all ancestors for which we are in the right subtree, and \(b\) is the smallest of the keys of all ancestors for which we are in the left subtree. If we recurse over the tree, keeping track of \(a\) and \(b\) as we go, then each node simply needs to check \(a \leq x.key \leq b\). This takes constant time per node, so applying this over the whole tree takes \(\Theta(n)\) time.

(b) Problem: Give an algorithm that determines the number of “violations” of the binary search tree invariant in a given binary tree. This algorithm should have running time \(O(n + k)\), where \(k\) is the number of violations.

There is not a lot to say about this problem. In order to solve it, you had to recall that insertion sort runs in \(O(n + k)\) time, where \(k\) is the number of inversions, and you had to realize that a “violation” in this problem is the same thing as an inversion in the list of keys that would result from an in-order traversal.

\(^2\)Here, we are assuming that the answer is \((\infty, -\infty)\) for an empty subtree. That is the reason why we need to compute \(\min\) and \(\max\) in the return value.
If you spotted those things, then the solution is easy to describe: perform an in-order traversal
to produce an array and then insertion sort the array, counting inversions as you go.

(c) Problem: Give an $O(n)$ algorithm that takes an arbitrary binary search tree as input and
returns a binary search tree with height $O(\log n)$.

This problem is particularly easy if we remember the correspondence between binary search
trees and sorted arrays that we used in part (a). Simply copy the elements into a sorted array
(in $O(n)$ time) and then build a perfectly balanced tree from this (in $O(n)$ time) by making
the middle element the root and then recursively making trees from the parts before and after
the middle in the array.

The second part can be viewed as applying the tree → array mapping in reverse, choosing from amongst all the BSTs that produce the
same array, the one that is perfectly balanced.

Many students attempted to solve the problem by performing rotations on the tree. This idea
has a couple of problems.

First of all, it is not clear that this will ever terminate. For example, some students simply
performed left or right rotations at each node until it was balanced. However, we saw before
that single rotations do not always restore balance at a node. This is why AVL trees need
double-rotations! In the cases where double rotations are used in AVL, a single rotation
simply moves the imbalance from one side to the other. Hence, this method will go into an
infinite loop.

It is important to note that we never even analyzed what happens when you perform rotations
with arbitrary imbalance between the two subtrees. We only considered the cases that occur
in an AVL INSERT, where one side has height at most two more than the other. So we do not
have any proof that rotations would improve balance in general.

Indeed, even checking the simplest case (the single rotation case) with an arbitrary imbalance
shows that it does not work properly:

We can see that, while we have restored balance at the root, we have created an imbalance
in a subtree. So any correct algorithm using this approach would have to process nodes
multiple times.

One could hope to prove that these rotations are improving some notion of the total imbalance in the tree. If true, that would show that
rotations eventually balance the tree. Unfortunately, the simplest ideas for how to define total imbalance do not work: in some cases,
imbalance does not improve because it is simply shifted from one place to another. (Perhaps a more refined notion of “total imbalance” could prove that this works, but the author could not think of one.)

Finally, even ignoring the (much more serious) correctness issues, it seems very unlikely that this method could achieve the desired running time. To achieve $O(n)$ time, we would need to prove not only that we can restore balance by rotations alone, but in fact, we can do so with only $O(n)$ rotations! That would be fascinating, if true. More likely is that it would require $\Omega(n \log n)$ rotations. However, if we are going to look for an $O(n \log n)$ algorithm, there is a much simpler option: just insert each element into a new AVL tree. This takes $\Theta(\log n)$ time per insert, so $\Theta(n \log n)$ overall.

1.2 Quiz #1, Problem 5

Problem 5 is a great example of a data structures problem. We are asked to figure out what information to store and how to organize that information in order to efficiently support two operations on a set of prices:

- **ADD**($x$) adds price $x$ to the set in $O(\log n + k)$ time
- **QUERY**() returns the $k$ “middle” prices — i.e., if $A$ contained a sorted list of these prices, this would return $A[\frac{1}{2}(n-k) : \frac{1}{2}(n+k)]$ — in $O(k)$ time

Here is an approach that works for solving many data structures problems. Consider each of the operations, one at a time, and ask yourself this question:

What could I store / maintain to make this easy?

If you can think of some information to store (or some invariant to maintain on the data you’re already storing) to make the operation easy, then you’re making progress. You’ve solved one operation. Now, you need to go through the other operations and make sure you can efficiently maintain that information through those other operations as well. This technique is easiest to explain through examples. This quiz problem will give our first example.

In principle, we can consider the operations we need to support in any order. However, it is usually a good heuristic to start with the hardest operation, that is, the operation whose constraints are the hardest to satisfy.

In this problem, the hardest operation is definitely **QUERY**() because we are required to return an answer in $O(k)$ time. Just writing down the answer takes $O(k)$ time, so they are asking us for an optimal algorithm. That is asking a lot.\(^3\)

Now that we’ve picked an operation, we can ask ourselves the question above: what could we store / maintain to make **QUERY** (in $O(k)$ time!) easy? Forget about **ADD** for the time being.\(^3\)

\(^3\)If there was any cleverness required on this problem, it was simply to notice that how much was being asked of us for **QUERY**. Once we spot that, we can deduce the solution in a straight-forward manner.
Imagine we can ask for any data structure we want and it will be available for us to use in QUERY. The data structure genie will grant a wish for anything we want. What should we ask for?

Surprisingly, there are very few options! (This is good news, it means we have fewer alternatives to consider.) We could ask for a hash table or an AVL tree containing the set of prices, but neither would give us the $k$ middle elements in $O(k)$ time. A heap can’t do that either. What would do it? What would make this operation easy?

As far as the author is aware, there are only two data structures that would let us get the $k$ middle elements in $O(k)$ time. We will consider each in turn.

**Option 1: A Sorted Array of Prices** If we stored the prices in a sorted array, $A$, then we can find the $k$ middle prices in $A[\lfloor \frac{1}{2} (n - k) \rfloor : \lceil \frac{1}{2} (n + k) \rceil]$. We can copy these in $O(k)$ time and return them. So this indeed makes it possible to support QUERY.

To get a complete solution, though, we need to be able to maintain our data structure during all the required operations. So we next need to consider the other operation: $\text{ADD}(x)$. If we are storing a sorted list of prices, then we need to insert $x$ into that list. Unfortunately, that takes $\Theta(n)$ time, much more than we are allowed.

So storing an array of prices doesn’t work, or rather, it works for QUERY but not ADD. We need to back up and think of another option for QUERY. Hopefully, our next idea will work for ADD also.

**Option 2: A List of the $k$ Middle Prices** You know what would make it really easy to find the $k$ middle prices: having a list of the $k$ middle prices. Why not ask the genie for that? Maybe it sounds too good to be true, but actually, this is the solution that works!

As above, we now need to consider $\text{ADD}(x)$. Somehow we need to update our list of the $k$ middle elements. There are a few different cases depending on how $x$ relates to the values currently in the list.

Suppose that $x$ is smaller than all the middle elements. Inserting a new element in the range $A[0 : \lfloor \frac{1}{2} (n - k) \rfloor]$ pushes all of the later prices back. This means that we potentially need to drop the largest element from the list of $k$ and add the element just before the smallest of the list of $k$.

Unfortunately, we don’t have any way to do this. (No, really. We’re not even storing the other prices yet, so there is absolutely no way to do it.) We’re stuck once again. But unlike above, we haven’t shown that this approach can’t work. So we should keep trying....

Here’s the situation: we need a way to get the element just before the $k$ middle. (Put another way, we need a way to get the largest of all those elements that are smaller than all the middle elements.) We can find a solution by returning to our favorite question: what could we store / maintain to make this easy?

This question is much easier to answer. We learned about data structures in class that will let us do this (two, in fact). We can, for example, use a max-heap: put all the elements smaller than all the $k$ middle elements into the heap, and we can call $\text{MAX}$ on the heap to get the one we want.

---

$^4$Note that there is nothing additional we could store to make this faster. Our invariant that the list is sorted means we may need to move $\Theta(n)$ elements to make room for $x$, and that takes $\Theta(n)$ time.
This solves our problem for the case when \( x \) is smaller than the \( k \) middle elements. But we have a similar problem when \( x \) is larger than the \( k \) middle elements: we might need to drop the smallest of the \( k \) middle elements and add the smallest of the elements larger than \( k \) middle. We can solve that in the analogous way: store a min-heap of the elements larger than the \( k \) middle.

Now that we have decided to store more data to make \textsc{Add} easy, we need to make sure that we can maintain this during the other operations. However, the only other operation is \textsc{Query}, so there is nothing to worry about.

**Solution**  Our solution is to maintain the following data:

- a list, \( \text{middle} \), containing the \( k \) middle elements
- a max-heap, \( \text{lower} \), containing the elements below the middle
- a min-heap, \( \text{upper} \), containing the elements above the middle

We maintain the following invariants:

1. All the keys in \( \text{lower} \) are \( \leq \) those in \( \text{middle} \), and all the keys in \( \text{upper} \) are \( \geq \) those in \( \text{middle} \).

2. The list \( \text{middle} \) has length \( k \), and the lengths of \( \text{lower} \) and \( \text{upper} \) differ by at most one.

The second invariant ensures that \( \text{middle} \) really contains the \( k \) middle elements.

For actually implementing our operations, the invariants are the key. They ensure that our answer to \textit{Query} is correct. So all we need to think about in \textit{Add} is how to maintain those invariants. In general, when we have the right invariants, the operations should basically write themselves. Here is pseudocode for the two operations.

**QUERY()
1 \hspace{1em} \text{return middle}

**ADD(x)
1 \hspace{1em} \text{if } x < \text{Min}(\text{middle})
2 \hspace{2em} \text{then Max-Heap-Insert(\text{lower}, x)}
3 \hspace{2em} \text{else if } x \leq \text{Max}(\text{middle})
4 \hspace{3em} \text{then Array-Append(\text{middle}, x)}
5 \hspace{3em} x := \text{Max}(\text{middle})
6 \hspace{3em} \text{Array-Remove(\text{middle}, x)}
7 \hspace{2em} \text{Min-Heap-Insert(\text{upper}, x)}
8 \hspace{1em} \text{if Max-Heap-Size(\text{lower}) < Min-Heap-Size(\text{upper}) - 1}
9 \hspace{2em} \text{then Shift-Right()}
10 \hspace{1em} \text{if Max-Heap-Size(\text{lower}) > Min-Heap-Size(\text{upper}) + 1}
11 \hspace{2em} \text{then Shift-Left()}

The **SHIFT-LEFT** and **SHIFT-RIGHT** routines handle the case when the two heaps have sizes that no longer match the invariant. These fix the size invariant while maintaining the key invariant.

**SHIFT-LEFT()**
1. \( x := \text{EXTRACT-MIN}(\text{upper}) \)
2. \( \text{ARRAY-APPEND}(\text{middle}, x) \)
3. \( y := \text{MIN}(\text{middle}) \)
4. \( \text{ARRAY-REMOVE}(\text{middle}, y) \)
5. \( \text{MAX-HEAP-INSERT}(\text{lower}, y) \)

**SHIFT-RIGHT()**
1. \( x := \text{EXTRACT-MAX}(\text{lower}) \)
2. \( \text{ARRAY-APPEND}(\text{middle}, x) \)
3. \( y := \text{MAX}(\text{middle}) \)
4. \( \text{ARRAY-REMOVE}(\text{middle}, y) \)
5. \( \text{MIN-HEAP-INSERT}(\text{lower}, y) \)

It is easy to check that QUERY and ADD have the required time complexity.

### 1.2.1 The Clever, Simple Solution

Now that we have seen the straight-forward, sensible solution. Let me mention one other solution.

Our analysis above was driven by the fact that QUERY had such a strong constraint on its running time. However, it is often possible to get around such constraints by moving work around. In particular, since we have so much more time available to us for ADD, you can imagine simply performing QUERY as part of the ADD operation and then remembering the result. We can then return the remembered result for later QUERY’s.

If we do this, then our problem becomes coming up with a solution that performs ADD followed by QUERY with total time \( O(\log n + k) \). This is much more manageable (which means there are more potential solutions).

The running time of \( O(\log n + k) \) is achieved by an operation called **RANGE-QUERY**(\(a, b\)) that can be performed on an AVL tree. This operation returns all keys \( x \) such that \( a \leq x \leq b \), and it takes \( O(\log n + k) \) time, where \( k \) is the number of keys returned. I’ll leave it as an easy exercise to figure out how to do this. (The operation itself is exactly what you’d expect. It’s only showing that it takes \( O(\log n + k) \) that takes a bit of thought.)

Now, suppose that, instead of a regular AVL tree, we store an AVL tree augmented with rank (as mentioned in lecture). Recall that this allows us to find the \( j \)-th smallest element (for any \( j \)) in \( O(\log n) \) time. What now want is a way to do

\[
\text{RANGE-QUERY-BY-RANK}(\frac{1}{2}(n - k), \frac{1}{2}(n + k))
\]

The same approach as for the usual **RANGE-QUERY** can be used for this, and the same analysis will show that it takes \( O(\log n + k) \) time. (Now, this \( k \) is the same as above.)

### 2 Arbitrary-Size Integers

The topic for this week’s lectures and recitations is performing numerical calculations to arbitrary precision. The basic objects needed for this are arbitrary-precision, floating-point numbers.

A floating-point number with \( d \) digits of precision can be thought of as a number of the form \( y \times 10^e \), where \( y \) is an integer in the range \([10^{d-1}, 10^d]\) and \( e \) is a (small) integer exponent. We will have much more to say about floating-point numbers on Friday, but for today, let us restrict to the case \( e = 0 \). That is, let us just consider arithmetic with \( d \)-digit integers.
2.1 Addition

Addition of arbitrary-size integers is easy. In fact, most of have already seen an optimal algorithm for adding large integers, back in the first grade.

To describe this as an algorithm, we will suppose that integers are stored in an array, where each element of the array stores a digit in the range \( \{0, 1, \ldots, 9\} \). We will assume that we can already add numbers with a few digits. That is, we will assume that we have a built-in integer type (like Python’s `int`) that can store and add numbers with at least a few digits.\(^5\)

Here is the grade-school algorithm. It adds the numbers stored in the arrays \( A \) and \( B \) and puts the result into the array \( C \). For simplicity, this assumes that \( A \) and \( B \) both have \( d \) digits.\(^6\) Also note that we are storing low-order bits at low-indices in the array and high-order bits at high-indices.

\[
\text{INTEGER-ADD}(A, B, d, C)
\]

1. \( \text{carry} := 0 \)
2. \( \text{for } i := 0 \text{ to } d - 1 \)
3. \( \quad \text{do } r := A[i] + B[i] + \text{carry} \)
4. \( \quad C[i] := r \mod 10 \)
5. \( \quad \text{carry} := \lfloor r/10 \rfloor \)
6. \( \text{if } \text{carry} > 0 \)
7. \( \quad \text{ARRAY-APPEND}(C, \text{carry}) \)

At each step, we simply add the two corresponding digits in \( A \) and \( B \). However, the result of this addition may have two digits (e.g., \( 5 + 6 = 11 \)). The extra digit gets stored in \( \text{carry} \) and added into \( C \) on the next iteration.

The last two lines handle the case when the loop terminates with \( \text{carry} \) being nonzero. This means that the result of adding \( A \) and \( B \) is an integer with one more digit.

The numbers \( r \) and \( \text{carry} \) only store numbers with \( O(1) \) digits (in fact, just two digits), and we can perform arithmetic on such numbers in \( O(1) \) time. Since we perform \( d \) iterations, we can see that the loop takes \( \Theta(d) \) time. The extra shift on the last line (if it executes) also takes \( \Theta(d) \) time. Thus, we the grade-school algorithm for addition runs in \( \Theta(d) \) time.

Finally, note that this algorithm is clearly optimal. Just writing the answer into \( C \) takes \( \Theta(d) \) time since \( C \) has \( d \) (or \( d + 1 \)) digits.

2.2 Toom-Cook Multiplication

In lecture, we saw that is possible to multiply two \( d \)-digit numbers by performing 3 multiplications of \( d/2 \)-digit numbers, along with a constant number of additions and shifts. As we saw above, addition requires \( \Theta(d) \) time, as do shifts (which are even easier!). So we get the following recurrence

---

\(^5\)Theoretically, if these numbers have \( O(1) \) digits, then there are only \( O(1) \) additions we might need to perform, so we can just record all the answers in a look-up table. This is just telling your computer to do what we all did in grade-school: memorize all possible additions of small numbers.

\(^6\)If this is not true, then we can prepend 0’s to the smaller one.
relation for its running time:
\[ T(d) = 3T(d/2) + \Theta(d). \]

By the Master Theorem, the running time of this algorithm, due to Karatsuba, is \( \Theta(d^{\log_3 2}) \approx \Theta(d^{1.585}). \) (The recursion tree has \( d^{\log_3 2} \) leaves, which each require \( O(1) \) time. This cost dominates that of every other level in the tree.)

Unfortunately, there is no way to reduce this to just two multiplications of \( d/2 \)-digit numbers. However, we can play similar tricks if we break the numbers up into three parts. Specifically, if we are to multiply two integers, \( x \) and \( y \), we define
\[
  x = x_2 2^{2d/3} + x_1 2^{d/3} + x_0 \\
  y = y_2 2^{2d/3} + y_1 2^{d/3} + y_0
\]
(For simplicity, assume that \( d \) is divisible by 3.) If the digits of \( x \) are stored in array \( A \) in base 2 (rather than base 10, as in the previous section — that is, each digit \( A[i] \) is in \{0, 1\} rather than \{0, 1, \ldots, 9\}), then \( x_2 \) is just \( A[0 : d/3] \), \( x_1 \) is \( A[d/3 : 2d/3] \), and \( x_0 \) is \( A[2d/3 : d] \).

Our goal is to compute the product
\[
  xy = x_2 y_2 2^{4d/3} + (x_2 y_1 + x_1 y_2) 2^{3d/3} + (x_2 y_0 + x_1 y_1 + x_0 y_2) 2^{2d/3} + (x_1 y_0 + x_0 y_1) 2^{d/3} + x_0 y_0.
\]

Naively, this seems to require 8 multiplications. However, as with Karatsuba, we can perform a different set of multiplications and then deduce the coefficients above from these, using addition and bit shifts. Specifically, we perform the following five multiplications:
\[
  a = x_0 \times y_0 \\
  b = (x_2 + x_1 + x_0) \times (y_2 + y_1 + y_0) \\
  c = (x_2 - x_1 + x_0) \times (y_2 - y_1 + y_0) \\
  d = (4x_2 + 2x_1 + x_0) \times (4y_2 + 2y_1 + y_0) \\
  e = x_2 \times y_2
\]

Then a little arithmetic shows that
\[
  (b + c)/2 - a - e = x_2 y_0 + x_1 y_1 + x_0 y_2 \\
  (d - c)/6 + (a - b)/2 + 2e = x_2 y_1 + x_1 y_2 \\
  d/6 + b - c/3 - a/2 + 2e = x_1 y_0 + x_0 y_1
\]
and the other two coefficients are just \( a \) and \( e \).

Thus, it is possible to reduce multiplication of two \( d \)-digit numbers to 5 multiplications of \( d/3 \)-digit numbers along with a constant number of additions and multiplication/division by constants.\(^7\)

This gives us the Toom-Cook multiplication algorithm, which satisfies the recurrence
\[
  T(d) = 5T(d/3) + \Theta(d).
\]
\(^7\)This latter part can also be done in \( \Theta(n) \) time, but it involves more work to see that.
By the Master Theorem, just as above, the solution is $\Theta(d^{\log_3 5}) \approx \Theta(d^{1.465})$.

The above method can be generalized from 3 to any $k$. As we do this, the exponent of the running time tends to 1. Unfortunately, the cost of the additions also increases with $k$, and at some point, the cost of these additions dominates. (The exact point at which this happens is not known.)

In practice, the gradeschool multiplication algorithm, which takes $\Theta(d^2)$ time, is fastest for “small” numbers, while Toom-Cook multiplication is fastest for “medium”-sized numbers. For large numbers, the fastest algorithm in practice is the one with the best asymptotic complexity. This is the Schönhage–Strassen algorithm, based on the Fast Fourier Transform, which runs in $\Theta(d \log d \log \log d)$ time.