Practice Final Exam Solutions

This handout contains solutions to the practice final exam questions. ***Use this handout strategically***. We have found, through past experience, that students who do not make effective use of practice exams tend to perform markedly worse on the exams than students who do.

Before reading the solutions, please work through this checklist.

1. If you are not sure how to solve one of the problems here, even after a good amount of effort, ***do not look at the solution***. Instead, write down, in detail, everything you have tried so far to solve the problem. For each of those items, write down whether you decided to go with that option or whether you have ruled that option out. Then, ***reach out to someone else*** – your section leader, a friend in the course, Piazza, office hours, or the CLaIR – and send them that list. Your goal here is not to learn how to solve the specific problem at hand here, but rather to learn how to approach problems like these in general.

   It is perfectly fine if the question you're asking that person is "I have absolutely no idea how to even begin thinking about this problem." If that's the case, they can give you some tips to help you get started, and in doing so you'll likely learn some generalizable techniques for taking on trickier problems.

   Remember, once you have looked over the solution, you cannot "unsee" it. Looking at the solution to a problem you have not solved is not an effective way to study for the exam.

2. If you think you know how to solve one of the coding problems, ***do not look at the solutions until you have written your solution down and traced through the code, one line at a time, on a simple test case***. You'd be amazed how effective this is at helping you determine whether the code you've written is correct. If you're failing that test case, take some time to revise your answer. If that sends you back to step (1), that's okay – at least you'll know something to watch out for the second time around.

3. If you have a written solution for one of the problems, and you've traced through a sample input, and you think your code works, then you can then look at the solution. In doing so, be sure to read the "Why We Asked This Question" and "Common Mistakes" sections. Then, answer the following questions.

   1. What are all the errors in your solution? How serious is each of them?
   2. If your solution differs from the one written here, is that because you found an alternative solution route, or is there a reason we didn't use the solution you're proposing?

   Having done so, ***send your answers to these questions to someone else*** (a friend in the class, your section leader, Keith or Katherine in office hours, the CLaIR, etc.) and get them to confirm them. This will help identify whether you have any conceptual misunderstandings and is a great way to "close the loop" on the problem.

There are two major themes throughout this advice. First, ***keep another human in the loop***. You will want to ensure that you have someone else looking at what you're doing and offering feedback. Second, ***learn to solve problems plural rather than problem singular***. Your goal isn't to solve the specific problem in front of you; it's to learn how to solve the sorts of problems that arise in practice.

Problem One: Data Structures

```
int function1(int n) {
   Stack<int> values;
   for (int i = 0; i < n; i++) {
      values.push(i);
   }

   int result;
   while (!values.isEmpty()) {
      result = values.pop();
   }

   return result;
}
```

```
int function2(int n) {
   Queue<int> values;
   for (int i = 0; i < n; i++) {
      values.enqueue(i);
   }

   int result;
   while (!values.isEmpty()) {
      result = values.dequeue();
   }

   return result;
}
```

```
int function5(int n) {
   Set<int> values;
   for (int i = 0; i < n; i++) {
      values.add(i);
   }

   int result;
   for (int value: values) {
       result = value;
   }

   return result;
}
```

```
int function4(int n) {
   Vector<int> values;
   for (int i = 0; i < n; i++) {
      values.add(i);
   }

   int result;
   while (!values.isEmpty()) {
      result = values[0];
      values.remove(0);
   }

   return result;
}
```

n	*Time*	*Return Value*
100,000	0.137s	`99999`
200,000	0.274s	`199999`
300,000	0.511s	`299999`
400,000	0.549s	`399999`
500,000	0.786s	`499999`
600,000	0.923s	`599999`
700,000	0.960s	`699999`
800,000	1.198s	`799999`
900,000	1.335s	`899999`
1,000,000	1.472s	`999999`

Please do the following:

i. **(4 Points)** For each of these pieces of code, tell us its big-O runtime as a function of n. No justification is required.

The runtimes are as follows:

- `function1` runs in time $O(n)$.
- `function2` runs in time $O(n)$.
- `function3` runs in time $O(n \log n)$.
- `function4` runs in time $O(n^2)$.

Some explanations:

In `function1`, we do n pushes followed by n pops. The cost of each stack operation is amortized $O(1)$, so this means we're doing $2n$ operations at an effective cost of $O(1)$ each for a net total of $O(n)$. The same is true about queue operations; each one takes amortized time $O(1)$, which is why `function2` takes time $O(n)$ as well.

For `function3`, inserting an element into a set takes time $O(\log n)$ because the set is backed by a balanced BST. This means that the cost of inserting n elements is $O(n \log n)$. The cost of iterating over the tree is only $O(n)$ – it's basically an inorder traversal – so the net runtime is $O(n \log n)$.

For `function4`, adding n elements to the end of a vector takes time $O(n)$. However, removing from the *front* of a vector with n elements takes time $O(n)$, since we have to shift all the other elements back one position. This means that the overall runtime is $O(n^2)$.

ii. **(2 Points)** For each of these pieces of code, tell us whether that function could have given rise to the return values reported in the rightmost column of the table. No justification is required.

The answers:

- `function1` ***cannot*** produce the given output.
- `function2` will always produce this output.
- `function3` will always produce this output.
- `function4` will always produce this output.

Some explanations:

In `function1`, since elements are stored in a stack, the last element popped is the first element pushed, which would always be zero. Therefore, we'd expect to see a column of zeroes in the table, which doesn't match what's actually there.

In `function2`, the last element removed from the queue is the last element added to the queue, which, here, would be $n - 1$, matching the output.

In `function3`, since the `Set` type is backed by a binary search tree, it stores its elements in sorted order. Iterating over the set, therefore, will visit the elements in ascending order, so the last element iterated over by the loop would be $n - 1$, matching the output.

Finally, in `function4`, we remove elements from the vector in the reverse order in which they're added, matching the queue's ordering and making the last element visited exactly $n - 1$.

iii. **(2 Points)** Which piece of code did we run? How do you know? Justify you answer in at most fifty words.

First, notice that the runtime appears to be $O(n)$; doubling the size of the inputs roughly doubles the runtime. That leaves `function1` and `function2` as choices, and `function1` has the wrong return value. Therefore, we must have run `function2`.

Why we asked this question: This question was designed to see whether you had a good intuitive feel for how the implementation of the abstractions we'd used this quarter influences the way that those abstractions behave.

The `Stack`, as you saw in lecture, is implemented with a dynamic array, but it could also have been implemented equally efficiently with a linked list. The `Queue` type could either be implemented using a linked list with a tail pointer or as a dynamic array, both of which give rise to the same runtime. The `Set` is implemented as a balanced BST (the `HashSet`, on the other hand, uses a hash table). Finally, the `Vector` is implemented using a dynamic array.

Part (i) of this question was there to see whether you use your knowledge of these implementation details to infer the runtime of various operations. Part (ii) of this question was designed to let you show us what you'd learned about both the interface (`Stack`, `Queue`) and implementation (`Set`) of the various data types. Finally, we included part (iii) so you could demonstrate your facility looking at a runtime plot and inferring, quantitatively, what the growth rate was.

Common mistakes: The most common mistake we saw on this problem was mixing up the implementation of the `Set` type. Many answers were completely consistent with the idea that the `Set` was implemented with a hash table rather than a binary search tree, which is a reasonable but incorrect assumption to make. Going forward, it is actually probably a good idea to brush up on how different set abstractions are implemented, since the answer varies from language to language and library to library.

Another common mistake was arguing that the the vector-backed `function4` ran in time $O(n)$ rather than $O(n^2)$. We suspect that this was either due to (1) assuming the `Vector` type is backed by a linked list rather than a dynamic array or (2) forgetting that elements needed to be shifted back after an element is removed from the front of a `Vector`.

A good number of people got tripped up by the chart of runtimes because they didn't exactly scale linearly. For example, some answers argued that the growth rate was $O(n \log n)$ because sometimes the growth was slightly bigger than linear, and some answers argued that the growth rate was $O(\log n)$ because sometimes the growth was slightly sublinear.

As you saw in lecture, there are a number of factors that contribute noise to a runtime analysis – other programs running on the computer at the same time, temperature or power fluctuations, etc. – that cause actual runtimes to deviate slightly from their predicted theoretical amount. As a result, to really size up the runtime of a function, it's helpful to look at all the data points and to try to figure out the best answer that accounts for all of them.

Problem Two: Recursive Problem-Solving I

There are many ways to solve the first part of this problem. Here are three options.

```
/* To find the winner of a tournament, split the tournament in half. Find the
 * winner in the first and second halves, then have them play a game against
 * one another.
 */
string overallWinnerOf(const Vector<string>& initialOrder) {
    /* Base Case: If there's one player left, that player wins! */
    if (initialOrder.size() == 1) return initialOrder[0];

    int half = initialOrder.size() / 2;
    return winnerOf(overallWinnerOf(initialOrder.subList(0, half)),
                    overallWinnerOf(initialOrder.subList(half, half)));
}

                                    /* … or … */

/* Pair off the players so that each plays a game against the next player in the
 * ordering, then form an elimination tournament from those players in the same
 * relative order and see who wins!
 */
string overallWinnerOf(const Vector<string>& initialOrder) {
    /* Base Case: If there's one player left, that player wins! */
    if (initialOrder.size() == 1) return initialOrder[0];

    Vector<string> nextRound;
    for (int i = 0; i < initialOrder.size(); i += 2) {
        nextRound += winnerOf(initialOrder[i], initialOrder[i + 1]);
    }

    return overallWinnerOf(nextRound);
}

                                    /* … or … */

/* Treat the Vector like a Queue! Pull off the first two players, have them play
 * a game against one another, then put the winner on the back. This process will
 * pair off the players in the same order as required by the tournament bracket.
 */
string overallWinnerOf(const Vector<string>& initialOrder) {
    /* Base Case: If there's one player left, that player wins! */
    if (initialOrder.size() == 1) return initialOrder[0];

    /* Drop off the first two players. */
    auto nextRound = initialOrder.sublist(2, initialOrder.size() - 2);

    /* Put the winner on the back. */
    nextRound += winnerOf(initialOrder[0], initialOrder[1]);
    return overallWinnerOf(nextRound);
}
```

The second part of this problem is essentially a permutations problem: we just list off all possible ways to order the players and see if our favorite player ever wins!

```
bool canRigFor(const string& player, const HashSet<string>& allPlayers,
               Vector<string>& initialOrder) {
    return canRigRec(player, allPlayers, initialOrder, {});
}

bool canRigRec(const string& player, const HashSet<string>& allPlayers,
               Vector<string>& initialOrder, const Vector<string>& soFar) {
    /* Base Case: If everyone is already placed, see if our player wins! */
    if (allPlayers.isEmpty()) {
        if (overallWinnerOf(soFar) == player) {
            initialOrder = soFar;
            return true;
        }
        return false;
    }
    /* Recursive case: Try all possible next players. */
    for (string nextPlayer: allPlayers) {
        auto nextOrder = soFar;
        nextOrder += nextPlayer;
        if (canRigRec(player, allPlayers - nextPlayer, initialOrder, nextOrder)) {
            return true;
        }
    }
    /* Oh well, guess it's not possible. */
    return false;
}
```

Why we asked this question: We chose this question primarily to make sure that you were comfortable with recursive backtracking and recursive problem-solving strategies. The first part of this problem was designed to assess whether you were comfortable looking at a tree structure and deducing some sort of recursive pattern from it, and we hoped that the fact that there are several different solution routes you can choose from would make that part a good warm-up. The second part of this problem is essentially just a permutations problem, and we hoped that you'd be comfortable looking over the structure of the problem and recognizing this particular detail.

Fun fact: this problem is based on some research done by one of my former CS103 TAs, Michael Kim, who is currently a Ph.D student here.

Common mistakes: Most people got the first part of this problem right, or otherwise had a solution that was almost entirely correct. Nice job! The most common mistakes we saw were choosing the wrong base case (for example, assuming there were always at least two players) or indexing errors in the `Vector`.

For the second part of the problem, by far the most common mistake we saw was not recognizing that this problem is fundamentally a permutations problem. Solutions that didn't approach the problem this way typically were much more complicated and didn't correctly try all the necessary options.

Problem Three: Linear Structures

Here's one possible solution:

```
class MoveToFrontSet {
public:
    MoveToFrontSet();
    ~MoveToFrontSet();
    bool contains(const string& str);
    void add(const string& str);
    void delete(const string& str);
private:
    struct Cell {
        string value;
        Cell* next;
    };
    Cell* head;
}

/* Constructor makes the head null to signify that no elements are present. */
MoveToFrontSet::MoveToFrontSet() {
    head = nullptr;
}

/* Destructor is our typical "deallocate a linked list" destructor. */
MoveToFrontSet::~MoveToFrontSet() {
    while (head != nullptr) {
        Cell* next = head->next;
        delete head;
        head = next;
    }
}

bool MoveToFrontSet::contains(const string& str) {
    Cell* prev = nullptr;
    Cell* curr = head;

    /* Scan the list, keeping track of the current pointer and previous pointer,
     * until we find what we want or fall off the list.
     */
    while (curr != nullptr && curr->value != str) {
        prev = curr;
        curr = curr->next;
    }

    /* If we didn't find it, curr will be null since we walked off the list. */
    if (curr == nullptr) return false;

    /* If we found it and it's not at the head of the list, move that element to
     * the front of the list.
     */
    if (curr != head) {
        prev->next = curr->next;
        curr->next = head;
        head = curr;
    }
    return true;
}
```

```
void MoveToFrontSet::add(const string& str) {
    /* If this element already exists, we're supposed to move it to the front.
     * That's automagically handled for us by the contains call!
     */
    if (contains(str)) return;

    /* Put a new cell at the front of the list. */
    Cell* cell = new Cell;
    cell->value = str;
    cell->next = head;
    head = cell;
}

void MoveToFrontSet::remove(const string& str) {
    /* See if the element is here. If not, there's nothing to do. */
    if (!contains(str)) return;

    /* Nifty fact: the element to remove is now at the front of the list, since
     * looking for it put it there! So just take it off the front.
     */
    Cell* toRemove = head;
    head = head->next;
    delete toRemove;
}
```

Why we asked this question: We included this question for a number of reasons. First, we wanted to give you a chance to demonstrate what you'd learned about class design and working with linked lists. We figured this particular problem worked well because it involved linked list manipulations (along the lines of what you did in the Splicing and Dicing assignment) and the idea of having different member functions call one another. Second, we thought this particular linked list exercise of splicing out a node from a singly-linked list and moving it to another location would allow you to demonstrate whether you were comfortable with the idea of maintaining two pointers into a linked list (something you likely needed in the course of building a linked list from scratch) and of rewiring cell pointers. Finally, we thought this problem was interesting in of itself. This is an example of a *self-adjusting data structure*, and this particular structure is often used in data compression (look up *move-to-front encoding*). It's also related to the more popular *splay tree*, an extremely fast and simple binary search tree data structure.

Common mistakes: We saw a number of solutions that contained memory errors, such as allocating cells unnecessarily (often, pointers were initialized to `new Cell` rather than `nullptr`) or reading from a cell after deleting it.

Many solutions attempted to implement contains in terms of insertion and deletion rather than the other way around. While in principle this works, it's not at all efficient (it's much faster to reorder existing linked list cells than it is to produce new cells from scratch) and makes the logic a lot trickier and therefore more error-prone.

Problem Four: Binary Search Trees

There are many ways of solving this problem, and we've included four of them in this solutions set!

These first two solutions are based on the idea that finding the bounds in a tree is very, very similar to running a binary search in an array. In a binary search in an array, we have two pointers representing the range where the element can be. At each point in time, we probe the midpoint of the range and decide how to adjust the bounds based on how the comparison goes.

We can adapt that same idea here. The difference is that instead of storing *indices* of the bounds, we'll store *pointers to nodes* representing those bounds. Instead of choosing the *midpoint* of the range, we'll pick whatever the root of the tree happens to be.

There are several ways to code this approach up. This first one is recursive, and the second iterative:

```
/* Returns the upper and lower bounds of the node, given that we know that all
 * elements in the tree are bounded from below and above by the specified nodes.
 */
Bounds boundsRec(Node* root, int key, Node* upper, Node* lower) {
    /* Base case: If we're out of nodes, whatever bounds we've discovered are
     * correct.
     */
    if (root == nullptr) return { upper, lower };

    /* Base case: If we have an exact match for the value, the current node is
     * both the upper and lower bound.
     */
    if (key == root->value) return { root, root };

    /* Recursive case: If the value is too small, then the root node is going to
     * be the lower bound unless we come across something even smaller.
     */
    else if (key < root->value) {
        return boundsRec(root->left, key, upper, root);
    }
    /* Recursive case: If the value is too big, then the root node is going to be
     * the upper bound unless we come across something even bigger.
     */
    else /* key > root->value */ {
        return boundsRec(root->right, key, root, lower);
    }
}

Bounds boundsOf(Node* root, int key) {
    return boundsRec(root, key, nullptr, nullptr);
}
```

Here's the iterative version. Note the similarity to binary search.

```
Bounds boundsOf(Node* root, int key) {
    /* Bounds found so far. */
    Node* lhs = nullptr;
    Node* rhs = nullptr;

    while (root != nullptr) {
        /* If we match exactly, great! We're done. */
        if (key == root->value) return { root, root };

        /* Otherwise, we have to go left or right. Adjust the lhs
         * and rhs accordingly.
         */
        else if (key < root->value) {
            rhs = root;
            root = root->left;
        }
        else /* key > root->value */ {
            lhs = root;
            root = root->right;
        }
    }

    return { lhs, rhs };
}
```

This next solution is based on a different insight that follows from the recursive definition of a binary search tree. First, if the tree is empty, then the bounds of the key we're looking for are both null, since there's nothing bigger or smaller than the key.

Otherwise, the tree consists of a node with two subtrees. Think about how the key relates to the parts of this schematic. If the key matches the root, then the root is both the upper and lower bound of the key. Otherwise, the key isn't a match. Let's suppose, just for expository purposes, that the key is less than the root. That tells us several things:

1. The root can't be the upper bound of the key. Why? Because the root is too big.
2. The upper bound of the key, wherever it is, has to be in the left subtree. Why? Because the upper bound can't be the root node (it's too big), nor can in be in the right subtree (because those values are all bigger than the root, which is already too big.)
3. The root *might* be the lower bound of the key, since it's bigger than the key, but only if nothing in the left subtree is both bigger than the key and less than the root.

Based on these insights, we can build a different recursive algorithm that works by descending into the appropriate subtree and, optionally, patching up one of the upper or lower bounds.

```
Bounds boundsOf(Node* root, int key) {
   /* Looking in an empty tree? Looking at something that's an exact
    * match? Then you have your answer.
    */
   if (root == nullptr || key == root->value) return { root, root };

   /* If the key should be in the left subtree, get the bounds purely for
    * that subtree, then see whether we should act as the lower bound.
    */
   if (key < root->value) {
       auto result = boundsOf(root->left, key);
       if (result.lowerBound == nullptr) result.lowerBound = root;
       return result;
   }
   /* Otherwise, it's in the right subtree. Use similar logic to the above. */
   else /* key > root->value */ {
       auto result = boundsOf(root->right, key);
       if (result.upperBound == nullptr) result.upperBound = root;
       return result;
   }
}
```

You could also implement the above idea as two separate helper functions, as shown here:

```
Node* upperBoundOf(Node* root, int key) {
    /* Got an empty tree? Have an exact match? We're done. */
    if (root == nullptr || key == root->value) return root;

    /* Otherwise, if the key is less than the root, the bound is
     * purely in the left subtree because we're too big to be a bound.
     */
    if (key < root->value) {
        return upperBoundOf(root->left, key);
    }

    /* Otherwise, we're bigger than the root. The root node may then
     * be the upper bound if one isn't found in the subtree.
     */
    else /* key > root->value */ {
       Node* result = upperBoundOf(root->right, key);
       return result == nullptr? root : result;
    }
}

Node* lowerBoundOf(Node* root, int key) {
    /* Got an empty tree? Have an exact match? We're done. */
    if (root == nullptr || key == root->value) return root;

    /* Otherwise, if the key is greater than the root, the bound is
     * purely in the right subtree because we're too small to be a bound.
     */
    if (key > root->value) {
        return lowerBoundOf(root->right, key);
    }

    /* Otherwise, we're smaller than the root. The root node may then
     * be the lower bound if one isn't found in the subtree.
     */
    else /* key < root->value */ {
        Node* result = lowerBoundOf(root->left, key);
        return result == nullptr? root : result;
    }
}

Bounds boundsOf(Node* root, int key) {
    return {
        upperBoundOf(root, key),
        lowerBoundOf(root, key)
    };
}
```

Why we asked this question: We included this question for a few different reasons. First, we wanted to let you show us what you'd learned about binary search trees and their structural properties. Could you navigate down a BST based on how a particular key relates to the root value? Could you propagate information up through a series of recursive calls?

Second, we wanted to give you a chance to work through an algorithmic question involving binary search trees. You weren't expected to immediately see how to compute the bounds of a particular key, but we hoped that your intuition about BSTs would help you determine what questions would be best to ask to arrive at a solution.

Common mistakes: There were two general classes of mistakes on this problem. First, there were regular, run-of-the-mill coding errors. Second, there were more problem-specific algorithmic concerns.

Let's begin with the coding errors. Perhaps the most common error we saw on this problem was trying to treat an object of type Bounds as a pointer. For example, we saw many solutions that included a line like this one:

```
⚠     Bounds result = new Bounds;     ⚠
```

Here, the variable `result` is an honest-to-goodness `Bounds` object, not a pointer to one, so it doesn't need to be (and in fact, can't legally be) initialized using the `new` keyword. Remember, the `new` keyword produces a pointer, so this statement tries to assign a pointer (`new Bounds`) to a non-pointer (`result`). We suspect that people got tripped up here because `Bounds` is a `struct` that contains pointers, but which itself is not actually a linked structure.

Second, we saw a number of solutions that forgot to handle the case where the tree was empty. Some solutions legitimately forgot to account for this case, while others attempted to handle it but did so incorrectly. For example, we saw many submissions that included code fragments like these:

```
⚠       if (key == root->value) return {root, root};          ⚠
⚠       if (root == nullptr) return {nullptr, nullptr};       ⚠
```

We call this the "shoot first and ask questions later approach," since you're following the root pointer (shooting) before you determine whether it's not null (asking questions). This code will crash in the case that `root` is null, since the the first line tries to dereference the pointer.

A more minor error we saw was mixing up `Node*` pointers, which represent pointers to nodes in the tree, with the integer values they contain. Sometimes we'd see people assigning numeric values to pointers, and (more frequently) we'd see solutions that compared integer keys directly against pointers.

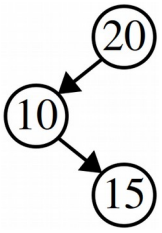
We also saw some common recursive errors, such as making recursive calls and forgetting to return the value of the recursive call.

Let's now turn to algorithmic errors. The most common algorithmic mistake we saw on this problem arose when people tried to look at the root and one of its children to try to size up where one of the bounds must be. For example, we saw many solutions along these lines:

```
⚠       if (key < root->value) {                                  ⚠
⚠           if (root->left == nullptr || root->left->value < key) {   ⚠
⚠               // lower bound is root                             ⚠
⚠           } else {                                              ⚠
⚠               // recursively explore left subtree               ⚠
⚠           }                                                     ⚠
⚠       }                                                         ⚠
```

The idea here is the following. Suppose we know that the key is less than the root's value. What node, then, is the lower bound of the key? The root node's value is bigger than the key, so it's a candidate for being the lower bound. So how do we tell if there's a better one? Well, if there's no left child, then there's nothing smaller than the root, so the root is the lower bound, and if there *is* a left child and the key's value is sandwiched between the left child and the root, then the root should be the lower bound.

Everything in the above paragraph is correct, *except* for the last part of the last sentence. For example, consider the BST shown to the right. Now, what is the lower bound of 15? The correct answer should be the node containing 15, but if we use the above code, we'll incorrectly report that the root is the lower bound because the key (15) is sandwiched between the left child's value and the root's value. That is, we have to look deeper in the tree to find the true lower bound. On the other hand, what's the lower bound of 16? In that case, the correct answer actually is the root, even though, as above, the key is sandwiched between the root's value and the left subtree's value. However, we can only tell that because we can look deeper in the tree to see what other nodes are there.



The reason for mentioning both cases here is that finding a bound in a tree can be tricky and isn't something you can typically do by looking at a single pair of linked nodes. Just knowing a key is between two nodes in the tree doesn't mean that one of those nodes has to be the bound. You may need to look more extensively in the tree to find the bound.

There was one last class of error we encountered that belongs to the category of "the code isn't wrong, but it's much more complex than it needs to be." We saw many recursive implementations that, before descending into a left or right subtree, would check that the current node was a better lower / upper bound, respectively, than the best bound found so far. However, it's not necessary to do this. For example, if you find that the key is less than the root's value and move to the left, then the root node is *guaranteed* to be a better lower bound than whatever lower bound has been discovered so far. (Do you see why?) Similarly, if you find that the key is greater than the root's value and move to the right, then the root node is *guaranteed* to be a better upper bound than whatever's been discovered so far. (Do you see why?)

Problem Five: Recursive Problem-Solving II

An interesting aspect of this problem is that we aren't required to actually return the playlist. We just need to see whether one exists. And if that's the case, we don't actually care about the order in which the songs appear in the playlist, just how many times each song appears. (Do you see why?)

This first recursive solution works by going one song at a time, asking how many times we'll use it.

```
/* Can we add up to exactly the workout length using only songs from index
 * start and forward?
 */
bool canMakeRec(const Vector<int>& songLengths, int start,
                int workoutLength, int maxTimes) {
    /* Base case: If the length is zero, yes, we can make it! Just have
     * a playlist with no songs on it.
     */
    if (workoutLength == 0) return true;

    /* Base case: If we're out of songs, then no, we can't make it! */
    if (start == songLengths.size()) return false;

    /* Recursive case: We need to determine how many times to use this first
     * song. See what those options are.
     */
    for (int times = 0; times <= maxTimes; times++) {
        /* If this will take too much time, stop searching. We know that using
         * it any more times will only make things worse.
         */
        int duration = songLengths[start] * times;
        if (duration > workoutLength) break;

        /* Otherwise, see what happens if we include this song that many times. */
        if (canMakeRec(songLengths, start + 1,
                       workoutLength - duration, maxTimes)) {
            return true;
        }
    }

    /* If we're here, no options worked. */
    return false;
}

bool canMakePlaylist(const Vector<int>& songLengths,
                     int workoutLength, int maxTimes) {
    return canMakeRec(songLengths, 0, workoutLength, maxTimes);
}
```

This next solution is based on the idea that we'll build up the playlist one song at a time, repeatedly choosing a next song that doesn't exceed the time limit and making sure not to exceed our allotment. One of the challenges here is that multiple songs might have the same length, so we need to do some extra bookkeeping to track how many times each song has appeared on the playlist so far.

This approach is not as fast as the other one, so we ***would not*** award full credit. In particular, note that this will generate the same playlist multiple times, requiring a lot of extra work to compute the solution. Do you see why?

```
/* Can you make a playlist whose total length is exactly workoutLength using
 * each song at most maxTimes times, given that the playlist already contains
 * some number of copies of each song?
 *
 * The playlist is encoded as a list of the indices into the songLengths list.
 */
bool canMakeRec(const Vector<int>& songLengths,
                int workoutLength, int maxTimes,
                const Vector<int>& soFar) {
    /* Base case: If the length of our playlist so far happens to match the
     * length of workout, we're done.
     */
    int length = lengthOf(soFar, songLengths);
    if (length == workoutLength) return true;

    /* Base case: If the current playlist is too long, it can't possibly
     * work.
     */
    if (length > workoutLength) return false;

    /* Recursive case: Try each song that doesn't appear too many times. */
    for (int i = 0; i < songLengths.size(); i++) {
        /* Can we fit this in? Or have we used it too much? */
        int copies = copiesOf(soFar, i);
        if (copies < maxTimes) {
            auto nextList = soFar;
            nextList += i;

            if (canMakeRec(songLengths, workoutLength, maxTimes, nextList)) {
                return true;
            }
        }
    }

    /* Oh fiddlesticks. */
    return false;
}

/* Given a list of which songs to play in the playlist, returns how long
 * the playlist is.
 */
int lengthOf(const Vector<int>& songIndices,
             const Vector<int>& songLengths) {
    int result = 0;
    for (int index: songIndices) {
        result += songLengths[index];
    }
    return result;
}
                        /* … continued on the next page … */
```

```
/* Given a list of songs to play, returns how many times the given song
 * appears.
 */
int copiesOf(const Vector<int>& songIndices, int index) {
    int result = 0;
    for (int song: songIndices) {
        if (song == index) result++;
    }
    return result;
}

bool canMakePlaylist(const Vector<int>& songLengths,
                     int workoutLength, int maxTimes) {
    return canMakeRec(songLengths, workoutLength, maxTimes, {});
}
```

Why we asked this question: We included this question as a final wrap-up to our treatment of recursive exploration. We hoped that this problem, which is very much in the same spirit as the questions on the midterm, would be a great way for folks to show how much they'd learned since the start of the quarter.

Common mistakes: Although many solutions treated this as a permutations problem, that's not the best fit for this problem because the order of the songs doesn't matter. Specifically, the total length of a playlist purely depends on the total lengths of the songs on that playlist, not the order in which they appear.

We also saw a number of solutions that approached this problem in a way that did introduce unnecessary inefficiencies. For example, some solutions would never check the total length of the playlist they'd produced until the very end of the recursion, focusing much of the search effort on dead-ends that couldn't pan out. We did deduct points for solutions like these, as they could be significantly improved with very little code without impacting the overall recursive strategy.

Aside from these efficiency concerns, the most common error was not handling the case where there were multiple songs with the same length in the `songLengths` vector. For example, solutions that tracked frequencies by associating each length with its frequency wouldn't work correctly if, for example, there were two songs that were exactly three minutes long.

Besides these problem-specific errors, we saw a number of general recursive issues here. Some solutions contained unconditional return statements inside of a for loop, cutting off the search too early. Others modified data across recursive calls in ways that caused future recursive calls to have incorrect information passed down into them. Otherwise interchanged the order of base cases in a way that caused the code to not work as expected.