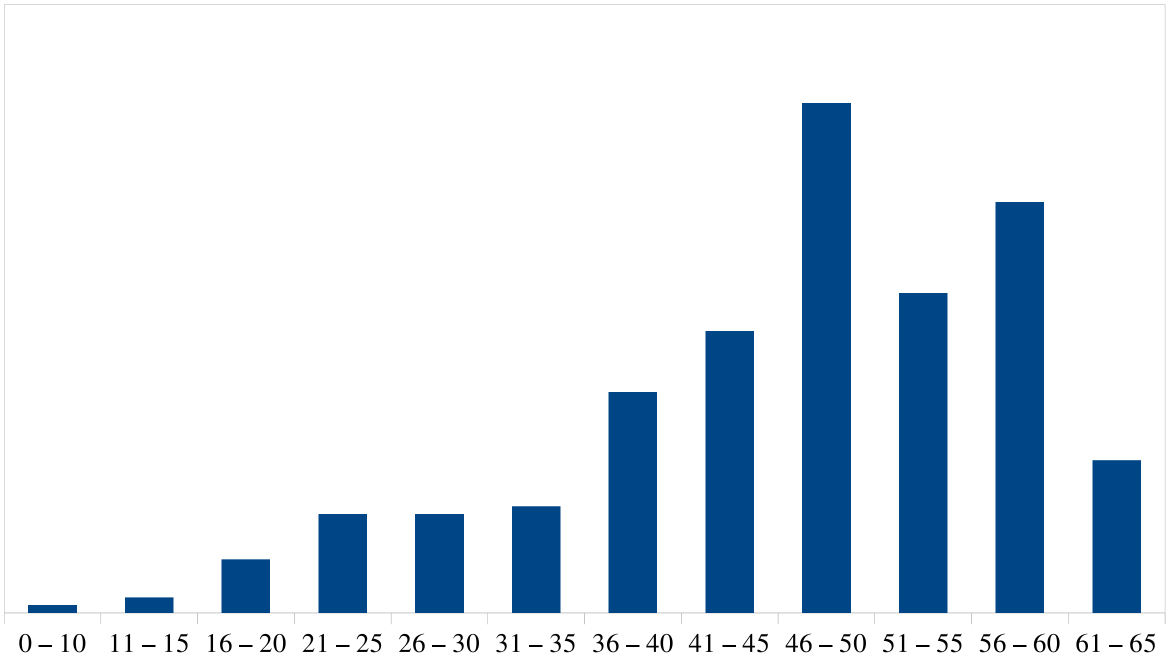
CS106B
Winter 2019

Handout 26S
March 25, 2019

CS106B Final Exam Solutions

Here's the overall histogram for the final exam:



For reference, here are the quintile markers.

80th Percentile: **56 / 65 (86%)**
60th Percentile: **50 / 65 (77%)**
40th Percentile: **46 / 65 (71%)**
20th Percentile: **37 / 65 (57%)**

If you've read over your exam and have concerns about how it was graded, please feel free to contact Keith over email. All exam regrades need to be submitted no later than the Friday of the first week of Spring quarter.

Problem One: Words of Encouragement

Hi Everybody!

You've come a long way since the start of the quarter. Over the past ten weeks, we've gone through a whirlwind tour of container classes, recursive problem-solving, recursive optimization, recursive backtracking, sorting algorithms, big-O notation, memory management, dynamic arrays, binary heaps, linked lists, binary search trees, tries, hashing, graph representations, and graph algorithms. That's a ton of material, so congratulations on making it this far! You've worked hard to get where you are now, and I wanted to thank you for putting in so much time and effort.

The questions on this exam are designed for us to see how much you've learned over the past ten weeks. Ten weeks ago you wouldn't have been able to understand most of these questions, let alone answer them. Treat this exam as an opportunity to show us what you've learned over the course of this quarter. Give it your best shot – ***you can do this!***

Good luck!

-Keith

For a free point, copy the text below.

I'm awesome, I've learned a ton, and I'm going to rock this exam.

I'm awesome, I've learned a ton, and I'm going to rock this exam.

Problem Two: Data Structures

Data Structure Sleuthing

```
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.

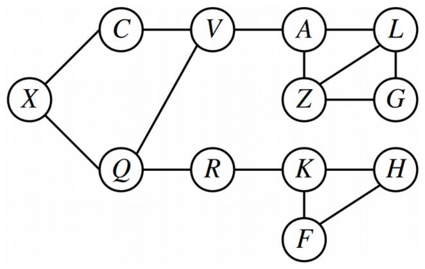
Another common mistake was arguing that the the vector-backed `function4` ran in time O(n) rather than O(n^2). We suspect that this was either due to (1) assuming the `Vector` type is backed by a linked list rather than a dynamic array or (2) forgetting that elements needed to be shifted back after an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

Graph Sleuthing



i. **(2 Points)** Explain why it is ***impossible*** to get back the following sequence of nodes when running BFS on the above graph, starting at node *X*.

X, *Q*, *C*, *R*, *V*, *K*, *Z*, *A*, *F*, *H*, *L*, *G*

There are many reasons why this sequence cannot be obtained. Here are a few:

"We should see the nodes in increasing order of distance. However, node *Z* is at distance 4 from the start node *X*, and it's reported between nodes *A* and *K*, which are at distance 3 from the start node *X*, which is impossible."

"Node *Z* appears before node *A* in the ordering, but there's no path that reaches node *A* without first going through node *V*."

"BFS cannot dequeue a node until one of its neighbors has first been dequeued. But *Z* appears before any of its neighbors appears, which breaks that rule."

ii. **(2 Points)** You have enough information to narrow down the mystery edge to one of two possibilities. What are those two possibilities? No justification is required.

The two options are {*V*, *Z*} and {*R*, *Z*}. The only way this BFS ordering is possible is if *Z* is at distance 3 from *X*. Since *Z* is currently at distance 4 from *X*, we know that *Z* has to be one of the endpoints of the edge. The other endpoint of the edge has to be something that's at distance 2 from *X*, which would correctly make node *Z* at distance 3. That limits our options to {*V*, *Z*} and {*R*, *Z*}.

X, *Q*, *R*, *K*, *F*, *H*, *Z*, *G*, *L*, *A*, *V*, *C*

iii. **(2 Points)** Explain why it is ***impossible*** to get the above sequence of nodes as the result of DFS in the graph drawn above. Please limit your answer to fifty words or fewer.

There are many ways to account for this. Here are a few:

"After we visit node *H*, we should backtrack to the last node with a neighbor we haven't yet seen. In the picture, that's node *Q*, so we should have seen *Q*'s neighbor *V* after seeing *H*. But instead, we see *Z*."

"This node ordering places *A* before *V*, but every path from *X* to *A* that passes through *V* first."

"Node *Z* can only appear in a DFS ordering if one of its neighbors appears before it in the ordering, and none of *Z*'s neighbors appear before it."

iv. **(2 Points)** You now have enough information to identify the mystery edge. Which edge is it, and why? Please limit your answer to fifty words or fewer.

The mystery edge is $\{R, Z\}$. Based on our answer to part (iii), we know that there has to be an edge from something in the bottom half up to Z, or otherwise we'd visit node C after node H. Combining that with part (ii), the only possibility is $\{R, Z\}$.

Why we asked this question: We included this question to let you demonstrate the intuition you'd built up in Assignment 7 for the mechanics of breadth-first search and depth-first search. We figured this question, which looked at the observable behavior of BFS and DFS rather than the implementation, would be a good way to see whether you'd internalized what BFS and DFS *do* rather than how BFS and DFS *work*. Plus, we just thought that this was a fun problem. 😃

Common mistakes: The most common mistake on this problem was reporting that the mystery edge was $\{H, Z\}$. This edge will fix the DFS ordering, but it won't fix the BFS ordering. Usually, the justification offered for this edge was that since Z appears after H in the DFS ordering, there has to be an edge directly from H to Z. However, that's not quite right. Since H precedes Z in the DFS ordering, it *might* be the case that the edge $\{H, Z\}$ exists in the graph, but it could also be the case that that edge isn't there and we found Z by backing up to some earlier point and moving forward from there.

Problem Four: Linked Lists

Random Bag Lists

There are several ways to code this up. Here's an initial version, which uses a doubly-linked list that doesn't have any dummy cells:

```
/****** RandomBag.h ******/
class RandomBag {
public:
    RandomBag();

    void add(int value);
    int  removeRandom();

private:
    struct Cell {
        int value;
        Cell* next;
        Cell* prev;
    };

    Cell* head;
    int numElems;
};

/****** RandomBag.cpp ******/
#include "RandomBag.h"
#include "random.h"
#include "error.h"

/* Constructor just sets things up so that everything is empty. */
RandomBag::RandomBag() {
    head = nullptr;
    numElems = 0;
}

/* Add prepends an element to the front of the list. */
void RandomBag::add(int value) {
    Cell* cell = new Cell;
    cell->value = value;
    cell->next = head;
    cell->prev = nullptr;

    /* Update the head to point back to us. */
    if (head != nullptr) head->prev = cell;

    /* Either way, we're at the front of the list. */
    head = cell;

    numElems++;
}


                        /* … continued on the next page … */
/* Remove chooses and removes a single element. */
int RandomBag::removeRandom() {
    if (numElems == 0) error("Such things are not to be found in the void.");

    /* Choose the element to remove. */
    int index = randomInteger(0, numElems - 1);

    /* Find that element. */
    Cell* toRemove = head;
    for (int i = 0; i < index; i++) {
        toRemove = toRemove->next;
    }

    /* Splice that cell out of the list. */
    if (toRemove->prev != nullptr) toRemove->prev->next = toRemove->next;
    if (toRemove->next != nullptr) toRemove->next->prev = toRemove->prev;

    /* We may be the head of the list. If so, go update that. */
    if (toRemove == head) head = toRemove->next;

    /* Update element count so we know how many items exist. */
    numElems--;

    /* Deallocate the memory for this cell. */
    int result = toRemove->value;
    delete toRemove;

    return result;
}
```

Here's an alternative that uses a dummy head and tail cell. This one also doesn't explicitly keep track of how many cells are in the list, requiring two passes to do a `removeRandom`.

```
/****** RandomBag.h ******/
class RandomBag {
public:
    RandomBag();

    void add(int value);
    int  removeRandom();

private:
    struct Cell {
        int value;
        Cell* next;
        Cell* prev;
    };

    Cell* head;
    Cell* tail;

    int numElems() const;
};

                        /* … continued on the next page … */
```

```
/****** RandomBag.cpp ******/
#include "RandomBag.h"
#include "random.h"
#include "error.h"

/* Create a dummy head and tail cell. */
RandomBag::RandomBag() {
    head = new Cell;
    tail = new Cell;

    /* They point to each other. */
    head->next = tail;
    tail->prev = head;

    /* They demarcate the bounds. */
    head->prev = tail->next = nullptr;
}

/* Add prepends an element to the front of the list. */
void RandomBag::add(int value) {
    Cell* cell = new Cell;
    cell->value = value;

    /* Splice this into the list. */
    cell->next = head->next;
    cell->next->prev = cell;

    cell->prev = head;
    cell->prev->next = cell;
}

/* Remove chooses and removes a single element. */
int RandomBag::removeRandom() {
    if (head->next == tail) error("Cannot create values ex nihilo.");

    /* Choose the element to remove. */
    int index = randomInteger(0, numElems() - 1);

    /* Find that element. */
    Cell* toRemove = head->next;
    for (int i = 0; i < index; i++) {
        toRemove = toRemove->next;
    }

    /* Splice that cell out of the list. */
    toRemove->prev->next = toRemove->next;
    toRemove->next->prev = toRemove->prev;

    /* Deallocate the memory for this cell. */
    int result = toRemove->value;
    delete toRemove;

    return result;
}
```

/* … continued on the next page … */

```
/* Counts the number of (non-dummy) cells in the list. */
int RandomBag::numElems() const {
    int result = 0;
    for (Cell* curr = head->next; curr != tail; curr = curr->next) {
        result++;
    }
    return result;
}
```

Why we asked this question: We included this question for a number of reasons. First, we wanted to let you show us what you'd learned about defining an implementing a class backed by a linked list, something you first saw how to do in Assignment 6. Do you need a head pointer or a tail pointer? Do you need dummy elements? These decisions are all worth pondering, and we hoped you'd think through them while working on this problem.

Second, we wanted to give you practice traversing linked lists (here, to find an element to remove), one of the major list operations. Third, we wanted to see how you'd handle boundary cases like removing from a list when there's only a single element or inserting into an otherwise empty list.

Finally, wanted to see how well you'd internalized pointer manipulation techniques and memory management. Given that `removeRandom` isn't supposed to leak memory, which cells, specifically, need to be reclaimed? How do you sequence that memory cleanup along with the other operations?

Common mistakes: Many of the issues we saw people run into on this problem stemmed from simple, run-of-the-mill pointer mistakes. Some solutions forgot to account for the case where the list was empty or had just a single element, accidentally reading or writing null pointers. Others didn't do all the proper linked list wiring (for example, forgetting to set one of the previous pointers of a linked list cell). Some solutions used dummy nodes but forgot not to count them when determining which node to select. Many answers forgot to deallocate the removed cell, or deallocated the cell and then tried reading its contents.

Other issues we encountered seemed to stem from a misunderstanding of what was being asked. Some solutions treated the `RandomBag::add` function's `value` parameter as a number of random elements to add to the random bag rather than as a specific value that was supposed to be stored. Other solutions didn't attempt to remove anything in `removeRandom`. A few solutions used a singly-linked list rather than a doubly-linked list (oops).

Some solutions came up with options that technically worked but which were highly inefficient. For example, some solutions would generate a random number representing a *value* to remove, then iterate over the list to see whether it was there. If this didn't work, they'd repeat this process. While this technically works, it's likely going to take a *long* time to remove an element from the random bag if, say, there's just a single element in the list.

Problem Five: Binary Search Trees (12 Points)

Lower and Upper Bounds ***(Recommended time: 35 minutes)***

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.

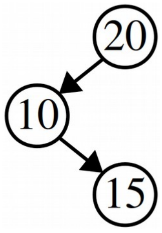
We also saw some common recursive errors, such as making recursive calls and forgetting to return the value of the recursive call.

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 Six: Tries

Trie Containment

There are a couple of ways you can solve this problem. This first solution works by walking both tries in parallel. The intuition is the following. For starters, assume that neither trie is completely empty, though we'll revisit that assumption in a bit. Zooming in and taking the view that a trie consists of a single node, which might be a word, that has some number of subtries, we can look at the roots of the two tries. If the first trie's root is a word, then the second trie's root has to be a word as well – otherwise, the first trie contains the empty string and the second doesn't. If there aren't any problems there, then we need to check each of the subtries of the first node against the subtries of the second, confirming that, indeed, each of those tries is a subtrie of the corresponding subtree.

Here's a recursive implementation based on that observation:

```
bool isSubsetOf(Node* one, Node* two) {
    /* Base case: The empty trie has no words, so everything
     * it contains is also in the second trie.
     */
    if (one == nullptr) return true;

    /* Base case: If the second trie is empty, then we are
     * not a subset of it.
     */
    if (two == nullptr) return false;

    /* If we contain the empty string and the other trie
     * doesn't, we are not a subset.
     */
    if (one->isWord && !two->isWord) return false;

    /* Otherwise, we're a subset if all our children are
     * subsets of the corresponding children.
     */
    for (char ch: one->children) {
        if (!two->children.containsKey(ch) ||
            !isSubsetOf(one->children[ch], two->children[ch])) {
            return false;
        }
    }

    /* Everything matches! We're good. */
    return true;
}
```

This second approach uses a different strategy. As you saw in Assignment 6, it's possible to walk a trie using a recursive enumeration approach to find all the words it contains. So let's walk the first trie, finding each word it contains. Each time we find a word, let's check whether it's contained in the second trie. If so, great! All is right and well so far. If not, horror of horrors! We've found a word in the first trie that isn't in the second.

Here's what that might look like:

```
/* Does this word appear within the given trie? */
bool contains(Node* root, string text) {
    /* If the trie is empty, this text doesn't appear anywhere. */
    if (root == nullptr) return false;

    /* If the string is empty, check if the root node - which represents
     * the empty string - is a word.
     */
    if (text == "") return root->isWord;

    /* Otherwise, descend deeper and check for the rest of the word. */
    return root->children.containsKey(text[0]) &&
           contains(root->children[text[0]], text.substr(1));
}

/* Are all the words in one also in two, given that every word in one
 * starts with the given prefix?
 */
bool isSubsetRec(Node* one, Node* two, string soFar) {
    /* Base case: If there are no words in one, we're done! */
    if (one == nullptr) return true;

    /* Base case: If we're a word and that word isn't in two, then
     * we aren't a subset.
     */
    if (one->isWord && !contains(two, soFar)) return false;

    /* Recursive case: Check if all words in deeper tries are contained
     * in two.
     */
    for (char ch: one->children) {
        if (!isSubsetRec(one->children[ch], two, soFar + ch)) {
            return false;
        }
    }

    /* Guess it all checks out! */
    return true;
}

bool isSubsetOf(Node* one, Node* two) {
    return isSubsetRec(one, two, "");
}
```

Why we asked this question: We included this question for three main reasons. First, we wanted to see whether you were comfortable recursively exploring the entirety of a linked structure (here, the first trie) to ascertain some property. Second, we wanted to see whether you could do a more targeted search of another linked structure (here, the second trie) to see whether it satisfied certain constraints. Finally, we included this problem because it has an interesting structure: it's essentially an "inverse backtracking" problem! Rather than returning *true* if any recursive subcall returns true, we return *false* if any recursive subcall returns false. Similarly, rather than returning *false* if we try all options without returning, here we return *true*.

Common mistakes: By far the most common mistake on this problem was forgetting to handle the case where the input pointer was null. Generally speaking, empty trees of any sort are represented using null pointers unless specified otherwise, and here is no exception. Although the trie children pointers never hold a null pointer, we might still need to consider the case where there aren't any nodes in the trie at all.

In a related note, another common error was to forget to check the root node itself to see whether it was a word. Many solutions worked by iterating over the child nodes, checking if those child nodes were words. While technically speaking this approach only mishandles the root node, it likely indicates a deeper logic error when thinking about linked structures. Remember – when working with linked structures, it's best to focus on the root of the structure rather than the children, since the root has no parent and won't get checked otherwise.

Aside from these errors, we saw a number of stray coding errors, such as forgetting to process the return value generated by a recursive call or unconditionally returning inside the loop over children.

Problem Seven: Recursive Problem-Solving

Workout Playlists

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, which is a very clean and efficient approach to solving the problem, works by simply going one song at a time, asking how many times we'll use that song.

```
/* 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, but we accepted it for full credit.

```
/* 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?
 *
 * Here, 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 we awarded full credit to solutions that treated this problem as a permutations problem, this really isn't a permutations problem because the ordering of the songs is irrelevant. 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 decided to give full credit here because we thought that this particular observation is a bit subtle and that jumping from "playlist" to "permutation" isn't generally a bad one to make, but it's worth making sure you understand why the order is irrelevant here.

We did, however, see 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.