Section Solutions 8

Problem One: Bits, Bytes, Characters, and Strings

The literal `0` represents the numeric value 0. The literal `'0'` is a character that represents the symbol for the digit 0, but its numeric value is actually 48. The literal `"0"` is a string with one character, which happens to be the character `'0'`.

The statement `0 == '0'` will compile, and it returns false because the left-hand side has numeric value 0 and the right-hand side has numeric value 48. The statement `'0' == "0"` does not compile, because the left-hand side is a character and the right-hand side is a string and C++ does not allow for direct comparisons between strings and characters.

Problem Two: Huffman Encoding

Our first step is to get a frequency table for "avicenna's canon", which looks like this:

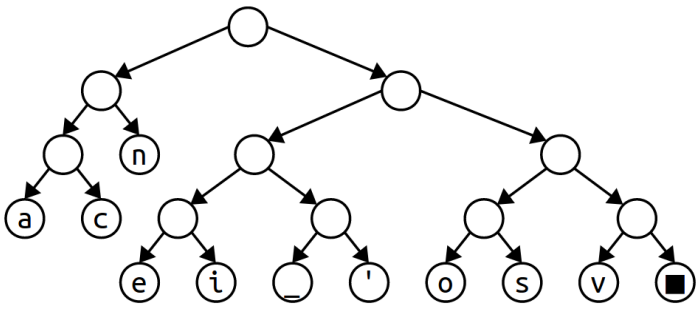
space	'	a	c	e	i	n	o	s	v	■
1	1	3	2	1	1	4	1	1	1	1

We now run the Huffman encoding algorithm to generate the encoding tree. Since there are many different letters here that are tied for the same frequency, there are many different possible Huffman trees that you could form from these letters. We've generated one of these trees by performing the following sequence of merges, tiebreaking somewhat arbitrarily. If you ended up with a different tree, that doesn't mean your answer was wrong; you may have just broken ties differently than us.

Here's what we did:

- Merge `v` and ■ into a tree of weight 2.
- Merge `o` and `s` into a tree of weight 2.
- Merge `e` and `i` into a tree of weight 2.
- Merge space and `'` into a tree of weight 2.
- Merge the tree containing `o` and `s` and the tree containing ■ and `v` into a tree of weight 4.
- Merge the tree containing space and `'` and the tree containing `e` and `i` into a tree of weight 4.
- Merge `a` and `c` into a tree of weight 5.
- Merge the tree of `e`, `i`, space, and `'` and the tree of `o`, `s`, `v`, and ■ into a tree of weight 8.
- Merge the tree holding `a` and `c` with `n` into a tree of weight 9.
- Merge the two remaining trees into the final tree.

The result is shown here:



Next, we need to figure out what the character encodings are using this tree. By tracing out the paths through the tree we find these encodings:

space	'	a	c	e	i	n	o	s	v	■
1010	1011	000	001	1000	1001	01	1100	1101	1110	1111

Finally, we have to write out the encoded bits for the message. That's done by replacing each character with its encoding, as shown here:

a	v	i	c	e	n	n	a	‘	s		c	a	n	o	n	■
000	1110	1001	001	1000	01	01	000	1011	1101	1010	001	000	01	1100	01	1111

So the final message is

```
00011101001001100001010001011110110100010000111000111111
```

Problem Three: Scrambled Hashes

Let's see what's wrong with these hash functions.

- Hash function 1: Always return 0.

This hash function is deterministic – the same inputs always produce the same outputs – but it does not produce a good distribution. Since everything has the same hash code, if we used this to produce a distribution over the buckets, we'd dump everything into bucket 0, producing a hash table that's essentially an unsorted array.

- Hash function 2: Return a random `int` value.

This hash function is not deterministic. If we hash the same value multiple times, we'll get different results. Imagine what would happen if you used this in a hash table – the bucket you place each item into could be totally different from the buckets where you look for it, so it will appear to not be present anywhere!

- Hash function 3: Return the sum of the ASCII values of the letters in the word.

This hash functions is deterministic, but it doesn't give a good spread. Any two words that are anagrams will have the same hash code, and closely-spelled words will likely hash to nearby locations. Additionally, this hash function does not make good use of the space of possible integers. ASCII values are all one byte each, so larger hash values (say, 78,979,871) will never be used. In a hash table with a large number of buckets, most buckets will be empty.

This just goes to show you that ***you should not try to come up with your own hash codes!*** Instead, use one that someone else has written, or follow a general template.

Problem Four: Rehashing

Here's one way to implement this:

```
void OurHashSet::add(const string& value) {
    /* Check for whether we've gotten large enough that we need to rehash. */
    if (size() / numBuckets >= 2) rehash();

    /* … the rest of this function is modified. … */
}

void OurHashSet::rehash() {
    /* Get a set of buckets that's twice as big as before. */
    Vector<Vector<string>> newBuckets(buckets.size() * 2, {});

    /* Copy all the old elements over. */
    for (Vector<string> bucket: buckets) {
        for (string elem: bucket) {
            /* We need to determine where this element goes in the new set of
             * buckets, because while its hash code hasn't changed, its hash code
             * modulo the number of buckets very well may have.
             */
            int index = hashCode(elem) % newBuckets.size();
            newBuckets[index] += elem;
        }
    }

    buckets = newBuckets;
}
```

Problem Five: Custom Hash Codes

Here's one way to do this:

```
int hashCode(const City& city) {
    int result = hashCode(city.name);
    result = 31 * result + hashCode(city.population);
    return 0x7FFFFF & result;
}

bool operator== (const City& lhs, const City& rhs) {
    return lhs.name == rhs.name && lhs.population == rhs.population;
}
```

As a note – if you decide to continue onward in pure C++, the way to specify a hash code without the Stanford libraries is significantly weirder and more complex. Here's what that same hash function would look like in standard C++. Oof. No wonder we took care of some of the details for you. ☺

```
namespace std {
    template <> struct hash<City> {
        size_t operator()(const City& city) const {
            size_t result = hash<string>()(city.name);
            result = 31 * result + hash<int>()(city.population);
            return result;
        }
    };
}
```

Problem Six: A Classic Job Interview Question

Here's one way to think about this. Imagine you scan the array from left to right. Suppose you're looking at a number x. If the number $137 - x$ is also in the array somewhere, then there's a pair of elements in the array that sums up to exactly 137, namely x and $137 - x$. So the question is: how can you efficiently determine whether that number appears somewhere else in the array? One option would be to take everything in the array and throw it into a hash set – that way, you can efficiently determine whether the number you care about is in the array or not. In fact, this is exceptionally efficient: it takes, on average, only O(1) time to insert something into a hash table or check whether it's there! Here's some code for this:

```
bool hasMagicPair(const Vector<int>& values) {
    /* Dump everything in a hash table. Total time: O(n), on average. */
    HashSet<int> elems;
    for (int elem: values) {
        elems += elem;
    }

    /* Look at each array element and see whether there's something else in the
     * array that would complete the sum.
     *
     * This does, on average, O(1) work per element, so it takes average time
     * O(n).
     */
    for (int elem: values) {
        if (elems.contains(137 - elem)) return true;
    }
    return false;
}
```

You can actually combine the two loops together to get this solution that also runs in expected time O(n):

```
bool hasMagicPair(const Vector<int>& values) {
    HashSet<int> elems;
    for (int elem: values) {
        if (elems.contains(137 - elem)) return true;
        elems += elem;
    }
    return false;
}
```

This really is a common interview question. Hopefully you can see why hash tables are so nice!

So what about this generalized version of the problem where you want to see if *three* numbers add up to 137? Well, you could just list all triples of numbers, but that takes time O(n^3), and it doesn't involve hash tables. Here's one option you can use that's based on the previous idea. As before, start by dumping all the contents of the array into a hash table. Then, iterate over all possible pairs of numbers in the array. If you're looking at x and y and the number $137 - x - y$ is in the hash table, then, as before, you know a triple exists. Oh happy day! Huzzah! On the other hand, if no pair x and y has $137 - x - y$ in the hash table, then you know such a sum doesn't exist. Here's one way to code this up:

```
bool hasMagicTriple(const Vector<int>& values) {
    /* Dump everything in a hash table. Total time: O(n), on average. */
    HashSet<int> elems;
    for (int elem: values) {
        elems += elem;
    }

    /* Look at all pairs of elements to see if any of them give a sum that can be
     * completed to 137. Since we're allowed to include duplicate values, we'll
     * iterate over all possible pairs of elements we can make that allow for
     * duplicates.
     */
    for (int i = 0; i < values.size(); i++) {
        for (int j = i; j < values.size(); j++) {
            if (elems.contains(137 - i - j)) return true;
        }
    }
    return false;
}
```

This problem has attracted a lot of theoretical attention recently because while there are good algorithms that on average run in time $O(n^2)$, it's surprisingly difficult to improve upon those algorithms. The state of the art for this problem, based on a comically-titled paper from 2014, runs in the unusual expected runtime of $O(n^2 (\log \log n)^2 / \log n)$.

Here's a fun problem: show that, using hash tables, you can determine whether there are exactly k numbers that sum up to 137 in time $O(n^{\lceil k/2 \rceil})$, where $\lceil k/2 \rceil$ denotes what you get when you divide k by two and round up. The algorithm you'll come up with is sometimes called a ***meet-in-the-middle*** algorithm and is related to some cool concepts in cryptography. Take CS255 for details!

Problem Seven: Linear Probing

There are many ways to code this up. Here's our final version of the class definition:

```
class LinearProbingTable {
public:
    LinearProbingTable();

    bool contains(int value) const;
    void add(int value);

    int size() const;
    bool isEmpty() const;

private:
    Vector<int> table; // All the slots, with empty slots marked with sentinels.
    int numElems = 0;  // How many slots are filled.

    /* Inserts the specified value into the table using the linear probing
     * algorithm, returning whether the insertion succeeded.
     */
    bool insertInto(int value);
    void rehash();
};
```

Here's the implementation:

```
const int kEmpty = -1; // Sentinel value indicating an empty slot.
const int kDefaultSize = 8; // Number of slots in a new table. Totally arbitrary.

LinearProbingTable::LinearProbingTable() {
    /* Fill in our array with a bunch of empty slots. */
    for (int i = 0; i < kDefaultSize; i++) {
        table += kEmpty;
    }
}

bool LinearProbingTable::contains(int value) const {
    /* Negative values aren't allowed and are never present. */
    if (value < 0) return false;

    /* Jump to the index at which the element should be placed. */
    int index = hashCode(value) % buckets.size();

    /* Keep walking forward until we either find the element or find an empty
     * slot. We'll wrap around if necessary using the mod operator.
     */
    while (table[index] != value && table[index] != kEmpty) {
        index = (index + 1) % table.size();
    }

    /* At this point we're either at the element or we're at an empty slot. If the
     * element is here, great! We're done. If not, it must not be anywhere.
     */
    return table[index] == value;
}

/* These last two are always the easiest. :-) */
int LinearProbingTable::size() const {
    return numElems;
}

bool LinearProbingTable::isEmpty() const {
    return size() == 0;
}

void LinearProbingTable::add(int value) {
    /* We don't allow negative values in this table. */
    if (value < 0) error("Only nonnegative values allowed - sorry!");

    /* If we've exceeded our maximum load factor, we have to rehash. */
    if (double(numElems) / table.size() >= 0.75) rehash();

    /* Insert into the table. */
    if (insertInto(value)) numElems++;
}


                                /* … continued … */
```

```
/* Performs an insertion into a table. */
bool LinearProbingTable::insertInto(int value) {
    /* Jump to the intended spot for this element. */
    int index = hashCode(value) % table.size();

    /* Walk until we find a free slot or find that the element is already here. */
    while (table[index] != kEmpty && table[index] != value) {
        index = (index + 1) % table.size();
    }

    /* If we're looking at the element that we'd like to insert, the element is
     * already here and there's nothing to do.
     */
    if (table[index] == value) return false;

    /* Otherwise, write in the value. */
    table[index] = value;
    return true;
}

/* To do a rehash, we copy out all the elements that are already present, make
 * the table bigger, then insert all the values again.
 */
void LinearProbingTable::rehash() {
    /* Step one: Gather all existing values and overwrite the table so that it's
     * fully empty.
     */
    Vector<int> existing;
    for (int i = 0; i < table.size(); i++) {
        if (table[i] != kEmpty) {
            existing += table[i];
            table[i] = kEmpty;
        }
    }

    /* Double the table size. */
    int newSize = table.size() * 2;
    while (table.size() < newSize) {
        table += kEmpty;
    }

    /* Reinsert everything. */
    for (int value: existing) {
        insertInto(value);
    }
}
```

Some of the other operations on a linear probing table are a bit more nuanced. For example, if you remove an element from the table, you can't just overwrite it with the sentinel value – do you see why? It's common to use an approach called ***tombstone deletion*** to address this. You choose some special sentinel value that means "something was here and has since been removed," and then adjust the insertion and lookup logic to handle it.

There's a variation on linear probing called ***Robin hood hashing*** that is extremely fast and lets the load factor in the table get much, much higher without a performance drop. Look it up if you're curious!

And as always, take CS166 if you want to see more of this!

Problem Eight: Word Ladders as a Graph Search

We can imagine a graph where each word in English is a node, and there's an edge between two nodes (words) if those words differ at exactly one letter. Notice that a path in this graph corresponds to a series of words where each one differs from the previous one in exactly one letter, since each edge you follow replaces one letter in the word. By the time you've gone from one word to another, you've found yourself a word ladder!

The task of finding the *shortest* word ladder then essentially becomes finding the shortest path in a graph, starting at the node for the initial word and ending at the node for the destination word. Look back at your code for the Word Ladders assignment and compare it against the breadth-first search code that we came up with in lecture. Do you see the similarities? If not, chat with your SL about it – it's a really great exercise!

Problem Nine: Graph Searches

Graph 1, DFS

The nodes are visited in this order:

A, B, E, D, G, H, F, C

We never visit node I, since there's no path from A to I.

Graph 1, BFS

Here's the contents of the queue at each point in time, along with the order that the nodes are visited.

Node Dequeued	*Actions*	*Queue Contents*
A	Enqueue B, D.	{B, D}
B	Enqueue E	{D, E}
D	Enqueue G	{E, G}
E	Enqueue F (D already visited)	{G, F}
G	Enqueue H	{F, H}
F	Enqueue C	{H, C}
H	*none* (E, G already visited)	{C}
C	*none* (B already visited)	{}

Not all nodes are visited. Notice that I isn't visited, since there's no path from A to I in the graph.

Graph 2, DFS

The nodes are visited in this order:

A, D, E, B, C, F

All nodes are visited!

Graph 2, BFS

Here's the contents of the queue at each point in time, along with the order that the nodes are visited.

Node Dequeued	*Actions*	*Queue Contents*
A	Enqueue D	{D}
D	Enqueue E (skipping A)	{E}
E	Enqueue B (skipping D)	{B}
B	Enqueue C, F (skipping E)	{C, F}
C	*none*	{F}
F	*none*	{}

Graph 3, DFS

The nodes are visited in this order:

A, B, D, E

Not all nodes are visited; in particular, node *C* isn't reachable from *A*.

Graph 3, BFS

Here's the contents of the queue at each point in time, along with the order that the nodes are visited.

Node Dequeued	*Actions*	*Queue Contents*
A	Enqueue B, D	{B, D}
B	*none* (A already visited)	{D}
D	Enqueue E	{E}
E	*none* (B, D already visited)	{}

Graph 4, DFS

The nodes are visited in this order:

A, B, C

Not all nodes are visited; in particular, nodes D and E are not reachable from A.

Graph 4, BFS

Here's the contents of the queue at each point in time, along with the order that the nodes are visited.

Node Dequeued	*Actions*	*Queue Contents*
A	Enqueue B, C	{B, C}
B	*none* (A visited, C already in queue)	{C}
C	*none* (A, B already visited)	{ }

Graph 4, Dijkstra's Algorithm

Here's the contents of the priority queue at each point in time, along with the order the nodes are dequeued and the distances reported. The first time each node is reported – which represents the shortest distance to the node – is shown in bold.

Node Dequeued	*Actions*	*Priority Queue Contents*
A (0)	Enqueue B (3), C (5)	{ B(3), C(5) }
B (3)	Enqueue C (5)	{ C(5), C(5)}
C (5)	Enqueue A(10), B(7)	{ C(5) }
C (5)	*none* (already visited)	{ }

Graph 5, DFS

The nodes are visited in this order:

A, B, C, D

Graph 5, BFS

Here's the contents of the queue at each point in time, along with the order that the nodes are visited.

Node Dequeued	*Actions*	*Queue Contents*
A	Enqueue B, C, D	{B, C, D}
B	*none* (A visited; C, D already enqueued)	{C, D}
C	*none* (A, B visited; D already enqueued)	{D}
D	*none* (A, B, C visited)	{ }

Graph 6, DFS

The nodes are visited in this order:

A, C, B, F, E, D, G

Graph 6, BFS

Here's the contents of the queue at each point in time, along with the order that the nodes are visited.

Node Dequeued	*Actions*	*Queue Contents*
A	Enqueue C, E	{C, E}
C	Enqueue B, D, G	{E, B, D, G}
E	Enqueue F (A visited)	{B, D, G, F}
B	*none* (A visited; C, F already enqueued)	{D, G, F}
D	*none*	{G, F}
G	*none* (D visited, F already enqueued)	{F}
F	*none* (B, E visited)	{ }

Graph 6, Dijkstra's Algorithm

Here's the contents of the priority queue at each point in time, along with the order the nodes are dequeued and the distances reported. The first time each node is reported – which represents the shortest distance to the node – is shown in bold.

Node Dequeued	*Actions*	*Priority Queue Contents*
A (0)	Enqueue C(8), E(1)	{ E(1), C(8) }
E (1)	Enqueue F(3)	{ F(3), C(8) }
F (3)	Enqueue B(5)	{ B(5), C(8) }
B (5)	Enqueue C(5)	{ C(5), C(8) }
C (5)	Enqueue G(10), D(12)	{ C(8), G(10), D(12) }
C (8)	*none* (already visited)	{ G(10), D(12) }
G (10)	Enqueue D(11)	{ D(11), D(12) }
D(11)	*none*	{ D(12) }
D(12)	*none* (already visited)	{ }

Problem Ten: Breadth-First Search and Binary Search Trees

This breadth-first search ends up being a bit nicer to code up than your typical breadth-first search because there's only one way to get to each node in the tree, meaning that we don't need to worry about visiting the same node multiple times. Here's some code based on that observation:

```
void breadthFirstSearch(Node* root) {
    /* Seed a queue with the root node. */
    Queue<Node*> worklist;
    worklist.enqueue(root);

    while (!worklist.isEmpty()) {
        /* Grab the next node. If we got back null, it means that we walked off
         * the tree and there's nothing to do here.
         */
        Node* curr = worklist.dequeue();
        if (curr == nullptr) continue;

        /* Process the node. */
        cout << curr->value << endl;
        worklist.enqueue(curr->left);
        worklist.enqueue(curr->right);
    }
}
```

With the tree provided in the handout, we'd see the nodes printed in this order:

f, b, j, a, d, h, k, c, e, g, i

Notice that this is what you get when you go through the tree one level at a time. This is sometimes called a ***level-by-level traversal*** of the tree.

So when will you see everything in sorted order? Well, let's begin with a key observation: ***if any node has a left child, the tree will not print in sorted order.*** Why is this? Well, imagine that there is some node x that has a left child l. Since one go through the tree one layer at a time, we'll first print x, then afterwards print l. But we know that l's value is less than x's value, meaning that l prints out of order.

This means that the only trees that will print out in sorted order are trees where no node has a left child. Those trees essentially are linked lists starting at the root node and continuing down and to the right.

Problem Eleven: Tournament Winners

To determine whether some player *p* is a winner, we need to look at all players that are exactly one and two hops away from *p* to see if we found everyone. There are many, many ways you can do this. Here are three: one based on directly looking at all pairs of players, one based on breadth-first search, and one based on depth-first search.

```
/* Version 1: Brute force. It works, but it ain't pretty. */
bool isWinner(const string& player, const Map<string, Set<string>>& tournament) {
    /* Loop over all other players to see if they're zero hops, one hop, or two
     * hops away.
     */
    for (string other: tournament) {
         if (other != player) { // Only care about other players
             if (!tournament[player].contains(other)) {
                 /* See if anyone that the player won against happened to beat
                  * this person.
                  */
                 bool success = false;
                 for (string vanquished: tournament[player]) {
                     if (tournament[vanquished].contains(other)) {
                         success = true;
                         break;
                     }
                 }
                 if (!success) return false;
             }
         }
     }
     return true;
}

Set<string> winnersOf(const Map<string, Set<string>>& tournament) {
    Set<string> result;
    for (string player: tournament) {
        if (isWinner(player, tournament)) result += player;
    }
    return result;
}
```

```
/* Version 2: BFS. Much nicer! This might not look like a BFS, but it essentially
 * is one. We find everyone one hop away, then everything two hops away.
 */
bool isWinner(const string& player, const Map<string, Set<string>>& tournament) {
    /* Find everyone who is one hop or two hops away from this player. */
    Set<string> oneHop = tournament[player];

    Set<string> twoHops;
    for (string vanquished: oneHop) {
        twoHops += tournament[vanquished];
    }

    /* See if everyone was covered. */
    return (oneHop + twoHops + player).size() == tournament.size();
}

Set<string> winnersOf(const Map<string, Set<string>>& tournament) {
    Set<string> result;
    for (string player: tournament) {
        if (isWinner(player, tournament)) result += player;
    }
    return result;
}
```

```
/* Version 3: DFS. Quite short! We just find everyone at most two steps away and
 * make sure we got everyone.
 */
void dfsFrom(const string& player, const Map<string, Set<string>>& tournament,
             Set<string>& visited, int distance) {
    /* If we've gone too far, we're done. */
    if (distance > 2) return;

    /* If we've seen this node before, we're done. */
    if (visited.contains(player)) return;

    /* Explore outward! */
    visited += player;
    for (string vanquished: tournament[player]) {
        dfsFrom(vanquished, tournament, visited, distance + 1);
    }
}

bool isWinner(const string& player, const Map<string, Set<string>>& tournament) {
    Set<string> visited;
    dfsFrom(player, tournament, visited, 0);
    return visited.size() == tournament.size();
}

Set<string> winnersOf(const Map<string, Set<string>>& tournament) {
    Set<string> result;
    for (string player: tournament) {
        if (isWinner(player, tournament)) result += player;
    }
    return result;
}
```

Problem Twelve: Tournament Victory Chains

The key operation we need to be able to perform is to extract, from a tournament, the two smaller tournaments you get by filtering down to just winners and just losers. The rest is handled by recursion.

```
/**
 * Given a larger tournament and a set of players, returns the tournament formed
 * from just the indicated set of players.
 *
 * @param tournament The larger tournament.
 * @param players The players in question.
 * @return A subtournament consisting just of the indicated players.
 */
Map<string, Set<string>>
subtournamentOf(const Map<string, Set<string>>& tournament,
                const Set<string>& players) {
    Map<string, Set<string>> result;
    for (string player: players) {
        /* Filter the victories just down to the people they beat who are in the
         * group we care about.
         */
        result[player] = tournament[player] * players;
    }
    return result;
}

Vector<string> victoryChainFor(const Map<string, Set<string>>& tournament) {
    /* Base case: If the tournament has no players, we can use the empty list. */
    if (tournament.isEmpty()) return {};

    /* Recursive step: Choose a player, split the tournament, concatenate the
     * paths we get back. For simplicity, we're going to start off by forming a
     * set of all the players so we can easily identify winners and losers.
     */
    Set<string> allPlayers;
    for (string player: tournament) {
        allPlayers += player;
    }

    /* Get a person to split on. */
    string player = allPlayers.first();

    /* Form the sets of people they won against and lost against. */
    auto vanquished = tournament[players];
    auto victors    = allPlayers - player - vanquished;

    /* Recursively form victory chains through those players. */
    auto chain = victoryChainFor(subtournamentOf(tournament, victors));
    chain += player;
    chain += victoryChainFor(subtournamentOf(tournament, vanquished));

    return chain;
}
```

Good question to ponder: Where have you seen this algorithm somewhere before? Or at least something very similar to it? What if you thought of "players" as "objects" and "games" as indicating which objects are bigger or smaller?

Problem Thirteen: Eccentricity

There are many ways you can code this up. Ours uses a cute insight. Imagine that we run a BFS starting at a given node. What can we say about the very last node that we dequeue out of the queue? That node has to be as far away as possible from the source node, since (1) BFS visits nodes in increasing order of distance and (2) no other nodes will be visited after it. As a result, to find the most distant node from a starting point, we can run a BFS starting at that node and look at the very last node we've found. That's the one that's as far away as possible.

```
int eccentricityOf(const Map<string, Set<string>>& graph, const string& node) {
    Queue<string> worklist;
    worklist.enqueue(node);

    /* Associate each element with a parent node. The parent node is the node that
     * added it into the queue, which means it'll be one step closer to the start
     * node. This is an alternative to using a queue of stacks or vectors that is
     * more commonly used and works much more efficiently.
     */
    HashMap<string, string> parents;

    /* Track the last node we've seen. */
    string last = node;

    /* Do the BFS! */
    while (!worklist.isEmpty()) {
        string curr = worklist.dequeue();

        for (string next: graph[curr]) {
            /* Don't revisit something we've already enqueued. */
            if (!parents.containsKey(next)) {
                parents[next] = curr; // We discovered this node.
                worklist.enqueue(next);
            }
        }

        /* Remember the last node we've seen. */
        last = curr;
    }

    /* Track back from this node to the start, counting how many steps were
     * needed.
     */
    int result = 0;
    while (parents.containsKey(last)) {
        last = parents[last];
        result++;
    }

    return result;
}
```