

## Section Solutions 2

---

### Problem One: Iteration Station

```
void iterateVec1(const Vector<int>& vals) {  
    for (int i = 0; i < vals.size(); i++) {  
        cout << vals[i] << endl;  
    }  
}
```

This one works just fine!

```
void iterateVec2(const Vector<int>& vals) {  
    for (int i: vals) {  
        cout << vals[i] << endl;  
    }  
}
```

This code will likely crash at runtime. The variable *i* here refers to a *value* in the *Vector*, not an *index* in the *Vector*, so using it as an index will jump to a spot that wasn't designed to be an index.

```
void iterateVec3(const Vector<int>& vals) {  
    for (int i: vals) {  
        cout << i << endl;  
    }  
}
```

This code works just fine!

```
void iterateSet1(const Set<int>& vals) {  
    for (int i = 0; i < vals.size(); i++) {  
        cout << vals[i] << endl;  
    }  
}
```

This code won't compile. You can't select elements of a set by index using the square bracket operator.

```
void iterateSet2(const Set<int>& vals) {  
    for (int i: vals) {  
        cout << i << endl;  
    }  
}
```

This code works just fine!

```
void iterateStack1(const Stack<int>& s) {  
    for (int i = 0; i < s.size(); i++) {  
        cout << s.pop() << endl;  
    }  
}
```

This code won't compile. The *Stack* here is passed in by *const* reference, which means that it can't be modified. However, calling *.pop()* modifies the stack.

```
void iterateStack2(Stack<int> s) {  
    for (int i = 0; i < s.size(); i++) {  
        cout << s.pop() << endl;  
    }  
}
```

This one fails for a subtle reason. Each iteration of the loop decreases *s.size()* by one. Coupled with the fact that *i* is increasing by one at each point, this will only look at the first half of the stack.

```
void iterateStack3(Stack<int> s) {  
    while (!s.isEmpty()) {  
        cout << s.pop() << endl;  
    }  
}
```

This code works just fine! It does make a copy of the *Stack*, but that's okay given that the only way to see everything in a stack is to destructively modify it.

## Problem Two: Debugging Deduplicating

There are three errors here:

1. Calling `.remove()` on the `Vector` while iterating over it doesn't work particularly nicely. Specifically, if you remove the element at index `i` and then increment `i` in the `for` loop, you'll skip over the element that shifted into the position you were previously in.
2. There's an off-by-one error here: when `i = vec.size() - 1`, the indexing `vec[i + 1]` reads off the end of the `Vector`.
3. The `Vector` is passed in by value, not by reference, so none of the changes made to it will persist to the caller.

Here's a corrected version of the code:

```
void deduplicate(Vector<string>& vec) {
    for (int i = 0; i < vec.size() - 1; ) {
        if (vec[i] == vec[i + 1]) {
            vec.remove(i);
        } else {
            i++;
        }
    }
}
```

## Problem Three: References Available Upon Request

*Topics: Reference parameters, range-based for loops*

Here's the output from the program:

```
1 3 7
1 3 7
2 4 8
2 4 8
```

Here's a breakdown of where this comes from:

- The `maui` function takes its argument by value, so it's making changes to a copy of the original vector, not the vector itself. That means that the values are unchanged back in `main`.
- The `moana` function uses a range-based `for` loop to access the elements of the vector. This makes a *copy* of each element of the vector, so the changes made in the loop only change the temporary copy and not the elements of the vector. That makes that the values are unchanged back in `main`.
- `heihēi`, on the other hand, uses `int&` as its type for the range-based `for` loop, so in a sense it's really iterating over the elements of the underlying vector. Therefore, its changes stick.
- The `teFiti` function creates and returns a new vector with a bunch of updated values, but the return value isn't captured back in `main`.

## Problem Four: The New Org Chart

Here's one possible implementation:

```
/* Given a person and the map of the bosses, returns the CEO of the company
 * that the indicated person works for
 *
 * We've taken in person by value rather than by reference here because inside
 * the body of the function we need to change its value, but we don't want to
 * change the value back in the caller.
 */
string ceoFor(string person, const Map<string, string>& bosses) {
    while (bosses.containsKey(person)) {
        person = bosses[person];
    }
    return person;
}

bool areAtSameCompany(const string& p1,
                    const string& p2,
                    const Map<string, string>& bosses) {
    return ceoFor(p1, bosses) == ceoFor(p2, bosses);
}
```

## Problem Five: Xzibit Words

One possible implementation is shown here:

```
string mostXzibitWord(const Lexicon& words) {
    /* Track the best string we've found so far and how many subwords it has. */
    string result;
    int numSubwords = 0;

    for (string word: words) {
        /* Store all the subwords we find. To avoid double-counting
         * words, we'll hold this in a Lexicon.
         */
        Lexicon ourSubwords;

        /* Consider all possible start positions. */
        for (int start = 0; start < word.length(); start++) {
            /* Consider all possible end positions. Note that we include
             * the string length itself, since that way we can consider
             * substrings that terminate at the end of the string.
             */
            for (int stop = start; stop <= word.length(); stop++) {
                /* Note the C++ way of getting a substring. */
                string candidate = word.substr(start, stop - start);

                /* As an optimization, if this isn't a prefix of any legal
                 * word, then there's no point in continuing to extend this
                 * substring.
                 */
                if (!words.containsPrefix(candidate)) break;

                /* If this is a word, then record it as a subword. */
                if (words.contains(candidate)) {
                    ourSubwords.add(candidate);
                }
            }
        }

        /* Having found all subwords, see if this is better than our
         * best guess so far.
         */
        if (numSubwords < ourSubwords.size()) {
            result = word;
            numSubwords = ourSubwords.size();
        }
    }

    return result;
}
```

In case you're curious, the most Xzibit word is “foreshadowers,” with 34 subwords!

## Problem Six: Jaccard Similarity

```
/* Given a Queue<string>, produces a string representing the k-grams it
 * contains. We take our parameter by value because the only way to read a queue
 * is to destructively modify it.
 */
string queueToString(Queue<string> kGram) {
    string result;
    while (!kGram.isEmpty()) {
        /* If we already had something before us, add a space. */
        if (result != "") result += " ";
        result += kGram.dequeue();
    }
    return result;
}

/* Returns a set of all the k-grams in the given input stream, represented as
 * strings.
 */
Set<string> kGramsIn(istream& input, int k) {
    /* Validate the input. */
    if (k <= 0) error("k must be positive.");

    TokenScanner scanner(input);
    scanner.addWordCharacters(""); // Not necessary, but nice!

    /* We're going to store the last k word tokens read in in this queue.
     * This makes it easy to shift in and shift out new words into our k-gram.
     */
    Queue<string> kGram;
    Set<string> result;

    while (scanner.hasMoreTokens()) {
        string token = scanner.next_token();
        if (scanner.getTokenType(token) == WORD) {
            kGram.enqueue(token);

            /* If this brought us up to size k, output what we have, then kick
             * out the oldest element so that we're down to size k - 1. The next
             * word token we find will then refresh us to capacity.
             */
            if (kGram.size() == k) {
                result.add(queueToString(kGram));
                (void) kGram.dequeue(); // Ignore return value
            }
        }
    }

    return result;
}

/* The set of all the words in the stream is just the set of 1-grams. Nifty! */
Set<string> wordsIn(istream& input) {
    return kGramsIn(input, 1);
}
```

```

/* Prompts the user for the size of a k-gram to use. */
int chooseKGramSize() {
    while (true) {
        int result = getInteger("Enter k: ");
        if (result > 0) return result;

        cout << "Please enter a positive integer." << endl;
    }
}

/* Prompts the user for a filename, then returns the k-grams in that file. */
Set<string> contentsOfUserFileChoice(int k) {
    ifstream input;
    promptUserForFilename(input, "Enter filename: ");
    return kGramsIn(input, k);
}

int main() {
    int k = chooseKGramSize();
    Set<string> s1 = contentsOfUserFileChoice(k);
    Set<string> s2 = contentsOfUserFileChoice(k);

    /* Compute |S1 n S2| and |S1 u S2| using the overloaded * and + operators. */
    double intersectSize = (s1 * s2).size();
    double unionSize     = (s1 + s2).size();

    cout << "Jaccard similarity: " << intersectSize / unionSize << endl;
    return 0;
}

```

## Problem Seven: Serving Your Constituents

There are *many* ways to solve this problem. Here's one.

```
Vector<string> topPriorities(const string& filename, int k) {
    if (k < 0) error("Cannot choose a negative number of elements.");

    /* If for whatever reason they want no suggestions, grant their wish! */
    if (k == 0) return {};

    /* Open the file for reading. */
    ifstream input(filename);
    if (!input) error("Cannot open file " + filename);

    /* Build a map associating each comment with its frequency. The syntax below
     * is the C++ standard way of saying "keep reading lines of the file until
     * there aren't any lines left to read."
     */
    Map<string, int> frequencies;
    for (string line; getline(input, line); ) {
        frequencies[line]++;
    }

    /* Now, build an mapping from frequencies to lines with that frequency. There
     * are a couple concerns to watch for here. First, there's the issue that the
     * Map type can only have one value associated with each key, and multiple
     * lines might have the same frequency. To address this, we'll associate each
     * frequency with a list of all lines with that frequency.
     *
     * Second, there's the issue of ordering. If we iterate over a Map, the list
     * of keys is returned in sorted (ascending) order. However, we want to see
     * things in descending order. There are many ways we could solve this
     * problem, and the one we'll pick here is to use as keys not the frequencies,
     * but the negative versions of the frequencies. To see why this works,
     * imagine we have items with frequencies 137, 42, and 6. If we make those
     * values negative, we get -137, -42, and -6, and visiting those values in
     * ascending order visits them in decreasing order of original size.
     */
    Map<int, Vector<string>> byRank; // Could also map to a Set, Lexicon, etc.
    for (string line: frequencies) {
        byRank[-frequencies[line]] += line;
    }

    /* Finally, pull off the top k elements. Visiting the Map in this order
     * gives the strings sorted by their frequency.
     */
    Vector<string> result;
    for (int key: byRank) {
        /* Visit all lines with this rank, adding them in. */
        for (string line: byRank[key]) {
            result += line;
            if (result.size() == k) return result;
        }
    }

    /* Didn't find enough lines? That's fine! Just return what we have. */
    return result;
}
```