Section 2. Containers

Section materials curated by Neel Kishnani. These problems are a joint effort by a number of different lecturers.

This week in section you'll be practicing with collection types and reference parameters. These problems will get you some good practice with topics that you'll see in assignment 2 and beyond. There's a starter package linked below. Have fun!

Starter code

Problem One: Iteration Station

Topics: Vectors, sets, stacks, range-based for loops, for loops, parameter passing

Below are a bunch of functions that attempt to iterate over a container of some type. For each function, determine whether

- the function works correctly
- the function compiles, but doesn't correctly iterate over the elements, or
- the function won't even compile

In each case make sure you can explain why!

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

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

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

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

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

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

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

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

Solution

Problem Two: Debugging Deduplicating

Topics: Vectors, strings, debugging

Consider the following ***incorrect*** C++ function, which accepts as input a **`Vector<string>`** and tries to modify it by removing adjacent duplicate elements:

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

The intent behind this function is that we could do something like this:

```
Vector<string> hiddenFigures = {
    "Katherine Johnson",
    "Katherine Johnson",
    "Katherine Johnson",
    "Mary Jackson",
    "Dorothy Vaughan",
    "Dorothy Vaughan"
};

deduplicate(hiddenFigures);
// hiddenFigures = ["Katherine Johnson", "Mary Jackson", "Dorothy Vaughan"]?
```

The problem is that the above implementation of **`deduplicate`** does not work correctly. In particular, it contains three bugs. Find those bugs, explain what the problems are, then fix those errors.

Solution

Problem Three: References Available Upon Request

Topic: Reference parameters, range-based for loops

Reference parameters are an important part of C++ programming, but can take some getting used to if you’re not familiar with them. Trace through the following code. What does it print?

```
void printVector(const Vector<int>& values) {
    for (int elem: values) {
        cout << elem << " ";
    }
    cout << endl;
}

void maui(Vector<int> values) {
    for (int i = 0; i < values.size(); i++) {
        values[i] = 1258 * values[i] * (values[2] - values[0]);
    }
}

void moana(Vector<int>& values) {
    for (int elem: values) {
        elem *= 137;
    }
}

void heihei(Vector<int>& values) {
    for (int& elem: values) {
        elem++;
    }
}

Vector<int> teFiti(const Vector<int>& values) {
    Vector<int> result;
    for (int elem: values) {
        result += (elem * 137);
    }
    return result;
}

int main() {
    Vector<int> values = { 1, 3, 7 };
    maui(values);
    printVector(values);
    moana(values);
    printVector(values);
    heihei(values);
    printVector(values);
    teFiti(values);
    printVector(values);
    return 0;
}
```

Solution

Problem Four: Another Way to Find the Maximum

Topics: Vectors, recursion

In lecture, we wrote a recursive function that found the maximum value in a **`Vector`**. In an appendix to that slide deck, there's code for a second, alternative recursive function to find the maximum value in a list. And now, here's a third algorithm for finding the maximum element:

```
int maxOf(const Vector<int>& elems) {
    if (elems.size() == 1) {
        return elems[0];
    } else {
        Vector<int> winners;
        for (int i = 0; i < elems.size(); i += 2) {
            if (i + 1 == elems.size()) {
                winners += elems[i];
            } else {
                winners += max(elems[i], elems[i + 1]);
            }
        }

        return maxOf(winners);
    }
}
```

Find a simple, intuitive explanation of how this code works. Then, mechanically trace through the execution of this function on the input

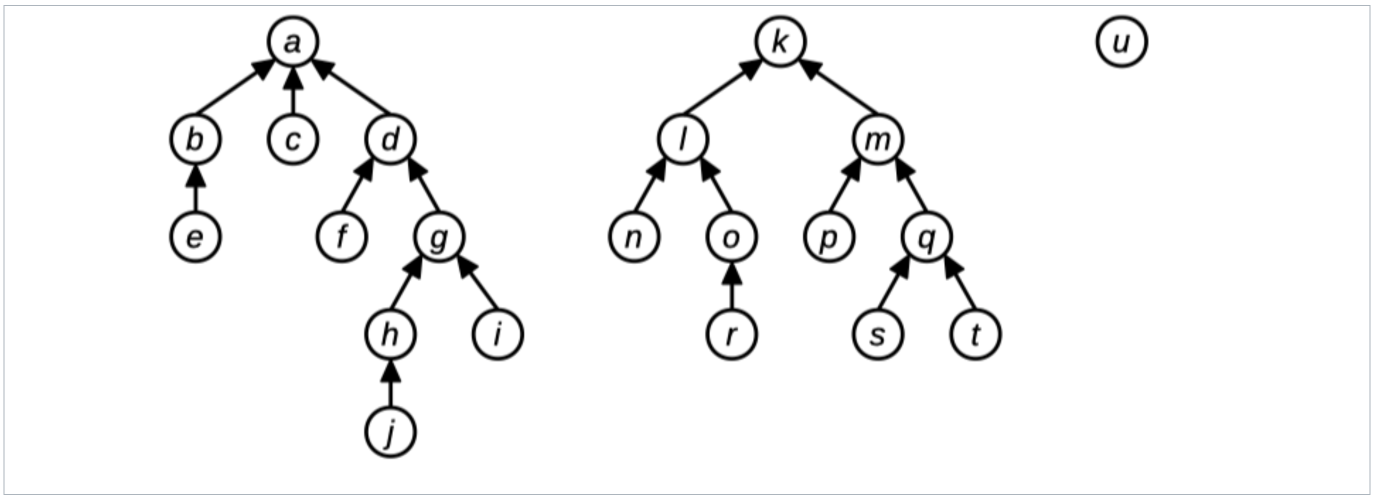
```
{ 271, 828, 182, 845, 904, 523 }
```

Solution

Problem Five: The New Org Chart

Topics: Maps, strings

Suppose you have the organizational hierarchies of various companies showing who reports to who. For example, across three different companies, you might have this hierarchy information:



In this diagram, person *e* reports directly to person *b*, who reports directly to person *a*, who is the CEO of her company and therefore doesn't report to anyone.

This hierarchical information can be represented as a **`Map<string, string>`** associating each person with their direct superior (that is, the person one level above them in the hierarchy). As an exception, the heads of each organization are not present in the **`Map`**, since they don't report to anyone. For example, in the above picture, the key *h* would have value *g*, the key *g* would have value *d*, the key *d* would have value *a*, and *a* would not be a key in the map. Since person *u* is at the top of her own company, the letter *u* would not be a key in the map either.

Given two people, you can tell whether they work for the same company by tracing from those people all the way up to their companies' CEOs, then seeing whether those CEOs are the same person. For example, in the above diagram, person *e* and person *j* work at the same company because both of them report (indirectly) to *a*. Similarly, person *n* and person *k* are in the same organization, as are person *c* and person *d*. However, person *j* and person *m* are not in the same company, since person *j*'s company is headed by person *a* and person *m*'s company is headed by person *k*. Along similar lines, person *u* and person *b* are in different companies, since person *u* runs her own company (she doesn't report to anyone) and person *b* works at a company headed by person *a*.

Your job is to write a method:

```
bool areAtSameCompany(const string& p1,
                      const string& p2,
                      const Map<string, string>& bosses);
```

that accepts as input two people's names and and a **Map** associating each person with their boss, then reports whether **p1** and **p2** work at the same company. You can assume the following:

- For simplicity, assume that each person has just one boss, that each company has just one CEO, and that no person works at two or more companies.
- Everyone, except for the heads of companies, appear as keys in the **Map**. Therefore, if someone doesn't appear in the **Map**, you can assume that they are the head of a company.
- Names are case-sensitive, so "Katherine" and "KATHERINE" are considered different people.
- **p1** and **p2** may be the same person.
- There can be any number of levels of management between **p1** and her CEO and between **p2** and his CEO, including 0, and that number of levels doesn't have to be the same.

Solution

Problem Six: Xzibit Words

Topics: Strings, lexicons

Some words contain other words as substrings. For example, the word "pirates" contains a huge number of words as substrings:

- a
- at
- ate
- ates
- es
- i
- irate
- pi
- pirate
- pirates
- rat
- rate
- rates

Note that "pirates" is a substring of itself. The word "pat" is not considered a substring of "pirates," since even though all the letters of "pat" are present in "pirate" in the right order, they aren't adjacent to one another.

Write a function

```
string mostXzibitWord(const Lexicon& words);
```

that accepts as input a Lexicon of all words in English, then returns the word with the largest number of words contained as substrings.

Solution

Problem Seven: Jaccard Similarity

Topics: Set, file reading, queues, strings

Let's suppose that you have two text files – say, web pages, inscriptions on pottery shards, political speeches, etc. – and you want to determine how "similar" those text files are. For example, you might be a search engine (like Google) and want to recommend pages similar to another one, or you might be trying to determine the author of an pseudonymous essay or piece of text. How might you go about doing this?

In this exercise, we're going to treat a text document not as a single piece of text, but as a *bag of words*. Instead of representing a piece of text as a linear sequence of words, we'll treat it as an unordered set of words. For example, Yogi Berra's famous quote:

"I didn't say all the things I said"

would be treated as the set:

```
{ "all", "didn't", "I", "said", "say", "the", "things" }
```

with the elements in no particular order and duplicate words removed.

The advantage of this approach is that we can approximate the similarity of two documents by using a measure called the ***Jaccard similarity***. Given two sets *S* and *T*, the Jaccard similarity of those sets, denoted *J(S, T)*, is defined as follows:

$$J(S,T) = \frac{|S \cap T|}{|S \cup T|}$$

Here, `|S ∩ T|` denotes the number of words in common to *S* and *T* (that is, the *cardinality* of their *intersection*), and `|S ∪ T|` denotes the number of words that appear in at least one of *S* and *T* (that is, the cardinality of their union). If the sets *S* and *T* are completely identical, then *J(S, T) = 1* (do you see why?), and if *S* and *T* have absolutely nothing in common, then *J(S, T) = 0* (again, do you see why?) As the overlap between *S* and *T* increases, *J(S, T)* starts to increase.

The advantage of Jaccard similarity is that two documents that have a lot of common words and phrases are likely to have a very large Jaccard similarity, while two documents that have very little in common are likely to have a low Jaccard similarity.

Write a function

```
Set<string> wordsIn(istream& input);
```

that takes as input an input stream, then returns a set of all the tokens in that input stream. Using this function, write a program that prompts the user for the names of two files, then computes their Jaccard similarity.

If you finish this one early, consider this variant on the problem. A ***k-gram*** is a sequence of *k* consecutive words or tokens out of a file, so given Yogi Berra's above quote, the 2-grams would be as follows:

```
"I didn't", "didn't say", "say all", "all the", "the things", "things I", "I said"
```

Jaccard similarity on documents tends to give much better answers when you compute it on 2-grams, 3grams, or 4-grams from the documents. Update your code so that it works with *k*-grams rather than individual tokens. This is similar to what you'll be doing on assignment 2!

Note: we won't be providing a fleshed out starter file for this problem, as it'll be good practice for you to write the entire program here!

Solution