CISC 3130
Exercise 1

Choosing Data Structures

In this exercise:

- you will be introduced to C++ libraries for timing, especially of code execution
- you'll do some practical, non-theoretical comparisons of different data structures
- you'll look closely at the available documentation and learn to extact useful information from it

Introduction

We'll be talking throughout the semester about the relative efficiency of different algorithms and data structures. Mostly, those discussions will be abstract—in the sense that we won't be talking about any *particular* implementation running on any *particular* computer. But it's also useful to be able to compare implementations on your actual computer, if for no other reason to provide some sense that our abstract discussions do have application in the real world.

The C++11 `chrono` library contains a rich infrastructure for dealing with time. For example, this little program times the execution of some code and outputs the result in milliseconds.

```
#include <iostream>
#include <chrono>

using namespace std;

int main(){

  auto start = chrono::steady_clock::now();

  // code to time here

  auto end = chrono::steady_clock::now();
  auto diff = end - start;

  cout << chrono::duration <double, milli> (diff).count() << " ms" << endl;
}
```

(Read this for a little more background and the original code.)

Measuring some Performance

So, let's use this to get a sense of what some the C++ library data structures can do. Declare a `set` of `ints`: `set<int> intset;`

(Note that we're `using namespace std`.) Put the values 0–10,000 into the set (say, with a `for` loop)—and measure the total time that takes. Then find each of those values in the set, and measure the total time of those find operations. Which took longer? How much longer? Based on this observation, sets are best used in applications where you expect more of which operation? Is that a common situation?

What about trying to find values that *aren't* in the set? Do you expect that to be more or less efficient than finding values that are in the set? Experiment with 10,000 find operations—maybe time 10,000 searches for –5, then try 10,000 searches for negative numbers, then 10,000 searches for numbers greater than 10,000. What pattern do you see, if any? Do your teammates, using different devices, see the same patterns? How do your timings compare to your teammates'?

Comparing Data Structures

Let's do the same thing with `stack`—time 10,000 adds and 10,000 removes. Before you run the code, make a hypothesis about what will happen: how will these timings compare to the `set` timings? Will `stack` add be faster, slower, or about the same as `stack` remove? When you have a theory, run the code and see what happens.

Probably, you will see that one of the two structures is considerably more efficient than the other. Does that mean you should choose that structure?

Now do this one more time, with `unordered_set`. Based on whatReaBeing the book says, how do you expect it to compare to `set`? Run the code and get the timings; were you right? What about searching for elements that aren't present? How does that compare, both to searching `set` for missing elements and to searching `unordered_set` for elements that are present?

Reading the Documentation

Being able to read the documentation effectively is key to being able to use the C++ libraries well. Revisit the `set` page. Where on this page is the information that tells you that `set<int> intset` is a legitimate declaration?

The page also says, "Search, removal, and insertion operations have logarithmic complexity." What does that mean? Is that good, or bad, or…?

Scroll down, and click on the link to the documentation for the `insert()` method. How many different versions of `insert()` does C++11 have? Which one did you use in your code?

Read about the "complexity" of the different versions. Which version is generally most efficient? In what situations are versions 5-6 going to be preferable? Why? Write some code that tests that hypothesis.