2.4 Submitting Your Work

When you're ready to submit your work for final grading on Coursera, you'll need to bundle it into a zip file in a specific way. To do that, you can type this command from your main subdirectory, where the Makefile is located:

```
make zip
```

This will automatically package the files to be graded into a new zip file, **`UnorderedMap_submission.zip`**, which will appear in the same directory. You can download the created zip file to your own computer from Cloud9 by right-clicking on the zip file in your Cloud9 file list and choosing "Download."

Once you've got the zip file on your local computer, you can log into Coursera and submit the zip file for this assignment. (If you need a reminder about how to use Cloud9 or how to submit work on Coursera, the first MOOC in this course sequence on Coursera has more detailed information about these topics in the lessons and homework assignments.)

3 Background Information

Here we discuss some of the topics, concepts, and syntax that the project touches upon. Much of this information is supplementary rather than essential, but please read to get the best understanding of what's going on in the project code.

3.1 Overview of `std::unordered_map`

The Standard Template Library in C++ offers a hash table implementation with **`std::unordered_map`**. It has this name because a hash table is a more specific implementation of the abstract data type generally called a "map," which associates each unique lookup key with an assigned value. It is called "unordered" in C++ to distinguish it from another map implementation, **`std::map`**. We will focus on **`std::unordered_map`** below, but let's briefly summarize how it's different from **`std::map`**.

The implementation of **`std::map`** uses some kind of balanced tree (the specifics may vary from system to system). It maintains keys in a *sorted* order as a result, which is desirable sometimes, such as when you want to iterate over the keys in the same order repeatedly. It offers $O(\log n)$ lookup times. In order for classes to be compatible as key types, they need to have an equality operation defined, as well as a "less than" operation defined for sorting.

By contrast, **`std::unordered_map`** is implemented as a hash table, and the keys are not arranged in any predictable order, but instead are placed in hashing buckets as described in the lectures on hash tables. It offers very fast $O(1)$ lookup times on average (that is, constant-time). For a class type to be compatible as a key, it must have an equality operation defined, and it also must have a hashing function defined. We'll talk more about that later in this document.

Although lookups on **`std::unordered_map`** are very efficient, after inserting many items, the table will be automatically "re-hashed" and resized, and that can periodically lead to temporary slowdown. In the future, if that is an issue you can do additional work to specify optimal sizes beforehand, but we will trust in the system defaults to balance the load factor and maintain the table organization.

We may refer to objects of type **`std::unordered_map`** simply as "maps" sometimes for shorthand, so in the future, please take note about what kind of map is being used. For this assignment, we will only use **`std::unordered_map`**.

3.2 Syntax for `std::unordered_map`

We import the necessary headers with:

```
#include <unordered_map>
```

To declare an instance of the class, we need to specify template parameters for the key type and the value type, like this:

```
// Create a single map instance where each string is mapped to one int
std::unordered_map<std::string, int> myMap;
```

In this case, the key type is **`std::string`**, and the value type is **`int`**. Note that STL already offers built-in hashing support for **`std::string`** and several other of the basic numeric types. (However, things will get a bit complicated later on when we need to specify our own class types as keys.)

We can do both lookups and assignments on our map using the **`[]`** operator:

```
myMap["five"] = 5;
std::cout << myMap["five"] << std::endl; // output: 5
```

It's *very important* to note that **`[]`** can only be used on non-**`const`** instances of a map, because it returns a direct reference to the mapped value item that it found (and that could be used to modify the entry). In addition, if the key item doesn't exist, it will be created as soon as you refer to it with **`[]`**, and initialized with some default value. Therefore, if you don't want to carelessly insert meaningless keys to your hash table, you should probably do something else to query whether a key exists before you look up its contents!

The **`count`** member function will tell you if a key exists already or not. The name is somewhat misleading; it only returns 0 (not found) or 1 (found). However, in conditional statements, C++ can treat those integers implicitly as "false" and "true." This is not to be confused with the **`size`** function, which tells you how many different keys there are total. Here is an example:

```
// Check how many keys there are before our query:
std::cout << "Map size: " << myMap.size() << std::endl;
if (myMap.count("five")) {
  std::cout << "Found \"five\" in the map. Value: " << myMap["five"] << std::endl;
}
else {
  // In this conditional branch, no lookup operation with [] happens at all!
  std::cout << "Did not find \"five\" in the map. The map is unchanged." << std::endl;
}
// Check how many keys there are at the end:
std::cout << "Map size: " << myMap.size() << std::endl;
```

Experiment with this to see whether the size changes or not after the check, depending on what you insert to the map beforehand! Note that although this example avoids bloating the map with new entries for keys not found, it is not optimally fast, because in those situations where the key does exist in the map, doing **`[]`** after **`count`** is essentially performing the same search a second time. In practice, if the hash table has few collisions, this shouldn't matter very much.

It's also worth noting that there are some other alternatives for performing lookups on **`std::unordered_map`**,

but they are a bit more advanced. For example, the **`at`** function will search for a given key and return a reference like **`[]`**, but **`at`** will throw an exception if the key is not found, instead of modifying the map. For this reason, **`at`** can be used on **`const`** map objects when **`[]`** cannot. There is also the **`find`** function, which actually returns an *iterator* type, which is like a special pointer. The iterator points to a key-value pair found, or otherwise to the map's end iterator given by **`end()`**. This is probably more cumbersome than using **`at`**, but it depends on how good you are at avoiding exceptions.

If you do iterate over a **`std::unordered_map`** (keeping in mind that there is no particular order to the items), the individual items are the key-value *pairs* themselves. C++ has a special representation of a pair, which we'll discuss next. You can see an example where we iterated over the pairs in **`UnorderedMapCommon.cpp`**, in the definition of the function **`sortWordCounts`**.

Lastly, the **`erase`** function can eliminate a key and its mapped value entirely.

3.3 Pairs

In C++, a special template allows you to pair one type with any other type as a single object. In many cases this is more convenient than creating an entire custom class just to bundle two things together. First, we should include this standard header:

```
#include <utility> // for std::pair
```

You may forget to do that sometimes because it is included along with certain other STL headers (like **`<unordered_map>`**), but pairs are also useful on their own. You can explicitly create a particular type of pair, and a particular instance of that type, like this:

```
std::pair<std::string, int> myPair;
myPair.first = "Hello, this is the string element.";
myPair.second = 42;
```

As you can see, the **`std::pair`** template implicitly has members **`first`** and **`second`** that correspond to the elements. There is another way to construct a new pair, using the helper function **`std::make_pair`**:

```
std::pair<std::string, int> anotherPair = std::make_pair("sevens", 777);
```

Since C++11, you can use a generic initializer list in brackets **`{...}`** to initialize a pair. (You can actually use this kind of generic syntax to initialize many STL types besides pairs. For that reason, it's not really compatible with the inferred **`auto`** type.)

```
std::pair<std::string, int> anotherPair = {"sevens", 777};
```

For convenience, we can make an alias to a particular type of pair we use a lot. You'll see some aliases like this defined in **`UnorderedMapCommon.h`**, and we use them in the project code.

```
using StringIntPair = std::pair<std::string, int>;
StringIntPair anotherPair = {"sevens", 777};
```

Some implementations of **`std::unordered_map`** actually uses **`std::pair`** internally for each key-value pair, and so some of the advanced helper functions of **`std::unordered_map`** directly deal with pairs. You can even initialize a map as a list of pairs:

```
std::unordered_map<int, int> lookupTable = {{1, 10}, {2, 20}, {3, 30}};
std::cout << lookupTable[2] << std::endl;
// output: 20
```

3.4 Issues with Hashing

Since hashing is required for key types to be compatible with **`std::unordered_map`**, in those cases where we want to use something elaborate as the key, such as a pair of items, we need to define a custom hashing function. The Standard Template Library provides hashing support for many of the primitive types you would want to use as keys, but it won't initially hash a custom class type you define yourself (not even a generic pair type that you've instantiated—but that may change in future versions of C++).

As Prof. Fagen-Ulmschneider explains in the lectures, it is *not easy* to make a good hashing function by yourself! There is a lot of advanced analysis in designing hashing functions for specific purposes, to ensure a low number of collisions, or to provide some other desirable guarantee. For example, "locality-sensitive hashing" is the idea that similar values should be hashed into the same, or nearly the same, bucket. (That can be useful for geometric applications. Points in 2D space are just pairs of coordinates, for example; and it's good to know when points are nearby.) The field of cryptography relies on hashing functions that have very strong theoretical guarantees about collisions. If you make a mistake in designing your hashing function, you may not only get very bad performance with your hash table, but you potentially could be creating a security risk. (In practice, the best way to avoid this particular danger is to *never* try to implement cryptography by yourself.)

For the sake of discussion, let's talk about a few practical and reasonable ways we can add hashing support for a custom type. The general principle will be to *avoid* literally defining our own hashing function, and instead rely on a known hashing function we already have access to.

If you are strictly dealing with integers, there are known ways to map a grouping of several integers *uniquely* to another integer. This is nice because in theory, it shouldn't have any collisions at all. However, in practice, our machine implementations of integers have finite precision, so we can still get collisions this way. (This concept is still useful as a basis for locality-sensitive hashing and some spatial algorithms. If you are interested, you might want to read about these topics: the Cantor pairing function, Morton code, Z-order curves.)

We won't use any of these methods for now. Instead, let's work with reliable hashing functions that STL already gives us. Next we'll discuss a trick that can apply these to new types.

3.5 String-Based Hashing

Instead of trying to come up with a mathematical solution ourselves, then, we can at least try this: We know that STL provides hashing support for **`std::string`** already, so we know it can hash an arbitrarily long sequence of bytes, and let us assume that it does this reasonably well for common applications—*not* cryptographic or security applications, but just for use with hash tables. Then we should be able to hash objects in general by doing this:

1. Convert the custom object type to a string representation. First, convert the separate parts of the object to strings.

2. Combine the partial strings in a unique, non-ambiguous way. Distinct objects should not coincidentally generate the same string after combination; an example is shown below. Any two unique objects must turn into different strings.

3. If the object is made up of sub-objects that contain more sub-objects, then the above steps might have to be applied recursively.

4. Take the overall combined string. Apply a known safe hashing function to the string.

For the sake of this project, we have already provided you the file **`IntPair.h`** which implements this for the type **`std::pair<int, int>`**. The standard include **`<string>`** gives us the default conversion function **`std::to_string`**, which can convert an int to a string. We convert both of the two ints to strings. Then, we combine them with the string "##" in the middle; this matters to ensure point 2 above, because otherwise string pairs like (1, 21) and (12, 1) would have the same representation:

- With simple concatenation: (1, 21) and (12, 1) both become "121".
- With "##" as a separator: (1, 21) becomes "1##21", while (12, 1) becomes "12##1".

If you want to use this separator trick for things other than integers, you need to make sure that the separator you insert cannot be anything that would appear in one of the component strings by itself. For example, if we were dealing with a pair of *strings* instead of ints, it is challenging to think of a separator sequence that would definitely not appear in one of the input strings.

3.6 STL Hashing Support

In the provided file **`IntPair.h`**, you can see how we defined the pair type **`IntPair`** and how we added hashing support for it based on **`std::string`**. There are quite a few comments in that file, but to summarize: In **`<functional>`**, STL provides **`std::hash`**, which is the basis for how types like **`int`** and **`std::string`** are hashed. Technically, the type **`std::hash`** is a "function object," which means it's an object type that is intended to be applied just as if it were a function, by overriding the **`()`** syntax. So, this is a class type that represents a hashing function. By default, STL containers like **`std::unordered_map`** that require a hashing function use **`std::hash`** internally to hash the key type. So, when we create a map, we don't have to tell it what hashing function to use, provided that **`std::hash`** already supports the key type we need.

The issue is that **`std::hash`** is templated to support *some* types by default, but not all. So, we need to extend the template to support our custom type. This is called *template specialization* in C++, and it has a special syntax beginning with a strange-looking, empty template argument list: **`template <>`**. If you are interested, for the rest of the details, please look over the source code in **`IntPair.h`**. We won't ask you to do this in the assignment, but you may want to do this yourself in the future as you use C++!

You may be wondering: What happens if we don't do this, and we try do declare our custom **`std::unordered_map`** anyway, even with an unsupported key type? Well, that just generates a compilation error, because **`std::unordered_map`** will try to instantiate **`std::hash`** in the necessary way, and it won't be able to find any definition for your key type. This seems to be a big point of confusion for beginners who want to experiment with hash tables. Just remember: you can't make a hash table until you have a hashing function.

3.7 Memoization

Here we'll briefly discuss a motivating example of how fast lookups with a hash table can accelerate an algorithm. We would like to cache any partial calculation results that could be reused efficiently while solving the same overall problem. This is one of the reasons why, at large scale, it is so important to have O(1) lookups (as with a hash table) instead of O($\log n$) (as with a balanced tree structure). This is

just scratching the surface of a much broader topic, and it's somewhat beyond the scope of this course, so we only introduce the topic for enrichment.

First, observe that it's a common problem-solving technique to break larger problems into smaller ones. This is one of the fundamental principles involved in recursion: the process must eventually reach a base case or recursion will never end, therefore making progress requires that the problem keep getting broken up into smaller problems. It's also implicit in iteration: When you iterate through a list and perform work on each item, you are *essentially* using a recursive approach, even if you are only writing a loop. You could state the logic like this:

Iterating over a list recursively:

1. If you are considering an empty list, stop.
2. Process the first item on the list.
3. Now, consider the rest of the list: all the items after the first. This is also a list itself, just smaller than before.
4. Run this same algorithm on the smaller list.

To recurse on the smaller list is basically the same thing as to "repeat from step 1." Hopefully you're beginning to see intuitively that there are only superficial differences between recursion and iteration. When we try to solve a new problem, regardless of whether we use loops or recursion, the important thing to focus on is breaking the problem down into smaller problems. The recurrence isn't always as simple as the linear relationship suggested in the example above. Sometimes, a problem breaks down in such a way that identical "subproblems" appear multiple times in its structure. There may be a way to index the distinct problems so that when the same essential problem comes up again, its previous result can simply be looked up from memory.

When we systematically label and record partial results, making sure to never recalculate anything needlessly, this is popularly called *memoization* (not "memorization," but *memo*-ization, as in "writing a memo" to oneself). In particular, this could be described as implicit memoization, since we can do this somewhat automatically just by adding a few hash table lookups and insertions to our code. One of the exercises in this project contains an interesting recursive problem that has already been solved for you. You'll get the opportunity to insert the hash table operations that accelerate it, and see the impact on the runtime.

In practice, it is sometimes possible to identify ahead of time the order of dependency among the subproblems in the overall structural pattern. In those situations, you could also potentially preallocate a static amount of memory for the partial results, and then explicitly calculate the subproblems in the order that maximizes the reuse of information. This analysis and problem-solving strategy in general is called *dynamic programming*. Learning how to identify the structural dependencies is more complicated than we can discuss here, but if you're interested, we recommend the book *Algorithms*, written by Prof. Jeff Erickson at the University of Illinois. You can find it here: http://jeffe.cs.illinois.edu/teaching/algorithms/

4 Exercises

Before you begin the exercises, please make sure you've read this PDF and examined the code in the provided files. It's always best to begin by reading through the code that you are given. There are comments throughout and many hints about how to do the exercises. Some of the helper functions, examples, and tests show coding techniques that are directly applicable to the exercises. (However, the code you need to write for the exercises is usually much simpler than any of the examples we provide.)

This week, there are three exercises to complete, located in **UnorderedMapExercises.cpp**. The exercises are clearly marked in the file with some additional instructions in code comments. Here we'll describe the problems as well. When you are working, be sure to compile and run both the **main** and **test** programs separately to make sure everything is correct.

4.1 Exercise 1: makeWordCounts

First, notice that this type alias is defined for convenience in **UnorderedMapCommon.h**:

```
using StringIntMap = std::unordered_map<std::string, int>;
```

You need to finish implementing the function **makeWordCounts** in **UnorderedMapExercises.cpp**, which has this function prototype:

```
StringIntMap makeWordCounts(const StringVec& words);
```

makeWordCounts takes a vector of strings, which are in no particular order and which may contain duplicates. You can assume that such a vector has already been created properly and passed to your function. (The **main** program tests this on the text of *Through the Looking-Glass*.) You need to make sure to prepare an empty StringIntMap on the stack in function scope, which should be returned from the function when you're done. You need to iterate over the vector **words** that has been passed in, using whatever form of iteration you choose, and for each unique string encountered in the vector, a entry for that string should be inserted in the map (the string itself is the key). The mapped value for each key should be the number of occurrences of that exact string.

Example: If the input is a vector containing {"dog," "cat," "dog"}, then the map should have these mappings:

Key: "cat" maps to value: 1
Key: "dog" maps to value: 2

You do not need to perform any string operations on the strings. For example, you do *not* need to change the strings to lowercase or parse the strings in any further way. You can handle each string exactly as it appears in the input.

4.2 Exercise 2: lookupWithFallback

You need to finish implementing the function **lookupWithFallback** in **UnorderedMapExercises.cpp**, which has this function prototype:

```
int lookupWithFallback(const StringIntMap& wordcount_map,
                       const std::string& key, int fallbackVal);
```

As described earlier in this document, sometimes it's not the best practice to use **[]** in situations where lookups would add undesired keys to your map. Moreover, you can't use **[]** with a **const** map object at all. This is a wrapper function for safely performing lookups on a read-only **std::unordered_map** object.

The function should *not* modify the original map that is passed in. It should search for the key in the map, and if the key exists, then the corresponding value should be returned. If the key sought is not found, then the provided fallback value argument should be returned instead. (In some other programming languages, there is a standard library function that behaves this way.)

4.3 Exercise 3: Memoizing a Function

This exercise is mostly a conceptual illustration of the memoization topic introduced earlier in this document. There is not that much you need to change in the code. But, the small changes you'll make to the provided code will have a drastic impact on the performance of the provided **`main`** example program.

A *palindrome* is a word that remains the same if its spelling is reversed. For example, *dad* and *mom* are palindromes. In the provided code file **`UnorderedMapCommon.cpp`**, there is a definition of a function called **`longestPalindromeLength`**, which is a very slow function. Given a string and two integer indices representing left and right limits for our search, we want to find the length of the longest palindrome substring anywhere between the left and right limit indices. (A substring needs to be made of contiguous characters, so we can't skip any characters to form a substring. Using the initial string "abcd" as an example, "bc" is a substring, but "bd" is not a substring.) We can understand the problem recursively by realizing that for any palindrome, these cases are true:

- An empty string is a palindrome.
- A single-character string is a palindrome.
- For other strings, if the first and last characters match, and the substring contained in between those is a palindrome, then the entire string is a palindrome. (For example, in the string *DAD*, since $D = D$, and the middle part A is a palindrome, then the entire string *DAD* is a palindrome.)

Since our function returns the length of the longest palindrome found anywhere between the limits, as an example, given this string: "xyzwDADxyzw," and these limits: 0 and 10 (which are the first and last character indices), we can calculate that the longest palindrome length is 3, because "DAD" is the longest palindrome substring to be found.

This calculation can be very slow because a naive program may exhaustively re-check the same internal substrings many times, and indeed, **`longestPalindromeLength`** will run very slow on large input strings.

In Exercise 3, there is an edited version of the palindrome finding function, which is called **`memoizedLongestPalindromeLength`**. This version of the function takes an extra parameter, **`LengthMemo& memo`**, which is the "memoization table," that is, a hash table to be used for caching calculation results. The **`LengthMemo`** type is defined in **`UnorderedMapCommon.h`** like this:

```
using LengthMemo = std::unordered_map<IntPair, int>;
```

Therefore, **`LengthMemo`** is an **`unordered_map`** where each key is an **`IntPair`** (using the custom hashing scheme we discussed earlier in this document), and each mapped value is an **`int`**. The **`IntPair`** key is a pair of left and right index limits, and the mapped **`int`** value is the recorded calculation result. So for the same "xyzwDADxyzw" example given above, one entry in the map would be this, which signifies the result for the entire length of the string:

Key: the pair (0, 10)
Mapped value: 3

Here is your task: In order to make use of the **`memo`** object for caching purposes, you need to edit the memoized function in two different, very specific places, which we have clearly marked with comments as "PART A" and "PART B." There are also other hints given in the function's comments, clearly marked "EXAMPLE."

If you examine the source code in **`main.cpp`**, you'll see that there are several sizes of test case strings that you can enable. Some of those will run extremely slow without memoization, and comparatively

extremely quickly after memoization is enabled. To prevent confusion, we've added a mechanism that passes the running time into the palindrome-finding function, and it will time out and stop if things take more than a few seconds. However, even on the small examples, after you have completed this exercise, you'll be able to see a measurable difference, which the **`main`** program output will report to you in the terminal.

The unit testing suite will check your code for correctness another way, by verifying that you have filled the memoization hash table accurately. When the table is correctly filled, the provided **`reconstructPalindrome`** function will be able to identify the actual palindrome substring based on the lengths recorded in the table. You can study that function's comments to see how it works too. In practice, it's common to use such a reconstruction method with dynamic programming algorithms.