## Problem 2

If we try to compile the given code, we get a compile-time error because the class *Coord* does not have a defined comparison operator, and some functions in *Map.h* (e.g. *combine* or *find*) compare objects with type *KeyType* (i.e. *Coord*). Since the compiler doesn’t automatically overload the comparison operator for classes either, the given code will not compile unless we define a comparison operator for *Coord*.

## Problem 3

### Part a.

First, we examine the innermost loop that performs some constant number of operations each time and iterates N times (with variable *k*). The innermost loop therefore has a time complexity of O(N).

Inside the second-innermost loop, we either do a simple comparison and *continue* or execute the innermost loop. We only *continue* in cases where *i* equals *j*; since *i* is fixed, this means we only continue 1 in N times, and execute the innermost loop N – 1 times. Since the second-innermost loop iterates N times and we run through the innermost loop (time complexity O(N)) almost each time, the second-innermost loop has time complexity O(N2).

Since the outermost loop always iterates N times and the code within it has time complexity O(N2), the outermost loop has time complexity O(N3).

Since it is preceded by a few lines of code that amount to a constant number of operations, the time complexity of the algorithm is **O(N3)**.

### Part b.

First, we examine the innermost loop that performs some constant number of operations each time and iterates N times (with variable *k*). The innermost loop therefore has a time complexity of O(N).

Inside the second-innermost loop, we now just perform basic operations and execute the innermost loop. Since the second-innermost loop iterates *i* times and we run through the innermost loop N times almost each time, the second-innermost loop has time complexity O(N\**i*).

Within each ith iteration of the outermost loop, then, we have to the order of N\**i* operations. We can then express the order of the total number of operations by summing N\**i* for *i* in [0, N). Since the summation of all *i* in [0, N) is to the order of N2, we get time complexity O(N3) for the outermost loop, with a smaller constant of proportionality than that in the outermost loop of the previous algorithm.

Since the outermost loop is only preceded by a few lines of code that amount to a constant number of operations, the time complexity of the algorithm is **O(N3)** with a smaller constant of proportionality.

### Problem 4

Let’s first evaluate the time complexity of the different functions called during the execution of *reassign*.

#### Get

If we call *get* with parameter *i*, the algorithm (with some optimisations) effectively traverses through the list to find the *i*th element. For a linked list of N elements, since it takes constant time to travel from one node to the next/previous node and in each step there are a constant number of operations (such as key comparison), the time complexity of the *get* function is O(N).

#### Insert

Since the *insert* function creates a new Node with the passed-in key and value (constant operations) and updates pointers in the linked list so as to insert it at the tail end of the list (constant operations), the *insert* function has time complexity O(1).

#### Swap

Since the *swap* function merely reassigns some pointer values and takes the same number of operations regardless of the respective sizes of the linked lists passed in, it has time complexity O(1).

#### Destructor for Map

The destructor Map performs a series of constant time operations (e.g. *Node\** initialisation, following a pointer, calling delete) for each Node in the Map i.e. it executes an O(1) algorithm for each of N elements. It therefore has time complexity O(N).

### Part a.

On average, *m* is not going to be empty – we will therefore ignore the *if* condition and evaluate the time complexity of the algorithm within it

We first evaluate the time complexity of the *for* loop. Inside the loop, all the operations we perform are constant time (including the call to *insert*, as proven) except the call to *get*, which has time complexity O(N). Since the loop is called N (equivalent to *m.size()*) times and the code within it has time complexity O(N), the loop has time complexity O(N2).

Outside the loop, we see that since the solitary call to *get* always takes parameter *i* = 0, it just finds the first node of the list and therefore takes constant time. The declaration of *prevKey* and *value0* are also naturally constant time. The call to *swap* too, as always, takes constant time. In summary, all the operations outside the *for* loop take constant time.

There is one more function call to consider: that of the destructor for *res* called automatically at the end of the function. Since the destructor has time complexity O(N) and the rest of the algorithm has time complexity O(N2), it doesn’t affect the total time complexity of the algorithm (N is lesser order since N2).

The time complexity of the *reassign* function is therefore **O(N2)**.

### Part b.

First, we evaluate the *for* loop. Inside the loop, we follow a few pointers and do a reassignment, all operations that take constant time. The body of the loop therefore has time complexity O(1). Furthermore, loop iterates until the *p* (initialised to a pointer to the first node object) points to *m\_head* – in other words, the loop iterates until we have traversed every single element of the list and arrived back to *m\_head*. The loop in total therefore has time complexity O(N), since the list is assumed to have N elements.

Since all the operations outside the *for* loop are constant time (simple comparison, initialisation, following pointers), the time complexity of the algorithm reduces to the time complexity of the loop. The time complexity of the algorithm is therefore **O(N)**.

This time complexity is order of N better than the time complexity of the algorithm in *part a*.