Joakim Rosenqvist
Assignment 2
1DV516 -HT19

Assignment 2

I will offer explanations and break-down of time complexities for Exercise 2 and 3, and additionally an explanation of the hash table in Exercise 1, since it is used in the subsequent exercises.

Exercise 1

The array keeping elements in the hash table is of size > 2 * elements, and always a prime number. This ensures every available location can be filled. When the array needs expanding, the closest prime larger than twice the current size is chosen. The algorithm for finding a new prime has a complexity of √(n).

Inserts will try to place a new elements in (*hash* % *array length*), and then according to the quadratic probing step i^2 until an empty slot is found. If necessary the array will expand and rehash.

The inserted elements are placed in a container object that has a flag for if the element is deleted or not. Elements marked for deletion will be skipped over in the *contains* and *delete* methods, and will be removed from the array on next rehash. The reason for doing so is to have contains and delete stop at a null index. So if two inserted elements would have the same original hash, the methods can still stop at null, and a rehash isn't necessary for every deletion.

Worst case time complexities for delete and contains are O(N/2) where N is size of array. Inserts have a worst case complexity of O(N), since two O(N/2) operations may be necessary.

Exercise 2

The itinerary is based on a A2Direction array that holds the steps. RotateRight is a O(N) operation that goes through the array and changes values based on what happens on a rotation, for example UP becoming RIGHT. The change is made mathematically based on the fact that rotational directions are two steps apart in the enumerator.

Width and height are calculated by iterating through the array, and whenever a direction of the correct axis is found, a new iterator starts and keeps count of how many times it can be found before opposing direction. The maximum count found is kept track of and finally returned. This operation is O(N^2), since for every iteration, up to N-1 additional iterations will be done.

The class constructor is O(1). It assigns the argument array a class-wide variable, and additionally an array is filled with the A2Direction values, which is a constant T(4).

GetIntersections works by adding all itinerary values to a custom coordinate class and inserting it to an instance of MyHashTable. On insertion a, contains operation is also performed on the hash table, to see if that certain coordinate already exists. Since every value of the itinerary also needs a contains operation on the hash table, time complexity of inserts is a worst-case of O(N^2). After every value has been inserted and intersections have been counted, a O(N) operation is performed to create a return array of every intersection, that has previously been tracked in a boolean array. Overall time complexity is thus O(N^2).

Exercise 3

IsSameCollection method makes use of MyHashTable. If arrays are of different length, the method will immediately return false. Otherwise, array1 is inserted into the HashTable, and a contains operation is performed for every value of array2. Both inserts and contains in the HashTable are worst case O(N) operations, meaning the overall time complexity of the method is O(N).

MinDifferences makes use of a custom merge sort implementation. If arrays are of equal length, both will be sorted using merge sort, and after that corresponding positions will be added together using the provided algorithm. The time complexity of merge sort is O(N log(N), meaning that is also the overall time complexity of this method.

GetPercentileRange also makes use of the merge sort implementation. The input arrays is sorted, and a return array of correct size is created. An iterator starts from the lower percentile of the input array and continues until it reaches the upper percentile of the array. The most time consuming part is the merge sort, which has a time complexity of O(N log(N)), giving an overall time complexity of this method of O(N log(N)).