**Exercise 2:**

**getIntersections():** The method goes throw the array and adds each point has visited ”MyHashtable”. Each time we visit a point, we check if it already exists in the HashTable. If ”Yes” that means we already visite it, else this is the first time and we should add it to the HashTable.

|  |  |  |  |
| --- | --- | --- | --- |
| public int[] getIntersections() { |  | |  |
| int x =0; | O(1) | | O(1) |
| int y=0; | O(1) | | O(1) |
| int counter=0; | O(1) | | O(1) |
| int [] result = new int [array.length]; | O(1) | | O(1) |
| MyHashTable <Position> table = new MyHashTable<Position>(); | O(1) | | O(1) |
| for (int i = 0; i<array.length; i++) { | N times | | O(N) |
| if (array[i] == A2Direction.LEFT) | O(1) | O(1) |
| x--; | O(1) |
| else if (array[i] == A2Direction.UP) | O(1) |
| y--; | O(1) |
| else if (array[i] == A2Direction.RIGHT) | O(1) |
| x++; | O(1) |
| else if (array[i] == A2Direction.DOWN) | O(1) |
| y++; | O(1) |
|  |  |  |
| Position point = new Position(x, y); | O(1) | O(1) |
| if (table.contains(point)) | O(1) (amortized) | O(1) |
| result[counter++] = i; | O(1) |
| else | O(1) |
| table.insert(point); | O(1**)** (amortized) |
| } |  | |
| int [] finalResult = new int [counter]; | O(1) | | O(1) |
| for (int i=0; i< counter ; i++) { | K times (the number of points that has been reached more than one times) | | O(K) |
| finalResult[i] = result[i]; | O(1) | |
| } |  | |
| return finalResult; | O(1) | | O(1) |
| } |  | |  |

**Complexity** = O(1)+ O(1)+ O(1)+ O(1)+ O(1)+ O(N)+ O(1)+ O(K)+ O(1) = O(N) + O(K)   
Since (K ≤ N) 🡺 **Complexity** = O(N)

**Exercise 3: isSameCollection(int[] array1, int[] array2):**

We modified the HashTable from ex1 to be able to contain (key, value) elements.

|  |  |  |  |
| --- | --- | --- | --- |
| public boolean isSameCollection(int[] array1, int[] array2) { |  | |  |
| if (array1.length != array2.length) | O(1) | | O(1) |
| return false; |  | |  |
| MyModifiedHashTable<Integer, Integer> hashtable = new MyModifiedHashTable<Integer, Integer>(); | O(1) | | O(1) |
| int count = 0; | O(1) | | O(1) |
| for (int i = 0; i < array1.length; i++) { | N times | | O(N) |
| if (hashtable.get(array1[i]) == null) | O(1)  (amortized) | O(1)  (amortized) |
| hashtable.insert(array1[i], 1); | O(1)  (amortized) |
| else { |  | O(1) (amortized) |
| count = hashtable.get(array1[i]) + 1; | O(1)  (amortized) |
| hashtable.insert(array1[i], count); | O(1)  (amortized) |
| } |  |  |
| } |  |  |
| for (int i = 0; i < array2.length; i++) { | N times |  | O(N) |
| if (!hashtable.contains(array2[i])) | O(1) (amortized) | O(1) |
| return false; |  |
| if (hashtable.get(array2[i]) == 0) | O(1) (amortized) |
| return false; |  |
| count = hashtable.get(array2[i]) - 1; | O(1) (amortized) |
| hashtable.insert(array2[i], count); | O(1) (amortized) |
| } |  |  |
| for (int i = 0; i < array1.length; i++) { | N times |  | O(N) |
| count = hashtable.get(array1[i]); | O(1) (amortized) | O(1) |
| if (count > 0) | O(1) |
| return false; |  |
| } |  |  |
| return true; |  |  |  |
| } |  |  |  |

**Complixity:** O(1)+ O(1)+ O(1)+ O(N)+ O(N)+ O(N) = O(N)

**Exercise 3:**

* **minDifferences(int[] array1, int[] array2):** the method does a mergeSort for both arrays. Then it computes the result.
* We implemented our own mergeSort method. The complexity for merge-sort is O(NLog(N)).

|  |  |  |
| --- | --- | --- |
| public int minDifferences(int[] array1, int[] array2) { |  | |
| if (array2.length != array1.length) | O(1) | O(1) |
| return 0; |  |  |
| int[] sortedArray1 = mergeSort(array1); | O(NLog(N)) | O(NLog(N)) |
| int[] sortedArray2 = mergeSort(array2); | O(NLog(N)) | O(NLog(N)) |
| int totalDiff = 0; | O(1) |  |
| for (int i = 0; i < sortedArray1.length; i++) { | N times | O(N) |
| totalDiff += (sortedArray1[i] - sortedArray2[i]) \*(sortedArray1[i] - sortedArray2[i]); | O(1) |
| } |  |  |
| return totalDiff; |  |  |
| } |  |  |

**Complexity:** O(1)+ O(NLog(N))+ O(NLog(N))+ O(N)= O(NLog(N))

* **getPercentileRange(int[] arr, int lower, int upper):** The method does a sort-merge for the array, then it returns the result.

|  |  |  |
| --- | --- | --- |
| public int[] getPercentileRange(int[] arr, int lower, int upper) { |  |  |
| int[] sortedArr = mergeSort(arr); | O(NLog(N)) | O(NLog(N)) |
| int lowIndex = lower \* sortedArr.length / 100; | O(1) | O(1) |
| int highIndex = (upper \* sortedArr.length / 100); | O(1) | O(1) |
| int[] result = new int[highIndex - lowIndex]; | O(1) | O(1) |
| for (int i = lowIndex; i < highIndex; i++) | N times | O(N) |
| result[i - lowIndex] = sortedArr[i]; | O(1) |
| return result; |  |  |
| } |  |  |

**Complexity:** O(NLog(N))+ O(1)+ O(1)+ O(1)+ O(N)= O(NLog(N))