Robin Hägg (rh222dy) & Björne Laanemäe (bl222ey) - 20/10 2018

1DV516 - Assignment 2

Exercise 1 - HashTable with quadratic probing

This is to be used in Exercise 2.

Given a prime-sized table, (n, n^2, n^3, …,) % tableSize will give (tableSize+1)/2 distinct values. As such, if the table is more than half full, there is no guarantee that we will find a free space. Therefore, the load factor is limited to max 0.5.

Insertions

The general idea behind inserting is:
Calculate hashcode O(1)
Check position in table O(1)
If fail, calculate next position, O(1), and check again - times ?
Rehash O(?)
Insert in empty place O(1)

Excluding rehashes, examining the case where each inserted element has the same hashcode we can see that the inserts will follow the pattern O(1), O(2), O(3), ... , O(N) where N is the number of elements to insert.The increasing time is because of already taken places, O(1) here is not one operation but “a few”.

Thus, worst case is O(N) and best case O(1) excluding rehashes.

When the load factor reaches 0.5, a rehash is done, and all elements are re-inserted in a fresh roughly twice as large table.

X - Place found instantly
XX - Skip first, then, place found
XXX - ….
…
N*X

The above text representation shows that the operations visualized as Xs will form a triangle of height N which has the area N^2/2. Amortized over N inserts this gives N/2 = O(N) operations per insert in addition to the O(N) we already have, which is still O(N).

Therefore it still holds that worst case is O(N).

Lookup

Lookup is the identical to the first step of insert where we try to find an empty position in the table, with no risk of rehash. This is O(N).

Exercise 2

For getIntersections() simply used a hash table.

Pseudocode:

```
init intersections
init position = origo
For [STEP]:
        position += STEP
        if lookup(position):
                push position to intersections
        insert(position)

return intersections
```

The idea is to iterate through the array of steps, derive current position, and use the hashtable to see if the position has already been visited. Using the analysis done before (X-triangle) we can see that the same applies here, but with a lookup/insert pair.

```
X (lookup)
X (insert)
XX ...
XX ...
XXX ...
XXX ...
...
```

Since we have already shown that amortized rehashing does not make inserts >O(N), we can conclude that getIntersections() is (O(N) + O(N))*N_STEPS = O(N^2) worst case but in reality much faster. This can be improved upon but not below O(N). For example we could have a boolean insert() operation which inserts if not in table and returns status.

Exercise 3 - MyMeasure

isSameCollection

For this exercise we have made a slight adjustment of *MyHashTable* called *CollectionComparationTable*. The differences are that the isDeleted property is replaced with a count, allowing for more of the same element to be inserted. Each insertion of the same

element increases the count by one, each deletion lowers it. A count of zero is equivalent to the element being deleted, and it's never lowered below this point.

The first thing our implementation does is to compare the length of the arrays. If they are not the same, the arrays must differ.

If not, we insert all the elements of one of the arrays into the CollectionComparationTable, and then we remove all the elements of the second one. If the two arrays do indeed contain the same elements, the size of the CollectionComparationTable becomes zero. If there is a mismatch between the arrays, some elements of the first one will never be deleted, and the size is non-zero.

Both insertion and deletion from the CollectionComparationTable is O(N) amortized. This means that *isSameCollection* has a time complexity of 2 + N*O(N) + N*O(N) + 1 = O(N^2).

minDifference

Here we begin with sorting the two arrays with merge sort, which we know is (if correctly implemented) O(N log N). Then the looping over both arrays simultaneously is an O(N) operation. Inside we have a one liner with cost O(1).
All in all, 2*O(N log N) + O(N) = O(N log N), which is our worst case.

Our merge sort

First we make a copy of the incoming array - O(N).
The recursive call *split* splits the array down the middle, until the length of each half is less than two. Such a halving operation is O(log N).
Next, we call *merge* on the two halves, to sort the two halves into one sorted part double its size. We create a copy of both the halves, and sort them back into place. this is ½O(N) + ½O(N) + O(N) = O(N). *Merge* is called O(log N) times, since it's part of the recursive tree of *split*.

In total, this gives is O(N) + O(log N) + O(N log N) = O(N log N).

getPercentileRange

We spent **a lot** of time trying to get a pivoting algorithm to work, which is why the report is somewhat rushed.

The idea behind this algorithm we devised is as follows (we have included code that does not work but shows the idea).

Given the input array of length N, transform the percentile range into an index range, eg 10-90 <-> 1,8 for a 10-length array. Then, pick a pivot as in quicksort and partition. If the

partition is below the low percentile or above the high percentile, adjust the “active range” (discard elements that we now know won’t be in the final set). Repeat this until: The pivot ends up between or on any of the “percentile indices” (1,8 in this example).

Then, start working with the same idea on the “lower part” and “higher part” as is shown in the lecture slides where one wants to find the median. Instead of finding the median, we end up with two partitioned sub-arrays where the last pivots is on the low index and high index.. Then, one can simply pick the lower part of the higher subarray and the higher part of the lower subarray.

Without actually fully sorting, this algorithm would have given a somewhat unsorted set with all element in the desired percentile range.

This was also done in an in-place fashion, so while there is talk of sub-arrays, they are written back instantly to the input. See code (Pivoteer.java, PercentilePartition.java) for a deeper explanation. Both are different attempts, Pivoteer being the newest.

Handed in getPercentileRange

The solution we hand in instead uses the merge sort we already implemented to sort the array, and then printing the values from the concerned indices. A much more boring solution.

Sorting the array with mergesort is O(N log N). We then find out the low and high index, a constant time operation. Finally we copy the range we care about into a new array (to be returned), which in case of the whole range being requested is O(N).

So, in the end, this comes down to O(N log N) + O(1) + O(N) = O(N log N).