Homework 8

Rumen Mitov

March 31, 2024

Problem 8.1

NOTE: I am using 0-based indexing for my pseudocode.

The algorithm starts off similar to count sort until it generates the cumulative array. Since we have the cumulative count at each position of the array, all we have to do is to return the difference in the counts of A[lo] and A[hi], provided that our boundary is [lo, hi].

Algorithm 1 NumbersInInterval(A, lo, hi)

```
for $i \leftarrow 0$ to $i < A.size()$ do
    $cumulative[A[i]] \leftarrow cumulative[A[i]] + 1$
end for
for $i \leftarrow 0$ to $i < cumulative.size()$ do
    $cumulative[i] \leftarrow cumulative[i] + cumulative[i-1]$
end for
      return $cumulative[hi] - cumulative[lo]$
```

Worst-case time complexity for bucket sort is when all the elements fall within the same range and are thus places in the same bucket. In this case we would have one bucket with n-elements and (n-1) empty buckets. Thus the time complexity will be determined by the time complexity of the underlying sort algorithm used to sort each individual bucket. Asumming we use insertion sort for this (as discussed in class), insertion sort operates at $O(n^2)$ and $\Omega(n)$. Taking the worst case for insertion sort (i.e. elements in the bucket are not pre-sorted) gives us $T(n) = O(n^2)$ for bucket sort.

Problem 8.2

Time Complexity

In terms of time, the only non-O(1) operations are the call to *count sort* and the *for-loop*. We already know that count sort is $\Theta(n + k)$. The for-loop will run $\Theta(hi - lo - 1) = \Theta(n)$ amount of times. In the body of the for-loop we have a recursive call, which esentially calls count sort k-amount of times (i.e. once for each digit). Hence, the last part of the algorithm has a time complexity of $\Theta(nk)$ Note, that the part of the algorithm after the for-loop is called to sort the section of values that have not yet been sorted. This would be the case if in the for-loop indexes 0 to i were sorted, and so the last part sorts i to n. Thus this conditional part is technically part of the $\Theta(nk)$ time complexity of

the for-loop.
Finally, since $\Theta(nk) > \Theta(n+k)$ we can say that the overall time complexity of this implementation is $\Theta(nk)$.

Space Complexity

In terms of memory, the only non-O(1) operations are once again the call to *count sort* and the *for-loop.* We already know that count sort takes up $\Theta(k)$.
The for-loop will run $\Theta(hi - lo - 1) = \Theta(n)$ amount of times. In the body of the for-loop is a recursive call which basically calls count sort k-times. Hence the last part of the algorithm will use $\Theta(nk)$ memory.
And since $\Theta(nk) > \Theta(k)$ the overall space complexity is $\Theta(nk)$.