dds Homework 8

P4

(a) see code implementation in VS code

(b) see code implementation in VS code

(c) Pseudocode:

```
COUNT-RANGE-SETUP(A, k)
let C[0..k] be a new array
n = A.length
for i = 0 to k
    C[i] = 0  /* we initialized */
for i = 0 to n-1
    C[A[i]] = C[A[i]]+1  /* we counted the frequencies */
for i = 1 to k
    C[i] = C[i] + C[i-1] /* we build the prefix sum */
return C // return the prefix sum array
COUNT-IN-RANGE (A, a, b)
  if a > 0  /* this means we can safely substract to get # in [a, b] */
      return C[b] - C[a-1] /* the count of elements in [a, b] = elements ≤ b minus the elements < a */
  else // if a = 0, we can't substract C[a-1] (out of bounds)
      return C[b] /* we just return the count of elements ≤ b, which is [0, b] */
```

(d) See code implementation in VS code;

Q) The worst-case for Bucket-Sort occurs when all of the elements fall into one single bucket. This would imply that the algorithm will need to sort all n elements within that bucket. In our implementation, we are using insertion sort to sort the individual buckets. Insertion sort is known to have a worst-case time-complexity of $O(k^2)$ (k - number of elements in a given bucket). If all elements of the array fall into a single bucket, by sorting the bucket with insertion sort it would take $O(n^2)$, because there would be n elements in that bucket. Breakdown of the complexities:

→ Distributing the elements into buckets: $O(n)$

→ Sorting each bucket (with insertionSort, while all n elements gathered in one bucket) = $O(n^2)$

→ Concatenating all of the sorted buckets (only one in our case) into the array: $O(n)$

⇒ Thus the overall complexity is $O(n^2)$ for the worst-case scenario.

P2. (a) The code implementation in VS code

(b) Time complexity analysis for Hollerith's original radix sort (starting from the most significant bit) work recursively, sorting numbers based on the most significant digit first and moving towards the least significant digit. We can go through our implementation from (a):

1) We use a function to determine the maximum value (Max(a)) => Time complexity (O(n)) the array is scanned once

2) The bucket distribution inside our MostSignificantRadixSort => Time complexity O(n) because each number is placed into one of the 10 buckets (based on the current significant digit)

3) The recursive calls in the MostSignificantRadixSort splits into smaller subarrays based on the buckets => In the worst case, the recursion depth is equal to the number of digits...

4) Sorting across the digits: Each pass processes n elements, and there are at most $O(\log_{10} M)$ passes (Max → maximum value from the array), In conclusion, the total time complexity is:

$$O(n \log M)$$

Space complexity analysis:
The space complexity depends on: Bucket storage, the recursive call stack and the

temporary arrays (in this case the buckets).

- Bucket Storage: we use to buckets (each storing at most n elements in worst-case scenario)
  ⇒ Space Complexity: $O(n)$.
- Recursive Call Stack: In the worst case the maximum depth of recursive calls is $O(\log n)$
- Temporary Arrays: each recursive step requires temporary storage for the numbers in buckets
  ⇒ Space Complexity is: $O(n)$

[redacted]

In conclusion ⇒ The total space complexity is: $O(n + \log M)$

$\log M$ is just a value that can be easily disregarded

⇒ Total time complexity: $O(n)$