Problem 8.1 Sorting in Linear Time

c) Algorithm (A, n, a, b)

```
k = max(A) + 1
C [k]    // Create a count array
// Initialize C array with zeros
for i = 0 to k do
    C [i] = 0
// Count the occurrences of each element in A
for i = 0 to n do
    C [A[i]] ++
// The cumulative count
for i = 1 to k do
    C [i] = C [i] + C [i-1]
// Count how many numbers fall into the interval [a, b]
return C [b] – C [a-1]
```

The pre-processing part is based on Counting Sort. The time complexity for the first loop is O(k), for the second is O(n) and for the third again O(k), so in overall the time complexity for the pre-processing part is O (n+k). The part of counting how many of the integers fall into the interval [a, b] includes just a subtraction, which takes constant time, so the time complexity is O(1).

e) The worst-case time complexity for Bucket Sort would happen if there were elements of close range in the array, so they would be placed in the same bucket. Then they would be sorted. Hence the time complexity would depend on the sorting algorithm. The algorithm used for sorting the elements usually is Insertion Sort, which has a time complexity of $O(n^2)$ in its average and worst case (the elements are in reverse order). So overall, the worst-case time complexity for Bucket Sort is $O(n^2)$.

Example: unsorted_array = {0.38, 0.36, 0.33, 0.31}

Proof: The 2 for loops, for inserting the elements in different buckets and for concatenating the elements of the buckets respectively, have a time complexity of O(n) and the for loop for sorting the elements has a time complexity of $O(n^2)$

$$T(n) = O(n) + O(n^2) + O(n)$$

$$\mathbf{= O(n^2)}$$

Problem 8.2 Radix Sort

b) Time Complexity:

Best case → This happens when the elements of the array are distributed evenly between the buckets and each bucket has only one element. This means the buckets are already sorted so the recursive step is not called. Hence the time complexity is **O(n)**.

Worst case → This happens when all the elements of the array are placed in just one bucket. The time depends on the recursive step and it is called d times (d represents the digits of the max element). Hence the time complexity is **O(dn)** where d can also be expressed as $\log_{base}(max)$.

Average case → This happens when the elements of the array are distributed into the buckets in such a way that half of the buckets have a lot of elements, so the recursive step is called d times. Meanwhile the other half of the buckets do not have elements or have just 1 element which means they are already sorted, so the recursive step is not performed at all. Hence the time complexity is O(d * n/2) + O(n) = **O(dn)**.

Space Complexity:

There is an array with n elements – O(n), there are also 10 buckets which is a constant number – O(c) where c is a constant, and the recursive call is performed d times where d is the number of digits of the max element. In recursive call, stack space counts and each call add a level to the stack and takes up actual memory – O(dn). Hence the space complexity is

$$O(n) + O(c) + O(dn) = \mathbf{O(dn)}$$

Space complexity is O(n) when the recursive call is performed only one time (which means max element has 1 digit, d=1) or when each bucket has only one element, so the recursive call is not performed at all → *this is the best case.*