Question 1: Radix Sort

1.1 Correctness

Lemma 1. *RadixSort will properly sort any n natural numbers.*

Proof. To prove Lemma 1, we will use induction on the number of digits that each value has, l. In the base case, for a list of single digits values, $l = 1$, assume that radix sort sorts the array based on the single digit and returns the now sorted list. So radix sort is able to sort a list of single digit values.

Now assume that radix sort correctly sorts $l - 1$ digits. Now all numbers are sorted up to their $(l - 1)$th digit, leaving the lth digit left to be sorted. And since we know that radix sort sorts one column at a time, without regard to other columns, sorting the lth digit is done in the same manner as before. The smaller digit is sorted to the left of the larger digit, also without regard for the other digits of the number. And thus it follows that Radix sort works for l digits. □

1.2 Runtime

Lemma 2. *RadixSort runs in $O(l(n + d))$ time, where d is the radix (the number of digits in the base) and l is the maximum length of the n numbers.*

To justify briefly this statement, we need to take a look at sorting individual positive numbers, for this we will use CountingSort which has a known complexity of $O(n + d)$, since we know that RadixSort sorts l digits independently of the others, this means that for RadixSort will pass over the entire list l times performing $(n + d)$ actions. And so it follows that the time complexity for Radix Sort is $O(l(n + d))$.

Question 2: Sort n integers from 0 to n^k

2.1 A

The time complexity for using Counting Sort would be $O(n^k)$ since the given range is from 0 to n^k numbers.

2.2 B

$$l = \lfloor log_d(n^k) \rfloor + 1$$

The value of l depends on the base, d, and the value, n^k. l is the number of digits in a number so for example if we have $n^k = 625$, in base 10, there are 3 individual digits and so $\lfloor log_{10}(625) \rfloor + 1$ should be 3, which it is, and it holds for all bases.

If we go from d to d^2, the value of l will always decrease.

2.3 C

If $d = 2$, then the runtime would be:

$$O((\lfloor log_2(n^k) \rfloor + 1) * (n + 2))$$

If we want to minimize runtime, then the best value of d would be where $d = n$, since $log_n(n) = 1$. So the time complexity would be:

$$O((\lfloor log_n(n^k) \rfloor + 1) * (n + n))$$
$$O((k + 1) * (2n))$$
$$O(n)$$

2.4 D

The time cost of converting our input from base 2 to base n would be the number of values, here that is n^k, multiplied by the time it takes to execute a change of base operation. Here I am assuming C(x) is a function that takes a base 2 number and returns a base n number. So it would follow that the time-complexity for preprocessing our input would be:

$$O(C(x)n^k)$$

Question 3: Time Complexity of Max Function

If we have a function $\max(x)$ where x is a form of list or array, then the number of times we must update the max is dependent on the location of the max within x. Suppose the max is the first element of x, then the number of updates we must make is only 1. This is our best case scenario, $O(1)$. Now suppose the max is at the end of x, then at worst we would need to update $O(n)$ times. For the average number of times we can expect to update the max variable, we take a look at the expected number of times we would need to iterate. We will assume that a number n has a probability of $\frac{1}{n}$ chance of being in each location. Let y_i be a random variable that denotes the position of the max within x and S be the sum of the possible positions of the max:

$$S = \textstyle\sum_{i=1}^{n} y_i$$

And so the expected value of S is $E(S) = E(\sum_{i=1}^{n} y_i) = \sum_{i=1}^{n}(E(y_i)$. Do to a uniform distribution assumption, we can expect the location of the max to occur anywhere in between the two extremes of either end. So hypothetically, $E(y_i) = \sum_{i=1}^{n}(\frac{i}{n})$. So we have:

$$E(S) = \textstyle\sum_{i=1}^{n}(\sum_{i=1}^{n}(\frac{i}{n})) = n(\sum_{i=1}^{n}(\frac{i}{n}))$$

This means that the average number of updates is actually $O(n)$ times.