Worked with: Jayden Personatt, Kelsey Chen

Problem 1

a. (Practice Using Asymptotic Notation)

f	g	O	o	Ω	ω	θ
$\sqrt{n}$	$\log n$	F	F	T	T	F
$n^{\sqrt{n}}$	$n^{\log n}$	F	F	T	T	F
$(\log n^{120})^2 \times \sqrt{n}$	n	T	T	F	F	F
$\log 2^n$	$\log e^n$	T	F	T	F	T
$2^{\sqrt{n}}$	$n^{\log n}$	F	F	T	T	F
$n^{n mod 2}$	$n^{\frac{1}{3}}$	F	F	F	F	F

b. (Rigorously Reasoning About Asymptotic Notation) For each of the following statements, I will justify why the statement holds or provide a counterexample.

- The statement - for all positive constants a and b, if $f(n) = \theta(a^n)$ and $g(n) = \theta(n^b)$, then $\theta(f(g(n)) = \theta(a^{n^b})$ - is false. A counter example is when $f(n) = 4 \times 2^n$ and $g(n) = 3 \times n^3$, which satisfies the conditions that $f(n) = \theta(a^n)$ and $g(n) = \theta(n^b)$. We re-write $f(n) = \theta(a^n)$ and $g(n) = \theta(n^b)$ for large n as follows:

$$a \times 2^n \leq 4 \times 2^n \leq b \times 2^n \tag{1}$$

$$c \times n^3 \leq 3 \times n^3 \leq d \times n^3 \tag{2}$$

where constants $a, b, c, d > 0$ and $a \leq 4$, $4 \leq b$, $c \leq 3$, $3 \leq d$.

$f(g(n)) = 4 \times 2^{3n^3}$, which can be rewritten as $f(g(n)) = 4 \times (2^{n^3})^3$. This negates the claim that $\theta(f(g(n)) = \theta(a^{n^b})$ because in our example, $\theta(f(g(n))$ is not upper bounded by 2^{n^3} multiplied by a constant because it can grow to $(2^{n^3})^3$ multiplied by a constant, which is much larger.

- The statement - for all positive constants a and b, if $f(n) = \theta(a^n)$ and $g(n) = \theta(n^b)$, then $\theta(g(f(n)) = \theta(a^n)^b)$ - is true. We first define $f(n) = \theta(a^n)$ and $g(n) = \theta(n^b)$ for large n as follows:

$$c \times a^n \leq f(n) \leq d \times a^n \tag{3}$$

$$s \times n^b \leq g(n) \leq t \times n^b \tag{4}$$

where $a, b, c, d > 0$. Therefore, the composition $g(f(n))$ will have an equality as follows:

$$s \times (c \times a^n)^b \leq g(f(n)) \leq t \times (d \times a^n)^b \tag{5}$$

where $s, t > 0$. This inequality eventually simplifies to:

$$i \times (a^n)^b \leq g(f(n)) \leq j \times (a^n)^b \tag{6}$$

where $i = s \times c^b$ and $j = t \times d^b$. We can simplify it down to equation (6) because $s \times c^b$ and $t \times d^b$ are constants and because n is growing infinitely large in our analysis, we care less about the constants and more about n. (6) is the same as the statement $\theta(g(f(n)) = \theta((a^n)^b)$ and therefore, we can say that $\theta(g(f(n)) = \theta((a^n)^b)$.

Problem 2

a. **Input** (n, b, k), the set of 3 natural numbers where n represents the number being converted, b

represents the base to which n will be converted in, and k represents the number of places/digits the base conversion provides (ie. the number of iterations we specify to convert n into base b). A digit in this definition delineates a base coefficient.
Output An array A of natural numbers that represents the value of n in base b given k places. That is A will be converted into k parts where each element in the array is a specific digit in the base such that n can be represented as a sequence of each element multiplied and weighted by its base and degree. However, if the natural number n cannot be converted into base b within k iterations, the computational problem II returns $\perp$.

b. By definition $f(x)$ is the set of valid outputs for the computation problem II. Therefore, $|f(x)| = 1$ or $|f(x)| = 0$. For the former, because there is only one way to express every unique natural number in a certain base, there is only one possible valid output. However, if $b < 2$ and/or if the program cannot compute the base conversion within k iterations, the set of valid outputs for the computation problem is empty. Thus, $|f(x)| = 1$ or $|f(x)| = 0$.

c.
- No. It does not follows that every algorithm A that solves II also solves II'. We prove this by providing a counterexample. Say $\mathcal{I} = \mathbb{Z}$ and $\mathcal{O} = \mathbb{Z}$. Let $f(x) = \{y : 2^y = x\}$ and $f'(x) = \{y : y = x\}$. Therefore, $\forall x$ in $\mathcal{I}$, $f(x) \subseteq f'(x)$. Let's say Algorithm A solves II. By **Definiton 4.2** in Section Notes, we know that there is a special output $\perp \notin \mathcal{O}$ such that for every input with $x \in \mathcal{I}$ with $f(x) = \emptyset$, we have $A(x) = \perp$. In our specific counterexample, if $x \leq 0$, $f(x) = \emptyset$ such that the Algorithm A correctly returns $\perp$ for II, but incorrectly returns $\perp$ for II' because $f'(x)$ should express a valid output.
- Yes. Let's say Algorithm A solves II. By **Definiton 4.2** in Section Notes, for every input $x \subseteq \mathcal{I}$ with $f(x) \neq \emptyset$ for every x, we have $A(x) \in f(x)$. Using this definition and the fact that for all $x \in \mathcal{I}$ we have $f(x) \subseteq f'(x)$, we know that every algorithm A that solves II also solves II' because every algorithm A that solves II is an element of $f(x)$ and $f(x)$ is a subset of $f'(x)$ so every algorithm A is also an element of $f'(x)$.

Problem 3

a. We prove the correctness of RadixSort by induction.

Proof. For every $i < n$, K_i' is a base-digit of the base-converted key K_i. Digit denominates c_i in $\{c_0, c_1, ..., c_{k-1}\}$ from the algorithm $BC(K_i, b, k)$. Assignments for K_i' are based on V_i', which is an array of digits from the base b conversion of K_i. Entries in V_i' are listed/indexed from least significant digit to most significant digit (based on b^{k-1}). Therefore, when iterating upon each digit in *RadixSort*, we begin by sorting the least significant digits of each key, gradually moving on to sorting more significant digits.

We know that the value K_i' can be no greater than the base b by the definition of base conversion. So the call CountingSort$(b, ((K'_0, (V_0, V'_0)), ..., (K'_{n-1}, (V_{n-1}, V'_{n-1}))))$ has indices in its counting array that properly account for all values of K_i'. CountingSort sorts the base-converted digits of each key at kth place. Therefore, every key in the original array $A = ((K_0, V_0), ...(K_{n-1}, V_{n-1}))$ will be represented in $((K_0', (V_0, V_0')), ..., (K_{n-1}', (V_{n-1}, V_{n-1}')))$ in the call to CountingSort (the representation will of K will be differrent though because K_i' is the kth digit of K_i).

We've established that RadixSort's call to CountingSort accounts for all digit entries of keys and that each iteration of CountingSort starts from the least-significant digit and ends with the most significant digit. We now prove by induction that on each iteration of CountingSort, the array $((K'_0, (V_0, V'_0)), ..., (K'_{n-1}, (V_{n-1}, V'_{n-1})))$ is sorted for both discrete K_i's and identical K_i's.

Definition 0.1 (P(n))**.** We define the predicate $P(n)$ to be the nth iteration of CountingSort such that $((K'_0, (V_0, V'_0)), ..., (K'_{n-1}, (V_{n-1}, V'_{n-1})))$ is sorted for both discrete K'_is and identical K'_is.

Base Case 0.1. $P(0)$ is true because the values of K'_i are sorted with respect to their first base-conversion digit (least significant digit). The call to CountingSort during the 0th iteration takes in the original input array A. Thus, K'_is with identical first base-conversion digits are sorted correctly because they sequentially follow the original ordering of A. K'_is with discrete first base-conversion digits are also sorted correctly as CountingSort correctly sorts over those values.

Inductive Hypothesis 0.1. Assume that the statement $P(n)$ is true.

Inductive Step 0.1. To show that $P(n+1)$ is true, we show that the $(n+1)$th iteration of CountingSort correctly sorts the array $((K'_0, (V_0, V'_0)), ..., (K'_{n-1}, (V_{n-1}, V'_{n-1})))$ for both discrete K'_is and identical K'_is while preserving prior iterations of digit sortings. Because CountingSort is a stable sort, each prior iteration's re-ordering will be accounted for because the order of the prior iterations will be preserved even when the key values in the current iteration are equal. In other words, if array B denotes the order of the keys after the nth iteration, then the $(n+1)$th iteration of CountingSort takes in the ordering of B so that any identical entries of K'_is utilize and preserve B's original ordering. By the Inductive Hypothesis, since we assume that $P(n)$ is true, we can assume that array B's orderings for prior base coefficients are properly sorted. Thus, during the $(n+1)$th iteration of CountingSort, $((K'_0, (V_0, V'_0)), ..., (K'_{n-1}, (V_{n-1}, V'_{n-1})))$ will be properly sorted according to its base coefficients indexed at $(n+1)$ for both discrete K'_is and identical K'_is because prior iterations of sorts are preserved.

We have therefore proven by induction that $P(n)$ holds true for all $n \geq 0$. Because each iteration of CountingSort builds off of prior iterations, the last iteration is weighted more than the prior ones. This is consistent with the fact that the the index starts from the least significant digit and ends with the most significant digit, so the most significant digit will be iterated last and weighted more. However, if the keys in the last iteration are identical, the ordering between the two will be determined by prior iteration orderings.

Therefore, when RadixSort has iterated through all digits of the key in the inputted array of keys, we return an array $((K'_0, (V_0, V'_0)), ..., (K'_{n-1}, (V_{n-1}, V'_{n-1})))$ that is sorted correctly by keys. RadixSort then assimilates all digits K'_i of K_i to return an array of the full representation of K_i with its associated values sorted correctly. □

b. We show that RadixSort has a run time of $\mathcal{O}((n+b) \times (\log_b U))$ line by line.
2 Line 2 has a run time of $\mathcal{O}(1)$ because it is considered a basic operation (mathematical calculation).
3-4 Line 3-4 describes a for loop that iterates n times performing the BC algorithm. When we analyze the BC algorithm, we find that it takes $\mathcal{O}(k)$ time because each call iterates k times. Therefore, Line 3-4 has a run time of $\mathcal{O}(n \times k)$
5 Line 5 describes a for loop that iterates k times. The for loop encompasses code up till Line 8. Therefore, code within Line 6-8 will have a run time multiplied by k.
6-7 Lines 6-7 describes a for loop that iterates n times and performs a basic operation (array assignment). Therefore, its run time is $\mathcal{O}(n)$.
8 Line 8 describes CountingSort which has a runtime of $\mathcal{O}(n+b)$.
5-8 Because Lines 6-8 are within the for loop in Line 5, each of their run times should be multiplied by k, which is the number of iterations the for loop in Line 5 specifies. Thus, Lines 5-8 have a run time of $\mathcal{O}(k \times (n+n+b) = \mathcal{O}(k \times (2n+b))$.
9-10 Lines 9-10 showcase a for loop that iterates n times. The for loop encompasses Line 10. Assignment takes $\mathcal{O}(1)$ time but we assigned it k times. Thus Lines 9-10 have a run time of $\mathcal{O}(kn)$.
Adding all these lines together, we find the RadixSort has a run time of $\mathcal{O}(1 + (n \times k) + (k \times (2n +$

$b)) + kn)$ which can be simplified to $\mathcal{O}(1 + (k \times (4n + b)))$. Because Big O Notation specifies the worst case scenario (when n is an extremely large number), the coefficients and entries that do not include n become insignificant, so we can simplify this equation to the highest terms. Therefore, we get $\mathcal{O}(k \times (n + b))$. Per Line 2, we find that $k = \log U / \log b = \log_b U$. Thus, we rewrite the run time of RadixSort to be:

$$\mathcal{O}(\log_b U \times (n + b)) \tag{7}$$

Let's set $b = \min(n, U)$. Our run time would therefore be $\mathcal{O}(n + n \log U / \log n)$. We show this by cases: the first is when $b = n$ and the second is when $b = U$

Case 0.1.

$$\mathcal{O}(\log_U U \times (n + U)) = \mathcal{O}(n + U) = \mathcal{O}(n) \tag{8}$$

Note: we can simplify $\mathcal{O}(n + U) = \mathcal{O}(n)$ because $n > U$. $\mathcal{O}(n)$ is in $\mathcal{O}(n + n \log U / \log n)$.

Case 0.2.

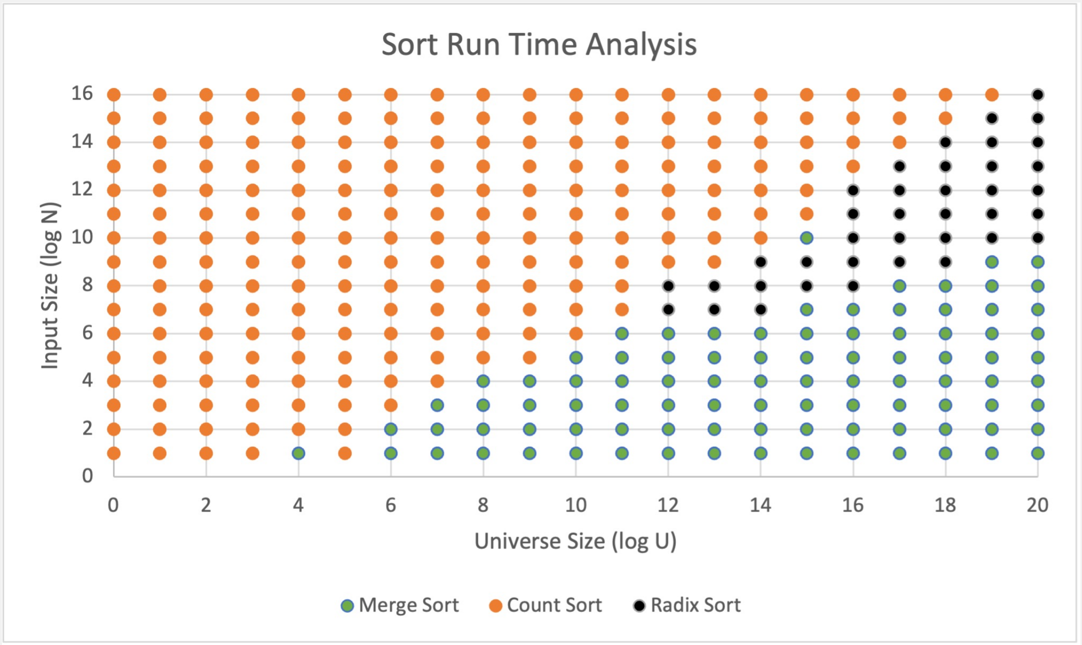
$$\mathcal{O}(\log_n U \times (n + n)) = \mathcal{O}(\log_n U \times n) = \mathcal{O}(n(\log U / \log n)) \tag{9}$$

$\mathcal{O}(n(\log U / \log n))$ is in $\mathcal{O}(n + n \log U / \log n)$ because the latter equation can be rewritten as $\mathcal{O}(n \times (1 + \log U / \log n))$ and since $\log U / \log n$ is less than $1 + \log U / \log n$ we can say that $\mathcal{O}(n(\log U / \log n))$ is in $\mathcal{O}(n + n \log U / \log n)$.

Therefore, when $b = \min(n, U)$, the run time for RadixSort is $\mathcal{O}(n + n \log U / \log n)$

c. (Implementing Algorithms)

d. (Experimentally Evaluating Algorithms) The below graph was compiled after after running 3 different trials on pairs of (n, U) where the keys chosen for the input array were randomly, uniformly, and independently determined from $[U]$.



The shapes of the transition graph below do fit what I'd expect from asymptotic theory. For inputs of size n that are roughly larger or equal to U, CountingSort will have the best run time because it runs $\mathcal{O}(n)$. This is because CountingSort runs $\mathcal{O}(n + b)$ where b is set to U because $n \geq U$. And, because U is less than n, $\mathcal{O}(n + U) = \mathcal{O}(n)$. This is considerably more efficient than

MergeSort's run time of $\mathcal{O}(n(\log n))$ and RadixSort's run time of $\mathcal{O}(n + n \log U/\log n)$. This is shown on the transition graph as the upper-left half is dominated by CountingSort because values of $\log n$ are greater than $\log U$. However, if U is considerably larger than n, CountingSort will have a runtime of $\mathcal{O}(n+U)$, which can be considered $\mathcal{O}(U)$ because U is considerably greater than n. On the other hand, MergeSort's run time will be $\mathcal{O}(n(\log n))$ and RadixSort's run time will be $\mathcal{O}(n + n \log U/\log n)$. Therefore, MergeSort outperforms CountingSort as well as RadixSort because the U in both RadixSort and CountingSort considerably slows both down. This is less so for RadixSort because we take $\log U$ which is why we see that for similar values of n and U, RadixSort can actually be the most efficient algorithm because it weighs changes in U and n more equally than MergeSort and CountingSort.

Thus, we find that when we start from the top left of the graph, CountingSort will in general perform best out of all 3 sorts. As we move further right and downwards slightly, we find that RadixSort becomes the most efficient algorithm of the 3 sorts, especially when n is slightly less than U. Lastly, when we move even more downwards, we find that MergeSort because the most efficient algorithm of the 3 sorts, especially when $U \approx n^2$ and above because $\mathcal{O}(n^2)$ encompasses $\mathcal{O}(n \log(n))$.