Question 1

Given: *blackBox(set S)* - returns true or false depending on if there exists x, y s.t. $xy = z$

Approach: We wish to create a set S' s.t. $x'y' = z' \in S'$ iff there exists a, b, and $c \in S$ s.t. $xy + z = 0$. If we use the black box on set S', we can show that there exists $xy + z = 0$ in the original set S.

Let S' be the negative set of S, i.e multiply every element in S by -1.

Prove: $x'y' = z' \implies xy + z = 0$

$$x' = -x \quad y' = -y \quad z' = -z$$

$$\begin{aligned} x'y' &= z' \\ (-1)(x) * (-1)(y) &= (-1)(z) \\ x * y &= -z \\ x * y + z &= 0 \end{aligned}$$

■

Prove: $x'y' = z' \impliedby xy + z = 0$

$$x = -x' \quad y = -y' \quad z = -z'$$

$$\begin{aligned} xy + z &= 0 \\ (-1)(x') * (-1)(y') + (-1)(z') &= 0 \\ x' * y' - z' &= 0 \\ x' * y' &= z' \end{aligned}$$

■

Therefore, $x'y' = z'$ implies $xy + z = 0$ and vice versa.

Q. E. D.

If *blackbox(S')* returns true, then it follows that $xy + z = 0$.

Algorithm 1: Product checking

```
1 Z-CHECK(S, z):
2     for i in len(S) do
3         S'[i] = (-1) * S[i]
4     return BLACKBOX(S')
5     end
```

Time Complexity: Blackbox(S') is O(1), and iterating through the for loop takes O(n). This function takes O(n) total.

Question 2

Approach: Since we know that the black box will output z which is contained in set S we can do the following:

1. Use mergesort or any sorting algorithm of O(nlogn) complexity to sort S in incremental order.
2. Iterate through the ordered list by indexing the starting and ending points of S to give the maximum and minimum values in S with a loop.
3. Use the values at the left and right indices and check if their product is equal to z. If the product is smaller, increment the left index. If the product is too large, decrement the right index. This continues until the left and right index meet each other.

Algorithm 2: Finding x and y from product.

```
XY-FINDER(S, z):
    l, r = 0, len(S) // index of start and end of list
    MERGESORT(S, l, r - 1)
    while r > l do
        if S[l] * S[r] == z then
            return S[l], S[r]
        if S[l] * S[r] > z then
            r = r − 1
        else if S[l] * S[r] < z then
            l = l + 1
    end
```

Time Complexity: Mergesort has time complexity O(nlogn) and the while loop that we iterate through has time complexity O(n). Therefore when we add the time complexities, we still have O(nlogn).

Question 3

Approach: We will use a sorting algorithm similar to bubble sort. However, instead of iterating through the entire array, we will only check three elements ahead for each key. This will give us $O(3n)$ as opposed to $O(n^2)$.

Background: Since bubble sort will only swap elements if they are in the wrong order (i.e if $A[0] = 5$ and $A[1] = 3$), we only need to consider the cases where $A[i]$ is bigger than $A[k]$, where $k > i$, and $i = [0, 1, ..., len(A)]$. Since the original array has n *distinct* sorted integers, we only need to compare up to $A[i+3]$. The minimum value of $A[i+4]$ will be equal to the maximum value of $A[i]$, bubble sort will never need to swap these elements.

$$\{A[i], A[i]+1, A[i]+2, A[i]+3, A[i]+4\}$$

Proof: Consider the above array. Assume that at any element $A[i]$, bubble sort must traverse more than 3 elements to find a smaller key (i.e. $A[k] < A[i]$ where $k \geq i+4$). We know that the minimum value of $A[i+m]$ is $A[i]+m$, since all keys are distinct integer values. Therefore, the minimum value of $A[k]$ is $A[i+4] = A[i]+4$.

Now assume each element in the array will change by +2/-2/+0. The new maximum value of $A[i]$ is $A[i]+2$. The minimum value of $A[i+4]$ is $(A[i]+4)-2$, which is $A[i]+2$. This contradicts are earlier assumption that $A[k] < A[i]$, where $k \geq i+4$. Therefore, only elements less than $A[i+3]$ can be smaller than $A[i]$, so we only need to compare up to three elements for each node in the array.

Q. E. D.

Algorithm 3: Resorting.

```
1  SORT-ARR(A):
2      for i in len(A) do
3          if A[i] > A[i+1] then
4              swap A[i], A[i+1]
5          if A[i] > A[i+2] then
6              swap A[i], A[i+2]
7          if A[i] > A[i+3] then
8              swap A[i], A[i+3]
9      end
10     return A
```

Time Complexity: We know that the for loop will iterate through every element in the array, which is $O(n)$. Within the for loop, we iterate over the next three elements. In total, the time is $O(3n)$, which is $O(n)$ complexity.

Question 4

Given a sorted sequence of distinct integers, $A[1 \dots n]$, we can define the algorithm below to determine whether an index i exists such that $A[i] = i$:

Approach. The methodology behind the algorithm is to check if $A[i] = i$ is true for the middle element in the list. And if it isn't repeat the same process on either the upper or lower half of the list depending on whether $A[i] < i$ or $A[i] > i$.

Algorithm 4: Index value matching

```
1  MATCH-CHECK(A, p, q):
2      i = floor((q+p)/2)                    // get middle index
3      if A[i] == i then
4          return True
5      else if p == q then
6          return False
7      else if A[i] > i then
8          return MATCH-CHECK(A, p, i -1)
9      else
10         return MATCH-CHECK(A, i+1, q)
```

Let A, p and q be the inputs to the algorithms, where A is a sorted sequence of distinct integers with n elements, $[p, q]$ is the range of indexes that the algorithm operates on. Also, let m be the input size, i.e. length of the range $[p, q]$.

Proof of correctness and termination. We will prove that given the preconditions, the execution of the algorithm will result in termination and the postcondition by induction on input size m.

Precondition.

- A is an array of strictly increasing integers
- p and q are valid array indexes into A.
- $p <= q$

Postcondition. If $\exists\ i \in [p, q],\ A[i] = i$ then the output is $True$ otherwise $False$

Basis. $m = 1$. Now we have $p = q$ and either $A[p] = p$ (1) or $A[p] \neq p$ (2). If (1) the algorithm will execute line 4, terminating and outputting $True$. Otherwise (2), then the algorithm will execute line 6, terminating and outputting $False$.

Hypothesis. Assume the algorithm terminates and is correct for all input sizes k where $1 \leq k < m$.

Inductive Step. Prove the algorithm terminates and is correct for m.

Line 2 executes and i is the middle value (odd m) or lower of two middle values (even m), there are three cases:

- Case 1: where $A[i] = i$, the algorithm executes line 4 and terminates.
- Case 2: where $A[i] > i$ In this case, by precondition 1 it is not possible for $A[j] = j$ for $j > i$, therefore the algorithm executes line 8 and calls itself on $A[p \dots i-1]$. By the hypothesis (since new input size is less than m), it terminates and outputs correctly.
- Case 3: where the conditions for cases 1 and 2 are not met, therefore $A[i] < i$ In this case, by precondition 1 it is not possible for $A[j] = j$ for $j < i$, therefore the algorithm executes line 10 and calls itself on $A[i+1 \dots q]$. By the hypothesis (since new input size is less than m) , it terminates and outputs correctly.

Q. E. D.

Time and Space Complexity. Since every recursive call operates on exactly half of the input size, the function can be represented by $T(n) = T(n/2)$ which has a time complexity of $O(log(n))$. Since the algorithm operates in place and does not transform the input list, the space complexity is $O(n)$

Question 5

Part A

Counting Sort: See textbook page 196 for the following calculations.

$$\begin{aligned} k &= 9,999,999,999,999,999 \\ n &= 100000 \\ \Theta(k+n) &= \Theta(10,000,000,000,099,999) \end{aligned}$$

Radix Sort: See textbook page 198, Lemma 8.3 for the following calculations.

$$\begin{aligned} k &= 10 \\ n &= 100000 \\ d &= 16 \\ \Theta(d(k+n)) &= \Theta(16(100,000+10)) \\ &= \Theta(1,600,160) \end{aligned}$$

Therefore, radix sort is better.

Part B

Counting Sort: See textbook page 196 for the following calculations.

$$\begin{aligned} n &= 4 * 10^9 \\ k &= 2^{32} = O(n) \\ \Theta(n) &= \Theta(4 * 10^9) \end{aligned}$$

Radix Sort: See textbook page 199, Lemma 8.4 for the following calculations.

$$\begin{aligned} n &= 4 * 10^9 \\ b &= 32 \\ r &= lg(n) = 32 \\ \Theta(\frac{b}{r}(n+2^r) &= \Theta(\frac{32}{32}(4 * 10^9 + 2^{32}) \\ &= \Theta(4 * 10^9 + 2^{32}) \end{aligned}$$

Therefore, counting sort is better.