1.

(a) Program would run forever when we put $z = \{3, 2, 3, 2 \cdots\}$.

if $z=3$, $x_1 = x_0 - 4$ and $y_1 = y_0 + 5$

if $z=2$, $x_2 = x_1 + 4 = x_0 - 4 + 4 = x_0$ and

$y_2 = y_1 - 5 = y_0 + 5 - 5 = y_0$

So, if inputs for 'z' alternating between odd and even,
the program runs forever.

(b) The program would halt for all possible sequences

Since the program runs while $x>0$ and $y>0$. If z is odd $y_{i+1} = y_i - 2$ and when z is even, $x_{i+1} = x_i - 3$, $y_{i+1} = y_i + 5$. We can see that x is always decreasing or stays the same no matter what z value is ($z>0$)

Therefore, it would execute the while loop

2. Let's say that

Euclid $(x_k, y_k) = (x_{k+1}, y_{k+1})$.

$x_{k+1} = y_k$

$y_{k+1} = x_k \% y_k = r_k$ $(x_k \bmod y_k)$

potential function

$S_k = cx_k + y_k$

$= c \cdot (q_k y_k + r_k) + y_k$ (where $q_k = [x_k / y_k]$)

$= (cq_k + 1) y_k + cr_k$

$\geq (c+1) y_k + cr_k$ (since $q_k \geq 1$)

$\rightarrow S_k \geq (c+1) y_k + cr_k = c^2 y_k + cr_k$. $(c+1 = c^2)$

and we know that $S_{k+1} = cx_{k+1} + y_{k+1} = cy_k + r_k$,

we found that $S_k \geq c S_{k+1}$. for all $(x_k \geq y_k \geq 1)$

Therefore, each step reduces the potential by a factor of at least ϕ,
the number of iterations made by Euclid's algorithm is at most $\log_\phi(x+y)$

3.

$(a_{k+1} + b_{k+1} i)(c + di) = a_k + b_k i$

So, $a_{k+1} c - b_{k+1} d = a_k$

$b_{k+1} c + a_{k+1} d = b_k$

Using potential function $\Phi(a+bi) = a^2 + b^2$,

$\Phi(a_k + b_k i) = (a_{k+1} c - b_{k+1} d)^2 + (b_{k+1} c + a_{k+1} d)^2$

$= (a_{k+1} c)^2 + (b_{k+1} d)^2 + (b_{k+1} c)^2 + (a_{k+1} d)^2$

$= a_{k+1}^2 (c^2 + d^2) + b_{k+1}^2 (c^2 + d^2) = (a_{k+1}^2 + b_{k+1}^2)(c^2 + d^2)$

Therefore, $\Phi(a_k + b_k i) = a_k^2 + b_k^2 = (a_{k+1}^2 + b_{k+1}^2)(c^2 + d^2)$

$\Phi(a_{k+1} + b_{k+1} i) = a_{k+1}^2 + b_{k+1}^2$

$\therefore \Phi(a_{k+1} + b_{k+1} i) < \Phi(a_k + b_k i)$

4.

For n=1, output is either 0 or 1

For larger values of n, we can divide the input into two equal-sized blocks of size n/2.

We'll recursively compute the Hamming weight of each block then use the circuit for adding two k-bit numbers to merge the result

$T(1) = O(1)$

$T(n) = 2T(n/2) + O(n^{\frac{1}{2}})$

5.

```
function CountInversions(arr)
    if length(arr) <= 1 then
        return arr, 0
    else
        mid = length(arr) / 2
        left_half, left_inversions = CountInversions(arr[1:mid])
        right_half, right_inversions = CountInversions(arr[mid+1:end])
        merged_arr, split_inversions = MergeAndCountInversions(left_half, right_half)
        total_inversions = left_inversions + right_inversions + split_inversions
        return merged_arr, total_inversions

function MergeAndCountInversions(left, right)
    merged = []
    split_inversions = 0
    i = j = 0
    while i < length(left) and j < length(right) do
        if left[i] <= right[j] then
            merged.append(left[i])
            i = i + 1
        else
            merged.append(right[j])
            j = j + 1
            split_inversions = split_inversions + (length(left) - i)
    merged.extend(left[i:])
    merged.extend(right[j:])
    return merged, split_inversions
```

Base case

The algorithm divides the array into two halves and recursively counts inversions in each half

if an element from the right sub array is smaller than an element from the left subarray, it means there's an inversion. The algorithm correctly counts this inversion by adding '(length(left)-i)' to 'split_inversion'

The array is divided into two halves, each with approximately n/2 elements.
The divide step takes O(1) time, and the conquer step involves recursively counting inversions in the left and right halves. Therefore, the time complexity of this part is $T(n)=2T(n/2)$, which is $O(n)$
The merging step takes linear time $O(n)$, as we iterate through both subarrays once.
This step is executed for each level of recursion, so it contributes $O(n \log n)$ to the overall time complexity.
Overall, the time complexity of the algorithm is dominated by the merging step, resulting in a total complexity of $O(n \log n)$