A variation of the bit strings problem from Wednesday.

Let $T(n)$ be the number of bit strings of length $n$ that DO NOT contain two consecutive zeros.

- Base cases: $T(1) = 2$, $T(2) = 3$.
- Determine a linear recurrence relation for $T(n)$ in terms of $T(n-1)$, $T(n-2)$.

Let $b_n$ be a bit string of length $n$ without consecutive zeros. Each $b_n$ either ends in 0 or 1.

Note: we can create $b_n$ from $b_{n-1}$, appending either 0 or 1 (without creating consecutive zeros).

$$b_n \text{ ends with } 1: \quad b_{n-1} \{1\} \quad \# \text{ bit strings without consecutive zeros} \quad T(n-1)$$

$$b_n \text{ ends with } 0: \quad b_{n-2} \{01, 10\} \quad (00 \text{ not included in } T(n-1))$$

$$T(n) = T(n-1) + T(n-2)$$
Revisiting the bit strings problem from Wednesday.

Let \( T(n) \) be the number of bit strings of length \( n \) that have two consecutive zeros. Consider a recurrence relation for \( T(n) \).

- Base cases: \( T(1) = 0, T(2) = 1 \).
- Determine a linear recurrence relation for \( T(n) \) in terms of \( T(n - 1), T(n - 2) \) and possibly \( n \).

Let \( b_n \) be a bit string of length \( n \) with two consecutive zeros. Each \( b_n \) either ends in 0 or 1.

\[
\begin{align*}
\text{\( b_n \) ends with 00:} & \quad 2^{n-2} \\
\text{\( b_n \) ends with 10:} & \quad T(n-2) \\
\text{\( b_n \) ends with 11:} & \quad 2 \text{ ways to create } b_{n-1} T(n-1)
\end{align*}
\]

\[
T(n) = T(n-1) + T(n-2) + 2^{n-2}
\]
Let's analyze the work done by merge-sort.

```python
1 def merge_sort(a: list) -> list:
2     n = len(a)
3     if n <= 1:  # base case: 1-item/empty list is already sorted
4         return a
5
6     # divide stage: sort two sublists
7     m = n // 2
8     a1 = merge_sort(a[:m])  # sort left sublist
9     a2 = merge_sort(a[m:])  # sort right sublist
10
11     # conquer stage: merge the sorted sublists
12     merged = []
13     while len(a1) != 0 or len(a2) != 0:
14         if len(a1) == 0:
15             merged.append(a2.pop(0))
16         elif len(a2) == 0:
17             merged.append(a1.pop(0))
18         else:
19             if a1[0] < a2[0]:  # this is the comparison to count
20                 merged.append(a1.pop(0))
21             else:
22                 merged.append(a2.pop(0))
23     return merged
```

2 subproblems each problem is $\frac{1}{2}$ original size.
How many times do you need to ask "which leading element is smallest?" when merging these two subarrays?

For merge sort (arbitrary n)

$$n - 1$$

In general, work done outside of recursive calls

$$n^d$$

$$d = 1$$
Adding up all the work done.

Questions we need to answer:
1. How many times is the original length-$n$ array broken up until we get to subarrays of length 1?
2. How much total work (operations) are done during the recursive step (i.e. to merge)?
The Tree method for divide & conquer recurrences.

general recurrence: \( f(n) = a \cdot f\left(\frac{n}{b}\right) + c \cdot n^d \)

\[
\sum_{k=0}^{\log_b n} a^k \left(\frac{n}{b^k}\right)^d = \sum_{k=0}^{\log_b n} \frac{n^d}{b^{kd}} = n^d \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k
\]

\[
f(n) = \begin{cases} 
O(n^d) & \text{if } a < b^d \\
O(n^d \log n) & \text{if } a = b^d \\
O(n^{\log_b a}) & \text{if } a > b^d 
\end{cases}
\]

\[
f(n) = 2f\left(\frac{n}{2}\right) + n-1
\]

[O(n \log n)]

a = 2
b = 2
d = 1

z = 2!
Exercise: use the Tree method to determine the complexity of binary search.

```python
1  def binary_search(a, x):
2       n = len(a)
3       if n == 0:
4           return False
5       elif n == 1:
6           if a[0] == value:
7               return True
8           return False
9
10      m = n // 2
11      if a[m] <= value:
12          return binary_search(a[m:], x)
13      else:
14          return binary_search(a[:m], x)
```

\[ f(n) = \begin{cases} 
O(n^d) & \text{if } a < b^d \\
O(n^d \log n) & \text{if } a = b^d \\
O(n^{\log_b a}) & \text{if } a > b^d 
\end{cases} \]