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).

$$\overbrace{\underbrace{\_\ \_\ \_\ \_\ \_}_{b_{n-1}}\ \_}^{b_n}$$

→ either 0 or 1 (2 cases)

<u>b_n ends with 1:</u> $b_{n-1}\{1\}$ # bit strings without consecutive zeros $T(n-1)$

<u>b_n ends with 0:</u> $b_{n-2}\{\_\ \_\}$

0 0 ✗
1 0 → $T(n-2)$
0 1
1 1] included in $T(n-1)$

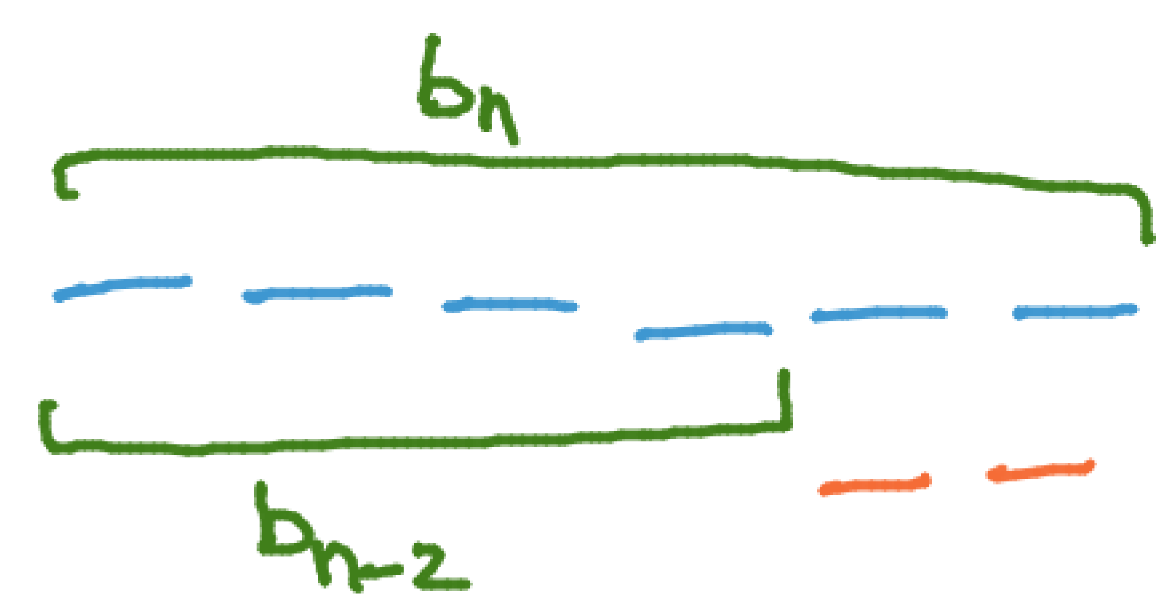
$$\boxed{T(n) = T(n-1) + T(n-2)}$$

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.



00
10
01
11

b_n ends with 00: 2^{n-2}

b_n ends with 10: $T(n-2)$

b_n ends with 01, 11: # ways to create b_{n-1} $T(n-1)$

$$\boxed{T(n) = T(n-1) + T(n-2) + 2^{n-2}}$$

Let's analyze the work done by `merge-sort`.

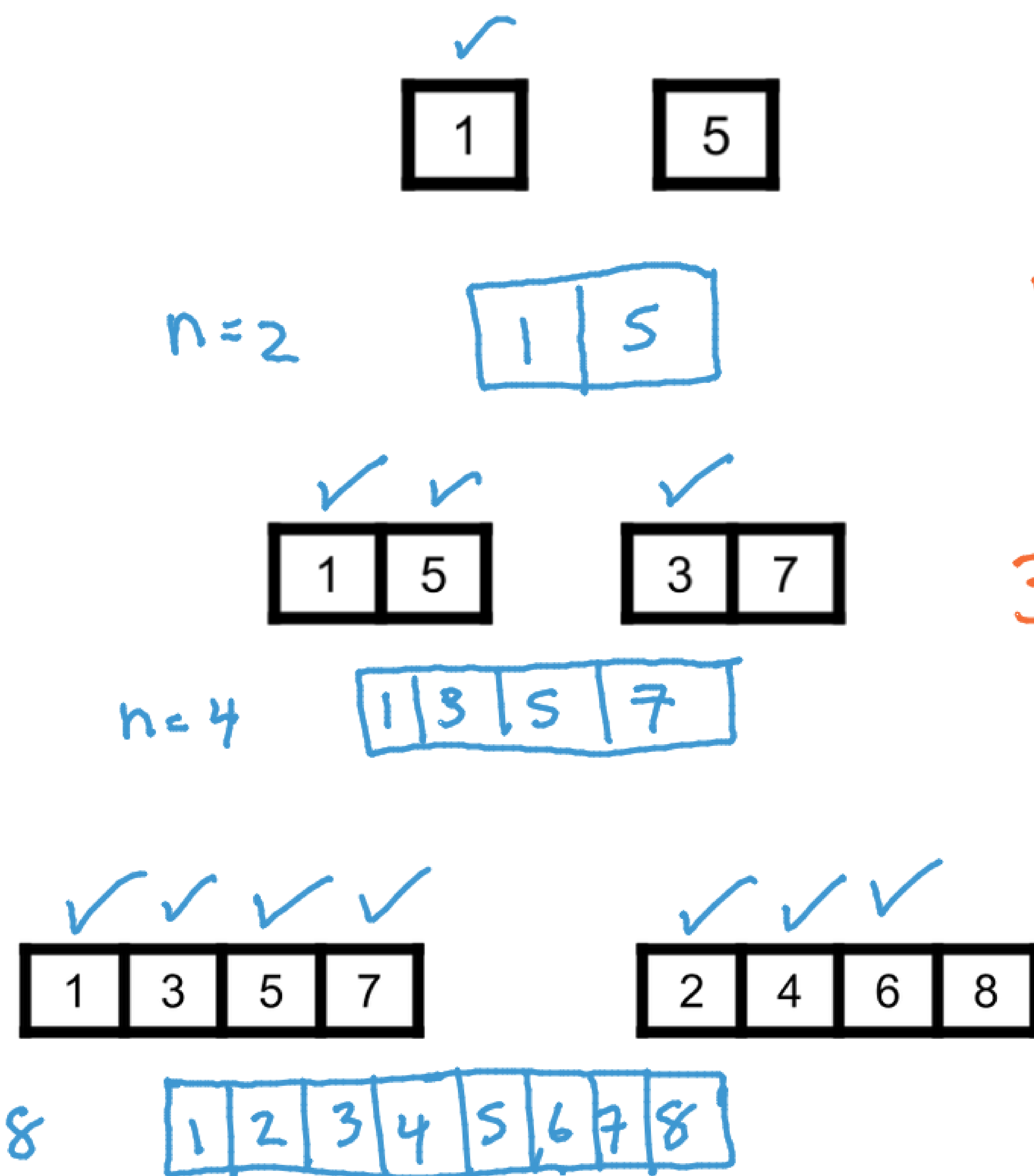
```python
def merge_sort(a: list) -> list:
    n = len(a)
    if n <= 1: # base case: 1-item/empty list is already sorted
      return a

    # divide stage: sort two sublists
    m = n // 2
    a1 = merge_sort(a[:m]) # sort left sublist
    a2 = merge_sort(a[m:]) # sort right sublist

    # conquer stage: merge the sorted sublists
    merged = []
    while len(a1) != 0 or len(a2) != 0:
      if len(a1) == 0:
        merged.append(a2.pop(0))
      elif len(a2) == 0:
        merged.append(a1.pop(0))
      else:
        if a1[0] < a2[0]: # this is the comparison to count
          merged.append(a1.pop(0))
        else:
          merged.append(a2.pop(0))
    return merged
```

2 subproblems
each problem size
is $\frac{1}{2}$ original size

How many times do you need to ask "which leading element is smallest?" when merging these two subarrays?



$1 < 5$

$n = 2$

1

$1 < 3$
$3 < 5$
$5 < 7$

$n = 4$

3

$1 < 2$
$2 < 3$
$3 < 4$
$4 < 5$
$5 < 6$
$6 < 7$
$7 < 8$

$n = 8$

7

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*)?

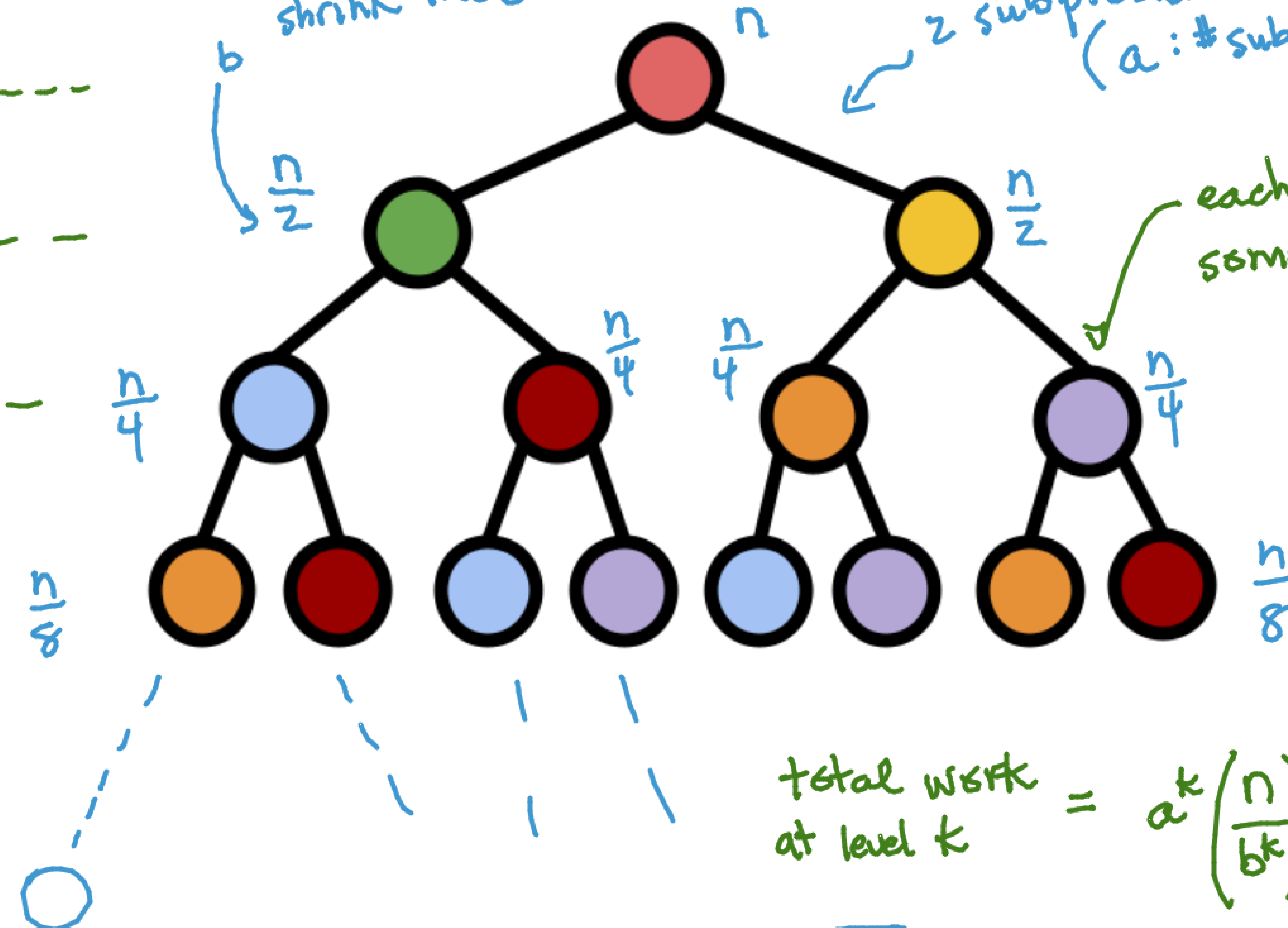
work at each vertex

n^d

$a = 2$

2 subproblems (a : # subproblems)

shrink factor b



$k = 0$

$k = 1$

$k = 2$

$k = 3$

final level

each vertex is doing some work

how many vertices at each level?

$2^k = a^k$

work at a vertex

$\left(\frac{n}{b^k}\right)^d$

total work at level k $= a^k \left(\frac{n}{b^k}\right)^d$

problem size of 1 $\quad \frac{n}{b^k} = 1 \quad b^k = n \quad \boxed{k = \log_b n}$

The Tree method for divide & conquer recurrences.

\# subproblems

how much work is done outside of recursive calls.

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

shrink factor

adding up work at each level

$$\sum_{k=0}^{\log_b n} a^k \left(\frac{n}{b^k}\right)^d = \sum_{k=0}^{\log_b n} \frac{a^k n^d}{(b^d)^k} = n^d \sum_{k=0}^{\log_b n} \underbrace{\left(\frac{a}{b^d}\right)}_{r}{}^k$$

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

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

$a = 2$
$b = 2$
$d = 1$

$2 = 2^1$

$\boxed{O(n \log n)}$

Exercise: use the Tree method to determine the complexity of binary search.

a: # subproblems
b: shrink factor on problem size
d: how much work is done outside recursive calls? n^d

```python
def binary_search(a, x):
  n = len(a)
  if n == 0:
    return False
  elif n == 1:
    if a[0] == value:
      return True
    return False

  m = n // 2
  if a[m] <= value:
    return binary_search(a[m:], x)
  else:
    return binary_search(a[:m], x)
```

The binary search algorithm is: (slido.com #3877567) 11

- O(n)
- O(n log n)
- O(log n)
- O(1)

Voting as Anonymous

Send

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

$b = 2$
$d = 0$
$a = 1 = 2^0$