Middlebury

CSCI 200: Math Foundations of Computing

Spring 2026

Lecture 4W: Recurrence Relations II

Goals for today:

- Identify divide-and-conquer recurrence relations,
- Use the Tree Theorem to analyze the complexity of divide-and-conquer recurrences.

```python
def binary_search(lst, val, lo, hi):
    if lo > hi:
        return -1
    m = (lo + hi) // 2
    if lst[m] == val:
        return m
    elif lst[m] > val:
        return binary_search(lst, val, lo, m - 1)
    else:
        return binary_search(lst, val, m + 1, hi)
```

Similar to Lab 3
sqrt 2
power3

Let's analyze the work done by Merge-Sort.

```python
def merge_sort(a: list) -> list:
    n = len(a)
    if n <= 1: # base case
      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 = []
    i = j = 0
    while i < len(a1) and j < len(a2):
      if a1[i] < a2[j]:
        merged.append(a1[i])
        i += 1
      else:
        merged.append(a2[j])
        j += 1

    while i < len(a1):
      merged.append(a1[i])
      i += 1

    while j < len(a2):
      merged.append(a2[j])
      j += 1
    return merged
```

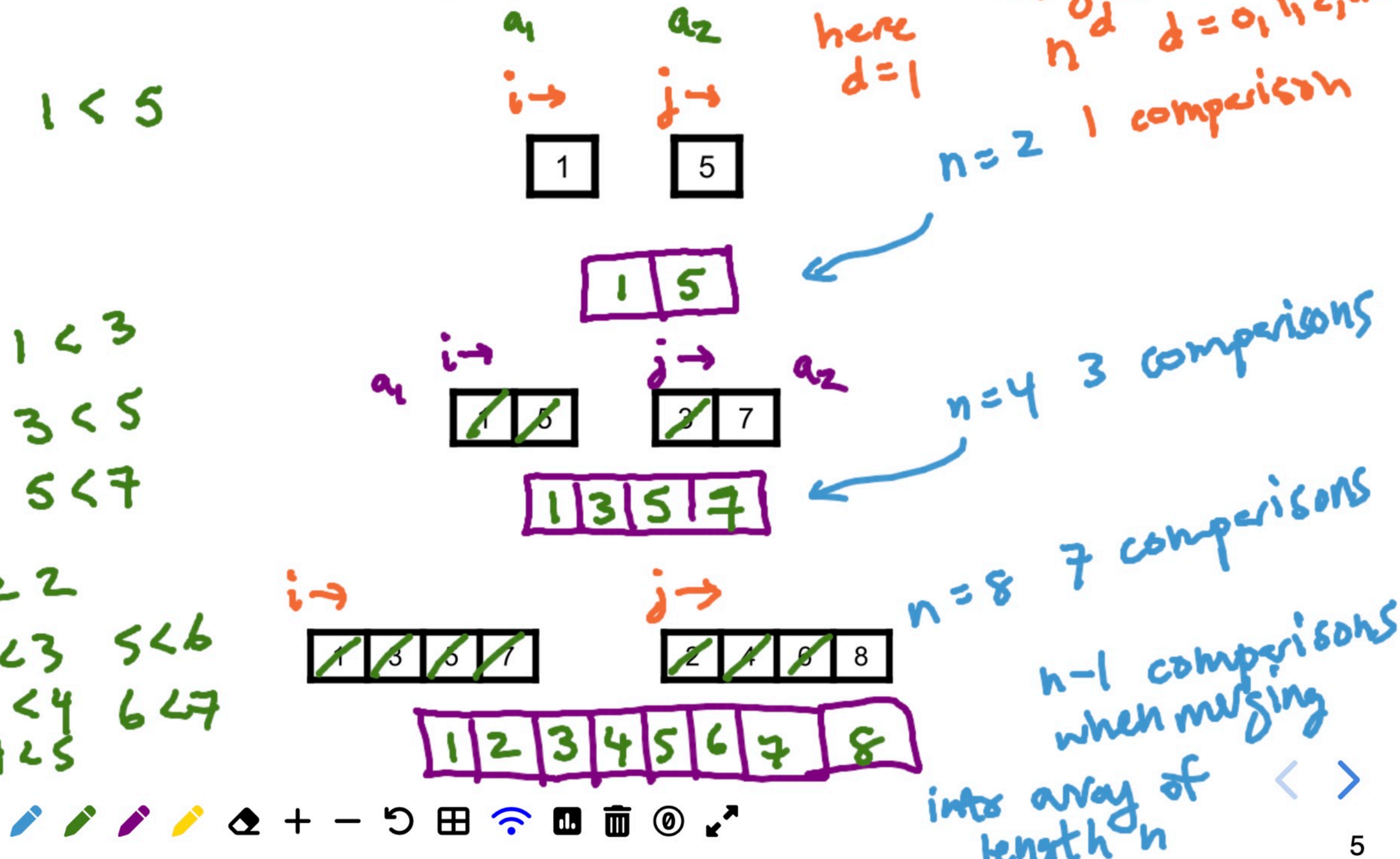
2 subproblems

this what we will count

`Merge-Sort` in action (from Wikipedia)

6 5 3 1 8 7 2 4

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



Adding up all the work done (Merge–Sort).

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

2 subproblems, breaking problem in half.

$\sum_{i=0}^{\square} r^i = \frac{1-r^{\square+1}}{1-r}$

$T(n) = (n-1) + 2T\left(\frac{n}{2}\right) \qquad T(\square) = (\square - 1) + 2T\left(\frac{\square}{2}\right)$

$T(n) = (n-1) + 2\left(\left(\frac{n}{2} - 1\right) + 2T\left(\frac{n}{4}\right)\right)$

$T(n) = (n-1) + (n-2) + 2^2 T\left(\frac{n}{4}\right)$

$T(n) = (n-1) + (n-2) + 2^2\left[\frac{n}{4} - 1 + 2T\left(\frac{n}{8}\right)\right]$

$T(n) = (n-1) + (n-2) + (n-4) + 2^3 T\left(\frac{n}{8}\right)$

$T(n) = \sum_{i=0}^{\log_2 n - 1} (n - 2^i) + 2^{\square} T\left(\frac{n}{2^{\square}}\right)$

$= n(\log_2 n) - \sum_{i}^{\log_2 n - 1} 2^i$ (A) $= n\log_2 n + (n-1)$

$O(n \log n)$

(A) $\frac{1 - 2^{\log_2 n}}{1-2}$

$= n - 1$

when $n = 1$

$T(1) = 0$

what is i here?

$\frac{n}{2^{i+1}} = 1 \rightarrow 2^{i+1} = n$

(upper bound of sum)

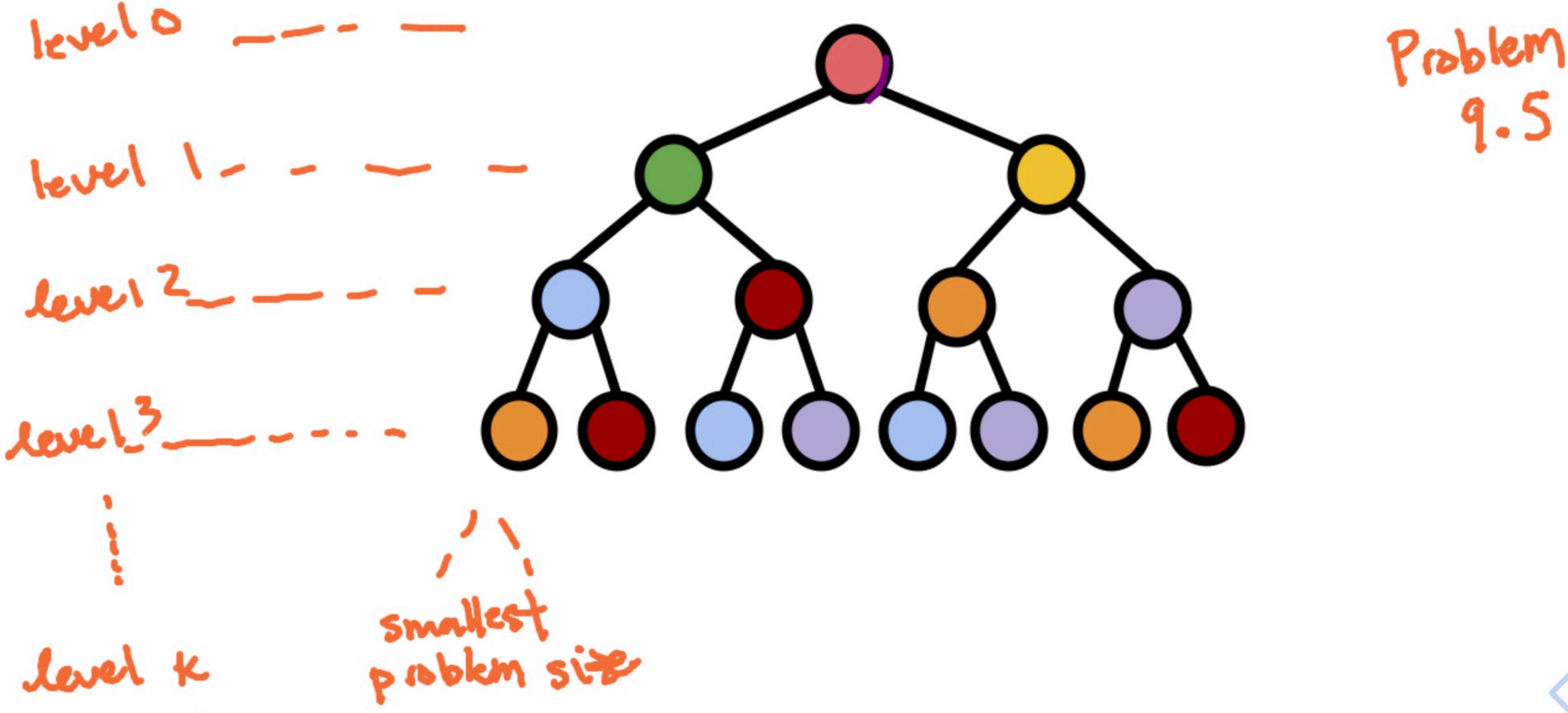
$i = \log_2 n - 1$

$D = \log_2 n$

Adding up all the work done (general divide-and-conquer recurrences).

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 Theorem for divide & conquer recurrences.

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

- a: # subproblems
- b: shrink factor on problem size
- $c \cdot n^d$: work done outside recursive calls

eg. merge sort

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

$a = 2$

$b = 2$

$d = 1$

problem 9.5

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

← merge-sort

$T(n) = O(n^1 \log n) = O(n \log n)$

Use the Tree Theorem to determine the complexity of binary search. Discuss and then vote.

```python
def binary_search(lst, val, lo, hi):
    if lo > hi:
        return -1
    m = (lo + hi) // 2
    if lst[m] == val:
        return m
    elif lst[m] > val:
        return binary_search(lst, val, lo, m - 1)
    else:
        return binary_search(lst, val, m + 1, hi)
```

A. $O(n)$
B. $O(n \log n)$
C. $O(\log n)$
D. $O(1)$

$$T(n) = 1\,T\left(\frac{n}{2}\right) + c$$

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

$b = 2$

$d = 0$

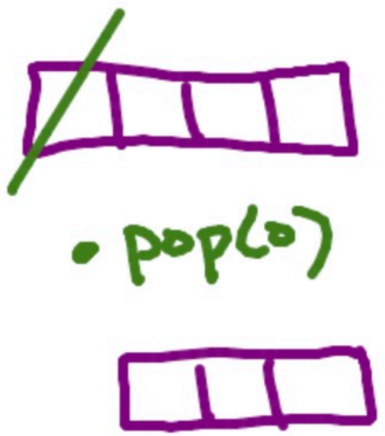
$1 = 2^0$

$O(n^0 \log n) = O(\log n)$

What about this implementation of `Merge-Sort`? Look up the complexity of `.pop(0)`. Discuss and then vote.

```python
def merge_sort(a: list) -> list:
    n = len(a)
    if n <= 1: # base case
      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 and len(a2) > 0:
      if a1[0] < a2[0]:
        merged.append(a1.pop(0))
      else:
        merged.append(a2.pop(0))

    while len(a1) > 0:
      merged.append(a1.pop(0))

    while len(a2) > 0:
      merged.append(a2.pop(0))
    return merged
```

$O(n^2)$



A. $O(n^2)$
B. $O(n \log n)$
C. $O(n^2 \log n)$
D. $O(\log(n^2))$

$T(n) = 2T(\frac{n}{2}) + n^d$

$d \; ? \; 2$

$a < b^d$

$2 < 2^2$

$O(n^2)$