

# CSE 332 Autumn 2026

## Lecture 6: Recurrences

Ruth Anderson

<http://www.cs.uw.edu/332>

# Announcements

- Exercise 0 – was due on Friday, closes tonight
- “Meet the staff” activity - Sometime during the first 4 weeks of class.
- Participate in Section, earn a late day (see Ed Board post)
- Exercise 1 Running Time:
  - Due TONIGHT Mon Oct 12, Closes Wed Oct 14
  - Entirely in Gradescope. Save your answers outside of Gradescope and confirm that you have answered/uploaded everything that is needed!
- Exercise 2 Heaps: due Fri Oct 16
- Exercise 3 Recurrences: coming soon, due Fri Oct 23

# Analyzing Recursive Code

1. Pick a Scenario/Case to analyze (usually worst)
2. Count the number of *chosen* operation(s) in the recursive code, come up with a Recurrence Relation in terms of  $n$  (input size)
3. Use the “Tree Method” to visualize the number of levels in the tree and the work done at each level of the recursion
4. Convert the Recurrence Relation into a summation
5. Convert the summation into a closed form (e.g.  $O(n)$ ,  $O(\log n)$ )

# Recursive Binary Search

|   |   |    |    |    |    |    |    |    |    |
|---|---|----|----|----|----|----|----|----|----|
| 5 | 8 | 13 | 42 | 75 | 79 | 88 | 90 | 95 | 99 |
| 0 | 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |

```
public static boolean binarySearch(List<Integer> lst, int k){
    return binarySearch(lst, k, 0, lst.size());
}

private static boolean binarySearch(List<Integer> lst, int k, int start, int end){
    if(start == end)
        return false;
    int mid = start + (end-start)/2;
    if(lst.get(mid) == k){
        return true;
    } else if(lst.get(mid) > k){
        return binarySearch(lst, k, start, mid);
    } else{
        return binarySearch(lst, k, mid+1, end);
    }
}
```

# Coming up with a Recurrence Relation

- Overall structure of a recursive function:
  1. Do some non-recursive “work”
  2. Do one or more recursive calls on some portion of your input
  3. (maybe) Do some more non-recursive “work”
    - (Repeat recursive calls until you reach a base case)
- Recurrence Relation:  $T(n) = T(p_1) + T(p_2) + \dots + T(p_x) + f(n)$
- The time it takes to run the algorithm on an input of size  $n$  is:
  - The sum of how long it takes to run the same function on each smaller input
  - Plus the total amount of non-recursive work done at that step
- Usually:
  - $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ 
    - Called “divide and conquer”
  - $T(n) = T(n - c) + f(n)$ 
    - Called “chip and conquer”

# How Efficient Is It?

- $T(n) = 1 + T\left(\left\lceil \frac{n}{2} \right\rceil\right)$
- Base case:  $T(1) = 1$

$T(n)$  = “cost” of running the entire algorithm on an array of length  $n$

# Let's Solve the Recurrence!

$$T(1) = 1$$

$$T(n) = 1 + \cancel{T(n/2)}$$

$$1 + \cancel{T(n/4)}$$

$$1 + \cancel{T(n/8)}$$

...

1

Substitute until  $T(1)$   
So  $\log_2 n$  steps

$$T(n) = \sum_{i=1}^{\log_2 n} 1 = \log_2 n$$

$$T(n) \in \Theta(\log n)$$

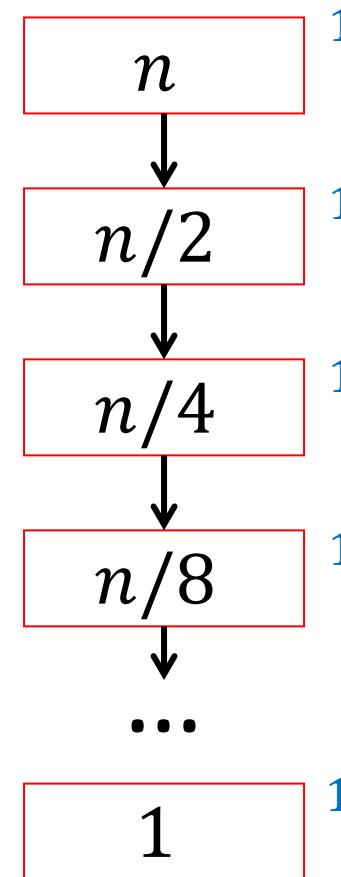
# Make our process “prettier”

- Draw a picture of the recursion
  - One box per stack frame
- Identify the work done per stack frame
- Add up all the work!
  - Sum is the answer!
  - In this case  $\Theta(\log_2 n)$

## The “Tree Method”

$$T(n) = \sum_{i=1}^{\log_2 n} 1 = \log_2 n \quad T(n) \in \Theta(\log n)$$

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



$\log_2 n$  levels  
of recursion

# Recursive Linear Search

|   |   |    |    |    |    |    |    |    |    |
|---|---|----|----|----|----|----|----|----|----|
| 5 | 8 | 13 | 42 | 75 | 79 | 88 | 90 | 95 | 99 |
| 0 | 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |

```
public static boolean linearSearch(List<Integer> lst, int k){
    return linearSearch(lst, k, 0, lst.size());
}

private static boolean linearSearch(List<Integer> lst, int k, int start, int end){
    if(start == end){
        return false;
    } else if(lst.get(start) == k){
        return true;
    } else{
        return linearSearch(lst, k, start+1, end);
    }
}
```

# Prettier List Sum Substitution

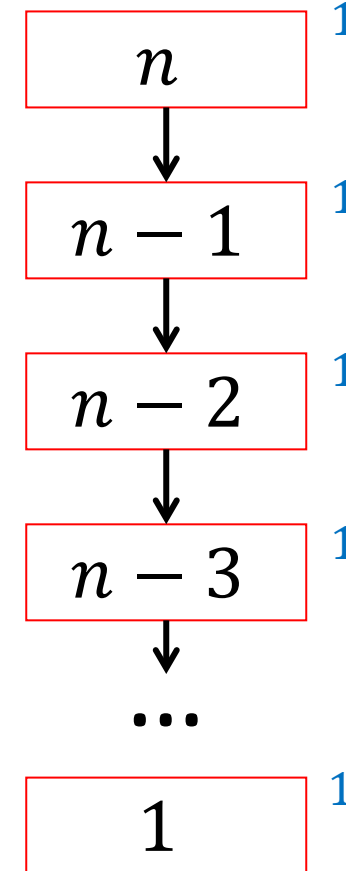
- Draw a picture of the recursion
- Identify the work done per stack frame
- Add up all the work!

Running time:  $\Theta(n)$

$$T(n) = \sum_{i=1}^n 1 = n$$

$$T(n) \in \Theta(n)$$

$$T(n) = T(n-1) + 1$$



$n$  levels  
of recursion

# Recursive List Summation Code

```
public int sum(int[] list){
    return sum_helper(list, 0, list.size);
}

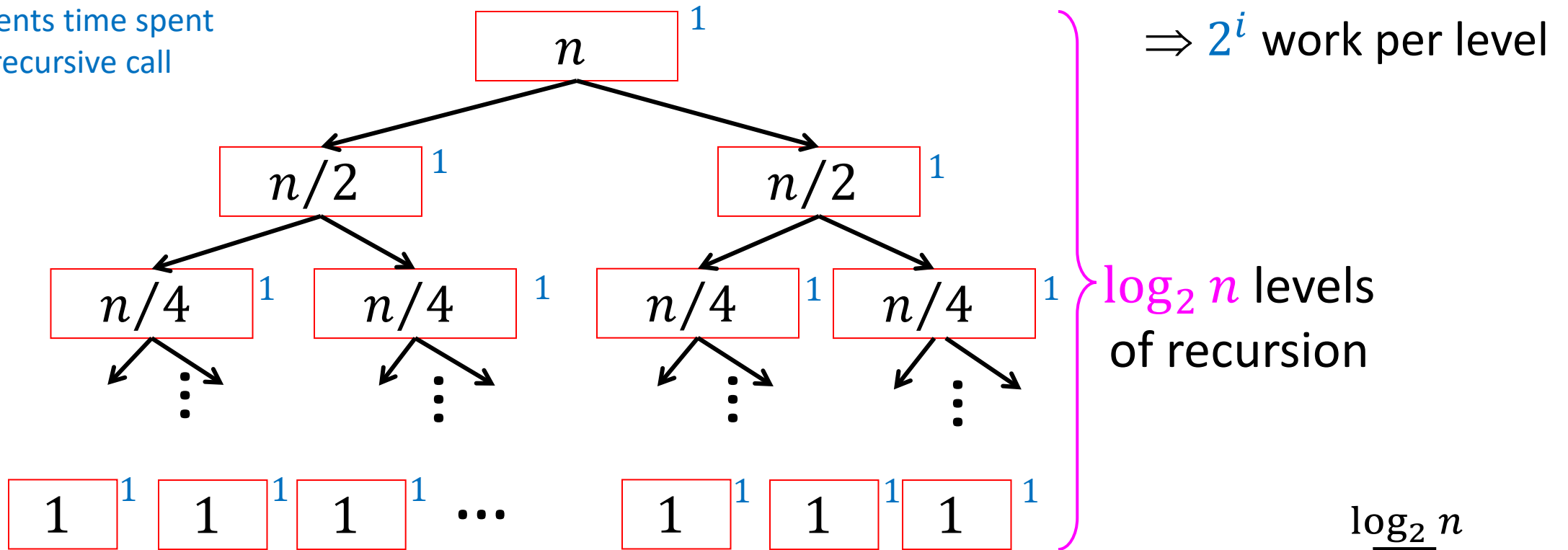
private int sum_helper(int[] list, int low, int high){
    if (low == high){ return 0; }
    if (low == high-1){ return list[low]; }
    int middle = (high+low)/2;
    return sum_helper(list, low, middle) + sum_helper(list, middle, high);
}
```

# Tree Method for Recursive Summation

Red box represents a problem instance

Blue value next to each box represents time spent at that recursive call

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



$$T(n) = \sum_{i=0}^{\log_2 n} 2^i$$

# Recursive List Summation – Solving the Series

$$T(n) = \sum_{i=0}^{\log_2 n} 2^i$$

Remember the  
[Math Identities Reference Sheet](#)  
(will be provided on exams)

$$= \left( \frac{2^{1+\log_2 n} - 1}{2 - 1} \right)$$

$$= 2 \cdot 2^{\log_2 n} - 1$$

$$= 2n - 1 = \Theta(n)$$

# Tree Method Summary: Chip and Conquer

- Recurrence looks like  $T(n) = aT(n - b) + f(n)$
- Use the recurrence to draw a tree
  - $a$  is the branching factor of the tree (e.g. if  $a = 2$  then it's a binary tree)
  - Subtract  $b$  from the parent's input size to get children's input size
  - Work done per node is given by applying  $f(n)$  to that node's input size
  - Height of the tree is  $\frac{n}{b}$ 
    - Because that is the number of times we must subtract  $b$  until reaching a base case
    - Answer to the question "how many times must we subtract  $b$  until we reach 0?"
      - Any base case is a constant, so to reach a larger value would just be a constant change
- Use the tree to express running time as a series
  - Adding work done for each node level-by-level
  - Identify a pattern to express work done at level  $i$  as a function of  $i$
  - Write a series using  $i = 0$  up to  $\frac{n}{b}$
- Solve the series

# Tree Method Summary: Divide and Conquer

- Recurrence looks like  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$
- Use the recurrence to draw a tree
  - $a$  is the branching factor of the tree (e.g. if  $a = 2$  then it's a binary tree)
  - Divide the parent's input size by  $b$  to get children's input size
  - Work done per node is given by applying  $f(n)$  to that node's input size
  - Height of the tree is  $\log_b n$ 
    - Because that is the number of times we must divide by  $b$  until reaching a base case
    - Answer to the question "how many times must we divide by  $b$  until we reach 1?"
      - Any base case is a constant, so to reach a larger value would just be a constant change
- Use the tree to express running time as a series
  - Adding work done for each node level-by-level
  - Identify a pattern to express work done at level  $i$  as a function of  $i$
  - Write a series using  $i = 0$  up to  $\log_b n$
- Solve the series

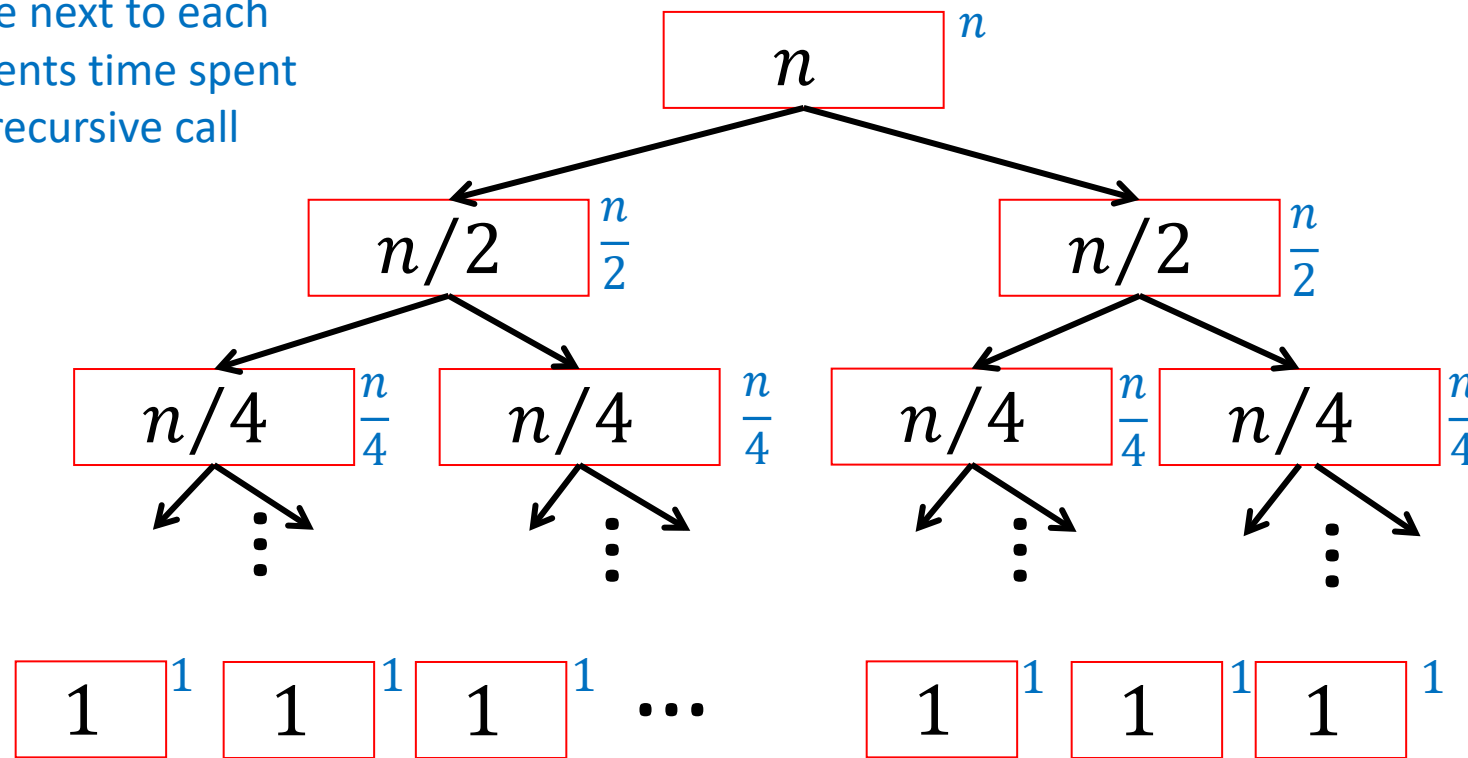
# Let's do some more!

- For each, assume the base case is  $n = 1$  and  $T(1) = 1$
- $T(n) = 2T\left(\frac{n}{2}\right) + n$
- $T(n) = 2T\left(\frac{n}{2}\right) + n^2$
- $T(n) = 2T\left(\frac{n}{8}\right) + 1$

# Tree Method $T(n) = 2T\left(\frac{n}{2}\right) + n$

Red box represents a problem instance

Blue value next to each box represents time spent at that recursive call



$\Rightarrow n$  work per level

$\log_2 n$  levels of recursion

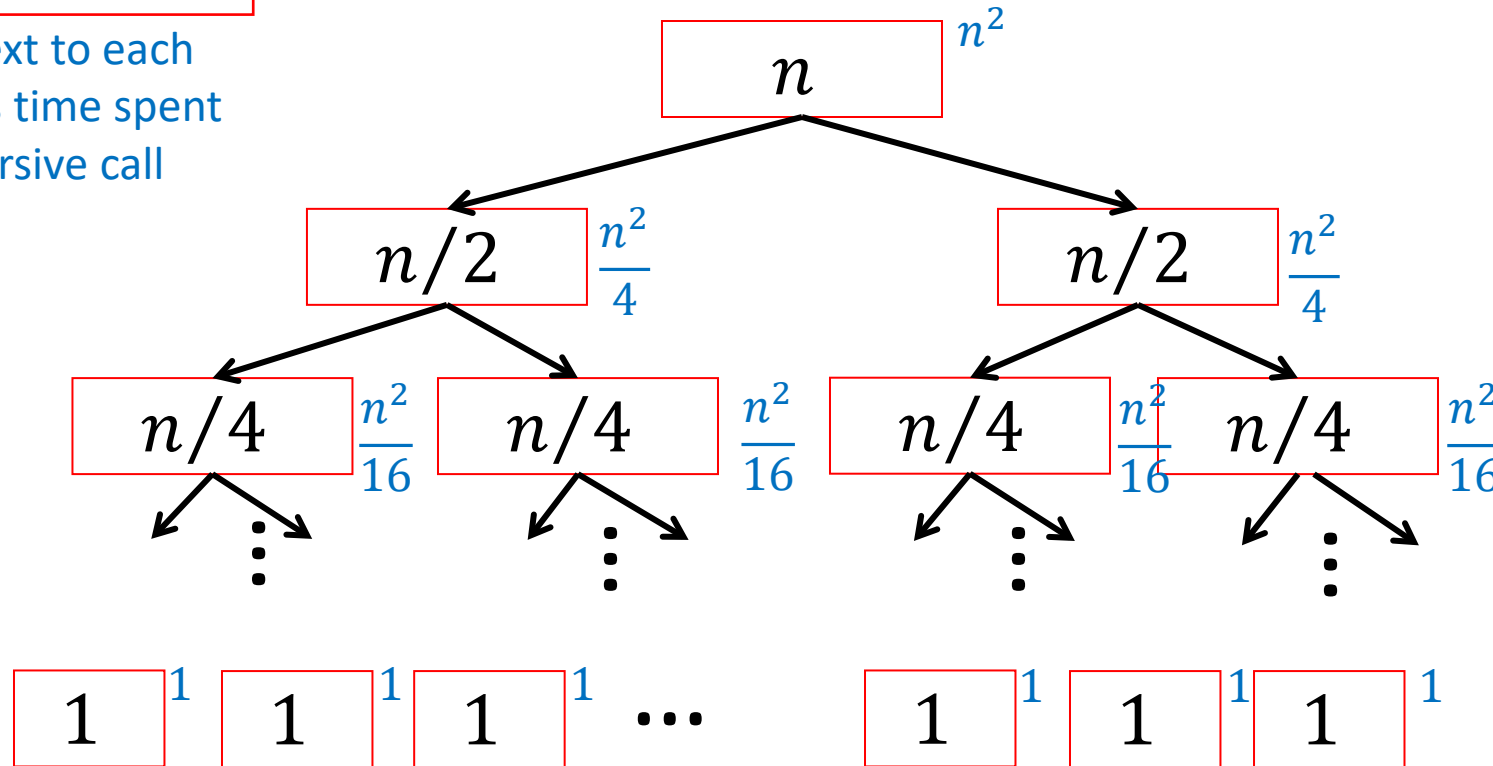
$$T(n) = \sum_{i=0}^{\log_2 n} n$$

$$T(n) \in \Theta(n \log n)$$

Tree Method  $T(n) = 2T\left(\frac{n}{2}\right) + n^2$

Red box represents a problem instance

Blue value next to each box represents time spent at that recursive call



⇒ ?? work per level

$\log_2 n$  levels  
of recursion

$$T(n) = \sum_{i=0}^{\log_2 n} ??$$

$$\text{Solving } T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

$$T(n) = \sum_{i=0}^{\log_2 n} \frac{n^2}{2^i}$$

$$= n^2 \cdot \sum_{i=0}^{\log_2 n} \left(\frac{1}{2}\right)^i$$

$$\leq n^2 * 2 = \Theta(n^2)$$

Remember the

[Math Identities Reference Sheet](#)

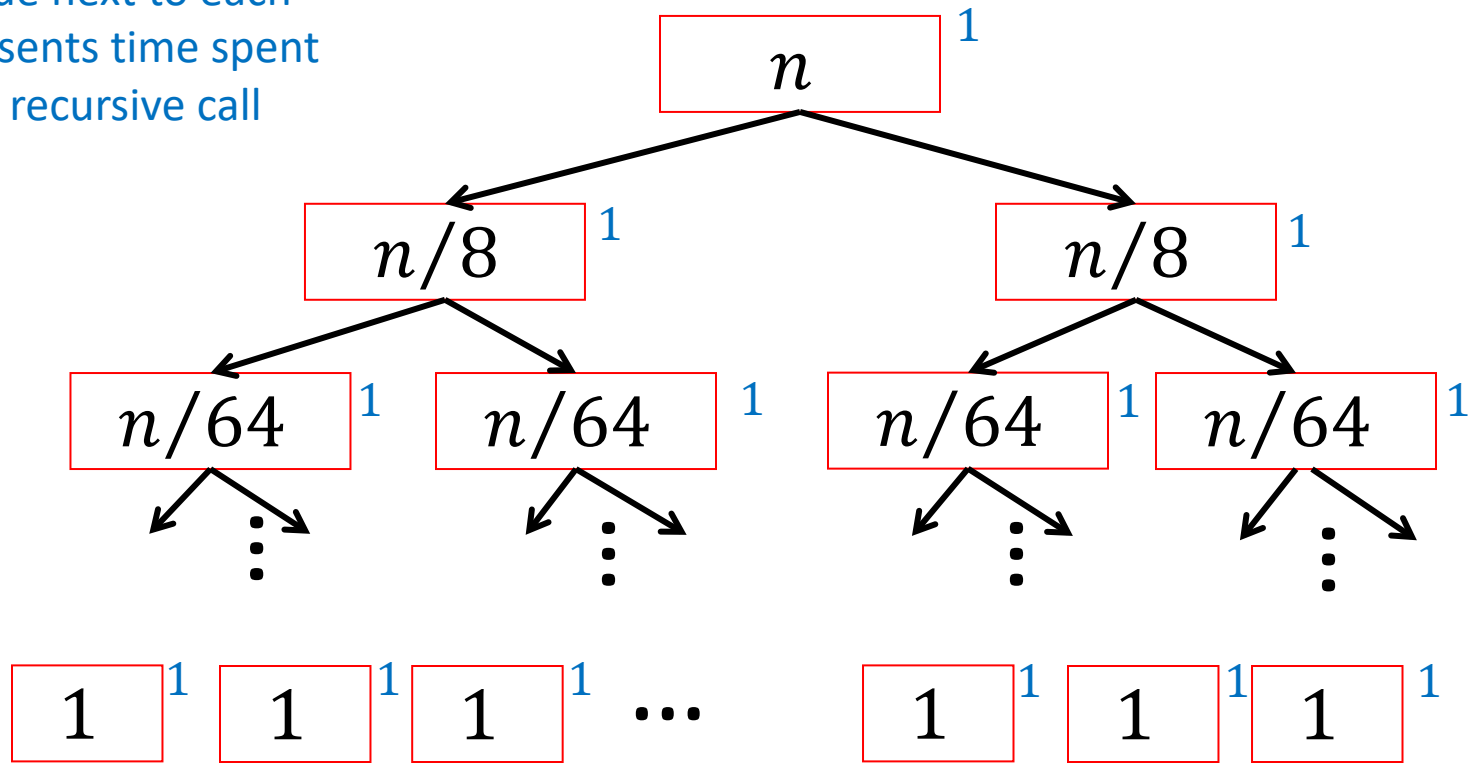
(will be provided on exams)

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \text{ for } |x| < 1$$

# Tree Method $T(n) = 2T\left(\frac{n}{8}\right) + 1$

Red box represents a problem instance

Blue value next to each box represents time spent at that recursive call



$\Rightarrow 2^i$  work per level

$\log_8 n$  levels of recursion

$$T(n) = \sum_{i=0}^{\log_8 n} 2^i$$

Solving  $T(n) = 2T\left(\frac{n}{8}\right) + 1$

$$T(n) = \sum_{i=0}^{\log_8 n} 2^i$$

$$= \left( \frac{2^{1+\log_8 n} - 1}{2 - 1} \right)$$

$$= 2 \cdot 2^{\log_8 n} - 1$$

$$= 2n^{\log_8 2} - 1 = \Theta\left(n^{\frac{1}{3}}\right)$$

Remember the  
[Math Identities Reference Sheet](#)  
(will be provided on exams)

$$a^{\log_b c} = c^{\log_b a}$$

# What matters, recursively

- For  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ 
  - The following are important for asymptotic behavior:
    - The value of  $a$
    - The value of  $b$
    - Asymptotic behavior of  $f(n)$
  - The following are not important for asymptotic behavior:
    - Constants and non-dominant terms in  $f(n)$
    - The base case

# Really common recurrences

Should know how to solve recurrences but also recognize some really common ones:

$$T(n) = O(1) + T(n/2)$$

logarithmic  $O(\log n)$

$$T(n) = O(1) + 2T(n/2)$$

linear  $O(n)$

$$T(n) = O(1) + T(n-1)$$

linear  $O(n)$

$$T(n) = O(n) + T(n-1)$$

quadratic  $O(n^2)$

$$T(n) = O(1) + 2T(n-1)$$

exponential  $O(2^n)$

$$T(n) = O(n) + T(n/2)$$

linear  $O(n)$

$$T(n) = O(n) + 2T(n/2)$$

loglinear  $O(n \log n)$