CSE 332: Data Structures & Parallelism

Lecture 6: Recurrences

Ruth Anderson

Winter 2025

Worst Case

binarySearch(15)

$\Theta(\log_2 n)$

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);
    }
}
```

1

4

2

2

$$\underbrace{T(n) = 9 + T\left(\frac{n}{2}\right)}_{\text{Recurrence Relation}}$$

$T(n) = T\left(\frac{n}{2}\right) + 9$

Analysis of Recursive Algorithms

1 call

- Overall structure of recursion:
  - Do some non-recursive "work"
  - Do one or more recursive calls on some portion of your input
  - Do some more non-recursive "work"
  - Repeat until you reach a base case
- Running time: $T(n) = T(p_1) + T(p_2) + \cdots + T(p_x) + f(n)$
  - 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 algorithm on each smaller input
  - Plus the total amount of non-recursive work done at that step
- Usually:
  - $T(n) = \overset{1}{a} \cdot T\left(\frac{n}{b}\right)^{2} + f(n)$
    - 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!

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

$\rightarrow T(1) = 1$

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

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

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

$\vdots$

1

Substitute until $T(1)$

So $\log_2 n$ steps

$$\frac{n}{2} \cdot \underbrace{\frac{1}{2} \cdot \frac{1}{2} \cdot \ldots \cdot \frac{1}{2}}_{?} \rightarrow 1$$

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

$\Downarrow$

$$\log_2(2^i) = \log_2(n)$$

$\Downarrow$

$$i = \log_2 n$$

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

i = number of substitutions needed until got to base case

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

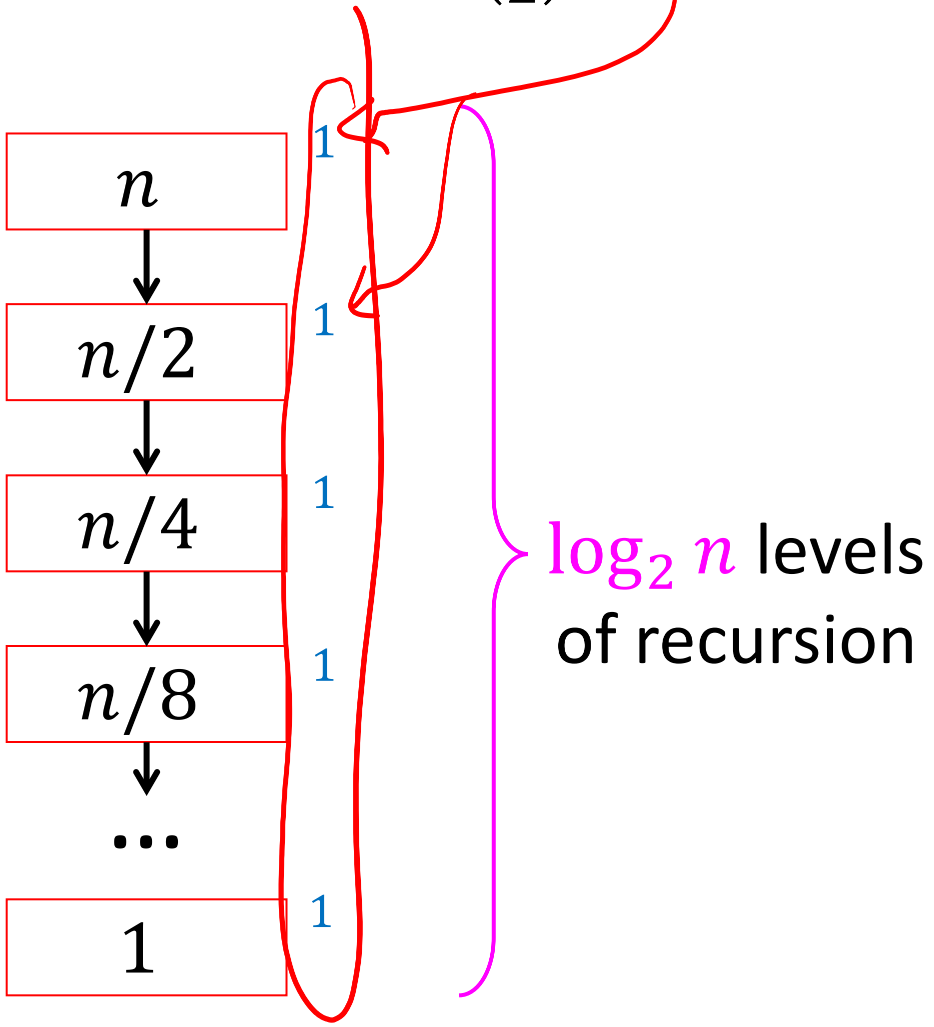
$$T(n) \in \Theta(\log n)$$

Make our process “prettier”

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

- Draw a picture of the recursion
- 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”



Recursive Linear Search

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

```java
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);
    }
}
```

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

1

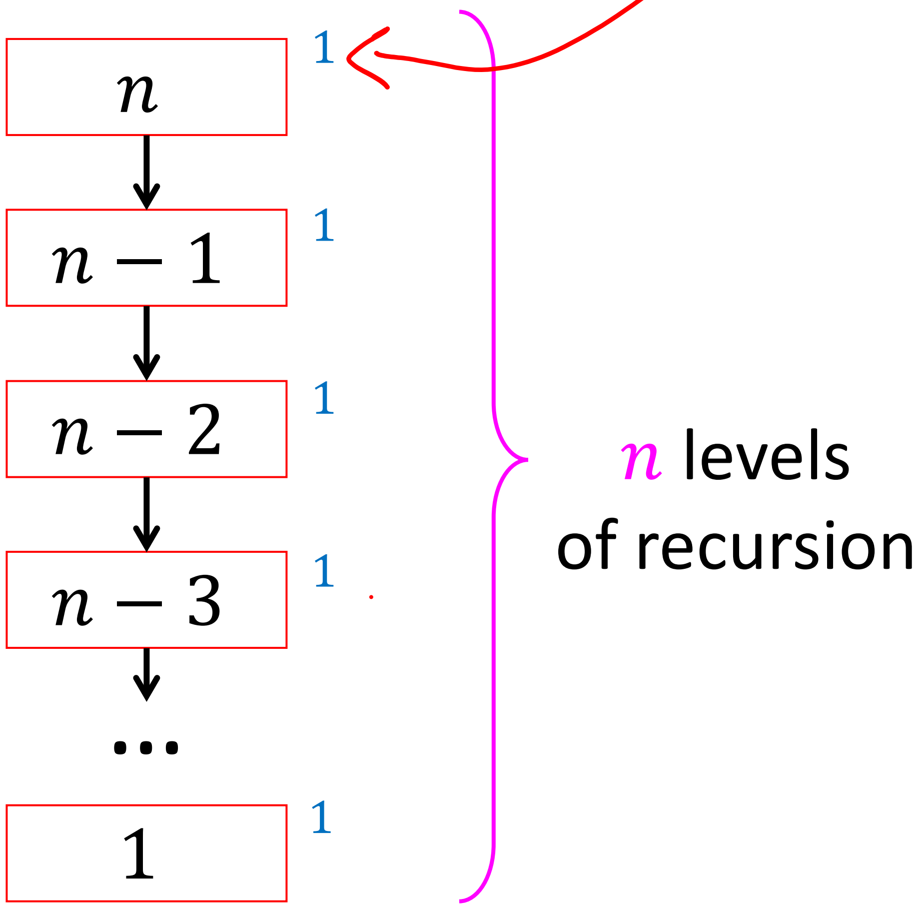
4

c

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

Make our method “prettier”

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



Running time: $\Theta(n)$

Recursive List Summation

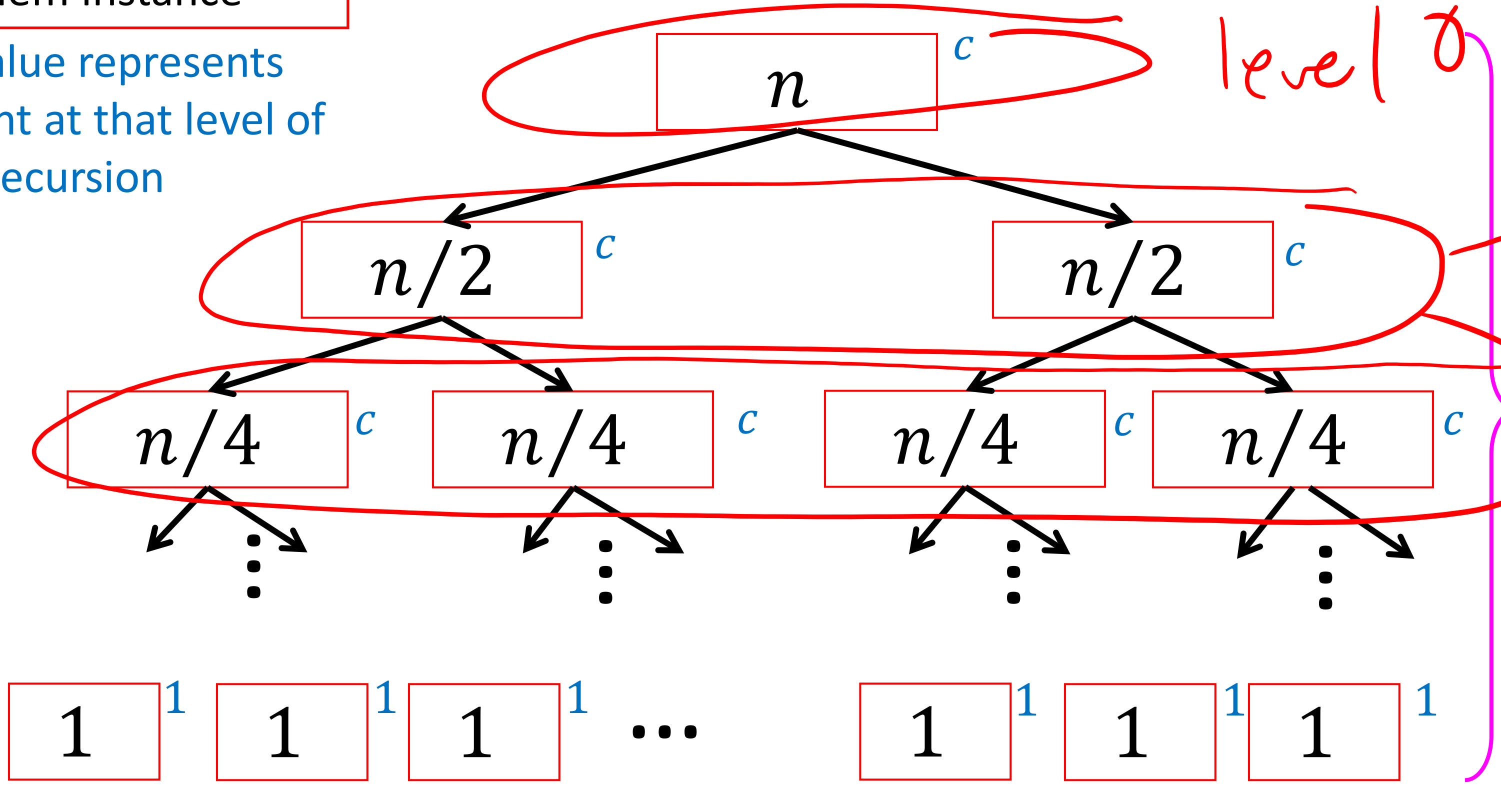
$T(n) = 2 \cdot T(\frac{n}{2}) + c$

```
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: $T(n) = 2T\left(\frac{n}{2}\right) + c$

Red box represents a problem instance

Blue value represents time spent at that level of recursion



$\Rightarrow 2^i \cdot c$ work per level

$\log_2 n$ levels of recursion

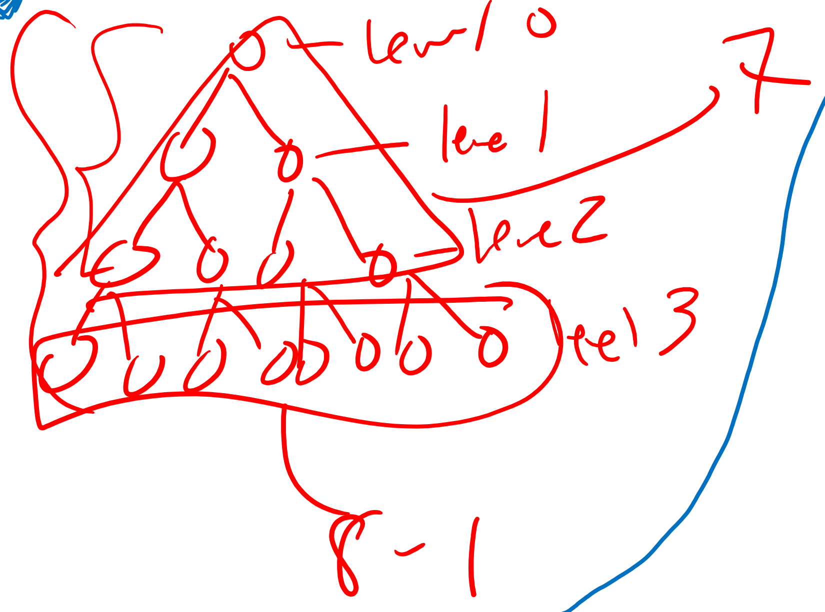
$$T(n) = \sum_{i=0}^{(\log_2 n)-1} 2^i \cdot c$$

Remember? (see lecture 4 slides)

Number of nodes in a perfect binary tree of height h?

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

Recursive List Summation



$$T(n) = \sum_{i=0}^{(\log_2 n)\,-\,1} 2^i \cdot c$$

$$= \boxed{c \cdot \sum_{i=0}^{(\log_2 n)\,-\,1} 2^i}$$

or use the ~~more~~ general formula

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

$$c\left(2^{\log_2 n} - 1\right)$$

$$c(n-1)$$

$$cn - c \rightarrow \Theta(n)$$

A "useful" Math Identity
(see link on exercises page)

$$\sum_{i=0}^{n-1} x^i = \frac{1-x^n}{1-x}$$

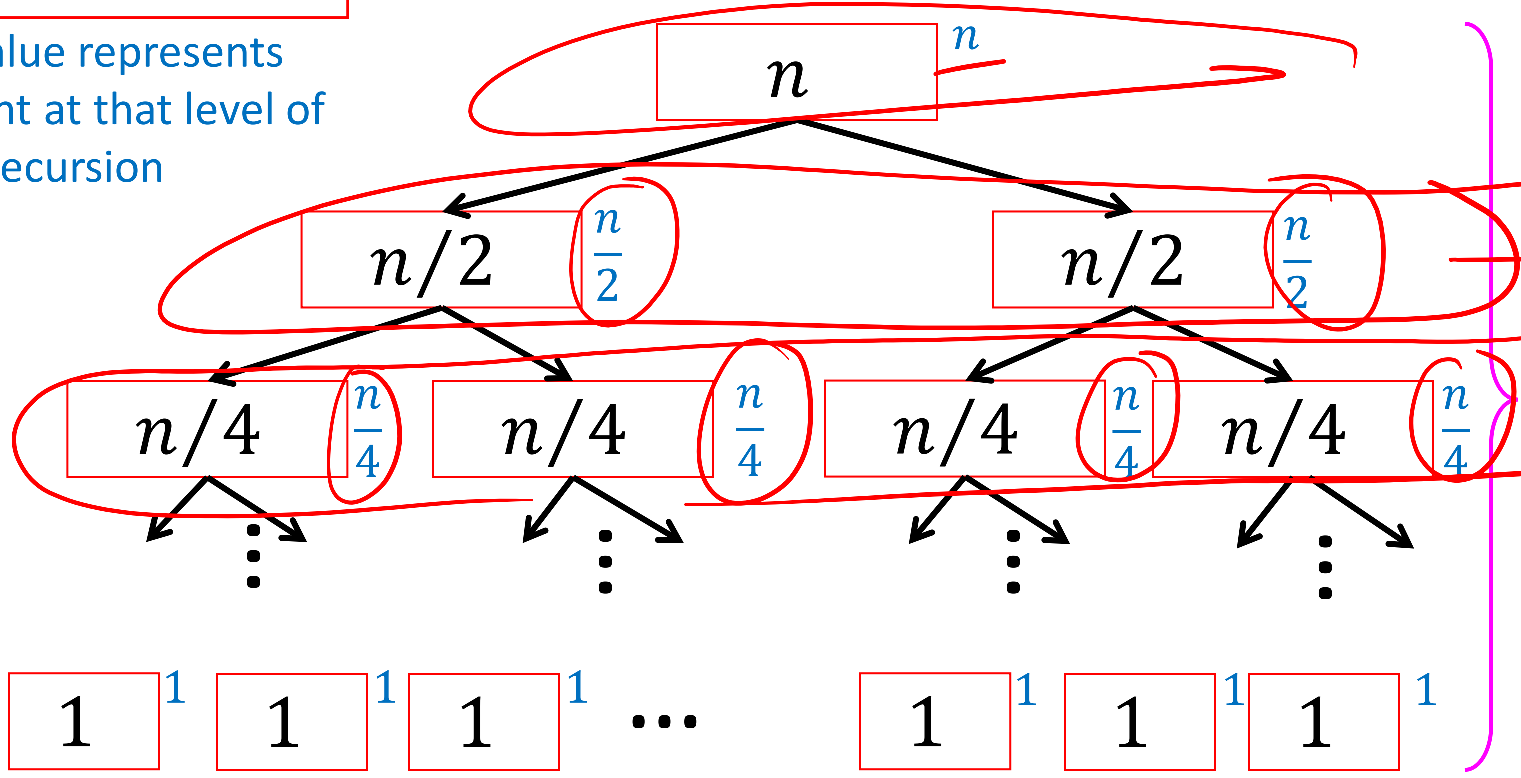
Let's do some more!

- 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 represents time spent at that level of recursion



Total

$\Rightarrow n$ work per level

$\frac{n}{2} + \frac{n}{2} = n$

$\frac{n}{4} + \frac{n}{4} + \frac{n}{4} + \frac{n}{4} = n$

$\log_2 n$ levels of recursion

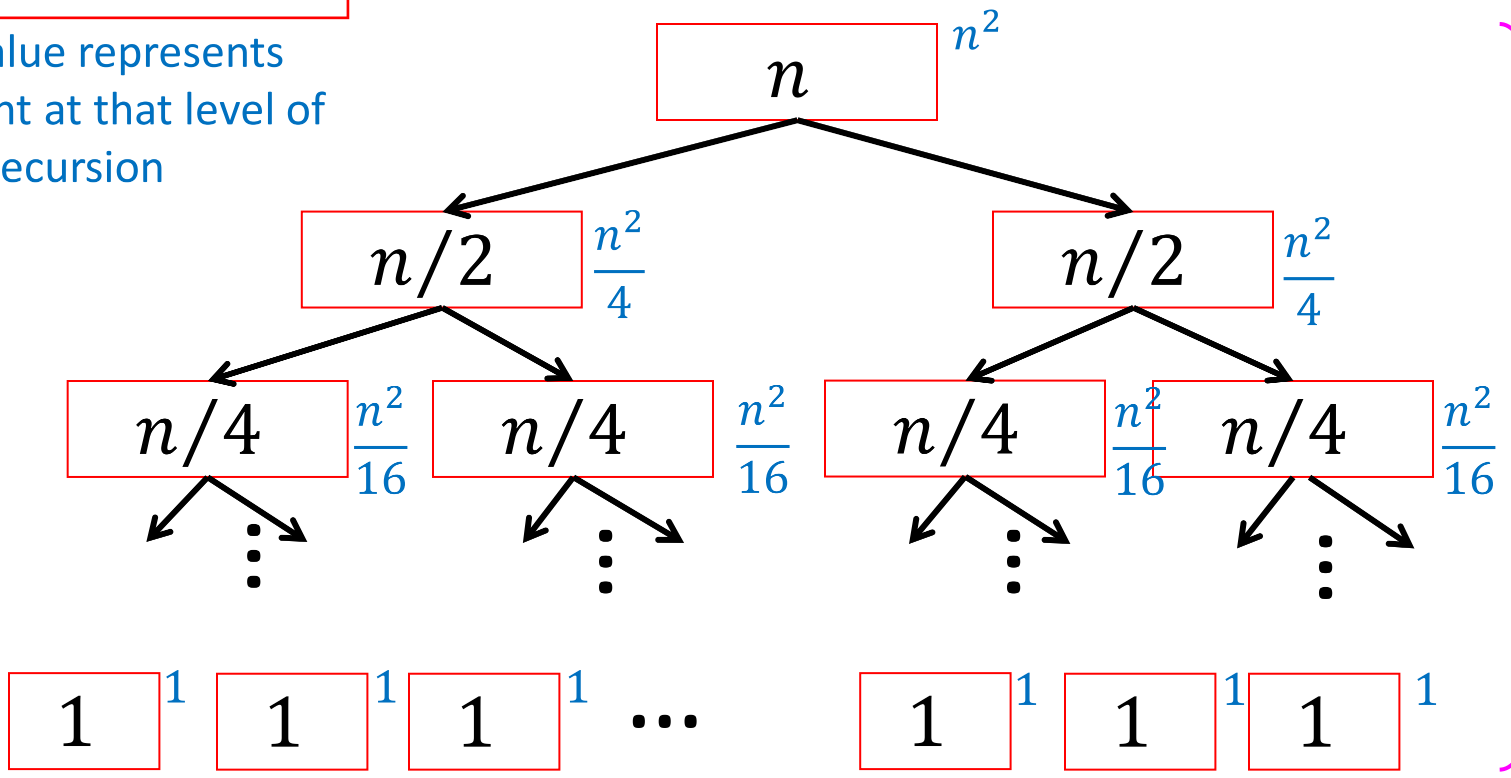
$$T(n) = \sum_{i=1}^{\log_2 n} n$$

$O(n \log n)$

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

Red box represents a problem instance

Blue value represents time spent at that level of recursion



$\Rightarrow$?? work per level

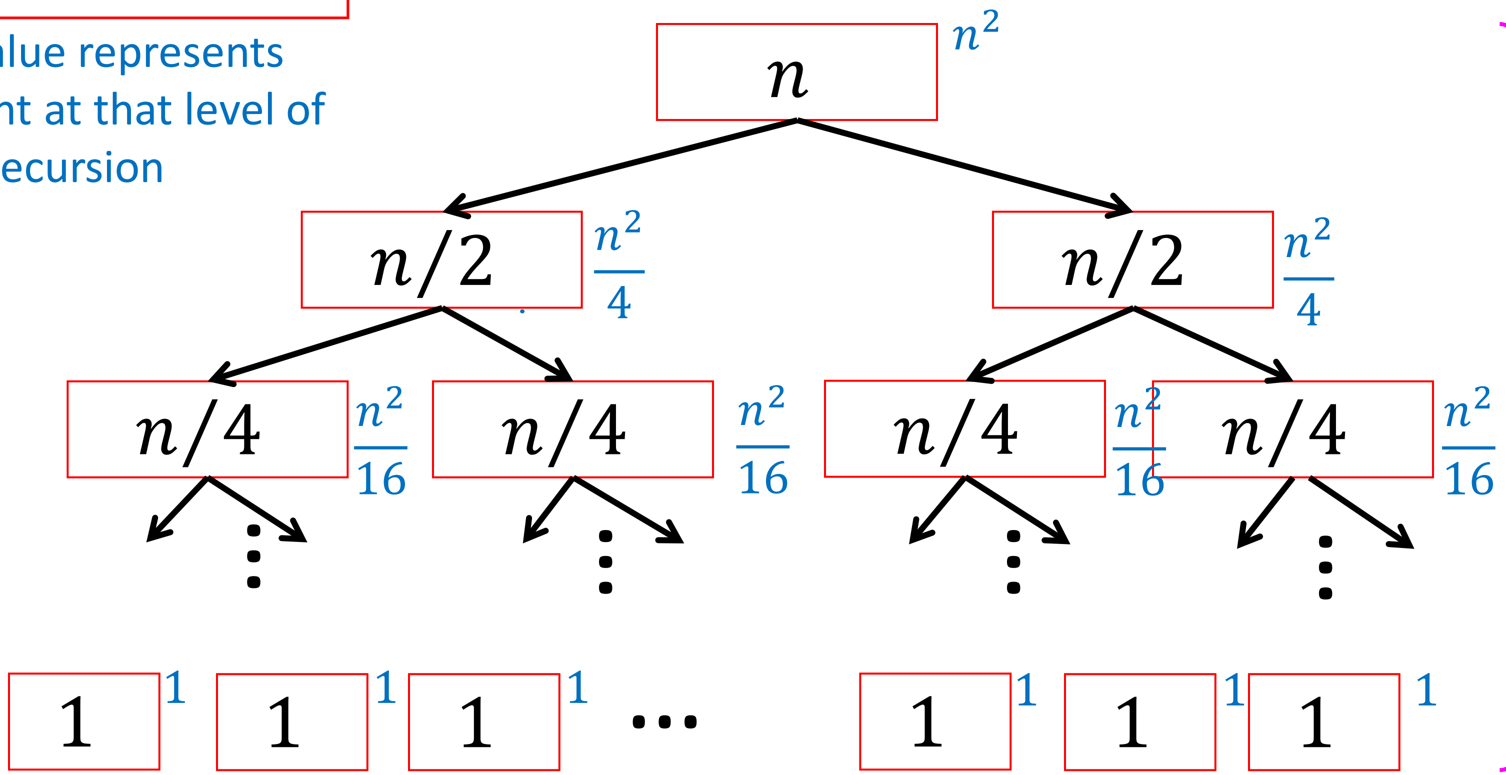
$\log_2 n$ levels of recursion

$$T(n) = \sum_{i=0}^{(\log_2 n)-1} ??$$

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

Red box represents a problem instance

Blue value represents time spent at that level of recursion



$\Rightarrow \frac{n^2}{2^i}$ work per level

$\log_2 n$ levels of recursion

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

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

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

[A "useful" Math Identity](#)
(see link on [exercises page](#))

$$\sum_{i=0}^{n-1} x^i = \frac{1-x^n}{1-x}$$

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

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

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

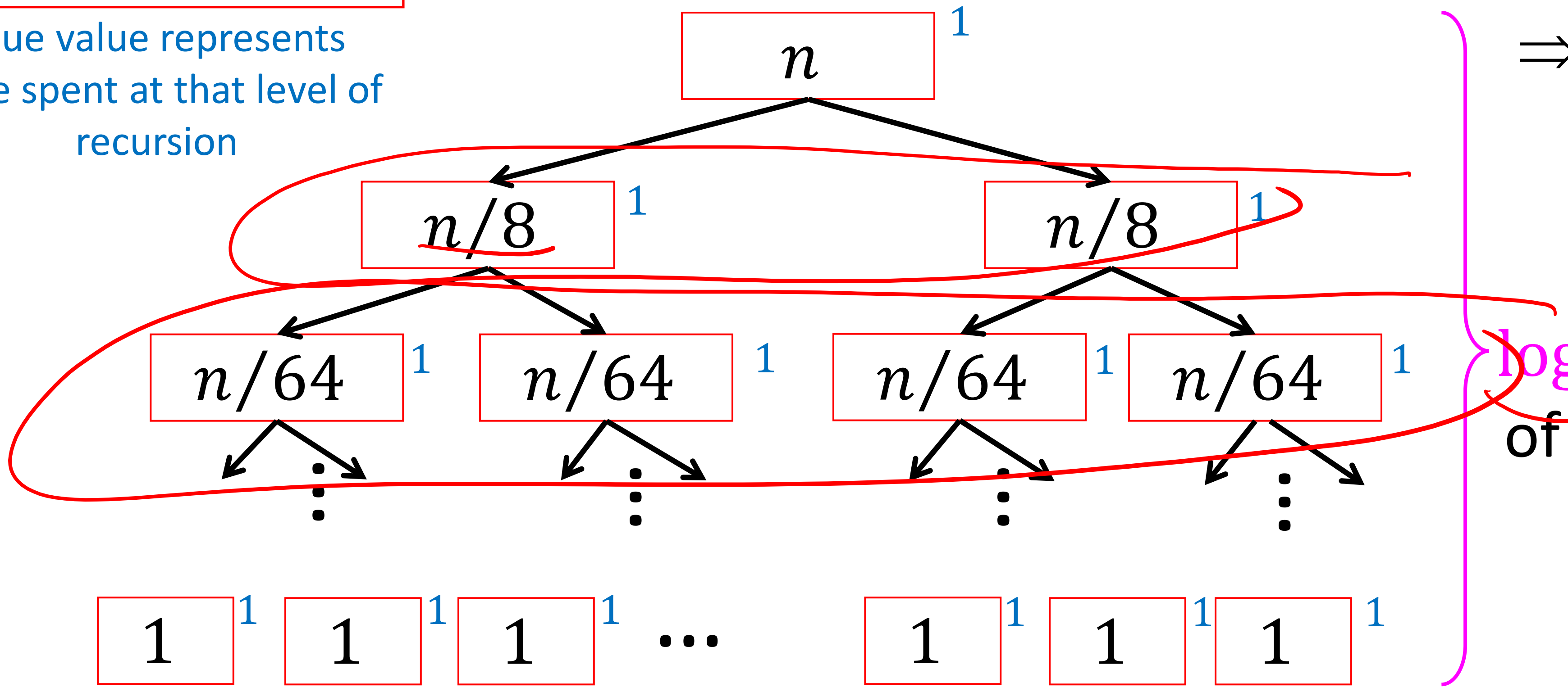
A "useful" Math Identity
(see link on exercises page)

$$\sum_{i=0}^{n-1} x^i = \frac{1-x^n}{1-x}$$

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

Red box represents a problem instance

Blue value represents time spent at that level of recursion



$\Rightarrow 2^i$ work per level

$\log_8 n$ levels of recursion

$$T(n) = \sum_{i=0}^{(\log_8 n)-1} 2^i$$

$$T(n) = \sum_{i=0}^{(\log_8 n)-1} 2^i$$

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

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

$$= n^{\log_8 2} = n^{\frac{1}{3}}$$

A “useful” Math Identity
(see link on exercises page)

$$\sum_{i=0}^{n-1} x^i = \frac{1-x^n}{1-x}$$

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

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	$O(n \log n)$