CSE 332 Winter 2026
Lecture 6: Recurrences

Nathan Brunelle

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

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);
    }
}
```

Analysis of Recursive Algorithms

- 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 in that stack frame
- 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!

$T(1) = 1$

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

$1 + T(n/4)$

$1 + T(n/8)$

$\dots$

1

Substitute until $T(1)$

So $\log_2 n$ steps

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

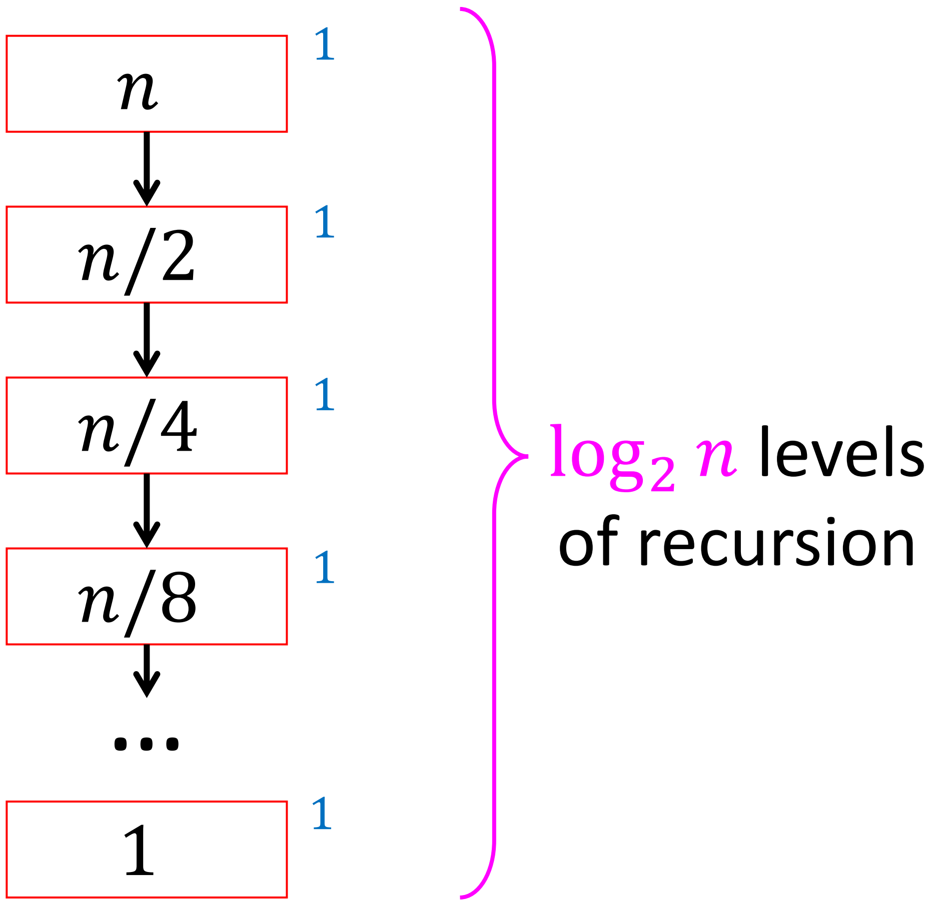
$$T(n) \in \Theta(\log n)$$

Make our process "prettier"

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

- 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

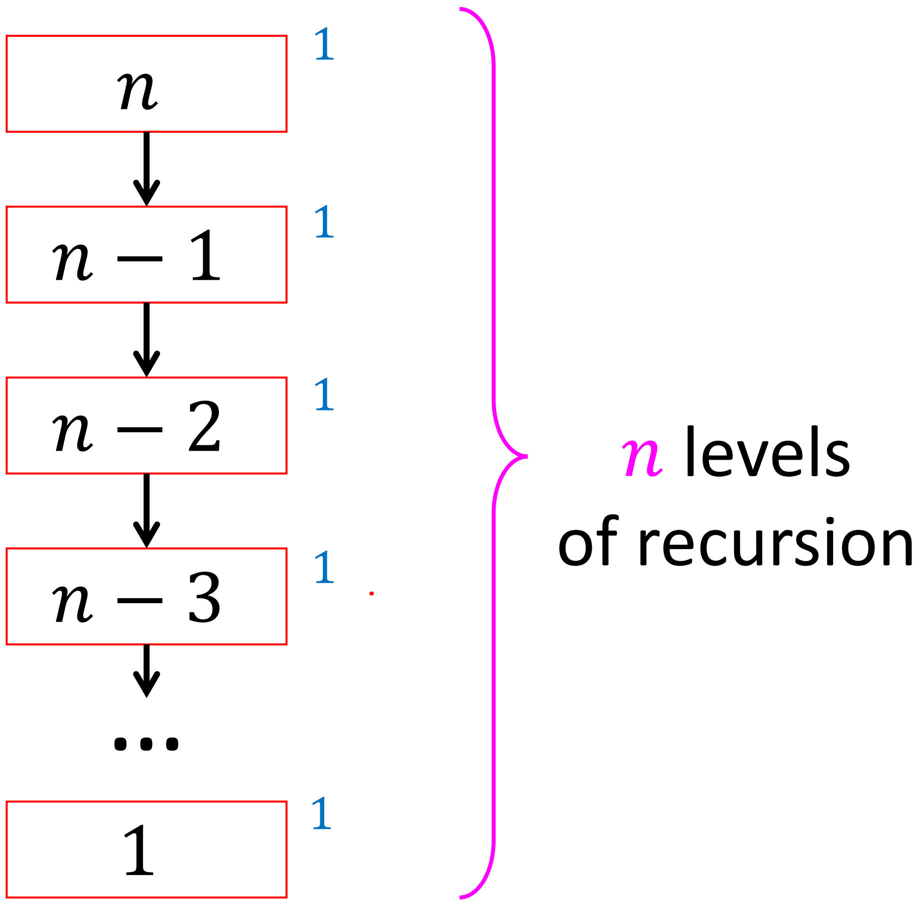
5	8	13	42	75	79	88	90	95	99
0	1	2	3	4	5	6	7	8	9

```
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);
    }
}
```

Make our method "prettier"

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

- 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

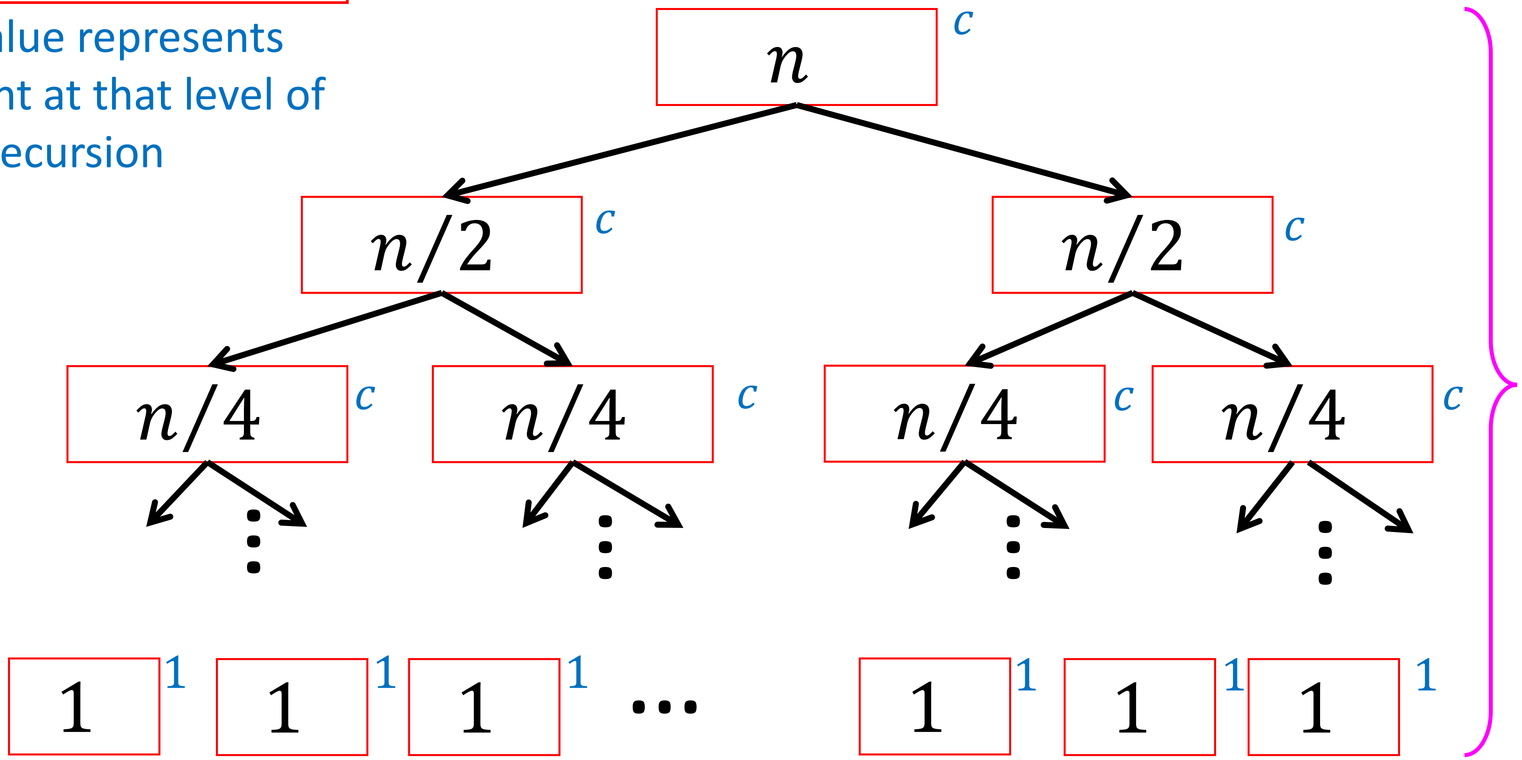
```
public int sum(int[] list){
    return sum_helper(list, 0, list.size);
}
private int sum_helper(int[] list, int low, int high){
    if (low == high){ return 0; }
    if (low == high-1){ return list[low]; }
    int middle = (high+low)/2;
    return sum_helper(list, low, middle) + sum_helper(list,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

Tree Method

Red box represents a problem instance

Blue value represents time spent at that level of recursion

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



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

$\log_2 n$ levels of recursion

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

Recursive List Summation

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

$$= c \cdot \sum_{i=1}^{\log_2 n} 2^i$$

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

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

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

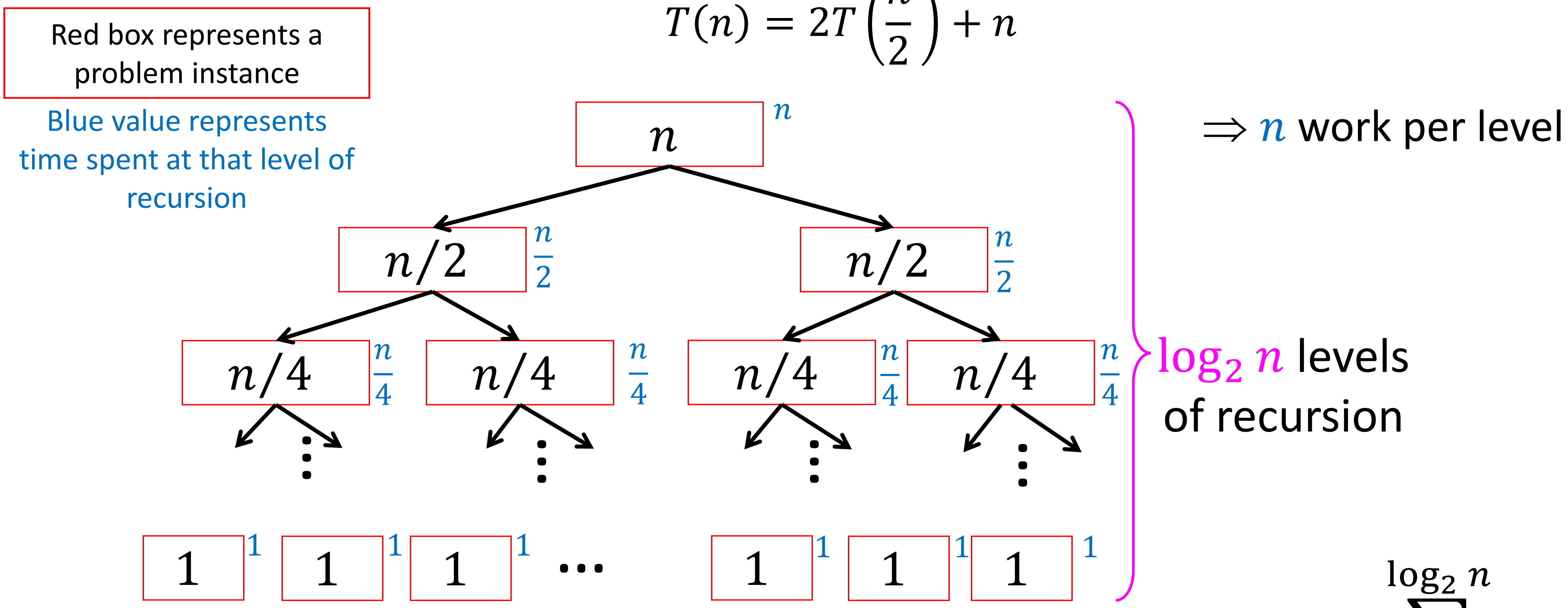
- Recurrence looks like $T(n) = aT\left(\frac{n}{b}\right) + f(n)$
- Use the recurrence to draw a tree
  - a is the branching factor of the tree (e.g. if $a = 2$ then it's a binary tree)
  - Divide the parent's input size by b to get children's input size
  - Work done per node is given by applying $f(n)$ 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

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



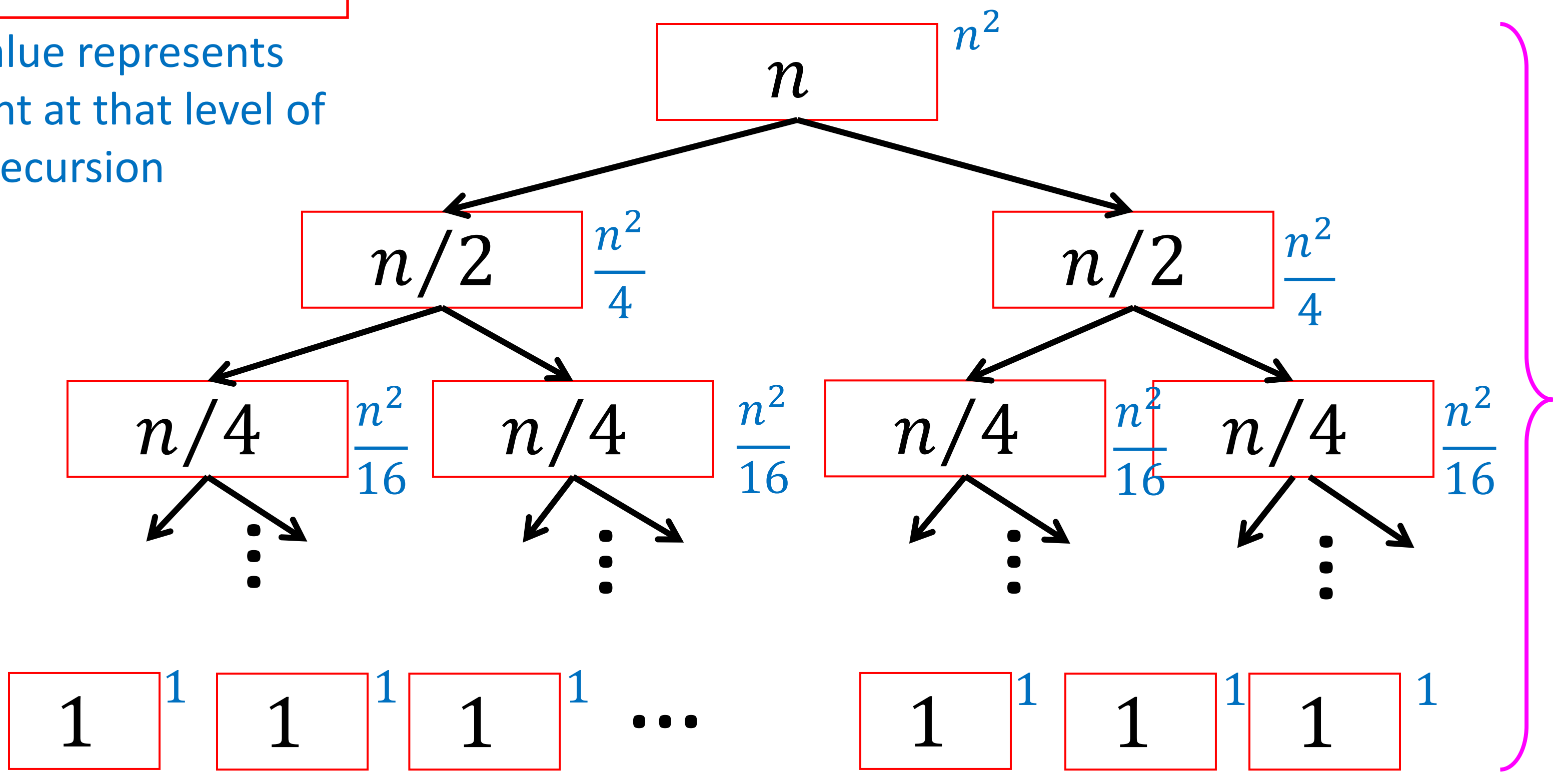
$$T(n) = \sum_{i=1}^{\log_2 n} n$$

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

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

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

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

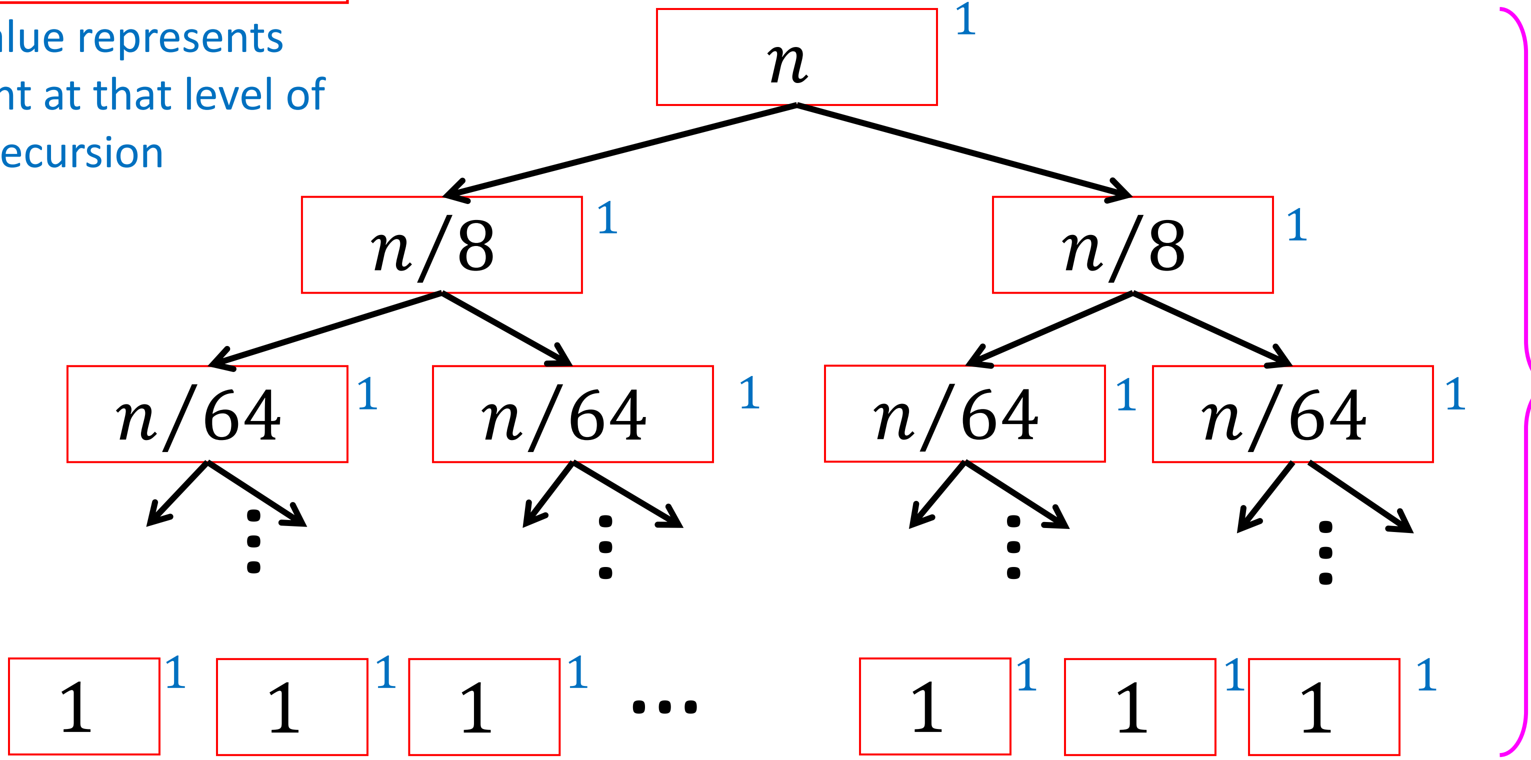
$$= n^2 \cdot \sum_{i=1}^{\log_2 n} \left(\frac{1}{2}\right)^i$$

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

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

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

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

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

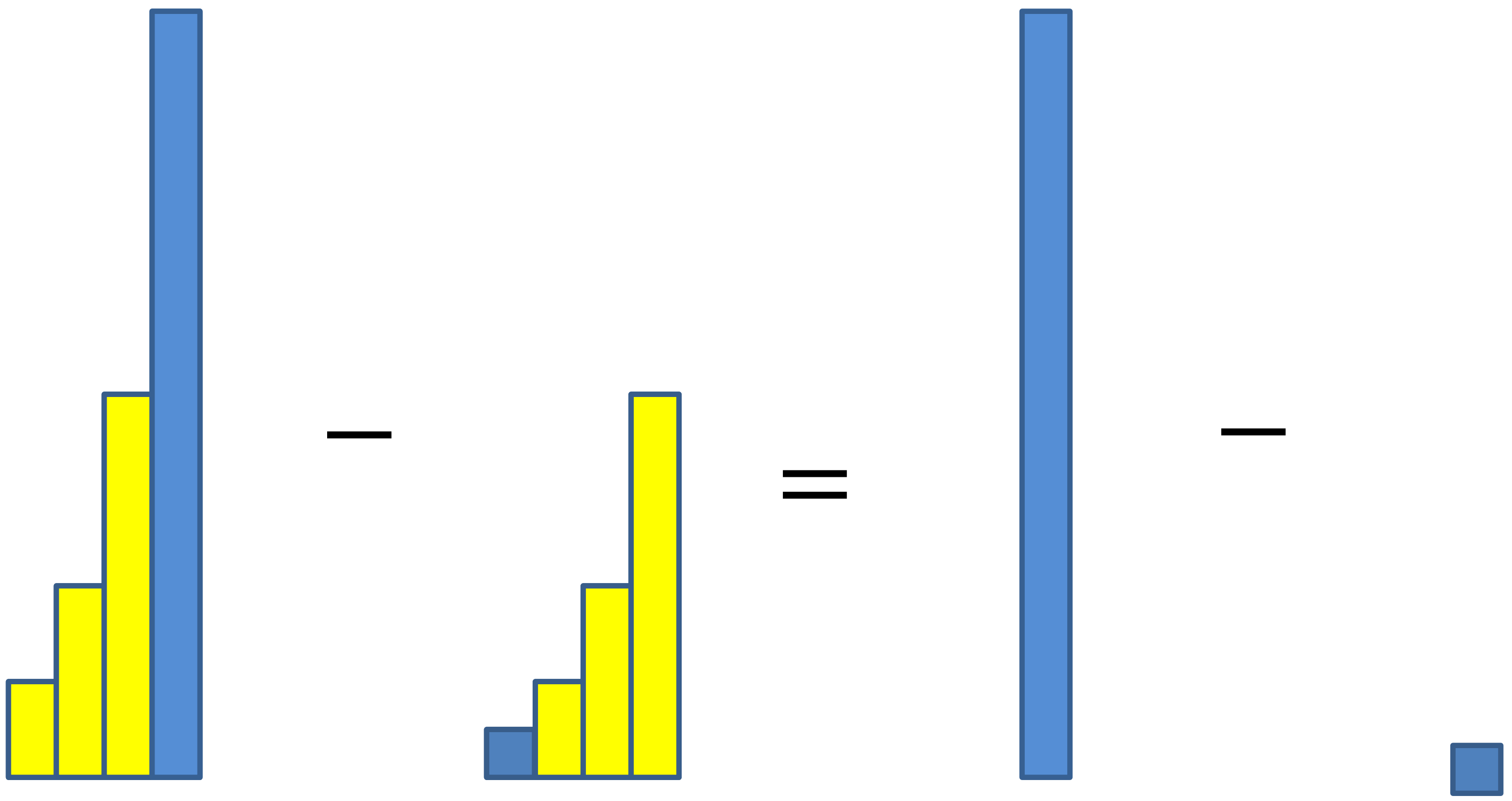
$$= 2^{\log_8 n} - 1$$

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

$$\sum_{i=0}^{L} a^i$$

Finite Geometric Series

If $a > 1$



The series multiplied by a

$(1 + a + a^2 + \cdots + a^L)a$

The series

$(1 + a + a^2 + \cdots + a^L)1$

The next term in the series

a^{L+1}

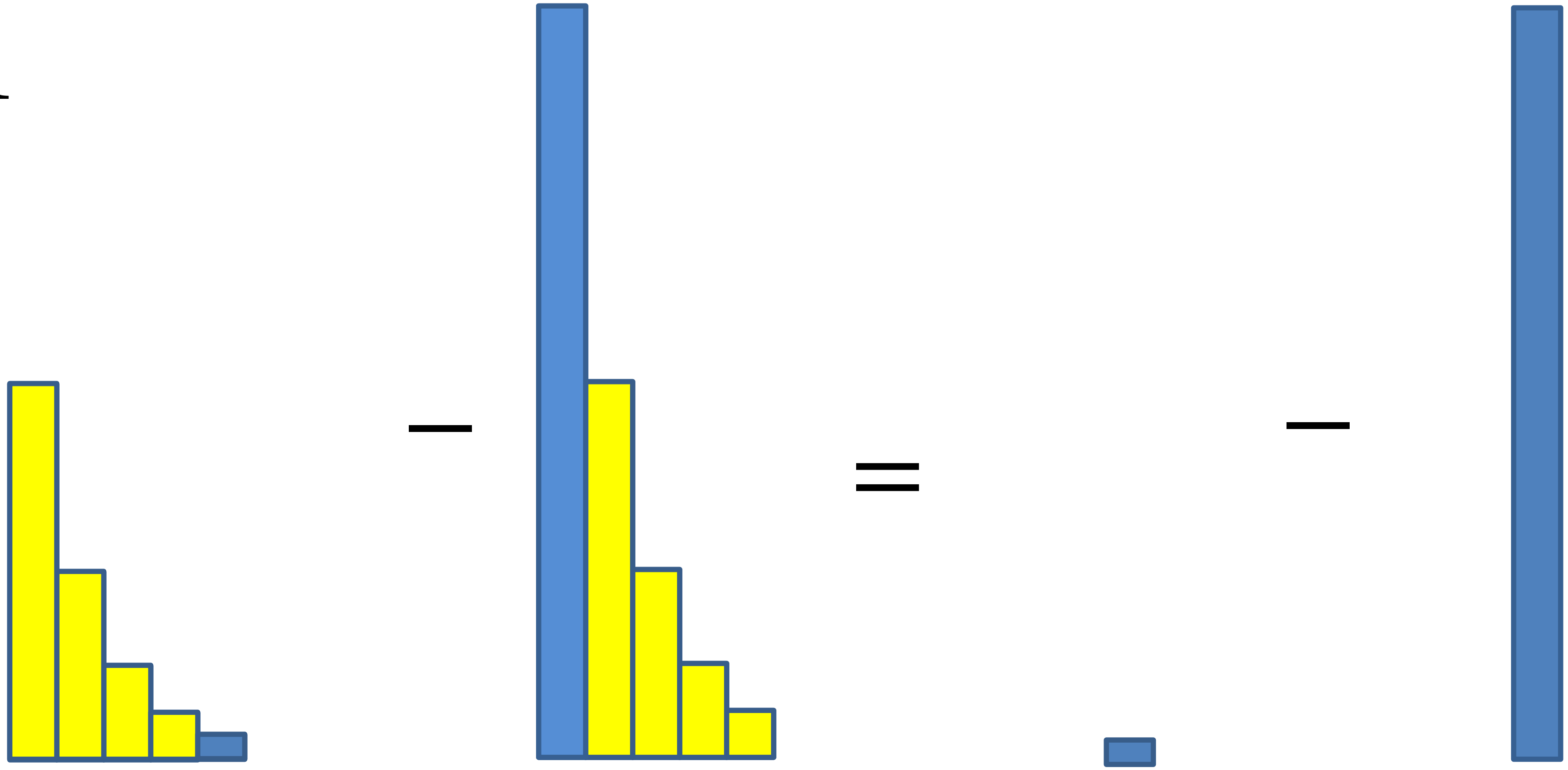
The first term

1

$$\sum_{i=0}^{L} a^i$$

Finite Geometric Series

If $a < 1$



The series multiplied by a

$(1 + a + a^2 + \cdots + a^L)a$

$-$

The series

$(1 + a + a^2 + \cdots + a^L)1$

$=$

The next term in the series

a^{L+1}

$-$

The first term

1

Solve for the series