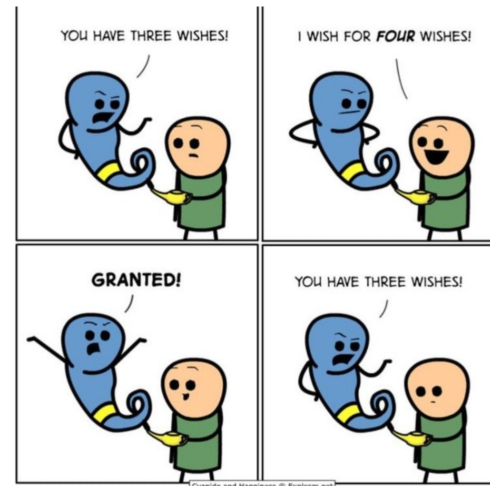
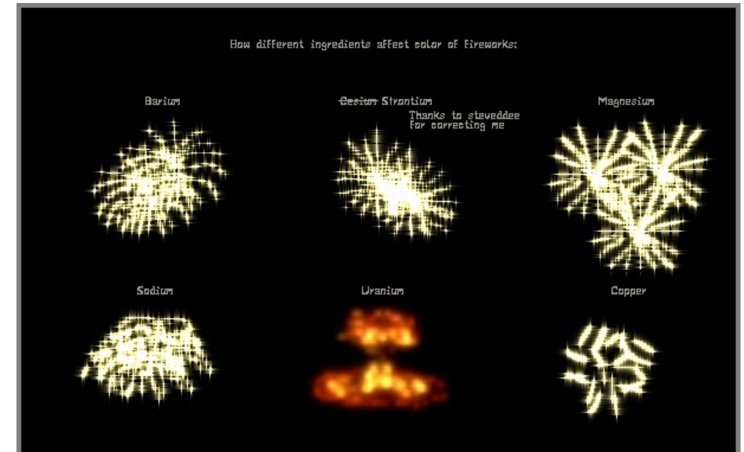
CSE 332: Data Structures & Parallelism
Lecture 5: Recurrences

Arthur Liu

Summer 2022



Announcements



- EX02 due tonight!
- P1 due next Tuesday!

- EX03, EX04 released! Due next Friday

- No class on Monday, Happy 4th of July

Today – Recurrence Relations

- Heaps – BuildHeap
- Recurrence Relations
  1. Writing a recurrence relation
  2. Solving a recurrence relation
- Amortization

Previously: Counting how long iterative functions run

```
b = b + 5
c = b / a
b = c + 100
```

T(n) = 6 operations = O(1)

```
for (i = 0; i < n; i++) {
  sum++;
}
```

T(n) = 1 + 5(n) = O(n)

```
if (j < 5) {
  sum++;
} else {
  for (i = 0; i < n; i++) {
    sum++;
  }
}
```

T(n) = 2 + 5(n) = O(n)

Now: Counting how long recursive functions run

```
int sum(int[] arr){
  return help(arr,0);
}

int help(int[]arr,int i) {
  if(i==arr.length)
    return 0;
  return arr[i] + help(arr,i+1);
}
```

How can we count this function?

If the function runs recursively, our formula for the running time should probably be recursive as well.

Such a formula is a **recurrence**

Example Recurrence Relation

$$T(n) = \begin{cases} T(n-1) + 2 & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

What does this say?

The input to T is the size of the input to the recursive function.

- ***size of input***, so NOT necessarily the actual input value!

If the input to T() is large, the running time depends on the recursive call.

If not we can just use the base case.

Trying Again: Counting how long recursive functions run

```
int sum(int[] arr){
  return help(arr,0);
}

int help(int[]arr,int i) {
  if(i==arr.length)
    return 0;
  return arr[i] + help(arr,i+1);
}
```

$$T(n) = \begin{cases} \\ \end{cases}$$

Trying Again: Counting how long recursive functions run

```
int sum(int[] arr){
  return help(arr,0);
}

int help(int[]arr,int i) {
  if(i==arr.length)
    return 0;
  return arr[i] + help(arr,i+1);
}
```

c_0

$$T(n) = \begin{cases} T(n-1) + 6\ (ish) & \text{if } n > 0 \\ 3\ (ish) & \text{otherwise} \end{cases}$$

c_0

Constants don't really matter so we often replace with c_0, c_1, etc. or even just O(1) or O(n)

Another example

```
Mystery(int n){
      if(n == 1)
            return 1;
      for(int i=0; i < n*n; i++){
            for(int j = 0; j < n; j++){
                  System.out.println("hi!");
            }
      }
      return 3 * Mystery(n/2) + Mystery(n/2)

}
```

$$T(n) = \begin{cases} \\ \end{cases}$$

Another example

```
Mystery(int n){
     if(n == 1)
          return 1;
     for(int i=0; i < n*n; i++){
          for(int j = 0; j < n; j++){
               System.out.println("hi!");
          }
     }
     return 3 * Mystery(n/2) + Mystery(n/2)

}
```

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + c_1 n^3 + c_2 & if\ n > 1 \\ c_0 & otherwise \end{cases}$$

General guidelines

Conceptually, in each recursive call we:

- Perform some amount of work, call it $w(n)$
- Call the function recursively X many times with a smaller problem size
  - *Multiplying a value != calling a value multiple times*

So, if we do $w(n)$ work per step, and reduce the problem size in the next recursive call by 1, we do total work:

- $T(n) = w(n) + X*T(n-1)$

With some base case, like $T(1)=5=O(1)$

You Try

pollev.com/artliu

```
Mystery(int n){
     if(n <= 4)
          return 1;
     for(int i=0; i < n; i++){
          if(i % 3 == 2)
               break;
     }
     return 4 * Mystery(n - 5) + Mystery(n / 2)
}
```

$$T(n) = \begin{cases} \\ \end{cases}$$

You Try

```
Mystery(int n){
     if(n <= 4)
          return 1;
     for(int i=0; i < n; i++){
          if(i % 3 == 2)
               break;
     }
     return 4 * Mystery(n - 5) + Mystery(n / 2)
}
```

$$T(n) = \begin{cases} T(n-5) + T(\frac{n}{2}) + c_1 & if\ n > 4 \\ c_0 & otherwise \end{cases}$$

Aside: Math Pedantics

$$T(n) = \begin{cases} T(n-1) + c_1 & \text{if } n > 0 \\ c_0 & \text{otherwise} \end{cases}$$

IS NOT THE SAME AS

$$T(n) = T(n-1) + c_1$$
$$T(0) = c_0$$

The second one is bad, lazy notation. (That being said, ¯_(ツ)_/¯)

Very cool, what do we do with that?

What's the big-Θ bound?

$$T(n) = \begin{cases} T(n-1) + c_1 & \text{if } n > 0 \\ c_0 & \text{otherwise} \end{cases}$$

```
int sum(int[] arr){
  return help(arr,0);
}

int help(int[]arr,int i) {
  if(i==arr.length)
    return 0;
  return arr[i] + help(arr,i+1);
}
```

Solving Recurrence Relations

$$T(n) = \begin{cases} T(n-1) + c_1 & \text{if } n > 0 \\ c_0 & \text{otherwise} \end{cases}$$

Goal: Get a closed form (ie: no more recursion)

Two (equivalent) strategies:

- Unrolling (algebra)
- Tree method **(recommended)**

Unrolling: Solving Recurrence Relations

$$T(n) = \begin{cases} T(n-1) + c_1 & \text{if } n > 0 \\ c_0 & \text{otherwise} \end{cases}$$

Write it out to get a general form, then solve for base-case:

Unrolling: Solving Recurrence Relations

$$T(n) = \begin{cases} T(n-1) + c_1 & \text{if } n > 0 \\ c_0 & \text{otherwise} \end{cases}$$

Write it out to get a general form:

$$\begin{aligned} T(n) &= c_1 + T(n-1) \\ &= c_1 + c_1 + T(n-2) \\ &= c_1 + c_1 + c_1 + T(n-3) \\ &\dots \\ &= kc_1 + T(n-k) \end{aligned}$$

Solve for $n - k = 0$ (base case), let $k = n$ and plug in

$$= nc_1 + c_0$$

Our Sum Array

Two “obviously” linear algorithms: $T(n) = O(1) + T(n\text{-}1)$

Iterative:

```
int sum(int[] arr){
  int ans = 0;
  for(int i=0; i<arr.length; ++i)
     ans += arr[i];
  return ans;
}
```

Recursive:

- Recurrence is $c + c + \ldots + c$ for n times

```
int sum(int[] arr){
  return help(arr,0);
}
int help(int[]arr,int i) {
  if(i==arr.length)
    return 0;
  return arr[i] + help(arr,i+1);
}
```

What about a binary version of sum?

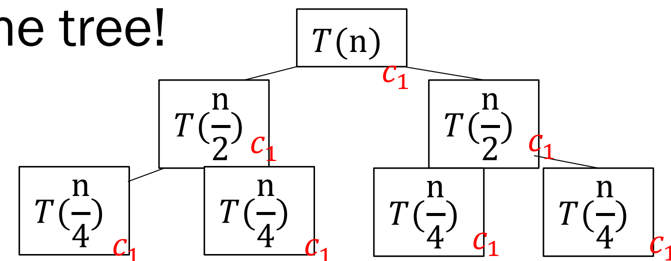
```
int sum(int[] arr){
   return help(arr,0,arr.length);
}
int help(int[] arr, int lo, int hi) {
   if(lo==hi)   return 0;
   if(lo==hi-1) return arr[lo];
   int mid = (hi+lo)/2;
   return help(arr,lo,mid) + help(arr,mid,hi);
}
```

$$T(n) = \begin{cases} 2 * T(n/2) + c_1 & \text{if } n > 1 \\ c_0 & \text{otherwise} \end{cases}$$

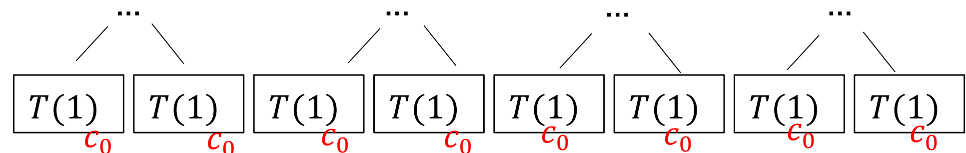
Tree Method General Idea

Create a tree that represents all the work that is done by our recursive method.

Sum up the work of all the nodes in the tree!



$$T(n) = \begin{cases} 2 * T(n/2) + c_1 & \text{if } n > 1 \\ c_0 & \text{otherwise} \end{cases}$$

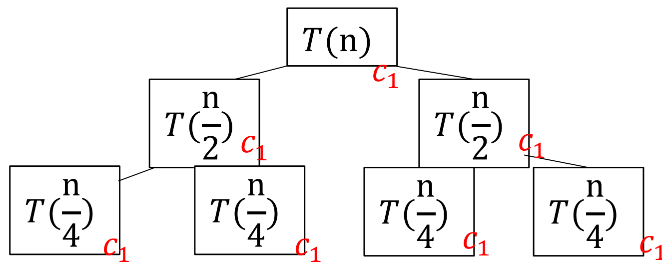


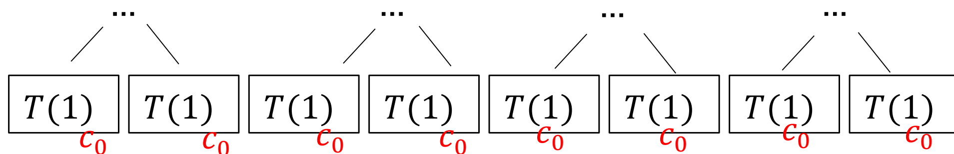
Tree Method

$$T(n) = \begin{cases} 2 * T(n/2) + c_1 & \text{if } n > 1 \\ c_0 & \text{otherwise} \end{cases}$$

Tree Method

$$T(n) = \begin{cases} 2 * T(n/2) + c_1 & \text{if } n > 1 \\ c_0 & \text{otherwise} \end{cases}$$





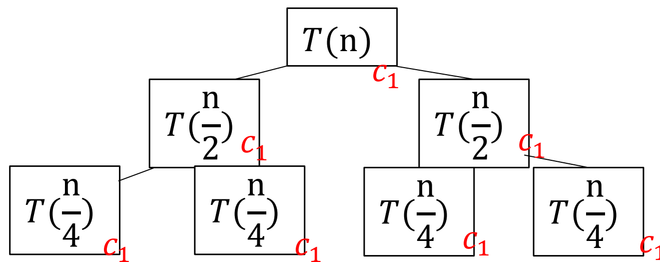
Level	# of nodes	Work/node
0	1	c_1
1	2	c_1
2	4	c_1
3	8	c_1
i	2^i	c_1
$\log_2 n$	n	c_0

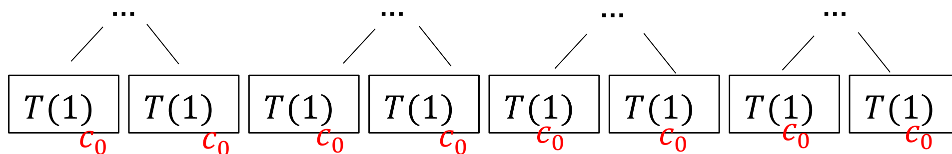
Total work is simply the sum of all the nodes! (== leaf nodes + non-leaf nodes)

$$(\#\ of\ leaves\ * work\ per\ leaf) + \sum_{i=0}^{\#\ of\ levels\ -1} (\#\ of\ nodes\ at\ level) * (work\ per\ node)$$

Tree Method

$$T(n) = \begin{cases} 2 * T(n/2) + c_1 & \text{if } n > 1 \\ c_0 & \text{otherwise} \end{cases}$$





Level	# of nodes	Work/node
0	1	c_1
1	2	c_1
2	4	c_1
3	8	c_1
i	2^i	c_1
$\log_2 n$	n	c_0

Total work is simply the sum of all the nodes! (== leaf nodes + non-leaf nodes)

$$(n * c_0) + \sum_{i=0}^{\log_2 n - 1} (2^i) * (c_1) = nc_0 + (n-1)c_1 = O(n)$$

We *can* unroll but be really, really careful

$T(n) = 2 * T\left(\frac{n}{2}\right) + c_1$

$=$

$$T(n) = \begin{cases} 2 * T(n/2) + c_1 & \text{if } n > 1 \\ c_0 & \text{otherwise} \end{cases}$$

We *can* unroll but be really, really careful

$$T(n) = \begin{cases} 2 * T(n/2) + c_1 & \text{if } n > 1 \\ c_0 & \text{otherwise} \end{cases}$$

$$T(n) = 2 * T\left(\frac{n}{2}\right) + c_1$$

$$= 2 * \left(2\,T\left(\frac{n}{4}\right) + c_1\right) + c_1$$

$$= 2 * 2 * T\left(\frac{n}{4}\right) + 2c_1 + c_1$$

$$= 2 * 2 * \left(2T\left(\frac{n}{8}\right) + c_1\right) + 2c_1 + c_1$$

$$= 2 * 2 * 2 * T\left(\frac{n}{8}\right) + 4c_1 + 2c_1 + c_1$$

Find a pattern! (CAREFUL!)

$$= 2^k * T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k} 2^k * c_1$$

this should look the same as tree method!

Solve for k. We have $\frac{\mathrm{n}}{2^{\mathrm{k}}} = 1$ *so* $k = \log(n)$

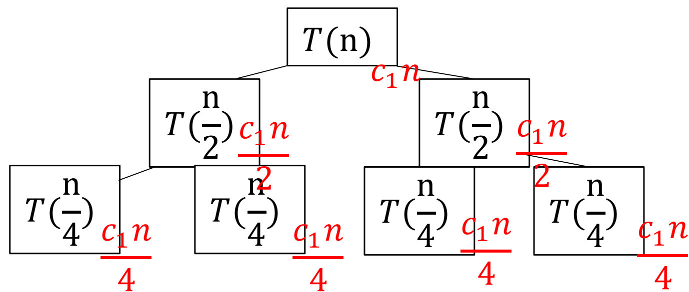
$$= \mathrm{nc_0} + (\mathrm{n} - 1)\mathrm{c_1} = \mathrm{O(n)}$$

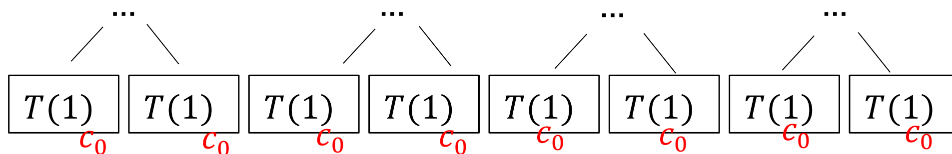
Another Example

$$T(n) = \begin{cases} 2 * T(n/2) + c_1 n & \text{if } n > 1 \\ c_0 & \text{otherwise} \end{cases}$$

Another Example

$$T(n) = \begin{cases} 2 * T(n/2) + c_1 n & \text{if } n > 1 \\ c_0 & \text{otherwise} \end{cases}$$





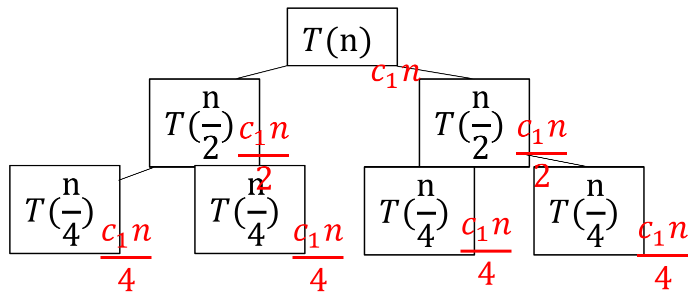
Level	# of nodes	Work/node
0	1	$c_1 n$
1	2	$\frac{c_1 n}{2}$
2	4	$\frac{c_1 n}{4}$
3	8	$\frac{c_1 n}{8}$
i	2^i	$\frac{c_1 n}{2^i}$
$\log_2 n$	n	c_0

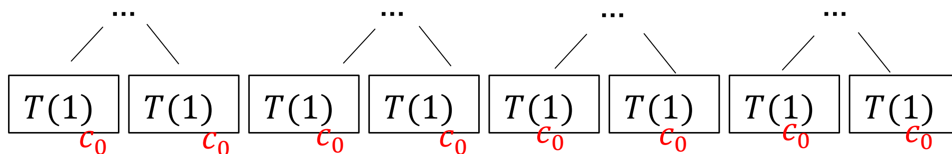
Total work is simply the sum of all the nodes! (== leaf nodes + non-leaf nodes)

$$(\#\ of\ leaves\ * work\ per\ leaf) + \sum_{i=0}^{\#\ of\ levels\ -1} (\#\ of\ nodes\ at\ level) * (work\ per\ node)$$

Another Example

$$T(n) = \begin{cases} 2 * T(n/2) + c_1 n & \text{if } n > 1 \\ c_0 & \text{otherwise} \end{cases}$$





Level	# of nodes	Work/node
0	1	$c_1 n$
1	2	$\frac{c_1 n}{2}$
2	4	$\frac{c_1 n}{4}$
3	8	$\frac{c_1 n}{8}$
i	2^i	$\frac{c_1 n}{2^i}$
$\log_2 n$	n	c_0

Total work is simply the sum of all the nodes! (== leaf nodes + non-leaf nodes)

$$(nc_0) + \sum_{i=0}^{\log_2 n - 1} (2^i) * (\frac{c_1 n}{2^i}) = (nc_0) + \sum_{i=0}^{\log_2 n - 1} (c_1 n) = nc_0 + n\log(n)c_1 = O(n log(n))$$

Parallelism teaser

- But suppose we could do two recursive calls *at the same time*
  - *Like having a friend do half the work for you!*

```
int sum(int[]arr){
   return help(arr,0,arr.length);
}
int help(int[]arr, int lo, int hi) {
   if(lo==hi)   return 0;
   if(lo==hi-1) return arr[lo];
   int mid = (hi+lo)/2;
   return help(arr,lo,mid) + help(arr,mid,hi);
}
```

- If you have as many "friends of friends" as needed, the recurrence is now $T(n) = O(1) + 1T(n/2)$
  - $O(\log n)$: same recurrence as for `find`

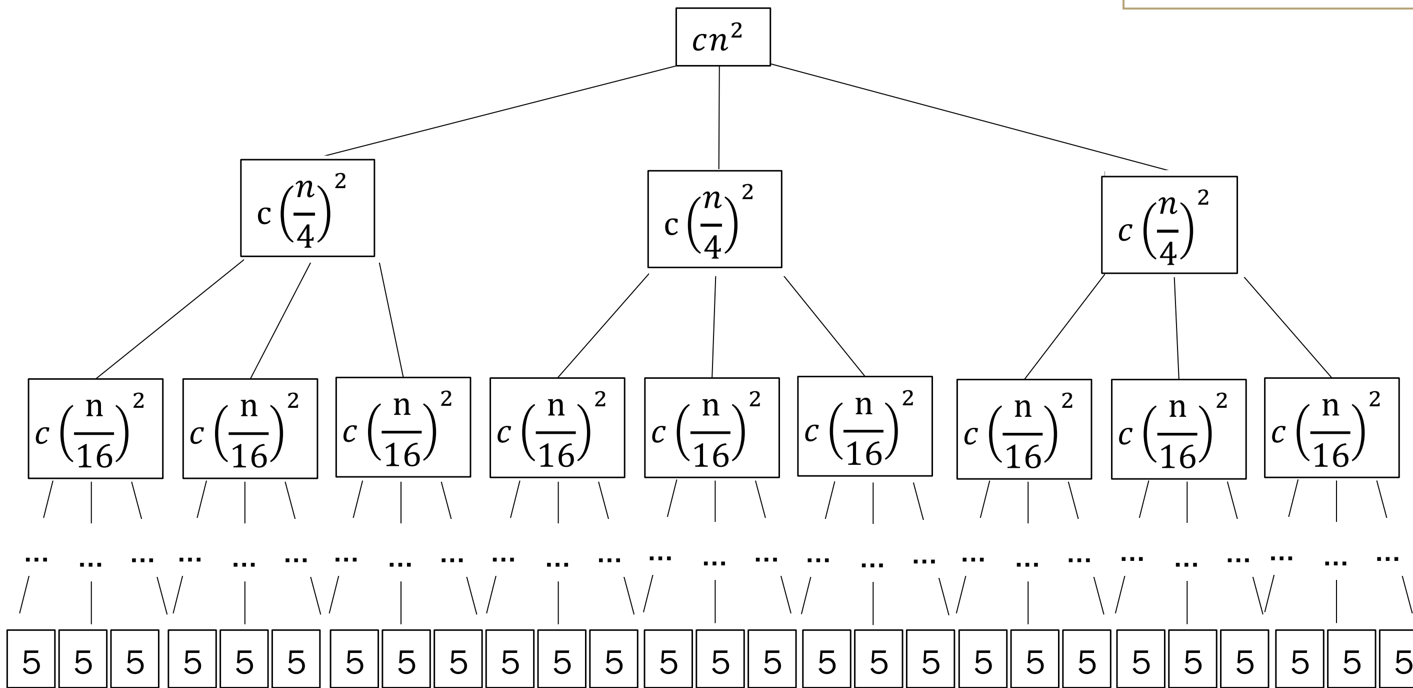
Really common recurrences

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

$T(n) = O(1) + T(n/2)$	logarithmic	$O(\log n)$	binary search
$T(n) = O(1) + 2T(n/2)$	linear	$O(n)$	recursive “binary” sum
$T(n) = O(1) + T(n-1)$	linear	$O(n)$	recursive sum
$T(n) = O(n) + T(n-1)$	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	$O(n \log n)$	merge sort

More: Solving Recurrences III

$$T(n) = \begin{cases} 5 \text{ when } n \leq 4 \\ 3T\left(\frac{n}{4}\right) + cn^2 \text{ otherwise} \end{cases}$$



Answer the following questions:

1. What is input size on level i?
2. Number of nodes at level i?
3. Work done at recursive level i?
4. Last level of tree?
5. Work done at base case?
6. What is sum over all levels?

Solving Recurrences III

$$T(n) = \begin{cases} 5 \text{ when } n \le 4 \\ 3T\left(\frac{n}{4}\right) + cn^2 \text{ otherwise} \end{cases}$$

1. Input size on level i? $\frac{n}{4^i}$

2. How many calls on level i? 3^i

3. How much work on level i? $3^i c\left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i cn^2$

4. What is the last level? When $\frac{n}{4^i} = 4 \rightarrow \log_4 n - 1$

5. A. How much work for each leaf node? 5

B. How many base case calls? $3^{\log_4 n - 1} = \frac{3^{\log_4 n}}{3}$ = $\frac{n^{\log_4 3}}{3}$

power of a log
$x^{\log_b y} = y^{\log_b x}$

Level (i)	Number of Nodes	Work per Node	Work per Level
0	1	cn^2	cn^2
1	3	$c\left(\frac{n}{4}\right)^2$	$\frac{3}{16}cn^2$
2	3^2	$c\left(\frac{n}{4^2}\right)^2$	$\left(\frac{3}{16}\right)^2 cn^2$
i	3^i	$c\left(\frac{n}{4^i}\right)^2$	$\left(\frac{3}{16}\right)^i cn^2$
Base = $\log_4 n - 1$	$3^{\log_4 n - 1}$	5	$\left(\frac{5}{3}\right)n^{\log_4 3}$

6. Combining it all together…

$$T(n) = \sum_{i=0}^{\log_4 n - 2} \left(\frac{3}{16}\right)^i cn^2 + \left(\frac{5}{3}\right)n^{\log_4 3}$$

Solving Recurrences III

7. Simplify…

$$T(n) = \sum_{i=0}^{\log_4 n - 2} \left(\frac{3}{16}\right)^i cn^2 + \left(\frac{5}{3}\right) n^{\log_4 3}$$

factoring out a constant

$$\sum_{i=a}^{b} cf(i) = c \sum_{i=a}^{b} f(i)$$

$$T(n) = cn^2 \sum_{i=0}^{\log_4 n - 2} \left(\frac{3}{16}\right)^i + \left(\frac{5}{3}\right) n^{\log_4 3}$$

finite geometric series

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

Closed form:

$$T(n) = cn^2 \left(\frac{\frac{3}{16}^{\log_4 n - 1} - 1}{\frac{3}{16} - 1} \right) + \left(\frac{5}{3}\right) n^{\log_4 3}$$

If we're trying to prove upper bound…

$$T(n) \leq cn^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i + 4n^{\log_4 3}$$

infinite geometric series

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

when -1 < x < 1

$$T(n) \leq cn^2 \left(\frac{1}{1 - \frac{3}{16}} \right) + \left(\frac{5}{3}\right) n^{\log_4 3}$$

$$T(n) \in O(n^2)$$