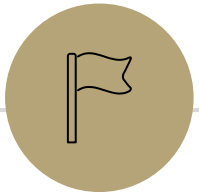
Algorithm Analysis 3 Amortization; Recurrences

CSE 332 Spring 2025
Lecture 6



Amortization

Amortization

How much does housing cost per day in Seattle?

Well, it depends on the day.

The day rent is due, it's $1800.

The other days of the month it's free.

Amortization

Amortization is an **accounting analysis**. It's a way to reflect the fact that even though the "first of the month" is very expensive, the reason that it's very expensive is that it's taking on responsibility for all the other days.

If we distributed the cost equally across the days, (because all days *should* be equally responsible), we "amortize" the cost.

Amortization

AMORTIZED

It costs $1800/month (which we pay once)

So the cost per day is $\frac{1800}{30} = 60$.

Good answer if the question is "what does my daily pay need to be to afford housing?"

UNAMORTIZED

On the first it costs $1800.

Every other day of the month it costs $0

Good answer if the question is "how much do I need to keep in my bank account so it doesn't get overdrawn?"

Amortization

What's the worst case for `enqueue` into an array-based queue?

- The running time is $O(n)$ when we need to resize, and $O(1)$ otherwise.

Is $O(n)$ a good description of the worst-case behavior?

Imagine you said:
"In the worst-case, rent costs $1800 per day. There are 30 days this month, so I need to set aside $30 \cdot 1800 = \$54{,}000$ in my budget; that's the worst-case for the month"

Or you said on Apr. 30, "rent costs $60/day, it's fine that I have only $70 in my bank account"

- Both of these are silly!

Amortization

AMORTIZED

It takes $O(n)$ time to resize once, the next $n-1$ calls take $O(1)$ time each.

So the cost per operation is

$$\frac{O(n)+[n-1]O(1)}{n} = O(1)$$

Good answer if the question is "what will happen when I do many insertions in a row?"

UNAMORTIZED

The resize will take $O(n)$ time. That's the worst thing that could happen.

Good answer if the question is "how long might one (unlucky) user need to wait on a single insertion?"

Amortization

The most common application of amortized bounds is for insertions/deletions and data structure resizing.

Let's see why we always do that doubling strategy.

How long in total does it take to do m insertions?

We might need to double a bunch, but the total resizing work is at most $O(m)$

And the regular insertions are at most $\mathrm{m} \cdot O(1) = O(m)$

So m insertions take $O(m)$ work total

Or amortized $\frac{O(m)}{m} = O(1)$ time.

Total Resizing work

For m insertions, the biggest the array could be is $2m$ (if m is arbitrarily large). So resizing will make arrays of size

$2m, m, \frac{m}{2}, \frac{m}{4}, \ldots$ down to whatever the starting point was.

Work is $c2m + cm + \frac{cm}{2} + \frac{cm}{4} + \cdots$ down to $c \cdot$(starting size)

Total work?

$\sum_{i=0}^{\log(m)} c \cdot \frac{2m}{2^i} = 2mc \cdot \sum_{i=0}^{\log(m)} 2^{-i} \leq 2mc \cdot \sum_{i=0}^{\infty} 2^{-i} = 4mc = O(m)$

Total Resizing work

For m insertions, the biggest the array could be is $2m$ (if m is arbitrarily large). So resizing will make arrays of size

$2m, m, \frac{m}{2}, \frac{m}{4}, \ldots$ down to whatever the starting point was.

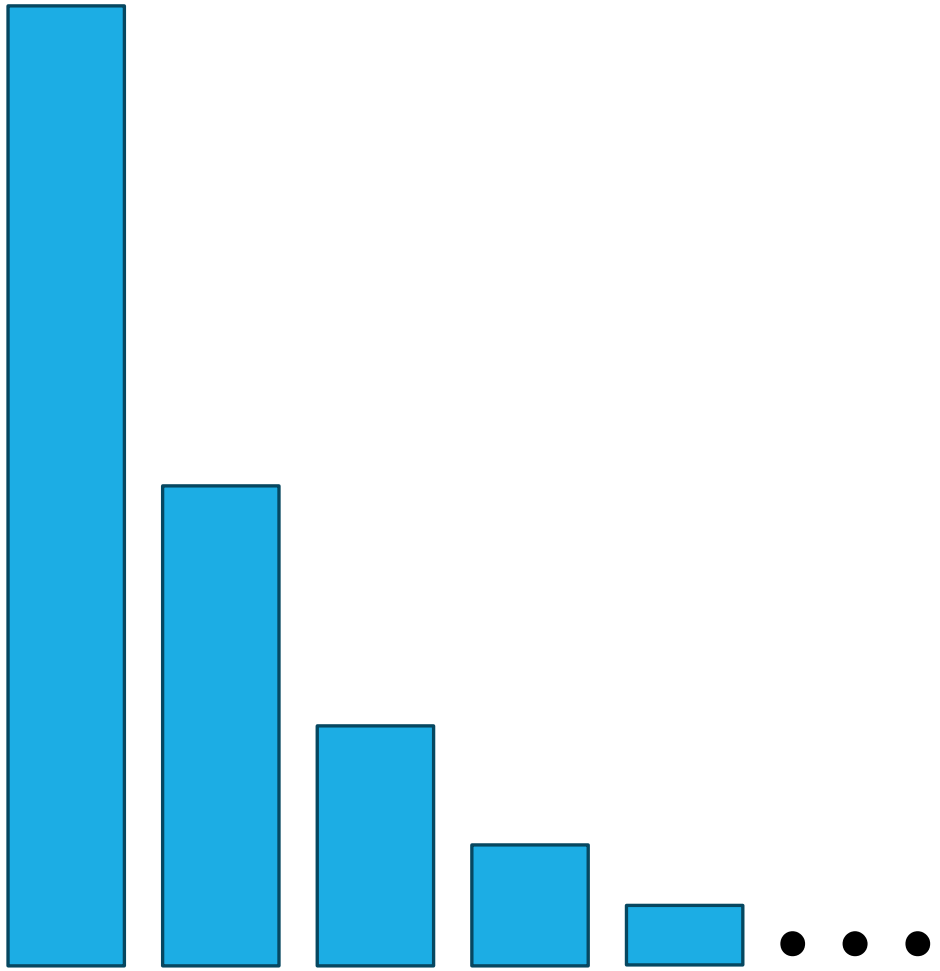
Work is $c2m + cm + \frac{cm}{2} + \frac{cm}{4} + \cdots$ down to $c \cdot$(starting size)

Total work?

$\sum_{i=0}^{\log(m)} c \cdot \frac{2m}{2^i} = 2mc \cdot \sum_{i=0}^{\log(m)} 2^{-i} \leq 2mc \cdot \sum_{i=0}^{\infty} 2^{-i} = 4mc = O(m)$

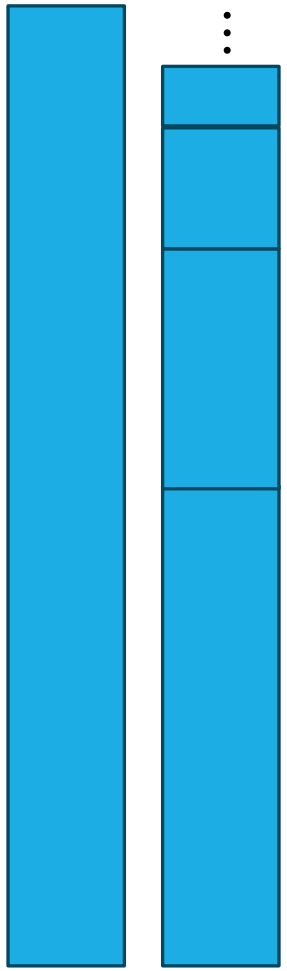
Summation Intuition

We'll see the summation $\sum_{i=0}^{\max} \frac{\text{constant}}{2^i}$ a lot. Why does it converge?



Summation Intuition

We'll see the summation $\sum_{i=0}^{\max} \frac{\text{constant}}{2^i}$ a lot. Why does it converge?



Every term in the summation fills half of the gap and leaves half the gap (because it's half as big).
but then the next term will fill only half the gap again (because it's half as big).

Half the total is in the first term, half the remaining total is in the next, …

Amortization

Why do we double? Why not increase the size by 10,000 each time we fill up?

How much work is done on resizing to get the size up to m?

Will need to do work on order of current size every 10,000 inserts

$$\sum_{i=0}^{\frac{m}{10000}} 10000i \approx 10{,}000 \cdot \frac{m^2}{10{,}000^2} = O(m^2)$$

The other inserts do $O(m)$ work total.

The amortized cost to insert is $O\left(\frac{m^2}{m}\right) = O(m)$.

Much worse than the $O(1)$ from doubling!

Amortization vs. Average-Case

Amortization and “average/best/worst” case are independent properties (you can have un-amortized average-case, or amortized worst-case, or un-amortized worst-case, or …).

Average case asks: “if I selected a possible input on random, how long would my code take?” (compare to worst-case: “if I select the worst value…”)

Amortized or not is “do we care about how much our bank account changes on one day or over the entire month?” (do we care about the running time of individual calls or only what happens over a sequence of them?)

Why use (or don't use) amortized analysis?

Worst Case	Amortized	Unamortized
`Enqueue` (circular array)	$\Theta(1)$	$\Theta(n)$

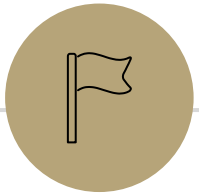
The appropriate analysis depends on your situation (and often it's worth knowing both).

A common use of data structures is as part of an algorithm.

- E.g., I'm trying to process everything in a data set, I insert everything into the data structure, remove them one at a time.
- In that case, we almost always want amortized analysis (we care about when the full analysis is done, not when we go from 49% done to 50% done).

But sometimes you care about individual calls

- Your data structure is feeding another process that the user is watching in real-time.



O, Omega, Theta vs. best/worst

O, Ω, Θ vs. Best, Worst, Average

It's a common misconception that $\Omega()$ is "best-case" and $O()$ is "worst-case". This is a misconception!!

$O()$ says "the complexity of this algorithm is at most" (think $\leq$)

$\Omega()$ says "the complexity of this algorithm is at least" (think $\geq$)

You can use $\leq$ on worst-case or best case; you can use $\geq$ on worst-case or best-case.

Best/Worst/Average say "what function f am I analyzing?"

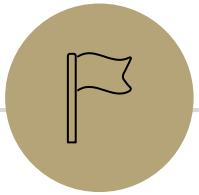
O, Ω, Θ say "let me summarize what I know about f, it's $\leq, \geq, =$..."

Some Example Sentences

	O	Ω	Θ
Best-Case Analysis	In the best-case, linear search will take at most as much time as binary search ever takes. That is, it's $O(\log n)$.	In the best case, linear search still has to look at array index 0; those operations still take time, so you will take at least $\Omega(1)$ time.	In the best case, linear-search is both $O(1)$ and $\Omega(1)$ so it is $\Theta(1)$.
Worst-Case Analysis	In the worst-case, binary search will take at most as much time as linear search ever takes. That is, it's $O(n)$.	In the worst-case, linear search will check at least as many locations in the array as binary search does in the worst-case, so it will take at least $\Omega(\log n)$ time	In the worst-case, binary search takes $\Theta(\log n)$ time. In the worst-case, linear search takes $\Theta(n)$ time.

Why Might you use it?

	O	Ω	Θ
Best-Case Analysis	In the best case, my algorithm is pretty good, it takes at most $O(time)$	Even in the best-case, this algorithm still takes a while; it takes at least $\Omega(time)$	In the best case, my algorithm takes exactly $\Theta(time)$
Worst-Case Analysis	Even in the worst-case my algorithm, isn't that bad! It takes at most $O(time)$ time in the worst-case	In the worst-case, there's still a lot of work the algorithm has to do; it takes at least $\Omega(time)$	In the worst case, my algorithm takes exactly $\Theta(time)$



Recursive Code Analysis

Calculating Running Times

Here's some code for calculating the length of a linked list:

What's its running time?

```
Length(Node curr){
    if(curr.next == null)
        return 1;
    return 1 + Length(curr.next);
}
```

We can analyze all the "non-recursive" work like usual

What about the recursive work?

Writing a Recurrence

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

Such a formula is a **recurrence**.

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

What does this say?

The input to T is the size of the input to the Length.

- If the input to $T()$ is large, the running time depends on the recursive call.
- If not, we can just use the base case.

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 Mystery(n/2)
}
```

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + c \cdot n^3 & \\ 1 & \text{if } n = 1 \end{cases}$$

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 Mystery(n/2)
}
```

$$T(n) = \begin{cases} T(n/2) + n^3 + n^2 + 1 & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

Try It On Your Own

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

Try It On Your Own

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

$$T(n) = \begin{cases} T(n-5)+3 & \text{if } n > 4 \\ 1 & \text{otherwise} \end{cases}$$

What Do We Do With That

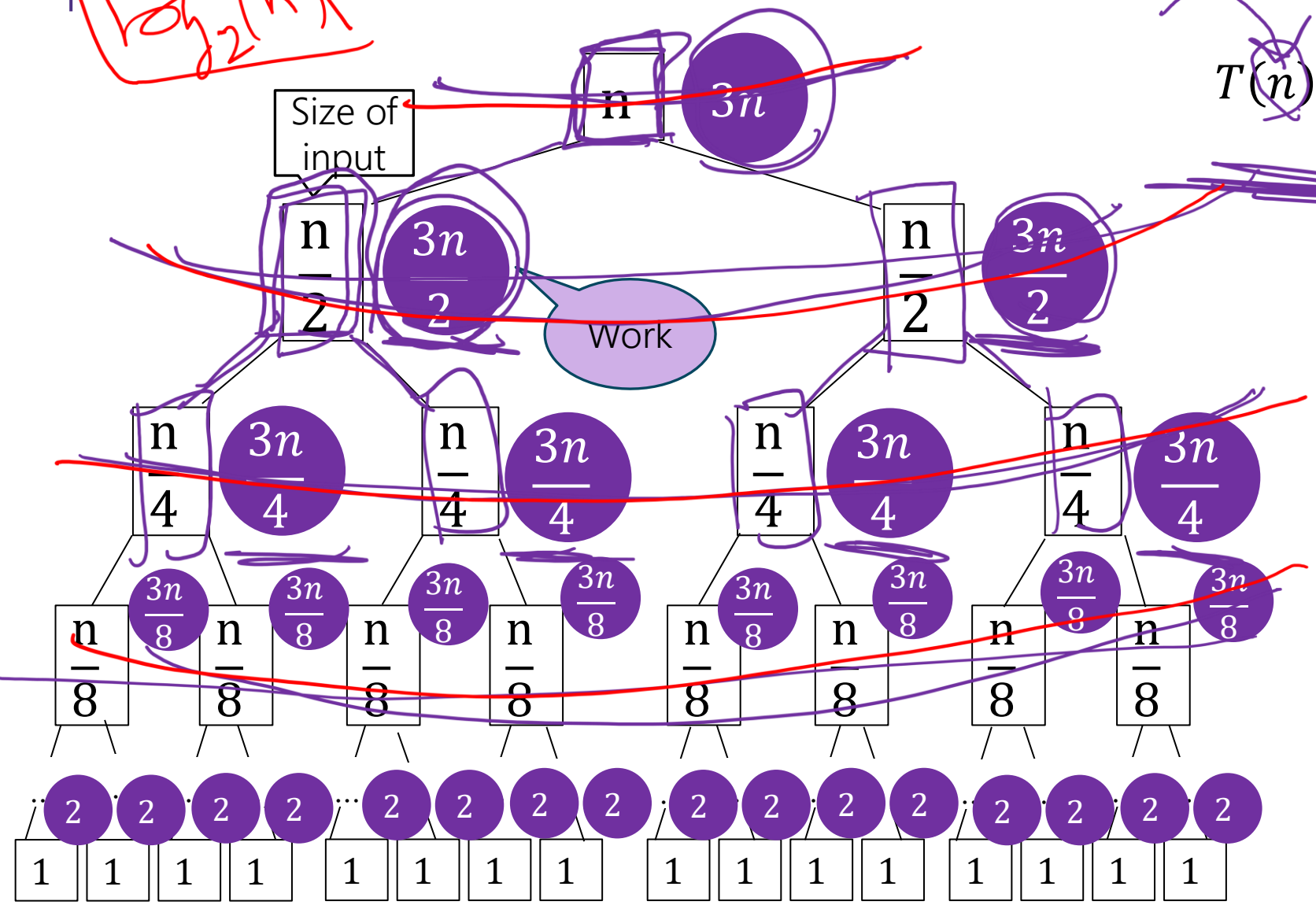
That's nice. So what's the big-Θ bound?

Tree Method

Idea:

- Since we're making recursive calls, let's just draw out a tree, with one node for each recursive call.
- Each of those nodes will do some work, and (if they make more recursive calls) have children.
- If we can just add up all the work, we can find a big-Θ bound.

Solving Recurrences II:



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

Tree Method Formulas

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

How much work is done by recursive levels (branch nodes)?

1. What is the input size at level i?
   - i= 0 is overall root level.
2. At each level i, how many calls are there?
3. At each level i, how much work is done??

$$Recursive\ work = \sum_{i=0}^{lastRecursiveLevel} NumNodes(i)WorkPerNode(i)$$

How much work is done by the base case level (leaf nodes)?

4. What is the last level of the tree?
5. What is the work done at the last level?

$$NonRecursive\ work = WorkPerBaseCase \cdot numberCalls$$

6. Combine and Simplify

Tree Method Formulas

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

How much work is done by recursive levels (branch nodes)?

1. What is the input size at level i?
   - i = 0 is overall root level.
2. At each level i, how many calls are there?
3. At each level i, how much work is done??

$$Recursive\ work = \sum_{i=0}^{lastRecursiveLevel} NumNodes(i) WorkPerNode(i)$$

$(n/2^i)$

2^i

$2^i(n/2^i) = 3n$

$$\sum_{i=0}^{\log_2 n - 1} 3 \cdot 2^i \left(\frac{n}{2^i}\right)$$

$3n \cdot (\log_2 n)$

How much work is done by the base case level (leaf nodes)?

4. What is the last level of the tree? $(n/2^i) = 1 \rightarrow 2^i = n \rightarrow i = \log_2 n$
5. What is the work done at the last level?

$$NonRecursive\ work = WorkPerBaseCase \cdot numberCalls$$

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

6. Combine and Simplify

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

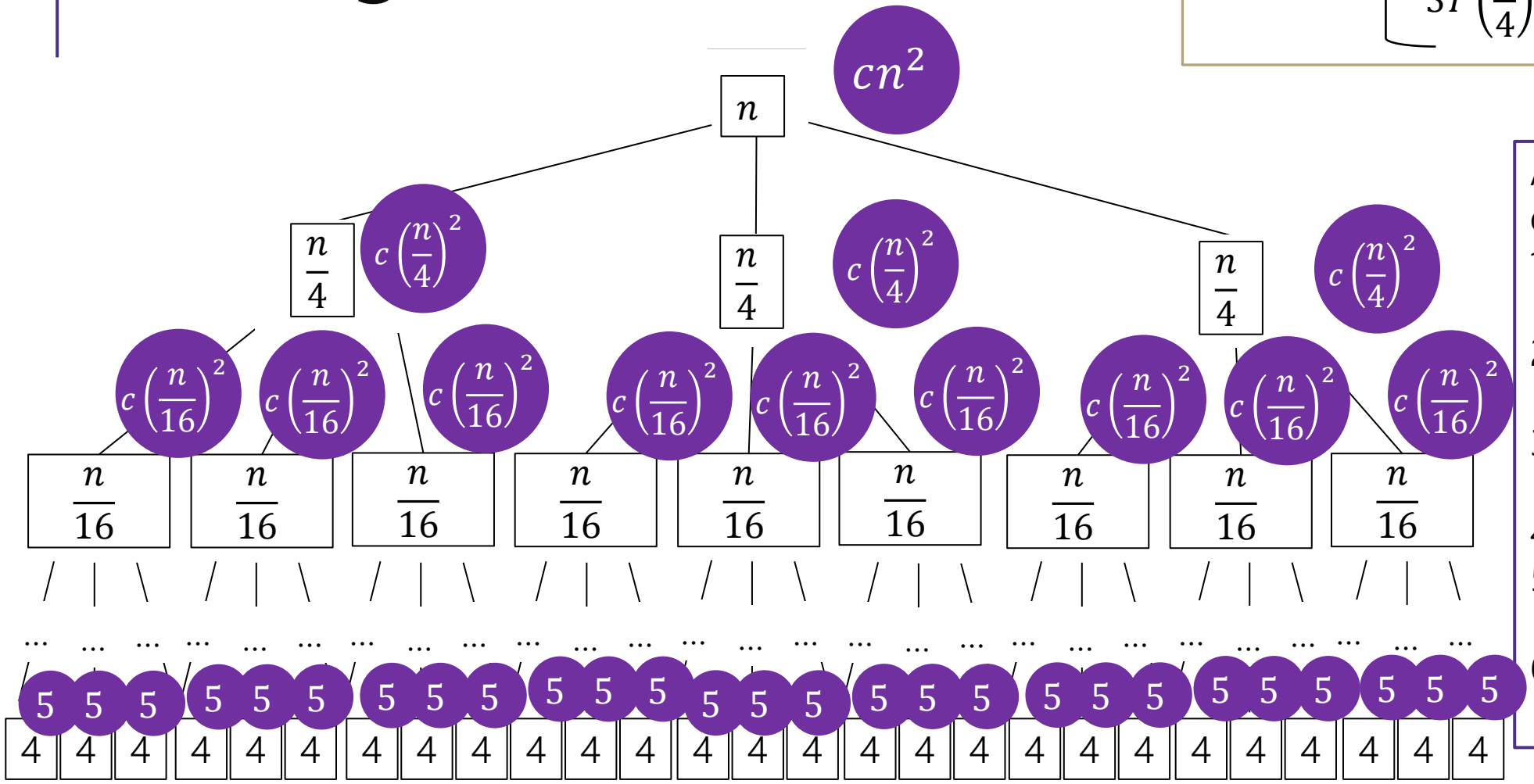
$\Theta(n \log n)$

Let’s try another

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

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

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

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

$$c\left(\frac{n}{4^i}\right)^2$$

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? $3^{\log_4 n - 1} = \frac{3^{\log_4 n}}{3}$

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

$= \frac{n^{\log_4 3}}{3}$

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

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

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

What about an upper bound on:

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

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

Which is $\in \Omega(n^2)$

A Note About Base Cases

Except in very rare circumstances, your code looks like

```
if(n < constant)
     Do something
 else
     Make recursive call(s) and do something.
```

(might have more base cases or edge cases, but core idea is this)

`Do Something` for n<constant is $\Theta(1)$, **even if** `do something` doesn't look constant

(5^2 or 2^{100} are both constants, so n^2 for $n = 5$ is still $\Theta(1)$)

So changing constant

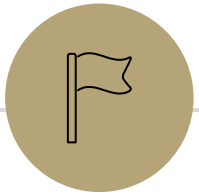
But Don't skip the base case

Sometimes there's so many recursive calls (and so little non-recursive work) that most of the work happens at the base case

Try making the tree for $T(n) = \begin{cases} 3T\left(\frac{n}{2}\right) + 1 \text{ if } n \geq 2 \\ 1 \qquad\qquad \text{otherwise} \end{cases}$

There's $n^{\log\_2(3)} \gg n$ base case nodes, you need to account for that!

For Ex3, we let you choose your base case. Choose $n \geq 2$ or $n > 2$ or $n \geq 1024$ etc. whatever makes the math easier for you. But don't just skip it or you might get the wrong result!



More Practice

Warm Up

Write a recurrence to describe the running time of Mystery.

If you have extra time, find the big-Θ running time.

```
Mystery(int[] arr){
    if(arr.length == 1)
        return arr[0];
    else if(arr.length == 2)
        return arr[0] + arr[1];
    //copies all but first two elements of arr.
    int[] smaller = Arrays.copyOfRange(arr, 2, arr.length);
    return a[0] + a[1] + Mystery(smaller);
}
```

Warm Up

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 & \text{if } n = 2 \\ T(n-2) + 4 & \text{otherwise} \end{cases}$$

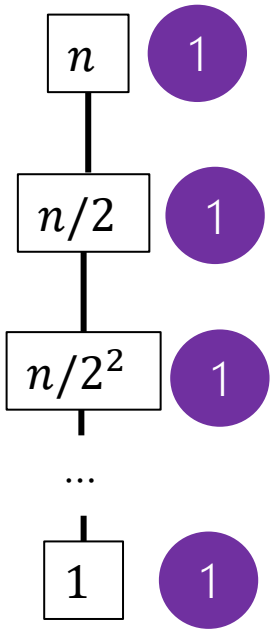
Solving Recurrences I: Binary Search

$$T(n) \equiv \begin{cases} 1 \; when \; n \leq 1 \\ T\left(\frac{n}{2}\right) + 1 \; otherwise \end{cases}$$

0. Draw the tree.
1. What is the input size at level i?
2. What is the number of nodes at level i?
3. What is the work done at recursive level i?
4. What is the last level of the tree?
5. What is the work done at the base case?
6. Sum over all levels (using 3,5).
7. Simplify

Solving Recurrences I: Binary Search

$$T(n) \equiv \begin{cases} 1 \; when \; n \leq 1 \\ T\left(\frac{n}{2}\right) + 1 \; otherwise \end{cases}$$



0. Draw the tree.
1. What is the input size at level i?
2. What is the number of nodes at level i?
3. What is the work done at recursive level i?
4. What is the last level of the tree?
5. What is the work done at the base case?
6. Sum over all levels (using 3,5).
7. Simplify

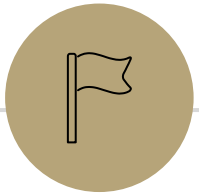
Solving Recurrences I: Binary Search

$$T(n) = \begin{cases} 1 \; when \; n \leq 1 \\ T\left(\frac{n}{2}\right) + 1 \; otherwise \end{cases}$$

Level	Input Size	Work/call	Work/level
0	n	1	1
1	$n/2$	1	1
2	$n/2^2$	1	1
i	$n/2^i$	1	1
$\log_2 n$	1	1	1

0. Draw the tree.
1. What is the input size at level i?
2. What is the number of nodes at level i?
3. What is the work done at recursive level i?
4. What is the last level of the tree?
5. What is the work done at the base case?
6. Sum over all levels (using 3,5).
7. Simplify

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



Extra Amortization Example

A Contrived Example

A `MakesYouWaitList` operates as follows:

When you call `find()`, it does a linear search through an array of n elements to find i. If the index is odd, it spins for $O(n)$ time, if the index is even it spins for $O(n^2)$ time. Additionally, every n^{th} call to find it spins for $O(n^{2.5})$ time. It looks like this:

```
class MakesYouWaitList{
      int callsToFind=0; Object arr[]


find(Object o){
      n=arr.length
      int indexOfI=LinearSearch(o);
      if(indexOfI % 2 == 1)
            for(int k=0; k<n; k++) { }
      else
            for(int k=0; k<n*n; k++) { }
      callsToFind++;
      if(callsToFind == n) {
            callsToFind = 0;
             for(int k=0; k<Math.pow(n,2.5); k++) { }
      }
}
```

All the running times

	Best	Worst	Average
Amortized	$O(n^{1.5})$ Every n operations trigger the last $n^{2.5}$ spin time. No matter what elements are chosen. The best choices (all at even indices) will add n work each, which is a lower order term.	$O(n^2)$ On an odd input we take $O(n^2)$. The $O(n^{1.5})$ we want to assign to each for the big spin-time at the end is a lower order term.	$O(n^2)$On average, half the inputs will be at odd indices and half even, so we'll have: $\frac{n}{2}O(n) + \frac{n}{2}O(n^2) + O(n^{2.5})$ work, which is $O(n^3)$ total. Giving each of the n operations its share we get $O(n^2)$
Unamortized	$O(n)$ as long as the element is stored at an even index and doesn't trigger the resize, we'll get $O(n)$ time.	$O(n^2)$ The worst-case total work for n operations is for all to be odd $n \cdot O(n^2)$, and one to trigger the resize $O(n^{2.5})$. The total is $O(n^3)$, which we distribute equally among the n inserts.	$O(n^2)$ We have a $1/n$ chance of causing the $O(n^{2.5})$ spin, a ½ chance of getting $O(n)$ and ½ of $O(n^2)$, this gives: $\frac{1}{n} \cdot O(n^{2.5}) + \frac{1}{2}O(n) + \frac{1}{2}O(n^2)$ Which gives $O(n^2)$