More Asymptotics

CSE 332 25Sp
Lecture 3

Announcements

	Monday	Tuesday	Wednesday	Thursday	Friday
This Week					TODAY Ex 1 Available
Next Week	Exercise 0 due Ex 2 available				Exercise 1 due

You'll have everything you need for Exercise 1 on today's slides.

Asymptotic Notation

That's a nice formula. But does everything in it matter?

Multiplying by constant factors doesn't matter – let's just ignore them.

Lower order terms don't matter – delete them.

Gives us a "simplified big-O"

$10n \log n + 3n$	$O(n \log n)$
$5n^2 \log n + 13n^3$	$O(n^3)$
$20n \log \log n + 2\, n \log n$	$O(n \log n)$
2^{3n}	$O(8^n)$

Formally Big-O

We wanted to find an upper bound on our algorithm's running time, but

- We don't want to care about constant factors.
- We only care about what happens as n gets large.

The formal, mathematical definition is Big-O.

Big-O

$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$

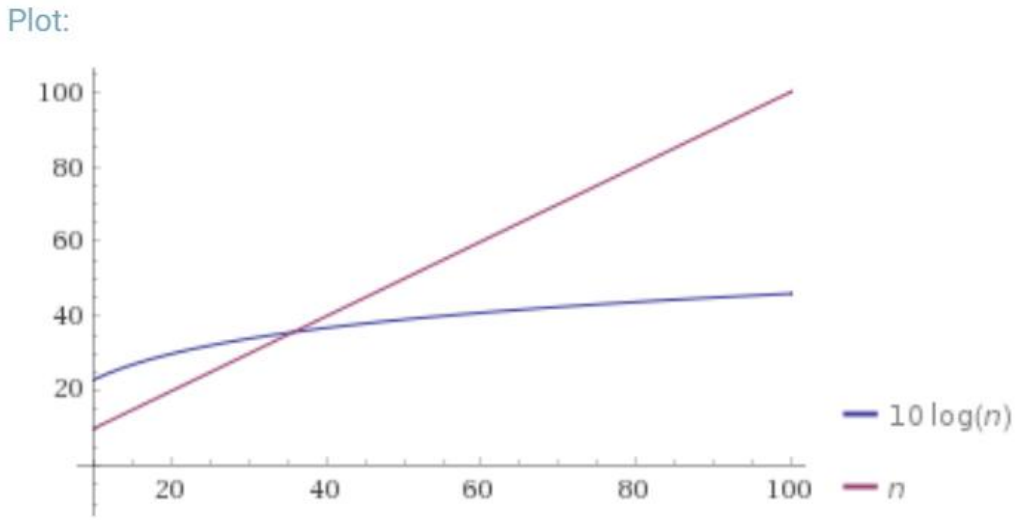
We also say that $g(n)$ "dominates" $f(n)$.

Why is that the definition?

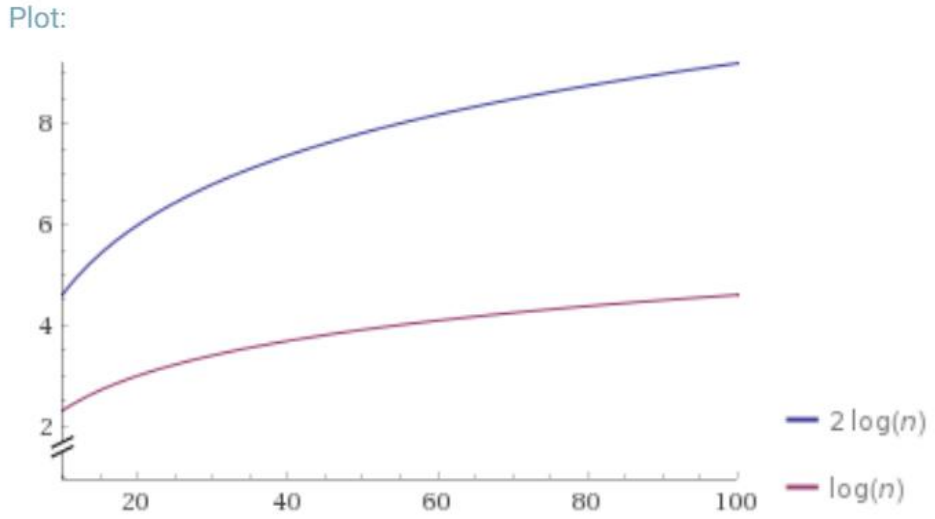
Big-O

$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$

Why n_0?



Why c?



Why Are We Doing This?

You already intuitively understand what big-O means.

Who needs a formal definition anyway?

- We will.

Your intuitive definition and my intuitive definition might be different.

We're going to be making more subtle big-O statements in this class.

- We need a mathematical definition to be sure we're on the same page.

Once we have a mathematical definition, we can go back to intuitive thinking.

- But when a weird edge case, or subtle statement appears, we can figure out what's correct.

Edge Cases

True or False: $10n^2 + 15n$ is $O(n^3)$

[this is an edge case]

It's true! – it fits the definition.

Big-O is just an upper bound. It doesn't have to be a good upper bound.

If we want the best upper bound, we'll ask you for a **tight** big-O bound.

$O(n^2)$ is the tight bound for this example.

It is (usually) technically correct to say your code runs in time $O(n^{n!})$.

- DO NOT TRY TO PULL THIS TRICK ON AN EXAM. Or in an interview.

O, Omega, Theta [oh my?]

Big-O is an **upper bound**

- My code uses at most this many resources (e.g. runs in at most this much time)

Big-Omega is a lower bound

Big-Omega

$f(n)$ is $\Omega(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

$$f(n) \geq c \cdot g(n)$$

Big Theta is "equal to"

Big-Theta

$f(n)$ is $\Theta(g(n))$ if
$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

Viewing O as a class

Sometimes you'll see big-O defined as a family or set of functions.

Big-O (alternative definition)

$O(g(n))$ is the set of all functions $f(n)$ such that there exist positive constants c, n_0 such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$

For that reason, some people write $f(n) \in O\big(g(n)\big)$ where we wrote "$f(n)$ is $O(g(n))$".
Other people write "$f(n) = O\big(g(n)\big)$" to mean the same thing.
We never write $O(5n)$ instead of $O(n)$ – they're the same thing!

It's like writing $\frac{6}{2}$ instead of 3. It just looks weird.

Useful Vocab

The most common running times all have fancy names:

$O(1)$ constant

$O(\log n)$ logarithmic

$O(n)$ linear

$O(n \log n)$ "n log n"

$O(n^2)$ quadratic

$O(n^3)$ cubic

$O(n^c)$ polynomial (where c is a constant)

$O(c^n)$ exponential (where c is a constant)

What's the base of the log?

If I write $\log n$, without specifying a base, I mean $\log_2 n$.

But does it matter for big-O?

Suppose we found an algorithm with running time $\log_5 n$ instead.

Is that different from $O(\log_2 n)$?

No!

$\log_c n = \frac{\log_2 n}{\log_2 c}$ If c is a constant, then $\log_2 c$ is just a constant, and we can hide it inside the $O()$.

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?"

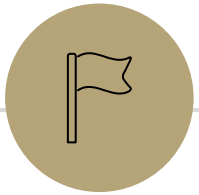
O, Ω, Θ say "let me summarize what I know about f, it's $\leq, \geq, =$..."

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 $\Theta(time)$



Some Log Review

Logs

$\log(n^k) = k \cdot \log(n)$.

$\log(2^n) = n$ (logs and exponents are inverse functions.

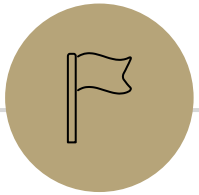
$\log(\log(x))$ is usually written `log log x`

Grows **VERY** slowly (as slowly as $2^{(2^x)}$ grows quickly)

$\log_2(\log_2(\#\text{ atoms in universe})) = \log_2(\log_2(10^{80})) = \log_2(80 \cdot \log_2(10)) \approx 8.054$

Don't confuse with $\log(x) \cdot \log(x)$ usually written $\log^2(x)$.

$\log\log x << \log x << \log^2(x)$ (where << is "asymptotically less than")



Writing Big-O Proofs

Proving Big-O, Formally

Big-O is an $\exists c, n_0 \forall n$ statement.

I.e., an exists statement with a “forall” inside.

How do you prove an exists statement?

How do you prove a for-all statement?

Proving Big-O, Formally

Big-O is an $\exists c, n_0 \forall n$ statement.

I.e., an exists statement with a "forall" inside.

How do you prove an exists statement?

Show the c, n_0 that will work. Give specific values.

How do you prove a for-all statement?

Introduce an arbitrary n.

Using the Definition

Let's show: $10n^2 + 15n$ is $O(n^2)$

Using the Definition

Let's show: $10n^2 + 15n$ is $O(n^2)$

Scratch work:
$10n^2 \leq 10n^2$
$15n \leq 15n^2$ for $n \geq 1$
$10n^2 + 15n \leq 25n^2$ for $n \geq 1$

Proof:
Take $c = 25$ and $n_0 = 1$. For an arbitrary $n \geq n_0$, we have
The inequality $10n^2 \leq 10n^2$ is always true. The inequality $15n \leq 15n^2$ is true for $n \geq 1$, as the right hand side is a factor of n more than the right hand side.
As long as both inequalities are true we can add them, thus
$10n^2 + 15n \leq 25n^2$ holds as long as $n \geq 1$.
This is exactly the inequality we needed to show.

Writing Proofs

Two more examples at the end of the slide deck.

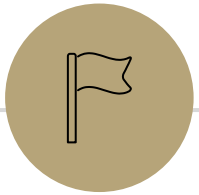
Where did that $c = 25$, $n_0 = 1$ come from?

That was some "scratch work" – the insight isn't explained in the final proof
- You just say "Consider"

Don't try to skip the scratch work when drafting your big-O proofs.
- But it won't appear in your final version.

Be sure you're arguing in correct logical order---you only assert something is true when you know it. Often that's the reverse of the scratch work order.

Don't just choose $c = 10^{10}, n_0 = 10^5$. That will be technically correct, but proofs are acts of communication; that won't convince your reader if they didn't already believe the claim; smaller values with algebra are more convincing than overkill.



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?

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.

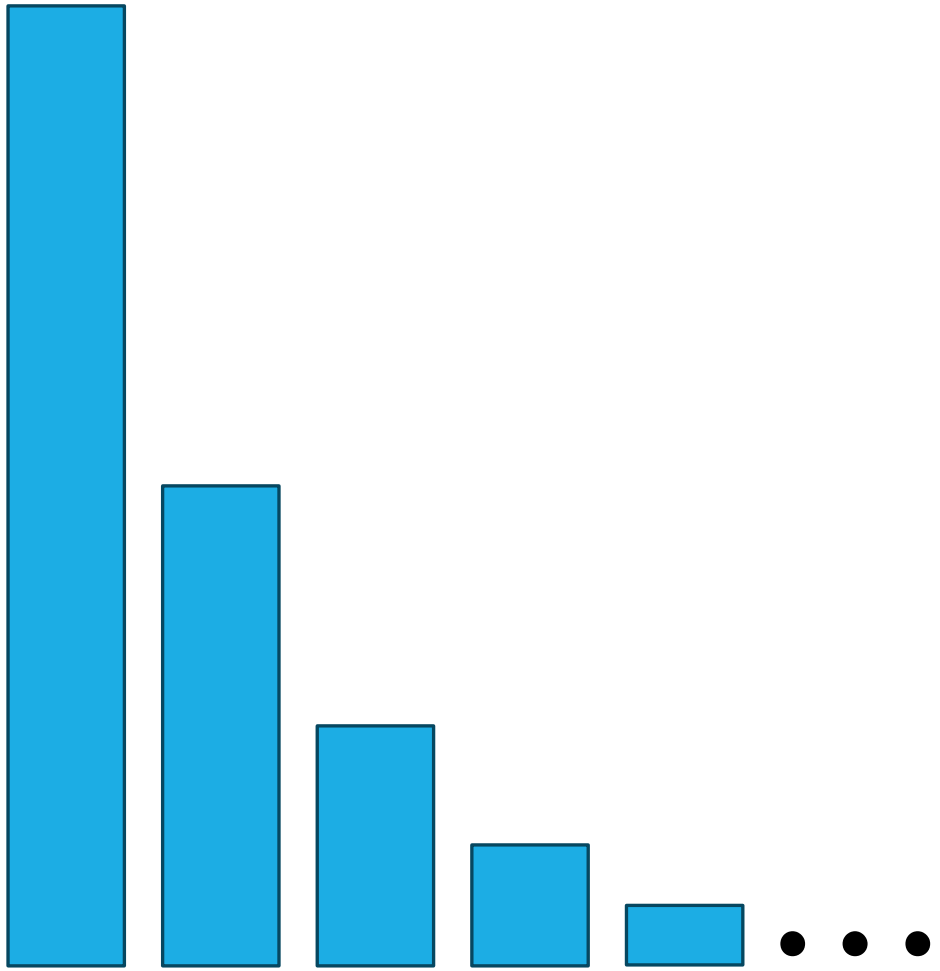
Work is $c2m + cm + \frac{cm}{2} + \frac{cm}{4} + \cdots$ down to $c \cdot$(starting size)

Total work?

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

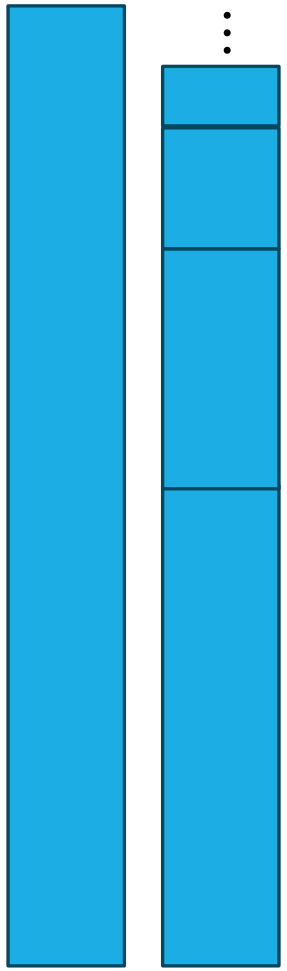
Summation Intuition

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?

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.

A Contrived Example

A `MakesYouWaitList` operates as follows:

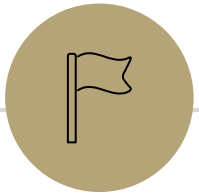
When you call `find()`, it does a linear search through an array of n elements to find i. If the index is odd, it spins for $O(n)$ time, if the index is even it spins for $O(n^2)$ time. Additionally, every n^{th} call to find it spins for $O(n^{2.5})$ time. It looks like this:

```
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 $O(n^2)$



More big-O examples

Rearranging Inequalities

Is $\mathrm{n}^2 + 10\mathrm{n} \in O(n^3)$

Can also rearrange a single inequality

Scratch work: (not part of proof)

$n^2 + 10n \leq ?\ n^3$

Iff $n + 10 \leq ? n^2$

Iff $10 \leq ? n^2 - n$ set $? = 1$, to make factoring easier

Iff $10 \leq n(n-1)$ which is definitely true for n at least 5. So take $n_0 = 5$

That's our scratch work, now what's our proof?

Rearranging Inequalities

Claim: $\mathrm{n}^2 + 10\mathrm{n} \in O(n^3)$

Proof: Take $c = 1$ and $n_0 = 5$. Observe that for all $n \geq n_0$, we have

$10 \leq n(n-1)$ (as the right-hand-side is increasing with n)

Multiplying by n, we get $10n \leq n^3 - n^2$

Rearranging, we have $10n + n^2 \leq n^3$

As this inequality holds for all $n \geq n_0$, we have met the definition as required.

String of Inequalities

Can also string together inequalities (all facing the same direction)

Claim: $\log_5(n+10) \in O(\log_5(n))$.

Proof: We claim that $c = 3$ and $n_0 = 5$ suffice.

Observe for arbitrary $n \geq n_0$:

$$\log_5(n+10) \leq \log_5(n+10n) \leq \log_5(11) + \log_5(n) \leq 2 + \log_5(n)$$

$$\leq 2\log_5(n) + \log_5(n) \text{ for } n \geq 5 \text{ } (\log_5(n) \text{ is increasing, } \log_5(n) = 1)$$

$$\leq 3\log_5(n)$$

This string of inequalities holds for all $n \geq 5$, meeting the definition of big-O.