CPSC 221
Basic Algorithms and Data Structures

May 20, 2015

Administrative stuff

Hw1 and Hw2 – coming soon, they may overlap

Lab 3 is posted – you need to do your C++ reading!!!

An office hours schedule is coming

Administrative stuff

Who do I borrow from?
(credit where credit is due)

Among others:

Steve Wolfman, UBC
Alan Hu, UBC
Kimberly Voll, UBC and Centre for Digital Media
George Bebis, University of Nevada, Reno
Thomas Anastasio and Richard Chang, University of Maryland, Baltimore County
Your textbook
Other textbooks, notably:
  Data Structures and Problem Solving using Java, by Mark Allen Weiss

and that's just for today's lecture!

Double-ended queue (deque) uses

- The undo function with an age (length) limit
- Job-scheduling algorithms (operating systems)

Big-O review

T(n) is O(n)

T(n) is our shorthand for the runtime of the function being analyzed.

The O in O(n) means “order of magnitude”, so Big-O notation is clearly not precise. It’s a formal notation for the approximation of the time (or space) requirements of running algorithms.

Big-O

The formal mathematical definition for Big-O:

T(n) is O(f(n)) if there are two constants, n_0 and c, both > 0, and a function f(n) such that cf(n) >= T(n) for all n > n_0

Big-O

The formal mathematical definition for Big-O:

T(n) is O(f(n)) if there are two constants, n_0 and c, both > 0, and a function f(n) such that cf(n) >= T(n) for all n > n_0

What this really means in a practical sense:

If you want to show that T(n) is O(f(n)) then find two positive constants, n_0 and c, and a function f(n) that satisfy the constraints above.

Big-O

The Big-O definition says that there is a point n_0 such that for all values of n that are past this point, T(n) is bounded by some multiple of f(n). Thus, if the running time T(n) of an algorithm is $O(n^2)$, we are guaranteeing that at some point, n_0, we can bound the running time by a quadratic function (a function whose high-order term involves n^2).

Or in other words, Big-O says there's a function that is an upper bound to the worst case performance for the algorithm.

Big-O

Note however that if T(n) is linear and not quadratic, you could still say that the running time is O(n^2). It's technically correct because the inequality holds. However, O(n) would be the more precise claim because it's an even lower upper bound.

What exactly are we trying to do?

Asymptotic analysis: we're studying the behaviour of functions f(n) as n gets very large, approaching ∞.

What exactly are we trying to do?

Asymptotic analysis: we're studying the behaviour of functions f(n) as n gets very large, approaching ∞.

Asymptotic analysis is based on the idea that as the problem size grows, the complexity can be described as a simple proportionality to some known function. This idea is incorporated in the Big-O notation for asymptotic performance. When we say T(n) is $O(n^2)$, we mean that the growth of the run time is proportional to n^2, and that there is at least one specific function, cn^2, that is an upper bound to the growth of the run time.

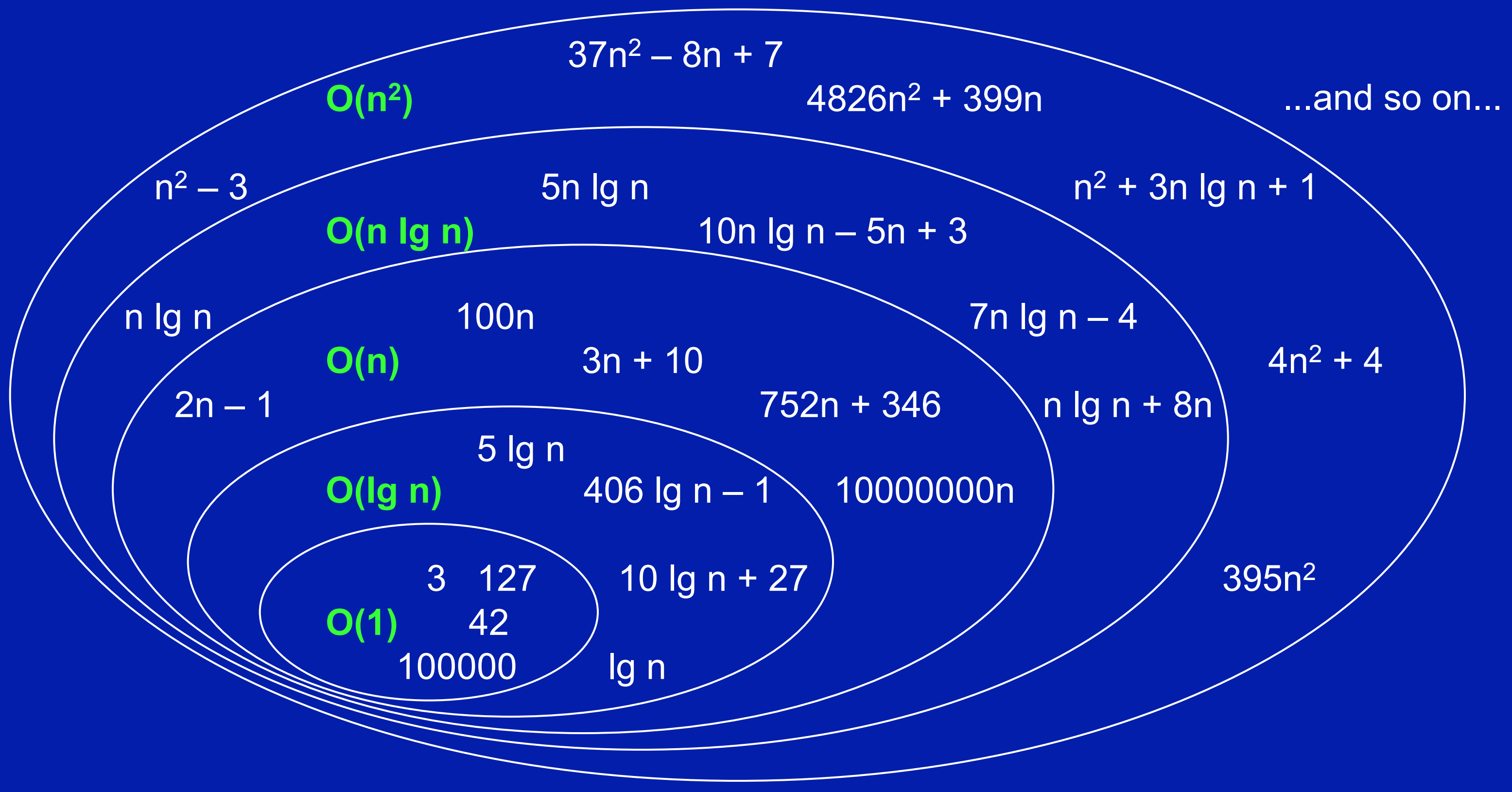
What exactly are we trying to do?

Asymptotic analysis: we're studying the behaviour of functions f(n) as n gets very large, approaching ∞.

Asymptotic analysis is based on the idea that as the problem size grows, the complexity can be described as a simple proportionality to some known function. This idea is incorporated in the Big-O notation for asymptotic performance. When we say T(n) is O(n^2), we mean that the growth of the run time is proportional to n^2, and that there is at least one specific function, cn^2, that is an upper bound to the growth of the run time.

We're just trying to classify algorithms into these standard complexity categories. We're not trying to say which algorithm among many within the category is better...that's where the constants that we ignore while doing the analysis come back into play again.

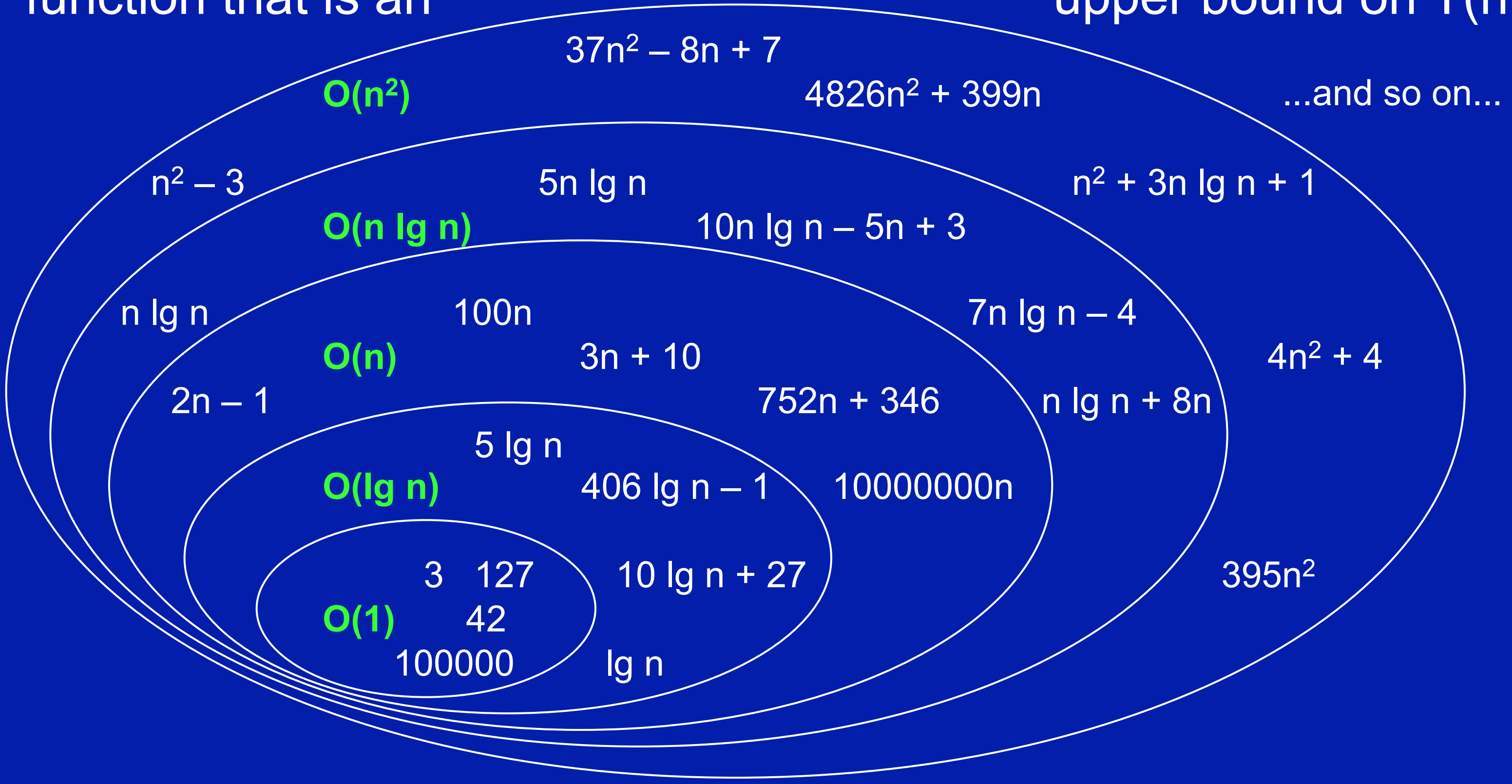
What exactly are we trying to do?



O(g(n)) is the set of functions with smaller or same order of growth as g(n)

What exactly are we trying to do?

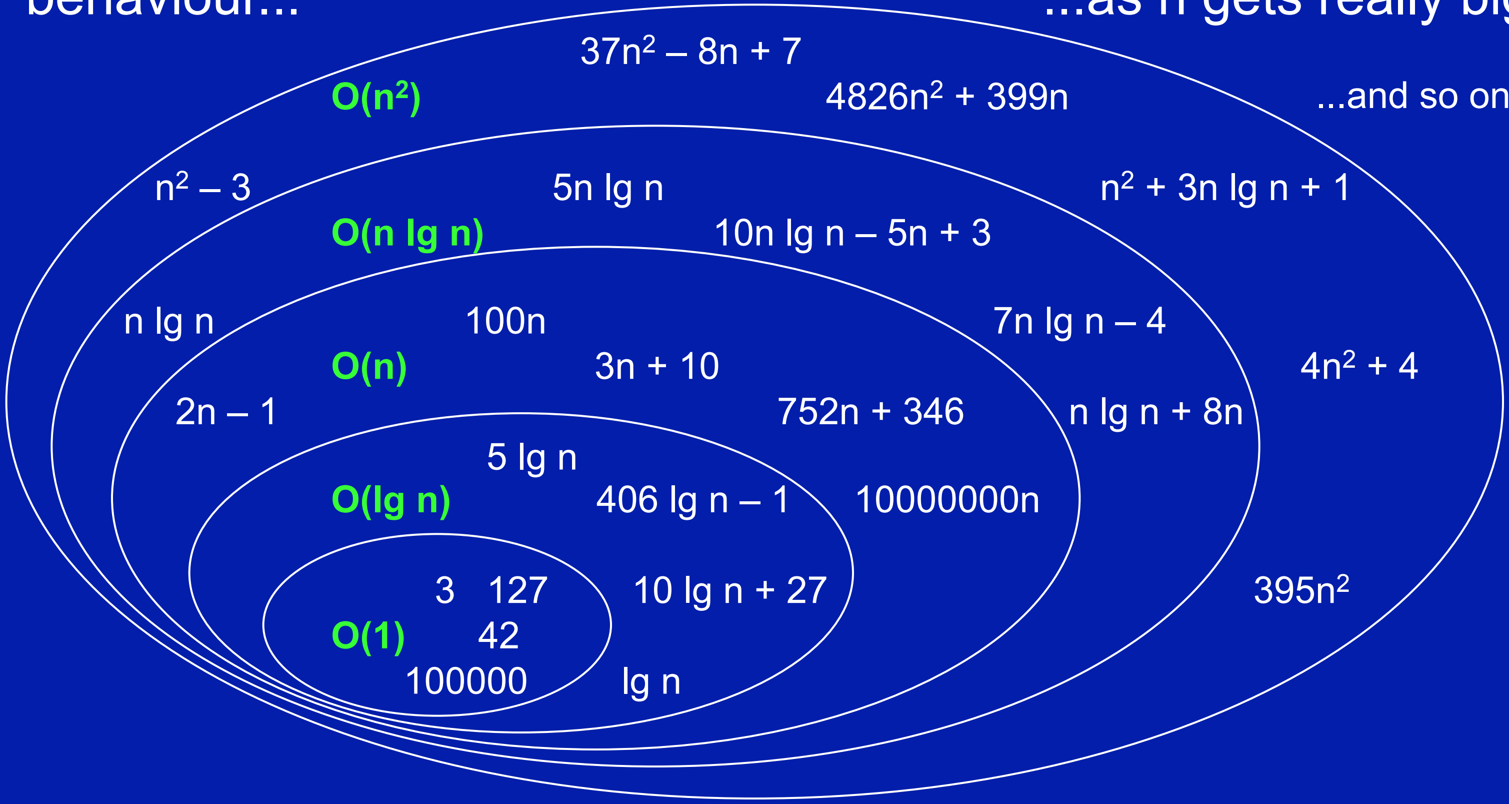
For any f(n) with run time T(n), we're trying to categorize the run time behaviour by placing it in the smallest possible group that contains a function that is an upper bound on T(n)



O(g(n)) is the set of functions with smaller or same order of growth as g(n)

What exactly are we trying to do?

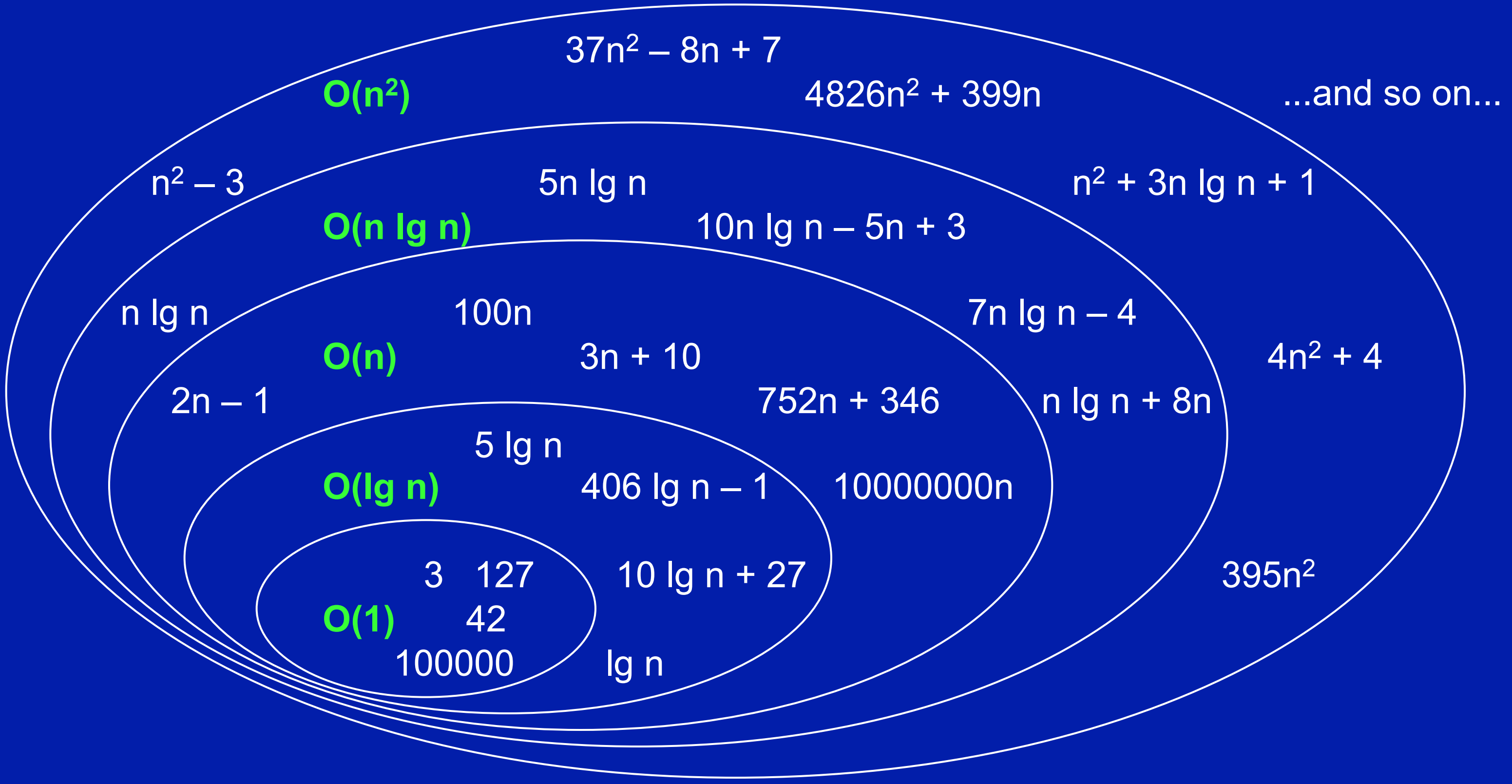
If you start with an algorithm that inherently has $O(n^2)$ complexity, there's no combination of constants that will give it $O(n \lg n)$ behaviour... ...as n gets really big.



O(g(n)) is the set of functions with smaller or same order of growth as g(n)

What exactly are we trying to do?

However, there may be a limited range of values of n for which the $O(n^2)$ algorithm gives better performance than the O(n lg n) algorithm.



O(g(n)) is the set of functions with smaller or same order of growth as g(n)

Common growth rates

Big-O is for expressing how run time or memory requirements grow as a function of the problem size. Rate of growth gets bigger as you go down the table.

Big-O	*name*
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(n^k)$	Polynomial – k is constant
$O(2^n)$	Exponential
O(n!)	Factorial

Impact of constants

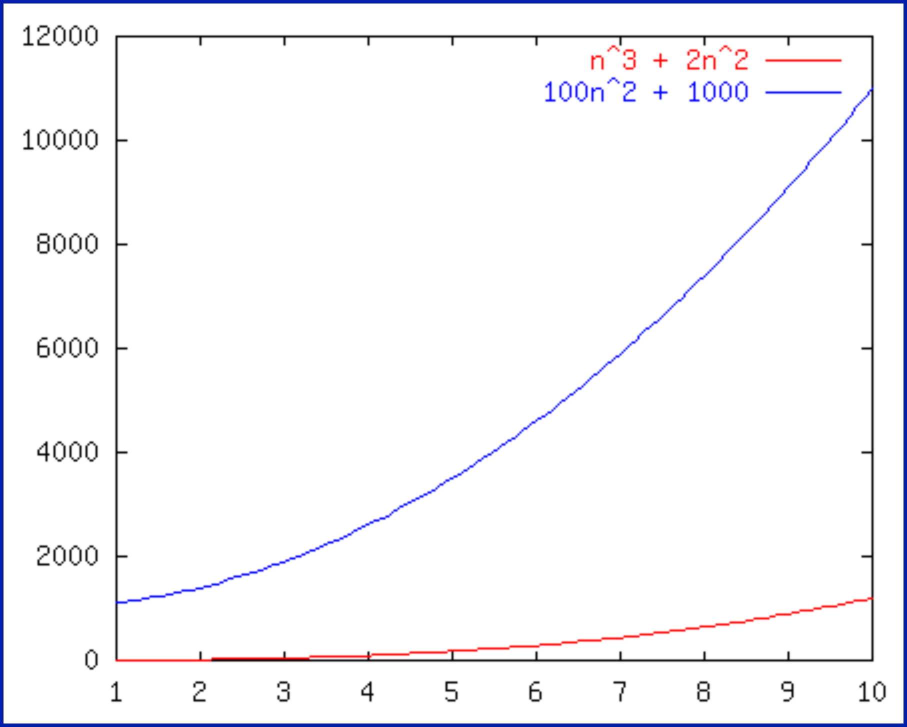
Which of these functions has the faster rate of growth as n gets bigger?

$n^3 + 2n^2$ or $100n^2 + 1000$

Impact of constants

Which of these functions has the faster rate of growth as n gets bigger?

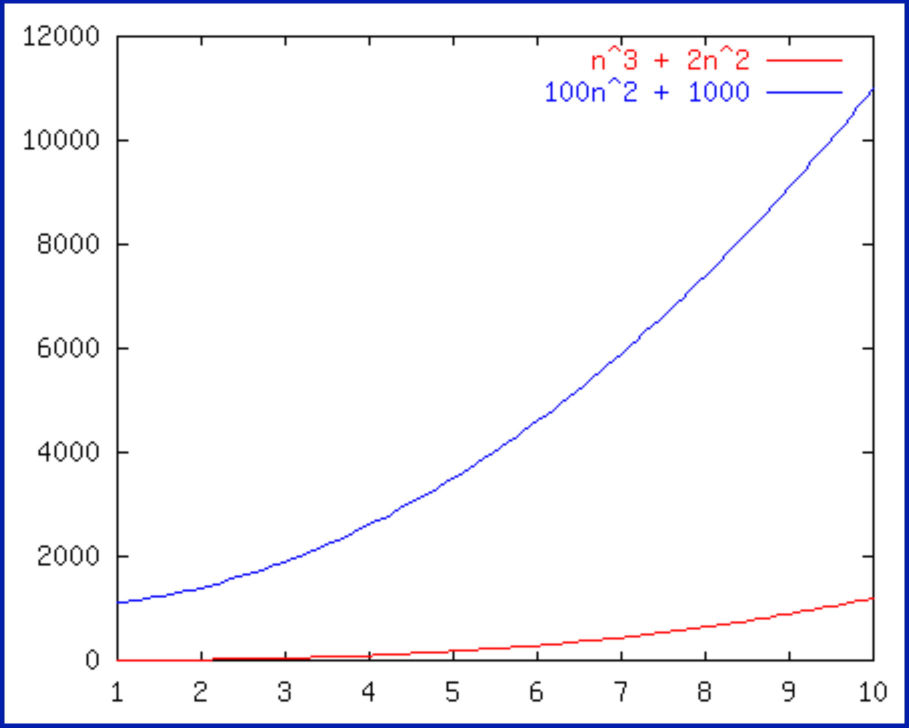
$n^3 + 2n^2$ or $100n^2 + 1000$

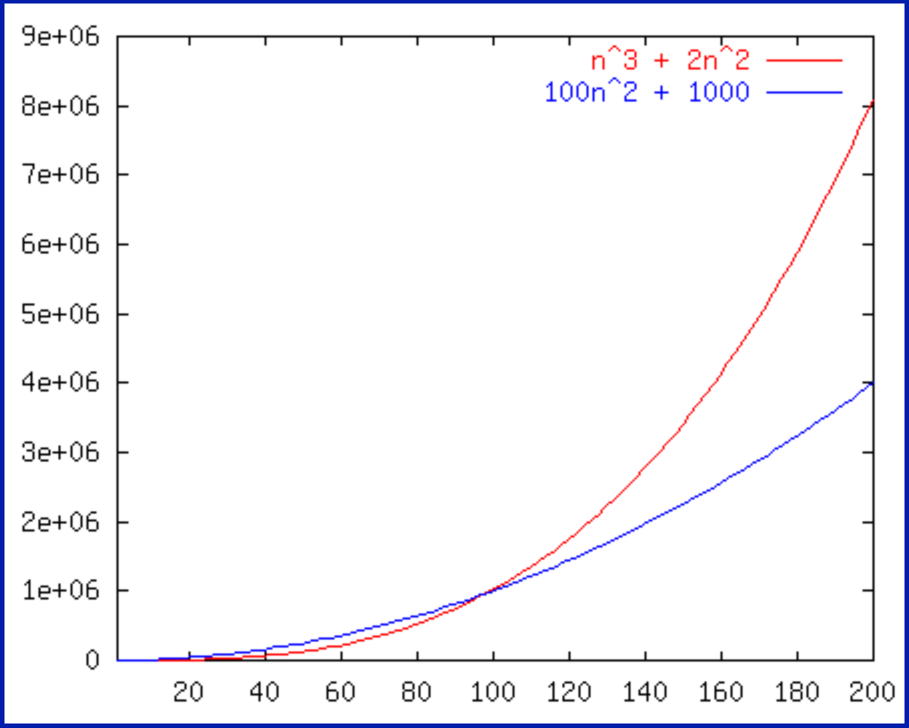


Impact of constants

Which of these functions has the faster rate of growth as n gets bigger?

$n^3 + 2n^2$ or $100n^2 + 1000$





Impact of constants

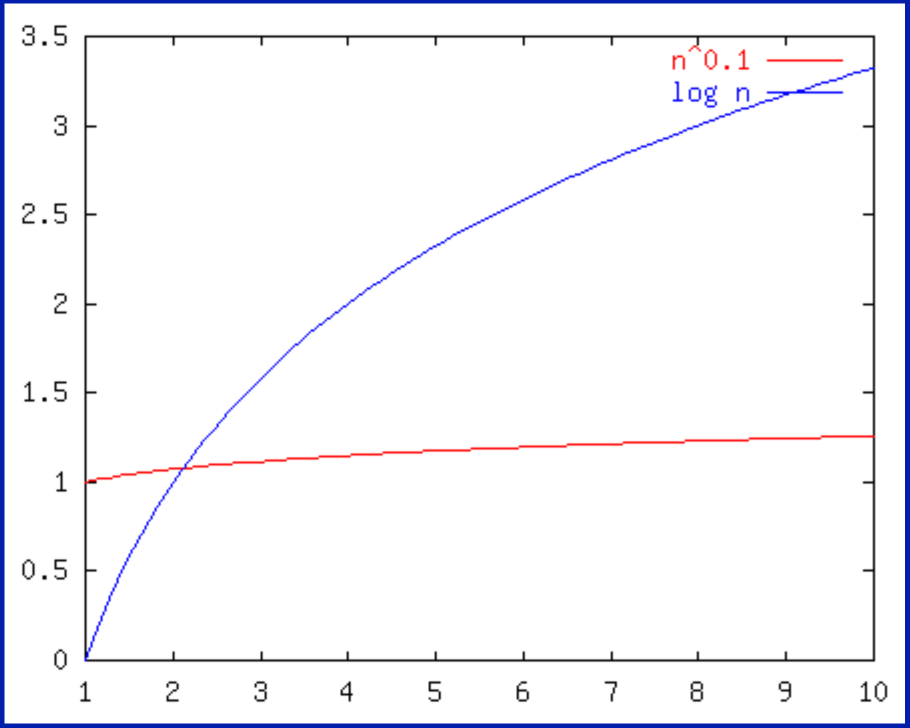
Which of these functions has the faster rate of growth as n gets bigger?

$n^{0.1}$ or $\log n$

Impact of constants

Which of these functions has the faster rate of growth as n gets bigger?

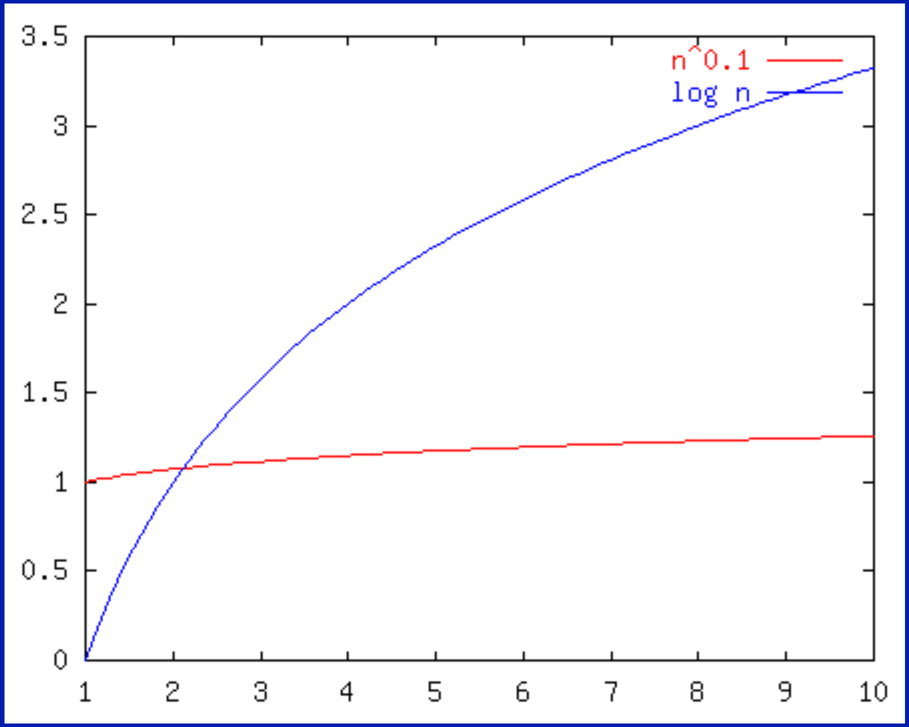
$n^{0.1}$ or $\log n$

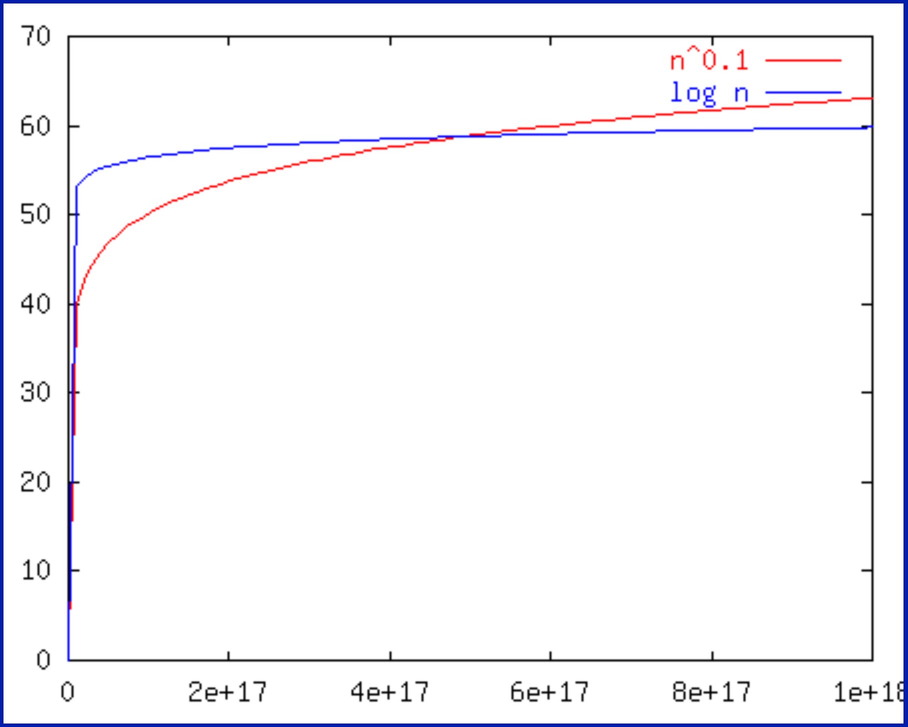


Impact of constants

Which of these functions has the faster rate of growth as n gets bigger?

$n^{0.1}$ or $\log n$





Impact of constants

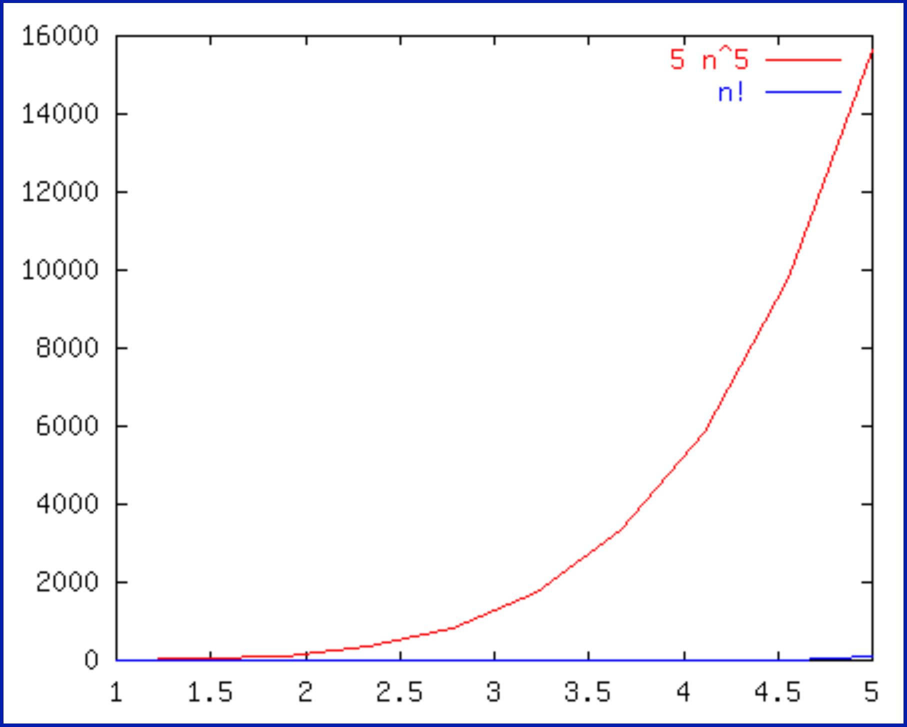
Which of these functions has the faster rate of growth as n gets bigger?

$5n^5$ or $n!$

Impact of constants

Which of these functions has the faster rate of growth as n gets bigger?

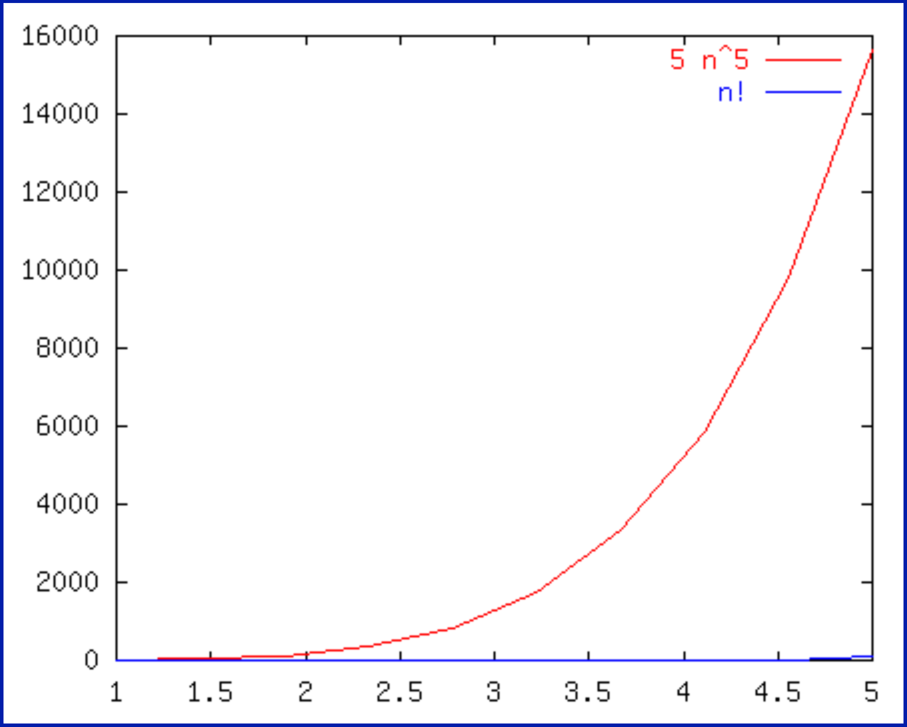
$5n^5$ or $n!$

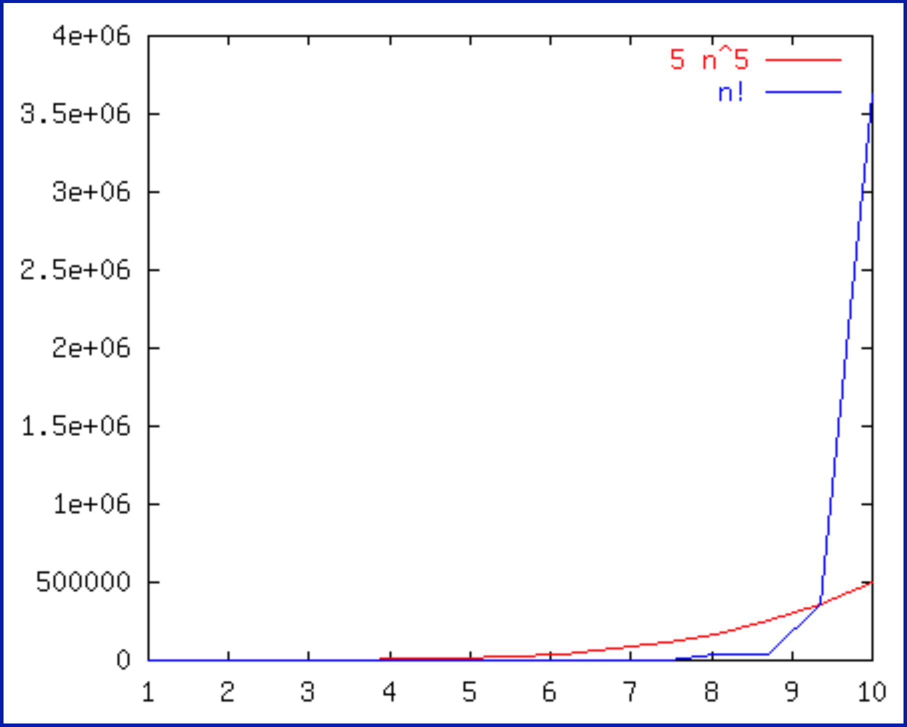


Impact of constants

Which of these functions has the faster rate of growth as n gets bigger?

$5n^5$ or $n!$





Common growth rates

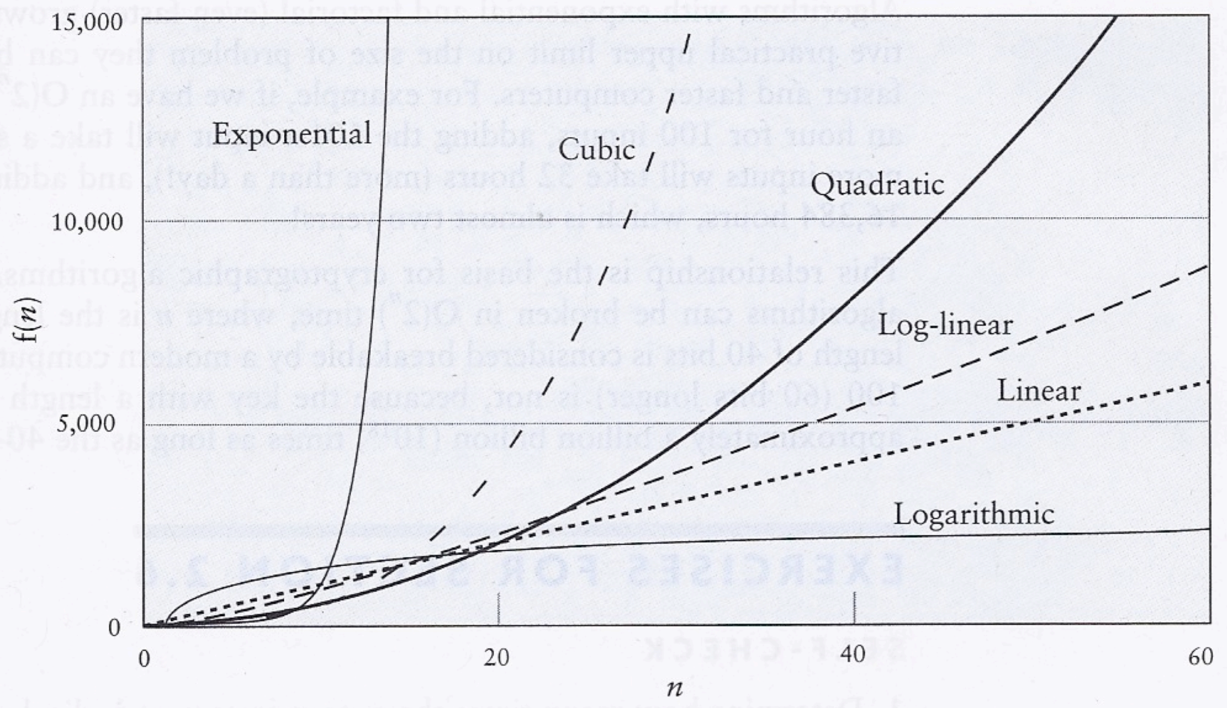
When graphed in comparison to each other, the representative function from each category exhibits its own characteristic trajectory...

Big-O	*name*
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(n^k)$	Polynomial – k is constant
$O(2^n)$	Exponential
$O(n!)$	Factorial

Common growth rates

FIGURE 2.8

Different Growth Rates

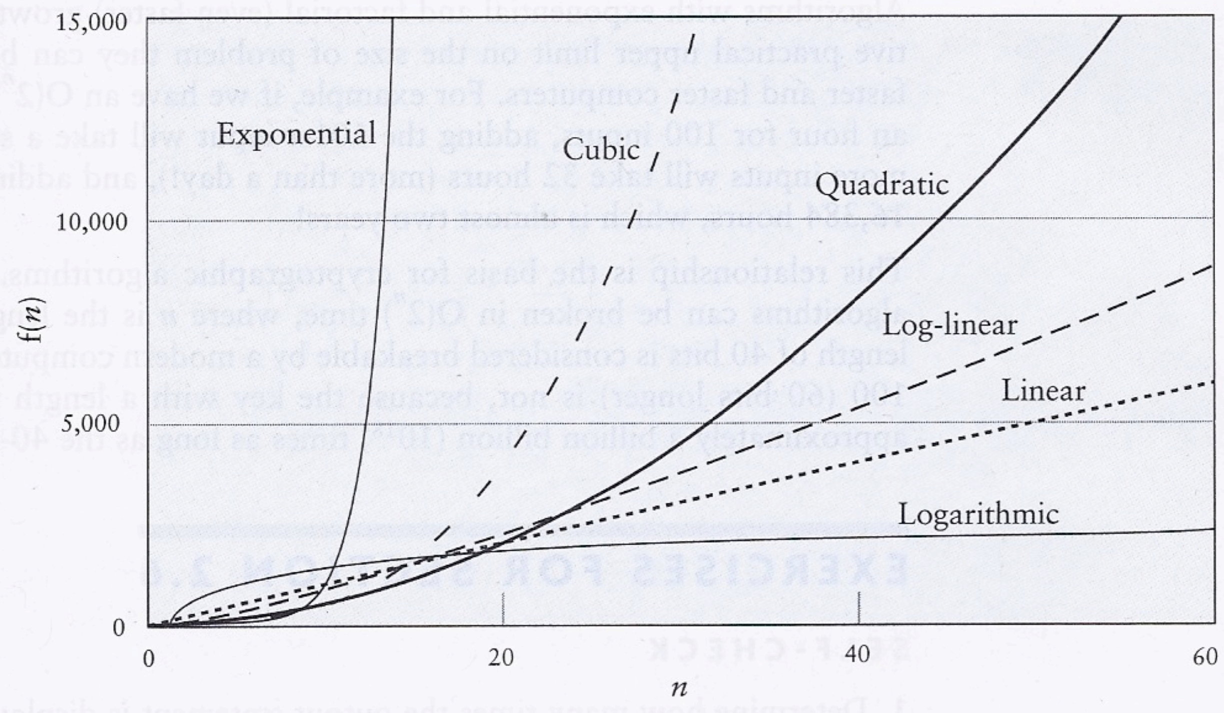


Common growth rates

You could, for example, find a constant that will make an $O(n^3)$ function perform better than an $O(n)$ function for a bigger range of n...

FIGURE 2.8

Different Growth Rates

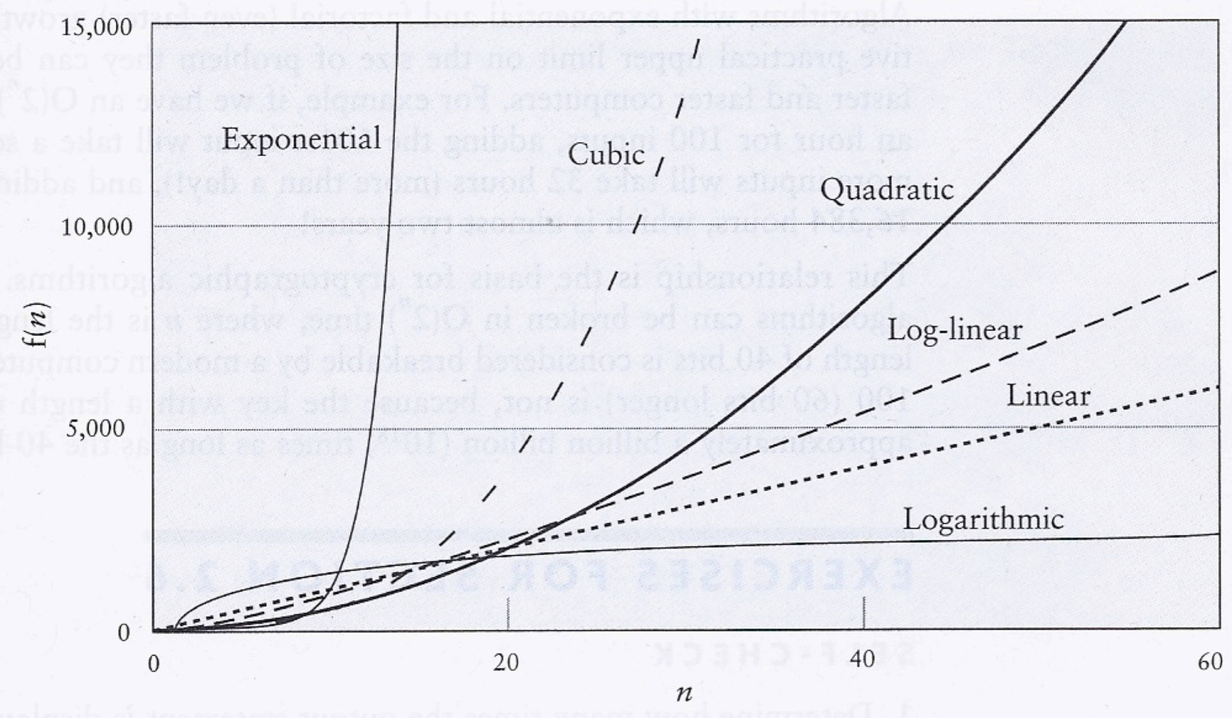


Common growth rates

...but at some point, the $O(n^3)$ function will head north in typical $O(n^3)$ fashion. No way around it. A function's gotta do what a function's gotta do.

FIGURE 2.8

Different Growth Rates



Low-order terms

That’s the story of constants. When we’re analyzing space or time complexity, we disregard the constants. That doesn’t mean the constants aren’t important if we want to talk about real performance (e.g., clock time, CPU time, actual memory usage), but when we want to talk just about the complexity inherent in the algorithm, regardless of the details, constants are one of the details we ignore.

Low-order terms

That’s the story of constants. When we’re analyzing space or time complexity, we disregard the constants. That doesn’t mean the constants aren’t important if we want to talk about real performance (e.g., clock time, CPU time, actual memory usage), but when we want to talk just about the complexity inherent in the algorithm, regardless of the details, constants are one of the details we ignore.

The other details we ignore are low-order terms. For example...

Low-order terms

Let's say we've calculated the run time for some bit of code as

$$T(n) = 42n^3 + 3n^2 - 17n - 8$$

We know we can disregard the constants:

$$T(n) = n^3 + n^2 - n$$

But we don't say that $T(n)$ is $O(n^3 + n^2 - n)$. Why?

Low-order terms

Let’s say we’ve calculated the run time for some bit of code as

$$T(n) = 42n^3 + 3n^2 - 17n - 8$$

We know we can disregard the constants:

$$T(n) = n^3 + n^2 - n$$

But we don’t say that T(n) is $O(n^3 + n^2 - n)$. Why? Because as n gets very big, the growth of n^3 dominates the growth of the other terms – the low-order terms – and those terms have no noticeable impact as n gets very big.

Low-order terms

So if the run time is this:

$$T(n) = 42n^3 + 3n^2 - 17n - 8$$

Then $T(n)$ is $O(n^3)$ because the dominant or high-order term is $O(n^3)$.

Low-order terms

The “order of dominance” is in this same table. Read from the bottom up. O(n!) dominates O(2^n). O(2^n) dominates O(n^k). And so on...

Big-O	*name*
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Log-linear
O(n^2)	Quadratic
O(n^3)	Cubic
O(n^k)	Polynomial – k is constant
O(2^n)	Exponential
O(n!)	Factorial

Analyzing code (revisited)

• C++ operations	- constant time
• consecutive stmts	- sum of times
• conditionals	- sum of branches + condition
• loops	- sum of iterations
• function calls	- cost of function body

Above all, use your head!

Analyzing code

```
// Linear search
find(key, array)
  for i = 1 to length(array) do
    if array[i] == key
      return i
  return -1
```

Step 1: What’s the input size n?

Analyzing code

```
// Linear search
find(key, array)
  for i = 1 to length(array) do
    if array[i] == key
      return i
  return -1
```

Step 2: What kind of analysis should we perform?
Worst-case? Best-case? Average-case? ...

Analyzing code

```
// Linear search
find(key, array)
  for i = 1 to length(array) do
    if array[i] == key
      return i
  return -1
```

Step 3: How much does each line cost? (Are lines the right unit?)

Analyzing code

```
// Linear search
find(key, array)
  for i = 1 to length(array) do
    if array[i] == key
      return i
  return -1
```

Step 4: What's T(n) in its raw form?

Analyzing code

```
// Linear search
find(key, array)
  for i = 1 to length(array) do
    if array[i] == key
      return i
  return -1
```

Step 5: Simplify T(n) and convert to Big-O notation.

Analyzing code

```
// Linear search
find(key, array)
  for i = 1 to length(array) do
    if array[i] == key
      return i
  return -1
```

Step 6: Prove the asymptotic upper bound by finding constants c and n_0 such that for all $n \geq n_0$, $T(n) \leq cn$.

You probably won't do this much in practice, but you should be able to do it for your exam. You just never know...

Analyzing code – another example

```
for $i$ = 1 to $n$ do
  for $j$ = 1 to $n$ do
    $sum$ = $sum$ + 1
```

Analyzing code – another example

```
for $i$ = 1 to $n$ do
  for $j$ = 1 to $n$ do
    $sum$ = $sum$ + 1
```

There’s a constant amount of work inside the inner loop.

Analyzing code – another example

```
for i = 1 to n do
  for j = 1 to n do
    sum = sum + 1
```

There’s a constant amount of work inside the inner loop.

The inner loop is executed n times. So that’s n * k.

Analyzing code – another example

```
for i = 1 to n do
  for j = 1 to n do
    sum = sum + 1
```

There’s a constant amount of work inside the inner loop.

The inner loop is executed n times. So that’s n * k.

The outer loop is executed n times. Now that’s n * n * k.

Analyzing code – another example

```
for i = 1 to n do
  for j = 1 to n do
    sum = sum + 1
```

There’s a constant amount of work inside the inner loop.

The inner loop is executed n times. So that’s n * k.

The outer loop is executed n times. Now that’s n * n * k.

What happens to the constant?

Analyzing code – another example

```
for i = 1 to n do
  for j = 1 to n do
    sum = sum + 1
```

There's a constant amount of work inside the inner loop.

The inner loop is executed n times. So that's n * k.

The outer loop is executed n times. Now that's n * n * k.

What happens to the constant?

T(n) is O(n^2)

Analyzing code – another example

```
for $i$ = 1 to $n$ do
  for $j$ = 1 to $n$ do
    $sum$ = $sum$ + 1
```

In pure mathematical terms, it looks like this:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} k$$

Analyzing code – another example

```
for $i$ = 1 to $n$ do
  for $j$ = 1 to $n$ do
    $sum$ = $sum$ + 1
```

In pure mathematical terms, it looks like this:

$\sum_{i=1}^{n}\sum_{j=1}^{n} k$ which is $\sum_{i=1}^{n} n * k$

Analyzing code – another example

```
for i = 1 to n do
  for j = 1 to n do
    sum = sum + 1
```

In pure mathematical terms, it looks like this:

$\sum_{i=1}^{n}\sum_{j=1}^{n} k$ which is $\sum_{i=1}^{n} n * k$ and that's just $n * n * k$

Analyzing code – another example

```
for i = 1 to n do
  for j = 1 to n do
    sum = sum + 1
```

In pure mathematical terms, it looks like this:

$\sum_{i=1}^{n}\sum_{j=1}^{n} k$ which is $\sum_{i=1}^{n} n * k$ and that's just $n * n * k$

You know the rest.

This is essentially example 2.16 from your textbook.

Analyzing code – yet another example

```
for i = 1 to n do                   i varies 1 to n
  for j = i to n do                 j varies i to n
    sum = sum + 1                   constant work k
```

Analyzing code – yet another example

```
for i = 1 to n do                i varies 1 to n
  for j = i to n do              j varies i to n
    sum = sum + 1                constant work k
```

There's a variable amount of work inside the inner loop, and it's dependent on the outer loop. Let's go to the outer loop and work inward.

Analyzing code – yet another example

```
for i = 1 to n do              i varies 1 to n
  for j = i to n do            j varies i to n
    sum = sum + 1              constant work k
```

There’s a variable amount of work inside the inner loop, and it’s dependent on the outer loop. Let’s go to the outer loop and work inward.

The outer loop is executed n times.

Analyzing code – yet another example

```
for i = 1 to n do                  i varies 1 to n
  for j = i to n do                j varies i to n
    sum = sum + 1                  constant work k
```

There's a variable amount of work inside the inner loop, and it's dependent on the outer loop. Let's go to the outer loop and work inward.

The outer loop is executed n times.

The inner loop is executed n times on the first pass of the outer loop.

Analyzing code – yet another example

```
for i = 1 to n do                i varies 1 to n
  for j = i to n do              j varies i to n
    sum = sum + 1                constant work k
```

There’s a variable amount of work inside the inner loop, and it’s dependent on the outer loop. Let’s go to the outer loop and work inward.

The outer loop is executed n times.

The inner loop is executed n times on the first pass of the outer loop.

The inner loop is executed n - 1 times on the second pass....

Analyzing code – yet another example

```
for i = 1 to n do                 i varies 1 to n
  for j = i to n do               j varies i to n
    sum = sum + 1                 constant work k
```

There's a variable amount of work inside the inner loop, and it's dependent on the outer loop. Let's go to the outer loop and work inward.

The outer loop is executed n times.

The inner loop is executed n times on the first pass of the outer loop.

The inner loop is executed n - 1 times on the second pass....

The inner loop is executed 1 time on the last pass of the outer loop.

Analyzing code – yet another example

```
for i = 1 to n do              i varies 1 to n
  for j = i to n do            j varies i to n
    sum = sum + 1              constant work k
```

So how many times is that statement inside the inner loop executed?

Analyzing code – yet another example

```
for i = 1 to n do            i varies 1 to n
  for j = i to n do          j varies i to n
    sum = sum + 1            constant work k
```

So how many times is that statement inside the inner loop executed?

$n + (n – 1) + (n – 2) + ... + 3 + 2 + 1$

Analyzing code – yet another example

`for i = 1 to n do`	`i varies 1 to n`
`  for j = i to n do`	`j varies i to n`
`    sum = sum + 1`	`constant work k`

So how many times is that statement inside the inner loop executed?

n + (n – 1) + (n – 2) + ... + 3 + 2 + 1 (n terms here)

Analyzing code – yet another example

```
for i = 1 to n do                    i varies 1 to n
   for j = i to n do                 j varies i to n
      sum = sum + 1                  constant work k
```

So how many times is that statement inside the inner loop executed?

$$n + (n - 1) + (n - 2) + \ldots + 3 + 2 + 1 \text{ (n terms here)}$$

Let's rewrite like this:

$$(n + 1) + ((n - 1) + 2) + ((n - 2) + 3) + \ldots \quad \text{(n/2 terms here)}$$

Analyzing code – yet another example

```
for i = 1 to n do                   i varies 1 to n
  for j = i to n do                 j varies i to n
    sum = sum + 1                   constant work k
```

So how many times is that statement inside the inner loop executed?

$n + (n - 1) + (n - 2) + \ldots + 3 + 2 + 1$ (n terms here)

Let's rewrite like this:

$(n + 1) + ((n - 1) + 2) + ((n - 2) + 3) + \ldots$ (n/2 terms here)

And that simplifies to:

$(n + 1) + (n + 1) + (n + 1) + \ldots + (n + 1)$ (n/2 terms here too)

Analyzing code – yet another example

```
for i = 1 to n do                    i varies 1 to n
  for j = i to n do                  j varies i to n
    sum = sum + 1                    constant work k
```

(n + 1) + (n + 1) + (n + 1) + ... + (n + 1)   (n/2 terms here too)

can be simplified to n(n + 1) / 2 so

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

$T(n) = (kn^2 + kn) / 2$

Analyzing code – yet another example

```
for i = 1 to n do                  i varies 1 to n
  for j = i to n do                j varies i to n
    sum = sum + 1                  constant work k
```

(n + 1) + (n + 1) + (n + 1) + ... + (n + 1)   (n/2 terms here too)

can be simplified to n(n + 1) / 2 so

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

$T(n) = (kn^2 + kn) / 2$

Using Big-O notation, what's your best guess as to the time complexity of this bit of code?

Analyzing code – yet another example

```
for i = 1 to n do                i varies 1 to n
  for j = i to n do              j varies i to n
    sum = sum + 1                constant work k
```

(n + 1) + (n + 1) + (n + 1) + ... + (n + 1)   (n/2 terms here too)

can be simplified to n(n + 1) / 2 so

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

$T(n) = (kn^2 + kn) / 2$

Using Big-O notation, what's your best guess as to the time complexity of this bit of code? $T(n)$ is $O(n^2)$. You can do the proof at home.

Analyzing code – yet another example

```
for i = 1 to n do                  i varies 1 to n
  for j = i to n do                j varies i to n
    sum = sum + 1                  constant work k
```

We can do this mathematically too:

$$\sum_{i=1}^{n}\sum_{j=i}^{n} k$$

Analyzing code – yet another example

```
for i = 1 to n do
  for j = i to n do
    sum = sum + 1
```

```
i varies 1 to n
j varies i to n
constant work k
```

We can do this mathematically too:

$\sum_{i=1}^{n}\sum_{j=i}^{n} k$ becomes $\sum_{i=1}^{n}(n-i+1)k$

Analyzing code – yet another example

```
for i = 1 to n do                i varies 1 to n
  for j = i to n do              j varies i to n
    sum = sum + 1                constant work k
```

We can do this mathematically too:

$\sum_{i=1}^{n}\sum_{j=i}^{n} k$ becomes $\sum_{i=1}^{n}(n-i+1)k$

to make things a little simpler for us, let’s just set k to 1 for now

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

Analyzing code – yet another example

```
for i = 1 to n do                    i varies 1 to n
  for j = i to n do                  j varies i to n
    sum = sum + 1                    constant work k
```

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

Analyzing code – yet another example

```
for i = 1 to n do                 i varies 1 to n
  for j = i to n do               j varies i to n
    sum = sum + 1                 constant work k
```

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

the n and 1 terms don't change as i changes, so we can pull them out and multiply by the number of times they're added – first the n

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

Analyzing code – yet another example

```
for i = 1 to n do              i varies 1 to n
  for j = i to n do            j varies i to n
    sum = sum + 1              constant work k
```

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

the n and 1 terms don't change as i changes, so we can pull them out and multiply by the number of times they're added – first the n

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

then the 1

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

Analyzing code – yet another example

```
for i = 1 to n do                  i varies 1 to n
  for j = i to n do                j varies i to n
    sum = sum + 1                  constant work k
```

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

Analyzing code – yet another example

```
for i = 1 to n do              i varies 1 to n
  for j = i to n do            j varies i to n
    sum = sum + 1              constant work k
```

$(n*n)+(n*1)+\sum_{i=1}^{n} -i$ is the same as

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

Analyzing code – yet another example

```
for i = 1 to n do                i varies 1 to n
  for j = i to n do              j varies i to n
    sum = sum + 1                constant work k
```

$(n*n)+(n*1)+\sum_{i=1}^{n}-i$ is the same as

$(n*n)+(n*1)-\sum_{i=1}^{n}i$ and that's the same as

$(n*n)+(n*1)-n(n-1)/2$

Analyzing code – yet another example

```
for i = 1 to n do              i varies 1 to n
  for j = i to n do            j varies i to n
    sum = sum + 1              constant work k
```

$(n*n)+(n*1)-n(n-1)/2$ $= n^2 + n – (n^2 + n)/2$

Analyzing code – yet another example

```
for i = 1 to n do              i varies 1 to n
  for j = i to n do            j varies i to n
    sum = sum + 1              constant work k
```

$(n*n)+(n*1)-n(n-1)/2$ $= n^2 + n – (n^2 + n)/2$

$= (n^2 + n)/2$

Analyzing code – yet another example

```
for $i$ = 1 to $n$ do
    for $j$ = $i$ to $n$ do
        $sum$ = $sum$ + 1
```

```
i varies 1 to n
j varies i to n
constant work k
```

$(n * n) + (n * 1) - n(n-1)/2$ $= n^2 + n - (n^2 + n)/2$

$= (n^2 + n)/2$

$= k(n^2 + n)/2$ if we bring back the k, so

Analyzing code – yet another example

```
for i = 1 to n do                i varies 1 to n
  for j = i to n do              j varies i to n
    sum = sum + 1                constant work k
```

$(n*n)+(n*1)-n(n-1)/2$ $= n^2 + n – (n^2 + n)/2$

$= (n^2 + n)/2$

$= k(n^2 + n)/2$ if we bring back the k, so

$T(n) = (kn^2 + kn)/2$ just like on slide 66

Analyzing code – yet another example

```
for i = 1 to n do              i varies 1 to n
  for j = i to n do            j varies i to n
    sum = sum + 1              constant work k
```

$T(n) = (kn^2 + kn)/2$

Analyzing code – yet another example

```
for i = 1 to n do                    i varies 1 to n
  for j = i to n do                  j varies i to n
    sum = sum + 1                    constant work k
```

$T(n) = (kn^2 + kn)/2$

$kn^2 + kn$ (dropping the ½ coefficient)

Analyzing code – yet another example

```
for i = 1 to n do                   i varies 1 to n
  for j = i to n do                 j varies i to n
    sum = sum + 1                   constant work k
```

$T(n) = (kn^2 + kn)/2$

$kn^2 + kn$ (dropping the ½ coefficient)

$n^2 + n$ (disregarding constant k)

Analyzing code – yet another example

```
for i = 1 to n do              i varies 1 to n
  for j = i to n do            j varies i to n
    sum = sum + 1              constant work k
```

$T(n) = (kn^2 + kn)/2$

$kn^2 + kn$	(dropping the ½ coefficient)
$n^2 + n$	(disregarding constant k)
n^2	(drop the low-order term)

Analyzing code – yet another example

```
for i = 1 to n do             i varies 1 to n
  for j = i to n do           j varies i to n
    sum = sum + 1             constant work k
```

$T(n) = (kn^2 + kn)/2$

$kn^2 + kn$	(dropping the ½ coefficient)
$n^2 + n$	(disregarding constant k)
n^2	(drop the low-order term)

$T(n)$ is $O(n^2)$ (like example 2.17)

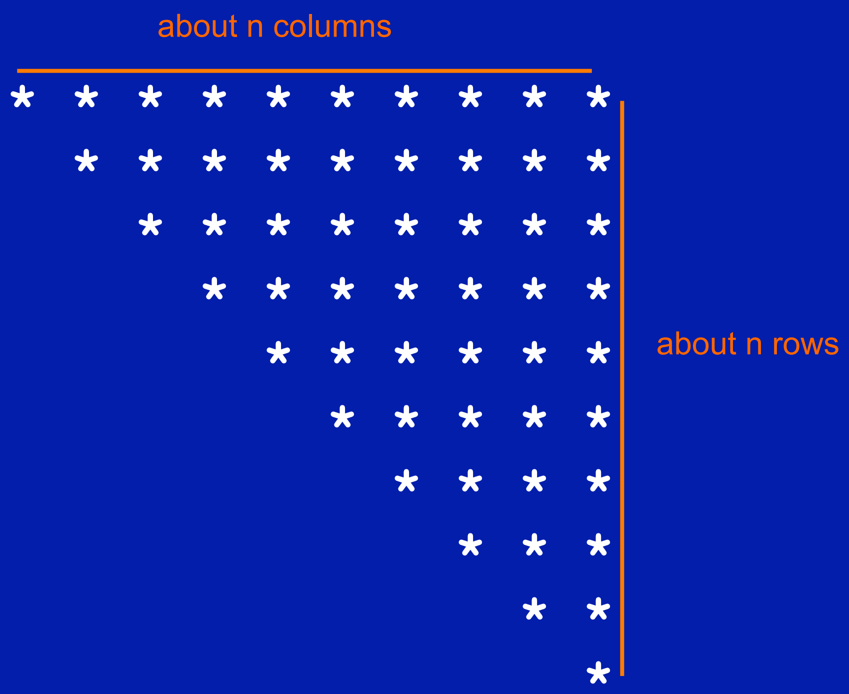
Analyzing code – yet another example

We can do the analysis using pictures!

```
* * * * * * * * * *
  * * * * * * * * *
    * * * * * * * *
      * * * * * * *
        * * * * * *
          * * * * *
            * * * *
              * * *
                * *
                  *
```

Analyzing code – yet another example

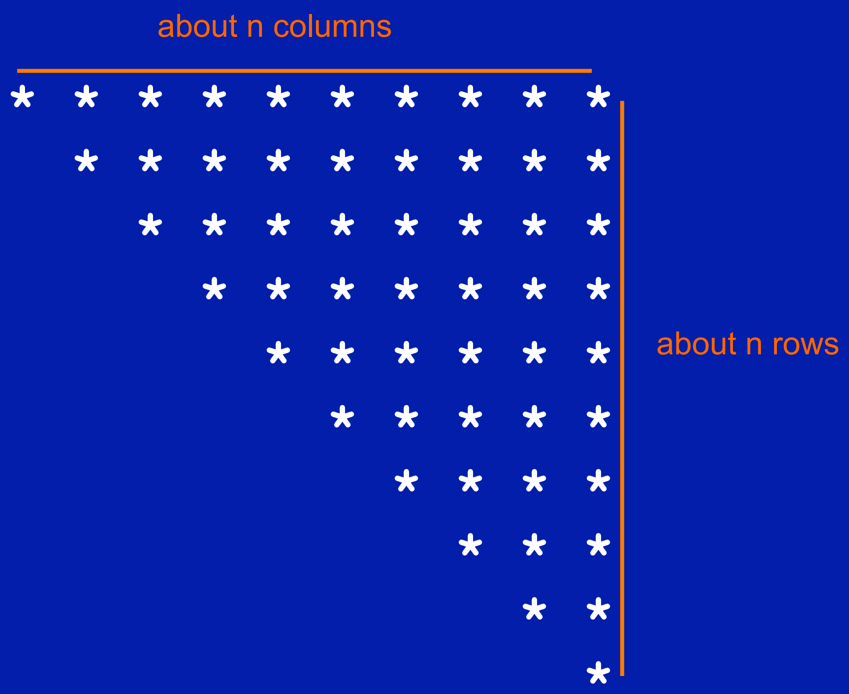
We can do the analysis using pictures!



It's a triangle and its area is proportional to run time.

Analyzing code – yet another example

We can do the analysis using pictures!

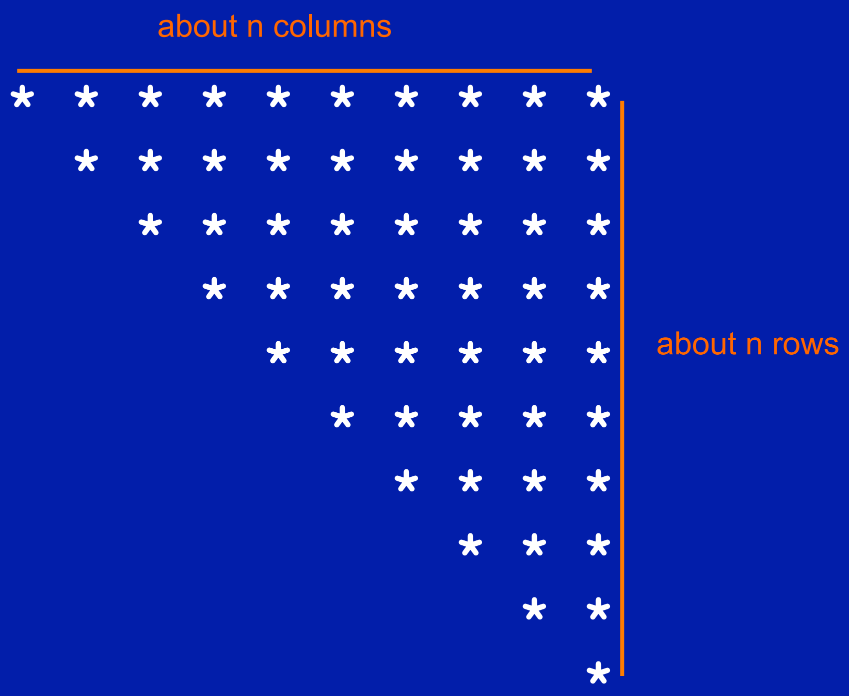


It's a triangle and its area is proportional to run time.

T(n) is approximately ½(base * height) or ½(n * n)

Analyzing code – yet another example

We can do the analysis using pictures!



It's a triangle and its area is proportional to run time.

T(n) is approximately ½(base * height) or ½(n * n)

T(n) is still O(n^2) no matter how we look at it.

Questions?

Big-O

The formal mathematical definition for Big-O:

T(n) is O(f(n)) if there are two constants, n_0 and c, both > 0, and a function f(n) such that cf(n) >= T(n) for all n > n_0

Big-O

The Big-O definition says that there is a point n_0 such that for all values of n that are past this point, T(n) is bounded by some multiple of f(n). Thus, if the running time T(n) of an algorithm is $O(n^2)$, we are guaranteeing that at some point, n_0, we can bound the running time by a quadratic function (a function whose high-order term involves n^2).

Or in other words, Big-O says there's a function that is an upper bound to the worst case performance for the algorithm. Big-O is sort of like less than or equal to when growth rates are being considered.

Big-Ω

What if you want to find a lower bound to the performance of an algorithm? That's where Big-Ω (Big-Omega) comes in.

Big-Ω says there's a function that is a lower bound to the worst case performance for the algorithm. In other words, the algorithm's performance will never get better than that lower bound.

Big-Ω

The formal mathematical definition for Big-Ω:

T(n) is Ω(f(n)) if there are two constants, n_0 and c, both > 0, and a function f(n) such that cf(n) <= T(n) for all n > n_0

If T(n) is Ω(f(n)), then we say "T(n) is at least order of f(n)" or "T(n) is bound from below by f(n)" or "T(n) is Big-Omega of f(n)".

For example, searching for an item in an unsorted array of n elements must be Ω(n) in time complexity because we must look at all n elements in the worst case.

Big-Θ

If we know that $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$, then we may say that $T(n)$ is $\Theta(f(n))$ (i.e., Big-Theta of $f(n)$).

The formal mathematical definition for Big-Θ:

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

This says that the growth rate of $T(n)$ equals the growth rate of $f(n)$. If we say that $T(n)$ is $\Theta(n^2)$, we're saying that $T(n)$ is bounded above and below by a quadratic function. It won't get worse than the upper bound, and it can't get better than the lower bound. Big-Θ assures the tightest possible bound on the algorithm's performance.

Big-Θ

Show that $T(n) = 48n + 8$ is $\Theta(n)$:

Big-Θ

Show that T(n) = 48n + 8 is Θ(n):

for O(n):

```
48n + 8 <= cn
```

Big-Θ

Show that T(n) = 48n + 8 is Θ(n):

for O(n):

```
  48n + 8 <= cn
8(6n + 1) <= cn
```

Big-Θ

Show that $T(n) = 48n + 8$ is $\Theta(n)$:

for $O(n)$:

```
  48n + 8 <= cn
8(6n + 1) <= cn
   6n + 1 <= cn/8
```

Big-Θ

Show that T(n) = 48n + 8 is Θ(n):

for O(n):

```
  48n + 8 <= cn
8(6n + 1) <= cn
   6n + 1 <= cn/8
       6n <= cn/8 – 1
```

Big-Θ

Show that T(n) = 48n + 8 is Θ(n):

for O(n):

```
  48n + 8 <= cn
8(6n + 1) <= cn
   6n + 1 <= cn/8
       6n <= cn/8 – 1
        n <= cn/48 – 1/6
```

Big-Θ

Show that T(n) = 48n + 8 is Θ(n):

for O(n):

```
  48n + 8 <= cn
8(6n + 1) <= cn
   6n + 1 <= cn/8
       6n <= cn/8 – 1
        n <= cn/48 – 1/6
        n <= n – 1/6? (if c = 48) no
```

Big-Θ

Show that T(n) = 48n + 8 is Θ(n):

for O(n):

```
  48n + 8 <= cn
8(6n + 1) <= cn
   6n + 1 <= cn/8
       6n <= cn/8 – 1
        n <= cn/48 – 1/6
        n <= n – 1/6? (if c = 48) no
        n <= 2n – 1/6? (if c = 96) maybe
```

Big-Θ

Show that T(n) = 48n + 8 is Θ(n):

for O(n):

```
  48n + 8 <= cn
8(6n + 1) <= cn
   6n + 1 <= cn/8
       6n <= cn/8 – 1
        n <= cn/48 – 1/6
        n <= n – 1/6? (if c = 48) no
        n <= 2n – 1/6? (if c = 96) maybe
        1 <= 2 – 1/6? (if n0 = 1) yes

T(n) is O(n)
```

Big-Θ

Show that $T(n) = 48n + 8$ is $\Theta(n)$:

Big-Θ

Show that $T(n) = 48n + 8$ is $\Theta(n)$:

for $\Omega(n)$:

```
48n + 8 >= cn
```

Big-Θ

Show that T(n) = 48n + 8 is Θ(n):

for Ω(n):

```
  48n + 8 >= cn
8(6n + 1) >= cn
```

Big-Θ

Show that T(n) = 48n + 8 is Θ(n):

for Ω(n):

```
  48n + 8 >= cn
8(6n + 1) >= cn
   6n + 1 >= cn/8
```

Big-Θ

Show that T(n) = 48n + 8 is Θ(n):

for Ω(n):

```
  48n + 8 >= cn
8(6n + 1) >= cn
   6n + 1 >= cn/8
       6n >= cn/8 – 1
```

Big-Θ

Show that T(n) = 48n + 8 is Θ(n):

for Ω(n):

```
  48n + 8 >= cn
8(6n + 1) >= cn
   6n + 1 >= cn/8
       6n >= cn/8 – 1
        n >= cn/48 – 1/6
```

Big-Θ

Show that T(n) = 48n + 8 is Θ(n):

for Ω(n):

```
  48n + 8 >= cn
8(6n + 1) >= cn
   6n + 1 >= cn/8
       6n >= cn/8 – 1
        n >= cn/48 – 1/6
        n >= n – 1/6? (if c = 48) maybe
```

Big-Θ

Show that T(n) = 48n + 8 is Θ(n):

for Ω(n):

```
  48n + 8 >= cn
8(6n + 1) >= cn
   6n + 1 >= cn/8
       6n >= cn/8 – 1
        n >= cn/48 – 1/6
        n >= n – 1/6? (if c = 48) maybe
        1 >= 1 – 1/6? (if n0 = 1) yes
```

```
T(n) is Ω(n) and T(n) is O(n) so T(n) is Θ(n).
```

One other thing

We write this: $T(n)$ is $O(f(n))$

or this: $T(n) \in O(f(n))$

but not this: $T(n) = O(f(n))$

because $O(f(n))$ describes a set of functions
(think back to the slides with the nested ovals)

Some books use = to mean is or $\in$ in this context, but it's not technically correct.