University of Colorado **Boulder**

Department of Computer Science

CSCI 2824: Discrete Structures

Chris Ketelsen

Algorithms and Complexity

Complexity and Growth of Functions

Algorithm Complexity

When does an algorithm provide an efficient solution to a problem?

Different measures of *efficiency*

- How long does it take to run? (**time complexity**)
- How much memory does it require? (**space complexity**)

Space complexity is for your Data Structures course

We'll worry about **time complexity**

Time complexity is a slight misnomer. Since different computers run at different speeds, we consider number of operations instead

- comparisons, additions, multiplications, etc

Algorithm Complexity

Example: What is the time complexity of a linear search?

procedure `LinearSearch`$(x, a_1, a_2, \ldots, a_n)$

$i := 1$

while $(i \leq n$ and $x \neq a_i)$

$\quad i := i + 1$

if $i \leq n$ **then** *location* $:= i$

else *location* $:= -1$

return *location*

Algorithm Complexity

Example: What is the time complexity of a linear search?

First, need to decide what operations we're going to count

Usually pick the most common or most expensive operation

Search is mostly comparisons so we'll count those

We'll start by doing a **worst-case** analysis

What is the worst case for linear search?

Algorithm Complexity

Example: What is the time complexity of a linear search?

First, need to decide what operations we're going to count

Usually pick the most common or most expensive operation

Search is mostly comparisons so we'll count those

We'll start by doing a **worst-case** analysis

What is the worst case for linear search?

Worst Case: When x is not in the list

Algorithm Complexity

Example: What is the worst-case time complexity of linear search?

procedure `LinearSearch`$(x, a_1, a_2, \ldots, a_n)$

$i := 1$

while $(i \leq n$ and $x \neq a_i)$

$\quad i := i + 1$

if $i \leq n$ **then** *location* $:= i$

else *location* $:= -1$

return *location*

Algorithm Complexity

Example: What is the worst-case time complexity of linear search?

Each time through the loop we do 2 comparisons

- is $i \leq n$
- is $x = a_i$

To exit the loop we do anther comparison ($i = n + 1 \leq n$)

Then we check $i \leq n$ to see if x was found

This is $n \times 2 + 1 + 1 = 2n + 2$ comparisons

So worst-case complexity of linear search is $2n + 2$

Algorithm Complexity

Example: What is worst-case complexity of binary search?

procedure `BinarySearch`$(x, a_1, a_2, \ldots, a_n)$

$\ell := 1,\ r := n$ # initialize left and right endpoints of interval

while $\ell < r$

$m := \lfloor (\ell + r)/2 \rfloor$ # m is index of largest in left list

if $x > a_m$ **then** $\ell := m + 1,$ **else** $r := m$

if $x = a_\ell$ **then** *location* $:= \ell,$ **else** *location* $:= -1$

return *location*

Algorithm Complexity

Example: What is worst-case complexity of binary search?

Again, we'll just look at comparisons

Simplifying Assumption: The number of list items is a power of 2

Let $n = 2^k$ for some integer k and note that $k = \log_2 n$

First step in while loop have two comparisons

- is $\ell < r$
- is $x > a_m$

After first step we repeat on a list of size 2^{k-1}

Algorithm Complexity

Example: What is worst-case complexity of binary search?

After first step we repeat on a list of size 2^{k-1}

After $\mathbf{k}$ **steps** (at **2** comparisons each) we have just one element left

One more comparison for $\ell < r$ to exit

One more comparison to test $x = a_m$

For worst-case complexity of $2k + 2 = 2\log_2 n + 2$

Algorithm Complexity

Example: What is worst-case complexity of binary search?

After first step we repeat on a list of size 2^{k-1}

After $\mathbf{k}$ **steps** (at **2** comparisons each) we have just one element left

One more comparison for $\ell < r$ to exit

One more comparison to test $x = a_m$

For worst-case complexity of $2k + 2 = 2\log_2 n + 2$

Question: Which is more efficient:

LinearSearch @ $2n + 2$ or BinarySearch @ $2\log_2 n + 2$?

Algorithm Complexity

Question: Which is more efficient:

LinearSearch @ $2n + 2$ or BinarySearch @ $2 \log_2 n + 2$?

Obviously, the $2\times$'s and the $+2$'s don't really matter

So we need to compare n vs. $\log_2 n$

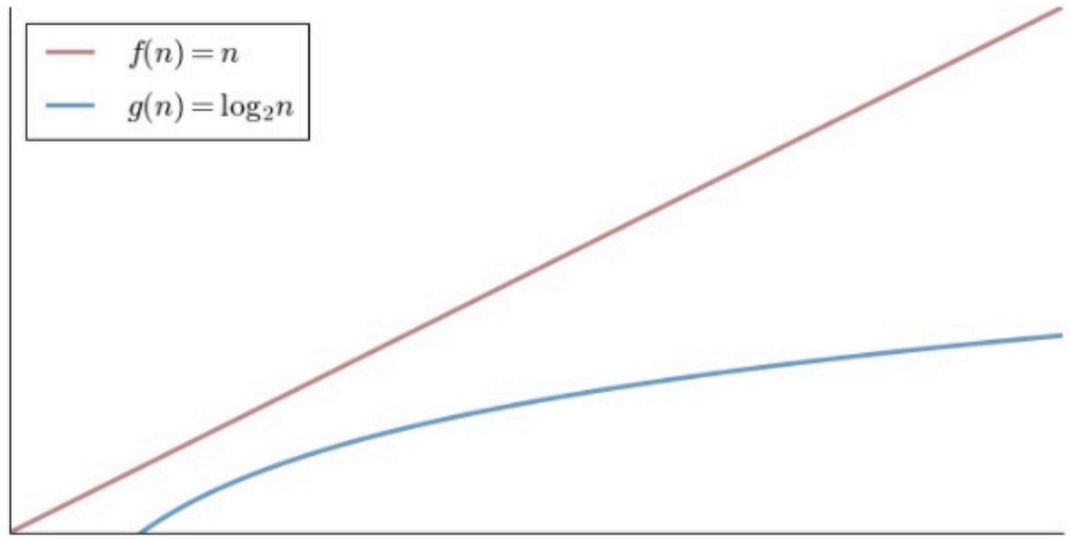
Algorithm Complexity

Question: Which is more efficient:

LinearSearch @ $2n + 2$ or BinarySearch @ $2 \log_2 n + 2$?

Obviously, the $2\times$'s and the $+2$'s don't really matter

So we need to compare n vs. $\log_2 n$



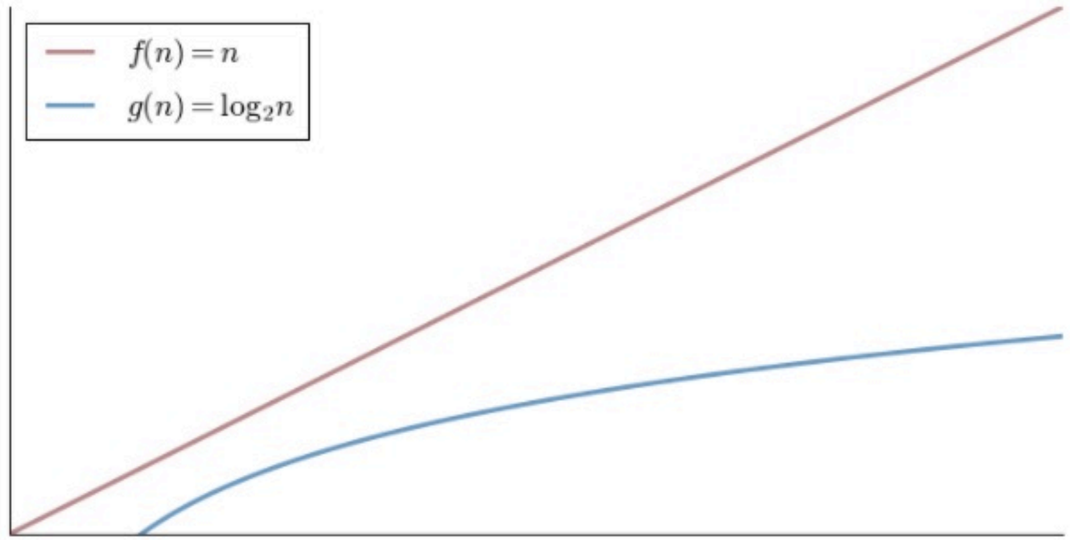
Algorithm Complexity

Question: Which is more efficient:

LinearSearch @ $2n + 2$ or BinarySearch @ $2 \log_2 n + 2$?

Obviously, the $2\times$'s and the $+2$'s don't really matter

Rule: Log grow slower than all polynomials (any n^p for $p > 0$)



Algorithm Complexity

Both of these complexity counts were for **worst-case** scenario

There is good chance in practice that they would finish much faster

Often instead we try to compute the **average-case** complexity

Algorithm Complexity

Example: What is the average time complexity of a linear search?

Assume an equal chance that x is at any position in list

Average the complexities for each possible position of x

If $x = a_1$ need 3 comparisons

If $x = a_2$ need 5 comparisons ...

If $x = a_i$ need $2i + 1$ comparisons

Algorithm Complexity

Example: What is the average time complexity of a linear search?

$$\frac{3+5+7+\cdots+(2n+1)}{n} = \frac{2(1+2+3+\cdots n)+n}{n}$$

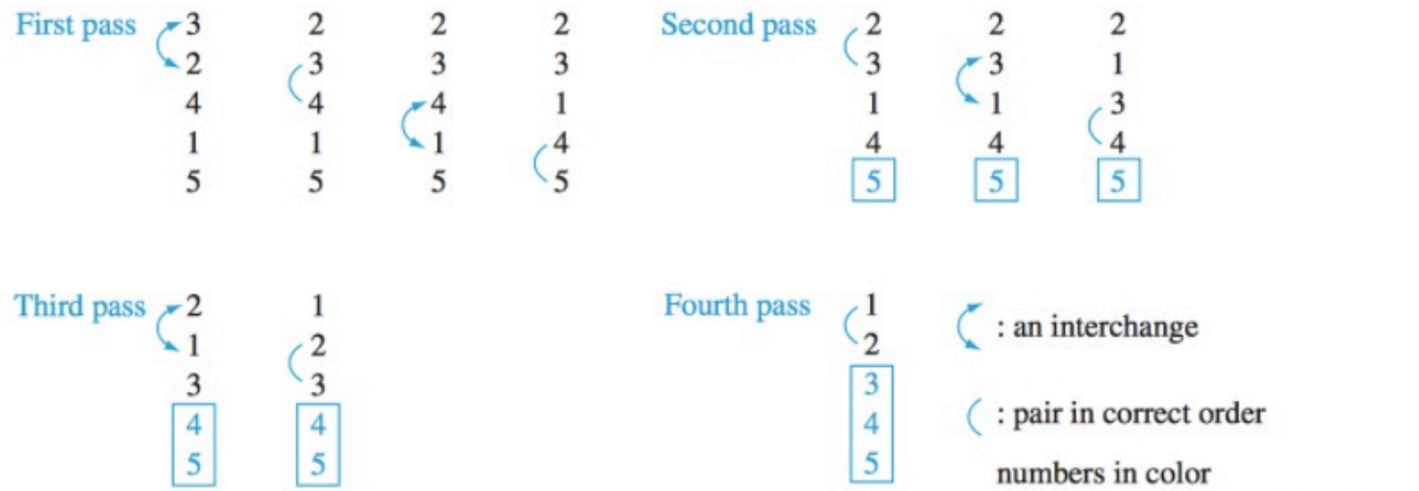
Recall: $1+2+3+\cdots n = \frac{n(n+1)}{2}$

$$= \frac{2(1+2+3+\cdots n)+n}{n} = \frac{n(n+1)+n}{n} = n+2$$

Average time complexity of LinearSearch is $n+2$

Algorithm Complexity

Example: What is the time complexity of Bubble Sort?



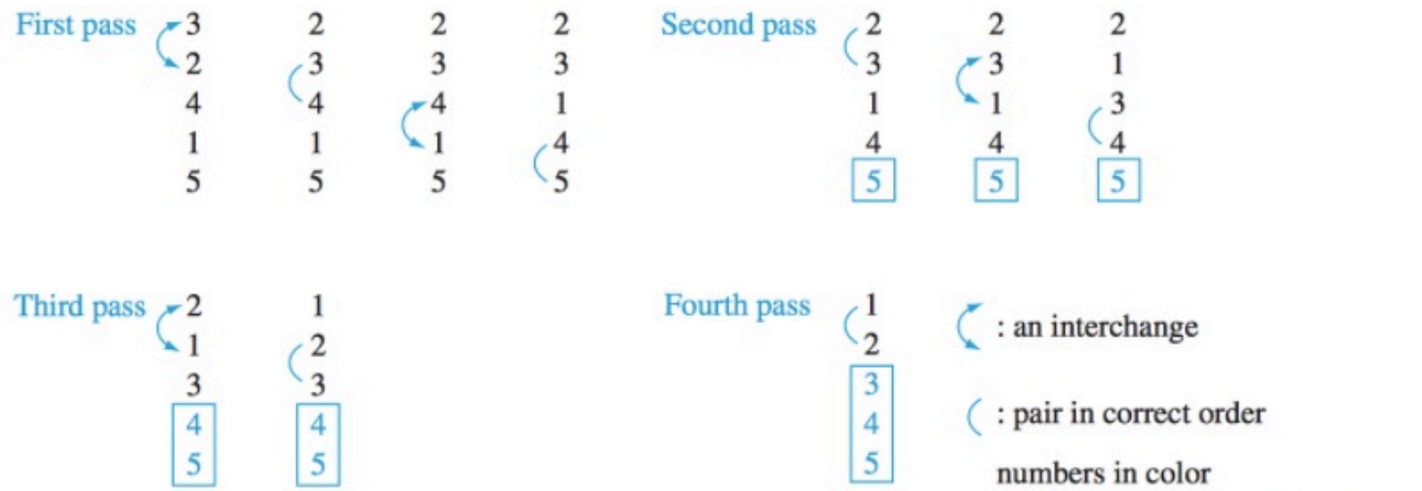
Again, we'll just count comparisons

Total: $(n-1)$

First Pass: $(n-1)$ comparisons

Algorithm Complexity

Example: What is the time complexity of Bubble Sort?



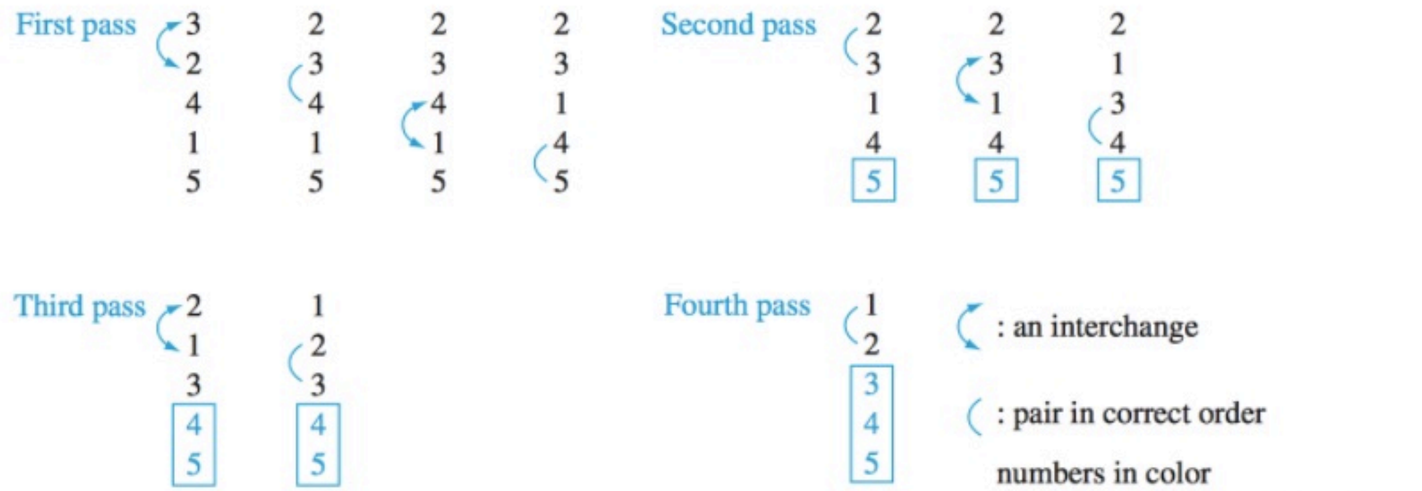
Again, we'll just count comparisons

Total: $(n-1)+(n-2)$

Second Pass: $(n-2)$ comparisons

Algorithm Complexity

Example: What is the time complexity of Bubble Sort?



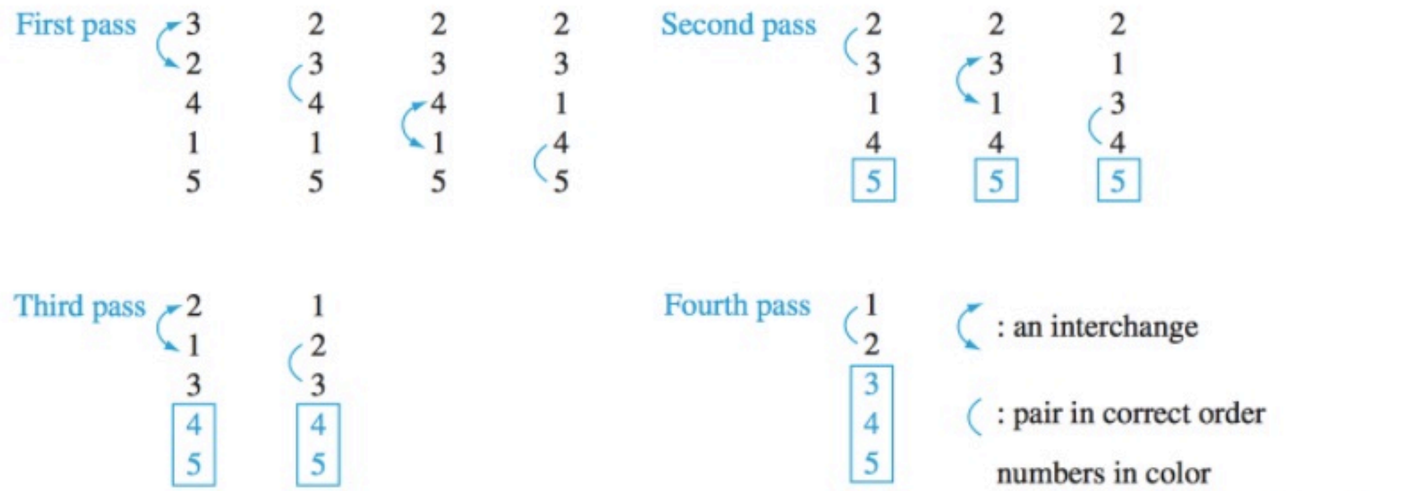
Again, we'll just count comparisons

Total: $(n-1)+(n-2)+\cdots+2+1$

Final Pass: 1 comparison

Algorithm Complexity

Example: What is the time complexity of Bubble Sort?



Again, we'll just count comparisons

Total: $(n-1)+(n-2)+\cdots+2+1=\dfrac{(n-1)n}{2}$

Algorithm Complexity

procedure `BubbleSort`$(a_1, a_2, \ldots, a_n)$

for $i := 1$ **to** $n - 1$

  for $j := 1$ **to** $n - i$

    if $a_j > a_{j+1}$ **then** interchange a_j and a_{j+1}

Slick way to do the count if you have pseudocode

Algorithm Complexity

procedure `BubbleSort`$(a_1, a_2, \ldots, a_n)$

for $i := 1$ **to** $n - 1$

  for $j := 1$ **to** $n - i$

    if $a_j > a_{j+1}$ **then** interchange a_j and a_{j+1}

Slick way to do the count if you have pseudocode

Count operations in inner-most loop

Turn loops into summations

Algorithm Complexity

procedure `BubbleSort`$(a_1, a_2, \ldots, a_n)$

for $i := 1$ **to** $n - 1$

    for $j := 1$ **to** $n - i$

        if $a_j > a_{j+1}$ **then** interchange a_j and a_{j+1}

Algorithm Complexity

procedure `BubbleSort`$(a_1, a_2, \ldots, a_n)$

for $i := 1$ **to** $n - 1$

  for $j := 1$ **to** $n - i$

    if $a_j > a_{j+1}$ **then** interchange a_j and a_{j+1}

$$\sum_{i=1}^{n-1} \sum_{j=1}^{n-i} 1 = \sum_{i=1}^{n-1} \left(\sum_{j=1}^{n-i} 1 \right) = \sum_{i=1}^{n-1} n - i = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i$$

$$= (n-1)n - \frac{(n-1)n}{2} = \frac{(n-1)n}{2}$$

Algorithm Complexity

Bubble Sort: $\frac{(n-1)n}{2}$ **Insertion Sort:** $\frac{n(n+1)}{2} - 1$

We compare algorithms at different levels of details

Bubble Sort: $\frac{1}{2}\left(n^2 - n\right)$ **Insertion Sort:** $\frac{1}{2}\left(n^2 - n\right) + n - 1$

So Insertion sort requires $n - 1$ more comparisons than Bubble sort

For large n though, the n^2 term dwarfs the n terms and the $1/2$ becomes irrelevant, so we might say they both use **roughly** n^2 comparisons

Algorithm Complexity

In a minute we'll precisely define what **roughly** means, but first ...

What can you say about the performance of an algorithm with n^2 complexity as n grows?

Q: How does the time compare between sorting a list and sorting a list twice as long?

Algorithm Complexity

In a minute we'll precisely define what **roughly** means, but first ...

What can you say about the performance of an algorithm with n^2 complexity as n grows?

Q: How does the time compare between sorting a list and sorting a list twice as long?

Can't really talk about specific time, but can talk about relative time

$$\frac{(2n)^2}{n^2} = \frac{4n^2}{n^2} = 4$$

Rule: If complexity is n^2 doubling size quadruples the time

Algorithm Complexity

There are several conventions that allow us to talk about the efficiency of algorithms without writing out their precise operation counts

Q: If we have two algorithms that solve the same problem, and one of them uses $100n^2 + 17n + 4$ operations and the other uses n^3 operations, which should you use?

Algorithm Complexity

There are several conventions that allow us to talk about the efficiency of algorithms without writing out their precise operation counts

Q: If we have two algorithms that solve the same problem, and one of them uses $100n^2 + 17n + 4$ operations and the other uses n^3 operations, which should you use?

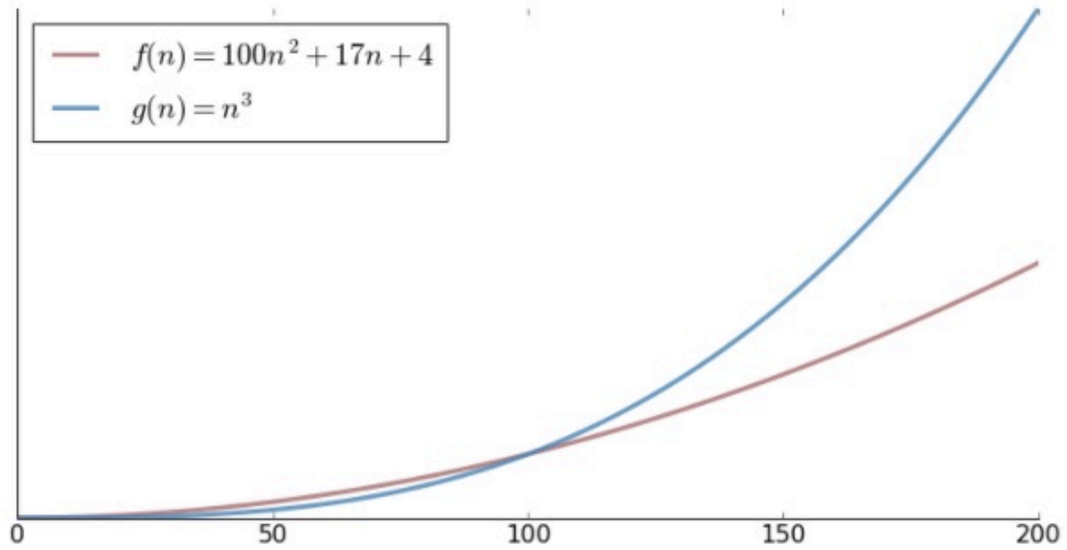
A: For small values of n, $100n^2 + 17n + 4$ might be less than n^3

But very quickly n^3 becomes much larger than $100n^2 + 17n + 4$

Algorithm Complexity

There are several conventions that allow us to talk about the efficiency of algorithms without writing out their precise operation counts

Q: If we have two algorithms that solve the same problem, and one of them uses $100n^2 + 17n + 4$ operations and the other uses n^3 operations, which should you use?



Algorithm Complexity

It's clear that the big difference in the two algorithms is that one depends on n^2 and the other on n^3.

We say that $100n^2 + 17n + 4$ is $\mathcal{O}(n^2)$

While n^3 is $\mathcal{O}(n^3)$

From these concise descriptions we see that the second algorithm will eventually be more expensive than the first.

Def: Let f and g be functions from the set of integers. We say that $f(n)$ is $\mathcal{O}(g(n))$ if there are constants C and k such that

$$|f(n)| \leq C|g(n)|$$

whenever $n > k$ [Said out loud: "$f(n)$ is big-oh of $g(n)$"]

Algorithm Complexity

Example: Show that $100n^2 + 17n + 4$ is $\mathcal{O}(n^2)$

Algorithm Complexity

Example: Show that $100n^2 + 17n + 4$ is $\mathcal{O}(n^2)$

We know that when $n > 1$ it's true that $n \leq n^2$ and $4 \leq n^2$

So, for $n > 1$ we have

$$100n^2 + 17n + 4 \leq 100n^2 + 17n^2 + n^2$$

So $100n^2 + 17n + 4 \leq 118n^2$ when $n > 1$

and so $100n^2 + 17n + 4$ is $\mathcal{O}(n^2)$ with $C = 118$ and $k = 1$

Algorithm Complexity

Notice that $f(n)$ is $\mathcal{O}(g(n))$ means that for some C and k, $f(n)$ is bounded from above by $g(n)$ after $n > k$

This def. also means that $100n^2 + 17n + 4$ is $\mathcal{O}(n^4)$ and $\mathcal{O}(n^{17})$

This is slightly unsatisfactory since saying that $100n^2 + 17n + 4$ is $\mathcal{O}(n^{17})$, while true, is not very informative

We'll come back to this in a second and see how we can say more reliable things

Algorithm Complexity

Example: Give a big-*O* estimate of $h(n) = 2n^2 + 5\log n$

Algorithm Complexity

Example: Give a big-*O* estimate of $h(n) = 2n^2 + 5\log n$

Note that for $n > 1, \log n \leq n \leq n^2$

So we have $h(n) = 2n^2 + 5\log n \leq 2n^2 + 5n^2 = 7n^2$ is $\mathcal{O}(n^2)$

Theorem: Suppose that $f_1(n)$ is $\mathcal{O}(g_1(n))$ and $f_2(n)$ is $\mathcal{O}(g_2(n))$. Then $(f_1 + f_2)(n)$ is $\mathcal{O}(\max(|g_1(n)|, |g_2(n)|))$

In words: A sum of functions is big-*O* of the biggest function

Theorem: Suppose that $f_1(n)$ is $\mathcal{O}(g_1(n))$ and $f_2(n)$ is $\mathcal{O}(g_2(n))$. Then $(f_1 f_2)(n)$ is $\mathcal{O}(g_1(n)g_2(n))$

Algorithm Complexity

OK, so $f(n)$ is $\mathcal{O}(g(n))$ tells us that f grows no faster than g

This gives us an upper bound on the growth of f

But it would also be nice to have a lower bound, to say for instance, $f(n)$ grows *at-least* as fast as $h(n)$

Def: Let f and g be functions from the set of integers. We say that $f(n)$ is $\Omega(h(n))$ if there are constants C and k such that

$$|f(n)| \geq C|h(n)|$$

whenever $n > k$ [Said out loud: ``$f(n)$ is big-Omega of $h(n)$"]

Algorithm Complexity

Example: Give a big-Omega estimate of $f(n) = 100n^2 + 17n + 4$

Algorithm Complexity

Example: Give a big-Omega estimate of $f(n) = 100n^2 + 17n + 4$

This one's pretty easy. For all $n > 0$

$$100n^2 + 17n + 4 \geq 100n^2$$

So $100n^2 + 17n + 4$ is $\Omega(n^2)$ with $C = 100$ and $k = 0$

Notice that $100n^2 + 17n + 4$ is both $\mathcal{O}(n^2)$ and $\Omega(n^2)$ we know that it grows no faster than a constant times n^2 and no slower than a constant times n^2

When this happens we say that $100n^2 + 17n + 4$ is $\Theta(n^2)$ [said ``big-Theta of n^2 "]

Algorithm Complexity

A big-Theta estimate of a function is kinda the holy grail

It says we know everything about the growth of the function

Def: If $f(n)$ and $g(n)$ are both $\Theta(h(n))$ we say that f and g have **order** $h(n)$, and further that f and g are the same order

Algorithm Complexity

A big-Theta estimate of a function is kinda the holy grail

It says we know everything about the growth of the function

Def: If $f(n)$ and $g(n)$ are both $\Theta(h(n))$ we say that f and g have **order** $h(n)$, and further that f and g are the same order