CS1113
Algorithm Complexity

Lecturer:

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

http://osullivan.ucc.ie/teaching/cs1113/

Classifying Algorithm Run-time

comparing linear and binary search

growth of functions

big-oh notation

proof techniques

Linear search vs Binary search

Previously, we looked at two algorithms for searching a sequence of data values: linear search and binary search.

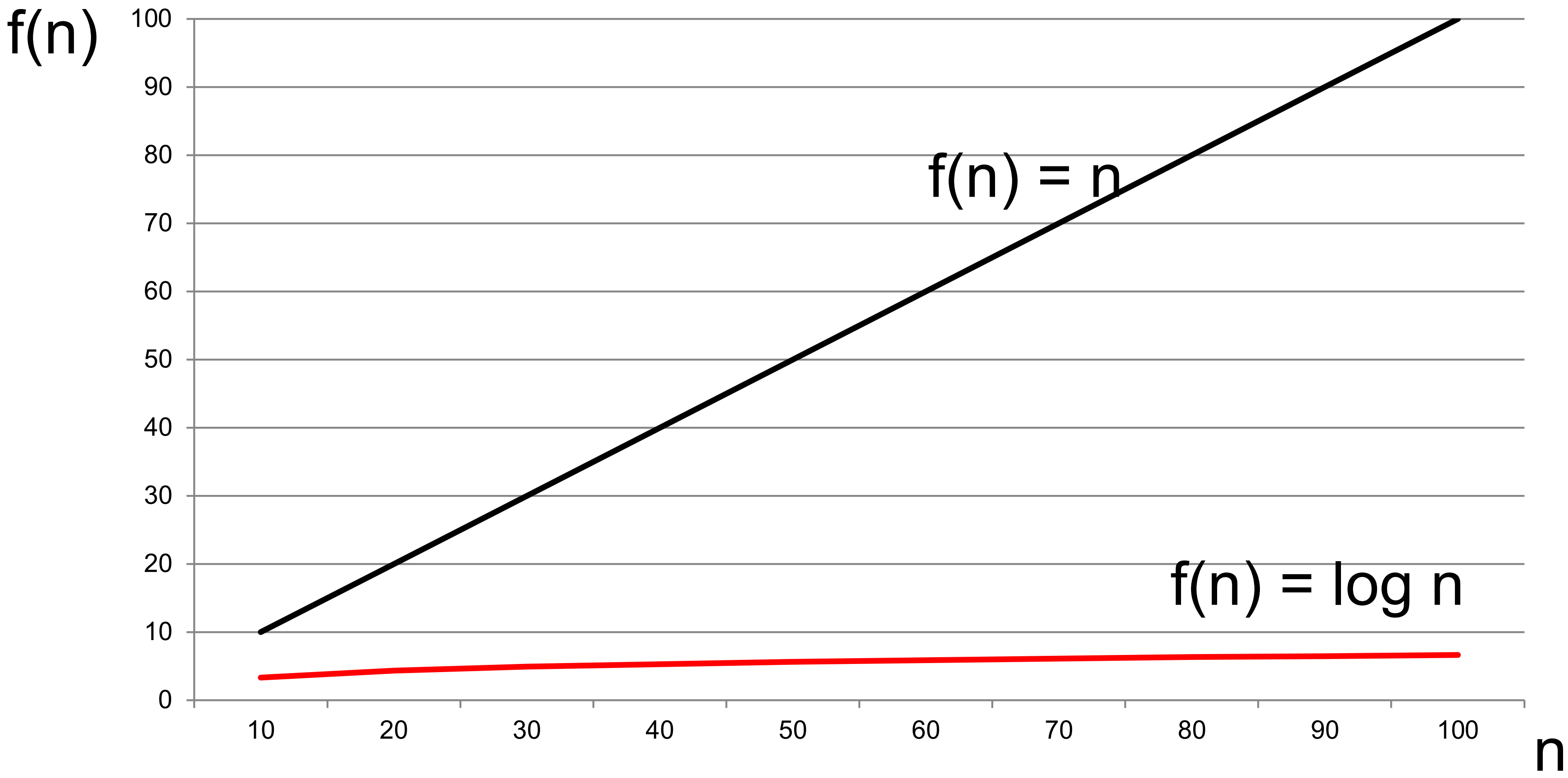
For each one, we worked out how much work it would have to do in the worst case. For a sequence of length *n*:

- linear search goes round its loop *n* times
- binary search goes round its loop $log_2\, n$ times

and they each do a similar amount of work inside their loop.

But what does this mean in practice? Is the difference significant?

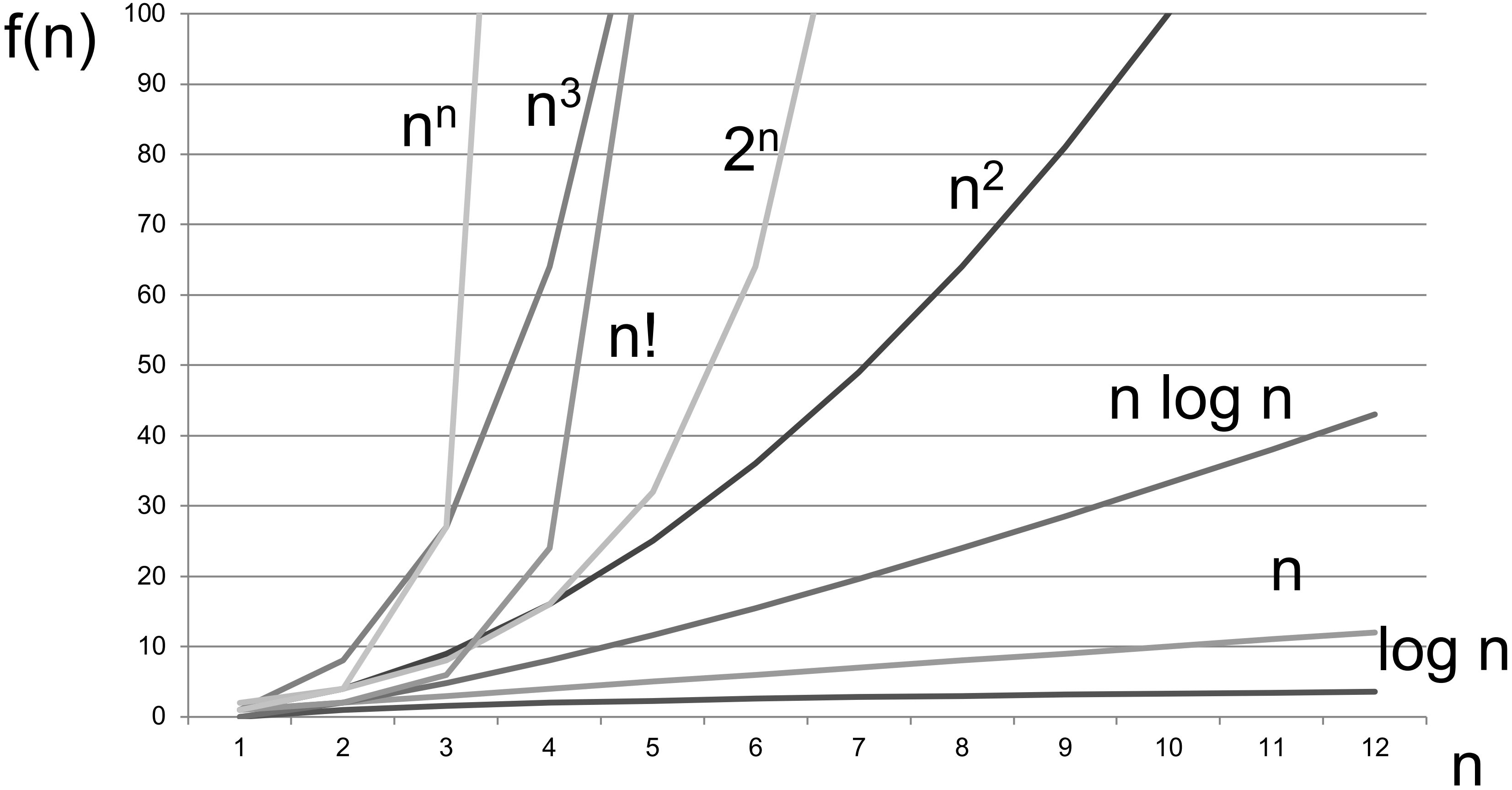
Comparing *n* and log *n*



If the sequence has 100 elements, linear search may be 15 times slower.

If the sequence has 1000 elements, linear search may be 100 times slower.

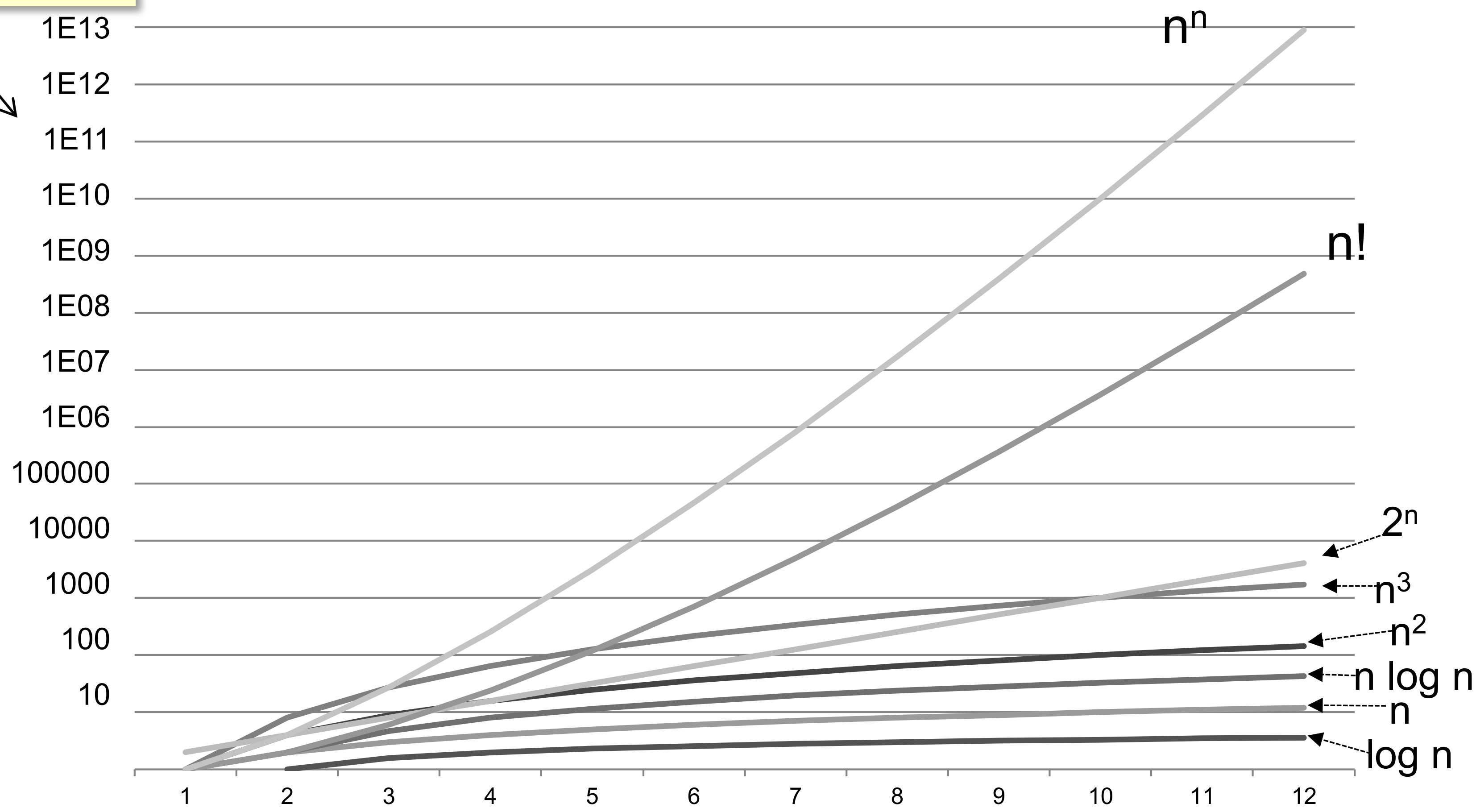
Comparing other functions



Note: n^n, n^3, n! off the scale by n=5.

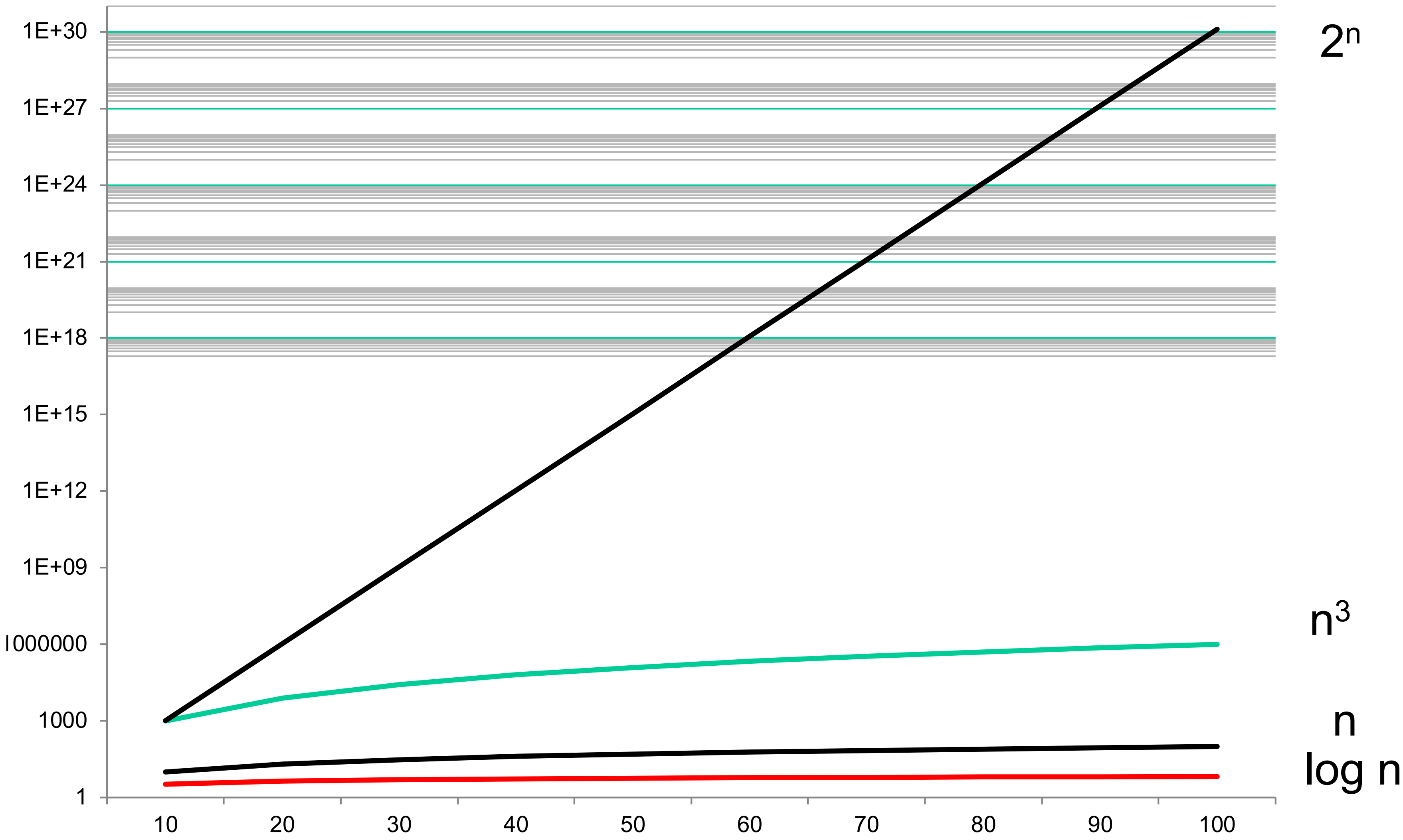
comparing functions

Note the scale



Note: when we reach n=12, n^3 is now less than 2^n and n!

Comparing functions for larger input



Comparing functions (continued)

- As we extended out the graph, the difference between the functions became clearer and more consistent

- For larger values of *n*, the differences are significant
  - choosing an algorithm with a high run-time will make your program useless for large data sets

- Also note that we were only comparing simple functions of *n*
  - we didn't look at, for example, n^2+n+3, $4n+1$, or $2n^2+5n$

- We only need a general picture of how quickly a function grows, and we use the simple functions as a reference
  - can we have any confidence in this comparison?

Big-oh notation

Consider two functions *f* and *g*, mapping positive integers to positive integers (so $f : N \rightarrow N$ and $g : N \rightarrow N$)

We will say *f(x)* is *O(g(x))* if there are two constant values *k* and *C* so that whenever *x* is bigger than *k*, $f(x) \leq C*g(x)$.

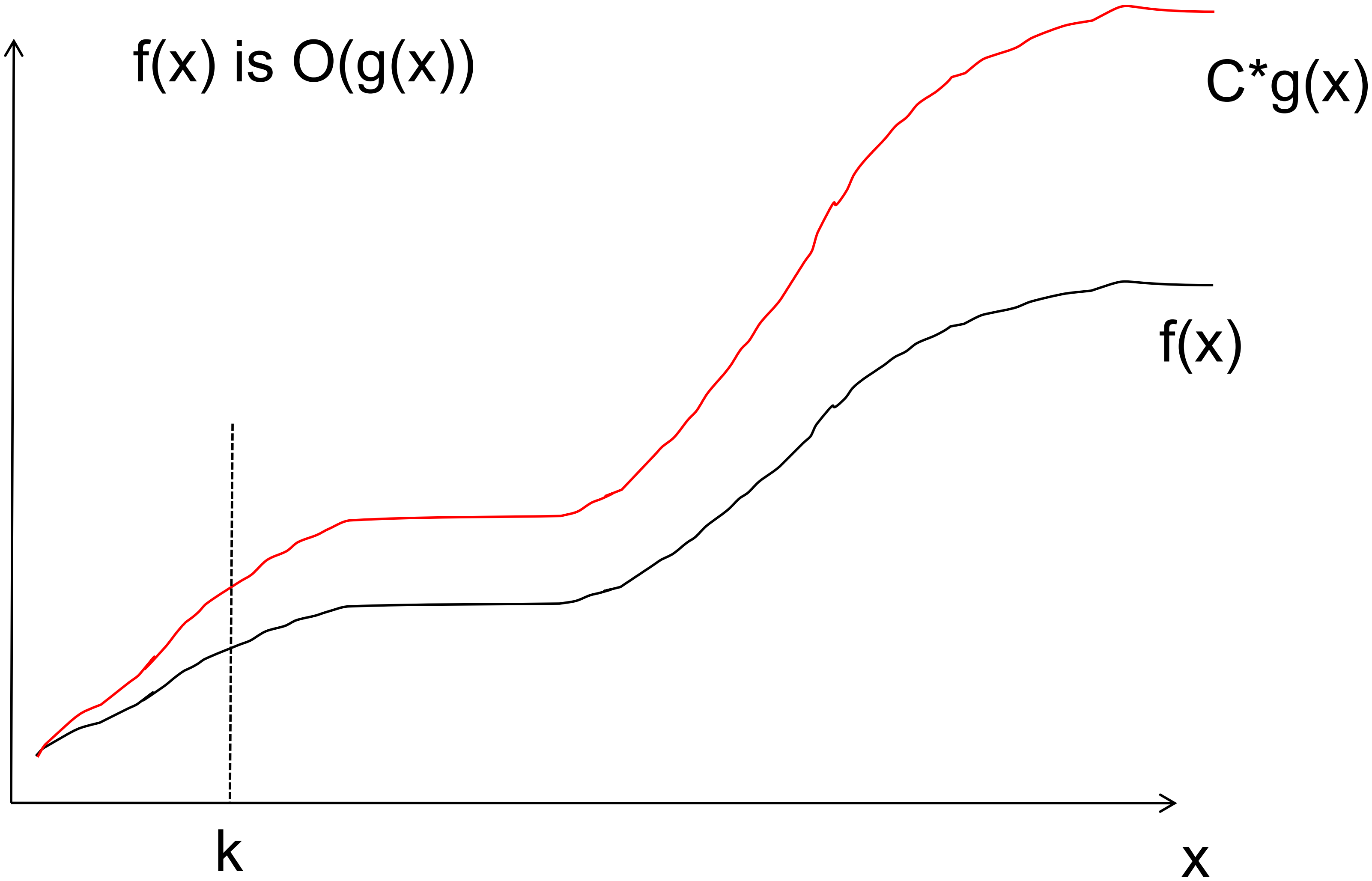
This means that when *x* is big enough, *f(x)* is never more than some constant multiple of *g(x)*, and so *f(x)* will not become drastically worse than *g(x)*.

We read this as "*f(x)* is big-oh of *g(x)*"

Formally,

$f(x)$ *is* $O(g(x))$ *if and only if* $\exists k \, \exists C \, \forall x > k \;\; f(x) \leq C*g(x)$

What does big-oh mean for function growth?



For all values of x > k, f(x) will be below the red line.

Big-oh Example

x^2+1 is $O(x^2)$

Proof [From the definition of O(.), this claims that for some *k*, there is a constant *C* so that whenever $x>k$, then $x^2+1 \leq C^*x^2$. That is what we want to prove – i.e. find *k* and *C*.]

When $x>0$, $x^2+1 \leq x^2+x^2 = 2x^2$ ($x \in$N, so $x \geq 1$, and $x^2 \geq 1$)

So if k=0 and C=2, we have: whenever x>k, $x^2+1 \leq C^*x^2$.

Therefore, x^2+1 *is* $O(x^2)$.

This says that x^2+1 does not grow significantly faster than x^2. So any algorithm taking x^2+1 steps will be roughly similar to an algorithm taking x^2 steps.

Proof techniques: direct proof

- On the previous slide, we proved that x^2+1 is $O(x^2)$.
- By proof, we mean a convincing argument that the statement is correct.
- In this case, we gave a direct proof
  - we started with what we knew, and applied reasoning to turn those known facts into the statement we wanted
- in logic, we used direct proof to prove that conclusions followed from initial facts
  - the logic proofs were formal – we listed all facts, showed all steps, and used only valid rules of inference
  - normally, our proofs will be more informal – we may skip steps, and rely on human understanding

Example 2

$4x^2+5x-3$ is $O(x^2)$

Proof

So $4x^2+5x-3$ does not grow too fast compared to x^2

Example 3

x^3+2x^2+2 is not $O(x^2)$

Proof [First, we will assume x^3+2x^2+2 is $O(x^2)$, and then we will show that this leads to a contradiction.]

Suppose x^3+2x^2+2 ***is*** $O(x^2)$. Then, by the definition of O(.), there is some pair of constants k and C such that for every $x>k$, $x^3+2x^2+2 \leq C*x^2$.

Consider any value of x bigger than k.
So we have $x^3+2x^2+2 \leq Cx^2$
Then we must have $x+2+2/x^2 \leq C$
Therefore $x+2 \leq C$ *(since $2/x^2 > 0$, and we are no longer adding it)*
But if we pick any value of x s.t. $x>C$, we must have $x+2 > C$.
These last two statements contradict each other, so something is wrong in our reasoning. The only thing we could have got wrong is the assumption that x^3+2x^2+2 is $O(x^2)$. Therefore, x^3+2x^2+2 is not $O(x^2)$.

The implications

By the previous example, x^3+2x^2+2 will grow much faster than x^2 when x is large.

So if we have two algorithms, A and B, for handling the same task, where the input is of size n, and

- the number of steps required by A is $O(n^2)$
- the number of steps required by B is n^3+2n^2+2,

then if the input might be large, we should prefer algorithm A.

For large input, algorithm B will require more time than A, and as the input gets larger, the gap in performance will continue to get bigger.

Proof Technique: proof by contradiction

The proof of the previous example used proof by contradiction.

We wanted to prove some statement.

We first assume the negation of the statement. We then apply reasoning to show that that assumption leads to a contradiction. If all our reasoning is correct, then the only thing that could be wrong is the assumption. Therefore, the statement must be correct.

Polynomial growth

If $f(x) = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_2 x^2 + a_1 x + a_0$
where each a_i is a constant, then $f(x)$ is $O(x^n)$

<u>Proof</u> Let $x>1$

$$
\begin{aligned}
f(x) &= a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_2 x^2 + a_1 x + a_0 \\
&\leq |a_n|x^n + |a_{n-1}|x^{n-1} + |a_{n-2}|x^{n-2} + \ldots + |a_2|x^2 + |a_1|x + |a_0| \\
&\leq |a_n|x^n + |a_{n-1}|x^n + |a_{n-2}|x^n + \ldots + |a_2|x^n + |a_1|x^n + |a_0|x^n \\
&= (|a_n| + |a_{n-1}| + |a_{n-2}| + \ldots + |a_2| + |a_1| + |a_0|)x^n \\
&= Cx^n, \text{ for } C = |a_n| + |a_{n-1}| + |a_{n-2}| + \ldots + |a_2| + |a_1| + |a_0|
\end{aligned}
$$

so we can take k=0, C = $|a_n| + |a_{n-1}| + |a_{n-2}| + \ldots + |a_2| + |a_1| + |a_0|$
and so $f(x)$ is $O(x^n)$

If the running time of an algorithm is a polynomial function of the input size x, we only need to worry about the highest power of x

Summary

- we are interested in how many steps our algorithms might need in the worst case
  - we call this the (worst case) run-time of the algorithm
- we state the run-time as a function of the size of the input
  - if the function grows too quickly, then programs that implement the algorithms become inefficient
- we use the big-oh notation to classify different functions
  - if $f(x)$ is not $O(g(x))$, then $f(x)$ will soon become significantly larger than $g(x)$
  - if the run-time function is a polynomial, we only care about the highest power
- we have a simple hierarchy of run-times based on big-oh:

$$\log n < n < n \log n < n^2 < n^3 < \ldots < 2^n < n! < n^n$$

Next lecture ...

algorithms for sorting data