Complexity
(Weiss chapter 2)

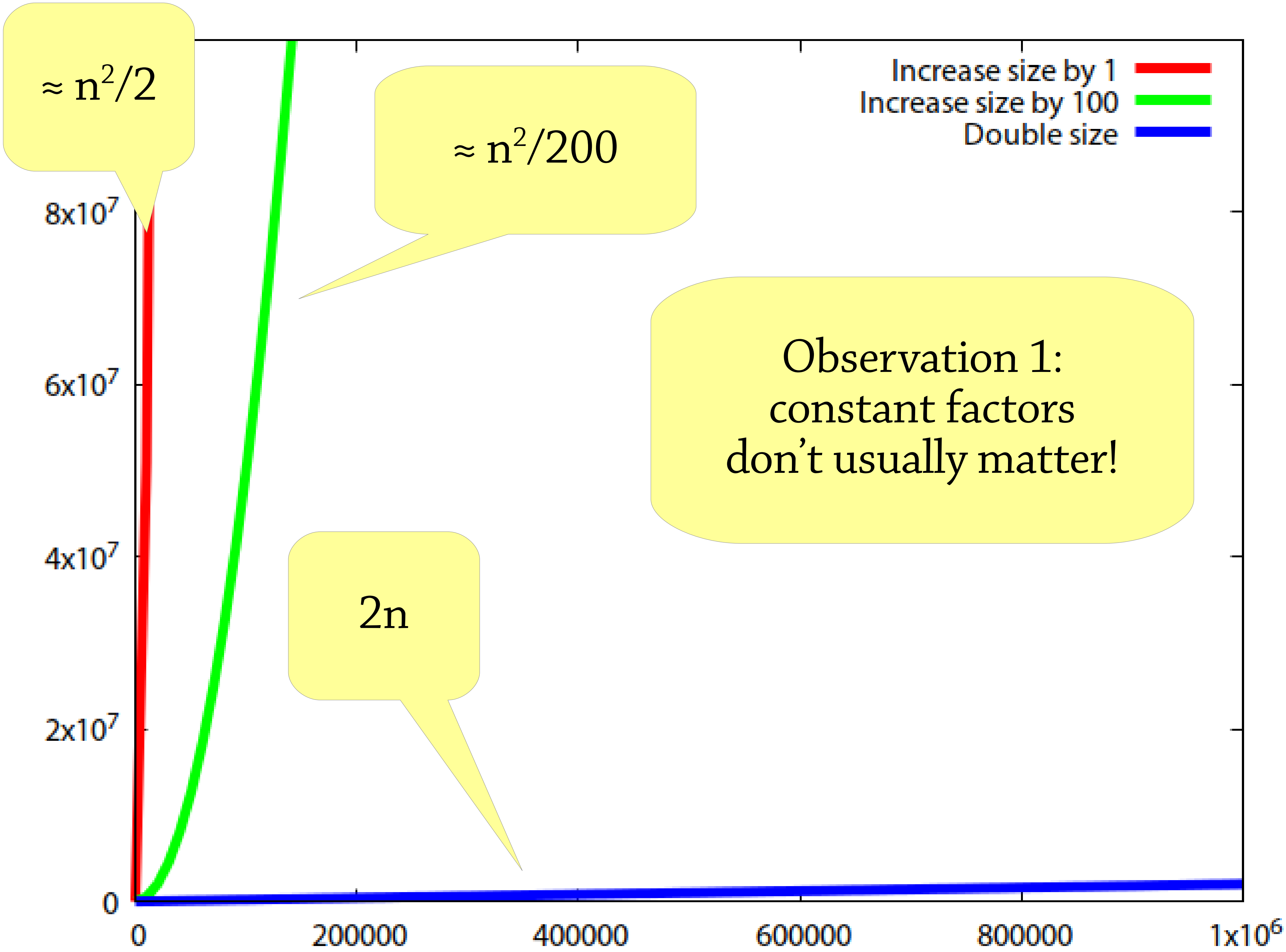
Complexity

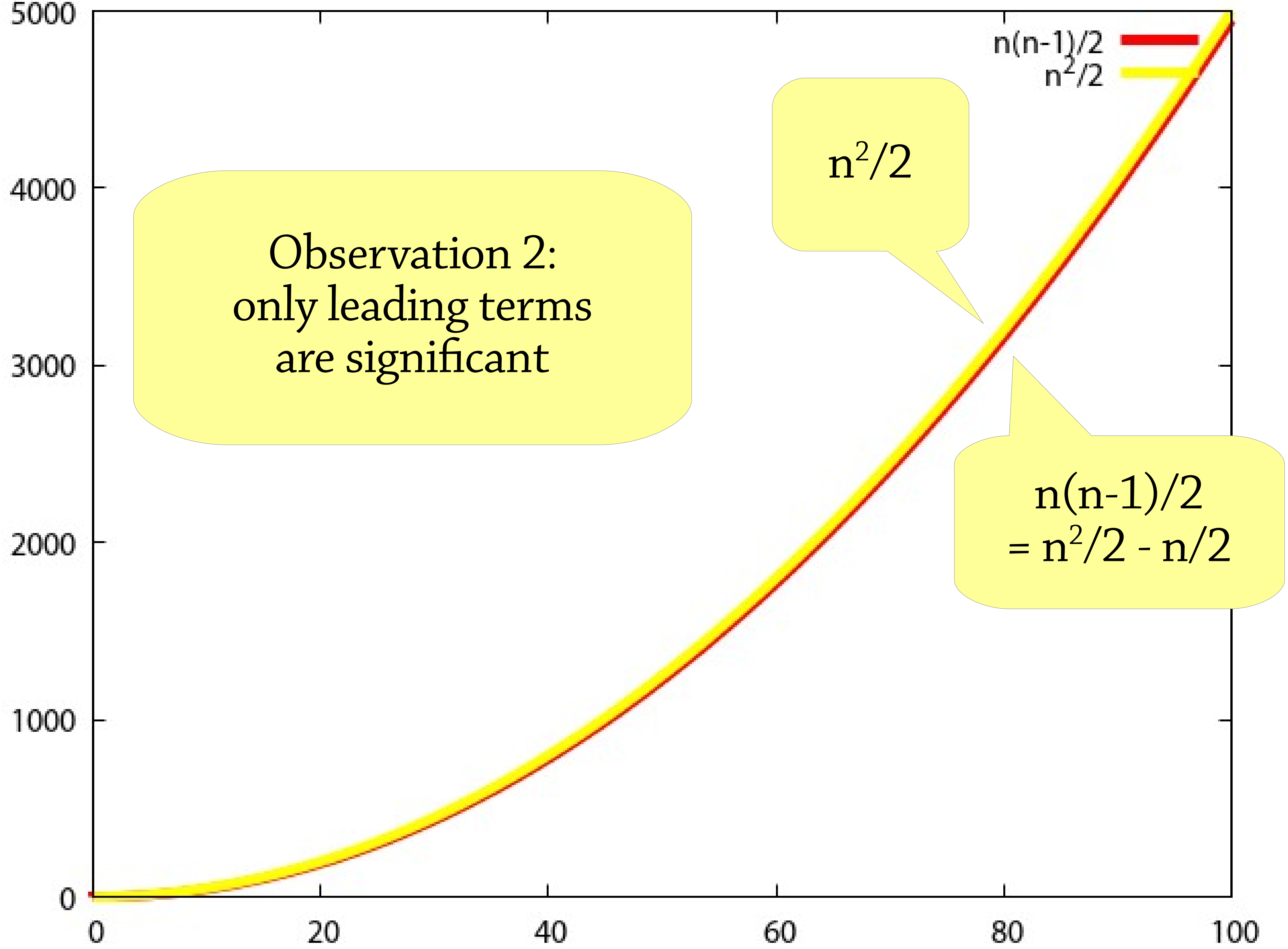
This lecture is all about *how to describe the performance of an algorithm*

Given an algorithm, and (e.g.) the size of the input, can we come up with a formula for the runtime of the algorithm?

- Problem: runtime may vary based on exact input – solution: look at *worst-case* runtime for a given size
- Problem: calculating an exact runtime requires deep knowledge of the machine the program will be run on – solution: count *number of steps* instead
- Problem: the formula is usually very large and annoying to calculate – solution: the rest of this lecture!

Idea: *asymptotic complexity* – what is the performance like when n is large?


≈ n²/2
≈ n²/200
Observation 1: constant factors don't usually matter!
2n
Increase size by 1
Increase size by 100
Double size
8x10⁷
6x10⁷
4x10⁷
2x10⁷
0
0
200000
400000
600000
800000
1x10⁶



5000
4000
3000
2000
1000
0
0
20
40
60
80
100
n(n-1)/2
n^2/2
Observation 2:
only leading terms
are significant
n^2/2
n(n-1)/2
= n^2/2 - n/2

Big-O notation

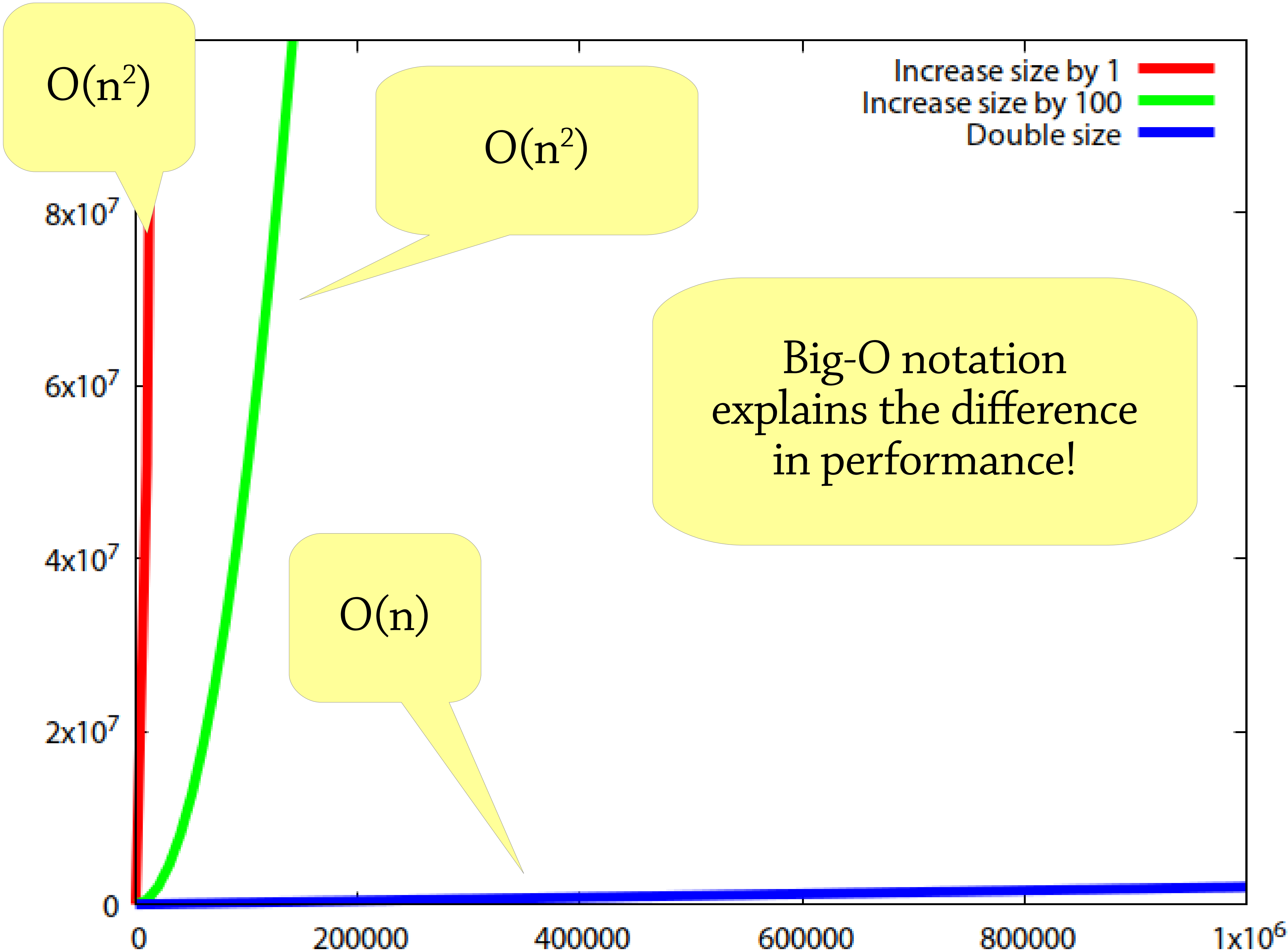
When n is large:

- only leading terms are significant
- constant factors don't (usually) matter

Main concept in this lecture: *big-O notation*, which allows us to ignore all those details in our formulas

The runtime of the three file copying programs is:

- The first one: $n(n-1)/2$ is $\mathbf{O(n^2)}$ ("big-O n-squared")
- The second one: $n(n-100)/2$ is $\mathbf{O(n^2)}$ too
- The third one: $2n$ is $\mathbf{O(n)}$
- **O(...)** means roughly: "proportional to ..., when n is large enough"


O(n²)
O(n²)
Big-O notation explains the difference in performance!
O(n)
Increase size by 1
Increase size by 100
Double size
8x10⁷
6x10⁷
4x10⁷
2x10⁷
0
0
200000
400000
600000
800000
1x10⁶

Time complexity

With big-O notation, it doesn't matter whether we count steps or time!

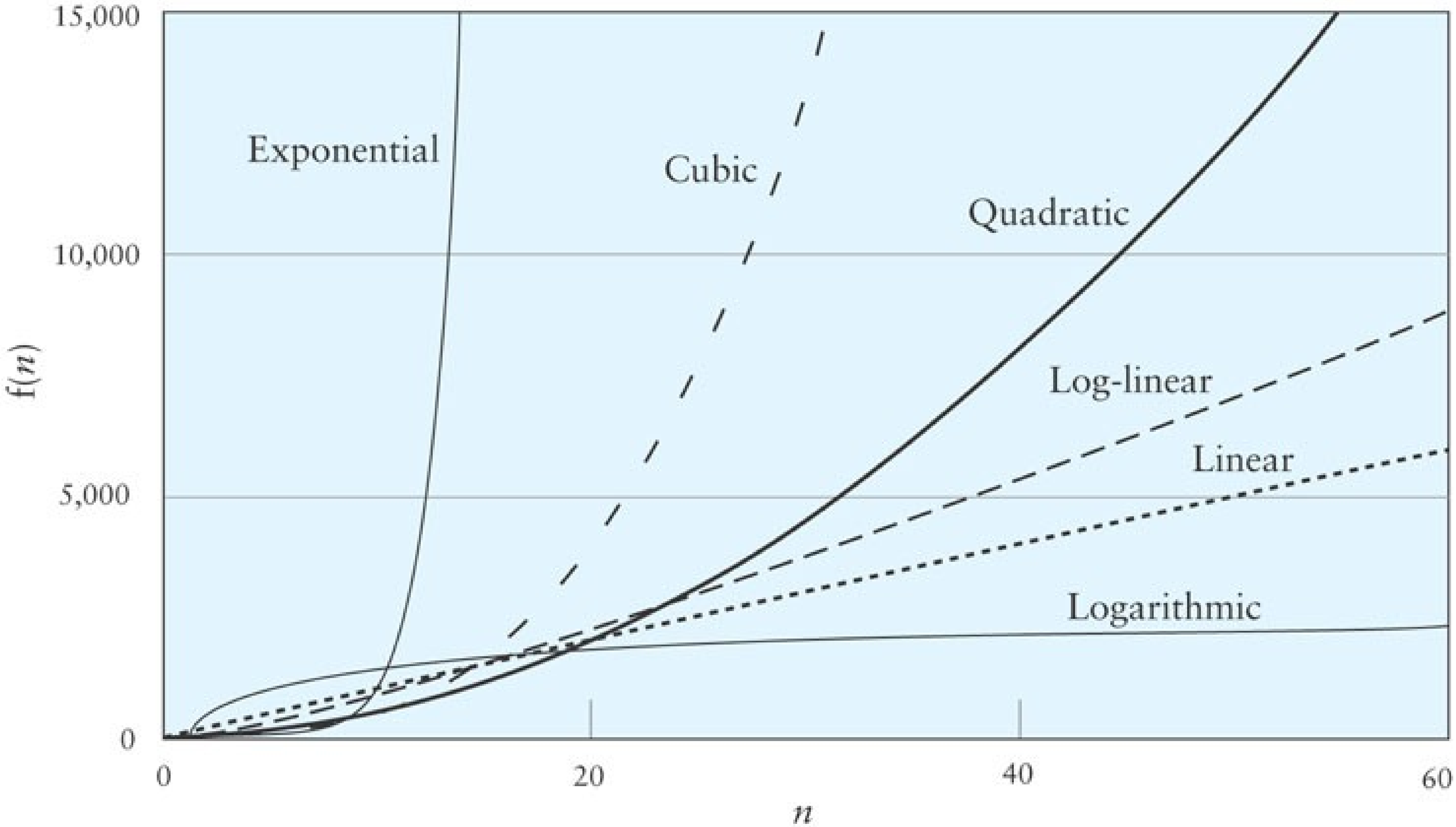
As long as each step takes a constant amount of time:

- if the number of steps is proportional to n^2
- then the amount of time is proportional to n^2

We say that the algorithm has $O(n^2)$ *time complexity* or simply *complexity*

Common complexities

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
$O(2^n)$	Exponential


15,000
10,000
5,000
0
f(n)
Exponential
Cubic
Quadratic
Log-linear
Linear
Logarithmic
0
20
40
60
n

Quiz

An algorithm takes $O(n)$ time to run. What happens to the runtime if the size of the input is doubled?

What about if the algorithm takes $O(n^2)$ time to run?

How does this explain the following facts:

- In the slow file-copying program, it started quickly but gradually got slower as it read the file
- In the fast file-copying program, it carried on at a constant rate

Growth rates

Imagine that we double the input size from n to 2n.

If an algorithm is...

- O(1), then it takes the same time as before
- O(log n), then it takes a constant amount more
- O(n), then it takes twice as long
- O(n log n), then it takes twice as long plus a little bit more
- O(n^2), then it takes four times as long
  - This explains why the slow file reading programs started quickly, but then gradually slowed down as they continued reading the file. How?

If an algorithm is O(2^n), then adding *one element* makes it take twice as long

Big O tells you *how the performance of an algorithm scales with the input size*

Big O mathematically

Big O, formally

Big O measures the growth of a *mathematical function*

- Typically a function T(*n*) giving the number of steps taken by an algorithm on input of size *n*
- But can also be used to measure *space complexity* (memory usage) or anything else

So for the file-copying program:

- $T(n) = n(n-1)/2$
- $T(n)$ is $O(n^2)$
- In general, $T(n)$ is $O(f(n))$, for some function f
- We often abuse notation and write "$T(n) = O(f(n))$"

Big O, formally

What does it mean to say "$T(n)$ is $O(f(n))$"?

- e.g. $T(n)$ is $O(n^2)$

We could say it means $T(n)$ is proportional to $f(n)$

- i.e. $T(n) = k \times f(n)$ for some k
- e.g. $T(n) = n^2/2$ is $O(n^2)$ (let $k = ½$)

But this is too restrictive!

- We want $T(n) = n(n-1)/2$ to be $O(n^2)$
- We want $T(n) = n^2 + 1$ to be $O(n^2)$

Big O, formally

Instead, we say that T(n) is O(f(n)) if:

- T(n) ≤ k × f(n) for some k,
  i.e. T(n) is proportional to f(n) *or lower*!
- This only has to hold for *big enough* n:
  i.e. for all n above some threshold n_0

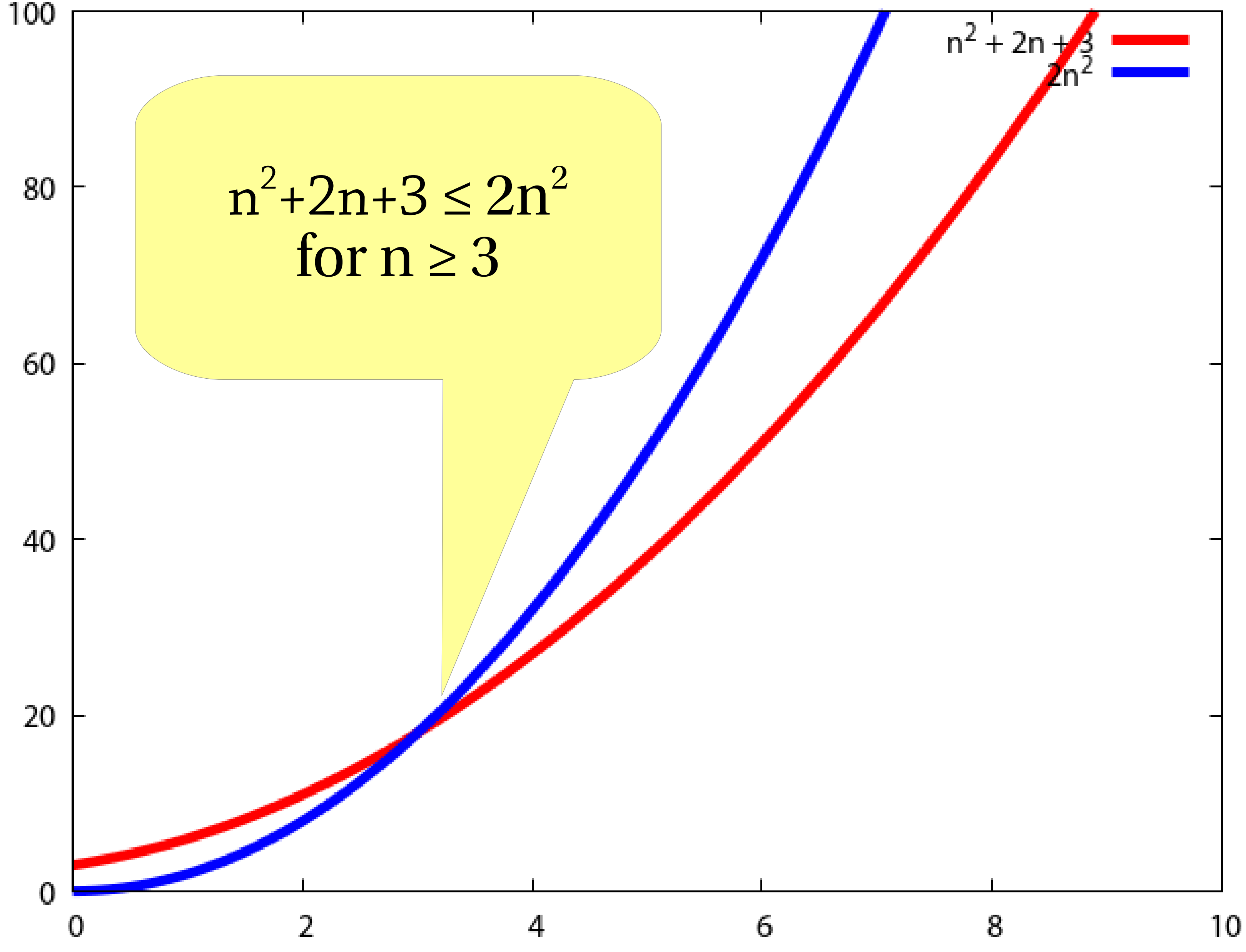
If you draw the graphs of T(n) and k × f(n), at some point the graph of k × f(n) must permanently overtake the graph of T(n)

- In other words, T(n) grows more slowly than k × f(n)

Note that big-O notation is allowed to *overestimate* the complexity!

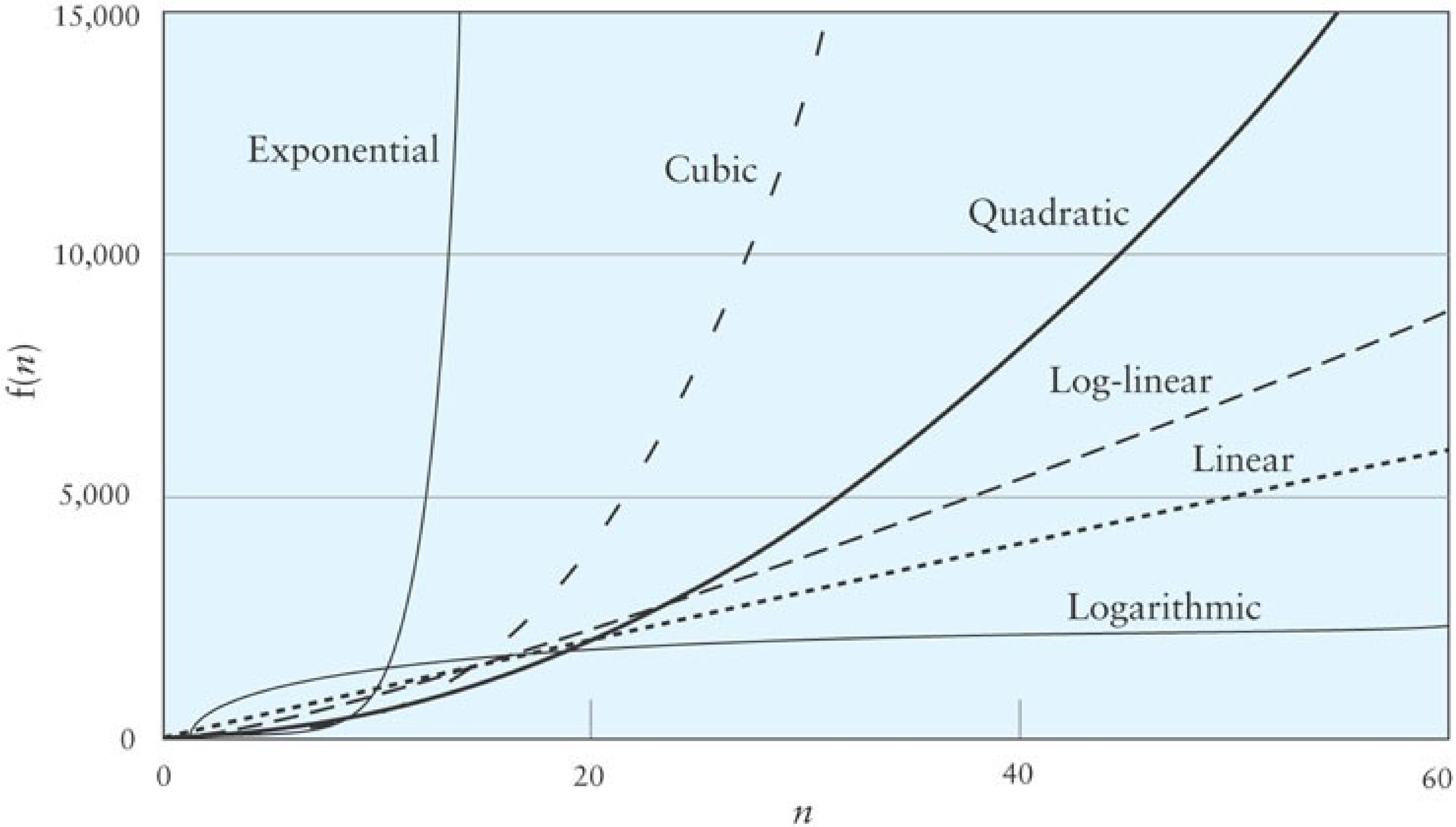
- k × f(n) is an *upper bound* on T(n)

An example: $n^2 + 2n + 3$ is $O(n^2)$



Quiz

- Is $3n + 5$ in $O(n)$?
- Is $n^2 + 2n + 3$ in $O(n^3)$?
- Is it in $O(n^2)$?
- Is it in $O(n)$?
- Why do we need the threshold n_0?



15,000
10,000
5,000
0
f(n)
Exponential
Cubic
Quadratic
Log-linear
Linear
Logarithmic
0
20
40
60
n

Adding big O

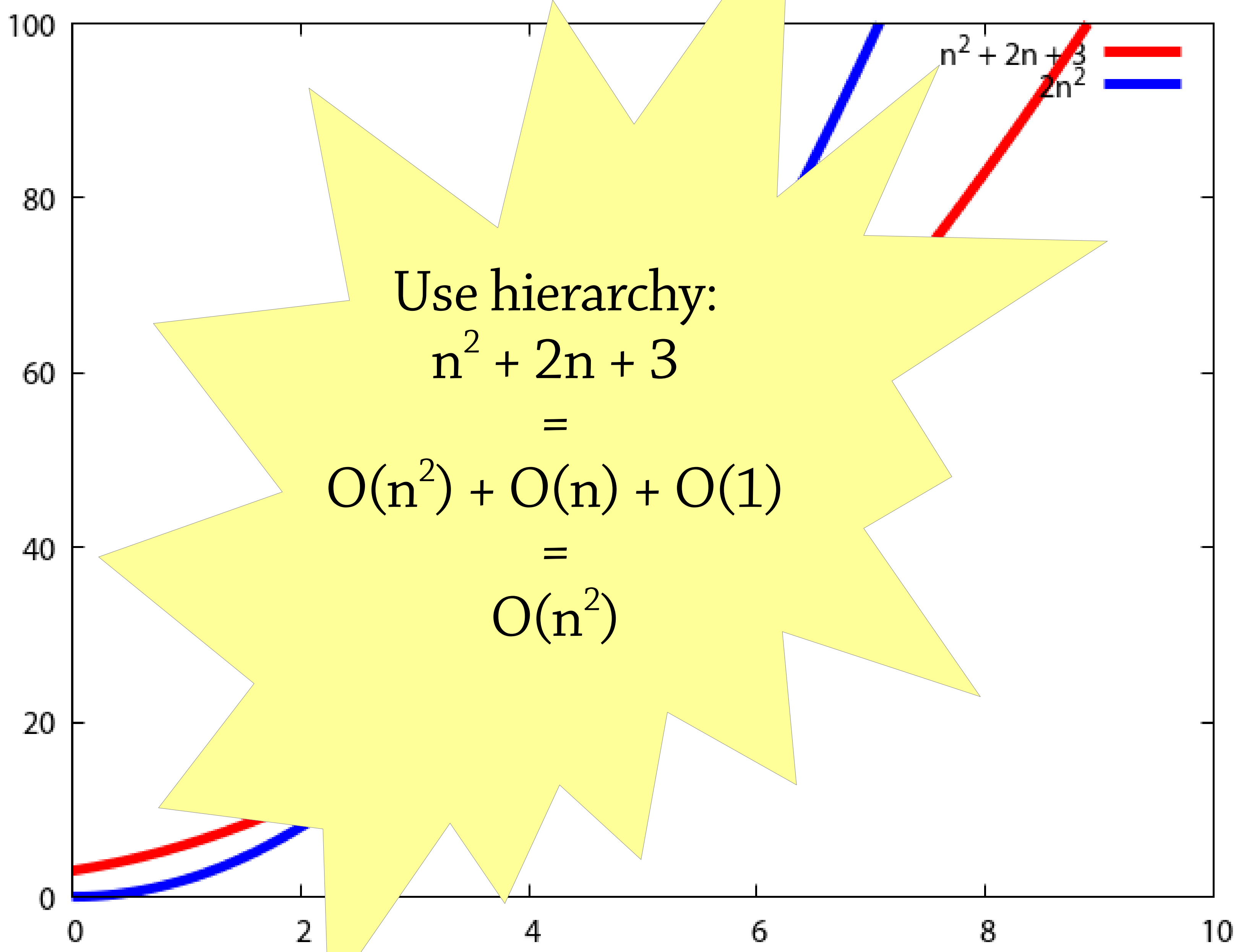
Some functions grow faster than others:

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

When adding two functions, the faster-growing function "wins":

- $O(1) + O(\log n) = O(\log n)$
- $O(\log n) + O(n^k) = O(n^k)$ (if $k \geq 0$)
- $O(n^j) + O(n^k) = O(n^k)$, if $j \leq k$
- $O(n^k) + O(2^n) = O(2^n)$

An example: $n^2 + 2n + 3$ is $O(n^2)$



Use hierarchy:

$$n^2 + 2n + 3$$
$$=$$
$$O(n^2) + O(n) + O(1)$$
$$=$$
$$O(n^2)$$

Quiz

What are these in Big O notation (simplified as far as possible)?

- $n^2 + 11$
- $2n^3 + 3n + 1$
- $n^4 + 2^n$

Just use hierarchy!

$$n^2 + 11 = O(n^2) + O(1) = O(n^2)$$

$$2n^3 + 3n + 1 = O(n^3) + O(n) + O(1) = O(n^3)$$

$$n^4 + 2^n = O(n^4) + O(2^n) = O(2^n)$$

Multiplying big O

$O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$

- e.g., $O(n^2) \times O(\log n) = O(n^2 \log n)$

You can drop constant factors:

- $k \times O(f(n)) = O(f(n))$, if k is constant
- e.g. $2 \times O(n) = O(n)$

(Exercise: show that these are true)

Quiz

What is $(n^2 + 3)(2^n \times n) + \log_{10} n$ in Big O notation?

Answer

$(n^2 + 3)(2^n \times n) + \log_{10} n$
$= O(n^2) \times O(2^n \times n) + O(\log n)$
$= O(2^n \times n^3) + O(\log n)$ (multiplication)
$= O(2^n \times n^3)$ (hierarchy)

$\log_{10} n = \log n / \log 10$
i.e. log n times a
constant factor

Reasoning about programs

Complexity of a program

Most "primitive" operations take O(1) time:

```
int add(int x, int y) {
  return x + y;
}
```

(Exception: creating an array of length n takes O(n) time)

This is called the *uniform cost model,* because all primitive operations are assigned the same cost

Complexity of a program

What about loops?

(Assume the array size is *n*)

```
boolean member(Object[] array, Object x) {
  for (int i = 0; i < array.length; i++)
    if (array[i].equals(x))
      return true;
  return false;
}
```

Complexity of a program

What about loops?

(Assume the array size is *n*)

```
boolean member(Object[] array, Object x) {
  for (int i = 0; i < array.length; i++)
    if (array[i].equals(x))
      return true;
  return false;
}
```

Loop runs
O(n) times

Loop body takes
O(1) time

O(1) × O(n) = **O(n)**

Complexity of loops

The complexity of a loop is:
the number of times it runs
times the complexity of the body

For nested loops, start from the innermost loop and work your way outwards!

What about this one?

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < a.length; j++)
      if (a[i].equals(a[j]) && i != j)
        return false;
  return true;
}
```

What about this one?

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.le
    for (int j = 0; j < a.length; j++)
      if(a[i].equals(a[j]) && i != j)
                e;
  ret
}
```

Outer loop runs
n times:
$O(n) \times O(n) = O(n^2)$

Inner loop runs
n times:
$O(n) \times O(1) = O(n)$

Loop body:
$O(1)$

What about this one?

```
void function(int n) {
  for(int i = 0; i < n*n; i++)
    for (int j = 0; j < n/2; j++)
      “something taking O(1) time”
}
```

What about this one?

```
void function(int n) {
  for(int i = 0; i < n*n;
    for (int j = 0; j < n/2; j++)
      “something taking O(1) time”
}
```

Outer loop runs
n^2 times:
$\mathbf{O(n^2)} \times O(n) = O(n^3)$

Inner loop runs
$n/2 = \mathbf{O(n)}$ times:
$O(n) \times O(1) = O(n)$

Loop body:
$O(1)$

Here's a new one

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < i; j++)
      if (a[i].equals(a[j]))
        return false;
  return true;
}
```

Here's a new one

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < i; j++)
      if (a[i].equals(a[j]))
              e;
  ret
}
```

Inner loop is
i × O(1) = O(i)??
But it should be
in terms of n?

Body is O(1)

Here's a new one

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < i; j++)
      i    a[i] equals(a[j]))
              e;
  ret
}
```

i < n, so **i is O(n)**
So loop runs **O(n)**
times, complexity:
$O(n) \times O(1) = O(n)$

Body is O(1)

Here's a new one

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.le
    for (int j = 0; j < i; j++)
      i      [i] equals(a[j]))
                      e;
  ret
}
```

Outer loop runs
n times:
$O(n) \times O(n) = O(n^2)$

i < n, so **i is O(n)**
So loop runs **O(n)**
times, complexity:
$O(n) \times O(1) = O(n)$

Body is O(1)

Three nested loops

```
void something(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < i; j++)
      for (int k = 0; k < j; k++)
        “something that takes 1 step”
}
```

$i < n$, $j < n$, $k < n$,
so all three loops run **O(n)** times
Total runtime is
$O(n) \times O(n) \times O(n) \times O(1) = \mathbf{O(n^3)}$

What's the complexity?

```
void something(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 1; j < a.length; j *= 2)
      … // something taking O(1) time
}
```

Outer loop is O(n log n)

What's the complexity?

Inner loop is O(log n)

```
void something(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 1; j < a.length; j *= 2)
      … // something taking O(1) time
}
```

A loop running through i = 1, 2, 4, …, n runs **O(log n)** times!

While loops

```
long squareRoot(long n) {
    long i = 0;
    long j = n;
    while (i < j) {
        long k = (i + j) / 2;
        if (k*k <= n) i = k;
        else j = k-1;
    }
    return i;
}
```

Each iteration takes O(1) time...
but how many times does the loop run?

While loops

```
long squareRoot(long n) {
    long i = 0;
    long j = n;
    while (i < j) {
        long k = (i + j) / 2;
        if (k*k <= n) i = k;
        else j = k-1;
    }
    return i;
}
```

Each iteration takes O(1) time

...and halves `j-i`, so **O(log n)** iterations

Summary: loops

Basic rule for complexity of loops:

- Number of iterations times complexity of body
- for (int i = 0; i < n; i++) ...: n iterations
- for (int i = 1; i ≤ n; i *= 2): O(log n) iterations
- While loops: have to work out number of iterations

If the complexity of the body depends on the value of the loop counter:

- e.g. O(i), where 0 ≤ i < n
- You can safely round i up to O(n)!

Sequences of statements

What's the complexity here?
(Assume that the loop bodies are O(1))

```
for (int i = 0; i < n; i++) …
for (int i = 1; i < n; i *= 2) …
```

Sequences of statements

What's the complexity here?
(Assume that the loop bodies are O(1))

```
for (int i = 0; i < n; i++) …
for (int i = 1; i < n; i *= 2) …
```

First loop: **O(n)**
Second loop: **O(log n)**
Total: O(n) + O(log n) = **O(n)**

For sequences, add the complexities!

Modelling a slow dynamic array

```
int[] array = {};
for (int i = 0; i < n; i+=100) {
  int[] newArray =
    new int[array.length+100];
  for (int j = 0; j < i; j++)
    newArray[j] = array[j];
  newArray = array;
}
```

Modelling a slow dynamic array

```
int[] array = {};
for (int i = 0; i < n;
  int[] newArray =
    new int[array.length+100];
  for (int j = 0; j < i; j++)
    newArray[  = array[i];
  newArray =
}
```

Rest of loop body **O(1)**,
so loop body
O(1) + O(n) = **O(n)**

Outer loop:
n iterations,
O(n) body,
so **O(n²)**

Inner loop
O(n)

Modelling a fast dynamic array

```
int[] array = {0};
for (int i = 1; i <= n; i*=2) {
  int[] newArray =
    new int[array.length*2];
  for (int j = 0; j < i; j++)
    newArray[j] = array[j];
  newArray = array;
}
```

Modelling a fast dynamic array

```
int[] array = {0};
for (int i = 1; i <= n; i*=2) {
  int[] newArray =
    new int[array.length*2];
  for (int j = 0; j < i; j++)
    newArray[j]   array[j];
  newArray =
}
```

Outer loop:
log n iterations,
O(n) body,
so **O(n log n)**??

Modelling a fast dynamic array

```
int[] array = {0};
for (int i = 1; i <= n; i*=2) {
  int[] newArray =
    new int[array.length*2];
  for (int j = 0; j < i; j++)
    newArray[j]   array[j];
  newArray =
}
```

Here we
“round up”
O(i) to O(n).
This causes an
overestimate!

A complication

Our algorithm has O(n) complexity, but we've calculated O(n log n)

- An overestimate, but not a severe one

  (If n = 1000000 then n log n = 20n)
- This can happen but is normally not severe
- To get the right answer: do the maths

Good news: for “normal” loops this doesn't happen

- If all bounds are n, or n^2, or another loop variable, or a loop variable squared, or …

Main exception: loop variable *i* doubles every time, body complexity depends on *i*

Doing the sums

In our example:

- The inner loop's complexity is $O(i)$
- In the outer loop, i ranges over $1, 2, 4, 8, \ldots, 2^a$

Instead of rounding up, we will add up the time for all the iterations of the loop:

$$1 + 2 + 4 + 8 + \ldots + 2^a$$

$$= 2 \times 2^a - 1 < 2 \times 2^a$$

Since $2^a \leq n$, the total time is at most $2n$, which is $O(n)$

A last example

```
for (int i = 1; i <= n; i *= 2) {
    for (int j = 0; j < n*n; j++)
        for (int k = 0; k <= j; k++)
            // O(1)
    for (int j = 0; j < n; j++)
        // O(1)
}
```

A last example

The outer loop runs O(log n) times

The j-loop runs n^2 times

```
for (int i = 1; i <= n; i *= 2) {
    for (int j = 0; j < n*n; j++)
        for (int k = 0; k <= j; k++)
            // O(1)
    for (int j = 0; j < n; j++)
        // O(1)
}
```

`k <= j < n*n` so this loop is $O(n^2)$

This loop is O(n)

Total: $O(\log n) \times (O(n^2) \times O(n^2) + O(n))$
$= O(n^4 \log n)$

A couple of loose ends

Big Ω

Recall that big-O allows us to *overestimate* the growth rate of a function:

- $2n^2+3n+1$ is $O(n^2)$, but also $O(n^3)$

Big-O has a cousin, big-Ω ("big-omega"), which allows us to *underestimate* the growth rate:

- $2n^2+3n+1$ is $\Omega(n^2)$, but also $\Omega(n)$

Formally we just replace a $\leq$ with a $\geq$ in the definition of big-O:

- $T(n)$ is $O(n^2)$ if $T(n) \leq kn^2$ for some k, for big enough n
- $T(n)$ is $\Omega(n^2)$ if $T(n) \geq kn^2$ for some k, for big enough n

Big Θ

There is also big-Θ ("big-theta"), which is like big-O but requires the complexity given to be tight:

- For example, $2n^2+3n+1$ is $\Theta(n^2)$ (and nothing else)
- $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$

You should recognise all three notations, but we will mostly stick to big-O in this course

- The other two are generally harder to calculate accurately
- Big-Ω is mostly useful for defining big-Θ
- Big-O gives you an upper bound, which can tell you that an algorithm is fast enough

Amortised time complexity

How long does it take to add one element to a dynamic array?

- Simple answer: O(n)
- But adding n elements to an empty array takes O(n) time, O(1) "per element". So it's somehow O(1) "on average"?
- If we measure the runtime of a program using dynamic arrays, it will look as if each operation took O(1) time!

To capture this, we say that adding an element to a dynamic array has O(1) *amortised complexity*

- An operation has O(f(n)) amortised complexity if, for any sequence of operations, the *total runtime* is as if each operation took O(f(n)) time
- e.g.: O(log n) amortised complexity → n operations take O(n log n) time
- Amortised complexity can occur when an expensive operation is always balanced out by many cheap ones

Be careful to distinguish amortised from "normal" complexity

- If your program has real-time constraints, then a data structure with amortised complexity may be totally unsuitable
- But for most applications, it works just fine

The uniform cost model

We assumed that all primitive operations took constant time – this is called the *uniform cost model*

But – if your programming language supports integers of unbounded size – then arithmetic on bigger numbers takes longer!

- Most arithmetic operations grow as O(log n), where n is the magnitude of the number
- This is called the *logarithmic cost model*
- It is common when integers can be unbounded size, and also in some specialised applications like cryptography

Life without
big O notation

What happens without big O?

How many steps does this function take on an array of length *n* (in the worst case)?

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < a.length; j++)
      if (a[i].equals(a[j]) && i != j)
        return false;
  return true;
}
```

Assume that loop body takes 1 step

What happens without big O?

How many steps does this func n take on an array of length n (i the w e)?

```
boolean unique(O
  for(int i                    ++)
    for (int              n;  j++)
      if (a[i           &  i != j)
        ret
  return true;
}
```

Outer loop runs n times
Each time, inner loop runs n times

Total: $n \times n = n^2$

What about this one?

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < i; j++)
      if (a[i].equals(a[j]))
        return false;
  return true;
}
```

Loop runs to *i* instead of *n*

Some hard sums

When i = 0, inner loop runs 0 times

When i = 1, inner loop runs 1 time

...

When i = n-1, inner loop runs n-1 times

Total:

- $\sum_{i=0}^{n-1} i = 0 + 1 + 2 + \ldots + n\text{-}1$

which is $n(n\text{-}1)/2$

What about this one?

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.lengt  ; i++)
    for (int j = 0;
      if (a[i].equa
        return fal
  return true;
}
```

Answer:
$n(n\text{-}1)/2$

What about this one?

```
void something(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < i; j++)
      for (int k = 0; k < j; k++)
        “something that takes 1 step”
}
```

More hard sums

$$\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} \sum_{k=0}^{j-1} 1$$

Outer loop:
i goes from 0 to *n-1*

Middle loop:
j goes from 0 to i-*1*

Inner loop:
k goes from 0 to j-*1*

Counts: how many values i, j, k where
$0 \le i < n$, $0 \le j < i$, $0 \le k \le j$

More hard sums

$$\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} \sum_{k=}^{j-1} 1$$

Answer (I looked it up):
$n(n\text{-}1)(n\text{-}2)/6$

Counts: how many values i, j, k where
$0 \leq i < n$, $0 \leq j < i$, $0 \leq k \leq j$

What about this one?

```
void something(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j <         ++)
      for (int k =               )
        “something                step”
}
```

Answer:
$n(n\text{-}1)(n\text{-}2)/6$

Sums vs integrals

$$\sum_{x=a}^{b} f(x) \approx \int_{a}^{b} f(x)$$

For example:

$$\sum_{i=0}^{n} i = n(n+1)/2 \qquad \int_{0}^{n} x\,dx = n^2/2$$

Not quite the same, but close! (usually gives the right complexity)

A better approach: *"Finite calculus: a tutorial for solving nasty sums"* - adapts rules of calculus to work with sums instead of integrals

Big O in retrospect

We do lose some precision by throwing away constant factors

- ...you probably *do* care about a factor of 100 performance improvement
- ...but in practice the constant factors don't get much higher than 2,

On the other hand, life gets much simpler:

- A small phrase like $O(n^2)$ tells you exactly how the performance *scales* when the input gets big
- It's a lot easier to calculate big-O complexity than a precise formula (lots of good rules to help you)

Big O is normally an excellent compromise!