# Complexity

*(Weiss chapter 5)*

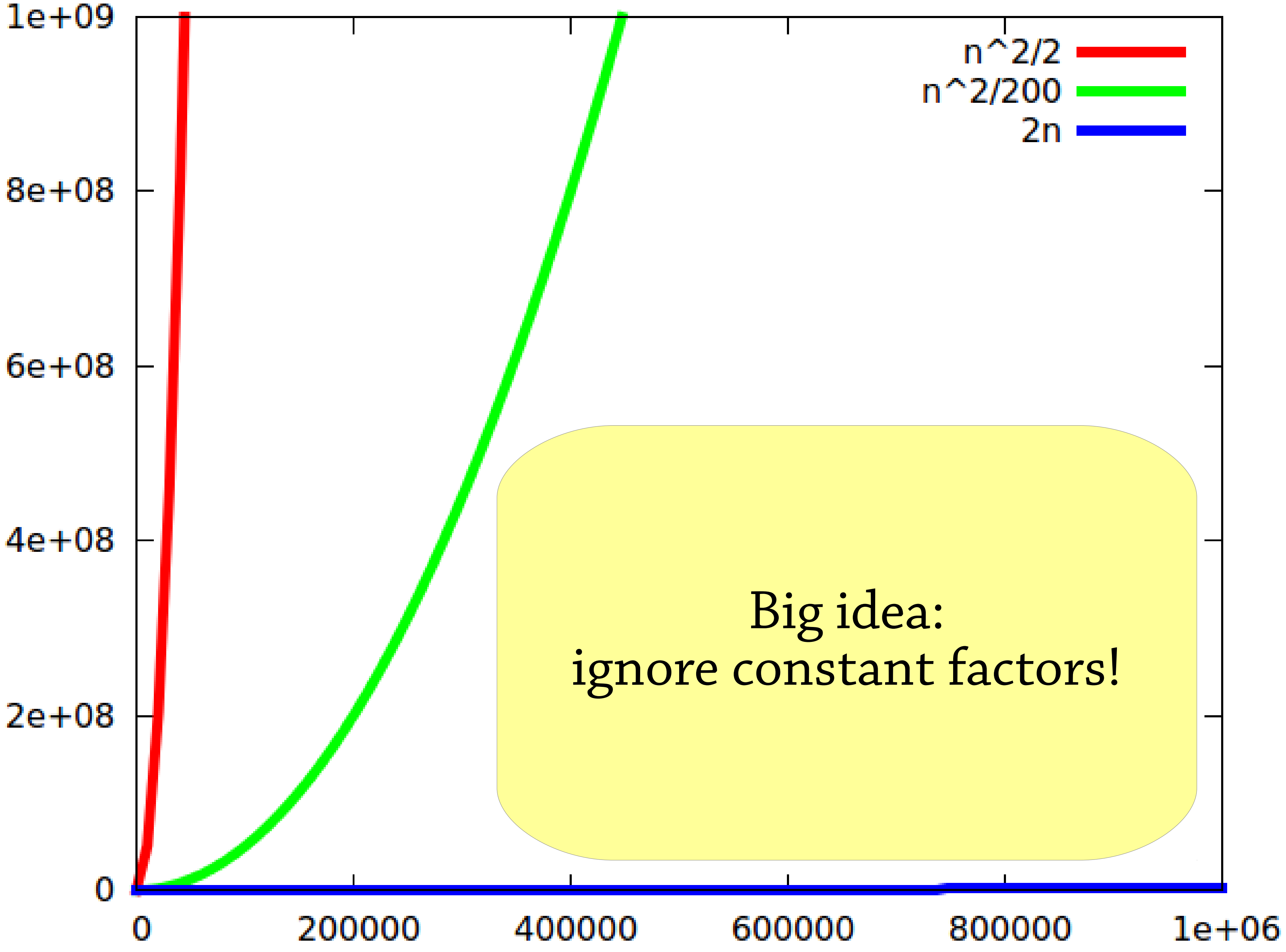
# Complexity

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

Last time we had three versions of the file-reading program. For a file of size *n*:

- The first one needed to copy $n^2/2$ characters
- The second one needed to copy $n^2/200$ characters
- The third needed to copy 2n characters

We worked out these formulas, but it was a bit of work – now we'll see an easier way



Big idea:
ignore constant factors!

# Why do we ignore constant factors?

Well, when n is 1000000…

- $\log_2 n$ is 20
- n is 1000000
- $n^2$ is 1000000000000
- $2^n$ is a number with 300,000 digits…

Given two algorithms:

- The first takes 1000000 $\log_2 n$ steps to run
- The second takes 0.00000001 × $2^n$

The first is miles better!
Constant factors *normally* don't matter

# Big O notation

Instead of saying…

- The first implementation copies $n^2/2$ characters
- The second copies $n^2/200$ characters
- The third copies $2n$ characters

We will just say…

- The first implementation copies **$O(n^2)$** characters
- The second copies **$O(n^2)$** characters
- The third copies **$O(n)$** characters

**$O(n^2)$ means “proportional to $n^2$” (almost)**

# Time complexity

Suppose an algorithm takes $n^2/2$ steps, and each step takes 100ns to run

- The total time taken is $50n^2$ ns
- This is $O(n^2)$
- The number of steps taken is also $O(n^2)$

It doesn't matter whether we count steps or time!

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

# Why ignore constant factors?

Big O really simplifies things:

- A small phrase like O($n^2$) tells you a lot
- It's easier to calculate than a precise formula
- We get the same answer whether we count *number of statements executed* or *time taken* (or in this case *number of elements copied*) – so we can be a bit careless what we count

On the other hand:

- Sometimes we do care about constant factors!

Big O is normally a good compromise

# What happens without big O?

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

```
Object search(Object[] a, Object x) {
  for(int i = 0; i < a.length; i++) {
    if (a[i].equals(target))
      return a[i];
  }
  return null;
}
```

Assume that loop body takes 1 step

# What happens without big O?

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

```
Object search(Object[] a      ject x) {
  for(int i = 0; i <            ) {
    if (a[i].equal
      return a[i
  }
  return null;
}
```

Answer:
$n$

# 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;  <            ++)
    for (int j =              +)
      if (a[i             != j)
        ret
  return tru
}
```

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:

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

which is $n(n-1)/2$

# What about this one?

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

Answer:
$n(n-1)/2$

# What about this one?

```
boolean unique(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

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

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

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

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

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

I have no idea
how to solve this!
Wolfram Alpha says it's
$n(n-1)(n-2)/6$

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

# What about this one?

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

Answer:
$n(n-1)(n-2)/6$,
apparently

This is just horrible!
Isn't there a better way?

# Using big O complexity

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < i; j++)
      for (int k = 0;   < j; k++)
        “something tha      s 1 step”
}
```

Three nested loops,
all running from 0 to n…
Answer: **$O(n^3)$**!

# Why ignore constant factors? (again)

Big O really simplifies things:

- A small phrase like $O(n^2)$ tells you a lot
- It's easier to calculate than a precise formula
- We get the same answer whether we count *number of statements executed* or *time taken* (or in this case *number of elements copied*) – so we can be a bit careless what we count

On the other hand:

- Sometimes we do care about constant factors!

Big O is normally a good compromise

# re constant factors? (again)

Our long calculation only told us how many steps the algorithm takes, not how much time!

Isn't it!

mplifies things:

- A sma ase like O($n^2$) tells you a t
- It's eas r to calculate than a precise formula
- We get the same answer *number of statements exec* in this case *number of eleme* be a bit careless what we

But normally not enough to go to all this trouble!

## On the other hand:

- Sometimes we do care about constant factors!

## Big O is normally a good compromise

# The rest of the lecture

How to calculate big-O complexity:

- We will first have to define formally what it means for an algorithm to have a certain complexity
- We will then come up with some rules for calculating complexity
- To come up with those rules, we will have to do “hard sums”, but once we have the rules we can forget the sums
- (very occasionally, you might still have to do the sums yourself)

# 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

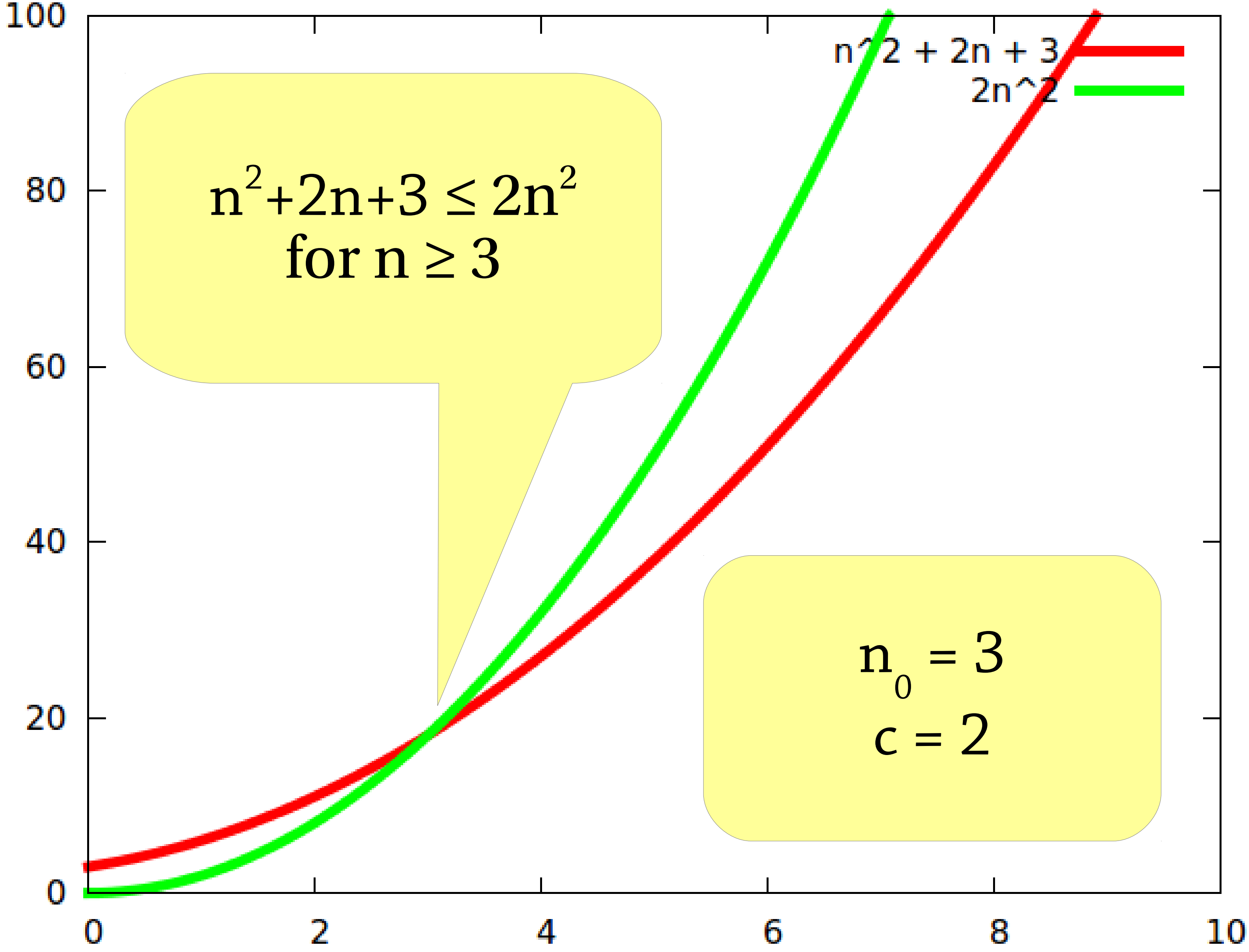
Formally, we say "T($n$) is O(f($n$))"

- E.g., "T(n) is O($n^2$)"

This means:

- T($n$) $\leq a \times$ f($n$), for some constant $a$ (i.e., T($n$) is proportional to f($n$) **or smaller**)
- **But** this need only hold for all $n$ above some threshold $n_0$

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



$n^2+2n+3 \leq 2n^2$
for $n \geq 3$

$n_0 = 3$
$c = 2$

# Exercises

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

| **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 |
| $O(n!)$ | Factorial |

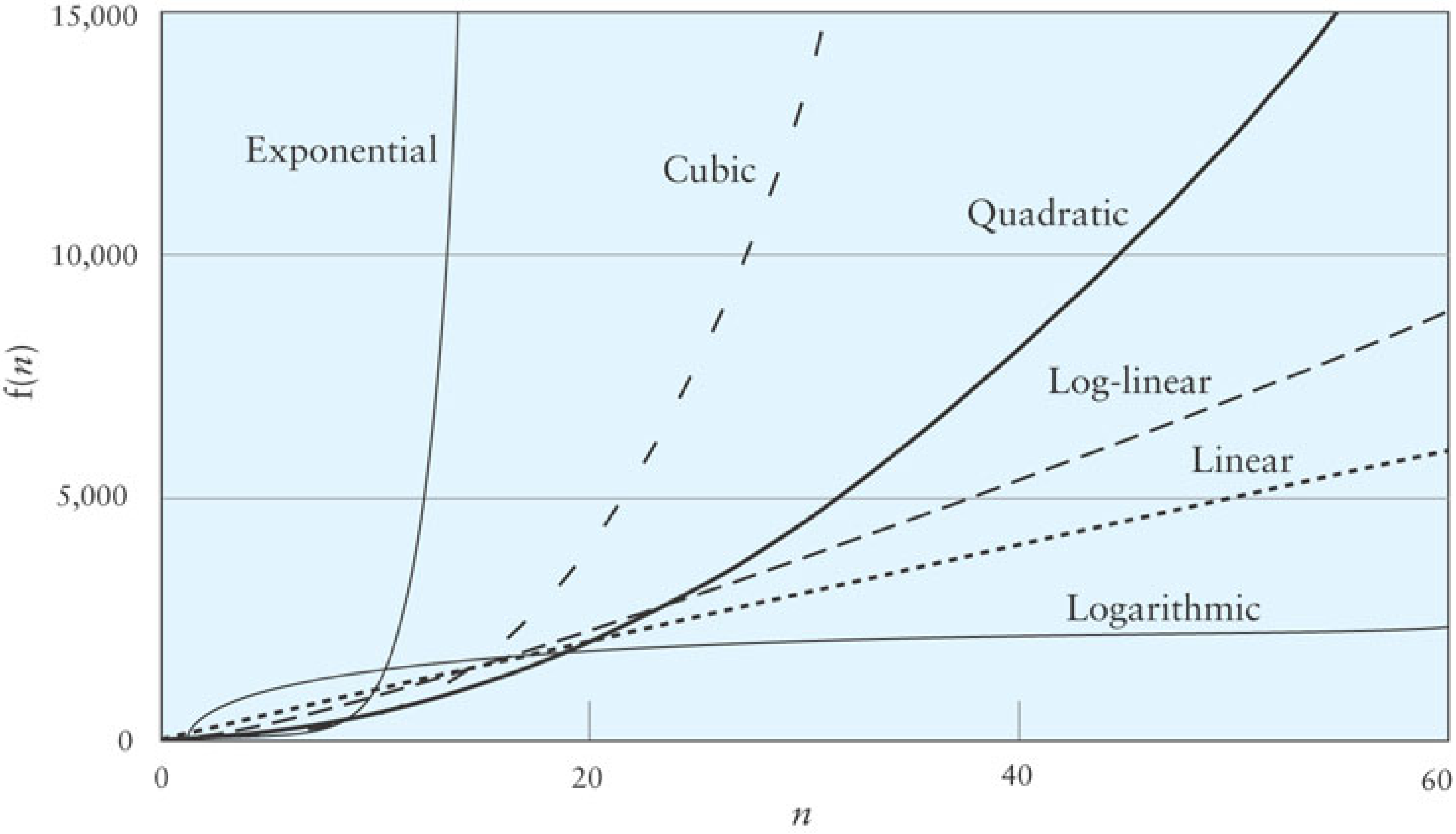
# Growth rates

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

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


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

# Adding big O (a hierarchy)

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

When adding a term lower in the hierarchy to one higher in the hierarchy, the lower-complexity term disappears:

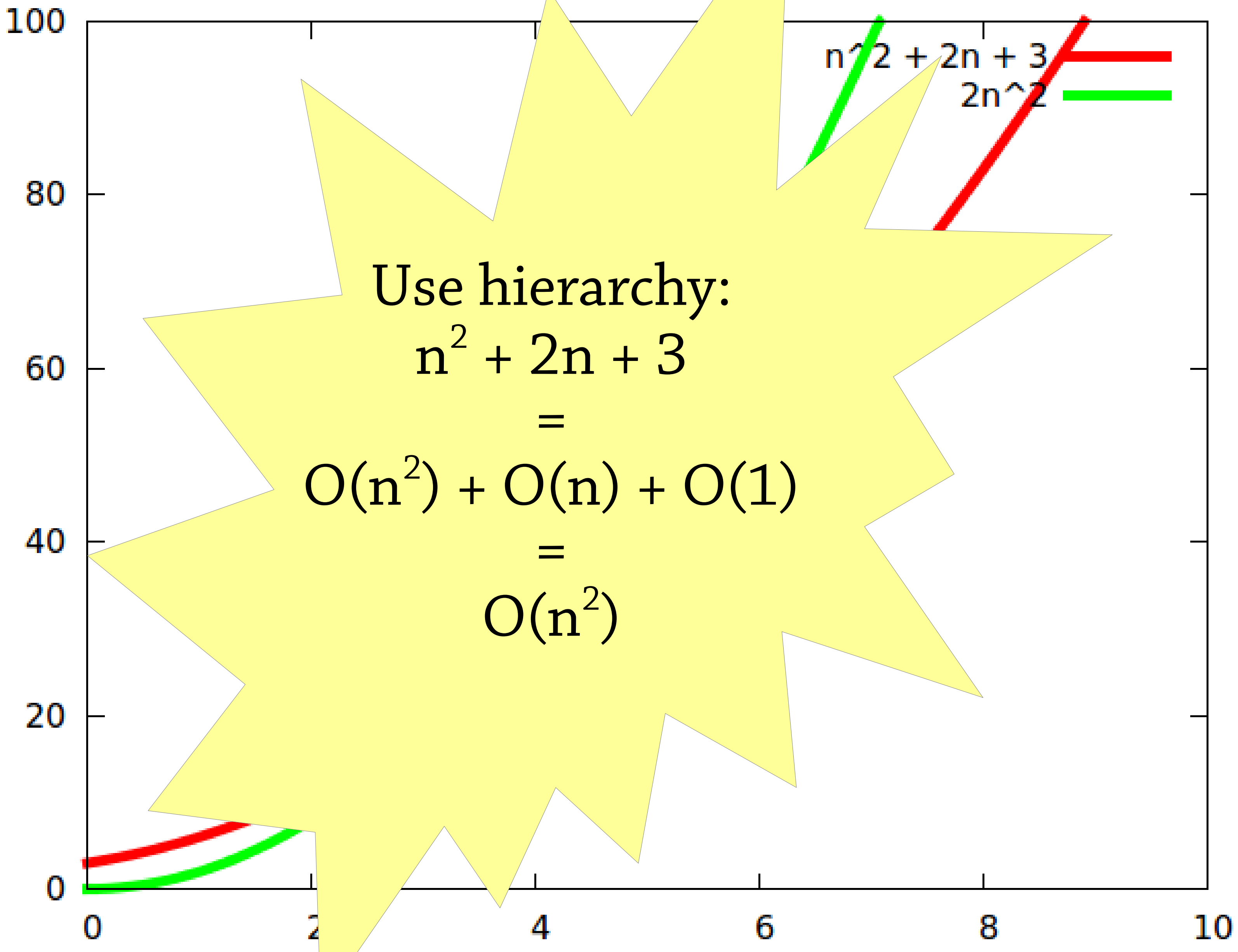
$O(1) + O(\log n) = O(\log n)$

$O(\log n) + O(n^k) = O(n^k)$ (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?

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

# Just use hierarchy!

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

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

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