Complexity

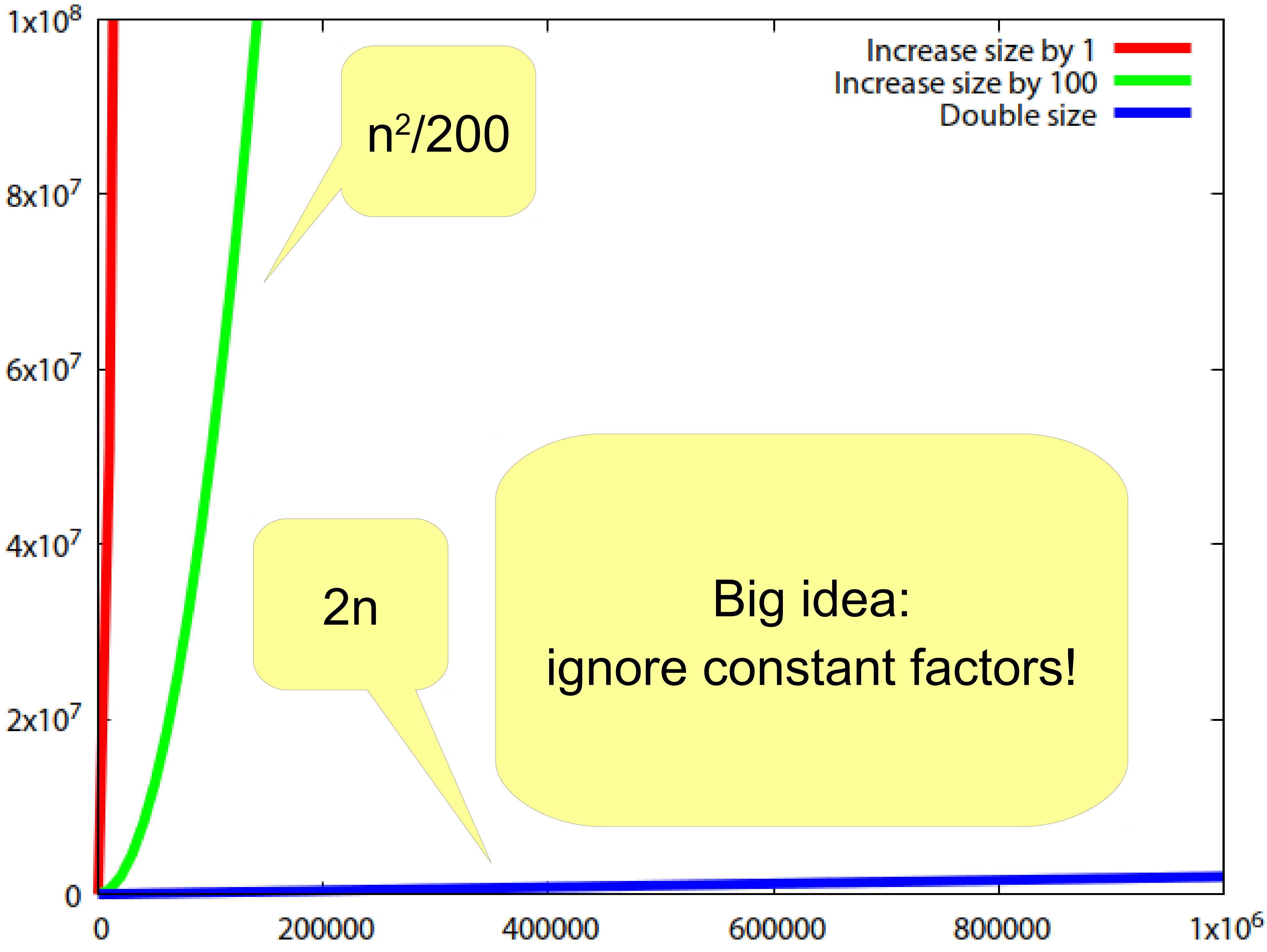
Complexity

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



Increase size by 1
Increase size by 100
Double size
n²/200
2n
Big idea:
ignore constant factors!
1x10⁸
8x10⁷
6x10⁷
4x10⁷
2x10⁷
0
0
200000
400000
600000
800000
1x10⁶

Big O (sv: Ordo) 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

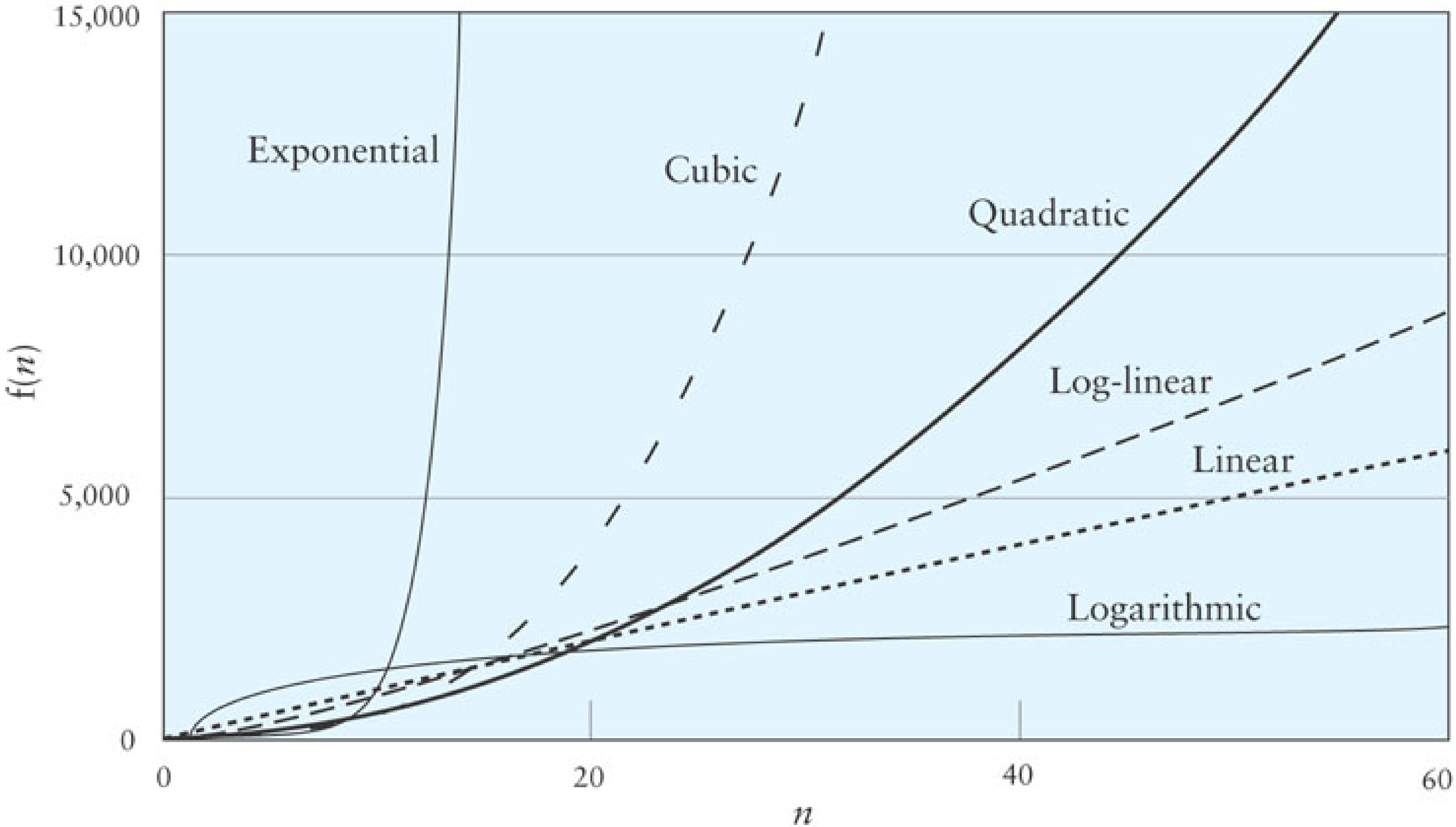
With big-O notation, it doesn't matter whether we count steps or time!

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*

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

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

Big O tells you *how the performance of an algorithm is affected by the input size*

A sneak peek

```
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;
}
```

Outer loop runs O(n) times

Inner loop runs O(n) times for each outer loop

$O(n) \times O(n) = \mathbf{O(n^2)}$

The mathematics of big O

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:

- $T(n) = n^2/2$
- $T(n)$ is $O(n^2)$

Big O, formally

What does it mean to say "$T(n)$ is $O(n^2)$"?

We could say it means $T(n)$ is proportional to n^2

- i.e. $T(n) = kn^2$ 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 $O(n^2)$ if:

- $T(n) \le kn^2$ for some k,
  i.e. T(n) is proportional to n^2 *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 kn^2, at some point the graph of kn^2 must permanently overtake the graph of T(n)

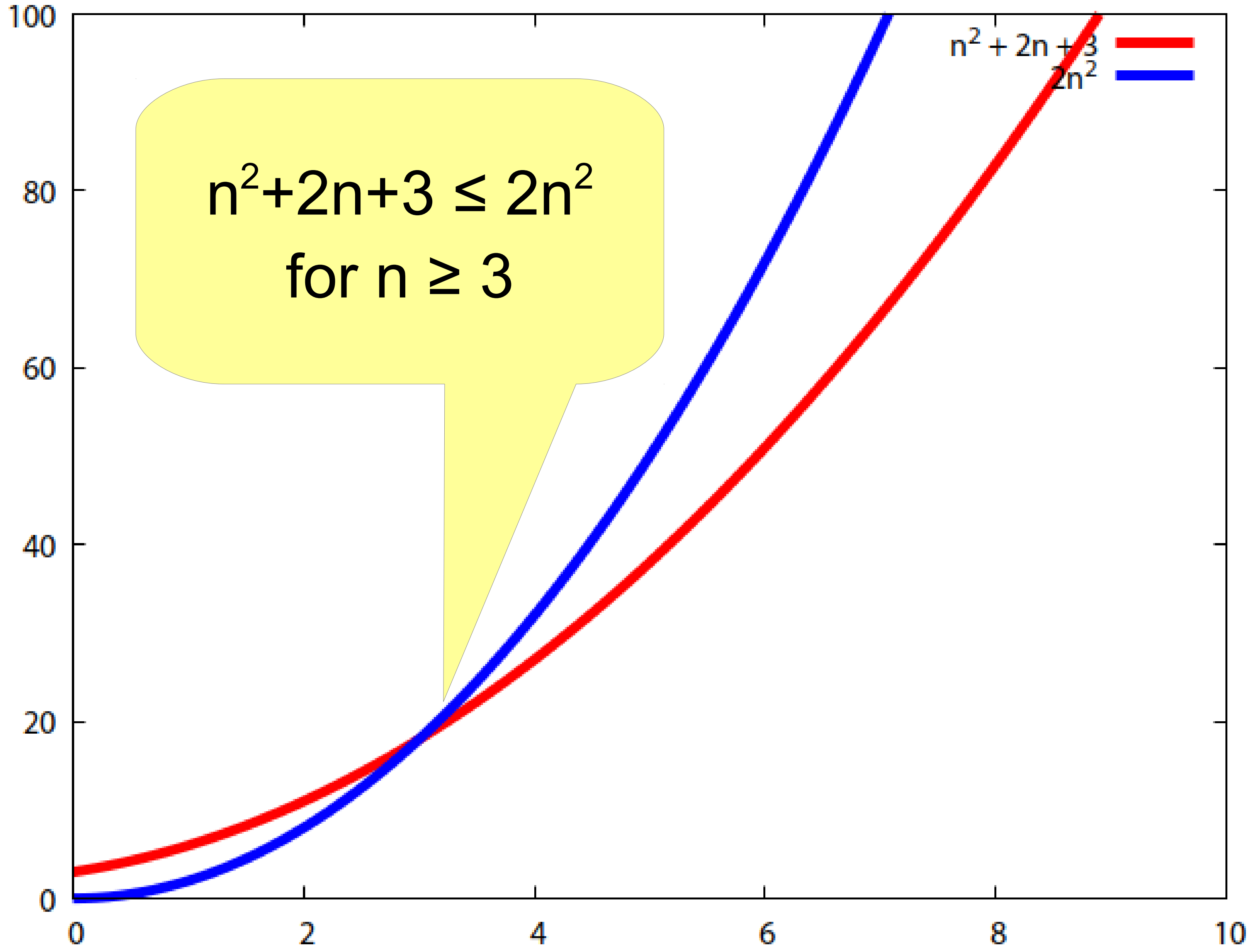
- In other words, T(n) grows more slowly than kn^2

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

Compact definition of Big O:

$T(n) \in O(f(n))$ when $\exists k,n_0 : T(n) \le kf(n)$ for $n \ge n_0$

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



More examples

- Is $3n + 5$ in $O(n)$?
- Is $n^2 + 2n + 3$ in $O(n^3)$?
- Is it in $O(n)$?
- Is 5 in $O(1)$?

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:

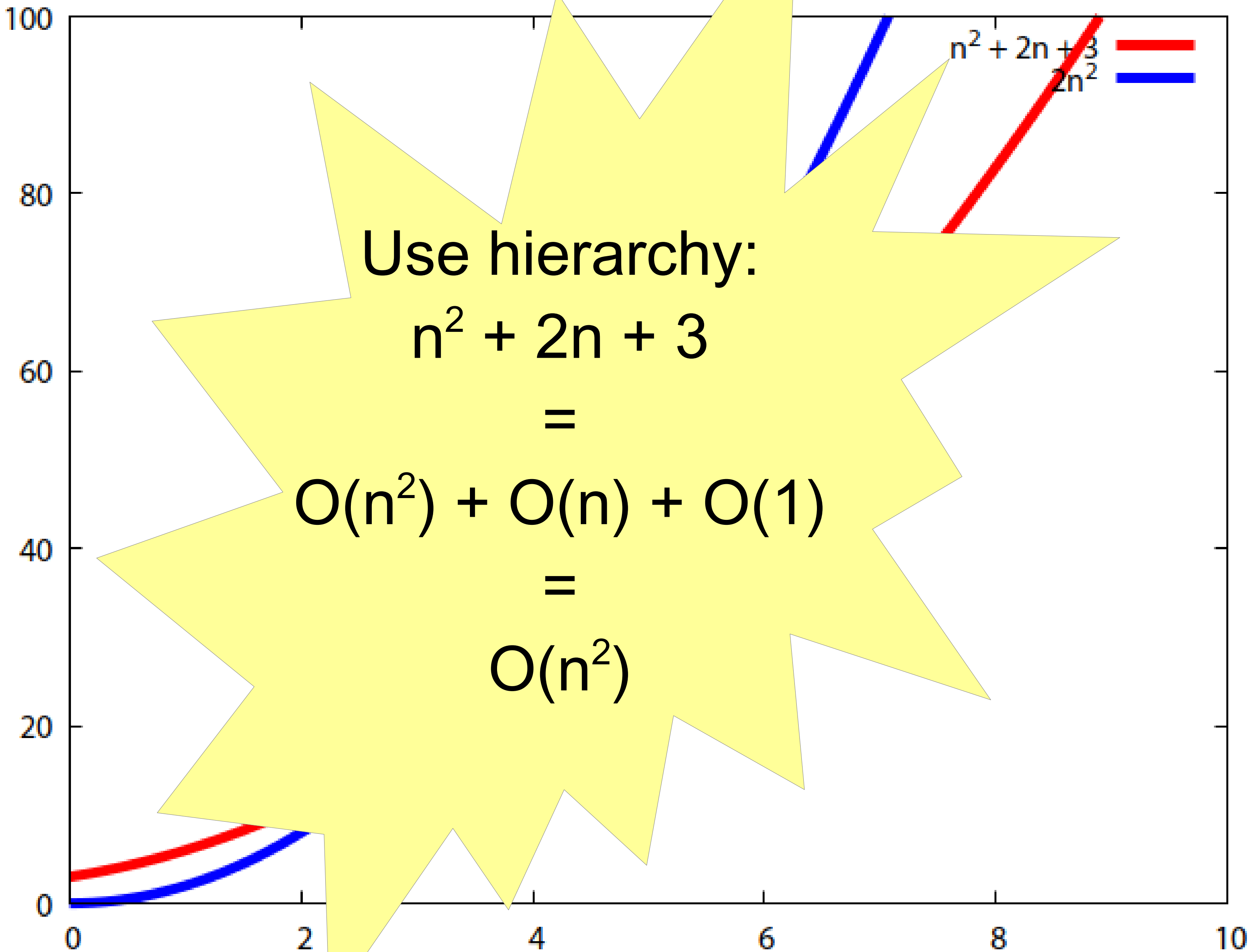
$O(1) + O(\log n) = O(\log n)$

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

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

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

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

Multiplying big O

$O(\text{this}) \times O(\text{that}) = O(\text{this} \times \text{that})$

- 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

Big O and related concepts

f(n) is asymptotically an upper bound of the growth rate of T(n):

$T(n) \in O(f(n))$ when $\exists k, n_0 : T(n) \leq kf(n)$ for $n \geq n_0$

f(n) is asymptotically a lower bound of the growth rate of T(n):

$T(n) \in \Omega(f(n))$ when $\exists k, n_0 : T(n) \geq kf(n)$ for $n \geq n_0$

f(n) is asymptotically a lower and upper bound of the growth rate of T(n):

$T(n) \in \Theta(f(n))$ when $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$

Reasoning about programs

Cost Models

We need to simplify how computers work.

- Uniform model:
  - Unbounded numbers (not limited to e.g. 64 bits)
  - Infinite memory
- Logarithmic model:
  - Data size is measured in number of bits
  - Infinite memory

In most cases we’ll use the uniform model.

Complexity of a program

Most "primitive" operations take constant time:

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

O(1)

Complexity of loops

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

Complexity of a program

What about loops?

(Assume the arrays size is *n*)

```
void add(double[] a, double[] b) {
  for (int i = 0; i < a.length; i++)
    a[i] += b[i];
}
```

Complexity of a program

What about loops?

(Assume the arrays size is *n*)

```
void add(double[] a, double[] b) {
  for (int i = 0; i < a.length; i++)
    a[i] += b[i];
}
```

Loop runs
O(n) times

Loop body takes
O(1) time

O(1) × O(n) = **O(n)**

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;
}
```

Worst case complexity

- Often not only the size of the data influences the running time, but also the values.
- The longest possible running time for a given data size is called the worst case complexity (sv: värsta falls-komplexiteten)
- You can also compute the best case complexity, but it’s not as useful since what you want in most cases is a guarantee that running a program will not take more than a certin time.

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 in worst case

Loop body takes O(1) time

Worst case complexity: O(1) × O(n) = **O(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; i < a.length; i++)
    for (in   = 0; j < a.length; j++)
      if                  && i != j
        r
  retur
}
```

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 on

void function(int n) {

for(int i = 0; i < **n*n**; i++)

for (in j = 0; j < **n/2**; j++)

“s O(1) tim ”

}

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 (in   = 0; j < i; j++)
      if (
        r
  retur
}
```

Inner loop is
$i \times O(1) = O(i)$??
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++)
      if (
        r
  retur
}
```

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

if (

r

retur

}

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)

The example from earlier

```
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++)
        “son
}
```

$i < n$, $j < n$, $k < n$,

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
}
```

What's the complexity?

Outer loop is O(n log n)

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+1;
    while (i + 1 != j) {
        long k = (i + j) / 2;
        if (k*k <= n) i = k;
        else j = k;
    }
    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+1;
    while (i + 1 != j) {
        long k = (i + j) / 2;
        if (k*k <= n) i = k;
        else j = k;
    }
    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: same rule, but can be trickier to count 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
- 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!

A familiar scene

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

Assume that each statement takes O(1) time

A familiar scene

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

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

Outer loop: n iterations, O(n) body, so **O(n^2)**

Inner loop **O(n)**

A familiar scene, take 2

```
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;
}
```

A familiar scene, take 2

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

Outer loop:
n/100 iterations,
which is O(n)
O(n) body,
so **O(n^2)** still

A familiar scene, take 3

```
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;
}
```

A familiar scene, take 3

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

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

A familiar scene, take 3

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

This loop is O(n)

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

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

Nested loops with dependent iteration intervals

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 f ncti ake on an array of length *n* (in wo

```
boolean unique(Objec
  for(int i = 0; i < a
    for (int j = 0; j
      if (a[i].equals(
        return false
  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 + … + n-1

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

What about this one?

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

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

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

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

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

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

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?

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

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

Amortized analysis

- A single append-operation for a dynamic array:

```
public void append(char c) {
        if (size == string.length) {
            // Create a new array, twice as big as before.
            char[] newString = new char[string.length*2];
            for (int i = 0; i < string.length; i++)
                newString[i] = string[i];
            string = newString;
        }
        string[size] = c;
        size++;
}
```

Time complexity: O(n) in worst case, which is pessimistic.

Amortized analysis

- Amortized analysis measures how much time each operation will take in a sequence of operations.
- For the append method of a dynamic array the amortized complexity is O(1)
- There are different methods for amortized analysis
- One is the potential method where you “pay” in advance for future high-cost executions in such a way that you never run out of saved “coins”.

Big O in retrospect

We lose some precision by throwing away constant factors

- ...you probably *do* care about a factor of 100 performance improvement

On the other hand, life gets much simpler:

- A small phrase like $O(n^2)$ tells you a lot about 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 a good compromise!

- Occasionally, need to do hard sums anyway :(

Complexity of recursive functions

Recurrence equations

- The general way to calculate complexity for a recursive function is to write a set of recurrence equations.
- E.g.:

```
fcn f(n) {
  if (n == 0) return x;
  somecode1
  f(n-1)
  somecode2
}
```

- If `somecode1` + `somecode2` has complexity O(n) the recurrence equations for this function's complexity is (we drop the O(..)):

$$T(0) = 1$$
$$T(n) = n + T(n-1) \quad \text{when } n > 0$$

Solving reccurrence equations

- There isn't a general way of solving *any* recurrence relation – we'll just see a few families of them.
- In general you have to guess a solution function (possible parameterized).
- You can then by induction confirm that the function is correct.

Example: $T(n) = O(n) + T(n-1)$

$T(n) = n + T(n-1)$

$= n + (n-1) + T(n-2)$

$= n + (n-1) + (n-2) + T(n-3)$

$= \ldots$

$= n + (n-1) + (n-2) + \ldots + 1 + T(0)$

$= n(n+1) / 2 + T(0)$

$= O(n^2)$

Example: $T(n) = O(1) + T(n-1)$

$T(n) = 1 + T(n-1)$

$= 2 + T(n-2)$

$= 3 + T(n-3)$

$= \ldots$

$= n + T(0)$

$= O(n)$

Example: $T(n) = O(1) + T(n/2)$

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

$= 2 + T(n/4)$

$= 3 + T(n/8)$

$= \ldots$

$= \log n + T(1)$

$= O(\log n)$

Another example: $T(n) = O(n) + T(n/2)$

$T(n) = n + T(n/2)$:

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

$= n + n/2 + T(n/4)$

$= n + n/2 + n/4 + T(n/8)$

$= \ldots$

$= n + n/2 + n/4 + \ldots$

$< 2n$

$= O(n)$

Functions that recurse once

$T(n) = O(1) + T(n-1)$: $T(n) = O(n)$

$T(n) = O(n) + T(n-1)$: $T(n) = O(n^2)$

$T(n) = O(1) + T(n/2)$: $T(n) = O(\log n)$

$T(n) = O(n) + T(n/2)$: $T(n) = O(n)$

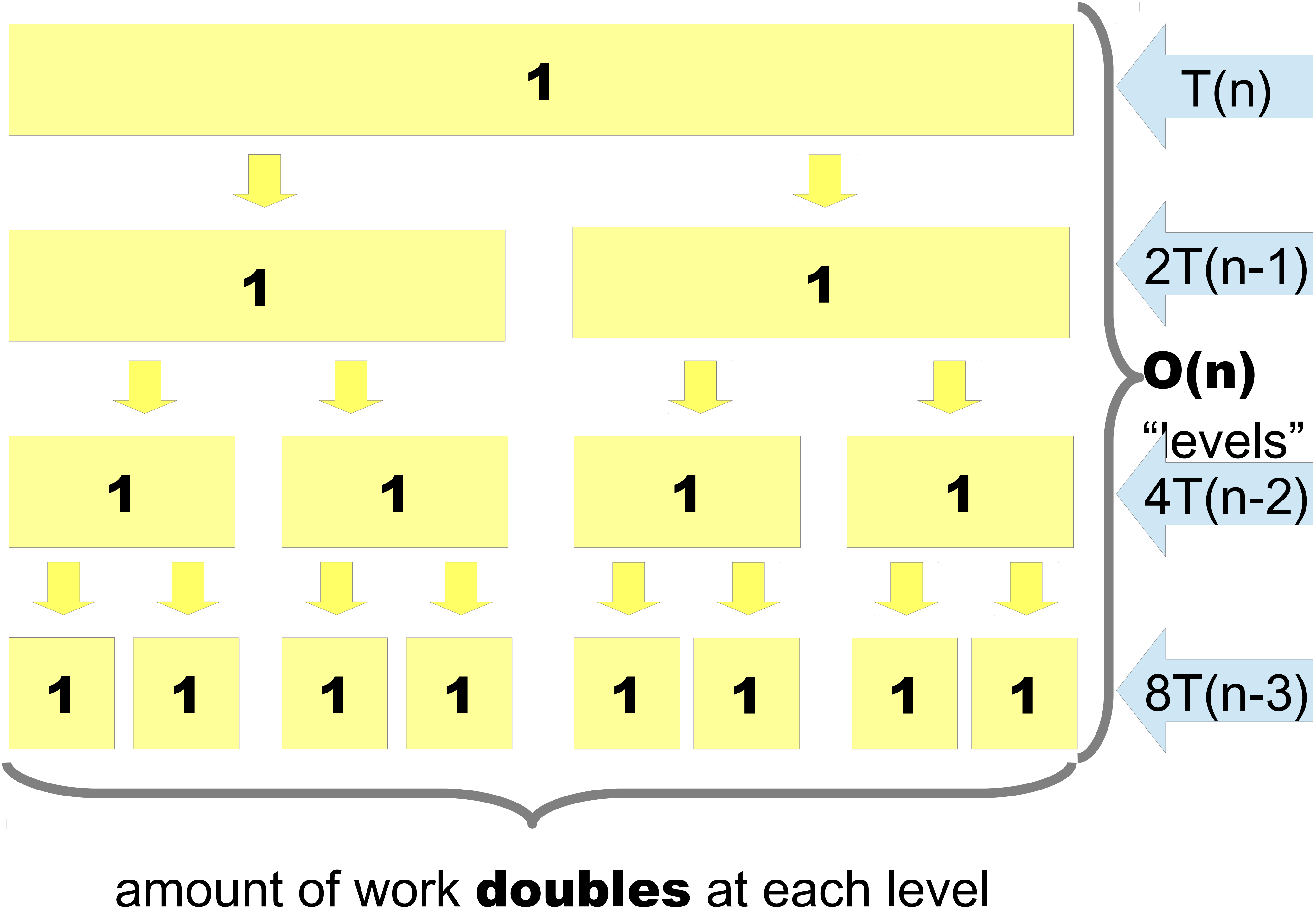
An *almost-rule-of-thumb*:

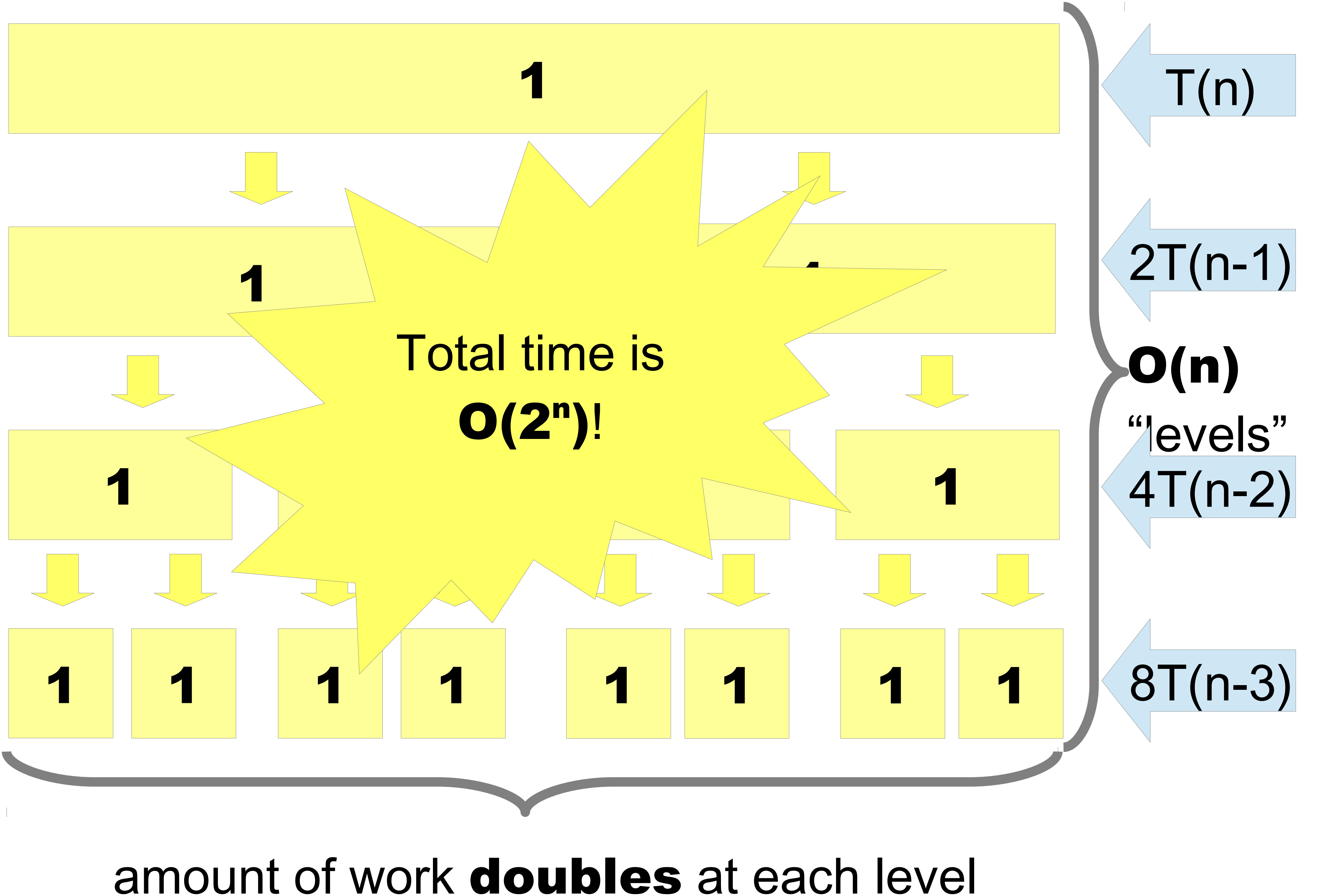
- Solution is *maximum recursion depth* times *amount of work in one call*

(except that this rule of thumb would give $O(n \log n)$ for the last case)

Example of function that does two recursive calls:

$$T(n) = O(1) + 2T(n-1)$$


1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
T(n)
2T(n-1)
4T(n-2)
8T(n-3)
O(n)
“levels”
amount of work **doubles** at each level


1
1
1
1
1
1
1
1
1
1
1
1
1
Total time is
O(2ⁿ)!
T(n)
2T(n-1)
4T(n-2)
8T(n-3)
O(n)
"levels"
amount of work **doubles** at each level

Complexity of recursive functions

Basic idea – recurrence relations

Easy enough to write down, hard to solve

- One technique: expand out the recurrence and see what happens
- Another rule of thumb: multiply work done per level with number of levels
- Drawing a diagram might help