CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Data structures

Complexity

Dr. Alex Gerdes

DIT961 - VT 2018

Summary previous lecture

- Course introduction
- Small example: dynamic arrays

- Aritmetisk summa!
- Google-group: do it now!
- Resources on course website
- Labpartner: after lecture or via Google-group
- `Arrays.copyOf(…)`
- Measuring time

Introduction

- 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 $n(n+1)/2$ characters
  - The second one needed to copy $n(n+1)/200$ 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

Big idea:
ignore constant
factors!

Why do we ignore constant factors?

- Well, when n is 1,000,000...
  - $\log_2 n \approx 20$
  - n is 1,000,000
  - n^2 is 1,000,000,000,000
  - 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 (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 $\mathbf{O}(\boldsymbol{n}^2)$ characters
  - The second copies $\mathbf{O}(\boldsymbol{n}^2)$ characters
  - The third copies $\mathbf{O}(\boldsymbol{n})$ 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 n*umber 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)?

Answer: n

```
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 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;
}
```

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

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:

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

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

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

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 $0 \le i < n$
$0 \le j < i$
$0 \le k < j$

I have no idea how to solve this! Wolfram Alpha says it's

$n(n-1)(n-2)/6$

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”
}
```

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; k < j; k++)
        "something that takes 1 step"
}
```

Three nested loops, all running from 0 to *n*...

Answer: $\mathbf{O}(n^3)$!

Why ignore constant factors? (again)

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

plifies things:

like O(n^2) tells you a lot

Isn't it!

- It's eas er to calculate than a precise formula
- We get the same answer whether we count *number of statements executed* or *time taken* (ber *of elements copied*) – so we can be we count

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

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

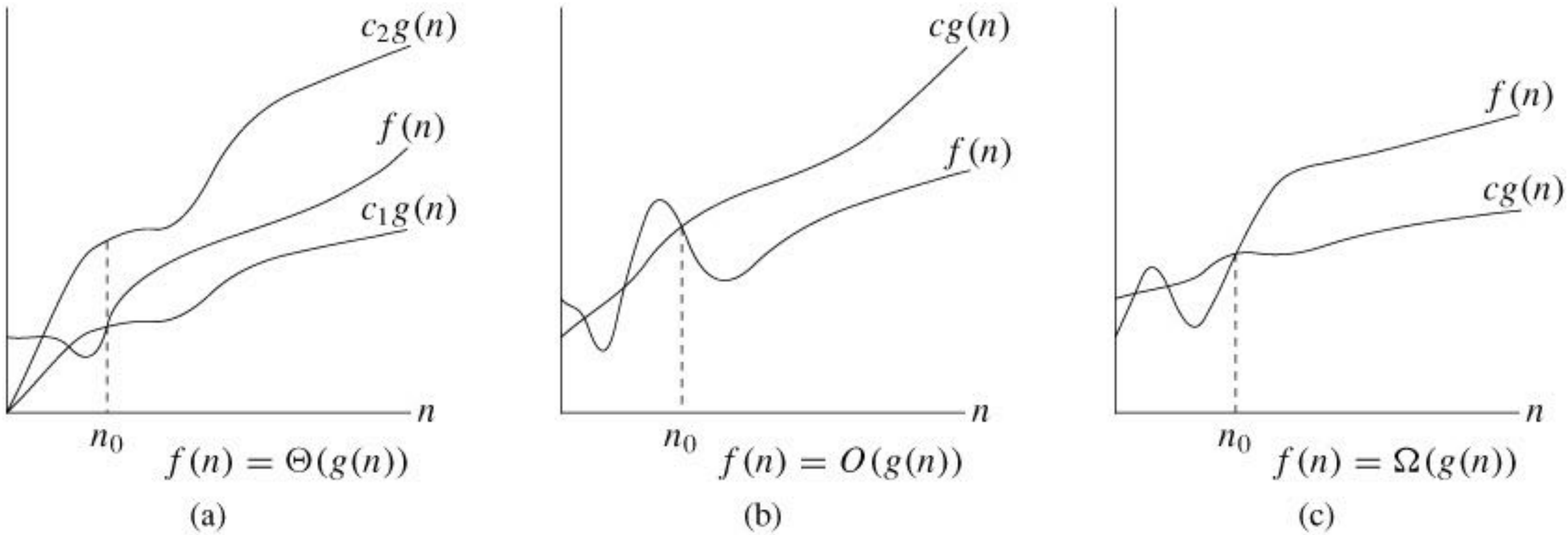
Big O and related concepts

- $T(n) = O(f(n))$ means $a \times f(n)$ is an *upper bound* on $T(n)$. Thus there exists some constant a such that $T(n)$ is always $\leq a \times f(n)$, for large enough n (i.e. , $n \geq n_0$ for some constant n_0).

- $T(n) = \Omega(f(n))$ means $a \times f(n)$ is a *lower bound* on $T(n)$. Thus there exists some constant a such that $T(n)$ is always $\geq a \times f(n)$, for all $n \geq n_0$.

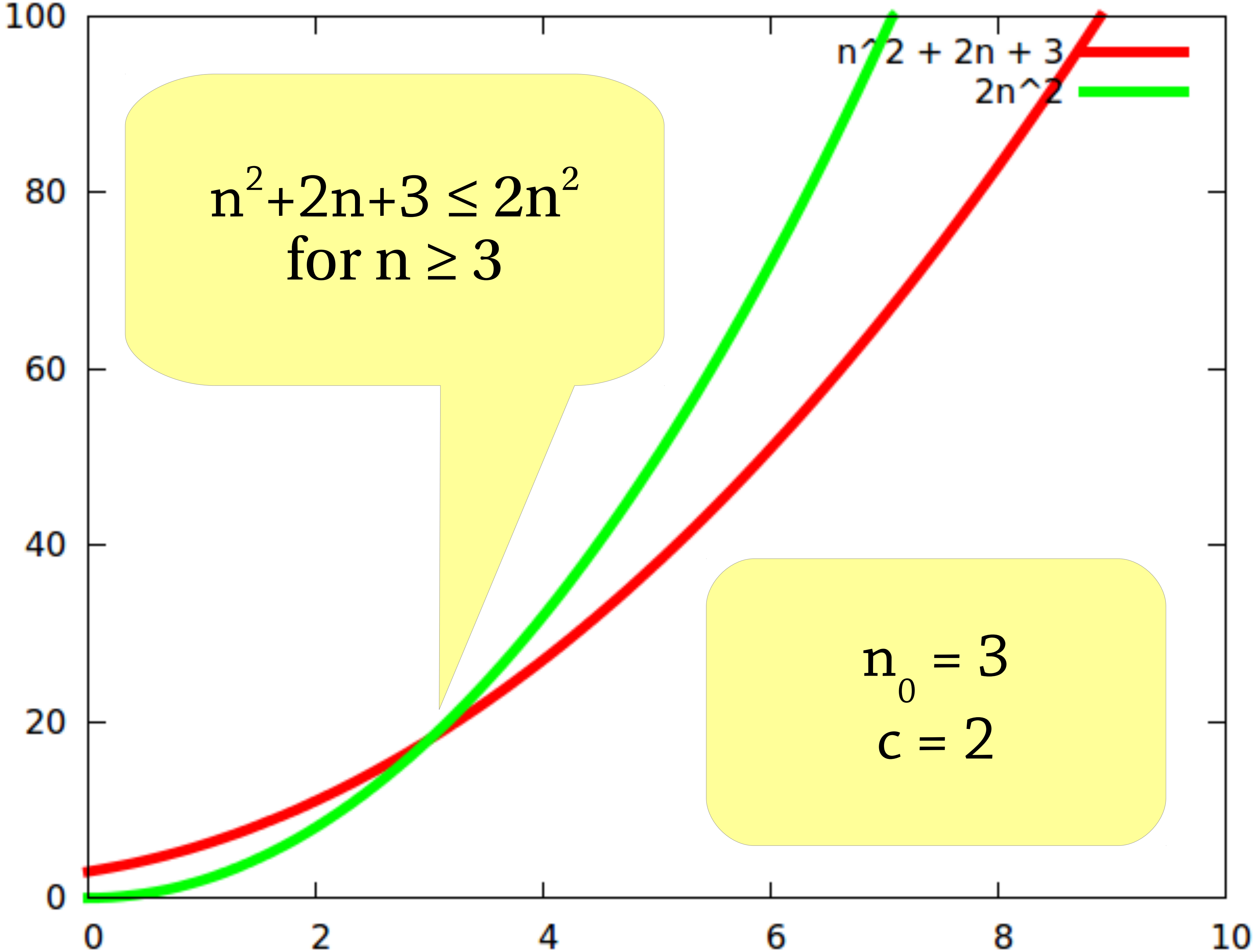
- $T(n) = \Theta(f(n))$ means $a \times f(n)$ is an upper bound on $T(n)$ and $b \times f(n)$ is a lower bound on $T(n)$, for all $n \geq n_0$. Thus there exist constants a and b such that $T(n) \leq a \times f(n)$ and $T(n) \geq b \times f(n)$. This means that $f(n)$ provides a nice, tight bound on $T(n)$.

Big O and related concepts



Source: "The Algorithm Design Manual" by S. Skiena

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



Exercises

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

Dominance classes

Big O	Class
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Linearithmic
$O(n^2)$	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

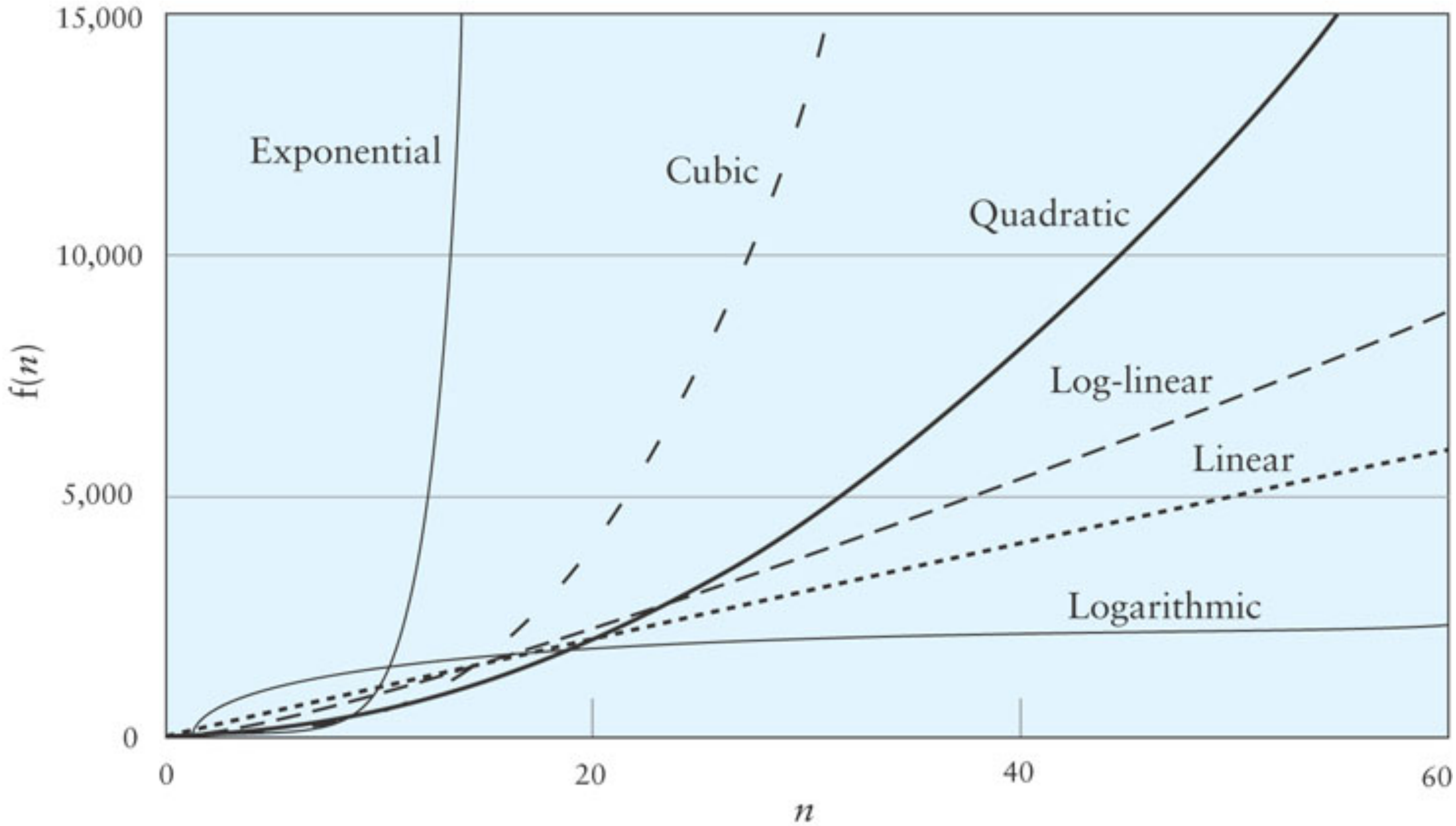
If an algorithm is $O(2^n)$, then adding one element makes it take **twice as long!**

Growth rates - table

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Source: "The Algorithm Design Manual" by S. Skiena

Growth rates - graphically



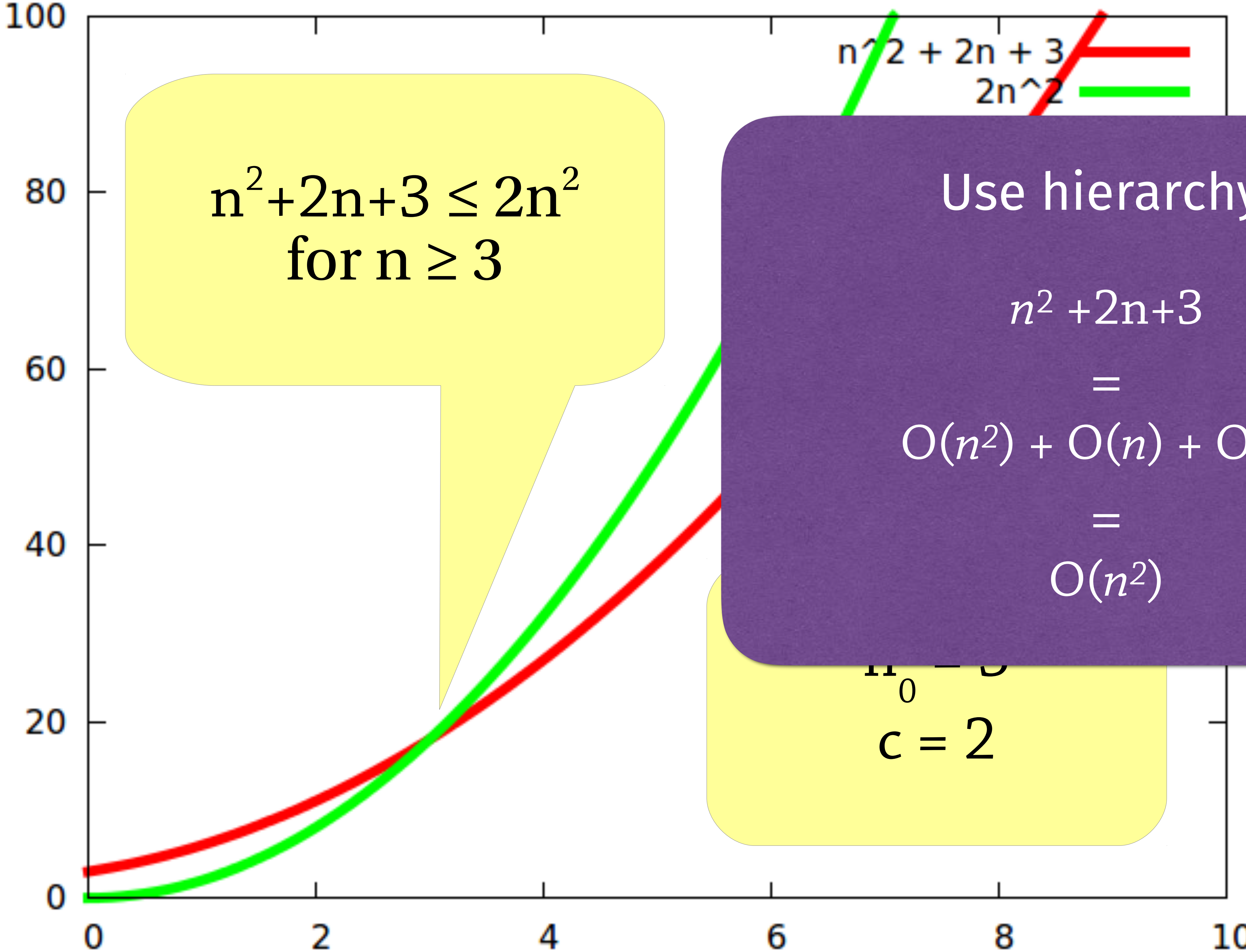
Adding big O (a hierarchy)

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

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 $O(n^2)$



$n^2+2n+3 \le 2n^2$
for $n \ge 3$

$n_0 = 3$
$c = 2$

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 the hierarchy!

- $n^2 + 11 = O(n^2) + O(1) = O(n^2)$
- $2n^3 + 3\text{n} - 1 = O(n^3) + O(\text{n}) + O(1) = O(n^3)$
- $n^4 + 2^n = O(n^4) + O(2^n) = O(2^n)$

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 certain time

Amortised analysis

A single append-operation for a dynamic array:

```
public void append(char c) {
  if (size == string.length) {
    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.

Amortised analysis

- Amortised analysis measures how much time each operation will take *in a sequence of operations*
- For the `append` method of a dynamic array the amortised complexity is O(1)
- There are different methods for amortising
  - 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...