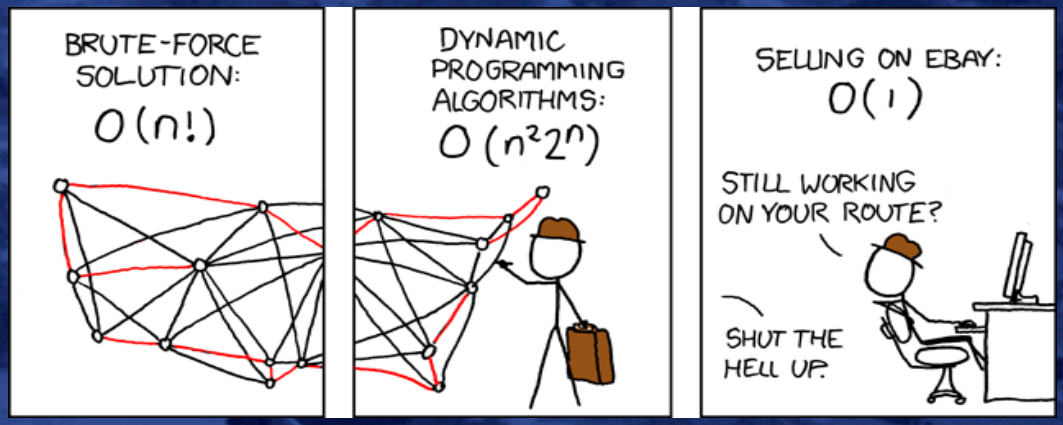
Algorithms Review

CITS3001 Algorithms, Agents and Artificial Intelligence



Tim French
Department of Computer Science and Software Engineering
The University of Western Australia

CLRS Chapters 2-3

2021, Semester 2

Summary

- We will review some basic algorithmic concepts from CITS2200 and CITS2211
  - Algorithm design
  - Complexity
  - Recurrence relations
  - Numerical stability
  - Proof techniques
- We will present examples of these concepts in action
  - Fibonacci numbers
  - Sorting algorithms

Problems, instances, algorithms, and programs

THE UNIVERSITY OF WESTERN AUSTRALIA

- A *computational problem* is a general (usually parameterised) description of a question to be answered

- A *problem instance* is a specific question usually obtained by providing concrete values for the parameters

- An *algorithm* is a well-defined finite set of rules that specifies a series of elementary operations that are applied to some *input*, producing after a finite amount
  of time some *output*
  - An algorithm solves any problem instance presented to it
  - Algorithms can be presented in many forms: code, pseudo-code, equations, tables, flowcharts, graphs, pictures, *etc*.

Problems, instances, algorithms, and programs (cont.)

THE UNIVERSITY OF WESTERN AUSTRALIA

- A *program* is (largely) the implementation of some algorithm(s) in some programming language

- Algorithms and data structures are the fundamental building blocks from which programs are constructed
  - Data structures represent the state of a program
  - Algorithms transform (parts of) this state as the program executes

Example

THE UNIVERSITY OF WESTERN AUSTRALIA

- Calculating Fibonacci numbers is a *problem*
  - The k^{th} Fibonacci number is the sum of the previous two Fibonacci numbers
- Calculating the eighth Fibonacci number is a *problem instance*
- The following two rules are a simple *algorithm* for solving this problem
  - $fib_k = fib_{k-1} + fib_{k-2}$, $k > 1$
  - $fib_0 = fib_1 = 1$
- A (basic, inefficient) *program* for fib might work by implementing these rules

Design and Analysis

THE UNIVERSITY OF WESTERN AUSTRALIA

- An algorithmic solution to a computational problem involves *designing* an algorithm for the problem, and *analysing* the algorithm, usually with respect to its correctness and performance

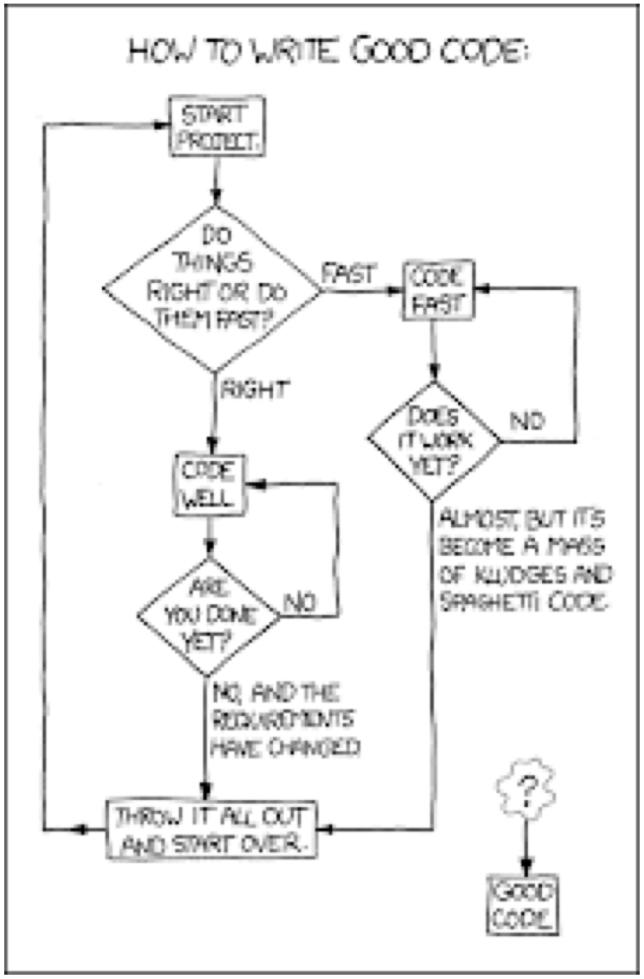
- Design requires both the background knowledge of algorithmic techniques and the creativity to apply them to new problems
  - Art as much as science

- Analysis is more mathematical/logical in nature, and usually more systematic
  - Science/engineering more than art

- In this unit (and in CITS2200) you will learn about both

Design Process

1. What problem are we trying to solve?
2. Does a solution exist already?
3. Otherwise can we find a solution, and are there multiple solutions?
4. Is the algorithm correct?
5. Is the algorithm efficient? Is it efficient *enough*?
6. Does the implementation of the algorithm present any problems?



An Example: Fibonacci Numbers

THE UNIVERSITY OF WESTERN AUSTRALIA

The most obvious algorithm for calculating Fibonacci numbers is the one we saw earlier

fib(k)= fib(k–1) + fib(k–2), if k > 1
= 1, if k <= 1

Consider the runtime of this algorithm: ->

What do you notice about these times?

fib(25)	25ms
fib(26)	40ms
fib(27)	65ms
fib(28)	108ms
fib(29)	173ms
fib(30)	283ms
fib(31)	455ms
fib(32)	734ms
fib(33)	1,194ms
fib(34)	1,932ms
fib(35)	3,116ms
fib(36)	5,069ms
fib(37)	8,187ms
fib(38)	13,237ms

Fibonacci Numbers cont.

THE UNIVERSITY OF WESTERN AUSTRALIA

We can improve this algorithm easily, just by calculating “from the bottom up”

fib(k) = fib’(k, 1, 1)

fib’(k, x, y) = x, if k == 0

= fib’(k–1, y, x+y), if k > 0

fib(38)	0.013ms
fib(10^3)	0.208ms
fib(10^4)	4.59ms
fib(10^5)	358ms

This is a linear algorithm: the times are bette

This is a (very) basic instance of *dynamic programming* (or *memoisation*), which we will meet later

We can also derive a closed form for the Fibonacci sequence, by solving the recurrence relation

Comparing sorting algorithms

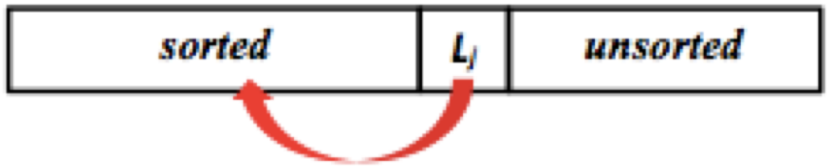
Sorting a sequence of items means generating a permutation of the sequence where each adjacent pair
of items obeys some comparison relation

- Sorting is one of the most studied problems in CS
- In pre-Internet days, it was often said that at any given moment, 90% of the computers in the world would be sorting

One common sorting algorithm is insertSort

```
procedure insertSort(L):
for j = 2 to length(L)
      key = L[j]
      i = j - 1
      while i > 0 and L[i] > key
            L[i + 1] = L[i]
            i = i - 1
      L[i + 1] = key
```

This is an example of *pseudo-code*



Correctness of insertSort

THE UNIVERSITY OF WESTERN AUSTRALIA

- We can establish the correctness of insertSort by considering an *invariant* INV
- The invariant in this case is that "after n iterations of the main loop, the first n+1 items on L are sorted"
- There are three parts to verification using invariants

- **Initialisation**: INV must be true at the start:
  - After zero iterations, the first item is sorted

- **Termination**: if INV is true at the end of the procedure, that must imply that the procedure has sorted L
  - The procedure performs length(L) – 1 iterations, after which length(L) items are sorted

- **Maintenance**: we must prove that if INV is true after n iterations, then it is also true after n+1 iterations
  - Usually done using mathematical induction
  - Informally, the inner loop doesn't change the order of the already-sorted items, and it places the new item into the list in front of all bigger items, and behind all smaller items

Aside 1: proof by contradiction

- Another proof technique you need to be familiar with is *proof by contradiction*
- The idea is that to prove that a proposition *P* is true,
  we assume that *P* is false, and show that that assumption leads to a contradiction
  - If ~*P* leads to a falsehood, *P* must be true
- e.g. prove that it is impossible to construct a set *S*
  that contains all of the real numbers between 0 and 1

Proof:

We assume that we can construct such a set: call the set *S*
List the elements of *S* as shown

$0.x_{11}x_{12}x_{13}x_{14}x_{15}x_{16}\ldots$
$0.x_{21}x_{22}x_{23}x_{24}x_{25}x_{26}\ldots$
$0.x_{31}x_{32}x_{33}x_{34}x_{35}x_{36}\ldots$
…
(we can pad any number with 0s)

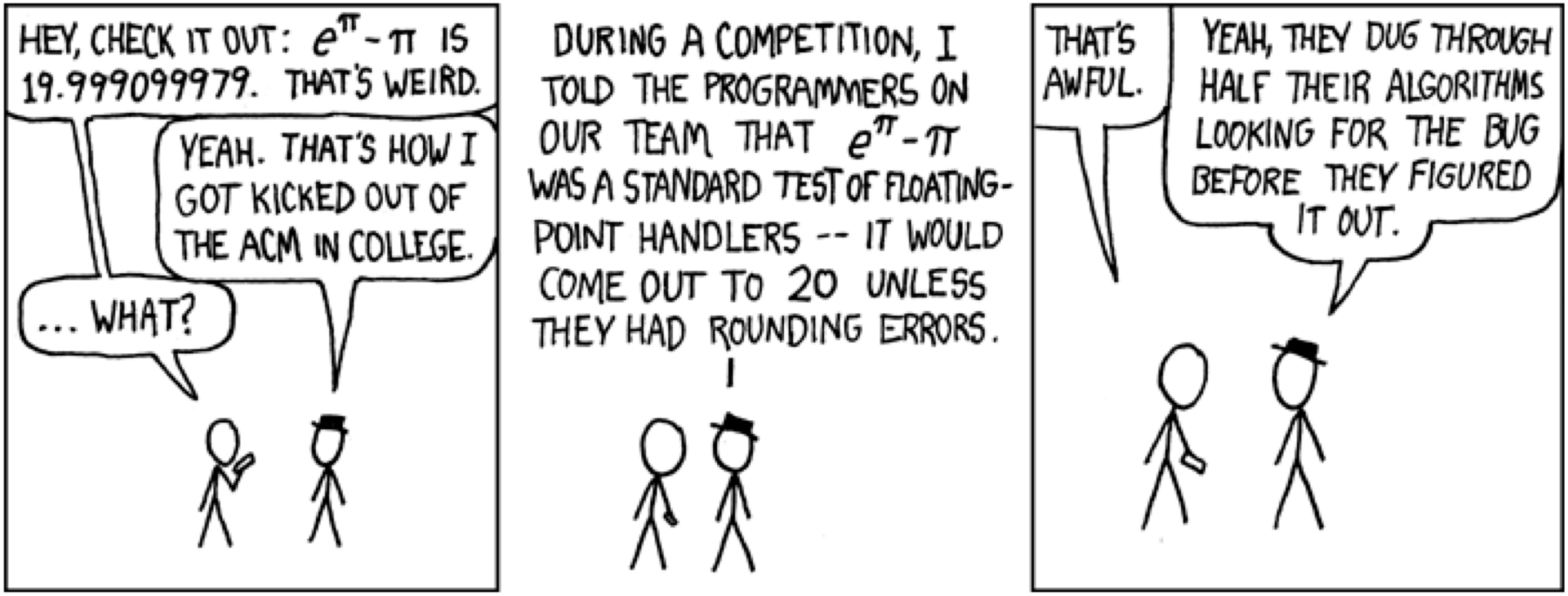
Now construct a number $0.y_1y_2y_3y_4y_5y_6\ldots$
where
$y_k = 5$, if $x_{kk} \neq 5$
$y_k = 6$, if $x_{kk} = 5$
Clearly $0 < y < 1$, and clearly $y \neq x_k$, for all k
So $y \notin S$: contradiction!

Aside 2: Numerical Stability

THE UNIVERSITY OF WESTERN AUSTRALIA

- Another aspect of the correctness of an algorithm is its numerical stability, if it performs real arithmetic
  - This is an implementation issue really
- Computers represent real numbers to a certain precision
  - Thus most real numbers must be approximated, implying small errors
- Computers use a lot of digits in their precision, so usually these errors don't matter
- But for algorithms with many arithmetic operations, the errors can accumulate and get big enough to matter
- Two common operations to watch out for are
  - subtracting nearly-equal numbers
    - if $x - y = z$, then $(x+e) - (y-e) = z+2e$, which might be substantially different if $z \approx e$
  - dividing by very small numbers
    - x / y and $x / (y \pm e)$ might be substantially different if $y \approx e$
- We won't explore this issue explicitly any further

Numerical Stability and the ACM



- Find out about the ACM ICPC and other programming competitions at:

  https://pcs.org.au/

Complexity of InsertSort

For simple procedures, it is easy to figure out the complexity from the pseudo-code (or from the code)

Given n = length(L), the main loop does n–1 iterations,
each of which does 3 assignments

In the worst case, the inner loop does j–1 iterations,
each of which does 2 assignments

Thus insertSort does 3(n–1)+2(1 + 2 + .. n–1) assignments, i.e. $n^2+2n–3$

How many assignments does it perform in the best case?

How many assignments does it perform if we use binary search to find the appropriate place for L[j]?

```
procedure insertSort(L):
for j = 2 to length(L)
  key = L[j]
  i = j - 1
  while i > 0 and L[i] > key
     L[i + 1] = L[i]
     i = i - 1
     L[i + 1] = key
```

Big-O notation

THE UNIVERSITY OF WESTERN AUSTRALIA

- The complexity of an algorithm gives us a device-independent measure of how much time it consumes
  - Does not depend on the details of any particular machine, or language, or programmer, *etc*.

- What we normally care about is the *growth-rate* of this measure, i.e. how it changes with the size of a problem instance
  - For insertSort, the size of an instance is the length of its argument

- *Big-O notation* abstracts away from the details of the calculation, focusing on the largest term
  - Thus insertSort is $O(n^2)$

Asymptotic Analysis

THE UNIVERSITY OF WESTERN AUSTRALIA

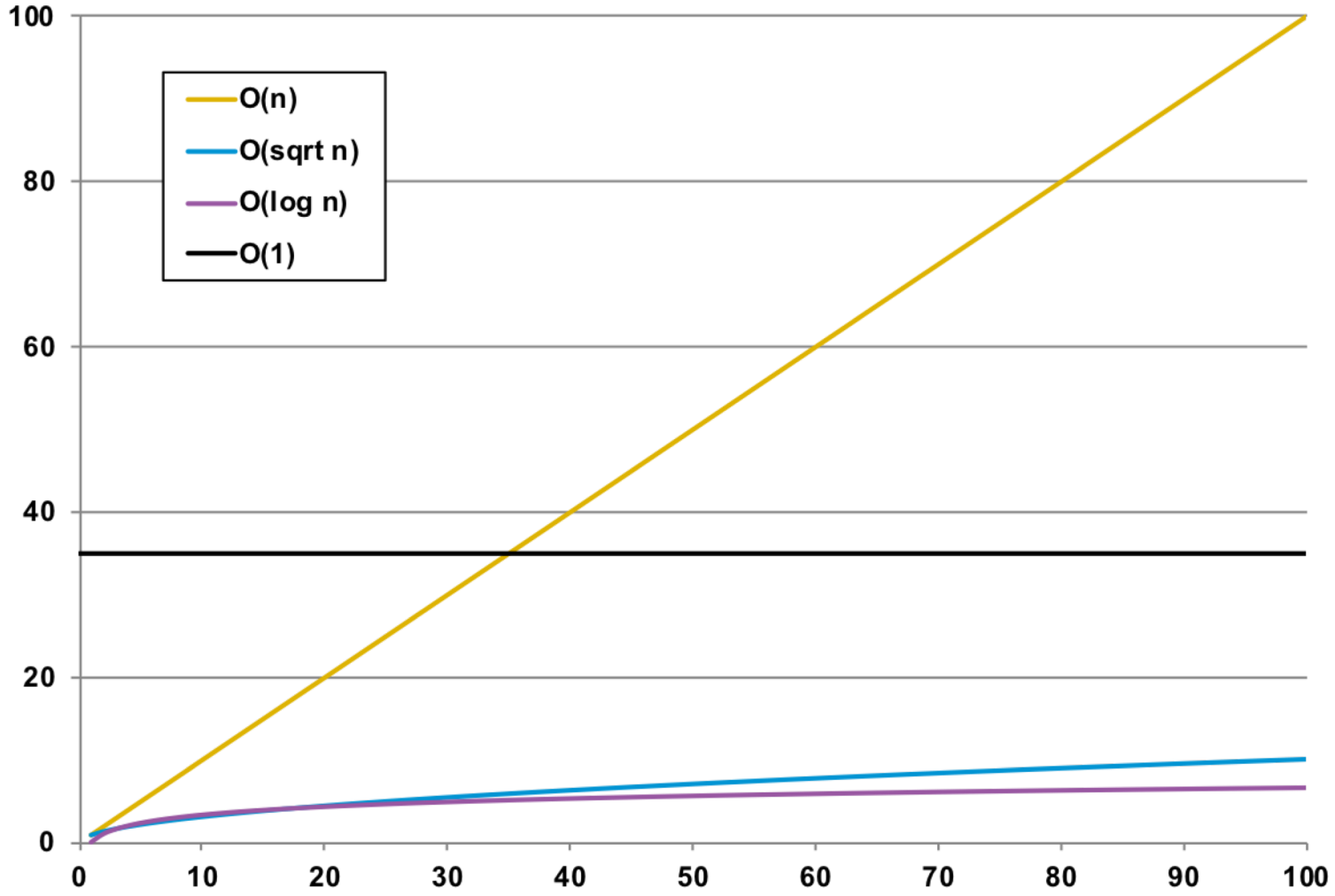
- Growth-wise, this is more important than the coefficients of the terms in the expression
  - Over time, we need to solve bigger instances
  - Over time, our capability tends to grow – leading us to look at bigger instances!

- Big-O notation specifies an upper-bound on growth
  - Technically an algorithm that is $O(n^2)$ is also $O(n^3)$

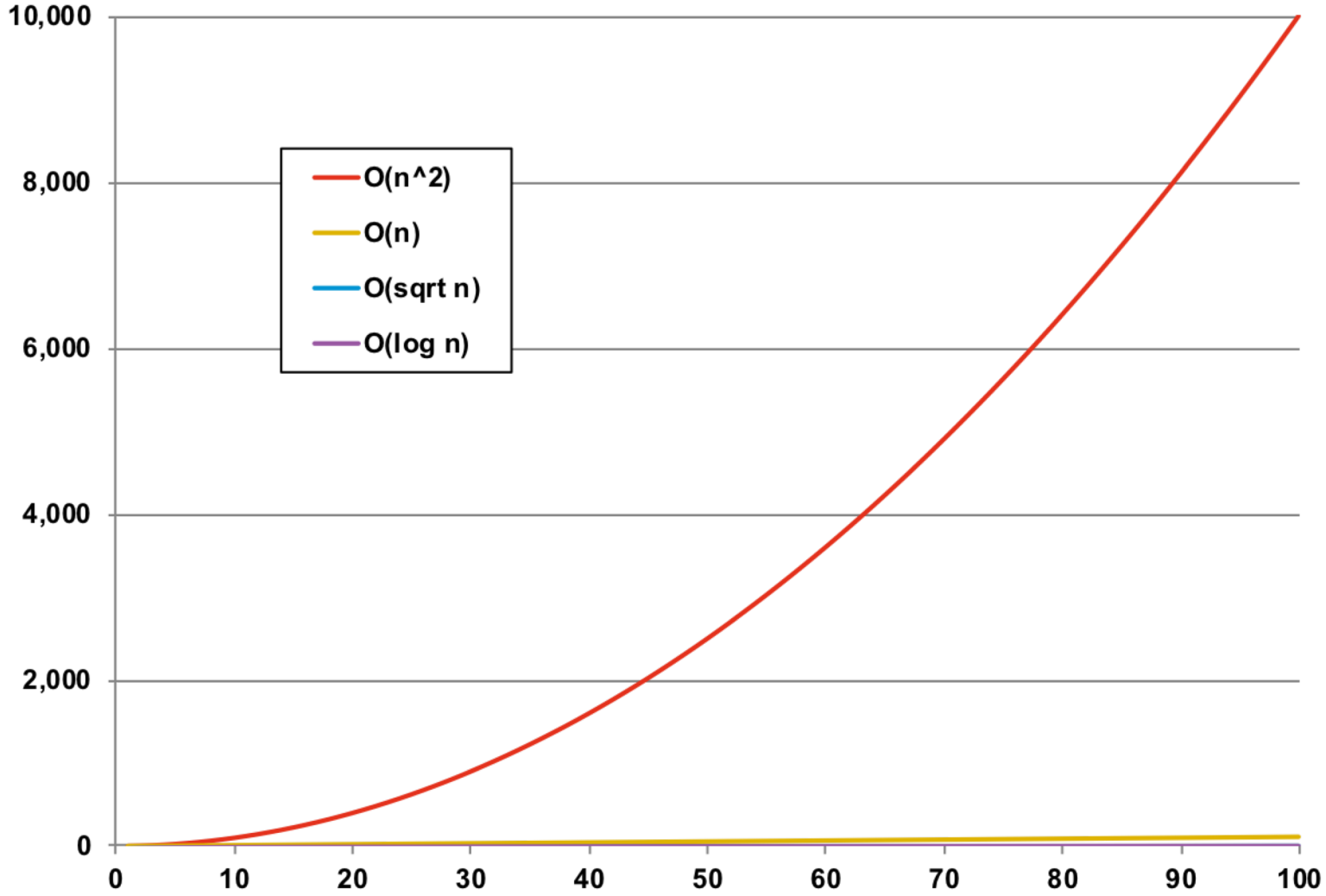
- Big-Θ notation specifies both lower- and upper-bounds
  - So an algorithm that is $\Theta(n^2)$ is *not* also $\Theta(n^3)$

- You may also come across Ω-notation, little-o notation, ω-notation, …

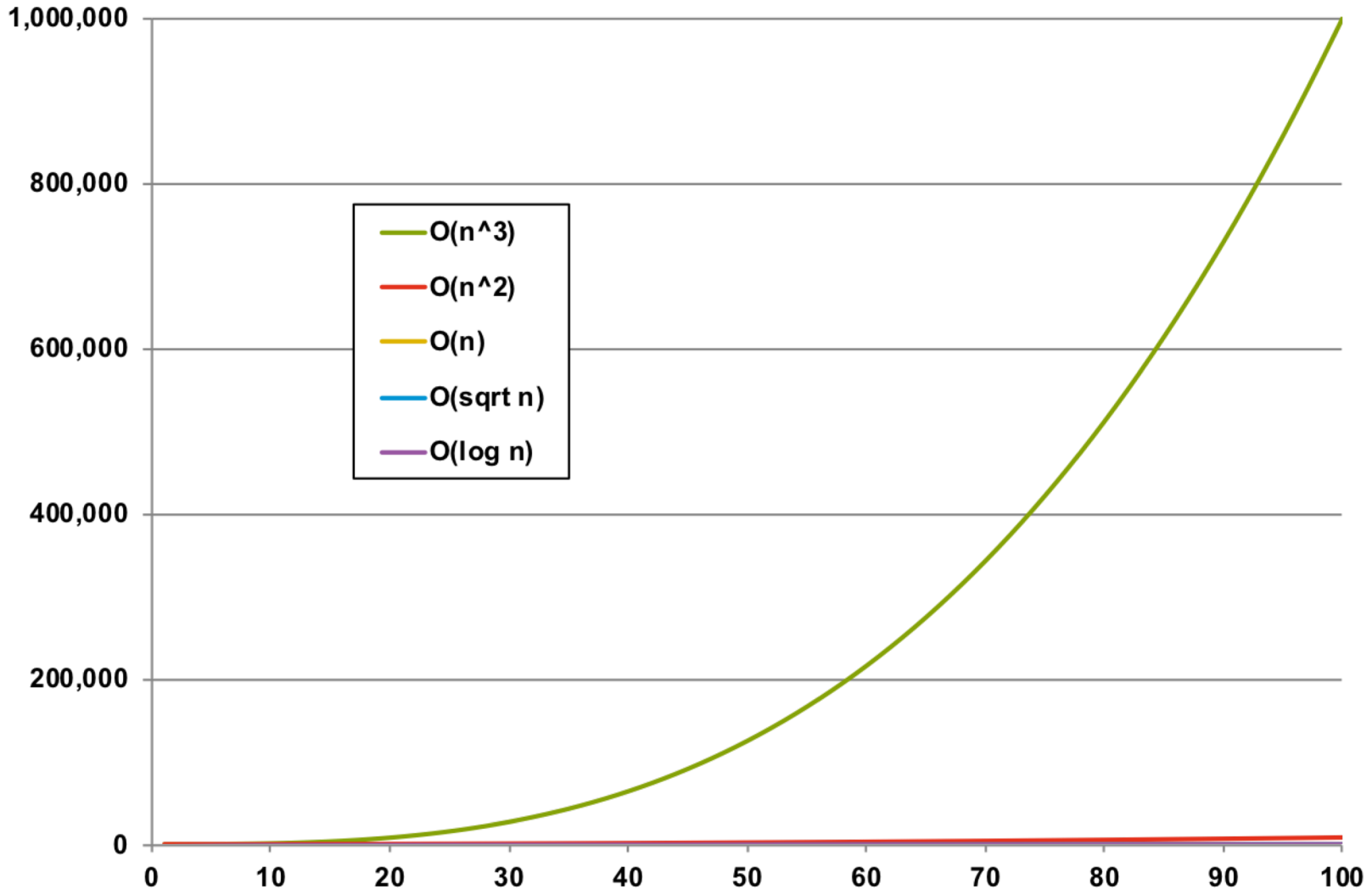
Some Graphs



More Graphs

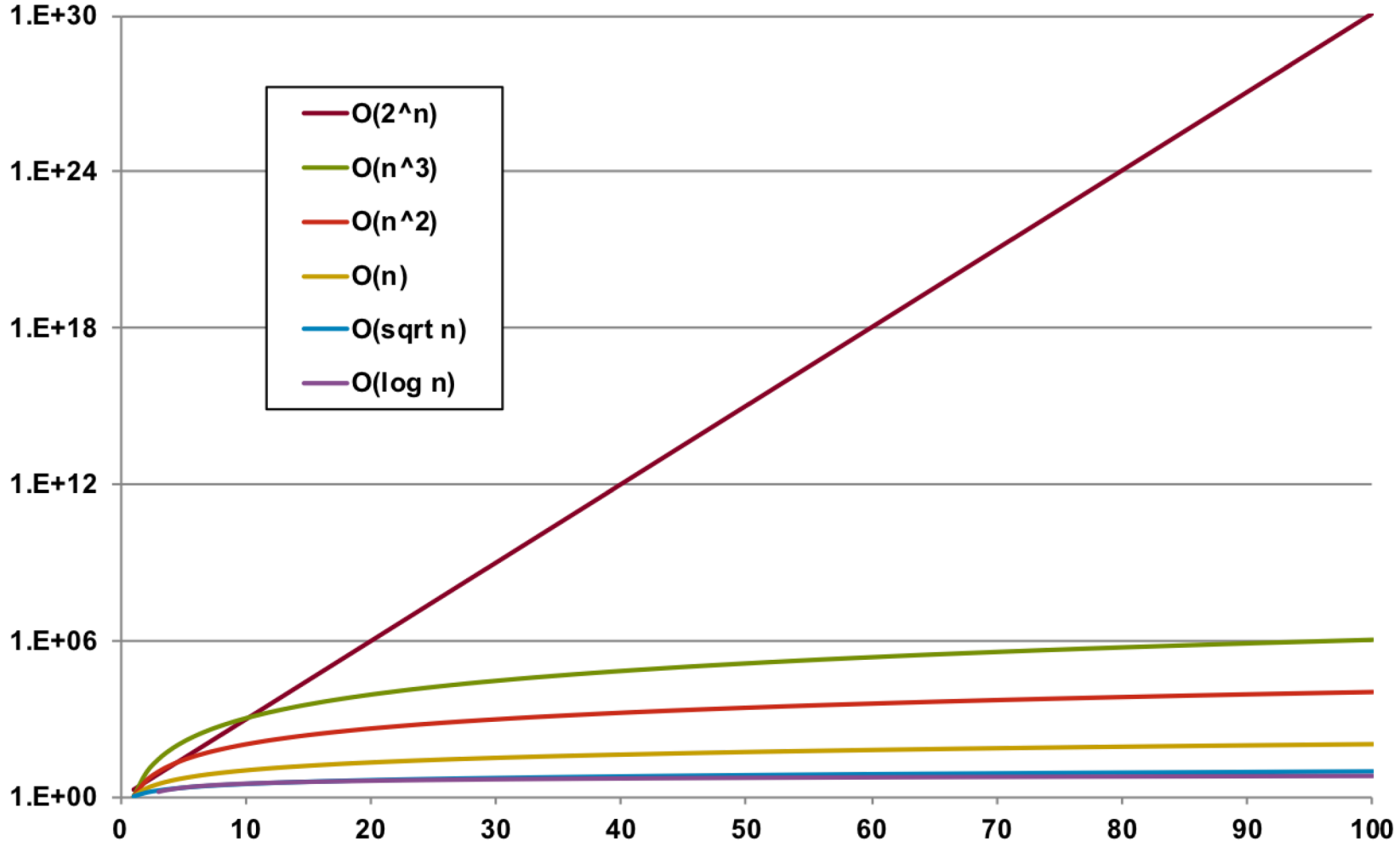

10,000
8,000
6,000
4,000
2,000
0
0
10
20
30
40
50
60
70
80
90
100
O(n^2)
O(n)
O(sqrt n)
O(log n)

…and More Graphs

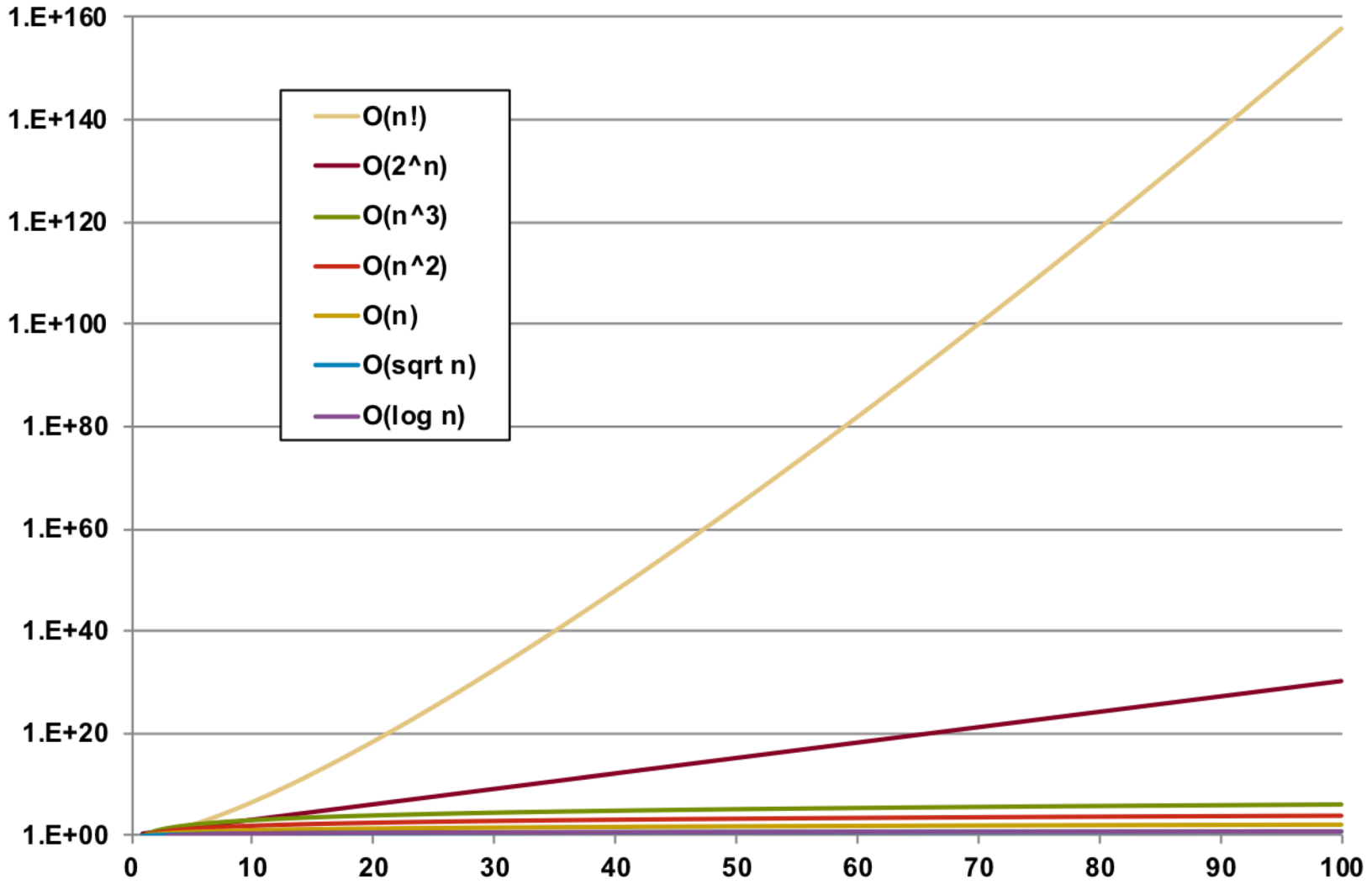


…and More Graphs

THE UNIVERSITY OF WESTERN AUSTRALIA



…and More Graphs



Complexity

THE UNIVERSITY OF WESTERN AUSTRALIA

Notice that each new line (complexity) completely dominates all previous ones

Mathematicians and theoretical computer scientists distinguish only between

- Polynomial/tractable algorithms
- Exponential/intractable algorithms

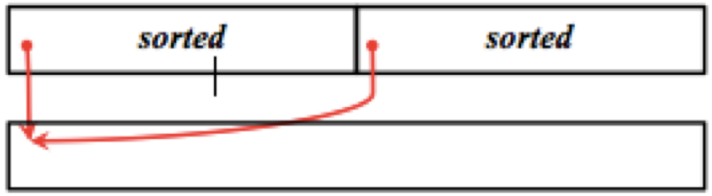
Engineers (and users!) also distinguish between other categories of algorithms

But people even care about intractable problems too…

Back to sorting

mergeSort works by separately sorting the front and back halves of a list, then merging them together whilst maintaining the ordering

```
procedure mergeSort(L, p, r): // sorting indices p – r of L
if p < r
     q = floor ((p + r) / 2)
     mergeSort(L, p, q)
     mergeSort(L, q+1, r)
     merge(L, p, q, r)
```



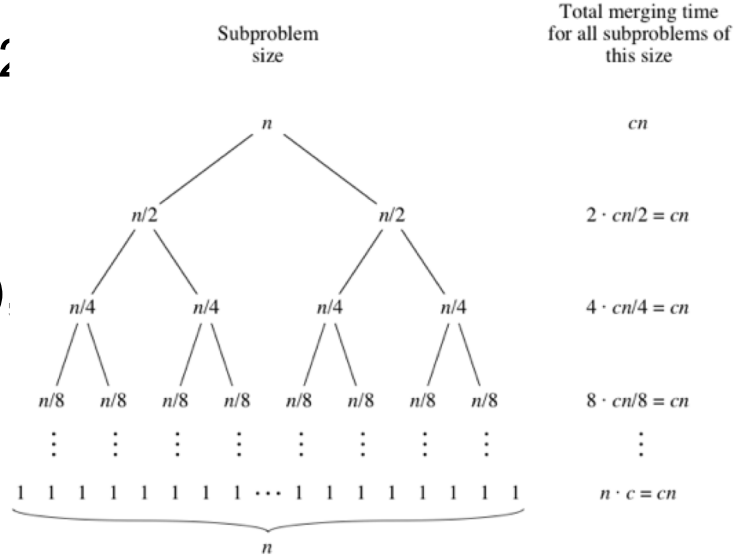

```
procedure merge(L, p, q, r)
// pre: items p – q are sorted, items q+1 – r are sorted
// post: items p – r are sorted
```

Complexity of mergeSort

- In each call to mergeSort:
  - The call to merge takes O(r – p) time
  - The argument-length is halved for the recursion

- Letting n be the length of the original list:
  - Because of the halving, there are log(n) “levels”
  - At the first level, there is 1 call to merge with size n
  - At the second level, 2 calls each with size n/2
  - At the third level, 4 calls each with size n/4

- At each level, the total time to run merge is O(n) so the total time taken is nlog(n)
  - Far better than insertSort!



Worst-case, best-case and average case analysis

THE UNIVERSITY OF WESTERN AUSTRALIA

- Different problem instances of the same size may take different amounts of time to process

- The analyses so far have been *worst-case*
  - Assumes the data is in its least favourable form
  - This is important because it gives a guaranteed upper bound on the algorithm's performance

- *Best-case* analysis assumes the data is in its most favourable form
  - Usually leads to less runtime
    (e.g. for insertSort, best-case is O(n))
  - But not always
    (e.g. makes no difference for mergeSort)

- *Average-case* analysis attempts to analyse how much time the algorithm needs on "typical" data
  - Much harder to do!

- Note though that worst-case analysis can be difficult too
  - Sometimes we have to imagine data that is worse than any possible real data