Optimizations and algorithms
MCNet computing school 2016

Chris Pollard

University of Glasgow, MCNet

2016 05 18

Intro

This is meant to be a short introduction to *thinking about* optimization strategies and well-established algorithms that are at your disposal.

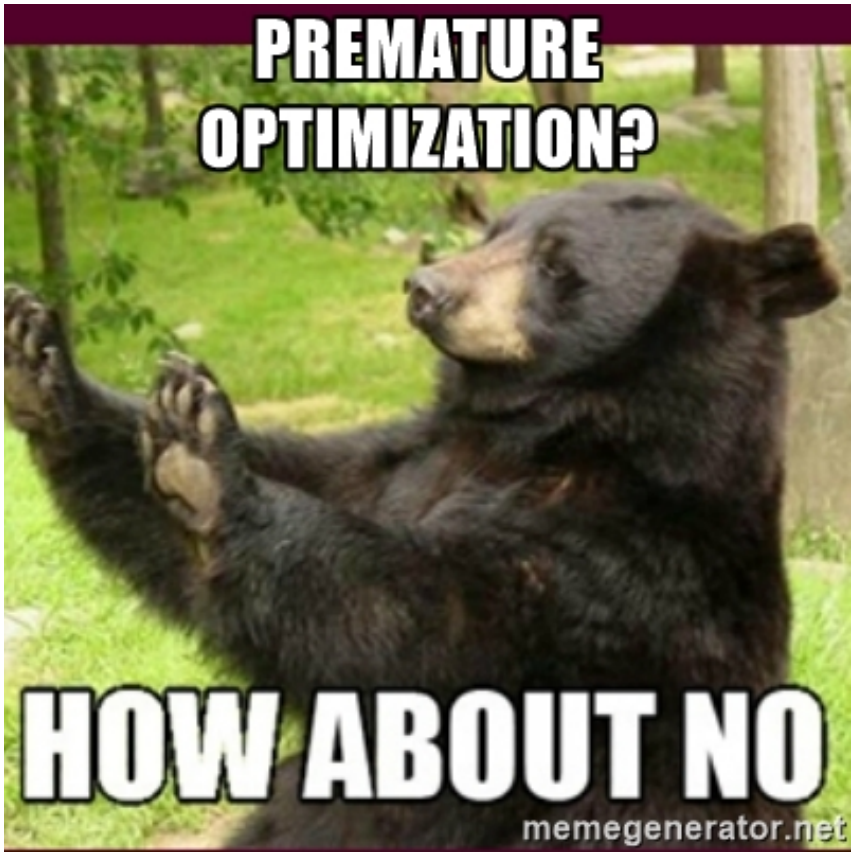
This lecture won't be *that* interactive, but please stop me and ask questions at any point.

I'm not a computer scientist, but I have a lot of experience wasting time: this talk is about learning to be efficient as a physicist when it comes to coding.

This is the most important slide in the talk

Never, ever optimize unless you have good reason to.

- Why do you need to optimize?
- Do you have a clear plan of action?
- What do you expect to gain?
- How long will it take?
- Are you still sure it's worth it?



Consider...

Things to consider when you're deciding if optimization is a good idea:

- CPU resources (memory, time) are cheap!
- Is there a simple solution that doesn't involve changing code?
- Have you identified the piece of code that is slowing things down (profile!)?
- Am I at risk of introducing new bugs?
- Will this make my program more difficult to use or understand?
- Am I premature?

Non-invasive fixes

"Non-invasive" strategies you should think about before diving in:

- Compiler optimizations (-OX) (careful!)
- Parallelization
- Others?

What to optimize?

If you've made it this far, then let's talk about what we want to make more efficient.

- time
- memory usage
- disk usage
- electricity
- compile times
- ease of use/API
- ease of understanding/development
- ...

Low-hanging fruit

I take this opportunity to encourage all of you to *be lazy*!

Spend some time looking for a previously-developed solution: take advantage of already-written tools (C++ and python libraries are very extensive to begin with).

Stay lazy

I take this opportunity to encourage all of you to *still be lazy*!

Identify the bottleneck before continuing. Other optimizations are very likely not worth your time.

Focus on the inner-most loop of your algorithm first.

Example: speeding up a once-per-event computation will pay off much more quickly than a once-per-run computation.

Thinking about optimizations

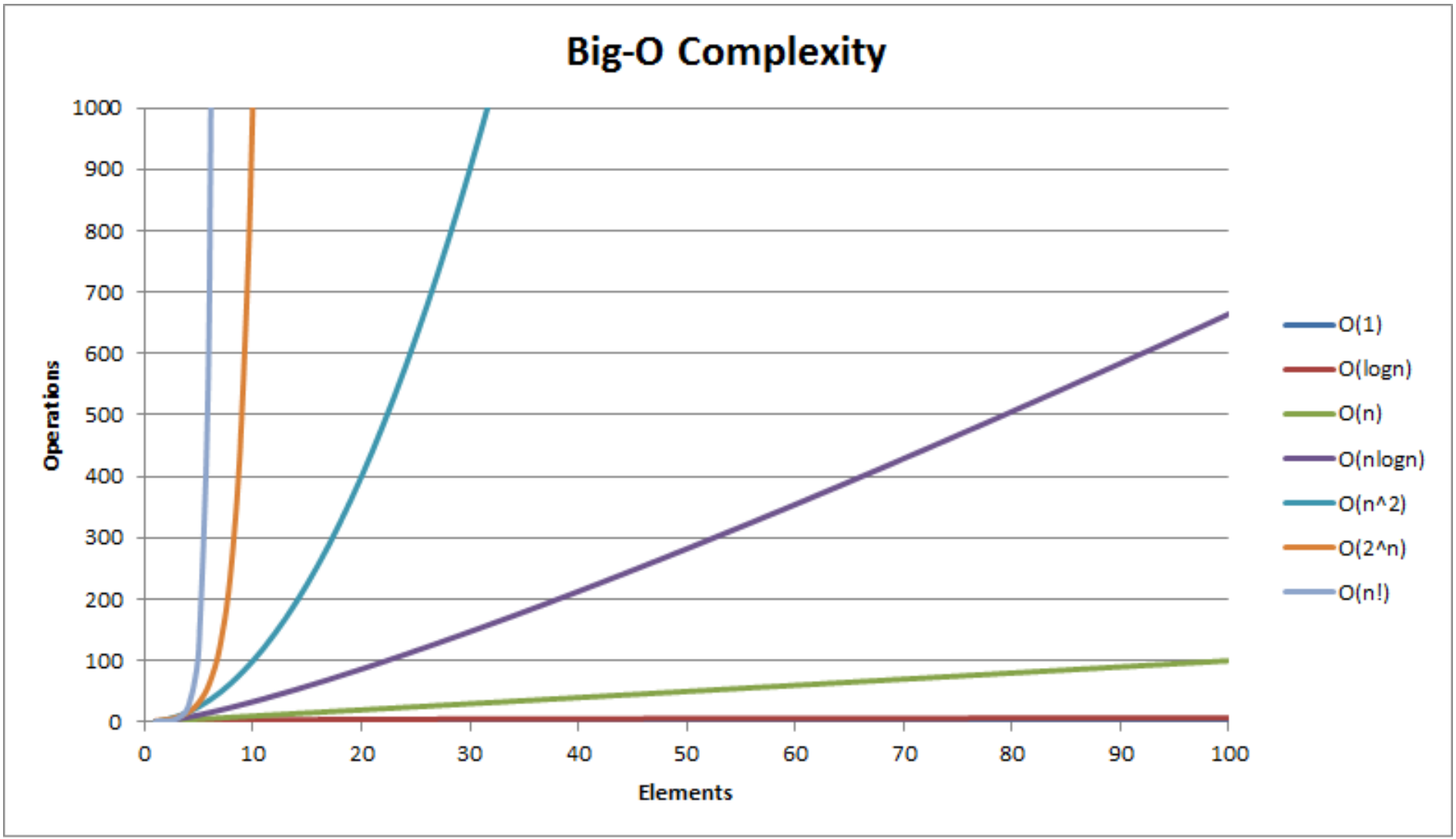
I'm going to focus on how to think about time complexity and memory usage optimizations.

We often talk about the scaling behavior of an algorithm or data structure in certain limiting cases.

Big-O notation is used to classify this scaling behavior.

$O(n^2)$ ("goes like n squared" or "scales as n squared") means that for large values of n, our computation or structure grows as n^2 (usually time or space).

Big-O scaling



Gotcha!

Warning: often the best-case scaling, average scaling, and worst-case scaling *are not the same*!

Be aware that certain corners of phase space may make your life difficult: think about this when you pick your algorithms.

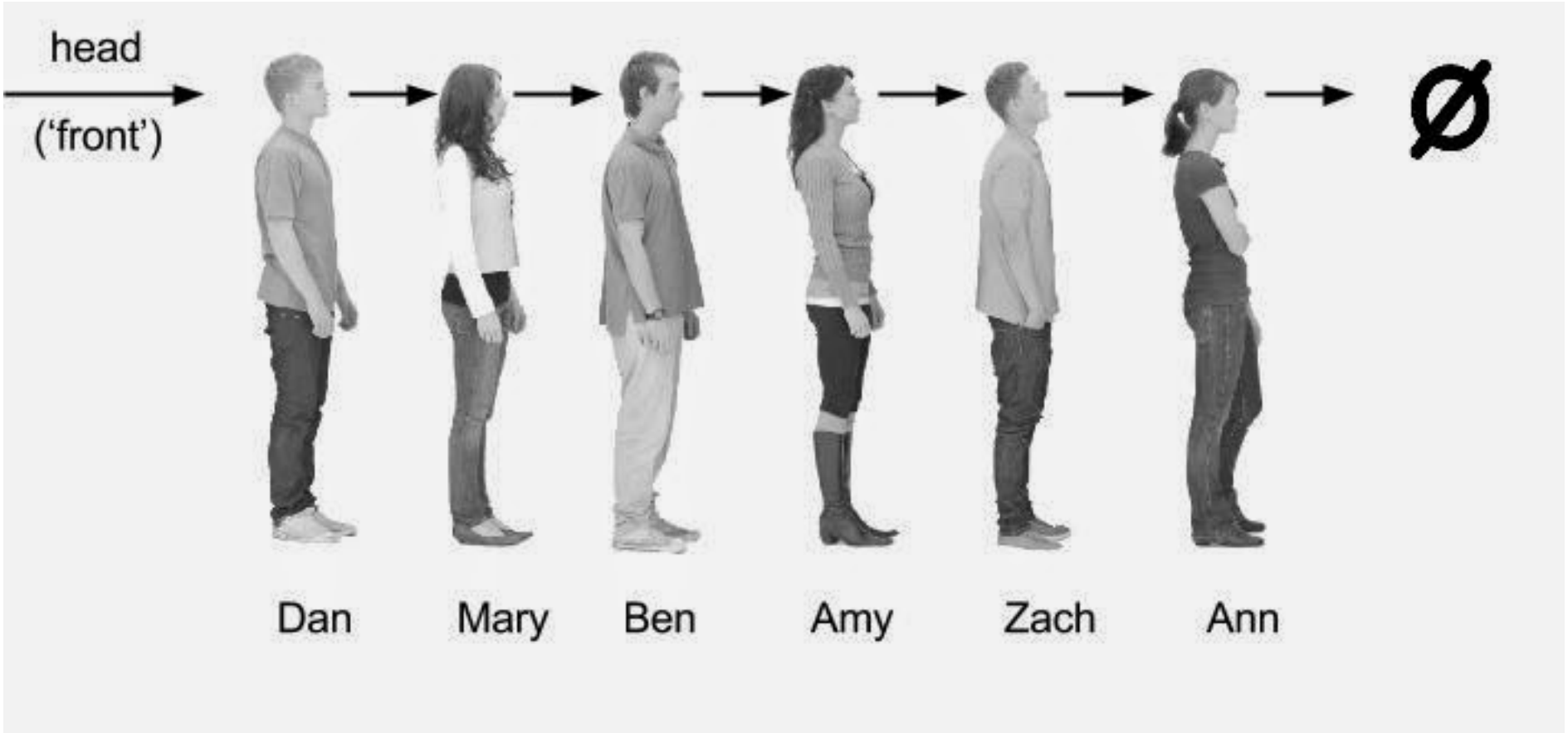
Common tasks

There are many common tasks that have well-established, optimized data structures or algorithms already.

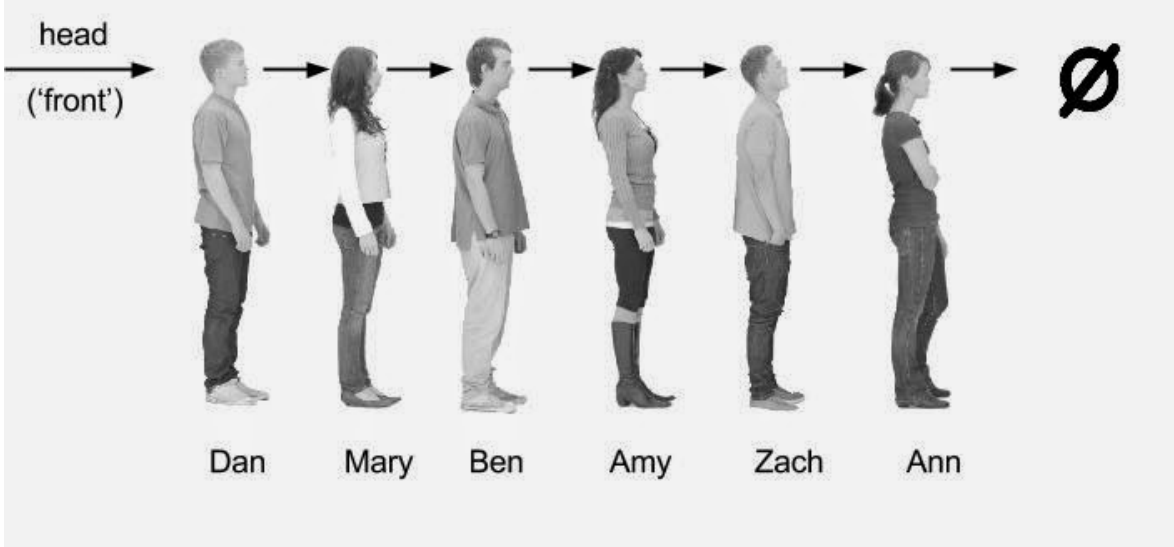
If you can decompose your algorithm into some combination of these, it should be easy to use predefined recipes for speed/memory improvements.

- access the *n*th item in a collection
- sort a collection
- search for an object
- access items sequentially
- insert an item
- prepend or append an item
- remove an item (from front or back?)

Example: linked list



Example: linked list



What is scaling behavior of. . .

- accessing the nth element?
- inserting an element at position n?
- prepending an element?
- determining if x is a member of the list?

Arrays

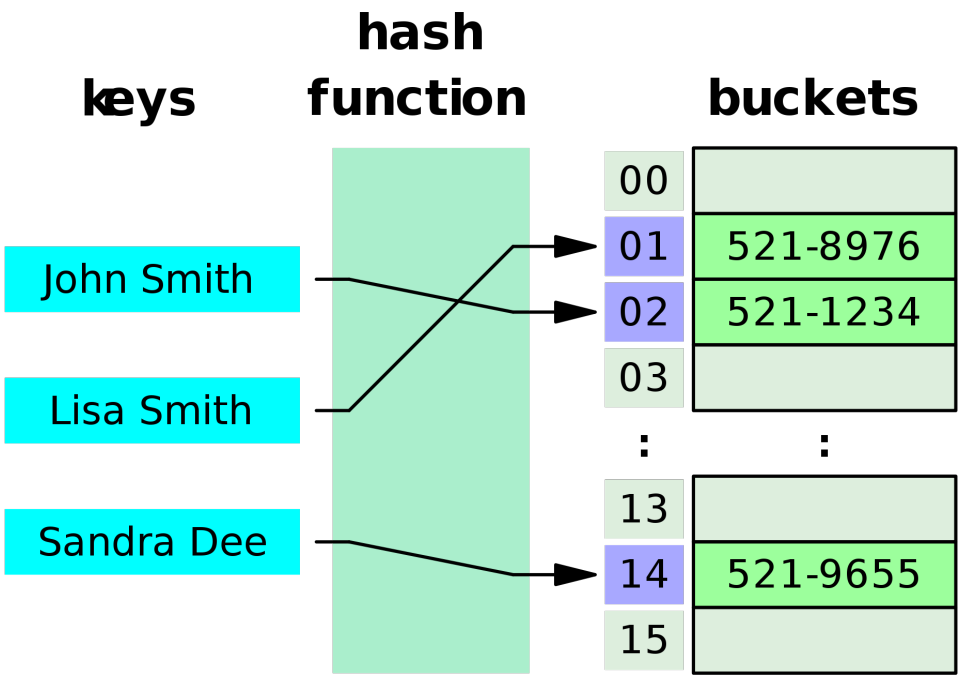
Arrays are stored sequentially in memory and therefore have very fast access (C arrays, std::vectors, python list). Average scaling:

- access: $O(1)$
- search: $O(n)$
- insert: $O(n)$
- delete: $O(n)$

Hash tables

Hash tables store objects by a(n almost unique) hash and have fast search, insert, delete (python dictionary). Average scaling:

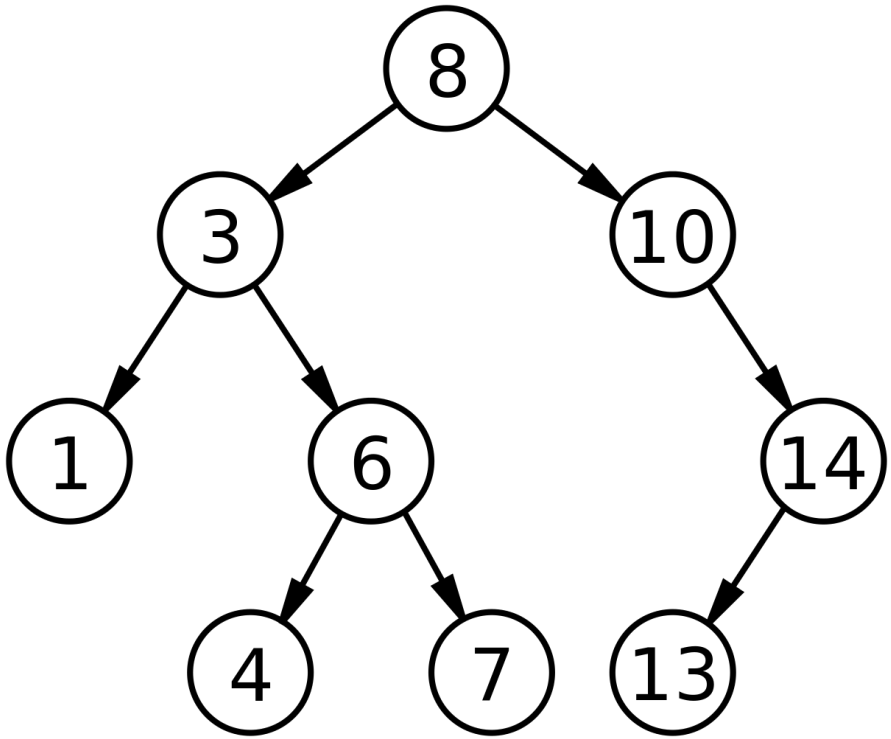
- access: -
- search: $O(1)$
- insert: $O(1)$
- delete: $O(1)$



Binary search trees

Binary search trees store objects based on their *ordering* and have a balance of speed for different operations. Average scaling:

- access: $O(\log n)$
- search: $O(\log n)$
- insert: $O(\log n)$
- delete: $O(\log n)$



Sorting algorithms

Algorithms for sorting a collection of objects is an extremely common task.

I'll cover two common ones and their strengths/weaknesses but otherwise leave it up to you to explore.

Remember: if you need sorted data, consider a sorted data structure!

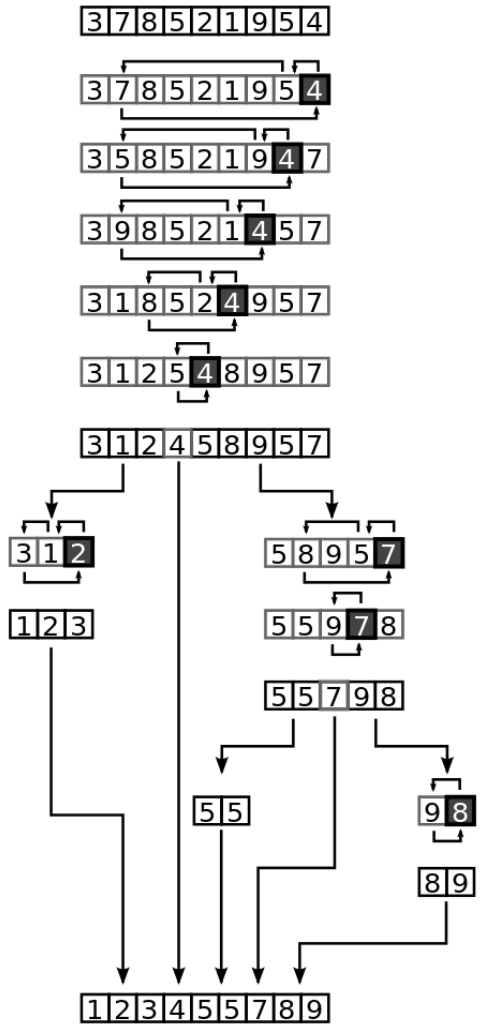
Quicksort

Procedure:

- pick an element, called a *pivot*.
- reorder elements so that all that are $>$ the pivot come after it and all that are $<$ come before.
- recursively apply the first two actions to the sub-arrays above and below the pivot.

Scaling:

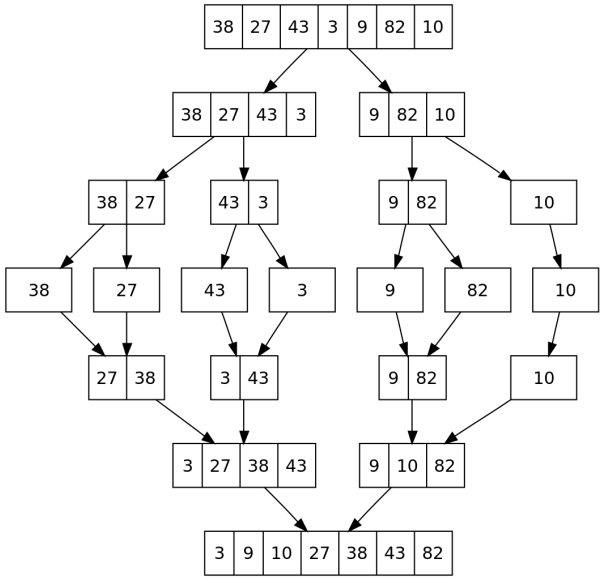
- average time complexity: $O(n \log n)$
- worst time complexity: $O(n^2)$
- worst extra space complexity: $O(\log n)$



Mergesort

Procedure:

- divide your collection into n subcollections
- repeatedly merge pairs of subcollections, sorting along the way (each subcollection is naturally already in order).
- end when there is only one sorted collection left.



Scaling:

- average time complexity: $O(n \log n)$
- worst time complexity: $O(n \log n)$
- worst extra space complexity: $O(n)$

Further info

As usual, there is a lot of information available on the web (wikipedia, stackoverflow, etc).

I recommend you take a look at the "Big-O Cheat Sheet" (http://bigocheatsheet.com/).

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)

Final remarks

Professional developers can write $O(5)$ bug-free lines of code each day (including appropriate validation).

This number is slightly higher in high-level languages like python, but also (slightly?) lower for non-professionals.

To give you an idea of how long computations, disk/network accesses, etc. take compared to every-day time scales, have a look at https://gist.github.com/hellerbarde/2843375.