Profiling

David Grellscheid

UNIVERSITETET I BERGEN

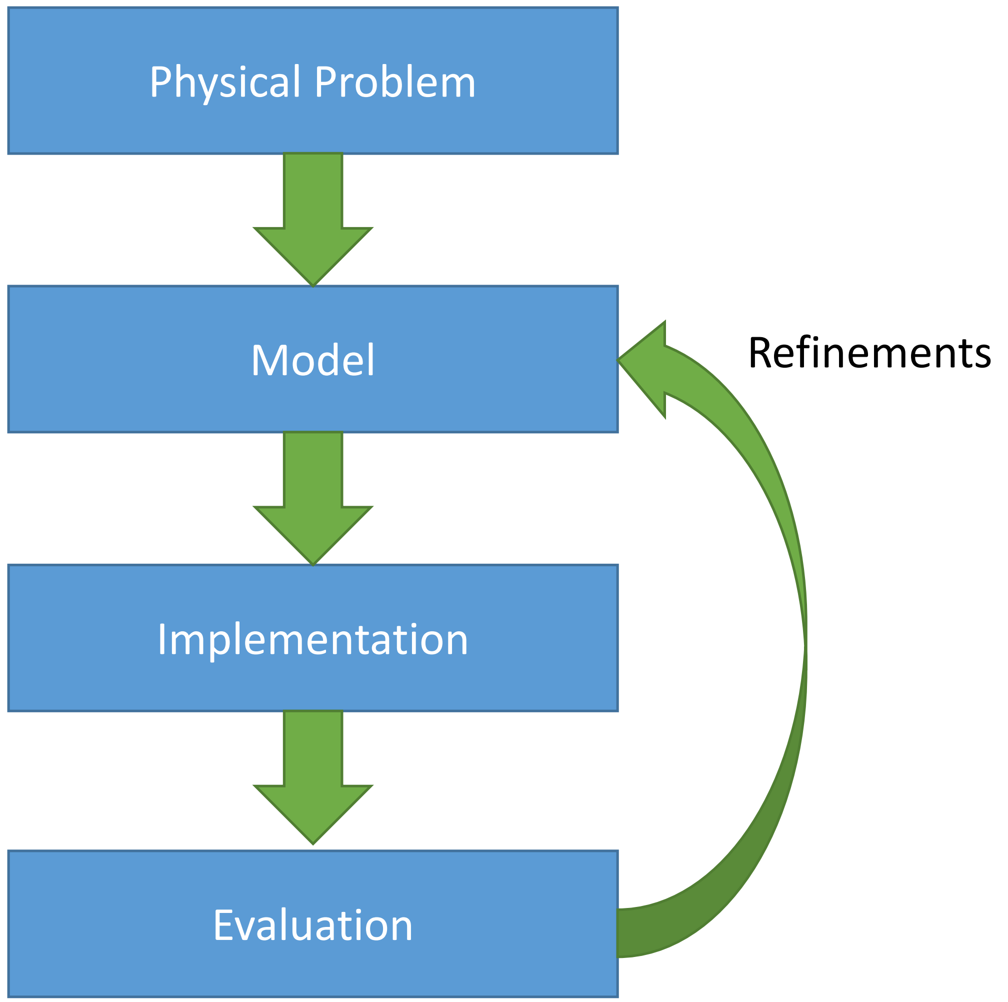
Typical scientific workflow

Correctness is main concern

Start coding without much planning

First version that looks like it works is kept

Sub-optimal choices only noticed later on
(if at all)



Typical scientific workflow

Correctness is main concern

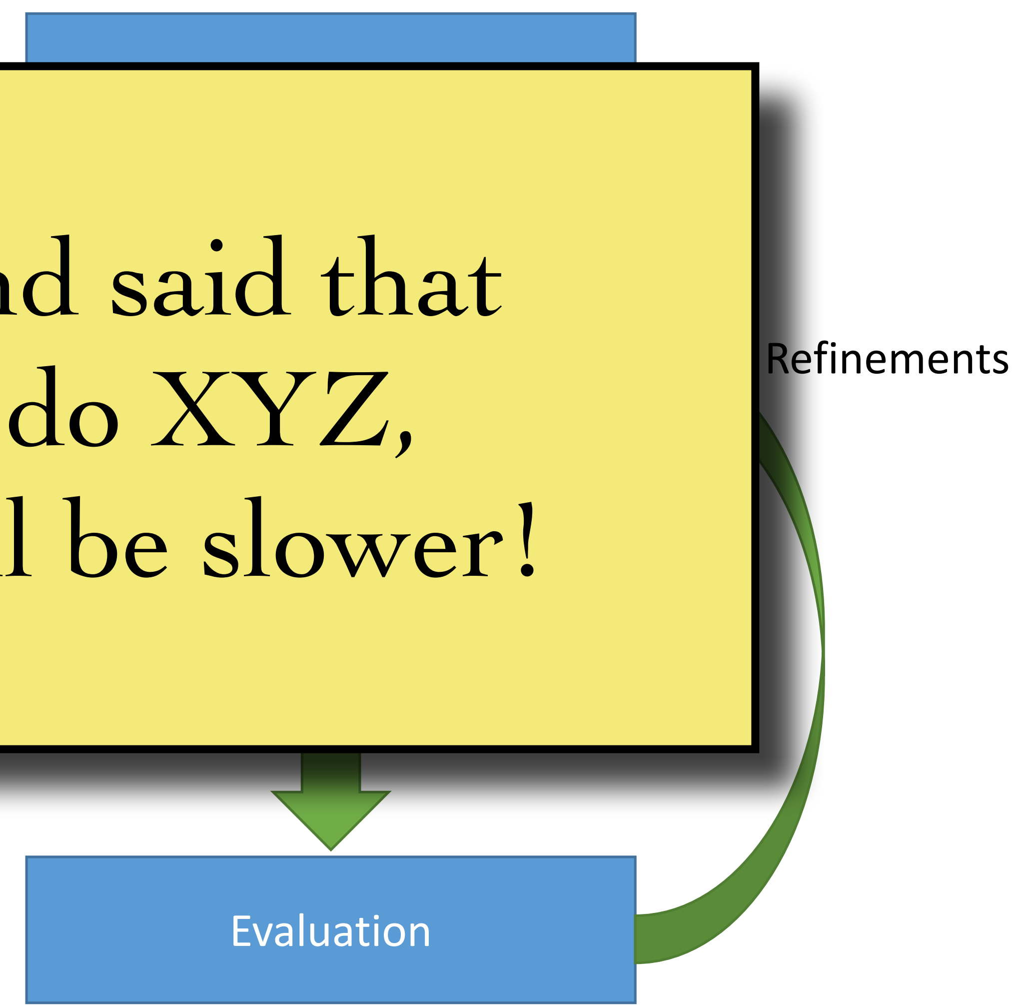
S

Fi

like it works is kept

Sub-optimal choices only noticed later on (if at all)

A friend of my friend said that you should never do XYZ, because the code will be slower!



Donald Knuth, December 1974:

> Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.
> Yet we should not pass up our opportunities in that critical 3%.

“Structured Programming with go to Statements”, Computing Surveys, Vol 6, No 4.

Runtime is not the only factor to consider,
need to think about trade off between time spent in:

development
debugging
validation
portability
runtime in your own usage
other developers' time (now/future)
total runtime for all users

CPU time much cheaper than human time!

Reusability is an efficiency!

If the student after you has to start from zero,
all your work is wasted

Optimization points

Someone else already solved (part of) the problem:

LAPACK, BLAS
GNU scientific library
C++ Boost
Numpy, Scipy, Pandas
…

Develop googling skills, evaluate what exists.
Quality often **much** better than self-written attempts

Optimization points

Choice of programming language

Be aware of what exists

Know strengths / weaknesses

But: needs to fit rest of project

take a look at Haskell, Erlang, Prolog
to get an idea how different the approaches can be

Optimization points

```
findLongestUpTo :: Int -> (Int,Int)
findLongestUpTo mx = maximum ( map f [1 .. mx] )
   where f x = (collatzLength x, x)


collatzLength :: Int -> Int
collatzLength 1 = 1
collatzLength n = 1 + collatzLength (collatzStep n)


collatzStep :: Int -> Int
collatzStep n
   | even n    = n `div` 2
   | otherwise = 3 * n + 1
```

Optimization points

Program design

First version: understand the problems

now start again!

Second version: you know what you're doing

refactor / clean up / make reusable

Done :-)

Optimization points

Algorithm / data structure choice

can get orders of magnitude in savings

Local and hardware-specific optimisations

- not in this course-

What are we optimizing?

Time

Memory

Disk

Electricity

Compile time

Ease of use

Ease of deployment

Ease of development

Complexity basics

Much simplified, skipping formal derivation

```
while not is_sorted(xs):
    random.shuffle(xs)
```

$O(N\,N!)$

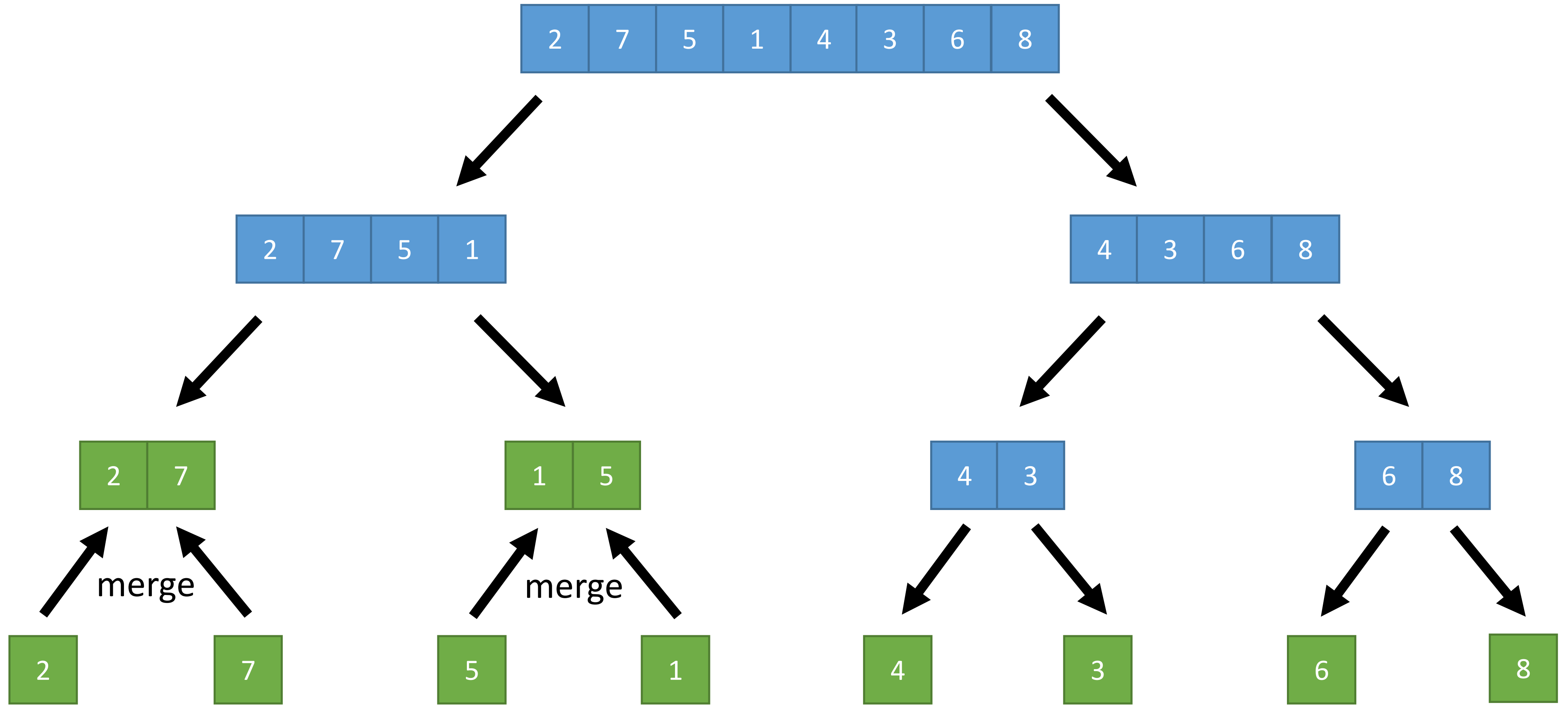
Scaling behaviour with size N of problem set:

$O(1)$ - constant time independent of N

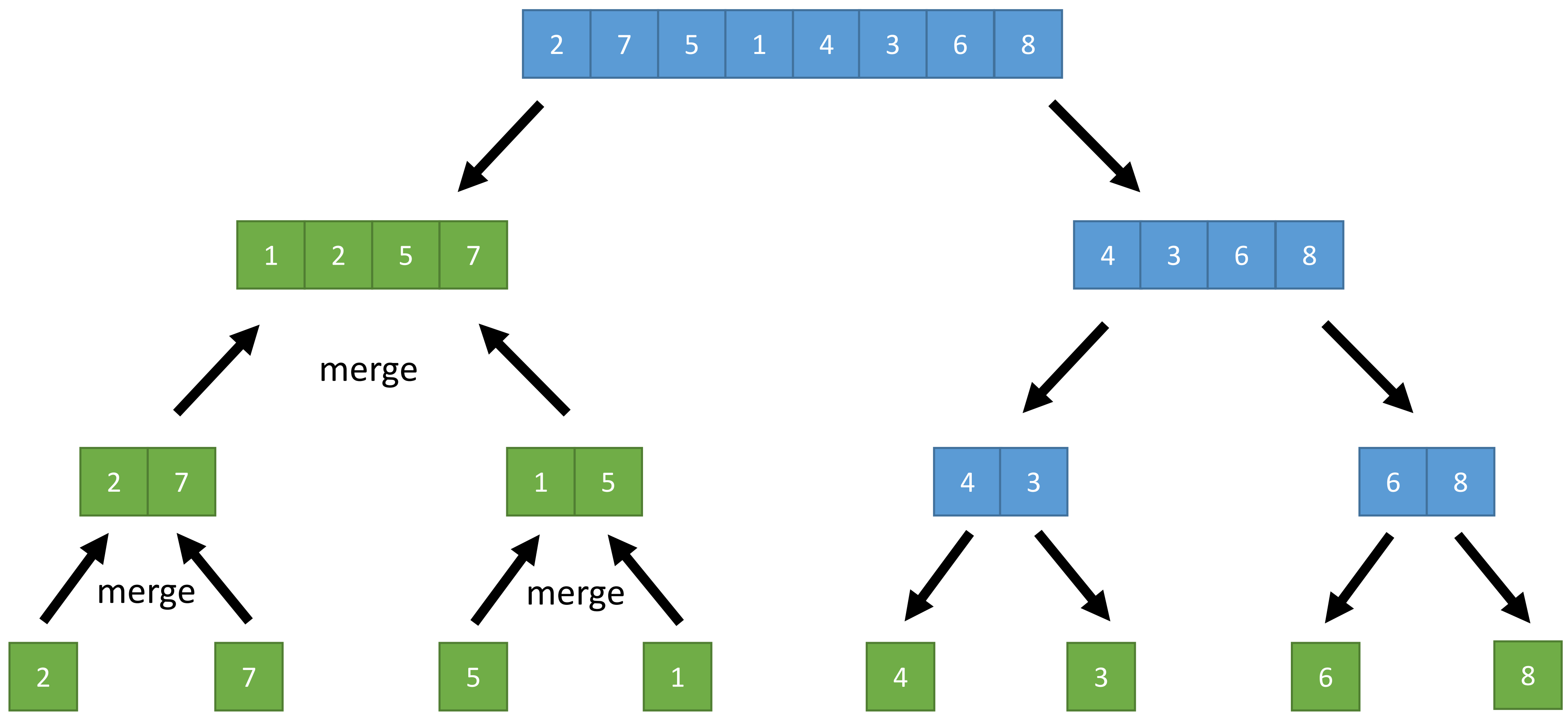
$O(N)$ - linear with N

$O(N^2)$ - quadratic in N

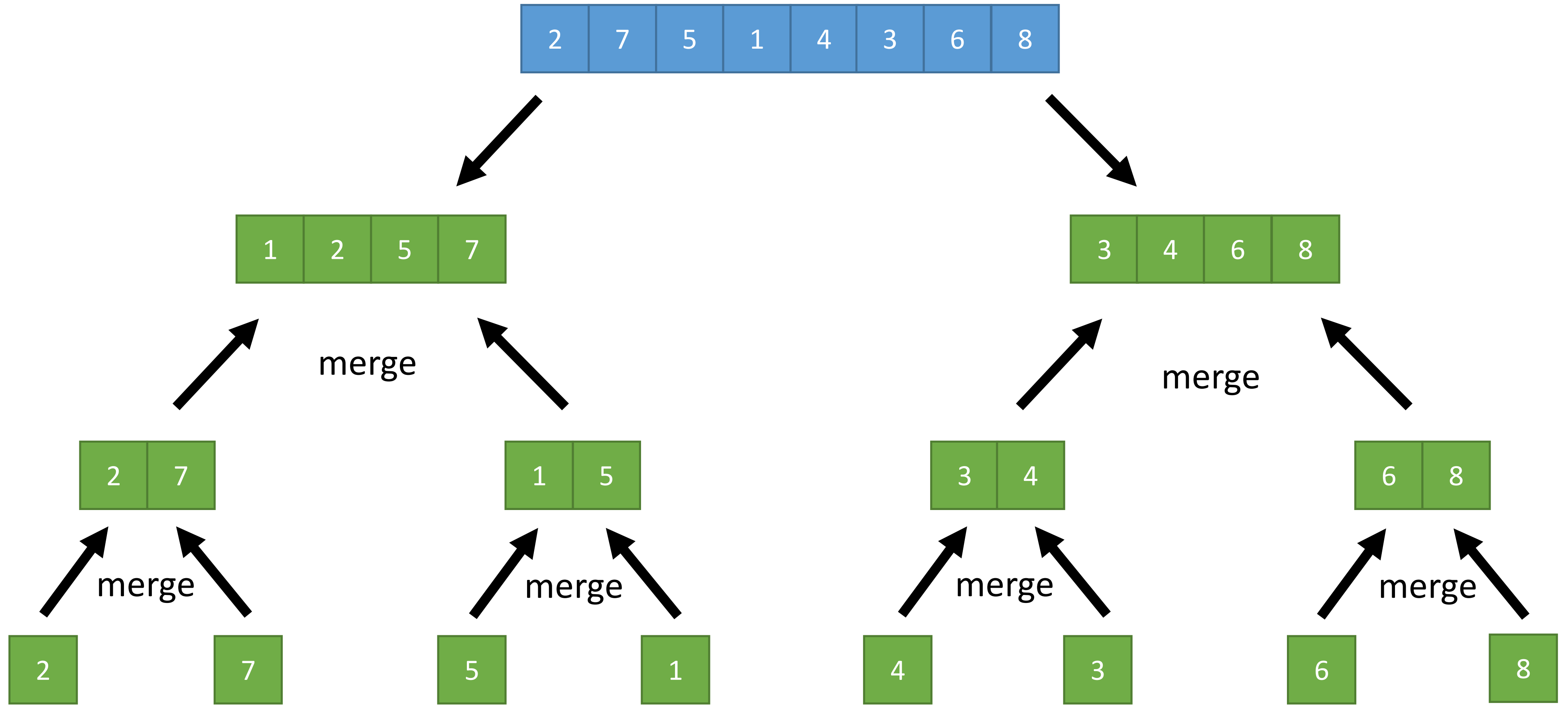
Merge Sort



Merge Sort



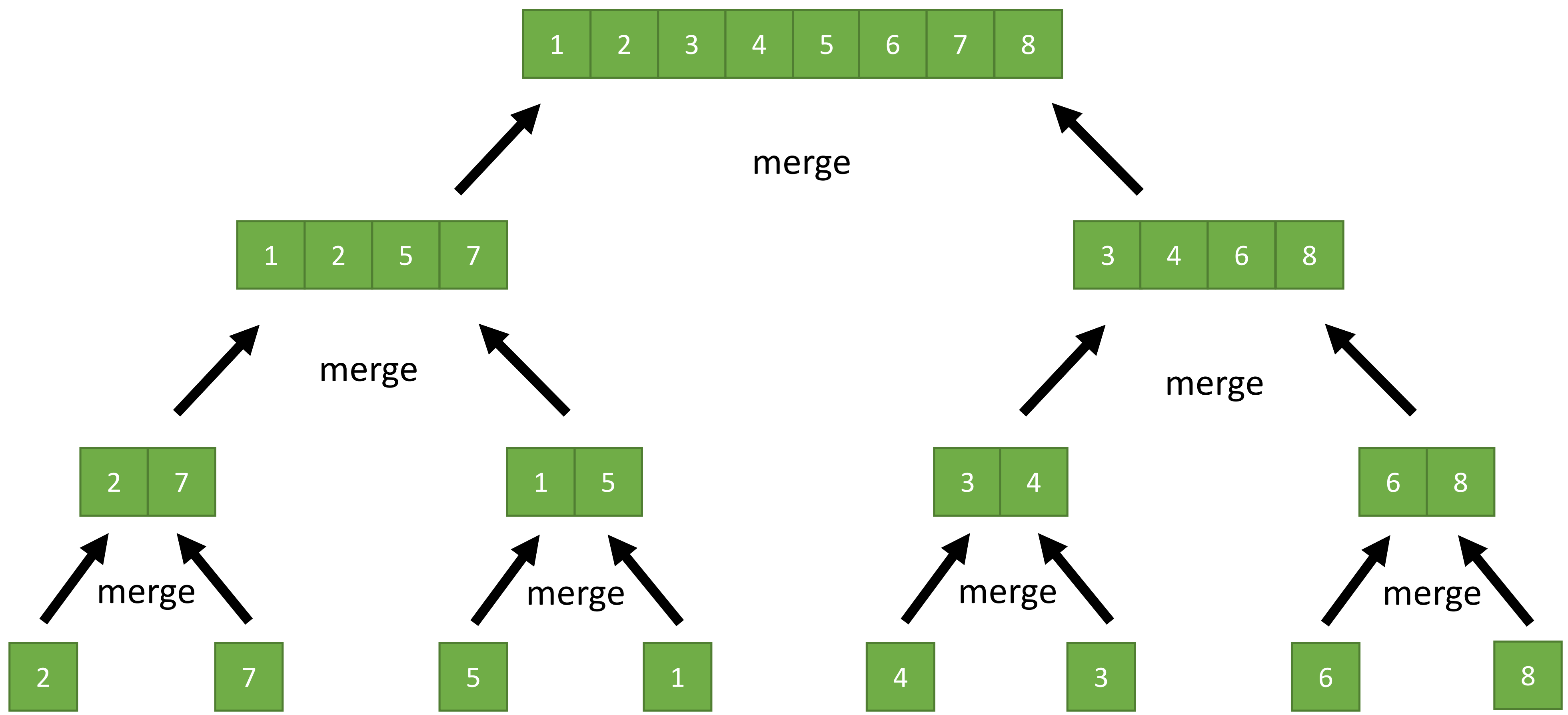
Merge Sort



2 7 5 1 4 3 6 8
1 2 5 7
3 4 6 8
merge
merge
2 7
1 5
3 4
6 8
merge
merge
merge
merge
2
7
5
1
4
3
6
8

Merge Sort

$O(N \log N)$



15 Sorting Algorithms in 6 Minutes
http://youtu.be/kPRA0W1kECg

Data structure complexity

Array
Vector

Linked list

Ordered map

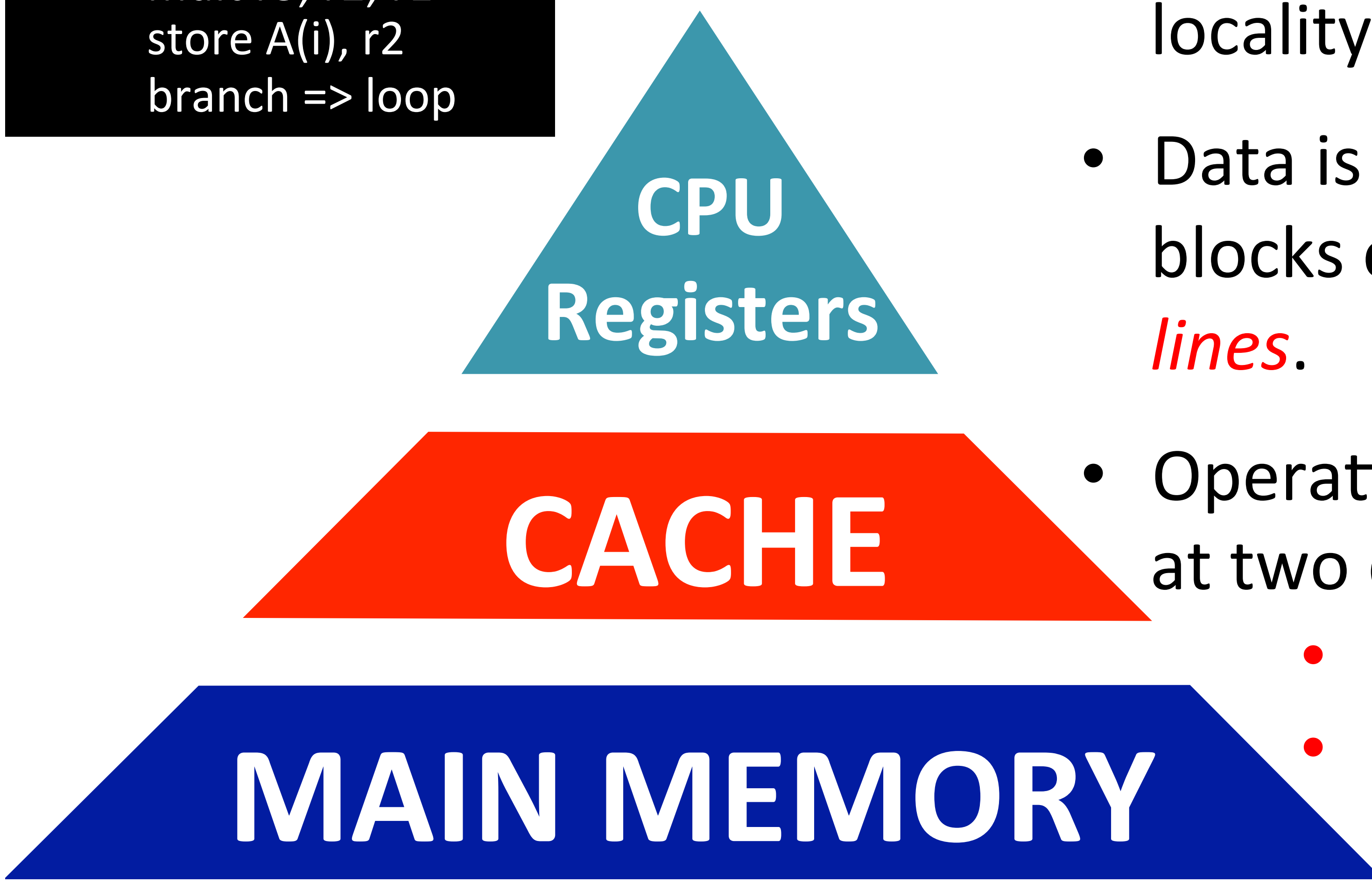
Hash table

http://bigocheatsheet.com/

Nicolai Josuttis, “The C++ Standard Library”

Cache Memory

```
Loop: load r1, A(i)
      load r2, s
      mult r3, r2, r1
      store A(i), r2
      branch => loop
```



- Designed for temporal/spatial locality
- Data is transferred to cache in blocks of fixed size, called *cache lines*.
- Operation of LOAD/STORE can lead at two different scenario:
  - *cache hit*
  - *cache miss*

L1 cache reference .... 0.5 ns

Branch mispredict .... 5 ns

L2 cache reference .... 7 ns

Mutex lock/unlock .... 25 ns

Main memory reference .... 100 ns

SSD random read .... 150,000 ns = 150 µs

Read 1 MB sequentially from memory ..... 250,000 ns = 250 µs

Read 1 MB sequentially from SSD ..... 1,000,000 ns = 1 ms

Disk seek .... 10,000,000 ns = 10 ms

Read 1 MB sequentially from disk .... 20,000,000 ns = 20 ms

Send packet EU->US->EU .... 150,000,000 ns = 150 ms

L1 cache reference	0.5 s
Branch mispredict	5 s
L2 cache reference	7 s
Mutex lock/unlock	25 s
Main memory reference	100 s
SSD random read	1.7 days
Read 1 MB sequentially from memory	2.9 days
Read 1 MB sequentially from SSD	11.6 days
Disk seek	16.5 weeks
Read 1 MB sequentially from disk	7.8 months
Send packet EU->US->EU	4.8 years

Optimization strategy

Don't optimize the whole code

Profile the code, find the bottlenecks
They may not always be where you thought they were

Break the problem down

Try to run the shortest possible test you can to get meaningful results
Isolate serial kernels

Keep a working version of the code!

Getting the wrong answer faster is not the goal.

Optimize on the architecture on which you intend to run

Optimizations for one architecture will not necessarily translate

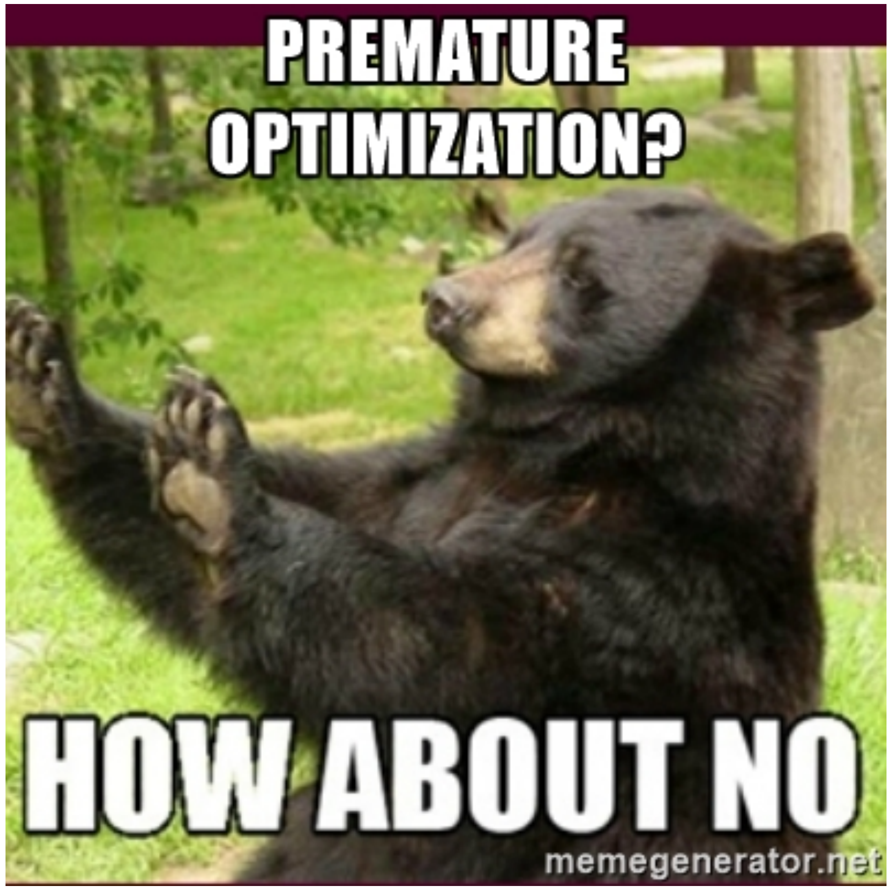
The compiler is your friend!

If you find yourself coding in machine language, you are doing to much.

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?



Python profiling options

time

```
from time import time
start = time()
somefunc(27)
end = time()
```

timeit

```
python -m timeit -s ‘import myfile as m;
x=27’ ‘m.somefunc(x)’
```

cProfile

```
import cProfile
cProfile.run(somefunc(27))
```

pyprof2calltree
qcachegrind

All are in the standard library

Link to compiled code

Try to stay with Python-only until performance becomes a problem. Numpy etc. make this possible

Interesting package to give just-in-time compilation on arbitrary code
https://numba.pydata.org/

Link to external code

ctypes for C (standard lib)

f2py for Fortran (part of numpy)

cython for C and C++ (on PyPI)

…