Australian National University

Files

C4

Java File IO

Streams

Standard IO

Buffering

What is a file?

A file is a collection of data on secondary storage (hard drive, USB key, network file server).

Data in a file is a sequence of bytes (integer $0 \le b \le 255$).

- The program reading a file must interpret the data (as text, image, sound, etc).
- Standard libraries provide support for interpreting data as text.

I/O streams

A **stream** is a standard abstraction used for files:

- A sequence of values are read.
- A sequence of values are written.

The stream reflects the sequential nature of file IO and the physical characteristics of the media on which files traditionally reside (e.g. tape or a spinning disk).

Other I/O (e.g., network, keyboard) is also typically accessed as streams.

Files C4

A
maxell
UR
90
POSITION NORMAL

TDK
DATA CARTRIDGE
DC-6250
Cartouche Numérique
FUJIFILM
DG4-150
DataLife

I/O in Java: Byte streams

The classes `java.io.InputStream` and `java.io.OutputStream` allow reading and writing bytes to and from streams.

- Subclasses: `FileInputStream` and `FileOutputStream` for files.
  - Open the stream (create stream object)
  - Read or write **`byte`**s from the stream
  - Wrap operations in a **`try`** clause
  - Use **`finally`** to close the streams

I/O in Java: Character streams

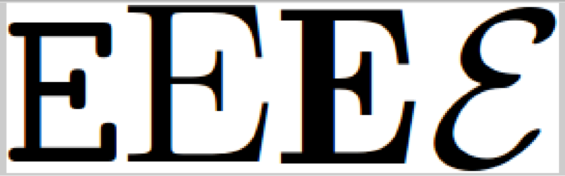
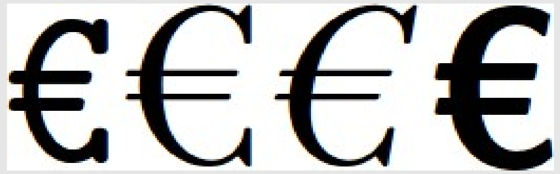
To read/write text files, use `java.io.Reader` and `java.io.Writer` which convert between bytes and characters according to a specified encoding.

- Subclasses: `InputStreamReader` and `OutputStreamWriter`
- Subclasses `FileReader` and `FileWriter` (shortcuts for wrapping a `FileInputStream` / `FileOutputStream` in a `InputStreamReader` / `OutputStreamWriter`).

Text encoding

Each character is assigned a number.

Unicode defines a unique number (“code point”) for > 120,000 characters (space for > 1 million).

Bytes $\xrightarrow{\text{Encoding (UTF-8)}}$	Code point $\xrightarrow{\text{Font}}$	Glyph
0100 0101 (69)	69	
1110 0010 (226) 1000 0010 (130) 1010 1100 (172)	8364	

Buffering I/O

In traditional storage media, accessing a specific byte (point in a file) is time consuming:

Disk: ~10ms   SSD: ~100μs   RAM: ~10ns   Register: ~1ns

But reading a consecutive “block” at one time is not much more so. Hence, buffering is used to absorb some of the overhead.

- `BufferedReader` and `BufferedWriter` can be wrapped around other reader/writer (e.g., `FileReader` and `FileWriter`) to buffer I/O.
- To flush the buffer, call `flush()`, or close the file.

Terminal I/O

Three standard I/O streams:

- standard input: (usually typed) input to the program
- standard output: normal printed program output
- standard error: program error messages (not buffered)
- Available in Java as `System.in`, and `System.out` and `System.err`.

```
byte b = (byte) System.in.read();
System.out.write(b);
System.out.flush();
System.err.write(b);
```

Australian National University

Computational Complexity

C5

Algorithm complexity

Big-O notation

Examples

Problem complexity

Algorithm complexity

The computational resources consumed by an algorithm:

- as a function of the size of it’s input (scaling behaviour);
- in the worst case (usually, but also average, amortised, etc).

What computational resources?

- Time (counting elementary operations)
- Memory
- Energy, communications, I/O, samples...

Algorithm complexity is important when (but only when) dealing with large problems, or problems solved very frequently.

Big-O notation

Suppose we have a problem of size *n* that takes *g(n)* time to execute in the average case.

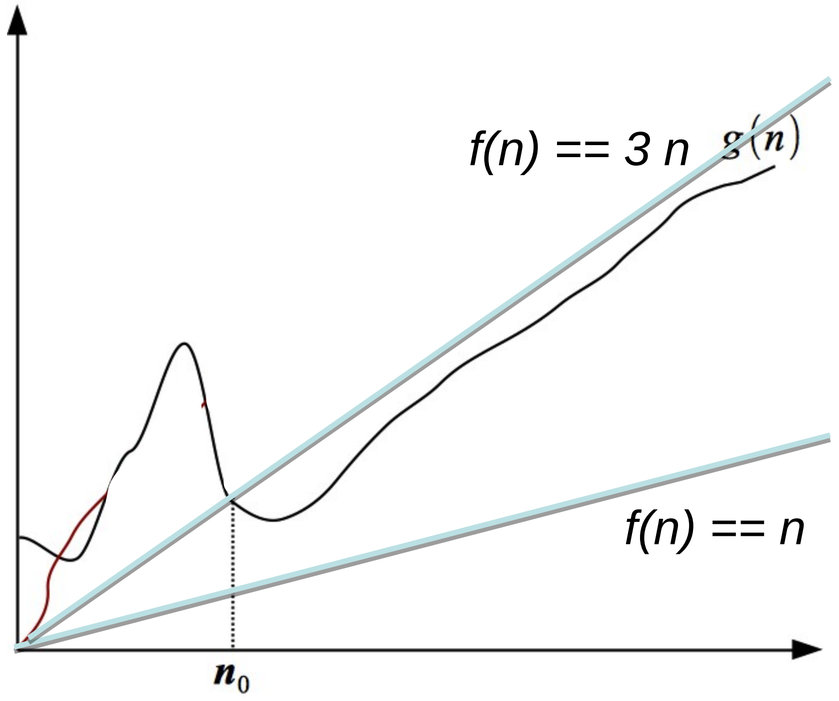
We say:

$g(\text{n}) \in O(f(n))$

if and only if there exists a constant $c > 0$

and a constant $n_0 > 0$ such that for all $n > n_0$:

$g(n) \leq \text{c} \times f(n)$



Time complexity

In analysis of algorithm time complexity, we are interested in the number of “elementary operations/statements” (not μs).

- Simple statements are constant time.
- Remember the factor c in O(f(n)).
- Beware: Library/subroutine calls can have arbitrary complexity.

Example

Find the greatest element ≤ x in an unsorted sequence of n elements. (For simplicity, assume some element ≤ x is in the sequence.)

Two approaches:

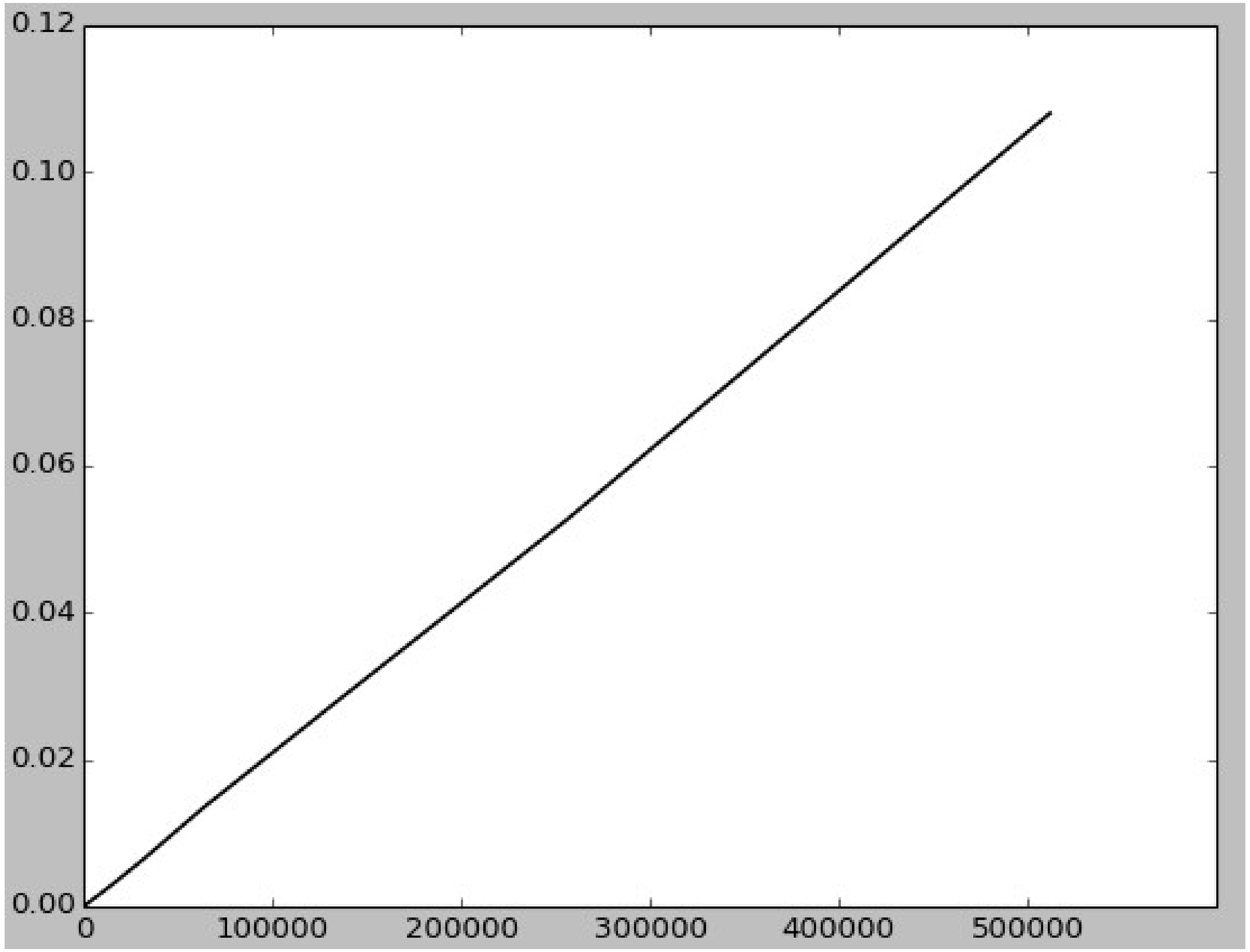
a) search the unsorted sequence; or

b) first sort the sequence, then search the sorted sequence.

```
int unsortedFind(int x, List<Integer> ulist) {
  Integer best = null;
  for (var e : ulist) {
    if (e == x) return e;
    if (e <= x) {
      if (best == null || e < best)
        best = e;
    }
  }
  return best;
}
```

Analysis

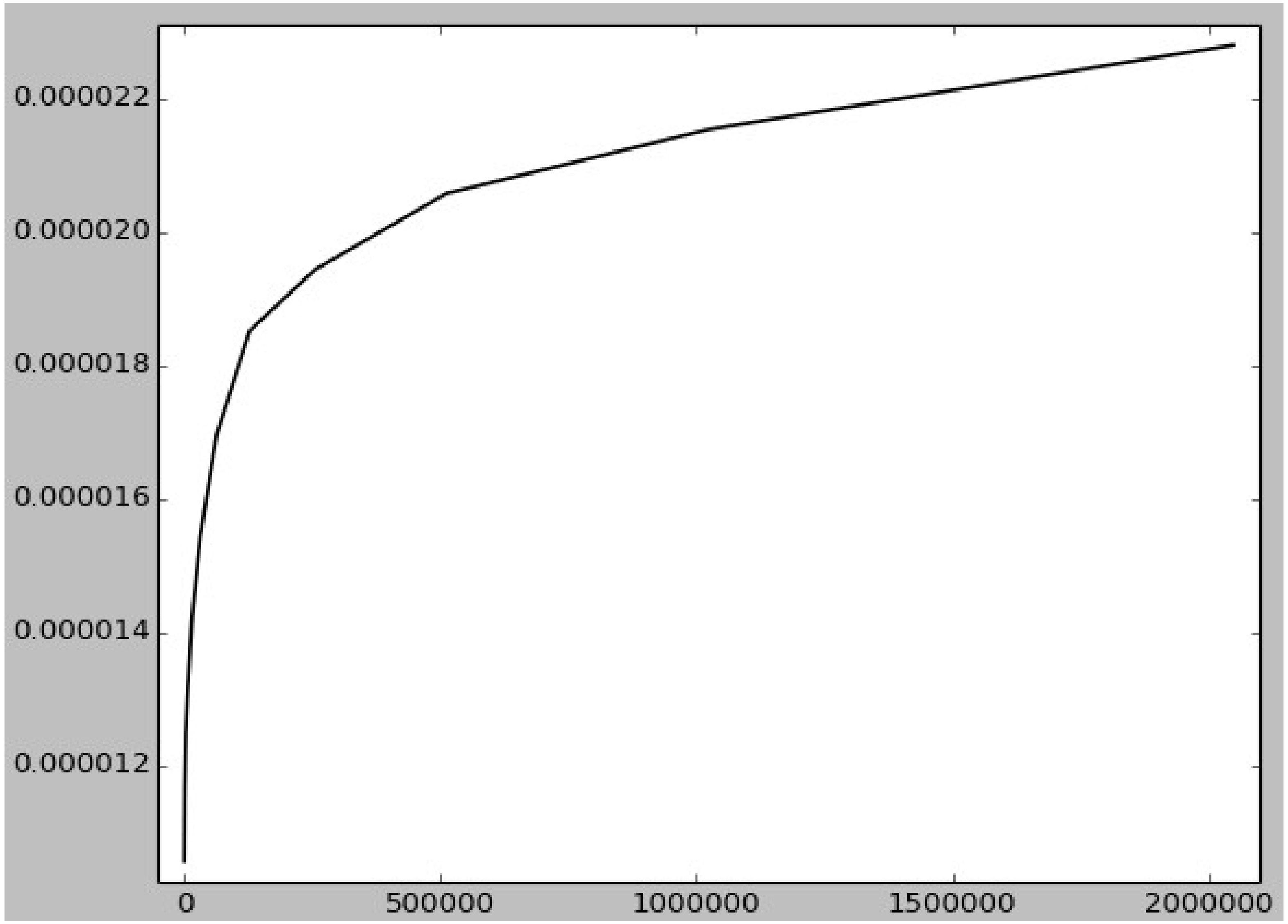
- Elementary operation: comparison.
- If we're lucky, ulist[0] = x.
- Worst case?
  - ulist = {0, 1, 2, ..., x – 1}
  - Compare each element with x and current value of best
- f(n) = 2n, so O(n)


0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
100000
200000
300000
400000
500000

```
int sortedFind(int x, List<Integer> slist) {
  if (slist.get(slist.size()) < x)
    return slist.get(slist.size());
  int lower = 0;
  int upper = slist.size();
  while (upper - lower > 1) {
    int mid = (lower + upper) / 2;
    if (slist.get(mid) <= x)
      lower = mid;
    else
      upper = mid;
  }
  return slist.get(lower);
}
```

Analysis

- Loop invariant: slist[lower] <= x and x < slist[upper].
- How many iterations of the loop?
- Initially, upper - lower = n − 1.
- The difference is halved in every iteration.
- Can halve it at most $\log_2(n)$ times before it becomes 1.
- $f(n) = \log_2(n) + 1$, so $O(\log(n))$.


0.000022
0.000020
0.000018
0.000016
0.000014
0.000012
0
500000
1000000
1500000
2000000

More examples

- Constant $O(1)$
  - evaluate a fixed expression; linked list insertion (given point reference).
- Logarithmic $O(\log(n))$
  - find an element in a balanced binary tree.
- Linear $O(n)$
  - find a given, or "best" (e.g., min/max) element in a list; any constant-time operation on n elements.
- $O(n \log(n))$
  - sort using mergesort
- Quadratic $O(n^2)$
  - compare n elements with each other pair-wise

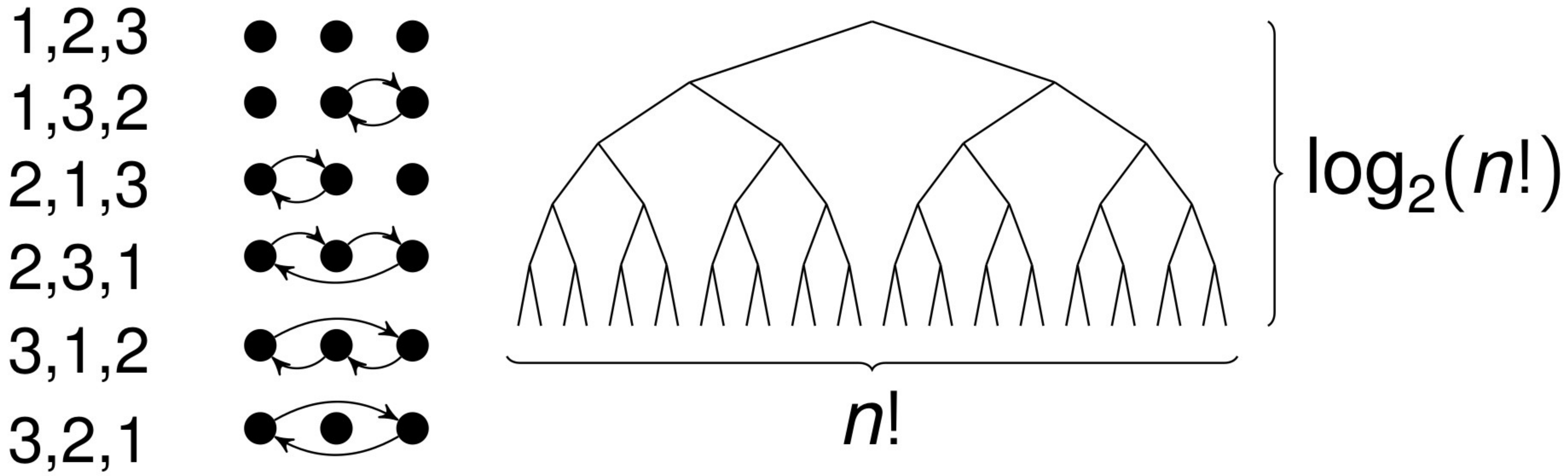
Problem complexity

The complexity of a problem is the resources (time, memory, etc) that *any* algorithm *must* use, in the worst case, to solve the problem, as a function of instance size.

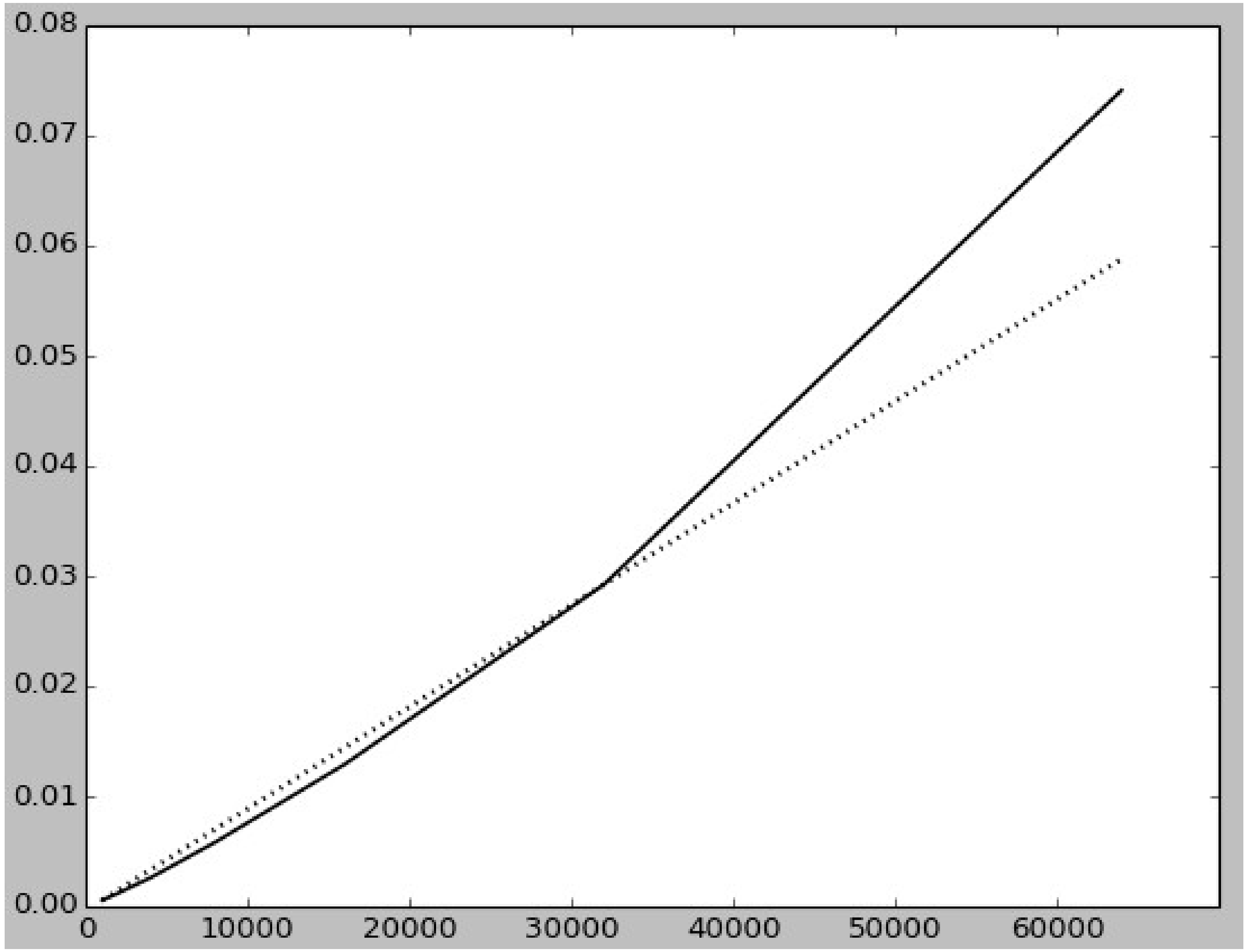
Hierarchy theorem: For any computable function f(n), there is a problem that requires time (memory) greater than f(n).

How fast can you sort?

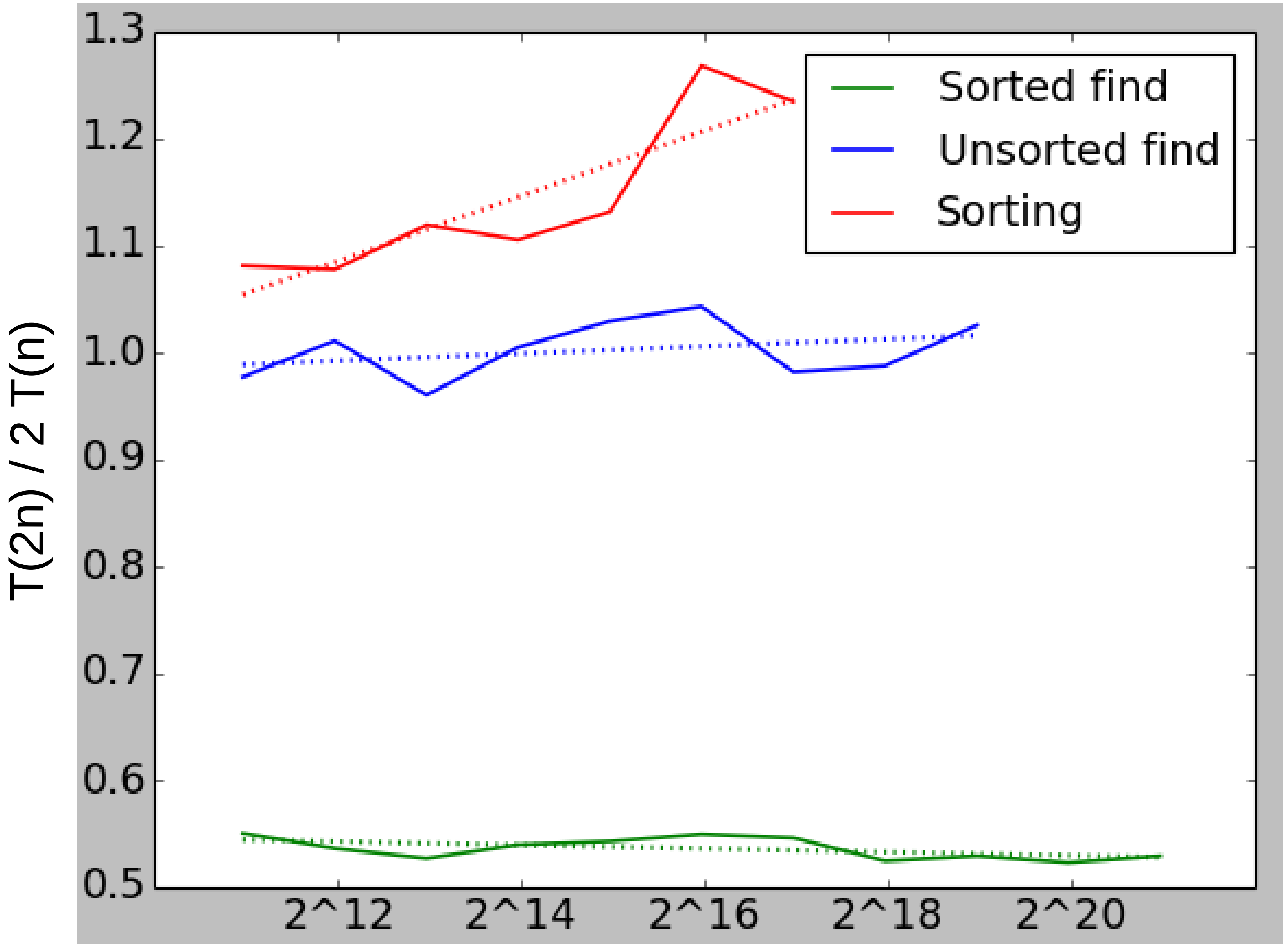
Any sorting algorithm that uses only pair-wise comparisons needs O(n log(n)) comparisons in the worst case.



$\log_2(n!) \geq n \log(n)$ for large enough n.


0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0
10000
20000
30000
40000
50000
60000

Rate of growth



Caution

“Premature optimization is the root of all evil in programming.”

(C.A.R. Hoare)

Scaling behaviour becomes important when (and only when) problems become large, or when they need to be solved very frequently.

Formal Grammars

C6

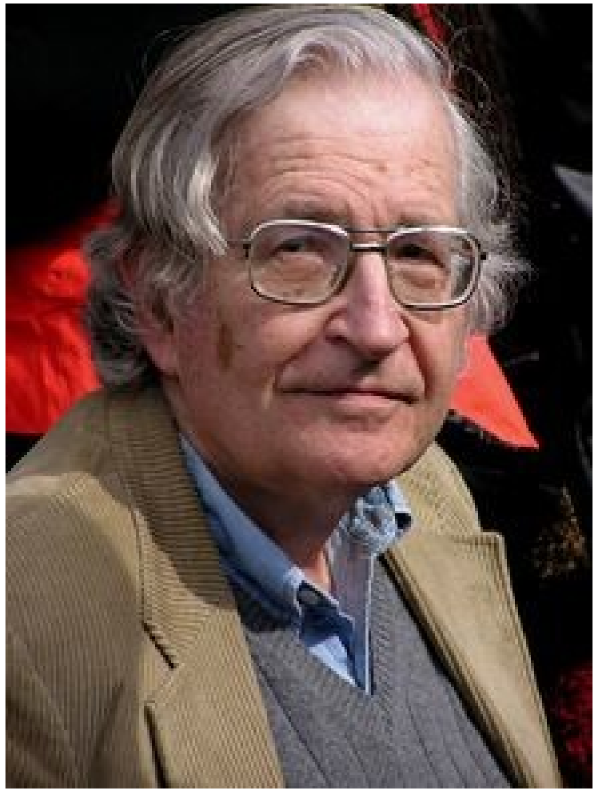
Grammars

EBNF

Formal grammars

Formal languages are distinguished from natural languages by their artificial construction (rather than natural emergence).

The syntax of a formal language is defined, typically using a formal grammar (but, in theory of computation, can also be by an accepting machine).


Duncan Rawlinson (Creative Commons)

Noam Chomsky

Generative grammars

- A set of terminal symbols (“words” of the language)
- A set of non-terminal symbols
- A set of production (rewrite) rules (symbol(s) => symbols)
  - A rule allows replacing an occurrence of the lhs within an expression with the rhs.
- A starting (non-terminal) symbol.

The language is the set of all terminal symbol-only strings that can be generated from the starting symbol using the production rules.

Extended Backus-Naur Form (EBNF)

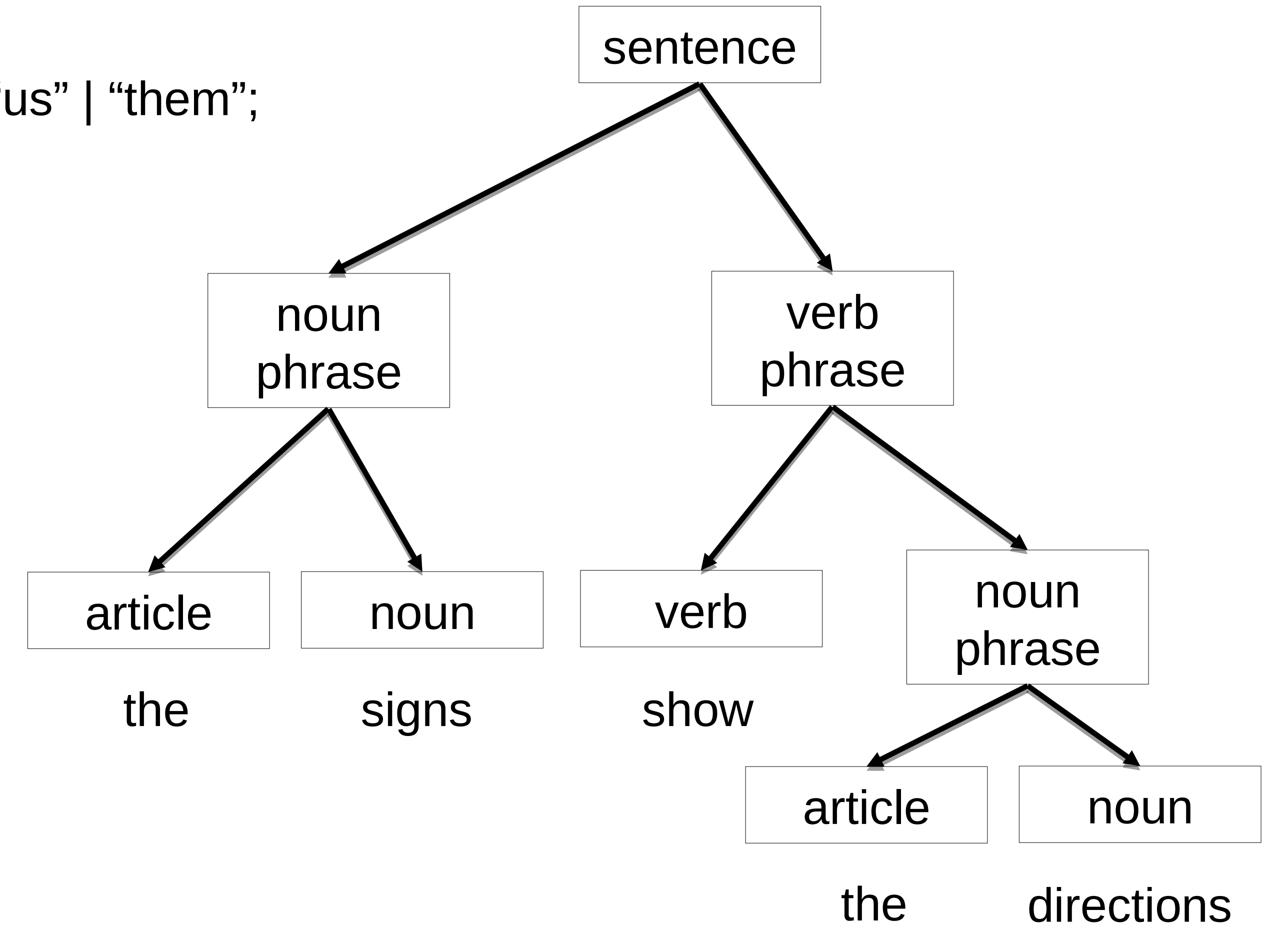
A “standard” way of representing a generative grammar, with some syntactic sugar. Syntax of EBNF production rules:

- ‘=’ defines a production rule
- Quotes (‘ ’ or “ ”) identify terminal strings
- ‘|’ identifies alternates (e.g. `‘1’ | ‘2’ | ‘3’`)
- ‘`{`’, ‘`}`’ expression that may occur zero or more times (e.g. `‘1’, { ‘0’ }`)
- ‘[’, ‘]’ expression that may occur zero or one time (e.g. `‘1’, [ ‘0’ ]`)
- ‘,’ identifies concatenation
- ‘-’ identifies exceptions
- ‘(’, ‘)’ identify groups
- ‘;’ terminates a rule

Grammars C6

sentence = noun phrase, verb phrase, [noun phrase];
noun phrase = pronoun | ([article], [adjective], noun | noun phrase);
verb phrase = verb, [noun phrase];
article = “a” | “an” | “the”
pronoun = “I” | “you” | “he” | “she” | “it” | “we” | “they” | “us” | “them”;
noun = ? all nouns in the dictionary ?
verb = ? all verbs in the dictionary ?
adjective = ? all adjectives in the dictionary ?

Examples:
“The signs show the directions”
“I sleep”
“The blue a yellow horse talk the we”
“Colourless green ideas sleep a furious calm”



Simple EBNF grammars

- Natural numbers

```
natural = ‘0’ | (nzdigit, { digit }) ;
nzdigit = ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’ ;
digit = ‘0’ | nzdigit ;
```

- Integers

```
integer = ‘0’ | ([‘-’], nzdigit, { digit }) ;
```

- Decimal numbers

```
real = ([‘-’], natural, [(‘.’ { digit }, nzdigit)]) – ‘-0’ ;
```

- Balanced parentheses

```
lists = ‘(‘ lists ‘)’ | { lists } | ‘’ ;
```