Week 1

Getting Started

Organisation

Lecturer Dr Oliver Kullmann

Lectures 2 hours lectures:

Tuesday	11-12	Zoom 910 0959 8294
Wednesday	10-11	Zoom 916 2877 9761

Tutorial Given by Stewart Powell:

Thursday	13-14	Zoom 964 9498 9004

Labs **One** of the sessions on Friday (allocation to be sent out); all in Computational Foundry CF204 (Linux Lab).

THIS (FIRST) WEEK NO LABS.

Also no tutorial this week by Stewart, however the Thursday slot is used this week for background material (this week only).

Course home page on Canvas

LINK

with

- messages
- general information
- up-to-date versions of the slides (which serve as the script)
- lab sheets
- course works (as online tests)
- VIDEOS:
  1. recordings of the lectures (they take a bit to show up)
  2. specific topics, in addition to the lecture and the slides of the week — perhaps best viewed after the lectures of the week completed.

Videos this week (Week01):

- explaining InsertionSort
- analysis of algorithms in a nutshell
- Two videos in preparation of next week: some words on logarithms and Big-Oh.

Assessment

- There will be two pieces of coursework, each worth 10%.
- Labs are an essential part of the course. Successful participations is worth 10%.
- The written examination counts 70%, it will take place in January.

Lectures

- The material is organised in Weeks 01 - 10, presented in the two lectures Tuesday + Wednesday.
- The tutorial-lecture (Thursday) practises the material of the week.
- Labs (Friday) practise the material, in various form (quizzes and small tasks).

References

There are very many algorithms textbooks in print.
In this module, we will roughly follow (and refer to as **CLRS**)

> **Introduction to Algorithms** *(Third Edition)*
> *by Cormen, Leiserson, Rivest and Stein,*
> *The MIT Press, 2009.*

`http://en.wikipedia.org/wiki/Introduction_to_Algorithms`

Reading from CLRS for week 1

- Chapter 1
- Chapter 2, Sections 2.1, 2.2

However the material of this module is self-contained, and the book needs only to be considered for deeper studies.

Computational problems

Computational Problem: A general description, from *inputs* to *outputs*.

Problem Instance: A concrete input ("problem" in such a context is always the **general** problem).

Computational problems

Computational Problem: A general description, from *inputs* to *outputs*.

Problem Instance: A concrete input ("problem" in such a context is always the **general** problem).

The main example for the first three weeks is SORTING:

Problem: Sorting (numbers).
Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots a'_n \rangle$ of the input such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Example instance:

$$\begin{array}{rl} \text{Input} & \langle 10, -15, 3, 3, 70, -17, -30 \rangle \\ \text{Output} & \langle -30, -17, -15, 3, 3, 10, 70 \rangle \end{array}$$

Note there are no algorithms here — we speak here about the **specification** of the (computational) problem.

Algorithms and programs

Algorithm: a well-defined computational procedure that from inputs computes outputs.

- An algorithms takes an input, and computes an output.
- In other words, an algorithm solves a problem instance (in a specific way!).
- Important here: the algorithm MUST work on ALL inputs / ALL problem instances.

There will always be many different algorithms for any given computational problem.

In this module we'll see for example four sorting algorithms (starting with InsertionSort this week).

Algorithms and programs (cont.)

- A **program** is a particular *implementation* of some algorithm.
- A program is not the same as an algorithm.
- There are many different implementations of any given algorithm.
- For real-world high performance, tuning the implementation is important (and non-trivial), but is not considered in this module.

Expressing algorithms

We express algorithms in whatever way is the clearest and most concise.

English is sometimes the best way.

When issues of control need to be made perfectly clear, we often use pseudocode.

And we will see some Java examples, to also improve your programming skills a bit.

Insertion Sort

Insertion Sort is a good algorithm for sorting a small number of elements.

It works the way you might sort a hand of playing cards:

Take the next card,
and insert it into the already sorted cards.

- The Wikipedia page is fine to use.
- And I've created a video (sorting 8 coins).

As with all algorithms, there are different levels of understanding:

From the rough idea to the details of the code.

The generic direction is to start with the rough idea, and move from the abstract to the concrete ("top-down").

But also the other direction ("bottom-up") can be helpful.

A simple example (showing the "two hands"):

1. To be sorted: 5 3 1 6 8 4 2 7

A simple example (showing the "two hands"):

1. To be sorted: 5 3 1 6 8 4 2 7
2. EMPTY — 5 3 1 6 8 4 2 7

A simple example (showing the "two hands"):

1. To be sorted: 5 3 1 6 8 4 2 7
2. EMPTY — 5 3 1 6 8 4 2 7
3. 5 — 3 1 6 8 4 2 7

A simple example (showing the "two hands"):

1. To be sorted: 5 3 1 6 8 4 2 7
2. EMPTY — 5 3 1 6 8 4 2 7
3. 5 — 3 1 6 8 4 2 7
4. 3 5 — 1 6 8 4 2 7

A simple example (showing the "two hands"):

1. To be sorted: 5 3 1 6 8 4 2 7
2. EMPTY — 5 3 1 6 8 4 2 7
3. 5 — 3 1 6 8 4 2 7
4. 3 5 — 1 6 8 4 2 7
5. 1 3 5 — 6 8 4 2 7

A simple example (showing the "two hands"):

1. To be sorted: 5 3 1 6 8 4 2 7
2. EMPTY — 5 3 1 6 8 4 2 7
3. 5 — 3 1 6 8 4 2 7
4. 3 5 — 1 6 8 4 2 7
5. 1 3 5 — 6 8 4 2 7
6. 1 3 5 6 — 8 4 2 7

A simple example (showing the "two hands"):

1. To be sorted: 5 3 1 6 8 4 2 7
2. EMPTY — 5 3 1 6 8 4 2 7
3. 5 — 3 1 6 8 4 2 7
4. 3 5 — 1 6 8 4 2 7
5. 1 3 5 — 6 8 4 2 7
6. 1 3 5 6 — 8 4 2 7
7. 1 3 5 6 8 — 4 2 7

A simple example (showing the "two hands"):

1. To be sorted: 5 3 1 6 8 4 2 7
2. EMPTY — 5 3 1 6 8 4 2 7
3. 5 — 3 1 6 8 4 2 7
4. 3 5 — 1 6 8 4 2 7
5. 1 3 5 — 6 8 4 2 7
6. 1 3 5 6 — 8 4 2 7
7. 1 3 5 6 8 — 4 2 7
8. 1 3 4 5 6 8 — 2 7

A simple example (showing the "two hands"):

1. To be sorted: 5 3 1 6 8 4 2 7
2. EMPTY — 5 3 1 6 8 4 2 7
3. 5 — 3 1 6 8 4 2 7
4. 3 5 — 1 6 8 4 2 7
5. 1 3 5 — 6 8 4 2 7
6. 1 3 5 6 — 8 4 2 7
7. 1 3 5 6 8 — 4 2 7
8. 1 3 4 5 6 8 — 2 7
9. 1 2 3 4 5 6 8 — 7

A simple example (showing the "two hands"):

1. To be sorted: 5 3 1 6 8 4 2 7
2. EMPTY — 5 3 1 6 8 4 2 7
3. 5 — 3 1 6 8 4 2 7
4. 3 5 — 1 6 8 4 2 7
5. 1 3 5 — 6 8 4 2 7
6. 1 3 5 6 — 8 4 2 7
7. 1 3 5 6 8 — 4 2 7
8. 1 3 4 5 6 8 — 2 7
9. 1 2 3 4 5 6 8 — 7
10. 1 2 3 4 5 6 7 8 — EMPTY.

How to actually do the insertion?

We are using an

ARRAY,

and thus can't insert something in the middle.

First idea: pairwise swapping.

Better:

1. Take the element x to be inserted OUT.
2. Shift the already sorted elements one to the right,
3. until the gap opens at the correct position,
4. then put x back IN.

How to actually do the insertion? (cont.)

For example:

1. 1 3 5 6 8 — 4 2 7
2. That means the array currently is

$$1, 3, 5, 6, 8, \underline{4}, 2, 7$$

   underlined the element $x = 4$ to be inserted.
3. Taking it OUT, and moving, and finally putting it IN:

$$1, 3, 5, 6, 8, ?, 2, 7$$
$$1, 3, 5, 6, ?, 8, 2, 7$$
$$1, 3, 5, ?, 6, 8, 2, 7$$
$$1, 3, ?, 5, 6, 8, 2, 7$$
$$1, 3, 4, 5, 6, 8, 2, 7.$$

Code

The input is an

1. array A of length $\mathtt{n}$,
2. where the elements of the array are $A[1], \ldots, A[n]$.

So for our pseudo-code we use, as in the book, 1-based arrays (if not said otherwise).

INSERTION-SORT(A)

```
1  for j = 2 to n
2      key = A[j]
3      // Insert A[j] into sorted sequence A[1..j−1].
4      i = j−1
5      while i > 0 and A[i] > key
6          A[i+1] = A[i]
7          i = i−1
8      A[i+1] = key
```

Algorithm analysis

We want to predict the resources that the algorithm requires.

Algorithm analysis

We want to predict the resources that the algorithm requires.

Typically, we are interested in

- runtime
- memory
- number of basic operations, such as:
  - arithmetic operations (eg, for multiplying matrices)
  - bit operations (eg, for multiplying integers)
  - comparisons (eg, for sorting and searching)

Every algorithm analysis states what to analyse; typically:

BASIC OPERATIONS.

How do we analyse running time?

The time taken by an algorithm depends on the input.

- Sorting 1000 numbers takes longer than sorting 3 numbers.
- A given sorting algorithm may even take differing amounts of time on two inputs of the same size.
- For example, we'll see that insertion sort takes less time to sort n elements when they are already sorted than when they are in reverse sorted order.
- Nevertheless, **input size** is the main parameter.

How do we analyse running time?

The time taken by an algorithm depends on the input.

- Sorting 1000 numbers takes longer than sorting 3 numbers.
- A given sorting algorithm may even take differing amounts of time on two inputs of the same size.
- For example, we'll see that insertion sort takes less time to sort n elements when they are already sorted than when they are in reverse sorted order.
- Nevertheless, **input size** is the main parameter.

The input size depends on the problem studied.

- Usually the number of items in the input. Like the size n of the array being sorted.
- But could be something else. If multiplying two integers, could be the total number of bits in the two integers.

Preparation: Summing up

$$1 + 2 + \cdots + (n-1) + n = \sum_{i=1}^{n} i = ?$$

Good approximation:

$$1 + \cdots + n \quad \approx \quad \underbrace{n/2 + \cdots + n/2}_{n \text{ times}}$$

$$= n \cdot n/2 = \tfrac{1}{2} n^2$$

Analysing Insertion-Sort

Input size: length n of array

Counting lines of pseudo-code.

INSERTION-SORT(A)

		cost	*repetitions*
1	**for** $j = 2$ **to** n	c_1	n
2	$key = A[j]$	c_2	$n-1$
3	// Insert $A[j]$ into sorted sequence $A[1..j-1]$.	0	$n-1$
4	$i = j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j = t_2 + \cdots + t_n$
6	$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	$A[i+1] = key$	c_8	$n-1$

where:

- t_j = number of times the **while** -loop test in line 5 is executed in the jth iteration of the **for** -loop.
- $\sum_{j=2}^{n} t_j = t_2 + t_3 + \cdots + t_n$.

Applying the exact analysis

Best possible situation: $t_j = 1$ for all j, i.e., when A is already sorted.

Runtime: $(c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$.

Worst possible situation: $t_j = j$ for all j, i.e., when A is sorted in *reverse* order.

Runtime: $\left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8)$.

Average situation: $t_j = \frac{j}{2}$ for all j, i.e., every permutation is equally likely, so expected value of t_j is $\frac{j}{2}$.

Runtime: $\left(\frac{c_5}{4} + \frac{c_6}{4} + \frac{c_7}{4}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{4} - \frac{3c_6}{4} - \frac{3c_7}{4} + c_8\right) n - \left(c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)$.

Remarks

Types of Analysis:

- exact analysis (rarely achieved)
- best-case (helpful for special cases)
- worst-case (the DEFAULT)
- average-case (hard to do and to interprete).

Remarks

Types of Analysis:

- exact analysis (rarely achieved)
- best-case (helpful for special cases)
- worst-case (the DEFAULT)
- average-case (hard to do and to interprete).

Worst-case versus average-case analysis: We usually concentrate on finding the worst-case running time. Average case is often as bad as the worst case.

Remarks

Types of Analysis:

- exact analysis (rarely achieved)
- best-case (helpful for special cases)
- worst-case (the DEFAULT)
- average-case (hard to do and to interprete).

Worst-case versus average-case analysis: We usually concentrate on finding the worst-case running time. Average case is often as bad as the worst case.

Order of growth is another abstraction to ease analysis and focus on the important features.
Will be discussed next week.

Mathematical Functions

We use standard mathematical functions, like exponentiation b^x and its inverse $\log_b x$:

$$\log_b a \text{ is the number } x \text{ such that } b^x = a.$$

We shall usually work with binary logarithms: $\lg x = \log_2 x$.

Some Useful Identities

Logarithms translate multiplication into addition:

$$\log_b(xy) = \log_b x + \log_b y$$

Logarithms translate exponentiation into multiplication:

$$\log_b(x^y) = y \log_b x$$

Different logarithms relate by a constant factor:

$$\frac{\log_b x}{\log_c x} = \log_b c$$

Mathematical Functions (cont.)

We shall use in the first few weeks the floor and the ceiling functions, $\lfloor x \rfloor$ and $\lceil x \rceil$:

$\lfloor x \rfloor$ is the largest integer $\leq x$, e.g., $\lfloor 5.3 \rfloor = 5$
$\lceil x \rceil$ is the smallest integer $\geq x$, e.g., $\lceil 5.3 \rceil = 6$

There is also the summation notation:

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + a_3 + \cdots + a_n \ .$$

Binary powers

$0 \to 1$
2
4
8
16
32
64
128
$8 \to 256$
512
$10 \to 1024$
2048
4096
8192
16384
32768
$16 \to 65536$
$17 \to 131072$
262144
524288
$20 \to 1048576$
2097152
4194304
8388608
16777216
33554432
67108864
134217728
268435456
536870912
$30 \to 1073741824$
$31 \to 2147483648$
$32 \to 4294967296$

On the left we have the binary powers 2^n for $0 \leq n \leq 32$.

$2^8 = 256$
$2^{16} = 65536$ (≈ 66 kilo)
$2^{32} \approx 4.3 \cdot 10^9$ (4.3 giga)
$2^{64} \approx 1.8 \cdot 10^{19}$ (18 exa).

Some examples for binary logarithms

$$\lg(10^3) \approx 10 \text{ (since } 2^{10} = 1024)$$

1. $\lg(128 \cdot 10^3) = \lg(128) + \lg(10^3) = 7 + \lg(10^3) \approx 17$
2. $\lg(10^6) = \lg(10^3 \cdot 10^3) = \lg(10^3) + \lg(10^3) \approx 20$
3. $\lg(7 \cdot 10^6) = \lg(7) + \lg(10^6) \approx 2.8 + 20 = 22.8$
4. $\lg(10^9) = \lg(10^3) + \lg(10^6) \approx 10 + 20 = 30$
5. $\lg(10^{10}) = \lg(10) + \lg(10^9) \approx 3.3 + 30 = 33.3$

Reminder: numbers

How to call larger numbers (standard English):

1. Thousand: $10^3 = 1,000$ — one kilobyte.
2. Million: $10^6 = 1,000,000$ — one megabyte.
3. Billion: $10^9 = 1,000,000,000$ — one gigabyte.
4. Trillion: $10^{12} = 1,000,000,000,000$ — one terabyte.
5. The prefixes in the metric system are:
   kilo, mega, giga, tera, peta, exa.

The names "billion, trillion" belong to the short scale, while other languages use "Milliard" resp. "Billion" for 10^9 and 10^{12} (long scale).

The corresponding small numbers:

1. 10^{-12}: pico (p)
2. 10^{-9}: nano (n)
3. 10^{-6}: micro (μ)
4. 10^{-3}: milli (m)

How many seconds in a year?

Number of seconds in

- a minute: 60
- an hour: $60 * 60 = 3,600$
- a day: $24 * 3600 = 72000 + 14400 = 86,400$
- a year: $365 * 86400 = 31,536,000 \approx 31.5 \cdot 10^6$.

So there are roughly 31.5 million seconds in a year. A related number is $\sqrt{1000} = 31.62277\ldots$, which can be roughly computed as follows:

$$\begin{aligned} 2^{10} &= 1024 \\ \sqrt{1000} &\approx (2^{10})^{1/2} = 2^{\frac{1}{2}\cdot 10} = 2^5 = 32. \end{aligned}$$

How many seconds in a year? (cont.)

So one billion-square (i.e., $(10^9)^2 = 10^{18}$) operations take roughly 32 years (more precisely, 31.7 years), since:

1. With one billion operations per second, one billion-square operations need precisely one billion seconds.
2. Above we learned $31.6^2 \approx 1000 = 10^3$.
3. So $31.6^2 \cdot 10^6 \approx 10^9$.
4. Roughly $31.6^2 \approx 31.5 \cdot 31.7$, and above we learned $31.5 \cdot 10^6$ is roughly the number of seconds per year.
5. So one billion seconds are roughly 31.7 years.

Two revealing tables

Time to solve a problem instance of size n using a $T(n)$-time algorithm, where one basic operations needs 1 ns $= 10^{-9}$s:

T(n)	n=10	n=20	n=50	n=100	n=500	n=1000
n	10 ns	20 ns	50 ns	0.1 μs	0.5 μs	1 μs
$n \lg n$	33 ns	86 ns	282 ns	664 ns	4.5 μs	10 μs
n^2	100 ns	400 ns	2.5 μs	10 μs	250 μs	1 ms
2^n	1 μs	1 ms	13 d	10^{13} yr	10^{134} yr	10^{284} yr
$n!$	4 ms	10^2 yr	10^{48} yr	10^{141} yr	10^{1117} yr	10^{2551} yr

Two revealing tables

Time to solve a problem instance of size n using a $T(n)$-time algorithm, where one basic operations needs 1 ns $= 10^{-9}$s:

T(n)	n=10	n=20	n=50	n=100	n=500	n=1000
n	10 ns	20 ns	50 ns	0.1 μs	0.5 μs	1 μs
$n \lg n$	33 ns	86 ns	282 ns	664 ns	4.5 μs	10 μs
n^2	100 ns	400 ns	2.5 μs	10 μs	250 μs	1 ms
2^n	1 μs	1 ms	13 d	10^{13} yr	10^{134} yr	10^{284} yr
$n!$	4 ms	10^2 yr	10^{48} yr	10^{141} yr	10^{1117} yr	10^{2551} yr

Largest problem instance solvable in 1 second:

$T(n)$		2× faster machine	10^3× faster machine
n	10^9	$2 \cdot 10^9$	10^{12}
$n \lg n$	$3.96 \cdot 10^7$	$7.64 \cdot 10^7$	$2.88 \cdot 10^{10}$
n^2	$31,623$	$44,721$	10^6
2^n	30	31	40

Using InsertionSort I

Can we sort one billion numbers with insertion sort? (How long would it take?)

Using InsertionSort I

Can we sort one billion numbers with insertion sort? (How long would it take?)

1. $n = 10^9$.

Using InsertionSort I

Can we sort one billion numbers with insertion sort? (How long would it take?)

1. $n = 10^9$.
2. Roughly 10^9 operations per second.

Using InsertionSort I

Can we sort one billion numbers with insertion sort? (How long would it take?)

1. $n = 10^9$.
2. Roughly 10^9 operations per second.
3. Rough complexity of insertion sort: n^2.

Using InsertionSort I

Can we sort one billion numbers with insertion sort? (How long would it take?)

1. $n = 10^9$.
2. Roughly 10^9 operations per second.
3. Rough complexity of insertion sort: n^2.
4. So roughly 10^9 seconds needed.

Using InsertionSort I

Can we sort one billion numbers with insertion sort? (How long would it take?)

1. $n = 10^9$.
2. Roughly 10^9 operations per second.
3. Rough complexity of insertion sort: n^2.
4. So roughly 10^9 seconds needed.
5. That's roughly 32 years (as seen above).

Using InsertionSort I

Can we sort one billion numbers with insertion sort? (How long would it take?)

1. $n = 10^9$.
2. Roughly 10^9 operations per second.
3. Rough complexity of insertion sort: n^2.
4. So roughly 10^9 seconds needed.
5. That's roughly 32 years (as seen above).

Using InsertionSort I

Can we sort one billion numbers with insertion sort? (How long would it take?)

1. $n = 10^9$.
2. Roughly 10^9 operations per second.
3. Rough complexity of insertion sort: n^2.
4. So roughly 10^9 seconds needed.
5. That's roughly 32 years (as seen above).

Can we do 32 years?

Using InsertionSort I

Can we sort one billion numbers with insertion sort? (How long would it take?)

1. $n = 10^9$.
2. Roughly 10^9 operations per second.
3. Rough complexity of insertion sort: n^2.
4. So roughly 10^9 seconds needed.
5. That's roughly 32 years (as seen above).

Can we do 32 years?

1. That's somewhat more than 10000 days.

Using InsertionSort I

Can we sort one billion numbers with insertion sort? (How long would it take?)

1. $n = 10^9$.
2. Roughly 10^9 operations per second.
3. Rough complexity of insertion sort: n^2.
4. So roughly 10^9 seconds needed.
5. That's roughly 32 years (as seen above).

Can we do 32 years?

1. That's somewhat more than 10000 days.
2. So with a cluster of 10,000 processors we can do it in a day.

Using InsertionSort I

Can we sort one billion numbers with insertion sort? (How long would it take?)

1. $n = 10^9$.
2. Roughly 10^9 operations per second.
3. Rough complexity of insertion sort: n^2.
4. So roughly 10^9 seconds needed.
5. That's roughly 32 years (as seen above).

Can we do 32 years?

1. That's somewhat more than 10000 days.
2. So with a cluster of 10,000 processors we can do it in a day.

Using InsertionSort I

Can we sort one billion numbers with insertion sort? (How long would it take?)

1. $n = 10^9$.
2. Roughly 10^9 operations per second.
3. Rough complexity of insertion sort: n^2.
4. So roughly 10^9 seconds needed.
5. That's roughly 32 years (as seen above).

Can we do 32 years?

1. That's somewhat more than 10000 days.
2. So with a cluster of 10,000 processors we can do it in a day.

Really?

Using InsertionSort II

- Algorithms like InsertionSort are inherently serial.
- Often, to achieve a good speed-up with just 4-10 cores is not easy.

Parallelisation is not part of the module, but QuickSort (and all algorithms following the *Divide-and-Conquer* paradigm) have good parallelisability.

Back to sizes: Which n is possible for InsertionSort within a second?

Using InsertionSort II

- Algorithms like InsertionSort are inherently serial.
- Often, to achieve a good speed-up with just 4-10 cores is not easy.

Parallelisation is not part of the module, but QuickSort (and all algorithms following the *Divide-and-Conquer* paradigm) have good parallelisability.

Back to sizes: Which n is possible for InsertionSort within a second?

$$n = \sqrt{10^9}.$$

We had already seen, in the second table: $n \approx 31,623$. Here's the approximate computation:

$$\sqrt{10^9} = \sqrt{10^3 \cdot 10^6} = \sqrt{1000} \cdot \sqrt{10^6} \approx 31.6 \cdot 1000 = 31,600.$$

Some orders of magnitude of the universe

Number of galaxies comparable to our galaxy (the Milky Way) in the observable universe:

Some orders of magnitude of the universe

Number of galaxies comparable to our galaxy (the Milky Way) in the observable universe: 100 billion ($= 10^{11}$).

Some orders of magnitude of the universe

Number of galaxies comparable to our galaxy (the Milky Way) in the observable universe: 100 billion ($= 10^{11}$).

So more than 10 galaxies per person on earth currently.

Number of stars on average in any of these larger galaxies:

Some orders of magnitude of the universe

Number of galaxies comparable to our galaxy (the Milky Way) in the observable universe: 100 billion ($= 10^{11}$).

So more than 10 galaxies per person on earth currently.

Number of stars on average in any of these larger galaxies: Also 100 billion.

Estimation on the number of stars in the observable universe: $3 \cdot 10^{23}$:

- Compare Avogadro constant: $6.022 \cdot 10^{23}$.
- $10^{23} = 10^2 \cdot (10^3)^7 \approx 10^2 \cdot (2^{10})^7 = 100 \cdot 2^{70} = 6400 \cdot 2^{64}$.

Time is roughly the order of the square root of the number of stars:

- Full orbit of the solar system around the galactic core: 240 million years.
- Age of the universe: 13.8 billion years.

Running insertion sort

Write down the following sequence

$$12, 14, 5, 4, 1, 14, 5, 3, 13, 10$$

and sort it by insertion-sort, counting the comparisons and the assignments

— as usual in this context,
only considering the "real objects".

Running insertion sort (cont.)

Showing number of comparisons and assignments for each insertion step, together with the *next* element to be inserted:

$12, \underline{14}, 5, 4, 1, 14, 5, 3, 13, 10$

1. $12, 14, \underline{5}, 4, 1, 14, 5, 3, 13, 10 : (1, 2)$
2. $5, 12, 14, \underline{4}, 1, 14, 5, 3, 13, 10 : (2, 4)$
3. $4, 5, 12, 14, \underline{1}, 14, 5, 3, 13, 10 : (3, 5)$
4. $1, 4, 5, 12, 14, \underline{14}, 5, 3, 13, 10 : (4, 6)$
5. $1, 4, 5, 12, 14, 14, \underline{5}, 3, 13, 10 : (1, 2)$
6. $1, 4, 5, 5, 12, 14, 14, \underline{3}, 13, 10 : (4, 5)$
7. $1, 3, 4, 5, 5, 12, 14, 14, \underline{13}, 10 : (7, 8)$
8. $1, 3, 4, 5, 5, 12, 13, 14, 14, \underline{10} : (3, 4)$
9. $1, 3, 4, 5, 5, 10, 12, 13, 14, 14 : (5, 6)$

$\sum$: $(30, 42)$

Number of binary digits

Consider a natural number $n \geq 0$:

Develop a formula
for the number $b(n) \geq 1$
of binary digits of n.

Hint: use lg.

Number of binary digits (cont.)

$$b(n) = \begin{cases} 1 & \text{if } n = 0 \\ \lfloor \lg(n) \rfloor + 1 & \text{if } n \geq 1 \end{cases}$$

Here are the values of $b(n)$ for $n \in \{0, \ldots, 16\}$:

$$1, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 5$$

Sums

Develop a formula for

$$f(n) := \sum_{i=1}^{n} i.$$

Start by creating a table of values.

Sums (cont.)

The list of values of $f(n)$ for $n = 0, \ldots, 10$:

$$0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55.$$

The differences between these values are

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10.$$

- Since these differences grow, the original function is *not linear* (not something like $a \cdot n + b$).
- However this list of differences *is linear*, and so the original function is *quadratic*, that is, $f(n) = a \cdot n^2 + b \cdot n + c$.

Playing around yields

$$f(n) = \sum_{i=1}^{n} i = \frac{1}{2} n(n+1) = \frac{1}{2}(n^2 + n) = \frac{1}{2} n^2 + \frac{1}{2} n.$$

Sums (cont.)

And indeed the argument made before for $f(n) \approx \frac{1}{2}n^2$ can be made precise here: The average of the first summand 1 and the last summand n is $\frac{n+1}{2}$, and we have n summands, thus $f(n) = n \cdot \frac{n+1}{2}$.

Taking the average works here, since the summands grow linearly:

It is the *nature* of linear growth,
that we can replace in a sum each summand
by their average
(more precisely, by the arithmetic mean).

Other algorithms (homework)

Can you imagine other algorithms for sorting?

- In what sense could they improve insertion-sort?
- How good could they be?