

Algorithms and Datastructures

Lecture 1

Manfred Jaeger



AALBORG UNIVERSITET

Logistics

Teachers

Manfred Jaeger `jaeger@cs.aau.dk`

TAs:

Robert Waury

Schedule

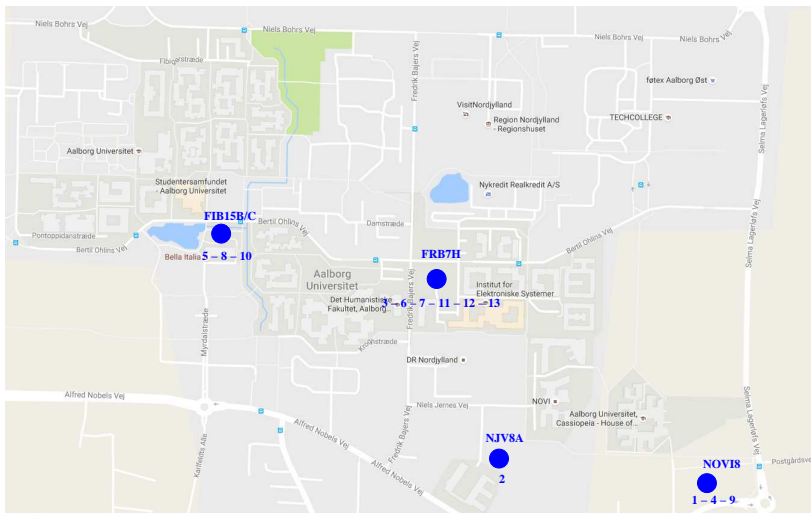
Mondays/Thursdays:

Exercises: 12:30-14:15

Lectures: 14:30-16:15

Fridays:

Self-Study (5 sessions) 8:00-12:00



See also map.aau.dk

Time Management

Lectures	13×2	26
Exercises	13×2	26
Self Study	6×4	24
Reading	13×2	26
Exam Preparation		40
Other		8
		<hr/> 150

Self Study

- ▶ Extended exercises with exam-like problems
- ▶ Limited supervision
- ▶ Each group can submit one written solution (pdf scan) via Moodle upload
- ▶ Feedback provided for submitted solutions

Exam

- ▶ Written exam
- ▶ Mix of multiple-choice and “open format” questions

Pre-requisites

- ▶ Basic experience with, and interest in computer programming
- ▶ Basic mathematical notation and graph theory

Connection with Discrete Mathematics

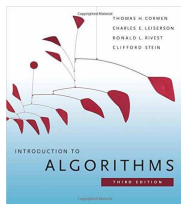
The DM course has already touched upon many topics we will deal with in this course:

- ▶ big- O notation
- ▶ loop invariants
- ▶ graphs
- ▶ Dijkstra's algorithm
- ▶ Divide and conquer algorithms
- ▶ ...

In the future ...

- ▶ Advanced Algorithms (DAT6/SW6 optional)
- ▶ Computability and Complexity (DAT5/SW5)

Textbook

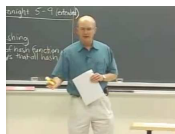


T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein:
Introduction to Algorithms, 3rd Edition

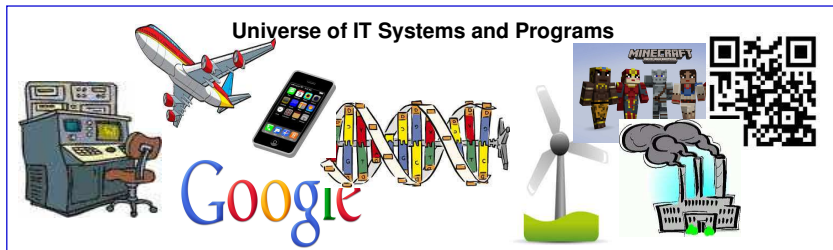
Video Lectures

Video lectures featuring Erik Demaine and Charles Leiserson at MIT:

<http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/>



Introduction



Fundamental Computational Problems

Sorting... Searching... Shortest Paths ...

Algorithms and Datastructures

Insertion Sort... Quicksort... Dijkstra... Hashtables...

Tools

- ▶ Standard problems and solutions

Analysis

- ▶ Correctness and Complexity

Design Principles

- ▶ Divide & Conquer, Dynamic Programming

Original

Pseudo-code

PROPOSITION II.

P R O B L E M.

Two Numbers not prime to each other being given: to find their greatest common Measure.

LET the two given Numbers not prime to each other be AB, CD, and let CD be the lesser; it is requir'd to find their greatest common Measure.

Now if CD measures AB, since it
 A B also measures itself, it shall be a com-
 C D mon Measure of AB, CD; and it is
 manifest it is the greatest, because no
 Number greater than CD can measure CD.

But if CD does not measure AB, continually taking
 away the less CD from the greater
 A . . . E B AB, there will at last remain some
 C . . F D Number that will measure the pre-
 G -- cedent one. Unity will not remain
 at last. Because if it should, AB,

CD are prime to each other, which is contrary to the
 Supposition; therefore there will remain a Number that
 will measure the precedent one. Now CD measuring
 AB, let AE remain, less than itself; AE measuring CD,
 let CF remain, less than itself; and let CF measure AE.
 Therefore because CF measures AE, and AE measures
 DF; CF will also measure DF. But it measures itself
 likewise; whence it shall measure the whole CD. But
 CD measures BE. Consequently likewise CF measures
 BE. But it also measures EA: whence it will measure
 the whole BA. It also measures CD: wherefore CF
 measures AB, CD. And accordingly CF is a common
 Measure of AB, CD. I say also it is the greatest. For if
 CF be not the greatest common Measure of AB, CD,
 let some Number greater than CF, measure AB, CD.
 Suppose this to be G. Then since G measures CD, and

input : Integers a, b

output: The greatest common divisor of a
 and b

EUCLID(a, b)

- 1 **if** $b == 0$ **then**
- 2 **return** a
- 3 **else**
- 4 **return** EUCLID($b, a \bmod b$)



Muḥammad ibn Mūsā al-Khwārizmī (780-850, Baghdad)

Author of: *Al-Kitāb al-mukhtaṣar fī ḥisāb al-jabr wa-l-muqābala*
(The Compendious Book on Calculation by Completion and Balancing)

Origin of the words **algorithm** and **algebra**

How to solve the equation $\frac{10-x}{x} = 4$:

Third Problem.

I have divided ten into two parts. I have afterwards divided the one by the other, and the quotient was four.†

Computation: Suppose one of the two parts to be thing, the other ten minus thing. Then you divide ten minus thing by thing, in order that four may be obtained. You know that if you multiply the quotient by the divisor, the sum which was divided is restored.

In the present question the quotient is four and the divisor is thing. Multiply, therefore, four by thing; the result is four things, which are equal to the sum to be divided, which was ten minus thing. You now reduce it by thing, which you add to the four things. Then we have five things equal to ten; therefore one thing is equal to two, and this is one of the two portions. This question refers you to one of the six cases, namely, “roots equal to numbers.”

(Engl. translation by F. Rosen, 1831)

Fundamentals

input : Integers a, b

output: The greatest common divisor
of a and b

EUCLID(a, b)

```
1 if  $b == 0$  then  
2   return  $a$   
3 else  
4   return EUCLID( $b, a \bmod b$ )
```

Pseudo-code representation of an algorithm:

- ▶ uses (imperative) programming-language style constructs and structure
- ▶ omits some technicalities (type declarations, brackets, constructors, ...)
- ▶ use of natural language or mathematical expressions
- ▶ can (and should) also contain comments (marked as such)
 - ▶ can include: input/output specification
- ▶ typically: executing one line of pseudo-code costs one unit computation time; exception: (recursive) algorithm calls.

To analyze the runtime of an algorithm, we need an underlying model of computation. We assume the **RAM (random-access machine)** model:

- ▶ Sequential computation, no parallelism
- ▶ The following operations take one time unit:
 - ▶ reading/writing the value of a variable from/to memory
 - ▶ performing a simple arithmetic operation (addition, multiplication, exponentiation,...)
 - ▶ testing a simple Boolean condition

A specification of a **finite, deterministic, terminating** procedure that computes an *output* for any possible *input* from a set of *possible inputs*.

An algorithm solves a *computational problem*

Computational Problem

Specification of

- ▶ a set of *possible inputs* (the *input/problem instances*)
- ▶ a *function* that maps each possible input to a unique *output*

Example: Sorting Integer Lists

Possible inputs: Lists of integers

Function: For problem instance $I = (i_1, i_2, \dots, i_N)$ return a list I' that contains the integers i_1, i_2, \dots, i_N in order of increasing values.

Sets

Sets are equal if they contain the same elements, e.g.:

$$\{1, 3, 2, 1\} = \{1, 2, 3\}$$

Multisets

Multisets (a.k.a. *bags*) are sets of elements together with their *count* (or *multiplicity*). We use $[\dots]$ to denote multisets:

$$[1, 3, 2, 1] = [1, 1, 2, 3] \neq [1, 2, 3]$$

Sorting problem reformulated

Possible inputs: Lists of integers

Function: For problem instance $I = (i_1, i_2, \dots, i_N)$ return a list $I' = (i'_1, i'_2, \dots, i'_N)$ such that $[i_1, i_2, \dots, i_N] = [i'_1, i'_2, \dots, i'_N]$ and $i'_k \leq i'_{k+1}$ for all $k = 1, \dots, N-1$.

Finite: the description of the algorithm has finite size.

Not an algorithm:

input : An array I of integers

output: An array containing the elements of I in ascending order

INFINITEFLIST(I)

```
1 if  $I = ()$  then
2   return ()
3 if  $I = (0)$  then
4   return (0)
5 if  $I = (1)$  then
6   return (1)
7 if  $I = (0, 0)$  then
8   return (0, 0)
9 if  $I = (0, 1)$  then
10  return (0, 1)
11 if  $I = (1, 0)$  then
12  return (0, 1)
```

...

```
7348644 if  $I = (14, 21, 7, 45, 13, 2)$  then
        return (2, 7, 13, 14, 21, 45)
```

...

Deterministic: At any point in the execution of the algorithm, the next step and its outcome is clearly and unambiguously defined.

Not an algorithm (in the narrow sense ...):

input : An array I of integers

output: An array containing the elements of I in ascending order

RANDOMQS(I)

- 1 Randomly select an element $i \in I$
- 2 Divide I into the sets $I_{<i}$ containing the elements $< i$, and $I_{\geq i}$ containing the elements $\geq i$
- 3 **return** (RANDOMQS($I_{<i}$), RANDOMQS($I_{\geq i}$))

 not an algorithm in the narrow, conservative sense, but a *randomized algorithm*.


Terminating: for any possible input, the algorithm terminates after a finite number of steps.

input : An array I of integers

output: An array containing the elements of I in ascending order

ALMOSTBUBBLESORT(I)

```
1 repeat  
2   continue = false  
3   for  $i=1 \dots I.length-1$  do  
4     if  $I[i] \geq I[i+1]$  then  
5       swap  $I[i]$  and  $I[i+1]$   
6     continue = true  
7 until continue = false
```

 It can be difficult to find out whether a proposed “algorithm” is terminating (very difficult indeed – stay tuned for more in *Computability and Complexity!*).

input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

while $i > 0$ and $I[i] > key$ **do**

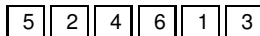
$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I

key=



input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

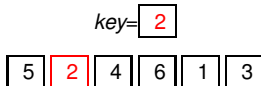
while $i > 0$ and $I[i] > key$ **do**

$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I



input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

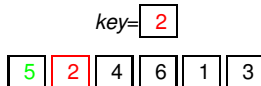
while $i > 0$ and $I[i] > key$ **do**

$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I



input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

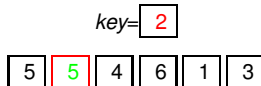
while $i > 0$ and $I[i] > key$ **do**

$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I



input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

while $i > 0$ and $I[i] > key$ **do**

$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I

key=

2	5	4	6	1	3
---	---	---	---	---	---

input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

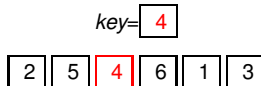
while $i > 0$ and $I[i] > key$ **do**

$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I



input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

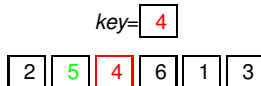
while $i > 0$ and $I[i] > key$ **do**

$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I



input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

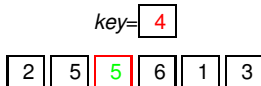
while $i > 0$ and $I[i] > key$ **do**

$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I



input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

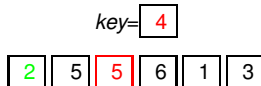
while $i > 0$ and $I[i] > key$ **do**

$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I



input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

while $i > 0$ and $I[i] > key$ **do**

$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I

key=



input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

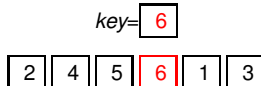
while $i > 0$ and $I[i] > key$ **do**

$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I



input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

while $i > 0$ and $I[i] > key$ **do**

$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I



input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

while $i > 0$ and $I[i] > key$ **do**

$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I

key=



input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

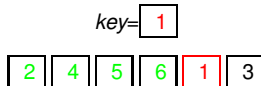
while $i > 0$ and $I[i] > key$ **do**

$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I



input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

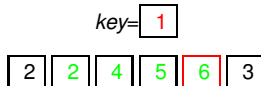
while $i > 0$ and $I[i] > key$ **do**

$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I



input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

while $i > 0$ and $I[i] > key$ **do**

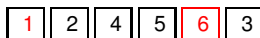
$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I

key=



input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

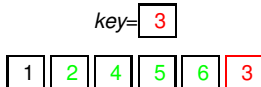
while $i > 0$ and $I[i] > key$ **do**

$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I



input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

while $i > 0$ and $I[i] > key$ **do**

$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I

key=



input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

while $i > 0$ and $I[i] > key$ **do**

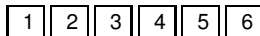
$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I

$key =$



input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

while $i > 0$ and $I[i] > key$ **do**

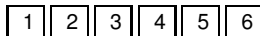
$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I

key=



Finite: ✓

input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

while $i > 0$ and $I[i] > key$ **do**

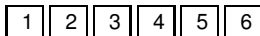
$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I

key=



Finite: ✓

Deterministic: ✓

input : An array I of integers

output: An array containing the elements of I in ascending order

INSERTIONSORT(I)

for $j = 2..I.length$ **do**

$key = I[j]$

$i = j - 1$

while $i > 0$ and $I[i] > key$ **do**

$I[i + 1] = I[i]$

$i = i - 1$

$I[i + 1] = key$

return I

key=

1	2	3	4	5	6
---	---	---	---	---	---

Finite: ✓

Deterministic: ✓

Terminating: ✓

The Algorithm Datastructure Nexus

Datastructures organize data in such a way that basic data retrieval and data manipulation operations required by specific *algorithms* are supported in an efficient manner.

Example: Stack

- ▶ Data: collection of objects
- ▶ Operations that are efficiently supported:
 - ▶ *push*: add another object to the stack
 - ▶ *pop*: remove and return the object most recently added

Other operations may be supported. E.g. in C#: `System.Collections.Stack` supports `Contains(Object o)`, but:

This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is `Count`. (msdn.microsoft.com)

Given

- ▶ a computational problem
- ▶ an algorithm

we are interested in the key questions:

Correctness Does the algorithm solve the computational problem (i.e., produce the correct output for all possible inputs)

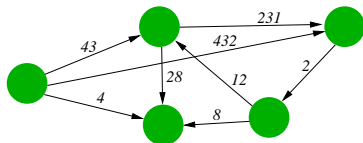
Complexity Does the algorithm solve the problem *efficiently*, i.e. using as little time and space resources as possible

Main aspect of complexity is:

Time complexity

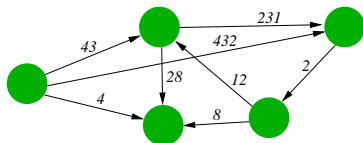
How does the *execution time* increase as a function of the *size of the input*?

Example: Weighted Graph



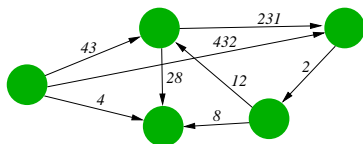
- ▶ A: Size = 5
- ▶ B: Size = 8
- ▶ C: Size = 15
- ▶ D: Size = 41

Example: Weighted Graph



- ▶ Size = 5 (Number of nodes)?
- ▶ Size = 8 (Number of edges)?
- ▶ Size = $1 + 1 + 1 + 2 + 2 + 2 + 3 + 3 = 15$
(total number of digits needed to represent the weights)?
- ▶ Size = $2 + 3 + 4 + 4 + 5 + 6 + 8 + 9 = 41$
(total number of bits needed to represent the weights)?

Example: Weighted Graph



- ▶ Size = 5 (Number of nodes)?
- ▶ Size = 8 (Number of edges)?
- ▶ Size = $1 + 1 + 1 + 2 + 2 + 2 + 3 + 3 = 15$
(total number of digits needed to represent the weights)?
- ▶ Size = $2 + 3 + 4 + 4 + 5 + 6 + 8 + 9 = 41$
(total number of bits needed to represent the weights)?

Example: Integer Array

(12, 3, 4367843, 54, 299)

- ▶ Size = 5 (length of array) ?
- ▶ Size = $4 + 2 + 23 + 6 + 9 = 44$ (total number of bits needed to represent integer components)?


(Number of bits needed for integer i : $\lfloor \log_2(i) \rfloor + 1$)

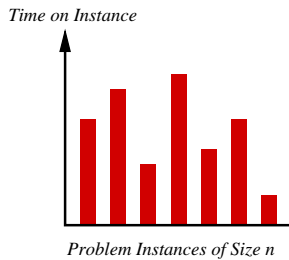
- ▶ Theory of computation: input size measured in number of bits needed to fully encode the input
- ▶ Algorithmics: higher-level description of inputs; (usually) not taking size of numbers into account (in line with RAM computation model)

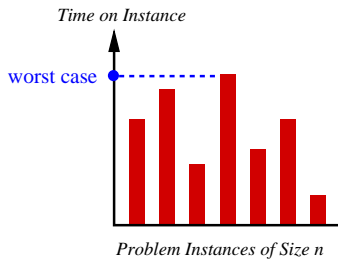
Typical cases:

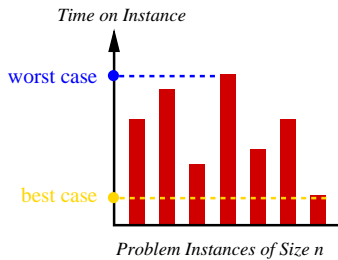
Integer Array I : $\text{size}(I) = \text{length of } I$

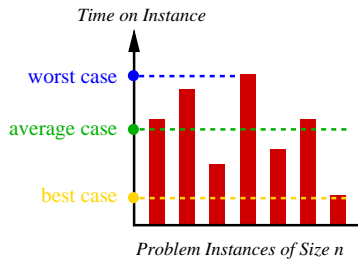
(Weighted) Graph G : $\text{size}(G) = \text{number of nodes}$ *or* $\text{size}(G) = \text{number of edges}$ (best choice can depend on specific computational problem or types of algorithms being considered).

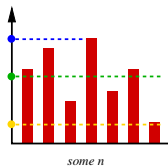
 Sometimes a more fine-grained analysis is conducted by characterizing the complexity of the input by several parameters (e.g. analyse complexity of graph algorithms as a function of number of nodes *and* number of edges of input graph).

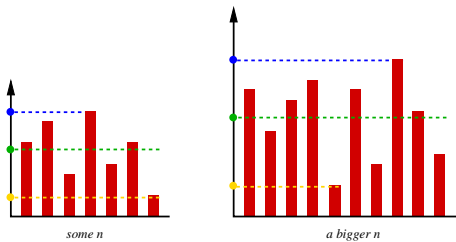


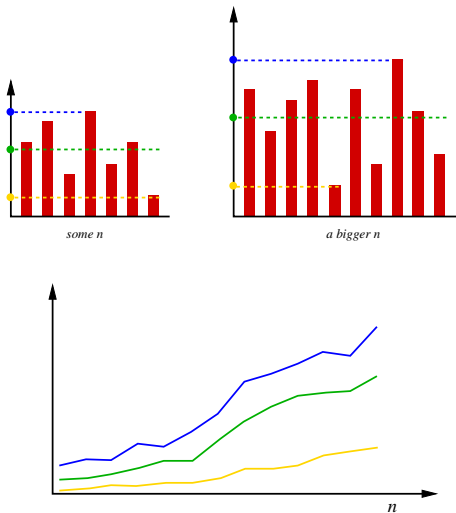












Our focus is on

$T(n)$ = *worst-case runtime for instances of size n*

```

INSERTSORT(I)
/* n = I.length */
1 for j = 2..n do
2   key = I[j]
3   i = j - 1
4   while i > 0 and I[i] > key do
5     I[i + 1] = I[i]
6     i = i - 1
7   I[i + 1] = key

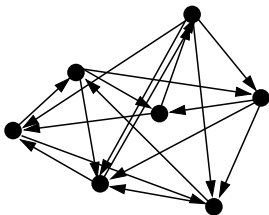
```

Line No.	Number of times executed		
	Best	Worst	Average
1	$n - 1$	$n - 1$	$n - 1$
2	$n - 1$	$n - 1$	$n - 1$
3	$n - 1$	$n - 1$	$n - 1$
4	$n - 1$	$2 + 3 + \dots + (n + 1)$	$\frac{2+3+\dots+(n+1)}{2}$
5	0	$1 + 2 + \dots + n$	$\frac{1+2+\dots+n}{2}$
6	0	$1 + 2 + \dots + n$	$\frac{1+2+\dots+n}{2}$
7	$n - 1$	$n - 1$	$n - 1$
total	$\sim n$	$\sim n^2$	$\sim n^2$

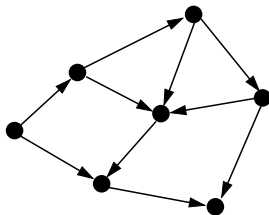
Why not average case analysis?

- ▶ Average case often close to worst case
- ▶ Average runtime for instances of size n difficult to compute/analyze
- ▶ Simple averaging over all instances of size n is often not representative for the average runtime over instances seen in practice

Example: Graphs

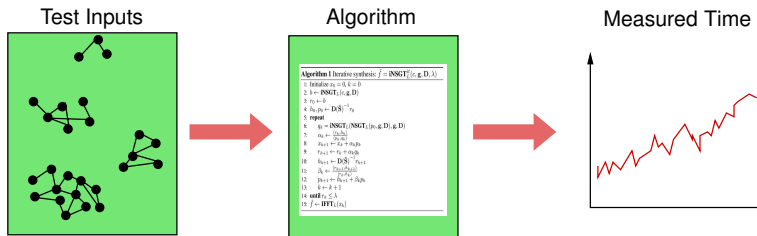


Random Graph



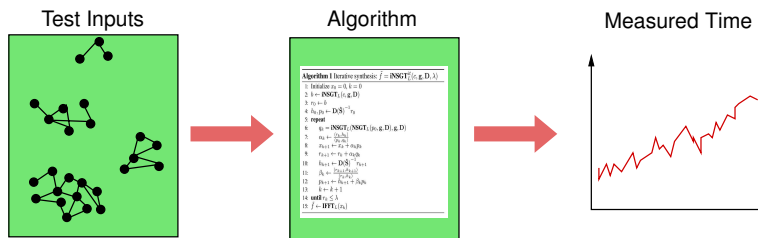
Typical input graph
(e.g. route planning in road network)

Instead of a theoretical average case analysis, see how algorithm performs on average in practice:



Large collection of input instances of different sizes (collected from applications, or synthetically generated)

Instead of a theoretical average case analysis, see how algorithm performs on average in practice:



Large collection of input instances of different sizes (collected from applications, or synthetically generated)

- ▶ Important in practice, especially for *comparison* of different algorithms
- ▶ Does not provide general, theoretical results
- ▶ Potential difficulties in generalizing results from test cases to performance on input instances in different input domains

Also important:

$S(n)$ = *worst-case memory consumption for instances of size n*

- ▶ for some applications critical: can run out of space faster than out of time!
- ▶ not the main concern for most of the “classical” problems we consider
- ▶ often a tradeoff: fast algorithm that uses much space vs. slower algorithm that uses less space