Algorithmic Thinking and Problem Solving Techniques

Getting Successful in Coding Interviews

Olivier Festor

Telecom Nancy – Semester 5

2020-2021

About this document

Authors:

- Martin Quinson (course founder, did most of the remaining content)
- Sébastien da Silva (author of the french version up to 2019)
- Olivier Festor (author of the problem solving chapter + Python support)

About this document

Technical aspects

- LaTeX document (class latex-beamer), compiled with latex-make
- Figures: Some xfig, some tikz, some inkscape, some Affinity Designer

About me

- Study: Computer Science (PhD, Henri Poincaré University).
- Experience: Researcher at IBM (3 years), Postdoc at EURECOM (1 year)
  Researcher at Inria (10 years)
  Research Director at Inria (until 2012)
  Director of Research EIT Digital (2010-2012)
  CS Professor at TELECOM Nancy (since 2012) Teaching: Télécom Nancy,

Teachings:

- Algorithms and Data Structures
- Problem Solving
- Java/Scala/C
- Advanced Data Structures
- Network Management
- Cyber-security Fundamentals and Network Security
- Network Programming
- Project Management (Agile/SCRUM/CICD/DevOps)

Outreach: Digital Democracy and Digital cities development, ...

About this module: Algorithms and Problem Solving

Programming? Let the computer do your work!

- How to explain what to do?
- How to make sure that it does what it is supposed to? That it is efficient?
- What if it does not?

About this module: Algorithms and Problem Solving

Programming? Let the computer do your work!

  - How to explain what to do?
  - How to make sure that it does what it is supposed to? That it is efficient?
  - What if it does not?

Module content and goals:

- Introduction to Algorithmics and Problem Solving
  - Master theoretical basements (computer science is a science)
  - Know some classical problem resolution techniques
  - Know how to evaluate solutions (correctness, performance)
- Learn-by-doing activity (you need to *practice*)

About this module: Algorithms and Problem Solving

Prerequisites

- COI at TELECOM Nancy
  - Running and operational development and Python environment (PyCharm, Python 3.8+)
  - Version management and scientific edition software (GitLab, LaTeX)
- CPGE programme algorithms mastering (loops, sorting, basic data structures)
- Sense of logic, intuition and good mathematical background

About this module: Algorithms and Problem Solving

Prerequisites

- COI at TELECOM Nancy
  - Running and operational development and Python environment (PyCharm, Python 3.8+)
  - Version management and scientific edition software (GitLab, LaTeX)
- CPGE programme algorithms mastering (loops, sorting, basic data structures)
- Sense of logic, intuition and good mathematical background

- Feeds future lectures
  - Discrete mathematics
  - Probability and Statistics
  - Object Oriented Programming; Object-Oriented Design
  - ...

Syllabus

1. Practical and Theoretical Foundations of Programming
   - CS vs. SE; Abstraction for complex algorithms; Algorithmic efficiency.
2. Iterative Sorting Algorithms
   - Specification; Selection, Insertion and Bubble sorts.
3. Recursion
   - Principles; Practice; Recursive sorts; Non-recursive From; Backtracking.
4. Testing Software
   - Testing techniques; Testing strategies; pytest; Design By Contract.
5. Problem Solving Strategies Zoo
6. Software Correction : done in the Mathematics for Computer Science lecture

This may change a bit to adapt and improve the class

Lectures, Labs, Exams, . . .

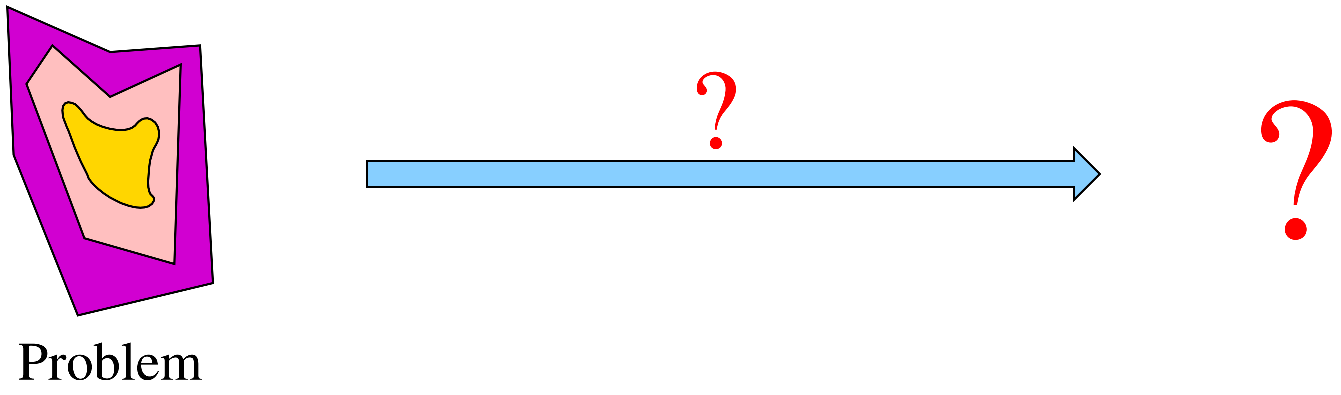
`https://arche.univ-lorraine.fr/course/view.php?id=39557`

First Chapter

Practical and Theoretical Foundations of Programming

- Introduction
  - From the problem to the code
  - Computer Science vs. Software Engineering
- Designing Algorithms for Complex Problems
  - Composition
  - Abstraction
- Python
- Comparing Algorithms' Efficiency
  - Best case, worst case, average analysis
  - Asymptotic complexity
- Algorithmic Stability
- Conclusion

Problems



Provided by clients (or teachers ;)

Problems

- Problems are generic

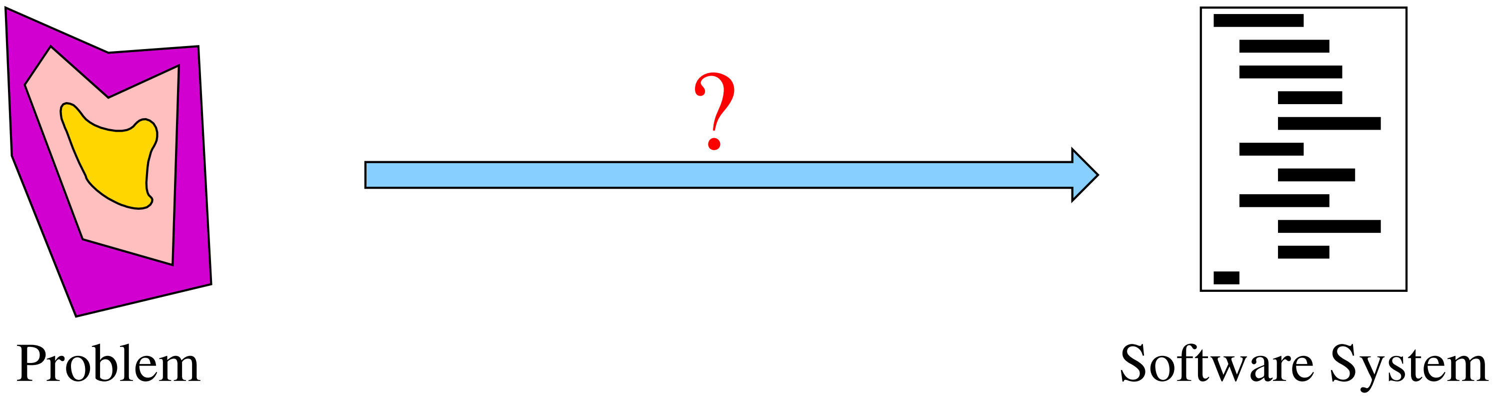
  Example: Determine the minimal value of a set of integers

Instances of a problem

- The problem for a given data set

  Example: Determine the minimal value of $\{17, 6, 42, 24\}$

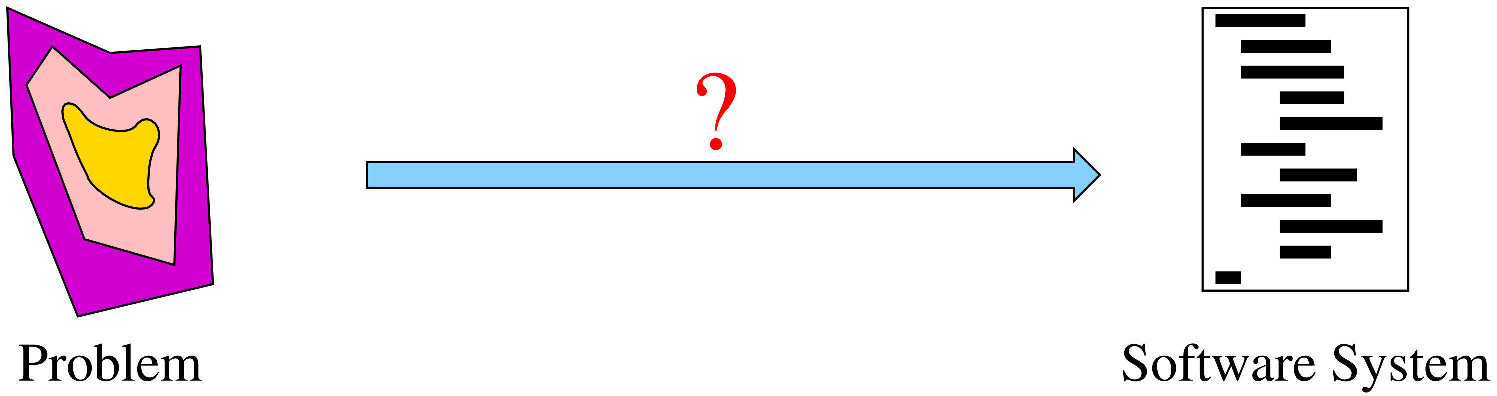
Problems and Programs



Software systems (*ie.*, Programs)

- Describes a set of actions to be achieved in a given order
- Doable (tractable) by computers

Problems and Programs



Software systems (*ie.*, Programs)

- Describes a set of actions to be achieved in a given order
- Doable (tractable) by computers

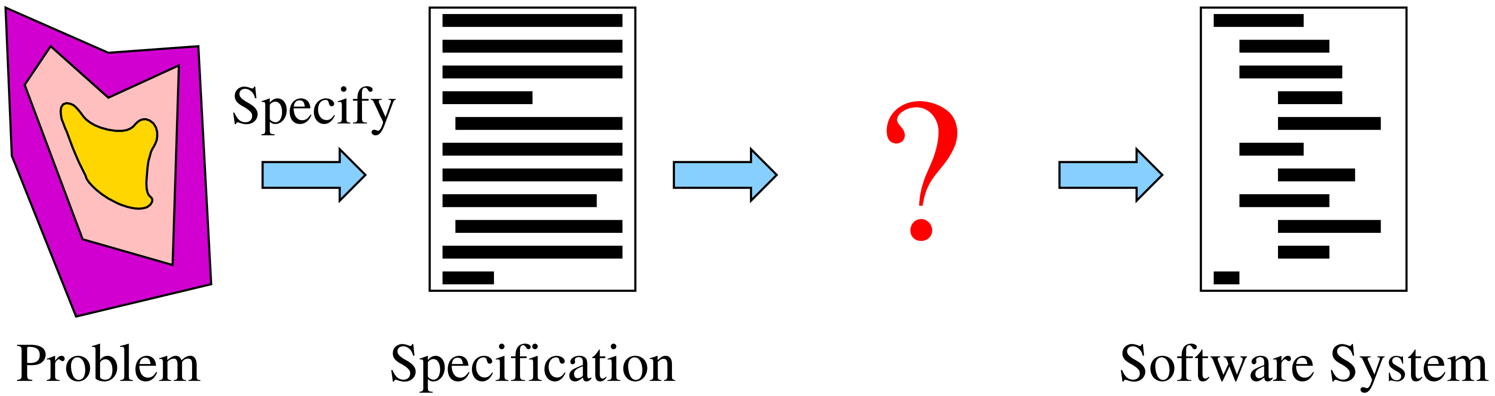
Problem Specification

- Must be clear, precise, complete, without ambiguities
  Bad example: find position of minimal element (two answers for $\{4, 2, 5, 2, 42\}$)
  Good example: Let L be the set of positions for which the value is minimal.
  Find the minimum of L

Using the Right Models

- Need simple models to understand complex artifacts (ex: city map)

Methodological Principles



Abstraction think before coding (!)

- Describe how to solve the problem

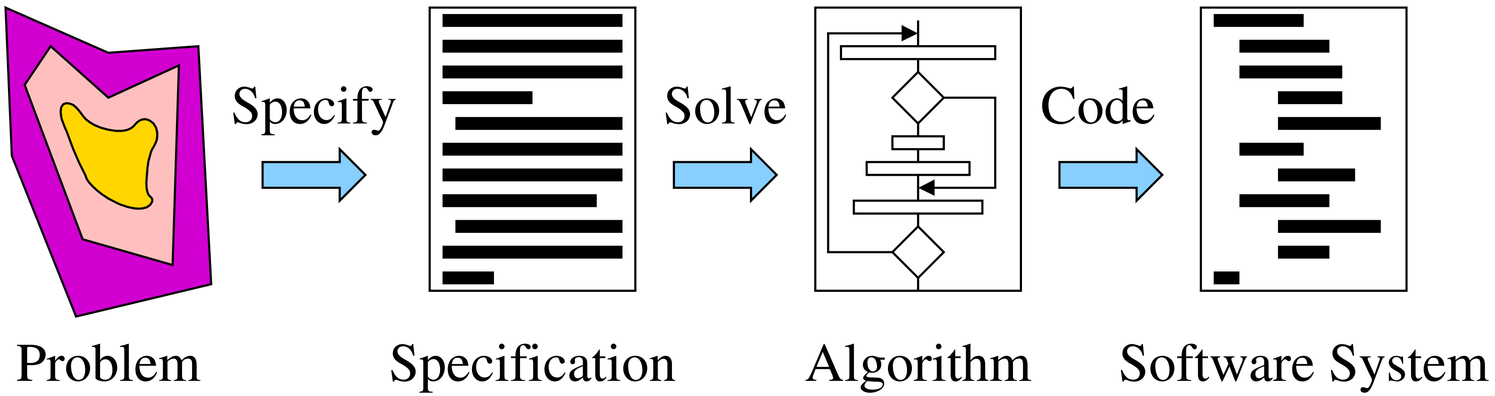
Divide, Conquer and Glue (top-down approach)

- Divide complex problem into simpler sub-problems (think of Descartes)
- Conquer each of them
- Glue (combine) partial solutions into the big one

Modularity

- Large systems built of components: **modules**
- Interface between modules allow to mix and match them

Algorithms



Precise description of the resolution process of a well specified problem

- Must be understandable (by human beings)
- Does not depend on target programming language, compiler or machine
- Can be an diagram (as pictured), but difficult for large problems
- Can be written in a simple language (called **pseudo-code**)

"Formal" definition

- Sequence of actions acting on problem data to induce the expected result

New to Algorithms?

Not quite, you use them since a long time

Lego bricks™	$\xrightarrow{\text{list of pictures}}$	Castle
Ikea™ desk	$\xrightarrow{\text{building instructions}}$	Desk
Home location	$\xrightarrow{\text{driving directions}}$	Party location
Eggs, Wheat, Milk	$\xrightarrow{\text{recipe}}$	Cake
Two 6-digits integers	$\xrightarrow{\text{arithmetic know-how}}$	sum

And now

List of students	$\xrightarrow{\text{sorting algorithm}}$	Sorted list
Maze map	$\xrightarrow{\text{appropriated algorithm}}$	Way out

Computer Science vs. Software Engineering

> *Computer science is a science of abstraction – creating the right model for a problem and devising the appropriate mechanizable technique to solve it.* – Aho and Ullman

NOT (only) Science of Computers

> *Computer science is not more related to computers than Astronomy to telescopes.* – Dijkstra

- Many concepts were framed and studied before the electronic computer
- To the logicians of the 20's, a *computer* was a person with pencil and paper

Science of Computing

- Automated problem solving
- Automated systems that produce solutions
- Methods to develop solution strategies for these systems
- Application areas for automatic problem solving

Foundations of Computing

Fundamental mathematical and logical structures

- To understand computing
- To analyze and verify the correctness of software and hardware

Main issues of interest in Computer Science

- Calculability
  - Given a problem, can we show whether there exist an algorithm solving it?
  - Which are the problems for which no algorithm exist? How to categorize them?
- Complexity
  - How long does my algorithm need to answer? (as function of input size)
  - How much memory does it take?
  - Is my algorithm optimal, or does a better one exist?
- Correctness
  - Can we be certain that a given algorithm always reaches a solution?
  - Can we be certain that a given algorithm always reaches the right solution?

Software Engineering vs. Computer Science

Producing technical answers to consumers' needs

Software Engineering Definition

- Study of methods for producing and evaluating software

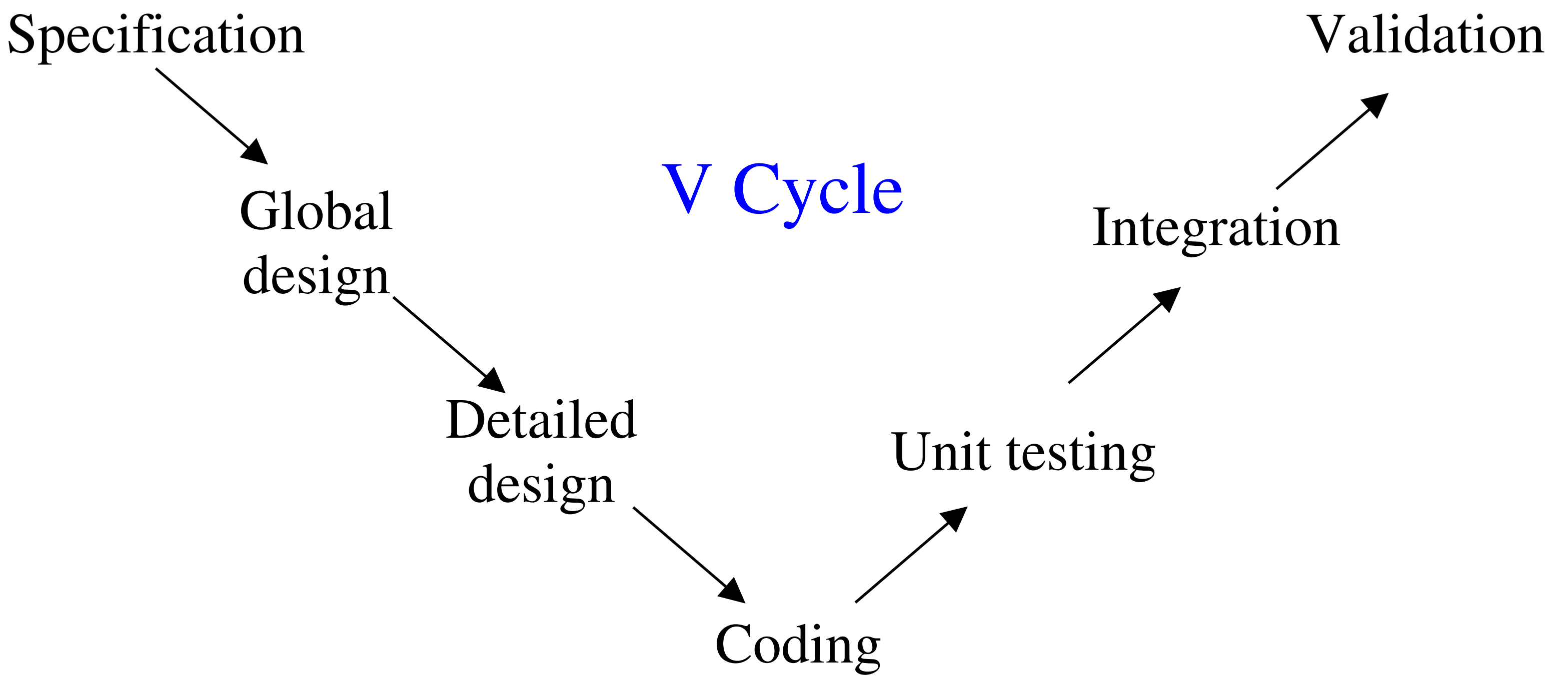
Software Engineering vs. Computer Science

Producing technical answers to consumers' needs

Software Engineering Definition

- Study of methods for producing and evaluating software

Life cycle of a software (*much* more details to come later)



- Global design: Identify application modules
- Detailed design: Specify within modules

As future IT engineers, you need both CS and SE

Without Software Engineering

- Your production will not match consumers' expectation
- You will induce more bugs and problems than solutions
- Each program will be a pain to develop and to maintain for you
- You won't be able to work in teams

Without Computer Science

- Your programs will run slowly, deal only with limited data sizes
- You won't be able to tackle difficult (and thus well paid) issues
- You won't be able to evaluate the difficulty of a task (and thus its price)
- You will reinvent the wheel (badly)

As future IT engineers, you need both CS and SE

Without Software Engineering

- Your production will not match consumers' expectation
- You will induce more bugs and problems than solutions
- Each program will be a pain to develop and to maintain for you
- You won't be able to work in teams

Without Computer Science

- Your programs will run slowly, deal only with limited data sizes
- You won't be able to tackle difficult (and thus well paid) issues
- You won't be able to evaluate the difficulty of a task (and thus its price)
- You will reinvent the wheel (badly)

Two approaches of the same issues

- Correctness: CS $\rightsquigarrow$ prove algorithms right; SE $\rightsquigarrow$ chase (visible) bugs
- Efficiency: CS $\rightsquigarrow$ theoretical bounds on performance, optimality proof;
  SE $\rightsquigarrow$ optimize execution time and memory usage

First Chapter

Practical and Theoretical Foundations of Programming

There are always several ways to solve a problem

Choice criteria between algorithms

- Correctness: provides the right answer
- Simplicity: KISS! (jargon acronym for *keep it simple, silly*)
- Efficiency: fast, use little memory
- Stability: small change in input does not change output

There are always several ways to solve a problem

Choice criteria between algorithms

- Correctness: provides the right answer
- Simplicity: KISS! (jargon acronym for *keep it simple, silly*)
- Efficiency: fast, use little memory
- Stability: small change in input does not change output

Real problems ain't easy

- They are not fixed, but dynamic
  - Specification helps users understanding the problem better
    That is why they often add wanted functionalities after specification
  - My text editor is v23.2.1 (hundreds of versions for "just a text editor")
- They are complex (composed of several interacting entities)

> *In computing, turning the obvious into the useful is a living definition of the word "frustration".*
>
> – "Epigrams in Programming", by Alan J. Perlis.

Dealing with Complexity

Some classical design principles help

- Composition: split problem in simpler sub-problems and compose pieces
- Abstraction: forget about details and focus on important aspects

Object Oriented Programming

- Classical answer to specification complexity and dynamicity Encapsulation, polymorphism, heritage, . . .
- That's one way to design applications in a modular manner
- Other approaches exists, but none have the same momentum currently

Dealing with Complexity

Some classical design principles help

- Composition: split problem in simpler sub-problems and compose pieces
- Abstraction: forget about details and focus on important aspects

Object Oriented Programming

- Classical answer to specification complexity and dynamicity Encapsulation, polymorphism, heritage, . . .
- That's one way to design applications in a modular manner
- Other approaches exists, but none have the same momentum currently

Rest of this module

- How to write each block / units / objects to be composed in OOP

Why algorithms before OOP and not the contrary?

- Coding at small before programming at large
- (that's an endless debate, pros and cons for both approaches)

Dealing with complexity: Composition

Composite structure

- Definition: a software system composed of manageable pieces
- ☺ The smaller the component, the easier it is to build and understand
- ☹ The more parts, the more possible interactions there are between parts
- $\Rightarrow$ the more complex the resulting structure
- Need to balance between simplicity and interaction minimization

Dealing with complexity: Composition

Composite structure

- Definition: a software system composed of manageable pieces
- ☺ The smaller the component, the easier it is to build and understand
- ☹ The more parts, the more possible interactions there are between parts
- ⇒ the more complex the resulting structure
- Need to balance between simplicity and interaction minimization

Good example: audio system

Easy to manage because:

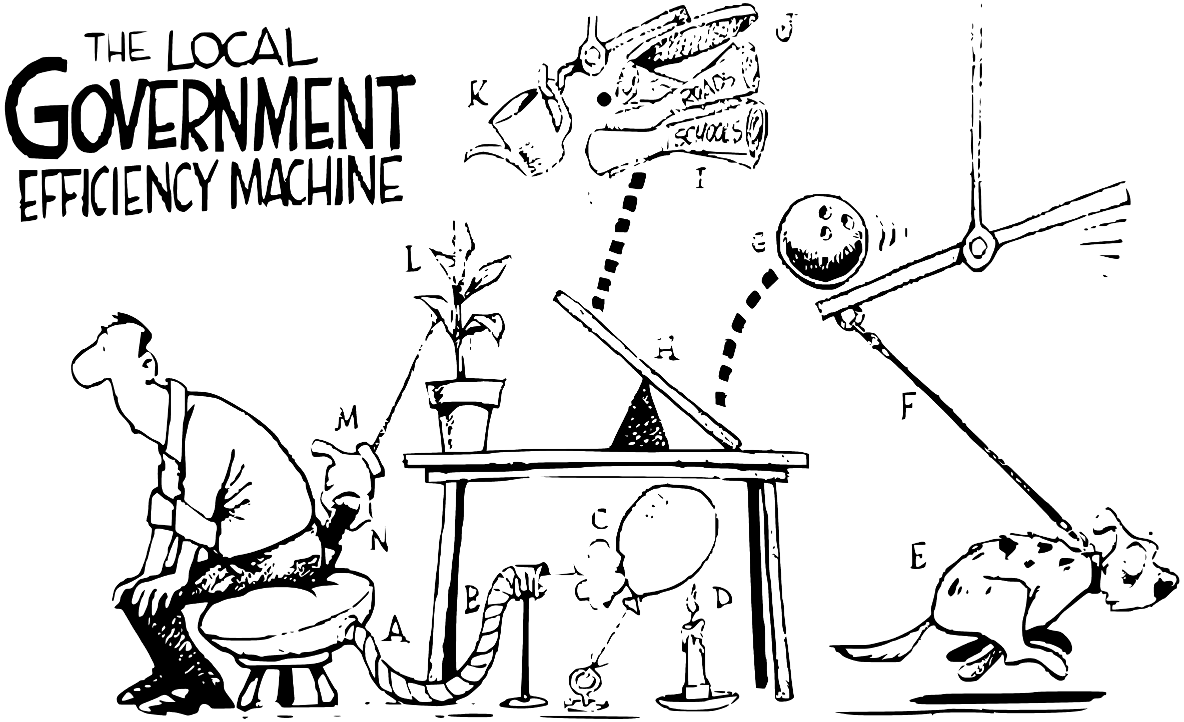
- each component has a carefully specified function
- components are easily integrated
- i.e. the speakers are easily connected to the amplifier

Composition counter-example (1/2)

Rube Goldberg machines

- Device not obvious, modification unthinkable
- Parts lack intrinsic relationship to the solved problem
- Utterly high complexity

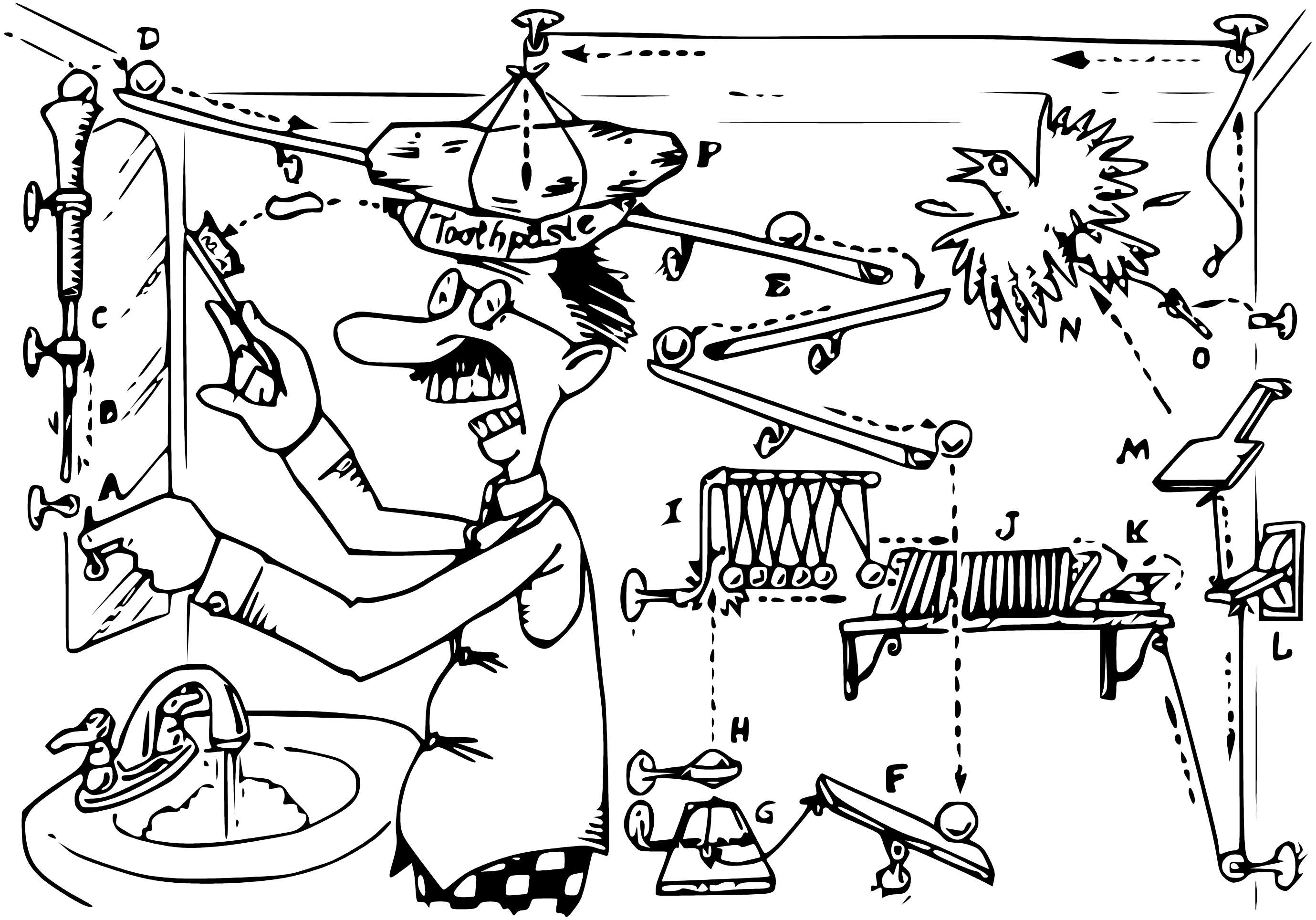
Example: Tax collection machine



A. Taxpayer sits on cushion
B. Forcing air through tube
C. Blowing balloon
D. Into candle
E. Explosion scares dog
F. Which pull leash
G. Dropping ball
H. On teeter totter
I. Launch plans
J. Which tilts lever
K. Then Pitcher
L. Pours water on plant
M. Which grows, pulling chain
N. Hand lifts the wallet

Composition counter-example (2/2)

Rube Goldberg's toothpaste dispenser

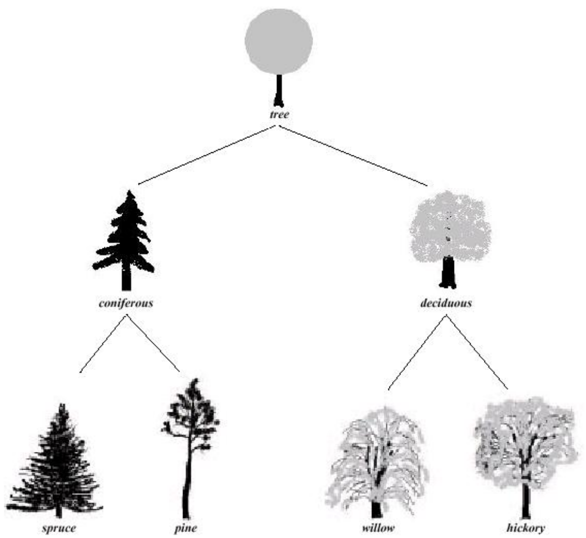


Such over engineered solutions should obviously remain jokes !

Dealing with complexity: Abstraction

Abstraction

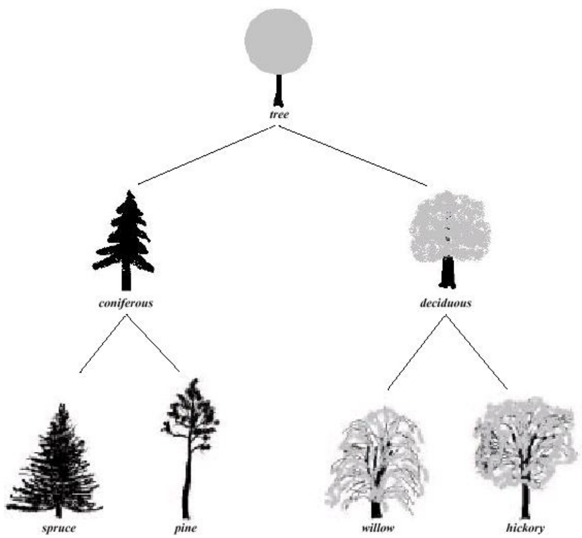
- Dealing with components and interactions without worrying about details
- Not “vague” or “imprecise”, but focused on few relevant properties
- Elimination of the irrelevant and amplification of the essential
- Capturing commonality between different things



Dealing with complexity: Abstraction

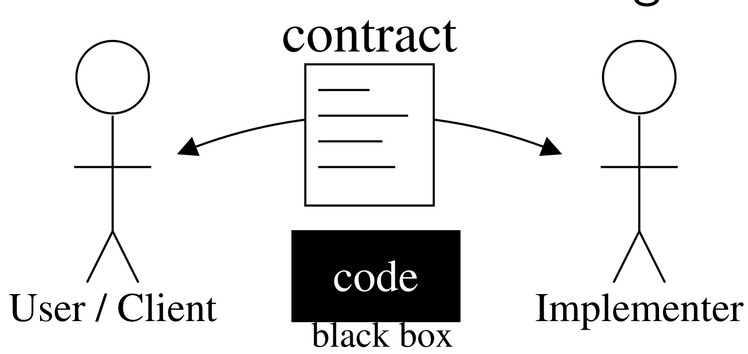
Abstraction

- Dealing with components and interactions without worrying about details
- Not “vague” or “imprecise”, but focused on few relevant properties
- Elimination of the irrelevant and amplification of the essential
- Capturing commonality between different things



Abstraction in programming

- Think about what your components should do before
- Ie, abstract their **interface** before coding



- Show your interface, hide your implementation

Practical and Theoretical Foundations of Programming

Why Python

Main reasons for us:

- Most, if not all, of You are familiar with Python
- We want to talk about algorithms, not to bother you about syntax
- 1st language used worldwide .... *subject to debate !!!*

You will use Python in other courses as well

- Systems scripting, Discrete Mathematics, Statistics, ...

Why Python

Main reasons for us:

- Most, if not all, of You are familiar with Python
- We want to talk about algorithms, not to bother you about syntax
- 1st language used worldwide .... *subject to debate !!!*

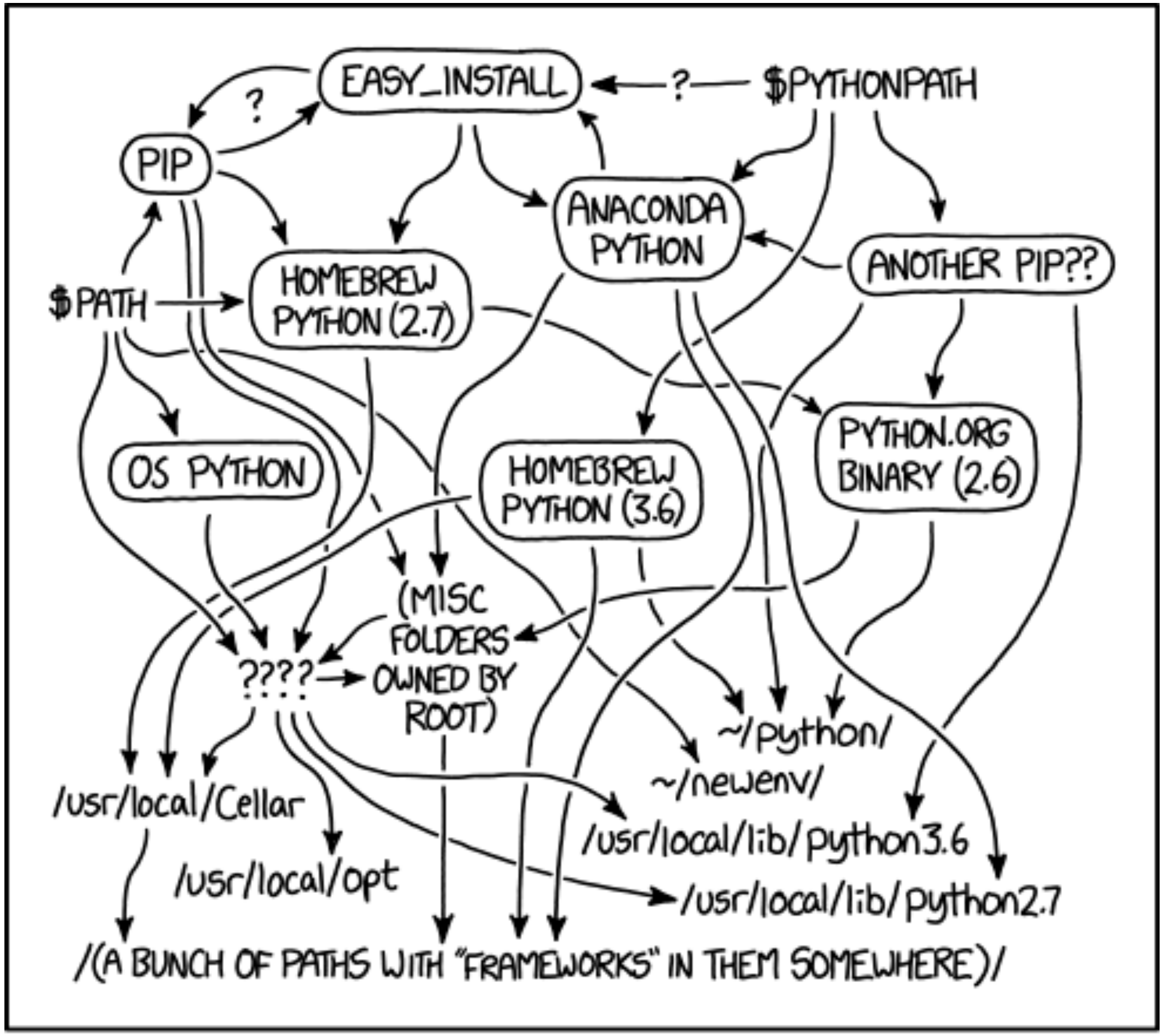
You will use Python in other courses as well

- Systems scripting, Discrete Mathematics, Statistics, ...
- But other languages will be needed for other purposes (C, Java, Scala, Haskell, Go, Javascript, . . .)

Starting Python

Installation: Get it from `https://www.python.org/downloads/` (version 3.8.5 and greater)

Always know what You do !!!



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED

Starting Python

Executing your code

myfile.py

```
print("Welcome to your new
world!");
```

Run directly

```
$ python myfile.py
Welcome to your new
world!
$
```

myscript

```
#!/usr/bin/python
x = 3
print("Welcome to your new
world!")
```

Turn it into a script

```
$ chmod +x myscript
$ ./myscript
$ Welcome to your new world!
```

Run interactively

```
$ python
>>> execfile('myfile.py')
```

First Chapter

Practical and Theoretical Foundations of Programming

Comparing Algorithms' Efficiency

There are always more than one way to solve a problem

Choice criteria between algorithms

- Correctness: provides the right answer
- Simplicity: *not* Rube Goldberg's machines
- Efficiency: fast, use little memory
- Stability: small change in input does not change output

Comparing Algorithms' Efficiency

There are always more than one way to solve a problem

Choice criteria between algorithms

- Correctness: provides the right answer
- Simplicity: *not* Rube Goldberg's machines
- Efficiency: fast, use little memory
- Stability: small change in input does not change output

Empirical efficiency measurements

- Code the algorithm, benchmark it and use runtime statistics
- ☹ Several factors impact performance:
  machine, language, programmer, compiler, compiler's options, operating system, ...

$\Rightarrow$ Performance not generic enough for comparison

Comparing Algorithms' Efficiency

There are always more than one way to solve a problem

Choice criteria between algorithms

- Correctness: provides the right answer
- Simplicity: *not* Rube Goldberg's machines
- Efficiency: fast, use little memory
- Stability: small change in input does not change output

Empirical efficiency measurements

- Code the algorithm, benchmark it and use runtime statistics
- ☹ Several factors impact performance:
  machine, language, programmer, compiler, compiler's options, operating system, ...
- $\Rightarrow$ Performance not generic enough for comparison

Mathematical efficiency estimation

- Count amount of basic instruction as function of input size
- ☺ Simpler, more generic and often sufficient

  (true in theory; in practice, optimization necessary **in addition** to this)

Best case, worst case, average analysis

Algorithm running time depends on the data

Example: Linear search in a List

```
def linearSearchInList(pList,pVal):
    for i in range(0,len(pList)-1):
        if pList[i]== pVal:
            return True
    return False
# End of linearSearchInList
```

- Case 1: search whether 42 is in $\{42, 3, 2, 6, 7, 8, 12, 16, 17, 32, 55, 44, 12\}$ answer found after one step
- Case 2: search whether 4 is in $\{42, 3, 2, 6, 7, 8, 12, 16, 17, 32, 55, 44, 12\}$ need to traverse the whole array to decide (n steps)

Best case, worst case, average analysis

Algorithm running time depends on the data

Example: Linear search in a List

```
def linearSearchInList(pList,pVal):
    for i in range(0,len(pList)-1):
        if pList[i]== pVal:
            return True
    return False
# End of linearSearchInList
```

- Case 1: search whether 42 is in $\{42, 3, 2, 6, 7, 8, 12, 16, 17, 32, 55, 44, 12\}$ answer found after one step
- Case 2: search whether 4 is in $\{42, 3, 2, 6, 7, 8, 12, 16, 17, 32, 55, 44, 12\}$ need to traverse the whole array to decide (n steps)

Counting the instructions to run in each case

- t_{min}: #instructions for the best case inputs
- t_{max}: #instructions for the worst case inputs
- t_{avg}: #instructions on average (average of values coefficiented by probability)
  $t_{avg} = p_1 t_1 + p_2 t_2 + \ldots + p_n t_n$

Linear search runtime analysis

```
for i in range(0,len(pList)):
    if pList[i]== pVal:
        return True
return False
```

- For simplicity, let's assume the value is in the array, positions are equally likely
- Let's count tests (noted t), additions (noted a) and value changes (noted c)

Best case: searched data in first position

- 1 value change (i=0); 2 tests (loop boundary + equality)
- $t_{min} = c + 2t$

Worst case: searched data in last position

- 1 value change (i=0); {2 tests, 1 change, 1 addition (i+=1)} per loop
- $t_{max} = c + n \times (2t + 1c + 1a) = (n+1) \times c + 2n \times t + n \times a$

Linear search runtime analysis

```
for i in range(0,len(pList)):
    if pList[i]== pVal:
        return True
return False
```

- For simplicity, let's assume the value is in the array, positions are equally likely
- Let's count tests (noted t), additions (noted a) and value changes (noted c)

Best case: searched data in first position

- 1 value change (i=0); 2 tests (loop boundary + equality)
- $t_{min} = c + 2t$

Worst case: searched data in last position

- 1 value change (i=0); {2 tests, 1 change, 1 addition (i+=1)} per loop
- $t_{max} = c + n \times (2t + 1c + 1a) = (n+1) \times c + 2n \times t + n \times a$

Average case: searched data in position p with probability $\frac{1}{n}$

- $t_{avg} = c + \sum_{p \in [1,n]} \frac{1}{n} \times (2t + c + a) \times p$

Linear search runtime analysis

```
for i in range(0,len(pList)):
    if pList[i]== pVal:
        return True
return False
```

- For simplicity, let's assume the value is in the array, positions are equally likely
- Let's count tests (noted t), additions (noted a) and value changes (noted c)

Best case: searched data in first position

- 1 value change (i=0); 2 tests (loop boundary + equality)
- $t_{min} = c + 2t$

Worst case: searched data in last position

- 1 value change (i=0); {2 tests, 1 change, 1 addition (i+=1)} per loop
- $t_{max} = c + n \times (2t + 1c + 1a) = (n+1) \times c + 2n \times t + n \times a$

Average case: searched data in position p with probability $\frac{1}{n}$

- $t_{avg} = c + \sum_{p \in [1,n]} \frac{1}{n} \times (2t + c + a) \times p = c + \frac{1}{n} \times (2t + c + a) \times \sum_{p \in [1,n]} p$

Linear search runtime analysis

```
for i in range(0,len(pList)):
    if pList[i]== pVal:
        return True
return False
```

- For simplicity, let's assume the value is in the array, positions are equally likely
- Let's count tests (noted t), additions (noted a) and value changes (noted c)

Best case: searched data in first position

- 1 value change (i=0); 2 tests (loop boundary + equality)
- $t_{min} = c + 2t$

Worst case: searched data in last position

- 1 value change (i=0); {2 tests, 1 change, 1 addition (i+=1)} per loop
- $t_{max} = c + n \times (2t + 1c + 1a) = (n+1) \times c + 2n \times t + n \times a$

Average case: searched data in position p with probability $\frac{1}{n}$

- $t_{avg} = c + \sum_{p \in [1,n]} \frac{1}{n} \times (2t + c + a) \times p = c + \frac{1}{n} \times (2t + c + a) \times \sum_{p \in [1,n]} p$

  $t_{avg} = c + \frac{n(n-1)}{2n} \times (2t + c + a)$

Linear search runtime analysis

```
for i in range(0,len(pList)):
    if pList[i]== pVal:
        return True
return False
```

- For simplicity, let's assume the value is in the array, positions are equally likely
- Let's count tests (noted t), additions (noted a) and value changes (noted c)

Best case: searched data in first position

- 1 value change (i=0); 2 tests (loop boundary + equality)
- $t_{min} = c + 2t$

Worst case: searched data in last position

- 1 value change (i=0); {2 tests, 1 change, 1 addition (i+=1)} per loop
- $t_{max} = c + n \times (2t + 1c + 1a) = (n+1) \times c + 2n \times t + n \times a$

Average case: searched data in position p with probability $\frac{1}{n}$

- $t_{avg} = c + \sum_{p \in [1,n]} \frac{1}{n} \times (2t + c + a) \times p = c + \frac{1}{n} \times (2t + c + a) \times \sum_{p \in [1,n]} p$

  $t_{avg} = c + \frac{n(n-1)}{2n} \times (2t + c + a) = (n-1) \times t + \frac{n+1}{2} \times c + \frac{n-1}{2} \times a$

Simplifying equations

$\boxed{t_{avg} = (n-1) \times t + \frac{n+1}{2} \times c + \frac{n-1}{2} \times a}$ is too complicated

Simplifying equations

$$\boxed{t_{avg} = (n-1) \times t + \frac{n+1}{2} \times c + \frac{n-1}{2} \times a} \text{ is too complicated}$$

Reducing amount of variables

- To simplify, we only count the most expensive operations
- But which it is is not always clear...
- Let's take write accesses c (classical but arbitrary choice)

Simplifying equations

$$\boxed{t_{avg} = (n-1) \times t + \frac{n+1}{2} \times c + \frac{n-1}{2} \times a}$$ is too complicated

Reducing amount of variables

- To simplify, we only count the most expensive operations
- But which it is is not always clear...
- Let's take write accesses c (classical but arbitrary choice)

Focusing on dominant elements

- We can forget about constant parts if there is n operations
- We can forget about linear parts if there is n^2 operations
- ...
- Only consider the most dominant elements when n is very big

$\Rightarrow$ This is called **asymptotic complexity**

Asymptotic Complexity: Big-O notation

Mathematical definition

- Let $T(n)$ be a non-negative function
- $\boxed{T(n) \in O(f(n))} \Leftrightarrow \exists$ constants c, n_0 so that $\forall n > n_0, \boxed{T(n) \leq c \times f(n)}$
- f(n) is an upper bound of T(n) ...

... after some point, and with a constant multiplier

Application to runtime evaluation

- $T(n) \in O(n^2) \Rightarrow$ when n is big enough, you need less than n^2 steps
- This gives a upper bound

Big-O examples

Example 1: Simplifying a formula

- Linear search: $t_{avg} = (n-1) \times t + \frac{n+1}{2} \times c + \frac{n-1}{2} \times a \Rightarrow \boxed{T(n) = O(n)}$
- Imaginary example: $T(n) = 17n^2 + \frac{32}{17}n + \pi \Rightarrow \boxed{T(n) = O(n^2)}$
- If T(n) is constant, we write T(n)=O(1)

Practical usage

- Since this is a upper bound, $T(n) = O(n^3)$ is also true when $T(n) = O(n^2)$
- But not as relevant

Big-O examples

Example 1: Simplifying a formula

- Linear search: $t_{avg} = (n-1) \times t + \frac{n+1}{2} \times c + \frac{n-1}{2} \times a \Rightarrow \boxed{T(n) = O(n)}$
- Imaginary example: $T(n) = 17n^2 + \frac{32}{17}n + \pi \Rightarrow \boxed{T(n) = O(n^2)}$
- If T(n) is constant, we write T(n)=O(1)

Practical usage

- Since this is a upper bound, $T(n) = O(n^3)$ is also true when $T(n) = O(n^2)$
- But not as relevant

Example 2: Computing big-O values directly

List initialization

```
def listInitialization(pList,pVal):
    for i in range(0,len(pList)):
        pList[i]= pVal
```

- We have n steps, each of them doing a constant amount of work
- $T(n) = c \times n \quad \Rightarrow \quad T(n) = O(n)$
  (don't bother counting the constant elements)

Big-Omega notation

Mathematical definition

- Let $T(n)$ be a non-negative function
- $\boxed{T(n) \in \Omega(f(n))} \Leftrightarrow \exists$ constants c, n_0 so that $\forall n > n_0$, $\boxed{T(n) \geq c \times f(n)}$
- Similar to Big-O, but gives a lower bound
- Note: similarly to before, we are interested in big lower bounds

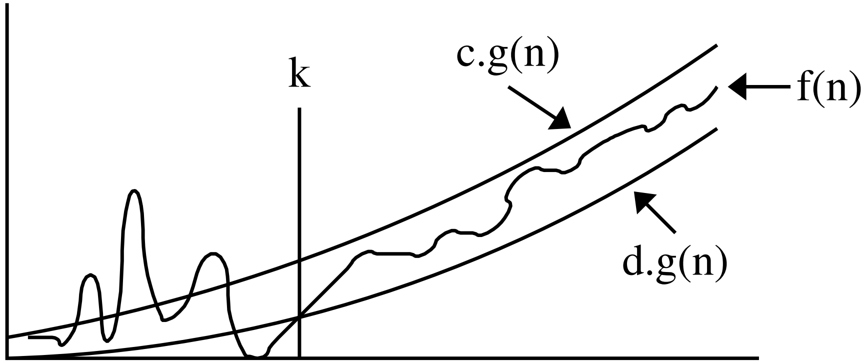
Example: $T(n) = c_1 \times n^2 + c_2 \times n$

- $T(n) = c_1 \times n^2 + c_2 \times n \geq c_1 \times n^2 \qquad \forall n > 1$
  $T(n) \geq c \times n^2$ for $c > c_1$
- Thus, $T(n) = \Omega(n^2)$

Theta notation

Mathematical definition

- $T(n) \in \Theta(g(n))$ if and only if $T(n) \in O(g(n))$ and $T(n) \in \Omega(g(n))$



Example

		n=10	n=1000	n=100000	
$\Theta(n)$	n 100n	10 1000	1000 10^5	10^5 10^7	seconds
$\Theta(n^2)$	n^2 $100n^2$	100 10^4	10^6 10^8	10^{10} 10^{12}	minutes
$\Theta(n^3)$	n^3 $100n^2$	1000 10^5	10^9 10^{11}	10^{15} 10^{17}	hours
$\Theta(2^n)$	2^n 100×2^n	1024 $> 10^5$	$> 10^{301}$ 10^{305}	∞ ∞	...
$\log(n)$	$\log(n)$ $100\log(n)$	3.3 332.2	9.9 996.5	16.6 1661	

Classical mistakes

Mistake notations

- Indeed, we have $O(\log(n)) = O(n) = O(n^2) = O(n^3) = O(2^n)$
- Likewise, we have $\Omega(\log(n)) = \Omega(n) = \Omega(n^2) = \Omega(n^3) = \Omega(2^n)$
- We only have $\Theta(\log(n)) \neq \Theta(n) \neq \Theta(n^2) \neq \Theta(n^3) \neq \Theta(2^n)$

Classical mistakes

Mistake notations

- Indeed, we have $O(\log(n)) = O(n) = O(n^2) = O(n^3) = O(2^n)$
  Because it's an upper bound; to be correct we should write $\subset$ instead of $=$
- Likewise, we have $\Omega(\log(n)) = \Omega(n) = \Omega(n^2) = \Omega(n^3) = \Omega(2^n)$
  Because it's a lower bound; we should write $\supset$ instead of $=$
- We only have $\Theta(\log(n)) \neq \Theta(n) \neq \Theta(n^2) \neq \Theta(n^3) \neq \Theta(2^n)$

Classical mistakes

Mistake notations

- Indeed, we have $O(\log(n)) = O(n) = O(n^2) = O(n^3) = O(2^n)$
  Because it's an upper bound; to be correct we should write $\subset$ instead of $=$
- Likewise, we have $\Omega(\log(n)) = \Omega(n) = \Omega(n^2) = \Omega(n^3) = \Omega(2^n)$
  Because it's a lower bound; we should write $\supset$ instead of $=$
- We only have $\Theta(\log(n)) \neq \Theta(n) \neq \Theta(n^2) \neq \Theta(n^3) \neq \Theta(2^n)$

  (but in practice, everybody use O() as if it were Θ() – although that's wrong)

Classical mistakes

Mistake notations

- Indeed, we have $O(\log(n)) = O(n) = O(n^2) = O(n^3) = O(2^n)$
  Because it's an upper bound; to be correct we should write $\subset$ instead of $=$
- Likewise, we have $\Omega(\log(n)) = \Omega(n) = \Omega(n^2) = \Omega(n^3) = \Omega(2^n)$
  Because it's a lower bound; we should write $\supset$ instead of $=$
- We only have $\Theta(\log(n)) \neq \Theta(n) \neq \Theta(n^2) \neq \Theta(n^3) \neq \Theta(2^n)$

  (but in practice, everybody use O() as if it were Θ() – although that's wrong)

Mistake worst case and upper bounds

- Worst case is the input data leading to the longest operation time
- Upper bound gives indications on increase rate when input size increases
  (same distinction between best case and lower bound)

Asymptotic Complexity in Practice

Rules to compute the complexity of an algorithm

Rule 1: Complexity of a sequence of instruction: Sum of complexity of each

Rule 2: Complexity of basic instructions (test, read/write memory): $O(1)$

Rule 3: Complexity of `if`/`switch` branching: Max of complexities of branches

Rule 4: Complexity of loops: Complexity of content $\times$ amount of loop

Rule 5: Complexity of methods: Complexity of content

Asymptotic Complexity in Practice

Rules to compute the complexity of an algorithm

Rule 1: Complexity of a sequence of instruction: Sum of complexity of each

Rule 2: Complexity of basic instructions (test, read/write memory): O(1)

Rule 3: Complexity of `if`/`switch` branching: Max of complexities of branches

Rule 4: Complexity of loops: Complexity of content $\times$ amount of loop

Rule 5: Complexity of methods: Complexity of content

Simplification rules

- Ignoring the constant:

  If $\mathrm{f(n)} = \mathrm{O}(k \times g(n))$ and $k > 0$ is constant then $\mathrm{f(n)} = \mathrm{O(g(n))}$

- Transitivity

  If $\mathrm{f(n)} = \mathrm{O(g(n))}$ and $\mathrm{g(n)} = \mathrm{O(h(n))}$ then $\mathrm{f(n)} = \mathrm{O(h(n))}$

- Adding big-Os

  If $\mathrm{A(n)} = \mathrm{O(f(n))}$ and $\mathrm{B(n)} = \mathrm{O(g(n))}$ then
  $$\mathrm{A(n)+B(n)} = \mathrm{O(max(f(n), g(n)))} = \mathrm{O(f(n)+g(n))}$$

- Multiplying big-Os

  If $\mathrm{A(n)} = \mathrm{O(f(n))}$ and $\mathrm{B(n)} = \mathrm{O(h(n))}$ then $A(n) \times B(n) = \mathrm{O}(f(n) \times g(n))$

Some examples

Example 1: $\boxed{\texttt{a=b}} \Rightarrow \Theta(1)$ (constant time)

Example 2

```
sum=0
n=10
for i in range(0,n):
        sum+= n
```

$\Theta(n)$

Example 3

```
for i in range(0,n):
    for j in range(0,n):
        sum +=1
for k in range(0,n):
        a[k]=k
```

$\Theta(1) + \Theta(n^2) + \Theta(n) = \Theta(n^2)$

Example 4

```
count = 0
for i in range(0,n):
    for j in range(0,i):
        count +=1
```

$\Theta(1) + O(n^2) = O(n^2)$
one can also show $\Theta(n^2)$

Example 5

```
sum=0;i=0;n=100
while i<n:
    sum +=1
    i *=2
```

$\Theta(\log(n))$ log is due to the $i \times 2$ *really*? : $-($

Going further on Algorithm Complexity

Problems' Classification

- Problems can also be sorted in class of complexities (not only algorithms) depending on the best existing algorithm to solve them
- Showing that no better algorithm exist for a given problem: Calculability
- Multi-million question: P=NP?

  P: polynomial algorithm to find the solution exists

  NP: candidate solution eval. in polynomial time, but no known polynomial algo

  NP-complete: set of NP problems for which if one P algorithm is found, it's applicable to every other NP-complete problems

Going further on Algorithm Complexity

Problems' Classification

- Problems can also be sorted in class of complexities (not only algorithms)
  depending on the best existing algorithm to solve them
- Showing that no better algorithm exist for a given problem: Calculability
- Multi-million question: P=NP?

  P: polynomial algorithm to find the solution exists
  NP: candidate solution eval. in polynomial time, but no known polynomial algo
  NP-complete: set of NP problems for which if one P algorithm is found,
  it's applicable to every other NP-complete problems

Time is not the only metric of interest: Space too

- In computation, there is a sort of tradeoff between space and time
  Faster algorithms need to pre-compute elements . . . requiring more storage memory

Going further on Algorithm Complexity

Problems' Classification

- Problems can also be sorted in class of complexities (not only algorithms)
  depending on the best existing algorithm to solve them
- Showing that no better algorithm exist for a given problem: Calculability
- Multi-million question: P=NP?

  P: polynomial algorithm to find the solution exists
  NP: candidate solution eval. in polynomial time, but no known polynomial algo
  NP-complete: set of NP problems for which if one P algorithm is found,
  it's applicable to every other NP-complete problems

Time is not the only metric of interest: Space too

- In computation, there is a sort of tradeoff between space and time
  Faster algorithms need to pre-compute elements . . . requiring more storage memory

So does Energy nowadays!

- Computational power of CPU grows linearly with frequency;
  Energy consumption grows (more than) quadratically with frequency
- To save energy (and money), split your task on several slower cores
  Parallel algorithms are the way to go (but it's *ways* harder)

Practical and Theoretical Foundations of Programming

Algorithmic stability

Computers use fixed precision numbers

- $10.0+1=11$

Algorithmic stability

Computers use fixed precision numbers

- $10.0+1=11$
- $10.0^{10}+1=10000000001$

Algorithmic stability

Computers use fixed precision numbers

- 10.0+1=11
- $10.0^{10} + 1 = 10000000001$
- $10.0^{16} + 1 = 10000000000000001$

Algorithmic stability

Computers use fixed precision numbers

- $10.0+1=11$
- $10.0^{10} + 1 = 10000000001$
- $10.0^{16} + 1 = 10000000000000001$
- $10.0^{17} + 1 = 10000000000000000\textcolor{red}{0} = 10.0^{17}$

Algorithmic stability

Computers use fixed precision numbers

- 10.0+1=11
- $10.0^{10} + 1 = 10000000001$
- $10.0^{16} + 1 = 10000000000000001$
- $10.0^{17} + 1 = 100000000000000000 = 10.0^{17}$

What is the value of $\sqrt{2^2}$?

- Old computers though it was 1.9999999

Other example

```
while value < 2E9 :
  value += 1E-8;
```

This is an infinite loop
(because when $value = 10^9$, $value + 10^{-8} = value$)

Algorithmic stability

Computers use fixed precision numbers

- 10.0+1=11
- $10.0^{10} + 1 = 10000000001$
- $10.0^{16} + 1 = 10000000000000001$
- $10.0^{17} + 1 = 100000000000000000 = 10.0^{17}$

What is the value of $\sqrt{2^2}$?

- Old computers though it was 1.9999999

Other example

```
while value < 2E9 :
  value += 1E-8;
```

This is an infinite loop
(because when $value = 10^9$, $value + 10^{-8} = value$)

Numerical instabilities are to be killed to predict weather, simulate a car crash or control a nuclear power plant

Algorithmic stability

Computers use fixed precision numbers

- 10.0+1=11
- $10.0^{10} + 1 = 10000000001$
- $10.0^{16} + 1 = 10000000000000001$
- $10.0^{17} + 1 = 100000000000000000 = 10.0^{17}$

What is the value of $\sqrt{2^2}$?

- Old computers though it was 1.9999999

Other example

```
while value < 2E9 :
    value += 1E-8;
```

This is an infinite loop
(because when *value* $= 10^9$, *value* $+ 10^{-8} =$ *value*)

Numerical instabilities are to be killed to predict weather, simulate a car crash or control a nuclear power plant

(but this is all *ways* beyond our goal this year ;)

Conclusion of this chapter

What tech guys tend to do when submitted a problem

- They code it directly, and rewrite everything once they understood
- And rewrite everything to improve performance
- And rewrite everything when the code needs to evolve

Conclusion of this chapter

What tech guys tend to do when submitted a problem

- They code it directly, and rewrite everything once they understood
- And rewrite everything to improve performance
- And rewrite everything when the code needs to evolve

What managers tend to do when submitted a problem

- They write up a long and verbose specification
- They struggle with the compiler in vain
- Then they pay a tech guy (and pay too much since they don't get the grasp)

Conclusion of this chapter

What tech guys tend to do when submitted a problem

- They code it directly, and rewrite everything once they understood
- And rewrite everything to improve performance
- And rewrite everything when the code needs to evolve

What managers tend to do when submitted a problem

- They write up a long and verbose specification
- They struggle with the compiler in vain
- Then they pay a tech guy (and pay too much since they don't get the grasp)

What theoreticians tend to do when submitted a problem

- They write a terse but formal specification
- They write an algorithm, and prove its optimality
  (the algorithm never gets coded)

Conclusion of this chapter

What tech guys tend to do when submitted a problem

- They code it directly, and rewrite everything once they understood
- And rewrite everything to improve performance
- And rewrite everything when the code needs to evolve

What managers tend to do when submitted a problem

- They write up a long and verbose specification
- They struggle with the compiler in vain
- Then they pay a tech guy (and pay too much since they don't get the grasp)

What theoreticians tend to do when submitted a problem

- They write a terse but formal specification
- They write an algorithm, and prove its optimality
  (the algorithm never gets coded)

What good programmers do when submitted a problem

- They write a clear specification
- They come up with a clean design
- They devise efficient data structures and algorithms
- Then (and only then), they write a clean and efficient code
- They ensure that the program does what it is supposed to do

Choice criteria between algorithms

Correctness

- Provides the right answer
- This crucial issue is delayed a bit further

Simplicity

- Keep it simple, silly
- Simple programs can evolve (problems and client's wishes often do)
- Rube Goldberg's machines cannot evolve

Efficiency

- Run fast, use little memory, dissipate little energy
- Asymptotic complexity must remain polynomial
- Note that you cannot have a decent complexity with the wrong data structure
- You still want to test the actual performance of your code in practice

Numerical stability

- Small change in input does not change output
- Advanced issue, critical for numerical simulations (but beyond our scope)

Second Chapter

Iterative Sorting Algorithms

Sorting Problem Specification

Input data

- A sequence of N comparable items $< a_1, a_2, a_3, \ldots, a_N >$
- Items are *comparable* iff $\forall a, b$ in set, either $\underline{a < b}$ or $\underline{a > b}$ or $\underline{a = b}$

Result

- Permutation[1] $< a'_1, a'_2, a'_3, \ldots, a'_N >$ so that: $a'_1 \leq a'_2 \leq a'_3 \leq \ldots \leq a'_N$

Sorting complex items

- For example, if items represent students, they encompass name, class, grade
- Key: value used for the sort
- Extra data: other data associated to items, permuted along with the keys

Problem simplification

- We assume that items are chars or integers to be sorted in ascending order (no loss of generality)

Memory consideration

- Sort *in place*, without any auxiliary List. Memory complexity: O(1)

[1] reordering

Selection Sort

Big lines

- First get the smallest value, and put it in first position
- Then get the second smallest value, and put it in second position
- and so on for all values

Example:

U	N	S	O	R	T	E	D

```
for (i <- 0 to length-1)
{var minpos=i for (j <- i
to length-1) {if (tab(j) <
tab(minpos)) {minpos = j
}}temp=tab(i) tab(i)=tab(minpos)
tab(minpos)=temp }
```

Selection Sort

Big lines

- First get the smallest value, and put it in first position
- Then get the second smallest value, and put it in second position
- and so on for all values

Example:

U	N	S	O	R	T	E	D

```
for (i <- 0 to length-1)
{var minpos=i for (j <- i
to length-1) {if (tab(j) <
tab(minpos)) {minpos = j
}}temp=tab(i) tab(i)=tab(minpos)
tab(minpos)=temp }
```

Selection Sort

Big lines

- First get the smallest value, and put it in first position
- Then get the second smallest value, and put it in second position
- and so on for all values

Example:

U	N	S	O	R	T	E	D
D	N	S	O	R	T	E	U

```
for (i <- 0 to length-1)
{var minpos=i for (j <- i
to length-1) {if (tab(j) <
tab(minpos)) {minpos = j
}}temp=tab(i) tab(i)=tab(minpos)
tab(minpos)=temp }
```

Selection Sort

Big lines

- First get the smallest value, and put it in first position
- Then get the second smallest value, and put it in second position
- and so on for all values

Example:

U	N	S	O	R	T	E	D
D	N	S	O	R	T	E	U
D	E	S	O	R	T	N	U

```
for (i <- 0 to length-1)
{var minpos=i for (j <- i
to length-1) {if (tab(j) <
tab(minpos)) {minpos = j
}}temp=tab(i) tab(i)=tab(minpos)
tab(minpos)=temp }
```

Selection Sort

Big lines

- First get the smallest value, and put it in first position
- Then get the second smallest value, and put it in second position
- and so on for all values

Example:

U	N	S	O	R	T	E	D
D	N	S	O	R	T	E	U
D	E	S	O	R	T	N	U
D	E	N	O	R	T	S	U

```
for (i <- 0 to length-1)
{var minpos=i for (j <- i
to length-1) {if (tab(j) <
tab(minpos)) {minpos = j
}}temp=tab(i) tab(i)=tab(minpos)
tab(minpos)=temp }
```

Selection Sort

Big lines

- First get the smallest value, and put it in first position
- Then get the second smallest value, and put it in second position
- and so on for all values

Example:

U	N	S	O	R	T	E	D
D	N	S	O	R	T	E	U
D	E	S	O	R	T	N	U
D	E	N	O	R	T	S	U
D	E	N	O	R	T	S	U

```
for (i <- 0 to length-1)
{var minpos=i for (j <- i
to length-1) {if (tab(j) <
tab(minpos)) {minpos = j
}}temp=tab(i) tab(i)=tab(minpos)
tab(minpos)=temp }
```

Selection Sort

Big lines

- First get the smallest value, and put it in first position
- Then get the second smallest value, and put it in second position
- and so on for all values

Example:

U	N	S	O	R	T	E	D
D	N	S	O	R	T	E	U
D	E	S	O	R	T	N	U
D	E	N	O	R	T	S	U
D	E	N	O	R	T	S	U
D	E	N	O	R	T	S	U

```
for (i <- 0 to length-1)
{var minpos=i for (j <- i
to length-1) {if (tab(j) <
tab(minpos)) {minpos = j
}}temp=tab(i) tab(i)=tab(minpos)
tab(minpos)=temp }
```

Selection Sort

Big lines

- First get the smallest value, and put it in first position
- Then get the second smallest value, and put it in second position
- and so on for all values

Example:

U	N	S	O	R	T	E	D
D	N	S	O	R	T	E	U
D	E	S	O	R	T	N	U
D	E	N	O	R	T	S	U
D	E	N	O	R	T	S	U
D	E	N	O	R	T	S	U
D	E	N	O	R	S	T	U

```
for (i <- 0 to length-1)
{var minpos=i for (j <- i
to length-1) {if (tab(j) <
tab(minpos)) {minpos = j
}}temp=tab(i) tab(i)=tab(minpos)
tab(minpos)=temp }
```

Selection Sort

Big lines

- First get the smallest value, and put it in first position
- Then get the second smallest value, and put it in second position
- and so on for all values

Example:

U	N	S	O	R	T	E	D
D	N	S	O	R	T	E	U
D	E	S	O	R	T	N	U
D	E	N	O	R	T	S	U
D	E	N	O	R	T	S	U
D	E	N	O	R	T	S	U
D	E	N	O	R	S	T	U
D	E	N	O	R	S	T	U

```
for (i <- 0 to length-1)
{var minpos=i for (j <- i
to length-1) {if (tab(j) <
tab(minpos)) {minpos = j
}}temp=tab(i) tab(i)=tab(minpos)
tab(minpos)=temp }
```

Selection Sort

Big lines

- First get the smallest value, and put it in first position
- Then get the second smallest value, and put it in second position
- and so on for all values

Example:

U	N	S	O	R	T	E	D
D	N	S	O	R	T	E	U
D	E	S	O	R	T	N	U
D	E	N	O	R	T	S	U
D	E	N	O	R	T	S	U
D	E	N	O	R	T	S	U
D	E	N	O	R	S	T	U
D	E	N	O	R	S	T	U
D	E	N	O	R	S	T	U

```
for (i <- 0 to length-1)
{var minpos=i for (j <- i
to length-1) {if (tab(j) <
tab(minpos)) {minpos = j
}}temp=tab(i) tab(i)=tab(minpos)
tab(minpos)=temp }
```

Selection sort discussion

We apply a very generic approach here:

- Do right now what you can, delay the rest for later (put min first)
- Progressively converge to what you are looking for (sort the remaining)

```
for (i <- 0 to length-1) {
  var minpos=i \
  for (j <- i to length-1) {
    if (tab(j) < tab(minpos)) {
      minpos = j
    }
  }
  temp=tab(i)
  tab(i)=tab(minpos)
  tab(minpos)=temp
}
```

Selection sort discussion

We apply a very generic approach here:

- Do right now what you can, delay the rest for later (put min first)
- Progressively converge to what you are looking for (sort the remaining)

```
for (i <- 0 to length-1) {
  var minpos=i \
  for (j <- i to length-1) {
    if (tab(j) < tab(minpos)) {
      minpos = j
    }
  }
  temp=tab(i)
  tab(i)=tab(minpos)
  tab(minpos)=temp
}
```

Memory Analysis

- 2 extra variables
  (only one at the same time, actually)

$\Rightarrow$ Constant amount of extra memory

$\Rightarrow$ Space complexity is $O(1)$

- $O(1)$ is the smallest complexity $\rightsquigarrow \Theta(1)$

Selection sort discussion

We apply a very generic approach here:

- Do right now what you can, delay the rest for later (put min first)
- Progressively converge to what you are looking for (sort the remaining)

```
for (i <- 0 to length-1) {
  var minpos=i \
  for (j <- i to length-1) {
    if (tab(j) < tab(minpos)) {
      minpos = j
    }
  }
  temp=tab(i)
  tab(i)=tab(minpos)
  tab(minpos)=temp
}
```

Memory Analysis

- 2 extra variables
  (only one at the same time, actually)

$\Rightarrow$ Constant amount of extra memory

$\Rightarrow$ Space complexity is $O(1)$

- $O(1)$ is the smallest complexity $\leadsto \Theta(1)$

Time Analysis

- Forget about constant times, focus on loops!

Selection sort discussion

We apply a very generic approach here:

- Do right now what you can, delay the rest for later (put min first)
- Progressively converge to what you are looking for (sort the remaining)

```
for (i <- 0 to length-1) {
  var minpos=i \
  for (j <- i to length-1) {
    if (tab(j) < tab(minpos)) {
      minpos = j
    }
  }
  temp=tab(i)
  tab(i)=tab(minpos)
  tab(minpos)=temp
}
```

Memory Analysis

- 2 extra variables
  (only one at the same time, actually)

$\Rightarrow$ Constant amount of extra memory

$\Rightarrow$ Space complexity is $O(1)$

- $O(1)$ is the smallest complexity $\leadsto \Theta(1)$

Time Analysis

- Forget about constant times, focus on loops!
- Two interleaved loops which length is *at most* N

Selection sort discussion

We apply a very generic approach here:

- Do right now what you can, delay the rest for later (put min first)
- Progressively converge to what you are looking for (sort the remaining)

```
for (i <- 0 to length-1) {
  var minpos=i \
  for (j <- i to length-1) {
    if (tab(j) < tab(minpos)) {
      minpos = j
    }
  }
  temp=tab(i)
  tab(i)=tab(minpos)
  tab(minpos)=temp
}
```

Memory Analysis

- 2 extra variables
  (only one at the same time, actually)

$\Rightarrow$ Constant amount of extra memory

$\Rightarrow$ Space complexity is $O(1)$

- $O(1)$ is the smallest complexity $\rightsquigarrow \Theta(1)$

Time Analysis

- Forget about constant times, focus on loops!
- Two interleaved loops which length is *at most* N

$\Rightarrow$ Time complexity is $O(N^2)$

Finer analysis of selection sort's time performance

```
for (i <- 0 to length-1) {
  var minpos=i
  for (j <- i to length-1)
    if (tab(j) < tab(minpos))
      minpos = j
  temp=tab(i)
  tab(i)=tab(minpos)
  tab(minpos)=temp
}
```

Best case, worst case, average case

- No matter the order of the data, 'selection sort' does the same

$\Rightarrow$ $t_{min} = t_{max} = t_{avg}$

Finer analysis of selection sort's time performance

```
for (i <- 0 to length-1) {
  var minpos=i
  for (j <- i to length-1)
    if (tab(j) < tab(minpos))
      minpos = j
  temp=tab(i)
  tab(i)=tab(minpos)
  tab(minpos)=temp
}
```

Best case, worst case, average case

- No matter the order of the data, 'selection sort' does the same

$\Rightarrow$ $t_{min} = t_{max} = t_{avg}$

Counting steps more precisely (but only dominant term)

- $T(N) = \sum_{i \in [1,N]} \left(\sum_{j \in [i,N[} 1 \right)$

Finer analysis of selection sort's time performance

```
for (i <- 0 to length-1) {
  var minpos=i
  for (j <- i to length-1)
    if (tab(j) < tab(minpos))
      minpos = j
  temp=tab(i)
  tab(i)=tab(minpos)
  tab(minpos)=temp
}
```

Best case, worst case, average case

- No matter the order of the data, 'selection sort' does the same

$\Rightarrow$ $t_{min} = t_{max} = t_{avg}$

Counting steps more precisely (but only dominant term)

- $$T(N) = \sum_{i \in [1,N]} \left(\sum_{j \in [i,N[} 1 \right) = \sum_{i \in [1,N]} (N - i) = \sum_{i \in [1,N]} N - \sum_{i \in [1,N]} i$$
$$= N^2 - \frac{N \times (N+1)}{2} = N^2 - \frac{N^2 + N}{2} = \frac{1}{2}N^2 - \frac{1}{2}N = \frac{1}{2}(N^2 - N)$$

Finer analysis of selection sort's time performance

```
for (i <- 0 to length-1) {
  var minpos=i
  for (j <- i to length-1)
    if (tab(j) < tab(minpos))
      minpos = j
  temp=tab(i)
  tab(i)=tab(minpos)
  tab(minpos)=temp
}
```

Best case, worst case, average case

- No matter the order of the data, 'selection sort' does the same

$\Rightarrow$ $t_{min} = t_{max} = t_{avg}$

Counting steps more precisely (but only dominant term)

- $$T(N) = \sum_{i \in [1,N]} \left(\sum_{j \in [i,N[} 1 \right) = \sum_{i \in [1,N]} (N-i) = \sum_{i \in [1,N]} N - \sum_{i \in [1,N]} i$$
  $$= N^2 - \frac{N \times (N+1)}{2} = N^2 - \frac{N^2+N}{2} = \frac{1}{2}N^2 - \frac{1}{2}N = \frac{1}{2}(N^2 - N)$$
- Let's prove that $T(n) \in \Omega(n^2)$. For that, we want:
- $\exists c, n_0 / \forall N > n_0,$ $\boxed{\frac{1}{2}(N^2 - N) \geq cN^2}$

Finer analysis of selection sort's time performance

```
for (i <- 0 to length-1) {
  var minpos=i
  for (j <- i to length-1)
    if (tab(j) < tab(minpos))
      minpos = j
  temp=tab(i)
  tab(i)=tab(minpos)
  tab(minpos)=temp
}
```

Best case, worst case, average case

- No matter the order of the data, 'selection sort' does the same

$\Rightarrow t_{min} = t_{max} = t_{avg}$

Counting steps more precisely (but only dominant term)

- $$T(N) = \sum_{i \in [1,N]} \left(\sum_{j \in [i,N[} 1 \right) = \sum_{i \in [1,N]} (N-i) = \sum_{i \in [1,N]} N - \sum_{i \in [1,N]} i$$
  $$= N^2 - \frac{N \times (N+1)}{2} = N^2 - \frac{N^2+N}{2} = \frac{1}{2}N^2 - \frac{1}{2}N = \frac{1}{2}(N^2 - N)$$
- Let's prove that $T(n) \in \Omega(n^2)$. For that, we want:
- $\exists c, n_0 / \forall N > n_0, \boxed{\frac{1}{2}(N^2 - N) \geq cN^2} \Leftarrow \boxed{N^2 - N \geq 2cN^2} \Leftarrow \boxed{N - 1 \geq 2cN}$

Finer analysis of selection sort's time performance

```
for (i <- 0 to length-1) {
  var minpos=i
  for (j <- i to length-1)
    if (tab(j) < tab(minpos))
      minpos = j
  temp=tab(i)
  tab(i)=tab(minpos)
  tab(minpos)=temp
}
```

Best case, worst case, average case

- No matter the order of the data, 'selection sort' does the same

$\Rightarrow t_{min} = t_{max} = t_{avg}$

Counting steps more precisely (but only dominant term)

- $$T(N) = \sum_{i\in[1,N]} \left(\sum_{j\in[i,N[} 1 \right) = \sum_{i\in[1,N]} (N-i) = \sum_{i\in[1,N]} N - \sum_{i\in[1,N]} i$$
  $$= N^2 - \frac{N\times(N+1)}{2} = N^2 - \frac{N^2+N}{2} = \frac{1}{2}N^2 - \frac{1}{2}N = \frac{1}{2}(N^2-N)$$
- Let's prove that $T(n) \in \Omega(n^2)$. For that, we want:
- $\exists c, n_0 / \forall N > n_0,\ \boxed{\frac{1}{2}(N^2-N) \geq cN^2} \Leftarrow \boxed{N^2 - N \geq 2cN^2} \Leftarrow \boxed{N-1 \geq 2cN}$
- So, we want $\exists c, n_0 / \forall N > n_0, N \geq \frac{1}{1-2c}$

Finer analysis of selection sort's time performance

```
for (i <- 0 to length-1) {
  var minpos=i
  for (j <- i to length-1)
    if (tab(j) < tab(minpos))
      minpos = j
  temp=tab(i)
  tab(i)=tab(minpos)
  tab(minpos)=temp
}
```

Best case, worst case, average case

- No matter the order of the data, 'selection sort' does the same

$\Rightarrow t_{min} = t_{max} = t_{avg}$

Counting steps more precisely (but only dominant term)

- $$T(N) = \sum_{i \in [1,N]} \left(\sum_{j \in [i,N[} 1 \right) = \sum_{i \in [1,N]} (N - i) = \sum_{i \in [1,N]} N - \sum_{i \in [1,N]} i$$
$$= N^2 - \frac{N \times (N+1)}{2} = N^2 - \frac{N^2+N}{2} = \frac{1}{2}N^2 - \frac{1}{2}N = \frac{1}{2}(N^2 - N)$$
- Let's prove that $T(n) \in \Omega(n^2)$. For that, we want:
- $\exists c, n_0 / \forall N > n_0, \boxed{\frac{1}{2}(N^2 - N) \geq cN^2} \Leftarrow \boxed{N^2 - N \geq 2cN^2} \Leftarrow \boxed{N - 1 \geq 2cN}$
- So, we want $\exists c, n_0 / \forall N > n_0, N \geq \frac{1}{1-2c}$
- Let's take anything for c $(\neq \frac{1}{2})$, and $n_0 = \frac{1}{1-2c}$. Trivially gives what we want.

$T(n) \in \Theta(n^2)$

Insertion Sort

How do you sort your card deck?

- No human would apply *selection sort* to sort a deck!

Algorithm used most of the time to sort a card deck:

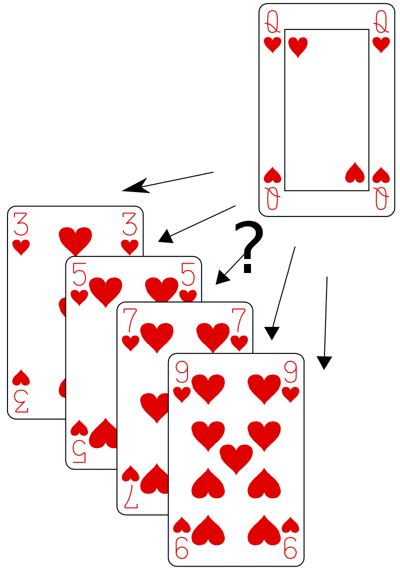
1. If the cards #1 and #2 need to be swapped, do it
2. Insert card #3 at its position in the [1,2] part of the deck
3. Insert card #4 at its position in the [1,3] part of the deck

   ...

Insertion Sort

How do you sort your card deck?

- No human would apply *selection sort* to sort a deck!



Algorithm used most of the time to sort a card deck:

1. If the cards #1 and #2 need to be swapped, do it
2. Insert card #3 at its position in the [1,2] part of the deck
3. Insert card #4 at its position in the [1,3] part of the deck

   ...

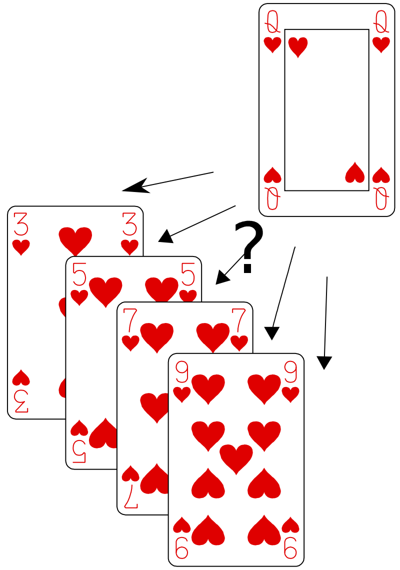
Finding the common pattern

- Step n ($\geq$ 2) is “insert card #(n+1) into [1,n]”

Insertion Sort

How do you sort your card deck?

- No human would apply *selection sort* to sort a deck!



Algorithm used most of the time to sort a card deck:

1. If the cards #1 and #2 need to be swapped, do it
2. Insert card #3 at its position in the [1,2] part of the deck
3. Insert card #4 at its position in the [1,3] part of the deck

   . . .

Finding the common pattern

- Step n ($\geq$ 2) is "insert card #(n+1) into [1,n]"
- Step 1 = insert the 2. card into [1,1]
- We may add a Step 0 to generalize the pattern (that's a no-op)

Insertion Sort

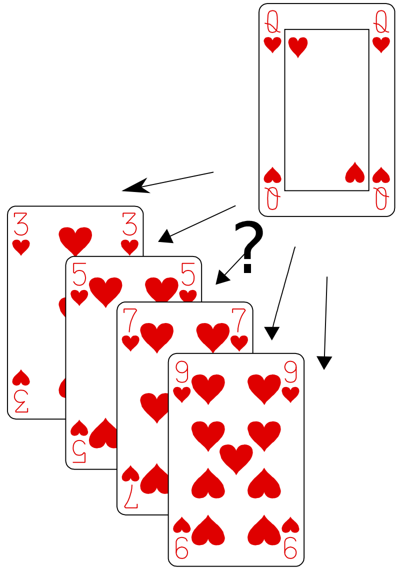
How do you sort your card deck?

- No human would apply *selection sort* to sort a deck!

Algorithm used most of the time to sort a card deck:

1. If the cards #1 and #2 need to be swapped, do it
2. Insert card #3 at its position in the [1,2] part of the deck
3. Insert card #4 at its position in the [1,3] part of the deck

   . . .



This is *Insertion Sort*

U	N	S	O	R	T	E	D

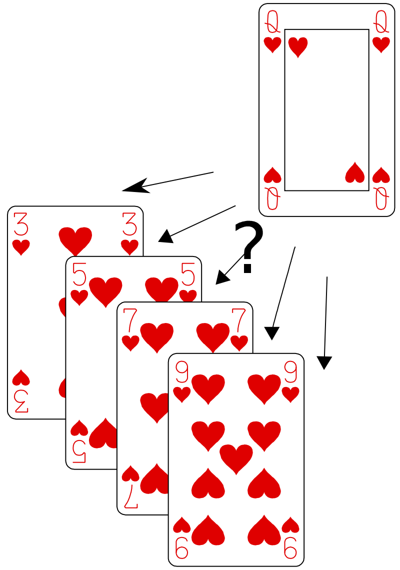
Finding the common pattern

- Step n (≥ 2) is “insert card #(n+1) into [1,n]”
- Step 1 = insert the 2. card into [1,1]
- We may add a Step 0 to generalize the pattern (that's a no-op)

Insertion Sort

How do you sort your card deck?

- No human would apply *selection sort* to sort a deck!



Algorithm used most of the time to sort a card deck:

1. If the cards #1 and #2 need to be swapped, do it
2. Insert card #3 at its position in the [1,2] part of the deck
3. Insert card #4 at its position in the [1,3] part of the deck

   . . .

Finding the common pattern

- Step n (≥ 2) is "insert card #(n+1) into [1,n]"
- Step 1 = insert the 2. card into [1,1]
- We may add a Step 0 to generalize the pattern (that's a no-op)

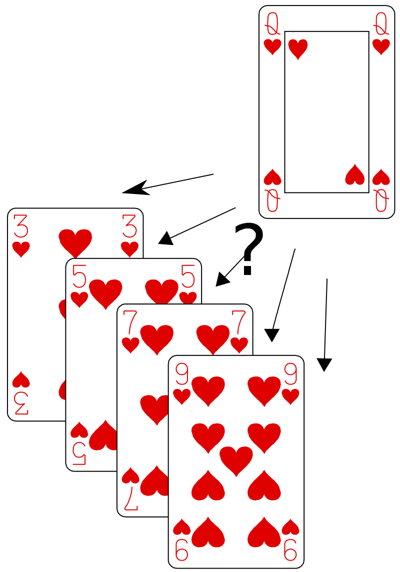
This is *Insertion Sort*

U	N	S	O	R	T	E	D
U	N	S	O	R	T	E	D

Insertion Sort

How do you sort your card deck?

- No human would apply *selection sort* to sort a deck!



Algorithm used most of the time to sort a card deck:

1. If the cards #1 and #2 need to be swapped, do it
2. Insert card #3 at its position in the [1,2] part of the deck
3. Insert card #4 at its position in the [1,3] part of the deck

. . .

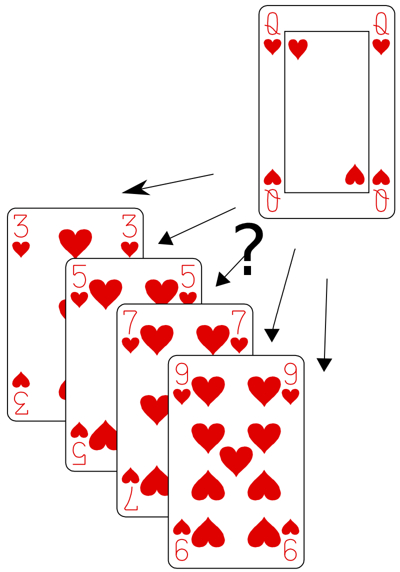
This is *Insertion Sort*

U	N	S	O	R	T	E	D
U	N	S	O	R	T	E	D
N	U	S	O	R	T	E	D

Finding the common pattern

- Step n ($\geq$ 2) is "insert card #(n+1) into [1,n]"
- Step 1 = insert the 2. card into [1,1]
- We may add a Step 0 to generalize the pattern (that's a no-op)

Insertion Sort



How do you sort your card deck?

- No human would apply *selection sort* to sort a deck!

Algorithm used most of the time to sort a card deck:

1. If the cards #1 and #2 need to be swapped, do it
2. Insert card #3 at its position in the [1,2] part of the deck
3. Insert card #4 at its position in the [1,3] part of the deck

   ...

Finding the common pattern

- Step n ($\geq$ 2) is "insert card #(n+1) into [1,n]"
- Step 1 = insert the 2. card into [1,1]
- We may add a Step 0 to generalize the pattern (that's a no-op)

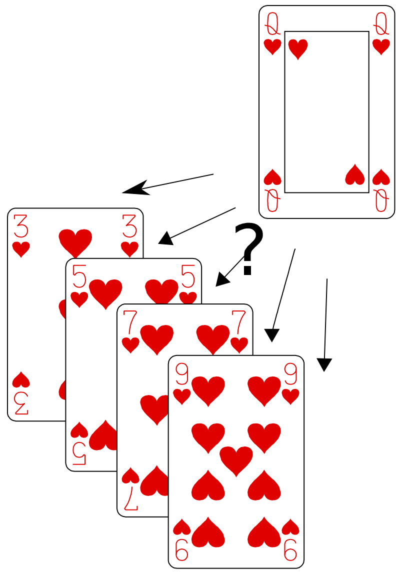
This is *Insertion Sort*

U	N	S	O	R	T	E	D
U	N	S	O	R	T	E	D
N	U	S	O	R	T	E	D
N	S	U	O	R	T	E	D

Insertion Sort

How do you sort your card deck?

- No human would apply *selection sort* to sort a deck!



Algorithm used most of the time to sort a card deck:

1. If the cards #1 and #2 need to be swapped, do it
2. Insert card #3 at its position in the [1,2] part of the deck
3. Insert card #4 at its position in the [1,3] part of the deck

   ...

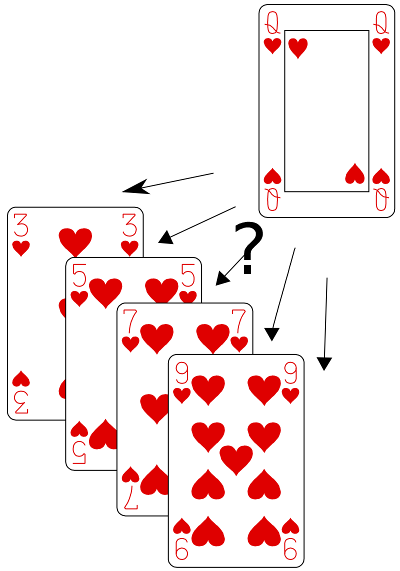
Finding the common pattern

- Step n ($\geq$ 2) is "insert card #(n+1) into [1,n]"
- Step 1 = insert the 2. card into [1,1]
- We may add a Step 0 to generalize the pattern (that's a no-op)

This is *Insertion Sort*

U	N	S	O	R	T	E	D
U	N	S	O	R	T	E	D
N	U	S	O	R	T	E	D
N	S	U	O	R	T	E	D
N	O	S	U	R	T	E	D

Insertion Sort



How do you sort your card deck?

- No human would apply *selection sort* to sort a deck!

Algorithm used most of the time to sort a card deck:

1. If the cards #1 and #2 need to be swapped, do it
2. Insert card #3 at its position in the [1,2] part of the deck
3. Insert card #4 at its position in the [1,3] part of the deck

   ...

Finding the common pattern

- Step n (≥ 2) is "insert card #(n+1) into [1,n]"
- Step 1 = insert the 2. card into [1,1]
- We may add a Step 0 to generalize the pattern (that's a no-op)

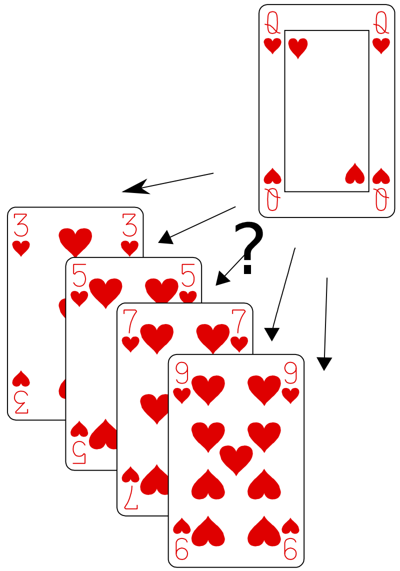
This is *Insertion Sort*

U	N	S	O	R	T	E	D
U	N	S	O	R	T	E	D
N	U	S	O	R	T	E	D
N	S	U	O	R	T	E	D
N	O	S	U	R	T	E	D
N	O	R	S	U	T	E	D

Insertion Sort

How do you sort your card deck?

- No human would apply *selection sort* to sort a deck!



Algorithm used most of the time to sort a card deck:

1. If the cards #1 and #2 need to be swapped, do it
2. Insert card #3 at its position in the [1,2] part of the deck
3. Insert card #4 at its position in the [1,3] part of the deck

   ...

Finding the common pattern

- Step n (≥ 2) is "insert card #(n+1) into [1,n]"
- Step 1 = insert the 2. card into [1,1]
- We may add a Step 0 to generalize the pattern (that's a no-op)

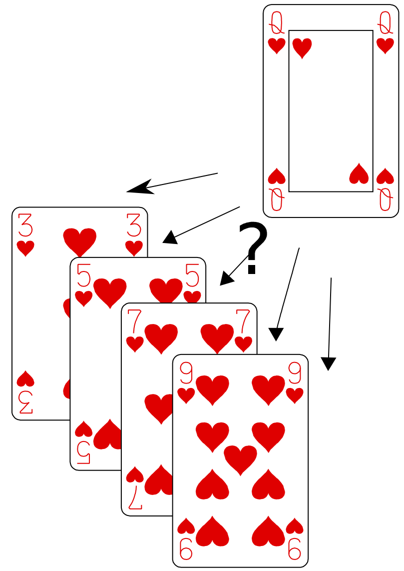
This is *Insertion Sort*

U	N	S	O	R	T	E	D
U	N	S	O	R	T	E	D
N	U	S	O	R	T	E	D
N	S	U	O	R	T	E	D
N	O	S	U	R	T	E	D
N	O	R	S	U	T	E	D
N	O	R	S	T	U	E	D

Insertion Sort

How do you sort your card deck?

- No human would apply *selection sort* to sort a deck!



Algorithm used most of the time to sort a card deck:

1. If the cards #1 and #2 need to be swapped, do it
2. Insert card #3 at its position in the [1,2] part of the deck
3. Insert card #4 at its position in the [1,3] part of the deck

   ...

Finding the common pattern

- Step n (≥ 2) is "insert card #(n+1) into [1,n]"
- Step 1 $=$ insert the 2. card into [1,1]
- We may add a Step 0 to generalize the pattern (that's a no-op)

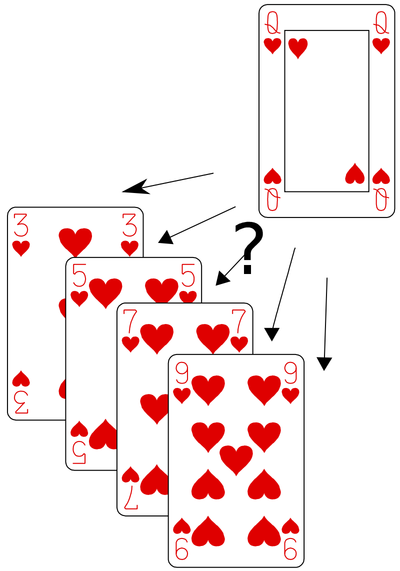
This is *Insertion Sort*

U	N	S	O	R	T	E	D
U	N	S	O	R	T	E	D
N	U	S	O	R	T	E	D
N	S	U	O	R	T	E	D
N	O	S	U	R	T	E	D
N	O	R	S	U	T	E	D
N	O	R	S	T	U	E	D
E	N	O	R	S	T	U	D

Insertion Sort

How do you sort your card deck?

- No human would apply *selection sort* to sort a deck!



Algorithm used most of the time to sort a card deck:

1. If the cards #1 and #2 need to be swapped, do it
2. Insert card #3 at its position in the [1,2] part of the deck
3. Insert card #4 at its position in the [1,3] part of the deck

   . . .

Finding the common pattern

- Step n ($\geq$ 2) is "insert card #(n+1) into [1,n]"
- Step 1 = insert the 2. card into [1,1]
- We may add a Step 0 to generalize the pattern (that's a no-op)

Algorithm big lines

```
For each element
  Find insertion position
  Move element to position
```

This is *Insertion Sort*

U	N	S	O	R	T	E	D
U	N	S	O	R	T	E	D
N	U	S	O	R	T	E	D
N	S	U	O	R	T	E	D
N	O	S	U	R	T	E	D
N	O	R	S	U	T	E	D
N	O	R	S	T	U	E	D
E	N	O	R	S	T	U	D
D	E	N	O	R	S	T	U

Writing the insertion sort algorithm

Fleshing the big lines

```
For each element
  Find insertion point
  Move element to position
```

- Finding the insertion point is easy (searching loop)
- Moving to position is a bit harder: “make room”
- We have to *shift* elements one after the other

Before:	N	S	U	O	R	T	E	D
After:	N	O	S	U	R	T	E	D

Writing the insertion sort algorithm

Fleshing the big lines

```
For each element
  Find insertion point
  Move element to position
```

- Finding the insertion point is easy (searching loop)
- Moving to position is a bit harder: “make room”
- We have to *shift* elements one after the other

	0	1	2	3	4	5	6	7		
step 1.	N	S	U		R	T	E	D	tmp	O

Before:	N	S	U	O	R	T	E	D
After:	N	O	S	U	R	T	E	D

Writing the insertion sort algorithm

Fleshing the big lines

```
For each element
  Find insertion point
  Move element to position
```

- Finding the insertion point is easy (searching loop)
- Moving to position is a bit harder: "make room"
- We have to *shift* elements one after the other

Before:	N	S	U	O	R	T	E	D
After:	N	O	S	U	R	T	E	D

	0	1	2	3	4	5	6	7		
step 1.	N	S	U		R	T	E	D	tmp	O
step 2.	N	S		U	R	T	E	D	tmp	O

Writing the insertion sort algorithm

Fleshing the big lines

```
For each element
  Find insertion point
  Move element to position
```

- Finding the insertion point is easy (searching loop)
- Moving to position is a bit harder: "make room"
- We have to *shift* elements one after the other

Before:	N	S	U	O	R	T	E	D
After:	N	O	S	U	R	T	E	D

	0	1	2	3	4	5	6	7		
step 1.	N	S	U		R	T	E	D	tmp	O
step 2.	N	S		U	R	T	E	D	tmp	O
step 3.	N		S	U	R	T	E	D	tmp	O

Writing the insertion sort algorithm

Fleshing the big lines

```
For each element
  Find insertion point
  Move element to position
```

- Finding the insertion point is easy (searching loop)
- Moving to position is a bit harder: “make room”
- We have to *shift* elements one after the other

Before:	N	S	U	O	R	T	E	D
After:	N	O	S	U	R	T	E	D

	0	1	2	3	4	5	6	7		
step 1.	N	S	U		R	T	E	D	tmp	O
step 2.	N	S		U	R	T	E	D	tmp	O
step 3.	N		S	U	R	T	E	D	tmp	O
step 4.	N	O	S	U	R	T	E	D	tmp	

Writing the insertion sort algorithm

Fleshing the big lines

```
For each element
  Find insertion point
  Move element to position
```

- Finding the insertion point is easy (searching loop)
- Moving to position is a bit harder: "make room"
- We have to *shift* elements one after the other

Before:	N	S	U	O	R	T	E	D
After:	N	O	S	U	R	T	E	D

	0	1	2	3	4	5	6	7		
step 1.	N	S	U		R	T	E	D	tmp	O
step 2.	N	S		U	R	T	E	D	tmp	O
step 3.	N		S	U	R	T	E	D	tmp	O
step 4.	N	O	S	U	R	T	E	D	tmp	

- Shifting elements induce a loop also
- We can do both searching insertion point and shifting at the same time

Writing the insertion sort algorithm

Fleshing the big lines

```
For each element
  Find insertion point
  Move element to position
```

- Finding the insertion point is easy (searching loop)
- Moving to position is a bit harder: "make room"
- We have to *shift* elements one after the other

Before:	N	S	U	O	R	T	E	D
After:	N	O	S	U	R	T	E	D

	0	1	2	3	4	5	6	7		
step 1.	N	S	U		R	T	E	D	tmp	O
step 2.	N	S		U	R	T	E	D	tmp	O
step 3.	N		S	U	R	T	E	D	tmp	O
step 4.	N	O	S	U	R	T	E	D	tmp	

- Shifting elements induce a loop also
- We can do both searching insertion point and shifting at the same time

```
/* save current value (it this case, that's O) */

/* shift to right any element on the left being smaller than tmp */




/* put tmp in cleared position */
}
```

Writing the insertion sort algorithm

Fleshing the big lines

```
For each element
  Find insertion point
  Move element to position
```

- Finding the insertion point is easy (searching loop)
- Moving to position is a bit harder: “make room”
- We have to *shift* elements one after the other

Before:	N	S	U	O	R	T	E	D
After:	N	O	S	U	R	T	E	D

	0	1	2	3	4	5	6	7		
step 1.	N	S	U		R	T	E	D	tmp	O
step 2.	N	S		U	R	T	E	D	tmp	O
step 3.	N		S	U	R	T	E	D	tmp	O
step 4.	N	O	S	U	R	T	E	D	tmp	

- Shifting elements induce a loop also
- We can do both searching insertion point and shifting at the same time

```
for (i <- 1 to length-1) {/* i: boundary between unsorted/sorted areas*/
  /* save current value (it this case, that's O) */

  /* shift to right any element on the left being smaller than tmp */





  /* put tmp in cleared position */

}
```

Writing the insertion sort algorithm

Fleshing the big lines

```
For each element
  Find insertion point
  Move element to position
```

- Finding the insertion point is easy (searching loop)
- Moving to position is a bit harder: "make room"
- We have to *shift* elements one after the other

Before:	N	S	U	O	R	T	E	D
After:	N	O	S	U	R	T	E	D

	0	1	2	3	4	5	6	7		
step 1.	N	S	U		R	T	E	D	tmp	O
step 2.	N	S		U	R	T	E	D	tmp	O
step 3.	N		S	U	R	T	E	D	tmp	O
step 4.	N	O	S	U	R	T	E	D	tmp	

- Shifting elements induce a loop also
- We can do both searching insertion point and shifting at the same time

```
for (i <- 1 to length-1) {/* i: boundary between unsorted/sorted areas*/
  /* save current value (it this case, that's O) */
  val tmp = tab(i)
  /* shift to right any element on the left being smaller than tmp */





  /* put tmp in cleared position */

}
```

Writing the insertion sort algorithm

Fleshing the big lines

```
For each element
  Find insertion point
  Move element to position
```

- Finding the insertion point is easy (searching loop)
- Moving to position is a bit harder: "make room"
- We have to *shift* elements one after the other

Before:	N	S	U	O	R	T	E	D
After:	N	O	S	U	R	T	E	D

	0	1	2	3	4	5	6	7		
step 1.	N	S	U		R	T	E	D	tmp	O
step 2.	N	S		U	R	T	E	D	tmp	O
step 3.	N		S	U	R	T	E	D	tmp	O
step 4.	N	O	S	U	R	T	E	D	tmp	

- Shifting elements induce a loop also
- We can do both searching insertion point and shifting at the same time

```
for (i <- 1 to length-1) {/* i: boundary between unsorted/sorted areas*/
  /* save current value (it this case, that's O) */
  val tmp = tab(i)
  /* shift to right any element on the left being smaller than tmp */
  var j = i
  while (j>0 && tab(j-1)>tmp) {/* while previous cell exists and is bigger */
    tab(j) = tab(j-1) /* copy that element */
    j = j -1 /* consider the next element */
  }
  /* put tmp in cleared position */

}
```

Writing the insertion sort algorithm

Fleshing the big lines

For each element
  Find insertion point
  Move element to position

- Finding the insertion point is easy (searching loop)
- Moving to position is a bit harder: “make room”
- We have to *shift* elements one after the other

Before:	N	S	U	O	R	T	E	D
After:	N	O	S	U	R	T	E	D

	0	1	2	3	4	5	6	7		
step 1.	N	S	U		R	T	E	D	tmp	O
step 2.	N	S		U	R	T	E	D	tmp	O
step 3.	N		S	U	R	T	E	D	tmp	O
step 4.	N	O	S	U	R	T	E	D	tmp	

- Shifting elements induce a loop also
- We can do both searching insertion point and shifting at the same time

```
for (i <- 1 to length-1) {/* i: boundary between unsorted/sorted areas*/
  /* save current value (it this case, that's O) */
  val tmp = tab(i)
  /* shift to right any element on the left being smaller than tmp */
  var j = i
  while (j>0 && tab(j-1)>tmp) {/* while previous cell exists and is bigger */
    tab(j) = tab(j-1) /* copy that element */
    j = j -1 /* consider the next element */
  }
  /* put tmp in cleared position */
  tab(j)=tmp
}
```

Bubble Sort

- All these sort algorithms are quite difficult to write. Can we do simpler?
- Like “while it’s not sorted, sort it a bit”

Bubble Sort

- All these sort algorithms are quite difficult to write. Can we do simpler?
- Like “while it’s not sorted, sort it a bit”

Detecting that it’s sorted

```
for (i <- 0 to length-2)
   /* if these two values are badly sorted */
   if (tab(i)>tab(i+1))
      return false
return true
```

Bubble Sort

- All these sort algorithms are quite difficult to write. Can we do simpler?
- Like “while it’s not sorted, sort it a bit”

Detecting that it’s sorted

```
for (i <- 0 to length-2)
   /* if these two values are badly sorted */
   if (tab(i)>tab(i+1))
      return false
return true
```

How to “sort a bit?”

- We may just swap these two values

```
val tmp=tab(i)
tab(i)=tab(i+1)
tab(i+1)=tmp
```

Bubble Sort

- All these sort algorithms are quite difficult to write. Can we do simpler?
- Like "while it's not sorted, sort it a bit"

Detecting that it's sorted

```
for (i <- 0 to length-2)
   /* if these two values are badly sorted */
   if (tab(i)>tab(i+1))
      return false
return true
```

How to "sort a bit?"

- We may just swap these two values

```
val tmp=tab(i)
tab(i)=tab(i+1)
tab(i+1)=tmp
```

All together

- Add boolean variable to check whether it sorted

```
var swapped = true
while (swapped) { /* until we do one traversal without swap */
   swapped = false
   for (i <- 0 to length-2)
      if (tab(i) > tab(i+1)) { /* if these 2 values are badly sorted */
         /* swap them */
         val tmp = tab(i)
         tab(i) = tab(i+1)
         tab(i+1) = tmp
         /* and remember we swapped something */
         swapped = true
      }
} /* repeat until a traversal without swapping */
```

Conclusion on Iterative Sorting Algorithms

Cost Theoretical Analysis

Amount of comparisons	Best Case	Average Case	Worst Case
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$

Which is the best in practice?

- We will explore practical performance during the lab
- But in practice, bubble sort is awfully slow and should never be used

(this ends the first lecture)

Conclusion on Iterative Sorting Algorithms

Cost Theoretical Analysis

Amount of comparisons	Best Case	Average Case	Worst Case
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$

Which is the best in practice?

- We will explore practical performance during the lab
- But in practice, bubble sort is awfully slow and should never be used

Is it optimal?

- The lower bound is $\Omega(n \log(n))$ – cf. TD lab
- Some other algorithms achieve it (Quick Sort, Merge Sort)
- We come back on these next week

(this ends the first lecture)

Third Chapter

Recursion

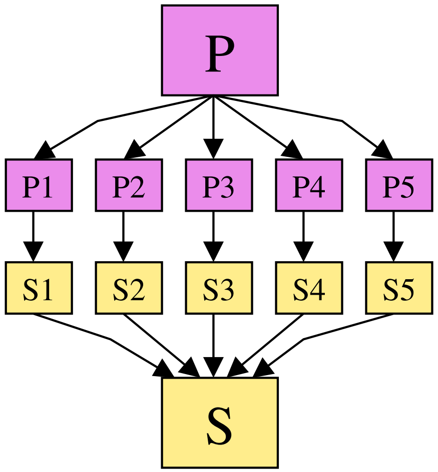
Divide & Conquer

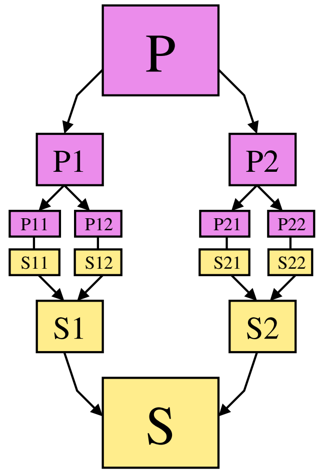
Classical Algorithmic Pattern

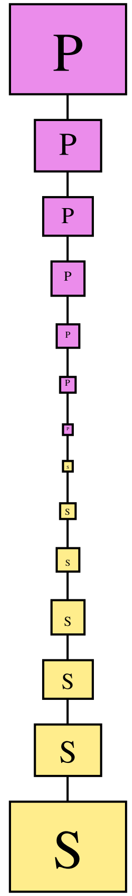
- When the problem is too complex to be solved directly, decompose it

When/How is it applicable?

1. Divide: Decompose problem into (simpler/smaller) sub-problems
2. Conquer: Solve sub-problems
3. Glue: Combine solutions of sub-problems to a solution as a whole







> *You don't have to see the whole staircase, just take the first step.*
> – Martin Luther King

Recursion

Divide & Conquer + sub-problems similar to big one

Recursion

Divide & Conquer + sub-problems similar to big one

Recursive object

- Defined using itself

Recursion

Divide & Conquer + sub-problems similar to big one

Recursive object

- Defined using itself
- Examples:
  - $U(n) = 3 \times U(n-1) + 1 \; ; \; U(0) = 1$
  - Char string = either a char followed by a string, or empty string
- Often possible to rewrite the object, in a non-recursive way (said *iterative way*)

Recursion

Divide & Conquer + sub-problems similar to big one

Recursive object

- Defined using itself
- Examples:
  - $U(n) = 3 \times U(n-1) + 1$; $U(0) = 1$
  - Char string = either a char followed by a string, or empty string
- Often possible to rewrite the object, in a non-recursive way (said *iterative way*)

Base case(s)

- Trivial cases that can be solved directly
- Avoids infinite loop

When the base case is missing. . .

Classical Aphorism

To understand recursion,
you first have to understand recursion

This is naturally to be avoided in algorithms

When the base case is missing. . .

There's a Hole in the Bucket (traditional)

There's a hole in the bucket, dear Liza, a hole.
So fix it dear Henry, dear Henry, fix it.
With what should I fix it, dear Liza, with what?
With straw, dear Henry, dear Henry, with straw.
The straw is too long, dear Liza, too long.
So cut it dear Henry, dear Henry, cut it!
With what should I cut it, dear Liza, with what?
Use the hatchet, dear Henry, the hatchet.
The hatchet's too dull, dear Liza, too dull.
So sharpen it dear Henry, dear Henry, sharpen it!
With what should I sharpen, dear Liza, with what?
Use the stone, dear Henry, dear Henry, the stone.
The stone is too dry, dear Liza, too dry.
So wet it dear Henry, dear Henry, wet it.
With what should I wet it, dear Liza, with what?
With water, dear Henry, dear Henry, water.
With what should I carry it dear Liza, with what?
Use the bucket, dear Henry, dear Henry, the bucket!

There's a hole in the bucket, dear Liza, a hole.

Classical Aphorism

To understand recursion,
you first have to understand recursion

This is naturally to be avoided in algorithms

When the base case is missing. . .

There's a Hole in the Bucket (traditional)

There's a hole in the bucket, dear Liza, a hole.
So fix it dear Henry, dear Henry, fix it.
With what should I fix it, dear Liza, with what?
With straw, dear Henry, dear Henry, with straw.
The straw is too long, dear Liza, too long.
So cut it dear Henry, dear Henry, cut it!
With what should I cut it, dear Liza, with what?
Use the hatchet, dear Henry, the hatchet.
The hatchet's too dull, dear Liza, too dull.
So sharpen it dear Henry, dear Henry, sharpen it!
With what should I sharpen, dear Liza, with what?
Use the stone, dear Henry, dear Henry, the stone.
The stone is too dry, dear Liza, too dry.
So wet it dear Henry, dear Henry, wet it.
With what should I wet it, dear Liza, with what?
With water, dear Henry, dear Henry, water.
With what should I carry it dear Liza, with what?
Use the bucket, dear Henry, dear Henry, the bucket!
There's a hole in the bucket, dear Liza, a hole.

Classical Aphorism

To understand recursion,
you first have to understand recursion

Recursive Acronyms

- GNU is Not Unix
- PHP: Hypertext Preprocessor
- PNG's Not GIF
- Wine Is Not an Emulator
- Visa International Service Association
- HIRD of Unix-Replacing Daemons
  Hurd of Interfaces Representing Depth
- Your Own Personal YOPY

This is naturally to be avoided in algorithms

In Mathematics: Natural Numbers and Induction

Peano postulates (1880)

Defines the set of natural integers $\mathbb{N}$

1. 0 is a natural number
2. If n is natural, its successor (noted $n + 1$) also
3. There is no number x so that $x + 1 = 0$
4. Distinct numbers have distinct successors ($x \neq y \Leftrightarrow x + 1 \neq y + 1$)
5. If a property holds (i) for 0 (ii) for each number's successor,
   it then holds for any number

Proof by Induction

- One shows that the property holds for 0 (or other base case)
- One shows that when it holds for n, it then holds for $n + 1$
- This shows that it holds for any number

In Computer Science

Two twin notions

- Functions and procedures defined recursively (generative recursion)
- Data structures defined recursively (structural recursion)

Naturally, recursive functions are well fitted to recursive data structures

This is an **algorithm** characteristic

- No problem is intrinsically recursive
- Some problems *easier* or more natural to solve recursively
- Every recursive algorithm can be *derecursived*

Third Chapter

Recursion

- Introduction

- Principles of Recursion
  - First Example: Factorial
  - Schemas of Recursion
  - Recursive Data Structures

- Recursion in Practice
  - Solving a Problem by Recursion: Hanoi Towers
  - Classical Recursive Functions
  - Recursive Sorting Algorithms
    - MergeSort
    - QuickSort

Recursive Functions and Procedures

Recursively Defined Function: its body contains calls to itself

The Scrabble™ word game

- Given 7 letter tiles, one should form existing English worlds
  $\boxed{T}\ \boxed{I}\ \boxed{R}\ \boxed{N}\ \boxed{E}\ \boxed{G}\ \boxed{S} \leadsto$ RIG, SIRE, GRINS, INSERT, RESTING, ...
- How many permutation exist?
  - First position: pick one tile from 7
  - Second position: pick one tile from 6 remaining
  - Third position: pick one tile from 5 remaining
  - ...
  - Total: $7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$

This is the Factorial

- Mathematical definition of factorial: $\begin{cases} n! = n \times (n-1)! \\ 0! = 1 \end{cases}$
- Factorial : integer $\rightarrow$ integer
  Precondition: factorial(n) defined if and only if $n \geq 0$
  Postcondition: factorial(n)$= n!$

Recursive Algorithm for Factorial

Literal Translation of the Mathematical Definition

$$
\begin{array}{|l|}
\hline
\textsc{Factorial}(\mathrm{n}): \\
\quad \textbf{if } \mathrm{n} = 0 \textbf{ then } r \leftarrow 1 \\
\qquad\qquad\qquad \textbf{else} \quad r \leftarrow n \times \mathit{factorial}(n-1) \\
\quad \textbf{end} \\
\hline
\end{array}
$$

Remarks:

- $\boxed{r \leftarrow 1}$ is the base case: no recursive call
- $\boxed{r \leftarrow n \times \mathit{factorial}(n-1)}$ is the general case: Achieves a recursive call
- Reaching the base case is mandatory for the algorithm to finish

Factorial Computation Details

FACTORIAL(n):
if n = 0 **then** $r \leftarrow 1$
else $r \leftarrow n \times factorial(n-1)$
end

$factorial(4) =$

Factorial Computation Details

$\text{FACTORIAL}(\mathsf{n})$:
  if $\mathsf{n} = 0$ **then** $r \leftarrow 1$
    else $\boldsymbol{r \leftarrow n \times factorial(n-1)}$
  end

$factorial(4) = 4 \times factorial(3)$

Factorial Computation Details

$\textsc{Factorial}(\mathsf{n})$:
if $\mathsf{n} = 0$ **then** $r \leftarrow 1$
else $\boldsymbol{r \leftarrow n \times factorial(n-1)}$
end

$$factorial(4) = 4 \times \overbrace{factorial(3)}^{}$$
$$3 \times factorial(2)$$

Factorial Computation Details

FACTORIAL(n):
  if n = 0 **then** $r \leftarrow 1$
    else $\boldsymbol{r \leftarrow n \times factorial(n-1)}$
  end

$$factorial(4) = 4 \times factorial(3)$$

$$\overbrace{3 \times factorial(2)}$$

$$\overbrace{2 \times factorial(1)}$$

Factorial Computation Details

$$\begin{array}{|l|}\hline \textsc{Factorial}(\mathrm{n}): \\ \quad \textbf{if } \mathrm{n} = 0 \textbf{ then } r \leftarrow 1 \\ \qquad\qquad\quad \textbf{else } \boldsymbol{r \leftarrow n \times factorial(n-1)} \\ \quad \textbf{end} \\ \hline \end{array}$$

$$factorial(4) = 4 \times \overbrace{factorial(3)}^{}$$

$$3 \times \overbrace{factorial(2)}^{}$$

$$2 \times \overbrace{factorial(1)}^{}$$

$$1 \times factorial(0)$$

Factorial Computation Details

FACTORIAL(n):
if n = 0 **then** $\boldsymbol{r \leftarrow 1}$
else $r \leftarrow n \times factorial(n-1)$
end

$$factorial(4) = 4 \times \overbrace{factorial(3)}^{}$$

$$\overbrace{3 \times factorial(2)}$$

$$\overbrace{2 \times factorial(1)}$$

$$\overbrace{1 \times factorial(0)}$$

$$\overbrace{1} \quad \Big\} \text{ Base Case}$$

Factorial Computation Details

FACTORIAL(n):
if n = 0 **then** $r \leftarrow 1$
else $r \leftarrow n \times factorial(n-1)$
end

$$
\left.
\begin{array}{l}
factorial(4) = 4 \times factorial(3) \\
\qquad \overbrace{3 \times factorial(2)} \\
\qquad\qquad \overbrace{2 \times factorial(1)} \\
\qquad\qquad\qquad \overbrace{1 \times factorial(0)}
\end{array}
\right\} \text{Recursive Descent}
$$

$$
4 \times 3 \times 2 \times 1 \times \overbrace{1} \quad \Big\} \text{Base Case}
$$

Factorial Computation Details

FACTORIAL(n):
  if n = 0 **then** $r \leftarrow 1$
       else $r \leftarrow n \times factorial(n-1)$
  end

$$\left.\begin{array}{l} factorial(4) = 4 \times factorial(3) \\ \qquad\qquad\qquad\quad \overbrace{3 \times factorial(2)} \\ \qquad\qquad\qquad\qquad\quad \overbrace{2 \times factorial(1)} \\ \qquad\qquad\qquad\qquad\qquad\quad \overbrace{1 \times factorial(0)} \end{array}\right\} \text{Recursive Descent}$$

$$4 \times 3 \times 2 \times 1 \times \overbrace{1} \quad \Big\} \text{ Base Case}$$

$$4 \times 3 \times 2 \times \overbrace{\quad 1 \quad}$$

Factorial Computation Details

$\textsc{Factorial}(\mathsf{n})$:
if $\mathsf{n} = 0$ **then** $r \leftarrow 1$
else $r \leftarrow n \times factorial(n-1)$
end

$$\left.\begin{array}{l} factorial(4) = 4 \times factorial(3) \\ \qquad \overbrace{3 \times factorial(2)} \\ \qquad\quad \overbrace{2 \times factorial(1)} \\ \qquad\qquad \overbrace{1 \times factorial(0)} \end{array}\right\} \text{Recursive Descent}$$

$$4 \times 3 \times 2 \times 1 \times \overbrace{1} \quad \} \text{ Base Case}$$

$$4 \times 3 \times 2 \times \overbrace{\quad 1 \quad}$$

$$4 \times 3 \times \overbrace{\quad 2 \quad}$$

Factorial Computation Details

$$
\begin{array}{|l|}
\hline
\text{FACTORIAL}(\mathsf{n})\text{:} \\
\quad \textbf{if } \mathsf{n} = 0 \textbf{ then } r \leftarrow 1 \\
\qquad\qquad\quad \textbf{else } \; r \leftarrow n \times factorial(n-1) \\
\quad \textbf{end} \\
\hline
\end{array}
$$

$$
\left.
\begin{array}{l}
factorial(4) = 4 \times factorial(3) \\
\qquad\qquad\qquad\quad \overbrace{3 \times factorial(2)} \\
\qquad\qquad\qquad\qquad\quad \overbrace{2 \times factorial(1)} \\
\qquad\qquad\qquad\qquad\qquad\quad \overbrace{1 \times factorial(0)}
\end{array}
\right\} \text{Recursive Descent}
$$

$$
4 \times 3 \times 2 \times 1 \times \overbrace{\;1\;} \quad \Big\} \text{ Base Case}
$$

$$
4 \times 3 \times 2 \times \overbrace{\qquad 1 \qquad}
$$

$$
4 \times 3 \times \overbrace{\qquad 2 \qquad}
$$

$$
4 \times \overbrace{\qquad 6 \qquad}
$$

Factorial Computation Details

FACTORIAL(n):
  if n = 0 **then** $r \leftarrow 1$
    else $r \leftarrow n \times factorial(n-1)$
  end

$$\left.\begin{array}{l} factorial(4) = 4 \times factorial(3) \\ \qquad\qquad\qquad\quad \overbrace{3 \times factorial(2)} \\ \qquad\qquad\qquad\qquad\quad \overbrace{2 \times factorial(1)} \\ \qquad\qquad\qquad\qquad\qquad\quad \overbrace{1 \times factorial(0)} \end{array}\right\} \text{Recursive Descent}$$

$$4 \times 3 \times 2 \times 1 \times \overbrace{1} \quad \} \ \text{Base Case}$$

$$4 \times 3 \times 2 \times \overbrace{1}$$

$$4 \times 3 \times \overbrace{2}$$

$$4 \times \overbrace{6}$$

$$\overbrace{24}$$

Factorial Computation Details

```
FACTORIAL(n):
  if n = 0 then r ← 1
           else  r ← n × factorial(n − 1)
  end
```

$$\left.\begin{array}{l} factorial(4) = 4 \times factorial(3) \\ \qquad\qquad\qquad \overbrace{3 \times factorial(2)} \\ \qquad\qquad\qquad\qquad \overbrace{2 \times factorial(1)} \\ \qquad\qquad\qquad\qquad\qquad \overbrace{1 \times factorial(0)} \end{array}\right\} \text{Recursive Descent}$$

$$4 \times 3 \times 2 \times 1 \times \overbrace{1} \quad \Big\} \text{ Base Case}$$

$$\left.\begin{array}{l} 4 \times 3 \times 2 \times \overbrace{\quad 1 \quad} \\ 4 \times 3 \times \overbrace{\quad 2 \quad} \\ 4 \times \overbrace{\quad 6 \quad} \\ \overbrace{\quad 24 \quad} \end{array}\right\} \text{Recursive Climb}$$

$$factorial(4) = 24$$

Third Chapter

Recursion

General Recursion Schema

$$\boxed{\begin{array}{lll} \textbf{if } \textsc{Cond} & \textbf{then} & \textsc{BaseCase} \\ & \textbf{else} & \textsc{GenCase} \\ \textbf{end} & & \end{array}}$$

- $\textsc{Cond}$ is a boolean expression
- If $\textsc{Cond}$ is true, execute the base case $\textsc{BaseCase}$ (without recursive call)
- If $\textsc{Cond}$ is false, execute the general case $\textsc{GenCase}$ (with recursive calls)

The factorial(n) example

$\textsc{BaseCase}$: $r \leftarrow 1$

$\textsc{GenCase}$: $r \leftarrow n \times factorial(n-1)$

Other Recursion Schema: Multiple Recursion

More than one recursive call

Example: Pascal's Rule and $\binom{n}{k}$

- Amount of k-long sets of n elements (order ignored)

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } n = k; \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{else } (1 \leq k < n). \end{cases}$$

- $\boxed{\binom{4}{2} = 6}$ $\rightsquigarrow$ 6 ways to build a pair of elements picked from 4 possibilities:

  {A;B},{A;C},{A;D},{B;C},{B;D},{C;D} (if order matters, 4×3 possibilities)

Corresponding Algorithm:

Pascal (n, k)

If $k = 0$ or $k = n$ **then** $r \leftarrow 1$
else $r \leftarrow$ Pascal $(n-1, k)$ + Pascal $(n-1, k-1)$

First rows

```
            1
          1   1
        1   2   1
      1   3   3   1
    1   4  (6)  4   1
  1   5  10  10   5   1
1   6  15  20  15   6   1
```

Other Recursion Schema: Mutual Recursion

Several functions calling each other

Example 1

$$A(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ B(n+2) & \text{if } n > 1 \end{cases} \qquad B(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ A(n-3)+4 & \text{if } n > 1 \end{cases}$$

Compute A(5):

Example 2: one definition of parity

$$\text{even?}(n) = \begin{cases} \mathit{true} & \text{if } n = 0 \\ \text{odd}(n-1) & \text{else} \end{cases} \quad \text{and} \quad \text{odd?}(n) = \begin{cases} \mathit{false} & \text{if } n = 0 \\ \text{even}(n-1) & \text{else} \end{cases}$$

Other examples

- Some Maze Traversal Algorithm also use Mutual Recursion (see lab)
- Mutual Recursion classical in Context-free Grammar (see compilation course)

Other Recursion Schema: Embedded Recursion

Recursive call as Parameter

Example: Ackerman function

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{else} \end{cases}$$

Thus the algorithm:

$\textsc{Ackerman}(m,\ n)$

if $m = 0$ **then** $n + 1$

else if $n = 0$ **then** $\textsc{Ackerman}(m-1,\ 1)$

else $\textsc{Ackerman}(m-1,\ \textsc{Ackerman}(m,\ n-1))$

Other Recursion Schema: Embedded Recursion

Recursive call as Parameter

Example: Ackerman function

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{else} \end{cases}$$

Thus the algorithm:

$\textsc{Ackerman}(m,\ n)$

if $m = 0$ **then** $n+1$

else if $n = 0$ **then** $\textsc{Ackerman}(m-1,\ 1)$

else $\textsc{Ackerman}(m-1,\ \textsc{Ackerman}(m,\ n-1))$

Warning, this function grows quickly:

$Ack(1,n) = n+2$

$Ack(2,n) = 2n+3$

$Ack(3,n) = 8 \cdot 2^n - 3$

$Ack(4,n) = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} \Big\}n$

$Ack(4,4) > 2^{65536} > 10^{80}$ (estimated amount of particles in universe)

Recursive Data Structures

Definition

Recursive datatype: Datatype defined using itself

Classical examples

List: element followed by a list or empty list

Binary tree: {value; left son; right son} or empty tree

This is the subject of the module "Data Structures"

- After TOP and POO in track

Example: Strings as (linked) lists

Defined operations

[ ]		*The empty string object*
cons	Char $\times$ String $\mapsto$ String	*Adds the char in front of the list*
car	String $\mapsto$ Char	*Get the first char of the list (not defined if empty?(str))*
cdr	String $\mapsto$ String	*Get the list without first char*
empty?	String $\mapsto$ Boolean	*Tests if the string is empty*

- As you can see, strings are defined recursively using strings

Examples

- "bo" = cons('b',cons('o',[ ]))
- "hello" = cons('h',cons('e',cons('l',cons(cons('l',cons(cons('o',[ ])))))))
- cdr(cons('b',cons('o',[ ]))) = "o" = cons('o',[ ])

These are native constructs in LISP programing language

- But, these constructs are hard to remember (cdr vs. car)
- But, all these parenthesis are nasty (too much syntaxic sugar)

Doing the same in Java

Element Class representing a letter and the string following (ie, non-empty strings)

String Class representing a string (either empty or not)

```java
public class Element {
  public char value;
  public Element tail;

  Element(char x, Element tail) {
      value = x;
      this.tail = tail;
  }
}
```

```java
public class StringRec {
  private Element head = null;

  public boolean isEmpty() {
      return head == null;
  }
  public void cons(char x) {
    // Create new elem and connect it
    Element newElem = new Element(x, head);
    // This is new head
    head = newElem;
} }
```

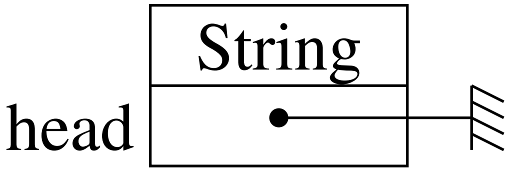
```java
StringRec plop = new StringRec().cons('p').cons('o').cons('l').cons('p');
```

Object Orientation is helping (only) when programming at large

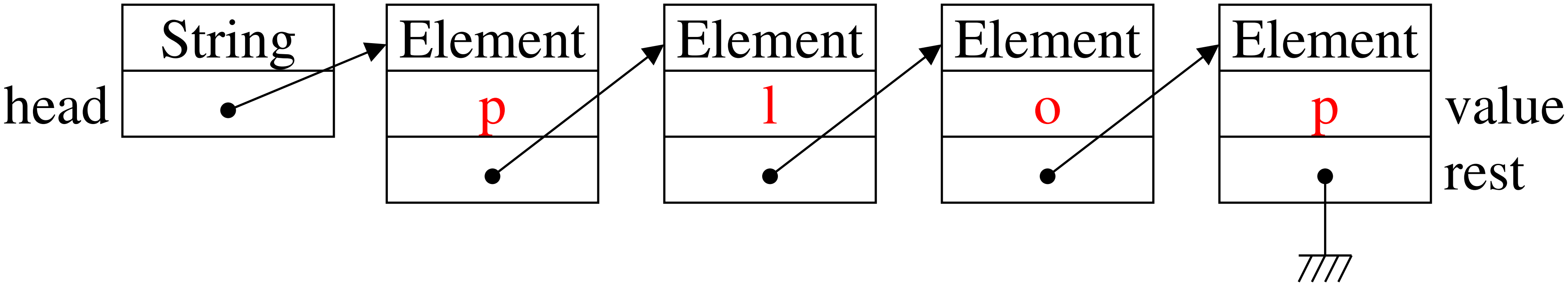
- It's "not really helping" when programming at small (both are orthogonal)
- Here, message lost under the syntaxic sugar
- Dotted notation not natural in this case (this could be improved? mail me!)

Some Memory Representation Examples in Java

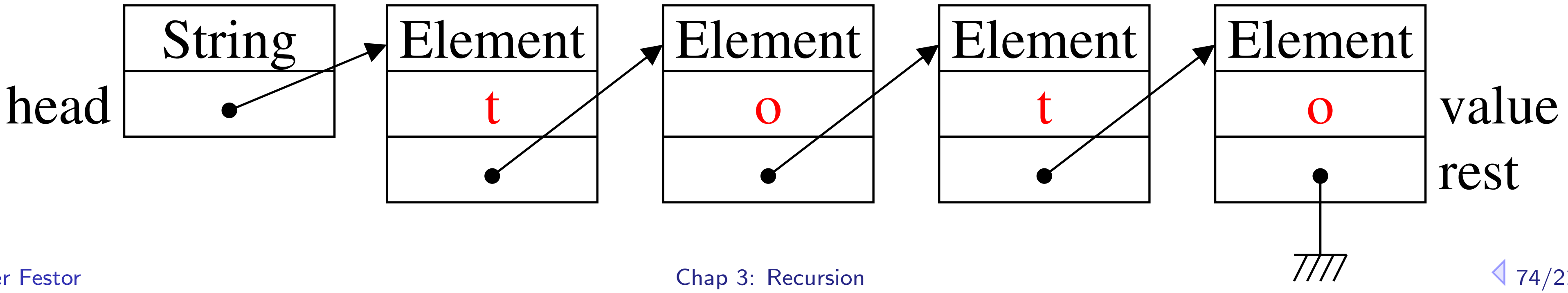
Empty String: `new StringRec();`



String "plop": new StringRec().cons('p').cons('o').cons('l').cons('p');



String "toto": : new StringRec().cons('o').cons('t').cons('o').cons('t');



Scala Lists

Nil			*The empty list object*
::	elm × List ↦ List		*Adds the element in front of the list (pronounced cons)*
head	List ↦ elm		*Get the first char of the list (not defined if lst.isEmpty)*
tail	List ↦ List		*Get the list without first char*
isEmpty	List ↦ Boolean		*Tests if the list is empty*

Example: "hello" ≡ 'h'::'e'::'l'::'l'::'o'::Nil

```
scala> val lst = 1::2::3::4::Nil
lst: List[Int] = List(1, 2, 3, 4)

scala> lst.head
res1: Int = 1

scala> lst.tail
res2: List[Int] = List(2, 3, 4)
```

```
scala> def sum(list:List[Int]): Int = list match {
     |   case Nil => 0
     |   case i::newlist => i + sum(newlist)
     | }
sum: (list: List[Int])Int

scala> lst.sum
res3: Int = 10
```

Functional orientation of Scala is a beauty

- Is is much more convenient than LISP, syntaxic-sugar-free compared to Java
- Plays very well with Scala's pattern-matching

Recursion in Practice

Recursion is a tremendously important tool in algorithmic

- Recursive algorithms often simple to understand, but hard to come up with
- Some learners even have a *trust issue* with regard to recursive algorithms

Holistic and Reductionist Points Of View

- Holism: *the whole is greater than the sum of its parts*
- Reductionism: *the whole can be understood completely if you understand its parts and the nature of their 'sum'.*

Writing a recursive algorithm

- Reductionism clearly induced since views problems as sum of parts
- But Holistic approach also mandatory:
  - When looking for general solution, assume that solution to subproblems given
  - Don't focus of every detail, keep a general point of view (not always natural, but) If you cannot see the forest out of trees, don't look at branches and leaves
- At the end, recursion is something that you can only learn through experience

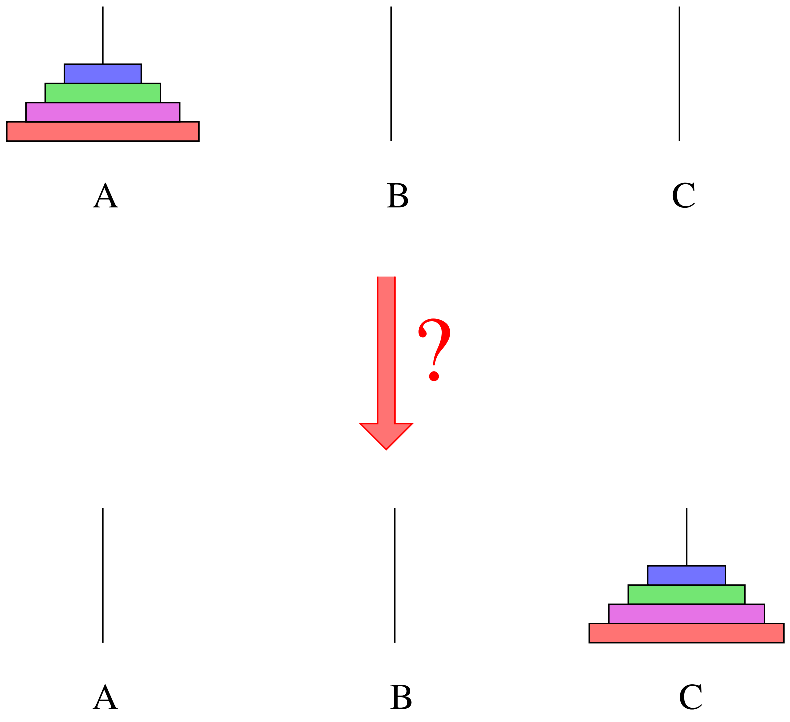
Third Chapter

Recursion

How to Solve a Problem Recursively?

1. Determine the parameter on which recursion will operate:
   Integer or Recursive datatype
2. Solve simple cases: the ones for which we get the answer directly
   They are the Base Cases
3. Setup Recursion:
   - Assume you know to solve the problem for one (or several) parameter value being strictly smaller (ordering to specify) than the value you got
   - How to solve the problem for the value you got with that knowledge?
4. Write the general case
   Express the searched solution as a function of the sub-solution you assume you know
5. Write Stopping Conditions (ie, base cases)
   Check that your recursion always reaches these values

A Classical Recursive Problem: Hanoï Towers



- Data: n disks of differing sizes
- Problem: change the stack location

  A third stick is available
- Constraint: no big disk over small one

Problem Analysis

- Parameters :
  - Amount n of disks stacked on initial stick
  - The sticks

Problem Analysis

- Parameters :
  - Amount n of disks stacked on initial stick
  - The sticks

$\leadsto$ We recurse on integer n

- How to solve problem for n disks when we know how to do with $n-1$ disks?

Problem Analysis

- Parameters :
  - Amount n of disks stacked on initial stick
  - The sticks

$\rightsquigarrow$ We recurse on integer n

- How to solve problem for n disks when we know how to do with $n-1$ disks?

$\rightsquigarrow$ Decomposition between bigger disk and (n-1) smaller ones

- We want to write procedure $\textsc{Hanoi}(\textsc{n}, \textsc{From}, \textsc{To})$.
  It moves the $\textsc{n}$ disks from stick $\textsc{From}$ to stick $\textsc{To}$

Problem Analysis

- Parameters :
  - Amount n of disks stacked on initial stick
  - The sticks

$\rightsquigarrow$ We recurse on integer n

- How to solve problem for n disks when we know how to do with $n-1$ disks?

$\rightsquigarrow$ Decomposition between bigger disk and (n-1) smaller ones

- We want to write procedure $\textsc{Hanoi}(\textsc{n}, \textsc{From}, \textsc{To})$.
  It moves the $\textsc{n}$ disks from stick $\textsc{From}$ to stick $\textsc{To}$

$\rightsquigarrow$ For simplicity sake, we introduce procedure $\textsc{Move}(\textsc{From},\textsc{To})$
It moves the upper disk from stick $\textsc{From}$ to stick $\textsc{To}$
(also checks that we don't move a big one over a small one)

Problem Analysis

- Parameters :
  - Amount n of disks stacked on initial stick
  - The sticks

$\rightsquigarrow$ We recurse on integer n

- How to solve problem for n disks when we know how to do with $n-1$ disks?

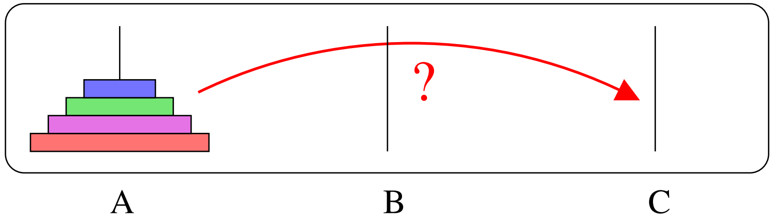
$\rightsquigarrow$ Decomposition between bigger disk and (n-1) smaller ones

- We want to write procedure $\textsc{Hanoi}(\textsc{n}, \textsc{From}, \textsc{To})$.
  It moves the $\textsc{n}$ disks from stick $\textsc{From}$ to stick $\textsc{To}$

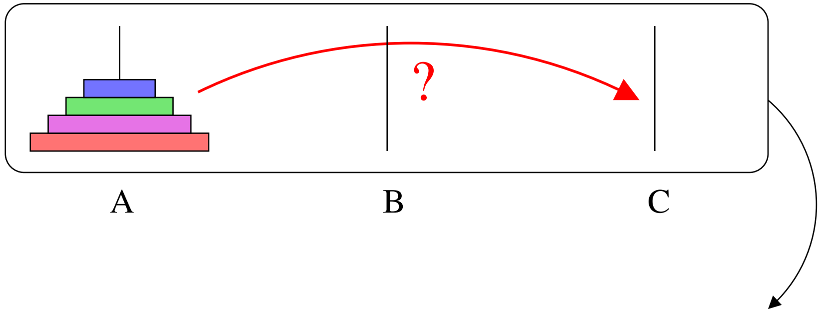
$\rightsquigarrow$ For simplicity sake, we introduce procedure $\textsc{Move}(\textsc{From},\textsc{To})$
It moves the upper disk from stick $\textsc{From}$ to stick $\textsc{To}$
(also checks that we don't move a big one over a small one)

- Stopping Condition: when only one disk remains, use $\textsc{Move}$
  $\textsc{Hanoi}(1,\textsc{x},\textsc{y})=\textsc{Move}(\textsc{x},\textsc{y})$

Possible Decomposition of Hanoi(n, A, C)

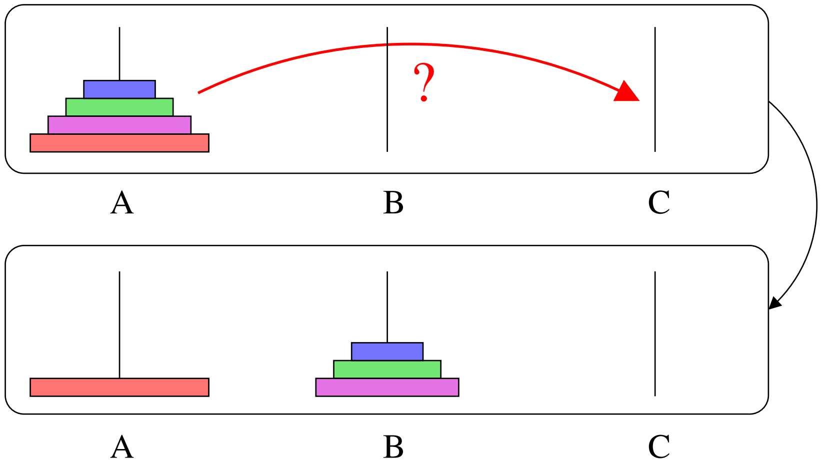


Possible Decomposition of Hanoi(n, A, C)



Assume "someone" can move N−1 disks for you

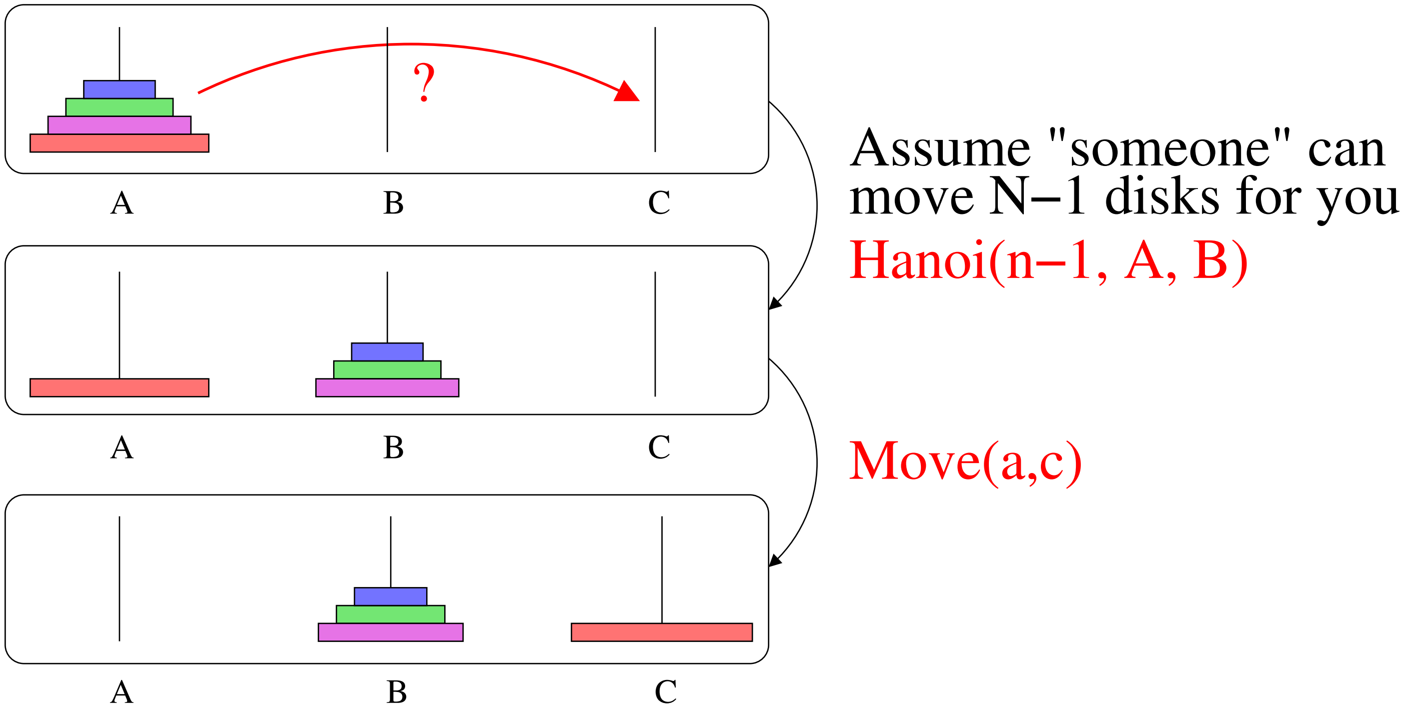
Possible Decomposition of Hanoi(n, A, C)



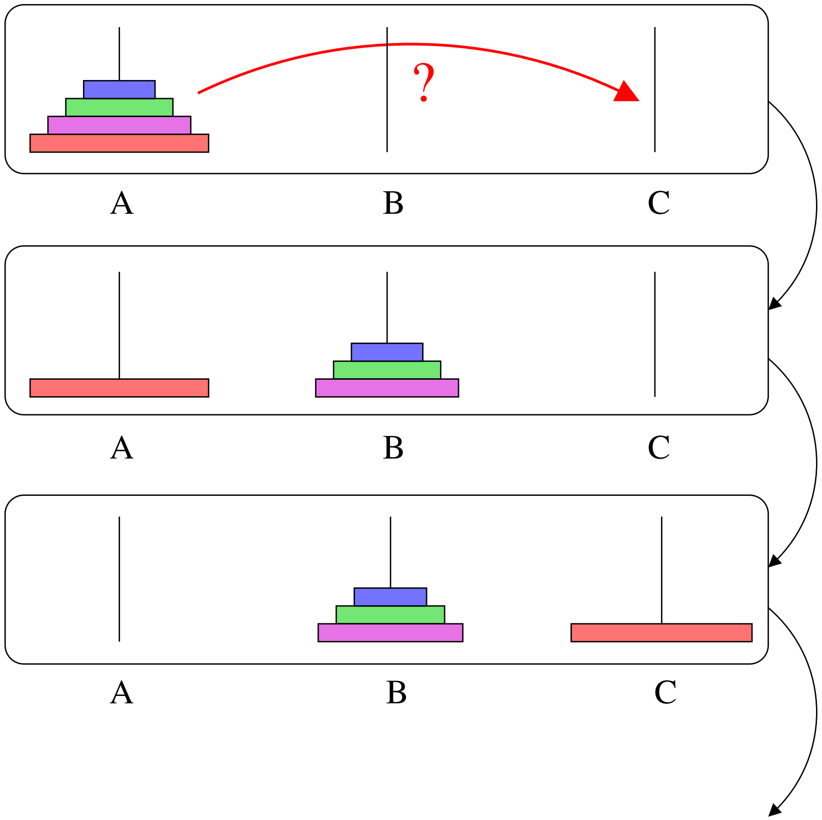
Assume "someone" can move N–1 disks for you

Hanoi(n–1, A, B)

Possible Decomposition of Hanoi(n, A, C)



Possible Decomposition of Hanoi(n, A, C)

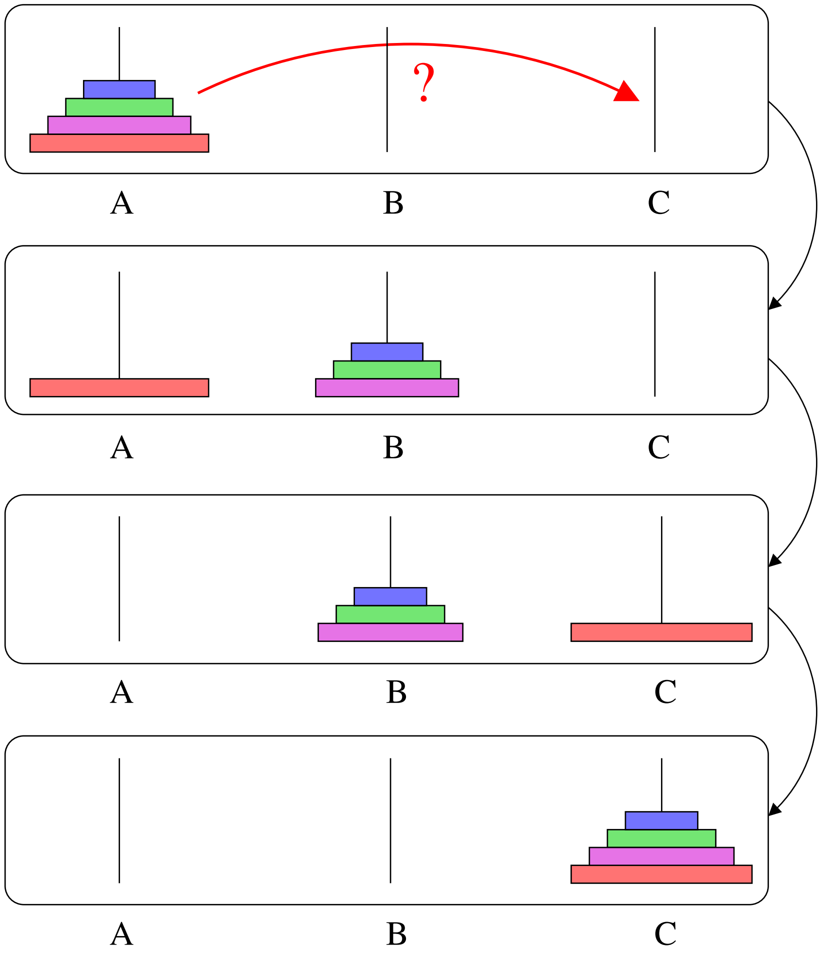


Assume "someone" can move N–1 disks for you
Hanoi(n–1, A, B)

Move(a,c)

Ask "your friend" again
Hanoi(n–1, B, C)

Possible Decomposition of Hanoi(n, A, C)

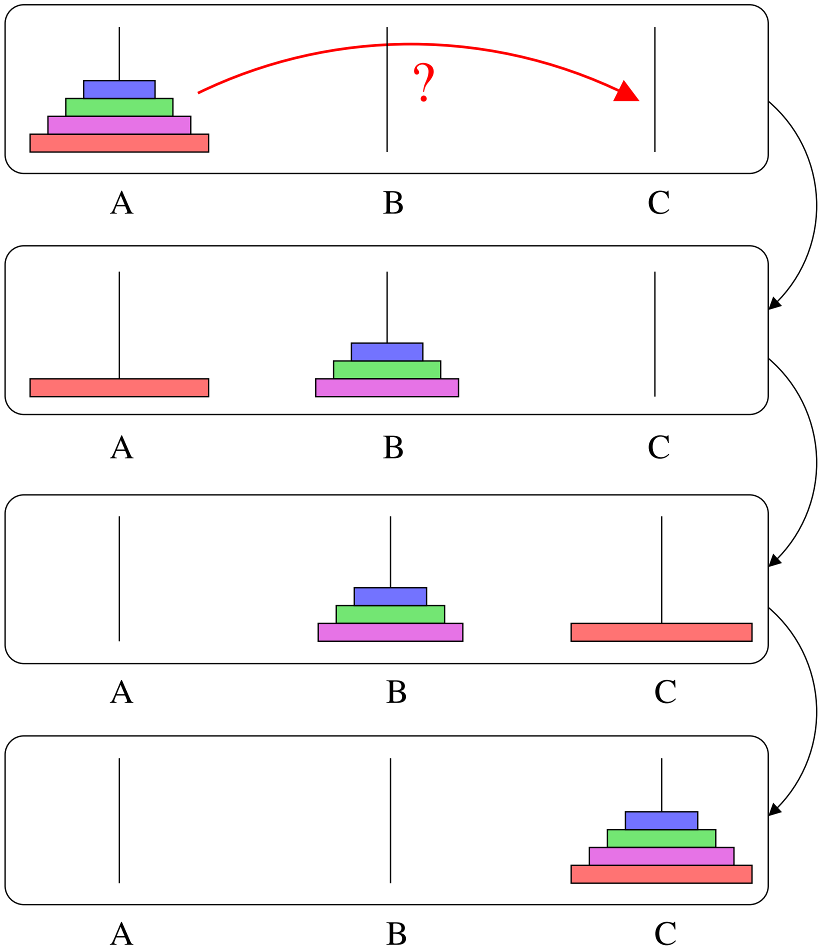


Assume "someone" can move N–1 disks for you
Hanoi(n–1, A, B)

Move(a,c)

Ask "your friend" again
Hanoi(n–1, B, C)

Possible Decomposition of Hanoi(n, A, C)



Assume "someone" can move N–1 disks for you
Hanoi(n–1, A, B)

Move(a,c)

Ask "your friend" again
Hanoi(n–1, B, C)

Do you feel the *trust issue* against recursive algorithms?

> *To iterate is human, to recurse is divine.* — *L Peter Deutsch*

(Deutsch: ghostview; first JIT compiler (for SmallTalk) 15 yr ahead; wrote LISP interpreter for PDP-1 by 12yr)

Corresponding Algorithm

Function `hanoi` *(n, from, to, other)* **is**

```
if n = 1 then
    move (from, to)
else
    hanoi (n-1, from, other)
    move (from, to)
    hanoi (n-1, other, to)
```

Corresponding Algorithm

```
Function hanoi (n, from, to, other) is
    if n = 1 then
        move (from, to)
    else
        hanoi (n-1, from, other)
        move (from, to)
        hanoi (n-1, other, to)
```

Variant with 0 as base case

```
Function hanoi (n, from, to, other) is
    if n ≠ 0 then
        hanoi (n-1, from, other, to)
        move (from, to)
        hanoi (n-1, other, to, from)
```

Corresponding Algorithm

Function `hanoi` *(n, from, to, other)* **is**
  if $n = 1$ **then**
    `move` (from, to)
  else
    `hanoi` (n-1, from, other)
    `move` (from, to)
    `hanoi` (n-1, other, to)

```
def hanoi(n:Int, from:Int,to:Int,other:Int)=
  if (n == 1) {
    move(from, to)
  } else {
    hanoi(n-1, from, other, to)
    move(from, to)
    hanoi(n-1, other, to, from)
  }
```

Variant with 0 as base case

Function `hanoi` *(n, from, to, other)* **is**
  if $n \neq 0$ **then**
    `hanoi` (n-1, from, other, to)
    `move` (from, to)
    `hanoi` (n-1, other, to, from)

```
def hanoi(n:Int, from:Int,to:Int,other:Int)=
  if (n != 0) {
    hanoi(n-1, from, other, to)
    move(from, to)
    hanoi(n-1, other, to, from)
  }
```

Back on the Hanoi Towers Problem

Problem first introduced in 1883 by Eduard Lucas, with a fake story

- Somewhere in India, Brahmane monks are doing this with 64 gold disks
- When they will be done, there will be the end of time

Anecdote Main Interest

- Amount of moves mandatory to move n disks: 1, 3, 7, 15, 31, 63, ...
- General term: $2^n - 1$
- The monks need $2^{64} - 1$ (ie 18 446 744 073 709 551 615) moves
- That's almost 600 000 000 000 years by playing one move per second

Other funny usage of the $2^n - 1$ suite

- Fibonacci searched the minimal amount of masses to weight any value up to N
- Tartaglia solution when masses are on the same arm:
  With n masses in the suite $(1, 2, 4, 8, \dots)$ you can weight any values up to $2^n - 1$
- *Mathematicians:* specialists of pointless stories leading to fundamental tools

[The Penguin Dictionary of Curious and Interesting Numbers, David Wells, 1997]

Third Chapter

Recursion

Classical Recursive Function: Fibonacci

Study of reproduction speed of rabbits (XII century)

- One pair at the beginning
- Each pair of fertile rabbits produces a new pair of offspring each month
- Rabbits become fertile in their second month of life
- Old rabbits never die
- $F_0 = 0$; $F_1 = 1$; $F_2 = 1$; $F_3 = 2$; $F_4 = 3$; $F_5 = 5$; $F_6 = 8$; $F_7 = 13$; $\ldots$

Classical Recursive Function: Fibonacci

Study of reproduction speed of rabbits (XII century)

- One pair at the beginning
- Each pair of fertile rabbits produces a new pair of offspring each month
- Rabbits become fertile in their second month of life
- Old rabbits never die
- $F_0 = 0$; $F_1 = 1$; $F_2 = 1$; $F_3 = 2$; $F_4 = 3$; $F_5 = 5$; $F_6 = 8$; $F_7 = 13$; $\ldots$

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ \forall n, F_n = F_{n-1} + F_{n-2} \end{cases}$$

Classical Recursive Function: Fibonacci

Study of reproduction speed of rabbits (XII century)

- One pair at the beginning
- Each pair of fertile rabbits produces a new pair of offspring each month
- Rabbits become fertile in their second month of life
- Old rabbits never die
- $F_0 = 0\ ;\ F_1 = 1\ ;\ F_2 = 1\ ;\ F_3 = 2\ ;\ F_4 = 3\ ;\ F_5 = 5\ ;\ F_6 = 8\ ;\ F_7 = 13\ ;\ \ldots$

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ \forall n, F_n = F_{n-1} + F_{n-2} \end{cases}$$

Corresponding Code

```
def fib(n:Int):Int =
  if (n <= 1)
    n // Base Case ('return' is optional)
  else
    fib(n-1) + fib(n-2)
```

(efficient implementations exist)

Classical Recursive Function: Fibonacci

Study of reproduction speed of rabbits (XII century)

- One pair at the beginning
- Each pair of fertile rabbits produces a new pair of offspring each month
- Rabbits become fertile in their second month of life
- Old rabbits never die
- $F_0 = 0$; $F_1 = 1$; $F_2 = 1$; $F_3 = 2$; $F_4 = 3$; $F_5 = 5$; $F_6 = 8$; $F_7 = 13$; ...

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ \forall n, F_n = F_{n-1} + F_{n-2} \end{cases}$$

Exercice :

Compute amount of recursive calls

Corresponding Code

```
def fib(n:Int):Int =
  if (n <= 1)
    n // Base Case ('return' is optional)
  else
    fib(n-1) + fib(n-2)
```

(efficient implementations exist)

Classical Recursive Function: Fibonacci

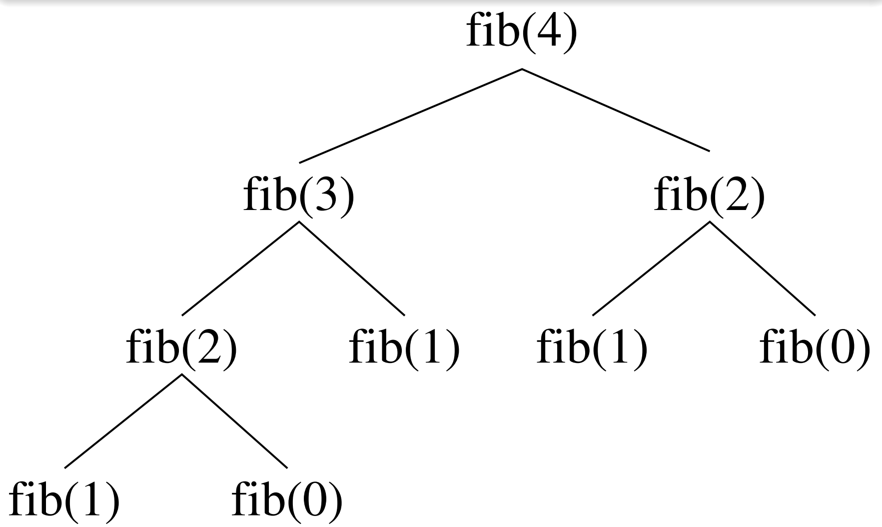
Study of reproduction speed of rabbits (XII century)

- One pair at the beginning
- Each pair of fertile rabbits produces a new pair of offspring each month
- Rabbits become fertile in their second month of life
- Old rabbits never die
- $F_0 = 0$; $F_1 = 1$; $F_2 = 1$; $F_3 = 2$; $F_4 = 3$; $F_5 = 5$; $F_6 = 8$; $F_7 = 13$; ...

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ \forall n, F_n = F_{n-1} + F_{n-2} \end{cases}$$

Exercice :

Compute amount of recursive calls



Corresponding Code

```
def fib(n:Int):Int =
  if (n <= 1)
    n // Base Case ('return' is optional)
  else
    fib(n-1) + fib(n-2)
```

(efficient implementations exist)

Classical Recursive Function: McCarthy 91

Definition

$$M(n) = \begin{cases} n - 10 & \text{if } n > 100 \\ M(M(n+11)) & \text{if } n \leq 100 \end{cases}$$

Interesting Property:

$\forall n \leq 101, M(n) = 91$
$\forall n > 101, M(n) = n - 10$

Proof

- When $90 \leq k \leq 100$, we have $f(k) = f(f(k+11)) = f(k+1)$
  In particular, $f(91) = f(92) = \ldots = f(101) = 91$
- When $k \leq 90$: Let r be so that: $90 \leq k + 11r \leq 100$
  $f(k) = f(f(k+11)) = \ldots = f^{(r+1)}(k+11r) = f^{(r+1)}(91) = 91$

John McCarthy (1927-)

Turing Award 1971, Inventor of language LISP, of expression "Artificial Intelligence" and of the Service Provider idea (back in 1961).

Classical Recursive Function: Syracuse

Function `syracuse` (n)
    if $n = 0$ **or** $n = 1$ **then**
        1
    else if $n \bmod 2 = 0$ **then**
        `syracuse`$(n/2)$
    else
        `syracuse`$(3 \times n + 1)$

- Question: Does this function always terminate?

  Hard to say: suite is not monotone

Classical Recursive Function: Syracuse

```
Function syracuse (n)
    if n = 0 or n = 1 then
        1
    else if n mod 2 = 0 then
        syracuse(n/2)
    else
        syracuse(3 × n + 1)
```

- Question: Does this function always terminate?

  Hard to say: suite is not monotone
- Collatz's Conjecture: $\forall n \in \mathbb{N}$, $\textsc{syracuse}(n) = 1$

Classical Recursive Function: Syracuse

```
Function syracuse (n)
    if n = 0 or n = 1 then
        1
    else if n mod 2 = 0 then
        syracuse(n/2)
    else
        syracuse(3 × n + 1)
```

- Question: Does this function always terminate?

  Hard to say: suite is not monotone
- Collatz's Conjecture: $\forall n \in \mathbb{N}$, $\textsc{syracuse}(n) = 1$
- Checked on computer $\forall n < 5 \cdot 2^{60} \approx 6 \cdot 10^{18}$

  (but other conjectures were proved false for bigger values only)

Classical Recursive Function: Syracuse

```
Function syracuse (n)
    if n = 0 or n = 1 then
        1
    else if n mod 2 = 0 then
        syracuse(n/2)
    else
        syracuse(3 × n + 1)
```

- Question: Does this function always terminate?

  Hard to say: suite is not monotone
- Collatz's Conjecture: $\forall n \in \mathbb{N}$, $\textsc{syracuse}(n) = 1$
- Checked on computer $\forall n < 5 \cdot 2^{60} \approx 6 \cdot 10^{18}$

  (but other conjectures were proved false for bigger values only)
- This is an open problem since 1937 (some rewards available)

> *Mathematics is not yet ready for such problems.*
>
> – Paul Erdös (1913–1996)

Third Chapter

Recursion

Back on Sorting Algorithms

Why don't CS profs ever stop talking about sorting?!

Sorting is the best studied problem in CS

- Variety of different algorithms (cf. PLM's lab for a small subset)
- Still some research on that topic (find best algorithm for a given workload kind)

Several Interesting ideas can be taught in that context

- Complexity: best case/worst case/average case as well as Big Oh notations
- Divide and Conquer and Recursion
- Randomized Algorithms

Sorting is a fundamental building block of algorithms

- Computers spend more time sorting than anything else (25% on mainframes)
- This is because a lot of problems come down to sorting elements

Applications of Sorting (1)

Searching

- Binary search algorithm: search item in dictionnary (sorted list) in $O(\log(n))$
- Speeding up searching perhaps the most important application of sorting

Closest pair

- Given n numbers, find the pair which are closest to each other
- Once the list is sorted, closest elements are next to each other

$\Rightarrow$ Linear scan is enough, thus $O(n\log(n)) + O(n) = O(n\log(n))$

Element uniqueness

- Given a list of n items, are they all unique or are there duplicates?
- Sort them, and do a linear scan of adjacent pairs
- (special case of closest pair, actually)

Applications of Sorting (2)

Frequency distribution

- Given a list of n items, which occures the largest number of times?
- Sort them, and do a linear scan to measure the length of adjacent runs

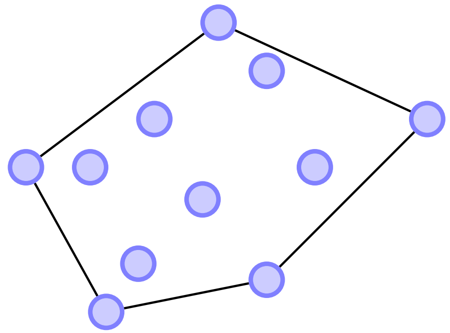
Median and Selection

- What is the kth largest item of a set?
- Sort keys, store them in an array (deal with dups)
- The kth larger can be found in constant time in kth pos of the array

Applications of Sorting (3)

Convex Hulls

- Given n points, find the smallest polygon containing them all (think of a elastic band stretched over the points)



- Sort points by x-coordinate, then y-coordinate
- Add them from left to right into the hull:
  - New rightmost point is on the boundary
  - Adding point to boundary may cause others to be deleted depending on whether the angle is convex or not

Huffman codes

- When storing a text, giving each letter's code the same length wastes space
- Example: e is more common than q, so give it a shorter code
- Huffman encoding: Sort letters by frequency, assign codes in order

Char	Freq.	Code
f	5	1100
e	6	1101
c	12	100

Char	Freq.	Code
b	13	101
d	16	111
a	45	0

- Simple & fast
- Not best compression
- Used in JPEG and MP3

Merge Sort

Recursive sorting

- Imagine the simpler way to sort recursively a list

Merge Sort

Recursive sorting

- Imagine the simpler way to sort recursively a list

1. Split your list in two sub-lists
2. Sort each of them recursively
3. Merge sorted sublists back

Merge Sort

Recursive sorting

- Imagine the simpler way to sort recursively a list

1. Split your list in two sub-lists
   One idea is to split evenly, but not the only one
2. Sort each of them recursively
   (base case: size$\leq$1)
3. Merge sorted sublists back
   at each step, pick smallest remaining elements of sublists, put it after already picked

Merge Sort

Recursive sorting

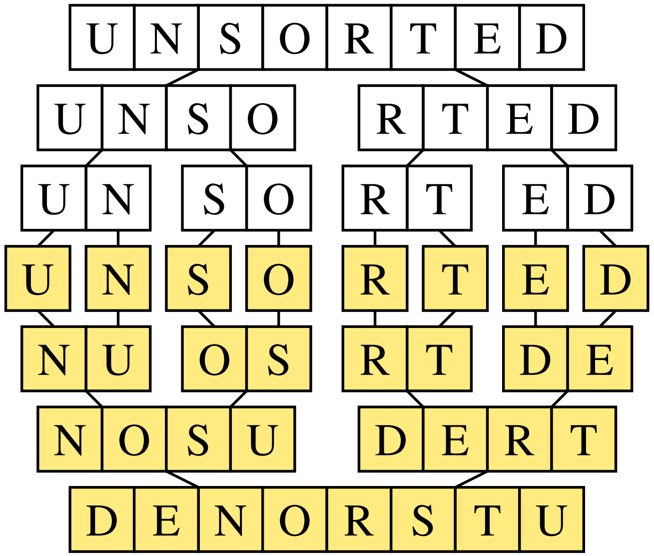
- Imagine the simpler way to sort recursively a list

1. Split your list in two sub-lists
   One idea is to split evenly, but not the only one
2. Sort each of them recursively
   (base case: size$\leq$1)
3. Merge sorted sublists back
   at each step, pick smallest remaining elements of sublists, put it after already picked

Merge Sort

- List splited evenly
- Sub-list copied away
- Merge trivial

(invented by John von Neumann in 1945)



Merge Sort

Scala code

```scala
def mergeSort(m: List[Int]):List[Int] ={
  // short enought to be already sorted
  if (m.length <= 1)
    return m

  // Slice (=cut) the array in two parts
  val middle = m.length / 2
  val left  = m.slice(0,middle)
  val right = m.slice(middle,m.length)

  // Sort each parts
  val leftSorted  = mergeSort(left)
  val rightSorted = mergeSort(right)

  // Merge them back
  return merge(leftSorted, rightSorted)
}
```

```scala
def merge(xs:List[Int], ys:List[Int])
    :List[Int] = {

  (xs,ys) match {

    case ( _ , Nil) => xs
    case (Nil,  _ ) => ys

    case (x::x2, y::y2) =>
       if (x < y) {
          x :: merge(x2 , ys)
       } else {
          y :: merge(xs , y2)
       }
  }
}
```

Complexity Analysis

- Time: $\log(n)$ recursive calls, each of them being linear $\rightsquigarrow$ $\Theta(n \times \log(n))$
- Space: Need to copy the array $\rightsquigarrow 2n$ (quite annoying) $+ \, log(n)$ for the stack

QuickSort

Presentation

- Invented by C.A.R. Hoare in 1962
- Widely used (in C library for example)

Big lines

- Pick one element, called *pivot* (random is ok)
- Reorder elements so that:
  - elements smaller to the pivot are before it
  - elements larger to the pivot are after it
- Recursively sort the parts before and after the pivot

Questions to answer

- How to pick the pivot? (random is ok)
- How to reorder the elements?
  - First solution: build sub-list (but this requires extra space)
  - Other solution: invert in place (but hinders stability, see below)

Simple Quick Sort

It's easy with sub-lists:

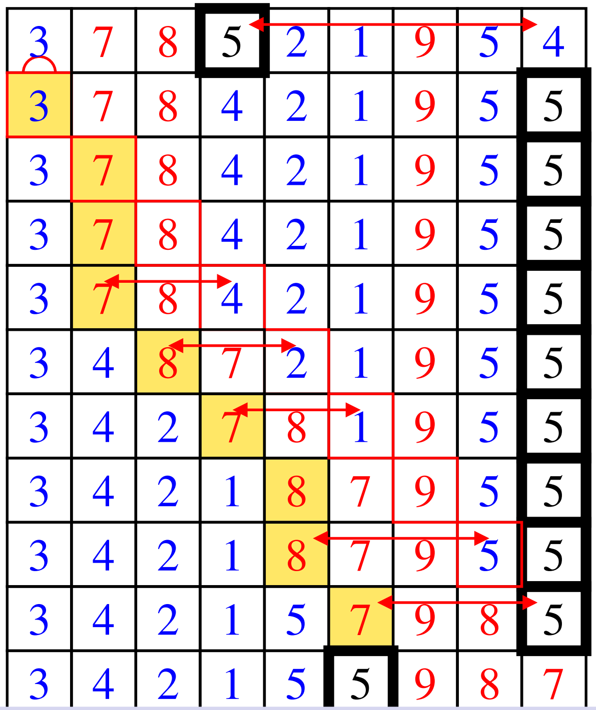
- Create two empty list variables
- Iterate over the original list; copy elements in correct sublist
- Recurse
- Concatenate results

```
def quicksort(lst:List[Int]):List[Int] = {
  if (lst.length <= 1) // Base case
    return lst

  // Randomly pick a pivot value
  val pivot = lst(lst.length / 2)

  // split the list
  var lows: List[Int] = Nil
  var mids: List[Int] = Nil
  var highs: List[Int] = Nil
  for (item <- lst) { // classify the items
    if ( item == pivot)    { mids  = item :: mids }
    else if (item < pivot) { lows  = item :: lows }
    else                   { highs = item :: highs}
  }

  // return sorted list appending chunks
  quicksort(lows) ::: mids ::: quicksort(highs)
}
```

Problem

- Space complexity is about $2n + \log(n)$)...
  (2n for array duplication, log(n) for the recursion stack)

In-place Quick Sort



Big lines of the list reordering

- Put the pivot at the end
- Traverse the list
  - If visited element is larger, do nothing
  - Else swap with "storage point" + shift storage right (storage point is on left initially)
- Swap pivot with storage point

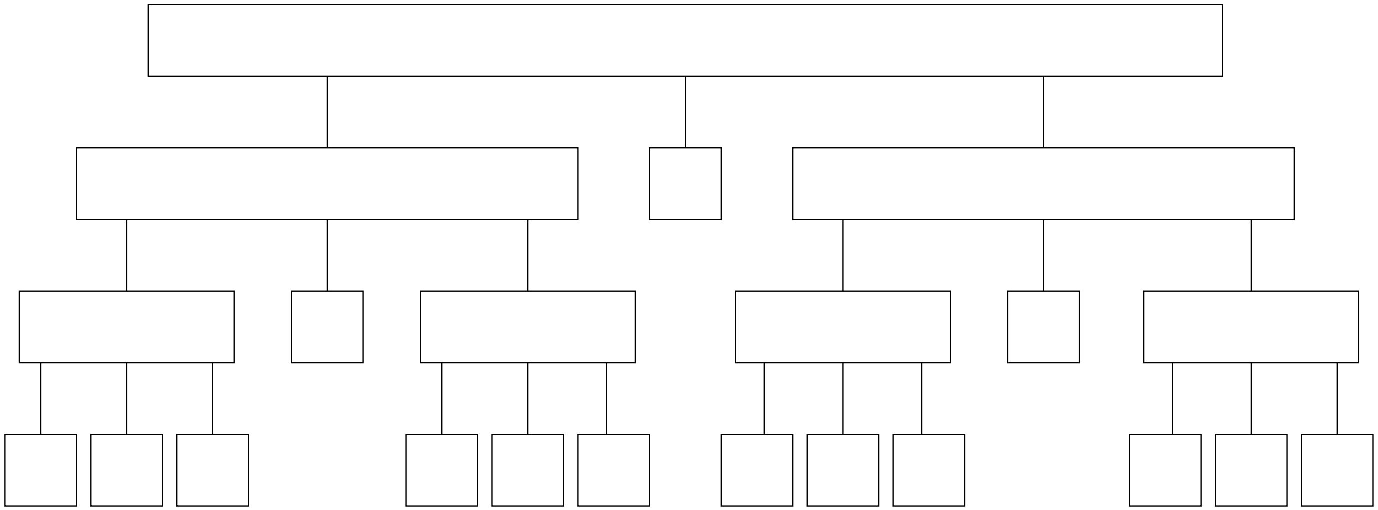
```
def quicksort(array:Array[Int]) {
  def lambda(array:Array[Int], left:Int, right:Int, pivotIndex:Int) {
    val pivotValue = array(pivotIndex)
    array.swap(pivotIndex, right) // Move pivot to end (swap() does not exist)
    val storeIndex = left
    for (i <- left to right-1) {
      if (array(i) <= pivotValue) {
        array.swap(i, storeIndex)
        storeIndex = storeIndex + 1
    } }
    array.swap(storeIndex, right) // Move pivot to its final place
    lambda(array, left, pivotIndex, (pivotIndex-left)/2)
    lambda(array, pivotIndex, right, (right-pivotIndex)/2)
  }
  lambda(array, 0, array.length-1, array.length/2) }
```

In-place QuickSort Complexity (1/2)

Best case for divide-and-conquer algorithms: Even Split

- Split the amount of work by 2 at each step (thus $\Theta(log(n))$ recursive calls)
- Work on each subproblem linear with its size (thus each call in $\Theta(n)$)

The recursion tree for best case:



What if we split 1%/99% at each step (instead of 50%/50%)?

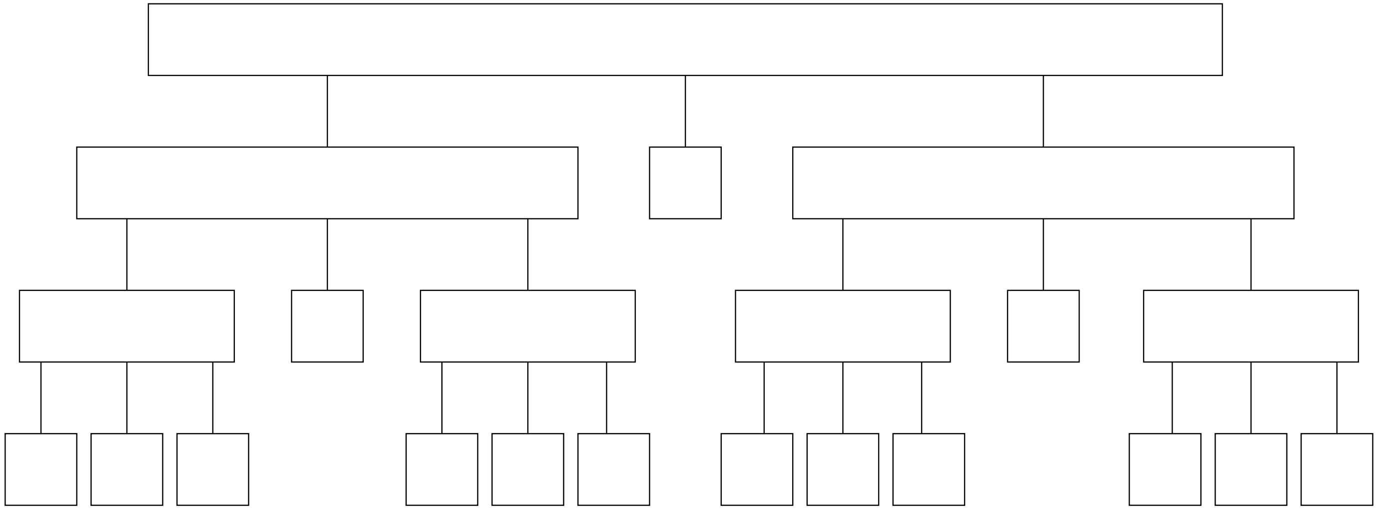
- We get steps $\rightsquigarrow$ whole algorithm in

In-place QuickSort Complexity (1/2)

Best case for divide-and-conquer algorithms: Even Split

- Split the amount of work by 2 at each step (thus $\Theta(log(n))$ recursive calls)
- Work on each subproblem linear with its size (thus each call in $\Theta(n)$)

The recursion tree for best case:



What if we split 1%/99% at each step (instead of 50%/50%)?

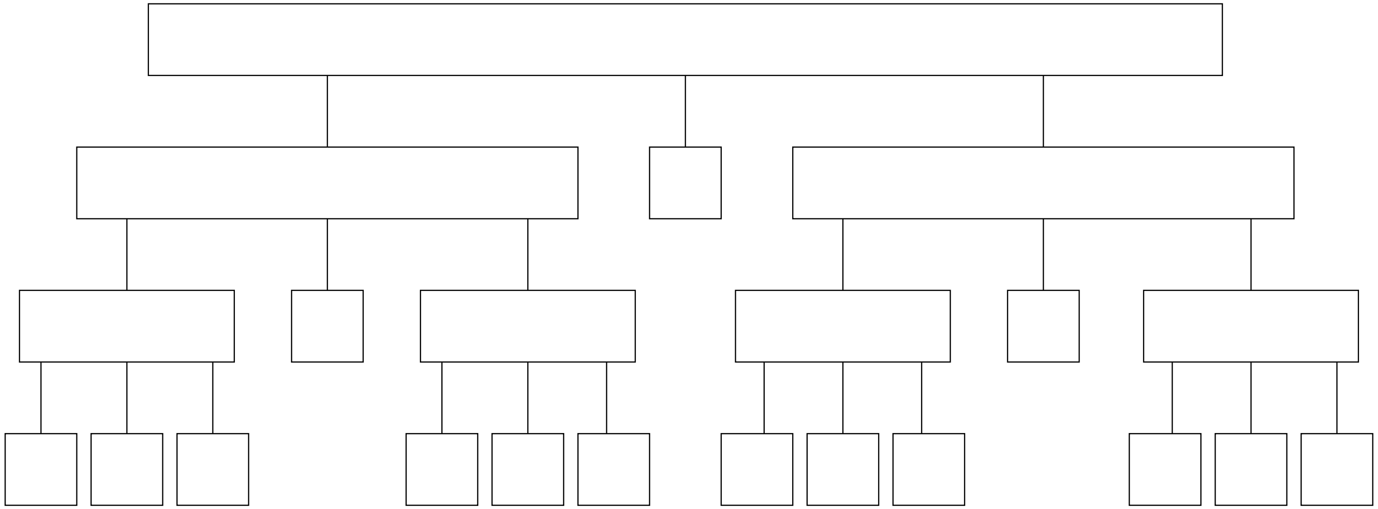
- We get $100 \times \log(n)$ steps $\rightsquigarrow$ whole algorithm in

In-place QuickSort Complexity (1/2)

Best case for divide-and-conquer algorithms: Even Split

- Split the amount of work by 2 at each step (thus $\Theta(log(n))$ recursive calls)
- Work on each subproblem linear with its size (thus each call in $\Theta(n)$)

The recursion tree for best case:



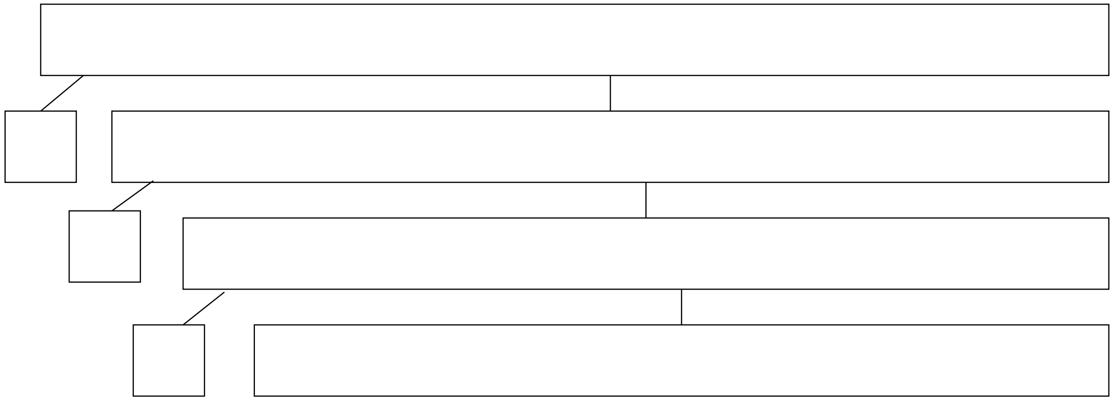
What if we split 1%/99% at each step (instead of 50%/50%)?

- We get $100 \times \log(n)$ steps $\rightsquigarrow$ whole algorithm in $\Theta(n \log(n))$

In-place QuickSort Complexity (2/2)

What if we have a fixed amount on one side?

- (happens when every values are duplicated, or with the wrong pivot)



- We get steps $\rightsquigarrow$ whole algorithm in in worst case

That's a fairly bad worst case time

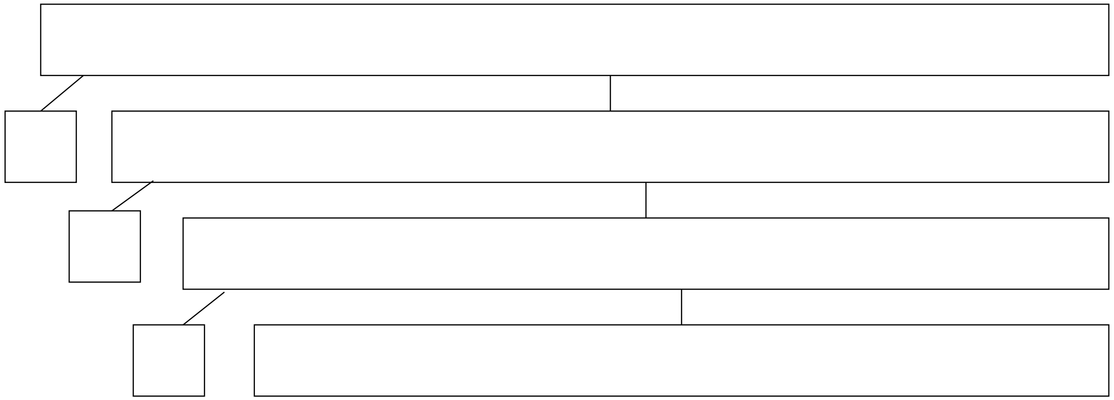
- Worst than MergeSort, for example
- But called Quicksort anyway because faster *in practice* than MergeSort
- In-Place version of both algorithms are not stable
- Both can be quite easily parallelized
- Space complexity: $O(\log(n))$ (to store the recursion stack)

(this ends the second lecture)

In-place QuickSort Complexity (2/2)

What if we have a fixed amount on one side?

- (happens when every values are duplicated, or with the wrong pivot)



- We get $O(n)$ steps $\leadsto$ whole algorithm in in worst case

That's a fairly bad worst case time

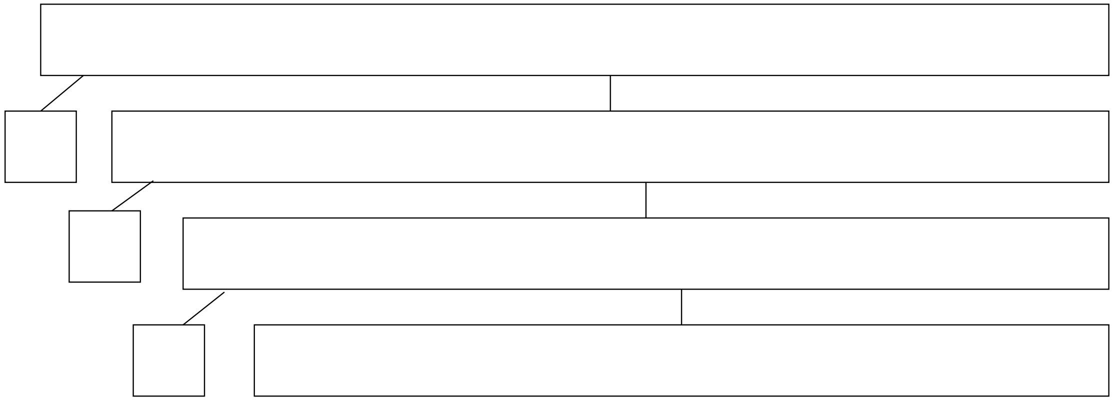
- Worst than MergeSort, for example
- But called Quicksort anyway because faster *in practice* than MergeSort
- In-Place version of both algorithms are not stable
- Both can be quite easily parallelized
- Space complexity: $O(\log(n))$ (to store the recursion stack)

(this ends the second lecture)

In-place QuickSort Complexity (2/2)

What if we have a fixed amount on one side?

- (happens when every values are duplicated, or with the wrong pivot)



- We get $O(n)$ steps $\rightsquigarrow$ whole algorithm in $O(n^2)$ in worst case

That's a fairly bad worst case time

- Worst than MergeSort, for example
- But called Quicksort anyway because faster *in practice* than MergeSort
- In-Place version of both algorithms are not stable
- Both can be quite easily parallelized
- Space complexity: $O(\log(n))$ (to store the recursion stack)

(this ends the second lecture)