# Techniques and tOols for Programming (TOP)

Martin Quinson <martin.quinson@loria.fr>

Telecom Nancy - Université de Lorraine – 1re année

2012-2013

# Module Presentation

## Algorithmic and Programming

### 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 Presentation

## Algorithmic and Programming

### 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 Algorithmic
  - Master theoretical basements (computer science is a science)
  - Know some classical problem resolution techniques
  - Know how to evaluate solutions (correctness, performance)
- Programming Techniques
  - Programming is an engineering task
  - Master the available tools (debugging, testing)
  - Notion of software engineering (software life cycle)

# Module Presentation

## Algorithmic and Programming

### 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 Algorithmic
  - Master theoretical basements (computer science is a science)
  - Know some classical problem resolution techniques
  - Know how to evaluate solutions (correctness, performance)
- Programming Techniques
  - Programming is an engineering task
  - Master the available tools (debugging, testing)
  - Notion of software engineering (software life cycle)

### Module Prerequisites

- Basics of Java (`if`, `for`, methods – *ie.*, tactical programming)
- Sense of logic, intuition

# Module organization

## Time organization

- 6 two-hours lectures (CM, with Abdelkader Lahmadi): Concepts introduction
- 8 two-hours exercise session (TD, with staff member[1]): Theoretical exercises
- 9 two-hours labs (TP, with staff member[1]): Coding exercises
- Homework: Systematically finish the in-class exercises

## Evaluation

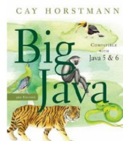
- Two hours table exam (16/03/2012)
- A project (to be given soon)
- Maybe some quiz at the beginning of labs

[1] Olivier Festor, Olivier Bouré, Maxime Rio, Mihai Andries, Hernàn Vanzetto, Abdelkader Lahmadi.

# Module bibliography

## Bibliography

- Introduction to programming and object oriented design, Nino & Hosch.
  Reference book. Very good for SE, less for CS ($120).
- Big Java, Cay S. Horstman. Less focused on programming ($110).
- Programmer en java, Claude Delannoy.
  Bon livre de référence (au format poche – 20€).
- Entraînez-vous et maîtrisez Java par la pratique, Alexandre Brillant.
  Nombreux exercices corrigés (25€).



## Webography

- IUT Orsay (in french): `http://www.iut-orsay.fr/~balkansk/`

# Table of Contents

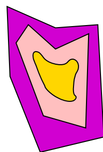

1. Practical and Theoretical Foundations of Programming
   - CS vs. SE; Abstraction for complex algorithms; Algorithmic efficiency.
2. Iterative Sorting Algorithms
   - Specification; Selection, Insertion and Bubble sorts.
3. Recursion
   - Principles; Practice; Recursive sorts; Non-recursive From; Backtracking.
4. Software Correction
   - Introduction; Specifying Systems; Hoare Logic; Proving Recursive Functions.
5. Testing Software
   - Testing techniques; Testing strategies; JUnit; Design By Contract.

# 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
- Comparing Algorithms' Efficiency
  - Best case, worst case, average analysis
  - Asymptotic complexity
- Algorithmic Stability
- Conclusion

# Problems



Provided by clients (or teachers ;)

## Problems

- Problems are generic

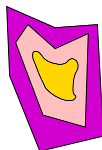
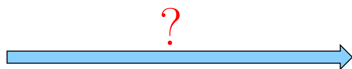
  Example: Determine the minimal value of a set of integers

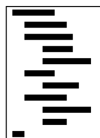
## Instances of a problem

- The problem for a given data set

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

# Problems and Programs



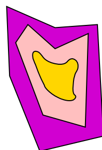
Problem

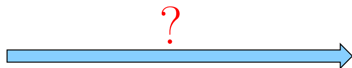
Software System

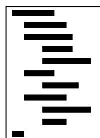
Software systems (*ie.*, Programs)

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

# Problems and Programs





Problem

Software System

## Software systems (*ie.*, Programs)

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

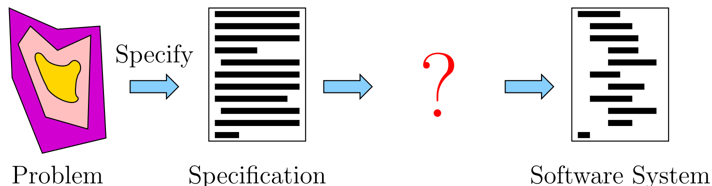
## Problem Specification

- 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

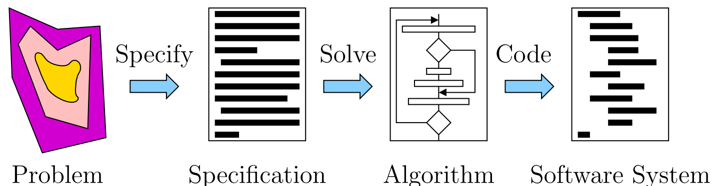
## Divide, Conquer and Glue (top-down approach)

- 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, Wheal, 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 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

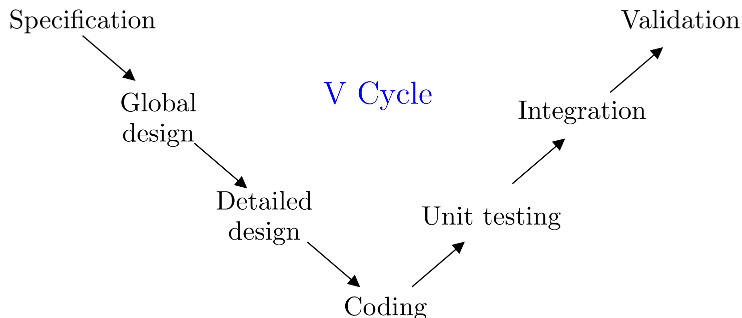
# Software Engineering vs. Computer Science

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


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

# 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: Divide complexity into manageable units
- Polymorphism: Use one (specialized) block instead of the expected one
- Heritage: Allow to factorize behavior between related units
- In short, 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: Divide complexity into manageable units
- Polymorphism: Use one (specialized) block instead of the expected one
- Heritage: Allow to factorize behavior between related units
- In short, 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 OOP before algorithms and not the contrary?

- Every mainstream languages (including Java) are object oriented
- Research in CS education shows that "object first" is a better approach

# 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:

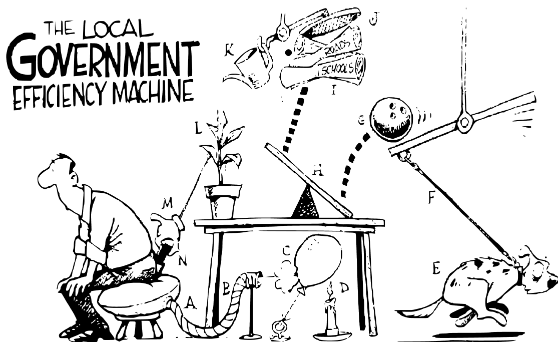
- each component has a carefully specified function
- components are easily integrated
- i.e. the speakers are easily connected to the amplifier

# 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)

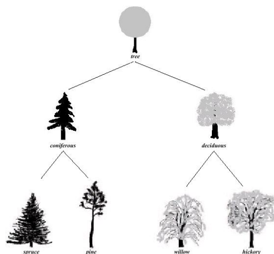
## Rube Goldberg's toothpaste dispenser

Such over engineered solutions should obviously remain jokes

# Dealing with complexity: Abstraction

## Abstraction

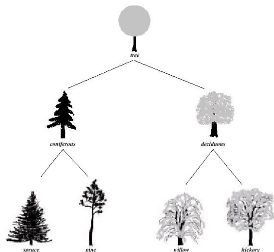
- Dealing with components and interactions without worrying about details
- Not "vague" or "imprecise", but focused on few relevant properties
- Elimination of the irrelevant and amplification of the essential
- Capturing commonality between different things



# Dealing with complexity: Abstraction

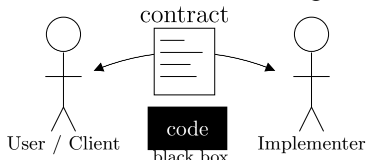
## Abstraction

- 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

# 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
- Comparing Algorithms' Efficiency
  - Best case, worst case, average analysis
  - Asymptotic complexity
- Algorithmic Stability
- Conclusion

# 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, ...
- ⇒ 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 an array

```
boolean linearSearch(int val, int[ ] tab) {
  for (int i=0; i<tab.length; i=i+1)
    if (tab[i] == val)
      return true;
  return false;
}
```

- 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 an array

```
boolean linearSearch(int val, int[ ] tab) {
  for (int i=0; i<tab.length; i=i+1)
    if (tab[i] == val)
      return true;
  return false;
}
```

- 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 (int i=0; i<tab.length; i=i+1)
  if (tab[i] == val)
    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++)} 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 (int i=0; i<tab.length; i=i+1)
  if (tab[i] == val)
    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++)} 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 (int i=0; i<tab.length; i=i+1)
  if (tab[i] == val)
    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++)} 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 (int i=0; i<tab.length; i=i+1)
  if (tab[i] == val)
    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++)} 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 (int i=0; i<tab.length; i=i+1)
  if (tab[i] == val)
    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++)} 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

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

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

array initialization

```
for (int i=0;i<tab.length;i++)
  tab[i] = 0;
```

- 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

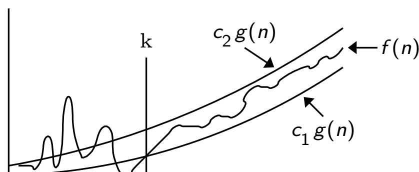
## Example: $T(n) = c_1 \times n^2 + c_2 \times n$

- $T(n) = c_1 \times n^2 + c_2 \times n \geq c_1 \times n^2 \qquad \forall n > 1$
  $T(n) \geq c \times n^2$ 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<br>100n | 10<br>1000 | 1000<br>$10^5$ | $10^5$<br>$10^7$ | seconds |
| $\Theta(n^2)$ | $n^2$<br>$100n^2$ | 100<br>$10^4$ | $10^6$<br>$10^8$ | $10^{10}$<br>$10^{12}$ | minutes |
| $\Theta(n^3)$ | $n^3$<br>$100n^2$ | 1000<br>$10^5$ | $10^9$<br>$10^{11}$ | $10^{15}$<br>$10^{17}$ | hours |
| $\Theta(2^n)$ | $2^n$<br>$100 \times 2^n$ | 1024<br>$> 10^5$ | $> 10^{301}$<br>$10^{305}$ | $\infty$<br>$\infty$ | ... |
| $\log(n)$ | $\log(n)$<br>$100 \log(n)$ | 3.3<br>332.2 | 9.9<br>996.5 | 16.6<br>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 $\Theta()$ – 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 $\mathrm{A(n)}+\mathrm{B(n)} = \mathrm{O(max(f(n), g(n)))} = \mathrm{O(f(n)+g(n))}$

- 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: `a=b;` $\Rightarrow \Theta(1)$ (constant time)

Example 2

```
sum=0;
for (i=0;i<n;i++)
  sum += n;
```

$\Theta(n)$

Example 3

```
sum=0;
for (i=0;i<n;i++)
  for (j=0;j<n;j++)
    sum ++;
for (k=0;k<n;k++)
  A[k] = k;
```

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

Example 4

```
sum=0;
for (i=0;i<n;i++)
  for (j=0;j<i;j++)
    sum ++;
```

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

Example 5

```
sum=0;
for (i=0;i<n;i*=2)
  sum ++;
```

$\Theta(\log(n))$ log is due to the $i \times 2$

# 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)

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
- Comparing Algorithms' Efficiency
  - Best case, worst case, average analysis
  - Asymptotic complexity
- Algorithmic Stability
- Conclusion

# Algorithmic stability

Computers use fixed precision numbers

- $10+1=11$

# Algorithmic stability

Computers use fixed precision numbers

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

# Algorithmic stability

Computers use fixed precision numbers

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

# Algorithmic stability

## Computers use fixed precision numbers

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

# Algorithmic stability

## Computers use fixed precision numbers

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

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

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

## 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


- Problem Specification
- Selection Sort
  - Presentation
  - Discussion
- Insertion Sort
  - Presentation
- Bubble Sort
  - Presentation
- Conclusion

# 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[2] $< a'_1, a'_2, a'_3, \ldots, a'_N >$ 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 array. Memory complexity: O(1)

[2] 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 |
|---|---|---|---|---|---|---|---|

# 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 |
|---|---|---|---|---|---|---|---|

# 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 |

# 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 |

# 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 |

# 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 |

# 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 |

# 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 |

# 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 |

# 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 |

# 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 |

### Pseudo-code:

```
For each elements, do:

  (1) search min on [i;N]

  (2) put min first
```

# 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 |

### Pseudo-code:

```
/* For each elements, do:  */
for (i=0; i<length; i++) {
  /* (1) search min on [i;N] */




  /* (2) put min first */



 }
```

# 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 |

### Pseudo-code:

```
/* For each elements, do:  */
for (i=0; i<length; i++) {
  /* (1) search min on [i;N] */
  minpos=i;
  for (j=i; j<length; j++)
    if (tab[j] < tab[minpos])
      minpos = j;
  /* (2) put min first */



 }
```

# 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 |

### Pseudo-code:

```
/* For each elements, do:  */
for (i=0; i<length; i++) {
  /* (1) search min on [i;N] */
  minpos=i;
  for (j=i; j<length; j++)
    if (tab[j] < tab[minpos])
      minpos = j;
  /* (2) put min first */
  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; i<length; i++) {
  minpos=i;
  for (j=i; j<length; j++)
    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; i<length; i++) {
  minpos=i;
  for (j=i; j<length; j++)
    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; i<length; i++) {
  minpos=i;
  for (j=i; j<length; j++)
    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; i<length; i++)

  for (j=i; j<length; j++)
```

## 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

# 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; i<length; i++)

  for (j=i; j<length; j++)
```

## 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; i<length; i++)
  minpos=i;
  for (j=i; j<length; j++)
    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; i<length; i++)
  minpos=i;
  for (j=i; j<length; j++)
    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; i<length; i++)
  minpos=i;
  for (j=i; j<length; j++)
    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; i<length; i++)
  minpos=i;
  for (j=i; j<length; j++)
    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; i<length; i++)
  minpos=i;
  for (j=i; j<length; j++)
    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; i<length; i++)
  minpos=i;
  for (j=i; j<length; j++)
    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; i<length; i++)
  minpos=i;
  for (j=i; j<length; j++)
    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?

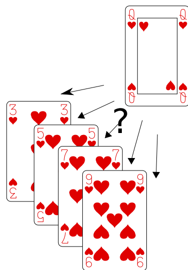
- No human would apply *selection sort* to sort a deck!

## 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:

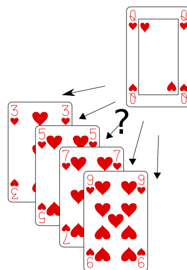
1. If the cards #1 and #2 need to be swapped, do it
2. Insert card #3 at its position in the [1,2] part of the deck
3. Insert card #4 at its position in the [1,3] part of the deck

   ...

## 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

   ...

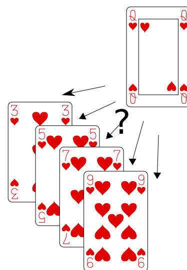
## Finding the common pattern

- 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

   ...

## 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)

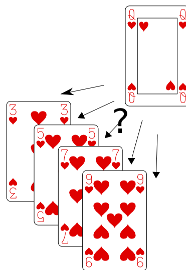
## This is *Insertion Sort*

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

# 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

   ...

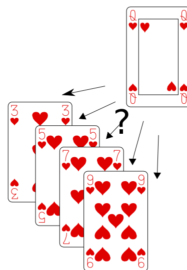
## Finding the common pattern

- 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 |

# 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)

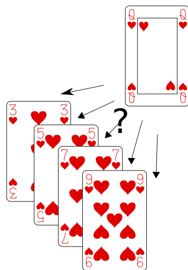
## This is *Insertion Sort*

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

# 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

   . . .

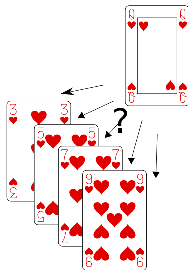
## Finding the common pattern

- 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)

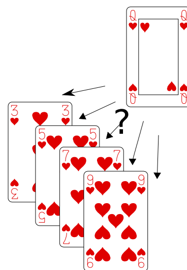
## This is *Insertion Sort*

| 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 ($\geq$ 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)

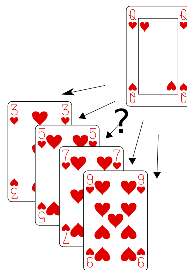
## This is *Insertion Sort*

| 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 |
| 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 ($\geq 2$) 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)

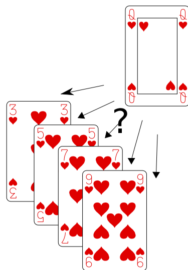
## This is *Insertion Sort*

| 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 |
| 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 ($\geq 2$) 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)

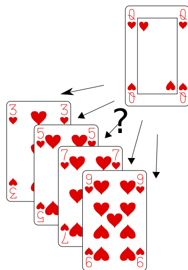
## This is *Insertion Sort*

| 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 |
| 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

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 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1. | N | S | U | | R | T | E | D | tmp | O |
| 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 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1. | N | S | U | | R | T | E | D | tmp | O |
| 2. | N | S | | U | R | T | E | D | tmp | O |
| 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 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1. | N | S | U | | R | T | E | D | tmp | O |
| 2. | N | S | | U | R | T | E | D | tmp | O |
| 3. | N | | S | U | R | T | E | D | tmp | O |
| 4. | N | O | S | U | R | T | E | D | tmp | |

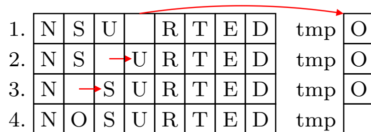
# Writing the insertion sort algorithm

## 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 |




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

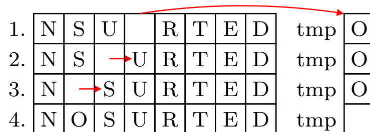
# Writing the insertion sort algorithm

## 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 |




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

```
/* for each element */

  /* save current value */

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




  /* Put value in cleared position */
```

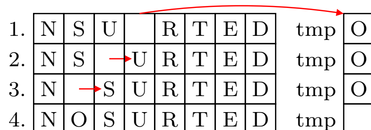
# Writing the insertion sort algorithm

## 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 |




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

```
/* for each element */
for (i=0; i<length; i++) {
  /* save current value */

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





  /* Put value in cleared position */
```

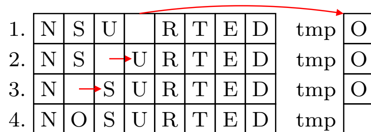
# Writing the insertion sort algorithm

## 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 |




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

```
/* for each element */
for (i=0; i<length; i++) {
  /* save current value */
  int value = tab[i];
  /* shift to right any element on the left being smaller than value */





  /* Put value in cleared position */
```

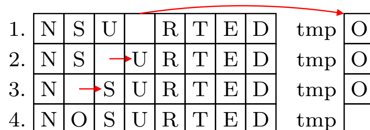
# Writing the insertion sort algorithm

## 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 |




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

```
/* for each element */
for (i=0; i<length; i++) {
  /* save current value */
  int value = tab[i];
  /* shift to right any element on the left being smaller than value */
  int j=i;
  while ((j > 0) && (tab[j-1]>value)) {
    tab[j] = tab[j-1];
    j–;
  }
  /* Put value in cleared position */
```

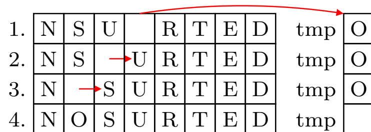
# Writing the insertion sort algorithm

## 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 |




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

```
/* for each element */
for (i=0; i<length; i++) {
  /* save current value */
  int value = tab[i];
  /* shift to right any element on the left being smaller than value */
  int j=i;
  while ((j > 0) && (tab[j-1]>value)) {
    tab[j] = tab[j-1];
    j–;
  }
  /* Put value in cleared position */
  tab[j]=value;
```

# 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 (int i=0; i<length-1; i++)
  /* 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 (int i=0; i<length-1; i++)
   /* 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

```
int 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 (int i=0; i<length-1; i++)
  /* 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

```
int tmp=tab[i];
tab[i]=tab[i+1];
tab[i+1]=tmp;
```

## All together

- Add boolean variable to check whether it sorted

```
boolean swapped;
do {
  swapped = false;
  for (int i=0; i<length-1; i++)
    /* if these two values are badly sorted */
    if (tab[i]>tab[i+1]) {
      /* swap them */
      int tmp=tab[i];
      tab[i]=tab[i+1];
      tab[i+1]=tmp;
      /* and remember we swapped something */
      swapped = true;
    }
} while (swapped);/* 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


- 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
- Avoiding Recursion
  - Non-Recursive Form of Tail Recursion
  - Transformation to Tail Recursion
  - Generic Algorithm Using a Stack
- Back-tracking
- Conclusion

# 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

# 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:
  - $U(n) = 3 \times U(n-1) + 1$ ; $\underline{U(0) = 1}$
  - 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
- Avoiding Recursion
  - Non-Recursive Form of Tail Recursion
  - Transformation to Tail Recursion
  - Generic Algorithm Using a Stack
- Back-tracking
- Conclusion

# 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
  T I R N E G S $\rightsquigarrow$ 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

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

Remarks:

- $r \leftarrow 1$ is the base case: no recursive call
- $r \leftarrow n \times factorial(n-1)$ 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 ← 1
           else r ← n × factorial(n − 1)
  end
```

$factorial(4) =$

# Factorial Computation Details

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

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

# Factorial Computation Details

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

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

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

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

# Factorial Computation Details

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

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

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

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

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

# Factorial Computation Details

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

$$factorial(4) = 4 \times \underbrace{factorial(3)}$$

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

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

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

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

# Factorial Computation Details

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

$$factorial(4) = 4 \times \overbrace{3 \times \overbrace{2 \times \overbrace{1 \times factorial(0)}^{factorial(1)}}^{factorial(2)}}^{factorial(3)}$$

Recursive Descent

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

Base Case

# Factorial Computation Details

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

$$factorial(4) = 4 \times \overbrace{factorial(3)}^{3 \times \overbrace{factorial(2)}^{2 \times \overbrace{factorial(1)}^{1 \times factorial(0)}}}$$

Recursive Descent

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

Base Case

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

# Factorial Computation Details

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

$$
factorial(4) = 4 \times \overbrace{factorial(3)}^{} \;\; 3 \times \overbrace{factorial(2)}^{} \;\; 2 \times \overbrace{factorial(1)}^{} \;\; 1 \times factorial(0)
$$

Recursive Descent

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

Base Case

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

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

# Factorial Computation Details

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

$$factorial(4) = 4 \times \overbrace{3 \times \overbrace{2 \times \overbrace{1 \times factorial(0)}^{factorial(1)}}^{factorial(2)}}^{factorial(3)}$$

Recursive Descent

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

Base Case

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

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

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

# Factorial Computation Details

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

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

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

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

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

} Recursive Descent

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

} 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
```

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

$$3 \times factorial(2)$$

$$2 \times factorial(1)$$

$$1 \times factorial(0)$$

Recursive Descent

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

Base Case

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

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

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

$$\overbrace{24}$$

Recursive Climb

$$factorial(4) = 24$$

# 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
- Avoiding Recursion
  - Non-Recursive Form of Tail Recursion
  - Transformation to Tail Recursion
  - Generic Algorithm Using a Stack
- Back-tracking
- Conclusion

# General Recursion Schema

```
if COND then BASECASE
         else GENCASE
end
```

- COND is a boolean expression
- If COND is true, execute the base case BASECASE (without recursive call)
- If COND is false, execute the general case GENCASE (with recursive calls)

## The factorial(n) example

BASECASE: $r \leftarrow 1$

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)

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

- $\boxed{\binom{4}{2} = 6}$ $\leadsto$ 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, $4 \times 3$ 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

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

### 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:

Ackerman($m$, $n$)
  **if** $m = 0$ **then** $n + 1$
    **else if** $n = 0$ **then** Ackerman($m - 1$, $1$)
      **else** Ackerman($m - 1$, 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:

ACKERMAN($m$, $n$)
  **if** $m = 0$ **then** $n + 1$
    **else if** $n = 0$ **then** ACKERMAN($m - 1$, $1$)
      **else** ACKERMAN($m - 1$, ACKERMAN($m$, $n - 1$))

Warning, this function grows quickly:

$Ack(1,n) = n + 2$ $\quad$ $Ack(2,n) = 2n + 3$

$Ack(3,n) = 8 \cdot 2^n - 3$ $\quad$ $Ack(4,n) = \left.2^{2^{2^{\cdots^2}}}\right\}n$

$Ack(4,4) > 2^{65536} > 10^{80}$ (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"

- Right after TOP in track

# Example: a list type

## 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 constructs are the base of the LISP programing language

# Implantation in Java

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

String Class representing a string (either empty or not)

**The Element class**

```
public class Element {
  public char value;
  public Element rest;

  // Constructor
  Element(char x, Element rest) {
      value = x;
      this.rest = rest;
  }
}
```

**The String class**

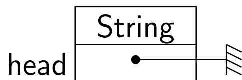
```
public class String {
  private Element head;

  // Constructor -- gives an empty string
  String() {
      head = null;
  }
  // Methods
  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;
} }
```

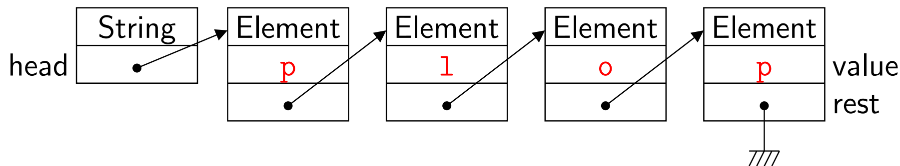
Need 2 classes to distinguish between empty string and uninitialized string variable

# Some Memory Representation Examples

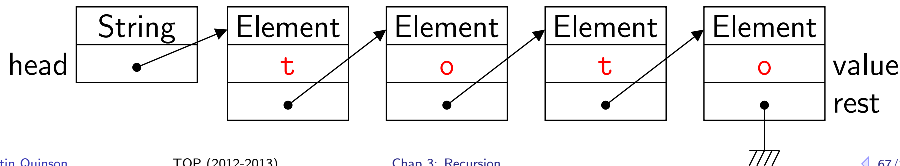
## Empty String




## String containing “plop”




## String containing “toto”



# 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 one thing that you can only learn through experience

# Third Chapter

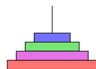
## Recursion

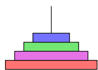

- Introduction
- Principles of Recursion
  - First Example: Factorial
  - Schemas of Recursion
  - Recursive Data Structures
- Recursion in Practice
  - Solving a Problem by Recursion: Hanoi Towers
  - Classical Recursive Functions
  - Recursive Sorting Algorithms
- Avoiding Recursion
  - Non-Recursive Form of Tail Recursion
  - Transformation to Tail Recursion
  - Generic Algorithm Using a Stack
- Back-tracking
- Conclusion

# 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

$\leadsto$ We recurse on integer $n$

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

$\leadsto$ 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

⇝ We recurse on integer $n$

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

⇝ Decomposition between bigger disk and (n-1) smaller ones

- We want to write procedure HANOI(N, FROM, TO).
  It moves the N disks from stick FROM to stick TO

⇝ For simplicity sake, we introduce procedure MOVE(FROM,TO)
It moves the upper disk from stick FROM to stick 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

⇝ We recurse on integer $n$

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

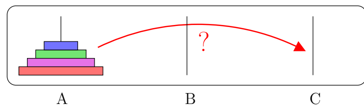
⇝ Decomposition between bigger disk and (n-1) smaller ones

- We want to write procedure HANOI(N, FROM, TO).
  It moves the N disks from stick FROM to stick TO

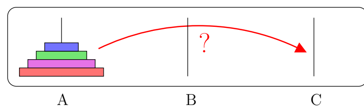
⇝ For simplicity sake, we introduce procedure MOVE(FROM,TO)
It moves the upper disk from stick FROM to stick TO
(also checks that we don't move a big one over a small one)

- Stopping Condition: when only one disk remains, use MOVE
  HANOI(1,X,Y)=MOVE(X,Y)

# Possible Decomposition of Hanoi(n, A, C)

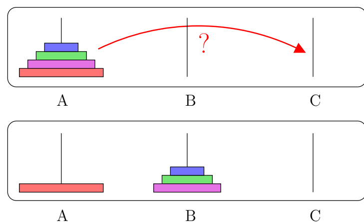


# Possible Decomposition of Hanoi(n, A, C)



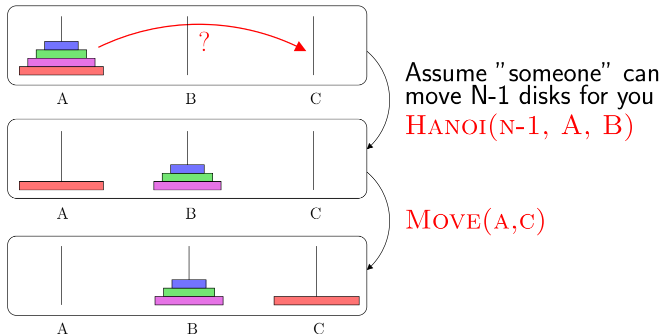

Assume "someone" can move N-1 disks for you

# Possible Decomposition of Hanoi(n, A, C)




Assume "someone" can move N-1 disks for you
HANOI(N-1, A, B)

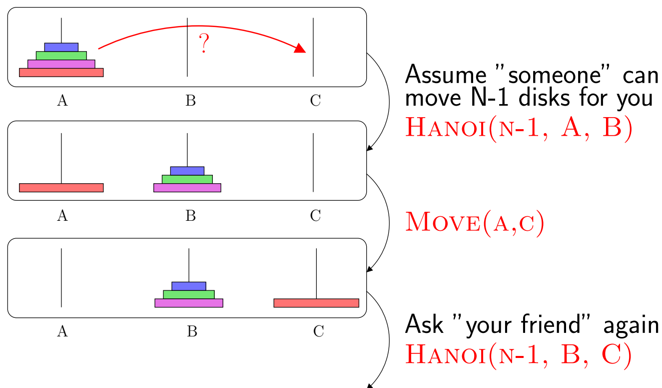
# Possible Decomposition of Hanoi(n, A, C)




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

Move(a,c)

# Possible Decomposition of Hanoi(n, A, C)

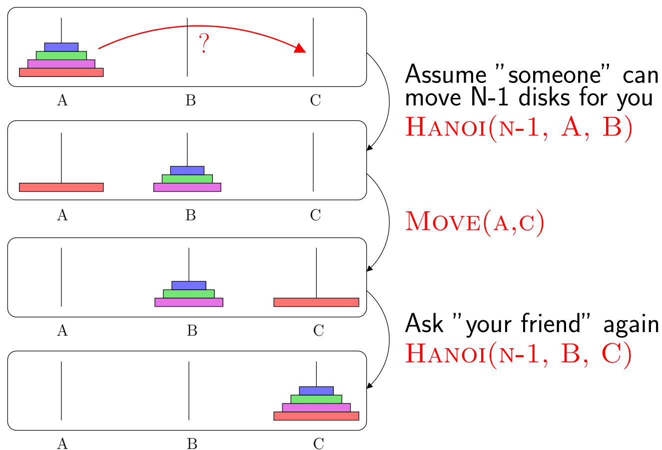



Assume "someone" can move N-1 disks for you
HANOI(N-1, A, B)

MOVE(A,C)

Ask "your friend" again
HANOI(N-1, B, C)

# Possible Decomposition of Hanoi(n, A, C)

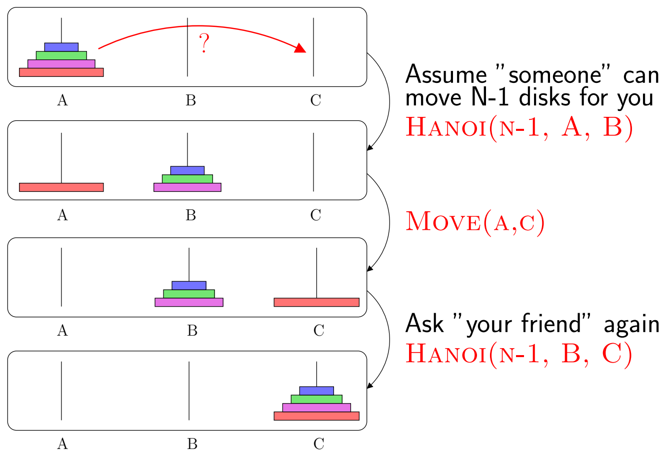



Assume "someone" can move N-1 disks for you
HANOI(N-1, A, B)

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.* – *Anonymous*

# Corresponding Algorithm

```
HANOI(n,a,b):
  if n = 1 then Move(a,b)
           else  Hanoi(n-1, a, c)
                 Move(a, b)
                 Hanoi(n-1, c, b)
  end
```

# Corresponding Algorithm

```
HANOI(n,a,b):
  if n = 1 then Move(a,b)
           else Hanoi(n-1, a, c)
                Move(a, b)
                Hanoi(n-1, c, b)
  end
```

Variant with 0 as base case

```
HANOI(n,a,b):
  if n ≠ 0 then Hanoi(n-1, a, c)
                Move(a, b)
                Hanoi(n-1, c, b)
  end
```

# Back on the Hanoï 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, ...) 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

- 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
- Avoiding Recursion
  - Non-Recursive Form of Tail Recursion
  - Transformation to Tail Recursion
  - Generic Algorithm Using a Stack
- Back-tracking
- Conclusion

# 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
- $F_0 = 0$ ; $F_1 = 1$ ; $F_2 = 1$ ; $F_3 = 2$ ; $F_4 = 3$ ; $F_5 = 5$ ; $F_6 = 8$ ; $F_7 = 13$ ; ...

# 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
- $F_0 = 0$ ; $F_1 = 1$ ; $F_2 = 1$ ; $F_3 = 2$ ; $F_4 = 3$ ; $F_5 = 5$ ; $F_6 = 8$ ; $F_7 = 13$ ; ...

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

# 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
- $F_0 = 0$ ; $F_1 = 1$ ; $F_2 = 1$ ; $F_3 = 2$ ; $F_4 = 3$ ; $F_5 = 5$ ; $F_6 = 8$ ; $F_7 = 13$ ; ...

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

**Corresponding Algorithm**

```
static int fib(int n) {
  if (n <= 1)
    return n; // Base Case
  else
    return 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 Algorithm**

```
static int fib(int n) {
  if (n <= 1)
    return n; // Base Case
  else
    return 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
- $F_0 = 0$ ; $F_1 = 1$ ; $F_2 = 1$ ; $F_3 = 2$ ; $F_4 = 3$ ; $F_5 = 5$ ; $F_6 = 8$ ; $F_7 = 13$ ; ...

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

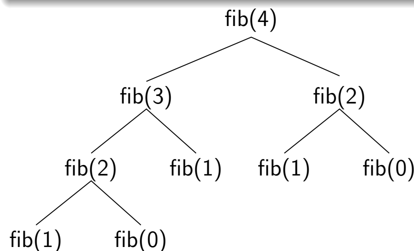
**Corresponding Algorithm**

```
static int fib(int n) {
  if (n <= 1)
    return n; // Base Case
  else
    return fib(n-1) + fib(n-2);
}
```

(efficient implementations exist)

**Exercice :**

Compute amount of recursive calls



# 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

$\textsc{Syracuse}(n)$:
**if** $n = 0$ or $n = 1$ **then** 1
**else if** $n \bmod 2 = 0$ **then** $\textsc{Syracuse}(n/2)$
**else** $\textsc{Syracuse}(3 \times n + 1)$
**end**

- Question: Does this function always terminate?
  Hard to say: suite is not monotone

# Classical Recursive Function: Syracuse

```
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)
  end
```

- 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

$\textsc{Syracuse}(n)$:
**if** $n = 0$ or $n = 1$ **then** 1
**else if** $n \bmod 2 = 0$ **then** $\textsc{Syracuse}(n/2)$
**else** $\textsc{Syracuse}(3 \times n + 1)$
**end**

- 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 $\forall n < 19 \cdot 2^{58} \approx 5 \cdot 10^{48}$
  (but other conjectures were proved false for bigger values only)

# Classical Recursive Function: Syracuse

```
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)
  end
```

- 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 $\forall n < 19 \cdot 2^{58} \approx 5 \cdot 10^{48}$

  (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


- 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
- Avoiding Recursion
  - Non-Recursive Form of Tail Recursion
  - Transformation to Tail Recursion
  - Generic Algorithm Using a Stack
- Back-tracking
- Conclusion

# 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. JLM'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 dupplicates?
- 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

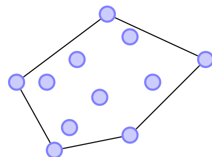
## Median and Selection

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

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

# Merge Sort

## Pseudo-code

```
function merge_sort(m)
    var list left, right, result
    if length(m) <= 1
        return m

    var middle = length(m) / 2
    for each x in m up to middle
         add x to left
    for each x in m after middle
         add x to right
    left = merge_sort(left)
    right = merge_sort(right)
    result = merge(left, right)
    return result
```

(C) Wikipedia

```
function merge(left,right)
    var list result
    while length(left)>0 and length(right)>0
        if first(left) <= first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
    end while
    while length(left) > 0
        append left to result
    while length(right) > 0
        append right to result
    return result
```

## 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) $+ \mathit{log}(n)$ 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

## Building sub-lists makes it easy:

- Create two empty list variables
- Iterate over the original list, and put elements in correct sublist
- Recurse
- Concatenate results

```
function quicksort(array)
    var list less, greater
    if length(array) <= 1
        return array
    select and remove a pivot value pivot from array
    for each x in array
        if x <= pivot then append x to less
        else append x to greater
    return concatenate(quicksort(less), pivot, quicksort(greater))
```

(C)wikipedia

## Problem

- Space complexity is about $2n + \log(n))$...
  (2n for array dupplication, 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

| 3 | 7 | 8 | 5 | 2 | 1 | 9 | 5 | 4 |
|---|---|---|---|---|---|---|---|---|
| 3 | 7 | 8 | 4 | 2 | 1 | 9 | 5 | 5 |
| 3 | 7 | 8 | 4 | 2 | 1 | 9 | 5 | 5 |
| 3 | 7 | 8 | 4 | 2 | 1 | 9 | 5 | 5 |
| 3 | 7 | 8 | 4 | 2 | 1 | 9 | 5 | 5 |
| 3 | 4 | 8 | 7 | 2 | 1 | 9 | 5 | 5 |
| 3 | 4 | 2 | 7 | 8 | 1 | 9 | 5 | 5 |
| 3 | 4 | 2 | 1 | 8 | 7 | 9 | 5 | 5 |
| 3 | 4 | 2 | 1 | 8 | 7 | 9 | 5 | 5 |
| 3 | 4 | 2 | 1 | 5 | 7 | 9 | 8 | 5 |
| 3 | 4 | 2 | 1 | 5 | 5 | 9 | 8 | 7 |

```
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right] // Move pivot to end
   storeIndex := left
   for i  from  left to right - 1
       if array[i] <= pivotValue
           swap array[i] and array[storeIndex]
           storeIndex := storeIndex + 1
   swap array[storeIndex] and array[right] // Move pivot to its final place
   return storeIndex
```

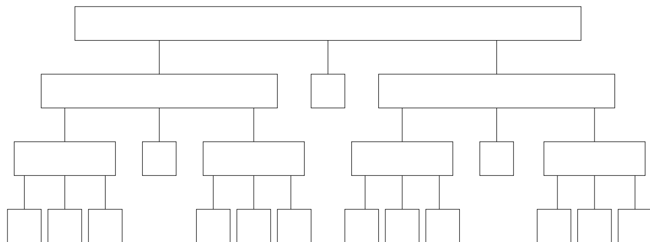
(C) wikipedia

# 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%)?

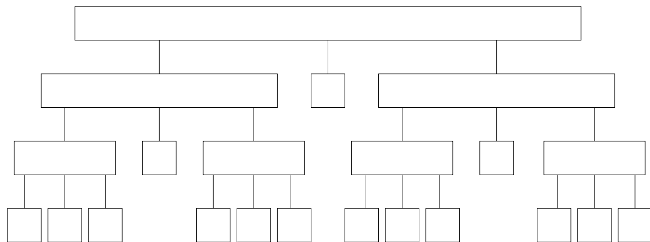
- We get steps $\leadsto$ whole algorithm in

# 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%)?

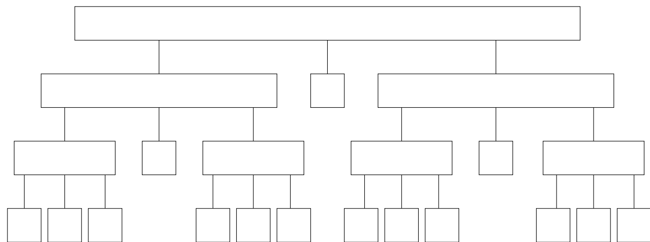
- We get $100 \times \log(n)$ steps $\rightsquigarrow$ whole algorithm in

# 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:



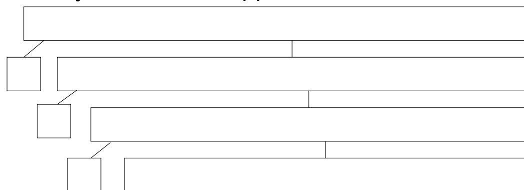
What if we split 1%/99% at each step (instead of 50%/50%)?

- We get $100 \times \log(n)$ steps $\leadsto$ 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 dupplicated, or with the wrong pivot)



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

## That's a fairly bad worst case time

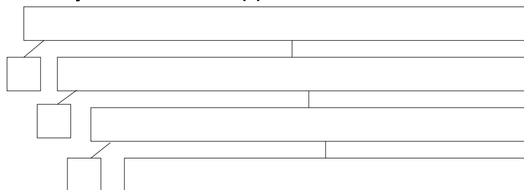
- Worst than MergeSort, for example
- But called Quicksort anyway because faster *in practice* than MergeSort
- In-Place version of both algorithms are not stable
- Both can be quite easily parallelized
- Space complexity: $O(\log(n))$ (to store the recursion stack)

(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 dupplicated, 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

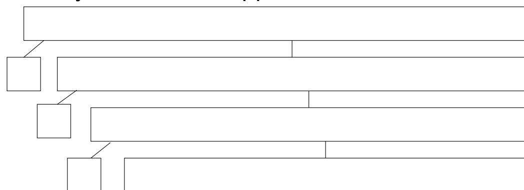
- Worst than MergeSort, for example
- But called Quicksort anyway because faster *in practice* than MergeSort
- In-Place version of both algorithms are not stable
- Both can be quite easily parallelized
- Space complexity: $O(\log(n))$ (to store the recursion stack)

(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 dupplicated, 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)

# 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
- Avoiding Recursion
  - Non-Recursive Form of Tail Recursion
  - Transformation to Tail Recursion
  - Generic Algorithm Using a Stack
- Back-tracking
- Conclusion

# Why do you want to avoid recursion

## What gets done on Function Calls

1. Create a function frame on the stack
2. Push (copy) value of parameters
3. Execute function
4. Pop return value
5. Destruct stack frame

## Recursion does not interfere with this schema

- Recursion can thus be less efficient than iterative solutions
- In time: function calling has a price
- In space: the call stack must be stored

# Example: gcd of two natural integers

Greatest Common Divisor

gcd(a, b : Integer) = (r : Integer)

- Precondition: $a \geq b \geq 0$
- Postcondition: $(a \bmod r = 0)$ and $(b \bmod r = 0)$ and $\neg(\exists s, (s > r) \wedge (a \bmod s = 0) \wedge (b \bmod s = 0))$

## Recursive Definition

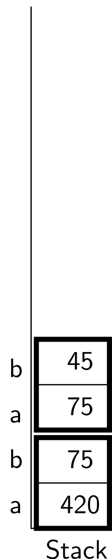
$$\begin{array}{l} \textbf{if } b = 0 \textbf{ then } r \leftarrow a \\ \qquad\qquad \textbf{else } r \leftarrow pgcd(b, a \bmod b) \end{array}$$

# Computation of gcd(420,75)

$$\begin{aligned}&\textbf{if } b = 0 \textbf{ then } r \leftarrow a\\ &\qquad\qquad \textbf{else } \; r \leftarrow gcd(b, a \bmod b)\end{aligned}$$

- $gcd(420, 75) =$

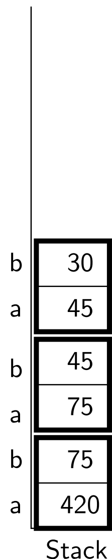
# Computation of gcd(420,75)




**if** $b = 0$ **then** $r \leftarrow a$
**else** $r \leftarrow gcd(b, a \bmod b)$

- $gcd(420, 75) = gcd(75, 45)$
- $gcd(75, 45) =$

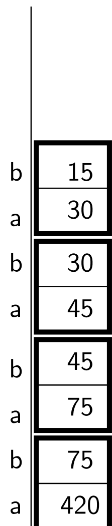
# Computation of gcd(420,75)




**if** $b = 0$ **then** $r \leftarrow a$
**else** $r \leftarrow gcd(b, a \bmod b)$

- $gcd(420, 75) = gcd(75, 45)$
- $gcd(75, 45) = gcd(45, 30)$
- $gcd(45, 30) =$

# Computation of gcd(420,75)




**if** $b = 0$ **then** $r \leftarrow a$
**else** $r \leftarrow gcd(b, a \bmod b)$

- $gcd(420, 75) = gcd(75, 45)$
- $gcd(75, 45) = gcd(45, 30)$
- $gcd(45, 30) = gcd(30, 15)$
- $gcd(30, 15) =$

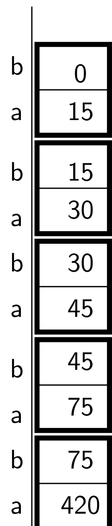
# Computation of gcd(420,75)

| | Stack |
|---|---|
| b | 0 |
| a | 15 |
| b | 15 |
| a | 30 |
| b | 30 |
| a | 45 |
| b | 45 |
| a | 75 |
| b | 75 |
| a | 420 |

$$\textbf{if } b = 0 \textbf{ then } r \leftarrow a$$
$$\textbf{else } r \leftarrow gcd(b, a \bmod b)$$

- $gcd(420, 75) = gcd(75, 45)$
- $gcd(75, 45) = gcd(45, 30)$
- $gcd(45, 30) = gcd(30, 15)$
- $gcd(30, 15) = gcd(15, 0)$
- $gcd(15, 0) =$

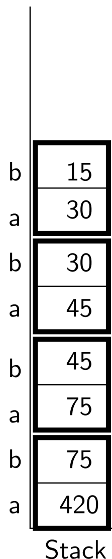
# Computation of gcd(420,75)




| |
|---|
| **if** $b = 0$ **then** $r \leftarrow a$<br>**else** $r \leftarrow gcd(b, a \bmod b)$ |

- $gcd(420, 75) = gcd(75, 45)$
- $gcd(75, 45) = gcd(45, 30)$
- $gcd(45, 30) = gcd(30, 15)$
- $gcd(30, 15) = gcd(15, 0)$
- $gcd(15, 0) = 15$
  this is the Base Case

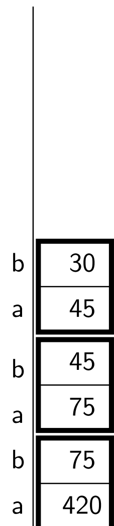
# Computation of gcd(420,75)




**if** $b = 0$ **then** $r \leftarrow a$
**else** $r \leftarrow gcd(b, a \bmod b)$

- $gcd(420, 75) = gcd(75, 45)$
- $gcd(75, 45) = gcd(45, 30)$
- $gcd(45, 30) = gcd(30, 15)$
- $gcd(30, 15) = gcd(15, 0)$
- $gcd(15, 0) = 15$
  this is the Base Case
- Let's pop parameters
- $r \leftarrow r_{int}$ (no other computation: $G(x, y) = y$)

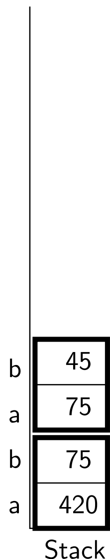
# Computation of gcd(420,75)




**if** $b = 0$ **then** $r \leftarrow a$
**else** $r \leftarrow gcd(b, a \bmod b)$

- $gcd(420, 75) = gcd(75, 45)$
- $gcd(75, 45) = gcd(45, 30)$
- $gcd(45, 30) = gcd(30, 15)$
- $gcd(30, 15) = gcd(15, 0) \mathbf{= 15}$
- $gcd(15, 0) = 15$
  this is the Base Case
- Let's pop parameters
- $r \leftarrow r_{int}$ (no other computation: $G(x, y) = y$)

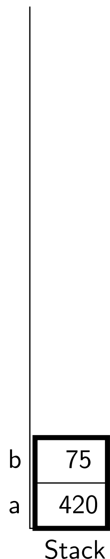
# Computation of gcd(420,75)




| **if** $b = 0$ **then** $r \leftarrow a$ |
|---|
| **else** $r \leftarrow gcd(b, a \bmod b)$ |

- $gcd(420, 75) = gcd(75, 45)$
- $gcd(75, 45) = gcd(45, 30)$
- $gcd(45, 30) = gcd(30, 15) \mathbf{= 15}$
- $gcd(30, 15) = gcd(15, 0) \mathbf{= 15}$
- $gcd(15, 0) = 15$
  this is the Base Case
- Let's pop parameters
- $r \leftarrow r_{int}$ (no other computation: $G(x, y) = y$)

# Computation of gcd(420,75)




$$\textbf{if } b = 0 \textbf{ then } r \leftarrow a \quad \textbf{else } r \leftarrow gcd(b, a \bmod b)$$

- $gcd(420, 75) = gcd(75, 45)$
- $gcd(75, 45) = gcd(45, 30)$ **= 15**
- $gcd(45, 30) = gcd(30, 15)$ **= 15**
- $gcd(30, 15) = gcd(15, 0)$ **= 15**
- $gcd(15, 0) = 15$
  this is the Base Case
- Let's pop parameters
- $r \leftarrow r_{int}$ (no other computation: $G(x, y) = y$)

# Computation of gcd(420,75)

$$\textbf{if } b = 0 \textbf{ then } r \leftarrow a \quad \textbf{else } r \leftarrow gcd(b, a \bmod b)$$

Stack

- $gcd(420, 75) = gcd(75, 45)$ **= 15**
- $gcd(75, 45) = gcd(45, 30)$ **= 15**
- $gcd(45, 30) = gcd(30, 15)$ **= 15**
- $gcd(30, 15) = gcd(15, 0)$ **= 15**
- $gcd(15, 0) = 15$
  this is the Base Case
- Let's pop parameters
- $r \leftarrow r_{int}$ (no other computation: $G(x, y) = y$)
- The result of initial call is known as early as from Base Case
  This is known as **Tail Recursion**

# Computation of gcd(420,75)

$$\begin{array}{l} \textbf{if } b = 0 \textbf{ then } r \leftarrow a \\ \qquad\qquad \textbf{else } \; r \leftarrow gcd(b, a \bmod b) \end{array}$$

Stack

- $gcd(420, 75) = gcd(75, 45)$ **= 15**
- $gcd(75, 45) = gcd(45, 30)$ **= 15**
- $gcd(45, 30) = gcd(30, 15)$ **= 15**
- $gcd(30, 15) = gcd(15, 0)$ **= 15**
- $gcd(15, 0) = 15$
  this is the Base Case
- Let's pop parameters
- $r \leftarrow r_{int}$ (no other computation: $G(x, y) = y$)

- The result of initial call is known as early as from Base Case
  This is known as **Tail Recursion**
- Factorial: multiplications during climb up
  $\Rightarrow$ **non-terminal** recursion

# Transformation to Non-Recursive Form

Every recursive function can be changed to a non-recursive form

Several Methods depending on function:

- Tail Recursion: very simple transformation
- Non-Tail Recursion: two methods (only one is generic)

Compilers use these optimization techniques (amongst much others)

# Non-Recursive Form of Tail Recursion

## Cookbook to change Tail Recursion to Non-Recursive Form

- Generic recursive algorithm

```
f(x):
  if cond(x) then BaseCase(x)
             else T(x); r ← f(x_int)
```

# Non-Recursive Form of Tail Recursion

## Cookbook to change Tail Recursion to Non-Recursive Form

- Generic recursive algorithm

```
f(x):
  if cond(x) then BaseCase(x)
             else  T(x); r ← f(x_int)
```

- Equivalent iterative algorithm

```
f'(x):
  u ← x
  until cond(u) do
    T(u)
    u ← h(u)
  end
  BaseCase(u)
```

# Non-Recursive Form of Tail Recursion

## Cookbook to change Tail Recursion to Non-Recursive Form

- Generic recursive algorithm

```
f(x):
  if cond(x) then BaseCase(x)
             else T(x); r ← f(x_int)
```

Example: get last char of string

```
last(s):
  if empty(cdr(s)) then r ← car(s)
                   else r ← last(cdr(s))
```

- Equivalent iterative algorithm

```
f'(x):
  u ← x
  until cond(u) do
    T(u)
    u ← h(u)
  end
  BaseCase(u)
```

# Non-Recursive Form of Tail Recursion

## Cookbook to change Tail Recursion to Non-Recursive Form

- Generic recursive algorithm

```
f(x):
  if cond(x) then BASECASE(x)
             else T(x); r ← f(x_int)
```

Example: get last char of string

```
last(s):
  if empty(cdr(s)) then r ← car(s)
                   else r ← last(cdr(s))
```

- Equivalent iterative algorithm

```
f'(x):
  u ← x
  until cond(u) do
    T(u)
    u ← h(u)
  end
  BASECASE(u)
```

```
last'(s):
  l ← s
  until empty(cdr(l)) do
    // T(u) does nothing
    l=cdr(l)
  end
  r ← car(l)
```

# Other Examples

nth(s,i): get char number $i$ out of $s$

nth(s,i):
  **if** n=0 **then** $r \leftarrow car(s)$
  **else** $r \leftarrow nth(cdr(s), i-1)$

Two arguments, still no T(u)

# Other Examples

nth(s,i): get char number $i$ out of $s$

```
nth(s,i):
  if n=0 then r ← car(s)
         else r ← nth(cdr(s), i − 1)
```

Two arguments, still no T(u)

```
nth'(s,i):
  l ← s; k ← i
  until k=0 do
    l=cdr(l); k=k-1
  end
  r ← car(l)
```

# Other Examples

## nth(s,i): get char number $i$ out of $s$

```
nth(s,i):
  if n=0 then r ← car(s)
         else r ← nth(cdr(s), i − 1)
```

Two arguments, still no T(u)

```
nth'(s,i):
  l ← s; k ← i
  until k=0 do
    l=cdr(l); k=k-1
  end
  r ← car(l)
```

## is_member(s,c): assess whether $c$ is member of $s$

```
is_member(s,c):
  if empty(s) then r ← FALSE
  if car(s)=c then r ← TRUE
              else r ← is_memb(cdr(s))
```

2 base cases, still no T(u)

# Other Examples

## nth(s,i): get char number $i$ out of $s$

```
nth(s,i):
  if n=0 then r ← car(s)
         else r ← nth(cdr(s), i − 1)
```

```
nth'(s,i):
  l ← s; k ← i
  until k=0 do
    l=cdr(l); k=k-1
  end
  r ← car(l)
```

Two arguments, still no T(u)

## is_member(s,c): assess whether $c$ is member of $s$

```
is_member(s,c):
  if empty(s) then r ← FALSE
  if car(s)=c then r ← TRUE
              else r ← is_memb(cdr(s))
```

```
is_memb'(s,c):
  l ← s
  until empty(l) OR car(l)=c do
    l=cdr(l)
  end
  if empty(s) then r ← FALSE
  r ← TRUE
```

2 base cases, still no T(u)

# Last Example

## Non-Recursive Form of GCD

```
pgcd(a, b):
  if b = 0 then r ← a
            else  r ← gcd(b, a mod b)
```

```
pgcd′(a, b):
  u ← a; v ← b
  until v=0 do
    temp ← v
    v ← u mod v
    u ← temp
  end
  r ← u
```

- This is given by an immediate rewriting
- Computers are good at this kind of game (ie compilers)
- Meta-programming troubling at first sight, but still fully mechanic

# Non-Recursive form of Non-Tail functions

## Previous approach is not often applicable

- If the function you consider is non-tail, this method does not apply
- Looking closer, that's because of those computations at recursive climb:
- Where should the ongoing computation be stored (they were stacked)?
  $fact(3) = 3 \times fact(2) = 3 \times 2 \times fact(1) = 3 \times 2 \times 1 = 3 \times 2 = 6$

## What's done is no more to be done

- Do intermediate computations during descent, not waiting for climbing
  $fact(3) = \boxed{3} \times fact(2) = 3 \times 2 \times fact(1) = \boxed{6} \times fact(1) = \boxed{6} \times 1 = \boxed{6} = 6$
  There's nothing left to do during climbing $\Rightarrow$ Tail Recursion
- One extra variable is enough for the storage of "ongoing" computation
  - Since these computations are done, store their result not the stack of operations
  - Adding an extra parameter to my recursive function does the trick
  - Prototype change $\leadsto$ put recursion into a *helper* function with more parameters

## Warning: this does not always work!

- Computations done out of order $\leadsto$ must be associative and commutative
- This (simple) method does not always work; another one comes afterward

# Example: Changing Factorial into Tail Recursion

FACT(n):
 **if** n = 0 **then** $r \leftarrow 1$
 **else** $r \leftarrow n \times fact(n-1)$

$\lambda(n, acc)$ :

## Step-by-step approach

0. (it works because the addition is commutative and associative)
1. Create a lambda function doing the recursion, with more parameters
   - Local copy of the parameters carrying the recursion
   - Add as many accumulators as operations done on climb up

# Example: Changing Factorial into Tail Recursion

FACT(n):
  **if** n = 0 **then** $r \leftarrow 1$
  **else** $r \leftarrow n \times fact(n-1)$

FACT'(n): $\lambda(n, 1)$
$\lambda(n, acc)$ :

## Step-by-step approach

0. (it works because the addition is commutative and associative)
1. Create a lambda function doing the recursion, with more parameters
   - Local copy of the parameters carrying the recursion
   - Add as many accumulators as operations done on climb up
2. Main function simply calls the lambda function
   - Copy of the parameters carrying the recursion
   - Initialize accumulators to the identity element of their operation

# Example: Changing Factorial into Tail Recursion

FACT(n):
  **if** n $= 0$ **then** $r \leftarrow 1$
  **else** $r \leftarrow n \times fact(n-1)$

FACT'(n): $\lambda(n, 1)$
$\lambda(n, acc)$ :
  **if** n $= 0$
  **else** $r \leftarrow \lambda(n-1, acc \times n)$

## Step-by-step approach

0. (it works because the addition is commutative and associative)
1. Create a lambda function doing the recursion, with more parameters
   - Local copy of the parameters carrying the recursion
   - Add as many accumulators as operations done on climb up
2. Main function simply calls the lambda function
   - Copy of the parameters carrying the recursion
   - Initialize accumulators to the identity element of their operation
3. Body of the lambda function:
   - General treatment: as before, but do intermediate ops into the accumulators

# Example: Changing Factorial into Tail Recursion

FACT(n):
  **if** n $= 0$ **then** $r \leftarrow 1$
    **else** $r \leftarrow n \times fact(n-1)$

FACT'(n): $\lambda(n, 1)$
$\lambda(n, acc)$ :
  **if** n $= 0$ **then** $r \leftarrow acc$
    **else** $r \leftarrow \lambda(n-1, acc \times n)$

## Step-by-step approach

0. (it works because the addition is commutative and associative)
1. Create a lambda function doing the recursion, with more parameters
   - Local copy of the parameters carrying the recursion
   - Add as many accumulators as operations done on climb up
2. Main function simply calls the lambda function
   - Copy of the parameters carrying the recursion
   - Initialize accumulators to the identity element of their operation
3. Body of the lambda function:
   - General treatment: as before, but do intermediate ops into the accumulators
   - Base case: get result directly from the accumulators

# Let's take another example

## Computation of a string's length

```
len(str):
  if empty(s) then 0
               else  1+len(cdr(s))
```

$\lambda$(str,acc):

0. (works because addition is commutative and associative)
1. Create a $\lambda$ function doing the recursion, adding one accumulator per operation

# Let's take another example

## Computation of a string's length

```
len(str):
  if empty(s) then 0
               else  1+len(cdr(s))
```

len'(str) = $\lambda$(str,0)

$\lambda$(str,acc):

0. (works because addition is commutative and associative)
1. Create a $\lambda$ function doing the recursion, adding one accumulator per operation
2. Main function: calls the lambda function and initializes the parameters

# Let's take another example

## Computation of a string's length

```
len(str):
  if empty(s) then 0
              else  1+len(cdr(s))
```

```
len'(str) = λ(str,0)

λ(str,acc):
  if empty(str)
              else  λ(cdr(str), acc+1)
```

0. (works because addition is commutative and associative)
1. Create a $\lambda$ function doing the recursion, adding one accumulator per operation
2. Main function: calls the lambda function and initializes the parameters
3. Body of the $\lambda$ function:
   - General treatment: as before, but do intermediate ops into the accumulator(s)

# Let's take another example

## Computation of a string's length

```
len(str):
  if empty(s) then 0
               else  1+len(cdr(s))
```

```
len'(str) = λ(str,0)

λ(str,acc):
  if empty(str) then acc
                 else  λ(cdr(str), acc+1)
```

0. (works because addition is commutative and associative)
1. Create a $\lambda$ function doing the recursion, adding one accumulator per operation
2. Main function: calls the lambda function and initializes the parameters
3. Body of the $\lambda$ function:
   - General treatment: as before, but do intermediate ops into the accumulator(s)
   - Base case: get result directly from the accumulator(s)

# Closing the loop: Non-recursive form of Factorial

FACT(n):
 **if** n $= 0$ **then** $r \leftarrow 1$
  **else** $r \leftarrow n \times fact(n-1)$

FACT'(n): $\lambda(n, 1)$
$\lambda(n, acc)$ :
 **if** n $= 0$ **then** $r \leftarrow acc$
  **else** $r \leftarrow \lambda(n-1, acc \times n)$

- This function uses Tail Recursion

$\leadsto$ We can turn the helper into non-recursion with the method seen before

# Closing the loop: Non-recursive form of Factorial

```
FACT(n):
  if n = 0 then r ← 1
           else r ← n × fact(n − 1)
```

```
FACT'(n): λ(n, 1)
λ(n, acc) :
  if n = 0 then r ← acc
           else r ← λ(n − 1, acc × n)
```

- This function uses Tail Recursion

$\leadsto$ We can turn the helper into non-recursion with the method seen before

```
λ'(n, acc) :
  td ← n; a ← acc
  until td = 0 do
    a ← a × td        // beware of the
    td ← td − 1       // updates' order
  end
  return a
```

# Closing the loop: Non-recursive form of Factorial

```
FACT(n):
  if n = 0 then r ← 1
           else r ← n × fact(n − 1)
```

```
FACT'(n): λ(n, 1)
λ(n, acc) :
  if n = 0 then r ← acc
           else r ← λ(n − 1, acc × n)
```

- This function uses Tail Recursion

$\rightsquigarrow$ We can turn the helper into non-recursion with the method seen before

- Then, we combine everything

```
λ'(n, acc) :
  td ← n; a ← acc
  until td = 0 do
    a ← a × td        // beware of the
    td ← td − 1       // updates' order
  end
  return a
```

```
FACT''(n):
  td ← n; a ← 1
  until td = 0 do
    a ← a × td
    td ← td − 1
  end
  return a
```

- These two transformations are simple, automatic and neat. . .

# Closing the loop: Non-recursive form of Factorial

FACT(n):
  **if** n = 0 **then** $r \leftarrow 1$
      **else** $r \leftarrow n \times fact(n-1)$

FACT'(n): $\lambda(n, 1)$
$\lambda(n, acc)$ :
  **if** n = 0 **then** $r \leftarrow acc$
      **else** $r \leftarrow \lambda(n-1, acc \times n)$

- This function uses Tail Recursion

$\rightsquigarrow$ We can turn the helper into non-recursion with the method seen before

- Then, we combine everything

$\lambda'(n, acc)$ :
  $td \leftarrow n$; $a \leftarrow acc$
  **until** $td = 0$ **do**
    $a \leftarrow a \times td$  // beware of the
    $td \leftarrow td - 1$  // updates' order
  **end**
  return $a$

FACT"(n):
  $td \leftarrow n$; $a \leftarrow 1$
  **until** $td = 0$ **do**
    $a \leftarrow a \times td$
    $td \leftarrow td - 1$
  **end**
  return $a$

- These two transformations are simple, automatic and neat...
- ...when applicable!!! ☹  If not, let's get angry and mean!

# Generic Algorithm Using a Stack

## Idea

- Processors are sequential and execute any recursive function
  $\Rightarrow$ *Always possible to express without recursion*
- Principle: simulating the function stack of processors
  **By using a stack explicitly**

# Generic Algorithm Using a Stack

## Idea

- Processors are sequential and execute any recursive function
  $\Rightarrow$ *Always possible to express without recursion*
- Principle: simulating the function stack of processors
  **By using a stack explicitly**

## Example with only one recursive call

**if** cond(x) **then** $r \leftarrow g(x)$
**else** $\textsc{T}$(x); $r \leftarrow G(x, f(x_{int}))$

# Generic Algorithm Using a Stack

## Idea

- Processors are sequential and execute any recursive function
  $\Rightarrow$ *Always possible to express without recursion*
- Principle: simulating the function stack of processors
  **By using a stack explicitly**

## Example with only one recursive call

$$\textbf{if } \text{cond}(x) \textbf{ then } r \leftarrow g(x) \quad \textbf{else } \text{T}(x);\ r \leftarrow G(x, f(x_{int}))$$

```
p ← emptyStack
a ← x (* a: locale variable *)
(* pushing on stack (descent) *)
until cond(a) do
  push(p, a)
  a ← h(a)
end
r ← g(a) (* Base Case *)
(* poping from stack (climb up) *)
until stackIsEmpty(p) do
  a ← top(p); pop(p); T(a)
  r ← G(a, r)
end
```

# Generic Algorithm Using a Stack

## Idea

- Processors are sequential and execute any recursive function
  $\Rightarrow$ *Always possible to express without recursion*
- Principle: simulating the function stack of processors
  **By using a stack explicitly**

## Example with only one recursive call

$$\textbf{if } \mathrm{cond}(x) \textbf{ then } r \leftarrow g(x) \quad \textbf{else } \mathrm{T}(x);\ r \leftarrow G(x, f(x_{int}))$$

```
p ← emptyStack
a ← x (* a: locale variable *)
(* pushing on stack (descent) *)
until cond(a) do
  push(p, a)
  a ← h(a)
end
r ← g(a) (* Base Case *)
(* poping from stack (climb up) *)
until stackIsEmpty(p) do
  a ← top(p); pop(p); T(a)
  r ← G(a, r)
end
```

Remark:
If $h()$ is invertible, no need for a stack: parameter reconstructed by $h^{-1}()$

Stopping Condition = counting calls

# Non-Recursive Form of Hanoï Towers (1/2)

```
HANOI(n,a,b,c): (a to b, with c as disposal)
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

One should mimic processor behavior wrt stacking

- H(4,a,b,c) = H(3,a,c,b)+D(a,b)+H(3,c,b,a)

We get:

H(4,a,b,c) = $\underbrace{\qquad\qquad\qquad}_{\text{H(3,a,c,b)}}$ +D(a,b)+H(3,c,b,a)

# Non-Recursive Form of Hanoï Towers (1/2)

```
HANOI(n,a,b,c): (a to b, with c as disposal)
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

## One should mimic processor behavior wrt stacking

- H(4,a,b,c) = H(3,a,c,b)+D(a,b)+H(3,c,b,a)
- Compute first unknown term: H(3,a,c,b) = H(2,a,b,c)+D(a,c)+H(2,b,c,a)

## We get:

$$H(4,a,b,c) = \underbrace{\underbrace{\qquad\qquad}_{H(2,a,b,c)} + D(a,c) + H(2,b,c,a)}_{H(3,a,c,b)} + D(a,b) + H(3,c,b,a)$$

# Non-Recursive Form of Hanoï Towers (1/2)

```
HANOI(n,a,b,c): (a to b, with c as disposal)
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

## One should mimic processor behavior wrt stacking

- H(4,a,b,c) = H(3,a,c,b)+D(a,b)+H(3,c,b,a)
- Compute first unknown term: H(3,a,c,b) = H(2,a,b,c)+D(a,c)+H(2,b,c,a)
- Compute first unknown term: H(2,a,b,c) = H(1,a,c,b)+D(a,b)+H(1,c,b,a)

## We get:

$$H(4,a,b,c) = \underbrace{\underbrace{\quad + D(a,b) + \quad}_{H(2,a,b,c)} + D(a,c) + H(2,b,c,a)}_{H(3,a,c,b)} + D(a,b) + H(3,c,b,a)$$

# Non-Recursive Form of Hanoï Towers (1/2)

```
HANOI(n,a,b,c): (a to b, with c as disposal)
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

## One should mimic processor behavior wrt stacking

- H(4,a,b,c) = H(3,a,c,b)+D(a,b)+H(3,c,b,a)
- Compute first unknown term: H(3,a,c,b) = H(2,a,b,c)+D(a,c)+H(2,b,c,a)
- Compute first unknown term: H(2,a,b,c) = H(1,a,c,b)+D(a,b)+H(1,c,b,a)
- Compute first unknown term: H(1,a,c,b) = D(a,c)

## We get:

$$\text{H(4,a,b,c)} = \underbrace{\underbrace{\text{D(a,c)+D(a,b)+}\qquad}_{\text{H(2,a,b,c)}}\text{+D(a,c)+H(2,b,c,a)}}_{\text{H(3,a,c,b)}}\text{+D(a,b)+H(3,c,b,a)}$$

# Non-Recursive Form of Hanoï Towers (1/2)

```
HANOI(n,a,b,c): (a to b, with c as disposal)
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

## One should mimic processor behavior wrt stacking

- H(4,a,b,c) = H(3,a,c,b)+D(a,b)+H(3,c,b,a)
- Compute first unknown term: H(3,a,c,b) = H(2,a,b,c)+D(a,c)+H(2,b,c,a)
- Compute first unknown term: H(2,a,b,c) = H(1,a,c,b)+D(a,b)+H(1,c,b,a)
- Compute first unknown term: H(1,a,c,b) = D(a,c)
- Take on something casted aside: H(1,c,b,a) = D(c,b)

## We get:

$$H(4,a,b,c) = \underbrace{\underbrace{D(a,c)+D(a,b)+D(c,b)}_{H(2,a,b,c)}+D(a,c)+H(2,b,c,a)}_{H(3,a,c,b)}+D(a,b)+H(3,c,b,a)$$

# Non-Recursive Form of Hanoï Towers (1/2)

```
HANOI(n,a,b,c): (a to b, with c as disposal)
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

## One should mimic processor behavior wrt stacking

- H(4,a,b,c) = H(3,a,c,b)+D(a,b)+H(3,c,b,a)
- Compute first unknown term: H(3,a,c,b) = H(2,a,b,c)+D(a,c)+H(2,b,c,a)
- Compute first unknown term: H(2,a,b,c) = H(1,a,c,b)+D(a,b)+H(1,c,b,a)
- Compute first unknown term: H(1,a,c,b) = D(a,c)
- Take on something casted aside: H(1,c,b,a) = D(c,b)
- and so on until everything casted aside is done (until stack is empty)

## We get:

$$H(4,a,b,c) = \underbrace{\underbrace{D(a,c)+D(a,b)+D(c,b)}_{H(2,a,b,c)}+D(a,c)+H(2,b,c,a)}_{H(3,a,c,b)}+D(a,b)+H(3,c,b,a)$$

# Non-Recursive Form of Hanoï Towers (2/2)

```
hanoi_derec(n, A, B) :
  push (n, A, B, 1) on stack
  while (stack non empty vide)
    (n, A, B, CallKind) ← pop()
    if (n > 0)
      if (CallKind == 1)
        push (n, A, B, 2) on stack (* Cast something aside for later *)
        push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
        move(A, B)
        push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

# Non-Recursive Form of Hanoï Towers (2/2)

```
hanoi_derec(n, A, B) :
  push (n, A, B, 1) on stack
  while (stack non empty vide)
    (n, A, B, CallKind) ← pop()
    if (n > 0)
      if (CallKind == 1)
        push (n, A, B, 2) on stack (* Cast something aside for later *)
        push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
        move(A, B)
        push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

4AB1

Step 1 Step 2 Step 3 Step 4 Step 5 Step 6 Step 8 Step 9 Step 11 Step 13

hanoi(4,a,b)=...

# Non-Recursive Form of Hanoï Towers (2/2)

hanoi_derec(n, A, B) :

```
push (n, A, B, 1) on stack
while (stack non empty vide)
   (n, A, B, CallKind) ← pop()
   if (n > 0)
      if (CallKind == 1)
         push (n, A, B, 2) on stack (* Cast something aside for later *)
         push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
         move(A, B)
         push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

| Step 1 | Step 2 | Step 3 | Step 4 | Step 5 | Step 6 | Step 8 | Step 9 | Step 11 | Step 13 |
|---|---|---|---|---|---|---|---|---|---|
| | 3AC1 | | | | | | | | |
| 4AB1 | 4AB2 | | | | | | | | |

hanoi(4,a,b)=. . .

# Non-Recursive Form of Hanoï Towers (2/2)

```
hanoi_derec(n, A, B) :
  push (n, A, B, 1) on stack
  while (stack non empty vide)
    (n, A, B, CallKind) ← pop()
    if (n > 0)
      if (CallKind == 1)
        push (n, A, B, 2) on stack (* Cast something aside for later *)
        push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
        move(A, B)
        push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 2AB1 | | | | | | | |
| | 3AC1 | 3AC2 | | | | | | | |
| 4AB1 | 4AB2 | 4AB2 | | | | | | | |
| Step 1 | Step 2 | Step 3 | Step 4 | Step 5 | Step 6 | Step 8 | Step 9 | Step 11 | Step 13 |

hanoi(4,a,b)=...

# Non-Recursive Form of Hanoï Towers (2/2)

```
hanoi_derec(n, A, B) :
  push (n, A, B, 1) on stack
  while (stack non empty vide)
    (n, A, B, CallKind) ← pop()
    if (n > 0)
      if (CallKind == 1)
        push (n, A, B, 2) on stack (* Cast something aside for later *)
        push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
        move(A, B)
        push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

| Step 1 | Step 2 | Step 3 | Step 4 | Step 5 | Step 6 | Step 8 | Step 9 | Step 11 | Step 13 |
|---|---|---|---|---|---|---|---|---|---|
| | | | 1AC1 | | | | | | |
| | | 2AB1 | 2AB2 | | | | | | |
| | 3AC1 | 3AC2 | 3AC2 | | | | | | |
| 4AB1 | 4AB2 | 4AB2 | 4AB2 | | | | | | |

hanoi(4,a,b)=. . .

# Non-Recursive Form of Hanoï Towers (2/2)

hanoi_derec(n, A, B) :

```
push (n, A, B, 1) on stack
while (stack non empty vide)
  (n, A, B, CallKind) ← pop()
  if (n > 0)
    if (CallKind == 1)
      push (n, A, B, 2) on stack (* Cast something aside for later *)
      push (n-1, A, C, 1) (* Compute first unknown soon *)
    else /* ie, CallKind == 2 */
      move(A, B)
      push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

| | | | | 0AB1 | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | 1AC1 | 1AC2 | | | | | |
| | | 2AB1 | 2AB2 | 2AB2 | | | | | |
| | 3AC1 | 3AC2 | 3AC2 | 3AC2 | | | | | |
| 4AB1 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | | | | | |
| Step 1 | Step 2 | Step 3 | Step 4 | Step 5 | Step 6 | Step 8 | Step 9 | Step 11 | Step 13 |

hanoi(4,a,b)=. . .

# Non-Recursive Form of Hanoï Towers (2/2)

```
hanoi_derec(n, A, B) :
  push (n, A, B, 1) on stack
  while (stack non empty vide)
    (n, A, B, CallKind) ← pop()
    if (n > 0)
      if (CallKind == 1)
        push (n, A, B, 2) on stack (* Cast something aside for later *)
        push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
        move(A, B)
        push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

| Step 1 | Step 2 | Step 3 | Step 4 | Step 5 | Step 6 | Step 8 | Step 9 | Step 11 | Step 13 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | 0AB1 | | | | | |
| | | | 1AC1 | 1AC2 | 0BC1 | | | | |
| | | 2AB1 | 2AB2 | 2AB2 | 2AB2 | | | | |
| | 3AC1 | 3AC2 | 3AC2 | 3AC2 | 3AC2 | | | | |
| 4AB1 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | | | | |

hanoi(4,a,b)=D(ac)+. . .

# Non-Recursive Form of Hanoï Towers (2/2)

```
hanoi_derec(n, A, B) :
  push (n, A, B, 1) on stack
  while (stack non empty vide)
    (n, A, B, CallKind) ← pop()
    if (n > 0)
      if (CallKind == 1)
        push (n, A, B, 2) on stack (* Cast something aside for later *)
        push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
        move(A, B)
        push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

| Step 1 | Step 2 | Step 3 | Step 4 | Step 5 | Step 6 | Step 8 | Step 9 | Step 11 | Step 13 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | 0AB1 | | | | | |
| | | | 1AC1 | 1AC2 | 0BC1 | | | | |
| | | 2AB1 | 2AB2 | 2AB2 | 2AB2 | 1CB1 | | | |
| | 3AC1 | 3AC2 | 3AC2 | 3AC2 | 3AC2 | 3AC2 | | | |
| 4AB1 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | | | |

hanoi(4,a,b)=D(ac)+D(ab)+...

# Non-Recursive Form of Hanoï Towers (2/2)

```
hanoi_derec(n, A, B) :
  push (n, A, B, 1) on stack
  while (stack non empty vide)
    (n, A, B, CallKind) ← pop()
    if (n > 0)
      if (CallKind == 1)
        push (n, A, B, 2) on stack (* Cast something aside for later *)
        push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
        move(A, B)
        push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

| Step 1 | Step 2 | Step 3 | Step 4 | Step 5 | Step 6 | Step 8 | Step 9 | Step 11 | Step 13 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | 0AB1 | | | | | |
| | | | 1AC1 | 1AC2 | 0BC1 | | 0AB1 | | |
| | | 2AB1 | 2AB2 | 2AB2 | 2AB2 | 1CB1 | 1CB2 | | |
| | 3AC1 | 3AC2 | 3AC2 | 3AC2 | 3AC2 | 3AC2 | 3AC2 | | |
| 4AB1 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | | |

hanoi(4,a,b)=D(ac)+D(ab)+...

# Non-Recursive Form of Hanoï Towers (2/2)

```
hanoi_derec(n, A, B) :
  push (n, A, B, 1) on stack
  while (stack non empty vide)
    (n, A, B, CallKind) ← pop()
    if (n > 0)
      if (CallKind == 1)
        push (n, A, B, 2) on stack (* Cast something aside for later *)
        push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
        move(A, B)
        push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

| Step 1 | Step 2 | Step 3 | Step 4 | Step 5 | Step 6 | Step 8 | Step 9 | Step 11 | Step 13 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | 0AB1 | | | | | |
| | | | 1AC1 | 1AC2 | 0BC1 | | 0AB1 | | |
| | | 2AB1 | 2AB2 | 2AB2 | 2AB2 | 1CB1 | 1CB2 | 0AB1 | |
| | 3AC1 | 3AC2 | 3AC2 | 3AC2 | 3AC2 | 3AC2 | 3AC2 | 3AC2 | |
| 4AB1 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | |

hanoi(4,a,b)=D(ac)+D(ab)+D(cb)+...

# Non-Recursive Form of Hanoï Towers (2/2)

```
hanoi_derec(n, A, B) :
  push (n, A, B, 1) on stack
  while (stack non empty vide)
    (n, A, B, CallKind) ← pop()
    if (n > 0)
      if (CallKind == 1)
        push (n, A, B, 2) on stack (* Cast something aside for later *)
        push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
        move(A, B)
        push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

| Step 1 | Step 2 | Step 3 | Step 4 | Step 5 | Step 6 | Step 8 | Step 9 | Step 11 | Step 13 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | 0AB1 | | | | | |
| | | | 1AC1 | 1AC2 | 0BC1 | | 0AB1 | | |
| | | 2AB1 | 2AB2 | 2AB2 | 2AB2 | 1CB1 | 1CB2 | 0AB1 | |
| | 3AC1 | 3AC2 | 3AC2 | 3AC2 | 3AC2 | 3AC2 | 3AC2 | 3AC2 | 2BC1 |
| 4AB1 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 |

hanoi(4,a,b)=D(ac)+D(ab)+D(cb)+D(ac)+...

# Non-Recursive Form of Hanoï Towers (2/2)

```
hanoi_derec(n, A, B) :
  push (n, A, B, 1) on stack
  while (stack non empty vide)
    (n, A, B, CallKind) ← pop()
    if (n > 0)
      if (CallKind == 1)
        push (n, A, B, 2) on stack (* Cast something aside for later *)
        push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
        move(A, B)
        push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

| Step 1 | Step 2 | Step 3 | Step 4 | Step 5 | Step 6 | Step 8 | Step 9 | Step 11 | Step 13 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | 0AB1 | | | | | |
| | | | 1AC1 | 1AC2 | 0BC1 | | 0AB1 | | |
| | | 2AB1 | 2AB2 | 2AB2 | 2AB2 | 1CB1 | 1CB2 | 0AB1 | |
| | 3AC1 | 3AC2 | 3AC2 | 3AC2 | 3AC2 | 3AC2 | 3AC2 | 3AC2 | 2BC1 |
| 4AB1 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 | 4AB2 |

hanoi(4,a,b)=D(ac)+D(ab)+D(cb)+D(ac)+...

Rq: simpler iterative algorithms exist (they are not automatic transformations)

# 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
- Avoiding Recursion
  - Non-Recursive Form of Tail Recursion
  - Transformation to Tail Recursion
  - Generic Algorithm Using a Stack
- **Back-tracking**
- Conclusion

# Combinatorial Search and Optimization

Large class of Problems with similar algorithmic approach

- Solutions are really numerous; A set of constraints make some solution invalids
- Combinatorial Search $\leadsto$ look for any valid solution
  Combinatorial Optimization $\leadsto$ look for the solution maximizing a function

## Examples

- Open the lock: Find the right 4-digits combination out of 10000
- Knapsac: Ali-Baba searches object set fitting in bag maximizing the value
- Minimum Spanning Tree of a given graph
- Traveling Salesman: visit $n$ cities in order minimizing the total distance

## First Resolution Approach: Exhaustive Search

- Study *every* solutions
  $\leadsto$ Test all lock combinations
  $\leadsto$ Enumerate all possible knapsack contents + get max value
- This often reveals to be exponential and thus infeasible

# Better Approach?

## Guessing the right number can become difficult that way

- 0001 $\leadsto$ *no*; 0002 $\leadsto$ *no*; 0003 $\leadsto$ *no*; 0004 $\leadsto$ *no*; 0005 $\leadsto$ *no*; Booooring
- Let's more information: length of correct suffix instead of yes/no answers
  0001 $\leadsto$ 0; 0002 $\leadsto$ 0; 0003 $\leadsto$ 1; 0024 $\leadsto$ 2; 0424 $\leadsto$ 3; 5424 $\leadsto$ 4, *bingo*

## This is Backtracking

- Tentative choices + cut branches leading to invalid solutions (backtrack)
- Restrict study to valid solutions only $\leadsto$ if bag is full, don't stuff something else
- Also factorize computations $\leadsto$ only sum up once the N first objects' value

# Better Approach?

## Guessing the right number can become difficult that way

- 0001 $\rightsquigarrow$ *no*; 0002 $\rightsquigarrow$ *no*; 0003 $\rightsquigarrow$ *no*; 0004 $\rightsquigarrow$ *no*; 0005 $\rightsquigarrow$ *no*; Booooring
- Let's more information: length of correct suffix instead of yes/no answers
  0001 $\rightsquigarrow$ 0; 0002 $\rightsquigarrow$ 0; 0003 $\rightsquigarrow$ 1; 0024 $\rightsquigarrow$ 2; 0424 $\rightsquigarrow$ 3; 5424 $\rightsquigarrow$ 4, *bingo*

## This leads to a much more efficient algorithm:

- Guess each position by testing every digit in that pos until response increases

## This is Backtracking

- Tentative choices + cut branches leading to invalid solutions (backtrack)
- Restrict study to valid solutions only $\rightsquigarrow$ if bag is full, don't stuff something else
- Also factorize computations $\rightsquigarrow$ only sum up once the N first objects' value

# Better Approach?

## Guessing the right number can become difficult that way

- 0001 $\leadsto$ *no*; 0002 $\leadsto$ *no*; 0003 $\leadsto$ *no*; 0004 $\leadsto$ *no*; 0005 $\leadsto$ *no*; Booooring
- Let's more information: length of correct suffix instead of yes/no answers
  0001 $\leadsto$ 0; 0002 $\leadsto$ 0; 0003 $\leadsto$ 1; 0024 $\leadsto$ 2; 0424 $\leadsto$ 3; 5424 $\leadsto$ 4, *bingo*

## This leads to a much more efficient algorithm:

- Guess each position by testing every digit in that pos until response increases
- That's even easy to write by mixing recursion with a for loop:

```
search(current,pos,len): // initial values: search({0,0,0,0}, 0, 0)
  for n ∈ [0;9] do
    put n into current at position pos
    if try(current) > len then search(current,pos+1, try(current))
                          else // no luck. Let's test the next value of n
```

## This is Backtracking

- Tentative choices + cut branches leading to invalid solutions (backtrack)
- Restrict study to valid solutions only $\leadsto$ if bag is full, don't stuff something else
- Also factorize computations $\leadsto$ only sum up once the N first objects' value

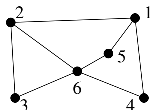
# Back-tracking

## Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

## First example: Independent Sets

- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack



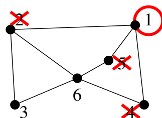
# Back-tracking

## Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

## First example: Independent Sets

- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack




- $\{1\}$

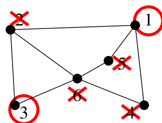
# Back-tracking

## Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

## First example: Independent Sets

- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack




- $\{1\}$, $\{1, 3\}$

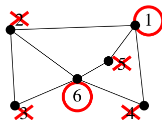
# Back-tracking

## Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

## First example: Independent Sets

- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack




- $\{1\}$, $\{1,3\}$. Stuck. Remove 3. $\{1,6\}$.

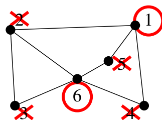
# Back-tracking

## Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

## First example: Independent Sets

- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack




- $\{1\}$, $\{1,3\}$. Stuck. Remove 3. $\{1,6\}$. Stuck.
  Removing 6 is not enough, remove everything.

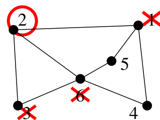
# Back-tracking

## Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

## First example: Independent Sets

- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack




- $\{1\}$, $\{1,3\}$. Stuck. Remove 3. $\{1,6\}$. Stuck.
  Removing 6 is not enough, remove everything.
- $\{2\}$

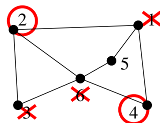
# Back-tracking

## Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

## First example: Independent Sets

- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack




- $\{1\}$, $\{1,3\}$. Stuck. Remove 3. $\{1,6\}$. Stuck.
  Removing 6 is not enough, remove everything.
- $\{2\}$, $\{2,4\}$

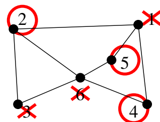
# Back-tracking

## Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

## First example: Independent Sets

- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack




- $\{1\}$, $\{1,3\}$. Stuck. Remove 3. $\{1,6\}$. Stuck.
  Removing 6 is not enough, remove everything.
- $\{2\}$, $\{2,4\}$, $\{2,4,5\}$

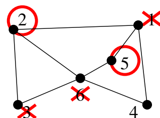
# Back-tracking

## Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

## First example: Independent Sets

- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack




- $\{1\}$, $\{1,3\}$. Stuck. Remove 3. $\{1,6\}$. Stuck.
  Removing 6 is not enough, remove everything.
- $\{2\}$, $\{2,4\}$, $\{2,4,5\}$ (Stuck; remove 5 then 4) $\{2,5\}$

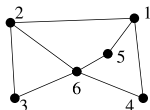
# Back-tracking

## Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

## First example: Independent Sets

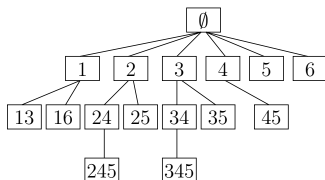
- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack




- $\{1\}$, $\{1,3\}$. Stuck. Remove 3. $\{1,6\}$. Stuck.
  Removing 6 is not enough, remove everything.
- $\{2\}$, $\{2,4\}$, $\{2,4,5\}$ (Stuck; remove 5 then 4) $\{2,5\}$
- $\{3\}$, $\{3,4\}$, $\{3,4,5\}$, $\{3,5\}$; $\{4\}$, $\{4,5\}$; $\{5\}$, $\{6\}$

# Algorithm Computation Time

## Solution Tree of this Algorithm




- Traverse every nodes (without building it explicitly)
- Amount of algorithm steps = amount of solutions
- Let $n$ be amount of nodes

## Amount of solutions for a given graph?

- Empty Graph (no edge) $\leadsto I_n = 2^n$ independent sets
- Full Graph (every edges) $\leadsto I_n = n + 1$ independent sets
- On average $\leadsto I_n = \sum_{k=0}^{n} \binom{k}{n} 2^{-k(k-1)/2}$

| $n$ | 2 | 3 | 4 | 5 | 10 | 15 | 20 | 30 | 40 |
|---|---|---|---|---|---|---|---|---|---|
| $I_n$ | 3,5 | 5,6 | 8,5 | 12,3 | 52 | 149,8 | 350,6 | 1342,5 | 3862,9 |
| $2^n$ | 4 | 8 | 16 | 32 | 1024 | 32768 | 1048576 | 1073741824 | 1099511627776 |

- Backtracking algorithm traverses $I_n$ nodes on average
- An exhaustive search traverses $2^n$ nodes

# Other example: $n$ queens puzzle

Goal:

- Put $n$ queens on a $n \times n$ board so than none of them can capture any other

Algorithm:

- Put a queen on first line
  There is $n$ choices, any implying constraints for the following
- Recursive call for next line

# Other example: $n$ queens puzzle

Goal:

- Put $n$ queens on a $n \times n$ board so than none of them can capture any other

Algorithm:

- Put a queen on first line
  There is $n$ choices, any implying constraints for the following
- Recursive call for next line

Pseudo-code `put_queens(int line, board)`

If *line* $>$ *line_count*, return `board` (success)

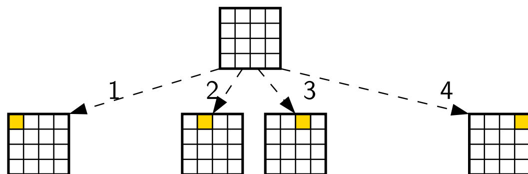
$\forall$ *cell* $\in$ *line*,

- Put a queen at position *cell* $\times$ *line* of `board`
- If conflict, then return (stopping descent – failure)
- (else) call `put_queens(ligne+1, board` $\cap$ $\{cell, line\}$)

$\Rightarrow$ Recursive Call within a Loop

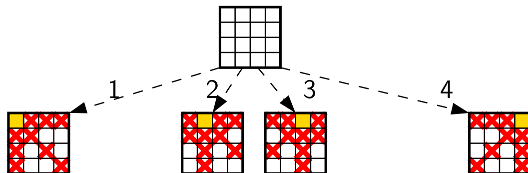
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions



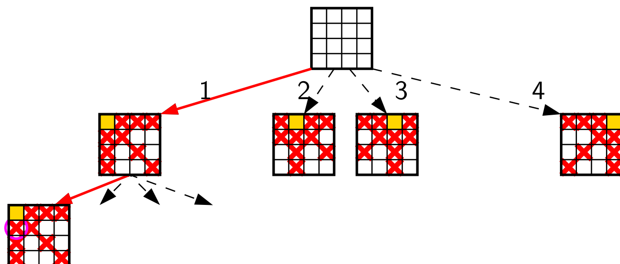
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following



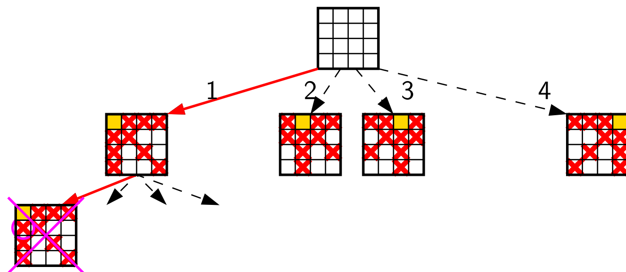
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent



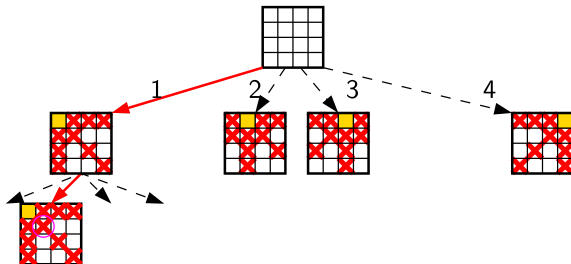
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back



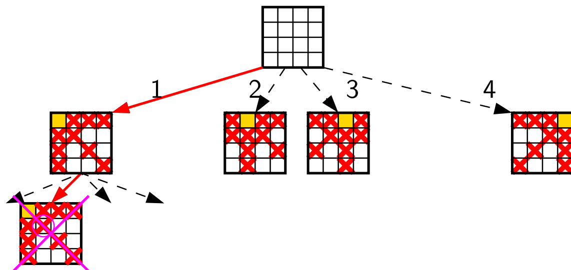
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



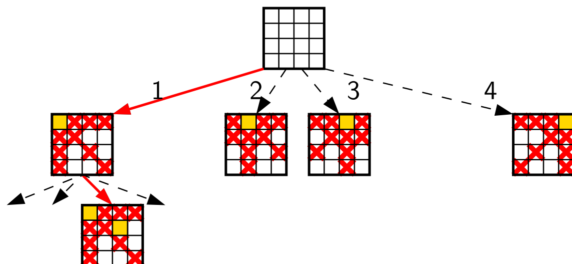
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



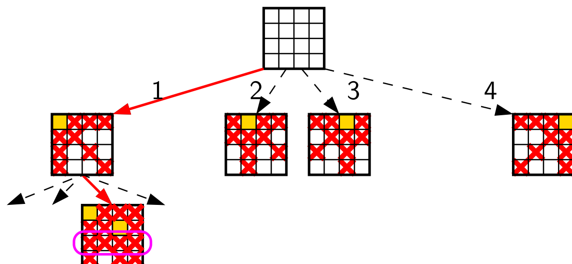
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



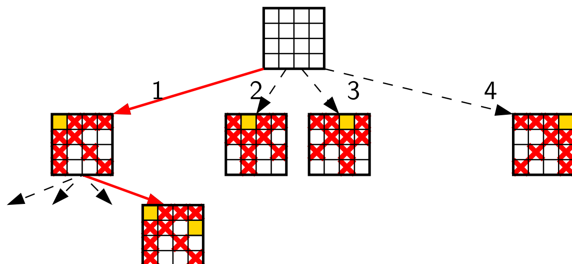
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



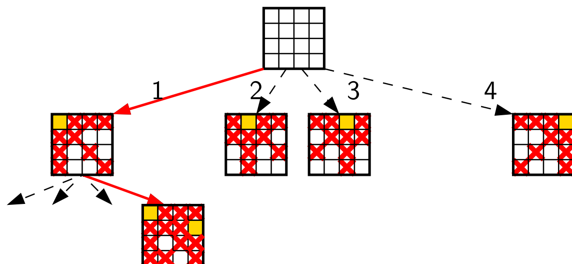
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



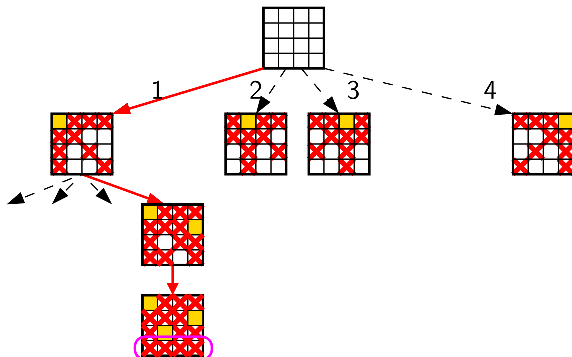
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



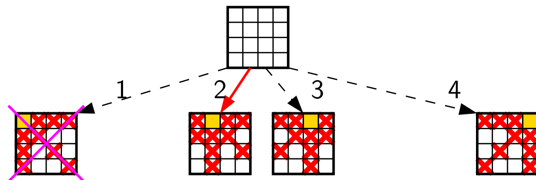
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



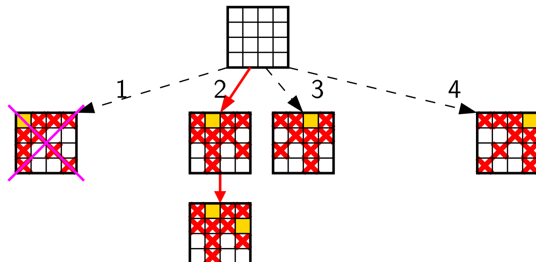
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



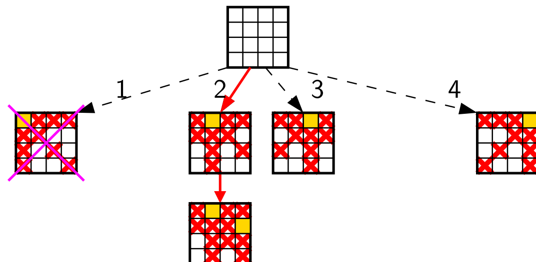
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



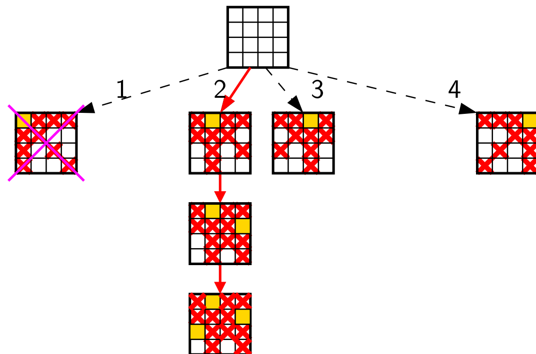
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



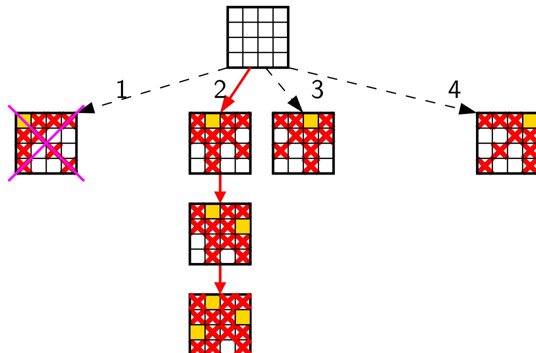
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



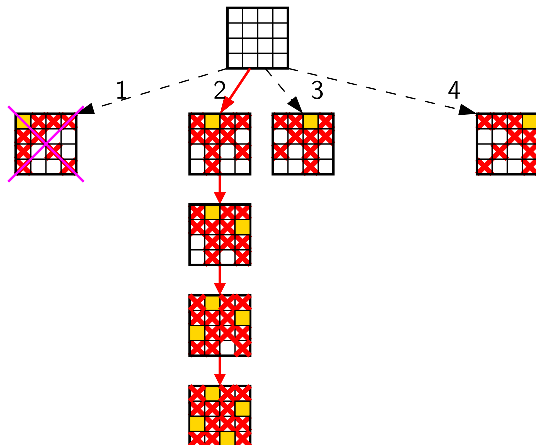
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



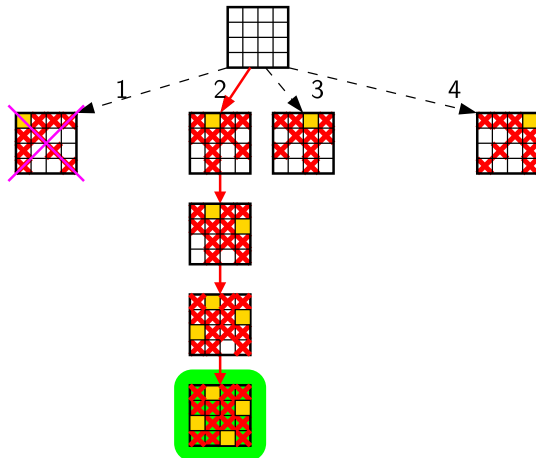
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



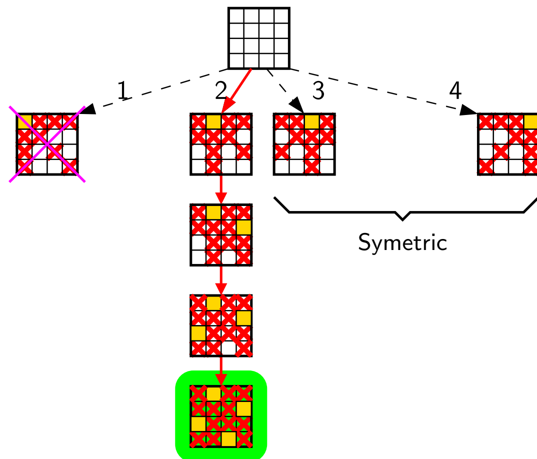
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)
- Until we find a solution (or not)



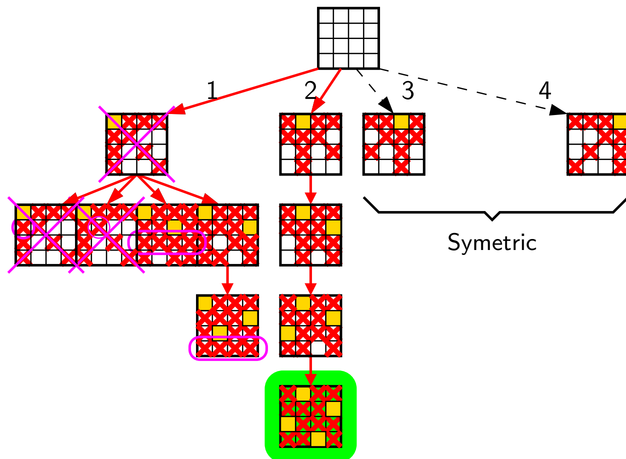
# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)
- Until we find a solution (or not)



# Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)
- Until we find a solution (or not)



# Java implementation of n queens puzzle

```
boolean Solution(boolean board[][], int line) {
  if (line >= board.length) // Base Case
     return true;

  for (int col = 0; col < board.length; col++) {  // loop on possibilities
     if (validPlacement(board, line, col)) {
       putQueen(board, line, col);
       if (Solution(plateau, line + 1)) // Recursive Call
           return true; // Let solution climb back
       removeQueen(board, line, col);
     }
  }
  return false;
}
```

# Some Principles on Backtracking

- Study "depth first" of solution tree
- On backtracking, restore state as before last choice
  Trivial here (parameters copied on recursive call), harder in iterative
- Strategy on branch ordering can improve things
- Progressive Construction of boolean function
- If function returns false, there is no solution

# Some Principles on Backtracking

- Study "depth first" of solution tree
- On backtracking, restore state as before last choice
  Trivial here (parameters copied on recursive call), harder in iterative
- Strategy on branch ordering can improve things
- Progressive Construction of boolean function
- If function returns false, there is no solution

- Probable Combinatorial Explosion ($4^4$ boards)
  $\Rightarrow$ Need for heuristics to limit amount of tries

# Conclusion on Recursion

## Essential Tool for Algorithms

- Recursion in Computer Science, induction in Mathematics
- Recursive Algorithms are frequent because easier to understand . . .
  (and thus easier to maintain)
  . . . but maybe slightly more difficult to write (that's a practice to get)
- Recursive programs maybe slightly less efficients. . .
  . . . but always possible to transform a code to non-recursive form
  (and compilers do it)
- Classical Functions: Factorial, gcd, Fibonacci, Ackerman, Hanoï, Syracuse, . . .
- Sorting Functions: MergeSort and QuickSort are amongst the most used
  (because efficient)
- BackTracking: exhaustive search in space of *valid* solutions
- Data Structure module: several recursive datatypes with associated algorithms
- Recursion is the root of computation since it trades description for time.
  – "Epigrams in Programming", by Alan J. Perlis of Yale University.

(this ends the third lecture)

# Fourth Chapter

## Correction of Software Systems


- Introduction
- Specification
- Hoare Logic
- Proving Recursive Functions
- Conclusion

# Sorting Algorithm Performance Discussion

We have shown that

| | Amount of comparisons | | | Memory Complexity |
|---|---|---|---|---|
| | Best Case | Average Case | Worst Case | |
| Selection Sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $\Theta(1)$ |
| Insertion Sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $\Theta(1)$ |
| Bubble Sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $\Theta(1)$ |
| Quick Sort | $O(n\log(n))$ | $O(n\log(n))$ | $O(n^2)$ | $O(\log(n))$ |
| Merge Sort | $O(n\log(n))$ | $O(n\log(n))$ | $O(n\log(n))$ | $O(n)$ |

- Very accurate knowledge on achieved performance

# Sorting Algorithm Performance Discussion

We have shown that

| | Amount of comparisons | | | Memory Complexity |
|---|---|---|---|---|
| | Best Case | Average Case | Worst Case | |
| Selection Sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $\Theta(1)$ |
| Insertion Sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $\Theta(1)$ |
| Bubble Sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $\Theta(1)$ |
| Quick Sort | $O(n\log(n))$ | $O(n\log(n))$ | $O(n^2)$ | $O(\log(n))$ |
| Merge Sort | $O(n\log(n))$ | $O(n\log(n))$ | $O(n\log(n))$ | $O(n)$ |

- Very accurate knowledge on achieved performance

But wait a second...

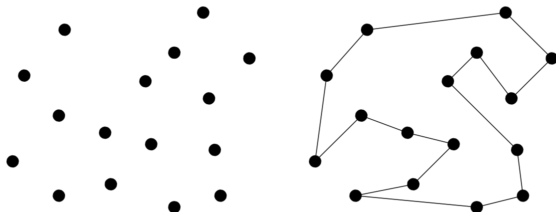
How do you know this code actually sorts the array?

- Because you see it, it's obvious (yeah, right...)
- Because the teacher / a friend says so
- Because it's written in a book / on the Internet
- Because you tested it

# Because it's obvious you said?

## Let's consider the following problem

- You have a robot arm equipped with a soldering iron (for example)
- You have several positions were the arm should do its soldering job
- You must decide the order in which the arm visit the positions
- You want to minimize the time (ie travel distance) it takes to visit all positions



# Nearest Neighbor Tour

## Here is a very popular algorithm to that problem

- Pick and visit an initial point $p_0$, and let $i = 0$
- While there are still unvisited points
  - $i = i + 1$
  - let $p_i$ be the closest unvisited point to $p_{i-1}$
  - Visit $p_i$
- Return to $p_0$ from $p_i$

## Advantage of that algorithm

- It is simple to understand and implement; It's very efficient: $O(n)$

# Nearest Neighbor Tour

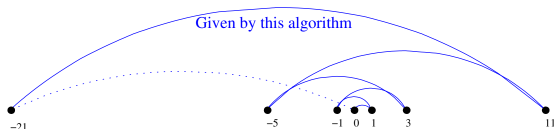
## Here is a very popular algorithm to that problem

- Pick and visit an initial point $p_0$, and let $i = 0$
- While there are still unvisited points
  - $i = i + 1$
  - let $p_i$ be the closest unvisited point to $p_{i-1}$
  - Visit $p_i$
- Return to $p_0$ from $p_i$

## Advantage of that algorithm

- It is simple to understand and implement; It's very efficient: $O(n)$

## But it is not correct!



# Nearest Neighbor Tour

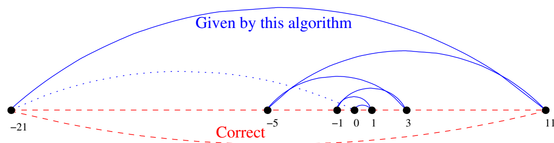
## Here is a very popular algorithm to that problem

- Pick and visit an initial point $p_0$, and let $i = 0$
- While there are still unvisited points
  - $i = i + 1$
  - let $p_i$ be the closest unvisited point to $p_{i-1}$
  - Visit $p_i$
- Return to $p_0$ from $p_i$

## Advantage of that algorithm

- It is simple to understand and implement; It's very efficient: $O(n)$

## But it is not correct!



# Nearest Neighbor Tour

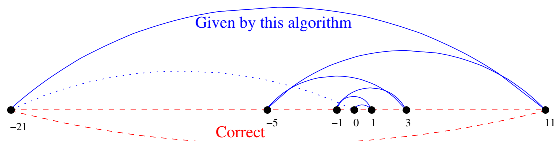
## Here is a very popular algorithm to that problem

- Pick and visit an initial point $p_0$, and let $i = 0$
- While there are still unvisited points
  - $i = i + 1$
  - let $p_i$ be the closest unvisited point to $p_{i-1}$
  - Visit $p_i$
- Return to $p_0$ from $p_i$

## Advantage of that algorithm

- It is simple to understand and implement; It's very efficient: $O(n)$

## But it is not correct!




Choosing carefully $p_0$
(left-most or whatever)
will **not** help

# Closest Pair Tour

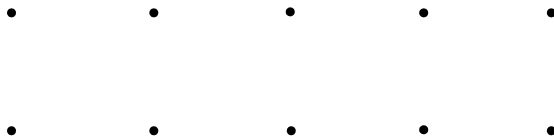
## Let's try to fix our algorithm

- Walking to the closest point seems too restrictive: traps into unwanted moves
- Let's repeatedly connect the closest pairs (w/o forming cycles or 3ways branches)

## The algorithm

- Let $n$ be the number of points in the set
- For $i = 1$ to $n - 1$ do
  - Let $d = \infty$
  - For each pair of endpoints $(x, y)$ of partial paths
    - If $dist(x, y) \leq d$ then $x_m = x$, $y_m = y$, $d = dist(x, y)$
  - Connect $(x_m, y_m)$ by an edge
- Connect the two endpoints by an edge

## Works correctly for previous data



# Closest Pair Tour

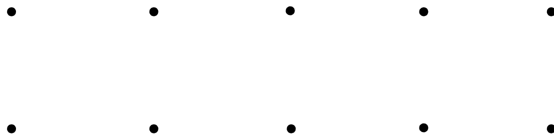
## Let's try to fix our algorithm

- Walking to the closest point seems too restrictive: traps into unwanted moves
- Let's repeatedly connect the closest pairs (w/o forming cycles or 3ways branches)

## The algorithm

- Let $n$ be the number of points in the set
- For $i = 1$ to $n - 1$ do
  - Let $d = \infty$
  - For each pair of endpoints $(x, y)$ of partial paths
    - If $dist(x, y) \leq d$ then $x_m = x$, $y_m = y$, $d = dist(x, y)$
  - Connect $(x_m, y_m)$ by an edge
- Connect the two endpoints by an edge

## Works correctly for previous data, but still not correct



# Closest Pair Tour

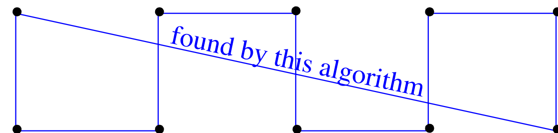
## Let's try to fix our algorithm

- Walking to the closest point seems too restrictive: traps into unwanted moves
- Let's repeatedly connect the closest pairs (w/o forming cycles or 3ways branches)

## The algorithm

- Let $n$ be the number of points in the set
- For $i = 1$ to $n - 1$ do
  - Let $d = \infty$
  - For each pair of endpoints $(x, y)$ of partial paths
    - If $dist(x, y) \leq d$ then $x_m = x$, $y_m = y$, $d = dist(x, y)$
  - Connect $(x_m, y_m)$ by an edge
- Connect the two endpoints by an edge

## Works correctly for previous data, but still not correct



# Closest Pair Tour

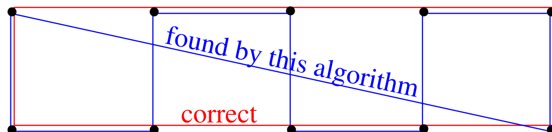
## Let's try to fix our algorithm

- Walking to the closest point seems too restrictive: traps into unwanted moves
- Let's repeatedly connect the closest pairs (w/o forming cycles or 3ways branches)

## The algorithm

- Let $n$ be the number of points in the set
- For $i = 1$ to $n - 1$ do
  - Let $d = \infty$
  - For each pair of endpoints $(x, y)$ of partial paths
    - If $dist(x, y) \leq d$ then $x_m = x$, $y_m = y$, $d = dist(x, y)$
  - Connect $(x_m, y_m)$ by an edge
- Connect the two endpoints by an edge

## Works correctly for previous data, but still not correct



# That's the Traveling Salesman Problem

## A correct algorithm

- $d = \infty$
- For each permutation $\Pi_i$ of the $n!$ existing ones
  - if $(cost(\Pi_i) \leq d)$ then
    - $d = cost(\Pi_i)$ and $P_{min} = \Pi_i$
- return $P_i$

## Actually no known correct and polynomial algorithm

- This algorithm is very slow (exponential time)
- But that's the only correct known
- (this problem is one of the NP-Complete set, by the way)

Conclusion: never trust "obviously correct" algorithms

# Other try to convince septics: you test it

## Issues

- a *whole* load of arrays exists out there. Cannot test them all...
- How much should you test to get convincing? Which ones do you pick?

## Let's look at another (simpler) problem

- Input: 3 integers values, representing the sides' length of a triangle
- Output: Tells whether the triangle is
  - Scalene: no two sides are equal
  - Isosceles: exactly two sides are equal
  - Equilateral: all sides are equal

## Quiz: Create a set of Test Cases for this program

- Ie, the list of tests you need to write to ensure that the program is robust

# Solutions – 1 point for each correct answer

- T1:
- T2:
- T3:
- T4:
- T5:
- T6:
- T7:
- T8:
- T9:
- T10:
- T11:
- T12:
- T13:

# Solutions – 1 point for each correct answer

- T1: (4,1,2): Invalid triangle (because $(a, b, c)$ with $a > b + c$)

- T2:
- T3:
- T4:
- T5:
- T6:
- T7:
- T8:
- T9:
- T10:
- T11:
- T12:
- T13:

# Solutions – 1 point for each correct answer

- T1: (4,1,2): Invalid triangle (because $(a, b, c)$ with $a > b + c$)

- T2: Permutations of the previous: $(1, 2, 4)$, $(2, 1, 4)$
- T3:
- T4:
- T5:
- T6:
- T7:
- T8:
- T9:
- T10:
- T11:
- T12:
- T13:

# Solutions – 1 point for each correct answer

- T1: (4,1,2): Invalid triangle (because $(a, b, c)$ with $a > b + c$)

- T2: Permutations of the previous: $(1, 2, 4)$, $(2, 1, 4)$

$\leadsto$ valid iff $a \leq b + c$ **and** $\mathbf{b \leq a + c}$ **and** $\mathbf{c \leq a + b}$

- T3:
- T4:
- T5:
- T6:
- T7:
- T8:
- T9:
- T10:
- T11:
- T12:
- T13:

# Solutions – 1 point for each correct answer

- T1: (4,1,2): Invalid triangle (because $(a, b, c)$ with $a > b + c$)
- T2: Permutations of the previous: $(1, 2, 4)$, $(2, 1, 4)$

  $\rightsquigarrow$ valid iff $a \leq b + c$ **and** $\mathbf{b} \leq \mathbf{a} + \mathbf{c}$ **and** $\mathbf{c} \leq \mathbf{a} + \mathbf{b}$
- T3: Invalid with equal sum: $(4, 2, 2)$ $\rightsquigarrow$ need to use $<$ instead of $\leq$
- T4:
- T5:
- T6:
- T7:
- T8:
- T9:
- T10:
- T11:
- T12:
- T13:

# Solutions – 1 point for each correct answer

- T1: (4,1,2): Invalid triangle (because $(a, b, c)$ with $a > b + c$)
- T2: Permutations of the previous: $(1, 2, 4)$, $(2, 1, 4)$

$\leadsto$ valid iff $a \leq b + c$ **and** $\mathbf{b} \leq \mathbf{a} + \mathbf{c}$ **and** $\mathbf{c} \leq \mathbf{a} + \mathbf{b}$

- T3: Invalid with equal sum: $(4, 2, 2)$ $\leadsto$ need to use $<$ instead of $\leq$
- T4: Permutations of the previous: $(2, 4, 2)$, $(2, 2, 4)$
- T5:
- T6:
- T7:
- T8:
- T9:
- T10:
- T11:
- T12:
- T13:

# Solutions – 1 point for each correct answer

- T1: (4,1,2): Invalid triangle (because $(a, b, c)$ with $a > b + c$)

- T2: Permutations of the previous: $(1, 2, 4)$, $(2, 1, 4)$

$\rightsquigarrow$ valid iff $a \leq b + c$ **and** $\mathbf{b} \leq \mathbf{a} + \mathbf{c}$ **and** $\mathbf{c} \leq \mathbf{a} + \mathbf{b}$

- T3: Invalid with equal sum: $(4, 2, 2)$ $\rightsquigarrow$ need to use $<$ instead of $\leq$
- T4: Permutations of the previous: $(2, 4, 2)$, $(2, 2, 4)$
- T5: A valid scalene triangle: $(3, 4, 5)$
- T6:
- T7:
- T8:
- T9:
- T10:
- T11:
- T12:
- T13:

# Solutions – 1 point for each correct answer

- T1: (4,1,2): Invalid triangle (because $(a, b, c)$ with $a > b + c$)
- T2: Permutations of the previous: $(1, 2, 4)$, $(2, 1, 4)$

  $\rightsquigarrow$ valid iff $a \leq b + c$ **and** $\mathbf{b} \leq \mathbf{a} + \mathbf{c}$ **and** $\mathbf{c} \leq \mathbf{a} + \mathbf{b}$
- T3: Invalid with equal sum: $(4, 2, 2)$ $\rightsquigarrow$ need to use $<$ instead of $\leq$
- T4: Permutations of the previous: $(2, 4, 2)$, $(2, 2, 4)$
- T5: A valid scalene triangle: $(3, 4, 5)$
- T6: A valid equilateral triangle: $(3, 3, 3)$
- T7:
- T8:
- T9:
- T10:
- T11:
- T12:
- T13:

# Solutions – 1 point for each correct answer

- T1: (4,1,2): Invalid triangle (because $(a, b, c)$ with $a > b + c$)

- T2: Permutations of the previous: $(1, 2, 4)$, $(2, 1, 4)$

  $\rightsquigarrow$ valid iff $a \leq b + c$ **and** $\mathbf{b} \leq \mathbf{a} + \mathbf{c}$ **and** $\mathbf{c} \leq \mathbf{a} + \mathbf{b}$
- T3: Invalid with equal sum: $(4, 2, 2)$ $\rightsquigarrow$ need to use $<$ instead of $\leq$
- T4: Permutations of the previous: $(2, 4, 2)$, $(2, 2, 4)$
- T5: A valid scalene triangle: $(3, 4, 5)$
- T6: A valid equilateral triangle: $(3, 3, 3)$
- T7: A valid isoceles triangle: $(3, 4, 3)$
- T8:
- T9:
- T10:
- T11:
- T12:
- T13:

# Solutions – 1 point for each correct answer

- T1: (4,1,2): Invalid triangle (because $(a, b, c)$ with $a > b + c$)

- T2: Permutations of the previous: $(1, 2, 4)$, $(2, 1, 4)$

  $\rightsquigarrow$ valid iff $a \leq b + c$ **and** $\mathbf{b} \leq \mathbf{a} + \mathbf{c}$ **and** $\mathbf{c} \leq \mathbf{a} + \mathbf{b}$
- T3: Invalid with equal sum: $(4, 2, 2)$ $\rightsquigarrow$ need to use $<$ instead of $\leq$
- T4: Permutations of the previous: $(2, 4, 2)$, $(2, 2, 4)$
- T5: A valid scalene triangle: $(3, 4, 5)$
- T6: A valid equilateral triangle: $(3, 3, 3)$
- T7: A valid isoceles triangle: $(3, 4, 3)$
- T8: All permutations of isoceles triangle: $(4, 3, 3)$,$(3, 3, 4)$
- T9:
- T10:
- T11:
- T12:
- T13:

# Solutions – 1 point for each correct answer

- T1: (4,1,2): Invalid triangle (because $(a, b, c)$ with $a > b + c$)

- T2: Permutations of the previous: $(1, 2, 4)$, $(2, 1, 4)$

$\leadsto$ valid iff $a \leq b + c$ **and** $\mathbf{b} \leq \mathbf{a} + \mathbf{c}$ **and** $\mathbf{c} \leq \mathbf{a} + \mathbf{b}$

- T3: Invalid with equal sum: $(4, 2, 2) \leadsto$ need to use $<$ instead of $\leq$
- T4: Permutations of the previous: $(2, 4, 2)$, $(2, 2, 4)$
- T5: A valid scalene triangle: $(3, 4, 5)$
- T6: A valid equilateral triangle: $(3, 3, 3)$
- T7: A valid isoceles triangle: $(3, 4, 3)$
- T8: All permutations of isoceles triangle: $(4, 3, 3)$,$(3, 3, 4)$
- T9: One side with **zero** value: $(0, 4, 3)$
- T10:
- T11:
- T12:
- T13:

# Solutions – 1 point for each correct answer

- T1: (4,1,2): Invalid triangle (because $(a, b, c)$ with $a > b + c$)

- T2: Permutations of the previous: $(1, 2, 4)$, $(2, 1, 4)$

  $\leadsto$ valid iff $a \leq b + c$ **and** $\mathbf{b} \leq \mathbf{a} + \mathbf{c}$ **and** $\mathbf{c} \leq \mathbf{a} + \mathbf{b}$
- T3: Invalid with equal sum: $(4, 2, 2)$ $\leadsto$ need to use $<$ instead of $\leq$
- T4: Permutations of the previous: $(2, 4, 2)$, $(2, 2, 4)$
- T5: A valid scalene triangle: $(3, 4, 5)$
- T6: A valid equilateral triangle: $(3, 3, 3)$
- T7: A valid isoceles triangle: $(3, 4, 3)$
- T8: All permutations of isoceles triangle: $(4, 3, 3)$,$(3, 3, 4)$
- T9: One side with **zero** value: $(0, 4, 3)$
- T10: One side with **negative** value: $(-1, 4, 3)$
- T11:
- T12:
- T13:

# Solutions – 1 point for each correct answer

- T1: (4,1,2): Invalid triangle (because $(a, b, c)$ with $a > b + c$)
- T2: Permutations of the previous: $(1, 2, 4)$, $(2, 1, 4)$

$\rightsquigarrow$ valid iff $a \leq b + c$ **and** $\mathbf{b} \leq \mathbf{a} + \mathbf{c}$ **and** $\mathbf{c} \leq \mathbf{a} + \mathbf{b}$

- T3: Invalid with equal sum: $(4, 2, 2)$ $\rightsquigarrow$ need to use $<$ instead of $\leq$
- T4: Permutations of the previous: $(2, 4, 2)$, $(2, 2, 4)$
- T5: A valid scalene triangle: $(3, 4, 5)$
- T6: A valid equilateral triangle: $(3, 3, 3)$
- T7: A valid isoceles triangle: $(3, 4, 3)$
- T8: All permutations of isoceles triangle: $(4, 3, 3)$,$(3, 3, 4)$
- T9: One side with **zero** value: $(0, 4, 3)$
- T10: One side with **negative** value: $(-1, 4, 3)$
- T11: All sides with **zero** values: $(0, 0, 0)$
- T12:
- T13:

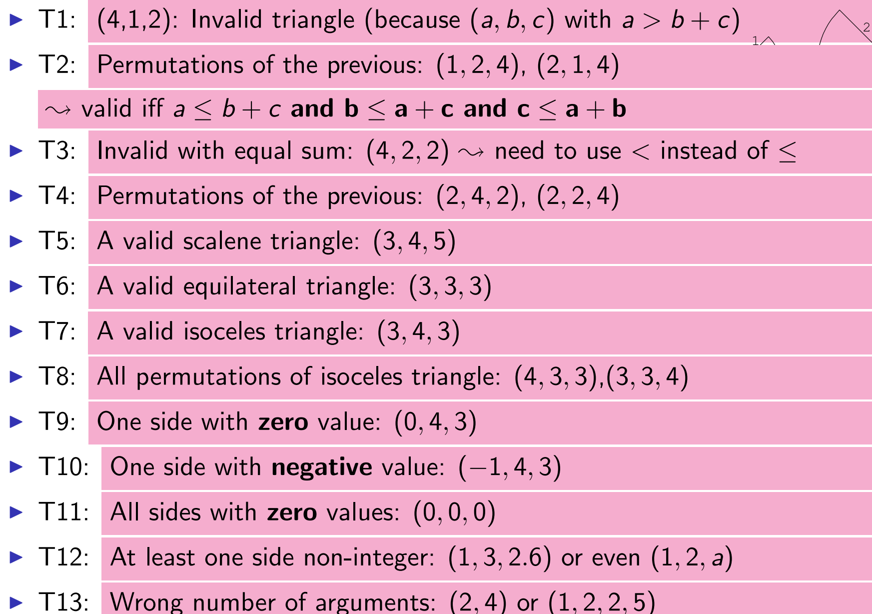
# Solutions – 1 point for each correct answer

- T1: (4,1,2): Invalid triangle (because $(a, b, c)$ with $a > b + c$)
- T2: Permutations of the previous: $(1, 2, 4)$, $(2, 1, 4)$

$\leadsto$ valid iff $a \leq b + c$ **and** $\mathbf{b} \leq \mathbf{a} + \mathbf{c}$ **and** $\mathbf{c} \leq \mathbf{a} + \mathbf{b}$

- T3: Invalid with equal sum: $(4, 2, 2) \leadsto$ need to use $<$ instead of $\leq$
- T4: Permutations of the previous: $(2, 4, 2)$, $(2, 2, 4)$
- T5: A valid scalene triangle: $(3, 4, 5)$
- T6: A valid equilateral triangle: $(3, 3, 3)$
- T7: A valid isoceles triangle: $(3, 4, 3)$
- T8: All permutations of isoceles triangle: $(4, 3, 3)$,$(3, 3, 4)$
- T9: One side with **zero** value: $(0, 4, 3)$
- T10: One side with **negative** value: $(-1, 4, 3)$
- T11: All sides with **zero** values: $(0, 0, 0)$
- T12: At least one side non-integer: $(1, 3, 2.6)$ or even $(1, 2, a)$
- T13:

# Solutions – 1 point for each correct answer



# First Conclusions on Testing

## About the Quiz

- All T1-T13 correspond to failures actually found in some implementations
- How many tests did you found yourself?
  $< 5$?  $5 - 7$?  $8 - 10$?  $> 10$?  All?
- Highly qualified, experienced programmers score 7.8 on average

## Testing aint easy

- Finding good and sufficiently many test cases is difficult
- Even a good set of test cases cannot exclude all failures
- Without a specification, it is not clear even what a failure is

# Stop academic examples, check Real Life!

## Cost of Software Errors: some numbers

- \$60 billion: Estimated cost of software errors for US economy per year [2002]
- \$240 billion: Size of US software industry [2002]
  incl. profit, sales, marketing, development (50% maybe)
- 50%: estimated part of each software project spent on testing
  (spans from 30% to 80%)
- Rough estimate: money spent on testing ≈ cost of remaining errors
- That's 50% of size of software industry!

## More on Testing next week

- We need systematic, efficient, tool supported testing and debugging methods

## To convince real septics, you have to **prove** correctness

- And you cannot do that without a proper specification (at least)

# Fourth Chapter

## Correction of Software Systems


- Introduction
- Specification
- Hoare Logic
- Proving Recursive Functions
- Conclusion

# How to prove that 'selection sort' sorts arrays?

## Back to the roots: what exactly do you want to prove?

- Proper specification mandatory to proof: gives what we have, what we want
- We also need a mathematical logic to carry the proof

## Hoare Logic [Hoare 1969]

- Set of logical rules to reason about the correctness of computer programs
- Central feature: description of state changes induced by code execution
- **Hoare triple:** $\{P\}\ C\ \{Q\}$
  - C is the code to be run
  - P is the **precondition** (assertion about previous state)
  - Q is the **postcondition** (assertion about next state)
  - This can be read as "If P is true, then when I run C, Q becomes true"
  - C is said to satisfy specification $(P, Q)$
- Such notation allows very precise algorithm specifications
- Axioms and Inference rules allow rigorous correctness demonstrations
- Note: other logics (temporal logic) proposed as replacement, but harder

# Introducing (bad) joke about precise specification

While traversing Scotland, 3 people see a cow

# Introducing (bad) joke about precise specification

While traversing Scotland, 3 people see a cow

The Economist says:

- Cows in Scotland are brown

# Introducing (bad) joke about precise specification

While traversing Scotland, 3 people see a cow

The Economist says:

- Cows in Scotland are brown

The Logician says:

- No, no. There are cows in Scotland of which one is brown

# Introducing (bad) joke about precise specification

While traversing Scotland, 3 people see a cow

The Economist says:

- Cows in Scotland are brown

The Logician says:

- No, no. There are cows in Scotland of which one is brown

The Computer Scientist says:

- No, no. There is at least one cow in Scotland of which one side is brown

# Specification: Putting it into Practice

## Back to our Example: A sorting program

```
public static Integer[ ] sort(Integer[ ] a) { ... }
```

## Specification

- Precondition: $a$ is an array
- Post-condition: returns the sorted argument array

# Specification: Putting it into Practice

## Back to our Example: A sorting program

```
public static Integer[ ] sort(Integer[ ] a) { ... }
```

## Specification

- Precondition: *a* is an array
- Post-condition: returns the sorted argument array
- Is it good enough?

# Specification: Putting it into Practice

## Back to our Example: A sorting program

`public static Integer[ ] sort(Integer[ ] a) { ... }`

## Specification

- Precondition: *a* is an array
- Post-condition: returns the sorted argument array
- Is it good enough? Not quite: `sort`($\{2,1,2\} \rightsquigarrow \{1,2,2,17\}$) ☹

# Specification: Putting it into Practice

## Back to our Example: A sorting program

```
public static Integer[ ] sort(Integer[ ] a) { ... }
```

## Specification V1

- Precondition: $a$ is an array
- Post-condition: returns the sorted argument array
- Is it good enough? Not quite: sort({2,1,2} $\leadsto$ {1,2,2,17} ☹

## Specification V2

- Precondition: $a$ is an array
- Post-condition: returns a sorted array with only elements of $a$

# Specification: Putting it into Practice

## Back to our Example: A sorting program

`public static Integer[ ] sort(Integer[ ] a) { ... }`

## Specification V1

- Precondition: *a* is an array
- Post-condition: returns the sorted argument array
- Is it good enough? Not quite: `sort({2,1,2}` $\leadsto$ `{1,2,2,17}` ☹

## Specification V2

- Precondition: *a* is an array
- Post-condition: returns a sorted array with only elements of *a*

☹ `sort({2,1,2}` $\leadsto$ `{1,1,2}`

# Specification: Putting it into Practice

## Back to our Example: A sorting program

`public static Integer[ ] sort(Integer[ ] a) { ... }`

## Specification V1

- Precondition: *a* is an array
- Post-condition: returns the sorted argument array
- Is it good enough? Not quite: `sort({2,1,2})` $\rightsquigarrow$ `{1,2,2,17}` ☹

## Specification V2

- Precondition: *a* is an array
- Post-condition: returns a sorted array with only elements of *a*

☹ `sort({2,1,2})` $\rightsquigarrow$ `{1,1,2}`

## Specification V3

- Precondition: *a* is an array
- Post-condition: returns a permutation of *a* that is sorted

# Specification: Putting it into Practice

## Back to our Example: A sorting program

`public static Integer[ ] sort(Integer[ ] a) { ... }`

## Specification V1

- Precondition: $a$ is an array
- Post-condition: returns the sorted argument array
- Is it good enough? Not quite: `sort({2,1,2}` $\rightsquigarrow$ `{1,2,2,17}` ☹

## Specification V2

- Precondition: $a$ is an array
- Post-condition: returns a sorted array with only elements of $a$

☹ `sort({2,1,2}` $\rightsquigarrow$ `{1,1,2}`

## Specification V3

- Precondition: $a$ is an array
- Post-condition: returns a permutation of $a$ that is sorted

☹ `sort(null) throws NullPointerException`

# Specification: Putting it into Practice

## Back to our Example: A sorting program

```
public static Integer[ ] sort(Integer[ ] a) { ... }
```

## Specification V1

- Precondition: *a* is an array
- Post-condition: returns the sorted argument array
- Is it good enough? Not quite: `sort({2,1,2}` $\leadsto$ `{1,2,2,17}` ☹

## Specification V2

- Precondition: *a* is an array
- Post-condition: returns a sorted array with only elements of *a*

☹ `sort({2,1,2}` $\leadsto$ `{1,1,2}`

## Specification V3

- Precondition: *a* is an array
- Post-condition: returns a permutation of *a* that is sorted

☹ `sort(null) throws NullPointerException`

## Specification V4

- Precondition: *a* is a non-null array
- Post-condition: returns a permutation of *a* that is sorted

# The Contract Metaphor

> *Contract is preferred specification metaphor for procedural and OO.*
> – B. Meyer, Computer 25(10)40-51, 1992

## Same Principles as Legal Contract between a Client and Supplier

Supplier aka Implementer, in Java, a class or method

Client Mostly a caller object, or human user for main()

Contract One or more pairs of ensures/requires clauses
defining mutual obligations of client and implementer

## Meaning of a Contract: Specification of method `C@m()`

- "If a caller of `C@m()` fulfills the required Precondition,
  then the class `C` ensures that the Postcondition holds after `m()` finishes."

## Wrong Interpretations:

- "Any caller of C@m() must fulfill the required Precondition."
- "Whenever the required Precondition holds, then C@m() is executed."

# What is Design By Contract?

> *View the relationship between two classes as a formal agreement, expressing each party's rights and obligations.* – Bertrand Meyer

## Example: Airline Reservation

| | Obligations | Rights |
|---|---|---|
| Customer | ▶ Be at Paris airport at least 3 hour before scheduled departure time<br>▶ Bring acceptable baggage<br>▶ Pay ticket price | ▶ Reach Los Angeles |
| Airline | ▶ Bring customer to Los Angeles | ▶ No need to carry passenger who is late<br>▶ has unacceptable baggage<br>▶ or has not paid ticket |

- Each party expects benefits (rights) and accepts obligations
- Usually, one party's benefits are the other party's obligations
- Contract is declarative: it is described so that both parties can understand *what* service will be guaranteed without saying *how*

# Testing vs. Verification

## Testing

- Goal: find evidence for presence of failures
- Testing means to execute a program with the **intent of detecting failure**
- Related techniques: code reviews, program inspections

## Verification

- Goal: find evidence for absence of failures
- Testing cannot guarantee correctness, i.e., absence of failures
- Related techniques: code generation, program synthesis (from spec)

## Debugging

- Systematic process to find and eliminate defects leading to observed failures

# Testing vs. Verification

## Testing

- Goal: find evidence for presence of failures
- Testing means to execute a program with the **intent of detecting failure**
- Related techniques: code reviews, program inspections
- Next week: How can we automatize the testing process?

## Verification

- Goal: find evidence for absence of failures
- Testing cannot guarantee correctness, i.e., absence of failures
- Related techniques: code generation, program synthesis (from spec)
- This week: How can we prove the correctness of an algorithm?

## Debugging

- Systematic process to find and eliminate defects leading to observed failures

# Fourth Chapter

## Correction of Software Systems


- Introduction
- Specification
- Hoare Logic
- Proving Recursive Functions
- Conclusion

# Back on Hoare Logic

## Hoare Logic [Hoare 1969]

- Set of logical rules to reason about the correctness of computer programs
- Central feature: description of state changes induced by code execution
- **Hoare triple:** $\boxed{\{P\}\ C\ \{Q\}}$
  - C is the code to be run
  - P is the **precondition** (assertion about previous state)
  - Q is the **postcondition** (assertion about next state)
  - This can be read as "If P is true, then when I run C, Q becomes true"
  - C is said to satisfy specification $(P, Q)$
- Such notation allows very precise algorithm specifications
- Axioms and Inference rules allow rigorous correctness demonstrations
- Note: other logics (temporal logic) proposed as replacement, but harder

## Game now

- See how we can **prove** that a Hoare triple holds

# Assertions

What exactly is an assertion?

## Definition

Formula of first order logic describing relationships between algorithm's variables

## Constituted of:

- Variables of algorithm pseudo-code
- Logical connectors: $\wedge$ (and) $\vee$ (or) $\neg$ (not) $\Rightarrow$, $\Leftarrow$
- Quantifiers: $\exists$ (exists), $\forall$ (for all)
- Value-specific elements (describing integers, reals, booleans, arrays, sets, ...)

## Example:

- $(x \times y = z) \ \wedge \ (x \leq 0)$
- $n^2 \geq x$

# Examples of specification

## Solving quadratic equations ($ax^2 + bx + c = 0$)

P: $a, b, c \in \mathbb{R}$ and $a \neq 0$

Q: $(solAmount \in \mathbb{N}) \wedge (s, t \in \mathbb{R}) \wedge$
$( (solAmount = 0) \vee$
$(solAmount = 1 \wedge as^2 + bs + c = 0) \vee$
$(solAmount = 2 \wedge as^2 + bs + c = 0 \wedge at^2 + bt + c = 0 \wedge s \neq t))$

## Possible implementation

```
Δ = b² − 4ac
if (Δ > 0)
  s = (−b+√Δ)/(2a); t = (−b−√Δ)/(2a);
  solAmount = 2
else if (Δ = 0)
  s = −b/(2a); solAmount = 1
else (ie, Δ < 0)
  solAmount=0
```

- Here, the proof will be difficult...
- ...because it is trivial.
- Correctness comes from definitions!

# Demonstration tool: inference rules

## Definitions

- Inference: deducting new facts by combining existing facts correctly
- Inference rule: mechanism specifying how facts can be combined

## Classical representation of each rule:

$$\frac{p_1, p_2, p_3, \ldots, p_n}{q}$$

- Can be read as "if all $p_1, p_2, p_3, \ldots, p_n$ are true, then $q$ is also true"
- Or "in order to prove $q$, you have to prove $p_1, p_2, p_3, \ldots, p_n$"
- Or "$q$ can be deduced from $p_1, p_2, p_3, \ldots, p_n$"

# First axioms and rules

Empty statement axiom

$$\frac{}{\{P\}skip\{P\}}$$

Assignment axiom

$$\frac{}{\{P[x/E]\}x := E\{P\}}$$

- $P[x/E]$ is $P$ with all free occurrences of variable $x$ replaced with expression $E$
- Example:
  - P: $x = a \wedge y = b$
  - Q: $x = b \wedge y = a$
  - SWAP: algorithm achieving transition; For example: $t = x; x = y; y = t$
  - We should prove: $\{P\}SWAP\{Q\}$

# First axioms and rules

Empty statement axiom

$$\frac{}{\{P\}skip\{P\}}$$

Assignment axiom

$$\frac{}{\{P[x/E]\}x := E\{P\}}$$

- $P[x/E]$ is $P$ with all free occurrences of variable $x$ replaced with expression $E$
- Example:
  - P: $x = a \wedge y = b$
  - Q: $x = b \wedge y = a$
  - SWAP: algorithm achieving transition; For example: $t = x; x = y; y = t$
  - We should prove: $\{P\}SWAP\{Q\}$

Consequence rule

$$\frac{P' \Rightarrow P, \{P\}\ C\ \{Q\}, Q \Rightarrow Q'}{\{P'\}\ C\ \{Q'\}}$$

- $P$ is said to be weaker than $P'$
- $Q$ is said to be stronger than $Q'$

# Rules for algorithmic constructs

Rule of composition $$\frac{\{P\}C_1\{Q\},\quad \{Q\}C_2\{R\}}{\{P\}C_1; C_2\{R\}}$$

- $C_1; C_2$ means that both code are executed one after the other.
- Can naturally be generalized to more than two codes

# Rules for algorithmic constructs

Rule of composition
$$\frac{\{P\}C_1\{Q\},\quad \{Q\}C_2\{R\}}{\{P\}C_1;C_2\{R\}}$$

- $C_1; C_2$ means that both code are executed one after the other.
- Can naturally be generalized to more than two codes

Conditional Rule
$$\frac{\{P \land Cond\}\ T\ \{Q\},\quad P \land \neg Cond \Rightarrow Q}{\{P\}\ \textbf{if}\ Cond\ \textbf{then}\ T\ \textbf{endif}\ \{Q\}}$$

# Rules for algorithmic constructs

Rule of composition
$$\frac{\{P\}C_1\{Q\},\quad \{Q\}C_2\{R\}}{\{P\}C_1;C_2\{R\}}$$

- $C_1; C_2$ means that both code are executed one after the other.
- Can naturally be generalized to more than two codes

Conditional Rule
$$\frac{\{P \land Cond\}\ T\ \{Q\},\quad P \land \neg Cond \Rightarrow Q}{\{P\}\ \textbf{if}\ Cond\ \textbf{then}\ T\ \textbf{endif}\ \{Q\}}$$

Conditional Rule 2
$$\frac{\{P \land Cond\}\ T\ \{Q\},\quad \{P \land \neg Cond\}\ E\ \{Q\}}{\{P\}\ \textbf{if}\ Cond\ \textbf{then}\ T\ \textbf{else}\ E\ \textbf{endif}\ \{Q\}}$$

# Rules for algorithmic constructs

Rule of composition
$$\frac{\{P\}C_1\{Q\}, \quad \{Q\}C_2\{R\}}{\{P\}C_1; C_2\{R\}}$$

- $C_1; C_2$ means that both code are executed one after the other.
- Can naturally be generalized to more than two codes

Conditional Rule
$$\frac{\{P \wedge Cond\}\ T\ \{Q\}, \quad P \wedge \neg Cond \Rightarrow Q}{\{P\}\ \textbf{if}\ Cond\ \textbf{then}\ T\ \textbf{endif}\ \{Q\}}$$

Conditional Rule 2
$$\frac{\{P \wedge Cond\}\ T\ \{Q\}, \quad \{P \wedge \neg Cond\}\ E\ \{Q\}}{\{P\}\ \textbf{if}\ Cond\ \textbf{then}\ T\ \textbf{else}\ E\ \textbf{endif}\ \{Q\}}$$

While Rule
$$\frac{\{I \wedge Cond\}\ L\ \{I\}}{\{I\}\ \textbf{while}\ Cond\ \textbf{do}\ L\ \textbf{endif}\ \{I \wedge \neg Cond\}}$$

- $\{I\}$ is said to be the **loop invariant**

# How to prove algorithms?

## The two things to prove about algorithms

- Correction proof: when it terminates, the algorithm produce a valid result with regard to problem specification
- Termination proof: the algorithm always terminate

## How to proceed? Two approaches

- Think forward: Go from precondition to post-condition
- Think backward:
  Compute the weakest precondition you need to get the postcondition you want

# How to prove algorithms?

## The two things to prove about algorithms

- Correction proof: when it terminates, the algorithm produce a valid result with regard to problem specification
- Termination proof: the algorithm always terminate

## How to proceed? Two approaches

- Think forward: Go from precondition to post-condition
- Think backward:
  Compute the weakest precondition you need to get the postcondition you want

## Advantages

- Thinking forward *seems* easy, but you need to know where you go
- Thinking backward is easier, because post-condition is what you know best
- We'll do both in labs...

# Proving backward: Weakest Preconditions

## Computing the WP to get the post-condition {Q} from C – WP(C,Q)

1. $\mathbf{WP}(nop, Q) \equiv Q$
2. $\mathbf{WP}(x := E, Q) \equiv Q[x := E]$
3. $\mathbf{WP}(C; D, Q) \equiv \mathbf{WP}(C, \mathbf{WP}(\mathrm{D,Q}))$
4. $\mathbf{WP}(\texttt{if } Cond \texttt{ then } C \texttt{ else } D, \mathrm{Q})$
   $\equiv (Cond = \texttt{true} \Rightarrow \mathbf{WP}(C, Q)) \;\wedge\; (Cond = \texttt{false} \Rightarrow \mathbf{WP}(D, Q))$
5. $\mathbf{WP}(\texttt{while } E \texttt{ do } C \texttt{ done }, \mathrm{Q}) \equiv I$ (with I invariant, V variant)
   Plus the following proof obligations:
   - $(E = \texttt{true} \wedge I \wedge V = z) \Rightarrow \mathbf{WP}(C, I \wedge V < z))$ (variant gets decremented)
   - $I \Rightarrow V \geq 0$ (variant remains valid)
   - $(E = \texttt{false} \wedge I) \Rightarrow Q$ (once done, Q is achieved)

## Seems complicated, but isn't that much

- The process is automated enough to keep quite mecanical and simple
- We'll come back on this in lab (and exam ;)

# Fourth Chapter

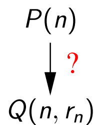
## Correction of Software Systems


- Introduction
- Specification
- Hoare Logic
- Proving Recursive Functions
- Conclusion

# Idea of the correction of recursive function

$P(n)$: Precondition at step $n$; $Q(n, r_n)$: Postcondition at step $n$ with result $r_n$

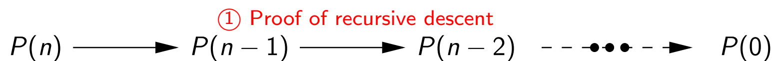
**We want to show $\mathbf{P(n)\ \{TREC\}\ Q(n, r_n)}$**



# Idea of the correction of recursive function

$P(n)$: Precondition at step $n$; $Q(n, r_n)$: Postcondition at step $n$ with result $r_n$

**We want to show $\mathbf{P(n)}$ {TREC} $\mathbf{Q(n, r_n)}$**




$Q(n, r_n)$

If $f(n)$ is expressed as function of $f(n-1)$, we need:

- In recursive case
  - Precondition of $f(n)$ implies precondition of $f(n-1)$ ①
    If not, the computation is impossible

# Idea of the correction of recursive function

$P(n)$: Precondition at step $n$; $Q(n, r_n)$: Postcondition at step $n$ with result $r_n$

**We want to show $\mathbf{P(n)\ \{TREC\}\ Q(n, r_n)}$**

① Proof of recursive descent

$P(n) \longrightarrow P(n-1) \longrightarrow P(n-2) \dashrightarrow \cdots \dashrightarrow P(0)$

② Proof of base case

$Q(n, r_n)$ $\qquad$ $Q(0, r_0)$

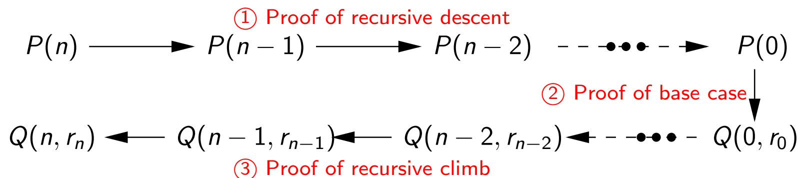
If $f(n)$ is expressed as function of $f(n-1)$, we need:

- In recursive case
  - Precondition of $f(n)$ implies precondition of $f(n-1)$ ①
    If not, the computation is impossible
- In base case
  - precondition and computation allow to prove postcondition ②

# Idea of the correction of recursive function

$P(n)$: Precondition at step $n$; $Q(n, r_n)$: Postcondition at step $n$ with result $r_n$

**We want to show $\mathbf{P(n)\ \{TREC\}\ Q(n, r_n)}$**




If $f(n)$ is expressed as function of $f(n-1)$, we need:

- In recursive case
  - Precondition of $f(n)$ implies precondition of $f(n-1)$ ①
    If not, the computation is impossible
  - Hyp: postcondition of $f(n-1)$ true. Proof postcondition of $f(n)$ ③
- In base case
  - precondition and computation allow to prove postcondition ②

# Proof of correctness (1/2)

$$P(n)\ \{\mathit{TREC}\}\ Q(n, r_n) \tag{1}$$

$$P(n)\ \{\textbf{if}\ \text{cond}\ \textbf{then}\ \text{TTER}\ \textbf{else}\ \text{TGEN}\}\ Q(n, r_n)$$

# Proof of correctness (1/2)

$$P(n)\ \{TREC\}\ Q(n, r_n) \quad (1)$$

$$P(n)\ \{\textbf{if}\ \text{cond}\ \textbf{then}\ \text{TTER}\ \textbf{else}\ \text{TGEN}\}\ Q(n, r_n)$$

Simple case: TGEN and TTER are affectations

TGEN: $r \leftarrow G(n, f(n_{int}))$

TTER: $r \leftarrow v(n)$

# Proof of correctness (1/2)

$$P(n)\ \{TREC\}\ Q(n, r_n) \quad (1)$$

$$P(n)\ \{\textbf{if } \text{cond } \textbf{then } \text{TTER } \textbf{else } \text{TGEN}\}\ Q(n, r_n)$$

**Simple case**: TGEN and TTER are affectations

TGEN: $r \leftarrow G(n, f(n_{int}))$

TTER: $r \leftarrow v(n)$

With

$n_{int}$ Value of recursive call

$f(x)$ The recursive call

$v(n)$ Function w/o call to $f(n)$

$G(n, y)$ Function:

- W/O recursive call to $f(n)$
- Defined $\forall n$ parameter, $\forall y$

# Proof of correctness (1/2)

$$P(n)\ \{\,TREC\,\}\ Q(n, r_n) \qquad (1)$$

$$P(n)\ \{\textbf{if}\ \text{cond}\ \textbf{then}\ \text{TTER}\ \textbf{else}\ \text{TGEN}\}\ Q(n, r_n)$$

**Simple case**: TGEN and TTER are affectations

TGEN: $r \leftarrow G(n, f(n_{int}))$

TTER: $r \leftarrow v(n)$

With

$n_{int}$ Value of recursive call

$f(x)$ The recursive call

$v(n)$ Function w/o call to $f(n)$

$G(n, y)$ Function:

- W/O recursive call to $f(n)$
- Defined $\forall n$ parameter, $\forall y$

**Example**: Factorial

TGEN: $r \leftarrow n \times facto(n-1)$

TTER: $r \leftarrow 1$

$n_{int} = n - 1$

$f(x)$ : $facto(x)$

v(n) $= 1$

G(n,y) $= n \times y$

# Proof of correctness (1/2)

$$P(n)\ \{\,TREC\,\}\ Q(n, r_n) \qquad (1)$$

$$P(n)\ \{\textbf{if}\ \text{cond}\ \textbf{then}\ \text{TTER}\ \textbf{else}\ \text{TGEN}\}\ Q(n, r_n)$$

**Simple case**: TGEN and TTER are affectations

TGEN: $r \leftarrow G(n, f(n_{int}))$

TTER: $r \leftarrow v(n)$

With

$n_{int}$ Value of recursive call

$f(x)$ The recursive call

$v(n)$ Function w/o call to $f(n)$

$G(n, y)$ Function:

- W/O recursive call to $f(n)$
- Defined $\forall n$ parameter, $\forall y$

**Example**: Factorial

TGEN: $r \leftarrow n \times facto(n-1)$

TTER: $r \leftarrow 1$

$n_{int} = n - 1$

$f(x)$ : $facto(x)$

v(n) $= 1$

G(n,y) $= n \times y$

$P(n)$ : $n \geq 0$

$Q(n, r)$ : $r = n!$

cond(n) : n=0

# Proof of correctness (2/2)

Simple case: TTER and TGEN are affectations

- Algorithm computing $r = f(n)$

  **if** cond(n) **then** $r \leftarrow v(x)$
  **else** $r \leftarrow G(n, f(n_{int}))$

- To prove $\boxed{P(n)\,\{\,TREC\,\}\;Q(n, r_n)}$, it is sufficient to prove:
  - In terminal case: precondition and computation imply postcondition
    $$P(n) \wedge cond(n) \Rightarrow Q(n, r)$$
  - In general case:
    - Recursive descent: precondition of $f(n)$ implies precondition of $f(n-1)$
      $$P(n) \wedge \neg cond(n) \Rightarrow P(n_{int})$$
    - Recursive Climb: postcondition of $f(n-1)$ implies postcondition of $f(n)$
      $$P(n) \wedge \neg cond(n) \wedge Q(n_{int}, r_{int}) \Rightarrow Q(n, r)$$

# Proof of correctness (2/2)

## Simple case: TTER and TGEN are affectations

- Algorithm computing $r = f(n)$

  **if** cond(n) **then** $r \leftarrow v(x)$
  **else** $r \leftarrow G(n, f(n_{int}))$

- To prove $\boxed{P(n)\ \{\ TREC\ \}\ Q(n, r_n)}$, it is sufficient to prove:
  - In terminal case: precondition and computation imply postcondition
    $$P(n) \wedge cond(n) \Rightarrow Q(n, r)$$
  - In general case:
    - Recursive descent: precondition of $f(n)$ implies precondition of $f(n-1)$
      $$P(n) \wedge \neg cond(n) \Rightarrow P(n_{int})$$
    - Recursive Climb: postcondition of $f(n-1)$ implies postcondition of $f(n)$
      $$P(n) \wedge \neg cond(n) \wedge Q(n_{int}, r_{int}) \Rightarrow Q(n, r)$$

## General Case: harder

- You need to combine with the proofs of the previous section
  $$P(x) \Rightarrow P(x_{int}) \qquad\qquad Q(x_{int}, r_{int}) \Rightarrow Q(x, r)$$

# Example of the factorial (how unexpected)

$$
\begin{array}{llr}
\textsc{factorial}(\mathsf{n}): & & \\
\quad \textbf{if } \mathsf{n} = 0 \textbf{ then} r \leftarrow 1 & & (\textsc{tter}) \\
\quad \qquad\qquad \textbf{else } r \leftarrow n \times \mathit{factorial}(n-1) & & (\textsc{tgen}) \\
\quad \textbf{endif} & &
\end{array}
$$

$P(n)$: $n \geq 0$ cond(n): $n = 0$ $Q(n, r)$: $r = n!$ $n_{int} = n - 1$

- Terminal Case: $P(n) \wedge cond(n) \Rightarrow Q(n, r)$
- General Case:
  - $P(n) \wedge \neg cond(n) \Rightarrow P(n_{int})$
  - $P(n) \wedge \neg cond(n) \wedge Q(n_{int}, r_{int}) \Rightarrow Q(n, r)$

# Example of the factorial (how unexpected)

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

$P(n)$: $n \geq 0$  cond(n): $n = 0$  $Q(n,r)$: $r = n!$  $n_{int} = n - 1$

- Terminal Case: $P(n) \wedge cond(n) \Rightarrow Q(n,r) \equiv n \geq 0 \wedge n = 0 \Rightarrow r = n!$
- General Case:
  - $P(n) \wedge \neg cond(n) \Rightarrow P(n_{int})$
  - $P(n) \wedge \neg cond(n) \wedge Q(n_{int}, r_{int}) \Rightarrow Q(n,r)$

# Example of the factorial (how unexpected)

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

$P(n)$: $n \geq 0$ cond(n): $n = 0$ $Q(n, r)$: $r = n!$ $n_{int} = n - 1$

- Terminal Case: $P(n) \wedge cond(n) \Rightarrow Q(n, r) \equiv n \geq 0 \wedge n = 0 \Rightarrow r = n!$
  True (since $1 = 0!$ no matter what happens)
- General Case:
  - $P(n) \wedge \neg cond(n) \Rightarrow P(n_{int})$
  - $P(n) \wedge \neg cond(n) \wedge Q(n_{int}, r_{int}) \Rightarrow Q(n, r)$

# Example of the factorial (how unexpected)

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

$P(n)$: $n \geq 0$ cond(n): $n = 0$ $Q(n, r)$: $r = n!$ $n_{int} = n - 1$

- Terminal Case: $P(n) \wedge cond(n) \Rightarrow Q(n, r) \equiv n \geq 0 \wedge n = 0 \Rightarrow r = n!$
  True (since $1 = 0!$ no matter what happens)
- General Case:
  - $P(n) \wedge \neg cond(n) \Rightarrow P(n_{int}) \equiv (n \geq 0) \wedge (n \neq 0) \Rightarrow (n - 1 \geq 0)$
  - $P(n) \wedge \neg cond(n) \wedge Q(n_{int}, r_{int}) \Rightarrow Q(n, r)$

# Example of the factorial (how unexpected)

```
FACTORIAL(n):
  if n = 0 thenr ← 1                        (TTER)
           else r ← n × factorial(n − 1)    (TGEN)
  endif
```

$P(n)$: $n \geq 0$ cond(n): $n = 0$ $Q(n, r)$: $r = n!$ $n_{int} = n - 1$

- Terminal Case: $P(n) \wedge cond(n) \Rightarrow Q(n, r) \equiv n \geq 0 \wedge n = 0 \Rightarrow r = n!$
  True (since $1 = 0!$ no matter what happens)
- General Case:
  - $P(n) \wedge \neg cond(n) \Rightarrow P(n_{int}) \equiv (n \geq 0) \wedge (n \neq 0) \Rightarrow (n - 1 \geq 0)$
    Trivial
  - $P(n) \wedge \neg cond(n) \wedge Q(n_{int}, r_{int}) \Rightarrow Q(n, r)$

# Example of the factorial (how unexpected)

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

$P(n)$: $n \geq 0$  cond(n): $n = 0$  $Q(n,r)$: $r = n!$  $n_{int} = n - 1$

- Terminal Case: $P(n) \wedge cond(n) \Rightarrow Q(n,r) \equiv n \geq 0 \wedge n = 0 \Rightarrow r = n!$
  True (since $1 = 0!$ no matter what happens)
- General Case:
  - $P(n) \wedge \neg cond(n) \Rightarrow P(n_{int}) \equiv (n \geq 0) \wedge (n \neq 0) \Rightarrow (n - 1 \geq 0)$
    Trivial
  - $P(n) \wedge \neg cond(n) \wedge Q(n_{int}, r_{int}) \Rightarrow Q(n,r)$
    $\equiv (n \geq 0) \wedge (n \neq 0) \wedge (r_{int} = n_{int}!) \Rightarrow (r = n!)$

# Example of the factorial (how unexpected)

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

$P(n)$: $n \geq 0$ cond(n): $n = 0$ $Q(n,r)$: $r = n!$ $n_{int} = n - 1$

- Terminal Case: $P(n) \wedge cond(n) \Rightarrow Q(n,r) \equiv n \geq 0 \wedge n = 0 \Rightarrow r = n!$
  True (since $1 = 0!$ no matter what happens)
- General Case:
  - $P(n) \wedge \neg cond(n) \Rightarrow P(n_{int}) \equiv (n \geq 0) \wedge (n \neq 0) \Rightarrow (n - 1 \geq 0)$
    Trivial
  - $P(n) \wedge \neg cond(n) \wedge Q(n_{int}, r_{int}) \Rightarrow Q(n,r)$
    $\equiv (n \geq 0) \wedge (n \neq 0) \wedge (r_{int} = n_{int}!) \Rightarrow (r = n!)$

    True because:
    - $r = n \times r_{int}$ in general case (by *looking at the code*)
    - $r_{int} = n_{int}! = (n-1)!$ by induction hypothesis
    - $r = n \times (n-1)! = n!$

# Proof of Termination

- Sufficient Conditions:
  - Successive values of parameter $x$: strictly monotonous suite (may need to specify the order)
  - Existence of an extrema $x_0$ verifying the stopping condition
- Remarque: The Syracuse suite seems to terminate without this
  So that's no necessary condition

# Proof of Termination

- Sufficient Conditions:
  - Successive values of parameter $x$: strictly monotonous suite (may need to specify the order)
  - Existence of an extrema $x_0$ verifying the stopping condition
- Remarque: The Syracuse suite seems to terminate without this So that's no necessary condition
- Example: the factorial, of course
  - $n \geq 0$
  - n strictly decreasing
  - $0 =$ stopping condition

# Fourth Chapter

## Correction of Software Systems


- Introduction
- Specification
- Hoare Logic
- Proving Recursive Functions
- Conclusion

# Conclusion on Software Correctness Proofs

## That's not easy

- Lot and lot of mathematical work to prove even simple algorithms
- That's something you do only when you really want:
  Aircraft, Nuclear power plant, Emergency room, ...
- But that's not impossible; One success story amongst hundreds:
  SACEM embedded system controling the train speed on the RER Line A in Paris.

## Support from language / automated tools would be welcomed

- Unfortunately, Java is not Ada, or even better: Eiffel
- Some solution exist to mimic Eiffel in Java: JML (Java Modeling Language)
- But non of them seem production-ready yet (but slowly progressing)

## In Java world, Testing is still prevalent

- Even if it's a pity to go on only one leg like this
- More on this in next lecture

# Fifth Chapter

# Testing


- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Testing in Practice
  - JUnit
  - Right BICEP + CORRECT
- Other Techniques for Practical Software Correctness
  - Design By Contract
  - Fuzzing
  - Formal Methods
- Conclusion

# Back on Proofs

## Most of you don't see the point of proofs. I know

- I even agree (to a given point: we need 2 legs to code – theory and practice)

> *Beware of bugs in the above code; I have only proved it correct, not tried it.* – *D.E. Knuth.*

- Cost/gain ratio: if you cannot afford to loose, prove it correct
  I hope that Nuclear Power Plants are [partially] proved
- Can give you a competitive advantage:
  With these, you may accept more complicated contracts
- You're studying in Nancy, there is a local history of algorithm proofs
- There will be 1/4 of points on proofs at the exam ...

## What's expected at the exam

- Well, that's similar to when you write code
- I don't bother a missing } in the code, as long as the idea is here
- I don't bother a partially wrong proof, as long as the method is here

And now, let's see that tests that you guys prefer are even worst

# Introduction

## Bugs

- Bugs are inevitable in complex software system
- A bug can be very visible or can hide in your code until a much later date

## Bugs can hide very well

Aug 2009 8 years old bug found in Linux (handling not implemented kernel fctions)

http://www.theregister.co.uk/2009/08/14/critical_linux_bug/

# Introduction

## Bugs

- Bugs are inevitable in complex software system
- A bug can be very visible or can hide in your code until a much later date

## Bugs can hide very well

Aug 2009 8 years old bug found in Linux (handling not implemented kernel fctions)

http://www.theregister.co.uk/2009/08/14/critical_linux_bug/

Jan 2010 Microsoft fixes a 17 years old bug (in code allowing NT to run 16bits apps)

http://www.esecurityplanet.com/features/article.phpr/3860131/Microsoft-Warns-About-17-year-old-Windows-Bug.htm

# Introduction

## Bugs

- Bugs are inevitable in complex software system
- A bug can be very visible or can hide in your code until a much later date

## Bugs can hide very well

Aug 2009 8 years old bug found in Linux (handling not implemented kernel fctions)
 `http://www.theregister.co.uk/2009/08/14/critical_linux_bug/`

Jan 2010 Microsoft fixes a 17 years old bug (in code allowing NT to run 16bits apps)
 `http://www.esecurityplanet.com/features/article.phpr/3860131/Microsoft-Warns-About-17-year-old-Windows-Bug.htm`

July 2008 25 years old bug found in BSD (`seekdir()` wrongly implemented)
 `http://www.vnode.ch/fixing_seekdir`

# Introduction

## Bugs

- Bugs are inevitable in complex software system
- A bug can be very visible or can hide in your code until a much later date

## Bugs can hide very well

Aug 2009 8 years old bug found in Linux (handling not implemented kernel fctions)
`http://www.theregister.co.uk/2009/08/14/critical_linux_bug/`

Jan 2010 Microsoft fixes a 17 years old bug (in code allowing NT to run 16bits apps)
`http://www.esecurityplanet.com/features/article.phpr/3860131/Microsoft-Warns-About-17-year-old-Windows-Bug.htm`

July 2008 25 years old bug found in BSD (`seekdir()` wrongly implemented)
`http://www.vnode.ch/fixing_seekdir`

July 2008 33 years old bug found in Unix (buffer overflow in YACC)
`http://www.computerworld.com/s/article/9108978/Developer_fixes_33_year_old_Unix_bug`

# Introduction

## Bugs

- Bugs are inevitable in complex software system
- A bug can be very visible or can hide in your code until a much later date

## Bugs can hide very well

Aug 2009 8 years old bug found in Linux (handling not implemented kernel fctions)

`http://www.theregister.co.uk/2009/08/14/critical_linux_bug/`

Jan 2010 Microsoft fixes a 17 years old bug (in code allowing NT to run 16bits apps)

`http://www.esecurityplanet.com/features/article.phpr/3860131/Microsoft-Warns-About-17-year-old-Windows-Bug.htm`

July 2008 25 years old bug found in BSD (`seekdir()` wrongly implemented)

`http://www.vnode.ch/fixing_seekdir`

July 2008 33 years old bug found in Unix (buffer overflow in YACC)

`http://www.computerworld.com/s/article/9108978/Developer_fixes_33_year_old_Unix_bug`

## Chasing bugs

- Once identified, use print statements of IDE's debugger to hunt them down
- But how to discover all bugs in the system, even those with low visibility?

# Introduction

## Bugs

- Bugs are inevitable in complex software system
- A bug can be very visible or can hide in your code until a much later date

## Bugs can hide very well

Aug 2009 8 years old bug found in Linux (handling not implemented kernel fctions)
: http://www.theregister.co.uk/2009/08/14/critical_linux_bug/

Jan 2010 Microsoft fixes a 17 years old bug (in code allowing NT to run 16bits apps)
: http://www.esecurityplanet.com/features/article.phpr/3860131/Microsoft-Warns-About-17-year-old-Windows-Bug.htm

July 2008 25 years old bug found in BSD (`seekdir()` wrongly implemented)
: http://www.vnode.ch/fixing_seekdir

July 2008 33 years old bug found in Unix (buffer overflow in YACC)
: http://www.computerworld.com/s/article/9108978/Developer_fixes_33_year_old_Unix_bug

## Chasing bugs

- Once identified, use print statements of IDE's debugger to hunt them down
- But how to discover all bugs in the system, even those with low visibility?

$\Rightarrow$ Testing and Quality Assurance practices

# Why to test?

> *Testing can only prove the presence of bugs, not their absence.*
> – E. W. Dijkstra

## Perfect Excuse

- Don't invest in testing: system will contain defects anyway

## Counter Arguments

- The more you test, the less likely such defects will *cause harm*
- The more you test, the more *confidence* you will have in the system

# Who should Test?

## Fact: Programmers are not necessarily the best testers

- Programming is a constructive activity: try to make things work
- Testing is a destructive activity: try to make things fail

## In practice

- Best case: Testing is part of quality assurance
  - done by developers when finishing a component (unit tests)
  - done by a specialized test team when finishing a subsystem (integration tests)
- Common case: done by rookies
  - testing seen as a beginner's job, assigned to least experienced team members
  - testing often done after completion (if at all)
  - but very difficult task; impossible to completely test a system
- Worst case (unfortunately very common too): no one does it
  - Not productive $\Rightarrow$ not done [yet], postponed "by a while"
  - But without testing, productivity decreases, so less time, so less tests

> *Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.* – Kernighan

# What is “Correct”?

different meanings depending on the context

## Correctness

- A system is correct if it behaves according to its specification

⇒ An absolute property (i.e., a system cannot be "almost correct")

⇒ ... undecideable in theory and practice

## Reliability

- The user may rely on the system behaving properly
- Probability that the system will operate as expected over a specified interval

⇒ Relative property (system mean time between failure (MTTF): 3 weeks)

## Robustness

- System behaves reasonably even in circumstances that were not specified

⇒ Vague property (specifying abnormal circumstances $\rightsquigarrow$ part of the requirements)

# Terminology

## Avoid the term "Bug"

- Implies that mistakes somehow creep into the software from the outside
- Imprecise because mixes various "mistakes"

## Error: incorrect software behavior

- A deviation between the specification and the running system
- A manifestation of a defect during system execution
- Inability to perform required function within specified limits
- *Example*: message box text said "Welcome null."
- Transient error: only with certain inputs; Permanent error: for any input

## Fault: cause of error

- Design or coding mistake that may cause abnormal behavior
- *Example:* account name field is not set properly.
- A fault is not an error, but it can lead to them

## Failure: particular instance of a general error, caused by a fault

# Quality Control Techniques

Large systems bound to have faults. How to deal with that?

**Fault Avoidance:** Prevent errors by finding faults before the release

- Development methodologies:
  Use requirements and design to minimize introduction of faults
  Get clear requirements; Minimize coupling
- Configuration management: don't allow changes to subsystem interfaces
- [Formal] Verification: find faults in system execution
  Maturity issue; Assumes requirements, pre/postconditions are correct & adequate
- Review: manual inspection of system by team members
  shown effective at finding errors

**Fault Detection:** Find existing faults without recovering from the errors

- Manual tests: Use debugger to move through steps to reach erroneous state
- Automatic Testing: tries to expose errors in planned way

**Fault Tolerance:** When system can recover from failure by itself

- Recovery from failure (*example:* DB rollbacks, FS logs)
- Sub-system redundancy (*example:* disk RAID-1)

# Quality Control Techniques

Large systems bound to have faults. How to deal with that?

**Fault Avoidance:** Prevent errors by finding faults before the release

- Development methodologies:
  Use requirements and design to minimize introduction of faults
  Get clear requirements; Minimize coupling
- Configuration management: don't allow changes to subsystem interfaces
- [Formal] Verification: find faults in system execution
  Maturity issue; Assumes requirements, pre/postconditions are correct & adequate
- Review: manual inspection of system by team members
  shown effective at finding errors

**Fault Detection:** Find existing faults without recovering from the errors

- Manual tests: Use debugger to move through steps to reach erroneous state
- Automatic Testing: tries to expose errors in planned way ← **We are here**

**Fault Tolerance:** When system can recover from failure by itself

- Recovery from failure (*example:* DB rollbacks, FS logs)
- Sub-system redundancy (*example:* disk RAID-1)

# Testing Concepts

## Recapping generic terms

- Error: Incorrect software behavior
- Fault: Cause of the error (programming, design, etc)
- Failure: Particular instance of a general error, caused by a fault

## Component

- A part of the system that can be isolated for testing

⇒ an object, a group of objects, one or more subsystems

## Test Case

- {inputs; expected results} set exercising component to cause failures
- Boolean method: whether component's answer matches expected results
- "expected results" includes exceptions, error codes ...

## Test Stub

- Partial implementation of components on which the tested compnt depends
- dummy code providing necessary input values and behavior to run test cases

## Test Driver

- Partial implementation of a component that depends on the tested part
- a "main()" function that executes a number of test cases

# Testing Concepts

## Recapping generic terms

- Error: Incorrect software behavior
- Fault: Cause of the error (programming, design, etc)
- Failure: Particular instance of a general error, caused by a fault

## Component

- A part of the system that can be isolated for testing (through *stub* and *driver*)

$\Rightarrow$ an object, a group of objects, one or more subsystems

## Test Case

- {inputs; expected results} set exercising component to cause failures
- Boolean method: whether component's answer matches expected results
- "expected results" includes exceptions, error codes ...

## Test Stub

- Partial implementation of components on which the tested compnt depends
- dummy code providing necessary input values and behavior to run test cases

## Test Driver

- Partial implementation of a component that depends on the tested part
- a "main()" function that executes a number of test cases

# Tests Campaign Planing

## Goal

- Should *verify* the requirements (are we building the product right?)
- NOT *validate* the requirements (are we building the right product?)

## Definitions

- **Testing**: activity of executing a program with the intent of finding a defect
  $\Rightarrow$ A successful test is one that finds defects!
- **Testing Techniques**: Techniques to find yet undiscovered mistakes
  $\Rightarrow$ Criterion: Coverage of the system
- **Testing Strategies**: Plans telling *when* to perform *what* testing technique
  $\Rightarrow$ Criterion: Confidence that you can safely proceed with the next activity

# Fifth Chapter

## Testing


- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Testing in Practice
  - JUnit
  - Right BICEP + CORRECT
- Other Techniques for Practical Software Correctness
  - Design By Contract
  - Fuzzing
  - Formal Methods
- Conclusion

# White Box Testing

Focuses on internal states of objects

## Use internal knowledge of the component to craft input data

- *Example:* internal data structure = array of size 256
  $\Rightarrow$ test for size = 255 and 257 (near boundary)
- Internal structure include design specs (like diagram sequence)
- Derive test cases to maximize structure coverage, yet minimize # of test cases

## Coverage criteria: Path testing

- every statement at least once
- all portions of control flow (= branches) at least once
- all possible values of compound conditions at least once (condition coverage)

- all portions of data flow at least once
- all loops, iterated at least 0, once, and N times (loop testing)

# White Box Testing

Focuses on internal states of objects

## Use internal knowledge of the component to craft input data

- *Example:* internal data structure = array of size 256
  $\Rightarrow$ test for size = 255 and 257 (near boundary)
- Internal structure include design specs (like diagram sequence)
- Derive test cases to maximize structure coverage, yet minimize # of test cases

## Coverage criteria: Path testing

- every statement at least once
- all portions of control flow (= branches) at least once
- all possible values of compound conditions at least once (condition coverage)
  Multiple condition coverage $\rightsquigarrow$ all true/false combinations for all simple conditions
  Domain testing $\rightsquigarrow$ {a < b; a == b; a > b}
- all portions of data flow at least once
- all loops, iterated at least 0, once, and N times (loop testing)

# White Box Testing

Focuses on internal states of objects

## Use internal knowledge of the component to craft input data

- *Example:* internal data structure = array of size 256
  $\Rightarrow$ test for size = 255 and 257 (near boundary)
- Internal structure include design specs (like diagram sequence)
- Derive test cases to maximize structure coverage, yet minimize # of test cases

## Coverage criteria: Path testing

- every statement at least once
- all portions of control flow (= branches) at least once
- all possible values of compound conditions at least once (condition coverage)
  Multiple condition coverage $\rightsquigarrow$ all true/false combinations for all simple conditions
  Domain testing $\rightsquigarrow$ {a < b; a == b; a > b}
- all portions of data flow at least once
- all loops, iterated at least 0, once, and N times (loop testing)

Main issue: white box testing negates object encapsulation

# Black Box Testing

Component $\equiv$ "black box"

## Test cases derived from external specification

- Behavior only determined by studying inputs and outputs
- Derive tests to maximize coverage of spec elements yet minimizing # of tests

## Coverage criteria

- All exceptions
- All data ranges (incl. invalid input) generating different classes of output
- All boundary values

## Equivalence Partitioning

- For each input value, divide value domain in classes of equivalences:
  - Expects value within [0, 12] $\leadsto$ negative value, within range, above range
  - Expects fixed value $\leadsto$ below that value, expected, above
  - Expects value boolean $\leadsto$ {true, false}
- Pick a value in each equivalence class (randomly or at boundary)
- Predict output, derive test case

# Fifth Chapter

## Testing

- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Testing in Practice
  - JUnit
  - Right BICEP + CORRECT
- Other Techniques for Practical Software Correctness
  - Design By Contract
  - Fuzzing
  - Formal Methods
- Conclusion

# Testing Strategies

## Unit testing

$\leadsto$ Looks for errors in objects or subsystems

## Integration testing

$\leadsto$ Find errors with connecting subsystems together

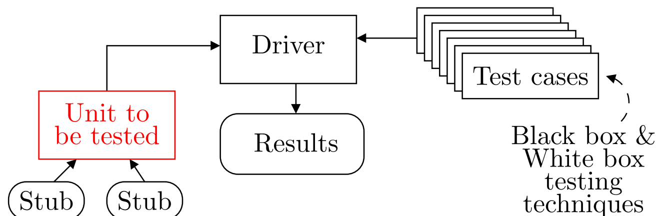
- System structure testing: integration testing all parts of system together

## System testing

$\leadsto$ Test entire system behavior as a whole, wrt use cases and requirements

- functional testing: test whether system meets requirements
- performance testing: nonfunctional requirements, design goals
- acceptance testing: done by client

# Unit Testing




## Why?

- Locate small errors (= within a unit) fast

## Who?

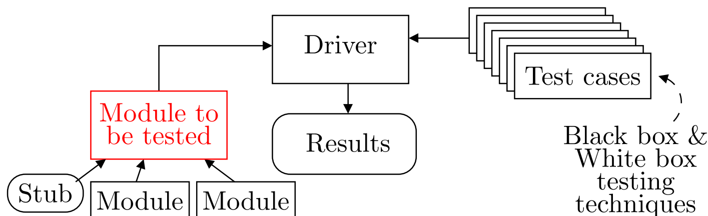
- Person developing the unit writes the tests

## When?

- At the latest when a unit is delivered to the rest of the team
  - No test ⇒ no unit
- Write the test first, i.e. before writing the unit

  ⇒ help to design the interface right

# Integration Testing




## Why?

- Sum is more than parts, interface may contain faults too

## Who?

- Person developing the module writes the tests

## When?

- Top-down: main module before constituting modules
- Bottom-up: constituting modules before main module
- In practice: a bit of both

Remark: Distinction between unit testing and integration testing not that sharp

# Regression Testing

Ensure that things that used to work still work after changes

## Regression test

- Re-execution of tests to ensure that changes have no unintended side effects
- Tests must avoid regression (= degradation of results)
- Regression tests must be repeated *often*
  (after every change, every night, with each new unit, with each fix,...)
- Regression tests may be conducted manually
  - Execution of crucial scenarios with verification of results
  - Manual test process is slow and cumbersome
    $\Rightarrow$ preferably completely automated

## Advantages

- Helps during iterative and incremental development + during maintenance

## Disadvantage

- Up front investment in maintainability is difficult to sell to the customer
- Takes a lot of work: more test code than production code

# Acceptance Testing

## Acceptance Tests

- conducted by the end-user (representatives)
- check whether requirements are correctly implemented
  borderline between verification ("Are we building the system right?")
  and validation ("Are we building the right system?")

## Alpha- & Beta Tests

- Acceptance tests for "off-the-shelves" software (many unidentified users)
- Alpha Testing
  - end-users are invited at the developer's site
  - testing is done in a controlled environment
- Beta Testing
  - software is released to selected customers
  - testing is done in "real world" setting, without developers present

# Other Testing Strategies

## Recovery Testing

- Forces system to fail and checks whether it recovers properly

⇝ For fault tolerant systems

## Stress Testing (Overload Testing): Tests extreme conditions

- e.g., supply input data twice as fast and check whether system fails

## Performance Testing: Tests run-time performance of system

- e.g., time consumption, memory consumption
- first do it, then do it right, then do it fast

## Back-to-Back Testing

- Compare test results from two different versions of the system

⇝ requires N-version programming or prototypes
`git` version control system does so to isolate regressions (`bisect` command)

# Fifth Chapter

## Testing


- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Testing in Practice
  - JUnit
  - Right BICEP + CORRECT
- Other Techniques for Practical Software Correctness
  - Design By Contract
  - Fuzzing
  - Formal Methods
- Conclusion

# Tool support

## Test Harness

- Framework merging all test code in environment
- Main example for Java is called JUnit
- It inspired CppUnit, PyUnit, ...

## Test Verifiers

- Measure test coverage for a set of test cases
- JCov for Java, gcov for gcc, ...

## Test Data Generators

- Assist in selecting test data
- Based on the formal specification such as JML

# Fifth Chapter

## Testing

- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Testing in Practice
  - JUnit
  - Right BICEP + CORRECT
- Other Techniques for Practical Software Correctness
  - Design By Contract
  - Fuzzing
  - Formal Methods
- Conclusion

# Introduction

## What is JUnit?

- It is a unit testing framework for Java.
- It provides tools for easy implementation of unit test plans
- It eases execution of tests
- It provides reports of test executions

## What is NOT JUnit?

- It cannot design your test plan
- It does only what you tell it to
- It does not fix bugs for you

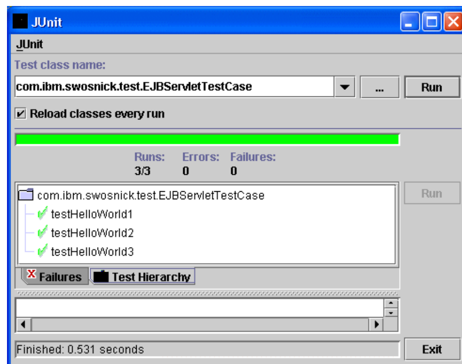
## JUnit has two major versions

- JUnit 3.x: uses convention on method naming
- JUnit 4.x: uses Java 5 annotations

# Structure of JUnit tests

Running a test suite consists of

- Setting up test environment
- For each test
  - Setting test up
  - Invoking test function
  - Tearing test down
- Tearing down everything
- Report result



# Setting up test environment

## Purpose

- Get things ready for testing.
- Create common instances, variables and data to use in tests.

## Two kinds may co-exist

- Setting up before each test function
  - Named `public void setUp()` in JUnit 3.x
  - Annotated `@Before` in JUnit 4.x
- Setting up once for all
  - Placed in constructor in JUnit 3.x
  - Annotated `@BeforeClass` in JUnit 4.x

# Cleaning test environment: Tearing down methods

## Purpose

- Clean up after testing
- I.e., closing any files or connexions, etc
- Not used as often as setup methods

## Two kinds may co-exist

- Tearing down after each test function
  - Named `public void tearDown()` in JUnit 3.x
  - Annotated `@After` in JUnit 4.x
- Tearing down once for all (JUnit 4.x only)
  - Annotated `@AfterClass`

# Actually doing the tests

## Test functions

- It is where the tests are performed
- Need one function per test case (which may call helper functions)
- Name must start with `test` in JUnit 3.x
- Annotated `@Test` (in JUnit 4.x)

## Verifying results

- All tests are verified with assertions.
- JUnit comes with an Assert class for this purpose
  - public void assertTrue(String message, boolean condition)
  - public void assertNotNull(String message, Object obj)
  - public void assertEquals(String message, Object expected, Object actual)
  - public void assertSame(String message, Object expected, Object actual)
    uses ==, not .equals()
  - public void assertFalse(String message, boolean condition)
  - public void assertNotEquals(String message, Object expected, Object actual)
  - public void assertNotSame(String message, Object expected, Object actual)
  - public void fail(String message)

# Example: Combination Lock (1/2)

## Data and setting up

```
public class CombinationLockTest {
    // Locks with the specified combinations
    private CombinationLock lock00; // comb. 00
    private CombinationLock lock03; // comb. 03
    private CombinationLock lock12; // comb. 12
    private CombinationLock lock99; // comb. 99
    @Before
    public void setUp () {
        lock00 = new CombinationLock(0);
        lock03 = new CombinationLock(3);
        lock12 = new CombinationLock(12);
        lock99 = new CombinationLock(99);
    }
    ...
}
```

## Tear down not necessary here

- object data will be deallocated automatically
- setup method overwrites instance variables

# Example: Combination Lock (2/2)

## Simple test method

```
@Test
public void testOpenLock () {
    lock12.enter(3);
    lock12.enter(4);
    assertTrue(lock12.isOpen());
}
```

## Test method with helper

```
@Test
public void testFirstDigitTwice () {
    closeLocks();
    firstDigitTwice(lock03,0,3);
    firstDigitTwice(lock12,1,2);
}
private void firstDigitTwice(CombinationLock lock, int first, int second) {
    lock.enter(first);
    lock.enter(first);
    assertFalse(lock.isOpen());
    lock.enter(second);
    assertTrue(lock.isOpen());
}
```

# Going further with JUnit: TDD

## Test-Driven Development

- That's a methodology to write code
- Aims at ease/productivity + code quality

## Principle: Write the Test Cases First (before the code)

- Ensures that the codes actually get written
- Improves the interface: you're user of your own code before coding it

## That's easy and pleasant to do

- It's one of the "agile development methodologies", very light-weighted
  More than just TDD in agile methods (but too long to say it all here)
- Eclipse correction suggestion and ability to generate stubs very helpful
- Try it for your next project

# Right BICEPS

## Thinking of all mandatory test cases is difficult

- I.e., challenging to discover all the ways a code might fail
- Good news: Experience quickly gives a feel for what is likely to fail

## Beginners need a bit of help (until they get experienced)

- Guidelines on what can fail
- Reminders of areas that are important to test
- These guidelines are not very complex, but quite useful/powerful
- See Software Systems and Architecture [Scott Miller] for details

# Right-BICEP

## Guidelines in a Nutshell

- Right: Are the results right?
- B: Are all the boundary conditions CORRECT?
- I: Can you check inverse relationships?
- C: Can you cross-check results using other means?
- E: Can you force error conditions to happen?
- P: Are performance characteristics within bounds?

## Right?

- We need to compute what the correct result should be to test
- Quite often these can be inferred from the specification
- If the "right" results cannot be determined... you shouldn't be writing code!
- If spec not completed [by client], assume what's correct, and fix afterward

# B: Boundary Tests (1/3)

## Discovering boundary conditions is crucial!

- This is where most of the bugs generally live
- These are also the "edges" of our code

## Remember our little experience

- We had to refine it several time our specifications
  - Triangle with negative length
  - Sort an empty array
- The algorithm in exercise 3 of proof lab were false
  - Failed to find smallest value if at the end of the array

# B: Boundary Tests (2/3)

Example of boundary conditions

- Totally bogus, inconsistent input values: filename of "#()*%)Q*#%&@"
- Badly formatted data: e-mail address without TLD (zastre@foo)
- Empty or missing values: 0, 0.0, "", null
- Values above some reasonable expectation: age of 10,000; #children == 30
- Duplicates in lists meant to be free of duplicates
- Ordered lists that are not ordered
  Also: Presorted lists passed to sort algorithms? reverse-sorted?
- Things which arrive out of order? or out of expected order?

# B: Boundary Tests (3/3)

Another guideline for boundaries: CORRECT

- Conformance: Does the value conform to an expected value?
- Ordering: Is the set of values ordered or unordered as appropriate?
- Range: Is the value within reasonable minimum and maximum values?
- Reference: Does the code reference anything external that isn't under control?
- Existence: Does the value exist? (e.g., is non-null, non- zero, present in a set)
- Cardinality: Are there exactly enough values?
- Time: Is everything happening in order? At the right time? In time?

# I: Check Inverse Relationships

## Some methods can be checked almost trivially

- Data inserted in table should appear in a search immediately afterwards
- Lossless compression algorithm $\leadsto$ data uncompressed to the original value
- Check square-root calculation by squaring result (ensure it is "close enough")

## Inverse Gotchas

- You usually write the function/method and its inverse
- What if both are buggy? errors gets be masked
- Ideally, the inverse function is written by somebody else
  - Square root example: use built-in multiplication
  - Database insert: vendor-provided search routine to test insert

# C: Cross-check Using Other Means

## Idea 1: For methods without an inverse

- Use a different algorithm to compute the result, and compare to yours
- For example, use a $O(n^2)$ sorting algorithm to check your $O(n \times log(n))$ one

## Idea 2: Ensure that different pieces of the class's data "add up"

- Example: library system with book, copies
- "books out" + "books in library" should equal "total copies"

# E: Force Error Conditions

## Production code defensive to system failures

- Disks: fill up
- Networks: go down
- E-mail: gets lost
- Programs: crash

## This also should to be tested

- Easy: invalid parameters
- Harder: environmental errors

## Environmental errors/constraints

- Out of memory; Out of disk space
- Network availability and errors
- System load
- Limited colour palette; Very high or very low video resolution
- ...

# P: Performance Characteristics

## What is the time performance as:

- Input size grows?
- Problem sets become more complex

## Idea: "regression test" on performance characteristics

- Ensure that version N+1 is not awfully slower than version N

## Very hard to ensure

- Bad performance can come from external factors
- Performance not portable from machine to machine
- (even harder to ensure *automatically*)

# When to stop writing tests?

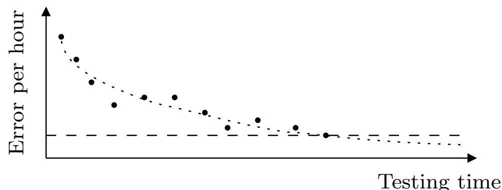
*Testing can only prove the presence of defects, not their absence.*
– E. W. Dijkstra

## Cynical answer (sad but true)

- You're never done: each run of the system is a new test
  ⇒ Each bug-fix should be accompanied by a new regression test
- You're done when you are out of time/money
  - Include test in project plan and **do not give in to pressure**
  - ... in the long run, tests **save** time

## Statistical testing

- Test until you've reduced failure rate under risk threshold



# Why to test? (continued)

Because it helps ensuring that the system matches its specification

But not only (more good reason to test)

- Traceability
  - Tests helps tracing back from components to the requirements that caused their presence
- Maintainability
  - Regression tests verify that post-delivery changes do not break anything
- Understandability
  - Newcomers to the system can read the test code to understand what it does
  - Writing tests first encourage to make the interface really useable

# Fifth Chapter

## Testing

- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Testing in Practice
  - JUnit
  - Right BICEP + CORRECT
- Other Techniques for Practical Software Correctness
  - Design By Contract
  - Fuzzing
  - Formal Methods
- Conclusion

# Introduction

## Design by Contract

- Programming methodology trying to prevent code to diverge from specs
- Mistakes are possible
  - while transforming requirements into a system
  - while system is changed during maintenance

# Introduction

## Design by Contract

- Programming methodology trying to prevent code to diverge from specs
- Mistakes are possible
  - while transforming requirements into a system
  - while system is changed during maintenance

## What's the difference with Testing?

- Testing tries to diagnose (and cure) errors after the facts
- Design by Contract tries to prevent certain types of errors

# Introduction

## Design by Contract

- Programming methodology trying to prevent code to diverge from specs
- Mistakes are possible
  - while transforming requirements into a system
  - while system is changed during maintenance

## What's the difference with Testing?

- Testing tries to diagnose (and cure) errors after the facts
- Design by Contract tries to prevent certain types of errors

## Design by Contract is particularly useful in an Object-Oriented context

- preventing errors in interfaces between classes
  incl. subclass and superclass via subcontracting
- preventing errors while reusing classes
  incl. evolving systems, thus incremental and iterative development
  Example of the Ariane 5 crash

# Introduction

## Design by Contract

- Programming methodology trying to prevent code to diverge from specs
- Mistakes are possible
  - while transforming requirements into a system
  - while system is changed during maintenance

## What's the difference with Testing?

- Testing tries to diagnose (and cure) errors after the facts
- Design by Contract tries to prevent certain types of errors

## Design by Contract is particularly useful in an Object-Oriented context

- preventing errors in interfaces between classes
  incl. subclass and superclass via subcontracting
- preventing errors while reusing classes
  incl. evolving systems, thus incremental and iterative development
  Example of the Ariane 5 crash

Use Design by Contract in combination with Testing!

# What is Design By Contract?

> *View the relationship between two classes as a formal agreement, expressing each party's rights and obligations.* – Bertrand Meyer

## Example: Airline Reservation

| | Obligations | Rights |
|---|---|---|
| Customer | ▶ Be at Paris airport at least 3 hour before scheduled departure time<br>▶ Bring acceptable baggage<br>▶ Pay ticket price | ▶ Reach Los Angeles |
| Airline | ▶ Bring customer to Los Angeles | ▶ No need to carry passenger who is late<br>▶ has unacceptable baggage<br>▶ or has not paid ticket |

- Each party expects benefits (rights) and accepts obligations
- Usually, one party's benefits are the other party's obligations
- Contract is declarative: it is described so that both parties can understand *what* service will be guaranteed without saying *how*

# Connecting back to Hoare logic

## Pre- and Post-conditions + Invariants

- Obligations are expressed via pre- and post-conditions

> *If you promise to call me with the precondition satisfied, then I, in return promise to deliver a final state in which the postcondition is satisfied.*

pre-condition: $x >= 9$ post-condition: $x >= 13$

component: x := x + 5

- and invariants

> *For all calls you make to me, I will make sure the invariant remains satisfied.*

## Isn't this pure documentation?

(a) Who will register these contracts for later reference (the notary)?

(b) Who will verify that the parties satisfy their contracts (the lawyers)?

# Connecting back to Hoare logic

## Pre- and Post-conditions + Invariants

- Obligations are expressed via pre- and post-conditions

> *If you promise to call me with the precondition satisfied, then I, in return promise to deliver a final state in which the postcondition is satisfied.*

pre-condition: $x >= 9$ post-condition: $x >= 13$

component: x := x + 5

- and invariants

> *For all calls you make to me, I will make sure the invariant remains satisfied.*

## Isn't this pure documentation?

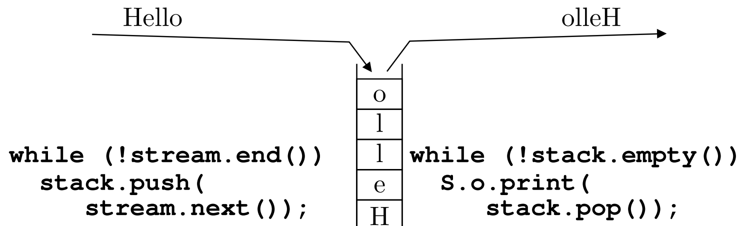
(a) Who will register these contracts for later reference (the notary)?
The source code

(b) Who will verify that the parties satisfy their contracts (the lawyers)?
The running system

# Example: Stack

## Specification

- Given
  - A stream of characters, length unknown
- Requested
  - Produce a stream containing the same characters but in reverse order
  - Specify the necessary intermediate abstract data structure



# Example: Stack Specification

```
class stack
  invariant: (isEmpty (this)) or (! isEmpty (this))
  /* Implementors promise that invariant holds after all methods return
     (incl. constructors)*/
  public char pop ()
     require: !isEmpty(this)              /* Clients' promise (precondition) */
     ensure: true                         /* Implementors' promise (postcondition)
                                             Here: nothing */

  public void push(char)
     require: true
     ensure: (!isEmpty(this))             /* Implementors' promise:
              and (top(this)==char)          Matches specification */
  public void top (char) : char
     require: ...                         /* left as an exercise */
     ensure: ...
  public void isEmpty() : boolean
     require: ...
     ensure: ...
```

# Defensive Programming

## Redundant checks

- Redundant checks are the naive way for including contracts in the source code

```
public char pop () {
  if (isEmpty (this)) {
    ... //Error-handling
  } else {
    ...}
```

This is redundant code: it is the responsibility of the client to ensure the pre-condition!

## Redundant Checks Considered Harmful

- Extra complexity

  due to extra (possibly duplicated) code ... which must be verified as well
- Performance penalty

  Redundant checks cost extra execution time
- Wrong context
  - How severe is the fault? How to rectify the situation?
  - A service provider cannot asses the situation, only the consumer can.
  - Again: What happens if the precondition is not satisfied?

# Assertions

Any boolean expression we expect to be true at some point

## Advantages

- Help in writing correct software (formalizing invariants, and pre/post-conditions)
- Aid in maintenance of documentation (specifying contracts in the source code)
  $\Rightarrow$ tools to extract interfaces and contracts from source code
- Serve as test coverage criterion (Generate test cases that falsify assertions)
- Should be configured at compile-time (to avoid performance penalties in prod)

## What happens if the precondition is not satisfied?

- When an assertion does not hold, throw an exception

# Assertions in Programming Languages

## Eiffel

- Eiffel is designed as such ... but only used for correction (not documentation)

## C++

- assert.h does not throw an exception, but close program
- Possible to mimick. Documentation extraction rather difficult

## Smalltalk

- Easy to mimic, but compilation option requires some language idioms
- Documentation extraction is possible (style JavaDoc)

## Java

- Assert is standard since Java 1.4 ... very limited
- JML provide a mechanism ... but not ported to Java 5 (damn genericity)
- Modern Jass seems very promising, but needs more polishing

# Design by Contract vs. Testing

## They serve the same purpose

- Design by contract *prevents* errors; Testing *detect* errors

$\rightsquigarrow$ One of them should be sufficient!

## They are complementary

None of the two guarantee correctness ... but the sum is more than the parts

- Testing detects wide range of coding mistakes
  ... design by contract prevents specific mistakes due to incorrect assumption
- Design by contract ease black box testing by formalizing spec
- Condition testing verify whether all assertions are satisfied
  (whether parties satisfy their obligations)

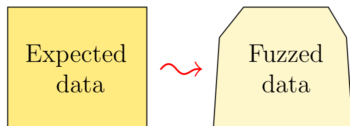
# Fifth Chapter

## Testing

- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Testing in Practice
  - JUnit
  - Right BICEP + CORRECT
- Other Techniques for Practical Software Correctness
  - Design By Contract
  - Fuzzing
  - Formal Methods
- Conclusion

# Fuzzing big picture

1. Intercept the data an application gets from its environment
2. Fuzz it, i.e. provide data vaguely resembling to expected one
   (sort of model-checking, exploring only execution paths close to the usual one)




## Why? Motivation

- It's a security assessment method:
  if you can get the application to segfault, it must be a buffer overflow to exploit
- Easy to write some tests w/o system knowledge
  ⇒ launch a fuzzer when arriving in company, it may find something interesting
- One of the best price/quality ratio

## Target applications

- Classically, network protocols; Recently used on media files

# How to actually fuzz the data?

## Brute force: random

- Pick a byte in the stream, and change its value
- ☺ Really easy to do
- ☹ Get easily caught by checksums
- ☹ "Interesting changes" are rather improbable

## Refined manner: templating

- Understand the logic of the stream
- ☺ Pass checksums and easy validity protection levels
- ☹ Rather hard to do
- ☹ Fuzzing process becomes stream-dependent
  no longer first tool to use: you need to understand stream first

## Some research leads

- Stateful fuzzing: Build a protocol automaton, test invalid transitions
- Automatic templating: Extend `file` format metagrammar,use almost valid data

# Fifth Chapter

## Testing


- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Testing in Practice
  - JUnit
  - Right BICEP + CORRECT
- Other Techniques for Practical Software Correctness
  - Design By Contract
  - Fuzzing
  - Formal Methods
- Conclusion

# Formal Methods

Goal: Develop safe software using automated methods

- Strong mathematical background

# Formal Methods

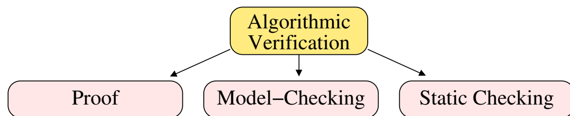
Goal: Develop safe software using automated methods

- Strong mathematical background
- Safe $\equiv$ respect some given properties

## Kind of properties shown

- Safety: the car does not start without the key
- Liveness: if I push the break paddle, the car will eventually stop

# Existing Formal Methods




## Proof of programs

- In theory, applicable to any class of program
- In practice, quite tedious to use
  often limited to help a specialist doing the actual work (system state explosion)

## Model-checking

- Goal: Shows that a system:

(safety) never evolves to a faulty state from a given initial state
(liveness) always evolve to the wanted state (stopping) from a given state (breaking)

☹ Less generic than proof: lack of faulty states for all initial state?
☺ Usable by non-specialists (at least, by *less-specialists*)

## Static Checking

- Automatic analyse of the source code (data flow analysis; theorem proving)
- Completely automated, you should use these tools, even if partial coverage

# Example of problem to detect: Race Condition

$x$ is a shared variable; *Alice* adds 2, *Bob* adds 5; Correct result : $x = 7$

a. Read the value of shared variable $x$ and store it locally
b. Modify the local value (add 5 or 2)
c. Propagate the local variable into the shared one

# Example of problem to detect: Race Condition

$x$ is a shared variable; *Alice* adds 2, *Bob* adds 5; Correct result : $x = 7$

a. Read the value of shared variable $x$ and store it locally
b. Modify the local value (add 5 or 2)
c. Propagate the local variable into the shared one

- Execution of *Alice* **then** *Bob* or opposite: result $= 7$
- Interleaved execution: result $= 2$ or $5$ (depending on last propagator)

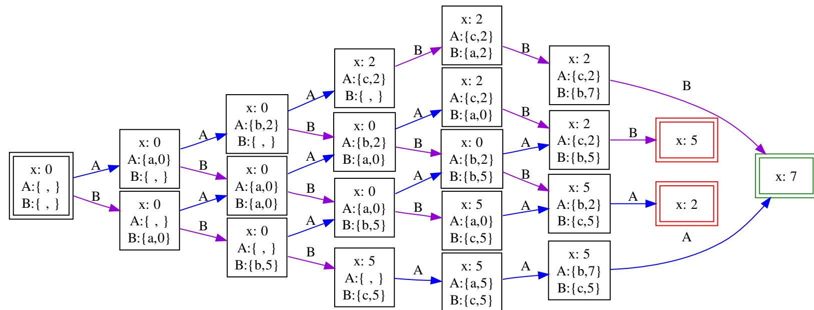
# Example of problem to detect: Race Condition

$x$ is a shared variable; *Alice* adds 2, *Bob* adds 5; Correct result : $x = 7$

a. Read the value of shared variable $x$ and store it locally
b. Modify the local value (add 5 or 2)
c. Propagate the local variable into the shared one

- Execution of *Alice* **then** *Bob* or opposite: result $= 7$
- Interleaved execution: result $= 2$ or $5$ (depending on last propagator)

**Model-checking**: traverse graph of executions checking for properties



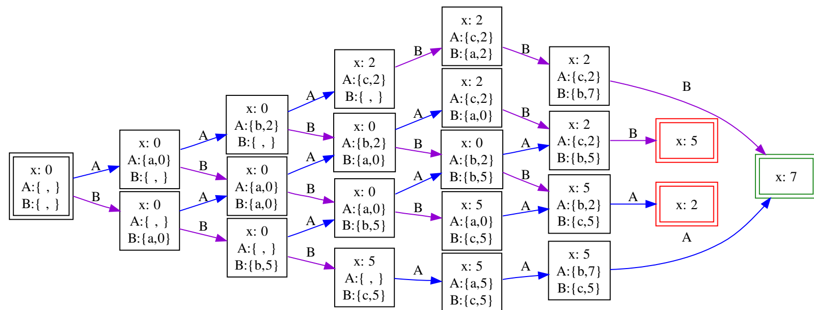
# Example of problem to detect: Race Condition

$x$ is a shared variable; *Alice* adds 2, *Bob* adds 5; Correct result : $x = 7$

a. Read the value of shared variable $x$ and store it locally
b. Modify the local value (add 5 or 2)
c. Propagate the local variable into the shared one

- Execution of *Alice* **then** *Bob* or opposite: result $= 7$
- Interleaved execution: result $= 2$ or $5$ (depending on last propagator)

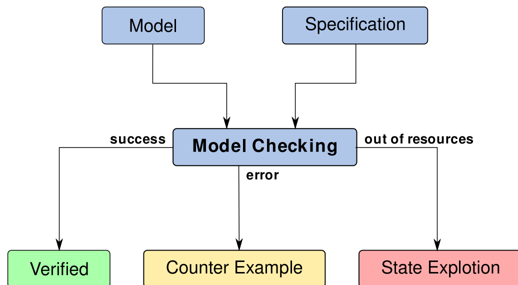
**Model-checking**: traverse graph of executions checking for properties




- Safety: assertions on each node
- Liveness by studying graph (cycle?)

# Model-Checking Big Picture

1. User writes Model (formal writing of algorithm) and Specification (set of properties)
2. Each decision point in model (if, input data) $\leadsto$ a branch in model state space
3. Check safety properties on each encountered node (state)
4. Store encountered nodes (to avoid looping) and transitions (to check liveness)
5. Process until:
   - State space completely traversed ($\Rightarrow$ model verified against this specification)
   - One of the property does not hold (the path until here is a counter-example)
   - We run out of resource ("state space explosion")



# Fifth Chapter

# Testing


- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Testing in Practice
  - JUnit
  - Right BICEP + CORRECT
- Other Techniques for Practical Software Correctness
  - Design By Contract
  - Fuzzing
  - Formal Methods
- Conclusion

# A Comparison of Bug Finding Tools for Java

Rutar, Almazan, Foster, U. Maryland – ISSRE04

Example code

```
1  import java.io.*;
2  public class Foo{
3   private byte[] b;
4   private int length;
5   Foo(){ length = 40;
6    b = new byte[length]; }
7   public void bar(){
8    int y;
9    try {
10    FileInputStream x =
11     new FileInputStream("z");
12    x.read(b,0,length);
13    x.close();}
14   catch(Exception e){
15    System.out.println("Oopsie");}
16   for(int i=1; i<=length; i++){
17    if (Integer.toString(50) ==
18              Byte.toString(b[i]))
19    System.out.print(b[i] + " ");
20   }
21  }
22 }
```

## Defects of this code

# A Comparison of Bug Finding Tools for Java

Rutar, Almazan, Foster, U. Maryland – ISSRE04

Example code

```
1  import java.io.*;
2  public class Foo{
3   private byte[] b;
4   private int length;
5   Foo(){ length = 40;
6    b = new byte[length]; }
7   public void bar(){
8    int y;
9    try {
10    FileInputStream x =
11     new FileInputStream("z");
12    x.read(b,0,length);
13    x.close();}
14   catch(Exception e){
15    System.out.println("Oopsie");}
16   for(int i=1; i<=length; i++){
17    if (Integer.toString(50) ==
18              Byte.toString(b[i]))
19    System.out.print(b[i] + " ");
20   }
21  }
22 }
```

## Defects of this code

- l8 W: Unused variable
- l12 W: Return of read() ignored
- l14 W: Stream may not be closed
- l17 E: == to compare strings
- l18 E: may acces out of bound
- l18 FP: possible null-dereference to b

# A Comparison of Bug Finding Tools for Java

Rutar, Almazan, Foster, U. Maryland – ISSRE04

Example code

```
 1 import java.io.*;
 2 public class Foo{
 3  private byte[] b;
 4  private int length;
 5  Foo(){ length = 40;
 6   b = new byte[length]; }
 7  public void bar(){
 8   int y;
 9   try {
10    FileInputStream x =
11     new FileInputStream("z");
12    x.read(b,0,length);
13    x.close();}
14   catch(Exception e){
15    System.out.println("Oopsie");}
16   for(int i=1; i<=length; i++){
17    if (Integer.toString(50) ==
18              Byte.toString(b[i]))
19    System.out.print(b[i] + " ");
20   }
21  }
22 }
```

## Defects of this code

- l8 W: Unused variable
- l12 W: Return of read() ignored
- l14 W: Stream may not be closed
- l17 E: == to compare strings
- l18 E: may acces out of bound
- l18 FP: possible null-dereference to b

## Some tools for Java

| Name | Input | Interface | Technology |
|---|---|---|---|
| Bandera | Source | CL, GUI | Model checking |
| ESC/Java | Source+spec | CL, GUI | Theorem proving |
| FindBugs | Bytecode | CL,GUI,Ant,IDE | Syntax, data-flow |
| JLint | Bytecode | CL | Syntax, data-flow |
| PMD | Source | CL,GUI,Ant,IDE | Syntax |

# A Comparison of Bug Finding Tools for Java

Rutar, Almazan, Foster, U. Maryland – ISSRE04

Example code

```
1  import java.io.*;
2  public class Foo{
3   private byte[] b;
4   private int length;
5   Foo(){ length = 40;
6    b = new byte[length]; }
7   public void bar(){
8    int y;
9    try {
10    FileInputStream x =
11     new FileInputStream("z");
12    x.read(b,0,length);
13    x.close();}
14   catch(Exception e){
15    System.out.println("Oopsie");}
16   for(int i=1; i<=length; i++){
17    if (Integer.toString(50) ==
18              Byte.toString(b[i]))
19    System.out.print(b[i] + " ");
20   }
21  }
22 }
```

## Defects of this code

- l8 W: Unused variable
- l12 W: Return of read() ignored
- l14 W: Stream may not be closed
- l17 E: == to compare strings
- l18 E: may acces out of bound
- l18 FP: possible null-dereference to b

## Some tools for Java

| Name | Input | Interface | Technology |
|---|---|---|---|
| Bandera | Source | CL, GUI | Model checking |
| ESC/Java | Source+spec | CL, GUI | Theorem proving |
| FindBugs | Bytecode | CL,GUI,Ant,IDE | Syntax, data-flow |
| JLint | Bytecode | CL | Syntax, data-flow |
| PMD | Source | CL,GUI,Ant,IDE | Syntax |

## No tool is perfect/sufficient

- All detect something
- Some give false positive (b inited in ctor)
- All have false negative

# Conclusion

## Failure is not an option. It comes bundled with software.

- Testing searches for defects
  But not that easy and endless?
- Proof tries to ensure that there is no defect
  But quite heavy-weighted
- Automatic tools (static checking, theorem provers) may help
  But none is enough/sufficient (false negatives); all have false positives
- Design by Contract constitutes a global (methodological) answer
  Too bad that nobody use it / that Java offers no tool support for it (yet?)

## Optimistic Last Note

- That's a hot research topic, things move fast
- Tools improve quickly, you really should learn to use them
- Methodologies exist (DBC, TDD), you should try to follow them

# Bibliography for this chapter (and previous one)


## Lectures

- Testing, Debugging, and Verification (W. Ahrendt, R. Hähnle – U. Göteborgs)
  www.cse.chalmers.se/edu/year/2009/course/TDA566
- Software Engineering (Serge Demeyer – U. Antwerpen)
  http://www.lore.ua.ac.be/Teaching/SE3BAC/
- Software Systems and Architecture (Scott Miller – U. of Victoria)
  http://samiller.ece.uvic.ca/courses/SENG271/2009/05/
- JUnit (Dirk Hasselbalch – U. Copenhagen)
  http://isis.ku.dk/kurser/blob.aspx?feltid=217458

## Other

- Test Infected: Programmers Love Writing Tests (Tutorial on JUnit)
  http://www.cril.univ-artois.fr/~leberre/MI32001/TESTING/junit3.7/doc/testinfected/ntesting_fr.htm

# Sixth Chapter

# Conclusion on TOP

# Lectures content

## What we saw

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. Software Correction
   - Introduction; Specifying Systems; Hoare Logic; Proving Recursive Functions.
5. Testing Software
   - Testing techniques; Testing strategies; JUnit; Design By Contract.

## What's missing

- A LOT! This module is an initiation on several domains
- End of CS: complexity (P vs. NP), data structures
- End of SE: Programming methodo, Refactoring and automated code handling
- Tools: Practical Performance Assesement (a bit in labs), VCS, bug trackers

# 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
- 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)

# Who you want to be?

## 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