Techniques and tOols for Programming (TOP)

Martin Quinson <martin.quinson@loria.fr>

École Supérieure d'Informatique et Applications de Lorraine – 1re année

2008-2009

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 Martin Quinson): Concepts introduction
- 10 two-hours exercise session (TD, with staff member[1]): Theoretical exercises
- 6 two-hours labs (TP, with staff member[1]): Coding exercises
- Homework: Systematically finish the in-class exercises

Evaluation

- Two hours table exam
- Quiz at the beginning of each lab
- Maybe an evaluated lab (TP noté) at the end

[1] Martin Quinson, Gérald Oster, Thomas Pietrzak or Rémi Badonnel.

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

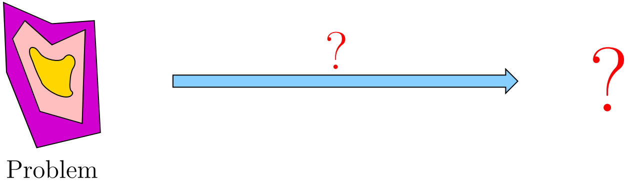
- IUT Orsay (in french): `http://www.iut-orsay.fr/~balkansk/`

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

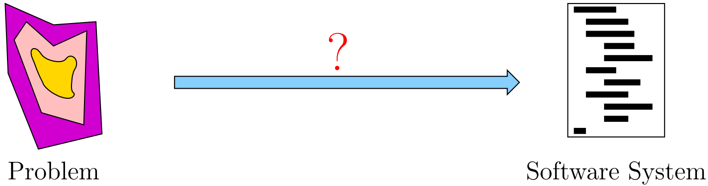
  Example: Determine the minimal value of a set of integers

Instances of a problem

- The problem for a given data set

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

Problems and Programs



Software systems (*ie.*, Programs)

- Describes a set of actions to be achieved in a given order
- Understandable (and doable) by computers

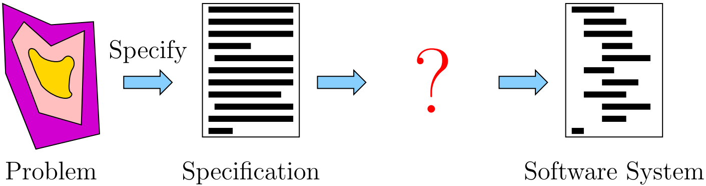
Problem Specification

- 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

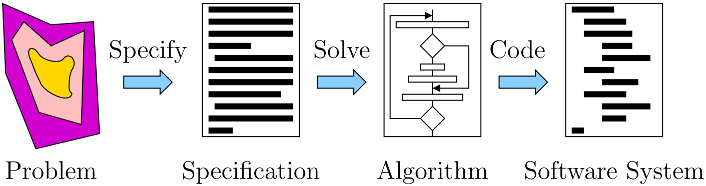
Divide, Conquer and Glue (top-down approach)

- 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?
  - Are there problems for which no algorithm exist?
- Complexity
  - How long does my algorithm need to reach the result?
  - 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?

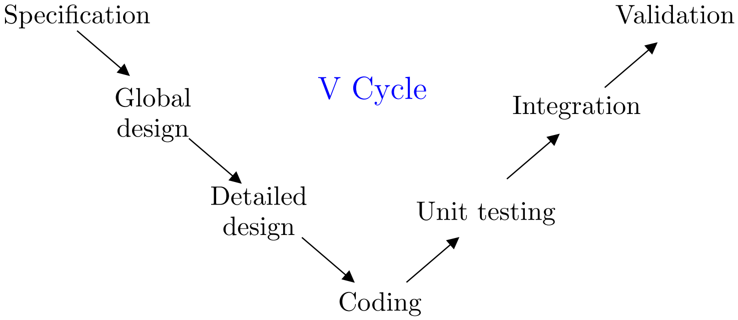
Software Engineering vs. Computer Science

Producing technical answers to consumers' needs

Software Engineering Definition

- Study of methods for producing and evaluating software

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)

Two approaches of the same issues

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

First Chapter

Practical and Theoretical Foundations of Programming

There are always several ways to solve a problem

Choice criteria between algorithms

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

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
  - Example: my text editor is v22.1 (hundreds of versions for "just a text editor")
- They are complex (composed of several interacting entities)

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

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

Good example: audio system

Easy to manage because:

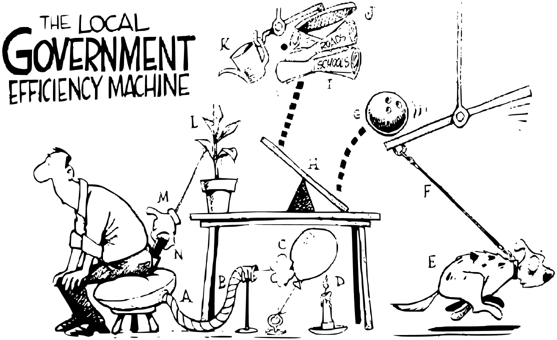
- each component has a carefully specified function
- components are easily integrated
- i.e. the speakers are easily connected to the amplifier

Composition counter-example (1/2)

Rube Goldberg machines

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

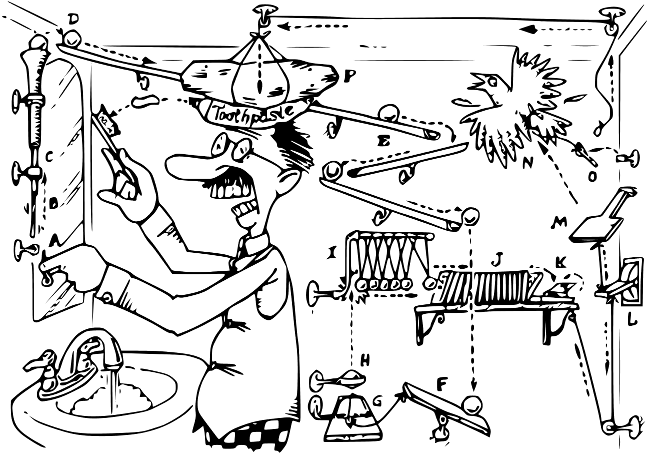
Example: Tax collection machine



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

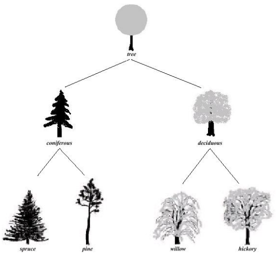


Such over engineered solutions should obviously remain jokes

Dealing with complexity: Abstraction

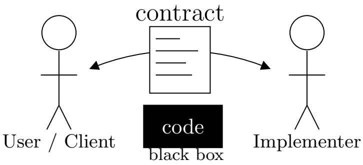
Abstraction

- 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



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

Empirical efficiency measurements

- Code the algorithm, benchmark it and use runtime statistics

☹ Several factors impact performance:
machine, language, programmer, compiler, compiler's options, operating system, ...

$\Rightarrow$ Performance not generic enough for comparison

Mathematical efficiency estimation

- Count amount of basic instruction as function of input size

☺ Simpler, more generic and often sufficient

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

Best case, worst case, average analysis

Algorithm running time depends on the data

Example: Linear search in 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
- $t_{max} = c + n \times (2t + 1c + 1a) = (n+1) \times c + 2n \times t + n \times a$

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

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

Reducing amount of variables

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

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

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

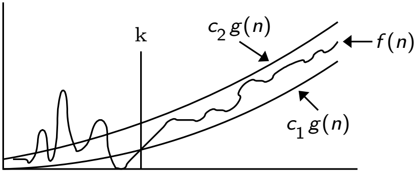
Example: $T(n) = c_1 \times n^2 + c_2 \times n$

- $T(n) = c_1 \times n^2 + c_2 \times n \geq c_1 \times n^2 \qquad \forall n > 1$
  $T(n) \geq c \times n^2$ 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 100n	10 1000	1000 10^5	10^5 10^7	seconds
$\Theta(n^2)$	n^2 $100n^2$	100 10^4	10^6 10^8	10^{10} 10^{12}	minutes
$\Theta(n^3)$	n^3 $100n^2$	1000 10^5	10^9 10^{11}	10^{15} 10^{17}	hours
$\Theta(2^n)$	2^n 100×2^n	1024 $> 10^5$	$> 10^{301}$ 10^{305}	∞ ∞	...
$\log(n)$	$\log(n)$ $100 \log(n)$	3.3 332.2	9.9 996.5	16.6 1661	

Classical mistakes

Mistake notations

- Indeed, we have $O(\log(n)) = O(n) = O(n^2) = O(n^3) = O(2^n)$
  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

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

- Multiplying big-Os

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

Some examples

Example 1: `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 $i \times 2$

First Chapter

Practical and Theoretical Foundations of Programming

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

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
- Asymptotic complexity must remain polynomial
- Note that you cannot have a decent complexity with the wrong data structure
- You still want to test the actual performance of your code in practice

Numerical stability

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

Second Chapter

Iterative Sorting Algorithms

Sorting Problem Specification

Input data

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

Result

- Permutation[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
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;
}
```

Memory Analysis

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

$\Rightarrow$ Constant amount of extra memory

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

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

Time Analysis

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

$\Rightarrow$ 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

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

Counting steps more precisely (but only dominant term)

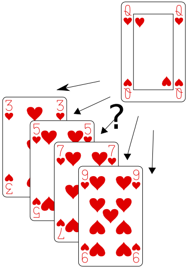
- $$T(N) = \sum_{i\in[1,N]} \left(\sum_{j\in[i,N[} 1 \right) = \sum_{i\in[1,N]} (N-i) = \sum_{i\in[1,N]} N - \sum_{i\in[1,N]} i$$
$$= N^2 - \frac{N\times(N+1)}{2} = N^2 - \frac{N^2+N}{2} = \frac{1}{2}N^2 - \frac{1}{2}N = \frac{1}{2}(N^2 - N)$$
- Let's prove that $T(n) \in \Omega(n^2)$. For that, we want: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

   . . .

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

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	

- 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"

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

Is it optimal?

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

Third Chapter

Recursion

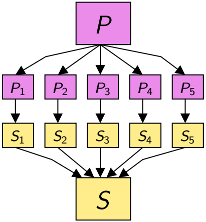
Divide & Conquer

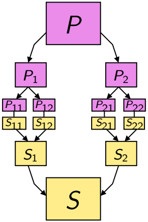
Classical Algorithmic Pattern

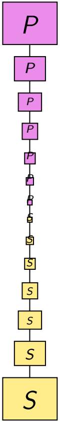
- When the problem is too complex to be solved directly, decompose it

When/How is it applicable?

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







Recursion

Divide & Conquer + sub-problems similar to big one

Recursive object

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

Base case(s)

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

When the base case is missing...

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

Recursive Functions and Procedures

Recursively Defined Function: its body contains calls to itself

The Scrabble™ word game

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

This is the Factorial

- Mathematical definition of factorial: $\begin{cases} n! = n \times (n-1)! \\ 0! = 1 \end{cases}$
- Factorial : integer $\longrightarrow$ 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

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

Remarks:

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

Factorial Computation Details

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

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

$$\left.4 \times 3 \times 2 \times 1 \times \overbrace{1}\right\} \text{Base Case}$$

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

$$\mathit{factorial}(4) = 24$$

Third Chapter

Recursion

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)

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

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

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

Corresponding Algorithm:

Pascal (n, k)

If $k = 0$ or $k = n$ **then** $r \leftarrow 1$

else $r \leftarrow$ Pascal $(n-1, k)$ + Pascal $(n-1, k-1)$

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

Other Recursion Schema: Mutual Recursion

Several functions calling each other

Example 1

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

Compute A(5):

Example 2: one definition of parity

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

Other examples

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

Other Recursion Schema: Embedded Recursion

Recursive call as Parameter

Example: Ackerman function

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

Thus the algorithm:

```
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:

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

$Ack(3,n) = 8 \cdot 2^n - 3$ $Ack(4,n) = 2^{2^{2^{\cdots^{2}}}}\Big\}n$

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

Recursive Data Structures

Definition

Recursive datatype: Datatype defined using itself

Classical examples

List: element followed by a list or empty list

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

This is the subject of the module "Data Structures"

- 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

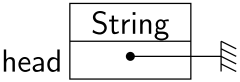
```
public class String {
  private Element head;

  // 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;
} }
```

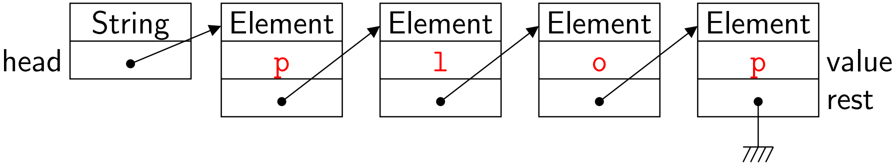
Need 2 classes to distinguish between empty string and uninitialized string variable

Some Memory Representation Examples

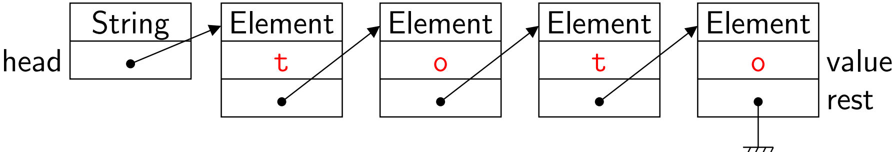
Empty String



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

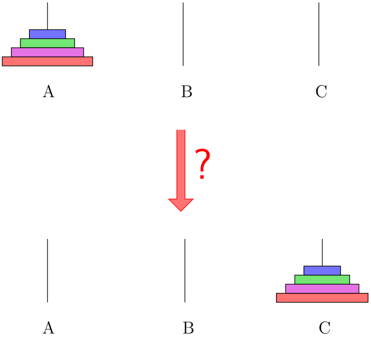
Third Chapter

Recursion

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

$\leadsto$ We recurse on integer n

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

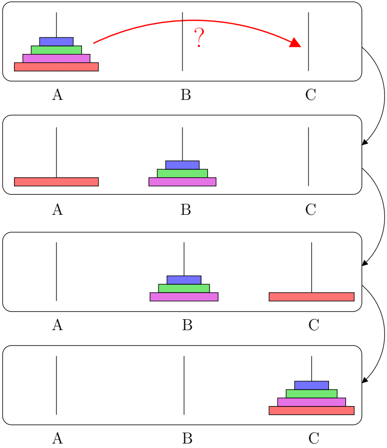
$\leadsto$ Decomposition between bigger disk and (n-1) smaller ones

- 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 $\textsc{To}$

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

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

Possible Decomposition of Hanoi(n, A, C)



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

MOVE(A,C)

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

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
```

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
```

Third Chapter

Recursion

Classical Recursive Function: Fibonacci

Study of reproduction speed of rabbits (XII century)

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

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

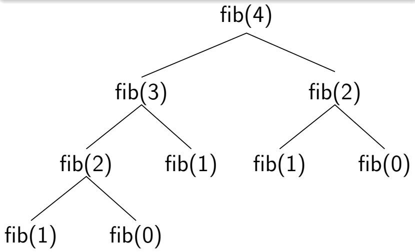
Exercice :

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: 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
- 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

Back on In-Memory Organization

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

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 $\neg\left(\exists s, (s > r) \wedge (a \bmod s = 0) \wedge (b \bmod s = 0)\right)$

Recursive Definition

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

Computation of gcd(420,75)

b	0
a	15
b	15
a	30
b	30
a	45
b	45
a	75
b	75
a	420

Stack

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

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

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

Compilers use these optimization techniques (amongst much others)

Non-Recursive Form of Terminal Recursion

- Consider the following Recursive Algorithm:

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

- The following Iterative Algorithm is equivalent:

```
f(x):
  u ← x
  until cond(u) do
    T(u)
    u ← u_int = h(u)
  end
  BASECASE(u)
```

With u_{int} being a locale computed by $\mathrm{T}(u)$

Example: Non-Recursive Form of GCD

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

cond(a,b): b=0

BaseCase(a,b): $r \leftarrow a$

GenCase(a,b): T(a,b);
$r \leftarrow gcd(a_{int}, b_{int})$

T(a,b): $a_{int} \leftarrow b$
$b_{int} \leftarrow a \bmod b$

Iterative Version (obtained by immediate rewriting):

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

(gcd has two parameters, thus some slight changes)

Transformation to Terminal Recursion

- Let **f(n)** be Non-Terminal Recursive Function
- Since non-terminal, previous method not applicable

- One can *sometimes* define an equivalent function **g()** being terminal recursion
  - $g()$ has more parameters than $f()$
  - Intermediate computations done on these accumulators during descent
  - Climb up thus useless

- $f()$ must have good properties (associativity, commutativity, ...)

- Approach: n operations during climb up $\Rightarrow$ n extra parameters
- One should check:
  - That the result can be obtained this way
  - That the resulting algorithm is Terminal Recursive

Example: non-recursive form of factorial

```
FACTO(n):
  if n = 0 then r ← 1
           else r ← n × facto(n − 1)
```

Functional Form: $facto(n) = (n = 0\ ?\ 1 : n \times facto(n-1))$

Multipl. by cste $a \neq 0$: $a \times facto(n) = (a \times n = 0\ ?\ a : a \times (n \times facto(n-1)))$

Def: $G(a,b) = a \times facto(b)$: $G(a,b) = (a \times b = 0\ ?\ a : a \times (b \times facto(b-1)))$

With associativity: $G(a,b) = (a \times b = 0\ ?\ a : (a \times b) \times facto(b-1))$

Thus: $G(a,b) = (a \times b = 0\ ?\ a : G(a \times b, b-1))$

$a \neq 0$: $G(a,b) = (b = 0\ ?\ a : G(a \times b, b-1))$

Note that G() is Terminal Recursion (no operation on climb up)

1 is neutral element of $\times \Rightarrow$ $facto(n) = G(1,n)$

$G()$ helps transforming *facto*() to Terminal Recursion

Non-recursive form of Factorial

```
FACTORIAL(n):
  result ← HELPER(1, n)

HELPER(a, b): (* that's G() of previous slide *)
  if b = 0 then result ← a
           else result ← HELPER(a × b, b − 1)
  end
```

- This function uses Terminal Recursion, transform to Non-Recursive Form:

```
HELPER(a,b):
  u ← a; v ← b            (* locales *)
  until v = 0 do
    u ← u × v             (* beware the order *)
    v ← v − 1             (* of updates *)
  end
  result ← u
```

- Other example: Non-Recursive Form of the computation of a string's length

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
- Example with only one recursive call

if cond(x) **then** $r \leftarrow g(x)$ **else** 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
```

Remarque:
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)
- 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 finished

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)
```

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

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

Combinatorial Optimization

Large class of Problems with similar approaches

Problem

- Solutions are really numerous; A set of constraints make some solution invalids
- We look for the solution maximizing a function

Examples

- 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
- Artificial Intelligence: select best solution from set of possibilities

Resolution Approaches

- Exhaustive Search: study *every* solutions (often exponential – ie infeasible)
  $\leadsto$ maximize value of any possible knapsack contents
- Backtracking: tentative choices + backtrack to previous decision point
  Restricting study to valid solutions $\leadsto$ if bag is full, don't stuff something else
  Factorizing computations $\leadsto$ only sum up once the N first objects' value

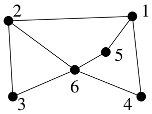
Back-tracking

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

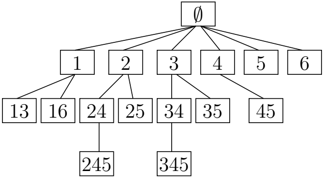
- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack



- $\{1\}$, $\{1,3\}$. 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) $\rightsquigarrow I_n = 2^n$ independent sets
- Full Graph (every edges) $\rightsquigarrow I_n = n + 1$ independent sets
- On average $\rightsquigarrow I_n = \sum_{k=0}^{n} \binom{k}{n} 2^{-k(k-1)/2}$

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

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

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

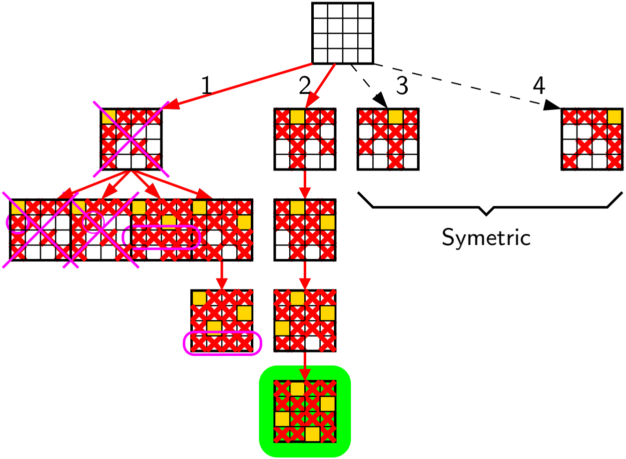
$\forall$ *cell* $\in$ *line*,

- 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
- 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

- 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, . . .
- BackTracking: exhaustive search in space of *valid* solutions
- Data Structure module: several recursive datatypes with associated algorithms