Computer Science 1

Introduction

Thomas Grenier

Summary

I. Introduction
   - Algorithms, complexity and programming

II. First example : sorting algorithms

III. Bases of C++ syntax

IV. Second example: sorting with linked list

V. C++ Classes and UML

VI. Useful data structures

VII. Exercises/project
   - Algorithms design and C++

Introduction

1. Why studying algorithms ?
   a) Definitions and interests of algorithms design
   b) *Pseudocode*
   c) Efficiency
   d) Correctness
2. Why learning C++ ?
   a) Languages history and future
   b) Object programming language
   c) C++

1.Why studying algorithms ?

a) Definitions and interests of algorithms design
b) P*seudocode*
c) Efficiency
d) Correctness

1.a – Definitions and interests

- Algorithm

  *Any well-defined computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output***

➔ Tool for solving well-specified computational problem

1.a – Definitions and interests

- Example
  - Sort a sequence of numbers into non-decreasing order
    - ***Input***: sequence of ***n*** numbers $\{a_1, a_2, \ldots a_n\}$
    - ***Output***: permutation (reordering) $\{a'_1, a'_2, \ldots a'_n\}$ of the input sequence such as $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

- Exercise:
  - Give an algorithm that computes the average value of ***n*** given numbers.
  - Give an algorithm that computes the median value of ***n*** given numbers.

1.a – Definitions and interests

- Conclusions of the exercise
  - How to specify an algorithm?
  - Is my algorithm correct ?
  - Is my algorithm better than another one ?
  - What kind of problems are solved by algorithms?
  - By the way…is it really interesting to study algorithms?
    - Computers are faster and faster!
    - Memory is cheaper and cheaper!

1.a – Definition and interests

How to specify an algorithms?

- Specifying an algorithm
  - Describe it in French or English or …
  - As computer program
  - As hardware design
  - …
  - ➔ *The only requirement is that the specification must provide a precise description of the computational procedure to be followed*
  - ➔ *But… How to convey the essence of the algorithm*
    - ➔ *Concisely?*
    - ➔ *Without issues of software engineering?*

➔ *We will use a dedicated language : **pseudocode***

1.a – Definitions and interests

Is my algorithm correct ?

- Correctness of an algorithm - definitions
  - An algorithm is said ***correct*** if, for every input instance, it halts with the correct output
  - A correct algorithm ***solves*** the given computational problem
- How to demonstrate an algorithm is correct?
  - ➔ We often use a ***loop invariant*** (similar to mathematical induction)
  - ➔ Sometimes difficult to use… (strange loop, recurrence)

1.a – Definitions and interests

Is my algorithm better than another one ?

- Algorithm analysis
  - Meaning: “predicting the resources that the algorithm requires”
    - ***Memory***, communication bandwidth, ***computational time***, …
  - ➔ By analyzing several algorithms for a problem, a most efficient one can be easily identified
- We generally focus on the computational time
  - The running time of an algorithm depends on the input and the size of the input !
  - ➔ Algorithms efficiency
    - ➔ ***Worst case***, best case, average case analysis
    - ➔ ***Order of growth*** of the running time function of the input size

1.a – Definitions and interests

What kind of problems are solved by algorithms?

- Problems **solved** by algorithms
  - All ? theoretically right (but practically wrong cause of bounded resources…)
  - ➔ Use of approximate/convergent algorithms and “incorrect” (with controlled error rate) algorithms
- Challenging problems are linked to
  - Optimization (minimization of a cost function $f(\mathbf{x})$): industrial, logistic, mathematical problems
  - Manipulation of large numbers (related to input size): DNA, chess, cosmology, cryptography, web search engine…
  - Real time

1.a – Definitions and interests

Is it really interesting to study algorithms?

- Guess in few years… Computers were infinitely fast and computer memory were free?
  - Computational time is null, no space memory limit…
  - All right but:
    - Does your process stop?
    - With the correct output ?
  - ➔ You have to demonstrate it
- In real world …
  - Computer are fast but not infinitely fast
  - Memory is cheap but not free
  - Energy?
  - Cost?
  - ➔ The most efficient algorithms is the best compromise

1.b – Pseudocode

How to specify an algorithm?

- *Pseudocode* to specify an algorithm
  - Example: compute the average of an array

```
AVERAGE(A)
1  ▷ Initialize sum at the first value of A : A[1].
2  sum ← A[1]
3  for j ← 2 to length[A]
4  ▷ Sum each value of A in sum : A[2 .. length[A]].
5        do sum ← sum + A[j]
6  return sum / length[A]
```

→ Indentation
→ Comment

1.b – Pseudocode

- Pseudocode conventions
  - Variables
    - Assignment is : ←
    - Local to the given procedure
    - Have the *right* type (described using mathematical notation)
    - Array elements are accessed by specifying the array name followed by the index in brackets. The first array element is at index 1.
    - Compound data are typically organized into **objects**, which are composed of **attributes** or **fields**. Objects and arrays are treated as pointer.
  - → Examples:

    A[1] ← A[2] * A[1]

    size ← *length[*A]

1.b – Pseudocode

- Pseudocode conventions
  - Tests: **if-then**, **if-then-else**
    - Indentation indicates block structure!
  - Examples

```
SIMPLE-IF-THEN-TEST(a, b)
1  if a > b
2      then
3           temp ← a
4           a ← b
5           b ← temp
```

```
SIMPLE-IF-THEN-ELSE-TEST(a, b)
       ▷ Test if a is greater than b.
1  if a > b
2      then return TRUE
3      else return FALSE
```

1.b – Pseudocode

- Pseudocode conventions
  - Loop : **for**, **while**, **repeat-until** (do-while)
    - Indentation indicates block structure!
  - Examples

```
SIMPLE-FOR()
1  for i ← 1 to 10
2        do
3            print " number : " i
```

```
SIMPLE-WHILE()
1  i ← 1
2  while i ≤ 10
3        do
4            print " number : " i
5            i ← i + 1
```

```
SIMPLE-REPEAT()
1  i ← 1
2  repeat
3            print " number : " i
4            i ← i + 1
5     until i > 10
```

1.b – Pseudocode

- Pseudocode conventions
  - Procedure parameters are passed by value!
    - Modification of a value parameter in procedure is not seen by the calling procedure
    - Modification of an object field (or an array element) is seen by the calling procedure (object are passed using pointer…)
  - Example

```
PARAMETERS-TO-PROCEDURE(A, b)
    ▷ the next assignment is not visible to the calling procedure
1   b ← 1
    ▷ the next assignment is visible to the calling procedure
2   A[i] ← 1
```

1.b – Pseudocode

- Pseudocode conventions
  - ***short circuiting*** *of Boolean operators (‘and’, ‘or’ …)*
    - Evaluate the expression “x *and* y”: first evaluate *x*, and then evaluate *y* to determine the whole expression only if *x* is evaluated to True.
    - Evaluate the expression “x *or* y”: first evaluate x, and then evaluate y only if x is evaluated to False.
  - Examples

```
SHORT-CIRCUITING(A, i, y)
    ▷ Cause of short circuiting we don't worry about what happens
    ▷ when we try to evaluate A[i] when i is NIL or out of range
1   if i ≠ NIL and A[i] = y
2       then
3               ...
4   if i ≤ lenght[A] and A[i] = y
5       then
6               ...
```

1.c – Efficiency

- Example: 2 algorithms for a same problem
  - i.e. insertion and merge sort
  - Computational time of these algorithms depends on the input size *n*
    - The first algorithm (insertion sort) takes time roughly equal to $c_1.n^2$ to sort *n* numbers
    - The second algorithm (merge sort) takes time roughly equal to $c_2.n.log_2(n)$ to sort *n* numbers
  - Insertion sort runs on a fast computer A (1 billion instructions per second) and is coded by the world's craftiest programmer. Resulting code requires $2n^2$ instructions to sort *n* numbers
  - Merge sort runs on a slower computer B (ten million instructions per second) and is coded by an average programmer using high level language with inefficient compiler. Resulting code requires $50n.log(n)$ instructions to sort *n* numbers
  - ➔ Give the computational times of each algorithm if n = {1 000, 38 000,1 000 000}

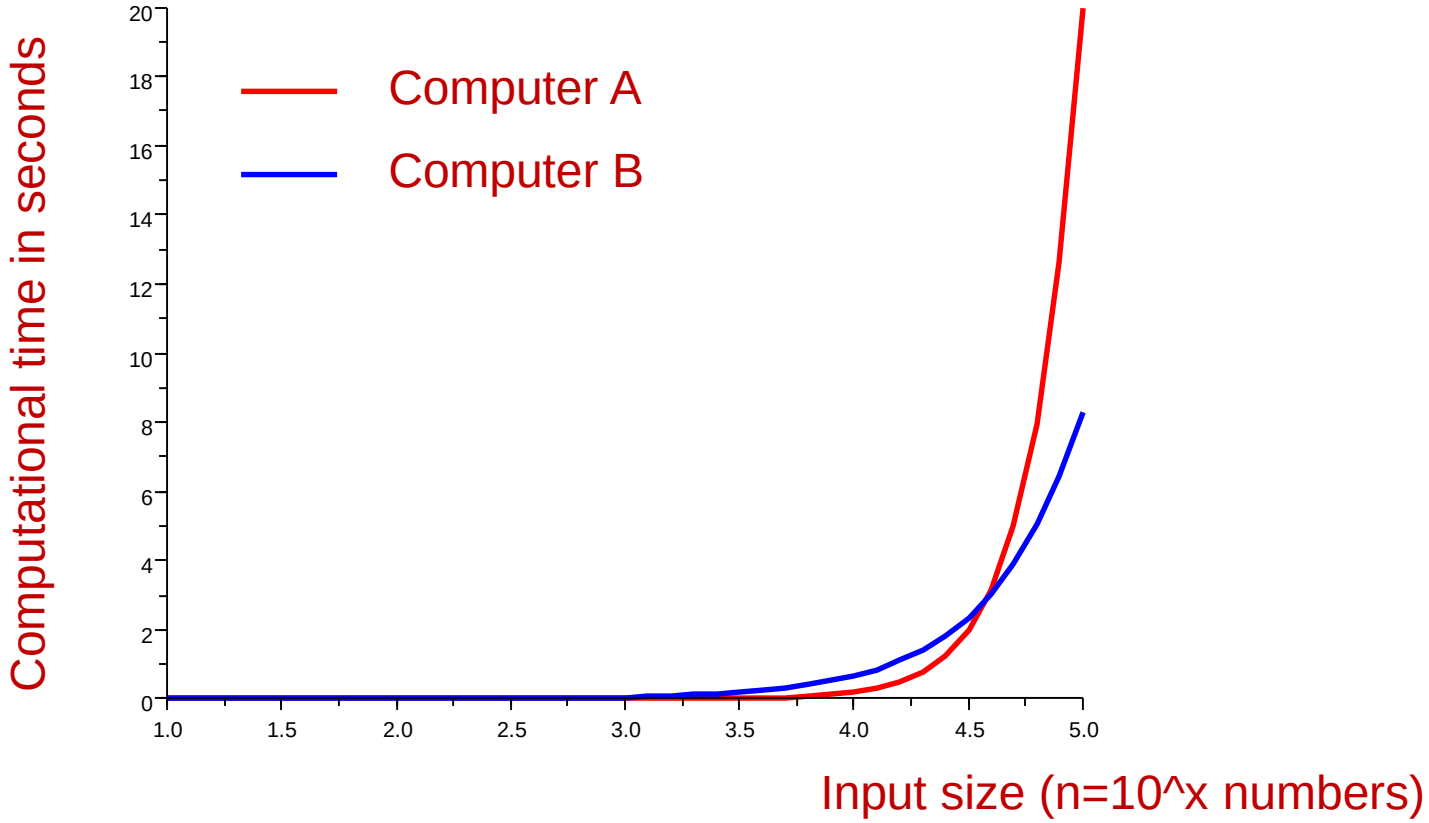
1.c – Efficiency

- Example: 2 algorithms for a same problem
  - Computer A : $\frac{2n^2 \text{instructio ns}}{10^9 \text{instructio ns / second}}$
  - Computer B : $\frac{50n\log_2(n) \text{instructio ns}}{10^6 \text{instructio ns / second}}$

n	1000	38000	1 000 000	10 000 000
A (in seconds)	0.002	2.888	2000	200000
B (in seconds)	0.050	2.890	99.65	1163

1.c – Efficiency

- Example: 2 algorithms for a same problem



1.c – Efficiency

- Definition : Running time of an algorithm $T(n)$
  - On a particular input, that's the number of executed primitive operations (or 'steps')
    - *step* should be defined as machine-independent as possible
    - Particular input can be: best case, worst case or average case
  - For pseudocode
    - a constant amount of time is required to execute each line of pseudocode:
      - The execution of the i^{th} line takes time $\boldsymbol{c_i}$
    - Running time of the algorithm is given by:

$$T(n) = \sum_{i=1}^{number\ of\ lines} c_i.(number\ of\ times\ line\ i\ is\ executed)$$

1.c – Efficiency

- Pseudocode example

```
AVERAGE(A)
1   ▷ Initialize sum at the first value of A : A[1].
2   sum ← A[1]
3   for j ← 2 to length[A]
4   ▷ Sum each value of A in sum : A[2 .. length[A]].
5         do sum ← sum + A[j]
6   return sum / length[A]
```

cost	*times*
0	1
c_2	1
c_3	n
0	n-1
c_5	n-1
c_6	1

$$T(n) = c_2 + n.c_3 + (n-1).c_5 + c_6$$
$$= n(c_3 + c_5) + c_2 - c_5 + c_6$$

1.c – Efficiency

- Order of growth
  - To simplify computational time analysis:
    - All c_i are equal to a constant time

$$\rightarrow T(n) = a.n + b$$

    - Consider only the leading term of formula, since the lower-order terms are relatively insignificant for large *n*

$$\rightarrow T(n) = a.n$$

- Asymptotic notation
  - We asymptotically bound the function $T(n)$
    - Asymptotic upper bound: *O*-notation $\rightarrow T(n) = O(n)$
    - Asymptotic upper and lower bounds: *Θ*-notation

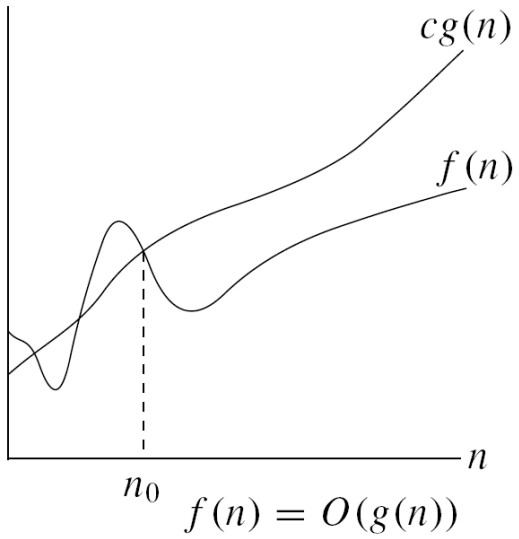
$$\rightarrow T(n) = \Theta(n)$$

Which is stronger than O(n)

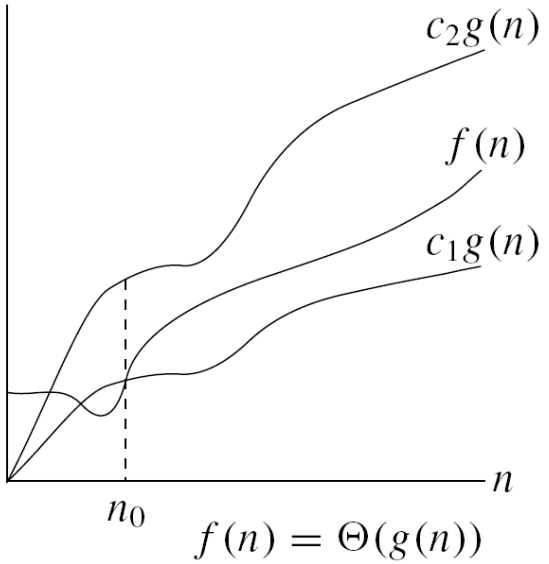
1.c – Efficiency

- Two classical asymptotic notations

$O(n)$ "big-oh"



$\Theta(n)$ "Theta"



$$O(g(n)) = \{f(n) : \exists c, n_0 > 0 | 0 \leq f(n) \leq cg(n), \forall n \geq n_0\}$$

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 | 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$$

1.c – Efficiency

- Exercise
  - Find the best-case and the worst-case running time of Insertion-Sort
  - Give the order of growth

→ Let t_j be the number of times the 'while' instruction is executed at iteration j

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          i ← j − 1
4          while i > 0 and A[i] > key
5               do A[i + 1] ← A[i]
6                  i ← i − 1
7          A[i + 1] ← key
```

1.d – Correctness

- We often use ***Loop invariant*** to understand why an algorithm gives the correct answer
  - *In computer science, a predicate that, if true, will remain true throughout a specific sequence of operations, is called (an)* ***invariant*** *to that sequence*. *(Wikipedia)*
- Proof of correctness is trivial… or very hard
  - Average proof of correctness is trivial
  - Insertion-Sort proof of correctness is also trivial!

1.d – Correctness

- To use **loop invariant** to prove correctness, we must show three things about it:
  - **Initialization**: It is true prior the first iteration of the loop
  - **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration
  - **Termination**: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct
- The **invariant** must be correctly defined

1.d – Correctness

- Example **Loop invariant** for Insertion-Sort
  - ➔ Invariant : the subarray A[1..j-1] is sorted
  - **Initialization**: before the first iteration j=2.
    - The subarray A[1] is sorted!
  - **Maintenance**:
    - Problem of the inner ***while*** loop
      - use another loop invariant,
      - or simply note that the ***while*** loop moves A[j-1], A[j-2],…by one position to the right until proper position for key is found
  - **Termination**:
    - The outer ‘for’ loop ends when j>n, this occur when j=n+1
    - Therefore n= j-1
    - Thus the subarray A[1..j-1] consists of the elements originally in A[1..n] but in sorted order

2.Why studying C++ ?

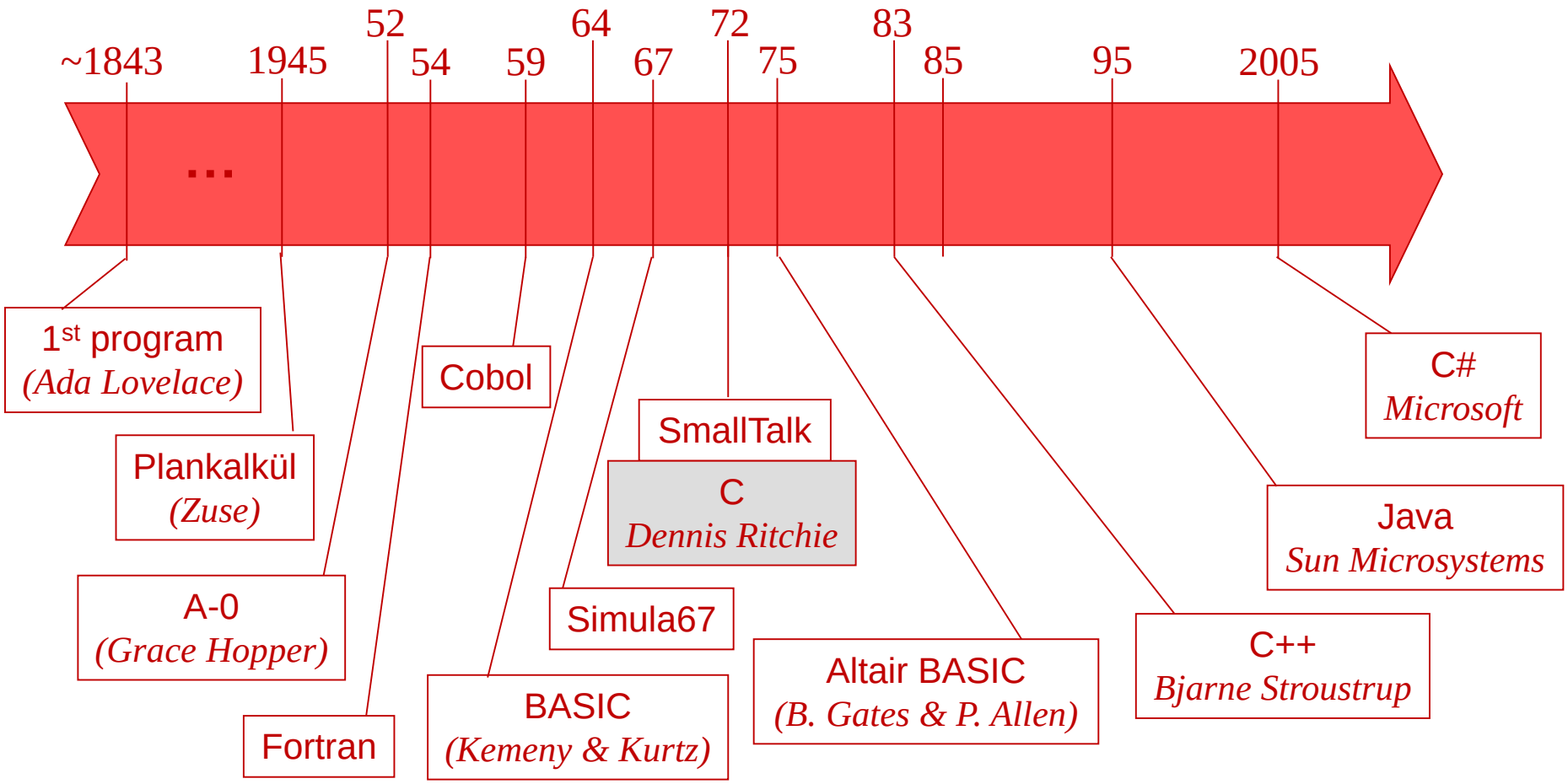
a) Languages history and future
b) Object programming language
c) C++ ?

2.a – Languages history and future

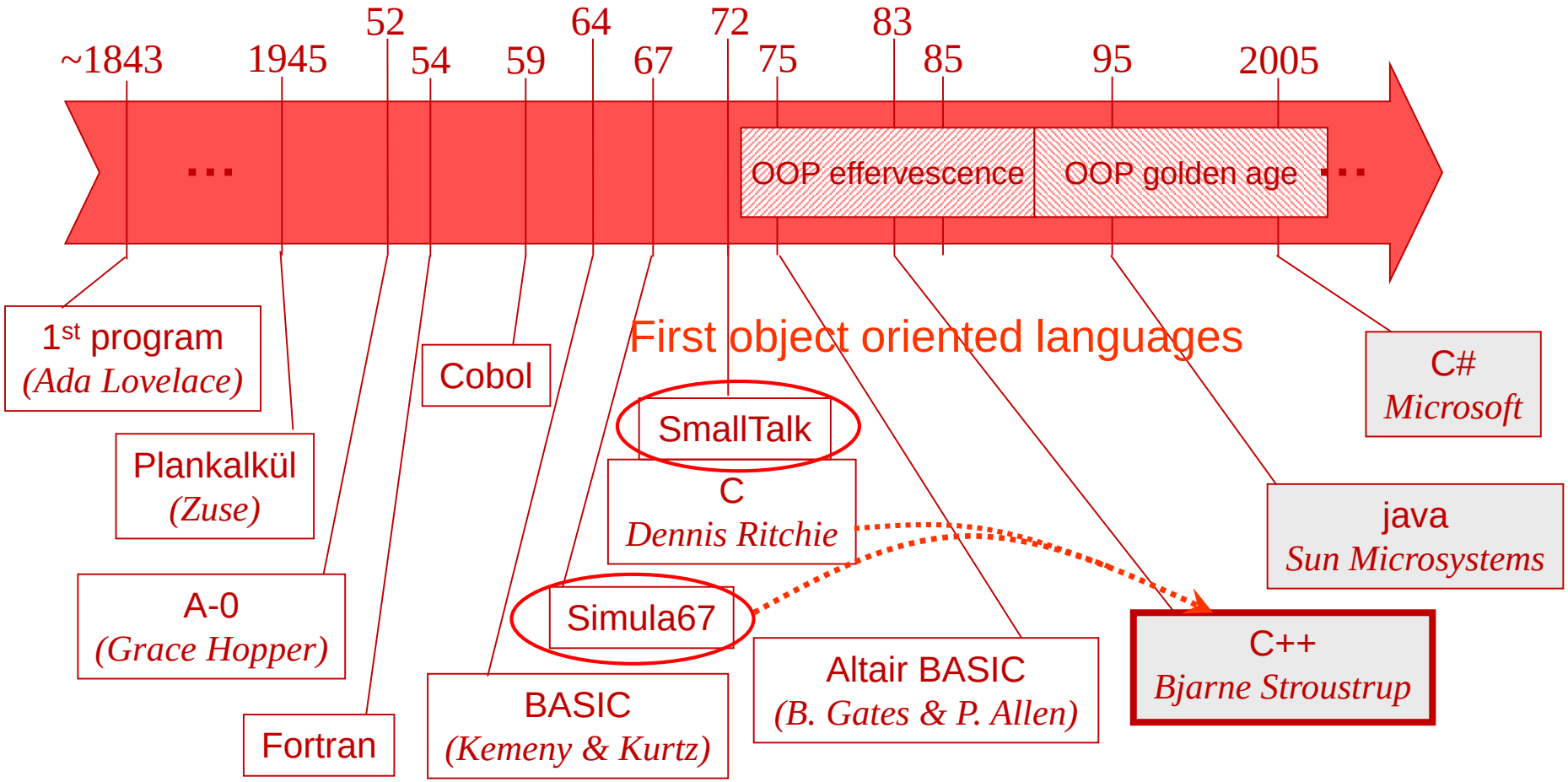
- When was the first program written ?

Ada Lovelace
1815-1852

2.a – Languages history and future



2.a – Languages history and future



OOP: oriented-object programming

2.a – Languages history and future

- What kind of language for the future?
  - Based on object-oriented language or on new paradigms
  - Very high level language
    - Hardware abstraction (like java)
    - Data type abstraction (cf. python)
    - Full object communication, reflection (ie. objectiveC, RUST)
    - Intelligent operators (*matlab*)
  - Visual (3D) programming language
    - Link objects and/or functions together (Access, Simulink)
    - No syntax language for algorithms (LabView, Alice)

2.b – Object-oriented languages

- The most *famous* are C/C++, java, python
- Now new standards of each language include object oriented capabilities
  - Cobol, Basic, matlab, fortran, Perl , PHP…
- In real-world
  - OOP can be used to translate from real-world phenomena to program elements (and vice versa)
- *Specify* OO Program → Modeling
  - UML: Unified Modeling Language

2.b – Object oriented languages

- Fundamental paradigms (top 3)
  - **Encapsulation**
    - ***fields*** and ***methods*** are merged into ***class***

    ➔ ***Object*** (or ***instance***) is a pattern (exemplar) of class
  - **Inheritance**
    - ***Subclasses*** are more specialized versions of a class, which ***inherit*** attributes and behaviors from their parent classes, and can introduce their own fields and methods
  - **Polymorphism**
    - Polymorphism allows the programmer to treat ***derived*** class members just like their parent class' members

2.c – C++

- Characteristics of C++
  - Combination of both high and low level language features
  - Statically typed
  - **Object oriented language**
  - Procedural programming
  - Data abstraction (or at least this possibility can be offered…)
  - Generic programming (template)
  - Operators overloading
  - …

New standard is C++0x … promising for the future

2.c – C++

- About the C++ syntax
  - Similar to C (useful for low level programming)
  - Similar to many languages (java, C#, Pascal, matlab, python, RUST)
  - Not so far from pseudocode ☺
    - The first array element is at index 0 …