Algorithms. Algorithmic language

Mădălina Răschip, Cristian Gațu

Faculty of Computer Science
Alexandru Ioan Cuza University of Iași, Romania

DS 2020/2021

Content

Algorithms. Introduction

Algorithmic language

Data types

Arrays and structures

Example

- Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Example

- Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- Fibonacci sequence

  mathematical definition: $F_n = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F_{n-1} + F_{n-2}, & \text{if } n > 1 \end{cases}$

Example

- Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- Fibonacci sequence

  mathematical definition: $F_n = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F_{n-1} + F_{n-2}, & \text{if } n > 1 \end{cases}$
- C code

```
int F(int n) {
  if (n == 0) return 0;
  else if (n == 1) return 1;
  else
    return F(n-1) + F(n-2);
}
```

Algorithms: etymology

Muhammad ibn Musa **al-Khwarizmi** - Persian mathematician ; wrote the first book on algebra (about 830).

- methods for number addition, multiplication and division.

Algorithms: definition

- There is no standard definition for the notion of algorithm.
- Cambridge Dictionary:

  "A set of mathematical instructions that must be followed in a fixed order, and that, especially if given to a computer, will help to calculate an answer to a mathematical problem."
- Schneider and Gersting 1995 (Invitation for Computer Science):

  "An algorithm is a well-ordered collection of unambiguous and effectively computable operations that when executed produces a result and halts in a finite amount of time."
- Gersting and Schneider 2012 (Invitation for Computer Science, 6nd edition):

  "An algorithm is ordered sequence of instructions that is guaranteed to solve a specific problem."

Algorithms: definition

- Wikipedia:

  "In mathematics and computer science, an algorithm is a step-by-step procedure for calculations. Algorithms are used for calculation, data processing, and automated reasoning. An algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function. Starting from an initial state and initial input (perhaps empty), the instructions describe a computation that, when executed, proceeds through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate random input."

Algorithms: computing model, solved problem

All definitions have something in common:

- data / information and their processing in steps.
- In general, this is described by a computing model.
  A **computing model** is given by:
  - **memory** - data representation.
  - **instructions**
    syntax - syntactic description of processing steps;
    semantic - describes the processing steps when executing one instruction; it is in general a transition relation between configurations (transition system).
- An algorithm has to produce a result, that is has **to solve a problem**.
- A problem can be seen as a pair (**input, output**), where input describes the input data (instance), while output describes the output data (result).

Algorithms and Data Structures

- Algorithm: method to solve a problem.
- Data structures: method to keep/represent data/information.

Algorithms and Data Structures

- Algorithm: method to solve a problem.
- Data structures: method to keep/represent data/information.

Algorithms + Data Structures = Programs. — Niklaus Wirth

I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships. — Linus Torvalds

Algorithms: properties

- **input** – zero or more supplied data entities.
- **output** – the information produced by the algorithm.
- **termination** – for any input, the algorithm executes a finite number of steps.
- **correctness** – the algorithm ends and produces the correct output for any input instance; we say that the algorithm **solves** the given problem.

Algorithms: efficiency

- An algorithm should use a reasonable amount of resources:: [space of] **memory** and [execution] **time**.

- Efficient algorithms are needed for:
  - save waiting times, storage space, energy consumption, etc.;
  - scalability: solving problems of higher dimensions using the same resources (CPU, memory, disk, etc.);
  - optimized solution.

Algorithms: efficiency

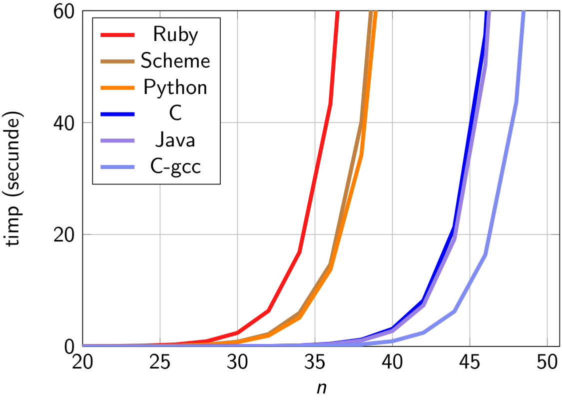


Fig.: Execution of the recursive algorithm F (Fibonacci).

Remark: The behavior depends on the implementation; still the differences are not substantial

$\Longrightarrow$ The algorithm is the problem! (exponential complexity).

Algorithm design

Algorithmic solving of a problem implies the following steps:

1. problem definition
   - irrelevant details abstraction;
2. identification of the problem class and of an algorithm that yields a solution;
3. algorithm efficiency and correctness analysis;
4. implementation;
5. (optimization and generalization).

Algorithm description

- **informal**: natural language.
- **formal**:
  - mathematical notation (Turing machine, lambda-calculus (Church), recursive functions, etc.);
  - programming languages: high level, low level, declarative (i.e., functional programming, logic programming). This can be also an informal model if there is no a formal semantic of the language.
- **semi-formal**:
  - pseudo-code: combines the formal notation of a programming language with natural language;
  - graphical notation: logic schemes, state machines, activity diagrams.

Content

Algorithmic language

There is a need for a language that is

- **simple**: easy to understand;
- **expressive**: in order to describe algorithms;
- **abstract**: when describing an algorithm the focus should be on the algorithmic thinking and not on the implementation details;

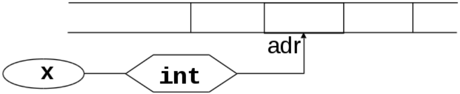
A computing models that is appropriate for complexity analysis, in particular the time complexity.

Variable

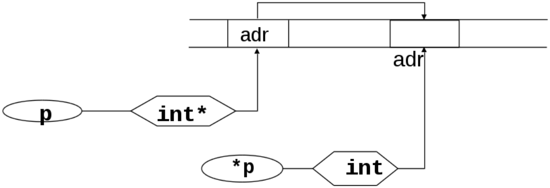
- Name
- Address
- Attributes (data type of the stored values)

- Variable instance

- Memory: linear structure of cells
  - variables



  - pointers



Content

Date type

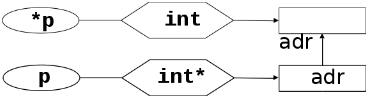
- Domain (object collection)
- Operations
- Data type categories:
  - Elementary data type
  - Low structured data types
    - operate at component level
  - High data types
    - operations are implemented by the user algorithms

Elementary data types

- Integer numbers
  - values: integer numbers
  - operators: $+$, -, ...
- Real numbers
  - values: rational numbers
  - operators: $+$, -, ...
- Boolean values
  - values: *true*, *false*
  - operators: *and*, *or*, *xor*, *not*

Elementary data types

- Char
  - values: 'a', 'b', ...
  - operators: —
- Pointers
  - values: same type variable addresses, NULL
  - operators: —
  - indirect reference: *p



Elementary data types

- Integer numbers operators:
  - arithmetic: a+b, a-b, a*b, a/b, a%b
  - relational: a==b, a!=b, a<b, a<=b, a>b, a>=b
  - logic: *and*, *or*, *xor*, *not*

operation	time	
	uniform cost	logarithmic cost
$a + b$	$O(1)$	$O(max(loga, logb))$
...		

Instructions

- Expressions
- Block: $\{$instructions$\}$
- Conditional: **if if-else**
- Iterative: **while repeat for**
- sequence break: **return**

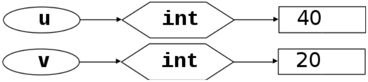
Instructions

- **Assignment**
  - **Syntax**: $< variable > \leftarrow < expression >$
  - **Semantic**:
    - $< expression >$ is evaluated and the results is stored in the memory location associated with $< variable >$;
    - it is the only instruction that can modify the memory content.
  - uniform cost $O(1)$, logarithmic $O(log < expression >)$

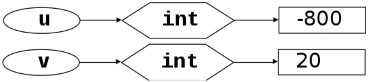
Assignment

Example:

- Before assignment :

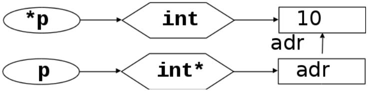


- After assignment $u \leftarrow -v * u$:



Instructions

- Pointer assignment
  - **Syntax**:
    $* < pointer\_variable > \leftarrow < expression >$
  - **Semantic**:
    - $< expression >$ is evaluated and the result is stored in the memory location that has the address $< pointer\_variable >$
  - Example: $*p \leftarrow 10$



Instructions

- **if**
  - **Syntax**:

    if $< expression >$ **then**
    $\quad < instruction - sequence_1 >$
    else
    $\quad < instruction - sequence_2 >$

    if $< expression >$ **then**
    $\quad < instruction - sequence_1 >$

    Remark: the $< expression >$ evaluation yields a Boolean result.

  - **Semantic**:
    - $< expression >$ is evaluated. If the result is *true*, then the $< instruction - sequence_1 >$ is executed, and if the result is *false*, then the $< instruction - sequence_2 >$ is executed; after-that the execution of the if instruction is terminated.

Instructions: if

- uniform cost $O(1)$, logarithmic cost $O(1)$
- Example: computing minimum of two numbers a and b:

if $a < b$ **then**
    $min \leftarrow a$
else
    $min \leftarrow b$

or
$min \leftarrow a$
if $b < a$ **then**
    $min \leftarrow b$

Instructions: while

- **while**
  - **Syntax**:

    while $< expression >$ **do**
    $< instruction - sequence >$

  - **Semantic**:
    1. $< expression >$ is evaluated
    2. If the result is *true* then $< instruction - sequence >$ is executed and the cycle is repeated from step 1. If the result is *false* then the execution of while is terminated.

while example

- smallest k such that $7^k >= n$ for a given n

$k \leftarrow 0$
$seven\_power\_k \leftarrow 1$
while $seven\_power\_k < n$ **do**
    $k \leftarrow k + 1$
    $seven\_power\_k \leftarrow seven\_power\_k * 7$

Instructions: **repeat**

- **repeat**
  - **Syntax**:

    repeat
    $< instruction - sequence >$
    until $< expression >$;

  - **Semantic**:
    Instruction:

    repeat
    S
    until e;

    simulates the execution of the following algorithm:

    S
    while *not* e **do**
    S

repeat: example

- smallest k such that $7^k >= n$ for a given n

  $k \leftarrow 0$

  $seven\_power\_k \leftarrow 1$

  repeat

      $k \leftarrow k + 1$

      $seven\_power\_k \leftarrow seven\_power\_k * 7$

  until $seven\_power\_k >= n$;

Instructions: **for**

- **for**
  - **Syntax**:

    for $< variable > \leftarrow < expression_1 >$ to $< expression_2 >$ **do**
    $< instruction - sequence >$

    or

    for $< variable > \leftarrow < expression_1 >$ downto $< expression_2 >$ **do**
    $< instruction - sequence >$

    Remark: $< variable >$ is an integer variable, and $< expression_1 >$ and $< expression_2 >$ evaluates to an integer

Instructions: **for**

- **for**
  - **Semantic**:

    for $i \leftarrow e1$ to $e2$ **do**
    $\quad S$

    is equivalent to:

    $i \leftarrow e1$
    $temp \leftarrow e2$
    while $i <= temp$ **do**
    $\quad S$
    $\quad i \leftarrow i + 1$

Instructions: **for**

- **for**
  - **Semantic**:

    for $i \leftarrow e1$ downto $e2$ **do**
    $\quad S$

    is equivalent to:

    $i \leftarrow e1$
    $temp \leftarrow e2$
    while $i >= temp$ **do**
    $\quad S$
    $\quad i \leftarrow i - 1$

Subprograms

- The language is modular: a program contains a number of modules.
- One module is identified by a subprogram.
- Subprograms:
  - Procedures
  - Functions

Subprograms

- **Procedures**:
  - **Syntax**:

    Procedure *name (formal-parameter-list)*
    begin
        instruction sequence
    end

  - **Call**: NAME(actual-parameter-list)
    - The interface between a procedure and a calling module is made through the parameters and global variables.

Procedures

- Example:

Procedure *SWAP (x,y)*
begin
  $aux \leftarrow x$
  $x \leftarrow y$
  $y \leftarrow aux$
end

Call:
SWAP(a, b)
SWAP(b, c)

Subprograms

- **Functions**:
  - **Syntax**:

    Function *name (formal-parameter-list)*
    begin
        instruction-sequence
    end
    instruction sequence contains at least one *return* $<$ *expr* $>$ instruction.

  - **Call**: NAME(actual-parameter-list)

    used within an expression: the returned value is the one obtained by evaluating $<$ *expr* $>$.

- Example:

  Function *max3(x,y,z)*
  begin
  $\quad temp \leftarrow x$
  $\quad$ **if** $y > temp$ **then**
  $\quad\quad temp \leftarrow y$
  $\quad$ **if** $z > temp$ **then**
  $\quad\quad temp \leftarrow z$
  $\quad$ return *temp*
  end

  Call:
  max3(a, b, c)
  2*max3(a, b, c) > 5

Content

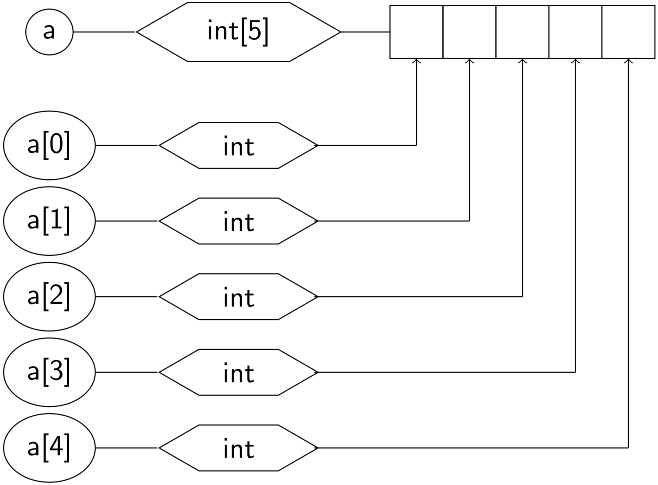
Arrays

- Homogeneous collection of variables called array components
- All components belong to the same type
- Components are identified through indices

- Arrays can be used to sets, sequences (the order of elements is important), matrices.

- Arrays can be:
  - one-dimension (vectors)
  - bi-dimensional (matrices)

Uni-dimensional arrays



Uni-dimensional arrays

- The memory is a contiguous sequence of locations.
- The storage order – indices order
- Operations are at component level
  Example:

  for $i \leftarrow 0$ to $n-1$ **do**
  $\quad a[i] \leftarrow 0$

  for $i \leftarrow 0$ to $n-1$ **do**
  $\quad c[i] \leftarrow a[i] + b[i]$

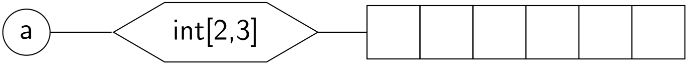
Arrays

Operation cost:

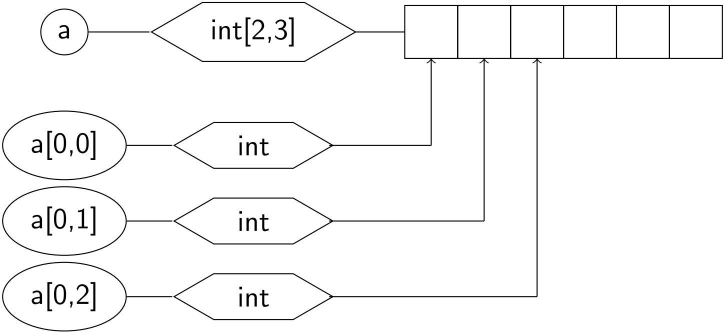
operation	time	
	uniform cost	logarithmic cost
$a[i]$	$O(1)$	$O(i + log a_i)$
$a[i] \leftarrow v$	$O(1)$	$O(i + log v)$

where a is a n–element array, with components $a[0], ..., a[n-1]$

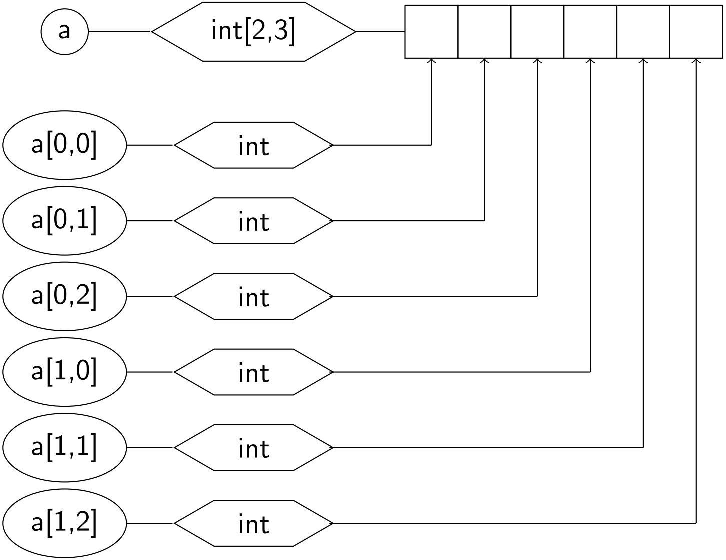
Bi-dimensional arrays



Bi-dimensional arrays



Bi-dimensional arrays



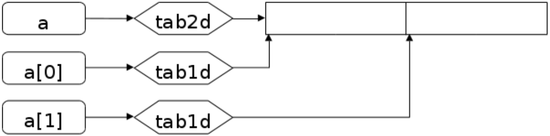
Bi-dimensional arrays

- Contiguous memory of $m \times n$ locations
- Components are identified by two indices:
  - first index ranges $\{0, 1, \ldots, m-1\}$
  - second index ranges $\{0, 1, \ldots, n-1\}$
  - component variables : $a[0,0], a[0,1], \ldots, a[0,n-1], a[1,0], a[1,1], \ldots, a[1,n-1], \ldots, a[m-1,0], a[m-1,1], \ldots, a[m-1,n-1]$
- The storage order follows the lexicographic order of indices.

Bi-dimensional arrays

- A bi-dimensional array can be also seen as an one dimensional array for which the components are one-dimensional arrays.

- Notation: $a[0][0], a[0][1], \ldots, a[0][n-1], \ldots,$ $a[m-1][0], a[m-1][1], \ldots, a[m-1][n-1]$



Bi-dimensional arrays

- Operations are at component level:

  for $i \leftarrow 0$ to $m - 1$ **do**
      for $j \leftarrow 0$ to n–1 **do**
          $c[i, j] \leftarrow 0$
          for $k \leftarrow 0$ to p–1 **do**
              $c[i, j] \leftarrow c[i, j] + a[i, k] * b[k, j]$

Strings

- They can be looked as uni-dimensional arrays with elements of type char.
- String constants are denoted by " ": "string"
- Operations: Concatenation denoted by $+$:
  "a string" $+$ "another string" $=$ "a stringanother string"

Structures

- Structure: Heterogeneous collection of variables called *fields*.
- A structure has a *name* and each field has its own *name* and *type*.
- Examples:
  1. a point in a plan can be represented by a structure "point" with two fields: x and y;
  2. One person can be represented by a structure "person" with three fields: name, age, address;
- Specifying a field:
  point.x, point.y
  person.name, person.age, person.address.street
  if p is pointer to a person structure then: $p- > age$

Structures

- The allocated memory to a structure is a contiguous one; fields are stored in the order of definition inside the structure.

Structures

Operation cost:

operation	time	
	uniform cost	logarithmic cost
$S.x$	$O(1)$	$O(log S_x)$
$S.x \leftarrow v$	$O(1)$	$O(log v)$

Algorithm execution

$x \leftarrow 0$
$i \leftarrow 1$
while $i < 6$ **do**
    $x \leftarrow x * 10 + i$
    $i \leftarrow i + 2$

Step	Instruction	i	x
0	$x \leftarrow 0$	–	–
1	$i \leftarrow 1$	–	0
2	$1 < 6$	1	0
3	$x \leftarrow x * 10 + i$	1	0
4	$i \leftarrow i + 2$	1	1
5	$3 < 6$	3	1
6	$x \leftarrow x * 10 + i$	3	1
7	$i \leftarrow i + 2$	3	13
8	$5 < 6$	5	13
9	$x \leftarrow x * 10 + i$	5	13
10	$i \leftarrow i + 2$	5	135
11	$7 < 6$	7	135
12		7	135

Algorithm execution

- **Computation**: sequence of elementary steps that yields when executing the algorithm instructions.
- **Configuration**: memory state + current instruction;

$x \leftarrow 0$
$i \leftarrow 1$
while $i < 6$ **do**
  $x \leftarrow x * 10 + i$
  $i \leftarrow i + 2$

Step	Instruction	i	x
0	$x \leftarrow 0$	–	–
1	$i \leftarrow 1$	–	0
2	$1 < 6$	1	0
3	$x \leftarrow x * 10 + i$	1	0
4	$i \leftarrow i + 2$	1	1
5	$3 < 6$	3	1
6	$x \leftarrow x * 10 + i$	3	1
7	$i \leftarrow i + 2$	3	13
8	$5 < 6$	5	13
9	$x \leftarrow x * 10 + i$	5	13
10	$i \leftarrow i + 2$	5	135
11	$7 < 6$	7	135
12		7	135

Algorithm execution

- **Computation**: sequence of elementary steps that yields when executing the algorithm instructions.
- **Configuration**: memory state + current instruction;
- In the example below the computations is the sequence of configurations ($c_0 \mapsto c_1 \mapsto \cdots \mapsto c_{12}$).

$x \leftarrow 0$
$i \leftarrow 1$
while $i < 6$ **do**
    $x \leftarrow x * 10 + i$
    $i \leftarrow i + 2$

Step	Instruction	i	x
0	$x \leftarrow 0$	–	–
1	$i \leftarrow 1$	–	0
2	$1 < 6$	1	0
3	$x \leftarrow x * 10 + i$	1	0
4	$i \leftarrow i + 2$	1	1
5	$3 < 6$	3	1
6	$x \leftarrow x * 10 + i$	3	1
7	$i \leftarrow i + 2$	3	13
8	$5 < 6$	5	13
9	$x \leftarrow x * 10 + i$	5	13
10	$i \leftarrow i + 2$	5	135
11	$7 < 6$	7	135
12		7	135