Informatics

Algorithmic Thinking

Claudio Sartori
Department of Computer Science and Engineering
claudio.sartori@unibo.it
https://www.unibo.it/sitoweb/claudio.sartori/

Elaboration on © Pearsons – Fluency with Information Technology – Snyder – Ch.10

Learning Objectives

- Explain similarities and differences among algorithms, programs, and heuristic solutions
- List the five essential properties of an algorithm
- Use the Intersect Alphabetized List algorithm to do the following:
  - Follow the flow of the instruction execution
  - Follow an analysis to pinpoint assumptions
- Demonstrate algorithmic thinking by being able to do the following:
  - Explain the importance of alphabetical order on the solution
  - Explain the importance of the barrier abstraction for correctness

The Letter Algorithm

- J.D. Bauby was paralyzed and could communicate only by blinking one eyelid
- An assistant would say or point to a letter and Bauby would indicate if it was the right one
  - One blink: no, two blinks: yes
- Point to letters until the correct one is reached
- Repeat to spell words and sentences

The Letter Algorithm

- This process is an algorithm: A precise, systematic method for producing a specified result
- We invent algorithms all the time
- An algorithm need not use numbers
- The agent running an algorithm may be a human being, rather than a computer
- There are better and poorer versions of this algorithm

Try…

- Try to specify the letters algorithm
- Starting point
- Actions
- Repeated actions
- When to stop

Goal

Convert **blinks** into a **written message**, one letter at a time.

- **1 blink** = “No”
- **2 blinks** = “Yes”

Algorithm sketch
“Scan-and-confirm spelling”

Setup

- Prepare an **ordered list of symbols** the assistant can offer (e.g., A–Z, plus space, punctuation).
- Have an **empty message** to build.

Body:
build the message letter-by-letter - I

Repeat until Bauby signals “done”:

- **Step A — Choose the next character**
  - Start at the **first symbol** in the list.
  - The assistant **says/points to the current symbol**.
  - Bauby responds:
    - If **1 blink (No)**: move to the **next symbol** and ask again.
    - If **2 blinks (Yes)**: select that symbol as the **next character**.

Body:
build the message letter-by-letter - II

- **Step B — Add it to the message**
  - Append the chosen character to the message.
- **Step C — Check if the message should end**
  - After each character (or after each word), the assistant asks: “Are you finished?”
    - If **2 blinks (Yes)**: stop and output the message.
    - If **1 blink (No)**: continue to the next character.

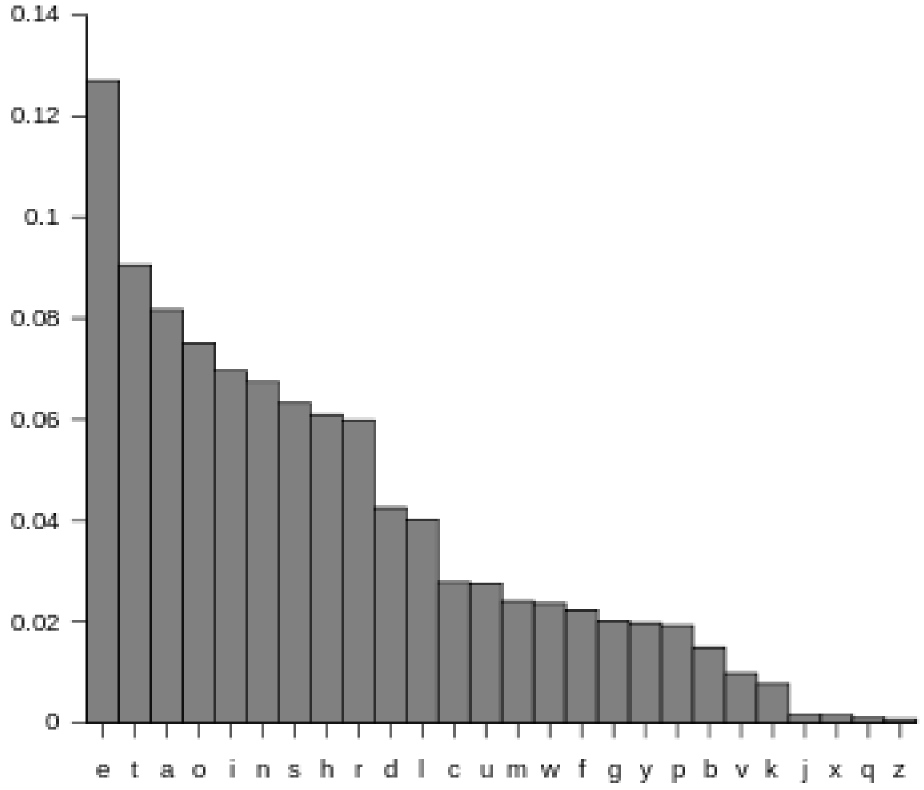
Lesson learned

- **A fixed set of steps** (repeatable procedure)
- **A loop** (keep scanning letters until “Yes”)
- **A decision rule** (one blink vs two blinks)
- **An output** (the growing message)

The Letter Algorithm – improvement?

- Making the process faster
  a) **Completion**: The assistant can guess a word before it is all spelled out
  b) **Ask the letters in frequency order**, and work from the most-frequently-used letter downward
  - What improvement in the process do you expect form the b) solution?

Relative frequencies of letters in english text



How can we specify an algorithm?

- It depends on the capabilities of the executor
- The more skilled is the executor, the less detail will be necessary
- Human executor
  - can use logic and experience
- Mechanical executor
  - must be instructed in detail

Discussion

1 Go to **wooclap.com**

2 Enter the event code in the top banner

Event code
JTQCHY

Programs are Algorithms

- **programs:** algorithms that have been specialized to a specific set of conditions and assumptions
- sometimes, the words *program* and *algorithm* are used interchangeably, but, in general
  - *algorithm* is a more abstract description of a solution
  - *program* is a precise description of a solution expressed in some *programming language*

Examples of Algorithms

- Binary to Decimal Conversion
  - if there is a 1, write down the place value for its position in decimal
  - add up those place values
  - *is this description precise enough?*
- Binary Addition
  - add as in decimal but limit digit values to two
  - *is this description precise enough?*

Another example:
Finding information on the web through a search engine

- begin with a general topic
  - examine the results
- choose descriptive words
  - examine the results
- refine by adding words
  - examine the results
- avoid overconstraining
  - examine the results
- remove specific words
  - examine the results

Algorithms vs. Heuristic Processes

- not all the description of processes are algorithms
- the process to find information on the web using a search engine was not an algorithm
  - not systematic
  - not guaranteed to find it (process could fail)
  - called a **heuristic process**: helpful procedure for finding a result

Algorithm Properties

- An algorithm *must* have five properties:
    1. Input specified
    2. Output specified
    3. Definiteness
    4. Effectiveness
    5. Finiteness

1. Input Specified

- The **input** is the data to be transformed during the computation to produce the output
- What data do you need to begin to get the result you want?
- Input precision requires that you know what kind of data, how much and what form the data should be

2. Output Specified

- The output is the data resulting from the computation (your intended result)
- Output precision also requires that you know what kind of data, how much and what form the output should be (or even if there will ***be*** any output at all!)

3. Definiteness

- Algorithms must specify every step and the order the steps must be taken in the process
- Definiteness means specifying the sequence of operations for turning input into output
- Details of each step must be spelled out (including how to handle errors)
- example:
  - compute x divided by y
  - input 6,0
  - the result is not defined

4. Effectiveness

- For an algorithm to be effective, each of its steps must be doable
- The agent must be able to perform each step without any outside help or extraordinary powers
- the description must be understood by de *executor agent*
- example of non-effectiveness:
  - tell the result of the coin toss I will do in the next minute

5. Finiteness

- The algorithm must stop, eventually!
- *Stopping* may mean that you get the expected output OR you get a response that no solution is possible
- Finiteness is not usually an issue for non-computer algorithms
- Computer algorithms often repeat instructions with different data and finiteness may be a problem
  - quite easy to write, by mistake, a never ending algorithm/program

Example 1:
division implemented with a sequence of subtractions – x/y

- subtract y to the current value of x
- repeat and stop only when x is equal to 0

- What if we start with x <- 6 and y <- 3?

the symbol **<-** is interpreted as an *arrow right to left* and is read as *store the <u>thing</u> on the right side into the <u>place</u> on the left side*

we will make clear what we mean as *thing* and *place*

Example 1: integer division implemented with a sequence of subtractions

```
counter <- 0
input x, y
repeat if x is greater than y
   x <- subtract y to the current value of x
   increment the counter by 1
output the value of the counter
```

- what happens if x is 6?
- what happens if x is 5?
- what happens if x is 0?
- what happens if x is -2

are there combinations of x and y that give problems?

Example 1: division implemented with a sequence of subtractions

- counter <- 0
- repeat if x is greater than y
  - subtract 3 to the current value of x
  - increment the counter by 1
- output the value of the counter

- What are the constraints on the inputs?

Example 2

- compute the result with all the decimals of 20/3

endless loop

Division algorithm - features

- Implements division with subtraction and a loop
- Not written in a programming language, instead just everyday English
    - but it does use “tech speak”
    - Writing in natural language instead of a programming language allows to omit details
- Written for people, not for computers

Algorithm Fact #1

1. Algorithms can be specified at different levels of detail
   – Algorithms use *functions* to simplify the algorithmic description
   – These functions (such as *scan*) may have their own algorithms associated with them

Algorithm Fact #2

2. Algorithms always build on functionality previously defined and known to the user
   - Assume the use familiar functions and algorithms
   - For example, “scan through” would use the ability to scan a list; b.t.w., what is a list?

Algorithm Fact #3

3. Different algorithms can solve the same problem differently, and the different solutions can take different amounts of time (or space)

Correctness

- anyone who creates an algorithm needs to know why it works
  - finding the algorithm's correctness-preserving properties and explaining why they do the job

Language

- If the executor is a human the algorithm can be specified in natural language?
  - Natural languages can easily be ambiguous
- *Formal languages* have been invented to precisely express concepts in well-defined contexts
  - E.g. the language of mathematics
- The *programming languages* are designed to specify algorithms for computer execution

Programming language

- The syntax is completely defined
- The semantics is completely explicit and known to the programmer
  - At least to the good one 😉
- Program = algorithm expressed with a programming language
- When a program does not work
  - Bad algorithm → design error
  - Bad translation → programming error

Life cycle of a program (i)

- Poblem definition
  - What are the possible inputs?
  - How can we obtain the solution? Alternatives? Possible errors?
- Algorithm definition
  - Precise definition of the steps necessary to generate the output from the given input
  - Semi-formal description, easy to understand
- Translation of the algorithm into a program
  - High-level programming language
    - It is expressed as a *text*, human-readable, it can be understood by a skilled reader
  - The translation requires a good skill

Life cycle of a program (ii)

- The designer of the programming language provides also the *translation software*
  - it generates the program expressed in *binary form*, that can be directly executed by the computer
- *Compiler*
  - The binary program can be executed independently, under the control of the operating system
- Interpreter
  - The programming language statements are executed in the context of a memory-resident software system

Life cycle of a program - synthesis

Correction of syntax errors

The compiler/interpreter finds errors

Problem definition

- Input definition
- Solution guidelines
- Forecast possible error condition

Algorithm definition

- Use a specific formalism
  - Flow chart
  - Pseudo-code

Programming

- Use a programming language
  - also related to the kind of computer

Compilation/interpretation

- Automatic translation into a machine-executable form

Execution

- Compilation vs interpretation
  - Compilation: the compiled program becomes a process
  - Interpretation : the translated program is executed in the context of the system interpreter process
- The input generates the output

Correction of algorithm and programming errors

The Input/Output transformation is not as intended

How can we describe an algorithm?

- What is the building material?
- Where do we store the information?
- How do we specify the operations
- How do we arrange the operations?
- How do we represent sequences
- Is there anything beyond sequences?

A concrete problem: deposit and interest

- We want to deposit an amount of money, say €100, in a bank
- At the end of each year the bank will pay the interest of 10% of the total amount
- At the end of first year the amount will be
  100 * (1 + 0.1)
- In the absence of withdrawals, what will be the amount after 5 years?

Wait and reflect

- Where do we store the values? And the partial results?

Variable

✓A memory area indicated by a *name*
✓Can be univocally referenced
✓Can contain a *value*
✓The value can be read/used
✓The value can be changed
✓The value has a *type*

Compound interest calculation: what variables do we need?

- A = initial amount of money
- r = interest rate
- $A_1 \leftarrow A*(1+r)$
- $A_2 \leftarrow A_1*(1+r)$
- $A_3 \leftarrow A_2*(1+r)$
- $A_4 \leftarrow A_3*(1+r)$
- $A_5 \leftarrow A_4*(1+r)$

Do we really need to keep the new amount at the end of each year?

Doing better

- Write an algorithm able to work for any positive amount of money, any positive interest rate, any integer number of years

→ the *parameters* of the problem

  - A, r, n

- The parameters can have different values for every *instance* of the problem
  - i.e. for every execution of the algorithm
- Do we need more variables?

Operations

- What is the *meaning* of
  $A_2 \leftarrow A_1*(1+r)$
- Compute the value of the expression on the right of the arrow, using the current values of variables and the constant values (e.g. the number 1)
- Store the result in the variable on the left of the arrow
- What happens to the variables after the operation?
  - The variables on the right are untouched
  - The variable on the left has a new value
- Can we have on the left something different from a single variable?

NO!

How many variables?

- Do we need to store all the amounts at the end of each year?
  - We are only interested in the final result, say A_f
- At each step the operation could be
  $$A_f \leftarrow A_f * (1 + r)$$
- The old value of A_f is lost after each computation

Let's try

- $A_f \leftarrow A_f*(1+r)$
- $A_f \leftarrow A_f*(1+r)$
- $A_f \leftarrow A_f*(1+r)$
- $A_f \leftarrow A_f*(1+r)$
- $A_f \leftarrow A_f*(1+r)$

What's missing?

The starting point!

Try again

- $A_f \leftarrow A$
- $A_f \leftarrow A_f*(1+r)$
- $A_f \leftarrow A_f*(1+r)$
- $A_f \leftarrow A_f*(1+r)$
- $A_f \leftarrow A_f*(1+r)$
- $A_f \leftarrow A_f*(1+r)$

It works!

Sequence

- The sequence is essential
- $A_f \leftarrow A$ must precede all the other operations
  - Otherwise it would not be possible to compute $A_f \leftarrow A_f*(1+r)$
    - The A_f value is still unknown
- Our computers are *sequential machines*
  - The statements are executed *one after the other*

Repetition

- In the initial problem definition an operation is repeated five times
- In general the repetition can happen any number n of times
- We should represent shortly the repetition

Repeat n times

Compound interest for n years

- $A_f \leftarrow A$

The indentation *delimits* the *scope* of the repetition

Add input and output

Compound interest for n years

- Use variables A, n, r, A_f
- Read A, n, r
- $A_f \leftarrow A$
- Repeat n times
  - $A_f \leftarrow A_f * (1 + r)$
- Write A_f