IALP 2011 - Algorithms

P. Stallinga

MIEET 1º ano

Algorithms are general descriptions to solve a problem. Imagine we want to find the greatest common divisor of two integer numbers. (The GCD of two numbers is the largest number that divides both of them without leaving a remainder). How would we go about doing that? We could do some puzzling for each set of two numbers, like solving a Sudoku puzzle, but what we want is a general solution, a sequence of precise and well defined steps to follow that always will lead to a solution. Such a set of well-defined instructions is called an algorithm. (Also for Sudoku we can design an algorithm).

The most popular and well known algorithms are probably for addition, subtraction, multiplication and long division, something that we all learned to do in primary school (at that time most of us probably did not realize that we were learning an algorithm here).
Something as simple as addition has an algorithm behind it. Something that we learn and blindly reproduce without noticing the intricacies behind it. But imagine as stupid idiotic machine trying to do our steps. We have to be VERY careful how we write down the steps. It may be something like this (to arrive at this, you have to **think like a machine!**):

1) Make the numbers of same length by adding 0's in front of the shortest
2) Define a 'carry' to temporarily store information
3) Start with the last digit (the digit on the right, the one with least weight). Reset the carry to 0
4) Add the two digits plus the carry. If the result is larger than 10, put the '1' in the carry and the remainder to the result. (If not, then reset the carry to 0 and copy the sum to the result)
5) Print the result
6) If there are more digits, move one digit to the left and go to step 4)
7) If the carry is nonzero, print the carry
8) Ready

An example is adding 24 to 2386:

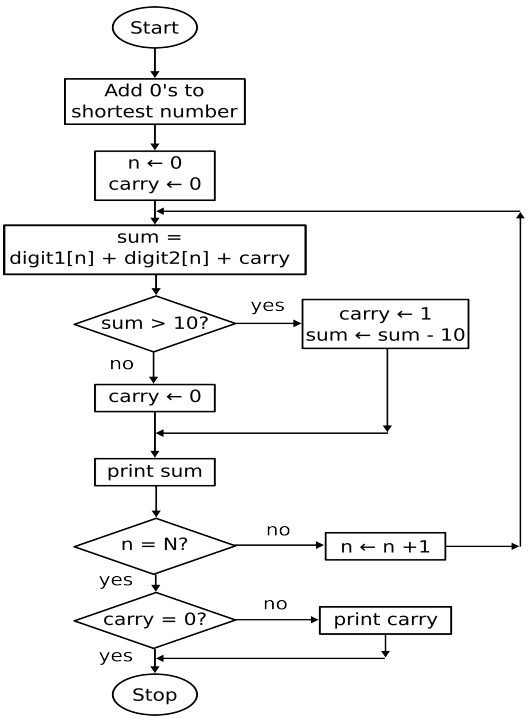
Pass	State before	State after
Start		2386 24 ---- +

1	2386 24 ---- +	2386 **00**24 ---- +
2	 2386 0024 ---- +	**x** 2386 0024 ---- +
3)	x 2386 0024 **---- +**	**0** 238**6** 002**4** ---- +
4)	**0** 238**6** 002**4** **---- +**	**1** 2386 0024 ---- + **0**
4)	**1** 23**8**6 00**2**4 ---- + 0	**1** 2386 0024 ---- + **1**0
4)	**1** 2**3**86 0**0**24 ---- + **1**0	**0** 2386 0024 ---- + **4**10
4)	**0** **2**386 **0**024 ---- + 410	**0** 2386 0024 ---- + 2410
6), 7), 8)	0 2386 0024 ---- + 2410	Ready!

Exercise

1) Design the passes for a subtraction or multiplication algorithm

When designing algorithms it is often very useful to write the 'program' in a flow chart. The passes above are linked by symbols that stand for 1) 'instructions' (things to do to change the state) and 2) 'questions' (as to the state) for decisions to make, i.e., which direction the program should make; which instructions and questions should follow. An instruction is normally put in a rectangular box(□)and a question in a lozenge (◇). Additionally, the start and stop of the algorithm are put in circles (◯). The addition algorithm thus becomes



This looks quite complicated, but don't forget that these are instructions, a 'program' for the most stupidest of stupid, yes, for a computer.

Exercise

2) Design the flow chart for your subtraction / multiplication algorithm

We made here an interesting intrinsic assumption. The flow chart cannot be altered during its execution! In other words, the 'program' cannot change the program itself, but only the state of the 'data' (in this case the numbers). This seems quite obvious for us human beings, but not so obvious for computers. After all, for a computer the program itself is also a form of data. The state of a computer is the state of the data **plus** the state of the program. Good programmers leave the state of the program alone and only alter the data. The idea of separating the data from the instructions comes from Von Neumann.

Exercise

3) Replace the boxes in the flow chart above:

carry = 0

By

Question above becomes “sum < 10?”
carry = 0

and

carry = 1

by

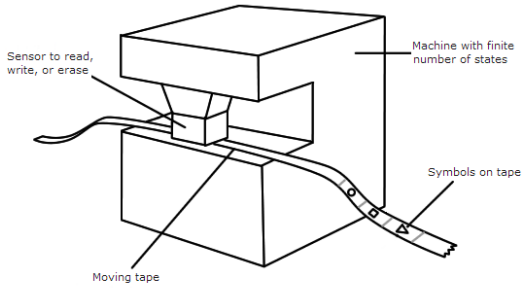
Question on left becomes “sum > 10?”
carry = 1

and check what will be happening to the program output.

The computer has no problem whatsoever in dealing with this programs. For humans to understand what is going on is more problematic.
This is very similar to the human game of Nomic. The rule of Nomic is that rules can be changed

> Nomic is a game in which changing the rules is a move. In that respect it differs from almost every other game. The primary activity of Nomic is proposing changes in the rules, debating the wisdom of changing them in that way, voting on the changes, deciding what can and cannot be done afterwards, and doing it. Even this core of the game, of course, can be changed.
>
> -- Peter Suber, the creator of Nomic

Another example to explain the state of a computer is via the finite-state machines. In principle all computers have a finite number of states, but modern computers have so many states that it is difficult to oversee. In a Turing Finite-State Machine (named after computer pioneer Alan Turing) the number of states and the possible number of inputs are limited and we can understand what is going on.



A Turing Machine
(http://www.ipod.org.uk/reality/reality_universe_computer.asp)

The machine can be in a finite number of states. Through the machine passes an infinite tape with 'instructions' (symbols) that can change the state. But, depending on the state

and the symbol read, not only the state changes, but also the symbol on the tape can be written. This results in an intricate interaction between state and tape, (in a weak analogy: between 'memory' and 'program'; note that the two can change each other). An example of a system: for a machine with two possible states and a tape with two possible symbols we could define the following 'action table' (what to do when a certain symbol is read and the machine is in a certain state):

State: / Symbol read:	A	B
0	1-R-B	1-L-A
1	1-L-B	1-R-H

Actions:
1 = write 1 on tape at current position
R = move right one place, L = move left one place
A = change state to A, B = change state to B, H = halt (ready)

Imagine we set the machine to an initial state A and insert a tape that has 0s all over it from plus to minus infinity (so much for finite state!). What will be the outcome of this program? The table below shows the state at each step. An underlined digit shows the position of the reading head

Step	Tape state	Machine state	What to do next?
0	… 0 0 0 0 0 **0** 0 0 0 0 0 …	A	1-R-B
1	… 0 0 0 0 0 1 **0** 0 0 0 0 …	B	1-L-A
2	… 0 0 0 0 0 **1** 1 0 0 0 0 …	A	1-L-B
3	… 0 0 0 0 **0** 1 1 0 0 0 0 …	B	1-L-A
4	… 0 0 0 **0** 1 1 1 0 0 0 0 …	A	1-R-B
5	… 0 0 0 1 **1** 1 1 0 0 0 0 …	B	1-R-H
6	… 0 0 0 1 1 **1** 1 0 0 0 0 …	H	-

Exercises

3) What would be the outcome if we fed a tape with all 1s to the above machine?

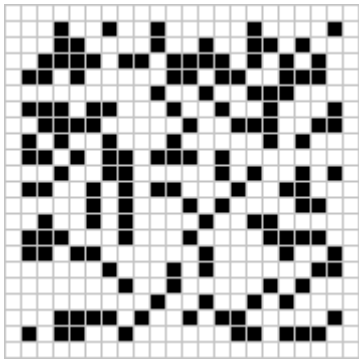
4) For a two-symbol three-state system the following action table is designed

State: / Symbol read:	A	B	C
0	1-R-B	0-R-C	1-L-C
1	1-R-H	1-R-B	1-L-A

What will be the outcome of feeding a tape with all 0s?

A more popular 2 dimensional finite-state-machine-like system is The Game of Life. It consists of a 2 dimensional grid of cells that can be either dead (white) or alive (black). The evolution of the system has the following rules for each pass:

- A dead cell with exactly three live neighbors becomes a live cell (birth).
- A live cell with two or three live neighbors stays alive (survival).
- In all other cases, a cell dies or remains dead (overcrowding or loneliness).



How will this grid evolve? Find out in the course of MIEET!

An example of a more advanced algorithm is the solution of finding a magic square. A magic square is a matrix of numbers in which the sum of all elements in each row or column is equal. An example is given here below, with the sum of each row and column equal to 15:

6	1	8
7	3	5
2	9	4

A simple algorithm (for magic squares with an odd number of columns/rows) is the following:

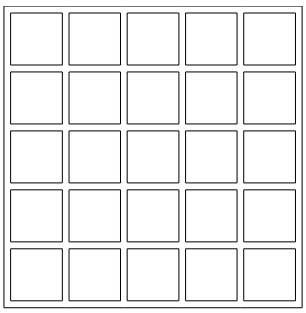
1. Begin by placing the number 1 in the middle of the top row
2. If possible, place the next number directly up and to the right of the current number
    3. If not possible because it is outside the matrix, place it at the opposite end of the matrix
    4. If not possible because the cell is occupied already, place the number directly below the current number
5. If still numbers left, go to step 2

This algorithm would result in the following magic square:

8	1	6
3	5	7
4	9	2

Exercises

5) Fill a 5x5 square with numbers from 1 to 25 with the above algorithm and check that the result is indeed a magic square



6) Design the flow chart of this algorithm

7) Note that the magic 3x3 square given before and the magic square of the algorithm are different. One is the mirror image of the other. What other geometric operations can be performed on any magic square that keeps it magic? (8 in total if we include the operation 'leave intact').

Postfix (Reverse Polish) notation vs. Infix notation

Normally when we write an arithmetic expression[1] we use the infix notation we learned at school:

(3x4 + 6)/2

which results in 9, as we all know. The problem is that computers do not understand this human readable notation. They have a CPU (central processing unit) that typically can operate on two values at a time. Like adding, or dividing. The above infix notation has thus first to be converted to postfix notation (better know as Reverse Polish Notation), which is of the form

3 4 x 6 + 2 /

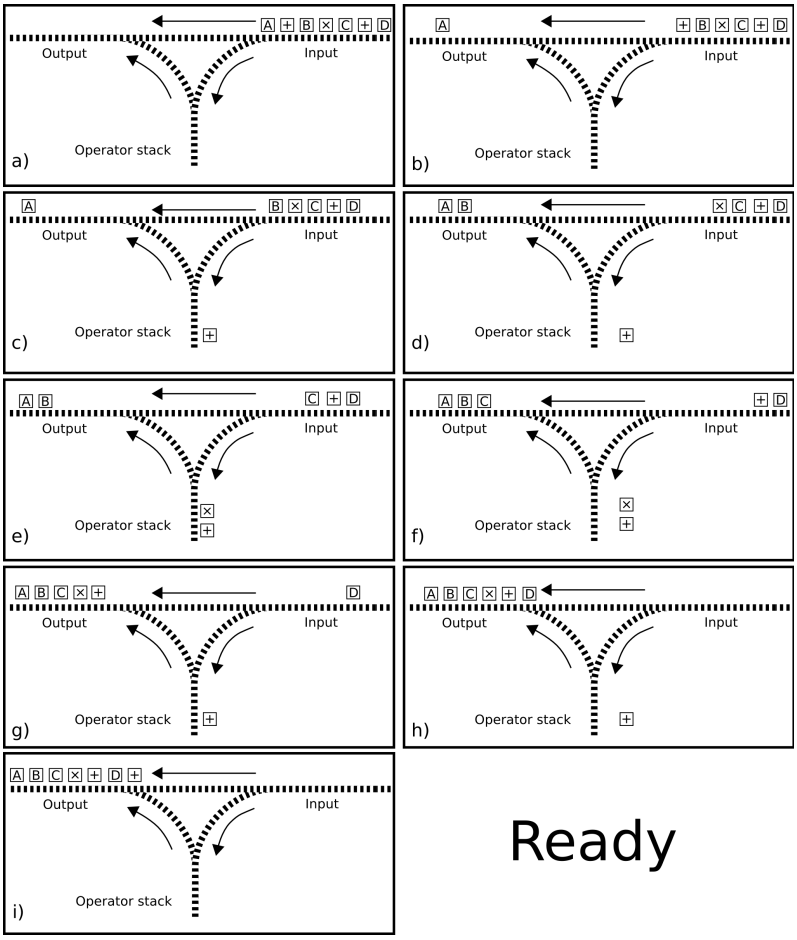
What the computer then does is interpreting the 'commands' above (tokens as they are

1 Expression: Something that returns a value

called) from left to right:

1) '3': Place it on the 'stack' (stack is a mountain of numbers). Stack: 3
2) '4': Place it on the stack. Stack: 4 3
3) 'x': Take the top two values (3 and 4) from the stack, multiply them and place result (12) on the stack. Stack: 12
4) '6': Place it on the stack. Stack: 6 12
5) '+': Take the top two values (12 and 6) from the stack, add them and place result (18) on the stack. Stack: 18
6) '2': Place it on the stack. Stack: 2 18
7) '/': Take the top two values (18 and 2) from the stack, divide second by the first and place result (9) on the stack. Stack: 9
8) No more tokens. Print top value of stack: 9

Some calculators, like most of Hewlett Packard, still work with this postfix notation. The algorithm to convert infix to postfix is also not difficult. See the figure below, which is called the shunting-yard algorithm, because it resembles shunting train wagons in a station. The figure and description of the algorithm are copied from Wikipedia:



The input is processed one token at a time, if a variable or number is found it is copied directly to the output (b, d, f, h). If the symbol is an operator it is pushed onto the operator stack, the vertical branch of the train tracks (c, e). However, if the precedence of the

operator is less than that of the operator at the top of the stack or the precedences are equal and the operator is left associative then that operator is popped off the stack and added to the output (g). Finally remaining operators are popped off the stack and added to the output (i). Ready.

A worked out example:

Input
3 + 4 x 2 / (1 - 5) ^ 2 ^ 3

Operator	Precedence	Associativity
^	4	Right
x	3	Left
/	3	Left
+	2	Left
-	2	Left

Token	Action	Output in RPN	Operator Stack	Notes
3	Add token to output	3		
+	Push token to stack	3	+	
4	Add token to output	3 4	+	
x	Push token to stack	3 4	+ x	x has higher precedence than +
2	Add token to output	3 4 2	+ x	
/	Pop stack to output	3 4 2 x	+	/ and x have same precedence
	Push token to stack	3 4 2 x	+ /	/ has higher precedence than +
(	Push token to stack	3 4 2 x	+ / (	
1	Add token to output	3 4 2 x 1	+ / (	
–	Push token to stack	3 4 2 x 1	+ / (–	
5	Add token to output	3 4 2 x 1 5	+ / (–	
)	Pop stack to output	3 4 2 x 1 5 –	+ / (	Repeated until "(" found
	Pop stack	3 4 2 x 1 5 –	+ /	Discard matching parenthesis
^	Push token to stack	3 4 2 x 1 5 –	+ / ^	^ has higher precedence than /
2	Add token to output	3 4 2 x 1 5 – 2	+ / ^	
^	Push token to stack	3 4 2 x 1 5 – 2	+ / ^ ^	^ is evaluated right-to-left
3	Add token to output	3 4 2 x 1 5 – 2 3	+ / ^ ^	
end	Pop entire stack to output	3 4 2 x 1 5 – 2 3 ^ ^ / +		

When we are using Matlab or Octave, or any modern memory-based programming language like C, Pascal, Java or Fortran, (in contrast to stack-based languages such as PostScript, Forth), our computer is doing this work for us behind the scenes. Converting to

postfix and executing.

Exercise

Convert the infix expression

(5x3^(4-2)+9x4)/(5-2)

to postfix and verify the result.

Greatest common divisor (GCD)

For finding the greatest common divisor we can use the Euclidean algorithm that is based on the principle that the greatest common divisor of two numbers does not change if the smaller number is subtracted from the larger number. For example, 21 is the GCD of 252 and 105 (252 = 21×12; 105 = 21×5); since 252 − 105 = 147, the GCD of 147 and 105 is also 21. Since the larger of the two numbers is reduced, repeating this process gives successively smaller numbers until one of them is zero. When that occurs, the GCD is the remaining nonzero number.)

Design here a flow-diagram of the algorithm. In the TP lectures we will write an Octave program that will do this.

And, finally, the algorithm that will save your trouble in the future: