How To Think About Algorithms

In Theory and Practice

Elisa Elshamy

Dr. Victoria Gitman (mentor)

New York City College of Technology

CLAC Seminar

December 2, 2008

What is an Algorithm?

An algorithm is a mechanical procedure that takes an input and carries out a collection of instructions to produce an output in finite time.

Algorithms are almost everywhere in nature

- Writing a paper (following format)
- Industrial processes
- Laundry
- Cooking (by recipe)

Key points

- Takes input, gives output
- Carried out mechanically
- Executed in finite time
- More than one algorithm for the same problem can exist
- Algorithms can only make sense if they aim to achieve a desired result

The word algorithm comes from the name of the 9th century Persian mathematician Abu Abdullah Muhammad ibn Musa al-Khwarizmi whose works introduced Indian numerals and algebraic concepts

Al-Khwarizmi (780 – 850)

In computer lingo an algorithm is a computer program

Algorithms Are Everywhere!

In simpler words - An algorithm is an explicit collection of steps for carrying out a task.

Addition is an example of a simple algorithm that we use almost daily!

Addition

For example consider 574 + 236

```
  11
  574
+ 236
------
  810
```

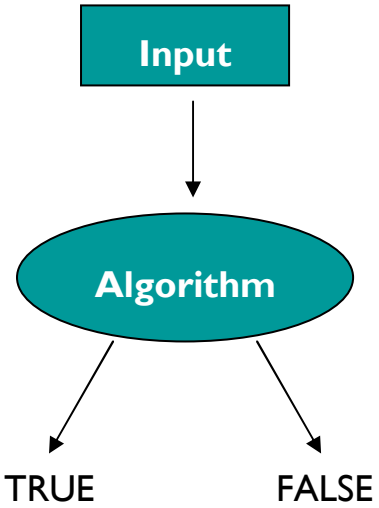
The word *algorism* originally referred only to the rules of performing arithmetic using Arabic numerals but evolved via European Latin translation of al-Khwarizmi's name into *algorithm* by the 18th century. The word evolved to include all definite procedures for solving problems or performing tasks.

“said, I like to know where you got the notion”

People became most interested in the notion of algorithms after David Hilbert and Wilhelm Ackermann put forth the Entscheidungsproblem (German for decision problem) in 1928

The Entscheidungsproblem wanted an algorithm that could take any math problem as an input and tell us if that math problem was solvable or not.

Hilbert believed this was so…



In computer science, an instance of the Entscheidungsproblem would be asking for a program (perhaps a compiler) that could take any other program to let us know in advance if that program runs infinitely (not solvable).

Answer: Rock The Boat by Hues Corporation

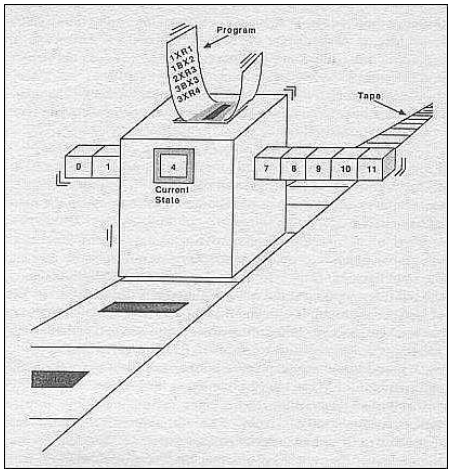
Formal Definition of an Algorithm?

Actually there is no formal definition of an algorithm...

But there has been many attempts

- ❑ Recursive Functions
- ☑ Turing's Machine
- ❑ Unlimited Register Machines (URM's)

A perception of what a Turing machine would be like if it was built.



One such formalization of an algorithm is the Turing Machine model of computability. Turing's machine was brought to existence by Alan Turing in his 1936 paper "On Computable Numbers, with an Application to the Entscheidungsproblem". A Turing machine is a very simple modeled theoretical computer. The hardware of a Turing machine consists of a tape which may extend left and right infinitely, divided into cells for storage and a reading/printing head that moves left and right to process programs. Turing machines demonstrate how computations are carried out mechanically.

Turing's Concept

Turing may have explained it somewhat differently...

Alan Mathison Turing originally envisioned a "computer" to not be the machine we have today, but actually a person who obediently takes instructions and carries out the process accordingly without question. Turing called the different instructions "states" relating to a human's states of mind. No matter the hardware one truth still stands:

Alan Mathison Turing (1912 – 1954)

An algorithm can be carried out on a Turing machine!

We must understand that Turing invented his machine about two decades before the advent of computers. A machine containing the required technology needed to do computations was probably still somewhat vague to Turing. However, the logic of Turing's machine grasped the concepts of hardware and software so well that if we analyze our computing devices carefully enough we can see they are really all Turing machines!

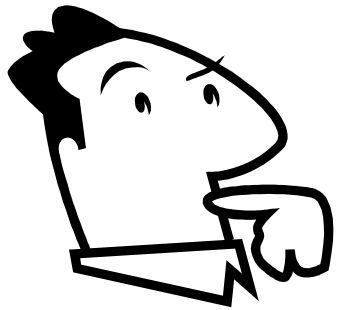
To date all computers and Turing machines are equivalent in terms of computational limits.

From States of Mind to Program Instructions

A Turing machine performs an action based on what it reads from the tape and the state of mind which it is in.

Think about your states of mind. Let's say you are taking a multiple choice test that you studied well for. Your state of mind when taking such a test is very straightforward:

- Read the question.
- Read every choice.
- Select the choice you perceive as "right".
- Move on to the proceeding question.



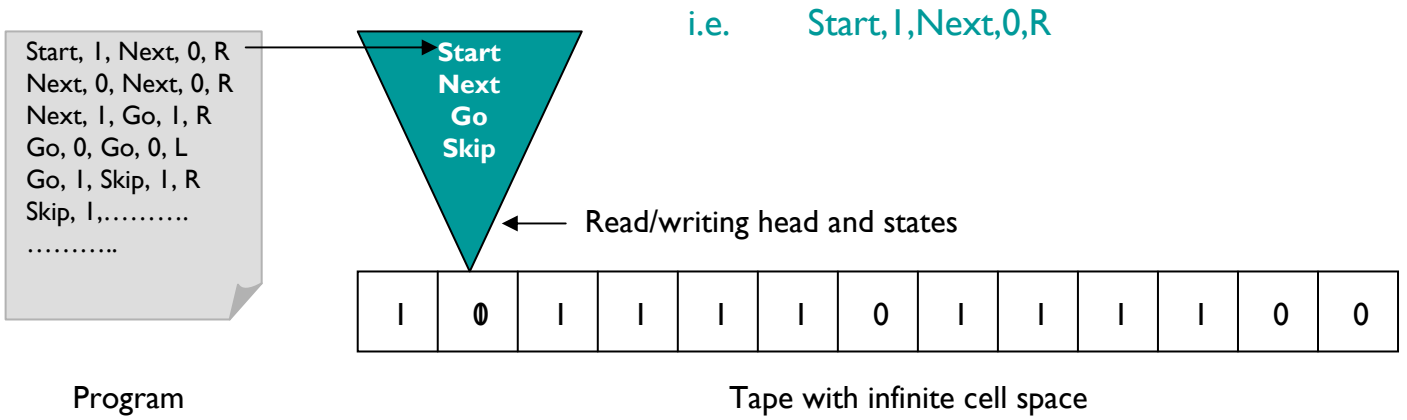
A computer performs an action by what it reads from an input (either user-entered or a program) and what the program's instruction tells it to do at that point.

Programming a Turing Machine

As with any programming language, we must follow a set of rules:

1. Each state/command must be uniquely named, so that the Turing machine head will know what write for each cell in the tape.
2. The states must be structured as so:

   State name, symbol read in current cell (0 or 1), state name for the next cell, what symbol to write in the current cell (0 or 1), and the direction the head should move to read from the tape (R or L)



3. The H state is reserved to halt the machine from doing further computation.
4. The numbers we are reading off the tape are the natural numbers. Since 0 has already been used for the symbol scheme, our zero will read off the tape as 1, one will read as 11, two as 111, and so on. However, after our computations we must ensure that our output still keeps true to our n+1 definitions. The tape always begins filled with a 0 in each cell that is not used by the input entered.

Addition Turing Machine

Addition Programmed

Start 0 -> H 0

Start 1 -> Concatenate 1 R

Concatenate 1 -> Concatenate 1 R

Concatenate 0 -> FindEnd 1 R

FindEnd 1 -> FindEnd 1 R

FindEnd 0 -> Delete1 0 L

Delete1 1 -> Delete2 0 L

Delete2 1 -> Finish 0 L

Finish 1 -> H 1

The input entered by the user is two numbers separated by a 0. The head skips 1's until it hits the 0 breaking the two inputs and fills it with a 1 to concatenate the two inputs into one number. Remember, our inputs are always entered as n+1 and we added an extra 1 when we concatenated the two inputs, thus the head needs to delete the two extra 1's. The head proceeds on skipping 1's until it hits another 0 signaling it has reached the end of the user input. It then moves left to delete the extra 1's.

Multiplication Turing Machine

marktape,_ H,_
marktape,1 marktape2,1,>
marktape2,_ zerocase,_,>
zerocase,1 zerocase,_,>
zerocase,_ H,_
marktape2,1 marktape3,_,>
marktape3,1 marktape3,1,>
marktape3,_ marktape4,1,>
marktape4,1 startmultiply,_
marktape4,_ H,_
startmultiply,_ zerocase2,_,<
zerocase2,_ zerocase2,_,<
zerocase2,1 delete,_,<
delete,_ finish,_,<
delete,1 ,delete,_,<
finish,1 H,_
startmult,1 backtofirstinput,1,<
backtofirstinput,_ backtofirstinput,_,<
backtofirstinput,1 numofcopies,1,<
numofcopies,_ extraone,_,>
extraone,1 extraone,_,>
extraone,_ findend,_,>
findend,1 findend,1,>
findend,_ cleanup,1,<
cleanup,1 cleanup,1,<
cleanup,_ cleanup2,_,<
cleanup2,_,cleanup2,_,<
cleanup2,1 H,_
numofcopies,1 minuscopy,1,<
minuscopy,1 minuscopy,1,<
minuscopy,_ deletecopy,_,>
deletecopy,1 gotomakecopy,_,>
gotomakecopy,1 gotomakecopy,1,>
gotomakecopy,_ readthroughfirstcopy,_,>
readthroughfirstcopy,1 readthroughfirstcopy,1,>
readthroughfirstcopy,_ findfreespace,_,>
findfreespace,_ moveback,1,<
findfreespace,1 readthroughnewcopies,1,>
readthroughnewcopies,1 readthroughnewcopies,1,>
readthroughnewcopies,_ moveback,1,<
moveback,1 moveback,1,<
moveback,_ backtofirstcopy,_,<
backtofirstcopy,1 numofones,1,<
numofones,1 minusone,1,<
minusone,1 minusone,1,<
minusone,_ deteleone,_,>
deleteone,1 readthroughfirstcopy,_,>
numofones,_ refillcopy,1,<
refillcopy,_ refillcopy,1<
refillcopy,1 forward,1,>
forward,1 backtofirstinput,_,<

As we can see, programming something as trivial as multiplication gets lengthy!

Higher-Level Programming for a Turing Machine

We can shorten the lines of code by breaking a simple rule:

Currently we have been programming our Turing machine with a low-level/assembly language convention. We wrote instructions where our states made decisions based on reading/writing 0's and 1's. But let's introduce more symbols and give our states more options to base their decisions on.

Say instead of just 0 and 1, we have a, b, and c where:

a = 11

b = 01

c = 10

And where our instructions now look as so:

State1, a State2, a, R

State1, b State2, a, R

State1, c State3, c, L

State2, a State2, b, R

State2, b State3, a, L

Our original lines of code would have been:

State1, 1 State2, 1, R

State2, 1 State3, 1, R

State1, 0 State2, 1, R

State2, 0 State2, 0, L

State3, 1 State4, 0, R

State4, 1 State4, 1, R

State4, 0 State5, 1, R

State5, 1 State5, 1, L

WARNING: Using more symbols will not increase the bounds of computing!

We will always be able and WILL HAVE TO translate back to the original 0's and 1's to run our programs!

Of course we must be mindful that our tape must be filled with the symbols a, b, or c. So if we wanted to initialize our tape with 1111011000..., we would actually be entering aabc

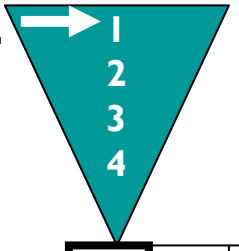
Multiple Tapes for a Turing Machine

Previously one tape was used to carry out all computations and execute an output.

But what if we add more tapes, and give ourselves more to work with:

It also means we would modify our instructions to know what to do for the top tape and what to do for the bottom tape. The structure of our instructions would now be:

<state, symbol read on top tape, symbol written on top tape, symbol read on bottom tape, symbol written on bottom tape, state to switch to, direction>

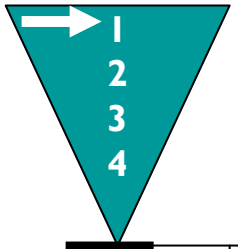


← Head reading two tapes and more symbols

i.e. 1, a, b b, a, 2, R

b	c	b	b	a	a	c	b	b	b
a	a	a	a	c	b	b	b	b	b

WARNING: Using more tapes will not increase the bounds of computing!



← Head reading one tape

a	b	c	a	b	a	b	a	a	c	a	b	c	b	b

If we were using one tape our program would have resembled something of the sort:

1, a 1, b, R

1, b 2, a, R

Universal Turing Machine

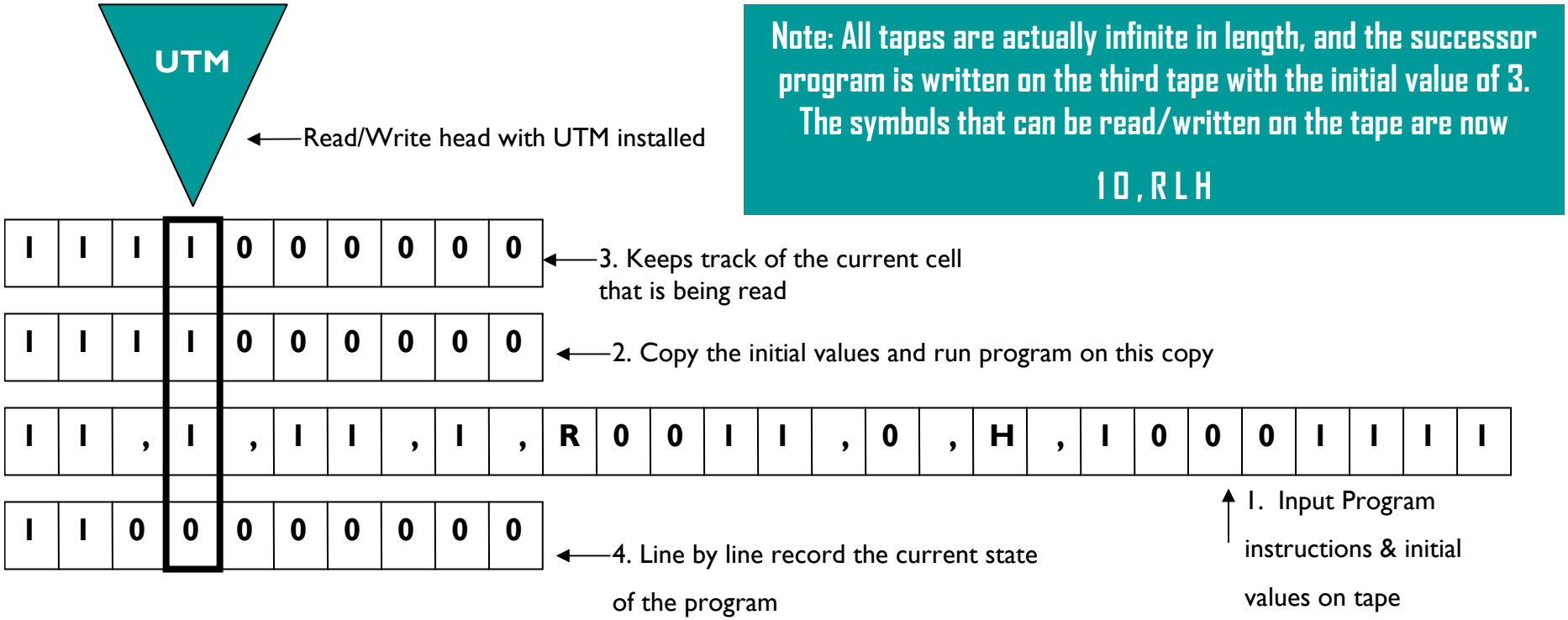
A Universal Turing machine is actually a universal algorithm for Turing machines!

A Universal Turing machine (UTM) algorithm can be thought of as a software for a Turing machine. Previously a Turing machine would take our instructions and would only be able to carry out that one algorithm. But if we think about this in terms of hardware that would become very expensive and not to mention tedious to reprogram a Turing machine over and over. So we need a way that we can enter our Turing machine programs as input and only instruct the machine once to handle any input (program) it is given. Our Universal algorithm does just that. It is a general program that is built into our Turing machine's memory (head) that will be able to process any Turing machine program and it's initial values that we write on the tape.

"Universal Turing Machine" © 2003 Jin Wicked, USA.

The Universal Turing Machine Algorithm

Using an extended Turing machine (more symbols and tape), we can easily organize a Universal Turing machine algorithm.



Note: All tapes are actually infinite in length, and the successor program is written on the third tape with the initial value of 3. The symbols that can be read/written on the tape are now

1 0 , R L H

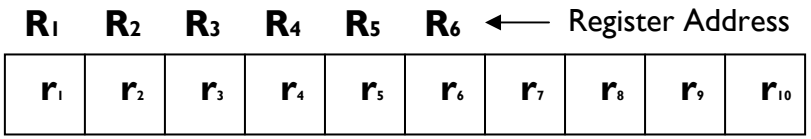
The read/write head and tapes can be compared to a computer (processor and memory). The UTM can be compared to an operating system for our Turing machine. The Turing machine algorithms can be thought of as computer programs and the symbols we use to write our Turing algorithms can be thought of as the programming syntax.

Unlimited Register Machines

After Turing machines, many models of computation shortly followed, In 1963, Shepherdson and Sturgis defined *Unlimited Register Machines*.

URMs are abstract computers that use *registers* R_1, R_2,…much like RAM memory would to store natural numbers r_1,r_2,…

The registers can be compared to the Turing machine's tape. However, registers are identified by a unique address denoted by Rn. Data on the registers can be accessed at *random.* In other words, data can be accessed regardless of its location and do not depend on a reading head or recorder to walk back. Unfortunately, register machines do not realistically portray the work of a computer as does a Turing machine.



***Please note URM's were formalized about a decade after actual computers had already been invented and nearly 30 years from Turing's machine, so it may be expected if some aspects are more realistic.

Unlimited Register Machines have also been known as Random Access Machines (RAM).

Programming the *URMs*

A URM program has the following finite list of instructions:

Type	Symbolism	Effect
Zero	*Z(n)*	$r_n = 0$
Successor	*S(n)*	$r_n = r_n + 1$
Transfer	*T(m,n)*	$r_m = r_n$
Jump	*J(m,n,q)*	If $r_n = r_m$ go to instruction q – else go to next instruction

The instructions in a URM program are followed in order, starting from number 1. Jump is the only instruction that may alter this course depending on the condition entered into the jump. It is possible to force the program to loop by comparing a value in a register against itself using the jump instruct. The program halts when there are no other instructions to follow or if no such instruction number exists.

Addition and Multiplication in URMs

Addition using a URM program

1. J(2,3,5)
2. S(1)
3. S(3)
4. J(1,1,1)

Multiplication using a URM program

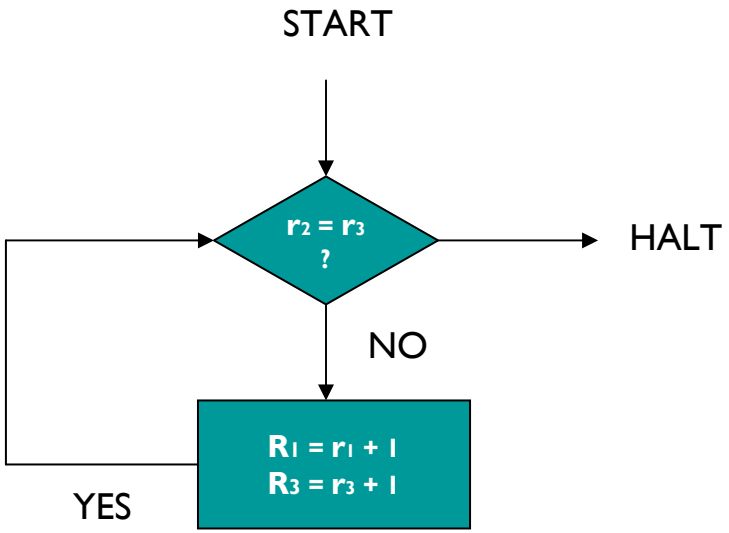
1. T(1,3)
2. Z(1)
3. J(2,4,100)
4. S(4)
5. J(3,5,9)
6. S(1)
7. S(5)
8. J(1,1,5)
9. Z(5)
10. J(1,1,3)

Running URM Addition

Running Addition

R_1	R_2	R_3
[illegible]	2	[illegible]

Flowchart for the addition URM



Addition Program

1. J(2,3,5)
2. S(1)
3. S(3)
4. J(1,1,1)