Computing and Complexity Theory

Chapter 1: Introduction - RAM

J. Roland

QuIC

Quantum Information & Communication

Outline

Outline

INFOH422: 2 Parts

Part Information Theory - Prof. N. Cerf (3 ECTS)

- Theory: Tuesdays 10-12
- Exercises: Mondays 8-10 (2nd half of quadrimester)
- Evaluation: Oral exam (3/5 of total grade for the course)

Part Computability and Complexity Theory - Prof. J. Roland (2 ECTS)

- Lectures on Fridays 14-16
- Evaluation: Oral exam (2/5 of total grade for the course)
  - Open book exam: course material available during answer preparation time

Part Computability and Complexity Theory

Organization

- Ex cathedra lectures
  - Live on campus (Fridays 14-16)
  - Broadcasted and recorded on MS Teams
- Reading assignments + exercises
  - Published on Virtual University: `http://uv.ulb.ac.be`
- Possibility to ask questions
  - In person before or after each lecture
  - By email (avoid MS Teams chat unless invited)

Note

The course is designed for live lectures, broadcasting and recording should be used as backup only (can sometimes fail)

Part Computability and Complexity Theory

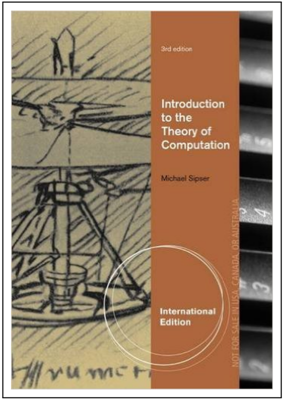
Course material

- Slides, exercises and reading assignments
  - Published on Virtual University: `http://uv.ulb.ac.be`
- Reference books
  - See last slide for correspondence between slides and book sections

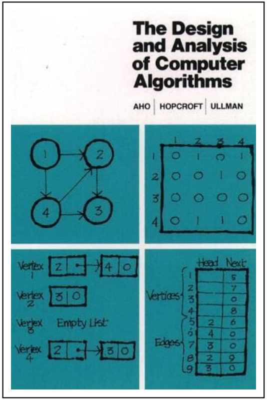
Note

The course slides are supportive material only and do not cover everything that will be examinated!

Reference books



Introduction to the Theory of Computation, M. Sipser
(3rd Edition - International Edition, Cengage Learning, 2013)
Section 3: Turing Machines
Section 7: Time Complexity



The Design and Analysis of Computer Algorithms, A. V. Aho, J. E. Hopcroft and J. D. Ullman (Addison-Wesley, 1974)
Section 1: Models of Computation

Outline

What is this course about?

The **Theory of Computation** is the branch of computer science that deals with how efficiently problems can be solved on **a model of computation**, using **an algorithm**.

Key questions:

- What are the mathematical properties of computer hardware and software?
- What is computation? What is an algorithm?
- What are the limitation of computers? Can everything be computed?
- Are all computations as easy or as difficult as others?

Note

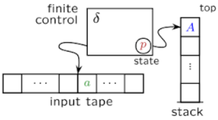
- These are abstract mathematical questions
- Requires a mathematical description of a “computer”
  - Different “Models of computation”

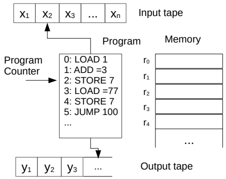
Course structure

Theory of Computation (and this course) is divided into three main branches:

- Models of Computation
- Computability Theory
- Complexity Theory

Models of computation





Models of Computation studies different types of mathematical models capable of describing the properties of real hardware and software.

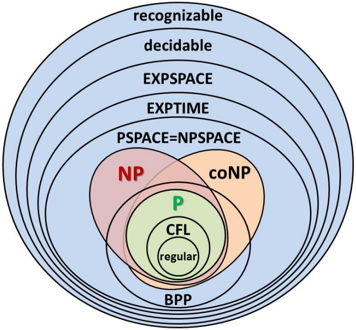
Examples of models:

- Finite State Automaton
- Pushdown Automaton
- Random Access Machine
- Turing Machine

Central questions

- Do these models have the same power, or can one model solve more problems than others? (computability question)
- If they have the same power, are they equally efficient? (complexity question)

Computability theory



Computability theory is concerned with classifying problems into those that are solvable by means of a computer, and those that are unsolvable.

Examples of unsolvable problem:

- Input: a diophantine equation, like

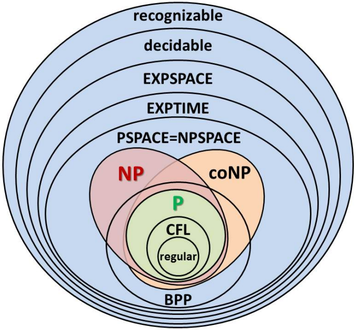
$$6x^3yz^2 + 3xy^2 - x^3 - 10 = 0$$

  in which all coefficients are integer.
- Determine: does the equation have an integer solution?

Central questions

- What does it mean to “solve a problem” on a computer?
- Are there problems we cannot solve with a computer?

Complexity theory



Complexity theory is concerned with classifying problems into those that are easy to solve by means of a computer, and those that are computationally hard.

Measures for computational efficiency: **time used**, or **space used**.

Examples of problems:

- Easy: sort a sequence of numbers; search for a name in a telephone directory.
- Hard: factor a 300-digit number in its prime factors.

Central questions

- What problems can we solve efficiently?
- How do we measure efficiency?
- What problems are “more difficult” than others?

Outline

Standard notions

The following standard mathematical notions are assumed known:

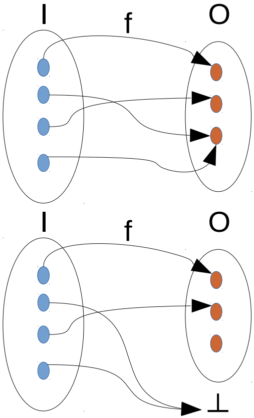
- Sets: unordered collections of elements, and their operations on them: $\cup$, $\cap$, $\setminus$, $\times$, $\in$, ...
- Tuples: ordered sequences of elements, such as $(a, 2, x, 4)$.
- Functions, relations.
- Graphs and trees.

See the book, p 3–13 for a refreshment.

Notation:

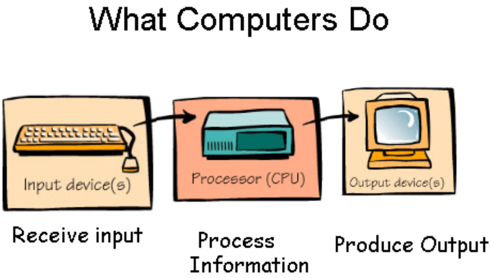
- $\mathbb{N} = \{0, 1, 2, \ldots\}$ is the set of natural numbers (denoted by $\mathcal{N}$ in the book);
- $\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots\}$ the set of integers (denoted by $\mathcal{Z}$ in the book)

Functions vs Partial Functions



- In mathematics, a function $f\colon I \to O$ from a set I to a set O is an object that associates to each $x \in I$ an output $f(x) \in O$.
- Mathematical functions are hence **total functions**.
- In the theory of computation, we will often have to deal with partial functions.
- We will denote partial functions as $f\colon I \to O \cup \{\perp\}$, where $\perp$ is a special symbol used to indicate on which inputs f is undefined (in which case we write $f(x) = \perp$).

Strings and Languages



- To describe what an algorithm does, we will need to describe the kind of input that it takes, and the kind of output that it produces.
- Both the input and the output will usually be modeled as **a string**. The **alphabet** over which the strings are defined may vary with the application.

Strings and Languages

Definition

- An **alphabet** is a set, whose elements are called **symbols**
- A string s over an alphabet Σ is a finite sequence $\sigma_1 \ldots \sigma_k$ of symbols, all in Σ.
- The length $|s|$ of the string $s = \sigma_1 \ldots \sigma_k$ is k.
- There is one string of length 0 (the empty string), which is denoted ε
- The set of all strings over alphabet Σ is denoted by Σ^*.

Example

Σ	example string s over Σ	$\lvert s \rvert$
$\{0, 1\}$	10101110	8
$\{a, b, c, d, r\}$	*abracadabra*	11
$\{0, 1, x, y, +\}$	$x + y + 1 + 0$	7

Strings and Languages

Definition

- A **language** L over alphabet Σ is a set of strings, i.e., it is a subset $L \subseteq \Sigma^*$.

Example

Σ	example language L over Σ
$\{0, 1\}$	$\{\underbrace{1 \ldots 1}_{k \text{ times}} \mid k \geq 0\}$
$\{a, b, c, d, r\}$	$\{asa \mid s \in \Sigma^*\}$

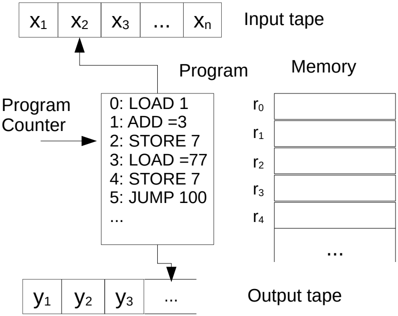
A comment on alphabets

Important remark

- In the theory of computation, it is often assumed and required that alphabets are finite (as in our examples above).
- For the RAM and RASP computation models, however, we will allow an infinite alphabet: namely the set $\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots\}$ of all integers.

Outline

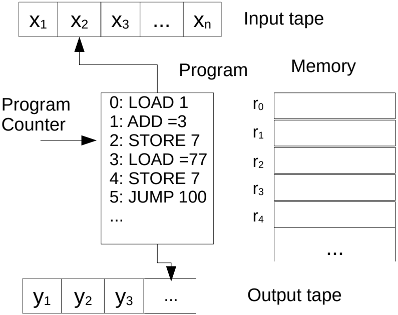
The Random Access Machine



A **Random Access Machine (RAM)**

- models a one-accumulator computer in which the program is fixed (not stored in memory, cannot be changed by the RAM itself)
- consists of
  - an input tape
  - an output tape
  - memory
  - a (fixed) program

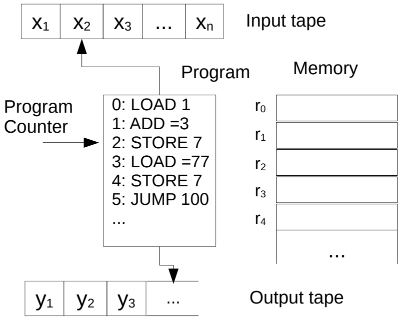
Input tape



Input tape

- Read-only
- Each cell can contain an (arbitrary-length) integer
- Whenever a symbol is read from the input tape, the tape head moves one cell to the right.

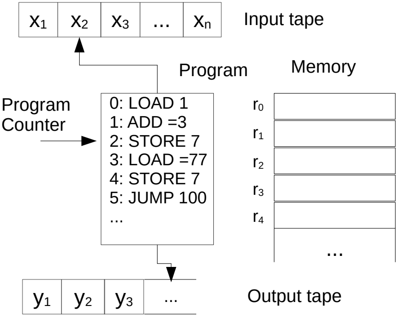
Output tape



Output tape

- Write-only
- Each cell can contain an (arbitrary-length) integer
- Whenever a symbol is written to the output tape, the tape head moves one cell to the right.

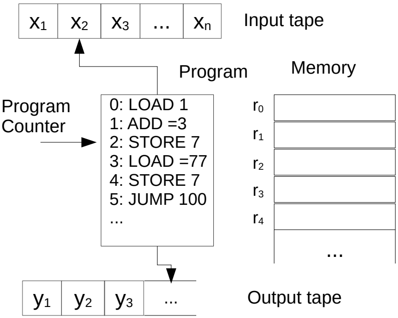
Memory



Memory

- An unbounded number of registers $r_0, r_1, \ldots, r_i, \ldots,$ each of which is capable of holding an integer of arbitrary size.
- Unbounded $\rightarrow$ during execution the RAM can always ask for more memory (but only uses a finite number of registers)
- Register r_0 is the **accumulator**

Fixed program



Fixed Program

- Sequence of instructions.
- An instruction may (optionally) be **labeled**.
- Instruction set:
  - integer operations;
  - reading from input tape / writing to output tape;
  - loading from memory /storing to memory;
  - conditional jump to other instructions

RAM Instructions

Example instruction	**Meaning**
LOAD =50	Set $r_0 = 50$
LOAD 50	Set $r_0 = x$ where $x =$ contents of r_{50}
LOAD $*$50	Set $r_0 = y$ where $y =$ contents of r_z with $z =$ contents of r_{50}. Computation halts if $z \leq 0$.
ADD =50	Set $r_0 = r_0 + 50$
ADD 50	Set $r_0 = r_0 + x$ where $x =$ contents of r_{50}

Definition

An **operand** is a term of the one of the following forms:

1. $=i$, with $i \in \mathbb{Z}$; (constant)
2. i, with $i \in \mathbb{N}$; (direct addressing)
3. $*i$, with $i \in \mathbb{N}$; (indirect addressing, for arrays and pointers)

RAM Instructions

Definition

A **RAM instruction** is one of the following instructions. (More can be added for convenience, without increasing expressive power.)

Operation code	Address
LOAD	operand
STORE	operand (no cst)
ADD	operand
SUB	operand
MULT	operand
DIV	operand
READ	operand (no cst)
WRITE	operand
JUMP	label
JGTZ	label
JZERO	label
HALT	

Example: x^x

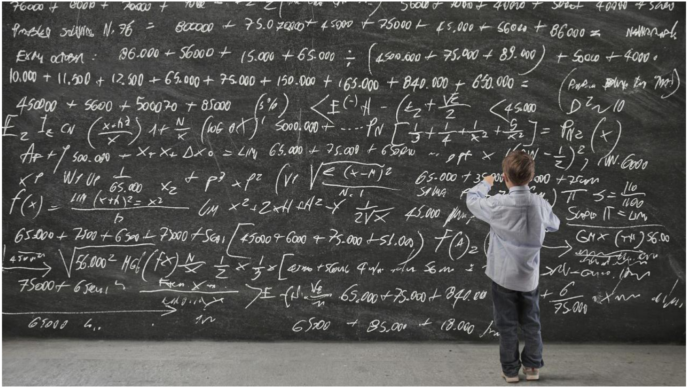
	RAM program		Explanation
	READ	1	$r_1 \leftarrow x$: read 1st integer on input tape into register r_1
	LOAD	1	$r_0 \leftarrow r_1$ load content of register r_1 into accumulator r_0
	JGTZ	pos	if $r_0 > 0$, jump to instruction pos
	WRITE	=0	otherwise, write 0 on output tape
	JUMP	endif	
pos:	LOAD	1	$r_0 \leftarrow r_1$
	STORE	2	$r_2 \leftarrow r_0\ (= r_1)$
	SUB	=1	$r_0 \leftarrow r_0 - 1\ (= r_1 - 1)$
	STORE	3	$r_3 \leftarrow r_0\ (= r_1 - 1)$
while:	LOAD	3	
	JGTZ	continue	
	JUMP	endwhile	while $r_3 > 0$ do
continue:	LOAD	2	
	MULT	1	
	STORE	2	$r_2 \leftarrow r_2 * r_1$
	LOAD	3	
	SUB	=1	
	STORE	3	$r_3 \leftarrow r_3 - 1$
	JUMP	while	
endwhile:	WRITE	2	write r_2 to output tape
endif:	HALT		

Example: language recognition

RAM program to check whether input tape contains as many 1's as other symbols
(input tape is terminated by a 0)

RAM program			Explanation
	LOAD	=0	
	STORE	2	Initialize $r_2 \leftarrow 0$
	READ	1	read first tape symbol, store it in register r_1
while:	LOAD	1	
	JZERO	endwhile	while $r_1 \neq 0$
	LOAD	1	
	SUB	=1	
	JZERO	one	if $r_1 \neq 1$
	LOAD	2	
	SUB	=1	
	STORE	2	then $r_2 \leftarrow r_2 - 1$
	JUMP	endif	
one:	LOAD	2	
	ADD	=1	
	STORE	2	else $r_2 \leftarrow r_2 + 1$
endif:	READ	1	read next tape symbol, store it in register r_1
	JUMP	while	
endwhile:	LOAD	2	
	JZERO	output	if $r_2 = 0$, jump to output
	HALT		
output:	WRITE	=1	else write 1 on output tape
	HALT		

Observation



- A RAM models a physical computer
- But it is amenable to complete formal, mathematical definition and analysis.

Formal definition of a RAM program

	READ	1
	LOAD	1
	JGTZ	pos
	WRITE	=0
	JUMP	endif
pos:	LOAD	1
	STORE	2
	LOAD	1
	SUB	=1
	STORE	3
while:	LOAD	3
	JGTZ	continue
	JUMP	endwhile
continue:	LOAD	2
	MULT	1
	STORE	2
	LOAD	3
	SUB	=1
	STORE	3
	JUMP	while
endwhile:	WRITE	2
endif:	HALT	

Definition

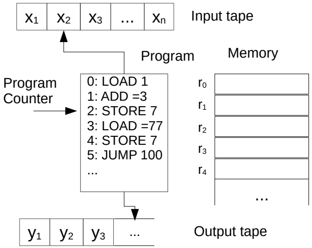
A **RAM** is a sequence $M = (l_1, i_1), \ldots, (l_k, i_k)$ where:

- every i_j is a valid RAM instruction; and
- l_j is the label for instruction i_j. Labels are optional; we set $l_j = \varepsilon$ to indicate that instruction i_j does not have a label. (ε is the empty string)

Well-formedness requirements:

- Every label (except ε) can occur only once in M.
- For every instruction of the form JUMP l, JGTZ l, and JZERO l, there has to be some instruction in M that is labeled by l.

Configuration of a RAM M

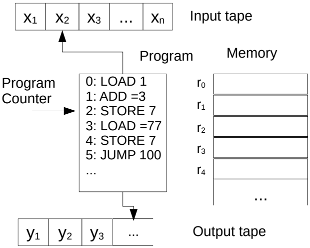


Definition

Let $M = (l_1, i_1), \ldots, (l_k, i_k)$ be a RAM. A **configuration** of M is a 6-tuple (in, j, out, pc, mem, h) where:

- $in = x_1 \ldots x_n \in \mathbb{Z}^*$ is the content of the input tape;
- j is the position of the head on the input tape, $1 \leq j \leq n$;
- $out = y_1 \ldots y_m \in \mathbb{Z}^*$ is the content of the output tape; the head is positioned on output cell $m + 1$
- pc is the program counter; it indicates the current instruction
- $mem\colon \mathbb{N} \to \mathbb{Z}$ is the memory map; $mem(i)$ holds the contents of register r_i, for every $i \in \mathbb{N}$.
- h is a boolean, which is TRUE if the execution has halted, and FALSE otherwise.

Initial configuration of a RAM M

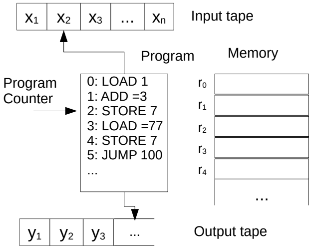


Definition

Let $in = x_1 \dots x_n$ be the input to RAM M. The **initial configuration** (also called **start configuration**) of M on in is the configuration $(in, 1, \varepsilon, 1, mem, \text{FALSE})$:

- $in = x_1 \dots x_n \in \mathbb{Z}^*$ is the content of the input tape;
- The input tape head is at its first cell;
- The output tape is empty (ε is the empty string)
- The program counter is position at the 1st instruction.
- All registers are initialized to 0, i.e., $mem(i) = 0$, for every $i \in \mathbb{N}$.
- Execution has not halted.

Formal definition of a run (1)



Definition

In configuration $C = (in, j, out, pc, mem, h)$ with $h =$ FALSE, the RAM executes the instruction at position pc in the program, **which yields a new configuration** $C' = (in, j', out', pc', mem', h')$.[1]

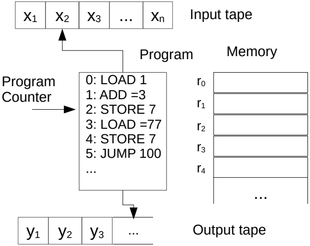
Example

$M[pc]$	C' equals C except that
LOAD =10	$mem'(0) := 10$; $pc' := pc + 1$
STORE i	$mem'(i) := mem(0)$; $pc' := pc + 1$
STORE $*i$	$mem'(mem(i)) := mem(0)$; $pc' := pc + 1$
WRITE =5	$out' := out \cdot 5$; $pc' := pc + 1$
READ 5	$mem'(5) := x_j$; $j' = j + 1$; $pc' := pc + 1$

Notation: we write $C \vdash C'$ to indicate that configuration C yields configuration C' in one step, and $C \vdash^* C'$ to indicate that there exists a sequence of configurations $C_1, \ldots, C_k$ with $k \geq 0$ such that $C \vdash C_1 \vdash \cdots \vdash C_k = C'$

[1] If $h =$ TRUE, execution has stopped, and no new configuration can be yielded.

Formal definition of a run (2)



Definition

- A **halting configuration** is a configuration $C = (in, j, out, pc, mem, h)$ where $h = \text{TRUE}$.[2]
- Let *in* be the input to RAM M, and let C_0 be the initial configuration of M on *in*. When M starts computing starting from C_0, there are two things that can happen:
  1. M reaches a halting configuration C_h such that $C_0 \vdash^* C_h$ (after a finite number of steps); or
  2. M keeps computing for ever.

[2]Note that such a configuration cannot yield any new configuration.

Partial function defined by a RAM

Definition

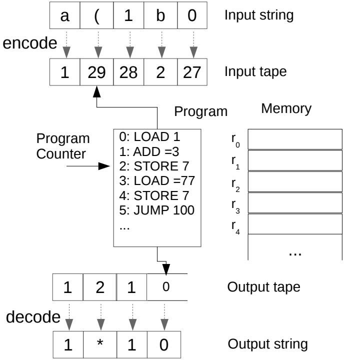
A **RAM M defines a partial function, denoted $[\![M]\!]$,** that maps input tapes to output tapes, i.e.,

$$[\![M]\!] : \mathbb{Z}^* \rightarrow \mathbb{Z}^* \cup \{\perp\}$$

- If, started on the initial configuration for *in*, M reaches a halting configuration C_h, then $[\![M]\!](in) = out$, where *out* is the contents of the outputtape in configuration C_h;
- If M never reaches a halting configuration, then $[\![M]\!](in) = \perp$.
- So only <u>halting</u> computations define a valid output.

Encoding and decoding functions

Question:
What if our input is not a sequence of integers, but a string from some alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ and the output is supposed to be some string over $\Gamma = \{\gamma_1, \ldots, \gamma_l\}$?



- Encode each input alphabet symbol $\sigma \in \Sigma$ as an integer $enc(\sigma)$. Run M on the encoded input.
- Once the machine halts, decode each integer i on the output tape as a symbol $dec(i) \in \Gamma$.

Encoding of Σ	Decoding of Γ
$a \mapsto 1$	$0 \mapsto 0$
$b \mapsto 2$	$1 \mapsto 1$
$c \mapsto 3$	$2 \mapsto *$
$\ldots$	$\ldots$
$0 \mapsto 27$	
$1 \mapsto 28$	
$(\mapsto 29$	

Given the encoding function $enc\colon \Sigma \to \mathbb{Z}$ and decoding function $dec\colon \mathbb{Z} \to \Gamma$, M hence also defines a partial function from $\Sigma^* \to \Gamma^* \cup \{\bot\}$.

Side note on looping

Important remark

- It may seem that a RAM that loops on an input just means that it has a faulty program.
- As we will later see, however, there are problems that cause all programs that aim to solve the problem to loop on some inputs!

Computational complexity of a RAM program

We'd also like to know how efficiently a RAM M computes, in terms of the size of the input $in \in \mathbb{Z}^*$.

Question: What is a reasonable definition of the time/space used by M on input $in \in \mathbb{Z}^*$?

We will look at two cost models:

- **Uniform cost model**
- **Logarithmic cost model**

Uniform cost model: time

Key Idea: each RAM instruction takes one unit of time to execute.

Definition

We define ***time***$(M,$ ***in***$)$ **to be the number of instructions that** M **executes on input *in***:

- If M halts on in, then M goes through a sequence of configurations $\rho = C_1, \ldots, C_k$ where C_1 is the initial configuration of M on in, and C_k is a halting configuration. In this case, $time(M, in) = k$.
- If M does not halt on in, $time(M, in) = \infty$.

Definition

The **(worst case) time complexity of a RAM** M **under the uniform cost model** is the function $f : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ such that $f(n) = \max\{time(M, in) \mid in \in \mathbb{Z}^*, |in| = n\}$.[3]

Note that the time complexity is ∞ as soon as there is one input on which M does not terminate. So it makes sense to look only at programs that terminate on all inputs.

[3] Recall that for a string in, the notation $|in|$ denotes the length of the string.

Uniform cost model: space

Key Idea: each register takes unit space to store. We only count registers that store a value other than 0.

Definition

- We consider a register r_i to be used in configuration $C = (in, j, out, pc, mem, h)$ if $mem(i) \neq 0$. The number of registers used in configuration C is denoted $space(C)$.
- We define ***space***$(M, \textbf{\textit{in}})$ **to be the number of registers that M uses throughout its execution on input *in***:
    - If M halts on in, then M goes through a sequence of configurations $\rho = C_1, \ldots, C_k$ where C_1 is the initial configuration of M on in, and C_k is a halting configuration. In this case, $space(M, in) = \max_{1 \leq i \leq k} space(C_i)$.
    - If M does not halt on in, $space(M, in) = \infty$.

Definition

The **(worst case) space complexity of a RAM M under the uniform cost model** is the function $f : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ such that $f(n) = \max\{space(M, in) \mid in \in \mathbb{Z}^*, |in| = n\}$.

Uniform cost model: language recognition

What is the time/space cost of the language recognition RAM in the uniform cost model?

RAM program			Explanation
	LOAD	=0	
	STORE	2	Initialize $r_2 \leftarrow 0$
	READ	1	read first tape symbol, store it in register r_1
while:	LOAD	1	
	JZERO	endwhile	while $r_1 \neq 0$
	LOAD	1	
	SUB	=1	
	JZERO	one	if $r_1 \neq 1$
	LOAD	2	
	SUB	=1	
	STORE	2	then $r_2 \leftarrow r_2 - 1$
	JUMP	endif	
one:	LOAD	2	
	ADD	=1	
	STORE	2	else $r_2 \leftarrow r_2 + 1$
endif:	READ	1	read next tape symbol, store it in register r_1
	JUMP	while	
endwhile:	LOAD	2	
	JZERO	output	if $r_2 = 0$, jump to output
	HALT		
output:	WRITE	=1	else write 1 on output tape
	HALT		

Uniform cost model: language recognition (analysis)

What is the time/space cost of the language recognition RAM in the uniform cost model?

Time:

- On an input *in* of length n, the RAM will only do a constant amount of instructions per input tape symbol. Hence $time(M, in) \leq k \times n + l$ for some constants k and l.
- Therefore, the worst-case complexity is $O(n)$.

Space:

- On an input *in* of length n, the RAM will only use registers r_0, r_1, r_2. Hence $space(M, in) \leq 3$.
- Therefore, the worst-case space complexity is $O(1)$.

Uniform cost model: x^x

What is the time/space cost of the RAM computing x^x in the uniform cost model?

	RAM program		Explanation
	READ	1	$r_1 \leftarrow x$: read 1st integer on input tape into register r_1
	LOAD	1	$r_0 \leftarrow r_1$ load content of register r_1 into accumulator r_0
	JGTZ	pos	if $r_0 > 0$, jump to instruction pos
	WRITE	=0	otherwise, write 0 on output tape
	JUMP	endif	
pos:	LOAD	1	$r_0 \leftarrow r_1$
	STORE	2	$r_2 \leftarrow r_0 \ (= r_1)$
	SUB	=1	$r_0 \leftarrow r_0 - 1 \ (= r_1 - 1)$
	STORE	3	$r_3 \leftarrow r_0 \ (= r_1 - 1)$
while:	LOAD	3	
	JGTZ	continue	
	JUMP	endwhile	while $r_3 > 0$ do
continue:	LOAD	2	
	MULT	1	
	STORE	2	$r_2 \leftarrow r_2 * r_1$
	LOAD	3	
	SUB	=1	
	STORE	3	$r_3 \leftarrow r_3 - 1$
	JUMP	while	
endwhile:	WRITE	2	write r_2 to output tape
endif:	HALT		

Uniform cost model: x^x (analysis - time)

What is the time/space cost of the RAM computing x^x in the uniform cost model?

Time:

- On an input *in* which consist of a single cell containing the natural number x, the RAM will do $x - 1$ iterations of the while loop. It executes a constant number of operations per iteration. Hence $time(M, in) \leq k \times x + l$ for some constants k and l.
- Note, that here, *in* always consists of a single cell.
  - Therefore, we have $|in| = 1 \; \forall x$
- This means that for the worst-case time complexity f of M, $f(1) = \max_x(k \times x + l) = \infty$.

Observation

The uniform cost model seems ill-defined if we measure the input in terms of the number of integers that it contains.

Uniform cost model: x^x (analysis - space)

What is the time/space cost of the RAM computing x^x in the uniform cost model?

Note:

- Note: in practice, the input could have length $n = |in| > 1$ (more than one register used in the input tape).
- In that case, registers beyond the first one are just ignored by the program.
- Therefore, we actually have $f(n) = \infty \; \forall n$.

Space:

- On any input *in* (of any length), the RAM will only use registers r_0, r_1, r_2, and r_3. Hence $space(M, in) \leq 4$.
- Therefore, the worst-case space complexity is $O(1)$.

Uniform cost model: discussion

Question: Is the uniform cost model realistic?

RAMs:

- Each RAM register can hold an integer of arbitrary length.
- The uniform cost model hence assumes that
  1. ADD/MUL/SUB/DIV of arbitrary-length integers is possible in unit time; and
  2. storing arbitrary length integers takes unit space; and (consequently)
  3. that an input tape of length n has size n.

Real machines:

- Can only store integers of bounded length in their registers (the word size, typically 64 bits).
- Inputs are counted in terms of number of words, not number of ints
- ADD/MUL/SUB/DIV integers of length more than 64 bits requires several instructions.

Conclusion

The uniform cost model will only accurately predict time/space usage if the input/registers used by M store only integers whose size is no longer than the real machines word size.

Logarithmic cost model: size of an operand

Key Idea: the time required to execute a RAM instruction is proportional to the size of its operands.

Definition

Let $i \in \mathbb{Z}$ be an integer. Let $|i|$ denote the absolute value of i. The size of i is the number of bits we require to represent integer i:

$$size(i) = \begin{cases} \lfloor \log |i| \rfloor + 1 & \text{if } i \neq 0 \\ 1 & \text{otherwise} \end{cases}$$

Logarithmic cost model: cost of an instruction

Key Idea: the time required to execute a RAM instruction is proportional to the size of its operands.

Definition

Let M be a RAM and suppose that $C = (in = x_1 \dots x_k, j, out, pc, mem, h)$ is the current configuration of M, with $h =$ FALSE. Under the logarithmic model, the **cost $cost(C)$ for computing the next configuration from C** is determined as follows. For a halting configuration, $time(C) = 0$.

Instruc $M[pc]$	Cost $cost(C)$
LOAD op	$cost(\text{op})$
STORE i	$size(mem(0)) + size(i)$
STORE $*i$	$size(mem(0)) + size(i)$ $+size(mem(i))$
ADD op	$size(mem(0)) + cost(\text{op})$
SUB op	$size(mem(0)) + cost(\text{op})$
MULT op	$size(mem(0)) + cost(\text{op})$
DIV op	$size(mem(0)) + cost(\text{op})$
READ i	$size(x_j) + size(i)$
READ $*i$	$size(x_j) + size(i)$ $+size(mem(i))$
WRITE op	$cost(\text{op})$
JUMP lab	1
JGTZ lab	$size(mem(0))$
JZERO lab	$size(mem(0))$
HALT	1

Operand op	Cost $cost(\text{op})$
$=i$	$size(i)$
i	$size(i) + size(mem(i))$
$*i$	$size(i) + size(mem(i))$ $+size(mem(mem(i)))$

Logarithmic cost model: time

Key Idea: the time required to execute a RAM instruction is proportional to the size of its operands.

Definition

Under the logarithmic cost model, we define ***time*(M, *in*), the time that M takes when started on *in*** as follows:

- If M halts on *in*, then M goes through a sequence of configurations $\rho = C_1, \ldots, C_k$ where C_1 is the initial configuration of M on *in*, and C_k is a halting configuration. In this case, $time(M, in) = \sum_{i=1}^{k} cost(C_i)$.
- If M does not halt on *in*, $time(M, in) = \infty$.

Definition

- If $in = x_1 \ldots x_k \in \mathbb{Z}^*$, then define $size(in) = \sum_{i=1}^{k} size(x_i)$.
- The **(worst case) time complexity of a RAM M under the logarithmic cost model** is the function $f : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ such that $f(n) = \max\{time(M, in) \mid in \in \mathbb{Z}^*, size(in) = n\}$.

Logarithmic cost model: space

Key Idea: storing an integer i in register r takes $size(i)$ space to store. We only count registers that store a value other than 0.

Definition

- We consider a register r_i to be used in configuration $C = (in, j, out, pc, mem, h)$ if $mem(i) \neq 0$.
- Hence, the space occupied in configuration C is $space(C) := \sum_{i, mem(i) \neq 0} size(mem(i))$.
- We define $space(M, in)$ to be the maximum amount of space used over all configurations that M goes through when started on in:
    - If M halts on in, then M goes through a sequence of configurations $\rho = C_1, \ldots, C_k$ where C_1 is the initial configuration of M on in, and C_k is a halting configuration. In this case, $space(M, in) = \max_{1 \leq i \leq k} space(C_i)$.
    - If M does not halt on in, $space(M, in) = \infty$.

Definition

The **(worst case) space complexity of a RAM M under the logarithmic cost model** is the function $f : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ such that $f(n) = \max\{space(M, in) \mid in \in \mathbb{Z}^*, size(in) = n\}$.

Logarithmic cost model: language recognition

What is the time/space cost of the language recognition RAM in the logarithmic cost model?

RAM program			Explanation
	LOAD	=0	
	STORE	2	Initialize $r_2 \leftarrow 0$
	READ	1	read first tape symbol, store it in register r_1
while:	LOAD	1	
	JZERO	endwhile	while $r_1 \neq 0$
	LOAD	1	
	SUB	=1	
	JZERO	one	if $r_1 \neq 1$
	LOAD	2	
	SUB	=1	
	STORE	2	then $r_2 \leftarrow r_2 - 1$
	JUMP	endif	
one:	LOAD	2	
	ADD	=1	
	STORE	2	else $r_2 \leftarrow r_2 + 1$
endif:	READ	1	read next tape symbol, store it in register r_1
	JUMP	while	
endwhile:	LOAD	2	
	JZERO	output	if $r_2 = 0$, jump to output
	HALT		
output:	WRITE	=1	else write 1 on output tape
	HALT		

Logarithmic cost model: language recognition (space)

What is the time/space cost of the language recognition RAM in the logarithmic cost model?

Setup

- Consider input $in = x_1 \dots x_k$, and denote by
  - $n = size(in) = \sum_{i=1}^{k} size(x_i)$
  - $m = \max_{1 \leq i \leq k} size(x_i)$

Space:

- On an input $in = x_1 \dots x_k$, the RAM will only store in its registers
  - symbols x_i from the input tape, such that $size(x_i) \leq m \leq n$
  - a counter with integer value between $-k$ and k, with size less than $size(k) = O(\log k) = O(\log n)$ (where we have used the fact that $n \geq k$).
- Therefore, the RAM only stores integers of size at most $\max(m, size(k)) \leq n$.
- The RAM will only use registers r_0, r_1, r_2. Hence $space(M, in) \leq 3 \times n$.
- Therefore, the worst-case space complexity is $O(n)$.

Logarithmic cost model: language recognition (time)

What is the time/space cost of the language recognition RAM in the logarithmic cost model?

Setup

- Consider input $in = x_1 \dots x_k$, and denote by
  - $n = size(in) = \sum_{i=1}^{k} size(x_i)$
  - $m = \max_{1 \le i \le k} size(x_i)$

Time:

- The while loop is executed k times. During iteration number i, the RAM only stores integer x_i (or $x_i - 1$) and a counter of size at most $size(k)$.
- Since none of the instructions concern indirect addressing, each instruction hence takes time $O(\max(size(x_i), size(k)))$.
- Each iteration of the loop executes a constant number of instructions, which all take time $O(\max(size(x_i), size(k)))$ to execute.
- Therefore, $time(M, in) = O(\sum_{i=1}^{k} \max(size(x_i), size(k))) = O(\max(n, k \times size(k))) = O(n \log(n))$ (where we used the facts that $\sum_{i=1}^{k} size(x_i) = n$ and $k \le n$)
- Therefore, the worst-case time complexity is $O(n \log(n))$.

Logarithmic cost model: x^x

What is the time/space cost of the RAM that computes x^x

Logarithmic cost model: discussion

Question: Is the logarithmic cost model realistic?

RAMs:

- The logarithmic cost model assumes that
  1. ADD/MUL/SUB/DIV of integers x is possible in time $O(\mathit{size}(x)) = O(\log(x))$

Real machines:

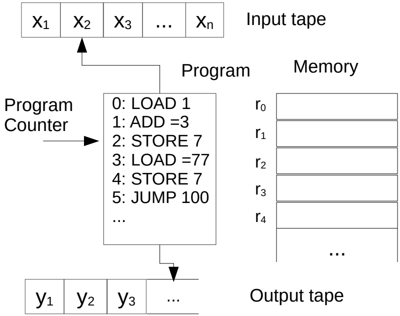
- Can do ADD/SUB of integers x in time proportional to $O(\log(x))$
- But for MUL/DIV, this is not necessarily realistic.

Conclusion

- Even the logarithmic cost model is an approximation of reality.
- However, most of the time it is accurate enough.

Outline

The Random Access Machine

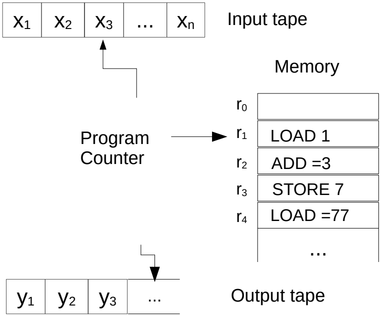


- A **Random Access Machine (RAM)** models a one-accumulator computer in which the program is fixed (not stored in memory, cannot be changed by the RAM itself).
- **Real computers** also store the program in memory, and programs can actually modify themselves.

Question

Are machines that store programs in memory, and can modify programs, more powerful than RAMs with fixed programs?

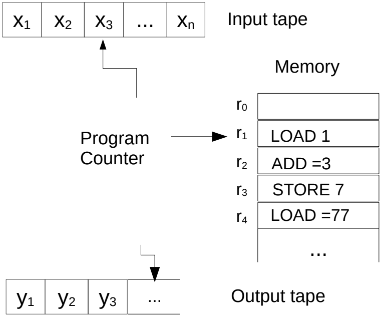
The Random Access Stored Program Machine



A **Random Access Stored Program Machine (RASP)**

- models a one-accumulator computer in which the program is stored in memory, and where instructions can be changed during runtime.
- consists of
  - an input tape
  - an output tape
  - memory
  - a program in memory

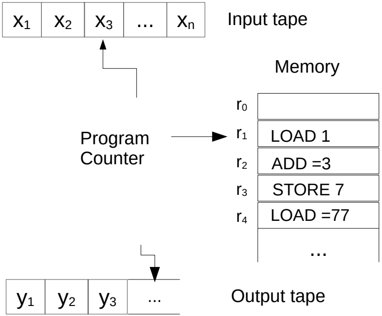
Input tape



Input tape (same as RAM)

- Read-only
- Each cell can contain an (arbitrary-length) integer
- Whenever a symbol is read from the input tape, the tape head moves one cell to the right.

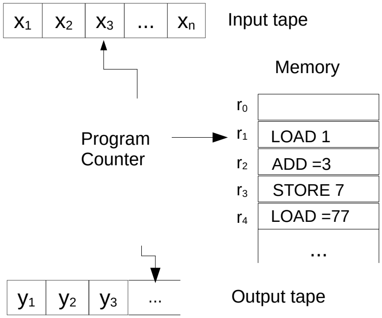
Output tape



Output tape (same as RAM)

- Write-only
- Each cell can contain an (arbitrary-length) integer
- Whenever a symbol is written to the output tape, the tape head moves one cell to the right.

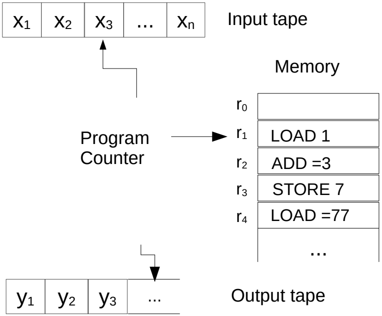
Memory



Memory (same as RAM)

- An unbounded number of registers $r_0, r_1, \ldots, r_i, \ldots,$ each of which is capable of holding an integer of arbitrary size.
- Unbounded → during execution the RAM can always ask for more memory (but only uses a finite number of registers)
- Register r_0 is the **accumulator**

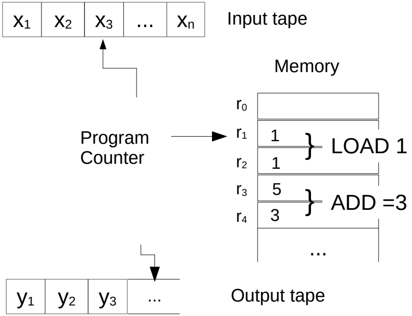
Program in memory



Program in memory (different from RAM)

- Instructions and their operands are represented as integers, and stored in registers.
- Instructions are not labeled; jumps arguments are register numbers, not labels.
- Indirect addressing is not possible; as we will see this is not necessary.

The Random Access Stored Program Machine



How do we encode instructions and their operands?

- Each instruction gets encoded as an integer, and is put in a register, say r_j
- The operand to the instruction is an integer, and put in register r_{j+1}.
- So one instruction takes two registers to encode.

Instruction	Encoding	Instruction	Encoding
LOAD i	1	DIV i	10
LOAD $= i$	2	DIV $= i$	11
STORE i	3	READ i	12
ADD i	4	WRITE i	13
ADD $= i$	5	WRITE $= i$	14
SUB i	6	JUMP $= i$	15
SUB $= i$	7	JGTZ $= i$	16
MULT i	8	JZERO $= i$	17
MULT $= i$	9	HALT	18

Formal definition of a RASP program

```
READ    1
LOAD    1
JGTZ    =6
WRITE   =0
JUMP    =22
LOAD    1
STORE   2
LOAD    1
SUB     =1
STORE   3
LOAD    3
JGTZ    =14
JUMP    =21
LOAD    2
MULT    1
STORE   2
LOAD    3
SUB     =1
STORE   3
JUMP    =11
WRITE   2
HALT
```

Definition

A **RASP** program is a sequence $R = i_1, \ldots, i_k$ where every i_j is a valid RASP instruction[a]

[a] Remember: there are no labels and indirect addressing is not allowed

Configuration of a RASP R

```
READ    1
LOAD    1
JGTZ    =6
WRITE   =0
JUMP    =22
LOAD    1
STORE   2
LOAD    1
SUB     =1
STORE   3
LOAD    3
JGTZ    =14
JUMP    =21
LOAD    2
MULT    1
STORE   2
LOAD    3
SUB     =1
STORE   3
JUMP    =11
WRITE   2
HALT
```

Definition

A **configuration** of R is a 6-tuple (in, j, out, pc, mem, h):

- $in = x_1 \dots x_n$ is the contents of the input tape;
- j is the position of the head on the input tape, $1 \leq j \leq n$;
- $out = y_1 \dots y_m$ is the contents of the output tape; the head is positioned on output cell $m + 1$
- pc is the program counter; it indicates the number of the register holding the current instruction
- $mem \colon \mathbb{N} \to \mathbb{Z}$ is the memory map; $mem(i)$ holds the contents of register r_i, for every $i \in \mathbb{N}$.
- h is a boolean, which is TRUE if the execution has halted, and FALSE otherwise.

Initial configuration of a RASP

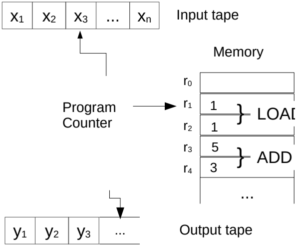
```
READ    1
LOAD    1
JGTZ    =6
WRITE   =0
JUMP    =22
LOAD    1
STORE   2
LOAD    1
SUB     =1
STORE   3
LOAD    3
JGTZ    =14
JUMP    =21
LOAD    2
MULT    1
STORE   2
LOAD    3
SUB     =1
STORE   3
JUMP    =11
WRITE   2
HALT
```

Definition

Let $in = x_1 \dots x_n$ be the input to RASP R. The **initial configuration** of R on in is the configuration $(in, 1, \varepsilon, 1, mem, \mathrm{FALSE})$:

- $in = x_1 \dots x_n$ is the contents of the input tape;
- The input tape head is at its first cell;
- The output tape is empty (ε is the empty string)
- The program counter is positioned at the 1st instruction.
- All registers are initialized to 0, i.e., $mem(i) = 0$; subsequently the program R is loaded in the registers, starting at register r_1.
- Execution has not halted.

Formal definition of a run (1)

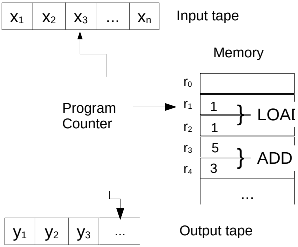


Definition (same as RAM)

The relation $C \vdash C'$ can be defined for RASP programs similarly to how we defined it for RAM programs:

- In configuration $C = (in, j, out, pc, mem, h)$ with $h =$ FALSE, the RASP executes the instruction encoded in register r_{pc} and r_{pc+1} in the program, **which yields a new configuration** $C' = (in, j', out', pc', mem', h')$.

Formal definition of a run (2)



Definition (same as RAM)

- A **halting configuration** is a configuration $C = (in, j, out, pc, mem, h)$ where $h = \text{TRUE}$.[4]
- Let *in* be the input to RASP R, and let C_0 be the initial configuration of R on *in*. When R starts computing starting from C_0, there are two things that can happen:
  1. R reaches a halting configuration C_h such that $C_0 \vdash^* C_h$ (after a finite number of steps); or
  2. R keeps computing for ever.

[4]Note that such a configuration cannot yield any new configuration.

Partial function defined by a RASP

Definition (same as RAM)

A **RASP R defines a partial function, denoted $[\![R]\!]$,** that maps input tapes to output tapes, i.e.,

$$[\![R]\!] : \mathbb{Z}^* \to \mathbb{Z}^* \cup \{\perp\}$$

- If, started on the initial configuration for *in*, R reaches a halting configuration C_h, then $[\![M]\!](in) = out$, where *out* is the contents of the outputtape in configuration C_h;
- If R never reaches a halting configuration, then $[\![R]\!](in) = \perp$.
- So only halting computations define a valid output.

Time and space complexity of RASP programs

We'd also like to know how efficiently a RASP R computes, in terms of the size of the input *in*, with $in \in \mathbb{Z}^*$.

Question: What is a reasonable definition of the time/space used by R on input $in \in \mathbb{Z}^*$?

Two models are possible, similar to RAM:

- **Uniform cost model** → exactly same as for RAM
- **Logarithmic cost model** → similar to RAM, account for accessing instruction.

Logarithmic cost model for RASP: cost of an instruction

Key Idea: the time required to execute a RASP instruction is proportional to the size of its operands. We also account for accessing the (2) register(s) that hold the instruction.

Definition

Let R be a RASP and suppose that $C = (in = x_1 \ldots x_k, j, out, pc, mem, h)$ is the current configuration of R, with $h =$ FALSE. Under the logarithmic model, the **cost $cost(C)$ for computing the next configuration from C** is determined as follows. For a halting configuration, $cost(C) = 0$.

Instruc	Cost $cost(C)$
LOAD op	$size(pc) + cost(\text{op})$
STORE i	$size(pc) + size(mem(0)) + size(i)$
ADD op	$size(pc) + size(mem(0)) + cost(\text{op})$
SUB op	$size(pc) + size(mem(0)) + cost(\text{op})$
MULT op	$size(pc) + size(mem(0)) + cost(\text{op})$
DIV op	$size(pc) + size(mem(0)) + cost(\text{op})$
READ i	$size(pc) + size(x_j) + size(i)$
WRITE op	$size(pc) + cost(\text{op})$
JUMP $=i$	$size(pc) + cost(=i)$
JGTZ $=i$	$size(pc) + size(mem(0)) + cost(=i)$
JZERO $=i$	$size(pc) + size(mem(0)) + cost(=i)$
HALT	$size(pc) + 1$

Operand op	Cost $cost(\text{op})$
$=i$	$size(i)$
i	$size(i) + size(mem(i))$

Logarithmic cost model for RASP: time

Key Idea: the time required to execute a RASP instruction is proportional to the size of its operands. We also account for accessing the (2) register(s) that hold the instruction.

Definition (same as RAM)

Under the logarithmic cost model, we define ***time*(R, in), the time that R takes when started on *in*** as follows:

- If R halts on in, then R goes through a sequence of configurations $\rho = C_1, \ldots, C_k$ where C_1 is the initial configuration of M on in, and C_k is a halting configuration. In this case, $time(R, in) = \sum_{i=1}^{k} cost(C_i)$.
- If R does not halt on in, $time(R, in) = \infty$.

Definition (same as RAM)

- If $in = x_1 \ldots x_k \in \mathbb{Z}^*$,then define $size(in) = \sum_{i=1}^{k} size(x_i)$.
- The **(worst case) time complexity of a RASP R under the logarithmic cost model** is the function $f\colon \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ such that $f(n) = \max\{time(R, in) \mid in \in \mathbb{Z}^*, size(in) = n\}$.

Logarithmic cost model for RASP: space

Key Idea: storing an integer i in register r takes $size(i)$ space to store. We only count registers that store a value other than 0.

Definition (same as RAM)

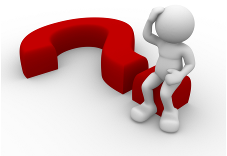
- We consider a register r_i to be used in configuration $C = (in, j, out, pc, mem, h)$ if $mem(i) \neq 0$.
- Hence, the space occupied in configuration C is $space(C) := \sum_{i, mem(i) \neq 0} size(mem(i))$.
- We define ***space***$(R, \textit{in})$ to be the maximum amount of space used over all configurations that R goes through when started on in:
  - If R halts on in, then R goes through a sequence of configurations $\rho = C_1, \ldots, C_k$ where C_1 is the initial configuration of R on in, and C_k is a halting configuration. In this case, $space(R, in) = \max_{1 \leq i \leq k} space(C_i)$.
  - If R does not halt on in, $space(R, in) = \infty$.

Definition (same as RAM)

The **(worst case) space complexity of a RASP R under the logarithmic cost model** is the function $f : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ such that $f(n) = \max\{space(R, in) \mid in \in \mathbb{Z}^*, size(in) = n\}$.

Outline

RAM vs RASP



What is the relationship between the class of partial functions definable by means of RAMs and the class of partial functions definable by means of RASPs?

- Are all functions definable by a RASP also definable by a RAM?
- Are functions that are definable by both "more efficient" on a RAM than on a RASP?

Theorem

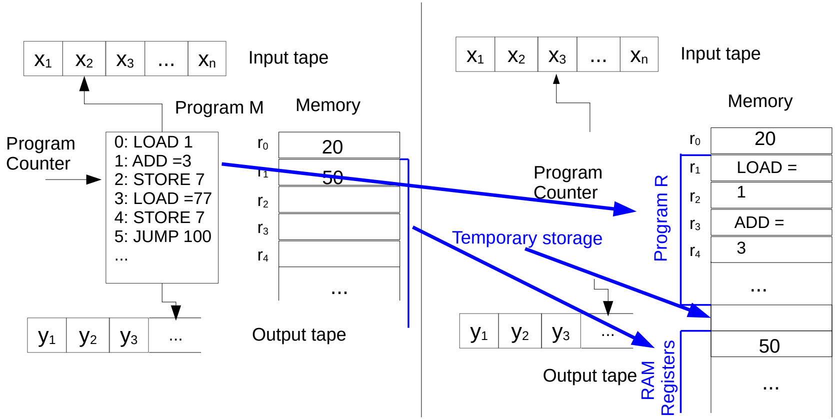
For every RAM M there exists a RASP R such that $[\![M]\!] = [\![R]\!]$. Moreover, there exist constants $k, l, c, d \in \mathbb{N}$ such that for all $in \in \mathbb{Z}^*$ on which M halts we have:

$$time(R, in) \leq k \times time(M, in) + c$$

$$space(R, in) \leq l \times space(M, in) + d$$

This holds under both the uniform and logarithmic cost model.

Simulating a RAM by means of a RASP



Given M, we will construct R so that it simulates M as follows:

- M's accumulator register r_0 will be simulated by means of R's accumulator register r_0.
- The actual program of R will be stored in registers r_1 until r_s, where s is the size of (the encoding of) program R
- We will use register r_{s+1} as temporary storage.
- The contents of M's register r_i with $i \geq 1$ is stored in R's register r_{s+1+i}.

Simulating RAM instructions by a RASP

Question

How do we mimick *M*'s instructions by means of RASP instructions?

Example RAM instruction	**Corresponding RASP instructions**
LOAD 50	LOAD x (with $x = s + 1 + 50$)
ADD 20	ADD x (with $x = s + 1 + 50$)
JUMP while	JUMP $=x$ (with x address of register containing the instruction corresponding to the RAM instruction with label while)

Conclusion

For every RAM instruction that does not involve indirect addressing, we can just use the corresponding RASP instruction, with memory references appropriately re-mapped.

Simulating indirect addressing by a RASP

Question

What about instructions that involve **indirect addressing?**

Register	Content	Meaning
0	y	(accumulator)
⋮		
100	3	STORE $s+1$
101	$s+1$	
102	1	LOAD $s+21$
103	$s+21$	
104	5	ADD $=(s+1)$
105	s+1	
106	3	STORE 111
107	111	
108	1	LOAD $s+1$
109	$s+1$	
110	6	SUB ?
111	0	
⋮		
$s+1$		(temporary storage)
⋮		
$s+21$	x	(register 20 of RAM)
⋮		
$x+s+1$	z	(register x of RAM)

Example: SUB $*20$.

The simulation requires 6 instructions (12 registers)

- temporarily store the content of the accumulator in register r_{s+1}
- load the content of register $s+21$ into the accumulator (register $s+21$ of RASP corresponds to register 20 of the RAM). Let this content be x.
- add $s+1$ to the accumulator, effectively computing $x+s+1$. (register $x+s+1$ of RASP corresponds to register x of the RAM).
- store the resulting number into the address field of a SUB instruction (effectively modifying the program itself)
- restore the original accumulator from temporary register $s+1$
- use the SUB instruction to perform the subtraction

Similar reasoning for e.g. *ADD* $*\,32$, *DIV* $*\,23$, …, always using 6 RASP instructions.

Uniform cost model

Uniform cost model:

Time

- Each RAM instruction is simulated by at most 6 RASP instructions.
- So, $time(R, in) \leq 6 \times time(M, in)$, for every $in \in \mathbb{Z}^*$

Space

- Every register r_i used by the RAM causes the RASP to use register r_{s+1+i}.
- In addition, the RAM uses registers $r_1, \ldots, r_s$ to store the program, and r_{s+1} as working space.
- So, $space(R, in) \leq space(M, in) + s + 1$.

Logarithmic cost model: time

Logarithmic cost model:

Time: essential idea

- Each RAM instruction is simulated by at most 6 RASP instructions.
- For each RAM instruction with operand a, the corresponding RASP instructions are executed with operand a' where a' can be at most as large as $a + s + 1$. Then, since s is constant, $cost(a + s + 1) \leq cost(a) + c$, for some constant c.
- So, $time(R, in) = O(time(M, in))$, for every $in \in \mathbb{Z}^*$

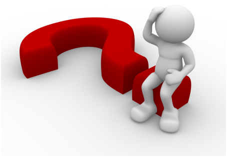
Logarithmic cost model: space

Logarithmic cost model:

Space: essential idea

- Every register r_i used by the RAM causes the RASP to use register r_{s+1+i}. At any time, the integer stored in the RASP r_{s+1+i} is the same as the integer stored in r_{s+1+i}. This occupies the same space.
- In addition, the RASP uses registers $r_1, \ldots, r_s$ to store the program. The program is independent of the input, and hence takes constant space.
- The RASP uses register r_0 and r_{s+1} as working space. At any point in time, the maximum integer stored here is bounded by the maximum integer stored anywhere by the RAM, possibly incremented by $s + 1$. This only increases the space by a constant (since s is constant).
- So, $space(R, in) = O(space(M, in))$

RAM vs RASP: other direction ?



What is the relationship between the class of partial functions definable by means of RAMs and the class of partial functions definable by means of RASPs?

- Are all functions definable by a RASP also definable by a RAM?
- Are functions that are definable by both "more efficient" on a RAM than on a RASP?

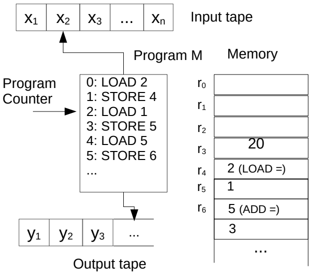
Theorem

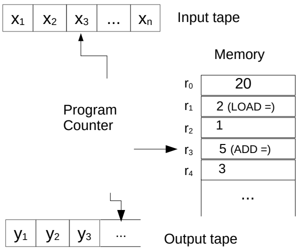
For every RASP R there exists a RAM M such that $[\![R]\!] = [\![M]\!]$. Moreover, there exist constants $k, l, c, d \in \mathbb{N}$ such that for all $in \in \mathbb{Z}^*$ on which R halts we have:

$$time(M, in) \leq k \times time(R, in) + c$$
$$space(M, in) \leq l \times space(R, in) + d$$

This holds under both the uniform and logarithmic cost model.

Simulating a RASP by means of a RAM (1)

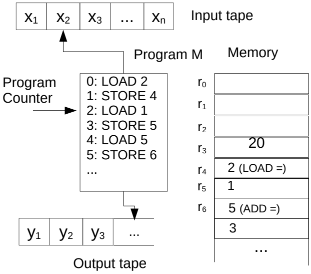


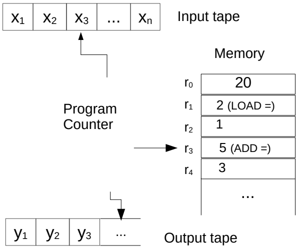


Given RASP R, we will construct RAM M so that it simulates R as follows:

- R's accumulator register r_0 will be simulated by means of M's register r_3.
- The contents of R's register r_i with $i \geq 1$ is stored in M's register r_{i+3}. Note that R may store an (encoding of an) instruction in register r_i and r_{i+1}; in that case M will also store that (encoding of the) instruction, but in r_{i+3} and r_{i+4}.
- We will use M's register r_1 as temporary storage for indirect addressing (which we will need to simulate R's instructions).
- We will use M's register r_2 to store the address of the instruction that R is currently executing.
- M's accumulator register r_0 is a working buffer.

Simulating a RASP by means of a RAM (2)





Key idea of ram *M*:

- First, write the (encoding of) program *R* to *M*'s memory, starting at register r_4. (Program *R* is fixed, so *M* can be programmed to write it to its memory.)
- Initialize r_2 (which simulates the program counter of *R*) to 4 (r_4 holds the first instruction of *R* in *M*).
- Now simulate *R*: inspect the contents of register $*r_2$.
  - If this is the encoding of LOAD=, then increment r_2, and execute *LOAD* $*2$
  - If this is the encoding of ADD=, then increment r_2, load the contents of r_3 (*R*'s accumulator), and execute *ADD* $*2$. Store the result in r_3.
  - If this is the encoding of . . . .
- Increment r_2 to go to the next instruction; then simulate that instruction.

Example: simulating RASP instruction SUB *i*

RAM instr		Meaning
LOAD	2	Increment the simulated program counter of *R* by 1, so
ADD	$= 1$	that it points to the register holding the operand *i* of the
STORE	2	SUB *i* instruction
LOAD	*2	Bring *i* to the accumulator, add 3, and store result in r_1
ADD	$= 3$	
STORE	1	
LOAD	3	Fetch the contents of RASP accumulator from r_3.
SUB	$*1$	Substract the contents of register r_{i+3}, and place result
STORE	3	back in r_3
LOAD	2	Increment the simulated program counter of *R* by 1 so
ADD	$= 1$	that it now points to the next RASP instruction
STORE	2	
JUMP	*a*	jump back to the beginning of the simulation loop (here assumed to be labeled *a*).

Uniform cost model

Uniform cost model:

Time

- Each RASP instruction is simulated by a constant number of RAM instructions.
- In addition, the RAM first executes a constant number of instructions to write the (encoding of) the RASP's program in the RAM's registers.
- So, for every $in \in \mathbb{Z}^*$, $time(M, in) \leq k \times time(R, in) + c$, for some constants $k, c \in \mathbb{N}$.

Space

- Every register r_i used by the RASP causes the RAM to use register r_{3+i}.
- In addition, the RAM uses registers r_1, r_2, r_3 for the simulation.
- So, $space(M, in) \leq space(R, in) + 3$.

Logarithmic cost model: time

Logarithmic cost model:

Time: essential idea

- The RAM first executes a constant number of instructions to write the (encoding of) the RASP's program in the RAM's registers. This takes constant time.
- Each RASP instruction is simulated by a constant number of RAM instructions.
- For each RASP instruction with operand a, the corresponding RAM instructions are executed with operands whose size is at most a constant factor larger than a.
- So, the cost of each RAM instruction is bounded by the cost of the corresponding RASP instruction + some constant.
- So, $time(M, in) = O(time(R, in))$, for every $in \in \mathbb{Z}^*$

Logarithmic cost model: space

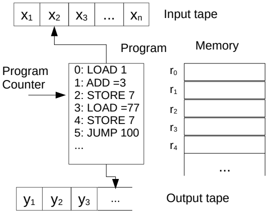
Logarithmic cost model:

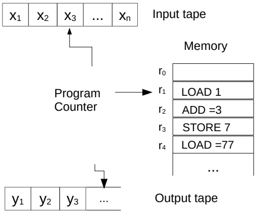
Space: essential idea

- Every register r_i used by the RASP causes the RAM to use register r_{3+i}. At any time, the integer stored in the RAM r_{3+i} is the same as the integer stored in RASP's r_i. This occupies the same space.
- In addition, the RAM uses registers $r_1, \ldots, r_3$ for the simulation. These registers store integers that are the values of the RASP's program counter/accumulator (plus some constant).
- How large can the RASP's program counter get? Either it points to an instruction of R (which is of constant size), or because of some JUMP instruction it points to some memory region that is dynamically "allocated" during execution. In that case, the address of the JUMP was also stored in the RASP memory. So, the space needed to store the RASP's simulated program counter is bounded by the space occupied by the RASP + some constant.
- So, $space(M, in) = O(space(R, in))$

Outline

Models of Computation





A **Computation Model** consists of:

- A description of the components of the machine being modeled
- A mathematical definition of the operations that the machine can execute during operation
- A mathematical definition of a configuration of the machine
- A mathematical definition of how, during execution, the machine can go from one configuration to the next
- (Usually): a cost model, that defines the time required to go from one configuration to the next, and the space that each configuration requires to store.

Central question: do these models have the same power (i.e., do they express the same partial functions from input to output). If they have the same power, are they equally efficient?

References

- These slides: published on the *Université Virtuelle*
  - Based on slides by S. Vansummeren
- *The Design and Analysis of Computer Algorithms*, A. V. Aho, J. E. Hopcroft and J. D. Ullman (Addison-Wesley, 1974)
  - Section 1.1 Algorithms and their complexity
  - Section 1.2 Random access machines
  - Section 1.3 Computational complexity of RAM programs
  - Section 1.4 A stored program model